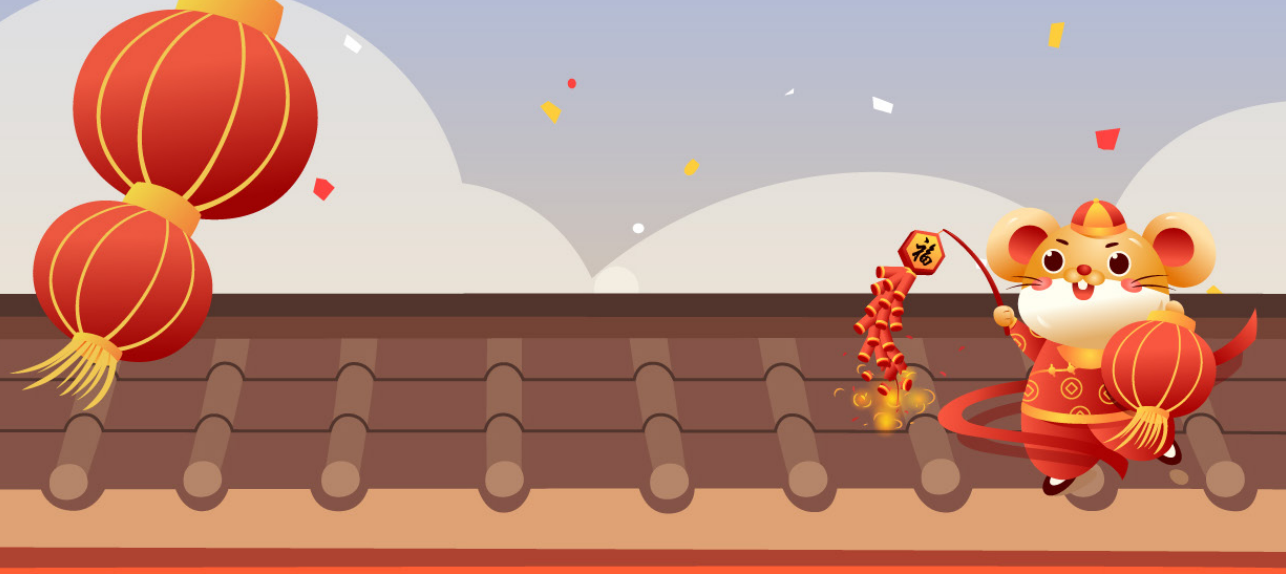


美团点评
2019 技术年货

CODE A BETTER LIFE

【后台篇】

美团点评

微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐

目录

后台

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。

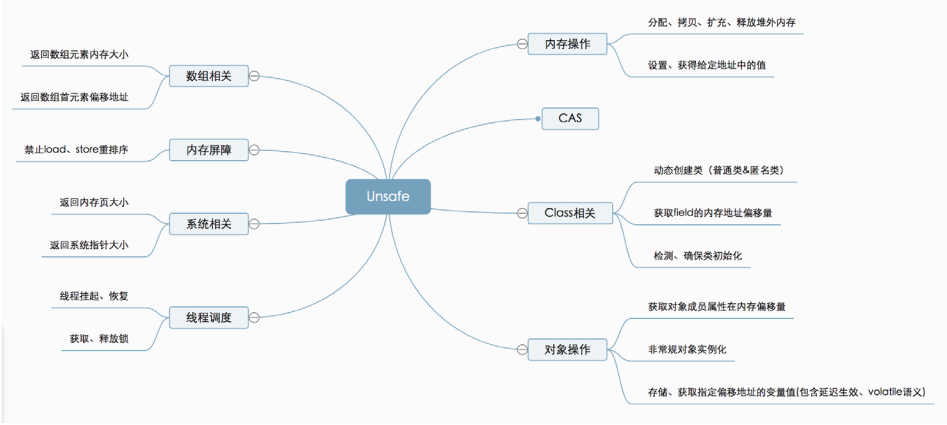
其一，从 getUnsafe 方法的使用限制条件出发，通过 Java 命令行命令 -Xbootclasspath/a 把调用 Unsafe 相关方法的类 A 所在 jar 包路径追加到默认的 bootstrap 路径中，使得 A 被引导类加载器加载，从而通过 Unsafe.getUnsafe 方法安全的获取 Unsafe 实例。

```
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。

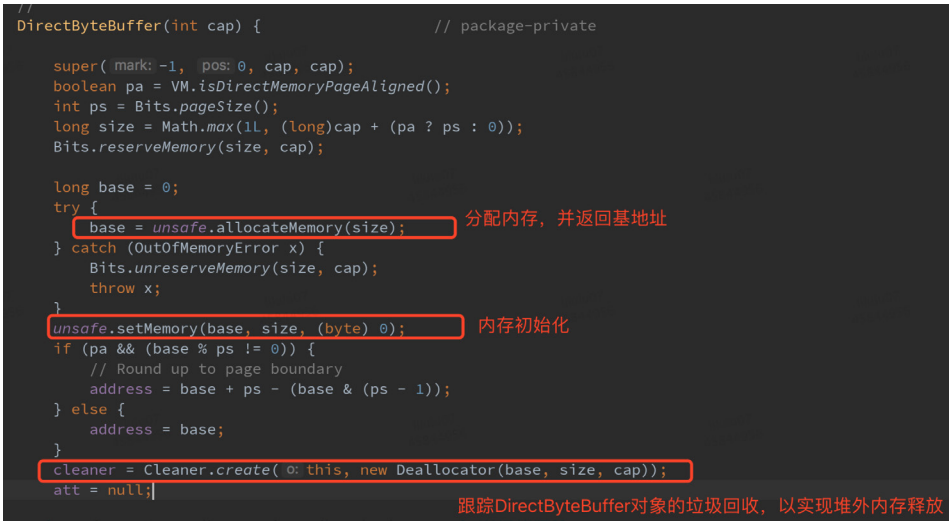
使用堆外内存的原因

- 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

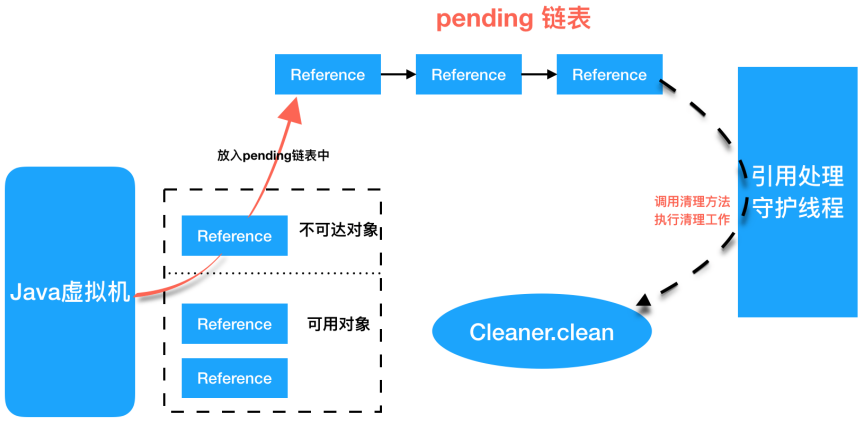
DirectByteBuffer 是 Java 用于实现堆外内存的一个重要类，通常用在通信过程中做缓冲池，如在 Netty、MINA 等 NIO 框架中应用广泛。DirectByteBuffer 对于堆外内存的创建、使用、销毁等逻辑均由 Unsafe 提供的堆外内存 API 来实现。

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。

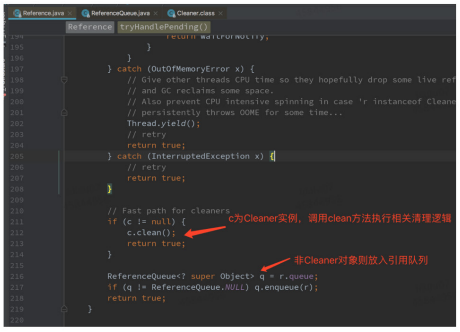


那么如何通过构建垃圾回收追踪对象 Cleaner 实现堆外内存释放呢?

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);
```

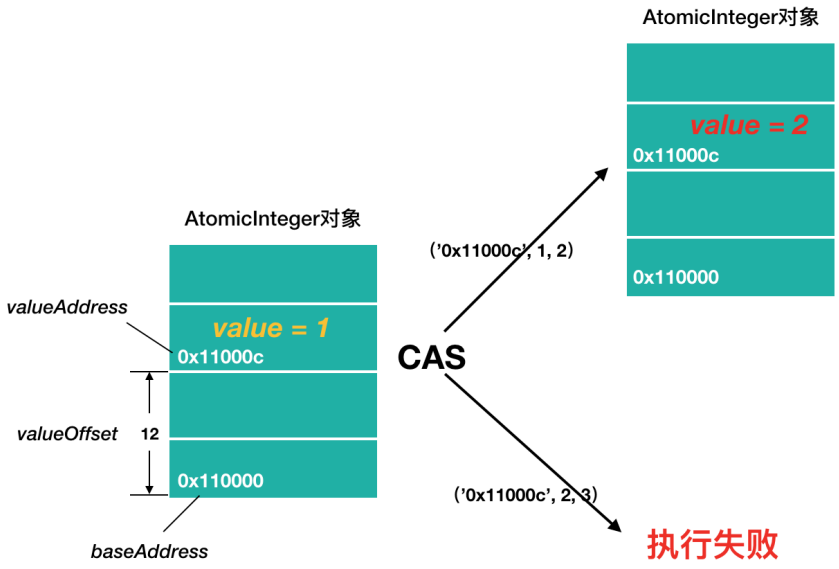
什么是 CAS？即比较并替换，实现并发算法时常用到的一种技术。CAS 操作包含三个操作数——内存位置、预期原值及新值。执行 CAS 操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作。我们都知道，CAS 是一条 CPU 的原子指令（cmpxchg 指令），不会造成所谓的数据不一致问题，Unsafe 提供的 CAS 方法（如 compareAndSwapXXX）底层实现即为 CPU 指令 cmpxchg。

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。

```
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。

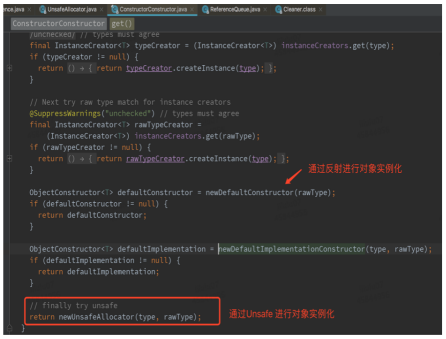
```
// 返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
// 获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，
getChar 等
public native Object getObject(Object o, long offset);
// 给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，
putChar 等
public native void putObject(Object 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

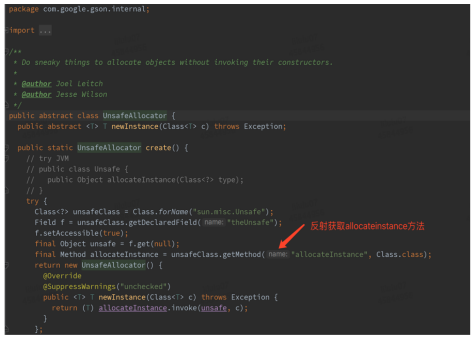
- **常规对象实例化方式**：我们通常所用到的创建对象的方式，从本质上来讲，都是通过 new 机制来实现对象的创建。但是，new 机制有个特点就是当类只提供有参的构造函数且无显示声明无参构造函数时，则必须使用有参构造函数进行对象构造，而使用有参构造函数时，必须传递相应个数的参数才能完成对象实例化。
- **非常规的实例化方式**：而 Unsafe 中提供 allocateInstance 方法，仅通过 Class 对象就可以创建此类的实例对象，而且不需要调用其构造函数、初始化代码、JVM 安全检查等。它抑制修饰符检测，也就是即使构造器是 private 修饰的也能通过此方法实例化，只需提类对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

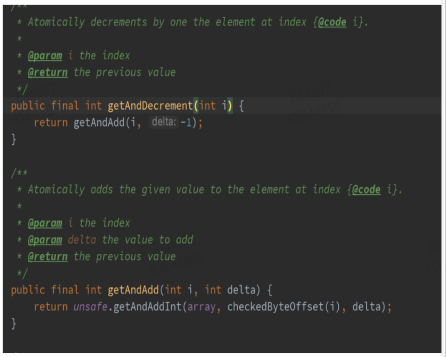
这部分主要介绍与数据操作相关的 arrayBaseOffset 与 arrayIndexScale 这两个方法，两者配合起来使用，即可定位数组中每个元素在内存中的位置。

```
// 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

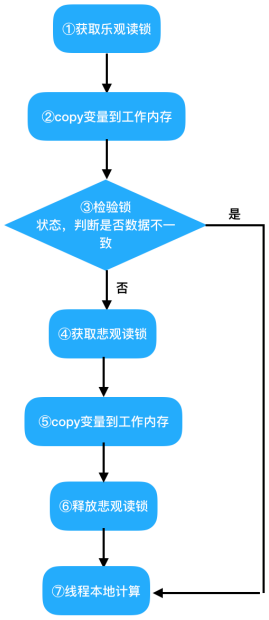
内存屏障

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。




```java
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”

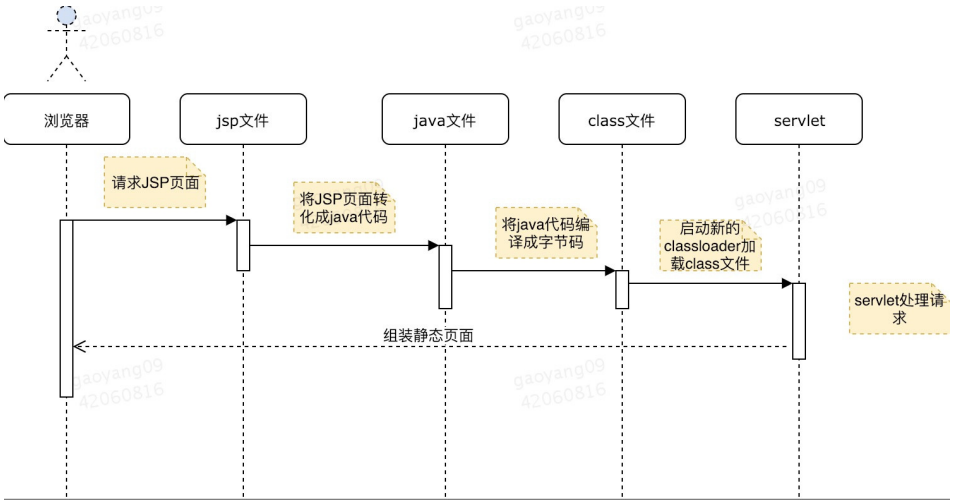
“嗯，好了，表格显示没问题了，但是，登录人的姓名没取到啊，是不是 Sesstion 获取有问题?”

