2019-2021美团技术年货

CODE A BETTER LIFE

【后端篇】



美团 美团

目录

后端 | 2021 1

如何优雅地记录操作日志? 1

基于 SSD 的 Kafka 应用层缓存架构设计与实现 29

美团图数据库平台建设及业务实践 41

OCTO 2.0：美团基于 Service Mesh 的服务治理系统详解 60

GraphQL 及元数据驱动架构在后端 BFF 中的实践 71

速度与压缩比如何兼得？压缩算法在构建部署中的优化 103

百亿规模 API 网关服务 Shepherd 的设计与实现 121

TensorFlow 在推荐系统中的分布式训练优化实践 142

美团 Serverless 平台 Nest 的探索与实践 165

算法平台在线服务体系的演进与实践 185

美团终端消息投递服务 Pike 的演进之路 207

新一代 CTR 预测服务的 GPU 优化实践 227

美团弹性伸缩系统的技术演进与落地实践 241

后端 | 2020 263

Java 线程池实现原理及其在美团业务中的实践 263

美团万亿级 KV 存储架构与实践 294

Java 中 9 种常见的 CMS GC 问题分析与解决 315

美团配送 A/B 评估体系建设实践 382

新一代垃圾回收器 ZGC 的探索与实践 396

设计模式在外卖营销业务中的实践 416

美团命名服务的挑战与演进 438
美团 MySQL 数据库巡检系统的设计与应用 459
Kubernetes 如何改变美团的云基础设施? 468
基本功 | Java 即时编译器原理解析及实践 486
MyBatis 版本升级引发的线上告警回顾及原理分析 519
复杂环境下落地 Service Mesh 的挑战与实践 535
C++ 服务编译耗时优化原理及实践 549
速度与压缩比如何兼得? 压缩算法在构建部署中的优化 572
美团 OCTO 万亿级数据中心计算引擎技术解析 590
Intel PAUSE 指令变化影响到 MySQL 的性能，该如何解决? 599
美团内部讲座 | 周煊：华东师范大学的数据库系统研究 619

后端 | 2019 651
Java 魔法类：Unsafe 应用解析 652
Java 动态追踪技术探究 669
字节码增强技术探索 681
JVM CPU Profiler 技术原理及源码深度解析 708
Java 动态调试技术原理及实践 731
从 ReentrantLock 的实现看 AQS 的原理及应用 760
美团点评 Kubernetes 集群管理实践 796
美团集群调度系统 HULK 技术演进 811
保障 IDC 安全：分布式 HIDS 集群架构设计 824
Leaf：美团分布式 ID 生成服务开源 847
美团大规模微服务通信框架及治理体系 OCTO 核心组件开源 854
美团下一代服务治理系统 OCTO2.0 的探索与实践 860
XGBoost 缺失值引发的问题及其深度分析 876
Spring Boot 引起的“堆外内存泄漏”排查及经验总结 885

美团点评效果广告实验配置平台的设计与实现 897
根因分析初探：一种报警聚类算法在业务系统的落地实施 908
全链路压测自动化实践 926
降低软件复杂性一般原则和方法 941

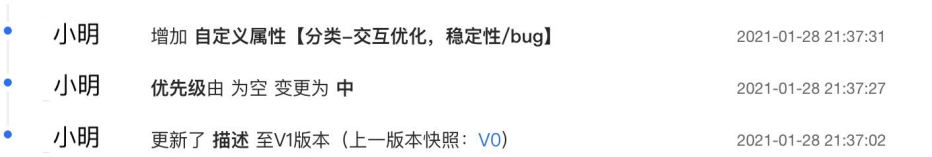
后端

如何优雅地记录操作日志?

作者：站通

操作日志几乎存在于每个系统中，而这些系统都有记录操作日志的一套 API。操作日志和系统日志不一样，操作日志必须要做到简单易懂。所以如何让操作日志不和业务逻辑耦合，如何让操作日志的内容易于理解，让操作日志的接入更加简单？上面这些都是本文要回答的问题，主要围绕着如何“优雅”地记录操作日志展开描述。

1. 操作日志的使用场景



例子

系统日志和操作日志的区别

系统日志：系统日志主要是为开发排查问题提供依据，一般打印在日志文件中；系统日志的可读性要求没那么高，日志中会包含代码的信息，比如在某个类的某一行打印了一个日志。

操作日志：主要是对某个对象进行新增操作或者修改操作后记录下这个新增或者修改，操作日志要求可读性比较强，因为它主要是给用户看的，比如订单的物流信息，用户需要知道在什么时间发生了什么事情。再比如，客服对工单的处理记录信息。

操作日志的记录格式大概分为下面几种：* 单纯的文字记录，比如：2021-09-16 10:00 订单创建。 * 简单的动态的文本记录，比如：2021-09-16 10:00 订单创建，订单号：NO.11089999，其中涉及变量订单号“NO.11089999”。 * 修改类型的文本，包含修改前和修改后的值，比如：2021-09-16 10:00 用户小明修改了订单的配送地址：从“金灿灿小区”修改到“银盏盏小区”，其中涉及变量配送的原地址“金灿灿小区”和新地址“银盏盏小区”。 * 修改表单，一次会修改多个字段。

2. 实现方式

2.1 使用 Canal 监听数据库记录操作日志

Canal 是一款基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费的开源组件，通过采用监听数据库 Binlog 的方式，这样可以从底层知道是哪些数据做了修改，然后根据更改的数据记录操作日志。

这种方式的优点是和业务逻辑完全分离。缺点也很明显，局限性太高，只能针对数据库的更改做操作日志记录，如果修改涉及到其他团队的 RPC 的调用，就没办法监听数据库了，举个例子：给用户发送通知，通知服务一般都是公司内部的公共组件，这时候只能在调用 RPC 的时候手工记录发送通知的操作日志了。

2.2 通过日志文件的方式记录

```
log.info("订单创建")
log.info("订单已经创建，订单编号：{}", orderNo)
log.info("修改了订单的配送地址：从"{}"修改到"{}"，"金灿灿小区"，"银盏盏小区")
```

这种方式的操作记录需要解决三个问题。

问题一：操作人如何记录

借助 SLF4J 中的 MDC 工具类，把操作人放在日志中，然后在日志中统一打印出来。首先在用户的拦截器中把用户的标识 Put 到 MDC 中。

```
@Component
public class UserInterceptor extends HandlerInterceptorAdapter {
  @Override
  public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
    // 获取到用户标识
    String userNo = getUserNo(request);
    // 把用户 ID 放到 MDC 上下文中
    MDC.put("userId", userNo);
    return super.preHandle(request, response, handler);
  }

  private String getUserNo(HttpServletRequest request) {
    // 通过 SSO 或者 Cookie 或者 Auth 信息获取到 当前登陆的用户信息
    return null;
  }
}
```

其次，把 userId 格式化到日志中，使用 %X{userId} 可以取到 MDC 中用户标识。

```
<pattern>"%d{yyyy-MM-dd HH:mm:ss.SSS} %t %-5level %X{userId}
%logger{30}.%method:%L - %msg%n"</pattern>
```

问题二：操作日志如何和系统日志区分开

通过配置 Log 的配置文件，把有关操作日志的 Log 单独放到一日志文件中。

```
// 不同业务日志记录到不同的文件
<appender name="businessLogAppender" class="ch.qos.logback.core.
rolling.RollingFileAppender">
    <File>logs/business.log</File>
    <append>true</append>
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>INFO</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
    <rollingPolicy class="ch.qos.logback.core.rolling.
TimeBasedRollingPolicy">
        <fileNamePattern>logs/ 业务 A.%d.%i.log</fileNamePattern>
        <maxHistory>90</maxHistory>
        <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.
logback.core.rolling.SizeAndTimeBasedFNATP">
            <maxFileSize>10MB</maxFileSize>
        </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>
```

```
    <encoder>
        <pattern>"%d{yyyy-MM-dd HH:mm:ss.SSS} %t %-5level %X{userId}
%logger{30}.%method:%L - %msg%n"</pattern>
        <charset>UTF-8</charset>
    </encoder>
</appender>

<logger name="businessLog" additivity="false" level="INFO">
    <appender-ref ref="businessLogAppender"/>
</logger>
```

然后在 Java 代码中单独的记录业务日志。

```
// 记录特定日志的声明
private final Logger businessLog = LoggerFactory.
getLogger("businessLog");

// 日志存储
businessLog.info(" 修改了配送地址 ");
```

问题三：如何生成可读懂的日志文案

可以采用 LogUtil 的方式，也可以采用切面的方式生成日志模板，后续内容将会进行介绍。这样就可以把日志单独保存在一个文件中，然后通过日志收集可以把日志保存在 Elasticsearch 或者数据库中，接下来看下如何生成可读的操作日志。

2.3 通过 LogUtil 的方式记录日志

```
  LogUtil.log(orderNo, " 订单创建 ", " 小明 ") 模板
  LogUtil.log(orderNo, " 订单创建，订单号 "+"NO.11089999",  " 小明 ")
  String template = " 用户 %s 修改了订单的配送地址：从 "%s" 修改到 "%s""
  LogUtil.log(orderNo, String.format(tempalte, " 小明 ", " 金灿灿小区 ", " 银
盏盏小区 "),  " 小明 ")
```

> 这里解释下为什么记录操作日志的时候都绑定了一个 OrderNo，因为操作日志记录的是：某一个“时间”“谁”对“什么”做了什么“事情”。当查询业务的操作日志的时候，会查询针对这个订单的的所有操作，所以代码中加上了 OrderNo，记录操作日志的时候需要记录下操作人，所以传了操作人“小明”进来。

上面看起来问题并不大，在修改地址的业务逻辑方法中使用一行代码记录了操作日志，接下来再看一个更复杂的例子：

```
private OnesIssueDO updateAddress(updateDeliveryRequest request) {
    DeliveryOrder deliveryOrder = deliveryQueryService.
queryOldAddress(request.getDeliveryOrderNo());
    // 更新派送信息，电话，收件人，地址
    doUpdate(request);
    String logContent = getLogContent(request, deliveryOrder);
    LogUtils.logRecord(request.getOrderNo(), logContent, request.
getOperator);
    return onesIssueDO;
}

private String getLogContent(updateDeliveryRequest request,
DeliveryOrder deliveryOrder) {
    String template = "用户 %s 修改了订单的配送地址：从 "%s" 修改到 "%s"";
    return String.format(tempalte, request.getUserName(), deliveryOrder.
getAddress(), request.getAddress);
}
```

可以看到上面的例子使用了两个方法代码，外加一个 getLogContent 的函数实现了操作日志的记录。当业务变得复杂后，记录操作日志放在业务代码中会导致业务的逻辑比较繁杂，最后导致 LogUtils.logRecord() 方法的调用存在于很多业务的代码中，而且类似 getLogContent() 这样的方法也散落在各个业务类中，对于代码的可读性和可维护性来说是一个灾难。下面介绍下如何避免这个灾难。

2.4 方法注解实现操作日志

为了解决上面问题，一般采用 AOP 的方式记录日志，让操作日志和业务逻辑解耦，接下来看一个简单的 AOP 日志的例子。

```
@LogRecord(content="修改了配送地址")
public void modifyAddress(updateDeliveryRequest request){
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

我们可以在注解的操作日志上记录固定文案，这样业务逻辑和业务代码可以做到解耦，让我们的业务代码变得纯净起来。可能有同学注意到，上面的方式虽然解耦了操

作日志的代码，但是记录的文案并不符合我们的预期，文案是静态的，没有包含动态的文案，因为我们需要记录的操作日志是：用户 %s 修改了订单的配送地址，从“%s”修改到“%s”。接下来，我们介绍一下如何优雅地使用 AOP 生成动态的操作日志。

3. 优雅地支持 AOP 生成动态的操作日志

3.1 动态模板

一提到动态模板，就会涉及到让变量通过占位符的方式解析模板，从而达到通过注解记录操作日志的目的。模板解析的方式有很多种，这里使用了 SpEL（Spring Expression Language，Spring 表达式语言）来实现。我们可以先写下期望的记录日志的方式，然后再看下能否实现这样的功能。

```
@LogRecord(content = "修改了订单的配送地址：从"#oldAddress", 修改到
"#request.address"")
public void modifyAddress(updateDeliveryRequest request, String oldAddress){
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

通过 SpEL 表达式引用方法上的参数，可以让变量填充到模板中达到动态的操作日志文本内容。 但是现在还有几个问题需要解决：* 操作日志需要知道是哪个操作人修改的订单配送地址。 * 修改订单配送地址的操作日志需要绑定在配送的订单上，从而可以根据配送订单号查询出对这个配送订单的所有操作。 * 为了在注解上记录之前的配送地址是什么，在方法签名上添加了一个和业务无关的 oldAddress 的变量，这样就不优雅了。

为了解决前两个问题，我们需要把期望的操作日志使用形式改成下面的方式：

```
@LogRecord(
     content = "修改了订单的配送地址：从"#oldAddress", 修改到“#request.address"",
     operator = "#request.userName", bizNo="#request.deliveryOrderNo")
public void modifyAddress(updateDeliveryRequest request, String oldAddress){
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

修改后的代码在注解上添加两个参数，一个是操作人，一个是操作日志需要绑定的对象。但是，在普通的 Web 应用中用户信息都是保存在一个线程上下文的静态方法中，所以 operator 一般是这样的写法（假定获取当前登陆用户的方式是 UserContext.getCurrentUser()）。

```
operator = "#{T(com.meituan.user.UserContext).getCurrentUser()}"
```

这样的话，每个 @LogRecord 的注解上的操作人都是这么长一串。为了避免过多的重复代码，我们可以把注解上的 operator 参数设置为非必填，这样用户可以填写操作人。但是，如果用户不填写我们就取 UserContext 的 user（下文会介绍如何取 user ）。最后，最简单的日志变成了下面的形式：

```
@LogRecord(content = "修改了订单的配送地址：从"#oldAddress", 修改到"#request.address"",
           bizNo="#request.deliveryOrderNo")
public void modifyAddress(updateDeliveryRequest request, String oldAddress){
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

接下来，我们需要解决第三个问题：为了记录业务操作记录添加了一个 oldAddress 变量，不管怎么样这都不是一个好的实现方式，所以接下来，我们需要把 oldAddress 变量从修改地址的方法签名上去掉。但是操作日志确实需要 oldAddress 变量，怎么办呢?

要么和产品经理 PK 一下，让产品经理把文案从“修改了订单的配送地址：从 xx 修改到 yy”改为“修改了订单的配送地址为：yy”。但是从用户体验上来看，第一种文案更人性化一些，显然我们不会 PK 成功的。那么我们就必须要把这个 oldAddress 查询出来然后供操作日志使用了。还有一种解决办法是：把这个参数放到操作日志的线程上下文中，供注解上的模板使用。我们按照这个思路再改下操作日志的实现代码。

```
@LogRecord(content = "修改了订单的配送地址：从"#oldAddress", 修改到
"#request.address"",
        bizNo="#request.deliveryOrderNo")
```

```
public void modifyAddress(updateDeliveryRequest request){
    // 查询出原来的地址是什么
    LogRecordContext.putVariable("oldAddress", DeliveryService.
queryOldAddress(request.getDeliveryOrderNo()));
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

这时候可以看到，LogRecordContext 解决了操作日志模板上使用方法参数以外变量的问题，同时避免了为了记录操作日志修改方法签名的设计。虽然已经比之前的代码好了些，但是依然需要在业务代码里面加了一行业务逻辑无关的代码，如果有“强迫症”的同学还可以继续往下看，接下来我们会讲解自定义函数的解决方案。下面再看另一个例子：

```
@LogRecord(content = "修改了订单的配送员：从"#oldDeliveryUserId", 修改到
"#request.userId"",
        bizNo="#request.deliveryOrderNo")
public void modifyAddress(updateDeliveryRequest request){
    // 查询出原来的地址是什么
    LogRecordContext.putVariable("oldDeliveryUserId", DeliveryService.
queryOldDeliveryUserId(request.getDeliveryOrderNo()));
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

这个操作日志的模板最后记录的内容是这样的格式：修改了订单的配送员：从“10090”，修改到“10099”，显然用户看到这样的操作日志是不明白的。用户对于用户 ID 是 10090 还是 10099 并不了解，用户期望看到的是：修改了订单的配送员：从“张三（18910008888）”，修改到“小明（13910006666）”。用户关心的是配送员的姓名和电话。但是我们方法中传递的参数只有配送员的 ID，没有配送员的姓名可电话。我们可以通过上面的方法，把用户的姓名和电话查询出来，然后通过 LogRecordContext 实现。

但是，“强迫症”是不期望操作日志的代码嵌入在业务逻辑中的。接下来，我们考虑另一种实现方式：自定义函数。如果我们可以通过自定义函数把用户 ID 转换为用户姓名和电话，那么就能解决这一问题，按照这个思路，我们把模板修改为下面的形式：

```
@LogRecord(content = "修改了订单的配送员：从{deliveryUser{#oldDeliveryUserId}}",
修改到"{deveryUser{#request.userId}}"",
        bizNo="#request.deliveryOrderNo")
public void modifyAddress(updateDeliveryRequest request){
    // 查询出原来的地址是什么
    LogRecordContext.putVariable("oldDeliveryUserId", DeliveryService.
queryOldDeliveryUserId(request.getDeliveryOrderNo()));
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

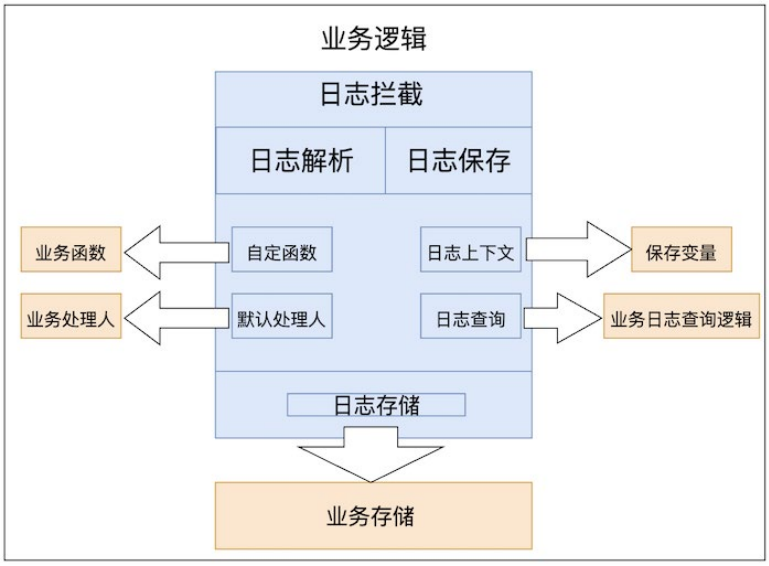
其中 deliveryUser 是自定义函数，使用大括号把 Spring 的 SpEL 表达式包裹起来，这样做的好处：一是把 SpEL（Spring Expression Language，Spring 表达式语言）和自定义函数区分开便于解析；二是如果模板中不需要 SpEL 表达式解析可以容易的识别出来，减少 SpEL 的解析提高性能。这时候我们发现上面代码还可以优化成下面的形式：

```
@LogRecord(content = "修改了订单的配送员：从"{queryOldUser{#request.
deliveryOrderNo()}}", 修改到"{deveryUser{#request.userId}}"",
        bizNo="#request.deliveryOrderNo")
public void modifyAddress(updateDeliveryRequest request){
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

这样就不需要在 modifyAddress 方法中通过 LogRecordContext.putVariable() 设置老的快递员了，通过直接新加一个自定义函数 queryOldUser() 参数把派送订单传递进去，就能查到之前的配送人了，只需要让方法的解析在 modifyAddress() 方法执行之前运行。这样的话，我们让业务代码又变得纯净了起来，同时也让“强迫症”不再感到难受了。

4. 代码实现解析

4.1 代码结构



上面的操作日志主要是通过一个 AOP 拦截器实现的，整体主要分为 AOP 模块、日志解析模块、日志保存模块、Starter 模块；组件提供了 4 个扩展点，分别是：自定义函数、默认处理人、业务保存和查询；业务可以根据自己的业务特性定制符合自己业务的逻辑。

4.2 模块介绍

有了上面的分析，已经得出一种我们期望的操作日志记录的方式，那么接下来看看如何实现上面的逻辑。实现主要分为下面几个步骤：* AOP 拦截逻辑 * 解析逻辑 * 模板解析 * LogContext 逻辑 * 默认的 operator 逻辑 * 自定义函数逻辑 * 默认的日志持久化逻辑 * Starter 封装逻辑

4.2.1 AOP 拦截逻辑

这块逻辑主要是一个拦截器，针对 @LogRecord 注解分析出需要记录的操作日志，然后把操作日志持久化，这里把注解命名为 @LogRecordAnnotation。接下来，我

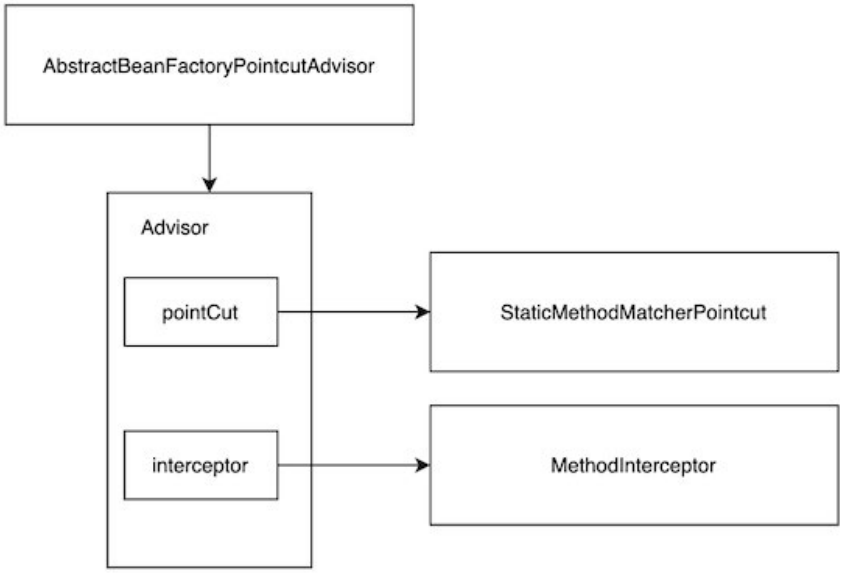
们看下注解的定义：

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface LogRecordAnnotation {
    String success();

    String fail() default "";

    String operator() default "";

    String bizNo();

    String category() default "";

    String detail() default "";

    String condition() default "";
}
```

注解中除了上面提到参数外，还增加了 fail、category、detail、condition 等参数，这几个参数是为了满足特定的场景，后面还会给出具体的例子。

参数名	描述	是否必填
success	操作日志的文本模板	是
fail	操作日志失败的文本版本	否
operator	操作日志的执行人	否
bizNo	操作日志绑定的业务对象标识	是
category	操作日志的种类	否
detail	扩展参数，记录操作日志的修改详情	否
condition	记录日志的条件	否

为了保持简单，组件的必填参数就两个。业务中的 AOP 逻辑大部分是使用 @Aspect 注解实现的，但是基于注解的 AOP 在 Spring boot 1.5 中兼容性是有问题的，组件为了兼容 Spring boot1.5 的版本我们手工实现 Spring 的 AOP 逻辑。



切面选择 `AbstractBeanFactoryPointcutAdvisor` 实现，切点是通过 `StaticMethodMatcherPointcut` 匹配包含 `LogRecordAnnotation` 注解的方法。通过实现 `MethodInterceptor` 接口实现操作日志的增强逻辑。

下面是拦截器的切点逻辑：

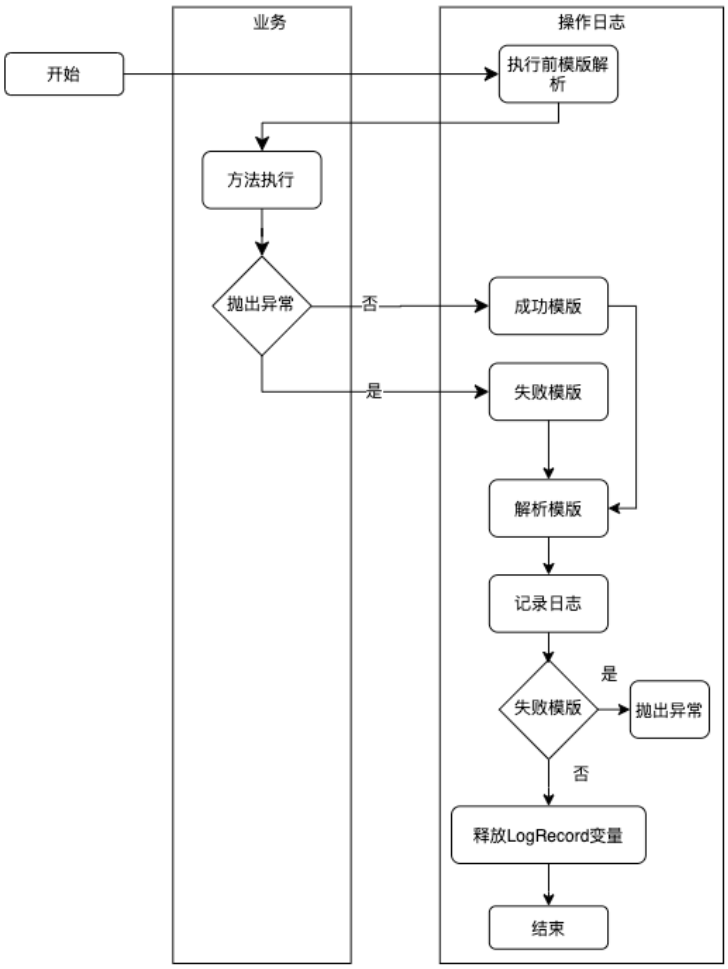
```
public class LogRecordPointcut extends StaticMethodMatcherPointcut
implements Serializable {
    // LogRecord 的解析类
    private LogRecordOperationSource logRecordOperationSource;

    @Override
    public boolean matches(@NonNull Method method, @NonNull Class<?>
targetClass) {
        // 解析 这个 method 上有没有 @LogRecordAnnotation 注解，有的话会
解析出来注解上的各个参数
        return !CollectionUtils.isEmpty(logRecordOperationSource.
computeLogRecordOperations(method, targetClass));
    }

    void setLogRecordOperationSource(LogRecordOperationSource
logRecordOperationSource) {
        this.logRecordOperationSource = logRecordOperationSource;
    }
}
```

切面的增强逻辑主要代码如下：

```
@Override
public Object invoke(MethodInvocation invocation) throws Throwable {
    Method method = invocation.getMethod();
    // 记录日志
    return execute(invocation, invocation.getThis(), method, invocation.
getArguments());
}

private Object execute(MethodInvocation invoker, Object target, Method
method, Object[] args) throws Throwable {
    Class<?> targetClass = getTargetClass(target);
    Object ret = null;
    MethodExecuteResult methodExecuteResult = new MethodExecuteResult(true,
null, "");
    LogRecordContext.putEmptySpan();
    Collection<LogRecordOps> operations = new ArrayList<>();
    Map<String, String> functionNameAndReturnMap = new HashMap<>();
    try {
        operations = logRecordOperationSource.
computeLogRecordOperations(method, targetClass);
        List<String> spElTemplates =
getBeforeExecuteFunctionTemplate(operations);
        // 业务逻辑执行前的自定义函数解析
        functionNameAndReturnMap =
processBeforeExecuteFunctionTemplate(spElTemplates, targetClass, method,
args);
    } catch (Exception e) {
        log.error("log record parse before function exception", e);
    }
    try {
        ret = invoker.proceed();
    } catch (Exception e) {
        methodExecuteResult = new MethodExecuteResult(false, e,
e.getMessage());
    }
    try {
        if (!CollectionUtils.isEmpty(operations)) {
            recordExecute(ret, method, args, operations, targetClass,
                    methodExecuteResult.isSuccess(), methodExecuteResult.
getErrorMsg(), functionNameAndReturnMap);
        }
    } catch (Exception t) {
        // 记录日志错误不要影响业务
        log.error("log record parse exception", t);
    } finally {
        LogRecordContext.clear();
    }
```

```
    if (methodExecuteResult.throwable != null) {
        throw methodExecuteResult.throwable;
    }
    return ret;
}
```

拦截逻辑的流程：



可以看到，操作日志的记录持久化是在方法执行完之后执行的，当方法抛出异常之后会先捕获异常，等操作日志持久化完成后再抛出异常。在业务的方法执行之前，会对提前解析的自定义函数求值，解决了前面提到的需要查询修改之前的内容。

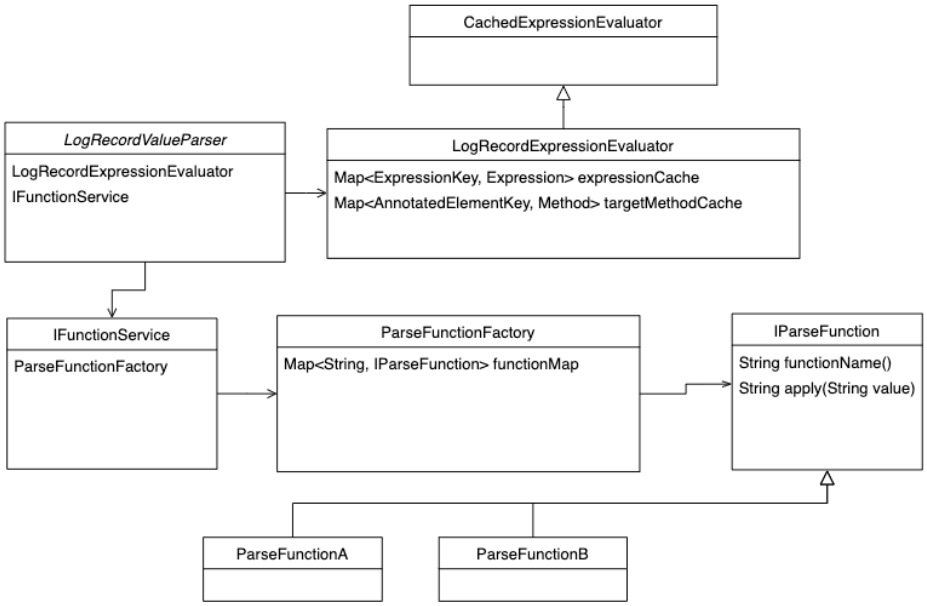
4.2.2 解析逻辑

模板解析

Spring 3 提供了一个非常强大的功能：Spring EL，SpEL 在 Spring 产品中是作为表达式求值的核心基础模块，它本身是可以脱离 Spring 独立使用的。举个例子：

```
public static void main(String[] args) {
        SpelExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression("#root.
purchaseName");
        Order order = new Order();
        order.setPurchaseName("张三");
        System.out.println(expression.getValue(order));
}
```

这个方法将打印“张三”。LogRecord 解析的类图如下：



解析核心类：`LogRecordValueParser` 里面封装了自定义函数和 SpEL 解析类 `LogRecordExpressionEvaluator`。

```
public class LogRecordExpressionEvaluator extends
CachedExpressionEvaluator {

    private Map<ExpressionKey, Expression> expressionCache = new
ConcurrentHashMap<>(64);

    private final Map<AnnotatedElementKey, Method> targetMethodCache =
new ConcurrentHashMap<>(64);

    public String parseExpression(String conditionExpression,
AnnotatedElementKey methodKey, EvaluationContext evalContext) {
        return getExpression(this.expressionCache, methodKey,
conditionExpression).getValue(evalContext, String.class);
    }
}
```

`LogRecordExpressionEvaluator` 继承自 `CachedExpressionEvaluator` 类，这个类里面有两个 Map，一个是 expressionCache 一个是 targetMethodCache。在上面的例子中可以看到，SpEL 会解析成一个 Expression 表达式，然后根据传入的 Object 获取到对应的值，所以 expressionCache 是为了缓存方法、表达式和 SpEL 的 Expression 的对应关系，让方法注解上添加的 SpEL 表达式只解析一次。下面的 targetMethodCache 是为了缓存传入到 Expression 表达式的 Object。核心的解析逻辑是上面最后一行代码。

```
getExpression(this.expressionCache, methodKey, conditionExpression).
getValue(evalContext, String.class);
```

`getExpression` 方法会从 expressionCache 中获取到 @LogRecordAnnotation 注解上的表达式的解析 Expression 的实例，然后调用 `getValue` 方法，`getValue` 传入一个 evalContext 就是类似上面例子中的 order 对象。其中 Context 的实现将会在下文介绍。

日志上下文实现

下面的例子把变量放到了 LogRecordContext 中，然后 SpEL 表达式就可以顺利的解析方法上不存在的参数了，通过上面的 SpEL 的例子可以看出，要把方法的参数和 LogRecordContext 中的变量都放到 SpEL 的 `getValue` 方法的 Object 中才可以

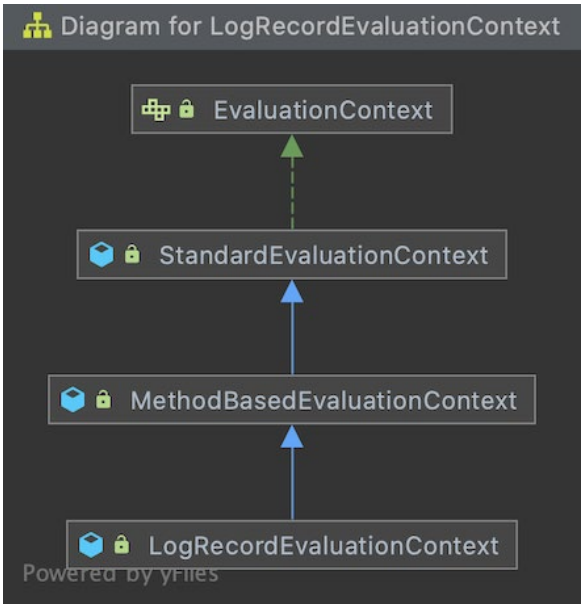
顺利的解析表达式的值。下面看下如何实现：

```
@LogRecord(content = "修改了订单的配送员：从"{deveryUser{#oldDeliveryUserId}}",
修改到"{deveryUser{#request.getUserId()}}"",
            bizNo="#request.getDeliveryOrderNo()")
public void modifyAddress(updateDeliveryRequest request){
    // 查询出原来的地址是什么
    LogRecordContext.putVariable("oldDeliveryUserId", DeliveryService.
queryOldDeliveryUserId(request.getDeliveryOrderNo()));
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

在 LogRecordValueParser 中创建了一个 EvaluationContext，用来给 SpEL 解析方法参数和 Context 中的变量。相关代码如下：

```
EvaluationContext evaluationContext = expressionEvaluator.
createEvaluationContext(method, args, targetClass, ret, errorMsg,
beanFactory);
```

在解析的时候调用 `getValue` 方法传入的参数 evalContext，就是上面这个 EvaluationContext 对象。下面是 LogRecordEvaluationContext 对象的继承体系：



LogRecordEvaluationContext 做了三个事情：* 把方法的参数都放到 SpEL 解析的 RootObject 中。 * 把 LogRecordContext 中的变量都放到 RootObject 中。 * 把方法的返回值和 ErrorMsg 都放到 RootObject 中。

LogRecordEvaluationContext 的代码如下：

```
public class LogRecordEvaluationContext extends
MethodBasedEvaluationContext {

    public LogRecordEvaluationContext(Object rootObject, Method
method, Object[] arguments,
                                      ParameterNameDiscoverer
parameterNameDiscoverer, Object ret, String errorMsg) {
        // 把方法的参数都放到 SpEL 解析的 RootObject 中
        super(rootObject, method, arguments, parameterNameDiscoverer);
        // 把 LogRecordContext 中的变量都放到 RootObject 中
         Map<String, Object> variables = LogRecordContext.getVariables();
         if (variables != null && variables.size() > 0) {
             for (Map.Entry<String, Object> entry : variables.entrySet())
{
                 setVariable(entry.getKey(), entry.getValue());
             }
         }
         // 把方法的返回值和 ErrorMsg 都放到 RootObject 中
         setVariable("_ret", ret);
         setVariable("_errorMsg", errorMsg);
    }
}
```

下面是 LogRecordContext 的实现，这个类里面通过一个 ThreadLocal 变量保持了一个栈，栈里面是个 Map，Map 对应了变量的名称和变量的值。

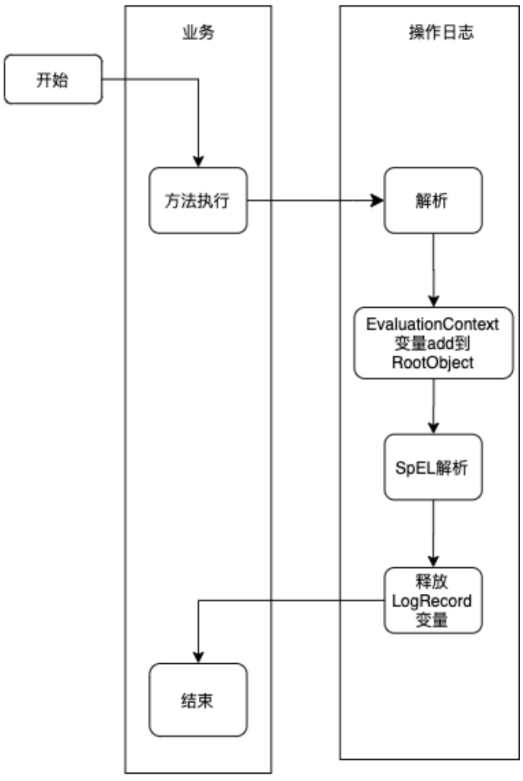
```
public class LogRecordContext {

    private static final InheritableThreadLocal<Stack<Map<String,
Object>>> variableMapStack = new InheritableThreadLocal<>();
   // 其他省略 ....
}
```

上面使用了 InheritableThreadLocal，所以在线程池的场景下使用 LogRecord-Context 会出现问题，如果支持线程池可以使用阿里巴巴开源的 TTL 框架。那这里为什么不直接设置一个 ThreadLocal<map> 对象，而是要设置一个 Stack 结构呢?

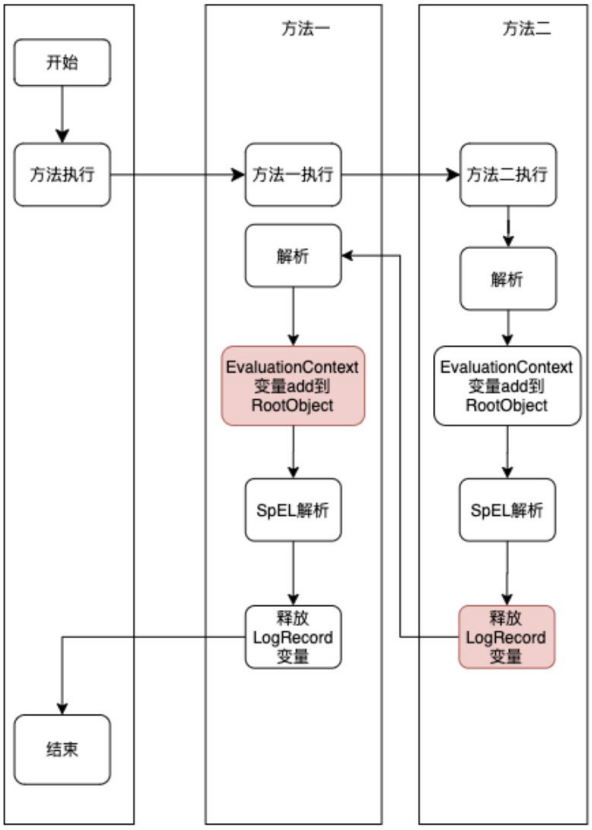
我们看一下这么做的原因是什么。</map

```
@LogRecord(content = "修改了订单的配送员:从"{deveryUser{#oldDeliveryUserId}}",
修改到"{deveryUser{#request.getUserId()}}"",
        bizNo="#request.getDeliveryOrderNo()")
public void modifyAddress(updateDeliveryRequest request){
    // 查询出原来的地址是什么
    LogRecordContext.putVariable("oldDeliveryUserId", DeliveryService.
queryOldDeliveryUserId(request.getDeliveryOrderNo()));
    // 更新派送信息 电话，收件人、地址
    doUpdate(request);
}
```

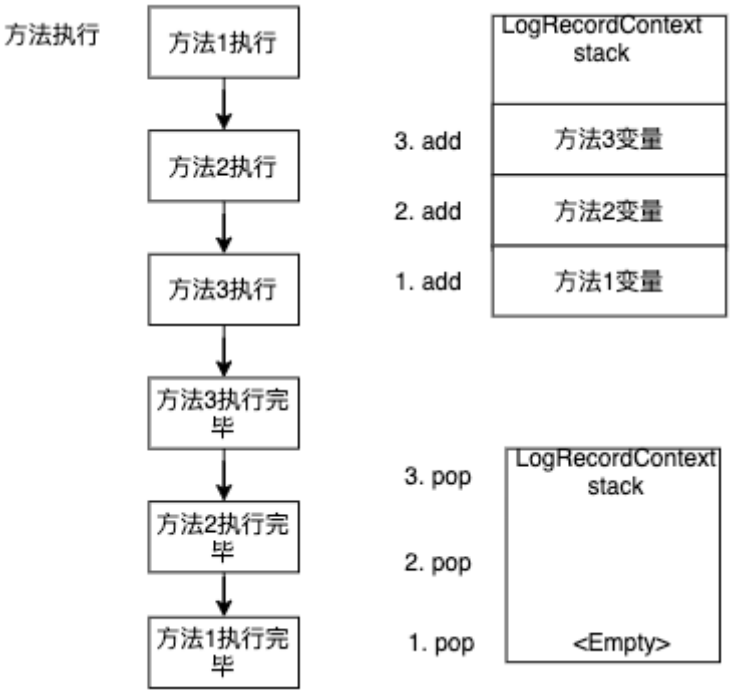
上面代码的执行流程如下：



看起来没有什么问题，但是使用 LogRecordAnnotation 的方法里面嵌套了另一个使用 LogRecordAnnotation 方法的时候，流程就变成下面的形式：



可以看到，当方法二执行了释放变量后，继续执行方法一的 logRecord 逻辑，此时解析的时候 ThreadLocal<map> 的 Map 已经被释放掉，所以方法一就获取不到对应的变量了。方法一和方法二共用一个变量 Map 还有个问题是：如果方法二设置了和方法一相同的变量两个方法的变量就会被相互覆盖。所以最终 LogRecordContext 的变量的生命周期需要是下面的形式：</map



LogRecordContext 每执行一个方法都会压栈一个 Map，方法执行完之后会 Pop 掉这个 Map，从而避免变量共享和覆盖问题。

默认操作人逻辑

在 LogRecordInterceptor 中 IOperatorGetService 接口，这个接口可以获取到当前的用户。下面是接口的定义：

```
public interface IOperatorGetService {

    /**
     * 可以在里面外部的获取当前登陆的用户，比如 UserContext.getCurrentUser()
     *
     * @return 转换成 Operator 返回
     */
    Operator getUser();
}
```

下面给出了从用户上下文中获取用户的例子：

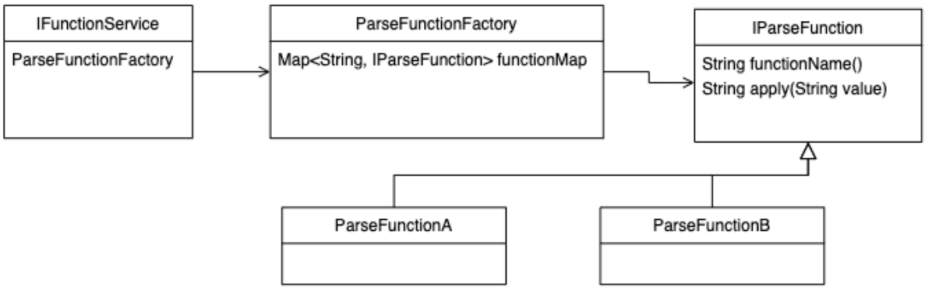
```
public class DefaultOperatorGetServiceImpl implements IOperatorGetService
{

    @Override
    public Operator getUser() {
    //UserUtils 是获取用户上下文的方法
         return Optional.ofNullable(UserUtils.getUser())
                        .map(a -> new Operator(a.getName(), a.getLogin()))
                        .orElseThrow(()->new IllegalArgumentException("user
is null"));

    }
}
```

组件在解析 operator 的时候，就判断注解上的 operator 是否是空，如果注解上没有指定，我们就从 IOperatorGetService 的 getUser 方法获取了。如果都获取不到，就会报错。

```
String realOperatorId = "";
if (StringUtils.isEmpty(operatorId)) {
    if (operatorGetService.getUser() == null || StringUtils.
isEmpty(operatorGetService.getUser().getOperatorId())) {
        throw new IllegalArgumentException("user is null");
    }
    realOperatorId = operatorGetService.getUser().getOperatorId();
} else {
    spElTemplates = Lists.newArrayList(bizKey, bizNo, action, operatorId,
detail);
}
```

自定义函数逻辑

自定义函数的类图如下：



下面是 IParseFunction 的接口定义：`executeBefore` 函数代表了自定义函数是否在业务代码执行之前解析，上面提到的查询修改之前的内容。

```
public interface IParseFunction {

  default boolean executeBefore(){
    return false;
  }

  String functionName();

  String apply(String value);
}
```

ParseFunctionFactory 的代码比较简单，它的功能是把所有的 IParseFunction 注入到函数工厂中。

```
public class ParseFunctionFactory {
  private Map<String, IParseFunction> allFunctionMap;

  public ParseFunctionFactory(List<IParseFunction> parseFunctions) {
    if (CollectionUtils.isEmpty(parseFunctions)) {
      return;
    }
    allFunctionMap = new HashMap<>();
    for (IParseFunction parseFunction : parseFunctions) {
      if (StringUtils.isEmpty(parseFunction.functionName())) {
        continue;
      }
      allFunctionMap.put(parseFunction.functionName(), parseFunction);
    }
  }

  public IParseFunction getFunction(String functionName) {
    return allFunctionMap.get(functionName);
  }

  public boolean isBeforeFunction(String functionName) {
    return allFunctionMap.get(functionName) != null && allFunctionMap.
get(functionName).executeBefore();
  }
}
```

DefaultFunctionServiceImpl 的逻辑就是根据传入的函数名称 functionName 找到

对应的 IParseFunction，然后把参数传入到 IParseFunction 的 apply 方法上最后返回函数的值。

```
public class DefaultFunctionServiceImpl implements IFunctionService {

  private final ParseFunctionFactory parseFunctionFactory;

  public DefaultFunctionServiceImpl(ParseFunctionFactory parseFunctionFactory) {
    this.parseFunctionFactory = parseFunctionFactory;
  }

  @Override
  public String apply(String functionName, String value) {
    IParseFunction function = parseFunctionFactory.
getFunction(functionName);
    if (function == null) {
      return value;
    }
    return function.apply(value);
  }

  @Override
  public boolean beforeFunction(String functionName) {
    return parseFunctionFactory.isBeforeFunction(functionName);
  }
}
```

4.2.3 日志持久化逻辑

同样在 LogRecordInterceptor 的代码中引用了 ILogRecordService，这个 Service 主要包含了日志记录的接口。

```
public interface ILogRecordService {
    /**
     * 保存 log
     *
     * @param logRecord 日志实体
     */
    void record(LogRecord logRecord);

}
```

业务可以实现这个保存接口，然后把日志保存在任何存储介质上。这里给了一个 2.2 节介绍的通过 log.info 保存在日志文件中的例子，业务可以把保存设置成异步或者同

步，可以和业务放在一个事务中保证操作日志和业务的一致性，也可以新开辟一个事务，保证日志的错误不影响业务的事务。业务可以保存在 Elasticsearch、数据库或者文件中，用户可以根据日志结构和日志的存储实现相应的查询逻辑。

```
@Slf4j
public class DefaultLogRecordServiceImpl implements ILogRecordService {

    @Override
//    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void record(LogRecord logRecord) {
        log.info("【logRecord】log={}", logRecord);
    }
}
```

4.2.4 Starter 逻辑封装

上面逻辑代码已经介绍完毕，那么接下来需要把这些组件组装起来，然后让用户去使用。在使用这个组件的时候只需要在 Springboot 的入口上添加一个注解 @EnableLogRecord(tenant = “com.mzt.test”)。其中 tenant 代表租户，是为了多租户使用的。

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@EnableTransactionManagement
@EnableLogRecord(tenant = "com.mzt.test")
public class Main {

    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }
}
```

再看下 EnableLogRecord 的代码，代码中 Import 了 `LogRecordConfigureSelector.class`，在 `LogRecordConfigureSelector` 类中暴露了 `LogRecordProxyAutoConfiguration` 类。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(LogRecordConfigureSelector.class)
public @interface EnableLogRecord {
```

```
    String tenant();

    AdviceMode mode() default AdviceMode.PROXY;
}
```

`LogRecordProxyAutoConfiguration` 就是装配上面组件的核心类了，代码如下：

```
@Configuration
@Slf4j
public class LogRecordProxyAutoConfiguration implements ImportAware {

  private AnnotationAttributes enableLogRecord;


  @Bean
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
  public LogRecordOperationSource logRecordOperationSource() {
    return new LogRecordOperationSource();
  }

  @Bean
  @ConditionalOnMissingBean(IFunctionService.class)
  public IFunctionService functionService(ParseFunctionFactory
parseFunctionFactory) {
    return new DefaultFunctionServiceImpl(parseFunctionFactory);
  }

  @Bean
  public ParseFunctionFactory parseFunctionFactory(@Autowired
List<IParseFunction> parseFunctions) {
    return new ParseFunctionFactory(parseFunctions);
  }

  @Bean
  @ConditionalOnMissingBean(IParseFunction.class)
  public DefaultParseFunction parseFunction() {
    return new DefaultParseFunction();
  }


  @Bean
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
  public BeanFactoryLogRecordAdvisor logRecordAdvisor(IFunctionService
functionService) {
    BeanFactoryLogRecordAdvisor advisor =
            new BeanFactoryLogRecordAdvisor();
    advisor.setLogRecordOperationSource(logRecordOperationSource());
```

```
    advisor.setAdvice(logRecordInterceptor(functionService));
    return advisor;
  }

  @Bean
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
  public LogRecordInterceptor logRecordInterceptor(IFunctionService
functionService) {
    LogRecordInterceptor interceptor = new LogRecordInterceptor();
    interceptor.setLogRecordOperationSource(logRecordOperationSource());
    interceptor.setTenant(enableLogRecord.getString("tenant"));
    interceptor.setFunctionService(functionService);
    return interceptor;
  }

  @Bean
  @ConditionalOnMissingBean(IOperatorGetService.class)
  @Role(BeanDefinition.ROLE_APPLICATION)
  public IOperatorGetService operatorGetService() {
    return new DefaultOperatorGetServiceImpl();
  }

  @Bean
  @ConditionalOnMissingBean(ILogRecordService.class)
  @Role(BeanDefinition.ROLE_APPLICATION)
  public ILogRecordService recordService() {
    return new DefaultLogRecordServiceImpl();
  }

  @Override
  public void setImportMetadata(AnnotationMetadata importMetadata) {
    this.enableLogRecord = AnnotationAttributes.fromMap(
            importMetadata.getAnnotationAttributes(EnableLogRecord.class.
getName(), false));
    if (this.enableLogRecord == null) {
      log.info("@EnableCaching is not present on importing class");
    }
  }
}
```

这个类继承 ImportAware 是为了拿到 EnableLogRecord 上的租户属性，这个类使用变量 logRecordAdvisor 和 logRecordInterceptor 装配了 AOP，同时把自定义函数注入到了 logRecordAdvisor 中。

对外扩展类：分别是 `IOperatorGetService`、`ILogRecordService`、`IParseFunction`。业务可以自己实现相应的接口，因为配置了 @ConditionalOnMissing-

Bean，所以用户的实现类会覆盖组件内的默认实现。

5. 总结

这篇文章介绍了操作日志的常见写法，以及如何让操作日志的实现更加简单、易懂；通过组件的四个模块，介绍了组件的具体实现。对于上面的组件介绍，大家如果有疑问，也欢迎在文末留言，我们会进行答疑。

6. 作者简介

站通，2020 年加入美团，基础研发平台 / 研发质量及效率部工程师。

7. 参考资料

- Canal
- spring-framework
- Spring Expression Language (SpEL)
- ThreadLocal、InheritableThreadLocal、TransmittableThreadLocal 三者之间区别

8. 招聘信息

美团研发质量及效率部 ，致力于建设业界一流的持续交付平台，现招聘基础组件方向相关的工程师，坐标北京 / 上海。欢迎感兴趣的同学加入。可投递简历至：chao.yu@meituan.com（邮件主题请注明：美团研发质量及效率部）。

基于 SSD 的 Kafka 应用层缓存架构设计与实现

作者：世吉　仕禄

Kafka 在美团数据平台的现状

Kafka 出色的 I/O 优化以及多处异步化设计，相比其他消息队列系统具有更高的吞吐，同时能够保证不错的延迟，十分适合应用在整个大数据生态中。

目前在美团数据平台中，Kafka 承担着数据缓冲和分发的角色。如下图所示，业务日志、接入层 Nginx 日志或线上 DB 数据通过数据采集层发送到 Kafka，后续数据被用户的实时作业消费、计算，或经过数仓的 ODS 层用作数仓生产，还有一部分则会进入公司统一日志中心，帮助工程师排查线上问题。

目前美团线上 Kafka 规模：

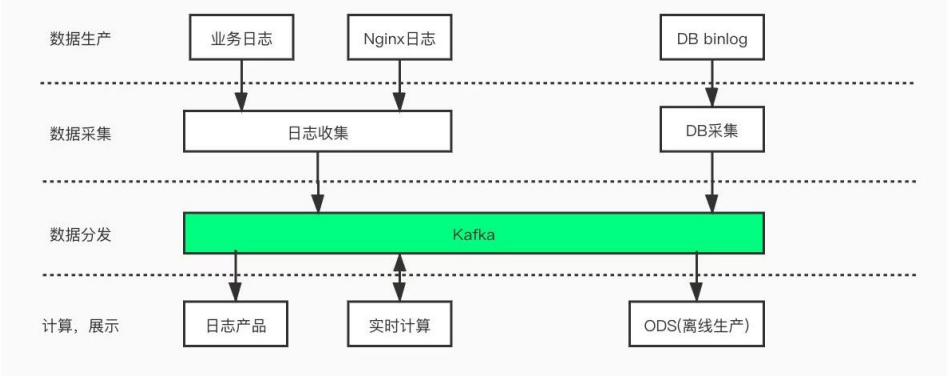
- **集群规模**：节点数达 6000+，集群数 100+。
- **集群承载**：Topic 数 6 万 +，Partition 数 41 万 +。
- **处理的消息规模**：目前每天处理消息总量达 8 万亿，峰值流量为 1.8 亿条 / 秒
- **提供的服务规模**：目前下游实时计算平台运行了 3 万 + 作业，而这其中绝大多数的数据源均来自 Kafka。

Kafka 线上痛点分析 & 核心目标

当前 Kafka 支撑的实时作业数量众多，单机承载的 Topic 和 Partition 数量很大。这种场景下很容易出现的问题是：同一台机器上不同 Partition 间竞争 PageCache 资源，相互影响，导致整个 Broker 的处理延迟上升、吞吐下降。

接下来，我们将结合 Kafka 读写请求的处理流程以及线上统计的数据来分析一下 Kafka 在线上的痛点。

原理分析



Kafka 处理读写流程的示意图

对于 Produce 请求：Server 端的 I/O 线程统一将请求中的数据写入到操作系统的 PageCache 后立即返回，当消息条数到达一定阈值后，Kafka 应用本身或操作系统内核会触发强制刷盘操作（如左侧流程图所示）。

对于 Consume 请求：主要利用了操作系统的 ZeroCopy 机制，当 Kafka Broker 接收到读数据请求时，会向操作系统发送 sendfile 系统调用，操作系统接收后，首先试图从 PageCache 中获取数据（如中间流程图所示）；如果数据不存在，会触发缺页异常中断将数据从磁盘读入到临时缓冲区中（如右侧流程图所示），随后通过 DMA 操作直接将数据拷贝到网卡缓冲区中等待后续的 TCP 传输。

综上所述，Kafka 对于单一读写请求均拥有很好的吞吐和延迟。处理写请求时，数据写入 PageCache 后立即返回，数据通过异步方式批量刷入磁盘，既保证了多数写请求都能有较低的延迟，同时批量顺序刷盘对磁盘更加友好。处理读请求时，实时消费的作业可以直接从 PageCache 读取到数据，请求延迟较小，同时 ZeroCopy 机制能够减少数据传输过程中用户态与内核态的切换，大幅提升了数据传输的效率。

但当同一个 Broker 上同时存在多个 Consumer 时，就可能会由于多个 Consumer 竞争 PageCache 资源导致它们同时产生延迟。下面我们以两个 Consumer 为例详细说明：

如上图所示，Producer 将数据发送到 Broker，PageCache 会缓存这部分数据。当所有 Consumer 的消费能力充足时，所有的数据都会从 PageCache 读取，全部 Consumer 实例的延迟都较低。此时如果其中一个 Consumer 出现消费延迟（图中的 Consumer Process2），根据读请求处理流程可知，此时会触发磁盘读取，在从磁盘读取数据的同时会预读部分数据到 PageCache 中。当 PageCache 空间不足时，会按照 LRU 策略开始淘汰数据，此时延迟消费的 Consumer 读取到的数据会替换 PageCache 中实时的缓存数据。后续当实时消费请求到达时，由于 PageCache 中的数据已被替换掉，会产生预期外的磁盘读取。这样会导致两个后果：

- 消费能力充足的 Consumer 消费时会失去 PageCache 的性能红利。
- 多个 Consumer 相互影响，预期外的磁盘读增多，HDD 负载升高。

我们针对 HDD 的性能和读写并发的影响做了梯度测试，如下图所示：

可以看到，随着读并发的增加，HDD 的 IOPS 和带宽均会明显下降，这会进一步影响整个 Broker 的吞吐以及处理延迟。

线上数据统计

目前 Kafka 集群 TP99 流量在 170MB/s，TP95 流量在 100MB/s，TP50 流量为 50-60MB/s；单机的 PageCache 平均分配为 80GB，取 TP99 的流量作为参考，在此流量以及 PageCache 分配情况下，PageCache 最大可缓存数据时间跨度为 80*1024/170/60 = 8min，可见当前 Kafka 服务整体对延迟消费作业的容忍性极低。该情况下，一旦部分作业消费延迟，实时消费作业就可能会受到影响。

同时，我们统计了线上实时作业的消费延迟分布情况，延迟范围在 0-8min（实时消费）的作业只占 80%，说明目前存在线上存在 20% 的作业处于延迟消费的状态。

痛点分析总结

总结上述的原理分析以及线上数据统计，目前线上 Kafka 存在如下问题：

- 实时消费与延迟消费的作业在 PageCache 层次产生竞争，导致实时消费产生非预期磁盘读。
- 传统 HDD 随着读并发升高性能急剧下降。
- 线上存在 20% 的延迟消费作业。

按目前的 PageCache 空间分配以及线上集群流量分析，Kafka 无法对实时消费作业提供稳定的服务质量保障，该痛点亟待解决。

预期目标

根据上述痛点分析，我们的预期目标为保证实时消费作业不会由于 PageCache 竞争而被延迟消费作业影响，保证 Kafka 对实时消费作业提供稳定的服务质量保障。

解决方案

为什么选择 SSD

根据上述原因分析可知，解决目前痛点可从以下两个方向来考虑：

- 消除实时消费与延迟消费间的 PageCache 竞争，如：让延迟消费作业读取的数据不回写 PageCache，或增大 PageCache 的分配量等。
- 在 HDD 与内存之间加入新的设备，该设备拥有比 HDD 更好的读写带宽与 IOPS。

对于第一个方向，由于 PageCache 由操作系统管理，若修改其淘汰策略，那么实现难度较为复杂，同时会破坏内核本身对外的语义。另外，内存资源成本较高，无法进行无限制的扩展，因此需要考虑第二个方向。

SSD 目前发展日益成熟，相较于 HDD，SSD 的 IOPS 与带宽拥有数量级级别的提升，很适合在上述场景中当 PageCache 出现竞争后承接部分读流量。我们对 SSD 的性能也进行了测试，结果如下图所示：

从图中可以发现，随着读取并发的增加，SSD 的 IOPS 与带宽并不会显著降低。通过该结论可知，我们可以使用 SSD 作为 PageCache 与 HDD 间的缓存层。

架构决策

在引入 SSD 作为缓存层后，下一步要解决的关键问题包括 PageCache、SSD、HDD 三者间的数据同步以及读写请求的数据路由等问题，同时我们的新缓存架构需要充分匹配 Kafka 引擎读写请求的特征。本小节将介绍新架构如何在选型与设计上解决上述提到的问题。

Kafka 引擎在读写行为上具有如下特性：

- 数据的消费频率随时间变化，越久远的数据消费频率越低。
- 每个分区（Partition）只有 Leader 提供读写服务。
- 对于一个客户端而言，消费行为是线性的，数据并不会重复消费。

下文给出了两种备选方案，下面将对两种方案给出我们的选取依据与架构决策。

备选方案一：基于操作系统内核层实现

目前开源的缓存技术有 FlashCache、BCache、DM-Cache、OpenCAS 等，其中 BCache 和 DM-Cache 已经集成到 Linux 中，但对内核版本有要求，受限于内核版本，我们仅能选用 FlashCache/OpenCAS。

如下图所示，FlashCache 以及 OpenCAS 二者的核心设计思路类似，两种架构的核心理论依据为“数据局部性”原理，将 SSD 与 HDD 按照相同的粒度拆成固定的管理单元，之后将 SSD 上的空间映射到多块 HDD 层的设备上（逻辑映射 or 物理映射）。在访问流程上，与 CPU 访问高速缓存和主存的流程类似，首先尝试访问 Cache 层，如果出现 CacheMiss，则会访问 HDD 层，同时根据数据局部性原理，这部分数据将回写到 Cache 层。如果 Cache 空间已满，会通过 LRU 策略替换部分数据。

FlashCache/OpenCAS 提供了四种缓存策略：WriteThrough、WriteBack、WriteAround、WriteOnly。 由于第四种不做读缓存，这里我们只看前三种。

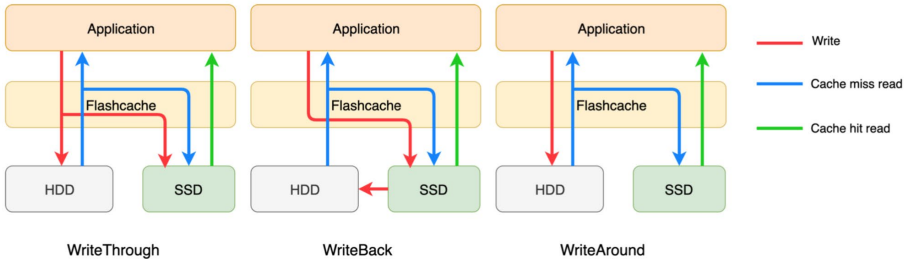
写入：

- **WriteThrough**：数据写操作在写入 SSD 的同时会写入到后端存储。

- **WriteBack**：数据写操作仅写入 SSD 即返回，由缓存策略 flush 到后台存储。
- **WriteAround**：数据写入操作直接写入后端存储，同时 SSD 对应的缓存会失效。

读取：

- **WriteThrough/WriteBack/WriteAround**：首先读取 SSD，命中不了的将再次读取后端存储，并数据会被刷入到 SSD 缓存中。



更多详细实现细节，极大可参见这二者的官方文档：

- FlashCache
- OpenCAS

备选方案二：Kafka 应用内部实现

上文提到的第一类备选方案中，核心的理论依据“数据局部性”原理与 Kafka 的读写特性并不能完全吻合，“数据回刷”这一特性依然会引入缓存空间污染问题。同时，上述架构基于 LRU 的淘汰策略也与 Kafka 读写特性存在矛盾，在多 Consumer 并发消费时，LRU 淘汰策略可能会误淘汰掉一些近实时数据，导致实时消费作业出现性能抖动。

可见，备选方案一并不能完全解决当前 Kafka 的痛点，需要从应用内部进行改造。整体设计思路如下，将数据按照时间维度分布在不同的设备中，近实时部分的数据缓存在 SSD 中，这样当出现 PageCache 竞争时，实时消费作业从 SSD 中读取数据，保证实时作业不会受到延迟消费作业影响。下图展示了基于应用层实现的架构处理读请求的流程：

当消费请求到达 Kafka Broker 时，Kafka Broker 直接根据其维护的消息偏移量（Offset）和设备的关系从对应的设备中获取数据并返回，并且在读请求中并不会将 HDD 中读取的数据回刷到 SSD，防止出现缓存污染。同时访问路径明确，不会由于 Cache Miss 而产生的额外访问开销。

下表对不同候选方案进行了更加详细的对比：

决策	优势	不足
基于操作系统内核层实现	1.数据路由对应用层透明，对应用代码改动量小。 2.开源软件自身的健壮性由社区维护，可用性较好（前提：社区比较活跃）。	1.FlashCache/OpenCAS每种模式下都会将数据回刷到SSD缓存中，与PageCache相似，都会发生缓存污染。 2.发生Cache Miss时会多一次对设备的访问，延迟增加。 3.所有的Meta数据都由操作系统维护，内核消耗的内存会增加，在与其他引擎混布的场景下会导致其他服务可申请的内存减少。
Kafka应用内部实现	1.设计缓存策略时充分考虑了Kafka的读写特性，确保近实时的数据消费请求全部落在SSD上，保证这部分请求处理的低延迟，同时从HDD读取的数据不会回刷到SSD防止缓存污染。 2.由于每个日志段都有唯一明确的状态，因此每次请求的查询路径最短，不存在因Cache Miss带来的额外性能开销。	1.需要在Server端代码上进行改进，涉及的开发及测试工作量较大。 2.随社区大版本升级，也需要迭代上这些改进的代码。但可将相关代码贡献社区，解决迭代问题。

最终，结合与 Kafka 读写特性的匹配度，整体工作量等因素综合考虑，我们采用 Kafka 应用层实现这一方案，因为该方案更贴近 Kafka 本身读写特性，能更加彻底地解决 Kafka 的痛点。

新架构设计

概述

根据上文对 Kafka 读写特性的分析，我们给出应用层基于 SSD 的缓存架构的设计目标：

- 数据按时间维度分布在不同的设备上，近实时数据分布在 SSD 上，随时间的推移淘汰到 HDD 上。
- Leader 分区中所有数据均写入 SSD 中。
- 从 HDD 中读取的数据不回刷到 SSD 中。

依据上述目标，我们给出应用层基于 SSD 的 Kafka 缓存架构实现：

Kafka 中一个 Partition 由若干 LogSegment 构成，每个 LogSegment 包含两个索

引文件以及日志消息文件。一个 Partition 的若干 LogSegment 按 Offset（相对时间）维度有序排列。

根据上一小节的设计思路，我们首先将不同的 LogSegment 标记为不同的状态，如图所示（图中上半部分）按照时间维度分为 OnlyCache、Cached 以及 WithoutCache 三种常驻状态。而三种状态的转换以及新架构对读写操作的处理如图中下半部分所示，其中标记为 OnlyCached 状态的 LogSegment 只存储在 SSD 上，后台线程会定期将 Inactive（没有写流量）的 LogSegment 同步到 SSD 上，完成同步的 LogSegment 被标记为 Cached 状态。最后，后台线程将会定期检测 SSD 上的使用空间，当空间达到阈值时，后台线程将会按照时间维度将距离现在最久的 LogSegment 从 SSD 中移除，这部分 LogSegment 会被标记为 WithoutCache 状态。

对于写请求而言，写入请求依然首先将数据写入到 PageCache 中，满足阈值条件后将会刷入 SSD。对于读请求（当 PageCache 未获取到数据时），如果读取的 offset 对应的 LogSegment 的状态为 Cached 或 OnlyCache，则数据从 SSD 返回（图中 LC2-LC1 以及 RC1），如果状态为 WithoutCache，则从 HDD 返回（图中 LC1）。

对于 Follower 副本的数据同步，可根据 Topic 对延迟以及稳定性的要求，通过配置决定写入到 SSD 还是 HDD。

关键优化点

上文介绍了基于 SSD 的 Kafka 应用层缓存架构的设计概要以及核心设计思路，包括读写流程、内部状态管理以及新增后台线程功能等。本小节将介绍该方案的关键优化点，这些优化点均与服务的性能息息相关。主要包括 LogSegment 同步以及 Append 刷盘策略优化，下面将分别进行介绍。

LogSegment 同步

LogSegment 同步是指将 SSD 上的数据同步到 HDD 上的过程，该机制在设计时主要有以下两个关键点：

- **同步的方式**：同步方式决定了 HDD 上对 SSD 数据的可见时效性，从而会影响故障恢复以及 LogSegment 清理的及时性。
- **同步限速**：LogSegment 同步过程中通过限速机制来防止同步过程中对正常读写请求造成影响

同步方式

关于 LogSegment 的同步方式，我们给出了三种备选方案，下表列举了三种方案的介绍以及各自的优缺点：

备选方案	简介	优势	不足
复用Follower同步机制	复用Follower同步Leader的机制，通过内部Fetcher发送网络请求将数据同步到HDD	有可以直接复用的模块	1. 需要与网络处理模块进行交互，同步流程复杂化 2.由于所有网络请求共用同一个请求队列，此类请求出现异常可能会影响其他读写请求
内部线程实时同步数据到HDD	后台线程实时同步数据到HDD	1.数据在HDD上的时效性较高 2.单次请求比较小，对IO较为友好	实时同步的数据均来自active的LogSegment，由于数据处于不断变化中，会带来额外的数据一致性维护的成本
后台线程只同步Inactive数据段到HDD	后台线程只同步Inactive的LogSegment到HDD	对于Inactive的LogSegment，由于已经停止写入，因此通常处于稳态，数据一致性较好维护	由于Inactive状态的LogSegment均为完整的一段，需要对拷贝速率与对设备负载进行Trade Off

最终，我们对一致性维护代价、实现复杂度等因素综合考虑，选择了后台同步 Inactive 的 LogSegment 的方式。

同步限速

LogSegment 同步行为本质上是设备间的数据传输，会同时在两个设备上产生额外的读写流量，占用对应设备的读写带宽。同时，由于我们选择了同步 Inactive 部分的数据，需要进行整段的同步。如果在同步过程中不加以限制会对服务整体延迟造成较大的影响，主要表现在下面两个方面：

- 从单盘性能角度，由于 SSD 的性能远高于 HDD，因此在数据传输时，HDD 写入带宽会被写满，此时其他的读写请求会出现毛刺，如果此时有延迟消费从 HDD 上读取数据或者 Follower 正在同步数据到 HDD 上，会造成服务抖动。
- 从单机部署的角度，单机会部署 2 块 SSD 与 10 块 HDD，因此在同步过程中，1 块 SSD 需要承受 5 块 HDD 的写入量，因此 SSD 同样会在同步过程中出现性能毛刺，影响正常的请求响应延迟。

基于上述两点，我们需要在 LogSegment 同步过程中增加限速机制，总体的限速原则为在不影响正常读写请求延迟的情况下尽可能快速地进行同步。因为同步速度过慢会导致 SSD 数据无法被及时清理而最终被写满。同时为了可以灵活调整，该配置也被设置为单 Broker 粒度的配置参数。

日志追加刷盘策略优化

除了同步问题，数据写入过程中的刷盘机制同样影响服务的读写延迟。该机制的设计不仅会影响新架构的性能，对原生 Kafka 同样会产生影响。

下图展示了单次写入请求的处理流程：

在 Produce 请求处理流程中，首先根据当前 LogSegment 的位置与请求中的数据信息确定是否需要滚动日志段，随后将请求中的数据写入到 PageCache 中，更新 LEO 以及统计信息，最后根据统计信息确定是否需要触发刷盘操作，如果需要则通过 `fileChannel.force` 强制刷盘，否则请求直接返回。

在整个流程中，除日志滚动与刷盘操作外，其他操作均为内存操作，不会带来性能问题。日志滚动涉及文件系统的操作，目前，Kafka 中提供了日志滚动的扰动参数，防止多个 Segment 同时触发滚动操作给文件系统带来压力。针对日志刷盘操作，目前 Kafka 给出的机制是以固定消息条数触发强制刷盘（目前线上为 50000），该机制只能保证在入流量一定时，消息会以相同的频率刷盘，但无法限制每次刷入磁盘的数据量，对磁盘的负载无法提供有效的限制。

如下图所示，为某磁盘在午高峰时间段 write_bytes 的瞬时值，在午高峰时间段，由于写入流量的上升，在刷盘过程中会产生大量的毛刺，而毛刺的值几乎接近磁盘最大的写入带宽，这会使读写请求延迟发生抖动。

针对该问题，我们修改了刷盘的机制，将原本的按条数限制修改为按实际刷盘的速率限制，对于单个 Segment，刷盘速率限制为 2MB/s。该值考虑了线上实际的平均消息大小，如果设置过小，对于单条消息较大的 Topic 会过于频繁的进行刷新，在流量较高时反而会加重平均延迟。目前该机制已在线上小范围灰度，右图展示了灰度后同

时间段对应的 write_bytes 指标，可以看到相比左图，数据刷盘速率较灰度前明显平滑，最高速率仅为 40MB/s 左右。

对于 SSD 新缓存架构，同样存在上述问题，因此在新架构中，在刷盘操作中同样对刷盘速率进行了限制。

方案测试

测试目标

- 验证基于应用层的 SSD 缓存架构能够避免实时作业受到延迟作业的影响。
- 验证相比基于操作系统内核层实现的缓存层架构，基于应用层的 SSD 架构在不同流量下读写延迟更低。

测试场景描述

- 构建 4 个集群：新架构集群、普通 HDD 集群、FlashCache 集群、Open-CAS 集群。
- 每个集群 3 个节点。
- 固定写入流量，比较读、写耗时。
- 延迟消费设置：只消费相对当前时间 10~150 分钟的数据（超过 PageCache 的承载区域，不超过 SSD 的承载区域）。

测试内容及重点关注指标

- Case1: 仅有延迟消费时，观察集群的生产和消费性能。
  - 重点关注的指标：写耗时、读耗时，通过这 2 个指标体现出读写延迟。
  - 命中率指标：HDD 读取量、HDD 读取占比（HDD 读取量 / 读取总量）、SSD 读取命中率，通过这 3 个指标体现出 SSD 缓存的命中率。
- Case2: 存在延迟消费时，观察实时消费的性能。
  - 重点指标：实时作业的 SLA（服务质量）的 5 个不同时间区域的占比情况。

测试结果

从单 Broker 请求延迟角度看：

在刷盘机制优化前，SSD 新缓存架构在所有场景下，较其他方案都具有明显优势。

刷盘机制优化后，其余方案在延迟上服务质量有提升，在较小流量下由于 flush 机制的优化，新架构与其他方案的优势变小。当单节点写入流量较大时（大于 170MB）优势明显。

从延迟作业对实时作业的影响方面看：

新缓存架构在测试所涉及的所有场景中，延迟作业都不会对实时作业产生影响，符合预期。

总结与未来展望

Kafka 在美团数据平台承担统一的数据缓存和分发的角色，针对目前由于 Page-Cache 互相污染、进而引发 PageCache 竞争导致实时作业被延迟作业影响的痛点，我们基于 SSD 自研了 Kafka 的应用层缓存架构。本文主要介绍 Kafka 新架构的设计思路以及与其他开源方案的对比。与普通集群相比，新缓存架构具备非常明显的优势：

- **降低读写耗时**：比起普通集群，新架构集群读写耗时降低 80%。
- **实时消费不受延迟消费的影响**：比起普通集群，新架构集群实时读写性能稳定，不受延时消费的影响。

目前，这套缓存架构优已经验证完成，正在灰度阶段，未来也优先部署到高优集群。其中涉及的代码也将提交给 Kafka 社区，作为对社区的回馈，也欢迎大家跟我们一起交流。

作者简介

世吉，仕禄，均为美团数据平台工程师。

美团图数据库平台建设及业务实践

作者：登昌　梁帅　高辰　杨鑫　尊远　王超

1. 前言

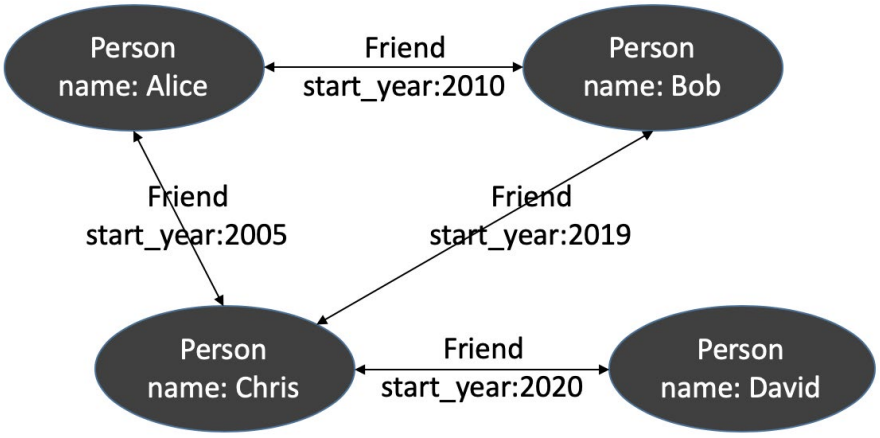
图数据结构，能够很自然地表征现实世界。比如用户、门店、骑手这些实体可以用图中的点来表示，用户到门店的消费行为、骑手给用户的送餐行为可以用图中的边来表示。使用图的方式对场景建模，便于描述复杂关系。在美团，也有比较多的图数据存储及多跳查询需求，概括起来主要包括以下 4 个方面：

- **图谱挖掘**：美团有美食图谱、商品图谱、旅游图谱、用户全景图谱在内的近 10 个领域知识图谱，数据量级大概在千亿级别。在迭代、挖掘数据的过程中，需要一种组件对这些图谱数据进行统一的管理。
- **安全风控**：业务部门有内容风控的需求，希望在商户、用户、评论中通过多跳查询来识别虚假评价；在支付时进行金融风控的验证，实时多跳查询风险点。
- **链路分析**：包括代码分析、服务治理、数据血缘管理，比如公司数据平台上有很多 ETL Job，Job 和 Job 之间存在强弱依赖关系，这些强弱依赖关系形成了一张图，在进行 ETL Job 的优化或者故障处理时，需要对这个图进行实时查询分析。
- **组织架构**：公司组织架构的管理，实线汇报链、虚线汇报链、虚拟组织的管理，以及商家连锁门店的管理。比如，维护一个商家在不同区域都有哪些门店，能够进行多层关系查找或者逆向关系搜索。

总体来说，美团需要一种组件来管理千亿级别的图数据，解决图数据存储以及多跳查询问题。海量图数据的高效存储和查询是图数据库研究的核心课题，如何在大规模分布式场景中进行工程落地是我们面临的痛点问题。传统的关系型数据库、NoSQL 数据库可以用来存储图数据，但是不能很好处理图上多跳查询这一高频的操作。

Neo4j 公司在社交场景（见图 1）里做了传统关系型数据库 MySQL 跟图数据库

Neo4j 的查询性能对比 [1]，在一个包含 100 万人、每人约有 50 个朋友的社交网络里找最大深度为 5 的朋友的朋友，实验结果表明多跳查询中图数据库优势明显（见图 2）。然而选取或者自主研发一款高吞吐、低查询延时、能存储海量数据且易用的图数据库非常困难。下面将介绍美团在图数据库选型及平台建设方面的一些工作。



社交场景示例

图 1

深度	关系型数据库的执行时间(s)	Neo4j的执行时间(s)	返回的记录条数
2	0.016	0.01	~2500
3	30.267	0.168	~110000
4	1543.505	1.359	~600000
5	未完成	2.132	~800000

关系数据库 vs 图数据库(多跳查询)

图 2

2. 图数据库选型

在图数据库的选型上我们主要考虑了以下 5 点：(A) 项目开源，暂不考虑需付费的图数据库；(B) 分布式架构设计，具备良好的可扩展性；© 毫秒级的多跳查询延迟；(D)

支持千亿量级点边存储；(E) 具备批量从数仓导入数据的能力。

分析 DB-Engines[2] 上排名前 30 的图数据库，剔除不开源的项目，我们将剩余的图数据库分为三类：

- **第一类：Neo4j[3]、ArangoDB[4]、Virtuoso[5]、TigerGraph[6]、RedisGraph[7]**。此类图数据库只有单机版本开源可用，性能优秀，但不能应对分布式场景中数据的规模增长，即不满足选型要求（B）、（D）。
- **第二类：JanusGraph[8]、HugeGraph[9]**。此类图数据库在现有存储系统之上新增了通用的图语义解释层，图语义层提供了图遍历的能力，但是受到存储层或者架构限制，不支持完整的计算下推，多跳遍历的性能较差，很难满足 OLTP 场景下对低延时的要求，即不满足选型要求（C）。
- **第三类：DGraph[10]、NebulaGraph[11]**。此类图数据库根据图数据的特点对数据存储模型、点边分布、执行引擎进行了全新设计，对图的多跳遍历进行了深度优化，基本满足我们的选型要求。

DGraph 是由前 Google 员工 Manish Rai Jain 离职创业后，在 2016 年推出的图数据库产品，底层数据模型是 RDF[12]，基于 Go 语言编写，存储引擎基于 BadgerDB[13] 改造，使用 RAFT 保证数据读写的强一致性。

NebulaGraph 是由前 Facebook 员工叶小萌离职创业后，在 2019 年 推出的图数据库产品，底层数据模型是属性图，基于 C++ 语言编写，存储引擎基于 RocksDB[14] 改造，使用 RAFT 保证数据读写的强一致性。

这两个项目的创始人都在互联网公司图数据库领域深耕多年，对图数据库的落地痛点有深刻认识，整体的架构设计也有较多相似之处。在图数据库最终的选型上，我们基于 LDBC-SNB 数据集 [15] 对 NebulaGraph、DGraph、HugeGraph 进行了深度性能测评，测试详情见文章：主流开源分布式图数据库 Benchmark，从测试结果看 NebulaGraph 在数据导入、实时写入及多跳查询方面性能均优于竞品。此外，NebulaGraph 社区活跃，问题响应速度快，所以团队最终选择基于 NebulaGraph

来搭建图数据库平台。

3. NebulaGraph 架构

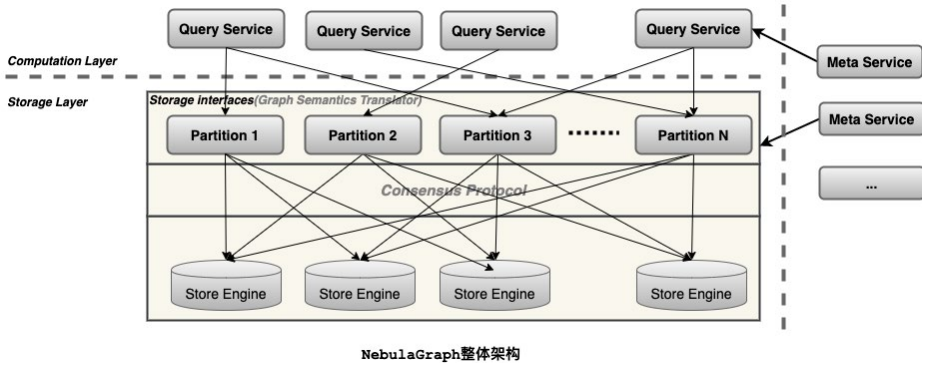


图 3

一个完整的 NebulaGraph 集群包含三类服务，即 Query Service、Storage Service 和 Meta Service。每类服务都有其各自的可执行二进制文件，既可以部署在同一节点上，也可以部署在不同的节点上。下面是 NebulaGraph 架构设计（见图 3）的几个核心点 [16][17]。

- **Meta Service**：架构图中右侧为 Meta Service 集群，它采用 Leader/Follower 架构。Leader 由集群中所有的 Meta Service 节点选出，然后对外提供服务；Followers 处于待命状态，并从 Leader 复制更新的数据。一旦 Leader 节点 Down 掉，会再选举其中一个 Follower 成为新的 Leader。Meta Service 不仅负责存储和提供图数据的 Meta 信息，如 Schema、数据分片信息等；同时还提供 Job Manager 机制管理长耗时任务，负责指挥数据迁移、Leader 变更、数据 compaction、索引重建等运维操作。
- **存储计算分离**：在架构图中 Meta Service 的左侧，为 NebulaGraph 的主要服务，NebulaGraph 采用存储与计算分离的架构，虚线以上为计算，以下为存储。存储计算分离有诸多优势，最直接的优势就是，计算层和存储层可以根据各自的情况弹性扩容、缩容。存储计算分离还带来了另一个优势：使水平扩

展成为可能。此外，存储计算分离使得 Storage Service 可以为多种类型的计算层或者计算引擎提供服务。当前 Query Service 是一个高优先级的 OLTP 计算层，而各种 OLAP 迭代计算框架会是另外一个计算层。

- **无状态计算层**：每个计算节点都运行着一个无状态的查询计算引擎，而节点彼此间无任何通信关系。计算节点仅从 Meta Service 读取 Meta 信息以及和 Storage Service 进行交互。这样设计使得计算层集群更容易使用 K8s 管理或部署在云上。每个查询计算引擎都能接收客户端的请求，解析查询语句，生成抽象语法树（AST）并将 AST 传递给执行计划器和优化器，最后再交由执行器执行。
- **Shared-nothing 分布式存储层**：Storage Service 采用 Shared-nothing 的分布式架构设计，共有三层，最底层是 Store Engine，它是一个单机版 Local Store Engine，提供了对本地数据的 get/put/scan/delete 操作，该层定义了数据操作接口，用户可以根据自己的需求定制开发相关 Local Store Plugin。目前，NebulaGraph 提供了基于 RocksDB 实现的 Store Engine。在 Local Store Engine 之上是 Consensus 层，实现了 Multi Group Raft，每一个 Partition 都对应了一组 Raft Group。在 Consensus 层上面是 Storage interfaces，这一层定义了一系列和图相关的 API。这些 API 请求会在这一层被翻译成一组针对相应 Partition 的 KV 操作。正是这一层的存在，使得存储服务变成了真正的图存储。否则，Storage Service 只是一个 KV 存储罢了。而 NebulaGraph 没把 KV 作为一个服务单独提出，最主要的原因便是图查询过程中会涉及到大量计算，这些计算往往需要使用图的 Schema，而 KV 层没有数据 Schema 概念，这样设计比较容易实现计算下推，是 NebulaGraph 查询性能优越的主要原因。

NebulaGraph 基于 C++ 实现，架构设计支持存储千亿顶点、万亿边，并提供毫秒级别的查询延时。我们在 3 台 48U192G 物理机搭建的集群上灌入 10 亿美食图谱数据对 NebulaGraph 的功能进行了验证。

- 一跳查询 TP99 延时在 5ms 内，两跳查询 TP99 延时在 20ms 内，一般的多

跳查询 TP99 延时在百毫秒内。

- 集群在线写入速率约为 20 万 Records/s。
- 支持通过 Spark 任务离线生成 RocksDB 底层 SST File，直接将数据文件载入到集群中，即类似 HBase BulkLoad 能力。
- 提供了类 SQL 查询语言，对于新增的业务需求，只需构造 NebulaGraph SQL 语句，易于理解且能满足各类复杂查询要求。
- 提供联合索引、GEO 索引，可通过实体属性或者关系属性查询实体、关系，或者查询在某个经纬度附近 N 米内的实体。
- 一个 NebulaGraph 集群中可以创建多个 Space（概念类似 MySQL 的 DataBase），并且不同 Space 中的数据在物理上是隔离的。

4. 图数据库平台建设

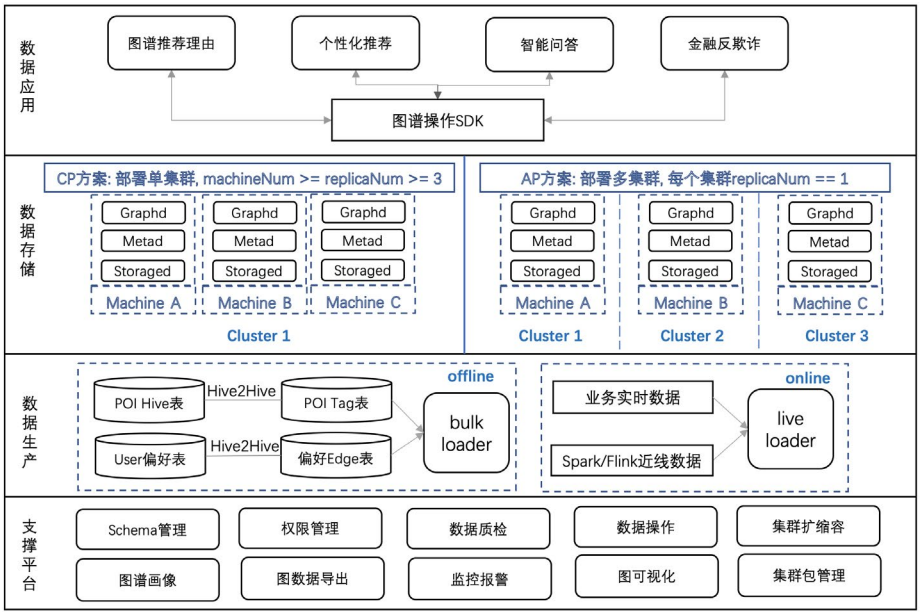


图 4

为了统一管理图数据，减少工程同学在图数据库集群上的运维压力，我们基于开源分布式图数据库 NebulaGraph，搭建了一套一站式图数据库自助管理平台（见图 4），

该平台包含以下 4 层：

- 数据应用层。业务方可以在业务服务中引入图谱 SDK，实时地对图数据进行增删改查。
- 数据存储层。支持两种图数据库集群的部署。
  - 第一种部署方式是 CP 方案，即 Consistency & Partition tolerance。单集群部署，集群中机器数量大于等于副本的数量，副本数量大于等于 3 。只要集群中有大于副本数一半的机器存活，整个集群就可以对外正常提供服务。CP 方案保证了数据读写的强一致性，但这种部署方式下集群可用性不高。
  - 第二种部署方式是 AP 方案，即 Availability & Partition tolerance。在一个应用中部署多个图数据库集群，每个集群数据副本数为 1 ，多集群之间进行互备。这种部署方式的好处在于整个应用对外的可用性高，但数据读写的一致性要差些。
- 数据生产层。图数据主要有两种来源，第一种是业务方把数仓中数据通过 ETL Job 转成点和边的 Hive 表，然后离线导入到图数据库中；第二种是业务线上实时产生的数据、或者通过 Spark/Flink 等流式处理产生的近线数据，调用在线批量写接口实时灌到图数据库中。
- 支撑平台。提供了 Schema 管理、权限管理、数据质检、数据增删改查、集群扩缩容、图谱画像、图数据导出、监控报警、图可视化、集群包管理等功能。

与业界方案相比，团队主导设计的图数据库平台除了支持存储千亿顶点、万亿边，具备毫秒级别查询能力外，还提供了如下四项能力：应用可用性 SLA 达 99.99%；支持每小时百亿量级数据导入；实时写入数据时保证多集群数据最终一致性；易用的图谱可视化能力。下面将介绍具体的设计思路。

4.1 高可用模块设计

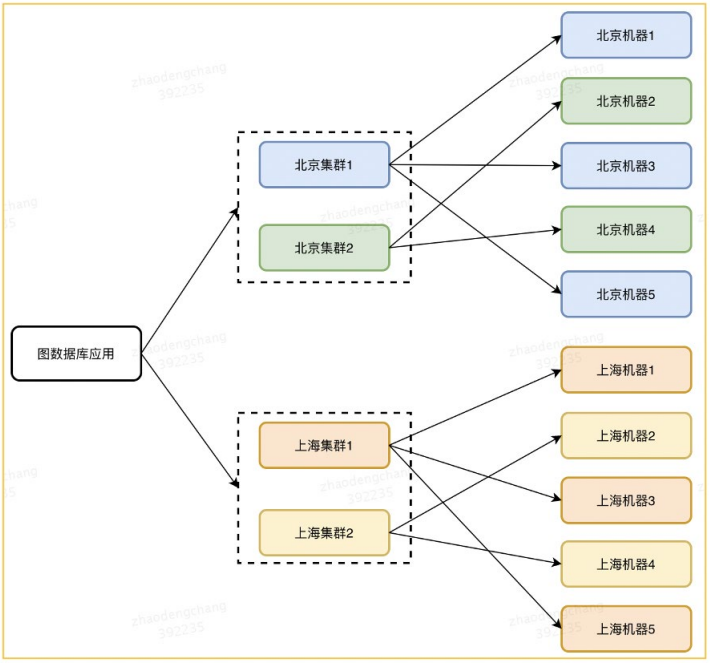


图 5

首先介绍单应用多集群高可用模块的设计（AP 方案）。为什么有 AP 方案的设计呢？因为接入图数据库平台的业务方比较在意的指标是集群可用性。在线服务对集群的可用性要求非常高，最基础的要求是集群可用性能达到 4 个 9，即一年里集群的不可用时间要小于一个小时。对于在线服务来说，服务或者集群的可用性是整个业务的生命线，如果这点保证不了，即使集群提供的能力再多再丰富，那么业务方也不会考虑使用，可用性是业务选型的基础。

另外，公司要求中间件要有跨区域容灾能力，即要具备在多个地域部署多集群的能力。我们分析了平台接入方的业务需求，大约 80% 的场景是 T+1 全量导入数据、线上只读。在这种场景下，对图数据的读写强一致性要求并不高，因此我们设计了单应用多集群这种部署方案。

AP 方案部署方式可以参考图 5，一个业务方在图数据库平台上创建了 1 个应用并部

署了 4 个集群，其中北京 2 个、上海 2 个，平时这 4 个集群同时对外提供服务。假如现在北京集群 1 挂了，那么北京集群 2 可以提供服务。如果说真那么不巧，北京集群都挂了，或者北京侧对外的网络不可用，那么上海的集群也可以提供服务。在这种部署方式下，平台会尽可能地通过一些方式来保证整个应用的可用性。然后每个集群内部尽量部署同机房的机器，因为图数据库集群内部 RPC 非常多，如果有跨机房或者跨区域的频繁调用，整个集群对外的性能会比较低。

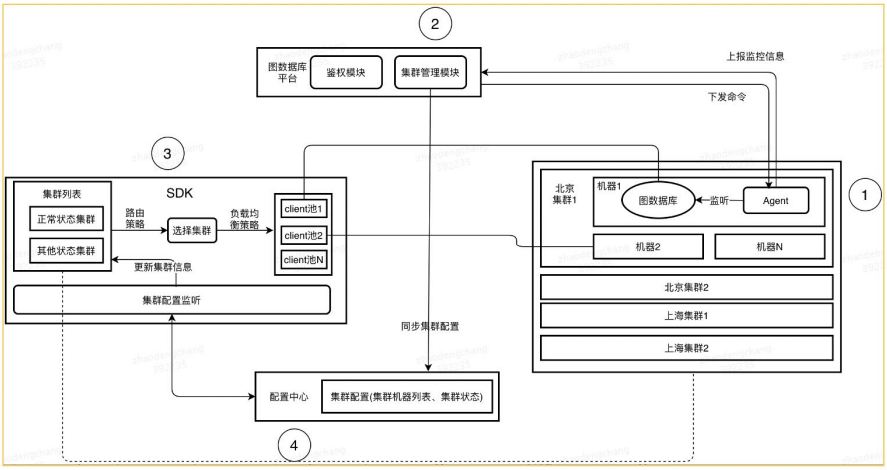


图 6

高可用模块主要包含下面 4 个部分，如上图 6 所示：

第一部分是右侧的图数据库 Agent，它是部署在图数据库集群的一个进程，用来收集机器和图数据库三个核心模块的信息，并上报到图数据库平台。Agent 能够接收图数据库平台的命令并对图数据库进行操作。

第二部分是图数据库平台，它主要是对集群进行管理，并同步图数据库集群的状态到配置中心。

第三部分是图数据库 SDK，主要负责管理连接到图数据库集群的连接。如果业务方发送了某个查询请求，SDK 会进行集群的路由和负载均衡，选择出一条高质量的连接来发送请求。此外，SDK 还会处理图数据库集群中问题机器的自动降级以及恢复，

并且支持平滑切换集群的数据版本。

第四部分是配置中心，类似 ZooKeeper，存储集群的当前状态。

4.2 每小时百亿量级数据导入模块设计

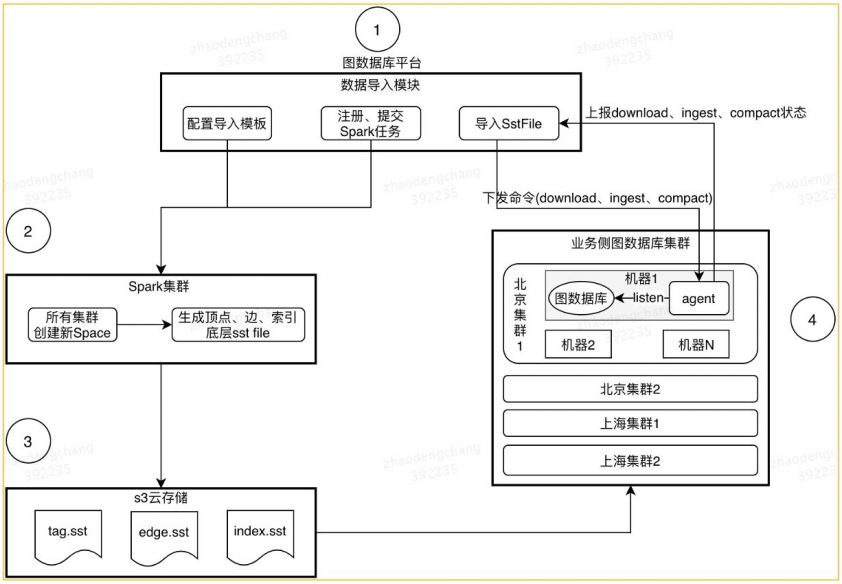


图 7

第二个模块是每小时百亿量级数据导入模块，平台在 2019 年底 - 2020 年初全量导入数据的方式是调用 NebulaGraph 对外提供的批量数据导入接口，这种方式的数据写入速率大概是每小时 10 亿级别，导入百亿数据大概要耗费 10 个小时，耗时较长。此外，在以几十万每秒的速度导数据的过程中，会长期占用机器的 CPU、IO 资源，一方面会对机器造成损耗，另一方面数据导入过程中集群对外提供的读性能会变弱。

为了解决上面两个问题，平台进行了如下优化：在 Spark 集群中直接生成图数据库底层文件 SST File，再借助 RocksDB 的 Bulkload 功能直接 ingest 文件到图数据库。

数据导入的核心流程可以参考图 7，当用户执行导数据操作后，图数据库平台会向公司的 Spark 集群提交一个 Spark 任务，在 Spark 任务中会生成图数据库里相关的点、边以及点索引、边索引相关的 SST 文件，并上传到美团的 S3 云存储上。文件

生成后，图数据库平台会通知应用中多个集群去下载这些存储文件，之后完成 ingest 跟 compact 操作，最后完成数据版本的切换。

为兼顾各个业务方的不同需求，平台统一了应用导入、集群导入、离线导入、在线导入，以及全量导入、增量导入这些场景，然后细分成下面九个阶段，从流程上保证在导数据过程中应用整体的可用性：SST File 生成 、SST File 下载 、ingest、compact、数据校验、增量回溯、数据版本切换、集群重启、数据预热。

4.3 实时写入多集群数据同步模块设计

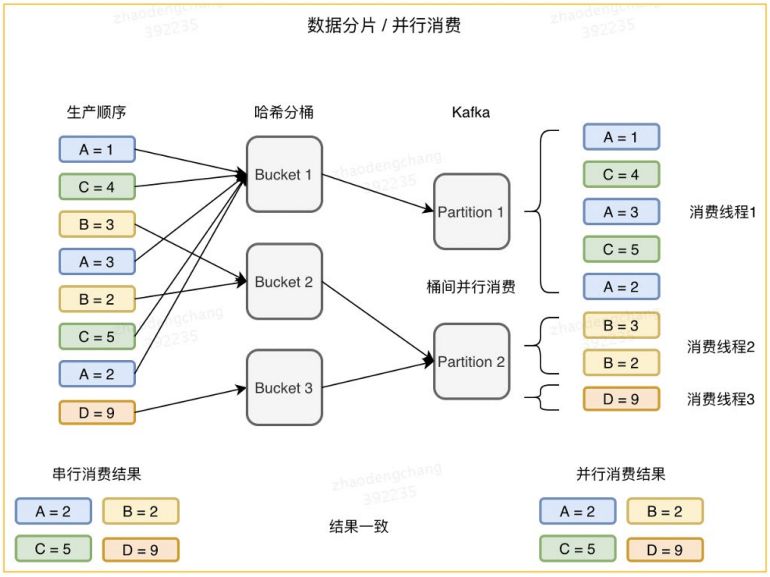


图 8

第三个模块是实时写入多集群数据同步模块，平台约有 15% 的需求场景是在实时读取数据时，还要把新产生的业务数据实时写入集群，并且对数据的读写强一致性要求不高。就是说，业务方写到图数据库里的数据，不需要立马能读到。针对上述场景，业务方在使用单应用多集群这种部署方案时，多集群里的数据需要保证最终一致性。针对这一需求，我们做了以下设计。

第一部分是引入 Kafka 组件，业务方在服务中通过 SDK 对图数据库进行写操作时，

SDK 并不直接写图数据库，而是把写操作写到 Kafka 队列里，之后由该应用下的多个集群异步消费这个 Kafka 队列。

第二部分是集群在应用级别可配置消费并发度，来控制数据写入集群的速度。具体流程如下：

- SDK 对用户写操作语句做语法解析，将其中点边的批量操作拆解成单个点边操作，即对写语句做一次改写。
- Agent 消费 Kafka 时确保每个点及其出边相关操作在单个线程里顺序执行（见图 8），保证这点就能保证各个集群执行完写操作后最终的结果是一致的。
- 并发扩展：通过改变 Kafka 分片数、Agent 中消费 Kafka 线程数来调整 Kafka 中操作的消费速度。如果未来图数据库支持事务的话，上面的配置需要调整成单分片单线程消费，有必要对设计方案再做优化调整。

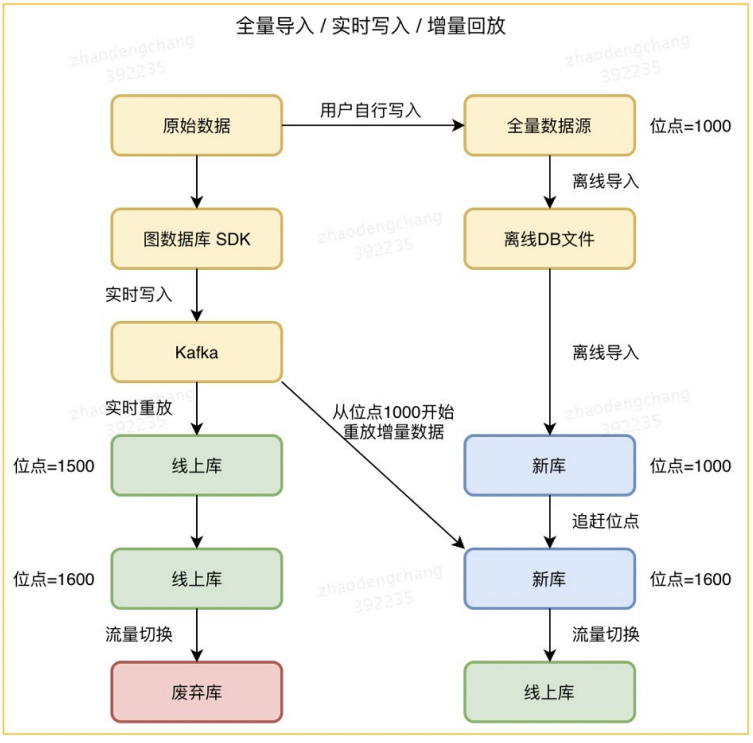


图 9

第三部分是在实时写入数据过程中，平台可以同步生成一个全量数据版本，并做平滑切换（见图 9），确保数据的不重、不漏、不延迟。

4.4 图可视化模块设计

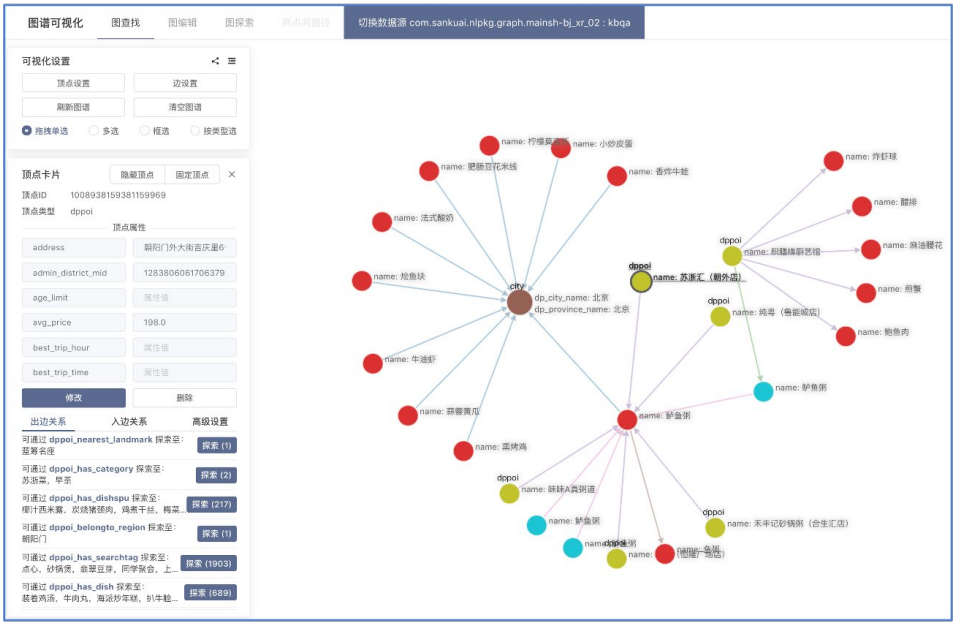


图 10

第四个模块是图可视化模块（见图 10），主要是用于解决子图探索问题。当用户在图数据库平台通过可视化组件查看图数据时，能尽量通过恰当的交互设计来避免因为节点过多而引发爆屏。主要包括以下几个功能：

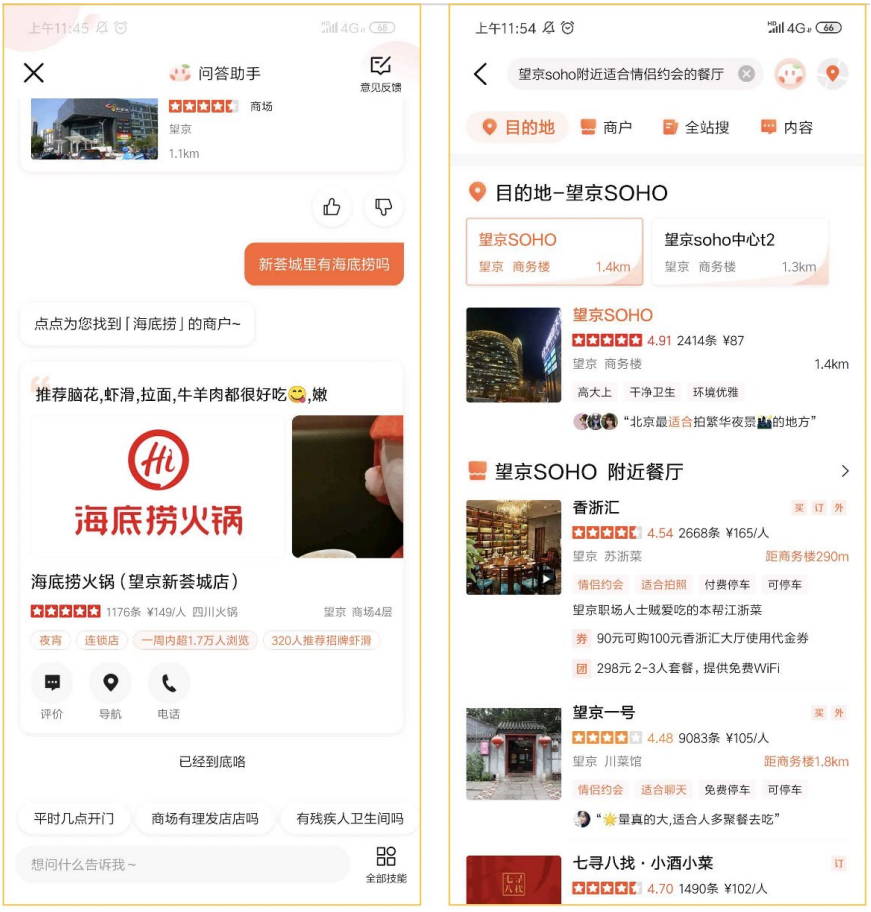
- 通过 ID 或者索引查找顶点。
- 能查看顶点和边的卡片（卡片中展示点边属性和属性值），可以单选、多选、框选以及按类型选择顶点。
- 图探索，当用户点击某个顶点时，系统会展示它的一跳邻居信息，包括该顶点有哪些出边？通过这个边它能关联到几个点？该顶点的入边又是什么情况？通过这种一跳信息的展示，用户在平台上探索子图的时候，可快速了解到周边的

邻居信息，更快地进行子图探索。在探索过程中，平台也支持通过属性对边进行过滤。

- 图编辑能力，让平台用户在不熟悉 NebulaGraph 语法的情况下也能增删改点边数据，对线上数据进行临时干预。

5. 业务实践

5.1 智能助理



该项目数据是基于美团商户数据、用户评论构建的餐饮娱乐知识图谱，覆盖美食、酒店、旅游等领域，包含 13 类实体和 22 类关系。目前，点边数量大概在百亿级别，

数据 T+1 全量更新，主要用于解决搜索或者智能助理里 KBQA（全称：Knowledge Based Question Answer）类问题。核心处理流程是通过 NLP 算法识别关系和实体后构造出 NebulaGraph SQL 语句，再到图数据库获取数据。

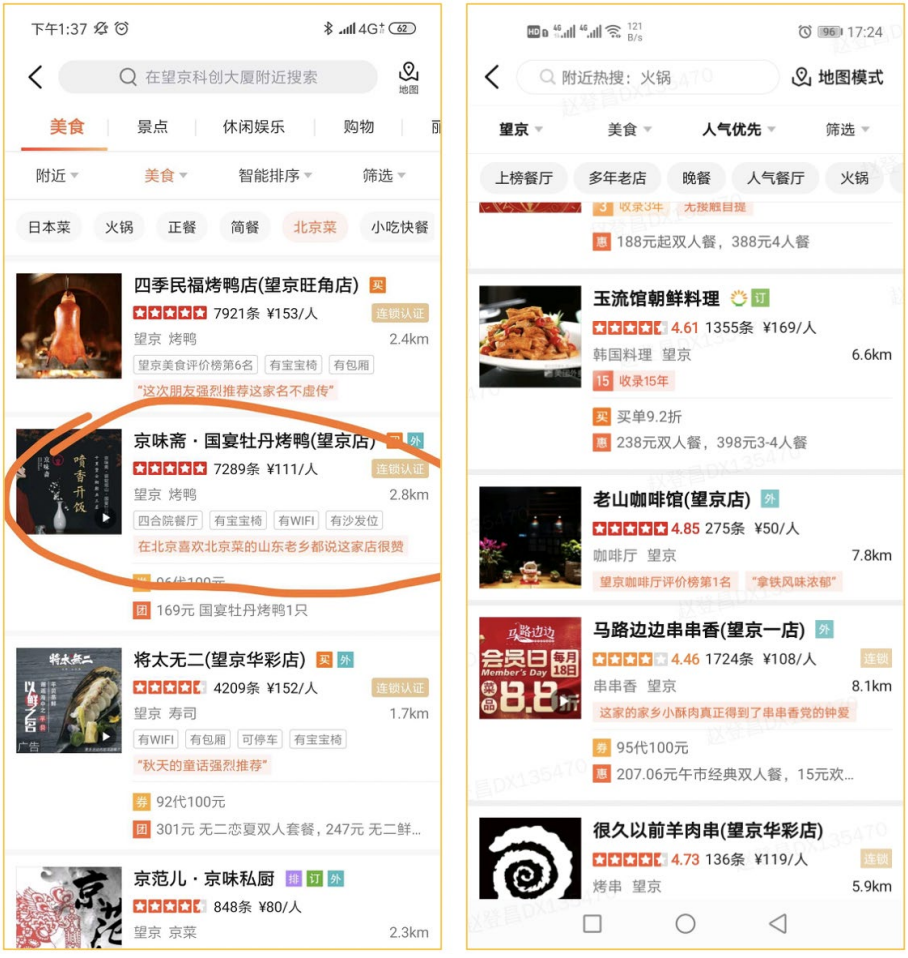
典型的应用场景包括商场找店，比如，某个用户想知道望京新荟城这个商场有没有海底捞，系统可以快速查出结果告诉用户；另一个场景是标签找店，用户想知道望京 SOHO 附近有没有适合情侣约会的餐厅，或者可以多加几个场景标签，系统都可以帮忙查找出来。

5.2 搜索召回



该项目数据是基于医美商家信息构建的医美知识图谱，包含 9 类实体和 13 类关系，点边数量在百万级别，同样也是 T+1 全量更新，主要用于大搜底层实时召回，返回与 Query 相关的商户、产品或医生信息，解决医美类搜索词少结果、无结果问题。比如，某个用户搜“啤酒肚”这种症状、或者“润百颜”这类品牌，系统可以召回相关的医美门店。

5.3 图谱推荐理由

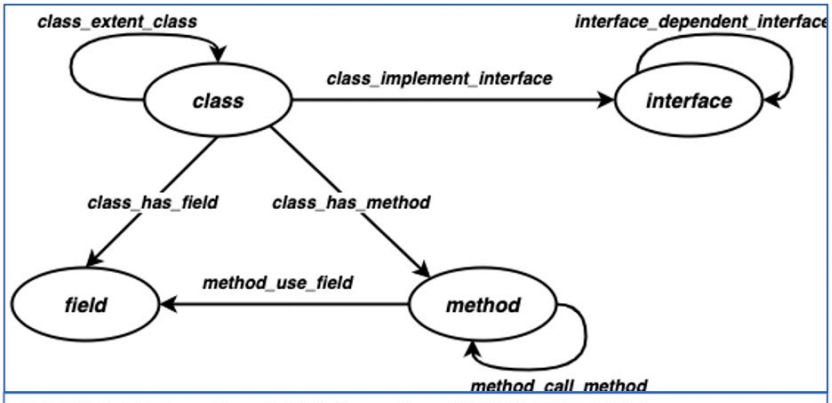


该项目数据来自用户的画像信息、商户的特征信息、用户半年内收藏 / 购买行为，数据量级是 10 亿级别，T+1 全量更新。现在美团 App 和点评 App 上默认的商户推荐列

表是由深度学习模型生成的，但模型并不会给出生成这个列表的理由，缺少可解释性。

而在图谱里用户跟商户之间天然存在多条连通路径，项目考虑选出一条合适路径来生成推荐理由，在 App 界面上展示给用户推荐某家店的原因。该项目基于用户的协同过滤算法来生成推荐理由，在家乡、消费水平、偏好类目、偏好菜系等多个组合维度中找出多条路径，然后给这些路径打分，选出一条分值较高的路径，之后按照特定 Pattern 产出推荐理由。通过上述方式，就可以获得“在北京喜欢北京菜的山东老乡都说这家店很赞”，或者“广州老乡都中意他家的正宗北京炸酱面”这类理由。

5.4 代码依赖分析



//查找最外层method到某个method的所有无环路径

```
(GO 1 to 30 STEPS FROM 2946345526231222882 OVER
method_call_method REVERSELY YIELD DISTINCT
method_call_method._dst AS id
MINUS
GO 1 to 30 STEPS FROM 2946345526231222882 OVER
method_call_method REVERSELY YIELD DISTINCT
method_call_method._src as id )
| FIND NOLOOP path FROM $-.id TO
2946345526231222882 OVER method_call_method UPTO
30 STEPS
```

//确认两个method间是否有路径

```
FIND SINGLE SHORTEST PATH FROM hash("method1")
TO hash("method2") OVER method_call_method UPTO
30 STEPS
```

该项目把代码库中代码依赖关系写入到图数据库。代码库中存在很多服务代码，这些服务会包括对外提供的接口，这些接口的实现依赖于该服务中某些类的成员函数，这些类的成员函数又依赖了本类的成员变量、成员函数或者其它类的成员函数，那么它们之间的依赖关系就形成了一张图，可以把这个图写到图数据库里做代码依赖分析。

典型应用场景是精准测试：当开发同学完成需求并向公司的代码仓库提交了 PR 后，可以把更改实时地写到图数据库中。这样的话，开发同学就能查到他所写的代码影响了哪些外部接口，并且借助图可视化组件查看调用路径。如果开发同学本来是要改接口 A 的行为，改了很多代码，但是他可能并不知道他改的代码也会影响到对外接口 B、C、D，这时候就可以用代码依赖分析来做个 Check，增加测试的完备性。

6. 总结与展望

目前，图数据库平台基本具备了对图数据的一站式自助管理功能。如果某个业务方要使用这种图数据库能力，那么业务方可以在平台上自助地创建图数据库集群、创建图的 Schema、导入图数据、配置导入数据的执行计划、引入平台提供的 SDK 对数据进行操作等等。平台侧主要负责各业务方图数据库集群的稳定性。目前，美团有三四十个业务已经在平台上落地，基本满足了各个业务方的需求。

未来规划主要有两个方向，第一，根据业务场景优化图数据库内核，提升平台稳定性，开发的通用 Feature 持续反哺 NebulaGraph 社区。第二，挖掘更多的图数据价值。现在平台仅支持图数据存储及多跳查询这种基本能力，后续将基于 NebulaGraph 去探索图学习、图计算的能力，为平台用户提供更多挖掘图数据价值的功能。

7. 作者信息

登昌、梁帅、高辰、杨鑫、尊远、王超等，均为美团搜索与 NLP 部工程师。

8. 招聘信息

如果你对“图存储”、“图学习”、“图计算”感兴趣，欢迎给我们投递简历，投递邮箱：zhaodengchang@meituan.com。

9. 参考资料

[1] Jim Webber, Emil Eifrem, Ian Robinson. Graph Databases(2nd Edition). O'Reilly Media, Inc. 2013: chapter 2.
[2] Neo4j Database, https://github.com/neo4j/neo4j.
[3] ArangoDB, https://github.com/arangodb/arangodb.
[4] Virtuoso, https://github.com/openlink/virtuoso-opensource.
[5] TigerGraph, https://github.com/tigergraph/ecosys.
[6] RedisGraph, https://github.com/RedisGraph/RedisGraph.
[7] JanusGraph, https://github.com/JanusGraph/janusgraph.
[8] HugeGraph, https://github.com/hugegraph/hugegraph.
[9] DGraph, https://github.com/dgraph-io/dgraph.
[10] NebulaGraph, [https://github.com/vesoft-inc/nebula.
[11] RDF, https://www.w3.org/RDF/](https://github.com/vesoft-inc/nebula.
[11] RDF, https://www.w3.org/RDF/).
[12] BadgerDB, https://github.com/dgraph-io/badger.
[13] RocksDB, https://github.com/facebook/rocksdb.
[14] LDBC-SNB 数据集：关联数据基准委员会提供的评测数据集，包含 26 亿顶点、177 亿关系 .
[15] NebulaGraph Architecture — A Bird's View, https://nebula-graph.io/posts/nebula-graph-architecture-overview/.
[16] An Introduction to NebulaGraph's Storage Engine.

OCTO 2.0：美团基于 Service Mesh 的服务治理系统详解

作者：舒超、世朋、来俊

OCTO 是美团内部的服务治理平台，包含服务通信框架、命名服务、服务数据中心和用户管理平台等组件，为公司内全部服务提供了整套的服务治理方案和统一的服务治理体验。我们在之前的几篇文章中分别从不同的角度介绍了 OCTO 平台的建设情况。包括：

- 《美团命名服务的挑战与演进》介绍了美团命名服务（MNS）从 1.0 到 2.0 演进的初衷、实现方案以及落地的效果，并介绍了命名服务作为一个技术中台组件，对业务的重要价值以及推动业务升级的一些成果。
- 《美团 OCTO 万亿级数据中心计算引擎技术解析》介绍了美团自研的 OCTO 数据中心计算引擎的演进历程及架构设计，同时详细地介绍其全面提升计算能力、吞吐能力、降低运维成本所采用的各项技术方案。
- 《团大规模微服务通信框架及治理体系 OCTO 核心组件开源》介绍了 OCTO 的核心组件 OCTO-RPC（JAVA 通信框架）、OCTO-NS（命名服务）和 OCTO-Portal（用户管理平台）的开源情况。
- 《复杂环境下落地 Service Mesh 的挑战与实践》从更高的视角介绍了美团当前的服务治理系统的发展史、所面临的困境和优化思路。对 Service Mesh 在美团大规模复杂服务场景下的落地做了深入的分析。

本文将继续介绍 OCTO 系统在 Service Mesh 化演进方面的工作。主要从数据面的角度，详细介绍各项技术方案的设计思路。

1. 整体架构

我们先看看 OCTO 2.0 的整体架构，如下图所示：

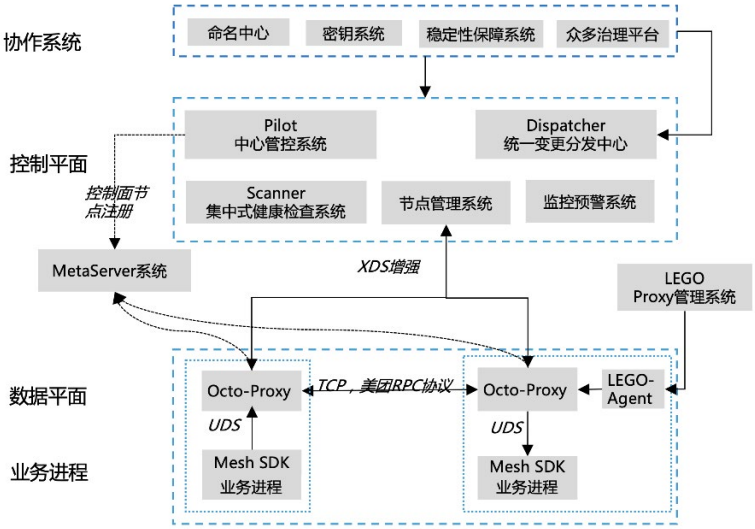


图 1　整体架构四个组成部分：基础设施、控制平面、数据平面和运维系统

基础设施是指美团现有的服务治理系统 OCTO1.0，包括 MNS、KMS（鉴权管理服务）、MCC（配置管理中心）、Rhino（熔断限流服务）等。这些系统接入到 OCTO 2.0 的控制平面，避免过多重构引入的不必要成本。

OCTO 2.0 控制平面摒弃了社区的 Istio，完全自研。数据平面基于开源 Envoy 改造。运维系统负责数据面组件的升级、发布等运维工作。更多的整体选型及原因可参考之前团队的文章：《美团下一代服务治理系统 OCTO2.0 的探索与实践》。

对 OCTO 2.0 中的功能，我们分为 Mesh 和服务治理两个维度。下面我们来重点看下流量劫持、无损重启、服务路由等几个常见的问题，并阐述美团是如何结合现有比较完善的服务治理系统来解决这些问题的。

2. Mesh 功能

2.1 流量劫持

OCTO 2.0 并未采用 Istio 的原生方案，而是使用 iptables 对进出 POD 的流量进行劫持。主要考量了以下两个因素：

- iptables 自身存在性能损失大、管控性差的问题：
  - iptables 在内核对于包的处理过程中定义了五个“hook point”，每个“hook point”各对应到一组规则链，outbond 流量将两次穿越协议栈并且经过这 5 组规则链匹配，在大并发场景下会损失转发性能。
  - iptables 全局生效，不能显式地禁止相关规则的修改，没有相关 ACL 机制，可管控性比较差。
- 在美团现有的环境下，使用 iptables 存在以下几个问题：
  - HULK 容器为富容器形态，业务进程和其他所有基础组件都处于同一容器中，这些组件使用了各种各样的端口，使用 iptables 容易造成误拦截。
  - 美团现在存在物理机、虚拟机、容器等多个业务运行场景，基于 iptables 的流量劫持方案在适配这些场景时复杂度较高。

鉴于上述问题，我们最终采用了 Unix Domain Socket 直连方式来实现业务进程和 OCTO-Proxy 之间的流量转发。

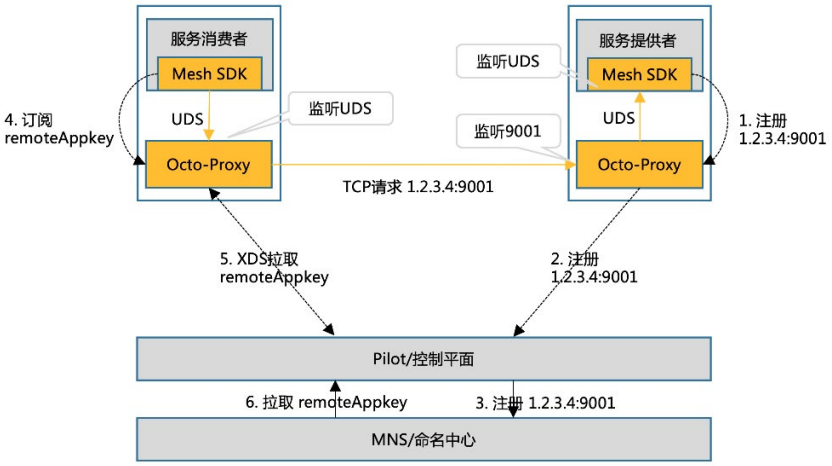


图 2　Unix Domain Socket 直连流量转发

在服务消费者一方，业务进程通过轻量的 Mesh SDK 和 OCTO-Proxy 所监听的 UDS 地址建立连接。在服务提供者一方，OCTO-Proxy 代替业务进程监听在 TCP 端口上，业务进程则监听在指定的 UDS 地址上。

该方案的优点是 Unix Domain Socket 相比于 iptable 劫持拥有更好的性能和更低的运维成本。缺点是需要 Mesh SDK 的支持。

2.2 服务订阅

原生 Envoy 的 CDS/EDS 请求是全量服务发现模式，即将系统中所有的服务列表都请求到数据面中。由于大规模集群中服务数量太多，真正所需的只是少数服务，所以需要改造成按需服务发现模式，仅请求需要访问的后端服务的节点列表。

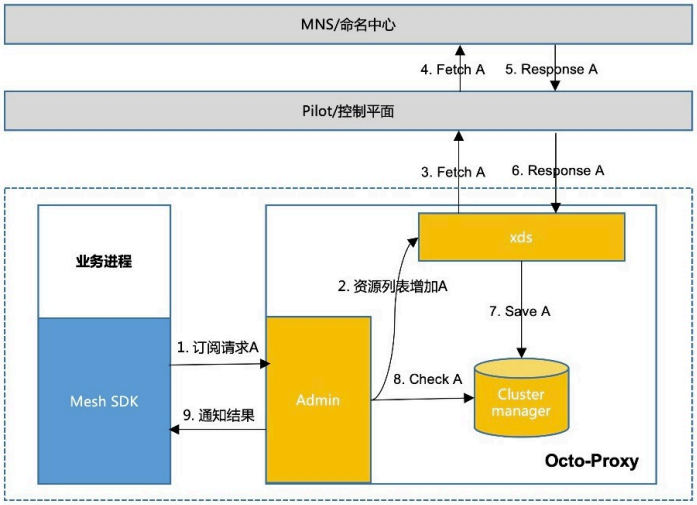


图 3　服务订阅流程

在业务进程启动后，需要通过 HTTP 的方式向 OCTO-Proxy 发起订阅请求。OCTO-Proxy 将所请求的后端服务 Appkey 更新到 XDS 中，XDS 再向控制面请求特定的服务资源。

为了增加整个过程中的健壮性，降低后续运维成本，我们做了部分适配。例如，OCTO-Proxy 的启动速度有可能慢于业务进程，所以在 Mesh SDK 中增加了请求的重试逻辑；将 Mesh SDK 和 OCTO-Proxy 之间的 http 请求改造成了同步请求，防止 Pilot 资源下发延时带来的问题；Mesh SDK 的订阅信息也会保存在本地文件中，以便 OCTO-Proxy 热重启或者故障重启时使用。

2.3 无损热重启

2.3.1 流量损失场景

如何在基础组件升级过程中提供持续、不间断的服务，做到业务流量无损及业务无感知，是所有基础组件面临的一大难题。这个问题对于 OCTO-Proxy 这种处于流量核心链路上的组件更加重要。社区原生的 Envoy 自身已经支持了热重启功能但并不彻底，在某些场景下还是不能做到完全的流量无损。

下图分别在短连接和长连接两种场景下来说明 OCTO-Proxy 热重启时的流量损耗问题。

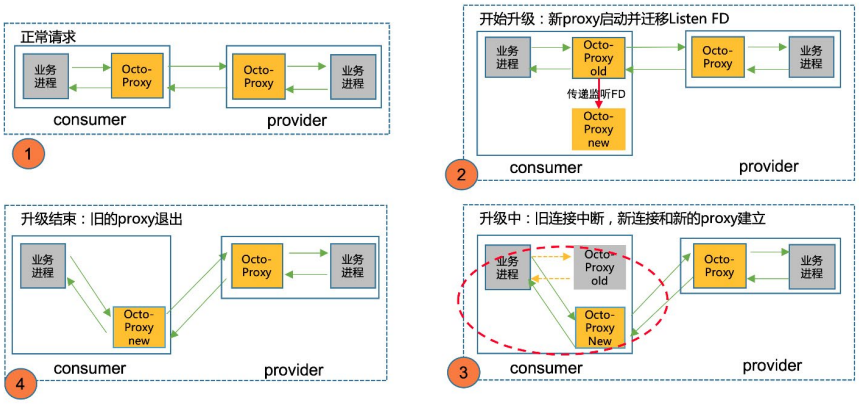


图 4　代理升级过程中流量损耗

对于短连接，所有新的连接会在新的 OCTO-Proxy 上创建，旧 OCTO-Proxy 上已有的连接在响应到来后主动断开。旧 OCTO-Proxy 的所有短连接逐渐断开，这就是“Drain”（排空）的过程。连接排空之后，旧 OCTO-Proxy 主动退出，新的 OCTO-Proxy 继续工作。整个过程中的流量，完全无损。

对于长连接方式，SDK 和旧 OCTO-Proxy 维持一条长连接不断开，并持续使用该连接发送请求。旧 OCTO-Proxy 进程最终退出时，该连接被动断开，此时可能尚有部分响应未返回，导致 Client 端请求超时。因此，Envoy 的热重启对长连接场景的支持并不完美。

为了支持基础组件在升级过程中提供不间断的服务，业界目前主要使用的是滚动发布（Rolling Update）的方式：服务器分批停止服务，执行更新，然后重新将其投入使用，直到集群中所有实例都更新为新版本。在此过程中会主动摘掉业务流量，保证升级过程中业务流量不丢失。

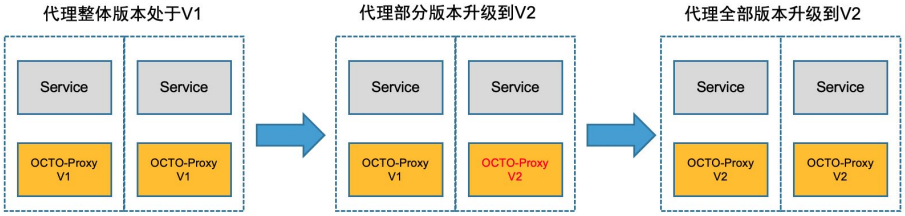


图 5　滚动升级

美团内部因为要兼容物理机、虚拟机、容器，业界云原生使用 K8s 滚动升级的方式不太适用，所以在现有环境下，如何保证服务治理系统的高可用性以及业务的高可靠性是一个非常棘手的事情。

2.3.2 适配方案

当前方案是把业务服务分为两种角色，即对外提供服务的 Server 端角色和对外发起请求的 Client 端角色，针对两种不同的角色采用不同的热更新支持。

Client 端 OCTO-Proxy 热更新：老的 OCTO-Proxy 在进入热重启状态后，对后续“新请求”直接返回含“热重启”标志的响应协议，Client SDK 在收到含“热重启”标志的响应协议时，应主动切换新连接并请求重试（以避免当前 Client SDK 持有的旧连接由于旧 OCTO-Proxy 热重启流程完成后被主动关闭的问题）。这里需要注意，Client SDK 需要“妥善”处理旧连接所在链路上遗留的所有“应答”协议。

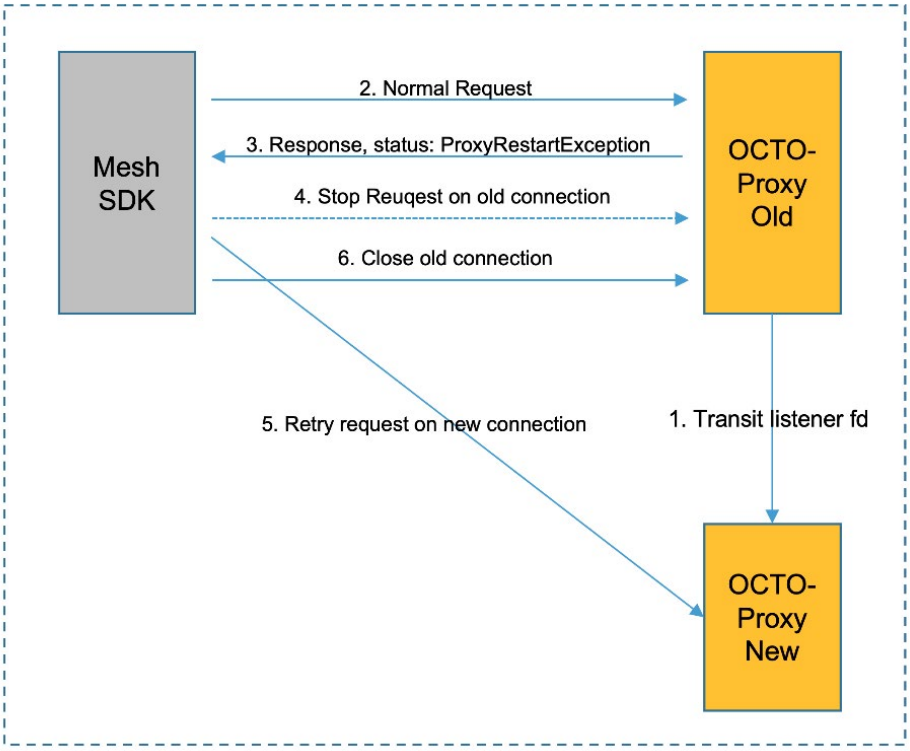


图 6　客户端 SDK/ 代理相互配合进行热升级

通过这种 Client SDK 和 OCTO-Proxy 间的交互配合，可以支持 Client 端在 OCTO-Proxy 升级过程中保证流量的安全性。

Server 端 OCTO-Proxy 热更新：Server 侧 OCTO-Proxy 在热重启开始后，即会主动向 Client 侧 OCTO-Proxy 发送 ProxyRestart 消息，通知 Client 侧 OCTO-Proxy 主动切换新连接，避免当前 Client 侧 OCTO-Proxy 持有的旧连接由于 Server 侧旧 OCTO-Proxy 热重启流程完成后被强制关闭的问题。Client 端 OCTO-Proxy 在收到“主动切换新连接”的请求后，应即时从可用连接池中移除，并“妥善”处理旧连接所在链路上遗留的所有“应答”协议（保守起见可标记连接不可用，并维持一段时间，比如 OCTO-Proxy 默认 drainTime）。

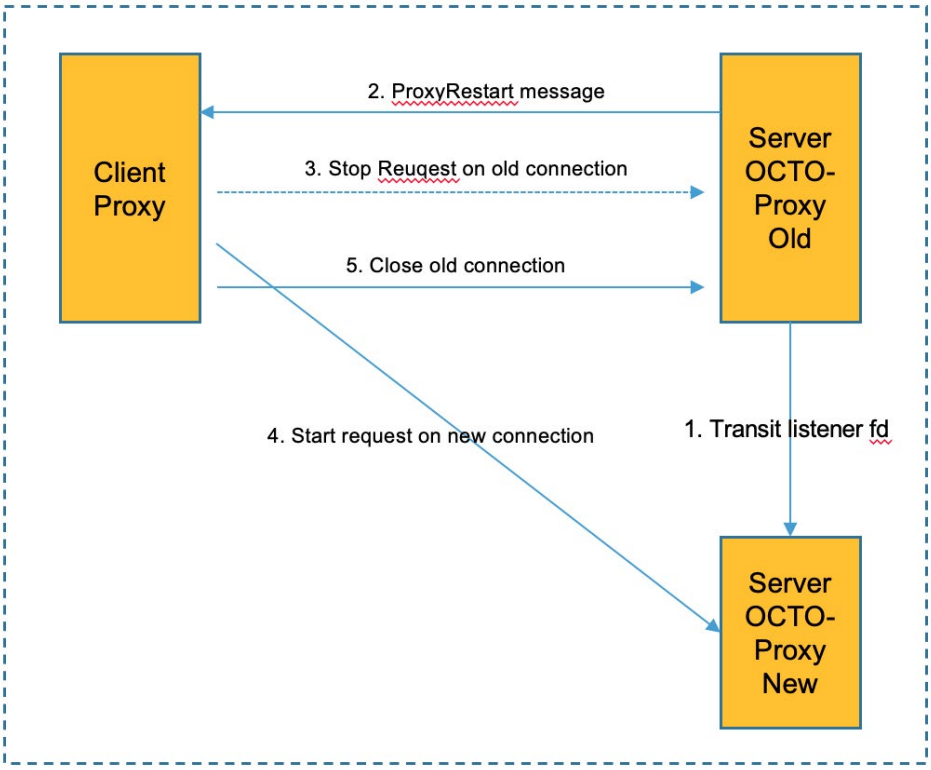


图 7　服务端反向主动通知客户端重连

2.4 数据面运维

2.4.1 LEGO 运维方案

云原生环境中，Envoy 运行在标准的 K8S Pod 中，通常会独立出一个 Sidecar 容器。可以使用 K8s 提供的能力来完成对 Envoy Sidecar 容器的管理，例如容器注入、健康检查、滚动升级、资源限制等。

美团内部所使用的容器运行时模式为“单容器模式”，即一个 Pod 内只包含一个容器（不考虑 Pause 容器）。由于业务进程以及所有的基础组件都运行在一个容器中，所以只能采用进程粒度的管理措施，无法做到容器粒度的管理。我们通过自研的 LEGO 平台解决了 OCTO-Proxy 的运维问题。

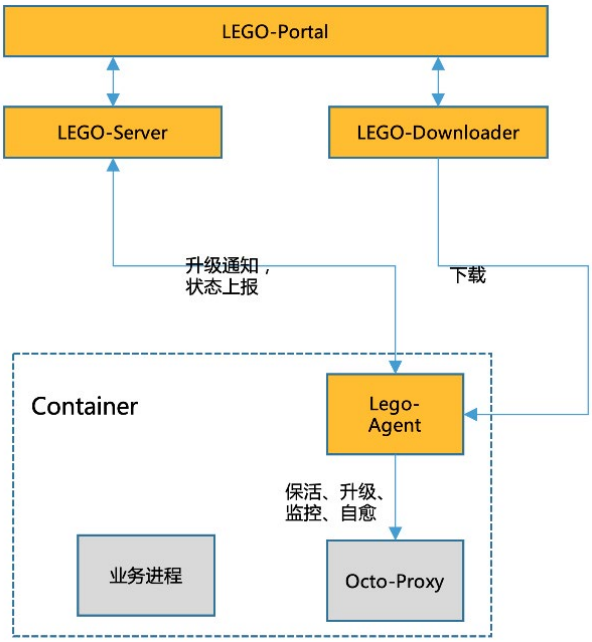


图 8　LEGO 数据面管理系统

我们对 LEGO-Agent 做了定制化改造，增加了对 OCTO-Proxy 的热重启过程的支持。另外，LEGO-Agent 还负责对 OCTO-Proxy 的健康检查、故障状态重启、监控信息上报和版本发布等。相对于原生 K8S 的容器重启方式，进程粒度重启会有更快的速度。

LEGO 系统的另一个优点是，可以同时支持容器、虚拟机和物理机等多种运行时环境。

2.4.2 云原生运维方案

目前，我们也正在探索 OCTO-Proxy 云原生的运维方案，在此场景下最大的区别就是从进程粒度运维转变为了容器粒度运维，与业务应用做到了更好的解耦，同时可享受不可变基础设施等理念带来的红利，但是此场景带来的一个问题就是如何管理容器粒度的热重启。

为此，我们通过自研的 Operator 对 OCTO-Proxy 容器进行全生命周期的运维管

理，并且期望通过 Operator 对 OCTO-Proxy 进行热重启，具体流程如下：

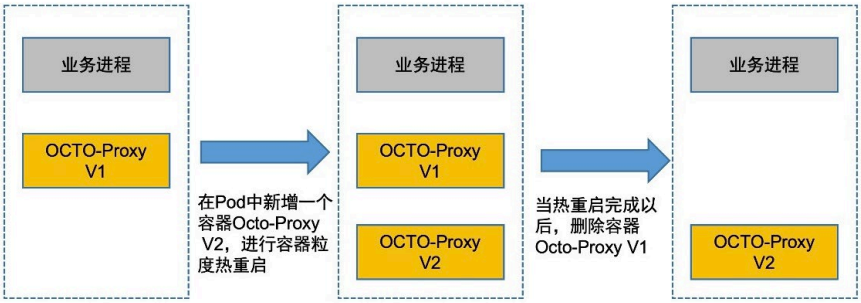


图 9　期望的 OCTO-Proxy 容器热重启流程

但是在实施过程中这个方案是有问题的，原因是 K8s 在底层设计上是不支持对运行中的 Pod 进行容器的增删操作，如果需要实现该方案，将需要对 K8s 底层组件进行定制化修改，这样将带来一定风险以及与社区的不兼容性。为了保证与社区的兼容性，我们对此热重启方案进行改造，最终使用双容器驻留热重启方案：

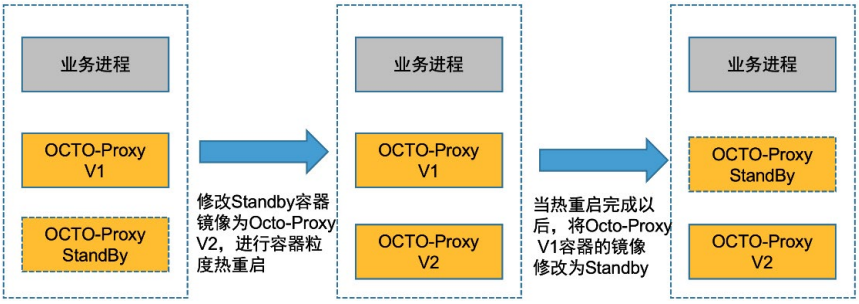


图 10　双容器驻留热重启方案

- 首先，我们在启动 Pod 时，给 OCTO-Proxy 不再只分配一个容器，而是分配两个容器，其中一个容器为正常运行状态，另一个容器为 standby 状态。
- 当需要对 OCTO-Proxy 进行热重启升级时，我们修改 standby 容器的镜像为最新 OCTO-Proxy 的镜像，此时开始热重启流程。
- 当热重启流程结束后，新容器进入正常工作状态，而旧容器进入等待状态，最终，我们将旧容器的镜像修改为 standby 镜像，结束整个热重启流程。

该方案利用一开始就驻留双容器在 Pod 内，避免了 K8s 对运行中 Pod 不能增删容器的限制，实现了在 Pod 中对 OCTO-Proxy 进行容器粒度的热重启。

3. 未来规划

目前接入 OCTO2.0 系统的服务数量数千，天粒度流量数十亿。随着 OCTO2.0 在公司的不断推广，越来越多的服务开始接入，对我们的稳定性和运维能力提出了更高的要求。如何更细致的了解线上 OCTO-Proxy 以及所称在的业务系统的健康状况、如何在出现故障时能更及时的监测到并快速恢复是我们最近 OCTO2.0 系统建设的重点。除此之外，未来 OCTO2.0 的规划重点还包括：

- 运维、发布和稳定性建设朝云原生化方向探索。
- 支持公司内部 Http 服务的 Mesh 化，降低对中心化网关的依赖。
- 全链路 mTLS 支持。

作者简介

舒超、世朋、来俊，均来自美团基础架构部基础开发组，从事 OCTO2.0 的开发工作。

团队简介

美团基础技术部 – 中间件研发中心 – 基础开发组，致力于研发公司级、业界领先的基础架构组件，研发范围涵盖分布式框架、命名服务、Service Mesh 等技术领域。欢迎感兴趣的同学发送简历至：edp.itu.zhaopin@meituan.com。

GraphQL 及元数据驱动架构在后端 BFF 中的实践

作者：陆晨　致远　陈琦

1. BFF 的由来

BFF 一词来自 Sam Newman 的一篇博文《Pattern:Backends For Frontends》，指的是服务于前端的后端。BFF 是解决什么问题的呢？据原文描述，随着移动互联网的兴起，原适应于桌面 Web 的服务端功能希望同时提供给移动 App 使用，而在这个过程中存在这样的问题：

- 移动 App 和桌面 Web 在 UI 部分存在差异。
- 移动 App 涉及不同的端，不仅有 iOS、还有 Android，这些不同端的 UI 之间存在差异。
- 原有后端功能和桌面 Web UI 之间已经存在了较大的耦合。

因为端的差异性存在，服务端的功能要针对端的差异进行适配和裁剪，而服务端的业务功能本身是相对单一的，这就产生了一个矛盾——服务端的单一业务功能和端的差异性诉求之间的矛盾。那么这个问题怎么解决呢？这也是文章的副标题所描述的”Single-purpose Edge Services for UIs and external parties”，引入 BFF，由 BFF 来针对多端差异做适配，这也是目前业界广泛使用的一种模式。

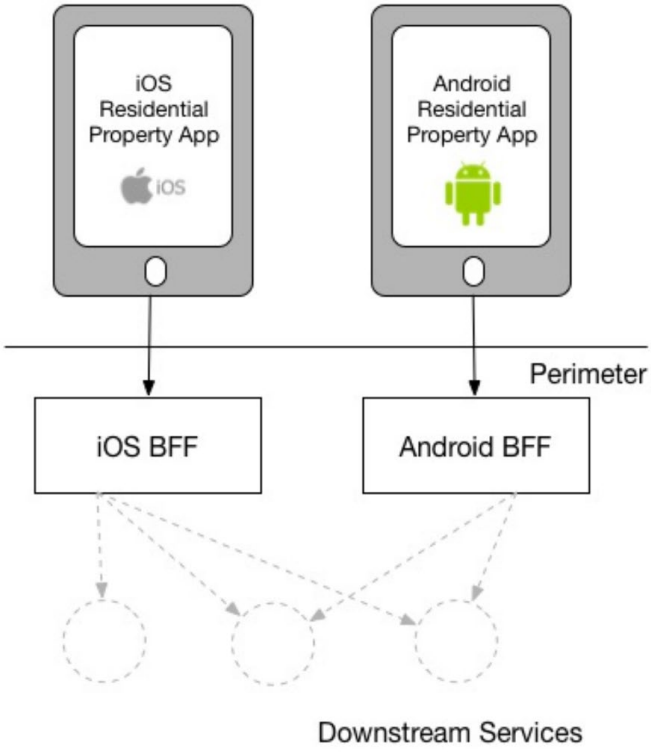


图 1　BFF 示意图

在实际业务的实践中，导致这种端差异性的原因有很多，有技术的原因，也有业务的原因。比如，用户的客户端是 Android 还是 iOS，是大屏还是小屏，是什么版本。再比如，业务属于哪个行业，产品形态是什么，功能投放在什么场景，面向的用户群体是谁等等。这些因素都会带来面向端的功能逻辑的差异性。

在这个问题上，笔者所在团队负责的商品展示业务有一定的发言权，同样的商品业务，在 C 端的展示功能逻辑，深刻受到商品类型、所在行业、交易形态、投放场所、面向群体等因素的影响。同时，面向消费者端的功能频繁迭代的属性，更是加剧并深化了这种矛盾，使其演化成了一种服务端单一稳定与端的差异灵活之间的矛盾，这也是商品展示（商品展示 BFF）业务系统存在的必然性原因。本文主要在美团到店商品展示场景的背景下，介绍面临的一些问题及解决思路。

2. BFF 背景下的核心矛盾

BFF 这层的引入是解决服务端单一稳定与端的差异灵活诉求之间的矛盾，这个矛盾并不是不存在，而是转移了。由原来后端和前端之间的矛盾转移成了 BFF 和前端之间的矛盾。笔者所在团队的主要工作，就是和这种矛盾作斗争。下面以具体的业务场景为例，结合当前的业务特点，说明在 BFF 的生产模式下，我们所面临的具体问题。下图是两个不同行业的团购货架展示模块，这两个模块我们认为是两个商品的展示场景，它们是两套独立定义的产品逻辑，并且会各自迭代。



图 2　展示场景

在业务发展初期，这样的场景不多。BFF 层系统“烟囱式”建设，功能快速开发上线满足业务的诉求，在这样的情况下，这种矛盾表现的不明显。而随着业务发展，行业的开拓，形成了许许多多这样的商品展示功能，矛盾逐渐加剧，主要表现在以下两个方面：

- 业务支撑效率：随着商品展示场景变得越来越多，API 呈爆炸趋势，业务支撑效率和人力成线性关系，系统能力难以支撑业务场景的规模化拓展。
- 系统复杂度高：核心功能持续迭代，内部逻辑充斥着 `if…else…`，代码过程式编写，系统复杂度较高，难以修改和维护。

那么这些问题是怎么产生的呢？这要结合“烟囱式”系统建设的背景和商品展示场景所面临的业务，以及系统特点来进行理解。

特点一：外部依赖多、场景间取数存在差异、用户体验要求高

图例展示了两个不同行业的团购货架模块，这样一个看似不大的模块，后端在 BFF 层要调用 20 个以上的下游服务才能把数据拿全，这是其一。在上面两个不同的场景中，需要的数据源集合存在差异，而且这种差异普遍存在，这是其二，比如足疗团购货架需要的某个数据源，在丽人团购货架上不需要，丽人团购货架需要的某个数据源，足疗团购货架不需要。尽管依赖下游服务多，同时还要保证 C 端的用户体验，这是其三。

这几个特点给技术带来了不小的难题：1）聚合大小难控制，聚合功能是分场景建设？还是统一建设？如果分场景建设，必然存在不同场景重复编写类似聚合逻辑的问题。如果统一建设，那么一个大而全的数据聚合中必然会存在无效的调用。2）聚合逻辑的复杂性控制问题，在这么多的数据源的情况下，不仅要考虑业务逻辑怎么写，还要考虑异步调用的编排，在代码复杂度未能良好控制的情况下，后续聚合的变更修改将会是一个难题。

特点二：展示逻辑多、场景之间存在差异，共性个性逻辑耦合

我们可以明显地识别某一类场景的逻辑是存在共性的，比如团单相关的展示场景。直观可以看出基本上都是展示团单维度的信息，但这只是表象。实际上在模块的生成过程中存在诸多的差异，比如以下两种差异：

- 字段拼接逻辑差异：比如以上图中两个团购货架的团购标题为例，同样是标题，在丽人团购货架中的展示规则是：**[类型] + 团购标题**，而在足疗团购货架的展示规则是：**团购标题**。
- 排序过滤逻辑差异：比如同样是团单列表，A 场景按照销量倒排序，B 场景按照价格排序，不同场景的排序逻辑不同。

诸如此类的**展示逻辑**的差异性还有很多。类似的场景实际上在内部存在很多差异的逻辑，后端如何应对这种差异性是一个难题，下面是最常见的一种写法，通过读取具体的条件字段来做判断实现逻辑路由，如下所示：

```
if(category == "丽人") {
  title = "[" + category + "]" + productTitle;
} else if (category == "足疗") {
  title = productTitle;
}
```

这种方案在功能实现方面没有问题，也能够复用共同的逻辑。但是实际上在场景非常多的情况下，将会有非常多的差异性判断逻辑叠加在一起，功能一直会被持续迭代的情况下，可以想象，系统将会变得越来越复杂，越来越难以修改和维护。

总结：在 BFF 这层，不同商品展示场景存在差异。在业务发展初期，系统通过独立建设的方式支持业务快速试错，在这种情况下，业务差异性带来的问题不明显。而随着业务的不断发展，需要搭建及运营的场景越来越多，呈规模化趋势。此时，业务对技术效率提出了更高的要求。**在这种场景多、场景间存在差异的背景下，如何满足场景拓展效率同时能够控制系统的复杂性，就是我们业务场景中面临的核心问题**。

3. BFF 应用模式分析

目前业界针对此类的解决方案主要有两种模式，一种是后端 BFF 模式，另一种是前端 BFF 模式。

3.1 后端 BFF 模式

后端 BFF 模式指的是 BFF 由后端同学负责，这种模式目前最广泛的实践是基于 GraphQL 搭建的后端 BFF 方案，具体是：后端将展示字段封装成展示服务，通过 GraphQL 编排之后暴露给前端使用。如下图所示：

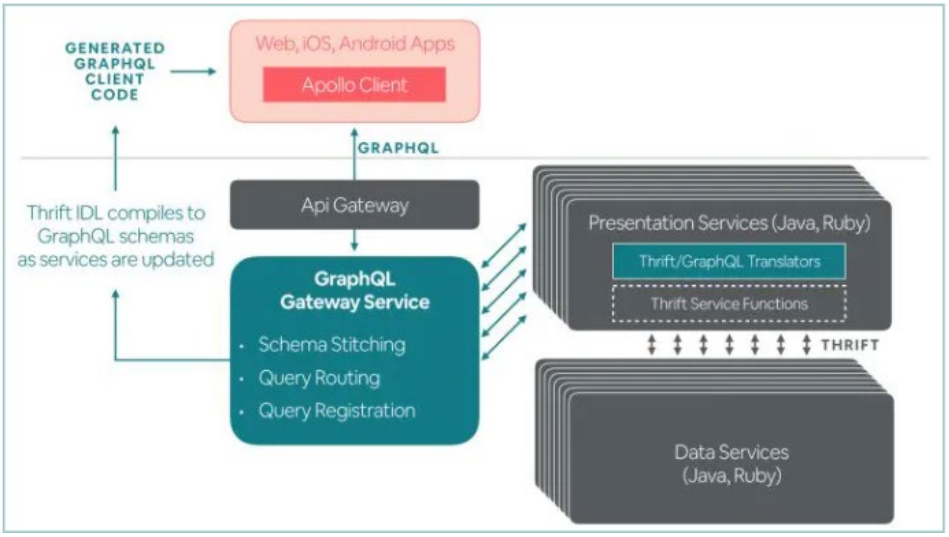


图 3　后端 BFF 模式

这种模式最大的特性和优势是，当展示字段已经存在的情况下，后端不需要关心前端差异性需求，按需查询的能力由 GraphQL 支持。这个特性可以很好地应对不同场景存在展示字段差异性这个问题，前端直接基于 GraphQL 按需查询数据即可，后端不需要变更。同时，借助 GraphQL 的编排和聚合查询能力，后端可以将逻辑分解在不同的展示服务中，因此在一定程度上能够化解 BFF 这层的复杂性。

但是基于这种模式，仍然存在几个问题：展示服务颗粒度问题、数据图划分问题以及字段扩散问题，下图是基于当前模式的具体案例：

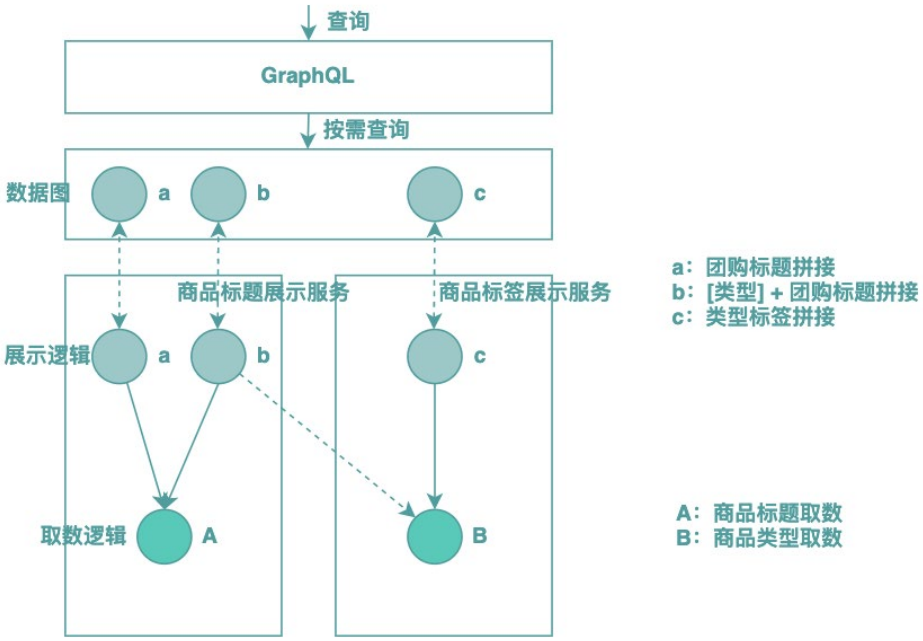


图 4　后端 BFF 模式（案例）

1）展示服务颗粒度设计问题

这种方案要求展示逻辑和取数逻辑封装在一个模块中，形成一个展示服务（Presentation Service），如上图所示。而实际上展示逻辑和取数逻辑是多对多的关系，还是以前文提到的例子说明：

> **背景**：有两个展示服务，分别封装了商品标题和商品标签的查询能力。**情景**：此时 PM 提了一个需求，希望商品在某个场景的标题以“[类型]+ 商品标题”的形式展示，此时商品标题的拼接依赖类型数据，而此时类型数据商品标签展示服务中已经调用了。**问题**：商品标题展示服务自己调用类型数据还是将两个展示服务合并到一起?

以上描述的问题的是展示服务颗粒度把控的问题，我们可以怀疑上述的示例是不是因为展示服务的颗粒度过小? 那么反过来看一看，如果将两个服务合并到一起，那么势必又会存在冗余。这是展示服务设计的难点，**核心原因在于，展示逻辑和取数逻辑本**

身是多对多的关系，结果却被设计放在了一起。

2）数据图划分问题

通过 GraphQL 将多个展示服务的数据聚合到一张图（GraphQL Schema）中，形成一个数据视图，需要数据的时候只要数据在图中，就可以基于 Query 按需查询。那么问题来了，这个图应该怎么组织？是一张图还是多张图？图过大的话，势必带来复杂的数据关系维护问题，图过小则将会降低方案本身的价值。

3）展示服务内部复杂性 + 模型扩散问题

上文提到过一个商品标题的展示存在不同拼接逻辑的情况，在商品展示场景，这种逻辑特别普遍。比如同样是价格，A 行业展示优惠后价格，B 行业展示优惠前价格；同样是标签位置，C 行业展示服务时长，而 D 行业展示商品特性等。那么问题来了，展示模型如何设计？以标题字段为例，是在展示模型上放个 `title` 字段就可以，还是分别放个 `title` 和 `titleWithCategory`？如果是前者那么服务内部必然会存在 `if…else…` 这种逻辑，用于区分 `title` 的拼接方式，这同样会导致展示服务内部的复杂性。如果是多个字段，那么可以想象，展示服务的模型字段也将会不断扩散。

总结：后端 BFF 模式能够在一定程度上化解后端逻辑的复杂性，同时提供一个展示字段的复用机制。但是仍然存在未决问题，如展示服务的颗粒度设计问题，数据图的划分问题，以及展示服务内部的复杂性和字段扩散问题。目前这种模式实践的代表有 Facebook、爱彼迎、eBay、爱奇艺、携程、去哪儿等等。

3.2 前端 BFF 模式

前端 BFF 模式在 Sam Newman 的文章中的”And Autonomy”部分有特别的介绍，指的是 BFF 本身由前端团队自己负责，如下示意图所示：

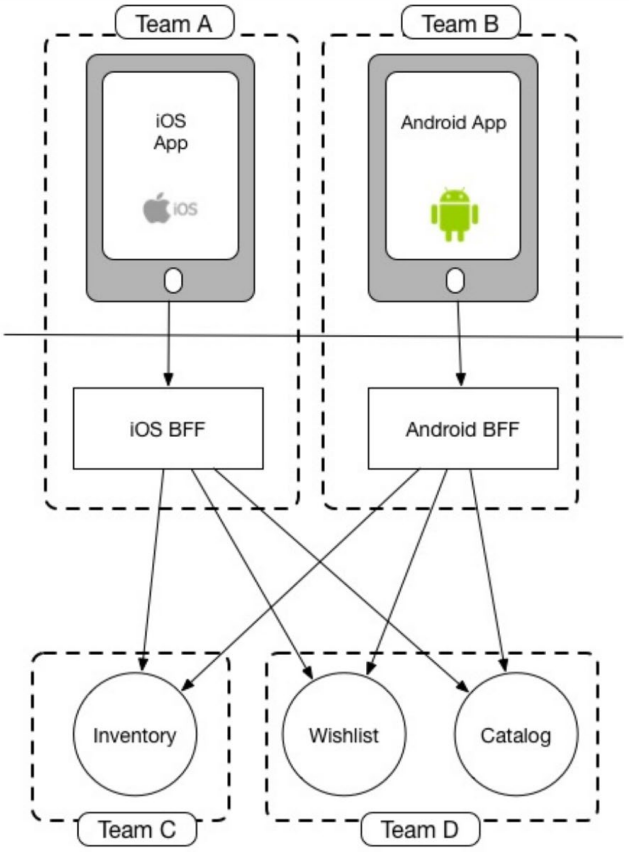


图 5　前端 BFF 模式

这种模式的理念是，本来能一个团队交付的需求，没必要拆成两个团队，两个团队本身带来较大的沟通协作成本。本质上，也是一种将“敌我矛盾”转化为“人民内部矛盾”的思路。前端完全接手 BFF 的开发工作，实现数据查询的自给自足，大大减少了前后端的协作成本。但是这种模式没有提到我们关心的一些核心问题，如：复杂性如何应对、差异性如何应对、展示模型如何设计等等问题。除此之外，这种模式也存在一些前提条件及弊端，比如较为完备的前端基础设施；前端不仅仅需要关心渲染、还需要了解业务逻辑等。

总结：前端 BFF 模式通过前端自主查询和使用数据，从而达到降低跨团队协作的成本，提升 BFF 研发效率的效果。目前这种模式的实践代表是阿里巴巴。

4. 基于 GraphQL 及元数据的信息聚合架构设计

4.1 整体思路

通过对后端 BFF 和前端 BFF 两种模式的分析，我们最终选择后端 BFF 模式，前端 BFF 这个方案对目前的研发模式影响较大，不仅需要大量的前端资源，而且需要建设完善的前端基础设施，方案实施成本比较高昂。

前文提到的后端 GraphQL BFF 模式代入我们的具体场景虽然存在一些问题，但是总体有非常大的参考价值，比如展示字段的复用思路、数据的按需查询思路等等。在商品展示场景中，**有 80% 的工作集中在数据的聚合和集成部分**，并且这部分具有很强的复用价值，因此信息的查询和聚合是我们面临的主要矛盾。因此，我们的思路是：**基于 GraphQL+ 后端 BFF 方案改进，实现取数逻辑和展示逻辑的可沉淀、可组合、可复用**，整体架构如下示意图所示：

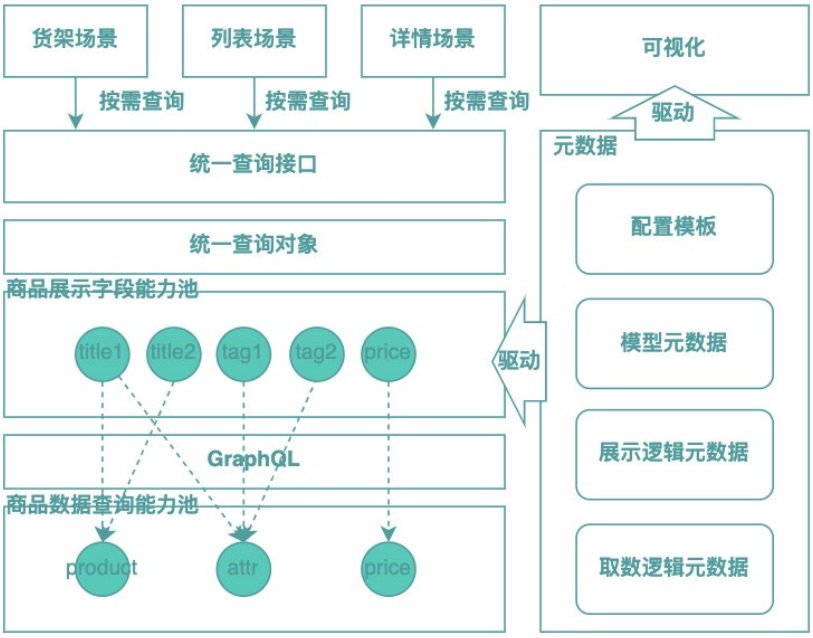


图 6　基于 GraphQL BFF 的改进思路

从上图可看出，与传统 GraphQL BFF 方案最大的差别在于我们将 GraphQL 下放

至数据聚合部分，由于数据来源于商品领域，领域是相对稳定的，因此数据图规模可控且相对稳定。除此之外，整体架构的核心设计还包括以下三个方面：1）取数展示分离；2）查询模型归一；3）元数据驱动架构。

我们通过取数展示分离解决展示服务颗粒度问题，同时使得展示逻辑和取数逻辑可沉淀、可复用；通过查询模型归一化设计解决展示字段扩散的问题；通过元数据驱动架构实现能力的可视化，业务组件编排执行的自动化，这能够让业务开发同学聚焦于业务逻辑的本身。下面将针对这三个部分逐一展开介绍。

4.2 核心设计

4.2.1 取数展示分离

上文提到，在商品展示场景中，展示逻辑和取数逻辑是多对多的关系，而传统的基于 GraphQL 的后端 BFF 实践方案把它们封装在一起，这是导致展示服务颗粒度难以设计的根本原因。思考一下取数逻辑和展示逻辑的关注点是什么？取数逻辑关注怎么查询和聚合数据，而展示逻辑关注怎么加工生成需要的展示字段，它们的关注点不一样，放在一起也会增加展示服务的复杂性。因此，我们的思路是将取数逻辑和展示逻辑分离开来，单独封装成逻辑单元，分别叫取数单元和展示单元。在取数展示分离之后，GraphQL 也随之下沉，用于实现数据的按需聚合，如下图所示：

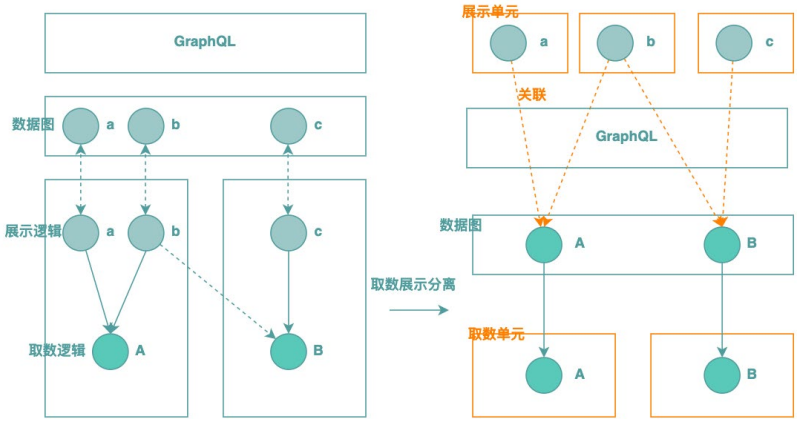


图 7　取数展示分离 + 元数据描述

那么取数和展示逻辑的封装颗粒度是怎么样的呢？不能太小也不能太大，在颗粒度的设计上，我们有两个核心考量：1）**复用**，展示逻辑和取数逻辑在商品展示场景中，都是可以被复用的资产，我们希望它们能沉淀下来，被单独按需使用；2）**简单**，保持简单，这样容易修改和维护。基于这两点考虑，颗粒度的定义如下：

- **取数单元**：尽量只封装 1 个外部数据源，同时负责对外部数据源返回的模型进行简化，这部分生成的模型我们称之为取数模型。
- **展示单元**：尽量只封装 1 个展示字段的加工逻辑。

分开的好处是简单且可被组合使用，那么具体如何实现组合使用呢？我们的思路是通过元数据来描述它们之间的关系，基于元数据由统一的执行框架来关联运行，具体设计下文会展开介绍。通过取数和展示的分离，元数据的关联和运行时的组合调用，可以保持逻辑单元的简单，同时又满足复用诉求，这也很好地解决了传统方案中存在的**展示服务的颗粒度问题**。

4.2.2 查询模型归一

展示单元的加工结果通过什么样的接口透出呢？接下来，我们介绍一下查询接口设计的问题。

1）查询接口设计的难点

常见查询接口的设计模式有以下两种：

- **强类型模式**：强类型模式指的是查询接口返回的是 POJO 对象，每一个查询结果对应 POJO 中的一个明确的具有特定业务含义的字段。
- **弱类型模式**：弱类型模式指的是查询结果以 K-V 或 JSON 模式返回，没有明确的静态字段。

以上两种模式在业界都有广泛应用，且它们都有明确的优缺点。强类型模式对开发者友好，但是业务是不断迭代的，与此同时，系统沉淀的展示单元会不断丰富，在这样的情况下，接口返回的 DTO 中的字段将会愈来愈多，每次新功能的支持，都要伴随着接口查询模型的修改，JAR 版本的升级。而 JAR 的升级涉及数据提供方和数据消

费两方，存在明显效率问题。另外，可以想象，查询模型的不断迭代，最终将会包括成百上千个字段，难以维护。

而弱类型模式恰好可以弥补这一缺点，但是弱类型模式对于开发者来说非常不友好，接口查询模型中有哪些查询结果对于开发者来说在开发的过程中完全没有感觉，但是程序员的天性就是喜欢通过代码去理解逻辑，而非配置和文档。其实，这两种接口设计模式都存在着一个共性问题——缺少抽象，下面两节，我们将介绍在接口返回的查询模型设计方面的抽象思路及框架能力支持。

2）查询模型归一化设计

回到商品展示场景中，一个展示字段有多种不同的实现，如商品标题的两种不同实现方式：1）商品标题；2）[类目]+ 商品标题。商品标题和这两种展示逻辑的关系本质上是一种抽象 – 具体的关系。识别这个关键点，思路就明了了，我们的思路是对查询模型做抽象。查询模型上都是抽象的展示字段，一个展示字段对应多个展示单元，如下图所示：

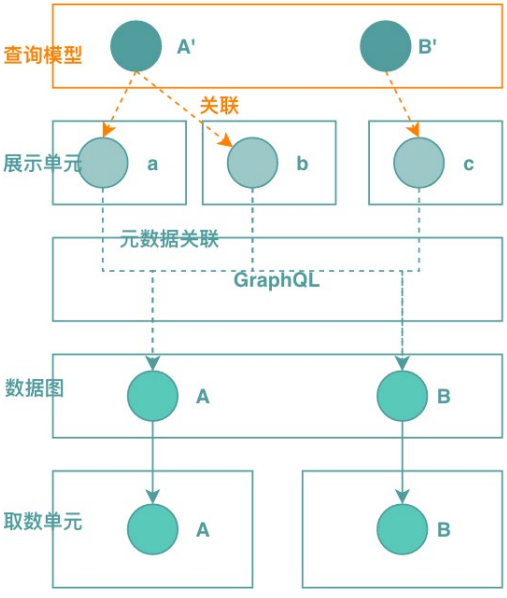


图 8　查询模型归一化 + 元数据描述

在实现层面同样基于元数据描述展示字段和展示单元之间的关系，基于以上的设计思路，可以在一定程度上减缓模型的扩散，但是还不能避免扩展。比如除了价格、库存、销量等每个商品都有的标准属性之外，不同的商品类型一般还会有这个商品特有的属性。比如密室主题拼场商品才有“几人拼”这样的描述属性，这种字段本身抽象的意义不大，且放在商品查询模型中作为一个单独的字段会导致模型扩张，针对这类问题，我们的解决思路是引入扩展属性，扩展属性专门承载这类非标准的字段。通过标准字段 + 扩展属性的方式建立查询模型，能够较好地解决**字段扩散**的问题。

4.2.3 元数据驱动架构

到目前为止，我们定义了如何分解**业务逻辑单元**以及如何设计**查询模型**，并提到用元数据描述它们之间的关系。基于以上定义实现的业务逻辑及模型，都具备很强的复用价值，可以作为业务资产沉淀下来。那么，为什么用元数据描述业务功能及模型之间的关系呢?

我们引入元数据描述主要有两个目的：1）代码逻辑的自动编排，通过元数据描述业务逻辑之间的关联关系，运行时可以自动基于元数据实现逻辑之间的关联执行，从而可以消除大量的人工逻辑编排代码；2）业务功能的可视化，元数据本身描述了业务逻辑所提供的功能，如下面两个示例：

> 团单基础售价字符串展示，例：30 元。 团单市场价展示字段，例：100 元。

这些元数据上报到系统中，可以用于展示当前系统所提供的功能。通过元数据描述组件及组件之间关联关系，通过框架解析元数据自动进行业务组件的调用执行，形成了如下的元数据架构：

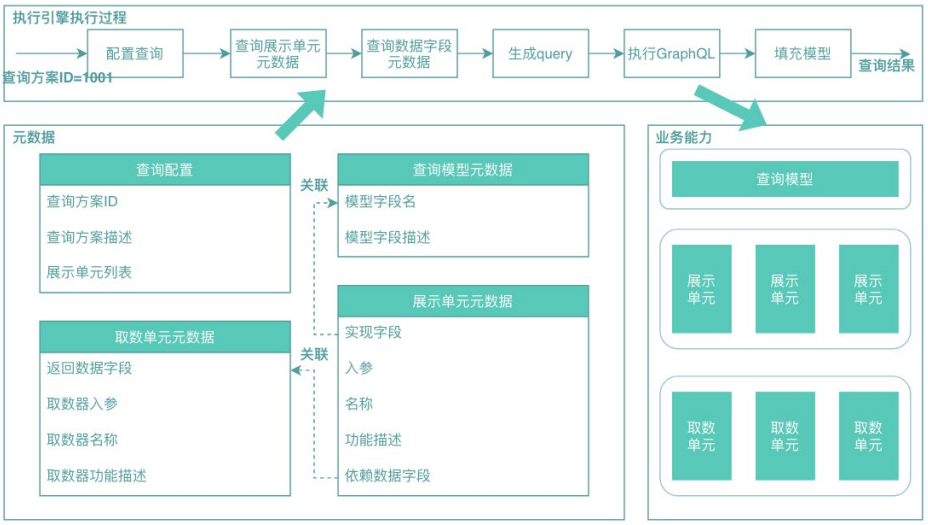


图 9　元数据驱动架构

整体架构由三个核心部分组成：

- 业务能力：标准的业务逻辑单元，包括取数单元、展示单元和查询模型，这些都是关键的可复用资产。
- 元数据：描述业务功能（如：展示单元、取数单元）以及业务功能之间的关联关系，比如展示单元依赖的数据，展示单元映射的展示字段等。
- 执行引擎：负责消费元数据，并基于元数据对业务逻辑进行调度和执行。

通过以上三个部分有机的组合在一起，形成了一个元数据驱动风格的架构。

5. 针对 GraphQL 的优化实践

5.1 使用简化

1）GraphQL 直接使用问题

引入 GraphQL，会引入一些额外的复杂性，比如会涉及到 GraphQL 带来的一些概念如：Schema、RuntimeWiring，下面是基于 GraphQL 原生 Java 框架的开发过程：

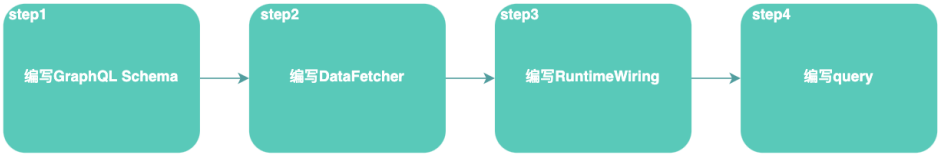


图 10　原生 GraphQL 使用流程

这些概念对于未接触过 GraphQL 的同学来说，增加了学习和理解的成本，而这些概念和业务领域通常没有什么关系。而我们仅仅希望使用 GraphQL 的按需查询特性，却被 GraphQL 本身拖累了，业务开发同学的关注点应该聚焦在业务逻辑本身才对，这个问题如何解决呢?

著名计算机科学家 David Wheeler 说了一句名言，”All problems in computer science can be solved by another level of indirection”。没有加一层解决不了的问题，本质上是需要有人来对这事负责，因此我们在原生 GraphQL 之上增加了一层执行引擎层来解决这些问题，目标是屏蔽 GraphQL 的复杂性，让开发人员只需要关注业务逻辑。

2）取数接口标准化

首先要简化数据的接入，原生的 `DataFetcher` 和 `DataLoader` 都是处在一个比较高的抽象层次，缺少业务语义，而在查询场景，我们能够归纳出，所有的查询都属于以下三种模式：

- **1 查 1**：根据一个条件查询一个结果。
- **1 查 N**：根据一个条件查询多个结果。
- **N 查 N**：一查一或一查多的批量版本。

由此，我们对查询接口进行了标准化，业务开发同学基于场景判断是那种，按需选择使用即可，取数接口标准化设计如下：

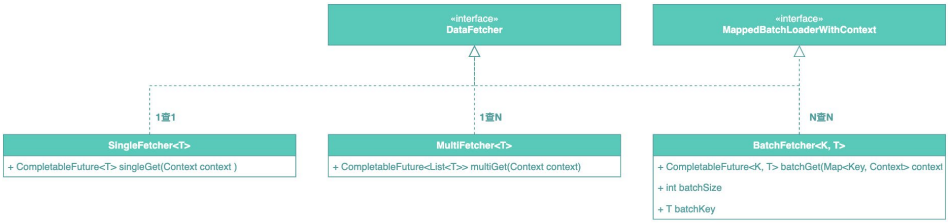


图 11　查询接口标准化

业务开发同学按需选择所需要使用的取数器，通过泛型指定结果类型，1 查 1 和 1 查 N 比较简单，N 查 N 我们对其定义为批量查询接口，用于满足”N+1”的场景，其中 `batchSize` 字段用于指定分片大小，`batchKey` 用于指定查询 Key，业务开发只需要指定参数，其他的框架会自动处理。除此之外，我们还约束了返回结果必须是 `CompleteFuture`，用于满足聚合查询的全链路异步化。

3）聚合编排自动化

取数接口标准化使得数据源的语义更清晰，开发过程按需选择即可，简化了业务的开发。但是此时业务开发同学写好 `Fetcher` 之后，还需要去另一个地方去写 `Schema`，而且写完 `Schema` 还要再写 `Schema` 和 `Fetcher` 的映射关系，业务开发更享受写代码的过程，不太愿意写完代码还要去另外一个地方取配置，并且同时维护代码和对应配置也提高了出错的可能性，能否将这些冗杂的步骤移除掉?

`Schema` 和 `RuntimeWiring` 本质上是想描述某些信息，如果这些信息换一种方式描述是不是也可以，我们的优化思路是：在业务开发过程中标记注解，通过注解标注的元数据描述这些信息，其他的事情交给框架来做。解决思路示意图如下：

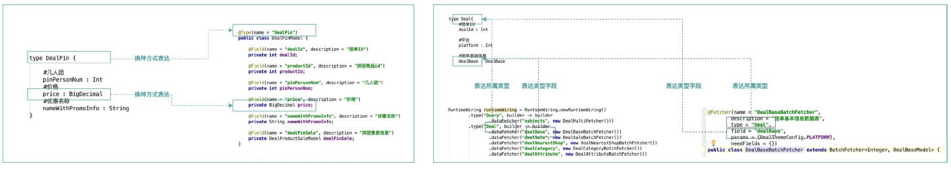


图 12　注解元数据描述 Schema 和 RuntimeWiring

5.2 性能优化

5.2.1 GraphQL 性能问题

虽然 GraphQL 已经开源了，但是 Facebook 只开源了相关标准，并没有给出解决方案。GraphQL-Java 框架是由社区贡献的，基于开源的 GraphQL-Java 作为按需查询引擎的方案，我们发现了 GraphQL 应用方面的一些问题，这些问题有部分是由于使用姿势不当所导致的，也有部分是 GraphQL 本身实现的问题，比如我们遇到的几个典型的问题：

- 耗 CPU 的查询解析，包括 `Schema` 的解析和 `Query` 的解析。
- 当查询模型比较复杂特别是存在大列表时候的延时问题。
- 基于反射的模型转换 CPU 消耗问题。
- `DataLoader` 的层级调度问题。

于是，我们对使用方式和框架做了一些优化与改造，以解决上面列举的问题。本章着重介绍我们在 GraphQL-Java 方面的优化和改造思路。

5.2.2 GraphQL 编译优化

1）GraphQL 语言原理概述

GraphQL 是一种查询语言，目的是基于直观和灵活的语法构建客户端应用程序，用于描述其数据需求和交互。GraphQL 属于一种领域特定语言（DSL），而我们所使用的 GraphQL-Java 客户端在语言编译层面是基于 ANTLR 4 实现的，ANTLR 4 是一种基于 Java 编写的语言定义和识别工具，ANTLR 是一种元语言（Meta-Language），它们的关系如下：

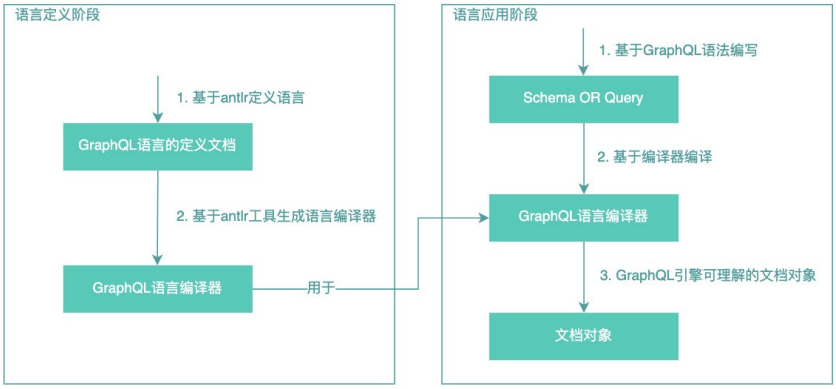


图 13　GraphQL 语言基本原理示意图

GraphQL 执行引擎所接受的 `Schema` 及 `Query` 都是基于 GraphQL 定义的语言所表达的内容，GraphQL 执行引擎不能直接理解 GraphQL，在执行之前必须由 GraphQL 编译器翻译成 GraphQL 执行引擎可理解的文档对象。而 GraphQL 编译器是基于 Java 的，经验表明在大流量场景实时解释的情况下，这部分代码将会成为 CPU 热点，而且还占用响应延迟，`Schema` 或 `Query` 越复杂，性能损耗越明显。

2）Schema 及 Query 编译缓存

`Schema` 表达的是数据视图和取数模型同构，相对稳定，个数也不多，在我们的业务场景一个服务也就一个。因此，我们的做法是在启动的时候就将基于 `Schema` 构造的 GraphQL 执行引擎构造好，作为单例缓存下来，对于 `Query` 来说，每个场景的 `Query` 有些差异，因此 `Query` 的解析结果不能作为单例，我们的做法是实现 `PreparsedDocumentProvider` 接口，基于 `Query` 作为 Key 将 `Query` 编译结果缓存下来。如下图所示：

```
private ConcurrentMap<String, PreparsedDocumentEntry> queryCacheEntry = Maps.newConcurrentMap();

@Override
public PreparsedDocumentEntry getDocument(ExecutionInput executionInput, Function<ExecutionInput, PreparsedDocumentEntry> computeFunction) {
    if (queryCacheEntry.get(executionInput.getQuery()) == null) {
        queryCacheEntry.putIfAbsent(executionInput.getQuery(), computeFunction.apply(executionInput));
    }
    return queryCacheEntry.get(executionInput.getQuery());
}
```

图 14　Query 缓存实现示意图

5.2.3 GraphQL 执行引擎优化

1) GraphQL 执行机制及问题

我们先一起了解一下 GraphQL-Java 执行引擎的运行机制是怎么样的。假设在执行策略上我们选取的是 `AsyncExecutionStrategy`，来看看 GraphQL 执行引擎的执行过程：

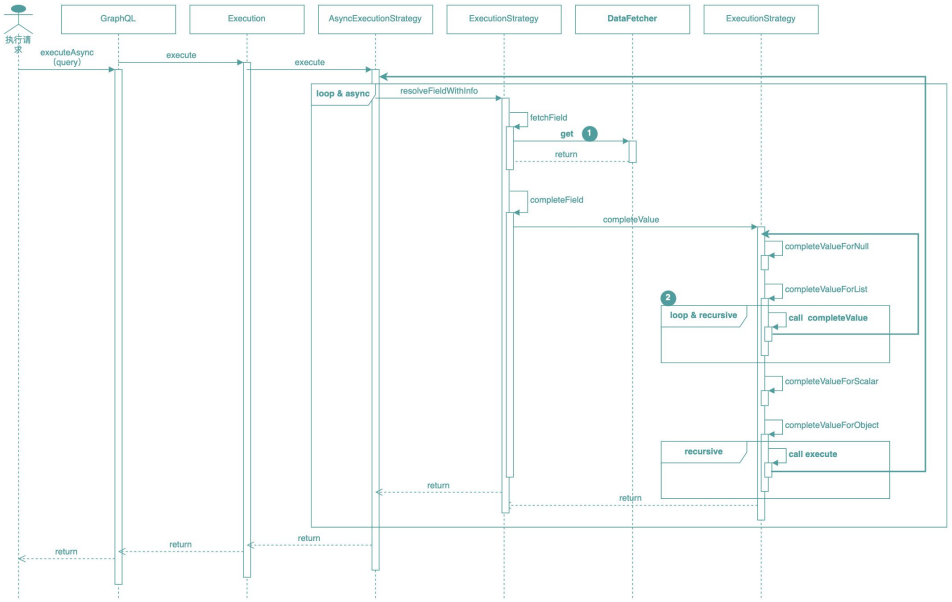


图 15　GraphQL 执行引擎执行过程

以上时序图做了些简化，去除了一些与重点无关的信息，`AsyncExecutionStrategy` 的 `execute` 方法是对象执行策略的异步化模式实现，是查询执行的起点，也是根节点查询的入口，`AsyncExecutionStrategy` 对对象的多个字段的查询逻辑，采取的是循环 + 异步化的实现方式，我们从 `AsyncExecutionStrategy` 的 `execute` 方法触发，理解 GraphQL 查询过程如下：

- 调用当前字段所绑定的 `DataFetcher` 的 `get` 方法，如果字段没有绑定 `DataFetcher`，则通过默认的 `PropertyDataFetcher` 查询字段，`PropertyDataFetcher` 的实现是基于反射从源对象中读取查询字段。

- 将从 `DataFetcher` 查询得到结果包装成 `CompletableFuture`，如果结果本身是 `CompletableFuture`，那么不会包装。
- 结果 `CompletableFuture` 完成之后，调用 `completeValue`，基于结果类型分别处理。
  - 如果查询结果是列表类型，那么会对列表类型进行遍历，针对每个元素在递归执行 `completeValue`。
  - 如果结果类型是对象类型，那么会对对象执行 `execute`，又回到了起点，也就是 `AsyncExecutionStrategy` 的 `execute`。

以上是 GraphQL 的执行过程，这个过程有什么问题呢？下面基于图上的标记顺序一起看看 GraphQL 在我们的业务场景中应用和实践所遇到的问题，这些问题不代表在其他场景也是问题，仅供参考：

问题 1：`PropertyDataFetcher` CPU 热点问题，`PropertyDataFetcher` 在整个查询过程中属于热点代码，而其本身的实现也有一些优化空间，在运行时 `PropertyDataFetcher` 的执行会成为 CPU 热点。（具体问题可参考 GitHub 上的 commit 和 Conversion：https://github.com/graphql-java/graphql-java/pull/1815）



图 16　PropertyDataFetcher 成为 CPU 热点

问题 2：列表的计算耗时问题，列表计算是循环的，对于查询结果中存在大列表的场景，此时循环会造成整体查询明显的延迟。我们举个具体的例子，假设查询结果中存在一个列表大小是 1000，每个元素的处理是 0.01ms，那么总体耗时就是 10ms，基于 GraphQL 的查机制，这个 10ms 会阻塞整个链路。

2）类型转换优化

通过 GraphQL 查询引擎拿到的 GraphQL 模型，和业务实现的 `DataFetcher` 返回的取数模型是同构，但是所有字段的类型都会被转换成 GraphQL 内部类型。`ProperyDataFetcher` 之所以会成为 CPU 热点，问题就在于这个模型转换过程，业务定义的模型到 GraphQL 类型模型转换过程示意图如下图所示：

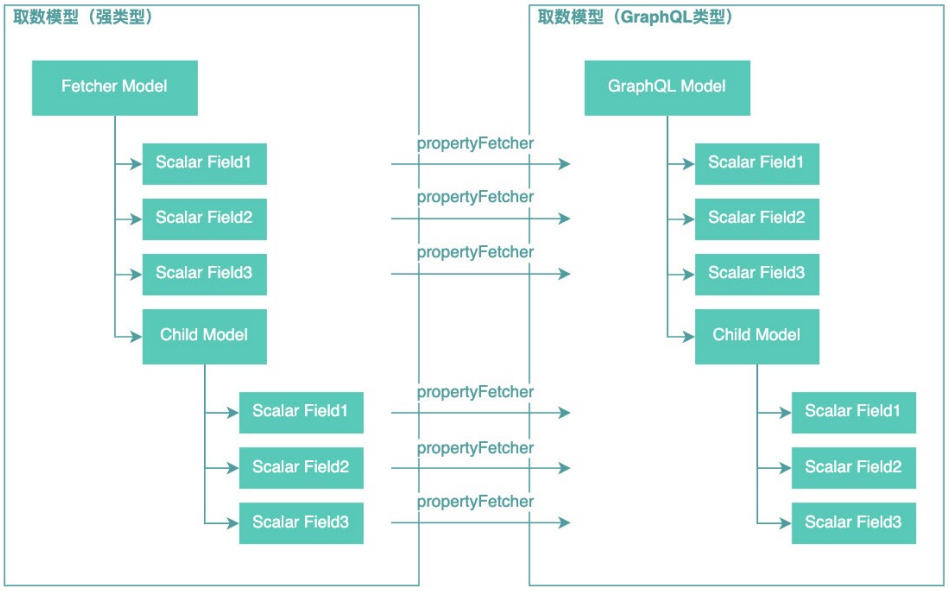


图 17　业务模型到 GraphQL 模型转换示意图

当查询结果模型中的字段非常多的时候，比如上万个，意味着每次查询有上万次的 `PropertyDataFetcher` 操作，实际就反映到了 CPU 热点问题上，这个问题我们的解决思路是保持原有业务模型不变，将非 `PropertyDataFetcher` 查询的结果反过来填充到业务模型上。如下示意图所示：

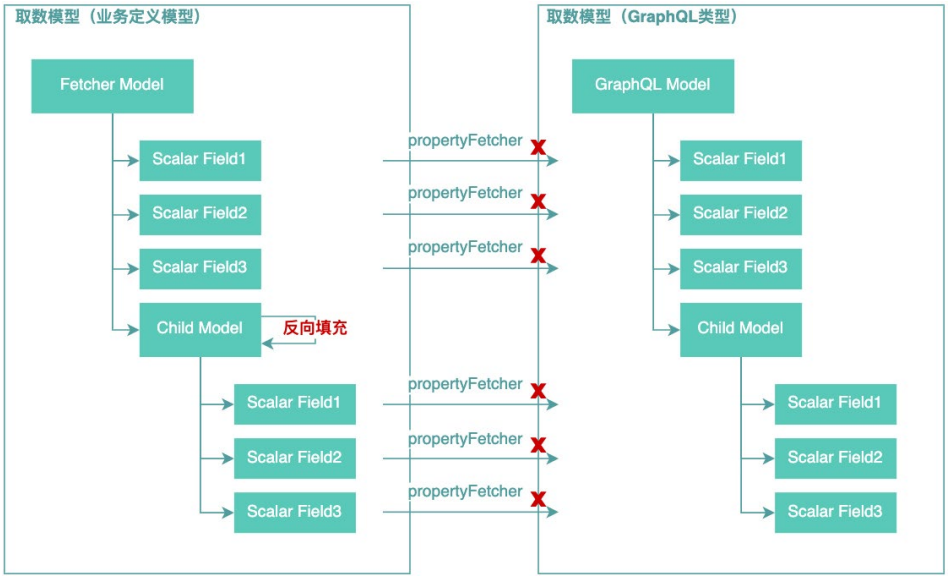


图 18　查询结果模型反向填充示意图

基于这个思路，我们通过 GraphQL 执行引擎拿到的结果就是业务 `Fetcher` 返回的对象模型，这样不仅仅解决了因字段反射转换带来的 CPU 热点问题，同时对于业务开发来说增加了友好性。因为 GraphQL 模型类似 JSON 模型，这种模型是缺少业务类型的，业务开发直接使用起来非常麻烦。以上优化在一个场景上试点测试，结果显示该场景的平均响应时间缩短 1.457ms，平均 99 线缩短 5.82ms，平均 CPU 利用率降低约 12%。

3）列表计算优化

当列表元素比较多的时候，默认的单线程遍历列表元素计算的方式所带来的延迟消耗非常明显，对于响应时间比较敏感的场景这个延迟优化很有必要。针对这个问题我们的解决思路是充分利用 CPU 多核心计算的能力，将列表拆分成任务，通过多线程并行执行，实现机制如下：

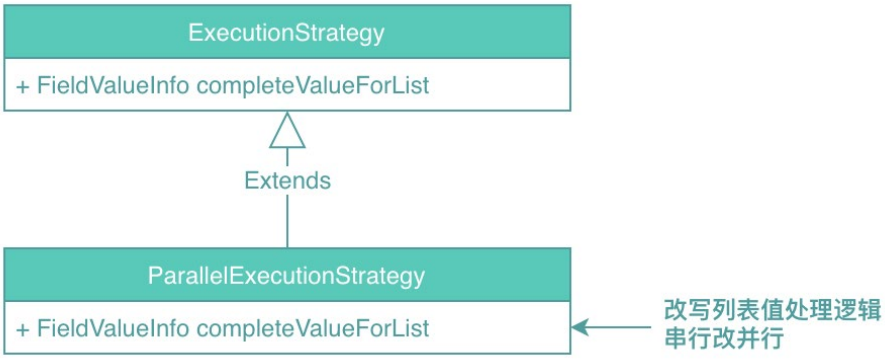


图 19　列表遍历多核计算思路

5.2.4 GraphQL-DataLoader 调度优化

1) DataLoader 基本原理

先简单介绍一下 DataLoader 的基本原理，DataLoader 有两个方法，一个是 `load`，一个是 `dispatch`，在解决 N+1 问题的场景中，DataLoader 是这么用的：

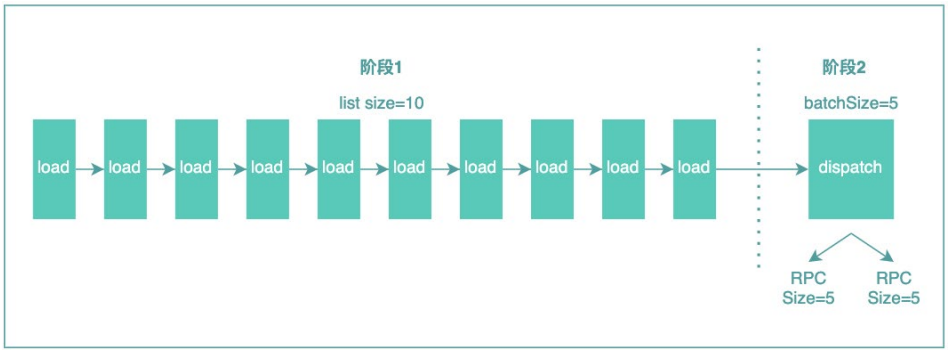


图 20　DataLoader 基本原理

整体分为 2 个阶段，第一个阶段调用 `load`，调用 N 次，第二个阶段调用 `dispatch`，调用 `dispatch` 的时候会真正的执行数据查询，从而达到批量查询 + 分片的效果。

2）DataLoader 调度问题

GraphQL-Java 对 DataLoader 的集成支持的实现在 `FieldLevelTrackingApproach` 中，`FieldLevelTrackingApproach` 的实现会存在怎样的问题呢？下面基于一张图表达原生 DataLoader 调度机制所产生的问题：

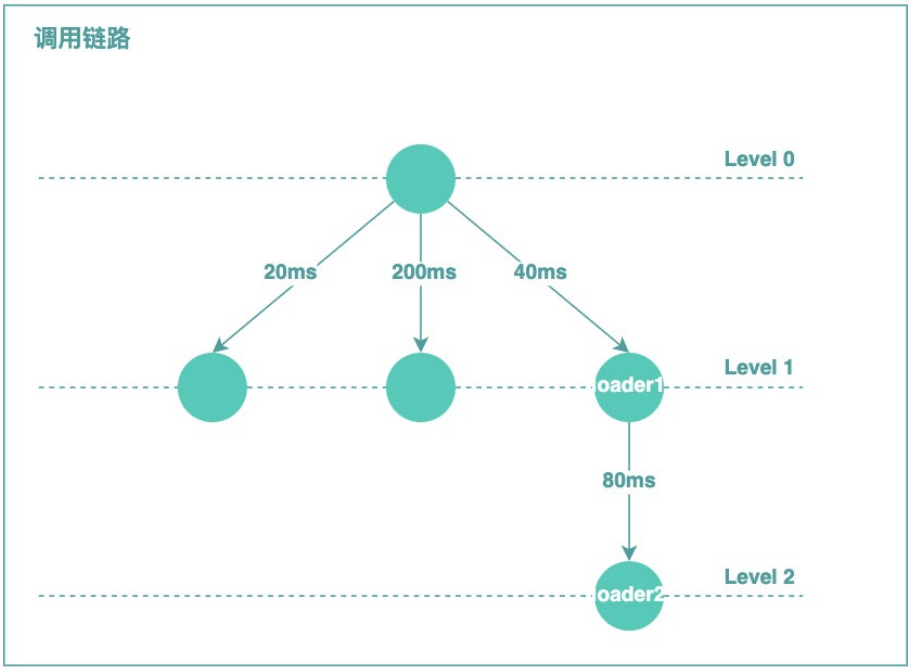


图 21　GraphQL-Java 对 DataLoader 调度存在的问题

问题很明显，基于 `FieldLevelTrackingApproach` 的实现，下一层级的 `DataLoader` 的 `dispatch` 是需要等到本层级的结果都回来之后才发出。基于这样的实现，查询总耗时的计算公式等于：TOTAL = MAX（Level 1 Latency）+ MAX（Level 2 Latency）+ MAX（Level 3 Latency）+ …，**总查询耗时等于每层耗时最大的值加起来，而实际上如果链路编排由业务开发同学自己来写的话，理论上的效果是总耗时等于所有链路最长的那个链路所耗的时间**，这个才是合理的。而 `FieldLevelTrackingApproach` 的实现所表现出来的结果是反常识的，至于为什么这么实现，目前我

们理解可能是设计者基于简单和通用方面的考虑。

问题在于以上的实现在有些业务场景下是不能接受的，比如我们的列表场景的响应时间约束一共也就不到 100ms，其中几十 ms 是因为这个原因搭进去的。针对这个问题的解决思路，一种方式是对于响应时间要求特别高的场景独立编排，不采用 GraphQL；另一种方式是在 GraphQL 层面解决这个问题，保持架构的统一性。接下来，介绍一下我们是如何扩展 GraphQL-Java 执行引擎来解决这个问题的。

3）DataLoader 调度优化

针对 DataLoader 调度的性能问题，**我们的解决思路是在最后一次调用某个 `DataLoader` 的 `load` 之后，立即调用 `dispatch` 方法发出查询请求**，问题是我们怎么知道哪一次的 load 是最后一次 load 呢？这个问题也是解决 DataLoader 调度问题的难点，以下举个例子来解释我们的解决思路：

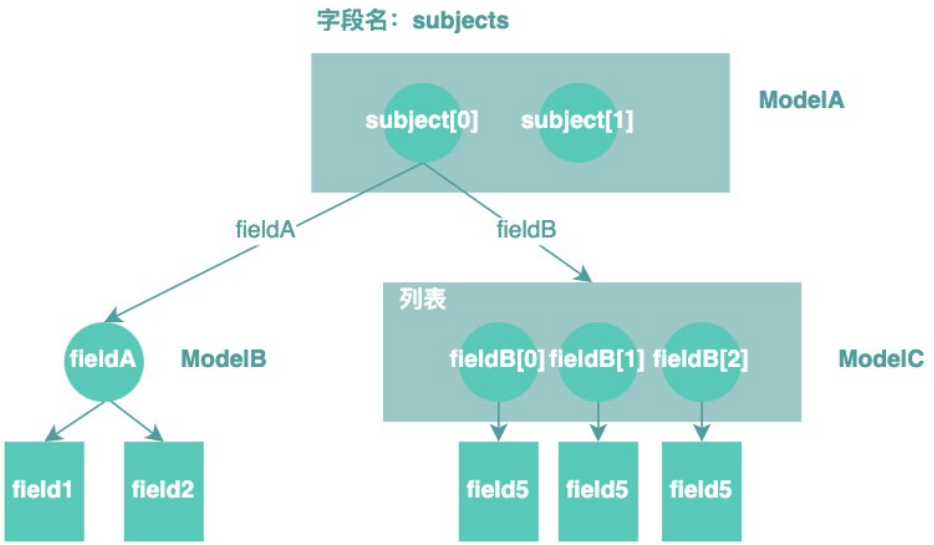


图 22　查询对象结果示意图

假设我们查询到的模型结构如下：根节点是 `Query` 下的字段，字段名叫 `subjects`，`subject` 引用的是个列表，`subject` 下有两个元素，都是 `ModelA` 的对象实例，`ModelA` 有两个字段，`fieldA` 和 `fieldB`，`subjects[0]` 的 `fieldA` 关联是

`ModelB` 的一个实例，`subjects[0]` 的 `fieldB` 关联多个 `ModelC` 实例。

为了方便理解，我们定义一些概念，字段、字段实例、字段实例执行完、字段实例值大小、字段实例值对象执行大小、字段实例值对象执行完等等：

- **字段**：具有唯一路径，是静态的，和运行时对象大小没有关系，如：`subjects` 和 `subjects/fieldA`。
- **字段实例**：字段的实例，具有唯一路径，是动态的，跟运行时对象大小有关系，如：`subjects[0]/fieldA` 和 `subjects[1]/fieldA` 是字段 `subjects/fieldA` 的实例。
- **字段实例执行完**：字段实例关联的对象实例都被 GraphQL 执行完了。
- **字段实例值大小**：字段实例引用对象实例的个数，如以上示例，`subjects[0]/fieldA` 字段实例值大小是 1，`subjects[0]/fieldB` 字段实例值大小是 3。

除了以上定义之外，我们的业务场景还满足以下条件：

- 只有 1 个根节点，且根节点是列表。
- `DataLoader` 一定属于某个字段，某个字段下的 `DataLoader` 应该被执行次数等于其下的对象实例个数。

基于以上信息，我们可以得出以下问题分析：

- 在执行字段实例的时候，我们可以知道当前字段实例的大小，字段实例的大小等于字段关联 `DataLoader` 在当前实例下需要执行 `load` 的次数，因此在执行 `load` 之后，我们可以知道当前对象实例是否是其所在字段实例的最后一个对象。
- 一个对象的实例可能会挂在不同的字段实例下，所以仅当当前对象实例是其所在字段实例的最后一个对象实例的时候，不代表当前对象实例是所有对象实例中的最后一个，当且仅当对象实例所在节点实例是节点的最后一个实例的时候才成立。
- 我们可从字段实例大小推算字段实例的个数，比如我们知道 `subjects` 的

大小是2，那么就知道`subjects`字段有两个字段实例`subjects[0]`和`subjects[1]`，也就知道字段subjects/fieldA有两个实例，`subjects[0]/fieldA`和`subjects[1]/fieldA`，因此我们从根节点可以往下推断出某个字段实例是否执行完。

通过以上分析，我们可以得出，一个对象执行完的条件是其所在的字段实例以及其所在的字段所有的父亲字段实例都执行完，且当前执行的对象实例是其所在字段实例的最后一个对象实例的时候。基于这个判断逻辑，我们的实现方案是在每次调用完`DataFetcher`的时候，判断是否需要发起`dispatch`，如果是则发起。另外，以上时机和条件存在漏发`dispatch`的问题，有个特殊情况，当当前对象实例不是最后一个，但是剩下的对象大小都为0的时候，那么就永远不会触发当前对象关联的`DataLoader`的`load`了，所以在对象大小为0的时候，需要额外再判断一次。

根据以上逻辑分析，我们实现了`DataLoader`调用链路的最优化，达到理论最优的效果。

6. 新架构对研发模式的影响

生产力决定生产关系，元数据驱动信息聚合架构是展示场景搭建的核心生产力，而业务开发模式和过程是生产关系，因此也会随之改变。下面我们将会从开发模式和流程两个角度来介绍新架构对研发带来的影响。

6.1 聚焦业务的开发模式

新架构提供了一套基于业务抽象出的标准化代码分解约束。以前开发同学对系统的理解很可能就是“查一查服务，把数据粘在一起”，而现在，研发同学对于业务的理解及代码分解思路将会是一致的。比如展示单元代表的是展示逻辑，取数单元代表的是取数逻辑。同时，很多冗杂且容易出错的逻辑已经被框架屏蔽掉了，研发同学能够有更多的精力聚焦于业务逻辑本身，比如：业务数据的理解和封装，展示逻辑的理解和编写，以及查询模型的抽象和建设。如下示意图所示：

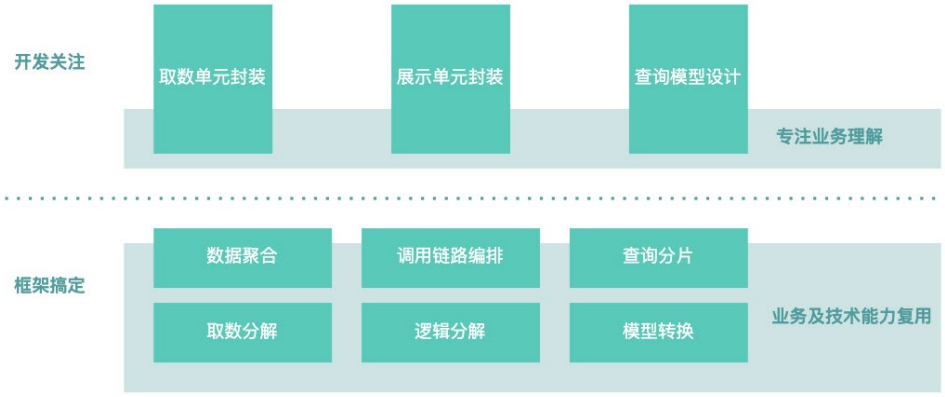


图 23　业务开发聚焦业务本身

6.2 研发流程升级

新架构不仅仅影响了研发的代码编写，同时也影响着研发流程的改进，基于元数据架构实现的可视化及配置化能力，现有研发流程和之前研发流程相比有了明显的区别，如下图所示：

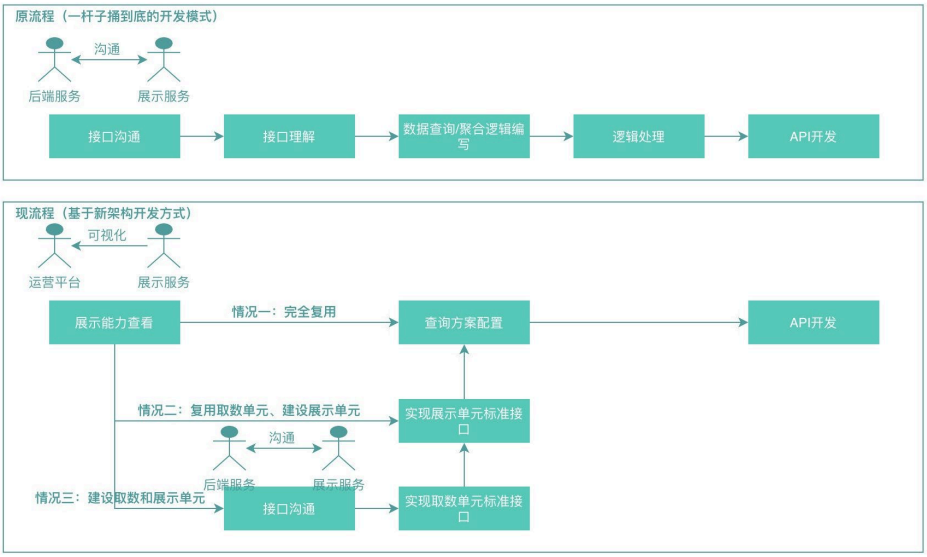


图 24　基于开发框架搭建展示场景前后研发流程对比

以前是“一杆子捅到底”的开发模式，每个展示场景的搭建需要经历过从接口的沟通到 API 的开发整个过程，基于新架构之后，系统自动具备多层复用及可视化、配置化能力。

情况一：这是最好的情况，此时取数功能和展示功能都已经被沉淀下来，研发同学需要做的只是创建查询方案，基于运营平台按需选择需要的展示单元，拿着查询方案 ID 基于查询接口就可以查到需要的展示信息了，可视化、配置化界面如下示意图所示：

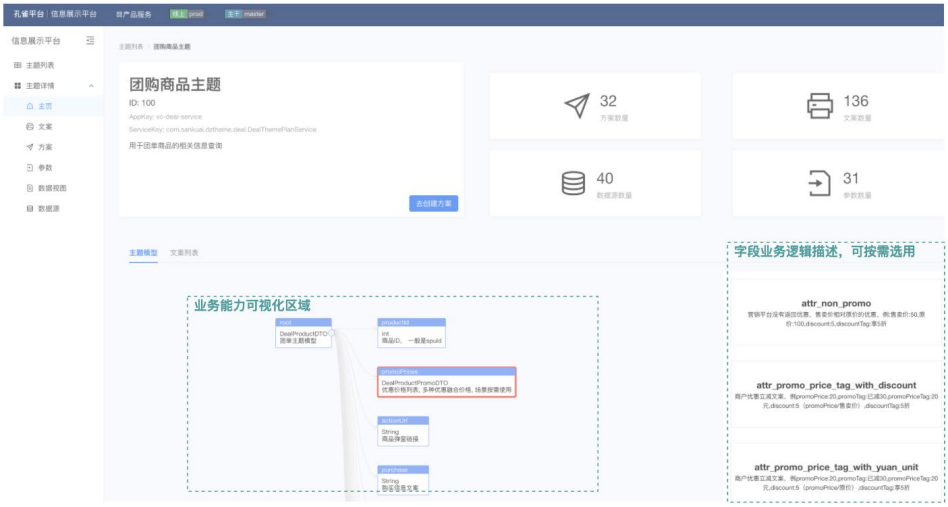


图 25　可视化及文案按需选用

情况二：此时可能没有展示功能，但是通过运营平台查看到，数据源已经接入过，那么也不难，只需要基于现有的数据源编写一段加工逻辑即可，这段加工逻辑是非常爽的一段纯逻辑的编写，数据源列表如下示意图所示：



图 26　数据源列表可视化

情况三：最坏的情况是此时系统不能满足当前的查询能力，这种情况比较少见，因为后端服务是比较稳定的，那么也无需惊慌，只需要按照标准规范将数据源接入进来，然后编写加工逻辑片段即可，之后这些能力是可以被持续复用的。

7. 总结

商品展示场景的复杂性体现在：场景多、依赖多、逻辑多，以及不同场景之间存在差异。在这样的背景下，如果是业务初期，怎么快怎么来，采用“烟囱式”个性化建设的方式不必有过多的质疑。但是随着业务的不断发展，功能的不断迭代，以及场景的规模化趋势，“烟囱式”个性化建设的弊端会慢慢凸显出来，包括代码复杂度高、缺少能力沉淀等问题。

本文以基于对美团到店商品展示场景所面临的核心矛盾分析，介绍了：

- 业界不同的 BFF 应用模式，以及不同模式的优势和缺点。
- 基于 GraphQL BFF 模式改进的元数据驱动的架构方案设计。
- 我们在 GraphQL 实践过程中遇到的问题及解决思路。
- 新架构对研发模式产生的影响呈现。

目前，笔者所在团队负责的核心商品展示场景都已迁入新架构，基于新的研发模式，我们实现了 50% 以上的展示逻辑复用以及 1 倍以上的效率提升，希望本文对大家能够有所帮助。

8. 参考文献

[1]https://samnewman.io/patterns/architectural/bff/
[2] https://www.thoughtworks.com/cn/radar/techniques/graphql-for-server-side-resource-aggregation
[3] 了解电商后台系统，看这篇就够了
[4] 框架定义 - 百度百科
[5] 高效研发 - 闲鱼在数据聚合上的探索与实践
[6]《系统架构 - 复杂系统的产品设计与开发》

9. 招聘信息

美团到店综合研发中心长期招聘前端、后端、数据仓库、机器学习 / 数据挖掘算法工程师，坐标上海，欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：美团到店综合研发中心—上海）。

速度与压缩比如何兼得？压缩算法在构建部署中的优化

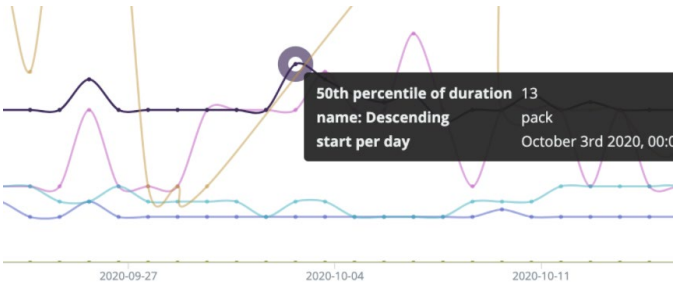
作者：宏达

背景

通常而言，服务发布平台的构建部署的流程（镜像部署除外）会经过**构建**（同步代码 -> 编译 -> 打包 -> 上传）、**部署**（下载包 -> 解压到目标机器 -> 重启服务）等步骤。以美团内部的发布平台 Plus 为例，最近我们发现一些发布项在构建产物打包压缩的过程中耗时比较久。如下图所示的 pack 步骤，一共消耗了 1 分 23 秒。

build	成功	2020-09-27 15:47:08	2020-09-27 15:53:40	查看日志
pack	成功	2020-09-27 15:53:40	2020-09-27 15:55:03	查看日志
collect	成功	2020-09-27 15:55:03	2020-09-27 15:55:03	查看日志

而在平常为用户解答运维问题的时候我们也发现，很多用户会习惯将一些较大的机器学习或者 NLP 相关的数据放入到仓库中，这部分数据往往占据几百兆，甚至占据几个 GB 的磁盘空间，十分影响打包的速度。 Java 项目也是如此，由于 Java 服务框架繁多，依赖也多，通常这些服务打包后也要占据百兆级别的空间，耗时也会达到十多秒。下图是我们的 pack 步骤的中位数，基本上大部分的 Java 服务和 Node.js 服务都至少要消耗 13s 左右的时间来做压缩打包 。



pack 作为几乎所有需要部署的服务必需步骤，它目前的耗时基本上仅低于编译和构

建镜像，因此，为了提高整体构建的效率，我们准备对 pack 打包压缩的步骤进行一轮优化工作。

方案对比

准备场景数据

发布项的包大小分析

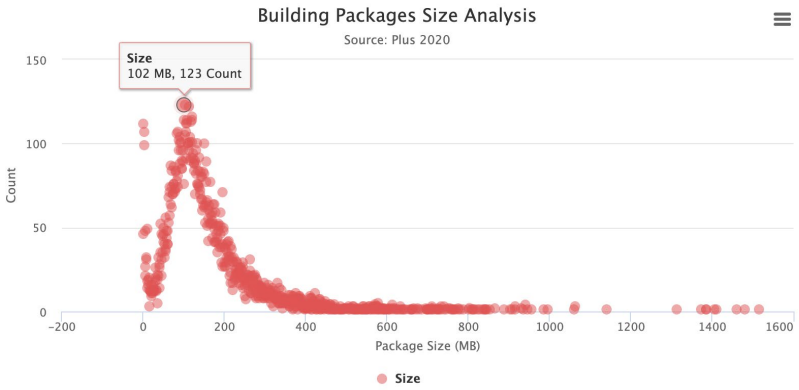
为了尽可能地模拟构建部署中的应用场景，我们将 2020 年的部分**构建包数据**进行了整理分析，其中压缩后的包大小如下图所示，钟形曲线说明了整体的包体积呈正态分布，并且有着较明显的长尾效应。压缩后体积主要在 200M 以内，压缩前的大小大致在 516.0MB 以内。

而 99% 的服务压缩包大小会在 1GB 以内，而对于压缩步骤而言，其实越大的项目耗时越明显，优化的空间越大。因此，我们在针对性的方案对比测试中选择了 1GB 左右的构建包进行压缩测试，既能覆盖 99% 的场景，也可以看出压缩算法之间比较明显的提升。

这样选择的主要原因如下：

- 数据大的情况下计算结果会比小数据误差小很多。
- 能够覆盖绝大多数应用场景。
- 效果对比明显，可以看到是否有明显的提升。

备注：由于在相同压缩库相同压缩比等配置的情况下，Compression Speed 并没有明显变化，因此没有做其它包体积的批量测试和数据汇总。



本文中我们使用的测试项目为美团内部的较大型的 C++ 项目，其中文件类型除去 C++、Python、Shell 代码文件，还有 NLP、工具等二进制数据（不包括 .git 中存储的提交数据），数据类型比较全面。

目录大小为 1.2G，也可以比较清晰地对比出不同方案之间的差距。

gzip

gzip 是基于 DEFLATE 的算法，它是 LZ77 和 Huffman 编码 的结合。DEFLATE 的目的是为了取代 LZW 和其他受专利保护的数据压缩算法，因为这些算法在当时限制了压缩和其他流行的存档器的可用性（Wikipedia）。

我们通常使用 `tar -czf` 命令来进行打包并且压缩的操作，z 参数正是使用 gzip 的方式来进行压缩。DEFLATE 标准（RFC1951）是一个被广泛使用的无损数据压缩标准。它的压缩数据格式由一系列块构成，对应输入数据的块，每一块通过 LZ77（基于字典压缩，就是将最高概率出现的字母以最短的编码表示）算法和 Huffman 编码进行压缩，LZ77 算法通过查找并替换重复的字符串来减小数据体积。

文件格式

- 一个 10 字节的报头，包含一个魔数 (1f 8b)，压缩方法（比如 08 用于 DEFLATE），1 字节的 header flags，4 字节的时间戳，compression flags 和操作系统 ID。

- 可选的额外 headers，包括原始文件名、注释字段、“extra”字段和 header 的 CRC-32 校验码 lower half 。
- DEFLATE 压缩主体。
- 8 字节的 footer，包含 CRC-32 校验以及原始未压缩的数据。

我们可以看到 gzip 是主要基于 CRC 和 Huffman LZ77 的 DEFLATE 算法，这也是后面 ISA-L 库的优化目标。

Brotli

Alakuijala 和 Szabadka 在 2013-2016 年完成了 Brotli 规范，该数据格式旨在进一步提高压缩比，它在优化网站速度上有大量应用。Brotli 规范的正式验证是由 Mark Adler 独立实现的。Brotli 是一个用于数据流压缩的数据格式规范，它使用了通用的 LZ77 无损压缩算法、Huffman 编码和二阶上下文建模（2nd order context modelling）的特定组合。大家可以参考这篇论文查看其实现原理。

因为语言本身的特性，基于上下文的建模方法（Context Modeling）可以得到更好的压缩比，但由于它的性能问题很难普及。当前比较流行的突破算法有两种：

- ANS：Zstd, lzfse
- Context Modeling：Brotli, bz2

具体测试数据见下文。

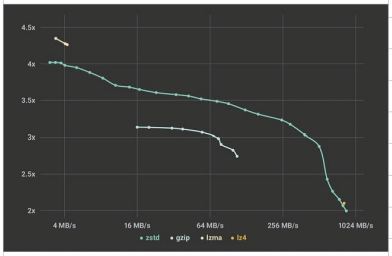
Zstd

Zstd 全称叫 Zstandard，是一个提供高压缩比的快速压缩算法，主要实现的编程语言为 C，是 Facebook 的 Yann Collet 于 2016 年发布的，Zstd 采用了有限状态熵（Finite State Entropy，缩写为 FSE）编码器。该编码器是由 Jarek Duda 基于 ANS 理论开发的一种新型熵编码器，提供了非常强大的压缩速度 / 压缩率的折中方案（事实上也的确做到了“鱼”和“熊掌”兼得）。Zstd 在其最大压缩级别上提供的压缩比接近 lzma、lzham 和 ppmx，并且性能优于 lza 或 bzip2。Zstandard 达到了

Pareto frontier（资源分配最佳的理想状态），因为它解压缩速度快于任何其他当前可用的算法，但压缩比类似或更好。

对于小数据，它还特别提供一个载入预置词典的方法优化速度，词典可以通过对目标数据进行训练从而生成。

官方 Benchmark 数据对比

zstd 与 gzip lzma lz4 对比	Compressor name	Ratio	Compression	Decompress
	Zstd 1.4.5 -1	2.884	500 MB/s	1660 MB/s
	zlib 1.2.11 -1	2.743	90 MB/s	400 MB/s
	brotli 1.0.7 -0	2.703	400 MB/s	450 MB/s
	Zstd 1.4.5 --fast=1	2.434	570 MB/s	2200 MB/s
	Zstd 1.4.5 --fast=3	2.312	640 MB/s	2300 MB/s
	Zstd 1.4.5 --fast=5	2.178	700 MB/s	2420 MB/s
	lzo1x 2.10 -1	2.106	690 MB/s	820 MB/s
	lz4 1.9.2	2.101	740 MB/s	4530 MB/s
	snappy 1.1.8	2.073	560 MB/s	1790 MB/s

压缩级别可以通过 –fast 指定，提供更快的压缩和解压缩速度，相比级别 1 会导致压缩比率的一些损失，如上表所示。Zstd 可以用压缩速度换取更强的压缩比。它是可配置的小增量，在所有设置下，解压缩速度都保持不变，这是大多数 LZ 压缩算法（如 zlib 或 lzma）共享的特性。

- 我们采用 Zstd 默认的参数进行了测试，压缩时间 8.471s 仅为原来的 11.266%，提升了 88.733%。
- 解压时间 3.211 仅为原来的 29.83%，提升约为 70.169%。
- 同时压缩率也从 2.548 提升到了 2.621。

LZ4

LZ4 是一种无损压缩算法，每核提供大于 500 MB/s 的压缩速度（大于 0.15 Bytes/cycle）。它的特点是解码速度极快，每核速度为多 GB/s（ 约 1 Bytes/cycle ）。

从上面的 Zstd 的 Benchmark 对比中，我们看到了 LZ4 算法效果十分出众，因此我们也对 LZ4 进行了对比，LZ4 更加侧重压缩解压速度，尤其是解压缩的速度，压缩

比并不是它的强项，它默认支持 1-9 的压缩参数，我们分别进行了测试。

LZ4 使用默认参数压缩速度十分优秀，比 Zstd 快很多，但是压缩比并不高，比 Zstd 压缩后多了 206 MB，足足多了 46%，这就意味着更多的数据传输时间和磁盘空间占用。即使是最大的压缩比也并不高，仅仅从 1.79 提升到了 2.11，但是耗时却从 5s 提升到了 51s。通过对比，LZ4 的确在压缩率上并不是最优秀的方案，在 2.x 级别压缩率上基本上时间优势荡然无存，而且还有一点，就是 LZ4 目前官方并没有对多核 CPU 并行压缩的支持，所以在后续的对比中，LZ4 丧失了压缩解压缩速度的优势。

Pigz

Pigz 的作者 Mark Adler，同时也是 Info-ZIP 的 zip 和 unzip、GNU 的 gzip 和 zlib 压缩库的共同作者，并且是 PNG 图像格式开发工作的参与者。

Pigz 是 gzip 的并行实现的缩写，它主要思想就是利用多个处理器和核。它将输入分成 128 KB 的块，每个块都被并行压缩。每个块的单个校验值也是并行计算的。它的实现直接使用了 zlib 和 pthread 库，比较易读，而且重要的是兼容 gzip 的格式。Pigz 使用一个线程（主线程）进行解压缩，但可以创建另外三个线程进行读、写和检查计算，所以在某些情况下可以加速解压缩。

一些博客在 i7 4790K 这样的家用 PC 平台中测试 Pigz 的压缩性能时，并没有十分高的速度，但在我们真机验证的数据中提升要明显很多。通过测试，它的压缩时间执行速度只用了 1.768s，充分发挥了我们平台物理机的性能，User 时间（CPU 时间之和）一共使用了 1m30.569s，这和前面的使用 gzip 单线程的方式速度几乎是一个级别。压缩率 2.5488 和正常使用 `tar -czf` 几乎相差不多。

ISA-L Acceleration Version

ISA-L 是一套在 IA 架构上加速算法执行的开源函数库，目的在于解决存储系统的计算需求。ISA-L 使用的是 BSD-3-Clause License，因此在商业上同样可以使用。

使用过 SPDK（Storage Performance Development Kit）或者 DPDK（Data

Plane Development Kit）应该也听说过 ISA-L ，前者使用了 ISA-L 的 CRC 部分，后者使用了它的压缩优化。ISA-L 通过使用高效的 SIMD（Single Instruction, Multiple Data）指令和专用指令，最大化的利用 CPU 的微架构来加速存储算法的计算过程。ISA-L 底层函数都是使用手工汇编代码编写，并在很多细节上做了调优（现在经常要考虑 ARM 平台，本文中所提及的部分指令在该平台支持度不高甚至是不支持）。

ISA-L 对压缩算法主要做了 CRC、DEFLATE 和 Huffman 编码的优化实现，官方的数据指出 ISA-L 相比 zlib-1 有 5 倍的速度提升。

举例来说，不少底层的存储开源软件实现的 CRC 都使用了查表法，代码中存储或者生成了一个 CRC value 的表格，然后计算过程中查询值，ISA-L 的汇编代码中包含了无进位乘法指令 PCLMULQDQ 对两个 64 位数做无进位乘法，最大化 intel PCLMULQDQ 指令的吞吐量来优化 CRC 的性能。更好的情况是 CPU 支持 AVX-512，就可以使用 VPCLMULQDQ（PCLMULQDQ 在 EVEX 编码的 512 bit 版本实现）等其它优化指令集（查看是否支持的方式见“附录”）。

```
244     .16B_reduction_loop:
245         vpclmulqdq  xmm8, xmm7, xmm10, 0x11
246         vpclmulqdq  xmm7, xmm7, xmm10, 0x00
247         vpxor       xmm7, xmm8
248         vmovdqu     xmm0, [arg2]
249         vpshufb     xmm0, xmm0, xmm18
250         vpxor       xmm7, xmm0
251         add     arg2, 16
252         sub     arg3, 16
```

备注：截图来自 crc32_ieee_by16_10.asm

使用

ISA-L 实现的压缩优化级别支持 [0,3]，3 为压缩比最大的版本，综合考虑我们采用了最大的压缩比，虽然压缩比 2.346 略低于 gzip 不过影响不大。在 2019 年 6 月发

布的 v2.27 版本里，ISA-L 加了多线程的 Feature，因此在后续的测试中，我们采用了多线程并发的参数，效果提升比较显著。

由于我们构建机器的系统为 Centos7 需要自行编译 ISA-L 的依赖，比如 nasm 等库，所以在安装配置上会比较复杂，ISA-L 支持多种优化参数比如，IGZIP_HIST_SIZE（压缩过程中加大滑动窗口），LONGER_HUFFTABLES，更大的 Huffman 编码表，这些编译参数也会对库有很大提升。

压缩时间

```
real  0m1.372s
user  0m5.512s
sys 0m1.791s
```

测试后的效果相当惊人，是目前对比方案中最为快速的，时间上节省了 98.09%。

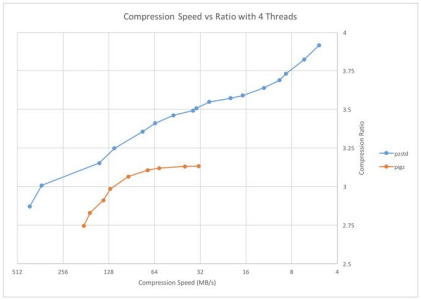
由于和 gzip 格式兼容，因此同样可以使用 `tar -xf` 命令进行解压，后续的解压缩测试过程中，我们使用的仍然是 ISA-L 提供的解压方式。

Pzstd

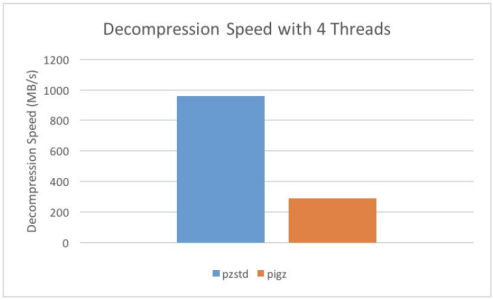
通过 Pigz 的测试，我们就在想，是否 Zstd 这样优秀的算法也可以支持并行呢，在官方的 Repo 中，我们十分惊喜地发现了一个“宝藏”。

Pzstd 是 C++11 实现的并行版本的 Zstandard（Zstd 也在这之后加入了多线程的支持），类似于 Pigz 的工具。 它提供了与 Zstandard 格式兼容的压缩和解压缩功能，可以利用多个 CPU 核心。 它将输入分成相等大小的块，并将每个块独立压缩为 Zstandard 帧。 然后将这些帧连接在一起以产生最终的压缩输出。 Pzstd 同样支持文件的并行解压缩。 解压缩使用 Zstandard 压缩的文件时，PZstandard 在一个线程中执行 IO，而在另一个线程中进行解压缩。

下图是和 Pigz 的压缩和解压缩速度对比，来自官方 Github 仓库（机器配置为：Intel Core i7 @ 3.1 GHz, 4 threads），效果比 Pigz 还要出色，具体对比数据见下文：



压缩速度



解压缩速度

Pied Piper (Middle-out compression)

Middle-out Compression 最初是在美剧中提到的一个概念，不过现在已经有了一个真正的实现 middle-out，该算法目前小范围应用在压缩时序数据中，由于缺乏成熟地理论支撑以及数据对比，没有正式进入方案的对标。

备注：Pied Piper 是美剧《硅谷》中虚拟出来的公司和算法。



兼容性

本文中调研所提及的对比方案，都是统一在构建集群中的机器中进行测试，由于构建系统 在线上的机器集群配置高度统一（包括硬件平台和系统版本），所以兼容性表现和测试数据也高度吻合。

选型

实际上，部分官方的测试机器的配置和美团构建平台的物理机配置并不一致，场景可能也有区别，上面引用的测试结果中所使用的 CPU 以及平台架构和编译环境和我们所用的也有些出入，而且大多数还是家用的硬件比如 i7-4790K 或者 i9-9900K，这也是需要使用构建平台的物理机和具体的构建包压缩场景来进行测试的原因，因为这样才能得出最接近我们使用场景的数据。

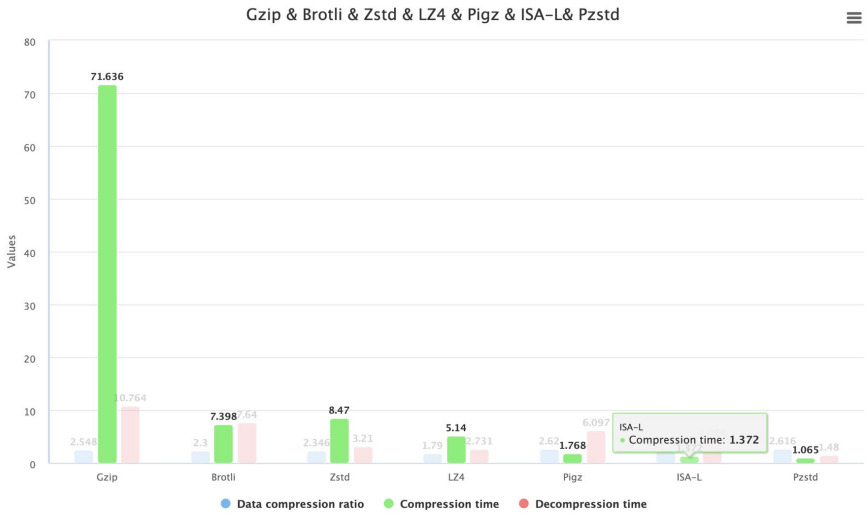
对比数据

几个方案的数据对比如下表格（在本文中的时间数据选择是通过多次运行后，选择结果的中位数）：

	Gzip	Brotli	Zstd	LZ4	Pigz	ISA-L	Pzstd
Data Compression Ratio	2.5482	2.3068	2.621	1.791	2.5488	2.346	2.616
Compression Time	71.636	7.398	8.471	5.141	1.768	1.372	1.065
Compression Speed	16.272 MB/s	157.57 MB/s	137.613 MB/s	226.75 MB/s	659.345 MB/s	849.652 MB/s	1094.5752 MB/s
Decompression Time	10.764	7.643	3.211	2.731	6.097	4.094	1.481

压缩时间对比

从整个构建后的压缩构建包的时间可视化图中可以看出，最初版本的 gzip 压缩相当耗时，而采取 Pzstd 是最快速的方案，ISA-L 稍慢，Pigz 略微慢一点，这**三者都可以达到从 1m11s 到 1s 左右的优化，最快可以节省 98.51% 的压缩时间**。



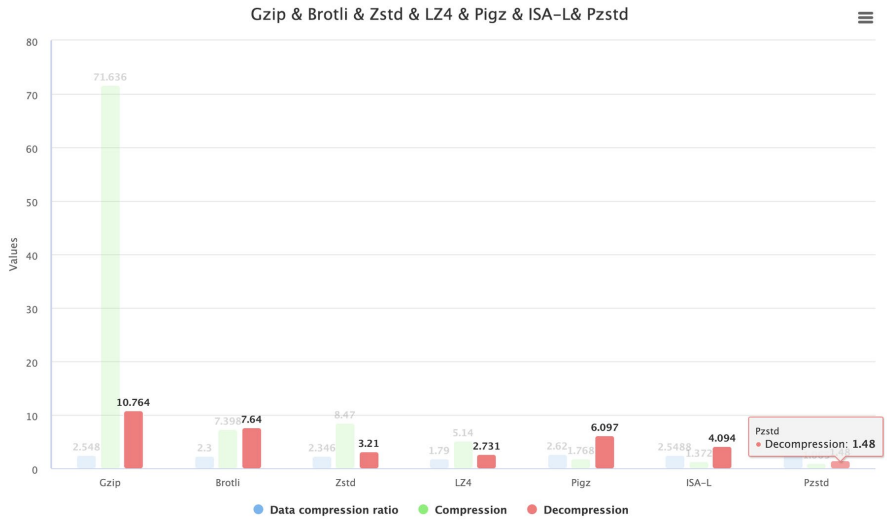
解压缩时间对比

解压缩的时间上并没有像压缩效率相差很多，在 GB 级别的项目中压缩比 2.5-2.6 范围内时，时间都在 11s 以内。而且为了最大兼容已有的实现和保持稳定性，解压方案优先考虑兼容 gzip 格式的策略，这样对部署目标机器的侵入性最小，即可以使用 `tar -xf` 解压的方案优先。

而在后面的方案实施中，由于部署需要稳定可靠的环境，所以我们暂时没有对部署机器做环境改造。

下面的时间对比是分别使用各自的解压方案的对比：

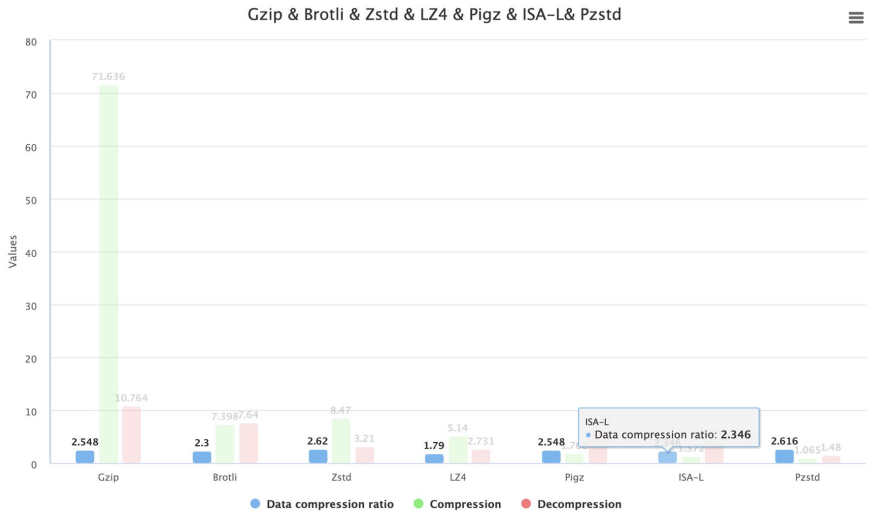
- Pzstd 解压速度最快，相比 Gzip 节省了 86.241% 的时间。
- Zstd 算法的解压缩效率其次，大约可以节省 70.169% 的解压时间。
- ISA-L 可节省 61.9658% 的时间。
- Pigz 可节省 43.357% 的解压时间。
- Brotli 解压可以节省 29.02% 的时间。



压缩比的对比

压缩比的对比中 Zstd 和 Pzstd 有一些优势，其中 Brotli 和 LZ4 由于支持的参数限制，比较难测试同级别压缩比下的速度，因此选择了压缩比稍低的参数，但是效率仍然距离 Pigz 和 Pzstd 存在一些差距。

而 ISA-L 的实现在压缩比上有一些牺牲，不过并没有差距很大。



优劣分析总结

	理念	优势	劣势
Brotli	1. **优秀的新算法**	• 对于小文件的速度可能会稍微明显一些 • 压缩解压缩的速度比较不错，但是并不突出。	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 Brotli 来进行处理，对于其它有可能已经使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 大型文件尤其是在构建场景中，它的压缩速度并不是十分突出。 • 解压必需 Brotli 库的安装（也就是说部署的目标机器需要安装 Brotli 库）
Zstd	1. **优秀的新算法**	• **压缩解压速度十分优秀。** • 安装依赖单一，避免遇到麻烦的依赖冲突问题。 • **压缩比十分优秀。**	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 zstd 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 zstd 库的安装（也就是说部署的目标机器需要安装 zstd 库）
LZ4	1. **优秀的新算法**	• **压缩解压速度十分优秀。** • 安装依赖单一，避免遇到麻烦的依赖冲突问题。	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 LZ4 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 LZ4 库的安装（也就是说部署的目标机器需要安装 LZ4 库） • 压缩率并不理想
Pigz	1. 并行策略	• **压缩结果完全兼容 gzip 格式**，即便解压不做改动也可以完全兼容，这对于使用了构建压缩包的其它服务可能是十分完美的结果。同时对于部署目标机器的侵入性最小。 • **压缩解压速度极其优秀，仅次于 Pzstd 和 ISA-L。** • 对 tar 支持比较友好，可以方便地使用外接压缩插件。 • 安装依赖单一，避免遇到麻烦的依赖冲突问题。	无
ISA-L	1. gzip 算法以及底层指令优化 2. 并行策略	• **压缩结果完全兼容 gzip 格式**，即使解压不做改动也可以完全兼容，这对于使用了构建压缩包的其它服务可能是十分完美的结果。同时对于部署目标机器没有侵入性。 • **压缩解压速度极其优秀，仅次于 Pzstd。** • 对 tar 支持比较友好。	• 压缩比 稍微低一点点，且压缩比的参数范围有限。 • 实现接入成本略微复杂（不过好在我们的机器集群配置环境十分统一）。
Pzstd	1. **优秀的新算法** 2. 并行策略	• **压缩和解压的速度都极其优秀。** • 对 tar 支持比较友好，只需要在机器上安装 zstd 并且修改 tar 参数即可，改动较小。 • 安装依赖单一，避免遇到麻烦的依赖冲突问题。 • **压缩比十分优秀。**	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 zstd 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 zstd 库的安装（也就是说部署的目标机器需要安装 zstd 库）

在本文开始阶段，我们介绍了构建部署流程，因此我们此次优化的目标时间大致计算公式如下：

$$TotalCost = Time_{compression} + \frac{PackageSize}{UploadSpeed} + \frac{PackageSize}{DownloadSpeed} + Time_{decompression}$$

在测试案例对比中，时间耗时的顺序为 Pzstd < ISA-L < Pigz < LZ4 < Zstd < Brotli < Gzip（排名越靠前越好），其中压缩和解压缩的时间在整体的耗时上占比较大，因此备选策略为 Pzstd、ISA-L、Pigz。

当然，这其中还存在磁盘空间的成本优化问题，即压缩比可能对磁盘空间产生优化，

但是对构建的耗时会产生负优化，不过由于目前时间维度是我们优化的主要目标，比磁盘成本和上传带宽成本要重要，因此压缩比采用了较为普遍或者默认最优的压缩比方案，即在 2.3 – 2.6 范围内。不过在一些内存型数据库等存储介质成本较为高的场景中，也许要综合多个方面需要更多考量，请大家知悉。

评分策略

比较于 gzip，新算法都具有想当不错的速度，但是由于压缩格式的向前不兼容，加上需要客户端（部署目标机器）的支持，可能方案实施周期会较长。

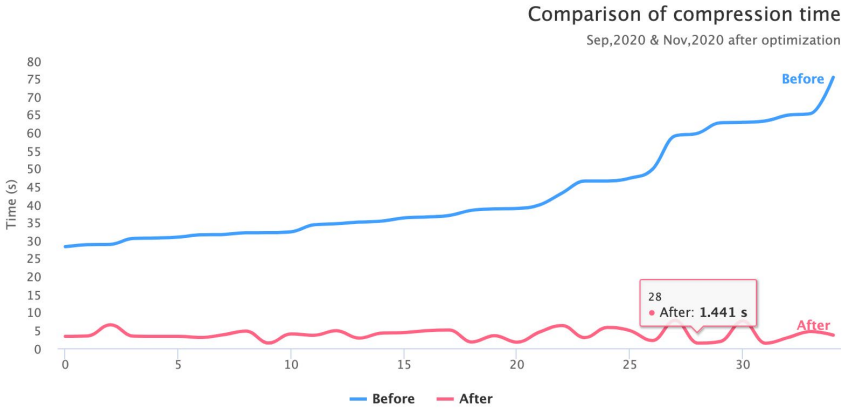
而对于部署来说，可能收益并不是十分明显，反而加重了一些维护和运维成本，所以我们暂时没有把 Pzstd 的方案放到较高的优先级。

选型的策略主要有基于如下原则：

- 整体耗时优化提升最大，这也是整体优化方案的出发点。
- 保证最大兼容性，为了让接入构建平台的业务和平台减少改动成本，需要保持方案的兼容性（优先考虑最大兼容的策略，即兼容 gzip 的方案优先）。
- 保证部署目标机器环境的稳定和可靠，选择对部署机器侵入最小的方案，这样无需安装客户端或者库。
- 压缩场景在真机模拟测试中完全契合美团构建平台的场景，即在我们现有的物理机平台和目标压缩场景中对比数据效果良好。
- 其实本问题更全面的评分角度有很多维度，比如对象存储的磁盘成本、带宽成本、任务耗时，甚至是机器成本，不过为了简化整体方案的选型，我们省略了一些计算，同时压缩比的对比选择上也选择了各自官方推荐的范围。

综合以上几点，决定一期采取 ISA-L 的方式加速，可以最稳定并且较高速地提升构建平台的效率，未来可能会实现 Pzstd 的方案，下面的数据为一期的结果。

优化效果

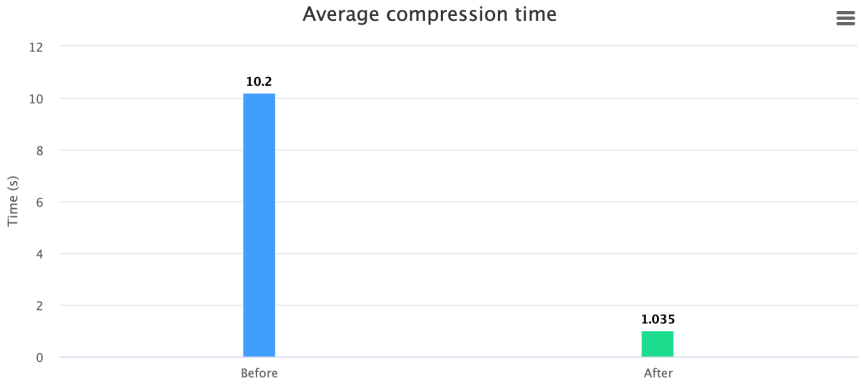


为了方便结果的展示，我们过滤出了部分打包时间较长的发布项展示出来（这些耗时很久的项目往往十分影响用户的使用体验，而且总体的占比在 10% 左右），这部分任务优化的时间从 27s 到 72s 不等，过去越是项目大的项目压缩时间越长，如今压缩时间都可以稳定在 10s 以内，而且是在我们的机器同时执行多个任务的情况下。

而后我们将优化前的 Pack 步骤（压缩 + 上传）部分打点数据，以及优化后的部分打点数据做了汇总，得出了平均的优化效果对比，数据如下：

- 在我们之前的一个构建包的统计中，多数的构建包压缩后在 100MB 左右，压缩前大概是在 250MB，按照 gzip 算法的压缩速度的确会在 10s 左右的级别。
- 由于构建的物理机可能同时运行多个任务，所以实际压缩效果会比测试中稍微耗时多一点。

	Pack 耗时之和	任务数	平均时间	压缩时间（去除复制文件和上传的时间）
优化前	908202	65613	13.8417996433634	10.2
优化后	215298	46853	4.59518067146181	1.035



压缩平均节省了 90% 的时间。

写在后面

- 由于文中提到的一些方案涉及到具体平台环境的 CPU 指令集，甚至库的编译环境，编译参数也会影响具体的效果，所以推荐在方案实施的时候对集群环境保持统一，也可为集群环境做特殊的定制优化。
- 为 Centos 打包 RPM 文件的时候尤其需要注意下编译环境的配置，否则可能效果和测试会有出入。
- Java 的 Jar 包 和 War 包也可能进行压缩，针对这种场景，压缩率的确提升不大，但是速度依旧有提升。

作者简介

宏达，美团基础技术部研发工程师。

团队简介

基础技术部 – 研发质量及效率部 – 代码仓库和构建组，团队旨在建设代码仓库管理、协作及代码构建能力，完善多维度的工作流执行引擎及构建工具链，与公司其他研发工具打通，提高业务整体的开发、交付效率。

附录

机器环境

文中的测试统一在如下物理机中进行，测试中使用相同的目标文件。测试 机使用的是非 PCIE SSD 磁盘。

```
inxi -Fx 省略部分数据输出如下，其中一些并行指令集在优化中可能会使用到。其中 flags 中可以看到支持 avx avx2 指令集，并不支持 avx-512 ，不过仍然有很大性能提升。
System:    Host: ****** Kernel: ****** bits: 64 compiler: gcc v: 4.8.5
Console: tty 7 Distro: CentOS Linux release 7.1.1503 (Core)

CPU:       Info: 2x 16-Core model: Intel Xeon Gold 5218 bits: 64
type: MT MCP SMP arch: Cascade Lake rev: 7 L2 cache: 44.0 MiB

           flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2
ssse3 vmx bogomips: 293978 Speed: 2300 MHz min/max: N/A

Info:        Processes: 764  Memory: 187.19 GiB used: 15.05 GiB
(8.0%) Init: systemd runlevel: 3 Compilers: gcc: 4.8.5


/proc/cpuinfo 文件中也可以查看 CPU 支持的指令集
```

```
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
at pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3
intel_ppin intel_pt ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm
mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw
avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm
_local dtherm ida arat pln pts pku ospke spec_ctrl intel_stibp flush_l1d
arch_capabilities
```

参考文献

- https://engineering.fb.com/core-data/smaller-and-faster-data-compression-with-zstandard/
- https://engineering.fb.com/core-data/zstandard/
- https://peazip.github.io/fast-compression-benchmark-brotli-zstandard.html
- https://news.ycombinator.com/item?id=16227526
- https://github.com/facebook/zstd

- http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html
- https://zlib.net/pigz/
- https://cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf
- https://bugs.python.org/issue41566
- https://01.org/sites/default/files/documentation/isa-l_api_2.28.0.pdf
- https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf

招聘信息

美团研发质量及效率部 – 研发平台团队，致力于建设业界一流的持续交付平台，现招聘构建与制品库、代码仓库、服务发布、流水线等多个方向的工程师，坐标北京 / 上海。欢迎感兴趣的同学加入。可投递简历至：dongjing04@meituan.com（邮件主题请注明：美团研发质量及效率部 – 研发平台）

百亿规模 API 网关服务 Shepherd 的设计与实现

作者：充泽　志洋　李敏

一、背景介绍

1.1 API 网关是什么?

API 网关是随着微服务（Microservice）概念兴起的一种架构模式。原本一个庞大的单体应用（All in one）业务系统被拆分成许多微服务（Microservice）系统进行独立的维护和部署，服务拆分带来的变化是 API 的规模成倍增长，API 的管理难度也在日益增加，使用 API 网关发布和管理 API 逐渐成为一种趋势。一般来说，API 网关是运行于外部请求与内部服务之间的一个流量入口，实现对外部请求的协议转换、鉴权、流控、参数校验、监控等通用功能。

1.2 为什么要做 Shepherd API 网关?