“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。

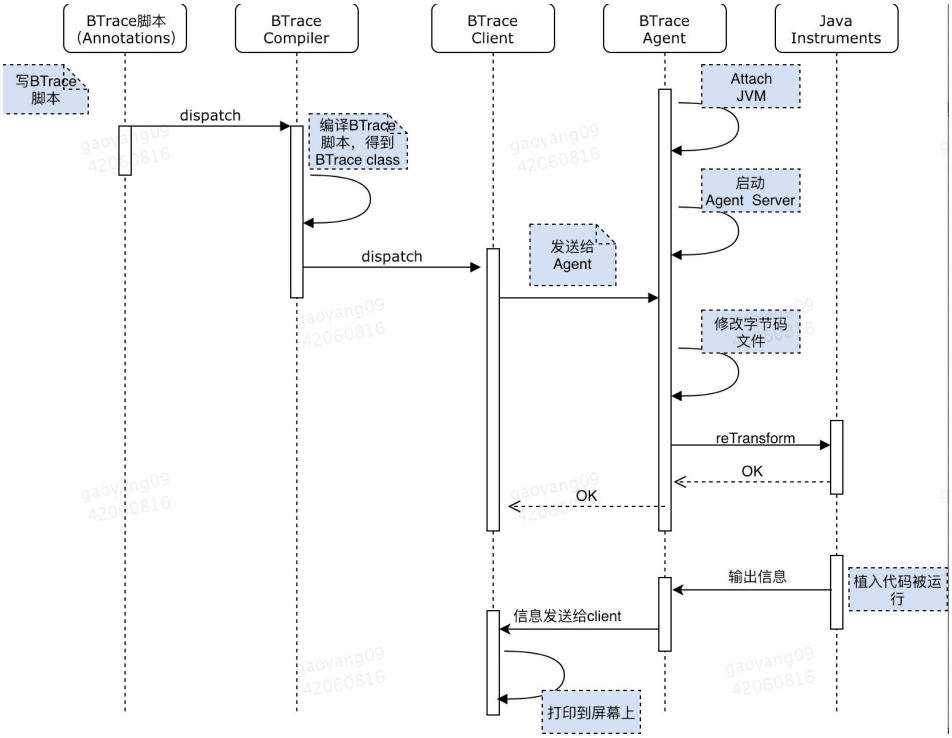
既然 BTrace 能解决上面我们提到的所有问题，那么 BTrace 的架构是怎样的呢？