在没有 Shepherd API 网关之前，美团业务研发人员如果要将内部服务输出为对外的 HTTP API 接口。通常要搭建一个 Web 应用，用于完成基础的鉴权、限流、监控日志、参数校验、协议转换等工作，同时需要维护代码逻辑、基础组件的升级，研发效率相对比较低。此外，每个 Web 应用都需要维护机器、配置、数据库等，资源利用率也非常差。

美团内部一些业务线苦于没有现成的解决方案，根据自身业务特点，研发了业务相关的 API 网关。放眼业界，亚马逊、阿里巴巴、腾讯等公司也都有成熟的 API 网关解决方案。

因此，Shepherd API 网关项目正式立项，我们的目标是为美团提供高性能、高可用、可扩展的统一 API 网关解决方案，让业务研发人员通过配置的方式即可对外开放功能和数据。

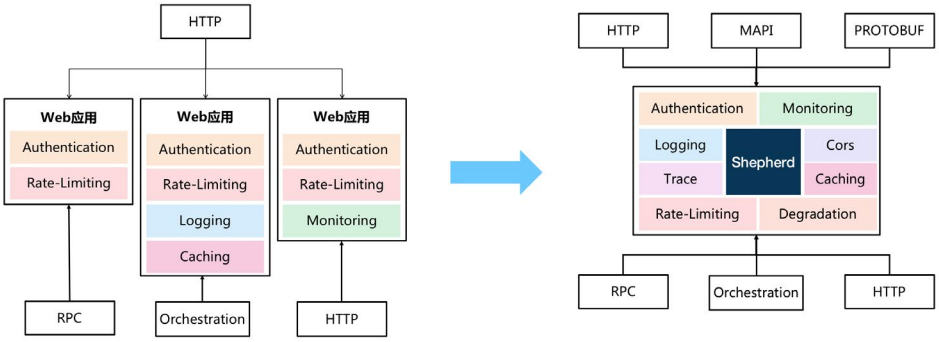


图 1

1.3 使用 Shepherd 带来的收益是什么?

从业务研发人员的角度来看，接入 Shepherd API 网关，能带来哪些收益呢？简而言之包括三个方面。

- **提升研发效率**
  - 业务研发人员只需要通过配置的方式即可快速开放服务接口。
  - Shepherd 统一提供鉴权、限流、熔断等非业务基础能力。
  - Shepherd 支持业务研发人员通过开发自定义组件的方式扩展 API 网关能力。
- **降低沟通成本**
  - 业务研发人员配置好 API，可以自动生成 API 的前后端交互文档和客户端 SDK，方便前后端开发人员进行交互、联调。
- **提升资源利用率**
  - 基于 Serverless 的架构思想，实现 API 全托管，业务研发人员无需关心机器资源问题。

二、技术设计与实现

2.1 整体架构

我们先来看看 Shepherd API 网关的整体架构，如下图所示：

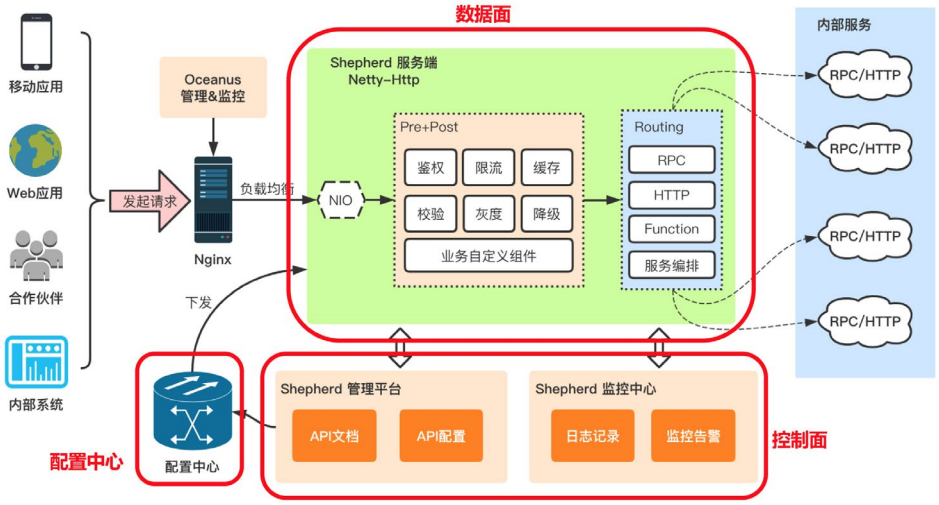


图 2

Shepherd API 网关的**控制面**由 Shepherd 管理平台和 Shepherd 监控中心组成。管理平台主要完成 API 的全生命周期管理以及配置下发的工作，监控中心完成 API 请求监控数据的收集和业务告警功能。

Shepherd API 网关的**配置中心**主要完成控制面与数据面的信息交互，通过美团统一配置服务 Lion 来实现。

Shepherd API 网关的**数据面**也就是 Shepherd 服务端。一次完整的 API 请求，可能是从移动应用、Web 应用，合作伙伴或内部系统发起，经过 Nginx 负载均衡系统后，到达服务端。服务端集成了一系列的基础功能组件和业务自定义组件，通过泛化调用请求后端 RPC 服务、HTTP 服务、函数服务或服务编排服务，最后返回响应结果。

下面我们将针对这三个主要模块做详细的介绍。

2.1.1 控制面

使用 API 网关的控制面，业务研发人员可以轻松的完成 API 的全生命周期管理，如下图所示：

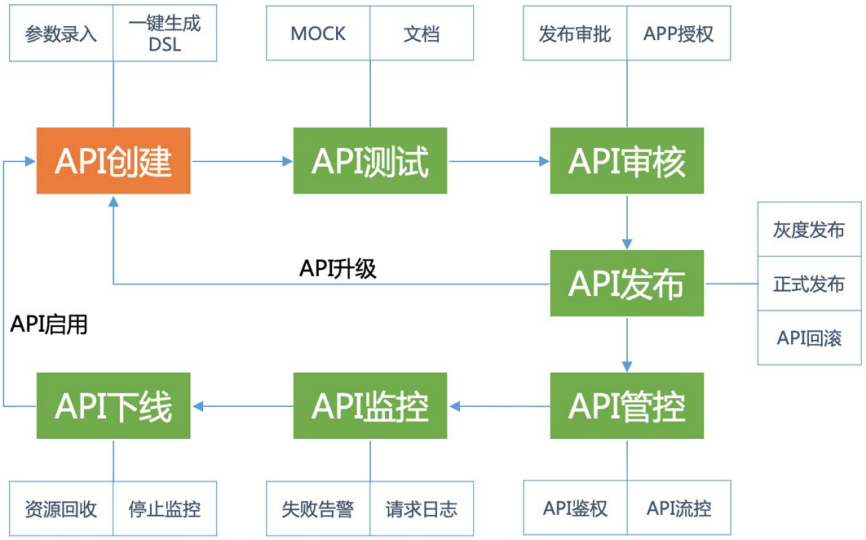


图 3

业务研发人员从创建 API 开始，完成参数录入、DSL 脚本生成；接着可以通过文档和 MOCK 功能进行 API 测试；API 测试完成后，为了保证上线稳定性，Shepherd 管理平台提供了发布审批、灰度上线、版本回滚等一系列安全保证措施；API 运行期间会监控 API 的调用失败情况、记录请求日志，一旦发现异常及时发出告警；最后，对于不再使用的 API 进行下线操作后，会回收 API 所占用的各类资源并等待重新启用。

整个生命周期，全部通过配置化、流程化的方式，由业务研发人员全自助管理，上手时间基本在 10 分钟以内，极大地提升了研发效率。

2.1.2 配置中心

API 网关的配置中心存放 API 的相关配置信息——使用自定义的 DSL（Domain-Specific Language，领域专用语言）来描述，用于向 API 网关的数据面下发 API 的路由、规则、组件等配置变更。

配置中心的设计上使用统一配置管理服务 Lion 和本地缓存结合的方式，实现动态配

置，不停机发布。API 的配置如下图所示：



图 4

API 配置的详细说明：

- **Name、Group**：名字、所属分组。
- **Request**：请求的域名、路径、参数等信息。
- **Response**：响应的结果组装、异常处理、Header、Cookies 信息。
- **Filters、FilterConfigs**：API 使用到的功能组件和配置信息。
- **Invokers**：后端服务 (RPC/HTTP/Function) 的请求规则和编排信息。

2.1.3 数据面

API 路由

API 网关的数据面在感知到 API 配置后，会在内存中建立请求路径与 API 配置的路由信息。通常 HTTP 请求路径上，会包含一些路径变量，考虑到性能问题，Shepherd 没有采用正则匹配的方式，而是设计了两种数据结构来存储。如下图所示：

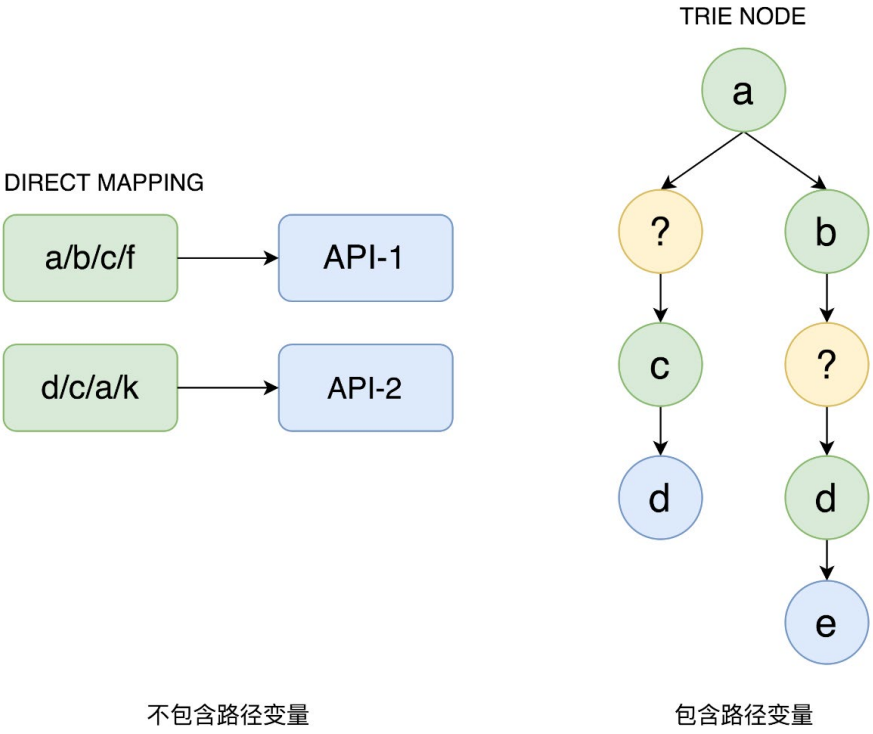


图 5

一种是不包含路径变量的直接映射的 MAP 结构。其中，Key 就是完整的域名和路径信息，Value 是具体的 API 配置。

另外一种是包含路径变量的前缀树数据结构。通过前缀匹配的方式，先进行叶子节点精确查找，并将查找节点入栈处理，如果匹配不上，则将栈顶节点出栈，再将同级的变量节点入栈，如果仍然找不到，则继续回溯，直到找到（或没找到）路径节点并退出。

功能组件

当请求流量命中 API 请求路径进入服务端，具体处理逻辑由 DSL 中配置的一系列功能组件完成。网关提供了丰富的功能组件集成，包括链路追踪、实时监控、访问日志、参数校验、鉴权、限流、熔断降级、灰度分流等，如下图所示：



图 6

协议转换 & 服务调用

API 调用的最后一步，就是协议转换以及服务调用了。网关需要完成的工作包括：获取 HTTP 请求参数、Context 本地参数，拼装后端服务参数，完成 HTTP 协议到后端服务的协议转换，调用后端服务获取响应结果并转换为 HTTP 响应结果。

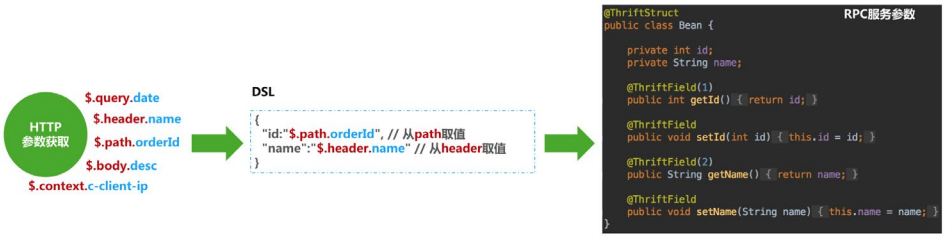


图 7

上图以调用后端 RPC 服务为例，通过 JsonPath 表达式获取 HTTP 请求不同部位的参数值，替换 RPC 请求参数相应部位的 Value，生成服务参数 DSL，最后借助 RPC 泛化调用完成本次服务调用。

2.2 高可用设计

Shepherd API 网关作为接入层的基础组件，高可用性一直是业务研发人员非常关心的部分。接下来。我们就来探索一下 Shepherd 在高可用设计方面的实践。

2.2.1 排除性能隐患

一个高可用的系统，预防故障的发生，首先要排除性能隐患，保证高性能。

Shepherd 对 API 请求做了全异步化处理，请求通过 Jetty IO 线程异步提交到业务处理线程池，调用后端服务使用 RPC 或 HTTP 框架的异步方式，释放了由于网络等待引起的线程占用，使线程数不再成为网关的瓶颈。下图是使用 Jetty 容器时，服务端的请求线程处理逻辑：

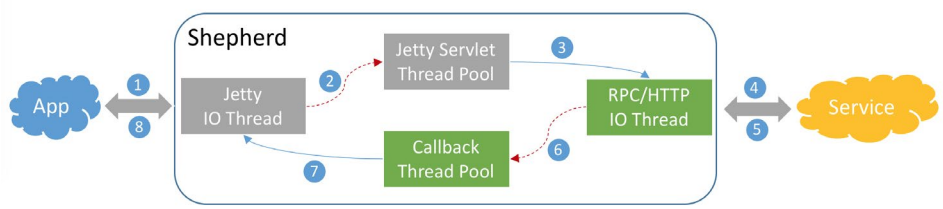


图 8

我们通过域名压测网关单机的端到端 QPS，发现 QPS 在超过 2000 时，会出现很多超时错误，而网关的服务端负载与性能却非常富余。调研发现，公司内其他的 Web 应用都存在这个问题，与 Oceanus 团队进行联合排查后，发现是 Nginx 与 Web 应用之间的长连接功能没有打开，且无法配置。Oceanus 团队经过紧急排期，研发并上线长连接功能后，Shepherd 端到端的 QPS 成功提升到了 10000 以上。

另外，我们对 Shepherd 服务端进行了 API 请求预热的优化，使得网关启动时能立刻达到最佳性能，减少毛刺的发生。然后，通过压测时的 CPU 热点排查，将性能瓶颈找出，减少主链路上的本地日志打印，对请求日志进行异步化、远程化改造。Shepherd 端到端的 QPS 再次提升 30% 以上。

在 Shepherd 服务上线稳定运行一年以后，我们再次对性能进行优化，并且做了一次网络框架升级，将 Jetty 容器全面替换为 Netty 网络框架，性能提升 10% 以上，Shepherd 端到端的 QPS 成功提升到 15000 以上。下图是使用 Netty 框架时，服务端的请求线程处理逻辑：

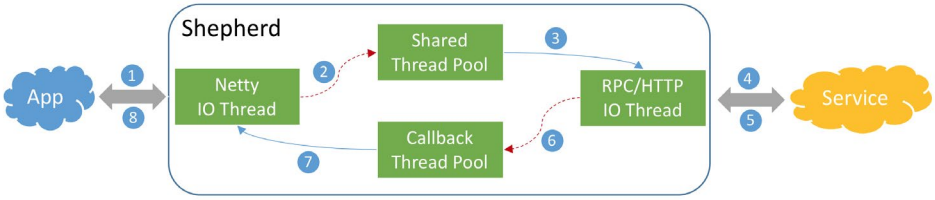


图 9

2.2.2 服务隔离

集群隔离

借鉴公司缓存、任务调度等成熟组件的经验，Shepherd 在设计之初，就考虑了按业务线维度进行集群隔离，也支持重要业务独立部署。如下图所示：

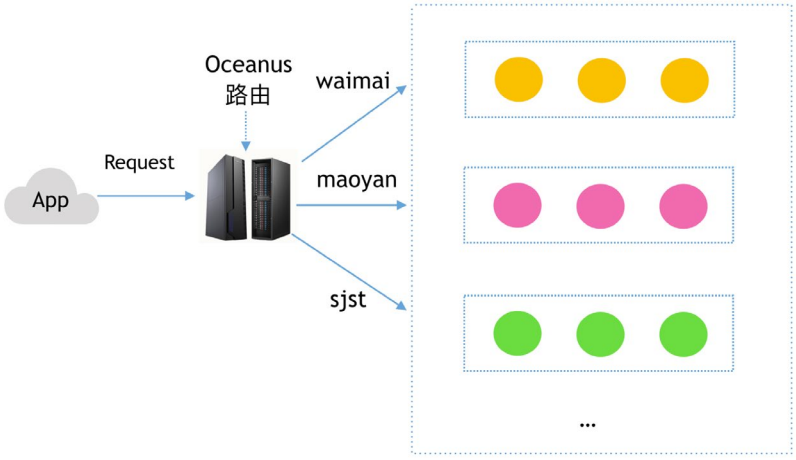


图 10

请求隔离

服务节点维度，Shepherd 支持请求的快慢线程池隔离。快慢线程池隔离主要用于一些使用了同步阻塞组件的 API，例如 SSO 鉴权、自定义鉴权等，可能导致长时间阻塞共享业务线程池。

快慢隔离的原理是统计 API 请求的处理时间，将请求处理耗时较长，超过容忍阈值的

API 请求隔离到慢线程池，避免影响其他正常 API 的调用。

除此之外，Shepherd 也支持业务研发人员配置自定义线程池进行隔离。具体的线程隔离模型如下图所示：

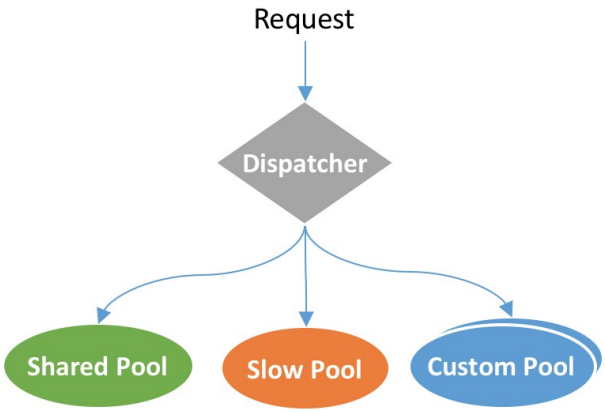


图 11

2.2.3 稳定性保障

Shepherd 提供了一些常规的稳定性保障手段，来保证自身和后端服务的可用性。如下图所示：

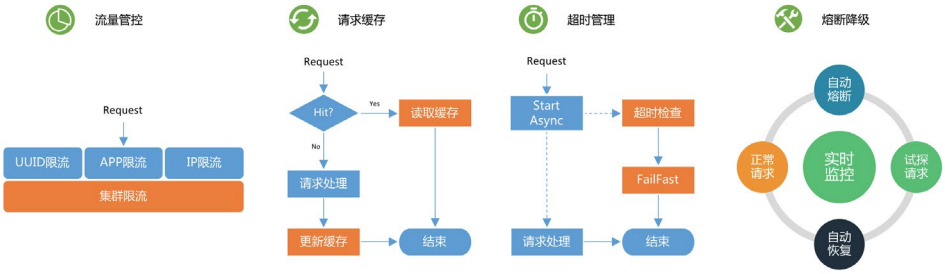


图 12

- **流量管控**：从用户自定义 UUID 限流、App 限流、IP 限流、集群限流等多个维度提供流量保护。
- **请求缓存**：对于一些幂等的、查询频繁的、数据及时性不敏感的请求，业务研

发人员可开启请求缓存功能。

- **超时管理**：每个 API 都设置了处理超时时间，对于超时的请求，进行快速失败的处理，避免资源占用。
- **熔断降级**：支持熔断降级功能，实时监控请求的统计信息，达到配置的失败阈值后，自动熔断，返回默认值。

2.2.4 请求安全

请求安全是 API 网关非常重要的能力，Shepherd 集成了丰富的安全相关的系统组件，包括有基础的请求签名、SSO 单点登录、基于 SSO 鉴权的 UAC/UPM 访问控制、用户鉴权 Passport、商家鉴权 EPassport、商家权益鉴权、反爬等等。业务研发人员只需要简单配置，即可使用。

2.2.5 可灰度

API 网关作为请求入口，往往肩负着请求流量灰度验证的重任。

灰度场景

Shepherd 在灰度能力上，支持灰度 API 自身逻辑，也支持灰度下游服务，也可以同时灰度 API 自身逻辑和下游服务。如下图所示：

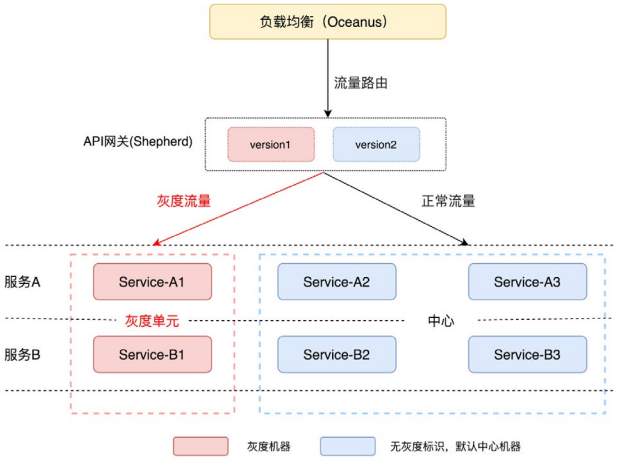


图 13

灰度 API 自身逻辑时，通过将流量分流到不同的 API 版本实现灰度能力；灰度下游服务时，通过给流量打标，分流到指定的下游灰度单元中。

灰度策略

Shepherd 支持丰富的灰度策略，可以按照比例数灰度，也可以按照特定条件灰度。

2.2.6 监控告警

立体化监控

Shepherd 提供 360 度的立体化监控，从业务指标、机器指标、JVM 指标提供 7x24 小时的专业守护，如下表：

	监控模块	主要功能
1	统一监控 Raptor	实时上报请求调用信息、系统指标，负责应用层（JVM）监控、系统层（CPU、IO、网络）监控
2	链路追踪 Mtrace	负责全链路参数透传、全链路追踪监控
3	日志监控 Logscan	监控本地日志异常关键字：如 5xx 状态码、空指针异常等
4	远程日志中心	API 请求日志、Debug 日志、组件日志等可上报远程日志中心
5	健康检查 Scanner	对网关节点进行心跳检测和 API 状态检测，及时发现异常节点和异常 API

多维度告警

有了全面的监控体系，自然少不了配套的告警机制，主要的告警能力包括：

	告警类型	触发时机
1	限流告警	API 请求达到限流规则阈值触发限流告警
2	请求失败告警	鉴权失败、请求超时、后端服务异常等触发请求失败告警
3	组件异常告警	自定义组件处理耗时长、失败率高告警
4	API 异常告警	API 发布失败、API 检查异常时触发 API 异常告警
5	健康检查失败告警	API 心跳检查失败、网关节点不通时触发健康检查失败告警

2.2.7 故障自愈

Shepherd 服务端接入了弹性伸缩模块，可根据 CPU 等指标进行快速扩容、缩容。除此之外，还支持快速摘除问题节点，以及更细粒度的问题组件摘除。

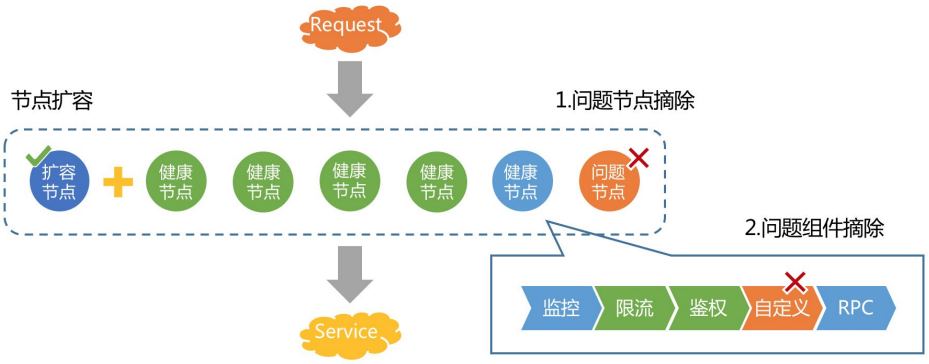


图 14

2.2.8 可迁移

对于一些已经在对外提供 API 的 Web 服务，业务研发人员为了减少运维成本和后续的研发提效，考虑将其迁移到 Shepherd API 网关。

对于一些非核心 API，可以考虑使用 Oceanus 的灰度发布功能直接迁移。但是对于一些核心 API，上面的灰度发布功能是机器级别的，粒度较大，不够灵活，不能很好的支持灰度验证过程。

解决方案

Shepherd 为业务研发人员提供了一个灰度 SDK，接入 SDK 的 Web 服务，可在识别灰度流量后转发到 Shepherd API 网关进行验证。

灰度哪些 API、灰度百分比可以在 Shepherd 管理端动态调节，实时生效，业务研发人员还可以通过 SPI 的方式自定义灰度策略。灰度验证通过后，再把 API 迁移到 Shepherd API 网关，保障迁移过程的稳定性。

灰度过程

灰度前：在 Shepherd 管理平台创建 API 分组，域名配置为目前使用的域名。在 Oceanus 上，原域名规则不变。

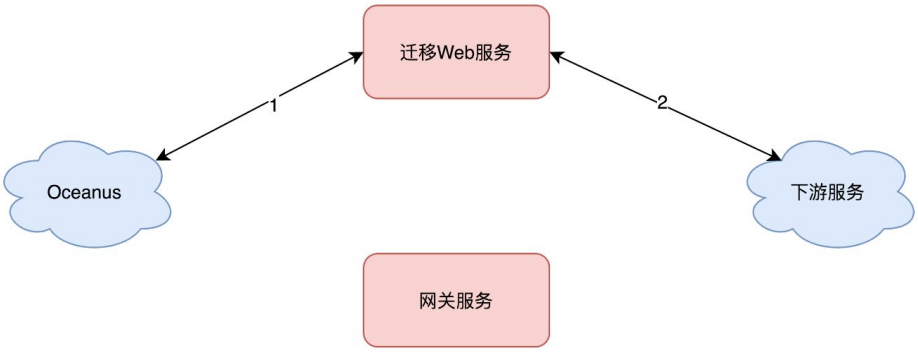


图 15

灰度中：在 Shepherd 管理平台开启灰度功能，灰度 SDK 将灰度流量转发到网关服务，进行验证。

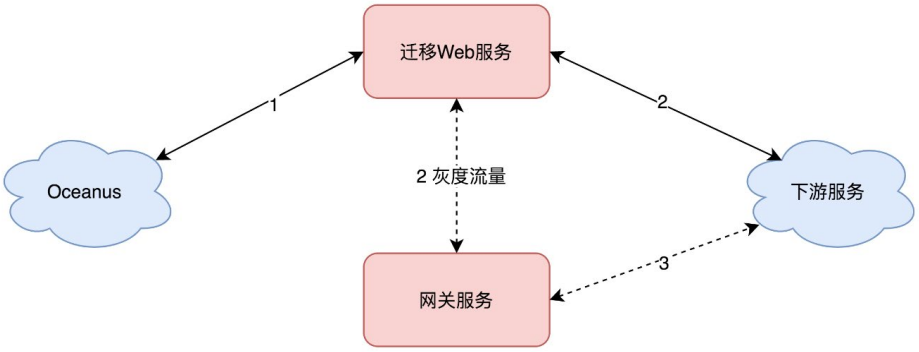


图 16

灰度后：通过灰度流量验证 Shepherd 上的 API 配置符合预期后再迁移。

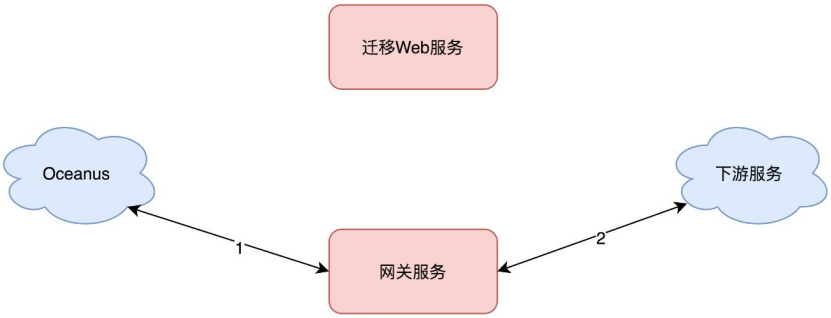


图 17

2.3 易用性设计

Shepherd API 网关的功能强大且复杂，易用性对业务研发人员来说至关重要，我们着重来看下自动生成 DSL、API 操作提效的解决方案。

2.3.1 自动生成 DSL

业务研发人员在实际使用网关管理平台时，我们尽量通过图形化的页面配置来减轻 DSL 的编写负担。但服务参数转换的 DSL 配置，仍然需要业务研发人员手工编写。一般来说，生成服务参数 DSL 的流程是：

- 引入服务的接口包依赖。
- 拿到服务参数类定义。
- 编写 Testcase 生成 JSON 模板。
- 填写参数映射规则。
- 最后手工录入管理平台，发布 API。

整个过程非常繁琐，且容易出错。如果需要录入的 API 多达几十上百个，全部由业务研发人员手工录入的效率是非常低下的。

解决方案

那么能不能将服务参数 DSL 的生成过程给自动化呢？ 答案是可以的，业务 RD 只需在网关录入 API 文档信息，然后录入服务的 Appkey、服务名、方法名信息，Shep-

herd 管理端会从最新发布的服务框架控制台获取到服务参数的 JSON Schema 信息，JSON Schema 定义了服务参数的类型和结构信息，管理端可根据这些信息，自动生成服务参数的 JSON Mock 数据。结合 API 文档的信息，自动替换参数名相同的 Value 值。 这套 DSL 自动生成方案，使用过程中对业务透明、标准化，业务方只需升级最新版本服务框架即可使用，极大提升研发效率，目前受到业务研发人员的广泛好评。

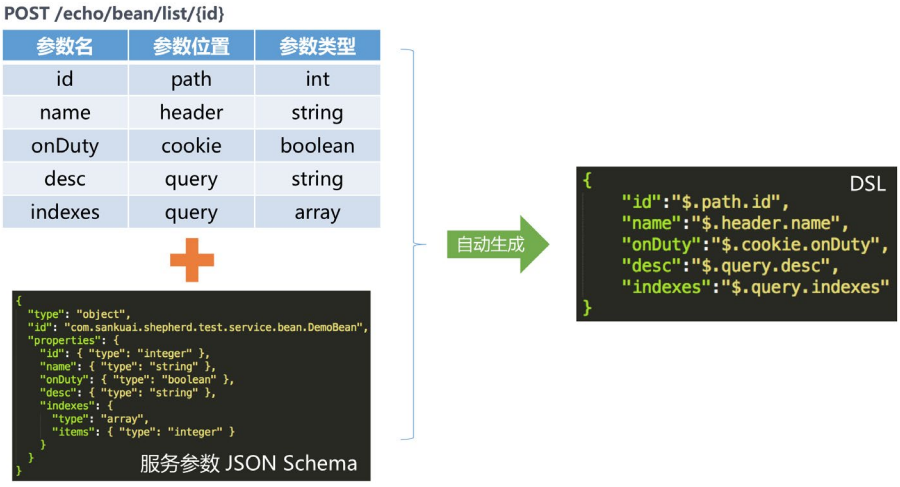


图 18

2.3.2 API 操作提效

快速创建 API

API 网关的核心能力是建立在 API 配置的基础上的，但提供强大功能的同时带来了较高的复杂性，不少业务研发人员吐槽 API 配置太繁琐，学习成本高。快速创建 API 的功能应运而生，业务研发人员只需要提供少量的信息就可以创建 API。快速创建 API 的功能当前分为 4 种类型（后端 RPC 服务 API、后端 HTTP 服务 API、SSO CallBack API、Nest API），未来会根据业务应用场景的不同，提供更多的快速创建 API 类型。

批量操作

业务研发人员在 API 网关上，需要管理非常多的业务分组，每个业务分组，最多可以有 200 个 API 配置，多个 API 可能有很多相同的配置，如组件配置，错误码配置和跨域配置的。每个 API 对于相同的配置都要配置一遍，操作重复度很高。因此 Shepherd 支持批量操作多个 API：勾选多个 API 后，通过【批量操作】功能可一次性完成多个 API 配置更新，降低业务重复配置的操作成本。

API 导入导出

Shepherd 提供在不同研发环境相互导入导出 API 的能力，业务研发人员在线下测试完成后，只需要使用 API 导入导出功能，即可将配置导出到线上生产环境，避免重复配置。

2.4 可扩展性设计

一个设计良好的基础组件，除了能提供强大的基础能力，还需要有良好的扩展能力。Shepherd 的可扩展性主要体现在：支持自定义组件、服务编排的能力。

2.4.1 自定义组件

Shepherd 提供了丰富的系统组件完成鉴权、限流、监控能力，能够满足大部分的业务需求。但仍有一些特殊的业务需求，如自定义验签、自定义结果处理等。Shepherd 通过提供加载自定义组件能力，支持业务完成一些自定义逻辑的扩展。

下图是自定义组件实现的一个实例。getName 中填写自定义组件申请时的名称，invoke 方法中实现自定义组件的业务逻辑，如继续执行、进行页面跳转、直接返回结果、抛出异常等。

```
public class TestFilter implements InvocationFilter<BaseProcessContext> {
    @Override
    public String getName() { return "test_filter"; }

    @Override
    public InvocationResponse invoke(InvocationHandler handler,
                                     BaseProcessContext processContext)
            throws Throwable {
        String opType = processContext.getRequest().getQueries().get("opType");

        if ("继续执行".equalsIgnoreCase(opType)) {
            return handler.handle(processContext);

        } else if ("页面跳转".equalsIgnoreCase(opType)) {
            return APIUtils.redirect( redirectUrl: "http://www.meituan.com", processContext);

        } else if ("直接返回结果".equalsIgnoreCase(opType)) {
            Response httpRes = new Response<>().success("默认结果");
            return APIUtils.writeSuccessResponse(httpRes, processContext);

        } else {
            throw new ParamCheckException("错误的操作类型");
        }
    }
}
```

图 19

目前 Shepherd 通过自定义组件已经成功支持了美团优选、外卖、餐饮、打车等重要业务，接入的自定义组件数量有 200 多个。

2.4.2 服务编排

一般情况下，网关上配置的一个 API 对应后端一个 RPC 或者 HTTP 服务。如果调用端有聚合和编排后端服务的需求，那么有多少后端服务，就必须发起多少次 HTTP 的请求调用。由此就会带来一些问题，调用端的 HTTP 请求次数过多，效率低，在调用端聚合服务的逻辑过重。

服务编排的需求应运而生，服务编排是对既有服务进行编排调用，同时对获取的数据进行处理。主要应用在数据聚合场景：一次 HTTP 请求返回的数据需要调用多个或多次服务（RPC 或 HTTP）才能获取到完整的结果。

经过前期调研，公司已经有一套成熟的服务编排框架，由客服团队开发的海盗组件（参见《海盗中间件：美团服务体验平台对接业务数据的最佳实践》一文），也是美团

内部的公共服务。

因此我们与海盗团队合作，设计了 Shepherd 的服务编排支持方案。海盗通过独立部署的方式提供服务编排能力，Shepherd 与海盗之间通过 RPC 进行调用。这样可以解耦 Shepherd 与海盗，避免因服务编排能力影响集群上的其他服务，同时多一次 RPC 调用并不会有明显耗时增加。使用上对业务研发人员也是透明的，非常方便，业务研发人员在管理端配置好服务编排的 API，通过配置中心同时下发到 Shepherd 服务端和海盗服务上，即可开始使用服务编排能力。整体的交互架构图如下：

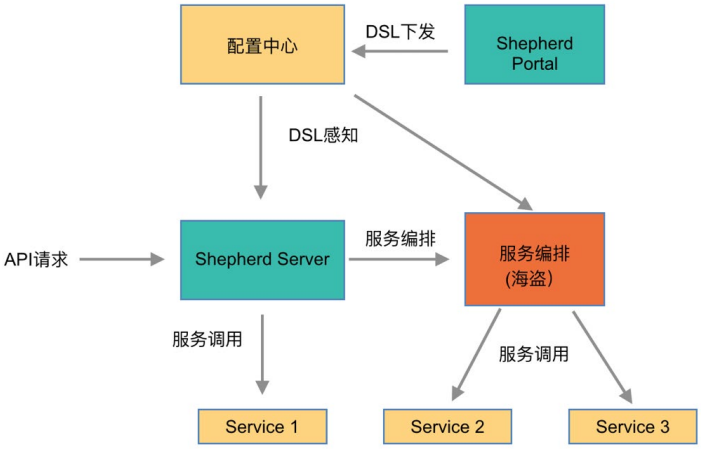


图 20

三、未来规划

目前接入 Shepherd API 网关的 API 数量超过 18000 多个，线上运行的集群数量 90 多个，日均总调用次数在百亿以上。随着 Shepherd API 网关业务规模的不断增长，对我们的可用性、易用性、可扩展性必将提出更高的要求。未来一年，Shepherd 的规划重点包括有云原生架构演进、静态网站托管、组件市场等。

3.1 云原生架构演进

Shepherd API 网关的云原生架构演进有三个目标：简化接入网关步骤，提升业

务研发人员的研发效率；减小服务端 War 包大小，提升安全性和稳定性；接入 Serverless 弹性，降低成本，提高资源利用率。

为了实现这个三个目标，我们计划整体迁移网关服务到公司的 Serverless 服务 Nest（参见《美团 Serverless 平台 Nest 的探索与实践》一文）上，同时通过抽取 Shepherd 核心功能到 SDK 的方式集成到业务的网关集群，业务研发人员可以只选择自己需要使用的自定义组件，从而大幅减小服务端的 War 包大小。

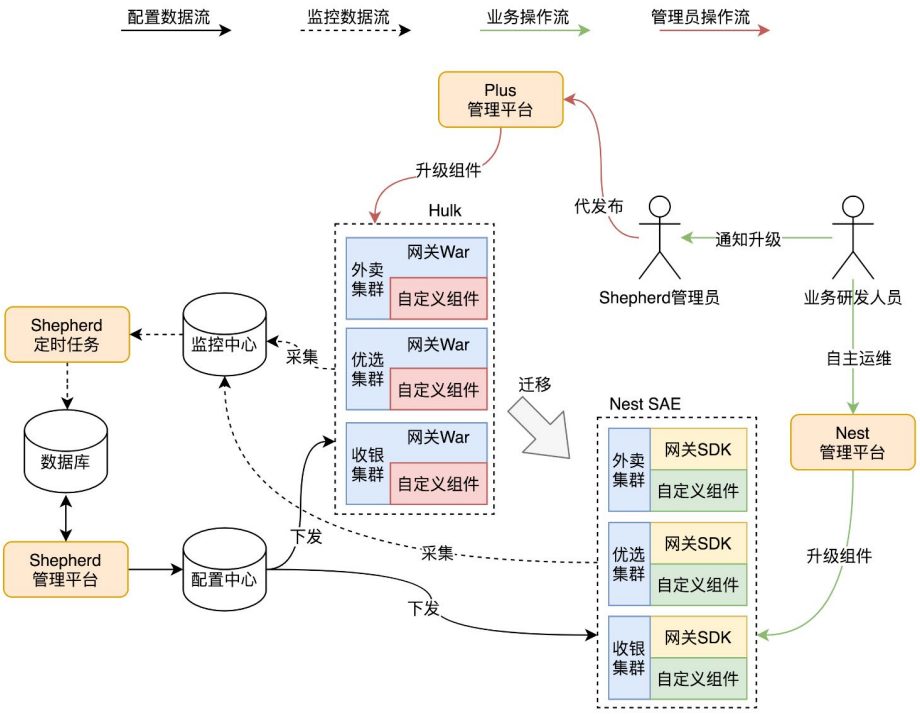


图 21

3.2 静态网站托管

依托 Shepherd API 网关实现静态网站托管的目标是：**建设通用的静态网站托管解决方案，为开发者提供便捷、稳定、高扩展性的静态网站托管服务**。

静态网站托管解决方案能为业务研发人员提供的主要功能包括：托管静态网站资源，

包括存储及访问；管理应用生命周期，包括自定义域配置以及身份验证和授权；CI/CD 集成等。

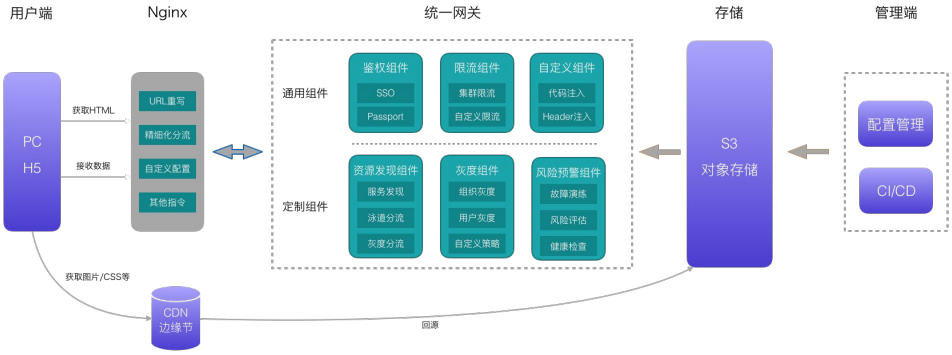


图 22

3.3 组件市场

Shepherd API 网关组件市场的目标是：**合作共赢，形成开发生态，业务研发人员可将开发的自定义组件提供给其他有需要的业务研发团队使用**。

我们希望让业务研发人员参与到自定义组件的开发，完善使用文档后设置为公共组件，开放给所有使用 Shepherd 的业务研发人员使用，避免重复造轮子。

作者简介

充泽、志洋、李敏等，均来自美团基础技术部 – 基础架构团队。

招聘信息

美团基础技术部 – 基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

TensorFlow 在推荐系统中的分布式训练优化实践

作者：逸帆 家恒 峥少 鹏鹏 永宇 正阳 黄军

1. 背景

TensorFlow（下文简称 TF）是谷歌推出的一个开源深度学习框架，在美团推荐系统场景中得到了广泛的使用。但 TensorFlow 官方版本对工业级场景的支持，目前做得并不是特别的完善。美团在大规模生产落地的过程中，遇到了以下几方面的挑战：

- 所有参数都是用 Variable 表达，对于百亿以上的稀疏参数开辟了大量的内存，造成了资源的浪费；
- 只支持百级别 Worker 的分布式扩展，对上千 Worker 的扩展性较差；
- 由于不支持大规模稀疏参数动态添加、删除，增量导出，导致无法支持 Online Learning；
- 大规模集群运行时，会遇到慢机和宕机；由于框架层不能处理，导会致任务运行异常。

以上这些问题，并不是 TensorFlow 设计的问题，更多是底层实现的问题。考虑到美团大量业务的使用习惯以及社区的兼容性，我们基于原生 TensorFlow 1.x 架构与接口，从大规模稀疏参数的支持、训练模式、分布式通信优化、流水线优化、算子优化融合等多维度进行了深度定制，从而解决了该场景的核心痛点问题。

首先新系统在支持能力层面，目前可以做到千亿参数模型，上千 Worker 分布式训练的近线性加速，全年样本数据能够 1 天内完成训练，并支持 Online Learning 的能力。同时，新系统的各种架构和接口更加友好，美团内部包括美团外卖、美团优选、美团搜索、广告平台、大众点评 Feeds 等业务部门都在使用。**本文将重点介绍大规模分布式训练优化的工作**，希望对大家能够有所帮助或启发。

2. 大规模训练优化挑战

2.1 业务迭代带来的挑战

随着美团业务的发展，推荐系统模型的规模和复杂度也在快速增长，具体表现如下：

- **训练数据**：训练样本从到百亿增长到千亿，增长了近 10 倍。
- **稀疏参数**：个数从几百到几千，也增长了近 10 倍；总参数量从几亿增长到百亿，增长了 10~20 倍。
- **模型复杂度**：越来越复杂，模型单步计算时间增长 10 倍以上。

对于大流量业务，一次训练实验，从几个小时增长到了几天，而此场景一次实验保持在 1 天之内是基本的需求。

2.2 系统负载分析

2.2.1 问题分析工具链

TensorFlow 是一个非常庞大的开源项目，代码有几百万行之多，原生系统的监控指标太粗，且不支持全局的监控，如果要定位一些复杂的性能瓶颈点，就比较困难。我们基于美团已经开源的监控系统 CAT[2]，构建了 TensorFlow 的细粒度监控链路（如下图 1 所示），可以精准定位到性能的瓶颈问题。

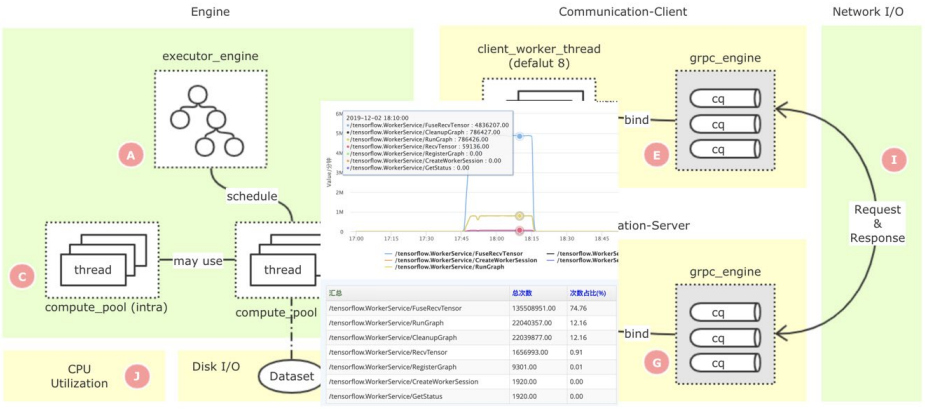


图 1　TensorFlow PS 架构全链路监控

同时，在性能优化的过程中，会涉及到大量的性能测试和结果分析，这也是一个非常耗费人力的工作。我们抽象了一套自动化的实验框架（如下图 2 所示），可以自动化、多轮次地进行实验，并自动采集各类监控指标，然后生成报告。

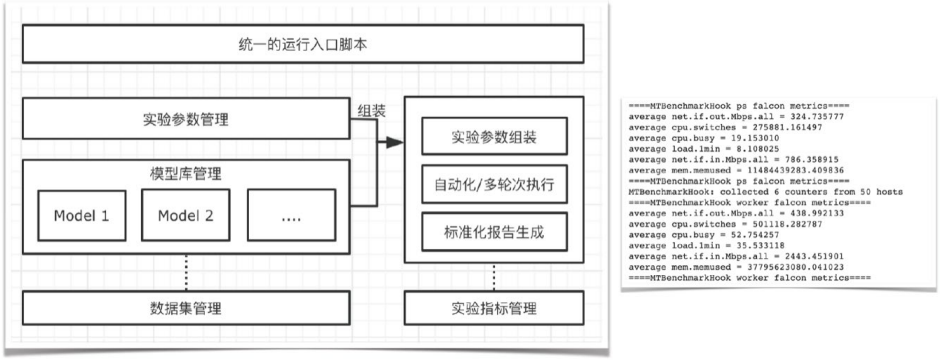


图 2　自动化实验框架

2.2.2 业务视角的负载分析

在推荐系统场景中，我们使用了 TensorFlow Parameter Server[3]（简称 PS）异步训练模式来支持业务分布式训练需求。对于这套架构，上述的业务变化会带来什么样的负载变化？如下图 3 所示：

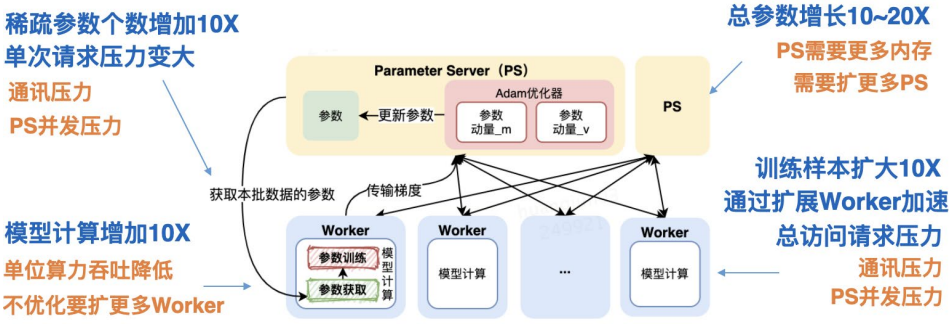


图 3　TensorFlow PS 架构大规模训练负载分析

总结来看，主要包括通信压力、PS 并发压力、Worker 计算压力。对于分布式系统来说，通常是通过横向扩展来解决负载问题。虽然看来起可以解决问题，但从实验结

果来看，当 PS 扩展到一定数量后，单步训练时间反而会增加，如下图 4 所示：

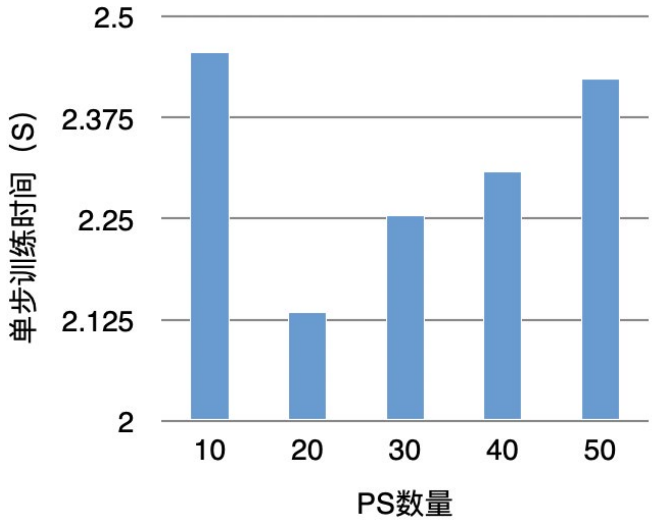


图 4　扩展 PS 提升训练性能实验

导致这种结果的核心原因是：Worker 单步训练需要和所有的 PS 通信同步完成，每增加 1 个 PS 要增加 N 条通信链路，这大大增加了链路延迟（如下图 5 所示）。而一次训练要执行上百万、上千万步训练。最终导致**链路延迟超过了加 PS 算力并发的收益**。

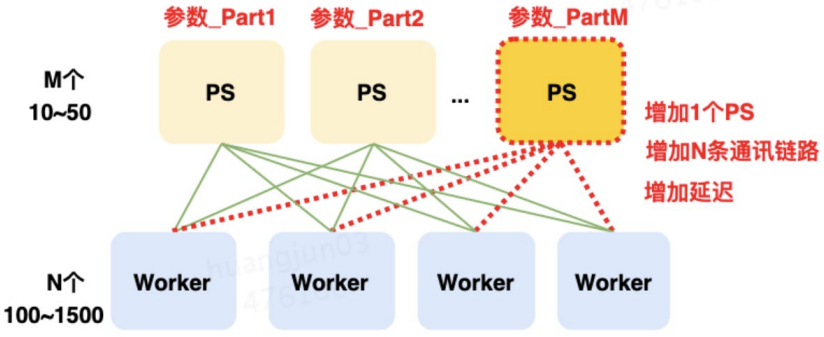


图 5　增加 PS 带来的链路开销

而对于这个系统，优化的核心难点在于：**如何在有限的 PS 实例下，进行分布式计算的优化**。

3. 优化实践

3.1 大规模稀疏参数介绍

对于推荐系统模型，绝大多数参数都是稀疏参数，而对稀疏参数来说有一个非常重要的操作是 Embedding，这个操作通常也是负载最重的，也是后续优化的重点。由于我们对稀疏参数进行了重新定义，后续的优化也基于此之上，所以我们先介绍一下这部分的工作。

在原生的 TensorFlow 中构建 Embedding 模块，用户需要首先创建一个足够装得下所有稀疏参数的 Variable，然后在这个 Variable 上进行 Embedding 的学习。然而，使用 Variable 来进行 Embedding 训练存在很多弊端：

- Variable 的大小必须提前设定好，对于百亿千亿的场景，该设定会带来巨大的空间浪费；
- 训练速度慢，无法针对稀疏模型进行定制优化。

我们首先解决了有无的问题，使用 HashTable 来替代 Variable，将稀疏特征 ID 作为 Key，Embedding 向量作为 Value。相比原生使用 Variable 进行 Embedding 的方式，具备以下的优势：

- HashTable 的大小可以在训练过程中自动伸缩，避免了开辟冗余的存储空间，同时用户无需关注申请大小，从而降低了使用成本。
- 针对 HashTable 方案实施了一系列定制优化，训练速度相比 Variable 有了很大的提高，可以进行千亿规模模型的训练，扩展性较好。
- 得益于稀疏参数的动态伸缩，我们在此基础上支持了 Online Learning。
- API 设计上保持与社区版本兼容，在使用上几乎与原生 Variable 一致，对接成本极低。

简化版的基于 PS 架构的实现示意如下图 6 所示：

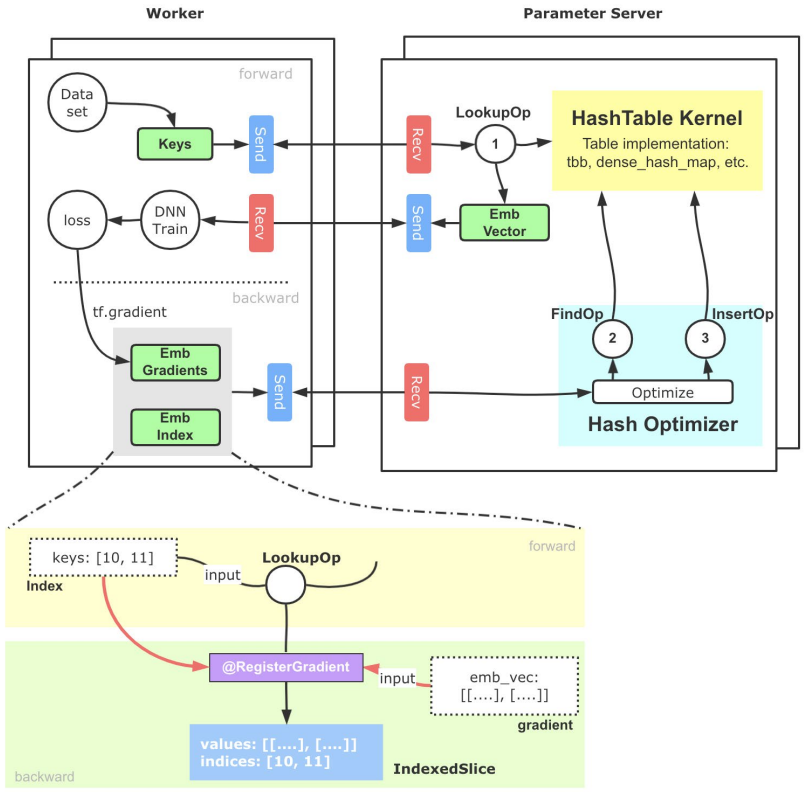


图 6　支撑大规模稀疏参数的 HashTable 方案

核心流程大致可以分为以下几步：

- 稀疏特征 ID（通常我们会提前完成统一编码的工作）进入 Embedding 模块，借助 TensorFlow 搭建的 Send-Recv 机制，这些稀疏特征 ID 被拉取到 PS 端，PS 端上的 Lookup 等算子会实际从底层 HashTable 中查询并组装 Embedding 向量。
- 上述 Embedding 向量被 Worker 拉回进行后续训练，并通过反向传播计算出这部分参数的梯度，这些梯度进一步被位于 PS 端的优化器拉回。
- PS 端的优化器首先调用 Find 算子，从 HashTable 获取到梯度对应的原始稀疏参数向量和相应的优化器参数，最终通过优化算法，完成对 Embedding 向量和优化器参数的更新计算，再通过 Insert 算子插入 HashTable 中。

3.2 分布式负载均衡优化

这部分优化，是分布式计算的经典优化方向。PS 架构是一个典型的“水桶模型”，为了完成一步训练，Worker 端需要和所有 PS 完成交互，因此 PS 之间的平衡就显得非常重要。但是在实践中，我们发现多个 PS 的耗时并不均衡，其中的原因，既包括 TensorFlow PS 架构简单的切图逻辑（Round-Robin）带来的负载不均衡，也有异构机器导致的不均衡。

对于推荐模型来说，我们的主要优化策略是，把所有稀疏参数和大的稠密参数自动、均匀的切分到每个 PS 上，可以解决大多数这类问题。而在实践过程中，我们也发现一个比较难排查的问题：原生 Adam 优化器，实现导致 PS 负载不均衡。下面会详细介绍一下。

在 Adam 优化器中，它的参数优化过程需要两个 β 参与计算，在原生 TensorFlow 的实现中，这两个 β 是所有需要此优化器进行优化的 Variabl（或 HashTable）所共享的，并且会与第一个 Variable（名字字典序）落在同一个 PS 上面，这会带来一个问题：每个优化器只拥有一个 β_1 和一个 β_2，且仅位于某个 PS 上。因此，在参数优化的过程中，该 PS 会承受远高于其他 PS 的请求，从而导致该 PS 成为性能瓶颈。

$$
\begin{aligned}
&t = t + 1 \\
&\alpha_t := \alpha * \sqrt{1 - \beta_2^t} / (1 - \beta_1^t) \\
&m_t := \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \\
&v_t := \beta_2 * v_{t-1} + (1 - \beta_2) * g_t * g_t \\
&variable = variable - \frac{\alpha_t}{(\sqrt{v_t} + \epsilon)} * m_t
\end{aligned}
$$

图 7　Adam 优化算法

但是通过观察 Adam 的优化算法，我们可以看到 β_1 和 β_2 都是常量，且蓝色高亮的部分都是相对独立的计算过程，各个 PS 之间可以独立完成。基于这样的发现，优化

的方法也就非常直观了，我们为每一个 PS 上的 Adam 优化器冗余创建了 β 参数，并在本地计算 t 和 alpha 值，去除了因此负载不均导致的 PS 热点问题。

该优化所带来的提升具备普适性且效果明显，在美团内部某业务模型上，通过 β 热点去除可以带来 9% 左右的性能提升。此外，由于摆脱了对 β 的全局依赖，该优化还能提高 PS 架构的可扩展性，在扩增 Worker 数量的时候相比之前会带来更好的加速比。

3.3 通信优化

通过 2.2 章节的分析可知，系统的通信压力也非常大，我们主要基于 RDMA 做了通信优化的工作。首先简单介绍一下 RDMA，相比较于传统基于套接字 TCP/IP 协议栈的通信过程，RDMA 具有零拷贝、内核旁路的优势，不仅降低了网络的延迟，同时也降低了 CPU 的占用率，RDMA 更适合深度学习模型的相关通信过程。

RDMA 主要包括三种协议 Infiniband、RoCE(V1, V2)、iWARP。在美团内部的深度学习场景中，RDMA 通信协议使用的是 RoCE V2 协议。目前在深度学习训练领域，尤其是在稠密模型训练场景（NLP、CV 等），RDMA 已经是大规模分布式训练的标配。然而，在大规模稀疏模型的训练中，开源系统对于 RDMA 的支持非常有限，TensorFlow Verbs[4] 通信模块已经很长时间没有更新了，通信效果也并不理想，我们基于此之上进行了很多的改进工作。

经过优化后的版本，在 1TB Click Logs[5] 公开数据集、DLRM[6] 模型、100 个 Worker 以上的训练，性能提升了 20%~40%。在美团的多个业务模型上，对比 TensorFlow Seastar[7] 改造的通信层实现也有 10%~60% 的速度提升。同时也把我们的工作回馈给了社区。

3.3.1 Memory Registration 优化

RDMA 有三种数据传输的方式 SEND/RECV、WRITE、READ，其中 WRITE、READ 类似于数据发送方直接在远程 Memory 进行读写，Receiver 无法感知，WRITE 和 READ 适用于批量数据传输。在 TensorFlow 内部，基于 RDMA 的数据传输方式使用的是 WRITE 单边通信模式。

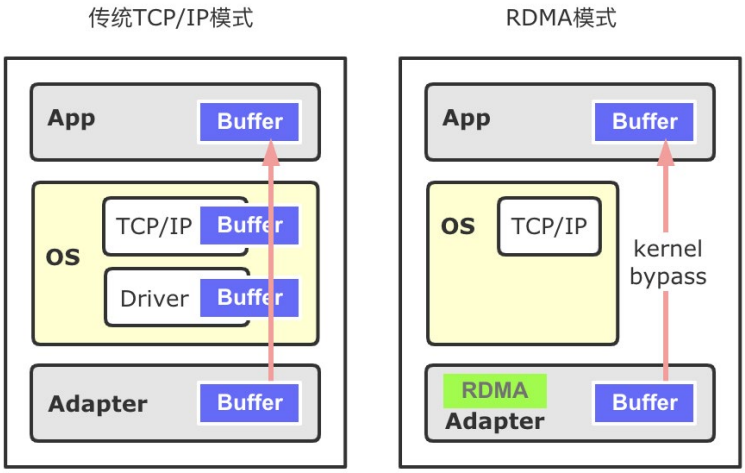


图 8　RDMA 传输方式

在 RDMA 传输数据时，需要提前开辟内存空间并将其注册到网卡设备上（Memory Registration 过程，下称 MR），使得这片空间可以被网卡直接操作。开辟新的内存并注册到设备上，整个过程是比较耗时的。下图 9 展示了不同大小的内存绑定到网卡设备上的耗时，可以看到随着注册内存的增大，绑定 MR 的耗时迅速增加。

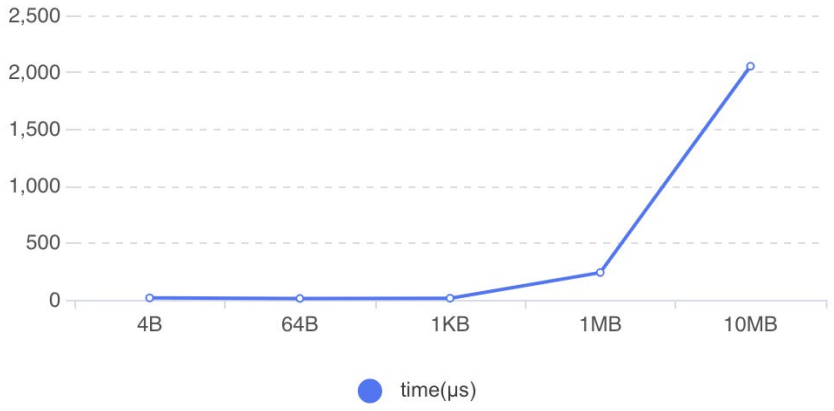


图 9　MR 过程开销

社区版 Tensorflow RDMA 实现，Tensor 创建依旧沿用了统一的 BFC Allocator，并将所有创建的 Tensor 都注册到 MR 上。正如上面所提到的，MR 的注册绑定具

有性能开销，高频、大空间的 MR 注册会带来显著的性能下降。而训练过程中的 Tensor，只有那些涉及到跨节点通信的 Tensor 有必要进行 MR，其余 Tensor 并不需要注册到 MR。因此，优化的方法也就比较直接了，我们识别并管理那些通信 Tensor，仅对这些跨节点通信的 Tensor 进行 MR 注册就好了。

3.3.2 RDMA 静态分配器

RDMA 静态分配器是上一个 MR 注册优化的延伸。通过 Memory Registration 优化，去除非传输 Tensor 的 MR 注册，我们降低了 MR 注册数量。但是在稀疏场景大规模的训练下，并行训练的 Worker 常有几百上千个，这会带来新的问题：

- PS 架构中的 PS 和 Worker 互为 Client-Server，这里以 PS 端为例，当 Worker 数目增加到上千个时，Worker 数目的增多，造成 PS 端 MR 注册频次还是非常高，增加了内存分配注册的耗时。
- 由于稀疏场景不同 Step 之间同一个算子输出 Tensor 的形状可能发生变化，导致了创建的 MR 可复用性较差，带来了较高的内存碎片和重复注册 MR 开销。

针对上面的问题，我们引入了 MR 静态分配器的策略。

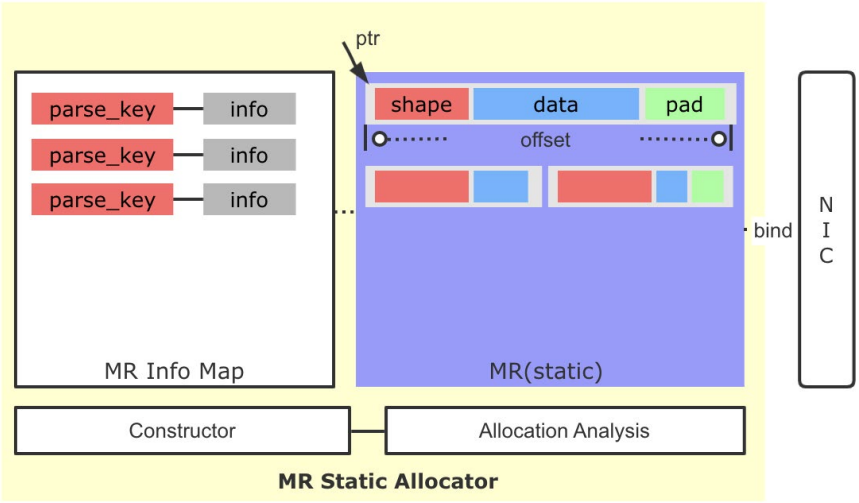


图 10　MR 静态分配器

这里核心的设计思路为：

- 虽然稀疏场景同一个算子输出 Tensor 的 Shape 存在变化的可能，但是整体变化幅度可控，通过监控与分析，是可以找到一个较为稳定的内存大小，满足多 Step 间 Tensor 的存储需求。
- 基于上面的信息，我们修改了原有逐 Tensor(Request) 的 MR 申请策略，通过一次性预申请一块较大的空间并注册到网卡端，后续通过自己维护的分配策略进行空间的分配，大大降低了 MR 申请的频率，绝大多数情况下，训练全过程中只需要一次 MR 注册申请即可。
- 我们引入了一种简单的交换协议，将传输 Tensor 的 Shape，Data 打包到一起，写到 Client 端。Client 端根据协议，解析出 Tensor 大小，并最终读取 Data，避免了原生实现中因 Tensor 的 Shape 变化而产生的多次协商过程。

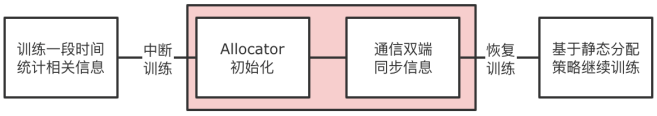


图 11　MR 静态分配器构造流程

具体到实现中，我们引入了 Allocation Analysis 模块，在训练开始的一段时间，我们会对分配的历史数据进行分析，以得到一个实际预开辟 MR 大小以及各个 Tensor 的预留空间大小。然后我们会暂停训练的进程，启动 Allocator 的构造过程，包括 MR 的创建以及通信双端的信息同步。利用相关信息构造 MR Info Map，这个 Map 的 Key 是传输 Tensor 的唯一标记（ParsedKey，计算图切图时确定），Info 结构体中包含了本地地址指针、offset 大小、ibv_send_wr 相关信息等。然后恢复训练，后续 Tensor 的传输就可以使用静态开辟好的 MR 进行收发，也免去了因 Shape 变化而产生的多次协商过程。

3.3.3 Multi RequestBuffer 与 CQ 负载均衡

TensorFlow 社区版的 RDMA 通信过程，不仅仅包含上面 Tensor 数据的发送和接收过程，还包括传输相关的控制消息的发送和接收过程，控制消息的发送和接收过程

同样是使用了 ibv_post_send 和 ibv_post_recv 原语。原生的控制流实现存在一些瓶颈，在大规模训练时会限制控制流的吞吐，进而影响数据收发的效率。具体体现在：

- 请求的发送通过同一片 RequestBuffer 内存进行写出，多个 Client 的请求均依赖这一片 Buffer，也就导致到控制流信息实际是串行发送的，只有等到对端的 Ack 信息后，才可以下一个 Request 的写出，限制了请求的发送吞吐。
- 在 Client 端需要轮询 RDMA Completion Queue 来获得请求的到达，以及相关状态的变更。原生实现仅有一个 Completion Queue，单线程进行轮询处理，在大规模分布式训练中，限制了应答的效率。

针对上面的问题，我们采用了 Multi RequestBuffer 与 CQ 负载均衡优化，破除了在请求发送和请求应答环节可能存在的吞吐瓶颈。

3.3.4 Send-Driven & Rendezvous-Bypass

对于 Tensorflow PS 架构熟悉的同学会了解，一整张计算图被切割为 Worker 端和 PS 端后，为了使两张计算图能够彼此交换数据，建立了基于 Rendezvous（汇合点）机制的异步数据交换模式。如下图 12 所示：

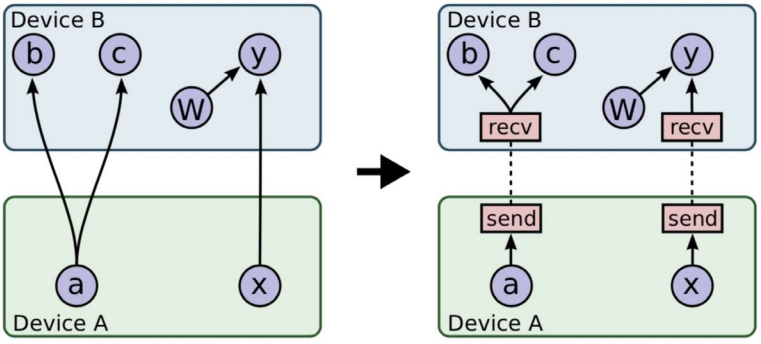


图 12　TensoFlow 切图之 Send-Recv 对添加

基于上图的切图逻辑，Recv 算子代表着这一侧计算图有 Tensor 的需求，而 Tensor 的生产者则位于与之配对的另一设备上的 Send 算子背后。

在具体实现上，Tensorflow 实现了 Recv-Driven 的数据交换模式，如上图所

示，位于 DeviceA 和 DeviceB 的两张计算图会异步并发的执行，位于 DeviceB 的 Recv 执行时会发起一条 RPC 请求发往 DeviceA，DeviceA 收到请求后，会将请求路由到 Rendezvous 中，如果在当中发现所需要的数据已经生产好，并被 Send 算子注册了进来，那么就地获取数据，返回给 DeviceB；如果此时数据还没有生产好，则将来自于 DeviceB 的 Recv 请求注册在 Rendezvous 中，等待后续 DeviceA 生产好后，由 Send 算子发送过来，找到注册的 Recv，触发回调，返回数据给 DeviceB。

我们看到，汇合点机制优雅地解决了生产者消费者节奏不同情况下数据交换的问题。不过 Recv-Driven 的模式也引入了两个潜在的问题：

- 据我们的观察，在实际业务模型中，在 Rendezvous 中 Recv 算子等待 Send 算子的比例和 Send 算子等待 Recv 算子的比例相当，也就是说对于 Send 等到 Recv 的数据，在 Send 准备好的那一刹那就可以发给对端，但是由于机制实现问题，还是等待 Recv 算子过来，才将数据拉取回去，通信过程耗时较长。
- Rendezvous 作为一个数据交换的热点，它内部的逻辑开销并不低。

针对上面提到的问题，我们在 RDMA 上实现了另外一种数据交换的模式，叫做 Send-Driven 模式。与 Recv-Driven 模式相对，顾名思义就是有 Send 算子直接将数据写到 Recv 端，Recv 端接收数据并注册到本地 Rendezvous 中，Recv 算子直接从本地的 Rendezvous 中获取数据。具体流程如下图 13 所示：

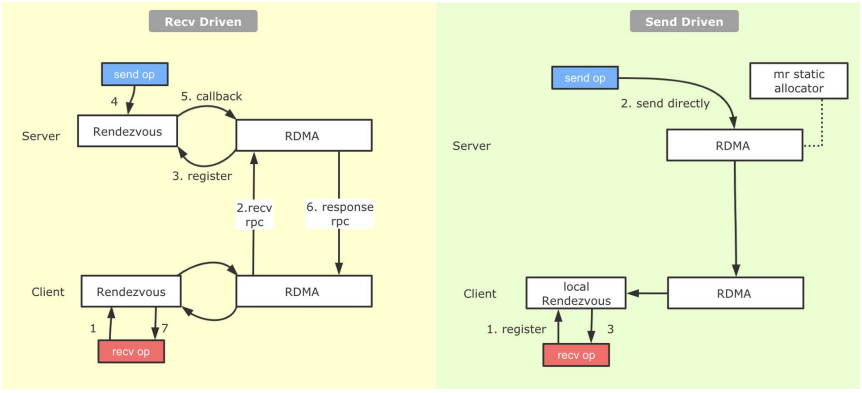


图 13　原生的 Recv-Driven 与补充的 Send-Driven 机制

从图中可以看到，相较于 Recv-Driven 模式，Send-Driven 模式的通信流程得到了比较大的简化，另外在数据 ready 后立即发送的特性，跳过了一侧的 Rendezvous，并且对于生产者先于消费者的情况，可以加快消费端数据获取的速度。

3.4 延迟优化

这部分优化，也是分布式计算的经典优化方向。整个流程链路上那些可以精简、合并、重叠需要不断去挖掘。对于机器学习系统来说，相比其它的系统，还可以用一些近似的算法来做这部分工作，从而获得较大的性能提升。下面介绍我们在两个这方面做的一些优化实践。

3.4.1 稀疏域参数聚合

在启用 HashTable 存储稀疏参数后，对应的，一些配套参数也需要替换为 HashTable 实现，这样整个计算图中会出现多张 HashTable 以及大量的相关算子。在实践中，我们发现需要尽量降低 Lookup/Insert 等算子的个数，一方面降低 PS 的负载，一方面降低 RPC QPS。因此，针对稀疏模型的常见用法，我们进行了相关的聚合工作。

以 Adam 优化器为例，需要创建 m、v 两个 slot，以保存优化中的动量信息，它的 Shape 与 Embedding 相同。在原生优化器中，这两个 Variable 是单独创建的，并在反向梯度更新的时候会去读写。同理，使用 HashTable 方案时，我们需要同时创建两张单独的 HashTable 用来训练 m、v 参数。那么在前向，反向中需要分别对 Embedding、m、v 进行一次 Lookup 和一次 Insert，总共需要三次 Lookup 和三次 Insert。

这里一个优化点就是将 Embedding、m、v，以及低频过滤的计数器（见下图 14 的 Counting HashTable）聚合到一起，作为 HashTable 的 Value，这样对稀疏参数的相关操作就可以聚合执行，大大减少了稀疏参数操作频次，降低了 PS 的压力。

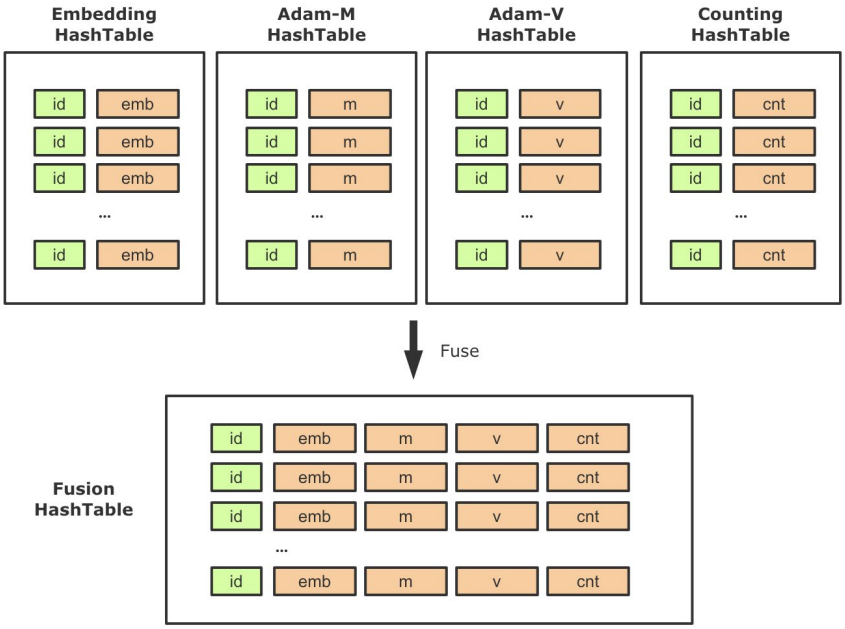


图 14　基于 HashTable 的参数融合策略

该特性属于一个普适型优化，开启聚合功能后，训练速度有了显著的提高，性能提升幅度随着模型和 Worker 规模的变化，效果总是正向的。在美团内部真实业务模型上，聚合之后性能相比非聚合方式能提升了 45% 左右。

3.4.2 Embedding 流水线优化

流水线，在工业生产中，指每一个生产单位只专注处理某个片段的工作，以提高工作效率及产量的一种生产方式。在计算机领域内，更为大家熟知的是，流水线代表一种多任务之间 Overlap 执行的并行化技术。例如在典型的 RISC 处理器中，用户的程序由大量指令构成，而一条指令的执行又可以大致分为：取指、译码、执行、访存、写回等环节。这些环节会利用到指令 Cache、数据 Cache、寄存器、ALU 等多种不同的硬件单元，在每一个指令周期内，这 5 个环节的硬件单元会并行执行，得以更加充分的利用硬件能力，以此提高整个处理器的指令吞吐性能。处理器的指令流水线是一套复杂而系统的底层技术，但其中的思想在分布式深度学习框架中也被大量的使用，例如：

- 如果将分布式训练简单的抽象为计算和通信两个过程，绝大多数主流的深度学习框架都支持在执行计算图 DAG 时，通信和计算的 Overlap。
- 如果将深度模型训练简单的分为前向和反向，在单步内，由于两者的强依赖性，无法做到有效并行，字节 BytePS[8] 中引入的通信调度打破了 step iteration 间的屏障，上一轮的部分参数更新完毕后，即可提前开始下轮的前向计算，增强了整体视角下前反向的 Overlap。
- 百度 AIBox[9] 为了解决 CTR 场景 GPU 训练时，参数位于主存，但计算位于 GPU 的问题，巧妙调度不同硬件设备，搭建起了主要利用 CPU/ 主存 / 网卡的参数预准备阶段和主要利用 GPU/NVLink 的网络计算阶段，通过两个阶段的 Overlap 达到更高的训练吞吐。

我们看到，在深度学习框架设计上，通过分析场景，可以从不同的视角发掘可并行的阶段，来提高整体的训练吞吐。

对于大规模稀疏模型训练时，核心模型流程是：先执行稀疏参数的 Embedding，然后执行稠密部分子网络。其中稀疏参数 Embedding 在远端 PS 上执行，主要耗费网络资源，而稠密部分子网络在本地 Worker 执行，主要耗费计算资源。这两部分占了整个流程的大部分时间，在美团某实际业务模型上分别耗时占比：40+%、50+%。

那我们是否可以提前执行稀疏参数的 Embedding，来做到通信和计算的 Overlap，隐藏掉这部分时间呢？从系统实现上肯定是可行的，但从算法上讲，这样做会引入参数 Staleness 的问题，可能会导致模型精度受到影响。但在实际的生产场景中，大规模异步训练时本身就会带来几十到几百个步的滞后性问题。经过我们测试，提前获取一两步的稀疏参数，模型精度并未受到影响。

在具体实现上，我们把整个计算图拆分为 Embedding Graph（EG）和 Main Graph（MG）两张子图，两者异步独立执行，做到拆分流程的 Overlap（整个拆分过程，可以做到对用户透明）。EG 主要覆盖从样本中抽取 Embedding Key，查询组装 Embedding 向量，Embedding 向量更新等环节。MG 主要包含稠密部分子网络计算、梯度计算、稠密参数部分更新等环节。

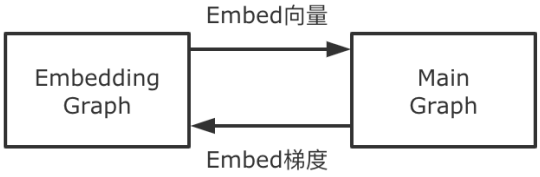


图 15　Embedding 流水线模块交互关系

两张子图的交互关系为：EG 向 MG 传递 Embeding 向量（从 MG 的视角看，是从一个稠密 Variable 读取数值）；MG 向 EG 传递 Embedding 参数对应的梯度。上述两个过程的表达都是 TensorFlow 的计算图，我们利用两个线程，两个 Session 并发的执行两张计算图，使得两个阶段 Overlap 起来，以此到达了更大的训练吞吐。

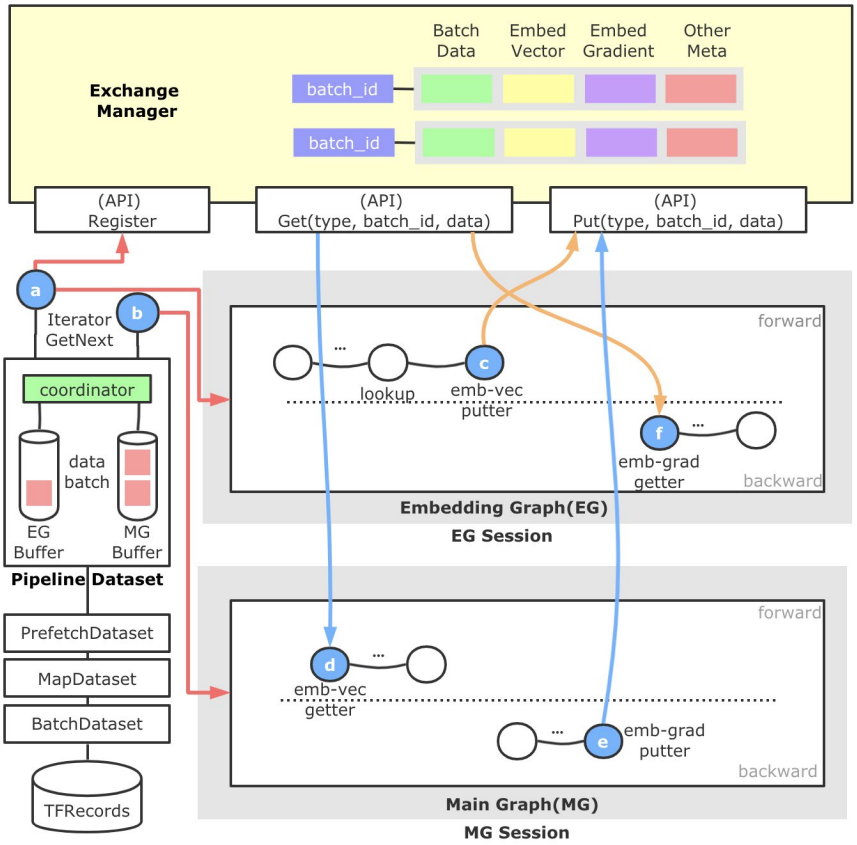


图 16　Embedding 流水线架构流程图

上图是 Embedding 流水线的架构流程图。直观来看分为左侧的样本分发模块，顶部的跨 Session 数据交换模块，以及自动图切分得到的 Embedding Graph 和 Main Graph，蓝色的圆圈代表新增算子，橙色箭头代表 EG 重点流程，蓝色箭头代表 MG 重点流程，红色箭头代表样本数据重点流程。

- 以对用户透明的形式引入了一层名为 Pipeline Dataset 的抽象层，这一层的产生是为了满足 EG/MG 两张计算图以不同节奏运行的需求，支持自定义配置。另外，为了使得整个流水线中的数据做到彼此的配套，这里还会负责进行一个全局 Batch ID 的生成及注册工作。Pipeline Dataset 对外暴露两种 Iterator，一个供 EG 使用，一个供 MG 使用。Pipeline Dataset 底部共享 TensorFlow 原生的各层 Dataset。
- 顶部的 ExchangeManager 是一个静态的，跨 Session 的数据交换媒介，对外暴露数据注册和数据拉取的能力。抽象这个模块的原因是，EG 和 MG 原本归属于一张计算图，因为流水线的原因拆解为拆为两张图，这样我们需要建立一种跨 Session 的数据交换机制，并准确进行配套。它内部以全局 Batch ID 做 Key，后面管理了样本数据、Embeding 向量、Embedding 梯度、Unique 后的 Index 等数据，并负责这些数据的生命周期管理。
- 中间的 Embedding Graph 由独立的 TF Session 运行于一个独立的线程中，通过 a 算子获得样本数据后，进行特征 ID 的抽取等动作，并进行基于 HashTable 方法的稀疏参数查询，查询结果通过 c 算子放置到 ExchangeManager 中。EG 中还包含用于反向更新的 f 算子，它会从 ExchangeManager 中获取 Embedding 梯度和与其配套的前向参数，然后执行梯度更新参数逻辑。
- 下面的 Main Graph 负责实际稠密子网络的计算，我们继承并实现一种可训练的 EmbeddingVariable，它的构建过程（d 算子）会从 ExchangeManager 查找与自己配套的 Embedding 向量封装成 EmbeddingVariable，给稠密子网络。此外，在 EmbeddingVariable 注册的反向方法中，我们添加了 e 算子使得 Embedding 梯度得以添加到 ExchangeManager 中，供 EG 中的 f 算子消费。

通过上面的设计，我们就搭建起了一套可控的 EG/MG 并发流水线训练模式。总体来看，Embedding 流水线训练模式的收益来源有：

- 经过我们对多个业务模型的 Profiling 分析发现，EG 和 MG 在时间的比例上在 3:7 或 4:6 的左右，通过将这两个阶段并行起来，可以有效的隐藏 Embedding 阶段，使得 MG 网络计算部分几乎总是可以立即开始，大大加速了整体模型的训练吞吐。
- TensorFlow 引擎中当使用多个优化器（稀疏与非稀疏）的时候，会出现重复构建反向计算图的问题，一定程度增加了额外计算，通过两张子图的拆分，恰好避免了这个问题。
- 在实施过程中的 ExchangeManager 不仅负责了 Embedding 参数和梯度的交换，还承担了元数据复用管理的职责。例如 Unique 等算子的结果保存，进一步降低了重复计算。

另外，在 API 设计上，我们做到了对用户透明，仅需一行代码即可开启 Embedding 流水线功能，对用户隐藏了 EG/MG 的切割过程。目前，在美团某业务训练中，Embedding 流水线功能在 CPU PS 架构下可以带来 20%~60% 的性能提升（而且 Worker 并发规模越大，性能越好）。

3.5 单实例 PS 并发优化

经过 2.2 章节的分析可知，我们不能通过持续扩 PS 来提升分布式任务的吞吐，单实例 PS 的并发优化，也是非常重要的优化方向。我们主要的优化工作如下。

3.5.1 高性能的 HashTable

PS 架构下，大规模稀疏模型训练对于 HashTable 的并发读写要求很高，因为每个 PS 都要承担成百乃至上千个 Worker 的 Embedding 压力，这里我们综合速度和稳定性考虑，选用了 tbb::concurrent_hash_map[10] 作为底层 HashTable 表实现，并将其包装成一个新的 TBBConcurrentHashTable 算子。经过测试，在千亿规模下 TBBConcurrentHashTable 比原生 MutableDenseHashTable 训练速度上快了 3 倍。

3.5.2 HashTable BucketPool

对于大规模稀疏模型训练来说，Embedding HashTable 会面对大量的并发操作，通过 Profiling 我们发现，频繁动态的内存申请会带来了较大性能开销（即使 TensorFlow 的 Tensor 有专门的内存分配器）。我们基于内存池化的思路优化了 HashTable 的内存管理。

我们在 HashTable 初始化时，会先为 Key 和 Value 分别创造两个 BucketPool，每个池子都会先 Malloc 较大一块内存备用，考虑到可能会有对 HashTable 进行中的 Key 和 Value 进行 Remove 的场景（如 Online Learning 训练时），需要对从 HashTable 中删除的 Key 和 Value 所使用的内存进行回收，因此每个 BucketPool 还有一个 ReuseQueue 来负责维护回收的内存。每次向内部的哈希表数据结构中 Insert Key 和 Value 的时候，Key 和 Value 内存和释放分配都进行池化管理。用这种方式降低了大规模稀疏训练中遇到稀疏内存分配开销，整体端到端训练性能提升了 5% 左右。

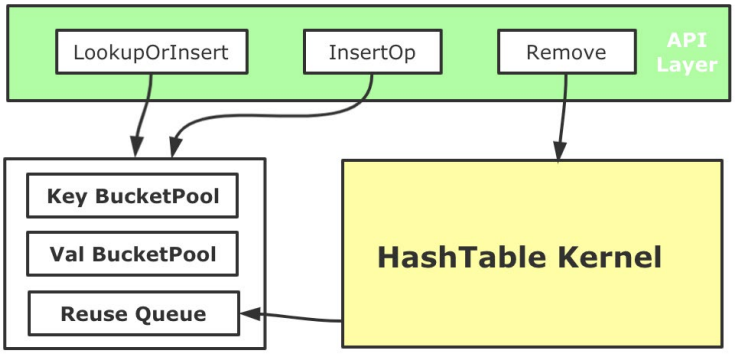


图 17 HashTable 内存优化

3.6 单位算力吞吐优化

经过 2.2 章节的分析，Worker 的计算压力也非常大，如果不优化 Worker，同时要保持吞吐，需要横向扩展更多的 Worker，给 PS 带来更大的压力。而对于用户来说，如果能在有限的计算资源下带来性能提升，对业务价值更高。我们通过 CAT 统计出了一些高频算子，并进行了专项优化。这里选取 Unique&DynamicPartition 算子融

合案例进行分享。

在 TensorFlow PS 架构中，包括 Embedding 向量在内的共享参数都存储在 PS 上，并通过网络与 Worker 交互，在进行 Embedding 查询过程中，往往会涉及如下两个环节：

- 由于稀疏参数的性质，从样本中抽取得到的待查询 Embedding ID，它的重复率往往高达 70%~90%，如果不进行去重查询，不论是对 HashTable 的查询还是网络的传输，都会带来不小的压力。因此，通常会在查询前进行 Unique 操作。
- 在大规模稀疏场景中，为了存储千亿规模的参数，会有多个 PS 机器共同承载。而 Worker 端会负责对查询请求按照设定的路由规则进行切分，这里通常会在查询前进行 DynamicPartition 动作。

通常这两个过程会利用 TensorFlow 既有的算子进行搭建，但在实际使用中，我们发现它并不是很高效，主要问题在于：

- Unique 算子原生实现，它内部使用的内存分配策略较为低效。使用了两倍输入参数（Embedding ID）的大小进行内存分配，但由于输入参数较大，而且重复率高，导致 HashTable 创建过大且非常稀疏。几乎每次插入都会产生一次 minor_page_fault，导致 HashTable 性能下降。我们使用 Intel Vtune 验证了这一点（参见图 18）。
- Unique 和 Dynamic Partition 算子存在冗余数据遍历，这些操作其实可以在一次数据遍历中全部做完，节省掉算子切换、冗余数据遍历的耗时。

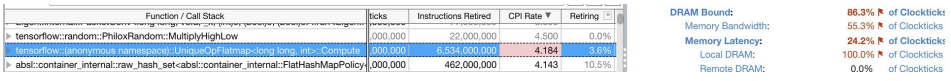


图 18　Unique 算子内部出现 DRAM Bound 问题

总结来说，HashTable 开辟过大会导致大量的 minor_page_fault，导致访存的时间增加，HashTable 过小又可能会导致扩容。我们采用了基于启发式算法的内存自适应 Unique 算子实现，通过对训练历史重复率的统计，我们可以得到一个相对合理的

HashTable 大小，来提高访存的性能；另外 Unique 算子内 HashTable 的具体选择上，经过我们的多种测试，选择了 Robin HashTable 替换了原生 TF 中的实现。

进一步，我们对围绕 Embedding ID 的 Unique 和 Partition 环节进行了算子合并，简化了逻辑实现。经过上述的优化，Unique 单算子可以取得 51% 的加速，在真实模型端到端上可以获得 10% 左右的性能提升，算子总数量降低了 4%。

在整个关键算子优化的过程中，Intel 公司的林立凡、张向泽、高明进行大量的技术支持，我们也复用了他们的部分优化工作，在此深表感谢！

4. 大规模稀疏算法建模

大规模稀疏能力在业务落地的过程中，算法层面还需要从特征和模型结构上进行对应升级，才能拿到非常好的效果。其中外卖广告从业务特点出发，引入大规模稀疏特征完成外卖场景下特征体系的升级，提供了更高维的特征空间和参数空间，增强了模型的拟合能力。重新设计了面向高维稀疏场景的特征编码方案，解决了特征编码过程中的特征冲突问题，同时编码过程去掉了部分冗余的特征哈希操作，一定程度上简化了特征处理逻辑，并降低了特征计算的耗时。

在系统层面，面对百亿参数、百亿样本以上量级的大规模稀疏模型的训练，会带来训练迭代效率的大大降低，单次实验从一天以内，增长到一周左右。美团机器学习平台训练引擎团队，除了上述 TensorFlow 框架层面的优化、还针对业务模型进行了专项优化，整体吞吐优化了 8 到 10 倍（如果投入更多计算资源，可以进一步加速），大大提升业务的迭代效率，助力外卖广告业务取得了较为明显的提升。

5. 总结与展望

TensorFlow 在大规模推荐系统中被广泛使用，但由于缺乏大规模稀疏的大规模分布式训练能力，阻碍了业务的发展。美团基于 TensorFlow 原生架构，支持了大规模稀疏能力，并从多个角度进行了深度优化，做到千亿参数、千亿样本高效的分布式训练，并在美团内部进行了大规模的使用。对于这类关键能力的缺失，TensorFlow 社

区也引起了共鸣，社区官方在 2020 年创建了 SIG Recommenders[11]，通过社区共建的方式来解决此类问题，美团后续也会积极的参与到社区的贡献当中去。

美团推荐系统场景的模型训练，目前主要运行在 CPU 上，但随着业务的发展，有些模型变得越来越复杂，CPU 上已经很难有优化空间（优化后的 Worker CPU 使用率在 90% 以上）。而近几年，GPU 的计算能力突飞猛进，新一代的 NVIDIA A100 GPU，算力达到了 156TFLOPS（TF32 Tensor Cores）、80G 显存、卡间带宽 600GB/s。对于这类复杂模型的 Workload，我们基于 A100 GPU 架构，设计了下一代的分布式训练架构，经过初步优化，在美团某大流量业务推荐模型上也拿到了较好的效果，目前还在进一步优化当中，后续我们会进行分享，敬请期待。

6. 作者简介

逸帆、家恒、峥少、鹏鹏、永宇、正阳、黄军等，来自美团基础研发平台，机器学习平台训练引擎组，主要负责美团分布式机器学习训练系统的性能优化与能力建设。
海涛，来自美团外卖广告策略团队，主要负责美团外卖广告业务的算法探索和策略落地工作。

7. 招聘信息

美团机器学习平台大量岗位持续招聘中，社招 / 校招均可（欢迎投递我们的校招北斗岗位：美团机器学习平台基础架构），坐标北京 / 上海，构建多领域的公司级机器学习平台，帮大家吃得更好，生活更好。简历可投递至：huangjun03@meituan.com。

8. 参考信息

[1] https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
[2] https://github.com/dianping/cat
[3] https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf
[4] https://github.com/tensorflow/networking/tree/master/tensorflow_networking/verbs
[5] https://labs.criteo.com/2013/12/download-terabyte-click-logs/
[6] https://arxiv.org/abs/1906.00091
[7] https://github.com/tensorflow/networking/tree/master/tensorflow_networking/seastar
[8] https://github.com/bytedance/byteps
[9] http://research.baidu.com/Public/uploads/5e18a1017a7a0.pdf
[10] https://github.com/oneapi-src/oneTBB
[11] https://github.com/tensorflow/recommenders-addons

美团 Serverless 平台 Nest 的探索与实践

作者：殷琦　华珅　飞飞　志洋　奕锟　佳文　凯鑫　亚辉

1. 背景

Serverless 一词于 2012 年被提出，2014 年由于亚马逊的 AWS Lambda 无服务器计算服务的兴起，而被大家广泛认知。Serverless 通常被直译成“无服务器”，无服务器计算是可以让用户在不考虑服务器的情况下构建并运行应用程序。使用无服务器计算，应用程序仍在服务器上运行，但所有服务器管理工作均由 Serverless 平台负责。如机器申请、代码发布、机器宕机、实例扩缩容、机房容灾等都由平台帮助自动完成，业务开发只需考虑业务逻辑的实现即可。

回顾计算行业的发展历程，基础设施从物理机到虚拟机，再从虚拟机到容器；服务架构从传统单体应用架构到 SOA 架构，再从 SOA 架构到微服务架构。从基础设施和服务架构两条主线来看整体技术发展趋势，大家可能会发现，不论是基础设施还是服务架构，都是从大往小或者由巨到微的方向上演进，这种演变的本质原则无非是解决资源成本或者研发效率的问题。当然，Serverless 也不例外，它也是用来解决这两个方面的问题：

- **资源利用率**：Serverless 产品支持快速弹性伸缩能力，能够帮助业务提升资源利用率，在业务流量高峰时，业务的计算能力、容量自动扩容，承载更多的用户请求，而在业务流量下降时，所使用的资源也会同时收缩，避免资源浪费。
- **研发运维效率**：在 Serverless 上开发人员一般只需要填写代码路径或者上传代码包，平台能够帮助完成构建、部署的工作。开发人员不直接面对机器，对于机器的管理，机器是否正常以及流量高低峰的是否需要扩缩容等问题，这些统统不需要去考虑，由 Serverless 产品帮助研发人员去完成。这样就能使他们从繁琐的运维工作中解放出来，从 DevOps 转向 NoOps，更加专注于业务逻辑的实现。

虽然 AWS 在 2014 年就推出了第一个 Serverless 产品 Lambda，但 Serverless 技术在国内的应用一直不温不火。不过近两三年，在容器、Kubernetes 以及云原生等技术的推动下，Serverless 技术迅速发展，国内各大互联网公司都在积极建设 Serverless 相关产品，探索 Serverless 技术的落地。在这种背景下，美团也于 2019 年初开始了 Serverless 平台的建设，**内部项目名称为 Nest**。

截止到目前，Nest 平台已经过两年的建设，回顾整体的建设过程，主要经历了以下三个阶段：

- **快速验证，落地 MVP 版本**：我们通过技术选型、产品与架构设计、开发迭代，快速落地了 Serverless 产品的基本的能力，如构建、发布、弹性伸缩、对接触发源、执行函数等。上线后，我们推进了一些业务的试点接入，帮助验证打磨产品。
- **优化核心技术，保障业务稳定性**：有了前期的试点业务验证，我们很快发现产品的存在的一些稳定性相关的问题，主要有弹性伸缩的稳定性、冷启动的速度、系统与业务的可用性、容器的稳定性。针对这些问题我们对各个问题涉及的技术点做了专项的优化改进。
- **完善技术生态，落实收益**：优化了核心技术点后，产品逐渐成熟稳定，但依然面临生态性问题，如研发工具欠缺，上下游产品没有打通、平台开放能力不足等问题，影响或阻碍了产品的推广使用。因此，我们继续完善产品的技术生态，扫清业务接入使用障碍，落实产品的业务收益。

2. 快速验证，落地 MVP 版本

2.1 技术选型

建设 Nest 平台，首要解决的就是技术选型问题，Nest 主要涉及三个关键点的选型：演进路线、基础设施、开发语言。

2.1.1 演进路线

起初 Serverless 服务主要包含 FaaS（Function as a Service）和 BaaS（Back-end as a Service），近几年 Serverless 的产品领域有所扩张，它还包含面向应用的 Serverless 服务。

- **FaaS**：是运行在一个无状态的计算容器中的函数服务，函数通常是事件驱动、生命周期很短（甚至只有一次调用）、完全由第三方管理的。业界相关 FaaS 产品有 AWS 的 Lambda、阿里云的函数计算等。
- **BaaS**：是建立在云服务生态之上的后端服务。业界相关 BaaS 产品包括 AWS 的 S3、DynamoDB 等。

面向应用的 Serverless 服务：如 Knative，它提供了从代码包到镜像的构建、部署，以及弹性伸缩等全面的服务托管能力，公有云产品有 Google Cloud Run（基于 Knative）、阿里云的 SAE（Serverless Application Engine）。

在美团内部，BaaS 产品其实就是内部的中间件以及底层服务等，它们经过多年的发展，已经非常丰富且成熟了。因此，在美团的 Serverless 产品演进主要在函数计算服务和面向应用的 Serverless 服务两个方向上。那究竟该如何演进呢？当时主要考虑到在业界 FaaS 函数计算服务相对于面向应用的 Serverless 服务来说，更加成熟且确定。因此，我们决定“**先建设 FaaS 函数计算服务，再建设面向应用的 Serverless 服务**”这样一条演进路线。

2.1.2 基础设施

由于弹性伸缩是 Serverless 平台必备的能力，因此 Serverless 必然涉及到底层资源的调度和管理。这也是为什么当前业界有很多开源的 Serverless 产品（如 OpenFaaS、Fission、Nuclio、Knative 等）是基于 Kubernetes 来实现的，因为这种选型能够充分利用 Kubernetes 的基础设施的管理能力。在美团内部基础设施产品是 Hulk，虽然 Hulk 是基于 Kubernetes 封装后的产品，但 Hulk 在落地之初考虑到落地难度以及各种原因，最终未按照原生的方式来使用 Kubernetes，并且在容器层采用的也是富容器模式。

在这种历史背景下，我们在做基础设施选型时就面临两种选项：一是使用公司的 Hulk 来作为 Nest 的基础设施（非原生 Kubernetes），二是采用原生 Kubernetes 基础设施。我们考虑到当前业界使用原生 Kubernetes 是主流趋势并且使用原生 Kubernetes 还能充分利用 Kubernetes 原生能力，可以减少重复开发。因此，最终考量的结果是我们采用了**原生 Kubernetes 作为我们的基础设施**。

2.1.3 开发语言

由于在云原生领域的主流语言是 Golang，并且 Kubernetes 的生态中，Golang 是绝对的主导语言。但在美团，Java 才是使用最广泛的语言，相比 Golang，Java 在公司内部生态比较好。因此，在语言的选型上我们**选择了 Java 语言**。在 Nest 产品开发之初，Kubernetes 社区的 Java 客户端还不够完善，但随着项目的推进，社区的 Java 客户端也逐渐丰富了起来，目前已经完全够用了。另外，我们也在使用过程中，也贡献了一些 Pull Request，反哺了社区。

2.2 架构设计

基于以上的演进路线、基础设施、开发语言的选型，我们进行了 Nest 产品的架构设计。

在整体的架构上，流量由 EventTrigger（事件触发源，如 Nginx、应用网关、定时任务、消息队列、RPC 调用等）触发到 Nest 平台，Nest 平台内会根据流量的特征路由到具体函数实例，触发函数执行，而函数内部代码逻辑可以调用公司内的各个 BaaS 服务，最终完成函数的执行，返回结果。

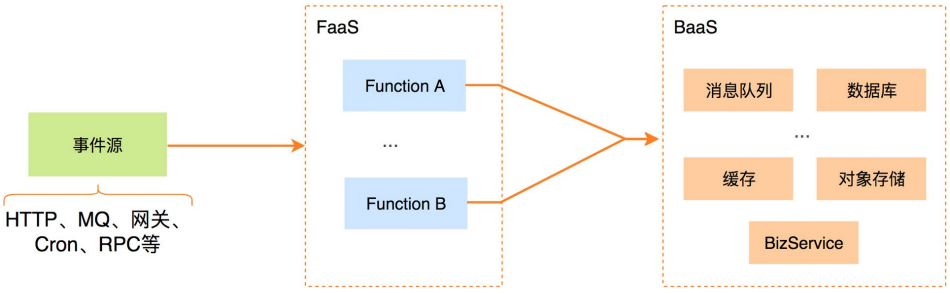


图 1　FaaS 架构图

在技术实现上，Nest 平台使用 Kubernetes 作为基础底座并适当参考了一些 Knative 的优秀设计，在其架构内部主要由以下几个核心部分组成：

- **事件网关**：核心能力是负责对接外部事件源的流量，然后路由到函数实例上；另外，网关还负责统计各个函数的进出流量信息，为弹性伸缩模块提供伸缩决策的数据支撑。
- **弹性伸缩**：核心能力是负责函数实例的弹性伸缩，伸缩主要根据函数运行的流量数据以及实例阈值配置计算函数目标实例个数，然后借助 Kubernetes 的资源控制能力，调整函数实例的个数。
- **控制器**：核心能力是负责 Kubernetes CRD（Custom Resource Definition）的控制逻辑实现。
- **函数实例**：函数的运行实例。当事件网关流量触发过来，会在函数实例内执行相应的函数代码逻辑。
- **治理平台**：面向用户使用的平台，负责函数的构建、版本、发布以及一些函数元信息的管理等。

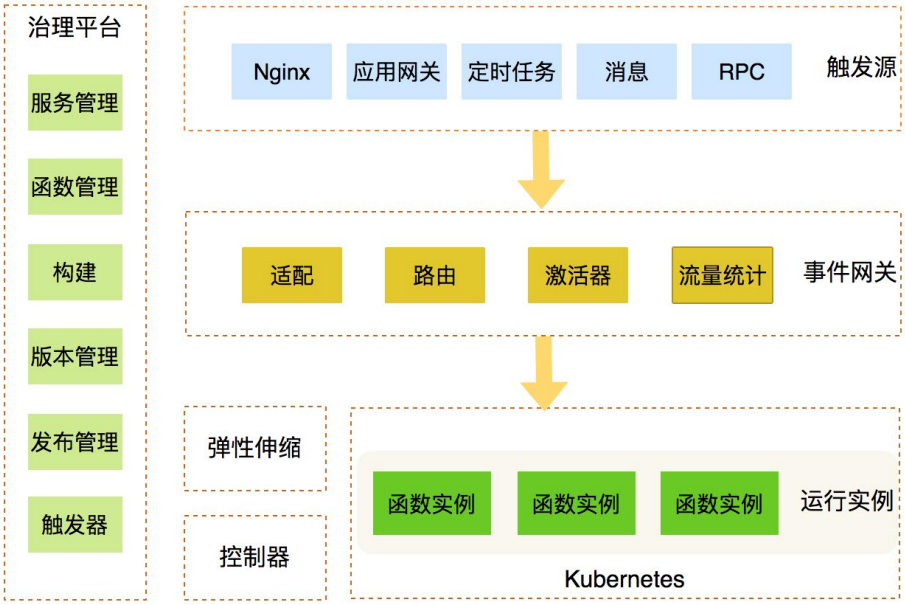


图 2　Nest 架构图

2.3 流程设计

在具体的 CI/CD 流程上，Nest 又与传统的模式有何区别呢？为了说明这个问题，我们先来看一看在 Nest 平台上函数的整体生命周期怎样的？具体有以下四个阶段：构建、版本、部署、伸缩。

- **构建**：开发的代码和配置通过构建生成镜像或可执行文件。
- **版本**：构建生成的镜像或可执行文件加上发布配置形成一个不可变的版本。
- **部署**：将版本发布，即完成部署。
- **伸缩**：根据函数实例的流量以及负载等信息，来进行实例的弹性扩缩容。

就这四个阶段来看，Nest 与传统的 CI/CD 流程本质区别在于部署和伸缩：传统的部署是感知机器的，一般是将代码包发布到确定的机器上，但 Serverless 是要向用户屏蔽机器的（在部署时，可能函数的实例数还是 0）；另外，传统的模式一般是不具备动态扩缩容的，而 Serverless 则不同，Serverless 平台会根据业务的自身流量需要，进行动态扩缩容。后续章节会详细讲解弹性伸缩，因此这里我们只探讨部署的设计。

部署的核心点在于如何向用户屏蔽机器？对于这个问题，我们**抽象了机器，提出了分组**的概念，分组是由 SET（单元化架构的标识，机器上会带有该标识）、泳道（测试环境隔离标识，机器上会带有该标识）、区域（上海、北京等）三个信息组成。用户部署只需在相应的分组上进行操作，而不用涉及到具体机器。能够做到这些的背后，是由 Nest 平台帮助用户管理了机器资源，每次部署会根据分组信息来实时初始化相应的机器实例。

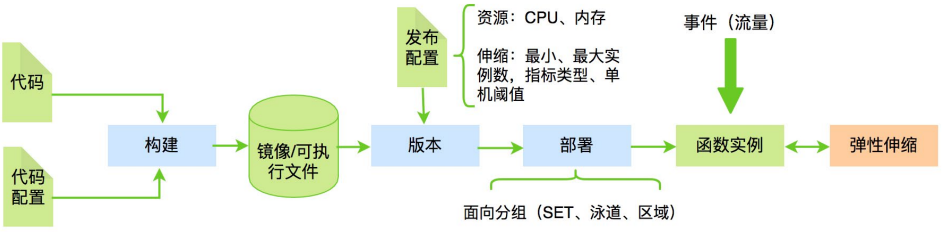


图 3　函数生命周期

2.4 函数触发

函数的执行是由事件触发的。完成函数的触发，需要实现以下四个流程：

- **流量引入**：向事件源注册事件网关的信息，将流量引入到事件网关。如针对 MQ 事件源，通过注册 MQ 的消费组，引入 MQ 的流量到事件网关。
- **流量适配**：事件网关对事件源进入的流量进行适配对接。
- **函数发现**：对函数元数据（函数实例信息、配置信息等）的获取过程，类似微服务的服务发现过程。事件网关接受的事件流量需要发送到具体的函数实例，这就需要对函数进行发现。这里发现实质是获取 Kubernetes 中的内置资源或者 CRD 资源中存储的信息。
- **函数路由**：事件流量的路由过程，路由到特定的函数实例上。这里为了支持传统路由逻辑（如 SET、泳道、区域路由等）以及版本路由能力，我们采用了多层路由，第一层路由到分组（SET、泳道、区域路由），第二层路由到具体版本。同版本内的实例，通过负载均衡器选择出具体实例。另外，通过该版本路由，我们很轻松的支持了金丝雀、蓝绿发布。

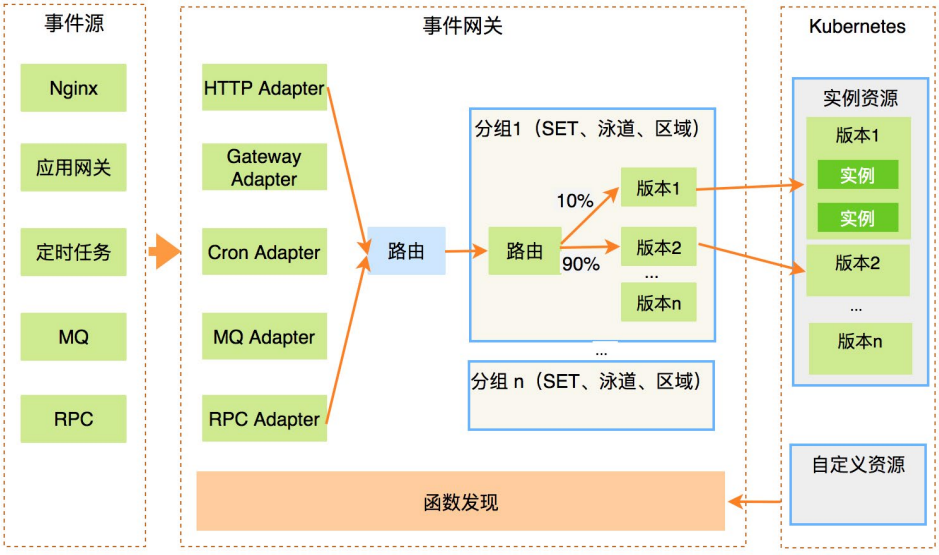


图 4　函数触发

2.5 函数执行

函数不同于传统的服务，传统的服务是个可执行的程序，但函数不同，函数是代码片段，自身是不能单独执行的。那流量触发到函数实例后，函数是如何执行的呢?

函数的执行的首要问题是函数的运行环境：由于 Nest 平台是基于 Kubernetes 实现的，因此函数一定是运行在 Kubernetes 的 Pod（实例）内，Pod 内部是容器，容器的内部是运行时，运行时是函数流量接收的入口，最终也是由运行时来触发函数的执行。一切看起来是那么的顺利成章，但我们在落地时是还是遇到了一些困难，最主要的困难是**让开发同学可以在函数内无缝的使用公司内的组件**，如 OCTO（服务框架）、Celler（缓存系统）、DB 等。

在美团的技术体系中，由于多年的技术沉淀，很难在一个纯粹的容器（没有任何其他依赖）中运行公司的业务逻辑。因为公司的容器中沉淀了很多环境或服务治理等能力，如服务治理的 Agent 服务以及实例环境配置、网络配置等。

因此，为了业务在函数内无缝的使用公司内的组件，我们复用公司的容器体系来降低业务编写函数的成本。但复用公司的容器体系也没那么简单，因为在公司内没有人试过这条路，Nest 是公司第一个基于原生 Kubernetes 建设的平台，“第一个吃螃蟹的人”总会遇到一些坑。对于这些坑，我们只能在推进过程中“逢山开路，遇水搭桥”，遇到一个解决一个。总结下来，其中最核心的是在容器的启动环节打通的 CMDB 等技术体系，让运行函数的容器与开发同学平时申请的机器用起来没有任何区别。

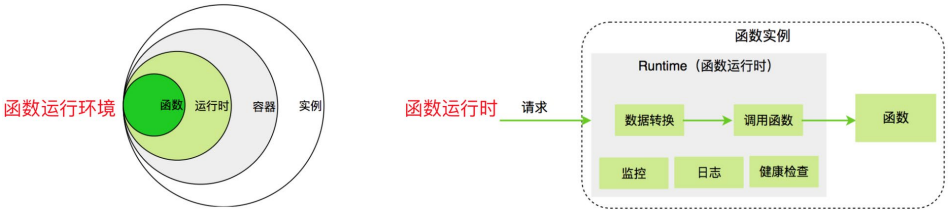


图 5　函数执行

2.6 弹性伸缩

弹性伸缩的核心问题主要有三个：什么时候伸缩，伸缩多少，伸缩的速度快不快？也就是伸缩时机、伸缩算法、伸缩速度的问题。

- **伸缩时机**：根据流量 Metrics 实时计算函数期望实例数，进行扩缩。流量的 Metrics 数据来自于事件网关，这里主要统计函数的并发度指标，弹性伸缩组件每秒中会主动从事件网关获取一次 Metrics 数据。
- **伸缩算法**：并发度 / 单实例阈值 = 期望实例数。根据收集的 Metrics 数据以及业务配置的阈值，通过算法计算出期望的实例数，然后通过 Kubernetes 接口设置具体实例数。整个算法看起来虽然简单，但非常稳定、鲁棒性好。
- **伸缩速度**：主要取决于冷启动时间，在下个章节会详细讲解这块内容。

除了基本的扩缩容能力，我们还支持了伸缩到 0，支持配置最大、最小实例数（最小实例即预留实例）。伸缩到 0 的具体实现是，我们在事件网关内部增加了激活器模块，当函数无实例时，会将函数的请求流量缓存在激活器内部，然后立即通过流量的 Metrics 去驱动弹性伸缩组件进行扩容，等扩容的实例启动完成后，激活器再将缓存的请求重试到扩容的实例上触发函数执行。

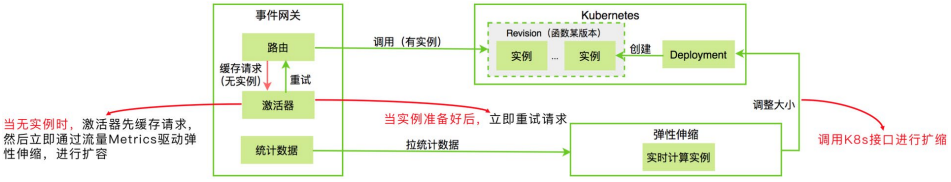


图 6　弹性伸缩

3. 优化核心技术，保障业务稳定性

3.1 弹性伸缩优化

上面提到的伸缩时机、伸缩算法、伸缩速度这三要素都是理想情况下的模型，尤其是伸缩速度，当前技术根本做不到毫秒级别的扩缩容。因此，在线上实际场景中，弹性

伸缩会存在一些不符合预期的情况，比如实例伸缩比较频繁或者扩容来不及，导致服务不太稳定的问题。

- **针对实例伸缩比较频繁问题**，我们在弹性伸缩组件内维护了统计数据的滑动窗口，通过计算均值来平滑指标，还通过延时缩容，实时扩容来缓解频繁扩缩问题。另外，我们增加了基于 QPS 指标的伸缩策略，因为 QPS 指标相对并发度指标会更加稳定。
- **针对扩容来不及问题**，我们采取提前扩容的手段，当达到实例阈值的 70% 就扩容，能够比较好的缓解这个问题。除此之外，我们还支持了多指标混合伸缩（并发度、QPS、CPU、Memory），定时伸缩等策略，满足各种业务需求。

下图展示的是线上弹性伸缩的真实案例（其配置的最小实例数为 4，单实例阈值 100，阈值使用率 0.7），其中上半部分是业务每秒的请求数，下半部分是扩缩实例的决策图，可以看到在成功率 100% 的情况下，业务完美应对流量高峰。

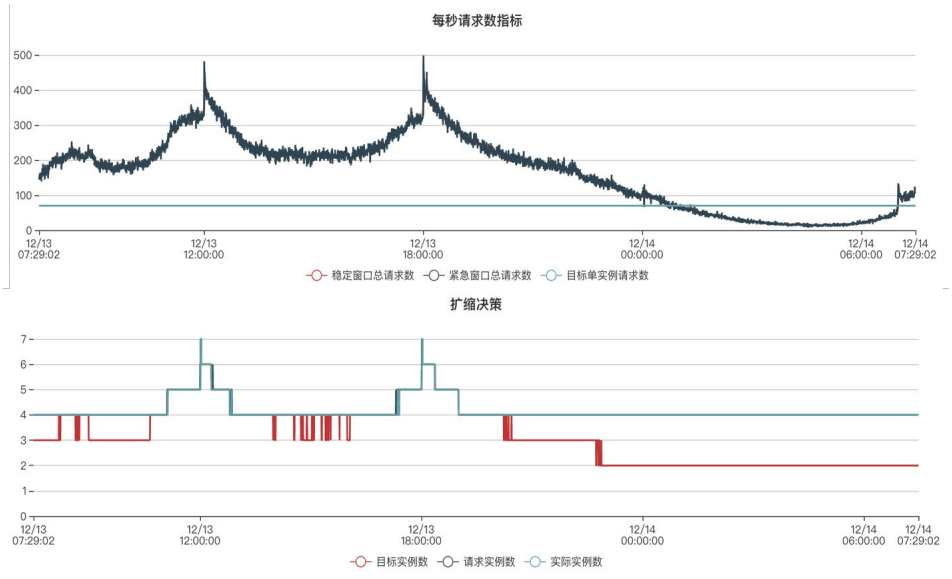


图 7　弹性伸缩案例

3.2 冷启动优化

冷启动是指在函数调用链路中包含了资源调度、镜像 / 代码下载、启动容器、运行时初始化、用户代码初始化等环节。当冷启动完成后，函数实例就绪，后续请求就能直接被函数执行。冷启动在 Serverless 领域至关重要，它的耗时决定了弹性伸缩的速度。

所谓“天下武功，无坚不破，唯快不破”，这句话在 Serverless 领域也同样受用。试想如果拉起一个实例足够快，快到毫秒级别，那几乎所有的函数实例都可以缩容到 0，等有流量时，再扩容实例处理请求，这对于存在高低峰流量的业务将极大的节省机器资源成本。当然，理想很丰满，现实很骨感。做到毫秒级别几乎不可能。但只要冷启动时间越来越短，成本自然就会越来越低，另外，极短的冷启动时间对伸缩时函数的可用性以及稳定性都有莫大的好处。



图 8　冷启动的各个阶段

冷启动优化是个循序渐进的过程，我们对冷启动优化主要经历了三个阶段：镜像启动优化、资源池优化、核心路径优化。

- **镜像启动优化**：我们对镜像启动过程中的耗时环节（启动容器和运行时初始化）进行了针对性优化，主要对容器 IO 限速、一些特殊 Agent 启动耗时、启动盘与数据盘数据拷贝等关键点的优化，最终将启动过程中的系统耗时从 42s 优化到 12s 左右。

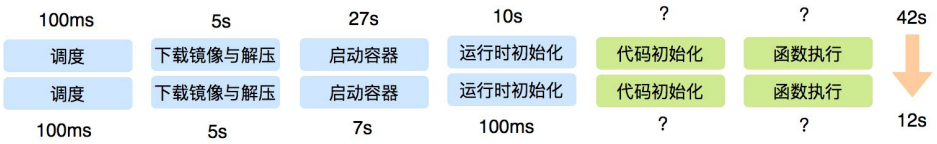


图 9　镜像启动优化成果

- **资源池优化**：镜像启动耗时优化到 12s，基本已经快达到瓶颈点，再继续优化空间不大。因此，我们想能否绕开镜像启动的耗时环节？最终，我们采用了一个比较简单思路“空间换时间”，用资源池方案：缓存一些已启动的实例，当需要扩容时，直接从资源池获取实例，绕开镜像启动容器的环节，最终效果很明显，将启动的系统耗时从 12s 优化到 3s。这里需要说明的是资源池自身也是通过 Kubernetes 的 Depolyment 进行管理，池中实例被取走会立即自动补充。

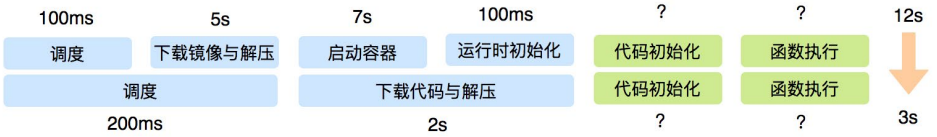


图 10　资源池优化成果

- **核心路径优化**：在资源池优化的基础上，我们再次精益求精，针对启动流程中的下载与解压代码两个耗时环节进行优化，过程中我们采用了高性能的压缩解压算法（LZ4 与 Zstd）以及并行下载和解压技术，效果非常好。另外，我们还支持了通用逻辑（中间件、依赖包等）下沉，通过预加载的方式，最终将函数端到端的启动耗时优化到 2s，这就意味着扩容一个函数实例只需要 2s（包含函数启动）。如果排除掉函数自身的初始化启动耗时，平台侧的耗时已在毫秒级别。

3.3 高可用保障

说到高可用，对于一般的平台，指的就是平台自身的高可用，但 Nest 平台有所不同，Nest 的高可用还包含托管在 Nest 平台上的函数。因此，Nest 的高可用保障需要从平台和业务函数两个方面着手。

3.3.1 平台高可用

对平台的高可用，Nest 主要从架构层、服务层、监控运营层、业务视角层面都做了全面的保障。

- **架构层**：我们针对有状态服务，如弹性伸缩模块，采用了主从架构，当主节点异常时从节点会立即替换。另外，我们还实现了架构上的多层隔离。 横向地域隔离：Kubernetes 两地两集群强隔离、服务（事件网关、弹性伸缩）集群内两地弱隔离（上海的弹性伸缩只负责上海 Kubernetes 集群内的业务伸缩，事件网关存在两地调用需求，需访问两地 Kubernetes）。纵向业务线隔离：服务业务线强隔离，不同业务线使用不同集群服务；在 Kubernetes 层的资源用 namespace 实现业务线弱隔离。

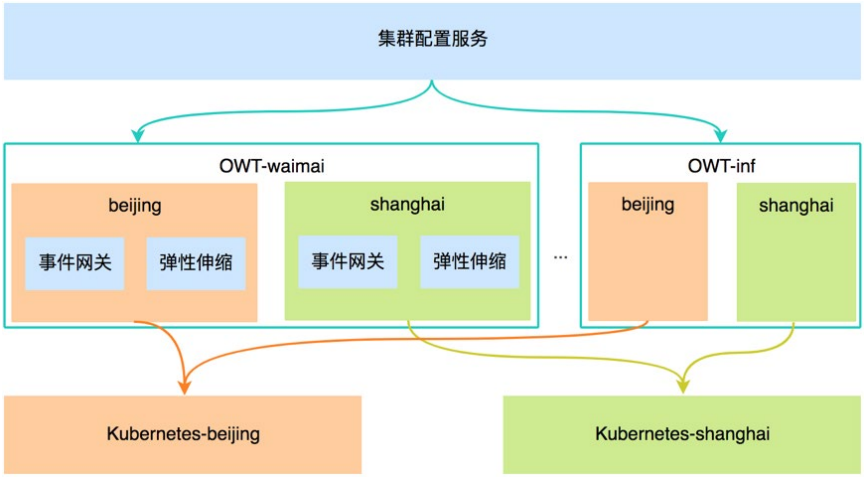


图 11　部署架构

- **服务层**：主要指的是事件网关服务，由于所有的函数流量都经过事件网关，因此事件网关的可用性尤为重要，这层我们支持了限流和异步化，保障服务的稳定性。
- **监控运营层**：主要通过完善系统监控告警、梳理核心链路并推动相关依赖方进行治理。另外，我们会定期梳理 SOP 并通过故障演练平台实施故障注入演练，发现系统隐患问题。

- **业务视角层**：我们开发了在线不间断实时巡检服务，通过模拟用户函数的请求流量，实时检测系统的核心链路是否正常。

3.3.2 业务高可用

对于业务高可用，Nest 主要从服务层、平台层两个层面做了相关的保障。

- **服务层**：支持了业务降级、限流能力：当后端函数故障时，可通过降级配置，返回降级结果。针对异常的函数流量，平台支持限制其流量，防止后端函数实例的被异常流量打垮。
- **平台层**：支持了实例保活、多层级容灾以及丰富的监控告警能力：当函数实例异常时，平台会自动隔离该实例并立即扩容新实例。平台支持业务多地区部署，在同地区将函数实例尽可能打散不同机房。当宿主机、机房、地区故障时，会立即在可用宿主机、可用机房或可用区重建新实例。另外，平台自动帮业务提供了函数在时延、成功率、实例伸缩、请求数等多种指标的监控，当在这些指标不符合预期时，自动触发告警，通知业务开发和管理员。



图 12　业务监控

3.4 容器稳定性优化

前文已提到，Serverless 与传统模式在 CI/CD 流程上是不同的，传统模式都是事先准备好机器然后部署程序，而 Serverless 则是根据流量的高低峰实时弹性扩缩容实例。当新实例扩容出来后，会立即处理业务流量。这听起来貌似没什么毛病，但在富容器生态下是存在一些问题的：我们发现刚扩容的机器负载非常高，导致一些业务请求执行失败，影响业务可用性。

分析后发现主要是因为容器启动后，运维工具会进行 Agent 升级、配置修改等操作，这些操作非常耗 CPU。同在一个富容器中，自然就抢占了函数进程的资源，导致用户进程不稳定。另外，函数实例的资源配置一般比传统服务的机器要小很多，这也加剧了该问题的严重性。基于此，我们参考业界，联合容器设施团队，落地了轻量级容器，将运维的所有 Agent 放到 Sidecar 容器中，而业务的进程单独放到 App 容器中。采用这种容器的隔离机制，保障业务的稳定性。同时，我们也推动了容器裁剪计划，去掉一些不必要的 Agent。

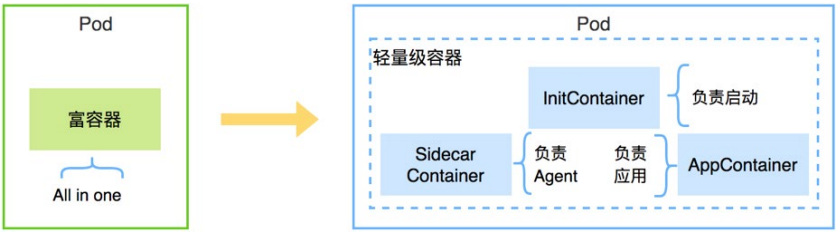


图 13 轻量级容器

4. 完善生态，落实收益

Serverless 是个系统工程，在技术上涉及到 Kubernetes、容器、操作系统、JVM、运行时等各种技术，在平台能力上涉及到 CI/CD 各个流程的方方面面。

为了给用户提供极致的开发体验，我们为用户提供了开发工具的支持，如 CLI（Command Line Interface）、WebIDE 等。为了解决现有上下游技术产品的交互的问题，我们与公司现有的技术生态做了融合打通，方便开发同学使用。为了方便下游

的集成平台对接，我们开放了平台的 API，实现 Nest 赋能各下游平台。针对容器过重，系统开销大，导致低频业务函数自身资源利用率不高的问题，我们支持了函数合并部署，成倍提升资源利用率。

4.1 提供研发工具

开发工具能够降低平台的使用成本，帮助开发同学快速的进行 CI/CD 流程。目前 Nest 提供了 CLI 工具，帮助开发同学快速完成创建应用、本地构建、本地测试、Debug、远程发布等操作。Nest 还提供了 WebIDE，支持在线一站式完成代码的修改、构建、发布、测试。

4.2 融合技术生态

仅支持这些研发工具还是不够的，项目推广使用后，我们很快就发现开发同学对平台有了新的需求，如无法在 Pipeline 流水线、线下服务实例编排平台上完成对函数的操作，这对我们项目的推广也形成了一些阻碍。因此，我们融合这些公司的成熟技术生态，打通了 Pipeline 流水线等平台，融入到现有的上下游技术体系内，解决用户的后顾之忧。

4.3 开放平台能力

有很多 Nest 的下游解决方案平台，如 SSR（Server Side Render）、服务编排平台等，通过对接 Nest 的 OpenAPI，实现了生产力的进一步解放。例如，不用让开发同学自己去申请、管理和运维机器资源，就能够让用户非常快速的实现一个 SSR 项目或者编排程序从 0 到 1 的创建、发布与托管。

Nest 除了开放了平台的 API，还对用户提供了自定义资源池的能力，拥有了该项能力，开发同学可以定制自己的资源池，定制自己的机器环境，甚至可以下沉一些通用的逻辑，实现冷启动的进一步优化。

4.4 支持合并部署

合并部署指的是将多个函数部署在一个机器实例内。合并部署的背景主要有两个：

- 当前的容器较重，容器自身的系统开销较大，导致业务进程资源利用率不高（尤其是低频业务）。
- 在冷启动耗时不能满足业务对时延的要求的情况下，我们通过预留实例来解决业务的需求。

基于这两个背景，我们考虑支持合并部署，将一些低频的函数部署到同一个机器实例内，来提升预留实例中业务进程的资源利用率。

在具体实现上，我们参考 Kubernetes 的设计方案，设计了一套基于 Sandbox 的函数合并部署体系（每个 Sandbox 就是一个函数资源），将 Pod 类比成 Kubernetes 的 Node 资源，Sandbox 类比成 Kubernetes 的 Pod 资源，Nest Sidecar 类比成 Kubelet。为了实现 Sandbox 特有的部署、调度等能力，我们还自定义了一些 Kubernetes 资源（如 SandboxDeployment、SandboxReplicaSet、Sandbox-Endpoints 等）来支持函数动态插拔到具体的 Pod 实例上。

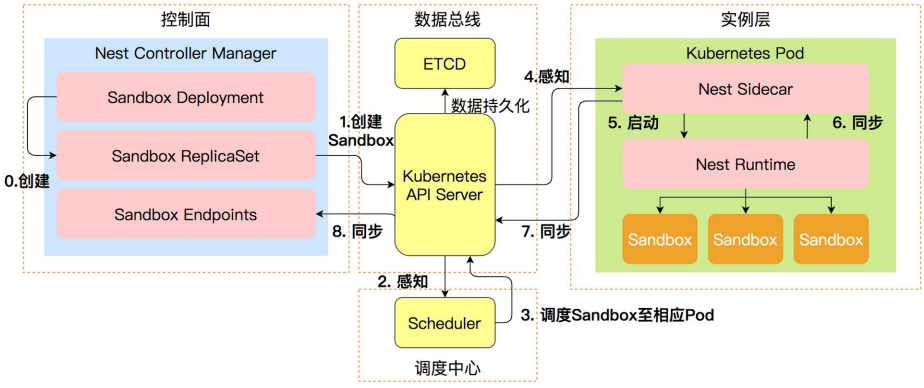


图 14　合并部署架构

除此之外，在合并部署的形态下，**函数之间的隔离性**也是不可回避的问题。为了尽可能的解决函数（合并在同一个实例中）之间的互相干扰问题，在 Runtime 的实现上，我们针对 Node.js 和 Java 语言的特点采取了不同的策略：Node.js 语言的函数使用不同的进程来实现隔离，而 Java 语言的函数，我们采用类加载隔离。采用这种策略的主要原因是由于 Java 进程占用内存空间相较于 Node.js 进程会大很多。

5. 落地场景、收益

目前 Nest 产品在美团前端 Node.js 领域非常受欢迎，也是落地最广泛的技术栈。当前 Nest 产品在美团前端已实现了规模化落地，几乎涵盖了所有业务线，接入了大量的 B/C 端的核心流量。

5.1 落地场景

具体的落地前端场景有：BFF（Backend For Frontend）、CSR（Client Side Render）/SSR（Server Side Render）、后台管理平台场景、定时任务、数据处理等。

- **BFF 场景**：BFF 层主要为前端页面提供数据，采用 Serverless 模式，前端同学不需要考虑不擅长的运维环节，轻松实现了 BFF 向 SFF（Serverless For Frontend）模式的转变。
- **CSR/SSR 场景**：CSR/SSR 指的是客户端渲染和服务端渲染，有了 Serverless 平台，不用考虑运维环节，更多的前端业务来尝试使用 SSR 来实现前端首屏的快速展现。
- **后台管理平台场景**：公司有很多的后台管理平台的 Web 服务，它们虽然相较于函数是比较重的，但完全可以直接托管 Serverless 平台，充分享受 Serverless 平台极致的发布和运维效率。
- **定时任务场景**：公司存在很多周期性任务，如每隔几秒拉取数据，每天 0 点清理日志，每小时收集全量数据并生成报表等，Serverless 平台直接与任务调度系统打通，只需写好任务的处理逻辑并在平台上配置定时触发器，即完成定时任务的接入，完全不用管理机器资源。
- **数据处理场景**：将 MQ Topic 作为事件源接入 Serverless 平台，平台会自动订阅 Topic 的消息，当有消息消费时，触发函数执行，类似定时任务场景，作为用户也只需写好数据处理的逻辑并在平台上配置好 MQ 触发器，即完成 MQ 消费端的接入，完全不用管理机器资源。

5.2 落地收益

Serverless 的收益是非常明显的，尤其在前端领域，大量的业务接入已是最好的说明。具体收益，从以下两个方面分别来看：

- **降成本**：通过 Serverless 的弹性伸缩能力，高频业务资源利用率能提升到 40% ~ 50%；低频业务函数通过合并部署，也能极大降低函数运行成本。
- **提效率**：整体研发研发效率提升约 40%。
  - 从代码开发来看，提供完备的 CLI、WebIDE 等研发工具，能够帮助开发同学生成代码脚手架，聚焦编写业务逻辑，快速完成本地测试；另外，让业务服务零成本具备在线查看日志与监控的能力。
  - 从发布来看，通过云原生的模式，业务无需申请机器，发布、回滚都是秒级别的体验。另外，还能利用平台天然能力，配合事件网关，实现切流、完成金丝雀测试等。
  - 从日常运维来看，业务无需关注机器故障、资源不足、机房容灾等传统模式该考虑的问题，另外，当业务进程异常时，Nest 能够自动完成异常实例的隔离，迅速拉起新实例实现替换，降低业务影响。

6. 未来规划

- **场景化解决方案**：接入 Serverless 的场景众多，如 SSR、后台管理端、BFF 等，不同的场景有不同的项目模板、场景配置，如伸缩配置、触发器配置等，另外，不同的语言，配置也有所不同。这无形中增加了业务的使用成本，给新业务的接入带来了阻碍。因此，我们考虑场景化的思路来建设平台，将平台的能力与场景强关联起来，平台深度沉淀各场景的基本配置和资源，这样不同的场景，业务只需要简单的配置就可以将 Serverless 玩转起来。
- **传统微服务 Serverless 化**：传统微服务的 Serverless 化即是路线选型中提到的面向应用的 Serverless 服务。在美团使用最广的开发语言是 Java，公司内部存在大量的传统的微服务项目，这些项目如果都迁移到函数模式，显然是不现实的。试想如果这些传统的微服务项目不用改造，也能直接

享受 Serverless 的技术红利，其业务价值不言而喻。因此，传统微服务的 Serverless 化是我们未来拓展业务的一个重要方向。在实施路径上，我们会考虑将服务治理体系（如 ServiceMesh）与 Serverless 做技术融合，服务治理组件为 Serverless 提供伸缩指标支持并在伸缩过程中实现精准的流量调配。

- **冷启动优化**：当前虽然函数的冷启动优化已经取得了较好的成绩，尤其是平台侧的系统启动耗时，提升空间已经非常有限，但业务代码自身的启动耗时还是非常突出，尤其是传统 Java 微服务，基本是分钟级别的启动耗时。因此，后续我们的冷启动优化会重点关注业务自身的启动耗时，争取极大降低业务自身的启动时间。在具体优化方法上，我们会考虑采用 AppCDS、GraalVM 等技术，降低业务自身启动耗时。
- **其他规划**
  - **丰富完善研发工具**，提升研发效率，如 IDE 插件等。
  - **打通上下游技术生态**，深度融入公司现有技术体系，减少因上下游平台带来使用障碍。
  - **容器轻量化**，轻量化的容器能够带来更优的启动耗时以及更佳的资源利用率，因此，容器轻量化一直是 Serverless 的不懈追求。在具体落地上，准备联合容器设施团队一起推进容器中的一些 Agent 采用 DaemonSet 方式部署，下沉到宿主机，提升容器的有效载荷。

作者简介

殷琦、华珅、飞飞、志洋、奕锟等，来自基础架构部应用中间件团队。
佳文、凯鑫，亚辉等，来自金融技术平台大前端团队。

招聘信息

美团基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到：tech@meituan.com。

算法平台在线服务体系的演进与实践

作者：永波　季尚　艳伟　非凡

0. 写在前面

AI 可以说是目前互联网行业炙手可热的“明星”。无论是老牌巨头，还是流量新贵，都在大力研发 AI 技术，为自家的业务赋能。美团很早就开始探索不同的机器学习模型在各种业务场景的应用，从最开始的线性模型、树模型，再到近几年的深度神经网络、BERT、DQN 等，并成功应用于搜索、推荐、广告、配送等业务，也取得了较好的效果与产出。

美团配送技术部建设的算法平台——Turing（下称图灵平台），旨在提供一站式的服务，覆盖数据预处理、特征生成、模型训练、模型评估、模型部署、在线预测、AB 实验、算法效果评估的全流程，降低了算法工程师的使用门槛，帮助他们脱离繁琐的工程化开发，把有限的精力聚焦于业务和算法逻辑的迭代优化。具体的实践，大家可参考美团技术团队此前推送的一篇技术博客《一站式机器学习平台建设实践》。

随着机器学习平台、特征平台、AB 平台等陆续完成，配送技术团队发现在线预测部分逐渐成为算法开发和迭代的瓶颈，为此，我们开始启动图灵在线服务框架的整体研发。本文将与大家详细探讨图灵平台中的在线服务框架——图灵 OS（Online Serving）的设计和实践，希望对大家能够有所帮助或者启发。

随着图灵平台逐渐成熟，包括美团配送在内，已经有超过 18 个业务方接入了图灵平台，整体概况大致如下：共接入 10+ 个 BU（业务单元），100% 覆盖美团配送核心业务场景，支持 500+ 个在线模型、2500+ 个特征、180+ 个算法策略，每天支持百亿次的在线预测。通过图灵平台赋能，算法迭代周期由天级别降至小时级别，大幅提升了配送算法的迭代效率。

1. 图灵平台介绍

图灵平台是一站式算法平台，总体架构如下图 1 所示，底层依托于 Kubernetes 和 Docker，实现了对 CPU/GPU 等资源的统一调度和管理，集成了 Spark ML、XGBoost、TensorFlow 等机器学习 / 深度学习框架，包含特征生产、模型训练、模型部署、在线推理、AB 实验等一站式平台功能，支撑了美团配送及闪购、骑行、买菜、地图等事业部的调度、时间预估、配送范围、搜索、推荐等各类 AI 应用。图灵平台主要包括机器学习平台、特征平台、图灵在线服务（Online Serving）、AB 实验平台四大功能。

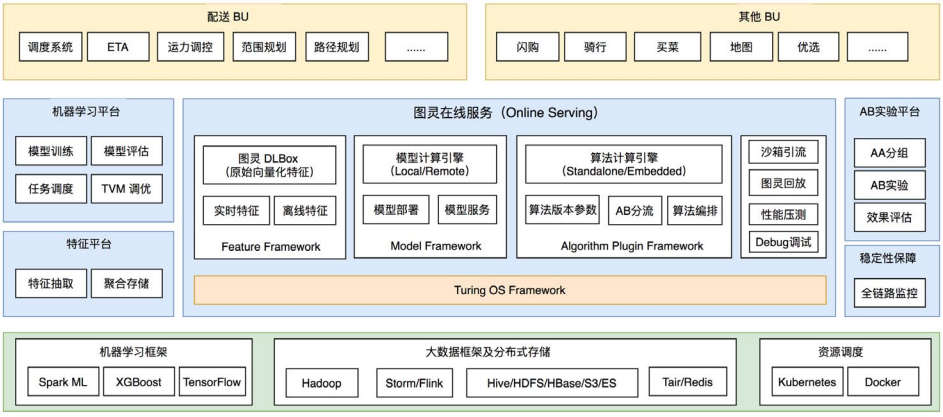


图 1　图灵平台总体架构

- **机器学习平台**：提供模型训练、任务调度、模型评估和模型调优等功能，基于 DAG 实现拖拽式的可视化模型训练。
- **特征平台**：提供在线和离线特征生产、特征抽取和特征聚合等功能，并推送到在线的特征库，提供高性能的特征获取服务。
- **图灵在线服务**：Online Serving，以下简称图灵 OS，为特征获取、数据预处理、模型和算法策略的线上部署及高性能计算提供统一的平台化解决方案。
- **AB 实验平台**：提供事前的 AA 分组，事中的 AB 分流和事后的效果评估等功能，覆盖 AB 实验的完整生命周期。

图灵 OS 主要指图灵平台的在线服务模块，聚焦于机器学习 / 深度学习在线服务，目标是让离线训练好的模型能够快速上线，有效提升各业务部门的算法迭代效率，快速拿到结果，对业务产生价值。以下将重点介绍图灵在线服务（Turing Online Serving）。

2. 图灵 OS 的建设背景

在美团配送业务发展初期，为了支撑业务的快速发展，快速支持算法上线、快速试错，各个业务线的工程方独自开发在线预测的一系列功能，也就是我们所熟知的“烟囱模式”。此种模式各自为战，非常灵活，能够快速支持业务的个性化需求。但随着业务规模的逐渐扩大，这种“烟囱模式”的缺点就凸显了出来，主要表现在以下三个方面：

- **重复造轮子**：特征获取和预处理、特征版本切换、模型加载和切换、在线预测和 AB 实验等都是各自研发，从零做起。
- **平台化能力缺失**：缺乏对特征、模型迭代上线的完整生命周期的平台化运维、管理、监控和追踪能力，研发效率低下。
- **算法与工程耦合严重**：算法与工程边界模糊，耦合严重，相互制约，算法迭代效率低下。

“烟囱模式”在业务发展早期做出了不可磨灭的贡献，但随着业务体量的增长，这种方式的边际收益逐渐降低到了不可忍受的程度，亟需一个统一的在线服务框架来进行改变。

目前，市面上大部分主流开源的机器学习在线服务框架仅提供了模型预测功能，不包含预处理和后处理模块，如下图 2 所示。

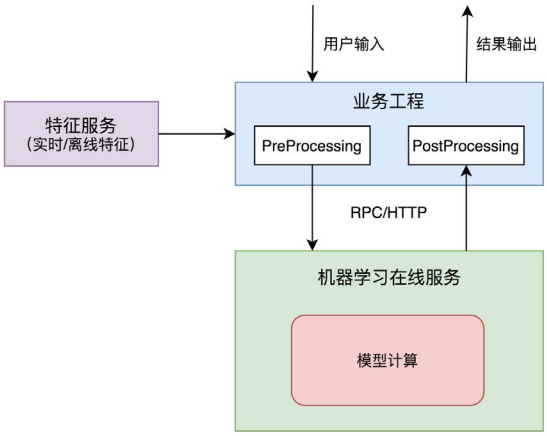


图 2　机器学习在线服务示意图

比如谷歌 TensorFlow Serving 是一个用于机器学习模型 Serving 的高性能开源在线服务框架，提供 gRPC/HTTP 接口供外部调用，支持模型热更新与自动模型版本管理，同时解决了资源调度、服务发现等痛点，对外提供稳定可靠的服务。但是 TensorFlow Serving 不包含预处理和后处理模块，需要将业务工程方将输入预处理成张量传递给 TensorFlow Serving 进行模型计算，然后再对模型计算结果进行后处理。预处理和后处理的逻辑对于算法策略非常重要，迭代也比较频繁，这部分跟模型结合比较密切，更适合由算法同学负责，如果由工程方实现，则工程同学只是单纯的实现算法同学设计的逻辑，耦合过于严重，迭代效率低，而且还容易导致设计和具体实现不一致，引发线上事故。

为了解决上述问题，为用户提供更方便易用的算法平台，图灵平台建设了统一的在线服务框架，通过整合模型计算和预处理 / 后处理等模块，以算法版本的形式进行呈现，并进行迭代，免去了与算法与工程之间复杂的交互。

这里我们对算法定义进行了扩展，本文中的算法（也称算法策略）可以理解成一个组合函数：y=f1(x)+fi(x)+…+fn(x)，其中 fi(x) 可以是规则计算、模型计算（机器学习和深度学习）或者非模型算法计算（比如遗传算法、运筹优化等）。该组合函数中任何组合因子的调整（比如模型输入输出变更、模型类型变更或者规则调整）都可看作是

一次算法版本的迭代。算法迭代是算法开发 – 上线 – 效果评估 – 改进的循环过程。Turing OS 的目标就是优化算法的迭代效率。

3. 图灵 OS 1.0

3.1 图灵 OS 1.0 介绍

为了解决“烟囱模式”开发过程中的重复造轮子和平台化能力缺失的问题，我们着手搭建了图灵 OS 1.0 框架。该框架整合了模型计算和预处理、后处理模块，把繁杂的特征获取和预处理、模型计算、后处理等逻辑都封装在图灵在线服务框架中以 SDK 的形式对外提供。算法工程师基于图灵在线服务 SDK 开发个性化的预处理和后处理逻辑；业务工程集成图灵在线服务 SDK 和算法包，调用 SDK 提供的接口进行模型计算和算法计算。

通过图灵 OS 1.0，我们解决了各业务方独自开发、独自迭代以及重复造轮子的问题，大大简化了算法工程师和工程研发人员的开发工作，而且工程是通过图灵在线服务框架间接调用算法预处理和模型计算，不直接跟算法进行交互，一定程度上也减轻了工程和算法的耦合问题。

如图 3 所示，该阶段的图灵在线服务框架集成了以下功能：

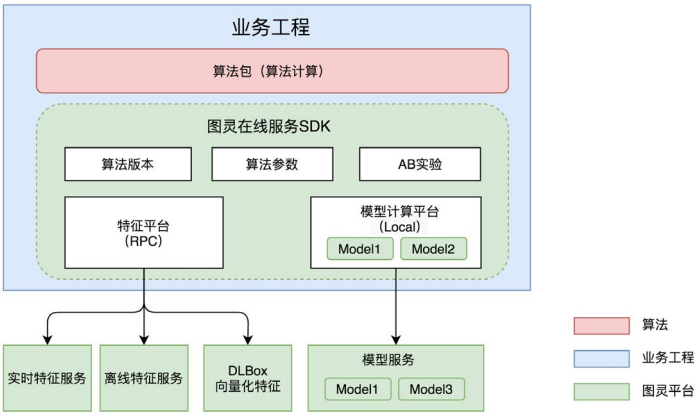


图 3　图灵 OS 1.0

3.1.1 特征获取

- 通过特征聚合、动态分组、本地缓存以及业务线级别物理资源隔离等手段，提供高可用、高性能的特征在线获取计算能力。
- 通过自定义 MLDL（Machine Learning Definition Language）将特征获取流程配置化，并统一特征获取流程，提升在线服务特征的易用性。
- DLBox（Deep Learning Box）支持将原始向量化特征和模型放在同一节点进行本地计算，解决深度学习场景下需要召回大规模数据的性能问题，支撑配送各个业务高并发及算法快速迭代。

3.1.2 模型计算

- 支持本地（Local）和远程（Remote）两种模型部署模式，分别对应将模型部署在业务服务本地和专用的模型在线服务集群中；通过多机异步并行计算，支持 CPU/GPU 资源异构等手段，解决大规模模型计算的性能问题；通过模型 Sharding 解决超大规模模型单机无法装载的问题。
- 在深度学习模型计算方面，利用高性能计算加速库 MKL-DNN 以及 TVM 等编译优化技术进一步提升深度学习模型的推理性能。
- 通过 MLDL 封装的模型特征关联关系以及预处理逻辑等配置，实现了特征获取、特征处理以及组装的自动化，提升了模型的开发迭代效率。

3.1.3 算法计算

- 支持算法版本管理、AB 路由，支持动态获取算法版本所关联的模型、特征和参数等，支持模型和参数的热更新。
- 支持 AB 实验以及灵活的灰度发布放量，并通过统一埋点日志实现 AB 实验效果评估。

3.2 图灵 OS 1.0 遗留问题

图灵 OS 1.0 解决了各业务线重复造轮子、特征混乱和平台能力缺失等问题，通过提供一站式平台化服务，支撑了美团配送各业务线大规模算法在线预测的场景和高性能计算的需求；使算法同学更加关注算法策略本身的迭代优化，提高了算法迭代的效率。但是对于前述的工程、算法、平台三方耦合问题，还没有很好的解决，主要体现在：

- 业务工程静态依赖算法包，算法包部署在业务工程中，算法包更新迭代上线需要业务工程发版。
- 算法包与业务工程运行在同一个 JVM 中，虽然减少一次 RPC 消耗，但是算法包的计算性能会影响业务工程的性能，业务工程稳定性不可控，比如 TensorFlow 模型计算时对 CPU 的消耗过大、大模型的加载和切换对内存的消耗等问题。
- 随着图灵平台提供的功能越来越丰富，图灵在线服务 SDK 变得越来越臃肿，业务工程必须升级图灵在线服务 SDK 才能使用图灵平台新功能，但是业务工程升级 SDK 风险较高，而且会拖慢业务工程部署的速度。

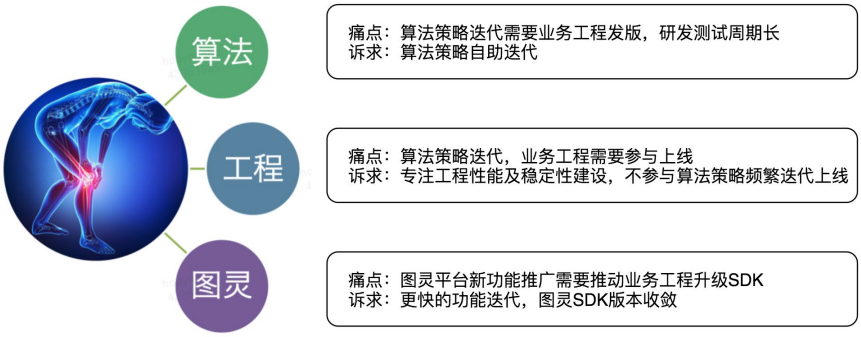


图 4　三方高耦合示意图

基于上述几点可知，算法、工程和图灵平台三方高耦合，导致各自都存在很多痛点，如图 4 所示。这些问题严重影响了算法迭代效率，算法迭代上线测试工期长，效率低：

- **算法痛点**：算法包迭代强依赖业务工程上线，每次工程发版都需要走一个完整的研发测试周期，流程长，效率低。
- **工程痛点**：算法包与业务工程在同一个 JVM 中，算法计算的性能将影响业务工程服务的性能；同时业务工程需要跟随算法包的迭代频繁发版，改动可能只涉及升级算法包的版本。
- **图灵平台痛点**：图灵在线服务 SDK 部署在业务工程中，版本收敛难度大，兼容难度大；同时图灵新功能推广难度大，需要业务工程升级图灵在线服务 SDK。

因此，必须将算法、工程和图灵平台更好的解耦，既满足算法快速迭代的需求，又能满足业务工程端稳定性的诉求，合作共赢。

4. 图灵 OS 2.0

针对图灵 OS 1.0 框架中算法、工程和图灵平台三方高耦合的痛点，我们研发了图灵 OS 2.0 框架，目标是解决算法、工程、图灵平台三者耦合的问题，让算法迭代无需依赖工程发版，图灵平台新功能上线无需业务工程升级 SDK，进一步提升算法迭代效率和工程开发效率。

围绕解耦算法、工程和图灵平台的目标，在图灵 OS 2.0 框架中，我们设计研发了算法包插件化热部署框架、算法数据通道和算法编排框架等功能，支持算法自助迭代上线。同时设计研发了以沙箱引流、实时回放、性能压测和 Debug 测试等功能为一体的算法验证平台，保证了算法策略的高性能、正确性及稳定性。图灵 OS 2.0 框架解耦了算法、工程和图灵平台，实现了算法与工程迭代的各自闭环。大部分算法迭代的整个流程无需工程研发人员、测试工程师的参与，算法工程师在小时级即可完成算法策略的迭代上线；通过图灵 OS 2.0 的赋能，算法的研发迭代效率得到了大幅提升。

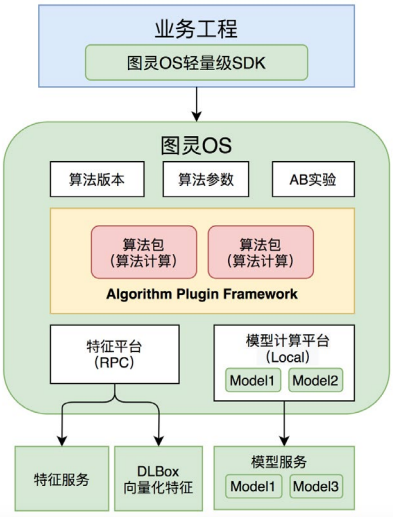


图 5　图灵 OS 框架 V2.0

图灵 OS 2.0 具体功能特性如下：

- **标准化轻量级 SDK**：业务工程只需依赖一个轻量级的图灵 OS SDK，无需频繁升级，降低工程端接入难度，解耦业务工程与图灵平台。
- **算法插件化**：自研图灵算法插件框架，支持算法包作为一个插件在图灵 OS 服务中热部署，解耦算法与工程；图灵 OS 服务中可部署多个算法包的多个版本，每个算法包拥有独立的线程池资源。
- **数据通道**：在一些复杂的算法场景下，算法策略还需依赖业务工程完成：1）算法内部获取数据，只能通过业务工程调用接口获取结果之后传递给算法；2）算法内部调用算法，只能通过业务工程中转同时调用算法 A 和算法 B。为了解决上述两点，我们提出了数据通道（Data Channel）的概念，使得算法本身具备自主获取数据的能力，而不是所有数据都需要业务工程获取然后再透传给算法。
- **算法编排**：多个算法按照串行或者并行的方式组合为有向无环图图（DAG），可以看作是一个算法编排；业务算法的抽象与沉淀，对应到新架构就是算法的组合与编排，算法编排为业务上线和算法迭代进一步赋能，进一步提升了业务算法迭代效率，进一步解耦算法和工程。
- **沙箱引流**：图灵沙箱是一个与图灵 OS 物理隔离，但运行环境完全一致的服务，流量经过沙箱不会对线上业务造成任何影响；沙箱可验证算法逻辑的正确性，同时评估算法计算的性能，提升研发测试流程的效率。
- **图灵回放及统一埋点**：在算法计算及模型计算的过程中会产生很多重要数据（算法策略、模型、特征、参数和数据通道等相关数据），这些数据不仅有助于快速排查定位系统问题，也为 AB 实验报告、沙箱引流和性能压测等模块提供了重要的数据基础，为了更好地自动记录、存储和使用这些数据，我们设计了实时回放平台和统一埋点。
- **性能压测**：图灵 OS 通过整合美团全链路压测系统 Quake 的能力，复用统一回放平台采集的流量数据来构造请求，对部署了新版本算法包的沙箱进行压力测试，保证算法策略迭代的性能及稳定性。

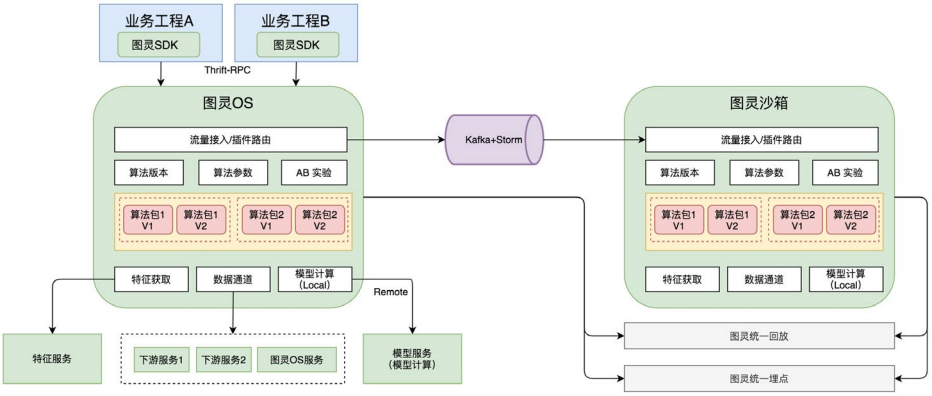


图 6　图灵 OS 2.0 总体架构

以下将对上述几个功能特性进行展开介绍，看看图灵 OS 2.0 是如何解决算法、工程和图灵平台三方耦合痛点的。

4.1 标准化轻量级 SDK

为了解决业务工程和图灵平台的耦合痛点，即图灵在线服务 SDK 部署在业务工程中，SDK 版本收敛难度大的问题，我们主要从 SDK 轻量化、简单易接入、稳定可扩展、安全可靠等几个方面考虑对图灵在线服务 SDK 进行了拆分和改造：

- **SDK 轻量化**：将原有图灵 OS SDK 逻辑下沉到图灵 OS 服务中，只提供简单通用的批量预测接口；该 SDK 无需过多暴露算法相关的细节，算法版本路由、实时 / 离线特征获取、模型计算等都隐藏到图灵 OS 内部。轻量级的 SDK 内部集成了图灵 OS 的自定义路由，业务方无需关注算法包部署在哪个图灵 OS 集群，对使用方完全透明。
- **简单易接入**：提供统一且通用的 Thrift 接口进行算法计算，使用 Protobuf/Thrift 来定义算法输入输出，相对于目前 Java 类定义接口的优势是兼容性有保障；Protobuf 接口定义完成后，算法和工程可以各自独立进行代码开发。
- **可扩展**：轻量级 SDK 版本稳定，无需工程端反复升级；Protobuf 天然支持序列化，后续流量拷贝、回放埋点等都可以基于此进行。
- **高性能**：针对大批量算法计算且要求高可用的场景，例如面向 C 端用户的批量

预测，我们设计了异步分批高度并行等手段提升算法计算性能；针对单任务计算耗时长、CPU 消耗高且要求高可用的场景，例如分城市区域的调度路径规划，我们设计了客户端快速失败最优重试机制保证高可用，也均衡了图灵 OS 的计算资源。

- **安全可靠**：针对单个图灵 OS 部署多个算法包的场景，提供线程池级别的资源隔离，针对各业务线不同的算法包，按业务场景垂直拆分，提供物理级别集群资源隔离，同时增加熔断降级机制，保证计算流程稳定可靠。

4.2 算法插件化

通过对图灵 OS SDK 进行标准化轻量化改造，我们解决了业务工程和图灵平台之间耦合的痛点。通过对图灵 OS 进行服务化改造，解决了算法和业务工程之间耦合的痛点。但是算法和图灵平台之间耦合的痛点依然存在且痛点增加：算法迭代上线依赖图灵 OS 服务发版，并未能达到三方解耦的目标。

为了解决算法与图灵平台之间的耦合痛点，进一步提升算法策略的迭代效率，我们下一步的设计思路是算法插件化，图灵 OS 容器化：将算法包作为一个插件，部署到图灵 OS 中，算法包发版不要求图灵 OS 发版，甚至不需要重启图灵 OS，如图 7 所示。

- **算法插件化**：我们自研了图灵 OS 算法插件框架，支持算法包以插件的形式部署到图灵 OS 服务中；具体实现方案是自定义算法类加载器 ClassLoader，不同的 ClassLoader 加载不同的算法包版本，通过加载多版本算法包以及指针替换，实现算法包热部署。
- **图灵 OS 容器化**：图灵 OS 充当一个插件容器，装载算法包不同的算法版本，执行算法版本路由以及算法策略计算，图灵 OS 经过容器化改造之后的流程：1）如果算法版本不需要新增参数，则工程端和图灵 OS 都不需要发版；2）业务工程主要工作是传递参数给算法，逻辑简单，如输入参数无变化则不需要发版，算法包发版节奏自己掌控。

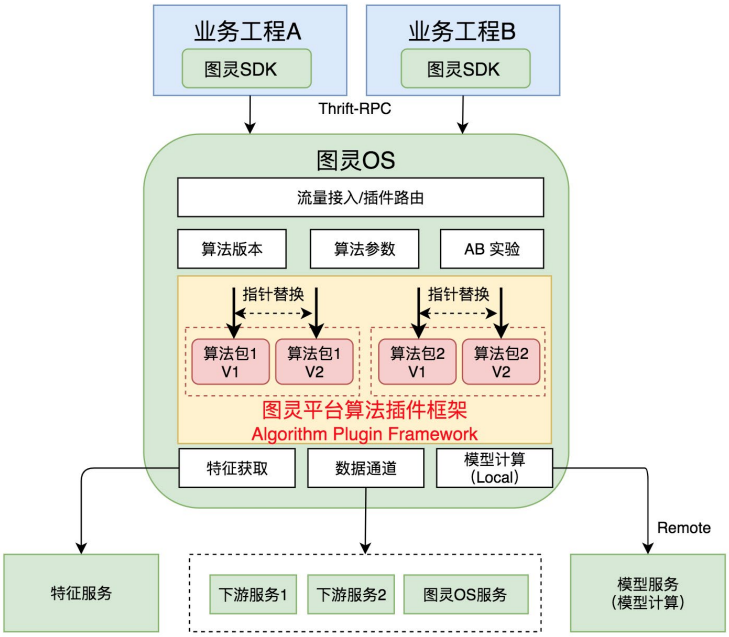


图 7　图灵 OS 容器化 – 算法插件化示意图

4.3 数据通道

通过上述手段，我们解决了算法、工程和图灵平台三者在发版迭代时的耦合问题。但是除了上述的耦合之外，还有一些复杂算法场景，算法与业务工程依然存在耦合，主要体现在算法依赖业务工程的以下两点数据：

- **算法内部获取数据**：目前是通过业务工程调用接口获取结果之后传递给算法，例如一些服务化接口数据、分布式 KV 缓存数据等，算法和业务工程都需要进行开发迭代上线。
- **算法内部调用算法**：目前通过业务工程同时调用算法 A 和算法 B 并编写中转逻辑来实现，例如算法 A 的输入需要用到算法 B 的结果，或者需要综合算法 A 和算法 B 的结果得到最终输出，这些操作一般都交由业务工程来处理。一种可选方案是将算法 A 和算法 B 合并成一个庞大的算法，但该方案的劣势是增加了算法 A 和算法 B 独立进行 AB 实验及灰度的研发成本。

为了解决上述两点，我们提出了数据通道（Data Channel）的概念，使算法本身具备自主获取数据的能力。在算法内部算法可通过图灵 OS 提供注解的方式支持数据通道，算法与业务工程的交互接口仅需传递一些关键参数及上下文数据即可，算法内部自行组装该数据通道所需参数。经过数据通道化的改造，算法接口进一步简化，算法与工程耦合度进一步降低，算法内部调用算法的问题，我们可通过下面介绍的算法编排来进行解决。

4.4 算法编排

一个完整的算法计算流程包括算法计算部分，以及针对输入的预处理逻辑和计算结果的后处理逻辑等，算法计算可以是 N 次规则计算，N 次模型计算（机器学习和深度学习等），或者非模型的算法计算（比如遗传算法、运筹优化等），或者多种类型算法组合。我们把这种具有独立输入输出的计算逻辑单元抽象为一个算子，算子可编排、可复用，通用的两类算子如下：

- **模型计算算子**：即模型计算引擎执行模型计算，我们支持 Local 和 Remote 两种模型计算模式，在 Remote 计算模式中，模型可能部署在不同的模型集群中，算子是对模型计算的进一步封装，将 Local 和 Remote 选择及模型集群路由等功能对用户透明，算法工程师无需感知，我们会根据整体计算性能进行动态调整。
- **算法计算算子**：即图灵 OS 中的算法计算引擎执行算法策略计算，不同的算法插件可能部署在不同的图灵 OS 中，同时也将图灵 OS 集群的路由功能进行了封装，对用户透明。

多个算子之间通过串行或者并行的方式组合为一个有向无环图（DAG），形成了算子编排，当前我们有两种方式实现算子编排：

- **算法数据通道**：不同图灵 OS 中的算法计算引擎互相调用或者算法计算引擎调用模型计算引擎，算法数据通道是实现算子编排的一种具体手段。
- **算法总控逻辑**：我们在算法调用的上层抽离出一层算法总控逻辑层，满足复杂

算法场景及多个算法关联依赖的情况，该算法总控逻辑由算法工程师在算法包中实现；通过算法总控逻辑功能，算法工程师可以任意编排算法之间的关系，进一步解耦算法和工程。

从算法工程师的视角来看，图灵 OS 以搭积木的方式提供服务，通过组合一个个独立的子功能及算子，以标准的方式串并联，从而形成满足各式各样需求的在线系统。

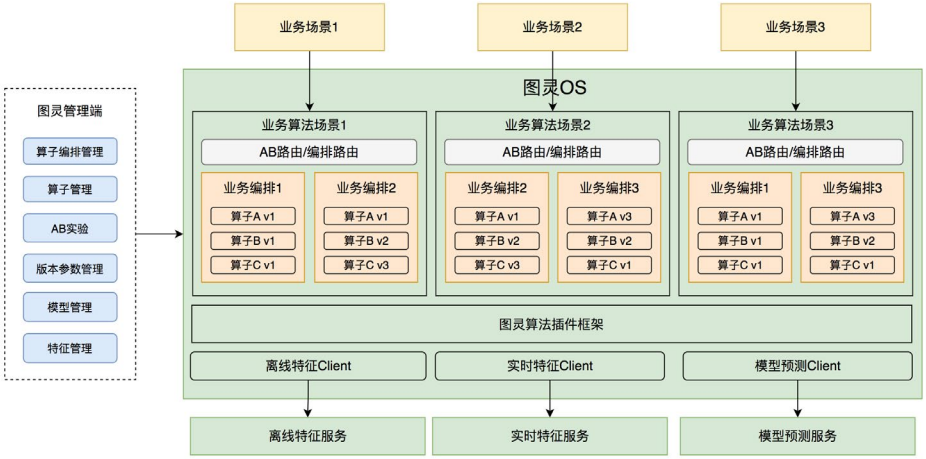


图 8　基于算子编排的算法在线服务架构

在该架构下，算法的工作主要有如下三部分：1）算法工程师进行业务流程的抽象与建模；2）算法工程师进行独立的算子开发与测试；3）算法工程师基于业务流程抽象进行算子的编排与组合。算子编排为业务功能上线和算法迭代进一步赋能，业务算法迭代效率进一步提升。

4.5 多模式集成

上文介绍了图灵 OS 作为一个容器可部署多个算法包的多个版本，并支持算法包热部署。图灵 OS 通过插件化热部署以及编排等功能，解耦了业务工程、算法以及图灵的三方耦合，极大地提升了算法的迭代效率。为了进一步满足业务的要求，我们提供了两种图灵 OS 部署集成模式：Standalone 模式和 Embedded 模式。

Standalone（独立模式）

Standalone 模式下，图灵 OS 是独立于业务服务单独部署的，业务服务通过轻量级 SDK 调用算法，图灵轻量级 SDK 内部封装了图灵 OS 的自定义路由，以及 Thrift-RPC 调用图灵 OS 服务的逻辑。

Embedded（内嵌模式）

在某些高并发及高性能要求的复杂场景中，对我们图灵 OS 的集成模式及性能提出了更高的要求。在独立部署模式下，业务工程每一次算法计算都有 RPC 的消耗，因此我们实现了图灵 OS 新的集成模式——Embedded。在 Embedded 模式下，我们对外提供图灵 OS 框架代码包，业务方在自己的工程服务中集成图灵 OS 框架包，业务服务同时也作为一个图灵 OS 容器，还是通过轻量级 SDK 调用算法，在业务服务本地进行算法计算。内嵌图灵 OS 的特点如下：

- 业务工程因集成了图灵 OS 框架代码，而继承了算法包插件化和热部署的功能，具备了业务功能和图灵 OS 容器的双重属性。
- 业务工程并不直接依赖算法包，而是由图灵 OS 框架进行动态管理，算法包进行插件化热部署，达到了算法和工程解耦的目的。
- 业务工程直接进行本地算法计算，减少了算法调用的 RPC 及序列化消耗，同时复用了业务工程服务器资源，进一步减少集群资源消耗，提升了资源利用率。

在算法包插件部署时，以内嵌模式集成的业务工程将作为容器装载相应的算法包，路由到本地进行算法计算，如下图 9 所示。

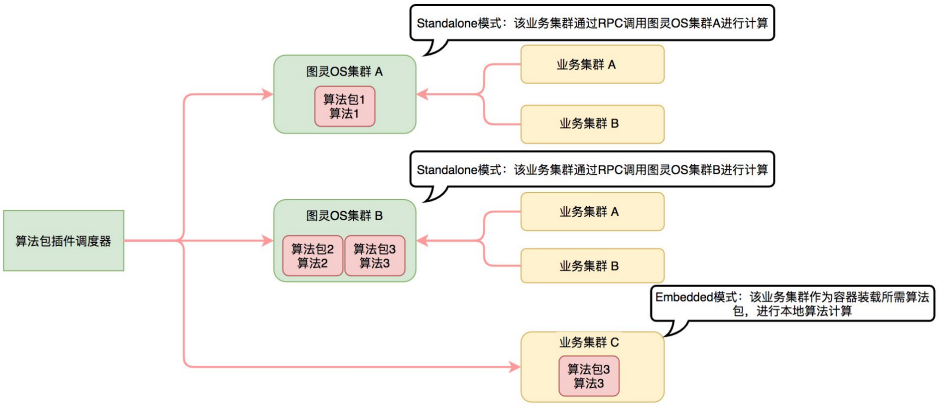


图 9　图灵 OS 集成模式 Embed/RPC 示意图

Standalone 和 Embedded 模式各有利弊，谁都没有绝对的优势，使用时需要根据具体的业务场景进行选择。两种模式的对比如下：

部署模式	优点	缺点	适用场景
Stand-alone	耦合度更低，业务方只依赖图灵轻量级 SDK	需要搭建图灵 OS 集群，占用机器资源；有 RPC 调用开销	适合大批量调用，需要分布式多机异步并行计算的业务场景
Embed-ded	复用业务方机器，资源利用率高；少了 RPC 调用，性能高	无法充分发挥多机异步分布式并行，只能单机并行	适合小批量调用，对单次调用 RT 性能要求较高的业务场景

4.6 图灵沙箱

在图灵 OS 支持算法插件热部署之后，算法迭代效率相比之前大幅提升，算法工程师的上线自由度也得到大幅增加，无需经过业务工程和测试的排期开发和测试；但是也引入了新的问题：

- 算法迭代上线前，无法引线上流量进行预计算，提前对算法效果进行上线前评测，上线前校验难，算法工程师测试效率较低。
- 当前线上实时评估和校验困难，算法策略的线上性能和效果评估缺少流程化自动化工具。
- 频繁的迭代上线对图灵 OS 服务以及业务的稳定性来说也是很大的挑战。

当时的可选方案是算法策略先部署上线，灰度切小流量，然后再分析统一埋点日志评测算法效果。该方案的缺陷是无法在上线前对算法效果进行评测，问题发现时间过晚。如果灰度的功能有问题，会对线上的业务造成影响，产生 Bad Case。针对上述上线前校验环节的各个问题，我们研发了图灵沙箱，在不干扰线上业务稳定的前提下，实现了算法的全链路仿真实验。

图灵沙箱是一个与图灵 OS 服务物理隔离但运行环境完全一致的服务，流量经过沙箱不会对线上业务造成任何影响。如下图 10 所示，线上流量引流到线上环境沙箱，图灵 OS 和图灵沙箱的各环境配置及数据都一致（版本、参数、特征、模型等）。算法新版本（如下图 10 中算法包 1 的版本 V3）先部署沙箱，引流验证算法正确性，同时还可以在沙箱内引流进行算法性能压测。图灵沙箱作为算法验证流程的自动化工具，提升了算法测试效率，进一步提升了算法版本的迭代效率。

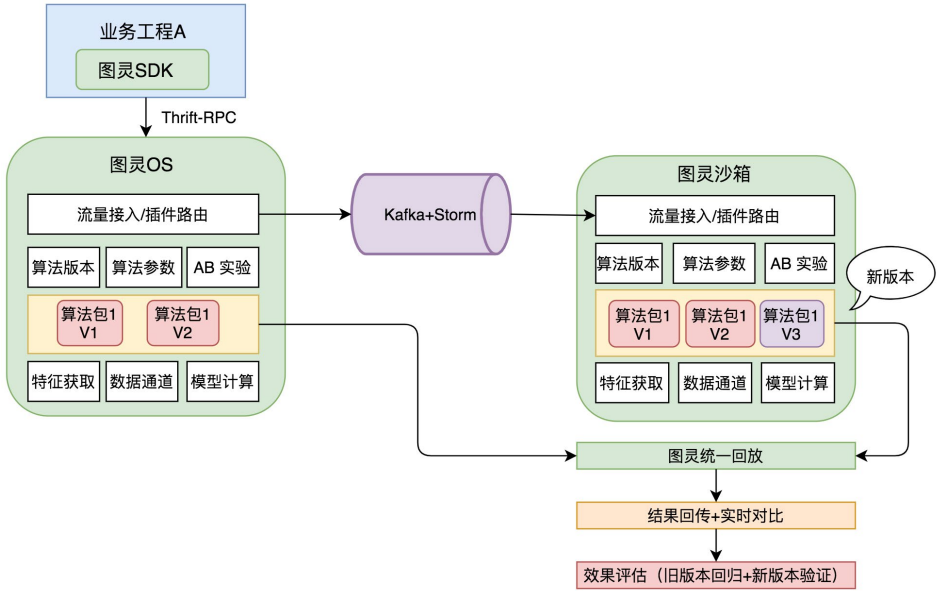


图 10　图灵沙箱引流验证示意图

4.7 统一回放平台

为了方便分析算法效果及异常时排查问题，我们需要把算法计算过程中的输入、输出、所用的特征以及模型等数据都记录下来，以便还原现场。但是算法计算过程中会产生大量的数据，对存储和记录带来了挑战：

- **数据量大**：一次请求可能对应多次算法模型计算，并且往往会用到丰富的特征值，导致中间计算数据数倍于请求量。
- **并发量高**：集中收集存储各图灵 OS 服务产生的数据，需要具备承载这些服务高峰期 QPS 流量之和的能力。
- **定制性强**：图灵 OS 部署了数十种不同的算法，他们的请求和响应格式千差万别，特征和数据源等数据更是难以统一。

为了更好地记录和存储这些重要数据，图灵 OS 设计研发了统一回放平台，针对上述问题给出了解决方案，如下图 11 所示：

- 采取 ES 和 HBase 结合存储回放数据，其中 ES 存储关键索引字段，HBase 存储完整数据记录，充分发挥二者的优势，同时满足了快速查询搜索和海量数据存储的要求。
- 利用 Google Protobuf 的 DynamicMessage 功能，对原始 Google Protobuf 格式进行扩展，动态支持回放数据格式的定义及数据组装，并支持与 ES 索引的同步，既保证序列化和存储的高性能，也保证各算法数据的高效接入。
- 考虑到对这些数据查询的时效性要求不高，使用消息队列将发送和存储进行解耦，达到对流量削峰填谷的效果，图灵 OS 平台中的各算法通过回放 Client 自动接入回放。

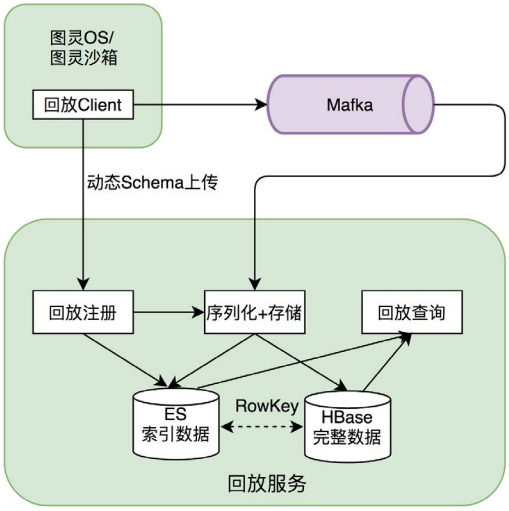


图 11　图灵回放平台示意图

4.8 性能压测及调优

通过图灵沙箱和统一回放，图灵 OS 具备了快速验证算法数据正确性的能力，但是在算法计算性能分析方面缺少自动化工具。图灵 OS 通过整合公司全链路压测系统 Quake（Quake 介绍详见《全链路压测平台（Quake）在美团中的实践》）的能力，复用统一回放平台采集的流量数据来构造请求，对部署了新版算法包的图灵 OS 或图灵沙箱进行压力测试。

压测过程中记录算法在不同 QPS 场景下的性能表现，主要包括 CPU 和内存等应用指标，TP 时延和超时率等响应耗时数据，并与线上真实性能、历史压测数据和服务承诺的 SLA 进行对比分析给出压测报告及优化指南，存在明显性能问题时将阻断算法包的上线流程。图灵 OS 也接入了美团内部性能诊断优化平台 Scalpel，可以生成压测过程中线程堆栈和性能热点的分析报告，辅助用户快速定位性能瓶颈点，为具体优化方向提供参考。

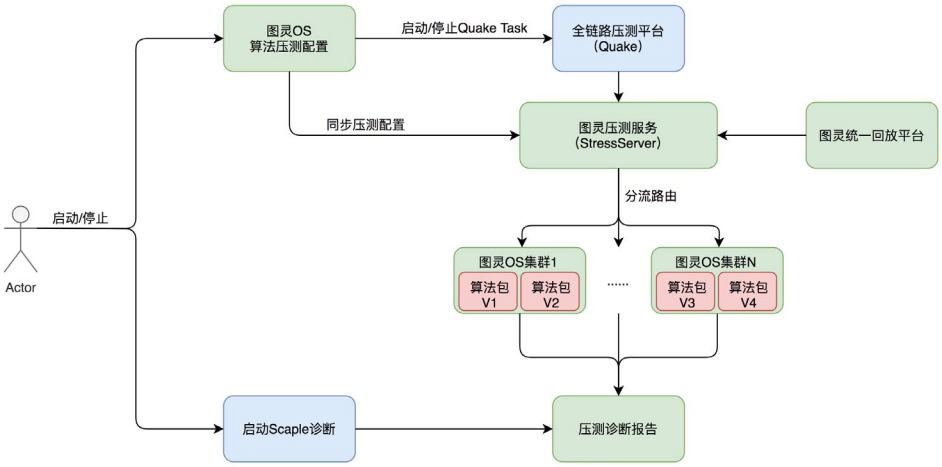


图 12　图灵全链路压测及性能诊断示意图

5. 图灵 OS 2.0 建设成果

5.1 算法研发流程

通过图灵 OS 的算法插件化改造和动态热部署的能力，我们解耦了算法、工程和图灵平台，实现了算法与工程迭代的各自闭环，提升了研发效率，算法迭代上线周期大幅缩短：

- 当模型迭代、特征变更及算法策略迭代时，算法工程师可以自主完成全链路的开发测试，无需工程研发人员和测试工程师的介入；同时算法包可独立部署，无需任何服务上线，上线后周知到工程侧及产品方关注相关指标变化即可。
- 当新业务场景和新算法策略接入时，还需要算法和工程共同开发，定义好 Protobuf 接口之后，算法工程师和工程研发人员可以各自独立开发代码，各自上线。

通过使用图灵 OS 提供的沙箱引流验证和性能压测诊断等自动化工具，算法策略迭代的效率进一步提升，算法迭代上线周期大幅缩短，由天级别提升至小时级别。算法工程师自主开发，然后部署图灵 OS 进行自测调试，部署沙箱进行引流测试，通过压测平台评估效果性能，最后自主部署上线，整个流程无需工程研发人员及图灵工程师的

参与，达到自动运维的目标；同时通过各种手段保证算法策略的执行性能及图灵 OS 的运行稳定性。

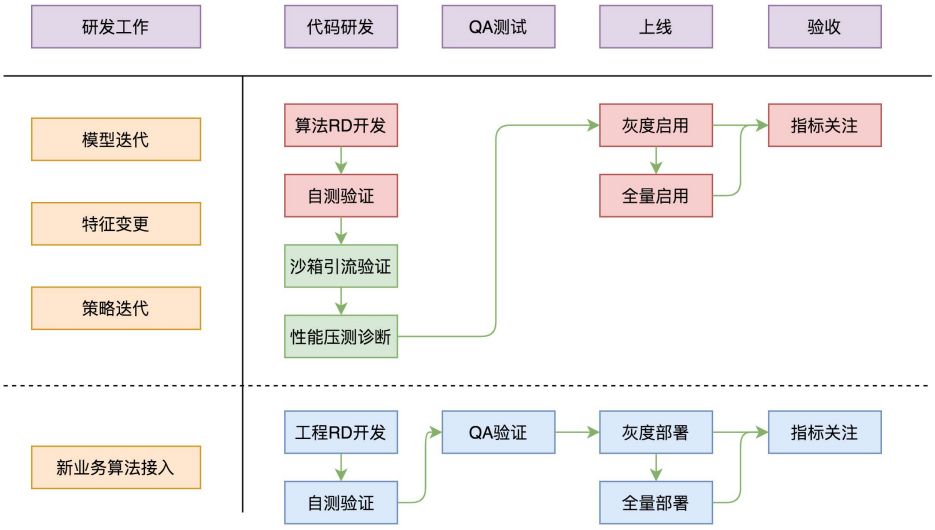


图 13　图灵算法研发流程

5.2 图灵 OS 2.0 使用汇总

图灵 OS（即图灵在线服务框架 2.0）建设已有大半年的时间，整体概况大致如下：当前已搭建 20+ 个图灵 OS 集群，已接入 25+ 个算法包、50+ 个算法，每月算法包部署上线次数 200+ 次；每天支持百亿次算法策略计算。通过图灵 OS 赋能，大部分算法迭代整个流程无需工程研发人员、测试工程师的参与，算法工程师在小时级即可完成算法策略的迭代上线。

当前，一个图灵 OS 集群可承载单业务线的多个算法包或单个部门的多个子业务线算法包，算法包和图灵 OS 集群可动态关联及动态部署，图灵 OS 同时支持业务线级别和算法包级别的物理资源隔离。为了方便业务方的使用，我们提供了完善的接入文档和视频课程。除了图灵平台方搭建图灵 OS 集群之外，任何一个业务方基本上可以在 1 小时内构建出自己的图灵 OS 服务。我们同时提供了最佳实践文档与性能调优配置等，使得业务方在没有指导的情况下可以自行解决大部分问题。目前我们正在建设自

动化运维工具，进一步降低了图灵 OS 的接入门槛和运维成本。

6. 总结及未来展望

当然，肯定没有完美的算法平台及算法在线服务框架，图灵 OS 还有很大的进步空间。随着我们对机器学习和深度学习线上服务的持续探索，会有越来越多的应用场景需要图灵 OS 支持，未来我们会在以下方面持续进行建设：

- 建设图灵 OS 自动化运维工具和自动化测试工具，支持算法半自动化开发，进一步降低平台接入成本和运维成本。
- 进一步完善图灵 OS 框架，完善算法支撑能力，支持在 Spark 环境运行，当算法迭代时，基于海量的数据验证算法新功能的正确性、性能及效果。
- 推进图灵 OS 全图化引擎的建设，通过抽象算法业务的通用组件，提供图形化流程编排工具和图执行引擎，为业务上线和算法迭代进一步赋能，进一步提升迭代效率。

7. 作者简介

永波、季尚、艳伟、非凡等，均来自美团配送技术部算法平台组，负责图灵平台建设等相关工作。

8. 招聘信息

如果你想近距离感受一下图灵平台及图灵 OS 的魅力，欢迎加入我们。美团配送技术团队诚招机器学习平台、算法工程方向等的技术专家和架构师，共同面对复杂业务和高并发流量的挑战，共建全行业最大的即时配送网络和平台，迎接美团配送业务全面智能化的时代。感兴趣同学可投递简历至：houyongbo@meituan.com（邮件标题注明：美团配送技术团队）。

美团终端消息投递服务 Pike 的演进之路

作者：健午　佳猛　陆凯　冯江

1. Pike 的前世今生

1.1 Pike 1.0 的诞生背景

2015 年，美团诞生了 Shark 终端网络通道，为公司移动端提供长连代理加速服务。Shark 通过网络接入点的全球多地部署和保持长连来提升网络请求的端到端成功率，降低端到端延时，从而提升用户体验。

Pike 1.0 是基于 Shark 长连通道实现的应用内推送服务。由于底层传输基于 Shark 长连通道，使得 Pike 1.0 天生便具有了低延时、高可靠、防 DNS 劫持等优秀基因。目前 Pike 1.0 在美团内部的实时互动、营销推送、状态下发、配置同步等业务场景都有广泛使用。

1.2 Pike 1.0 的工作流程

移动端 SDK 会在每次长连接创建成功后，使用 APPID、设备唯一标识 UnionID（美团唯一标识、点评唯一标识等）向服务器发起注册，在注册成功之后业务服务端就可以通过 Pike 1.0 服务端 SDK 提供的接口，主动向设备的 App 推送消息。服务端推送的消息通过长连接通道抵达客户端，最后通过注册的回调接口投递给业务方。整体工作流程参见下图：

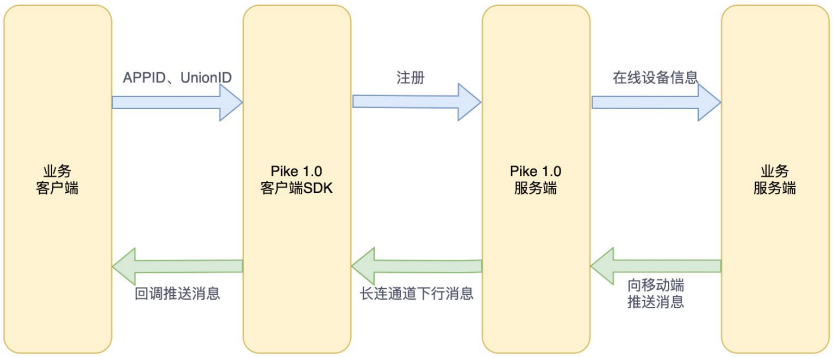


图 1　Pike 1.0 工作流程图

1.3 Pike 1.0 的优势

Pike 1.0 底层传输基于 Shark 长连通道，所以 Pike 1.0 在以下几个方面有不错的表现：

- **防 DNS 劫持**：底层通道直接使用 IP 直连，省去 DNS 解析耗时的同时也避免了被 DNS 劫持的风险。
- **低延时**：Shark 长连接采用就近接入点长连接的方式，省去了传统 HTTP 传输需要多次建连、握手的消耗，端到端数据传输延时相比 HTTP 大幅缩短。
- **安全性好**：Shark 采用自定义二进制协议进行数据传输，进行了通道级别的 TLS 加密，防篡改，更安全。
- **更好的境外体验**：Pike 1.0 与 Shark 共享服务集群，Shark 长连通道在海外多地都部署了接入点，代理加速接入，网络延时及成功率表现要优于常规请求。

1.4 Pike 1.0 的痛点

Pike 1.0 作为 Shark 的衍生产品固然有其闪光的地方，但是对 Shark 的强依赖所带来的痛点更是让开发人员叫苦不迭，主要痛点如下。

1.4.1 代码结构耦合

在客户端 SDK 方面，Pike 1.0 代码与 Shark 代码结构耦合，共用底层通道建连、数据加解密、二进制协议等逻辑。如图展示了 Pike 1.0 与 Shark 在代码结构上的关系。

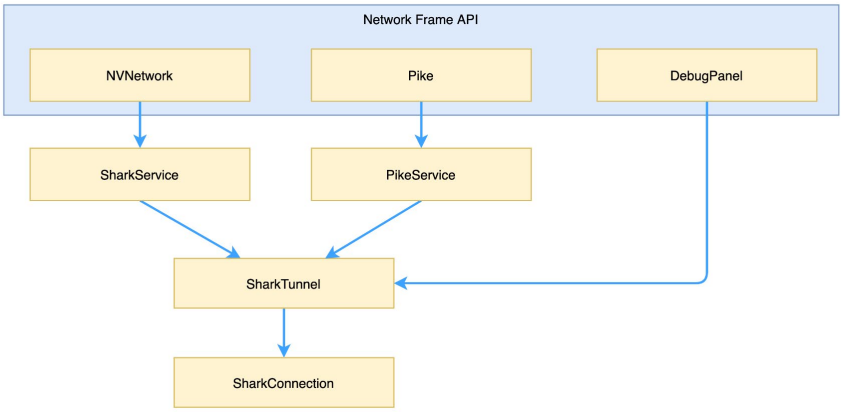


图 2　Pike 1.0 与 Shark 代码结构示意图

耦合带来的弊端一：代码优化升级困难。针对一个 SDK 的变更经常需要更多地考虑对另一个 SDK 是否有负面影响，是否影响面可控，这就无端地增加了开发成本。

耦合带来的弊端二：Shark 与 Pike 1.0 的网络配置环境共用，如图所示，通过 DebugPanel 对 SharkTunnel 进行网络环境配置都会同时对 Shark 和 Pike 1.0 生效，但是业务方在使用的时候往往只关注其中的一个 SDK，不同 SDK 之间的相互影响引入了很多客服问题，也给客服问题的排查带来了较多干扰因素。

1.4.2 账号体系混乱

Pike 1.0 在同一个 App 上只支持一种设备唯一标识 UnionID，不同 App 上注册使用的 UnionID 会有不同，例如美团使用美团唯一标识，点评则使用点评唯一标识。假如一个业务只在一个 App 上使用的话 Pike 1.0 自然可以很好地工作，但是同一个业务有可能需要在多个 App 上同时使用（如图所示），如果业务方不对账号体系进行兼容的话，美团 App 上使用点评唯一标识作为推送标识的业务将无法工作，点评 App 上使用美团唯一标识作为推送标识的的业务也会无法工作。这就导致同一个业务在不同 App 上的推送标识 ID 逻辑会非常复杂，后端要同时维护多套账号体系之间的映射，才能解决账号体系混乱的问题。

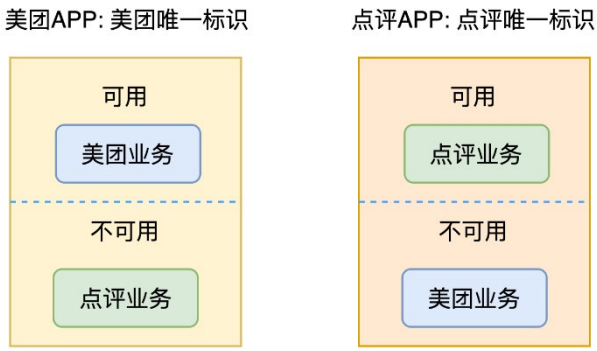


图 3　Pike 1.0 账号体系不兼容示意图

1.4.3 推送连接不稳定

Pike 1.0 由于共用 Shark 的通道逻辑而缺乏推送场景专项优化，在检测通道异常、断连恢复等方面表现不够优秀。在通道可用性上，Shark 与 Pike 1.0 关注的 SLA 也有着很大的不同。

例如，Shark 在长连接通道不可用的情况下，可以通过降级短连接来规避业务网络请求持续失败所带来的成功率下降问题。但是对于 Pike 1.0 此时如果通道不能快速恢复的话就会造成业务消息投送失败，将直接影响消息投递成功率。所以 Shark 通道针对连接保活的公共逻辑并不能完美地应用在 Pike 1.0 业务场景上。

虽然 Pike 1.0 在 Shark 通道的基础上进一步在协议层强化了心跳探测机制以提高通道可用性，但通道不能及时检测异常还是时有发生。此外，Pike 1.0 内部使用的事件分发技术的可靠性还暂时没能达到 100%，零星地会上报一些异常断连而导致推送不成功的客服问题。综上，针对推送连接不稳定专项优化的诉求也就不断被提上日程。

1.5 Pike 2.0 的诞生

Pike 1.0 现有的痛点在业务场景日益丰富的现状下遭遇了诸多挑战。力求解决 Pike 1.0 现有在 Android 和 iOS 平台运营上遇到的问题，一方面我们重新梳理产品架构与代码实现，另一方面我们决定与基础技术部另一个服务于 H5 的消息投递服务 Pike Web 进行产品融合，进而推出全新的升级产品——Pike 2.0。

下图展示了 Pike 2.0 的产品全景，针对 Pike 1.0 的现状，Pike 2.0 前后端都做了诸多优化，包括技术架构升级、集群独立、协议扩展等。其中在客户端方面 Pike 2.0 提供了基于多语言实现服务于多平台的 SDK，在服务端方面 Pike 使用部署 Java 应用的分布式集群来提供服务。

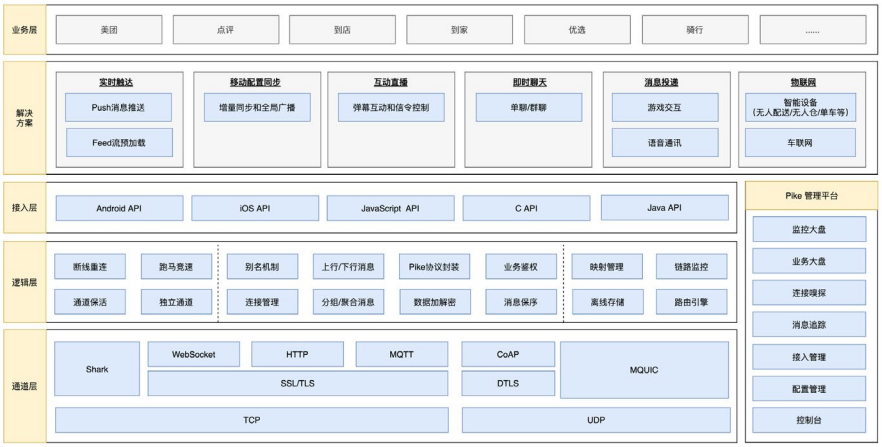


图 4　Pike 2.0 产品全景图

本文主要从客户端视角，详细阐述 Pike 2.0 客户端 SDK 的技术方案设计，从原理上说明 Pike 2.0 带来的技术优势。

2. Pike 2.0 架构设计

针对上文提及的 Pike 1.0 代码结构耦合的痛点，Pike 2.0 进行了全新的架构升级，在代码结构、环境配置、服务集群等方面上都与 Shark 保持产品隔离。

2.1 设计思想

经过接近一年的技术积累与沉淀，从 Shark 提炼的 TunnelKit 长连内核组件和 TNTunnel 通用通道组件已经趋于稳定，所以 Pike 2.0 选择基于 TunnelKit 与 TNTunnel 来构建双向消息通道服务。具体优势有：

- Pike 2.0 基于 TunnelKit 长连内核构建，能有效地复用现有长连接控制相关的

功能，减少不必要的开发工作。

- Pike 2.0 能够共享 TNTunnel 的相关通用特性，如 Shark 协议封装、数据加解密等，后期维护成本较小。
- Pike 2.0 协议作为 Shark 协议的 Payload 传输，可以灵活定制自己特性相关的协议。

2.2 整体架构

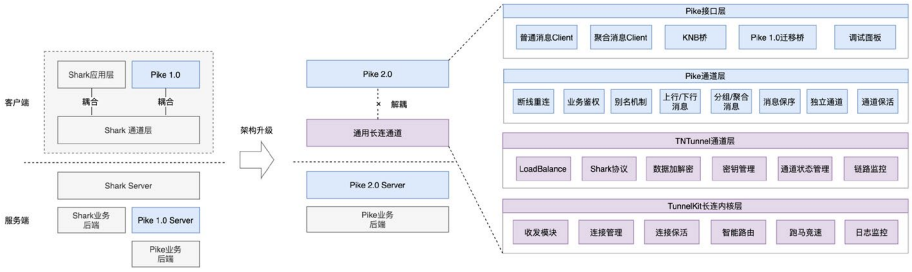


图 5　客户端架构演进图

整体架构如图所示，包括 Pike 接口层，Pike 通道层，TNTunnel 通道层和 Tunnel-Kit 长连内核层。

2.2.1 Pike 接口层

Pike 接口层旨在为主流前端技术栈中所有需要应用内消息服务的业务提供简洁可靠的接口：

- Pike 2.0 提供了 Android、iOS、MRN 等公司主流技术栈的接入 SDK，业务可以根据自己的需求灵活选择。
- Pike 2.0 针对不同的消息 QPS，设计了两种不同 Client。对于消息量超过 50 条每秒的业务，例如直播弹幕推送，我们推荐接入聚合消息 Client；对于消息量较小的其他业务，普通消息 Client 则可以满足需求。
- Pike 2.0 针对线上 Pike 1.0 系统提供了业务无感的迁移方案，业务方无需任何人力投入便可以从之前的 Pike 1.0 系统迁移至 Pike 2.0 系统来进行消息的收发。

2.2.2 Pike 通道层

Pike 通道层是特性的实现层，所有 Pike 接口层的 API 调用都会通过线程调度转变成封装的 Task 在 Pike 通道层完成具体的操作，Pike 通道层是单线程模型，最大程度规避掉了线程安全问题。

Pike 特性如下：

- **断线重连**：鉴于长连接的不稳定特征，Pike 2.0 通道通过断线重连机制来使的业务方可以认为在网络没有发生故障的情况下是持续可用的。
- **业务鉴权**：业务后端可以通过 Pike 2.0 通道对连接的监控来感知连接变更，同时对接入网络的客户端设备进行可用性判别。
- **别名机制**：针对不同业务方对业务标识做了隔离，每个业务可以自定义标识 ID，解决了 Pike 1.0 同一个 App 平台不同业务必须强制使用相同标识 ID 的痛点。
- **上行 / 下行消息**：Pike 2.0 是双向通道服务，不仅支持 Pike 1.0 原有的消息推送能力，即服务端向客户端发送下行消息；同时也支持客户端主动发送消息，即客户端向服务端发送上行消息。业务只要通过 Pike 2.0 系统便可以形成消息的闭环。
- **分组 / 聚合消息**：Pike 2.0 支持消息分组和消息聚合来满足高 QPS 业务场景的使用。其中消息分组表示业务可以通过自定义标签来对一组用户进行消息广播；消息聚合表示将短时间内井喷式的消息进行聚合下发以提高系统的吞吐量。
- **消息保序**：Pike 2.0 支持同一客户端发送的上行消息有序投递到固定的业务服务器。
- **独立通道**：Pike 2.0 默认所有业务是使用一条共享的通道，针对业务量大或者对吞吐量有要求的业务可以自动切换独享的通道来保证消息的投递成功率和时延。
- **通道保活**：Pike 2.0 在连接保活的基础上增加了通道巡检，能够在检测到异常的情况下自动重启通道，确保在要求长稳的环境下进一步提升通道可用性。

2.2.3 TNTunnel 通道层

TNTunnel 通道层是封装通用通道逻辑的功能层。主要涉及通道状态管理、协议封装、数据加解密等通用核心模块，是将通用通道逻辑从原先 Shark 通道中提炼而成的独立分层。Pike 协议虽然是构建在现有 Shark 协议之上的应用层协议，但是 Pike 通道已经和原先的 Shark 通道在逻辑上完全解耦。一方面，Pike 2.0 会最大限度地复用 Shark 协议已成熟的技术，但是又不依赖于原有的 Shark 逻辑；另一方面，后续涉及二进制协议、安全协议等协议方面的升级优化都可以同时服务于 Pike 2.0。

2.2.4 TunnelKit 长连内核层

TunnelKit 长连内核层主要功能是对接 Socket 来处理 TCP 或者 UDP 数据的发送与接收，管理各个连接的可用性等。每条 Pike 2.0 通道在 TunnelKit 中都是维护一条连接的，通过心跳保活机制和连接管理来保证在网络环境正常的情况下永远有一条连接来承载 Pike 数据。TunnelKit 作为所有通道层的基础，是决定上层长连接通道稳定性最重要的一层。

3. Pike 2.0 工作机制

在进行了全新架构升级的基础上，Pike 针对上文提及的 Pike 1.0 账号体系混乱、推送连接不稳定的痛点重新设计并完善了工作机制。

其中，PikeClient 作为 Pike 系统对接业务方的门户，在整个 Pike 2.0 系统中起着至关重要的作用，本文将以 PikeClient 为切入点介绍 Pike 的工作机制。

3.1 PikeClient 生命周期

为了更好地维护 Pike 2.0 内部状态，PikeClient 使用状态机来负责生命周期管理。

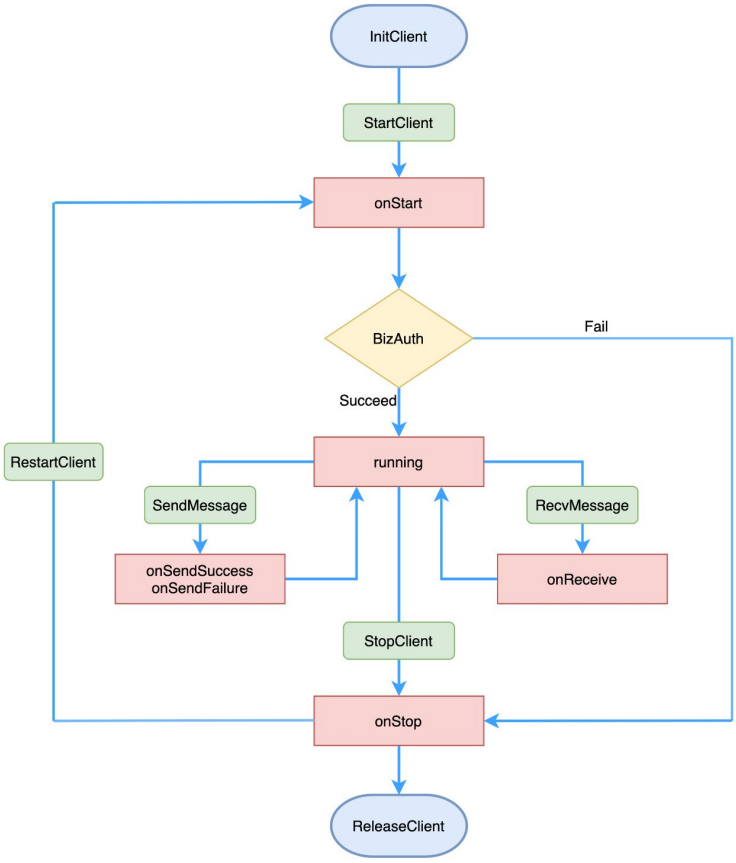


图 6　PikeClient 生命周期图

如图所示，PikeClient 生命周期主要包括如下几个部分：

- onStart：该状态是业务方调用 StartClient 或者 RestartClient 之后进入的状态，此时 PikeClient 已经正常启动，之后 Pike 2.0 内部会发起业务鉴权并根据鉴权结果流转到其他的状态，如图所示如果业务鉴权失败则进入 onStop 状态，如果业务鉴权成功则进入 running 状态。
- onStop：该状态是业务方调用 StopClient 或者业务鉴权失败之后进入的状态，此时 PikeClient 已经停止工作，客户端进入该状态之后需要 Restart 才能重新使用。
- running：该状态是 PikeClient 长稳工作的状态，此时 Pike 2.0 等待响应服务

推送的下行消息或者随时准备发送上行消息。作为双向消息通道，Pike 2.0 处理上下行消息的能力是完全并行的。

- onReceive: 该状态是 PikeClient 成功接收到下行消息之后进入的状态，Pike 2.0 将接收到的消息投递给业务方之后重新进入 running 状态等待下一次操作。
- onSendSuccess/onSendFailure：该状态是 PikeClient 发送上行消息之后进入的状态，业务方可以通过监听该状态来获取本次消息发送的结果。

通过基于状态机的生命周期管理，既严格定义了 PikeClient 的工作流程，也可以准确监控其内部状态，提高了 PikeClient 的可维护性。

3.2 PikeClient 工作模式

针对 Pike 1.0 混乱的账号体系痛点，Pike 2.0 设计了全新的工作模式。如下图所示，Pike 通过通道代理模块提供共享通道和独立通道两种模式来满足不通业务场景的需求。

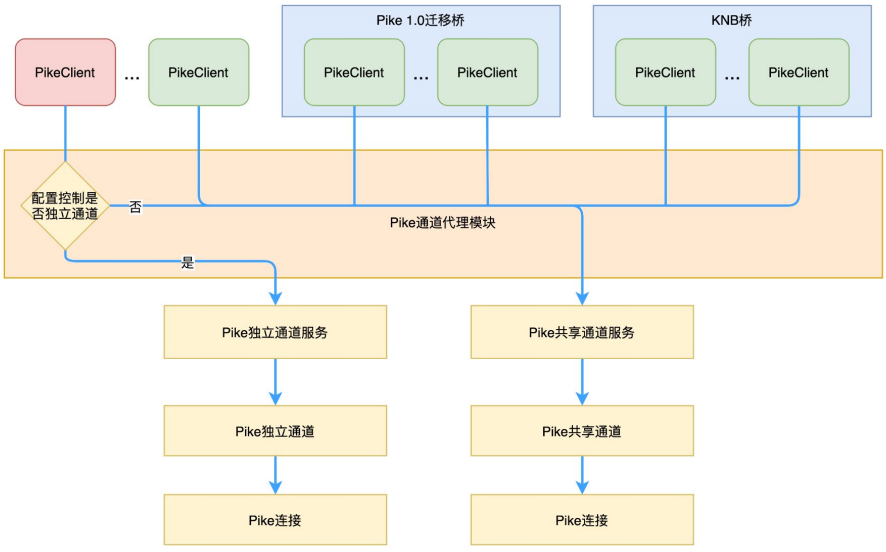


图 7　PikeClient 工作模式示意图

3.2.1 共享通道模式

共享通道模式是 Pike 2.0 基本的工作模式，新增的业务方在默认情况下都会使用该模式接入 Pike 2.0。

在 Pike 2.0 中 PikeClient 与 Pike 通道服务是多对一的共享关系，每个业务方都会有自己的 PikeClient，每个 PikeClient 都可以自定义消息推送标识 ID 而避免使用全局标识 ID。业务后端可以精简推送标识逻辑，避免同时维护多套账号体系。

不同业务的 PikeClient 仅在接入层面做了业务隔离，在 Pike 2.0 通道中会由 Pike 通道服务完成统一的管理。这种多对一的共享关系使得所有 Pike 业务共享 Pike 2.0 通道特性，同时又可以针对每个业务的使用场景设置其特定的消息处理能力，每个接入 Pike 2.0 的业务方都只需要关注其自己的 PikeClient 即可。

3.2.2 独立通道模式

独立通道模式是共享通道模式的拓展能力，Pike 2.0 通过配置控制来决策是否切换至该模式。

Pike 2.0 默认情况下所有业务方都是共享同一个 Pike 通道服务，然而鉴于业务场景的不同，每个业务对于消息吞吐量，消息时延等 SLA 指标的诉求也有差异，例如游戏业务对于消息时延过长的容忍性就比较差。针对特殊业务 Pike 2.0 提供了独立通道切换的能力支持。

所有 PikeClient 都通过 Pike 通道代理模块来对接 Pike 通道服务，Pike 通道代理模块可以通过开关配置来控制 PikeClient 与特定的 Pike 通道服务协同工作。通过运用代理模式，既保证了原有结构的完整性，在不需要调整 Pike 通道代码逻辑的基础上就能够完成独立通道能力支持；又可以扩展通道切换能力，有效地管理通道切换的流程，让 Pike 2.0 通道最大化提供业务能力的同时避免资源浪费。

3.3 PikeClient 保活机制

PikeClient 的保活完全依赖 Pike 2.0 通道的保活，针对 Pike 1.0 推送连接不稳定的痛点，Pike 2.0 通道在吸收 Pike 1.0 在保活机制方面沉淀的技术的基础上继续优化，

最后设计出基于心跳探测、重连机制和通道巡检的三重保活机制。保活机制如下图：

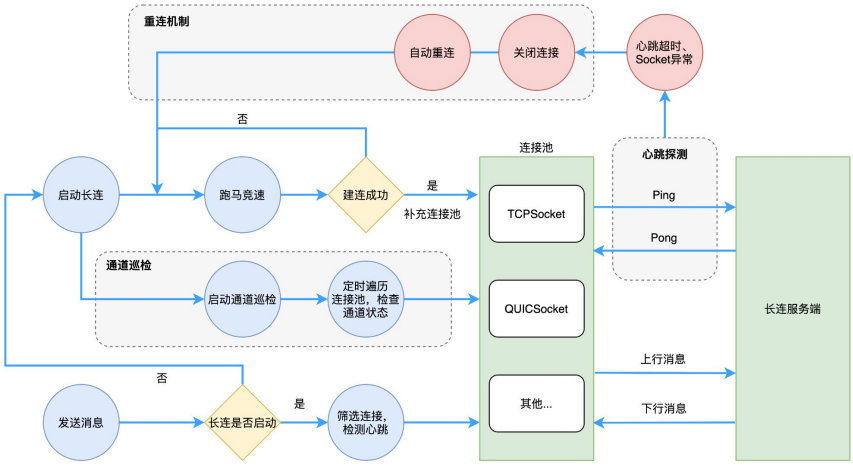


图 8　长连通道保活机制示意图

3.3.1 心跳探测

心跳探测是一种检查网络连接状态的常见手段，Pike 长连接是 TCP 连接，TCP 是虚拟连接，如果实际物理链路中出现诸如异常网络节点等因素导致连接出现异常，客户端和服务端并不能及时感应到连接异常，这时就会出现连接的状态处于 ESTABLISHED 状态，但连接可能已死的现象，心跳探测就是为了解决这种网络异常的技术方案。

客户端在心跳巡检计时器设置的心跳周期到达时判断是否存在上次心跳超时的异常，如果心跳超时则认为该连接已经不可用了，则会从连接池移除该连接并触发下文的重连机制。为了更快地发现通道异常，Pike 2.0 对于心跳周期与心跳超时都是可配置的，针对不同 App 使用的场景可以灵活地设置；而且在每次发送上行数据的时候都会及时检测上次心跳是否超时，使得心跳探测结果不必等到下次心跳周期到达的时刻才知悉。

Pike 2.0 并不是采用固定心跳频率来发送心跳包，Pike 2.0 会利用通道的上下行数据包来动态减少心跳包的发送次数。此外，智能心跳也是 Pike 2.0 持续关注的话题。

3.3.2 重连机制

重连机制是 Pike 2.0 作为长连接通道最核心的特性，也是 Pike 2.0 连接稳定性建设最重要的一环。

客户端会在发送消息、接收消息和心跳探测三个环节来决策是否需要触发重，一方面，如果主动发现连接池中可用连接不足则自动启动重连机制；另一方面，当现有可用连接关闭时也会自动触发重连机制。

Pike 2.0 在重连的过程中会采用斐波那契数列退避算法来发起建连请求直至建连成功，一方面，Pike 2.0 保证只要在网络可用的情况下总能够维持可用的长连接来服务于业务消息；另一方面，Pike 2.0 在网络持续不可用的情况下避免连续建连使得系统满载。

3.3.3 通道巡检

通道巡检是在心跳探测和重连机制的基础上进一步提升 Pike 2.0 稳定性的有效机制。

客户端会根据心跳周期设置一个全局的巡检定时器，在每次定时器设置的时刻到达时，客户端会触发通道异常检测逻辑，一旦发现异常都会尝试重启通道。

Pike 2.0 首先会在触发通道异常检测的时候获取当前通道状态，如果通道当前没有主动关闭但是通道处于不可用的状态，Pike 2.0 会强制执行一次自启动；此外，在通道巡检的过程中，巡检管理器会不断收集消息收发过程中出现的超时异常，当超时异常次数连续累计超过配置的最大阈值时，Pike 2.0 会认为当前通道可用性较低，需要强制关闭并执行一次自启动。

4. Pike 2.0 新增特性

Pike 2.0 作为 Pike 1.0 的升级产品，不只是为了解决 Pike 1.0 的痛点，通过新增特性以开拓新的应用场景也是 Pike 2.0 时刻关注的点。

4.1 聚合消息

随着公司内直播业务的兴起，公司内部也有很多业务方使用 Pike 1.0 作为弹幕、评论、直播间控制信令等下行实时消息的传输通道。但 Pike 1.0 基于早先的设计架构为弹幕、评论这种短时间需要处理海量消息的场景提供可靠服务的能力渐渐力不从心，主要表现在 QPS 大幅增长时，消息投递成功率降低、延时增加和系统性能开销增长等方面。Pike 通过引入聚合消息为直播场景中消息的投递提出更加通用的解决方案。

4.1.1 设计思想

直播场景中涉及的消息主要具备以下特点：

- 弹幕作为一种实时互动的载体，短时间内需处理大量的图片、文本等信息，如果不做聚合会浪费大量的带宽。
- 直播间相比普通推送场景，由于用户已经进入直播间，用户行为也相对统一可控，所以更需要一种群组消息来统一处理。
- 直播间对于不同类型的消息处理逻辑可以区分优先级，比如抽奖、控制信令是要求可靠性不能丢弃，而对于弹幕则可根据直播间热度、服务承受能力适当丢弃。

聚合消息在设计上主要采用下述思想：

- 从时间维度对消息进行聚合，减少不必要的带宽消耗。
- 采用消息分级策略，根据消息的类型设定不同的优先级，保证重要消息的可靠性。
- 抽象出类似直播间的聚合单元，统一管理加入聚合单元的用户行为。
- 采用客户端主动拉取的策略。相比传统的服务端推送策略，主动拉取是利用客户端天然分布式的特点将用户状态保存在客户端，服务端通过减少状态维护进而可以留出更多的资源用于业务处理。
- 提供上行消息能力，提供更完整的消息流通路径。

4.1.2 方案流程

Pike 2.0 针对每个聚合单元都使用环形队列来维护消息列表，发送到该聚合单元的

消息在经过优先级过滤之后都会插入队列 tail 指针标示的位置，随着该聚合单元内消息不断增加最后达到最大队列长度时，head 指针会不断移动来给 tail 指针腾出位置。聚合单元通过控制最大长度的环形队列来避免消息短时间井喷式增长带来的服务性能问题。

客户端在主动拉取的时候都会携带上一次获取到的消息处在环形队列中的偏移量，这样服务就会将偏移量标示的位置到 tail 指针标示的位置之间的消息进行聚合作为本次拉取的结果一次性返回给客户端。不同客户端各自维护自己的偏移量，以此来避免服务端对于客户端的状态维护。

客户端与服务端的具体交互如图所示，客户端在加入聚合单元之后主动拉取，如果本次拉取携带的偏移量能够从服务的环形队列中获取到聚合消息，那么就将消息回调给业务之后马上进行下一次拉取操作。如果本次携带的偏移量已经位于环形队列 tail 指针的位置，那么服务端将不做任何响应，客户端等待本次拉取超时之后开始下一次拉取操作，重复该流程直至客户端离开该聚合单元。与此同时，业务服务端如果有消息需要推送，则通过 RPC 的方式发送给 Pike 服务端，消息处理模块将执行消息分级策略过滤之后的有效消息插入环形队列。

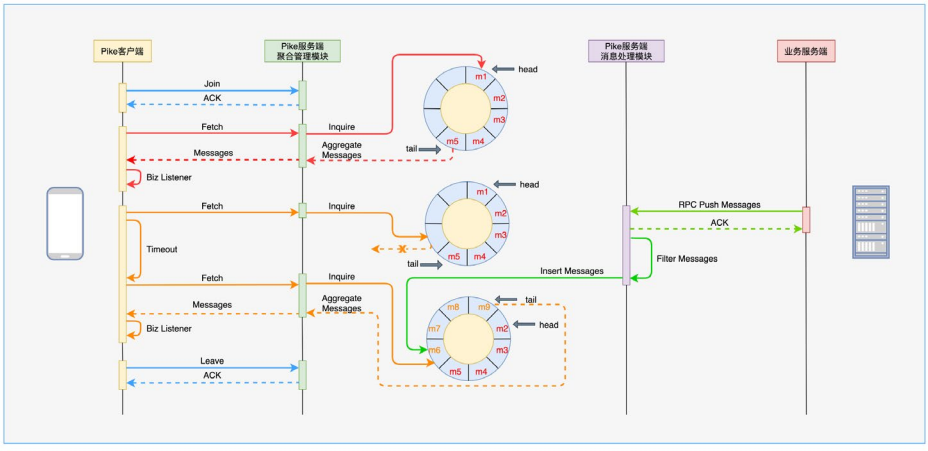


图 9　聚合消息交互流程图

4.2 消息保序

Pike 1.0 在设计之初就只适用于消息推送的场景，而 Pike 2.0 在其基础上演进为双向消息投递服务，即不仅支持下行的消息推送，还支持上行的消息投递。Pike 2.0 在上行的消息投递方面进一步拓展了消息保序的功能，这里的消息保序主要包含两个层面的含义，首先每一个业务客户端发送的消息都最大程度地到达同一个业务服务器，其次这些消息是按照客户端发送的时序一致地到达该业务服务器。

4.2.1 粘性会话

为了使每一个业务客户端发送的消息都最大程度地到达同一个业务服务器，Pike 2.0 引入了粘性会话的概念。粘性会话指的是同一客户端连接上的消息固定转发至某一特定的业务方机器处理，客户端断连重连后，保持新连接上的消息仍转发至该业务机器。

粘性会话可以归纳为如下的流程。首次业务登录的时候 Pike 2.0 服务器会利用负载均衡算法选择一台业务服务器，并将该业务服务器的路由标识通过业务登录结果通知客户端并保存，之后如果通道状态稳定的话所有的上行消息就都会投递到该业务服务器。如果期间通道状态波动出现断连的情况，Pike 2.0 在发起重连之后会重新进行业务登录，这一次业务登录会将之前保存的路由标识重新上报给 Pike 2.0 服务器，这样 Pike 2.0 服务器就会通过路由标识重新绑定该业务服务器。当然，如果路由标识指示的业务服务器已经停止提供服务，那么 Pike 2.0 服务器会重新通过负载均衡算法选择新的一台业务服务器，同时客户端会获取到新的路由标识，之后的逻辑重复该过程直至 Pike 2.0 客户端退出。

4.2.2 时序一致性

我们都知道 TCP 是有序的，那么在同一个 TCP 连接的前提下什么情况会出现客户端发送的消息乱序到达业务服务器呢？原因就是 Pike 2.0 服务器从 TCP 中读出消息之后将其投递给业务服务器是通过 RPC 异步调用的。

为了解决这种问题，最简单的方案当然是客户端将消息队列的发送窗口限定为 1，每

一条发送消息都在 Pike 2.0 服务器投递给业务服务器之后才能收到 ACK，这时再发送下一条消息。但是考虑到网络传输在链路上的时延远远大于端上处理的时延，所以该方案的 QPS 被网络传输设了瓶颈，假设一个 RTT 是 200ms，那么该方案理论也只能达到 5 的 QPS。

Pike 2.0 为了提高上行消息保序投递的 QPS，采用服务端设置消息队列缓存的方案。如图所示，客户端可以在发送窗口允许的范围内一次性将多条消息发送出去，服务端把收到的消息都按顺序缓存在消息队列中，然后串行的通过 RPC 调用将这些缓存的消息依序投递给业务服务器。这种保序方案将 QPS 性能的瓶颈点从之前网络传输在链路上的时延转移到了 RPC 调用的时延上，而实际场景中一次 RPC 调用往往在几个毫秒之间，远远小于网络传输在链路上的时延，继而显著地提升了 QPS。

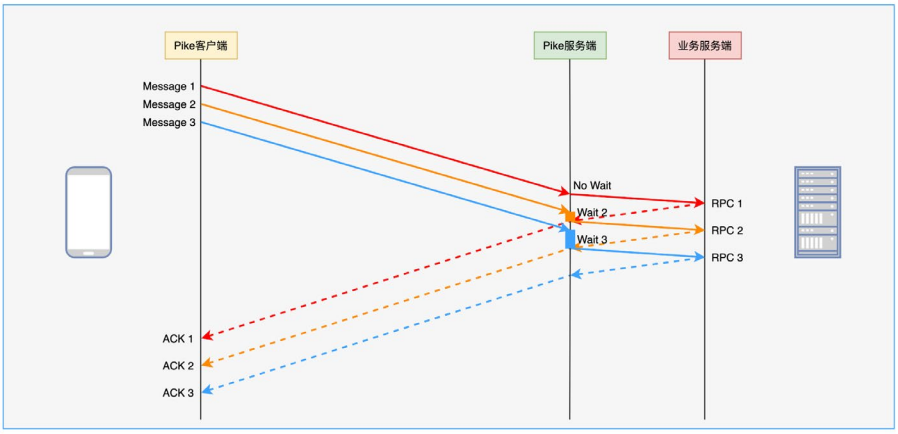


图 10　时序一致性示意图

5. Pike 2.0 稳定性保障

5.1 Pike 2.0 监控体系

Pike 2.0 依赖美团监控平台 Raptor 完成监控体系建设，服务端和客户端都建设了各自完善的指标监控。Pike 2.0 客户端通过利用 Raptor 的端到端指标能力和自定义指标能力输出了超过 10+ 个监控指标来实时监控 Pike 系统，这些指标覆盖通道建立、

消息投递、业务登录、系统异常等多维度。在实时指标监控的基础上 Pike 2.0 针对不同指标配置了报警阈值，以推送消息为例，如果特定 App 的大盘数据在每分钟的上下波动幅度超过 10%，那么 Raptor 系统就会向 Pike 项目组成员推送告警信息。

基于所有 Raptor 监控指标，Pike 2.0 提炼核心 SLA 指标如下：

指标名称	指标定义	指标意义
上行消息投递成功率	上行消息从发送到收到 ACK 的成功率	代表 Pike 2.0 业务上行消息投递能力
上行消息投递延时	上行消息投递 RTT	同上
下行消息投递成功率	下行消息从发送到收到 ACK 的成功率	代表 Pike 2.0 业务下行消息投递能力
下行消息投递延时	下行消息投递 RTT	同上
通道可用耗时	通道从建立到可以传递消息的时间	代表 Pike 2.0 通道建连能力
日均消息量	日均投递的上下行消息总量	代表 Pike 2.0 产品服务能力

Pike 2.0 会定期输出基于核心 SLA 指标的大盘数据报表，同时可以基于 App、业务类型、网络类型等多维度对数据进行筛选以满足不同用户对于指标数据的需求。

5.2 Pike 2.0 个案用户追踪

监控体系能从全局的角度反映 Pike 2.0 系统稳定性，针对个案用户，Pike 管理平台提供完整的链路追踪信息。

每个 Pike 2.0 连接都由唯一标识 Token 来区分，通过该唯一标识 Token 在 Pike 管理平台的“连接嗅探”模块主动探测便能获得对应连接上所有信令的交互流程。如图所示，流程中明确标注了客户端建立连接、发起鉴权、绑定别名等信令，点击对应信令可以跳转信令详情进一步查看该信令所携带的信息，再结合 SDK 埋点在美团日志服务 Logan 的离线日志就可以快速发现并定位问题。

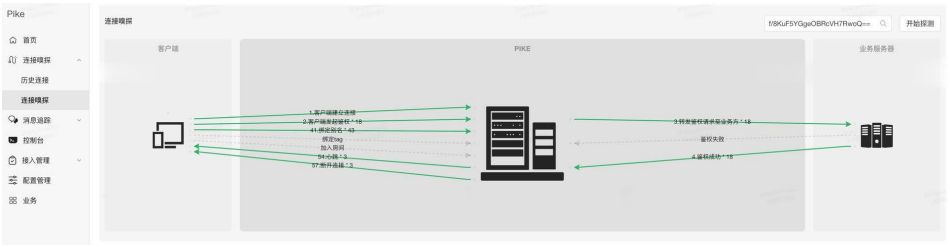


图 11　个案用户追踪信令交互图

6. Pike 2.0 建设成果

截至 2021 年 6 月，Pike 共接入业务 200+ 个，日均消息总量约 50 亿 +，Pike 2.0 消息到达率 >99.5%（相比 Pike 1.0 提升 0.4% ），Pike 2.0 平均端到端延时 <220ms（相比 Pike 1.0 减少约 37%）。

部分应用案例：

- 直播场景消息服务方案。支持直播业务的直播互动功能，具备了支持同时在线百万级别大型直播的能力。
- 消息推送、Feed 流预加载等实时触达方案。支持营销类、控制类等业务消息实时推送，业务消息到达率最高提升 10%，长连通道建联耗时减少 5%。
- IoT 设备接入方案。支持取餐柜业务 IoT 接入能力，帮助业务消息到达率从 98.4% 提升到 99.6%。
- 小游戏场景消息投递方案。支持美团小游戏场景通信能力，消息到达率达到 99.8% 以上，上行延时低至 195ms。

7. 总结与未来展望

Pike 在美团应用广泛，目前主要集中在实时触达、互动直播、移动同步等业务场景，随着公司业务的快速发展，Pike 对可用性、易用性、可扩展性提出了更高要求，希望提升各种业务场景下的网络体验，因此 Pike 未来的规划重点主要是：提供多端、多场景下的网络通信方案，不断完善协议生态，在各种应用场景下对抗复杂网络。

- **拓展 Pike 通用基础能力，提升通道性能**。通过优化保序方案，提供专用通道，

优化传输协议等方式，进一步提升吞吐量和稳定性，降低推送时延。

- **建设 Pike IoT，提供 IoT 接入层方案**。为公司内物联网应用场景（单车、充电宝、取餐柜、智能头盔、仓库、门店设备等）提供统一的 IoT 接入层解决方案，支持多种接入协议（HTTP、MQTT、CoAP 等），为业务提供安全可靠的设备连接通信能力。
- **优化弱网环境通信体验**。在移动端和 IoT 端基于美团自研 MQUIC 网络协议库，探索 Pike over QUIC，在桌面端探索 WebTransport 技术，通过全面支持 QUIC 协议，提升弱网大包场景下的网络性能，降低长尾分布的请求耗时。

作者简介

健午、佳猛、陆凯、冯江等，均来自美团基础技术部 – 前端技术中心。

招聘信息

美团基础技术部 – 前端技术中心诚招高级、资深技术专家，Base 上海、北京。我们致力于为美团海量业务建设高性能、高可用、高体验的前端基础技术服务，涵盖网络通信、资源托管、设计体系、研发协同、低代码、性能监控、运维保障、Node 基建、IoT 基础设施等技术领域，欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

新一代 CTR 预测服务的 GPU 优化实践

作者：伟龙　小卓　文魁　駃飞　小新

1. 背景

CTR（Click-Through-Rate）即点击通过率，是指网络广告的点击到达率，即该广告的实际点击次数除以广告的展现量。为 CTR 指标服务的打分模型，一般称为 CTR 模型。我们可以将此概念进一步扩展到互联网应用中各种预估转化率的模型。CTR 模型在推荐、搜索、广告等场景被广泛应用。相对于 CV（计算机视觉）、NLP（自然语音处理）场景的模型，CTR 模型的历史结构比较简单，计算量较小。美团的 CTR 模型一直沿用 CPU 推理的方式。随着近几年深度神经网络的引入，CTR 模型结构逐渐趋于复杂，计算量也越来越大，CPU 开始不能满足模型对于算力的需求。

而 GPU 拥有几千个计算核心，可以在单机内提供密集的并行计算能力，在 CV、NLP 等领域展示了强大的能力。通过 CUDA[1] 及相关 API，英伟达建立了完整的 GPU 生态。基于此，美团基础研发平台通过一套方案将 CTR 模型部署到 GPU 上。单从模型预测阶段看，我们提供的基于英伟达 T4 的 GPU 深度优化方案，在相同成本约束下，对比 CPU，提升了 10 倍的吞吐能力。同时，在典型的搜索精排场景中，从端到端的维度来看，整体吞吐能力提升了一倍以上。

除了提高吞吐、降低成本外，GPU 方案还为 CTR 模型的应用带来了额外的可能。例如，在某搜索框自动补全的场景，由于天然的交互属性，时延要求非常苛刻，一般来说无法使用复杂的模型。而在 GPU 能力的加持下，某复杂模型的平均响应时间从 15 毫秒降低至 6~7 毫秒，已经达到了上线要求。

接下来，本文将与大家探讨美团机器学习平台提供的新一代 CTR 预测服务的 GPU 优化思路、效果、优势与不足，希望对从事相关工作的同学有所帮助或者启发。

2. CTR 模型 GPU 推理的挑战

2.1 应用层的挑战

- CTR 模型结构多变，包含大量业务相关的结构，同时新的 SOTA 模型也层出不穷，硬件供应商由于人力受限，会重点优化常用的经典结构，如 ResNet。对于没有收敛的结构，官方没有端到端的优化工具可以支持。
- CTR 模型中通常包含较大的 Embedding 表结构，要考虑到 Embedding 表存在显存放不下的情况。
- 在典型的推荐场景中，为了达到更快的 POI 曝光的目的，模型的时效性要求很高，在线模型服务需要提供增量更新模型的能力。

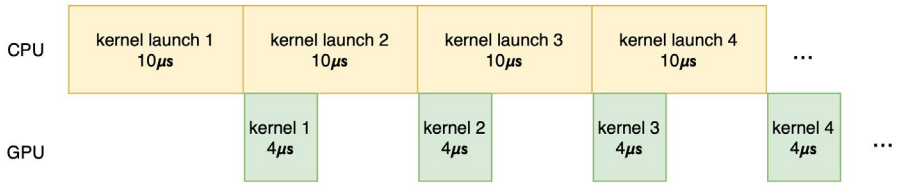
2.2 框架层的挑战

- **算子层面**：目前主流的深度学习框架，如 TensorFlow 和 PyTorch，可以说是深度学习第二代框架，它们首先要解决第一代框架 Caffe 的问题，Caffe 有一个明显问题就是 Layer 的粒度过粗，导致那个时代的算法开发者都必须有“自己写自定义层”的能力。TensorFlow 和 PyTorch 都把模型表达能力放在较高的优先级，导致算子粒度比较小，无论是对 CPU 还是 GPU 架构，都会带来很大的额外开销。
- **框架层面**：TensorFlow 和 PyTorch 本质都是训练框架，对算法开发者比较友好，但非部署友好。其中隐含了很多为了方便分布式训练做的设计，比如 TensorFlow 为了方便将 Variable 拆到不同的 PS 上，内置了 Partitioned_Variable 的设计。在基于 GPU 单机预测的场景下，这些结构也会带来额外的开销。

2.3 硬件层的挑战

第一，TensorFlow 的算子粒度划分较细，导致一个模型通常由几千个算子构成，这些算子在 GPU 上的执行转变为对应的 GPU kernel 的执行。kernel 是 GPU 上并行执行的函数。

GPU kernel 大体上可以划分为传输数据、kernel 启动、kernel 计算等几个阶段，其中每个 kernel 的启动需要约 10μs 左右。大量的小算子导致每个 kernel 的执行时间很短，kernel 启动的耗时占了大部分。相邻的 kernel 之间需要通过读写显存进行数据的传输，产生大量的访存开销。而 GPU 的访存吞吐远远低于计算吞吐，导致性能低下，GPU 利用率并不高。



第二，GPU 卡上包含多个计算单元，理论上，不同计算单元是可以跑不同 kernel 的，但实际上为了编程简单，CUDA 默认假设在同一时刻一个 Stream 里跑同一个 kernel。虽然可以通过多 Stream 的方式跑，但是多 Steam 之间又缺少细粒度的协同机制。

在经过充分调研与讨论后，我们决定第一期重点关注 TensorFlow 框架下如何解决常见 CTR 模型结构在英伟达 GPU 上执行效率不高的问题，我们先将问题收敛为以下两个子问题：1. 算子粒度过细，GPU 执行效率低下。 2. 模型结构多变，手工优化投入大，通用性差。

3. 优化手段

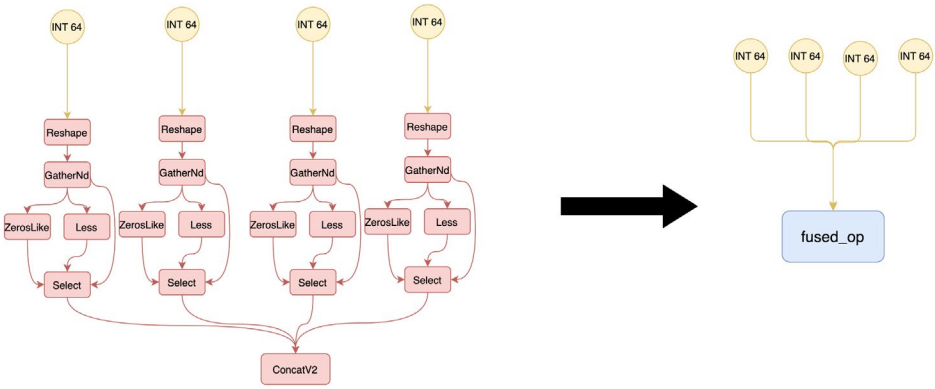
为了解决上面的问题，我们对业界深度学习加速器进行了一些调研。业界比较成熟的推理优化方案主要是 TensorRT/XLA/TVM。TensorRT 采用手工优化，对一些定制的模型结构进行算子融合，并对计算密集型算子（如卷积）进行了高效调优。XLA 是 TensorFlow 内置的编译优化工具，主要针对访存密集型结构，通过编译手段，实现算子的融合。TVM[2] 具备较全面的优化能力，使用编译手段进行算子的融合，同时可以通过机器学习的方式实现计算密集型算子的自动调优。

经过广泛的调研和对比，我们最终选择了 TVM 作为优化工具。TVM 通过编译手段，

可以较好地应对多变的模型结构，解决了手工优化通用性差的问题。但 TVM 应用在业务模型也存在一系列问题：支持的算子数较少，而且目前对动态 Shape 的支持还不够好。针对这两个问题，我们将 TVM 和 TensorFlow 结合起来，结合 CTR 模型的结构特点与 GPU 的硬件特性，开发一系列流程，实现了对 CTR 模型的优化。

3.1 算子融合

通过将多个小算子融合为一个语义等价的大算子，可以有效减少 GPU 上的 kernel 数量。一方面，kernel 数量减少直接降低了 kernel 发射的开销；另一方面，融合后的大 kernel 执行的计算量增加，避免了多个 kernel 间数据传输导致的频繁访存，提高了计算的访存比。



可以看到，上图中的左右等价结构，左侧的 21 个算子执行的运算，可以在 1 个等价算子中完成。反映到 GPU 的活动上，左侧至少有 21 个 GPU kernel 以及 21 次显存的读写，而右侧只需要执行 1 个 kernel 以及 1 次显存读写。对于每个融合后的算子，需要有对应的 kernel 实现。然而，模型的算子组合是无穷的，对每种融合后算子手工实现 kernel 是不现实的。TVM 通过编译手段，可以自动进行算子的融合以及设备代码生成，避免了逐一手写 kernel 的负担。

3.1.1 TF-TVM 自动切图优化

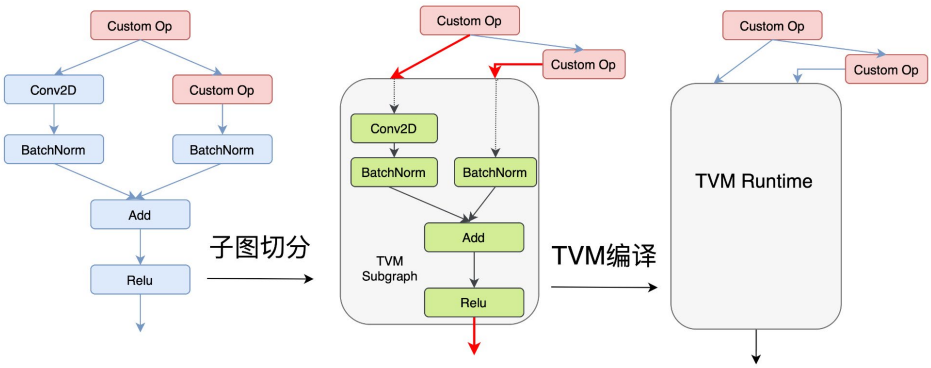
TensorFlow 模型中，如果包含 TVM 不支持的算子，会导致无法执行 TVM 转换。

我们的思路是将可以用 TVM 优化的部分切出来，转为 TVM 的 engine，其他部分依然使用 TensorFlow 的算子。在 XLA 和 TRT 转换的时候也有类似问题，我们分析了 TF-XLA 和 TF-TRT 二者的实现：