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。

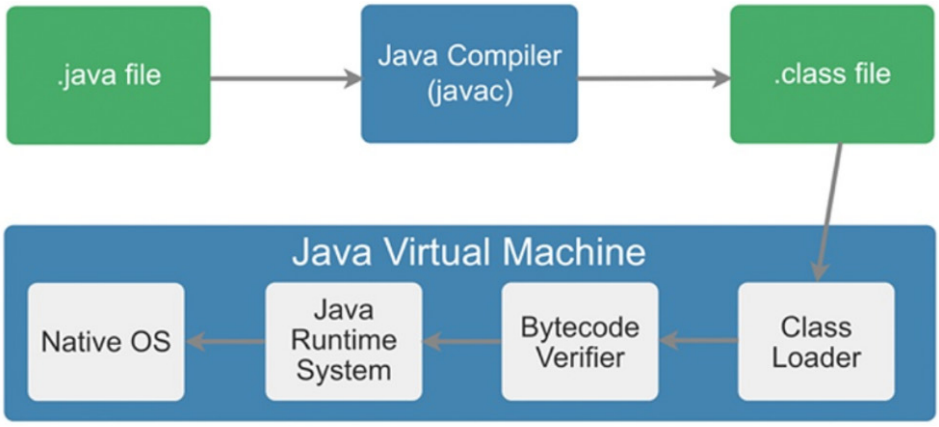


图 1　Java 运行示意图

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：

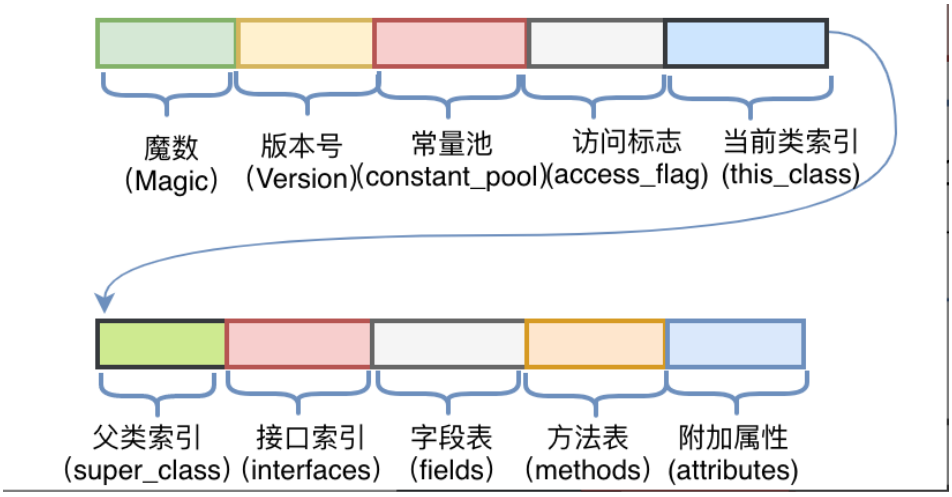


图 3　JVM 规定的字节码结构

（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。

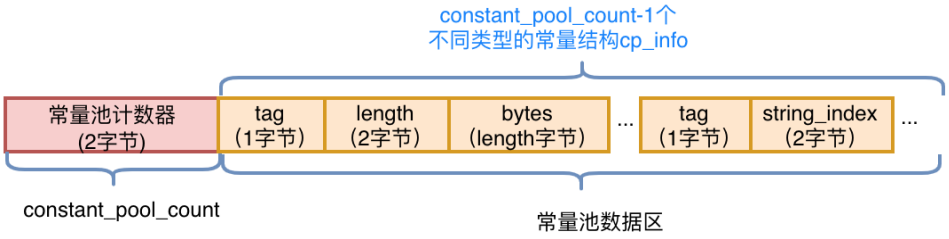


图 4　常量池的结构

- 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。

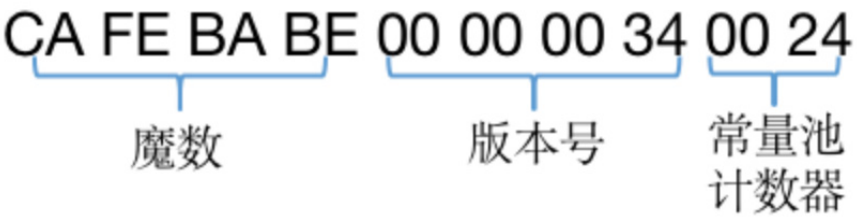


图 5　前十个字节及含义

- 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

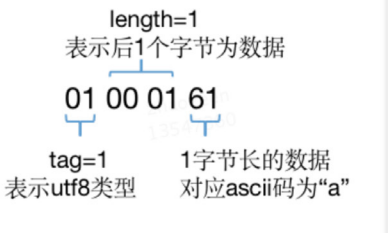


图 7　CONSTANT_utf8_info 的结构（左）及示例（右）

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：

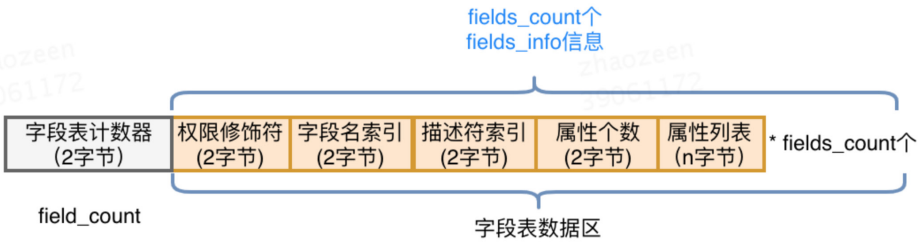


图 10　字段表结构

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：

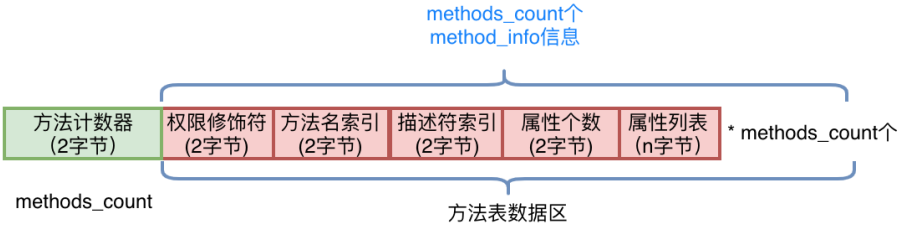


图 12　方法表结构

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：

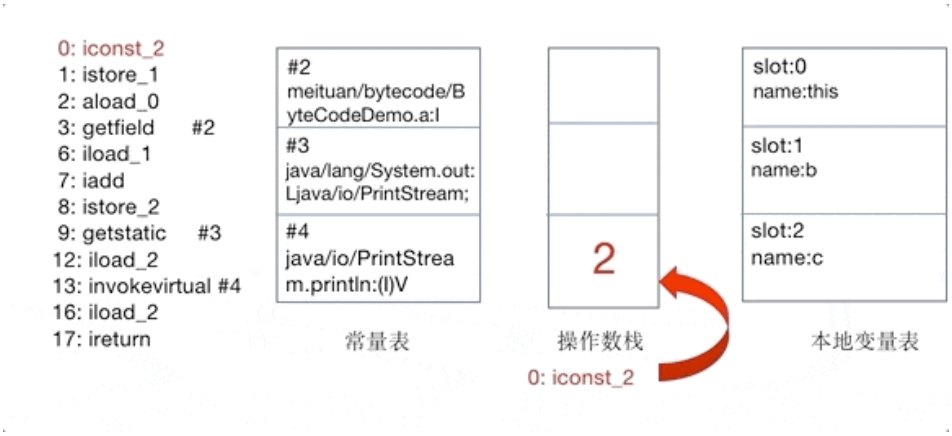


图 14　控制操作数栈示意图

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。

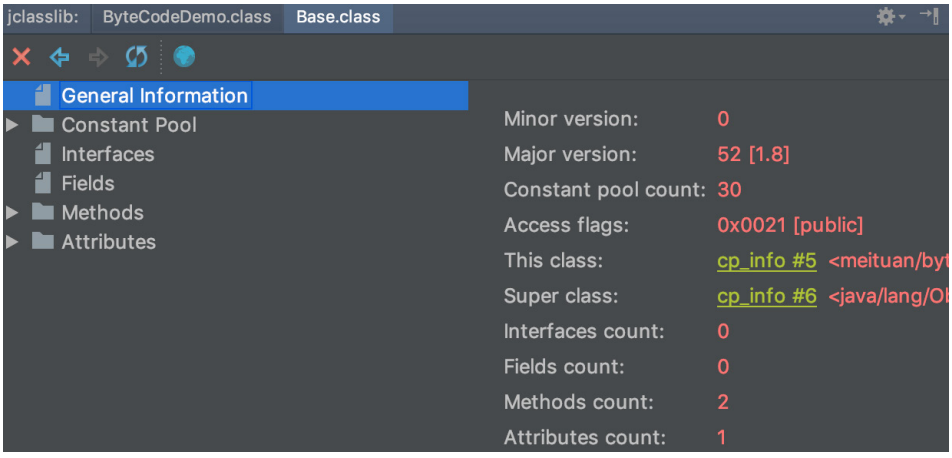


图 15　jclasslib 查看字节码

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。

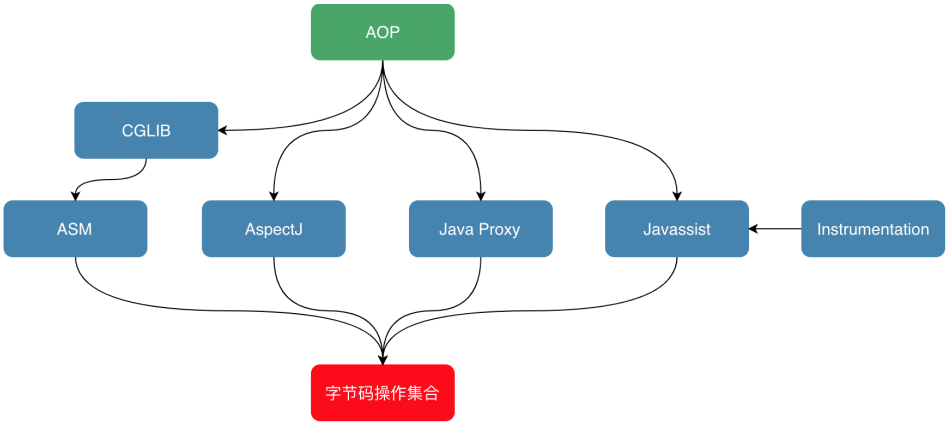


图 16　字节码增强技术

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。

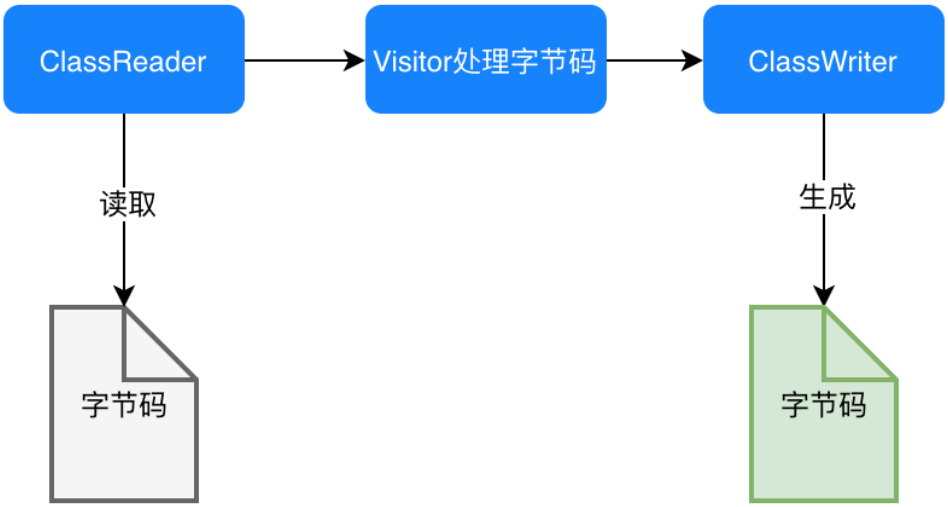


图 17　ASM 修改字节码

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：

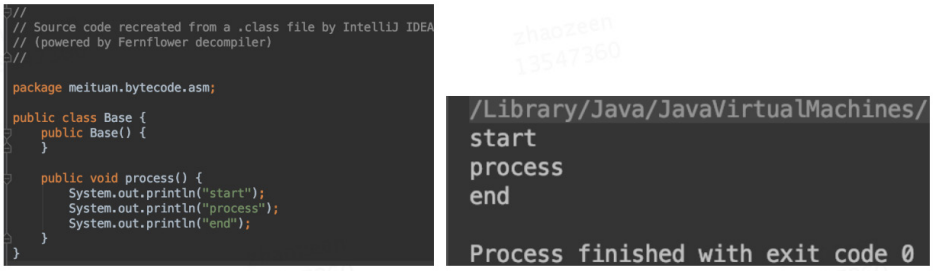


图 18　ASM 实现 AOP 的效果

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。

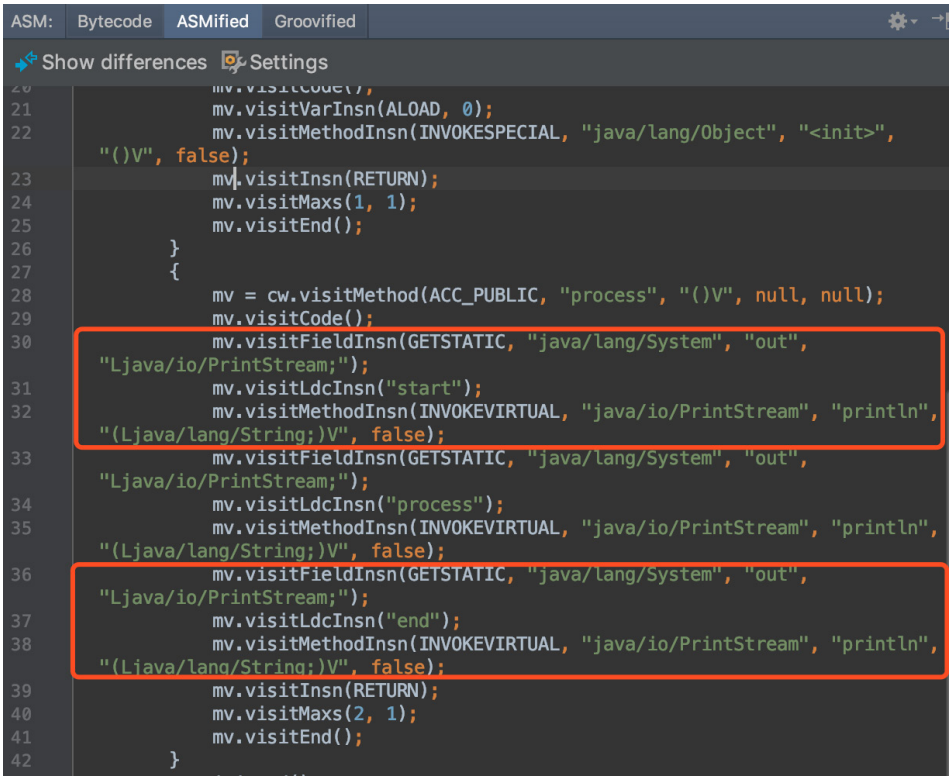


图 19　ASM Bytecode Outline

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：

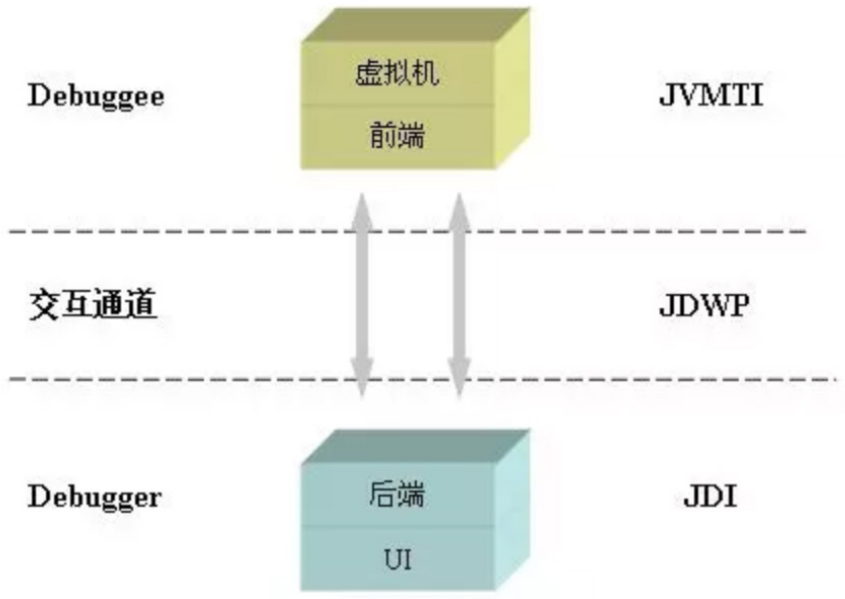


图 21 JPDA

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

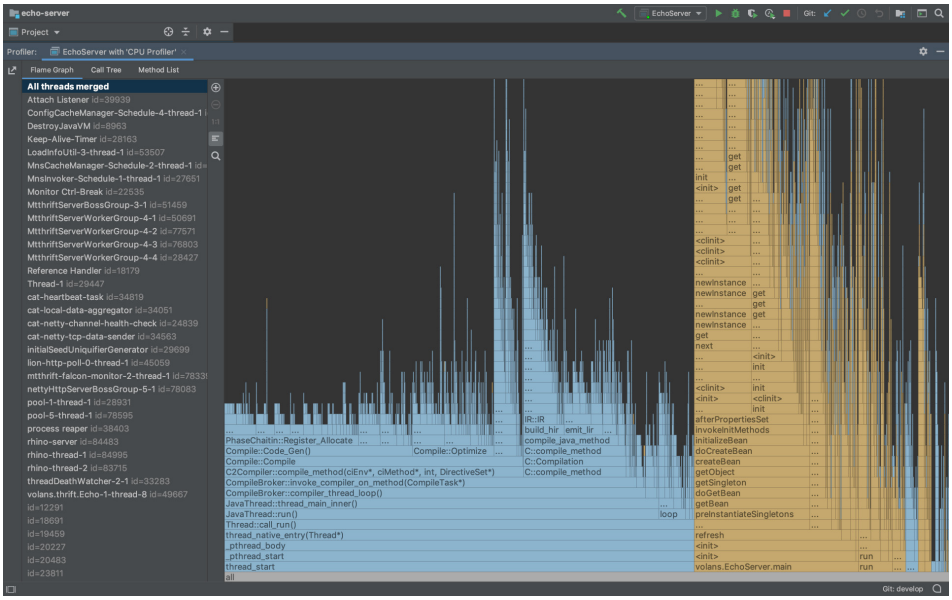
研发人员在遇到线上报警或需要优化系统性能时，常常需要分析程序运行行为和性能瓶颈。Profiling 技术是一种在应用运行时收集程序相关信息的动态分析手段，常用的 JVM Profiler 可以从多个方面对程序进行动态分析，如 CPU、Memory、Thread、Classes、GC 等，其中 CPU Profiling 的应用最为广泛。CPU Profiling 经常被用于分析代码的执行热点，如“哪个方法占用 CPU 的执行时间最长”、“每个方法占用 CPU 的比例是多少”等等，通过 CPU Profiling 得到上述相关信息后，研发人员就可以轻松针对热点瓶颈进行分析和性能优化，进而突破性能瓶颈，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。

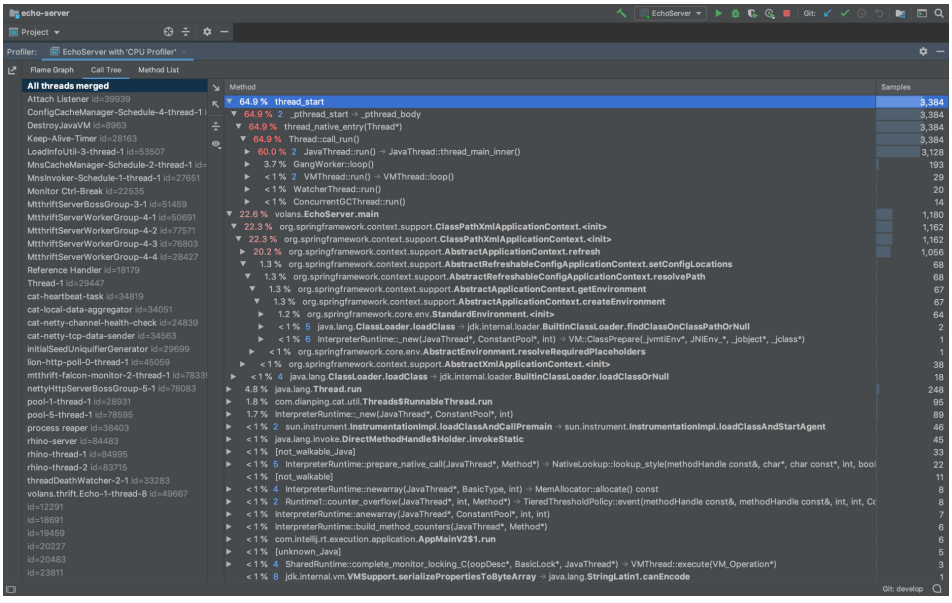
CPU Profiler 简介

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：

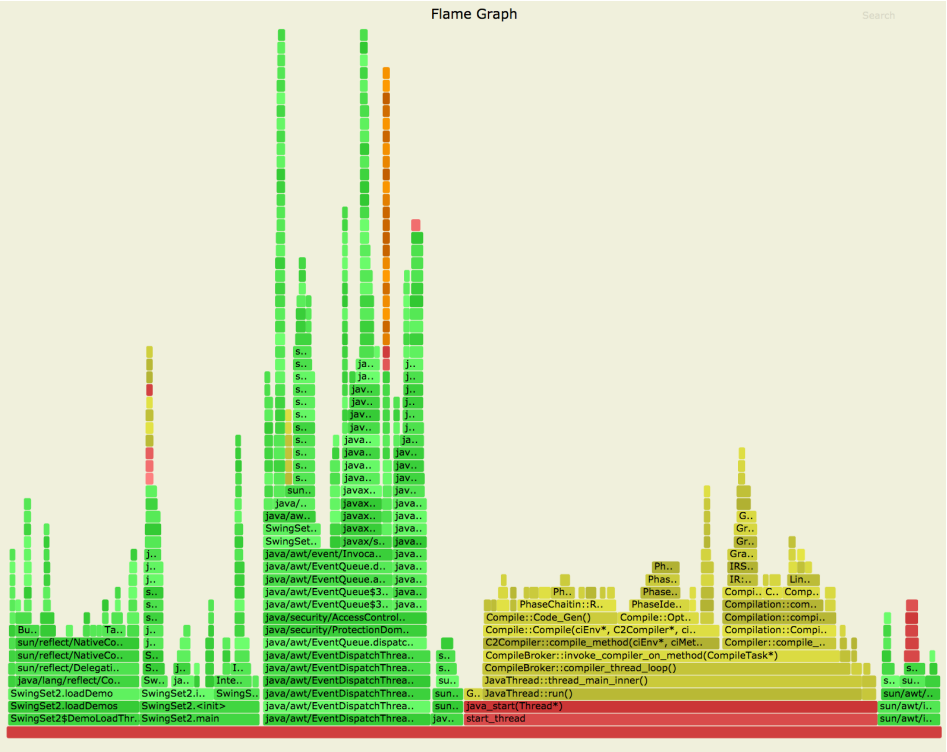
```
base_func;func1;func2;func3 10
base_func;funca;funcb 15
```

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：

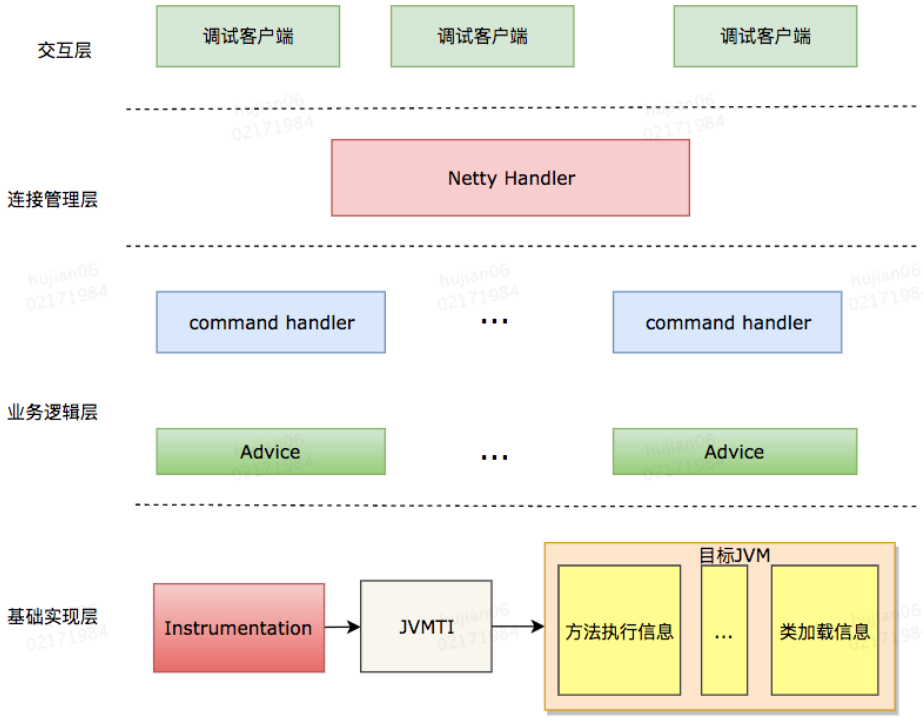


图 1

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：

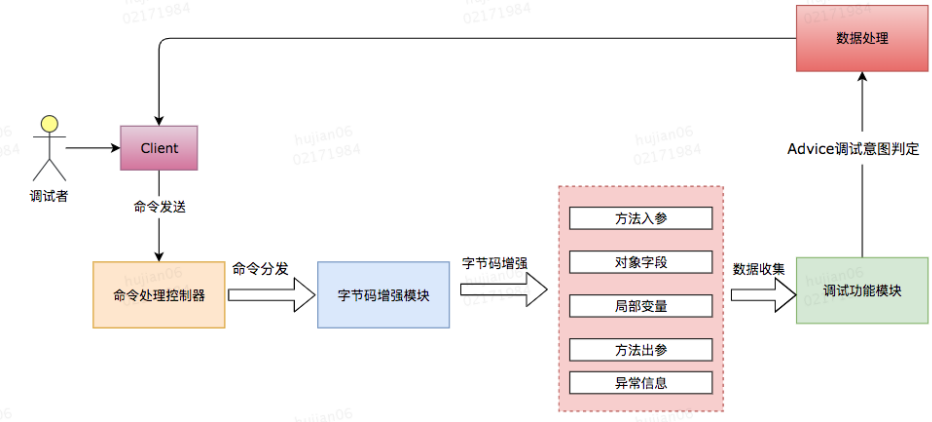


图 2

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。

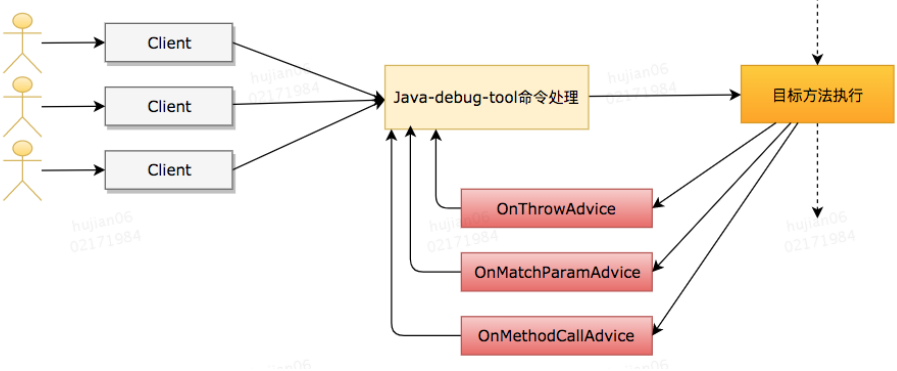


图 3

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：

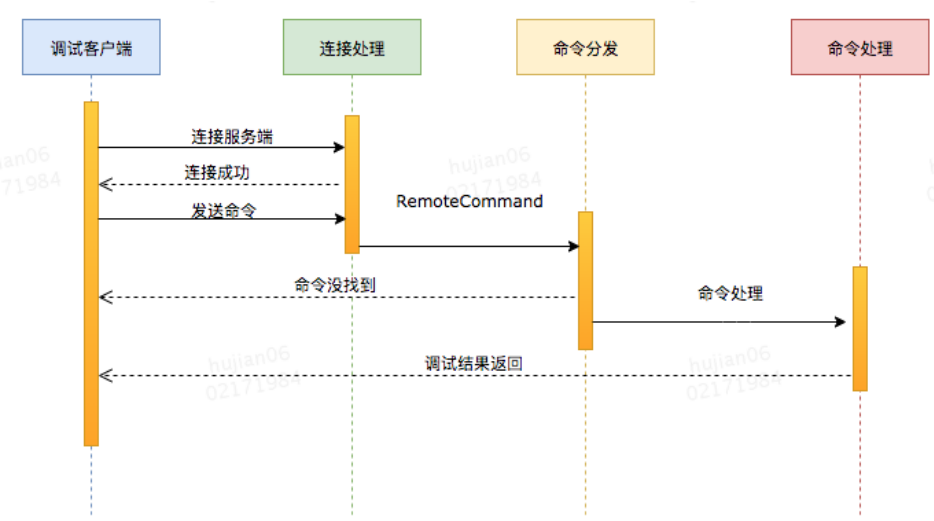


图 4

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）

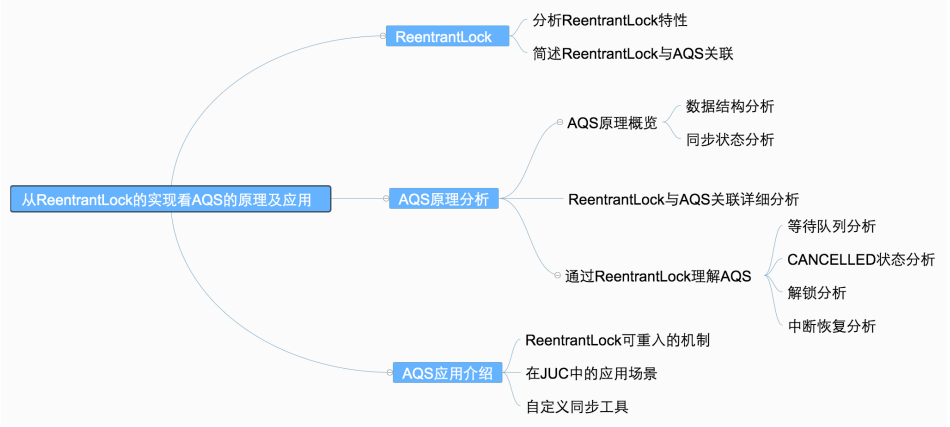
从 ReentrantLock 的实现看 AQS 的原理及应用

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

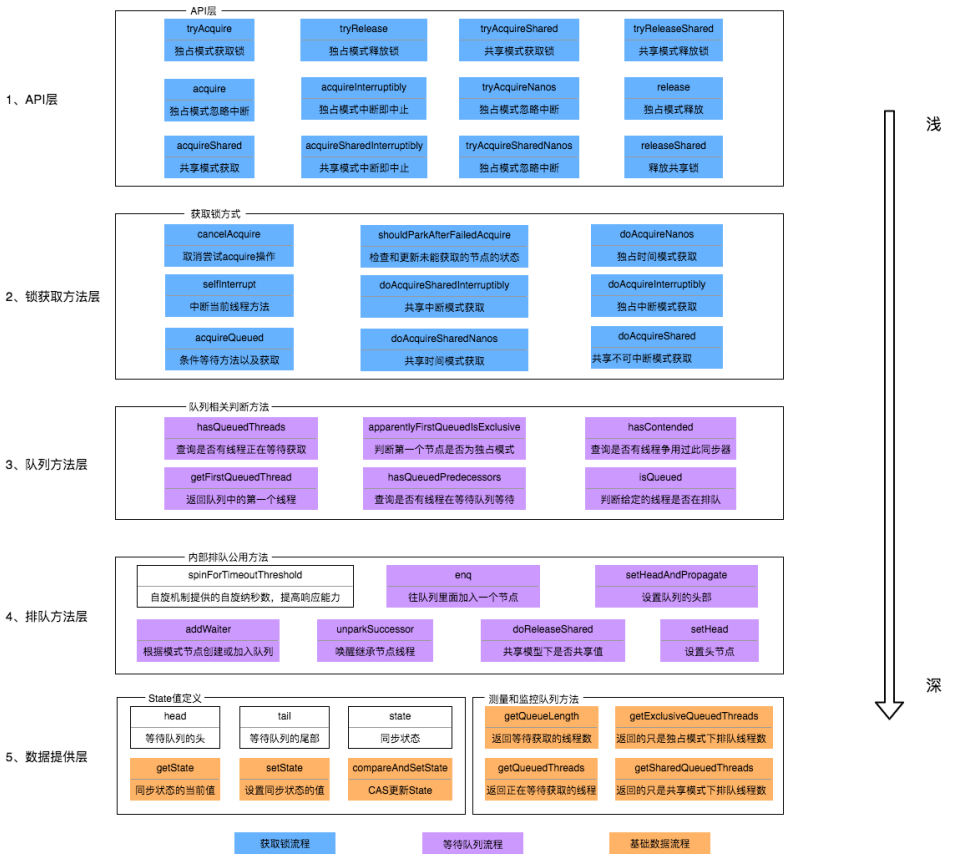
锁，但是具体是如何加锁的呢?