- TF-XLA 的实现方案，在 Grappler[4] 优化图之后，有一个 POST_REWRITE_FOR_EXEC（通过这个关键字可以在源码中搜索到）阶段，在这个阶段，会执行三个针对 Graph 的 Pass，分别是用来标记算子，封装子图，改写子图并构建 LaunchOp。
- TF-TRT 的实现方案，TF-TRT 在 Grappler 中注册了一个优化器，在这个优化器中，找到连通子图，并将其替换为 TRT Engine。

在最终方案实现上，我们参考了 TF-TRT 的设计。这个设计对比 XLA 的优势在于 XLA 切图方案与 TensorFlow 源码紧耦合，直接将 XLA 的三个 Pass 嵌入到了启动 Session 的主流程中。而切图策略，优化策略后续会有非常频繁的迭代，我们不希望与 TensorFlow 的源码太过耦合。我们扩展了 TF-TVM 的方案，在实际使用中我们把这个切图过程为一个独立流程。在模型部署或更新时，自动触发。

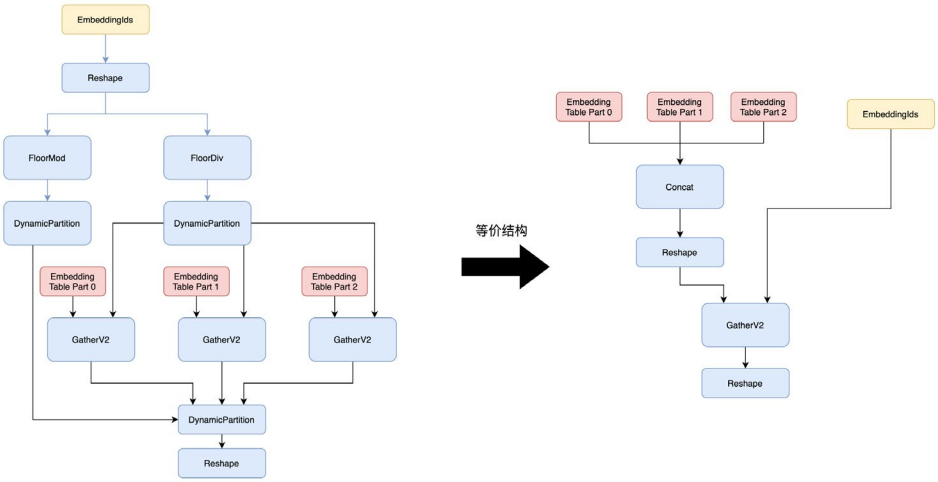
在推理阶段，优化过的子图使用 TVM 执行，其余的计算图使用 TensorFlow 原生实现执行，将两者结合共同完成模型的推理。由于 TVM 和 TensorFlow 的 Runtime 各自使用独立的内存管理，数据在不同框架间传输会导致额外的性能开销。为了降低这部分开销，我们打通了两个框架的底层数据结构，尽可能避免额外的数据拷贝。



3.1.2 计算图等价替换

TensorFlow 模型中过多的不被 TVM 支持的算子会导致 TF-TVM 切图零碎，影响最终的优化效果。为了让 TF-TVM 切图尽量大且完整，以及让 TVM 优化过程中的融合力度更大，我们对模型中的一些复杂结构进行检测，替换为执行更高效或更易于融合的等价结构。

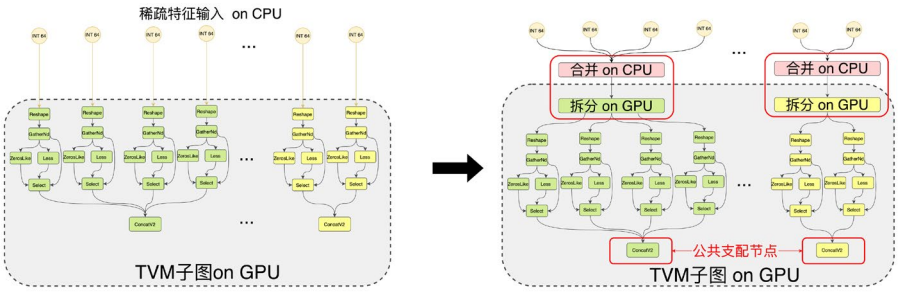
例如，TensorFlow 原生 EmbeddingLookup 结构，为了支持分布式训练，会对 Embedding 表进行切分，产生 DynamicPartition 和 ParallelDynamicStitch 等动态算子。这些动态算子不被 TVM 支持，导致 TF-TVM 图切分过于细碎。为了让 TF-TVM 切图更完整，我们通过图替换，对这种结构进行修改，通过将 Embedding 分表提前合并，得到简化的 EmbeddingLookup 结构。



3.2 CPU-GPU 数据传输优化

TVM 优化后的子图被替换为一个节点，该节点在 GPU 上执行，通常有几十甚至几百个输入，该节点的前置输入（如 Placeholder）通常是在 CPU 上执行，会涉及多次的 CPU-GPU 传输。频繁的小数据量传输，无法充分利用带宽。为了解决这个问题，我们对模型结构进行修改，在计算图中添加合并与拆分节点，控制切图的位置，减少数据传输的次数。

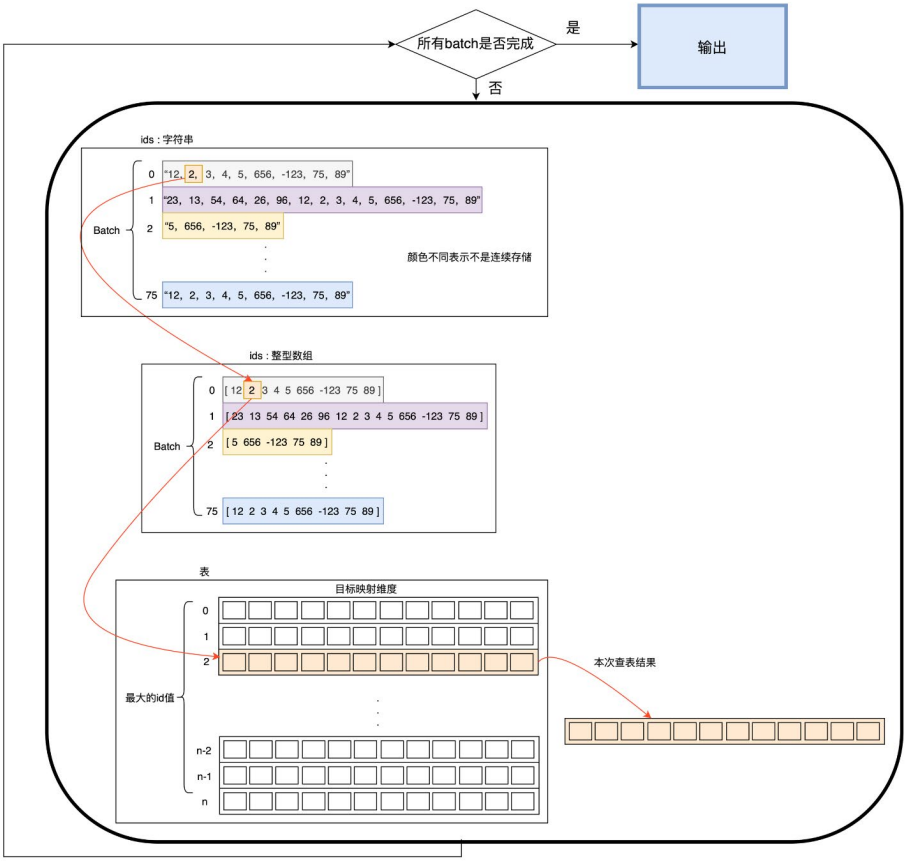
一种可能的合并方式是，对这些输入按相同的 Shape 和 Dtype 进行合并，后续进行拆分，将拆分节点切入 TVM 的子图一起优化。这种方式会导致一些问题，如部分子图的算子融合效果不佳；另一方面，GPU kernel 函数的参数传递内存限制在 4KB，对于 TVM 节点输入非常多的情况（如超过 512 个），会遇到生成代码不合法的情况。



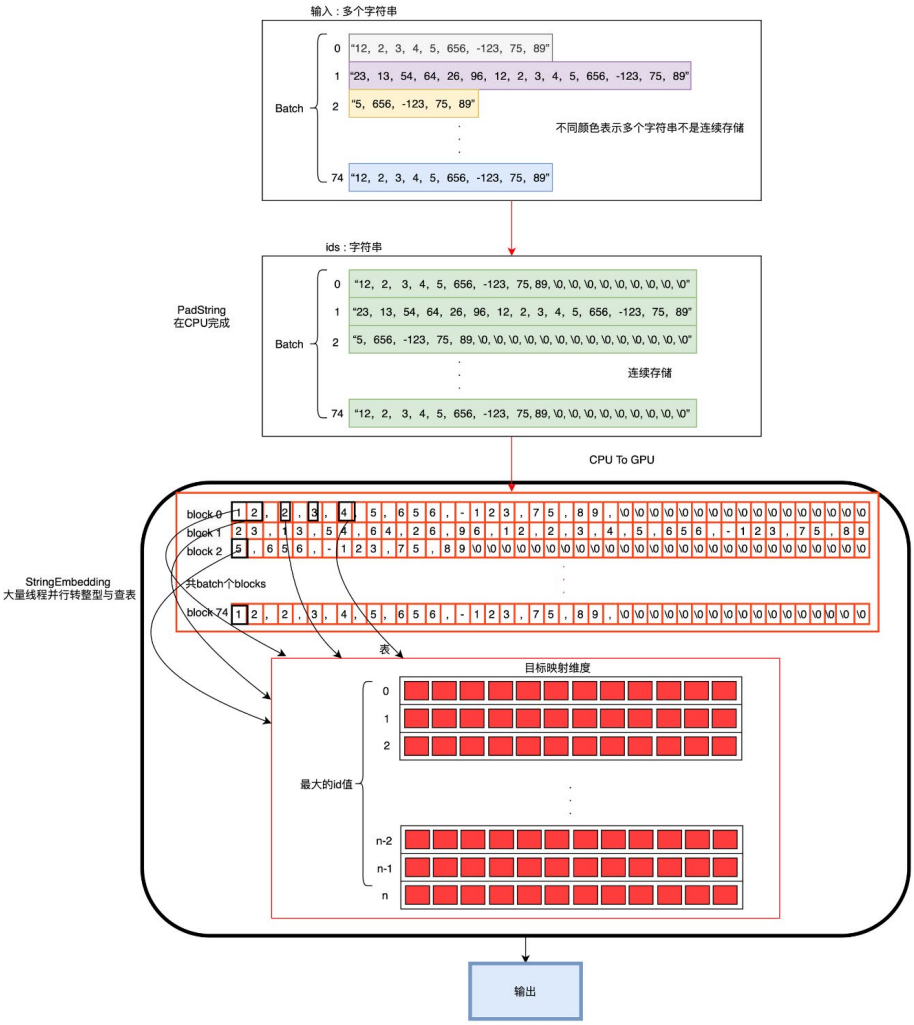
3.3 高频子图手工优化

对于 TVM 无法支持的子图，我们对业务中高频使用的结构进行抽象，采用手写自定义算子的方式，进行了高效 GPU 实现。

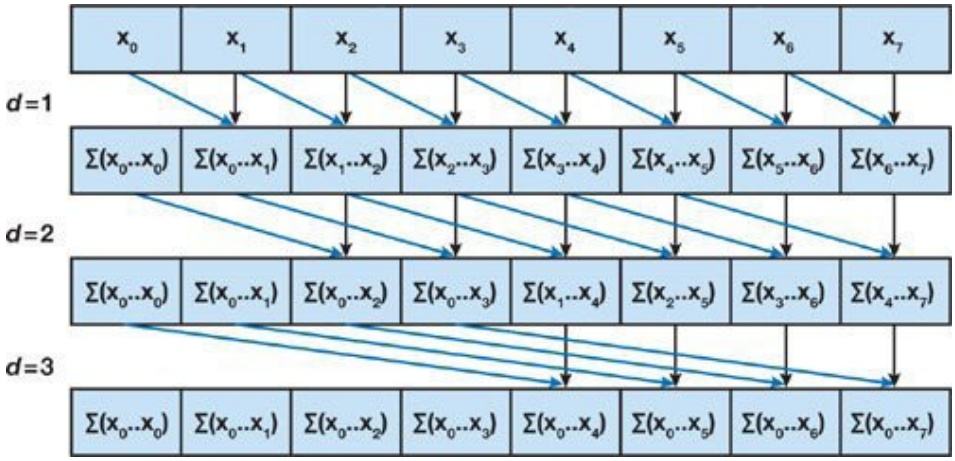
例如，模型中有部分时序特征使用 String 类型输入，将输入的字符串转为补齐的数字 Tensor，将 int 类型的 Tensor 作为下标进行 Embedding 操作。这部分子图的语义如图，以下简称 SE 结构（StringEmbedding）：



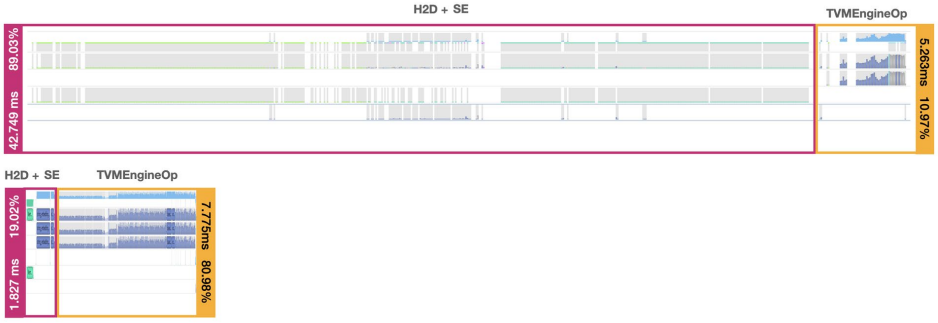
这一部分结构，TensorFlow 的原生实现只有基于 CPU 的版本，在数据量较大且并行度较高的情景下，性能下降严重，成为整个模型的瓶颈。为了优化这部分结构的性能，我们在 GPU 上实现了高效的等价操作。



如图所示，PadString 算子在 CPU 端将多个字符串按最大长度进行补齐，拼接成一个内存连续的 uint8 类型 Tensor，以便一次性传输到 GPU。StringEmbedding 接收到补齐后的字符串后，利用 GPU 并行计算的特性，协同大量线程完成字符串的切分与查表操作。在涉及规约求和、求前缀和等关键过程中，使用了 GPU 上的 Reduce/Scan 算法，编码过程使用 `warp_shuffle` 指令，不同线程通过寄存器交换数据，避免了频繁访存的开销，获得了很好的性能。

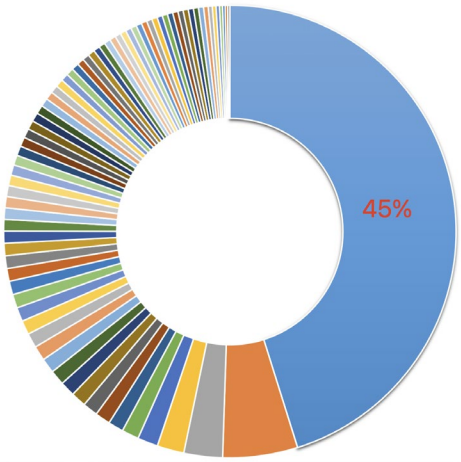


GPU Scan 算法示意，一个 8 个元素的前缀和操作，只需要 3 个迭代周期。在一个有几十路类似操作的模型中，手工优化前后的 GPU timeline 对比如下图，可以看到 H2D + StringEmbedding 这部分结构的耗时有很大的缩减，从 42 毫秒缩减到 1.83 毫秒。



除了 StringEmbedding 结构，我们对 StringSplit + ToNumber + SparseSegmentSqrt、多路并行 StringEmbedding 等结构都进行了高效融合实现，在优化流程中通过结构匹配进行相应的替换。

3.4 CPU-GPU 分流

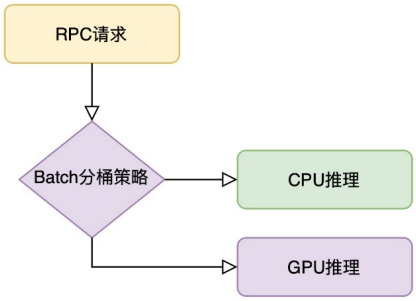


实际线上的 RPC 请求，每个请求内的样本数（下文称 Batch）是在 [1,MaxValue] 范围内变化的，MaxValue 受上游业务系统，其他基础系统能力等多方面因素制约，相对固定。如上图所示，以某个搜索服务为例，我们统计了线上的 Batch 数值分布，Batch=MaxValue 的请求占比约 45%，Batch=45 占比 7.4%，Batch=1 占比 2.3%。其余的 Batch 占比从 0.5% 到 1% 不等。对于 GPU 来说，提高单个请求的 Batch 能更好地利用硬件资源，发挥 GPU 的并行计算能力，表现出相对 CPU 更优的延迟和吞吐；当 Batch 较小时，GPU 相对 CPU 的优势就不明显了（下图是我们测试同样的模型在固定压力下，CPU/GPU 上延迟的变化）。

大部分请求都由 GPU 在做了，CPU 资源有较多空余，我们将一些小 Batch 的碎请求放在 CPU 运行，这样可以让整个 Worker 的资源利用更加均衡，提高系统整体的性能。我们根据测试设定了一个 Batch 阈值，以及计算图在异构硬件上区别执行的判断逻辑：对于小 Batch 的情况，直接在 CPU 上执行计算图，只有 Batch 超过阈值的请求才会在 GPU 上推理。从线上的统计数据来看，整体流量的 77% 跑在 GPU 上，23% 跑在 CPU 上。

在 GPU 的一系列优化策略和动作中，Batch 大小是很重要的信息，不同 Batch 下优化出的 kernel 实现可能是不同的，以达到对应 workload 下最优的计算性能；由于线

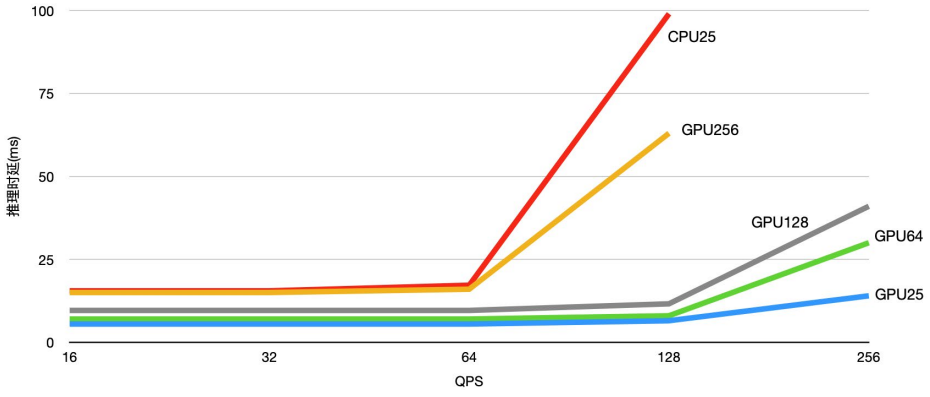
上的流量特点，发送到 GPU 的请求 Batch 分布比较细碎，如果我们针对每个 Batch 都优化一个模型的 kernel 实现显然是不够经济和通用的。因此，我们设计了一个 Batch 分桶策略，生成 N 个固定 Batch 的优化模型，在实际请求到来时找到 Batch 距离最近的一个 Bucket，将请求向上 Padding 到对应的 Batch 计算，从而提高了 GPU 的利用效率。



4. 压测性能分析

我们选取一个模型进行线上性能压测分析。

- CPU 模型测试环境为 16 核 Intel® Xeon® Gold 5218 CPU @ 2.30GHz，16G 内存。
- GPU 模型测试环境为 8 核 Intel® Xeon® Gold 5218 CPU @ 2.30GHz，Tesla T4 GPU，16G 内存。

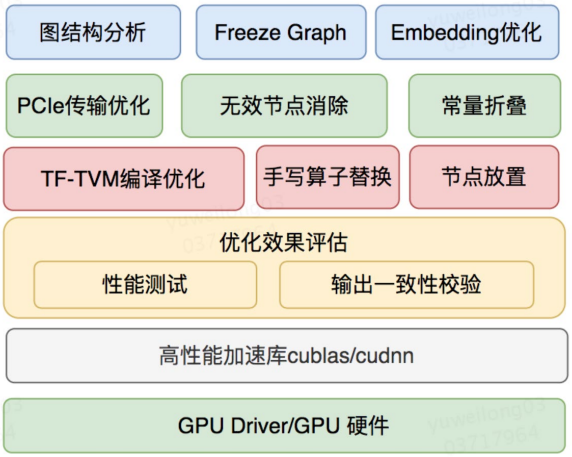


下图对比了在不同的 QPS 下（x 轴），GPU 模型在各 BatchSize 下的推理时延（y 轴）。GPU 模型在 BatchSize=128 以下，推理耗时差异不明显，较大的 BatchSize 更有利于吞吐；对比 BatchSize=256 的 GPU 模型与 BatchSize 为 25 的 CPU 模型，在 QPS 低于 64 的情况下，二者推理耗时基本持平；QPS 超过 64 的情况下，GPU 的推理时延低于 CPU。GPU 的吞吐相比 CPU 提升了 10 倍。

同时，我们可以看到不同曲线的陡峭程度，CPU 在 QPS 高出 64 后，时延会迅速上升，GPU 则依然保持平稳，直到 QPS 超过 128 才会有明显上升，但仍旧比 CPU 更平稳。

5. 整体架构

针对 CTR 模型的结构特点，我们抽象出了一套平台化的通用优化流程。通过对模型结构的分析，自动应用合适的优化策略，通过性能评估和一致性校验，保证模型的优化效果。



6. 不足之处与未来规划

在易用性层面，目前的方案形式是提供了一套在线优化脚本，用户提交模型后，自动优化部署。由于涉及对计算图结构的分析、编辑以及 TVM 的编译等过程，目前的模

型优化耗时较长，大部分模型优化耗时在 20 分钟左右。后续需要考虑加速 TVM 编译的效率。

在通用性层面，从我们的实际应用情况来看，TVM 编译优化和高性能手写算子是最主要的收益来源。手工优化很考验开发同学对业务模型的理解和 GPU 编程的能力。编写一个高性能的融合算子已经不太容易，要做到有一定的迁移能力和扩展性则更有难度。

总的来说，CTR 模型推理在 GPU 上未来需要考虑的问题还有很多。除了要基于业务理解提供更好的性能外，还要考虑模型规模巨大后无法完整放入显存的问题以及支持在线模型更新的问题。

作者简介

伟龙、小卓、文魁、驶飞、小新等，均来自美团基础研发平台 – 机器学习预测引擎组。

参考资料

[1] CUDA C++ Programming Guide
[2] TVM Documentation
[3] Accelerating Inference In TF-TRT User Guide
[4] TensorFlow graph optimization with Grappler

招聘信息

美团机器学习平台大量岗位持续招聘中，实习、社招均可，坐标北京 / 上海，欢迎感兴趣的同学加入我们，构建多领域公司级机器学习平台，帮大家吃得更好，生活更好。简历可投递至：wangxin66@meituan.com。

美团弹性伸缩系统的技术演进与落地实践

作者：tuyang

弹性伸缩具有应突发、省成本、自动化的业务价值。平台侧将各业务零散、闲置资源进行整合，形成一个大规模资源池，通过弹性调度、库存管控技术在公司运营成本和业务体感中寻求较好的平衡。

本文将介绍美团弹性伸缩系统落地过程中面临的技术挑战、推广以及在运营层面的一些思考。在美团这种多样化的业务场景中落地弹性伸缩，与业界公有云、自建私有云的公司相比，既有共性又有自己的特点，希望能为大家提供新的思路或者启发。

前言

稳定、高效、可靠的基础设施是互联网企业应对业务高峰流量的底层基石。作为美团统一的基础技术平台，基础技术部一直致力于通过业内前沿技术的落地，保障公司内部所有业务在线生产系统所依赖的基础技术平台能稳定、安全、低成本、可持续地运行与发展。

弹性伸缩系统是基于 Docker 开发的自动弹性伸缩平台，在美团经历了多年的发展。

早在 2016 年，美团就在线上环境中尝试使用容器环境，推出了基于 OpenStack 的**容器集群平台 Hulk 1.0**。随着容器的落地，弹性伸缩 1.0 版本应运而生，它解决了扩容实例慢、扩容上线慢、资源回收慢、计算资源冗余等问题。

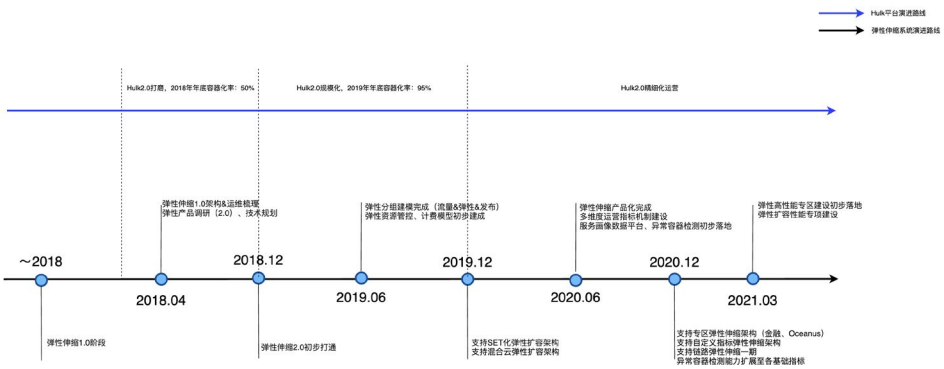
结合前两年的探索和实践以及业界相关领域技术的成熟度，2018 年至今，我们对容器集群平台进行了一次自底向上的的整体升级，也就是现在的 **Hulk 2.0 平台**。在底层，Hulk 2.0 建设了自研的操作系统、容器引擎，并将 OpenStack 替换成了容器编排的事实标准 Kubernetes；在上层弹性伸缩系统 PaaS 层面，推出了弹性伸缩系统 2.0，解决了弹性伸缩 1.0 实践过程中面临的很多业务痛点，包括：

- **扩出的业务代码版本不一致**：导致业务逻辑异常，造成资损。
- **部分服务高峰期资源不足**：导致业务无法有效承载流量，引起资损，
- **平台维护成本高**：北京、上海的业务各自一套弹性伸缩用户端管理平台、弹性逻辑（因为美团、大众点评合并前期，服务治理框架、CMDB 系统、发布系统尚未标准化）。
- **配置使用灵活性低**：业务集群每增 / 减一个 IDC 均需在平台做相匹配的配置操作，在资源的调控上无法和公司的流量调度体系包括单元化架构、同地域、同中心调用有效地进行匹配。

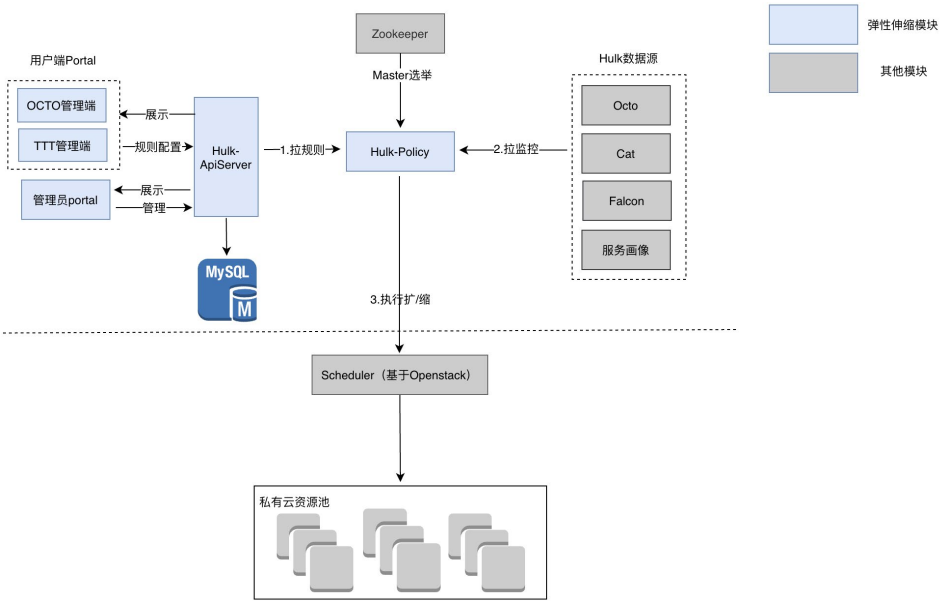
一、弹性伸缩系统演进

弹性伸缩 1.0 架构

我们首先看一下，产品演进路线和弹性伸缩 1.0 架构图。



产品演进路线



弹性伸缩 1.0 架构

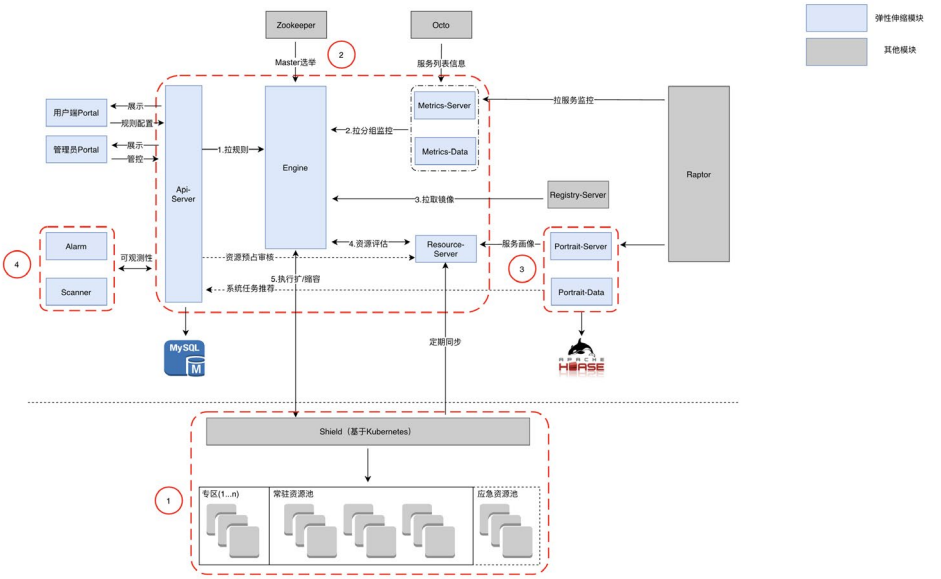
自左向右、自上而下进行模块介绍：

- **用户端 Portal**：OCTO 管理端，OCTO 是美团服务治理平台，出于北京业务操作服务节点习惯的考虑，在其上开发了对应的弹性伸缩管理页面；TTT 管理端，TTT 是上海侧（原大众点评侧）的 CMDB 管理平台，出于上海侧业务操作服务节点习惯的考虑，在其上开发了对应的弹性伸缩管理页面。
- **Hulk-ApiServer**：Hulk 1.0 容器平台的网关服务。
- **Hulk-Policy**：弹性伸缩系统 1.0 的业务逻辑模块，其中涵盖了具体的指标聚合、扩缩容逻辑、策略等，主从架构模式，使用 Zookeeper 进行 Master 选举，从是冷备。
- **Hulk 数据源**：OCTO，美团服务治理平台；CAT，美团服务端、移动端监控平台，主要负责应用层面；Falcon，基于开源 Falcon 定制化的系统监控平台，主要负责机器层面。
- **Scheduler**：Hulk 1.0 容器平台的调度系统，基于 OpenStack 做了二次开发。

弹性伸缩 2.0 架构

弹性伸缩系统 2.0 主要在以下四个方面进行了架构演进：

1）调度系统 – 替换：将 OpenStack 替换成了容器编排的事实标准 Kubernetes，并且在常驻资源池的基础上，新托管了专区资源池以及应急资源池。 2）**单体服务 – 微服务化**。 a. API-Server：弹性伸缩 – 网关服务。 b. Engine：弹性伸缩 – 引擎，负责各服务实例扩缩的发起、终止，主从架构模式，使用 Zookeeper 进行 Master 选举，从是热备。 c. Metrics-Server、Metrics-Data：【新增】弹性伸缩指标服务，负责 Raptor、OCTO 等数据源的采集 & 聚合。 d. Resource-Server：【新增】弹性伸缩 – 库存管控服务，为各服务自动扩缩的资源需求保驾护航。 3）**服务画像 – 数据平台化**：Portrait-Server、Portrait-Data，弹性伸缩系统内部自建的服务画像数据体系。 4）**可观测性**：【新增】Alarm、Scanner，负责弹性伸缩的监控告警、运营治理等工作。



弹性伸缩 2.0 架构

二、挑战与应对之策

在介绍前，有必要重点说明下 2018 年这个时间节点。如前言中所述，2018 年以前弹性伸缩 1.0 这个 PaaS 平台存在的一些问题，以及整体 Hulk 1.0 容器平台落地过程中遇到的一些问题，在产品战略上我们提出了 Hulk 2.0 的计划，它涵盖调度系统、容器引擎、内核隔离增强、弹性伸缩增强等领域；在战术上，遵循自顶向下的设计原则、自底向上的实施原则。

在 2018 年 4 月前比较长的一段时间内，Hulk 容器平台的研发主要聚焦在底层系统的升级上（优先打通 Hulk 2.0 容器编排 Kubernetes、容器创建 / 销毁的流程），在弹性伸缩 PaaS 平台的投入约为 0.5 个人力，增量业务停止接入，存量业务继续维护。在 Hulk 2.0 底层系统基本成型后，容器平台团队一方面开始着手将未接入弹性伸缩平台的 Hulk 1.0 容器平滑迁移至 Hulk 2.0 容器。另外一方面，开始着手组建人力建设可对接 Hulk 2.0 容器平台编排能力的弹性伸缩 2.0 系统，为已接入弹性伸缩平台的 Hulk 1.0 容器平滑迁移过来做技术储备。

在弹性伸缩 2.0 系统的演进过程中遵循“技术”、“推广”、“运营”的演进策略。我们可以先来看下在搭建平台的过程中面临的一些挑战以及解法。

2.1 技术挑战

结合当下弹性伸缩系统面临的处境，并对业界弹性伸缩产品做过一番调研后，在平台搭建上确定了如下三期目标：

- **一期目标**：弹性伸缩 2.0 系统 MVP 版本，简单来说是把底层 OpenStack 相关生态更换成 Kubernetes 周边生态，上层功能先维持不变。
- **二期目标**：业务上，找同部门业务先行试点，基于反馈，小步快跑，快速迭代；技术上，对北京、上海服务治理平台，CMDB 系统等相关业务逻辑进行融合。
- **三期目标**：1.0 系统的用户端融合为一套，减少业务侧的理解成本、研发侧的开发 / 维护成本。

在上述三期目标基本落地后，弹性伸缩 2.0 系统基本可以跑起来，处于一个不比弹性

伸缩 1.0 系统差的阶段，接下来基于过往运维问题汇总以及业务访谈诉求，在后端架构上做了几个比较大的升级。

2.1.1 弹性调度

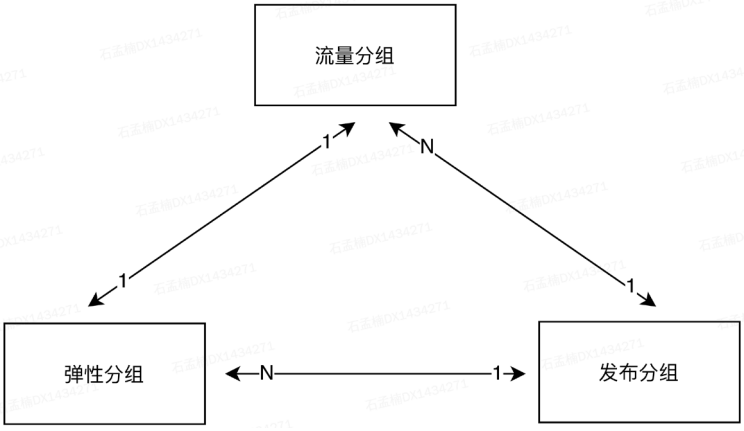
正如前言所述，和大部分弹性伸缩产品类似，早期启用弹性伸缩的过程如下：

- **创建弹性分组**：弹性分组是弹性伸缩管理的基本单元，弹性分组用来管理同质化的业务实例，在美团的实践中，主要是同一个 IDC 中的实例。
- **创建弹性伸缩配置**：用于确定扩容出来的弹性伸缩实例的机型配置，比如：CPU、内存、硬盘大小等。
- **创建弹性伸缩规则**：具体来讲就是扩容几台、缩容几台。
- **创建弹性伸缩任务**：监控任务、定时任务。

在美团的落地场景中，我们遇到如下问题：

- **该扩未扩**，业务集群新部署一个 IDC 的实例时，不容易联想到需要给这个 IDC 实例创建弹性分组，导致的问题是高峰期其他 IDC 可正常扩容，这个 IDC 无法正常扩容。
- **不该扩却扩**，业务集群不再使用某个 IDC 后，未联想到需要关停这个弹性分组，导致的问题是一些定时任务依旧扩容出这个 IDC 的实例，部分情况下会引发业务异常。
- IDC 视角的扩缩局限性，未站在“上帝视角”做扩缩容决策，会导致部分 IDC 资源紧缺时，比较难以 Fail-Fast。
- 业务在不同 IDC 的业务逻辑不一致，部分业务把具体的策略耦合在业务逻辑中，一个业务集群使用一套镜像弹性扩容，会引发业务异常。

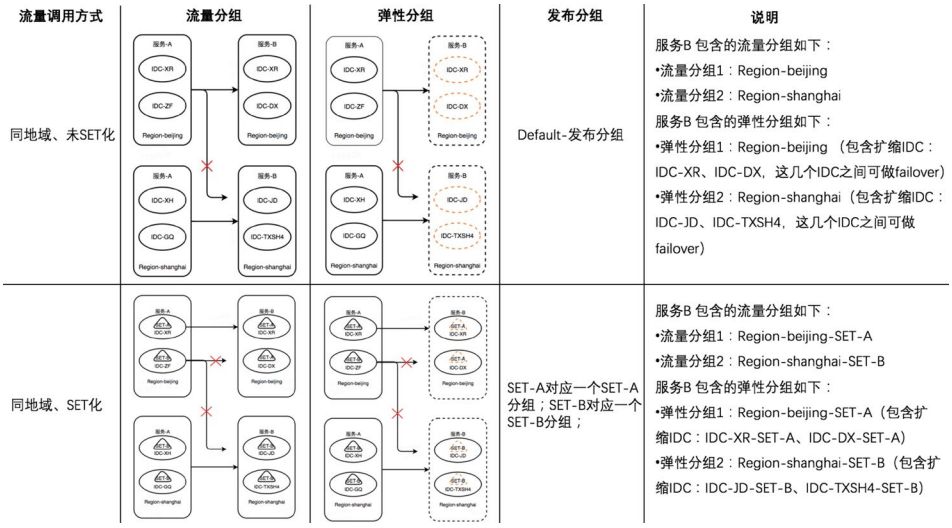
基于以上种种原因，我们拉通了各 PaaS 方，梳理了流量分组、弹性分组、发布分组之间的关系，以保证弹性伸缩在美团私有云中的扩容准确性。



分组关系图

- **流量分组**：划分来源于美团服务治理平台 -OCTO、SET 化平台 - 大禹，比如这个服务走的是 SET 化方式（类业界单元化架构），那么每一个 SET 就是一个流量分组；依次类推，设置的是无差别调用方式，那么全局就是一个流量分组；设置的是同地域（Region）调用方式，那么每个 Region 就是一个流量分组；设置的是同中心调用方式，那么每个中心就是一个流量分组；设置的是同 IDC 优先调用方式，那么每个 IDC 就是一个流量分组。
- **弹性分组**：无需手动划分，一个流量分组就对应一个弹性分组，直接 Mapping 创建即可。
- **发布分组**：划分来源于美团发布平台 -Plus，对于未采用 SET 化架构的业务集群，则仅有一个 Default- 发布分组；针对采用 SET 化架构的集群，每个 SET 对应一个 SET- 发布分组，它们的代码 / 镜像可以不一样；基于同一个 SET 下又可能有多个地域、多个中心、多个 IDC（目前美团的 SET 化架构以同 IDC 调用为主），所以和流量分组之间是 1 对 N 的关系。

同地域访问的方式比较有代表性，这里对同地域调用 & 未 SET 化、同地域调用 &SET 化的业务集群自动扩容方式进行展开说明，其他组合可依次类推。



分组明细图

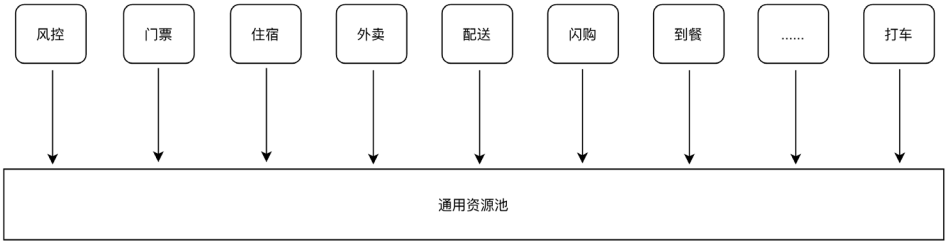
2.1.2 库存管控

弹性伸缩系统如果只是单纯地提供一个 IaaS 资源的自动供给能力，这件事情并不难。然而实际情况下，弹性伸缩系统在落地过程中需要解决资源上面临的问题。

- **业务关注点**：资源如何保障？过往在弹性伸缩系统 1.0 遇到过多次扩不出资源的情况。
- **组织关注点**：弹性资源池该划分为多大合适？如果储备大量的资源，却无法较好的进行分时复用，作为 PaaS 平台本身会造成资源闲置。

针对业务的这个关注点，目前业界公有云厂商的做法基本是不做 SLA 承诺或者说很难做到 SLA 承诺，因此也自然不会面临到组织的关注点问题。具体来说，在美团主要是通过以下几个措施来解决。

2.1.2.1 多租户管理



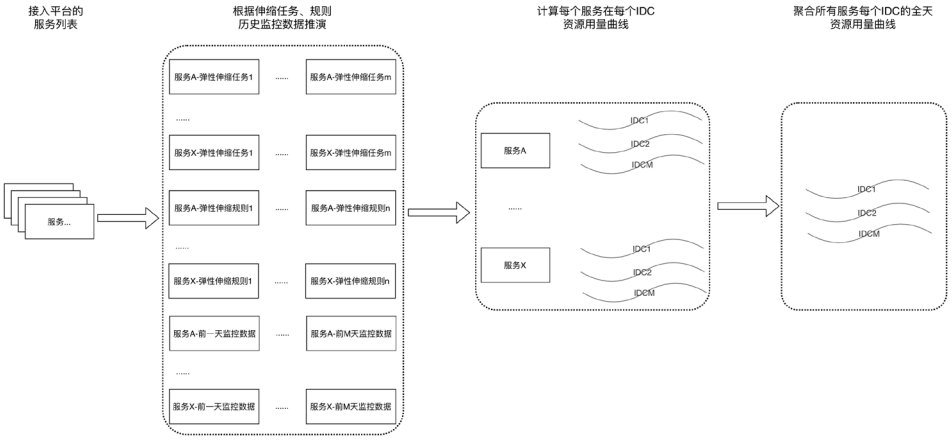
资源多租户图

平台给到每个业务线的资源并非无限，会给每个业务集群的弹性分组，设置一个默认的资源 Quota。如果业务觉得默认 Quota 不够，可通过工单的方式进行一轮评估调整（从实践来看，绝大部分情况无需调整）。针对这个资源 Quota，平台侧承诺的 SLA：99.9% 的扩容成功率。

这里会有个问题：是不是给业务 Quota 后，这部分资源就直接划分给这个业务独占了？答案：不是的。这只是一个逻辑上的划分，资源本身是各业务混用的，平台需控制资源闲置率，本身会做些“超售”，但过程中需解决好“超售”可能面临的问题，这就是水位线监管机制。

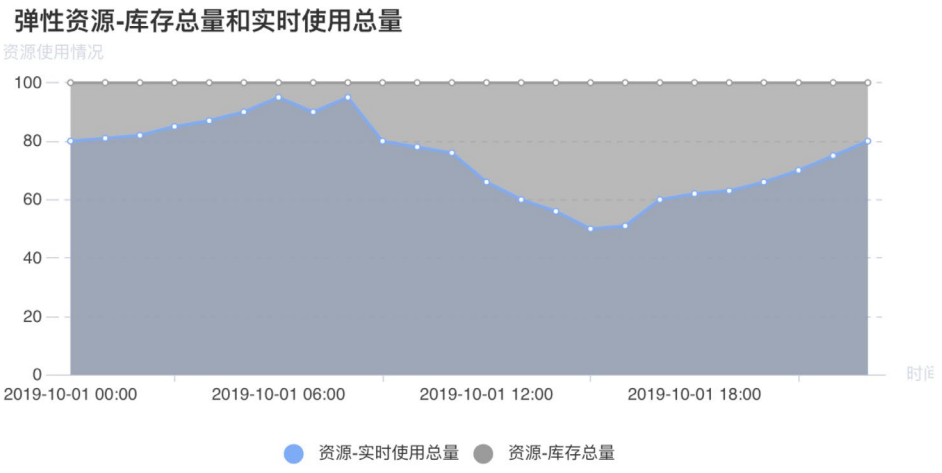
2.1.2.2 常态 – 资源保障

常规 – 资源保障，指的是平日、小节假日下的资源供给机制，这部分的资源来源于常驻资源池（架构图所示），平台会对已接入平台的所有服务，进行小时粒度的资源重预估。具体示意图如下：

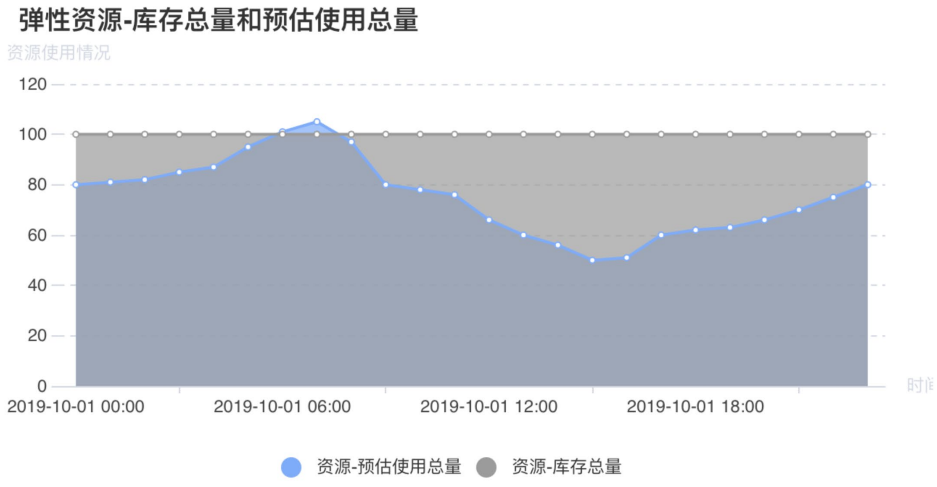


资源保障图

新增 / 更新服务的伸缩任务、伸缩规则时，会对当前变更对整体资源池水位的影响进行预估，如果在叠加时间点将超过整体资源池的水位，平台会拒绝执行变更操作，并给到用户和平台管理员消息推送，用户可通过工单方式进行资源请求说明（需保障存量服务的资源可靠性）。



库存 & 实时用量



库存 & 预估用量

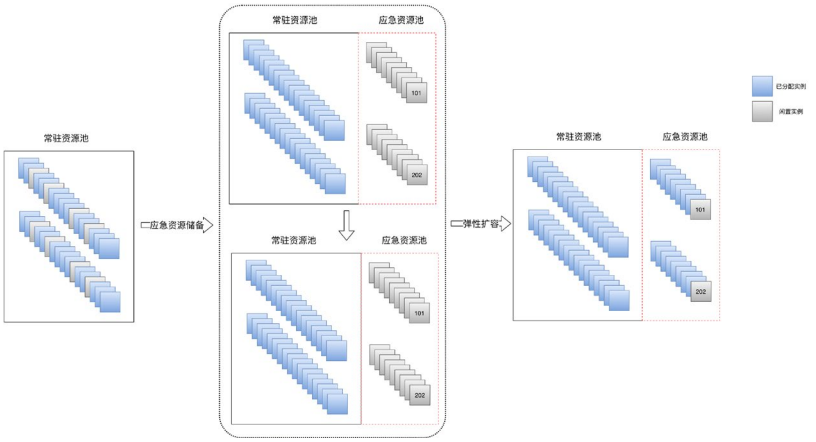
2.1.2.3 应急 – 资源保障

常驻资源池的大小是有限的，应急 – 资源保障指的是业务拉新、大促、大节假日等非常态的资源供给机制，这部分的资源来源于常驻资源池 + 应急资源池（如架构图所示）。应急资源池简单来说就是，我们按用量计费模式购买的公有云资源，这是一种更灵活的资源池模式（具备一定的租约）。在此基础上，我们构建了更省成本的混合云弹性调度能力（在此之前仅在私有云的常驻资源池调度）。

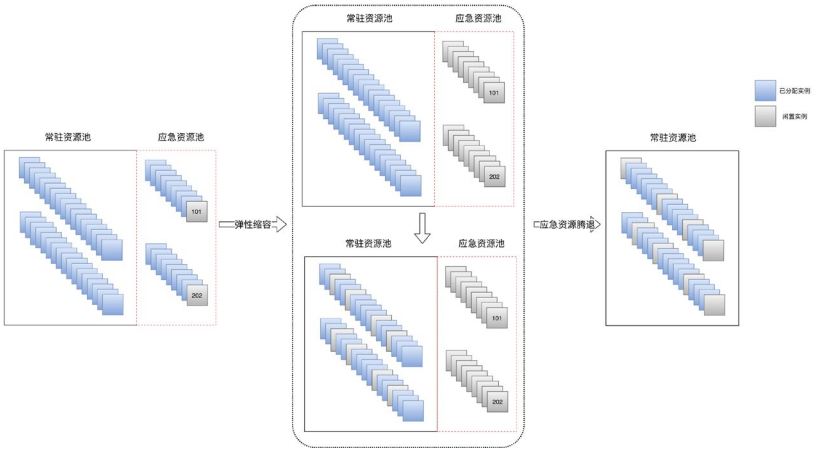
- **弹性扩容**：常驻资源池实例优先占用，应急资源池实例次之。
- **弹性缩容**：应急资源池实例先缩容，常驻资源池实例次之。

以下示意图展示的是一个服务在大促期间（维持 T 天）需要 208 台容器实例，其中 8 台为常态下的资源诉求，200 台为应急资源诉求。

大促前：



大促后（T+1）：



T+1 天后，这些应急资源池将被腾退回公有云厂商，公司无需为后续继续付费。多业务都应急的场景其实会偏复杂些；特殊情况下，需要采用重调度、治理。

2.2 推广思路

在 2020 年之前，是弹性伸缩 2.0 系统的练内功阶段，并未大规模向业务推广 Hulk 2.0 弹性伸缩系统，主要原因归结为两方面：

- 公司还处在从虚拟机、Hulk 1.0 容器迁移至 Hulk 2.0 容器阶段，Hulk 2.0 容器实例还处于打磨以及逐步赢得业务信任的过程中，推广弹性弹性伸缩 2.0 系

统，容器化是第一步。

- Hulk 2.0 系统在基础功能上不足以满足相对复杂的业务场景，比如 SET 化架构的业务。

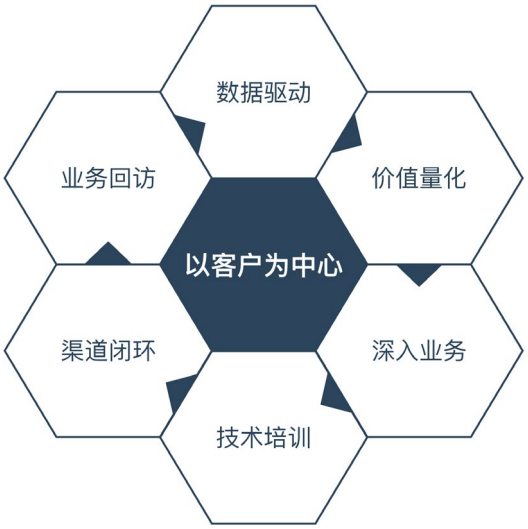
截止 2020 年年底，共接入了约 500 个服务。这里主要以弹性伸缩 1.0 系统中平滑迁移过来的服务，同部门的服务（Eat Your Own Dog Food）为主。

在 2020 年 −2021 年，是弹性伸缩 2.0 系统的规模化阶段。

- **数据驱动**：从数万个服务中，通过自建的服务画像数据体系挖掘出数千个美团适合接入弹性伸缩的服务，主要是参考高低峰、是否有状态、实例配置、业务集群规模等因素，并将其下钻到各业务部门，共建设 30+ 运营大盘，锁定了平台侧希望去赋能的业务群体。
- **价值量化**：这里面经历了一些认知迭代的过程，最后提炼出来主要是三方面：“应突发”、“省成本”、“自动化”。
- **深入业务**：在前面 500 多个服务相对比较稳定之后，大概花了两三个月，和美团的各个业务线负责人去聊，主要围绕业务介绍（只有了解它，才有可能为它赋能），弹性伸缩价值介绍（横向拉通其他业务的最佳实践），业务今年的 OKR 是哪几块（评估弹性伸缩三方面的价值是否可以帮助业务更好的达成业绩、合作共赢），一起讨论当前业务接入后可能看到的收益。
- **技术培训**：根据前期深入业务，获得的反馈。业务比较关注技术原理、接入成本、系统健壮性、最佳实践、有哪些潜在的坑。团队同学把这些进行了提炼总结，输出了弹性伸缩白皮书（技术原理、FAQ 手册、最佳实践等）、避坑指南，并在有需要的业务部门进行 VIP 式的技术分享，一次不够、可以来两次，大大小小的业务团队、公司级的分享，我们做了十几二十次。
- **渠道闭环**：公司层面上要做一些大促活动，如“安心复苏计划”、“517 吃货节”、“1001 夜直播”，这些活动只要我们知道了，我们就主动加入进去看看能帮助哪些，从结果来看，效果还是挺好的。我们还在公司的 COE 系统（故障复盘分析工具）去搜“负载”、“秒杀”、“暴涨”、“扩容”之类的关键字，学习

问题的处理过程以及待办事项，评估后发现能帮上的，我们就主动联系业务去沟通。

- **业务回访**：虽然我们会在技术支持群内会做答疑，且每周会进行值班问题的汇总 Review，但相对来说会偏零散些。我们开始是采用问卷调查的方式去获取反馈，但是践行下来效果比较一般。因此，我们还是采用比较原始的方式——“促膝长谈”，我们主动从业务侧去拉取接入后遇到的问题，在评估优先级后快速迭代系统本身。



经过以上这些工作，我们 80%+ 的服务接入了弹性，覆盖了公司 90%+ 的业务线。回到那句话，如果弹性伸缩系统只是单纯地提供一个 IaaS 资源的自动供给能力，这件事情并不难，而我们的定位是 PaaS 平台。

2.3 运营难题

这部分主要介绍向业务交付弹性容器实例后，碰到的三类典型问题：**配置、启动、性能**。这些问题大部分不是弹性伸缩 2.0 这个 PaaS 平台本身领域内可直接闭环解决的事项，这往往取决于各 PaaS 平台是否已全量标准化、业务自身逻辑、以及基础设施层性能等因素，催化我们多去做这一步的原因只有一个：**弹性扩容出的实例只有很好的帮助业务分担负载，才可真正帮助业务解决问题**。

2.3.1 配置问题

配置问题	现象	解法
非标准化的发布模式	从上述弹性扩容架构的介绍中可知，弹性伸缩系统是和发布系统紧密相连的，只有开启了标准化发布模式（内部称之为"灰度发布"，因为历史原因/特殊场景也会有非标准化发布模式）的业务才能接入平台，这块是弹性伸缩系统的准入门槛，但是在实际运行的过程中业务在代码回滚、强制发布的时候可能会退化到非标准化发布模式，在高峰时候触发了弹性扩容，在镜像决策逻辑上会存在二义性问题；	启动"safe mode"模式，可以不work，但是不能work出错。平台会自动把这块的信息通过告警的形式发送给业务。（当然在实践过程中也会有定期巡检来提前发现这个问题）
任务配置不合理	过往存在部分业务接入了弹性伸缩平台，在真正高峰期时候并未触发扩容承载流量的现象；复盘去看，存在一些现象：部分业务新人对自己参与的业务不够熟悉，配置上无法准确把握；部分同学对阈值的理解在集群维度/单机维度等出现了偏差，导致配置错误。这些都是偏惨痛的代价；	针对存量业务，结合业务的监控数据、扩缩容记录、配置情况综合评估出目前业务配置合理程度，针对不合理的配置，给到业务侧分级推送风险（高、中、低）；针对增量业务，提供基础的弹性任务推荐能力，在业务进行任务配置、规则配置的过程中给到推荐值作为引导。

2.3.2 启动问题

启动问题，大部分解决的是容器的这种开箱即用方式所面临的问题，而非过往的采用先申请机器、再在这上面发布代码包的方式。而且这里面会经常要面临着业务的质疑，为什么我手动发布的时候不存在这个问题，采用弹性扩容就出现了?

启动问题	现象	解法
端口冲突	业务端口使用不规范导致（公司统一配置10240-65000段端口，用于系统随机分配的临时端口。当业务注册服务的端口也在该区间，应用启动，Listen端口时，可能出现对应端口，被系统其它进程临时占用，导致冲突，应用小概率启动失败）	方案一，推动业务改造，存量不规范服务修改端口，但会涉及上下游一并改造，成本高/收益小，难度较大； 方案二，推动RPC框架升级，增加端口注册重试逻辑，发现冲突时，休眠一定时间再次注册，但该问题无法根治，因成本较小已落地； 方案三，修改内核参数，容器启动时，设置ip_local_reserved_ports参数（容器粒度，不影响宿主机），将用户使用的端口排除，避免随机端口使用；经过试验该方案只在3.10内核版本的服务器上才生效，而生产环境上也有一定量的低于这个版本的服务器，该问题无法根治，且成本高/收益小，未推该方案； 兜底方案，弹性伸缩系统提供了运营治理工具：自动优雅重启容器，使其自愈；
数据库连接配置	批量扩容机器时，多个客户端实例创建大量的连接，导致服务端超时	将风险系统化告知业务，并推荐业务做一些数据库客户端连接池的配置修改
慢启动	部分服务启动时长过程超过10分钟，平台会将其归类为慢启动实例	将风险推系统化告知业务（比如：未来如果存在突发流量，即使毫秒级扩容出实例，业务也无法承载流量），将是否改造的决定权交给业务。
未挂载负载均衡系统	未内嵌自动挂载负载均衡系统SDK的业务，在手动发布的过程中会借助于发布平台上提供的Hook机制；	将风险推系统化告知业务，将改造方法以及是否改造的决定权交给业务。
启动瞬间-响应超时比较严重	业务镜像初始化过程中，会涉及到通用环境初始化、Agent版本更新等现象，导致CPU被非业务进程抢占；	1.基础镜像层面，推动建设agent发版规范； 2.通过nice方式降低Agent更新对CPU的抢占； 3.将通用环境初始化左移到基础镜像侧；
访问外网超时	基于安全的考虑，美团的业务访问外网需要预先申请白名单，在机器创建的过程中会进行机器粒度网络白名单的初始化以及验证逻辑，但这块和业务的代码是解耦的。过往采用手工发布的方式，在申请机器和部署代码之间有一个人工操作的时间空挡，这部分时间对于网络白名单初始化这个逻辑而言足够了，但采用镜像扩容这种开箱即用的方式，会面临业务启动了，白名单还没生效的问题；	协同调度系统层、容器层、网络层做全链路的改造升级，只有网络白名单初始化完成后，才将容器实例进行交付（无需业务关注的逻辑尽可能下沉）；

2.3.3 性能问题

生产环境中，业务容器的性能问题其实是比较复杂的，可能涉及到重 IO 容器影响、宿主机 Load 过高、多个容器突发占用 CPU 过高导致影响某个业务容器问题。相对于不使用弹性时囤积算力的稳定场景，弹性扩缩容场景中，因对资源的频繁调配会更容易遇到资源性能的问题。为了保障使用效果，Hulk 项目组主要从三个角度对性能问题完整解决：

- **事前**：服务粒度配置个性化调度策略。
- **事中**：基于服务画像数据平台提供服务时序特征、宿主机时序特征，建设多维度资源打散能力的调度算法。
- **事后**：针对存量 Node 上的重 IO、重 CPU 等容器进行重调度。

三、业务赋能场景

3.1 节假日扩缩容

业务场景：有着明显的“七节二月”特性，就流量而言周末是工作日的 1.5 倍左右，节假日流量是周末的 3~5 倍左右。服务机器基本上是按照节假日的流量部署，平时机器使用率很低。

如何使用弹性伸缩：

- 接入弹性伸缩定时任务，节假日期间弹性扩容，应对节假日流量高峰；非节假日高峰，弹性缩容，减少服务器资源开销。
- 任务配置示例：节前配置定时任务扩容；节后配置定时任务缩容；监控扩容任务作为托底。



接入效果：业务侧平均节约成本 20.4%。

3.2 日常高峰期扩容

业务场景：配送是外卖服务的下游，具有明显的午高峰特性。

如何使用弹性伸缩：

- 设置定时任务：在午高峰来临前扩容出足量的机器，在午高峰结束后缩掉富余的机器，如示例，分组【global - banner-east - 无 swimlane】绑定了 2 个定时任务，每天 09:55 扩容 125 台机器；每天 14:00 缩容 125 台机器。



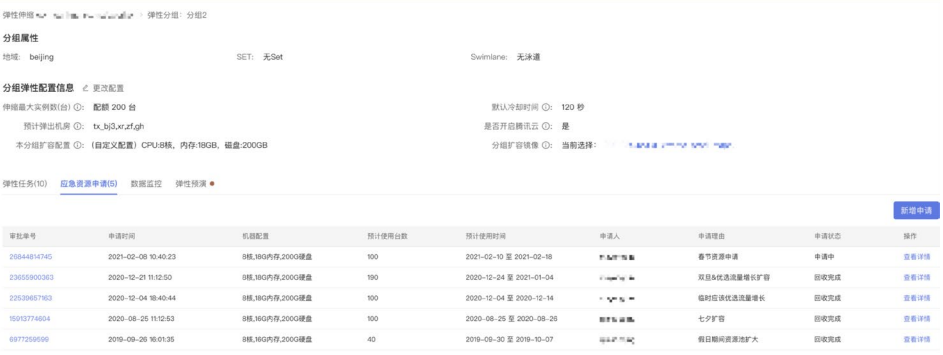


接入效果：接入弹性之前，为应对午高峰流量，该服务需要常驻机器 2420 台；接入弹性之后常驻机器释放了 365 台，高峰期弹性机器占总机器数的 15%。

3.3 应急资源保障

业务场景：风控反爬服务，是公司业务的服务安全盾和数据安全盾。目前，已有上百个业务方接入反爬服务，每天处理百亿级重点流量，为各大业务方防控恶意流量，保障业务稳定、数据安全及统计数据的正确性。风控反爬相关服务，活动节假日期间，服务负载会受业务影响增长明显，需要活动节假日期间补充大量资源。

如何使用弹性策略：活动节假日期间，风控反爬业务，申请弹性应急资源（采购公有云宿主机），提升活动期间弹性扩容总量，提升服务稳定性。活动节假日过后，缩容应急资源，腾退公有云宿主机，节约资源。



服务 A



服务 B

接入效果：为风控业务支持了 5 次大规模线上活动，基于弹性伸缩的应急 – 资源保障机制，累计提供公有云资源 700+ 台高配容器，约 7000+CPU 资源。

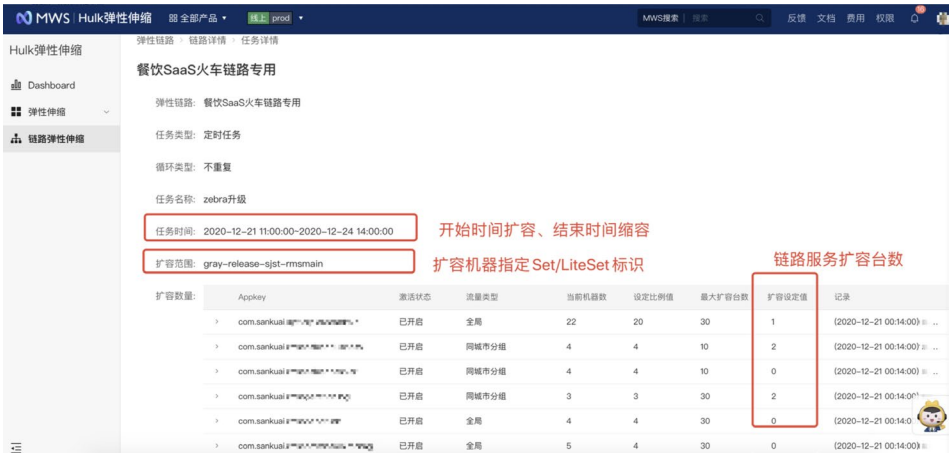
3.4 服务链路扩缩容

业务场景：餐饮 SaaS 采取“火车模型发布的模式”交付功能，需要为 70+ 服务申请专用的灰度链路机器，用于云端功能的灰度验证。但机器每月仅需使用 2~3 个工作日，一直持有机器，造成机器资源浪费；最初团队是通过每月定时手动申请、释放机器来解决，耗费较大人力，一定程度上影响到了研发的效率。

如何使用弹性策略：

- 配置链路拓扑。
- 每月活动开始前，配置链路任务，设置：扩缩容时间、机器 SET/LiteSet 标识、链路服务扩容机器数量。
- 到达设定时间，自动扩容、缩容机器。







接入效果

- 使用前：火车发版涉及 70+ 服务，每月需要 70+ 服务负责人各自在发版前扩容机器，验证完成后缩容机器。
- 使用后：首次配置链路拓扑后，此后每月仅需一名 RD 同学，统一配置一个链

路任务，达到预设时间，自动扩、缩容，大大提高效率。

四、弹性伸缩未来规划

随着弹性伸缩系统 2.0 在美团的规模化落地，我们未来会从稳定性、易用性、业务解决方案、新技术探索四个方面去做深、做广：

（1）**稳定性**：基础服务的稳定性是一切的基石，而这往往是不少研发同学容易忽视的一点，研发同学需“在晴天时修屋顶”。

- **系统自身的健壮性**：集群全链路拆分（缩爆炸半径）、池化、资源 QoS 保障能力建设。
- **弹性实例的稳定性**：加固发现能力，持续完善异常检测工具（区别于常规的健康检测，会从异构算力的宿主机、不同内核参数、各系统指标层面进行综合决策），自动进行异常实例的替换；加强数据运营，提升反馈能力，持续协同调度算法进行演进。

（2）**易用性**

- **强化预演模式**：可以预测当前的弹性伸缩规则，服务接下来 24 小时的扩缩是怎么样的。这块我们目前做了个 MVP 版本，接下来除了会进一步提升用户触达率，另外也计划在用户端为接入的用户呈现使用后收益数据。
- **自动任务配置**：基于阈值的配置方式不小程度上取决于工程师经验，可能会因为工程师过于“年轻”而没有做正确的配置，导致一些异常；目前在接入侧已经对部分业务放开了任务推荐的功能；计划基于运营数据进一步打磨工具后，会周期性的自动帮助业务调整配置。

（3）**业务解决方案**

- **链路伸缩**：目前已经实现了基于链路拓扑批量配置、对各服务伸缩规则进行拆分的能力；接下来会将服务与服务之间，服务与中间件、存储之间的背压反馈作用于弹性决策中。

- **专区弹性伸缩**：目前已在金融业务线、美团七层负载均衡服务网关 Oceanus 中发挥弹性伸缩系统的“应突发”价值，未来计划为更多有专区场景的业务提供弹性伸缩支持。

（4）**新技术探索**：借鉴 Knative、Keda 的设计理念，为云原生业务场景的弹性伸缩做技术储备。

作者简介

涂扬，现任基础架构部弹性策略团队负责人。

招聘信息

美团基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

后端

Java 线程池实现原理及其在美团业务中的实践

作者：致远　陆晨

随着计算机行业的飞速发展，摩尔定律逐渐失效，多核 CPU 成为主流。使用多线程并行计算逐渐成为开发人员提升服务器性能的基本武器。J.U.C 提供的线程池：ThreadPoolExecutor 类，帮助开发人员管理线程并方便地执行并行任务。了解并合理使用线程池，是一个开发人员必修的基本功。

本文开篇简述线程池概念和用途，接着结合线程池的源码，帮助读者领略线程池的设计思路，最后回归实践，通过案例讲述使用线程池遇到的问题，并给出了一种动态化线程池解决方案。

一、写在前面

1.1 线程池是什么

线程池（Thread Pool）是一种基于池化思想管理线程的工具，经常出现在多线程服务器中，如 MySQL。

线程过多会带来额外的开销，其中包括创建销毁线程的开销、调度线程的开销等等，同时也降低了计算机的整体性能。线程池维护多个线程，等待监督管理者分配可并发执行的任务。这种做法，一方面避免了处理任务时创建销毁线程开销的代价，另一方面避免了线程数量膨胀导致的过分调度问题，保证了对内核的充分利用。

而本文描述线程池是 JDK 中提供的 ThreadPoolExecutor 类。

当然，使用线程池可以带来一系列好处：

- **降低资源消耗**：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
- **提高响应速度**：任务到达时，无需等待线程创建即可立即执行。
- **提高线程的可管理性**：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。
- **提供更多更强大的功能**：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池 ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

1.2 线程池解决的问题是什么

线程池解决的核心问题就是资源管理问题。在并发环境下，系统不能够确定在任意时刻中，有多少任务需要执行，有多少资源需要投入。这种不确定性将带来以下若干问题：

1. 频繁申请 / 销毁资源和调度资源，将带来额外的消耗，可能会非常巨大。
2. 对资源无限申请缺少抑制手段，易引发系统资源耗尽的风险。
3. 系统无法合理管理内部的资源分布，会降低系统的稳定性。

为解决资源分配这个问题，线程池采用了“池化”（Pooling）思想。池化，顾名思义，是为了最大化收益并最小化风险，而将资源统一在一起管理的一种思想。

Pooling is the grouping together of resources (assets, equipment, personnel, effort, etc.) for the purposes of maximizing advantage or minimizing risk to the users. The term is used in finance, computing and equipment management.——wikipedia

“池化”思想不仅仅能应用在计算机领域，在金融、设备、人员管理、工作管理等领域也有相关的应用。

在计算机领域中的表现为：统一管理 IT 资源，包括服务器、存储、和网络资源等等。通过共享资源，使用户在低投入中获益。除去线程池，还有其他比较典型的几种使用策略包括：

1. 内存池 (Memory Pooling)：预先申请内存，提升申请内存速度，减少内存碎片。
2. 连接池 (Connection Pooling)：预先申请数据库连接，提升申请连接的速度，降低系统的开销。
3. 实例池 (Object Pooling)：循环使用对象，减少资源在初始化和释放时的昂贵损耗。

在了解完“是什么”和“为什么”之后，下面我们来一起深入一下线程池的内部实现原理。

二、线程池核心设计与实现

在前文中，我们了解到：线程池是一种通过“池化”思想，帮助我们管理线程而获取并发性的工具，在 Java 中的体现是 ThreadPoolExecutor 类。那么它的的详细设计与实现是什么样的呢？我们会在本章进行详细介绍。

2.1 总体设计

Java 中的线程池核心实现类是 ThreadPoolExecutor，本章基于 JDK 1.8 的源码来分析 Java 线程池的核心设计与实现。我们首先来看一下 ThreadPoolExecutor 的 UML 类图，了解下 ThreadPoolExecutor 的继承关系。

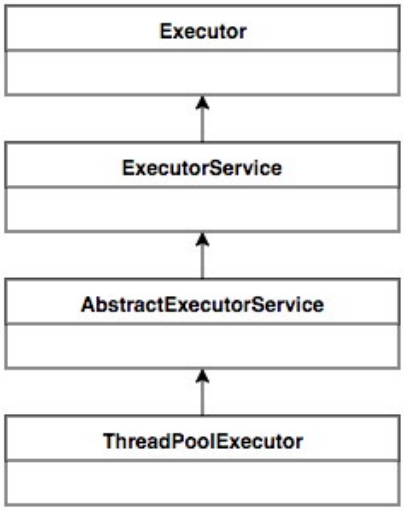


图 1　ThreadPoolExecutor UML 类图

ThreadPoolExecutor 实现的顶层接口是 Executor，顶层接口 Executor 提供了一种思想：将任务提交和任务执行进行解耦。用户无需关注如何创建线程，如何调度线程来执行任务，用户只需提供 Runnable 对象，将任务的运行逻辑提交到执行器 (Executor) 中，由 Executor 框架完成线程的调配和任务的执行部分。ExecutorService 接口增加了一些能力：(1) 扩充执行任务的能力，补充可以为一个或一批异步任务生成 Future 的方法；(2) 提供了管控线程池的方法，比如停止线程池的运行。AbstractExecutorService 则是上层的抽象类，将执行任务的流程串联了起来，保证下层的实现只需关注一个执行任务的方法即可。最下层的实现类 ThreadPoolExecutor 实现最复杂的运行部分，ThreadPoolExecutor 将会一方面维护自身的生命周期，另一方面同时管理线程和任务，使两者良好的结合从而执行并行任务。

ThreadPoolExecutor 是如何运行，如何同时维护线程和执行任务的呢？其运行机制如下图所示：

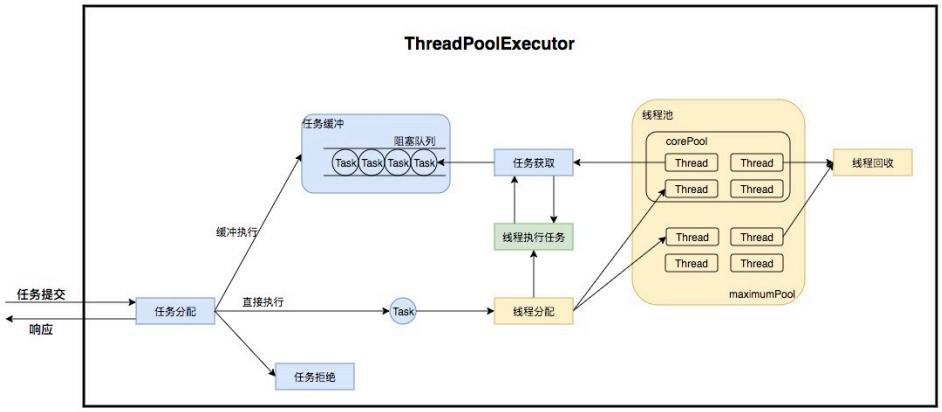


图 2　ThreadPoolExecutor 运行流程

线程池在内部实际上构建了一个生产者消费者模型，将线程和任务两者解耦，并不直接关联，从而良好的缓冲任务，复用线程。线程池的运行主要分成两部分：任务管理、线程管理。任务管理部分充当生产者的角色，当任务提交后，线程池会判断该任务后续的流转：(1) 直接申请线程执行该任务；(2) 缓冲到队列中等待线程执行；(3) 拒绝该任务。线程管理部分是消费者，它们被统一维护在线程池内，根据任务请求进行线程的分配，当线程执行完任务后则会继续获取新的任务去执行，最终当线程获取不到任务的时候，线程就会被回收。

接下来，我们会按照以下三个部分去详细讲解线程池运行机制：

1. 线程池如何维护自身状态。
2. 线程池如何管理任务。
3. 线程池如何管理线程。

2.2 生命周期管理

线程池运行的状态，并不是用户显式设置的，而是伴随着线程池的运行，由内部来维护。线程池内部使用一个变量维护两个值：运行状态 (runState) 和线程数量 (workerCount)。在具体实现中，线程池将运行状态 (runState)、线程数量 (workerCount) 两个关键参数的维护放在了一起，如下代码所示：

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

`ctl` 这个 AtomicInteger 类型，是对线程池的运行状态和线程池中有效线程的数量进行控制的一个字段，它同时包含两部分的信息：线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)，高 3 位保存 runState，低 29 位保存 workerCount，两个变量之间互不干扰。用一个变量去存储两个值，可避免在做相关决策时，出现不一致的情况，不必为了维护两者的一致，而占用锁资源。通过阅读线程池源代码也可以发现，经常出现要同时判断线程池运行状态和线程数量的情况。线程池也提供了若干方法去供用户获得线程池当前的运行状态、线程个数。这里都使用的是位运算的方式，相比于基本运算，速度也会快很多。

关于内部封装的获取生命周期状态、获取线程池线程数量的计算方法如以下代码所示：

```
private static int runStateOf(int c)     { return c & ~CAPACITY; } //计算当前运行状态
private static int workerCountOf(int c)  { return c & CAPACITY; }  //计算当前线程数量
private static int ctlOf(int rs, int wc) { return rs | wc; }   //通过状态和线程数生成 ctl
```

ThreadPoolExecutor 的运行状态有 5 种，分别为：

运行状态	状态描述
RUNNING	能接受新提交的任务，并且也能处理阻塞队列中的任务。
SHUTDOWN	关闭状态，不再接受新提交的任务，但却可以继续处理阻塞队列中已保存的任务。
STOP	不能接受新任务，也不处理队列中的任务，会中断正在处理任务的线程。
TIDYING	所有的任务都已终止了，workerCount (有效线程数) 为0。
TERMINATED	在terminated() 方法执行完后进入该状态。

其生命周期转换如下入所示：

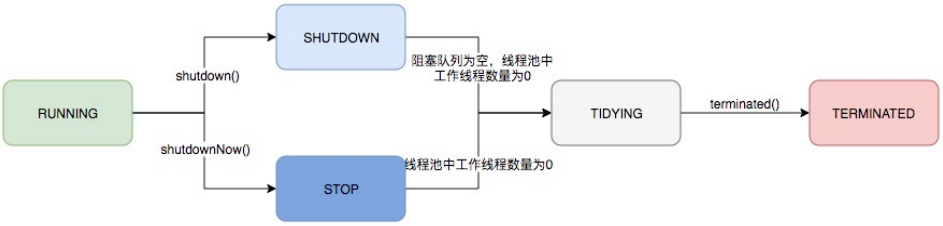


图 3　线程池生命周期

2.3 任务执行机制

2.3.1 任务调度

任务调度是线程池的主要入口，当用户提交了一个任务，接下来这个任务将如何执行都是由这个阶段决定的。了解这部分就相当于了解了线程池的核心运行机制。

首先，所有任务的调度都是由 execute 方法完成的，这部分完成的工作是：检查现在线程池的运行状态、运行线程数、运行策略，决定接下来执行的流程，是直接申请线程执行，或是缓冲到队列中执行，亦或是直接拒绝该任务。其执行过程如下：

1. 首先检测线程池运行状态，如果不是 RUNNING，则直接拒绝，线程池要保证在 RUNNING 的状态下执行任务。
2. 如果 workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务。
3. 如果 workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中。
4. 如果 workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务。
5. 如果 workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。

其执行流程如下图所示：

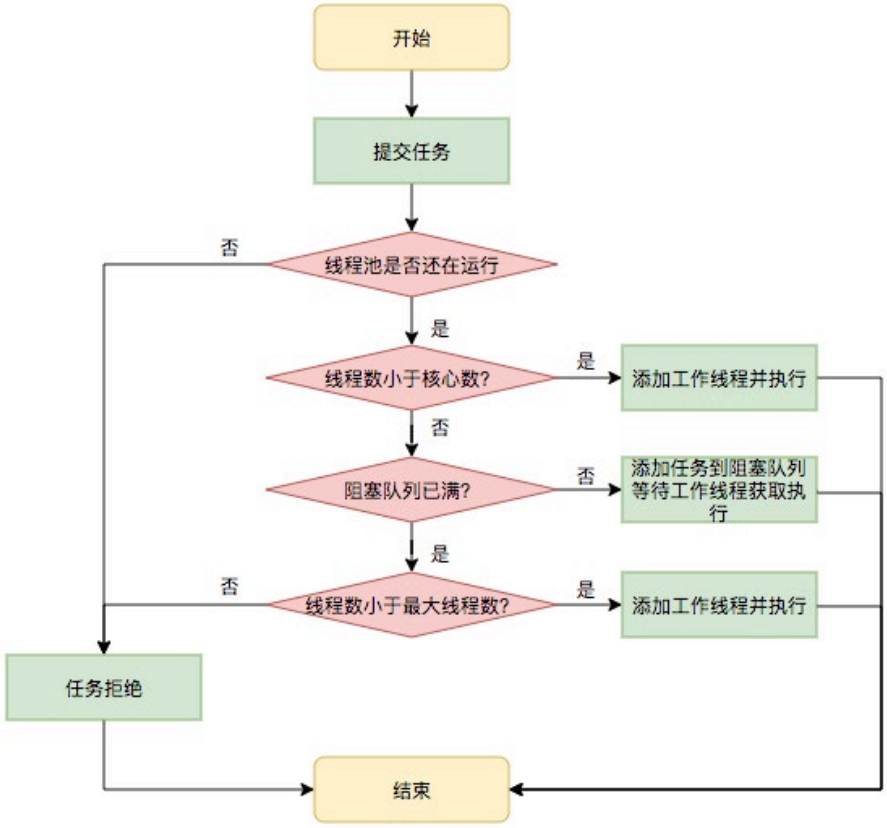


图 4　任务调度流程

2.3.2 任务缓冲

任务缓冲模块是线程池能够管理任务的核心部分。线程池的本质是对任务和线程的管理，而做到这一点最关键的思想就是将任务和线程两者解耦，不让两者直接关联，才可以做后续的分配工作。线程池中是以生产者消费者模式，通过一个阻塞队列来实现的。阻塞队列缓存任务，工作线程从阻塞队列中获取任务。

阻塞队列 (BlockingQueue) 是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，

而消费者也只从容器里拿元素。

下图中展示了线程 1 往阻塞队列中添加元素，而线程 2 从阻塞队列中移除元素：

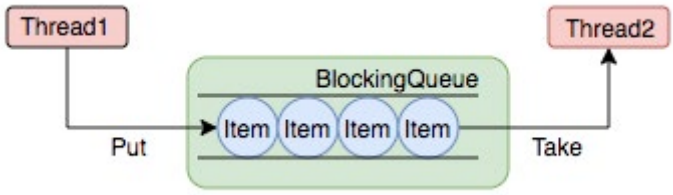


图 5　阻塞队列

使用不同的队列可以实现不一样的任务存取策略。在这里，我们可以再介绍下阻塞队列的成员：

名称	描述
ArrayBlockingQueue	一个用数组实现的有界阻塞队列，此队列按照先进先出(FIFO)的原则对元素进行排序。支持公平锁和非公平锁。
LinkedBlockingQueue	一个由链表结构组成的有界队列，此队列按照先进先出(FIFO)的原则对元素进行排序。此队列的默认长度为Integer.MAX_VALUE，所以默认创建的该队列有容量危险。
PriorityBlockingQueue	一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序。
DelayQueue	一个实现PriorityBlockingQueue实现延迟获取的无界队列，在创建元素时，可以指定多久才能从队列中获取当前元素。只有延时期满后才能从队列中获取元素。
SynchronousQueue	一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素。支持公平锁和非公平锁。SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。
LinkedTransferQueue	一个由链表结构组成的无界阻塞队列，相当于其它队列，LinkedTransferQueue队列多了transfer和tryTransfer方法。
LinkedBlockingDeque	一个由链表结构组成的双向阻塞队列。队列头部和尾部都可以添加和移除元素，多线程并发时，可以将锁的竞争最多降到一半。

2.3.3 任务申请

由上文的任务分配部分可知，任务的执行有两种可能：一种是任务直接由新创建的线程执行。另一种是线程从任务队列中获取任务然后执行，执行完任务的空闲线程会再次去从队列中申请任务再去执行。第一种情况仅出现在线程初始创建的时候，第二种是线程获取任务绝大多数的情况。

线程需要从任务缓存模块中不断地取任务执行，帮助线程从阻塞队列中获取任务，实

现线程管理模块和任务管理模块之间的通信。这部分策略由 getTask 方法实现，其执行流程如下图所示：

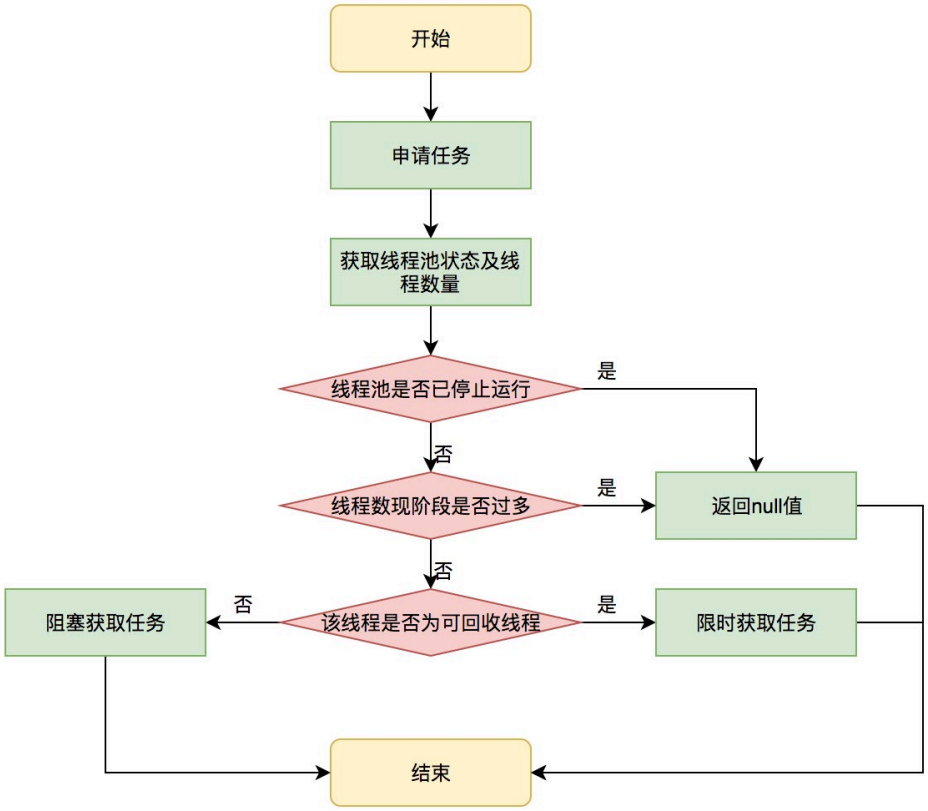


图 6　获取任务流程图

getTask 这部分进行了多次判断，为的是控制线程的数量，使其符合线程池的状态。如果线程池现在不应该持有那么多线程，则会返回 null 值。工作线程 Worker 会不断接收新任务去执行，而当工作线程 Worker 接收不到任务的时候，就会开始被回收。

2.3.4 任务拒绝

任务拒绝模块是线程池的保护部分，线程池有一个最大的容量，当线程池的任务缓存队列已满，并且线程池中的线程数目达到 maximumPoolSize 时，就需要拒绝掉该

任务，采取任务拒绝策略，保护线程池。

拒绝策略是一个接口，其设计如下：

```
public interface RejectedExecutionHandler {
    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

用户可以通过实现这个接口去定制拒绝策略，也可以选择 JDK 提供的四种已有拒绝策略，其特点如下：

	名称	描述
1	ThreadPoolExecutor.AbortPolicy	丢弃任务并抛出RejectedExecutionException异常。 这是线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐使用此拒绝策略，这样子在系统不能承载更大的并发量的时候，能够及时的通过异常发现。
2	ThreadPoolExecutor.DiscardPolicy	丢弃任务，但是不抛出异常。 使用此策略，可能会使我们无法发现系统的异常状态。建议是一些无关紧要的业务采用此策略。
3	ThreadPoolExecutor.DiscardOldestPolicy	丢弃队列最前面的任务，然后重新提交被拒绝的任务。是否要采用此种拒绝策略，还得根据实际业务是否允许丢弃老任务来认真衡量。
4	ThreadPoolExecutor.CallerRunsPolicy	由调用线程（提交任务的线程）处理该任务。这种情况是需要让所有任务都执行完毕，那么就适合大量计算的任务类型去执行，多线程仅仅是增大吞吐量的手段，最终必须要让每个任务都执行完毕。

2.4 Worker 线程管理

2.4.1 Worker 线程

线程池为了掌握线程的状态并维护线程的生命周期，设计了线程池内的工作线程 Worker。我们来看一下它的部分代码：

```
private final class Worker extends AbstractQueuedSynchronizer
implements Runnable{
    final Thread thread;//Worker 持有的线程
    Runnable firstTask;// 初始化的任务，可以为 null
}
```

Worker 这个工作线程，实现了 Runnable 接口，并持有一个线程 thread，一个初始化的任务 firstTask。thread 是在调用构造方法时通过 ThreadFactory 来创建的线程，可以用来执行任务；firstTask 用它来保存传入的第一个任务，这个任务可以有也可以为 null。如果这个值是非空的，那么线程就会在启动初期立即执行这个任务，也

就对应核心线程创建时的情况；如果这个值是 null，那么就需要创建一个线程去执行任务列表（workQueue）中的任务，也就是非核心线程的创建。

Worker 执行任务的模型如下图所示：

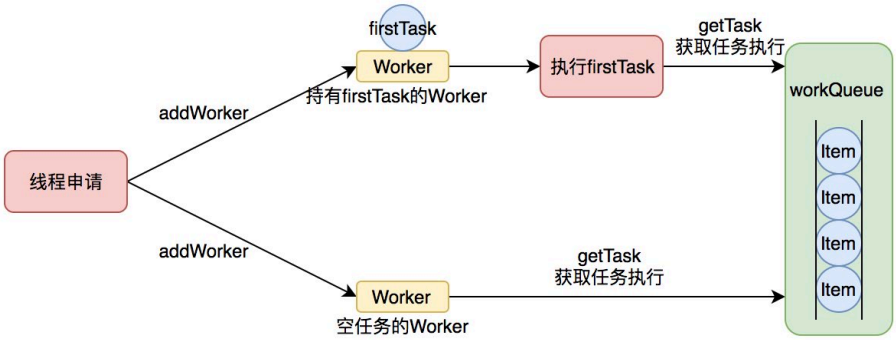


图 7　Worker 执行任务

线程池需要管理线程的生命周期，需要在线程长时间不运行的时候进行回收。线程池使用一张 Hash 表去持有线程的引用，这样可以通过添加引用、移除引用这样的操作来控制线程的生命周期。这个时候重要的就是如何判断线程是否在运行。

Worker 是通过继承 AQS，使用 AQS 来实现独占锁这个功能。没有使用可重入锁 ReentrantLock，而是使用 AQS，为的就是实现不可重入的特性去反应线程现在的执行状态。

1. lock 方法一旦获取了独占锁，表示当前线程正在执行任务中。
2. 如果正在执行任务，则不应该中断线程。
3. 如果该线程现在不是独占锁的状态，也就是空闲的状态，说明它没有在处理任务，这时可以对该线程进行中断。
4. 线程池在执行 shutdown 方法或 tryTerminate 方法时会调用 interruptIdleWorkers 方法来中断空闲的线程，interruptIdleWorkers 方法会使用 tryLock 方法来判断线程池中的线程是否是空闲状态；如果线程是空闲状态则可以安全回收。

在线程回收过程中就使用到了这种特性，回收过程如下图所示：

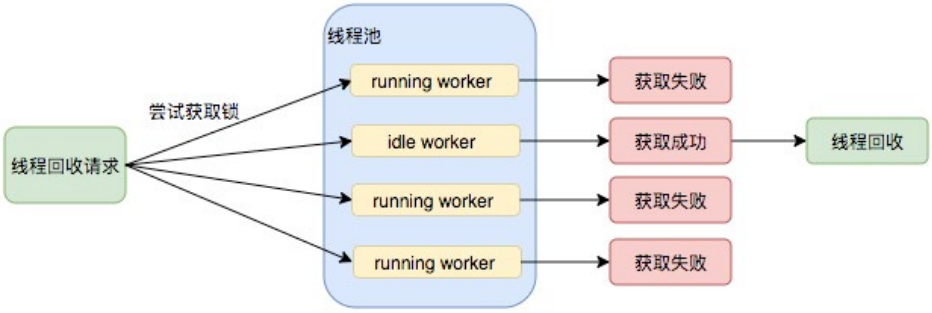


图 8　线程池回收过程

2.4.2 Worker 线程增加

增加线程是通过线程池中的 addWorker 方法，该方法的功能就是增加一个线程，该方法不考虑线程池是在哪个阶段增加的该线程，这个分配线程的策略是在上个步骤完成的，该步骤仅仅完成增加线程，并使它运行，最后返回是否成功这个结果。addWorker 方法有两个参数：firstTask、core。firstTask 参数用于指定新增的线程执行的第一个任务，该参数可以为空；core 参数为 true 表示在新增线程时会判断当前活动线程数是否少于 corePoolSize，false 表示新增线程前需要判断当前活动线程数是否少于 maximumPoolSize，其执行流程如下图所示：

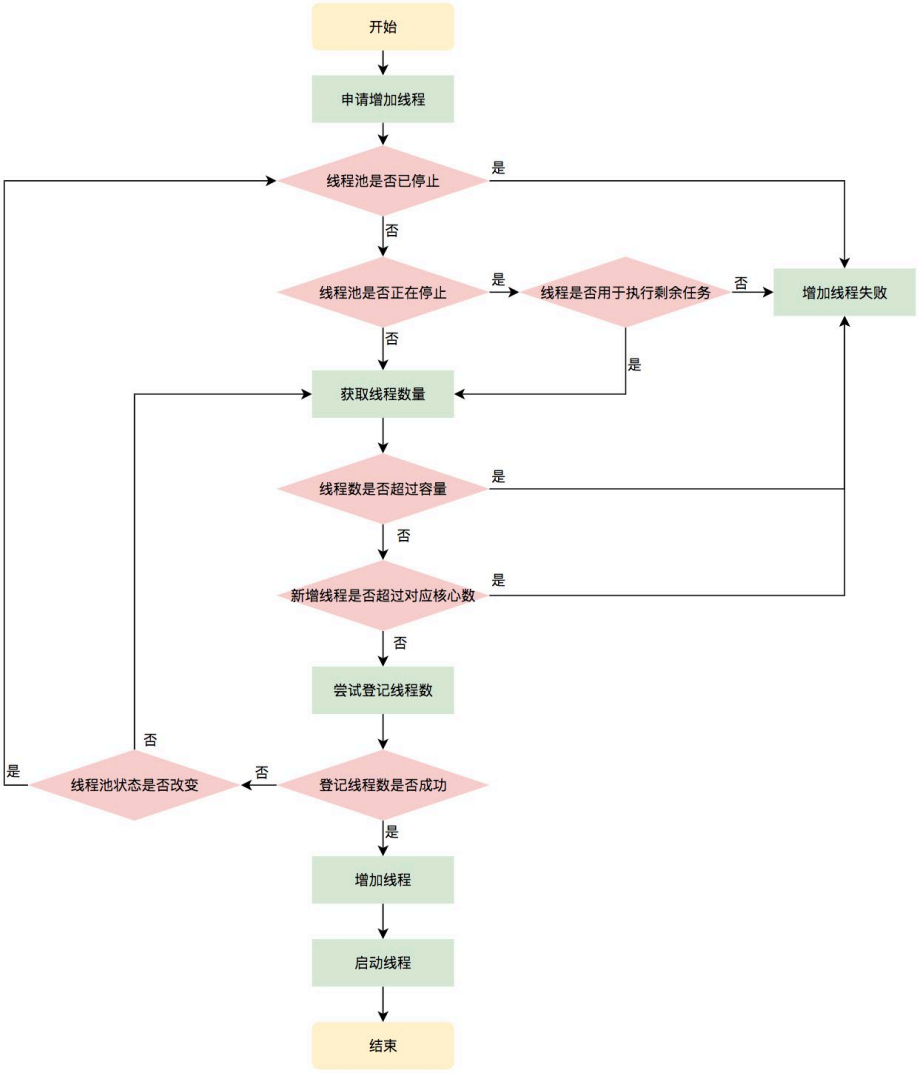


图 9　申请线程执行流程图

2.4.3 Worker 线程回收

线程池中线程的销毁依赖 JVM 自动的回收，线程池做的工作是根据当前线程池的状态维护一定数量的线程引用，防止这部分线程被 JVM 回收，当线程池决定哪些线程需要回收时，只需要将其引用消除即可。Worker 被创建出来后，就会不断地进行轮询，然后获取任务去执行，核心线程可以无限等待获取任务，非核心线程要限

时获取任务。当 Worker 无法获取到任务，也就是获取的任务为空时，循环会结束，Worker 会主动消除自身在线程池内的引用。

```
try {
  while (task != null || (task = getTask()) != null) {
    // 执行任务
  }
} finally {
  processWorkerExit(w, completedAbruptly);// 获取不到任务时，主动回收自己
}
```

线程回收的工作是在 processWorkerExit 方法完成的。

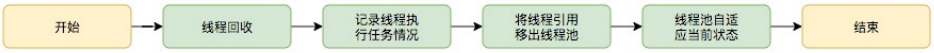


图 10　线程销毁流程

事实上，在这个方法中，将线程引用移出线程池就已经结束了线程销毁的部分。但由于引起线程销毁的可能性有很多，线程池还要判断是什么引发了这次销毁，是否要改变线程池的现阶段状态，是否要根据新状态，重新分配线程。

2.4.4 Worker 线程执行任务

在 Worker 类中的 run 方法调用了 runWorker 方法来执行任务，runWorker 方法的执行过程如下：

1. while 循环不断地通过 getTask() 方法获取任务。
2. getTask() 方法从阻塞队列中取任务。
3. 如果线程池正在停止，那么要保证当前线程是中断状态，否则要保证当前线程不是中断状态。
4. 执行任务。
5. 如果 getTask 结果为 null 则跳出循环，执行 processWorkerExit() 方法，销毁线程。

执行流程如下图所示：

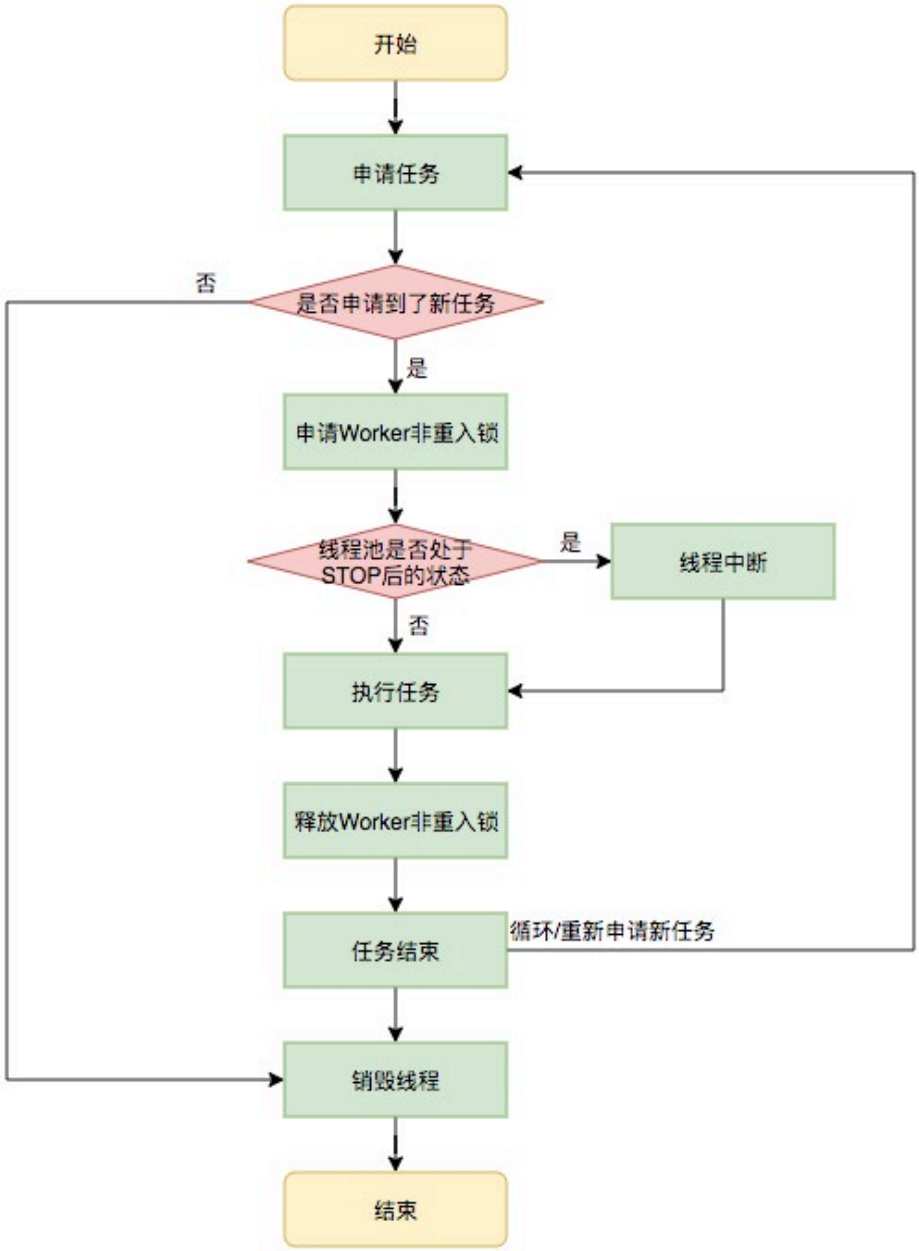


图 11　执行任务流程

三、线程池在业务中的实践

3.1 业务背景

在当今的互联网业界，为了最大程度利用 CPU 的多核性能，并行运算的能力是不可或缺的。通过线程池管理线程获取并发性是一个非常基础的操作，让我们来看两个典型的使用线程池获取并发性的场景。

场景 1：快速响应用户请求

描述：用户发起的实时请求，服务追求响应时间。比如说用户要查看一个商品的信息，那么我们需要将商品维度的一系列信息如商品的价格、优惠、库存、图片等等聚合起来，展示给用户。

分析：从用户体验角度看，这个结果响应的越快越好，如果一个页面半天都刷不出，用户可能就放弃查看这个商品了。而面向用户的功能聚合通常非常复杂，伴随着调用与调用之间的级联、多级级联等情况，业务开发同学往往会选择使用线程池这种简单的方式，将调用封装成任务并行的执行，缩短总体响应时间。另外，使用线程池也是有考量的，这种场景最重要的就是获取最大的响应速度去满足用户，所以应该不设置队列去缓冲并发任务，调高 corePoolSize 和 maxPoolSize 去尽可能创造多的线程快速执行任务。

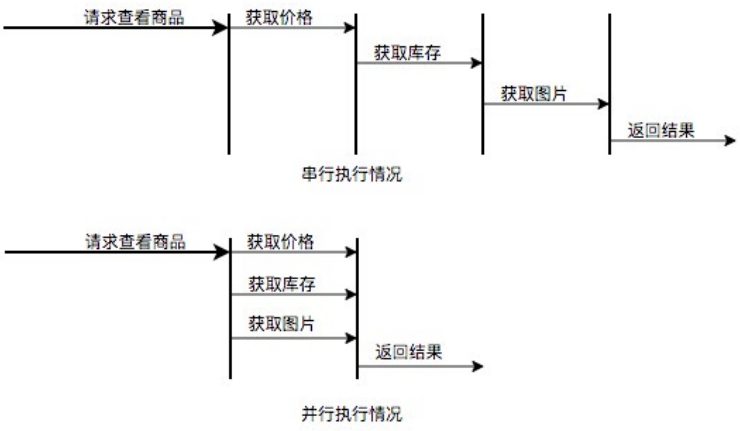


图 12　并行执行任务提升任务响应速度

场景 2：快速处理批量任务

描述：离线的大量计算任务，需要快速执行。比如说，统计某个报表，需要计算出全国各个门店中有哪些商品有某种属性，用于后续营销策略的分析，那么我们需要查询全国所有门店中的所有商品，并且记录具有某属性的商品，然后快速生成报表。

分析：这种场景需要执行大量的任务，我们也会希望任务执行的越快越好。这种情况下，也应该使用多线程策略，并行计算。但与响应速度优先的场景区别在于，这类场景任务量巨大，并不需要瞬时的完成，而是关注如何使用有限的资源，尽可能在单位时间内处理更多的任务，也就是吞吐量优先的问题。所以应该设置队列去缓冲并发任务，调整合适的 corePoolSize 去设置处理任务的线程数。在这里，设置的线程数过多可能还会引发线程上下文切换频繁的问题，也会降低处理任务的速度，降低吞吐量。

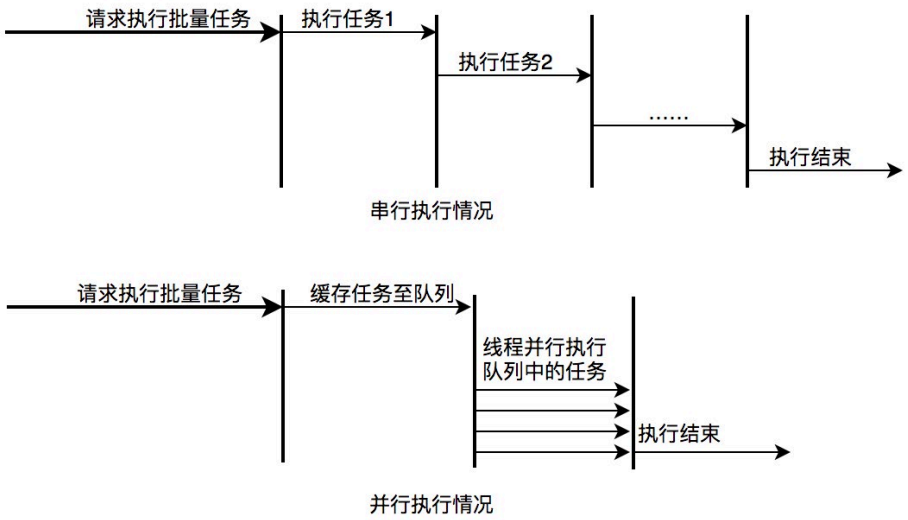


图 13　并行执行任务提升批量任务执行速度

3.2 实际问题及方案思考

线程池使用面临的核心的问题在于：**线程池的参数并不好配置**。一方面线程池的运行

机制不是很好理解，配置合理需要强依赖开发人员的个人经验和知识；另一方面，线程池执行的情况和任务类型相关性较大，IO 密集型和 CPU 密集型的任务运行起来的情况差异非常大，这导致业界并没有一些成熟的经验策略帮助开发人员参考。

关于线程池配置不合理引发的故障，公司内部有较多记录，下面举一些例子：

Case1：2018 年 XX 页面展示接口大量调用降级：

事故描述：XX 页面展示接口产生大量调用降级，数量级在几十到上百。

事故原因：该服务展示接口内部逻辑使用线程池做并行计算，由于没有预估好调用的流量，导致最大核心数设置偏小，大量抛出 RejectedExecutionException，触发接口降级条件，示意图如下：

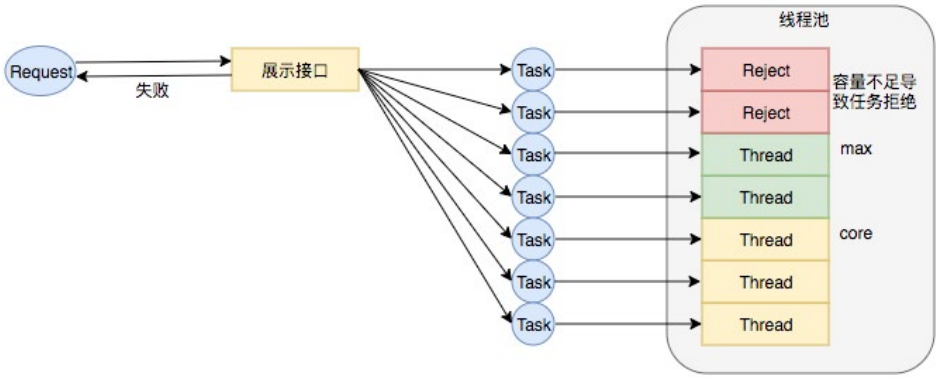


图 14　线程数核心设置过小引发 RejectExecutionException

Case2：2018 年 XX 业务服务不可用 S2 级故障

事故描述：XX 业务提供的服务执行时间过长，作为上游服务整体超时，大量下游服务调用失败。

事故原因：该服务处理请求内部逻辑使用线程池做资源隔离，由于队列设置过长，最大线程数设置失效，导致请求数量增加时，大量任务堆积在队列中，任务执行时间过长，最终导致下游服务的大量调用超时失败。示意图如下：

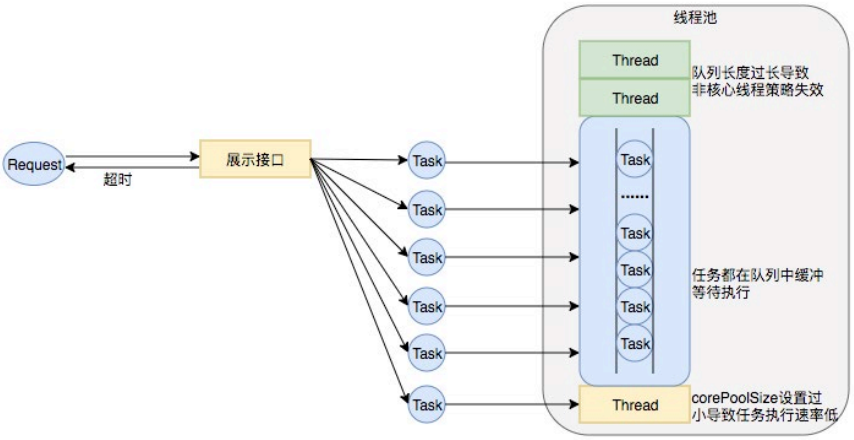


图 15　线程池队列长度设置过长、corePoolSize 设置过小导致任务执行速度低

业务中要使用线程池，而使用不当又会导致故障，那么我们怎样才能更好地使用线程池呢？针对这个问题，我们下面延展几个方向：

1. 能否不用线程池？

回到最初的问题，业务使用线程池是为了获取并发性，对于获取并发性，是否可以有什么其他的方案呢替代？我们尝试进行了一些其他方案的调研：

名称	描述	优势	劣势
Disruptor框架	线程池内部是通过一个工作队列去维护任务的执行的，它有一个根本性的缺陷：连续争用问题。也就是多个线程在申请任务时，为了合理地分配任务要付出锁资源，对比快速的任务执行来说，这部分申请的损耗是巨大的。 高性能进程间消息库LMAX使用了一个叫作环形缓冲的数据结构，用这种这个特殊的数据结构替代队列，将会避免申请任务时出现的连续争用状况。	• 避免连续争用，性能更佳	• 缺乏线程管理的能力，使用场景较少
Actor框架	Actor模型通过维护多个Actor去处理并发的任务，它放弃了直接使用线程去获取并发性，而是自己定义了一系列系统组件应该如何动作和交互的通用规则，不需要开发者直接使用线程。 通过在原生的线程或协程的级别上做了更高层次的封装，只需要开发者关心每个Actor的逻辑即可实现并发操作。由于避免了直接使用锁，很大程度解决了传统并发编程模式下大量依赖悲观锁导致的资源竞争情况。	• 无锁策略，性能更佳 • 避免直接使用线程，安全性更高	• 在Java中缺乏成熟的应用 • 内部复杂，难以排查和调试
协程框架	协程是一种用户态的轻量级线程，其拥有自己的寄存器上下文和栈，当调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。这种切换上下文的方式要小于线程的开销。在瓶颈侧重IO的情况，使用协程获得并发性要优于使用线程。	• 侧重IO情况时，性能更佳 • 与多线程策略无冲突，可结合使用	• 在Java中缺乏成熟的应用

综合考虑，这些新的方案都能在某种情况下提升并行任务的性能，然而本次重点解决的问题是如何更简易、更安全地获得的并发性。另外，Actor 模型的应用实际上甚少，只在 Scala 中使用广泛，协程框架在 Java 中维护的也不成熟。这三者现阶段都不是足够的易用，也并不能解决业务上现阶段的问题。

2. 追求参数设置合理性?

有没有一种计算公式，能够让开发同学很简易地计算出某种场景中的线程池应该是什么参数呢?

带着这样的疑问，我们调研了业界的一些线程池参数配置方案：

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ $The\ optimal\ pool\ size\ for\ keeping\ the\ processors\ at\ the\ desired\ utilization\ is:$ $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

调研了以上业界方案后，我们并没有得出通用的线程池计算方式。并发任务的执行情况和任务类型相关，IO 密集型和 CPU 密集型的任务运行起来的情况差异非常大，但这种占比是较难合理预估的，这导致很难有一个简单有效的通用公式帮我们直接计算出结果。

3. 线程池参数动态化?

尽管经过谨慎的评估，仍然不能够保证一次计算出来合适的参数，那么我们是否可以

将修改线程池参数的成本降下来，这样至少可以发生故障的时候可以快速调整从而缩短故障恢复的时间呢？基于这个思考，我们是否可以将线程池的参数从代码中迁移到分布式配置中心上，实现线程池参数可动态配置和即时生效，线程池参数动态化前后的参数修改流程对比如下：

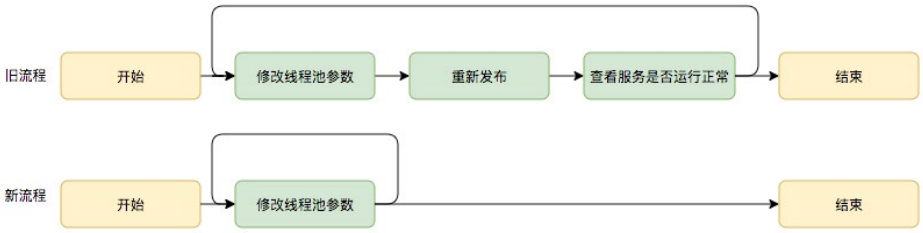


图 16　动态修改线程池参数新旧流程对比

基于以上三个方向对比，我们可以看出参数动态化方向简单有效。

3.3 动态化线程池

3.3.1 整体设计

动态化线程池的核心设计包括以下三个方面：

1. 简化线程池配置：线程池构造参数有 8 个，但是最核心的是 3 个：corePoolSize、maximumPoolSize，workQueue，它们最大程度地决定了线程池的任务分配和线程分配策略。考虑到在实际应用中我们获取并发性的场景主要是两种：（1）并行执行子任务，提高响应速度。这种情况下，应该使用同步队列，没有什么任务应该被缓存下来，而是应该立即执行。（2）并行执行大批次任务，提升吞吐量。这种情况下，应该使用有界队列，使用队列去缓冲大批量的任务，队列容量必须声明，防止任务无限制堆积。所以线程池只需要提供这三个关键参数的配置，并且提供两种队列的选择，就可以满足绝大多数的业务需求，Less is More。
2. 参数可动态修改：为了解决参数不好配，修改参数成本高等问题。在 Java 线程池留有高扩展性的基础上，封装线程池，允许线程池监听同步外部的消息，

根据消息进行修改配置。将线程池的配置放置在平台侧，允许开发同学简单的查看、修改线程池配置。

3. 增加线程池监控：对某事物缺乏状态的观测，就对其改进无从下手。在线程池执行任务的生命周期添加监控能力，帮助开发同学了解线程池状态。

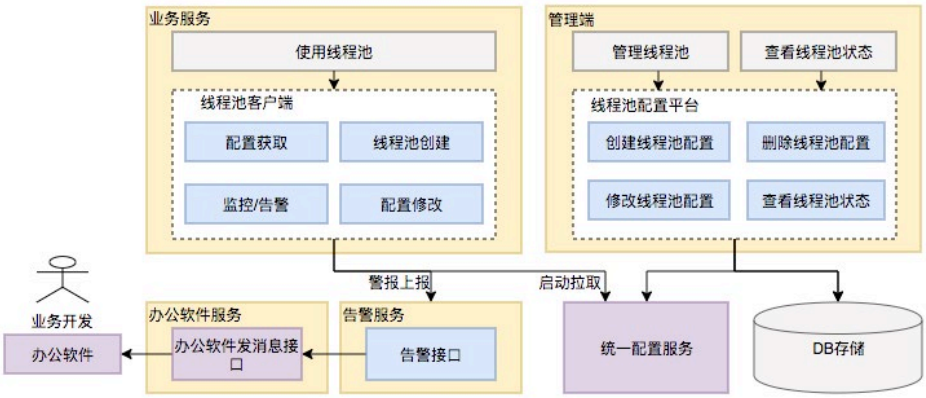


图 17　动态化线程池整体设计

3.3.2 功能架构

动态化线程池提供如下功能：

动态调参：支持线程池参数动态调整、界面化操作；包括修改线程池核心大小、最大核心大小、队列长度等；参数修改后及时生效。 **任务监控**：支持应用粒度、线程池粒度、任务粒度的 Transaction 监控；可以看到线程池的任务执行情况、最大任务执行时间、平均任务执行时间、95/99 线等。 **负载告警**：线程池队列任务积压到一定值的时候会通过大象（美团内部通讯工具）告知应用开发负责人；当线程池负载数达到一定阈值的时候会通过大象告知应用开发负责人。 **操作监控**：创建 / 修改和删除线程池都会通知到应用的开发负责人。 **操作日志**：可以查看线程池参数的修改记录，谁在什么时候修改了线程池参数、修改前的参数值是什么。 **权限校验**：只有应用开发负责人才能够修改应用的线程池参数。

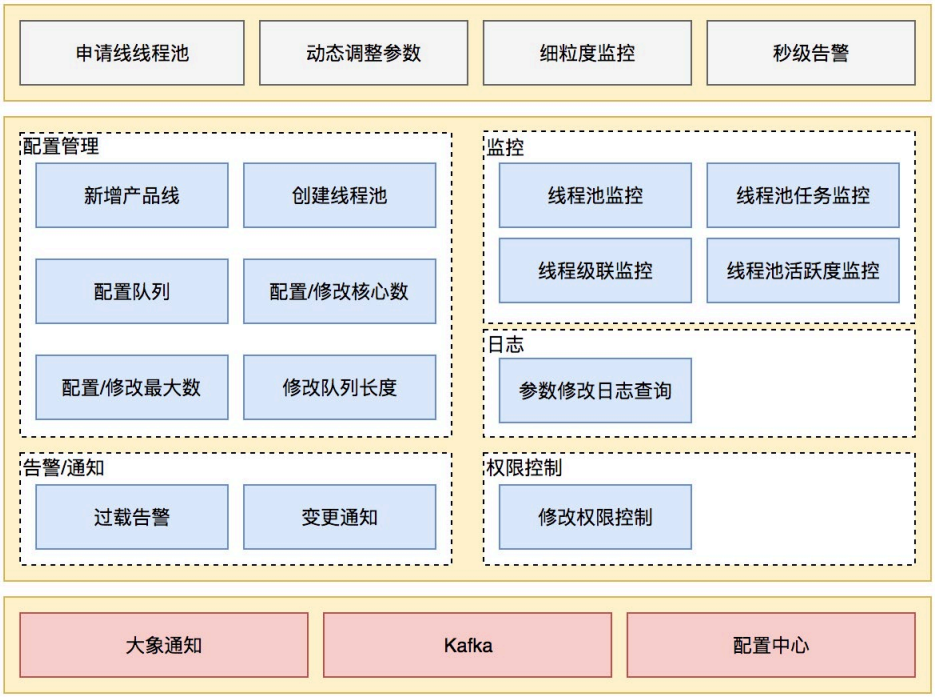


图 18　动态化线程池功能架构

参数动态化

JDK 原生线程池 ThreadPoolExecutor 提供了如下几个 public 的 setter 方法，如下图所示：

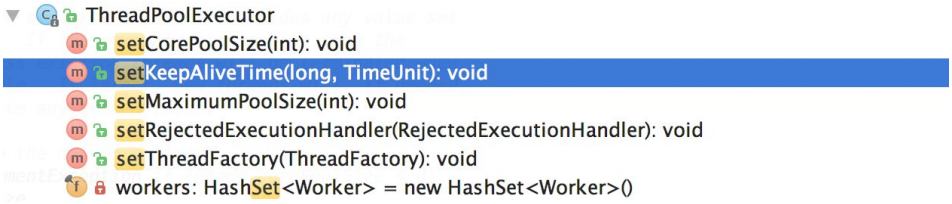


图 19　JDK 线程池参数设置接口

JDK 允许线程池使用方通过 ThreadPoolExecutor 的实例来动态设置线程池的核心策略，以 setCorePoolSize 为方法例，在运行期线程池使用方调用此方法设置 corePoolSize 之后，线程池会直接覆盖原来的 corePoolSize 值，并且基于当前

值和原始值的比较结果采取不同的处理策略。对于当前值小于当前工作线程数的情况，说明有多余的 worker 线程，此时会向当前 idle 的 worker 线程发起中断请求以实现回收，多余的 worker 在下次 idel 的时候也会被回收；对于当前值大于原始值且当前队列中有待执行任务，则线程池会创建新的 worker 线程来执行队列任务，setCorePoolSize 具体流程如下：

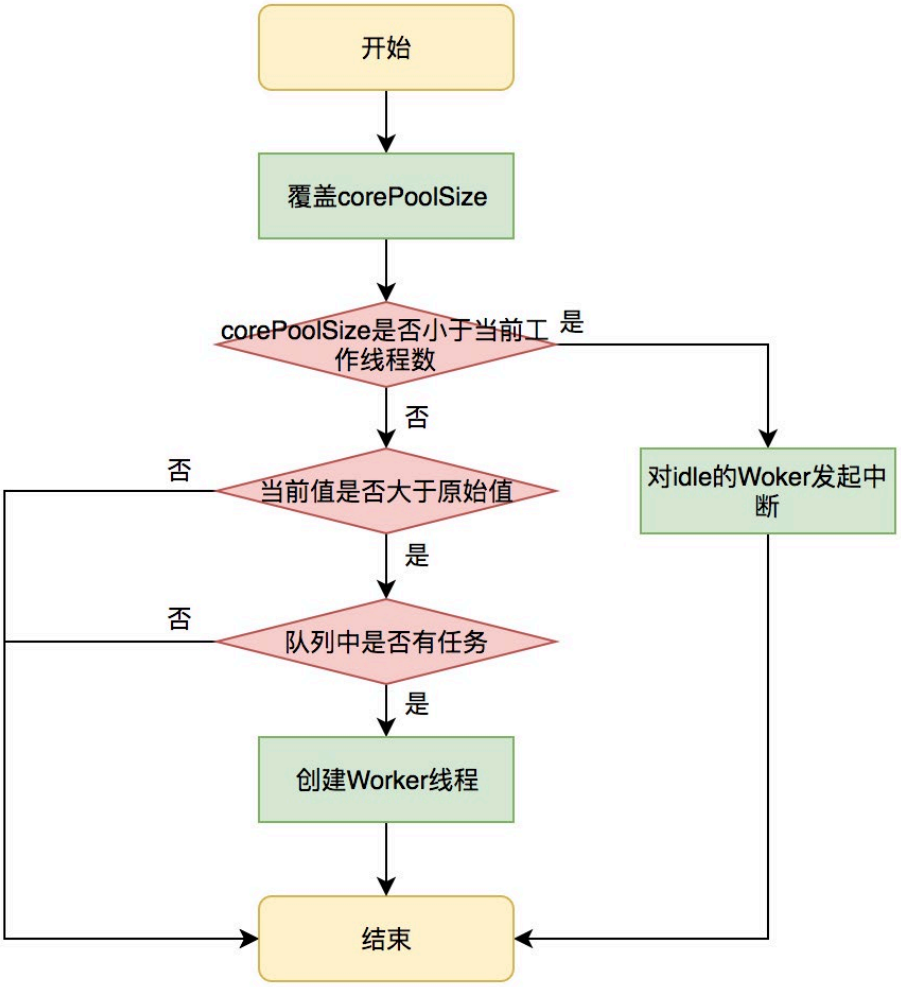


图 20　setCorePoolSize 方法执行流程

线程池内部会处理好当前状态做到平滑修改，其他几个方法限于篇幅，这里不一一

介绍。重点是基于这几个 public 方法，我们只需要维护 ThreadPoolExecutor 的实例，并且在需要修改的时候拿到实例修改其参数即可。基于以上的思路，我们实现了线程池参数的动态化、线程池参数在管理平台可配置可修改，其效果图如下图所示：



图 21　可动态修改线程池参数

用户可以在管理平台上通过线程池的名字找到指定的线程池，然后对其参数进行修改，保存后会实时生效。目前支持的动态参数包括核心数、最大值、队列长度等。除此之外，在界面中，我们还能看到用户可以配置是否开启告警、队列等待任务告警阈

值、活跃度告警等等。关于监控和告警，我们下面一节会对齐进行介绍。

线程池监控

除了参数动态化之外，为了更好地使用线程池，我们需要对线程池的运行状况有感知，比如当前线程池的负载是怎么样的？分配的资源够不够用？任务的执行情况是怎么样的？是长任务还是短任务？基于对这些问题的思考，动态化线程池提供了多个维度的监控和告警能力，包括：线程池活跃度、任务的执行 Transaction（频率、耗时）、Reject 异常、线程池内部统计信息等等，既能帮助用户从多个维度分析线程池的使用情况，又能在出现问题第一时间通知到用户，从而避免故障或加速故障恢复。

1. 负载监控和告警

线程池负载关注的核心问题是：基于当前线程池参数分配的资源够不够。对于这个问题，我们可以从事前和事中两个角度来看。事前，线程池定义了“活跃度”这个概念，来让用户在发生 Reject 异常之前能够感知线程池负载问题，线程池活跃度计算公式为：线程池活跃度 = activeCount/maximumPoolSize。这个公式代表当活跃线程数趋向于 maximumPoolSize 的时候，代表线程负载趋高。事中，也可以从两方面来看线程池的过载判定条件，一个是发生了 Reject 异常，一个是队列中有等待任务（支持定制阈值）。以上两种情况发生了都会触发告警，告警信息会通过大象推送给服务所关联的负责人。

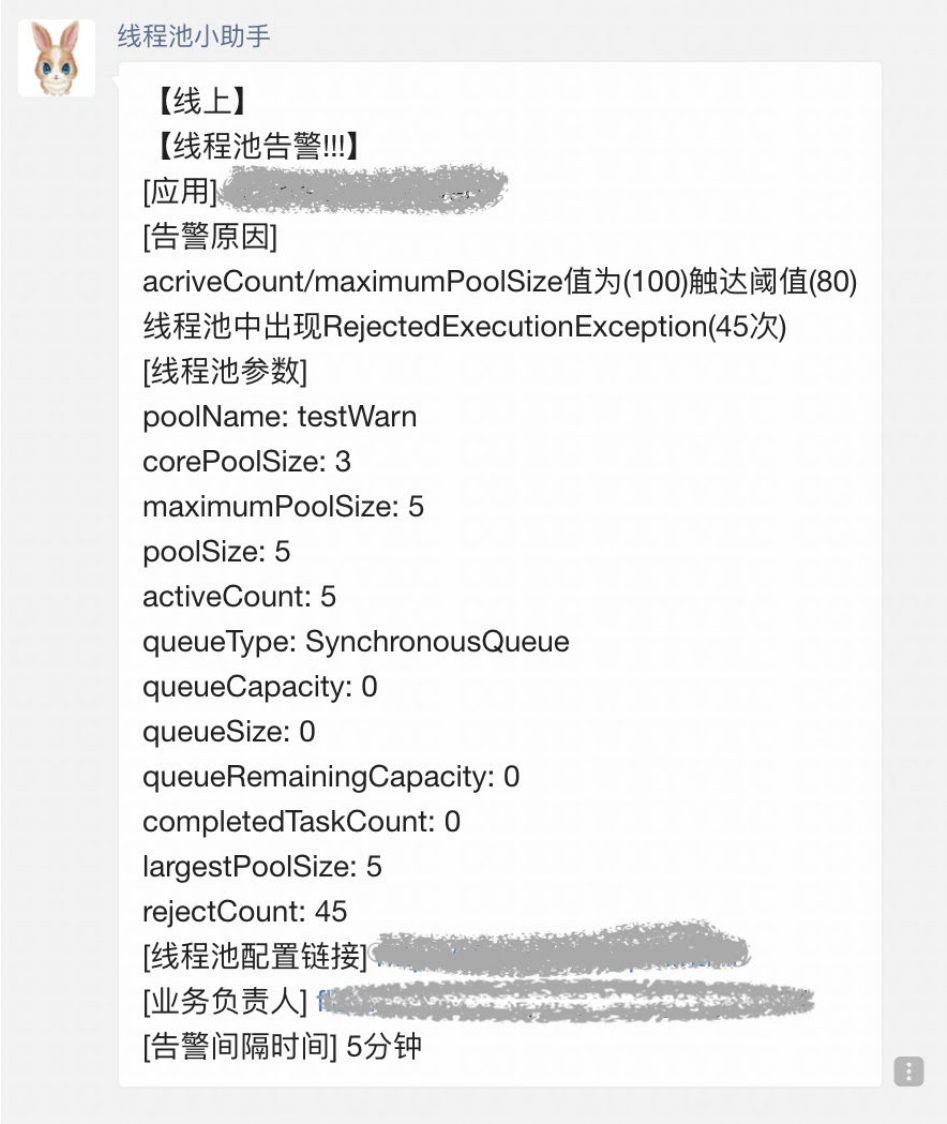


图 22　大象告警通知

2. 任务级精细化监控

在传统的线程池应用场景中，线程池中的任务执行情况对于用户来说是透明的。比如在一个具体的业务场景中，业务开发申请了一个线程池同时用于执行两种任务，一个是发消息任务、一个是发短信任务，这两类任务实际执行的频率和时长对于用户来说

没有一个直观的感受，很可能这两类任务不适合共享一个线程池，但是由于用户无法感知，因此也无从优化。动态化线程池内部实现了任务级别的埋点，且允许为不同的业务任务指定具有业务含义的名称，线程池内部基于这个名称做 Transaction 打点，基于这个功能，用户可以看到线程池内部任务级别的执行情况，且区分业务，任务监控示意图如下图所示：



图 23　线程池任务执行监控

3. 运行时状态实时查看

用户基于 JDK 原生线程池 ThreadPoolExecutor 提供的几个 public 的 getter 方法，可以读取到当前线程池的运行状态以及参数，如下图所示：

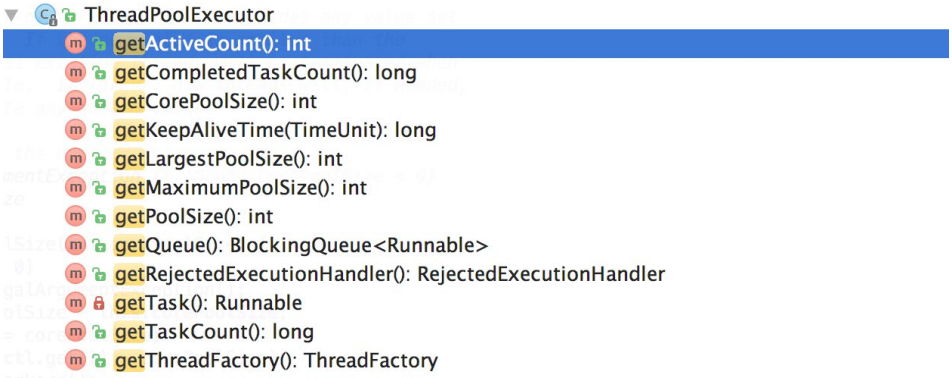
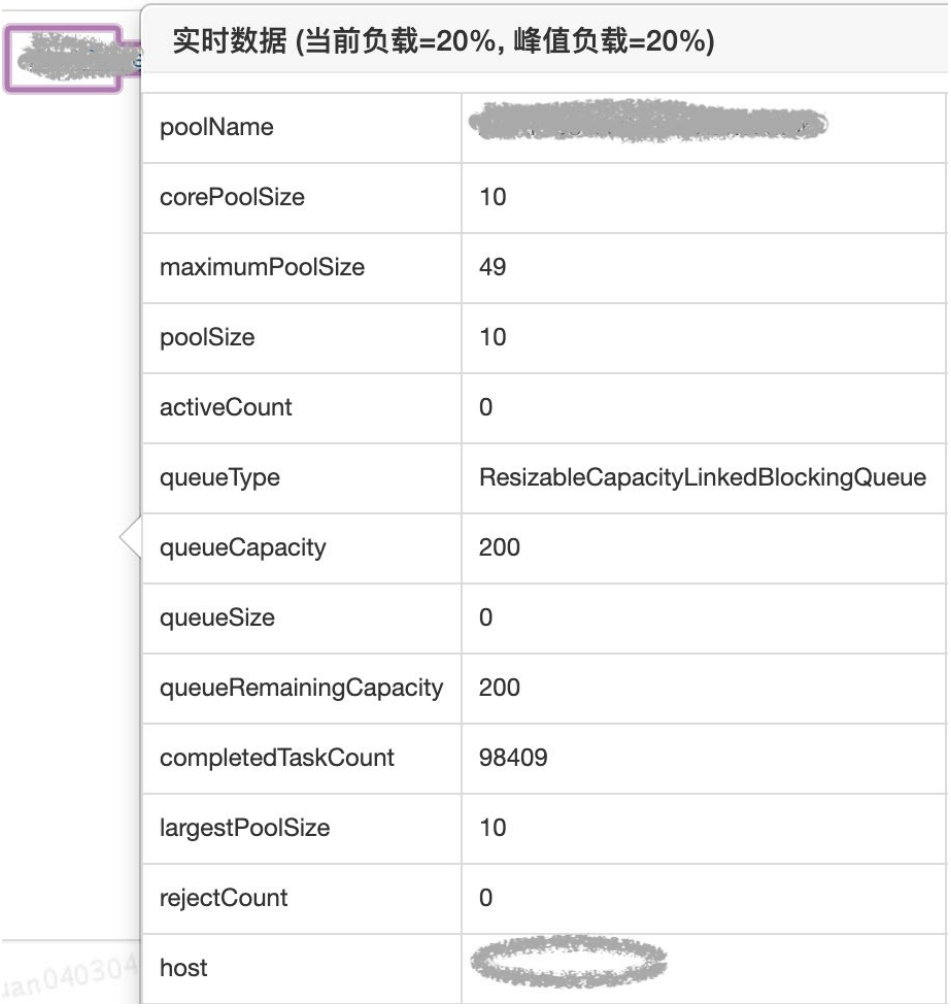


图 24　线程池实时运行情况

动态化线程池基于这几个接口封装了运行时状态实时查看的功能，用户基于这个功能可以了解线程池的实时状态，比如当前有多少个工作线程，执行了多少个任务，队列中等待的任务数等等。效果如下图所示：



实时数据 (当前负载=20%, 峰值负载=20%)

poolName	[redacted]
corePoolSize	10
maximumPoolSize	49
poolSize	10
activeCount	0
queueType	ResizableCapacityLinkedBlockingQueue
queueCapacity	200
queueSize	0
queueRemainingCapacity	200
completedTaskCount	98409
largestPoolSize	10
rejectCount	0
host	[redacted]

图 25　线程池实时运行情况

3.4 实践总结

面对业务中使用线程池遇到的实际问题，我们曾回到支持并发性问题本身来思考有没有取代线程池的方案，也曾尝试着去追求线程池参数设置的合理性，但面对业界方案具体落地的复杂性、可维护性以及真实运行环境的不确定性，我们在前两个方向上可谓“举步维艰”。最终，我们回到线程池参数动态化方向上探索，得出一个且可以解决业务问题的方案，虽然本质上还是没有逃离使用线程池的范畴，但是在成本和收益

之间，算是取得了一个很好的平衡。成本在于实现动态化以及监控成本不高，收益在于：在不颠覆原有线程池使用方式的基础之上，从降低线程池参数修改的成本以及多维度监控这两个方面降低了故障发生的概率。希望本文提供的动态化线程池思路能对大家有帮助。

四、参考资料

[1] JDK 1.8 源码
[2] 维基百科 – 线程池
[3] 更好的使用 Java 线程池
[4] 维基百科 Pooling(Resource Management)
[5] 深入理解 Java 线程池：ThreadPoolExecutor
[6]《Java 并发编程实践》

作者简介

致远，2018 年加入美团点评，美团到店综合研发中心后台开发工程师。
陆晨，2015 年加入美团点评，美团到店综合研发中心后台技术专家。

招聘信息

美团到店综合研发中心长期招聘前端、后端、数据仓库、机器学习 / 数据挖掘算法工程师，欢迎感兴趣的同学发送简历到：tech@meituan.com（邮件标题注明：美团到店综合研发中心 – 上海）

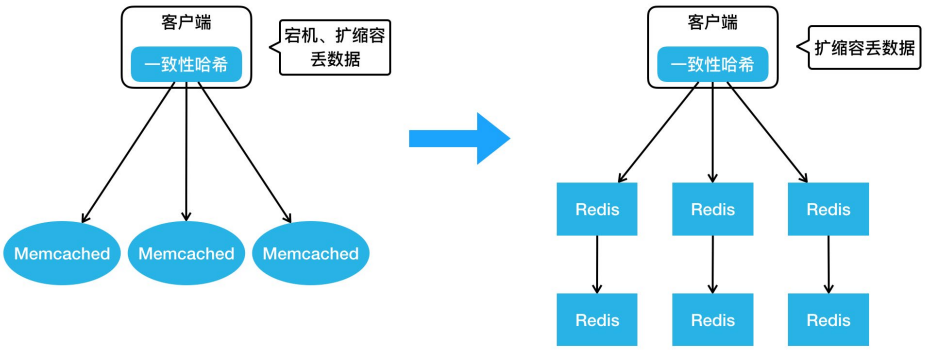
美团万亿级 KV 存储架构与实践

作者：泽斌

KV 存储作为美团一项重要的在线存储服务，承载了在线服务每天万亿级的请求量。在 2019 年 QCon 全球软件开发大会（上海站）上，美团高级技术专家齐泽斌分享了《美团点评万亿级 KV 存储架构与实践》，本文系演讲内容的整理，主要分为四个部分：第一部分讲述了美团 KV 存储的发展历程；第二部分阐述了内存 KV Squirrel 架构和实践；第三部分介绍了持久化 KV Cellar 架构和实践；最后分享了未来的发展规划和业界新趋势。

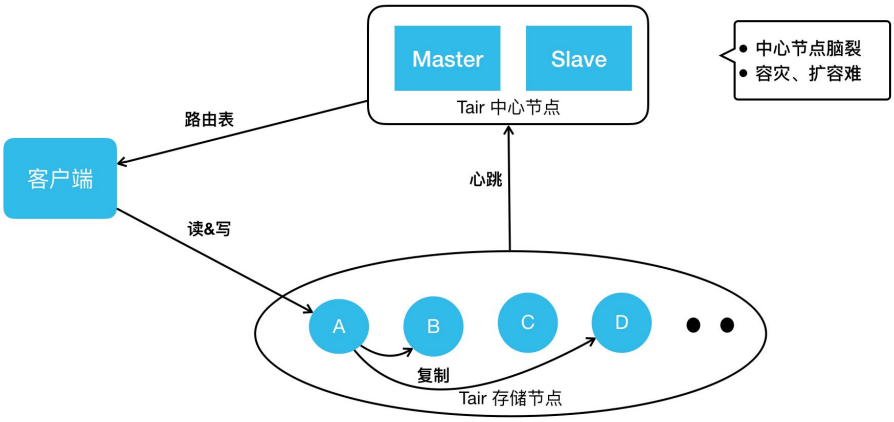
美团点评 KV 存储发展历程

美团第一代的分布式 KV 存储如下图左侧的架构所示，相信很多公司都经历过这个阶段。在客户端内做一致性哈希，在后端部署很多的 Memcached 实例，这样就实现了最基本的 KV 存储分布式设计。但这样的设计存在很明显的问题：比如在宕机摘除节点时，会丢数据，缓存空间不够需要扩容，一致性哈希也会丢失一些数据等等，这样会给业务开发带来的很多困扰。



随着 Redis 项目的成熟，美团也引入了 Redis 来解决我们上面提到的问题，进而演进出来如上图右侧这样一个架构。大家可以看到，客户端还是一样，采用了一致性哈

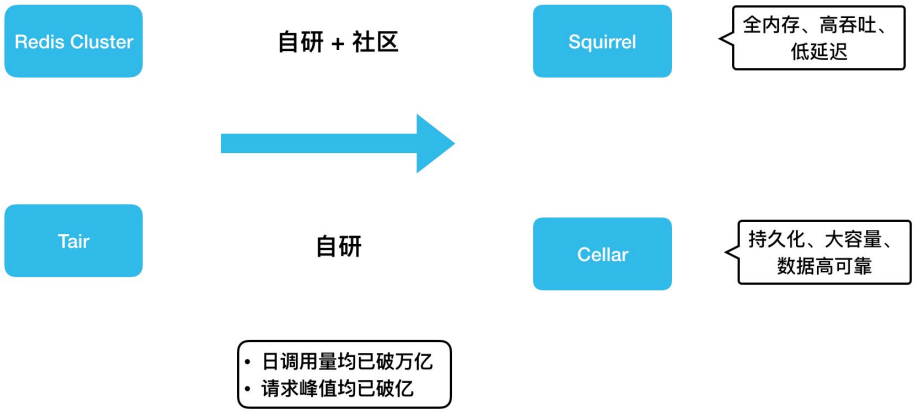
希算法，服务器端变成了 Redis 组成的主从结构。当任何一个节点宕机，我们可以通过 Redis 哨兵完成 Failover，实现高可用。但有一个问题还是没有解决，如果扩缩容的话，一致性哈希仍然会丢数据，那么这个问题该如何解决呢?



这个时候，我们发现有了一个比较成熟的 KV 存储开源项目：阿里 Tair 。2014 年，我们引入了 Tair 来满足业务 KV 存储方面的需求。Tair 开源版本的架构主要分成三部分：上图下边是存储节点，存储节点会上报心跳到它的中心节点，中心节点内部有两个配置管理节点，会监控所有的存储节点。当有任何存储节点宕机或者扩容时，它会做集群拓扑的重新构建。当客户端启动时，它会直接从中心节点拉来一个路由表。这个路由表简单来说就是一个集群的数据分布图，客户端根据路由表直接去存储节点读写。针对之前 KV 的扩容丢数据问题，它也有数据迁移机制来保证数据的完整性。

但是，我们在使用的过程中，还遇到了一些其他问题，比如中心节点虽然是主备高可用的，但实际上它没有类似于分布式仲裁的机制，所以在网络分割的情况下，它是有可能发生“脑裂”的，这个也给我们的业务造成过比较大的影响。另外，在容灾扩容时，也遇到过数据迁移影响到业务可用性的问题。另外，我们之前用过 Redis ，业务会发现 Redis 的数据结构特别丰富，而 Tair 还不支持这些数据结构。虽然我们用 Tair 解决了一些问题，但是 Tair 也无法完全满足业务需求。毕竟，在美团这样一个业务规模较大和业务复杂度较高的场景下，很难有开源系统能很好地满足我们的需

求。最终，我们决定在已应用的开源系统之上进行自研。



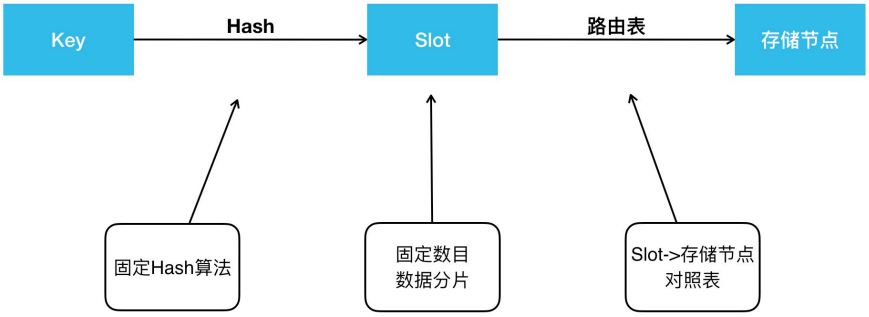
刚好在 2015 年，Redis 官方正式发布了集群版本 Redis Cluster。所以，我们紧跟社区步伐，并结合内部需求做了很多开发工作，演进出了全内存、高吞吐、低延迟的 KV 存储 Squirrel。另外，基于 Tair，我们还加入了很多自研的功能，演进出持久化、大容量、数据高可靠的 KV 存储 Cellar 。因为 Tair 的开源版本已经有四五年没有更新了，所以，Cellar 的迭代完全靠美团自研，而 Redis 社区一直很活跃。总的来说，Squirrel 的迭代是自研和社区并重，自研功能设计上也会尽量与官方架构进行兼容。后面大家可以看到，因为这些不同，Cellar 和 Squirrel 在解决同样的问题时也选取了不同的设计方案。

这两个存储其实都是 KV 存储领域不同的解决方案。在实际应用上，如果业务的数据量小，对延迟敏感，我们建议大家用 Squirrel ；如果数据量大，对延迟不是特别敏感，我们建议用成本更低的 Cellar 。目前这两套 KV 存储系统在美团内部每天的调用量均已突破万亿，它们的请求峰值也都突破了每秒亿级。

内存 KV Squirrel 架构和实践

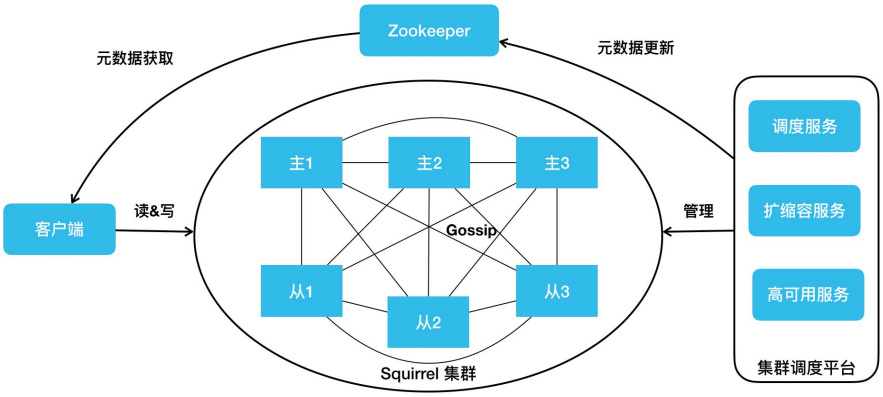
在开始之前，本文先介绍两个存储系统共通的地方。比如分布式存储的经典问题：数据是如何分布的？这个问题在 KV 存储领域，就是 Key 是怎么分布到存储节点上的。这

里 Squirrel 跟 Cellar 是一样的。当我们拿到一个 Key 后，用固定的哈希算法拿到一个哈希值，然后将哈希值对 Slot 数目取模得到一个 Slot id，我们两个 KV 现在都是预分片 16384 个 Slot 。得到 Slot id 之后，再根据路由表就能查到这个 Slot 存储在哪个存储节点上。这个路由表简单来说就是一个 Slot 到存储节点的对照表。



KV 数据分布介绍

接下来讲一下对高可用架构的认知，个人认为高可用可以从宏观和微观两个角度来看。从宏观的角度来看，高可用就是指容灾怎么做。比如说挂掉了一个节点，你该怎么做？一个机房或者说某个地域的一批机房宕机了，你该怎么做？而从微观的角度看，高可用就是怎么能保证端到端的高成功率。我们在做一些运维升级或者扩缩容数据迁移的时候，能否做到业务请求的高可用？ 本文也会从宏观和微观两个角度来分享美团做的一些高可用工作。



Squirrel 架构

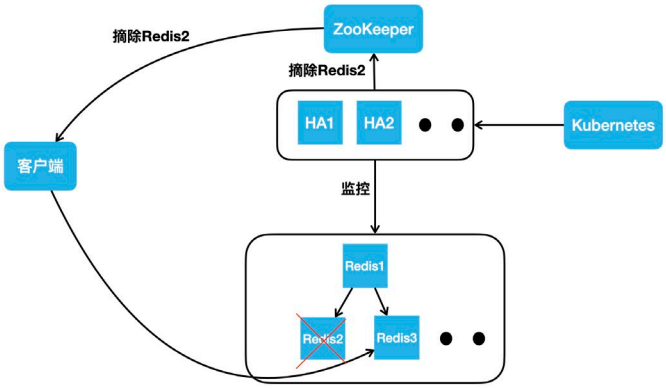
上图就是我们的 Squirrel 架构。中间部分跟 Redis 官方集群是一致的。它有主从的结构，Redis 实例之间通过 Gossip 协议去通信。我们在右边添加了一个集群调度平台，包含调度服务、扩缩容服务和高可用服务等，它会去管理整个集群，把管理结果作为元数据更新到 ZooKeeper。我们的客户端会订阅 ZooKeeper 上的元数据变更，实时获取到集群的拓扑状态，直接在 Redis 集群进行读写操作。

Squirrel 节点容灾

然后再看一下 Squirrel 容灾怎么做。对于 Redis 集群而言，节点宕机已经有完备的处理机制了。官方提供的方案，任何一个节点从宕机到被标记为 FAIL 摘除，一般需要经过 30 秒。主库的摘除可能会影响数据的完整性，所以，我们需要谨慎一些。但是对于从库呢？我们认为这个过程完全没必要。另一点，我们都知道内存的 KV 存储数据量一般都比较小。对于业务量很大的公司来说，它往往会有很多的集群。如果发生交换机故障，会影响到很多的集群，宕机之后去补副本就会变得非常麻烦。为了解决这两个问题，我们做了 HA 高可用服务。

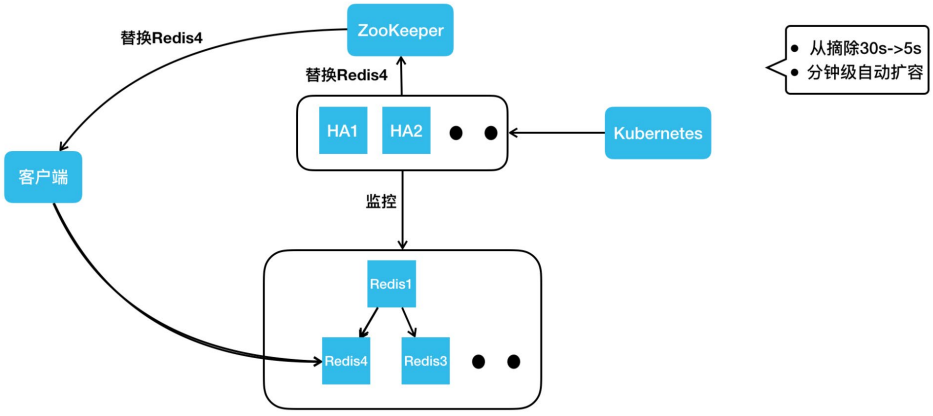
它的架构如下图所示，它会实时监控集群的所有节点。不管是网络抖动，还是发生了宕机（比如说 Redis 2），它可以实时更新 ZooKeeper，告诉 ZooKeeper 去摘除 Redis 2，客户端收到消息后，读流量就直接路由到 Redis 3 上。如果 Redis 2 只

是几十秒的网络抖动，过几十秒之后，如果 HA 节点监控到它恢复后，会把它重新加回。



Squirrel—节点容灾

如果过了一段时间，HA 判断它属于一个永久性的宕机，HA 节点会直接从 Kubernetes 集群申请一个新的 Redis 4 容器实例，把它加到集群里。此时，拓扑结构又变成了一主两从的标准结构，HA 节点更新完集群拓扑之后，就会去写 ZooKeeper 通知客户端去更新路由，客户端就能到 Redis 4 这个新从库上进行读操作。

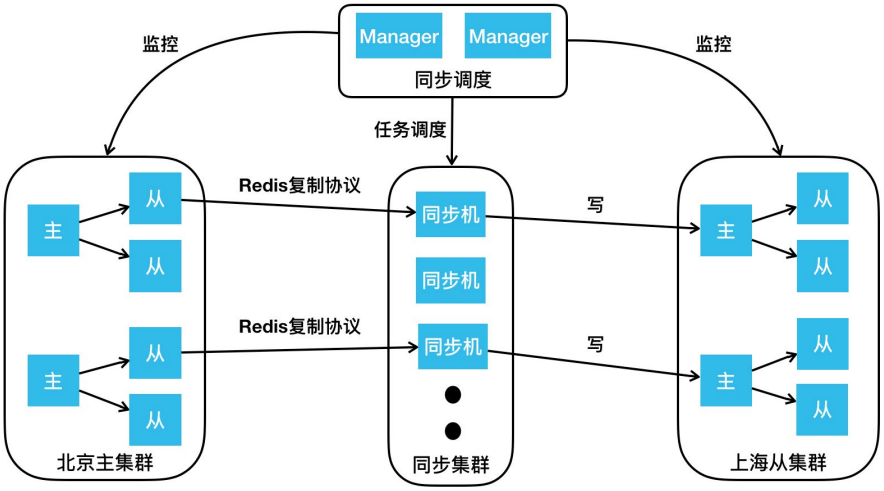


通过上述方案，我们把从库的摘除时间从 30 秒降低到了 5 秒。另外，我们通过 HA 自动申请容器实例加入集群的方式，把宕机补副本变成了一个分钟级的自动操作，不

需要任何人工的介入。

Squirrel 跨地域容灾

我们解决了单节点宕机的问题，那么跨地域问题如何解决呢？我们首先来看下跨地域有什么不同。第一，相对于同地域机房间的网络而言，跨地域专线很不稳定；第二，跨地域专线的带宽是非常有限且昂贵的。而集群内的复制没有考虑极端的网络环境。假如我们把主库部署到北京，两个从库部署在上海，同样一份数据要在北上专线传输两次，这样会造成巨大的专线带宽浪费。另外，随着业务的发展和演进，我们也在做单元化部署和异地多活架构。用官方的主从同步，满足不了我们的这些需求。基于此，我们又做了集群间的复制方案。



如上图所示，这里画出了北京的主集群以及上海的从集群，我们要做的是通过集群同步服务，把北京主集群的数据同步到上海从集群上。按照流程，首先要向我们的同步调度模块下发“在两个集群间建立同步链路”的任务，同步调度模块会根据主从集群的拓扑结构，把主从集群间的同步任务下发到同步集群，同步集群收到同步任务后会扮成 Redis 的 Slave，通过 Redis 的复制协议，从主集群上的从库拉取数据，包括 RDB 以及后续的增量变更。同步机收到数据后会把它转成客户端的写命令，写到上

海从集群的主节点里。通过这样的方式，我们把北京主集群的数据同步到了上海的从集群。同样的，我们要做异地多活也很简单，再加一个反向的同步链路，就可以实现集群间的双向同步。

接下来我们讲一下如何做好微观角度的高可用，也就是保持端到端的高成功率。对于 Squirrel ，主要讲如下三个影响成功率的问题：

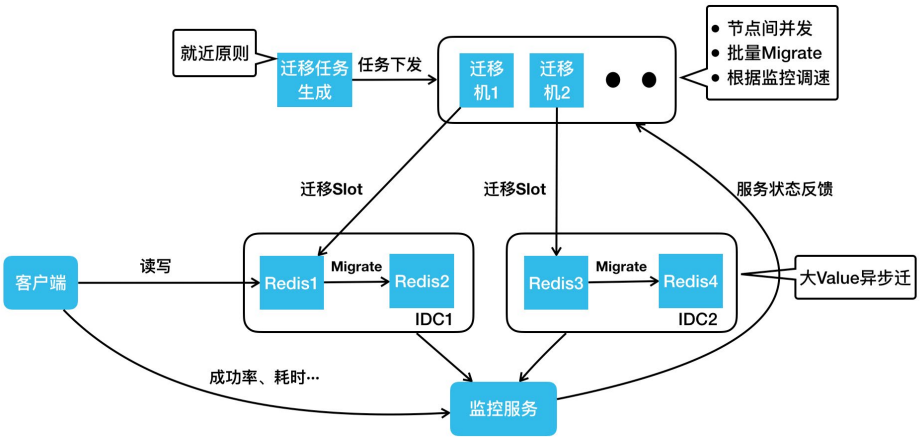
- 数据迁移造成超时抖动。
- 持久化造成超时抖动。
- 热点 Key 请求导致单节点过载。

Squirrel 智能迁移

对于数据迁移，我们主要遇到三个问题：

- Redis Cluster 虽然提供了数据迁移能力，但是对于要迁哪些 Slot，Slot 从哪迁到哪，它并不管。
- 做数据迁移的时候，大家都想越快越好，但是迁移速度过快又可能影响业务正常请求。
- Redis 的 Migrate 命令会阻塞工作线程，尤其在迁移大 Value 的时候会阻塞特别久。

为了解决这些问题，我们做了全新的迁移服务。



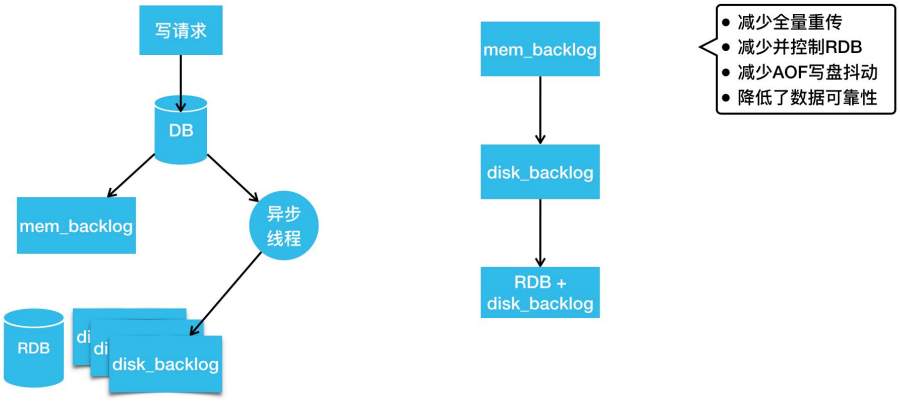
下面我们按照工作流，讲一下它是如何运行的。首先生成迁移任务，这步的核心是“就近原则”，比如说同机房的两个节点做迁移肯定比跨机房的两个节点快。迁移任务生成之后，会把任务下发到一批迁移机上。迁移机迁移的时候，有这样几个特点：

- 第一，会在集群内迁出节点间做并发，比如同时给 Redis 1、Redis 3 下发迁移命令。
- 第二，每个 Migrate 命令会迁移一批 Key。
- 第三，我们会用监控服务去实时采集客户端的成功率、耗时，服务端的负载、QPS 等，之后把这个状态反馈到迁移机上。迁移数据的过程就类似 TCP 慢启动的过程，它会把速度一直往上加，若出现请求成功率下降等情况，它的速度就会降低，最终迁移速度会在动态平衡中稳定下来，这样就达到了最快速的迁移，同时又尽可能小地影响业务的正常请求。

接下来，我们看一下大 Value 的迁移，我们实现了一个异步 Migrate 命令，该命令执行时，Redis 的主线程会继续处理其他的正常请求。如果此时有对正在迁移 Key 的写请求过来，Redis 会直接返回错误。这样最大限度保证了业务请求的正常处理，同时又不会阻塞主线程。

Squirrel 持久化重构

Redis 主从同步时会生成 RDB。生成 RDB 的过程会调用 Fork 产生一个子进程去写数据到硬盘，Fork 虽然有操作系统的 COW 机制，但是当内存用量达到 10 G 或 20 G 时，依然会造成整个进程接近秒级的阻塞。这对在线业务来说几乎是无法接受的。我们也会为数据可靠性要求高的业务去开启 AOF，而开 AOF 就可能因 IO 抖动造成进程阻塞，这也会影响请求成功率。对官方持久化机制的这两个问题，我们的解决方案是重构持久化机制。



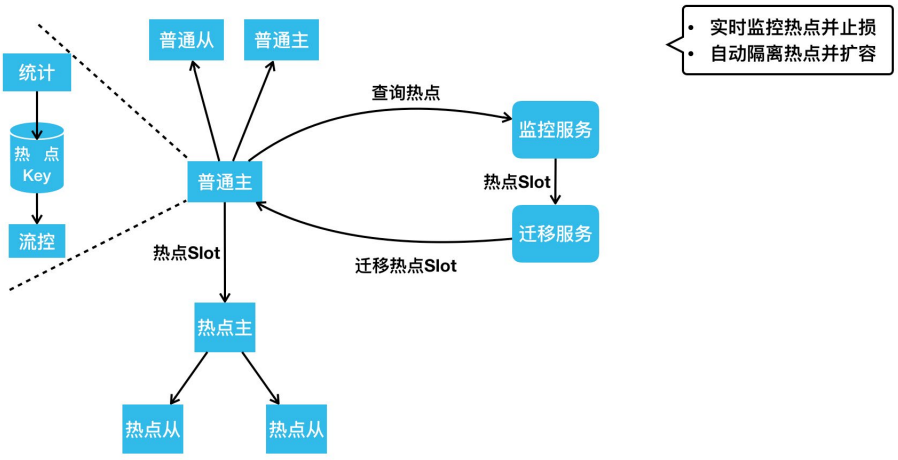
上图是我们最新版的 Redis 持久化机制，写请求会先写到 DB 里，然后写到内存 Backlog，这跟官方是一样的。同时它会把请求发给异步线程，异步线程负责把变更刷到硬盘的 Backlog 里。当硬盘 Backlog 过多时，我们会主动在业务低峰期做一次 RDB ，然后把 RDB 之前生成的 Backlog 删除。

如果这时候我们要做主从同步，去寻找同步点的时候，该怎么办？第一步还是跟官方一样，我们会从内存 Backlog 里找有没有要求的同步点，如果没有，我们会去硬盘 Backlog 找同步点。由于硬盘空间很大，硬盘 Backlog 可以存储特别多的数据，所以很少会出现找不到同步点的情况。如果硬盘 Backlog 也没有，我们就会触发一次类似于全量重传的操作，但这里的全量重传是不需要当场生成 RDB 的，它可以直接用硬盘已存的 RDB 及其之后的硬盘 Backlog 完成全量重传。通过这个设计，我们减

少了很多的全量重传。另外，我们通过控制在低峰区生成 RDB ，减少了很多 RDB 造成的抖动。同时，我们也避免了写 AOF 造成的抖动。不过，这个方案因为写 AOF 是完全异步的，所以会比官方的数据可靠性差一些，但我们认为这个代价换来了可用性的提升，这是非常值得的。

Squirrel 热点 Key

下面看一下 Squirrel 的热点 Key 解决方案。如下图所示，普通主、从是一个正常集群中的节点，热点主、从是游离于正常集群之外的节点。我们看一下它们之间怎么发生联系。

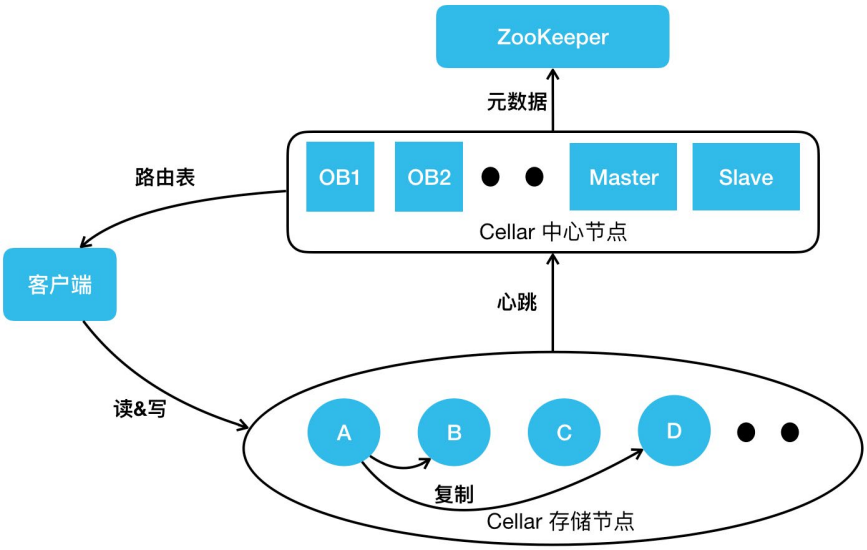


当有请求进来读写普通节点时，节点内会同时做请求 Key 的统计。如果某个 Key 达到了一定的访问量或者带宽的占用量，会自动触发流控以限制热点 Key 访问，防止节点被热点请求打满。同时，监控服务会周期性的去所有 Redis 实例上查询统计到的热点 Key。如果有热点，监控服务会把热点 Key 所在 Slot 上报到我们的迁移服务。迁移服务这时会把热点主从节点加入到这个集群中，然后把热点 Slot 迁移到这个热点主从上。因为热点主从上只有热点 Slot 的请求，所以热点 Key 的处理能力得到了大幅提升。通过这样的设计，我们可以做到实时的热点监控，并及时通过流控去

止损；通过热点迁移，我们能做到自动的热点隔离和快速的容量扩充。

持久化 KV Cellar 架构和实践

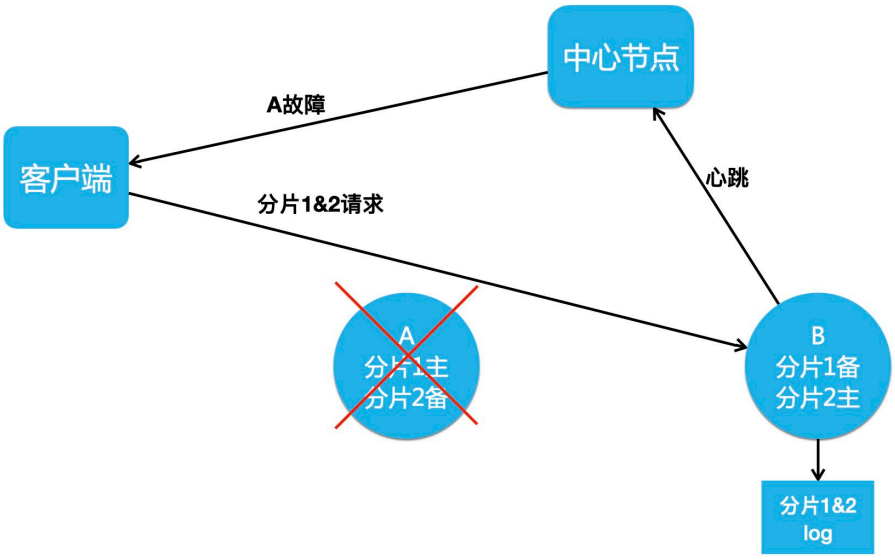
下面看一下持久化 KV Cellar 的架构和实践。下图是我们最新的 Cellar 架构图。



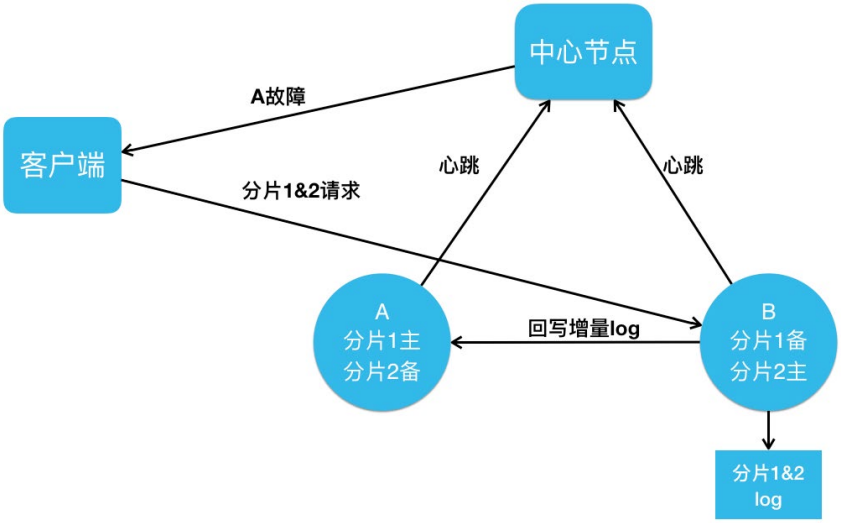
跟阿里开源的 Tair 主要有两个架构上的不同。第一个是 OB，第二个是 ZooKeeper。我们的 OB 跟 ZooKeeper 的 Observer 是类似的作用，提供 Cellar 中心节点元数据的查询服务。它可以实时与中心节点的 Master 同步最新的路由表，客户端的路由表都是从 OB 去拿。 这样做的好处主要有两点，第一，把大量的业务客户端跟集群的大脑 Master 做了天然的隔离，防止路由表请求影响集群的管理。第二，因为 OB 只供路由表查询，不参与集群的管理，所以它可以进行水平扩展，极大地提升了我们路由表的查询能力。另外，我们引入了 ZooKeeper 做分布式仲裁，解决我刚才提到的 Master、Slave 在网络分割情况下的“脑裂”问题，并且通过把集群的元数据存储到 ZooKeeper，我们保证了元数据的高可靠。

Cellar 节点容灾

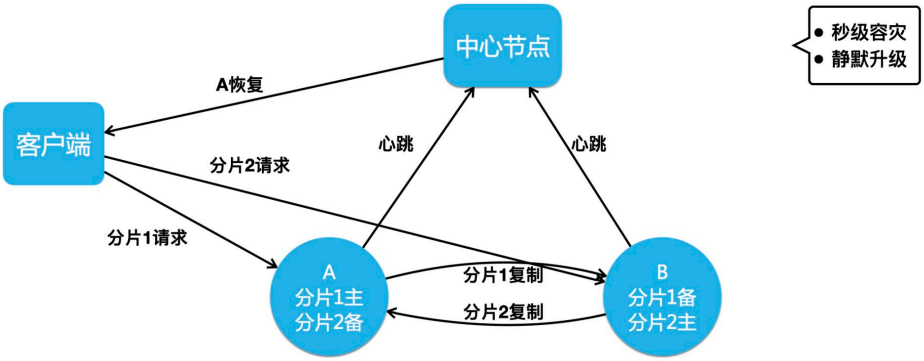
介绍完整体的架构，我们看一下 Cellar 怎么做节点容灾。一个集群节点的宕机一般是临时的，一个节点的网络抖动也是临时的，它们会很快地恢复，并重新加入集群。因为节点的临时离开就把它彻底摘除，并做数据副本补全操作，会消耗大量资源，进而影响到业务请求。所以，我们实现了 Handoff 机制来解决这种节点短时故障带来的影响。



如上图所示，如果 A 节点宕机了，会触发 Handoff 机制，这时候中心节点会通知客户端 A 节点发生了故障，让客户端把分片 1 的请求也打到 B 上。B 节点正常处理完客户端的读写请求之后，还会把本应该写入 A 节点的分片 1&2 数据写入到本地的 Log 中。



如果 A 节点宕机后 3~5 分钟，或者网络抖动 30~50 秒之后恢复了，A 节点就会上报心跳到中心节点，中心节点就会通知 B 节点：“ A 节点恢复了，你去把它不在期间的数据传给它。”这时候，B 节点就会把本地存储的 Log 回写到 A 节点上。等到 A 节点拥有了故障期间的全量数据之后，中心节点就会告诉客户端，A 节点已经彻底恢复了，客户端就可以重新把分片 1 的请求打回 A 节点。

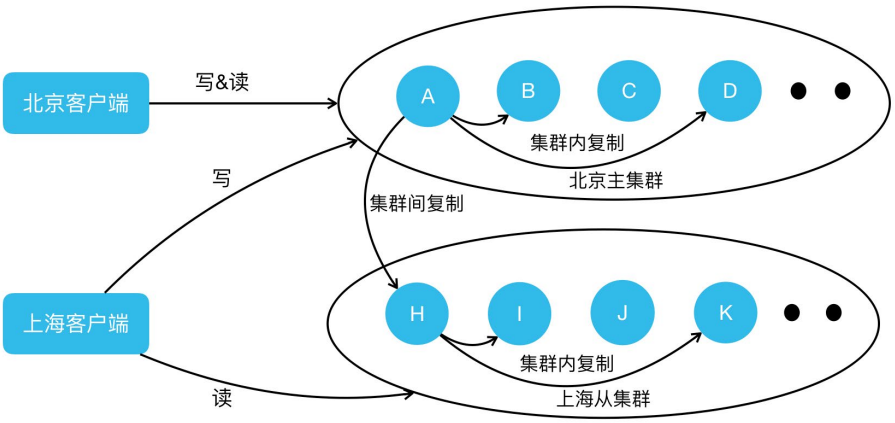


通过这样的操作，我们可以做到秒级的快速节点摘除，而且节点恢复后加回，只需补齐少量的增量数据。另外如果 A 节点要做升级，中心节点先通过主动 Handoff 把 A 节点流量切到 B 节点，A 升级后再回写增量 Log，然后切回流量加入集群。这样通

过主动触发 Handoff 机制，我们就实现了静默升级的功能。

Cellar 跨地域容灾

下面我介绍一下 Cellar 跨地域容灾是怎么做的。Cellar 跟 Squirrel 面对的跨地域容灾问题是一样的，解决方案同样也是集群间复制。以下图一个北京主集群、上海从集群的跨地域场景为例，比如说客户端的写操作到了北京的主集群 A 节点，A 节点会像正常集群内复制一样，把它复制到 B 和 D 节点上。同时 A 节点还会把数据复制一份到从集群的 H 节点。H 节点处理完集群间复制写入之后，它也会做从集群内的复制，把这个写操作复制到从集群的 I 、K 节点上。通过在主从集群的节点间建立这样一个复制链路，我们完成了集群间的数据复制，并且这个复制保证了最低的跨地域带宽占用。同样的，集群间的两个节点通过配置两个双向复制的链路，就可以达到双向同步异地多活的效果。

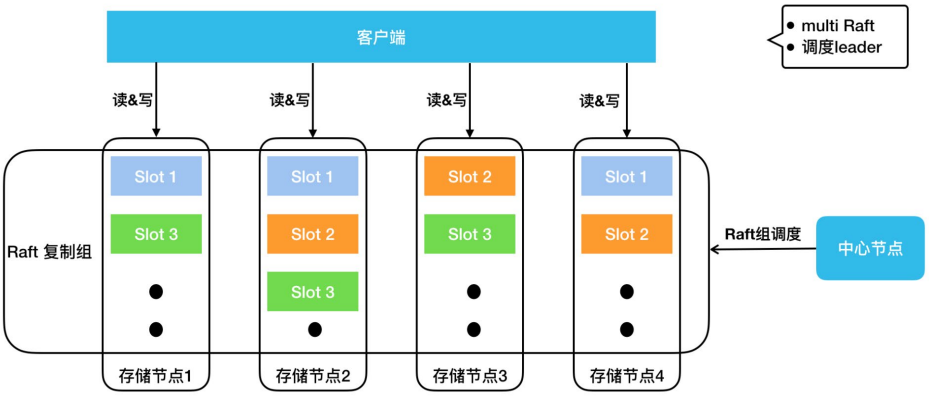


Cellar 强一致

我们做好了节点容灾以及跨地域容灾后，业务又对我们提出了更高要求：强一致存储。我们之前的数据复制是异步的，在做故障摘除时，可能因为故障节点数据还没复制出来，导致数据丢失。但是对于金融支付等场景来说，它们是不容许数据丢失的。

面对这个难题，我们该怎么解决？目前业界主流的解决方案是基于 Paxos 或 Raft 协议的强一致复制。我们最终选择了 Raft 协议。主要是因为 Raft 论文是非常详实的，是一篇工程化程度很高的论文。业界也有不少比较成熟的 Raft 开源实现，可以作为我们研发的基础，进而能够缩短研发周期。

下图是现在 Cellar 集群 Raft 复制模式下的架构图，中心节点会做 Raft 组的调度，它会决定每一个 Slot 的三副本存在哪些节点上。

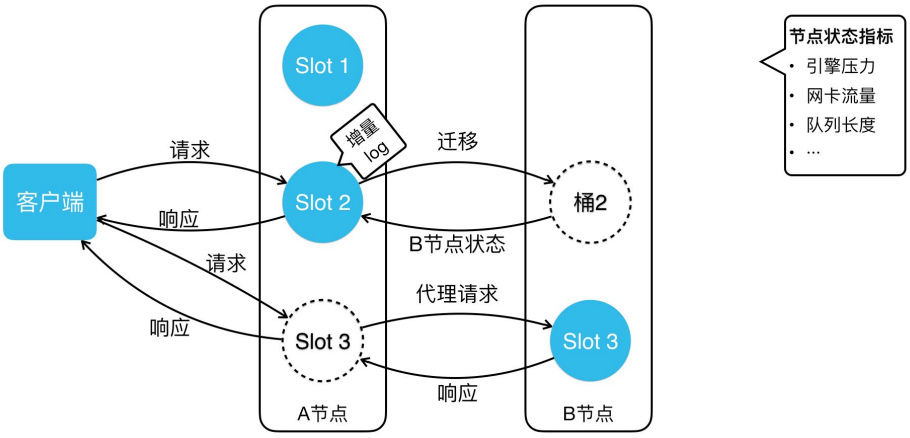


大家可以看到 Slot 1 在存储节点 1、2、4 上，Slot 2 在存储节点 2、3、4 上。每个 Slot 组成一个 Raft 组，客户端会去 Raft Leader 上进行读写。由于我们是预分配了 16384 个 Slot，所以，在集群规模很小的时候，我们的存储节点上可能会有数百甚至上千个 Slot 。这时候如果每个 Raft 复制组都有自己的复制线程、复制请求和 Log 等，那么资源消耗会非常大，写入性能会很差。所以我们做了 Multi Raft 实现，Cellar 会把同一个节点上所有的 Raft 复制组写一份 Log，用同一组线程去做复制，不同 Raft 组间的复制包也会按照目标节点做整合，以保证写入性能不会因 Raft 组过多而变差。 Raft 内部其实是有自己的选主机制，它可以控制自己的主节点，如果有任何节点宕机，它可以通过选举机制选出新的主节点。那么，中心节点是不是就不需要管理 Raft 组了吗？不是的。这里讲一个典型的场景，如果一个集群的部分节点经过几轮宕机恢复的过程，Raft Leader 在存储节点之间会变得极其不均。而为了保证数据的强一致，客户端的读写流量又必须发到 Raft Leader，这时候集群的

节点流量会很不均衡。所以我们的中心节点还会做 Raft 组的 Leader 调度。比如说 Slot 1 存储在节点 1、2、4，并且节点 1 是 Leader。如果节点 1 挂了，Raft 把节点 2 选成了 Leader。然后节点 1 恢复了并重新加入集群，中心节点这时会让节点 2 把 Leader 还给节点 1 。这样，即便经过一系列宕机和恢复，我们存储节点之间的 Leader 数目仍然能保证是均衡的。

接下来，我们看一下 Cellar 如何保证它的端到端高成功率。这里也讲三个影响成功率的问题。Cellar 遇到的数据迁移和热点 Key 问题与 Squirrel 是一样的，但解决方案不一样。这是因为 Cellar 走的是自研路径，不用考虑与官方版本的兼容性，对架构改动更大些。另一个问题是慢请求阻塞服务队列导致大面积超时，这是 Cellar 网络、工作多线程模型设计下会遇到的不同问题。

Cellar 智能迁移



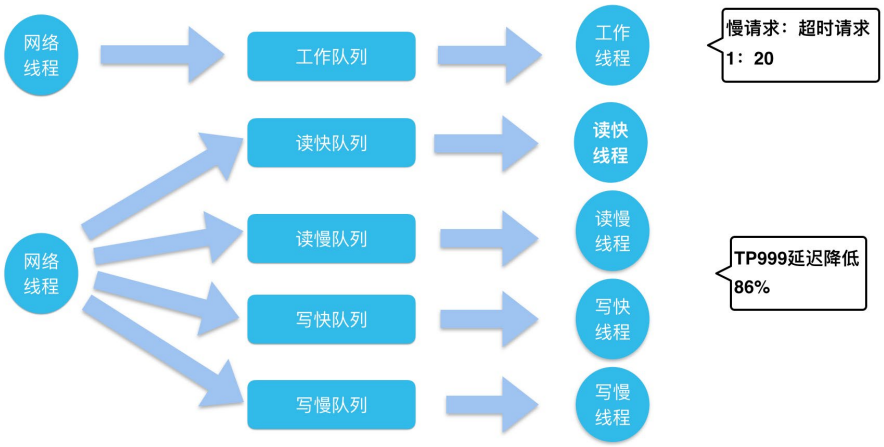
上图是 Cellar 智能迁移架构图。我们把桶的迁移分成了三个状态。第一个状态就是正常的状态，没有任何迁移。如果这时候要把 Slot 2 从 A 节点迁移到 B 节点，A 会给 Slot 2 打一个快照，然后把这个快照全量发到 B 节点上。在迁移数据的时候，B 节点的回包会带回 B 节点的状态。B 的状态包括什么？引擎的压力、网卡流量、队列长度等。A 节点会根据 B 节点的状态调整自己的迁移速度。像 Squirrel 一样，它经

过一段时间调整后，迁移速度会达到一个动态平衡，达到最快速的迁移，同时又尽可能小地影响业务的正常请求。

当 Slot 2 迁移完后，会进入图中 Slot 3 的状态。客户端这时可能还没更新路由表，当它请求到了 A 节点，A 节点会发现客户端请求错了节点，但它不会返回错误，它会把请求代理到 B 节点上，然后把 B 的响应包再返回客户端。同时它会告诉客户端，需要更新一下路由表了，此后客户端就能直接访问到 B 节点。这样就解决了客户端路由更新延迟造成的请求错误。

Cellar 快慢列队

下图上方是一个标准的线程队列模型。网络线程池接收网络流量解析出请求包，然后把请求放到工作队列里，工作线程池会从工作队列取请求来处理，然后把响应包放回网络线程池发出。

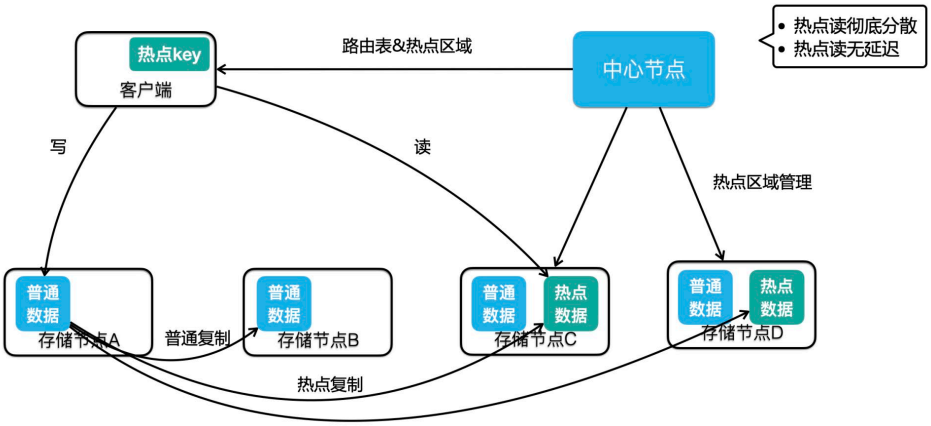


我们分析线上发生的超时案例时发现，一批超时请求当中往往只有一两个请求是引擎处理慢导致的，大部分请求，只是因为在队列等待过久导致整体响应时间过长而超时了。从线上分析来看，真正的慢请求占超时请求的比例只有 1/20。

我们的解法是什么样？很简单，拆线程池、拆队列。我们的网络线程在收到包之后，

会根据它的请求特点，是读还是写，快还是慢，分到四个队列里。读写请求比较好区分，但快慢怎么分开？我们会根据请求的 Key 个数、Value 大小、数据结构元素数等对请求进行快慢区分。然后用对应的四个工作线程池处理对应队列的请求，就实现了快慢读写请求的隔离。这样如果我有一个读的慢请求，不会影响另外三种请求的正常处理。不过这样也会带来一个问题，我们的线程池从一个变成四个，那线程数是不是变成原来的四倍？其实并不是的，我们某个线程池空闲的时候会去帮助其它的线程池处理请求。所以，我们线程池变成了四个，但是线程总数并没有变。我们线上验证中这样的设计能把服务 TP999 的延迟降低 86%，可大幅降低超时率。

Cellar 热点 Key



上图是 Cellar 热点 Key 解决方案的架构图。我们可以看到中心节点加了一个职责，多了热点区域管理，它现在不只负责正常的数据副本分布，还要管理热点数据的分布，图示这个集群在节点 C、D 放了热点区域。我们通过读写流程看一下这个方案是怎么运转的。如果客户端有一个写操作到了 A 节点，A 节点处理完成后，会根据实时的热点统计结果判断写入的 Key 是否为热点。如果这个 Key 是一个热点，那么它会在做集群内复制的同时，还会把这个数据复制有热点区域的节点，也就是图中的 C、D 节点。同时，存储节点在返回结果给客户端时，会告诉客户端，这个 Key 是热点，这时客户端内会缓存这个热点 Key。当客户端有这个 Key 的读请求时，它就会直接

去热点区域做数据的读取。通过这样的方式，我们可以做到只对热点数据做扩容，不像 Squirrel ，要把整个 Slot 迁出来做扩容。有必要的话，中心节点也可以把热点区域放到集群的所有节点上，所有的热点读请求就能均衡的分到所有节点上。另外，通过这种实时的热点数据复制，我们很好地解决了类似客户端缓存热点 KV 方案造成的一致性问题。

发展规划和业界趋势

最后，一起来看看我们项目的规划和业界的技术趋势。这部分内容会按照服务、系统、硬件三层来进行阐述。首先在服务层，主要有三点：

- 第一，Redis Gossip 协议优化。大家都知道 Gossip 协议在集群的规模变大之后，消息量会剧增，它的 Failover 时间也会变得越来越长。所以当集群规模达到 TB 级后，集群的可用性会受到很大的影响，所以我们后面会重点在这方面做一些优化。
- 第二，我们已经在 Cellar 存储节点的数据副本间做了 Raft 复制，可以保证数据强一致，后面我们会在 Cellar 的中心点内部也做一个 Raft 复制，这样就不用依赖于 ZooKeeper 做分布式仲裁、元数据存储了，我们的架构也会变得更加简单、可靠。
- 第三，Squirrel 和 Cellar 虽然都是 KV 存储，但是因为它们是基于不同的开源项目研发的，所以 API 和访问协议不同，我们之后会考虑将 Squirrel 和 Cellar 在 SDK 层做整合，虽然后端会有不同的存储集群，但业务侧可以用一套 SDK 进行访问。

在系统层面，我们正在调研并去落地一些 Kernel Bypass 技术，像 DPDK、SPDK 这种网络和硬盘的用户态 IO 技术。它可以绕过内核，通过轮询机制访问这些设备，可以极大提升系统的 IO 能力。存储作为 IO 密集型服务，性能会获得大幅的提升。

在硬件层面，像支持 RDMA 的智能网卡能大幅降低网络延迟和提升吞吐；还有像 3D XPoint 这样的闪存技术，比如英特尔新发布的 AEP 存储，其访问延迟已经比较接近

内存了，以后闪存跟内存之间的界限也会变得越来越模糊；最后，看一下计算型硬件，比如通过在闪存上加 FPGA 卡，把原本应该 CPU 做的工作，像数据压缩、解压等，下沉到卡上执行，这种硬件能在解放 CPU 的同时，也可以降低服务的响应延迟。

作者简介

泽斌，美团点评高级技术专家，2014 年加入美团。

招聘信息

美团基础技术部存储技术中心长期招聘 C/C++、Go、Java 高级 / 资深工程师和技术专家，欢迎加入美团基础技术部大家庭。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：基础技术部 – 存储技术中心）

Java 中 9 种常见的 CMS GC 问题分析与解决

作者：新宇　湘铭　祥璞

1. 写在前面

| 本文主要针对 Hotspot VM 中“CMS + ParNew”组合的一些使用场景进行总结。重点通过部分源码对根因进行分析以及对排查方法进行总结，排查过程会省略较多，另外本文专业术语较多，有一定的阅读门槛，如未介绍清楚，还请自行查阅相关材料。

| 总字数 2 万左右（不包含代码片段），整体阅读时间约 30min ，文章较长，可以选择你感兴趣的场景进行研究。

1.1 引言

自 Sun 发布 Java 语言以来，开始使用 GC 技术来进行内存自动管理，避免了手动管理带来的悬挂指针（Dangling Pointer）问题，很大程度上提升了开发效率，从此 GC 技术也一举成名。GC 有着非常悠久的历史，1960 年有着“Lisp 之父”和“人工智能之父”之称的 John McCarthy 就在论文中发布了 GC 算法，60 年以来，GC 技术的发展也突飞猛进，但不管是多么前沿的收集器也都是基于三种基本算法的组合或应用，也就是说 GC 要解决的根本问题这么多年一直都没有变过。笔者认为，在不太远的将来，GC 技术依然不会过时，比起日新月异的新技术，GC 这门古典技术更值得我们学习。

目前，互联网上 Java 的 GC 资料要么是主要讲解理论，要么就是针对单一场景的 GC 问题进行了剖析，对整个体系总结的资料少之又少。前车之鉴，后事之师，美团的几位工程师搜集了内部各种 GC 问题的分析文章，并结合个人的理解做了一些总结，希望能起到“抛砖引玉”的作用，文中若有错误之处，还请大家不吝指正。

GC 问题处理能力能不能系统性掌握？一些影响因素都是**互为因果**的问题该怎么分析？比如一个服务 RT 突然上涨，有 GC 耗时增大、线程 Block 增多、慢查询增多、CPU 负载高四个表象，到底哪个是诱因？如何判断 GC 有没有问题？使用 CMS 有哪些常见问题？如何判断根因是什么？如何解决或避免这些问题？阅读完本文，相信你将会对 CMS GC 的问题处理有一个系统性的认知，更能游刃有余地解决这些问题，下面就让我们开始吧！

1.2 概览

想要系统性地掌握 GC 问题处理，笔者这里给出一个学习路径，整体文章的框架也是按照这个结构展开，主要分四大步。



- **建立知识体系：**从 JVM 的内存结构到垃圾收集的算法和收集器，学习 GC 的基础知识，掌握一些常用的 GC 问题分析工具。
- **确定评价指标：**了解基本 GC 的评价方法，摸清如何设定独立系统的指标，以及在业务场景中判断 GC 是否存在问题的手段。
- **场景调优实践：**运用掌握的知识和系统评价指标，分析与解决九种 CMS 中常见 GC 问题场景。
- **总结优化经验：**对整体过程做总结并提出笔者的几点建议，同时将总结到的经验完善到知识体系之中。

2. GC 基础

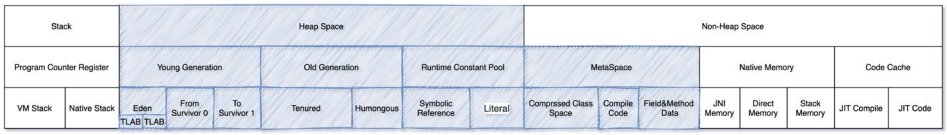
在正式开始前，先做些简要铺垫，介绍下 JVM 内存划分、收集算法、收集器等常用概念介绍，基础比较好的同学可以直接跳过这部分。

2.1 基础概念

- **GC**：GC 本身有三种语义，下文需要根据具体场景带入不同的语义：
  - **Garbage Collection**：垃圾收集技术，名词。
  - **Garbage Collector**：垃圾收集器，名词。
  - **Garbage Collecting**：垃圾收集动作，动词。
- **Mutator**：生产垃圾的角色，也就是我们的应用程序，垃圾制造者，通过 Allocator 进行 allocate 和 free。
- **TLAB**：Thread Local Allocation Buffer 的简写，基于 CAS 的独享线程（Mutator Threads）可以优先将对象分配在 Eden 中的一块内存，因为是 Java 线程独享的内存区没有锁竞争，所以分配速度更快，每个 TLAB 都是一个线程独享的。
- **Card Table**：中文翻译为卡表，主要是用来标记卡页的状态，每个卡表项对应一个卡页。当卡页中一个对象引用有写操作时，写屏障将会标记对象所在的卡表状态改为 dirty，卡表的本质是用来解决跨代引用的问题。具体怎么解决的可以参考 StackOverflow 上的这个问题 how-actually-card-table-and-writer-barrier-works，或者研读一下 cardTableRS.app 中的源码。

2.2 JVM 内存划分

从 JCP（Java Community Process）的官网中可以看到，目前 Java 版本最新已经到了 Java 16，未来的 Java 17 以及现在的 Java 11 和 Java 8 是 LTS 版本，JVM 规范也在随着迭代在变更，由于本文主要讨论 CMS，此处还是放 Java 8 的内存结构。



GC 主要工作在 Heap 区和 MetaSpace 区（上图蓝色部分），在 Direct Memory 中，如果使用的是 DirectByteBuffer，那么在分配内存不够时则是 GC 通过 `Cleaner#-`

`clean` 间接管理。

任何自动内存管理系统都会面临的步骤：为新对象分配空间，然后收集垃圾对象空间，下面我们就展开介绍一下这些基础知识。

2.3 分配对象

Java 中对象地址操作主要使用 Unsafe 调用了 C 的 allocate 和 free 两个方法，分配方法有两种：

- **空闲链表（free list）：** 通过额外的存储记录空闲的地址，将随机 IO 变为顺序 IO，但带来了额外的空间消耗。
- **碰撞指针（bump pointer）：** 通过一个指针作为分界点，需要分配内存时，仅需把指针往空闲的一端移动与对象大小相等的距离，分配效率较高，但使用场景有限。

2.4 收集对象

2.4.1 识别垃圾

- **引用计数法（Reference Counting）：** 对每个对象的引用进行计数，每当有一个地方引用它时计数器 +1、引用失效则 −1，引用的计数放到对象头中，大于 0 的对象被认为是存活对象。虽然循环引用的问题可通过 Recycler 算法解决，但是在多线程环境下，引用计数变更也要进行昂贵的同步操作，性能较低，早期的编程语言会采用此算法。
- **可达性分析，又称引用链法（Tracing GC）：** 从 GC Root 开始进行对象搜索，可以被搜索到的对象即为可达对象，此时还不足以判断对象是否存活 / 死亡，需要经过多次标记才能更加准确地确定，整个连通图之外的对象便可以作为垃圾被回收掉。目前 Java 中主流的虚拟机均采用此算法。

备注：引用计数法是可以处理循环引用问题的，下次面试时不要再这么说啦 ~ ~

2.4.2 收集算法

自从有自动内存管理出现之时就有的一些收集算法，不同的收集器也是在不同场景下进行组合。

- **Mark-Sweep（标记 - 清除）：** 回收过程主要分为两个阶段，第一阶段为追踪（Tracing）阶段，即从 GC Root 开始遍历对象图，并标记（Mark）所遇到的每个对象，第二阶段为清除（Sweep）阶段，即回收器检查堆中每一个对象，并将所有未被标记的对象进行回收，整个过程不会发生对象移动。整个算法在不同的实现中会使用三色抽象（Tricolour Abstraction）、位图标记（BitMap）等技术来提高算法的效率，存活对象较多时较高效。
- **Mark-Compact（标记 - 整理）：** 这个算法的主要目的就是解决在非移动式回收器中都会存在的碎片化问题，也分为两个阶段，第一阶段与 Mark-Sweep 类似，第二阶段则会对存活对象按照整理顺序（Compaction Order）进行整理。主要实现有双指针（Two-Finger）回收算法、滑动回收（Lisp2）算法和引线整理（Threaded Compaction）算法等。
- **Copying（复制）：** 将空间分为两个大小相同的 From 和 To 两个半区，同一时间只会使用其中一个，每次进行回收时将一个半区的存活对象通过复制的方式转移到另一个半区。有递归（Robert R. Fenichel 和 Jerome C. Yochelson 提出）和迭代（Cheney 提出）算法，以及解决了前两者递归栈、缓存行等问题的近似优先搜索算法。复制算法可以通过碰撞指针的方式进行快速地分配内存，但是也存在着空间利用率不高的缺点，另外就是存活对象比较大时复制的成本比较高。

三种算法在是否移动对象、空间和时间方面的一些对比，假设存活对象数量为 *L*、堆空间大小为 *H*，则：

	移动对象	空间开销	时间开销
Mark-Sweep	否	低（有碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，sweep 阶段与整堆大小成正比 *O(H)*
Mark-Compact	是	低（无碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，compaction 阶段与存活对象的大小成正比 *O(L)*
Copying	是	高	与存活对象大小成正比 *O(L)*

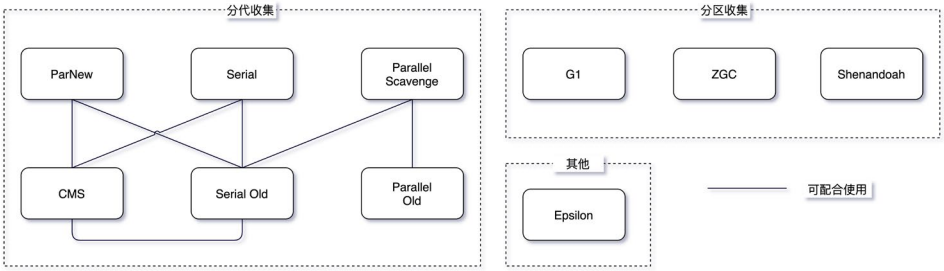
把 mark、sweep、compaction、copying 这几种动作的耗时放在一起看，大致有这样的关系：

$$\begin{cases} compaction \geq copying > mark > sweep \\ mark + sweep > copying \end{cases}$$

虽然 compaction 与 copying 都涉及移动对象，但取决于具体算法，compaction 可能要先计算一次对象的目标地址，然后修正指针，最后再移动对象。copying 则可以把这几件事情合为一体来做，所以可以快一些。另外，还需要留意 GC 带来的开销不能只看 Collector 的耗时，还得看 Allocator 。如果能保证内存没碎片，分配就可以用 pointer bumping 方式，只需要挪一个指针就完成了分配，非常快。而如果内存有碎片就得用 freelist 之类的方式管理，分配速度通常会慢一些。

2.5 收集器

目前在 Hotspot VM 中主要有分代收集和分区收集两大类，具体可以看下面的这个图，不过未来会逐渐向分区收集发展。在美团内部，有部分业务尝试用了 ZGC（感兴趣的同学可以学习下这篇文章 新一代垃圾回收器 ZGC 的探索与实践），其余基本都停留在 CMS 和 G1 上。另外在 JDK11 后提供了一个不执行任何垃圾回收动作的回收器 Epsilon（A No-Op Garbage Collector）用作性能分析。另外一个就是 Azul 的 Zing JVM，其 C4（Concurrent Continuously Compacting Collector）收集器也在业内有一定的影响力。



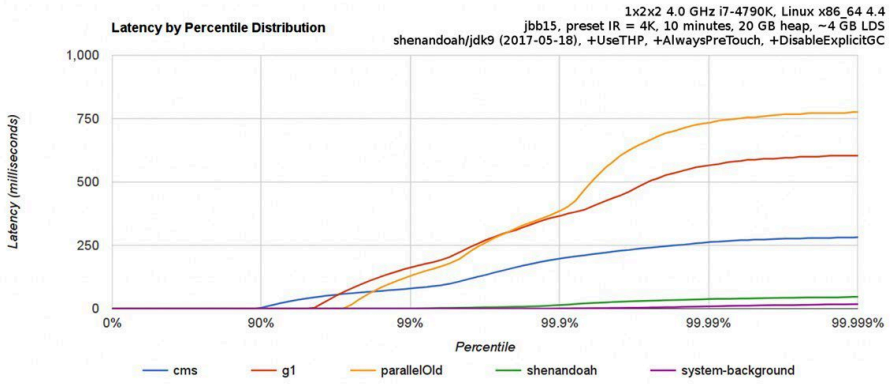
备注：值得一提的是，早些年国内 GC 技术的布道者 RednaxelaFX（江湖人称 R 大）也曾就职于 Azul，本文的一部分材料也参考了他的一些文章。

2.5.1 分代收集器

- **ParNew：**一款多线程的收集器，采用复制算法，主要工作在 Young 区，可以通过 `-XX:ParallelGCThreads` 参数来控制收集的线程数，整个过程都是 STW 的，常与 CMS 组合使用。
- **CMS：**以获取最短回收停顿时间为目标，采用“标记－清除”算法，分 4 大步进行垃圾收集，其中初始标记和重新标记会 STW ，多数应用于互联网站或者 B/S 系统的服务器端上，JDK9 被标记弃用，JDK14 被删除，详情可见 JEP 363。

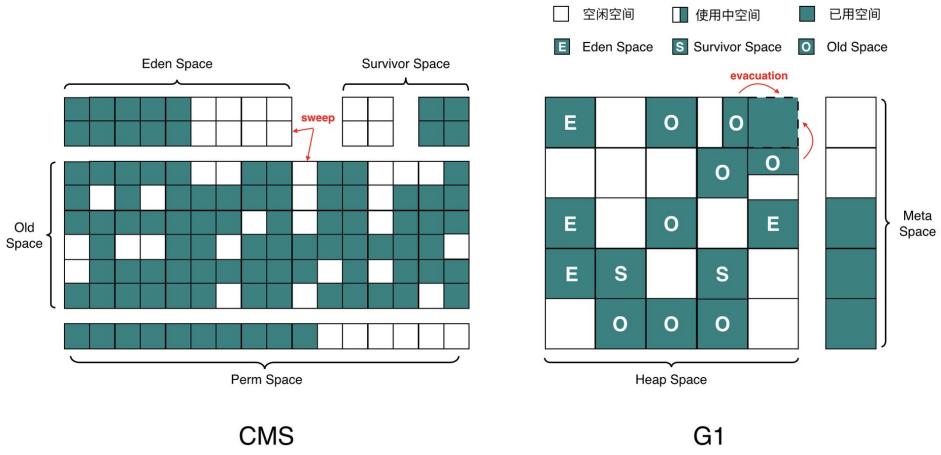
2.5.2 分区收集器

- **G1：**一种服务器端的垃圾收集器，应用在多处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能地满足垃圾收集暂停时间的要求。
- **ZGC：**JDK11 中推出的一款低延迟垃圾回收器，适用于大内存低延迟服务的内存管理和回收，SPECjbb 2015 基准测试，在 128G 的大堆下，最大停顿时间才 1.68 ms，停顿时间远胜于 G1 和 CMS。
- **Shenandoah：**由 Red Hat 的一个团队负责开发，与 G1 类似，基于 Region 设计的垃圾收集器，但不需要 Remember Set 或者 Card Table 来记录跨 Region 引用，停顿时间和堆的大小没有任何关系。停顿时间与 ZGC 接近，下图为与 CMS 和 G1 等收集器的 benchmark。



2.5.3 常用收集器

目前使用最多的是 CMS 和 G1 收集器，二者都有分代的概念，主要内存结构如下：



2.5.4 其他收集器

以上仅列出常见收集器，除此之外还有很多，如 Metronome、Stopless、Staccato、Chicken、Clover 等实时回收器，Sapphire、Compressor、Pauseless 等并发复制 / 整理回收器，Doligez-Leroy-Conthier 等标记整理回收器，由于篇幅原因，不在此一一介绍。

2.6 常用工具

工欲善其事，必先利其器，此处列出一些笔者常用的工具，具体情况大家可以自由选择，本文的问题都是使用这些工具来定位和分析的。

2.6.1 命令行终端

- 标准终端类：jps、jinfo、jstat、jstack、jmap
- 功能整合类：jcmd、vjtools、arthas、greys

2.6.2 可视化界面

- 简易：JConsole、JVisualvm、HA、GCHisto、GCViewer
- 进阶：MAT、JProfiler

命令行推荐 arthas ，可视化界面推荐 JProfiler，此外还有一些在线的平台 gceasy、heaphero、fastthread ，美团内部的 Scalpel（一款自研的 JVM 问题诊断工具，暂时未开源）也比较好用。

3. GC 问题判断

在做 GC 问题排查和优化之前，我们需要先来明确下到底是不是 GC 直接导致的问题，或者应用代码导致的 GC 异常，最终出现问题。

3.1 判断 GC 有没有问题?

3.1.1 设定评价标准

评判 GC 的两个核心指标：

- **延迟（Latency）：** 也可以理解为最大停顿时间，即垃圾收集过程中一次 STW 的最长时间，越短越好，一定程度上可以接受频次的增大，GC 技术的主要发展方向。
- **吞吐量（Throughput）：** 应用系统的生命周期内，由于 GC 线程会占用 Mutator 当前可用的 CPU 时钟周期，吞吐量即为 Mutator 有效花费的时间占

系统总运行时间的百分比，例如系统运行了 100 min，GC 耗时 1 min，则系统吞吐量为 99%，吞吐量优先的收集器可以接受较长的停顿。

目前各大互联网公司的系统基本都更追求低延时，避免一次 GC 停顿的时间过长对用户体验造成损失，衡量指标需要结合一下应用服务的 SLA，主要如下两点来判断：

$$t_{stw} \le t_{tp9999}$$ （t_{stw}为单次停顿时间，t_{tp9999}为应用4个9耗时）

$$(1 - \frac{\sum_{k=1}^{n} c_k \bar{t}_k}{T_n}) \times 100\% \ge 99.99\%$$ （c_k为一个时间周期内gc次数，$\bar{t}_k$为该时间周期平均停顿时间，T_n为总时间）

简而言之，即为**一次停顿的时间不超过应用服务的 TP9999，GC 的吞吐量不小于 99.99%**。举个例子，假设某个服务 A 的 TP9999 为 80 ms，平均 GC 停顿为 30 ms，那么该服务的最大停顿时间最好不要超过 80 ms，GC 频次控制在 5 min 以上一次。如果满足不了，那就需要调优或者通过更多资源来进行并联冗余。（大家可以先停下来，看看监控平台上面的 gc.meantime 分钟级别指标，如果超过了 6 ms 那单机 GC 吞吐量就达不到 4 个 9 了。）

备注：除了这两个指标之外还有 Footprint（资源量大小测量）、反应速度等指标，互联网这种实时系统追求低延迟，而很多嵌入式系统则追求 Footprint。

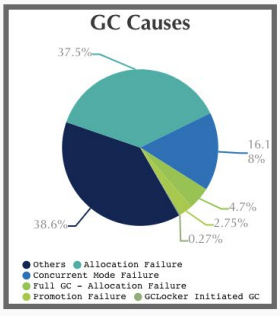
3.1.2 读懂 GC Cause

拿到 GC 日志，我们就可以简单分析 GC 情况了，通过一些工具，我们可以比较直观地看到 Cause 的分布情况，如下图就是使用 gceasy 绘制的图表：

GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Others	3	n/a	n/a	1 min 55 sec 640 ms	38.6%
Allocation Failure	1638	68.6 ms	18 sec 200 ms	1 min 52 sec 340 ms	37.5%
Concurrent Mode Failure	8	6 sec 59 ms	11 sec 960 ms	48 sec 470 ms	16.18%
Full GC - Allocation Failure	5	2 sec 816 ms	3 sec 100 ms	14 sec 80 ms	4.7%
Promotion Failure	1	8 sec 240 ms	8 sec 240 ms	8 sec 240 ms	2.75%
GCLocker Initiated GC	13	62.3 ms	130 ms	810 ms	0.27%
Total	1668	n/a	n/a	4 min 59 sec 580 ms	100.0%



如上图所示，我们很清晰的就能知道是什么原因引起的 GC，以及每次的时间花费情况，但是要分析 GC 的问题，先要读懂 GC Cause，即 JVM 什么样的条件下选择进行 GC 操作，具体 Cause 的分类可以看一下 Hotspot 源码：src/share/vm/gc/shared/gcCause.hpp 和 src/share/vm/gc/shared/gcCause.cpp 中。

```
const char* GCCause::to_string(GCCause::Cause cause) {
  switch (cause) {
    case _java_lang_system_gc:
      return "System.gc()";

    case _full_gc_alot:
      return "FullGCAlot";

    case _scavenge_alot:
      return "ScavengeAlot";

    case _allocation_profiler:
      return "Allocation Profiler";

    case _jvmti_force_gc:
      return "JvmtiEnv ForceGarbageCollection";

    case _gc_locker:
      return "GCLocker Initiated GC";

    case _heap_inspection:
      return "Heap Inspection Initiated GC";

    case _heap_dump:
      return "Heap Dump Initiated GC";

    case _wb_young_gc:
      return "WhiteBox Initiated Young GC";

    case _wb_conc_mark:
      return "WhiteBox Initiated Concurrent Mark";

    case _wb_full_gc:
      return "WhiteBox Initiated Full GC";

    case _no_gc:
      return "No GC";

    case _allocation_failure:
      return "Allocation Failure";
```

```
    case _tenured_generation_full:
      return "Tenured Generation Full";

    case _metadata_GC_threshold:
      return "Metadata GC Threshold";

    case _metadata_GC_clear_soft_refs:
      return "Metadata GC Clear Soft References";

    case _cms_generation_full:
      return "CMS Generation Full";

    case _cms_initial_mark:
      return "CMS Initial Mark";

    case _cms_final_remark:
      return "CMS Final Remark";

    case _cms_concurrent_mark:
      return "CMS Concurrent Mark";

    case _old_generation_expanded_on_last_scavenge:
      return "Old Generation Expanded On Last Scavenge";

    case _old_generation_too_full_to_scavenge:
      return "Old Generation Too Full To Scavenge";

    case _adaptive_size_policy:
      return "Ergonomics";

    case _g1_inc_collection_pause:
      return "G1 Evacuation Pause";

    case _g1_humongous_allocation:
      return "G1 Humongous Allocation";

    case _dcmd_gc_run:
      return "Diagnostic Command";

    case _last_gc_cause:
      return "ILLEGAL VALUE - last gc cause - ILLEGAL VALUE";

    default:
      return "unknown GCCause";
  }
  ShouldNotReachHere();
}
```

重点需要关注的几个 GC Cause：

- **System.gc()：**手动触发 GC 操作。
- **CMS：**CMS GC 在执行过程中的一些动作，重点关注 CMS Initial Mark 和 CMS Final Remark 两个 STW 阶段。
- **Promotion Failure：**Old 区没有足够的空间分配给 Young 区晋升的对象（即使总可用内存足够大）。
- **Concurrent Mode Failure：**CMS GC 运行期间，Old 区预留的空间不足以分配给新的对象，此时收集器会发生退化，严重影响 GC 性能，下面的一个案例即为这种场景。
- **GCLocker Initiated GC：**如果线程执行在 JNI 临界区时，刚好需要进行 GC，此时 GC Locker 将会阻止 GC 的发生，同时阻止其他线程进入 JNI 临界区，直到最后一个线程退出临界区时触发一次 GC。

什么时机使用这些 Cause 触发回收，大家可以看一下 CMS 的代码，这里就不讨论了，具体在 /src/hotspot/share/gc/cms/concurrentMarkSweepGeneration.cpp 中。

```
bool CMSCollector::shouldConcurrentCollect() {
  LogTarget(Trace, gc) log;

  if (_full_gc_requested) {
    log.print("CMSCollector: collect because of explicit  gc request
(or GCLocker)");
    return true;
  }

  FreelistLocker x(this);
  // ------------------------------------------------------------------
--
  // Print out lots of information which affects the initiation of
  // a collection.
  if (log.is_enabled() && stats().valid()) {
    log.print("CMSCollector shouldConcurrentCollect: ");

    LogStream out(log);
    stats().print_on(&out);
```

```
    log.print("time_until_cms_gen_full %3.7f", stats().time_until_cms_
gen_full());
    log.print("free=" SIZE_FORMAT, _cmsGen->free());
    log.print("contiguous_available=" SIZE_FORMAT, _cmsGen->contiguous_
available());
    log.print("promotion_rate=%g", stats().promotion_rate());
    log.print("cms_allocation_rate=%g", stats().cms_allocation_rate());
    log.print("occupancy=%3.7f", _cmsGen->occupancy());
    log.print("initiatingOccupancy=%3.7f", _cmsGen->initiating_
occupancy());
    log.print("cms_time_since_begin=%3.7f", stats().cms_time_since_
begin());
    log.print("cms_time_since_end=%3.7f", stats().cms_time_since_
end());
    log.print("metadata initialized %d", MetaspaceGC::should_concurrent_
collect());
  }
  // ------------------------------------------------------------------
--

  // If the estimated time to complete a cms collection (cms_
duration())
  // is less than the estimated time remaining until the cms
generation
  // is full, start a collection.
  if (!UseCMSInitiatingOccupancyOnly) {
    if (stats().valid()) {
      if (stats().time_until_cms_start() == 0.0) {
        return true;
      }
    } else {

      if (_cmsGen->occupancy() >= _bootstrap_occupancy) {
        log.print(" CMSCollector: collect for bootstrapping
statistics: occupancy = %f, boot occupancy = %f",
                  _cmsGen->occupancy(), _bootstrap_occupancy);
        return true;
      }
    }
  }
  if (_cmsGen->should_concurrent_collect()) {
    log.print("CMS old gen initiated");
    return true;
  }

  CMSHeap* heap = CMSHeap::heap();
  if (heap->incremental_collection_will_fail(true /* consult_young */))
{
    log.print("CMSCollector: collect because incremental collection
will fail ");
```

```
    return true;
  }

  if (MetaspaceGC::should_concurrent_collect()) {
    log.print("CMSCollector: collect for metadata allocation ");
    return true;
  }

  // CMSTriggerInterval starts a CMS cycle if enough time has passed.
  if (CMSTriggerInterval >= 0) {
    if (CMSTriggerInterval == 0) {
      // Trigger always
      return true;
    }

    // Check the CMS time since begin (we do not check the stats
validity
    // as we want to be able to trigger the first CMS cycle as well)
    if (stats().cms_time_since_begin() >= (CMSTriggerInterval / ((double)
MILLIUNITS))) {
      if (stats().valid()) {
        log.print("CMSCollector: collect because of trigger interval
(time since last begin %3.7f secs)",
                  stats().cms_time_since_begin());
      } else {
        log.print("CMSCollector: collect because of trigger interval
(first collection)");
      }
      return true;
    }
  }

  return false;
}
```

3.2 判断是不是 GC 引发的问题?

到底是结果（现象）还是原因，在一次 GC 问题处理的过程中，如何判断是 GC 导致的故障，还是系统本身引发 GC 问题。这里继续拿在本文开头提到的一个 Case：“GC 耗时增大、线程 Block 增多、慢查询增多、CPU 负载高等四个表象，如何判断哪个是根因?”，笔者这里根据自己的经验大致整理了四种判断方法供参考：

- **时序分析：**先发生的事件是根因的概率更大，通过监控手段分析各个指标的异常时间点，还原事件时间线，如先观察到 CPU 负载高（要有足够的时间

Gap），那么整个问题影响链就可能是：CPU 负载高 -> 慢查询增多 -> GC 耗时增大 -> 线程 Block 增多 -> RT 上涨。

- **概率分析：**使用统计概率学，结合历史问题的经验进行推断，由近到远按类型分析，如过往慢查的问题比较多，那么整个问题影响链就可能是：慢查询增多 -> GC 耗时增大 -> CPU 负载高 -> 线程 Block 增多 -> RT 上涨。
- **实验分析：**通过故障演练等方式对问题现场进行模拟，触发其中部分条件（一个或多个），观察是否会发生问题，如只触发线程 Block 就会发生问题，那么整个问题影响链就可能是：线程 Block 增多 -> CPU 负载高 -> 慢查询增多 -> GC 耗时增大 -> RT 上涨。
- **反证分析：**对其中某一表象进行反证分析，即判断表象的发不发生跟结果是否有相关性，例如我们从整个集群的角度观察到某些节点慢查和 CPU 都正常，但也出了问题，那么整个问题影响链就可能是：GC 耗时增大 -> 线程 Block 增多 -> RT 上涨。

不同的根因，后续的分析方法是完全不同的。如果是 CPU 负载高那可能需要用火焰图看下热点、如果是慢查询增多那可能需要看下 DB 情况、如果是线程 Block 引起那可能需要看下锁竞争的情况，最后如果各个表象证明都没有问题，那可能 GC 确实存在问题，可以继续分析 GC 问题了。

3.3 问题分类导读

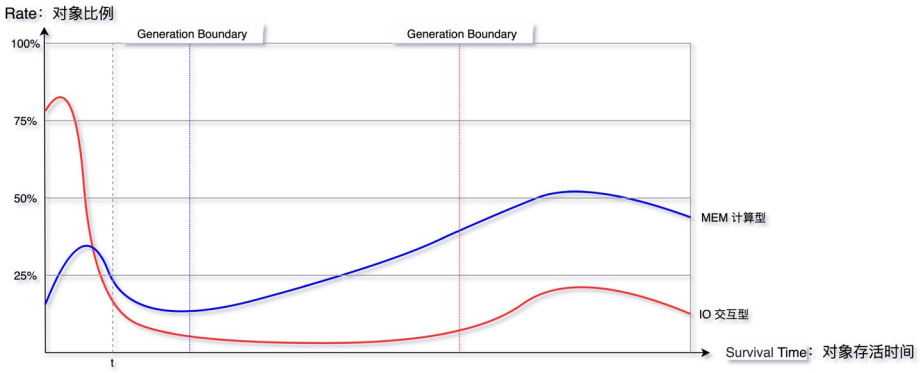
3.3.1 Mutator 类型

Mutator 的类型根据对象存活时间比例图来看主要分为两种，在弱分代假说中也提到类似的说法，如下图所示“Survival Time”表示对象存活时间，“Rate”表示对象分配比例：

- **IO 交互型：**互联网上目前大部分的服务都属于该类型，例如分布式 RPC、MQ、HTTP 网关服务等，对内存要求并不大，大部分对象在 TP9999 的时间内都会死亡，Young 区越大越好。

- **MEM 计算型：**主要是分布式数据计算 Hadoop，分布式存储 HBase、Cassandra，自建的分布式缓存等，对内存要求高，对象存活时间长，Old 区越大越好。

当然，除了二者之外还有介于两者之间的场景，本篇文章主要讨论第一种情况。对象 Survival Time 分布图，对我们设置 GC 参数有着非常重要的指导意义，如下图就可以简单推算分代的边界。



3.3.2 GC 问题分类

笔者选取了九种不同类型的 GC 问题，覆盖了大部分场景，如果有更好的场景，欢迎在评论区给出。

- **Unexpected GC：**意外发生的 GC，实际上不需要发生，我们可以通过一些手段去避免。
  - **Space Shock：**空间震荡问题，参见“场景一：动态扩容引起的空间震荡”。
  - **Explicit GC：**显示执行 GC 问题，参见“场景二：显式 GC 的去与留”。
- **Partial GC：**部分收集操作的 GC，只对某些分代 / 分区进行回收。
  - **Young GC：**分代收集里面的 Young 区收集动作，也可以叫做 Minor GC。
    - **ParNew：**Young GC 频繁，参见“场景四：过早晋升”。
  - **Old GC：**分代收集里面的 Old 区收集动作，也可以叫做 Major GC，有些也会叫做 Full GC，但其实这种叫法是不规范的，在 CMS 发生 Fore-

ground GC 时才是 Full GC，CMSScavengeBeforeRemark 参数也只是在 Remark 前触发一次 Young GC。

  - **CMS：** Old GC 频繁，参见“场景五：CMS Old GC 频繁”。
  - **CMS：** Old GC 不频繁但单次耗时大，参见“场景六：单次 CMS Old GC 耗时长”。
- **Full GC：** 全量收集的 GC，对整个堆进行回收，STW 时间会比较长，一旦发生，影响较大，也可以叫做 Major GC，参见“场景七：内存碎片 & 收集器退化”。
- **MetaSpace：** 元空间回收引发问题，参见“场景三：MetaSpace 区 OOM”。
- **Direct Memory：** 直接内存（也可以称作为堆外内存）回收引发问题，参见“场景八：堆外内存 OOM”。
- **JNI：** 本地 Native 方法引发问题，参见“场景九：JNI 引发的 GC 问题”。

3.3.3 排查难度

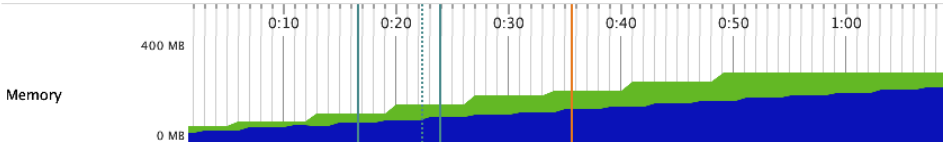
一个问题的**解决难度跟它的常见程度成反比**，大部分我们都可以通过各种搜索引擎找到类似的问题，然后用同样的手段尝试去解决。当一个问题在各种网站上都找不到相似的问题时，那么可能会有两种情况，一种这不是一个问题，另一种就是遇到一个隐藏比较深的问题，遇到这种问题可能就要深入到源码级别去调试了。以下 GC 问题场景，排查难度从上到下依次递增。

4. 常见场景分析与解决

4.1 场景一：动态扩容引起的空间震荡

4.1.1 现象

服务**刚刚启动时 GC 次数较多**，最大空间剩余很多但是依然发生 GC，这种情况我们可以通过观察 GC 日志或者通过监控工具来观察堆的空间变化情况即可。GC Cause 一般为 Allocation Failure，且在 GC 日志中会观察到经历一次 GC ，堆内各个空间的大小会被调整，如下图所示：



4.1.2 原因

在 JVM 的参数中 `-Xms` 和 `-Xmx` 设置的不一致，在初始化时只会初始 `-Xms` 大小的空间存储信息，每当空间不够用时再向操作系统申请，这样的话必然要进行一次 GC。具体是通过 `ConcurrentMarkSweepGeneration::compute_new_size()` 方法计算新的空间大小：

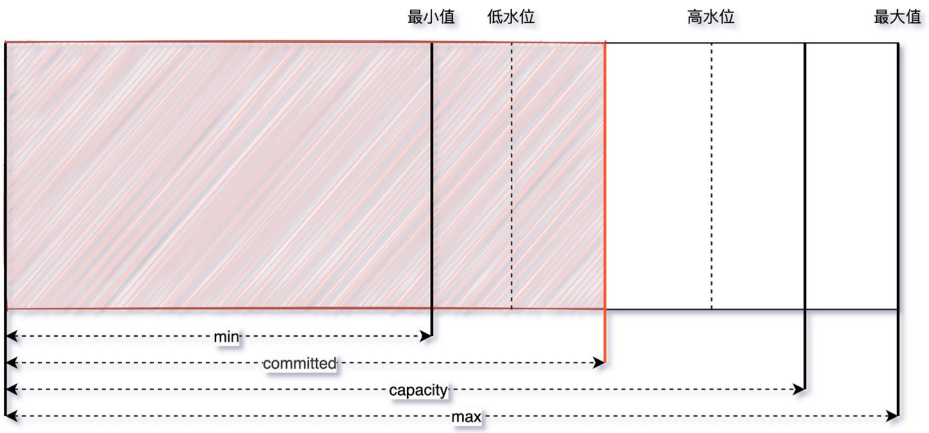
```
void ConcurrentMarkSweepGeneration::compute_new_size() {
  assert_locked_or_safepoint(Heap_lock);

  // If incremental collection failed, we just want to expand
  // to the limit.
  if (incremental_collection_failed()) {
    clear_incremental_collection_failed();
    grow_to_reserved();
    return;
  }

  // The heap has been compacted but not reset yet.
  // Any metric such as free() or used() will be incorrect.

  CardGeneration::compute_new_size();

  // Reset again after a possible resizing
  if (did_compact()) {
    cmsSpace()->reset_after_compaction();
  }
}
```

另外，如果空间剩余很多时也会进行缩容操作，JVM 通过 `-XX:MinHeapFreeRatio` 和 `-XX:MaxHeapFreeRatio` 来控制扩容和缩容的比例，调节这两个值也可以控制伸缩的时机，例如扩容便是使用 `GenCollectedHeap::expand_heap_and_allocate()` 来完成的，代码如下：

```
HeapWord* GenCollectedHeap::expand_heap_and_allocate(size_t size, bool
is_tlab) {
```

```
  HeapWord* result = NULL;
  if (_old_gen->should_allocate(size, is_tlab)) {
    result = _old_gen->expand_and_allocate(size, is_tlab);
  }
  if (result == NULL) {
    if (_young_gen->should_allocate(size, is_tlab)) {
      result = _young_gen->expand_and_allocate(size, is_tlab);
    }
  }
  assert(result == NULL || is_in_reserved(result), "result not in
heap");
  return result;
}
```

整个伸缩的模型理解可以看这个图，当 committed 的空间大小超过了低水位 / 高水位的大小，capacity 也会随之调整：



4.1.3 策略

定位：观察 CMS GC 触发时间点 Old/MetaSpace 区的 committed 占比是不是一个固定的值，或者像上文提到的观察总的内存使用率也可以。

解决：尽量**将成对出现的空间大小配置参数设置成固定的**，如 `-Xms` 和 `-Xmx`，`-XX:-MaxNewSize` 和 `-XX:NewSize`，`-XX:MetaSpaceSize` 和 `-XX:MaxMetaSpace-Size` 等。

4.1.4 小结

一般来说，我们需要保证 Java 虚拟机的堆是稳定的，确保 `-Xms` 和 `-Xmx` 设置的是一个值（即初始值和最大值一致），获得一个稳定的堆，同理在 MetaSpace 区也有类似的问题。不过在不追求停顿时间的情况下震荡的空间也是有利的，可以动态地伸缩以节省空间，例如作为富客户端的 Java 应用。

这个问题虽然初级，但是发生的概率还真不小，尤其是在一些规范不太健全的情况下。

4.2 场景二：显式 GC 的去与留

4.2.1 现象

除了扩容缩容会触发 CMS GC 之外，还有 Old 区达到回收阈值、MetaSpace 空间不足、Young 区晋升失败、大对象担保失败等几种触发条件，如果这些情况都没有发生却触发了 GC？这种情况有可能是代码中手动调用了 System.gc 方法，此时可以找到 GC 日志中的 GC Cause 确认下。那么这种 GC 到底有没有问题，翻看网上的一些资料，有人说可以添加 `-XX:+DisableExplicitGC` 参数来避免这种 GC，也有人说不能加这个参数，加了就会影响 Native Memory 的回收。先说结论，笔者这里建议保留 System.gc，那为什么要保留？我们一起来分析下。

4.2.2 原因

找到 System.gc 在 Hotspot 中的源码，可以发现增加 `-XX:+DisableExplicitGC` 参数后，这个方法变成了一个空方法，如果没有加的话便会调用 `Universe::heap()::collect` 方法，继续跟进到这个方法中，发现 System.gc 会引发一次 STW 的 Full GC，对整个堆做收集。

```
JVM_ENTRY_NO_ENV(void, JVM_GC(void))
  JVMWrapper("JVM_GC");
  if (!DisableExplicitGC) {
    Universe::heap()->collect(GCCause::_java_lang_system_gc);
  }
JVM_END
```

```
void GenCollectedHeap::collect(GCCause::Cause cause) {
  if (cause == GCCause::_wb_young_gc) {
    // Young collection for the WhiteBox API.
    collect(cause, YoungGen);
  } else {
#ifdef ASSERT
  if (cause == GCCause::_scavenge_alot) {
    // Young collection only.
    collect(cause, YoungGen);
  } else {
    // Stop-the-world full collection.
    collect(cause, OldGen);
  }
#else
    // Stop-the-world full collection.
    collect(cause, OldGen);
#endif
  }
}
```

保留 System.gc

此处补充一个知识点，**CMS GC 共分为 Background 和 Foreground 两种模式**，前者就是我们常规理解中的并发收集，可以不影响正常的业务线程运行，但 Foreground Collector 却有很大的差异，他会进行一次压缩式 GC。此压缩式 GC 使用的是跟 Serial Old GC 一样的 Lisp2 算法，其使用 Mark-Compact 来做 Full GC，一般称之为 MSC（Mark-Sweep-Compact），它收集的范围是 Java 堆的 Young 区和 Old 区以及 MetaSpace。由上面的算法章节中我们知道 compact 的代价是巨大的，那么使用 Foreground Collector 时将会带来非常长的 STW。如果在应用程序中 System.gc 被频繁调用，那就非常危险了。

去掉 System.gc

如果禁用掉的话就会带来另外一个内存泄漏问题，此时就需要说一下 DirectByteBuffer，它有着零拷贝等特点，被 Netty 等各种 NIO 框架使用，会使用到堆外内存。堆内存由 JVM 自己管理，堆外内存必须要手动释放，DirectByteBuffer 没有 Finalizer，它的 Native Memory 的清理工作是通过 `sun.misc.Cleaner` 自动完成

的，是一种基于 PhantomReference 的清理工具，比普通的 Finalizer 轻量些。

为 DirectByteBuffer 分配空间过程中会显式调用 System.gc ，希望通过 Full GC 来强迫已经无用的 DirectByteBuffer 对象释放掉它们关联的 Native Memory，下面为代码实现：

```
// These methods should be called whenever direct memory is allocated or
// freed.  They allow the user to control the amount of direct memory
// which a process may access.  All sizes are specified in bytes.
static void reserveMemory(long size) {

    synchronized (Bits.class) {
        if (!memoryLimitSet && VM.isBooted()) {
            maxMemory = VM.maxDirectMemory();
            memoryLimitSet = true;
        }
        if (size <= maxMemory - reservedMemory) {
            reservedMemory += size;
            return;
        }
    }

    System.gc();
    try {
        Thread.sleep(100);
    } catch (InterruptedException x) {
        // Restore interrupt status
        Thread.currentThread().interrupt();
    }
    synchronized (Bits.class) {
        if (reservedMemory + size > maxMemory)
            throw new OutOfMemoryError("Direct buffer memory");
        reservedMemory += size;
    }

}
```

HotSpot VM 只会在 Old GC 的时候才会对 Old 中的对象做 Reference Processing，而在 Young GC 时只会对 Young 里的对象做 Reference Processing。Young 中的 DirectByteBuffer 对象会在 Young GC 时被处理，也就是说，做 CMS GC 的话会对 Old 做 Reference Processing，进而能触发 Cleaner 对已死的 DirectByteBuffer 对象做清理工作。但如果很长一段时间里没做过 GC 或者只做了

Young GC 的话则不会在 Old 触发 Cleaner 的工作，那么就可能让本来已经死亡，但已经晋升到 Old 的 DirectByteBuffer 关联的 Native Memory 得不到及时释放。这几个实现特征使得依赖于 System.gc 触发 GC 来保证 DirectByteMemory 的清理工作能及时完成。如果打开了 `-XX:+DisableExplicitGC`，清理工作就可能得不到及时完成，于是就有发生 Direct Memory 的 OOM。

4.2.3 策略

通过上面的分析看到，无论是保留还是去掉都会有一定的风险点，不过目前互联网中的 RPC 通信会大量使用 NIO，所以笔者在这里建议保留。此外 JVM 还提供了 `-XX:+ExplicitGCInvokesConcurrent` 和 `-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses` 参数来将 System.gc 的触发类型从 Foreground 改为 Background，同时 Background 也会做 Reference Processing，这样的话就能大幅降低了 STW 开销，同时也不会发生 NIO Direct Memory OOM。

4.2.4 小结

不止 CMS，在 G1 或 ZGC 中开启 `ExplicitGCInvokesConcurrent` 模式，都会采用高性能的并发收集方式进行收集，不过还是建议在代码规范方面也要做好约束，规范好 System.gc 的使用。

P.S. HotSpot 对 System.gc 有特别处理，最主要的地方体现在一次 System.gc 是否与普通 GC 一样会触发 GC 的统计 / 阈值数据的更新，HotSpot 里的许多 GC 算法都带有自适应的功能，会根据先前收集的效率来决定接下来的 GC 中使用的参数，但 System.gc 默认不更新这些统计数据，避免用户强行 GC 对这些自适应功能的干扰（可以参考 -XX:+UseAdaptiveSizePolicyWithSystemGC 参数，默认是 false）。

4.3 场景三：MetaSpace 区 OOM

4.3.1 现象

JVM 在启动后或者某个时间点开始，**MetaSpace 的已使用大小在持续增长，同时每次 GC 也无法释放，调大 MetaSpace 空间也无法彻底解决**。

4.3.2 原因

在讨论为什么会 OOM 之前，我们先来看一下这个区里面会存什么数据，Java7 之前字符串常量池被放到了 Perm 区，所有被 intern 的 String 都会被存在这里，由于 String.intern 是不受控的，所以 `-XX:MaxPermSize` 的值也不太好设置，经常会出现 `java.lang.OutOfMemoryError: PermGen space` 异常，所以在 Java7 之后常量池等字面量（Literal）、类静态变量（Class Static）、符号引用（Symbols Reference）等几项被移到 Heap 中。而 Java8 之后 PermGen 也被移除，取而代之的是 MetaSpace。

在最底层，JVM 通过 mmap 接口向操作系统申请内存映射，每次申请 2MB 空间，这里是虚拟内存映射，不是真的就消耗了主存的 2MB，只有之后在使用的时候才会真的消耗内存。申请的这些内存放到一个链表中 VirtualSpaceList，作为其中的一个 Node。

在上层，MetaSpace 主要由 Klass Metaspace 和 NoKlass Metaspace 两大部分组成。

- **Klass MetaSpace：**就是用来存 Klass 的，就是 Class 文件在 JVM 里的运行时数据结构，这部分默认放在 Compressed Class Pointer Space 中，是一块连续的内存区域，紧接着 Heap。Compressed Class Pointer Space 不是必须有的，如果设置了 `-XX:-UseCompressedClassPointers`，或者 `-Xmx` 设置大于 32 G，就不会有这块内存，这种情况下 Klass 都会存在 NoKlass Metaspace 里。

- **NoKlass MetaSpace：**专门来存 Klass 相关的其他的内容，比如 Method，ConstantPool 等，可以由多块不连续的内存组成。虽然叫做 NoKlass Metaspace，但是也其实可以存 Klass 的内容，上面已经提到了对应场景。

具体的定义都可以在源码 shared/vm/memory/metaspace.hpp 中找到：

```
class Metaspace : public AllStatic {

  friend class MetaspaceShared;

 public:
  enum MetadataType {
    ClassType,
    NonClassType,
    MetadataTypeCount
  };
  enum MetaspaceType {
    ZeroMetaspaceType = 0,
    StandardMetaspaceType = ZeroMetaspaceType,
    BootMetaspaceType = StandardMetaspaceType + 1,
    AnonymousMetaspaceType = BootMetaspaceType + 1,
    ReflectionMetaspaceType = AnonymousMetaspaceType + 1,
    MetaspaceTypeCount
  };

 private:

  // Align up the word size to the allocation word size
  static size_t align_word_size_up(size_t);

  // Aligned size of the metaspace.
  static size_t _compressed_class_space_size;

  static size_t compressed_class_space_size() {
    return _compressed_class_space_size;
  }

  static void set_compressed_class_space_size(size_t size) {
    _compressed_class_space_size = size;
  }

  static size_t _first_chunk_word_size;
  static size_t _first_class_chunk_word_size;

  static size_t _commit_alignment;
  static size_t _reserve_alignment;
```

```
  DEBUG_ONLY(static bool   _frozen;)

  // Virtual Space lists for both classes and other metadata
  static metaspace::VirtualSpaceList* _space_list;
  static metaspace::VirtualSpaceList* _class_space_list;

  static metaspace::ChunkManager* _chunk_manager_metadata;
  static metaspace::ChunkManager* _chunk_manager_class;

  static const MetaspaceTracer* _tracer;
}
```

MetaSpace 的对象为什么无法释放，我们看下面两点：

- **MetaSpace 内存管理：**类和其元数据的生命周期与其对应的类加载器相同，只要类的类加载器是存活的，在 Metaspace 中的类元数据也是存活的，不能被回收。每个加载器有单独的存储空间，通过 ClassLoaderMetaspace 来进行管理 SpaceManager* 的指针，相互隔离的。
- **MetaSpace 弹性伸缩：**由于 MetaSpace 空间和 Heap 并不在一起，所以这块的空间可以不用设置或者单独设置，一般情况下避免 MetaSpace 耗尽 VM 内存都会设置一个 MaxMetaSpaceSize，在运行过程中，如果实际大小小于这个值，JVM 就会通过 `-XX:MinMetaspaceFreeRatio` 和 `-XX:-MaxMetaspaceFreeRatio` 两个参数动态控制整个 MetaSpace 的大小，具体使用可以看 `MetaSpaceGC::compute_new_size()` 方法（下方代码），这个方法会在 CMSCollector 和 G1CollectorHeap 等几个收集器执行 GC 时调用。这个里面会根据 `used_after_gc`，`MinMetaspaceFreeRatio` 和 `MaxMetaspaceFreeRatio` 这三个值计算出来一个新的 `_capacity_until_GC` 值（水位线）。然后根据实际的 `_capacity_until_GC` 值使用 `MetaspaceGC::inc_capacity_until_GC()` 和 `MetaspaceGC::dec_capacity_until_GC()` 进行 expand 或 shrink，这个过程也可以参照场景一中的伸缩模型进行理解。

```
void MetaspaceGC::compute_new_size() {
  assert(_shrink_factor <= 100, "invalid shrink factor");
  uint current_shrink_factor = _shrink_factor;
```

```
  _shrink_factor = 0;
  const size_t used_after_gc = MetaspaceUtils::committed_bytes();
  const size_t capacity_until_GC = MetaspaceGC::capacity_until_GC();

  const double minimum_free_percentage = MinMetaspaceFreeRatio / 100.0;
  const double maximum_used_percentage = 1.0 - minimum_free_percentage;

  const double min_tmp = used_after_gc / maximum_used_percentage;
  size_t minimum_desired_capacity =
    (size_t)MIN2(min_tmp, double(max_uintx));
  // Don't shrink less than the initial generation size
  minimum_desired_capacity = MAX2(minimum_desired_capacity,
                                  MetaspaceSize);

  log_trace(gc, metaspace)("MetaspaceGC::compute_new_size: ");
  log_trace(gc, metaspace)("    minimum_free_percentage: %6.2f
maximum_used_percentage: %6.2f",
                           minimum_free_percentage, maximum_used_
percentage);
  log_trace(gc, metaspace)("     used_after_gc       : %6.1fKB", used_
after_gc / (double) K);


  size_t shrink_bytes = 0;
  if (capacity_until_GC < minimum_desired_capacity) {
    // If we have less capacity below the metaspace HWM, then
    // increment the HWM.
    size_t expand_bytes = minimum_desired_capacity - capacity_until_GC;
    expand_bytes = align_up(expand_bytes, Metaspace::commit_alignment());
    // Don't expand unless it's significant
    if (expand_bytes >= MinMetaspaceExpansion) {
      size_t new_capacity_until_GC = 0;
      bool succeeded = MetaspaceGC::inc_capacity_until_GC(expand_bytes,
&new_capacity_until_GC);
      assert(succeeded, "Should always succesfully increment HWM when
at safepoint");

      Metaspace::tracer()->report_gc_threshold(capacity_until_GC,
                                               new_capacity_until_GC,

MetaspaceGCThresholdUpdater::ComputeNewSize);
      log_trace(gc, metaspace)("    expanding:  minimum_desired_
capacity: %6.1fKB  expand_bytes: %6.1fKB  MinMetaspaceExpansion:
%6.1fKB  new metaspace HWM:  %6.1fKB",
                               minimum_desired_capacity / (double) K,
                               expand_bytes / (double) K,
                               MinMetaspaceExpansion / (double) K,
                               new_capacity_until_GC / (double) K);
    }
```

```
    return;
  }

  // No expansion, now see if we want to shrink
  // We would never want to shrink more than this
  assert(capacity_until_GC >= minimum_desired_capacity,
         SIZE_FORMAT " >= " SIZE_FORMAT,
         capacity_until_GC, minimum_desired_capacity);
  size_t max_shrink_bytes = capacity_until_GC - minimum_desired_capacity;

  // Should shrinking be considered?
  if (MaxMetaspaceFreeRatio < 100) {
    const double maximum_free_percentage = MaxMetaspaceFreeRatio /
100.0;
    const double minimum_used_percentage = 1.0 - maximum_free_
percentage;
    const double max_tmp = used_after_gc / minimum_used_percentage;
    size_t maximum_desired_capacity = (size_t)MIN2(max_tmp, double(max_
uintx));
    maximum_desired_capacity = MAX2(maximum_desired_capacity,
                                    MetaspaceSize);
    log_trace(gc, metaspace)("    maximum_free_percentage: %6.2f
minimum_used_percentage: %6.2f",
                             maximum_free_percentage, minimum_used_
percentage);
    log_trace(gc, metaspace)("    minimum_desired_capacity: %6.1fKB
maximum_desired_capacity: %6.1fKB",
                             minimum_desired_capacity / (double) K,
maximum_desired_capacity / (double) K);

    assert(minimum_desired_capacity <= maximum_desired_capacity,
           "sanity check");

    if (capacity_until_GC > maximum_desired_capacity) {
      // Capacity too large, compute shrinking size
      shrink_bytes = capacity_until_GC - maximum_desired_capacity;
      shrink_bytes = shrink_bytes / 100 * current_shrink_factor;

      shrink_bytes = align_down(shrink_bytes, Metaspace::commit_
alignment());

      assert(shrink_bytes <= max_shrink_bytes,
             "invalid shrink size " SIZE_FORMAT " not <= " SIZE_FORMAT,
             shrink_bytes, max_shrink_bytes);
      if (current_shrink_factor == 0) {
        _shrink_factor = 10;
      } else {
        _shrink_factor = MIN2(current_shrink_factor * 4, (uint) 100);
      }
```

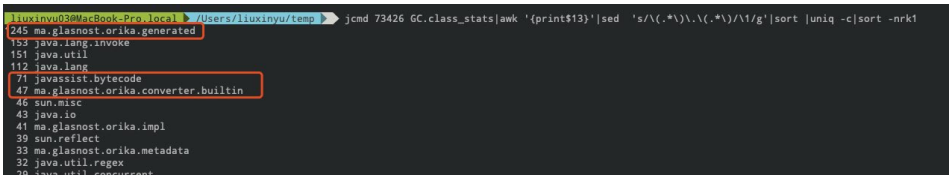
```
      log_trace(gc, metaspace)("    shrinking:  initThreshold: %.1fK
maximum_desired_capacity: %.1fK",
                               MetaspaceSize / (double) K, maximum_
desired_capacity / (double) K);
      log_trace(gc, metaspace)("    shrink_bytes: %.1fK  current_
shrink_factor: %d  new shrink factor: %d  MinMetaspaceExpansion:
%.1fK",
                               shrink_bytes / (double) K, current_shrink_
factor, _shrink_factor, MinMetaspaceExpansion / (double) K);
    }
  }

  // Don't shrink unless it's significant
  if (shrink_bytes >= MinMetaspaceExpansion &&
      ((capacity_until_GC - shrink_bytes) >= MetaspaceSize)) {
    size_t new_capacity_until_GC = MetaspaceGC::dec_capacity_until_
GC(shrink_bytes);
    Metaspace::tracer()->report_gc_threshold(capacity_until_GC,
                                             new_capacity_until_GC,

MetaspaceGCThresholdUpdater::ComputeNewSize);
  }
}
```

由场景一可知，为了避免弹性伸缩带来的额外 GC 消耗，我们会将 `-XX:MetaSpaceSize` 和 `-XX:MaxMetaSpaceSize` 两个值设置为固定的，但是这样也会导致在空间不够的时候无法扩容，然后频繁地触发 GC，最终 OOM。所以关键原因就是 ClassLoader 不停地在内存中 load 了新的 Class ，一般这种问题都发生在动态类加载等情况上。

4.3.3 策略

了解大概什么原因后，如何定位和解决就很简单了，可以 dump 快照之后通过 JProfiler 或 MAT 观察 Classes 的 Histogram（直方图）即可，或者直接通过命令即可定位，jcmd 打几次 Histogram 的图，看一下具体是哪个包下的 Class 增加较多就可以定位了。不过有时候也要结合 InstBytes、KlassBytes、Bytecodes、MethodAll 等几项指标综合来看下。如下图便是笔者使用 jcmd 排查到一个 Orika 的问题。

```
jcmd <PID> GC.class_stats|awk '{print$13}'|sed  's/\(.*\)\.\(.*\)/\1/
g'|sort |uniq -c|sort -nrk1
```



如果无法从整体的角度定位，可以添加 `-XX:+TraceClassLoading` 和 `-XX:+TraceClassUnLoading` 参数观察详细的类加载和卸载信息。

4.3.4 小结

原理理解比较复杂，但定位和解决问题会比较简单，经常会出问题的几个点有 Orika 的 classMap、JSON 的 ASMSerializer、Groovy 动态加载类等，基本都集中在反射、Javasisit 字节码增强、CGLIB 动态代理、OSGi 自定义类加载器等的技术点上。另外就是及时给 MetaSpace 区的使用率加一个监控，如果指标有波动提前发现并解决问题。

4.4 场景四：过早晋升 *

4.4.1 现象

这种场景主要发生在分代的收集器上面，专业的术语称为“Premature Promotion”。90% 的对象朝生夕死，只有在 Young 区经历过几次 GC 的洗礼后才会晋升到 Old 区，每经历一次 GC 对象的 GC Age 就会增长 1，最大通过 `-XX:MaxTenuringThreshold` 来控制。

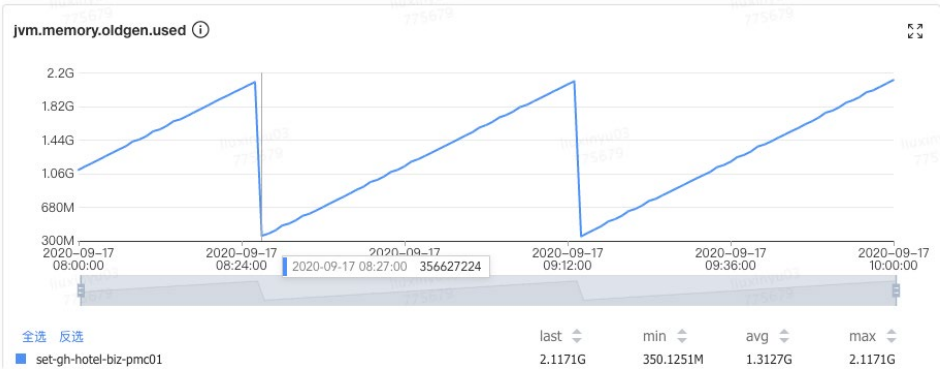
过早晋升一般不会直接影响 GC，总会伴随着浮动垃圾、大对象担保失败等问题，但这些问题不是立刻发生的，我们可以观察以下几种现象来判断是否发生了过早晋升。

分配速率接近于晋升速率，对象晋升年龄较小。

GC 日志中出现“Desired survivor size 107347968 bytes, **new threshold 1(max 6)**”等信息，说明此时经历过一次 GC 就会放到 Old 区。

Full GC 比较频繁，且经历过一次 GC 之后 Old 区的**变化比例非常大**。

比如说 Old 区触发的回收阈值是 80%，经历过一次 GC 之后下降到了 10%，这就说明 Old 区的 70% 的对象存活时间其实很短，如下图所示，Old 区大小每次 GC 后从 2.1G 回收到 300M，也就是说回收掉了 1.8G 的垃圾，只有 **300M 的活跃对象**。整个 Heap 目前是 4G，活跃对象只占了不到十分之一。



过早晋升的危害：

- Young GC 频繁，总的吞吐量下降。
- Full GC 频繁，可能会有较大停顿。

4.4.2 原因

主要的原因有以下两点：

- **Young/Eden 区过小：**过小的直接后果就是 Eden 被装满的时间变短，本应该回收的对象参与了 GC 并晋升，Young GC 采用的是复制算法，由基础篇我们知道 copying 耗时远大于 mark，也就是 Young GC 耗时本质上就是 copy 的时间（CMS 扫描 Card Table 或 G1 扫描 Remember Set 出问题的情况另说），没来及回收的对象增大了回收的代价，所以 Young GC 时间增加，同时又无法快速释放空间，Young GC 次数也跟着增加。
- **分配速率过大：**可以观察出问题前后 Mutator 的分配速率，如果有明显波动可以尝试观察网卡流量、存储类中间件慢查询日志等信息，看是否有大量数据被加载到内存中。

同时无法 GC 掉对象还会带来另外一个问题，引发动态年龄计算：JVM 通过 `-XX:-MaxTenuringThreshold` 参数来控制晋升年龄，每经过一次 GC，年龄就会加一，达到最大年龄就可以进入 Old 区，最大值为 15（因为 JVM 中使用 4 个比特来表示对象的年龄）。设定固定的 MaxTenuringThreshold 值作为晋升条件：

- MaxTenuringThreshold 如果设置得过大，原本应该晋升的对象一直停留在 Survivor 区，直到 Survivor 区溢出，一旦溢出发生，Eden + Survivor 中对象将不再依据年龄全部提升到 Old 区，这样对象老化的机制就失效了。
- MaxTenuringThreshold 如果设置得过小，过早晋升即对象不能在 Young 区充分被回收，大量短期对象被晋升到 Old 区，Old 区空间迅速增长，引起频繁的 Major GC，分代回收失去了意义，严重影响 GC 性能。

相同应用在不同时间的表现不同，特殊任务的执行或者流量成分的变化，都会导致对象的生命周期分布发生波动，那么固定的阈值设定，因为无法动态适应变化，会造成和上面问题，所以 Hotspot 会使用动态计算的方式来调整晋升的阈值。

具体动态计算可以看一下 Hotspot 源码，具体在 /src/hotspot/share/gc/shared/ageTable.cpp 的 `compute_tenuring_threshold` 方法中：

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
  //TargetSurvivorRatio 默认 50，意思是：在回收之后希望 survivor 区的占用率达到这个比例
  size_t desired_survivor_size = (size_t)((((double) survivor_capacity)*TargetSurvivorRatio)/100);
  size_t total = 0;
  uint age = 1;
  assert(sizes[0] == 0, "no objects with age zero should be recorded");
  while (age < table_size) {//table_size=16
    total += sizes[age];
    // 如果加上这个年龄的所有对象的大小之后，占用量 > 期望的大小，就设置 age 为新的晋升阈值
    if (total > desired_survivor_size) break;
    age++;
  }

  uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
  if (PrintTenuringDistribution || UsePerfData) {
```

```
    // 打印期望的 survivor 的大小以及新计算出来的阈值，和设置的最大阈值
    if (PrintTenuringDistribution) {
      gclog_or_tty->cr();
      gclog_or_tty->print_cr("Desired survivor size " SIZE_FORMAT "
bytes, new threshold %u (max %u)",
        desired_survivor_size*oopSize, result, (int)
MaxTenuringThreshold);
    }

    total = 0;
    age = 1;
    while (age < table_size) {
      total += sizes[age];
      if (sizes[age] > 0) {
        if (PrintTenuringDistribution) {
          gclog_or_tty->print_cr("- age %3u: " SIZE_FORMAT_W(10) "
bytes, " SIZE_FORMAT_W(10) " total",
                                 age,    sizes[age]*oopSize,
total*oopSize);
        }
      }
      if (UsePerfData) {
        _perf_sizes[age]->set_value(sizes[age]*oopSize);
      }
      age++;
    }
    if (UsePerfData) {
      SharedHeap* sh = SharedHeap::heap();
      CollectorPolicy* policy = sh->collector_policy();
      GCPolicyCounters* gc_counters = policy->counters();
      gc_counters->tenuring_threshold()->set_value(result);
      gc_counters->desired_survivor_size()->set_value(
        desired_survivor_size*oopSize);
    }
  }

  return result;
}
```

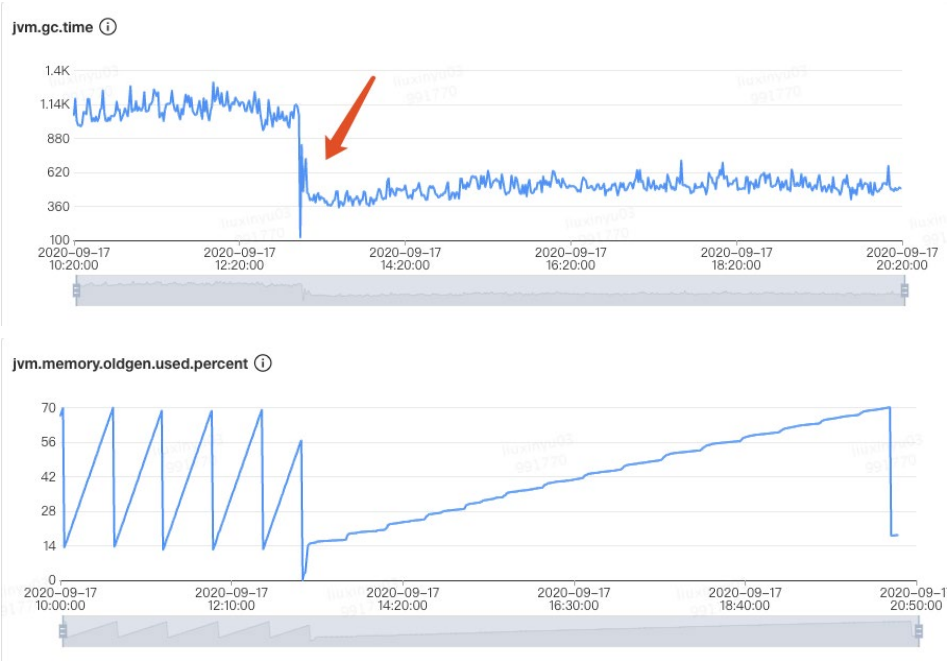
可以看到 Hotspot 遍历所有对象时，从所有年龄为 0 的对象占用的空间开始累加，如果加上年龄等于 n 的所有对象的空间之后，使用 Survivor 区的条件值（Target-SurvivorRatio / 100，TargetSurvivorRatio 默认值为 50）进行判断，若大于这个值则结束循环，将 n 和 MaxTenuringThreshold 比较，若 n 小，则阈值为 n，若 n 大，则只能去设置最大阈值为 MaxTenuringThreshold。**动态年龄触发后导致更多**

的对象进入了 Old 区，造成资源浪费。

4.4.3 策略

知道问题原因后我们就有解决的方向，如果是 **Young/Eden 区过小**，我们可以在总的 Heap 内存不变的情况下适当增大 Young 区，具体怎么增加？一般情况下 Old 的大小应当为活跃对象的 2~3 倍左右，考虑到浮动垃圾问题最好在 3 倍左右，剩下的都可以分给 Young 区。

拿笔者的一次典型过早晋升优化来看，原配置为 Young 1.2G + Old 2.8G，通过观察 CMS GC 的情况找到存活对象大概为 300~400M，于是调整 Old 1.5G 左右，剩下 2.5G 分给 Young 区。仅仅调了一个 Young 区大小参数（`-Xmn`），整个 JVM 一分钟 Young GC 从 26 次降低到了 11 次，单次时间也没有增加，总的 GC 时间从 1100ms 降低到了 500ms，CMS GC 次数也从 40 分钟左右一次降低到了 7 小时 30 分钟一次。



如果是分配速率过大：

- **偶发较大**：通过内存分析工具找到问题代码，从业务逻辑上做一些优化。
- **一直较大**：当前的 Collector 已经不满足 Mutator 的期望了，这种情况要么扩容 Mutator 的 VM，要么调整 GC 收集器类型或加大空间。

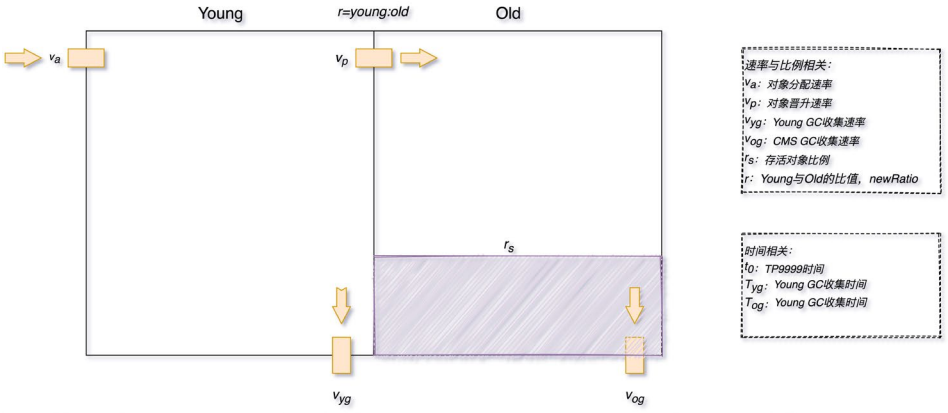
4.4.4 小结

过早晋升问题一般不会特别明显，但日积月累之后可能会爆发一波收集器退化之类的问题，所以我们还是要提前避免掉的，可以看看自己系统里面是否有这些现象，如果比较匹配的话，可以尝试优化一下。一行代码优化的 ROI 还是很高的。

如果在观察 Old 区前后比例变化的过程中，发现可以回收的比例非常小，如从 80% 只回收到了 60%，说明我们大部分对象都是存活的，Old 区的空间可以适当调大些。

4.4.5 加餐

关于在调整 Young 与 Old 的比例时，如何选取具体的 NewRatio 值，这里将问题抽象成为一个蓄水池模型，找到以下关键衡量指标，大家可以根据自己场景进行推算。



$$(1) r = f(v_a, v_p, v_{yc}, v_{oc}, r_s)$$

$$(2) T_{stw} = T_{yc} + T_{oc} = \int_0^t g(v_{yc}, v_p)dt + \int_0^t g(v_{oc})dt, t \in (0, +\infty)$$

$$(3) f(t) = \lim_{\bar{T_{yg}} \to 0, T_{oc} \to 0} \frac{\Sigma_{k=1}^{n}(t_{ak} - t_{bk} + \bar{T_{yg}}) + T_{oc}}{n} \Rightarrow f(t) = \frac{T_n}{n} < t_0$$

($\bar{T_{yg}}$为平均一次$YoungGC$时间，t_{ak}为第k次$YoungGC$结束时间，t_{bk}为第k次$YoungGC$开始时间，T_n为总运行时间)

- NewRatio 的值 r 与 va、vp、vyc、voc、rs 等值存在一定函数相关性（rs 越小 r 越大、r 越小 vp 越小，…，之前尝试使用 NN 来辅助建模，但目前还没有完全算出具体的公式，有想法的同学可以在评论区给出你的答案）。
- 总停顿时间 T 为 Young GC 总时间 Tyc 和 Old GC 总时间 Toc 之和，其中 Tyc 与 vyc 和 vp 相关，Toc 与 voc 相关。
- 忽略掉 GC 时间后，两次 Young GC 的时间间隔要大于 TP9999 时间，这样尽量让对象在 Eden 区就被回收，可以减少很多停顿。

4.5 场景五：CMS Old GC 频繁 *

4.5.1 现象

Old 区频繁的做 CMS GC，但是每次耗时不是特别长，整体最大 STW 也在可接受范围内，但由于 GC 太频繁导致吞吐下降比较多。

4.5.2 原因

这种情况比较常见，基本都是一次 Young GC 完成后，负责处理 CMS GC 的一个后台线程 concurrentMarkSweepThread 会不断地轮询，使用 `shouldConcurrentCollect()` 方法做一次检测，判断是否达到了回收条件。如果达到条件，使用 `collect_in_background()` 启动一次 Background 模式 GC。轮询的判断是使用 `sleepBeforeNextCycle()` 方法，间隔周期为 `-XX:CMSWaitDuration` 决定，默认为 2s。

具体代码在：src/hotspot/share/gc/cms/concurrentMarkSweepThread.cpp。

```
void ConcurrentMarkSweepThread::run_service() {
  assert(this == cmst(), "just checking");

  if (BindCMSThreadToCPU && !os::bind_to_processor(CPUForCMSThread)) {
```

```
    log_warning(gc)("Couldn't bind CMS thread to processor " UINTX_
FORMAT, CPUForCMSThread);
  }

  while (!should_terminate()) {
    sleepBeforeNextCycle();
    if (should_terminate()) break;
    GCIdMark gc_id_mark;
    GCCause::Cause cause = _collector->_full_gc_requested ?
      _collector->_full_gc_cause : GCCause::_cms_concurrent_mark;
    _collector->collect_in_background(cause);
  }
  verify_ok_to_terminate();
}
```

```
void ConcurrentMarkSweepThread::sleepBeforeNextCycle() {
  while (!should_terminate()) {
    if(CMSWaitDuration >= 0) {
      // Wait until the next synchronous GC, a concurrent full gc
      // request or a timeout, whichever is earlier.
      wait_on_cms_lock_for_scavenge(CMSWaitDuration);
    } else {
      // Wait until any cms_lock event or check interval not to call
shouldConcurrentCollect permanently
      wait_on_cms_lock(CMSCheckInterval);
    }
    // Check if we should start a CMS collection cycle
    if (_collector->shouldConcurrentCollect()) {
      return;
    }
    // .. collection criterion not yet met, let's go back
    // and wait some more
  }
}
```

判断是否进行回收的代码在：/src/hotspot/share/gc/cms/concurrentMarkSweep-Generation.cpp。

```
bool CMSCollector::shouldConcurrentCollect() {
  LogTarget(Trace, gc) log;

  if (_full_gc_requested) {
    log.print("CMSCollector: collect because of explicit  gc request
(or GCLocker)");
    return true;
  }
```

```
  FreelistLocker x(this);
  // ------------------------------------------------------------------
--
  // Print out lots of information which affects the initiation of
  // a collection.
  if (log.is_enabled() && stats().valid()) {
    log.print("CMSCollector shouldConcurrentCollect: ");

    LogStream out(log);
    stats().print_on(&out);

    log.print("time_until_cms_gen_full %3.7f", stats().time_until_cms_
gen_full());
    log.print("free=" SIZE_FORMAT, _cmsGen->free());
    log.print("contiguous_available=" SIZE_FORMAT, _cmsGen->contiguous_
available());
    log.print("promotion_rate=%g", stats().promotion_rate());
    log.print("cms_allocation_rate=%g", stats().cms_allocation_rate());
    log.print("occupancy=%3.7f", _cmsGen->occupancy());
    log.print("initiatingOccupancy=%3.7f", _cmsGen->initiating_
occupancy());
    log.print("cms_time_since_begin=%3.7f", stats().cms_time_since_
begin());
    log.print("cms_time_since_end=%3.7f", stats().cms_time_since_
end());
    log.print("metadata initialized %d", MetaspaceGC::should_concurrent_
collect());
  }
  // ------------------------------------------------------------------
--
  if (!UseCMSInitiatingOccupancyOnly) {
    if (stats().valid()) {
      if (stats().time_until_cms_start() == 0.0) {
        return true;
      }
    } else {

      if (_cmsGen->occupancy() >= _bootstrap_occupancy) {
        log.print(" CMSCollector: collect for bootstrapping
statistics: occupancy = %f, boot occupancy = %f",
                  _cmsGen->occupancy(), _bootstrap_occupancy);
        return true;
      }
    }
  }

  if (_cmsGen->should_concurrent_collect()) {
    log.print("CMS old gen initiated");
    return true;
```

```
  }

  // We start a collection if we believe an incremental collection
may fail;
  // this is not likely to be productive in practice because it's
probably too
  // late anyway.
  CMSHeap* heap = CMSHeap::heap();
  if (heap->incremental_collection_will_fail(true /* consult_young */))
{
    log.print("CMSCollector: collect because incremental collection
will fail ");
    return true;
  }

  if (MetaspaceGC::should_concurrent_collect()) {
    log.print("CMSCollector: collect for metadata allocation ");
    return true;
  }

  // CMSTriggerInterval starts a CMS cycle if enough time has passed.
  if (CMSTriggerInterval >= 0) {
    if (CMSTriggerInterval == 0) {
      // Trigger always
      return true;
    }

    // Check the CMS time since begin (we do not check the stats
validity
    // as we want to be able to trigger the first CMS cycle as well)
    if (stats().cms_time_since_begin() >= (CMSTriggerInterval / ((double)
MILLIUNITS))) {
      if (stats().valid()) {
        log.print("CMSCollector: collect because of trigger interval
(time since last begin %3.7f secs)",
                  stats().cms_time_since_begin());
      } else {
        log.print("CMSCollector: collect because of trigger interval
(first collection)");
      }
      return true;
    }
  }

  return false;
}
```

分析其中逻辑判断是否触发 GC，分为以下几种情况：

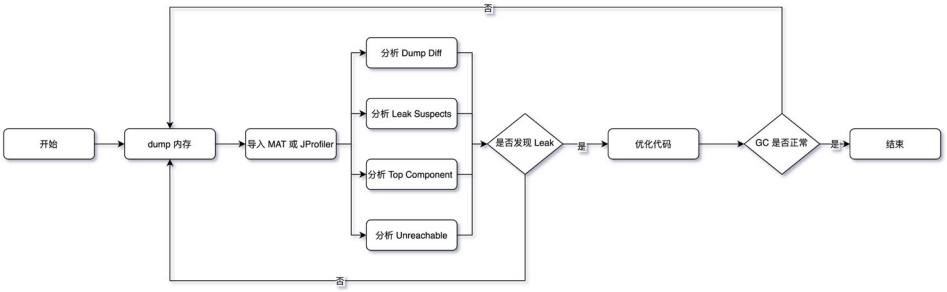
- **触发 CMS GC：** 通过调用 `_collector->collect_in_background()` 进行触发 Background GC 。
  - CMS 默认采用 JVM 运行时的统计数据判断是否需要触发 CMS GC，如果需要根据 `-XX:CMSInitiatingOccupancyFraction` 的值进行判断，需要设置参数 `-XX:+UseCMSInitiatingOccupancyOnly`。
  - 如果开启了 `-XX:UseCMSInitiatingOccupancyOnly` 参数，判断当前 Old 区使用率是否大于阈值，则触发 CMS GC，该阈值可以通过参数 `-XX:CMSInitiatingOccupancyFraction` 进行设置，如果没有设置，默认为 92%。
  - 如果之前的 Young GC 失败过，或者下次 Young 区执行 Young GC 可能失败，这两种情况下都需要触发 CMS GC。
  - CMS 默认不会对 MetaSpace 或 Perm 进行垃圾收集，如果希望对这些区域进行垃圾收集，需要设置参数 `-XX:+CMSClassUnloadingEnabled`。
- **触发 Full GC：** 直接进行 Full GC，这种情况到场景七中展开说明。
  - 如果 `_full_gc_requested` 为真，说明有明确的需求要进行 GC，比如调用 System.gc。
  - 在 Eden 区为对象或 TLAB 分配内存失败，导致一次 Young GC，在 `GenCollectorPolicy` 类的 `satisfy_failed_allocation()` 方法中进行判断。

大家可以看一下源码中的日志打印，通过日志我们就可以比较清楚地知道具体的原因，然后就可以着手分析了。

4.5.3 策略

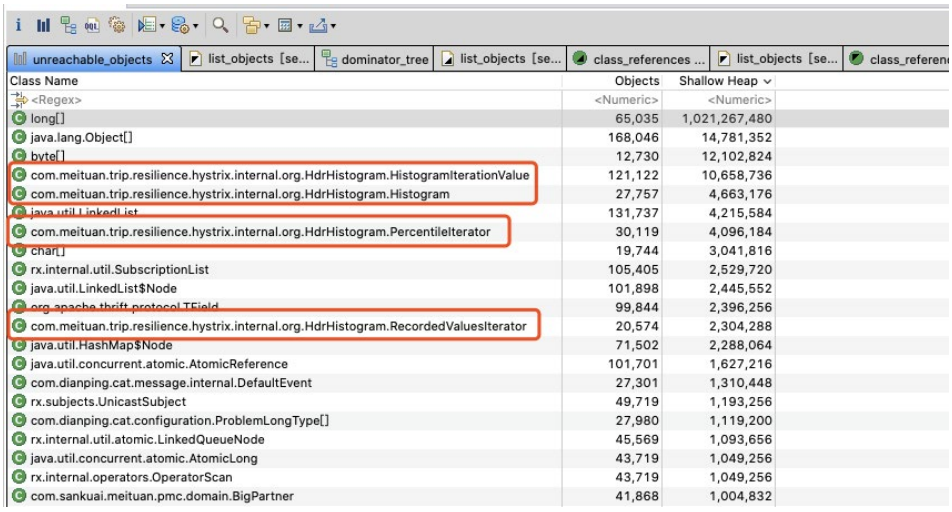
我们这里还是拿最常见的达到回收比例这个场景来说，与过早晋升不同的是这些对象确实存活了一段时间，Survival Time 超过了 TP9999 时间，但是又达不到长期存活，如各种数据库、网络链接，带有失效时间的缓存等。

处理这种常规内存泄漏问题基本是一个思路，主要步骤如下：



Dump Diff 和 Leak Suspects 比较直观就不介绍了，这里说下其它几个关键点：

- **内存 Dump：**使用 jmap、arthas 等 dump 堆进行快照时记得摘掉流量，同时**分别在 CMS GC 的发生前后分别 dump 一次**。
- **分析 Top Component：**要记得按照对象、类、类加载器、包等多个维度观察 Histogram，同时使用 outgoing 和 incoming 分析关联的对象，另外就是 Soft Reference 和 Weak Reference、Finalizer 等也要看一下。
- **分析 Unreachable：**重点看一下这个，关注下 Shallow 和 Retained 的大小。如下图所示，笔者之前一次 GC 优化，就根据 Unreachable Objects 发现了 Hystrix 的滑动窗口问题。



unreachable_objects | list_objects [se... | dominator_tree | list_objects [se... | class_references ... | list_objects [se... | class_referen

Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
long[]	65,035	1,021,267,480
java.lang.Object[]	168,046	14,781,352
byte[]	12,730	12,102,824
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.HistogramIterationValue	121,122	10,658,736
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.Histogram	27,757	4,663,176
java.util.LinkedList	131,737	4,215,584
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.PercentileIterator	30,119	4,096,184
char[]	19,744	3,041,816
rx.internal.util.SubscriptionList	105,405	2,529,720
java.util.LinkedList$Node	101,898	2,445,552
org.apache.thrift.protocol.TField	99,844	2,396,256
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.RecordedValuesIterator	20,574	2,304,288
java.util.HashMap$Node	71,502	2,288,064
java.util.concurrent.atomic.AtomicReference	101,701	1,627,216
com.dianping.cat.message.internal.DefaultEvent	27,301	1,310,448
rx.subjects.UnicastSubject	49,719	1,193,256
com.dianping.cat.configuration.ProblemLongType[]	27,980	1,119,200
rx.internal.util.atomic.LinkedQueueNode	45,569	1,093,656
java.util.concurrent.atomic.AtomicLong	43,719	1,049,256
rx.internal.operators.OperatorScan	43,719	1,049,256
com.sankuai.meituan.pmc.domain.BigPartner	41,868	1,004,832

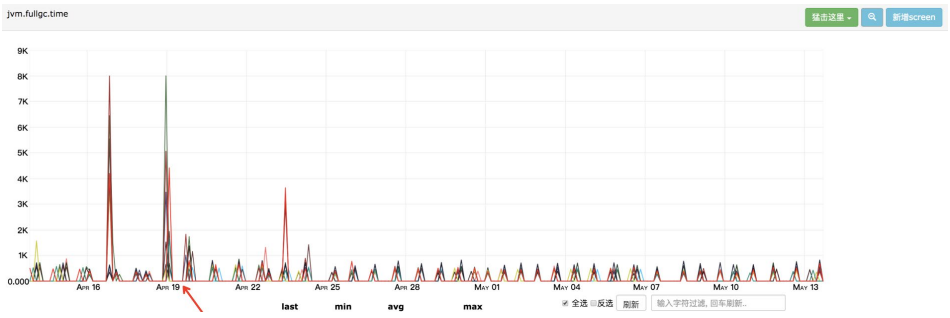
4.5.4 小结

经过整个流程下来基本就能定位问题了，不过在优化的过程中记得使用**控制变量**的方法来优化，防止一些会加剧问题的改动被掩盖。

4.6 场景六：单次 CMS Old GC 耗时长 *

4.6.1 现象

CMS GC 单次 STW 最大超过 1000ms，不会频繁发生，如下图所示最长达到了 8000ms。某些场景下会引起“雪崩效应”，这种场景非常危险，我们应该尽量避免出现。



4.6.2 原因

CMS 在回收的过程中，STW 的阶段主要是 Init Mark 和 Final Remark 这两个阶段，也是导致 CMS Old GC 最多的原因，另外有些情况就是在 STW 前等待 Mutator 的线程到达 SafePoint 也会导致时间过长，但这种情况较少，我们在此处主要讨论前者。发生收集器退化或者碎片压缩的场景请看场景七。

想要知道这两个阶段为什么会耗时，我们需要先看一下这两个阶段都会干什么。

核心代码都在 /src/hotspot/share/gc/cms/concurrentMarkSweepGeneration.cpp 中，内部有个线程 ConcurrentMarkSweepThread 轮询来校验，Old 区的垃圾回收相关细节被完全封装在 `CMSCollector` 中，调用入口就是 ConcurrentMarkSweepThread 调用的 `CMSCollector::collect_in_background` 和

`ConcurrentMarkSweepGeneration` 调用的 `CMSCollector::collect` 方法，此处我们讨论大多数场景的 `collect_in_background`。整个过程中会 STW 的主要是 initial Mark 和 Final Remark，核心代码在 `VM_CMS_Initial_Mark` / `VM_CMS_Final_Remark` 中，执行时需要将执行权交由 VMThread 来执行。

- CMS Init Mark 执行步骤，实现在 `CMSCollector::checkpointRootsInitialWork()` 和 `CMSParInitialMarkTask::work` 中，整体步骤和代码如下：

```
void CMSCollector::checkpointRootsInitialWork() {
  assert(SafepointSynchronize::is_at_safepoint(), "world should be
stopped");
  assert(_collectorState == InitialMarking, "just checking");

  // Already have locks.
  assert_lock_strong(bitMapLock());
  assert(_markBitMap.isAllClear(), "was reset at end of previous
cycle");

  // Setup the verification and class unloading state for this
  // CMS collection cycle.
  setup_cms_unloading_and_verification_state();

  GCTraceTime(Trace, gc, phases) ts("checkpointRootsInitialWork", _gc_
timer_cm);

  // Reset all the PLAB chunk arrays if necessary.
  if (_survivor_plab_array != NULL && !CMSPLABRecordAlways) {
    reset_survivor_plab_arrays();
  }

  ResourceMark rm;
  HandleMark  hm;

  MarkRefsIntoClosure notOlder(_span, &_markBitMap);
  CMSHeap* heap = CMSHeap::heap();

  verify_work_stacks_empty();
  verify_overflow_empty();

  heap->ensure_parsability(false);  // fill TLABs, but no need to
retire them
  // Update the saved marks which may affect the root scans.
  heap->save_marks();
```

```
  // weak reference processing has not started yet.
  ref_processor()->set_enqueuing_is_done(false);

  // Need to remember all newly created CLDs,
  // so that we can guarantee that the remark finds them.
  ClassLoaderDataGraph::remember_new_clds(true);

  // Whenever a CLD is found, it will be claimed before proceeding to
mark
  // the klasses. The claimed marks need to be cleared before marking
starts.
  ClassLoaderDataGraph::clear_claimed_marks();

  print_eden_and_survivor_chunk_arrays();

  {
    if (CMSParallelInitialMarkEnabled) {
      // The parallel version.
      WorkGang* workers = heap->workers();
      assert(workers != NULL, "Need parallel worker threads.");
      uint n_workers = workers->active_workers();

      StrongRootsScope srs(n_workers);

      CMSParInitialMarkTask tsk(this, &srs, n_workers);
      initialize_sequential_subtasks_for_young_gen_rescan(n_workers);
      // If the total workers is greater than 1, then multiple workers
      // may be used at some time and the initialization has been set
      // such that the single threaded path cannot be used.
      if (workers->total_workers() > 1) {
        workers->run_task(&tsk);
      } else {
        tsk.work(0);
      }
    } else {
      // The serial version.
      CLDToOopClosure cld_closure(&notOlder, true);
      heap->rem_set()->prepare_for_younger_refs_iterate(false); // Not
parallel.

      StrongRootsScope srs(1);

      heap->cms_process_roots(&srs,
                              true,   // young gen as roots
                              GenCollectedHeap::ScanningOption(roots_
scanning_options()),
                              should_unload_classes(),
                              &notOlder,
```

```
                                &cld_closure);
    }
  }

  // Clear mod-union table; it will be dirtied in the prologue of
  // CMS generation per each young generation collection.
  assert(_modUnionTable.isAllClear(),
       "Was cleared in most recent final checkpoint phase"
       " or no bits are set in the gc_prologue before the start of the
next "
       "subsequent marking phase.");

  assert(_ct->cld_rem_set()->mod_union_is_clear(), "Must be");
  // Save the end of the used_region of the constituent generations
  // to be used to limit the extent of sweep in each generation.
  save_sweep_limits();
  verify_overflow_empty();
}
```

```
void CMSParInitialMarkTask::work(uint worker_id) {
  elapsedTimer _timer;
  ResourceMark rm;
  HandleMark   hm;

  // ---------- scan from roots --------------
  _timer.start();
  CMSHeap* heap = CMSHeap::heap();
  ParMarkRefsIntoClosure par_mri_cl(_collector->_span, &(_collector->_
markBitMap));

  // ---------- young gen roots --------------
  {
    work_on_young_gen_roots(&par_mri_cl);
    _timer.stop();
    log_trace(gc, task)("Finished young gen initial mark scan work in
%dth thread: %3.3f sec", worker_id, _timer.seconds());
  }

  // ---------- remaining roots --------------
  _timer.reset();
  _timer.start();

  CLDToOopClosure cld_closure(&par_mri_cl, true);

  heap->cms_process_roots(_strong_roots_scope,
                          false,     // yg was scanned above
                          GenCollectedHeap::ScanningOption(_collector-
>CMSCollector::roots_scanning_options()),
                          _collector->should_unload_classes(),
```

```
                            &par_mri_cl,
                            &cld_closure,
                            &_par_state_string);

  assert(_collector->should_unload_classes()
         || (_collector->CMSCollector::roots_scanning_options() &
GenCollectedHeap::SO_AllCodeCache),
         "if we didn't scan the code cache, we have to be ready to
drop nmethods with expired weak oops");
  _timer.stop();
  log_trace(gc, task)("Finished remaining root initial mark scan work
in %dth thread: %3.3f sec", worker_id, _timer.seconds());
}
```



整个过程比较简单，从 GC Root 出发标记 Old 中的对象，处理完成后借助 BitMap 处理下 Young 区对 Old 区的引用，整个过程基本都比较快，很少会有较大的停顿。

- CMS Final Remark 执行步骤，实现在 `CMSCollector::checkpointRootsFinalWork()` 中，整体代码和步骤如下：

```
void CMSCollector::checkpointRootsFinalWork() {
  GCTraceTime(Trace, gc, phases) tm("checkpointRootsFinalWork", _gc_
timer_cm);

  assert(haveFreelistLocks(), "must have free list locks");
  assert_lock_strong(bitMapLock());

  ResourceMark rm;
  HandleMark   hm;

  CMSHeap* heap = CMSHeap::heap();

  if (should_unload_classes()) {
    CodeCache::gc_prologue();
  }
  assert(haveFreelistLocks(), "must have free list locks");
  assert_lock_strong(bitMapLock());

  heap->ensure_parsability(false);  // fill TLAB's, but no need to
retire them
  // Update the saved marks which may affect the root scans.
  heap->save_marks();
```

```
  print_eden_and_survivor_chunk_arrays();

  {
    if (CMSParallelRemarkEnabled) {
      GCTraceTime(Debug, gc, phases) t("Rescan (parallel)", _gc_timer_
cm);
      do_remark_parallel();
    } else {
      GCTraceTime(Debug, gc, phases) t("Rescan (non-parallel)", _gc_
timer_cm);
      do_remark_non_parallel();
    }
  }
  verify_work_stacks_empty();
  verify_overflow_empty();

  {
    GCTraceTime(Trace, gc, phases) ts("refProcessingWork", _gc_timer_
cm);
    refProcessingWork();
  }
  verify_work_stacks_empty();
  verify_overflow_empty();

  if (should_unload_classes()) {
    CodeCache::gc_epilogue();
  }
  JvmtiExport::gc_epilogue();
  assert(_markStack.isEmpty(), "No grey objects");
  size_t ser_ovflw = _ser_pmc_remark_ovflw + _ser_pmc_preclean_ovflw +
                     _ser_kac_ovflw        + _ser_kac_preclean_ovflw;
  if (ser_ovflw > 0) {
    log_trace(gc)("Marking stack overflow (benign) (pmc_pc=" SIZE_
FORMAT ", pmc_rm=" SIZE_FORMAT ", kac=" SIZE_FORMAT ", kac_preclean="
SIZE_FORMAT ")",
                          _ser_pmc_preclean_ovflw, _ser_pmc_remark_ovflw,
_ser_kac_ovflw, _ser_kac_preclean_ovflw);
    _markStack.expand();
    _ser_pmc_remark_ovflw = 0;
    _ser_pmc_preclean_ovflw = 0;
    _ser_kac_preclean_ovflw = 0;
    _ser_kac_ovflw = 0;
  }
  if (_par_pmc_remark_ovflw > 0 || _par_kac_ovflw > 0) {
     log_trace(gc)("Work queue overflow (benign) (pmc_rm=" SIZE_FORMAT
", kac=" SIZE_FORMAT ")",
                           _par_pmc_remark_ovflw, _par_kac_ovflw);
     _par_pmc_remark_ovflw = 0;
    _par_kac_ovflw = 0;
```

```
  }
   if (_markStack._hit_limit > 0) {
     log_trace(gc)(" (benign) Hit max stack size limit (" SIZE_FORMAT
")",
                           _markStack._hit_limit);
   }
   if (_markStack._failed_double > 0) {
     log_trace(gc)(" (benign) Failed stack doubling (" SIZE_FORMAT "),
current capacity " SIZE_FORMAT,
                           _markStack._failed_double, _markStack.
capacity());
   }
  _markStack._hit_limit = 0;
  _markStack._failed_double = 0;

  if ((VerifyAfterGC || VerifyDuringGC) &&
      CMSHeap::heap()->total_collections() >= VerifyGCStartAt) {
    verify_after_remark();
  }

  _gc_tracer_cm->report_object_count_after_gc(&_is_alive_closure);

  // Change under the freelistLocks.
  _collectorState = Sweeping;
  // Call isAllClear() under bitMapLock
  assert(_modUnionTable.isAllClear(),
      "Should be clear by end of the final marking");
  assert(_ct->cld_rem_set()->mod_union_is_clear(),
      "Should be clear by end of the final marking");
}
```



Final Remark 是最终的第二次标记，这种情况只有在 Background GC 执行了 InitialMarking 步骤的情形下才会执行，如果是 Foreground GC 执行的 Initial-Marking 步骤则不需要再次执行 FinalRemark。Final Remark 的开始阶段与 Init Mark 处理的流程相同，但是后续多了 Card Table 遍历、Reference 实例的清理并将其加入到 Reference 维护的 `pend_list` 中，如果要收集元数据信息，还要清理 SystemDictionary、CodeCache、SymbolTable、StringTable 等组件中不再使用的资源。

4.6.3 策略

知道了两个 STW 过程执行流程，我们分析解决就比较简单了，由于大部分问题都出在 Final Remark 过程，这里我们也拿这个场景来举例，主要步骤：

- **【方向】**观察详细 GC 日志，找到出问题时 Final Remark 日志，分析下 Reference 处理和元数据处理 real 耗时是否正常，详细信息需要通过 `-XX:+PrintReferenceGC` 参数开启。**基本在日志里面就能定位到大概是哪个方向出了问题，耗时超过 10% 的就需要关注**。

```
2019-02-27T19:55:37.920+0800: 516952.915: [GC (CMS Final Remark)
516952.915: [ParNew516952.939: [SoftReference, 0 refs, 0.0003857
secs]516952.939: [WeakReference, 1362 refs, 0.0002415 secs]516952.940:
[FinalReference, 146 refs, 0.0001233 secs]516952.940: [PhantomReference,
0 refs, 57 refs, 0.0002369 secs]516952.940: [JNI Weak Reference,
0.0000662 secs]
[class unloading, 0.1770490 secs]516953.329: [scrub symbol table,
0.0442567 secs]516953.373: [scrub string table, 0.0036072 secs][1 CMS-
remark: 1638504K(2048000K)] 1667558K(4352000K), 0.5269311 secs] [Times:
user=1.20 sys=0.03, real=0.53 secs]
```

- **【根因】**有了具体的方向我们就可以进行深入的分析，一般来说最容易出问题的地方就是 Reference 中的 FinalReference 和元数据信息处理中的 scrub symbol table 两个阶段，想要找到具体问题代码就需要内存分析工具 MAT 或 JProfiler 了，注意要 dump 即将开始 CMS GC 的堆。在用 MAT 等工具前也可以先用命令行看下对象 Histogram，有可能直接就能定位问题。
  - 对 FinalReference 的分析主要观察 `java.lang.ref.Finalizer` 对象的 dominator tree，找到泄漏的来源。经常会出现问题的几个点有 Socket 的 `SocksSocketImpl`、Jersey 的 `ClientRuntime`、MySQL 的 `ConnectionImpl` 等等。
  - scrub symbol table 表示清理元数据符号引用耗时，符号引用是 Java 代码被编译成字节码时，方法在 JVM 中的表现形式，生命周期一般与 Class 一致，当 `_should_unload_classes` 被设置为 true 时在 `CMSCollector::refProcessingWork()` 中与 Class Unload、String Table 一起被处理。

```
if (should_unload_classes()) {
    {
      GCTraceTime(Debug, gc, phases) t("Class Unloading", _gc_timer_cm);

      // Unload classes and purge the SystemDictionary.
      bool purged_class = SystemDictionary::do_unloading(_gc_timer_cm);

      // Unload nmethods.
      CodeCache::do_unloading(&_is_alive_closure, purged_class);

      // Prune dead klasses from subklass/sibling/implementor lists.
      Klass::clean_weak_klass_links(purged_class);
    }

    {
      GCTraceTime(Debug, gc, phases) t("Scrub Symbol Table", _gc_timer_cm);
      // Clean up unreferenced symbols in symbol table.
      SymbolTable::unlink();
    }

    {
      GCTraceTime(Debug, gc, phases) t("Scrub String Table", _gc_timer_
cm);
      // Delete entries for dead interned strings.
      StringTable::unlink(&_is_alive_closure);
    }
  }
```

- 【**策略**】知道 GC 耗时的根因就比较好处理了，这种问题不会大面积同时爆发，不过有很多时候单台 STW 的时间会比较长，如果业务影响比较大，及时摘掉流量，具体后续优化策略如下：
  - FinalReference：找到内存来源后通过优化代码的方式来解决，如果短时间无法定位可以增加 `-XX:+ParallelRefProcEnabled` 对 Reference 进行并行处理。
  - symbol table：观察 MetaSpace 区的历史使用峰值，以及每次 GC 前后的回收情况，一般没有使用动态类加载或者 DSL 处理等，MetaSpace 的使用率上不会有什么变化，这种情况可以通过 `-XX:-CMSClassUnloadingEnabled` 来避免 MetaSpace 的处理，JDK8 会默认开启 CMSClassUnloadingEnabled，这会使得 CMS 在 CMS-Remark 阶段尝试进行类的卸载。

4.6.4 小结

正常情况进行的 Background CMS GC，出现问题基本都集中在 Reference 和 Class 等元数据处理上，在 Reference 类的问题处理方面，不管是 FinalReference，还是 SoftReference、WeakReference 核心的手段就是找准时机 dump 快照，然后用内存分析工具来分析。Class 处理方面目前除了关闭类卸载开关，没有太好的方法。

在 G1 中同样有 Reference 的问题，可以观察日志中的 Ref Proc，处理方法与 CMS 类似。

4.7 场景七：内存碎片 & 收集器退化

4.7.1 现象

并发的 CMS GC 算法，退化为 Foreground 单线程串行 GC 模式，STW 时间超长，有时会长达十几秒。其中 CMS 收集器退化后单线程串行 GC 算法有两种：

- 带压缩动作的算法，称为 MSC，上面我们介绍过，使用标记 - 清理 - 压缩，单线程全暂停的方式，对整个堆进行垃圾收集，也就是真正意义上的 Full GC，暂停时间要长于普通 CMS。
- 不带压缩动作的算法，收集 Old 区，和普通的 CMS 算法比较相似，暂停时间相对 MSC 算法短一些。

4.7.2 原因

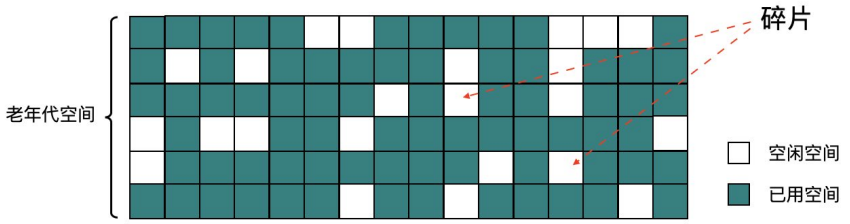
CMS 发生收集器退化主要有以下几种情况：

晋升失败（Promotion Failed）

顾名思义，晋升失败就是指在进行 Young GC 时，Survivor 放不下，对象只能放入 Old，但此时 Old 也放不下。直觉上乍一看这种情况可能会经常发生，但其实因为有 concurrentMarkSweepThread 和担保机制的存在，发生的条件是很苛刻的，除非是短时间将 Old 区的剩余空间迅速填满，例如上文中说的动态年龄判断导

致的过早晋升（见下文的增量收集担保失败）。另外还有一种情况就是内存碎片导致的 Promotion Failed，Young GC 以为 Old 有足够的空间，结果到分配时，晋级的大对象找不到连续的空间存放。

使用 CMS 作为 GC 收集器时，运行过一段时间的 Old 区如下图所示，清除算法导致内存出现多段的不连续，出现大量的内存碎片。



碎片带来了两个问题：

- **空间分配效率较低**：上文已经提到过，如果是连续的空间 JVM 可以通过使用 pointer bumping 的方式来分配，而对于这种有大量碎片的空闲链表则需要逐个访问 freelist 中的项来访问，查找可以存放新建对象的地址。
- **空间利用效率变低**：Young 区晋升的对象大小大于了连续空间的大小，那么将会触发 Promotion Failed ，即使整个 Old 区的容量是足够的，但由于其不连续，也无法存放新对象，也就是本文所说的问题。

增量收集担保失败

分配内存失败后，会判断统计得到的 Young GC 晋升到 Old 的平均大小，以及当前 Young 区已使用的大小也就是最大可能晋升的对象大小，是否大于 Old 区的剩余空间。只要 CMS 的剩余空间比前两者的任意一者大，CMS 就认为晋升还是安全的，反之，则代表不安全，不进行 Young GC，直接触发 Full GC。

显式 GC

这种情况参见场景二。

并发模式失败（Concurrent Mode Failure）

最后一种情况，也是发生概率较高的一种，在 GC 日志中经常能看到 Concurrent Mode Failure 关键字。这种是由于并发 Background CMS GC 正在执行，同时又有 Young GC 晋升的对象要放入到了 Old 区中，而此时 Old 区空间不足造成的。

为什么 CMS GC 正在执行还会导致收集器退化呢？主要是由于 CMS 无法处理浮动垃圾（Floating Garbage）引起的。CMS 的并发清理阶段，Mutator 还在运行，因此不断有新的垃圾产生，而这些垃圾不在这次清理标记的范畴里，无法在本次 GC 被清除掉，这些就是浮动垃圾，除此之外在 Remark 之前那些断开引用脱离了读写屏障控制的对象也算浮动垃圾。所以 Old 区回收的阈值不能太高，否则预留的内存空间很可能不够，从而导致 Concurrent Mode Failure 发生。

4.7.3 策略

分析到具体原因后，我们就可以针对性解决了，具体思路还是从根因出发，具体解决策略：

- **内存碎片：** 通过配置 `-XX:UseCMSCompactAtFullCollection=true` 来控制 Full GC 的过程中是否进行空间的整理（默认开启，注意是 Full GC，不是普通 CMS GC），以及 `-XX: CMSFullGCsBeforeCompaction=n` 来控制多少次 Full GC 后进行一次压缩。
- **增量收集：** 降低触发 CMS GC 的阈值，即参数 `-XX:CMSInitiatingOccupancyFraction` 的值，让 CMS GC 尽早执行，以保证有足够的连续空间，也减少 Old 区空间的使用大小，另外需要使用 `-XX:+UseCMSInitiatingOccupancyOnly` 来配合使用，不然 JVM 仅在第一次使用设定值，后续则自动调整。
- **浮动垃圾：** 视情况控制每次晋升对象的大小，或者缩短每次 CMS GC 的时间，必要时可调节 NewRatio 的值。另外就是使用 `-XX:+CMSScavengeBeforeRemark` 在过程中提前触发一次 Young GC，防止后续晋升过多对象。

4.7.4 小结

正常情况下触发并发模式的 CMS GC，停顿非常短，对业务影响很小，但 CMS GC 退化后，影响会非常大，建议发现一次后就彻底根治。只要能定位到内存碎片、浮动垃圾、增量收集相关等具体产生原因，还是比较好解决的，关于内存碎片这块，如果 `-XX:CMSFullGCsBeforeCompaction` 的值不好选取的话，可以使用 `-XX:PrintFLSStatistics` 来观察内存碎片率情况，然后再设置具体的值。

最后就是在编码的时候也要避免需要连续地址空间的大对象的产生，如过长的字符串，用于存放附件、序列化或反序列化的 byte 数组等，还有就是过早晋升问题尽量在爆发问题前就避免掉。

4.8 场景八：堆外内存 OOM

4.8.1 现象

内存使用率不断上升，甚至开始使用 SWAP 内存，同时可能出现 GC 时间飙升，线程被 Block 等现象，**通过 top 命令发现 Java 进程的 RES 甚至超过了 `-Xmx` 的大小**。出现这些现象时，基本可以确定是出现了堆外内存泄漏。

4.8.2 原因

JVM 的堆外内存泄漏，主要有两种的原因：

- 通过 `UnSafe#allocateMemory`，`ByteBuffer#allocateDirect` 主动申请了堆外内存而没有释放，常见于 NIO、Netty 等相关组件。
- 代码中有通过 JNI 调用 Native Code 申请的内存没有释放。

4.8.3 策略

哪种原因造成的堆外内存泄漏?

首先，我们需要确定是哪种原因导致的堆外内存泄漏。这里可以使用 NMT（Native-MemoryTracking）进行分析。在项目中添加 `-XX:NativeMemoryTracking=detail` JVM 参数后重启项目（需要注意的是，打开 NMT 会带来 5%~10% 的性能损

耗)。使用命令 `jcmd pid VM.native_memory detail` 查看内存分布。重点观察 total 中的 committed，因为 jcmd 命令显示的内存包含堆内内存、Code 区域、通过 `Unsafe.allocateMemory` 和 `DirectByteBuffer` 申请的内存，但是不包含其他 Native Code（C 代码）申请的堆外内存。

如果 total 中的 committed 和 top 中的 RES 相差不大，则应为主动申请的堆外内存未释放造成的，如果相差较大，则基本可以确定是 JNI 调用造成的。

原因一：主动申请未释放

JVM 使用 `-XX:MaxDirectMemorySize=size` 参数来控制可申请的堆外内存的最大值。在 Java8 中，如果未配置该参数，默认和 `-Xmx` 相等。

NIO 和 Netty 都会取 `-XX:MaxDirectMemorySize` 配置的值，来限制申请的堆外内存的大小。NIO 和 Netty 中还有一个计数器字段，用来计算当前已申请的堆外内存大小，NIO 中是 `java.nio.Bits#totalCapacity`、Netty 中 `io.netty.util.internal.PlatformDependent#DIRECT_MEMORY_COUNTER`。

当申请堆外内存时，NIO 和 Netty 会比较计数器字段和最大值的大小，如果计数器的值超过了最大值的限制，会抛出 OOM 的异常。

NIO 中是：`OutOfMemoryError: Direct buffer memory`。

Netty 中是：`OutOfDirectMemoryError: failed to allocate capacity byte(s) of direct memory (used: usedMemory , max: DIRECT_MEMORY_LIMIT )`。

我们可以检查代码中是如何使用堆外内存的，NIO 或者是 Netty，通过反射，获取到对应组件中的计数器字段，并在项目中对该字段的数值进行打点，即可准确地监控到这部分堆外内存的使用情况。

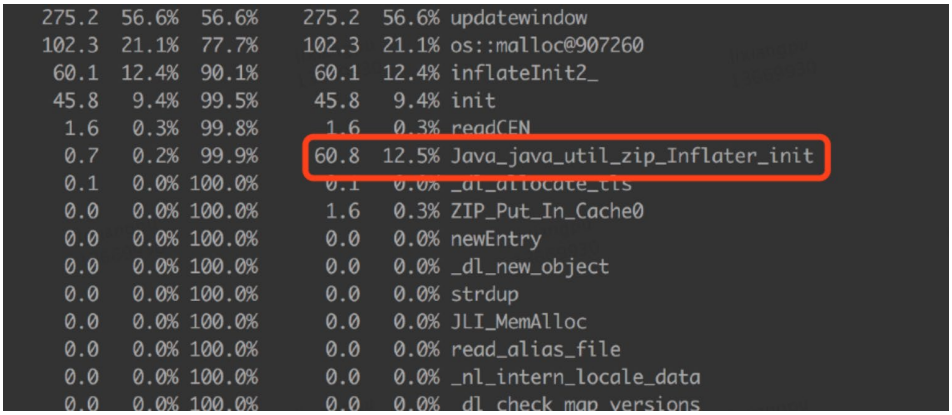
此时，可以通过 Debug 的方式确定使用堆外内存的地方是否正确执行了释放内存的代码。另外，需要检查 JVM 的参数是否有 `-XX:+DisableExplicitGC` 选项，如

果有就去掉，因为该参数会使 System.gc 失效。（场景二：显式 GC 的去与留）

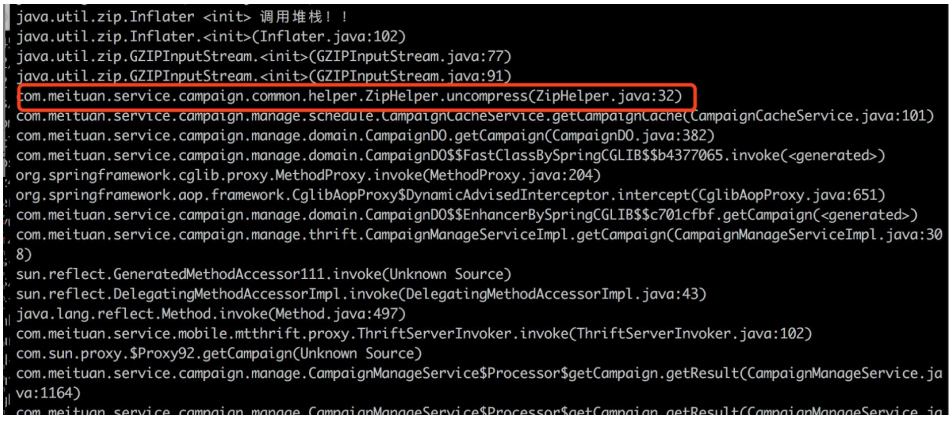
原因二：通过 JNI 调用的 Native Code 申请的内存未释放

这种情况排查起来比较困难，我们可以通过 Google perftools + Btrace 等工具，帮助我们分析出问题的代码在哪里。

gperftools 是 Google 开发的一款非常实用的工具集，它的原理是在 Java 应用程序运行时，当调用 malloc 时换用它的 libtcmalloc.so，这样就能对内存分配情况做一些统计。我们使用 gperftools 来追踪分配内存的命令。如下图所示，通过 gperftools 发现 `Java_java_util_zip_Inflater_init` 比较可疑。



接下来可以使用 Btrace，尝试定位具体的调用栈。Btrace 是 Sun 推出的一款 Java 追踪、监控工具，可以在不停机的情况下对线上的 Java 程序进行监控。如下图所示，通过 Btrace 定位出项目中的 `ZipHelper` 在频繁调用 `GZIPInputStream`，在堆外内存分配对象。



```
java.util.zip.Inflater <init> 调用堆栈！！
java.util.zip.Inflater.<init>(Inflater.java:102)
java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:77)
java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:91)
com.meituan.service.campaign.common.helper.ZipHelper.uncompress(ZipHelper.java:32)
com.meituan.service.campaign.manage.schedule.CampaignCacheService.getCampaignCache(CampaignCacheService.java:101)
com.meituan.service.campaign.manage.domain.CampaignDO.getCampaign(CampaignDO.java:382)
com.meituan.service.campaign.manage.domain.CampaignDO$$FastClassBySpringCGLIB$$b4377065.invoke(<generated>)
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:651)
com.meituan.service.campaign.manage.domain.CampaignDO$$EnhancerBySpringCGLIB$$c701cfbf.getCampaign(<generated>)
com.meituan.service.campaign.manage.thrift.CampaignManageServiceImpl.getCampaign(CampaignManageServiceImpl.java:308)
sun.reflect.GeneratedMethodAccessor111.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:497)
com.meituan.service.mobile.mtthrift.proxy.ThriftServerInvoker.invoke(ThriftServerInvoker.java:102)
com.sun.proxy.$Proxy92.getCampaign(Unknown Source)
com.meituan.service.campaign.manage.CampaignManageService$Processor$getCampaign.getResult(CampaignManageService.java:1164)
```

最终定位到是，项目中对 `GIPInputStream` 的使用错误，没有正确的 close()。

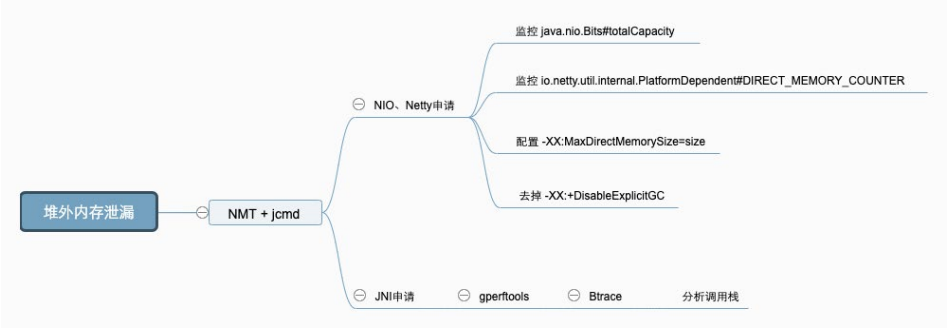


```
26 @    public static String uncompress(String str) throws Exception {
27          if (str == null || str.length() == 0) {
28              return str;
29          }
30          ByteArrayOutputStream out = new ByteArrayOutputStream();
31          ByteArrayInputStream in = new ByteArrayInputStream(str.getBytes( charsetName: "ISO-8859-1"));
32          GZIPInputStream gunzip = new GZIPInputStream(in);
33          byte[] buffer = new byte[1024];
34          int n;
35          while ((n = gunzip.read(buffer)) >= 0) {
36              out.write(buffer,  off: 0, n);
37          }
38          return out.toString();
39      }
40  }
```

除了项目本身的原因，还可能有外部依赖导致的泄漏，如 Netty 和 Spring Boot，详细情况可以学习下这两篇文章，Spring Boot 引起的“堆外内存泄漏”排查及经验总结、Netty 堆外内存泄露排查盛宴。

4.8.4 小结

首先可以使用 NMT + jcmd 分析泄漏的堆外内存是哪里申请，确定原因后，使用不同的手段，进行原因定位。



4.9 场景九：JNI 引发的 GC 问题

4.9.1 现象

在 GC 日志中，出现 GC Cause 为 GCLocker Initiated GC。

```
2020-09-23T16:49:09.727+0800: 504426.742: [GC (GCLocker Initiated GC)
504426.742: [ParNew (promotion failed): 209716K->6042K(1887488K),
0.0843330 secs] 1449487K->1347626K(3984640K), 0.0848963 secs] [Times:
user=0.19 sys=0.00, real=0.09 secs]
2020-09-23T16:49:09.812+0800: 504426.827: [Full GC (GCLocker Initiated
GC) 504426.827: [CMS: 1341583K->419699K(2097152K), 1.8482275 secs]
1347626K->419699K(3984640K), [Metaspace: 297780K->297780K(1329152K)],
1.8490564 secs] [Times: user=1.62 sys=0.20, real=1.85 secs]
```

4.9.2 原因

JNI（Java Native Interface）意为 Java 本地调用，它允许 Java 代码和其他语言写的 Native 代码进行交互。

JNI 如果需要获取 JVM 中的 String 或者数组，有两种方式：

- 拷贝传递。
- 共享引用（指针），性能更高。

由于 Native 代码直接使用了 JVM 堆区的指针，如果这时发生 GC，就会导致数据错误。因此，在发生此类 JNI 调用时，禁止 GC 的发生，同时阻止其他线程进入 JNI 临界区，直到最后一个线程退出临界区时触发一次 GC。

GC Locker 实验：

```
public class GCLockerTest {

  static final int ITERS = 100;
  static final int ARR_SIZE =  10000;
  static final int WINDOW = 10000000;

  static native void acquire(int[] arr);
  static native void release(int[] arr);

  static final Object[] window = new Object[WINDOW];

  public static void main(String... args) throws Throwable {
    System.loadLibrary("GCLockerTest");
    int[] arr = new int[ARR_SIZE];

    for (int i = 0; i < ITERS; i++) {
      acquire(arr);
      System.out.println("Acquired");
      try {
        for (int c = 0; c < WINDOW; c++) {
          window[c] = new Object();
        }
      } catch (Throwable t) {
        // omit
      } finally {
        System.out.println("Releasing");
        release(arr);
      }
    }
  }
}
```

```
#include <jni.h>
#include "GCLockerTest.h"

static jbyte* sink;

JNIEXPORT void JNICALL Java_GCLockerTest_acquire(JNIEnv* env, jclass
klass, jintArray arr) {
sink = (*env)->GetPrimitiveArrayCritical(env, arr, 0);
}

JNIEXPORT void JNICALL Java_GCLockerTest_release(JNIEnv* env, jclass
klass, jintArray arr) {
(*env)->ReleasePrimitiveArrayCritical(env, arr, sink, 0);
}
```

运行该 JNI 程序，可以看到发生的 GC 都是 GCLocker Initiated GC，并且注意在“Acquired”和“Released”时不可能发生 GC。

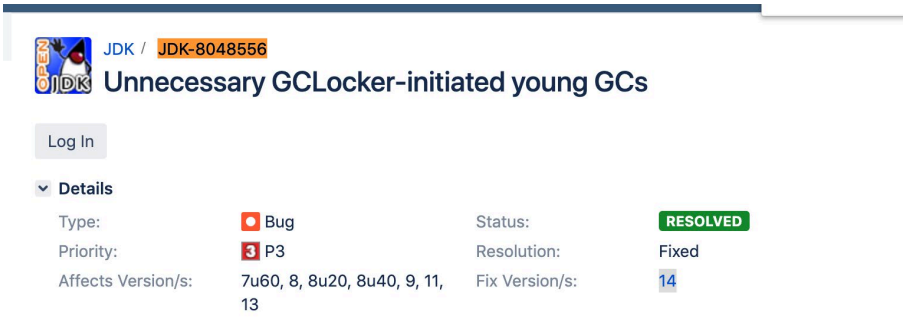
```
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
[GC (GCLocker Initiated GC)  1801127K->1269053K(4126208K), 0.1635153 secs]
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
[GC (GCLocker Initiated GC)  1942063K->1401284K(4126208K), 0.1379408 secs]
```

GC Locker 可能导致的不良后果有：

- 如果此时是 Young 区不够 Allocation Failure 导致的 GC，由于无法进行 Young GC，会将对象直接分配至 Old 区。
- 如果 Old 区也没有空间了，则会等待锁释放，导致线程阻塞。
- 可能触发额外不必要的 Young GC，JDK 有一个 Bug，有一定的几率，本来只该触发一次 GCLocker Initiated GC 的 Young GC，实际发生了一次 Allocation Failure GC 又紧接着一次 GCLocker Initiated GC。是因为 GCLocker Initiated GC 的属性被设为 full，导致两次 GC 不能收敛。

4.9.3 策略

- 添加 `-XX+PrintJNIGCStalls` 参数，可以打印出发生 JNI 调用时的线程，进一步分析，找到引发问题的 JNI 调用。
- JNI 调用需要谨慎，不一定可以提升性能，反而可能造成 GC 问题。
- 升级 JDK 版本到 14，避免 JDK-8048556 导致的重复 GC。



4.9.4 小结

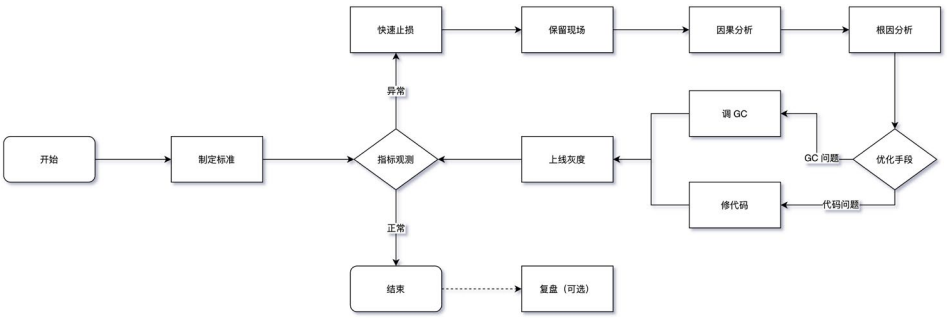
JNI 产生的 GC 问题较难排查，需要谨慎使用。

5. 总结

在这里，我们把整个文章内容总结一下，方便大家整体地理解回顾。

5.1 处理流程（SOP）

下图为整体 GC 问题普适的处理流程，重点的地方下面会单独标注，其他的基本都是标准处理流程，此处不再赘述，最后在整个问题都处理完之后有条件的话建议做一下复盘。



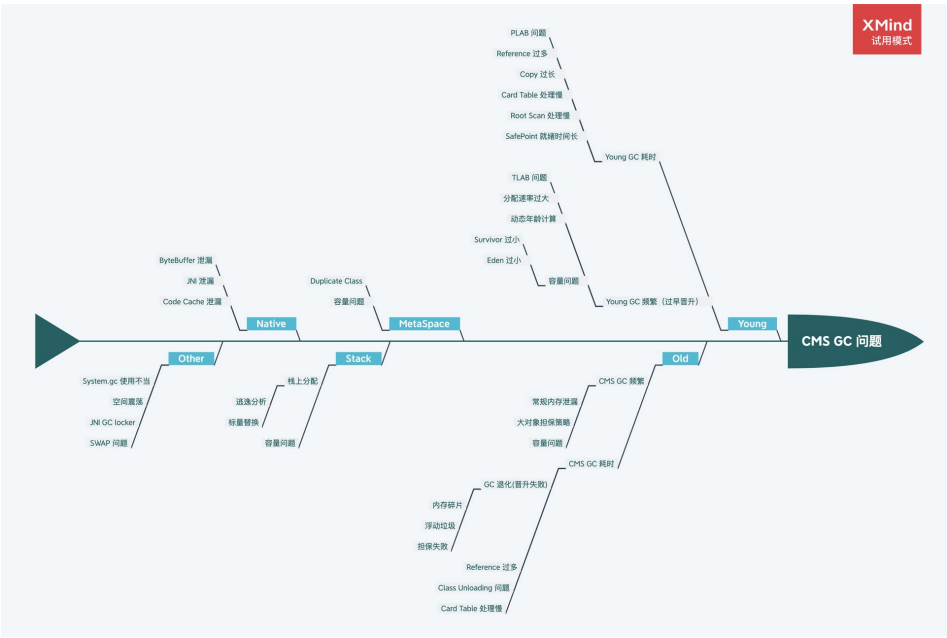
- **制定标准：**这块内容其实非常重要，但大部分系统都是缺失的，笔者过往面试的同学中只有不到一成的同学能给出自己的系统 GC 标准到底什么样，其他的都是用的统一指标模板，缺少预见性，具体指标制定可以参考 3.1 中的内容，

需要结合应用系统的 TP9999 时间和延迟、吞吐量等设定具体的指标，而不是被问题驱动。

- **保留现场：**目前线上服务基本都是分布式服务，某个节点发生问题后，如果条件允许一定不要直接操作重启、回滚等动作恢复，优先通过摘掉流量的方式来恢复，这样我们可以将堆、栈、GC 日志等关键信息保留下来，不然错过了定位根因的时机，后续解决难度将大大增加。当然除了这些，应用日志、中间件日志、内核日志、各种 Metrics 指标等对问题分析也有很大帮助。
- **因果分析：**判断 GC 异常与其他系统指标异常的因果关系，可以参考笔者在 3.2 中介绍的时序分析、概率分析、实验分析、反证分析等 4 种因果分析法，避免在排查过程中走入误区。
- **根因分析：**确实是 GC 的问题后，可以借助上文提到的工具并通过 5 why 根因分析法以及跟第三节中的九种常见的场景进行逐一匹配，或者直接参考下文的根因鱼骨图，找出问题发生根因，最后再选择优化手段。

5.2 根因鱼骨图

送上一张问题根因鱼骨图，一般情况下我们在处理一个 GC 问题时，只要能定位到问题的“病灶”，有的放矢，其实就相当于解决了 80%，如果在某些场景下不太好定位，大家可以借助这种根因分析图通过**排除法**去定位。



5.3 调优建议

- **Trade Off：**与 CAP 注定要缺一角一样，GC 优化要在延迟（Latency）、吞吐量（Throughput）、容量（Capacity）三者之间进行权衡。
- **最终手段：**GC 发生问题不是一定要对 JVM 的 GC 参数进行调优，大部分情况下是通过 GC 的情况找出一些业务问题，切记上来就对 GC 参数进行调整，当然有明确配置错误的场景除外。
- **控制变量：**控制变量法是在蒙特卡洛（Monte Carlo）方法中用于减少方差的一种技术方法，我们调优的时候尽量也要使用，每次调优过程尽可能只调整一个变量。
- **善用搜索：**理论上 99.99% 的 GC 问题基本都被遇到了，我们要学会使用搜索引擎的高级技巧，重点关注 StackOverFlow、Github 上的 Issue、以及各种论坛博客，先看看其他人是怎么解决的，会让解决问题事半功倍。能看到这篇文章，你的搜索能力基本过关了 ~
- **调优重点：**总体上来讲，我们开发的过程中遇到的问题类型也基本都符合正态分布，太简单或太复杂的基本遇到的概率很低，笔者这里将中间最重要的三个

场景添加了“*”标识，希望阅读完本文之后可以观察下自己负责的系统，是否存在上述问题。

- **GC 参数：**如果堆、栈确实无法第一时间保留，一定要保留 GC 日志，这样我们最起码可以看到 GC Cause，有一个大概的排查方向。关于 GC 日志相关参数，最基本的 `-XX:+HeapDumpOnOutOfMemoryError` 等一些参数就不再提了，笔者建议添加以下参数，可以提高我们分析问题的效率。

分类	参数	作用
基本参数	`-XX:+PrintGCDetails`、`-XX:+PrintGCDateStamps`、`-XX:+PrintGCTimeStamps`	GC 日志的基本参数
时间相关	`-XX:+PrintGCApplicationConcurrentTime`、`-XX:+PrintGCApplicationStoppedTime`	详细步骤的并行时间，STW 时间等等
年龄相关	`-XX:+PrintTenuringDistribution`	可以观察 GC 前后的对象年龄分布，方便发现过早晋升问题
空间变化	`-XX:+PrintHeapAtGC`	各个空间在 GC 前后的回收情况，非常详细
引用相关	`-XX:+PrintReferenceGC`	观察系统的软引用，弱引用，虚引用等回收情况

- **其他建议：**上文场景中没有提到，但是对 GC 性能也有提升的一些建议。
  - **主动式 GC：**也有另开生面的做法，通过监控手段监控观测 Old 区的使用情况，即将到达阈值时将应用服务摘掉流量，手动触发一次 Major GC，减少 CMS GC 带来的停顿，但随之系统的健壮性也会减少，如非必要不建议引入。
  - **禁用偏向锁：**偏向锁在只有一个线程使用到该锁的时候效率很高，但是在竞争激烈情况会升级成轻量级锁，此时就需要先**消除偏向锁，这个过程是 STW** 的。如果每个同步资源都走这个升级过程，开销会非常大，所以在已知并发激烈的前提下，一般会禁用偏向锁 `-XX:-UseBiasedLocking` 来提高性能。
  - **虚拟内存：**启动初期有些操作系统（例如 Linux）并没有真正分配物理内存给 JVM ，而是在虚拟内存中分配，使用的时候才会在物理内存中分配内存页，这样也会导致 GC 时间较长。这种情况可以添加 `-XX:+AlwaysPreTouch` 参数，让 VM 在 commit 内存时跑个循环来强制保证申请的内存真的 commit，避免运行时触发缺页异常。在一些大内存的场景下，有时候能将前几次的 GC 时间降一个数量级，但是添加这个参数后，启动的过程可能会变慢。

6. 写在最后

最后，再说笔者个人的一些小建议，遇到一些 GC 问题，如果有精力，一定要探本穷源，找出最深层次的原因。另外，在这个信息泛滥的时代，有一些被“奉为圭臬”的经验可能都是错误的，尽量养成看源码的习惯，有一句话说到“源码面前，了无秘密”，也就意味着遇到搞不懂的问题，我们可以从源码中一窥究竟，某些场景下确有奇效。但也不是只靠读源码来学习，如果硬啃源码但不理会其背后可能蕴含的理论基础，那很容易“捡芝麻丢西瓜”，“只见树木，不见森林”，让“了无秘密”变成了一句空话，我们还是要结合一些实际的业务场景去针对性地学习。

你的时间在哪里，你的成就就会在哪里。笔者也是在前两年才开始逐步地在 GC 方向上不断深入，查问题、看源码、做总结，每个 Case 形成一个小的闭环，目前初步摸到了 GC 问题处理的一些门道，同时将经验总结应用于生产环境实践，慢慢地形成一个良性循环。

本篇文章主要是介绍了 CMS GC 的一些常见场景分析，另外一些，如 CodeCache 问题导致 JIT 失效、SafePoint 就绪时间长、Card Table 扫描耗时等问题不太常见就没有花太多篇幅去讲解。Java GC 是在“分代”的思想下内卷了很多年才突破到了“分区”，目前在美团也已经开始使用 G1 来替换使用了多年的 CMS，虽然在小的堆方面 G1 还略逊色于 CMS，但这是一个趋势，短时间无法升级到 ZGC，所以未来遇到的 G1 的问题可能会逐渐增多。目前已经收集到 Remember Set 粗化、Humongous 分配、Ergonomics 异常、Mixed GC 中 Evacuation Failure 等问题，除此之外也会给出 CMS 升级到 G1 的一些建议，接下来笔者将继续完成这部分文章整理，敬请期待。

“防火”永远要胜于“救火”，**不放过任何一个异常的小指标**（一般来说，任何**不平滑的曲线**都是值得怀疑的），就有可能避免一次故障的发生。作为 Java 程序员基本都会遇到一些 GC 的问题，独立解决 GC 问题是我们必须迈过的一道坎。开篇中也提到过 GC 作为经典的技术，非常值得我们学习，一些 GC 的学习材料，如《The Garbage Collection Handbook》《深入理解 Java 虚拟机》等也是常读常新，赶紧

动起来，苦练 GC 基本功吧。

最后的最后，再多啰嗦一句，目前所有 GC 调优相关的文章，第一句讲的就是“不要过早优化”，使得很多同学对 GC 优化望而却步。在这里笔者提出不一样的观点，熵增定律（在一个孤立系统里，如果没有外力做功，其总混乱度（即熵）会不断增大）在计算机系统同样适用，**如果不主动做功使熵减，系统终究会脱离你的掌控**，在我们对业务系统和 GC 原理掌握得足够深的时候，可以放心大胆地做优化，因为我们基本可以预测到每一个操作的结果，放手一搏吧，少年！

7. 参考资料

[1]《ガベージコレクションのアルゴリズムと実装》中村 成洋 / 相川 光
[2]《The Garbage Collection Handbook》Richard Jones/ Antony Hosking / Eliot Moss
[3]《深入理解 Java 虚拟机（第 3 版）》周志明
[4]《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》
[5]《Shipilev One Page Blog》Shipilëv
[6] https://openjdk.java.net/projects/jdk/15/
[7] https://jcp.org/en/home/index
[8]《A Generational Mostly-concurrent Garbage Collector》Tony Printezis / David Detlefs
[9]《Java Memory Management White Paper》
[10]《Stuff Happens：Understanding Causation in Policy and Strategy》AA Hill

8. 作者简介

新宇：2015 年加入美团，到店住宿门票业务开发工程师。
湘铭：2018 年加入美团，到店客户平台开发工程师。
祥璞：2018 年加入美团，到店客户平台开发工程师。

9. 招聘信息

美团到店事业群住宿门票数据智能组诚招小伙伴，从供、控、选、售等层面全方位提升业务竞争力，十万级 QPS 处理，亿级数据分析，完整业务闭环，目前有海量 HC，有兴趣的请将邮件发送至 hezhiming@meituan.com，我们会在第一时间与你联系。

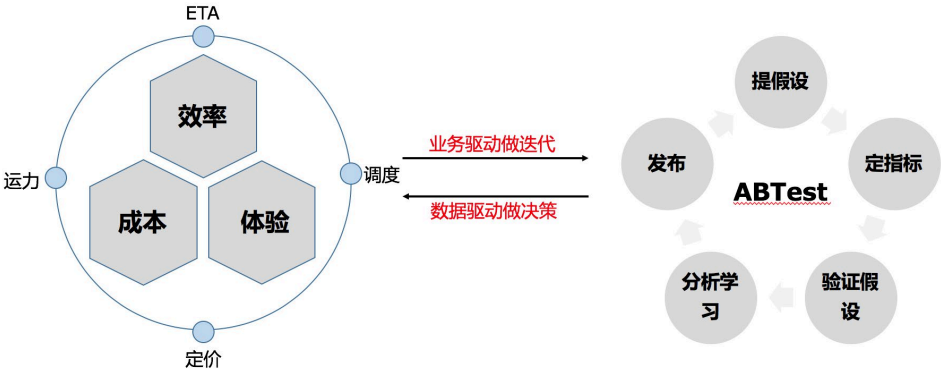
美团配送 A/B 评估体系建设实践

作者：王鹏　启政　连恒

2019 年 5 月 6 日，美团点评正式推出新品牌“美团配送”，发布了美团配送新愿景：“每天完成一亿次值得信赖的配送服务，成为不可或缺的生活基础设施。”现在，美团配送已经服务于全国 400 多万商家和 4 亿多用户，覆盖 2800 余座市县，日活跃骑手超过 70 万人，成为全球领先的分钟级配送网络。

即时配送的三要素是“效率”、“成本”、“体验”，通过精细化的策略迭代来提升效率，降低成本，提高体验，不断地扩大规模优势，从而实现正向循环。但是，策略的改变，不是由我们随便“拍脑袋”得出，而是一种建立在数据基础上的思维方式，数据反馈会告诉我们做的好不好，哪里有问题，以及衡量可以带来多少确定性的增长。而 A/B-test 就是我们精细化迭代的一个“利器”，通过为同一个迭代目标制定两个或多个版本的方案，在同一时间维度，让组成成分相同（或相似）的 A/B 群组分别采用这些版本，然后收集各群组的体验数据和业务数据，最后分析、评估出最好的版本，帮助我们作出正确的决策，使迭代朝着更好的方向去演进。基于此，构建一个适用于配送业务的 A/B 平台就应运而生了。

1. A/B 平台简介

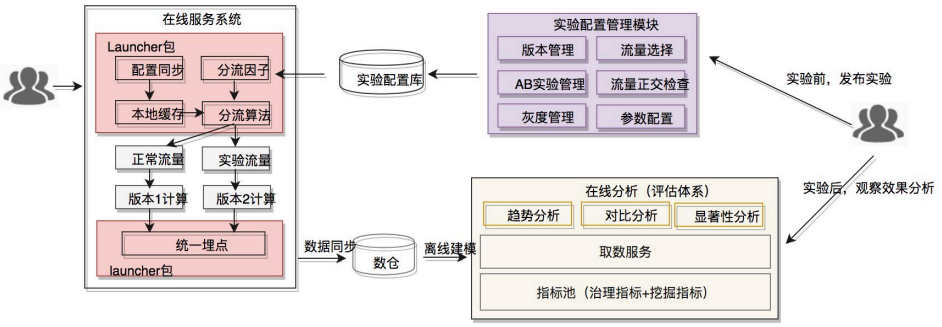


如上图所示，A/B 实验可以看作一个“无尽”的学习环，我们通过提出假设、定义成功指标、检验假设（A/B 实验）、分析学习、发布、建立另一个假设，这就形成一个完整的闭环，通过多轮实验迭代，使策略趋于更优。基于上述对 A/B 实验划分的 5 个步骤，我们将 A/B 实验的完整生命周期分为三个阶段：

- 实验前，提出该实验假设，定义实验成功的指标，确定分流策略；
- 实验中，即验证假设的阶段，根据配置阶段的分流策略进行分流和埋点上报；
- 实验后，进行实验分析与学习，并基于实验报告决定是否发布。



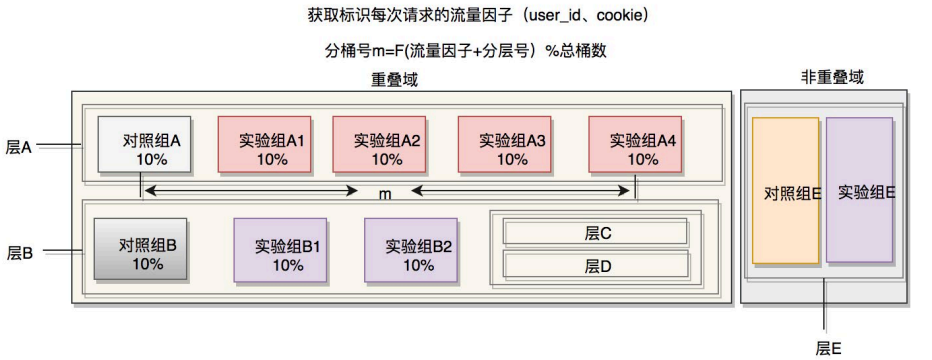
按照功能划分，我们将 A/B 平台分为三个模块，实验配置管理模块、分流以及埋点上报模块和在线分析模块，分别对应于 A/B 实验生命周期的实验前、实验中和实验后三个阶段。在实验配置模块，用户可以基于实验前提出的假设、定义的成功指标快速创建实验，并基于特定的分流策略完成分流配置；分流以及埋点上报模块，提供 JAR 包接入的形式，异步获取实验配置进行本地分流计算和埋点上报；在线分析模块，依据用户在实验配置管理模块选取的用于说明实验效果的指标、分流埋点上报模块记录的日志，自动地产生各实验的实验报告，供实验观察者使用，然后根据实验效果帮助他们作出正确的决策。具体流程如下图所示：



2. 为什么要强调评估体系建设

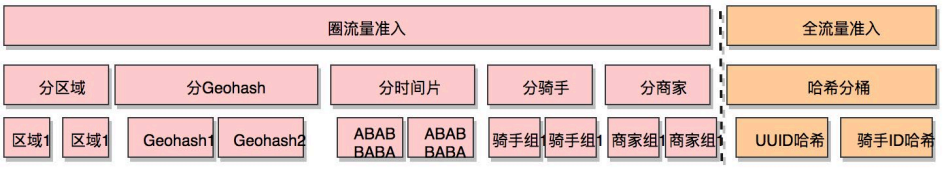
2.1 分流业务场景需要

业界的 A/B 平台建设基本以《Overlapping Experiment Infrastructure: More, Better, Faster Experimentation》这篇论文为蓝本进行展开，引入分层模型以及在分流算法中加入层编号因子来解决“流量饥饿”和“正交”问题，并且通过引入域的概念，支持域和层之间的相互嵌套，使分层实验模型更加灵活，进而满足多种场景下的 A/B 诉求。如下图所示，将流量通过 Hash 取模的方式即可实现流量的均匀划分。



这种是面向 C 端用户进行流量选择的传统 A/B 实验，采用上述的分流方式基于这样的假设：参与实验的流量因子是相互独立的、随机的，服从独立同分布。但是，配送业务场景下的 A/B 实验，涉及到用户、骑手、商家三端，请求不独立，策略之间相互影响并且受线下因素影响较大。传统 A/B 实验的分流方式，无法保证分出的两个群组

实验组和对照组的流量都是无差别的，无法避免因流量分配不平衡而导致的 A/B 群组差异过大问题，很容易造成对实验结果的误判。为满足不同业务场景的诉求，我们的 A/B 平台建设采取了多种分流策略，如下图所示：



针对策略之间的相互影响、请求不独立场景下的 A/B 实验，我们采取限流准入的分流方式，针对不同的实验，选取不同的分流因子。在实验前，我们通过 AA 分组，找出无差别的实验组和对照组，作为我们实验分流配置的依据，这种分流方式要求我们要有一套完整刻画流量因子的指标体系，只要刻画流量因子的指标间无统计显著性，我们就认为分出的实验组和对照组无差别。

2.2 业务决策的重要依据

在实验后的效果评估环节，通常允许实验者用自定义的指标来衡量不同策略带来的影响。但这样做会带来如下两个问题：

- 首先，由实验者来负责实验效果的评估，很难做到客观。同时也无法避免实验者仅仅选择支持自己假设的指标，来证明自己的实验结论；
- 其次，所有的策略迭代都是为业务服务，如果实验者用自定义的、与业务认知不一致的指标，来说明实验效果、推动业务灰度，这种方式往往难以被采纳。

因此，权威的评估体系对于对齐大家认知，并帮助我们在策略迭代方面作出正确的决策，尤为重要。

3.A/B 评估体系构建

A/B 评估体系的构建，要解决 A/B 平台两个核心问题：

- 第一，要有一套用于刻画流量因子（区域、骑手、商家）的权威的、完备的指标体系，帮助实验者完成实验前的 AA 分组和实验后的效果评估；
- 第二，要建立一套科学的评估方法，帮助实验者作出正确的决策。

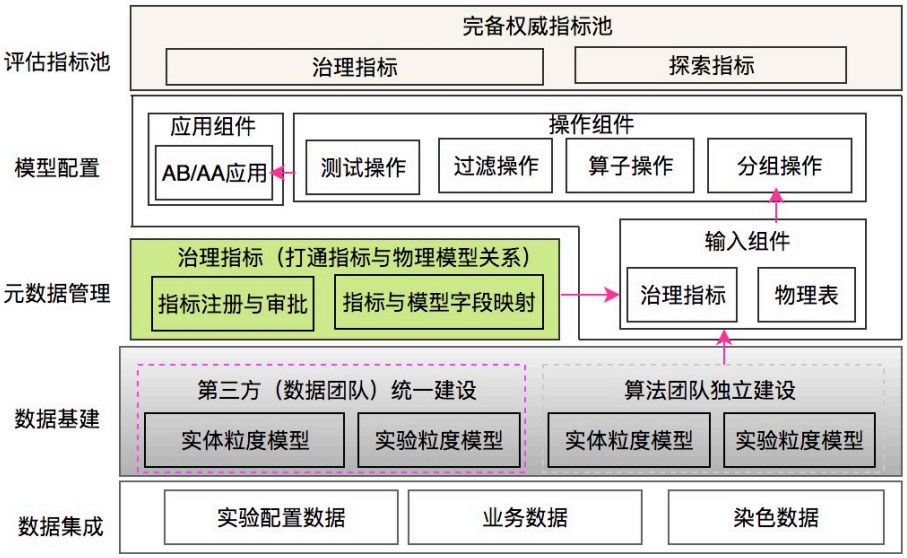
3.1 权威完备的指标体系

指标的权威性体现在：刻画分流因子，用于实验前 AA 分组和证明实验假设的指标，必须经过治理且业务认知一致，这样才能对齐认知，使得实验结果更具说服力；指标的完备性体现在：评估体系中的指标，不仅要有经过第三方独立生产治理且各业务方认知一致的治理指标，而且还要有实验者为了更全面的分析，描述实验过程，自定义的探索指标。

3.1.1 整体架构

治理指标强调的是指标的权威性和生产的规范性，而探索性指标强调的是指标的多样性和生产的灵活性。在评估体系中要实现这两类指标的统一，既要包含用于说明实验效果的治理指标，又要包含帮助实验者更好迭代实验所需的探索指标。

为实现上述的统一，指标层面要有分级运营的策略：治理指标按照业务认知一致性和算法内部认知一致性分别定级为 P0、P1，这一类指标在生产前必须要有严格的注册、评审，生产环节需要交给独立的第三方团队（数据团队）生产，保证指标的权威性，产出后打通指标与字段的映射关系，对用户屏蔽底层实现逻辑；对于探索性指标，定级为 P2，强调的是生产的灵活性和快速实现，因此，它的生产就不宜带有指标注册和评审等环节。为保证其快速实现，希望基于物理表和简单的算子配置就可以实现效果分析时即席查询使用。基于如上的问题拆解，我们进行了如下的架构设计：



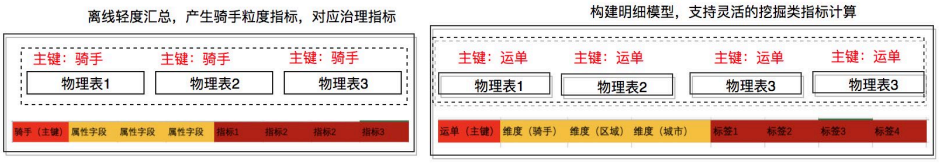
3.1.2 数据集成

为了支持监控和分析，在数据集成环节，我们集成了实验配置数据、业务数据和染色数据，以便实验者在效果评估环节不仅可以查看流量指标（PV、UV 和转化率），也可以深入探索策略变动对业务带来的影响。对于那些在实验配置环节不能确定流量是否真正参加实验的场景（例如：选择了特定区域进行实验，该区域产生的单只有满足特定条件时才能触发实验），我们不能直接通过限制确定的区域来查看业务指标。因为，此时查看的指标并不是真正参与实验的流量所对应的指标。因此在数据集成环节，我们同时将实验前的实验配置数据和实验中的染色数据（针对每个参与实验的流量，每次操作所产生的数据，都会打上实验场景、实验组以及具体的分组标记，我们该数据为染色数据）同步到数仓。在数据基建环节，将业务数据模型和染色数据模型通过流量实体作为关联条件进行关联，构建实验粒度模型。

3.1.3 数据基建

在数据基建层，我们基于指标分级运营的思路，由数据团队和算法团队分别构建实体粒度（区域、骑手、GeoHash）和实验粒度的实体宽表模型，以满足 P0/P1 指标和 P2 指标的诉求；为实现指标的规范化建设和灵活建设的统一，在物理模型和对外

提供应用的指标池之间，我们提供了元数据管理工具和模型配置工具，从而实现离线数据快速接入评估体系的指标池。由数据团队建设的实体宽表模型，对应着治理指标（P0/P1 指标），必须在生产后通过元数据管理工具完成指标与物理字段的映射，将指标的加工口径封装在数据层，对用户屏蔽物理实现，确保治理指标的一致性。由算法团队独立建设的实体宽表模型，对应着挖掘指标（P2 指标），为确保其接入评估体系指标池的灵活性和方便性，我们在数据基建环节，通过标签的形式对指标口径做部分封装，在模型配置环节完成指标逻辑的最终加工。

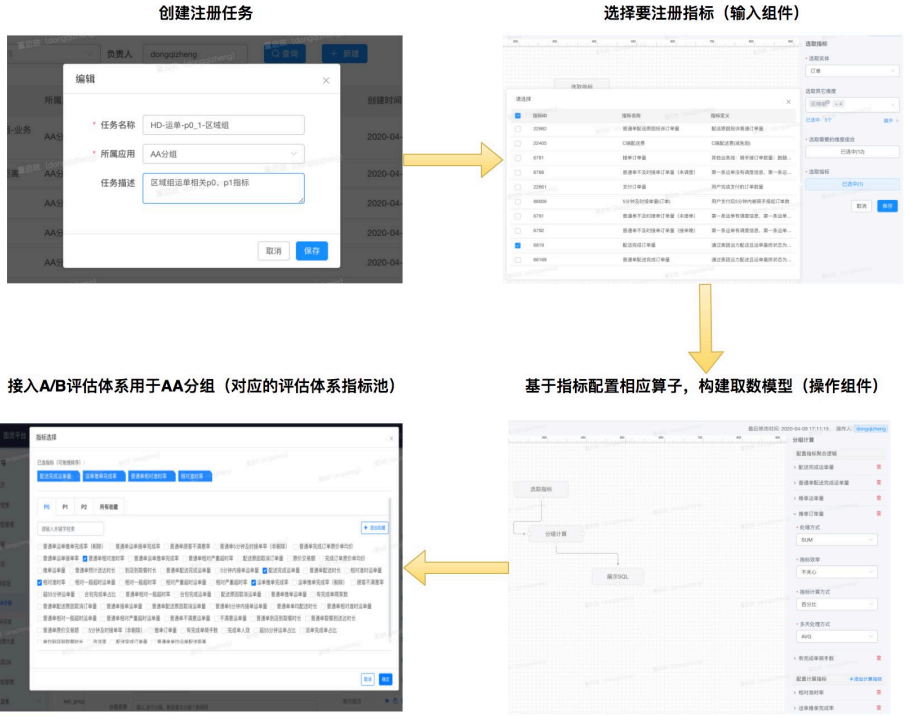


3.1.4 元数据管理

元数据管理层，是实现指标权威性的关键。治理指标在本层实现注册、评审，达到业务认知一致性和算法内部认知一致性的目的。同时，本层还完成了治理指标与数据基建层物理模型之间的绑定，为后续的模型配置建立基础。

3.1.5 模型配置

模型配置工具，是打通物理模型与评估指标池的桥梁，它通过输入组件、操作组件和应用组件，将离线数据接入到评估体系中，满足实验前 AA 分组和实验后 AB 评估的需求。首先，输入组件可以对应不同的数据源，既可以接入治理的离线指标，也可以接入特定库下的物理表。其次，操作组件提供了分组操作、算子操作、过滤操作和测试操作，通过分组操作，确定模型包含的维度；通过算子操作，将算子作用在指标或标签字段上，在取数环节实现指标的二次计算；通过过滤操作，实现数据的过滤；通过测试操作，保证模型配置质量。最后，应用组件可以将配置的模型注册到不同的应用上，针对 A/B 场景主要是 AA 分组和 AB 评估。具体接入流程如下图所示：



3.2 科学权威的评估方式

评估报告的可靠和权威性主要体现在两个方面：一是评估指标的可靠性和权威性；二是评估方式的科学性。在上一节中，我们重点讨论了如何构建可靠权威的指标体系。在这一节，我们重点讨论如何进行科学的评估。

在讨论科学评估之前，我们再重温一下 A/B 实验的定义：A/B 实验，简单来说，就是为同一个目标制定两个版本或多个版本的方案，在同一时间维度，分别让组成成分相同（相似）的 A/B 群组分别采用这些版本，收集各群组的体验数据和业务数据，最后分析、评估出最好版本，正式采用。其中 A 方案为现行的设计（称为控制组），B 方案是新的设计（称为实验组）。分析 A/B 实验的定义，要实现科学权威的评估，最重要的两点在于：

- 第一，确保在实验前分出无差别的实验组和对照组，避免因流量分配不平衡导致的 AB 群组差异过大，最终造成对于实验结果的误判；

- 第二，确保对实验结果作出准确的判断，能够准确的判断新策略相对于旧策略的优势是不是由自然波动引起的，它的这一优势能否在大规模的推广中反映出来。

无论是实验前确保实验组和对照组流量无显著性差异，还是实验后新策略较旧策略的指标变动是否具有统计上的显著性，无一例外，它们都蕴含着统计学的知识。接下来，我们重点论述一下 A/B 实验所依赖的统计学基础以及如何依据统计学理论做出科学评估。

3.2.1 假设检验

3.2.1.1 两个假设

A/B 测试是一种对比试验，我们圈定一定的流量进行实验，实验结束后，我们基于实验样本进行数据统计，进而验证实验前假设的正确性，我们得出这一有效结论的科学依据便是假设检验。假设检验是利用样本统计量估计总体参数的方法，在假设检验中，先对总体均值提出一个假设，然后用样本信息去检验这个假设是否成立。我们把提出的这个假设叫做原假设，与原假设对立的结论叫做备择假设，如果原假设不成立，就要拒绝原假设，进而接受备择假设。

3.2.1.2 两类错误

对于原假设提出的命题，我们需要作出判断，要么原假设成立，要么原假设不成立。因为基于样本对总体的推断，会面临着犯两种错误的可能：第一类错误，原假设为真，我们却拒绝了；第二类错误，原假设为伪，我们却接受了。显然，我们希望犯这两类错误的概率越小越好，但对于一定的样本量 n，不能同时做到犯这两类错误的概率很小。

在假设检验中，就有一个对两类错误进行控制的问题。一般来说，哪一类错误所带来的后果严重、危害越大，在假设检验中就应该把哪一类错误作为首要的控制目标。在假设检验中，我们都执行这样一个原则，首先控制犯第一类错误的概率。这也是为什么我们在实际应用中会把要推翻的假设作为原假设，这样得出的结论更具说服力（我

们有足够充分的证据证明原来确定的结论是错误的），所以通常会看到，我们把要证明的结论作为备择假设。

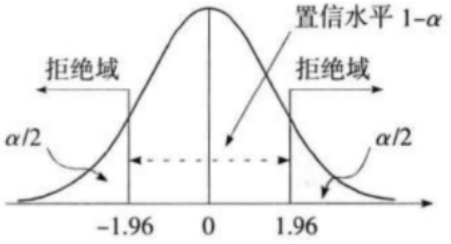
3.2.1.3 T 检验

常见的假设检验方法有 Z 检验、T 检验和卡方检验等，不同的方法有不同的适用条件和检验目标。Z 检验和 T 检验都是用来推断两个总体均值差异的显著性水平，具体选择哪种检验由样本量的大小、总体的方差是否已知决定。在样本量较小且总体的方差未知的情况下，这时只能使用样本方差代替总体方差，样本统计量服从 T 分布，应该采用 T 统计量进行检验。T 统计量具体构造公式如下图所示，其中 f 是 T 统计量的自由度，S1、S2 是样本标准差。

$$t=\frac{(\bar{x}_1-\bar{x}_2)-(\mu_1-\mu_2)}{\sqrt{\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}}}$$

$$f=\frac{\left(\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1-1}+\frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2-1}}$$

T 检验的流程是，在给定的弃真错误概率下（一般取 0.05），依据样本统计量 T 是否落在拒绝域来判断接受还是拒绝原假设。实际上在确定弃真错误概率以后，拒绝域的位置也就相应地确定了。使用 T 统计量进行判断的好处是，进行决策的界限清晰，但缺陷是决策面临的风险是笼统的。例如 T=3 落入拒绝域，我们拒绝原假设，犯弃真错误的概率为 0.05；T=2 也落入拒绝域，我们拒绝原假设，犯弃真错误的概率也是 0.05。事实上，依据不同的统计量进行决策，面临的风险也是有差别的。为了精确地反映决策的风险度，我们仍然需要 P 值来帮助业务来做决策。



3.2.1.4 利用 P 值决策

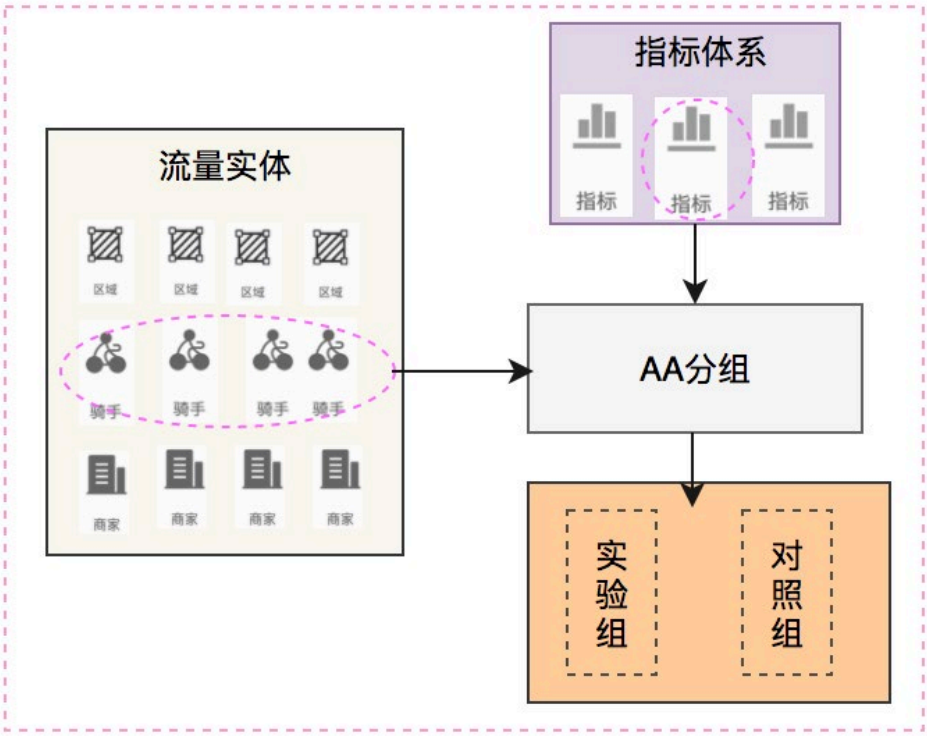
P 值是当原假设为真时，所得到的样本观察结果或更极端的结果出现的概率。如果 P 值很小，说明这种情况发生的概率很小，但是在这次试验中却出现了，根据小概率原理，我们有理由拒绝原假设，P 值越小，我们拒绝原假设的理由越充分。P 值可以理解为犯弃真错误的概率，在确定的显著性水平下（一般取 0.05），P 值小于显著性水平，则拒绝原假设。

3.2.2 基于假设检验的科学评估

围绕着科学评估要解决的两个问题，实验前，针对圈定的流量使用假设检验加上动态规划算法，确保分出无差别的实验组和对照组；实验后，基于实验前选定的用于验证假设结论的指标，构造 T 统计量并计算其对应的 P 值，依据 P 值帮我们做决策。

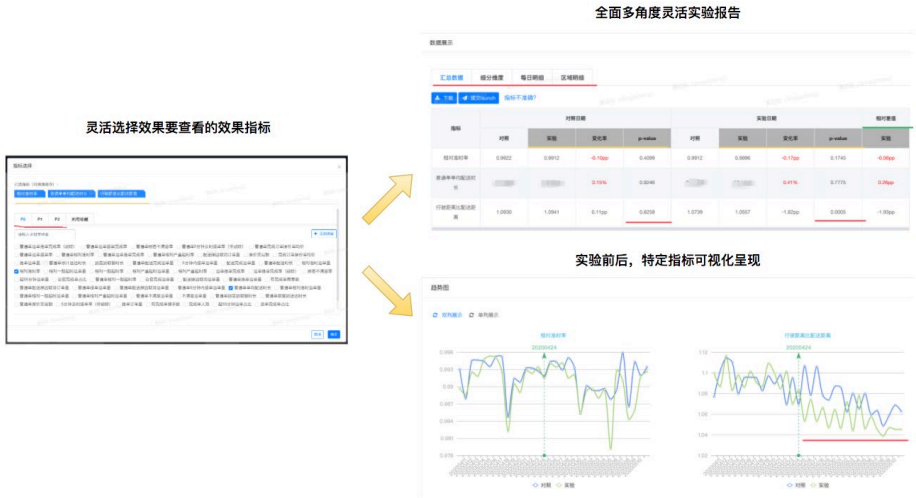
3.2.2.1 AA 分组

首先看如何解决第一个问题：避免因流量分配不平衡，A/B 组本身差异过大造成对实验结果的误判。为解决该问题，我们引入了 AA 分组：基于实验者圈定的流量，通过 AA 分组将该流量分为无显著性差异的实验组和对照组。我们这样定义无显著性差异这一约束：首先，实验者选取的用于刻画实验流量的指标，在实验组和对照组之间无统计上的显著性（即上节所描述的基于均值的假设检验）；其次，在所分出的实验组和对照组之间，这些指标的差值最小，即一个寻找最优解的过程。从实验者的实验流程看，在实验前，圈定进入该实验的流量，然后确定用于刻画实验流量的指标，最后调用 AA 分组，为其将流量分成合理的实验组和对照组。



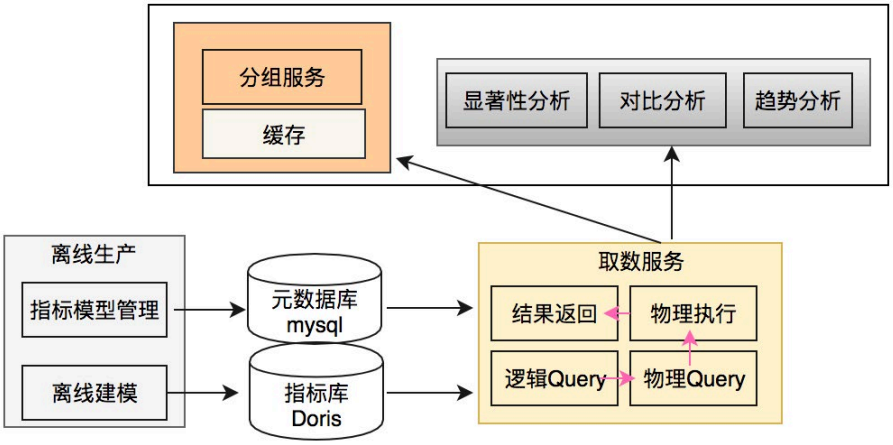
3.2.2.2 A/B 效果评估

A/B 效果评估是实验者在实验后，依据评估报告进行决策的重要依据。 因此，我们在实验后的效果评估环节，效果评估要达成三个目标即权威、灵活性和方便。首先，权威性体现在用于作出实验结论所依赖的指标都是经过治理、各方达成一致的指标，并且确保数据一致性，最终通过假设检验给出科学的实验结论，帮助实验者作出正确的判断。其次，灵活性主要体现在采用列转行的形式，按需自动生成报表告别“烟囱式”的报表开发方式。第三，方便主要体现在不仅可以查看用于说明实验效果的指标，还可以选择查看接入到评估体系里的任意指标；不仅可以查看其实验前后对比以及趋势变化，还可以做到从实验粒度到流量实体粒度的下钻。效果如下图所示：



3.2.2.3 技术实现

不管是实验前的 AA 分组，还是实验后的效果评估，我们要解决的一个核心问题就是如何灵活地“取数”，为我们的 AA 分组和 AB 效果分析提供一个灵活稳定的取数服务。因此，我们整个架构的核心就是构建稳定、灵活的取数服务，具体架构如下图所示。离线建模和指标模型管理完成数据和元数据建设，建立权威完备的指标体系；中间的取数服务作为上层各应用服务和指标体系的“桥梁”，为上层各应用服务提供其所依赖的指标。



4. 总结与展望

目前，A/B 测试已成为许多互联网公司评估其新产品策略和方法的“金标准”，在美团配送业务场景下，它被广泛应用于调度策略、定价策略、运力优化、ETA 时间预估等业务场景，为我们的策略迭代制定数据驱动型决策。特别是针对配送场景下这种策略之间相互影响，请求不独立场景下的 A/B 实验，结合配送技术团队的具体实践，跟大家分享了我们目前的解决思路。

最后再补充一点，在 A/B 测试领域，实验的流量规模应该有足够的统计能力，才能确保指标的变化有统计意义的，为了更好地达到这个目标，未来我们将通过辅助工具建设，在实验前，依据实验者所关注的指标以及敏感度给出流量规模的建议，方便实验者在实验前快速地圈定其实验所需的流量。

5. 作者简介

王鹏，2016 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、数据治理及系统化和 A/B 评估体系建设相关工作。

启政，2018 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、A/B 评估体系建设相关工作。

连恒，2016 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、A/B 评估体系建设相关工作。

新一代垃圾回收器 ZGC 的探索与实践

作者：王东　王伟

ZGC（The Z Garbage Collector）是 JDK 11 中推出的一款低延迟垃圾回收器，它的设计目标包括：

- 停顿时间不超过 10ms；
- 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
- 支持 8MB~4TB 级别的堆（未来支持 16TB）。

从设计目标来看，我们知道 ZGC 适用于大内存低延迟服务的内存管理和回收。本文主要介绍 ZGC 在低延时场景中的应用和卓越表现，文章内容主要分为四部分：

- **GC 之痛**：介绍实际业务中遇到的 GC 痛点，并分析 CMS 收集器和 G1 收集器停顿时间瓶颈；
- **ZGC 原理**：分析 ZGC 停顿时间比 G1 或 CMS 更短的本质原因，以及背后的技术原理；
- **ZGC 调优实践**：重点分享对 ZGC 调优的理解，并分析若干个实际调优案例；
- **升级 ZGC 效果**：展示在生产环境应用 ZGC 取得的效果。

GC 之痛

很多低延迟高可用 Java 服务的系统可用性经常受 GC 停顿的困扰。GC 停顿指垃圾回收期间 STW（Stop The World），当 STW 时，所有应用线程停止活动，等待 GC 停顿结束。以美团风控服务为例，部分上游业务要求风控服务 65ms 内返回结果，并且可用性要达到 99.99%。但因为 GC 停顿，我们未能达到上述可用性目标。当时使用的是 CMS 垃圾回收器，单次 Young GC 40ms，一分钟 10 次，接口平均响应时间 30ms。通过计算可知，有（40ms + 30ms）* 10 次 / 60000ms = 1.12% 的请求的响应时间会增加 0 ~ 40ms 不等，其中 30ms * 10 次 / 60000ms = 0.5% 的请求

响应时间会增加 40ms。可见，GC 停顿对响应时间的影响较大。为了降低 GC 停顿对系统可用性的影响，我们从降低单次 GC 时间和降低 GC 频率两个角度出发进行了调优，还测试过 G1 垃圾回收器，但这三项措施均未能降低 GC 对服务可用性的影响。

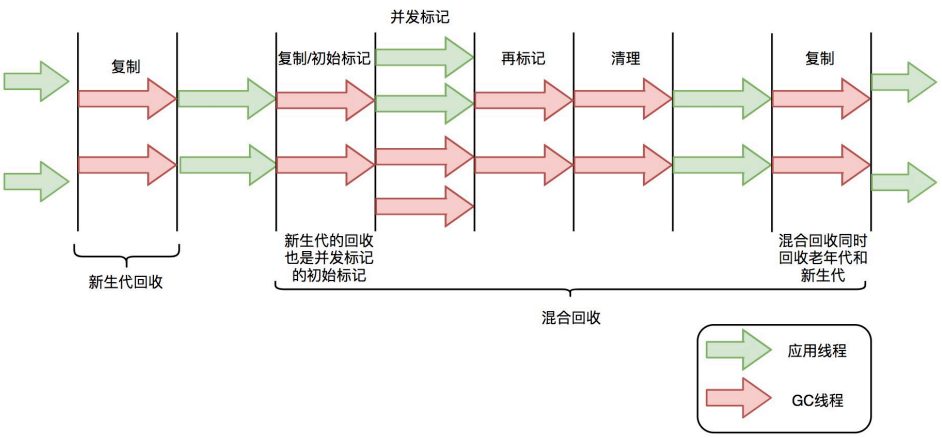
CMS 与 G1 停顿时间瓶颈

在介绍 ZGC 之前，首先回顾一下 CMS 和 G1 的 GC 过程以及停顿时间的瓶颈。CMS 新生代的 Young GC、G1 和 ZGC 都基于标记 – 复制算法，但算法具体实现的不同就导致了巨大的性能差异。

标记 – 复制算法应用在 CMS 新生代（ParNew 是 CMS 默认的新生代垃圾回收器）和 G1 垃圾回收器中。标记 – 复制算法可以分为三个阶段：

- 标记阶段，即从 GC Roots 集合开始，标记活跃对象；
- 转移阶段，即把活跃对象复制到新的内存地址上；
- 重定位阶段，因为转移导致对象的地址发生了变化，在重定位阶段，所有指向对象旧地址的指针都要调整到对象新的地址上。

下面以 G1 为例，通过 G1 中标记 – 复制算法过程（G1 的 Young GC 和 Mixed GC 均采用该算法），分析 G1 停顿耗时的主要瓶颈。G1 垃圾回收周期如下图所示：



G1 的混合回收过程可以分为标记阶段、清理阶段和复制阶段。

标记阶段停顿分析

- 初始标记阶段：初始标记阶段是指从 GC Roots 出发标记全部直接子节点的过程，该阶段是 STW 的。由于 GC Roots 数量不多，通常该阶段耗时非常短。
- 并发标记阶段：并发标记阶段是指从 GC Roots 开始对堆中对象进行可达性分析，找出存活对象。该阶段是并发的，即应用线程和 GC 线程可以同时活动。并发标记耗时相对长很多，但因为不是 STW，所以我们不太关心该阶段耗时的长短。
- 再标记阶段：重新标记那些在并发标记阶段发生变化的对象。该阶段是 STW 的。

清理阶段停顿分析

- 清理阶段清点出有存活对象的分区和没有存活对象的分区，该阶段不会清理垃圾对象，也不会执行存活对象的复制。该阶段是 STW 的。

复制阶段停顿分析

- 复制算法中的转移阶段需要分配新内存和复制对象的成员变量。转移阶段是 STW 的，其中内存分配通常耗时非常短，但对象成员变量的复制耗时有可能较长，这是因为复制耗时与存活对象数量与对象复杂度成正比。对象越复杂，复制耗时越长。

四个 STW 过程中，初始标记因为只标记 GC Roots，耗时较短。再标记因为对象数少，耗时也较短。清理阶段因为内存分区数量少，耗时也较短。转移阶段要处理所有存活的对象，耗时会较长。因此，G1 停顿时间的瓶颈主要是标记 – 复制中的转移阶段 STW。为什么转移阶段不能和标记阶段一样并发执行呢？主要是 G1 未能解决转移过程中准确定位对象地址的问题。

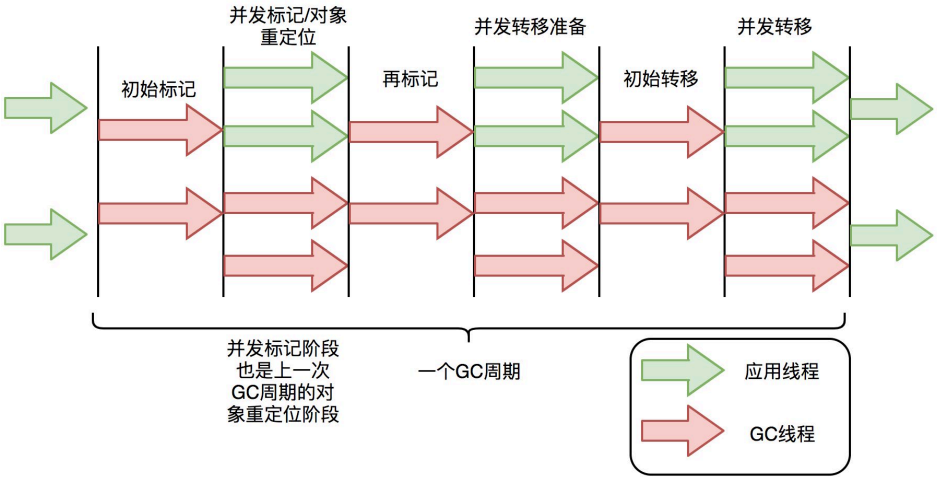
G1 的 Young GC 和 CMS 的 Young GC，其标记 – 复制全过程 STW，这里不再详细阐述。

ZGC 原理

全并发的 ZGC

与 CMS 中的 ParNew 和 G1 类似，ZGC 也采用标记 – 复制算法，不过 ZGC 对该算法做了重大改进：ZGC 在标记、转移和重定位阶段几乎都是并发的，这是 ZGC 实现停顿时间小于 10ms 目标的最关键原因。

ZGC 垃圾回收周期如下图所示：



ZGC 只有三个 STW 阶段：**初始标记**，**再标记**，**初始转移**。其中，初始标记和初始转移分别都只需要扫描所有 GC Roots，其处理时间和 GC Roots 的数量成正比，一般情况耗时非常短；再标记阶段 STW 时间很短，最多 1ms，超过 1ms 则再次进入并发标记阶段。即，ZGC 几乎所有暂停都只依赖于 GC Roots 集合大小，停顿时间不会随着堆的大小或者活跃对象的大小而增加。与 ZGC 对比，G1 的转移阶段完全 STW 的，且停顿时间随存活对象的大小增加而增加。

ZGC 关键技术

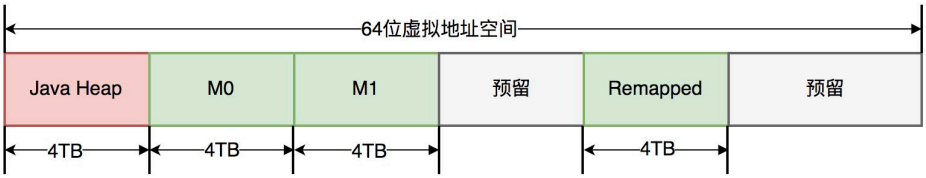
ZGC 通过着色指针和读屏障技术，解决了转移过程中准确访问对象的问题，实现了并发转移。大致原理描述如下：并发转移中“并发”意味着 GC 线程在转移对象的过

程中，应用线程也在不停地访问对象。假设对象发生转移，但对象地址未及时更新，那么应用线程可能访问到旧地址，从而造成错误。而在 ZGC 中，应用线程访问对象将触发“读屏障”，如果发现对象被移动了，那么“读屏障”会把读出来的指针更新到对象的新地址上，这样应用线程始终访问的都是对象的新地址。那么，JVM 是如何判断对象被移动过呢？就是利用对象引用的地址，即着色指针。下面介绍着色指针和读屏障技术细节。

着色指针

着色指针是一种将信息存储在指针中的技术。

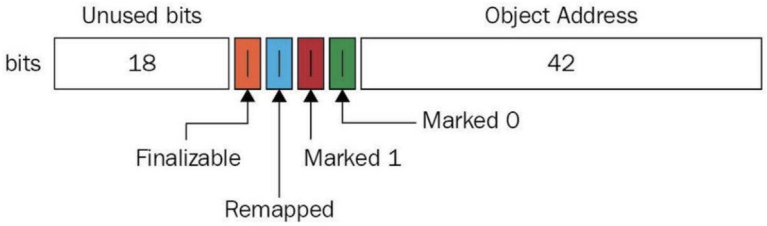
ZGC 仅支持 64 位系统，它把 64 位虚拟地址空间划分为多个子空间，如下图所示：



其中，[0~4TB) 对应 Java 堆，[4TB ~ 8TB) 称为 M0 地址空间，[8TB ~ 12TB) 称为 M1 地址空间，[12TB ~ 16TB) 预留未使用，[16TB ~ 20TB) 称为 Remapped 空间。

当应用程序创建对象时，首先在堆空间申请一个虚拟地址，但该虚拟地址并不会映射到真正的物理地址。ZGC 同时会为该对象在 M0、M1 和 Remapped 地址空间分别申请一个虚拟地址，且这三个虚拟地址对应同一个物理地址，但这三个空间在同一时间有且只有一个空间有效。ZGC 之所以设置三个虚拟地址空间，是因为它使用“空间换时间”思想，去降低 GC 停顿时间。“空间换时间”中的空间是虚拟空间，而不是真正的物理空间。后续章节将详细介绍这三个空间的切换过程。

与上述地址空间划分相对应，ZGC 实际仅使用 64 位地址空间的第 0~41 位，而第 42~45 位存储元数据，第 47~63 位固定为 0。



ZGC 将对象存活信息存储在 42~45 位中，这与传统的垃圾回收并将对象存活信息放在对象头中完全不同。

读屏障

读屏障是 JVM 向应用代码插入一小段代码的技术。当应用线程从堆中读取对象引用时，就会执行这段代码。需要注意的是，仅“从堆中读取对象引用”才会触发这段代码。

读屏障示例：

```
Object o = obj.FieldA   // 从堆中读取引用，需要加入屏障
<Load barrier>
Object p = o  // 无需加入屏障，因为不是从堆中读取引用
o.dosomething() // 无需加入屏障，因为不是从堆中读取引用
int i =  obj.FieldB  // 无需加入屏障，因为不是对象引用
```

ZGC 中读屏障的代码作用：在对象标记和转移过程中，用于确定对象的引用地址是否满足条件，并作出相应动作。

ZGC 并发处理演示

接下来详细介绍 ZGC 一次垃圾回收周期中地址视图的切换过程：

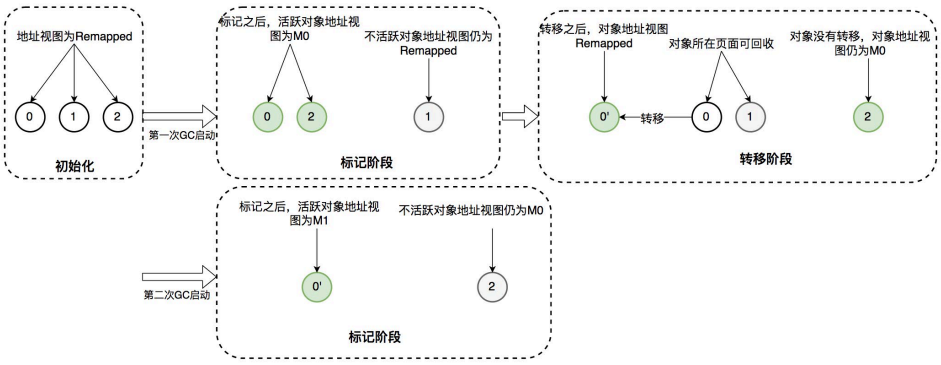
- **初始化**：ZGC 初始化之后，整个内存空间的地址视图被设置为 Remapped。程序正常运行，在内存中分配对象，满足一定条件后垃圾回收启动，此时进入标记阶段。
- **并发标记阶段**：第一次进入标记阶段时视图为 M0，如果对象被 GC 标记线程或者应用线程访问过，那么就将对象的地址视图从 Remapped 调整

为 M0。所以，在标记阶段结束之后，对象的地址要么是 M0 视图，要么是 Remapped。如果对象的地址是 M0 视图，那么说明对象是活跃的；如果对象的地址是 Remapped 视图，说明对象是不活跃的。

- **并发转移阶段**：标记结束后就进入转移阶段，此时地址视图再次被设置为 Remapped。如果对象被 GC 转移线程或者应用线程访问过，那么就将对象的地址视图从 M0 调整为 Remapped。

其实，在标记阶段存在两个地址视图 M0 和 M1，上面的过程显示只用了一个地址视图。之所以设计成两个，是为了区别前一次标记和当前标记。也即，第二次进入并发标记阶段后，地址视图调整为 M1，而非 M0。

着色指针和读屏障技术不仅应用在并发转移阶段，还应用在并发标记阶段：将对象设置为已标记，传统的垃圾回收器需要进行一次内存访问，并将对象存活信息放在对象头中；而在 ZGC 中，只需要设置指针地址的第 42~45 位即可，并且因为是寄存器访问，所以速度比访问内存更快。



ZGC 调优实践

ZGC 不是“银弹”，需要根据服务的具体特点进行调优。网络上能搜索到实战经验较少，调优理论需自行摸索，我们在此阶段也耗费了不少时间，最终才达到理想的性能。本文的一个目的是列举一些使用 ZGC 时常见的问题，帮助大家使用 ZGC 提高服务可用性。

调优基础知识

理解 ZGC 重要配置参数

以我们服务在生产环境中 ZGC 参数配置为例，说明各个参数的作用：

重要参数配置样例：

```
-Xms10G -Xmx10G
-XX:ReservedCodeCacheSize=256m -XX:InitialCodeCacheSize=256m
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC
-XX:ConcGCThreads=2 -XX:ParallelGCThreads=6
-XX:ZCollectionInterval=120 -XX:ZAllocationSpikeTolerance=5
-XX:+UnlockDiagnosticVMOptions -XX:-ZProactive
-Xlog:safepoint,classhisto*=trace,age*,gc*=info:file=/opt/logs/logs/gc-
%t.log:time,tid,tags:filecount=5,filesize=50m
```

-Xms -Xmx：堆的最大内存和最小内存，这里都设置为 10G，程序的堆内存将保持 10G 不变。**-XX:ReservedCodeCacheSize -XX:InitialCodeCacheSize**：设置 CodeCache 的大小，JIT 编译的代码都放在 CodeCache 中，一般服务 64m 或 128m 就已经足够。我们的服务因为有一定特殊性，所以设置的较大，后面会详细介绍。**-XX:+UnlockExperimentalVMOptions -XX:+UseZGC**：启用 ZGC 的配置。**-XX:ConcGCThreads**：并发回收垃圾的线程。默认是总核数的 12.5%，8 核 CPU 默认是 1。调大后 GC 变快，但会占用程序运行时的 CPU 资源，吞吐会受到影响。**-XX:ParallelGCThreads**：STW 阶段使用线程数，默认是总核数的 60%。**-XX:ZCollectionInterval**：ZGC 发生的最小时间间隔，单位秒。**-XX:ZAllocationSpikeTolerance**：ZGC 触发自适应算法的修正系数，默认 2，数值越大，越早的触发 ZGC。**-XX:+UnlockDiagnosticVMOptions -XX:-ZProactive**：是否启用主动回收，默认开启，这里的配置表示关闭。**-Xlog**：设置 GC 日志中的内容、格式、位置以及每个日志的大小。

理解 ZGC 触发时机

相比于 CMS 和 G1 的 GC 触发机制，ZGC 的 GC 触发机制有很大不同。ZGC 的核心特点是并发，GC 过程中一直有新的对象产生。如何保证在 GC 完成之前，新

产生的对象不会将堆占满，是 ZGC 参数调优的第一大目标。因为在 ZGC 中，当垃圾来不及回收将堆占满时，会导致正在运行的线程停顿，持续时间可能长达秒级之久。

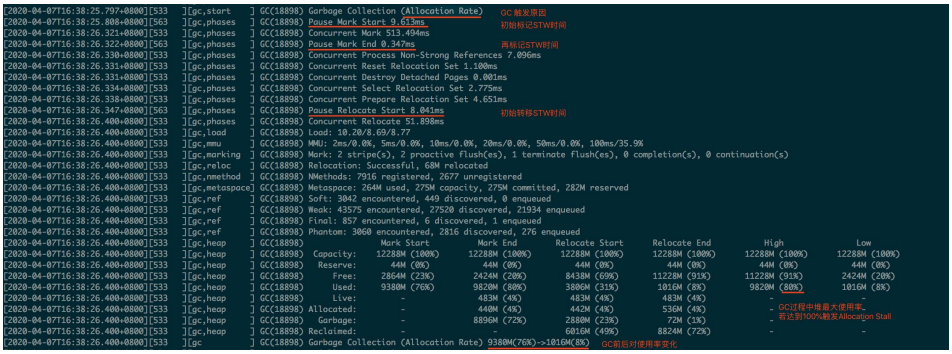
ZGC 有多种 GC 触发机制，总结如下：

- 阻塞内存分配请求触发：当垃圾来不及回收，垃圾将堆占满时，会导致部分线程阻塞。我们应当避免出现这种触发方式。日志中关键字是“Allocation Stall”。
- 基于分配速率的自适应算法：最主要的 GC 触发方式，其算法原理可简单描述为”ZGC 根据近期的对象分配速率以及 GC 时间，计算出当内存占用达到什么阈值时触发下一次 GC”。自适应算法的详细理论可参考彭成寒《新一代垃圾回收器 ZGC 设计与实现》一书中的内容。通过 ZAllocationSpikeTolerance 参数控制阈值大小，该参数默认 2，数值越大，越早的触发 GC。我们通过调整此参数解决了一些问题。日志中关键字是“Allocation Rate”。
- 基于固定时间间隔：通过 ZCollectionInterval 控制，适合应对突增流量场景。流量平稳变化时，自适应算法可能在堆使用率达到 95% 以上才触发 GC。流量突增时，自适应算法触发的时机可能会过晚，导致部分线程阻塞。我们通过调整此参数解决流量突增场景的问题，比如定时活动、秒杀等场景。日志中关键字是“Timer”。
- 主动触发规则：类似于固定间隔规则，但时间间隔不固定，是 ZGC 自行算出来的时机，我们的服务因为已经加了基于固定时间间隔的触发机制，所以通过 -ZProactive 参数将该功能关闭，以免 GC 频繁，影响服务可用性。 日志中关键字是“Proactive”。
- 预热规则：服务刚启动时出现，一般不需要关注。日志中关键字是“Warmup”。
- 外部触发：代码中显式调用 System.gc() 触发。 日志中关键字是“System.gc()”。

- 元数据分配触发：元数据区不足时导致，一般不需要关注。 日志中关键字是“Metadata GC Threshold”。

理解 ZGC 日志

一次完整的 GC 过程，需要注意的点已在图中标出。

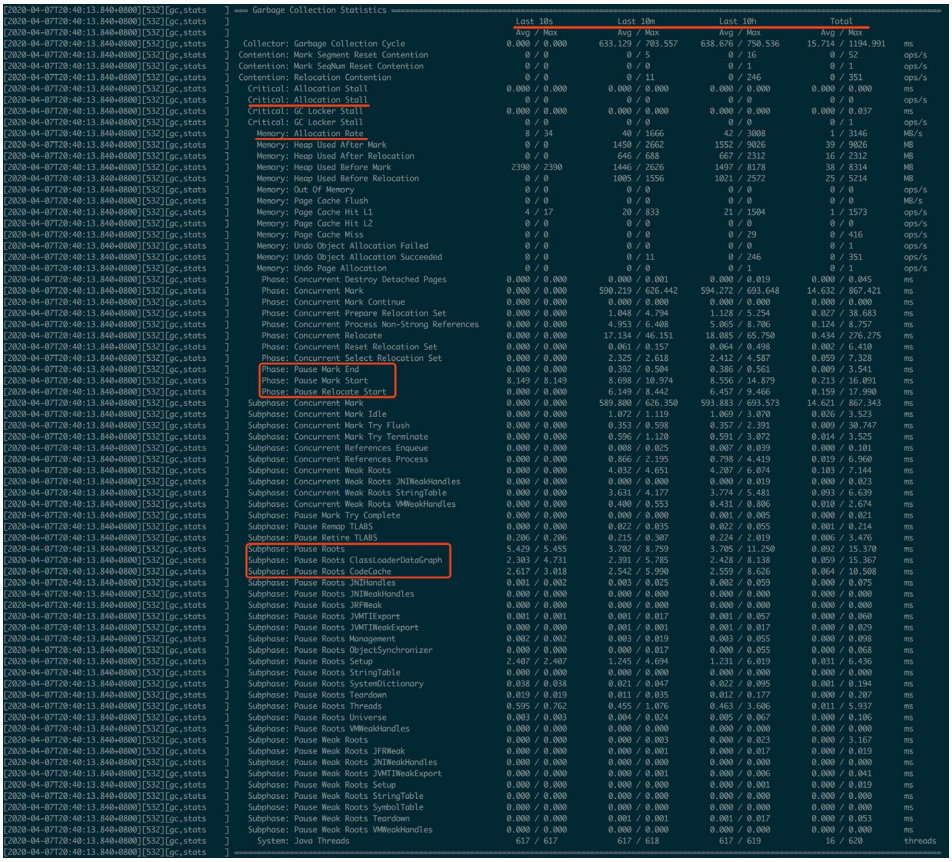


注意：该日志过滤了进入安全点的信息。正常情况，在一次 GC 过程中还穿插着进入安全点的操作。

GC 日志中每一行都注明了 GC 过程中的信息，关键信息如下：

- **Start**：开始 GC，并标明的 GC 触发的原因。上图中触发原因是自适应算法。
- **Phase-Pause Mark Start**：初始标记，会 STW。
- **Phase-Pause Mark End**：再次标记，会 STW。
- **Phase-Pause Relocate Start**：初始转移，会 STW。
- **Heap 信息**：记录了 GC 过程中 Mark、Relocate 前后的堆大小变化状况。High 和 Low 记录了其中的最大值和最小值，我们一般关注 High 中 Used 的值，如果达到 100%，在 GC 过程中一定存在内存分配不足的情况，需要调整 GC 的触发时机，更早或者更快地进行 GC。
- **GC 信息统计**：可以定时的打印垃圾收集信息，观察 10 秒内、10 分钟内、10 个小时内，从启动到现在的所有统计信息。利用这些统计信息，可以排查定位一些异常点。

日志中内容较多，关键点已用红线标出，含义较好理解，更详细的解释大家可以自行在网上查阅资料。



```
[2020-04-07T20:40:13.840+0800][532][gc,stats    ] === Garbage Collection Statistics =======================================================================================================================
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]                                                                 Last 10s              Last 10m              Last 10h                Total
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]                                                                 Avg / Max             Avg / Max             Avg / Max               Avg / Max
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]   Collector: Garbage Collection Cycle                         0.000 / 0.000         633.129 / 703.557     638.676 / 750.536       15.714 / 1194.991     ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]  Contention: Mark Segment Reset Contention                       0 / 0                 0 / 5                 0 / 16                  0 / 52              ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]  Contention: Mark SeqNum Reset Contention                        0 / 0                 0 / 0                 0 / 1                   0 / 1               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]  Contention: Relocation Contention                               0 / 0                 0 / 11                0 / 246                 0 / 351             ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Critical: Allocation Stall                                   0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Critical: Allocation Stall                                     0 / 0                 0 / 0                 0 / 0                   0 / 0               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Critical: GC Locker Stall                                   0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.037       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Critical: GC Locker Stall                                     0 / 0                 0 / 0                 0 / 0                   0 / 1               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Allocation Rate                                     8 / 34                40 / 1666             42 / 3008               1 / 3146            MB/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Heap Used After Mark                                0 / 0                 1450 / 2662           1552 / 9026             39 / 9026           MB
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Heap Used After Relocation                          0 / 0                 646 / 688             667 / 2312              16 / 2312           MB
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Heap Used Before Mark                               2390 / 2390           1446 / 2626           1497 / 8178             38 / 8314           MB
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Heap Used Before Relocation                         0 / 0                 1005 / 1556           1021 / 2572             25 / 5214           MB
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Out Of Memory                                       0 / 0                 0 / 0                 0 / 0                   0 / 0               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Page Cache Flush                                    0 / 0                 0 / 0                 0 / 0                   0 / 0               MB/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Page Cache Hit L1                                   4 / 17                20 / 833              21 / 1504               1 / 1573            ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Page Cache Hit L2                                   0 / 0                 0 / 0                 0 / 0                   0 / 0               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Page Cache Miss                                     0 / 0                 0 / 0                 0 / 29                  0 / 416             ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Undo Object Allocation Failed                       0 / 0                 0 / 0                 0 / 0                   0 / 1               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Undo Object Allocation Succeeded                    0 / 0                 0 / 11                0 / 246                 0 / 351             ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]      Memory: Undo Page Allocation                                0 / 0                 0 / 0                 0 / 1                   0 / 1               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Destroy Detached Pages                   0.000 / 0.000         0.000 / 0.001         0.000 / 0.019           0.000 / 0.045       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Mark                                     0.000 / 0.000         590.219 / 626.442     594.272 / 693.648       14.632 / 867.421    ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Mark Continue                            0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Prepare Relocation Set                   0.000 / 0.000         1.048 / 4.794         1.128 / 5.254           0.027 / 38.683      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Process Non-Strong References            0.000 / 0.000         4.953 / 6.408         5.065 / 8.706           0.124 / 8.757       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Relocate                                 0.000 / 0.000         17.134 / 46.151       18.085 / 65.750         0.434 / 276.275     ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Reset Relocation Set                     0.000 / 0.000         0.061 / 0.157         0.064 / 0.498           0.002 / 6.410       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Concurrent Select Relocation Set                    0.000 / 0.000         2.325 / 2.618         2.412 / 4.587           0.059 / 7.328       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Pause Mark End                                      0.000 / 0.000         0.392 / 0.504         0.386 / 0.561           0.009 / 3.541       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Pause Mark Start                                    8.149 / 8.149         8.698 / 10.974        8.556 / 14.879          0.213 / 16.091      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]       Phase: Pause Relocate Start                                0.000 / 0.000         6.149 / 8.442         6.457 / 9.466           0.159 / 17.990      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Mark                                     0.000 / 0.000         589.800 / 626.350     593.883 / 693.573       14.621 / 867.343    ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Mark Idle                                0.000 / 0.000         1.072 / 1.119         1.069 / 3.070           0.026 / 3.523       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Mark Try Flush                           0.000 / 0.000         0.353 / 0.598         0.357 / 2.391           0.009 / 30.747      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Mark Try Terminate                       0.000 / 0.000         0.596 / 1.120         0.591 / 3.072           0.014 / 3.525       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent References Enqueue                       0.000 / 0.000         0.008 / 0.025         0.007 / 0.039           0.000 / 0.101       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent References Process                       0.000 / 0.000         0.866 / 2.195         0.798 / 4.419           0.019 / 6.960       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Weak Roots                               0.000 / 0.000         4.032 / 4.651         4.207 / 6.074           0.103 / 7.144       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Weak Roots JNIWeakHandles                0.000 / 0.000         0.000 / 0.000         0.000 / 0.019           0.000 / 0.023       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Weak Roots StringTable                   0.000 / 0.000         3.631 / 4.177         3.774 / 5.481           0.093 / 6.639       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Concurrent Weak Roots VMWeakHandles                 0.000 / 0.000         0.400 / 0.553         0.431 / 0.806           0.010 / 2.674       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Mark Try Complete                             0.000 / 0.000         0.000 / 0.000         0.001 / 0.005           0.000 / 0.021       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Remap TLABS                                   0.000 / 0.000         0.022 / 0.035         0.022 / 0.055           0.001 / 0.214       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Retire TLABS                                  0.206 / 0.206         0.215 / 0.307         0.224 / 2.019           0.006 / 3.476       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots                                         5.429 / 5.455         3.702 / 8.759         3.705 / 11.250          0.092 / 15.370      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots ClassLoaderDataGraph                    2.303 / 4.731         2.391 / 5.785         2.428 / 8.138           0.059 / 15.367      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots CodeCache                               2.617 / 3.018         2.542 / 5.990         2.559 / 8.626           0.064 / 10.508      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots JNIHandles                              0.001 / 0.002         0.003 / 0.025         0.002 / 0.059           0.000 / 0.075       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots JNIWeakHandles                          0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots JRFWeak                                 0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots JVMTIExport                             0.001 / 0.001         0.001 / 0.017         0.001 / 0.057           0.000 / 0.060       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots JVMTIWeakExport                         0.000 / 0.000         0.001 / 0.001         0.001 / 0.017           0.000 / 0.029       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots Management                              0.002 / 0.002         0.003 / 0.019         0.003 / 0.055           0.000 / 0.098       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots ObjectSynchronizer                      0.000 / 0.000         0.000 / 0.017         0.000 / 0.055           0.000 / 0.068       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots Setup                                   2.407 / 2.407         1.245 / 4.694         1.231 / 6.019           0.031 / 6.436       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots StringTable                             0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots SystemDictionary                        0.038 / 0.038         0.021 / 0.047         0.022 / 0.095           0.001 / 0.194       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots Teardown                                0.019 / 0.019         0.011 / 0.035         0.012 / 0.177           0.000 / 0.207       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots Threads                                 0.595 / 0.762         0.455 / 1.076         0.463 / 3.606           0.011 / 5.937       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots Universe                                0.003 / 0.003         0.004 / 0.024         0.005 / 0.067           0.000 / 0.106       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Roots VMWeakHandles                           0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots                                    0.000 / 0.000         0.000 / 0.003         0.000 / 0.023           0.000 / 3.167       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots JFRWeak                            0.000 / 0.000         0.000 / 0.001         0.000 / 0.017           0.000 / 0.019       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots JNIWeakHandles                     0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots JVMTIWeakExport                    0.000 / 0.000         0.000 / 0.001         0.000 / 0.006           0.000 / 0.041       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots Setup                              0.000 / 0.000         0.000 / 0.000         0.000 / 0.001           0.000 / 0.019       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots StringTable                        0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots SymbolTable                        0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots Teardown                           0.000 / 0.000         0.001 / 0.001         0.001 / 0.017           0.000 / 0.053       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]    Subphase: Pause Weak Roots VMWeakHandles                      0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats    ]     System: Java Threads                                         617 / 617             617 / 618             617 / 619               16 / 620            threads
[2020-04-07T20:40:13.840+0800][532][gc,stats    ] =======================================================================================================================
```

理解 ZGC 停顿原因

我们在实战过程中共发现了 6 种使程序停顿的场景，分别如下：

- **GC 时，初始标记**：日志中 Pause Mark Start。
- **GC 时，再标记**：日志中 Pause Mark End。
- **GC 时，初始转移**：日志中 Pause Relocate Start。
- **内存分配阻塞**：当内存不足时线程会阻塞等待 GC 完成，关键字是”Allocation Stall”。

```
Allocation Stall (zeus-fiber-worker-common-pool-3) 364.489ms
Allocation Stall (zeus-fiber-worker-common-pool-4) 364.293ms
Allocation Stall (zeus-fiber-worker-common-pool-14) 363.021ms
Allocation Stall (MtthriftClientNioGroup-5-thread-29) 363.921ms
Allocation Stall (MtthriftClientNioGroup-5-thread-5) 362.064ms
Allocation Stall (MtthriftClientNioGroup-5-thread-17) 359.804ms
```

- **安全点**：所有线程进入到安全点后才能进行 GC，ZGC 定期进入安全点判断是否需要 GC。先进入安全点的线程需要等待后进入安全点的线程直到所有线程挂起。
- **dump 线程、内存**：比如 jstack、jmap 命令。

```
Entering safepoint region: ThreadDump
Leaving safepoint region
Total time for which application threads were stopped: 1.1186756 seconds, Stopping threads took: 0.0004182 seconds
```

```
Entering safepoint region: HeapDumper
GC(34778) Garbage Collection (Timer)
Leaving safepoint region
Total time for which application threads were stopped: 8.2160410 seconds Stopping threads took: 0.0003233 seconds
```

调优案例

我们维护的服务名叫 Zeus，它是美团的规则平台，常用于风控场景中的规则管理。规则运行是基于开源的表达式执行引擎 Aviator。Aviator 内部将每一条表达式转化成 Java 的一个类，通过调用该类的接口实现表达式逻辑。

Zeus 服务内的规则数量超过万条，且每台机器每天的请求量几百万。这些客观条件导致 Aviator 生成的类和方法会产生很多的 ClassLoader 和 CodeCache，这些在使用 ZGC 时都成为过 GC 的性能瓶颈。接下来介绍两类调优案例。

内存分配阻塞，系统停顿可达到秒级

案例一：秒杀活动中流量突增，出现性能毛刺

日志信息：对比出现性能毛刺时间点的 GC 日志和业务日志，发现 JVM 停顿了较长时间，且停顿时 GC 日志中有大量的“Allocation Stall”日志。

分析：这种案例多出现在“自适应算法”为主要 GC 触发机制的场景中。ZGC 是一款并发的垃圾回收器，GC 线程和应用线程同时活动，在 GC 过程中，还会产生新的

对象。GC 完成之前，新产生的对象将堆占满，那么应用线程可能因为申请内存失败而导致线程阻塞。当秒杀活动开始，大量请求打入系统，但自适应算法计算的 GC 触发间隔较长，导致 GC 触发不及时，引起了内存分配阻塞，导致停顿。

解决方法：

（1）开启”基于固定时间间隔“的 GC 触发机制：-XX:ZCollectionInterval。比如调整为 5 秒，甚至更短。

（2）增大修正系数 -XX:ZAllocationSpikeTolerance，更早触发 GC。ZGC 采用正态分布模型预测内存分配速率，模型修正系数 ZAllocationSpikeTolerance 默认值为 2，值越大，越早的触发 GC，Zeus 中所有集群设置的是 5。

案例二：压测时，流量逐渐增大到一定程度后，出现性能毛刺

日志信息：平均 1 秒 GC 一次，两次 GC 之间几乎没有间隔。

分析：GC 触发及时，但内存标记和回收速度过慢，引起内存分配阻塞，导致停顿。

解决方法：增大 -XX:ConcGCThreads，加快并发标记和回收速度。ConcGC-Threads 默认值是核数的 1/8，8 核机器，默认值是 1。该参数影响系统吞吐，如果 GC 间隔时间大于 GC 周期，不建议调整该参数。

GC Roots 数量大，单次 GC 停顿时间长

案例三：单次 GC 停顿时间 30ms，与预期停顿 10ms 左右有较大差距

日志信息：观察 ZGC 日志信息统计，“Pause Roots ClassLoaderDataGraph”一项耗时较长。

分析：dump 内存文件，发现系统中有上万个 ClassLoader 实例。我们知道 Class-Loader 属于 GC Roots 一部分，且 ZGC 停顿时间与 GC Roots 成正比，GC Roots 数量越大，停顿时间越久。再进一步分析，ClassLoader 的类名表明，这些 ClassLoader 均由 Aviator 组件生成。分析 Aviator 源码，发现 Aviator 对每一个表

达式新生成类时，会创建一个 ClassLoader，这导致了 ClassLoader 数量巨大的问题。在更高 Aviator 版本中，该问题已经被修复，即仅创建一个 ClassLoader 为所有表达式生成类。

解决方法：升级 Aviator 组件版本，避免生成多余的 ClassLoader。

案例四：服务启动后，运行时间越长，单次 GC 时间越长，重启后恢复

日志信息：观察 ZGC 日志信息统计，“Pause Roots CodeCache”的耗时会随着服务运行时间逐渐增长。

分析：CodeCache 空间用于存放 Java 热点代码的 JIT 编译结果，而 CodeCache 也属于 GC Roots 一部分。通过添加 -XX:+PrintCodeCacheOnCompilation 参数，打印 CodeCache 中的被优化的方法，发现大量的 Aviator 表达式代码。定位到根本原因，每个表达式都是一个类中一个方法。随着运行时间越长，执行次数增加，这些方法会被 JIT 优化编译进入到 Code Cache 中，导致 CodeCache 越来越大。

解决方法：JIT 有一些参数配置可以调整 JIT 编译的条件，但对于我们的问题都不太适用。我们最终通过业务优化解决，删除不需要执行的 Aviator 表达式，从而避免了大量 Aviator 方法进入 CodeCache 中。

值得一提的是，我们并不是在所有这些问题都解决后才全量部署所有集群。即使开始有各种各样的毛刺，但计算后发现，有各种问题的 ZGC 也比之前的 CMS 对服务可用性影响小。所以从开始准备使用 ZGC 到全量部署，大概用了 2 周的时间。在之后的 3 个月时间里，我们边做业务需求，边跟进这些问题，最终逐个解决了上述问题，从而使 ZGC 在各个集群上达到了一个更好表现。

升级 ZGC 效果

延迟降低

TP(Top Percentile) 是一项衡量系统延迟的指标：TP999 表示 99.9% 请求都能被响

应的最小耗时；TP99 表示 99% 请求都能被响应的最小耗时。

在 Zeus 服务不同集群中，ZGC 在低延迟（TP999 < 200ms）场景中收益较大：

- **TP999**：下降 12~142ms，下降幅度 18%~74%。
- **TP99**：下降 5~28ms，下降幅度 10%~47%。

超低延迟（TP999 < 20ms）和高延迟（TP999 > 200ms）服务收益不大，原因是这些服务的响应时间瓶颈不是 GC，而是外部依赖的性能。

吞吐下降

对吞吐量优先的场景，ZGC 可能并不适合。例如，Zeus 某离线集群原先使用 CMS，升级 ZGC 后，系统吞吐量明显降低。究其原因有二：第一，ZGC 是单代垃圾回收器，而 CMS 是分代垃圾回收器。单代垃圾回收器每次处理的对象更多，更耗费 CPU 资源；第二，ZGC 使用读屏障，读屏障操作需耗费额外的计算资源。

总结

ZGC 作为下一代垃圾回收器，性能非常优秀。ZGC 垃圾回收过程几乎全部是并发，实际 STW 停顿时间极短，不到 10ms。这得益于其采用的着色指针和读屏障技术。

Zeus 在升级 JDK 11+ZGC 中，通过将风险和问题分类，然后各个击破，最终顺利实现了升级目标，GC 停顿也几乎不再影响系统可用性。

最后推荐大家升级 ZGC，Zeus 系统因为业务特点，遇到了较多问题，而风控其他团队在升级时都非常顺利。欢迎大家加入“ZGC 使用交流”群。

参考文献

ZGC 官网
彭成寒 .《新一代垃圾回收器 ZGC 设计与实现》. 机械工业出版社 , 2019.
从实际案例聊聊 Java 应用的 GC 优化
Java Hotspot G1 GC 的一些关键技术

附录

如何使用新技术

在生产环境升级 JDK 11，使用 ZGC，大家最关心的可能不是效果怎么样，而是这个新版本用的人少，网上实践也少，靠不靠谱，稳不稳定。其次是升级成本会不会很大，万一不成功岂不是白白浪费时间。所以，在使用新技术前，首先要做的是评估收益、成本和风险。

评估收益

对于 JDK 这种世界关注的程序，大版本升级所引入的新技术一般已经在理论上经过验证。我们要做的事情就是确定当前系统的瓶颈是否是新版本 JDK 可解决的问题，切忌问题未诊断清楚就采取措施。评估完收益之后再评估成本和风险，收益过大或者过小，其他两项影响权重就会小很多。

以本文开头提到的案例为例，假设 GC 次数不变（10 次 / 分钟），且单次 GC 时间从 40ms 降低 10ms。通过计算，一分钟内有 100/60000 = 0.17% 的时间在进行 GC，且期间所有请求仅停顿 10ms，GC 期间影响的请求数和因 GC 增加的延迟都有所减少。

评估成本

这里主要指升级所需要的人力成本。此项相对比较成熟，根据新技术的使用手册判断改动点。跟做其他项目区别不大，不再具体细说。

在我们的实践中，两周时间完成线上部署，达到安全稳定运行的状态。后续持续迭代 3 个月，根据业务场景对 ZGC 进行了更契合的优化适配。

评估风险

升级 JDK 的风险可以分为三类：

- **兼容性风险**：Java 程序 JAR 包依赖很多，升级 JDK 版本后程序是否能运行

起来。例如我们的服务是从 JDK 7 升级到 JDK 11，需要解决较多 JAR 包不兼容的问题。

- **功能风险**：运行起来后，是否会有一些组件逻辑变更，影响现有功能的逻辑。
- **性能风险**：功能如果没有问题，性能是否稳定，能稳定的在线上运行。

经过分类后，每类风险的应对转化成了常见的测试问题，不再属于未知风险。风险是指不确定的事情，如果不确定的事情都能转化成可确定的事情，意味着风险已消除。

升级 JDK 11

选择 JDK 11，是因为在 JDK 11 中首次支持 ZGC，而且 JDK 11 属于长期支持（Long Term Support，LTS）版本，至少会被维护三年，普通版本（如 JDK 12、JDK 13 和 JDK 14）只有 6 个月的维护周期，不建议使用。

本地测试环境安装

从两个源 OpenJDK 和 OracleJDK 下载 JDK 11，二个版本的 JDK 主要区别是长时期的免费和付费，短期内都免费。注意 JDK 11 版本中的 ZGC 不支持 Mac OS 系统，在 Mac OS 系统上使用 JDK 11 只能用其他垃圾回收器，如 G1。

生产环境安装

升级 JDK 11 不仅仅是升级自己项目的 JDK 版本，还需要编译、发布部署、运行、监控、性能内存分析工具等项目支持。美团内部的实践：

编译打包：美团发布系统支持选择 JDK 11 进行编译打包。 **线上运行 & 全量部署**：要求线上机器已安装 JDK11，有 3 种方式：

1. 新申请默认安装 JDK 11 的虚拟机：试用 JDK 11 时可用这种方式；全量部署时，如果新申请机器数量过多，可能没有足够机器资源。
2. 通过手写脚本给存量虚拟机安装 JDK 11：不推荐，业务同学过多参与到运维当中。

3. 使用容器提供的镜像部署功能，在打包镜像时安装 JDK 11：推荐方式，不需要新申请资源。

监控指标：主要是 GC 的时间和频率，我们通过美团的 CAT 监控系统支持 ZGC 数据的收集（CAT 已开源）。**性能内存分析**：线上遇到性能问题时，还需要借助 Profiling 工具，美团的性能诊断优化平台 Scalpel 已支持 JDK 11 的性能内存分析。如果你的公司没有相关工具，推荐使用 JProfier。

解决组件兼容性

我们的项目包含二十多万行代码，需要从 JDK 7 升级到 JDK 11，依赖组件众多。虽然看起来升级会比较复杂，但实际只花了两天时间即解决了兼容性问题。具体过程如下：

1. 编译，需要修改 pom 文件中的 build 配置，根据报错作修改，主要有两类：

a. 一些类被删除：比如“sun.misc.BASE64Encoder”，找到替换类 java.util.Base64 即可。

b. 组件依赖版本不兼容 JDK 11 问题：找到对应依赖组件，搜索最新版本，一般都支持 JDK 11。

2. 编译成功后，启动运行，此时仍有可能组件依赖版本问题，按照编译时的方式处理即可。

升级所修改的依赖：

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
```

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.4</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-parent</artifactId>
    <version>6.0.16.Final</version>
</dependency>
<dependency>
    <groupId>com.sankuai.inf</groupId>
    <artifactId>patriot-sdk</artifactId>
    <version>1.2.1</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.9</version>
</dependency>
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.39.Final</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

JDK 11 已经出来两年，常见的依赖组件都有兼容性版本。但是，如果是公司内部提供的公司级组件，可能会不兼容 JDK 11，需要推动相关组件进行升级。如果对方升级较为困难，可以考虑拆分功能，将依赖这些组件的功能单独部署，继续使用低版本 JDK。随着 JDK11 的卓越性能被大家悉知，相信会有更多团队会用 JDK 11 解决 GC 问题，使用者越多，各个组件升级的动力也会越大。

验证功能正确性

通过完备的单测、集成和回归测试，保证功能正确性。

作者简介

王东，美团信息安全资深工程师。
王伟，美团信息安全技术专家。

设计模式在外卖营销业务中的实践

作者：亮亮

一、前言

随着美团外卖业务的不断迭代与发展，外卖用户数量也在高速地增长。在这个过程中，外卖营销发挥了“中流砥柱”的作用，因为用户的快速增长离不开高效的营销策略。而由于市场环境和业务环境的多变，营销策略往往是复杂多变的，营销技术团队作为营销业务的支持部门，就需要快速高效地响应营销策略变更带来的需求变动。因此，设计并实现易于扩展和维护的营销系统，是美团外卖营销技术团队不懈追求的目标和必修的基本功。

本文通过自顶向下的方式，来介绍设计模式如何帮助我们构建一套易扩展、易维护的营销系统。本文会首先介绍设计模式与领域驱动设计（Domain-Driven Design，以下简称为 DDD）之间的关系，然后再阐述外卖营销业务引入业务中用到的设计模式以及其具体实践案例。

二、设计模式与领域驱动设计

设计一个营销系统，我们通常的做法是采用自顶向下的方式来解构业务，为此我们引入了 DDD。从战略层面上讲，DDD 能够指导我们完成从问题空间到解决方案的剖析，将业务需求映射为领域上下文以及上下文间的映射关系。从战术层面上，DDD 能够细化领域上下文，并形成有效的、细化的领域模型来指导工程实践。建立领域模型的一个关键意义在于，能够确保不断扩展和变化的需求在领域模型内不断地演进和发展，而不至于出现模型的腐化和领域逻辑的外溢。关于 DDD 的实践，大家可以参考此前美团技术团队推出的《领域驱动设计在互联网业务开发中的实践》一文。

同时，我们也需要在代码工程中贯彻和实现领域模型。因为代码工程是领域模型在工程实践中的直观体现，也是领域模型在技术层面的直接表述。而设计模式，可以说是连接领域模型与代码工程的一座桥梁，它能有效地解决从领域模型到代码工程的转化。

为什么说设计模式天然具备成为领域模型到代码工程之间桥梁的作用呢？其实，2003 年出版的《领域驱动设计》一书的作者 Eric Evans 在这部开山之作中就已经给出了解释。他认为，立场不同会影响人们如何看待什么是“模式”。因此，无论是领域驱动模式还是设计模式，本质上都是“模式”，只是解决的问题不一样。站在业务建模的立场上，DDD 的模式解决的是如何进行领域建模。而站在代码实践的立场上，设计模式主要关注于代码的设计与实现。既然本质都是模式，那么它们天然就具有一定的共通之处。

所谓“模式”，就是一套反复被人使用或验证过的方法论。从抽象或者更宏观的角度上看，只要符合使用场景并且能解决实际问题，模式应该既可以应用在 DDD 中，也可以应用在设计模式中。事实上，Evans 也是这么做的。他在著作中阐述了 Strategy 和 Composite 这两个传统的 GOF 设计模式是如何来解决领域模型建设的。因此，当领域模型需要转化为代码工程时，同构的模式，天然能够将领域模型翻译成代码模型。

三、设计模式在外卖营销业务中的具体案例

3.1 为什么需要设计模式

营销业务的特点

如前文所述，营销业务与交易等其他模式相对稳定的业务的区别在于，营销需求会随着市场、用户、环境的不断变化而进行调整。也正是因此，外卖营销技术团队选择了 DDD 进行领域建模，并在适用的场景下，用设计模式在代码工程的层面上实践和反映了领域模型。以此来做到在支持业务变化的同时，让领域和代码模型健康演进，避

免模型腐化。

理解设计模式

软件设计模式（Design pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码，让代码更容易被他人理解，保证代码可靠性，程序的重用性。可以理解为："世上本来没有设计模式，用的人多了，便总结出了一套设计模式。"

设计模式原则

面向对象的设计模式有七大基本原则：

- 开闭原则（Open Closed Principle，OCP）
- 单一职责原则（Single Responsibility Principle, SRP）
- 里氏代换原则（Liskov Substitution Principle，LSP）
- 依赖倒转原则（Dependency Inversion Principle，DIP）
- 接口隔离原则（Interface Segregation Principle，ISP）
- 合成 / 聚合复用原则（Composite/Aggregate Reuse Principle，CARP）
- 最少知识原则（Least Knowledge Principle，LKP）或者迪米特法则（Law of Demeter，LOD）

简单理解就是：开闭原则是总纲，它指导我们要对扩展开放，对修改关闭；单一职责原则指导我们实现类要职责单一；里氏替换原则指导我们不要破坏继承体系；依赖倒置原则指导我们要面向接口编程；接口隔离原则指导我们在设计接口的时候要精简单一；迪米特法则指导我们要降低耦合。

设计模式就是通过这七个原则，来指导我们如何做一个好的设计。但是设计模式不是一套"奇技淫巧"，它是一套方法论，一种高内聚、低耦合的设计思想。我们可以在此基础上自由的发挥，甚至设计出自己的一套设计模式。

当然，学习设计模式或者是在工程中实践设计模式，必须深入到某一个特定的业务场

景中去，再结合对业务场景的理解和领域模型的建立，才能体会到设计模式思想的精髓。如果脱离具体的业务逻辑去学习或者使用设计模式，那是极其空洞的。接下来我们将通过外卖营销业务的实践，来探讨如何用设计模式来实现可重用、易维护的代码。

3.2 “邀请下单”业务中设计模式的实践

3.2.1 业务简介

“邀请下单”是美团外卖用户邀请其他用户下单后给予奖励的平台。即用户 A 邀请用户 B，并且用户 B 在美团下单后，给予用户 A 一定的现金奖励（以下简称返奖）。同时为了协调成本与收益的关系，返奖会有多个计算策略。邀请下单后台主要涉及两个技术要点：

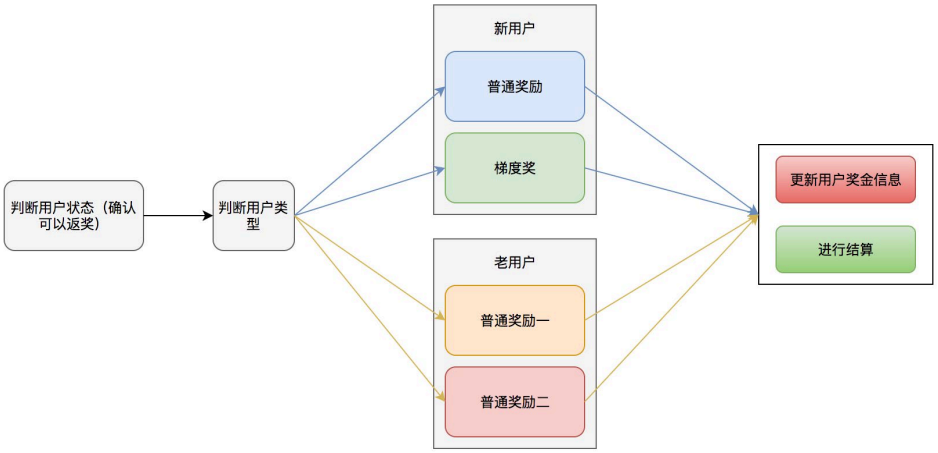
1. 返奖金额的计算，涉及到不同的计算规则。
2. 从邀请开始到返奖结束的整个流程。



3.2.2 返奖规则与设计模式实践

业务建模

如图是返奖规则计算的业务逻辑视图：



从这份业务逻辑图中可以看到返奖金额计算的规则。首先要根据用户状态确定用户是否满足返奖条件。如果满足返奖条件，则继续判断当前用户属于新用户还是老用户，从而给予不同的奖励方案。一共涉及以下几种不同的奖励方案：

新用户

- 普通奖励（给予固定金额的奖励）
- 梯度奖（根据用户邀请的人数给予不同的奖励金额，邀请的人越多，奖励金额越多）

老用户

- 根据老用户的用户属性来计算返奖金额。为了评估不同的邀新效果，老用户返奖会存在多种返奖机制。

计算完奖励金额以后，还需要更新用户的奖金信息，以及通知结算服务对用户的金额进行结算。这两个模块对于所有的奖励来说都是一样的。

可以看到，无论是何种用户，对于整体返奖流程是不变的，唯一变化的是返奖规则。此处，我们可参考**开闭原则**，对于返奖流程保持封闭，对于可能扩展的返奖规则进行开放。我们将返奖规则抽象为**返奖策略**，即针对不同用户类型的不同返奖方案，我们

视为不同的返奖策略，不同的返奖策略会产生不同的返奖金额结果。

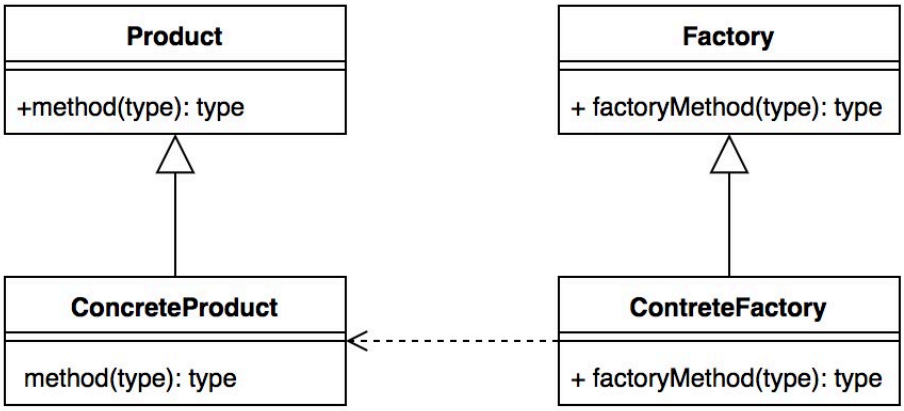
在我们的领域模型里，返奖策略是一个**值对象**，我们通过工厂的方式生产针对不同用户的奖励策略值对象。下文我们将介绍以上领域模型的工程实现，即**工厂模式**和**策略模式**的实际应用。

模式：工厂模式

工厂模式又细分为工厂方法模式和抽象工厂模式，本文主要介绍工厂方法模式。

模式定义：定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法是一个类的实例化延迟到其子类。

工厂模式通用类图如下：



我们通过一段较为通用的代码来解释如何使用工厂模式：

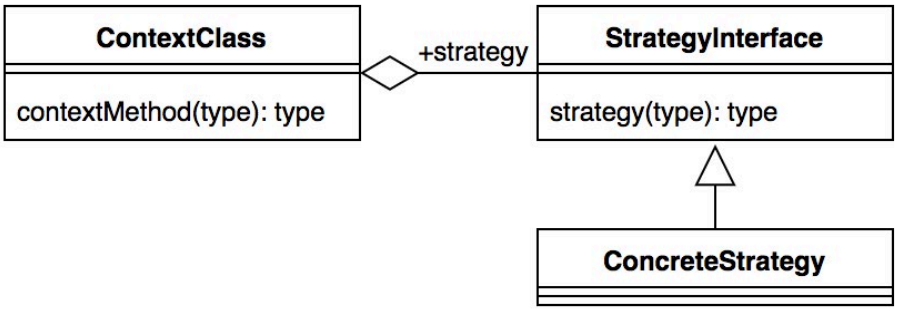
```
// 抽象的产品
public abstract class Product {
    public abstract void method();
}
// 定义一个具体的产品 （可以定义多个具体的产品）
class ProductA extends Product {
    @Override
    public void method() {}  // 具体的执行逻辑
}
// 抽象的工厂
abstract class Factory<T> {
```

```
    abstract Product createProduct(Class<T> c);
}
// 具体的工厂可以生产出相应的产品
class FactoryA extends Factory{
    @Override
    Product createProduct(Class c) {
        Product product = (Product) Class.forName(c.getName()).
newInstance();
        return product;
    }
}
```

模式：策略模式

模式定义：定义一系列算法，将每个算法都封装起来，并且它们可以互换。策略模式是一种对象行为模式。

策略模式通用类图如下：



我们通过一段比较通用的代码来解释怎么使用策略模式：

```
// 定义一个策略接口
public interface Strategy {
    void strategyImplementation();
}

// 具体的策略实现（可以定义多个具体的策略实现）
public class StrategyA implements Strategy{
    @Override
    public void strategyImplementation() {
        System.out.println("正在执行策略A");
    }
}
```

```
// 封装策略，屏蔽高层模块对策略、算法的直接访问，屏蔽可能存在的策略变化
public class Context {
    private Strategy strategy = null;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void doStrategy() {
        strategy.strategyImplementation();
    }
}
```

工程实践

通过上文介绍的返奖业务模型，我们可以看到返奖的主流程就是选择不同的返奖策略的过程，每个返奖策略都包括返奖金额计算、更新用户奖金信息、以及结算这三个步骤。 我们可以使用工厂模式生产出不同的策略，同时使用策略模式来进行不同的策略执行。首先确定我们需要生成出 n 种不同的返奖策略，其编码如下：

```
// 抽象策略
public abstract class RewardStrategy {
    public abstract void reward(long userId);

    public void insertRewardAndSettlement(long userId, int reward) {}
; // 更新用户信息以及结算
}
// 新用户返奖具体策略 A
public class newUserRewardStrategyA extends RewardStrategy {
    @Override
    public void reward(long userId) {}  // 具体的计算逻辑，...
}

// 老用户返奖具体策略 A
public class OldUserRewardStrategyA extends RewardStrategy {
    @Override
    public void reward(long userId) {}  // 具体的计算逻辑，...
}

// 抽象工厂
public abstract class StrategyFactory<T> {
    abstract RewardStrategy createStrategy(Class<T> c);
}
```

```
// 具体工厂创建具体的策略
public class FactorRewardStrategyFactory extends StrategyFactory {
    @Override
    RewardStrategy createStrategy(Class c) {
        RewardStrategy product = null;
        try {
            product = (RewardStrategy) Class.forName(c.getName()).
newInstance();
        } catch (Exception e) {}
        return product;
    }
}
```

通过工厂模式生产出具体的策略之后，根据我们之前的介绍，很容易就可以想到使用策略模式来执行我们的策略。具体代码如下：

```
public class RewardContext {
    private RewardStrategy strategy;

    public RewardContext(RewardStrategy strategy) {
        this.strategy = strategy;
    }

    public void doStrategy(long userId) {
        int rewardMoney = strategy.reward(userId);
        insertRewardAndSettlement(long userId, int reward) {
          insertReward(userId, rewardMoney);
          settlement(userId);
       }
    }
}
```

接下来我们将工厂模式和策略模式结合在一起，就完成了整个返奖的过程：

```
public class InviteRewardImpl {
    // 返奖主流程
    public void sendReward(long userId) {
        FactorRewardStrategyFactory strategyFactory = new
FactorRewardStrategyFactory();  // 创建工厂
        Invitee invitee = getInviteeByUserId(userId);  // 根据用户 id 查询用
户信息
        if (invitee.userType == UserTypeEnum.NEW_USER) {  // 新用户返奖策略
            NewUserBasicReward newUserBasicReward = (NewUserBasicReward)
strategyFactory.createStrategy(NewUserBasicReward.class);
            RewardContext rewardContext = new
RewardContext(newUserBasicReward);
```

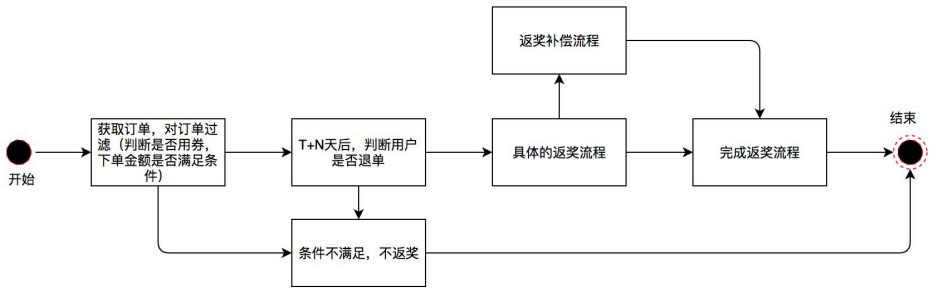
```
            rewardContext.doStrategy(userId); // 执行返奖策略
        }if(invitee.userType == UserTypeEnum.OLD_USER){}  // 老用户返奖策略,
...
    }
}
```

工厂方法模式帮助我们直接产生一个具体的策略对象，策略模式帮助我们保证这些策略对象可以自由地切换而不需要改动其他逻辑，从而达到解耦的目的。通过这两个模式的组合，当我们系统需要增加一种返奖策略时，只需要实现 RewardStrategy 接口即可，无需考虑其他的改动。当我们需要改变策略时，只要修改策略的类名即可。不仅增强了系统的可扩展性，避免了大量的条件判断，而且从真正意义上达到了高内聚、低耦合的目的。

3.2.3 返奖流程与设计模式实践

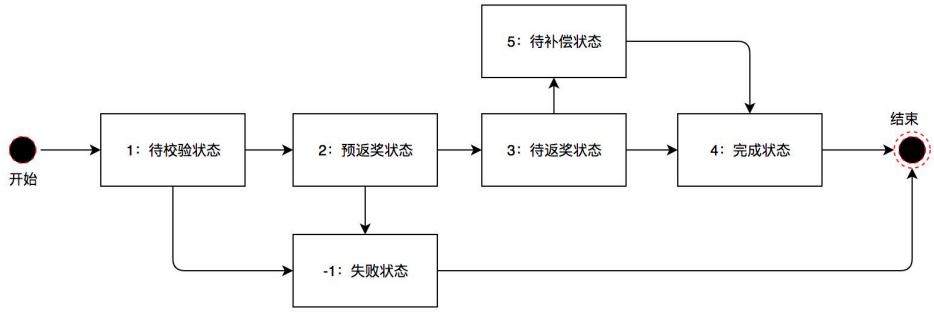
业务建模

当受邀人在接受邀请人的邀请并且下单后，返奖后台接收到受邀人的下单记录，此时邀请人也进入返奖流程。首先我们订阅用户订单消息并对订单进行返奖规则校验。例如，是否使用红包下单，是否在红包有效期内下单，订单是否满足一定的优惠金额等等条件。当满足这些条件以后，我们将订单信息放入延迟队列中进行后续处理。经过 T+N 天之后处理该延迟消息，判断用户是否对该订单进行了退款，如果未退款，对用户进行返奖。若返奖失败，后台还有返奖补偿流程，再次进行返奖。其流程如下图所示：



我们对上述业务流程进行领域建模：

1. 在接收到订单消息后，用户进入待校验状态；
2. 在校验后，若校验通过，用户进入预返奖状态，并放入延迟队列。若校验未通过，用户进入不返奖状态，结束流程；
3. T+N 天后，处理延迟消息，若用户未退款，进入待返奖状态。若用户退款，进入失败状态，结束流程；
4. 执行返奖，若返奖成功，进入完成状态，结束流程。若返奖不成功，进入待补偿状态；
5. 待补偿状态的用户会由任务定期触发补偿机制，直至返奖成功，进入完成状态，保障流程结束。

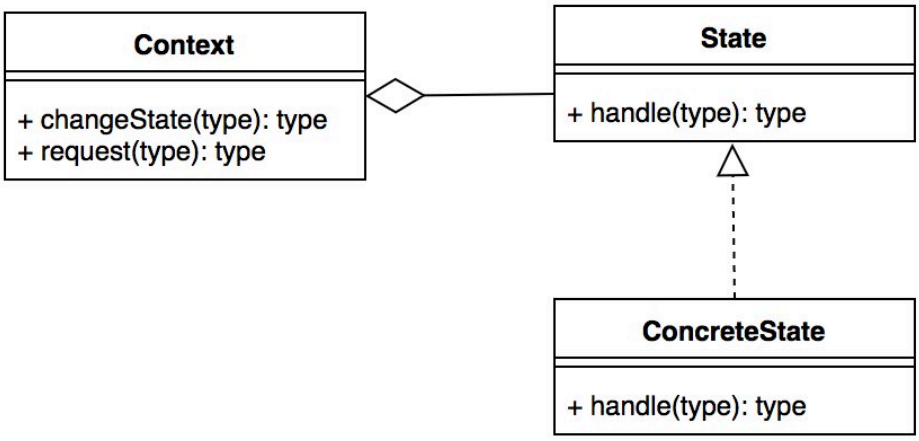


可以看到，我们通过建模将返奖流程的多个步骤映射为系统的状态。对于系统状态的表述，DDD 中常用到的概念是领域事件，另外也提及过事件溯源的实践方案。当然，在设计模式中，也有一种能够表述系统状态的代码模型，那就是状态模式。在邀请下单系统中，我们的主要流程是返奖。对于返奖，每一个状态要进行的动作和操作都是不同的。因此，使用状态模式，能够帮助我们对系统状态以及状态间的流转进行统一的管理和扩展。

模式：状态模式

模式定义：当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。

状态模式的通用类图如下图所示：



对比策略模式的类型会发现和状态模式的类图很类似，但实际上有很大的区别，具体体现在 concrete class 上。策略模式通过 Context 产生唯一一个 ConcreteStrategy 作用于代码中，而状态模式则是通过 context 组织多个 ConcreteState 形成一个状态转换图来实现业务逻辑。接下来，我们通过一段通用代码来解释怎么使用状态模式：

```
// 定义一个抽象的状态类
public abstract class State {
    Context context;
    public void setContext(Context context) {
        this.context = context;
    }
    public abstract void handle1();
    public abstract void handle2();
}
// 定义状态 A
public class ConcreteStateA extends State {
    @Override
    public void handle1() {}  // 本状态下必须要处理的事情

    @Override
    public void handle2() {
        super.context.setCurrentState(Context.contreteStateB);  // 切换到
状态 B
        super.context.handle2();  // 执行状态 B 的任务
```

```
    }
}
// 定义状态 B
public class ConcreteStateB extends State {
    @Override
    public void handle2() {}  // 本状态下必须要处理的事情，...

    @Override
    public void handle1() {
        super.context.setCurrentState(Context.contreteStateA);  // 切换到
状态 A
        super.context.handle1();  // 执行状态 A 的任务
    }
}
// 定义一个上下文管理环境
public class Context {
    public final static ConcreteStateA contreteStateA = new
ConcreteStateA();
    public final static ConcreteStateB contreteStateB = new
ConcreteStateB();

    private State CurrentState;
    public State getCurrentState() {return CurrentState;}

    public void setCurrentState(State currentState) {
        this.CurrentState = currentState;
        this.CurrentState.setContext(this);
    }

    public void handle1() {this.CurrentState.handle1();}
    public void handle2() {this.CurrentState.handle2();}
}
// 定义 client 执行
public class client {
    public static void main(String[] args) {
        Context context = new Context();
        context.setCurrentState(new ContreteStateA());
        context.handle1();
        context.handle2();
    }
}
```

工程实践

通过前文对状态模式的简介，我们可以看到当状态之间的转换在不是非常复杂的情况下，通用的状态模式存在大量的与状态无关的动作从而产生大量的无用代码。在我们

的实践中，一个状态的下游不会涉及特别多的状态装换，所以我们简化了状态模式。当前的状态只负责当前状态要处理的事情，状态的流转则由第三方类负责。其实践代码如下：

```
// 返奖状态执行的上下文
public class RewardStateContext {

    private RewardState rewardState;

    public void setRewardState(RewardState currentState) {this.
rewardState = currentState;}
    public RewardState getRewardState() {return rewardState;}
    public void echo(RewardStateContext context, Request request) {
        rewardState.doReward(context, request);
    }
}

public abstract class RewardState {
    abstract void doReward(RewardStateContext context, Request
request);
}

// 待校验状态
public class OrderCheckState extends RewardState {
    @Override
    public void doReward(RewardStateContext context, Request request)
{
        orderCheck(context, request);  // 对进来的订单进行校验，判断是否用券，
是否满足优惠条件等等
    }
}

// 待补偿状态
public class CompensateRewardState extends RewardState {
    @Override
    public void doReward(RewardStateContext context, Request request)
{
        compensateReward(context, request);  // 返奖失败，需要对用户进行返奖
补偿
    }
}

// 预返奖状态，待返奖状态，成功状态，失败状态（此处逻辑省略）
//..

public class InviteRewardServiceImpl {
    public boolean sendRewardForInvtee(long userId, long orderId) {
```

```
        Request request = new Request(userId, orderId);
        RewardStateContext rewardContext = new RewardStateContext();
        rewardContext.setRewardState(new OrderCheckState());
        rewardContext.echo(rewardContext, request);  // 开始返奖，订单校验
        // 此处的 if-else 逻辑只是为了表达状态的转换过程，并非实际的业务逻辑
        if (rewardContext.isResultFlag()) {  // 如果订单校验成功，进入预返奖
状态
            rewardContext.setRewardState(new BeforeRewardCheckState());
            rewardContext.echo(rewardContext, request);
        } else {// 如果订单校验失败，进入返奖失败流程，...
            rewardContext.setRewardState(new RewardFailedState());
            rewardContext.echo(rewardContext, request);
            return false;
        }
        if (rewardContext.isResultFlag()) {// 预返奖检查成功，进入待返奖流程，
...
            rewardContext.setRewardState(new SendRewardState());
            rewardContext.echo(rewardContext, request);
        } else {  // 如果预返奖检查失败，进入返奖失败流程，...
            rewardContext.setRewardState(new RewardFailedState());
            rewardContext.echo(rewardContext, request);
            return false;
        }
        if (rewardContext.isResultFlag()) {  // 返奖成功，进入返奖结束流程，
...
            rewardContext.setRewardState(new RewardSuccessState());
            rewardContext.echo(rewardContext, request);
        } else {  // 返奖失败，进入返奖补偿阶段，...
            rewardContext.setRewardState(new CompensateRewardState());
            rewardContext.echo(rewardContext, request);
        }
        if (rewardContext.isResultFlag()) {  // 补偿成功，进入返奖完成阶段，
...
            rewardContext.setRewardState(new RewardSuccessState());
            rewardContext.echo(rewardContext, request);
        } else {  // 补偿失败，仍然停留在当前态，直至补偿成功（或多次补偿失败后人
工介入处理）
            rewardContext.setRewardState(new CompensateRewardState());
            rewardContext.echo(rewardContext, request);
        }
        return true;
    }
}
```

状态模式的核心是封装，将状态以及状态转换逻辑封装到类的内部来实现，也很好的体现了“开闭原则”和“单一职责原则”。每一个状态都是一个子类，不管是修改还是增加状态，只需要修改或者增加一个子类即可。在我们的应用场景中，状态数量以

及状态转换远比上述例子复杂，通过“状态模式”避免了大量的 if-else 代码，让我们的逻辑变得更加清晰。同时由于状态模式的良好的封装性以及遵循的设计原则，让我们在复杂的业务场景中，能够游刃有余地管理各个状态。

3.3 点评外卖投放系统中设计模式的实践

3.3.1 业务简介

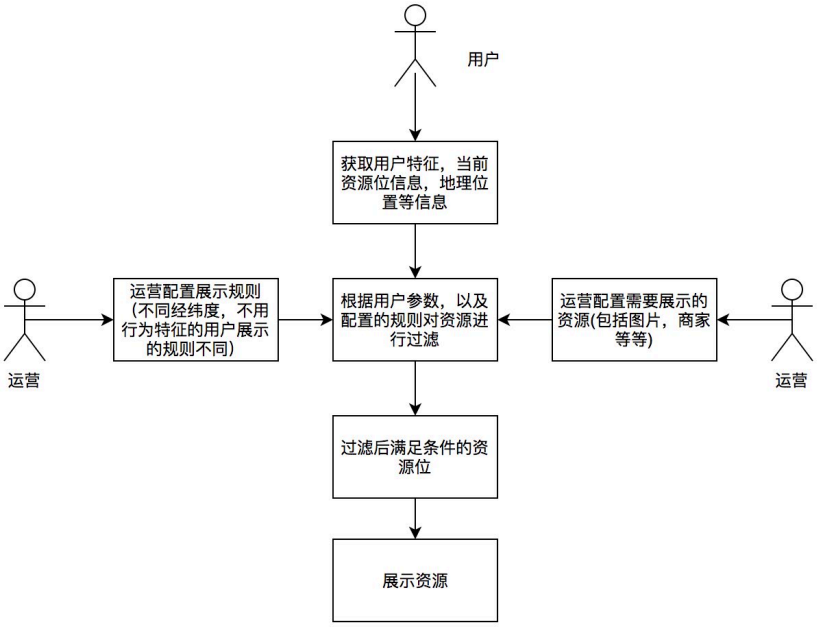
继续举例，点评 App 的外卖频道中会预留多个资源位为营销使用，向用户展示一些比较精品美味的外卖食品，为了增加用户点外卖的意向。当用户点击点评首页的“美团外卖”入口时，资源位开始加载，会通过一些规则来筛选出合适的展示 Banner。



3.3.2 设计模式实践

业务建模

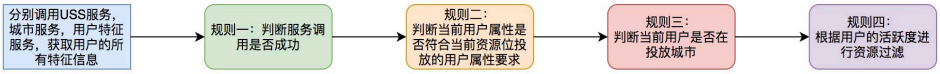
对于投放业务，就是要在这些资源位中展示符合当前用户的资源。其流程如下图所示：



从流程中我们可以看到，首先运营人员会配置需要展示的资源，以及对资源进行过滤的规则。我们资源的过滤规则相对灵活多变，这里体现为三点：

1. 过滤规则大部分可重用，但也会有扩展和变更。
2. 不同资源位的过滤规则和过滤顺序是不同的。
3. 同一个资源位由于业务所处的不同阶段，过滤规则可能不同。

过滤规则本身是一个个的值对象，我们通过领域服务的方式，操作这些规则值对象完成资源位的过滤逻辑。下图介绍了资源位在进行用户特征相关规则过滤时的过程：

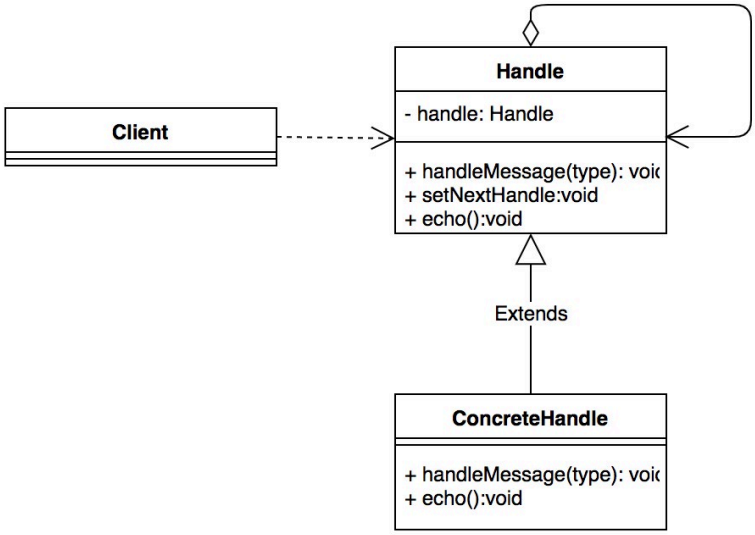


为了实现过滤规则的解耦，对单个规则值对象的修改封闭，并对规则集合组成的过滤链条开放，我们在资源位过滤的领域服务中引入了责任链模式。

模式：责任链模式

模式定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

责任链模式通用类图如下：



我们通过一段比较通用的代码来解释如何使用责任链模式：

```
// 定义一个抽象的 handle
public abstract class Handler {
    private Handler nextHandler;  // 指向下一个处理者
    private int level;  // 处理者能够处理的级别

    public Handler(int level) {
        this.level = level;
    }

    public void setNextHandler(Handler handler) {
        this.nextHandler = handler;
    }

    // 处理请求传递，注意 final，子类不可重写
    public final void handleMessage(Request request) {
```

```
        if (level == request.getRequstLevel()) {
            this.echo(request);
        } else {
            if (this.nextHandler != null) {
                this.nextHandler.handleMessage(request);
            } else {
                System.out.println(" 已经到最尽头了 ");
            }
        }
    }
    // 抽象方法，子类实现
    public abstract void echo(Request request);
}

// 定义一个具体的 handleA
public class HandleRuleA extends Handler {
    public HandleRuleA(int level) {
        super(level);
    }
    @Override
    public void echo(Request request) {
        System.out.println(" 我是处理者 1，我正在处理 A 规则 ");
    }
}

// 定义一个具体的 handleB
public class HandleRuleB extends Handler {}  //...

// 客户端实现
class Client {
    public static void main(String[] args) {
        HandleRuleA handleRuleA = new HandleRuleA(1);
        HandleRuleB handleRuleB = new HandleRuleB(2);
        handleRuleA.setNextHandler(handleRuleB);  // 这是重点，将 handleA 和
handleB 串起来
        handleRuleA.echo(new Request());
    }
}
```

工程实践

下面通过代码向大家展示如何实现这一套流程：

```
// 定义一个抽象的规则
public abstract class BasicRule<CORE_ITEM, T extends RuleContext<CORE_
ITEM>>{
    // 有两个方法，evaluate 用于判断是否经过规则执行，execute 用于执行具体的规则
内容。
```

```
    public abstract boolean evaluate(T context);
    public abstract void execute(T context) {
}

// 定义所有的规则具体实现
// 规则 1：判断服务可用性
public class ServiceAvailableRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {
    @Override
    public boolean evaluate(UserPortraitRuleContext context) {
        TakeawayUserPortraitBasicInfo basicInfo = context.getBasicInfo();
        if (basicInfo.isServiceFail()) {
              return false;
        }
        return true;
    }

    @Override
    public void execute(UserPortraitRuleContext context) {}

}
// 规则 2：判断当前用户属性是否符合当前资源位投放的用户属性要求
public class UserGroupRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {
    @Override
    public boolean evaluate(UserPortraitRuleContext context) {}

    @Override
    public void execute(UserPortraitRuleContext context) {
        UserPortrait userPortraitPO = context.getData();
        if(userPortraitPO.getUserGroup() == context.getBasicInfo().
getUserGroup().code) {
          context.setValid(true);
        } else {
          context.setValid(false);
        }
    }
}

// 规则 3：判断当前用户是否在投放城市，具体逻辑省略
public class CityInfoRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {}
// 规则 4：根据用户的活跃度进行资源过滤，具体逻辑省略
public class UserPortraitRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {}

// 我们通过 spring 将这些规则串起来组成一个一个请求链
    <bean name="serviceAvailableRule" class="com.dianping.takeaway.
ServiceAvailableRule"/>
```

```
    <bean name="userGroupValidRule" class="com.dianping.takeaway.
UserGroupRule"/>
    <bean name="cityInfoValidRule" class="com.dianping.takeaway.
CityInfoRule"/>
    <bean name="userPortraitRule" class="com.dianping.takeaway.
UserPortraitRule"/>

    <util:list id="userPortraitRuleChain" value-type="com.dianping.
takeaway.Rule">
        <ref bean="serviceAvailableRule"/>
        <ref bean="userGroupValidRule"/>
        <ref bean="cityInfoValidRule"/>
        <ref bean="userPortraitRule"/>
    </util:list>

// 规则执行
public class DefaultRuleEngine{
    @Autowired
    List<BasicRule> userPortraitRuleChain;