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）。

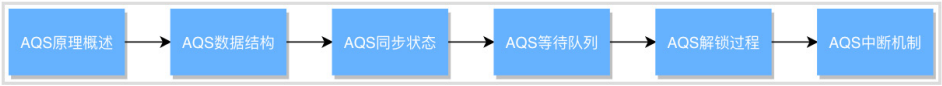
2. AQS

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：

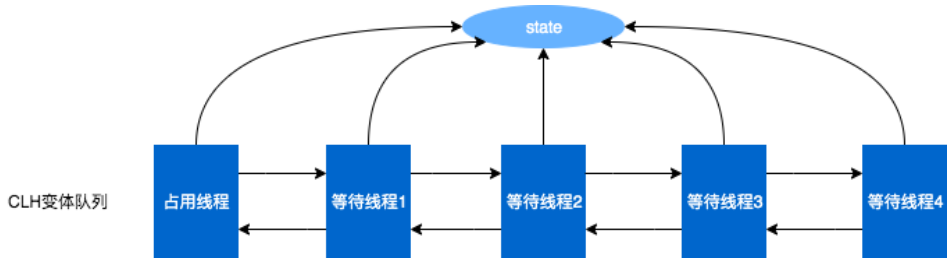


2.1 原理概览

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。

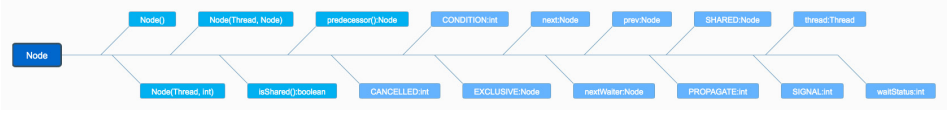
主要原理图如下：



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

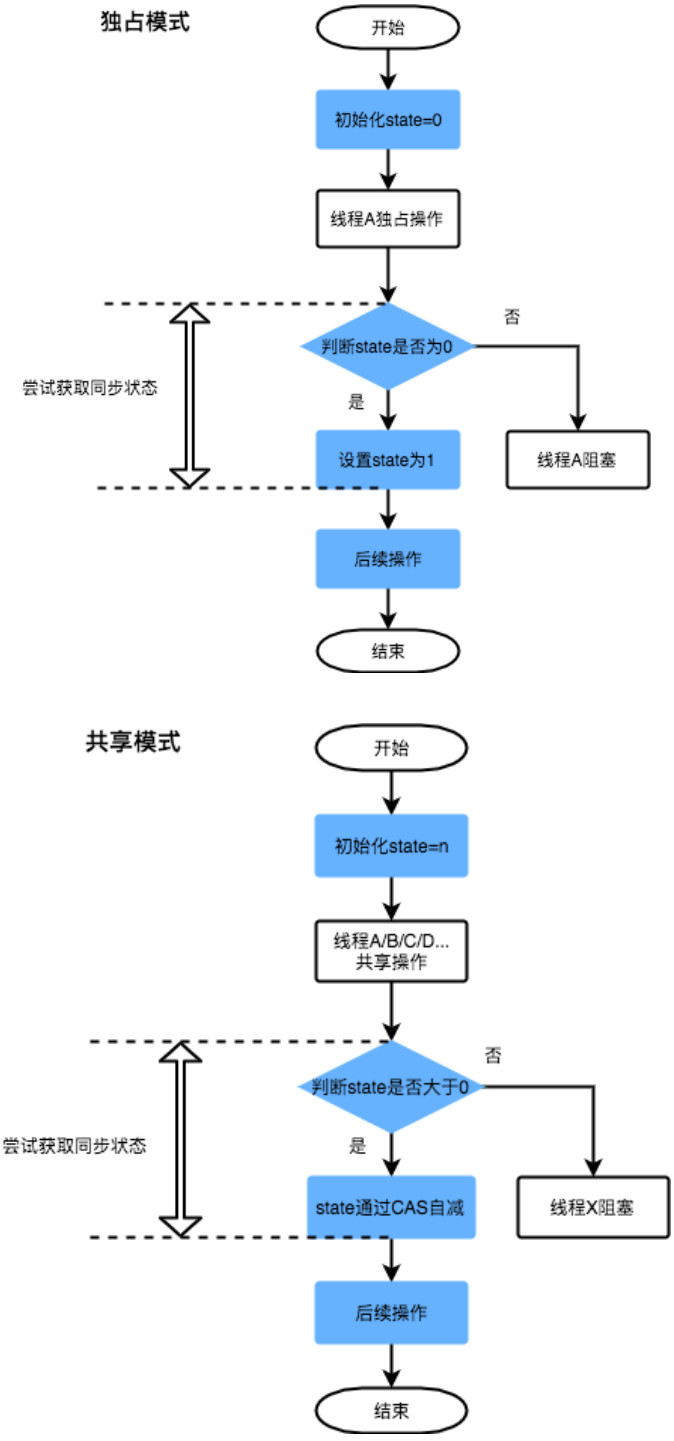
在了解数据结构后，接下来了解一下 AQS 的同步状态——State。AQS 中维护了一个名为 state 的字段，意为同步状态，是由 Volatile 修饰的，用于展示当前临界资源的获锁情况。

```
// 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。

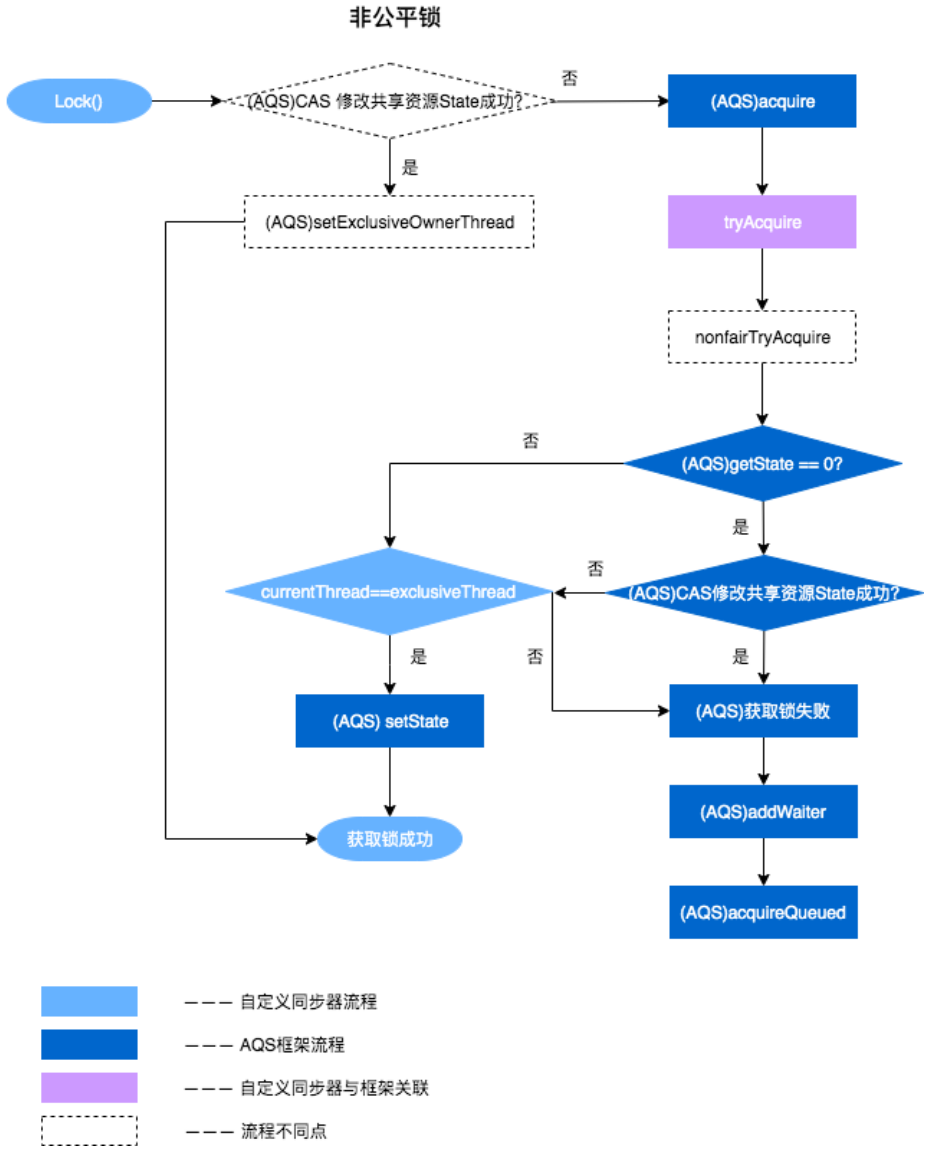
2.2 AQS 重要方法与 ReentrantLock 的关联

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）：

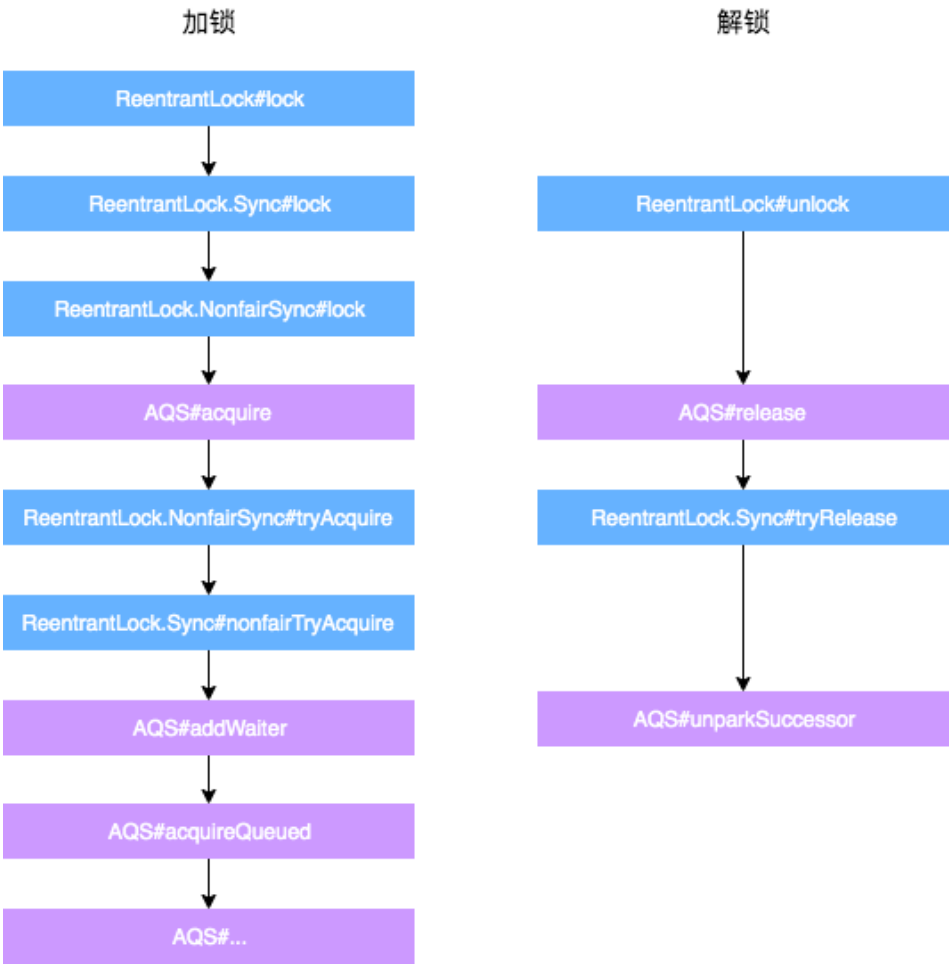
方法名	描述
protected boolean isHeldExclusively()	该线程是否正在独占资源。只有用到 Condition 才需要去实现它。
protected boolean tryAcquire(int arg)	独占方式。arg 为获取锁的次数，尝试获取资源，成功则返回 True，失败则返回 False。
protected boolean tryRelease(int arg)	独占方式。arg 为释放锁的次数，尝试释放资源，成功则返回 True，失败则返回 False。
protected int tryAcquireShared(int arg)	共享方式。arg 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。



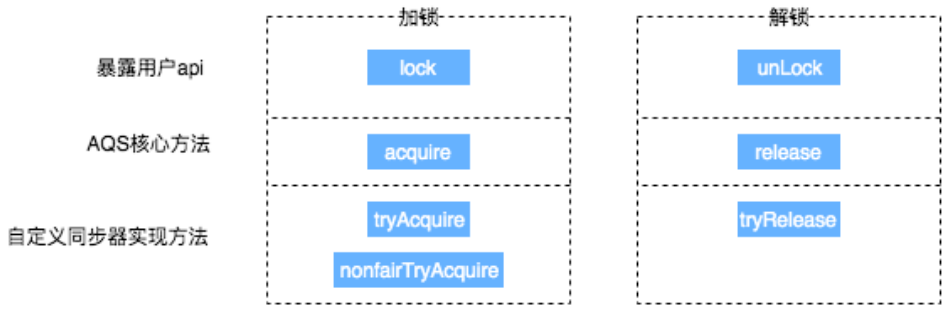
加锁：

- 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。

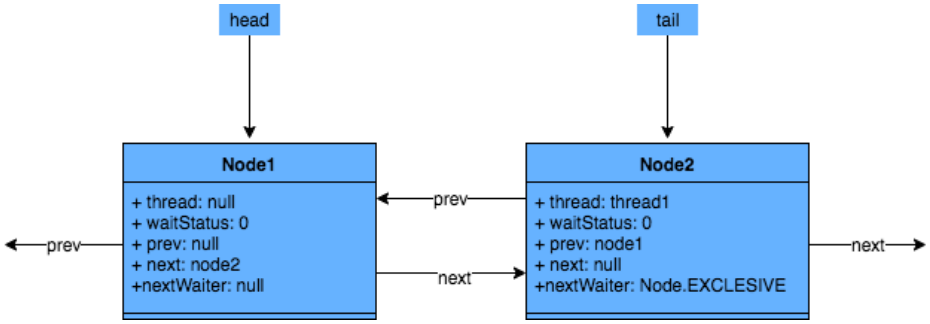
```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

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);
      }
}
```

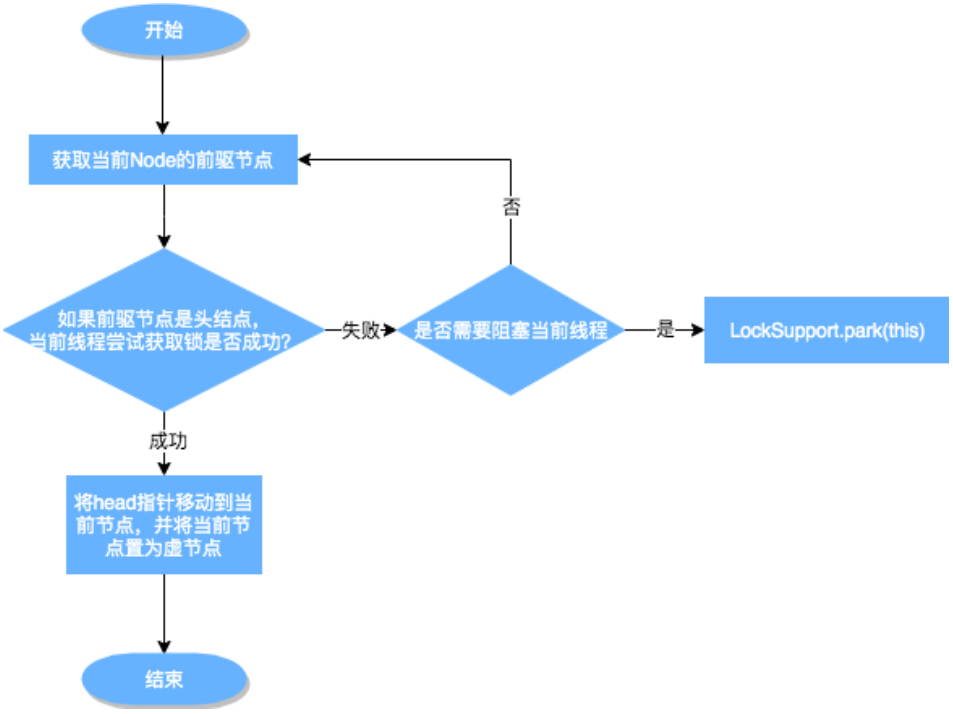
注：setHead 方法是把当前节点置为虚节点，但并没有修改 waitStatus，因为它是一直需要用的数据。