    public void invokeAll(RuleContext ruleContext) {
        for(Rule rule : userPortraitRuleChain) {
            rule.evaluate(ruleContext)
        }
    }
}
```

责任链模式最重要的优点就是解耦，将客户端与处理者分开，客户端不需要了解是哪个处理者对事件进行处理，处理者也不需要知道处理的整个流程。在我们的系统中，后台的过滤规则会经常变动，规则和规则之间可能也会存在传递关系，通过责任链模式，我们将规则与规则分开，将规则与规则之间的传递关系通过 Spring 注入到 List 中，形成一个链的关系。当增加一个规则时，只需要实现 BasicRule 接口，然后将新增的规则按照顺序加入 Spring 中即可。当删除时，只需删除相关规则即可，不需要考虑代码的其他逻辑。从而显著地提高了代码的灵活性，提高了代码的开发效率，同时也保证了系统的稳定性。

四、总结

本文从营销业务出发，介绍了领域模型到代码工程之间的转化，从 DDD 引出了设计模式，详细介绍了工厂方法模式、策略模式、责任链模式以及状态模式这四种模式在

营销业务中的具体实现。除了这四种模式以外，我们的代码工程中还大量使用了代理模式、单例模式、适配器模式等等，例如在我们对 DDD 防腐层的实现就使用了适配器模式，通过适配器模式屏蔽了业务逻辑与第三方服务的交互。因篇幅原因不再进行过多的阐述。

对于营销业务来说，业务策略多变导致需求多变是我们面临的主要问题。如何应对复杂多变的需求，是我们提炼领域模型和实现代码模型时必须要考虑的内容。DDD 以及设计模式提供了一套相对完整的方法论帮助我们完成了领域建模及工程实现。其实，设计模式就像一面镜子，将领域模型映射到代码模型中，切实地提高代码的复用性、可扩展性，也提高了系统的可维护性。

当然，设计模式只是软件开发领域内多年来的经验总结，任何一个或简单或复杂的设计模式都会遵循上述的七大设计原则，只要大家真正理解了七大设计原则，设计模式对我们来说应该就不再是一件难事。但是，使用设计模式也不是要求我们循规蹈矩，只要我们的代码模型设计遵循了上述的七大原则，我们会发现原来我们的设计中就已经使用了某种设计模式。

五、参考资料

软件设计模式 – 百度百科
快速理解 – 设计模式六大原则
Software design pattern
《设计模式之禅》，秦小波，机械工业出版社
《领域驱动设计 – 软件核心复杂性应对之道》，Eric Evans，人民邮电出版社。
领域驱动设计在互联网业务开发中的实践

六、作者简介

吴亮亮，2017 年加入美团外卖，美团外卖营销后台团队开发工程师。

美团命名服务的挑战与演进

作者：舒超　张翔

本文根据美团基础架构部技术专家舒超在 2019 ArchSummit（全球架构师峰会）上的演讲内容整理而成。

命名服务主要解决微服务拆分后带来的服务发现、路由隔离等需求，是服务治理的基石。美团命名服务（以下简称 MNS）作为服务治理体系 OCTO 的核心模块，目前承载美团上万项服务，日均调用达到万亿级别。为了更好地支撑美团各项飞速发展的业务，MNS 开始从 1.0 向 2.0 演进。本文将围绕本次演进的初衷、实现方案以及落地的效果等方面进行展开，同时本文还介绍了命名服务作为一个技术中台组件，对业务的重要价值以及推动业务升级的一些成果。希望本文对大家能够有所启发。

一、MNS 1.0 简介

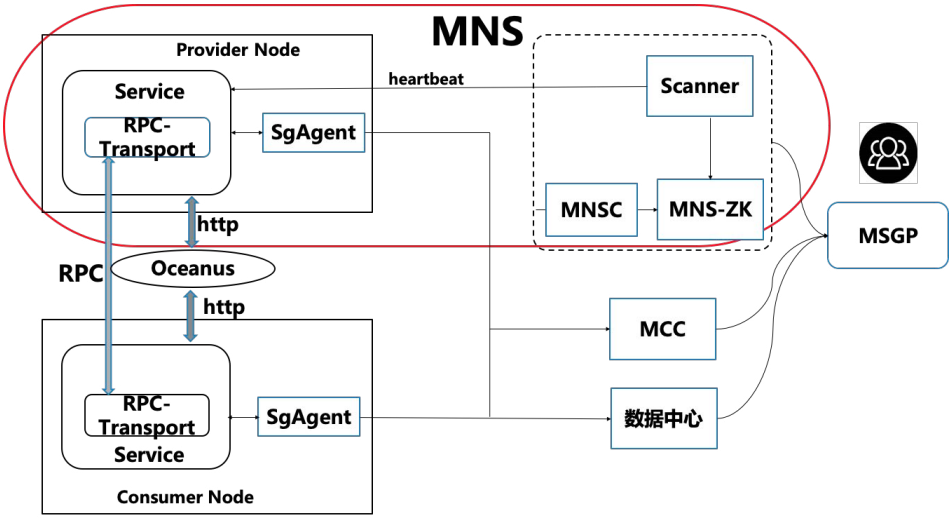


图 1　MNS 1.0 架构

从架构上看，MNS 1.0 主要分为三层：首先是嵌入业务内部的 SDK，用作业务自定义调用；然后是驻守在每个机器上的 SgAgent，以代理的方式将一些易变的、消耗性能的计算逻辑与业务进程分离开来，从而降低 SDK 对业务的侵入，减少策略变动对业务的干扰；远端是集中式的组件，包括健康检查模块 Scanner，鉴权缓存模块 MNSC，以及基于 ZooKeeper（以下简称 ZK）打造的一致性组件 MNS-ZK，作为通知和存储模块。在层级之间设立多级缓存，利用“边缘计算”思想拆分逻辑，简化数据，尽量将路由分配等工作均摊到端上，从而降低中心组件负载。更多详情大家可参考《美团大规模微服务通信框架及治理体系 OCTO 核心组件开源》一文中的 OCTO-NS 部分。

在体量方面，MNS 1.0 已经接入了美团所有的在线应用，涉及上万项服务、数十万个节点，并覆盖了美团所有的业务线，日均调用达万亿级别，目前我们已将其开源。

总的来讲，作为公司级核心服务治理组件，MNS 1.0 在架构上带有明显的 CP 属性，在保持架构简洁、流程清晰的同时，还支持了业务大量迭代的需求，较好地帮助公司业务实现了服务化、标准化的目标。

二、MNS 1.0 遇到的问题和挑战

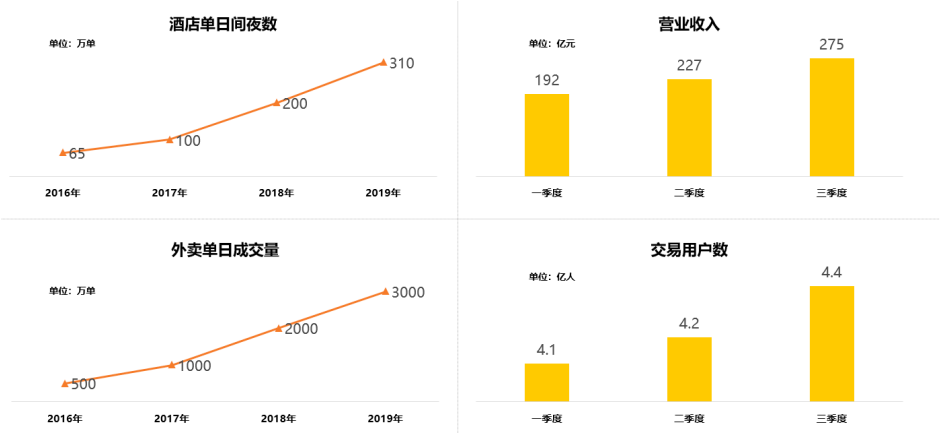


图 2　近三年美团业务增长数据

但是随着美团业务的快速增长，公司的服务数、节点数、服务信息量、服务变动频次等维度都在快速增长，有些服务甚至呈现出跨数量级的增长。在这样的情况下，命名服务也面临着一些新的问题和挑战：

- **可用性**：公司业务持续高速增长，很多都需要进行跨地域的部署。在 MNS 1.0 架构下，地域之间的专线如果断连，会发生一个地域命名服务整体不可用的情况；其次，强一致组件存在单点问题，Leader 出现故障，整个集群中断服务；另外，在多客户端和大数据传输的命名服务场景下，CP 系统恢复困难，RTO 达到小时级别。MNS 1.0 曾经出现过后端集中式组件单机连接超过十万且活跃链接数过半的情况，出现问题之后，现场恢复的负荷可想而知，这些都给命名系统的可用性带来很大的风险。
- **扩展性**：相对于需要支持的业务数量，MNS 1.0 整体平行扩展能力不足。MNS-ZK 的单集群规模上限不超过 300（实际只能达到 250 左右，这与 ZooKeeper 内部的 myid 等机制有关），否则同步性能会急剧恶化；其次，集群写入不可扩展，参与写入节点越多性能越差；另外，服务信息快照在固定的时间片内持续增长，增加 IO 压力的同时也延长了迁移的同步时间。而扩展性上的短板，导致 MNS 1.0 难以应对突发的流量洪峰，易出现“雪崩”。
- **性能**：写入操作受 CP 属性限制，串行性能较低。MNS-ZK 整体到 7000 左右的写量就已接近上限。其次，在热点服务场景下，数据分发较慢，这跟数据粒度较粗也有一定关系。而数据粒度粗、量大，也会在组件间传输消息时，导致临时对象频繁生成，引起 GC。另外，MNS 1.0 的后端集群负载还存在均衡性问题，一是因为原架构中缺乏集中管控服务，无法进行动态的集群与数据拆分伸缩，二是因为强一致属性下，集群节点间基于 Session 的迁移现场较重，很容易因一个节点挂掉而引起连锁反应。

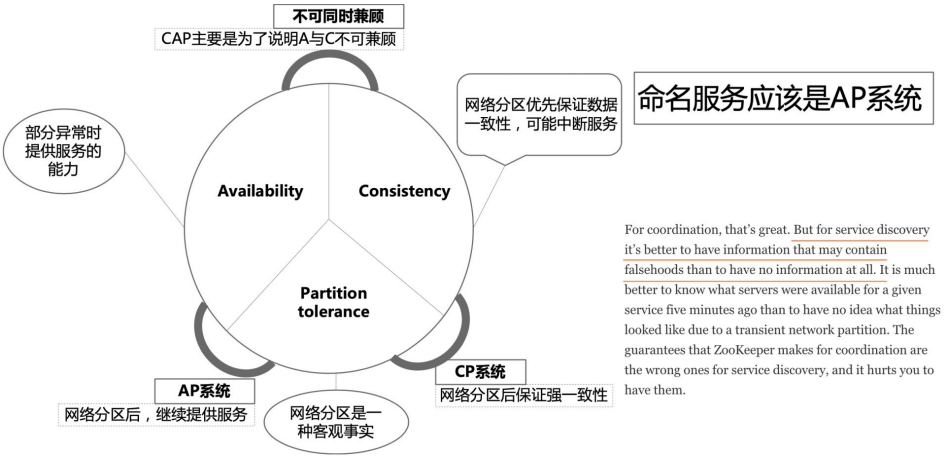


图 3　命名服务应该是 AP 系统

从可用性、扩展性、性能等三个方面，MNS 1.0 暴露出很多的问题，究其根源，原来的命名服务作为一个 CP 系统，为获得“数据一致性”而牺牲了部分情况下的可用性。其实，对于命名服务而言，一致性并没有这么重要，最多是调用方在一定时间内调多了或调漏了服务方而已，这个后果在不可靠网络的大前提下，即使命名服务没有出现问题，也可能经常会出现。借助端的自我检查调整机制，其影响可以说微乎其微。而另一方面，如果是一个机房或一个地域的调用方和服务方没有问题，但是因为这个区域和命名服务主集群断开了链接，从而导致本域内不能互调，这个就显得不太合理了。

所以总结一下，我们认为，**命名服务的本质是建立和增强服务本身的连通性，而不是主动去破坏连通性**。命名服务本质上应该是一个 AP 系统。

其实，业界对命名服务 AP/CP 模式都有相应实现和应用，很多企业使用 CP 模式，原因可能有以下几点：

- 架构行为简单：CP 系统在出现分区时行为比较简单，冷冻处理，粗暴但相对不容易出错。
- 启动门槛低：一些成熟的开源一致性组件，比如 ZK、etcd 都是 CP 系统，能够开箱即用，在数据规模可控的情况下，基本能够满足企业的需求。

另外，我们了解到，一些公司使用特殊的方式弱化了这个问题，比如将 CP 系统进一步拆分到更小的域中（比如一个 IDC），缩小分区的粒度，而全局有多个 CP 系统自治。当然，这个可能跟调用方服务方的跨度限制或者说调用配套部署有关系，但也有可能带来更复杂的问题（比如 CP 系统之间的数据同步），这里就不做详细的讨论了。

除去 MNS 1.0 本身的架构缺陷，我们还需要面临另一个问题，当初在项目启动时，云原生尚处于起步阶段，而如今，一些基于云原生理念兴起的网络基础设施，尤其是 Service Mesh 在美团快速发展，也需要 MNS 进行改造去适配新的流量通道和管控组件，这也是此次 MNS 2.0 演进的目标之一。

综上，我们以 AP 化、Mesh 化为主要目标，正式开始了从 MNS 1.0 向 MNS 2.0 的演进。

三、MNS 2.0

1. 整体架构

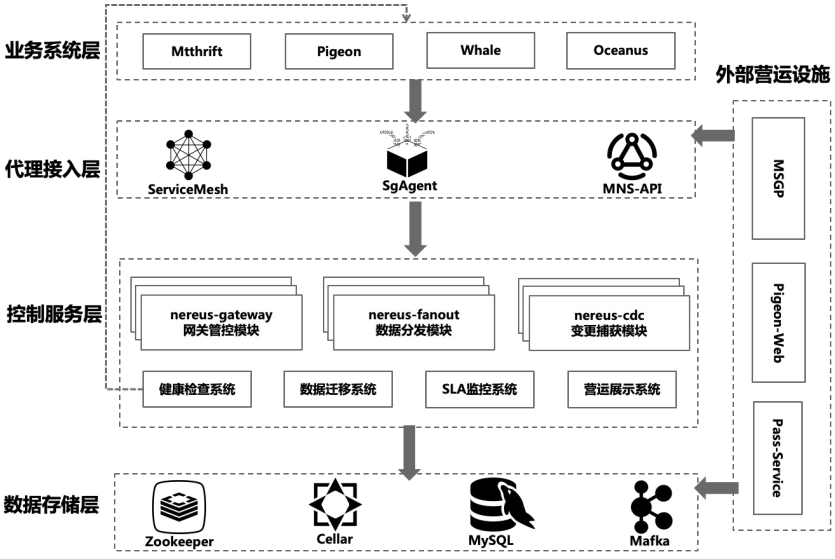


图 4　MNS 2.0 整体架构

MNS 2.0 的整体架构自上而下主要分为四层：

- 业务系统层：这一层与 MNS 1.0 架构类似，依然是嵌入到业务系统中的 SDK 或框架，但是不再感知服务列表和路由计算，从而变得更加的轻薄。
- 代理接入层：这一层主要变化是加入了 Service Mesh 的 Sidecar 和 MNS-API。前者将命名服务的部分链路接入 Mesh；后者为 MNS 增加了更丰富的 HTTP 调用，帮助一些没有使用 SDK 或框架的业务快速接入到命名服务。整个代理接入层的改造使得 MNS 对接入业务更加亲和。
- 控制服务层：新增注册中心控制服务，这也是 MNS 2.0 的核心。主要分为以下三个模块：
  - 网关管控模块：提供集中式的鉴权、限流 / 熔断、统计等 SOA 服务化的管控能力，同时避免海量代理组件直连存储层。
  - 数据分发模块：数据的通道，包括注册数据的上传、订阅数据的下发，可进行精细化数据拆分和平行扩展来适配热点服务。
  - 变更捕获模块：服务注册发现的审计，包括对第三方系统进行事件通知和回调，支持多元化的服务数据营运需求。
  - 另外，控制服务层还包括全链路 SLA 监控等新的子模块，以及健康检查 Scanner 这样的传统 MNS 组件。
- **数据存储层**：我们进一步丰富了命名服务的存储和分发介质，提高了数据层的整体性能。主力存储使用 K/V 存储系统（美团 Cellar 系统）替代 MNS-ZK，有效地提高了数据的吞吐能力，支持网络分区后的数据读写以及宕机灾备，同时保留 ZK 做一些轻量级的 Notify 功能。新增的关系型数据库和消息队列（美团 Mafka 系统），配合控制层的变更捕获模块，提供更方便的数据挖掘结构和外部扇出。

旁路于上面 4 层的是外部营运设施，主要是业务端的可视化 Portal，用户可以在上面对自身服务进行监控和操作，美团的基础研发部门也可以在上面进行一些集中式的管控。

综上所述，MNS 2.0 整体架构在兼容 1.0 的前提下，重点变化是：新增了控制服务层并对底层存储介质进行拆解。下面，我们来看一下服务注册发现功能在 MNS 2.0 中的实现流程：

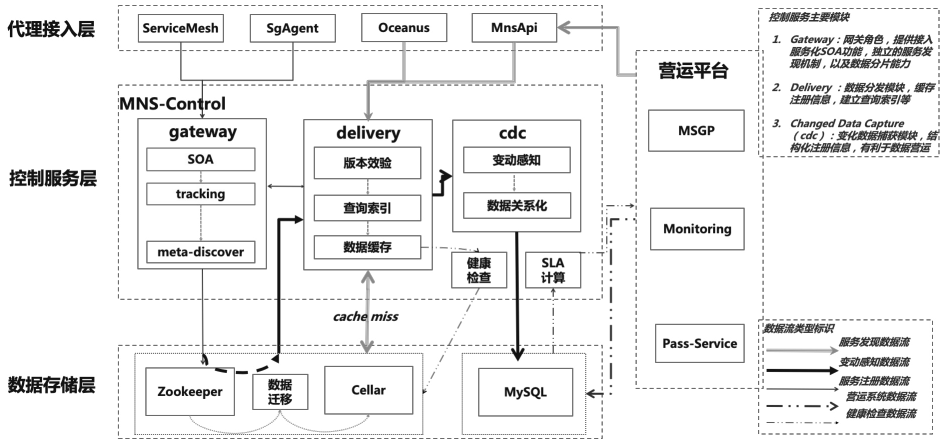


图 5　MNS 2.0 中的服务注册发现流程

1. **服务注册**：代理接入层透传业务注册请求，经过控制服务层的管控模块（Gateway）一系列 SOA 和审计流程后，写入注册数据到存储层。
2. **数据感知**：控制服务监听数据变动，服务注册写入新信息后，分发模块（Delivery）更新内存中的缓存，数据流经过捕获模块（CDC）将注册信息关系化后存储到 DB。
3. **服务发现**：代理层请求经过控制服务的管控模块（Gateway）效验后，从分发模块（Delivery）的缓存中批量获取服务端注册信息。Cache Miss 场景时从存储层读取数据。
4. **外部交互层**：外部系统当下通过代理层接入整个 MNS 体系，避免直连存储带来的各类风险问题。未来，营运平台可直接使用准实时 DB 数据，以 OLAP 方式进行关系化数据的分析。

接下来，我们一起来看下 MNS2.0 的主要演进成果。

2. 演进成果

2.1 流量洪峰 & 平行扩展

流量洪峰对于不同领域而言有不同的时段。对于 O2O 领域比如美团来说，就是每天中午的外卖高峰期，然后每天晚上也会有酒旅入住的高峰等等。当然，基于其它的业务场景，也会有不同的高峰来源。比如通过“借势营销”等运营手段，带来的高峰量可能会远超预期，进而对服务造成巨大的压力。



图 6　流量洪峰

MNS 1.0 受制于 MNS-ZK 集群数量上限和强一致性的要求，无法做到快速、平行扩展。MNS 2.0 的数据存储层重心是保证数据安全读写和分布式协调，在扩展能力层面不应该对其有太多的要求。MNS 2.0 的平行扩展能力主要体现在控制服务层，其具体功能包括以下两个方面：

集群分片：控制服务提供全量注册数据的分片能力，解决命名服务单独进行大集群部署时不能进行业务线隔离的风险。MNS-Control 网关模块分为 Master 和 Shard 等两类角色，协作提供大集群分片能力。

- **Master**：维护服务注册信息与各个分片集群（Shard）的映射关系，向代理层组件提供该类 Meta 数据。Master 接收各个 Shard 集群事件，新增、清理 Shard 中维护的注册信息。

- **Shard**：数据分片自治向代理组件提供服务注册 / 发现功能，实现按业务线隔离命名服务，同时按照 Master 发出的指令，调整维护的注册信息内容。
- **Services**：业务服务通过代理组件接入 MNS，代理组件启动时通过与 Master 的交互，获知归属的 Shard 集群信息以及 Fallback 处理措施。此后 Agent 只与业务线 Shard 进行命名数据交互，直到 Shard 集群不可用，重新执行“自发现”流程。
- **Fallback 措施**：设置一个提供全量服务信息的默认 Shard 集群，当业务线 Shard 异常时。一方面，Agent 重启“自发现”流程，同时将命名请求重定向到默认的 Shard 集群，直到业务线 Shard 恢复后，流量再切回。

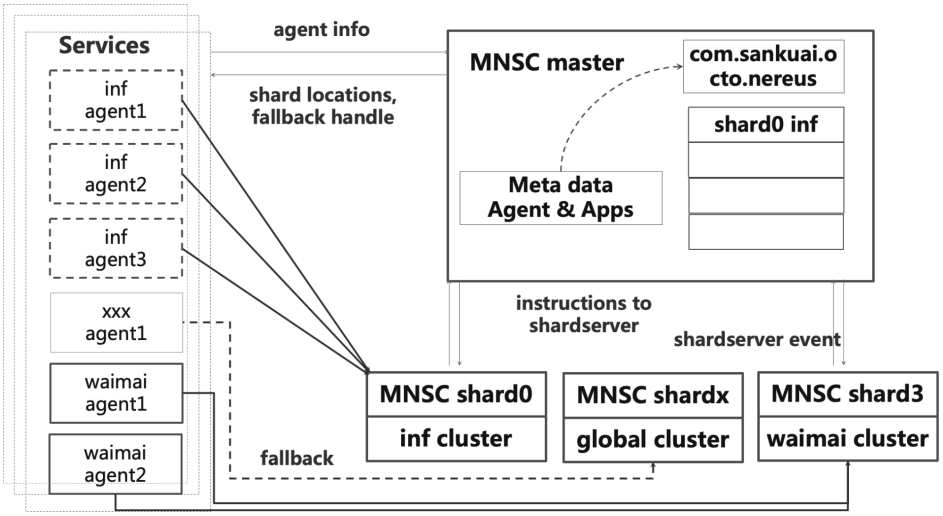


图 7　控制服务数据分片示意图

网络分区可用：MNS 1.0 阶段，网络分区对可用性的影响巨大，分区后 MNS-ZK 直接拒绝服务。新架构将存储迁移到 KV 系统后，在网络专线抖动等极端情况下，各区域依然能正常提供数据读取功能。同时，我们与公司存储团队共建 C++ SDK 的地域就近读写功能。一方面，提高跨域服务注册发现的性能，另一方面，实现了网络分区后，各区域内命名服务可读、可写的目标，提高了系统的可用性，整个命名服务对网络的敏感度降低。

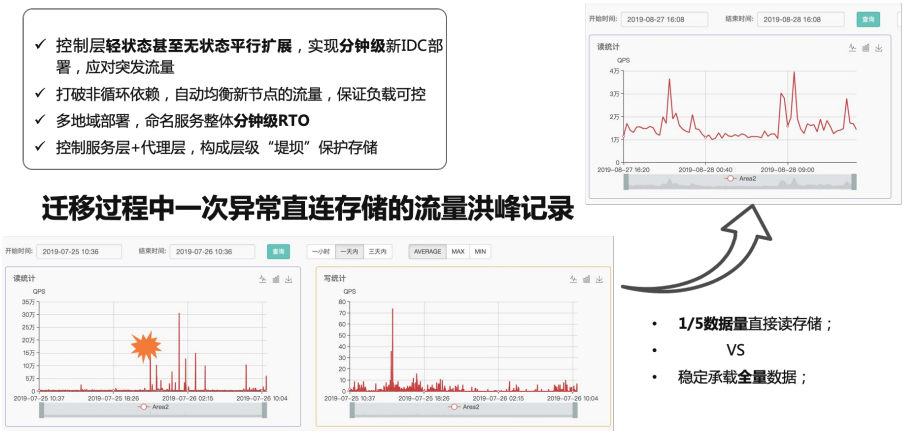
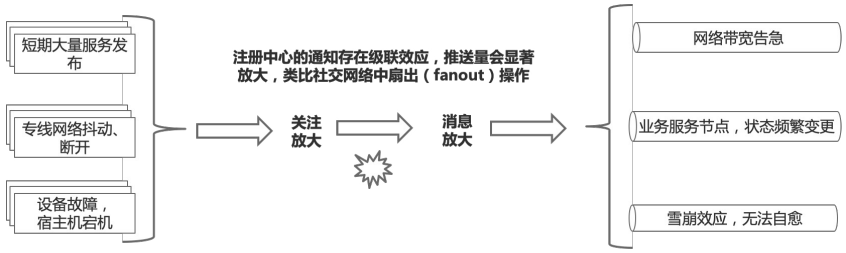


图 8　命名服务的平行扩展

目前，控制服务层在平行扩缩容时间和集群原地恢复时间等方面，都达到了分钟级，集群也没有节点上限，从而能够有效应对突发的流量，防止服务出现“雪崩”。

2.2 推送风暴 & 性能提升

另一个典型的场景是推送“风暴”，在服务集中发布、出现网络抖动等情况下，会导致命名服务出现”一横一纵”两种类型的放大效应：横向是“关注放大”，类似于社交网络中某大 V 消息需要分发给众多的粉丝，服务越核心，放大的效果越明显。纵向是“级联放大”，命名服务的上下游会逐级进行拷贝发送，甚至一级上下游会针对一个消息有多次的交互（Notify+Pull）。



$$\text{推送规模} = \boldsymbol{N}\text{服务变化} \times \mathbf{1}\text{次放大} \times \mathbf{2}\text{次放大}$$

$$\mathbf{1000}\boldsymbol{N} = \boldsymbol{N} \times \mathbf{100} \times \mathbf{10}$$

图 9　命名服务领域的消息放大现象

“关注放大”和“级联放大”本身都是无法避免的，这是由系统属性决定，而我们能做的就是从两方面去平滑其带来的影响：

正面提升核心模块性能，增强吞吐、降低延迟

1. 结构化聚合注册信息：控制服务的数据分发模块，在内存中存储结构化的服务注册信息，提供批量数据的读取能力；降低多次网络 RTT 传输单个数据以及数据结构转化带来的高耗时。 2. 高并发的吞吐能力：控制服务通过无锁编程处理关键路径的竞争问题，借助应用侧协程机制，提供高并发、低延迟的数据分发功能。
2. 与存储团队共建，实现 KV 系统就近区域读写，提高命名服务的服务注册（数据写入）性能。

另外，包括前面提到的控制服务集群的平行扩展能力，其实也是整体性能提升的一种方式。

侧面疏通，区分冷热数据，降低推送的数据量，提高效能

自然界中普遍存在“80/20 法则”，命名服务也不例外。服务注册信息的结构体中元素，80% 的改动主要是针对 20% 的成员，比较典型的就是服务状态。因此，我们将单个整块的服务信息结构体，拆分为多个较小的结构体分离存储；当数据变动发生时，按需分发对应的新结构体，能够降低推送的数据量，有效减少网络带宽的占用，避免代理组件重复计算引起的 CPU 开销，数据结构变小后，内存开销也得到显著降低。

那是否需要做到完全的“按需更新”，仅推送数据结构中变动的元素呢？我们认为，完全的“按需更新”需要非常精细的架构设计，并会引入额外的计算存储开销。比如，我们需要将变动成员分开存储以实现细粒度 Watcher，或用专门服务识别变动元素然后进行推送。

同时，还要保证不同成员变动时，每次都要推送成功，否则就会出现不一致等问题。在组件计算逻辑所需的数据发生变化时，也会带来更多的改动。在命名服务这样的大

型分布式系统中，“复杂”、“易变”就意味着稳定性的下降和风险的上升。所以根据“80/20 法则”，我们进行尺度合理的冷热数据分离，这是在方案有效性和风险性上进行权衡后的结果。

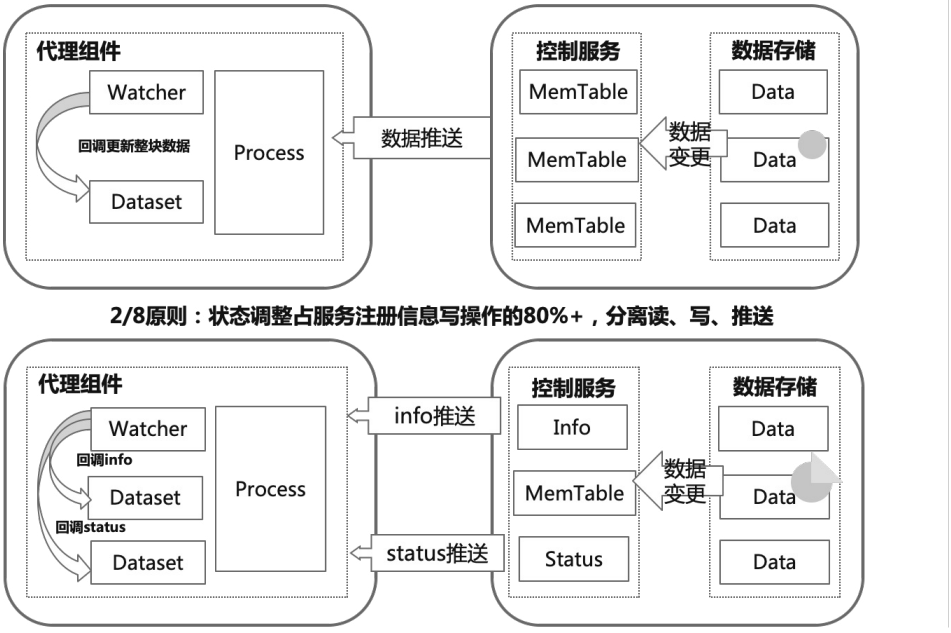


图 10　冷热数据分拆推送

经过改造，MNS 2.0 相比 MNS 1.0 的吞吐能力提升 8 倍以上，推送成功率从 96% 提升到 99%+，1K 大小服务列表服务发现的平均耗时，从 10s 降低到 1s，TP999 从 90s 下降到 10s，整体优化效果非常明显。

2.3 融入 Service Mesh

在 MNS 2.0 中，我们将代理服务 SgAgent 部分注册发现功能合并到了 Mesh 数据面，其流程如下图所示：

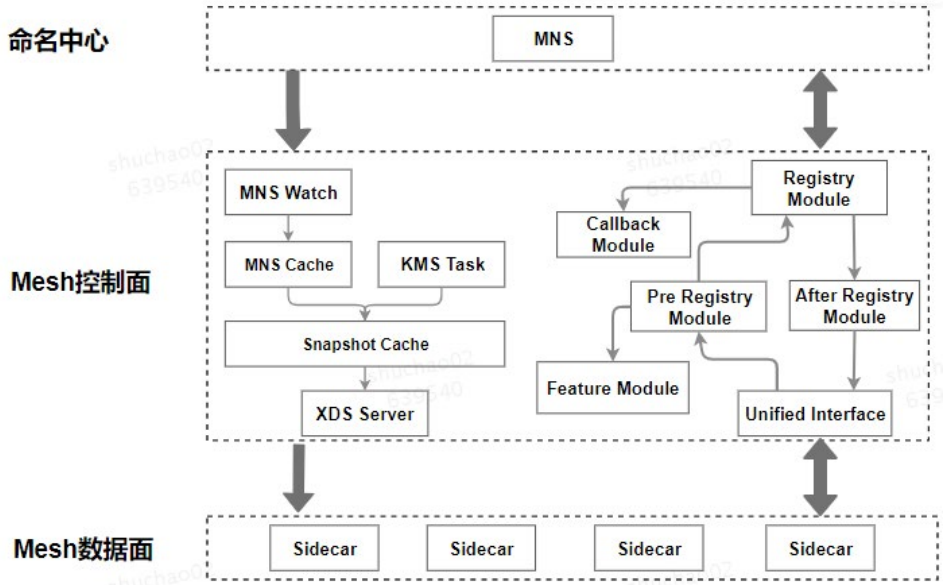


图 11　命名服务与 Service Mesh 的融合

关于美团服务治理功能与 Service Mesh 结合的技术细节，这部分的内容，我们今年会单独做一个专题来进行分享，感兴趣的同学可以关注“美团技术团队”微信公众号，敬请期待。

2.4 无损迁移

除了上面说到的 MNS 2.0 的这些重点演进成果之外，我们还想谈一下整个命名服务的迁移过程。像 MNS 这样涉及多个组件、部署在公司几十万个机器节点上、支撑数万业务系统的大规模分布式系统，如何进行平滑的数据迁移而不中断业务正常服务，甚至不让业务感知到，是 MNS 2.0 设计的一个重点。围绕业务服务无感知、具备快速回滚能力、新 / 老体系互备数据不丢失等要求，我们设计了如下图所示的迁移流程：

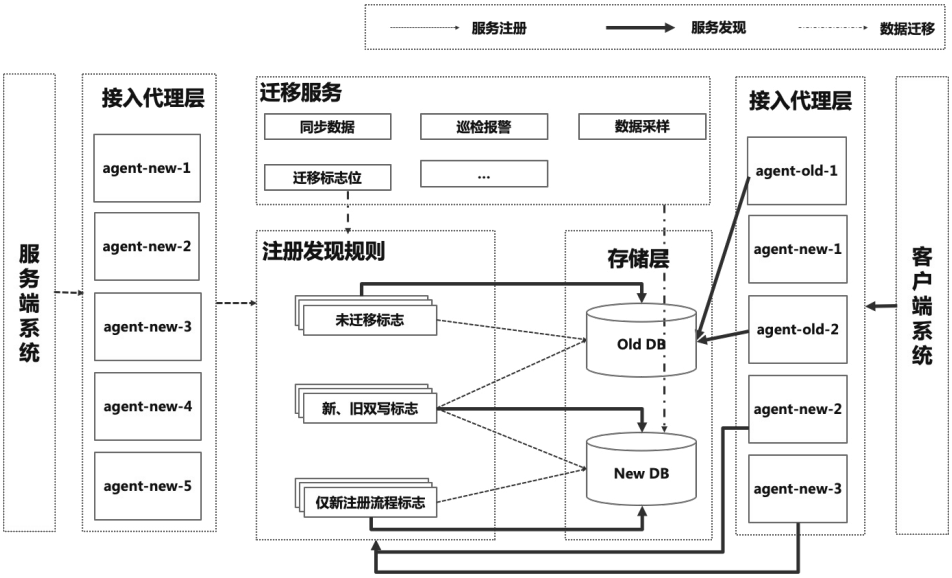


图 12　MNS 2.0 的无损迁移

MNS 2.0 整体以服务为粒度进行迁移操作，设置标志位说明服务所处的状态，由接入代理层组件识别该标志做出相应的处理。标志位包括：

1. 未迁移标志：服务注册 / 发现走 MNS 1.0 的流程，注册信息存储到重构前的 MNS-ZK 中。
2. 迁移中标志：服务注册并行走 MNS 1.0 和 MNS 2.0 流程，数据同时写入新旧两个地方，服务发现执行 MNS 2.0 流程。
3. 已迁移标志：服务注册 / 发现全部走 MNS 2.0 流程，注册信息仅存储到 MNS 2.0 的数据层。该阶段无法进行平滑的回滚，是项目长期验证后最终的迁移状态。

上述可以总结为：**聚焦单个服务，阶段性迁移服务发现流量，从而达到类似系统新功能发布时“灰度上线”的能力**。当然，这个策略还涉及一些细节在其中，比如，分开存储在双写时需要重点去保证异常情况下的最终一致性等等，鉴于篇幅原因，这里就不详细展开讨论了，而且业界针对这种情况都有一些成熟的做法。

另外，我们通过优化命名服务发布系统的发版形式，实现自动化的流量灰度策略，降低了人力成本，同时构建了自动化的迁移工具、巡检工具，高效地进行自动化的数据迁移工作，能够快速巡检迁移后的链路风险，保障新 MNS 2.0 的稳定上线。

2.5 演进总结

在美团命名服务这样的大型分布式系统优化过程中，具体到架构和功能设计层面，我们也做了不少的权衡和取舍，前面我们也提到，放弃了增量式的精准变动信息推送方式。在考虑性能的前提下，也没有使用数据多维度营运能力更好的 MySQL 作为主要存取介质等等。取舍的核心原则是：改造目标时强调业务收益，落地过程中减少业务感知。

目前，MNS 2.0 主要成果如下：

- 重构了 5 个已有的核心组件，研发了 2 个全新的系统，完成公司 PaaS 层数十个服务的功能的适配改造，目前已成功迁移全公司 80% 以上的服务，且在迁移过程中无重大事故。
- RTO 从数小时降到分钟级，RPO 为 0；全链路推送耗时 TP999 从 90s 降到了 10s，推送成功率从 96% 提升到 99% 以上，基本完成了百万级别服务节点治理能力的建设。
- 集群数据按照业务线等多维度进行拆分，建立了基于分级染色的 SLA 指标和定期巡检机制，同时根据公司实际场景增加了众多的服务风控审计功能，保障了业务安全。
- 云原生方面，支持了 Service Mesh，注册发现流程融入了 Mesh 底层基础设施。

四、命名服务对业务的赋能

命名服务本身作为基础的技术中台设施，在坚持“以客户为中心”，升级自身架构的同时，也从如下几个方面对美团的多个业务进行赋能。

4.1 单元化 & 泳道

单元化（SET 化）是业界比较流行的容灾扩展方案，关于美团单元化的详细内容，可参考 OCTO 团队在本次 ArchSummit 中的另一个专题分享《SET 化技术与美团点评实践解密》。这里主要是从命名服务对单元化支撑的角度去解答这个问题。

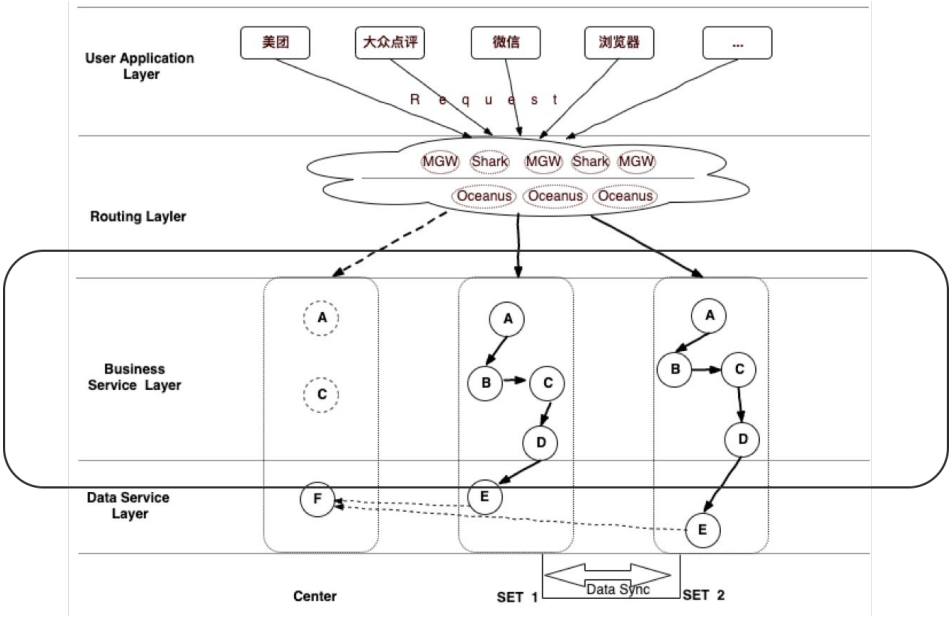


图 13　命名服务对单元化的支持

如上图所示，业务多种来源的外网流量在通过网关进入内网后，会借助命名服务提供的能力（SDK/Agent），并按照业务自定义的核心数据维度和机器属性，给流量打上单元化标签，然后路由到标签匹配的下一跳，从而实现了单元间流量隔离。一个单元内部，从服务节点到各种存储组件，都依赖于命名服务提供的单元识别和路由能力来完成隔离，所以命名服务在单元化中主要起底层支撑的作用。目前单元化在美团的重点业务，比如外卖、配送已经建设的比较完备，通过一定的单元冗余度，能在一个单元出现问题时，切换到另一个可用的镜像单元，显著提高了业务整体可用性。

接下来，我们再来看一下泳道场景。目前泳道在美团这边主要用于业务做完代码 UT 之后的线下集成测试阶段，同时结合容器，实现一个即插即用的上下游调用环境去验

证逻辑。命名服务在其中起到的作用与单元化类似，根据泳道发布平台对机器的配置，自动编排上下游的调用关系。如下图所示：

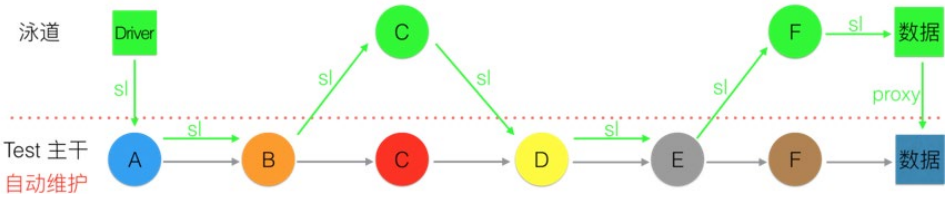


图 14　泳道流量示意图

当下一跳存在泳道节点时，测试流量进入泳道。反之，测试流量回流到主干。每个节点重复上述过程。

4.2 平滑发布 & 弹性伸缩

命名服务另外一个重要场景是服务的平滑发布，我们与发布平台配合，控制发布流量的自动摘除与恢复，实现服务发布操作的自动化与透明化。具体流程如下图所示：

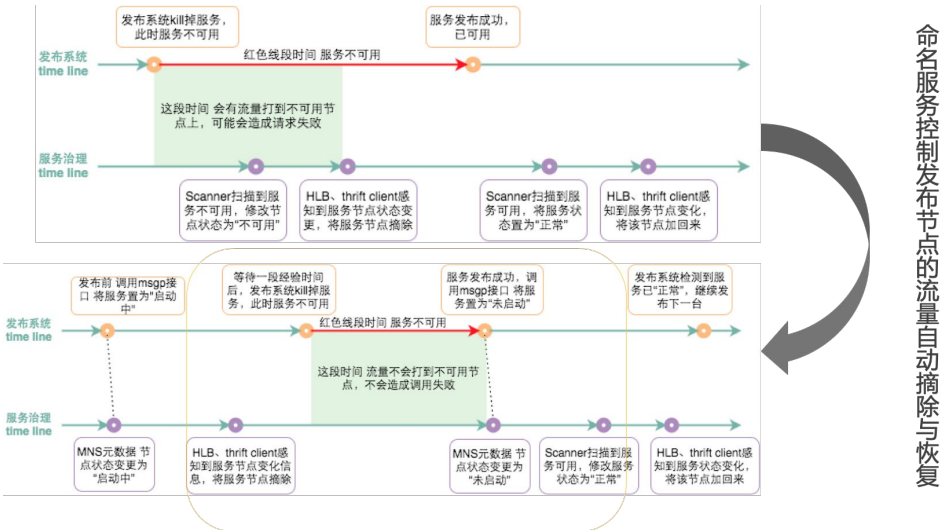


图 15　命名服务支持平滑发布

早期的发布流程，存在异常调用的风险，上线过程中存在一段服务实例不可用的“时间窗口”，此时流量再进入可能导致调用失败，然后会报错。为保证发布的稳定性，需要操作人员手动进行流量排空，流程上效率低、易出错，浪费业务团队的时间。针对这项业务痛点，命名服务向发布系统提供针对服务实例的流量管控能力，实现进程重启前自动摘除并清空来源请求，升级完毕后，提供自动流量恢复的平滑发布功能。智能地解决发版过程中的异常调用问题，提高公司的服务上线效率，降低了业务团队的运维成本。

弹性伸缩是容器一个很大的卖点。但是在伸缩过程中有很多问题需要考虑，比如扩缩容策略，包括调用亲和度配置，路由均衡等等。命名服务和容器化合作，提供业务的上下游调用关系、分组设置信息、容器下线流量无损摘除等服务，同时保障伸缩服务注册及时、高效。如下图所示：

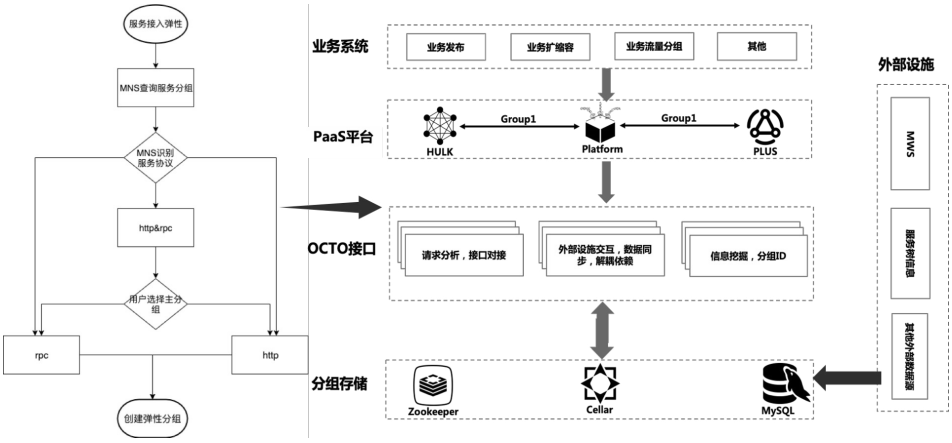


图 16　命名服务支持弹性伸缩

4.3 服务数据

MNS 服务数据对业务的赋能主要分为两个部分，一是将自身运行状态以业务可理解的 SLA 暴露出来，方便业务评估命名服务健康状况。对于命名服务来说，SLA 指标主要有推送成功率和推送耗时两种（其实，推送耗时也可以看做成功率的一个衡量维度，这里暂时不做太详细的区分）。

MNS 1.0 精准量化运行状况的困难在于，一方面 MNS 的发布 / 订阅机制重度依赖 ZK，受限于 ZK 的自身实现和链路上众多不同组件的异构性，及时获取完整链路的推送行为数据很困难。另一方面，由于节点数众多，全量采集服务发现数据，对公司的监控上报系统以及采集之后的计算也是很大的负担。

在 MNS 2.0 中，我们首先在架构上显著弱化了 ZK 的地位，它基本上只作为一致性通知组件。我们自研的 MNS-Control 可以充分 DIY 埋点场景，保证了主要推送行为抓取的可控性。其次，我们采用了分级采样计算，在全面梳理了公司现有的服务节点数比例后，将典型的服务列表数划分为几个档次，比如 1000 个节点的服务为一档，100 个又为一档，并创建相应的非业务哨兵服务，然后在不同机房选取适量的采样机器定期注册 + 拉取来评估整体的运行情况。这样既做到了上报量的精简，又兼顾了上报数据的有效性。详细操作流程，如下图所示：



图 17　MNS 2.0 SLA 采集

控制层周期性修改采样服务中的服务数据，触发服务发现的数据推送流程。参与指标统计的机器节点，本地代理进程获取到注册信息推送后，上报送达时间到运维平台并由其写入存储层。 控制层对数据库中的数据进行聚合计算，最后上报监控系统展示指标数据。此外，通过与监控团队合作，解决全量部署的 Agent 埋点监控问题。

命名服务存储的服务信息有很高的业务价值，从中可以知道服务的部署情况、发布频次以及上下游拓扑信息等等。所以“赋能”的第二个部分在于，借助这些信息，我们可以挖掘出业务服务部署运维上不合理的地方或风险点，不断推动优化，从来避免一

些不必要的损失。目前已经在推动的项目包括：

- 单进程多端口改造：业务使用这种方式去区分不同的调用端口，从而造成端口资源的浪费，而且调用下游还要感知部署信息，这种机器属性粒度细节的暴露在云原生时代是不太恰当的。我们协助业务将调用行为改成单端口多接口的形式，在协议内部去区分调用逻辑。
- 大服务列表的拆分：随着业务的发展，单个服务的节点数呈爆发式增长，甚至出现了一个服务有接近 7W 的节点数的情况，对发布、监控、服务治理等底层设施产生很大的压力。我们通过和业务的沟通，发现大列表产生的原因主要有两点：1. 单机性能不够，只能以实例抗；2. 服务内容不够聚焦。换句话说，因为服务还不够“微”，所以导致逻辑越来越庞大，有需求的调用方和自身实例相应增加，代码风险也在增大。因此，我们和业务一起梳理分析架构和调用，将核心共用模块与业务逻辑群分拆出来，减少冗余调用，最终使一些大服务节点数量从数万降低到数千，实现了数量级的下降。
- 上下游均衡部署：这个比较容易理解，结合调用端与服务端比例、服务方机器性能以及调用失败率等信息，可以作为服务业务在各机房间均衡调整机器数量的参考。另一方面，我们也在减少基本就近调用策略的粒度，目前只保留了“同 IDC”和“同城”两种，去掉了之前的“同中心策略”，减轻业务服务部署的心智负担。后期，随着机房数量的降低和同地域机房间延时的可控，同城路由可能是最终的方案。

在架构上，MNS 2.0 依赖 DB 和其它一些数仓介质，进行多种维度的数据上卷和下钻，并结合一些定制的风险策略逻辑去帮助业务发现和规避问题，目前这个事情也在进行中。

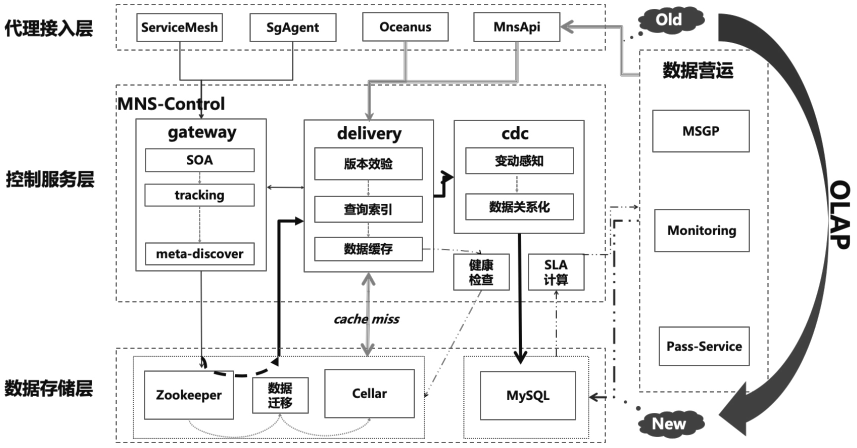


图 18　MNS 2.0 数据赋能

五、未来展望

未来，美团命名服务主要会朝两个方向发展：

- 进一步收集、挖掘服务数据的价值，打造服务数据平台，赋能业务升级。从单纯实现业务注册发现路由等需求，到通过数据反向推动业务架构流程改造升级，这也是美团核心价值观“以客户为中心”的一种体现。
- 深度结合 Service Mesh 等云原生基础设施的演进。云原生理念及相关设施架构的快速发展，必然会造成传统服务治理组件架构上流程的变动，深度拥抱云原生，并享受基础功能下沉带来的“红利”，是命名服务一个比较确定的方向。

作者简介

舒超，美团资深技术专家，OCTO 服务治理团队研发核心成员。
张翔，美团高级工程师，美团 OCTO 服务治理团队研发核心成员。

招聘信息

美团 OCTO 服务治理团队诚招 C++/Java 高级工程师、技术专家。我们致力于研发公司级、业界领先的基础架构组件，研发范围涵盖分布式框架、命名服务、Service Mesh 等技术领域。欢迎有兴趣的同学投送简历至 tech@meituan.com（邮件备注：美团 OCTO 团队）。

美团 MySQL 数据库巡检系统的设计与应用

作者：王琦

巡检工作是保障系统平稳有效运行必不可少的一个环节，目的是能及时发现系统中存在的隐患。我们生活中也随处可见各种巡检，比如电力巡检、消防检查等，正是这些巡检工作，我们才能在稳定的环境下进行工作、生活。巡检对于数据库或者其他 IT 系统来说也同样至关重要，特别是在降低风险、提高服务稳定性方面起到了非常关键作用。

本文介绍了美团 MySQL 数据库巡检系统的框架和巡检内容，希望能够帮助大家了解什么是数据库巡检，美团的巡检系统架构是如何设计的，以及巡检系统是如何保障 MySQL 服务稳定运行的。

一、背景

为了保障数据库的稳定运行，以下核心功能组件必不可少：

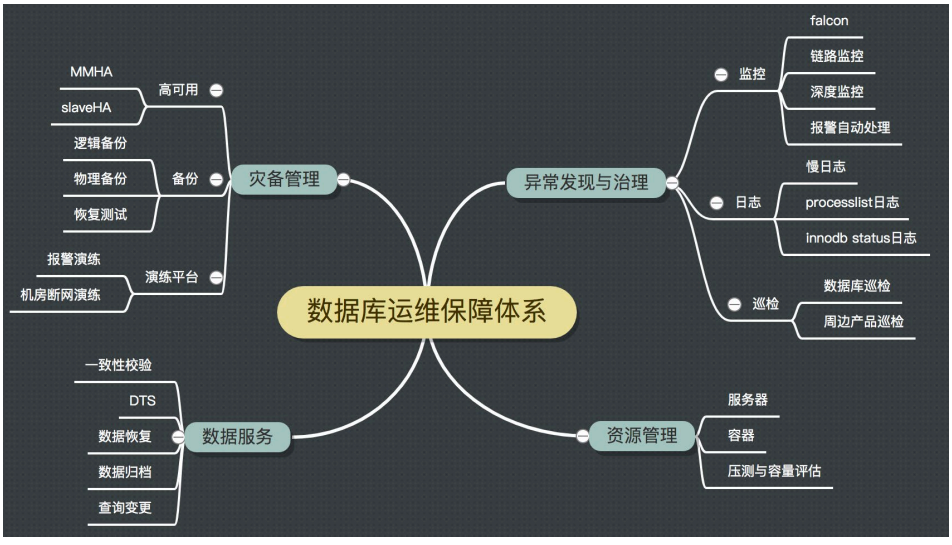


图 1　数据库运维保障核心功能组件

其中，数据库巡检作为运维保障体系最重要的环节之一，能够帮助我们发现数据库存在的隐患，提前治理，做到防患于未然。对于大规模集群而言，灵活健壮的自动化巡检能力，至关重要。

任何系统都会经历一个原始的阶段，最早的巡检是由中控机 + 定时巡检脚本 + 前端展示构成的。但是，随着时间的推移，老巡检方案逐渐暴露出了一些问题：

- 巡检定时任务执行依赖中控机，存在单点问题；
- 巡检结果分散在不同的库表，无法进行统计；
- 巡检脚本没有统一开发标准，不能保证执行的成功率；
- 每个巡检项都需要单独写接口取数据，并修改前端用于巡检结果展示，比较繁琐；
- 巡检发现的隐患需要 DBA 主动打开前端查看，再进行处理，影响整体隐患的治理速度；
- ……

所以我们需要一个灵活、稳定的巡检系统来帮助我们解决这些痛点，保障数据库的稳定。

二、设计原则

巡检系统的设计原则，我们从以下三个方面进行考虑：

稳定：巡检作为保证数据库稳定的工具，它自身的稳定性也必须有所保证；

高效：以用户为中心，尽量化繁为简，降低用户的使用成本，让新同学也能迅速上手治理和管理隐患；提高新巡检部署效率，随着架构、版本、基础模块等运维环境不断变化，新的巡检需求层出不穷，更快的部署等于更早的保障；

可运营：用数据做基础，对巡检隐患进行运营，包括推进隐患治理，查看治理效率、趋势、薄弱点等。

三、系统架构

美团 MySQL 数据库巡检系统架构图设计如下。接下来，我们按照架构图从下到上的顺序来对巡检系统主要模块进行简单的介绍：

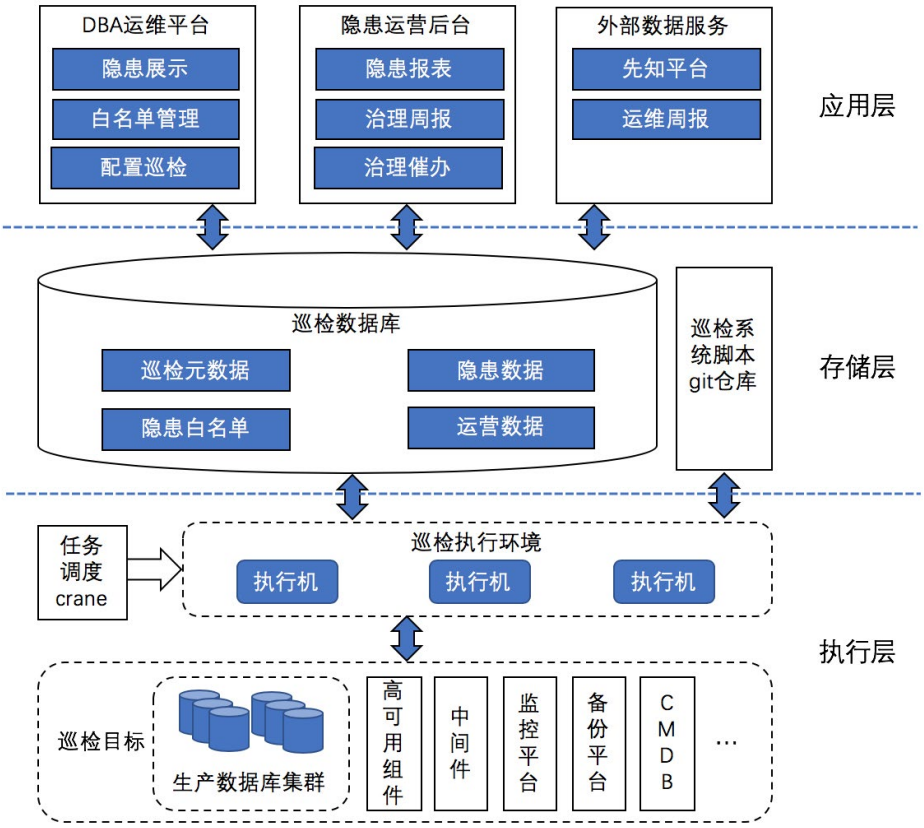


图 2　美团 MySQL 数据库巡检系统架构图

1. 执行层

巡检执行环境：由多台巡检执行机组成，巡检任务脚本会同时部署在所有执行机上。执行机会定时从巡检 Git 仓库拉取最新的脚本，脚本使用 Python Virtualenv + Git 进行管理，方便扩充新的执行机。

任务调度：巡检任务使用了美团基础架构部研发的分布式定时任务系统 Crane 进行

调度，解决传统定时任务单点问题。Crane 会随机指派某一台执行机执行任务，假如这台执行机出现故障，会指派其他执行机重新执行任务。一般一个巡检任务对应着一个巡检项，巡检任务会针对特定的巡检目标根据一定的规则来判断是否存在隐患。

巡检目标：除了对生产数据库进行巡检以外，还会对高可用组件、中间件等数据库周边产品进行巡检，尽可能覆盖所有会引发数据库故障的风险点。

2. 存储层

巡检数据库：主要用来保存巡检相关数据。为了规范和简化流程，我们将巡检发现的隐患保存到数据库中，提供了通用的入库函数，能够实现以下功能：

- 自动补齐隐患负责人、隐患发现时间等信息；
- 入库操作幂等；
- 支持半结构化的巡检结果入库，不同巡检的隐患结果包括不同的属性，比如巡检 A 的隐患有“中间件类型”，巡检 B 有“主库 CPU 核数”，以上不同结构的数据均可解析入库；
- 针对表粒度的隐患项，如果分库分表的表出现隐患，会自动合并成一个逻辑表隐患入库。

巡检脚本 Git 仓库：用来管理巡检脚本。为了方便 DBA 添加巡检，在系统建设过程中，我们增加了多个公共函数，用来降低开发新巡检的成本，也方便将老的巡检脚本迁移到新的体系中。

3. 应用层

集成到数据库运维平台：作为隐患明细展示、配置巡检展示、管理白名单等功能的入口。为了提高隐患治理效率。我们做了以下设计。

- 隐患明细展示页面会标注每个隐患出现的天数，便于追踪隐患出现原因。
- 配置新的巡检展示时必须要同时制定隐患解决方案，确保隐患治理有章可循，避免错误的治理方式导致“错上加错”。

隐患运营后台：这个模块主要目的是推进隐患的治理。

- 运营报表，帮助管理者从全局角度掌握隐患治理进展，报表包括隐患趋势、存量分布、增量分布、平均治理周期等核心内容，进而由上到下推动隐患治理；报表数据同样是通过 crane 定时任务计算获得。
- 隐患治理催办功能，用来督促 DBA 处理隐患。催办内容中会带有隐患具体内容、出现时长、处理方案等。催办形式包括大象消息、告警，具体选用哪种形式可根据巡检关键程度做相应配置。

外部数据服务：主要是将巡检隐患数据提供给美团内部其他平台或项目使用，让巡检数据发挥更大的价值。

- 对接先知平台（美团 SRE 团队开发的主要面向 RD 用户的风险发现和运营平台），平台接收各服务方上报的隐患数据，以 RD 视角从组织架构维度展示各服务的风险点，并跟进 RD 处理进度。巡检系统会把需要 RD 参与治理的隐患，比如大表、无唯一键表等，借助先知平台统一推送给 RD 进行治理。
- 运维周报，主要面向业务线 RD 负责人和业务线 DBA，以静态报告形式展示业务线数据库运行情况以及存在的问题，巡检隐患是报告内容之一。

四、巡检项目

巡检项目根据负责方分为 DBA 和 RD，DBA 主要负责处理数据库基础功能组件以及影响服务稳定性的隐患。RD 主要负责库表设计缺陷、数据库使用不规范等引起的业务故障或性能问题的隐患。也存在需要他们同时参与治理的巡检项，比如“磁盘可用空间预测”等。目前巡检项目共 64 个，类目分布情况如下图所示：

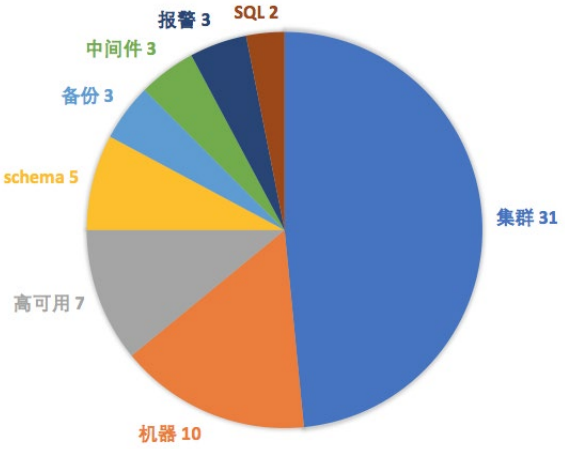


图 3　巡检项类目分布

集群：主要检查集群拓扑、核心参数等集群层面的隐患；**机器**：主要检查服务器硬件层面的隐患；**Schema/SQL**：检查表结构设计、数据库使用、SQL 质量等方面的隐患；**高可用 / 备份 / 中间件 / 报警**：主要检查相关核心功能组件是否存在隐患。

下面，我们通过列举几个巡检任务来对巡检项做简单的说明：

负责方	类目	巡检项	说明
DBA	备份	恢复测试结果检查	用来发现异常备份集。数据备份的重要性不言而喻，为了保证备份的有效性，备份系统会对每一个备份集做恢复测试，测试失败的情况会记录到巡检系统。
		Binlog无法衔接最近一次备份	通过最近一次备份和集群最早的Binlog日志情况，来判断备份和Binlog是否衔接，衔接不上会导致无法增加从库或基于时间点的数据恢复等。
	机器	磁盘可用空间预测	通过监控平台采集到的磁盘空间情况，预测磁盘空间是否会在近期使用耗尽。
	集群	集群GTID集异常	GTID即全局事务ID，在MySQL中每执行一个事务，就会分配一个GTID，并且从库会同步主库每一个GTID的操作，在没有延迟的情况下，集群所有实例的GTID集应该是一致的。由于MySQL主从复制原理，如果实例间GTID集不一致可能会导致主从切换失败，巡检通过比对每个实例的GTID集来判断是否存在异常。
		集群下多实例共宿主机	这种情况下，假如宿主机宕机，会导致集群下多台实例同时挂掉，严重影响数据库服务的稳定。虽然实例初始化时加入了相关策略主动避免这个问题，但不排除极特殊情况仍会造成这个隐患。
	高可用	集群高可用功能未开启	通常人工维护、或其他异常情况下，集群高可用功能会被关闭，巡检会检查CMDB中高可用开关来判断是否存在该隐患。
		同地域没有候选主库	这种情况下，主库宕机会提升其他地域实例作为主库，会导致写入延时增加，可能会影响业务。巡检会检查CMDB中集群实例的机房部署来判断是否存在该隐患。
RD	Schema	数值类型溢出	MySQL表的整型数据类型字段能够保存的数值是有上限的，超出最大值的数据无法写入表中。巡检系统根据一段时间采集到的整型类型字段值，判断该字段是否即将超过最大值。除了自增字段以外，还会对非自增的字段值进行预测。
		大表	表体积过大会出现性能问题，并且会增加维护成本。通过扫描所有数据库集群中表大小的情况，并结合监控平台采集到的表读写流量情况，找出问题大表。

五、成果

美团 MySQL 巡检系统已稳定运行近一年时间，基于新巡检体系上线的巡检项 49 个。通过巡检体系持续运行，在团队的共同努力下，我们共治理了 8000+ 核心隐患，近 3 个月隐患治理周期平均不超过 4 天，将隐患总数持续保持在极小的量级，有效地保障了数据库的稳定。

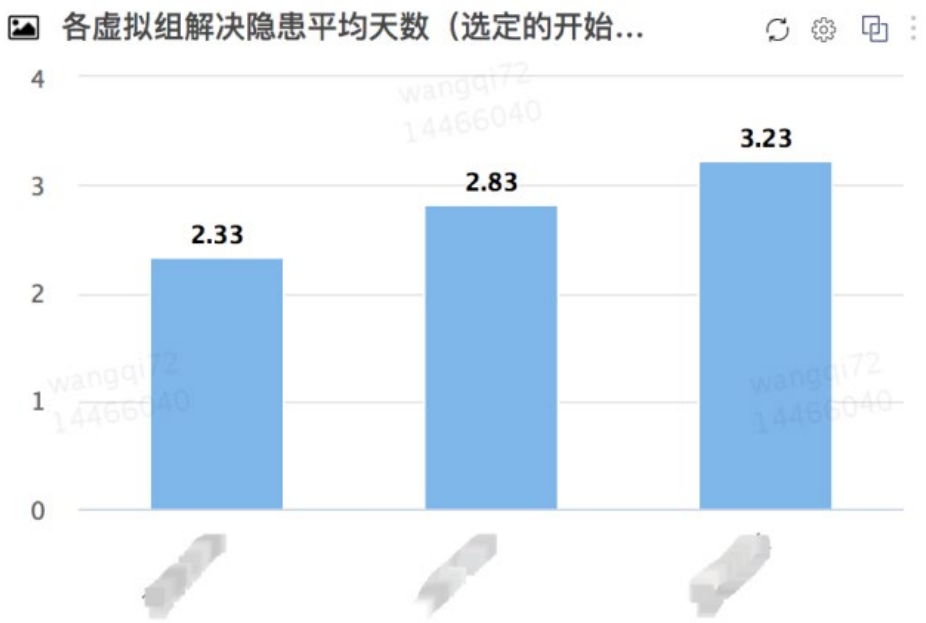


图 4　隐患运营 – 团队内各虚拟小组隐患平均治理周期

下面的隐患趋势图，展示了近一年中隐患的个数，数量突然增长是由于新的巡检项上线。从整体趋势上看，隐患存量有非常明显的下降。

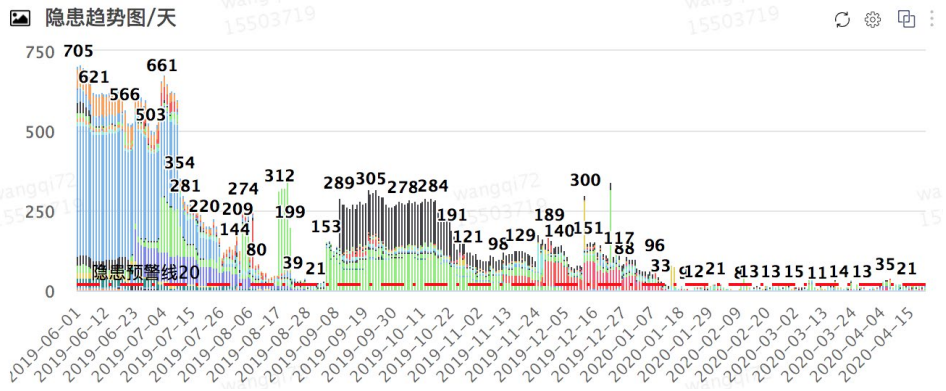


图 5　隐患运营 – 隐患总量趋势情况

除了推动内部隐患治理之外，我们还通过对接先知平台，积极推动 RD 治理隐患数量超过 5000 个。



图 6　对接先知 – 推动 RD 治理隐患

为了提升用户体验，我们在提升准确率方面也做了重点的投入，让每一个巡检在上线前都会经过严格的测试和校验。

对比其他先知接入方，DBA 上报隐患在总量、转化率、反馈率几个指标上都处于较高水平，可见我们上报的隐患风险也得到了 RD 的认可。

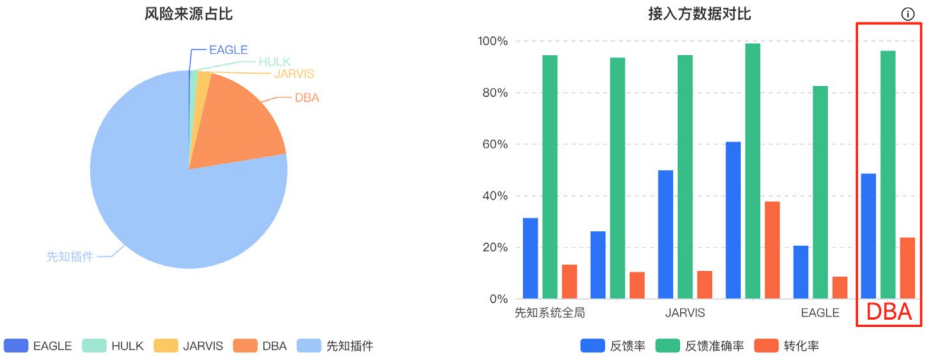


图 7　对接先知 – 各接入方上报隐患情况

指标说明：

- 反馈率 = 截止到当前时刻反馈过的风险事件数量 / 截止到当前时刻产生的风险事件总量 * 100%；
- 反馈准确率 = 截止到当前时刻反馈准确的风险事件数量 / 截止到当前时刻反馈过的风险事件总量 * 100%；
- 转化率 = 截止到当前时刻用户反馈准确且需要处理的风险事件数量 / 截止到当前时刻产生的风险事件总量 * 100%。

六、未来规划

除了继续完善补充巡检项以外，未来巡检系统还会在以下几个方向继续探索迭代：

- 提高自动化能力，完善 CI 和审计；
- 加强运营能力，进一步细化每个隐患的重要程度，辅助决策治理优先级；
- 隐患自动修复。

作者简介

王琦，基础架构部 DBA 组成员，2018 年加入美团，负责 MySQL 数据库运维 / 数据库巡检系统 / 监控 / 自动化运维周报 / 运维数据集市建设等工作。

Kubernetes 如何改变美团的云基础设施?

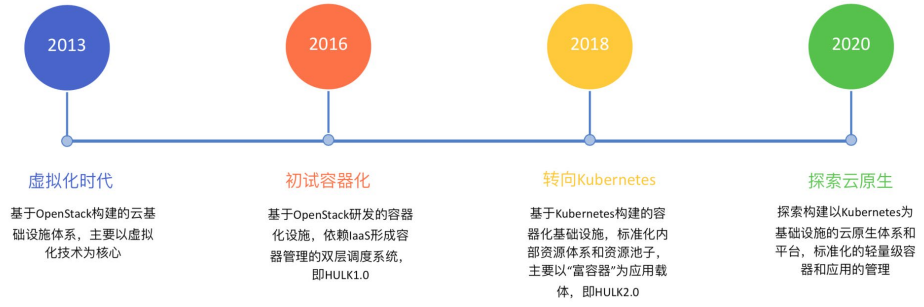
作者：王国梁

本文根据美团基础架构部王国梁在 KubeCon 2020 云原生开源峰会 Cloud Native + Open Source Virtual Summit China 2020 上的演讲内容整理而成。

一、背景与现状

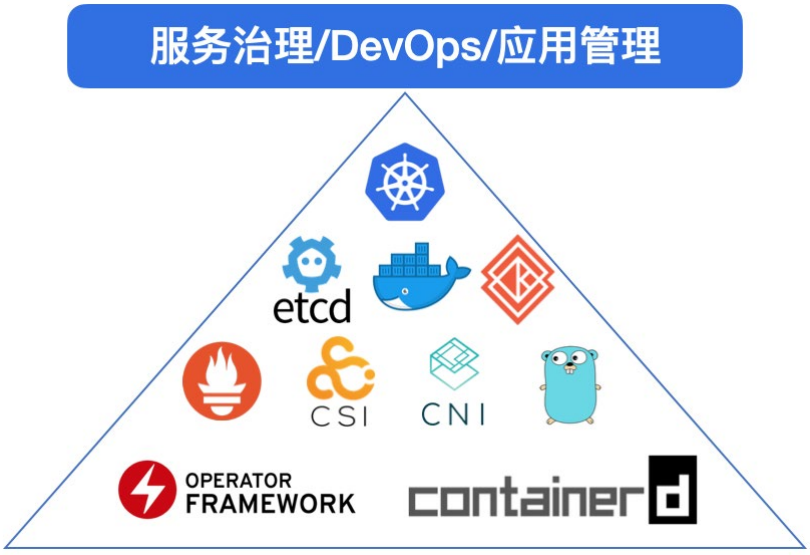
Kubernetes 是让容器应用进入大规模工业生产环境的开源系统，也是集群调度领域的事实标准，目前已被业界广泛接受并得到了大规模的应用。Kubernetes 已经成为美团云基础设施的管理引擎，它带来的不仅仅是高效的资源管理，同时也大幅降低了成本，而且为美团云原生架构的推进打下了坚实的基础，支持了 Serverless、云原生分布式数据库等一些平台完成容器化和云原生化的建设。

从 2013 年开始，美团就以虚拟化技术为核心构建了云基础设施平台；2016 年，开始探索容器技术并在内部进行落地，在原有 OpenStack 的资源管理能力之上构建了 Hulk1.0 容器平台；2018 年，美团开始打造以 Kubernetes 技术为基础的 Hulk2.0 平台；2019 年年底，我们基本完成了美团云基础设施的容器化改造；2020 年，我们坚信 Kubernetes 才是未来的云基础设施标准，又开始探索云原生架构落地和演进。

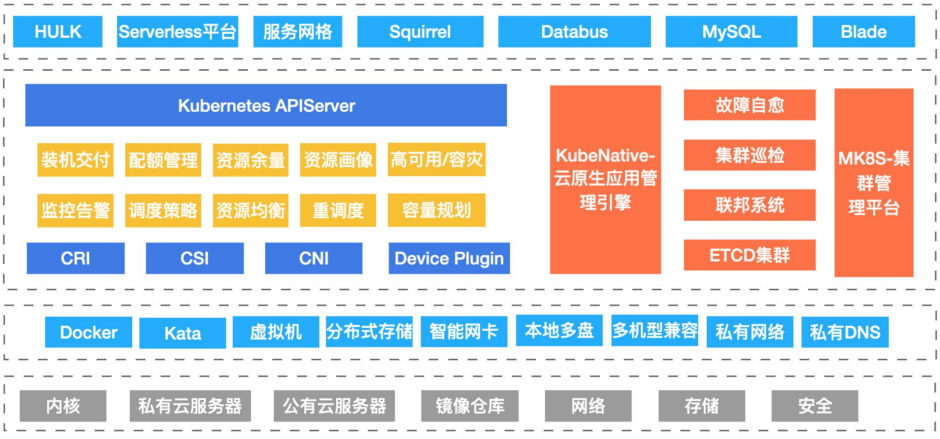


当前，我们构建了以 Kubernetes、Docker 等技术为代表的云基础设施，支持整个美团的服务和应用管理，容器化率达到 98% 以上，目前已有数十个大小 Kuberne-

tes 集群，数万的管理节点以及几十万的 Pod。不过出于容灾考虑，我们最大单集群设置为 5K 个节点。



下图是当前我们基于 Kubrnetes 引擎的调度系统架构，构建了以 Kubernetes 为核心的统一的资源管理系统，服务于各个 PaaS 平台和业务。除了直接支持 Hulk 容器化之外，也直接支持了 Serverless、Blade 等平台，实现了 PaaS 平台的容器化和云原生化。



二、OpenStack 到 Kubernetes 转变的障碍和收益

对于一个技术栈比较成熟的公司而言，整个基础设施的转变并不是一帆风顺的，在 OpenStack 云平台时期，我们面临的主要问题包括以下几个方面：

1. 架构复杂，运维和维护比较困难：OpenStack 的整个架构中计算资源的管理模块是非常庞大和复杂，问题排查和可靠性一直是很大的问题。
2. 环境不一致问题突出：环境不一致问题是容器镜像出现之前业界的通用问题，不利于业务的快速上线和稳定性。
3. 虚拟化本身资源占用多：虚拟化本身大概占用 10% 的宿主机资源消耗，在集群规模足够大的时候，这是一块非常大的资源浪费。
4. 资源交付和回收周期长，不易灵活调配：一方面是整个虚拟机创建流程冗长；另一方面各种初始化和配置资源准备耗时长且容易出错，所以就导致整个机器资源从申请到交付周期长，快速的资源调配是个难题。
5. 高低峰明显，资源浪费严重：随着移动互联网的高速发展，公司业务出现高低峰的时间越来越多，为了保障服务稳定不得不按照最高的资源需求来准备资源，这就导致低峰时资源空闲严重，进而造成浪费。

2.1 容器化的过程和障碍

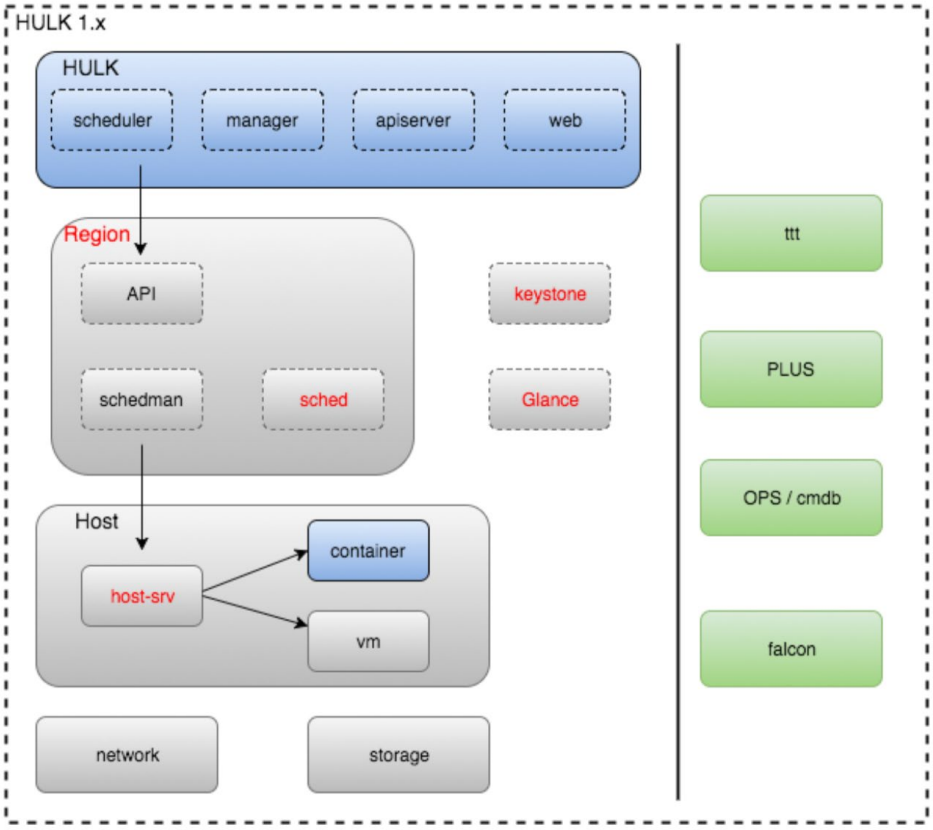
为了解决虚拟机存在的问题，美团开始探索更加轻量级的容器技术的落地，也就是 Hulk1.0 项目。不过基于当时的资源环境和架构，Hulk1.0 是以原有的 OpenStack 为基础资源管理层实现的容器平台，OpenStack 提供底层的宿主机的资源管理能力，解决了业务对弹性资源的需求，并且整个资源交付周期从分钟级别降低到了秒级。

但是，随着 Hulk1.0 的推广和落地，也暴露出一些新的问题：

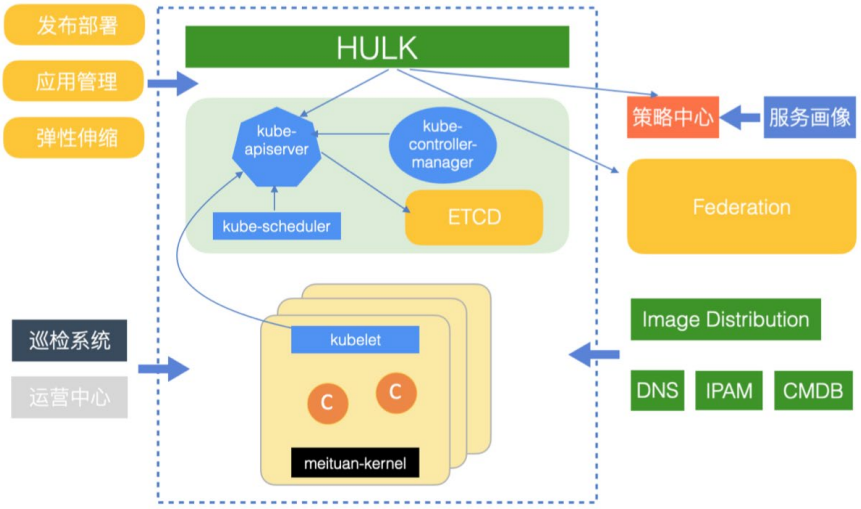
1. 稳定性差：因为复用了 OpenStack 的底层资源管理能力，整个扩容过程包括两层的资源调度，且数据同步流程复杂，机房的隔离性也比较差，经常出现一个机房出现问题，其他机房的扩缩容也受到影响。
2. 能力欠缺：由于涉及的系统多，并且是跨部门协作，故障节点的迁移和恢复能

力不易实现，资源类型也比较单一，整个故障排查和沟通效率低下。

3. 扩展性差：Hulk1.0 的控制层面对底层资源的管理能力受限，无法根据场景和需求快速扩展。
4. 性能：业务对于扩缩容和弹性资源的交付速度需求进一步提高，且容器技术的弱隔离性导致业务的服务受到的干扰增多，负面反馈增加。



上述的问题经过一段时间的优化和改善，始终不能彻底解决。在这种情况下，我们不得不重新思考整个容器平台的架构合理性，而此时 Kubernetes 已逐步被业界认可和应用，它清晰的架构和先进的设计思路让我们看到了希望。所以我们基于 Kubernetes 构建了新的容器平台，在新的平台中 Hulk 完全基于原生的 Kubernetes API，通过 Hulk API 来对接内部的发布部署系统，这样两层 API 将整个架构解耦开来，领域明确，应用管理和资源管理可以独立迭代，Kubernetes 强大的编排和资源管理能力凸显。



容器化的核心思路是让 Kubernetes 做好资源层面的管理，而通过上层的控制层解决对美团应用管理系统和运维系统的依赖问题，保持 Kubernetes 的原生兼容性，减少后续的维护成本，并完成了快速收敛资源管理的需求。同时，也减少了用户基于新平台的资源申请的学习成本，这点非常重要，也是后续我们能快速大规模迁移基础设施资源的“基础”。

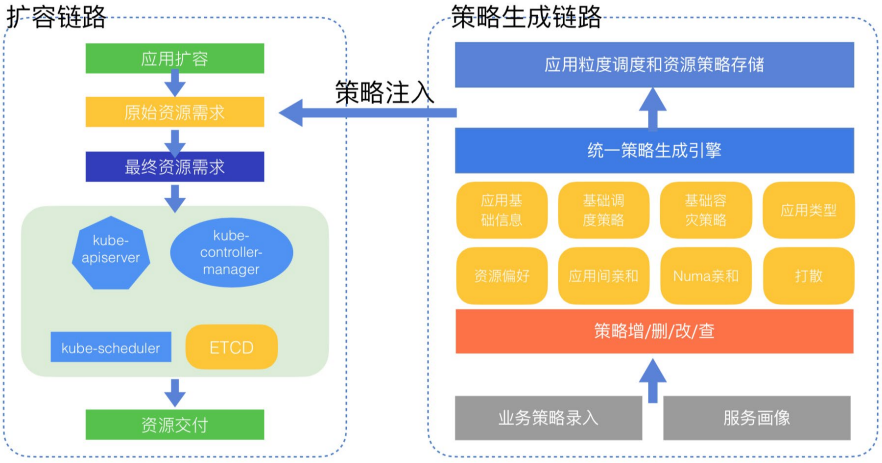
2.2 容器化过程的挑战和应对策略

2.2.1 复杂灵活、动态和可配置的调度策略

美团产品众多，业务线和应用特点五花八门，所以相应的，我们对于资源类型和调度策略的需求也是非常多。例如，有些业务需要特定的资源类型（SSD、高内存、高 IO 等等），有些业务需要特定的打散策略（例如机房、服务依赖等），所以如何很好地应对这些多样化的需求，就是一个很大的问题。

为了解决这些问题，我们为扩容链路增加了策略引擎，业务可以对自己的应用 APPKEY 自定义录入策略需求，同时基于大数据分析的服务画像，也会根据业务特点和公司的应用管理策略为业务策略推荐，最终这些策略会保存到策略中心。在扩容过程中，我们会自动为应用的实例打上对应的需求标签，并最终在 Kubenretes 中生

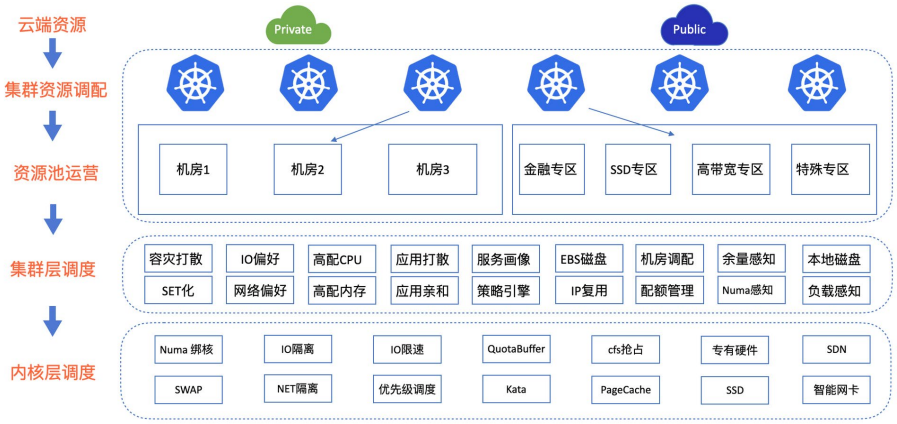
效，完成预期的资源交付。



2.2.2 精细化的资源调度和运营

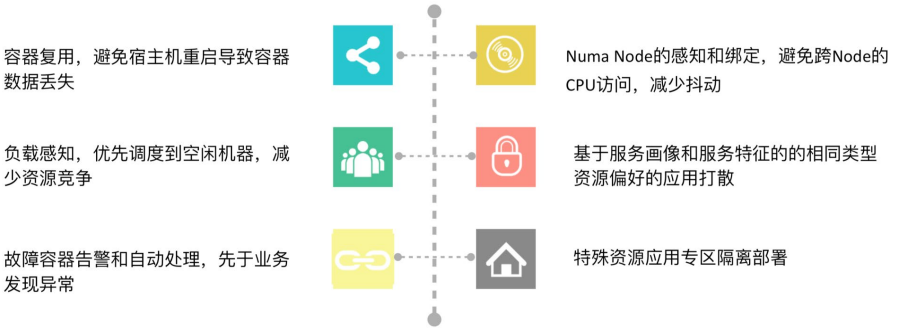
精细化的资源调度和运营，之所以做精细化运营主要是出于两点考虑：业务的资源需求场景复杂，以及资源不足的情况较多。

我们依托私有云和公有云资源，部署多个 Kubenretes 集群，这些集群有些是承载通用业务，有些是为特定应用专有的集群，在集群维度对云端资源进行调配，包括机房的划分、机型的区分等。在集群之下，我们又根据不同的业务需要，建设不同业务类型的专区，以便做到资源池的隔离来应对业务的需要。更细的维度，我们针对应用层面的资源需求、容灾需求以及稳定性等做集群层的资源调度，最后基于底层不同硬件以及软件，实现 CPU、MEM 和磁盘等更细粒度的资源隔离和调度。



2.2.3 应用稳定性的提升和治理

不管是 VM，还是最初的容器平台，在应用稳定性方面一直都存在问题。为此，我们需要在保障应用的 SLA 上做出更多的努力。



2.2.3.1 容器复用

在生产环境中，宿主机的发生重启是一种非常常见的场景，可能是主动重启也可能是被动，但用户角度来看，宿主机重启意味着用户的一些系统数据就可能丢失，代价还是比较高的。我们需要避免容器的迁移或重建，直接重启恢复。但我们都知道，在 Kubernetes 中，对于 Pod 中的容器的重启策略有以下几种：Always、OnFailure 和 Never，宿主机重启后容器会重新被创建。

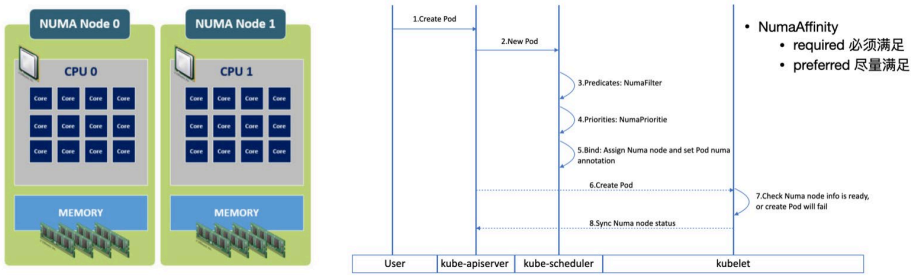


为了解决这个问题，我们为容器的重启策略类型增加了 Reuse 策略。流程如下：

1. kubelet 在 SyncPod 时，重启策略如果是 Reuse 则会获取对应 Pod 已退出状态的 App 容器，如果存在则拉起最新的 App 容器（可能有多个），如果不存在则直接新建。
2. 更新 App 容器映射的 pauseID 为新的 pause 容器 ID，这样就建立了 Pod 下新的 pause 容器和原先 App 容器的映射。
3. 重新拉起 App 容器即可完成 Pod 状态同步，最终即使宿主机重启或内核升级，容器数据也不会丢失。

2.2.3.2 Numa 感知与绑定

用户的另一个痛点与容器性能和稳定性相关。我们不断收到业务反馈，同样配置的容器性能存在不小的差异，主要表现为部分容器请求延迟很高，经过我们测试和深入分析发现：这些容器存在跨 Numa Node 访问 CPU，在我们将容器的 CPU 使用限制在同一个 Numa Node 后问题消失。所以，对于一些延迟敏感型的业务，我们要保证应用性能表现的一致性和稳定性，需要做到在调度侧感知 Numa Node 的使用情况。



为了解决这个问题，我们在 Node 层采集了 Numa Node 的分配情况，在调度器层

增加了对 Numa Node 的感知和调度，并保证资源使用的均衡性。对于一些强制需要绑定 Node 的敏感型应用，如果找不到合适的 Node 则扩容失败；对于一些不需要绑定 Numa Node 的应用，则可以选择尽量满足的策略。

2.2.3.3 其他稳定性优化

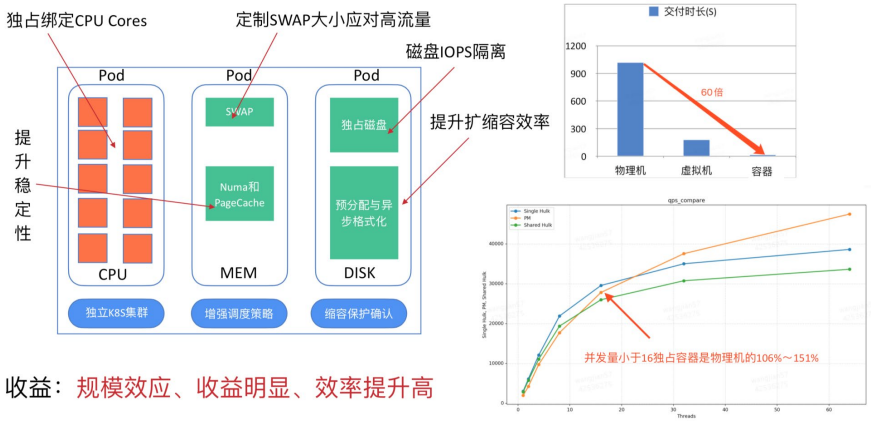
1. 在调度层面，我们为调度器增加了负载感知和基于服务画像应用特征的打散和优选策略。
2. 在故障容器发现和处理上，基于特征库落地的告警自愈组件，能够秒级发现 – 分析 – 处理告警。
3. 对于一些有特殊资源需求，例如高 IO、高内存等应用采用专区隔离，避免对其他应用造成影响。

2.2.4 平台型业务容器化

相信做过 ToB 业务的同学应该都了解，任何产品都存在大客户方案，那么对于美团这样的公司，内部也会存在这种情况。平台型业务的容器化有个特点是：实例数多，以千或万计，所以资源成本就比较高；业务地位比较高，一般都是非常核心的业务，对性能和稳定性要求很高。所以，如果想要通过“一招鲜”的方式解决此类业务的问题，就有些不切实际。

这里，我们以 MySQL 平台为例，数据库业务对于稳定性、性能和可靠性要求非常高，业务自己又主要以物理机为主，所以成本压力非常大。针对数据库的容器化，我们主要是从宿主机端的资源分配定制和优化为切入口。

1. 针对 CPU 资源分配，采用独占 CPU 集合的方式，避免 Pod 之间发生争抢。
2. 通过允许自定义 SWAP 大小来应对短暂的高流量，并关闭 Numa Node 和 PageCache 来提升稳定性。
3. 在磁盘分配中采用 Pod 独占磁盘进行 IOPS 的隔离，以及通过预分配和格式化磁盘来提升扩容的速度，提升资源交付效率。
4. 调度支持独有的打散策略和缩容确认，规避缩容风险。



最终，我们将数据库的交付效率提升了 60 倍，并且在大多数情况下性能比之前的物理机器还要好。

2.2.5 业务资源优先级保障

对于一个企业而言，基于成本考虑，资源一直会处于不足的状态，那么如何保障资源的供给和分配就显得非常重要。

1. 业务预算配额确定资源供给，通过专区来做专有资源专用。
2. 建设弹性资源池和打通公有云来应对突发资源需求。
3. 按照业务和应用类型的优先级保障资源使用，确保核心业务先拿到资源。
4. 多个 Kubenretes 集群和多机房来做容灾，应对集群或机房的故障。

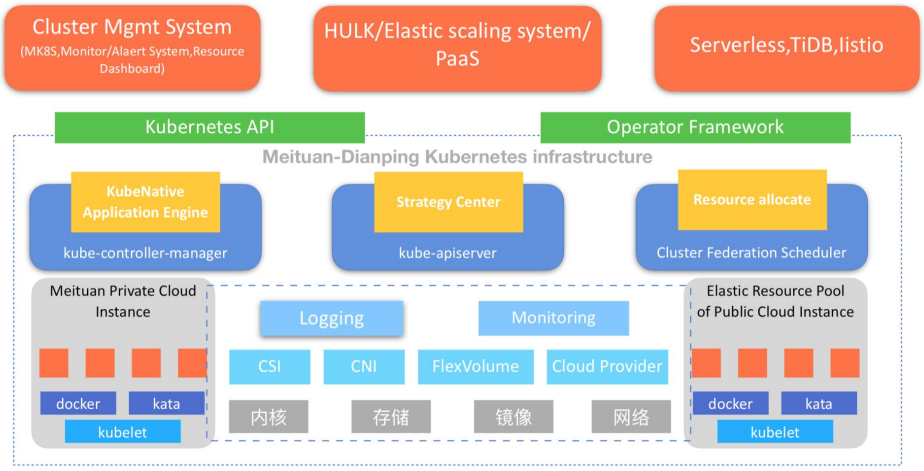
2.2.6 云原生架构的落地

在迁移到 Kubernetes 之后，我们进一步实现了云原生架构的落地。

为了解决云原生应用管理的障碍，我们设计实现了美团特色的云原生应用管理引擎——KubeNative，将应用的配置和信息管理对平台透明化，业务平台只需要创建原生的 Pod 资源即可，不需要关注应用的信息同步和管理细节，并支持各 PaaS 平台自己来扩展控制层面的能力，运行自己的 Operator。

下图就是目前我们整个的云原生应用管理架构，已支持 Hulk 容器平台、Serverless

以及 TiDB 等平台的落地。



2.3 基础设施迁移后的收益

1. 完成全公司业务 98% 的容器化，显著提升了资源管理的效率和业务稳定性。
2. Kubernetes 稳定性 99.99% 以上。
3. Kubernetes 成为美团内部集群管理平台的标准。

三、运营大规模 Kubernetes 集群的挑战和应对策略

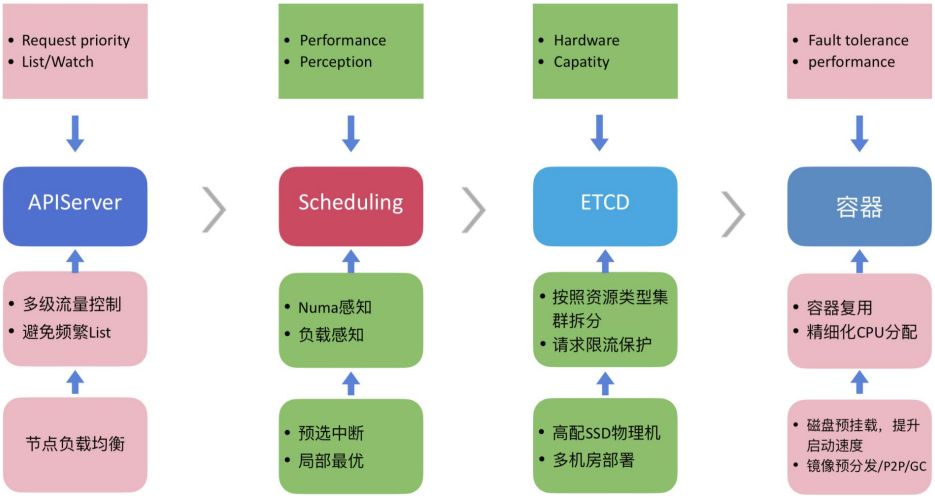
在整个基础设施迁移过程中，除了解决历史遗留问题和系统建设，随着 Kubernetes 集群规模和数量快速增长，我们遇到的新的挑战是：如何稳定、高效地运营大规模 Kubernetes 集群。我们在这几年的 Kubernetes 运营中，也逐渐摸索出了一套验证可行的运营经验。

3.1 核心组件优化与升级

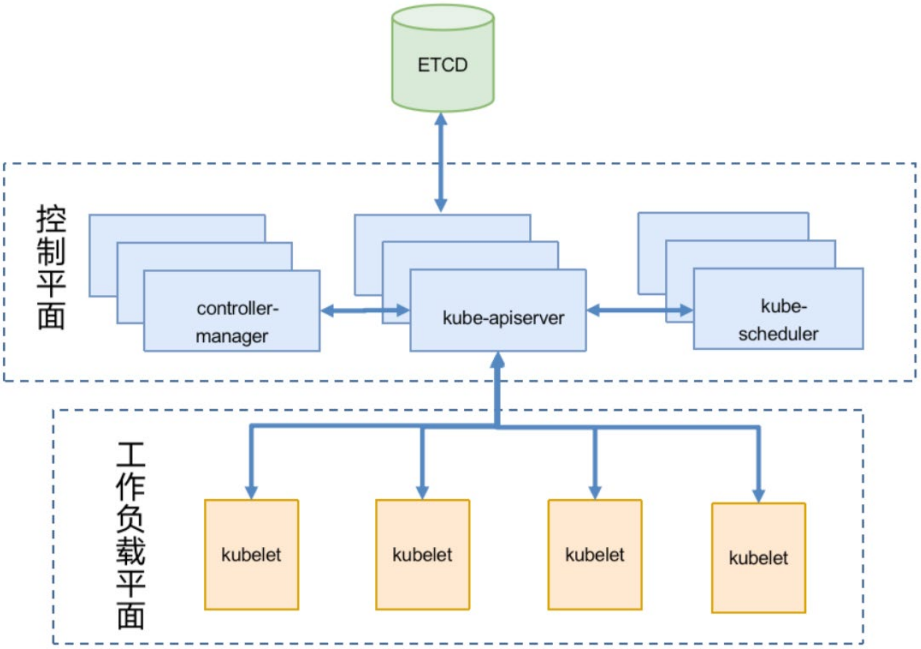
我们最初使用的 Kubernetes 是 1.6 版本，性能和稳定性是比较差的，当我们达到 1K 节点的时候就逐渐出现问题，达到 5K 节点时基本集群不可用。例如，调度性能非常差，集群吞吐量也比较低，偶尔还发生“雪崩”的情况，扩缩容链路耗时也在变长。

针对核心组件的分析和优化，这里从 kube-apiserver、kube-scheduler、etcd 以及容器等四个方面来概括下。

1. 针对 kube-apiserver，为了减少重启过程长时间地发生 429 请求重试，我们实现了多级的流量控制，将不可用窗口从 15min 降低为 1min，并通过减少和避免外部系统的 List 操作降低集群负载，通过内部的 VIP 来做节点的负载均衡，保障控制节点的稳定性。
2. 在 kube-scheduler 层，我们增强了调度的感知策略，调度效果相比之前更稳定；对调度性能的优化提出的预选中断和局部最优策略也已合并到社区，并成为通用的策略。
3. 针对 etcd 的运营，通过拆分出独立的 Event 集群降低主库的压力，并且基于高配的 SSD 物理机器部署可以达到日常 5 倍的高流量访问。
4. 在容器层面，容器复用提升了容器的故障容忍能力，并通过精细化的 CPU 分配提升应用稳定性；通过容器的磁盘预挂载提升 Node 的故障恢复速度。

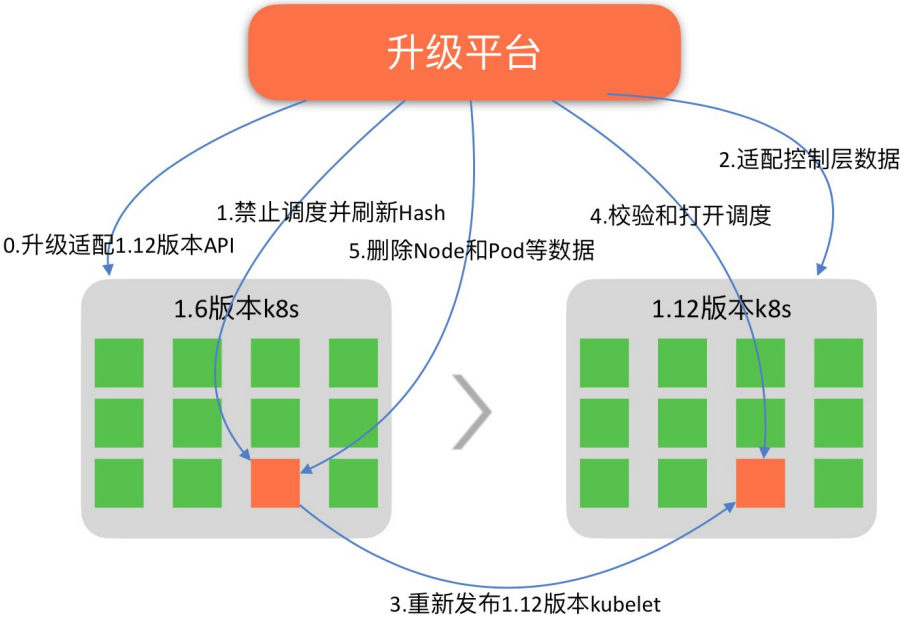


另外，社区版本的迭代是非常快的，高版本在稳定性和特性支持上更好，不可避免我们需要进行版本的升级，但如何确保升级成功是一个很大的挑战，尤其是我们在没有足够的 Buffer 资源进行资源腾挪情况下。



集群升级，业界通用的方案是直接基于原有集群升级，方案存在以下几点问题：

1. 升级版本有限制，不能跨大版本升级：只能一点点从低版本升级到高版本，耗时费力，而且成功率低。
2. 控制平面升级风险不可控：尤其是有 API 变更的时候，会覆盖之前的数据，甚至是不可回滚的。
3. 用户有感知，容器需要新建，成本和影响较高：这个是比较痛的点，无可避免会发生容器新建。



为此，我们深入研究了 Kubernetes 对容器层面的控制方式，设计实现了一种能够平滑将容器数据从低版本集群迁移到高版本集群的方案，将集群升级细化为 Node 粒度的逐个宿主机上容器的原地热升级，随时可以暂停和回滚。新方案主要是通过外部工具将 Node 和 Pod 数据从低版本集群迁移到高版本集群，并解决 Pod 对象和容器的兼容性问题。核心思路是两点：通过低版本兼容高版本的 API，通过刷新容器的 Hash 保障 Pod 下的容器不会被新；通过工具实现 Pod 和 Node 资源数据从低版本集群迁移到高版本集群。

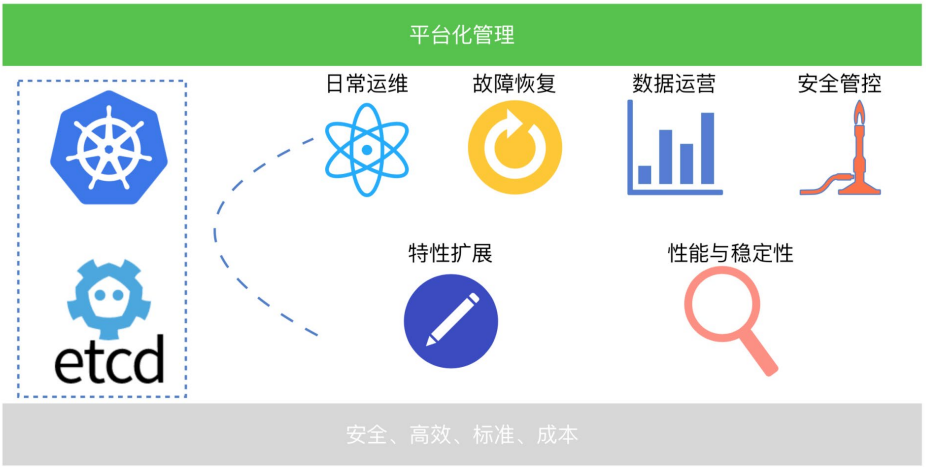
该方案亮点主要包括以下 4 个方面：

1. 大规模生产环境的集群升级不再是难题。
2. 解决了现有技术方案风险不可控的问题，风险降到了宿主机级别，升级更为安全。
3. 通用性强，可做到任意版本的升级，且方案生命周期长。
4. 优雅地解决了升级过程中容器新建问题，真正做到了原地热升级。

3.2 平台化与运营效率

大规模的集群运营是非常有挑战的事情，满足业务的快速发展和用户需求也是对团队极大的考验，我们需要从不同纬度的考虑集群的运营和研发能力。

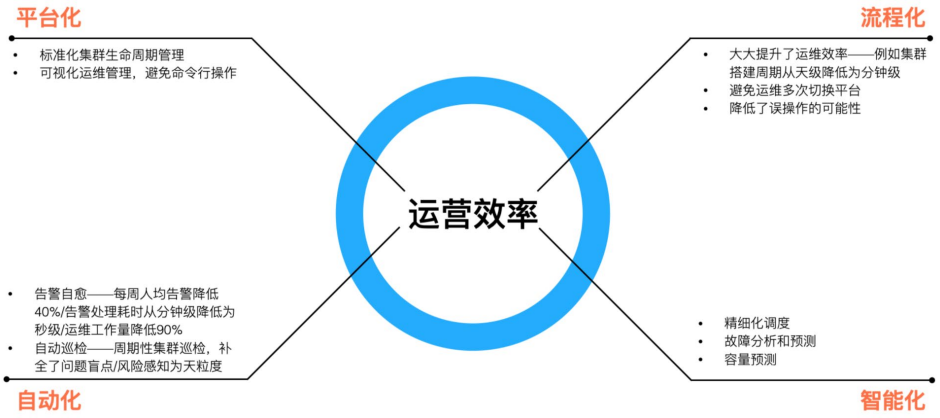
在 Kubernetes 与 etcd 集群的整个运营和运维能力建设上，我们关注的目标是安全运营、高效运维、标准化管理以及节约成本。所以针对 Kubernetes 与 etcd 集群，我们已经完成了平台化的管理运营，覆盖了特性扩展、性能与稳定性、日常运维、故障恢复、数据运营以及安全管控等 6 个方面。



对于一个非公有云业务的 Kubernetes 团队，人力还是非常有限的，除了集群的日常运营还有研发任务，所以我们对于运营效率的提升非常关注。我们将日常运维逐步的沉淀转换，构建了一套美团内部的 Kubernetes 集群管理平台。

1. 将集群的管理标准化、可视化，避免了黑白屏的运维操作。
2. 通过告警自愈和自动巡检将问题处理收敛掉，所以虽然我们有大几十个集群，但我们的运维效率还是比较高的，值班同学很少需要关注。
3. 全部的运维操作流程化，不仅提升了运维效率，人为操作导致的故障的概率也减小了。
4. 通过运营数据的分析进一步做了资源的精细化调度和故障预测，进一步提前

发现风险，提升了运营的质量。

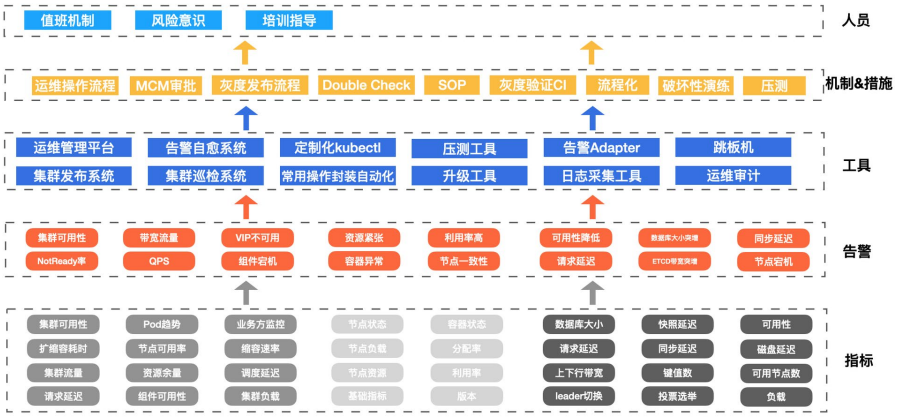


3.3 风险控制和可靠性保障

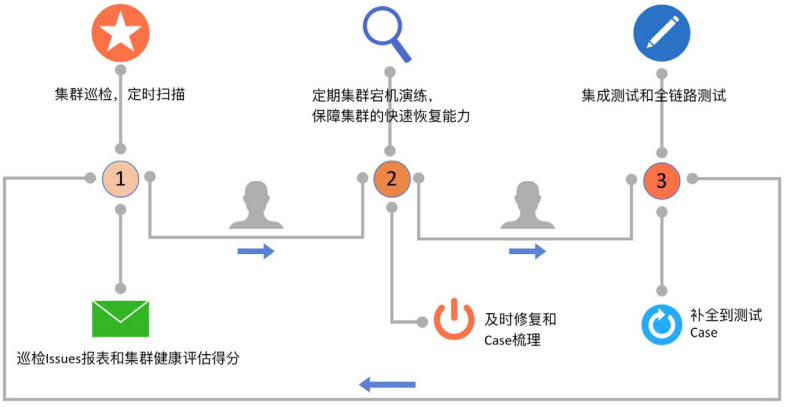
规模大、覆盖业务广，任何的集群故障都会直接影响到服务的稳定性甚至用户的体验，在经历了多次运维故障和安全压力下，我们形成了一套可复制的风险控制和可靠性保障策略。

在整个风险管控链路中，我们分为指标、告警、工具、机制 & 措施和人员 5 个层面：

1. 指标数据采集，从节点、集群、组件以及资源层面采集核心指标作为数据源。
2. 风险推送，覆盖核心指标的多级、多维度的告警机制。
3. 在工具支持上，通过主动、被动以及流程化等减少误操作风险。
4. 机制保障上，打通测试、灰度验证、发布确认以及演练等降低疏忽大意的情况。
5. 人是风险的根本，这块我们一直也在努力建设和轮值，确保问题的响应。



在可靠性验证和运营方面，我们笃信需要把功夫用在评审，通过集群巡检来评估集群的健康情况，并推送报表；定期的宕机演练保障真实故障能够快速恢复，并将日常问题补全到全链路测试中，形成闭环。



四、总结与未来展望

4.1 经验心得

1. Kubernetes 的落地完全兼容社区的 Kubernetes API；只会做插件化的扩展，并尽量不改控制层面的原有行为。
2. 对社区的一些特性，取长补短，并且有预期的升级，不盲目升级和跟进社区版本，尽量保持每年度的一个核心稳定版本。

3. 落地以用户痛点为突破口，业务是比较实际的，为什么需要进行迁移？业务会怕麻烦、不配合，所以推进要找到业务痛点，从帮助业务的角度出发，效果就会不一样。
4. 内部的集群管理运营的价值展现也是很重要的一环，让用户看到价值，业务看到潜在的收益，他们会主动来找你。

在容器时代，不能只看 Ku1. bernetes 本身，对于企业内的基础设施，“向上”和“向下”的融合和兼容问题也很关键。“向上”是面向业务场景为用户提供对接，因为容器并不能直接服务于业务，它还涉及到如何部署应用、服务治理、调度等诸多层面。“向下”，即容器与基础设施相结合的问题，这里更多的是兼容资源类型、更强大的隔离性、更高的资源使用效率等都是关键问题。

4.2 未来展望

1. 统一调度：VM 会少量长期存在一段时间，但如果同时维护两套基础设施产品成本是非常高的，所以我们也在落地 Kubernetes 来统一管理 VM 和容器。
2. VPA：探索通过 VPA 来进一步提升整个资源的使用效率。
3. 云原生应用管理：当前，我们已将云原生应用管理在生产环境落地，未来我们会进一步扩大云原生应用的覆盖面，不断提升研发效率。
4. 云原生架构落地：推进各个中间件、存储系统、大数据以及搜索业务合作落地各个领域的云原生系统。

作者介绍

国梁，美团点评技术专家，现负责美团点评 Kubernetes 集群的整体运营和维护以及云原生技术落地支持。

招聘信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度、Kubernetes、云原生等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件主题注明：基础架构）

基本功 | Java 即时编译器原理解析及实践

作者：吴天　珩智　薛超

一、导读

常见的编译型语言如 C++，通常会把代码直接编译成 CPU 所能理解的机器码来运行。而 Java 为了实现“一次编译，处处运行”的特性，把编译的过程分成两部分，首先它会先由 javac 编译成通用的中间形式——字节码，然后再由解释器逐条将字节码解释为机器码来执行。所以在性能上，Java 通常不如 C++ 这类编译型语言。

为了优化 Java 的性能，JVM 在解释器之外引入了即时（Just In Time）编译器：当程序运行时，解释器首先发挥作用，代码可以直接执行。随着时间推移，即时编译器逐渐发挥作用，把越来越多的代码编译优化成本地代码，来获取更高的执行效率。解释器这时可以作为编译运行的降级手段，在一些不可靠的编译优化出现问题时，再切换回解释执行，保证程序可以正常运行。

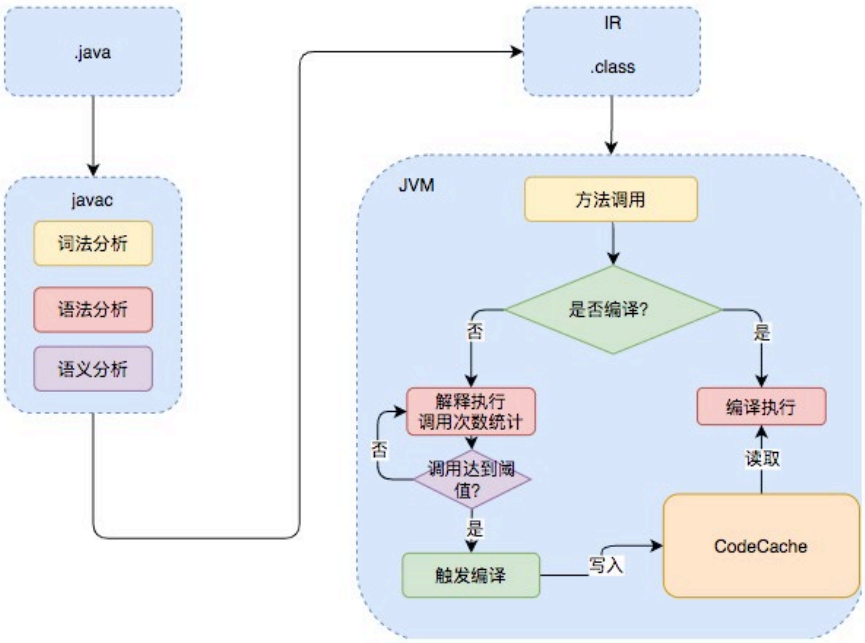
即时编译器极大地提高了 Java 程序的运行速度，而且跟静态编译相比，即时编译器可以选择性地编译热点代码，省去了很多编译时间，也节省很多的空间。目前，即时编译器已经非常成熟了，在性能层面甚至可以和编译型语言相比。不过在这个领域，大家依然在不断探索如何结合不同的编译方式，使用更加智能的手段来提升程序的运行速度。

二、Java 的执行过程

Java 的执行过程整体可以分为两个部分，第一步由 javac 将源码编译成字节码，在这个过程中会进行词法分析、语法分析、语义分析，编译原理中这部分的编译称为前端编译。接下来无需编译直接逐条将字节码解释执行，在解释执行的过程中，虚拟机同时对程序运行的信息进行收集，在这些信息的基础上，编译器会逐渐发挥作用，它

会进行后端编译——把字节码编译成机器码，但不是所有的代码都会被编译，只有被 JVM 认定为的热点代码，才可能被编译。

怎么样才会被认为是热点代码呢？ JVM 中会设置一个阈值，当方法或者代码块的在一定时间内的调用次数超过这个阈值时就会被编译，存入 codeCache 中。当下次执行时，再遇到这段代码，就会从 codeCache 中读取机器码，直接执行，以此来提升程序运行的性能。整体的执行过程大致如下图所示：



1. JVM 中的编译器

JVM 中集成了两种编译器，Client Compiler 和 Server Compiler，它们的作用也不同。Client Compiler 注重启动速度和局部的优化，Server Compiler 则更加关注全局的优化，性能会更好，但由于会进行更多的全局分析，所以启动速度会变慢。两种编译器有着不同的应用场景，在虚拟机中同时发挥作用。

Client Compiler

HotSpot VM 带有一个 Client Compiler C1 编译器。这种编译器启动速度快，但是性能比较 Server Compiler 来说会差一些。C1 会做三件事：

- 局部简单可靠的优化，比如字节码上进行的一些基础优化，方法内联、常量传播等，放弃许多耗时较长的全局优化。
- 将字节码构造成高级中间表示（High-level Intermediate Representation，以下称为 HIR），HIR 与平台无关，通常采用图结构，更适合 JVM 对程序进行优化。
- 最后将 HIR 转换成低级中间表示（Low-level Intermediate Representation，以下称为 LIR），在 LIR 的基础上会进行寄存器分配、窥孔优化（局部的优化方式，编译器在一个基本块或者多个基本块中，针对已经生成的代码，结合 CPU 自己指令的特点，通过一些认为可能带来性能提升的转换规则或者通过整体的分析，进行指令转换，来提升代码性能）等操作，最终生成机器码。

Server Compiler

Server Compiler 主要关注一些编译耗时较长的全局优化，甚至会还会根据程序运行的信息进行一些不可靠的激进优化。这种编译器的启动时间长，适用于长时间运行的后台程序，它的性能通常比 Client Compiler 高 30% 以上。目前，Hotspot 虚拟机中使用的 Server Compiler 有两种：C2 和 Graal。

C2 Compiler

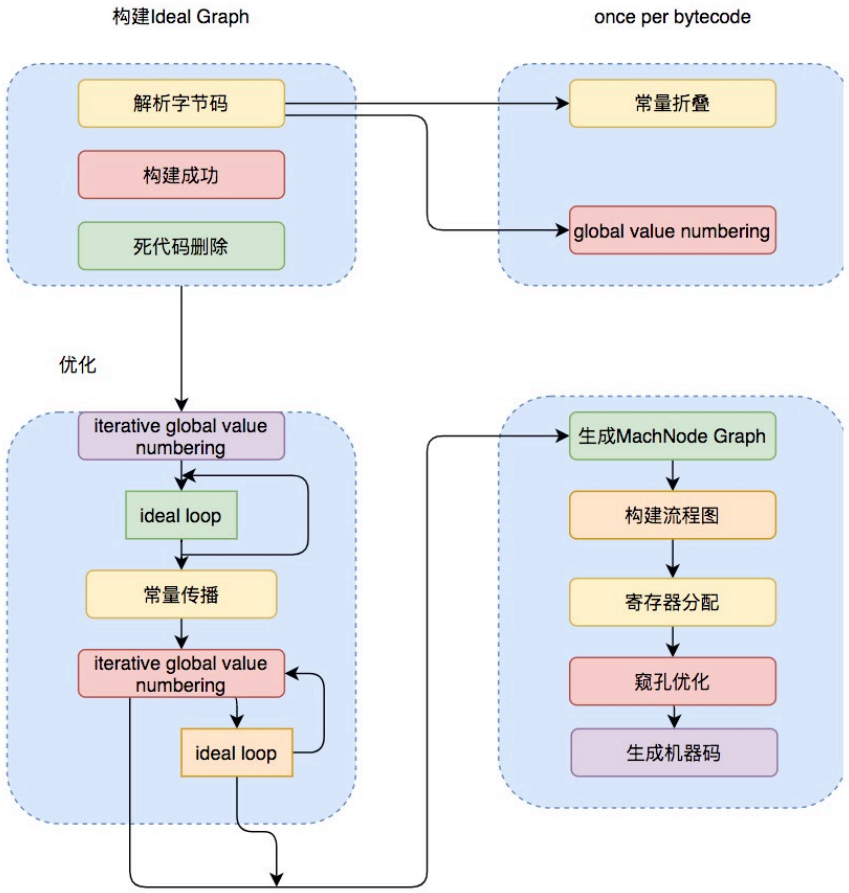
在 Hotspot VM 中，默认的 Server Compiler 是 C2 编译器。

C2 编译器在进行编译优化时，会使用一种控制流与数据流结合的图数据结构，称为 Ideal Graph。 Ideal Graph 表示当前程序的数据流向和指令间的依赖关系，依靠这种图结构，某些优化步骤（尤其是涉及浮动代码块的那些优化步骤）变得不那么复杂。

Ideal Graph 的构建是在解析字节码的时候，根据字节码中的指令向一个空的 Graph

中添加节点，Graph 中的节点通常对应一个指令块，每个指令块包含多条相关联的指令，JVM 会利用一些优化技术对这些指令进行优化，比如 Global Value Numbering、常量折叠等，解析结束后，还会进行一些死代码剔除的操作。生成 Ideal Graph 后，会在这个基础上结合收集的程序运行信息来进行一些全局的优化，这个阶段如果 JVM 判断此时没有全局优化的必要，就会跳过这部分优化。

无论是否进行全局优化，Ideal Graph 都会被转化为一种更接近机器层面的 MachNode Graph，最后编译的机器码就是从 MachNode Graph 中得的，生成机器码前还会有一些包括寄存器分配、窥孔优化等操作。关于 Ideal Graph 和各种全局的优化手段会在后面的章节详细介绍。Server Compiler 编译优化的过程如下图所示：



Graal Compiler

从 JDK 9 开始，Hotspot VM 中集成了一种新的 Server Compiler，Graal 编译器。相比 C2 编译器，Graal 有这样几种关键特性：

- 前文有提到，JVM 会在解释执行的时候收集程序运行的各种信息，然后编译器会根据这些信息进行一些基于预测的激进优化，比如分支预测，根据程序不同分支的运行概率，选择性地编译一些概率较大的分支。Graal 比 C2 更加青睐这种优化，所以 Graal 的峰值性能通常要比 C2 更好。
- 使用 Java 编写，对于 Java 语言，尤其是新特性，比如 Lambda、Stream 等更加友好。
- 更深层次的优化，比如虚函数的内联、部分逃逸分析等。

Graal 编译器可以通过 Java 虚拟机参数 -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 启用。当启用时，它将替换掉 HotSpot 中的 C2 编译器，并响应原本由 C2 负责的编译请求。

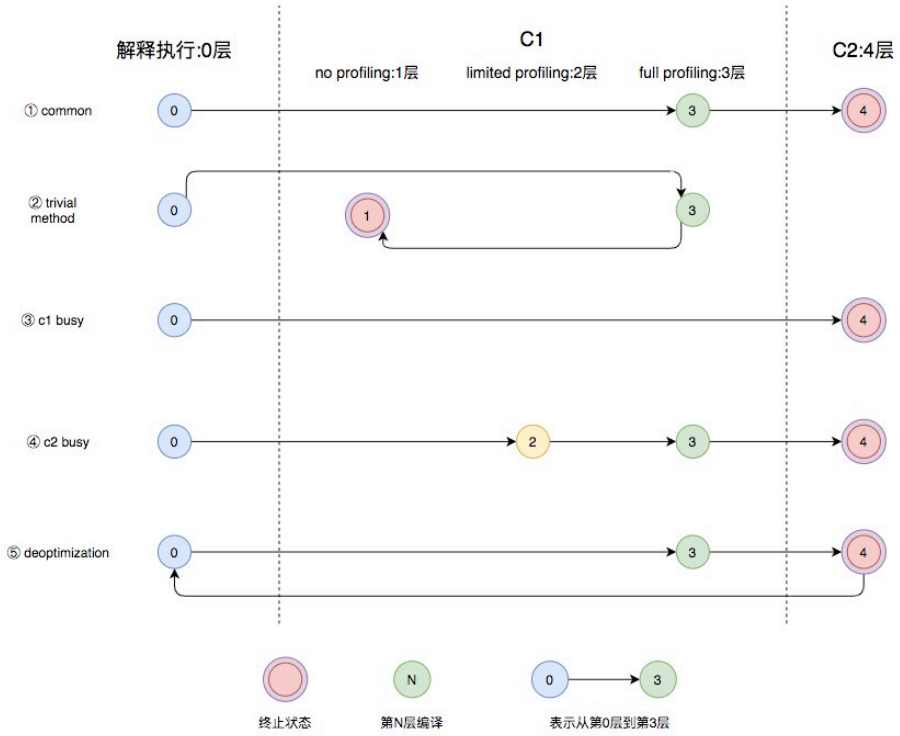
2. 分层编译

在 Java 7 以前，需要研发人员根据服务的性质去选择编译器。对于需要快速启动的，或者一些不会长期运行的服务，可以采用编译效率较高的 C1，对应参数 -client。长期运行的服务，或者对峰值性能有要求的后台服务，可以采用峰值性能更好的 C2，对应参数 -server。Java 7 开始引入了分层编译的概念，它结合了 C1 和 C2 的优势，追求启动速度和峰值性能的一个平衡。分层编译将 JVM 的执行状态分为了五个层次。五个层级分别是：

1. 解释执行。
2. 执行不带 profiling 的 C1 代码。
3. 执行仅带方法调用次数以及循环回边执行次数 profiling 的 C1 代码。
4. 执行带所有 profiling 的 C1 代码。
5. 执行 C2 代码。

profiling 就是收集能够反映程序执行状态的数据。其中最基本的统计数据就是方法的调用次数，以及循环回边的执行次数。

通常情况下，C2 代码的执行效率要比 C1 代码的高出 30% 以上。C1 层执行的代码，按执行效率排序从高至低则是 1 层 >2 层 >3 层。这 5 个层次中，1 层和 4 层都是终止状态，当一个方法到达终止状态后，只要编译后的代码并没有失效，那么 JVM 就不会再次发出该方法的编译请求的。服务实际运行时，JVM 会根据服务运行情况，从解释执行开始，选择不同的编译路径，直到到达终止状态。下图中就列举了几种常见的编译路径：



- 图中第①条路径，代表编译的一般情况，热点方法从解释执行到被 3 层的 C1 编译，最后被 4 层的 C2 编译。
- 如果方法比较小（比如 Java 服务中常见的 getter/setter 方法），3 层的 profiling 没有收集到有价值的数据，JVM 就会断定该方法对于 C1 代码和 C2

代码的执行效率相同，就会执行图中第②条路径。在这种情况下，JVM 会在 3 层编译之后，放弃进入 C2 编译，直接选择用 1 层的 C1 编译运行。

- 在 C1 忙碌的情况下，执行图中第③条路径，在解释执行过程中对程序进行 profiling ，根据信息直接由第 4 层的 C2 编译。
- 前文提到 C1 中的执行效率是 1 层 >2 层 >3 层，第 3 层一般要比第 2 层慢 35% 以上，所以在 C2 忙碌的情况下，执行图中第④条路径。这时方法会被 2 层的 C1 编译，然后再被 3 层的 C1 编译，以减少方法在 3 层的执行时间。
- 如果编译器做了一些比较激进的优化，比如分支预测，在实际运行时发现预测出错，这时就会进行反优化，重新进入解释执行，图中第⑤条执行路径代表的就是反优化。

总的来说，C1 的编译速度更快，C2 的编译质量更高，分层编译的不同编译路径，也就是 JVM 根据当前服务的运行情况来寻找当前服务的最佳平衡点的一个过程。从 JDK 8 开始，JVM 默认开启分层编译。

3. 即时编译的触发

Java 虚拟机根据方法的调用次数以及循环回边的执行次数来触发即时编译。循环回边是一个控制流图中的概念，程序中可以简单理解为往回跳转的指令，比如下面这段代码：

循环回边

```
public void nlp(Object obj) {
  int sum = 0;
  for (int i = 0; i < 200; i++) {
    sum += i;
  }
}
```

上面这段代码经过编译生成下面的字节码。其中，偏移量为 18 的字节码将往回跳至偏移量为 4 的字节码中。在解释执行时，每当运行一次该指令，Java 虚拟机便会将该方法的循环回边计数器加 1。

字节码

```
public void nlp(java.lang.Object);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: sipush        200
       8: if_icmpge     21
      11: iload_1
      12: iload_2
      13: iadd
      14: istore_1
      15: iinc          2, 1
      18: goto          4
      21: return
```

在即时编译过程中，编译器会识别循环的头部和尾部。上面这段字节码中，循环体的头部和尾部分别为偏移量为 11 的字节码和偏移量为 15 的字节码。编译器将在循环体结尾增加循环回边计数器的代码，来对循环进行计数。

当方法的调用次数和循环回边的次数的和，超过由参数 -XX:CompileThreshold 指定的阈值时（使用 C1 时，默认值为 1500；使用 C2 时，默认值为 10000），就会触发即时编译。

开启分层编译的情况下，-XX:CompileThreshold 参数设置的阈值将会失效，触发编译会由以下的条件来判断：

- 方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数。
- 方法调用次数大于由参数 -XX:TierXMINInvocationThreshold 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 -XX:TierXCompileThreshold 指定的阈值乘以系数时。

分层编译触发条件公式

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s
&& i + b > TierXCompileThreshold * s)
i 为调用次数，b 是循环回边次数
```

上述满足其中一个条件就会触发即时编译，并且 JVM 会根据当前的编译方法数以及编译线程数动态调整系数 s。

三、编译优化

即时编译器会对正在运行的服务进行一系列的优化，包括字节码解析过程中的分析，根据编译过程中代码的一些中间形式来做局部优化，还会根据程序依赖图进行全局优化，最后才会生成机器码。

1. 中间表达形式（Intermediate Representation）

在编译原理中，通常把编译器分为前端和后端，前端编译经过词法分析、语法分析、语义分析生成中间表达形式（Intermediate Representation，以下称为 IR），后端会对 IR 进行优化，生成目标代码。

Java 字节码就是一种 IR，但是字节码的结构复杂，字节码这样代码形式的 IR 也不适合做全局的分析优化。现代编译器一般采用图结构的 IR，静态单赋值（Static Single Assignment，SSA）IR 是目前比较常用的一种。这种 IR 的特点是每个变量只能被赋值一次，而且只有当变量被赋值之后才能使用。举个例子：

SSA IR

```
Plain Text
{
  a = 1;
  a = 2;
  b = a;
}
```

上述代码中我们可以轻易地发现 a = 1 的赋值是冗余的，但是编译器不能。传统的编

译器需要借助数据流分析，从后至前依次确认哪些变量的值被覆盖掉。不过，如果借助了 SSA IR，编译器则可以很容易识别冗余赋值。

上面代码的 SSA IR 形式的伪代码可以表示为：

SSA IR

```
Plain Text
{
  a_1 = 1;
  a_2 = 2;
  b_1 = a_2;
}
```

由于 SSA IR 中每个变量只能赋值一次，所以代码中的 a 在 SSA IR 中会分成 a_1、a_2 两个变量来赋值，这样编译器就可以很容易通过扫描这些变量来发现 a_1 的赋值后并没有使用，赋值是冗余的。

除此之外，SSA IR 对其他优化方式也有很大的帮助，例如下面这个死代码删除（Dead Code Elimination）的例子：

DeadCodeElimination

```
public void DeadCodeElimination{
  int a = 2;
  int b = 0
  if(2 > 1){
    a = 1;
  } else{
    b = 2;
  }
  add(a,b)
}
```

可以得到 SSA IR 伪代码：

DeadCodeElimination

```
a_1 = 2;
b_1 = 0
```

```
if true:
  a_2 = 1;
else
  b_2 = 2;
add(a,b)
```

编译器通过执行字节码可以发现 b_2 赋值后不会被使用，else 分支不会被执行。经过死代码删除后就可以得到代码：

DeadCodeElimination

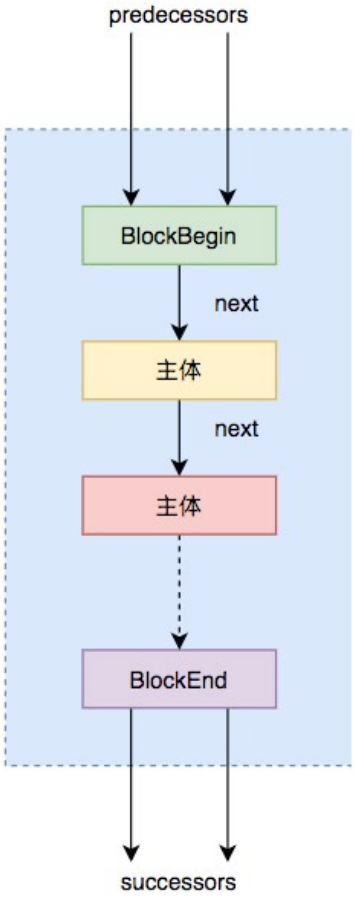
```
public void DeadCodeElimination{
  int a = 1;
  int b = 0;
  add(a,b)
}
```

我们可以将编译器的每一种优化看成一个图优化算法，它接收一个 IR 图，并输出经过转换后的 IR 图。编译器优化的过程就是一个个图节点的优化串联起来的。

C1 中的中间表达形式

前文提及 C1 编译器内部使用高级中间表达形式 HIR，低级中间表达形式 LIR 来进行各种优化，这两种 IR 都是 SSA 形式的。

HIR 是由很多基本块（Basic Block）组成的控制流图结构，每个块包含很多 SSA 形式的指令。基本块的结构如下图所示：



其中，predecessors 表示前驱基本块（由于前驱可能是多个，所以是 BlockList 结构，是多个 BlockBegin 组成的可扩容数组）。同样，successors 表示多个后继基本块 BlockEnd。除了这两部分就是主体块，里面包含程序执行的指令和一个 next 指针，指向下一个执行的主体块。

从字节码到 HIR 的构造最终调用的是 GraphBuilder，GraphBuilder 会遍历字节码构造所有代码基本块储存为一个链表结构，但是这个时候的基本块只有 BlockBegin，不包括具体的指令。第二步 GraphBuilder 会用一个 ValueStack 作为操作数栈和局部变量表，模拟执行字节码，构造出对应的 HIR，填充之前空的基本块，这里给出简单字节码块构造 HIR 的过程示例，如下所示：

字节码构造 HIR

```
        字节码                    Local Value            operand stack
HIR
      5: iload_1                  [i1,i2]                [i1]
      6: iload_2                  [i1,i2]                [i1,i2]
                                  ........................................
.........   i3: i1 * i2
      7: imul
      8: istore_3                 [i1,i2, i3]             [i3]
```

可以看出，当执行 iload_1 时，操作数栈压入变量 i1，执行 iload_2 时，操作数栈压入变量 i2，执行相乘指令 imul 时弹出栈顶两个值，构造出 HIR i3 : i1 * i2，生成的 i3 入栈。

C1 编译器优化大部分都是在 HIR 之上完成的。当优化完成之后它会将 HIR 转化为 LIR，LIR 和 HIR 类似，也是一种编译器内部用到的 IR，HIR 通过优化消除一些中间节点就可以生成 LIR，形式上更加简化。

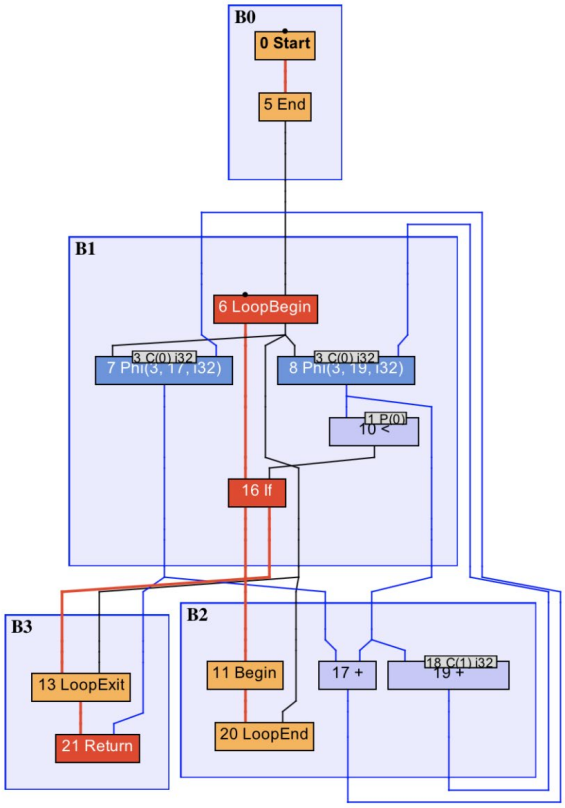
Sea-of-Nodes IR

C2 编译器中的 Ideal Graph 采用的是一种名为 Sea-of-Nodes 中间表达形式，同样也是 SSA 形式的。它最大特点是去除了变量的概念，直接采用值来进行运算。为了方便理解，可以利用 IR 可视化工具 Ideal Graph Visualizer（IGV），来展示具体的 IR 图。比如下面这段代码：

example

```
public static int foo(int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += i;
  }
  return sum;
}
```

对应的 IR 图如下所示：



图中若干个顺序执行的节点将被包含在同一个基本块之中，如图中的 B0、B1 等。B0 基本块中 0 号 Start 节点是方法入口，B3 中 21 号 Return 节点是方法出口。红色加粗线条为控制流，蓝色线条为数据流，而其他颜色的线条则是特殊的控制流或数据流。被控制流边所连接的是固定节点，其他的则是浮动节点（浮动节点指只要能满足数据依赖关系，可以放在不同位置的节点，浮动节点变动的这个过程称为 Schedule）。

这种图具有轻量级的边结构。 图中的边仅由指向另一个节点的指针表示。节点是 Node 子类的实例，带有指定输入边的指针数组。这种表示的优点是改变节点的输入边很快，如果想要改变输入边，只要将指针指向 Node，然后存入 Node 的指针数组就可以了。

依赖于这种图结构，通过收集程序运行的信息，JVM 可以通过 Schedule 那些浮动节点，从而获得最好的编译效果。

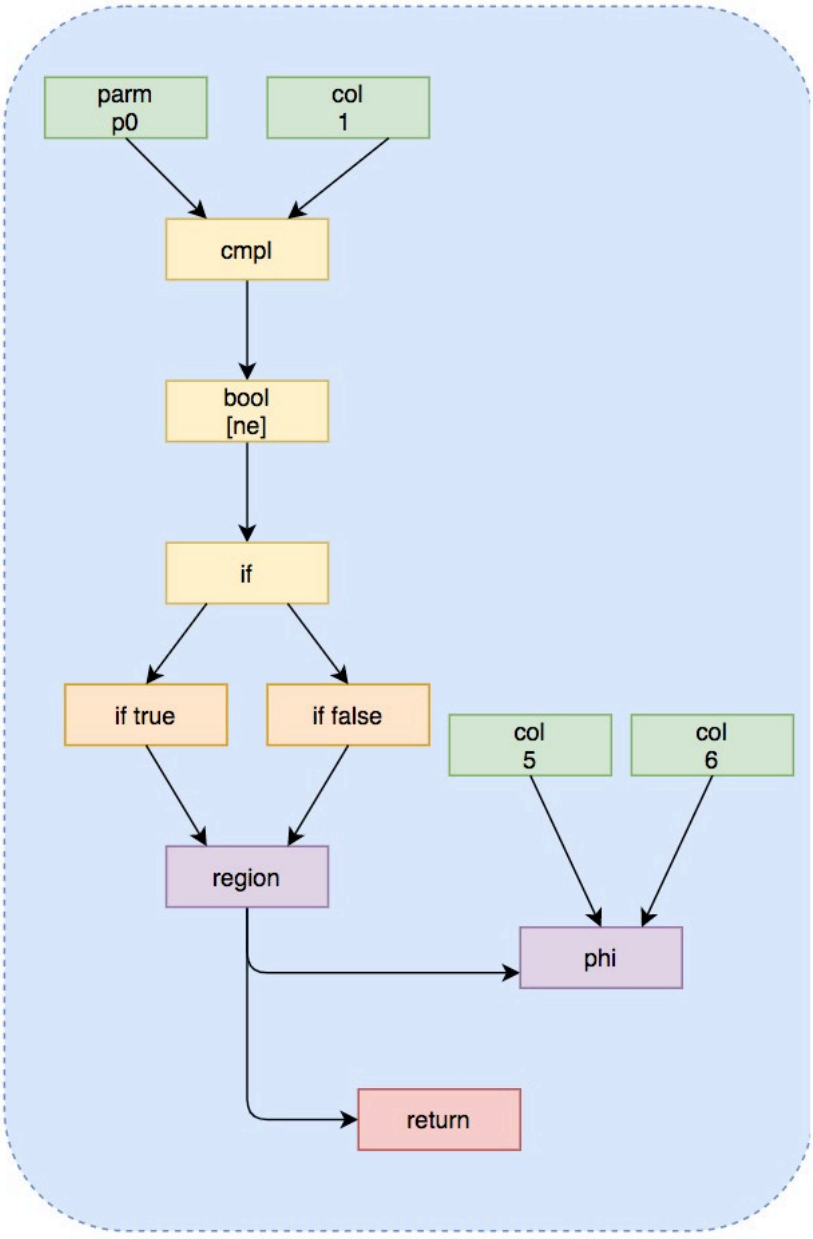
Phi And Region Nodes

Ideal Graph 是 SSA IR。 由于没有变量的概念，这会带来一个问题，就是不同执行路径可能会对同一变量设置不同的值。例如下面这段代码 if 语句的两个分支中，分别返回 5 和 6。此时，根据不同的执行路径，所读取到的值很有可能不同。

example

```
int test(int x) {
int a = 0;
  if(x == 1) {
    a = 5;
  } else {
    a = 6;
  }
  return a;
}
```

为了解决这个问题，就引入一个 Phi Nodes 的概念，能够根据不同的执行路径选择不同的值。于是，上面这段代码可以表示为下面这张图：



Phi Nodes 中保存不同路径上包含的所有值，Region Nodes 根据不同路径的判断条件，从 Phi Nodes 取得当前执行路径中变量应该赋予的值，带有 Phi 节点的 SSA 形式的伪代码如下：

Phi Nodes

```
int test(int x) {
  a_1 = 0;
  if(x == 1){
    a_2 = 5;
  }else {
    a_3 = 6;
  }
  a_4 = Phi(a_2,a_3);
  return a_4;
}
```

Global Value Numbering

Global Value Numbering（GVN）是一种因为 Sea-of-Nodes 变得非常容易的优化技术。

GVN 是指为每一个计算得到的值分配一个独一无二的编号，然后遍历指令寻找优化的机会，它可以发现并消除等价计算的优化技术。如果一段程序中出现了多次操作数相同的乘法，那么即时编译器可以将这些乘法合并为一个，从而降低输出机器码的大小。如果这些乘法出现在同一执行路径上，那么 GVN 还将省下冗余的乘法操作。在 Sea-of-Nodes 中，由于只存在值的概念，因此 GVN 算法将非常简单：即时编译器只需判断该浮动节点是否与已存在的浮动节点的编号相同，所输入的 IR 节点是否一致，便可以将这两个浮动节点归并成一个。比如下面这段代码：

GVN

```
a = 1;
b = 2;
c = a + b;
d = a + b;
e = d;
```

GVN 会利用 Hash 算法编号，计算 a = 1 时，得到编号 1，计算 b = 2 时得到编号 2，计算 c = a + b 时得到编号 3，这些编号都会放入 Hash 表中保存，在计算 d = a + b 时，会发现 a + b 已经存在 Hash 表中，就不会再进行计算，直接从 Hash 表中

取出计算过的值。最后的 e = d 也可以由 Hash 表中查到而进行复用。

可以将 GVN 理解为在 IR 图上的公共子表达式消除（Common Subexpression Elimination，CSE）。两者区别在于，GVN 直接比较值的相同与否，而 CSE 是借助词法分析器来判断两个表达式相同与否。

2. 方法内联

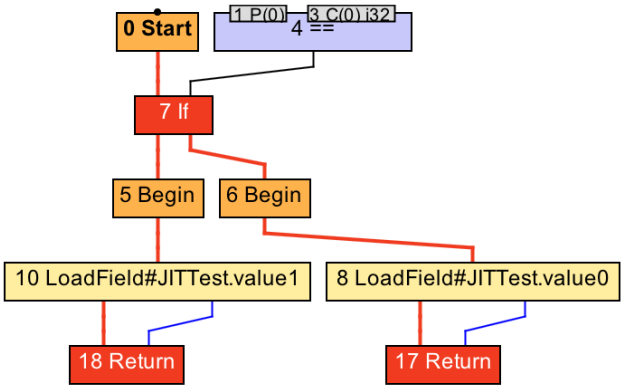
方法内联，是指在编译过程中遇到方法调用时，将目标方法的方法体纳入编译范围之中，并取代原方法调用的优化手段。JIT 大部分的优化都是在内联的基础上进行的，方法内联是即时编译器中非常重要的一环。

Java 服务中存在大量 getter/setter 方法，如果没有方法内联，在调用 getter/setter 时，程序执行时需要保存当前方法的执行位置，创建并压入用于 getter/setter 的栈帧、访问字段、弹出栈帧，最后再恢复当前方法的执行。内联了对 getter/setter 的方法调用后，上述操作仅剩字段访问。在 C2 编译器 中，方法内联在解析字节码的过程中完成。当遇到方法调用字节码时，编译器将根据一些阈值参数决定是否需要内联当前方法的调用。如果需要内联，则开始解析目标方法的字节码。比如下面这个示例（来源于网络）：

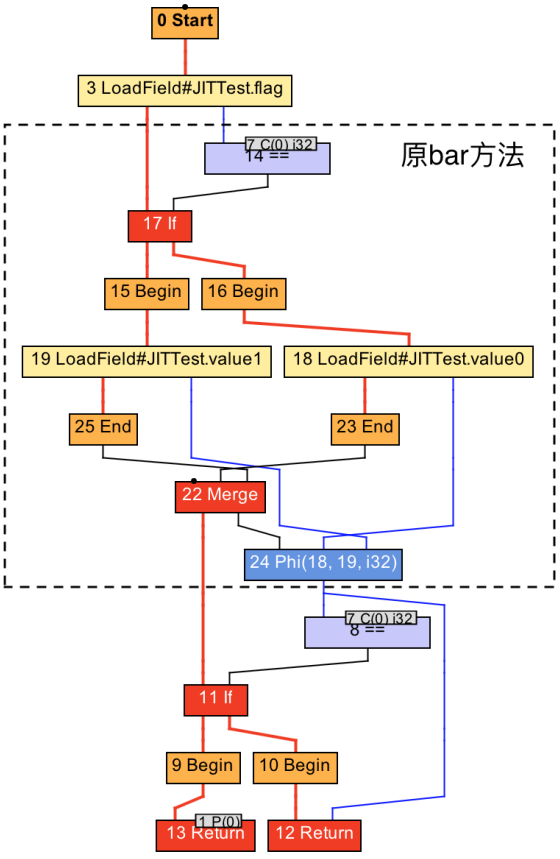
方法内联的过程

```
public static boolean flag = true;
public static int value0 = 0;
public static int value1 = 1;

public static int foo(int value) {
    int result = bar(flag);
    if (result != 0) {
        return result;
    } else {
        return value;
    }
}

public static int bar(boolean flag) {
    return flag ? value0 : value1;
}
```

bar 方法的 IR 图：



内联后的 IR 图：



内联不仅将被调用方法的 IR 图节点复制到调用者方法的 IR 图中，还要完成其他操作。

被调用方法的参数替换为调用者方法进行方法调用时所传入参数。上面例子中，将 bar 方法中的 1 号 P(0) 节点替换为 foo 方法 3 号 LoadField 节点。

调用者方法的 IR 图中，方法调用节点的数据依赖会变成被调用方法的返回。如果存在多个返回节点，会生成一个 Phi 节点，将这些返回值聚合起来，并作为原方法调用节点的替换对象。图中就是将 8 号 == 节点，以及 12 号 Return 节点连接到原 5 号 Invoke 节点的边，然后指向新生成的 24 号 Phi 节点中。

如果被调用方法将抛出某种类型的异常，而调用者方法恰好有该异常类型的处理器，并且该异常处理器覆盖这一方法调用，那么即时编译器需要将被调用方法抛出异常的路径，与调用者方法的异常处理器相连接。

方法内联的条件

编译器的大部分优化都是在方法内联的基础上。所以一般来说，内联的方法越多，生成代码的执行效率越高。但是对于即时编译器来说，内联的方法越多，编译时间也就越长，程序达到峰值性能的时刻也就比较晚。

可以通过虚拟机参数 -XX:MaxInlineLevel 调整内联的层数，以及 1 层的直接递归调用（可以通过虚拟机参数 -XX:MaxRecursiveInlineLevel 调整）。一些常见的内联相关的参数如下表所示：

参数名	默认值	说明
-XX:InlineSmallCode	2000	如果目标方法已被编译，且其生成的机器码大小超过该值，则无法内联
-XX:MaxTrivialSize	6	如果方法的字节码大小小于该值，则直接内联
-XX:MinInliningThreshold	250	如果目标方法的调用次数低于该值，则无法内联
-XX:InlineFrequencyCount	100	如果方法调用指令执行次数超过该值，则认为是热点方法
-XX:MaxInlineSize	35	如果非热点方法的字节码大小超过该值，则无法内联
-XX:FreqInlineSize	325	如果热点方法的字节码大小超过该值，则无法内联
-XX:LineNodeCountInliningCutof	40000	编译过程中IR节点数目的上限

虚函数内联

内联是 JIT 提升性能的主要手段，但是虚函数使得内联是很难的，因为在内联阶段并不知道他们会调用哪个方法。例如，我们有一个数据处理的接口，这个接口中的一个方法有三种实现 add、sub 和 multi，JVM 是通过保存虚函数表 Virtual Method Table（以下称为 VMT）存储 class 对象中所有的虚函数，class 的实例对象保存着一个 VMT 的指针，程序运行时首先加载实例对象，然后通过实例对象找到 VMT，通过 VMT 找到对应方法的地址，所以虚函数的调用比直接指向方法地址的 classic call 性能上会差一些。很不幸的是，Java 中所有非私有的成员函数的调用都是虚调用。

C2 编译器已经足够智能，能够检测这种情况并会对虚调用进行优化。比如下面这段代码例子：

virtual call

```
public class SimpleInliningTest
{
    public static void main(String[] args) throws InterruptedException
{
        VirtualInvokeTest obj = new VirtualInvokeTest();
        VirtualInvoke1 obj1 = new VirtualInvoke1();
        for (int i = 0; i < 100000; i++) {
            invokeMethod(obj);
            invokeMethod(obj1);
```

```
        }
        Thread.sleep(1000);
    }

    public static void invokeMethod(VirtualInvokeTest obj) {
        obj.methodCall();
    }

    private static class VirtualInvokeTest {
        public void methodCall() {
            System.out.println("virtual call");
        }
    }

    private static class VirtualInvoke1 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
}
```

经过 JIT 编译器优化后，进行反汇编得到下面这段汇编代码：

```
 0x0000000113369d37: callq  0x00000001132950a0  ; OopMap{off=476}
                                                ;*invokevirtual
methodCall  // 代表虚调用
                                                ; -
SimpleInliningTest::invokeMethod@1 (line 18)
                                                ;   {optimized virtual_
call}  // 虚调用已经被优化
```

可以看到 JIT 对 methodCall 方法进行了虚调用优化 optimized virtual_call。经过优化后的方法可以被内联。但是 C2 编译器的能力有限，对于多个实现方法的虚调用就“无能为力”了。

比如下面这段代码，我们增加一个实现：

多实现的虚调用

```
public class SimpleInliningTest
{
    public static void main(String[] args) throws InterruptedException
{
```

```
        VirtualInvokeTest obj = new VirtualInvokeTest();
        VirtualInvoke1 obj1 = new VirtualInvoke1();
        VirtualInvoke2 obj2 = new VirtualInvoke2();
        for (int i = 0; i < 100000; i++) {
            invokeMethod(obj);
            invokeMethod(obj1);
        invokeMethod(obj2);
        }
        Thread.sleep(1000);
    }

    public static void invokeMethod(VirtualInvokeTest obj) {
        obj.methodCall();
    }

    private static class VirtualInvokeTest {
        public void methodCall() {
            System.out.println("virtual call");
        }
    }

    private static class VirtualInvoke1 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
    private static class VirtualInvoke2 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
}
```

经过反编译得到下面的汇编代码：

代码块

```
 0x000000011f5f0a37: callq  0x000000011f4fd2e0  ; OopMap{off=28}
                                                ;*invokevirtual
methodCall  // 代表虚调用
                                                ; -
SimpleInliningTest::invokeMethod@1 (line 20)
                                                ;   {virtual_call}  // 虚
调用未被优化
```

可以看到多个实现的虚调用未被优化，依然是 virtual_call。

Graal 编译器针对这种情况，会去收集这部分执行的信息，比如在一段时间，发现前面的接口方法的调用 add 和 sub 是各占 50% 的几率，那么 JVM 就会在每次运行时，遇到 add 就把 add 内联进来，遇到 sub 的情况再把 sub 函数内联进来，这样这两个路径的执行效率就会提升。在后续如果遇到其他不常见的情况，JVM 就会进行去优化的操作，在那个位置做标记，再遇到这种情况时切换回解释执行。

3. 逃逸分析

逃逸分析是“一种确定指针动态范围的静态分析，它可以分析在程序的哪些地方可以访问到指针”。Java 虚拟机的即时编译器会对新建的对象进行逃逸分析，判断对象是否逃逸出线程或者方法。即时编译器判断对象是否逃逸的依据有两种：

1. 对象是否被存入堆中（静态字段或者堆中对象的实例字段），一旦对象被存入堆中，其他线程便能获得该对象的引用，即时编译器就无法追踪所有使用该对象的代码位置。
2. 对象是否被传入未知代码中，即时编译器会将未被内联的代码当成未知代码，因为它无法确认该方法调用会不会将调用者或所传入的参数存储至堆中，这种情况，可以直接认为方法调用的调用者以及参数是逃逸的。

逃逸分析通常是在方法内联的基础上进行的，即时编译器可以根据逃逸分析的结果进行诸如锁消除、栈上分配以及标量替换的优化。下面这段代码的就是对象未逃逸的例子：

```
pulbic class Example{
    public static void main(String[] args) {
      example();
    }
    public static void example() {
      Foo foo = new Foo();
      Bar bar = new Bar();
      bar.setFoo(foo);
    }
  }
```

```
  class Foo {}

  class Bar {
    private Foo foo;
    public void setFoo(Foo foo) {
      this.foo = foo;
    }
  }
}
```

在这个例子中，创建了两个对象 foo 和 bar，其中一个作为另一个方法的参数提供。该方法 setFoo() 存储对收到的 Foo 对象的引用。如果 Bar 对象在堆上，则对 Foo 的引用将逃逸。但是在这种情况下，编译器可以通过逃逸分析确定 Bar 对象本身不会对逃逸出 example() 的调用。这意味着对 Foo 的引用也不能逃逸。因此，编译器可以安全地在栈上分配两个对象。

锁消除

在学习 Java 并发编程时会了解锁消除，而锁消除就是在逃逸分析的基础上进行的。

如果即时编译器能够证明锁对象不逃逸，那么对该锁对象的加锁、解锁操作没就有意义。因为线程并不能获得该锁对象。在这种情况下，即时编译器会消除对该不逃逸锁对象的加锁、解锁操作。实际上，编译器仅需证明锁对象不逃逸出线程，便可以进行锁消除。由于 Java 虚拟机即时编译的限制，上述条件被强化为证明锁对象不逃逸出当前编译的方法。不过，基于逃逸分析的锁消除实际上并不多见。

栈上分配

我们都知道 Java 的对象是在堆上分配的，而堆是对所有对象可见的。同时，JVM 需要对所分配的堆内存进行管理，并且在对象不再被引用时回收其所占据的内存。如果逃逸分析能够证明某些新建的对象不逃逸，那么 JVM 完全可以将其分配至栈上，并且在 new 语句所在的方法退出时，通过弹出当前方法的栈桢来自动回收所分配的内存空间。这样一来，我们便无须借助垃圾回收器来处理不再被引用的对象。不过 Hotspot 虚拟机，并没有进行实际的栈上分配，而是使用了标量替换这一技术。所谓

的标量，就是仅能存储一个值的变量，比如 Java 代码中的基本类型。与之相反，聚合量则可能同时存储多个值，其中一个典型的例子便是 Java 的对象。编译器会在方法内将未逃逸的聚合量分解成多个标量，以此来减少堆上分配。下面是一个标量替换的例子：

标量替换

```
public class Example{
  @AllArgsConstructor
  class Cat{
    int age;
    int weight;
  }
  public static void example(){
    Cat cat = new Cat(1,10);
    addAgeAndWeight(cat.age,Cat.weight);
  }
}
```

经过逃逸分析，cat 对象未逃逸出 example() 的调用，因此可以对聚合量 cat 进行分解，得到两个标量 age 和 weight，进行标量替换后的伪代码：

```
public class Example{
  @AllArgsConstructor
  class Cat{
    int age;
    int weight;
  }
  public static void example(){
    int age = 1;
    int weight = 10;
    addAgeAndWeight(age,weight);
  }
}
```

部分逃逸分析

部分逃逸分析也是 Graal 对于概率预测的应用。通常来说，如果发现一个对象逃逸出了方法或者线程，JVM 就不会去进行优化，但是 Graal 编译器依然会去分析当前程序的执行路径，它会在逃逸分析基础上收集、判断哪些路径上对象会逃逸，哪

些不会。然后根据这些信息，在不会逃逸的路径上进行锁消除、栈上分配这些优化手段。

4. Loop Transformations

在文章中介绍 C2 编译器的部分有提及到，C2 编译器在构建 Ideal Graph 后会进行很多的全局优化，其中就包括对循环的转换，最重要的两种转换就是循环展开和循环分离。

循环展开

循环展开是一种循环转换技术，它试图以牺牲程序二进制码大小为代价来优化程序的执行速度，是一种用空间换时间的优化手段。

循环展开通过减少或消除控制程序循环的指令，来减少计算开销，这种开销包括增加指向数组中下一个索引或者指令的指针算数等。如果编译器可以提前计算这些索引，并且构建到机器代码指令中，那么程序运行时就可以不必进行这种计算。也就是说有些循环可以写成一些重复独立的代码。比如下面这个循环：

循环展开

```
public void loopRolling(){
  for(int i = 0;i<200;i++){
    delete(i);
  }
}
```

上面的代码需要循环删除 200 次，通过循环展开可以得到下面这段代码：

循环展开

```
public void loopRolling(){
  for(int i = 0;i<200;i+=5){
    delete(i);
    delete(i+1);
    delete(i+2);
    delete(i+3);
    delete(i+4);
  }
}
```

这样展开就可以减少循环的次数，每次循环内的计算也可以利用 CPU 的流水线提升效率。当然这只是一个示例，实际进行展开时，JVM 会去评估展开带来的收益，再决定是否进行展开。

循环分离

循环分离也是循环转换的一种手段。它把循环中一次或多次的特殊迭代分离出来，在循环外执行。举个例子，下面这段代码：

循环分离

```
int a = 10;
for(int i = 0;i<10;i++){
  b[i] = x[i] + x[a];
  a = i;
}
```

可以看出这段代码除了第一次循环 a = 10 以外，其他的情况 a 都等于 i−1。所以可以把特殊情况分离出去，变成下面这段代码：

循环分离

```
b[0] = x[0] + 10;
for(int i = 1;i<10;i++){
  b[i] = x[i] + x[i-1];
}
```

这种等效的转换消除了在循环中对 a 变量的需求，从而减少了开销。

5. 窥孔优化与寄存器分配

前文提到的窥孔优化是优化的最后一步，这之后就会程序就会转换成机器码，窥孔优化就是将编译器所生成的中间代码（或目标代码）中相邻指令，将其中的某些组合替换为效率更高的指令组，常见的比如强度削减、常数合并等，看下面这个例子就是一个强度削减的例子：

强度削减

```
y1=x1*3   经过强度削减后得到   y1=(x1<<1)+x1
```

编译器使用移位和加法削减乘法的强度，使用更高效率的指令组。

寄存器分配也是一种编译的优化手段，在 C2 编译器中普遍的使用。它是通过把频繁使用的变量保存在寄存器中，CPU 访问寄存器的速度比内存快得多，可以提升程序的运行速度。

寄存器分配和窥孔优化是程序优化的最后一步。经过寄存器分配和窥孔优化之后，程序就会被转换成机器码保存在 codeCache 中。

四、实践

即时编译器情况复杂，同时网络上也很少有实战经验，以下是我们团队的一些调整经验。

1. 编译相关的重 * 要参数

- -XX:+TieredCompilation：开启分层编译，JDK8 之后默认开启
- -XX:+CICompilerCount=N：编译线程数，设置数量后，JVM 会自动分配线程数，C1:C2 = 1:2
- -XX:TierXBackEdgeThreshold：OSR 编译的阈值
- -XX:TierXMinInvocationThreshold：开启分层编译后各层调用的阈值
- -XX:TierXCompileThreshold：开启分层编译后的编译阈值
- -XX:ReservedCodeCacheSize：codeCache 最大大小
- -XX:InitialCodeCacheSize：codeCache 初始大小

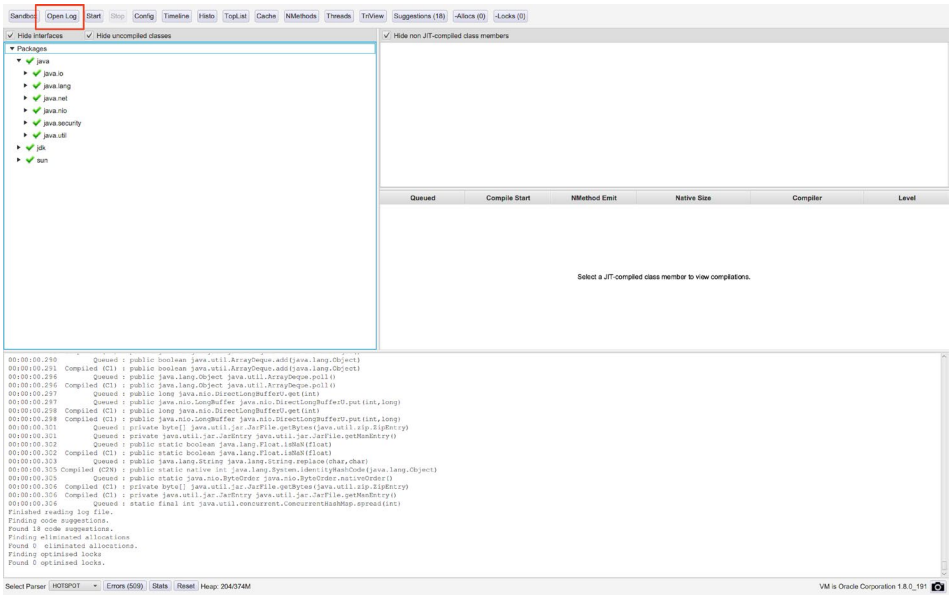
-XX:TierXMinInvocationThreshold 是开启分层编译的情况下，触发编译的阈值参数，当方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数，或者当方法调用次数大于由参数 -XX:TierXMINInvocationThreshold 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 -XX:TierX-

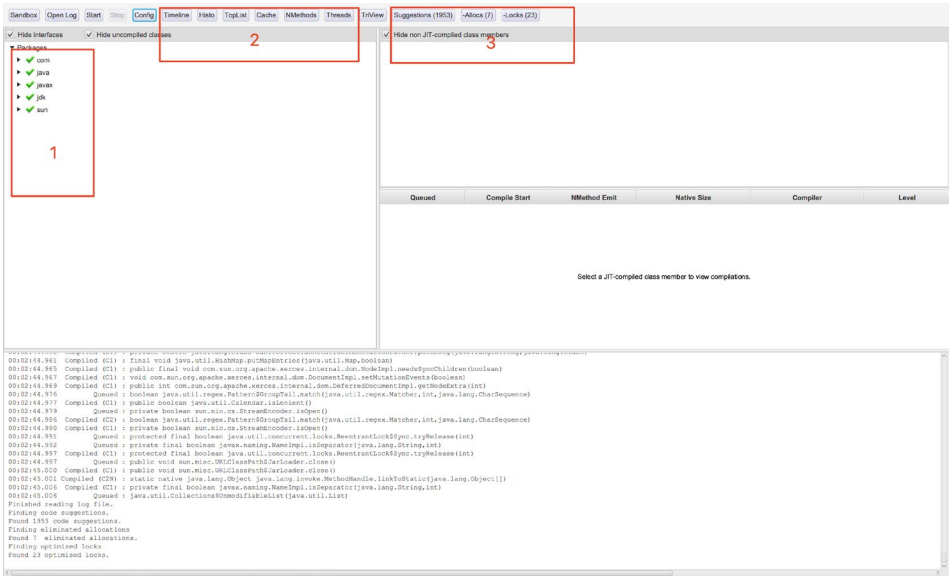
CompileThreshold 指定的阈值乘以系数时，便会触发 X 层即时编译。分层编译开启下会乘以一个系数，系数根据当前编译的方法和编译线程数确定，降低阈值可以提升编译方法数，一些常用但是不能编译的方法可以编译优化提升性能。

由于编译情况复杂，JVM 也会动态调整相关的阈值来保证 JVM 的性能，所以不建议手动调整编译相关的参数。除非一些特定的 Case，比如 codeCache 满了停止了编译，可以适当增加 codeCache 大小，或者一些非常常用的方法，未被内联到，拖累了性能，可以调整内敛层数或者内联方法的大小来解决。

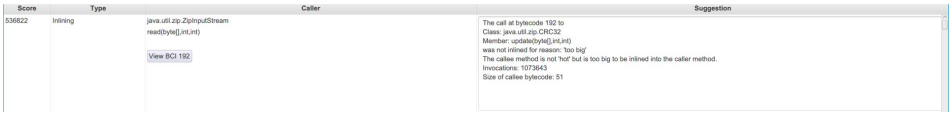
2. 通过 JITwatch 分析编译日志

通过增加 -XX:+UnlockDiagnosticVMOptions -XX:+PrintCompilation -XX-:+PrintInlining -XX:+PrintCodeCache -XX:+PrintCodeCacheOnCompilation -XX:+TraceClassLoading -XX:+LogCompilation -XX:LogFile=LogPath 参数可以输出编译、内联、codeCache 信息到文件。但是打印的编译日志多且复杂很难直接从其中得到信息，可以使用 JITwatch 的工具来分析编译日志。JITwatch 首页的 Open Log 选中日志文件，点击 Start 就可以开始分析日志。





如上图所示，区域 1 中是整个项目 Java Class 包括引入的第三方依赖；区域 2 是功能区 Timeline 以图形的形式展示 JIT 编译的时间轴，Histo 是直方图展示一些信息，TopList 里面是编译中产生的一些对象和数据的排序，Cache 是空闲 codeCache 空间，NMethod 是 Native 方法，Threads 是 JIT 编译的线程；区域 3 是 JITwatch 对日志分析结果的展示，其中 Suggestions 中会给出一些代码优化的建议，举个例子，如下图中：



我们可以看到在调用 ZipInputStream 的 read 方法时，因为该方法没有被标记为热点方法，同时又“太大了”，导致无法被内联到。使用 -XX:CompileCommand 中 inline 指令可以强制方法进行内联，不过还是建议谨慎使用，除非确定某个方法内联会带来不少的性能提升，否则不建议使用，并且过多使用对编译线程和 codeCache 都会带来不小的压力。

区域 3 中的 -Allocs 和 -Locks 逃逸分析后 JVM 对代码做的优化，包括栈上分配、锁消除等。

3. 使用 Graal 编译器

由于 JVM 会去根据当前的编译方法数和编译线程数对编译阈值进行动态的调整，所以实际服务中对这一部分的调整空间是不大的，JVM 做的已经足够多了。

为了提升性能，在服务中尝试了最新的 Graal 编译器。只需要使用 -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 就可以启动 Graal 编译器来代替 C2 编译器，并且响应 C2 的编译请求，不过要注意的是，Graal 编译器与 ZGC 不兼容，只能与 G1 搭配使用。

前文有提到过，Graal 是一个用 Java 写的即时编译器，它从 Java 9 开始便被集成自 JDK 中，作为实验性质的即时编译器。Graal 编译器就是脱身于 GraalVM，GraalVM 是一个高性能的、支持多种编程语言的执行环境。它既可以在传统的 OpenJDK 上运行，也可以通过 AOT（Ahead-Of-Time）编译成可执行文件单独运行，甚至可以集成至数据库中运行。

前文提到过数次，Graal 的优化都基于某种假设（Assumption）。当假设出错的情况下，Java 虚拟机会借助去优化（Deoptimization）这项机制，从执行即时编译器生成的机器码切换回解释执行，在必要情况下，它甚至会废弃这份机器码，并在重新收集程序 profile 之后，再进行编译。

这些中激进的手段使得 Graal 的峰值性能要好于 C2，而且在 Scale、Ruby 这种语言 Graal 表现更加出色，Twitter 目前已经在服务中大量的使用 Graal 来提升性能，企业版的 GraalVM 使得 Twitter 服务性能提升了 22%。

使用 Graal 编译器后性能表现

在我们的线上服务中，启用 Graal 编译后，TP9999 从 60ms -> 50ms ，下降 10ms，下降幅度达 16.7%。

运行过程中的峰值性能会更高。可以看出对于该服务，Graal 编译器带来了一定的性能提升。

Graal 编译器的问题

Graal 编译器的优化方式更加激进，因此在启动时会进行更多的编译，Graal 编译器本身也需要被即时编译，所以服务刚启动时性能会比较差。

考虑的解决办法：JDK 9 开始提供工具 jaotc，同时 GraalVM 的 Native Image 都是可以通过静态编译，极大地提升服务的启动速度的方式，但是 GraalVM 会使用自己的垃圾回收，这是一种很原始的基于复制算法的垃圾回收，相比 G1、ZGC 这些优秀的新型垃圾回收器，它的性能并不好。同时 GraalVM 对 Java 的一些特性支持也不够，比如基于配置的支持，比如反射就需要把所有需要反射的类配置一个 JSON 文件，在大量使用反射的服务，这样的配置会是很大的工作量。我们也在做这方面的调研。

五、总结

本文主要介绍了 JIT 即时编译的原理以及在美团一些实践的经验，还有最前沿的即时编译器的使用效果。作为一项解释型语言中提升性能的技术，JIT 已经比较成熟了，在很多语言中都有使用。对于 Java 服务，JVM 本身已经做了足够多，但是我们还应该不断深入了解 JIT 的优化原理和最新的编译技术，从而弥补 JIT 的劣势，提升 Java 服务的性能，不断追求卓越。

六、参考文献

《深入理解 Java 虚拟机》

《Proceedings of the Java™ Virtual Machine Research and Technology Symposium》Monterey, California, USA April 23 – 24, 2001

《Visualization of Program Dependence Graphs》Thomas W ü rthinger

《深入拆解 Java 虚拟机》郑宇迪

JIT 的 Profile 神器 JITWatch

作者简介

珩智，昊天，薛超，均来自美团 AI 平台 / 搜索与 NLP 部。

招聘信息

美团搜索与 NLP 部，长期招聘搜索、对话、NLP 算法工程师，坐标北京 / 上海，感兴趣的同学可投递简历至：tech@meituan.com（邮件标题请注明：搜索与 NLP 部）。

MyBatis 版本升级引发的线上告警回顾及原理分析

作者：凯伦

背景

某天晚上，美团到店事业群某项系统服务正在进行常规需求的上线。因为在内部的 Plus 系统发布时，提示 inf-bom 版本需要升级，于是我们就将 inf-bom 版本从 1.3.9.6 升级至 1.4.2.1，如下图 1 所示：

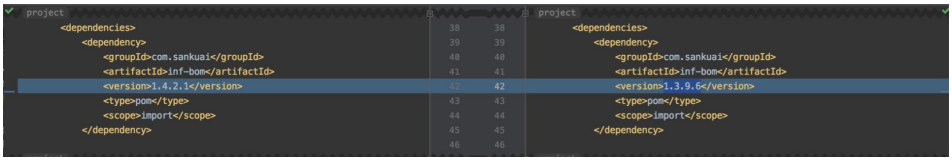


图 1　版本升级

不过，当服务上线后，开始陆续出现了一些更新系统交互日志方面的报警，这属于系统的辅助流程，报警如下方代码所示。我们发现都是跟 MyBatis 相关的报警，说明在进行类型转换的时候，系统产生了强转错误。

```
更新开票请求返回日志 , id:{#######},
response:{{"code":XXX,"data":{"callType":3,"code":XXX,"msg":"XXXX",
"shopId":XXXXX,"taxPlateDockType":"XXXXXXX"},"msg":"XXXXX",
"success":XXXX}}
nested execption is org.apache.ibatis.type.TypeException: Could not set
parameters for mapping: ParameterMapping{property='updateTime', mode=IN,
javaType=class java.lang.String,
jdbcTyp=null,resultMapId='null',jdbcTypeName='null',expression='null'}.
Cause org.apache.ibatis.type.TypeException,Error setting non null
parameter #2 with JdbcType null. Try setting a
different Jdbc Type for this parameter or a different configuration
property.Cause java.lang.ClassCastException:java.time.LocalDateTime
cannot be cast to java.lang.String
```

因为报警这一块代码，属于历史功能，如果失败并不会影响主流程。但在定位期间，如果频繁报警的话，就会造成一定的干扰。因此，我们马上采取了回滚操作，将 inf-

bom 的版本回滚至历史版本，直至报警消失，然后再进行问题的定位和分析。以下章节就是我们对报警原因的定位及原因详细分析的介绍，希望这些思路能够对大家有所启发和帮助。

报警原因定位

在回滚完毕后，我们开始具体分析报警产生的主要原因，于是进行了以下几步的排查。

第一步，查看了报警的 Mapper 方法，如下代码段所示。这个是接收返回参数，根据主键 id，更新具体响应内容和时间的代码，入参有 3 个，类型分别为 long、String 和 LocalDateTime。

```
int updateResponse(@Param("id")long id, @Param("response")String
response, @Param("updateTime")LocalDateTime updateTime);
```

第二步，我们查看了 Mapper 方法对应的 XML 文件，如下代码段所示，对应的 parameterType 类型是 String，而实际参数的类型包括 long、String 以及 LocalDateTime。

```
<update id="updateResponse" parameterType="java.lang.String">
UPDATE invoice_log
  SET response = #{response}, update_time = #{updateTime}
WHERE id = #{id}
</update>
```

第三步，我们查看了 MyBatis 上线前后的版本，报警的内容是：MyBatis 在处理 SQL 语句时，发现不能将 LocalDateTime 转型为 String，这一段逻辑在上线前是可以正常运行的，并且上线的业务逻辑对这段历史代码无改动。因此，我们猜测是因为 inf-bom 的升级，从而导致 MyBatis 的版本发生了变化，对某些历史功能不再支持了。MyBatis 版本上线前后的变化如下表所示：

inf-bom版本	MyBatis版本
1.3.9.6	3.2.3
1.4.2.1	3.4.6

表 1　MyBatis 版本升级前后对比

第四步，我们通过第三步可以得到，在这次 inf-bom 的版本升级中，MyBatis 的版本直接升了两个大版本，因此我们可以基本将原因猜测为 MyBatis 升级跨度较大，导致部分历史功能没有兼容支持，从而引起线上 SQL 的更新报错。

第五步，为了具体验证第四步的想法，我们通过 UT 的方式，将 MyBatis 的版本不断从 3.4.6 往下降，直至没有报错的位置。最终的定位是：当 MyBatis 版本为 3.2.3 时，线上代码是正常可用的，但只要升一个版本，也就是自 3.2.4 开始，就开始不兼容目前的用法。不过，我们当时的思路并不是很好，应该从小版本逐个往上升或者使用二分法，可以加速定位版本的效率。

最后，我们定位到了产生报警的根本问题。总的来说，MyBatis 版本由 inf-bom 引入而来，inf-bom 从 3.2.3 升级到了 3.4.6 版本，而 MyBatis 自 3.2.4 开始就不支持目前系统内的 SQL Mapper 的用法，因此在升级后，线上就出现了频繁报警的问题。

问题已经定位，但是还有很多事情我们需要弄清楚。为什么版本升级后就不兼容历史的用法？具体是哪一块内容不兼容？背后的原理又是什么？下文，我们会详细进行分析。

详细分析

MyBatis 升级 3.2.4 版本的官方 Release 公告

首先，从报错的原因上来看，请注意这句话：“Caused by: java.lang.ClassCastException: java.lang.LocalDateTime cannot be cast to java.lang.String.”MyBatis 在构建 SQL 语句时，发现时间字段类型 LocalDateTime 不能强制转为 String 类型。而这个 SQL 对应的 XML 配置在 3.2.3 的版本是可以正常使用的，那么我们先从 MyBatis 的 Release Log 上查看 3.2.4 版本到底发生了什么变化。

> An special remark about this feature. Previous versions ignored the “parameterType” attribute and used the actual parameter to calculate bindings. This version builds the binding information during startup and the “parameterType” attribute is used if present (though it is still optional), so in case you had a wrong value for it you will have to change it.

从官网的 Release Log 可以看到，MyBatis 在 3.2.4 以前的版本，会忽略 XML 中的 parameterType 这个属性，并且使用真实的变量类型进行值的处理。但在 3.2.4 及以后的版本中，这个属性就被启用了，如果出现类型不匹配的话，就会出现转型失败的报错。这也提示我们开发者，在升级版本时，需要检查系统内的 XML 配置，使类型进行匹配，或者不设置该属性，让 MyBatis 自行进行计算。

根据以上内容，我们可以了解到，在版本升级后，MyBatis 在构建 SQL 语句，在获取字段值时的逻辑发生了变化。接下来我们将通过一个简单的示例，来了解一下 MyBatis 在获取字段值这一块的具体代码流程是怎样的，以 3.2.3 版本为例。

以版本 3.2.3 为例，MyBatis 构建 SQL 语句过程的原理分析

我们看一下配置，首先定义一个通过主键 id 获取学生信息的方法，仿造系统内的历史代码，我们将 parameterType 定义为 java.lang.String，这和方法对应的参数 int 并不相同。

```
public StudentEntity getStudentById(@Param("id") int id);
<select id="getStudentById" parameterType="java.lang.String"
resultType="entity.StudentEntity">
SELECT id,name,age FROM student WHERE id = #{id}
</select>
```

MyBatis 框架要做的事情，就是在运行 getStudentById(2) 的时候，将 #{id} 进行替换，使 SQL 语句变成 SELECT id,name,age FROM student WHERE id = 2。MyBatis 要将 SQL 语句完整替换成带参数值的版本，需要经历框架初始化以及实际运行时动态替换这两个部分。因为 MyBatis 的代码非常多，接下来我们主要阐释和本次案例相关的内容。

在框架初始化阶段，主要包括以下流程，如下图 2 所示：



图 2　框架初始化流程

在框架初始化阶段，有一些组件会被构建，逐一做个简单的介绍：

- **SqlSession**：作为 MyBatis 工作的主要顶层 API，表示和数据库交互的会话，完成必要的数据库增删改查功能。
- **数据库增删改查功能**：负责根据用户传递的 parameterObject，动态地生成 SQL 语句，将信息封装到 BoundSql 对象中，并返回。
- **Configuration**：MyBatis 所有的配置信息都维持在 Configuration 对象之中。

接下来，我们主要关注 SqlSource，这个类会负责生成 SQL 语句，这也是本次案例中，3.2.3 和 3.2.4 差异比较大的一个地方。下面，我们会介绍一些源码。

在构建 Configuration 的过程中，会涉及到构建对应每一条 SQL 语句对应的 MappedStatement，parameterTypeClass 就是根据我们在 XML 配置中写的 parameterType 转换而来，值为 java.lang.String，在构建 SqlSource 时，传入这个参数。如下图 3 所示：

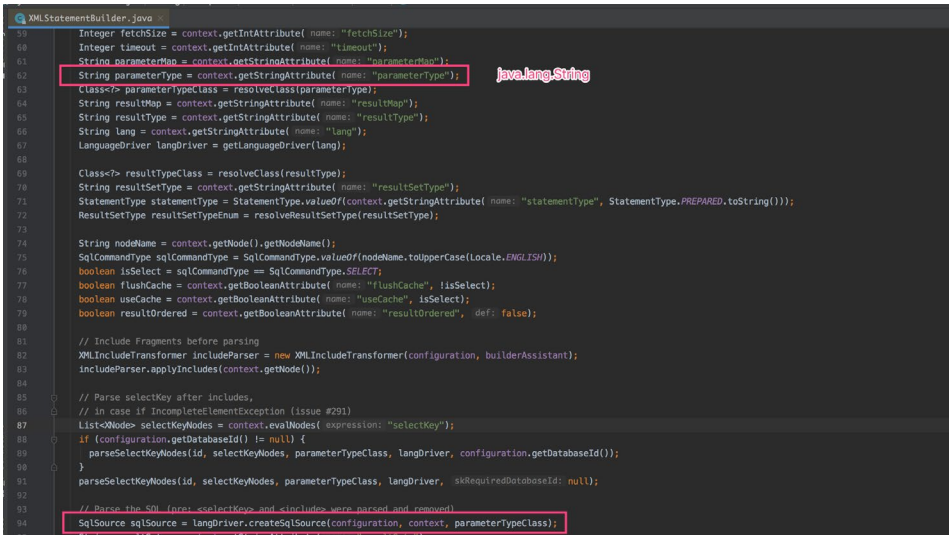


图 3　SqlSource 依赖参数

在 SqlSource 的构建中，parameterType 参数其实是被忽略不用的，并没有继续往下传递，这跟官方的描述是一致的。因为 3.2.4 之前这个 parameterType 属性被

忽略了，然后就创建了 DynamicSqlSource，这个类主要是用于处理 MyBatis 动态 SQL 的类。如下图 4 所示：

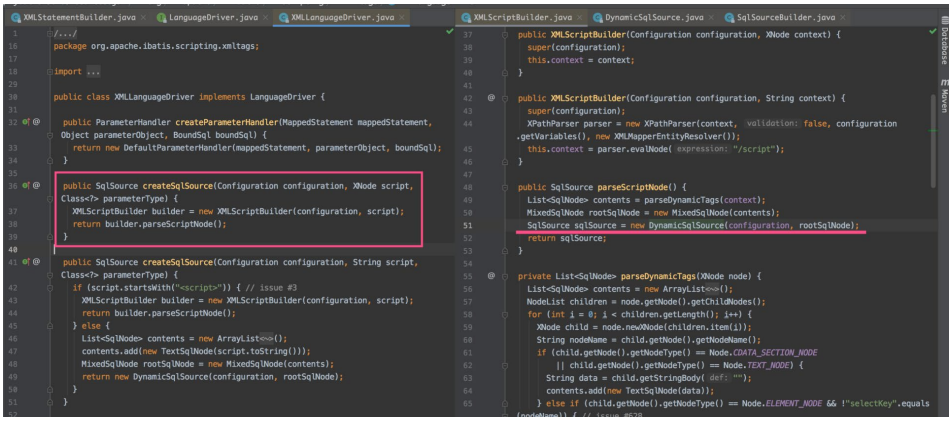


图 4　SqlSource 构建

在框架初始化的阶段，需要介绍的内容，在 3.2.3 版本已经介绍完毕。当执行 getStudentById 方法时，MyBatis 的流程如下图 5 所示。因受限于图片长度，我们对布局进行了一些调整：

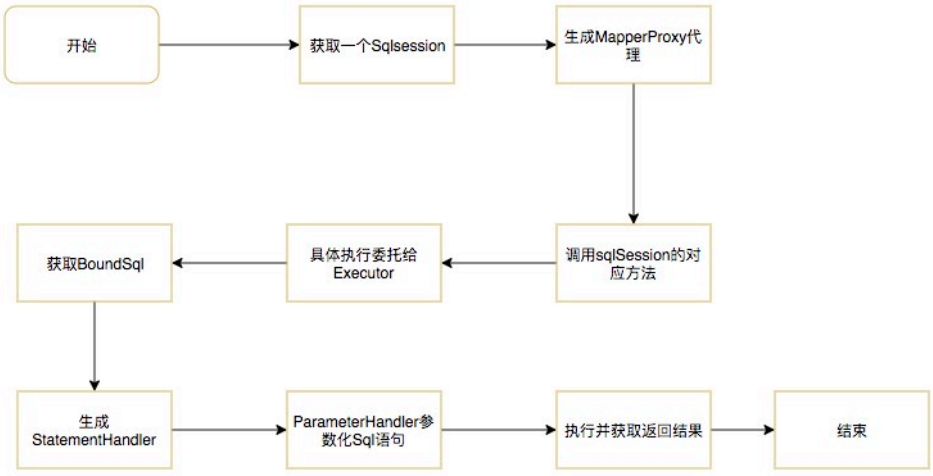


图 5　运行流程

在具体执行阶段，也涉及到一些组件，我们需要做简单的了解：

- **SqlSession**：作为 MyBatis 工作的主要顶层 API，表示和数据库交互的会话，完成必要数据库增删改查功能。
- **Executor**：MyBatis 执行器，这是 MyBatis 调度的核心，负责 SQL 语句的生成和查询缓存的维护。
- **BoundSql**：表示动态生成的 SQL 语句以及相应的参数信息。
- **StatementHandler**：封装了 JDBC Statement 操作，负责对 JDBC statement 的操作，如设置参数、将 Statement 结果集转换成 List 集合等等。
- **ParameterHandler**：负责对用户传递的参数转换成 JDBC Statement 所需要的参数。
- **TypeHandler**：负责 Java 数据类型和 JDBC 数据类型之间的映射和转换。

我们主要关注获取 BoundSql 以及参数化语句的流程，这也是 3.2.3 和 3.2.4 差异比较大的一个地方。在进入 Executor 的 Query 方法后，会首先通过对应的 MappedStatement 来获取 BoundSql，用来帮助我们动态生成 SQL 语句，里面绑定了对应的 SQL 以及参数映射关系。在构建框架阶段，我们使用的 SqlSource 是 DynamicSqlSource，通过这个类来生成获取 BoundSql，如下图 6 所示：

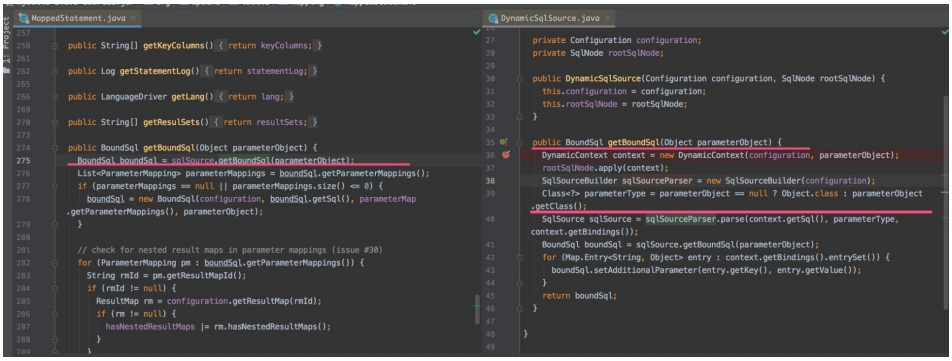


图 6　获取 BoundSql

通过图 6 的代码，我们可以得知，parameterType 在初始化阶段未被使用，而是在 SQL 执行时获取到的，但获取到的类型是 parameterObject 对应的类型，这个类是用来记录 Mapper 方法上对应的参数。如下图 7 所示，它并非在 SQL 配置文件中标注的 java.lang.String。

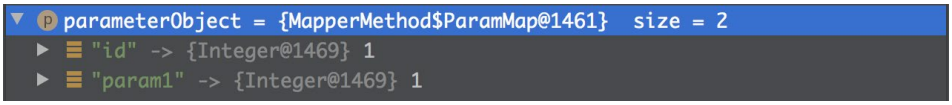


图 7　parameterObject 类型

然后我们通过 SqlSourceBuilder 的 parse 方法对 SQL 以及获取到的类型进行再次处理，其中的流程代码比较长。在这个过程中，我们主要去构建 SQL 的参数和 Java 类型的绑定关系，MyBatis 依赖这个绑定关系，使用对应的 TypeHandler 去进行值的转换。

调用链路是 `SqlSourceParser.parse -> 内部类 ParameterMappingTokenHandler.handleToken -> 私有方法 buildParameterMapping`，如下图 8 中的代码所示。因为当前的 parameterType 为 MapperMethod$ParamMap，经过了多个 if 判断，判定当前 property id 的 propertyType 为 Object.class 类型。接下来，构建 SQL 的参数和 Java 类型的绑定关系 ParameterMapping，再进行返回。

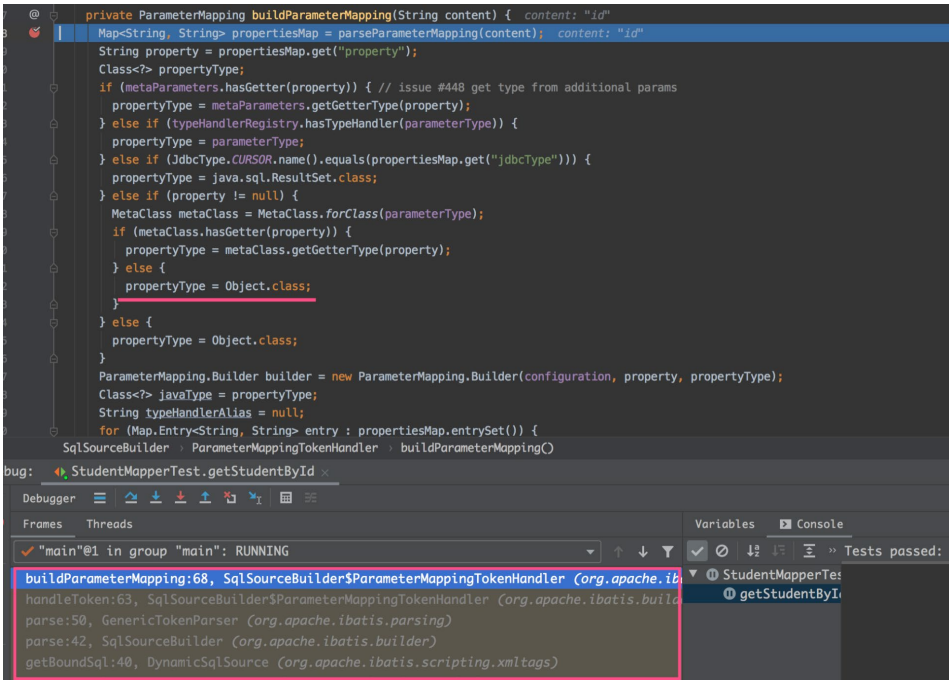


图 8　buildParameterMapping 过程

构建完成的 ParameterMapping 的结构如下图 9 中的代码所示，参数 id 对应的 javaType 类型为 java.lang.Object，对应的 TypeHander 处理器为 UnknownTypeHandler，也就是未找到合适的 TypeHandler 的兜底选项。

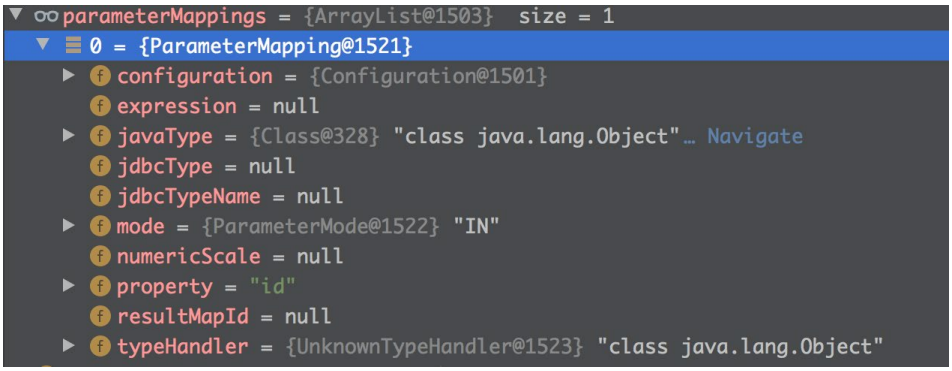


图 9　ParameterMapping 结构

接下来，流程就会流转到Executor，在`org.apache.ibatis.executor.SimpleExecutor#doQuery`进行查询时，会根据当前的SQL类型，生成对应的StatementHandler。因为我们目前都是用的预编译SQL，因此生成的statementHandler就是PreparedStatementHandler，熟悉JDBC的小伙伴应该马上可以猜到对应的语句是什么类型了。然后，我们对这句SQL语句进行填充，如下图10中的代码所示。我们会通过PreparedStatementHandler的parameterize方法对Statement进行参数化，也就是进行填充。

```
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {  handler: RoutingStatementHandler@1552
  Statement stmt;  stmt: "com.mysql.jdbc.JDBC42PreparedStatement@662b4c69: SELECT id,name,age FROM student WHERE id = ** NOT SPECIFIED **"
  Connection connection = getConnection(statementLog);  connection: "com.mysql.jdbc.JDBC4Connection@7ea9e1e2"
  stmt = handler.prepare(connection);  connection: "com.mysql.jdbc.JDBC4Connection@7ea9e1e2"
  handler.parameterize(stmt);  handler: RoutingStatementHandler@1552  stmt: "com.mysql.jdbc.JDBC42PreparedStatement@662b4c69: SELECT id,name,age FROM st
  return stmt;
}
```

图10　PrepareStatement处理过程

在PreparedStatementHandler进行参数化时，会将参数化的职责交给DefaultParameterHandler处理。如下图11中的代码所示，我们主要关注红线部分，首先会获取ParameterMapping对应的TypeHander，如前文所述，获取到的是UnknownTypeHandler，然后会通过setParameter方法，将参数id替换成对应的值。

```
public void setParameters(PreparedStatement ps) throws SQLException {
  ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());
  List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
  if (parameterMappings != null) {
    MetaObject metaObject = parameterObject == null ? null : configuration.newMetaObject(parameterObject);
    for (int i = 0; i < parameterMappings.size(); i++) {
      ParameterMapping parameterMapping = parameterMappings.get(i);
      if (parameterMapping.getMode() != ParameterMode.OUT) {
        Object value;
        String propertyName = parameterMapping.getProperty();
        if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params
          value = boundSql.getAdditionalParameter(propertyName);
        } else if (parameterObject == null) {
          value = null;
        } else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
          value = parameterObject;
        } else {
          value = metaObject == null ? null : metaObject.getValue(propertyName);
        }
        TypeHandler typeHandler = parameterMapping.getTypeHandler();
        JdbcType jdbcType = parameterMapping.getJdbcType();
        if (value == null && jdbcType == null) jdbcType = configuration.getJdbcTypeForNull();
        typeHandler.setParameter(ps,  i: i + 1, value, jdbcType);
      }
    }
  }
}
```

在 Typehandler 的流程里，首先会进入 BaseTypeHandler，然后在具体设置时，会进入子类的方法。在 UnknownTypeHandler，首先会再次对参数 parameter 进行解析，判断最正确的 TypeHandler 类型，如下图 12 中的代码所示：

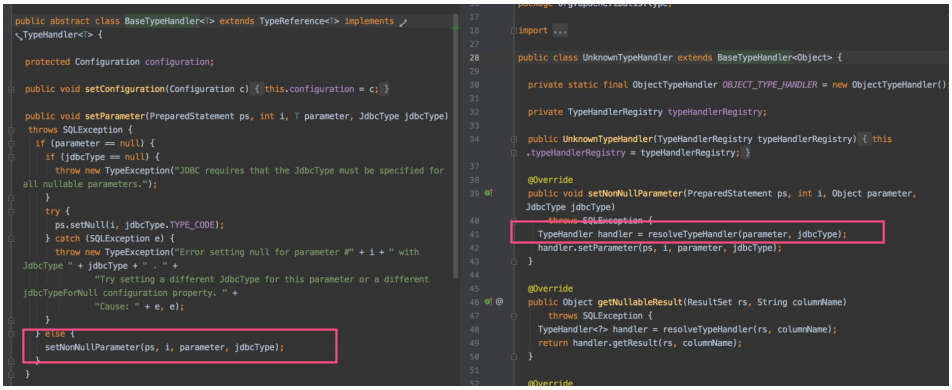


图 12　获取可用 TypeHandler

在 resolveTypeHandler 方法中，因为已知了参数值的类型，通过 Integer 这个 class 在 typeHandlerRegistry 中寻找对应的 TypeHandler，TypeHandlerRegistry 是 MyBatis 启动时内置好的，代表 Java 对象类型和 TypeHandler 的映射关系，有兴趣的同学可以进入这个类详细看下。在这个例子中，我们会直接获取到 IntegerHandler，如下图 13 中的代码所示：

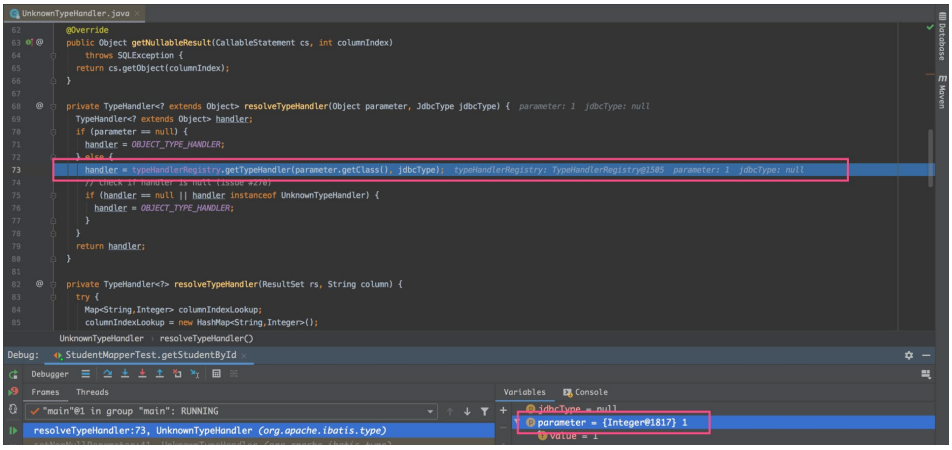


图 13　获取 IntegerHandler

在获取到 IntegerHandler 后，我们就可以使用 IntegerTypeHandler 的 setInt 方法，对 SQL 语句中的参数进行替换。如图 14 中的代码所示，SQL 语句被成功替换：



图 14　IntegerHander 值替换

后续就是执行 SQL 并处理返回结果，这就不在本文的讨论范围内了。从上文的分析中，我们可以了解到，在 3.2.3 及以下版本，MyBatis 会忽略 parameterType，在真正进行 SQL 转换时，重新根据 SQL 方法入参类型，然后计算合适的 TypeHandler 处理器，所以本案例中的代码在 3.2.3 版本时，它在运行时是正常的。

以版本 3.2.4 为例，相比版本 3.2.3，MyBatis 构建 SQL 语句过程的变化分析

在前一章节中，我们得知 MyBatis 在运行 SQL 阶段重新计算参数对应的 TypeHandler，然后进行 SQL 参数的替换。那么，在版本 3.2.4 中，MyBatis 做了什么改动，从而导致了原有的使用方式变得不可用呢？从官方的 Release Log 来看，版本 3.2.4 做了这样的一个改动。

> This version builds the binding information during startup and the “parameterType” attribute is used

这个意思是说：parameterType 会在框架初始化阶段阶段就被使用到。我们将分析的重点放在构建阶段，因为负责处理绑定关系的 BoundSql 由配置阶段的 SqlSource 生成，我们主要查看 SqlSource 的构建，在 3.2.4 中发生了什么变化。如图 15 所示，与 3.2.3 不同，3.2.4 首先判断了是否为动态 SQL，在非动态 SQL 情况下，才会将 parameterType java.lang.String 作为参数，传入 SqlSource 的构造方法。

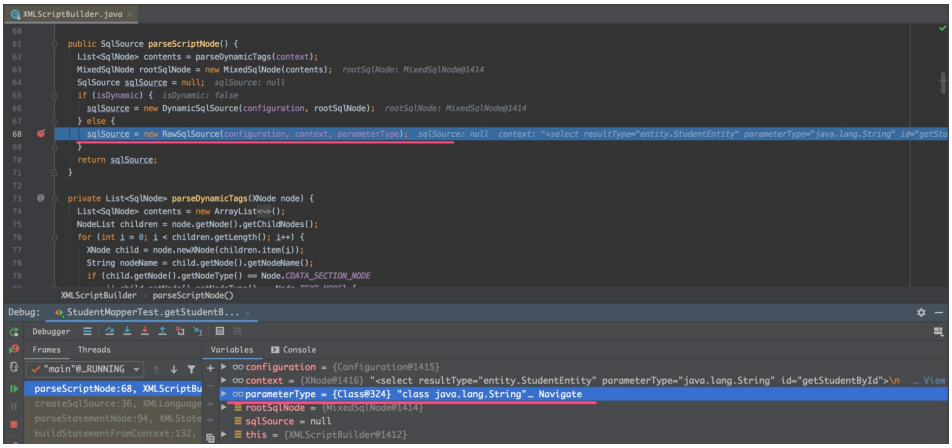


图 15　生成 SqlSource

而后续流程与 3.2.3 一致，因为 parameter 类型为 java.lang.String，在构建 parameterMapping 时，使用的类型就是 java.lang.String。

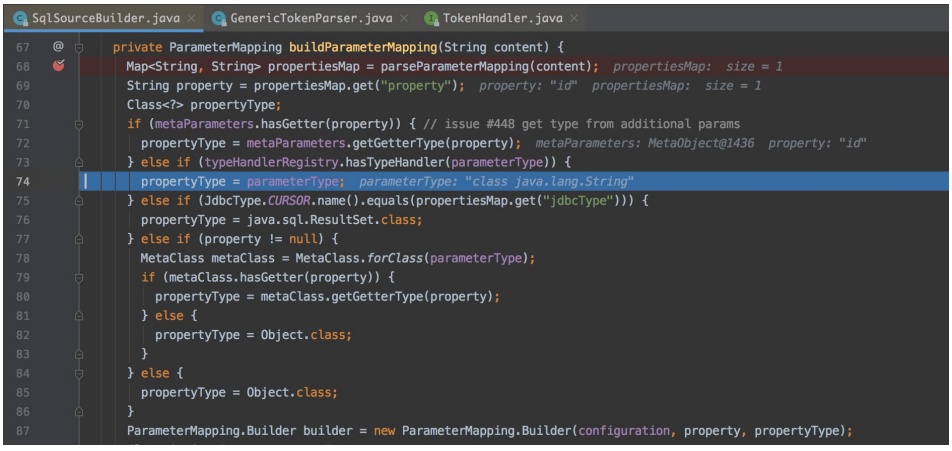


图 16　构建 ParameterMapping 与 3.2.3 版本的差异

因为在框架初始化阶段，SqlSource 的 ParameterMapping 中 id 对应的类型就是 java.lang.String，这就导致在进行 SQL 语句的替换时，获取到的 TypeHandler 是 StringTypeHandler，如下图 17 所示：

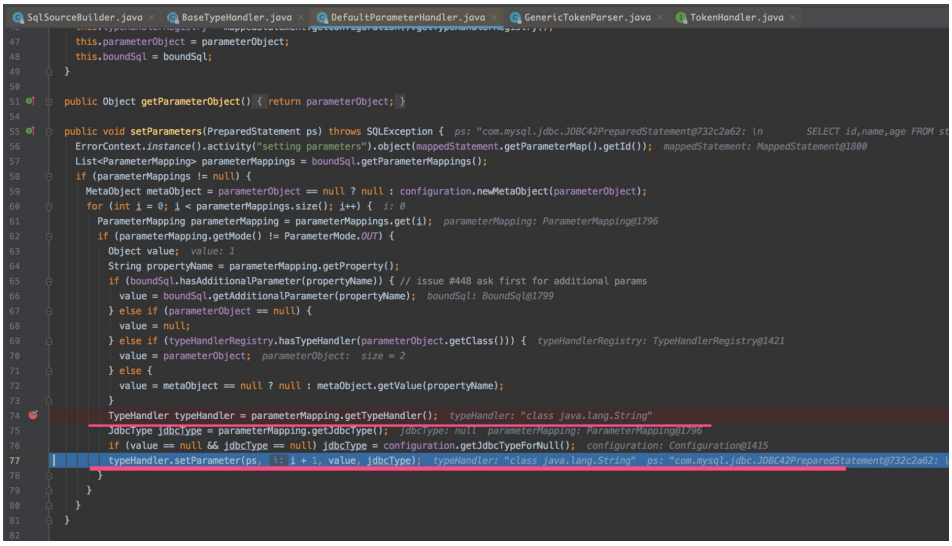


图 17　整数类型的参数获取到了 StringTypeHandler

后面的报错原因就比较好理解了，在调用 StringTypeHandler 的 setString 方法时，报出了 `java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String` 的错误。

总结

我们总结一下这个案例因：

MyBatis 3.2.3 版本支持 parameterType 和实际参数类型不匹配，在执行 SQL 阶段，动态计算值处理器类型。在大版本升级 2 个版本号后，parameterType 实际的类型开始生效，使用对应这个类型的 TypeHandler 对 SQL 进行参数替换，会导致 Mapper 方法中的参数和 XML 中的 parameterType 不匹配时，进而会出现类型转换报错。

这一段排查的经历，对自己后续编写代码及在系统上线时也有一些启发，主要包括以下几个方面：

- 在 inf-bom 升级时，需要线下进行全面回归，要避免框架存在不兼容的用法，不然的话，就容易导致线上错误。
- 开发同学可以检查自己系统内的 MyBatis 版本，如果是 3.2.4 以下，需要全面检查下现在的 Mapper 文件里对于 parameterType 的使用和 Mapper 方法中实际的参数类型是否一致，避免升级到 3.2.4 及以上版本时发生转型报错。如果有不匹配的情况存在，需要进行修正或者不使用 parameterType，让 MyBatis 在运行 SQL 时自动计算对应的类型。
- 可以考虑使用 MyBatis-Generator 来自动生成 XML 和 Mapper 文件，毕竟是专业团队在维护，稳定性相对来说会更好一些，同时能够避免手动修改 XML 文件带来的误操作。
- 可以主动关注强依赖的一些开源框架的 Release Log，不要错过了重要的信息。

参考资料

带你一步一步手撕 MyBatis 源码加手绘流程图——构建部分
带你一步一步手撕 MyBatis 源码加手绘流程图——执行部分
MyBatis 源码解析（三）一缓存篇
面试官问你 MyBatis SQL 是如何执行的？把这篇文章甩给他
源码分析 (1.4 万字) | MyBatis 接口没有实现类为什么可以执行增删改查
MyBatis/MyBatis-3/Comparing changes

作者简介

凯伦，2016 年校招加入美团，后端开发工程师。

招聘信息

电子发票技术团队负责美团发票平台建设和相关创新探索工作，我们的宗旨是提升美团用户在美团各业务场景下的开票体验，同时赋能商家提升发票管理效率。欢迎志同道合的小伙伴加入，通过技术的力量为亿万用户提升电子发票服务体验。感兴趣的同学可投递简历至：tech@meituan.com（邮件标题注明：电子发票业务）

复杂环境下落地 Service Mesh 的挑战与实践

作者：继东　薛晨　业祥　张昀

导读

在私有云集群环境下建设 Service Mesh ，往往需要对现有技术架构做较大范围的改造，同时会面临诸如兼容困难、规模化支撑技术挑战大、推广困境多等一系列复杂性问题。本文会系统性地讲解在美团在落地 Service Mesh 过程中，我们面临的一些挑战及实践经验，希望能对大家有所启发或者帮助。

一、美团服务治理建设进展

1.1 服务治理发展史

首先讲一下 OCTO，此前美团技术团队公众号也分享过很相关的文章，它是美团标准化的服务治理基础设施，现应用于美团所有事业线。OCTO 的治理生态非常丰富，性能及易用性表现也很优异，可整体概括为 3 个特征：

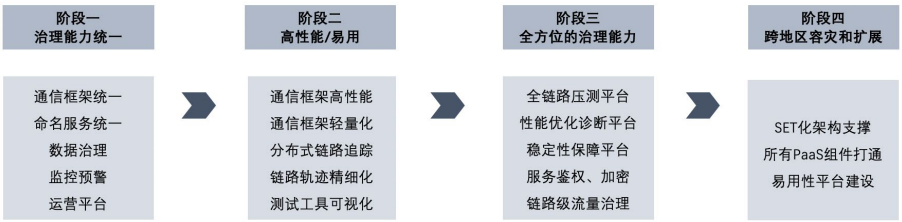
1. 属于公司级的标准化基础设施。技术栈高度统一，覆盖了公司 90% 以上的应用，日均调用量达数万亿次。
2. 经历过较大规模的技术考验。覆盖数万个服务、数十万个节点。
3. 治理能力丰富。协同周边治理生态，实现了 SET 化、链路级复杂路由、全链路压测、鉴权加密、限流熔断等治理能力。

回顾美团服务治理体系的发展史，历程整体上划分为四个阶段：

1. **第一阶段是基础治理能力统一**。实现通信框架及注册中心的统一，由统一的治理平台支撑节点管理、流量管理、监控预警等运营能力。
2. **第二阶段重点提升性能及易用性**。4 核 4GB 环境下使用 1KB 数据进行 echo 测试，QPS 从 2 万提升至接近 10 万，99 分位线 1ms；也建设了分布式链

路追踪、分阶段耗时精细埋点等功能。

3. **第三阶段是全方位丰富治理能力**。落地了全链路压测平台、性能诊断优化平台、稳定性保障平台、鉴权加密等一系列平台，也实现了链路级别的流量治理，如全链路灰度发布等。
4. **第四阶段是建设了跨地域的容灾及扩展能力**。在每天数千万订单量级下实现了单元化，也实现了所有 PaaS 层组件及核心存储系统的打通。



1.2 服务治理体系的困境

目前，美团已具备了较完善的治理体系，但仍有较多的痛点及挑战。大的背景是公司业务蓬勃发展，业务愈发多元化，治理也愈发精细化，这带来了较多新的问题：

1. 业务与中间件强耦合，制约彼此迭代。当中间件引入 Bug，可能成百上千、甚至数千个业务需要做配合升级，中间件的新特性也依赖业务升级后才能使用，成本很高。
2. 中间件版本碎片化严重。发布出去的组件基本托管在业务侧，很难统一进行管控，这也频繁造成业务多类的问题。
3. 异构体系融合难。新融入公司的技术体系往往与美团不兼容，治理体系打通的成本很高，难度也很大。此前，美团与大众点评打通治理，不包含业务迁移，就历时 1 年半的时间；近期，摩拜使用的 gRPC 框架也无法与系统进行通信，但打通迫在眉睫。
4. 非 Java 语言治理体系能力弱，多个主流语言无官方 SDK。多元业务场景下，未来多语言也是个趋势，比如在机器学习领域，Python 语言不太可能被其他语言完全代替。

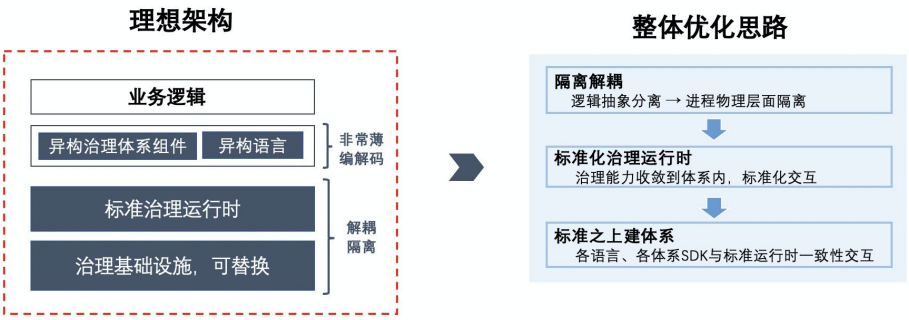
二、服务治理体系优化的思路与挑战

2.1 优化思路

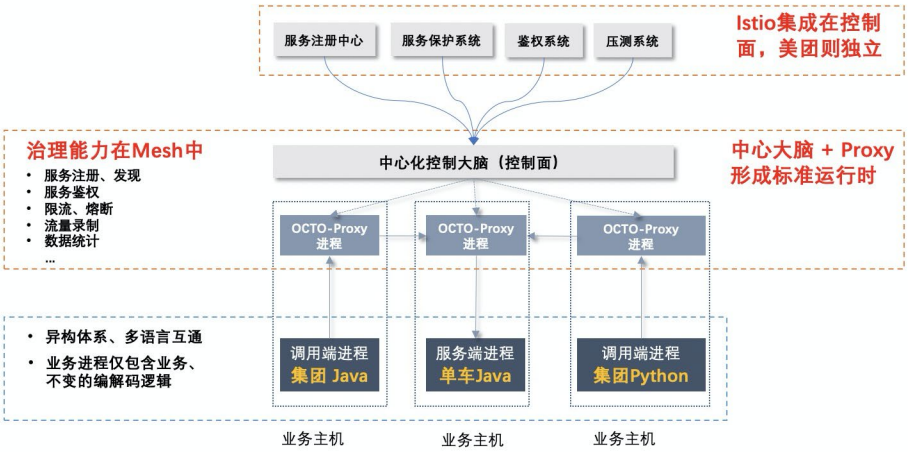
总结来看，OCTO 在服务层实现了统一抽象来支撑业务发展，但它并未解决这层架构可以独立演进的问题。

1.2 节中问题 1 与问题 2 的本质是“耦合”，问题 3 与问题 4 的本质是“缺乏标准服务治理运行时”。在理想的架构中，异构语言、异构治理体系可以共用统一的标准治理运行时，仅在业务使用的 SDK 部分有轻微差异。

所以，我们整体的优化思路分为三步：**隔离解耦，在隔离出的基础设施层建设标准化治理运行时，标准之上建体系**。



上述解决方案所对应的新架构模式下，各业务进程会附属一个 Proxy 进程，SDK 发出以及接收的流量均会被附属的 Proxy 拦截。像限流、路由等治理功能均由 Proxy 和中心化的控制大脑完成，并由控制面对接所有治理子系统集成。这种模式下 SDK 很轻薄，异构的语言、异构的治理体系就很容易互通，从而实现了物理解耦，业界将这种模式称为 Service Mesh（其中 Proxy 被称为数据面、中心化控制大脑被称为控制面）。



2.2 复杂性挑战

美团在实践中所面临的复杂性划主要包括以下 4 类：

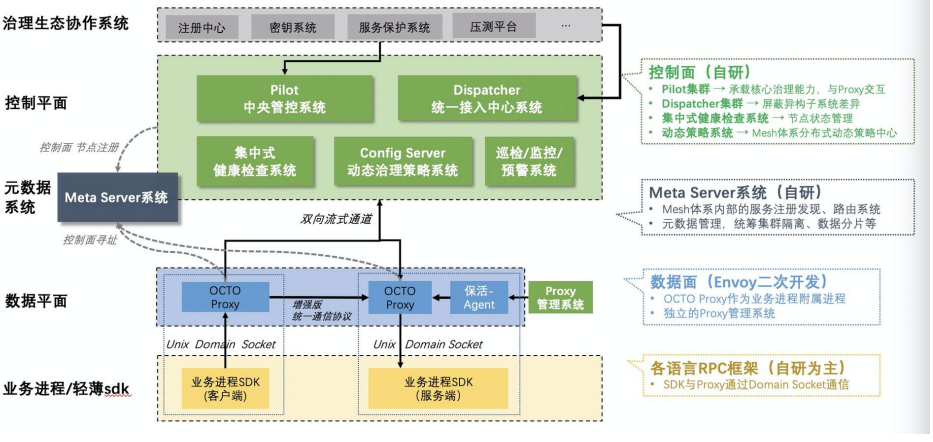
1. **兼容性**：技术改造涉及范围较大，一方面需要通过保证现有通信方式及平台使用方式不变，从而来保障业务研发效率，另一方面也要解决运行载体多样性、运维体系兼容等问题。
2. **异构性**：第一是多语言互通问题；第二是打通治理体系内的众多治理子系统，像服务鉴权、注册中心等系统的存储及发布订阅机制都是不同的；第三是快速打通新融入公司的异构治理体系。
3. **大规模支撑**：出于性能方面考虑，开源 Istio 等产品不宜直接应用于大规模的生产环境，美团控制面需具备百万级链接下高吞吐、低延迟、高精确的系统能力。
4. **重交易型业务容错性低**：交易型业务场景下，业务对 Service Mesh 的性能、稳定性往往持怀疑态度；美团基础架构团队也强调在业务价值导向下，基于实际业务价值进行运营推广，而不是采用从上至下的偏政策性推广方式。

三、美团落地 Service Mesh 的解决方案

3.1 整体架构

美团采用数据面基于 Envoy 二次开发、控制面自研为主、SDK 协同升级的方案（内部项目名是 OCTO Mesh ）。架构简介如下：

- 各语言轻薄的 SDK 与 Proxy 通过 UDS（Unix Domain Socket）交互，主要出发点是考虑到相比透明流量劫持，UDS 性能与可运维性更好。
- 控制面与 Proxy 通过双向流通信，控制面与治理生态的多个子系统交互，并将计算处理过的治理数据及策略数据下发给 Proxy 执行，协同配合完成路由、限流等所有核心治理功能。
- 控制面内部的 5 个模块都是自研的独立服务。
  - Pilot 承载核心治理能力，与 Proxy 直接交互。
  - Dispatcher 负责屏蔽异构子系统差异。
  - 集中式健康检查管理节点状态。
  - Config Server 管理 Mesh 体系内相关的策略，并将 Pilot 有状态的部分尽量迁移出来。
  - 监控及巡检系统负责提升稳定性。
- 自研了的 Meta Server 系统实现 Mesh 体系内部的节点注册和寻址，通过管理控制面与数据面的链接关系，也实现了按事业群隔离、水平扩展等能力。



3.2 兼容性解决方案

兼容性的目标及特征用一句话来总结就是：业务接入无感知。为此，我们做了以下三件事情：

(1) 与现有基础设施及治理体系兼容

- 将 Service Mesh 与 OCTO 深度打通，确保各治理子系统的使用方式都不变。
- 运行载体方面，同时支持容器、虚拟机、物理机。
- 打通运维体系，保证服务治理基础设施处于可管理、可监测的状态。

(2) 协议兼容

- 服务间调用往往是多对多的关系，一般调用端与服务端无法同时升级，为支持 Mesh 与非 Mesh 的互通，增强后的协议对业务完全透明。
- 与语义相关的所有内容（比如异常等），均在 SDK 与 Proxy 之间达成共识，保证兼容。
- 无法在控制面及数据面中实现的能力，在 SDK 中执行并通过上下文传递给 Proxy，保障功能完全对齐，当然这种情况应该尽量避免的。

(3) Mesh 与非 Mesh 模式的无缝切换

- 基于 UDS 通信必然需要业务升级一次 SDK 版本，我们在 2020 年初时预先发布早做部署，确保当前大部分业务已经升级到新版本，但默认仍是不开启 Mesh 的状态。
- 在可视化平台上面通过开关操作，几乎无成本实现从 Mesh 模式与非 Mesh 模式的切换，并具备实时生效的能力。

3.3 异构性解决方案

异构性的目标及特征用一句话总结就是：标准化服务治理运行时。具体可拆分为 3 个子目标：

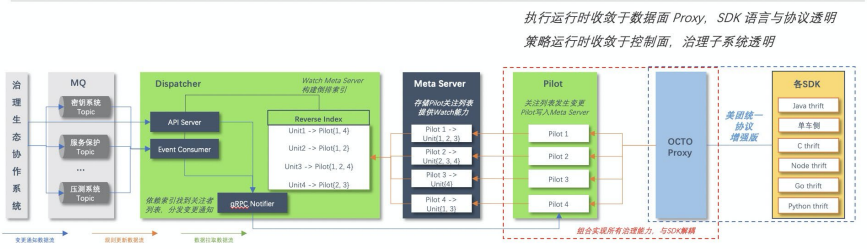
- 标准化美团内部 6 种语言的治理体系。
- 架构层面由控制面统一对接各个治理子系统，屏蔽注册中心、鉴权、限流等系

统具体实现机制的异构性。

- 支持摩拜及未来新融入公司的异构治理体系与公司整体的快速融合。

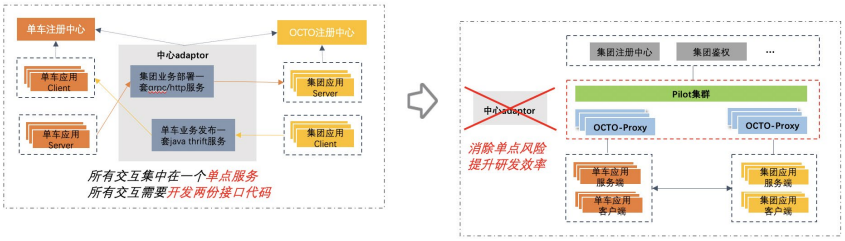
针对上述 3 个子目标，我们所采取的方案如下：

- 将数据面 + 控制面定义为标准化的服务治理运行时，在标准运行时内打通所有治理能力。
- 建设统一接入中心系统 Dispatcher ，并由其对接并屏蔽治理子系统的异构性，从而实现外部系统的差异对 Pilot 透明；下图中 Dispatcher 与 Pilot 直接交互，Meta Server 的作用是避免广播降低冗余。
- 重构或从零建设 SDK，目前使用的 6 种语言 SDK 均已落地并使用。
- 异构语言、异构体系均使用增强的统一协议交互，实现互通。



通过 Service Mesh 实现体系融合的前后对比如下：

- 引入 Service Mesh 前，单车向公司的流量以及公司向单车的流量，均是由中间的 adaptor 单点服务承接。除稳定性有较大隐患外，所有交互逻辑均需要开发两份代码，效率较差。
- 引入 Service Mesh 后，在一套服务治理设施内打通并直接交互，消除了中心 adaptor 带来的稳定性及研发效率方面的缺陷；此外整个打通在 1 个月内完成，异构体系融合效率很高。



通过上述方案，针对异构性方面取得了较好的效果：

- 标准化 6 种语言治理体系，非 Java 语言的核心治理能力基本对齐 Java；新语言也很容易融入，提供的官方 Python 语言、Golang 语言的通信框架新版本（依托于 OCTO Mesh），开发成本均控制在 1 个月左右。
- 支持异构治理子系统通过统一接入中心快速融入，架构简洁、扩展性强。
- 支持异构治理体系快速融合并在单车侧落地，异构治理体系打通成本也从 1.5 年降低到 1 个月。

3.4 规模化解决方案

3.4.1 开源 Istio 问题分析

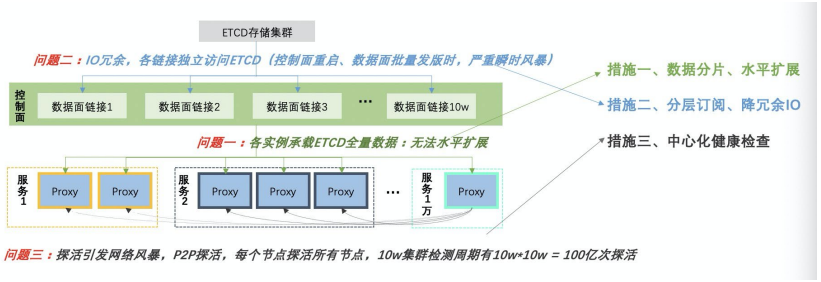
规模化的目标及特征用一句话总结是：**具备支撑数万服务、百万节点体量的系统能力，并支持水平扩展**。挑战主要有 3 个：

- 美团体量是最流行开源产品 Istio 上限的上千倍。
- 极高的实时性、准确性要求；配置下发错误或丢失会直接引发流量异常。
- 起步较早，业界参考信息很少。

经过对 Istio 架构进行深入分析，我们发现核心问题聚焦在以下 3 个瓶颈点：

- 每个控制面实例有 ETCD 存储系统的全部数据，无法水平扩展。
- 每个 Proxy 链接相当于独立与 ETCD 交互，而同一个服务的 Proxy 请求内容都相同，独立交互有大量的 I/O 冗余。当 Proxy 批量发版或网络抖动时，瞬时风暴很容易压垮控制面及 ETCD。

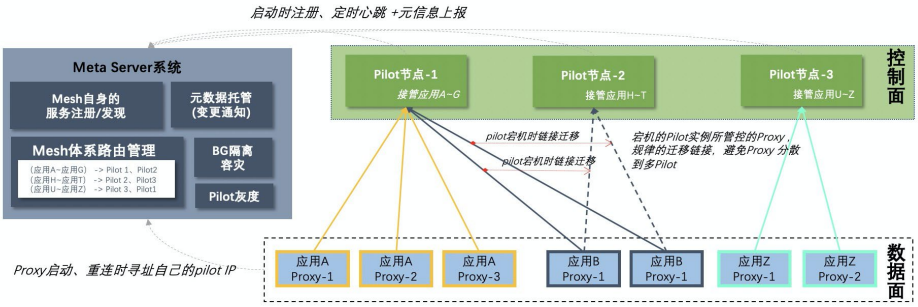
- 每个节点都会探活所有其他节点。10 万节点规模的集群，1 个检测周期有 100 亿次探活，会引发网络风暴。



3.4.2 措施一：横向数据分片

针对 Istio 控制面各实例承载全集群数据的问题，对应的措施是通过横向逻辑数据分片支持扩展性，具体方案设计如下：

- Proxy 启动时会去向 Meta Server 系统请求需要连接的 Pilot IP，Meta Server 将相同服务的 Proxy 尽量落到同一个控制面节点（内部策略更为复杂，还要考虑地域、负载等情况），这样每个 Pilot 实例按需加载而不必承载所有数据。
- 控制面节点异常或发布更新时，其所管理的 Proxy 也会有规律的迁移，恢复后在一定时间后还会接管其负责的 Proxy，从而实现了会话粘滞，也就实现逻辑上面的数据分片。

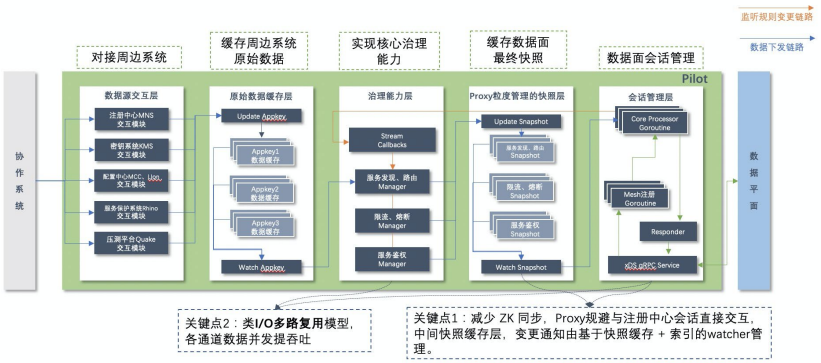


通过管理链接关系实现了按事业群隔离、按服务灰度等平台能力，最关键的还是解决了 Mesh 体系水平扩展的问题。

3.4.3 措施二：纵向分层订阅

针对 Istio 独立管理各 Proxy 链接的 I/O 冗余问题，对应的措施是通过分层订阅减少冗余 I/O。Proxy 不直接与存储等系统对接，而是在中间经过一系列的处理，关键点有两个：

- 关键点 1：基于快照缓存 + 索引的机制来减少 ZK watcher 同步。以注册中心为例，常规实现方式下，如果每个 Proxy 关注 100 个节点，1 万个节点就会注册 100 万个 watcher，相同服务的 Proxy 所关注内容是相同的，另外不同服务 Proxy 所关注的也有很多交集，其中包含大量的冗余。分层订阅模式下，Proxy 不与注册中心直接交互，通过中间的快照缓存与分层，确保每个 Pilot 实例中 ZK 相同路径的监听最多只用 1 个 watcher，获取到 watcher 通知后，Pilot 根据内部的快照缓存 + 索引向所有关注者分发，大大降低了冗余。
- 关键点 2：治理能力层及会话管理层实现了类似于 I/O 多路复用能力，通过并发提升吞吐。

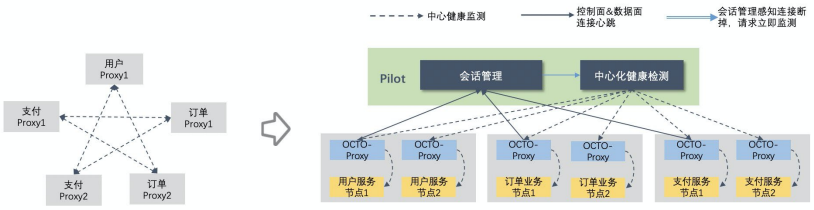


结果方面有效应对了网络抖动或批量发版的瞬间风暴压力，压测单 Pilot 实例可以承载 6 万以上的链接，时延 TP99 线 < 2.3ms、数据零丢失。

3.4.4 措施三：集中式健康检测

针对大规模集群内指数级膨胀的节点间健康监测次数，对应的措施是摒弃了 P2P 检测模式，我们参考并优化了 Google 的 Traffic Drector 中心化管理的健康检测模式。这种模式下检测次数大大减少，一个周期内 10 万节点集群的检测次数，从 100 亿次下降到 10 万次。

此外，当 Pilot 感知到 Proxy 异常时，会立即通知中心化健康检测系统启动检测，而不是等待检测周期窗口的到来，这可以有效提升业务调用的成功率。

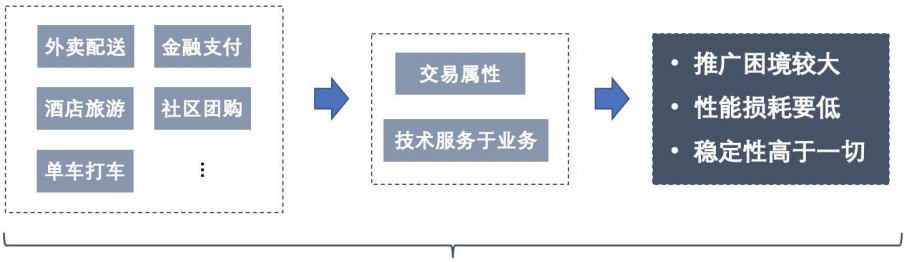


3.5 交易型场景困境下的解决方案

3.5.1 业务属性分析

美团内部业务线较多，包括外卖、配送、酒店、旅游、单车、团购等，其中绝大多数业务都带有交易属性，交易链路上一个流量异常就可能影响到订单。业务系统对新技术领域的探索往往比较慎重，期望在新技术充分验证后再启动试点，所以除小语种及亟待与公司打通的单车业务外，推广的难度是非常大的。此外，基础架构部秉承“以客户为中心”的原则，研发、运维、测试人员均是我们的“客户”，所以技术升级会重点从业务价值入手，并非简单依靠从上至下的政策推动力。

所以，我们对外的承诺是：**通信足够快、系统足够稳定、接入足够平滑高效**。



通信足够快、系统足够稳、接入足够平滑高效！

3.5.2 精细化运营体系建设

针对推广的困境，我们首先做了两件事情：

- 寻找具备强诉求的业务试点，客观来说，美团技术栈内这类业务数量非常有限。
- 寻求标杆核心业务试点，充分验证后推广给其他业务，但效果并不理想，与业务稳定性的诉求并不匹配。

针对上述困境，我们进行深度思考后建立了一个精细化的运营体系：

- 服务接入 Mesh 前。基于 SOA 分级将服务划分为非核心与核心两类，先针对非核心服务以及所有服务的线下环境进行重点突破，实现了在广泛的业务场景下，全面且充分的验证系统能力。
- 服务接入 Mesh 中。运营系统通过校验 SDK 版本、运行时环境等信息，自动筛选出满足条件的服务，业务同学只需要在平台上做（1）开启开关、（2）选择节点（3）指定 Mesh 流量比例三个步骤，就完成了到 Mesh 模式的切换，不需代码改造也不需发布服务，整个过程基本在 1 分钟左右完成；此外，通过与 IM 工具深度联动，提升了推广与数据运营的效率。
- 服务接入 Mesh 后。一方面，业务侧包括架构侧的运营有详细的数据指标做对比参考；另一方面，运营系统支持预先设置稳定性策略并做准实时的检测，当某个接入服务 Mesh 模式异常时，即时自动切换回非 Mesh 模式。

运营体系具备“接入过程无感”、“精细化流量粒度灰度”、“异常自动回滚恢复”三个核心能力，在运营体系建设后推广运营较为顺利，目前线上接入的 600+ 服务、线下接入的 3500+ 服务中，90% 以上是依托运营平台接入 Mesh 的。

3.5.3 通信性能优化

在性能损耗优化这个方向，除使用 UDS 规避网络栈外，我们也通过增量聚合下发、序列化优化两个措施减少不必要的解包，提升了通信性能。

经过压测，去除非核心功能在 2 核 4G 环境用 1KB 数据做 echo 测试，QPS 在 34000 以上，一跳平均延迟 0.207ms，时延 TP99 线 0.4ms 左右。

3.5.4 流量多级保护

美团落地 Service Mesh 在稳定性保障方面建设投入较多，目前尚无 Service Mesh 引发的故障，具体包含三个方面：

- 首先做了流量多级保护
  - 一方面，当 Proxy 不可用时，流量会自动 fallback 到非 Mesh 模式；另一方面，支持最精细支持按单节点的 1/1000 比例灰度。下图是具体的交互流程，当然，这两个特性与 Service Mesh 的最终形态是冲突的，只是作为系统建设初期优先保证业务稳定性的过渡性方案，长期来看必然是要去除的（包括美团一些核心服务已经完全去除）。
  - 基于 FD 迁移 + SDK 配合协议交互，实现 Proxy 无损热重启的能力。
- 控制面下发错误配置比停发配置的后果更为严重，我们建设了应用层面及系统层面的周期巡检，从端到端的应用视角验证正确性，避免或减少因变更引发的异常。
- 系统交互方面，通过限流、熔断对中心化控制面做服务保护；系统内柔性可用，当控制面全部异常时，缓存机制也能协助 Proxy 在一定时间内可用。

四、总结

本文系统性的介绍美团在 Service Mesh 落地进程中面临的“兼容性”、“异构性”、“规模化”、“交易属性业务容错性低”这四类复杂性挑战，针对上述挑战，我们也详细介绍了大规模私有云集群场景下的优化思考及实践方案。

基于上述实践，目前美团线上落地服务数超过 600，线下服务数超过 3500+，初步验证了模式的可行性。短期价值方面，我们支持了摩拜等异构治理体系的快速融合、多语言治理能力的统一；长期价值仍需在实践中继续探索与验证，但在标准化服务治理运行时并与业务解耦、中心化管控下更丰富的治理能力输出两个方面，是非常值得期待的。

作者简介

继东、薛晨、业祥、张昀，均来自美团基础技术部 – 基础架构部。

招聘信息

基础技术部 – 基础架构部 – 中间件研发中心 – 服务框架组涉及领域主要包括 RPC 通信框架、Service Mesh、服务治理门户、Set 化、流程引擎系统 Gravity 等。感兴趣的同学可投递简历至：tech@meituan.com（邮件主题请注明：基础架构部）。

C++ 服务编译耗时优化原理及实践

作者：周磊

一、背景

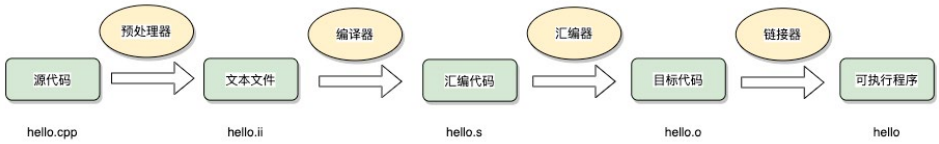
大型 C++ 工程项目，都会面临编译耗时较长的问题。不管是开发调试迭代、准入测试，亦或是持续集成阶段，编译行为无处不在，降低编译时间对提高研发效率来说具有非常重要意义。

美团搜索与 NLP 部为公司提供基础的搜索平台服务，出于性能的考虑，底层的基础服务通过 C++ 语言实现，其中我们负责的深度查询理解服务（DeepQueryUnderstanding，下文简称 DQU）也面临着编译耗时较长这个问题，整个服务代码在优化前编译时间需要二十分钟左右（32 核机器并行编译），已经影响到了团队开发迭代的效率。基于这样的背景，我们针对 DQU 服务的编译问题进行了专项优化。在这个过程中，我们也积累了一些优化的知识和经验，在这里分享给大家。

二、编译原理及分析

2.1 编译原理介绍

为了更好地理解编译优化方案，在介绍优化方案之前，我们先简单介绍一下编译原理，通常我们在进行 C++ 开发时，编译的过程主要包含下面四个步骤：



预处理器：宏定义替换，头文件展开，条件编译展开，删除注释。

- gcc -E 选项可以得到预处理后的结果，扩展名为 .i 或 .ii。

- C/C++ 预处理不做任何语法检查，不仅是因为它不具备语法检查功能，也因为预处理命令不属于 C/C++ 语句（这也是定义宏时不要加分号的原因），语法检查是编译器要做的事情。
- 预处理之后，得到的仅仅是真正的源代码。

编译器：生成汇编代码，得到汇编语言程序（把高级语言翻译为机器语言），该种语言程序中的每条语句都以一种标准的文本格式确切的描述了一条低级机器语言指令。

- gcc -S 选项可以得到编译后的汇编代码文件，扩展名为 .s。
- 汇编语言为不同高级语言的不同编译器提供了通用的输出语言。

汇编器：生成目标文件。

- gcc -c 选项可以得到汇编后的结果文件，扩展名为 .o。
- .o 文件，是按照的二进制编码方式生成的文件。

链接器：生成可执行文件或库文件。

- **静态库**：指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了，其后缀名一般为“.a”。
- **动态库**：在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可执行文件比较小，动态库一般后缀名为“.so”。
- **可执行文件**：将所有的二进制文件链接起来融合成一个可执行程序，不管这些文件是目标二进制文件还是库二进制文件。

2.2 C++ 编译特点

（1）每个源文件独立编译

C/C++ 的编译系统和其他高级语言存在很大的差异，其他高级语言中，编译单元是整个 Module，即 Module 下所有源码，会在同一个编译任务中执行。而在 C/C++ 中，编译单元是以文件为单位。每个 .c/.cc/.cxx/.cpp 源文件是一个独立的编译单

元，导致编译优化时只能基于本文件内容进行优化，很难跨编译单元提供代码优化。

（2）每个编译单元，都需要独立解析所有包含的头文件

如果 N 个源文件引用到了同一个头文件，则这个头文件需要解析 N 次（对于 Thrift 文件或者 Boost 头文件这类动辄几千上万行的头文件来说，简直就是“鬼故事”）。

如果头文件中有模板（STL/Boost），则该模板在每个 cpp 文件中使用时都会做一次实例化，N 个源文件中的 std::vector 会实例化 N 次。

（3）模板函数实例化

在 C++ 98 语言标准中，对于源代码中出现的每一处模板实例化，编译器都需要去做实例化的工作；而在链接时，链接器还需要移除重复的实例化代码。显然编译器遇到一个模板定义时，每次都去进行重复的实例化工作，进行重复的编译工作。此时，如果能够让编译器避免此类重复的实例化工作，那么可以大大提高编译器的工作效率。在 C++ 0x 标准中一个新的语言特性 – 外部模板的引入解决了这个问题。

在 C++ 98 中，已经有一个叫做显式实例化（Explicit Instantiation）的语言特性，它的目的是指示编译器立即进行模板实例化操作（即强制实例化）。而外部模板语法就是在显式实例化指令的语法基础上进行修改得到的，通过在显式实例化指令前添加前缀 extern，从而得到外部模板的语法。

① 显式实例化语法：template class vector。 ② 外部模板语法：extern template class vector。

一旦在一个编译单元中使用了外部模板声明，那么编译器在编译该编译单元时，会跳过与该外部模板声明匹配的模板实例化。

（4）虚函数

编译器处理虚函数的方法是：给每个对象添加一个指针，存放了指向虚函数表的地址，虚函数表存储了该类（包括继承自基类）的虚函数地址。如果派生类重写了虚函数的新定义，该虚函数表将保存新函数的地址，如果派生类没有重新定义虚函数，该

虚函数表将保存函数原始版本的地址。如果派生类定义了新的虚函数，则该函数的地址将被添加到虚函数表中。

调用虚函数时，程序将查看存储在对象中的虚函数表地址，转向相应的虚函数表，使用类声明中定义的第几个虚函数，程序就使用数组的第几个函数地址，并执行该函数。

使用虚函数后的变化：

① 对象将增加一个存储地址的空间（32 位系统为 4 字节，64 位为 8 字节）。 ② 每个类编译器都创建一个虚函数地址表。 ③ 对每个函数调用都需要增加在表中查找地址的操作。

（5）编译优化

GCC 提供了为了满足用户不同程度的的优化需要，提供了近百种优化选项，用来对编译时间，目标文件长度，执行效率这个三维模型进行不同的取舍和平衡。优化的方法不一而足，总体上将有以下几类：

① 精简操作指令。 ② 尽量满足 CPU 的流水操作。 ③ 通过对程序行为地猜测，重新调整代码的执行顺序。 ④ 充分使用寄存器。 ⑤ 对简单的调用进行展开等等。

如果全部了解这些编译选项，对代码针对性的优化还是一项复杂的工作，幸运的是 GCC 提供了从 O0-O3 以及 Os 这几种不同的优化级别供大家选择，在这些选项中，包含了大部分有效的编译优化选项，并且可以在这个基础上，对某些选项进行屏蔽或添加，从而大大降低了使用的难度。

- O0：不做任何优化，这是默认的编译选项。
- O 和 O1：对程序做部分编译优化，编译器会尝试减小生成代码的尺寸，以及缩短执行时间，但并不执行需要占用大量编译时间的优化。
- O2：是比 O1 更高级的选项，进行更多的优化。GCC 将执行几乎所有的不包含时间和空间折中的优化。当设置 O2 选项时，编译器并不进行循环展开以及函数内联优化。与 O1 比较而言，O2 优化增加了编译时间的基础上，提高了

生成代码的执行效率。

- O3：在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。
- Os：主要是对代码大小的优化，通常各种优化都会打乱程序的结构，让调试工作变得无从着手。并且会打乱执行顺序，依赖内存操作顺序的程序需要做相关处理才能确保程序的正确性。

编译优化有可能带来的问题：

① **调试问题**：正如上面所提到的，任何级别的优化都将带来代码结构的改变。例如：对分支的合并和消除，对公用子表达式的消除，对循环内 load/store 操作的替换和更改等，都将会使目标代码的执行顺序变得面目全非，导致调试信息严重不足。

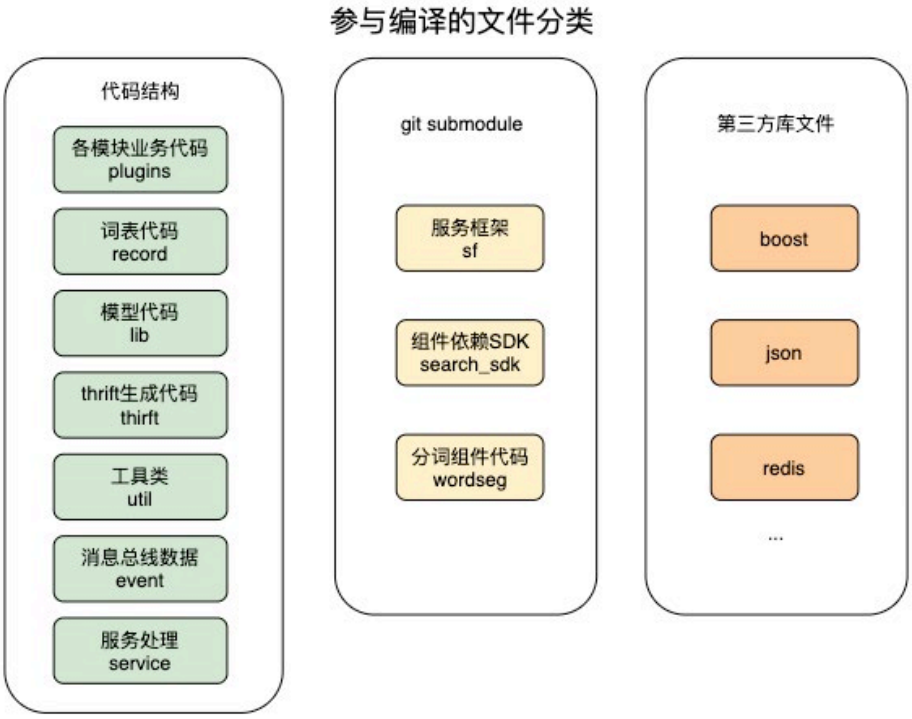
② **内存操作顺序改变问题**：在 O2 优化后，编译器会对影响内存操作的执行顺序。例如：-fschedule-insns 允许数据处理时先完成其他的指令；-fforce-mem 有可能导致内存与寄存器之间的数据产生类似脏数据的不一致等。对于某些依赖内存操作顺序而进行的逻辑，需要做严格的处理后才能进行优化。例如，采用 Volatile 关键字限制变量的操作方式，或者利用 Barrier 迫使 CPU 严格按照指令序执行。

(6)C/C++ 跨编译单元的优化只能交给链接器

当链接器进行链接的时候，首先决定各个目标文件在最终可执行文件里的位置。然后访问所有目标文件的地址重定义表，对其中记录的地址进行重定向（加上一个偏移量，即该编译单元在可执行文件上的起始地址）。然后遍历所有目标文件的未解决符号表，并且在所有的导出符号表里查找匹配的符号，并在未解决符号表中所记录的位置上填写实现地址，最后把所有的目标文件的内容写在各自的位置上，就生成一个可执行文件。链接的细节比较复杂，链接阶段是单进程，无法并行加速，导致大项目链接极慢。

三、服务问题分析

DQU 是美团搜索使用的查询理解平台，内部包含了大量的模型、词表、在代码结构上，包含 20 多个 Thrift 文件 ，使用大量 Boost 处理函数 ，同时引入了 SF 框架，公司第三方组件 SDK 以及分词三个 Submodule，各个模块采用动态库编译加载的方式，模块之间通过消息总线做数据的传输，消息总线是一个大的 Event 类，这样这个类就包含了各个模块需要的数据类型的定义，所以各个模块都会引入 Event 头文件，不合理的依赖关系造成这个文件被改动，几乎所有的模块都会重新编译。



每个服务所面临的编译问题都有各自的特点，但是遇到问题的本质原因是类似的，结合编译的过程和原理，我们从预编译展开、头文件依赖以及编译过程耗时 3 个方面对 DQU 服务编译问题进行了分析。

3.1 编译展开分析

编译展开分析就是通过 C++ 的预编译阶段保留的 .ii 文件，查看通过展开后的编译文件大小，具体可以通过在 cmake 中指定编译选型“-save-temps”保留编译中间文件。

```
set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS} -ggdb -Og -fPIC -w
-Wl,--export-dynamic -Wno-deprecated -fpermissive -save-temps")
```

编译耗时的最直接原因就是编译文件展开之后比较大，通过编译展开后的文件大小和内容，通过预编译展开分析能看到文件展开后的文件有 40 多万行，发现有大量的 Boost 库引用及头文件引用造成的展开文件比较大，影响到编译的耗时。通过这个方式能够找到各个文件编译耗时的共性，下图是编译展开后文件大小截图。

```
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/entity_recognition/entity_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/query_correct/querycorrect_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/query_preprocess/query_preprocess_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:50 ./plugins/query_async_call/query_entity_link.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:42 ./services/query_analysis_server.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/platform/query_classification_pack/query_classification_pack.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/location_recognition/location_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:44 ./plugins/query_chunk/query_chunk_platform_business.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:51 ./plugins/query_async_call/async_query_manager.ii
```

3.2 头文件依赖分析

头文件依赖分析是从引用头文件数量的角度来看代码是否合理的一种分析方式，我们实现了一个脚本，用来统计头文件的依赖关系，并且分析输出头文件依赖引用计数，用来辅助判断头文件依赖关系是否合理。

(1) 头文件引用总数结果统计

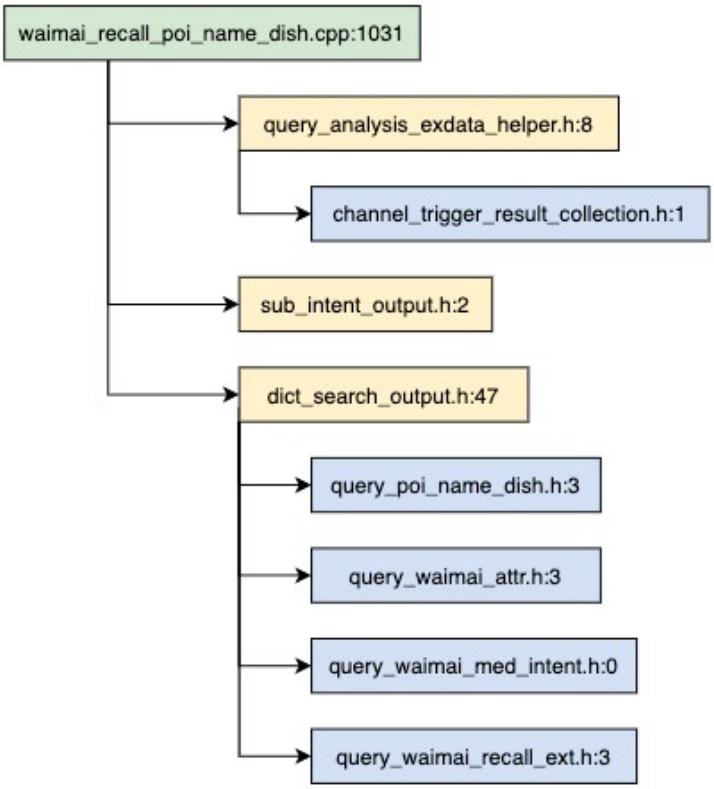
通过工具统计出编译源文件直接和间接依赖的头文件的总个数，用来从头文件引入数量上分析问题。

```
('total', 208934)
('querycorrect_handler.cpp', '\t', 1359)
('recall_strategy.cpp', '\t', 1280)
('query_dqu_hotel_pack.cpp', '\t', 1275)
('processors_handler.cpp', '\t', 1266)
('query_rank_pack.cpp', '\t', 1228)
('query_entity_link.cpp', '\t', 1215)
('query_floating_red_pack.cpp', '\t', 1202)
('async_query_manager.cpp', '\t', 1183)
```

(2) 单个头文件依赖关系统计

通过工具分析头文件依赖关系，生成依赖关系拓扑图，能够直观的看到依赖不合理的地方。

图中包含引用层次关系，以及引用头文件个数。



3.3 编译耗时结果分段统计

编译耗时分段统计是从结果上看各个文件的编译耗时以及各个编译阶段的耗时情况，这个是直观的一个结果，正常情况下，是和文件展开大小以及头文件引用个数是正相关的，cmake 通过指定环境变量能打印出编译和链接阶段的耗时情况，通过这个数据能直观的分析出耗时情况。

```
set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE "${CMAKE_COMMAND} -E
time")
set_property(GLOBAL PROPERTY RULE_LAUNCH_LINK "${CMAKE_COMMAND} -E
time")
```

编译耗时结果输出：

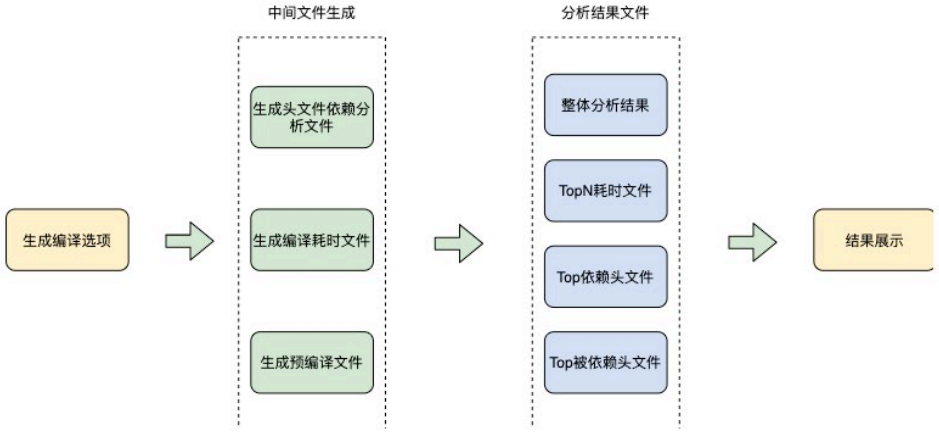
```
Elapsed time: 17 s. (time), 0 s. (clock)
[ 32%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/router_config.cpp.o
Elapsed time: 4 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/rewrite_graph.cpp.o
Elapsed time: 14 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/rewrite_pack_info.cpp.o
Elapsed time: 62 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/util.cpp.o
Elapsed time: 12 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/channel_trigger_common.cpp.o
Elapsed time: 0 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/query_analysis_config_manager.cpp.o
Elapsed time: 7 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/movie_name.cpp.o
Elapsed time: 1 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/channel_utils.cpp.o
Elapsed time: 32 s. (time), 0 s. (clock)
```

3.4 分析工具建设

通过上面的工具分析能拿到几个编译数据：

① 头文件依赖关系及个数。 ② 预编译展开大小及内容。 ③ 各个文件编译耗时。 ④ 整体链接耗时。 ⑤ 可以计算出编译并行度。

通过这几个数据的输入我们考虑可以做个自动化分析工具，找出优化点以及界面化展示。基于这个目的，我们建设了全流程自动化分析工具，能够自动分析耗时共性问题以及 TopN 耗时文件。分析工具处理流程如下图所示：



(1) 整体统计分析效果

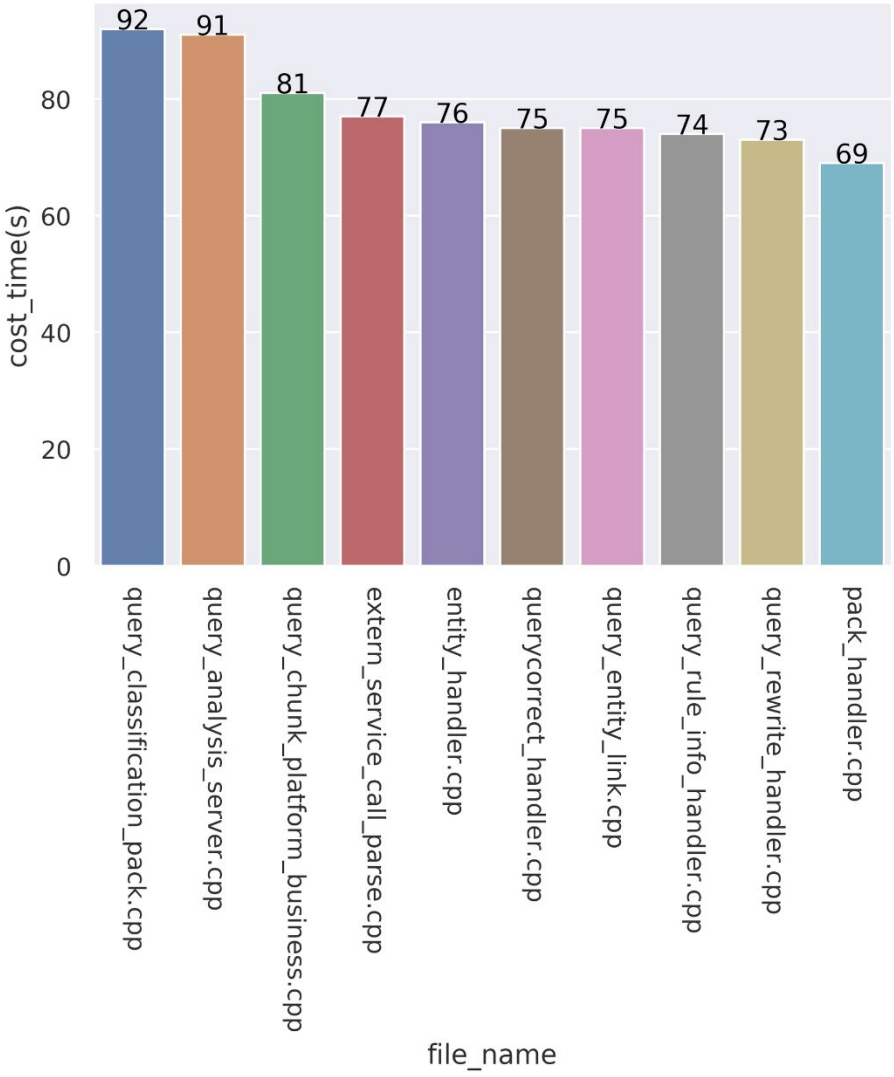
	cost_time(s)	file_compile_size(M)	file_name	include_h_nums	top_h_files_info
0	92	10.863007	query_classification_pack.cpp	1434	[(channel_utils.h, 763)]
1	91	10.788443	query_analysis_server.cpp	1020	[(signal_common_resource.h, 488)]
2	81	10.540730	query_chunk_platform_business.cpp	1038	[(record_search.h, 422)]
3	77	10.198677	extern_service_call_parse.cpp	677	[(platform_query_analysis_event.h, 217)]
4	76	10.832691	entity_handler.cpp	1099	[(record_factory.h, 419)]
5	75	10.839204	querycorrect_handler.cpp	1375	[(resourcemanager.h, 475)]
6	75	10.544925	query_entity_link.cpp	1205	[(address_ner.h, 509)]
7	74	6.678957	query_rule_info_handler.cpp	1022	[(record_search.h, 422)]
8	73	9.101572	query_rewrite_handler.cpp	981	[(resourcemanager.h, 475)]
9	69	9.786721	pack_handler.cpp	1453	[(channel_utils.h, 763)]
10	69	10.463621	location_handler.cpp	818	[(landmark_feature_utils.h, 200)]
11	66	8.549806	query_analysis_event.cpp	728	[(query_analysis_event.h, 139)]
12	66	10.535795	async_query_manager.cpp	1203	[(address_ner.h, 509)]
13	63	10.558627	query_classification_handler.cpp	1204	[(channel_utils.h, 763)]
14	61	10.508329	query_travel_intention.cpp	1153	[(record_search.h, 422)]
15	58	10.313571	query_substring_handler.cpp	802	[(resourcemanager.h, 475)]

具体字段说明：

① cost_time 编译耗时，单位是秒。 ② file_compile_size，编译中间文件大小，单位是 M。 ③ file_name，文件名称。 ④ include_h_nums，引入头文件个数，单位是个。 ⑤ top_h_files_info，引入最多的 TopN 头文件。

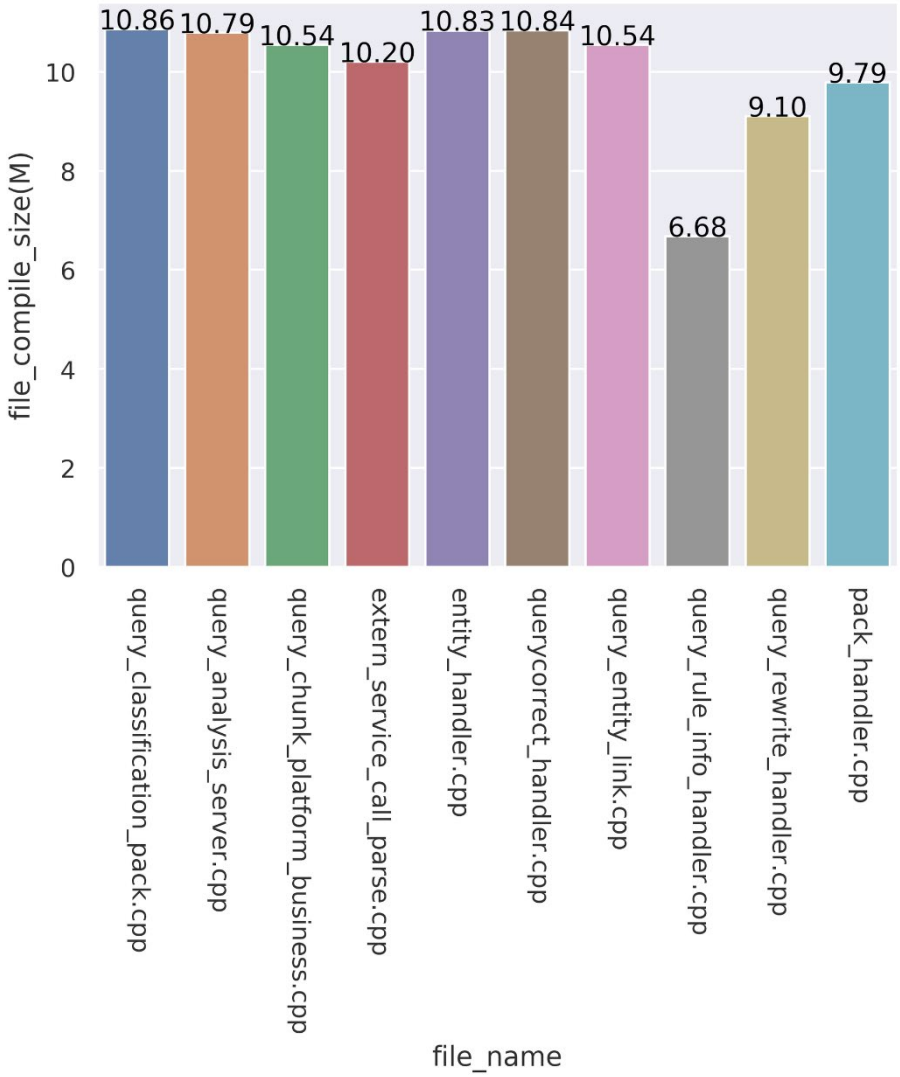
(2) Top10 编译耗时文件统计

用来展示统计编译耗时最久的 TopN 文件，N 可以自定义指定。

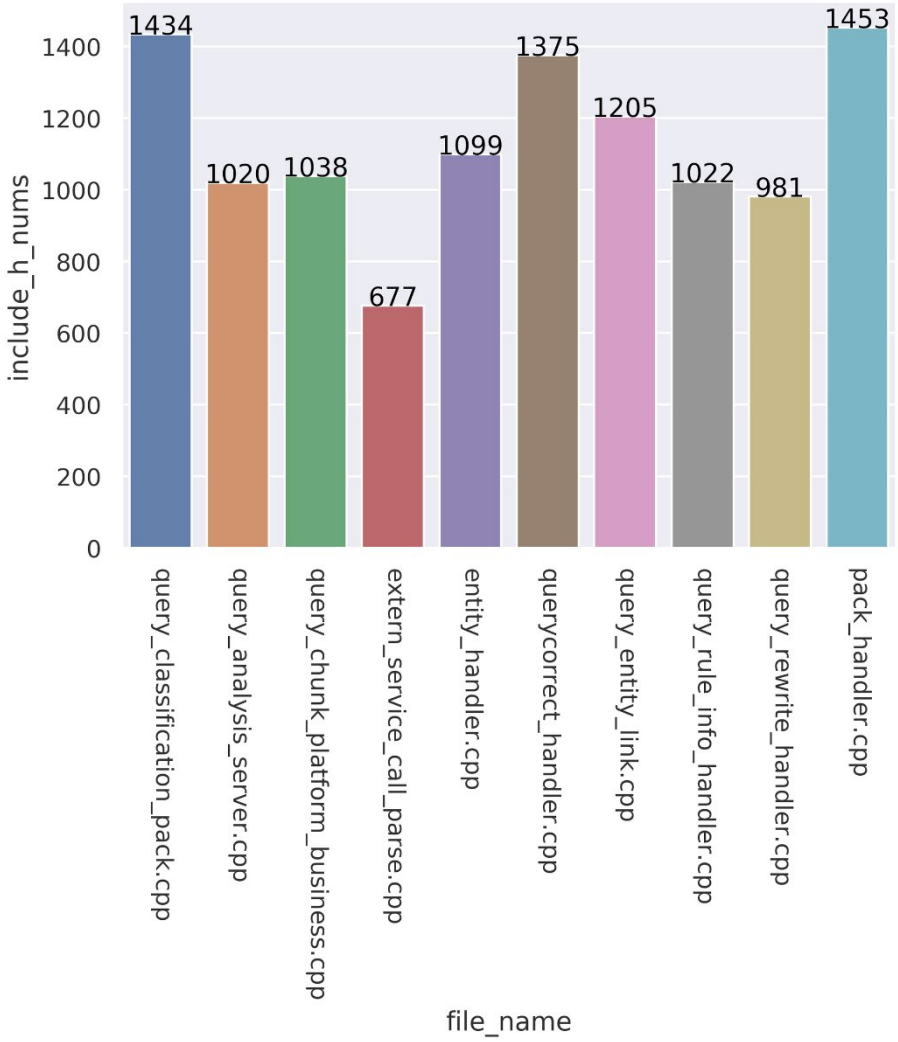


(3) Top10 编译中间文件大小统计

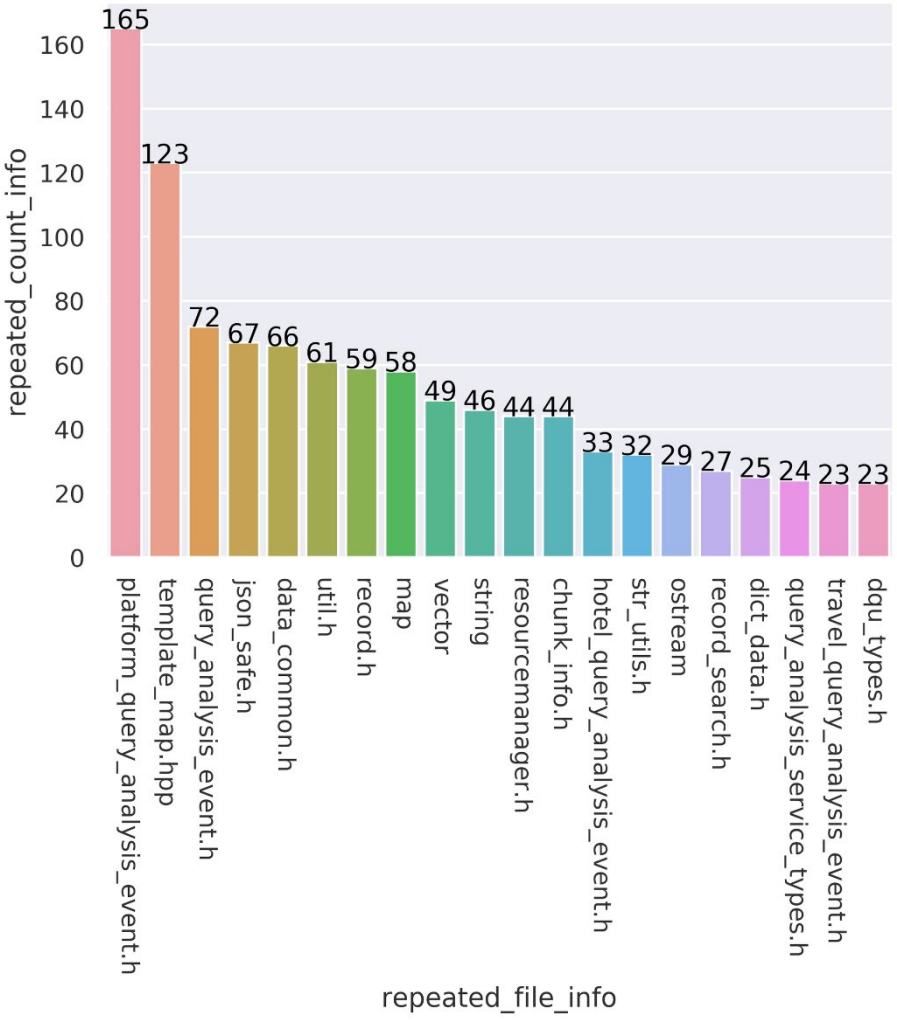
通过统计和展示编译文件大小，用来判断这块是否符合预期，这个是和编译耗时一一对应的。



（4）Top10 引入最多头文件的头文件统计



(5) Top10 头文件重复次数统计



目前，这个工具支持一键化生成编译耗时分析结果，其中几个小工具，比如依赖文件个数工具已经集成到公司的上线集成测试流程中，通过自动化工具检查代码改动对编译耗时的影响，工具的建设还在不断迭代优化中，后续会集成到公司的 MCD 平台中，可以自动分析来定位编译耗时长的问题，解决其它部门编译耗时问题。

四、优化方案与实践

通过运用上述相关工具，我们能够发现 Top10 编译耗时文件的共性，比如都依赖消息总线文件 platform_query_analysis_enent.h，这个文件又直接间接引入 2000 多个头文件，我们重点优化了这类文件，通过工具的编译展开，找出了 Boost 使用、模板类展开、Thrift 头文件展开等共性问题，并针对这些问题做专门的优化。此外，我们也使用了一些业内通用的编译优化方案，并取得了不错的效果。下面详细介绍我们采用的各种优化方案。

4.1 通用编译加速方案

业内有不少通用编译加速工具（方案），无需侵入代码就能提高编译速度，非常值得尝试。

（1）并行编译

在 Linux 平台上一般使用 GNU 的 Make 工具进行编译，在执行 make 命令时可以加上 `-j` 参数增加编译并行度，如 `make -j 4` 将开启 4 个任务。在实践中我们并不将该参数写死，而是通过 `$(nproc)` 方法动态获取编译机的 CPU 核数作为编译并发度，从而最大限度利用多核的性能优势。

（2）分布式编译

使用分布式编译技术，比如利用 Distcc 和 Dmucs 构建大规模、分布式 C++ 编译环境，Linux 平台利用网络集群进行分布式编译，需要考虑网络时延与网络稳定性。分布式编译适合规模较大的项目，比如单机编译需要数小时甚至数天。DQU 服务从代码规模以及单机编译时长来说，暂时还不需要使用分布式的方式来加速，具体细节可以参考 Distcc 官方文档说明。

（3）预编译头文件

PCH（Precompiled Header），该方法预先将常用头文件的编译结果保存起来，这样编译器在处理对应的头文件引入时可以直接使用预先编译好的结果，从而加快整个

编译流程。PCH 是业内十分常用的加速编译的方法，且大家反馈效果非常不错。在我们的项目中，由于涉及到很多 Shared Library 的编译生成，而 Shared Library 相互之间无法共享 PCH，因此没有取得预想效果。

（4）CCache

CCache（Compiler Cache 是一个编译缓存工具，其原理是将 cpp 的编译结果保存在文件缓存中，以后编译时若对应文件无变动可直接从缓存中获取编译结果。需要注意的是，Make 本身也有一定缓存功能，当目标文件已编译（且依赖无变化）时，若源文件时间戳无变化也不会再次编译；但 CCache 是按文件内容做的缓存，且同一机器的多个项目可以共享缓存，因此适用面更大。

（5）Module 编译

如果你的项目是用 C++ 20 进行开发的，那么恭喜你，Module 编译也是一个优化编译速度的方案，C++20 之前的版本会把每一个 cpp 当做一个编译单元处理，会存在引入的头文件被多次解析编译的问题。而 Module 的出现就是解决这一问题，Module 不再需要头文件（只需要一个模块文件，不需要声明和实现两个文件），它会将你的（.ixx 或者 .cppm）模块实体直接编译，并自动生成一个二进制接口文件。import 和 include 预处理不同，编译好的模块下次 import 的时候不会重复编译，可以大幅度提高编译器的效率。

（6）自动依赖分析

Google 也推出了开源的 Include-What-You-Use 工具（简称 IWYU），基于 Clang 的 C/C++ 工程冗余头文件检查工具。IWYU 依赖 Clang 编译套件，使用该工具可以扫描出文件依赖问题，同时该工具还提供脚本解决头文件依赖问题，我们尝试搭建了这套分析工具，这个工具也提供自动化头文件解决方案，但是由于我们的代码依赖比较复杂，有动态库、静态库、子仓库等，这个工具提供的优化功能不能直接使用，其它团队如果代码结构比较简单的话，可以考虑使用这个工具分析优化，会生成如下结果文件，指导哪些头文件需要删除。

```
>>> Fixing #includes in '/opt/meituan/zhoulei/query_analysis/src/common/
qa/record/brand_record.h'
@@ -1,9 +1,10 @@

 #ifndef _MTINTENTION_DATA_BRAND_RECORD_H_
 #define _MTINTENTION_DATA_BRAND_RECORD_H_
-#include "qa/data/record.h"
-#include "qa/data/template_map.hpp"
-#include "qa/data/template_vector.hpp"
-#include <boost/serialization/version.hpp>
+#include <boost/serialization/version.hpp>  // for BOOST_CLASS_
VERSION
+#include <string>                          // for string
+#include <vector>                          // for vector
+
+#include "qa/data/file_buffer.h"           // for REG_TEMPLATE_FILE_
HANDLER
```

4.2 代码优化方案与实践

(1) 前置类型声明

通过分析头文件引用统计，我们发现项目中被引用最多的是总线类型 Event，而该类型中又放置了各种业务需要的成员，示例如下：

```
#include “a.h”
#include "b.h"
class Event {
// 业务A, B, C ...
  A1 a1;
  A2 a2;
     // ...
  B1 b1;
  B2 b2;
  // ...
};
```

这导致 Event 中包含了数量庞大的头文件，在头文件展开后，文件大小达到 15M；而各种业务都会需要使用 Event，自然会严重拖累编译性能。

我们通过前置类型声明来解决这个问题，即不引入对应类型的头文件，只做前置声明，在 Event 中只使用对应类型的指针，如下所示：

```
class A2;
// ...
class Event {
// 业务A, B, C ...
  shared_ptr<A1> a1;
  shared_ptr<A2> a2;
  // ...
  shared_ptr<B1> b1;
  shared_ptr<B2> b2;
  // ...
};
```

只有在真正使用对应成员变量时，才需要引入对应头文件；这样真正做到了按需引入头文件。

(2) 外部模板

由于模板被使用时才会实例化这一特性，相同的实例可以出现在多个文件对象中。编译器要对每一处模板进行实例化，链接器还要移除重复的实例化代码。当在广泛使用模板的项目中，编译器会产生大量的冗余代码，这会极大地增加编译时间和链接时间。C++ 11 新标准中可以通过外部模板来避免。

```
// util.h
template <typename T>
void max(T) { ... }
// A.cpp
extern template void max<int>(int);
#include "util.h"
template void max<int>(int); // 显式地实例化
void test1()
{
    max(1);
}
```

在编译 A.cpp 的时候，实例化出一个 max(int) 版本的函数。

```
// B.cpp
#include "util.h"
extern template void max<int>(int); // 外部模板的声明
void test2()
{
    max(2);
}
```

在编译 B.cpp 的时候，就不再生成 max(int) 实例化代码，这样就节省了前面提到的实例化，编译以及链接的耗时了。

（3）多态替换模板使用

我们的项目重度使用词典相关操作，如加载词典、解析词典、匹配词典（各种花式匹配），这些操作都是通过 Template 模板扩展支持各种不同类型的词典。据统计，词典的类型超过 150 个，这也造成模板展开的代码量膨胀。

```
template <class R>
class Dict {
public:
  // 匹配 key 和 condition，赋值给 record
  bool match(const string &key, const string &condition, R &record);
// 对每种类型的 Record 都会展开一次
private:
  map<string, R> dict;
};
```

幸运的是，我们词典的绝大部分操作都可以抽象出几类接口，因此可以只实现针对基类的操作：

```
class Record {  // 基类
public:
  virtual bool match(const string &condition);  // 派生类需实现
};

class Dict {
public:
  shared_ptr<Record> match(const string &key, const string &condition);
// 使用方传入派生类的指针即可
private:
  map<string, shared_ptr<Record>> dict;
};
```

通过继承和多态，我们有效避免了大量的模板展开。需要注意的是，使用指针作为 Map 的 Value 会增加内存分配的压力，推荐使用 Tcmalloc 或 Jemalloc 替换默认的 Ptmalloc 优化内存分配。

(4) 替换 Boost 库

Boost 是一个广泛使用的基础库，涵盖了大量常用函数，十分方便、好用，然而也存在一些不足之处。一个显著缺点是其实现采用了 hpp 的形式，即声明和实现均放在头文件中，这会造成预编译展开后十分巨大。

```
// 字符串操作是常用功能，仅仅引入该头文件展开大小就超过 4M
#include <boost/algorithm/string.hpp>
// 与此相对的，引入多个 STL 的头文件，展开后仅仅只有 1M
#include <vector>
#include <map>
// ...
```

在我们项目中主要使用的 Boost 函数不超过二十个，部分可以在 STL 中找到替代，部分我们手动做了实现，使得项目从重度依赖 Boost 转变成绝大部分达到 Boost-Free，大大降低了编译的负担。

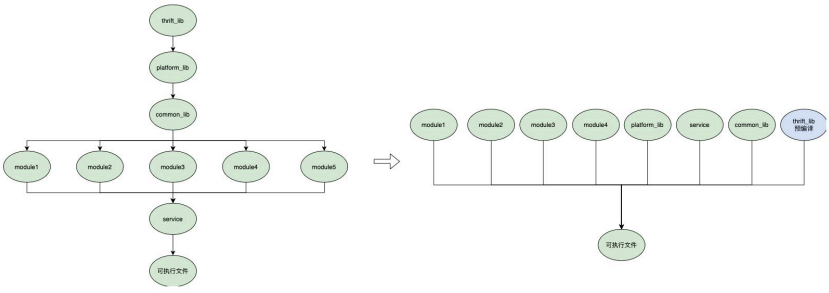
(5) 预编译

代码中有一些平常改动比较少，但是对编译耗时产生一定的影响，比如 Thrift 生成的文件，模型库文件以及 Common 目录下的通用文件，我们采取提起预编译成动态库，减少后续文件的编译耗时，也解决了部分编译依赖。

(6) 解决编译依赖，提高编译并行度

在我们项目中有大量模块级别的动态库文件需要编译，cmake 文件指定的编译依赖关系在一定程度上限制了编译并行度的执行。

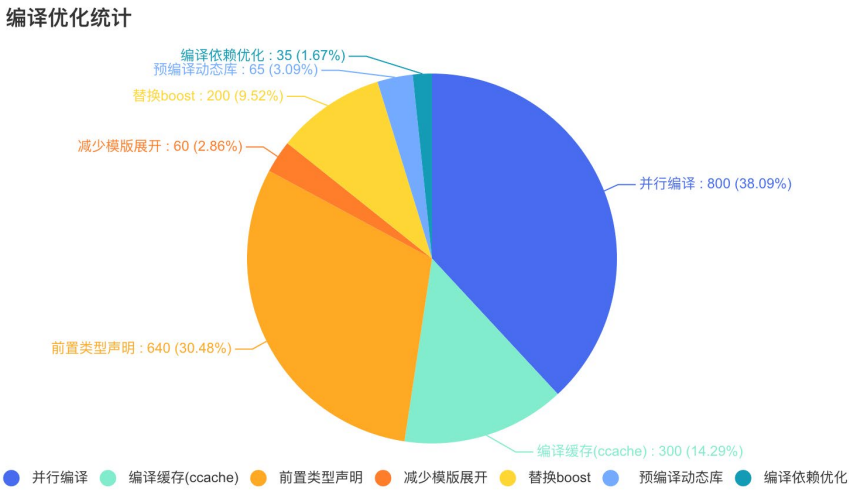
比如下面这个场景，通过合理设置库文件依赖关系，可以提高编译并行度。



4.3 优化效果

我们通过 32C、64G 内存机器做了编译耗时优化前后的效果对比，统计结果如下：

序号	优化方案	优化效果	说明
1	并行编译	800秒	根据CPU核数，提高编译并行度，16换32并行度
2	编译缓存（CCache）	300秒	首次编译无效果。CCache的优化和缓存命中情况有关，以普通更改部分cpp或者头文件测试方式
3	前置类型声明	640秒	通过前置声明，解决头文件重复依赖问题
4	减少模板展开	60秒	通过外部模板和多态替换模板来解决
5	替换Boost	200秒	减少Boost库使用
6	预编译动态库	65秒	提前编译Thrift文件打包成动态库
7	编译依赖优化	35秒	调整makefile，提高编译并行度
	总体效果	2100秒	首次编译从优化前的2200秒优化到500秒，非首次编译优化到100秒左右



4.4 守住优化成果

编译优化是一件“逆水行舟”的事情，开发人员总是倾向于不断增加新的功能、新的

库乃至新的框架，而要删除旧代码、旧库、下线旧框架总是困难重重（相信一线开发人员一定深有体会）。因此，如何守住之前取得的优化成果也是至关重要的。我们在实践中有以下几点体会：

- 代码审核是困难的（引起编译耗时增加的改动，往往无法通过审核代码直观地发现）。
- 工具、流程才值得依赖。
- 关键在于控制增量。

我们发现，cpp 文件的编译耗时，和其预编译展开文件（.ii）大小呈正相关（绝大部分情况下）；对每一个上线版本，将其所有 cpp 文件的预编译展开大小记录下来，就形成了其编译指纹（CF，Compile Fingerprint）。通过比较相邻两个版本的 CF，就能较准确的知道新版带来的编译耗时主要由哪些改动引入，并可以进一步分析耗时上涨是否合理，是否有优化空间。

我们将该种方式制作成脚本工具并引入上线流程，从而能够很清楚的了解每次代码发版带来的编译性能影响，并有效地帮助我们守住前期的优化成果。

五、总结

DQU 项目是美团搜索业务环节中重要的一环，该系统需要对接 20+RPC、数十个模型、加载超过 300 个词典，使用内存数十 G，日均响应请求超过 20 亿的大型 C++ 服务。在业务高速迭代的情况，冗长的编译时间为开发同学带来较大的困扰，一定程度上制约了开发效率。最终我们通过编译优化分析工具建设，结合采用了通用编译优化加速方案和代码层面的优化，将 DQU 的编译时间缩短了 70%，并通过引 CCache 等手段，使得本地开发的编译，能够在 100s 内完成，给开发团队节省了大量的时间。

在取得阶段性成果之后，我们总结整个问题解决的过程，并沉淀出一些分析方法、工具以及流程规范。这些工具在后续的开发迭代过程中，能够快速有效地检测新的

代码变更带来的编译时间变化，并成为了我们的上线流程检查中的一环检测标准。这一点与我们以往一次性的或者针对性的编译优化，产生了很大的区别。毕竟代码的维护是一个持久的过程，系统化的解决这一问题，不只是需要有效的方法和便捷的工具，更需要一个标准化的，规范化的上线流程来保持成果。希望本文对大家能有所帮助。

参考文献

[1]《编译原理透视 · 图解编译原理》
[2] CCache
[3] 分布式编译
[4] 头文件预编译
[5] 头文件预编译
[6] C++ Templates
[7] Include-what-you-use

作者简介

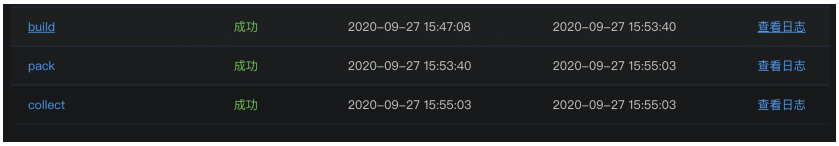
本文作者周磊、识瀚、朱超、王鑫、刘亮、昌术、李超、云森、永超等，均来自美团 AI 平台搜索与 NLP 部。

速度与压缩比如何兼得？压缩算法在构建部署中的优化

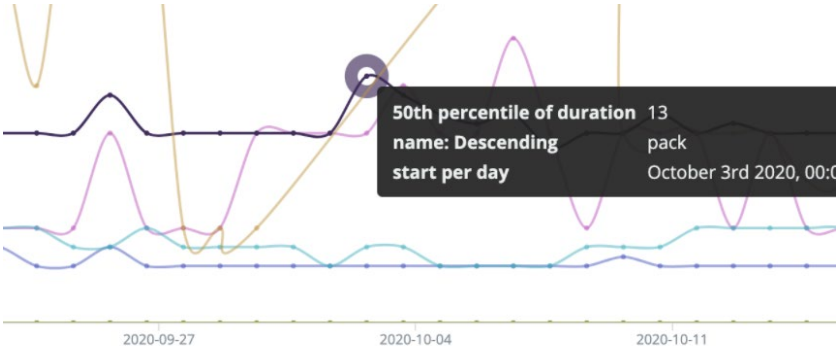
作者：宏达

背景

通常而言，服务发布平台的构建部署的流程（镜像部署除外）会经过**构建**（同步代码 -> 编译 -> 打包 -> 上传）、**部署**（下载包 -> 解压到目标机器 -> 重启服务）等步骤。以美团内部的发布平台 Plus 为例，最近我们发现一些发布项在构建产物打包压缩的过程中耗时比较久。如下图所示的 pack 步骤，一共消耗了 1 分 23 秒。



而在平常为用户解答运维问题的时候我们也发现，很多用户会习惯将一些较大的机器学习或者 NLP 相关的数据放入到仓库中，这部分数据往往占据几百兆，甚至占据几个 GB 的磁盘空间，十分影响打包的速度。 Java 项目也是如此，由于 Java 服务框架繁多，依赖也多，通常这些服务打包后也要占据百兆级别的空间，耗时也会达到十多秒。下图是我们的 pack 步骤的中位数，基本上大部分的 Java 服务和 Node.js 服务都至少要消耗 13s 左右的时间来做压缩打包 。



pack 作为几乎所有需要部署的服务必需步骤，它目前的耗时基本上仅低于编译和构建镜像，因此，为了提高整体构建的效率，我们准备对 pack 打包压缩的步骤进行一轮优化工作。

方案对比

准备场景数据

发布项的包大小分析

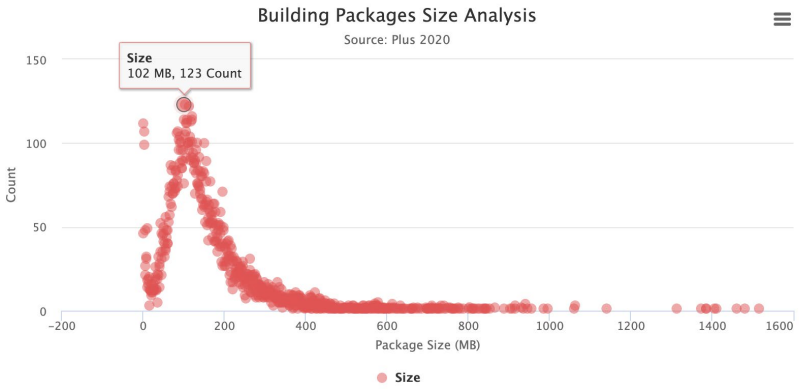
为了尽可能地模拟构建部署中的应用场景，我们将 2020 年的部分**构建包数据**进行了整理分析，其中压缩后的包大小如下图所示，钟形曲线说明了整体的包体积呈正态分布，并且有着较明显的长尾效应。压缩后体积主要在 200M 以内，压缩前的大小大致在 516.0MB 以内。

而 99% 的服务压缩包大小会在 1GB 以内，而对于压缩步骤而言，其实越大的项目耗时越明显，优化的空间越大。因此，我们在针对性的方案对比测试中选择了 1GB 左右的构建包进行压缩测试，既能覆盖 99% 的场景，也可以看出压缩算法之间比较明显的提升。

这样选择的主要原因如下：

1. 数据大的情况下计算结果会比小数据误差小很多。
2. 能够覆盖绝大多数应用场景。
3. 效果对比明显，可以看到是否有明显的提升。

备注：由于在相同压缩库相同压缩比等配置的情况下，Compression Speed 并没有明显变化，因此没有做其它包体积的批量测试和数据汇总。



本文中我们使用的测试项目为美团内部的较大型的 C++ 项目，其中文件类型除去 C++、Python、Shell 代码文件，还有 NLP、工具等二进制数据（不包括 .git 中存储的提交数据），数据类型比较全面。

目录大小为 1.2G，也可以比较清晰地对比出不同方案之间的差距。

gzip

gzip 是基于 DEFLATE 的算法，它是 LZ77 和 Huffman 编码 的结合。DEFLATE 的目的是为了取代 LZW 和其他受专利保护的数据压缩算法，因为这些算法在当时限制了压缩和其他流行的存档器的可用性（Wikipedia）。

我们通常使用 `tar -czf` 命令来进行打包并且压缩的操作，z 参数正是使用 gzip 的方式来进行压缩。DEFLATE 标准（RFC1951）是一个被广泛使用的无损数据压缩标准。它的压缩数据格式由一系列块构成，对应输入数据的块，每一块通过 LZ77（基于字典压缩，就是将最高概率出现的字母以最短的编码表示）算法和 Huffman 编码进行压缩，LZ77 算法通过查找并替换重复的字符串来减小数据体积。

文件格式

- 一个 10 字节的报头，包含一个魔数 (1f 8b)，压缩方法（比如 08 用于 DEFLATE），1 字节的 header flags，4 字节的时间戳，compression flags 和操作系统 ID。

- 可选的额外 headers，包括原始文件名、注释字段、“extra”字段和 header 的 CRC-32 校验码 lower half 。
- DEFLATE 压缩主体。
- 8 字节的 footer，包含 CRC-32 校验以及原始未压缩的数据。

我们可以看到 gzip 是主要基于 CRC 和 Huffman LZ77 的 DEFLATE 算法，这也是后面 ISA-L 库的优化目标。

Brotli

Alakuijala 和 Szabadka 在 2013-2016 年完成了 Brotli 规范，该数据格式旨在进一步提高压缩比，它在优化网站速度上有大量应用。Brotli 规范的正式验证是由 Mark Adler 独立实现的。Brotli 是一个用于数据流压缩的数据格式规范，它使用了通用的 LZ77 无损压缩算法、Huffman 编码和二阶上下文建模（2nd order context modelling）的特定组合。大家可以参考这篇论文查看其实现原理。

因为语言本身的特性，基于上下文的建模方法（Context Modeling）可以得到更好的压缩比，但由于它的性能问题很难普及。当前比较流行的突破算法有两种：

- ANS：Zstd, lzfse
- Context Modeling：Brotli, bz2

具体测试数据见下文。

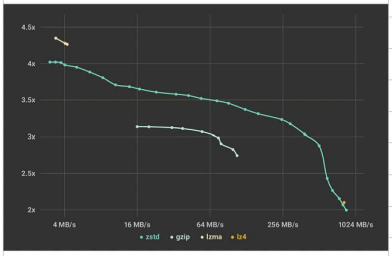
Zstd

Zstd 全称叫 Zstandard，是一个提供高压缩比的快速压缩算法，主要实现的编程语言为 C，是 Facebook 的 Yann Collet 于 2016 年发布的，Zstd 采用了有限状态熵（Finite State Entropy，缩写为 FSE）编码器。该编码器是由 Jarek Duda 基于 ANS 理论开发的一种新型熵编码器，提供了非常强大的压缩速度 / 压缩率的折中方案（事实上也的确做到了“鱼”和“熊掌”兼得）。Zstd 在其最大压缩级别上提供的压缩比接近 lzma、lzham 和 ppmx，并且性能优于 lza 或 bzip2。Zstandard 达到了

Pareto frontier（资源分配最佳的理想状态），因为它解压缩速度快于任何其他当前可用的算法，但压缩比类似或更好。

对于小数据，它还特别提供一个载入预置词典的方法优化速度，词典可以通过对目标数据进行训练从而生成。

官方 Benchmark 数据对比

zstd 与 gzip lzma lz4 对比	Compressor name	Ratio	Compression	Decompress
	Zstd 1.4.5 -1	2.884	500 MB/s	1660 MB/s
	zlib 1.2.11 -1	2.743	90 MB/s	400 MB/s
	brotli 1.0.7 -0	2.703	400 MB/s	450 MB/s
	Zstd 1.4.5 --fast=1	2.434	570 MB/s	2200 MB/s
	Zstd 1.4.5 --fast=3	2.312	640 MB/s	2300 MB/s
	Zstd 1.4.5 --fast=5	2.178	700 MB/s	2420 MB/s
	lzo1x 2.10 -1	2.106	690 MB/s	820 MB/s
	lz4 1.9.2	2.101	740 MB/s	4530 MB/s
	snappy 1.1.8	2.073	560 MB/s	1790 MB/s

压缩级别可以通过 –fast 指定，提供更快的压缩和解压缩速度，相比级别 1 会导致压缩比率的一些损失，如上表所示。Zstd 可以用压缩速度换取更强的压缩比。它是可配置的小增量，在所有设置下，解压缩速度都保持不变，这是大多数 LZ 压缩算法（如 zlib 或 lzma）共享的特性。

- 我们采用 Zstd 默认的参数进行了测试，压缩时间 8.471s 仅为原来的 11.266%，提升了 88.733%。
- 解压时间 3.211 仅为原来的 29.83%，提升约为 70.169%。
- 同时压缩率也从 2.548 提升到了 2.621。

LZ4

LZ4 是一种无损压缩算法，每核提供大于 500 MB/s 的压缩速度（大于 0.15 Bytes/cycle）。它的特点是解码速度极快，每核速度为多 GB/s（ 约 1 Bytes/cycle ）。

从上面的 Zstd 的 Benchmark 对比中，我们看到了 LZ4 算法效果十分出众，因此我们也对 LZ4 进行了对比，LZ4 更加侧重压缩解压速度，尤其是解压缩的速度，压缩

比并不是它的强项，它默认支持 1-9 的压缩参数，我们分别进行了测试。

LZ4 使用默认参数压缩速度十分优秀，比 Zstd 快很多，但是压缩比并不高，比 Zstd 压缩后多了 206 MB，足足多了 46%，这就意味着更多的数据传输时间和磁盘空间占用。即使是最大的压缩比也并不高，仅仅从 1.79 提升到了 2.11，但是耗时却从 5s 提升到了 51s。通过对比，LZ4 的确在压缩率上并不是最优秀的方案，在 2.x 级别压缩率上基本上时间优势荡然无存，而且还有一点，就是 LZ4 目前官方并没有对多核 CPU 并行压缩的支持，所以在后续的对比中，LZ4 丧失了压缩解压缩速度的优势。

Pigz

Pigz 的作者 Mark Adler，同时也是 Info-ZIP 的 zip 和 unzip、GNU 的 gzip 和 zlib 压缩库的共同作者，并且是 PNG 图像格式开发工作的参与者。

Pigz 是 gzip 的并行实现的缩写，它主要思想就是利用多个处理器和核。它将输入分成 128 KB 的块，每个块都被并行压缩。每个块的单个校验值也是并行计算的。它的实现直接使用了 zlib 和 pthread 库，比较易读，而且重要的是兼容 gzip 的格式。Pigz 使用一个线程（主线程）进行解压缩，但可以创建另外三个线程进行读、写和检查计算，所以在某些情况下可以加速解压缩。

一些博客在 i7 4790K 这样的家用 PC 平台中测试 Pigz 的压缩性能时，并没有十分高的速度，但在我们真机验证的数据中提升要明显很多。通过测试，它的压缩时间执行速度只用了 1.768s，充分发挥了我们平台物理机的性能，User 时间（CPU 时间之和）一共使用了 1m30.569s，这和前面的使用 gzip 单线程的方式速度几乎是一个级别。压缩率 2.5488 和正常使用 `tar -czf` 几乎相差不多。

ISA-L Acceleration Version

ISA-L 是一套在 IA 架构上加速算法执行的开源函数库，目的在于解决存储系统的计算需求。 ISA-L 使用的是 BSD-3-Clause License ，因此在商业上同样可以使用。

使用过 SPDK（Storage Performance Development Kit ）或者 DPDK（Data Plane

Development Kit）应该也听说过 ISA-L ，前者使用了 ISA-L 的 CRC 部分，后者使用了它的压缩优化。ISA-L 通过使用高效的 SIMD（Single Instruction, Multiple Data）指令和专用指令，最大化的利用 CPU 的微架构来加速存储算法的计算过程。ISA-L 底层函数都是使用手工汇编代码编写，并在很多细节上做了调优（现在经常要考虑 ARM 平台，本文中所提及的部分指令在该平台支持度不高甚至是不支持）。

ISA-L 对压缩算法主要做了 CRC、DEFLATE 和 Huffman 编码的优化实现，官方的数据指出 ISA-L 相比 zlib-1 有 5 倍的速度提升。

举例来说，不少底层的存储开源软件实现的 CRC 都使用了查表法，代码中存储或者生成了一个 CRC value 的表格，然后计算过程中查询值，ISA-L 的汇编代码中包含了无进位乘法指令 PCLMULQDQ 对两个 64 位数做无进位乘法，最大化 intel PCLMULQDQ 指令的吞吐量来优化 CRC 的性能。更好的情况是 CPU 支持 AVX-512，就可以使用 VPCLMULQDQ（PCLMULQDQ 在 EVEX 编码的 512 bit 版本实现）等其它优化指令集（查看是否支持的方式见“附录”）。

```
244     .16B_reduction_loop:
245         vpclmulqdq  xmm8, xmm7, xmm10, 0x11
246         vpclmulqdq  xmm7, xmm7, xmm10, 0x00
247         vpxor       xmm7, xmm8
248         vmovdqu     xmm0, [arg2]
249         vpshufb     xmm0, xmm0, xmm18
250         vpxor       xmm7, xmm0
251         add     arg2, 16
252         sub     arg3, 16
```

备注：截图来自 crc32_ieee_by16_10.asm

使用

ISA-L 实现的压缩优化级别支持 [0,3]，3 为压缩比最大的版本，综合考虑我们采用

了最大的压缩比，虽然压缩比 2.346 略低于 gzip 不过影响不大。在 2019 年 6 月发布的 v2.27 版本里，ISA-L 加了多线程的 Feature，因此在后续的测试中，我们采用了多线程并发的参数，效果提升比较显著。

由于我们构建机器的系统为 Centos7 需要自行编译 ISA-L 的依赖，比如 nasm 等库，所以在安装配置上会比较复杂，ISA-L 支持多种优化参数比如，IGZIP_HIST_SIZE（压缩过程中加大滑动窗口），LONGER_HUFFTABLES，更大的 Huffman 编码表，这些编译参数也会对库有很大提升。

压缩时间

```
real  0m1.372s
user  0m5.512s
sys 0m1.791s
```

测试后的效果相当惊人，是目前对比方案中最为快速的，时间上节省了 98.09%。

由于和 gzip 格式兼容，因此同样可以使用 `tar -xf` 命令进行解压，后续的解压缩测试过程中，我们使用的仍然是 ISA-L 提供的解压方式。

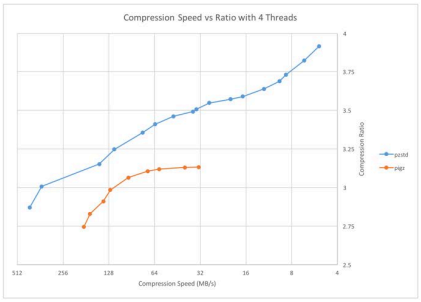
Pzstd

通过 Pigz 的测试，我们就在想，是否 Zstd 这样优秀的算法也可以支持并行呢，在官方的 Repo 中，我们十分惊喜地发现了一个“宝藏”。

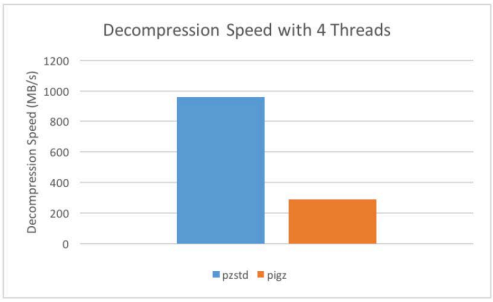
Pzstd 是 C++11 实现的并行版本的 Zstandard（Zstd 也在这之后加入了多线程的支持），类似于 Pigz 的工具。 它提供了与 Zstandard 格式兼容的压缩和解压缩功能，可以利用多个 CPU 核心。 它将输入分成相等大小的块，并将每个块独立压缩为 Zstandard 帧。 然后将这些帧连接在一起以产生最终的压缩输出。 Pzstd 同样支持文件的并行解压缩。 解压缩使用 Zstandard 压缩的文件时，PZstandard 在一个线程中执行 IO，而在另一个线程中进行解压缩。

下图是和 Pigz 的压缩和解压缩速度对比，来自官方 Github 仓库（机器配置为：

Intel Core i7 @ 3.1 GHz, 4 threads），效果比 Pigz 还要出色，具体对比数据见下文：



压缩速度



解压缩速度

Pied Piper (Middle-out compression)

Middle-out Compression 最初是在美剧中提到的一个概念，不过现在已经有了一个真正的实现 middle-out，该算法目前小范围应用在压缩时序数据中，由于缺乏成熟地理论支撑以及数据对比，没有正式进入方案的对标。

备注：Pied Piper 是美剧《硅谷》中虚拟出来的公司和算法。



兼容性

本文中调研所提及的对比方案，都是统一在构建集群中的机器中进行测试，由于构建系统 在线上的机器集群配置高度统一（包括硬件平台和系统版本），所以兼容性表现和测试数据也高度吻合。

选型

实际上，部分官方的测试机器的配置和美团构建平台的物理机配置并不一致，场景可能也有区别，上面引用的测试结果中所使用的 CPU 以及平台架构和编译环境和我们所用的也有些出入，而且大多数还是家用的硬件比如 i7-4790K 或者 i9-9900K，这也是需要使用构建平台的物理机和具体的构建包压缩场景来进行测试的原因，因为这样才能得出最接近我们使用场景的数据。

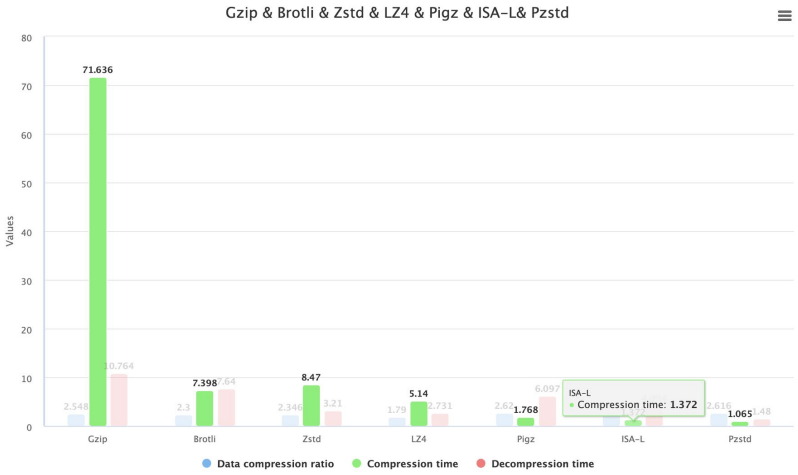
对比数据

几个方案的数据对比如下表格（在本文中的时间数据选择是通过多次运行后，选择结果的中位数）：

	Gzip	Brotli	Zstd	LZ4	Pigz	ISA-L	Pzstd
Data Compression Ratio	2.5482	2.3068	2.621	1.791	2.5488	2.346	2.616
Compression Time	71.636	7.398	8.471	5.141	1.768	1.372	1.065
Compression Speed	16.272 MB/s	157.57 MB/s	137.613 MB/s	226.75 MB/s	659.345 MB/s	849.652 MB/s	1094.5752 MB/s
Decompression Time	10.764	7.643	3.211	2.731	6.097	4.094	1.481

压缩时间对比

从整个构建后的压缩构建包的时间可视化图中可以看出，最初版本的 gzip 压缩相当耗时，而采取 Pzstd 是最快速的方案，ISA-L 稍慢，Pigz 略微慢一点，这**三者都可以达到从 1m11s 到 1s 左右的优化，最快可以节省 98.51% 的压缩时间**。



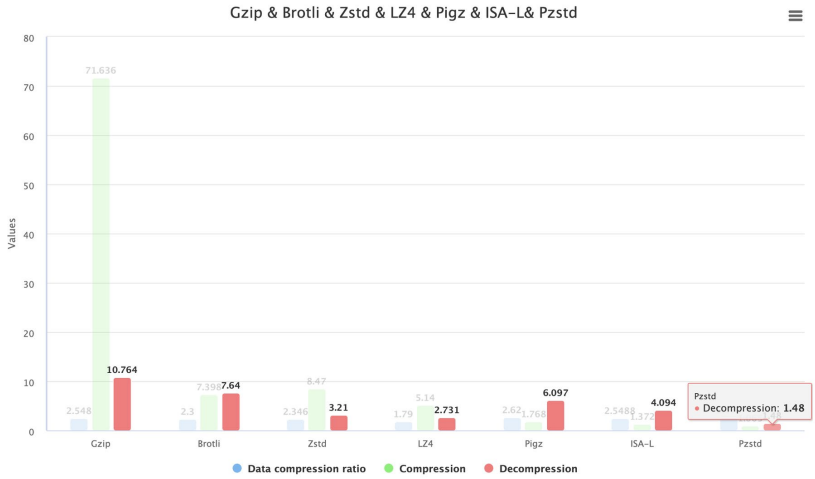
解压缩时间对比

解压缩的时间上并没有像压缩效率相差很多，在 GB 级别的项目中压缩比 2.5-2.6 范围内时，时间都在 11s 以内。而且为了最大兼容已有的实现和保持稳定性，解压方案优先考虑兼容 gzip 格式的策略，这样对部署目标机器的侵入性最小，即可以使用 `tar -xf` 解压的方案优先。

而在后面的方案实施中，由于部署需要稳定可靠的环境，所以我们暂时没有对部署机器做环境改造。

下面的时间对比是分别使用各自的解压方案的对比：

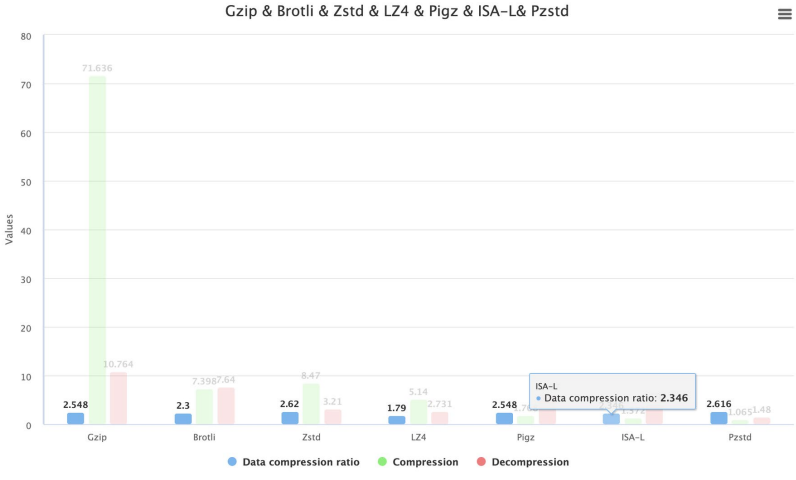
- Pzstd 解压速度最快，相比 Gzip 节省了 86.241% 的时间。
- Zstd 算法的解压缩效率其次，大约可以节省 70.169% 的解压时间。
- ISA-L 可节省 61.9658% 的时间。
- Pigz 可节省 43.357% 的解压时间。
- Brotli 解压可以节省 29.02% 的时间。



压缩比的对比

压缩比的对比中 Zstd 和 Pzstd 有一些优势，其中 Brotli 和 LZ4 由于支持的参数限制，比较难测试同级别压缩比下的速度，因此选择了压缩比稍低的参数，但是效率仍然距离 Pigz 和 Pzstd 存在一些差距。

而 ISA-L 的实现在压缩比上有一些牺牲，不过并没有差距很大。



优劣分析总结

	理念	优势	劣势
Brotli	1. **优秀的新算法**	• 对于小文件的速度可能会稍微明显一些 • 压缩解压缩的速度比较不错，但是并不突出。	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 Brotli 来进行处理，对于其它有可能已经使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 大型文件尤其是在构建场景中，它的压缩速度并不是十分突出。 • 解压必需 Brotli 库的安装（也就是说部署的目标机器需要安装 Brotli 库）
Zstd	1. **优秀的新算法**	• **压缩解压速度十分优秀。** • 安装依赖单一，避免遇到麻烦的依赖冲突问题。 • **压缩比十分优秀。**	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 zstd 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 zstd 库的安装（也就是说部署的目标机器需要安装 zstd 库）
LZ4	1. **优秀的新算法**	• **压缩解压速度十分优秀。** • 安装依赖单一，避免遇到麻烦的依赖冲突问题。	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 LZ4 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 LZ4 库的安装（也就是说部署的目标机器需要安装 LZ4 库） • 压缩率并不理想
Pigz	1. 并行策略	• **压缩结果完全兼容 gzip 格式**，即便解压不做改动也可以完全兼容，这对于使用了构建压缩包的其它服务可能是十分完美的结果。同时对于部署目标机器的侵入性最小。 • **压缩解压速度极其优秀，仅次于 Pzstd 和 ISA-L。** • 对 tar 支持比较友好，可以方便地使用外接压缩插件。 • 安装依赖单一，避免遇到麻烦的依赖冲突问题。	无
ISA-L	1. gzip 算法以及底层指令优化 2. 并行策略	• **压缩结果完全兼容 gzip 格式**，即便解压不做改动也可以完全兼容，这对于使用了构建压缩包的其它服务可能是十分完美的结果。同时对于部署目标机器没有侵入性。 • **压缩解压速度极其优秀，仅次于 Pzstd。** • 对 tar 支持比较友好。	• 压缩比 稍微低一点点，且压缩比的参数范围有限。 • 实现接入成本略微复杂（不过好在我们的机器集群配置环境十分统一）。
Pzstd	1. **优秀的新算法** 2. 并行策略	• **压缩和解压的速度都极其优秀。** • 对 tar 支持比较友好，只需要在机器上安装 zstd 并且修改 tar 参数即可，改动较小。 • 安装依赖单一，避免遇到麻烦的依赖冲突问题。 • **压缩比十分优秀。**	• **不兼容之前的 gzip 格式**，由于中间生成的压缩文件格式需要严格使用 zstd 来进行处理，对于其它有可能使用 Plus 平台构建包的平台来说，兼容性是一个挑战。 • 解压必需 zstd 库的安装（也就是说部署的目标机器需要安装 zstd 库）

在本文开始阶段，我们介绍了构建部署流程，因此我们此次优化的目标时间大致计算公式如下：

$$TotalCost = Time_{compression} + \frac{PackageSize}{UploadSpeed} + \frac{PackageSize}{DownloadSpeed} + Time_{decompression}$$

在测试案例对比中，时间耗时的顺序为 Pzstd < ISA-L < Pigz < LZ4 < Zstd < Brotli < Gzip（排名越靠前越好），其中压缩和解压缩的时间在整体的耗时上占比较大，因此备选策略为 Pzstd、ISA-L、Pigz。

当然，这其中还存在磁盘空间的成本优化问题，即压缩比可能对磁盘空间产生优化，但是对构建的耗时会产生负优化，不过由于目前时间维度是我们优化的主要目标，比磁盘成本和上传带宽成本要重要，因此压缩比采用了较为普遍或者默认最优的压缩比方案，即在 2.3 – 2.6 范围内。不过在一些内存型数据库等存储介质成本较为高的场景中，也许要综合多个方面需要更多考量，请大家知悉。

评分策略

比较于 gzip，新算法都具有想当不错的速度，但是由于压缩格式的向前不兼容，加上需要客户端（部署目标机器）的支持，可能方案实施周期会较长。

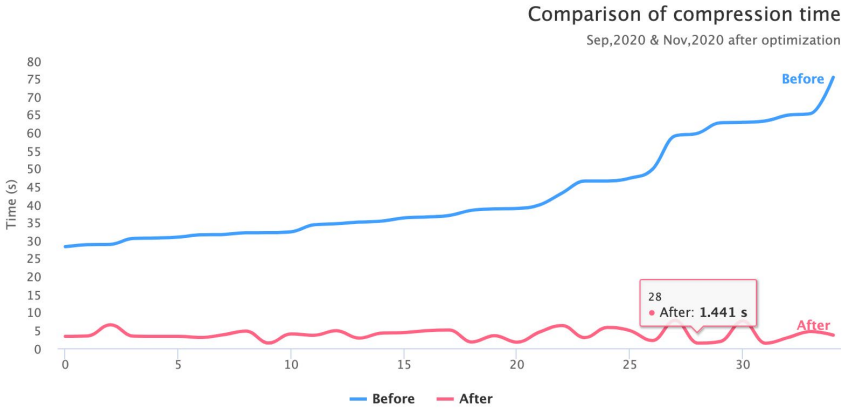
而对于部署来说，可能收益并不是十分明显，反而加重了一些维护和运维成本，所以我们暂时没有把 Pzstd 的方案放到较高的优先级。

选型的策略主要有基于如下原则：

- 整体耗时优化提升最大，这也是整体优化方案的出发点。
- 保证最大兼容性，为了让接入构建平台的业务和平台减少改动成本，需要保持方案的兼容性（优先考虑最大兼容的策略，即兼容 gzip 的方案优先）。
- 保证部署目标机器环境的稳定和可靠，选择对部署机器侵入最小的方案，这样无需安装客户端或者库。
- 压缩场景在真机模拟测试中完全契合美团构建平台的场景，即在我们现有的物理机平台和目标压缩场景中对比数据效果良好。
- 其实本问题更全面的评分角度有很多维度，比如对象存储的磁盘成本、带宽成本、任务耗时，甚至是机器成本，不过为了简化整体方案的选型，我们省略了一些计算，同时压缩比的对比选择上也选择了各自官方推荐的范围。

综合以上几点，决定一期采取 ISA-L 的方式加速，可以最稳定并且较高速地提升构建平台的效率，未来可能会实现 Pzstd 的方案，下面的数据为一期的结果。

优化效果

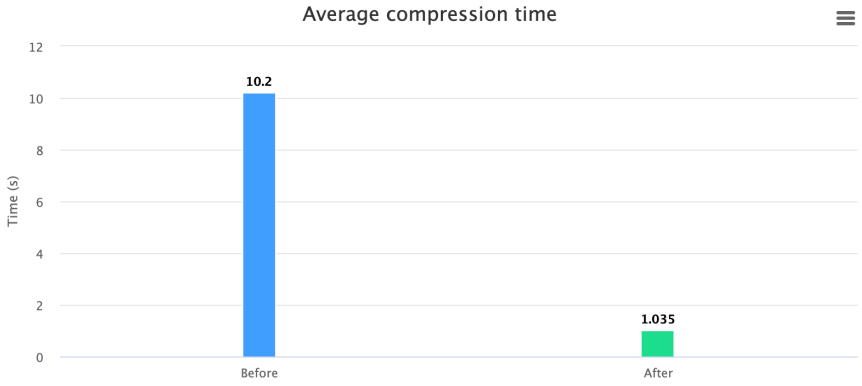


为了方便结果的展示，我们过滤出了部分打包时间较长的发布项展示出来（这些耗时很久的项目往往十分影响用户的使用体验，而且总体的占比在 10% 左右），这部分任务优化的时间从 27s 到 72s 不等，过去越是项目大的项目压缩时间越长，如今压缩时间都可以稳定在 10s 以内，而且是在我们的机器同时执行多个任务的情况下。

而后我们将优化前的 Pack 步骤（压缩 + 上传）部分打点数据，以及优化后的部分打点数据做了汇总，得出了平均的优化效果对比，数据如下：

1. 在我们之前的一个构建包的统计中，多数的构建包压缩后在 100MB 左右，压缩前大概是在 250MB，按照 gzip 算法的压缩速度的确会在 10s 左右的级别。
2. 由于构建的物理机可能同时运行多个任务，所以实际压缩效果会比测试中稍微耗时多一点。

	Pack 耗时之和	任务数	平均时间	压缩时间（去除复制文件和上传的时间）
优化前	908202	65613	13.8417996433634	10.2
优化后	215298	46853	4.59518067146181	1.035



压缩平均节省了 90% 的时间。

写在后面

- 由于文中提到的一些方案涉及到具体平台环境的 CPU 指令集，甚至库的编译环境，编译参数也会影响具体的效果，所以推荐在方案实施的时候对集群环境保持统一，也可为集群环境做特殊的定制优化。
- 为 Centos 打包 RPM 文件的时候尤其需要注意下编译环境的配置，否则可能效果和测试会有出入。
- Java 的 Jar 包 和 War 包也可能进行压缩，针对这种场景，压缩率的确提升不大，但是速度依旧有提升。

作者简介

宏达，美团基础技术部研发工程师。

团队简介

基础技术部 – 研发质量及效率部 – 代码仓库和构建组，团队旨在建设代码仓库管理、协作及代码构建能力，完善多维度的工作流执行引擎及构建工具链，与公司其他研发工具打通，提高业务整体的开发、交付效率。

附录

机器环境

文中的测试统一在如下物理机中进行，测试中使用相同的目标文件。测试 机使用的是非 PCIE SSD 磁盘。

```
inxi -Fx 省略部分数据输出如下，其中一些并行指令集在优化中可能会使用到。其中
flags 中可以看到支持 avx avx2 指令集，并不支持 avx-512 ，不过仍然有很大性能提升。
System:   Host: ****** Kernel: ****** bits: 64 compiler: gcc v: 4.8.5
Console: tty 7 Distro: CentOS Linux release 7.1.1503 (Core)

CPU:       Info: 2x 16-Core model: Intel Xeon Gold 5218 bits: 64 type:
MT MCP SMP arch: Cascade Lake rev: 7 L2 cache: 44.0 MiB

           flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3
vmx bogomips: 293978 Speed: 2300 MHz min/max: N/A

Info:        Processes: 764  Memory: 187.19 GiB used: 15.05 GiB (8.0%)
Init: systemd runlevel: 3 Compilers: gcc: 4.8.5
```

```
/proc/cpuinfo 文件中也可以查看 CPU 支持的指令集

flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64
monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca
sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch epb cat_l3 cdp_l3 intel_ppin intel_pt ssbd mba ibrs ibpb
stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle
avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx
smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1
cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts
pku ospke spec_ctrl intel_stibp flush_l1d arch_capabilities
```

参考文献

https://engineering.fb.com/core-data/smaller-and-faster-data-compression-with-zstandard/

https://engineering.fb.com/core-data/zstandard/

https://peazip.github.io/fast-compression-benchmark-brotli-zstandard.html

https://news.ycombinator.com/item?id=16227526

https://github.com/facebook/zstd

http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html
https://zlib.net/pigz/
https://cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf
https://bugs.python.org/issue41566
https://01.org/sites/default/files/documentation/isa-l_api_2.28.0.pdf
https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf

招聘信息

美团研发质量及效率部 – 研发平台团队，致力于建设业界一流的持续交付平台，现招聘构建与制品库、代码仓库、服务发布、流水线等多个方向的工程师，坐标北京 / 上海。欢迎感兴趣的同学加入。可投递简历至：dongjing04@meituan.com（邮件主题请注明：美团研发质量及效率部 – 研发平台）

美团 OCTO 万亿级数据中心计算引擎技术解析

作者：继东　业祥　成达　张昀

美团自研的 OCTO 数据中心（简称 Watt）日均处理万亿级数据量，该系统具备较好的扩展能力及实时性，千台实例集群周运维成本低于 10 分钟。本文将详细阐述 Watt 计算引擎的演进历程及架构设计，同时详细介绍其全面提升计算能力、吞吐能力、降低运维成本所采用的各项技术方案。希望能给大家一些启发或者帮助。

一、OCTO 数据中心简介

1.1 系统介绍

1.1.1 OCTO 系统介绍

OCTO 是美团标准化的服务治理基础设施，目前几乎覆盖公司所有业务的治理与运营。OCTO 提供了服务注册发现、数据治理、负载均衡、容错、灰度发布等治理功能，致力于提升研发效率，降低运维成本，并提升应用的稳定性。OCTO 最新演进动态细节可参考《美团下一代服务治理系统 OCTO2.0 的探索与实践》一文。

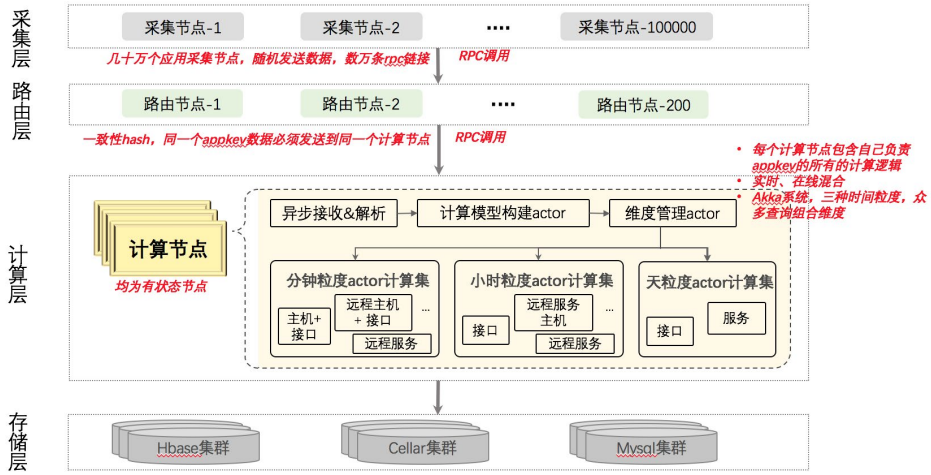
1.1.2 OCTO 数据中心业务介绍

OCTO 数据中心为业务提供了立体化的数字驱动服务治理能力，提供了多维度的精确时延 TP（Top Percent，分位数，最高支持 6 个 9）、QPS、成功率等一系列核心指标，粒度方面支持秒级、分钟级、小时级、天级，检索维度支持多种复杂查询（如指定调用端 IP + 服务端各接口的指标，指定主机 + 接口的指标等）。这些功能有效地帮助开发人员在复杂的分布式调用关系拓扑内出现异常时，能快速定位到问题，也有助于研发人员全方位掌控系统的稳定性状况。

目前 Watt 承载日均超万亿条数据的 10 余个维度精确准实时统计。而伴随着数据量的迅猛增长，其整个系统架构也经历了全面的技术演进。

1.1.3 OCTO 原架构介绍

OCTO 计算引擎在重构之前，也面临诸多的问题，其原始架构设计如下：



1. **采集层**：每个业务应用实例部署一个采集代理，代理通过将采集数据用批量 RPC 的方式发送给路由节点。
2. **路由层**：每个路由节点随机收到各服务的数据后，将同一服务的所有数据，用类似 IP 直连的方式通过 RPC 发送到计算层的同一个计算节点。同服务数据汇总到同计算节点才能进行特定服务各个维度的聚合计算。
3. **计算层**：每个计算节点采用 Akka 模型，节点同时负责分钟、小时、天粒度的数据计算集。每个计算集里面又有 10 个子计算 actor，每个子计算 actor 对应的是一个维度。采用“先计算指标，再存储数据”的准实时模式。
4. **存储层**：准实时数据使用 HBase 存储，元数据及较大数据采用 KV 存储（美团内部系统 Cellar）及 MySQL 存储。

1.2 问题、目标与挑战

1.2.1 原架构面临的问题

1. **计算节点有状态，异常无法自动化迁移**。计算层部署的每个节点平均负责 200+ 服务的统计。一个节点 OOM 或宕机时，其管理的 200 个服务的数据

会丢失或波动，报警等依托数据的治理功能也会异常。此外计算节点 OOM 时也不宜自动迁移受影响的服务，需人工介入处理（异常的原因可能是计算节点无法承载涌入的数据量，简单的迁移易引发“雪崩”），每周投入的运维时间近 20 小时。

2. **系统不支持水平扩展**。计算节点的压力由其管理的服务调用量、服务内维度复杂度等因素决定。大体量的服务需单独分配高配机器，业务数据膨胀计算节点能力不匹配时，只能替换更高性能的机器。
3. **系统整体稳定性不高**。数据传输采用 RPC，单计算节点响应慢时，易拖垮所有路由层的节点并引发系统“雪崩”。
4. **热点及数据倾斜明显，策略管理复杂**。按服务划分热点明显，存在一个服务数据量比整个计算节点 200 个服务总数多的情况，部分机器的 CPU 利用率不到 10%，部分利用率在 90%+。改变路由策略易引发新的热点机器，大体量服务数据增长时需要纵向扩容解决。旧架构时人工维护 160 余个大服务到计算节点的映射关系供路由层使用。

旧架构日承载数据量约 3000 亿，受上述缺陷影响，系统会频繁出现告警丢失、误告警、数据不准、数据延迟几小时、服务发布后 10 分钟后才能看到流量等多种问题。此外，数据体量大的服务也不支持部分二级维度的数据统计。

1.2.2 新架构设计的目标

基于上述问题总结与分析，我们新架构整体的目标如下：

1. 高吞吐、高度扩展能力。具备 20 倍 + 的水平扩展能力，支持日 10 万亿数据的处理能力。
2. 数据高度精确。解决节点宕机后自愈、解决数据丢失问题。
3. 高可靠、高可用。无计算单点，所有计算节点无状态；1/3 计算节点宕机无影响；具备削峰能力。
4. 延时低。秒级指标延迟 TP99<10s；分钟指标延迟 TP99<2min。

1.2.3 新架构面临的挑战

在日计算量万亿级别的体量下，实现上述挑战如下：

1. 数据倾斜明显的海量数据，数据指标的维度多、指标多、时间窗口多，服务间体量差异达百万倍。
2. TP 分位数长尾数据是衡量系统稳定性最核心的指标，所有数据要求非采样拟合，实现多维度下精确的分布式 TP 数据。
3. 架构具备高稳定性，1/3 节点宕机不需人工介入。
4. 每年数据膨胀至 2.4 倍 +，计算能力及吞吐能力必须支持水平扩展。
5. TP 数据是衡量服务最核心的指标之一，但在万亿规模下，精确的准实时多维度分布式 TP 数据是一个难题，原因详细解释下：常规的拆分计算后聚合是无法计算精确 TP 数据的，如将一个服务按 IP（一般按 IP 划分数据比较均匀）划分到 3 个子计算节点计算后汇总，会面临如下问题：
   - 假设计算得出 IP1 的 TP99 是 100ms、QPS 为 50；IP2 的 TP99 是 10ms、QPS 为 50；IP3 的 TP99 是 1ms，QPS 为 50。那么该服务整体 TP99 是（100ms x 50 + 10ms x 50 + 1ms x 50）/ (50 + 50 + 50) = 37ms 吗？ 并非如此，该服务的真实 TP99 可能是 100ms，在没有全量样本情况下无法保证准确的 TP 值。
   - 假设不需要服务全局精确的时延 TP 数据，也不可以忽略该问题。按上述方式拆分并合并后，服务按接口维度计算的 TP 数据也失去了准确性。进一步说，只要不包含 IP 查询条件的所有 TP 数据都失真了。分位数这类必须建立在全局样本之上才能有正确计算的统计值。

二、计算引擎技术设计解析

2.1 方案选型

大数据计算应用往往基于实时或离线计算技术体系建设，但 Flink、Spark、OLAP 等技术栈在日超万亿级别量级下，支持复杂维度的准实时精确 TP 计算，对资源的消耗非常较大，总结如下：

技术体系	特定场景不适宜原因分析
Hadoop 离线任务	（1）时延不满足，未纳入考虑
Storm 实时计算	（1）内存密集型，时间窗口大 （2）批处理实时性受较大影响
Flink 实时流计算	（1）大窗口排序（2）大量shuffle和sort，降低DAG优化（3）同一条数据需做广播管理
Spark 实时流计算	（1）大窗口排序，微批架构吞吐有限（2）热点明显散列管理较复杂（3）集群资源较难控制
OLAP 存储引擎	（1）高实时性要求下精确TP计算较难支持

2.2 系统设计思路

1. 解决稳定性问题，思路是（1）将计算节点无状态化（2）基于心跳机制自动剔除异常节点并由新节点承载任务（3）消息队列削峰。
2. 解决海量数据计算问题，思路是（1）在线 & 离线计算隔离，两者的公共子计算前置只计算一次（2）高并发高吞吐能力设计（3）理论无上限的水平扩展能力设计。
3. 解决热点问题，思路是（1）优化计算量分配算法，如均衡 Hash（2）理论无上限的水平扩展能力设计。
4. 解决水平扩展问题，思路（1）是将单节点无法承载的计算模式改为局部分布式子计算并汇总，但这种方式可能会对数据准确性造成较大影响，数据统计领域精确的分布式分位数才是最难问题，另外多维条件组织对分布式改造也相当不利。（备注：其中在 1.2.3 第五条有详细的解释）
5. 解决海量数据分布式多维精确 TP 计算，采用局部分布式计算，然后基于拓扑树组织数据计算流，在前置的计算结果精度不丢失的前提下，多阶段逐级降维得到所有的计算结果。

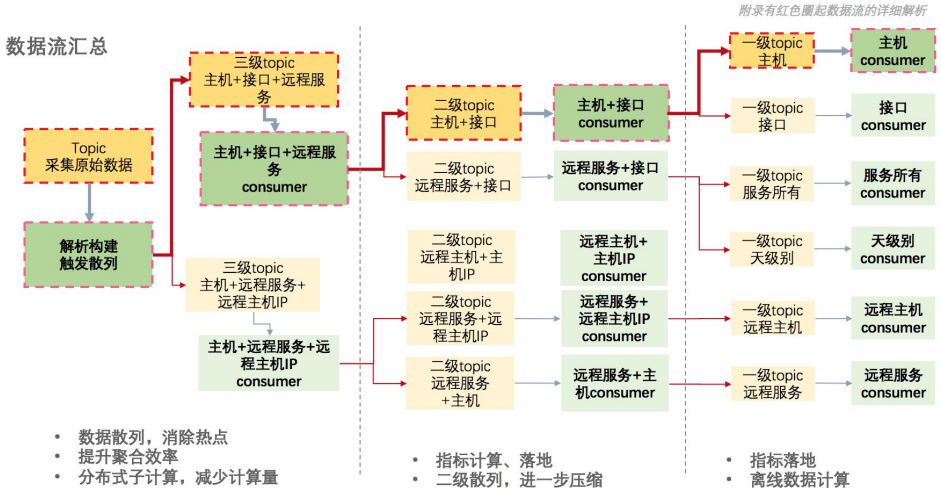
2.3 技术方案详细解析

2.3.1 数据流解析

系统根据待统计的维度构建了一棵递推拓扑树，如下图所示。其中黄色的部分代表消

息队列（每个矩形代表一个 topic），绿色部分代表一个计算子集群（消费前置 topic 多个 partition，统计自己负责维度的数据指标并存储，同时聚合压缩向后继续发）。除“原始采集数据 topic 外”，其他 topic 和 consumer 所在维度对应数据的检索条件（如标红二级 topic ：主机 + 接口，代表同时指定主机和接口的指标查询数据），红色箭头代表数据流向。

拓扑树形结构的各个子集群所对应的维度标识集合，必为其父计算集群对应维度标识集合的真子集（如下图最上面链路，“主机 + 接口 + 远程服务”3 个维度一定包含“主机 + 接口”两个维度。“主机 + 接口”两个维度包含“主机”维度）。集群间数据传输，采用批量聚合压缩后在消息队列媒介上通信完成，在计算过程中实现降维。



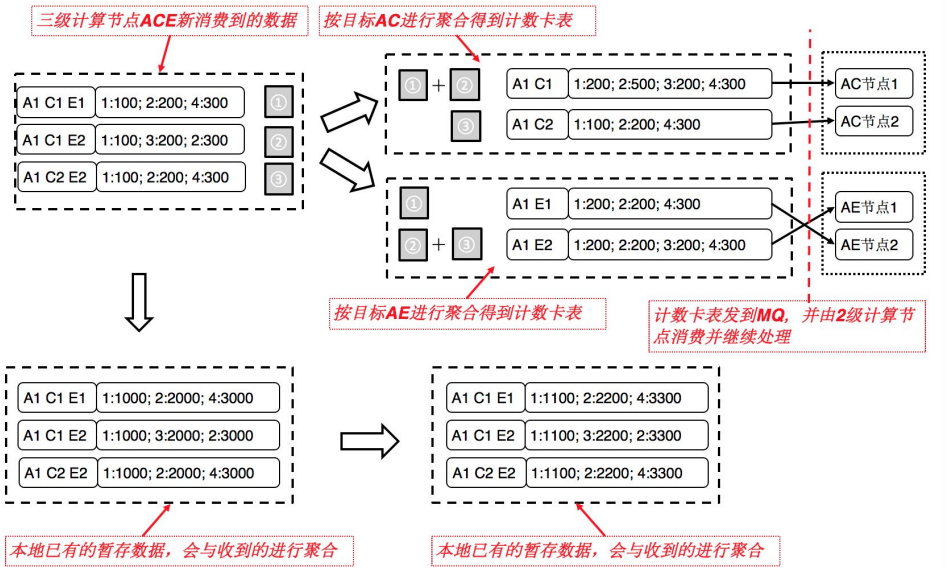
2.3.2 计算模式解析

下面详细介绍数据拓扑树中分布式子集群的计算模式：

1. 首先，维度值相同的所有数据会在本层级集群内落到同一计算节点。每个计算子集群中的各计算节点，从消息队列消费得到数据并按自身维度进行聚合（前置集群已经按当前集群维度指定分发，所以聚合率很高），得到若干计数卡表（计数卡表即指定维度的时延、错误数等指标与具体计数的映射 Map）。

2. 其次，将聚合后的计数卡表与现有的相同维度合并计算，并在时间窗口存储指标。
3. 若计算集群有后续的子计算集群，则基于后继集群的目标维度，根据目标维度属性做散列计算，并将相同散列码的计数卡表聚合压缩后发到后继 partition。
4. 离线的天级计算复用了三级维度、二级维度的多项结果，各环节前面计算的结果为后面多个计算集群复用，任何一个服务的数据都是在分布式计算。此外，整个计算过程中维护着技术卡表的映射关系，对于 TP 数据来说就是精确计算的，不会失真。

整个计算过程中，前置计算结果会被多个直接及间接后续子计算复用（如三级聚合计算对二级和一级都是有效的），在很大程度上减少了计算量。



2.3.3 关键技术总结

1. 高吞吐 & 扩展性关键设计

- 去计算热点：组织多级散列数据流，逐级降维。
- 降计算量：前置子计算结果复用，分布式多路归并。

- 降网络 IO，磁盘 IO：优化 PB 序列化算法，自管理 MQ 批量。
- 去存储热点：消除 HBase Rowkey 热点。
- 无锁处理：自研线程分桶的流批处理模型，全局无锁。
- 全环节水平扩展：计算、传输、存储均水平扩展。

2. 系统高可靠、低运维、数据准确性高于 5 个 9 关键设计

- 无状态化 + 快速自愈：节点无状态化，宕机秒级感知自愈。
- 异常实时感知：异常节点通过心跳摘除，异常数据迁移回溯。
- 故障隔离容灾：各维度独立隔离故障范围；多机房容灾。
- 逐级降维过程中数据不失真，精确的 TP 计算模式。

3. 提升实时性关键设计

- 流式拓扑模型，分布式子计算结果复用，计算量逐级递减。
- 无锁处理：自研线程分桶的流批处理模型，全局无锁。
- 异常实时监测自愈：计算节点异常时迅速 Rebalance，及时自愈。
- 秒级计算流独立，内存存储。

三、优化效果

1. 目前日均处理数据量超万亿，系统可支撑日 10 万亿 + 量级并具备良好的扩展能力；秒峰值可处理 5 亿 + 数据；单服务日吞吐上限提升 1000 倍 +，单服务可以支撑 5000 亿数据计算。
2. 最大时延从 4 小时 + 降低到 2min−，秒级指标时延 TP99 达到 6s；平均时延从 4.7 分 + 降低到 1.5 分 −，秒级指标平均时延 6s。
3. 上线后集群未发生雪崩，同时宕机 1/3 实例 2 秒内自动化自愈。
4. 支持多维度的准实时精确 TP 计算，最高支持到 TP 6 个 9；支持所有服务所有维度统计。
5. 千余节点集群运维投入从周 20 小时 + 降低至 10 分 −。

四、总结

本文提供了一种日均超万亿规模下多维度精确 TP 计算的准实时数据计算引擎方案，适用于在超大规模数字化治理体系建设中，解决扩展性、实时性、精确性、稳定性、运维成本等问题。美团基础研发平台欢迎业界同行一起交流、探讨。

五、作者简介

继东，业祥，成达，张昀，均来自基础架构服务治理团队，研发工程师。

招聘信息

美团基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：美团基础架构团队）。

Intel PAUSE 指令变化影响到 MySQL 的性能，该如何解决?

作者：春林

MySQL 得益于其开源属性、成熟的商业运作、良好的社区运营以及功能的不断迭代与完善，已经成为互联网关系型数据库的标配。可以说，X86 服务器、Linux 作为基础设施，跟 MySQL 一起构建了互联网数据存储服务的基石，三者相辅相成。本文将分享一个工作中的实践案例：因 Intel PAUSE 指令周期的迭代，引发了 MySQL 的性能瓶颈，美团 MySQL DBA 团队如何基于这三者来一步步进行分析、定位和优化。希望这些思路能对大家有所启发。

1. 背景

在 2017 年，Intel 发布了新一代的服务器平台 Purley，并将 Intel Xeon Scalable Processor（至强可扩展处理器）重新划分为：Platinum（铂金）、Gold（金）、Silver（银）、Broze（铜）等四个等级。产品定位和框架也变得更加清晰。

因美团线上海量数据交易和存储等后端服务依赖大量高性能服务器的支撑。随着线上部分 Grantly 平台 E 系列服务器生命周期的临近，以及产品本身的发展和迭代。从 2019 年开始，RDS（关系型数据库服务）后端存储（MySQL）开始大量上线 Purley 平台的 Skylake CPU 服务器，其中包含 Silver 4110 等。

Silver 4110 相比上一代 E5-2620 V4，支持更高的内存频率、更多的内存通道、更大的 L2 Cache、更快的总线传输速率等。Intel 官方数据显示 Silver 4110 的性能比上一代 E5-2620 V4 提升了 10%。

然而，随着线上 Skylake 服务器数量的增加，以及越来越多的业务接入。美团 MySQL DBA 团队发现部分 MySQL 实例性能与预期并不相符，有时甚至出现较大程度的下降。经过持续的性能问题分析，我们定位到 Skylake 服务器存在性能瓶颈：

- CPU 负载相对较高。
- TPS 等吞吐量下降。

接下来，我们将从 Intel CPU、ut_delay 函数、PAUSE 指令三方面入手，进行剖析定位，并探索相关优化方案。

2. 性能问题分析

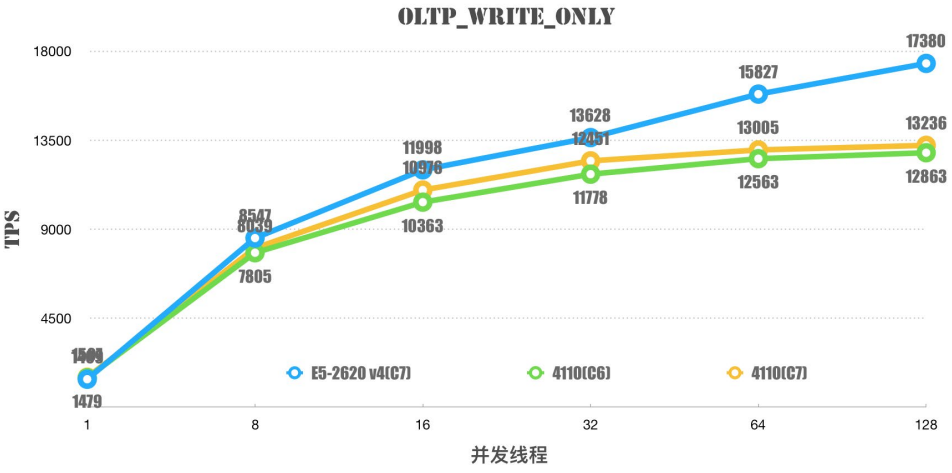
2.1 Grantly 与 Purley CPU 性能差异

首先，基于上述两代平台的 CPU（Grantly 和 Purley），通过基准测试，横向对比在不同 OS 下的性能表现。

通过基准测试数据，总结如下：

1. 在 oltp_write_only（只写）的场景下 Purley 4110 的性能下降较为明显。 2. 同为 Purley 4110，CentOS 7 比 CentOS 6 oltp_write_only（只写）性能有提升。

我们通过二维折线图，来展示性能之间的差异：



在上图中，同为 Purley 4110，CentOS 7 比 CentOS 6 性能有提升。具体提升原因，因不涉及本文重点内容，所以不在这里详细展开了。

> New MCS-based Locking Mechanism
>
> Red Hat Enterprise Linux 7.1 introduces a new locking mechanism, MCS locks. This new locking mechanism significantly reduces spinlock overhead in large systems, which makes spinlocks generally more efficient in Red Hat Enterprise Linux 7.1.

红帽官网 Release Notes 显示，从内核 3.10.0-229 开始，引入了新的加锁机制，MCS 锁。可以降低 spinlock 的开销，从而更高效地运行。普通 spinlock 在多 CPU Core 下，同时只能有一个 CPU 获取变量，并自旋，而缓存一致性协议为了保证数据的正确，会对所有 CPU Cache Line 状态、数据，同步、失效等操作，导致性能下降。而 MSC 锁实现每个 CPU 都有自己的“spinlock”本地变量，只在本地自旋。避免 Cache Line 同步等，从而提升了相关性能。不过，社区对于 spinlock 的优化争议还是比较大的，后续又有大牛基于 MSC 实现了 qspinlock，并在 4.x 的版本上 patch 了。具体实现可以参看：MCS locks and qspinlocks。

在大致了解 CentOS 7 性能的迭代后，接下来我们深入分析一下 Skylake CPU 4110 导致性能下降的缘由。

3.CPU 性能跟踪

3.1 定位热点函数

具体定位 4110 性能瓶颈，分如下几步：

1. 首先，通过 perf top 来跟踪一下 Linux CPU 性能开销。
2. 然后，通过 perf record 记录函数 CPU 周期的消耗占比。
3. 最后，通过火焰图来验证定位热点函数。

```
  Children      Self  Shared Object       Symbol
+   77.13%     0.00%  mysqld              [.] handle_connection
+   77.12%     0.15%  mysqld              [.] do_command
+   74.41%     0.35%  mysqld              [.] dispatch_command
+   71.34%     0.02%  mysqld              [.] mysqld_stmt_execute
+   71.22%     0.15%  mysqld              [.] Prepared_statement::execute_loop
+   70.08%     0.01%  mysqld              [.] Prepared_statement::execute
+   68.54%     0.22%  mysqld              [.] mysql_execute_command
+   25.24%     0.05%  mysqld              [.] trans_commit_stmt
+   25.17%     0.11%  mysqld              [.] ha_commit_trans
+   21.73%     0.05%  mysqld              [.] Sql_cmd_update::execute
+   21.67%     0.02%  mysqld              [.] Sql_cmd_update::try_single_table_update
+   20.38%     0.14%  mysqld              [.] mysql_update
+   19.46%     0.08%  mysqld              [.] MYSQL_BIN_LOG::commit
+   18.80%     0.02%  mysqld              [.] MYSQL_BIN_LOG::ordered_commit
-   18.61%    17.89%  mysqld              [.] ut_delay
   - 2.77% start_thread
      - 1.89% pfs_spawn_thread
           handle_connection
           do_command
           dispatch_command
+   13.50%     0.03%  [kernel]            [k] system_call_fastpath
+   11.22%     0.00%  libpthread-2.17.so  [.] start_thread
+   11.15%     0.00%  mysqld              [.] row_update_for_mysql
+   10.97%     0.00%  mysqld              [.] mtr_t::commit
+   10.83%     0.02%  mysqld              [.] row_upd
+   10.80%     0.57%  mysqld              [.] mtr_t::Command::execute
+   10.36%     0.00%  mysqld              [.] row_upd_step
+    9.65%     0.03%  mysqld              [.] handler::ha_update_row
+    9.21%     0.06%  mysqld              [.] Sql_cmd_delete::mysql_delete
+    9.06%     0.05%  mysqld              [.] Sql_cmd_insert::execute
+    9.06%     0.00%  mysqld              [.] Sql_cmd_delete::execute
+    8.99%     0.68%  mysqld              [.] mtr_t::Command::prepare_write
+    8.70%     0.00%  mysqld              [.] pfs_spawn_thread
+    8.67%     0.11%  mysqld              [.] Sql_cmd_insert::mysql_insert
+    8.21%     0.14%  mysqld              [.] ha_innobase::update_row
+    8.17%     0.50%  mysqld              [.] btr_cur_search_to_nth_level
+    7.93%     0.02%  mysqld              [.] MYSQL_BIN_LOG::change_stage
+    7.31%     0.03%  mysqld              [.] QUICK_RANGE_SELECT::get_next
+    7.21%     0.04%  mysqld              [.] handler::multi_range_read_next
+    7.14%     0.10%  mysqld              [.] ha_innobase::index_read
     6.93%     0.02%  mysqld              [.] handler::read_range_first
+    6.85%     0.01%  mysqld              [.] handler::ha_write_row
+    6.85%     0.18%  mysqld              [.] row_search_mvcc
     6.80%     0.00%  mysqld              [.] handler::ha_index_read_map
```

可以看到，其中占 CPU 消耗占比较大为：ut_delay 函数。

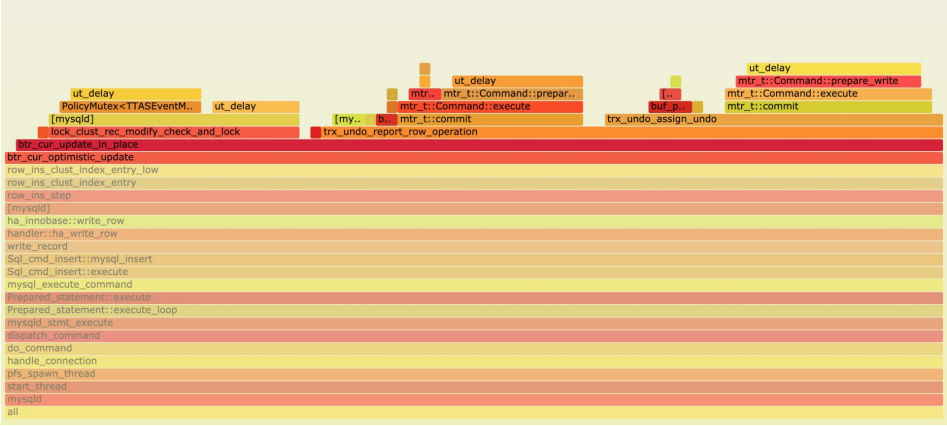
我们继续深挖一下函数链调用关系：

```
# Children      Self  Command  Shared Object
Symbol

# ........  ........  .......  ..................  ..................
.....................................................................
.....................................................................
.....................
#
    93.54%     0.00%  mysqld   libpthread-2.17.so   [.] start_thread
```

```
            |
            ---start_thread
               |
               |--77.07%--pfs_spawn_thread
               |          |
               |           --77.05%--handle_connection
               |                     |
               |                      --76.97%--do_command
               |                                |
               |                                |--74.30%--dispatch_
command
               |                                |          |
               |                                |
|--71.16%--mysqld_stmt_execute
               |                                |          |          |
               |                                |          |
--70.74%--Prepared_statement::execute_loop
               |                                |          |
|
               |                                |          |
|--69.53%--Prepared_statement::execute
               |                                |          |
|          |
               |                                |          |
|          |--67.90%--mysql_execute_command
               |                                |          |
|          |          |
               |                                |          |
|          |          |--23.43%--trans_commit_stmt
               |                                |          |
|          |          |          |
               |                                |          |
|          |          |           --23.30%--ha_commit_trans
               |                                |          |
|          |          |                     |
               |                                |          |
|          |          |                     |--18.86%--MYSQL_BIN_
LOG::commit
               |                                |          |
|          |          |                     |          |
               |                                |          |
|          |          |                     |           --18.18%--MYSQL_
BIN_LOG::ordered_commit
               |                                |          |
|          |          |                     |                     |
               |                                |
|                     |          |          |                     |
|--8.02%--MYSQL_BIN_LOG::change_stage
               |                                |          |
```

```
|          |          |                    |                    |
|
              |                              |          |
|          |          |                    |                    |
|--2.35%--__lll_unlock_wake
              |                              |          |
|          |          |                    |                    |
|          |
              |                              |          |
|          |          |                    |                    |
|           --2.24%--system_call_fastpath
              |                              |          |
|          |          |                    |                    |
|                     |
              |                              |          |
|          |          |                    |                    |
|                      --2.24%--sys_futex
              |                              |          |
|          |          |                    |                    |
|                                |
              |                              |          |
|          |          |                    |                    |
|                                --2.23%--do_futex
              |                              |          |
|          |          |                    |                    |
|                                          |
              |                              |          |
|          |          |                    |                    |
|                                           --2.14%--futex_wake
              |                              |          |
|          |          |                    |                    |
|                                                    |
              |                              |          |
|          |          |                    |                    |
|                                                     --1.38%--wake_
up_q
              |                              |          |
|          |          |                    |                    |
|                                                                |
              |                              |
|                     |          |
|                     |                    |          |
--1.33%--try_to_wake_up
              ...
```

将上述调用通过火焰图进行直观展示：



现在基本可以确定，所有的函数调用，最后大部分的消耗都在 ut_delay 上。

3.2 ut_delay 和 PAUSE 之间的关联与性能影响

3.2.1 MySQL ut_delay 实现

接下来，我们继续看一下 MySQL 源码中 ut_delay 函数的功能：

```
/**********************************************************//**
Runs an idle loop on CPU. The argument gives the desired delay
in microseconds on 100 MHz Pentium + Visual C++.
@return dummy value */
ulint
ut_delay(
/*=====*/
  ulint delay)  /*!< in: delay in microseconds on 100 MHz Pentium */
{
  ulint i, j;

  UT_LOW_PRIORITY_CPU();

  j = 0;

  for (i = 0; i < delay * 50; i++) {
    j += i;
    UT_RELAX_CPU();
  }

  UT_RESUME_PRIORITY_CPU();

  return(j);
```

```
}
...

#   define UT_RELAX_CPU() asm ("pause" )
#   define UT_RELAX_CPU() __asm__ __volatile__ ("pause")
```

可以了解到，MySQL 自旋会调用 PAUSE 指令，从而提升 spin-wait loop 的性能。

3.2.2 PAUSE 指令周期的演变

我们可以看下 Intel 官网，也描述了在新平台架构 PAUSE 的改动：

> Pause Latency in Skylake Microarchitecture
>
> The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.
>
> ...
>
> The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake microarchitecture it has been extended to as many as 140 cycles.
>
> The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have

negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.
As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.
...

- 上一代架构中（Grantly 平台 E 系列）PAUSE 的周期时长为 10 cycles，新一代的 Skylake 架构中则为 140 cycles。
- 如果程序中使用固定次数的 PAUSE 循环来实现一段时间的延迟，以此阻塞程序执行，可能引发非预期的延迟。
- 由于 PAUSE 周期增加，对于 PAUSE 敏感的应用会有一定的性能损失。

衡量程序执行性能的简化公式：

ExecutionTime(T)=InstructionCount*TimePerCycle*CPI

即：程序执行时间 = 程序总指令数 x 每 CPU 时钟周期时间 x 每指令执行所需平均时钟周期数。

MySQL 内部自旋，就是通过固定次数的 PAUSE 循环实现。可知，PAUSE 指令周期的增加，那么执行自旋的时间也会增加，即程序执行的时间也会相对增加，对系统整体的吞吐量就会有影响。

显然，Intel 文档已说明不同平台、不同架构 CPU PAUSE 定义的周期是不一样的。

下面，我们通过一个测试用例来大致验证、对比一下新老架构 CPU 执行 PAUSE 的 cycles：

```
 #include <stdio.h>
#define TIMES 5

static inline unsigned long long rdtsc(void)
{
    unsigned long low, high;
```

```
    asm volatile("rdtsc" : "=a" (low), "=d" (high) );
    return ((low) | (high) << 32);
}

void pause_test()
{
    int i = 0;
    for (i = 0; i < TIMES; i++) {
        asm(
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"
                ::
                :);
    }
}

unsigned long pause_cycle()
{
    unsigned long start, finish, elapsed;
    start = rdtsc();
    pause_test();
    finish = rdtsc();
    elapsed = finish - start;
    printf("Pause 的 cycles 约为 :%ld\n", elapsed / 100);
    return 0;
}

int main()
{
    pause_cycle();
    return 0;
}
```

其运行结果统计如下：

CPU	Cycles	备注
4110	122	• 4110 、5118 跟E5-2620 V4 相差超过10倍。 • 5218 跟E5-2620 V4 相差约4倍。
5118	119	
5218	35	
4210	32	
E5-2620 V4	8	

- 4110 和 5118 PAUSE 周期较大，均为 100 多，它们属于 Purley 第一代架构：Skylake。
- 4210 和 5218 PAUSE 相比前一代有提升，是因为它们同属 Purley 第二代架构：Cascadelake，该代 CPU PAUSE 指令有优化。

3.2.3 Intel 提升 PAUSE 猜想

Intel 提高 PAUSE 指令周期的原因，推测可能是减少自旋锁冲突的概率，以及降低功耗；但反而导致 PAUSE 执行时间变长，降低了整体的吞吐量。

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor read to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions.

3.3 PAUSE 导致写瓶颈分析

接下来，我们深入分析一下 PAUSE 指令导致 MySQL 写瓶颈的原因。

首先，通过 MySQL 内部统计信息，查看一下 InnoDB 信号量监控数据：

```
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 153720
--Thread 139868617205504 has waited at row0row.cc line 1075 for 0.00
seconds the semaphore:
X-lock on RW-latch at 0x7f4298084250 created in file buf0buf.cc line 1425
a writer (thread id 139869284108032) has reserved it in mode  SX
number of readers 0, waiters flag 1, lock_word: 10000000
Last time read locked in file not yet reserved line 0
Last time write locked in file /mnt/workspace/percona-server-5.7-redhat-
binary-rocks-new/label_exp/min-centos-7-x64/test/rpmbuild/BUILD/percona-
server-5.7.26-29/percona-server-5.7.26-29/storage/innobase/buf/buf0flu.cc
line 1216
OS WAIT ARRAY INFO: signal count 441329
RW-shared spins 0, rounds 1498677, OS waits 111991
RW-excl spins 0, rounds 717200, OS waits 9012
RW-sx spins 47596, rounds 366136, OS waits 4100
Spin rounds per wait: 1498677.00 RW-shared, 717200.00 RW-excl, 7.69 RW-sx
```

可见写操作并阻塞在：storage/innobase/buf/buf0flu.cc 第 1216 行调用上。

跟踪一下发生等待的源码：buf0flu.cc line 1216：

```
    if (flush_type == BUF_FLUSH_LIST
        && is_uncompressed
        && !rw_lock_sx_lock_nowait(rw_lock, BUF_IO_WRITE)) {    // 加锁前，
判断锁冲突

        if (!fsp_is_system_temporary(bpage->id.space())) {
        /* avoiding deadlock possibility involves
         doublewrite buffer, should flush it, because
         it might hold the another block->lock. */
        buf_dblwr_flush_buffered_writes(
          buf_parallel_dblwr_partition(bpage,
                flush_type));
      } else {
        buf_dblwr_sync_datafiles();
      }
      rw_lock_sx_lock_gen(rw_lock, BUF_IO_WRITE);        //  加 sx 锁
    }
...
 #define rw_lock_sx_lock_nowait(M, P)        \
  rw_lock_sx_lock_low((M), (P), __FILE__, __LINE__)
...

rw_lock_sx_lock_func(                                    // 加 sx 锁函数
/*=================*/
```

```
  rw_lock_t*  lock, /*!< in: pointer to rw-lock */
  ulint   pass, /*!< in: pass value; != 0, if the lock will
        be passed to another thread to unlock */
  const char* file_name,/*!< in: file name where lock requested */
  ulint   line) /*!< in: line where requested */

{
  ulint   i = 0;
  sync_array_t* sync_arr;
  ulint   spin_count = 0;
  uint64_t  count_os_wait = 0;
  ulint   spin_wait_count = 0;

  ut_ad(rw_lock_validate(lock));
  ut_ad(!rw_lock_own(lock, RW_LOCK_S));

lock_loop:

  if (rw_lock_sx_lock_low(lock, pass, file_name, line)) {

    if (count_os_wait > 0) {
      lock->count_os_wait +=
        static_cast<uint32_t>(count_os_wait);
      rw_lock_stats.rw_sx_os_wait_count.add(count_os_wait);
    }

    rw_lock_stats.rw_sx_spin_round_count.add(spin_count);
    rw_lock_stats.rw_sx_spin_wait_count.add(spin_wait_count);

    /* Locking succeeded */
    return;

  } else {

    ++spin_wait_count;

    /* Spin waiting for the lock_word to become free */
    os_rmb;
    while (i < srv_n_spin_wait_rounds
           && lock->lock_word <= X_LOCK_HALF_DECR) {

      if (srv_spin_wait_delay) {
        ut_delay(ut_rnd_interval(
            0, srv_spin_wait_delay));                          // 加锁失败,
调用 ut_delay
      }

      i++;
    }
```

```
    spin_count += i;

    if (i >= srv_n_spin_wait_rounds) {

      os_thread_yield();

    } else {

      goto lock_loop;
    }
...
ulong srv_n_spin_wait_rounds   = 30;
ulong srv_spin_wait_delay = 6;
```

上述源码可知，MySQL 锁等待是通过调用 ut_delay 做空循环实现的。

InnoDB 层有三种锁：S（共享锁）、X（排他锁）和 SX（共享排他锁）。 SX 与 SX、X 是互斥锁。加 SX 不会影响读，只会阻塞写。所以在大量写入操作时，会造成大量的锁等待，即大量的 PAUSE 指令。

分析到这里，我们总结一下影响吞吐量的两个因素：

- 自旋的时长，在 MySQL5.7 以及之前版本的源码定位为：spin_wait_delay * 50。
- Intel CPU PAUSE 的指令周期。

接下来，我们就从这两方面入手，评估优化空间以及效果。

4. 针对 PAUSE 指令和 spin 参数优化与探索

4.1 MySQL spin 参数优化

4.1.1 MySQL 5.7 spin 参数优化

我们可以基于现有 MySQL 版本、硬件等方面，来寻找优化点。

MySQL 针对 spin 控制这块有个参数可以调整，根据参数特点进行相关优化：

innodb_spin_wait_delay

innodb_spin_wait_delay 的单位，是 100MHZ 的奔腾处理器处理 1 毫秒的时间，默认 innodb_spin_wait_delay 配置成 6，表示最多在 100MHZ 的奔腾处理器上自旋 6 毫秒。

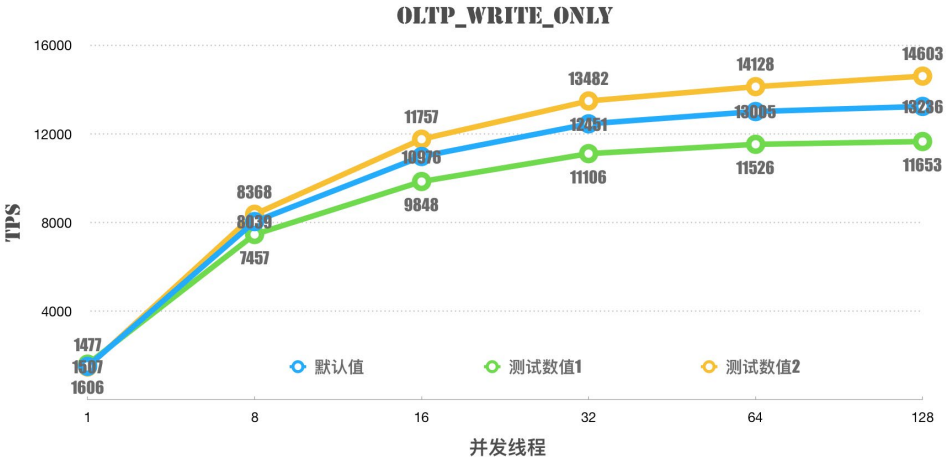
innodb_sync_spin_loops

当 innodb 线程获取 mutex 资源而得不到满足时，会最多进行 innodb_sync_spin_loops 次尝试获取 mutex 资源。

其中 innodb_spin_wait_delay 参数对 PAUSE 运行时长是有影响的。针对此参数，我们进行调优测试。

具体项	默认	测试数值 1	测试数值 2
innodb_spin_wait_delay	6	12	3
innodb_sync_spin_loops	30	30	20

同样，针对上述参数优化，我们通过基准测试来对比性能和效果：



可以总结为：

- innodb_spin_wait_delay 的调整对 TPS、QPS 一定影响，其值趋于小，则 MySQL 性能有提升。反之，下降。
- innodb_spin_wait_delay 参数调整性能优化效果有限，性能提升的幅度还是无法满足线上业务需求。

4.2 MySQL8.0 spin 新特性移植

4.2.1 spin_wait_pause_multiplier 移植

针对 Skylake CPU，PAUSE 造成的吞吐量下降，我们对 MySQL 5.7 spin 控制参数 innodb_spin_wait_delay 的调优并未取得明显效果。

于是，我们将目光投向了 MySQL 8.0 的新特性：MySQL 8.0 针对 PAUSE，源码中新增了 spin_wait_pause_multiplier 参数，来替换之前写死的循环次数。

4.2.2 spin_wait_pause_multiplier 实现

MySQL 8.0 源码中，之前循环 50 次的逻辑修改成了可以调整循环次数的参数：spin_wait_pause_multiplier。

```
ulint ut_delay(ulint delay) {
  ulint i, j;
```

```
  /* We don't expect overflow here, as ut::spin_wait_pause_multiplier
is limited
  to 100, and values of delay are not larger than @@innodb_spin_wait_
delay
  which is limited by 1 000. Anyway, in case an overflow happened,
the program
  would still work (as iterations is unsigned). */
  const ulint iterations = delay * ut::spin_wait_pause_multiplier;
  UT_LOW_PRIORITY_CPU();

  j = 0;

  for (i = 0; i < iterations; i++) {
    j += i;
    UT_RELAX_CPU();
  }

  UT_RESUME_PRIORITY_CPU();

  return (j);
}
...
namespace ut {
ulong spin_wait_pause_multiplier = 50;
}
```

4.2.3 移植 spin_wait_pause_multiplier patch 优化

既然 MySQL 8.0 参数 spin_wait_pause_multiplier 可以控制 PAUSE 执行的时长，那么就可以减少该值，从而降低整体 PAUSE 影响。

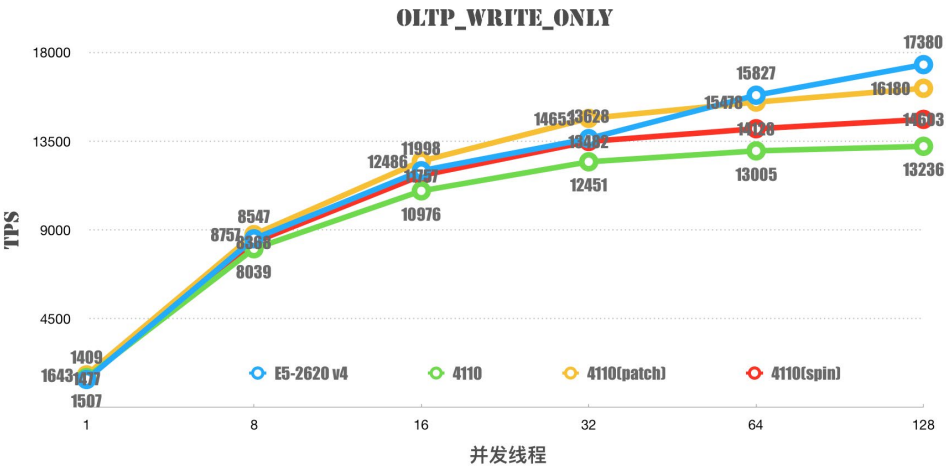
了解 MySQL 8.0 相关代码后，我们将该 patch 移植到线上的稳定版本：

```
MySQ >select version();
+-------------------+
| version()         |
+-------------------+
| 5.7.26-29-mt-log |
+-------------------+
1 row in set (0.00 sec)

MySQL>show global variables like '%spin%';
+----------------------------------+-------+
| Variable_name                    | Value |
+----------------------------------+-------+
| innodb_spin_wait_delay           | 6     |
```

```
| innodb_spin_wait_pause_multiplier | 5     |
| innodb_sync_spin_loops            | 30    |
+-----------------------------------+-------+
3 rows in set (0.00 sec)
```

由上述可知，Silver 4110 的 PAUSE cycles 是 E5-2620 v4 的 14 倍左右。基于此，将 innodb_spin_wait_pause_multiplier 值调整为默认值的 1/14，取稍大值：5。即将该参数由原默认的 50 调整为 5。

最后，还是通过二维折线图来对比该 patch 调优后的基准测试数据：



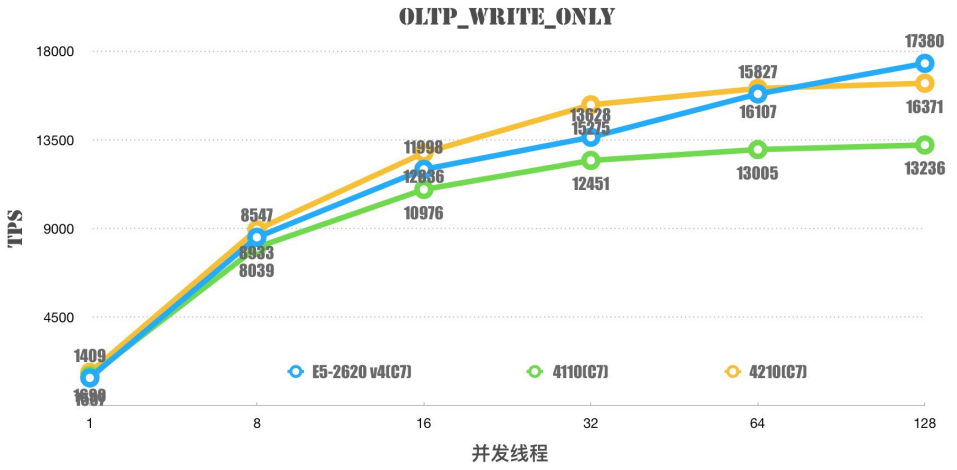
- Silver 4110 移植 spin_wait_pause_multiplier patch，并调整优化后，4110（patch）性能有了较大的提升。
- Silver 4110（patch）相对调优 innodb_spin_wait_delay 性能上更优。
- Silver 4110（patch）并发线程大于 64 的只写场景，性能略低于 E5-2620 V4，其他均优。
- 按照真实的线上读写比例，4110（patch）可以将吞吐量恢复到原先的性能水平。

4.3 PAUSE 指令周期优化

上述章节中，我们测出 Cascadelake CPU PAUSE 周期下降了。在跟 Intel 技术专家确认后得知：从 Purley 的第二代产品 Cascadelake 开始，Intel 将 PAUSE 的

指令周期降低到了 44。（估计 Intel 也发现了第一代增加 PAUSE 周期后的性能瓶颈问题。）

我们针对第二代 CPU 产品继续做基准测试，来看一下性能表现：



接着用 perf diff 来对比一下 4110 和 4210 在 ut_delay 上的开销：

```
# Baseline  Delta Abs  Shared Object                      Symbol
# ........  .........  .................................  .....................................
#
    15.62%     -8.01%  mysqld                             [.] ut_delay
               +1.27%  [kernel.kallsyms]                  [k] native_safe_halt
     0.06%     +0.74%  [kernel.kallsyms]                  [k] reschedule_interrupt
     0.80%     +0.59%  [kernel.kallsyms]                  [k] native_queued_spin_lock_slowpath
     1.25%     +0.58%  libpthread-2.17.so                 [.] __pthread_mutex_cond_lock
     1.49%     +0.43%  libpthread-2.17.so                 [.] pthread_mutex_lock
     6.92%     +0.37%  mysqld                             [.] 0x000000000036a168
               +0.28%  [kernel.kallsyms]                  [k] mlx5_eq_int
     0.58%     +0.23%  mysqld                             [.] mtr_t::Command::prepare_write
     0.94%     +0.23%  [kernel.kallsyms]                  [k] __schedule
               +0.21%  [kernel.kallsyms]                  [k] mlx5e_poll_tx_cq
     0.74%     +0.21%  [kernel.kallsyms]                  [k] _raw_spin_lock_irqsave
               +0.20%  [kernel.kallsyms]                  [k] call_function_interrupt
     0.29%     +0.19%  [kernel.kallsyms]                  [k] native_write_msr_safe
     0.02%     +0.16%  [kernel.kallsyms]                  [k] scheduler_ipi
               +0.15%  [kernel.kallsyms]                  [k] smp_call_function_many
               +0.15%  [kernel.kallsyms]                  [k] mlx5e_napi_poll
               +0.14%  [kernel.kallsyms]                  [k] eq_update_ci.isra.4
     0.05%     +0.13%  [kernel.kallsyms]                  [k] llist_add_batch
     0.03%     +0.13%  [kernel.kallsyms]                  [k] __x2apic_send_IPI_mask
     0.47%     +0.13%  [kernel.kallsyms]                  [k] _raw_qspin_lock
               +0.12%  [kernel.kallsyms]                  [k] mlx5_cmd_comp_handler
```

- 可以看到 4210 比 4110 占比下降了 8%。
- 由于 PAUSE 指令周期还是数倍于 E5 系列 CPU，4210 在高负载下，

PAUSE 的开销对 MySQL 吞吐量还是有较大的影响。而在 128 并发线程以下，性能相比 4110 有了较大的提升。按理，可以满足线上业务需求（该测试结果跟移植 spin_wait_pause_multiplier patch 性能测试数据曲线一致）。

5. 总结

最后针对本篇内容，我们可以做个简单的总结：

Intel 在新平台 CPU 产品调大了 PAUSE 指令周期，在高并发 spinlock 竞争激烈场景下，可能会造成程序性能较大损耗（特别是执行固定 PAUSE 次数的程序）。 针对 Skylake 架构 CPU（比如：4110 等）PAUSE 指令周期较长引起性能问题的优化方法如下：

将 MySQL 8.0 innodb_spin_wait_pause_multiplier patch 移植到线上稳定版本（或升级到 MySQL 8.0），通过降低 PAUSE 执行时长，来提升吞吐量。 如果是 OS 为 CentOS 6，可以升级到 CentOS 7，CentOS 7 本身 spinlock 优化，对 MySQL 性能也有一定提升。 最简单、直接的方法可以替换为 Cascadelake 架构 CPU。

针对 Cascadelake 架构 CPU，由于 Intel 本身在 PAUSE 周期已经优化，性能上已经做了修复。当然也可以采用上述优化方案，让性能提升一个台阶。

6. 作者简介

春林，2017 年加入美团，主要负责 MySQL 运维开发和优化工作。

招聘信息

美团 DBA 团队招聘各类人才，Base 北京、上海均可。我们致力于为公司提供稳定、可靠、高效的在线存储服务，打造业界领先的数据库团队。这里有数万各类架构的 MySQL 实例，每天提供万亿级的 OLTP 访问请求。真正的海量、分布式、高并发环境。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：美团 DBA 团队）

美团内部讲座｜周烜：华东师范大学的数据库系统研究

作者：周烜

【Top Talk/ 大咖说】由美团技术学院和科研合作部主办，面向全体技术同学，定期邀请美团资深技术专家、业界大咖、高校学者及畅销书作者，为大家分享最佳实践、互联网热门话题、学术界前沿技术进展等内容，帮助美团同学开拓视野、提升认知。2020 年 10 月 27 日，Top Talk 邀请到了华东师范大学周烜老师，请他带来题为《华东师范大学的数据库系统研究》的分享。本文系周烜老师分享报告的文字版，希望能对大家有所帮助或者启发。

/ 报告嘉宾 /



周烜

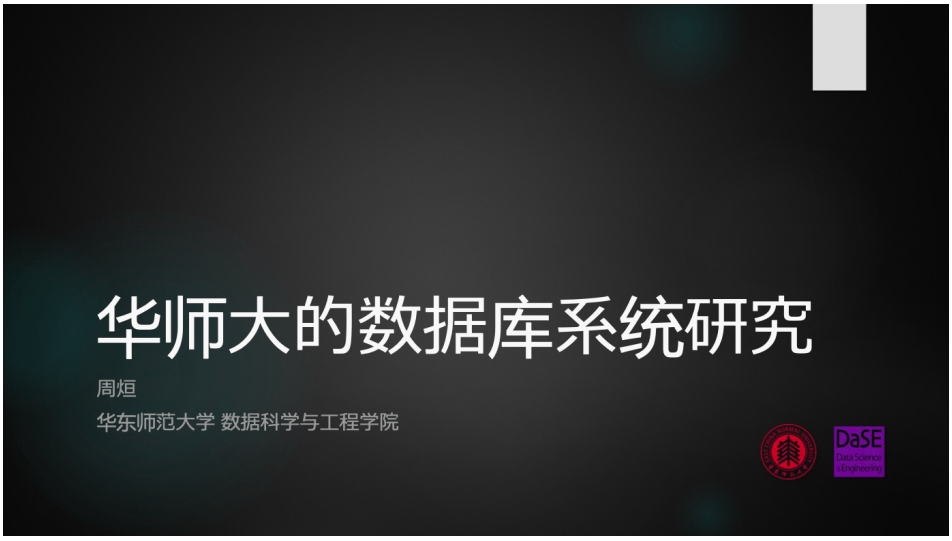
华东师范大学教授，数据科学与工程学院副院长

2001年本科毕业于复旦大学，2005年在新加坡国立大学取得博士学位，2005年至2010年期间先后在德国L3S研究中心和澳大利亚联邦科工组织从事科研工作，随后在中国人民大学信息学院任教6年，于2017年3月加入华东师范大学。研究兴趣包括数据库系统和信息检索技术。曾参与和负责多个国内外的科研项目和工业合作项目，积累了丰富的数据管理系统研发经验。研究成果被发表于众多国际一流的学术会议和期刊。凭借在分布式数据库领域的成果转化获得国家科技进步二等奖和教育部科技进步一等奖。入选教育部“新世纪优秀人才”支持计划。

/ 报告摘要 /

华东师范大学是国内为数不多长期坚持数据库内核技术研究的高校，在学术界和工业界均建立了较好的声誉。本次讲座将分享华东师范大学数据库团队近期的一些科研思路和研究成果。首先分析驱动数据库技术发展的主要因素，谈一谈未来有价值的研究方向。再聊一聊团队近来取得的一些有趣的研究成果，领域包括新硬件的数据库适配、分布式事务处理、HTAP、系统实现模块化（Modularization）等等。

00 引言



今天我代表华东师范大学的数据库团队，来分享一下对数据库这种技术或者这种产品的一些研究心得以及当前的研究成果。其实，高校跟企业实际上是处在两个不同的生态领域当中，高校更关注关于研究理论的一些问题，但是企业更多的关注企业本身的产品以及用户，所以两者面向的目标是不太一样的。但是经过我们在华师大这么多年的一些摸索，特别是我们自己研究上的一些摸索，以及跟企业合作的经历，我们觉得实际上高校和企业应该一起来做我们称为数据库系统或者基础软件的研究，因为只有这样才能够更好的推动这个行业本身的发展。

我的报告内容会分成两个部分。首先分享一下我们对数据库这种技术发展动态的看法，我们团队在数据库这个领域也做了很多年，包括我之前在中国人民大学也是做数据库系统的，我在这个领域里面有可能不到 20 年的积累。我希望能够提出我们的看法，并且能够得到大家的一些反馈，纯粹是做一种探讨，因为对技术的发展方向的探索，没有标准的答案。我觉得大家应该集思广益，共同去探讨，才能把这个问题看得更清楚，这是讲座前面一部分的内容。

后面一部分的内容我会聚焦到我们的一些研究成果上，这些研究成果可能就会比较技

术细节了，会更适合搞技术、数据库、系统的这些同学，但是我会尽量把讲座的形式变得更大众化一点，尽量用更通俗的方式去给大家介绍。我希望通过这种细节的介绍，也能够让大家了解一下，我们在设计系统的时候，通常一个工程师、一个研究者或者一个学者，他的思路大概是怎么样的，我不能代表所有的人，但是因为我们这个团队是一个典型的做系统的团队，所以这个思路可能仅代表了一部分做系统的学者的思路。

01 数据库系统的形态变化

1.1 什么引起了数据库系统的形态变化?

首先问大家一个问题，什么引起了数据库系统的形态变化？我们知道数据库系统实际上是一个有很长历史的系统，是现代软件开发的一个核心部件。任何应用都离不开某种数据库，但是我们其实也可以看到，如果你有一定的经验，比如大概 10 年的工作经验，你会看到数据库系统的形态是在发生变化的，10 年前用的很流行的东西，现在不见得普遍被采用。这个系统虽然有很长很长的历史，也很成熟了，但是它的形态还是在发生变化的。

什么在驱动数据库系统的形态变化？这是一个很重要的问题，也是比较有趣的问题，但并不是一个好回答或者能够全面的回答的问题。我这里就直接抛出我们对这个问题的一些看法，我们觉得数据库系统的形态变化，主要来自三个方面的推动力：

- 第一个方面是应用需求的变化。我们的软件产品一直在变得越来越丰富，应对的场景越来越多，实际上应用需求是在变化的，它的变化提出了不同的要求需求，它在促使数据库系统的形态在发生改变。
- 第二个方面在学术界其实比较少去触及的，就是软件开发模式的变化。数据库是 70 年代 80 年代设计的，面对的是当时的软件开发模式，我们知道在九十年代以后，软件开发模式实际上发生了很大的改变。软件开发的模式的变化，也在引起数据库使用方式的改变，也在推进数据库系统的演变。
- 第三个方面就是硬件平台的革新，硬件在改变处理器、存储器件，整个平台以

前是一个大型的计算机，大型机、中型机，然后现在其实就是云平台，这些东西的变化实际上是在影响数据库本身的形态的改变。

我们看到的推动力主要就是这三个。我们认为未来推动数据库变化的原因也不出于这三个，我们可以通过这个东西去预知未来应该朝什么方面去推进我们的数据库技术的进步。下面我就大概做一个简单的展开。

1.1.1 应用需求的变化

首先第一点就是应用需求的变化。我们认为这实际上是在推动底层技术，像数据库系统这种技术变化的一个主要原因。



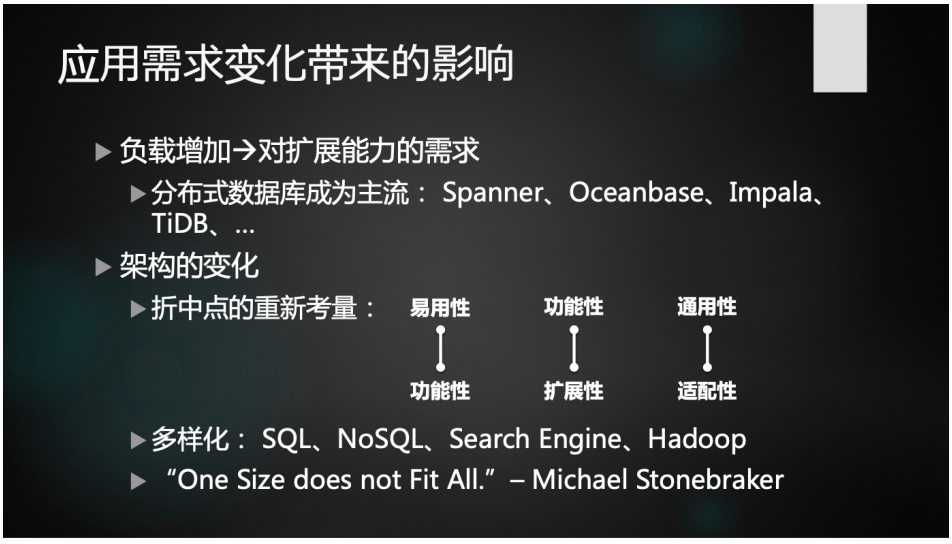
通过上面这三幅图大家能够很容易去理解数据库诞生的年代，那个时候磁盘作为一个存储介质，刚刚在市场上推广，磁盘取代了磁带，数据的随机访问变的可能了。那个时候对数据管理功能的需求一下子就增长了，当时出现了各种各样的数据库，包括网状数据库，包括后面的关系数据库，那个时候是我们叫前互联网时代就出来了。但那个时候应用的规模并不大，数据库的用户量一般，终端用户其实很少，对于一个银行来讲就是那些银行职员在使用数据库，普通用户在银行排队，他们不是终端的使用者。

后来进入到互联网时代之后，我们发现终端的使用者一下子就爆炸性的增长了。现在我们每个人有一个手机，随时都在使用手机上的 App，然后这个 App 他随时都会把请求发送给后台的数据库。我们看到应用规模在最近 20 年有一个很大的增长，如果

往后看的话，未来我们认为增长有可能还会继续。我们的工业互联网、物联网使用的话，我们的终端会变得更多，他可能对数据管理系统的压力会变得更大。

所以我们看到应用不断的扩张，给数据库这种底层系统有一个持续增长的压力，要应对这样的压力，以前对数据库的设计，它逐渐就变得不太实用，必须去革新，必须去改变它。

这个是我们看到的一个最主要的推动力。但除了应用规模的扩展，当然就还有一个应用领域的扩展，最开始的数据库它就是金融领域或者电信领域去使用，并不会在互联网、销售或者传媒等等这些领域去大规模的使用，但是我们发现 IT 的渗透到各行各业之后，它的应用范围也在增加，应用范围增加对传统的系统来讲是不友好的，或者是说你的传统数据库系统对这些应用是不友好的，所以这个也在推动它的一个变革。



应用需求带来什么变化？我们回顾历史的话，我们看到分布式数据库现在变得越来越被大家所需要，大家看到了谷歌的一些分布式数据库的产品，它现在作为一种标杆的产品，然后国内也有一些分布式数据库的场景，包括美团在内，听说美团内部也在研制自己的分布式数据库，实际上是在对需求的负载增加的一个应对。

应对它实际上并不是一个很简单的事情，如果要增加你的数据库的扩展性，有时候你

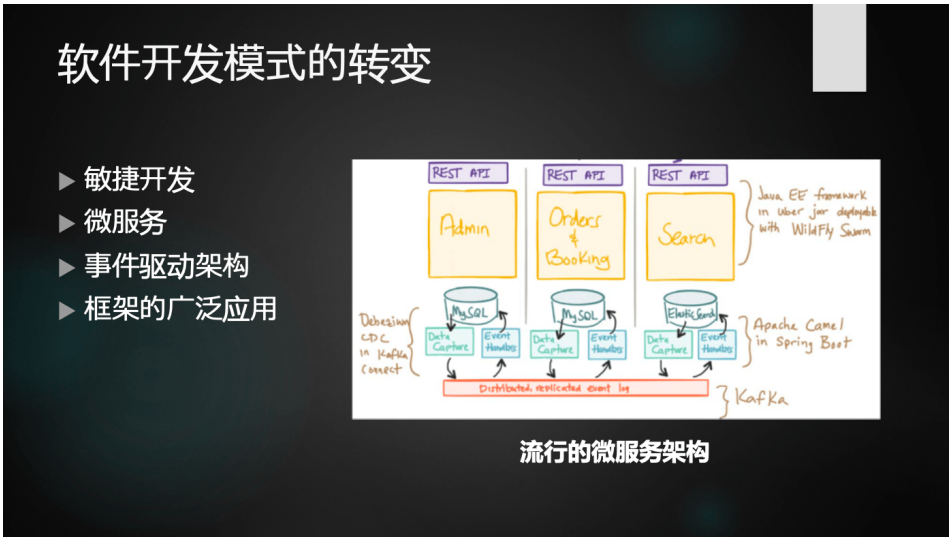
必须要重新去设计你的数据库的架构。我们通常做系统的同学应该了解，其实在做系统的时候，你需要做很多折中的考量的，有些东西是不能兼顾的，比如说你的功能性和应用性对吧？一个东西特别简单去使用的话，它功能性有时候就是比较简单，它复杂的功能处理不了，或者是你的功能很复杂的时候，你的扩展性又上不去。

在系统设计的时候你必须去做一种权衡，去获得一部分这方面的能力，你就必须丢失其他的一些能力。当你需要它的扩展性非常强的时候，你有可能就要去重新考量它，你要去丢弃什么东西，你要去忍受什么其他的一些东西，然后这个时候就产生架构的变化，这样的话我们就会发现有新的系统出来，比如说以前的是 SQL，我们现在有讲 NoSQL 大家认为它的扩展性容易做的更好一些，然后会有其他形态的一些数据管理产品。

针对这个问题，学界工业界都有很多的讨论，Michael Stonebrake 大概 10 多年前发表的一些言论，就说“One size does not fit all”，你不太能指望某一个系统能够能够处理所有的应用需求。因为不同的应用需求，你可能必须要做不同的折中、重新的考量，你只能顾此，你顾此只能失彼，所以这样就产生了一个多样化的形态。

我们现在看到的数据库，如果你们在使用的话，你会面临很多选择，你到底用 MySQL 还是 MongoDB 对吧？你的分析的时候你要用什么？用 Hadoop 还是用传统的数仓 MPP 产品，有不同的需求，可能有不同的考量。这个我们看到应用需求它的变化，它的增加，它在实际上对这个系统起到了一个很重要的推动的这个作用，这是我展开的第一点。

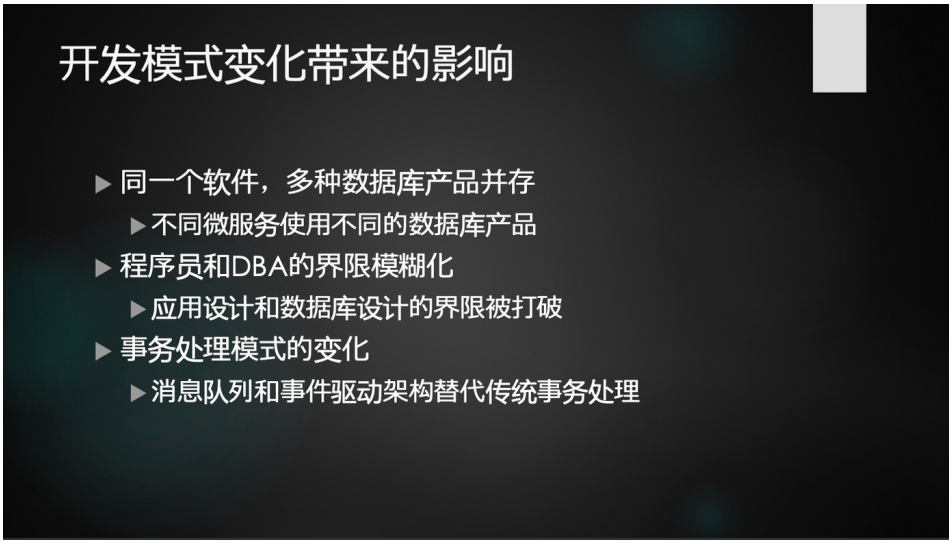
1.1.2 软件开发模式的转变



流行的微服务架构

然后，第二点就是我们发现软件开发模式的变化，也在促进数据库本身的改变。就像我刚才提到的，现在用到的很多软件开发的模式跟以前不一样。以前关系数据库这样的产品刚刚被广泛应用的时候，当时的软件开发它是以数据为中心的，一个数据库设计出来，有好多应用都会去用它，它是一个 Shared 底层系统。那个时候数据库的设计过程，它是相对独立的，DBA 根据 App 开发人员的需求、用户的需求，以 DBA 的方式去设计数据库，按照对数据模型的理解，把它设计得非常的规整，要满足各种各样的范式，然后给不同的应用去用。但现在我们发现一旦用到微服务这种新的开发模式的时候，很多时候这种横向的分割变得不是那么重要了。原本数据是一层、应用逻辑是一层，这两层之间的解耦是很重要的，但现在不是了。

现在是微服务的形式，是纵向的切割，把业务整个分成一块一块的，每一块里面都有单独的数据库，有单独的功能设计，这弱化了数据库跟应用之间的界限，强化了应用里面不同模块的界限。这样的话，每个模块可以用不同的数据库产品，比如说一个设备用 MySQL，另外一个设备可能用 MongoDB，第三个设备可能用 ES，这些模块之间的数据有同步有交互，可以用一些比如像事件驱动的架构，像 Kafka 这种 MQ (Message Queue) 去连接在一起，形成一个总体的架构。



这种设计跟以前的数据库是不太一样的，对于数据库本身的要求也不太一样。不同的 Service 会用不同形态的数据库产品，根据需求或者根据软件开发者的习惯，去采用各种缓存、消息队列等，去把这些东西给嫁接在一起。这样的形态对数据库有不同的要求，所以 NoSQL 被很多人接受。

在某些软件开发的场景下，NoSQL 就是比关系数据库使用起来更简单。然后事务的处理方式也变得很不一样，现在的消息队列在事务处理中，它的权重非常的高，而不是完全依赖于传统数据库内部支持事务处理的模式。这个是我们也看到这样的一些变化，这个是软件开发模式带来的一些改变。

1.1.3 硬件平台的革新



第三个方面就是平台的革新。其实我们不会对传统的数据库的硬件平台灵活性有太多的关注，但现在数据库产品面向的基本上都是云平台，不是以前的 IBM 大型机了，不是 Oracle 那个时候的硬件平台。我们面对的是一个云平台，云平台自身是在发展的。以后的云平台其实可以预见得到，它不单单是现在我们看到的，是由一个个虚拟机或者是一个个容器组成的一个计算平台，还有可能就像一个大型的计算机一样，只是这种大型计算机它的资源非常的丰富，需要调用什么样的资源都可以获得，需要更多的内存、CPU、存储，都可以直接的获取。

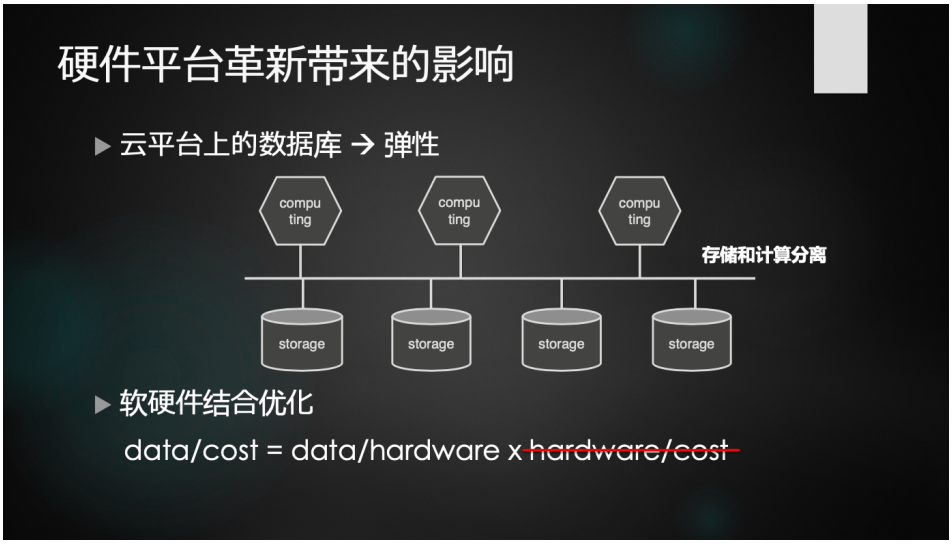
云平台通过比较高效的网络方式把这些资源全部连在一起，然后给用户的接口也很简单，很多维护功能是在云平台内部去实现的。数据库要扎根在这样的一个云平台上面，其实对数据库系统就会有新的要求。以前的数据库，大家都记得有几种架构可以选择，叫 Shared Everything、Shared Nothing、Shared Disk 这样的一些架构。

但是我觉得在“云”层面上，数据库其实不再是那么简单去划分的，就是说数据库系统的产品，必须要做到具有很好的弹性。任何的资源在短缺的时候，可以通过云的这

种方式很快的把资源调度过来，从而增加数据库的能力。对用户的话，只是提供一种数据库的服务，用户用多少？就提供多少。这样的一个云架构下的数据库，形态可能跟以前要有一些变化。除了这种云的体系之外，其实还有一些新硬件出现，硬件种类也变得越来越多。不同的种类的硬件，在不同的条件下，算力也在不断增加。当然除了计算，还有存储也发生很多变化，把这些放到云的里面去，云的资源变得更加丰富，对数据库的要求也会变得更高。

因此，我们看到这种硬件平台的变革，它实际上在推动数据库的一些发展。现在大家很常见的就是这种计算与存储分离的这种架构的数据库。我把数据库本身这种系统，它的计算层跟存储层完全的分割开，计算层可以自己扩展，存储层也可以自己扩展，两层就通过云这种高速的、互联的通道能够连接在一起，这种实际上就是针对云的一个特殊的处理。我们知道存储是便宜的，所以在存储需要扩展的时候，没有必要在扩展存储的基础上去加 CPU 的资源，因为 CPU 比较贵。如果分开扩展的话，这样确实会在成本上有极大的提升。

未来各种资源加进去之后，它都有可能有扩展的需求。比如说缓存，新的存储器，新的内存加进去之后，有可能需要它跟底层的存储分开，进行一个隔离，再分别去扩展，这都是有可能的。但现在的数据库产品其实面对这种扩展能力，实际上是非常有限的，存储和计算分开扩展到一定程度，实际上它的能力就达到一个峰值了。那么怎样去推动它进一步的这种弹性的增加，实际上是一个挺难的问题，但是也是挺有趣的问题。



新硬件对数据库产品的影响，这里有一个简单的公式，公式的左边叫做 Data/Cost，单位数据处理，单位代价上面可以处理多少数据。这个是我们需要提升的，因为可以想象以后的数据量会越来越大，如果不把单位价格上面能处理的数据这个值提升的话，应对数据的能力就没办法提升。数据越多，需要花费的资源或者代价就越多，这个是我们不希望看到的。希望左边这个式子中 Data/Cost 的值随着技术的进步，它可以逐渐的提升。

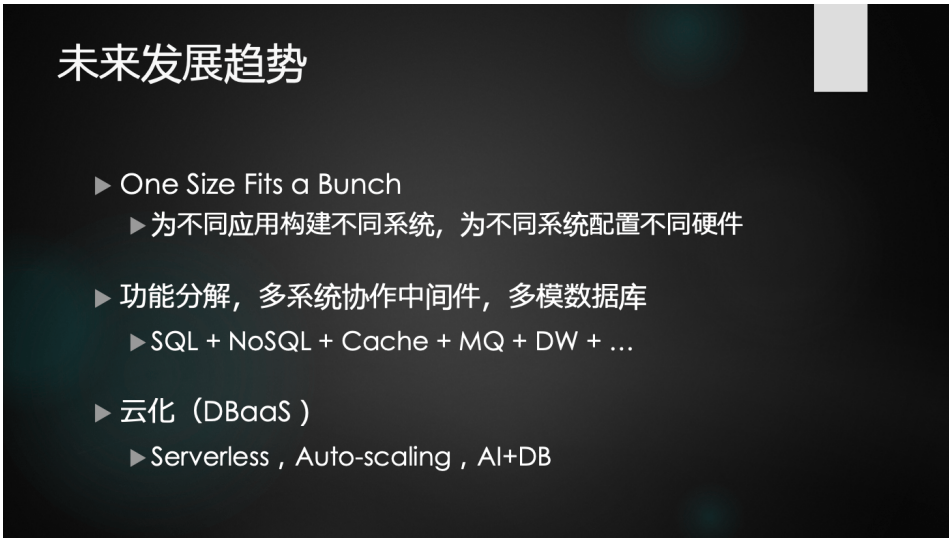
然后把左边这个式子它分解一下，分解成 Data/Hardware 和 Hardware/Cost 的一个乘积。这其实是很简单的一个因素分解，大家能够看得很明白。我们发现后面这个式子 Hardware/Cost，实际上它的增长在逐渐的趋缓甚至停滞，主要的一个问题就是单位价格能够买到的硬件资源，要在这上面做更进一步的提升，会变得非常的困难。

最后，如果想实现 Data/Cost 的提升的话，只能去提升左边这个式子 Data/Hardware，这个是未来一个很明显的趋势。怎么样能够提升 Data/Hardware？就是单位硬件下处理数据的能力要怎样提升？我们认为只有两种途径，第一种是硬件定制化，面对不同的应用需求，需要为这种应用需求做特殊的硬件。其实我们能看到像 GPU、TPU 这种出现，其实就已经在揭示这种规律了，专用硬件效率总是要比通用

硬件好的。然后软件是一样的，专用的软件的效率肯定比通用的软件好。

这个趋势我觉得可以从长期来看，短期可能并不是那么的明显，但长期来看的话，这个过程应该是不可阻挡的，也就是说我们可能会面临要去为应用去定制系统，要为专门的这种系统配置适合它的硬件。

1.2 数据库系统的未来发展趋势



我们刚才讲了三点，第一点就是应用的变化在推动系统的演进；第二个是软件开发模式的变化实际上也在带来系统的功能的一种变革；最后是硬件平台。因此，我们觉得未来数据库发展的趋势：

- 第一个是“One size fits a bunch”，为不同应用构建不同系统，为不同系统配置不同硬件。就像 Michael Stonebraker 说的那样，数据库不是一种系统就能应对所有的应用了。我觉得 Stenberg 当时是针对的是应用负载的增加带来的问题而提出的观点，但是硬件的发展瓶颈到来的时候，同样会引领我们认同这个观点，“One size does not fit all”，不可能为所有应用产生一个系统，应该是一类应用对应一套系统的，这个是我们看到的一个很明显的发展的趋势。

- 第二个发展的趋势是现在的系统变多了之后，它需要协同，会有各种中间件，还会有各种形态的数据库，我们需要把它协同起来，这些数据库形态太多，对程序开发人员，维护人员都是一个 Disaster，代价会变大，怎么更好地把它协同起来，这也是未来一个发展的方向，我们会看到这些系统相互之间会变得越来越配合，越来越融合。
- 第三个就是云平台成为一个主流的硬件平台之后，我们看到数据库会朝云这个方向有更深入这种发展，它会更适合云这种形态，它的弹性，自我维护、自我修复的能力是会进一步提升的。这个是我们对未来发展的展望。

02 华师大的数据库系统研究

2.1 研究团队



然后介绍一下我们现在这个团队，华东师范大学数据科学工程学院大概有 20 多个老师，大概 10 个老师是从事数据库内核的研究的。整个团队的历史是超过 20 年的，我们近 10 年其实做了非常多的系统内核研发的工作，跟业界的很多的公司也有合作。我们的学生其实也做了很多工程性的工作。

去年我们有拿到一个国家科技进步二等奖，这个是基于我们当时和某银行一起做的一款数据库产品，是我们基于 OceanBase 的一个早期开源的版本上实现的一个系统，具体的这种内容我就不做过多的介绍，这个系统实际上在某银行得到了比较深入的应用，是我们比较引以为豪的一个研究成果。

2.2 研究成果

我们团队的研究其实还是蛮广泛的，我们在事务型数据库和分析型数据库其实都有研究，但我们更多的精力还是集中在事务型数据库上面。然后接下来，我会介绍一些典型的研究成果，让大家了解一下我们的研究是在做一些什么事情，主要分三个部分，第一个是分布式事务，第二个是数据库系统解耦合，第三个是新硬件。

2.2.1 分布式事务

分布式事务是一个几十年来大家都在探讨的话题，实际上也是有一定的争论：分布式事务到底合不合用？我们在使用事务处理这种功能的时候，是不是应该去规避分布式事务？还是我们应该进一步去增强数据库支撑分布式事务的能力，让程序员不要刻意去规避分布式思维？这实际上是一个疑问，目前没有一个明确的答案。

如果分布式事务确实是不行的，那我们就应该做一些其他方面的处理，来弥补分布式事务本身的缺陷，这样做有两种方式：

- 第一种是干脆不要数据库提供分布式事务功能，将事务处理推给应用，根据应用的特点去规避这种分布式事务的一些缺陷。
- 第二种是尽可能提升集中式事务处理的能力，集中式事务能够达到和分布式同样的效果，就不再需要分布式事务了。

我们认为现在以 NoSQL 为代表的这些系统的推动者，实际上是持这样的观点的。比如典型的 NoSQL 客户系统 MongoDB，它的一般的事务处理或者数据库的访问，都是 Single Document 一个文档一个文档去处理的。实际上，就是如果真要进行复杂的事务处理，那就到上层应用去处理，就用最定制化的方式去应对这种事务，它的效

率可以比较高。

另外一种观点认为分布式事务本身应该是可行的，我们应该提升数据库处理分布式事务的能力，解放开发者。

这样的观点就是那些推动 NewSQL 这一类系统的人所持的观点，比如说谷歌的 Spanner、TiDB、OceanBase。那么想要把分布式事务做好，怎样去优化，提升分布式事务的能力？首先要处理异常，分布式事务最害怕的一个问题就是出现异常，出现异常之后，如果是一个分布式事务，一旦事务锁掉一个节点上的数据，另一个节点出现故障的话，就会很麻烦。那为了处理异常，然后以 Spanner 为代表，使用了很多高可用的系统架构，用 Paxos/Raft 创建这种在云平台，在这种廉价计算机上同样能够有高可用能力的基础设施，在这个上面我可以去规避分布式事务所遇到的这种异常的问题。

但除了异常问题之外，实际上还有分布式事务的扩展性的问题。虽然分布式事务可以比较放心地应用于可靠的、高可用的系统上，但是它的性能会比较差。因此，我们必须要优化它的性能，为此学术界也做了很多尝试，对于团队来讲，最近几年，我们也做过一些研究和尝试。接下来我就大概讲一下我们的大概思路。

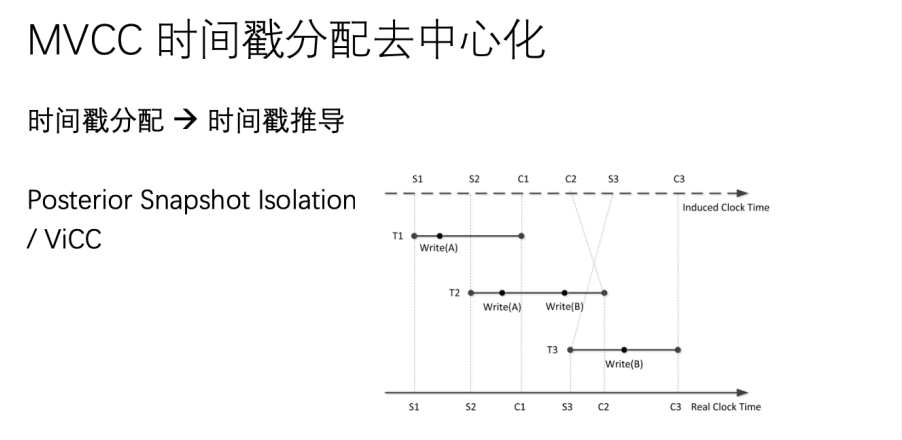
首先，我们要了解到底是什么限制了分布式事务的扩展能力，大家是比较公认的一种观点是制约分布式事务扩展的主要是事物之间的这种阻塞 Blocking。你可以想象一下，当一个事物它去访问一个数据，特别是修改一个数据之后，它会加锁，然后在加锁的过程中，又要去跟其他的节点进行各种通信。

其实这种通信有时候是很耗时的，有时候甚至要跟异地的节点，比如说要做高可用，就需要跟异地的节点建立通信。在加锁的过程中去通信，由于通信的过程很长，然后就会把加锁的时长变得特别的长，阻塞就会变得很严重。一旦事务处理的阻塞时间增加，它的事物的吞吐有可能会受到很严重的影响，特别对于有一些热点的数据出现的话，不是时长增加一倍，性能就降低一倍的，有时候时长增加一倍，性能可能会降低若干倍，这是一个很麻烦的问题。

所以说，真的想要解决分布式的扩展能力的话，在已经有高可用的前提下，我们最重要的目标就是要降低阻塞。怎样降低阻塞？其实可以用到很多的技术。比如说 MVCC/OCC、MVCC 是多版本的数据管理，它可以降低阻塞，这个很直观，就是我一个数据有多个版本，当我的一个事务去改动数据的时候，我直接产生一个新的版本，这样就不用去阻止别人读你的旧版本。

因为你有多个版本，新版本在产生的过程中，你没有办法去读，可以去读旧版本不用阻塞，这就是 MVCC 的使用。然后 OCC 也是一样的，就是 OCC 就是乐观并发控制，也就是说默认不需要加锁，到最后再来检测这个事务是不是执行正确，如果执行不正确推翻就行了。然后还有一些锁的优化，通过各种各样的技术，来把阻塞的现象把它尽量的减少，从而来提升分布式事务的扩展能力。

MVCC 时间戳分配去中心化



在 MVCC 上我们做过一些工作。当然 MVCC 有一个时间戳分配的问题，就是说它来判断一个事务或者一个数据或者一个数据的版本，它是不是应该由某一个事务去读取的话，它要通过一些时间戳的判断来做。时间戳的分配很麻烦，按道理来说，它是应该有一个中心的时钟，大家都去中心的时钟去拿时间戳，这样就可以保证事务处理正确无误。但通常如果有一个中心的时钟的话，那扩展性就受限了，比如谷歌的 Spanner 用一些原子钟去规避这个问题。然后我们做的一个研究就是去中心化，去

优化了 SI 的隔离级别。

SI 是一种典型的 MVCC 的隔离算法或者并发控制的算法，它是需要时间戳的。我们做了一个去中心化。想法是我在给事务分配时间戳的时候，不是在事务开始的时候分配，而是在事务要结束的时候，根据它跟其他事务的关系、冲突情况，来给它指定一个合适的时间戳，所以叫后验时间戳，Posterior SI。这种方式使得我们可以不需要用一个中心的时间戳去做这个事情，但这个东西做起来其实蛮复杂的，我们大概 5 年前做了这件事情，最后有一些实验没有实现在现实的系统里面去，因为实现相对来说比较复杂，而实际应用中使用一个统一的中心的时间戳，基本的应用还是可以满足的。

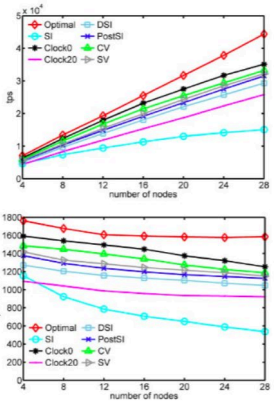


Fig. 7. TPC-C Performance (20% distributed txns)

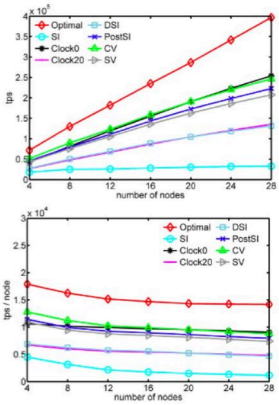
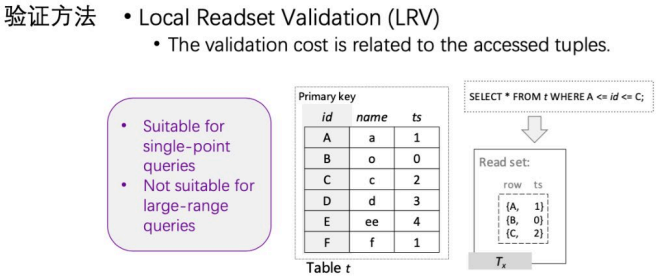
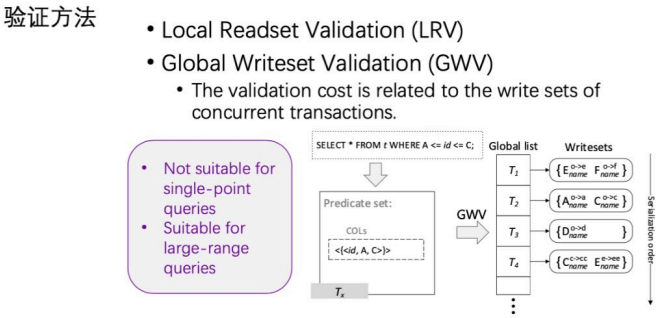


Fig. 9. SmallBank Performance (20% distributed txns)

我们做了很多的实验，实验结果可以表明我们的方法，当扩展到一定程度的时候，这种时间戳的分配是不会成为一个扩展性的瓶颈的。

降低 OCC 的阻塞时间



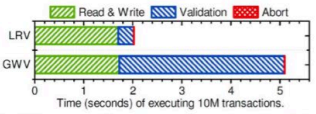


我们也做了一些 OCC 的工作。OCC 在事务访问数据的时候，就放开让事务去访问，访问完了事务要结束的时候会做一个验证叫 Validation，做完验证，再决定这个事务是提交还是回滚。这种方式也是降低事务之间阻塞的一种方法。其实这种事务的最后的正确性验证，有时候会挺耗时间的，所以在这个上面做了一个优化。

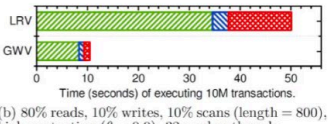
我们认为做正确性验证的方式有两种，一种主要的方式叫 Local Readset Validation，每个事务把它读过的数据记录下来，最后再去查读过的数据有没有被改动过。这种方式的缺点是当读取的东西特别多的时候，它的代价就会相当的大。

然后，还有一种方式叫 Global Writeset Validation，这种方式就是我不去记录每个事务读过的数据，只记录现在有多少正在运行的事务改动了那些数据，也就是说记录的是那些被改动的数据。然后读的时候，观察它的范围有没有包括改的数据，如果包括了验证就失败。这种方式对读取数据内容比较多的事务是友好的，但对那种小的、短的事务并没有那么友好。

- Local Readset Validation (LRV)
- Global Writeset Validation (GWV)



(a) 80% reads, 20% writes, low contention ($\theta = 0.6$), 32 worker threads.

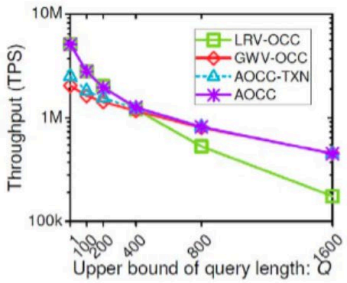


(b) 80% reads, 10% writes, 10% scans (length = 800), high contention ($\theta = 0.9$), 32 worker threads

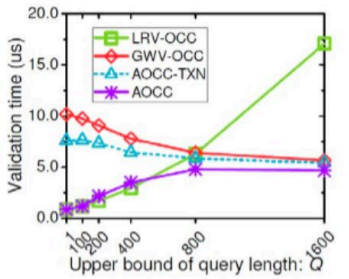


Adaptive Optimistic Concurrency Control (AOCC) that combines LRV and GWV works well in any workload.

所以我们就做了一个叫 AOCC，Adaptive OCC，就是把这两种方式给结合起来，我们会判断一个事务的运行情况，如果读取的数据很多，就用 Writeset Validation；如果读取得很少，就用 Local Readset Validation，这样的话就把两种方式的优点结合起来。

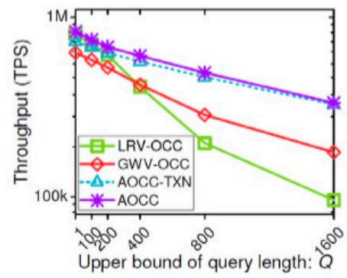


(a) Transaction throughput.

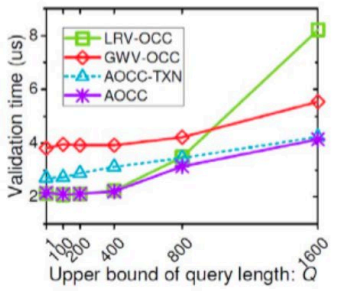


(b) Validation time of each transaction.

YCSB



(a) Transaction throughput.



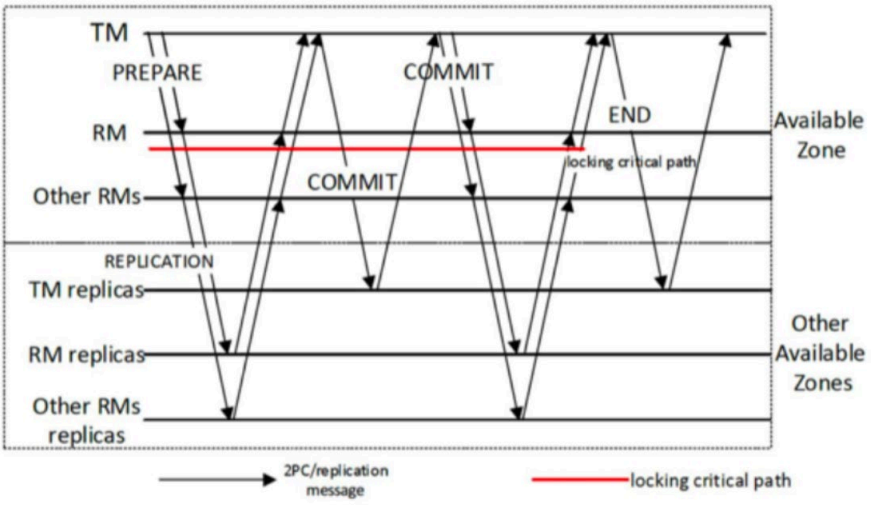
(b) Validation time of each transaction.

TPCC

我们做了一些实验，实际上结果确实证明这种方式没有极端的情况的缺陷。因为以前的那种就是读集 Receipt Validation 和 Receipt，德行在在各自的极端情况下都会呈现出特别差的一个性能。但是我们这种方法实际上它是比较均衡的。

跨区域高可用系统的锁时长

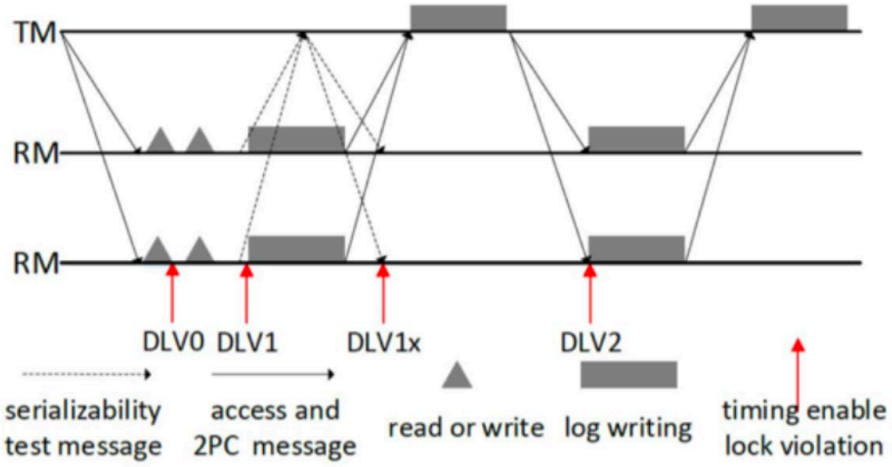
然后第三个工作也是关于分布式事务的，就是我们做了类似于 Spanner 这样的系统，跨区域的高可用的系统。



就像我刚才提到的，一旦加锁，加锁的过程中，出现了跨区域数据的通信，这个持锁的时间就会特别的长。这里是一个例子，在跨区域高可用的系统上做的一个两阶段提交。可以看到红色的线是一个加锁的过程，这个过程已经是在一个正常的事务里面最短的加锁过程了，它是在事务提交开始之前加锁，一直到事务提交完成之后释放锁。

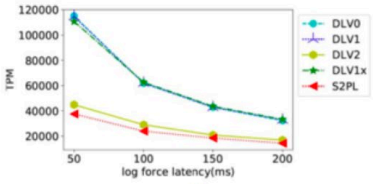
对一个普通的事务来说，它本身就是要加锁的，但这个加锁的时间可以看到，在加锁过程中，Prepare 阶段会有大量的本地节点跟异地节点之间的同步，然后在 Commit 阶段，同样的也有大量的本地节点跟异地节点之间的同步，这样的一个同步是很耗时间的，如果在这个时间上去做加锁的话，一旦遇到热点的数据访问，这个事务处理的性能就会极度的下降。所以在这样的条件下，我们就想可不可以用提前释放锁的方式

去规避加锁，缩短加锁的长度，直接降低阻塞概率。

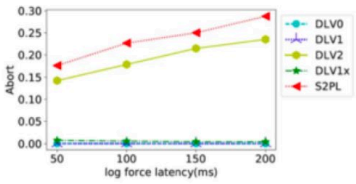


然后我们就设计了叫 DLV，LV 的意思是 Lock Violation，实际上就是提前释放锁。DLV 的话我们叫 Distributed Lock Violation，同样是一个两阶段提交的一个协议。我们就看在什么地方释放所，它的效率是好的。我们选择了四个时间点：

- 第一个是在 prepare 阶段之前访问数据的时候，访问一个数据就放一个数据锁，相当于就不加锁，这是一种最极端的方式。但这种方式到后面的回滚率会非常的高，只能通过验证的方式来判断事务是否正确的执行，因此遇到死锁的情况也会很多，然后事务不正确的情况也很多。
- 第二个时间点就是我们叫 DLV1，就是在事务基本上数据访问的差不多了，但是协调节点还不太清楚，就是说所有的事务节点是不是已经完全做完了不太清楚，但是所有的事务节点各自都认为它自己做完的时候，这个时候释放锁。
- 第三个时间点就是多加了一个协调的过程，协调的节点会跟所有的事务处理的节点做一个通信，通信完了之后，它认为这个时候所有节点都做完了，这个时候释放锁。
- 第四个时间点就是两阶段提交的第一个阶段结束之后，再释放锁，这个是最安全的。

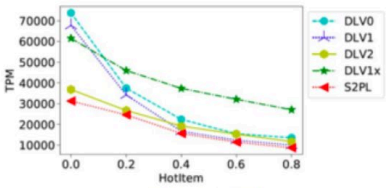


(a)

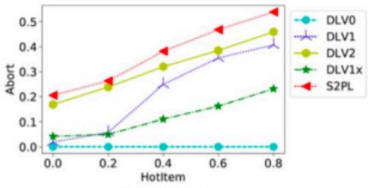


(b)

Fig. 12: Impact of Geo-distance on Performance



(a) NewOrder TPM



(b) NewOrder abort rate

Fig. 14: Impact of Hot Spot on Performance (TPC-C)

然后我们就实验了这些不同的加锁和释放锁的方式，得到的一个结论，左边这两张图（横轴是远程通信的时长）说明两个计算机中心离得越远，它通信时间越长，然后用这种传统的方式，会看到时长越长，它的性能就会越差。在高冲突的情况下，分布式事务处理性能就会比较差。但是如果使用提前释放锁的方式，性能就是绿色蓝色的线，表示着它的性能会有一个比较大的提升，这个就说明提前释放锁是有用的。

但什么时候提前释放锁最合适呢？右边这个图我们做的一个实验最后的结论是第三个时间点就是 DLV1x 这种方式，协调节点跟事务处理阶段有一个短通信，这个是本地通信，不是异地通信，通信之后确认所有节点都做完了，这个时候释放锁，这样的负面作用是最少的，而且它的加锁时间也会很短，这种方式它的效率是最高的。

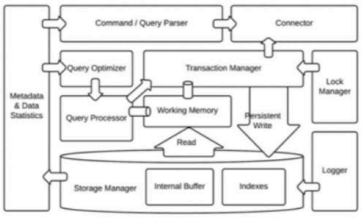
2.2.2 数据库系统解耦合

1. 关系数据库的内核实现极具工程挑战

- 代码耦合度高（Monolithic）。
- 测试困难，特别对于分布式系统。
- 导致开发成本高，功能演进缓慢。

2. 为了实现 One Size Fits a Bunch

A Decomposition of DBMS



我刚才介绍了三项研究，都是关于分布式事务处理的，然后我接下来再讲一讲我们在数据库系统解耦上面的一些研究，这也是非常有趣的一个问题。什么叫数据库系统的解耦？实际上教科书会把数据库系统拆成一个一个的模块，比如说这个是 collector 就是跟应用对接的连接器，还有 Query Optimize、Query Evaluator 或者叫 Query Processor，就是查询处理查询优化的模块，New Storage Manager、New Transaction Manager，还有各种日志、Lock、Manager 等等，这个是我们教科书上面对数据库一个模块的切分。

但实际上当我们去真正的看一个数据库系统的实现，就会发现这些模块之间实际上没有切分的那么干净的，而且有时候是实际上模块之间很高耦合的，很紧的耦合在一起。对于一个刚开始做数据库的人会觉得跟我们学的东西会不完全一样。然后很少的人真的去探究为什么会是这样。我们在实现数据库系统的时候，实际上这种高耦合的系统架构给我们带来了很大的困扰。

我们当时在某银行改那个 OceanBase 的系统时，改动一个数据类型，我记得好像花了好几个月的时间，很多人去做这个事情。这实际上一听上去会让人比较诧异，但实际上你去看系统的实现，它就是这么回事。一个数据类型好多地方都会用到，必须把每一个地方都清除掉，这个时候就必须花很多时间去读代码去理解去测试的。其实我们觉得如果一个系统的耦合度能够变低，模块之间能够分得很清楚，实际上对系统工程来讲是有很大收益的。

我们回顾刚才讲的一个数据库的发展趋势，叫“One size fits a bunch”，也就是说我们认为以后系统会变得很定制化。如果一个系统的模块化做得很好的话，去定制去改动这个系统也会变得很简单。一旦一个新的硬件出现的时候，我们要去使用新的硬件去对这个系统进行优化，会变得更简单。

其实一个数据库系统的实现，是有需要去做进一步的解耦合，这里面其实有很多问题。我们去做了一些探索，但其实是非常有限的探索，我觉得这个工作其实可以有更多的人去做。我们做的探索，就是说想把并发控制直接从数据库的存储层抽离出来，然后让存储的代码跟并发控制的代码尽量的互不相关。

B-tree’ s Search Function in Textbook

```
search(root, key) {
  while !is_leaf(node) {        //寻找叶子结点
    child_node = get_child(node, key)
  return find(node, key)        //在叶子结点中寻找特定键值
}
```

这是一个 B-Tree 的 Search Function 的例子，在教科书里面，关于 B-Tree 的 Search，你可能会看到这样一个代码，非常简单。

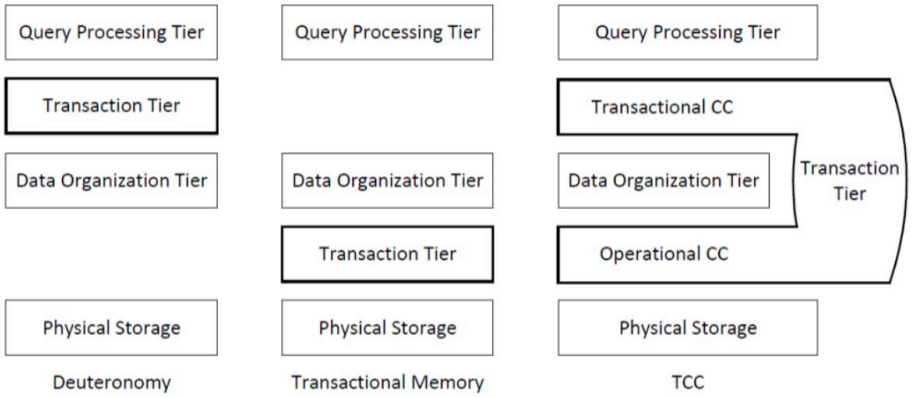
B-tree’ s Search Function in Real World:

```
search(root, key, value) {
  Lock(root, key)  //谓词锁，用来禁止幻象
  node = root
  Latch(node, S)              //施加共享门栓
  while !is_leaf(node) {
    child_node = get_child(node, key)
    Latch(child_node, S)      //对叶子节点施加门闩
    UnLatch(node)             //叶子节点获得门闩后
                              //释放父节点门闩
    if FindSMO(child_node) {//检查结构修改标记，
                              //防止有并发插入操作分裂页面
      Unlatch(child_node)     //如果有并发插入操作分裂页面，
                              //等待分裂结束后重新查找
      GetSMOLatch()           //通过获得结构修改门闩等待分裂结束
      ReleaseSMOLatch()       //一旦获得结构修改门闩
                              //说明分裂已经结束，这时立即释放
      retry
    }
    node = child_node
  }
  return find(node, key)
}
```

但实际上一个 B-Tree 的 Search 没有那么简单，可以看到这里面有好多的东西，这还只是一个例子。如果对开源系统比较熟悉的话，一般的一个开源系统的 B-Tree，差不多要将近十万行代码，非常复杂。

这个代码为什么这么复杂？可以看到 B-Tree 里面有很多的锁，有比如 Latch、Lock 之类的很多东西，它实际上是在做并发控制。当然并发控制只是导致代码复杂的原因之一，但还有其他的原因，并发控制把这个代码变得远远的复杂于 B-tree 本

身功能的程度。其实这就是数据库解耦合的一个动机，如果可以把耦合度解开，并发控制可以交给一个单独的模块去做，B-Tree 的代码就可以像第一个例子那样写，事情就变得很简单。

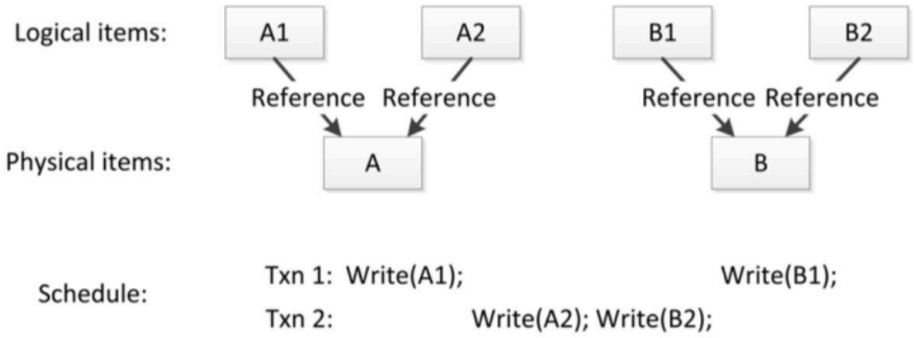


于是，我们就想如果把 CC 就是并发控制从数据库这种体系里面解耦出来应该怎么做？有很多种方式，最左边的这种其实是比较传统的方式，这种方式实际上并没有让数据库存储层变得更简单，只是在存储地上面做了一个事务处理层，这是一个比较浅的做法。中间的这种就是一个很暴力的做法，它是在物理的存储上面加一个 Transaction Tier，然后在上面做存储做运算。

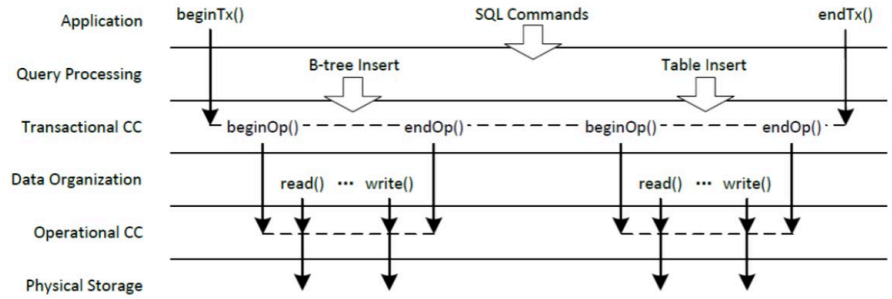
大家应该听说过 Transactional Memory，就是事务内存，这种就是直接用事务内存做事务处理，是一种很暴力的方式。最右边是我们提出来的方式，把并发控制的层次分成了两层，一种是我们叫操作层的并发控制，一种是事务层的并发控制，把它们合在一起变成一个新的模块。实践下来肯定是我们这种方式效果明显地更好，这种很暴力的事务内存的方式，实际上性能是不可以接受的。

我们其实看到现在有一些做存储的同学，他有一些比较天真的想法，他认为把事务做到存储的最底层，然后上面就不需要关心事务了，实际上那是不行的。事务是跟系统的功能是有很多耦合的因子在里面的，不能完全把它抛弃掉。然后最左边这种方式的结果是不彻底的，实现 B-tree 的时候还是会挺复杂的。然后我们提出的这种方式，

可以清楚的把 CC 给抽离出来。



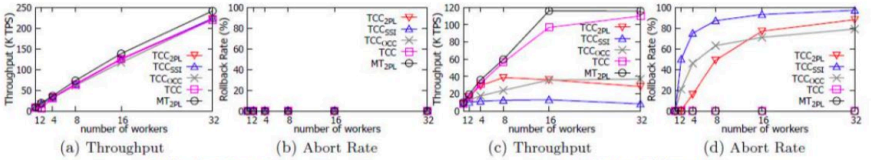
这个事情其实并不是那么的简单，上面是一个很简单的例子，A 和 B 是两个物理数据，然后在数据库的数据结构里面，定义两个引用（Reference），A1 和 A2，B1 和 B2，A1 和 A2 都是指向 A 的，B1 和 B2 都是指向 B 的。上面的事务层实际上是对 A1、A2、B1、B2 进行访问的。这个时候如果只是通过逻辑层去决定事物跟事物是否冲突的话，是会出错的。因为这里的逻辑层跟物理层，有一个重复引用的关系，可以看到下面这个事务处理的 Schedule，从 A1、A2、B1、B2 这种方式去看，好像这两个事务这样处理是没问题的，它是有这种可串行化的能力的，但实际上并没有，因为 A1 和 A2 指向的是同样一个数据。



这种实际上就是说如果真的要去把这个事务抽离出来的时候，会有很多的问题需要去解决，我这里没有办法深入地探讨，总之我们做了这样一个尝试，我们叫 Transpar-

ent Concurrency Control（TCC），就是透明的并发控制的一种模式。

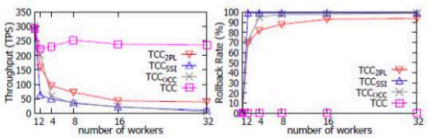
我们把这个事务层抽成两层，一个是事务层次的并发控制，一个是操作层次的并发控制，让这两种东西能够配合起来使用。然后用户去编写程序的时候，他可以不要去关心操作层面的并发控制，他直接去写他的 B-Tree 就行了。但是写好 B-Tree 之后，在事务层次上的这种并发控制，需要提供一些语义的信息说明哪种操作跟哪种操作之间实际上是不会冲突的，这样整个事务处理过程就可以保证正确性和高效性。



(a) Throughput (b) Abort Rate (c) Throughput (d) Abort Rate

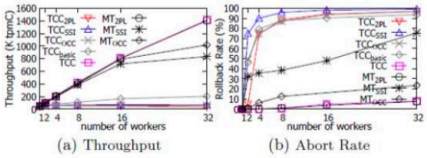
I. Random Insertion. II. Sequential Insertion.

Figure 7: Performance on B-tree Insertion.



(a) Throughput (b) Abort Rate

Figure 8: Performance on a Corner Case.

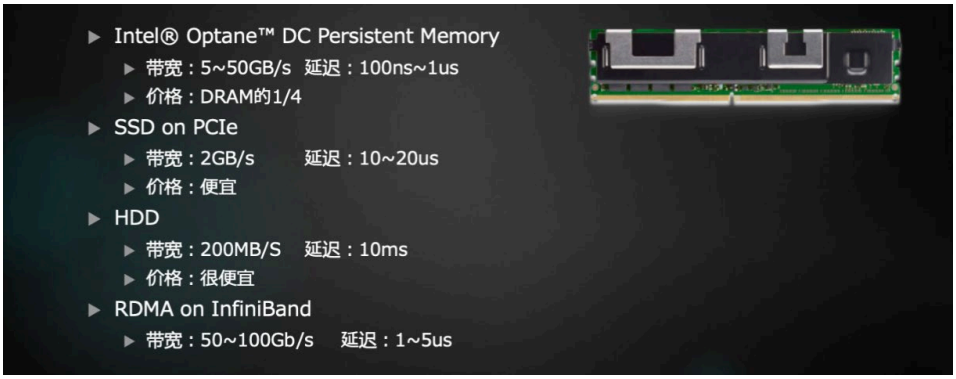


(a) Throughput (b) Abort Rate

Figure 9: Performance on Revised New-Order Transactions.

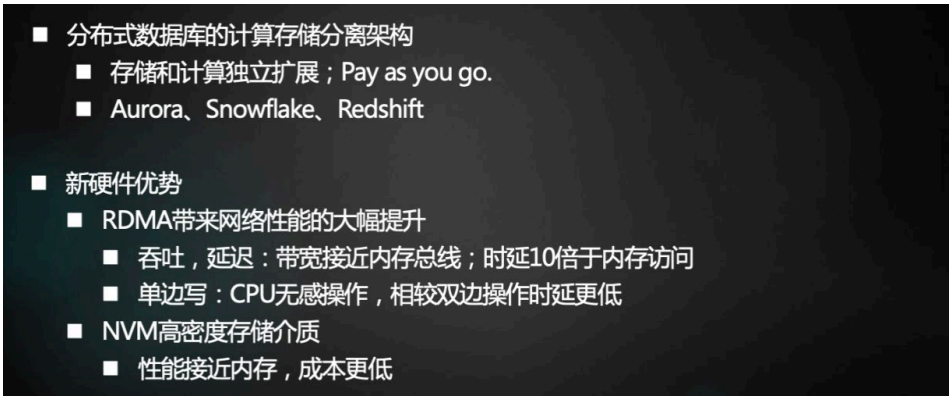
我们做了一些实验，去验证这样的一个解耦。右边的这些图，这种圆圈的线代表原始数据库实现的性能，可以看到我们的方式（TCC）的性能在很多情况下可以接近原始数据库的性能，这是解耦之后数据库的表现。很多时候解耦之后的表现可以接近原始数据库的性能，所以我们觉得这种解耦实际上还是可行的。但如果真的要把它用到一个现实的系统当中，其实并没有那么简单。这是我介绍的第二个研究工作，就是我们在数据库解耦上面的一些有趣的发现。

2.2.3 新硬件

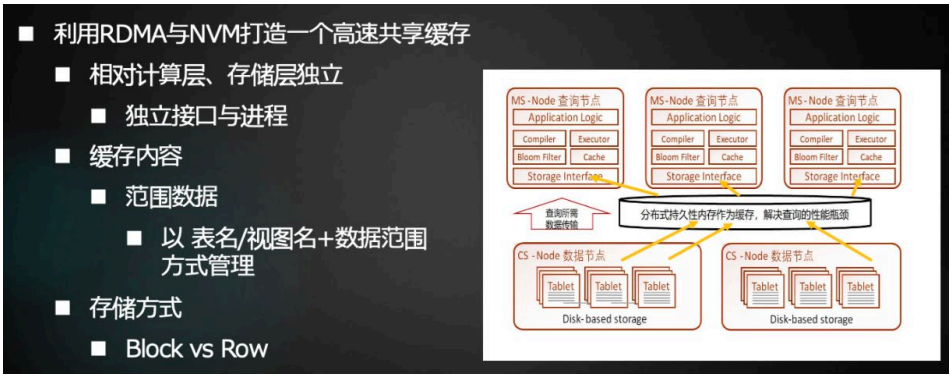


最后我再谈一谈新硬件，就是这种非易失内存。英特尔的傲腾是现在市面上唯一的一个真正的非易失内存产品，图中是产品的相关指标。对于存储器件的话，我们一般看两个指标，一个是带宽，一个是延迟。

可以看到它的带宽和延迟都是远远超过 SSD，当然更加超过这种硬盘。它的价格会比 SSD 和硬盘要昂贵不少，但相对于内存而言，它还是便宜的，我们可以预测它后面会越来越便宜，它跟 SSD 之间的一个价格的差异会变得越来越小，所以以后它有可能会取代 SSD 这种固态硬盘，但是不会太早。这种新的存储，它的性能更好，又比内存的造价低，所以它以后在系统当中肯定是有很重要的一个位置的。现在有了这种硬件之后，我们需要讨论在数据库系统里面，这个硬件到底起什么作用，一个新的数据库的架构应该怎么去使用它？怎么定位它的价值和位置？



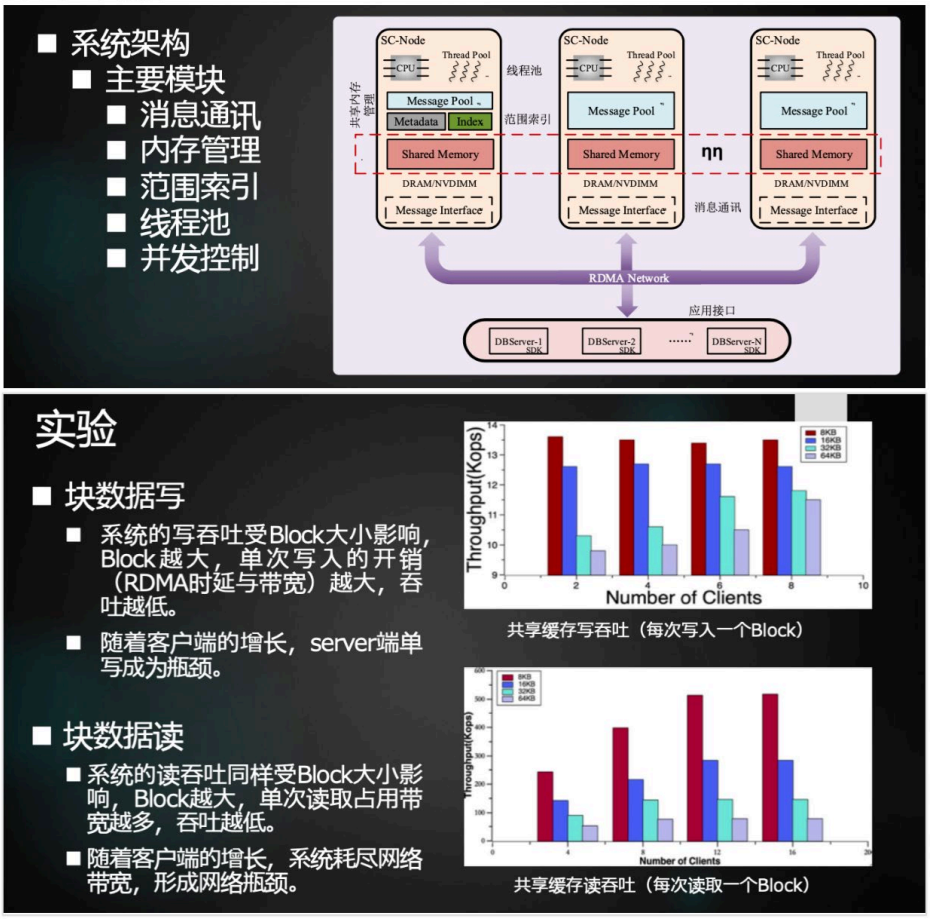
我们团队对这个东西讨论了很长的时间，最后有一个这样的设计，首先非易失内存这个东西，当然要用到数据库里面，数据库有各种形态的数据库，不同的形态的数据库使用方式是不一样的，但我们最后把它定位在云的数据库上面，因为我们知道云是未来的最主要的架构。在云的数据库上面怎么去用这个东西，我们觉得它跟 RDMA 的使用应该结合在一起。



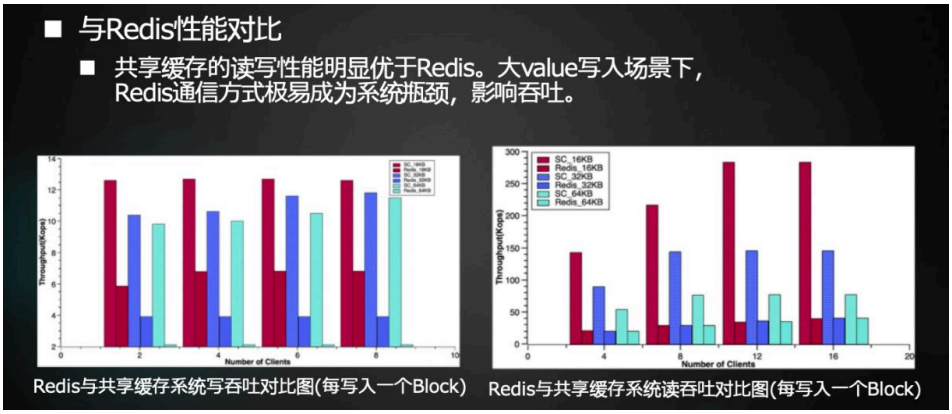
现在的云数据库变成一种计算节点跟存储节点是相对分开的架构方式。然后一旦 NVM 加进来之后，我们希望它成为计算节点跟存储节点之间的一个缓存。我们觉得现在暂时不能用它来做全量数据的存储，因为它的价格实在是比较昂贵，很多冷的数据，完全没有必要存在这样昂贵的存储里面，所以它作为一种缓存，比较合适的。

另外一方面，它作为一个缓存，不应该是一个割裂的节点，因为我们去看它的性能指标，可以看到实际上这种非易失内存的吞吐、延迟，和在高速网络上的 RDMA 的吞吐和延迟是比较接近的。如果比较接近的话，这个器件是通过 RDMA 的远程去访问，还是通过本地访问的速度差异有可能并不会很大。如果这个速度的差异并不会很大的话，我们实际上是可以把多个节点的 NVM 联合在一起，作为共享的缓存，缓存共享有非常多的优势，省去了很多缓存数据同步的代价，然后还可以让系统的负载均衡变得更好。

我们决定去设计这样一个系统架构。这个是 NVM，我们叫存储节点和计算节点之间的一个缓存。



我们这个事情正在做的过程中，所以目前为止我们是实现了一个分布式的缓存，可以作为共享缓存来用。我们开始讨论它应该是作为数据库而言，是行级别的缓存还是块级别的缓存，最后我们的选择是块级别的缓存，主要的原因还是因为实现起来更简单。我们先试一试，如果做到行级别的缓存的话，是有很多的工作量的，我们后期可能还会去尝试。



然后缓存的基本的测试，我们觉得我们的实现基本已经到位了，就是它的带宽的瓶颈基本上压到了 RDMA 访问带宽的瓶颈，如果要对它进行读写的话，它的瓶颈基本上就是 RDMA 远程访问的瓶颈，然后它的性能是远远高于像 Redis 这样的一个系统的，我们希望用这个缓存把它放在数据库里面，去提升这种云数据库的一个性能，但这个过程我们还在实现当中，我们有一些初步的结果，它是有一些效果的，特别是面对底层是 SSD 或者磁盘这样的系统的时候。我们希望后面有更明确结果的时候，再给大家介绍。

03 相关论文列表

Jinwei Guo, et al.: Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. PVLDB 2019

Huan Zhou, et al.: Plover: Parallel Logging for Replication Systems. FCS 2019

Ningnan Zhou, et al.: Transparent Concurrency Control: Decoupling Concurrency Control from DBMS. arXiv 2019

Donghui Wang, et al.: Fast Quorum-Based Log Replication and Replay for Fast Databases. DASFAA (1) 2019: 209-226

Tao Zhu, et al.: Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage. USENIX ATC 2018

Jiahao Wang, et al.: Range Optimistic Concurrency Control for a Composite OLTP and Bulk Processing Workload. ICDE 2018

Jinwei Guo, et al.: Efficient Snapshot Isolation in Paxos-Replicated Database Systems. DASFAA 2018

Xuan Zhou, et al.: Posterior Snapshot Isolation. ICDE 2017

这是我们实验室的一些代表性论文，不是很全，我刚才讲的部分技术并不在列，因为

还没有公开发表。如果感兴趣的话，大家可以阅读一下。

写在后面

华东师范大学周烜教授也是 2020-2021 年度美团科研课题合作学者。当前美团技术团队与超过 30 位来自国内外高校和科研院所的学者建立了科研课题合作。美团科研合作计划，基于美团在生活服务领域全场景里提炼出的科研命题，面向学术界征集前沿解决方案。

我们致力于与学术界“一起解决真实世界的问题”，愿与学术界共同推动产学研成果落地。2021 年将更加精彩纷呈，敬请期待。

后端

所谓「万变不离其宗」。无论前端框架如何在演进、变化，后台总能像「中流砥柱」一样，承载着各种「滔天巨浪」。

2019 年，美团技术博客共发布 18 篇后台的文章，我们分成了基本功、架构、实践与经验的总结等 3 个版块进行展示。《字节码增强技术探索》、《Java 动态追踪技术探究》，带领我们探究了技术底层的实现原理；《美团点评 Kubernetes 集群管理实践》、《美团集群调度系统 HULK 技术演进》，展现了对成千上万台机器如何进行管理的艺术。

「泰山不让土壤，故能成其大；河海不择细流，故能就其深」，因后台涉及的领域比较广、比较杂，任何一个想在这个领域取得成就的工程师，都要具有非凡的恒心与耐力，这些文章只是「后台」体系的冰山一角，我们需要学习的还有很多很多。这是我们的「宿命」，也是我们的「使命」。

这 18 篇文章，献给那些默默抗住亿万并发、管理数万台机器、操纵千百个数据库的后台工程师们，感谢你们的负重前行。

基本功

Java 魔法类：Unsafe 应用解析

璐璐

前言

Unsafe 是位于 sun.misc 包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升 Java 运行效率、增强 Java 语言底层资源操作能力方面起到了很大的作用。但由于 Unsafe 类使 Java 语言拥有了类似 C 语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用 Unsafe 类会使得程序出错的概率变大，使得 Java 这种安全的语言变得不再“安全”，因此对 Unsafe 的使用一定要慎重。

注：本文对 sun.misc.Unsafe 公共 API 功能及相关应用场景进行介绍。

基本介绍

如下 Unsafe 源码所示，Unsafe 类为一单例实现，提供静态方法 getUnsafe 获取 Unsafe 实例，当且仅当调用 getUnsafe 方法的类为引导类加载器所加载时才合法，否则抛出 SecurityException 异常。

```
public final class Unsafe {
  // 单例对象
  private static final Unsafe theUnsafe;

  private Unsafe() {
  }
  @CallerSensitive
  public static Unsafe getUnsafe() {
    Class var0 = Reflection.getCallerClass();
```

```
    // 仅在引导类加载器 `BootstrapClassLoader` 加载时才合法
    if(!VM.isSystemDomainLoader(var0.getClassLoader())) {
      throw new SecurityException("Unsafe");
    } else {
      return theUnsafe;
    }
  }
}
```

那如若想使用这个类，该如何获取其实例？有如下两个可行方案。

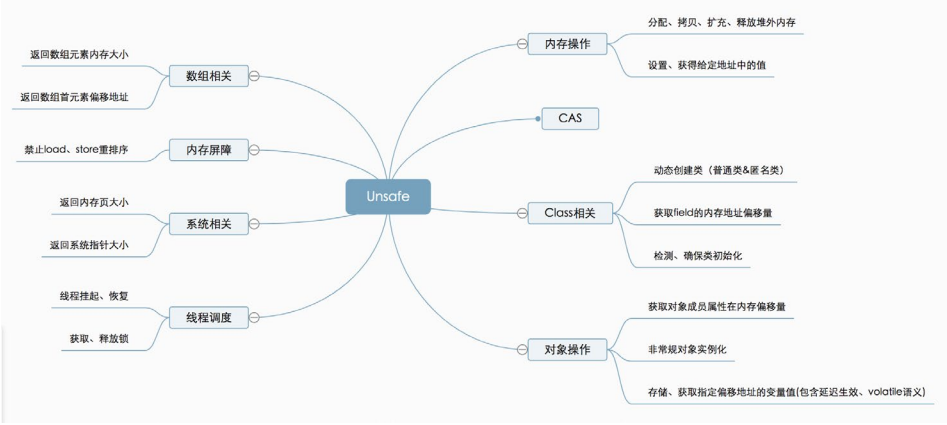
其一，从 getUnsafe 方法的使用限制条件出发，通过 Java 命令行命令 -Xbootclasspath/a 把调用 Unsafe 相关方法的类 A 所在 jar 包路径追加到默认的 bootstrap 路径中，使得 A 被引导类加载器加载，从而通过 Unsafe.getUnsafe 方法安全的获取 Unsafe 实例。

```
java -Xbootclasspath/a: ${path}   // 其中 path 为调用 Unsafe 相关方法的类所在
jar 包路径
```

其二，通过反射获取单例对象 theUnsafe。

```
private static Unsafe reflectGetUnsafe() {
    try {
      Field field = Unsafe.class.getDeclaredField("theUnsafe");
      field.setAccessible(true);
      return (Unsafe) field.get(null);
    } catch (Exception e) {
      log.error(e.getMessage(), e);
      return null;
    }
}
```

功能介绍



如上图所示，Unsafe 提供的 API 大致可分为内存操作、CAS、Class 相关、对象操作、线程调度、系统信息获取、内存屏障、数组操作等几类，下面将对其相关方法和应用场景进行详细介绍。

内存操作

这部分主要包含堆外内存的分配、拷贝、释放、给定地址值操作等方法。

```
// 分配内存，相当于 C++ 的 malloc 函数
public native long allocateMemory(long bytes);
// 扩充内存
public native long reallocateMemory(long address, long bytes);
// 释放内存
public native void freeMemory(long address);
// 在给定的内存块中设置值
public native void setMemory(Object o, long offset, long bytes, byte value);
// 内存拷贝
public native void copyMemory(Object srcBase, long srcOffset, Object
destBase, long
destOffset, long bytes);
// 获取给定地址值，忽略修饰限定符的访问限制。与此类似操作还有：getInt，getDouble，
getLong，getChar 等
public native Object getObject(Object o, long offset);
// 为给定地址设置值，忽略修饰限定符的访问限制，与此类似操作还有：
putInt,putDouble，
putLong，putChar 等
public native void putObject(Object o, long offset, Object x);
```

```
// 获取给定地址的 byte 类型的值（当且仅当该内存地址为 allocateMemory 分配时，此方法结
果为确定的）
public native byte getByte(long address);
// 为给定地址设置 byte 类型的值（当且仅当该内存地址为 allocateMemory 分配时，此方法结
果才是确定的）
public native void putByte(long address, byte x);
```

通常，我们在 Java 中创建的对象都处于堆内内存（heap）中，堆内内存是由 JVM 所管控的 Java 进程内存，并且它们遵循 JVM 的内存管理机制，JVM 会采用垃圾回收机制统一管理堆内存。与之相对的是堆外内存，存在于 JVM 管控之外的内存区域，Java 中对堆外内存的操作，依赖于 Unsafe 提供的操作堆外内存的 native 方法。

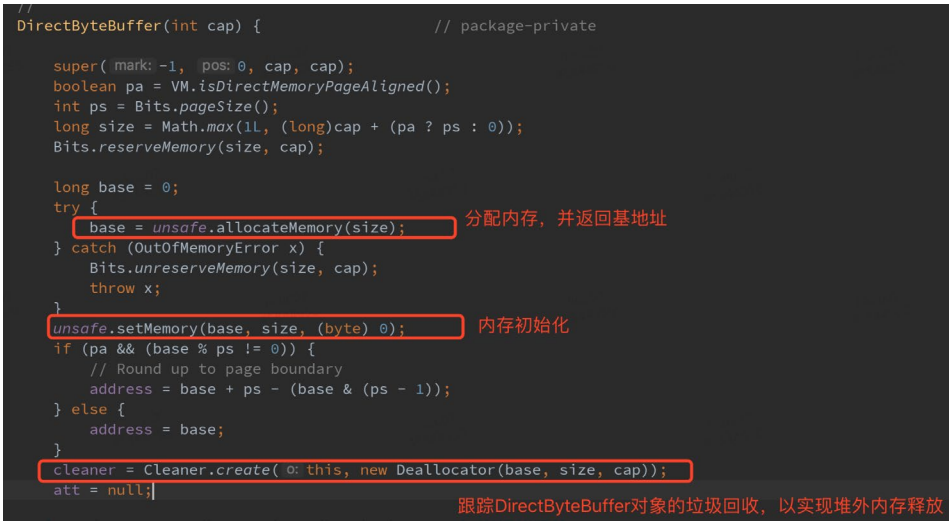
使用堆外内存的原因

- 对垃圾回收停顿的改善。由于堆外内存是直接受操作系统管理而不是 JVM，所以当我们使用堆外内存时，即可保持较小的堆内内存规模。从而在 GC 时减少回收停顿对于应用的影响。
- 提升程序 I/O 操作的性能。通常在 I/O 通信过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内存间数据拷贝且生命周期较短的暂存数据，都建议存储到堆外内存。

典型应用

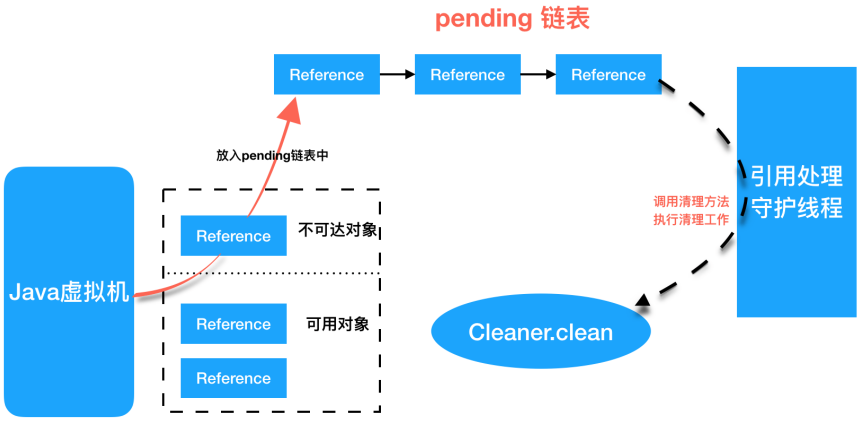
DirectByteBuffer 是 Java 用于实现堆外内存的一个重要类，通常用在通信过程中做缓冲池，如在 Netty、MINA 等 NIO 框架中应用广泛。DirectByteBuffer 对于堆外内存的创建、使用、销毁等逻辑均由 Unsafe 提供的堆外内存 API 来实现。

下图为 DirectByteBuffer 构造函数，创建 DirectByteBuffer 的时候，通过 Unsafe.allocateMemory 分配内存、Unsafe.setMemory 进行内存初始化，而后构建 Cleaner 对象用于跟踪 DirectByteBuffer 对象的垃圾回收，以实现当 DirectByteBuffer 被垃圾回收时，分配的堆外内存一起被释放。

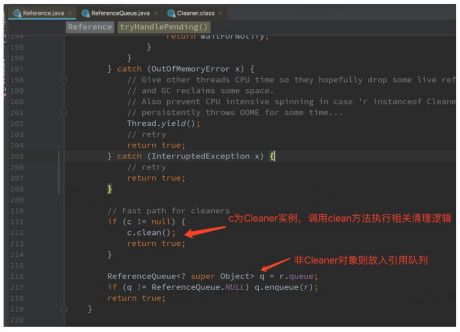


那么如何通过构建垃圾回收追踪对象 Cleaner 实现堆外内存释放呢?

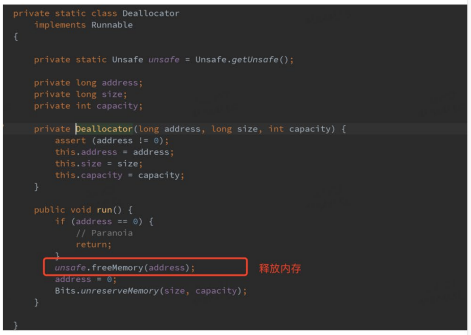
Cleaner 继承自 Java 四大引用类型之一的虚引用 PhantomReference（众所周知，无法通过虚引用获取与之关联的对象实例，且当对象仅被虚引用引用时，在任何发生 GC 的时候，其均可被回收），通常 PhantomReference 与引用队列 ReferenceQueue 结合使用，可以实现虚引用关联对象被垃圾回收时能够进行系统通知、资源清理等功能。如下图所示，当某个被 Cleaner 引用的对象将被回收时，JVM 垃圾收集器会将此对象的引用放入到对象引用中的 pending 链表中，等待 Reference-Handler 进行相关处理。其中，Reference-Handler 为一个拥有最高优先级的守护线程，会循环不断的处理 pending 链表中的对象引用，执行 Cleaner 的 clean 方法进行相关清理工作。



所以当 DirectByteBuffer 仅被 Cleaner 引用（即为虚引用）时，其可以在任意 GC 时段被回收。当 DirectByteBuffer 实例对象被回收时，在 Reference-Handler 线程操作中，会调用 Cleaner 的 clean 方法根据创建 Cleaner 时传入的 Deallocator 来进行堆外内存的释放。



图一



图二

CAS 相关

如下源代码释义所示，这部分主要为 CAS 相关操作的方法。

```
/**
     *  CAS
 * @param o          包含要修改 field 的对象
```

```
    * @param offset     对象中某 field 的偏移量
    * @param expected   期望值
    * @param update     更新值
    * @return           true | false
    */
public final native boolean compareAndSwapObject(Object o, long
offset,  Object expected,
Object update);

public final native boolean compareAndSwapInt(Object o, long offset,
int expected,int
update);

public final native boolean compareAndSwapLong(Object o, long offset,
long expected,
long update);
```

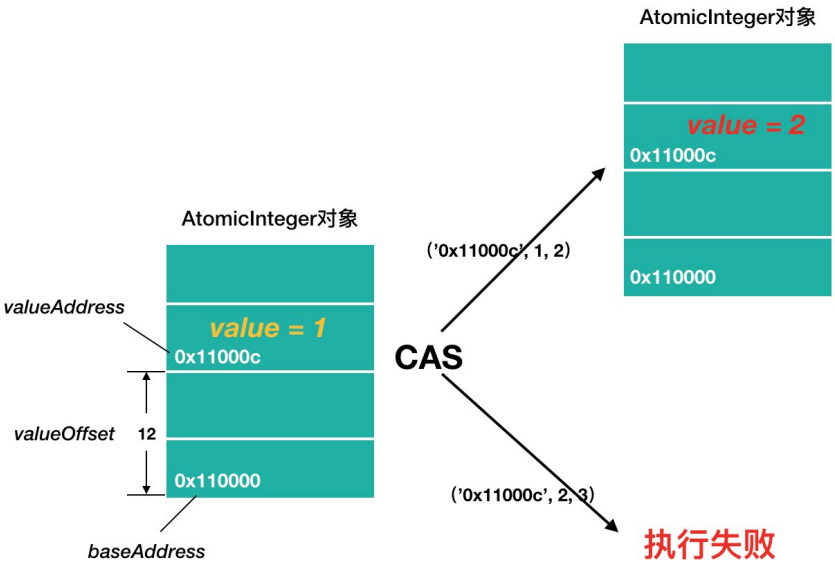
什么是 CAS？即比较并替换，实现并发算法时常用到的一种技术。CAS 操作包含三个操作数——内存位置、预期原值及新值。执行 CAS 操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作。我们都知道，CAS 是一条 CPU 的原子指令（cmpxchg 指令），不会造成所谓的数据不一致问题，Unsafe 提供的 CAS 方法（如 compareAndSwapXXX）底层实现即为 CPU 指令 cmpxchg。

典型应用

CAS 在 java.util.concurrent.atomic 相关类、Java AQS、CurrentHashMap 等实现上有非常广泛的应用。如下图所示，AtomicInteger 的实现中，静态字段 valueOffset 即为字段 value 的内存偏移地址，valueOffset 的值在 AtomicInteger 初始化时，在静态代码块中通过 Unsafe 的 objectFieldOffset 方法获取。在 AtomicInteger 中提供的线程安全方法中，通过字段 valueOffset 的值可以定位到 AtomicInteger 对象中 value 的内存地址，从而可以根据 CAS 实现对 value 字段的原子操作。

```java
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

下图为某个 AtomicInteger 对象自增操作前后的内存示意图，对象的基地址 baseAddress=“0x110000”，通过 baseAddress+valueOffset 得到 value 的内存地址 valueAddress=“0x11000c”；然后通过 CAS 进行原子性的更新操作，成功则返回，否则继续重试，直到更新成功为止。



线程调度

这部分，包括线程挂起、恢复、锁机制等方法。

```
// 取消阻塞线程
public native void unpark(Object thread);
// 阻塞线程
public native void park(boolean isAbsolute, long time);
// 获得对象锁（可重入锁）
@Deprecated
public native void monitorEnter(Object o);
// 释放对象锁
@Deprecated
public native void monitorExit(Object o);
// 尝试获取对象锁
@Deprecated
public native boolean tryMonitorEnter(Object o);
```

如上源码说明中，方法 park、unpark 即可实现线程的挂起与恢复，将一个线程进行挂起是通过 park 方法实现的，调用 park 方法后，线程将一直阻塞直到超时或者中断等条件出现；unpark 可以终止一个挂起的线程，使其恢复正常。

典型应用

Java 锁和同步器框架的核心类 AbstractQueuedSynchronizer，就是通过调用 LockSupport.park() 和 LockSupport.unpark() 实现线程的阻塞和唤醒的，而 LockSupport 的 park、unpark 方法实际是调用 Unsafe 的 park、unpark 方式来实现。

Class 相关

此部分主要提供 Class 和它的静态字段的操作相关方法，包含静态字段内存定位、定义类、定义匿名类、检验 & 确保初始化等。

```
// 获取给定静态字段的内存地址偏移量，这个值对于给定的字段是唯一且固定不变的
public native long staticFieldOffset(Field f);
// 获取一个静态类中给定字段的对象指针
public native Object staticFieldBase(Field f);
// 判断是否需要初始化一个类，通常在获取一个类的静态属性的时候（因为一个类如果没初始化，
它的静态属性也不会初始化）使用。 当且仅当 ensureClassInitialized 方法不生效时返回 false。
public native boolean shouldBeInitialized(Class<?> c);
// 检测给定的类是否已经初始化。通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。
```

```
public native void ensureClassInitialized(Class<?> c);
// 定义一个类，此方法会跳过 JVM 的所有安全检查，默认情况下，ClassLoader（类加载器）
和 ProtectionDomain（保护域）实例来源于调用者
public native Class<?> defineClass(String name, byte[] b, int off, int
len, ClassLoader loader,
ProtectionDomain protectionDomain);
// 定义一个匿名类
public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[]
data, Object[]
cpPatches);
```

典型应用

从 Java 8 开始，JDK 使用 invokedynamic 及 VM Anonymous Class 结合来实现 Java 语言层面上的 Lambda 表达式。

- **invokedynamic**：invokedynamic 是 Java 7 为了实现在 JVM 上运行动态语言而引入的一条新的虚拟机指令，它可以实现在运行期动态解析出调用点限定符所引用的方法，然后再执行该方法，invokedynamic 指令的分派逻辑是由用户设定的引导方法决定。
- **VM Anonymous Class**：可以看做是一种模板机制，针对于程序动态生成很多结构相同、仅若干常量不同的类时，可以先创建包含常量占位符的模板类，而后通过 Unsafe.defineAnonymousClass 方法定义具体类时填充模板的占位符生成具体的匿名类。生成的匿名类不显式挂在任何 ClassLoader 下面，只要当该类没有存在的实例对象、且没有强引用来引用该类的 Class 对象时，该类就会被 GC 回收。故而 VM Anonymous Class 相比于 Java 语言层面的匿名内部类无需通过 ClassClassLoader 进行类加载且更易回收。

在 Lambda 表达式实现中，通过 invokedynamic 指令调用引导方法生成调用点，在此过程中，会通过 ASM 动态生成字节码，而后利用 Unsafe 的 defineAnonymousClass 方法定义实现相应的函数式接口的匿名类，然后再实例化此匿名类，并返回与此匿名类中函数式方法的方法句柄关联的调用点；而后可以通过此调用点实现调用相应 Lambda 表达式定义逻辑的功能。下面以如下图所示的 Test

类来举例说明。

```
import java.util.function.Consumer;


public class Test {

    public static void main(String[] args) throws Exception {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept( t: "lambda");
    }
}
```

Test 类编译后的 class 文件反编译后的结果如下图一所示（删除了对本文说明无意义的部分），我们可以从中看到 main 方法的指令实现、invokedynamic 指令调用的引导方法 BootstrapMethods、及静态方法 lambda$main$0（实现了 Lambda 表达式中字符串打印逻辑）等。在引导方法执行过程中，会通过 Unsafe.defineAnonymousClass 生成如下图二所示的实现 Consumer 接口的匿名类。其中，accept 方法通过调用 Test 类中的静态方法 lambda$main$0 来实现 Lambda 表达式中定义的逻辑。而后执行语句 consumer.accept（"lambda"）其实就是调用下图二所示的匿名类的 accept 方法。

```
Compiled from "Test.java"
public class com.sankuai.meituan.trippackage.api.Test {

  public static void main(java.lang.String[]) throws java.lang.Exception;
    Code:
       0: invokedynamic #2,  0      // InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
       5: astore_1
       6: aload_1
       7: ldc           #3          // String lambda
       9: invokeinterface #4,  2    // InterfaceMethod java/util/function/Consumer.accept:(Ljava/lang/Object;)V
      14: return

  private static void lambda$main$0(java.lang.String);
    Code:
       0: getstatic     #5          // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: invokevirtual #6          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       7: return

SourceFile: "Test.java"
InnerClasses:
     public static final #75= #74 of #78; //Lookup=class java/lang/invoke/MethodHandles$Lookup
                of class java/lang/invoke/MethodHandles
BootstrapMethods:
  0: #38 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
         Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;
         Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #39 (Ljava/lang/Object;)V
      #40 invokestatic com/sankuai/meituan/trippackage/api/Test.lambda$main$0:(Ljava/lang/String;)V
      #41 (Ljava/lang/String;)V
}
```

引导方法

图一

```
import java.lang.invoke.LambdaForm.Hidden;
import java.util.function.Consumer;

// $FF: synthetic class
final class Test$$Lambda$1 implements Consumer {
    private Test$$Lambda$1() {
    }

    @Hidden
    public void accept(Object var1) {
        Test.lambda$main$0((String)var1);
    }
}
```

图二

对象操作

此部分主要包含对象成员属性相关操作及非常规的对象实例化方式等相关方法。

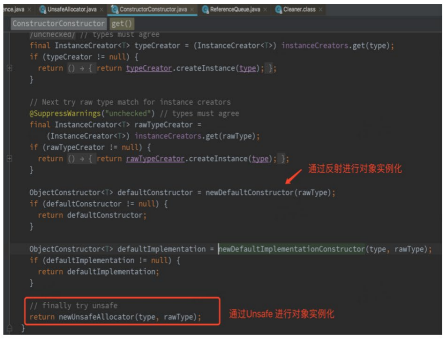
```
// 返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
// 获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，
getChar 等
public native Object getObject(Object o, long offset);
// 给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，
putChar 等
public native void putObject(Object o, long offset, Object x);
// 从对象的指定偏移量处获取变量的引用，使用 volatile 的加载语义
public native Object getObjectVolatile(Object o, long offset);
// 存储变量的引用到对象的指定的偏移量处，使用 volatile 的存储语义
public native void putObjectVolatile(Object o, long offset, Object x);
// 有序、延迟版本的 putObjectVolatile 方法，不保证值的改变被其他线程立即看到。只
有在 field 被 volatile 修饰符修饰时有效
public native void putOrderedObject(Object o, long offset, Object x);
// 绕过构造方法、初始化代码来创建对象
public native Object allocateInstance(Class<?> cls) throws
InstantiationException;
```

典型应用

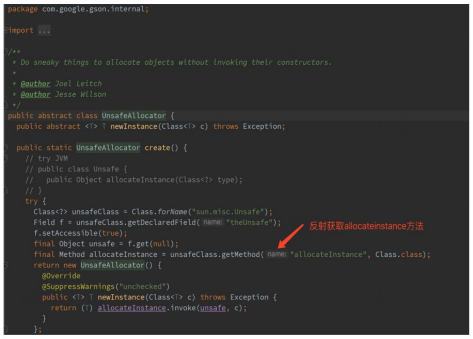
- **常规对象实例化方式**：我们通常所用到的创建对象的方式，从本质上来讲，都是通过 new 机制来实现对象的创建。但是，new 机制有个特点就是当类只提供有参的构造函数且无显示声明无参构造函数时，则必须使用有参构造函数进行对象构造，而使用有参构造函数时，必须传递相应个数的参数才能完成对象实例化。
- **非常规的实例化方式**：而 Unsafe 中提供 allocateInstance 方法，仅通过 Class 对象就可以创建此类的实例对象，而且不需要调用其构造函数、初始化代码、JVM 安全检查等。它抑制修饰符检测，也就是即使构造器是 private 修饰的也能通过此方法实例化，只需提类对象即可创建相应的对象。由于这种特性，allocateInstance 在 java.lang.invoke、Objenesis（提供绕过类构造器的对象生成方式）、Gson（反序列化时用到）中都有相应的应用。

如下图所示，在 Gson 反序列化时，如果类有默认构造函数，则通过反射调

用默认构造函数创建实例，否则通过 UnsafeAllocator 来实现对象实例的构造，UnsafeAllocator 通过调用 Unsafe 的 allocateInstance 实现对象的实例化，保证在目标类无默认构造函数时，反序列化不够影响。



图一



图二

数组相关

这部分主要介绍与数据操作相关的 arrayBaseOffset 与 arrayIndexScale 这两个方法，两者配合起来使用，即可定位数组中每个元素在内存中的位置。

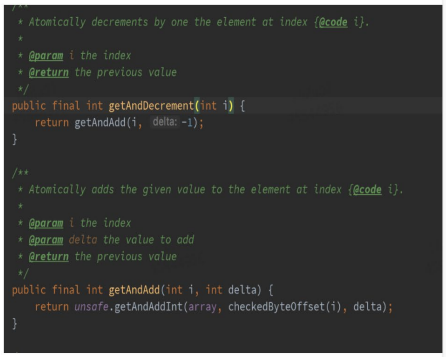
```
// 返回数组中第一个元素的偏移地址
public native int arrayBaseOffset(Class<?> arrayClass);
// 返回数组中一个元素占用的大小
public native int arrayIndexScale(Class<?> arrayClass);
```

典型应用

这两个与数据操作相关的方法，在 java.util.concurrent.atomic 包下的 AtomicIntegerArray（可以实现对 Integer 数组中每个元素的原子性操作）中有典型的应用，如下图 AtomicIntegerArray 源码所示，通过 Unsafe 的 arrayBaseOffset、arrayIndexScale 分别获取数组首元素的偏移地址 base 及单个元素大小因子 scale。后续相关原子性操作，均依赖于这两个值进行数组中元素的定位，如下图二所示的 getAndAdd 方法即通过 checkedByteOffset 方法获取某数组元素的偏移地址，而后通过 CAS 实现原子性操作。



图一



图二

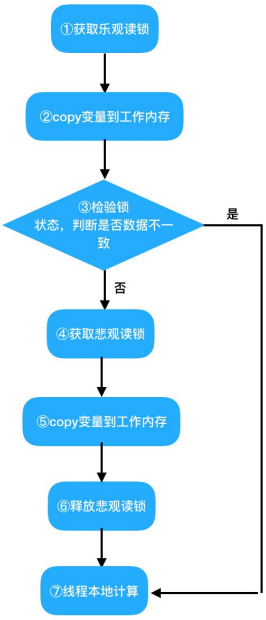
内存屏障

在 Java 8 中引入，用于定义内存屏障（也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是 CPU 或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作），避免代码重排序。

```
// 内存屏障，禁止 load 操作重排序。屏障前的 load 操作不能被重排序到屏障后，屏障后的 load 操作不能被重排序到屏障前
public native void loadFence();
// 内存屏障，禁止 store 操作重排序。屏障前的 store 操作不能被重排序到屏障后，屏障后的 store 操作不能被重排序到屏障前
public native void storeFence();
// 内存屏障，禁止 load、store 操作重排序
public native void fullFence();
```

典型应用

在 Java 8 中引入了一种锁的新机制——StampedLock，它可以看成是读写锁的一个改进版本。StampedLock 提供了一种乐观读锁的实现，这种乐观读锁类似于无锁的操作，完全不会阻塞写线程获取写锁，从而缓解读多写少时写线程“饥饿”现象。由于 StampedLock 提供的乐观读锁不阻塞写线程获取读锁，当线程共享变量从主内存 load 到线程工作内存时，会存在数据不一致问题，所以当使用 StampedLock 的乐观读锁时，需要遵从如下图用例中使用的模式来确保数据的一致性。




```
class Point {

    private double x, y;

    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();    // 使用写锁-独占操作
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
    double distanceFromOrigin() {
        long stamp = sl.tryOptimisticRead();         // ①
        double currentX = x, currentY = y;           // ②
        if (!sl.validate(stamp)) {                   // ③
            stamp = sl.readLock();                   // ④
            try {
                currentX = x;                        // ⑤
                currentY = y;
            } finally {
                sl.unlockRead(stamp);                // ⑥
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY); // ⑦
    }
}
```

如上图用例所示计算坐标点 Point 对象，包含点移动方法 move 及计算此点到原点的距离的方法 distanceFromOrigin。在方法 distanceFromOrigin 中，首先，通过 tryOptimisticRead 方法获取乐观读标记；然后从主内存中加载点的坐标值 (x,y)；而后通过 StampedLock 的 validate 方法校验锁状态，判断坐标点 (x,y) 从主内存加载到线程工作内存过程中，主内存的值是否已被其他线程通过 move 方法修改，如果 validate 返回值为 true，证明 (x, y) 的值未被修改，可参与后续计算；否则，需加悲观读锁，再次从主内存加载 (x,y) 的最新值，然后再进行距离计算。其中，校验锁状态这步操作至关重要，需要判断锁状态是否发生改变，从而判断之前 copy 到线程工作内存中的值是否与主内存的值存在不一致。

下图为 StampedLock.validate 方法的源码实现，通过锁标记与相关常量进行位运算、比较来校验锁状态，在校验逻辑之前，会通过 Unsafe 的 loadFence 方法加入一个 load 内存屏障，目的是避免上图用例中步骤②和 StampedLock.validate 中锁状态校验运算发生重排序导致锁状态校验不准确的问题。

```
/**
 * Returns true if the lock has not been exclusively acquired
 * since issuance of the given stamp. Always returns false if the
 * stamp is zero. Always returns true if the stamp represents a
 * currently held lock. Invoking this method with a value not
 * obtained from {@link #tryOptimisticRead} or a locking method
 * for this lock has no defined effect or result.
 *
 * @param stamp a stamp
 * @return {@code true} if the lock has not been exclusively acquired
 * since issuance of the given stamp; else false
 */
public boolean validate(long stamp) {
    U.loadFence();                    load 内存屏障
    return (stamp & SBITS) == (state & SBITS);
}

/**
 * If the lock state matches the given stamp, releases the
```

系统相关

这部分包含两个获取系统相关信息的方法。

```
// 返回系统指针的大小。返回值为 4（32 位系统）或 8（64 位系统）。
public native int addressSize();
// 内存页的大小，此值为 2 的幂次方。
public native int pageSize();
```

典型应用

如下图所示的代码片段，为 java.nio 下的工具类 Bits 中计算待申请内存所需内存页数量的静态方法，其依赖于 Unsafe 中 pageSize 方法获取系统内存页大小实现后续计算逻辑。

```
    private static int pageSize = -1;

    static int pageSize() {
        if (pageSize == -1)
            pageSize = unsafe().pageSize();
        return pageSize;
    }

    static int pageCount(long size) {
        return (int)(size + (long)pageSize() - 1L) / pageSize();
    }
```

结语

本文对 Java 中的 sun.misc.Unsafe 的用法及应用场景进行了基本介绍，我们可以看到 Unsafe 提供了很多便捷、有趣的 API 方法。即便如此，由于 Unsafe 中包含大量自主操作内存的方法，如若使用不当，会对程序带来许多不可控的灾难。因此对它的使用我们需要慎之又慎。

参考资料

- OpenJDK Unsafe source
- Java Magic. Part 4: sun.misc.Unsafe
- JVM crashes at libjvm.so
- Java 中神奇的双刃剑 - Unsafe
- JVM 源码分析之堆外内存完全解读
- 堆外内存 之 DirectByteBuffer 详解
- 《深入理解 Java 虚拟机（第 2 版）》

作者简介

璐璐，美团点评 Java 开发工程师。2017 年加入美团点评，负责美团点评境内度假的后端开发。

Java 动态追踪技术探究

高扬

引子

在遥远的希艾斯星球爪哇国塞沃城中，两名年轻的程序员正在为一件事情苦恼，程序出问题了，一时看不出问题出在哪里，于是有了以下对话：

“Debug 一下吧。”

“线上机器，没开 Debug 端口。”

“看日志，看看请求值和返回值分别是什么?”

“那段代码没打印日志。”

“改代码，加日志，重新发布一次。”

“怀疑是线程池的问题，重启会破坏现场。”

长达几十秒的沉默之后：“据说，排查问题的最高境界，就是只通过 Review 代码来发现问题。”

比几十秒长几十倍的沉默之后：“我轮询了那段代码一十七遍之后，终于得出一个结论。”

“结论是?”

“我还没到达只通过 Review 代码就能发现问题的至高境界。”

从 JSP 说起

对于大多数 Java 程序员来说，早期的时候，都会接触到一个叫做 JSP（Java Server Pages）的技术。虽然这种技术，在前后端代码分离、前后端逻辑分离、前后端组织架构分离的今天来看，已经过时了，但是其中还是有一些有意思的东西，值得拿出来说一说。

当时刚刚处于 Java 入门时期的我们，大多数精力似乎都放在了 JSP 的页面展示效果上了：

“这个表格显示的行数不对”

“原来是 for 循环写的有问题，改一下，刷新页面再试一遍”

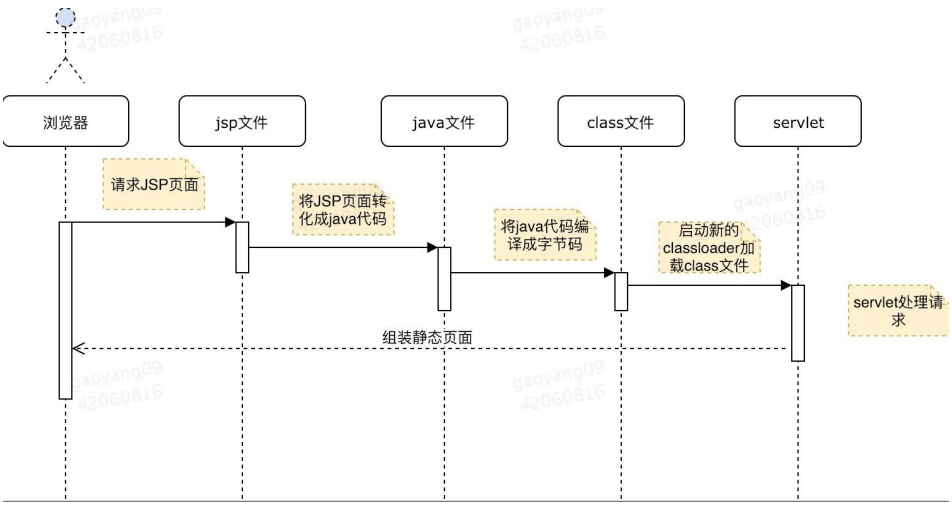
“嗯，好了，表格显示没问题了，但是，登录人的姓名没取到啊，是不是 Sesstion 获取有问题?”

“有可能，我再改一下，一会儿再刷新试试”

……

在一遍一遍修改代码刷新浏览器页面重试的时候，我们自己也许并没有注意到一件很酷的事情：我们修改完代码，居然只是简单地刷新一遍浏览器页面，修改就生效了，整个过程并没有重启 JVM。按照我们的常识，Java 程序一般都是在启动时加载类文件，如果都像 JSP 这样修改完代码，不用重启就生效的话，那文章开头的问题就可以解决了啊：Java 文件中加一段日志打印的代码，不重启就生效，既不破坏现场，又可以定位问题。忍不住试一试：修改、编译、替换 class 文件。额，不行，新改的代码并没有生效。那为什么偏偏 JSP 可以呢? 让我们先来看看 JSP 的运行原理。

当我们打开浏览器，请求访问一个 JSP 文件的时候，整个过程是这样的：



JSP 文件处理过程

JSP 文件修改过后，之所以能及时生效，是因为 Web 容器（Tomcat）会检查请求的 JSP 文件是否被更改过。如果发生过更改，那么就将 JSP 文件重新解析翻译成

一个新的 Sevlet 类，并加载到 JVM 中。之后的请求，都会由这个新的 Servet 来处理。这里有个问题，根据 Java 的类加载机制，在同一个 ClassLoader 中，类是不允许重复的。为了绕开这个限制，Web 容器每次都会创建一个新的 ClassLoader 实例，来加载新编译的 Servlet 类。之后的请求都会由这个新的 Servlet 来处理，这样就实现了新旧 JSP 的切换。

HTTP 服务是无状态的，所以 JSP 的场景基本上都是一次性消费，这种通过创建新的 ClassLoader 来“替换”class 的做法行得通，但是对于其他应用，比如 Spring 框架，即便这样做了，对象多数是单例，对于内存中已经创建好的对象，我们无法通过这种创建新的 ClassLoader 实例的方法来修改对象行为。

我就是想不重启应用加个日志打印，就这么难吗?

Java 对象行为

既然 JSP 的办法行不通，那我们来看看还有没有其他的办法。仔细想想，我们会发现，文章开头的问题本质上是动态改变内存中已存在对象的行为的问题。所以，我们得先弄清楚 JVM 中和对象行为有关的地方在哪里，有没有更改的可能性。

我们都知道，对象使用两种东西来描述事物：行为和属性。举个例子：

```
public class Person{
  private int age;
  private String name;
  public void speak(String str) {
    System.out.println(str);
 }
 public Person(int age, String name) {
    this.age = age;
    this.name = name;
 }
}
```

上面 Person 类中 age 和 name 是属性，speak 是行为。对象是类的事例，每个对象的属性都属于对象本身，但是每个对象的行为却是公共的。举个例子，比如我们现在基于 Person 类创建了两个对象，personA 和 personB：

```
Person personA = new Person(43, "lixunhuan");

personA.speak(" 我是李寻欢 ");

Person personB = new Person(23, "afei");

personB.speak(" 我是阿飞 ");
```

personA 和 personB 有各自的姓名和年龄，但是有共同的行为：speak。想象一下，如果我们是 Java 语言的设计者，我们会怎么存储对象的行为和属性呢?

“很简单，属性跟着对象走，每个对象都存一份。行为是公共的东西，抽离出来，单独放到一个地方。”

“咦? 抽离出公共的部分，跟代码复用好像啊。”

“大道至简，很多东西本来都是殊途同归。”

也就是说，第一步我们首先得找到存储对象行为的这个公共的地方。一番搜索之后，我们发现这样一段描述：

> Method area is created on virtual machine startup, shared among all Java virtual machine threads and it is logically part of heap area. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors.

Java 的对象行为（方法、函数）是存储在方法区的。

“方法区中的数据从哪来?”

“方法区中的数据是类加载时从 class 文件中提取出来的。”

“class 文件从哪来?”

“从 Java 或者其他符合 JVM 规范的源代码中编译而来。”

“源代码从哪来?”

“废话，当然是手写!”

“倒着推，手写没问题，编译没问题，至于加载……有没有办法加载一个已经加载过的类呢？如果有的话，我们就能修改字节码中目标方法所在的区域，然后重新加载这个类，这样方法区中的对象行为（方法）就被改变了，而且不改变对象的属性，也不影响已经存在对象的状态，那么就可以搞定这个问题了。可是，这岂不是违背了 JVM 的类加载原理？毕竟我们不想改变 ClassLoader。”

“少年，可以去看看 java.lang.instrument.Instrumentation。”

java.lang.instrument.Instrumentation

看完文档之后，我们发现这么两个接口：redefineClasses 和 retransformClasses。一个是重新定义 class，一个是修改 class。这两个大同小异，看 reDefineClasses 的说明：

> This method is used to replace the definition of a class without reference to the existing class file bytes, as one might do when recompiling from source for fix-and-continue debugging. Where the existing class file bytes are to be transformed (for example in bytecode instrumentation) retransformClasses should be used.

都是替换已经存在的 class 文件，redefineClasses 是自己提供字节码文件替换掉已存在的 class 文件，retransformClasses 是在已存在的字节码文件上修改后再替换之。

当然，运行时直接替换类很不安全。比如新的 class 文件引用了一个不存在的类，或者把某个类的一个 field 给删除了等等，这些情况都会引发异常。所以如文档中所言，instrument 存在诸多的限制：

> The redefinition may change method bodies, the constant pool and attributes. The redefinition must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance. These restrictions maybe be lifted in future versions. The class file bytes are not checked, verified and installed until after the transforma-

tions have been applied, if the resultant bytes are in error this method will throw an exception.

我们能做的基本上也就是简单修改方法内的一些行为，这对于我们开头的问题，打印一段日志来说，已经足够了。当然，我们除了通过 reTransform 来打印日志，还能做很多其他非常有用的事情，这个下文会进行介绍。

那怎么得到我们需要的 class 文件呢？一个最简单的方法，是把修改后的 Java 文件重新编译一遍得到 class 文件，然后调用 redefineClasses 替换。但是对于没有（或者拿不到，或者不方便修改）源码的文件我们应该怎么办呢？其实对于 JVM 来说，不管是 Java 也好，Scala 也好，任何一种符合 JVM 规范的语言的源代码，都可以编译成 class 文件。JVM 的操作对象是 class 文件，而不是源码。所以，从这种意义上来讲，我们可以说“JVM 跟语言无关”。既然如此，不管有没有源码，其实我们只需要修改 class 文件就行了。

直接操作字节码

Java 是软件开发人员能读懂的语言，class 字节码是 JVM 能读懂的语言，class 字节码最终会被 JVM 解释成机器能读懂的语言。无论哪种语言，都是人创造的。所以，理论上（实际上也确实如此）人能读懂上述任何一种语言，既然能读懂，自然能修改。只要我们愿意，我们完全可以跳过 Java 编译器，直接写字节码文件，只不过这并不符合时代的发展罢了，毕竟高级语言设计之始就是为我们人类所服务，其开发效率也比机器语言高很多。

对于人类来说，字节码文件的可读性远远没有 Java 代码高。尽管如此，还是有一些杰出的程序员们创造出了可以用来直接编辑字节码的框架，提供接口可以让我们方便地操作字节码文件，进行注入修改类的方法，动态创造一个新的类等等操作。其中最著名的框架应该就是 ASM 了，cglib、Spring 等框架中对于字节码的操作就建立在 ASM 之上。

我们都知道，Spring 的 AOP 是基于动态代理实现的，Spring 会在运行时动态创建代理类，代理类中引用被代理类，在被代理的方法执行前后进行一些神秘的

操作。那么，Spring 是怎么在运行时创建代理类的呢？动态代理的美妙之处，就在于我们不必手动为每个需要被代理的类写代理类代码，Spring 在运行时会根据需要动态地创造出一个类，这里创造的过程并非通过字符串写 Java 文件，然后编译成 class 文件，然后加载。Spring 会直接“创造”一个 class 文件，然后加载，创造 class 文件的工具，就是 ASM 了。

到这里，我们知道了用 ASM 框架直接操作 class 文件，在类中加一段打印日志的代码，然后调用 retransformClasses 就可以了。

BTrace

截止到目前，我们都是停留在理论描述的层面。那么如何进行实现呢？先来看几个问题：

1. 在我们的工程中，谁来做这个寻找字节码，修改字节码，然后 reTransform 的动作呢？我们并非先知，不可能知道未来有没有可能遇到文章开头的这种问题。考虑到性价比，我们也不可能在每个工程中都开发一段专门做这些修改字节码、重新加载字节码的代码。
2. 如果 JVM 不在本地，在远程呢？
3. 如果连 ASM 都不会用呢？能不能更通用一些，更“傻瓜”一些。

幸运的是，因为有 BTrace 的存在，我们不必自己写一套这样的工具了。什么是 BTrace 呢？ BTrace 已经开源，项目描述极其简短：

> A safe, dynamic tracing tool for the Java platform.

BTrace 是基于 Java 语言的一个安全的、可提供动态追踪服务的工具。BTrace 基于 ASM、Java Attach Api、Instruments 开发，为用户提供了很多注解。依靠这些注解，我们可以编写 BTrace 脚本（简单的 Java 代码）达到我们想要的效果，而不必深陷于 ASM 对字节码的操作中不可自拔。

看 BTrace 官方提供的一个简单例子：拦截所有 java.io 包中所有类中以 read 开头的方法，打印类名、方法名和参数名。当程序 IO 负载比较高的时候，就可以从输

出的信息中看到是哪些类所引起，是不是很方便?

```
package com.sun.btrace.samples;

import com.sun.btrace.annotations.*;
import com.sun.btrace.AnyType;
import static com.sun.btrace.BTraceUtils.*;

/**
 * This sample demonstrates regular expression
 * probe matching and getting input arguments
 * as an array - so that any overload variant
 * can be traced in "one place". This example
 * traces any "readXX" method on any class in
 * java.io package. Probed class, method and arg
 * array is printed in the action.
 */
@BTrace public class ArgArray {
    @OnMethod(
        clazz="/java\\.io\\..*/",
        method="/read.*/"
    )
    public static void anyRead(@ProbeClassName String pcn, @
ProbeMethodName String
pmn, AnyType[] args) {
        println(pcn);
        println(pmn);
        printArray(args);
    }
}
```

再来看另一个例子：每隔 2 秒打印截止到当前创建过的线程数。

```
package com.sun.btrace.samples;

import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.Export;

/**
 * This sample creates a jvmstat counter and
 * increments it everytime Thread.start() is
 * called. This thread count may be accessed
 * from outside the process. The @Export annotated
 * fields are mapped to jvmstat counters. The counter
 * name is "btrace." + <className> + "." + <fieldName>
 */
```

```
@BTrace public class ThreadCounter {

    // create a jvmstat counter using @Export
    @Export private static long count;

    @OnMethod(
        clazz="java.lang.Thread",
        method="start"
    )
    public static void onnewThread(@Self Thread t) {
        // updating counter is easy. Just assign to
        // the static field!
        count++;
    }

    @OnTimer(2000)
    public static void ontimer() {
        // we can access counter as "count" as well
        // as from jvmstat counter directly.
        println(count);
        // or equivalently ...
        println(Counters.perfLong("btrace.com.sun.btrace.samples.
ThreadCounter.count"));
    }
}
```

看了上面的用法是不是有所启发？忍不住冒出来许多想法。比如查看 HashMap 什么时候会触发 rehash，以及此时容器中有多少元素等等。

有了 BTrace，文章开头的问题可以得到完美的解决。至于 BTrace 具体有哪些功能，脚本怎么写，这些 Git 上 BTrace 工程中有大量的说明和举例，网上介绍 BTrace 用法的文章更是恒河沙数，这里就不再赘述了。

我们明白了原理，又有好用的工具支持，剩下的就是发挥我们的创造力了，只需在合适的场景下合理地进行使用即可。

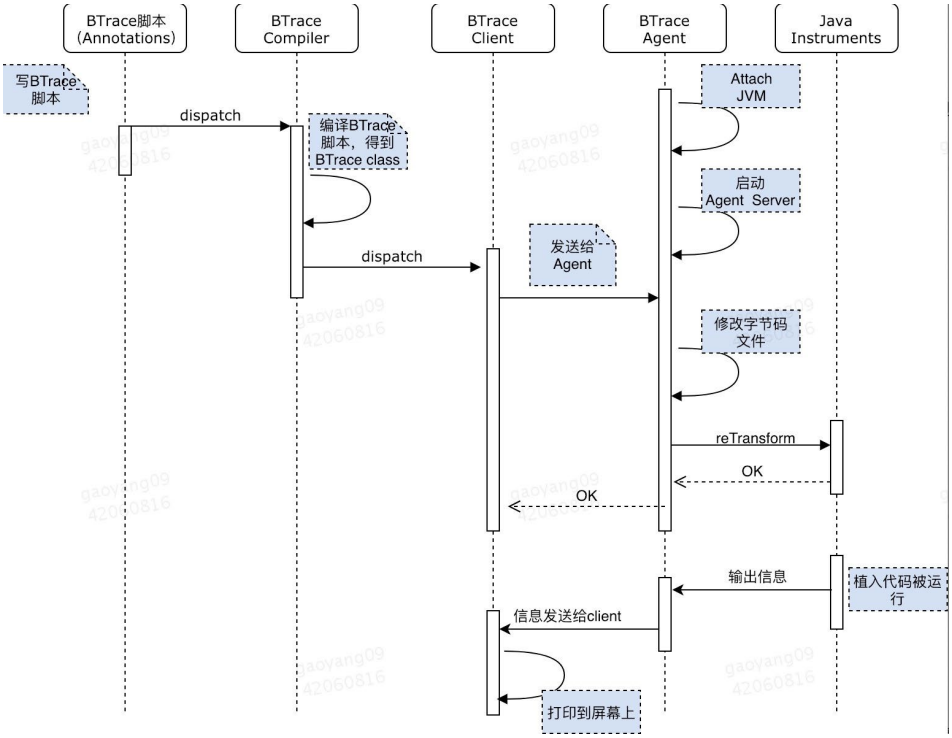
既然 BTrace 能解决上面我们提到的所有问题，那么 BTrace 的架构是怎样的呢?

BTrace 主要有下面几个模块：

1. BTrace 脚本：利用 BTrace 定义的注解，我们可以很方便地根据需要进行脚本的开发。

2. Compiler：将 BTrace 脚本编译成 BTrace class 文件。
3. Client：将 class 文件发送到 Agent。
4. Agent：基于 Java 的 Attach Api，Agent 可以动态附着到一个运行的 JVM 上，然后开启一个 BTrace Server，接收 client 发过来的 BTrace 脚本；解析脚本，然后根据脚本中的规则找到要修改的类；修改字节码后，调用 Java Instrument 的 reTransform 接口，完成对对象行为的修改并使之生效。

整个 BTrace 的架构大致如下：



BTrace 工作流程

BTrace 最终借 Instruments 实现 class 的替换。如上文所说，出于安全考虑，Instruments 在使用上存在诸多的限制，BTrace 也不例外。BTrace 对 JVM 来说是“只读的”，因此 BTrace 脚本的限制如下：

1. 不允许创建对象
2. 不允许创建数组
3. 不允许抛异常
4. 不允许 catch 异常
5. 不允许随意调用其他对象或者类的方法，只允许调用 com.sun.btrace.BTraceUtils 中提供的静态方法（一些数据处理和信息输出工具）
6. 不允许改变类的属性
7. 不允许有成员变量和方法，只允许存在 **static public void** 方法
8. 不允许有内部类、嵌套类
9. 不允许有同步方法和同步块
10. 不允许有循环
11. 不允许随意继承其他类（当然，java.lang.Object 除外）
12. 不允许实现接口
13. 不允许使用 assert
14. 不允许使用 Class 对象

如此多的限制，其实可以理解。BTrace 要做的是，虽然修改了字节码，但是除了输出需要的信息外，对整个程序的正常运行并没有影响。

Arthas

BTrace 脚本在使用上有一定的学习成本，如果能把一些常用的功能封装起来，对外直接提供简单的命令即可操作的话，那就再好不过了。阿里的工程师们早已想到这一点，就在去年（2018 年 9 月份），阿里巴巴开源了自己的 Java 诊断工具——Arthas。Arthas 提供简单的命令行操作，功能强大。究其背后的技术原理，和本文中提到的大致无二。Arthas 的文档很全面，想详细了解的话可以戳这里。

本文旨在说明 Java 动态追踪技术的来龙去脉，掌握技术背后的原理之后，只要愿意，各位读者也可以开发出自己的“冰封王座”出来。

尾声：三生万物

现在，让我们试着站在更高的地方“俯瞰”这些问题。

Java 的 Instruments 给运行时的动态追踪留下了希望，Attach API 则给运行时动态追踪提供了“出入口”，ASM 则大大方便了“人类”操作 Java 字节码的操作。

基于 Instruments 和 Attach API 前辈们创造出了诸如 JProfiler、Jvisualvm、BTrace、Arthas 这样的工具。以 ASM 为基础发展出了 cglib、动态代理，继而是应用广泛的 Spring AOP。

Java 是静态语言，运行时不允许改变数据结构。然而，Java 5 引入 Instruments，Java 6 引入 Attach API 之后，事情开始变得不一样了。虽然存在诸多限制，然而，在前辈们的努力下，仅仅是利用预留的近似于“只读”的这一点点狭小的空间，仍然创造出了各种大放异彩的技术，极大地提高了软件开发人员定位问题的效率。

计算机应该是人类有史以来最伟大的发明之一，从电磁感应磁生电，到高低电压模拟 0 和 1 的比特，再到二进制表示出几种基本类型，再到基本类型表示出无穷的对象，最后无穷的对象组合交互模拟现实生活乃至整个宇宙。

两千五百年前，《道德经》有言：“道生一，一生二，二生三，三生万物。”

两千五百年后，计算机的发展过程也大抵如此吧。

作者简介

高扬，2017 年加入美团打车，负责美团打车结算系统的开发。

字节码增强技术探索

赵泽恩

1. 字节码

1.1 什么是字节码?

Java 之所以可以“一次编译，到处运行”，一是因为 JVM 针对各种操作系统、平台都进行了定制，二是因为无论在什么平台，都可以编译生成固定格式的字节码（.class 文件）供 JVM 使用。因此，也可以看出字节码对于 Java 生态的重要性。之所以被称之为字节码，是因为字节码文件由十六进制值组成，而 JVM 以两个十六进制值为一组，即以字节为单位进行读取。在 Java 中一般是用 javac 命令编译源代码为字节码文件，一个 .java 文件从编译到运行的示例如图 1 所示。

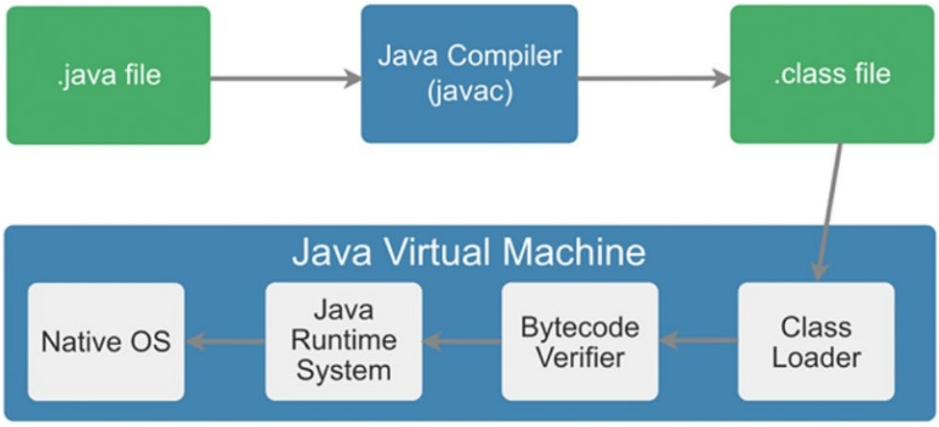


图 1　Java 运行示意图

对于开发人员，了解字节码可以更准确、直观地理解 Java 语言中更深层次的东西，比如通过字节码，可以很直观地看到 Volatile 关键字如何在字节码上生效。另外，字节码增强技术在 Spring AOP、各种 ORM 框架、热部署中的应用屡见不鲜，深入理解其原理对于我们来说大有裨益。除此之外，由于 JVM 规范的存在，只要最

终可以生成符合规范的字节码就可以在 JVM 上运行，因此这就给了各种运行在 JVM 上的语言（如 Scala、Groovy、Kotlin）一种契机，可以扩展 Java 所没有的特性或者实现各种语法糖。理解字节码后再学习这些语言，可以“逆流而上”，从字节码视角看它的设计思路，学习起来也“易如反掌”。

本文重点着眼于字节码增强技术，从字节码开始逐层向上，由 JVM 字节码操作集合到 Java 中操作字节码的框架，再到我们熟悉的各类框架原理及应用，也都会一一进行介绍。

1.2 字节码结构

.java 文件通过 javac 编译后将得到一个 .class 文件，比如编写一个简单的 ByteCodeDemo 类，如下图 2 的左侧部分：

```
1  public class ByteCodeDemo {
2      private int a = 1;
3
4      public int add() {
5          int b = 2;
6          int c = a + b;
7          System.out.println(c);
8          return c;
9      }
10 }
11
```

```
CA FE BA BE 00 00 00 34 00 24 0A 00 06 00 16 09 00 05 00 17 09 00 18 00 19 0A 00
1A 00 1B 07 00 1C 07 00 1D 01 00 01 61 01 00 01 49 01 00 06 3C 69 6E 69 74 3E 01
00 03 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61
62 6C 65 01 00 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 62 6C 65 01 00 04
74 68 69 73 01 00 1F 4C 6D 65 69 74 75 61 6E 2F 62 79 74 65 63 6F 64 65 2F 42 79
74 65 43 6F 64 65 44 65 6D 6F 3B 01 00 03 61 64 64 01 00 03 28 29 49 01 00 01 62
01 00 01 63 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 11 42 79 74 65 43 6F 64
65 44 65 6D 6F 2E 6A 61 76 61 0C 00 09 00 0A 0C 00 07 00 08 07 00 1E 0C 00 1F 00
20 07 00 21 0C 00 22 00 23 01 00 1D 6D 65 69 74 75 61 6E 2F 62 79 74 65 63 6F 64
65 2F 42 79 74 65 43 6F 64 65 44 65 6D 6F 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F
4F 62 6A 65 63 74 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00
03 6F 75 74 01 00 15 4C 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D
3B 01 00 13 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D 01 00 07 70
72 69 6E 74 6C 6E 01 00 04 28 49 29 56 00 21 00 05 00 06 00 00 00 01 00 02 00 07
00 08 00 00 00 02 00 01 00 09 00 0A 00 01 00 0B 00 00 00 38 00 02 00 01 00 00 00
0A 2A B7 00 01 2A 04 B5 00 02 B1 00 00 00 02 00 0C 00 00 00 0A 00 02 00 00 00 07
00 04 00 08 00 0D 00 00 00 0C 00 01 00 00 00 0A 00 0E 00 0F 00 00 00 01 00 10 00
11 00 01 00 0B 00 00 00 5C 00 02 00 03 00 00 00 12 05 3C 2A B4 00 02 1B 60 3D B2
00 03 1C B6 00 04 1C AC 00 00 00 02 00 0C 00 00 00 12 00 04 00 00 00 0B 00 02 00
0C 00 09 00 0D 00 10 00 0E 00 0D 00 00 00 20 00 03 00 00 00 12 00 0E 00 0F 00 00
00 02 00 10 00 12 00 08 00 01 00 09 00 09 00 13 00 08 00 02 00 01 00 14 00 00 00
02 00 15
```

图 2　示例代码（左侧）及对应的字节码（右侧）

编译后生成 ByteCodeDemo.class 文件，打开后是一堆十六进制数，按字节为单位进行分割后展示如图 2 右侧部分所示。上文提及过，JVM 对于字节码是有规范要求的，那么看似杂乱的十六进制符合什么结构呢？ JVM 规范要求每一个字节码文件都要由十部分按照固定的顺序组成，整体结构如图 3 所示。接下来我们将一一介绍这十部分：

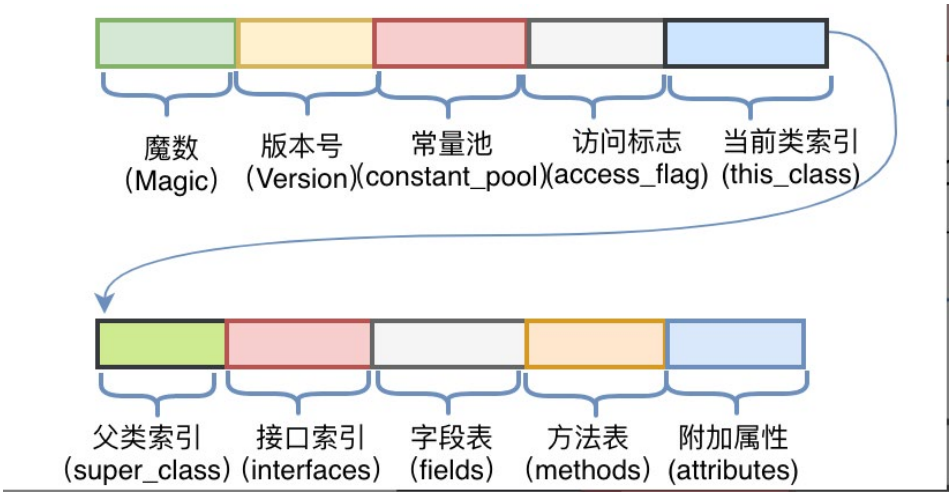


图 3　JVM 规定的字节码结构

（1）魔数（Magic Number）

所有的 .class 文件的前四个字节都是魔数，魔数的固定值为：0xCAFEBABE。魔数放在文件开头，JVM 可以根据文件的开头来判断这个文件是否可能是一个 .class 文件，如果是，才会继续进行之后的操作。

有趣的是，魔数的固定值是 Java 之父 James Gosling 制定的，为 CafeBabe（咖啡宝贝），而 Java 的图标为一杯咖啡。

（2）版本号

版本号为魔数之后的 4 个字节，前两个字节表示次版本号（Minor Version），后两个字节表示主版本号（Major Version）。上图 2 中版本号为“00 00 00 34”，次版本号转化为十进制为 0，主版本号转化为十进制为 52，在 Oracle 官网中查询序号 52 对应的主版本号为 1.8，所以编译该文件的 Java 版本号为 1.8.0。

（3）常量池（Constant Pool）

紧接着主版本号之后的字节为常量池入口。常量池中存储两类常量：字面量与符号引用。字面量为代码中声明为 Final 的常量值，符号引用如类和接口的全局限定名、字段的名称和描述符、方法的名称和描述符。常量池整体上分为两部分：常量池计数器以及常量池数据区，如下图 4 所示。

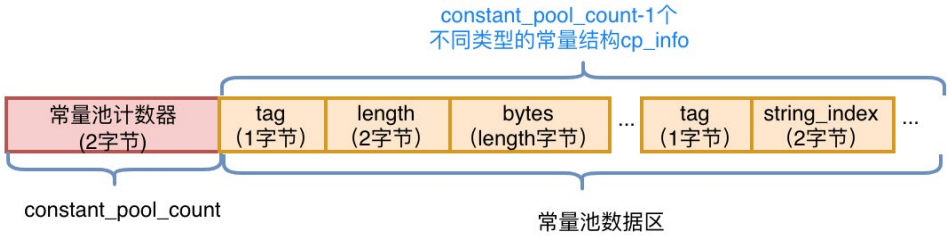


图 4　常量池的结构

- 常量池计数器（constant_pool_count）：由于常量的数量不固定，所以需要先放置两个字节来表示常量池容量计数值。图 2 中示例代码的字节码前 10 个字节如下图 5 所示，将十六进制的 24 转化为十进制值为 36，排除掉下标“0”，也就是说，这个类文件中共有 35 个常量。

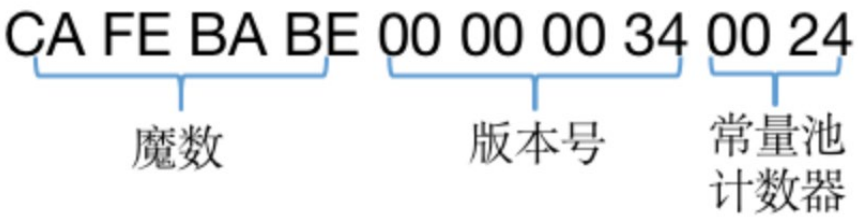


图 5　前十个字节及含义

- 常量池数据区：数据区是由（constant_pool_count−1）个 cp_info 结构组成，一个 cp_info 结构对应一个常量。在字节码中共有 14 种类型的 cp_info（如下图 6 所示），每种类型的结构都是固定的。

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	1byte	值为 1
	length	2byte	UTF-8 编码的字符串占用的字符数
	bytes	1byte	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	1byte	值为 3
	bytes	4byte	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	1byte	值为 4
	bytes	4byte	按照高位在前存储的 float 值
CONSTANT_Long_info	tag	1byte	值为 5
	bytes	8byte	按照高位在前存储 long 值
CONSTANT_Double_info	tag	1byte	值为 6
	bytes	8byte	按照高位在前存储 double 值
CONSTANT_Class_info	tag	1byte	值为 7
	index	2byte	指向全限定名常量项的索引
CONSTANT_String_info	tag	1byte	值为 8
	index	2byte	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	1byte	值为 9
	index	2byte	指向声明字段的类或者接口描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向字段描述符 CONSTANT_NameAndType 的索引项
CONSTANT_Methodref_info	tag	1byte	值为 10
	index	2byte	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_InterfaceMethodref_info	tag	1byte	值为 11
	index	2byte	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_NameAndType_info	tag	1byte	值为 12
	index	2byte	指向该字段或方法名称常量项的索引
	index	2byte	指向该字段或方法描述符常量项的索引
CONSTANT_MethodHandle_info	tag	1byte	值为 15
	reference_kind	1byte	值必须在 1~9 之间，它决定了方法句柄的类型方法句柄类型的值表示方法句柄的字节码行为
	reference_index	2byte	值必须是对常量池的有效索引
CONSTANT_MethodType_info	tag	1byte	值为 16
	descriptor_index	2byte	值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info 结构，表示方法的描述符
CONSTANT_InvokeDynamic_info	tag	1byte	值为 18
	bootstarp_method_attr_index	2byte	值必须是对当前 Class 文件中引导方法表的 bootstrap_methods[]数组的有效索引
	name_and_type_index	2byte	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_NameAndType_info 结构，表示方法名和方法描述符

图 6　各类型的 cp_info

具体以 CONSTANT_utf8_info 为例，它的结构如下图 7 左侧所示。首先一个字节“tag”，它的值取自上图 6 中对应项的 Tag，由于它的类型是 utf8_info，所以值为“01”。接下来两个字节标识该字符串的长度 Length，然后 Length 个字节为这个字符串具体的值。从图 2 中的字节码摘取一个 cp_info 结构，如下图 7 右侧所示。将它翻译过来后，其含义为：该常量类型为 utf8 字符串，长度为一字节，数据为“a”。

长度	名称	值
1字节	tag	01 对应图5中utf8_info的"标志"栏中的值
2字节	length	该utf8字符串的长度
length个字节	bytes	length个字节的具体数据

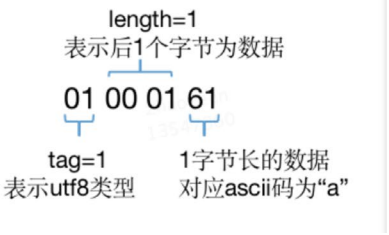


图 7　CONSTANT_utf8_info 的结构（左）及示例（右）

其他类型的 cp_info 结构在本文不再赘述，整体结构大同小异，都是先通过 Tag 来标识类型，然后后续 n 个字节来描述长度和（或）数据。先知其所以然，以后可以通过 javap-verbose ByteCodeDemo 命令，查看 JVM 反编译后的完整常量池，如下图 8 所示。可以看到反编译结果将每一个 cp_info 结构的类型和值都很明确地呈现了出来。

```
 #1 = Methodref          #6.#22         // java/lang/Object."<init>":()V
 #2 = Fieldref           #5.#23         // meituan/bytecode/ByteCodeDemo.a:I
 #3 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream
 #4 = Methodref          #26.#27        // java/io/PrintStream.println:(I)V
 #5 = Class              #28            // meituan/bytecode/ByteCodeDemo
 #6 = Class              #29            // java/lang/Object
 #7 = Utf8               a
 #8 = Utf8               I
 #9 = Utf8               <init>
#10 = Utf8               ()V
#11 = Utf8               Code
#12 = Utf8               LineNumberTable
#13 = Utf8               LocalVariableTable
#14 = Utf8               this
#15 = Utf8               Lmeituan/bytecode/ByteCodeDemo;
#16 = Utf8               add
#17 = Utf8               ()I
#18 = Utf8               b
#19 = Utf8               c
#20 = Utf8               SourceFile
#21 = Utf8               ByteCodeDemo.java
#22 = NameAndType        #9:#10         // "<init>":()V
#23 = NameAndType        #7:#8          // a:I
#24 = Class              #30            // java/lang/System
#25 = NameAndType        #31:#32        // out:Ljava/io/PrintStream;
#26 = Class              #33            // java/io/PrintStream
#27 = NameAndType        #34:#35        // println:(I)V
#28 = Utf8               meituan/bytecode/ByteCodeDemo
#29 = Utf8               java/lang/Object
#30 = Utf8               java/lang/System
#31 = Utf8               out
#32 = Utf8               Ljava/io/PrintStream;
#33 = Utf8               java/io/PrintStream
#34 = Utf8               println
```

图 8　常量池反编译结果

（4）访问标志

常量池结束之后的两个字节，描述该 Class 是类还是接口，以及是否被 Public、Abstract、Final 等修饰符修饰。JVM 规范规定了如下图 9 的访问标志（Access_Flag）。需要注意的是，JVM 并没有穷举所有的访问标志，而是使用按位或操作来进行描述的，比如某个类的修饰符为 Public Final，则对应的访问修饰符的值为 ACC_PUBLIC | ACC_FINAL，即 0x0001 | 0x0010=0x0011。

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为public
ACC_PRIVATE	0x0002	字段是否为private
ACC_PROTECTED	0x0004	字段是否为protected
ACC_STATIC	0x0008	字段是否为static
ACC_FINAL	0x0010	字段是否为final
ACC_VOLATILE	0x0040	字段是否为volatile
ACC_TRANSTENT	0x0080	字段是否为transient
ACC_SYNCHETIC	0x1000	字段是否为由编译器自动产生
ACC_ENUM	0x4000	字段是否为enum

图 9　访问标志

（5）当前类名

访问标志后的两个字节，描述的是当前类的全限定名。这两个字节保存的值为常量池中的索引值，根据索引值就能在常量池中找到这个类的全限定名。

（6）父类名称

当前类名后的两个字节，描述父类的全限定名，同上，保存的也是常量池中的索引值。

（7）接口信息

父类名称后为两字节的接口计数器，描述了该类或父类实现的接口数量。紧接着的 n 个字节是所有接口名称的字符串常量的索引值。

（8）字段表

字段表用于描述类和接口中声明的变量，包含类级别的变量以及实例变量，但是不包含方法内部声明的局部变量。字段表也分为两部分，第一部分为两个字节，描述字段个数；第二部分是每个字段的详细信息 fields_info。字段表结构如下图所示：

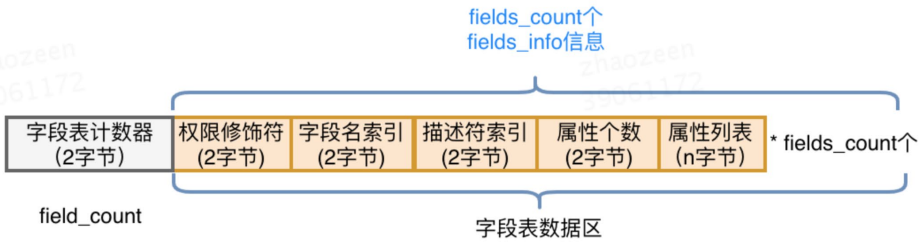


图 10　字段表结构

以图 2 中字节码的字段表为例，如下图 11 所示。其中字段的访问标志查图 9，0002 对应为 Private。通过索引下标在图 8 中常量池分别得到字段名为“a”，描述符为“I”（代表 int）。综上，就可以唯一确定出一个类中声明的变量 private int a。



图 11　字段表示例

（9）方法表

字段表结束后为方法表，方法表也是由两部分组成，第一部分为两个字节描述方法的个数；第二部分为每个方法的详细信息。方法的详细信息较为复杂，包括方法的访问标志、方法名、方法的描述符以及方法的属性，如下图所示：

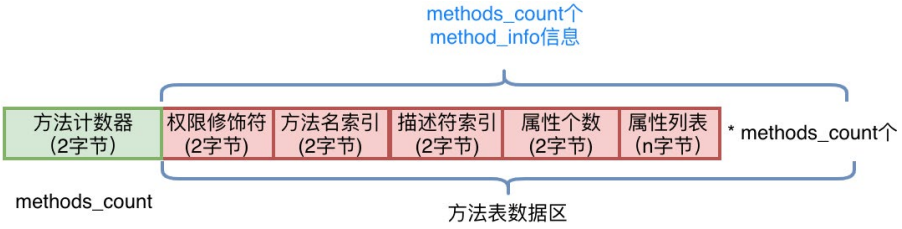


图 12　方法表结构

方法的权限修饰符依然可以通过图 9 的值查询得到，方法名和方法的描述符都是常量池中的索引值，可以通过索引值在常量池中找到。而“方法的属性”这一部分较为复杂，直接借助 javap -verbose 将其反编译为人可以读懂的信息进行解读，如图 13 所示。可以看到属性中包括以下三个部分：

```
public int add();                                   //方法名
  descriptor: ()I                                   //方法的描述符
  flags: ACC_PUBLIC                                 //方法的访问标志
  Code:                                             //Code开始
    stack=2, locals=3, args_size=1                  //最大栈深度、局部变量数、参数列表size
       0: iconst_2                                  //0-17为.java中的代码对应的操作码
       1: istore_1
       2: aload_0
       3: getfield      #2                         // 引用常量池#2的值: Field a:I
       6: iload_1
       7: iadd
       8: istore_2
       9: getstatic     #3                        // 引用常量池#3的值: Field
                                                  //java/lang/System.out:Ljava/io/PrintStrea
      12: iload_2
      13: invokevirtual #4                         //引用常量池#4的值
                                                   //Method java/io/PrintStream.println:(I)V
      16: iload_2
      17: ireturn
    LineNumberTable:                              //行号表，将上面的操作码与.java中的行号做对应
      line 11: 0
      line 12: 2
      line 13: 9
      line 14: 16
    LocalVariableTable:                           //本地变量表，包含局部变量的定义
      Start  Length  Slot  Name   Signature
          0      18     0  this   Lmeituan/bytecode/ByteCodeDemo;    //slot=0为this
          2      16     1     b   I
          9       9     2     c   I
```

图 13　反编译后的方法表

- “Code 区”：源代码对应的 JVM 指令操作码，在进行字节码增强时重点操作的就是“Code 区”这一部分。
- “LineNumberTable”：行号表，将 Code 区的操作码和源代码中的行号对应，Debug 时会起到作用（源代码走一行，需要走多少个 JVM 指令操作码）。
- “LocalVariableTable”：本地变量表，包含 This 和局部变量，之所以可以在每一个方法内部都可以调用 This，是因为 JVM 将 This 作为每一个方法的第一个参数隐式进行传入。当然，这是针对非 Static 方法而言。

（10）附加属性表

字节码的最后一部分，该项存放了在该文件中类或接口所定义属性的基本信息。

1.3 字节码操作集合

在上图 13 中，Code 区的红色编号 0 ~ 17，就是 .java 中的方法源代码编译后让 JVM 真正执行的操作码。为了帮助人们理解，反编译后看到的是十六进制操作码所对应的助记符，十六进制值操作码与助记符的对应关系，以及每一个操作码的用处可以查看 Oracle 官方文档进行了解，在需要用到时进行查阅即可。比如上图中第一个助记符为 iconst_2，对应到图 2 中的字节码为 0x05，用处是将 int 值 2 压入操作数栈中。以此类推，对 0~17 的助记符理解后，就是完整的 add() 方法的实现。

1.4 操作数栈和字节码

JVM 的指令集是基于栈而不是寄存器，基于栈可以具备很好的跨平台性（因为寄存器指令集往往和硬件挂钩），但缺点在于，要完成同样的操作，基于栈的实现需要更多指令才能完成（因为栈只是一个 FILO 结构，需要频繁压栈出栈）。另外，由于栈是在内存实现的，而寄存器是在 CPU 的高速缓存区，相较而言，基于栈的速度要慢很多，这也是为了跨平台性而做出的牺牲。

我们在上文所说的操作码或者操作集合，其实控制的就是这个 JVM 的操作数栈。为了更直观地感受操作码是如何控制操作数栈的，以及理解常量池、变量表的作用，将 add() 方法的对操作数栈的操作制作为 GIF，如下图 14 所示，图中仅截取了常量池中被引用的部分，以指令 iconst_2 开始到 ireturn 结束，与图 13 中 Code 区

0~17 的指令一一对应：

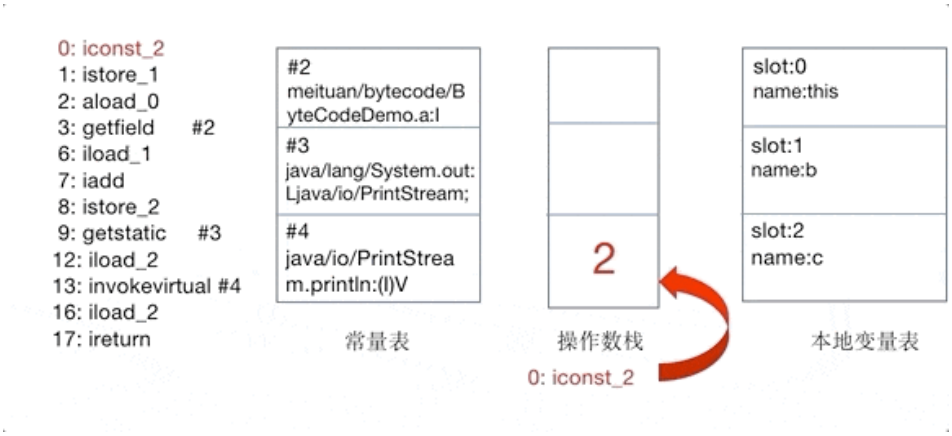


图 14　控制操作数栈示意图

1.5　查看字节码工具

如果每次查看反编译后的字节码都使用 javap 命令的话，好非常繁琐。这里推荐一个 Idea 插件：jclasslib。使用效果如图 15 所示，代码编译后在菜单栏”View”中选择”Show Bytecode With jclasslib”，可以很直观地看到当前字节码文件的类信息、常量池、方法区等信息。

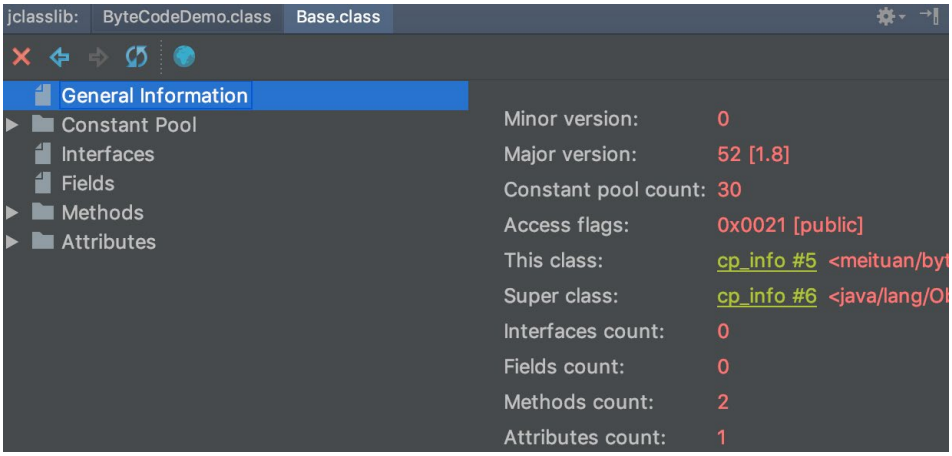


图 15　jclasslib 查看字节码

2. 字节码增强

在上文中，着重介绍了字节码的结构，这为我们了解字节码增强技术的实现打下了基础。字节码增强技术就是一类对现有字节码进行修改或者动态生成全新字节码文件的技术。接下来，我们将从最直接操纵字节码的实现方式开始深入进行剖析。

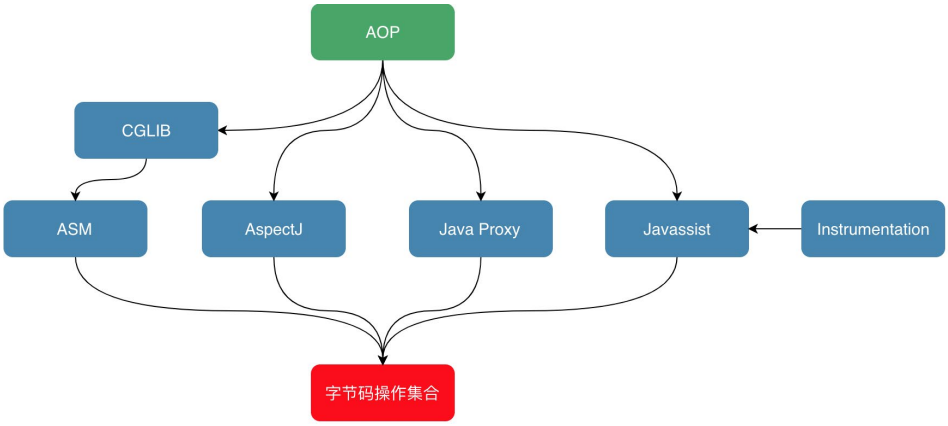


图 16　字节码增强技术

2.1　ASM

对于需要手动操纵字节码的需求，可以使用 ASM，它可以直接生产 .class 字节码文件，也可以在类被加载入 JVM 之前动态修改类行为（如下图 17 所示）。ASM 的应用场景有 AOP（Cglib 就是基于 ASM）、热部署、修改其他 jar 包中的类等。当然，涉及到如此底层的步骤，实现起来也比较麻烦。接下来，本文将介绍 ASM 的两种 API，并用 ASM 来实现一个比较粗糙的 AOP。但在此之前，为了让大家更快地理解 ASM 的处理流程，强烈建议读者先对访问者模式进行了解。简单来说，访问者模式主要用于修改或操作一些数据结构比较稳定的数据，而通过第一章，我们知道字节码文件的结构是由 JVM 固定的，所以很适合利用访问者模式对字节码文件进行修改。

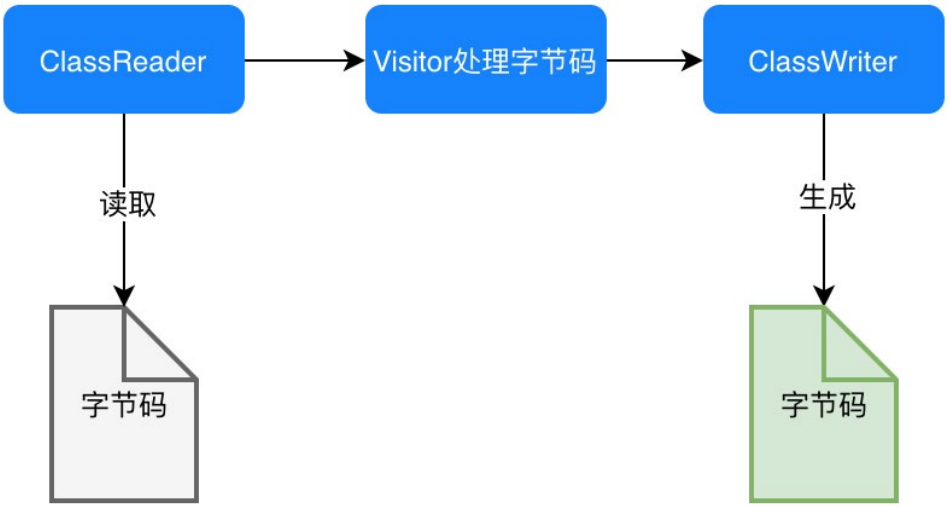


图 17　ASM 修改字节码

2.1.1　ASM API

2.1.1.1　核心 API

ASM Core API 可以类比解析 XML 文件中的 SAX 方式，不需要把这个类的整个结构读取进来，就可以用流式的方法来处理字节码文件。好处是非常节约内存，但是编程难度较大。然而出于性能考虑，一般情况下编程都使用 Core API。在 Core API 中有以下几个关键类：

- ClassReader：用于读取已经编译好的 .class 文件。
- ClassWriter：用于重新构建编译后的类，如修改类名、属性以及方法，也可以生成新的类的字节码文件。
- 各种 Visitor 类：如上所述，CoreAPI 根据字节码从上到下依次处理，对于字节码文件中不同的区域有不同的 Visitor，比如用于访问方法的 MethodVisitor、用于访问类变量的 FieldVisitor、用于访问注解的 AnnotationVisitor 等。为了实现 AOP，重点要使用的是 MethodVisitor。

2.1.1.2　树形 API

ASM Tree API 可以类比解析 XML 文件中的 DOM 方式，把整个类的结构读

取到内存中，缺点是消耗内存多，但是编程比较简单。TreeApi 不同于 CoreAPI，TreeAPI 通过各种 Node 类来映射字节码的各个区域，类比 DOM 节点，就可以很好地理解这种编程方式。

2.1.2 直接利用 ASM 实现 AOP

利用 ASM 的 CoreAPI 来增强类。这里不纠结于 AOP 的专业名词如切片、通知，只实现在方法调用前、后增加逻辑，通俗易懂且方便理解。首先定义需要被增强的 Base 类：其中只包含一个 process() 方法，方法内输出一行“process”。增强后，我们期望的是，方法执行前输出“start”，之后输出”end”。

```
public class Base {
    public void process(){
        System.out.println("process");
    }
}
```

为了利用 ASM 实现 AOP，需要定义两个类：一个是 MyClassVisitor 类，用于对字节码的 visit 以及修改；另一个是 Generator 类，在这个类中定义 ClassReader 和 ClassWriter，其中的逻辑是，classReader 读取字节码，然后交给 MyClassVisitor 类处理，处理完成后由 ClassWriter 写字节码并将旧的字节码替换掉。Generator 类较简单，我们先看一下它的实现，如下所示，然后重点解释 MyClassVisitor 类。

```
import org.objectweb.asm.ClassReader;
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.ClassWriter;

public class Generator {
    public static void main(String[] args) throws Exception {
            // 读取
        ClassReader classReader = new ClassReader("meituan/bytecode/asm/
Base");
        ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_
MAXS);
        // 处理
        ClassVisitor classVisitor = new MyClassVisitor(classWriter);
        classReader.accept(classVisitor, ClassReader.SKIP_DEBUG);
```

```
        byte[] data = classWriter.toByteArray();
        // 输出
        File f = new File("operation-server/target/classes/meituan/
bytecode/asm/Base.class");
        FileOutputStream fout = new FileOutputStream(f);
        fout.write(data);
        fout.close();
        System.out.println("now generator cc success!!!!!");
    }
}
```

MyClassVisitor 继承自 ClassVisitor，用于对字节码的观察。它还包含一个内部类 MyMethodVisitor，继承自 MethodVisitor 用于对类内方法的观察，它的整体代码如下：

```
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

public class MyClassVisitor extends ClassVisitor implements Opcodes {
    public MyClassVisitor(ClassVisitor cv) {
        super(ASM5, cv);
    }
    @Override
    public void visit(int version, int access, String name, String
signature,
                      String superName, String[] interfaces) {
        cv.visit(version, access, name, signature, superName,
interfaces);
    }
    @Override
    public MethodVisitor visitMethod(int access, String name, String
desc, String signature,
String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
                exceptions);
        //Base 类中有两个方法：无参构造以及 process 方法，这里不增强构造方法
        if (!name.equals("<init>") && mv != null) {
            mv = new MyMethodVisitor(mv);
        }
        return mv;
    }
    class MyMethodVisitor extends MethodVisitor implements Opcodes {
        public MyMethodVisitor(MethodVisitor mv) {
            super(Opcodes.ASM5, mv);
        }
```

```
        @Override
        public void visitCode() {
            super.visitCode();
            mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
            mv.visitLdcInsn("start");
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/
String;)V", false);
        }
        @Override
        public void visitInsn(int opcode) {
            if ((opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN)
                    || opcode == Opcodes.ATHROW) {
                // 方法在返回之前，打印 "end"
                mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
                mv.visitLdcInsn("end");
                mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/
String;)V", false);
            }
            mv.visitInsn(opcode);
        }
    }
}
```

利用这个类就可以实现对字节码的修改。详细解读其中的代码，对字节码做修改的步骤是：

- 首先通过 MyClassVisitor 类中的 visitMethod 方法，判断当前字节码读到哪一个方法了。跳过构造方法 <init> 后，将需要被增强的方法交给内部类 MyMethodVisitor 来进行处理。
- 接下来，进入内部类 MyMethodVisitor 中的 visitCode 方法，它会在 ASM 开始访问某一个方法的 Code 区时被调用，重写 visitCode 方法，将 AOP 中的前置逻辑就放在这里。
- MyMethodVisitor 继续读取字节码指令，每当 ASM 访问到无参数指令时，都会调用 MyMethodVisitor 中的 visitInsn 方法。我们判断了当前指令是否为无参数的“return”指令，如果是就在它的前面添加一些指令，也就是将 AOP

的后置逻辑放在该方法中。

- 综上，重写 MyMethodVisitor 中的两个方法，就可以实现 AOP 了，而重写方法时就需要用 ASM 的写法，手动写入或者修改字节码。通过调用 methodVisitor 的 visitXXXXInsn() 方法就可以实现字节码的插入，XXXX 对应相应的操作码助记符类型，比如 mv.visitLdcInsn(“end”) 对应的操作码就是 ldc“end”，即将字符串“end”压入栈。

完成这两个 visitor 类后，运行 Generator 中的 main 方法完成对 Base 类的字节码增强，增强后的结果可以在编译后的 target 文件夹中找到 Base.class 文件进行查看，可以看到反编译后的代码已经改变了（如图 18 左侧所示）。然后写一个测试类 MyTest，在其中 new Base()，并调用 base.process() 方法，可以看到下图右侧所示的 AOP 实现效果：

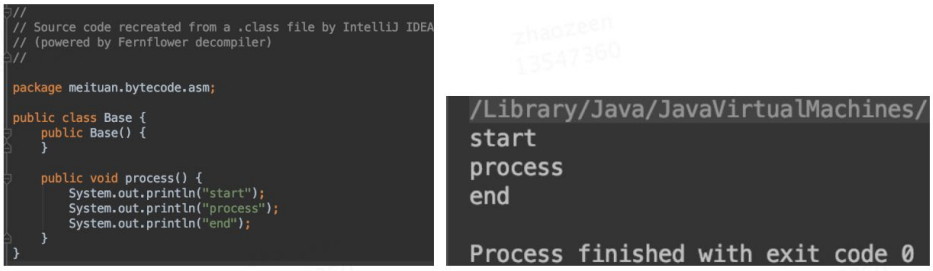


图 18　ASM 实现 AOP 的效果

2.1.3　ASM 工具

利用 ASM 手写字节码时，需要利用一系列 visitXXXXInsn() 方法来写对应的助记符，所以需要先将每一行源代码转化为一个个的助记符，然后通过 ASM 的语法转换为 visitXXXXInsn() 这种写法。第一步将源码转化为助记符就已经够麻烦了，不熟悉字节码操作集合的话，需要我们将代码编译后再反编译，才能得到源代码对应的助记符。第二步利用 ASM 写字节码时，如何传参也很令人头疼。ASM 社区也知道这两个问题，所以提供了工具 ASM ByteCode Outline。

安装后，右键选择“Show Bytecode Outline”，在新标签页中选择“ASMi-

fied”这个 tab，如图 19 所示，就可以看到这个类中的代码对应的 ASM 写法了。图中上下两个红框分别对应 AOP 中的前置逻辑于后置逻辑，将这两块直接复制到 visitor 中的 visitMethod() 以及 visitInsn() 方法中，就可以了。

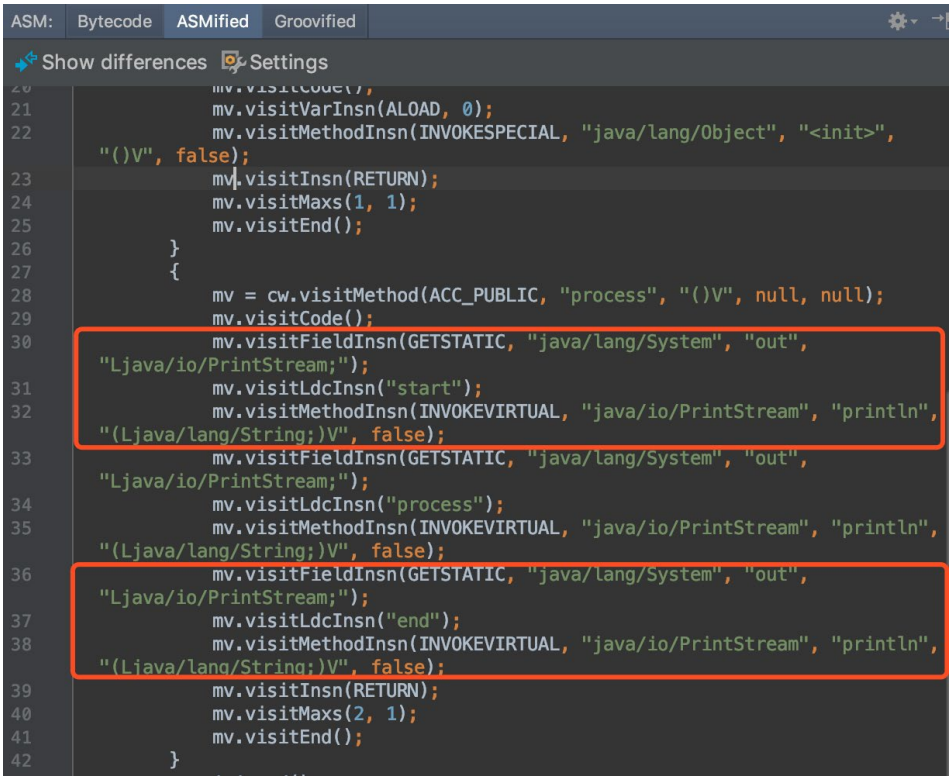


图 19　ASM Bytecode Outline

2.2　Javassist

ASM 是在指令层次上操作字节码的，阅读上文后，我们的直观感受是在指令层次上操作字节码的框架实现起来比较晦涩。故除此之外，我们再简单介绍另外一类框架：强调源代码层次操作字节码的框架 Javassist。

利用 Javassist 实现字节码增强时，可以无须关注字节码刻板的结构，其优点就在于编程简单。直接使用 java 编码的形式，而不需要了解虚拟机指令，就能动态改变类的结构或者动态生成类。其中最重要的是 ClassPool、CtClass、CtMethod、

CtField 这四个类：

- CtClass（compile-time class）：编译时类信息，它是一个 class 文件在代码中的抽象表现形式，可以通过一个类的全限定名来获取一个 CtClass 对象，用来表示这个类文件。
- ClassPool：从开发视角来看，ClassPool 是一张保存 CtClass 信息的 HashTable，key 为类名，value 为类名对应的 CtClass 对象。当我们需要对某个类进行修改时，就是通过 pool.getCtClass(“className”) 方法从 pool 中获取到相应的 CtClass。
- CtMethod、CtField：这两个比较好理解，对应的是类中的方法和属性。

了解这四个类后，我们可以写一个小 Demo 来展示 Javassist 简单、快速的特点。我们依然是对 Base 中的 process() 方法做增强，在方法调用前后分别输出”start”和”end”，实现代码如下。我们需要做的就是从 pool 中获取到相应的 CtClass 对象和其中的方法，然后执行 method.insertBefore 和 insertAfter 方法，参数为要插入的 Java 代码，再以字符串的形式传入即可，实现起来也极为简单。

```
import com.meituan.mtrace.agent.javassist.*;

public class JavassistTest {
    public static void main(String[] args) throws NotFoundException,
CannotCompileException, IllegalAccessException, InstantiationException,
IOException {
        ClassPool cp = ClassPool.getDefault();
        CtClass cc = cp.get("meituan.bytecode.javassist.Base");
        CtMethod m = cc.getDeclaredMethod("process");
        m.insertBefore("{ System.out.println(\"start\"); }");
        m.insertAfter("{ System.out.println(\"end\"); }");
        Class c = cc.toClass();
        cc.writeFile("/Users/zen/projects");
        Base h = (Base)c.newInstance();
        h.process();
    }
}
```

3. 运行时类的重载

3.1 问题引出

上一章重点介绍了两种不同类型的字节码操作框架，且都利用它们实现了较为粗糙的 AOP。其实，为了方便大家理解字节码增强技术，在上文中我们避重就轻将 ASM 实现 AOP 的过程分为了两个 main 方法：第一个是利用 MyClassVisitor 对已编译好的 class 文件进行修改，第二个是 new 对象并调用。这期间并不涉及到 JVM 运行时对类的重加载，而是在第一个 main 方法中，通过 ASM 对已编译类的字节码进行替换，在第二个 main 方法中，直接使用已替换好的新类信息。另外在 Javassist 的实现中，我们也只加载了一次 Base 类，也不涉及到运行时重加载类。

如果我们在一个 JVM 中，先加载了一个类，然后又对其进行字节码增强并重新加载会发生什么呢？模拟这种情况，只需要我们在上文中 Javassist 的 Demo 中 main() 方法的第一行添加 Base b=new Base()，即在增强前就先让 JVM 加载 Base 类，然后在执行到 c.toClass() 方法时会抛出错误，如下图 20 所示。跟进 c.toClass() 方法中，我们会发现它是在最后调用了 ClassLoader 的 native 方法 defineClass() 时报错。也就是说，JVM 是不允许在运行时动态重载一个类的。



图 20　运行时重复 load 类的错误信息

显然，如果只能在类加载前对类进行强化，那字节码增强技术的使用场景就变得很窄了。我们期望的效果是：在一个持续运行并已经加载了所有类的 JVM 中，还能利用字节码增强技术对其中的类行为做替换并重新加载。为了模拟这种情况，我们将 Base 类做改写，在其中编写 main 方法，每五秒调用一次 process() 方法，在 process() 方法中输出一行“process”。

我们的目的就是，在 JVM 运行中的时候，将 process() 方法做替换，在其前后分别打印“start”和“end”。也就是在运行中时，每五秒打印的内容由”process”

变为打印”start process end”。那如何解决 JVM 不允许运行时重加载类信息的问题呢？为了达到这个目的，我们接下来一一来介绍需要借助的 Java 类库。

```
import java.lang.management.ManagementFactory;

public class Base {
    public static void main(String[] args) {
        String name = ManagementFactory.getRuntimeMXBean().getName();
        String s = name.split("@")[0];
        //打印当前 Pid
        System.out.println("pid:"+s);
        while (true) {
            try {
                Thread.sleep(5000L);
            } catch (Exception e) {
                break;
            }
            process();
        }
    }

    public static void process() {
        System.out.println("process");
    }
}
```

3.2 Instrument

instrument 是 JVM 提供的一个可以修改已加载类的类库，专门为 Java 语言编写的插桩服务提供支持。它需要依赖 JVMTI 的 Attach API 机制实现，JVMTI 这一部分，我们将在下一小节进行介绍。在 JDK 1.6 以前，instrument 只能在 JVM 刚启动开始加载类时生效，而在 JDK 1.6 之后，instrument 支持了在运行时对类定义的修改。要使用 instrument 的类修改功能，我们需要实现它提供的 ClassFileTransformer 接口，定义一个类文件转换器。接口中的 transform() 方法会在类文件被加载时调用，而在 transform 方法里，我们可以利用上文中的 ASM 或 Javassist 对传入的字节码进行改写或替换，生成新的字节码数组后返回。

我们定义一个实现了 ClassFileTransformer 接口的类 TestTransformer，依然在其中利用 Javassist 对 Base 类中的 process() 方法进行增强，在前后分别打印

“start”和“end”，代码如下：

```
import java.lang.instrument.ClassFileTransformer;

public class TestTransformer implements ClassFileTransformer {
    @Override
    public byte[] transform(ClassLoader loader, String className,
Class<?> classBeingRedefined,
ProtectionDomain protectionDomain, byte[] classfileBuffer) {
        System.out.println("Transforming " + className);
        try {
            ClassPool cp = ClassPool.getDefault();
            CtClass cc = cp.get("meituan.bytecode.jvmti.Base");
            CtMethod m = cc.getDeclaredMethod("process");
            m.insertBefore("{ System.out.println(\"start\"); }");
            m.insertAfter("{ System.out.println(\"end\"); }");
            return cc.toBytecode();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

现在有了 Transformer，那么它要如何注入到正在运行的 JVM 呢？还需要定义一个 Agent，借助 Agent 的能力将 Instrument 注入到 JVM 中。我们将在下一小节介绍 Agent，现在要介绍的是 Agent 中用到的另一个类 Instrumentation。在 JDK 1.6 之后，Instrumentation 可以做启动后的 Instrument、本地代码（Native Code）的 Instrument，以及动态改变 Classpath 等等。我们可以向 Instrumentation 中添加上文中定义的 Transformer，并指定要被重加载的类，代码如下所示。这样，当 Agent 被 Attach 到一个 JVM 中时，就会执行类字节码替换并重载入 JVM 的操作。

```
import java.lang.instrument.Instrumentation;

public class TestAgent {
    public static void agentmain(String args, Instrumentation inst) {
        // 指定我们自己定义的 Transformer，在其中利用 Javassist 做字节码替换
        inst.addTransformer(new TestTransformer(), true);
        try {
            // 重定义类并载入新的字节码
            inst.retransformClasses(Base.class);
            System.out.println("Agent Load Done.");
```

```
        } catch (Exception e) {
            System.out.println("agent load failed!");
        }
    }
}
```

3.3 JVMTI & Agent & Attach API

上一小节中，我们给出了 Agent 类的代码，追根溯源需要先介绍 JPDA（Java Platform Debugger Architecture）。如果 JVM 启动时开启了 JPDA，那么类是允许被重新加载的。在这种情况下，已被加载的旧版本类信息可以被卸载，然后重新加载新版本的类。正如 JDPA 名称中的 Debugger，JDPA 其实是一套用于调试 Java 程序的标准，任何 JDK 都必须实现该标准。

JPDA 定义了一整套完整的体系，它将调试体系分为三部分，并规定了三者之间的通信接口。三部分由低到高分别是 Java 虚拟机工具接口（JVMTI），Java 调试协议（JDWP）以及 Java 调试接口（JDI），三者之间的关系如下图所示：

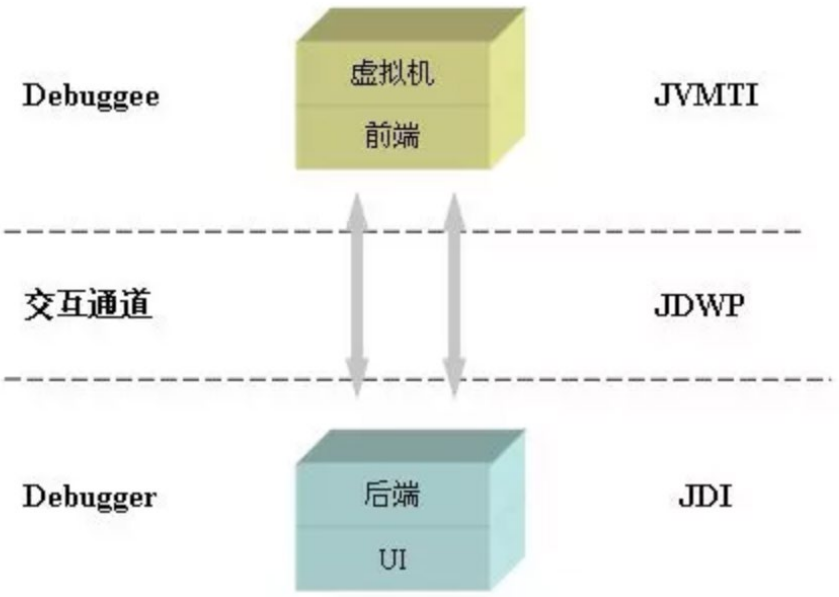


图 21 JPDA

现在回到正题，我们可以借助 JVMTI 的一部分能力，帮助动态重载类信息。JVM TI（JVM TOOL INTERFACE，JVM 工具接口）是 JVM 提供的一套对 JVM 进行操作的工具接口。通过 JVMTI，可以实现对 JVM 的多种操作，它通过接口注册各种事件勾子，在 JVM 事件触发时，同时触发预定义的勾子，以实现对各个 JVM 事件的响应，事件包括类文件加载、异常产生与捕获、线程启动和结束、进入和退出临界区、成员变量修改、GC 开始和结束、方法调用进入和退出、临界区竞争与等待、VM 启动与退出等等。

而 Agent 就是 JVMTI 的一种实现，Agent 有两种启动方式，一是随 Java 进程启动而启动，经常见到的 java -agentlib 就是这种方式；二是运行时载入，通过 attach API，将模块（jar 包）动态地 Attach 到指定进程 id 的 Java 进程内。

Attach API 的作用是提供 JVM 进程间通信的能力，比如说我们为了让另外一个 JVM 进程把线上服务的线程 Dump 出来，会运行 jstack 或 jmap 的进程，并传递 pid 的参数，告诉它要对哪个进程进行线程 Dump，这就是 Attach API 做的事情。在下面，我们将通过 Attach API 的 loadAgent() 方法，将打包好的 Agent jar 包动态 Attach 到目标 JVM 上。具体实现起来的步骤如下：

- 定义 Agent，并在其中实现 AgentMain 方法，如上一小节中定义的代码块 7 中的 TestAgent 类；
- 然后将 TestAgent 类打成一个包含 MANIFEST.MF 的 jar 包，其中 MANIFEST.MF 文件中将 Agent-Class 属性指定为 TestAgent 的全限定名，如下图所示；

```
Manifest-Version: 1.0
Agent-Class: meituan.bytecode.jvmti.TestAgent
Created-By: zhaozeen
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Boot-Class-Path: javassist-3.20.0-GA.jar
```

图 22　Manifest.mf

- 最后利用 Attach API，将我们打包好的 jar 包 Attach 到指定的 JVM pid 上，代码如下：

```
import com.sun.tools.attach.VirtualMachine;

public class Attacher {
    public static void main(String[] args) throws
AttachNotSupportedException, IOException,
AgentLoadException, AgentInitializationException {
        // 传入目标 JVM pid
        VirtualMachine vm = VirtualMachine.attach("39333");
        vm.loadAgent("/Users/zen/operation_server_jar/operation-
server.jar");
    }
}
```

- 由于在 MANIFEST.MF 中指定了 Agent-Class，所以在 Attach 后，目标 JVM 在运行时会走到 TestAgent 类中定义的 agentmain() 方法，而在这个方法中，我们利用 Instrumentation，将指定类的字节码通过定义的类转化器 TestTransformer 做了 Base 类的字节码替换（通过 javassist），并完成了类的重新加载。由此，我们达成了“在 JVM 运行时，改变类的字节码并重新载入类信息”的目的。

以下为运行时重新载入类的效果：先运行 Base 中的 main() 方法，启动一个 JVM，可以在控制台看到每隔五秒输出一次”process”。接着执行 Attacher 中的 main() 方法，并将上一个 JVM 的 pid 传入。此时回到上一个 main() 方法的控制台，可以看到现在每隔五秒输出”process”前后会分别输出”start”和”end”，也就是说完成了运行时的字节码增强，并重新载入了这个类。

```
process
process
process
process
objc[32480]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/
WARNING: javassist-3.20.0-GA.jar not added to bootstrap class loader search: Illegal argument or not JAR file
Transforming meituan/bytecode/jvmti/Base
Agent Load Done.
start
process
end
start
process
end
```

图 23　运行时重载入类的效果

3.4 使用场景

至此，字节码增强技术的可使用范围就不再局限于 JVM 加载类前了。通过上述几个类库，我们可以在运行时对 JVM 中的类进行修改并重载了。通过这种手段，可以做的事情就变得很多了：

- 热部署：不部署服务而对线上服务做修改，可以做打点、增加日志等操作。
- Mock：测试时候对某些服务做 Mock。
- 性能诊断工具：比如 bTrace 就是利用 Instrument，实现无侵入地跟踪一个正在运行的 JVM，监控到类和方法级别的状态信息。

4. 总结

字节码增强技术相当于是一把打开运行时 JVM 的钥匙，利用它可以动态地对运行中的程序做修改，也可以跟踪 JVM 运行中程序的状态。此外，我们平时使用的动态代理、AOP 也与字节码增强密切相关，它们实质上还是利用各种手段生成符合规范的字节码文件。综上所述，掌握字节码增强后可以高效地定位并快速修复一些棘手的问题（如线上性能问题、方法出现不可控的出入参需要紧急加日志等问题），也可以在开发中减少冗余代码，大大提高开发效率。

5. 参考文献

- 《ASM4-Guide》
- Oracle:The class File Format
- Oracle:The Java Virtual Machine Instruction Set
- javassist tutorial
- JVM Tool Interface - Version 1.2

作者简介

泽恩，美团点评研发工程师。

招聘信息

美团到店住宿业务研发团队负责美团酒店核心业务系统建设，致力于通过技术践行“帮大家住得更好”的使命。美团酒店屡次刷新行业记录，最近 12 个月酒店预订间夜量达到 3 个亿，单日

入住间夜量峰值突破 280 万。团队的愿景是：建设打造旅游住宿行业一流的技术架构，从质量、安全、效率、性能多角度保障系统高速发展。

美团到店事业群住宿业务研发团队现诚聘后台开发工程师 / 技术专家，欢迎有兴趣的同学投简历至：tech@meituan.com（注明：美团到店事业群住宿业务研发团队）

JVM CPU Profiler 技术原理及源码深度解析

业祥　继东

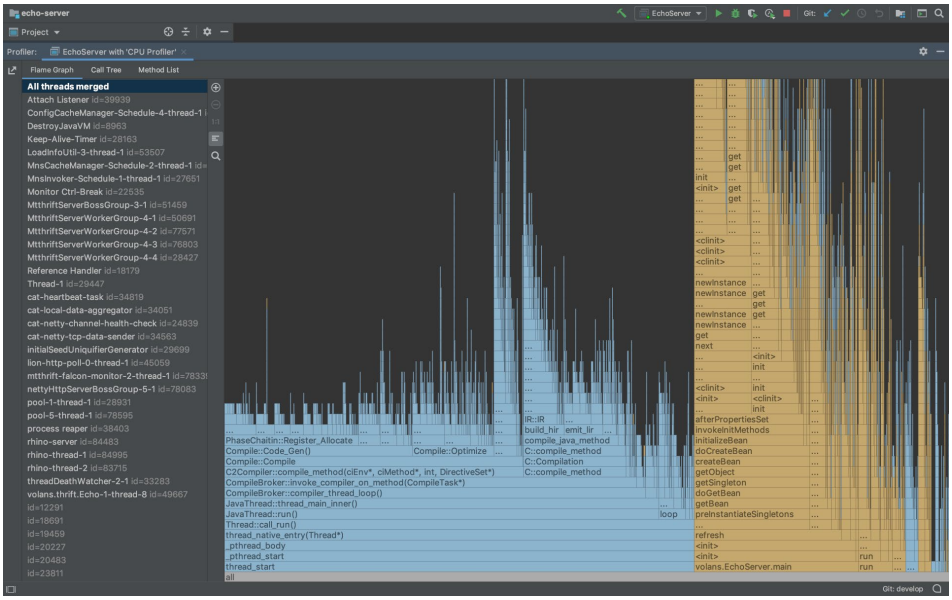
研发人员在遇到线上报警或需要优化系统性能时，常常需要分析程序运行行为和性能瓶颈。Profiling 技术是一种在应用运行时收集程序相关信息的动态分析手段，常用的 JVM Profiler 可以从多个方面对程序进行动态分析，如 CPU、Memory、Thread、Classes、GC 等，其中 CPU Profiling 的应用最为广泛。CPU Profiling 经常被用于分析代码的执行热点，如“哪个方法占用 CPU 的执行时间最长”、“每个方法占用 CPU 的比例是多少”等等，通过 CPU Profiling 得到上述相关信息后，研发人员就可以轻松针对热点瓶颈进行分析和性能优化，进而突破性能瓶颈，大幅提升系统的吞吐量。

本文介绍了 JVM 平台上 CPU Profiler 的实现原理，希望能帮助读者在使用类似工具的同时也能清楚其内部的技术实现。

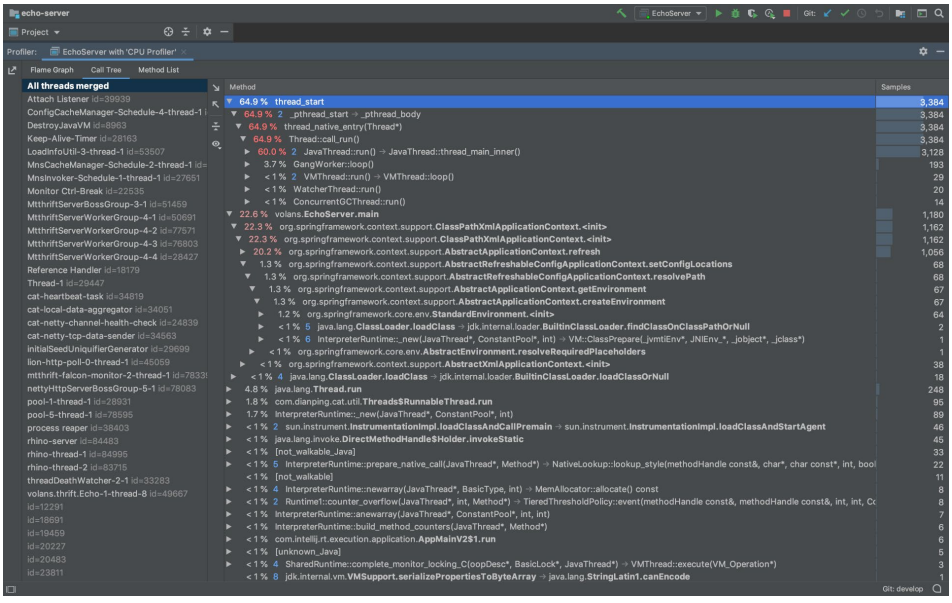
CPU Profiler 简介

社区实现的 JVM Profiler 很多，比如已经商用且功能强大的 JProfiler，也有免费开源的产品，如 JVM-Profiler，功能各有所长。我们日常使用的 Intellij IDEA 最新版内部也集成了一个简单好用的 Profiler，详细的介绍参见官方 Blog。

在用 IDEA 打开需要诊断的 Java 项目后，在“Preferences -> Build, Execution, Deployment -> Java Profiler”界面添加一个“CPU Profiler”，然后回到项目，单击右上角的“Run with Profiler”启动项目并开始 CPU Profiling 过程。一定时间后（推荐 5min），在 Profiler 界面点击“Stop Profiling and Show Results”，即可看到 Profiling 的结果，包含火焰图和调用树，如下图所示：



Intellij IDEA – 性能火焰图



Intellij IDEA – 调用堆栈树

火焰图是根据调用栈的样本集生成的可视化性能分析图，《如何读懂火焰图?》一

文对火焰图进行了不错的讲解，大家可以参考一下。简而言之，看火焰图时我们需要关注“平顶”，因为那里就是我们程序的 CPU 热点。调用树是另一种可视化分析的手段，与火焰图一样，也是根据同一份样本集而生成，按需选择即可。

这里要说明一下，因为我们没有在项目中引入任何依赖，仅仅是“Run with Profiler”，Profiler 就能获取我们程序运行时的信息。这个功能其实是通过 JVM Agent 实现的，为了更好地帮助大家系统性的了解它，我们在这里先对 JVM Agent 做个简单的介绍。

JVM Agent 简介

JVM Agent 是一个按一定规则编写的特殊程序库，可以在启动阶段通过命令行参数传递给 JVM，作为一个伴生库与目标 JVM 运行在同一个进程中。在 Agent 中可以通过固定的接口获取 JVM 进程内的相关信息。Agent 既可以是用 C/C++/Rust 编写的 JVMTI Agent，也可以是用 Java 编写的 Java Agent。

执行 Java 命令，我们可以看到 Agent 相关的命令行参数：

```
Plain Text
    -agentlib:< 库名 >[=< 选项 >]
                  加载本机代理库 < 库名 >, 例如 -agentlib:jdwp
                  另请参阅 -agentlib:jdwp=help
    -agentpath:< 路径名 >[=< 选项 >]
                  按完整路径名加载本机代理库
    -javaagent:<jar 路径 >[=< 选项 >]
                  加载 Java 编程语言代理 , 请参阅 java.lang.instrument
```

JVMTI Agent

JVMTI（JVM Tool Interface）是 JVM 提供的一套标准的 C/C++ 编程接口，是实现 Debugger、Profiler、Monitor、Thread Analyser 等工具的统一基础，在主流 Java 虚拟机中都有实现。

当我们要基于 JVMTI 实现一个 Agent 时，需要实现如下入口函数：

```
// $JAVA_HOME/include/jvmti.h

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved);
```

使用 C/C++ 实现该函数，并将代码编译为动态连接库（Linux 上是 .so），通过 -agentpath 参数将库的完整路径传递给 Java 进程，JVM 就会在启动阶段的合适时机执行该函数。在函数内部，我们可以通过 JavaVM 指针参数拿到 JNI 和 JVMTI 的函数指针表，这样我们就拥有了与 JVM 进行各种复杂交互的能力。

更多 JVMTI 相关的细节可以参考官方文档。

Java Agent

在很多场景下，我们没有必要必须使用 C/C++ 来开发 JVMTI Agent，因为成本高且不易维护。JVM 自身基于 JVMTI 封装了一套 Java 的 Instrument API 接口，允许使用 Java 语言开发 Java Agent（只是一个 jar 包），大大降低了 Agent 的开发成本。社区开源的产品如 Greys、Arthas、JVM-Sandbox、JVM-Profiler 等都是纯 Java 编写的，也是以 Java Agent 形式来运行。

在 Java Agent 中，我们需要在 jar 包的 MANIFEST.MF 中将 Premain-Class 指定为一个入口类，并在该入口类中实现如下方法：

```
public static void premain(String args, Instrumentation ins) {
    // implement
}
```

这样打包出来的 jar 就是一个 Java Agent，可以通过 -javaagent 参数将 jar 传递给 Java 进程伴随启动，JVM 同样会在启动阶段的合适时机执行该方法。

在该方法内部，参数 Instrumentation 接口提供了 Retransform Classes 的能力，我们利用该接口就可以对宿主进程的 Class 进行修改，实现方法耗时统计、故障注入、Trace 等功能。Instrumentation 接口提供的能力较为单一，仅与 Class 字节码操作相关，但由于我们现在已经处于宿主进程环境内，就可以利用 JMX 直接获取宿主进程的内存、线程、锁等信息。无论是 Instrument API 还是 JMX，它们内部仍是统一基于 JVMTI 来实现。

更多 Instrument API 相关的细节可以参考官方文档。

CPU Profiler 原理解析

在了解完 Profiler 如何以 Agent 的形式执行后，我们可以开始尝试构造一个简单的 CPU Profiler。但在此之前，还有必要了解下 CPU Profiling 技术的两种实现方式及其区别。

Sampling vs Instrumentation

使用过 JProfiler 的同学应该都知道，JProfiler 的 CPU Profiling 功能提供了两种方式选项：Sampling 和 Instrumentation，它们也是实现 CPU Profiler 的两种手段。

Sampling 方式顾名思义，基于对 StackTrace 的“采样”进行实现，核心原理如下：

1. 引入 Profiler 依赖，或直接利用 Agent 技术注入目标 JVM 进程并启动 Profiler。
2. 启动一个采样定时器，以固定的采样频率每隔一段时间（毫秒级）对所有线程的调用栈进行 Dump。
3. 汇总并统计每次调用栈的 Dump 结果，在一定时间内采到足够的样本后，导出统计结果，内容是每个方法被采样到的次数及方法的调用关系。

Instrumentation 则是利用 Instrument API，对所有必要的 Class 进行字节码增强，在进入每个方法前进行埋点，方法执行结束后统计本次方法执行耗时，最终进行汇总。二者都能得到想要的结果，那么它们有什么区别呢？或者说，孰优孰劣？

Instrumentation 方式对几乎所有方法添加了额外的 AOP 逻辑，这会导致对线上服务造成巨额的性能影响，但其优势是：绝对精准的方法调用次数、调用时间统计。

Sampling 方式基于无侵入的额外线程对所有线程的调用栈快照进行固定频率抽样，相对前者来说它的性能开销很低。但由于它基于“采样”的模式，以及 JVM 固有的只能在安全点（Safe Point）进行采样的“缺陷”，会导致统计结果存在一定的

偏差。譬如说：某些方法执行时间极短，但执行频率很高，真实占用了大量的 CPU Time，但 Sampling Profiler 的采样周期不能无限调小，这会导致性能开销骤增，所以会导致大量的样本调用栈中并不存在刚才提到的”高频小方法“，进而导致最终结果无法反映真实的 CPU 热点。更多 Sampling 相关的问题可以参考《Why (Most) Sampling Java Profilers Are Fucking Terrible》。

具体到“孰优孰劣”的问题层面，这两种实现技术并没有非常明显的高下之判，只有在分场景讨论下才有意义。Sampling 由于低开销的特性，更适合用在 CPU 密集型的应用中，以及不可接受大量性能开销的线上服务中。而 Instrumentation 则更适合用在 I/O 密集的应用中、对性能开销不敏感以及确实需要精确统计的场景中。社区的 Profiler 更多的是基于 Sampling 来实现，本文也是基于 Sampling 来进行讲解。

基于 Java Agent + JMX 实现

一个最简单的 Sampling CPU Profiler 可以用 Java Agent + JMX 方式来实现。以 Java Agent 为入口，进入目标 JVM 进程后开启一个 ScheduledExecutorService，定时利用 JMX 的 threadMXBean.dumpAllThreads() 来导出所有线程的 StackTrace，最终汇总并导出即可。

Uber 的 JVM-Profiler 实现原理也是如此，关键部分代码如下：

```
// com/uber/profiling/profilers/StacktraceCollectorProfiler.java

/*
 * StacktraceCollectorProfiler 等同于文中所述 CpuProfiler，仅命名偏好不同而已
 * jvm-profiler 的 CpuProfiler 指代的是 CpuLoad 指标的 Profiler
 */

// 实现了 Profiler 接口，外部由统一的 ScheduledExecutorService 对所有 Profiler 定时执行
@Override
public void profile() {
    ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false, false);
    // ...
    for (ThreadInfo threadInfo : threadInfos) {
        String threadName = threadInfo.getThreadName();
        // ...
```

```
        StackTraceElement[] stackTraceElements = threadInfo.
getStackTrace();
        // ...
        for (int i = stackTraceElements.length - 1; i >= 0; i--) {
            StackTraceElement stackTraceElement = stackTraceElements[i];
            // ...
        }
        // ...
    }
}
```

Uber 提供的定时器默认 Interval 是 100ms，对于 CPU Profiler 来说，这略显粗糙。但由于 dumpAllThreads() 的执行开销不容小觑，Interval 不宜设置的过小，所以该方法的 CPU Profiling 结果会存在不小的误差。

JVM-Profiler 的优点在于支持多种指标的 Profiling（StackTrace、CPUBusy、Memory、I/O、Method），且支持将 Profiling 结果通过 Kafka 上报回中心 Server 进行分析，也即支持集群诊断。

基于 JVMTI + GetStackTrace 实现

使用 Java 实现 Profiler 相对较简单，但也存在一些问题，譬如说 Java Agent 代码与业务代码共享 AppClassLoader，被 JVM 直接加载的 agent.jar 如果引入了第三方依赖，可能会对业务 Class 造成污染。截止发稿时，JVM-Profiler 都存在这个问题，它引入了 Kafka-Client、http-Client、Jackson 等组件，如果与业务代码中的组件版本发生冲突，可能会引发未知错误。Greys/Arthas/JVM-Sandbox 的解决方式是分离入口与核心代码，使用定制的 ClassLoader 加载核心代码，避免影响业务代码。

在更底层的 C/C++ 层面，我们可以直接对接 JVMTI 接口，使用原生 C API 对 JVM 进行操作，功能更丰富更强大，但开发效率偏低。基于上节同样的原理开发 CPU Profiler，使用 JVMTI 需要进行如下这些步骤：

1. 编写 Agent_OnLoad()，在入口通过 JNI 的 JavaVM* 指针的 GetEnv() 函数拿到 JVMTI 的 jvmtiEnv 指针：

```
// agent.c

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void
*reserved) {
    jvmtiEnv *jvmti;
    (*vm)->GetEnv((void **)&jvmti, JVMTI_VERSION_1_0);
    // ...
    return JNI_OK;
}
```

2. 开启一个线程定时循环，定时使用 jvmtiEnv 指针配合调用如下几个 JVMTI 函数：

```
// 获取所有线程的 jthread
jvmtiError GetAllThreads(jvmtiEnv *env, jint *threads_count_ptr,
jthread **threads_ptr);

// 根据 jthread 获取该线程信息（name、daemon、priority...）
jvmtiError GetThreadInfo(jvmtiEnv *env, jthread thread,
jvmtiThreadInfo* info_ptr);

// 根据 jthread 获取该线程调用栈
jvmtiError GetStackTrace(jvmtiEnv *env,
                         jthread thread,
                         jint start_depth,
                         jint max_frame_count,
                         jvmtiFrameInfo *frame_buffer,
                         jint *count_ptr);
```

主逻辑大致是：首先调用 GetAllThreads() 获取所有线程的“句柄”jthread，然后遍历根据 jthread 调用 GetThreadInfo() 获取线程信息，按线程名过滤掉不需要的线程后，继续遍历根据 jthread 调用 GetStackTrace() 获取线程的调用栈。

3. 在 Buffer 中保存每一次的采样结果，最终生成必要的统计数据即可。

按如上步骤即可实现基于 JVMTI 的 CPU Profiler。但需要说明的是，即便是基于原生 JVMTI 接口使用 GetStackTrace() 的方式获取调用栈，也存在与 JMX 相同的问题——只能在安全点（Safe Point）进行采样。

SafePoint Bias 问题

基于 Sampling 的 CPU Profiler 通过采集程序在不同时间点的调用栈样本来近似地推算出热点方法，因此，从理论上来讲 Sampling CPU Profiler 必须遵循以下两个原则：

1. 样本必须足够多。
2. 程序中所有正在运行的代码点都必须以相同的概率被 Profiler 采样。

如果只能在安全点采样，就违背了第二条原则。因为我们只能采集到位于安全点时刻的调用栈快照，意味着某些代码可能永远没有机会被采样，即使它真实耗费了大量的 CPU 执行时间，这种现象被称为“SafePoint Bias”。

上文我们提到，基于 JMX 与基于 JVMTI 的 Profiler 实现都存在 SafePoint Bias，但一个值得了解的细节是：单独来说，JVMTI 的 GetStackTrace() 函数并不需要在 Caller 的安全点执行，但当调用 GetStackTrace() 获取其他线程的调用栈时，必须等待，直到目标线程进入安全点；而且，GetStackTrace() 仅能通过单独的线程同步定时调用，不能在 UNIX 信号处理器的 Handler 中被异步调用。综合来说，GetStackTrace() 存在与 JMX 一样的 SafePoint Bias。更多安全点相关的知识可以参考《Safepoints: Meaning, Side Effects and Overheads》。

那么，如何避免 SafePoint Bias？社区提供了一种 Hack 思路——AsyncGetCallTrace。

基于 JVMTI + AsyncGetCallTrace 实现

如上节所述，假如我们拥有一个函数可以获取当前线程的调用栈且不受安全点干扰，另外它还支持在 UNIX 信号处理器中被异步调用，那么我们只需注册一个 UNIX 信号处理器，在 Handler 中调用该函数获取当前线程的调用栈即可。由于 UNIX 信号会被发送给进程的随机一线程进行处理，因此最终信号会均匀分布在所有线程上，也就均匀获取了所有线程的调用栈样本。

OracleJDK/OpenJDK 内部提供了这么一个函数——AsyncGetCallTrace，它

的原型如下：

```
// 栈帧
typedef struct {
 jint lineno;
 jmethodID method_id;
} AGCT_CallFrame;

// 调用栈
typedef struct {
    JNIEnv *env;
    jint num_frames;
    AGCT_CallFrame *frames;
} AGCT_CallTrace;

// 根据 ucontext 将调用栈填充进 trace 指针
void AsyncGetCallTrace(AGCT_CallTrace *trace, jint depth, void
*ucontext);
```

通过原型可以看到，该函数的使用方式非常简洁，直接通过 ucontext 就能获取到完整的 Java 调用栈。

顾名思义，AsyncGetCallTrace 是“async”的，不受安全点影响，这样的话采样就可能发生在任何时间，包括 Native 代码执行期间、GC 期间等，在这时我们是无法获取 Java 调用栈的，AGCT_CallTrace 的 num_frames 字段正常情况下标识了获取到的调用栈深度，但在如前所述的异常情况下它就表示为负数，最常见的 −2 代表此刻正在 GC。

由于 AsyncGetCallTrace 非标准 JVMTI 函数，因此我们无法在 jvmti.h 中找到该函数声明，且由于其目标文件也早已链接进 JVM 二进制文件中，所以无法通过简单的声明来获取该函数的地址，这需要通过一些 Trick 方式来解决。简单说，Agent 最终是作为动态链接库加载到目标 JVM 进程的地址空间中，因此可以在 Agent_OnLoad 内通过 glibc 提供的 dlsym() 函数拿到当前地址空间（即目标 JVM 进程地址空间）名为“AsyncGetCallTrace”的符号地址。这样就拿到了该函数的指针，按照上述原型进行类型转换后，就可以正常调用了。

通过 AsyncGetCallTrace 实现 CPU Profiler 的大致流程：

1. 编写 Agent_OnLoad()，在入口拿到 jvmtiEnv 和 AsyncGetCallTrace 指针，获取 AsyncGetCallTrace 方式如下：

```
typedef void (*AsyncGetCallTrace)(AGCT_CallTrace *traces, jint depth,
void *ucontext);
// ...
AsyncGetCallTrace agct_ptr = (AsyncGetCallTrace)dlsym(RTLD_DEFAULT,
"AsyncGetCallTrace");
if (agct_ptr == NULL) {
    void *libjvm = dlopen("libjvm.so", RTLD_NOW);
    if (!libjvm) {
        // 处理 dlerror()...
    }
    agct_ptr = (AsyncGetCallTrace)dlsym(libjvm, "AsyncGetCallTrace");
}
```

2. 在 OnLoad 阶段，我们还需要做一件事，即注册 OnClassLoad 和 OnClassPrepare 这两个 Hook，原因是 jmethodID 是延迟分配的，使用 AGCT 获取 Traces 依赖预先分配好的数据。我们在 OnClassPrepare 的 CallBack 中尝试获取该 Class 的所有 Methods，这样就使 JVMTI 提前分配了所有方法的 jmethodID，如下所示：

```
void JNICALL OnClassLoad(jvmtiEnv *jvmti, JNIEnv* jni, jthread thread,
jclass klass) {}

void JNICALL OnClassPrepare(jvmtiEnv *jvmti, JNIEnv *jni, jthread
thread, jclass klass) {
    jint method_count;
    jmethodID *methods;
    jvmti->GetClassMethods(klass, &method_count, &methods);
    delete [] methods;
}

// ...

jvmtiEventCallbacks callbacks = {0};
callbacks.ClassLoad = OnClassLoad;
callbacks.ClassPrepare = OnClassPrepare;
jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_CLASS_LOAD,
NULL);
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_CLASS_PREPARE, NULL);
```

3. 利用 SIGPROF 信号来进行定时采样：

```
// 这里信号 handler 传进来的 ucontext 即 AsyncGetCallTrace 需要的 ucontext
void signal_handler(int signo, siginfo_t *siginfo, void *ucontext) {
    // 使用 AsyncCallTrace 进行采样，注意处理 num_frames 为负的异常情况
}

// ...

// 注册 SIGPROF 信号的 handler
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = signal_handler;
sa.sa_flags = SA_RESTART | SA_SIGINFO;
sigaction(SIGPROF, &sa, NULL);

// 定时产生 SIGPROF 信号
// interval 是 nanoseconds 表示的采样间隔，AsyncGetCallTrace 相对于同步采样来说可以适当高频一些
long sec = interval / 1000000000;
long usec = (interval % 1000000000) / 1000;
struct itimerval tv = {{sec, usec}, {sec, usec}};
setitimer(ITIMER_PROF, &tv, NULL);
```

4. 在 Buffer 中保存每一次的采样结果，最终生成必要的统计数据即可。

按如上步骤即可实现基于 AsyncGetCallTrace 的 CPU Profiler，这是社区中目前性能开销最低、相对效率最高的 CPU Profiler 实现方式，在 Linux 环境下结合 perf_events 还能做到同时采样 Java 栈与 Native 栈，也就能同时分析 Native 代码中存在的性能热点。该方式的典型开源实现有 Async-Profiler 和 Honest-Profiler，Async-Profiler 实现质量较高，感兴趣的话建议大家阅读参考文章。有趣的是，IntelliJ IDEA 内置的 Java Profiler，其实就是 Async-Profiler 的包装。更多关于 AsyncGetCallTrace 的内容，大家可以参考《The Pros and Cons of AsyncGet-CallTrace Profilers》。

生成性能火焰图

现在我们拥有了采样调用栈的能力，但是调用栈样本集是以二维数组的数据结构形式存在于内存中的，如何将其转换为可视化的火焰图呢?