```
// 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;
}
```

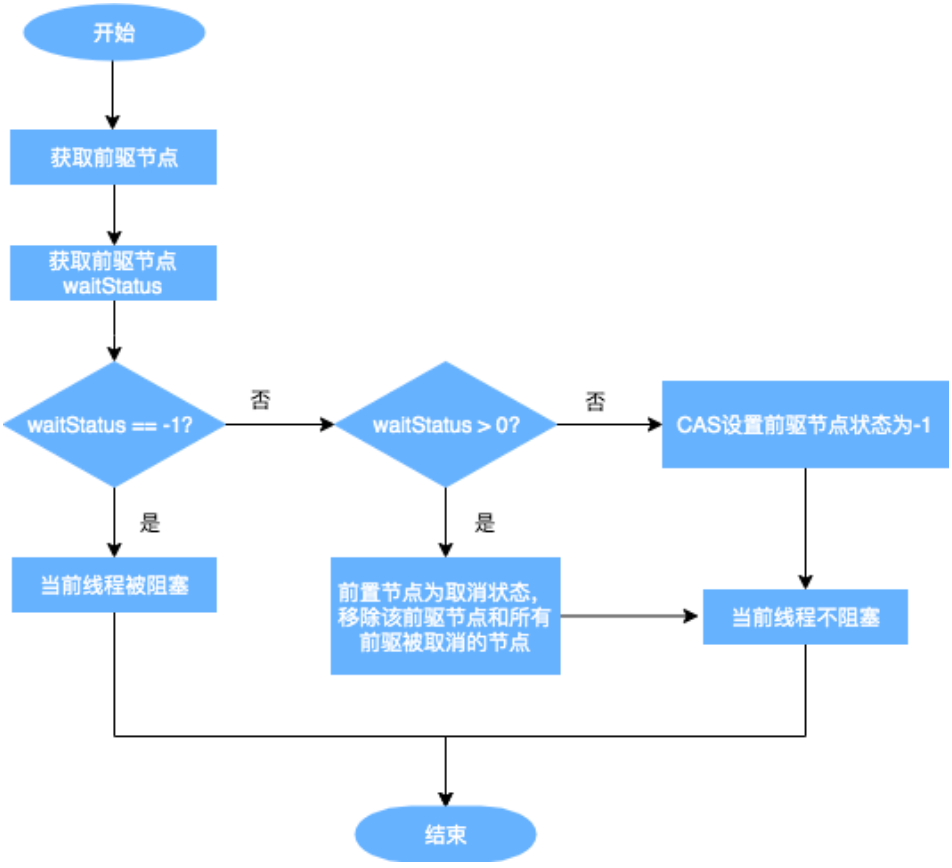
parkAndCheckInterrupt 主要用于挂起当前线程，阻塞调用栈，返回当前线程的中断状态。

```
// 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：

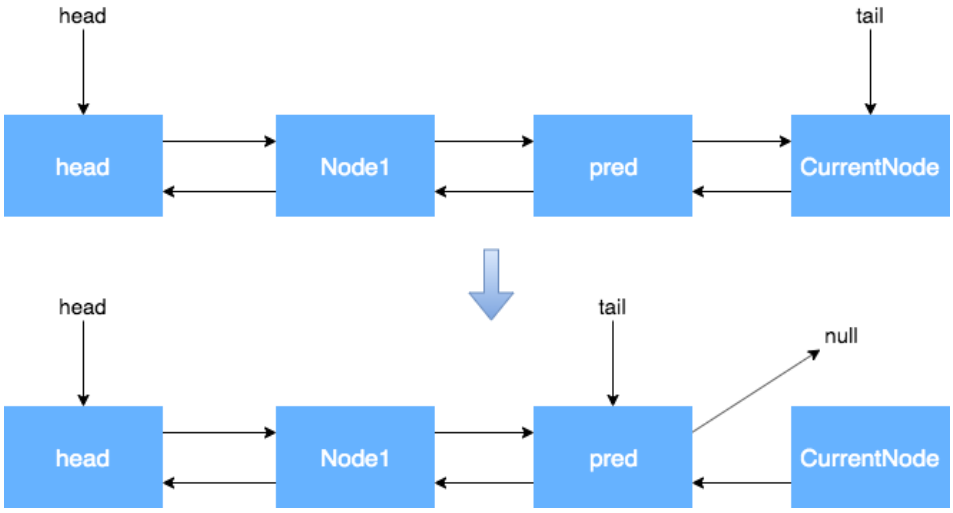
（1）当前节点是尾节点。

（2）当前节点是 Head 的后继节点。

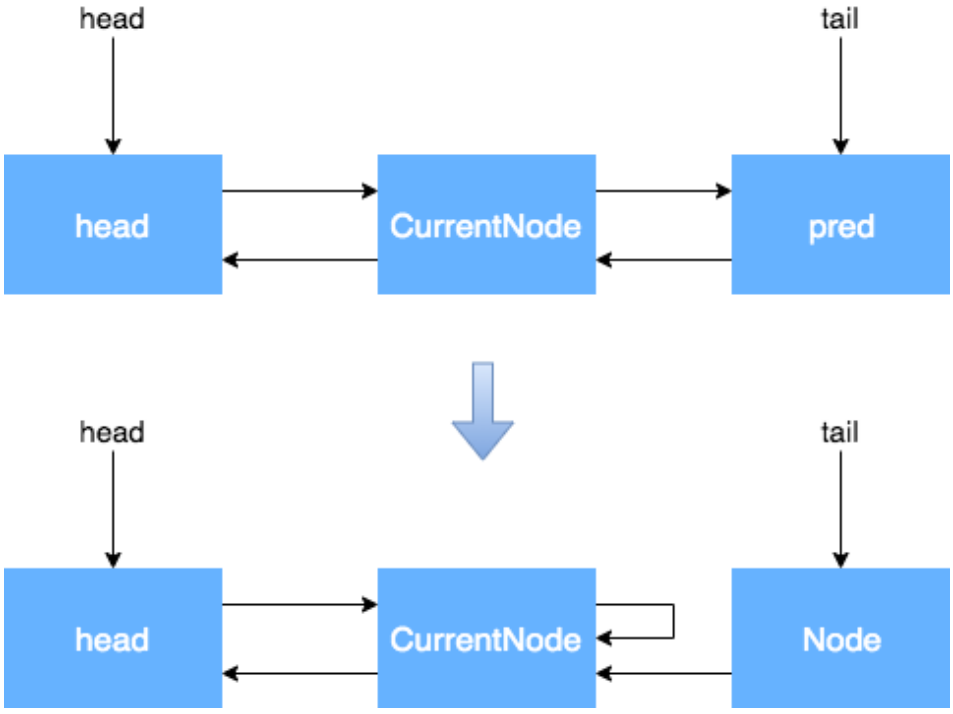
（3）当前节点不是 Head 的后继节点，也不是尾节点。

根据上述第二条，我们来分析每一种情况的流程。

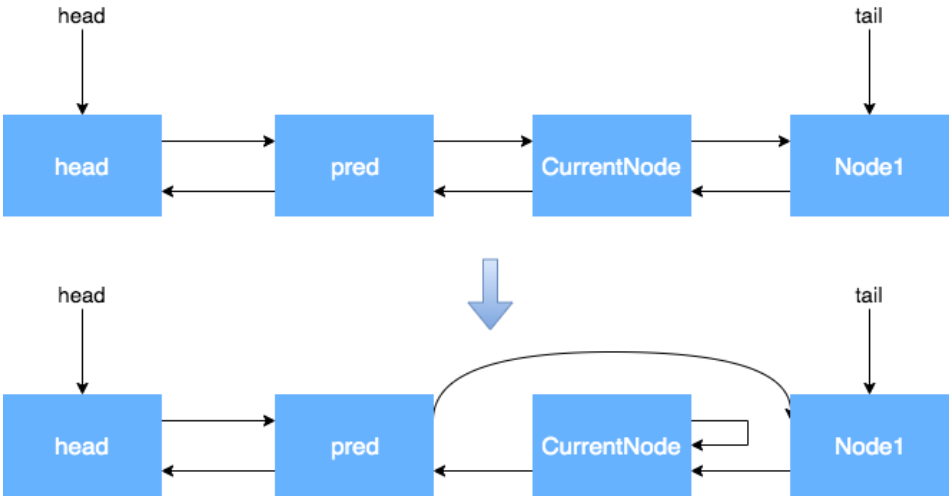
当前节点是尾节点。



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。

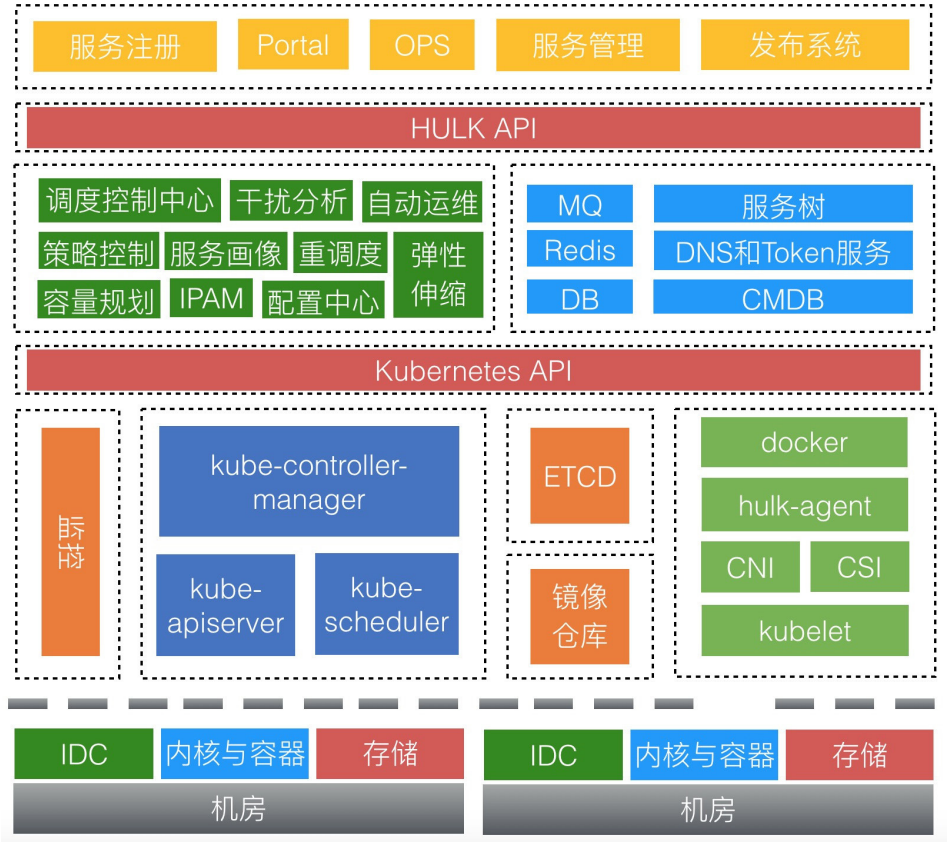
之后，在生产环境中经过不断摸索和尝试，我们逐渐意识到，仅仅满足于集群的弹性伸缩能力是不够的，成本和效率肯定是未来必将面临且更为棘手的问题。我们吸取了 2 年来 HULK 1.0 的开发和运维经验，在架构和支撑系统层面做了进一步优化和改进，并借助于生态和开源的力量来为 HULK 赋能，即引入了开源的集群管理与调度系统 Kubernetes，期望能进一步提升集群管理、运行的效率和稳定性，同时降低资源成本。所以我们从自研平台转向了开源的 Kubernetes 系统，并基于 Kubernetes 系统打造了更加智能化的集群管理与调度系统——HULK2.0。

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

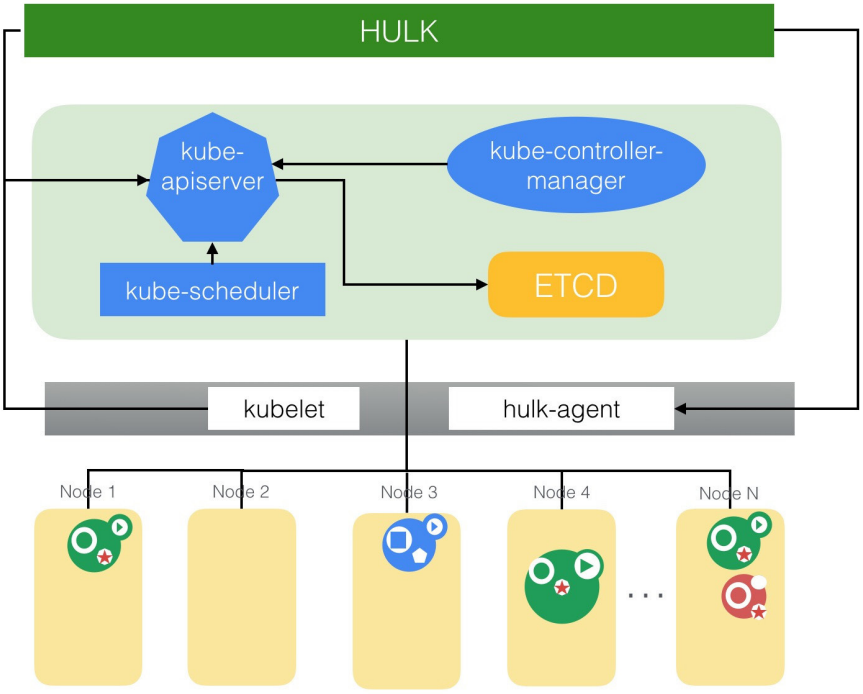
自上而下来看，美团集群管理与调度平台面向全公司服务，有各个主要业务线、统一的 OPS 平台以及 Portal 平台，HULK 不可能针对每个平台定制化接口和解决方案，所以需要将多样的业务和需求抽象收敛，最终统一通过 HULK API 来屏蔽 HULK 系统的细节，做到 HULK 与上层业务方的解耦。HULK API 是对业务层和资源需求的抽象，是外界访问 HULK 的唯一途径。

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。

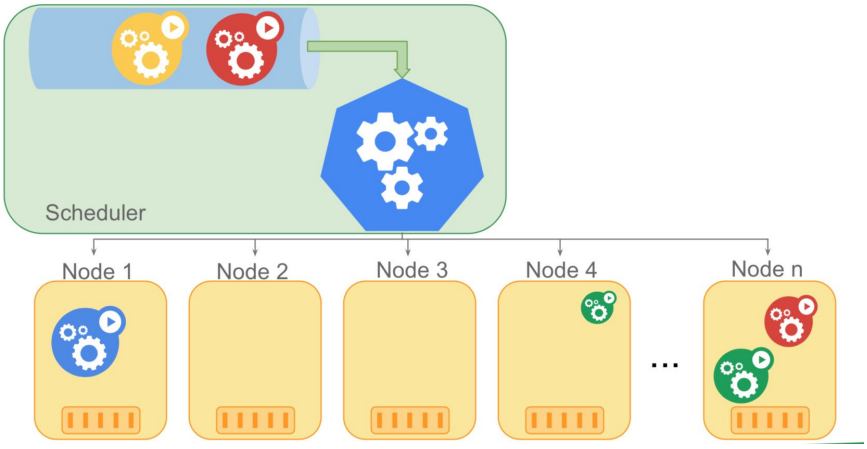
- 集群数据可视化：对当前集群状态，包括宿主机资源状态、服务数、Pod 数、容器化率、服务状态、扩缩容数据等等可视化；并提供了界面化的服务配置、宿主机下线以及 Pod 迁移操作入口。
- 容量规划与预测：提前感知集群资源状态，预先准备资源；基于规则和机器学习的方式感知流量和高峰，保证业务正常、稳定、高效地运行。

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：



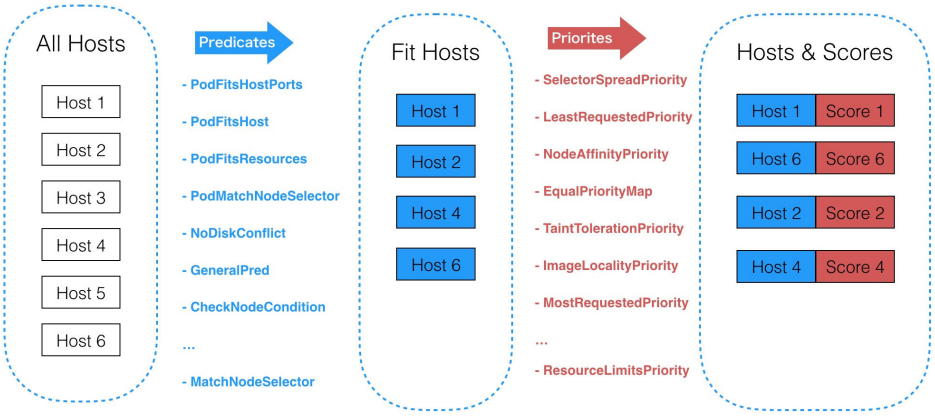
kube-scheduler 示意图

（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。

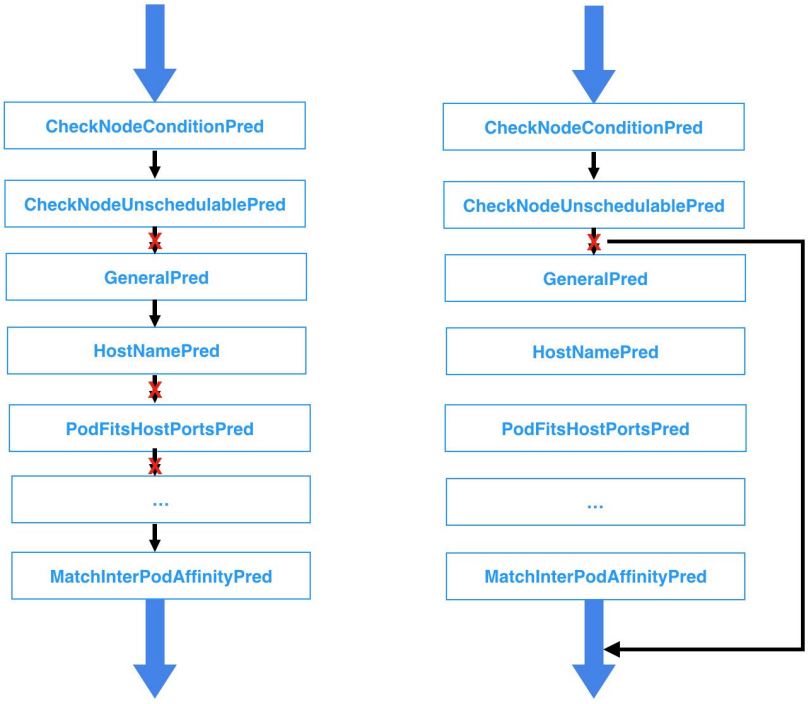


kube-scheduler 调度过程

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

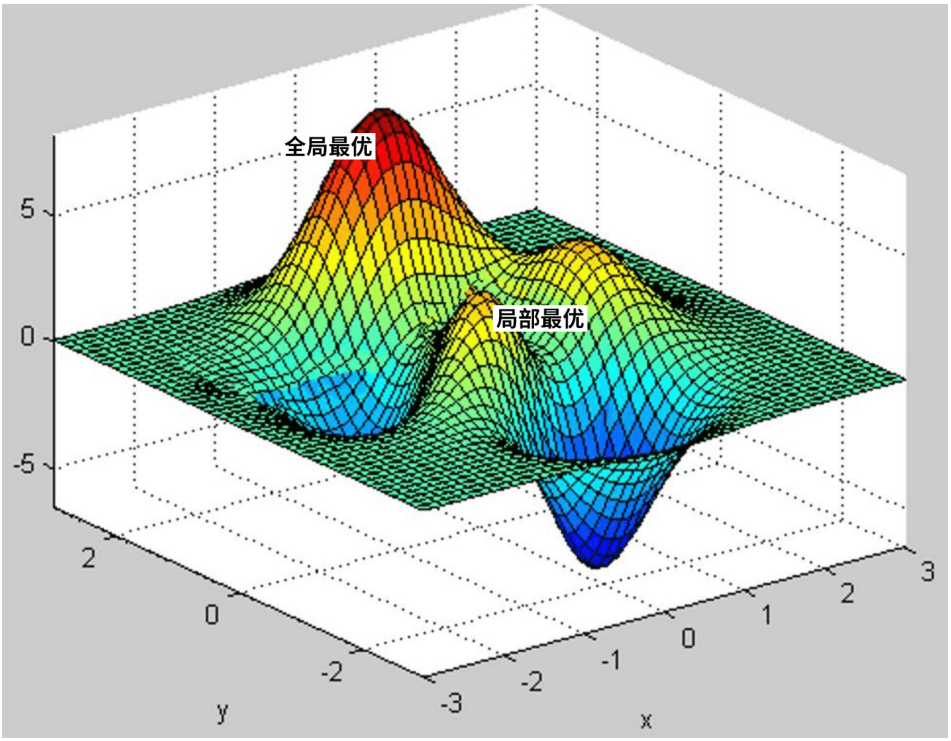
我们把该项优化贡献给了 Kubernetes 社区（详见 PR），增加了 alwaysCheckAllPredicates 策略选项，并在 Kubernetes1.10 版本发布并开始作为默认的调度策略，当然你也可以通过设置 alwaysCheckAllPredicates=true 使用原先的调度策略。

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。

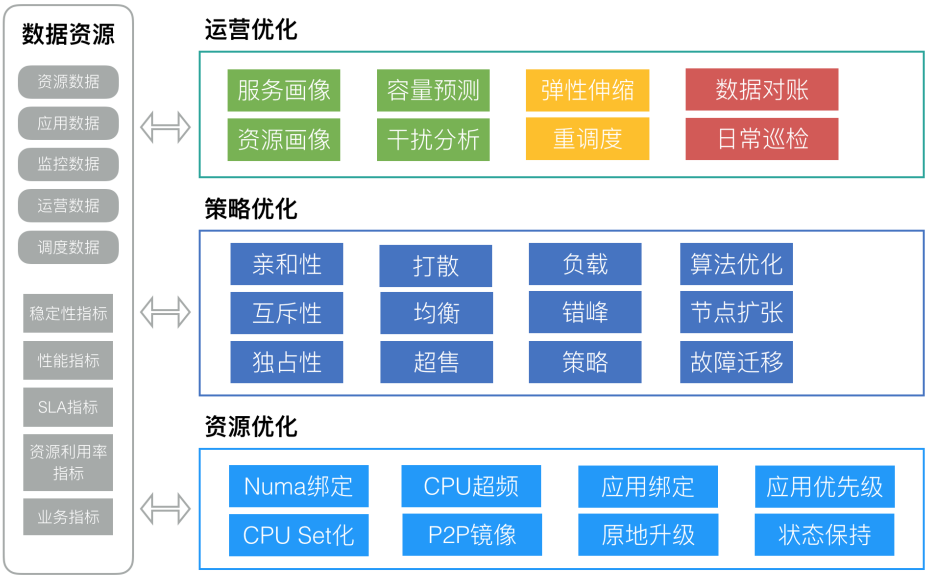
镜像分发

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

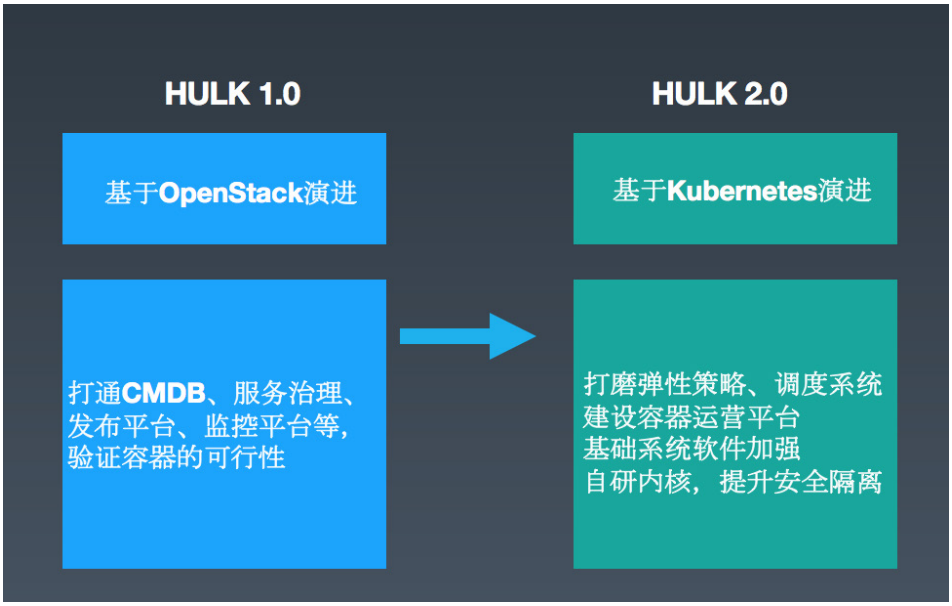
HULK 是美团的容器集群管理平台。在 HULK 之前，美团的在线服务大部分部署都是在 VM 上，在此期间，我们遇到了很大的挑战，主要包括以下两点：

- 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。

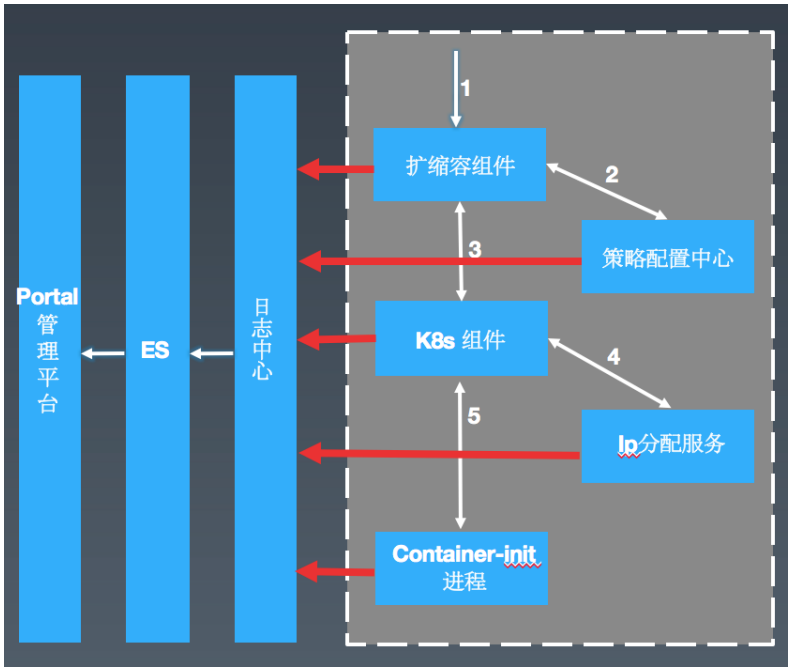
最底层的 HULK Agent 是我们在每个 Node 上的代理程序。此前，在美团技术团队官方博客上，我们也分享过底层的镜像管理和容器运行时相关内容，参见《美团容器技术研发实践》一文。而本文将重点阐述容器编排（调度系统）和容器弹性（弹性伸缩平台），以及团队遇到的一些问题以及对应的解决方案，希望对大家能有所启发。

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。

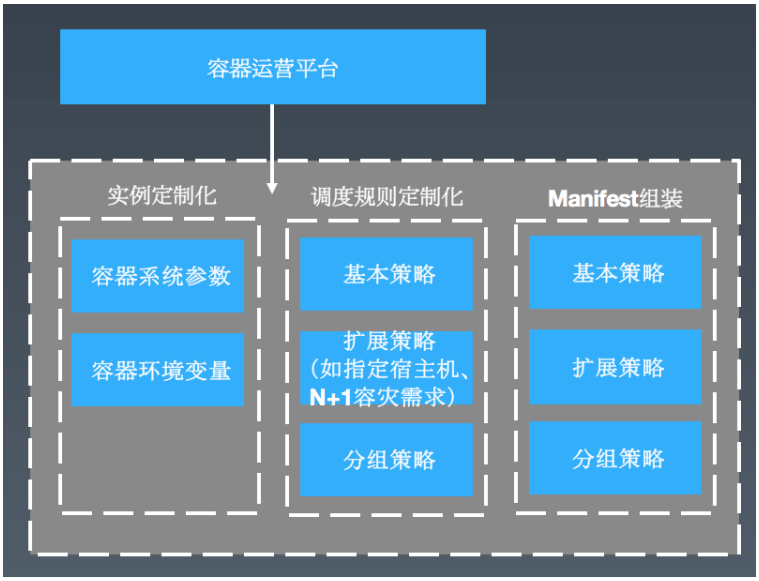
3.2 业务定制化需求

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。

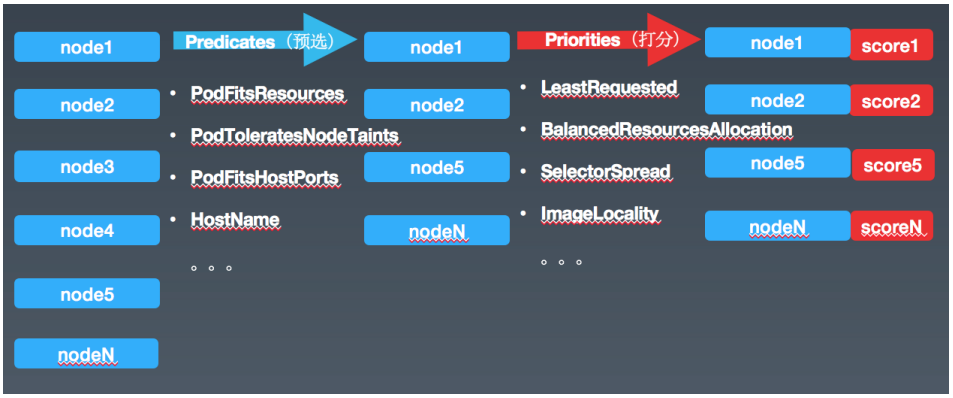


- 实例基本配置，比如业务想给机器加 Set 化、泳道标识。
- 实例的扩展配置：如部分业务，比如某些服务想将实例部署在包含特定硬件的宿主机，会对核心业务有 N+1 的容灾需求，并且还需要将实例部署在不同的 IDC 上。
- 相同配置的应用可以创建一个组，将应用和组进行关联。

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。

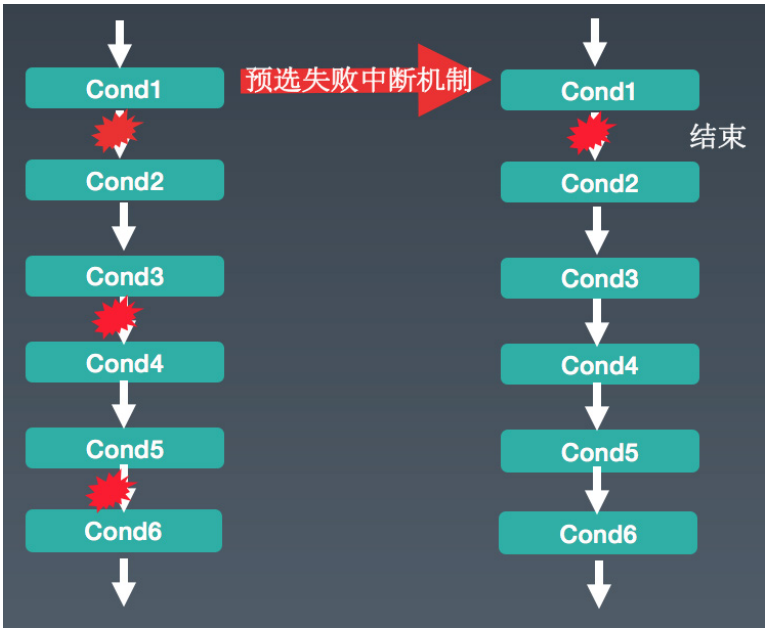
Priorities 打分阶段（一堆的优先级函数）：

- 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。

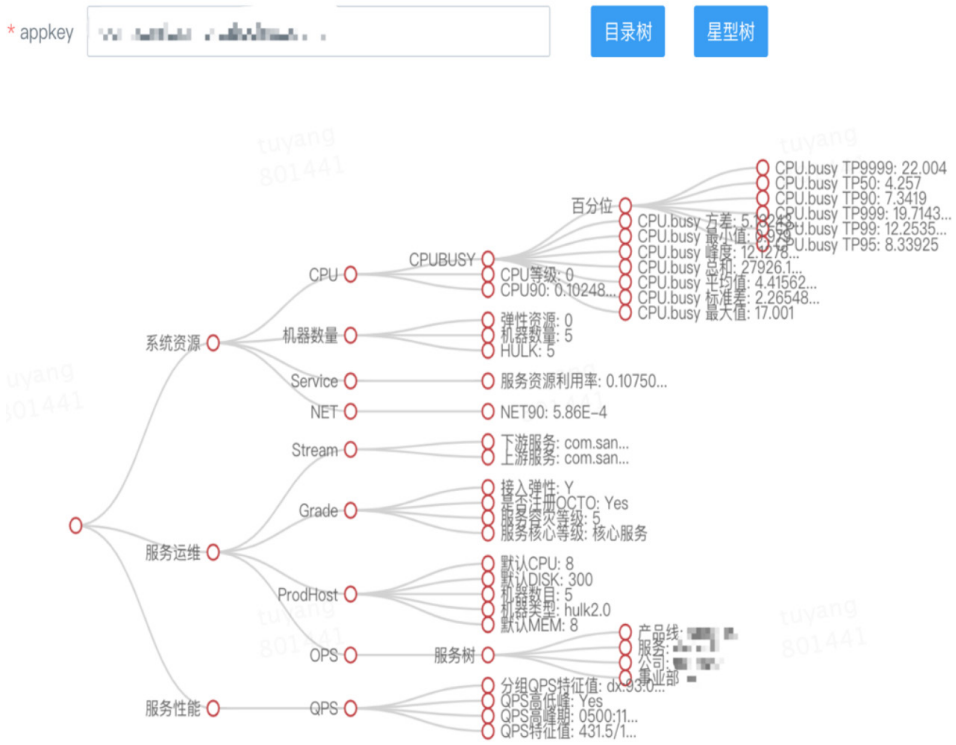
解法：当前 Node 中，如果遇到一个预选条件不满足（比较像是短路径原则），就将这个 Node 过滤掉，大大减少了计算量，调度性能也得到大幅提升。