火焰图通常是一个 svg 文件，部分优秀项目可以根据文本文件自动生成火焰图文

件，仅对文本文件的格式有一定要求。FlameGraph 项目的核心只是一个 Perl 脚本，可以根据我们提供的调用栈文本生成相应的火焰图 svg 文件。调用栈的文本格式相当简单，如下所示：

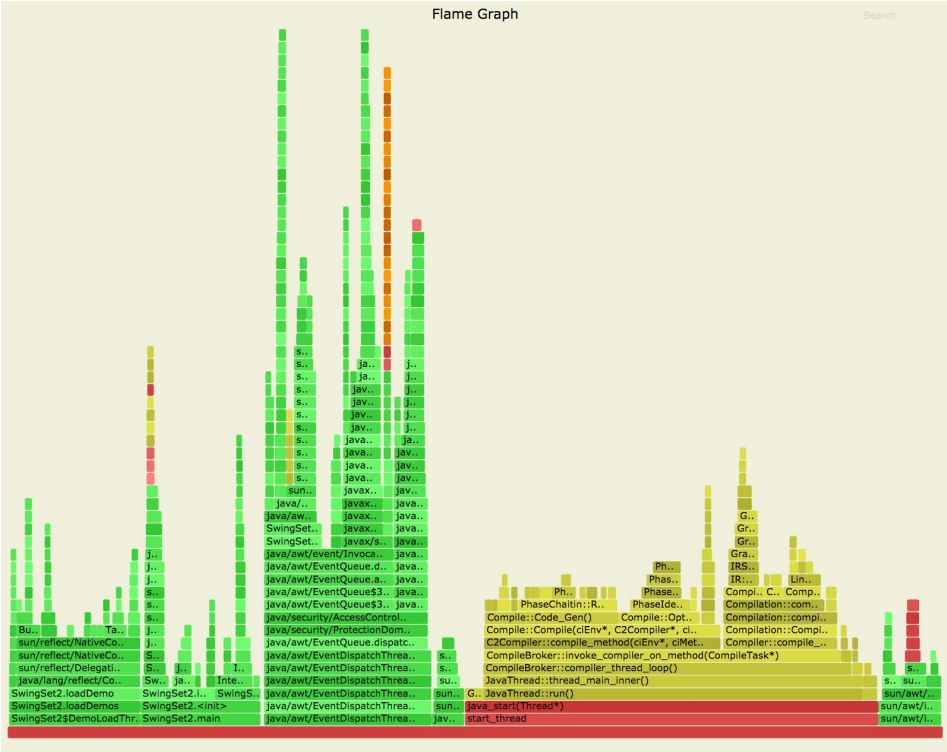
```
base_func;func1;func2;func3 10
base_func;funca;funcb 15
```

将我们采样到的调用栈样本集进行整合后，需输出如上所示的文本格式。每一行代表一“类“调用栈，空格左边是调用栈的方法名排列，以分号分割，左栈底右栈顶，空格右边是该样本出现的次数。

将样本文件交给 flamegraph.pl 脚本执行，就能输出相应的火焰图了：

```
$ flamegraph.pl stacktraces.txt > stacktraces.svg
```

效果如下图所示：



通过 flamegraph.pl 生成的火焰图

HotSpot 的 Dynamic Attach 机制解析

到目前为止，我们已经了解了 CPU Profiler 完整的工作原理，然而使用过 JProfiler/Arthas 的同学可能会有疑问，很多情况下可以直接对线上运行中的服务进行 Profling，并不需要在 Java 进程的启动参数添加 Agent 参数，这是通过什么手段做到的？答案是 Dynamic Attach。

JDK 在 1.6 以后提供了 Attach API，允许向运行中的 JVM 进程添加 Agent，这项手段被广泛使用在各种 Profiler 和字节码增强工具中，其官方简介如下：

> This is a Sun extension that allows a tool to 'attach' to another process running Java code and launch a JVM TI agent or a java.lang.instrument agent in that process.

总的来说，Dynamic Attach 是 HotSpot 提供的一种特殊能力，它允许一个进程向另一个运行中的 JVM 进程发送一些命令并执行，命令并不限于加载 Agent，还包括 Dump 内存、Dump 线程等等。

通过 sun.tools 进行 Attach

Attach 虽然是 HotSpot 提供的能力，但 JDK 在 Java 层面也对其做了封装。

前文已经提到，对于 Java Agent 来说，PreMain 方法在 Agent 作为启动参数运行的时候执行，其实我们还可以额外实现一个 AgentMain 方法，并在 MANIFEST.MF 中将 Agent-Class 指定为该 Class：

```
public static void agentmain(String args, Instrumentation ins) {
    // implement
}
```

这样打包出来的 jar，既可以作为 -javaagent 参数启动，也可以被 Attach 到运行中的目标 JVM 进程。JDK 已经封装了简单的 API 让我们直接 Attach 一个 Java Agent，下面以 Arthas 中的代码进行演示：

```
// com/taobao/arthas/core/Arthas.java
```

```
import com.sun.tools.attach.VirtualMachine;
import com.sun.tools.attach.VirtualMachineDescriptor;

// ...

private void attachAgent(Configure configure) throws Exception {
    VirtualMachineDescriptor virtualMachineDescriptor = null;

    // 拿到所有 JVM 进程，找出目标进程
    for (VirtualMachineDescriptor descriptor : VirtualMachine.list()) {
        String pid = descriptor.id();
        if (pid.equals(Integer.toString(configure.getJavaPid()))) {
            virtualMachineDescriptor = descriptor;
        }
    }
    VirtualMachine virtualMachine = null;
    try {
        // 针对某个 JVM 进程调用 VirtualMachine.attach() 方法，拿到
VirtualMachine 实例
        if (null == virtualMachineDescriptor) {
            virtualMachine = VirtualMachine.attach("" + configure.
getJavaPid());
        } else {
            virtualMachine = VirtualMachine.
attach(virtualMachineDescriptor);
        }

        // ...

        // 调用 VirtualMachine#loadAgent()，将 arthasAgentPath 指定的 jar
attach 到目标 JVM
进程中
        // 第二个参数为 attach 参数，即 agentmain 的首个 String 参数 args
        virtualMachine.loadAgent(arthasAgentPath, configure.
getArthasCore() + ";" + configure.
toString());
    } finally {
        if (null != virtualMachine) {
            // 调用 VirtualMachine#detach() 释放
            virtualMachine.detach();
        }
    }
}
```

直接对 HotSpot 进行 Attach

sun.tools 封装的 API 足够简单易用，但只能使用 Java 编写，也只能用在 Java Agent 上，因此有些时候我们必须手工对 JVM 进程直接进行 Attach。对于 JVMTI，除了 Agent_OnLoad() 之外，我们还需实现一个 Agent_OnAttach() 函数，当将 JVMTI Agent Attach 到目标进程时，从该函数开始执行：

```
// $JAVA_HOME/include/jvmti.h

JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM *vm, char *options, void
*reserved);
```

下面我们以 Async-Profiler 中的 jattach 源码为线索，探究一下如何利用 Attach 机制给运行中的 JVM 进程发送命令。jattach 是 Async-Profiler 提供的一个 Driver，使用方式比较直观：

```
Usage:
    jattach <pid> <cmd> [args ...]
Args:
    <pid>   目标 JVM 进程的进程 ID
    <cmd>   要执行的命令
    <args>  命令参数
```

使用方式如：

```
$ jattach 1234 load /absolute/path/to/agent/libagent.so true
```

执行上述命令，libagent.so 就被加载到 ID 为 1234 的 JVM 进程中并开始执行 Agent_OnAttach 函数了。有一点需要注意，执行 Attach 的进程 euid 及 egid，与被 Attach 的目标 JVM 进程必须相同。接下来开始分析 jattach 源码。

如下所示的 Main 函数描述了一次 Attach 的整体流程：

```
// async-profiler/src/jattach/jattach.c

int main(int argc, char** argv) {
    // 解析命令行参数
    // 检查 euid 与 egid
```

```
    // ...

    if (!check_socket(nspid) && !start_attach_mechanism(pid, nspid)) {
        perror("Could not start attach mechanism");
        return 1;
    }

    int fd = connect_socket(nspid);
    if (fd == -1) {
        perror("Could not connect to socket");
        return 1;
    }

    printf("Connected to remote JVM\n");
    if (!write_command(fd, argc - 2, argv + 2)) {
        perror("Error writing to socket");
        close(fd);
        return 1;
    }
    printf("Response code = ");
    fflush(stdout);

    int result = read_response(fd);
    close(fd);
    return result;
}
```

忽略掉命令行参数解析与检查 euid 和 egid 的过程。jattach 首先调用了 check_socket 函数进行了“socket 检查?”，check_socket 源码如下：

```
// async-profiler/src/jattach/jattach.c

// Check if remote JVM has already opened socket for Dynamic Attach
static int check_socket(int pid) {
    char path[MAX_PATH];
    snprintf(path, MAX_PATH, "%s/.java_pid%d", get_temp_directory(),
pid); // get_temp_
directory() 在 Linux 下固定返回 "/tmp"
    struct stat stats;
    return stat(path, &stats) == 0 && S_ISSOCK(stats.st_mode);
}
```

我们知道，UNIX 操作系统提供了一种基于文件的 Socket 接口，称为“UNIX Socket”（一种常用的进程间通信方式）。在该函数中使用 S_ISSOCK 宏来判断该文

件是否被绑定到了 UNIX Socket，如此看来，“/tmp/.java_pid”文件很有可能就是外部进程与 JVM 进程间通信的桥梁。

查阅官方文档，得到如下描述：

> The attach listener thread then communicates with the source JVM in an OS dependent manner: – On Solaris, the Doors IPC mechanism is used. The door is attached to a file in the file system so that clients can access it. – On Linux, a Unix domain socket is used. This socket is bound to a file in the filesystem so that clients can access it. – On Windows, the created thread is given the name of a pipe which is served by the client. The result of the operations are written to this pipe by the target JVM.

证明了我们的猜想是正确的。目前为止 check_socket 函数的作用很容易理解了：判断外部进程与目标 JVM 进程之间是否已经建立了 UNIX Socket 连接。

回到 Main 函数，在使用 check_socket 确定连接尚未建立后，紧接着调用 start_attach_mechanism 函数，函数名很直观地描述了它的作用，源码如下：

```
// async-profiler/src/jattach/jattach.c

// Force remote JVM to start Attach listener.
// HotSpot will start Attach listener in response to SIGQUIT if it
sees .attach_pid file
static int start_attach_mechanism(int pid, int nspid) {
    char path[MAX_PATH];
    snprintf(path, MAX_PATH, "/proc/%d/cwd/.attach_pid%d", nspid,
nspid);

    int fd = creat(path, 0660);
    if (fd == -1 || (close(fd) == 0 && !check_file_owner(path))) {
        // Failed to create attach trigger in current directory. Retry
in /tmp
        snprintf(path, MAX_PATH, "%s/.attach_pid%d", get_temp_
directory(), nspid);
        fd = creat(path, 0660);
        if (fd == -1) {
            return 0;
        }
        close(fd);
```

```
    }

    // We have to still use the host namespace pid here for the kill() call
    kill(pid, SIGQUIT);

    // Start with 20 ms sleep and increment delay each iteration
    struct timespec ts = {0, 20000000};
    int result;
    do {
        nanosleep(&ts, NULL);
        result = check_socket(nspid);
    } while (!result && (ts.tv_nsec += 20000000) < 300000000);

    unlink(path);
    return result;
}
```

start_attach_mechanism 函数首先创建了一个名为“/tmp/.attach_pid”的空文件，然后向目标 JVM 进程发送了一个 SIGQUIT 信号，这个信号似乎触发了 JVM 的某种机制？紧接着，start_attach_mechanism 函数开始陷入了一种等待，每 20ms 调用一次 check_socket 函数检查连接是否被建立，如果等了 300ms 还没有成功就放弃。函数的最后调用 Unlink 删掉 .attach_pid 文件并返回。

如此看来，HotSpot 似乎提供了一种特殊的机制，只要给它发送一个 SIGQUIT 信号，并预先准备好 .attach_pid 文件，HotSpot 会主动创建一个地址为“/tmp/.java_pid”的 UNIX Socket，接下来主动 Connect 这个地址即可建立连接执行命令。

查阅文档，得到如下描述：

> Dynamic attach has an attach listener thread in the target JVM. This is a thread that is started when the first attach request occurs. On Linux and Solaris, the client creates a file named .attach_pid(pid) and sends a SIGQUIT to the target JVM process. The existence of this file causes the SIGQUIT handler in HotSpot to start the attach listener thread. On Windows, the client uses the Win32 CreateRemoteThread function to create a new thread in the target process.

这样一来就很明确了，在 Linux 上我们只需创建一个“/tmp/.attach_pid”文

件，并向目标 JVM 进程发送一个 SIGQUIT 信号，HotSpot 就会开始监听“/tmp/.java_pid”地址上的 UNIX Socket，接收并执行相关 Attach 的命令。至于为什么一定要创建 .attach_pid 文件才可以触发 Attach Listener 的创建，经查阅资料，我们得到了两种说法：一是 JVM 不止接收从外部 Attach 进程发送的 SIGQUIT 信号，必须配合外部进程创建的外部文件才能确定这是一次 Attach 请求；二是为了安全。

继续看 jattach 的源码，果不其然，它调用了 connect_socket 函数对“/tmp/.java_pid”进行连接，connect_socket 源码如下：

```
// async-profiler/src/jattach/jattach.c

// Connect to UNIX domain socket created by JVM for Dynamic Attach
static int connect_socket(int pid) {
    int fd = socket(PF_UNIX, SOCK_STREAM, 0);
    if (fd == -1) {
        return -1;
    }

    struct sockaddr_un addr;
    addr.sun_family = AF_UNIX;
    snprintf(addr.sun_path, sizeof(addr.sun_path), "%s/.java_pid%d",
get_temp_directory(), pid);

    if (connect(fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        close(fd);
        return -1;
    }
    return fd;
}
```

一个很普通的 Socket 创建函数，返回 Socket 文件描述符。

回到 Main 函数，主流程紧接着调用 write_command 函数向该 Socket 写入了从命令行传进来的参数，并且调用 read_response 函数接收从目标 JVM 进程返回的数据。两个很常见的 Socket 读写函数，源码如下：

```
// async-profiler/src/jattach/jattach.c

// Send command with arguments to socket
static int write_command(int fd, int argc, char** argv) {
    // Protocol version
```

```
    if (write(fd, "1", 2) <= 0) {
        return 0;
    }

    int i;
    for (i = 0; i < 4; i++) {
        const char* arg = i < argc ? argv[i] : "";
        if (write(fd, arg, strlen(arg) + 1) <= 0) {
            return 0;
        }
    }
    return 1;
}

// Mirror response from remote JVM to stdout
static int read_response(int fd) {
    char buf[8192];
    ssize_t bytes = read(fd, buf, sizeof(buf) - 1);
    if (bytes <= 0) {
        perror("Error reading response");
        return 1;
    }

    // First line of response is the command result code
    buf[bytes] = 0;
    int result = atoi(buf);

    do {
        fwrite(buf, 1, bytes, stdout);
        bytes = read(fd, buf, sizeof(buf));
    } while (bytes > 0);
    return result;
}
```

浏览 write_command 函数就可知外部进程与目标 JVM 进程之间发送的数据格式相当简单，基本如下所示：

```
<PROTOCOL VERSION>\0<COMMAND>\0<ARG1>\0<ARG2>\0<ARG3>\0
```

以先前我们使用的 Load 命令为例，发送给 HotSpot 时格式如下：

```
1\0load\0/absolute/path/to/agent/libagent.so\0true\0\0
```

至此，我们已经了解了如何手工对 JVM 进程直接进行 Attach。

Attach 补充介绍

Load 命令仅仅是 HotSpot 所支持的诸多命令中的一种，用于动态加载基于 JVMTI 的 Agent，完整的命令表如下所示：

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             JvmtiExport::load_agent_library },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

读者可以尝试下 threaddump 命令，然后对相同的进程进行 jstack，对比观察输出，其实是完全相同的，其它命令大家可以自行进行探索。

总结

总的来说，善用各类 Profiler 是提升性能优化效率的一把利器，了解 Profiler 本身的实现原理更能帮助我们避免对工具的各种误用。CPU Profiler 所依赖的 Attach、JVMTI、Instrumentation、JMX 等皆是 JVM 平台比较通用的技术，在此基础上，我们去实现 Memory Profiler、Thread Profiler、GC Analyzer 等工具也没有想象中那么神秘和复杂了。

参考资料

- JVM Tool Interface
- The Pros and Cons of AsyncGetCallTrace Profilers
- Why (Most) Sampling Java Profilers Are Fucking Terrible
- Safepoints: Meaning, Side Effects and Overheads
- Serviceability in HotSpot
- 如何读懂火焰图?
- IntelliJ IDEA 2018.3 EAP: Git Submodules, JVM Profiler (macOS and Linux) and more

作者简介

业祥，继东，美团基础架构部 / 服务框架组工程师。

团队信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历至：tech@meituan.com。

Java 动态调试技术原理及实践

胡健

断点调试是我们最常使用的调试手段，它可以获取到方法执行过程中的变量信息，并可以观察到方法的执行路径。但断点调试会在断点位置停顿，使得整个应用停止响应。在线上停顿应用是致命的，动态调试技术给了我们创造新的调试模式的想象空间。本文将研究 Java 语言中的动态调试技术，首先概括 Java 动态调试所涉及的技术基础，接着介绍我们在 Java 动态调试领域的思考及实践，通过结合实际业务场景，设计并实现了一种具备动态性的断点调试工具 Java-debug-tool，显著提高了故障排查效率。

JVMTI（JVM Tool Interface）是 Java 虚拟机对外提供的 Native 编程接口，通过 JVMTI，外部进程可以获取到运行时 JVM 的诸多信息，比如线程、GC 等。Agent 是一个运行在目标 JVM 的特定程序，它的职责是负责从目标 JVM 中获取数据，然后将数据传递给外部进程。加载 Agent 的时机可以是目标 JVM 启动之时，也可以是在目标 JVM 运行时进行加载，而在目标 JVM 运行时进行 Agent 加载具备动态性，对于时机未知的 Debug 场景来说非常实用。下面将详细分析 Java Agent 技术的实现细节。

2.1 Agent 的实现模式

JVMTI 是一套 Native 接口，在 Java SE 5 之前，要实现一个 Agent 只能通过编写 Native 代码来实现。从 Java SE 5 开始，可以使用 Java 的 Instrumentation 接口（java.lang.instrument）来编写 Agent。无论是通过 Native 的方式还是通过 Java Instrumentation 接口的方式来编写 Agent，它们的工作都是借助 JVMTI 来进行完成，下面介绍通过 Java Instrumentation 接口编写 Agent 的方法。

2.1.1 通过 Java Instrumentation API

- 实现 Agent 启动方法

Java Agent 支持目标 JVM 启动时加载，也支持在目标 JVM 运行时加载，这两种不同的加载模式会使用不同的入口函数，如果需要在目标 JVM 启动的同时加载 Agent，那么可以选择实现下面的方法：

```
[1] public static void premain(String agentArgs, Instrumentation inst);
[2] public static void premain(String agentArgs);
```

JVM 将首先寻找 [1]，如果没有发现 [1]，再寻找 [2]。如果希望在目标 JVM 运行时加载 Agent，则需要实现下面的方法：

```
[1] public static void agentmain(String agentArgs, Instrumentation inst);
[2] public static void agentmain(String agentArgs);
```

这两组方法的第一个参数 AgentArgs 是随同“- javaagent”一起传入的程序参数，如果这个字符串代表了多个参数，就需要自己解析这些参数。inst 是 Instrumentation 类型的对象，是 JVM 自动传入的，我们可以拿这个参数进行类增强等操作。

- 指定 Main-Class

Agent 需要打包成一个 jar 包，在 ManiFest 属性中指定“Premain-Class”或者“Agent-Class”：

```
Premain-Class: class
Agent-Class: class
```

- 挂载到目标 JVM

将编写的 Agent 打成 jar 包后，就可以挂载到目标 JVM 上去了。如果选择在目标 JVM 启动时加载 Agent，则可以使用“-javaagent:[=]“，具体的使用方法可以使用“Java -Help”来查看。如果想要在运行时挂载 Agent 到目标 JVM，就需要做一些额外的开发了。

com.sun.tools.attach.VirtualMachine 这个类代表一个 JVM 抽象，可以通过这个类找到目标 JVM，并且将 Agent 挂载到目标 JVM 上。下面是使用 com.sun.tools.attach.VirtualMachine 进行动态挂载 Agent 的一般实现：

```
    private void attachAgentToTargetJVM() throws Exception {
        List<VirtualMachineDescriptor> virtualMachineDescriptors =
VirtualMachine.list();
        VirtualMachineDescriptor targetVM = null;
        for (VirtualMachineDescriptor descriptor :
virtualMachineDescriptors) {
            if (descriptor.id().equals(configure.getPid())) {
                targetVM = descriptor;
                break;
            }
        }
        if (targetVM == null) {
            throw new IllegalArgumentException("could not find the
target jvm by process id:" +
configure.getPid());
        }
        VirtualMachine virtualMachine = null;
        try {
            virtualMachine = VirtualMachine.attach(targetVM);
            virtualMachine.loadAgent("{agent}", "{params}");
        } catch (Exception e) {
            if (virtualMachine != null) {
                virtualMachine.detach();
            }
        }
    }
```

首先通过指定的进程 ID 找到目标 JVM，然后通过 Attach 挂载到目标 JVM 上，执行加载 Agent 操作。VirtualMachine 的 Attach 方法就是用来将 Agent 挂载到目标 JVM 上去的，而 Detach 则是将 Agent 从目标 JVM 卸载。关于 Agent 是如何挂载到目标 JVM 上的具体技术细节，将在下文中进行分析。

2.2 启动时加载 Agent

2.2.1 参数解析

创建 JVM 时，JVM 会进行参数解析，即解析那些用来配置 JVM 启动的参数，比如堆大小、GC 等；本文主要关注解析的参数为 -agentlib、-agentpath、-javaagent，这几个参数用来指定 Agent，JVM 会根据这几个参数加载 Agent。下面来分析一下 JVM 是如何解析这几个参数的。

```
  // -agentlib and -agentpath
  if (match_option(option, "-agentlib:", &tail) ||
          (is_absolute_path = match_option(option, "-agentpath:",
&tail))) {
      if(tail != NULL) {
        const char* pos = strchr(tail, '=');
        size_t len = (pos == NULL) ? strlen(tail) : pos - tail;
        char* name = strncpy(NEW_C_HEAP_ARRAY(char, len + 1,
mtArguments), tail, len);
        name[len] = '\0';
        char *options = NULL;
        if(pos != NULL) {
          options = os::strdup_check_oom(pos + 1, mtArguments);
        }
#if !INCLUDE_JVMTI
        if (valid_jdwp_agent(name, is_absolute_path)) {
          jio_fprintf(defaultStream::error_stream(),
            "Debugging agents are not supported in this VM\n");
          return JNI_ERR;
        }
#endif // !INCLUDE_JVMTI
        add_init_agent(name, options, is_absolute_path);
      }
    // -javaagent
    } else if (match_option(option, "-javaagent:", &tail)) {
#if !INCLUDE_JVMTI
      jio_fprintf(defaultStream::error_stream(),
        "Instrumentation agents are not supported in this VM\n");
      return JNI_ERR;
#else
      if (tail != NULL) {
        size_t length = strlen(tail) + 1;
        char *options = NEW_C_HEAP_ARRAY(char, length, mtArguments);
        jio_snprintf(options, length, "%s", tail);
        add_init_agent("instrument", options, false);
        // java agents need module java.instrument
        if (!create_numbered_property("jdk.module.addmods", "java.
instrument", addmods_
count++)) {
          return JNI_ENOMEM;
        }
      }
#endif // !INCLUDE_JVMTI
    }
```

上面的代码片段截取自 hotspot/src/share/vm/runtime/arguments.cpp 中的 Arguments::parse_each_vm_init_arg(const JavaVMInitArgs* args, bool* patch_

mod_javabase, Flag::Flags origin) 函数，该函数用来解析一个具体的 JVM 参数。这段代码的主要功能是解析出需要加载的 Agent 路径，然后调用 add_init_agent 函数进行解析结果的存储。下面先看一下 add_init_agent 函数的具体实现：

```
  // -agentlib and -agentpath arguments
  static AgentLibraryList _agentList;
  static void add_init_agent(const char* name, char* options, bool
absolute_path)
    { _agentList.add(new AgentLibrary(name, options, absolute_path,
NULL)); }
```

AgentLibraryList 是一个简单的链表结构，add_init_agent 函数将解析好的、需要加载的 Agent 添加到这个链表中，等待后续的处理。

这里需要注意，解析 -javaagent 参数有一些特别之处，这个参数用来指定一个我们通过 Java Instrumentation API 来编写的 Agent，Java Instrumentation API 底层依赖的是 JVMTI，对 -JavaAgent 的处理也说明了这一点，在调用 add_init_agent 函数时第一个参数是“instrument”，关于加载 Agent 这个问题在下一小节进行展开。到此，我们知道在启动 JVM 时指定的 Agent 已经被 JVM 解析完存放在了一个链表结构中。下面来分析一下 JVM 是如何加载这些 Agent 的。

2.2.2 执行加载操作

在创建 JVM 进程的函数中，解析完 JVM 参数之后，下面的这段代码和加载 Agent 相关：

```
  // Launch -agentlib/-agentpath and converted -Xrun agents
  if (Arguments::init_agents_at_startup()) {
    create_vm_init_agents();
  }
  static bool init_agents_at_startup() {
    return !_agentList.is_empty();
  }
```

当 JVM 判断出上一小节中解析出来的 Agent 不为空的时候，就要去调用函数 create_vm_init_agents 来加载 Agent，下面来分析一下 create_vm_init_agents 函数是如何加载 Agent 的。

```
void Threads::create_vm_init_agents() {
  AgentLibrary* agent;
  for (agent = Arguments::agents(); agent != NULL; agent = agent->next())
{
    OnLoadEntry_t  on_load_entry = lookup_agent_on_load(agent);
    if (on_load_entry != NULL) {
      // Invoke the Agent_OnLoad function
      jint err = (*on_load_entry)(&main_vm, agent->options(), NULL);
    }
  }
}
```

create_vm_init_agents 这个函数通过遍历 Agent 链表来逐个加载 Agent。通过这段代码可以看出，首先通过 lookup_agent_on_load 来加载 Agent 并且找到 Agent_OnLoad 函数，这个函数是 Agent 的入口函数。如果没找到这个函数，则认为是加载了一个不合法的 Agent，则什么也不做，否则调用这个函数，这样 Agent 的代码就开始执行起来了。对于使用 Java Instrumentation API 来编写 Agent 的方式来说，在解析阶段观察到在 add_init_agent 函数里面传递进去的是一个叫做”instrument”的字符串，其实这是一个动态链接库。在 Linux 里面，这个库叫做 libinstrument.so，在 BSD 系统中叫做 libinstrument.dylib，该动态链接库在 {JAVA_HOME}/jre/lib/ 目录下。

2.2.3 instrument 动态链接库

libinstrument 用来支持使用 Java Instrumentation API 来编写 Agent，在 libinstrument 中有一个非常重要的类称为：JPLISAgent（Java Programming Language Instrumentation Services Agent），它的作用是初始化所有通过 Java Instrumentation API 编写的 Agent，并且也承担着通过 JVMTI 实现 Java Instrumentation 中暴露 API 的责任。

我们已经知道，在 JVM 启动的时候，JVM 会通过 -javaagent 参数加载 Agent。最开始加载的是 libinstrument 动态链接库，然后在动态链接库里面找到 JVMTI 的入口方法：Agent_OnLoad。下面就来分析一下在 libinstrument 动态链接库中，Agent_OnLoad 函数是怎么实现的。

```
JNIEXPORT jint JNICALL
DEF_Agent_OnLoad(JavaVM *vm, char *tail, void * reserved) {
    initerror = createNewJPLISAgent(vm, &agent);
    if ( initerror == JPLIS_INIT_ERROR_NONE ) {
        if (parseArgumentTail(tail, &jarfile, &options) != 0) {
            fprintf(stderr, "-javaagent: memory allocation failure.\n");
            return JNI_ERR;
        }
        attributes = readAttributes(jarfile);
        premainClass = getAttribute(attributes, "Premain-Class");
        /* Save the jarfile name */
        agent->mJarfile = jarfile;
        /*
         * Convert JAR attributes into agent capabilities
         */
        convertCapabilityAttributes(attributes, agent);
        /*
         * Track (record) the agent class name and options data
         */
        initerror = recordCommandLineData(agent, premainClass, options);
    }
    return result;
}
```

上述代码片段是经过精简的 libinstrument 中 Agent_OnLoad 实现的，大概的流程就是：先创建一个 JPLISAgent，然后将 ManiFest 中设定的一些参数解析出来，比如（Premain-Class）等。创建了 JPLISAgent 之后，调用 initializeJPLISAgent 对这个 Agent 进行初始化操作。跟进 initializeJPLISAgent 看一下是如何初始化的：

```
JPLISInitializationError initializeJPLISAgent(JPLISAgent *agent, JavaVM
*vm, jvmtiEnv *jvmtienv)
{
    /* check what capabilities are available */
    checkCapabilities(agent);
    /* check phase - if live phase then we don't need the VMInit event */
    jvmtierror = (*jvmtienv)->GetPhase(jvmtienv, &phase);
    /* now turn on the VMInit event */
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
        jvmtiEventCallbacks callbacks;
        memset(&callbacks, 0, sizeof(callbacks));
        callbacks.VMInit = &eventHandlerVMInit;
        jvmtierror = (*jvmtienv)->SetEventCallbacks(jvmtienv,&callbacks,
sizeof(callbacks));
    }
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
```

```
        jvmtierror = (*jvmtienv)->SetEventNotificationMode(jvmtienv,JVM
TI_ENABLE,JVMTI_EVENT_
VM_INIT,NULL);
    }
    return (jvmtierror == JVMTI_ERROR_NONE)? JPLIS_INIT_ERROR_NONE :
JPLIS_INIT_ERROR_
FAILURE;
}
```

这里，我们关注 callbacks.VMInit = &eventHandlerVMInit; 这行代码，这里设置了一个 VMInit 事件的回调函数，表示在 JVM 初始化的时候会回调 eventHandlerVMInit 函数。下面来看一下这个函数的实现细节，猜测就是在这里调用了 Premain 方法：

```
void JNICALL  eventHandlerVMInit( jvmtiEnv *jvmtienv,JNIEnv
*jnienv,jthread thread) {
   // ...
   success = processJavaStart( environment->mAgent, jnienv);
  // ...
}
jboolean  processJavaStart(JPLISAgent *agent,JNIEnv *jnienv) {
    result = createInstrumentationImpl(jnienv, agent);
    /*
     *  Load the Java agent, and call the premain.
     */
    if ( result ) {
        result = startJavaAgent(agent, jnienv, agent->mAgentClassName,
agent->mOptionsString,
agent->mPremainCaller);
    }
    return result;
}
jboolean startJavaAgent( JPLISAgent *agent,JNIEnv *jnienv,const char
*classname,const char
*optionsString,jmethodID agentMainMethod) {
  // ...
  invokeJavaAgentMainMethod(jnienv,agent->mInstrumentationImpl,agentMainM
ethod,
classNameObject,optionsStringObject);
  // ...
}
```

看到这里，Instrument 已经实例化，invokeJavaAgentMainMethod 这个方法将我们的 premain 方法执行起来了。接着，我们就可以根据 Instrument 实例来做我

们想要做的事情了。

2.3 运行时加载 Agent

比起 JVM 启动时加载 Agent，运行时加载 Agent 就比较有诱惑力了，因为运行时加载 Agent 的能力给我们提供了很强的动态性，我们可以在需要的时候加载 Agent 来进行一些工作。因为是动态的，我们可以按照需求来加载所需要的 Agent，下面来分析一下动态加载 Agent 的相关技术细节。

2.3.1 AttachListener

Attach 机制通过 Attach Listener 线程来进行相关事务的处理，下面来看一下 Attach Listener 线程是如何初始化的。

```
// Starts the Attach Listener thread
void AttachListener::init() {
  // 创建线程相关部分代码被去掉了
  const char thread_name[] = "Attach Listener";
  Handle string = java_lang_String::create_from_str(thread_name, THREAD);
  { MutexLocker mu(Threads_lock);
    JavaThread* listener_thread = new JavaThread(&attach_listener_thread_
entry);
    // ...
  }
}
```

我们知道，一个线程启动之后都需要指定一个入口来执行代码，Attach Listener 线程的入口是 attach_listener_thread_entry，下面看一下这个函数的具体实现：

```
static void attach_listener_thread_entry(JavaThread* thread, TRAPS) {
  AttachListener::set_initialized();
  for (;;) {
      AttachOperation* op = AttachListener::dequeue();
      // find the function to dispatch too
      AttachOperationFunctionInfo* info = NULL;
      for (int i=0; funcs[i].name != NULL; i++) {
        const char* name = funcs[i].name;
        if (strcmp(op->name(), name) == 0) {
          info = &(funcs[i]); break;
        }}
       // dispatch to the function that implements this operation
```

```
        res = (info->func)(op, &st);
      //...
    }
}
```

整个函数执行逻辑，大概是这样的：

- 拉取一个需要执行的任务：AttachListener::dequeue。
- 查询匹配的命令处理函数。
- 执行匹配到的命令执行函数。

其中第二步里面存在一个命令函数表，整个表如下：

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             load_agent },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

对于加载 Agent 来说，命令就是“load”。现在，我们知道了 Attach Listener 大概的工作模式，但是还是不太清楚任务从哪来，这个秘密就藏在 AttachListener::dequeue 这行代码里面，接下来我们来分析一下 dequeue 这个函数：

```
LinuxAttachOperation* LinuxAttachListener::dequeue() {
  for (;;) {
    // wait for client to connect
    struct sockaddr addr;
    socklen_t len = sizeof(addr);
    RESTARTABLE(::accept(listener(), &addr, &len), s);
    // get the credentials of the peer and check the effective uid/guid
    // - check with jeff on this.
    struct ucred cred_info;
    socklen_t optlen = sizeof(cred_info);
    if (::getsockopt(s, SOL_SOCKET, SO_PEERCRED, (void*)&cred_info,
&optlen) == -1) {
```

```
        ::close(s);
        continue;
      }
      // peer credential look okay so we read the request
      LinuxAttachOperation* op = read_request(s);
      return op;
    }
  }
```

这是 Linux 上的实现，不同的操作系统实现方式不太一样。上面的代码表面，Attach Listener 在某个端口监听着，通过 accept 来接收一个连接，然后从这个连接里面将请求读取出来，然后将请求包装成一个 AttachOperation 类型的对象，之后就会从表里查询对应的处理函数，然后进行处理。

Attach Listener 使用一种被称为“懒加载”的策略进行初始化，也就是说，JVM 启动的时候 Attach Listener 并不一定会启动起来。下面我们来分析一下这种“懒加载”策略的具体实现方案。

```
  // Start Attach Listener if +StartAttachListener or it can't be
started lazily
  if (!DisableAttachMechanism) {
    AttachListener::vm_start();
    if (StartAttachListener || AttachListener::init_at_startup()) {
      AttachListener::init();
    }
  }
// Attach Listener is started lazily except in the case when
// +ReduseSignalUsage is used
bool AttachListener::init_at_startup() {
  if (ReduceSignalUsage) {
    return true;
  } else {
    return false;
  }
}
```

上面的代码截取自 create_vm 函数，DisableAttachMechanism、StartAttachListener 和 ReduceSignalUsage 这三个变量默认都是 false，所以 AttachListener::init(); 这行代码不会在 create_vm 的时候执行，而 vm_start 会执行。下面来看一下这个函数的实现细节：

```
void AttachListener::vm_start() {
  char fn[UNIX_PATH_MAX];
  struct stat64 st;
  int ret;
  int n = snprintf(fn, UNIX_PATH_MAX, "%s/.java_pid%d",
           os::get_temp_directory(), os::current_process_id());
  assert(n < (int)UNIX_PATH_MAX, "java_pid file name buffer overflow");
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == 0) {
    ret = ::unlink(fn);
    if (ret == -1) {
      log_debug(attach)("Failed to remove stale attach pid file at %s", fn);
    }
  }
}
```

这是在 Linux 上的实现，是将 /tmp/ 目录下的 .java_pid{pid} 文件删除，后面在创建 Attach Listener 线程的时候会创建出来这个文件。上面说到，AttachListener::init() 这行代码不会在 create_vm 的时候执行，这行代码的实现已经在上文中分析了，就是创建 Attach Listener 线程，并监听其他 JVM 的命令请求。现在来分析一下这行代码是什么时候被调用的，也就是“懒加载”到底是怎么加载起来的。

```
  // Signal Dispatcher needs to be started before VMInit event is posted
  os::signal_init();
```

这是 create_vm 中的一段代码，看起来跟信号相关，其实 Attach 机制就是使用信号来实现“懒加载“的。下面我们来仔细地分析一下这个过程。

```
void os::signal_init() {
  if (!ReduceSignalUsage) {
    // Setup JavaThread for processing signals
    EXCEPTION_MARK;
    Klass* k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_
Thread(), true, CHECK);
    instanceKlassHandle klass (THREAD, k);
    instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);
    const char thread_name[] = "Signal Dispatcher";
    Handle string = java_lang_String::create_from_str(thread_name,
CHECK);
    // Initialize thread_oop to put it into the system threadGroup
    Handle thread_group (THREAD, Universe::system_thread_group());
    JavaValue result(T_VOID);
```

```
    JavaCalls::call_special(&result, thread_oop,klass,vmSymbols::object_
initializer_name(),vmSymbols::threadgroup_string_void_signature(),
                            thread_group,string,CHECK);
    KlassHandle group(THREAD, SystemDictionary::ThreadGroup_klass());
    JavaCalls::call_special(&result,thread_group,group,vmSymbols::add_
method_name(),vmSymbols::thread_void_signature(),thread_oop,CHECK);
    os::signal_init_pd();
    { MutexLocker mu(Threads_lock);
      JavaThread* signal_thread = new JavaThread(&signal_thread_entry);
     // ...
    }
    // Handle ^BREAK
    os::signal(SIGBREAK, os::user_handler());
  }
}
```

JVM 创建了一个新的进程来实现信号处理，这个线程叫“Signal Dispatcher”，一个线程创建之后需要有一个入口，“Signal Dispatcher”的入口是 signal_thread_entry：

```
263
264     switch (sig) {
265       case SIGBREAK: {
266         // Check if the signal is a trigger to start the Attach Listener - in that
267         // case don't print stack traces.
268         if (!DisableAttachMechanism && AttachListener::is_init_trigger()) {
269           continue;
270         }
```

这段代码截取自 signal_thread_entry 函数，截取中的内容是和 Attach 机制信号处理相关的代码。这段代码的意思是，当接收到“SIGBREAK”信号，就执行接下来的代码，这个信号是需要 Attach 到 JVM 上的信号发出来，这个后面会再分析。我们先来看一句关键的代码：AttachListener::is_init_trigger()：

```
bool AttachListener::is_init_trigger() {
  if (init_at_startup() || is_initialized()) {
    return false;               // initialized at startup or already
initialized
  }
  char fn[PATH_MAX+1];
  sprintf(fn, ".attach_pid%d", os::current_process_id());
  int ret;
  struct stat64 st;
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == -1) {
```

```
    log_trace(attach)("Failed to find attach file: %s, trying
alternate", fn);
    snprintf(fn, sizeof(fn), "%s/.attach_pid%d", os::get_temp_
directory(), os::current_process_
id());
    RESTARTABLE(::stat64(fn, &st), ret);
  }
  if (ret == 0) {
    // simple check to avoid starting the attach mechanism when
    // a bogus user creates the file
    if (st.st_uid == geteuid()) {
      init();
      return true;
    }
  }
  return false;
}
```

首先检查了一下是否在 JVM 启动时启动了 Attach Listener，或者是否已经启动过。如果没有，才继续执行，在 /tmp 目录下创建一个叫做 .attach_pid%d 的文件，然后执行 AttachListener 的 init 函数，这个函数就是用来创建 Attach Listener 线程的函数，上面已经提到多次并进行了分析。到此，我们知道 Attach 机制的奥秘所在，也就是 Attach Listener 线程的创建依靠 Signal Dispatcher 线程，Signal Dispatcher 是用来处理信号的线程，当 Signal Dispatcher 线程接收到“SIGBREAK”信号之后，就会执行初始化 Attach Listener 的工作。

2.3.2 运行时加载 Agent 的实现

我们继续分析，到底是如何将一个 Agent 挂载到运行着的目标 JVM 上，在上文中提到了一段代码，用来进行运行时挂载 Agent，可以参考上文中展示的关于“attachAgentToTargetJvm”方法的代码。这个方法里面的关键是调用 VirtualMachine 的 attach 方法进行 Agent 挂载的功能。下面我们就来分析一下 VirtualMachine 的 attach 方法具体是怎么实现的。

```
    public static VirtualMachine attach(String var0) throws
AttachNotSupportedException,
IOException {
        if (var0 == null) {
            throw new NullPointerException("id cannot be null");
```

```
        } else {
            List var1 = AttachProvider.providers();
            if (var1.size() == 0) {
                throw new AttachNotSupportedException("no providers
installed");
            } else {
                AttachNotSupportedException var2 = null;
                Iterator var3 = var1.iterator();
                while(var3.hasNext()) {
                    AttachProvider var4 = (AttachProvider)var3.next();
                    try {
                        return var4.attachVirtualMachine(var0);
                    } catch (AttachNotSupportedException var6) {
                        var2 = var6;
                    }
                }
                throw var2;
            }
        }
    }
```

这个方法通过 attachVirtualMachine 方法进行 attach 操作，在 MacOS 系统中，AttachProvider 的实现类是 BsdAttachProvider。我们来看一下 BsdAttach-Provider 的 attachVirtualMachine 方法是如何实现的：

```
public VirtualMachine attachVirtualMachine(String var1) throws
AttachNotSupportedException, IOException {
        this.checkAttachPermission();
        this.testAttachable(var1);
        return new BsdVirtualMachine(this, var1);
    }
BsdVirtualMachine(AttachProvider var1, String var2) throws
AttachNotSupportedException,
IOException {
        int var3 = Integer.parseInt(var2);
        this.path = this.findSocketFile(var3);
        if (this.path == null) {
            File var4 = new File(tmpdir, ".attach_pid" + var3);
            createAttachFile(var4.getPath());
            try {
                sendQuitTo(var3);
                int var5 = 0;
                long var6 = 200L;
                int var8 = (int)(this.attachTimeout() / var6);
                do {
                    try {
```

```
                        Thread.sleep(var6);
                    } catch (InterruptedException var21) {
                        ;
                    }
                    this.path = this.findSocketFile(var3);
                    ++var5;
                } while(var5 <= var8 && this.path == null);
            } finally {
                var4.delete();
            }
        }
        int var24 = socket();
        connect(var24, this.path);
    }
    private String findSocketFile(int var1) {
        String var2 = ".java_pid" + var1;
        File var3 = new File(tmpdir, var2);
        return var3.exists() ? var3.getPath() : null;
    }
```

findSocketFile 方法用来查询目标 JVM 上是否已经启动了 Attach Listener，它通过检查”tmp/“目录下是否存在 java_pid{pid} 来进行实现。如果已经存在了，则说明 Attach 机制已经准备就绪，可以接受客户端的命令了，这个时候客户端就可以通过 connect 连接到目标 JVM 进行命令的发送，比如可以发送“load”命令来加载 Agent。如果 java_pid{pid} 文件还不存在，则需要通过 sendQuitTo 方法向目标 JVM 发送一个“SIGBREAK”信号，让它初始化 Attach Listener 线程并准备接受客户端连接。可以看到，发送了信号之后客户端会循环等待 java_pid{pid} 这个文件，之后再通过 connect 连接到目标 JVM 上。

2.3.3 load 命令的实现

下面来分析一下，“load”命令在 JVM 层面的实现：

```
static jint load_agent(AttachOperation* op, outputStream* out) {
  // get agent name and options
  const char* agent = op->arg(0);
  const char* absParam = op->arg(1);
  const char* options = op->arg(2);
  // If loading a java agent then need to ensure that the java.
instrument module is loaded
  if (strcmp(agent, "instrument") == 0) {
```

```
    Thread* THREAD = Thread::current();
    ResourceMark rm(THREAD);
    HandleMark hm(THREAD);
    JavaValue result(T_OBJECT);
    Handle h_module_name = java_lang_String::create_from_str("java.
instrument", THREAD);
    JavaCalls::call_static(&result,SystemDictionary::module_Modules_
klass(),vmSymbols::loadModule_name(),
                           vmSymbols::loadModule_signature(),h_module_
name,THREAD);
  }
  return JvmtiExport::load_agent_library(agent, absParam, options, out);
}
```

这个函数先确保加载了 java.instrument 模块，之后真正执行 Agent 加载的函数是 load_agent_library，这个函数的套路就是加载 Agent 动态链接库，如果是通过 Java instrument API 实现的 Agent，则加载的是 libinstrument 动态链接库，然后通过 libinstrument 里面的代码实现运行 agentmain 方法的逻辑，这一部分内容和 libinstrument 实现 premain 方法运行的逻辑其实差不多，这里不再做分析。至此，我们对 Java Agent 技术已经有了一个全面而细致的了解。

3.1 动态字节码修改的限制

上文中已经详细分析了 Agent 技术的实现，我们使用 Java Instrumentation API 来完成动态类修改的功能，在 Instrumentation 接口中，通过 addTransformer 方法来增加一个类转换器，类转换器由类 ClassFileTransformer 接口实现。ClassFileTransformer 接口中唯一的方法 transform 用于实现类转换，当类被加载的时候，就会调用 transform 方法，进行类转换。在运行时，我们可以通过 Instrumentation 的 redefineClasses 方法进行类重定义，在方法上有一段注释需要特别注意：

```
     * The redefinition may change method bodies, the constant pool and
attributes.
     * The redefinition must not add, remove or rename fields or methods,
change the
     * signatures of methods, or change inheritance.  These restrictions
maybe be
```

```
     * lifted in future versions.  The class file bytes are not checked,
verified and installed
     * until after the transformations have been applied, if the
resultant bytes are in
     * error this method will throw an exception.
```

这里面提到，我们不可以增加、删除或者重命名字段和方法，改变方法的签名或者类的继承关系。认识到这一点很重要，当我们通过 ASM 获取到增强的字节码之后，如果增强后的字节码没有遵守这些规则，那么调用 redefineClasses 方法来进行类的重定义就会失败。那 redefineClasses 方法具体是怎么实现类的重定义的呢？它对运行时的 JVM 会造成什么样的影响呢？下面来分析 redefineClasses 的实现细节。

3.2 重定义类字节码的实现细节

上文中我们提到，libinstrument 动态链接库中，JPLISAgent 不仅实现了 Agent 入口代码执行的路由，而且还是 Java 代码与 JVMTI 之间的一道桥梁。我们在 Java 代码中调用 Java Instrumentation API 的 redefineClasses，其实会调用 libinstrument 中的相关代码，我们来分析一下这条路径。

```
    public void redefineClasses(ClassDefinition... var1) throws
ClassNotFoundException {
        if (!this.isRedefineClassesSupported()) {
            throw new UnsupportedOperationException("redefineClasses is
not supported in this environment");
        } else if (var1 == null) {
            throw new NullPointerException("null passed as
'definitions' in redefineClasses");
        } else {
            for(int var2 = 0; var2 < var1.length; ++var2) {
                if (var1[var2] == null) {
                    throw new NullPointerException("element of
'definitions' is null in redefineClasses");
                }
            }
            if (var1.length != 0) {
                this.redefineClasses0(this.mNativeAgent, var1);
            }
        }
    }
```

```
    private native void redefineClasses0(long var1, ClassDefinition[]
var3) throws
ClassNotFoundException;
```

这是 InstrumentationImpl 中的 redefineClasses 实现，该方法的具体实现依赖一个 Native 方法 redefineClasses()，我们可以在 libinstrument 中找到这个 Native 方法的实现：

```
JNIEXPORT void JNICALL Java_sun_instrument_InstrumentationImpl_
redefineClasses0
  (JNIEnv * jnienv, jobject implThis, jlong agent, jobjectArray
classDefinitions) {
    redefineClasses(jnienv, (JPLISAgent*)(intptr_t)agent,
classDefinitions);
}
```

redefineClasses 这个函数的实现比较复杂，代码很长。下面是一段关键的代码片段：

```
if (!errorOccurred) {
    jvmtiError  errorCode = JVMTI_ERROR_NONE;
    errorCode = (*jvmtienv)->RedefineClasses(jvmtienv, numDefs, classDefs);
    if (errorCode == JVMTI_ERROR_WRONG_PHASE) {
        /* insulate caller from the wrong phase error */
        errorCode = JVMTI_ERROR_NONE;
    } else {
        errorOccurred = (errorCode != JVMTI_ERROR_NONE);
        if ( errorOccurred ) {
            createAndThrowThrowableFromJVMTIErrorCode(jnienv, errorCode);
        }
    }
}
```

可以看到，其实是调用了 JVMTI 的 RetransformClasses 函数来完成类的重定义细节。

```
// class_count - pre-checked to be greater than or equal to 0
// class_definitions - pre-checked for NULL
jvmtiError JvmtiEnv::RedefineClasses(jint class_count, const
jvmtiClassDefinition* class_
definitions) {
//TODO: add locking
  VM_RedefineClasses op(class_count, class_definitions, jvmti_class_
load_kind_redefine);
  VMThread::execute(&op);
```

```
    return (op.check_error());
  } /* end RedefineClasses */
```

重定义类的请求会被 JVM 包装成一个 VM_RedefineClasses 类型的 VM_Operation，VM_Operation 是 JVM 内部的一些操作的基类，包括 GC 操作等。VM_Operation 由 VMThread 来执行，新的 VM_Operation 操作会被添加到 VMThread 的运行队列中去，VMThread 会不断从队列里面拉取 VM_Operation 并调用其 doit 等函数执行具体的操作。VM_RedefineClasses 函数的流程较为复杂，下面是 VM_RedefineClasses 的大致流程：

- 加载新的字节码，合并常量池，并且对新的字节码进行校验工作

```
    // Load the caller's new class definition(s) into _scratch_classes.
    // Constant pool merging work is done here as needed. Also calls
    // compare_and_normalize_class_versions() to verify the class
    // definition(s).
    jvmtiError load_new_class_versions(TRAPS);
```

- 清除方法上的断点

```
    // Remove all breakpoints in methods of this class
    JvmtiBreakpoints& jvmti_breakpoints = JvmtiCurrentBreakpoints::get_
  jvmti_breakpoints();
    jvmti_breakpoints.clearall_in_class_at_safepoint(the_class());
```

- JIT 逆优化

```
    // Deoptimize all compiled code that depends on this class
    flush_dependent_code(the_class, THREAD);
```

- 进行字节码替换工作，需要进行更新类 itable/vtable 等操作
- 进行类重定义通知

```
    SystemDictionary::notice_modification();
```

VM_RedefineClasses 实现比较复杂的，详细实现可以参考 RedefineClasses 的实现。

Java-debug-tool 是一个使用 Java Instrument API 来实现的动态调试工具，它通过在目标 JVM 上启动一个 TcpServer 来和调试客户端通信。调试客户端通过命令行来发送调试命令给 TcpServer，TcpServer 中有专门用来处理命令的 handler，handler 处理完命令之后会将结果发送回客户端，客户端通过处理将调试结果展示出来。下面将详细介绍 Java-debug-tool 的整体设计和实现。

4.1 Java-debug-tool 整体架构

Java-debug-tool 包括一个 Java Agent 和一个用于处理调试命令的核心 API，核心 API 通过一个自定义的类加载器加载进来，以保证目标 JVM 的类不会被污染。整体上 Java-debug-tool 的设计是一个 Client-Server 的架构，命令客户端需要完整的完成一个命令之后才能继续执行下一个调试命令。Java-debug-tool 支持多人同时进行调试，下面是整体架构图：

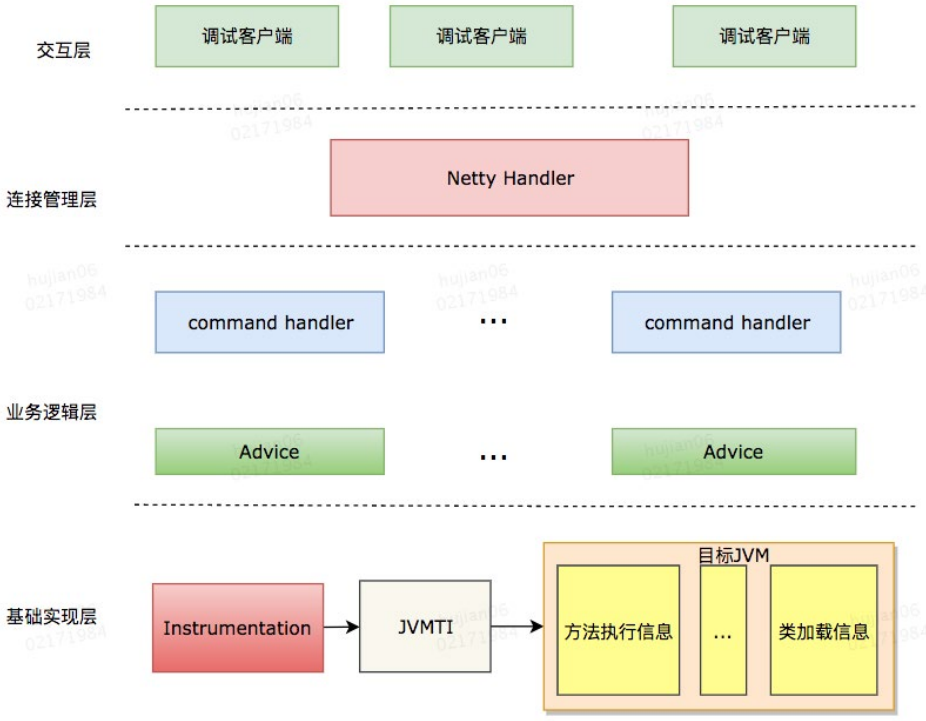


图 1

下面对每一层做简单介绍：

- 交互层：负责将程序员的输入转换成调试交互协议，并且将调试信息呈现出来。
- 连接管理层：负责管理客户端连接，从连接中读调试协议数据并解码，对调试结果编码并将其写到连接中去；同时将那些超时未活动的连接关闭。
- 业务逻辑层：实现调试命令处理，包括命令分发、数据收集、数据处理等过程。
- 基础实现层：Java-debug-tool 实现的底层依赖，通过 Java Instrumentation 提供的 API 进行类查找、类重定义等能力，Java Instrumentation 底层依赖 JVMTI 来完成具体的功能。

在 Agent 被挂载到目标 JVM 上之后，Java-debug-tool 会安排一个 Spy 在目标 JVM 内活动，这个 Spy 负责将目标 JVM 内部的相关调试数据转移到命令处理模块，命令处理模块会处理这些数据，然后给客户端返回调试结果。命令处理模块会增强目标类的字节码来达到数据获取的目的，多个客户端可以共享一份增强过的字节码，无需重复增强。下面从 Java-debug-tool 的字节码增强方案、命令设计与实现等角度详细说明。

4.2 Java-debug-tool 的字节码增强方案

Java-debug-tool 使用字节码增强来获取到方法运行时的信息，比如方法入参、出参等，可以在不同的字节码位置进行增强，这种行为可以称为“插桩”，每个“桩”用于获取数据并将他转储出去。Java-debug-tool 具备强大的插桩能力，不同的桩负责获取不同类别的数据，下面是 Java-debug-tool 目前所支持的“桩”：

- 方法进入点：用于获取方法入参信息。
- Fields 获取点 1：在方法执行前获取到对象的字段信息。
- 变量存储点：获取局部变量信息。
- Fields 获取点 2：在方法退出前获取到对象的字段信息。
- 方法退出点：用于获取方法返回值。
- 抛出异常点：用于获取方法抛出的异常信息。

通过上面这些代码桩，Java-debug-tool 可以收集到丰富的方法执行信息，经过处理可以返回更加可视化的调试结果。

4.2.1 字节码增强

Java-debug-tool 在实现上使用了 ASM 工具来进行字节码增强，并且每个插桩点都可以进行配置，如果不想要什么信息，则没必要进行对应的插桩操作。这种可配置的设计是非常有必要的，因为有时候我们仅仅是想要知道方法的入参和出参，但 Java-debug-tool 却给我们返回了所有的调试信息，这样我们就得在众多的输出中找到我们所关注的内容。如果可以进行配置，则除了入参点和出参点外其他的桩都不插，那么就可以快速看到我们想要的调试数据，这种设计的本质是为了让调试者更加专注。下面是 Java-debug-tool 的字节码增强工作方式：

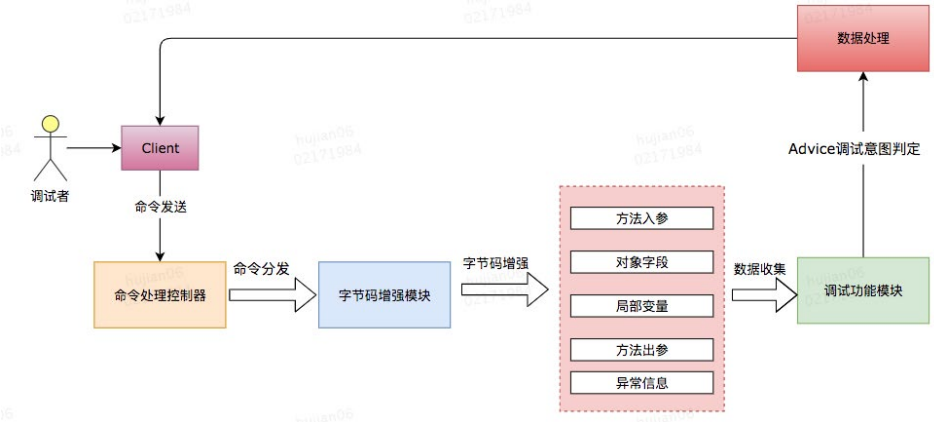


图 2

如图 2 所示，当调试者发出调试命令之后，Java-debug-tool 会识别命令并判断是否需要进行字节码增强，如果命令需要增强字节码，则判断当前类 + 当前方法是否已经被增强过。上文已经提到，字节码替换是有一定损耗的，这种具有损耗的操作发生的次数越少越好，所以字节码替换操作会被记录起来，后续命令直接使用即可，不需要重复进行字节码增强，字节码增强还涉及多个调试客户端的协同工作问题，当一个客户端增强了一个类的字节码之后，这个客户端就锁定了该字节码，其他客户端

变成只读，无法对该类进行字节码增强，只有当持有锁的客户端主动释放锁或者断开连接之后，其他客户端才能继续增强该类的字节码。

字节码增强模块收到字节码增强请求之后，会判断每个增强点是否需要插桩，这个判断的根据就是上文提到的插桩配置，之后字节码增强模块会生成新的字节码，Java-debug-tool 将执行字节码替换操作，之后就可以进行调试数据收集了。

经过字节码增强之后，原来的方法中会插入收集运行时数据的代码，这些代码在方法被调用的时候执行，获取到诸如方法入参、局部变量等信息，这些信息将传递给数据收集装置进行处理。数据收集的工作通过 Advice 完成，每个客户端同一时间只能注册一个 Advice 到 Java-debug-tool 调试模块上，多个客户端可以同时注册自己的 Advice 到调试模块上。Advice 负责收集数据并进行判断，如果当前数据符合调试命令的要求，Java-debug-tool 就会卸载这个 Advice，Advice 的数据就会被转移到 Java-debug-tool 的命令结果处理模块进行处理，并将结果发送到客户端。

4.2.2 Advice 的工作方式

Advice 是调试数据收集器，不同的调试策略会对应不同的 Advice。Advice 是工作在目标 JVM 的线程内部的，它需要轻量级和高效，意味着 Advice 不能做太过于复杂的事情，它的核心接口“match”用来判断本次收集到的调试数据是否满足调试需求。如果满足，那么 Java-debug-tool 就会将其卸载，否则会继续让他收集调试数据，这种“加载 Advice”->“卸载 Advice”的工作模式具备很好的灵活性。

关于 Advice，需要说明的另外一点就是线程安全，因为它加载之后会运行在目标 JVM 的线程中，目标 JVM 的方法极有可能是多线程访问的，这也就是说，Advice 需要有能力处理多个线程同时访问方法的能力，如果 Advice 处理不当，则可能会收集到杂乱无章的调试数据。下面的图片展示了 Advice 和 Java-debug-tool 调试分析模块、目标方法执行以及调试客户端等模块的关系。

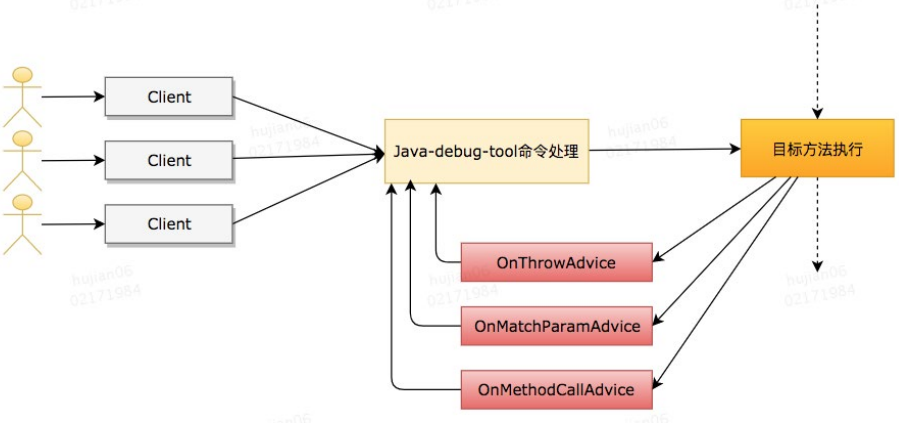


图 3

Advice 的首次挂载由 Java-debug-tool 的命令处理器完成，当一次调试数据收集完成之后，调试数据处理模块会自动卸载 Advice，然后进行判断，如果调试数据符合 Advice 的策略，则直接将数据交由数据处理模块进行处理，否则会清空调试数据，并再次将 Advice 挂载到目标方法上去，等待下一次调试数据。非首次挂载由调试数据处理模块进行，它借助 Advice 按需取数据，如果不符合需求，则继续挂载 Advice 来获取数据，否则对调试数据进行处理并返回给客户端。

4.3 Java-debug-tool 的命令设计与实现

4.3.1 命令执行

上文已经完整的描述了 Java-debug-tool 的设计以及核心技术方案，本小节将详细介绍 Java-debug-tool 的命令设计与实现。首先需要将一个调试命令的执行流程描述清楚，下面是一张用来表示命令请求处理流程的图片：

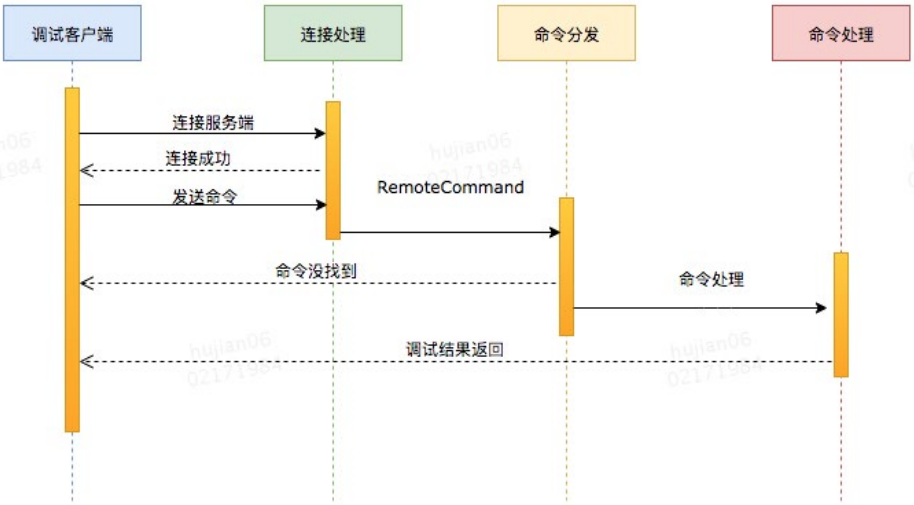


图 4

图 4 简单的描述了 Java-debug-tool 的命令处理方式，客户端连接到服务端之后，会进行一些协议解析、协议认证、协议填充等工作，之后将进行命令分发。服务端如果发现客户端的命令不合法，则会立即返回错误信息，否则再进行命令处理。命令处理属于典型的三段式处理，前置命令处理、命令处理以及后置命令处理，同时会对命令处理过程中的异常信息进行捕获处理，三段式处理的好处是命令处理被拆成了多个阶段，多个阶段负责不同的职责。前置命令处理用来做一些命令权限控制的工作，并填充一些类似命令处理开始时间戳等信息，命令处理就是通过字节码增强，挂载 Advice 进行数据收集，再经过数据处理来产生命令结果的过程，后置处理则用来处理一些连接关闭、字节码解锁等事项。

Java-debug-tool 允许客户端设置一个命令执行超时时间，超过这个时间则认为命令没有结果，如果客户端没有设置自己的超时时间，就使用默认的超时时间进行超时控制。Java-debug-tool 通过设计了两阶段的超时检测机制来实现命令执行超时功能：首先，第一阶段超时触发，则 Java-debug-tool 会友好的警告命令处理模块处理时间已经超时，需要立即停止命令执行，这允许命令自己做一些现场清理工作，当然需要命令执行线程自己感知到这种超时警告；当第二阶段超时触发，则

Java-debug-tool 认为命令必须结束执行，会强行打断命令执行线程。超时机制的目的是为了不让命令执行太长时间，命令如果长时间没有收集到调试数据，则应该停止执行，并思考是否调试了一个错误的方法。当然，超时机制还可以定期清理那些因为未知原因断开连接的客户端持有的调试资源，比如字节码锁。

4.3.2 获取方法执行视图

Java-debug-tool 通过下面的信息来向调试者呈现出一次方法执行的视图：

- 正在调试的方法信息。
- 方法调用堆栈。
- 调试耗时，包括对目标 JVM 造成的 STW 时间。
- 方法入参，包括入参的类型及参数值。
- 方法的执行路径。
- 代码执行耗时。
- 局部变量信息。
- 方法返回结果。
- 方法抛出的异常。
- 对象字段值快照。

图 5 展示了 Java-debug-tool 获取到正在运行的方法的执行视图的信息。

```
127.0.0.1:11234>mt -c R -m call -t watch -i #p0+#p1<10 -timeout 10 -s fd

-----------------------------------------------------------------------------------------------
命令                    : mt
命令执行Round           : 3
客户端ID                : 10000
客户端类型              : client:1
协议版本                : version:1
命令耗时                : 427 (ms)
STW时间                 : 0 (ms)
-----------------------------------------------------------------------------------------------
[R.call] by invoking:Thread[Thread-0,5,main]
with params
[
[0] @class:java.lang.Integer -> 2,
[1] @class:java.lang.Integer -> 4,
[2] @class:C -> {"@class":"C","a":0}
]
[1 ms] (34) [sa = 1]
[0 ms] (35)
[0 ms] (36) [ii = 0]
[0 ms] (37) [ij = 0]
[0 ms] (38) [jk = 1]
[0 ms] (39) [f = 1.0]
[0 ms] (40) [d = 0.123]
[0 ms] (41) [name = hello2,4]
[0 ms] (42) [list = [5]]
[0 ms] (43)
[0 ms] (47)
[0 ms] (50)
[0 ms] (51)
[0 ms] (53)
[0 ms] (57)
return value:[1]  at line:57 with cost:4 ms

Before Invoking Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[11]@int

Before Exiting Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[6]@int

-----------------------------------------------------------------------------------------------
```

图 5

4.4 Java-debug-tool 与同类产品对比分析

Java-debug-tool 的同类产品主要是 greys，其他类似的工具大部分都是基于 greys 进行的二次开发，所以直接选择 greys 来和 Java-debug-tool 进行对比。

对比项	greys	Java-debug-tool	说明
多调试客户端支持	支持	支持	都是用观察者模式来支持多调试客户端同时调试；
方法增强	入参/返回/抛出异常	入参/执行路径/局部变量/出参/抛出异常/对象字段	相比greys，Java-debug-tool有更多的插桩选择，意味着可以获取到更丰富的调试信息
工具使用难度	适中，但是命令较多，需要配和多个命令进行调试	适中，调试集中在很少几个命令上	因为是基于shell的交互模式，两种工具都存在一定的使用难度
类隔离	自定义类加载器加载工具代码与目标JVM的类进行隔离	使用自定义类加载器	这样对目标JVM不会产生类污染
Agent加载模式	运行时加载	运行时加载	这也符合"动态调试"的场景需求
交互模式	命令行	命令行	greys的调试服务端是一个使用NIO实现的tcpServer，Java-debug-tool使用Netty实现调试服务端
命令数量	10+	5+	greys提供了大量的命令，而Java-debug-tool则专注于方法级别的链路追踪，没有提供太多的命令
表达式支持	支持，使用OGNL表达式；	支持，使用Spring表达式	Java-debug-tool的表达式仅在方法追踪时有用；
方法追踪调试信息获取	指定具体的调试信息，比如方法退出前；tt命令可以记录方法的执行信息；	执行mt命令，默认即可获取到包括方法入参、执行路径、执行耗时（包括单行耗时）、方法出参、抛出的异常等丰富的信息；	Java-debug-tool的mt命令和单步调试更类似，区别就是Java-debug-tool没有断点；
对象打印	如果类没有覆盖toString方法则不能很好的获取到对象的信息；	当检测到类没有覆盖toString方法之后，对象会被打印成json；	
命令特点	命令多，每个命令各司其职	命令较少，mt命令相当于聚合了多个greys的命令，调试信息较为丰富；	
网络连接管理	一个客户端连接idle一段时间会被关闭	一个客户端连接idle一段时间会被关闭，可配置服务端是否在所有连接都关闭的情况下关闭服务；	
语言	Java	Java	

本文详细剖析了 Java 动态调试关键技术的实现细节，并介绍了我们基于 Java 动态调试技术结合实际故障排查场景进行的一点探索实践；动态调试技术为研发人员进行线上问题排查提供了一种新的思路，我们基于动态调试技术解决了传统断点调试存在的问题，使得可以将断点调试这种技术应用在线上，以线下调试的思维来进行线上调试，提高问题排查效率。

参考文献

- ASM 4 guide
- Java Virtual Machine Specification
- JVM Tool Interface
- alibaba arthas
- openjdk

作者简介

胡健，美团到店餐饮研发中心研发工程师。

招聘

美团到店餐饮研发中心支撑了美团核心业务 – 餐饮团购业务。团队核心成员并肩作战与美团一起打赢了千团大战，实力爆表，更有来自知名互联网公司的各路大牛加盟支持。团队正在经历从优秀到卓越的蜕变，挑战多，发展空间大。期待优秀的你加入我们，共同努力，让所有人吃的更好、生活更好！ 美团到店餐饮研发中心诚聘 Java 高级工程师、架构师、专家，欢迎有兴趣的同学投递简历到 tech@meituan.com（邮件标题注明：美团到店餐饮研发中心）

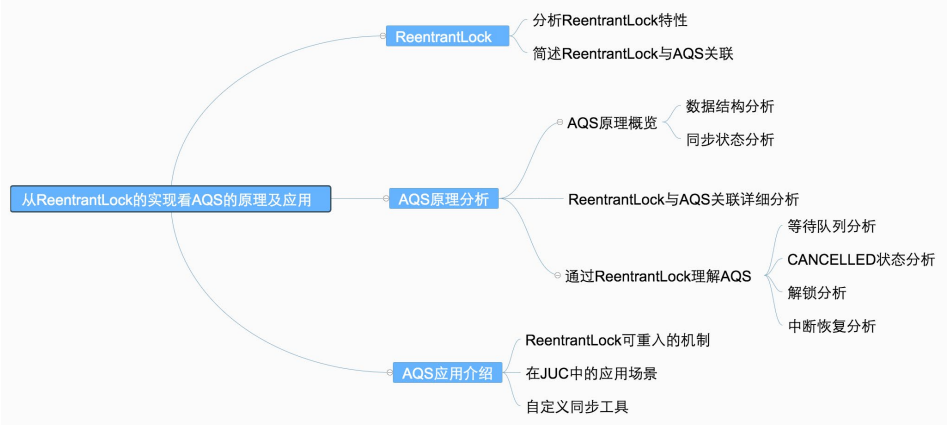
从 ReentrantLock 的实现看 AQS 的原理及应用

李卓

前言

Java 中的大部分同步类（Lock、Semaphore、ReentrantLock 等）都是基于 AbstractQueuedSynchronizer（简称为 AQS）实现的。AQS 是一种提供了原子式管理同步状态、阻塞和唤醒线程功能以及队列模型的简单框架。本文会从应用层逐渐深入到原理层，并通过 ReentrantLock 的基本特性和 ReentrantLock 与 AQS 的关联，来深入解读 AQS 相关独占锁的知识点，同时采取问答的模式来帮助大家理解 AQS。由于篇幅原因，本篇文章主要阐述 AQS 中独占锁的逻辑和 Sync Queue，不讲述包含共享锁和 Condition Queue 的部分（本篇文章核心为 AQS 原理剖析，只是简单介绍了 ReentrantLock，感兴趣同学可以阅读一下 ReentrantLock 的源码）。

下面列出本篇文章的大纲和思路，以便于大家更好地理解：



1. ReentrantLock

1.1 ReentrantLock 特性概览

ReentrantLock 意思为可重入锁，指的是一个线程能够对一个临界资源重复加锁。为了帮助大家更好地理解 ReentrantLock 的特性，我们先将 ReentrantLock 跟常用的 Synchronized 进行比较，其特性如下（蓝色部分为本篇文章主要剖析的点）：

	ReentrantLock	Synchronized
锁实现机制	依赖AQS	监视器模式
灵活性	支持响应中断、超时、尝试获取锁	不灵活
释放形式	必须显示调用unlock()释放锁	自动释放监视器
锁类型	公平锁&非公平锁	非公平锁
条件队列	可关联多个条件队列	关联一个条件队列
可重入性	可重入	可重入

下面通过伪代码，进行更加直观的比较：

```
// **************************Synchronized 的使用方式
**************************
// 1. 用于代码块
synchronized (this) {}
// 2. 用于对象
synchronized (object) {}
// 3. 用于方法
public synchronized void test () {}
// 4. 可重入
for (int i = 0; i < 100; i++) {
      synchronized (this) {}
}
// **************************ReentrantLock 的使用方式
**************************
```

```
public void test () throw Exception {
      // 1. 初始化选择公平锁、非公平锁
      ReentrantLock lock = new ReentrantLock(true);
      // 2. 可用于代码块
      lock.lock();
      try {
            try {
                  // 3. 支持多种加锁方式，比较灵活 ; 具有可重入特性
                  if(lock.tryLock(100, TimeUnit.MILLISECONDS)){ }
            } finally {
                  // 4. 手动释放锁
                  lock.unlock()
            }
      } finally {
            lock.unlock();
      }
}
```

1.2 ReentrantLock 与 AQS 的关联

通过上文我们已经了解，ReentrantLock 支持公平锁和非公平锁（关于公平锁和非公平锁的原理分析，可参考《不可不说的 Java“锁”事》），并且 ReentrantLock 的底层就是由 AQS 来实现的。那么 ReentrantLock 是如何通过公平锁和非公平锁与 AQS 关联起来呢？ 我们着重从这两者的加锁过程来理解一下它们与 AQS 之间的关系（加锁过程中与 AQS 的关联比较明显，解锁流程后续会介绍）。

非公平锁源码中的加锁流程如下：

```
// java.util.concurrent.locks.ReentrantLock#NonfairSync

// 非公平锁
static final class NonfairSync extends Sync {
      ...
      final void lock() {
            if (compareAndSetState(0, 1))
                  setExclusiveOwnerThread(Thread.currentThread());
            else
                  acquire(1);
            }
  ...
}
```

这块代码的含义为：

- 若通过 CAS 设置变量 State（同步状态）成功，也就是获取锁成功，则将当前线程设置为独占线程。
- 若通过 CAS 设置变量 State（同步状态）失败，也就是获取锁失败，则进入 Acquire 方法进行后续处理。

第一步很好理解，但第二步获取锁失败后，后续的处理策略是怎么样的呢？这块可能会有以下思考：

- 某个线程获取锁失败的后续流程是什么呢？有以下两种可能：

（1）将当前线程获锁结果设置为失败，获取锁流程结束。这种设计会极大降低系统的并发度，并不满足我们实际的需求。所以就需要下面这种流程，也就是 AQS 框架的处理流程。

（2）存在某种排队等候机制，线程继续等待，仍然保留获取锁的可能，获取锁流程仍在继续。

- 对于问题 1 的第二种情况，既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢？
- 处于排队等候机制中的线程，什么时候可以有机会获取锁呢？
- 如果处于排队等候机制中的线程一直无法获取锁，还是需要一直等待吗，还是有别的策略来解决这一问题？

带着非公平锁的这些问题，再看下公平锁源码中获锁的方式：

```
// java.util.concurrent.locks.ReentrantLock#FairSync

static final class FairSync extends Sync {
  ...
      final void lock() {
              acquire(1);
      }
  ...
}
```

看到这块代码，我们可能会存在这种疑问：Lock 函数通过 Acquire 方法进行加

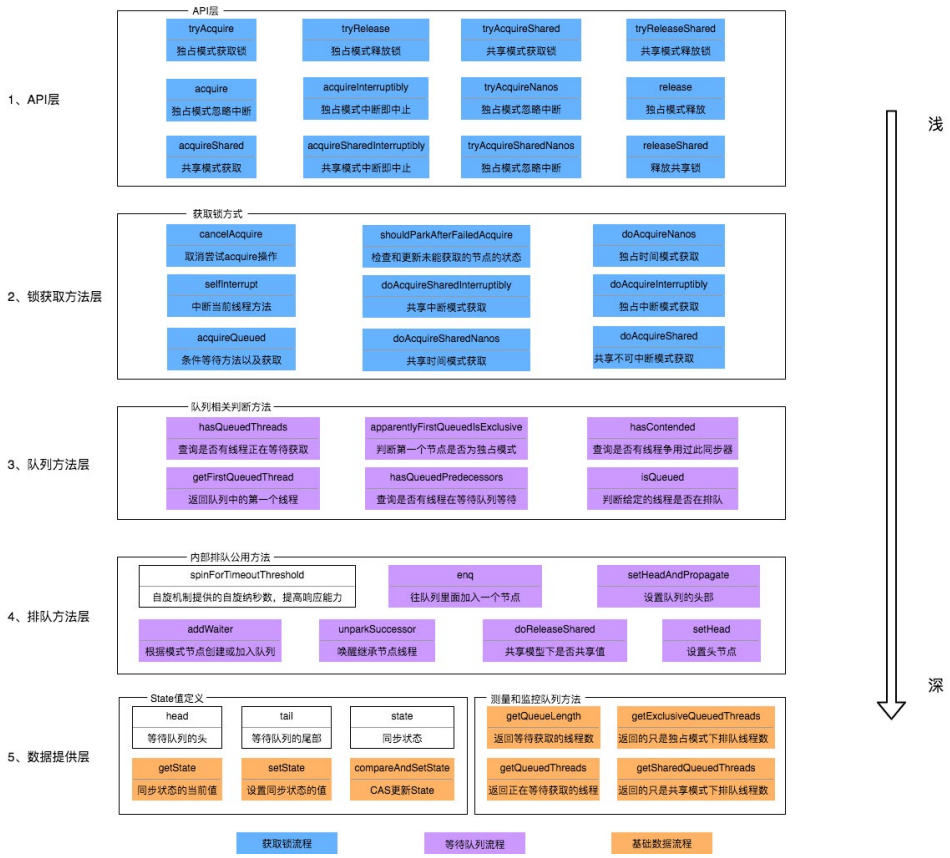
锁，但是具体是如何加锁的呢?

结合公平锁和非公平锁的加锁流程，虽然流程上有一定的不同，但是都调用了 Acquire 方法，而 Acquire 方法是 FairSync 和 UnfairSync 的父类 AQS 中的核心方法。

对于上边提到的问题，其实在 ReentrantLock 类源码中都无法解答，而这些问题的答案，都是位于 Acquire 方法所在的类 AbstractQueuedSynchronizer 中，也就是本文的核心——AQS。下面我们会对 AQS 以及 ReentrantLock 和 AQS 的关联做详细介绍（相关问题答案会在 2.3.5 小节中解答）。

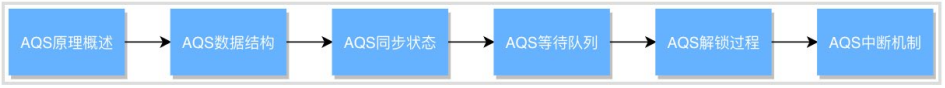
2. AQS

首先，我们通过下面的架构图来整体了解一下 AQS 框架：



- 上图中有颜色的为 Method，无颜色的为 Attribution。
- 总的来说，AQS 框架共分为五层，自上而下由浅入深，从 AQS 对外暴露的 API 到底层基础数据。
- 当有自定义同步器接入时，只需重写第一层所需要的部分方法即可，不需要关注底层具体的实现流程。当自定义同步器进行加锁或者解锁操作时，先经过第一层的 API 进入 AQS 内部方法，然后经过第二层进行锁的获取，接着对于获取锁失败的流程，进入第三层和第四层的等待队列处理，而这些处理方式均依赖于第五层的基础数据提供层。

下面我们会从整体到细节，从流程到方法逐一剖析 AQS 框架，主要分析过程如下：

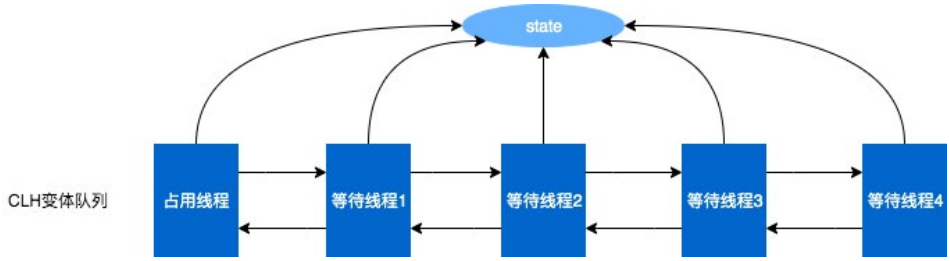


2.1 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，那么就将当前请求资源的线程设置为有效的工作线程，将共享资源设置为锁定状态；如果共享资源被占用，就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是 CLH 队列的变体实现的，将暂时获取不到锁的线程加入到队列中。

CLH：Craig、Landin and Hagersten 队列，是单向链表，AQS 中的队列是 CLH 变体的虚拟双向队列（FIFO），AQS 是通过将每条请求共享资源的线程封装成一个节点来实现锁的分配。

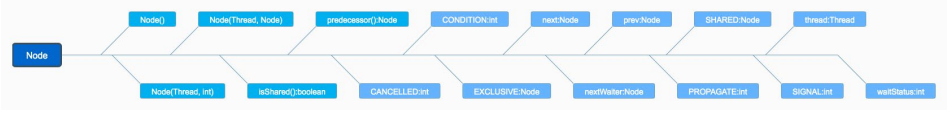
主要原理图如下：



AQS 使用一个 Volatile 的 int 类型的成员变量来表示同步状态，通过内置的 FIFO 队列来完成资源获取的排队工作，通过 CAS 完成对 State 值的修改。

2.1.1 AQS 数据结构

先来看下 AQS 中最基本的数据结构——Node，Node 即为上面 CLH 变体队列中的节点。



解释一下几个方法和属性值的含义：

方法和属性值	含义
waitStatus	当前节点在队列中的状态
thread	表示处于该节点的线程
prev	前驱指针
predecessor	返回前驱节点，没有的话抛出 npe
nextWaiter	指向下一个处于 CONDITION 状态的节点（由于本篇文章不讲述 Condition Queue 队列，这个指针不多介绍）
next	后继指针

线程两种锁的模式：

模式	含义
SHARED	表示线程以共享的模式等待锁
EXCLUSIVE	表示线程正在以独占的方式等待锁

waitStatus 有下面几个枚举值：

枚举	含义
0	当一个 Node 被初始化的时候的默认值
CANCELLED	为 1，表示线程获取锁的请求已经取消了
CONDITION	为 -2，表示节点在等待队列中，节点线程等待唤醒
PROPAGATE	为 -3，当前线程处在 SHARED 情况下，该字段才会使用
SIGNAL	为 -1，表示线程已经准备好了，就等资源释放了

2.1.2 同步状态 State

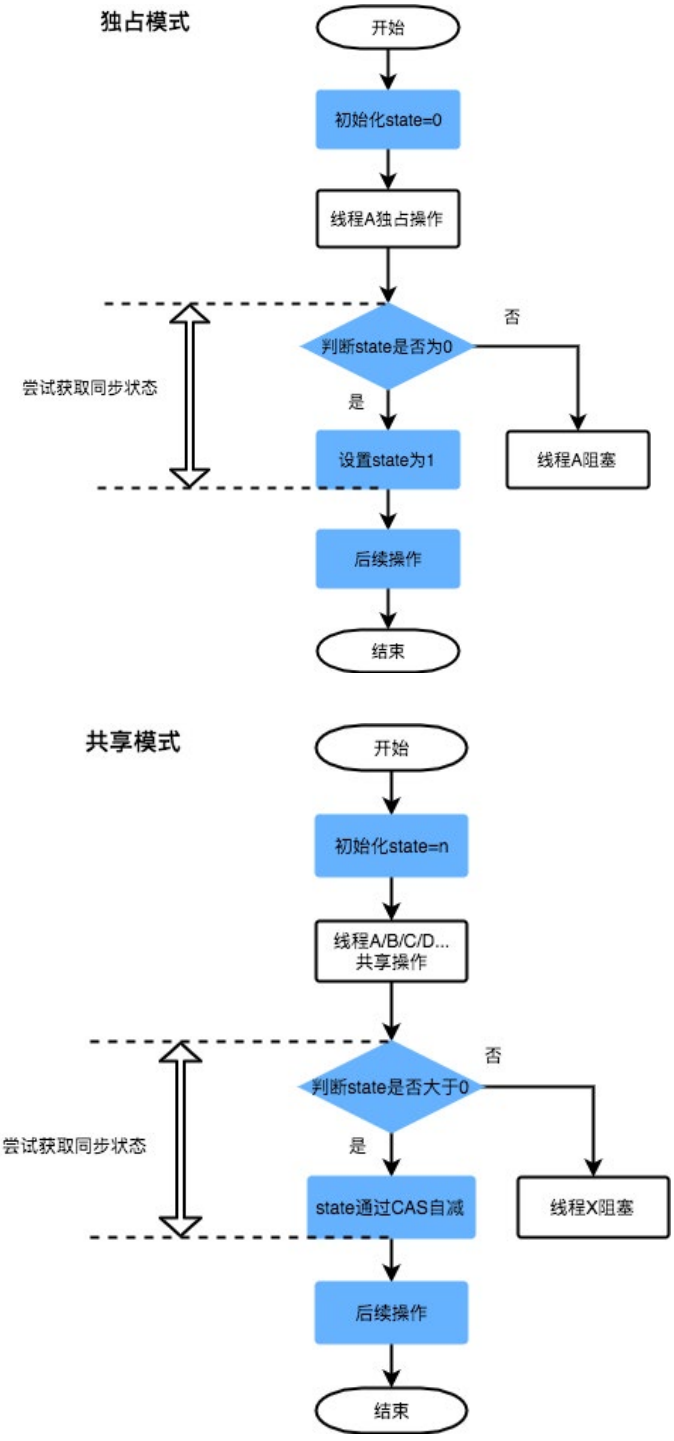
在了解数据结构后，接下来了解一下 AQS 的同步状态——State。AQS 中维护了一个名为 state 的字段，意为同步状态，是由 Volatile 修饰的，用于展示当前临界资源的获锁情况。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private volatile int state;
```

下面提供了几个访问这个字段的方法：

方法名	描述
protected final int getState()	获取 State 的值
protected final void setState(int newState)	设置 State 的值
protected final boolean compareAndSetState(int expect, int update)	使用 CAS 方式更新 State

这几个方法都是 Final 修饰的，说明子类中无法重写它们。我们可以通过修改 State 字段表示的同步状态来实现多线程的独占模式和共享模式（加锁过程）。


独占模式
开始
初始化state=0
线程A独占操作
判断state是否为0
否
是
尝试获取同步状态
设置state为1
线程A阻塞
后续操作
结束
共享模式
开始
初始化state=n
线程A/B/C/D...
共享操作
判断state是否大于0
否
是
尝试获取同步状态
state通过CAS自减
线程X阻塞
后续操作
结束

对于我们自定义的同步工具，需要自定义获取同步状态和释放状态的方式，也就是 AQS 架构图中的第一层：API 层。

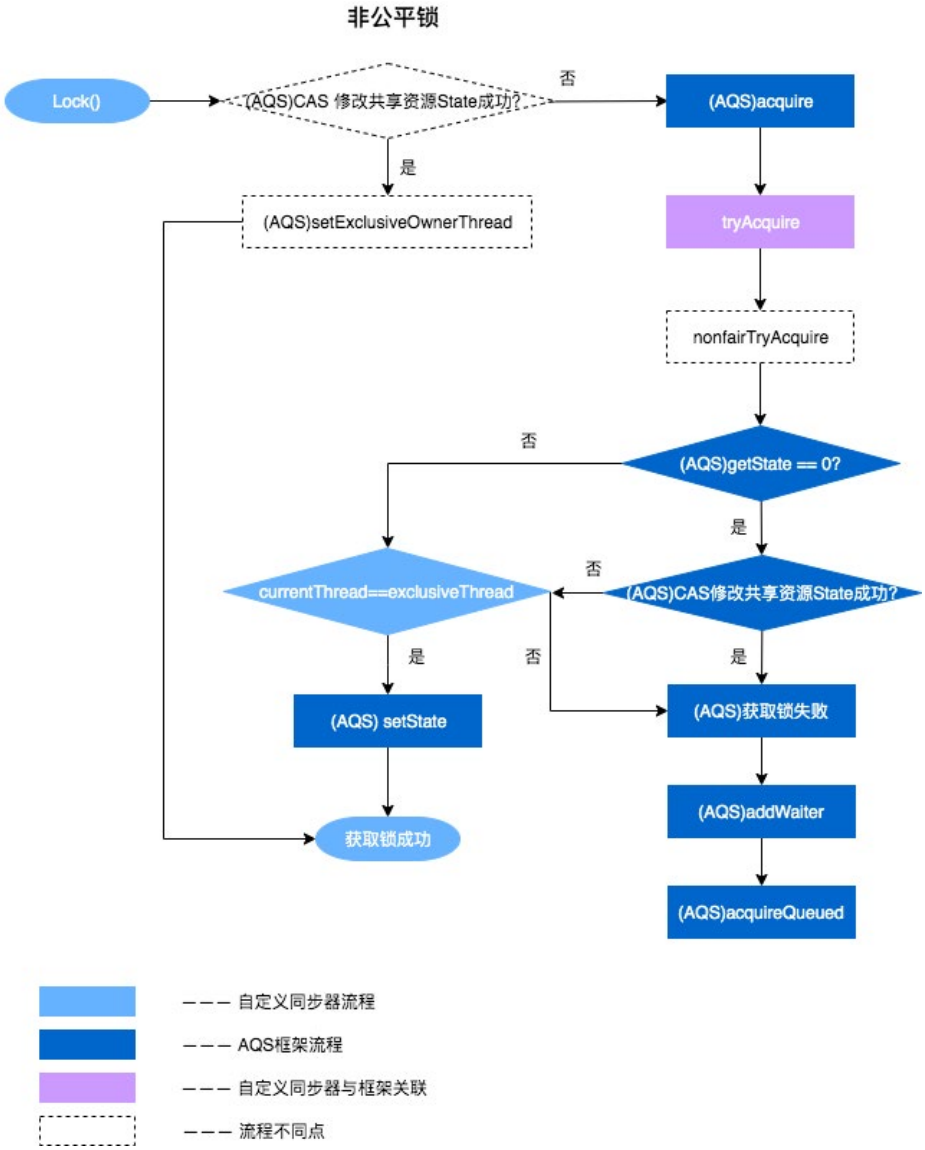
2.2 AQS 重要方法与 ReentrantLock 的关联

从架构图中可以得知，AQS 提供了大量用于自定义同步器实现的 Protected 方法。自定义同步器实现的相关方法也只是为了通过修改 State 字段来实现多线程的独占模式或者共享模式。自定义同步器需要实现以下方法（ReentrantLock 需要实现的方法如下，并不是全部）：

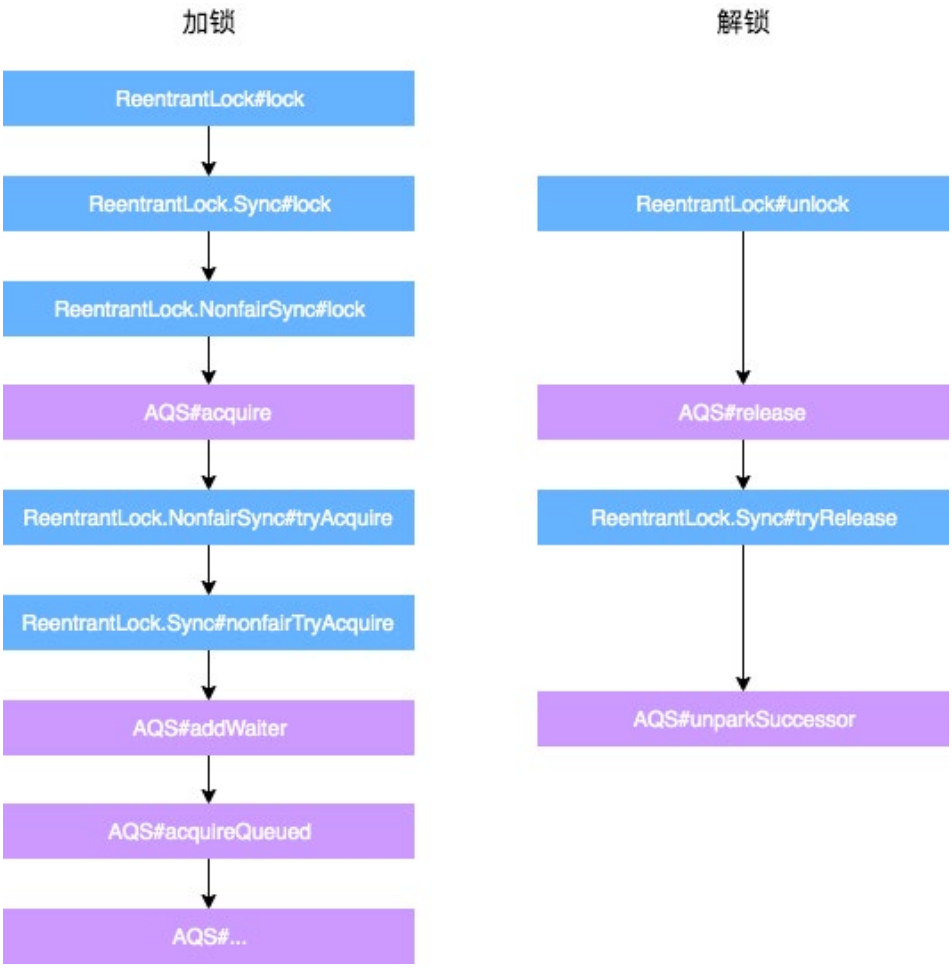
方法名	描述
protected boolean isHeldExclusively()	该线程是否正在独占资源。只有用到 Condition 才需要去实现它。
protected boolean tryAcquire(int arg)	独占方式。arg 为获取锁的次数，尝试获取资源，成功则返回 True，失败则返回 False。
protected boolean tryRelease(int arg)	独占方式。arg 为释放锁的次数，尝试释放资源，成功则返回 True，失败则返回 False。
protected int tryAcquireShared(int arg)	共享方式。arg 为获取锁的次数，尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
protected boolean tryReleaseShared(int arg)	共享方式。arg 为释放锁的次数，尝试释放资源，如果释放后允许唤醒后续等待结点返回 True，否则返回 False。

一般来说，自定义同步器要么是独占方式，要么是共享方式，它们也只需实现 tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock。ReentrantLock 是独占锁，所以实现了 tryAcquire-tryRelease。

以非公平锁为例，这里主要阐述一下非公平锁与 AQS 之间方法的关联之处，具体每一处核心方法的作用会在文章后面详细进行阐述。



为了帮助大家理解 ReentrantLock 和 AQS 之间方法的交互过程，以非公平锁为例，我们将加锁和解锁的交互流程单独拎出来强调一下，以便于对后续内容的理解。



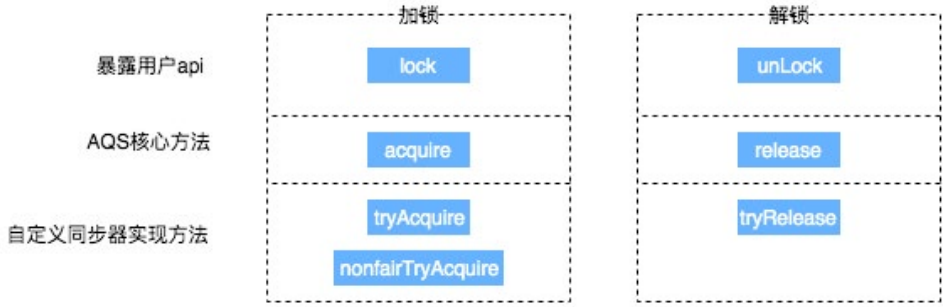
加锁：

- 通过 ReentrantLock 的加锁方法 Lock 进行加锁操作。
- 会调用到内部类 Sync 的 Lock 方法，由于 Sync#lock 是抽象方法，根据 ReentrantLock 初始化选择的公平锁和非公平锁，执行相关内部类的 Lock 方法，本质上都会执行 AQS 的 Acquire 方法。
- AQS 的 Acquire 方法会执行 tryAcquire 方法，但是由于 tryAcquire 需要自定义同步器实现，因此执行了 ReentrantLock 中的 tryAcquire 方法，由于 ReentrantLock 是通过公平锁和非公平锁内部类实现的 tryAcquire 方法，因

此会根据锁类型不同，执行不同的 tryAcquire。

- tryAcquire 是获取锁逻辑，获取失败后，会执行框架 AQS 的后续逻辑，跟 ReentrantLock 自定义同步器无关。
- 解锁：
- 通过 ReentrantLock 的解锁方法 Unlock 进行解锁。
- Unlock 会调用内部类 Sync 的 Release 方法，该方法继承于 AQS。
- Release 中会调用 tryRelease 方法，tryRelease 需要自定义同步器实现，tryRelease 只在 ReentrantLock 中的 Sync 实现，因此可以看出，释放锁的过程，并不区分是否为公平锁。
- 释放成功后，所有处理由 AQS 框架完成，与自定义同步器无关。

通过上面的描述，大概可以总结出 ReentrantLock 加锁解锁时 API 层核心方法的映射关系。



2.3 通过 ReentrantLock 理解 AQS

ReentrantLock 中公平锁和非公平锁在底层是相同的，这里以非公平锁为例进行分析。

在非公平锁中，有一段这样的代码：

```
// java.util.concurrent.locks.ReentrantLock

static final class NonfairSync extends Sync {
      ...
      final void lock() {
```

```
            if (compareAndSetState(0, 1))
                    setExclusiveOwnerThread(Thread.currentThread());
            else
                    acquire(1);
    }
  ...
}
```

看一下这个 Acquire 是怎么写的：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final void acquire(int arg) {
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
}
```

再看一下 tryAcquire 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
```

可以看出，这里只是 AQS 的简单实现，具体获取锁的实现方法是由各自的公平锁和非公平锁单独实现的（以 ReentrantLock 为例）。如果该方法返回了 True，则说明当前线程获取锁成功，就不用往后执行了；如果获取失败，就需要加入到等待队列中。下面会详细解释线程是何时以及怎样被加入进等待队列中的。

2.3.1 线程加入等待队列

2.3.1.1 加入队列的时机

当执行 Acquire(1) 时，会通过 tryAcquire 获取锁。在这种情况下，如果获取锁失败，就会调用 addWaiter 加入到等待队列中去。

2.3.1.2 如何加入队列

获取锁失败后，会执行 addWaiter(Node.EXCLUSIVE) 加入等待队列，具体实现方法如下：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node addWaiter(Node mode) {
      Node node = new Node(Thread.currentThread(), mode);
      // Try the fast path of enq; backup to full enq on failure
      Node pred = tail;
      if (pred != null) {
              node.prev = pred;
              if (compareAndSetTail(pred, node)) {
                      pred.next = node;
                      return node;
              }
      }
      enq(node);
      return node;
}
private final boolean compareAndSetTail(Node expect, Node update) {
      return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
}
```

主要的流程如下：

- 通过当前的线程和锁模式新建一个节点。
- Pred 指针指向尾节点 Tail。
- 将 New 中 Node 的 Prev 指针指向 Pred。
- 通过 compareAndSetTail 方法，完成尾节点的设置。这个方法主要是对 tailOffset 和 Expect 进行比较，如果 tailOffset 的 Node 和 Expect 的 Node 地址是相同的，那么设置 Tail 的值为 Update 的值。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

static {
      try {
              stateOffset = unsafe.objectFieldOffset(AbstractQueuedSynchronizer.
class.getDeclaredField("state"));
              headOffset = unsafe.
objectFieldOffset(AbstractQueuedSynchronizer.class.
getDeclaredField("head"));
              tailOffset = unsafe.
objectFieldOffset(AbstractQueuedSynchronizer.class.
getDeclaredField("tail"));
              waitStatusOffset = unsafe.objectFieldOffset(Node.class.
getDeclaredField("waitStatus"));
```

```
            nextOffset = unsafe.objectFieldOffset(Node.class.
getDeclaredField("next"));
      } catch (Exception ex) {
    throw new Error(ex);
  }
}
```

从 AQS 的静态代码块可以看出，都是获取一个对象的属性相对于该对象在内存当中的偏移量，这样我们就可以根据这个偏移量在对象内存当中找到这个属性。tailOffset 指的是 tail 对应的偏移量，所以这个时候会将 new 出来的 Node 置为当前队列的尾节点。同时，由于是双向链表，也需要将前一个节点指向尾节点。

- 如果 Pred 指针是 Null（说明等待队列中没有元素），或者当前 Pred 指针和 Tail 指向的位置不同（说明被别的线程已经修改），就需要看一下 Enq 的方法。

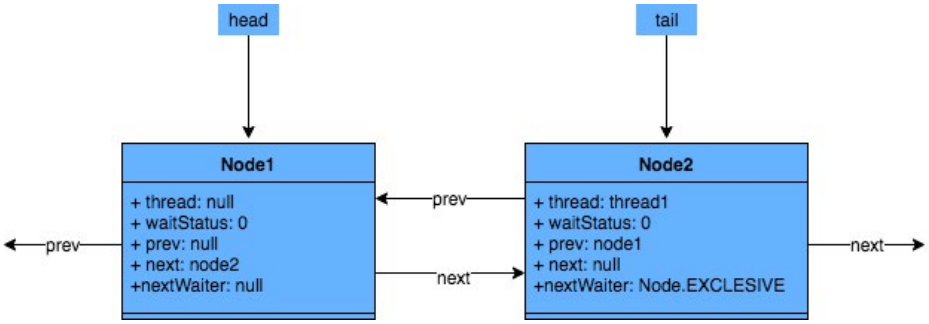
```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node enq(final Node node) {
      for (;;) {
            Node t = tail;
            if (t == null) { // Must initialize
                  if (compareAndSetHead(new Node()))
                        tail = head;
            } else {
                  node.prev = t;
                  if (compareAndSetTail(t, node)) {
                        t.next = node;
                        return t;
                  }
            }
      }
}
```

如果没有被初始化，需要进行初始化一个头结点出来。但请注意，初始化的头结点并不是当前线程节点，而是调用了无参构造函数的节点。如果经历了初始化或者并发导致队列中有元素，则与之前的方法相同。其实，addWaiter 就是一个在双端链表添加尾节点的操作，需要注意的是，双端链表的头结点是一个无参构造函数的头结点。

总结一下，线程获取锁的时候，过程大体如下：

1. 当没有线程获取到锁时，线程 1 获取锁成功。

2. 线程 2 申请锁，但是锁被线程 1 占有。



3. 如果再有线程要获取锁，依次在队列中往后排队即可。

回到上边的代码，hasQueuedPredecessors 是公平锁加锁时判断等待队列中是否存在有效节点的方法。如果返回 False，说明当前线程可以争取共享资源；如果返回 True，说明队列中存在有效节点，当前线程必须加入到等待队列中。

```
// java.util.concurrent.locks.ReentrantLock

public final boolean hasQueuedPredecessors() {
      // The correctness of this depends on head being initialized
      // before tail and on head.next being accurate if the current
      // thread is first in queue.
      Node t = tail; // Read fields in reverse initialization order
      Node h = head;
      Node s;
      return h != t && ((s = h.next) == null || s.thread != Thread.
currentThread());
}
```

看到这里，我们理解一下 h != t && ((s = h.next) == null || s.thread != Thread.currentThread()); 为什么要判断的头结点的下一个节点？第一个节点储存的数据是什么？

双向链表中，第一个节点为虚节点，其实并不存储任何信息，只是占位。真正的第一个有数据的节点，是在第二个节点开始的。当 h != t 时：如果 (s =

h.next) == null，等待队列正在有线程进行初始化，但只是进行到了 Tail 指向 Head，没有将 Head 指向 Tail，此时队列中有元素，需要返回 True（这块具体见下边代码分析）。 如果 (s = h.next) != null，说明此时队列中至少有一个有效节点。如果此时 s.thread == Thread.currentThread()，说明等待队列的第一个有效节点中的线程与当前线程相同，那么当前线程是可以获取资源的；如果 s.thread != Thread.currentThread()，说明等待队列的第一个有效节点线程与当前线程不同，当前线程必须加入进等待队列。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer#enq

if (t == null) { // Must initialize
      if (compareAndSetHead(new Node()))
              tail = head;
} else {
      node.prev = t;
      if (compareAndSetTail(t, node)) {
              t.next = node;
              return t;
      }
}
```

节点入队不是原子操作，所以会出现短暂的 head != tail，此时 Tail 指向最后一个节点，而且 Tail 指向 Head。如果 Head 没有指向 Tail（可见 5、6、7 行），这种情况下也需要将相关线程加入队列中。所以这块代码是为了解决极端情况下的并发问题。

2.3.1.3 等待队列中线程出队列时机

回到最初的源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final void acquire(int arg) {
      if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
              selfInterrupt();
}
```

上文解释了 addWaiter 方法，这个方法其实就是把对应的线程以 Node 的数据结构形式加入到双端队列里，返回的是一个包含该线程的 Node。而这个 Node 会作

为参数，进入到 acquireQueued 方法中。acquireQueued 方法可以对排队中的线程进行“获锁”操作。

总的来说，一个线程获取锁失败了，被放入等待队列，acquireQueued 会把放入队列中的线程不断去获取锁，直到获取成功或者不再需要获取（中断）。

下面我们从“何时出队列?”和“如何出队列?”两个方向来分析一下 acquireQueued 源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      // 标记是否成功拿到资源
      boolean failed = true;
      try {
              // 标记等待过程中是否中断过
              boolean interrupted = false;
              // 开始自旋，要么获取锁，要么中断
              for (;;) {
                      // 获取当前节点的前驱节点
                      final Node p = node.predecessor();
                      // 如果 p 是头结点，说明当前节点在真实数据队列的首部，就尝试获取锁（别忘了头结点是虚节点）
                      if (p == head && tryAcquire(arg)) {
                              // 获取锁成功，头指针移动到当前 node
                              setHead(node);
                              p.next = null; // help GC
                              failed = false;
                              return interrupted;
                      }
                      // 说明 p 为头节点且当前没有获取到锁（可能是非公平锁被抢占了）或者是 p 不为头结点，这个时候就要判断当前 node 是否要被阻塞（被阻塞条件：前驱节点的 waitStatus 为 -1），防止无限循环浪费资源。具体两个方法下面细细分析
                      if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
                              interrupted = true;
              }
      } finally {
              if (failed)
                      cancelAcquire(node);
      }
}
```

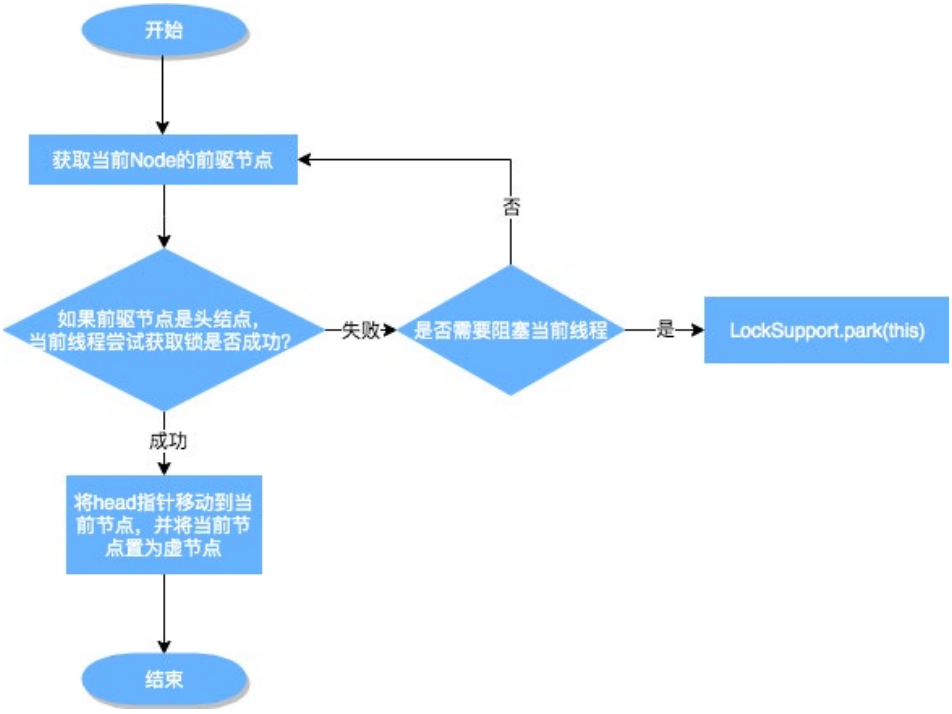
注：setHead 方法是把当前节点置为虚节点，但并没有修改 waitStatus，因为它是一直需要用的数据。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void setHead(Node node) {
      head = node;
      node.thread = null;
      node.prev = null;
}

// java.util.concurrent.locks.AbstractQueuedSynchronizer

// 靠前驱节点判断当前线程是否应该被阻塞
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
      // 获取头结点的节点状态
      int ws = pred.waitStatus;
      // 说明头结点处于唤醒状态
      if (ws == Node.SIGNAL)
              return true;
      // 通过枚举值我们知道 waitStatus>0 是取消状态
      if (ws > 0) {
              do {
                      // 循环向前查找取消节点，把取消节点从队列中剔除
                      node.prev = pred = pred.prev;
              } while (pred.waitStatus > 0);
              pred.next = node;
      } else {
              // 设置前任节点等待状态为 SIGNAL
              compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
      }
      return false;
}
```

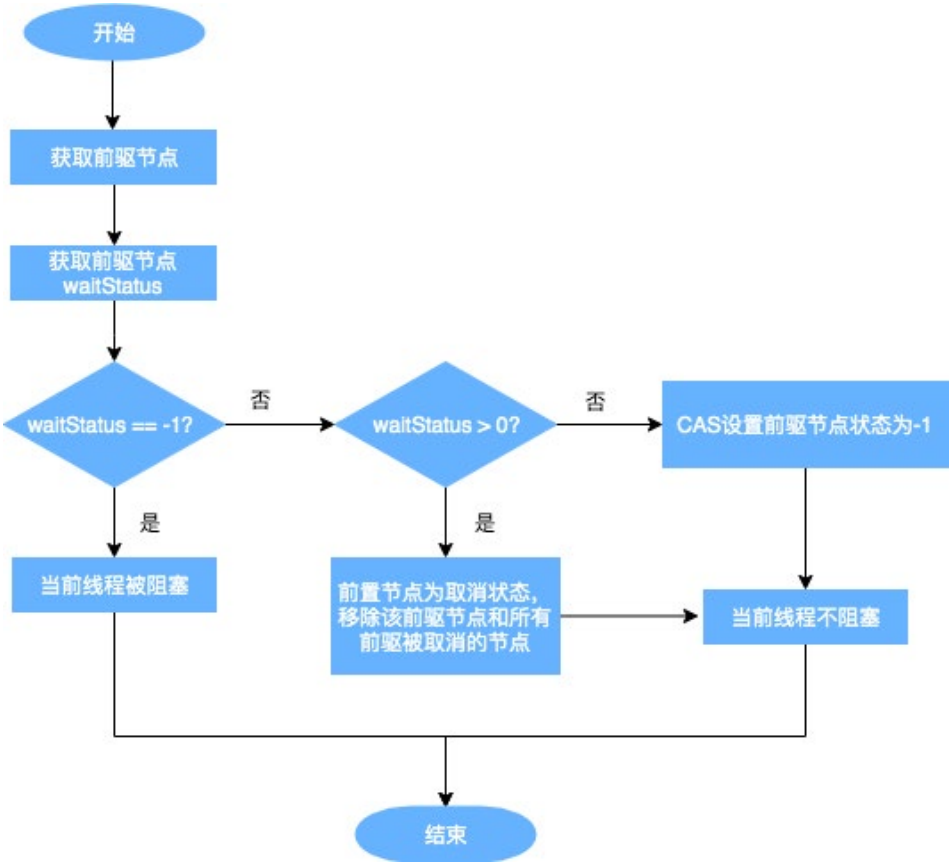
parkAndCheckInterrupt 主要用于挂起当前线程，阻塞调用栈，返回当前线程的中断状态。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

上述方法的流程图如下：



从上图可以看出，跳出当前循环的条件是当“前置节点是头结点，且当前线程获取锁成功”。为了防止因死循环导致 CPU 资源被浪费，我们会判断前置节点的状态来决定是否要将当前线程挂起，具体挂起流程用流程图表示如下（shouldParkAfter-FailedAcquire 流程）：



从队列中释放节点的疑虑打消了，那么又有新问题了：

- shouldParkAfterFailedAcquire 中取消节点是怎么生成的呢？什么时候会把一个节点的 waitStatus 设置为 -1？
- 是在什么时间释放节点通知到被挂起的线程呢？

2.3.2 CANCELLED 状态节点生成

acquireQueued 方法中的 Finally 代码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      boolean failed = true;
      try {
```

```
    ...
            for (;;) {
                    final Node p = node.predecessor();
                    if (p == head && tryAcquire(arg)) {
                            ...
                            failed = false;
        ...
                    }
                    ...
      } finally {
            if (failed)
                    cancelAcquire(node);
            }
}
```

通过 cancelAcquire 方法，将 Node 的状态标记为 CANCELLED。接下来，我们逐行来分析这个方法的原理：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void cancelAcquire(Node node) {
  // 将无效节点过滤
      if (node == null)
              return;
  // 设置该节点不关联任何线程，也就是虚节点
      node.thread = null;
      Node pred = node.prev;
  // 通过前驱节点，跳过取消状态的 node
      while (pred.waitStatus > 0)
              node.prev = pred = pred.prev;
  // 获取过滤后的前驱节点的后继节点
      Node predNext = pred.next;
  // 把当前 node 的状态设置为 CANCELLED
      node.waitStatus = Node.CANCELLED;
  // 如果当前节点是尾节点，将从后往前的第一个非取消状态的节点设置为尾节点
  // 更新失败的话，则进入 else，如果更新成功，将 tail 的后继节点设置为 null
      if (node == tail && compareAndSetTail(node, pred)) {
              compareAndSetNext(pred, predNext, null);
      } else {
              int ws;
    // 如果当前节点不是 head 的后继节点，1：判断当前节点前驱节点的是否为 SIGNAL，
2：如果不是，则把前驱节点设置为 SINGAL 看是否成功
    // 如果 1 和 2 中有一个为 true，再判断当前节点的线程是否为 null
    // 如果上述条件都满足，把当前节点的前驱节点的后继指针指向当前节点的后继节点
              if (pred != head && ((ws = pred.waitStatus) == Node.SIGNAL
|| (ws <= 0 &&
compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) && pred.thread != null)
```

```
{
                    Node next = node.next;
                    if (next != null && next.waitStatus <= 0)
                            compareAndSetNext(pred, predNext, next);
            } else {
    // 如果当前节点是 head 的后继节点，或者上述条件不满足，那就唤醒当前节点的后
继节点
                    unparkSuccessor(node);
            }
            node.next = node; // help GC
    }
}
```

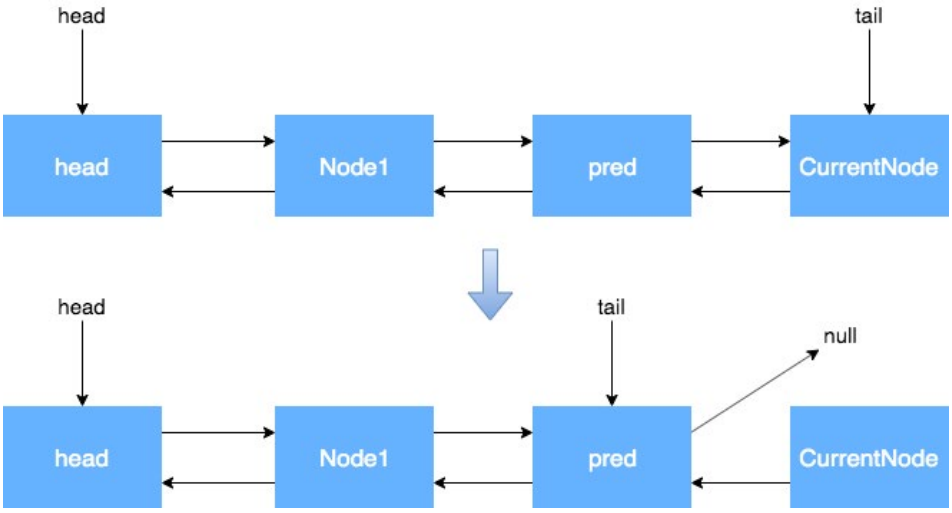
当前的流程：

- 获取当前节点的前驱节点，如果前驱节点的状态是 CANCELLED，那就一直往前遍历，找到第一个 waitStatus <= 0 的节点，将找到的 Pred 节点和当前 Node 关联，将当前 Node 设置为 CANCELLED。

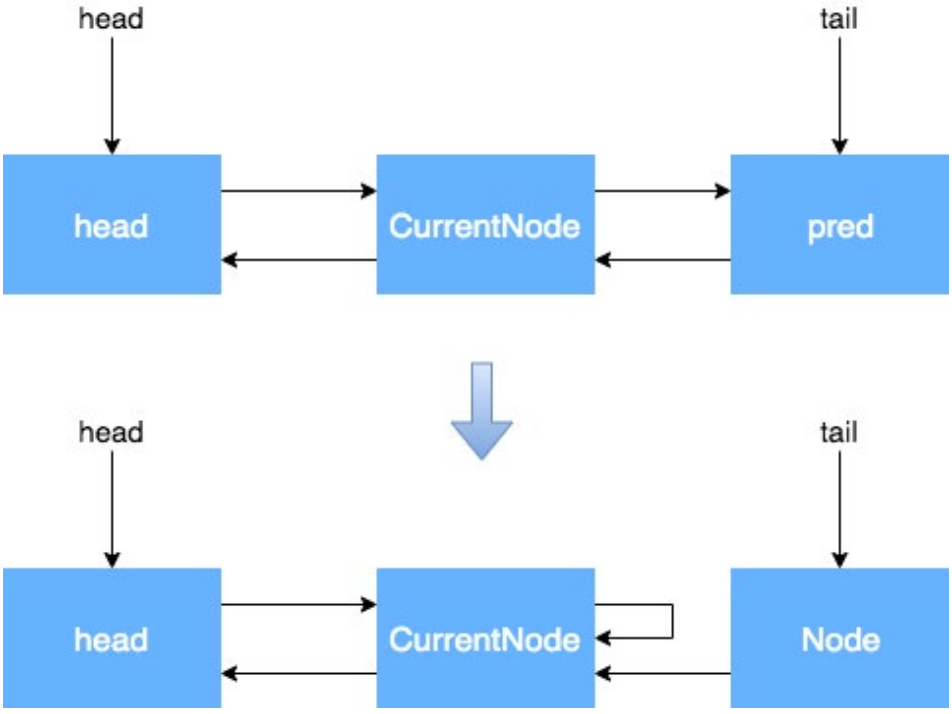
根据当前节点的位置，考虑以下三种情况：

（1）当前节点是尾节点。
（2）当前节点是 Head 的后继节点。
（3）当前节点不是 Head 的后继节点，也不是尾节点。

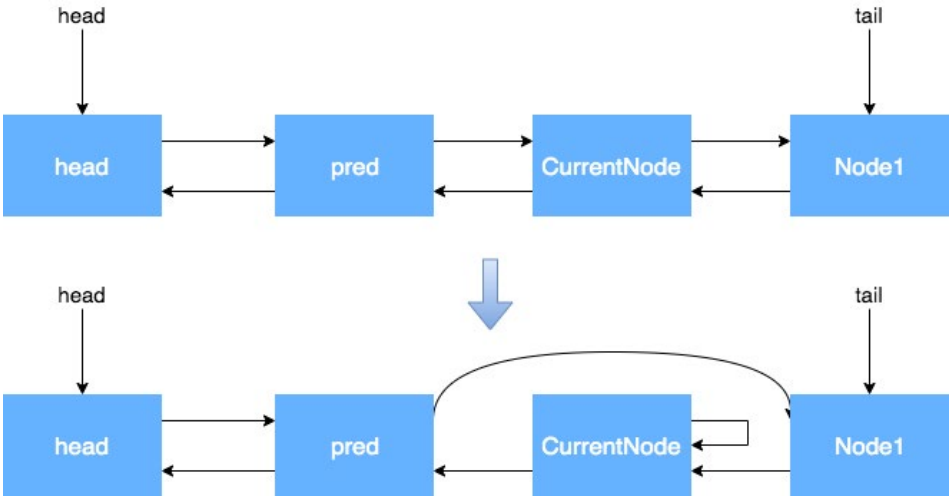
根据上述第二条，我们来分析每一种情况的流程。
当前节点是尾节点。



当前节点是 Head 的后继节点。



当前节点不是 Head 的后继节点，也不是尾节点。



通过上面的流程，我们对于 CANCELLED 节点状态的产生和变化已经有了大致的了解，但是为什么所有的变化都是对 Next 指针进行了操作，而没有对 Prev 指针进行操作呢？什么情况下会对 Prev 指针进行操作？

- 执行 cancelAcquire 的时候，当前节点的前置节点可能已经从队列中出去了（已经执行过 Try 代码块中的 shouldParkAfterFailedAcquire 方法了），如果此时修改 Prev 指针，有可能会导致 Prev 指向另一个已经移除队列的 Node，因此这块变化 Prev 指针不安全。 shouldParkAfterFailedAcquire 方法中，会执行下面的代码，其实就是在处理 Prev 指针。shouldParkAfterFailedAcquire 是获取锁失败的情况下才会执行，进入该方法后，说明共享资源已被获取，当前节点之前的节点都不会出现变化，因此这个时候变更 Prev 指针比较安全。

```
do {
      node.prev = pred = pred.prev;
} while (pred.waitStatus > 0);
```

2.3.3 如何解锁

我们已经剖析了加锁过程中的基本流程，接下来再对解锁的基本流程进行分析。

由于 ReentrantLock 在解锁的时候，并不区分公平锁和非公平锁，所以我们直接看解锁的源码：

```
// java.util.concurrent.locks.ReentrantLock

public void unlock() {
      sync.release(1);
}
```

可以看到，本质释放锁的地方，是通过框架来完成的。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final boolean release(int arg) {
      if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                  unparkSuccessor(h);
            return true;
      }
      return false;
}
```

在 ReentrantLock 里面的公平锁和非公平锁的父类 Sync 定义了可重入锁的释放锁机制。

```
// java.util.concurrent.locks.ReentrantLock.Sync

// 方法返回当前锁是不是没有被线程持有
protected final boolean tryRelease(int releases) {
      // 减少可重入次数
      int c = getState() - releases;
      // 当前线程不是持有锁的线程，抛出异常
      if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
      boolean free = false;
      // 如果持有线程全部释放，将当前独占锁所有线程设置为 null，并更新 state
      if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
      }
      setState(c);
      return free;
}
```

我们来解释下述源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final boolean release(int arg) {
      // 上边自定义的 tryRelease 如果返回 true，说明该锁没有被任何线程持有
      if (tryRelease(arg)) {
              // 获取头结点
              Node h = head;
              // 头结点不为空并且头结点的 waitStatus 不是初始化节点情况，解除线
程挂起状态
              if (h != null && h.waitStatus != 0)
                      unparkSuccessor(h);
              return true;
      }
      return false;
}
```

这里的判断条件为什么是 h != null && h.waitStatus != 0？

h == null Head 还没初始化。初始情况下，head == null，第一个节点入队，Head 会被初始化一个虚拟节点。所以说，这里如果还没来得及入队，就会出现 head == null 的情况。

h != null && waitStatus == 0 表明后继节点对应的线程仍在运行中，不需要唤醒。

h != null && waitStatus < 0 表明后继节点可能被阻塞了，需要唤醒。

再看一下 unparkSuccessor 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void unparkSuccessor(Node node) {
      // 获取头结点 waitStatus
      int ws = node.waitStatus;
      if (ws < 0)
              compareAndSetWaitStatus(node, ws, 0);
      // 获取当前节点的下一个节点
      Node s = node.next;
      // 如果下个节点是 null 或者下个节点被 cancelled，就找到队列最开始的非
cancelled 的节点
      if (s == null || s.waitStatus > 0) {
              s = null;
              // 就从尾部节点开始找，到队首，找到队列第一个 waitStatus<0 的节点。
```

```
                for (Node t = tail; t != null && t != node; t = t.prev)
                        if (t.waitStatus <= 0)
                                s = t;
        }
        // 如果当前节点的下个节点不为空，而且状态 <=0，就把当前节点 unpark
        if (s != null)
                LockSupport.unpark(s.thread);
}
```

为什么要从后往前找第一个非 Cancelled 的节点呢？原因如下。

之前的 addWaiter 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        if (pred != null) {
                node.prev = pred;
                if (compareAndSetTail(pred, node)) {
                        pred.next = node;
                        return node;
                }
        }
        enq(node);
        return node;
}
```

我们从这里可以看到，节点入队并不是原子操作，也就是说，node.prev = pred; compareAndSetTail(pred, node) 这两个地方可以看作 Tail 入队的原子操作，但是此时 pred.next = node; 还没执行，如果这个时候执行了 unparkSuccessor 方法，就没办法从前往后找了，所以需要从后往前找。还有一点原因，在产生 CANCELLED 状态节点的时候，先断开的是 Next 指针，Prev 指针并未断开，因此也是必须要从后往前遍历才能够遍历完全部的 Node。

综上所述，如果是从前往后找，由于极端情况下入队的非原子操作和 CANCELLED 节点产生过程中断开 Next 指针的操作，可能会导致无法遍历所有的节点。所以，唤醒对应的线程后，对应的线程就会继续往下执行。继续执行

acquireQueued 方法以后，中断如何处理?

2.3.4 中断恢复后的执行流程

唤醒后，会执行 return Thread.interrupted();，这个函数返回的是当前执行线程的中断状态，并清除。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private final boolean parkAndCheckInterrupt() {
      LockSupport.park(this);
      return Thread.interrupted();
}
```

再回到 acquireQueued 代码，当 parkAndCheckInterrupt 返回 True 或者 False 的时候，interrupted 的值不同，但都会执行下次循环。如果这个时候获取锁成功，就会把当前 interrupted 返回。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      boolean failed = true;
      try {
            boolean interrupted = false;
            for (;;) {
                  final Node p = node.predecessor();
                  if (p == head && tryAcquire(arg)) {
                        setHead(node);
                        p.next = null; // help GC
                        failed = false;
                        return interrupted;
                  }
                  if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
                        interrupted = true;
                  }
      } finally {
            if (failed)
                  cancelAcquire(node);
      }
}
```

如果 acquireQueued 为 True，就会执行 selfInterrupt 方法。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

static void selfInterrupt() {
      Thread.currentThread().interrupt();
}
```

该方法其实是为了中断线程。但为什么获取了锁以后还要中断线程呢？这部分属于 Java 提供的协作式中断知识内容，感兴趣同学可以查阅一下。这里简单介绍一下：

1. 当中断线程被唤醒时，并不知道被唤醒的原因，可能是当前线程在等待中被中断，也可能是释放了锁以后被唤醒。因此我们通过 Thread.interrupted() 方法检查中断标记（该方法返回了当前线程的中断状态，并将当前线程的中断标识设置为 False），并记录下来，如果发现该线程被中断过，就再中断一次。
2. 线程在等待资源的过程中被唤醒，唤醒后还是会不断地去尝试获取锁，直到抢到锁为止。也就是说，在整个流程中，并不响应中断，只是记录中断记录。最后抢到锁返回了，那么如果被中断过的话，就需要补充一次中断。

这里的处理方式主要是运用线程池中基本运作单元 Worder 中的 runWorker，通过 Thread.interrupted() 进行额外的判断处理，感兴趣的同学可以看下 ThreadPoolExecutor 源码。

2.3.5 小结

我们在 1.3 小节中提出了一些问题，现在来回答一下。

Q：某个线程获取锁失败的后续流程是什么呢?

A：存在某种排队等候机制，线程继续等待，仍然保留获取锁的可能，获取锁流程仍在继续。

Q：既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢?

A：是 CLH 变体的 FIFO 双端队列。

Q：处于排队等候机制中的线程，什么时候可以有机会获取锁呢?

A：可以详细看下 2.3.1.3 小节。

Q：如果处于排队等候机制中的线程一直无法获取锁，需要一直等待么？还是有别的策略来解决这一问题?

A：线程所在节点的状态会变成取消状态，取消状态的节点会从队列中释放，具体可见 2.3.2 小节。

Q：Lock 函数通过 Acquire 方法进行加锁，但是具体是如何加锁的呢?

A：AQS 的 Acquire 会调用 tryAcquire 方法，tryAcquire 由各个自定义同步器实现，通过 tryAcquire 完成加锁过程。

3. AQS 应用

3.1 ReentrantLock 的可重入应用

ReentrantLock 的可重入性是 AQS 很好的应用之一，在了解完上述知识点以后，我们很容易得知 ReentrantLock 实现可重入的方法。在 ReentrantLock 里面，不管是公平锁还是非公平锁，都有一段逻辑。

公平锁：

```
// java.util.concurrent.locks.ReentrantLock.FairSync#tryAcquire

if (c == 0) {
      if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {
              setExclusiveOwnerThread(current);
              return true;
      }
}
else if (current == getExclusiveOwnerThread()) {
      int nextc = c + acquires;
      if (nextc < 0)
              throw new Error("Maximum lock count exceeded");
      setState(nextc);
      return true;
}
```

非公平锁：

```
// java.util.concurrent.locks.ReentrantLock.Sync#nonfairTryAcquire

if (c == 0) {
      if (compareAndSetState(0, acquires)){
```

```
                setExclusiveOwnerThread(current);
                return true;
        }
}
else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
}
```

从上面这两段都可以看到，有一个同步状态 State 来控制整体可重入的情况。State 是 Volatile 修饰的，用于保证一定的可见性和有序性。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private volatile int state;
```

接下来看 State 这个字段主要的过程：

1. State 初始化的时候为 0，表示没有任何线程持有锁。
2. 当有线程持有该锁时，值就会在原来的基础上 +1，同一个线程多次获得锁是，就会多次 +1，这里就是可重入的概念。
3. 解锁也是对这个字段 -1，一直到 0，此线程对锁释放。

3.2 JUC 中的应用场景

除了上边 ReentrantLock 的可重入性的应用，AQS 作为并发编程的框架，为很多其他同步工具提供了良好的解决方案。下面列出了 JUC 中的几种同步工具，大体介绍一下 AQS 的应用场景：

同步工具	同步工具与AQS的关联
ReentrantLock	使用 AQS 保存锁重复持有的次数。当一个线程获取锁时，ReentrantLock 记录当前获得锁的线程标识，用于检测是否重复获取，以及错误线程试图解锁操作时异常情况的处理。
Semaphore	使用 AQS 同步状态来保存信号量的当前计数。tryRelease 会增加计数，acquireShared 会减少计数。
CountDownLatch	使用 AQS 同步状态来表示计数。计数为 0 时，所有的 Acquire 操作（CountDownLatch 的 await 方法）才可以通过。

同步工具	同步工具与AQS的关联
ReentrantRead-WriteLock	使用 AQS 同步状态中的 16 位保存写锁持有的次数，剩下的 16 位用于保存读锁的持有次数。
ThreadPoolExec-utor	Worker 利用 AQS 同步状态实现对独占线程变量的设置（tryAcquire 和 tryRelease）。

3.3 自定义同步工具

了解 AQS 基本原理以后，按照上面所说的 AQS 知识点，自己实现一个同步工具。

```
public class LeeLock  {

    private static class Sync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire (int arg) {
            return compareAndSetState(0, 1);
        }

        @Override
        protected boolean tryRelease (int arg) {
            setState(0);
            return true;
        }

        @Override
        protected boolean isHeldExclusively () {
            return getState() == 1;
        }
    }

    private Sync sync = new Sync();

    public void lock () {
        sync.acquire(1);
    }

    public void unlock () {
        sync.release(1);
    }
}
```

通过我们自己定义的 Lock 完成一定的同步功能。

```
public class LeeMain {

    static int count = 0;
    static LeeLock leeLock = new LeeLock();

    public static void main (String[] args) throws InterruptedException {

        Runnable runnable = new Runnable() {
            @Override
            public void run () {
                try {
                    leeLock.lock();
                    for (int i = 0; i < 10000; i++) {
                        count++;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    leeLock.unlock();
                }

            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(count);
    }
}
```

上述代码每次运行结果都会是 20000。通过简单的几行代码就能实现同步功能，这就是 AQS 的强大之处。

总结

我们日常开发中使用并发的场景太多，但是对并发内部的基本框架原理了解的人却不多。由于篇幅原因，本文仅介绍了可重入锁 ReentrantLock 的原理和 AQS 原理，希望能够成为大家了解 AQS 和 ReentrantLock 等同步器的“敲门砖”。

参考资料

- Lea D. The java. util. concurrent synchronizer framework[J]. Science of Computer Programming, 2005, 58(3): 293-309.
- 《Java 并发编程实战》
- 不可不说的 Java“锁”事

作者简介

李卓，美团点评住宿度假研发中心 Java 研发工程师，2018 年加入美团点评。

招聘信息

美团到店住宿门票业务研发团队负责美团酒店和门票核心业务系统建设。

- 美团酒店屡次刷新行业记录，最近 12 个月酒店预订间夜量达到 3 个亿，单日入住间夜量峰值突破 300 万，单季度间夜突破 1 亿间。
- 美团门票 2018 年出票量达到一亿张，成为国内门票预订规模顶尖的平台。技术团队的愿景是：建设打造旅游住宿行业一流的技术架构，从质量、安全、效率、性能多角度保障系统高速发展。

美团到店住宿门票业务研发团队期待优秀的技术伙伴加入，欢迎投简历至：tech@meituan.com（邮件标题注明：美团到店住宿门票业务研发团队）

架构

美团点评 Kubernetes 集群管理实践

国梁

背景

作为国内领先的生活服务平台，美团点评很多业务都具有非常显著、规律的“高峰”和“低谷”特征。尤其遇到节假日或促销活动，流量还会在短时间内出现爆发式的增长。这对集群中心的资源弹性和可用性有非常高的要求，同时也会使系统在支撑业务流量时的复杂度和成本支出呈现指数级增长。而我们需要做的，就是利用有限的资源最大化地提升集群的吞吐能力，以保障用户体验。

本文将介绍美团点评 Kubernetes 集群管理与使用实践，包括美团点评集群管理与调度系统介绍、Kubernetes 管理与实践、Kubernetes 优化与改造以及资源管理与优化等。

美团点评集群管理与调度系统

美团点评在集群管理和资源优化这条道路上已经“摸爬滚打”多年。2013 年，开始构建基于传统虚拟化技术的资源交付方式；2015 年 7 月，开始建立完善的集群管理与调度系统——HULK，目标是推动美团点评服务容器化；2016 年，完成基于 Docker 容器技术自研实现了弹性伸缩能力，来提升交付速度和应对快速扩缩容的需求，实现弹性扩容、缩容，提升资源利用率，提升业务运维效率，合理有效的降低企业 IT 运维成本；2018 年，开始基于 Kubernetes 来进行资源管理和调度，进一步提升资源的使用效率。



美团点评集群管理与调度平台演进

最初，美团点评通过基于 Docker 容器技术自研实现了弹性伸缩能力，主要是为了解决基于虚拟化技术的管理及部署机制在应对服务快速扩容、缩容需求时存在的诸多不足。例如资源实例创建慢、无法统一运行环境、实例部署和交付流程长、资源回收效率低、弹性能力差等等。经过调研与测试，结合业界的实践经验，我们决定基于 Docker 容器技术自研集群管理与调度系统，有效应对快速扩缩容的需求，提升资源的利用效率。我们把它叫做“绿巨人”——HULK，这个阶段可以看作是 HULK1.0。

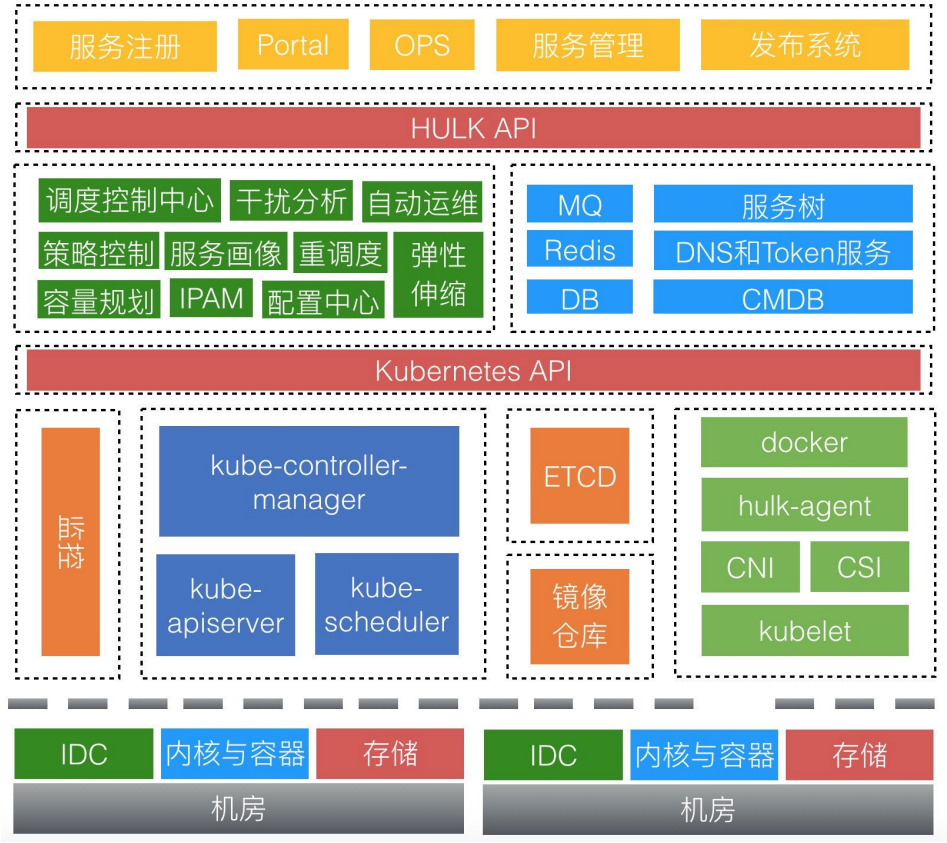
之后，在生产环境中经过不断摸索和尝试，我们逐渐意识到，仅仅满足于集群的弹性伸缩能力是不够的，成本和效率肯定是未来必将面临且更为棘手的问题。我们吸取了 2 年来 HULK 1.0 的开发和运维经验，在架构和支撑系统层面做了进一步优化和改进，并借助于生态和开源的力量来为 HULK 赋能，即引入了开源的集群管理与调度系统 Kubernetes，期望能进一步提升集群管理、运行的效率和稳定性，同时降低资源成本。所以我们从自研平台转向了开源的 Kubernetes 系统，并基于 Kubernetes 系统打造了更加智能化的集群管理与调度系统——HULK2.0。

架构全览

在架构层面，HULK2.0 如何能与上层业务和底层 Kubernetes 平台更好地分层和解耦，是我们在设计之初就优先考虑的问题。我们期望它既要能对业务使用友好，又能最大限度地发挥 Kubernetes 的调度能力，使得业务层和使用方毋需关注资源关系细节，所求即所得；同时使发布、配置、计费、负载等逻辑层与底层的 Kubernetes 平台解耦分层，并保持兼容原生 Kubernetes API 来访问 Kubernetes

集群。从而可以借助于统一的、主流的、符合业界规范的标准，来解决美团点评基础架构面临的复杂的、多样的、不统一的管理需求。

架构介绍



HULK2.0 架构图

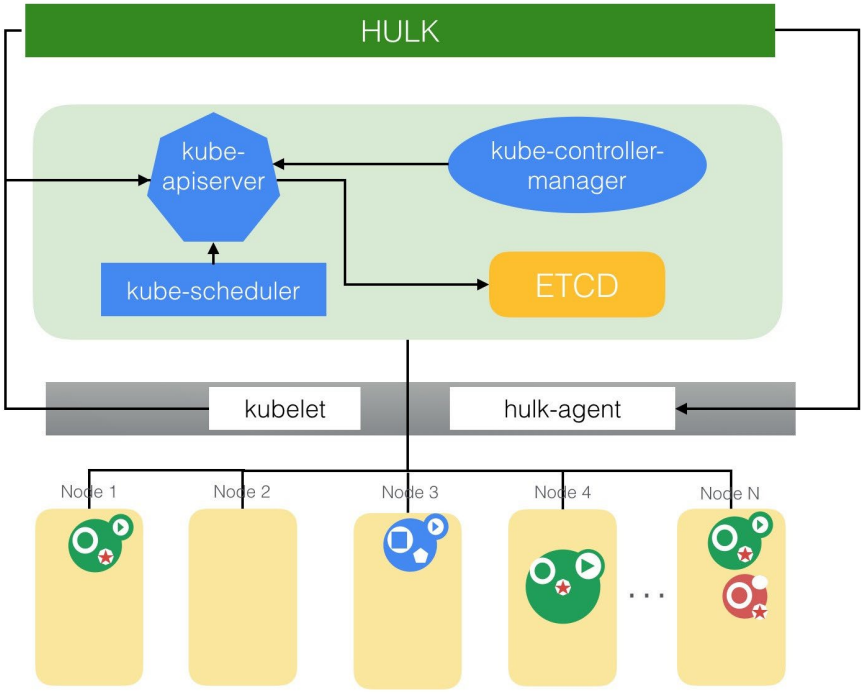
自上而下来看，美团集群管理与调度平台面向全公司服务，有各个主要业务线、统一的 OPS 平台以及 Portal 平台，HULK 不可能针对每个平台定制化接口和解决方案，所以需要将多样的业务和需求抽象收敛，最终统一通过 HULK API 来屏蔽 HULK 系统的细节，做到 HULK 与上层业务方的解耦。HULK API 是对业务层和资源需求的抽象，是外界访问 HULK 的唯一途径。

解决了上层的问题后，我们再来看与下层 Kubernetes 平台的解耦。HULK 接到上层资源请求后，首先要进行一系列的初始化工作，包括参数校验、资源余量、IP 和 Hostname 的分配等等，之后向 Kubernetes 平台实际申请分配机器资源，最终将资源交付给用户，Kubernetes API 进一步将资源需求收敛和转换，让我们可以借助于 Kubernetes 的资源管理优势。Kubernetes API 旨在收敛 HULK 的资源管理逻辑并与业界主流对齐。此外，因为完全兼容 Kubernetes API，可以让我们借助社区和生态的力量，共同建设和探索。

可以看到，HULK API 和 Kubernetes API 将我们整个系统分为三层，这样可以让每一层都专注于各自的模块。

Kubernetes 管理与实践

为什么会选择 Kubernetes 呢？ Kubernetes 并不是市面上唯一的集群管理平台（其他如 Docker Swarm 或 Mesos），之所以选择它，除了它本身优秀的架构设计，我们更加看重的是 Kubernetes 提供的不是一个解决方案，而是一个平台和一种能力。这种能力能够让我们真正基于美团点评的实际情况来扩展，同时能够依赖和复用多年来的技术积累，给予我们更多选择的自由，包括我们可以快速地部署应用程序，而无须面对传统平台所具有的风险，动态地扩展应用程序以及更好的资源分配策略。



HULK-Kubernetes 架构图

Kubernetes 集群作为整个 HULK 集群资源管理与平台的基础，需求是稳定性和可扩展性，风险可控性和集群吞吐能力。

集群运营现状

- 集群规模：10 万 + 级别线上实例，多地域部署，还在不断快速增长中。
- 业务的监控告警：集群对应用的启动和状态数据进行采集，container-init 自动集成业务监控信息，业务程序毋需关注，做到可插拔、可配置。
- 资源的健康告警：从资源的角度对 Node、Pod 和 Container 等重要数据监控采集，及时发现它们的状态信息，例如 Node 不可用、Container 不断重启等等。
- 定时巡检与对账：每天自动对所有宿主机进行状态检查，包括剩余磁盘量（数据卷）、D 进程数量、宿主机状态等，并对 AppKey 扩容数据和实际的 Pod 和容器数据同步校验，及时发现不一致情况。

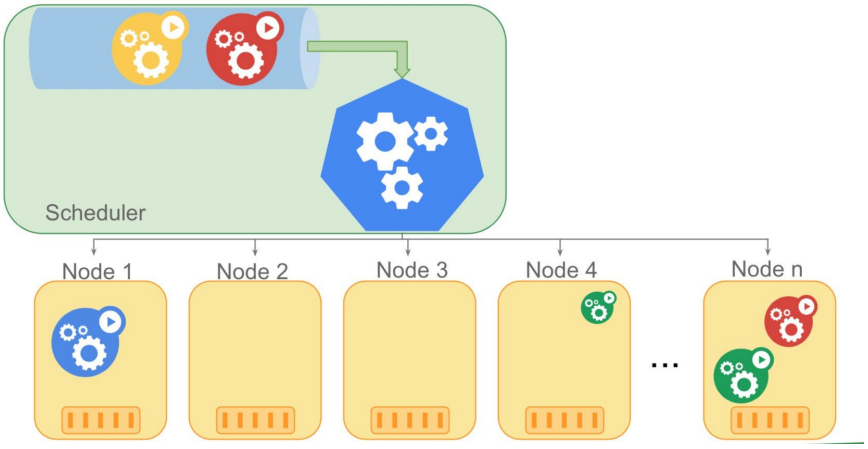
- 集群数据可视化：对当前集群状态，包括宿主机资源状态、服务数、Pod 数、容器化率、服务状态、扩缩容数据等等可视化；并提供了界面化的服务配置、宿主机下线以及 Pod 迁移操作入口。
- 容量规划与预测：提前感知集群资源状态，预先准备资源；基于规则和机器学习的方式感知流量和高峰，保证业务正常、稳定、高效地运行。

Kubernetes 优化与改造

kube-scheduler 性能优化

我们有集群在使用 1.6 版本的调度器，随着集群规模的不断增长，旧版本的 Kubernetes 调度器（1.10 之前版本）在性能和稳定性的问题逐渐凸显，由于调度器的吞吐量低，导致业务扩容超时失败，在规模近 3000 台的集群上，一次 Pod 的调度耗时在 5s 左右。Kubernetes 的调度器是队列化的调度器模型，一旦扩容高峰等待的 Pod 数量过多就会导致后面 Pod 的扩容超时。为此，我们对调度器性能进行了大幅度的优化，并取得了非常明显的提升，根据我们的实际生产环境验证，性能比优化前提升了 400% 以上。

Kubernetes 调度器工作模型如下：



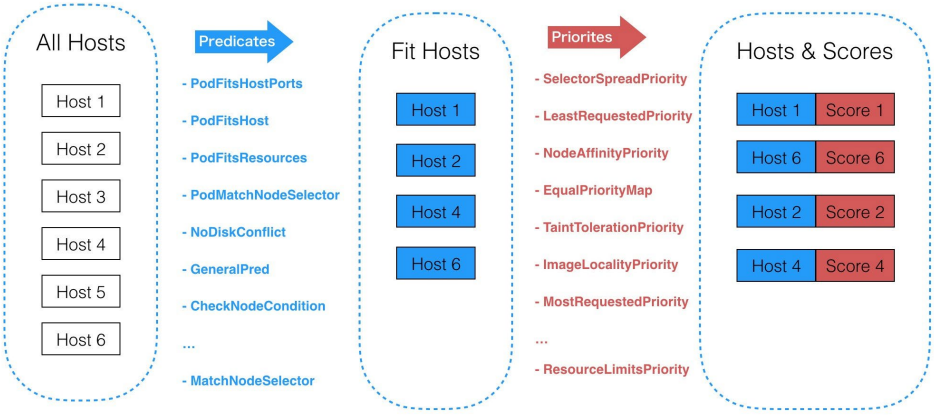
kube-scheduler 示意图

（kubernetes 调度器，图片来源于网络）

预选失败中断机制

一次调度过程在判断一个 Node 是否可作为目标机器时，主要分为三个阶段：

- 预选阶段：硬性条件，过滤掉不满足条件的节点，这个过程称为 Predicates。这是固定先后顺序的一系列过滤条件，任何一个 Predicate 不符合则放弃该 Node。
- 优选阶段：软性条件，对通过的节点按照优先级排序，称之为 Priorities。每一个 Priority 都是一个影响因素，都有一定的权重。
- 选定阶段：从优选列表中选择优先级最高的节点，称为 Select。选择的 Node 即为最终部署 Pod 的机器。

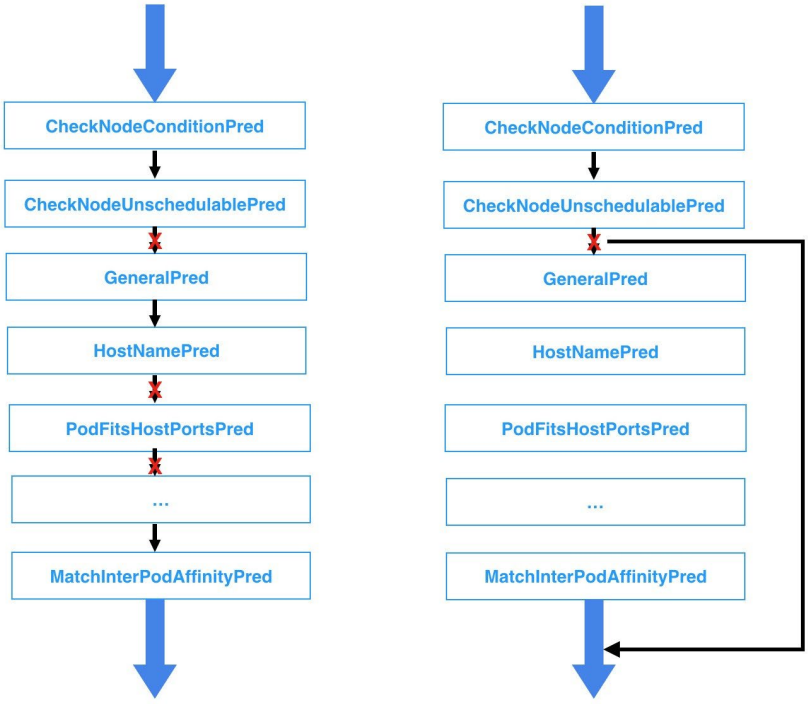


kube-scheduler 调度过程

通过深入分析调度过程可以发现，调度器在预选阶段即使已经知道当前 Node 不符合某个过滤条件仍然会继续判断后续的过滤条件是否符合。试想如果有上万台 Node 节点，这些判断逻辑会浪费很多计算时间，这也是调度器性能低下的一个重要因素。

为此，我们提出了“预选失败中断机制”，即一旦某个预选条件不满足，那么该 Node 即被立即放弃，后面的预选条件不再做判断计算，从而大大减少了计算量，调

度性能也大大提升。如下图所示：



kube-scheduler 的 Predicates 过程

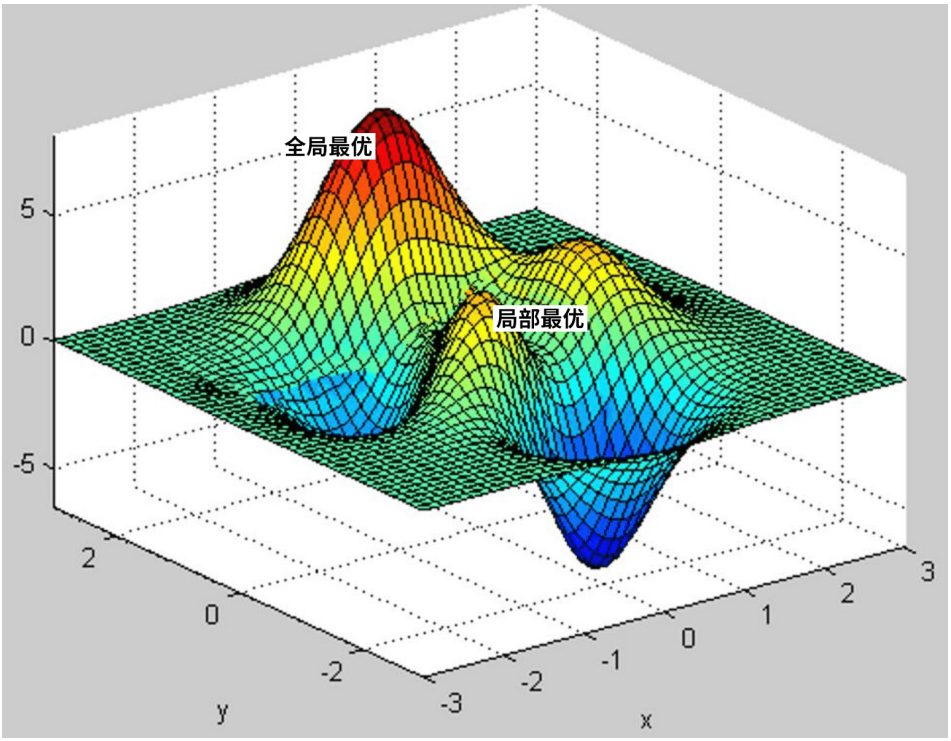
我们把该项优化贡献给了 Kubernetes 社区（详见 PR），增加了 alwaysCheckAllPredicates 策略选项，并在 Kubernetes1.10 版本发布并开始作为默认的调度策略，当然你也可以通过设置 alwaysCheckAllPredicates=true 使用原先的调度策略。

在实际测试中，调度器至少可以提升 40% 的性能，如果你目前在使用的 Kube-scheduler 的版本低于 1.10，那么建议你尝试升级到新的版本。

局部最优解

对于优化问题尤其是最优化问题，我们总希望找到全局最优的解或策略，但是当问题的复杂度过高，要考虑的因素和处理的信息量过多时，我们往往会倾向于接受局部最优解，因为局部最优解的质量不一定都是差的。尤其是当我们有确定的评判标

准，同时标明得出的解是可以接受的话，通常会接收局部最优的结果。这样，从成本、效率等多方面考虑，才是我们在实际工程中真正会采取的策略。



kube-scheduler 的局部最优解（图片来源于网络）

当前调度策略中，每次调度调度器都会遍历集群中所有的 Node，以便找出最优的节点，这在调度领域称之为 BestFit 算法。但是在生产环境中，我们是选取最优 Node 还是次优 Node，其实并没有特别大的区别和影响，有时候我们还是会避免选取最优的 Node（例如我们集群为了解决新上线机器后频繁在该机器上创建应用的问题，就将最优解随机化）。换句话说，找出局部最优解就能满足需求。

假设集群一共 1000 个 Node，一次调度过程 PodA，这其中有 700 个 Node 都能通过 Predicates（预选阶段），那么我们就会把所有的 Node 遍历并找出这 700 个 Node，然后经过得分排序找出最优的 Node 节点 NodeX。但是采用局部最优算法，即我们认为只要能找出 N 个 Node，并在这 N 个 Node 中选择得分最高的 Node 即

能满足需求，比如默认找出 100 个可以通过 Predicates（预选阶段）的 Node 即可，最优解就在这 100 个 Node 中选择。当然全局最优解 NodeX 也可能不在这 100 个 Node 中，但是我们在这 100 个 Node 中选择最优的 NodeY 也能满足要求。最好的情况是遍历 100 个 Node 就找出这 100 个 Node，也可能遍历了 200 个或者 300 个 Node 等等，这样我们可以大大减少计算时间，同时也不会对我们的调度结果产生太大的影响。

局部最优的策略是我们与社区合作共同完成的，这里面还涉及到如何做到公平调度和计算任务优化的细节（详见 PR1，PR2），该项优化在 Kubernetes 1.12 版本中发布，并作为当前默认调度策略，可以大幅度提升调度性能，尤其在大规模集群中的提升，效果非常明显。

kubelet 改造

风险可控性

前面提到，稳定性和风险可控性对大规模集群管理来说非常重要。从架构上来看，Kubelet 是离真实业务最近的集群管理组件，我们知道社区版本的 Kubelet 对本机资源管理有着很大的自主性，试想一下，如果某个业务正在运行，但是 Kubelet 由于出发了驱逐策略而把这个业务的容器干掉了会发生什么？这在我们的集群中是不应该发生的，所以需要收敛和封锁 Kubelet 的自决策能力，它对本机上业务容器的操作都应该从上层平台发起。

容器重启策略

Kernel 升级是日常的运维操作，在通过重启宿主机来升级 Kernel 版本的时候，我们发现宿主机重启后，上面的容器无法自愈或者自愈后版本不对，这会引发业务的不满，也造成了我们不小的运维压力。后来我们为 Kubelet 增加了一个重启策略（Reuse），同时保留了原生重启策略（Rebuild），保证容器系统盘和数据盘的信息都能保留，宿主机重启后容器也能自愈。

IP 状态保持

根据美团点评的网络环境，我们自研了 CNI 插件，并通过基于 Pod 唯一标识来申请和复用 IP。做到了应用 IP 在 Pod 迁移和容器重启之后也能复用，为业务上线和运维带来了不少的收益。

限制驱逐策略

我们知道 Kubelet 拥有节点自动修复的能力，例如在发现异常容器或不合规容器后，会对它们进行驱逐删除操作，这对于我们来说风险太大，我们允许容器在一些次要因素方面可以不合规。例如当 Kubelet 发现当前宿主机上容器个数比设置的最大容器个数大时，会挑选驱逐和删除某些容器，虽然正常情况下不会轻易发生这种问题，但是我们也需要对此进行控制，降低此类风险。

可扩展性

资源调配

在 Kubelet 的扩展性方面我们增强了资源的可操作性，例如为容器绑定 Numa 从而提升应用的稳定性；根据应用等级为容器设置 CPUShare，从而调整调度权重；为容器绑定 CPUSet 等等。

增强容器

我们打通并增强了业务对容器的配置能力，支持业务给自己的容器扩展 ulimit、io limit、pid limit、swap 等参数的同时也增强容器之间的隔离能力。

应用原地升级

大家都知道，Kubernetes 默认只要 Pod 的关键信息有改动，例如镜像信息，就会出发 Pod 的重建和替换，这在生产环境中代价是很大的，一方面 IP 和 HostName 会发生改变，另一方面频繁的重建也给集群管理带来了更多的压力，甚至还可能导致无法调度成功。为了解决该问题，我们打通了自上而下的应用原地升级功能，即可以动态高效地修改应用的信息，并能在原地（宿主机）进行升级。

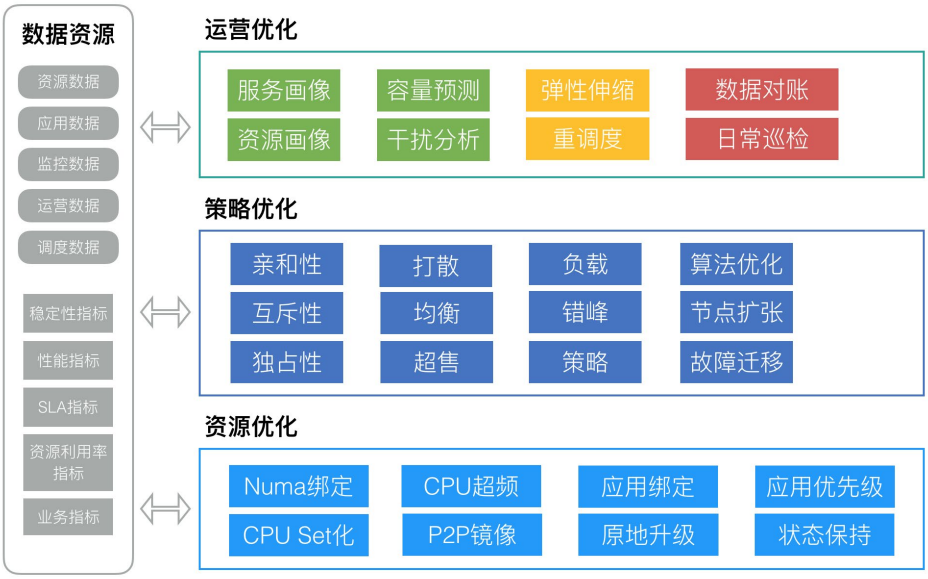
镜像分发

镜像分发是影响容器扩容时长的一个重要环节，我们采取了一系列手段来优化，

保证镜像分发效率高且稳定：

- 跨 Site 同步：保证服务器总能从就近的镜像仓库拉取到扩容用的镜像，减少拉取时间，降低跨 Site 带宽消耗。
- 基础镜像预分发：美团点评的基础镜像是构建业务镜像的公共镜像。业务镜像层是业务的应用代码，通常比基础镜像小很多。在容器扩容的时候如果基础镜像已经在本地，就只需要拉取业务镜像的部分，可以明显的加快扩容速度。为达到这样的效果，我们会把基础镜像事先分发到所有的服务器上。
- P2P 镜像分发：基础镜像预分发在有些场景会导致上千个服务器同时从镜像仓库拉取镜像，对镜像仓库服务和带宽带来很大的压力。因此我们开发了镜像 P2P 分发的功能，服务器不仅能从镜像仓库中拉取镜像，还能从其他服务器上获取镜像的分片。

资源管理与优化



资源管理与优化

优化关键技术

- 服务画像：对应用的 CPU、内存、网络、磁盘和网络 I/O 容量和负载画像，了解应用的特征、资源规格和应用类型以及不同时间对资源的真实使用，然后从服务角度和时间维度进行相关性分析，从而进行整体调度和部署优化。
- 亲和性和互斥性：哪些应用放在一起使整体计算能力比较少而吞吐能力比较高，它们就存在一定亲和性；反之如果应用之间存在资源竞争或相互影响，则它们之间就存在着互斥性。
- 场景优先：美团点评的业务大都是基本稳定的场景，所以场景划分很有必要。例如一类业务对延迟非常敏感，即使在高峰时刻也不允许有太多的资源竞争产生，这种场景就要避免和减少资源竞争引起的延迟，保证资源充足；一类业务在有些时间段需要的 CPU 资源可能会突破配置的上限，我们通过 CPU Set 化的方式让这类业务共享这部分资源，以便能够突破申请规格的机器资源限制，不仅服务能够获得更高的性能表现，同时也把空闲的资源利用了起来，资源使用率进一步提升。
- 弹性伸缩：应用部署做到流量预测、自动伸缩、基于规则的高低峰伸缩以及基于机器学习的伸缩机制。
- 精细化资源调配：基于资源共享和隔离技术做到了精细化的资源调度和分配，例如 Numa 绑定、任务优先级、CPU Set 化等等。

策略优化

调度策略的主要作用在两方面，一方面是按照既定策略部署目标机器；二是能做到集群资源的排布最优。

- 亲和性：有调用关系和依赖的应用，或哪些应用放在一起能使整体计算能力比较少、吞吐能力比较高，这些应用间就存在一定亲和性。我们的 CPU Set 化即是利用了对 CPU 的偏好构建应用的亲和性约束，让不同 CPU 偏好的应用互补。

- 互斥性：跟亲和性相对，主要是对有竞争关系或业务干扰的应用在调度时尽量分开部署。
- 应用优先级：应用优先级的划分是为我们解决资源竞争提供了前提。当前当容器发生资源竞争时，我们无法决策究竟应该让谁获得资源，当有了应用优先级的概念后，我们可以做到，在调度层，限制单台宿主机上重要应用的个数，减少单机的资源竞争，也为单机底层解决资源竞争提供可能；在宿主机层，根据应用优先级分配资源，保证重要应用的资源充足，同时也可运行低优先级应用。
- 打散性：应用的打散主要是为了容灾，在这里分为不同级别的打散。我们提供了不同级别的打散粒度，包括宿主机、Tor、机房、Zone 等等。
- 隔离与独占：这是一类特殊的应用，必须是独立使用一台宿主机或虚拟机隔离环境部署，例如搜索团队的业务。
- 特殊资源：特殊资源是满足某些业务对 GPU、SSD、特殊网卡等特殊硬件需求。

在线集群优化

在线集群资源的优化问题，不像离线集群那样可以通过预知资源需求从而达到非常好的效果，由于未来需求的未知性，在线集群很难在资源排布上达到离线集群的效果。针对在线集群的问题，我们从上层调度到底层的资源使用都采取了一系列的优化。

- Numa 绑定：主要是解决业务侧反馈服务不稳定的问题，通过绑定 Numa，将同一个应用的 CPU 和 Memory 绑定到最合适的 Numa Node 上，减少跨 Node 访问的开销，提升应用性能。
- CPU Set 化：将一组特性互补的应用绑定在同一组 CPU 上，从而让他们能充分使用 CPU 资源。
- 应用错峰：基于服务画像数据为应用错开高峰，减少资源竞争和相互干扰，提升业务 SLA。
- 重调度：资源排布优化，用更少的资源提升业务性能和 SLA；解决碎片问题，

提升资源的分配率。

- 干扰分析：基于业务监控数据指标和容器信息判断哪些容器有异常，提升业务 SLA，发现并处理异常应用。

结束语

当前，在以下几个方面我们正在积极探索：

- 在线 – 离线业务混合部署，进一步提升资源使用效率。
- 智能化调度，业务流量和资源使用感知调度，提升服务 SLA。
- 高性能、强隔离和更安全的容器技术。

作者简介

国梁，美团点评基础研发平台集群调度中心高级工程师。

招聘信息

美团点评基础研发平台集群调度中心，致力于打造高效的业界领先的集群管理与调度平台，通过企业级集群管理平台建设业界领先的云化解决方案，提高集群管理能力和稳定性，同时降低 IT 成本，加速公司的创新发展。同时随着 Kubernetes 已经成为业界的事实标准，美团点评也在逐步拥抱社区，参与开源并且在集群调度领域已经取得很大进展，也期待和业界同仁一起努力，共同提高集群管理和调度能力，降低整个行业的 IT 成本，协同创新发展。美团点评基础研发平台长期招聘集群管理与调度、弹性调度、Kubernetes 以及 Linux 内核方面的人才，有兴趣的同学可以发送简历到 tech@meituan.com 。

美团集群调度系统 HULK 技术演进

涂扬

本文根据美团基础架构部 / 弹性策略团队负责人涂扬在 2019 QCon（全球软件开发大会）上的演讲内容整理而成。本文涉及 Kubernetes 集群管理技术，美团相关的技术实践可参考此前发布的《美团点评 Kubernetes 集群管理实践》。

一、背景

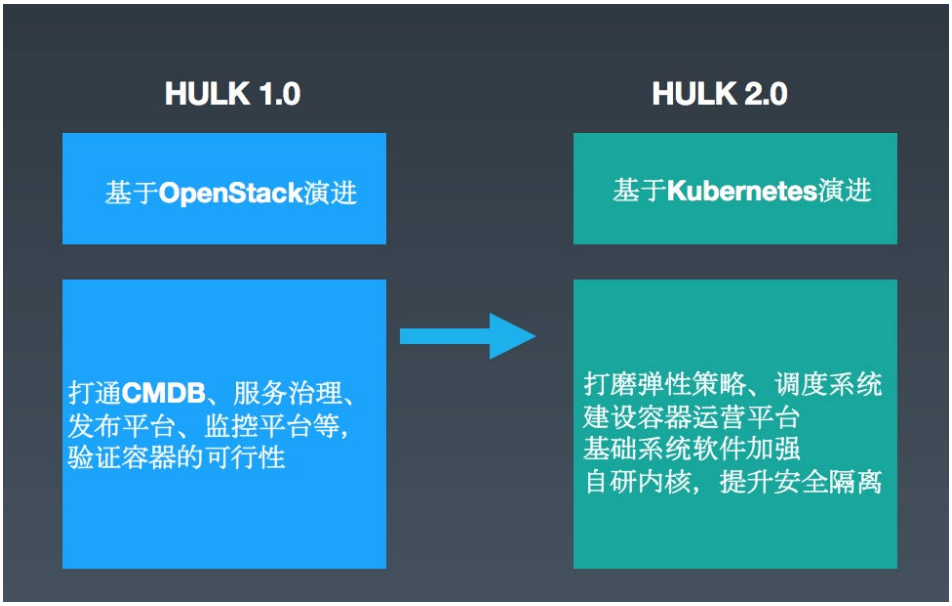
HULK 是美团的容器集群管理平台。在 HULK 之前，美团的在线服务大部分部署都是在 VM 上，在此期间，我们遇到了很大的挑战，主要包括以下两点：

- 环境配置信息不一致：部分业务线下验证正常，但线上验证却不正常。
- 业务扩容流程长：从申请机器、资源审核到服务部署，需要 5 分钟才能完成。

因为美团很多业务都具有明显的高低峰特性，大家一般会根据最高峰的流量情况来部署机器资源，然而在业务低峰期的时候，往往用不了那么多的资源。在这种背景下，我们希望打造一个容器集群管理平台来解决上述的痛点问题，于是 HULK 项目就应运而生了。

HULK 平台包含容器以及弹性调度系统，容器可以统一运行环境、提升交付效率，而弹性调度可以提升业务的资源利用率。在漫威里有个叫 HULK 的英雄，在情绪激动的时候会变成“绿巨人”，情绪平稳后则恢复人身，这一点跟我们容器的”弹性伸缩“特性比较相像，所以我们的系统就取名为”HULK “。

总的来讲，美团 HULK 的演进可以分为 1.0 和 2.0 两个阶段，如下图所示：



在早期，HULK 1.0 是基于 OpenStack 演进的一个集群调度系统版本。这个阶段工作的重点是将容器和美团的基础设施进行融合，比如打通 CMDB 系统、公司内部的服务治理平台、发布平台以及监控平台等等，并验证容器在生产环境的可行性。2018 年，基础架构部将底层的 OpenStack 升级为容器编排标准 Kubernetes，然后我们把这个版本称之为 HULK 2.0，新版本还针对在 1.0 运营过程中遇到的一些问题，对系统专门进行了优化和打磨，主要包括以下几个方面：

- 进一步打磨了弹性策略和调度系统。
- 构建了一站式容器运营平台。
- 对基础系统软件进行加强，自研内核，提升安全隔离能力。

截止发稿时，美团生产环境超过 1 万个应用在使用容器，容器数过 10 万。

二、HULK2.0 集群调度系统总体架构图



上图中，最上层是集群调度系统对接的各个平台，包括服务治理、发布平台、测试部署平台、CMDB 系统、监控平台等，我们将这些系统打通，业务就可以无感知地从 VM 迁移到容器中。其中：

- **容器弹性**：可以让接入的业务按需使用容器实例。
- **服务画像**：负责应用运行情况的搜集和统计，如 CPU/IO 使用、服务高峰期、上下游等信息，为弹性伸缩、调度系统提供支持。
- **容器编排和镜像管理**：负责对实例进行调度与应用实例构建。

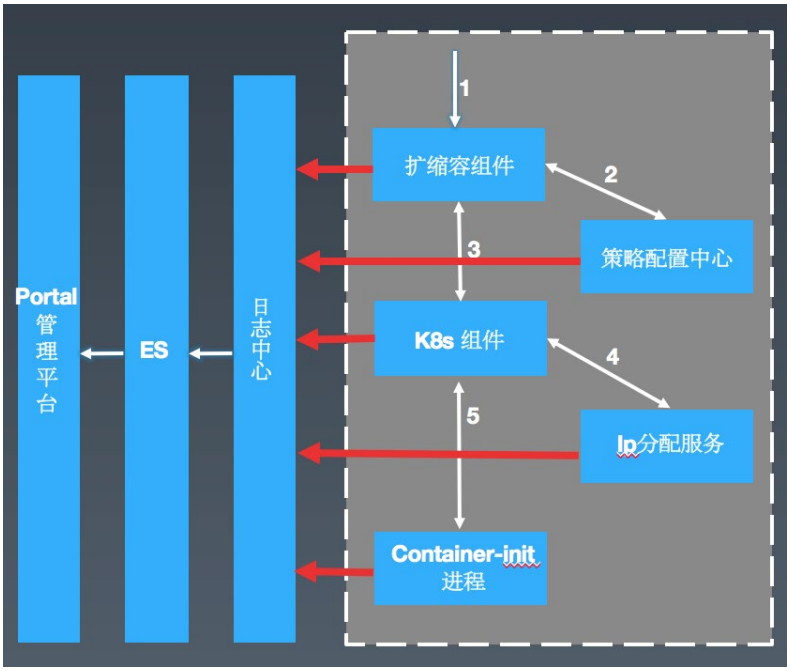
最底层的 HULK Agent 是我们在每个 Node 上的代理程序。此前，在美团技术团队官方博客上，我们也分享过底层的镜像管理和容器运行时相关内容，参见《美团容器技术研发实践》一文。而本文将重点阐述容器编排（调度系统）和容器弹性（弹性伸缩平台），以及团队遇到的一些问题以及对应的解决方案，希望对大家能有所启发。

三、调度系统痛点、解法

3.1 业务扩缩容异常

痛点：集群运维人员排查成本较高。

为了解决这个问题，我们可以先看一下调度系统的简化版架构，如下图所示：



可以看到，一次扩缩容请求基本上会经历以下这些流程：

a. 用户或者上层系统发起扩缩容请求。

b. 扩缩容组件从策略配置中心获取对应服务的配置信息。

c. 将对应的配置信息提交到美团自研的一个 API 服务（扩展的 K8s 组件），然后 K8s 各 Master 组件就按照原生的工作流程开始 Work。

d. 当某个实例调度到具体的 Node 上的时候，开始通过 IP 分配服务获取对应的 Hostname 和 IP。

e. Container-init 是一号进程，在容器内部拉起各个 Agent，然后启动应用程序。针对已经标准化接入的应用，会自动进行服务注册，从而承载流量。

而这些模块是由美团内部的不同同学分别进行维护，每次遇到问题时，就需要多个同学分别核对日志信息。可想而知，这种排查问题的方式的成本会有多高。

解法：类似于分布式调用链中的 traceId，每次扩缩容会生成一个 TaskId，我们在关键链路上进行打点的同时带上 TaskId，并按照约定的格式统一接入到美团点评

日志中心，然后在可视化平台 HULK Portal 进行展示。

落地效果：

- **问题排查提效**：之前排查类似问题，多人累计耗时平均需要半个小时。目前，1 个管理员通过可视化的界面即可达到分钟级定位到问题。
- **系统瓶颈可视化**：全链路上每个时段的平均耗时信息一览无遗。

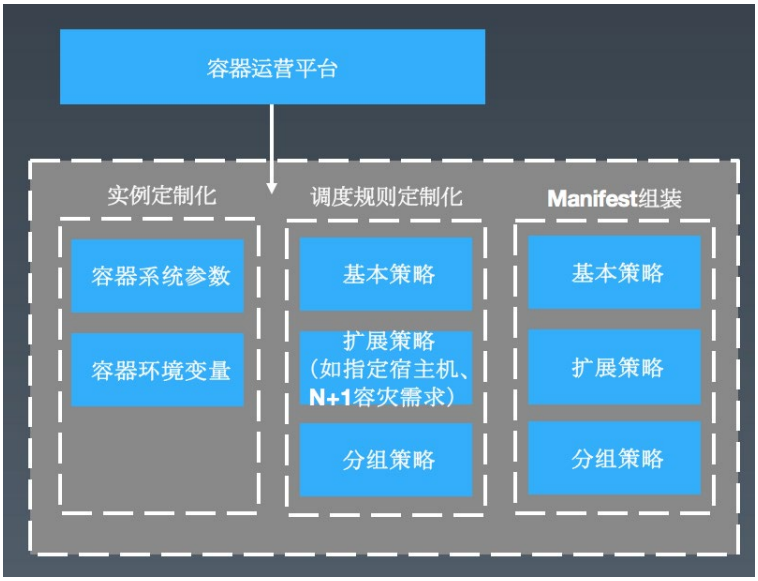
3.2 业务定制化需求

痛点：每次业务的特殊配置都可能变更核心链路代码，导致整体系统的灵活性不够。

具体业务场景如下：

- 业务希望能够去设置一些系统参数，比如开启 swap，设置 memlock、ulimit 等。
- 环境变量配置，比如应用名、ZooKeeper 地址等。

解法：建设一体化的调度策略配置中心，通过调度策略配置中心，可定制化调度规则。

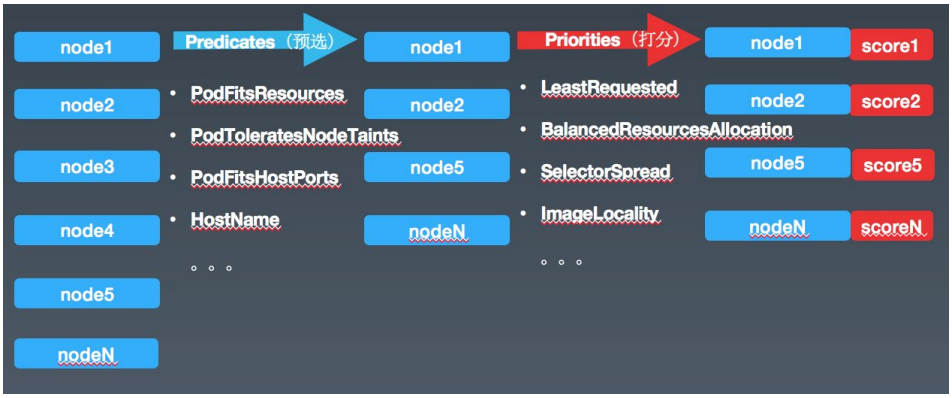


- 实例基本配置，比如业务想给机器加 Set 化、泳道标识。
- 实例的扩展配置：如部分业务，比如某些服务想将实例部署在包含特定硬件的宿主机，会对核心业务有 N+1 的容灾需求，并且还需要将实例部署在不同的 IDC 上。
- 相同配置的应用可以创建一个组，将应用和组进行关联。

在策略配置中心，我们会将这些策略进行 Manifest 组装，然后转换成 Kubernetes 可识别的 YAML 文件。

落地效果：实现了平台自动化配置，运维人员得到解放。

3.3 调度策略优化



接下来，介绍一下 Kubernetes 调度器 Scheduler 的默认行为：它启动之后，会一直监听 ApiServer，通过 ApiServer 去查看未 Bind 的 Pod 列表，然后根据特定的算法和策略选出一个合适的 Node，并进行 Bind 操作。具体的调度策略分为两个阶段：Predicates 预选阶段和 Priorities 打分阶段。

Predicates 预选阶段（一堆的预选条件）：PodFitsResources 检查是否有足够的资源（比如 CPU、内存）来满足一个 Pod 的运行需求，如果不满足，就直接过滤掉这个 Node。

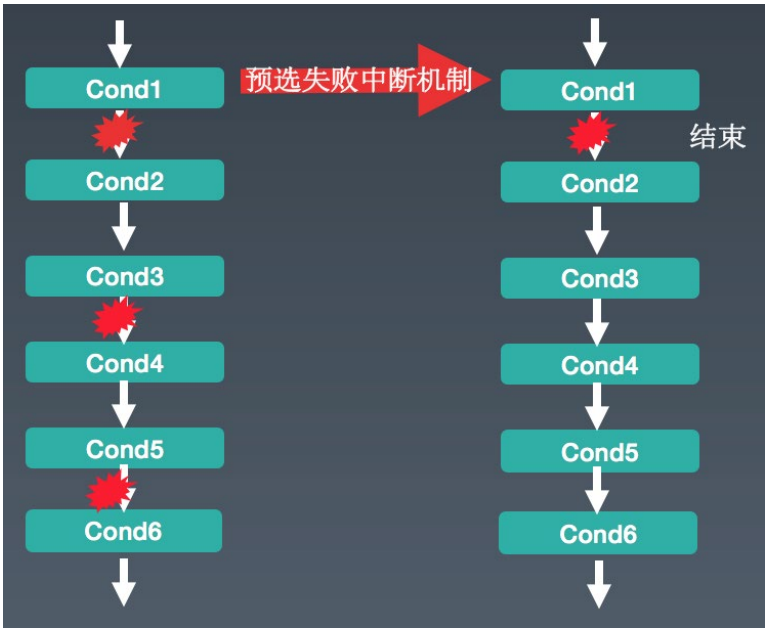
Priorities 打分阶段（一堆的优先级函数）：

- LeastRequested：CPU 和内存具有相同的权重，资源空闲比越高的节点得分越高。
- BalancedResourcesAllocation：CPU 和内存使用率越接近的节点得分越高。

将以上优先级函数算出来的值加权平均算出来一个得分（0−10），分数越高，节点越优。

痛点一：当集群达到 3000 台规模的时候，一次 Pod 调度耗时 5s 左右（K8s 1.6 版本）。如果在预选阶段，当前 Node 不符合过滤条件，依然会判断后续的过滤条件是否符合。假设有上万台 Node 节点，这种判断逻辑便会浪费较多时间，造成调度器的性能下降。

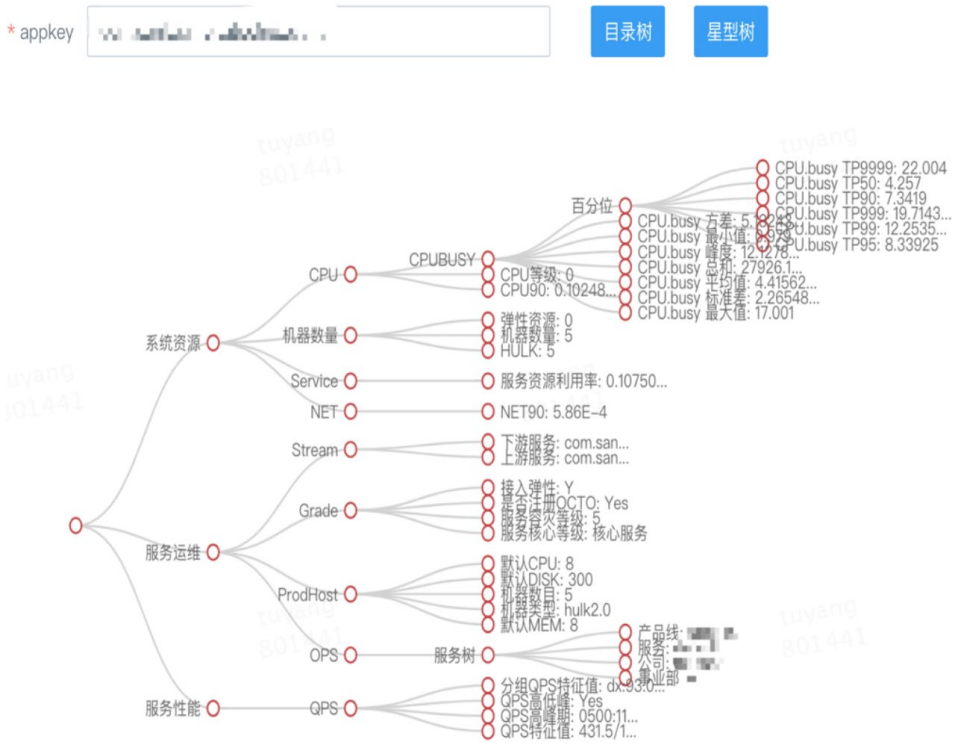
解法：当前 Node 中，如果遇到一个预选条件不满足（比较像是短路径原则），就将这个 Node 过滤掉，大大减少了计算量，调度性能也得到大幅提升。



成效：生产环境验证，提升了 40% 的性能。这个方案目前已经成为社区 1.10 版本默认的调度策略，技术细节可以参考 GitHub 上的 PR。

痛点二：资源利用率最大化和服务 SLA 保障之间的权衡。

解法：我们基于服务的行为数据构建了服务画像系统，下图是我们针对某个应用进行服务画像后的树图展现。



调度前：可以将有调用关系的 Pod 设置亲和性，竞争相同资源的 Pod 设置反亲和性，相同宿主机上最多包含 N 个核心应用。 **调度后**：经过上述规则调度后，在宿主机上如果依然出现了资源竞争，优先保障高优先级应用的 SLA。

3.4 重编排问题

痛点：

（1）容器重启 / 迁移场景：

- 容器和系统盘的信息丢失。
- 容器的 IP 变更。

（2）驱逐场景：Kubelet 会自动杀死一些违例容器，但有可能是非常核心的业务。

解法：

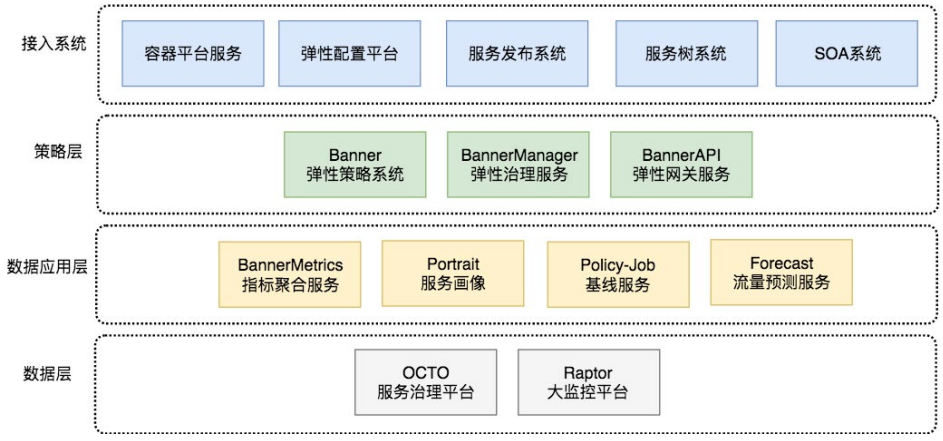
（1）容器重启 / 迁移场景：

- 新增 Reuse 策略，保留原生重启策略（Rebuild）。
- 定制化 CNI 插件，基于 Pod 标识申请和复用 IP。

（2）关闭原生的驱逐策略，通过外部组件来做决策。

四、弹性伸缩平台痛点、解法

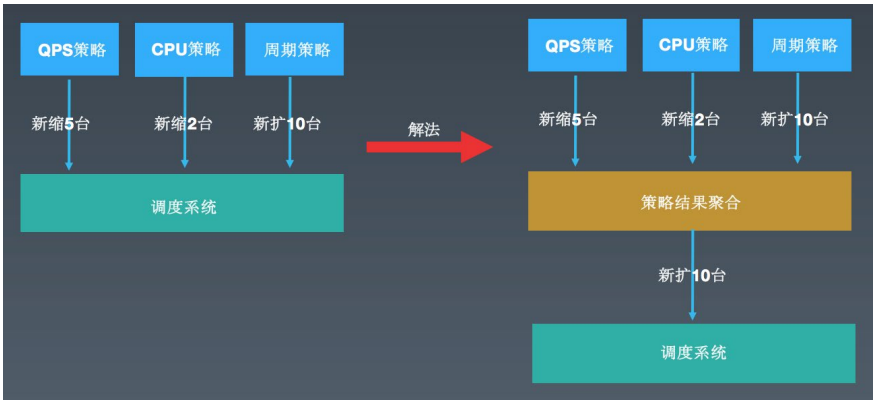
弹性伸缩平台整体架构图如下：



注：Raptor 是美团点评内部的大监控平台，整合了 CAT、Falcon 等监控产品。

在弹性伸缩平台演进的过程中，我们主要遇到了以下 5 个问题。

4.1 多策略决策不一致



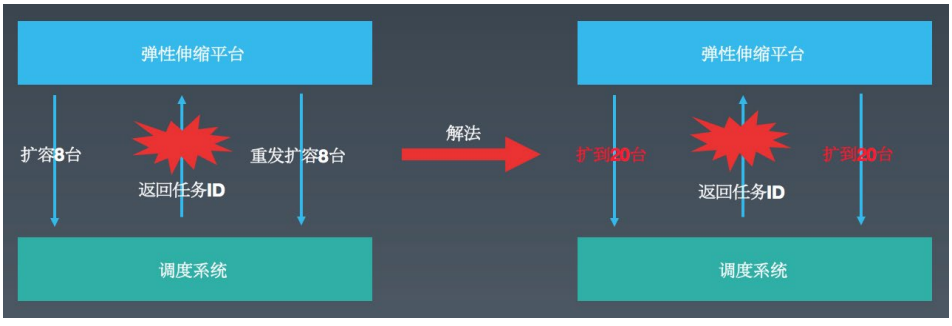
如上图所示，一个业务配置了 2 条监控策略和 1 条周期策略：

- **监控策略**：当某个指标（比如 QPS、CPU）超过阈值上限后开始扩容，低于阈值下限后开始缩容。
- **周期策略**：在某个固定的时间开始扩容，另外一个固定的时间开始缩容。

早期的设计是各条策略独自决策，扩容顺序有可能是：缩 5 台、缩 2 台、扩 10 台，也有可能是：扩 10 台、缩 5 台、缩 2 台，就可能造成一些无效的扩缩行为。

解法：增加了一个聚合层（或者把它称之为策略协商层），提供一些聚合策略：默认策略（多扩少缩）和权重策略（权重高的来决策扩缩行为），减少了大量的无效扩缩现象。

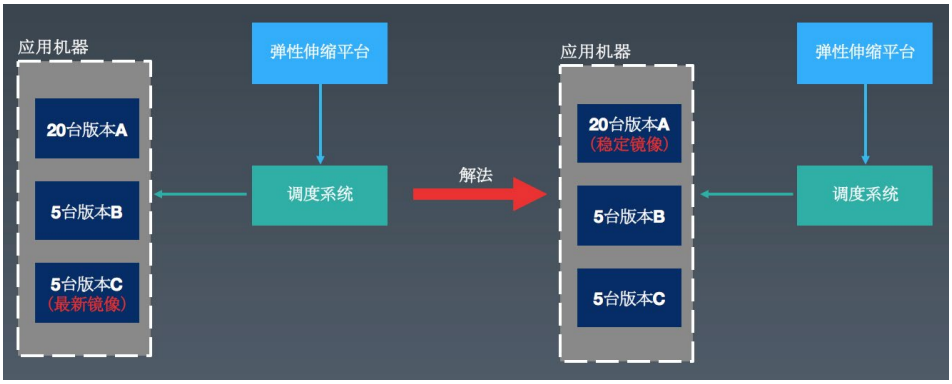
4.2 扩缩不幂等



如上图所示，聚合层发起具体扩缩容的时候，因之前采用的是增量扩容方式，在一些场景下会出现频繁扩缩现象。比如，原先 12 台，这个时候弹性伸缩平台告诉调度系统要扩容 8 台，在返回 TaskId 的过程中超时或保存 TaskId 失败了，这个时候弹性伸缩平台会继续发起扩容 8 台的操作，最后导致服务下有 28 台实例（不幂等）。

解法：采用按目标扩容方式，直接告诉对端，希望能扩容到 20 台，避免了短时间内的频繁扩缩容现象。

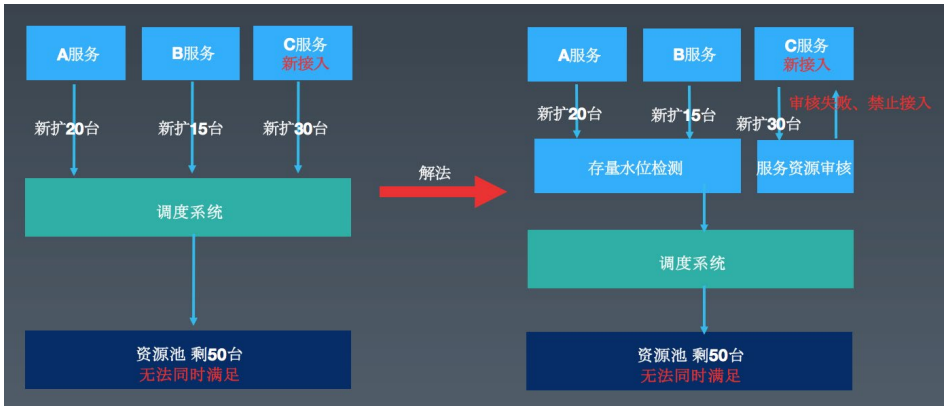
4.3 线上代码多版本



如上图所示，一个业务线上有 30 台机器，存在 3 个版本（A、B、C）。之前我们弹性扩容的做法是采用业务构建的最新镜像进行扩容，但在实际生产环境运行过程中却遇到问题。比如一些业务构建的最新镜像是用来做小流量测试的，本身的稳定性没有保障，高峰期扩容的时候会提升这个版本在线上机器中的比例，低峰期的时候又把之前稳定版本给缩容了，经过一段时间的频繁扩缩之后，最后线上遗留的实例可能都存在问题。

解法：基于约定优于配置原则，我们采用业务的稳定镜像（采用灰度发布流程将线上所有实例均覆盖过一遍的镜像，会自动标记为稳定镜像）进行扩容，这样就比较好地解决了这个问题。

4.4 资源保障问题

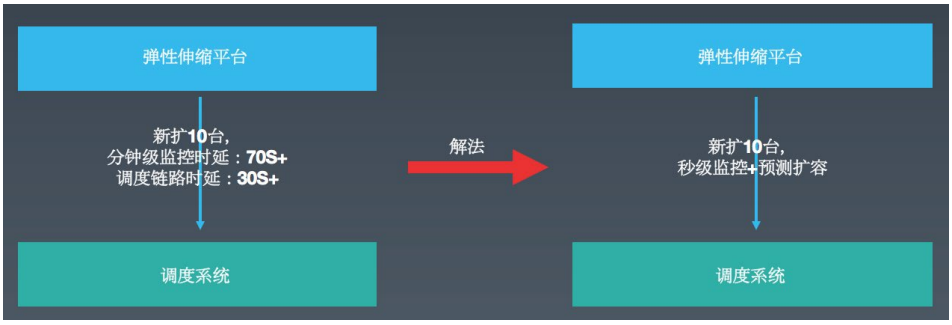


如上图所示，存量中有 2 个服务，一个需要扩容 20 台，一个需要扩容 15 台，这个时候如果新接入一个服务，同一时间需要扩容 30 台，但是资源池只剩余 50 台实例了。这个时候就意味着，谁先扩容谁就可以获得资源保障，后发起的请求就无法获得资源保障。

解法：

（1）**存量资源水位检测**：当存量资源的使用水位超过阈值的时候，比如达到 80% 的时候会有报警，告诉我们需要做资源补充操作。（2）**增量服务弹性资源预估**：如果这个服务通过预判算法评估，接入之后可能会导致存量服务的扩容得不到保障，则拒绝或者补充资源后，再让这个业务接入。

4.5 端到端时效问题



如图所示，我们的分钟级监控时延（比如 1:00:00~1:01:00 的监控数据，大概需要到 1:01:10 后可将采集到的所有数据聚合完成）是 70s+，调度链路时延是 30s+，整体需要上 100s+，在生产环境的业务往往会比较关注扩容时延。

解法：监控系统这块已经建设秒级监控功能。基于这些做法都属于后验性扩容，存在一定的延迟性，目前我们也在探索基于历史行为数据进行服务预测，在监控指标达到扩容阈值前的 1~2 分钟进行提前扩容。

五、经验总结

技术侧：

- 开源产品“本土化”：原生的 Kubernetes 需要和内部已有的基础设施，如服务树、发布系统、服务治理平台、监控系统等做融合，才能更容易在公司内进行落地。
- 调度决策：增量的调度均使用新策略来进行规范化，存量的可采用重调度器进行治理。
- 弹性伸缩：公有云在弹性伸缩这块是没有 SLA 保障的，但是做内部私有云，就需要做好扩容成功率、端到端时延这两块的 SLA 保障。

业务侧：

- 业务迁移：建设了全自动化迁移平台，帮助业务从 VM 自动迁移到容器，极大地降低了因迁移而带来的人力投入。
- 业务成本：使用 HULK 可较好地提升业务运维效率（HULK 具备资源利用率更高、弹性扩容、一键扩容等特点），降低了业务成本。

作者简介

涂扬，美团点评技术专家，现任基础架构部弹性策略团队负责人。

招聘信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：基础架构部弹性策略团队）

保障 IDC 安全：分布式 HIDS 集群架构设计

陈驰

背景

近年来，互联网上安全事件频发，企业信息安全越来越受到重视，而 IDC 服务器安全又是纵深防御体系中的重要一环。保障 IDC 安全，常用的是基于主机型入侵检测系统 Host-based Intrusion Detection System，即 HIDS。在 HIDS 面对几十万台甚至上百万台规模的 IDC 环境时，系统架构该如何设计呢？复杂的服务器环境，网络环境，巨大的数据量给我们带来了哪些技术挑战呢？

需求描述

对于 HIDS 产品，我们安全部门的产品经理提出了以下需求：

1. 满足 50W-100W 服务器量级的 IDC 规模。
2. 部署在高并发服务器生产环境，要求 Agent 低性能低损耗。
3. 广泛的部署兼容性。
4. 偏向应用层和用户态入侵检测（可以和内核态检测部分解耦）。
5. 针对利用主机 Agent 排查漏洞的最急需场景提供基本的能力，可以实现海量环境下快速查找系统漏洞。
6. Agent 跟 Server 的配置下发通道安全。
7. 配置信息读取写入需要鉴权。
8. 配置变更历史记录。
9. Agent 插件具备自更新功能。

分析需求

首先，服务器业务进程优先级高，HIDS Agent 进程自己可以终止，但不能影响宿

主机的主要业务，这是第一要点，那么业务需要具备熔断功能，并具备自我恢复能力。

其次，进程保活、维持心跳、实时获取新指令能力，百万台 Agent 的全量控制时间一定要短。举个极端的例子，当 Agent 出现紧急情况，需要全量停止时，那么全量停止的命令下发，需要在 1–2 分钟内完成，甚至 30 秒、20 秒内完成。这些将会是很大的技术挑战。

还有对配置动态更新，日志级别控制，细分精确控制到每个 Agent 上的每个 HIDS 子进程，能自由地控制每个进程的启停，每个 Agent 的参数，也能精确的感知每台 Agent 的上线、下线情况。

同时，Agent 本身是安全 Agent，安全的因素也要考虑进去，包括通信通道的安全性，配置管理的安全性等等。

最后，服务端也要有一致性保障、可用性保障，对于大量 Agent 的管理，必须能实现任务分摊，并行处理任务，且保证数据的一致性。考虑到公司规模不断地扩大，业务不断地增多，特别是美团和大众点评合并后，面对的各种操作系统问题，产品还要具备良好的兼容性、可维护性等。

总结下来，产品架构要符合以下特性：

1. 集群高可用。
2. 分布式，去中心化。
3. 配置一致性，配置多版本可追溯。
4. 分治与汇总。
5. 兼容部署各种 Linux 服务器，只维护一个版本。
6. 节省资源，占用较少的 CPU、内存。
7. 精确的熔断限流。
8. 服务器数量规模达到百万级的集群负载能力。

技术难点

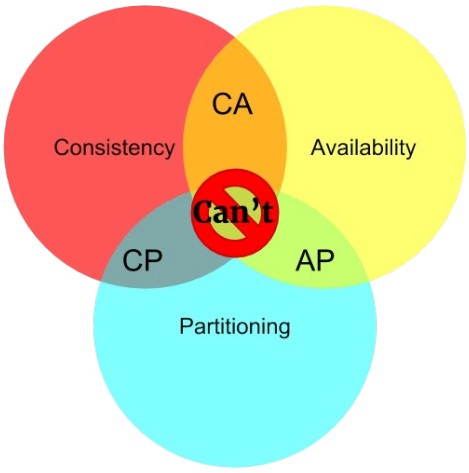
在列出产品需要实现的功能点、技术点后，再来分析下遇到的技术挑战，包括不限于以下几点：

- 资源限制，较小的 CPU、内存。
- 五十万甚至一百万台服务器的 Agent 处理控制问题。
- 量级大了后，集群控制带来的控制效率，响应延迟，数据一致性问题。
- 量级大了后，数据传输对整个服务器内网带来的流量冲击问题。
- 量级大了后，运行环境更复杂，Agent 异常表现的感知问题。
- 量级大了后，业务日志、程序运行日志的传输、存储问题，被监控业务访问量突增带来监控数据联动突增，对内网带宽，存储集群的爆发压力问题。

我们可以看到，技术难点几乎都是**服务器到达一定量级**带来的，对于大量的服务，集群分布式是业界常见的解决方案。

架构设计与技术选型

对于管理 Agent 的服务端来说，要实现高可用、容灾设计，那么一定要做多机房部署，就一定会遇到数据一致性问题。那么数据的存储，就要考虑分布式存储组件。分布式数据存储中，存在一个定理叫 CAP 定理：



CAP-theorem.png

CAP 的解释

关于 CAP 定理，分为以下三点：

- 一致性（Consistency）：分布式数据库的数据保持一致。
- 可用性（Availability）：任何一个节点宕机，其他节点可以继续对外提供服务。
- 分区容错性（网络分区）Partition Tolerance：一个数据库所在的机器坏了，如硬盘坏了，数据丢失了，可以添加一台机器，然后从其他正常的机器把备份的数据同步过来。

根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。理解 CAP 定理的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了 Consistency。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了 Availability。除非两个节点可以互相通信，才能既保证 Consistency 又保证 Availability，这又会导致丧失 Partition Tolerance。

参见：CAP Theorem。

CAP 的选择

为了容灾上设计，集群节点的部署，会选择的异地多机房，所以「Partition tolerance」是不可能避免的。**那么可选的是 AP 与 CP。**

在 HIDS 集群的场景里，各个 Agent 对集群持续可用性没有非常强的要求，在短暂时间内，是可以出现异常，出现无法通讯的情况。但最终状态必须要一致，不能存在集群下发关停指令，而出现个别 Agent 不听从集群控制的情况出现。所以，我们需要一个满足 CP 的产品。

满足 CP 的产品选择

在开源社区中，比较出名的几款满足 CP 的产品，比如 etcd、ZooKeeper、Consul 等。我们需要根据几款产品的特点，根据我们需求来选择符合我们需求的产品。

插一句，网上很多人说 Consul 是 AP 产品，这是个错误的描述。既然 Consul 支持分布式部署，那么一定会出现「网络分区」的问题，那么一定要支持「Partition tolerance」。另外，在 consul 的官网上自己也提到了这点 Consul uses a CP architecture, favoring consistency over availability.

> Consul is opinionated in its usage while Serf is a more flexible and general purpose tool. In CAP terms, Consul uses a CP architecture, favoring consistency over availability. Serf is an AP system and sacrifices consistency for availability. This means Consul cannot operate if the central servers cannot form a quorum while Serf will continue to function under almost all circumstances.

etcd、ZooKeeper、Consul 对比

借用 etcd 官网上 etcd 与 ZooKeeper 和 Consul 的比较图。

	ETCD	ZOOKEEPER	CONSUL	NEWSQL (CLOUD SPANNER, COCKROACHDB, TIDB)
Concurrency Primitives	Lock RPCs, Election RPCs, command line locks, command line elections, recipes in go	External curator recipes in Java	Native lock API	Rare, if any
Linearizable Reads	Yes	No	Yes	Sometimes
Multi-version Concurrency Control	Yes	No	No	Sometimes
Transactions	Field compares, Read, Write	Version checks, Write	Field compare, Lock, Read, Write	SQL-style
Change Notification	Historical and current key intervals	Current keys and directories	Current keys and prefixes	Triggers (sometimes)
User permissions	Role based	ACLs	ACLs	Varies (per-table GRANT, per-database roles)
HTTP/JSON API	Yes	No	Yes	Rarely
Membership Reconfiguration	Yes	>3.5.0	Yes	Yes
Maximum reliable database size	Several gigabytes	Hundreds of megabytes (sometimes several gigabytes)	Hundreds of MBs	Terabytes+
Minimum read linearization latency	Network RTT	No read linearization	RTT + fsync	Clock barriers (atomic, NTP)

etcd-ZooKeeper-Consul

在我们 HIDS Agent 的需求中，除了基本的服务发现 、配置同步 、配置多版本控制 、变更通知等基本需求外，我们还有基于产品安全性上的考虑，比如传输通道加密、用户权限控制、角色管理、基于 Key 的权限设定等，这点 etcd 比较符合我们要求。很多大型公司都在使用，比如 Kubernetes、AWS、OpenStack、Azure、Google Cloud、Huawei Cloud 等，并且 etcd 的社区支持非常好。基于这几点因

素，我们选择 etcd 作为 HIDS 的分布式集群管理。

选择 etcd

对于 etcd 在项目中的应用，我们分别使用不同的 API 接口实现对应的业务需求，按照业务划分如下：

- Watch 机制来实现配置变更下发，任务下发的实时获取机制。
- 脑裂问题在 etcd 中不存在，etcd 集群的选举，只有投票达到 N/2+1 以上，才会选做 Leader，来保证数据一致性。另外一个网络分区的 Member 节点将无主。
- 语言亲和性，也是 Golang 开发的，Client SDK 库稳定可用。
- Key 存储的数据结构支持范围性的 Key 操作。
- User、Role 权限设定不同读写权限，来控制 Key 操作，避免其他客户端修改其他 Key 的信息。
- TLS 来保证通道信息传递安全。
- Txn 分布式事务 API 配合 Compare API 来确定主机上线的 Key 唯一性。
- Lease 租约机制，过期 Key 释放，更好的感知主机下线信息。
- etcd 底层 Key 的存储为 BTree 结构，查找时间复杂度为 O（$\log n$），百万级甚至千万级 Key 的查找耗时区别不大。

etcd Key 的设计

前缀按角色设定：

- Server 配置下发使用 /hids/server/config/{hostname}/master。
- Agent 注册上线使用 /hids/agent/master/{hostname}。
- Plugin 配置获取使用 /hids/agent/config/{hostname}/plugin/ID/conf_name。

Server Watch /hids/server/config/{hostname}/master，实现 Agent 主机上线的瞬间感知。Agent Watch /hids/server/config/{hostname}/ 来获取配置变更，

任务下发。Agent 注册的 Key 带有 Lease Id，并启用 keepalive，下线后瞬间感知。（异常下线，会有 1/3 的 keepalive 时间延迟）

关于 Key 的权限，根据不同前缀，设定不同 Role 权限。赋值给不同的 User，来实现对 Key 的权限控制。

etcd 集群管理

在 etcd 节点容灾考虑，考虑 DNS 故障时，节点会选择部署在多个城市，多个机房，以我们服务器机房选择来看，在大部分机房都有一个节点，综合承载需求，我们选择了 N 台服务器部署在个别重要机房，来满足负载、容灾需求。但对于 etcd 这种分布式一致性强的组件来说，每个写操作都需要 N/2−1 的节点确认变更，才会将写请求写入数据库中，再同步到各个节点，那么意味着节点越多，需要确认的网络请求越多，耗时越多，反而会影响集群节点性能。这点，我们后续将提升单个服务器性能，以及牺牲部分容灾性来提升集群处理速度。

客户端填写的 IP 列表，包含域名、IP。IP 用来规避 DNS 故障，域名用来做 Member 节点更新。最好不要使用 Discover 方案，避免对内网 DNS 服务器产生较大压力。

同时，在配置 etcd 节点的地址时，也要考虑到内网 DNS 故障的场景，地址填写会混合 IP、域名两种形式。

1. IP 的地址，便于规避内网 DNS 故障。
2. 域名形式，便于做个别节点更替或扩容。

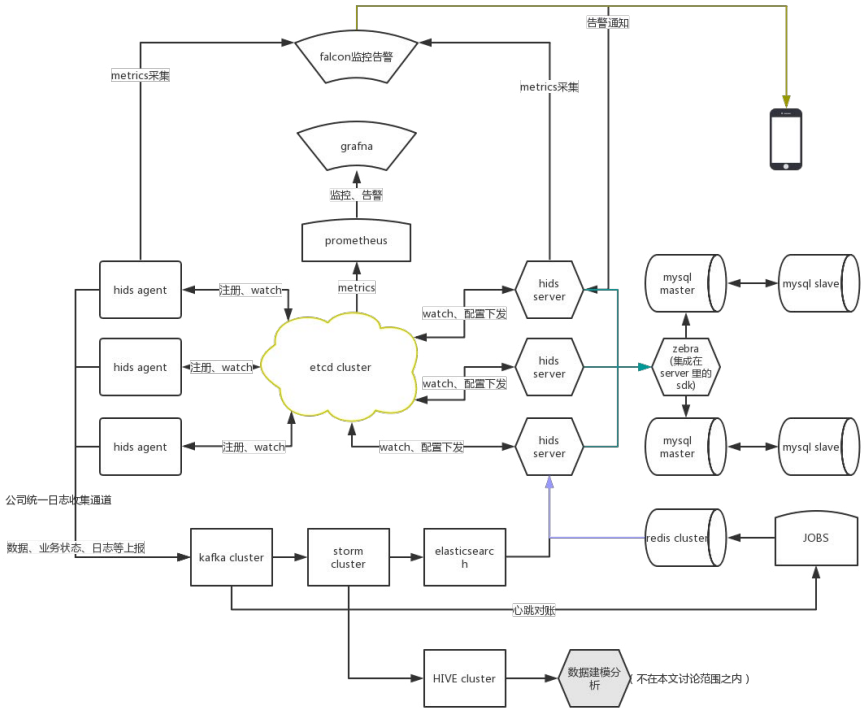
我们在设计产品架构时，为了安全性，开启了 TLS 证书认证，当节点变更时，证书的生成也同样要考虑到上面两种方案的影响，证书里需要包含固定 IP，以及 DNS 域名范围的两种格式。

etcd Cluster 节点扩容

节点扩容，官方手册上也有完整的方案，etcd 的 Client 里实现了健康检测与故障迁移，能自动的迁移到节点 IP 列表中的其他可用 IP。也能定时更新 etcd Node

List，对于 etcd Cluster 的集群节点变更来说，不存在问题。需要我们注意的是，TLS 证书的兼容。

分布式 HIDS 集群架构图



hids-cluster-architecture

集群核心组件高可用，所有 Agent、Server 都依赖集群，都可以无缝扩展，且不影响整个集群的稳定性。即使 Server 全部宕机，也不影响所有 Agent 的继续工作。

在以后 Server 版本升级时，Agent 不会中断，也不会带来雪崩式的影响。etcd 集群可以做到单节点升级，一直到整个集群升级，各个组件全都解耦。

编程语言选择

考虑到公司服务器量大，业务复杂，需求环境多变，操作系统可能包括各种

Linux 以及 Windows 等。为了保证系统的兼容性，我们选择了 Golang 作为开发语言，它具备以下特点：

1. 可以静态编译，直接通过 syscall 来运行，不依赖 libc，兼容性高，可以在所有 Linux 上执行，部署便捷。
2. 静态编译语言，能将简单的错误在编译前就发现。
3. 具备良好的 GC 机制，占用系统资源少，开发成本低。
4. 容器化的很多产品都是 Golang 编写，比如 Kubernetes、Docker 等。
5. etcd 项目也是 Golang 编写，类库、测试用例可以直接用，SDK 支持快速。
6. 良好的 CSP 并发模型支持，高效的协程调度机制。

产品架构大方向

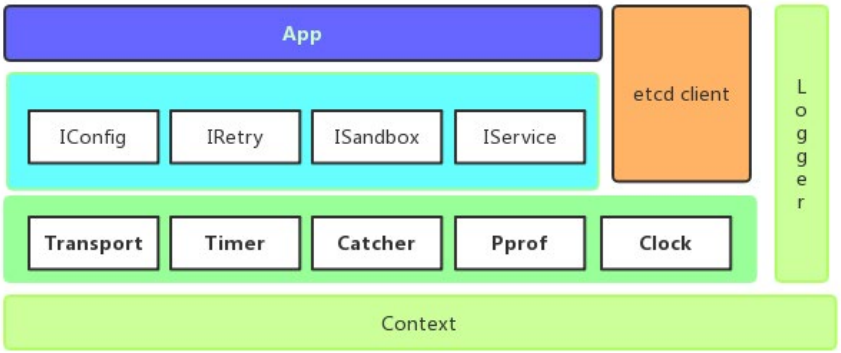
HIDS 产品研发完成后，部署的服务都运行着各种业务的服务器，业务的重要性排在第一，我们产品的功能排在后面。为此，确定了几个产品的大方向：

- 高可用，数据一致，可横向扩展。
- 容灾性好，能应对机房级的网络故障。
- 兼容性好，只维护一个版本的 Agent。
- 依赖低，不依赖任何动态链接库。
- 侵入性低，不做 Hook，不做系统类库更改。
- 熔断降级可靠，宁可自己挂掉，也不影响业务 。

产品实现

篇幅限制，仅讨论框架设计、熔断限流、监控告警、自我恢复以及产品实现上的主进程与进程监控。

框架设计



hids-framework

如上图，在框架的设计上，封装常用类库，抽象化定义 Interface，剥离 etcd Client，全局化 Logger，抽象化 App 的启动、退出方法。使得各模块（以下简称 App）只需要实现自己的业务即可，可以方便快捷的进行逻辑编写，无需关心底层实现、配置来源、重试次数、熔断方案等等。

沙箱隔离

考虑到子进程不能无限的增长下去，那么必然有一个进程包含多个模块的功能，各 App 之间既能使用公用底层组件（Logger、etcd Client 等），又能让彼此之间互不影响，这里进行了沙箱化处理，各个属性对象仅在各 App 的 sandbox 里生效。同样能实现了 App 进程的性能熔断，停止所有的业务逻辑功能，但又能具有基本的自我恢复功能。

IConfig

对各 App 的配置抽象化处理，实现 IConfig 的共有方法接口，用于对配置的函数调用，比如 Check 的检测方法，检测配置合法性，检测配置的最大值、最小值范围，规避使用人员配置不在合理范围内的情况，从而避免带来的风险。

框架底层用 Reflect 来处理 JSON 配置，解析读取填写的配置项，跟 Config 对象对比，填充到对应 Struct 的属性上，允许 JSON 配置里只填写变化的配置，没填

写的配置项，则使用 Config 对应 Struct 的默认配置。便于灵活处理配置信息。

```
type IConfig interface {
      Check() error //检测配置合法性
}

func ConfigLoad(confByte []byte, config IConfig) (IConfig, error) {
...
// 反射生成临时的 IConfig
      var confTmp IConfig
      confTmp = reflect.New(reflect.ValueOf(config).Elem().Type()).
Interface().(IConfig)
...

      // 反射 confTmp 的属性
      confTmpReflect := reflect.TypeOf(confTmp).Elem()
      confTmpReflectV := reflect.ValueOf(confTmp).Elem()

      // 反射 config IConfig
      configReflect := reflect.TypeOf(config).Elem()
      configReflectV := reflect.ValueOf(config).Elem()
...
      for i = 0; i < num; i++ {
    // 遍历处理每个 Field
              envStructTmp := configReflect.Field(i)
      // 根据配置中的项，来覆盖默认值
      if envStructTmp.Type == confStructTmp.Type {
        configReflectV.FieldByName(envStructTmp.Name).
Set(confTmpReflectV.Field(i))
```

Timer、Clock 调度

在业务数据产生时，很多地方需要记录时间，时间的获取也会产生很多系统调用。尤其是在每秒钟产生成千上万个事件，这些事件都需要调用获取时间接口，进行 clock_gettime 等系统调用，会大大增加系统 CPU 负载。 而很多事件产生时间的准确性要求不高，精确到秒，或者几百个毫秒即可，那么框架里实现了一个颗粒度符合需求的（比如 100ms、200ms、或者 1s 等）间隔时间更新的时钟，即满足事件对时间的需求，又减少了系统调用。

同样，在有些 Ticker 场景中，Ticker 的间隔颗粒要求不高时，也可以合并成一个 Ticker，减少对 CPU 时钟的调用。

Catcher

在多协程场景下，会用到很多协程来处理程序，对于个别协程的 panic 错误，上层线程要有一个良好的捕获机制，能将协程错误抛出去，并能恢复运行，不要让进程崩溃退出，提高程序的稳定性。

抽象接口

框架底层抽象化封装 Sandbox 的 Init、Run、Shutdown 接口，规范各 App 的对外接口，让 App 的初始化、运行、停止等操作都标准化。App 的模块业务逻辑，不需要关注 PID 文件管理，不关注与集群通讯，不关心与父进程通讯等通用操作，只需要实现自己的业务逻辑即可。App 与框架的统一控制，采用 Context 包以及 Sync.Cond 等条件锁作为同步控制条件，来同步 App 与框架的生命周期，同步多协程之间同步，并实现 App 的安全退出，保证数据不丢失。

限流

网络 IO

- 限制数据上报速度。
- 队列存储数据任务列表。
- 大于队列长度数据丢弃。
- 丢弃数据总数计数。
- 计数信息作为心跳状态数据上报到日志中心，用于数据对账。

磁盘 IO

程序运行日志，对日志级别划分，参考 /usr/include/sys/syslog.h：

- LOG_EMERG
- LOG_ALERT
- LOG_CRIT
- LOG_ERR
- LOG_WARNING

- LOG_NOTICE
- LOG_INFO
- LOG_DEBUG

在代码编写时，根据需求选用级别。级别越低日志量越大，重要程度越低，越不需要发送至日志中心，写入本地磁盘。那么在异常情况排查时，方便参考。

日志文件大小控制，分 2 个文件，每个文件不超过固定大小，比如 20M、50M 等。并且，对两个文件进行来回写，避免日志写满磁盘的情况。

IRetry

为了加强 Agent 的鲁棒性，不能因为某些 RPC 动作失败后导致整体功能不可用，一般会有重试功能。Agent 跟 etcd Cluster 也是 TCP 长连接（HTTP2），当节点重启更换或网络卡顿等异常时，Agent 会重连，那么重连的频率控制，不能是死循环般的重试。假设服务器内网交换机因内网流量较大产生抖动，触发了 Agent 重连机制，不断的重连又加重了交换机的负担，造成雪崩效应，这种设计必须要避免。 在每次重试后，需要做一定的回退机制，常见的指数级回退，比如如下设计，在规避雪崩场景下，又能保障 Agent 的鲁棒性，设定最大重试间隔，也避免了 Agent 失控的问题。

```
// 网络库重试 Interface
type INetRetry interface {
      // 开始连接函数
      Connect() error
      String() string
      // 获取最大重试次数
      GetMaxRetry() uint
     ...
}
// 底层实现
func (this *Context) Retry(netRetry INetRetry) error {
...
    maxRetries = netRetry.GetMaxRetry() // 最大重试次数
    hashMod = netRetry.GetHashMod()
for {
      if c.shutting {
              return errors.New("c.shutting is true...")
```

```
        }
        if maxRetries > 0 && retries >= maxRetries {
                c.logger.Debug("Abandoning %s after %d retries.",
netRetry.String(), retries)
                return errors.New(" 超过最大重试次数 ")
        }
...
        if e := netRetry.Connect(); e != nil {
                delay = 1 << retries
                if delay == 0 {
                        delay = 1
                }
                delay = delay * hashInterval
...
                c.logger.Emerg("Trying %s after %d seconds ,
retries:%d,error:%v", netRetry.String(), delay, retries, e)
                time.Sleep(time.Second * time.Duration(delay))
        }
...
}
```

事件拆分

百万台 IDC 规模的 Agent 部署，在任务执行、集群通讯或对宿主机产生资源影响时，务必要错峰进行，根据每台主机的唯一特征取模，拆分执行，避免造成雪崩效应。

监控告警

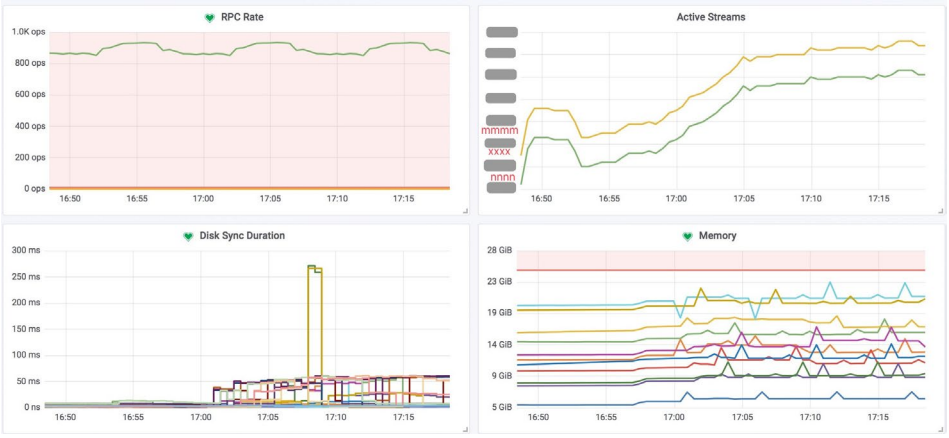
古时候，行军打仗时，提倡「兵马未动，粮草先行」，无疑是冷兵器时代决定胜负走向的重要因素。做产品也是，尤其是大型产品，要对自己运行状况有详细的掌控，做好监控告警，才能确保产品的成功。

对于 etcd 集群的监控，组件本身提供了 Metrics 数据输出接口，官方推荐了 Prometheus 来采集数据，使用 Grafana 来做聚合计算、图标绘制，我们做了 Alert 的接口开发，对接了公司的告警系统，实现 IM、短信、电话告警。

Agent 数量感知，依赖 Watch 数字，实时准确感知。

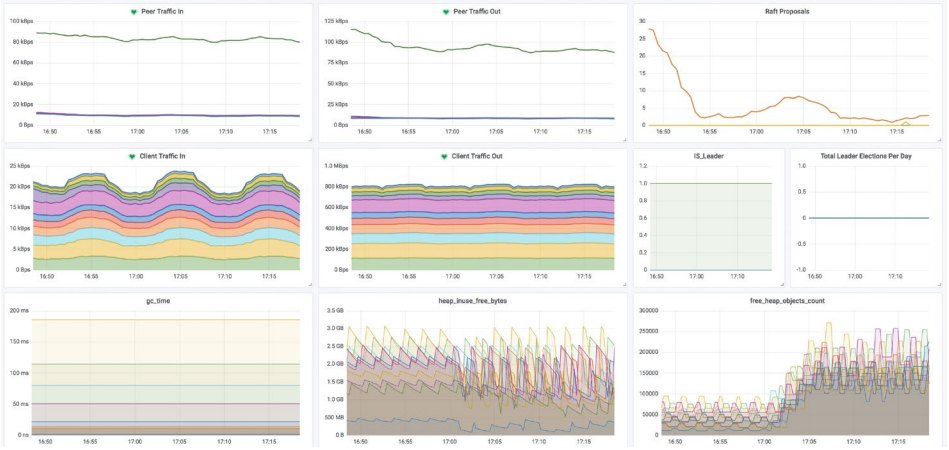
如下图，来自产品刚开始灰度时的某一时刻截图，Active Streams（即 etcd Watch 的 Key 数量）即为对应 Agent 数量，每次灰度的产品数量。因为该操作，是

Agent 直接与集群通讯，并且每个 Agent 只 Watch 一个 Key。且集群数据具备唯一性、一致性，远比心跳日志的处理要准确的多。



etcd-Grafana-Watcher-Monitor

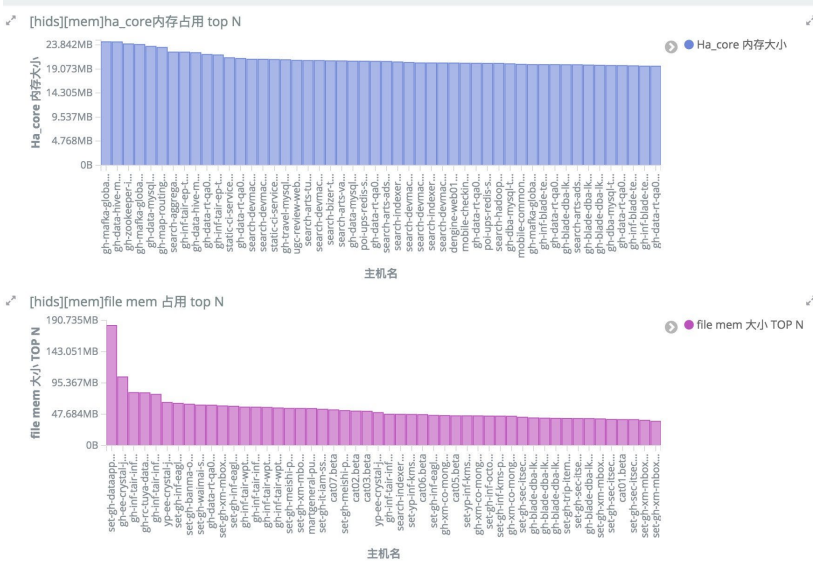
etcd 集群 Members 之间健康状况监控



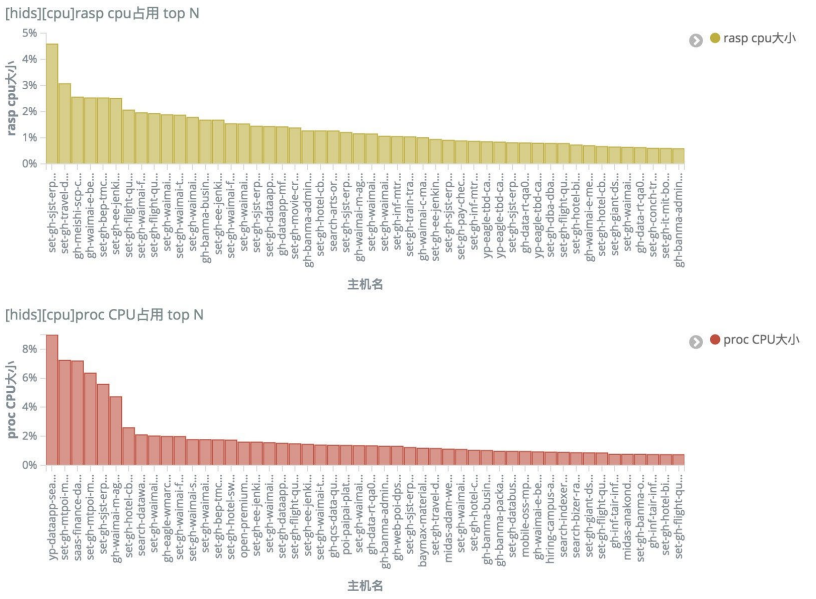
etcd-Grafana-GC-Heap-Objects

用于监控管理 etcd 集群的状况，包括 Member 节点之间数据同步，Leader 选举次数，投票发起次数，各节点的内存申请状况，GC 情况等，对集群的健康状况做全面掌控。

程序运行状态监控告警



agent-mem-es



agent-cpu-es

全量监控 Agent 的资源占用情况，统计每天使用最大 CPU\ 内存的主机 Agent，确定问题的影响范围，及时做策略调整，避免影响到业务服务的运行。并在后续版本上逐步做调整优化。

百万台服务器，日志告警量非常大，这个级别的告警信息的筛选、聚合是必不可少的。减少无用告警，让研发运维人员疲于奔命，也避免无用告警导致研发人员放松了警惕，前期忽略个例告警，先解决主要矛盾。

- 告警信息分级，告警信息细分 ID。
- 根据告警级别过滤，根据告警 ID 聚合告警，来发现同类型错误。
- 根据告警信息的所在机房、项目组、产品线等维度来聚合告警，来发现同类型错误。

数据采集告警

- 单机数据数据大小、总量的历史数据对比告警。
- 按机房、项目组、产品线等维度的大小、总量等维度的历史数据对比告警。
- 数据采集大小、总量的对账功能，判断经过一系列处理流程的日志是否丢失的监控告警。

熔断

- 针对单机 Agent 使用资源大小的阈值熔断，CPU 使用率，连续 N 次触发大于等于 5%，则进行保护性熔断，退出所有业务逻辑，以保护主机的业务程序优先。
- Master 进程进入空闲状态，等待第二次时间 Ticker 到来，决定是否恢复运行。
- 各个 App 基于业务层面的监控熔断策略。

灰度管理

在前面的配置管理中的 etcd Key 设计里，已经细分到每个主机（即每个 Agent）一个 Key。那么，服务端的管理，只要区分该主机所属机房、环境、群组、产品线即可，那么，我们的管理 Agent 的颗粒度可以精确到每个主机，也就是支持任意纬度

的灰度发布管理与命令下发。

数据上报通道

组件名为 log_agent ，是公司内部统一日志上报组件，会部署在每一台 VM、Docker 上。主机上所有业务均可将日志发送至该组件。 log_agent 会将日志上报到 Kafka 集群中，经过处理后，落入 Hive 集群中。（细节不在本篇讨论范围）

主进程

主进程实现跟 etcd 集群通信，管理整个 Agent 的配置下发与命令下发；管理各个子模块的启动与停止；管理各个子模块的 CPU、内存占用情况，对资源超标进行进行熔断处理，让出资源，保证业务进程的运行。

插件化管理其他模块，多进程模式，便于提高产品灵活性，可更简便的更新启动子模块，不会因为个别模块插件的功能、BUG 导致整个 Agent 崩溃。

进程监控

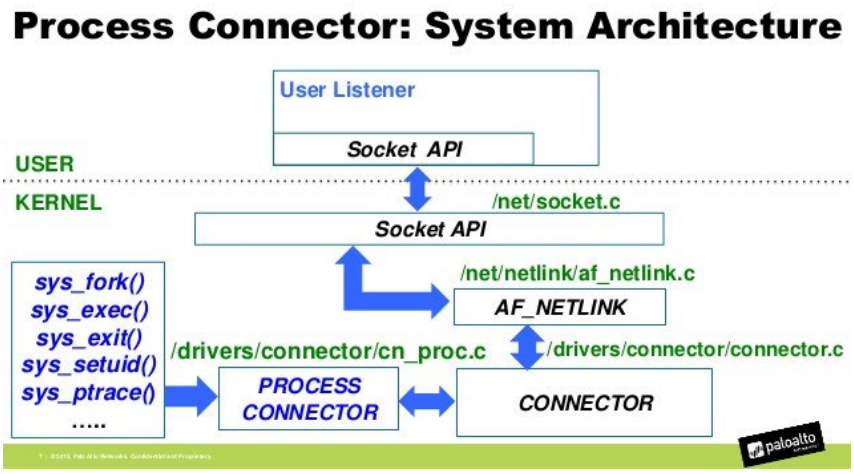
方案选择

我们在研发这产品时，做了很多关于 linux 进程创建监控的调研，不限于安全产品，大约有下面三种技术方案：

方案	Docker兼容性	开发难度	数据准确性	系统侵入性
cn_proc	不支持 Docker	一般	存在内核拿到的 PID，在 /proc/ 下丢失的情况	无
Audit	不支持 Docker	一般	同 cn_proc	弱，但依赖 Auditd
Hook	定制	高	精确	强

对于公司的所有服务器来说，几十万台都是已经在运行的服务器，新上的任何产品，都尽量避免对服务器有影响，更何况是所有服务器都要部署的 Agent。 意味着我们在选择系统侵入性来说，优先选择最小侵入性的方案。

对于 Netlink 的方案原理，可以参考这张图（来自 :kernel-proc-connector-and-containers）



process-connector

系统侵入性比较

- cn_proc 跟 Autid 在「系统侵入性」和「数据准确性」来说，cn_proc 方案更好，而且使用 CPU、内存等资源情况，更可控。
- Hook 的方案，对系统侵入性太高了，尤其是这种最底层做 HOOK syscall 的做法，万一测试不充分，在特定环境下，有一定的概率会出现 Bug，而在百万 IDC 的规模下，这将成为大面积事件，可能会造成重大事故。

兼容性上比较

- cn_proc 不兼容 Docker，这个可以在宿主机上部署来解决。
- Hook 的方案，需要针对每种 Linux 的发行版做定制，维护成本较高，且不符合长远目标（收购外部公司时遇到各式各样操作系统问题）

数据准确性比较

在大量 PID 创建的场景，比如 Docker 的宿主机上，内核返回 PID 时，因为 PID 返回非常多非常快，很多进程启动后，立刻消失了，另外一个线程都还没去读取 /proc/，进程都丢失了，场景常出现在 Bash 执行某些命令。

最终，我们选择 Linux Kernel Netlink 接口的 cn_proc 指令作为我们进程监控

方案，借助对 Bash 命令的收集，作为该方案的补充。当然，仍然存在丢数据的情况，但我们为了系统稳定性，产品侵入性低等业务需求，牺牲了一些安全性上的保障。

对于 Docker 的场景，采用宿主机运行，捕获数据，关联到 Docker 容器，上报到日志中心的做法来实现。

遇到的问题

内核 Netlink 发送数据卡住

内核返回数据太快，用户态 ParseNetlinkMessage 解析读取太慢，导致用户态网络 Buff 占满，内核不再发送数据给用户态，进程空闲。对于这个问题，我们在用户态做了队列控制，确保解析时间的问题不会影响到内核发送数据。对于队列的长度，我们做了定值限制，生产速度大于消费速度的话，可以丢弃一些数据，来保证业务正常运行，并且来控制进程的内存增长问题。

疑似“内存泄露”问题

在一台 Docker 的宿主机上，运行了 50 个 Docker 实例，每个 Docker 都运行了复杂的业务场景，频繁的创建进程，在最初的产品实现上，启动时大约 10M 内存占用，一天后达到 200M 的情况。

经过我们 Debug 分析发现，在 ParseNetlinkMessage 处理内核发出的消息时，PID 频繁创建带来内存频繁申请，对象频繁实例化，占用大量内存。同时，在 Golang GC 时，扫描、清理动作带来大量 CPU 消耗。在代码中，发现对于 **linux/connector.h** 里的 struct cb_msg、**linux/cn_proc.h** 里的 struct proc_event 结构体频繁创建，带来内存申请等问题，以及 Golang 的 GC 特性，内存申请后，不会在 GC 时立刻归还操作系统，而是在后台任务里，逐渐的归还到操作系统，见：debug.FreeOSMemory

> FreeOSMemory forces a garbage collection followed by an attempt to return as much memory to the operating system as possible. (Even if this is not called, the runtime gradually returns memory to the operating system in a background task.)

但在这个业务场景里，大量频繁的创建 PID，频繁的申请内存，创建对象，那么

申请速度远远大于释放速度，自然内存就一直堆积。

从文档中可以看出，FreeOSMemory 的方法可以将内存归还给操作系统，但我们并没有采用这种方案，因为它治标不治本，没法解决内存频繁申请频繁创建的问题，也不能降低 CPU 使用率。

为了解决这个问题，我们采用了 sync.Pool 的内置对象池方式，来复用回收对象，避免对象频繁创建，减少内存占用情况，在针对几个频繁创建的对象做对象池化后，同样的测试环境，内存稳定控制在 15M 左右。

大量对象的复用，也减少了对象的数量，同样的，在 Golang GC 运行时，也减少了对象的扫描数量、回收数量，降低了 CPU 使用率。

项目进展

在产品的研发过程中，也遇到了一些问题，比如：

1. etcd Client Lease Keepalive 的 Bug。
2. Agent 进程资源限制的 Cgroup 触发几次内核 Bug。
3. Docker 宿主机上瞬时大量进程创建的性能问题。
4. 网络监控模块在处理 Nginx 反向代理时，动辄几十万 TCP 链接的网络数据获取压力。
5. 个别进程打开了 10W 以上的 fd。

方法一定比困难多，但方法不是拍脑袋想出来的，一定要深入探索问题的根本原因，找到系统性的修复方法，具备高可用、高性能、监控告警、熔断限流等功能后，对于出现的问题，能够提前发现，将故障影响最小化，提前做处理。在应对产品运营过程中遇到的各种问题时，逢山开路，遇水搭桥，都可以从容的应对。

经过我们一年的努力，已经部署了除了个别特殊业务线之外的其他所有服务器，数量达几十万台，产品稳定运行。在数据完整性、准确性上，还有待提高，在精细化运营上，需要多做改进。

本篇更多的是研发角度上软件架构上的设计，关于安全事件分析、数据建模、运营策略等方面的经验和技巧，未来将会由其他同学进行分享，敬请期待。

总结

我们在研发这款产品过程中，也看到了网上开源了几款同类产品，也了解了他们的设计思路，发现很多产品都是把主要方向放在了单个模块的实现上，而忽略了产品架构上的重要性。

比如，有的产品使用了 syscall hook 这种侵入性高的方案来保障数据完整性，使得对系统侵入性非常高，Hook 代码的稳定性，也严重影响了操作系统内核的稳定。同时，Hook 代码也缺少了监控熔断的措施，在几十万服务器规模的场景下部署，潜在的风险可能让安全部门无法接受，甚至是致命的。

这种设计，可能在服务器量级小时，对于出现的问题多花点时间也能逐个进行维护，但应对几十万甚至上百万台服务器时，对维护成本、稳定性、监控熔断等都是很大的技术挑战。同时，在研发上，也很难实现产品的快速迭代，而这种方式带来的影响，几乎都会导致内核宕机之类致命问题。这种事故，使用服务器的业务方很难进行接受，势必会影响产品的研发速度、推进速度；影响同事（SRE 运维等）对产品的信心，进而对后续产品的推进带来很大的阻力。

以上是笔者站在研发角度，从可用性、可靠性、可控性、监控熔断等角度做的架构设计与框架设计，分享的产品研发思路。

笔者认为大规模的服务器安全防护产品，首先需要考虑的是架构的稳定性、监控告警的实时性、熔断限流的准确性等因素，其次再考虑安全数据的完整性、检测方案的可靠性、检测模型的精确性等因素。

九层之台，起于累土。只有打好基础，才能运筹帷幄，决胜千里之外。

参考资料

1. https://en.wikipedia.org/wiki/CAP_theorem
2. https://www.consul.io/intro/vs/serf.html
3. https://golang.org/src/runtime/debug/garbage.go?h=FreeOSMemory#L99
4. https://www.ibm.com/developerworks/cn/linux/l-connector/
5. https://www.kernel.org/doc/
6. https://coreos.com/etcd/docs/latest/

作者简介

陈驰，美团点评技术专家，2017 年加入美团，十年以上互联网产品研发经验，专注于分布式系统架构设计，目前主要从事安全防御产品研发工作。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级 IDC 规模攻防对抗的经验。安全部也不乏 CVE“挖掘圣手”，有受邀在 Black Hat 等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web 防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级 IDC 规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化 / 容器层、Server 软件层（内核态 / 用户态）、语言虚拟机层（JVM/JS V8）、Web 应用层、数据访问层等，并能够基于“大数据 + 机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募 Web& 二进制攻防、后台 & 系统开发、机器学习 & 算法等各路小伙伴。

如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里：https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心 MTSRC 主页：security.meituan.com

Leaf：美团分布式 ID 生成服务开源

志桐

Leaf 是美团基础研发平台推出的一个分布式 ID 生成服务，名字取自德国哲学家、数学家莱布尼茨的一句话：“There are no two identical leaves in the world.”Leaf 具备高可靠、低延迟、全局唯一等特点。目前已经广泛应用于美团金融、美团外卖、美团酒旅等多个部门。具体的技术细节，可参考此前美团技术博客的一篇文章：《Leaf 美团分布式 ID 生成服务》。近日，Leaf 项目已经在 Github 上开源：https://github.com/Meituan-Dianping/Leaf，希望能和更多的技术同行一起交流、共建。

Leaf 特性

Leaf 在设计之初就秉承着几点要求：

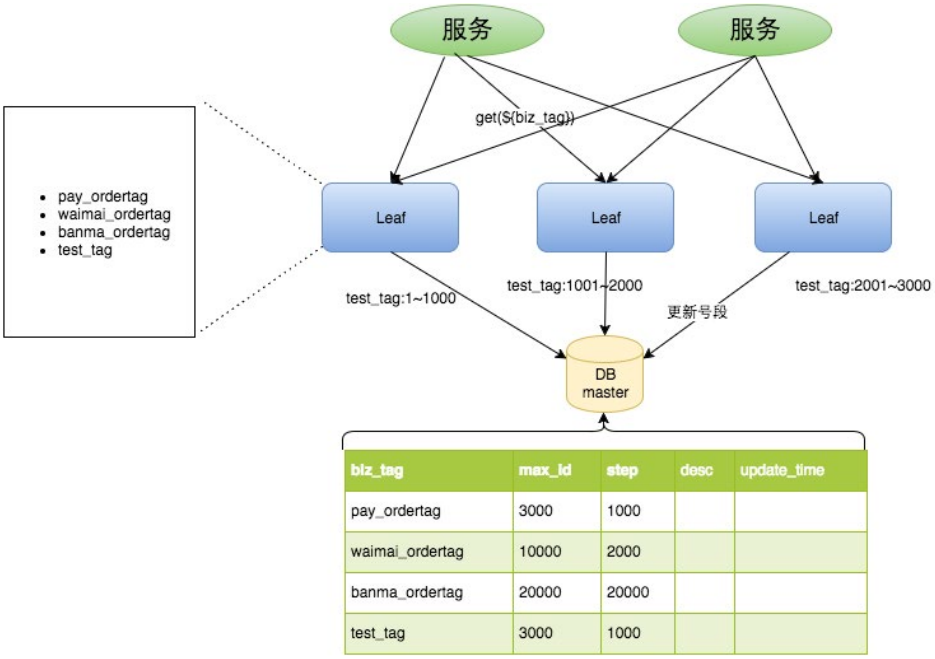
1. 全局唯一，绝对不会出现重复的 ID，且 ID 整体趋势递增。
2. 高可用，服务完全基于分布式架构，即使 MySQL 宕机，也能容忍一段时间的数据库不可用。
3. 高并发低延时，在 CentOS 4C8G 的虚拟机上，远程调用 QPS 可达 5W+，TP99 在 1ms 内。
4. 接入简单，直接通过公司 RPC 服务或者 HTTP 调用即可接入。

Leaf 诞生

Leaf 第一个版本采用了预分发的方式生成 ID，即可以在 DB 之上挂 N 个 Server，每个 Server 启动时，都会去 DB 拿固定长度的 ID List。这样就做到了完全基于分布式的架构，同时因为 ID 是由内存分发，所以也可以做到很高效。接下来是数据持久化问题，Leaf 每次去 DB 拿固定长度的 ID List，然后把最大的 ID 持久化下来，也就是并非每个 ID 都做持久化，仅仅持久化一批 ID 中最大的那一个。这个方式

有点像游戏里的定期存档功能，只不过存档的是未来某个时间下发给用户的 ID，这样极大地减轻了 DB 持久化的压力。

整个服务的具体处理过程如下：



biz_tag	max_id	step	desc	update_time
pay_ordertag	3000	1000		
waimai_ordertag	10000	2000		
banma_ordertag	20000	20000		
test_tag	3000	1000		

- Leaf Server 1：从 DB 加载号段 [1，1000]。
- Leaf Server 2：从 DB 加载号段 [1001，2000]。
- Leaf Server 3：从 DB 加载号段 [2001，3000]。

用户通过 Round-robin 的方式调用 Leaf Server 的各个服务，所以某一个 Client 获取到的 ID 序列可能是：1，1001，2001，2，1002，2002……也可能是：1，2，1001，2001，2002，2003，3，4……当某个 Leaf Server 号段用完之后，下一次请求就会从 DB 中加载新的号段，这样保证了每次加载的号段是递增的。

Leaf 数据库中的号段表格式如下：

```
+-------------+--------------+------+-----+-------------------+------
-----------------------+
| Field       | Type         | Null | Key | Default           | Extra
|
```

```
+-------------+--------------+------+-----+-------------------+------
----------------------+
| biz_tag     | varchar(128) | NO   | PRI |                   |
|
| max_id      | bigint(20)   | NO   |     | 1                 |
|
| step        | int(11)      | NO   |     | NULL              |
|
| desc        | varchar(256) | YES  |     | NULL              |
|
| update_time | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update
CURRENT_
TIMESTAMP |
+-------------+--------------+------+-----+-------------------+------
----------------------+
```

Leaf Server 加载号段的 SQL 语句如下：

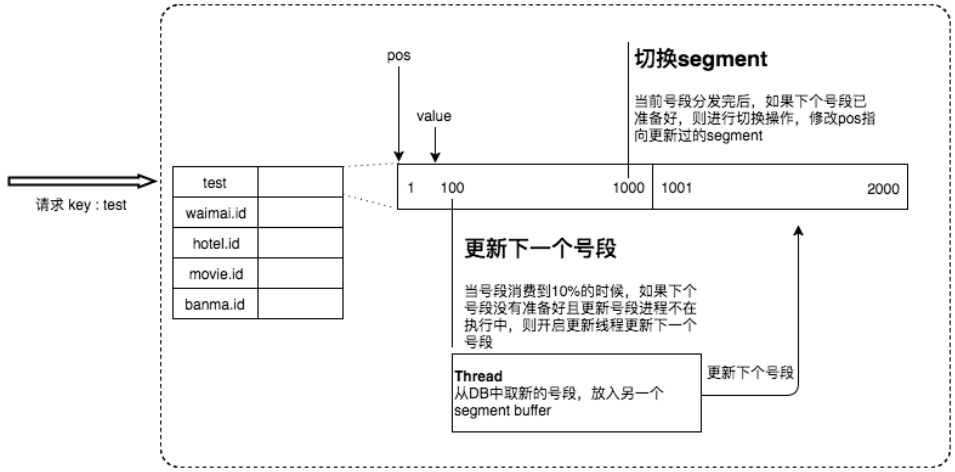
```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

整体上，V1 版本实现比较简单，主要是为了尽快解决业务层 DB 压力的问题，而快速迭代出的一个版本。因而在生产环境中，也发现了些问题。比如：

1. 在更新 DB 的时候会出现耗时尖刺，系统最大耗时取决于更新 DB 号段的时间。
2. 当更新 DB 号段的时候，如果 DB 宕机或者发生主从切换，会导致一段时间的服务不可用。

Leaf 双 Buffer 优化

为了解决这两个问题，Leaf 采用了异步更新的策略，同时通过双 Buffer 的方式，保证无论何时 DB 出现问题，都能有一个 Buffer 的号段可以正常对外提供服务，只要 DB 在一个 Buffer 的下发的周期内恢复，就不会影响整个 Leaf 的可用性。



这个版本代码在线上稳定运行了半年左右，Leaf 又遇到了新的问题：

1. 号段长度始终是固定的，假如 Leaf 本来能在 DB 不可用的情况下，维持 10 分钟正常工作，那么如果流量增加 10 倍就只能维持 1 分钟正常工作了。
2. 号段长度设置的过长，导致缓存中的号段迟迟消耗不完，进而导致更新 DB 的新号段与前一次下发的号段 ID 跨度过大。

Leaf 动态调整 Step

假设**服务 QPS 为 Q，号段长度为 L，号段更新周期为 T**，那么 **Q * T = L**。最开始 L 长度是固定的，导致随着 Q 的增长，T 会越来越小。但是 Leaf 本质的需求是**希望 T 是固定的**。那么如果 L 可以和 Q 正相关的话，T 就可以趋近一个定值了。所以 Leaf 每次更新号段的时候，根据上一次更新号段的周期 T 和号段长度 step，来决定下一次的号段长度 nextStep：

- T < 15min，nextStep = step * 2
- 15min < T < 30min，nextStep = step
- T > 30min，nextStep = step / 2

至此，满足了号段消耗稳定趋于某个时间区间的需求。当然，面对瞬时流量几

十、几百倍的暴增，该种方案仍不能满足可以容忍数据库在一段时间不可用、系统仍能稳定运行的需求。因为本质上来讲，Leaf 虽然在 DB 层做了些容错方案，但是号段方式的 ID 下发，最终还是需要强依赖 DB。

MySQL 高可用

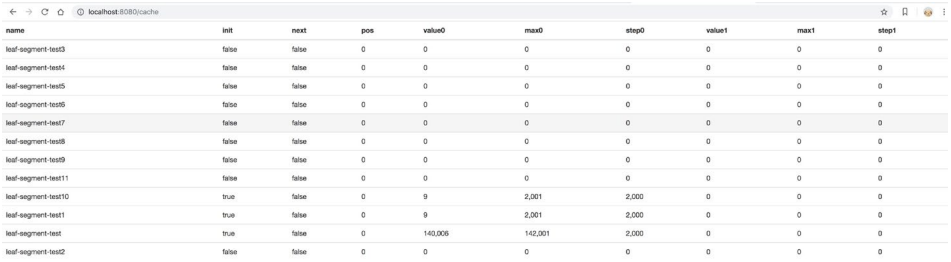
在 MySQL 这一层，Leaf 目前采取了半同步的方式同步数据，通过公司 DB 中间件 Zebra 加 MHA 做的主从切换。未来追求完全的强一致，会考虑切换到 MySQL Group Replication。

现阶段由于公司数据库强一致的特性还在演进中，Leaf 采用了一个临时方案来保证机房断网场景下的数据一致性：

- 多机房部署数据库，每个机房一个实例，保证都是跨机房同步数据。
- 半同步超时时间设置到无限大，防止半同步方式退化为异步复制。

Leaf 监控

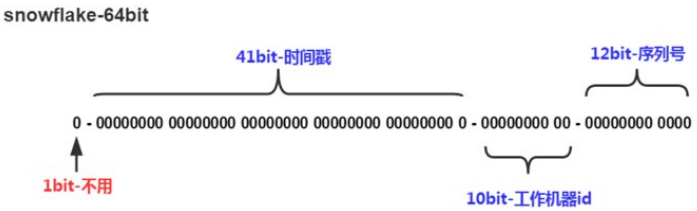
针对服务自身的监控，Leaf 提供了 Web 层的内存数据映射界面，可以实时看到所有号段的下发状态。比如每个号段双 buffer 的使用情况，当前 ID 下发到了哪个位置等信息都可以在 Web 界面上查看。



name	init	next	pos	value0	max0	step0	value1	max1	step1
leaf-segment-test3	false	false	0	0	0	0	0	0	0
leaf-segment-test4	false	false	0	0	0	0	0	0	0
leaf-segment-test5	false	false	0	0	0	0	0	0	0
leaf-segment-test6	false	false	0	0	0	0	0	0	0
leaf-segment-test7	false	false	0	0	0	0	0	0	0
leaf-segment-test8	false	false	0	0	0	0	0	0	0
leaf-segment-test9	false	false	0	0	0	0	0	0	0
leaf-segment-test11	false	false	0	0	0	0	0	0	0
leaf-segment-test10	true	false	0	9	2,001	2,000	0	0	0
leaf-segment-test1	true	false	0	9	2,001	2,000	0	0	0
leaf-segment-test	true	false	0	140,006	142,001	2,000	0	0	0
leaf-segment-test2	false	false	0	0	0	0	0	0	0

Leaf Snowflake

Snowflake，Twitter 开源的一种分布式 ID 生成算法。基于 64 位数实现，下图为 Snowflake 算法的 ID 构成图。



- 第 1 位置为 0。
- 第 2-42 位是相对时间戳，通过当前时间戳减去一个固定的历史时间戳生成。
- 第 43-52 位是机器号 workerID，每个 Server 的机器 ID 不同。
- 第 53-64 位是自增 ID。

这样通过时间 + 机器号 + 自增 ID 的组合来实现了完全分布式的 ID 下发。

在这里，Leaf 提供了 Java 版本的实现，同时对 Zookeeper 生成机器号做了弱依赖处理，即使 Zookeeper 有问题，也不会影响服务。Leaf 在第一次从 Zookeeper 拿取 workerID 后，会在本机文件系统上缓存一个 workerID 文件。即使 ZooKeeper 出现问题，同时恰好机器也在重启，也能保证服务的正常运行。这样做到了对第三方组件的弱依赖，一定程度上提高了 SLA。

未来规划

- 号段加载优化：Leaf 目前重启后的第一次请求还是会同步加载 MySQL，之所以这么做而非服务初始化加载号段的原因，主要是 MySQL 中的 Leaf Key 并非一定都被这个 Leaf 服务节点所加载，如果每个 Leaf 节点都在初始化加载所有的 Leaf Key 会导致号段的大量浪费。因此，未来会在 Leaf 服务 Shutdown 时，备份这个服务节点近一天使用过的 Leaf Key 列表，这样重启后会预先从 MySQL 加载 Key List 中的号段。
- 单调递增：简易的方式，是只要保证同一时间、同一个 Leaf Key 都从一个 Leaf 服务节点获取 ID，即可保证递增。需要注意的问题是 Leaf 服务节点切换时，旧 Leaf 服务用过的号段需要废弃。路由逻辑，可采用主备的模型或者每个 Leaf Key 配置路由表的方式来实现。

关于开源

分布式 ID 生成的方案有很多种，Leaf 开源版本提供了两种 ID 的生成方式：

- 号段模式：低位趋势增长，较少的 ID 号段浪费，能够容忍 MySQL 的短时间不可用。
- Snowflake 模式：完全分布式，ID 有语义。

读者可以按需选择适合自身业务场景的 ID 下发方式。希望美团的方案能给予大家一些帮助，同时也希望各位能够一起交流、共建。

Leaf 项目 Github 地址：https://github.com/Meituan-Dianping/Leaf。

如有任何疑问和问题，欢迎提交至 Github issues。

美团大规模微服务通信框架及治理体系 OCTO 核心组件开源

舒超　张翔

微服务通信框架及治理平台 OCTO 作为美团基础架构设施的重要组成部分，目前已广泛应用于公司技术线，稳定承载上万应用、日均支撑千亿级的调用。业务基于 OCTO 提供的标准化技术方案，能够轻松实现服务注册 / 发现、负载均衡、容错处理、降级熔断、灰度发布、调用数据可视化等服务治理功能。

现在我们将 OCTO 的核心组件 OCTO-RPC、OCTO-NS、OCTO-Portal 开源，欢迎大家使用和共建。OCTO-RPC、OCTO-NS、OCTO-Portal 深入了解。

背景

OCTO 项目始于 2014 年底，当时美团正处在新业务拓展期，服务数量不断增长，服务间调用拓扑日益复杂，逐渐暴露出了一些典型的问题：

- 研发效率低：缺乏标准的服务治理框架和组件，不少团队“重复造轮子”造成资源浪费，研发质量参差不齐，系统整体可用性不高。
- 运维成本高：缺乏完善、便捷的体系化运维手段，难以进行准确的监控告警，难以对涉及多级链路的故障快速定位。
- 服务运营难：服务指标数据收集困难，缺乏自动化深度分析，难以满足业务快速对指标进行评估决策的要求。

针对这些痛点问题，提升研发效率和质量，降低运营成本，对于支撑业务快速稳定发展显得尤为重要。因此，基础架构团队研发了公司级统一的分布式微服务通信框架及治理平台——OCTO。OCTO 是英文单词章鱼（Octopus）的缩写，章鱼众多的触手表征 OCTO 平台能触达治理海量的服务，涵盖服务治理领域的各个部分，因此取名。自平台推出以来，在全公司各业务线被广泛进行使用，显著提升了全公司的技术研发效率与运营质量，稳定支撑着美团业务的高速发展。

有别于传统的开源服务治理组件，OCTO 提供了体系化的服务治理方案，从通信框架到注册中心、从文件配置到埋点告警、从灰度链路到数据报表，立体化的提升微服务运营能力、赋能业务。此外，奉行“策略下沉”的设计思想，OCTO 剥离传统通信框架具备的服务治理策略功能，下沉到代理组件实现，有效提升通信框架的稳定性、降低业务系统额外开销。历经海量调用验证的 OCTO，依托低耦合、模块化、单元化系统架构，能够有效满足各类复杂业务场景需求，实现异地多活等容灾目标。

功能特性

针对暴露出的问题，OCTO 围绕包括定义、开发、测试、部署、运维、优化、下线在内的服务的全生命周期，打造了一系列组件和系统，方便研发同学专注于自身业务逻辑开发的同时，又能享受完善的服务治理功能。例如，我们在开发阶段提供了高性能、高可用、功能模块化的通信框架供业务便捷使用；在测试阶段提供了 SET 化、泳道、服务分组等各种灰度策略供业务无损试错；在运维阶段提供机器级、框架级、业务级等层级分明的监控告警供业务快速定位问题；在优化阶段提供了详尽的服务指标供业务自定义分析改进。

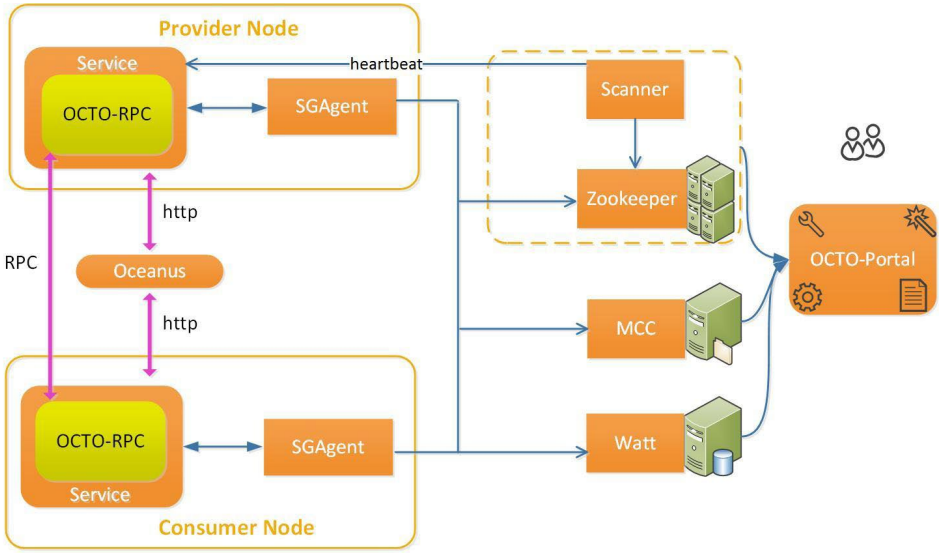
OCTO 主要功能特性包括但不限于：

- 命名服务：服务注册 / 发现。
- 服务管理：服务状态监测；服务启动、停止；服务负载均衡。
- 容错处理：实时屏蔽异常的服务，自动调配请求流量。
- 流量分发：灰度发布、节点动态流量分配等场景。
- 数据可视化：服务调用统计上报分析，提供清晰的数据图表展示，直观定位服务间依赖关系。
- 服务分组：支持服务动态自动归组与不同场景下的自定义分组，解决在多机房场景下跨机房调用穿透、沙盒测试等问题。
- 服务监控报警：支持服务与接口级别多指标、多维度的监控，支持多种报警方式。
- 统一配置管理：支持服务配置统一管理，灵活设置不同环境间差异，支持历史

版本，配置项变更后实时下发。

- 分布式服务跟踪：轻松诊断服务访问慢、异常抖动等问题。
- 过载保护：灵活定义分配给上游服务消费者的配额，当服务调用量超出最大阈值时，基于不同服务消费者进行 QoS 区分，触发流控进行过载保护。
- 服务访问控制：支持多粒度、多阶段的鉴权方式。

OCTO 整体架构



OCTO 架构图

- OCTO-RPC（开源 Java/C++）：分布式 RPC 通信框架，支持 Java、C++、Node.js 等多种语言。
- SGAgent：部署在各服务节点，承担服务注册 / 发现、动态路由解析、负载均衡、配置传输、性能数据上报等功能。
- Oceanus（待开源）：HTTP 定制化路由负载器，具体可参考 Oceanus：美团 HTTP 流量定制化路由的实践一文。
- OCTO-NS：包括 SDK（Java/C++）、基础代理 SGAgent、命名服务缓存 NSC、健康检查服务 Scanner 等组件，提供命名服务，服务注册等信息的存

储，服务状态检测的扫描等功能。

- Watt（待开源）：统计链路信息，计算服务各类指标作为监控报警依据。
- MCC（待开源）：统一配置中心，可进行服务配置的管理下发。
- OCTO-Portal：服务注册、管理、诊断、配置、配额等功能的一站式管理平台。

OCTO 开源组件

OCTO 首批开源的核心组件包括：分布式服务通信框架（OCTO-RPC）、服务注册中心（OCTO-NS）、服务治理平台（OCTO-Portal）。未来，我们还将持续开源更多的组件与功能。

分布式服务通信框架（OCTO-RPC）

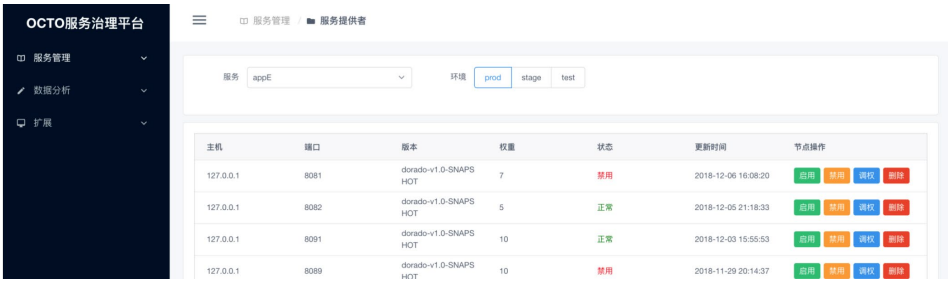
分布式服务通信框架是 OCTO 的重要组成部分，具备高性能、高可用、易扩展、易接入等特点，已覆盖美团 90% 以上的服务（Java/C++），支撑每天千亿级别的调用量。目前 Java 和 C++ 版本已经开源，更多技术实现详见：OCTO-RPC。

服务注册中心（OCTO-NS）

服务注册中心基于服务描述信息，实现服务注册 / 发现、配置管理、路由分组、负载均衡、健康检测等功能，搭配服务治理平台能够更便捷地进行服务节点数据的可视化运营。目前开源出来的子模块包括 SDK（Java/C++）、基础代理 SGAgent、命名服务缓存 NSC、健康检查服务 Scanner。更多技术实现详见：OCTO-NS。

服务治理平台（OCTO-Portal）

服务治理的一站式平台，为服务关注方提供服务节点管理、性能数据分析、全链路跟踪诊断等服务治理核心能力。更多技术实现详见：OCTO-Portal。



OCTO-Portal 示意图

关于开源

在过去四年中，OCTO 是美团在架构中间件领域研发的一个重要技术项目，在复杂多元业务、大规模并发调用的场景下已被充分验证。目前 OCTO 支撑了美团大规模微服务每天千亿级的调用，接口调用成功率达到了 99.999%，是全公司 SOA、服务治理的核心基础。我们期望通过将其开源，不断反馈给社区、贡献给行业。同时，我们希望在行业优秀工程师的帮助下，OCTO 平台能得到更快地升级更新迭代。欢迎大家多提宝贵意见和建议。

未来规划

为了进一步支撑美团业务飞速发展的需求，同时对标业界先进的服务治理理念与实践，未来一段时间内，OCTO 将在以下几方面规划演进：

1. **命名服务 AP 化：**弱化强一致性，聚焦异常状态下注册中心的可用性。
2. **框架反应式编程：**RPC 框架提供反应式编程支持，帮助业务构建高性能异步无阻塞的服务。
3. **Service Mesh：**将现有命名服务等功能与控制面 / 数据面结合，以服务网格的思路进一步演进 OCTO 服务治理体系。

作者简介

舒超，2015 年加入美团，高级技术专家，基础开发负责人。

张翔，2017 年加入美团，现负责公司命名服务和通信框架的研发。

招聘信息

美团 OCTO 服务治理团队诚招 C++/Java 高级工程师、技术专家。我们致力于研发公司级、业界领先的基础架构组件，研发范围涵盖分布式框架、命名服务、Service Mesh 等技术领域。欢迎有兴趣的同学投送简历至 tech@meituan.com。

美团下一代服务治理系统 OCTO2.0 的探索与实践

郭继东

本文根据美团基础架构部服务治理团队工程师郭继东在 2019 QCon（全球软件开发大会）上的演讲内容整理而成，主要阐述美团大规模治理体系结合 Service Mesh 演进的探索实践，希望对从事此领域的同学有所帮助。

一、OCTO 现状分析

OCTO 是美团标准化的服务治理基础设施，治理能力统一、性能及易用性表现优异、治理能力生态丰富，已广泛应用于美团各事业线。关于 OCTO 的现状，可整体概括为：

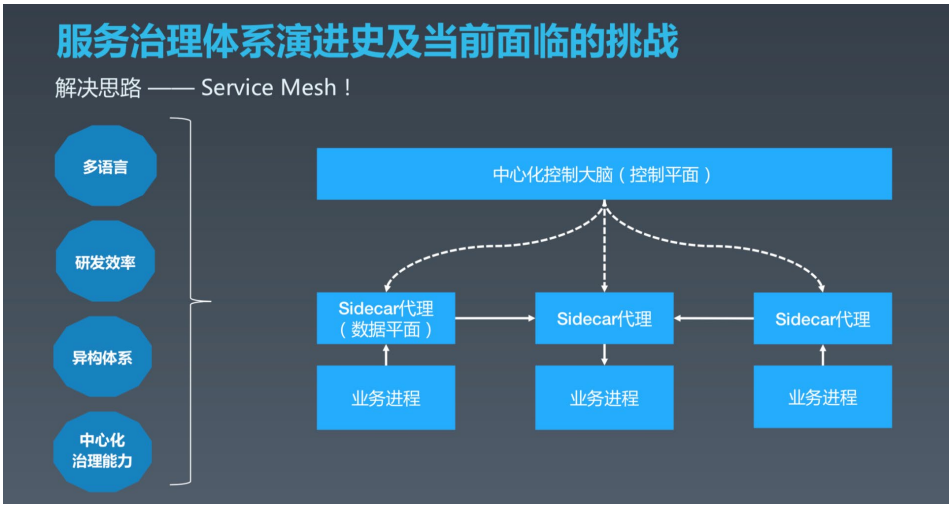
- 已成为美团高度统一的服务治理技术栈，覆盖了公司 90% 的应用，日均调用超万亿次。
- 经历过大规模的技术考验，覆盖数万个服务 / 数十万个节点。
- 协同周边治理生态提供的治理能力较为丰富，包含但不限于 SET 化、链路级复杂路由、全链路压测、鉴权加密、限流熔断等治理能力。
- 一套系统支撑着多元业务，覆盖公司所有事业线。

目前美团已经具备了相对完善的治理体系，但仍有较多的痛点及挑战：

- 对多语言支持不够好。美团技术栈使用的语言主要是 Java，占比到达 80% 以上，上面介绍的诸多治理能力也集中在 Java 体系。 但美团同时还有其他近 10 种后台服务语言在使用，这些语言的治理生态均十分薄弱，同时在多元业务的模式下必然会有增长的多语言需求，为每一种语言都建设一套完善的治理体系成本很高，也不太可能落地。
- 中间件和业务绑定在一起，制约着彼此迭代。一般来说，核心的治理能力主要由通信框架承载，虽然做到了逻辑隔离，但中间件的逻辑不可避免会和业务

在物理上耦合在一起。这种模式下，中间件引入 Bug 需要所有业务配合升级，这对业务的研发效率也会造成损害；新特性的发布也依赖业务逐个升级，不具备自主的控制能力。

- 异构治理体系技术融合成本很高。
- 治理决策比较分散。每个节点只能根据自己的状态进行决策，无法与其他节点协同仲裁。



针对以上痛点，我们考虑依托于 Service Mesh 解决。Service Mesh 模式下会为每个业务实例部署一个 Sidecar 代理，所有进出应用的业务流量统一由 Sidecar 承载，同时服务治理的工作也主要由 Sidecar 执行，而所有的 Sidecar 由统一的中心化控制大脑控制面来进行全局管控。这种模式如何解决上述四个问题的呢?

- Service Mesh 模式下，各语言的通信框架一般仅负责编解码，而编解码的逻辑往往是不变的。核心的治理功能（如路由、限流等）主要由 Sidecar 代理和控制大脑协同完成，从而实现一套治理体系，所有语言通用。
- 中间件易变的逻辑尽量下沉到 Sidecar 和控制大脑中，后续升级中间件基本不需要业务配合。SDK 主要包含很轻薄且不易变的逻辑，从而实现了业务和中间件的解耦。
- 新融入的异构技术体系可以通过轻薄的 SDK 接入美团治理体系（技术体系难

兼容，本质是它们各自有独立的运行规范，在 Service Mesh 模式下运行规范核心内容就是控制面和 Sidecar），目前美团线上也有这样的案例。

- 控制大脑集中掌控了所有节点的信息，进而可以做一些全局最优的决策，比如服务预热、根据负载动态调整路由等能力。

总结一下，在当前治理体系进行 Mesh 化改造可以进一步提升治理能力，美团也将 Mesh 化改造后的 OCTO 定义为下一代服务治理系统 OCTO2.0（内部名字是 OCTO Mesh）。

二、技术选型及架构设计

2.1 OCTO Mesh 技术选型

美团的 Service Mesh 建设起步于 2018 年底，当时所面临一个核心问题是整体方案最关键的考量应该关注哪几个方面。在启动设计阶段时，我们有一个非常明确的意识：在大规模、同时治理能力丰富的前提下进行 Mesh 改造需要考虑的问题，与治理体系相对薄弱且期望依托于 Service Mesh 丰富治理能力的考量点，还是有非常大的差异的。总结下来，技术选型需要重点关注以下四个方面：

- OCTO 体系已经历近 5 年的迭代，形成了一系列的标准与规范，进行 Service Mesh 改造治理体系架构的升级范围会很大，在确保技术方案可以落地的同时，也要屏蔽技术升级或只需要业务做很低成本的改动。
- 治理能力不能减弱，在保证对齐的基础上逐渐提供更精细化、更易用的运营能力。
- 能应对超大规模的挑战，技术方案务必能确保支撑当前量级甚至当前 N 倍的增量，系统自身也不能成为整个治理体系的瓶颈。
- 尽量与社区保持亲和，一定程度上与社区协同演进。



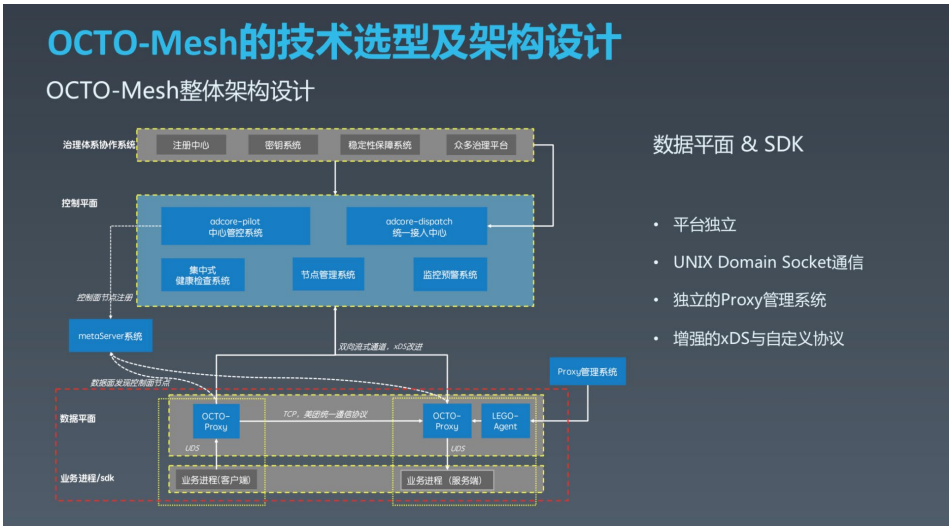
针对上述考量，我们选择的方式是数据面基于 Envoy 二次开发，控制面自研为主。

数据面方面，当时 Envoy 有机会成为数据面的事实标准，同时 Filter 模式及 xDS 的设计对扩展比较友好，未来功能的丰富、性能优化也与标准关系较弱。

控制面自研为主的决策需要考量的内容就比较复杂了，总体而言需要考虑如下几个方面：

- 截止发稿前，美团容器化主要采用富容器的模式，这种模式下强行与 Istio 及 Kubernetes 的数据模型匹配改造成本极高，同时 Istio API 也尚未确定。
- 截止发稿前，Istio 在集群规模变大时较容易出现性能问题，无法支撑美团数万应用、数十万节点的的体量，同时数十万节点规模的 Kubernetes 集群也需要持续优化探索。
- Istio 的功能无法满足 OCTO 复杂精细的治理需求，如流量录制回放压测、更复杂的路由策略等。
- 项目启动时非容器应用占比较高，技术方案需要兼容存量非容器应用。

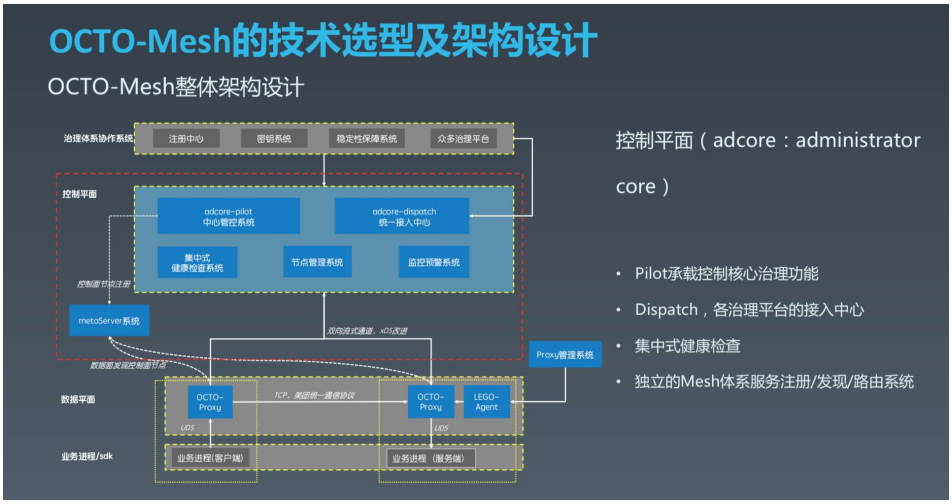
2.2 OCTO Mesh 架构设计



上面这张图展示了 OCTO Mesh 的整体架构。从下至上来看，逻辑上分为业务进程及通信框架 SDK 层、数据平面层、控制平面层、治理体系协作的所有周边生态层。

- 先来重点介绍下业务进程及 SDK 层、数据平面层：
- OCTO Proxy（数据面 Sidecar 代理内部叫 OCTO Proxy）与业务进程采用 1 对 1 的方式部署。
- OCTO Proxy 与业务进程采用 UNIX Domain Socket 做进程间通信（这里没有选择使用 Istio 默认的 iptables 流量劫持，主要考虑美团内部基本是使用的统一化私有协议通信，富容器模式没有用 Kubernetes 的命名服务模型，iptables 管理起来会很复杂，而 iptables 复杂后性能会出现较高的损耗。）；OCTO Proxy 间跨节点采用 TCP 通信，采用和进程间同样的协议，保证了客户端和服务端具备独立升级的能力。
- 为了提升效率同时减少人为错误，我们独立建设了 OCTO Proxy 管理系统，部署在每个实例上的 LEGO Agent 负责 OCTO Proxy 的保活和热升级，类似于 Istio 的 Pilot Agent，这种方式可以将人工干预降到较低，提升运维效率。

- 数据面与控制面通过双向流式通信。路由部分交互方式是增强语义的 xDS，增强语义是因为当前的 xDS 无法满足美团更复杂的路由需求；除路由外，该通道承载着众多的治理功能的指令及配置下发，我们设计了一系列的自定义协议。



控制面（内部名字是 Adcore）自研为主，整体分为：Adcore Pilot、Adcore Dispatcher、集中式健康检查系统、节点管理模块、监控预警模块。此外独立建设了统一元数据管理及 Mesh 体系内的服务注册发现系统 Meta Server 模块。每个模块的具体职责如下：

- Adcore Pilot 是个独立集群，模块承载着大部分核心治理功能的管控，相当于整个系统的大脑，也是直接与数据面交互的模块。
- Adcore Dispatcher 也是独立集群，该模块是供治理体系协作的众多子系统便捷接入 Mesh 体系的接入中心。
- 不同于 Envoy 的 P2P 节点健康检查模式，OCTO Mesh 体系使用的是集中式健康检查。
- 控制面会节点管理系统负责采集每个节点的运行时信息，并根据节点的状态做全局性的最优治理的决策和执行。
- 监控预警系统是保障 Mesh 自身稳定性而建设的模块，实现了自身的可观测性，当出现故障时能快速定位，同时也会对整个系统做实时巡检。

- 与 Istio 基于 Kubernetes 来做寻址和元数据管理不同，OCTO Mesh 由独立的 Meta Server 负责 Mesh 自身众多元信息的管理和命名服务。

三、关键设计解析

大规模治理体系 Mesh 化建设成功落地的关键点有：

- 系统水平扩展能力方面，可以支撑数万应用 / 百万级节点的治理。
- 功能扩展性方面，可以支持各类异构治理子系统融合打通。
- 能应对 Mesh 化改造后链路复杂的可用性、可靠性要求。
- 具备成熟完善的 Mesh 运维体系。

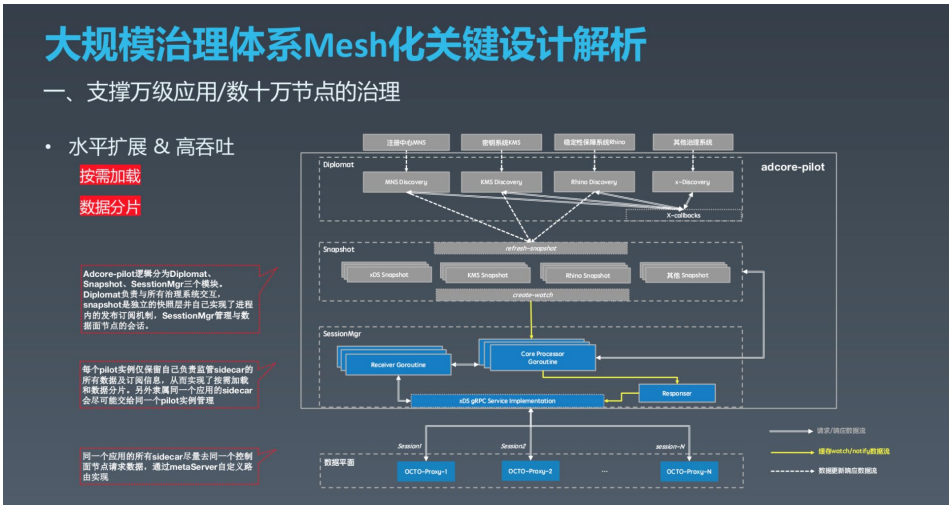
围绕这四点，便可以在系统能力、治理能力、稳定性、运营效率方面支撑美团当前多倍体量的新架构落地。

3.1 大规模系统 Mesh 化系统能力建设



对于社区 Istio 方案，要想实现超大规模应用集群落地，需要完成较多的技术改造。主要是因为 Istio 水平扩展能力相对薄弱，内部冗余操作较多，整体稳定性建设较为薄弱。针对上述问题，我们的解决思路如下：

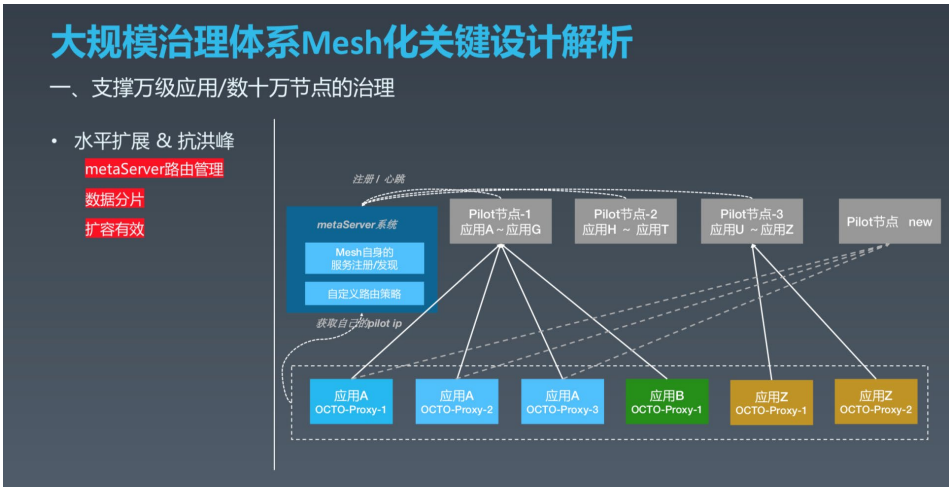
- 控制面每个节点并不承载所有治理数据，系统整体做水平扩展，在此基础上提升每个实例的整体吞吐量和性能。
- 当出现机房断网等异常情况时，可以应对瞬时流量骤增的能力。
- 只做必要的 P2P 模式健康检查，配合集中式健康检查进行百万级节点管理。



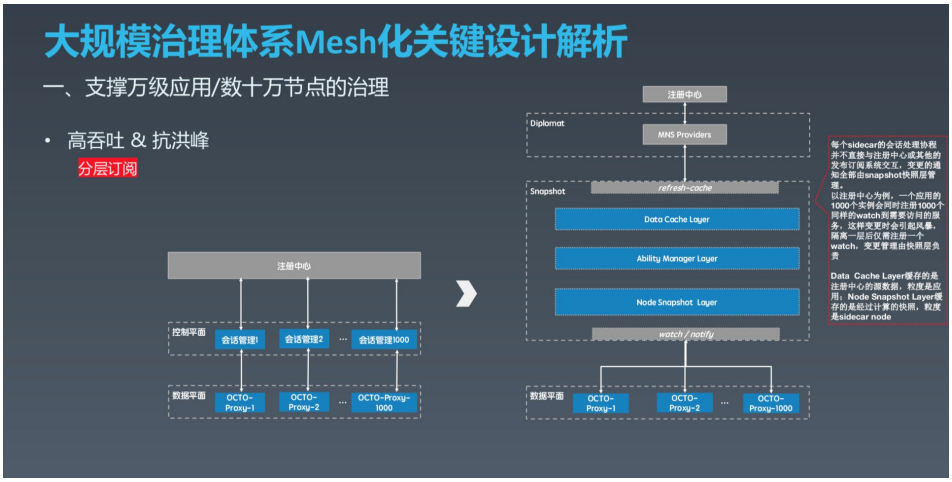
按需加载和数据分片主要由 Adcore Pilot 配合 Meta Server 实现。

Pilot 的逻辑架构分为 SessionMgr、Snapshot、Diplomat 三个部分，其中 SessionMgr 管理每个数据面会话的全生命周期、会话的创建、交互及销毁等一系列动作及流程；Snapshot 维护数据最新的一致性快照，对下将资源的更新同步给 SessionMgr 处理，对上响应各平台的数据变更通知，进行计算并将存在关联关系的一组数据做快照缓存。Diplomat 模块负责与服务治理系统的众多平台对接，只有该模块会与第三方平台直接产生依赖。

控制面每个 Pilot 节点并不会把整个注册中心及其他数据都加载进来，而是按需加载自己管控的 Sidecar 所需要的相关治理数据，即从 SessionMgr 请求的应用所负责的相关治理数据，以及该应用关注的对端服务注册信息。另外同一个应用的所有 OCTO Proxy 应该由同一个 Pilot 实例管控，否则全局状态下又容易趋近于全量了。具体是怎么实现的呢？答案是 Meta Server，自己实现控制面机器服务发现的同时精细化控制路由规则，从而在应用层面实现了数据分片。



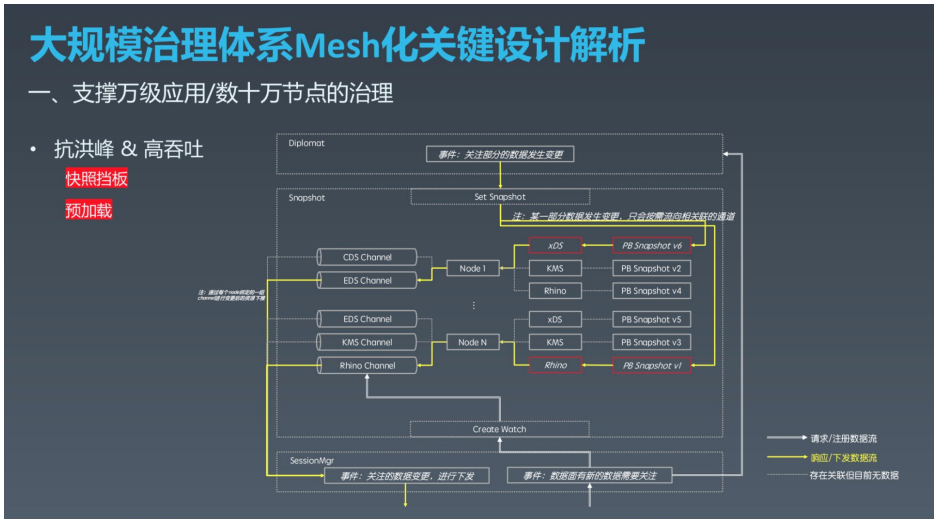
Meta Server 管控每个 Pilot 节点负责应用 OCTO Proxy 的归属关系。当 Pilot 实例启动会注册到 Meta Server，此后定时发送心跳进行续租，长时间心跳异常会自动剔除。在 Meta Server 内部实现了较为复杂的一致性哈希策略，会综合节点的应用、机房、负载等信息进行分组。当一个 Pilot 节点异常或发布时，隶属该 Pilot 的 OCTO Proxy 都会有规律的连接到接替节点，而不会全局随机连接对后端注册中心造成风暴。当异常或发布后的节点恢复后，划分出去的 OCTO Proxy 又会有规则的重新归属当前 Pilot 实例管理。对于关注节点特别多的应用 OCTO Proxy，也可以独立部署 Pilot，通过 Meta Server 统一进行路由管理。



Mesh 体系的命名服务需要 Pilot 与注册中心打通，常规的实现方式如左图所示（以 Zookeeper 为例），每个 OCTO Proxy 与 Pilot 建立会话时，作为客户端角色会向注册中心订阅自身所关注的服务端变更监听器，假设这个服务需要访问 100 个应用，则至少需要注册 100 个 Watcher 。假设该应用存在 1000 个实例同时运行，就会注册 100*1000 = 100000 个 Watcher，超过 1000 个节点的应用在美团内部还是蛮多的。另外还有很多应用关注的对端节点相同，会造成大量的冗余监听。当规模较大后，网络抖动或业务集中发布时，很容易引发风暴效应把控制面和后端的注册中心打挂。

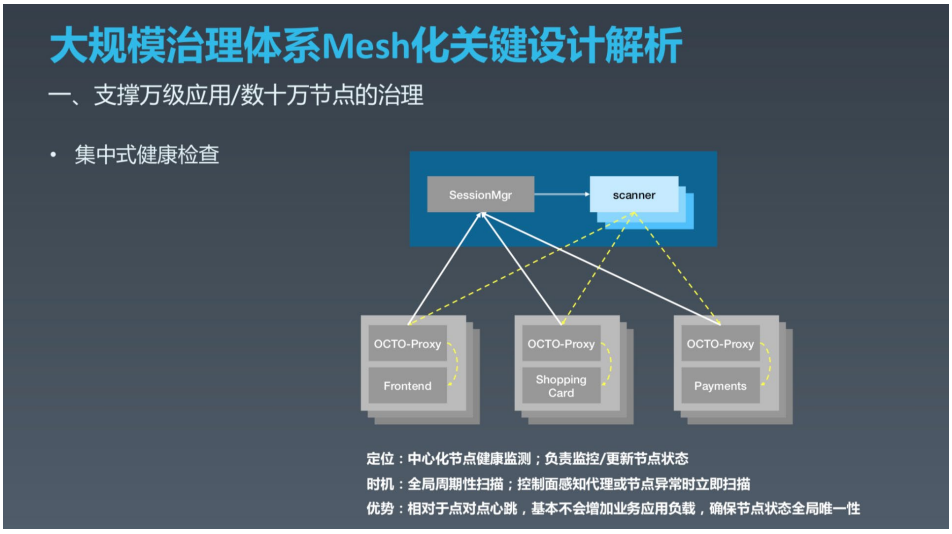
针对这个问题，我们采用分层订阅的方案解决。每个 OCTO Proxy 的会话并不直接与注册中心或其他的发布订阅系统交互，变更的通知全部由 Snapshot 快照层管理。Snapshot 内部又划分为 3 层，Data Cache 层对接并缓存注册中心及其他系统的原始数据，粒度是应用；Node Snapshot 层则是保留经过计算的节点粒度的数据；Ability Manager 层内部会做索引和映射的管理，当注册中心存在节点状态变更时，会通过索引将变更推送给关注变更的 OCTO Proxy。

对于刚刚提到的场景，隔离一层后 1000 个节点仅需注册 100 个 Watcher，一个 Watcher 变更后仅会有一条变更信息到 Data Cache 层，再根据索引向 1000 个 OCTO Proxy 通知，从而极大的降低了注册中心及 Pilot 的负载。



Snapshot 层除了减少不必要交互提升性能外，也会将计算后的数据格式化缓存下来，一方面瞬时大量相同的请求会在快照层被缓存挡住，另一方面也便于将存在关联的数据统一打包到一起，避免并发问题。这里参考了 Envoy-Control-Plane 的设计，Envoy-Control-Plane 会将包含 xDS 的所有数据全部打包在一起，而我们是将数据隔离开，如路由、鉴权完全独立，当路由数据变更时不会去拉取并更新鉴权信息。

预加载主要目的是提升服务冷启动性能，Meta Server 的路由规则由我们制定，所以这里提前在 Pilot 节点中加载好最新的数据，当业务进程启动时，Proxy 就可以立即从 Snapshot 中获取到数据，避免了首次访问慢的问题。



Istio 默认每个 Envoy 代理对整个集群中所有其余 Envoy 进行 P2P 健康检测，当集群有 N 个节点时，一个检测周期内（往往不会很长）就需要做 N 的平方次检测，另外当集群规模变大时所有节点的负载就会相应提高，这都将成为扩展部署的极大障碍。

不同于全集群扫描，美团采用了集中式的健康检查方式，同时配合必要的 P2P 检测。具体实现方式是：由中心服务 Scanner 监测所有节点的状态，当 Scanner 主动检测到节点异常或 Pilot 感知连接变化通知 Scanner 扫描确认节点异常时，Pilot 立刻通过 eDS 更新节点状态给 Proxy，这种模式下检测周期内仅需要检测 N 次。

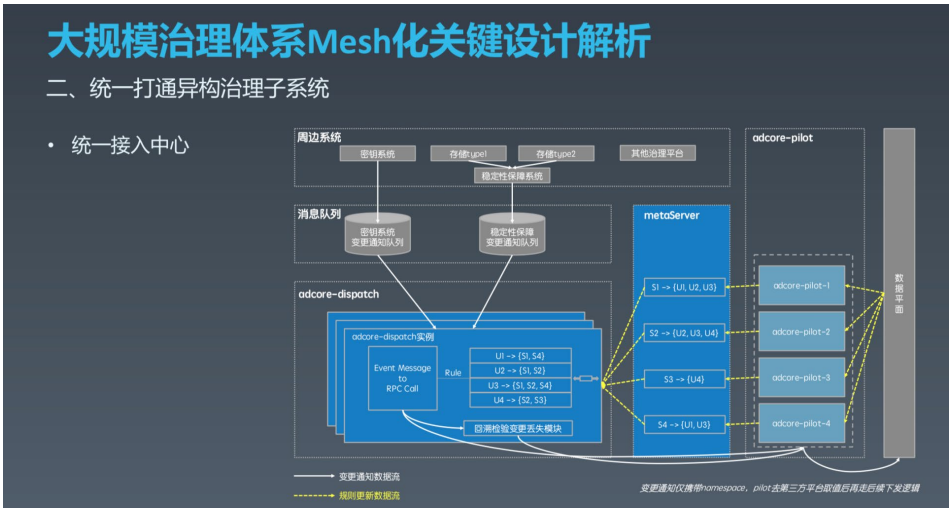
Google 的 Traffic Director 也采用了类似的设计，但大规模使用需要一些技巧：第一个是为了避免机房自治的影响而选择了同机房检测方式，第二个是为了减少中心检测机器因自己 GC 或网络异常造成误判，而采用了 Double Check 的机制。

此外除了集中健康检查，还会对频繁失败的对端进行心跳探测，根据探测结果进行降权或摘除操作提升成功率。

3.2 异构治理系统融合设计

OCTO Mesh 需要对齐当前体系的核心治理能力，这就不可避免的将 Mesh 与治理生态的所有周边子系统打通。Istio 和 Kubernetes 将所有的数据存储、发布订阅机制都依赖 Etcd 统一实现，但美团的 10 余个治理子系统功能各异、存储各异、发布订阅模式各异，呈现出明显的异构特征，如果接入一个功能就需要平台进行存储或其他大规模改造，这样是完全不可行的。一个思路是由一个模块来解耦治理子系统与 Pilot ，这个模块承载所有的变更并将这个变更下发给 Pilot，但这种方式也有一些问题需要考虑，之前介绍每个 Pilot 节点关注的数据并不同，而且分片的规则也可能时刻变化，有一套机制能将消息发送给关注的 Pilot 节点。

总体而言需要实现三个子目标：打通所有系统，治理能力对齐；快速应对未来新系统的接入；变更发送给关注节点。我们解法是：独立的统一接入中心，屏蔽所有异构系统的存储、发布订阅机制；Meta Server 承担实时分片规则的元数据管理。

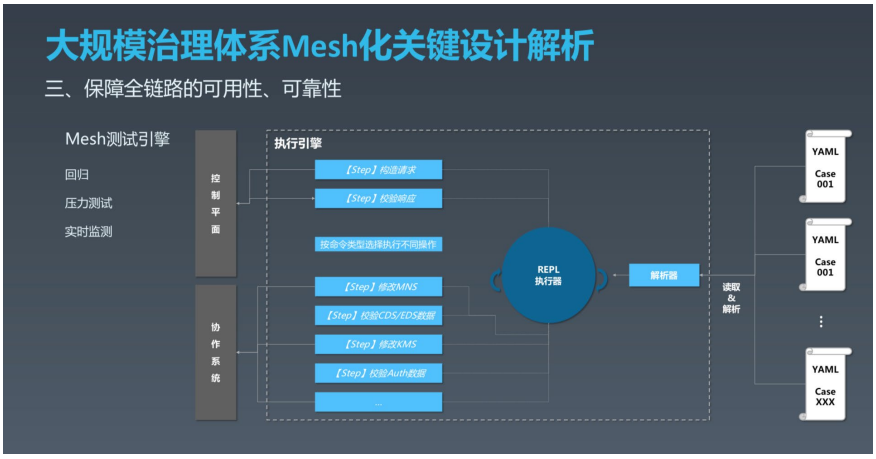


具体执行机制如上图所示：各系统变更时使用客户端将变更通知推送到消息队列，只推送变更但不包含具体值（当 Pilot 接收到变更通知是会主动 Fetch 全量数据，这种方式一方面确保 Mafka 的消息足够小，另一方面多个变更不需要在队列中保序解决版本冲突问题）；Adcore Dispatcher 消费信息并根据索引将变更推送到关注的 Pilot 机器，当 Pilot 管控的 Proxy 变更时会同步给 Meta Server，Meta Server 实时将索引关系更新并同步给 Dispatcher。为了解决 Pilot 与应用的映射变更间隙出现消息丢失，Dispatcher 使用回溯检验变更丢失的模式进行补偿，以提升系统的可靠性。

3.3 稳定性保障设计

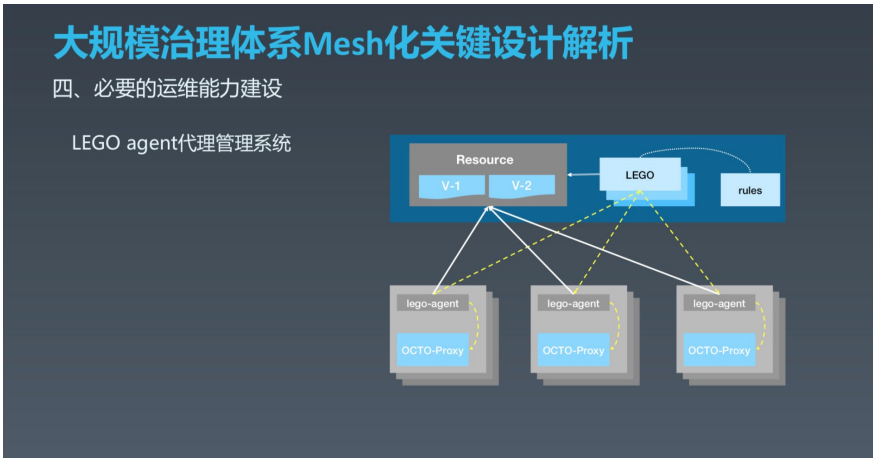


Service Mesh 改造的系统避不开“新”和“复杂”两个特征，其中任意一个特征都可能会给系统带来稳定性风险，所以必须提前做好整个链路的可用性及可靠性建设，才能游刃有余的推广。美团主要是围绕控制故障影响范围、异常实时自愈、可实时回滚、柔性可用、提升自身可观测性及回归能力进行建设。



这里单独介绍控制面的测试问题，这块业界可借鉴的内容不多。xDS 双向通信比较复杂，很难像传统接口那样进行功能测试，定制多个 Envoy 来模拟数据面进行测试成本也很高。 我们开发了 Mock-Sidecar 来模拟真正数据面的行为来对控制面进行测试，对于控制面来说它跟数据面毫无区别。Mock-Sidecar 把数据面的整体行为拆分为一个个可组合的 Step，机制与策略分离。执行引擎就是所谓的机制，只需要按步骤执行 Step 即可。YAML 文件就是 Step 的组合，用于描述策略。我们人工构造各种 YAML 来模拟真正 Sidecar 的行为，对控制面进行回归验证，同时不同 YAML 文件执行是并行的，可以进行压力测试。

3.4 运维体系设计



为了应对未来百万级 Proxy 的运维压力，美团独立建设了 OCTO Proxy 运维系统 LEGO，除 Proxy 保活外也统一集中控制发版。具体的操作流程是：运维人员在 LEGO 平台发版，确定发版的范围及版本，新版本资源内容上传至资源仓库，并更新规则及发版范围至 DB，发升级指令下发至所要发布的范围，收到发版命令机器的 LEGO Agent 去资源仓库拉取要更新的版本（中间如果有失败，会有主动 Poll 机制保证升级成功），新版本下载成功后，由 LEGO Agent 启动新版的 OCTO Proxy。

四、总结与展望

4.1 经验总结

- 服务治理建设应该围绕体系标准化、易用性、高性能三个方面开展。
- 大规模治理体系 Mesh 化应该关注以下内容：
  - 适配公司技术体系比新潮技术更重要，重点关注容器化 & 治理体系兼容打通。
  - 建设系统化的稳定性保障体系及运维体系。
- OCTO Mesh 控制面 4 大法宝：Meta Server 管控 Mesh 内部服务注册发现及元数据、分层分片设计、统一接入中心解耦并打通 Mesh 与现有治理子系统、集中式健康检查。

4.2 未来展望

未来，我们会继续在 OCTO Mesh 道路上探索，包括但不限于以下几个方面：

- 完善体系：逐渐丰富的 OCTO Mesh 治理体系，探索其他流量类型，全面提升服务治理效率。
- 大规模落地：持续打造健壮的 OCTO Mesh 治理体系，稳步推动在公司的大规模落地。
- 中心化治理能力探索：新治理模式的中心化管控下，全局最优治理能力探索。

作者简介

继东，基础架构部服务治理团队。

招聘信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：美团点评基础架构团队）。

实践与经验总结

XGBoost 缺失值引发的问题及其深度分析

李兆军

1. 背景

XGBoost 模型作为机器学习中的一大“杀器”，被广泛应用于数据科学竞赛和工业领域，XGBoost 官方也提供了可运行于各种平台和环境的对应代码，如适用于 Spark 分布式训练的 XGBoost on Spark。然而，在 XGBoost on Spark 的官方实现中，却存在一个因 XGBoost 缺失值和 Spark 稀疏表示机制而带来的不稳定问题。

事情起源于美团内部某机器学习平台使用方同学的反馈，在该平台上训练出的 XGBoost 模型，使用同一个模型、同一份测试数据，在本地调用（Java 引擎）与平台（Spark 引擎）计算的结果不一致。但是该同学在本地运行两种引擎（Python 引擎和 Java 引擎）进行测试，两者的执行结果是一致的。因此质疑平台的 XGBoost 预测结果会不会有问题?

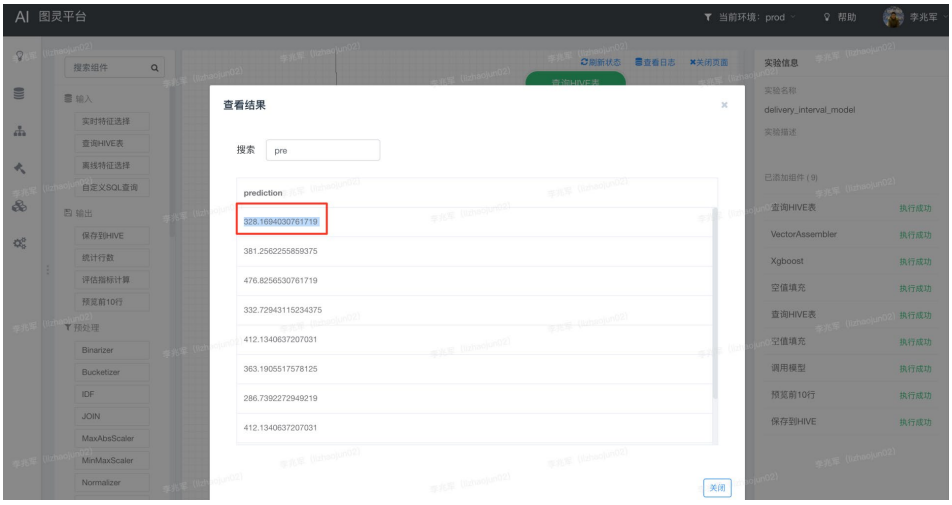
该平台对 XGBoost 模型进行过多次定向优化，在 XGBoost 模型测试时，并没有出现过本地调用（Java 引擎）与平台（Spark 引擎）计算结果不一致的情形。而且平台上运行的版本，和该同学本地使用的版本，都来源于 Dmlc 的官方版本，JNI 底层调用的应该是同一份代码，理论上，结果应该是完全一致的，但实际中却不同。

从该同学给出的测试代码上，并没有发现什么问题：

```
// 测试结果中的一行，41 列
double[] input = new double[]{1, 2, 5, 0, 0, 6.666666666666667, 31.14,
29.28, 0, 1.303333,
2.8555, 2.37, 701, 463, 3.989, 3.85, 14400.5, 15.79, 11.45, 0.915,
7.05, 5.5, 0.023333, 0.0365,
0.0275, 0.123333, 0.4645, 0.12, 15.082, 14.48, 0, 31.8425, 29.1,
7.7325, 3, 5.88, 1.08, 0, 0, 0,
32];
// 转化为 float[]
```

```
float[] testInput = new float[input.length];
for(int i = 0, total = input.length; i < total; i++){
  testInput[i] = new Double(input[i]).floatValue();
}
// 加载模型
Booster booster = XGBoost.loadModel("${model}");
// 转为 DMatrix，一行，41 列
DMatrix testMat = new DMatrix(testInput, 1, 41);
// 调用模型
float[][] predicts = booster.predict(testMat);
```

上述代码在本地执行的结果是 333.67892，而平台上执行的结果却是 328.1694030761719。



两次结果怎么会不一样，问题出现在哪里呢?

2. 执行结果不一致问题排查历程

如何排查？首先想到排查方向就是，两种处理方式中输入的字段类型会不会不一致。如果两种输入中字段类型不一致，或者小数精度不同，那结果出现不同就是可解释的了。仔细分析模型的输入，注意到数组中有一个 6.666666666666667，是不是它的原因?

一个个 Debug 仔细比对两侧的输入数据及其字段类型，完全一致。

这就排除了两种方式处理时，字段类型和精度不一致的问题。

第二个排查思路是，XGBoost on Spark 按照模型的功能，提供了 XGBoostClassifier 和 XGBoostRegressor 两个上层 API，这两个上层 API 在 JNI 的基础上，加入了很多超参数，封装了很多上层能力。会不会是在这两种封装过程中，新加入的某些超参数对输入结果有着特殊的处理，从而导致结果不一致?

与反馈此问题的同学沟通后得知，其 Python 代码中设置的超参数与平台设置的完全一致。仔细检查 XGBoostClassifier 和 XGBoostRegressor 的源代码，两者对输出结果并没有做任何特殊处理。

再次排除了 XGBoost on Spark 超参数封装问题。

再一次检查模型的输入，这次的排查思路是，检查一下模型的输入中有没有特殊的数值，比方说，NaN、-1、0 等。果然，输入数组中有好几个 0 出现，会不会是因为缺失值处理的问题?

快速找到两个引擎的源码，**发现两者对缺失值的处理真的不一致!**

XGBoost4j 中缺失值的处理

XGBoost4j 缺失值的处理过程发生在构造 DMatrix 过程中，默认将 0.0f 设置为缺失值:

```
  /**
   * create DMatrix from dense matrix
   *
   * @param data data values
   * @param nrow number of rows
   * @param ncol number of columns
   * @throws XGBoostError native error
   */
  public DMatrix(float[] data, int nrow, int ncol) throws XGBoostError {
    long[] out = new long[1];

    //0.0f 作为 missing 的值
    XGBoostJNI.checkCall(XGBoostJNI.XGDMatrixCreateFromMat(data, nrow,
ncol, 0.0f, out));

    handle = out[0];
  }
```

XGBoost on Spark 中缺失值的处理

而 xgboost on Spark 将 NaN 作为默认的缺失值。

```
  /**
   * @return A tuple of the booster and the metrics used to build
training summary
   */
  @throws(classOf[XGBoostError])
  def trainDistributed(
      trainingDataIn: RDD[XGBLabeledPoint],
      params: Map[String, Any],
      round: Int,
      nWorkers: Int,
      obj: ObjectiveTrait = null,
      eval: EvalTrait = null,
      useExternalMemory: Boolean = false,

      //NaN 作为 missing 的值
      missing: Float = Float.NaN,

      hasGroup: Boolean = false): (Booster, Map[String, Array[Float]])
= {
      //...
      }
```

也就是说，本地 Java 调用构造 DMatrix 时，如果不设置缺失值，默认值 0 被当作缺失值进行处理。而在 XGBoost on Spark 中，默认 NaN 会被为缺失值。原来 Java 引擎和 XGBoost on Spark 引擎默认的缺失值并不一样。而平台和该同学调用时，都没有设置缺失值，造成两个引擎执行结果不一致的原因，就是因为缺失值不一致！

修改测试代码，在 Java 引擎代码上设置缺失值为 NaN，执行结果为 328.1694，与平台计算结果完全一致。

```
    // 测试结果中的一行，41 列
    double[] input = new double[]{1, 2, 5, 0, 0, 6.666666666666667,
31.14, 29.28, 0, 1.303333,
2.8555, 2.37, 701, 463, 3.989, 3.85, 14400.5, 15.79, 11.45, 0.915,
7.05, 5.5, 0.023333, 0.0365,
0.0275, 0.123333, 0.4645, 0.12, 15.082, 14.48, 0, 31.8425, 29.1,
7.7325, 3, 5.88, 1.08, 0, 0, 0,
32];
    float[] testInput = new float[input.length];
    for(int i = 0, total = input.length; i < total; i++){
      testInput[i] = new Double(input[i]).floatValue();
    }
```

```
Booster booster = XGBoost.loadModel("${model}");
// 一行，41 列
DMatrix testMat = new DMatrix(testInput, 1, 41, Float.NaN);
float[][] predicts = booster.predict(testMat);
```

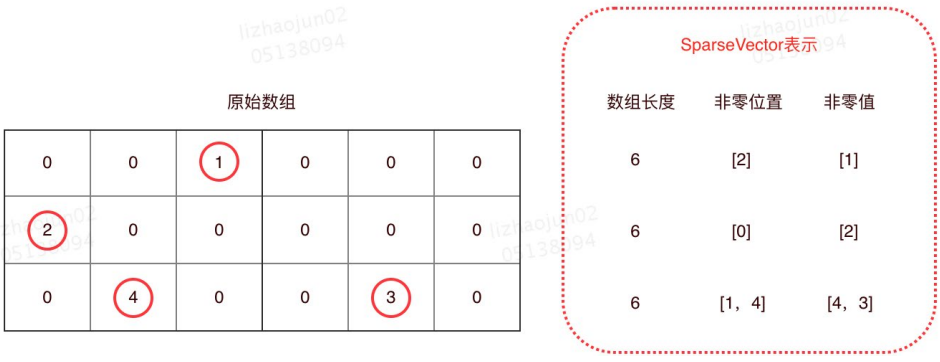
3. XGBoost on Spark 源码中缺失值引入的不稳定问题

然而，事情并没有这么简单。

Spark ML 中还有隐藏的缺失值处理逻辑：SparseVector，即稀疏向量。

- SparseVector 和 DenseVector 都用于表示一个向量，两者之间仅仅是存储结构的不同。
- 其中，DenseVector 就是普通的 Vector 存储，按序存储 Vector 中的每一个值。
- 而 SparseVector 是稀疏的表示，用于向量中 0 值非常多场景下数据的存储。
- SparseVector 的存储方式是：仅仅记录所有非 0 值，忽略掉所有 0 值。具体来说，用一个数组记录所有非 0 值的位置，另一个数组记录上述位置所对应的数值。有了上述两个数组，再加上当前向量的总长度，即可将原始的数组还原回来。
- 因此，对于 0 值非常多的一组数据，SparseVector 能大幅节省存储空间。

SparseVector 存储示例见下图：



如上图所示，SparseVector 中不保存数组中值为 0 的部分，仅仅记录非 0 值。因此对于值为 0 的位置其实不占用存储空间。下述代码是 Spark ML 中 VectorAssembler 的实现代码，从代码中可见，如果数值是 0，在 SparseVector 中是不进行记录的。

```
    private[feature] def assemble(vv: Any*): Vector = {
    val indices = ArrayBuilder.make[Int]
    val values = ArrayBuilder.make[Double]
    var cur = 0
    vv.foreach {
      case v: Double =>

        //0 不进行保存
        if (v != 0.0) {

          indices += cur
          values += v
        }
        cur += 1
      case vec: Vector =>
        vec.foreachActive { case (i, v) =>

          //0 不进行保存
          if (v != 0.0) {

            indices += cur + i
            values += v
          }
        }
        cur += vec.size
      case null =>
        throw new SparkException("Values to assemble cannot be null.")
      case o =>
        throw new SparkException(s"$o of type ${o.getClass.getName} is
not supported.")
    }
    Vectors.sparse(cur, indices.result(), values.result()).compressed
    }
```

不占用存储空间的值，也是某种意义上的一种缺失值。SparseVector 作为 Spark ML 中的数组的保存格式，被所有的算法组件使用，包括 XGBoost on Spark。而事实上 XGBoost on Spark 也的确将 Sparse Vector 中的 0 值直接当作缺失值进行处理：

```
    val instances: RDD[XGBLabeledPoint] = dataset.select(
      col($(featuresCol)),
      col($(labelCol)).cast(FloatType),
      baseMargin.cast(FloatType),
      weight.cast(FloatType)
    ).rdd.map { case Row(features: Vector, label: Float, baseMargin:
Float, weight: Float) =>
      val (indices, values) = features match {

        //SparseVector 格式，仅仅将非 0 的值放入 XGBoost 计算
        case v: SparseVector => (v.indices, v.values.map(_.toFloat))

        case v: DenseVector => (null, v.values.map(_.toFloat))
      }
      XGBLabeledPoint(label, indices, values, baseMargin = baseMargin,
weight = weight)
    }
```

XGBoost on Spark 将 SparseVector 中的 0 值作为缺失值**为什么会引入不稳定的问题呢?**

重点来了，Spark ML 中对 Vector 类型的存储是有优化的，它会自动根据 Vector 数组中的内容选择是存储为 SparseVector，还是 DenseVector。也就是说，一个 Vector 类型的字段，在 Spark 保存时，同一列会有两种保存格式：SparseVector 和 DenseVector。而且对于一份数据中的某一列，两种格式是同时存在的，有些行是 Sparse 表示，有些行是 Dense 表示。选择使用哪种格式表示通过下述代码计算得到：

```
  /**
   * Returns a vector in either dense or sparse format, whichever
uses less storage.
   */
  @Since("2.0.0")
  def compressed: Vector = {
    val nnz = numNonzeros
    // A dense vector needs 8 * size + 8 bytes, while a sparse vector
needs 12 * nnz + 20 bytes.
    if (1.5 * (nnz + 1.0) < size) {
      toSparse
    } else {
      toDense
    }
  }
```

在 XGBoost on Spark 场景下，默认将 Float.NaN 作为缺失值。如果数据集中的某一行存储结构是 DenseVector，实际执行时，该行的缺失值是 Float.NaN。而如果数据集中的某一行存储结构是 SparseVector，由于 XGBoost on Spark 仅仅使用了 SparseVector 中的非 0 值，也就导致该行数据的缺失值是 Float.NaN 和 0。

也就是说，如果数据集中某一行数据适合存储为 DenseVector，则 XGBoost 处理时，该行的缺失值为 Float.NaN。而如果该行数据适合存储为 SparseVector，则 XGBoost 处理时，该行的缺失值为 Float.NaN 和 0。

即，数据集中一部分数据会以 Float.NaN 和 0 作为缺失值，另一部分数据会以 Float.NaN 作为缺失值！ 也就是说在 XGBoost on Spark 中，0 值会因为底层数据存储结构的不同，同时会有两种含义，而底层的存储结构是完全由数据集决定的。

因为线上 Serving 时，只能设置一个缺失值，因此被选为 SparseVector 格式的测试集，可能会导致线上 Serving 时，计算结果与期望结果不符。

4. 问题解决

查了一下 XGBoost on Spark 的最新源码，依然没解决这个问题。

赶紧把这个问题反馈给 XGBoost on Spark，同时修改了我们自己的 XGBoost on Spark 代码。

```
    val instances: RDD[XGBLabeledPoint] = dataset.select(
       col($(featuresCol)),
       col($(labelCol)).cast(FloatType),
       baseMargin.cast(FloatType),
       weight.cast(FloatType)
    ).rdd.map { case Row(features: Vector, label: Float, baseMargin:
Float, weight: Float) =>

       // 这里需要对原来代码的返回格式进行修改
       val values = features match {

         //SparseVector 的数据，先转成 Dense
         case v: SparseVector => v.toArray.map(_.toFloat)

         case v: DenseVector => v.values.map(_.toFloat)
       }
       XGBLabeledPoint(label, null, values, baseMargin = baseMargin,
```

```
weight = weight)
    }
    /**
     * Converts a [[Vector]] to a data point with a dummy label.
     *
     * This is needed for constructing a [[ml.dmlc.xgboost4j.scala.
DMatrix]]
     * for prediction.
     */
    def asXGB: XGBLabeledPoint = v match {
      case v: DenseVector =>
        XGBLabeledPoint(0.0f, null, v.values.map(_.toFloat))
      case v: SparseVector =>

        //SparseVector 的数据，先转成 Dense
        XGBLabeledPoint(0.0f, null, v.toArray.map(_.toFloat))

    }
```

问题得到解决，而且用新代码训练出来的模型，评价指标还会有些许提升，也算是意外之喜。

希望本文对遇到 XGBoost 缺失值问题的同学能够有所帮助，也欢迎大家一起交流讨论。

作者简介

兆军，美团配送事业部算法平台团队技术专家。

招聘信息

美团配送事业部算法平台团队，负责美团一站式大规模机器学习平台图灵平台的建设。围绕算法整个生命周期，利用可视化拖拽方式定义模型训练和预测流程，提供强大的模型管理、线上模型预测和特征服务能力，提供多维立体的 AB 分流支持和线上效果评估支持。团队的使命是为算法相关同学提供统一的，端到端的，一站式自助服务平台，帮助算法同学降低算法研发复杂度，提升算法迭代效率。

现面向数据工程，数据开发，算法工程，算法应用等领域招聘资深研发工程师 / 技术专家 / 方向负责人（机器学习平台 / 算法平台），欢迎有兴趣的同学一起加入，简历可投递至：tech@meituan.com（注明：美团配送事业部）

Spring Boot 引起的“堆外内存泄漏”排查及经验总结

纪兵

背景

为了更好地实现对项目的管理，我们将组内一个项目迁移到 MDP 框架（基于 Spring Boot），随后我们就发现系统会频繁报出 Swap 区域使用量过高的异常。笔者被叫去帮忙查看原因，发现配置了 4G 堆内内存，但是实际使用的物理内存竟然高达 7G，确实不正常。JVM 参数配置是“-XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+AlwaysPreTouch -XX:ReservedCodeCacheSize=128m -XX:InitialCodeCacheSize=128m, -Xss512k -Xmx4g -Xms4g,-XX:+UseG1GC -XX:G1HeapRegionSize=4M”，实际使用的物理内存如下图所示：

```
top - 09:03:58 up 8 days, 10:56,  1 user,  load average: 0.18, 0.18, 0.24
Tasks: 142 total,   1 running, 139 sleeping,   0 stopped,   2 zombie
Cpu(s):  4.4%us,  1.4%sy,  0.0%ni, 93.9%id,  0.0%wa,  0.0%hi,  0.1%si,  0.1%st
Mem:   8057844k total,  7824052k used,   233792k free,     3664k buffers
Swap:  2096440k total,    57732k used,  2038708k free,   245948k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31708 sankuai   20   0 9448m 6.8g 3156 S 117.0 88.0 210:43.85 java
  702 sankuai   20   0  839m  22m 1980 S  2.0  0.3 118:53.38 log_agent
    1 root      20   0 54640  348  164 S  0.0  0.0   0:00.78 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
```

top 命令显示的内存情况

排查过程

1. 使用 Java 层面的工具定位内存区域（堆内内存、Code 区域或者使用 unsafe.allocateMemory 和 DirectByteBuffer 申请的堆外内存）

笔者在项目中添加 -XX:NativeMemoryTracking=detailJVM 参数重启项目，使用命令 jcmd pid VM.native_memory detail 查看到的内存分布如下：

```
Native Memory Tracking:

Total: reserved=6478032KB, committed=5440716KB
-                 Java Heap (reserved=4194304KB, committed=4194304KB)
                            (mmap: reserved=4194304KB, committed=4194304KB)

-                     Class (reserved=1140931KB, committed=103619KB)
                            (classes #16301)
                            (malloc=2243KB #40399)
                            (mmap: reserved=1138688KB, committed=101376KB)

-                    Thread (reserved=570885KB, committed=570885KB)
                            (thread #555)
                            (stack: reserved=569512KB, committed=569512KB)
                            (malloc=722KB #2777)
                            (arena=650KB #1109)

-                      Code (reserved=146450KB, committed=146450KB)
                            (malloc=13330KB #16372)
                            (mmap: reserved=133120KB, committed=133120KB)

-                        GC (reserved=205314KB, committed=205314KB)
                            (malloc=16898KB #26279)
                            (mmap: reserved=188416KB, committed=188416KB)

-                  Compiler (reserved=1921KB, committed=1921KB)
                            (malloc=1790KB #2122)
                            (arena=131KB #3)

-                  Internal (reserved=189371KB, committed=189367KB)
                            (malloc=189335KB #74055)
                            (mmap: reserved=36KB, committed=32KB)

-                    Symbol (reserved=22818KB, committed=22818KB)
                            (malloc=18911KB #189399)
                            (arena=3908KB #1)

-    Native Memory Tracking (reserved=5822KB, committed=5822KB)
                            (malloc=254KB #3754)
                            (tracking overhead=5569KB)

-               Arena Chunk (reserved=215KB, committed=215KB)
                            (malloc=215KB)

Virtual memory map:
```

jcmd 显示的内存情况

发现命令显示的 committed 的内存小于物理内存，因为 jcmd 命令显示的内存包含堆内内存、Code 区域、通过 unsafe.allocateMemory 和 DirectByteBuffer 申请的内存，但是不包含其他 Native Code（C 代码）申请的堆外内存。所以猜测是使用 Native Code 申请内存所导致的问题。

为了防止误判，笔者使用了 pmap 查看内存分布，发现大量的 64M 的地址；而这些地址空间不在 jcmd 命令所给出的地址空间里面，基本上就断定就是这些 64M 的内存所导致。



pmap 显示的内存情况

2. 使用系统层面的工具定位堆外内存

因为笔者已经基本上确定是 Native Code 所引起，而 Java 层面的工具不便于排查此类问题，只能使用系统层面的工具去定位问题。

首先，使用了 gperftools 去定位问题

gperftools 的使用方法可以参考 gperftools，gperftools 的监控如下：

```
ap-gperftools/heap.hprof_21187.0030.heap (2796 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0030.heap
ap-gperftools/heap.hprof_21187.0031.heap (2896 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0031.heap
ap-gperftools/heap.hprof_21187.0032.heap (2996 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0032.heap
ap-gperftools/heap.hprof_21187.0033.heap (3096 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0033.heap
ap-gperftools/heap.hprof_21187.0034.heap (9043 MB allocated cumulatively, 640 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0034.heap
ap-gperftools/heap.hprof_21187.0035.heap (10068 MB allocated cumulatively, 729 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0035.heap
ap-gperftools/heap.hprof_21187.0036.heap (11106 MB allocated cumulatively, 762 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0036.heap
```

gperftools 监控

从上图可以看出：使用 malloc 申请的的内存最高到 3G 之后就释放了，之后始终维持在 700M-800M。笔者第一反应是：难道 Native Code 中没有使用 malloc 申请，直接使用 mmap/brk 申请的?（gperftools 原理就使用动态链接的方式替换了操作系统默认的内存分配器（glibc）。）

然后，使用 strace 去追踪系统调用

因为使用 gperftools 没有追踪到这些内存，于是直接使用命令“strace -f -e”brk,mmap,munmap”-p pid”追踪向 OS 申请内存请求，但是并没有发现有可疑内存申请。strace 监控如下图所示：

```
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
```

strace 监控

接着，使用 GDB 去 dump 可疑内存

因为使用 strace 没有追踪到可疑内存申请；于是想着看看内存中的情况。就是直接使用命令 gdp –pid pid 进入 GDB 之后，然后使用命令 dump memory mem.bin startAddress endAddressdump 内存，其中 startAddress 和 endAddress 可以从 /proc/pid/smaps 中查找。然后使用 strings mem.bin 查看 dump 的内容，如下：

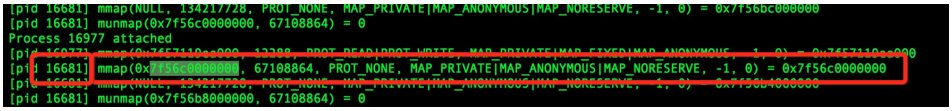


gperftools 监控

从内容上来看，像是解压后的 JAR 包信息。读取 JAR 包信息应该是在项目启动的时候，那么在项目启动之后使用 strace 作用就不是很大了。所以应该在项目启动的时候使用 strace，而不是启动完成之后。

再次，项目启动时使用 strace 去追踪系统调用

项目启动使用 strace 追踪系统调用，发现确实申请了很多 64M 的内存空间，截图如下：



strace 监控

使用该 mmap 申请的地址空间在 pmap 对应如下：

```
00007f56b7fff000      4      0      0 -----  [ anon ]
00007f56b8000000  65480  37932  37932 rw---  [ anon ]
00007f56bbff2000     56      0      0 -----  [ anon ]
00007f56bc000000  65508  23792  23792 rw---  [ anon ]
00007f56c0000000  65528  25196  25196 rw---  [ anon ]
00007f56c4000000  65508  24252  24252 rw---  [ anon ]
00007f56c7ff9000     28      0      0 -----  [ anon ]
00007f56c8000000  65524  29372  29372 rw---  [ anon ]
00007f56cbffd000     12      0      0 -----  [ anon ]
00007f56cc000000 131044  44876  44876 rw---  [ anon ]
00007f56d3ff9000     28      0      0 -----  [ anon ]
00007f56d4000000  65532  25716  25716 rw---  [ anon ]
00007f56d7fff000      4      0      0 -----  [ anon ]
00007f56d8000000  65516  23228  23228 rw---  [ anon ]
00007f56dbffb000     20      0      0 -----  [ anon ]
00007f56dc000000 131052  47168  47168 rw---  [ anon ]
00007f56e3ffb000     20      0      0 -----  [ anon ]
00007f56e4000000  65520  25208  25208 rw---  [ anon ]
```

strace 申请内容对应的 pmap 地址空间

最后，使用 jstack 去查看对应的线程

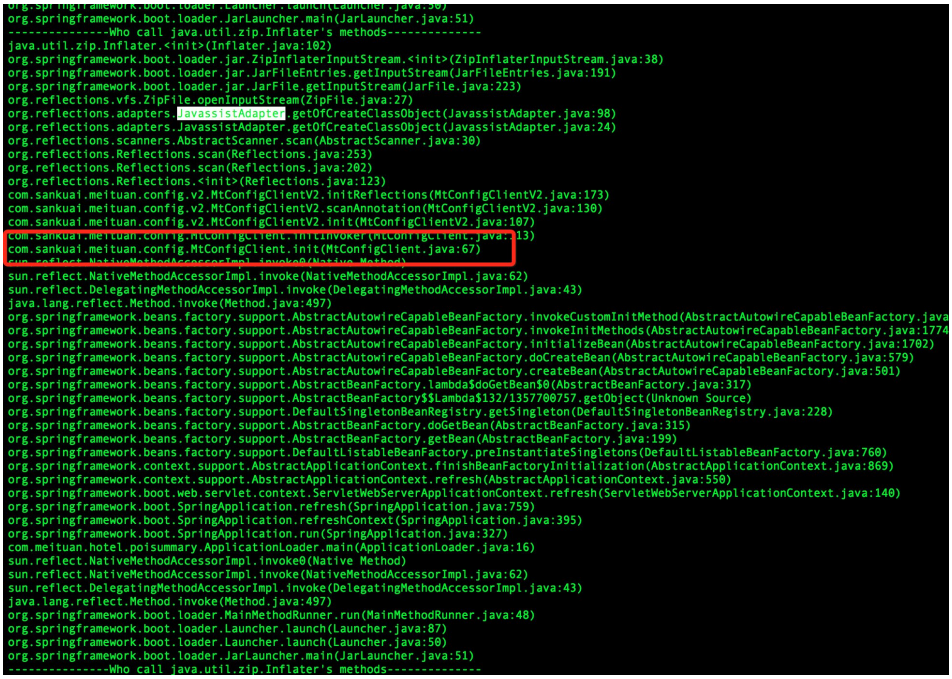
因为 strace 命令中已经显示申请内存的线程 ID。直接使用命令 jstack pid 去查看线程栈，找到对应的线程栈（注意 10 进制和 16 进制转换）如下：

```
"main" #1 prio=5 os_prio=0 tid=0x00007f574400ae10 nid=0x4129 runnable [0x00007f574d100000]
   java.lang.Thread.State: RUNNABLE
        at java.io.RandomAccessFile.readBytes(Native Method)
        at java.io.RandomAccessFile.read(RandomAccessFile.java:377)
        at org.springframework.boot.loader.data.RandomAccessDataFile$FileAccess.read(RandomAccessDataFile.java:224)
        - locked <0x00000006c02a56a0> (a java.lang.Object)
        at org.springframework.boot.loader.data.RandomAccessDataFile$FileAccess.access$400(RandomAccessDataFile.java:206)
        at org.springframework.boot.loader.data.RandomAccessDataFile.read(RandomAccessDataFile.java:117)
        at org.springframework.boot.loader.data.RandomAccessDataFile.read(RandomAccessDataFile.java:101)
        at org.springframework.boot.loader.jar.JarFileEntries.getEntryData(JarFileEntries.java:209)
        at org.springframework.boot.loader.jar.JarFileEntries.getInputStream(JarFileEntries.java:189)
        at org.springframework.boot.loader.jar.JarFile.getInputStream(JarFile.java:223)
        - locked <0x00000006c05ab658> (a org.springframework.boot.loader.jar.JarFile)
        at org.reflections.vfs.ZipFile.openInputStream(ZipFile.java:27)
        at org.reflections.adapters.JavassistAdapter.getOfCreateClassObject(JavassistAdapter.java:98)
        at org.reflections.adapters.JavassistAdapter.getOfCreateClassObject(JavassistAdapter.java:24)
        at org.reflections.scanners.AbstractScanner.scan(AbstractScanner.java:30)
        at org.reflections.Reflections.scan(Reflections.java:253)
        at org.reflections.Reflections.scan(Reflections.java:202)
        at org.reflections.Reflections.<init>(Reflections.java:123)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.initReflections(MtConfigClientV2.java:173)
        - locked <0x00000006c00be680> (a java.lang.Object)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.scanAnnotation(MtConfigClientV2.java:130)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.init(MtConfigClientV2.java:107)
        at com.sankuai.meituan.config.MtConfigClient.initInvoker(MtConfigClient.java:113)
        at com.sankuai.meituan.config.MtConfigClient.init(MtConfigClient.java:67)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
```

strace 申请空间的线程栈

这里基本上就可以看出问题来了：MCC（美团统一配置中心）使用了 Reflec-

tions 进行扫包，底层使用了 Spring Boot 去加载 JAR。因为解压 JAR 使用 Inflater 类，需要用到堆外内存，然后使用 Btrace 去追踪这个类，栈如下：



btrace 追踪栈

然后查看使用 MCC 的地方，发现没有配置扫包路径，默认是扫描所有的包。于是修改代码，配置扫包路径，发布上线后内存问题解决。

3. 为什么堆外内存没有释放掉呢?

虽然问题已经解决了，但是有几个疑问：

- 为什么使用旧的框架没有问题?
- 为什么堆外内存没有释放?
- 为什么内存大小都是 64M，JAR 大小不可能这么大，而且都是一样大?
- 为什么 gperftools 最终显示使用的的内存大小是 700M 左右，解压包真的没有使用 malloc 申请内存吗?

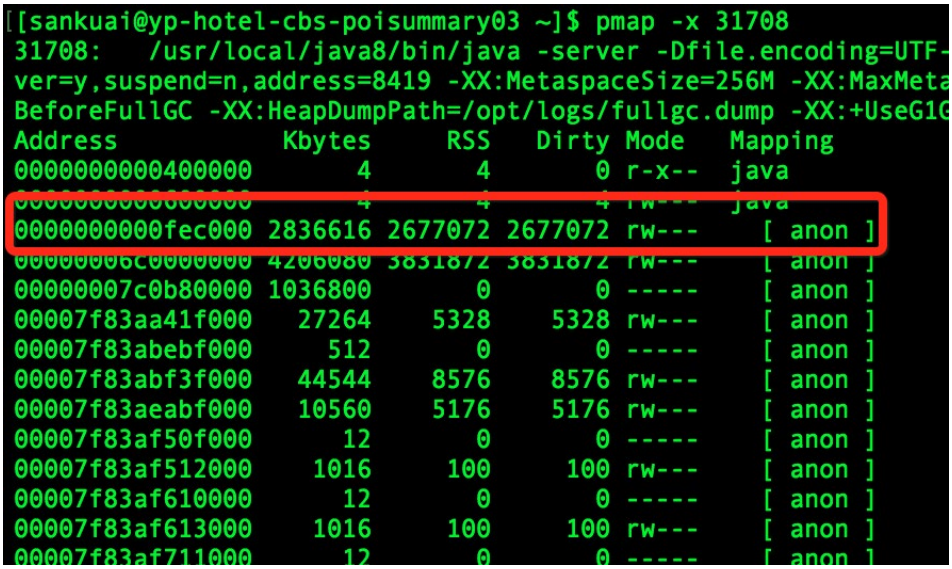
带着疑问，笔者直接看了一下 Spring Boot Loader 那一块的源码。发现 Spring Boot 对 Java JDK 的 InflaterInputStream 进行了包装并且使用了 Inflater，而 Inflater 本身用于解压 JAR 包的需要用到堆外内存。而包装之后的类 ZipInflaterInputStream 没有释放 Inflater 持有的堆外内存。于是笔者以为找到了原因，立马向 Spring Boot 社区反馈了这个 bug。但是反馈之后，笔者就发现 Inflater 这个对象本身实现了 finalize 方法，在这个方法中有调用释放堆外内存的逻辑。也就是说 Spring Boot 依赖于 GC 释放堆外内存。

笔者使用 jmap 查看堆内对象时，发现已经基本上没有 Inflater 这个对象了。于是就怀疑 GC 的时候，没有调用 finalize。带着这样的怀疑，笔者把 Inflater 进行包装在 Spring Boot Loader 里面替换成自己包装的 Inflater，在 finalize 进行打点监控，结果 finalize 方法确实被调用了。于是笔者又去看了 Inflater 对应的 C 代码，发现初始化的使用了 malloc 申请内存，end 的时候也调用了 free 去释放内存。

此刻，笔者只能怀疑 free 的时候没有真正释放内存，便把 Spring Boot 包装的 InflaterInputStream 替换成 Java JDK 自带的，发现替换之后，内存问题也得以解决了。

这时，再返过来看 gperftools 的内存分布情况，发现使用 Spring Boot 时，内存使用一直在增加，突然某个点内存使用下降了好多（使用量直接由 3G 降为 700M 左右）。这个点应该就是 GC 引起的，内存应该释放了，但是在操作系统层面并没有看到内存变化，那是不是没有释放到操作系统，被内存分配器持有了呢?

继续探究，发现系统默认的内存分配器（glibc 2.12 版本）和使用 gperftools 内存地址分布差别很明显，2.5G 地址使用 smaps 发现它是属于 Native Stack。内存地址分布如下：



gperftools 显示的内存地址分布

到此，基本上可以确定是内存分配器在捣鬼；搜索了一下 glibc 64M，发现 glibc 从 2.11 开始对每个线程引入内存池（64 位机器大小就是 64M 内存），原文如下：

- An enhanced dynamic memory allocation (malloc) behaviour enabling higher scalability across many sockets and cores. This is achieved by assigning threads their own memory pools and by avoiding locking in some situations. The amount of additional memory used for the memory pools (if any) can be controlled using the environment variables MALLOC_ARENA_TEST and MALLOC_ARENA_MAX. MALLOC_ARENA_TEST specifies that a test for the number of cores is performed once the number of memory pools reaches this value. MALLOC_ARENA_MAX sets the maximum number of memory pools used, regardless of the number of cores.

glib 内存池说明

按照文中所说去修改 MALLOC_ARENA_MAX 环境变量，发现没什么效果。查看 tcmalloc（gperftools 使用的内存分配器）也使用了内存池方式。

为了验证是内存池搞的鬼，笔者就简单写个不带内存池的内存分配器。使用命令 gcc zjbmalloc.c -fPIC -shared -o zjbmalloc.so 生成动态库，然后使用 export LD_PRELOAD=zjbmalloc.so 替换掉 glibc 的内存分配器。其中代码 Demo 如下：

```
#include<sys/mman.h>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
// 作者使用的 64 位机器，sizeof(size_t) 也就是 sizeof(long)
void* malloc ( size_t size )
{
   long* ptr = mmap( 0, size + sizeof(long), PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_
ANONYMOUS, 0, 0 );
   if (ptr == MAP_FAILED) {
      return NULL;
   }
   *ptr = size;                      // First 8 bytes contain length.
   return (void*)(&ptr[1]);          // Memory that is after length
variable
}

void *calloc(size_t n, size_t size) {
 void* ptr = malloc(n * size);
 if (ptr == NULL) {
      return NULL;
 }
 memset(ptr, 0, n * size);
 return ptr;
}
void *realloc(void *ptr, size_t size)
{
 if (size == 0) {
      free(ptr);
      return NULL;
 }
 if (ptr == NULL) {
      return malloc(size);
 }
 long *plen = (long*)ptr;
 plen--;                           // Reach top of memory
 long len = *plen;
 if (size <= len) {
      return ptr;
 }
 void* rptr = malloc(size);
 if (rptr == NULL) {
      free(ptr);
      return NULL;
 }
 rptr = memcpy(rptr, ptr, len);
 free(ptr);
 return rptr;
```

```
}

void free (void* ptr )
{
   if (ptr == NULL) {
       return;
   }
   long *plen = (long*)ptr;
   plen--;                              // Reach top of memory
   long len = *plen;                   // Read length
   munmap((void*)plen, len + sizeof(long));
}
```

通过在自定义分配器当中埋点可以发现其实程序启动之后应用实际申请的堆外内存始终在 700M-800M 之间，gperftools 监控显示内存使用量也是在 700M-800M 左右。但是从操作系统角度来看进程占用的内存差别很大（这里只是监控堆外内存）。

笔者做了一下测试，使用不同分配器进行不同程度的扫包，占用的内存如下：

内存分配器名称	没有扫全包	一次扫描全包	两次扫描全包
glibc	750M	1.5G	2.3G
tcmalloc	900M	2.0G	2.77g
自定义	1.7G	1.7G	1.7G

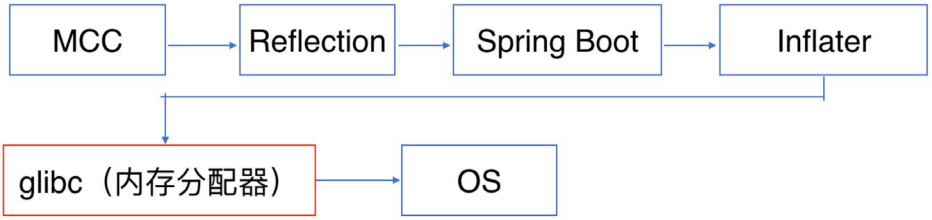
内存测试对比

为什么自定义的 malloc 申请 800M，最终占用的物理内存在 1.7G 呢?

因为自定义内存分配器采用的是 mmap 分配内存，mmap 分配内存按需向上取整到整数个页，所以存在着巨大的空间浪费。通过监控发现最终申请的页面数目在 536k 个左右，那实际上向系统申请的内存等于 512k * 4k（pagesize）= 2G。 为什么这个数据大于 1.7G 呢?

因为操作系统采取的是延迟分配的方式，通过 mmap 向系统申请内存的时候，系统仅仅返回内存地址并没有分配真实的物理内存。只有在真正使用的时候，系统产生一个缺页中断，然后再分配实际的物理 Page。

总结



流程图

整个内存分配的流程如上图所示。MCC 扫包的默认配置是扫描所有的 JAR 包。在扫描包的时候，Spring Boot 不会主动去释放堆外内存，导致在扫描阶段，堆外内存占用量一直持续飙升。当发生 GC 的时候，Spring Boot 依赖于 finalize 机制去释放了堆外内存；但是 glibc 为了性能考虑，并没有真正把内存归返到操作系统，而是留下来放入内存池了，导致应用层以为发生了“内存泄漏”。所以修改 MCC 的配置路径为特定的 JAR 包，问题解决。笔者在发表这篇文章时，发现 Spring Boot 的最新版本（2.0.5.RELEASE）已经做了修改，在 ZipInflaterInputStream 主动释放了堆外内存不再依赖 GC；所以 Spring Boot 升级到最新版本，这个问题也可以得到解决。

参考资料

1. GNU C Library (glibc)
2. Native Memory Tracking
3. Spring Boot
4. gperftools
5. Btrace

作者简介

纪兵，2015 年加入美团，目前主要从事酒店 C 端相关的工作。

美团点评效果广告实验配置平台的设计与实现

哲琪　仓魁　刘铮

一、背景

效果广告的主要特点之一是可量化，即广告系统的所有业务指标都是可以计算并通过数字进行展示的。因此，可以通过业务指标来表示广告系统的迭代效果。那如何在全量上线前确认迭代的结果呢？通用的方法是采用 AB 实验（如图 1）。所谓 AB 实验，是指单个变量具有两个版本 A 和 B 的随机实验。在实际应用中，是一种比较单个（或多个）变量多个版本的方法，通常是通过测试受试者对多个版本的反应，并确定多个版本中的哪个更有效。Google 工程师在 2000 年进行了首次 AB 实验，试图确定在其搜索引擎结果页上显示的最佳结果数。到了 2011 年，Google 进行了 7,000 多次不同的 AB 实验。现在很多公司使用“设计实验”的方法来制定营销决策，期望在实验样本上可以得到积极的转化结果，并且随着工具和专业知识在实验领域的发展，AB 实验已成为越来越普遍的一种做法。

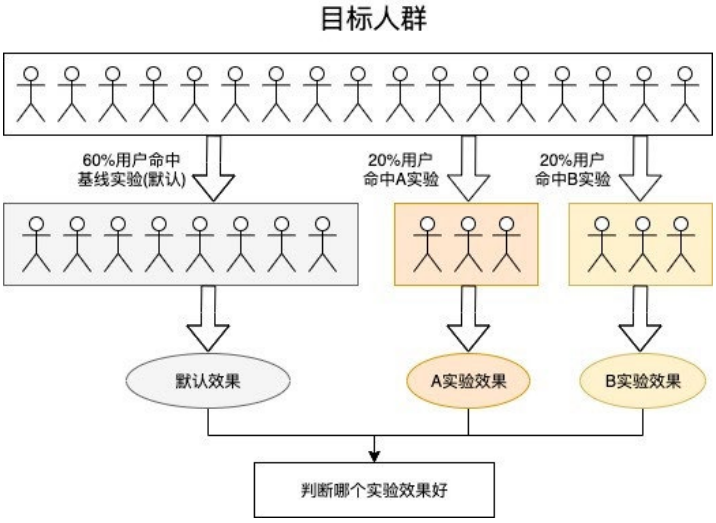


图 1　什么是 AB 实验

我们都知道，机器学习在广告投放中的作用是举足轻重的，广告收入的提升离不开算法模型的优化迭代，因此算法模型的迭代也需要进行 AB 实验。除了算法模型的迭代之外，工程中较大规模的重构和优化也需要 AB 实验来验证效果的有效性和正确性。此外，目前在大部分应用中，应用参数配置采用最多是单键值的配置方式，这种配置方式的确满足了大部分配置的需要，但是在结合业务需求的情况下，使用起来可能会很乏力。

因此，我们需要搭建一个平台（Wedge），来满足算法、工程在迭代过程中的实验需求，并且满足在不同流量下应用参数配置的需求。

二、方案设计

目标

Wedge 平台的目标如下：

- 支持各类算法实验场景，可灵活支持后续的功能扩展。
- 实验配置、使用、效果回收等全链路对使用者透明，降低解释成本。
- 提供不同流量下应用参数的配置，降低参数解析成本。
- 支持版本控制，可快速回滚。
- 提供简洁、易用的操作界面。

设计思路

在《Overlapping Experiment Infrastructure: More, Better, Faster Experimentation》中，Google 给出了一套通用的分层实验解决方案。我们以此为蓝本，结合美团点评效果广告的 LBS 特性，针对不同的业务场景，实现了更适合日常迭代的实验配置框架。目前，该框架已在搜索广告投放系统上全量上线。

实验分类

基于 Google 分层实验平台，结合实际需求进行了以下实验分类。

根据实验种类分类

- **水平实验**：类似于 Overlapping Layer 中的实验，是属于同个“层”的实验，实验是互斥的，在同一“层”上实验可以理解为是同一种实验，例如：关键词“层”表示这一层的实验都是关键词相关的，该层上存在实验 H1 和 H2，那么流量绝对不会同时命中 H1 和 H2。
- **垂直实验**：类似于 Non-overlapping Layer 中的实验，分布于不同“层”之间，实验是不互斥的，例如在关键词“层”和 CTR“层”上在相同的分桶上配置了实验 V1 和 V2，那么流量可以同时命中 V1 和 V2。
- **条件实验**：表示进入某“层”的实验需要满足某些条件，水平实验和垂直实验都可以是条件实验。
- 根据流量类别分类
- 这种分类主要了为了用户体验，使平台在操作上更加的简单、易用：
- **普通实验**：最基本的实验，根据流量类别进行配置。
- **引用实验**：流量分类是整个配置中心基础，但实际上存在一些实验是跨流量了，而引用实验则可以配置在不同的流量种类中。
- **全局实验**：可以理解为特殊的引用实验，全局实验在所有流量上都生效。

架构图

图 2 为整体架构图，比较便于大家理解，我们可以看到整体架构分为四层：

- **Web 层**：提供平台 UI，负责应用参数配置、实验配置、实验效果查看以及其他。
- **服务层**：提供权限控制、实验管理、拉取实验效果等功能。
- **存储层**：主要是数据存储功能。
- **业务层**：业务层结合 SDK 完成获取实验参数和获取应用参数的功能。

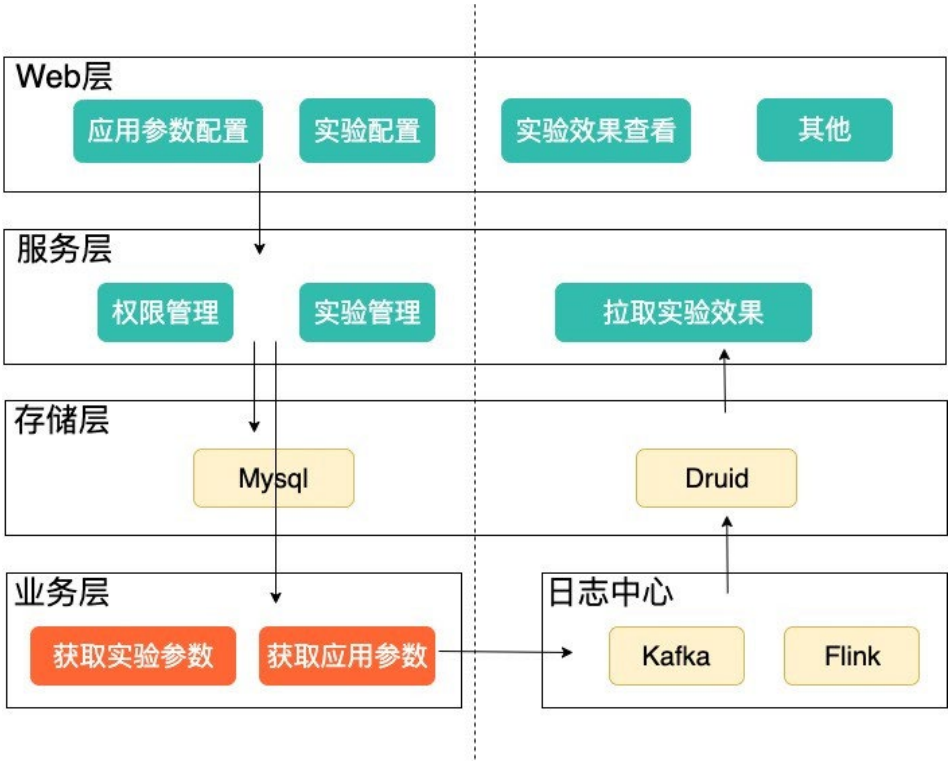


图 2　架构图

三、模型设计

1. 分流模型

实验模型

如图 3 所示，整体模型分为以下几个部分：

- App：表示一个应用，不同的应用对应的实验配置完全不同，首先从 App 上进行区分可以更加明确实验的归属。
- Scene：表示某一类流量的集合，例如：搜索、美食筛选、到综筛选等，在这些流量上配置的实验互不干扰。
- Layer：表示某一层（种）实验的集合。例如可以将在 matching 上做的实验放入 Matching Layer 中。流量命中时依次进入每个 Layer 获取实验配置参数，

此时的 Layer 更像一个抽象概念，与具体的业务或者逻辑相关。

- **条件 Layer**：是一种更加精细的流量控制方式，表示某一流量的某个或者某几个参数在满足一定条件下才会进行实验。进一步说就是相同 Scene 下，某一流量的参数 A 满足条件一时，采用一种实验配置策略；满足条件二时，采用另一种实验配置策略，那可以分为两层，如图 3 所示的 Layer_3 和 Layer_4。例如：某流量需要在城市北京单独做实验，这种情况下，可以分为参数相同但是先决条件（即城市）互斥的两个 Layer。此时的 Layer 在抽象的基础上更加的具体化。
- **Exp**：表示具体实验，包括实验的分桶、实验参数、是否为垂直流量等等。

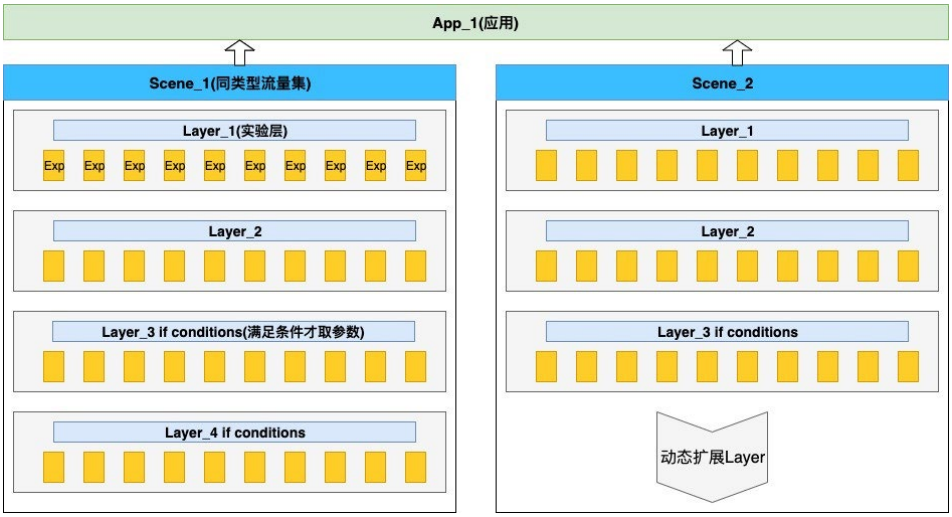


图 3　实验模型

水平、垂直分流模型

如图 4 所示，水平、垂直分流模型分为以下两个部分：

- **仅包含水平实验**：最基础的实验需求，全部实验独占一个 Layer，每个实验覆盖若干个桶，例如图 4 中的 Layer_1，将流量分为 10 份，包含三个实验，这三个实验分别占用 3、3、4 份流量。
- **同时包含水平、垂直实验**：一个 Layer 中同时包含垂直、水平两种类型的实

验。例如图 4 中的 Layer_2 和 Layer_3，将最后的 4 份流量用来做垂直实验，包含两个垂直实验，分别是 Exp_6 和 Exp_7。

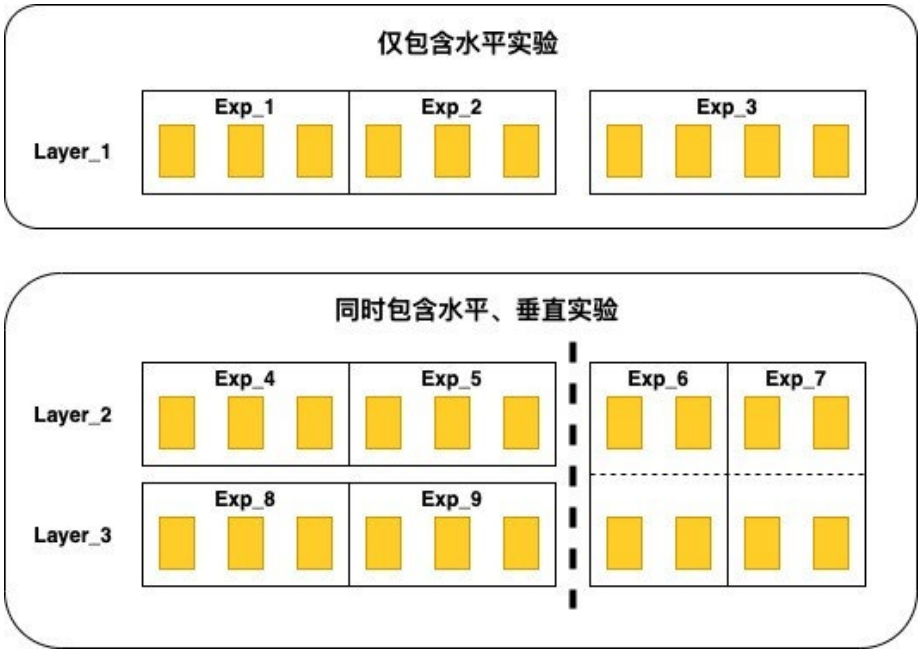


图 4　水平、垂直分流模型

2. 实验命中模型

实验命中模型是指，当一个请求过来时，返回全局统一的实验参数。所有的请求都会平均地落入每一个分桶中，并且不同的 Layer 之间能够保证流量的正交。

名词解释

Bucket	分桶
流量正交	流量在不同Layer之间的分桶是完全相互独立的
Hash优先级	表示计算Hash值的先后顺序，用于垂直和水平实验切分
Hash因子	分桶的唯一标识
Hash串	计算Hash的字符串
取模数	Hash值计算之后的除数

- Hash 优先级：在实验命中过程中，第一次 Hash 首先判断命中垂直流量，如果没有命中，则进行第二次 Hash 再判断水平流量。
- Hash 因子：目前美团侧一般情况下为 uuid，点评侧为 dpid。
- 垂直流量 Hash 串：Hash 因子 +scene_id。
- 水平流量 Hash 串：Hash 因子 +scene_id+layer_id+layer_name。
- 取模数：在 Hash 过程中，垂直流量按照总 Bucket（默认取值 100）取模；水平流量按照总 Bucket 数减去垂直流量 Bucket 数取模。这样的命中模型能保证无论是垂直的 Bucket，还是水平的 Bucket 都是全局的 1%。

实例解析

以最复杂的流量分配为例，如图 5，水平、垂直流量各占全局 50% 流量。

水平流量上包含两个实验：Exp_1、Exp_2 各占全局 20% 流量，还有 10% 流量未分配实验，垂直流量与水平流量相同。

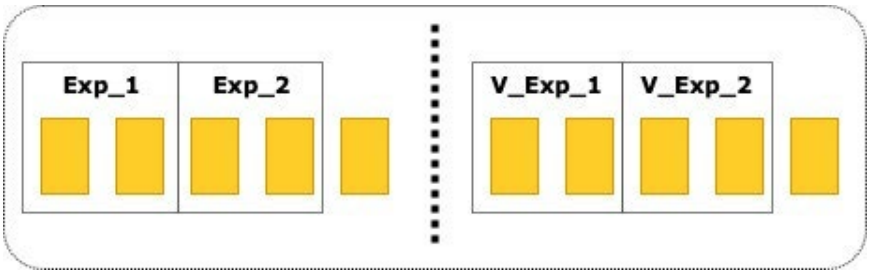


图 5　流量分配示例

典型实验命中如图 6 所示：

- 当一个请求过来时，命中顺序按照 Hash 优先级为先垂直流量后水平流量。
- 首先利用垂直流量 Hash 串进行 Hash 并取模得到一个 hashNum，如果命中了 50 ~ 99，就会认为命中了垂直流量，直接返回 V_Exp_1、V_Exp_2 或者未命中。
- 如果没有命中 50 ~ 99，则认为命中了水平流量。再利用水平 Hash 因子进行 Hash 并取模得到一个 hashNum，根据配置直接返回 Exp_1、Exp_2 或者未命中。

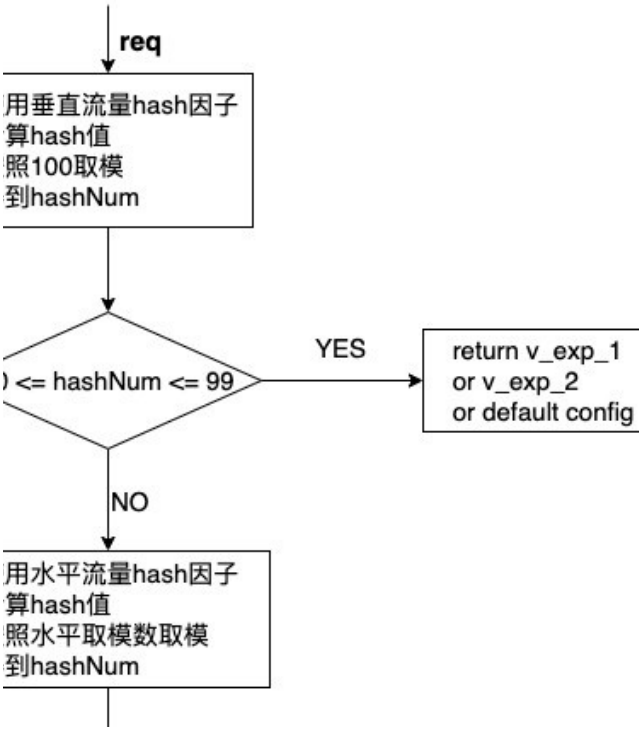


图 6　实验命中流程

应用参数模型

应用参数模型如图 7 所示，分为两层结构：

- **全局参数**：所有流量都能拿到的参数。
- **同类型流量参数**：相同类型的流量能拿到的参数。

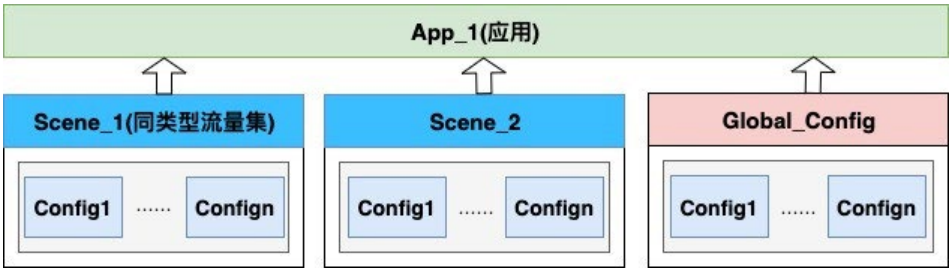


图 7　应用参数模型

值得注意的是，应用参数的两个层级以及之前提到的实验参数可能出现重复的情况，在出现重复的情况下参数的优先级为：实验参数 > 同类型流量参数 > 全局参数。

3. 回滚模型

很明显，平台的分流模型是层次结构的，所以在数据设计上也是每一层一个table。这样就会出现一个问题，一般的参数配置回滚都是单值的回滚，但是存在多个表的情况下没有办法这么简单地去回滚。

因此，我们设计了如下的回滚模型：

1. 首先在配置发布时，会将所有修改的表名、列名、列类型、列新旧值、修改类型存入表中。
2. 回滚时获取上次发布的所有修改的表名、列名、列类型、列新旧值、修改类型，反向操作数据库，达到回滚的目的。

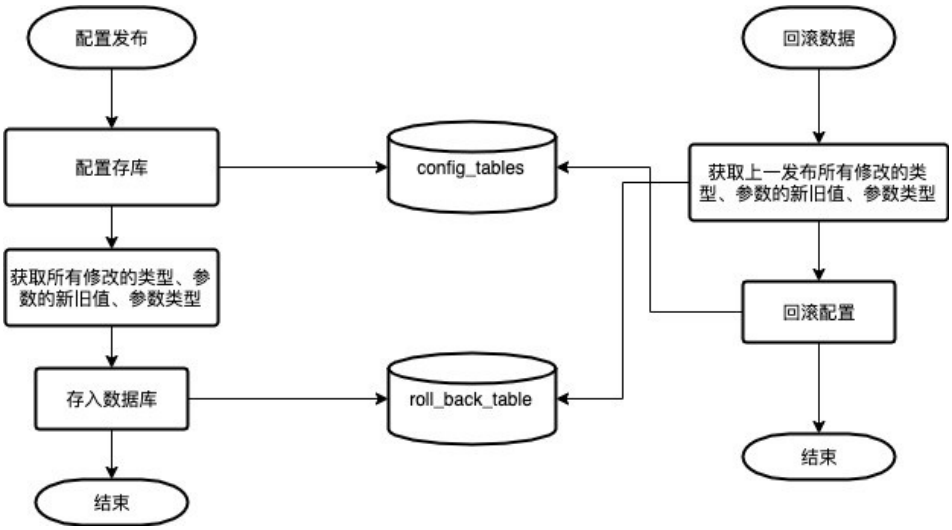


图 8　回滚模型

数据库表的设计如下：

列名	数据类型	说明
operation_id	BIGINT	表示配置发布的ID
old_value	TEXT	旧值
column_name	VARCHAR	列名
new_value	TEXT	新值
operation_step	INT	修改的数据表的顺序
table_name	VARCHAR	修改的数据表名
operation_type	INT	修改类型，包括增加、修改、删除
value_type	INT	值类型

四、AB 实验实时效果

实时效果指标包含实际 CTR、预估 CTR、请求 PV、广告密度、有效曝光、RPS 等，由于很多数据分布在不同的日志中，所以需要实时处理。

广告的打点一般分为请求（PV）打点、SPV（Server PV）打点、CPV（Client PV）曝光打点和 CPV 点击打点，在所有打点中都会包含一个流量的 requestId 和命中的实验路径。根据 requestId 和命中的实验路径可以将所有的日志进行 join，得到一个 request 中需要的所有数据，然后将数据存入 Durid 中。

计算各项效果指标，就是在日志 join 后带有实验路径的数据上做 OLAP。为了支持高效的实时查询，平台采用时序数据库 Druid 作为底层存储。Druid 作为分布式时序数据库，提供了丰富的 OLAP 能力和强悍的性能。Druid 将数据分为时间戳、维度和指标三个部分，其中维度多用于过滤，指标用于聚合和计算等。平台将数据中的实验路径同其他用于过滤的字段一同作为维度，结合时间戳和指标字段，完成指定标签的广告效果指标计算。

五、总结

目前 Wedge 平台已经完全上线，除了满足目标之外，还带来了以下的成果：

- 配合完成算法工程架构调整、更新流大版本升级。
- 算法各方向之间可以任意做垂直实验，满足算法灵活、快速迭代。
- 具备配置审核功能，保证配置发布的正确性。

六 . 作者简介

哲琪、仓魁、刘铮，美团点评效果广告引擎团队工程师。

七 . 招聘信息

美团点评效果广告引擎团队，招募出色的后端开发工程师。我们希望您：具有扎实的后端服务开发功底，能够熟练使用 Java 或 C++ 开发语言，致力于互联网技术领域。

有兴趣的同学，欢迎投递简历至：tech@meituan.com（邮件标题注明：美团点评效果广告引擎团队）

根因分析初探：一种报警聚类算法在业务系统的落地实施

刘场　千钊

背景

众所周知，日志是记录应用程序运行状态的一种重要工具，在业务服务中，日志更是十分重要。通常情况下，日志主要是记录关键执行点、程序执行错误时的现场信息等。系统出现故障时，运维人员一般先查看错误日志，定位故障原因。当业务流量小、逻辑复杂度低时，应用出现故障时错误日志一般较少，运维人员一般能够根据错误日志迅速定位到问题。但是，随着业务逻辑的迭代，系统接入的依赖服务不断增多，引入的组件不断增多，当系统出现故障时（如 Bug 被触发、依赖服务超时等等），错误日志的量级会急剧增加。极端情况下甚至出现“疯狂报错”的现象，这时候错误日志的内容会存在相互掩埋、相互影响的问题，运维人员面对报错一时难以理清逻辑，有时甚至顾此失彼，没能第一时间解决最核心的问题。

错误日志是系统报警的一种，实际生产中，运维人员能够收到的报警信息多种多样。如果在报警流出现的时候，通过处理程序，将报警进行聚类，整理出一段时间内的报警摘要，那么运维人员就可以在摘要信息的帮助下，先对当前的故障有一个大致的轮廓，再结合技术知识与业务知识定位故障的根本原因。

围绕上面描述的问题，以及对于报警聚类处理的分析假设，本文主要做了以下事情：

1. 选定聚类算法，简单描述了算法的基本原理，并给出了针对报警日志聚类的一种具体的实现方案。
2. 在分布式业务服务的系统下构造了三种不同实验场景，验证了算法的效果，并且对算法的不足进行分析阐述。

目标

对一段时间内的报警进行聚类处理，将具有相同根因的报警归纳为能够涵盖报警内容的泛化报警（Generalized Alarms），最终形成仅有几条泛化报警的报警摘要。如下图 1 所示意。

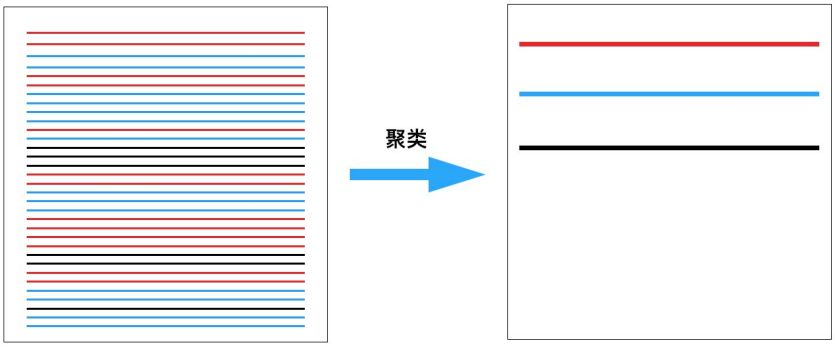


图 1

我们希望这些泛化报警既要具有很强的概括性，同时尽可能地保留细节。这样运维人员在收到报警时，便能快速定位到故障的大致方向，从而提高故障排查的效率。

设计

如图 2 所示，异常报警根因分析的设计大致分为四个部分：收集报警信息、提取报警信息的关键特征、聚类处理、展示报警摘要。

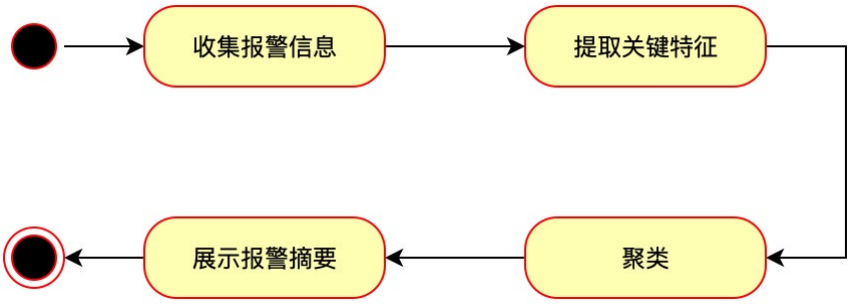


图 2

算法选择

聚类算法采用论文“Clustering Intrusion Detection Alarms to Support Root Cause Analysis [KLAUS JULISCH, 2002]”中描述的根因分析算法。该算法基于一个假设：将报警日志集群经过泛化，得到的泛化报警能够表示报警集群的主要特征。以下面的例子来说明，有如下的几条报警日志：

```
server_room_a-biz_tag-online02 Thrift get deal ProductType deal error.
server_room_b-biz_tag-offline01 Pigeon query deal info error.
server_room_a-biz_tag-offline01 Http query deal info error.
server_room_a-biz_tag-online01 Thrift query deal info error.
server_room_b-biz_tag-offline02 Thrift get deal ProductType deal error.
```

我们可以将这几条报警抽象为：“全部服务器 网络调用 故障”，该泛化报警包含的范围较广；也可以抽象为：“server_room_a 服务器 网络调用 产品信息获取失败”和“server_room_b 服务器 RPC 获取产品类型信息失败”，此时包含的范围较小。当然也可以用其他层次的抽象来表达这个报警集群。

我们可以观察到，抽象层次越高，细节越少，但是它能包含的范围就越大；反之，抽象层次越低，则可能无用信息越多，包含的范围就越小。

这种抽象的层次关系可以用一些有向无环图（DAG）来表达，如图 3 所示：

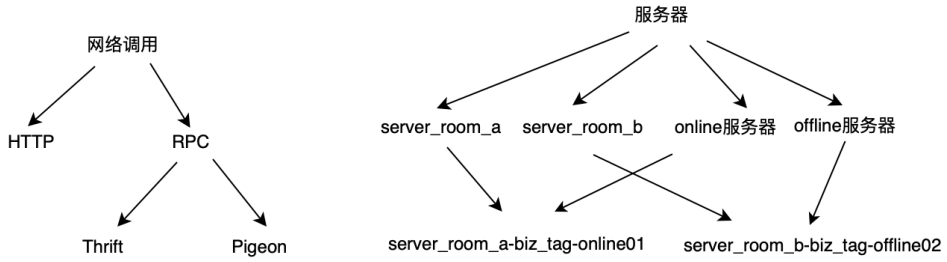


图 3　泛化层次结构示例

为了确定报警聚类泛化的程度，我们需要先了解一些定义：

- 属性（Attribute）：构成报警日志的某一类信息，如机器、环境、时间等，文中用 Ai 表示。

- 值域（Domain）：属性 Ai 的域（即取值范围），文中用 Dom(Ai) 表示。
- 泛化层次结构（Generalization Hierarchy）：对于每个 Ai 都有一个对应的泛化层次结构，文中用 Gi 表示。
- 不相似度（Dissimilarity）：定义为 d(a1, a2)。它接受两个报警 a1、a2 作为输入，并返回一个数值量，表示这两个报警不相似的程度。与相似度相反，当 d(a1, a2) 较小时，表示报警 a1 和报警 a2 相似。为了计算不相似度，需要用户定义泛化层次结构。

为了计算 d(a1, a2)，我们先定义两个属性的不相似度。令 x1、x2 为某个属性 Ai 的两个不同的值，那么 x1、x2 的不相似度为：在泛化层次结构 Gi 中，通过一个公共点父节点 p 连接 x1、x2 的最短路径长度。即 d(x1, x2) := min{d(x1, p) + d(x2, p) | p ∈ Gi, x1 ⊴ p, x2 ⊴ p}。例如在图 3 的泛化层次结构中，d（“Thrift”, “Pigeon”) = d(“RPC”, “Thrift”) + d(“RPC”, “Pigeon”) = 1 + 1 = 2。

对于两个报警 a1、a2，其计算方式为：

$$d(\mathbf{a_1}, \mathbf{a_2}) := \sum_{i=1}^{n} d(\mathbf{a_1}[A_i], \mathbf{a_2}[A_i])$$

公式 1

例如：a1 = (“server_room_b-biz_tag-offline02”, “Thrift”), a2 = (“server_room_a-biz_tag-online01”, “Pigeon”), 则 d(a1, a2) = d(“server_room_b-biz_tag-offline02”, “server_room_a-biz_tag-online01”) + d((“Thrift”, “Pigeon”) = d(“server_room_b-biz_tag-offline02”, “服务器”) + d(“server_room_a-biz_tag-online01”, “服务器”) + d(“RPC”, “Thrift”) + d(“RPC”, “Pigeon”) = 2 + 2 + 1 + 1 = 6。

我们用 C 表示报警集合，g 是 C 的一个泛化表示，即满足 ∀ a ∈ C, a ⊴ g。以报警集合{“dx-trip-package-api02 Thrift get deal list error.”, “dx-trip-package-api01 Thrift get deal list error.”}为例，“dx 服务器 thrift 调用 获取产品信息失败”是一个泛化表示，“服务器 网络调用 获取产品信息失败”也是一个泛化表

示。对于某个报警聚类来说，我们希望获得既能够涵盖它的集合又有最具象化的表达的泛化表示。为了解决这个问题，定义以下两个指标：

$$\bar{d}(\mathbf{g}, \mathcal{C}) \ := \ 1/|\mathcal{C}| \times \sum_{\mathbf{a} \in \mathcal{C}} d(\mathbf{g}, \mathbf{a})$$

$$H(\mathcal{C}) \ := \ \min\{\, \bar{d}(\mathbf{g}, \mathcal{C}) \,|\, \mathbf{g} \in \mathsf{X}_{i=1}^{n} \mathrm{Dom}(A_i),\ \forall \mathbf{a} \in \mathcal{C} : \mathbf{a} \preceq \mathbf{g} \,\}$$

公式 2

H© 值最小时对应的 g，就是我们要找的最适合的泛化表示，我们称 g 为 C 的“覆盖”(Cover)。

基于以上的概念，将报警日志聚类问题定义为：定义 L 为一个日志集合，min_size 为一个预设的常量，Gi(i = 1, 2, 3…⋯n) 为属性 Ai 的泛化层次结构，目标是找到一个 L 的子集 C，满足 |C| >= min_size，且 H© 值最小。min_size 是用来控制抽象程度的，极端情况下如果 min_size 与 L 集合的大小一样，那么我们只能使用终极抽象了，而如果 min_size = 1，则每个报警日志是它自己的抽象。找到一个聚类之后，我们可以去除这些元素，然后在 L 剩下的集合里找其他的聚类。

不幸的是，这是个 NP 完全问题，因此论文提出了一种启发式算法，该算法满足 |C| >= min_size，使 H© 值尽量小。

算法描述

1. 算法假设所有的泛化层次结构 Gi 都是树，这样每个报警集群都有一个唯一的、最顶层的泛化结果。
2. 将 L 定义为一个原始的报警日志集合，算法选择一个属性 Ai，将 L 中所有报警的 Ai 值替换为 Gi 中 Ai 的父值，通过这一操作不断对报警进行泛化。
3. 持续步骤 2 的操作，直到找到一个覆盖报警数量大于 min_size 的泛化报警为止。
4. 输出步骤 3 中找到的报警。

算法伪代码如下所示：

```
输入：报警日志集合 L，min_size，每个属性的泛化层次结构 G1,.....,Gn
输出：所有符合条件的泛化报警
T := L;                // 将报警日志集合保存至表 T
for all alarms a in T do
    a[count] := 1;    // "count" 属性用于记录 a 当前覆盖的报警数量
while ∀a ∈ T : a[count] < min_size do {
    使用启发算法选择一个属性 Ai;
    for all alarms a in T do
        a[Ai] := parent of a[Ai] in Gi;
        while identical alarms a, a' exist do
            Set a[count] := a[count] + a'[count];
            delete a' from T;
}
```

其中第 7 行的启发算法为：

```
首先计算 Ai 对应的 Fi
fi(v) := SELECT sum(count) FROM T WHERE Ai = v    // 统计在 Ai 属性上值为 v 的报警的数量
Fi := max{fi(v) | v ∈ Dom(Ai)}
选择 Fi 值最小的属性 Ai
```

这里的逻辑是：如果有一个报警 a 满足 a[count]>= min_size，那么对于所有属性 Ai，均能满足 Fi >= fi(a[Ai]) >= min_size。反过来说，如果有一个属性 Ai 的 Fi 值小于 min_size，那么 a[count] 就不可能大于 min_size。所以选择 Fi 值最小的属性 Ai 进行泛化，有助于尽快达到聚类的条件。

此外，关于 min_size 的选择，如果选择了一个过大的 min_size，那么会迫使算法合并具有不同根源的报警。另一方面，如果过小，那么聚类可能会提前结束，具有相同根源的报警可能会出现在不同的聚类中。

因此，设置一个初始值，可以记作 ms0。定义一个较小的值 $\varepsilon(0 < \varepsilon < 1)$，当 min_size 取值为 ms0、ms0 * (1 − ε)、ms0 * (1 + ε) 时的聚类结果相同时，我们就说此时聚类是 ε− 鲁棒的。如果不相同，则使 ms1 = ms0 * (1 − ε)，重复这个测试，直到找到一个鲁棒的最小值。

需要注意的是，ε− 鲁棒性与特定的报警日志相关。因此，给定的最小值，可能

相对于一个报警日志来说是鲁棒的，而对于另一个报警日志来说是不鲁棒的。

实现

1. 提取报警特征

根据线上问题排查的经验，运维人员通常关注的指标包括时间、机器（机房、环境）、异常来源、报警日志文本提示、故障所在位置（代码行数、接口、类）、Case 相关的特殊 ID（订单号、产品编号、用户 ID 等等）等。

但是，我们的实际应用场景都是线上准实时场景，时间间隔比较短，因此我们不需要关注时间。同时，Case 相关的特殊 ID 不符合我们希望获得一个抽象描述的要求，因此也无需关注此项指标。

综上，我们选择的特征包括：机房、环境、异常来源、报警日志文本关键内容、故障所在位置（接口、类）共 5 个。

2. 算法实现

（1）提取关键特征

我们的数据来源是日志中心已经格式化过的报警日志信息，这些信息主要包含：报警日志产生的时间、服务标记、在代码中的位置、日志内容等。

- 故障所在位置

优先查找是否有异常堆栈，如存在则查找第一个本地代码的位置；如果不存在，则取日志打印位置。

- 异常来源

获得故障所在位置后，优先使用此信息确定异常报警的来源（需要预先定义词典支持）；如不能获取，则在日志内容中根据关键字匹配（需要预先定义词典支持）。

- 报警日志文本关键内容

优先查找是否有异常堆栈，如存在，则查找最后一个异常（通常为真正的故障原因）；如不能获取，则在日志中查找是否存在“code=……,message=……”这样形式的错误提示；如不能获取，则取日志内容的第一行内容（以换行符为界），并去除其中可能存在的 Case 相关的提示信息

- 提取“机房和环境”这两个指标比较简单，在此不做赘述。

（2）聚类算法

算法的执行，我们以图 4 来表示。

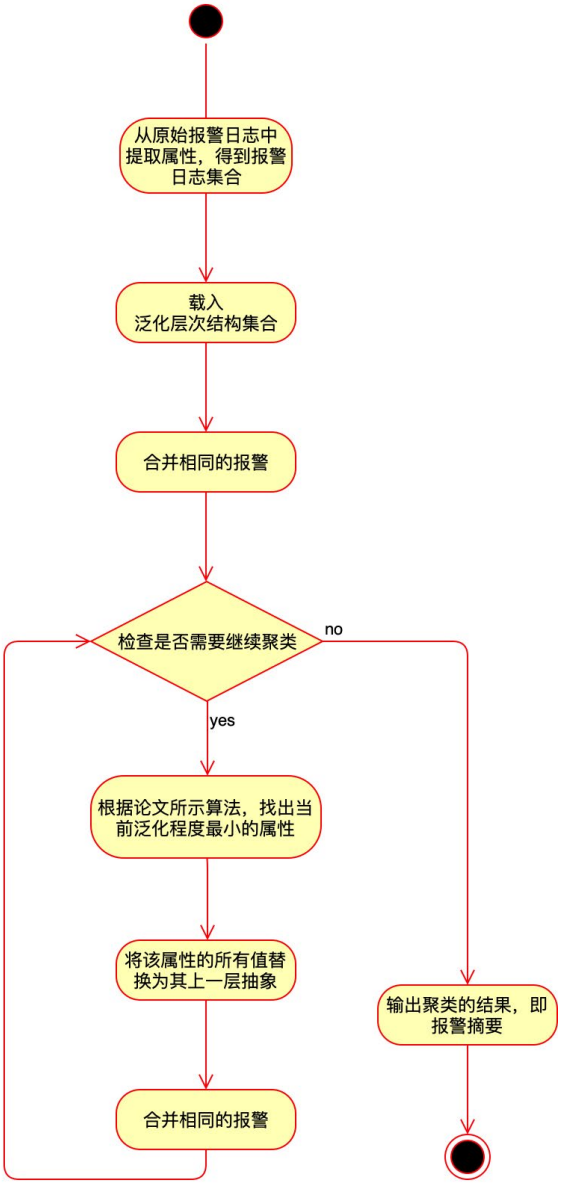


图 4　报警日志聚类流程图

（3）min_size 选择

考虑到日志数据中可能包含种类极多，且根据小规模数据实验表明，min_size = 1/5 * 报警日志数量时，算法已经有较好的表现，再高会增加过度聚合的风险，因此我们取 min_size = 1/5 * 报警日志数量，ε 参考论文中的实验，取 0.05。

（4）聚类停止条件

考虑到部分场景下，报警日志可能较少，因此 min_size 的值也较少，此时聚类已无太大意义，因此设定聚类停止条件为：聚类结果的报警摘要数量小于等于 20 或已经存在某个类别的 count 值达到 min_size 的阈值，即停止聚类。

3. 泛化层次结构

泛化层次结构，用于记录属性的泛化关系，是泛化时向上抽象的依据，需要预先定义。

根据实验所用项目的实际使用环境，我们定义的泛化层次结构如下：

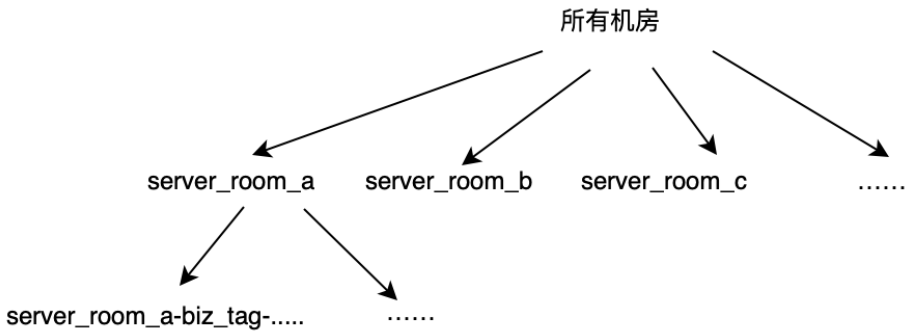


图 5　机房泛化层次结构

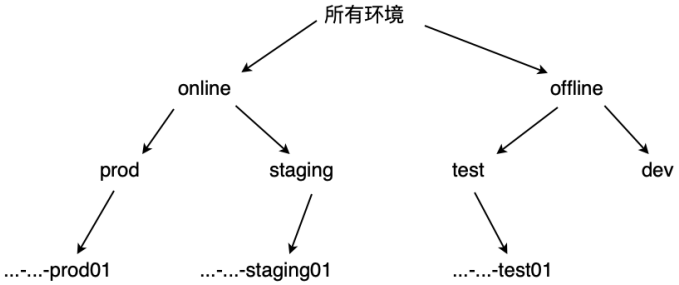


图 6　环境泛化层次结构

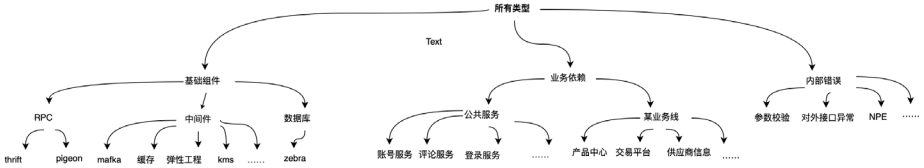


图 7　错误来源泛化层次结构

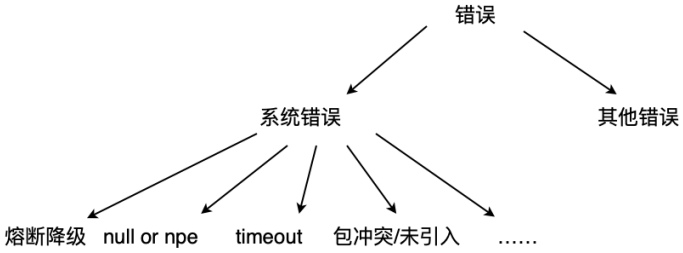


图 8　日志文本摘要泛化层次结构

“故障所在位置”此属性无需泛化层次结构，每次泛化时直接按照包路径向上层截断，直到系统包名。

实验

以下三个实验均使用 C 端 API 系统。

1. 单依赖故障

实验材料来自于线上某业务系统真实故障时所产生的大量报警日志。

- 环境：线上
- 故障原因：产品中心线上单机故障
- 报警日志数量：939 条

部分原始报警日志如图 9 所示，初次观察时，很难理出头绪。

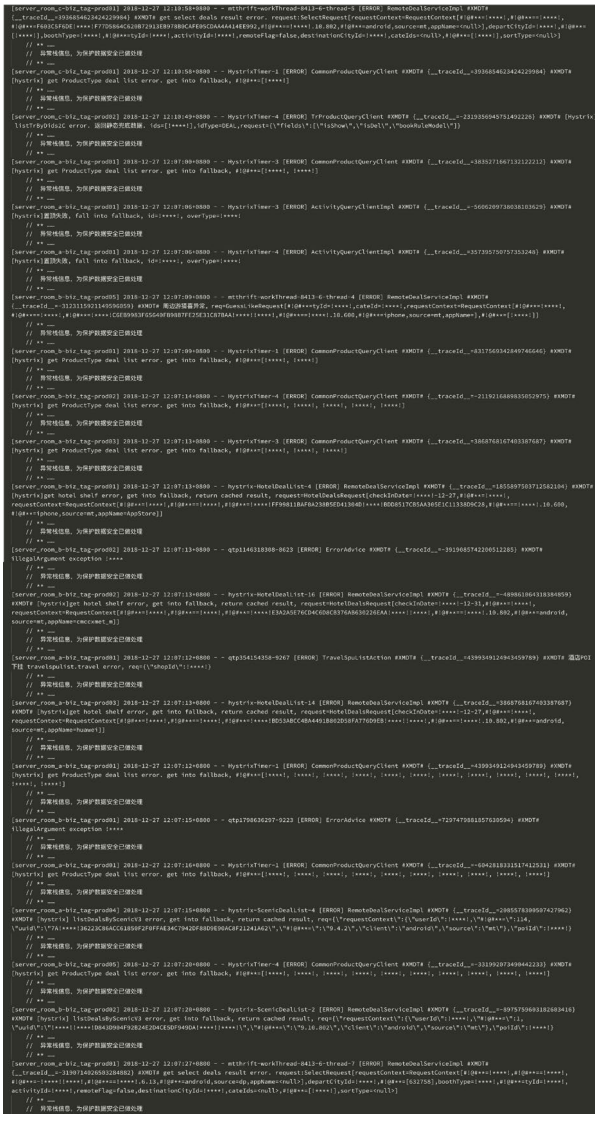


图 9　单依赖故障报警日志节选

经过聚类后的报警摘要如表 1 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	产品中心	Prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getProductType execution timeout after waiting for 150ms.	249
2	所有机房	业务插件	Prod	com.*.*.*.PluginRegistry.lambda	java.lang.IllegalArgumentException: 未找到业务插件 : 所有产品类型	240
3	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: TrQueryClient.listTrByDids2C execution timeout after waiting for 1000ms.	145
4	所有机房	对外接口（猜喜 / 货架 / 目的地）	Prod	com.*.*.*.RemoteDealServiceImpl	com.netflix.hystrix.exception.HystrixTimeoutException: ScenicDealList.listDealsByScenic execution timeout after waiting for 300ms.	89
5	所有机房	产品中心	Prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.listTrByDids2C execution timeout after waiting for 1000ms.	29
6	所有机房	产品中心	Prod	com.*.*.*.ActivityQueryClientImpl	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	21

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
7	所有机房	产品中心	prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	21
8	所有机房	对外接口（猜喜 / 货架 / 目的地）	Prod	com.*.*.*.RemoteDealServiceImpl	com.netflix.hystrix.exception.HystrixTimeoutException: HotelDealList.hotelShelf execution timeout after waiting for 500ms.	17
9	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	Caused by: java.lang.InterruptedException	16
10	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	Caused by: java.lang.InterruptedException	13

我们可以看到前三条报警摘要的 Count 远超其他报警摘要，并且它们指明了故障主要发生在产品中心的接口。

2. 无相关的多依赖同时故障

实验材料为利用故障注入工具，在 Staging 环境模拟运营置顶服务和 A/B 测试服务同时产生故障的场景。

- 环境：Staging（使用线上录制流量和压测平台模拟线上正常流量环境）
- 模拟故障原因：置顶与 A/B 测试接口大量超时
- 报警日志数量：527 条

部分原始报警日志如图 10 所示：

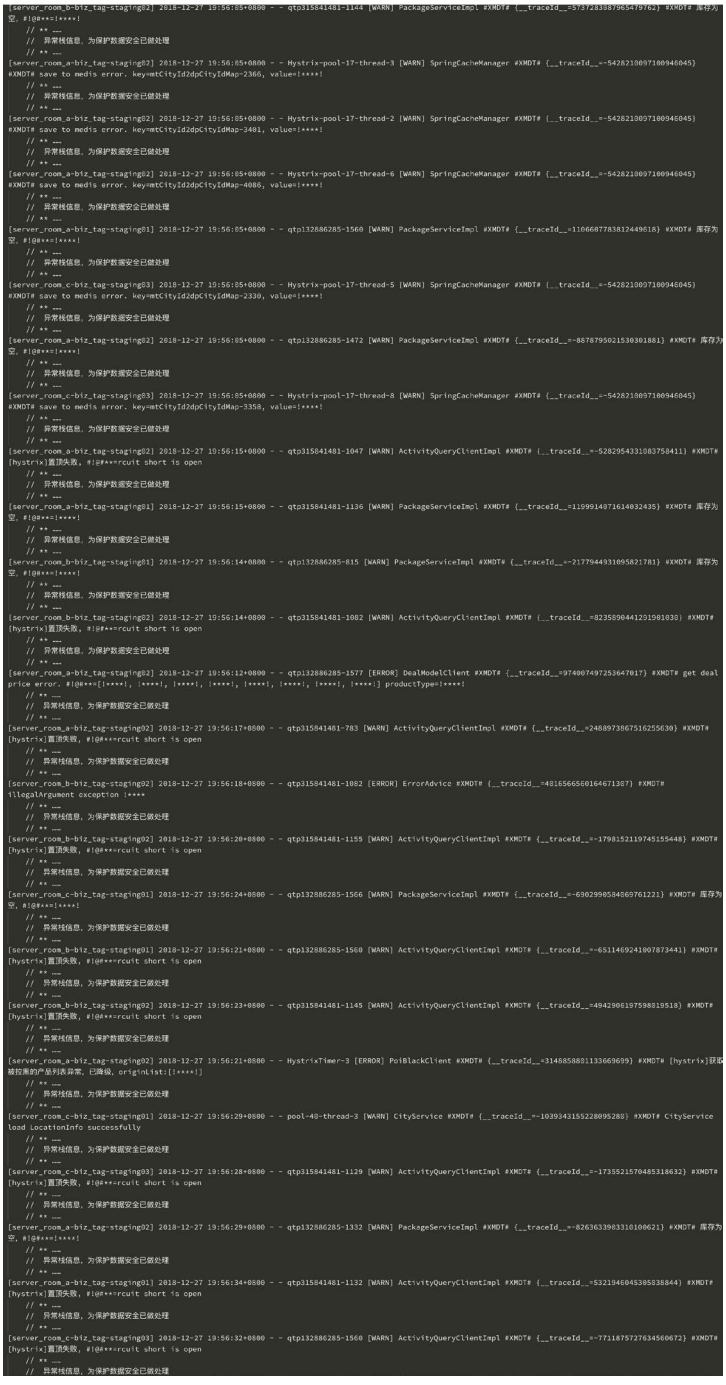


图 10　无相关的多依赖同时故障报警日志节选

经过聚类后的报警摘要如表 2 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	运营活动	Staging	com.*.*.*.ActivityQueryClientImpl	[hystrix] 置顶失败 , circuit short is open	291
2	所有机房	A/B 测试	Staging	com.*.*.*.AbExperimentClient	[hystrix] tripExperiment error, circuit short is open	105
3	所有机房	缓存	Staging	com.*.*.*.CacheClientFacade	com.netflix.hystrix.exception.HystrixTimeoutException: c-cache-rpc.common_deal_base.rpc execution timeout after waiting for 1000ms.	15
4	所有机房	产品信息	Staging	com.*.*.*.queryDealModel	Caused by: com.meituan.service.mobile.mtthrift.netty.exception.RequestTimeoutException: request timeout	14
5	所有机房	产品中心	Staging	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	9
6	所有机房	产品中心	Staging	com.*.*.*.getOrderForm	java.lang.IllegalArgumentException: 产品无库存	7
7	所有机房	弹性工程	Staging	com.*.*.*.PreSaleChatClient	com.netflix.hystrix.exception.HystrixTimeoutException: CustomerService.PreSaleChat execution timeout after waiting for 50ms.	7
8	所有机房	缓存	Staging	com.*.*.*.SpringCacheManager	Caused by: java.net.SocketTimeoutException: Read timed out	7
9	所有机房	产品信息	Staging	com.*.*.*.queryDetailUrlVO	java.lang.IllegalArgumentException: 未知的产品类型	2
10	所有机房	产品信息	Staging	com.*.*.*.queryDetailUrlVO	java.lang.IllegalArgumentException: 无法获取链接地址	1

从上表可以看到，前两条报警摘要符合本次试验的预期，定位到了故障发生的原因。说明在多故障的情况下，算法也有较好的效果。

3. 中间件与相关依赖同时故障

实验材料为利用故障注入工具，在 Staging 环境模拟产品中心服务和缓存服务同时产生超时故障的场景。

- 环境：Staging（使用线上录制流量和压测平台模拟线上正常流量环境）
- 模拟故障原因：产品中心所有接口超时，所有缓存服务超时
- 报警日志数量：2165

部分原始报警日志如图 11 所示：

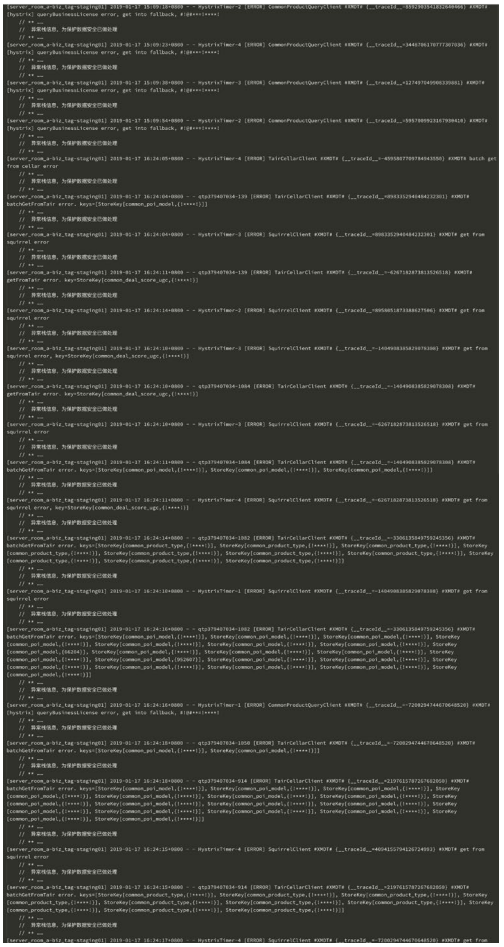


图 11　中间件与相关依赖同时故障报警日志节选

经过聚类后的报警摘要如表 3 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	Squirrel	Staging	com.*.*.*.cache	Timeout	491
2	所有机房	Cellar	Staging	com.*.*.*.cache	Timeout	285
3	所有机房	Squirrel	Staging	com.*.*.*.TdcServiceImpl	Other Exception	149
4	所有机房	评论	Staging	com.*.*.*.cache	Timeout	147
5	所有机房	Cellar	Staging	com.*.*.*.TdcServiceImpl	Other Exception	143
6	所有机房	Squirrel	Staging	com.*.*.*.PoiManagerImpl	熔断	112
7	所有机房	产品中心	Staging	com.*.*.*.CommonProductQueryClient	Other Exception	89
8	所有机房	评论	Staging	com.*.*.*.TrDealProcessor	Other Exception	83
9	所有机房	评论	Staging	com.*.*.*.poi.PoiInfoImpl	Other Exception	82
10	所有机房	产品中心	Staging	com.*.*.*.client	Timeout	74

从上表可以看到，缓存（Squirrel 和 Cellar 双缓存）超时最多，产品中心的超时相对较少，这是因为我们系统针对产品中心的部分接口做了兜底处理，当超时发生时后先查缓存，如果缓存查不到会穿透调用一个离线信息缓存系统，因此产品中心超时总体较少。

综合上述三个实验得出结果，算法对于报警日志的泛化是具有一定效果。在所进行实验的三个场景中，均能够定位到关键问题。但是依然存在一些不足，报警摘要中，有的经过泛化的信息过于笼统（比如 Other Exception）。

经过分析，我们发现主要的原因有：其一，对于错误信息中关键字段的提取，在一定程度上决定了向上泛化的准确度。其二，系统本身日志设计存在一定的局限性。

同时，在利用这个泛化后的报警摘要进行分析时，需要使用者具备相应领域的知识。

未来规划

本文所关注的工作，主要在于验证聚类算法效果，还有一些方向可以继续完善和优化：

1. 日志内容的深度分析。本文仅对报警日志做了简单的关键字提取和人工标记，未涉及太多文本分析的内容。我们可以通过使用文本分类、文本特征向量相似度等，提高日志内容分析的准确度，提升泛化效果。
2. 多种聚类算法综合使用。本文仅探讨了处理系统错误日志时表现较好的聚类算法，针对系统中多种不同类型的报警，未来也可以配合其他聚类算法（如K-Means）共同对报警进行处理，优化聚合效果。
3. 自适应报警阈值。除了对报警聚类，我们还可以通过对监控指标的时序分析，动态管理报警阈值，提高告警的质量和及时性，减少误报和漏告数量。

参考资料

1. Julisch, Klaus. “Clustering intrusion detection alarms to support root cause analysis.” ACM transactions on information and system security (TISSEC) 6.4 (2003): 443-471.
2. https://en.wikipedia.org/wiki/Cluster_analysis

作者简介

刘场，美团点评后端工程师。2017 年加入美团点评，负责美团点评境内度假的业务开发。
干钊，美团点评后端工程师。2017 年加入美团点评，负责美团点评境内度假的业务开发。

全链路压测自动化实践

欧龙

背景与意义

境内度假是一个低频、与节假日典型相关的业务，流量在节假日较平日会上涨五到十几倍，会给生产系统带来非常大的风险。因此，在 2018 年春节前，我们把整个境内度假业务接入了全链路压测，来系统性地评估容量和发现隐患，最终确保了春节期间系统的稳定。

在整个过程中，我们意识到，全链路压测在整个系统稳定性建设中占有核心重要的位置，也是最有效的方案。结合实际业务节假日的频率（基本平均一个月一次），如果能够把它作为稳定性保障的常规手段，我们的系统质量也能够得到很好的保障。同时，为了解决周期常态化压测过程中人力成本高、多个团队重复工作、压测安全不可控，风险高等痛点，我们提出了全链路压测自动化的设想。

通过对压测实施的具体动作做统一的梳理，在压测各个阶段推进标准化和自动化，尽力提升全流程的执行效率，最终达到常态化的目标，如下图 1 所示：

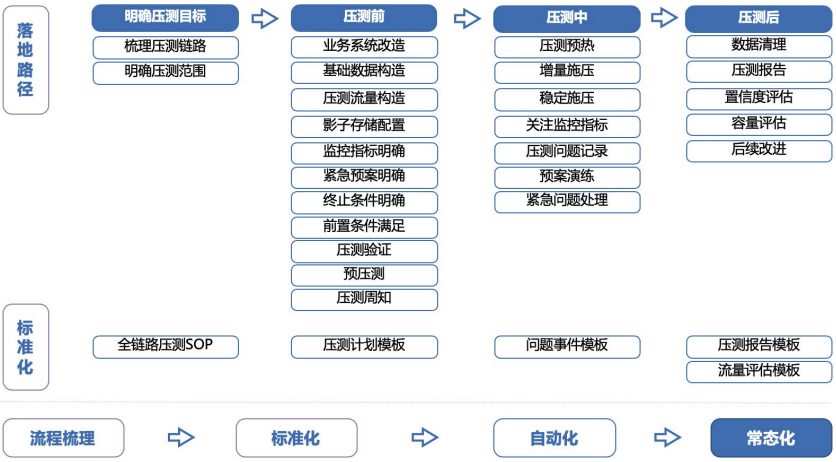


图 1　自动化落地整体思路

另外，在全链路压测的整个周期中，压测安全和压测有效性也是需要一直关注的质量属性。基于这些思考，如下图 2 所示，我们把压测自动化需要解决的关键问题进行了归类和分解：

- 基础流程如何自动化，提高人效；
- 如何自动做好压测验证，保障压测安全；
- 压测置信度量化如何计算，保证压测有效。



图 2　问题分析

最终，基于美团基础的压测平台（Quake 在整个系统，主要提供流量录制、回放、施压的功能），设计并实现了全链路自动化压测系统，为不同业务实施全链路压测提效，并确保压测安全。该系统：

- 提供链路梳理工具，能够自动构建压测入口链路完整的依赖信息，辅助链路梳理；
- 支持链路标注和配置功能，对于无需压测触达的依赖接口，可以通过配置化手段，完成相关接口的 Mock 配置，不用在业务代码中嵌入压测判断逻辑；
- 提供抽象的数据构造接口，通过平台，用户可以配置任意的数据构造逻辑和流程；
- 在压测前 / 压测中，自动对压测服务和流量做多项校验，保障压测安全性；
- 在平日，基于压测计划提供周期性小流量的压测校验，使得业务迭代变更带来

的压测安全风险被尽早发现；

- 提供压测计划管理功能，通过系统自动调度和控制施压过程，解放人力；同时强制前置预压测，也提高了安全性；
- 一键压测，自动生成报告，收集链路入口和告警信息，提供问题记录和跟进功能。

系统设计

系统总体设计



图 3　系统总体逻辑架构

系统的总体逻辑架构，如图 3 所示，主要包括链路构建 / 比对、事件 / 指标收集、链路治理、压测配置管理、压测验证检查、数据构造、压测计划管理、报告输出等功能模块。通过这些模块，为全链路压测的整个流程提供支持，尽力降低业务部门使用全链路压测的门槛和成本。

链路构建 / 比对：负责服务接口方法调用链路的构建、更新、存储。

链路治理：基于构建的链路关系，提供链路中核心依赖、出口 Mock 接口等标注、上下游分析、展示，以及出口 Mock 的配置等功能。

压测配置管理：自动发现注册服务的 Mafka/Cellar/Squirrel/Zebra 的压测配置，

辅助压测方核查和配置相关配置项。

压测验证检查：确保系统可压测，通过多种校验手段和机制设计，来保证压测的安全性。

数据构造：为不同业务压测实施准备基础和流量数据。

压测计划管理：设定压测执行计划，并依赖“压测控制”模块，自动调度整个压测执行过程。

故障诊断：依据收集的关键业务 / 服务指标、报警等信息，判断分析服务是否异常，以及是否终止压测。

置信度评估：从数据覆盖、链路覆盖、技术指标等维度评估压测结果的置信度，即与真实流量情况下各评估维度的相似性。

非功能性需求说明：

- 可扩展性
  - 能够兼容不同业务线数据构造逻辑的差异性。
  - 能够支持不同的流量录制方式。
- 安全性
  - 集成 SSO，按用户所属团队分组，展示所属的压测服务信息。对关键操作留存操作日志。
  - 压测验证检查，是确保压测安全的关键。支持周期性压测验证，能发现待压测服务可压测性随时间的退化。
- 可重用性
  - 长远看，链路构建、事件 / 指标收集 / 故障诊断等模块，在稳定性领域是可重用的基础设施，按独立通用模块建设。

约束说明：

- 基于 Quake 搭建，流量的录制、回放、施压等依赖 Quake。

以下对部分关键模块设计做详细介绍。

链路治理模块设计

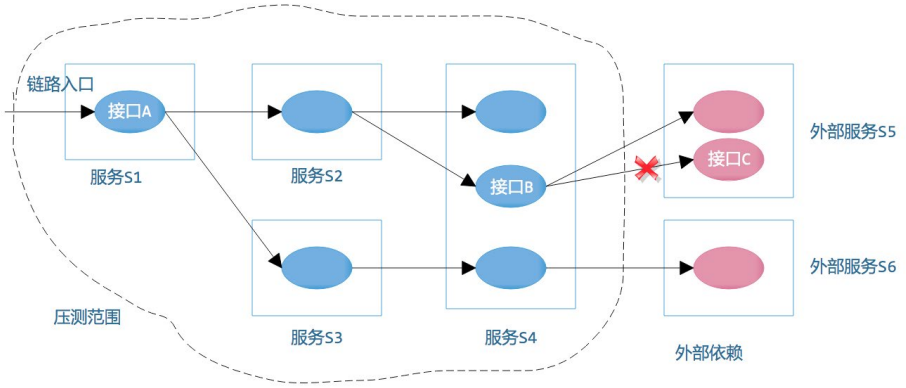


图 4　链路治理示意图

链路治理模块是基于链路构建模块实现的。链路构建模块，底层是以闭包表的方式存储两个维度（服务和接口）的链路关系的，会周期自动地构建或更新。

链路治理模块主要提供链路入口选取、链路标注、服务出口分析、出口 Mock 配置等功能。如图 4 所示，注册压测的服务构成了压测服务的范围，也就确定了各个链路的边界。通过系统自动构建的树结构方式的链路关系，可以辅助压测方对整个链路的梳理，它解决了以往链路梳理靠翻代码等低效手段，缺少全链路视角无法做到完备梳理等问题。



图 5　出口 Mock 配置化

同时，针对整个压测范围，依赖接口可以做人工标注。哪些需要 Mock，哪些不需要 Mock，如此压测特有的链路信息能够得到持续的维护。

对于需要 Mock 的外部接口（如图 4 中的接口 C），待压测系统通过引入专有 SDK 的方式，获得出口配置化 Mock 的能力。如图 5 所示，这里使用了美团酒旅 Mock 平台的基础能力，采用 JVM-Sandbox 作为 AOP 工具，对配置的需要 Mock 的外部接口做动态能力增强。在接口调用时，判断是否是压测流量，是的话走 Mock 逻辑，做模拟时延处理，返回提前配置的响应数据。这样的话，第一，简化了出口 Mock 的操作，业务代码里 Mock 逻辑 0 侵入；第二，把之前本地 Mock 与借助 Mockserver 的两种解决方案用一种方案替代，便于统一管理；第三，在实际压测时，平台还可以通过 SDK 收集 Mock 逻辑执行的数据，自动与后台标注的 Mock 数据对比，来确保应该被 Mock 的出口确实被 Mock 掉。

数据构造模块设计

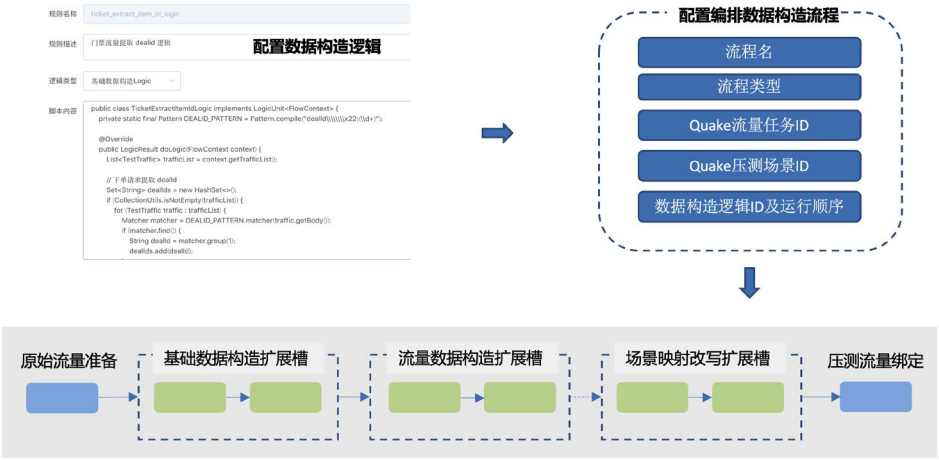


图 6　数据构造

数据构造模块是为了解决不同业务对于基础数据和流量数据的差异化构造流程。提出了两个关键的概念：数据构造逻辑和数据构造流程。数据构造逻辑，是数据构造的细粒度可复用的基本单元，由一段 Java 代码表示。平台提供统一抽象的数据构造接口，基于 Java 动态编译技术，开发了一个 Java 版的脚本引擎，支持构造逻辑的

在线编辑与更新。同时，基于美团 RPC 中间件泛化调用能力，构建了泛化调用工具，帮助用户把外部基础数据构造接口的调用集成到一个数据构造逻辑中。

数据构造流程，定义了压测基础数据和流量数据生成的整个流程。通过与 Quake 的交互，获取原始真实的线上数据；构建了一个简版的流程引擎，在统一设定的流程中，如图 6 所示，通过在标准扩展槽中，配置不同类型的数据构造逻辑和执行顺序，来定义整个数据构造执行的流程；最后，把构造的流量数据与 Quake 压测场景绑定，作为后续 Quake 压测施压中，场景回放流量的来源。

通过这样的设计，能够支持任意数据构造逻辑，通用灵活。同时集成了 Quake 已有的流量录制功能，一键执行数据构造流程，大大地提升了效率。

压测验证模块设计

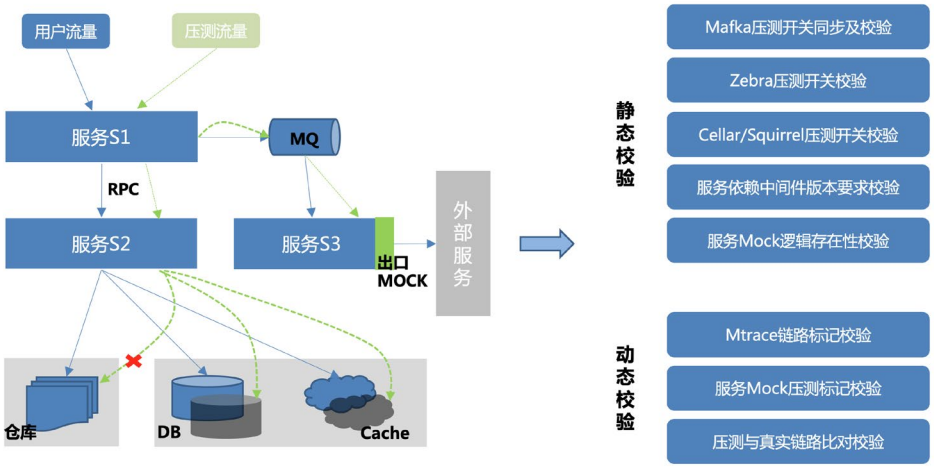


图 7　美团服务压测验证示意

对于压测安全性的保障，一直是自动化的难点。之前的经验多是在非生产环境压测或预压测过程中，依靠不同服务相关负责人的人工确认。这里针对压测验证，提供两条新的思考角度：一个是从待压测服务系统可压测性的角度看；一个是从压测流量特征的角度看。对于第一个角度，一个服务支持压测需要满足压测数据和流量的隔离。对于不同的系统生态，需要满足的点是不同的，对于美团生态下的服务，可压测的条件包括组件版本支持压测、影子存储配置符合预期等等。

从这些条件出发，就可以得到下面这些静态的校验项：

- 服务依赖中间件版本要求校验；
- Zebra 压测配置校验；
- Cellar/Squirrel 压测配置校验；
- Mafka 压测开关同步及校验；
- 服务 Mock 逻辑存在性校验。

而从第二个角度来看，就是关注压测流量下会产生哪些特有的流量特征数据，通过这些特有的数据来确保压测的安全性。这里主要有三类数据：美团分布式追踪系统（MTrace）中调用链路的压测标记数据（正常的压测链路应该是一直带有压测标记，直到压测范围的边界节点，可参考图 4）；标记 Mock 的外部接口被调用时，上报的运行数据；基于监控系统得到的压测流量特有的监控数据。利用这些数据，我们设计了三种动态的校验项，发现压测标记丢失、Mock 出口被调用等异常情况：

- MTrace 链路标记校验，从压测链路入口出发，收集压测链路信息，校验压测标记信息传递是否符合预期。

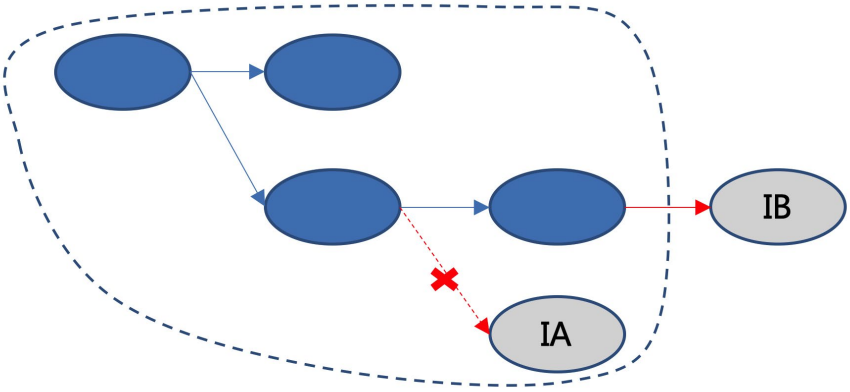


图 8　MTrace 链路标记校验示意

- 服务 Mock 逻辑压测标记校验，通过增强的校验逻辑，把执行信息上报到平

台，与 Mock 配置时的标注数据对比验证。

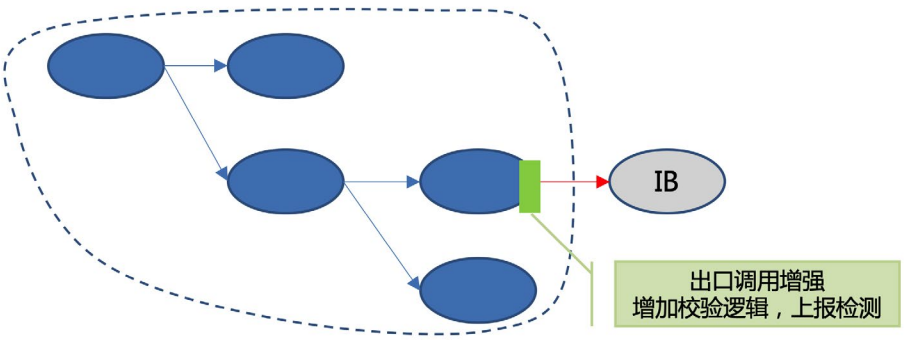


图 9　服务 Mock 压测校验示意

- 压测与真实链路比对校验，利用链路治理模块构建链路的能力，采集压测监控数据重构链路，与真实链路对比验证。

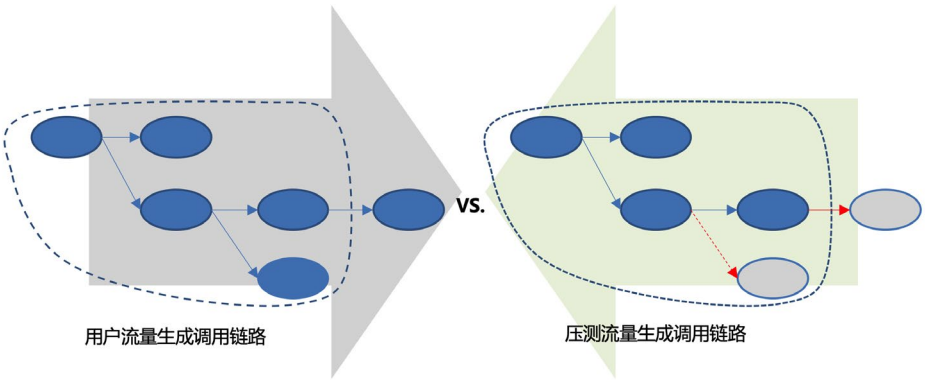


图 10　压测与真实链路对比示意

除了明确静态和动态两类压测校验规则，在具体流程安排上，在压测时和平日两个时期执行这些规则。既能把压测校验的压力分散到平时，也能尽快地发现服务因代码迭代引入的新风险。

在压测时，通过强制前置预压测的流程设计以及静态 / 动态压测校验项的自动执行，保障安全这个事情。校验不通过，给出告警，甚至在允许的情况下直接终止设定

的压测计划。

在平日，通过执行周期性小流量压测校验，在施压过程中对 QPS 做个位数的精细控制，以尽量小的代价快速发现压测范围内压测安全性的退化。

压测计划管理模块设计

压测计划管理模块，提供压测计划的提前设定，然后模块能够自动调度和控制整个施压过程。如图 11 所示，这里的压测计划是多个压测场景的组合，包含 QPS 的增长计划等信息，主要分为预压测和正式压测两个阶段。压测计划的自动实施，能够解决尤其多场景组合压测，操作耗时多、多场景压测 QPS 无法同步变更、压测方无法兼顾操作和观测等问题，提升了效率。同时，在压测计划执行状态机里，预压测正常执行完成，状态才能迁移到正式压测的开始状态，提高了压测安全性。

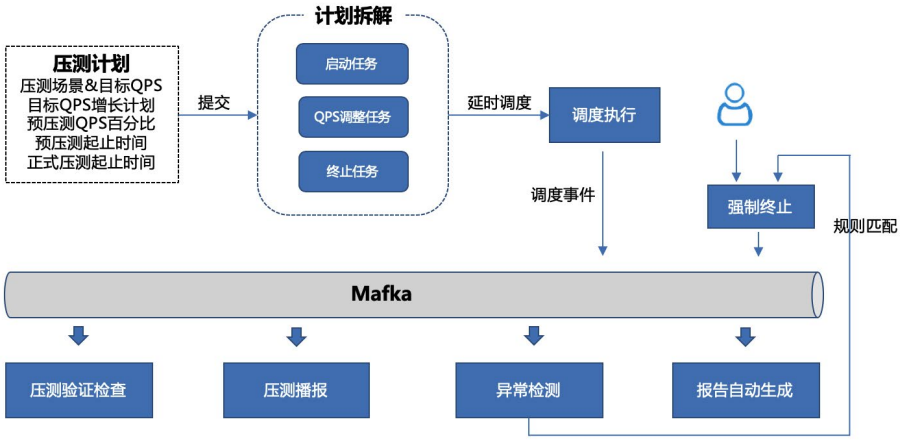


图 11　压测计划执行

从图 11 可以看到，压测计划模块，是整个自动化压测的核心，协同起了各个模块。通过具体的计划任务执行产生的事件，触发了压测验证检查、压测进展播报、收集压测监控 / 告警等数据，来检测服务是否异常，并根据配置来终止压测，能够故障时及时止损。最后，报告生成模块收到压测终止事件，汇总各种信息，自动生成包括压测基本信息等多维度信息的压测报告，节省了一些压测后分析的时间。

案例分享

以下以实际压测的过程来做个案例分享。

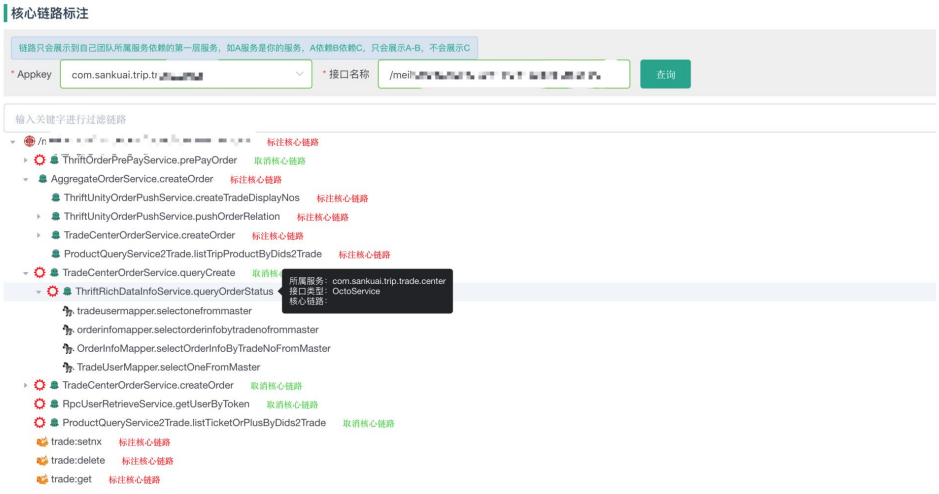
团队 / 服务注册

- 设定实施压测的虚拟团队和压测覆盖范围的应用服务。



链路治理

- 选定压测链路入口，可以得到入口以下的接口链路关系树，便于梳理。
- 明确需要 Mock 的外部接口，并做配置，参考“链路治理模块设计”一节。



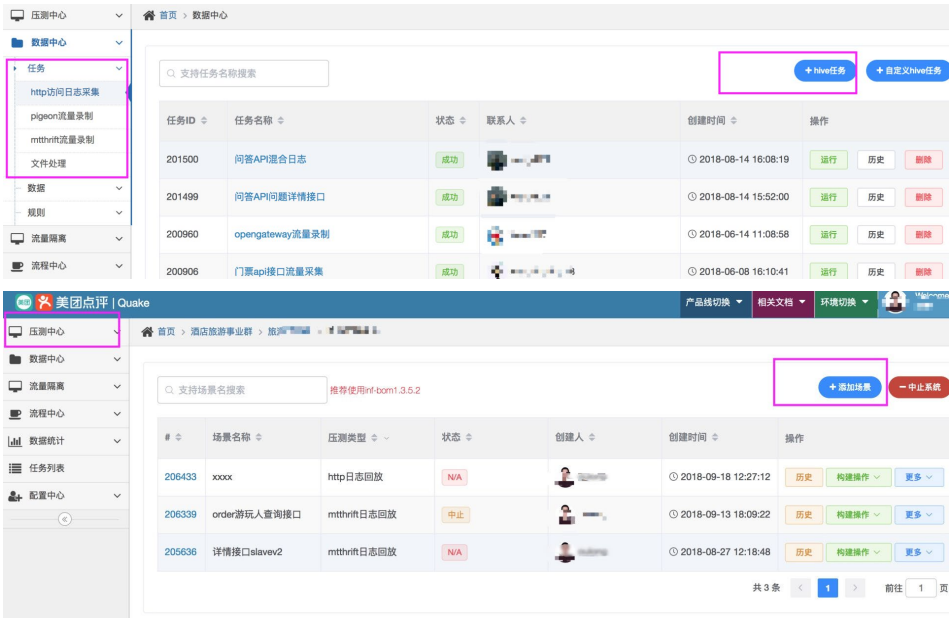
应用改造与压测配置

- 对待接入压测应用改造，满足“服务的可压测条件”，参考图 7。
- 压测应用依赖中间件配置，系统依据构建的链路信息，能够自动发现。提供统一配置和核对的页面功能。



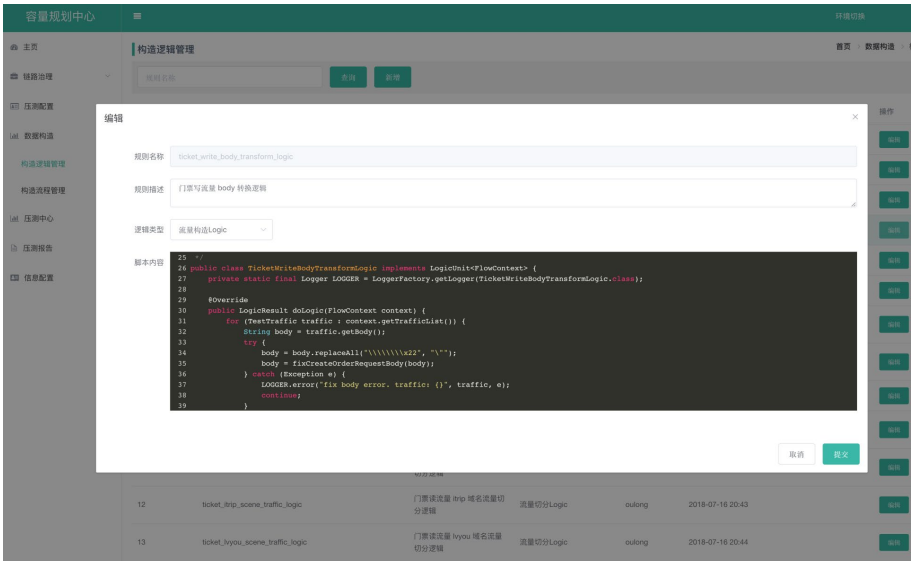
Quake 准备

- 压测自动化系统是基于 Quake 构建的，流量录制、回放、施压等依赖于此。因此需要到 Quake 上配置流量录制的“流量任务”和压测执行的“压测场景”。

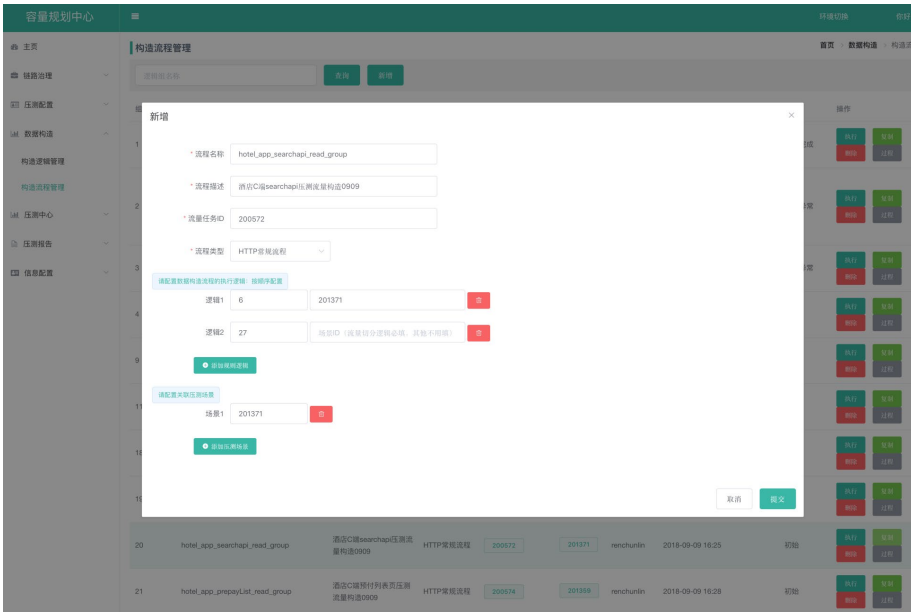


数据构造

- 配置数据构造逻辑，当然已有的逻辑都是可复用的单元，可以先查看已有逻辑是否能满足自己的需要。

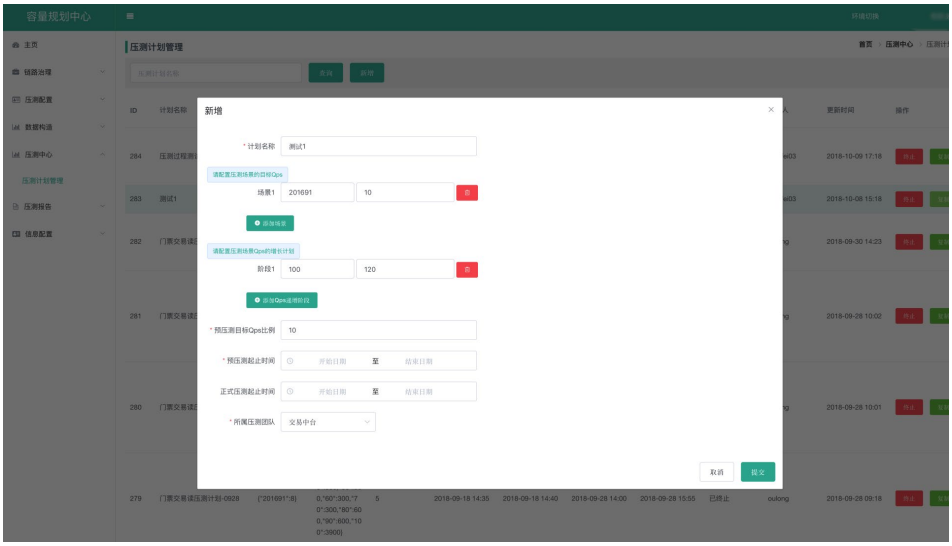


- 配置数据构造流程。



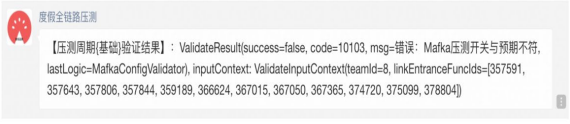
压测实施

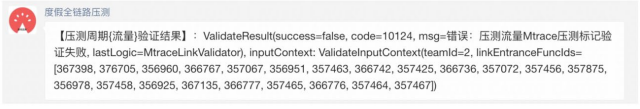
- 设定压测计划，到启动时间，系统会自动启动压测。



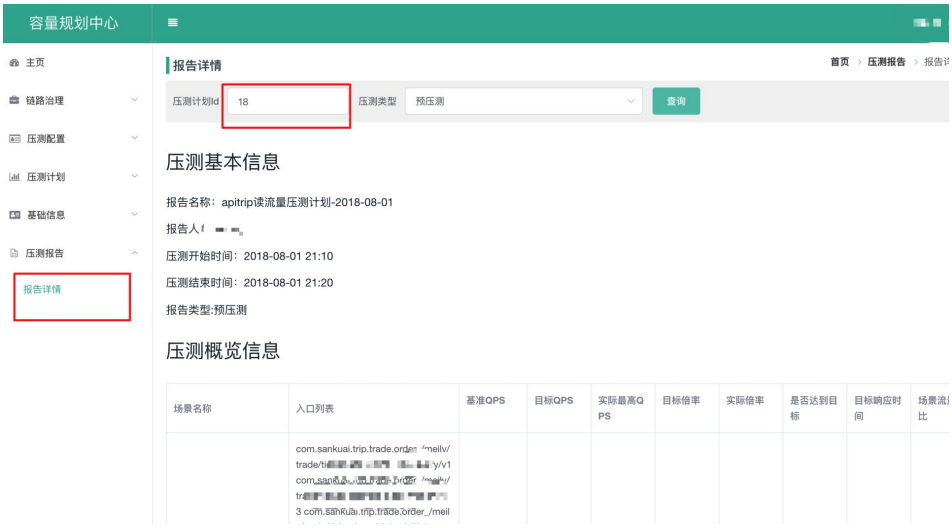
- 压测中，注意关注压测验证校验的告警信息，及时处理。







- 压测后，可查看压测报告。记录和跟进发现的问题。



总结与展望

目前，压测自动化系统已经投入使用，美团酒店和境内度假的全部团队已经接入，有效地提升了压测效率。后续会在两个大方向上持续建设升级，一个是把全链路压测放到“容量评估与优化”领域来看，不仅关注整体系统的稳定性，同时也期望兼顾成本的平衡；另一个是与稳定性其他子领域的生态集成，比如故障演练、弹性伸缩等等，在更多场景发挥压测的作用。最后，通过这些努力，使得线上系统的稳定性成为一个确定性的事情。

参考资料

[1] 全链路压测平台（Quake）在美团中的实践
[2] 阿里 JVM-Sandbox
[3] Dubbo 的泛化调用
[4] Java 的动态编译

作者简介

欧龙，美团研发工程师，2013 年加入美团，目前主要负责境内度假交易稳定性建设等工作。

降低软件复杂性一般原则和方法

邹政华

一、前言

斯坦福教授、Tcl 语言发明者 John Ousterhout 的著作《A Philosophy of Software Design》[1]，自出版以来，好评如潮。按照 IT 图书出版的惯例，如果冠名为“实践”，书中内容关注的是某项技术的细节和技巧；冠名为“艺术”，内容可能是记录一件优秀作品的设计过程和经验；而冠名为“哲学”，则是一些通用的原则和方法论，这些原则方法论串起来，能够形成一个体系。正如”知行合一”、“世界是由原子构成的”、“我思故我在”，这些耳熟能详的句子能够一定程度上代表背后的人物和思想。用一句话概括《A Philosophy of Software Design》，软件设计的核心在于降低复杂性。

本篇文章是围绕着“降低复杂性”这个主题展开的，很多重要的结论来源于 John Ousterhout，笔者觉得很有共鸣，就做了一些相关话题的延伸、补充了一些实例。虽说是”一般原则“，也不意味着是绝对的真理，整理出来，只是为了引发大家对软件设计的思考。

二、如何定义复杂性

关于复杂性，尚无统一的定义，从不同的角度可以给出不同的答案。可以用数量来度量，比如芯片集成的电子器件越多越复杂 (不一定对)；按层次性 [2] 度量，复杂度在于层次的递归性和不可分解性。在信息论中，使用熵来度量信息的不确定性。

John Ousterhout 选择从认知的负担和开发工作量的角度来定义软件的复杂性，并且给出了一个复杂度量公式：

$$C = \sum_{p} c_p t_p$$

子模块的复杂度 cp 乘以该模块对应的开发时间权重值 tp，累加后得到系统的整体复杂度 C。系统整体的复杂度并不简单等于所有子模块复杂度的累加，还要考虑该模块的开发维护所花费的时间在整体中的占比（对应权重值 tp）。也就是说，即使某个模块非常复杂，如果很少使用或修改，也不会对系统的整体复杂度造成大的影响。

子模块的复杂度 cp 是一个经验值，它关注几个现象：

- 修改扩散，修改时有连锁反应。
- 认知负担，开发人员需要多长时间来理解功能模块。
- 不可知（Unknown Unknowns），开发人员在接到任务时，不知道从哪里入手。

造成复杂的原因一般是代码依赖和晦涩 (Obscurity)。其中，依赖是指某部分代码不能被独立地修改和理解，必定会牵涉到其他代码。代码晦涩，是指从代码中难以找到重要信息。

三、解决复杂性的一般原则

首先，互联网行业的软件系统，很难一开始就做出完美的设计，通过一个个功能模块衍生迭代，系统才会逐步成型；对于现存的系统，也很难通过一个大动作，一劳永逸地解决所有问题。系统设计是需要持续投入的工作，通过细节的积累，最终得到一个完善的系统。因此，好的设计是日拱一卒的结果，在日常工作中要重视设计和细节的改进。

其次，专业化分工和代码复用促成了软件生产率的提升。比如硬件工程师、软件工程师（底层、应用、不同编程语言）可以在无需了解对方技术背景的情况下进行合作开发；同一领域服务可以支撑不同的上层应用逻辑等等。其背后的思想，无非是通过将系统分成若干个水平层、明确每一层的角色和分工，来降低单个层次的复杂性。同时，每个层次只要给相邻层提供一致的接口，可以用不同的方法实现，这就为软件重用提供了支持。分层是解决复杂性问题的重要原则。

第三，与分层类似，分模块是从垂直方向来分解系统。分模块最常见的应用场景，是如今广泛流行的微服务。分模块降低了单模块的复杂性，但是也会引入新的复

杂性，例如模块与模块的交互，后面的章节会讨论这个问题。这里，我们将第三个原则确定为分模块。

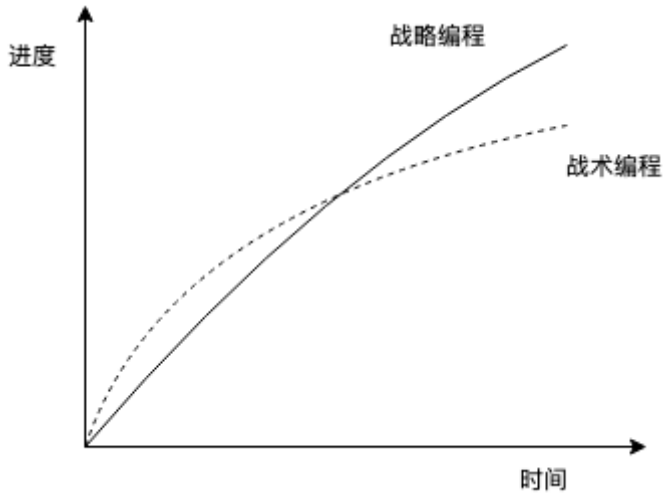
最后，代码能够描述程序的工作流程和结果，却很难描述开发人员的思路，而注释和文档可以。此外，通过注释和文档，开发人员在不阅读实现代码的情况下，就可以理解程序的功能，注释间接促成了代码抽象。好的注释能够帮助解决软件复杂性问题，尤其是认知负担和不可知问题（Unknown Unknowns）。

四、解决复杂性之日拱一卒

4.1 拒绝战术编程

战术编程致力于完成任务，新增加特性或者修改 Bug 时，能解决问题就好。这种工作方式，会逐渐增加系统的复杂性。如果系统复杂到难以维护时，再去重构会花费大量的时间，很可能会影响新功能的迭代。

战略编程，是指重视设计并愿意投入时间，短时间内可能会降低工作效率，但是长期看，会增加系统的可维护性和迭代效率。



设计系统时，很难在开始阶段就面面俱到。好的设计应该体现在一个个小的模块上，修改 bug 时，也应该抱着设计新系统的心态，完工后让人感觉不到“修补”的痕迹。经过累积，最终形成一个完善的系统。从长期看，对于中大型的系统，将日常开

发时间的 10-15% 用于设计是值得的。有一种观点认为，创业公司需要追求业务迭代速度和节省成本，可以容忍糟糕的设计，这是用错误的方法去追求正确的目标。降低开发成本最有效的方式是雇佣优秀的工程师，而不是在设计上做妥协。

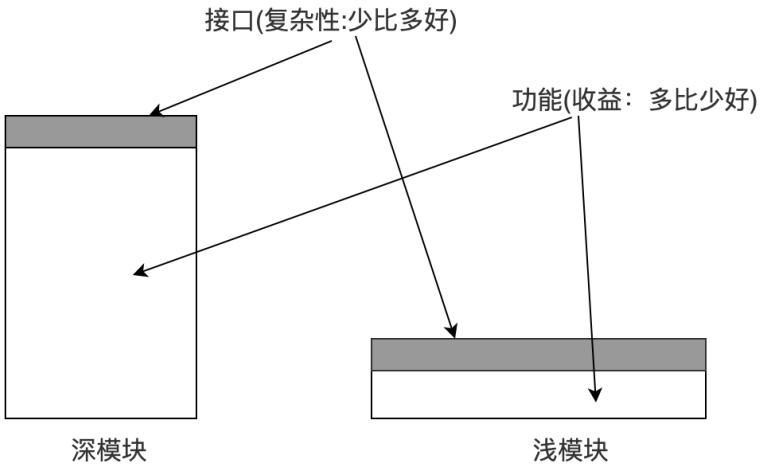
4.2 设计两次

为一个类、模块或者系统的设计提供两套或更多方案，有利于我们找到最佳设计。以我们日常的技术方案设计为例，技术方案本质上需要回答两个问题，其一，为什么该方案可行？ 其二，在已有资源限制下，为什么该方案是最优的？为了回答第一个问题，我们需要在技术方案里补充架构图、接口设计和时间人力估算。而要回答第二个问题，需要我们在关键点或争议处提供二到三种方案，并给出建议方案，这样才有说服力。通常情况下，我们会花费很多的时间准备第一个问题，而忽略第二个问题。其实，回答好第二个问题很重要，大型软件的设计已经复杂到没人能够一次就想到最佳方案，一个仅仅“可行”的方案，可能会给系统增加额外的复杂性。对聪明人来说，接受这点更困难，因为他们习惯于“一次搞定问题”。但是聪明人迟早也会碰到自己的瓶颈，在低水平问题上徘徊，不如花费更多时间思考，去解决真正有挑战性的问题。

五、解决复杂性之分层

5.1 层次和抽象

软件系统由不同的层次组成，层次之间通过接口来交互。在严格分层的系统里，内部的层只对相邻的层次可见，这样就可以将一个复杂问题分解成增量步骤序列。由于每一层最多影响两层，也给维护带来了很大的便利。分层系统最有名的实例是 TCP/IP 网络模型。



在分层系统里，每一层应该具有不同的抽象。TCP/IP 模型中，应用层的抽象是用户接口和交互；传输层的抽象是端口和应用之间的数据传输；网络层的抽象是基于 IP 的寻址和数据传输；链路层的抽象是适配和虚拟硬件设备。如果不同的层具有相同的抽象，可能存在层次边界不清晰的问题。

5.2 复杂性下沉

不应该让用户直面系统的复杂性，即便有额外的工作量，开发人员也应当尽量让用户使用更简单。如果一定要在某个层次处理复杂性，这个层次越低越好。举个例子，Thrift 接口调用时，数据传输失败需要引入自动重试机制，重试的策略显然在 Thrift 内部封装更合适，开放给用户（下游开发人员）会增加额外的使用负担。与之类似的是系统里随处可见的配置参数（通常写在 XML 文件里），在编程中应当尽量避免这种情况，用户（下游开发人员）一般很难决定哪个参数是最优的，如果一定要开放参数配置，最好给定一个默认值。

复杂性下沉，并不是说把所有功能下移到一个层次，过犹不及。如果复杂性跟下层的功能相关，或者下移后，能大大下降其他层次或整体的复杂性，则下移。

5.3 异常处理

异常和错误处理是造成软件复杂的罪魁祸首之一。有些开发人员错误的认为处理和上报的错误越多越好，这会导致过度防御性的编程。如果开发人员捕获了异常并不知道如何处理，直接往上层扔，这就违背了封装原则。

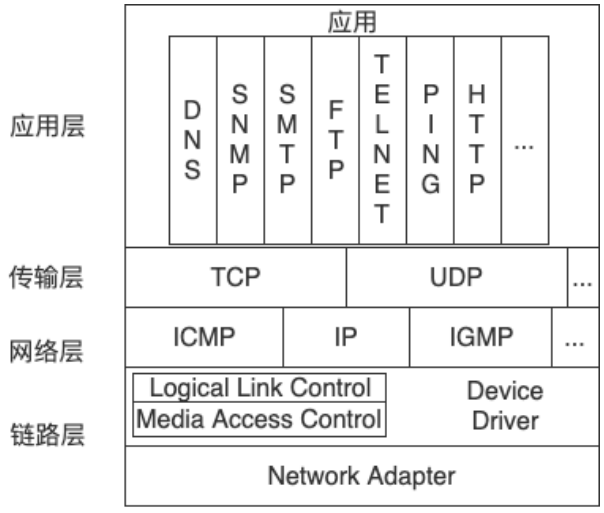
降低复杂度的一个原则就是尽可能减少需要处理异常的可能性。而最佳实践就是确保错误终结，例如删除一个并不存在的文件，与其上报文件不存在的异常，不如什么都不做。确保文件不存在就好了，上层逻辑不但不会被影响，还会因为不需要处理额外的异常而变得简单。

六、解决复杂性之分模块

分模块是解决复杂性的重要方法。理想情况下，模块之间应该是相互隔离的，开发人员面对具体的任务，只需要接触和了解整个系统的一小部分，而无需了解或改动其他模块。

6.1 深模块和浅模块

深模块 (Deep Module) 指的是拥有强大功能和简单接口的模块。深模块是抽象的最佳实践，通过排除模块内部不重要的信息，让用户更容易理解和使用。



TCP/IP网络模型

Unix 操作系统文件 I/O 是典型的深模块，以 Open 函数为例，接口接受文件名为参数，返回文件描述符。但是这个接口的背后，是几百行的实现代码，用来处理文件存储、权限控制、并发控制、存储介质等等，这些对用户是不可见的。

```
int open(const char* path, int flags, mode_t permissions);
```

与深模块相对的是浅模块 (Shallow Module)，功能简单，接口复杂。通常情况下，浅模块无助于解决复杂性。因为他们提供的收益（功能）被学习和使用成本抵消了。以 Java I/O 为例，从 I/O 中读取对象时，需要同时创建三个对象 FileInput-Stream、BufferedInputStream、ObjectInputStream，其中前两个创建后不会被直接使用，这就给开发人员造成了额外的负担。默认情况下，开发人员无需感知到 BufferedInputStream，缓冲功能有助于改善文件 I/O 性能，是个很有用的特性，可以合并到文件 I/O 对象里。假如我们想放弃缓冲功能，文件 I/O 也可以设计成提供对应的定制选项。

```
FileInputStream fileStream = new FileInputStream(fileName);
BufferedInputStream bufferedStream = new BufferedInputStream(fileStream);
ObjectInputStream objectStream = new ObjectInputStream(bufferedStream);
```

关于浅模块有一些争议，大多数情况是因为浅模块是不得不接受的既定事实，而不见得是因为合理性。当然也有例外，比如领域驱动设计里的防腐层，系统在与外部系统对接时，会单独建立一个服务或模块去适配，用来保证原有系统技术栈的统一和稳定性。

6.2 通用和专用

设计新模块时，应该设计成通用模块还是专用模块？一种观点认为通用模块满足多种场景，在未来遇到预期外的需求时，可以节省时间。另外一种观点则认为，未来的需求很难预测，没必要引入用不到的特性，专用模块可以快速满足当前的需求，等有后续需求时再重构成通用的模块也不迟。

以上两种思路都有道理，实际操作的时候可以采用两种方式各自的优点，即在功能实现上满足当前的需求，便于快速实现；接口设计通用化，为未来留下余量。

举个例子。

```
void backspace(Cursor cursor);
void delete(Cursor cursor);
void deleteSelection(Selection selection);

// 以上三个函数可以合并为一个更通用的函数
void delete(Position start, Position end);
```

设计通用性接口需要权衡，既要满足当前的需求，同时在通用性方面不要过度设计。一些可供参考的标准：

- 满足当前需求最简单的接口是什么？在不减少功能的前提下，减少方法的数量，意味着接口的通用性提升了。
- 接口使用的场景有多少？如果接口只有一个特定的场景，可以将多个这样的接口合并成通用接口。
- 满足当前需求情况下，接口的易用性？如果接口很难使用，意味着我们可能过度设计了，需要拆分。

6.3 信息隐藏

信息隐藏是指，程序的设计思路以及内部逻辑应当包含在模块内部，对其他模块不可见。如果一个模块隐藏了很多信息，说明这个模块在提供很多功能的同时又简化了接口，符合前面提到的深模块理念。软件设计领域有个技巧，定义一个”大”类有助于实现信息隐藏。这里的“大”类指的是，如果要实现某功能，将该功能相关的信息都封装进一个类里面。

信息隐藏在降低复杂性方面主要有两个作用：一是简化模块接口，将模块功能以更简单、更抽象的方式表现出来，降低开发人员的认知负担；二是减少模块间的依赖，使得系统迭代更轻量。举个例子，如何从 B+ 树中存取信息是一些数据库索引的核心功能，但是数据库开发人员将这些信息隐藏了起来，同时提供简单的对外交互接口，也就是 SQL 脚本，使得产品和运营同学也能很快地上手。并且，因为有足够的抽象，数据库可以在保持外部兼容的情况下，将索引切换到散列或其他数据结构。

与信息隐藏相对的是信息暴露，表现为：设计决策体现在多个模块，造成不同模块间的依赖。举个例子，两个类能处理同类型的文件。这种情况下，可以合并这两个类，或者提炼出一个新类（参考《重构》[3] 一书）。工程师应当尽量减少外部模块需要的信息量。

6.4 拆分和合并

两个功能，应该放在一起还是分开？“不管黑猫白猫”，能降低复杂性就好。这里有一些可以借鉴的设计思路：

- 共享信息的模块应当合并，比如两个模块都依赖某个配置项。
- 可以简化接口时合并，这样可以避免客户同时调用多个模块来完成某个功能。
- 可以消除重复时合并，比如抽离重复的代码到一个单独的方法中。
- 通用代码和专用代码分离，如果模块的部分功能可以通用，建议和专用部分分离。举个例子，在实际的系统设计中，我们会将专用模块放在上层，通用模块放在下层以供复用。

七、解决复杂性之注释

注释可以记录开发人员的设计思路和程序功能，降低开发人员的认知负担和解决不可知 (Unkown Unkowns) 问题，让代码更容易维护。通常情况下，在程序的整个生命周期里，编码只占了少部分，大量时间花在了后续的维护上。有经验的工程师懂得这个道理，通常也会产出更高质量的注释和文档。

注释也可以作为系统设计的工具，如果只需要简单的注释就可以描述模块的设计思路和功能，说明这个模块的设计是良好的。另一方面，如果模块很难注释，说明模块没有好的抽象。

7.1 注释的误区

关于注释，很多开发者存在一些认识上的误区，也是造成大家不愿意写注释的原因。比如“好代码是自注释的”、”没有时间“、“现有的注释都没有用，为什么还要浪费时间”等等。这些观点是站不住脚的。“好代码是自注释的”只在某些场景下是

合理的，比如为变量和方法选择合适的名称，可以不用单独注释。但是更多的情况，代码很难体现开发人员的设计思路。此外，如果用户只能通过读代码来理解模块的使用，说明代码里没有抽象。好的注释可以极大地提升系统的可维护性，获取长期的效率，不存在“没有时间”一说。注释也是一种可以习得的技能，一旦习得，就可以在后续的工作中应用，这就解决了“注释没有用”的问题。

7.2 使用注释提升系统可维护性

注释应当能提供代码之外额外的信息，重视 What 和 Why，而不是代码是如何实现的 (How)，最好不要简单地使用代码中出现过的单词。

根据抽象程度，注释可以分为低层注释和高层注释，低层次的注释用来增加精确度，补充完善程序的信息，比如变量的单位、控制条件的边界、值是否允许为空、是否需要释放资源等。高层次注释抛弃细节，只从整体上帮助读者理解代码的功能和结构。这种类型的注释更好维护，如果代码修改不影响整体的功能，注释就无需更新。在实际工作中，需要兼顾细节和抽象。低层注释拆散与对应的实现代码放在一起，高层注释一般用于描述接口。

注释先行，注释应该作为设计过程的一部分，写注释最好的时机是在开发的开始环节，这不仅会产生更好的文档，也会帮助产生好的设计，同时减少写文档带来的痛苦。开发人员推迟写注释的理由通常是：代码还在修改中，提前写注释到时候还得再改一遍。这样的话就会衍生两个问题：

- 首先，推迟注释通常意味着根本就没有注释。一旦决定推迟，很容易引发连锁反应，等到代码稳定后，也不会有注释这回事。这时候再想添加注释，就得专门抽出时间，客观条件可能不会允许这么做。
- 其次，就算我们足够自律抽出专门时间去写注释，注释的质量也不会很好。我们潜意识中觉得代码已经写完了，急于开展下一个项目，只是象征性地添加一些注释，无法准确复现当时的设计思路。

避免重复的注释。如果有重复注释，开发人员很难找到所有的注释去更新。解决方法是，可以找到醒目的地方存放注释文档，然后在代码处注明去查阅对应文档的地

址。如果程序已经在外部文档中注释过了，不要在程序内部再注释了，添加注释的引用就可以了。

注释属于代码，而不是提交记录。一种错误的做法是将功能注释放在提交记录里，而不是放在对应代码文件里。因为开发人员通常不会去代码提交记录里去查看程序的功能描述，很不方便。

7.3 使用注释改善系统设计

良好的设计基础是提供好的抽象，在开始编码前编写注释，可以帮助我们提炼模块的核心要素：模块或对象中最重要的功能和属性。这个过程促进我们去思考，而不是简单地堆砌代码。另一方面，注释也能够帮助我们检查自己的模块设计是否合理，正如前文中提到，深模块提供简单的接口和强大的功能，如果接口注释冗长复杂，通常意味着接口也很复杂；注释简单，意味着接口也很简单。在设计的早期注意和解决这些问题，会为我们带来长期的收益。

八、后记

John Ousterhout 累计写过 25 万行代码，是 3 个操作系统的重要贡献者，这些原则可以视为作者编程经验的总结。有经验的工程师看到这些观点会有共鸣，一些著作如《代码大全》、《领域驱动设计》也会有类似的观点。本文中提到的原则和方法具有一定实操和指导价值，对于很难有定论的问题，也可以在实践中去探索。

关于原则和方法论，既不必刻意拔高，也不要嗤之以鼻。指导实践的不是更多的实践，而是实践后的总结和思考。应用原则和方法论实质是借鉴已有的经验，可以减少我们自行摸索的时间。探索新的方法可以帮助我们适应新的场景，但是新方法本身需要经过时间检验。

九、参考文档

- John Ousterhout. A Philosophy of Software Design. Yaknyam Press, 2018.
- 梅拉尼 · 米歇尔 . 复杂 . 湖南科学技术出版社 , 2016.
- Martin Fowler. Refactoring: Improving the Design of Existing Code (2nd Edition) . Addison-Wesley Signature Series, 2018.

作者简介

政华，顺谱，陶鑫，美团打车调度系统工程团队工程师。

招聘信息

美团打车调度系统工程团队诚招高级工程师 / 技术专家，我们的目标，是与算法、数据团队密切协作，建设高性能、高可用、可配置的打车调度引擎，为用户提供更好的出行体验。欢迎有兴趣的同学发送简历到 tech@meituan.com（邮件标题注明：打车调度系统工程团队）。




虎年大吉