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：

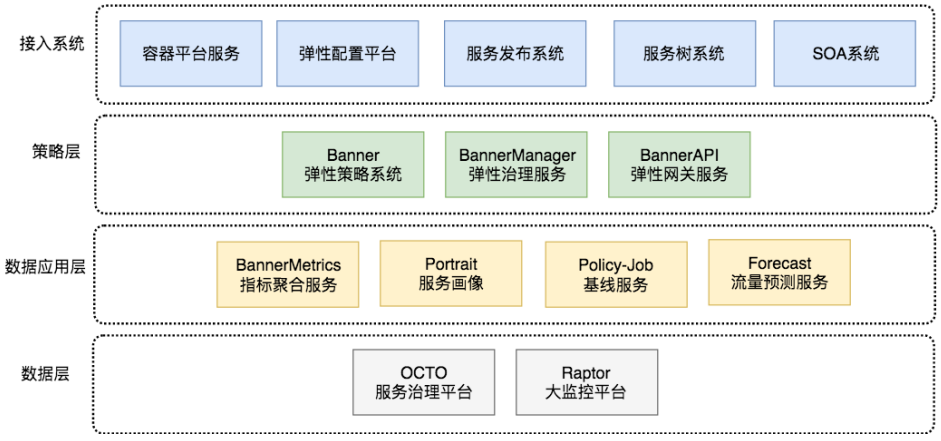
（1）容器重启 / 迁移场景：

- 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

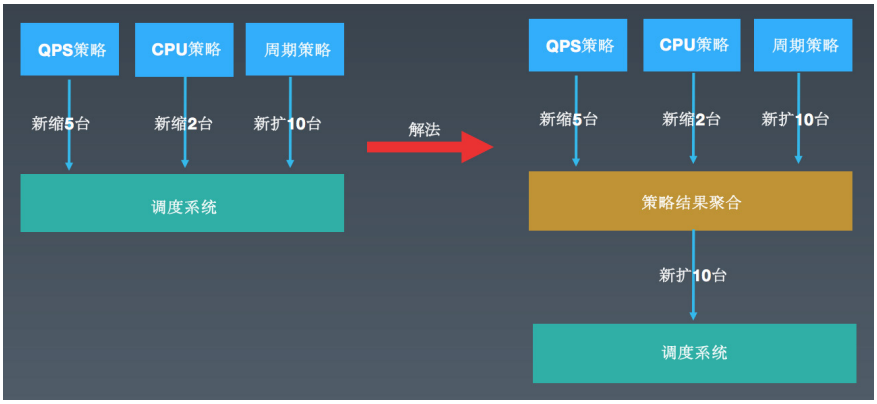
弹性伸缩平台整体架构图如下：



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



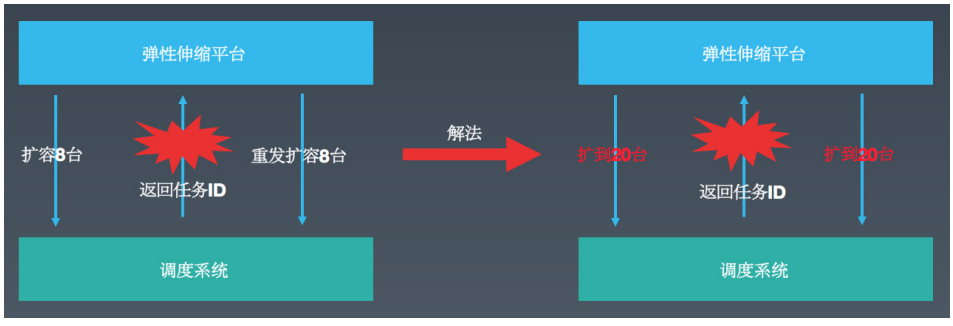
如上图所示，一个业务配置了 2 条监控策略和 1 条周期策略：

- **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。

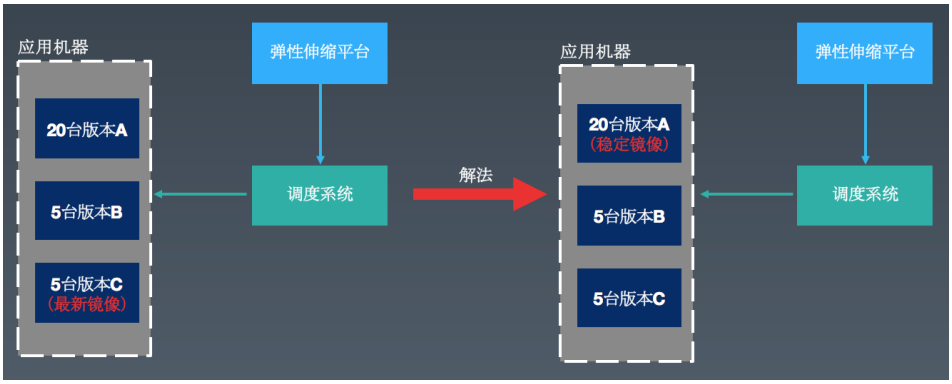
4.2 扩缩不幂等



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。

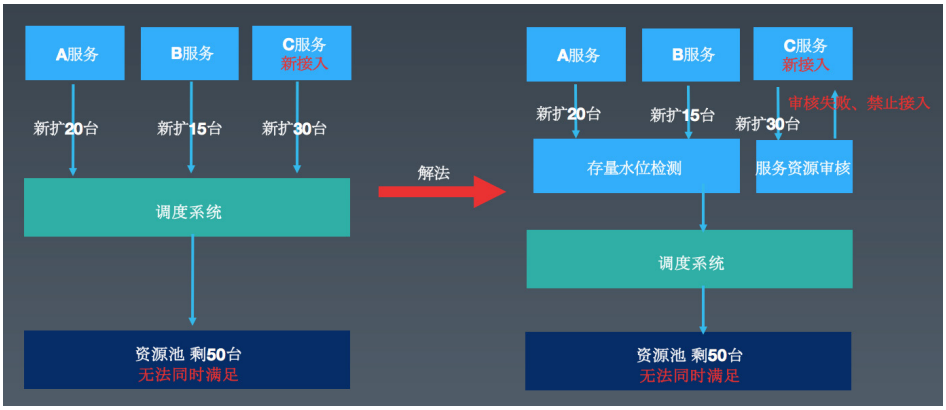
4.3 线上代码多版本



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

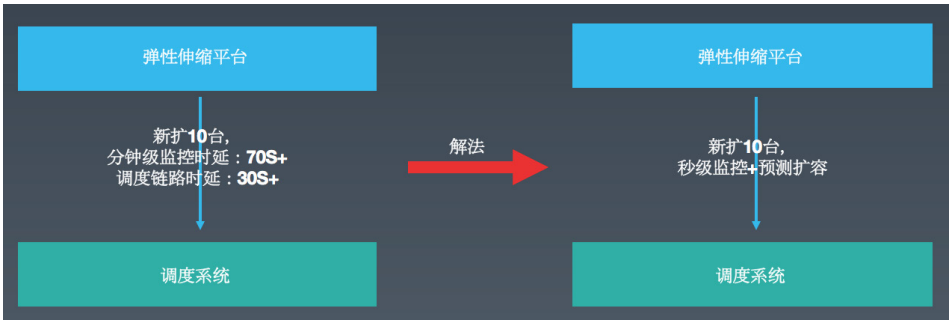


如上图所示，存量中有 2 个服务，一个需要扩容 20 台，一个需要扩容 15 台，这个时候如果新接入一个服务，同一时间需要扩容 30 台，但是资源池只剩余 50 台实例了。这个时候就意味着，谁先扩容谁就可以获得资源保障，后发起的请求就无法获得资源保障。

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

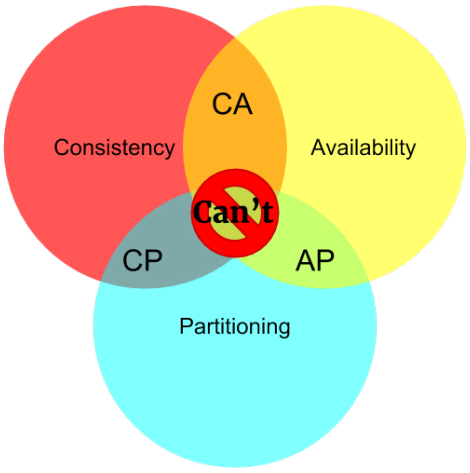
在列出产品需要实现的功能点、技术点后，再来分析下遇到的技术挑战，包括不限于以下几点：

- 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（$\log_8 n$），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。

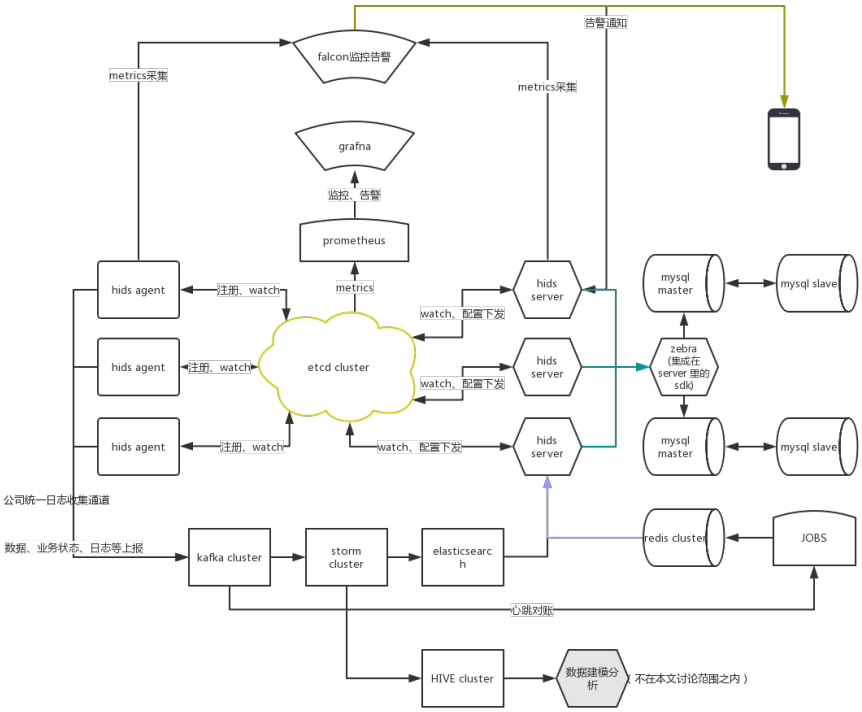
我们在设计产品架构时，为了安全性，开启了 TLS 证书认证，当节点变更时，证书的生成也同样要考虑到上面两种方案的影响，证书里需要包含固定 IP，以及 DNS 域名范围的两种格式。

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

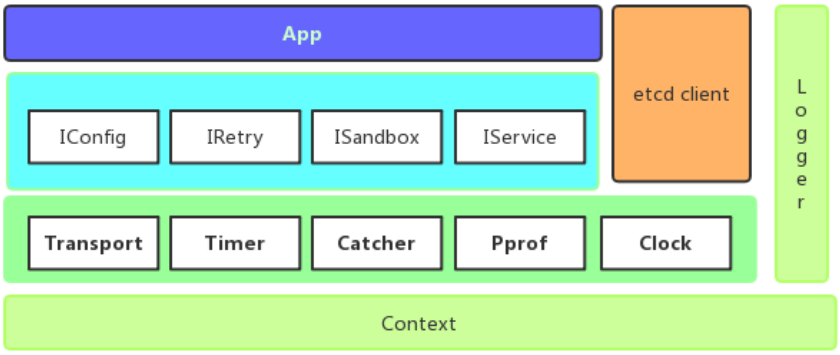
HIDS 产品研发完成后，部署的服务都运行着各种业务的服务器，业务的重要性排在第一，我们产品的功能排在后面。为此，确定了几个产品的大方向：

- 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。

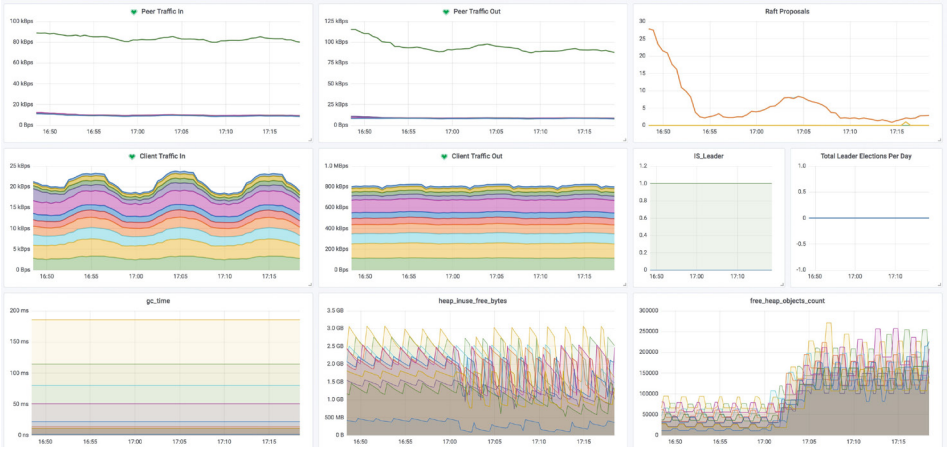
Agent 数量感知，依赖 Watch 数字，实时准确感知。

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

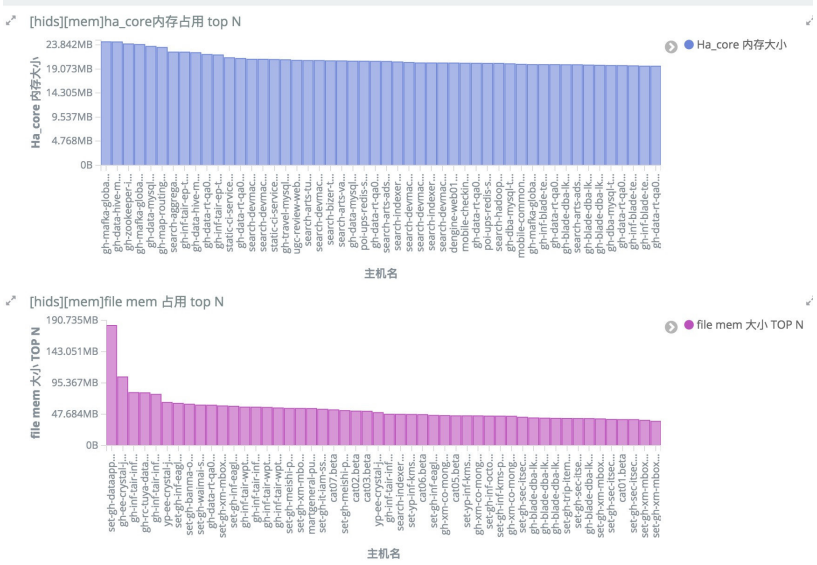
etcd 集群 Members 之间健康状况监控



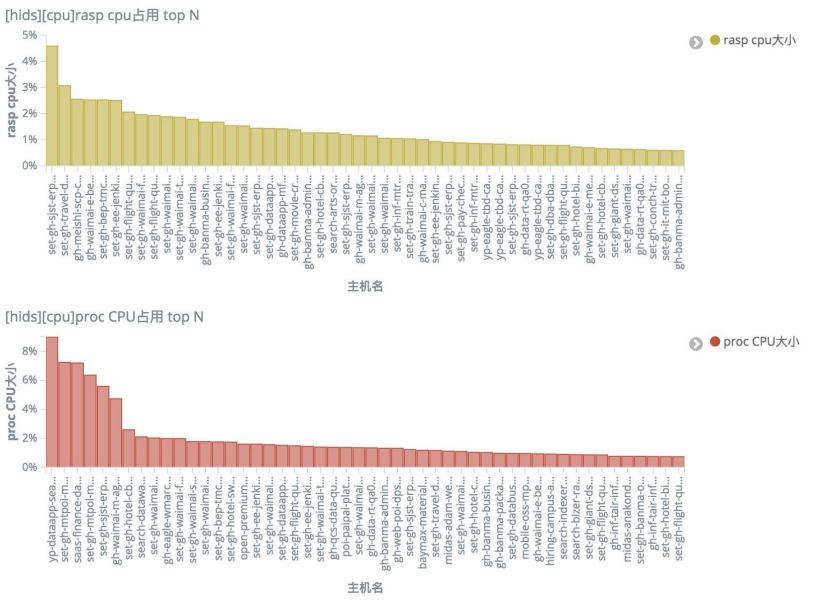
etcd-Grafana-GC-Heap-Objects

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

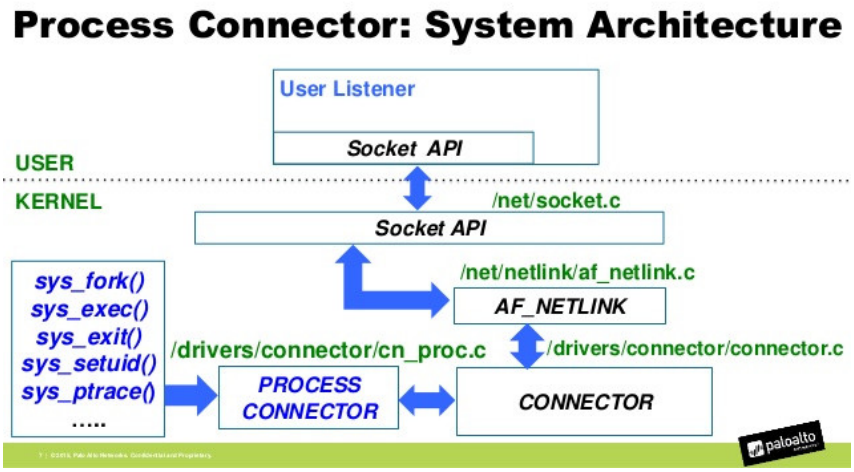
方案选择

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：

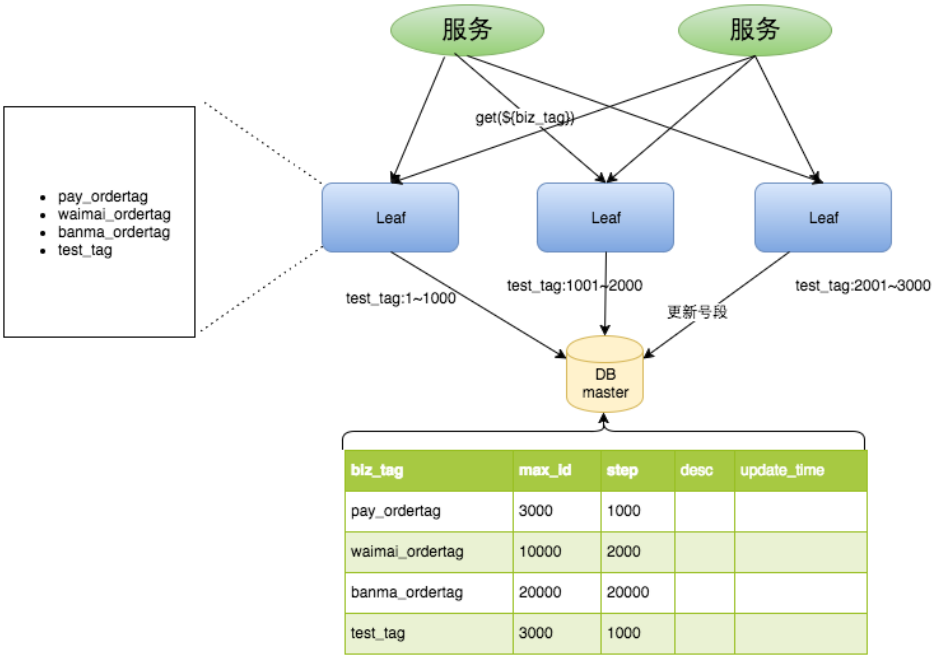
1. 全局唯一，绝对不会出现重复的 ID，且 ID 整体趋势递增。
2. 高可用，服务完全基于分布式架构，即使 MySQL 宕机，也能容忍一段时间的数据库不可用。
3. 高并发低延时，在 CentOS 4C8G 的虚拟机上，远程调用 QPS 可达 5W+，TP99 在 1ms 内。
4. 接入简单，直接通过公司 RPC 服务或者 HTTP 调用即可接入。

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：

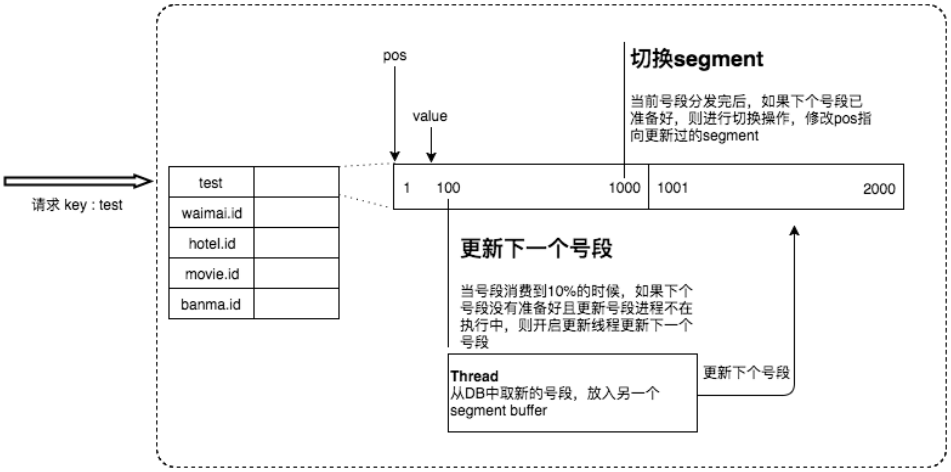
```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

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

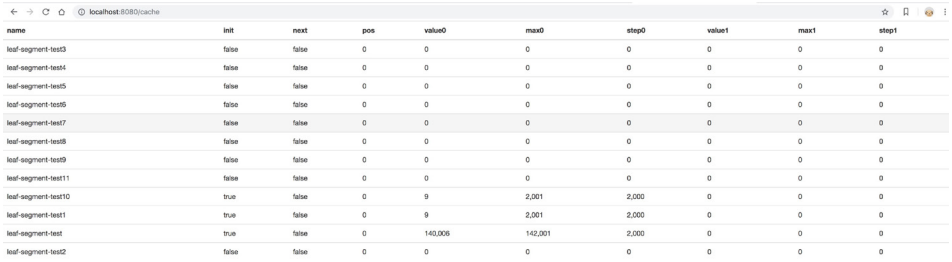
在 MySQL 这一层，Leaf 目前采取了半同步的方式同步数据，通过公司 DB 中间件 Zebra 加 MHA 做的主从切换。未来追求完全的强一致，会考虑切换到 MySQL Group Replication。

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

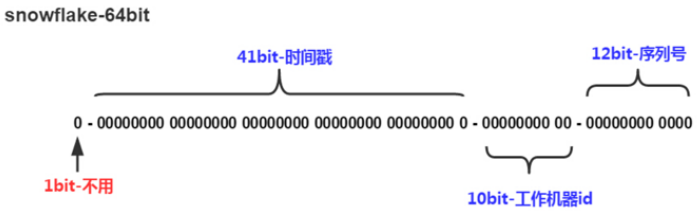
针对服务自身的监控，Leaf 提供了 Web 层的内存数据映射界面，可以实时看到所有号段的下发状态。比如每个号段双 buffer 的使用情况，当前 ID 下发到了哪个位置等信息都可以在 Web 界面上查看。



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。

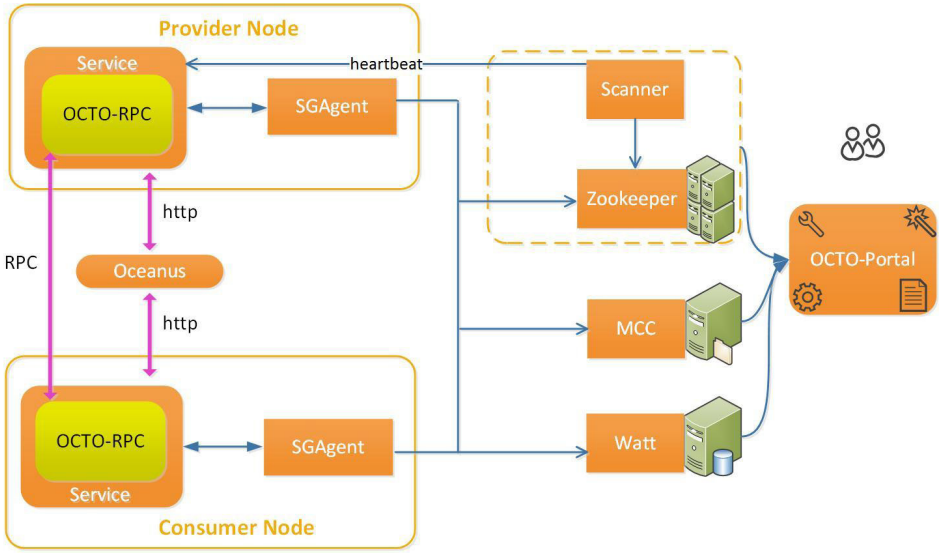
OCTO 主要功能特性包括但不限于：

- 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阀值时，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）

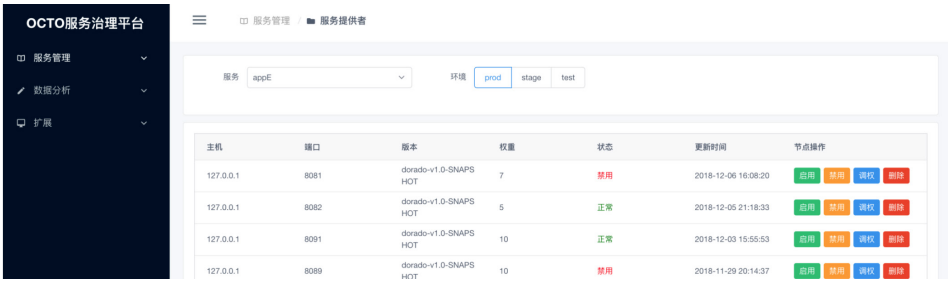
分布式服务通信框架是 OCTO 的重要组成部分，具备高性能、高可用、易扩展、易接入等特点，已覆盖美团 90% 以上的服务（Java/C++），支撑每天千亿级别的调用量。目前 Java 和 C++ 版本已经开源，更多技术实现详见：OCTO-RPC。

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：

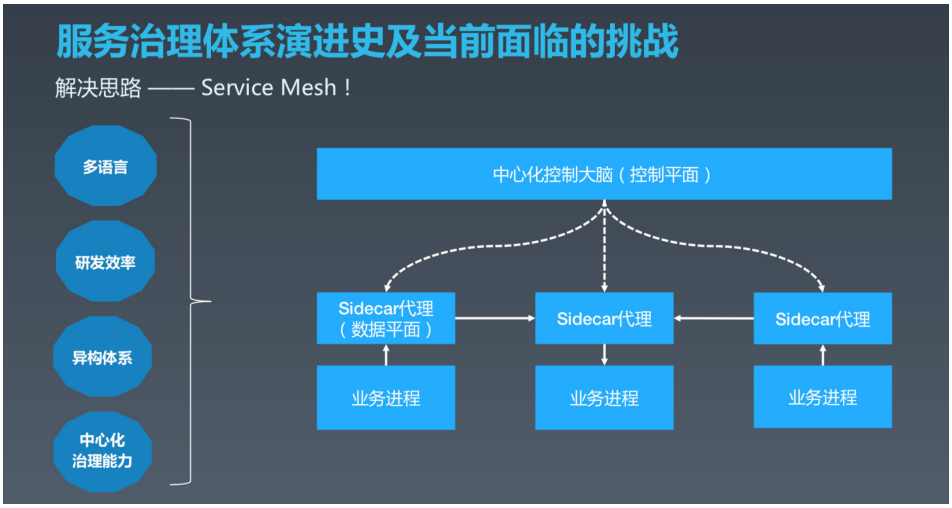
- 已成为美团高度统一的服务治理技术栈，覆盖了公司 90% 的应用，日均调用超万亿次。
- 经历过大规模的技术考验，覆盖数万个服务 / 数十万个节点。
- 协同周边治理生态提供的治理能力较为丰富，包含但不限于 SET 化、链路级复杂路由、全链路压测、鉴权加密、限流熔断等治理能力。
- 一套系统支撑着多元业务，覆盖公司所有事业线。

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。

OCTO-Mesh的技术选型及架构设计

OCTO-Mesh技术选型

子模块	技术方案选型	核心考量点
数据面	基于Envoy二次开发	• 有机会成为数据面标准 • Filter模式及xDS设计扩展性强 • 功能丰富与标准关联性弱
控制面	自研为主	• 兼容存量非容器应用 • 特定容器模式不兼容Istio，Istio api易变 • Istio及Kubernetes的规模限制 • 现有的治理能力比社区产品更丰富、更精细

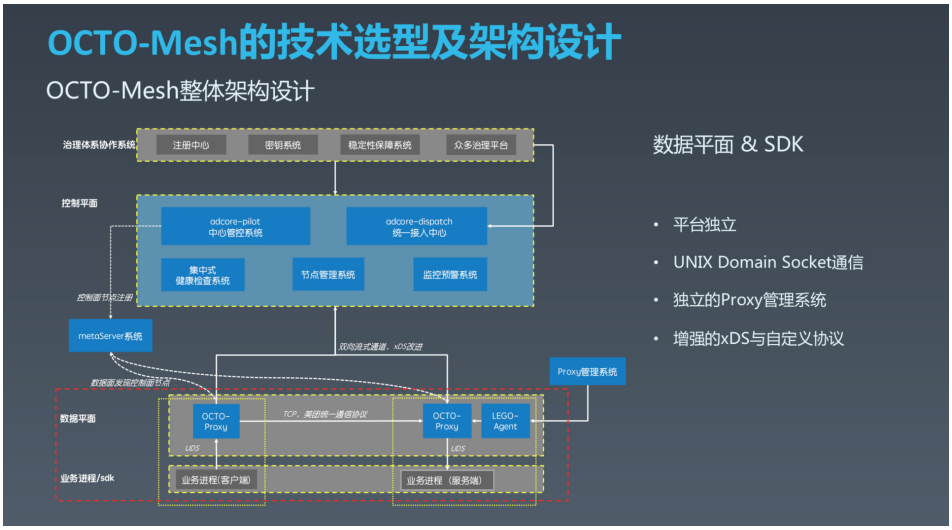
针对上述考量，我们选择的方式是数据面基于 Envoy 二次开发，控制面自研为主。

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。

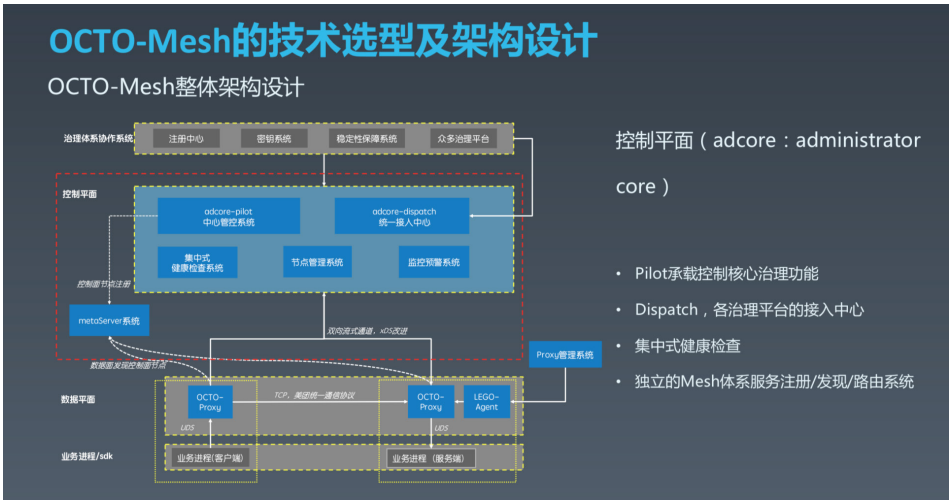
2.2 OCTO Mesh 架构设计



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。

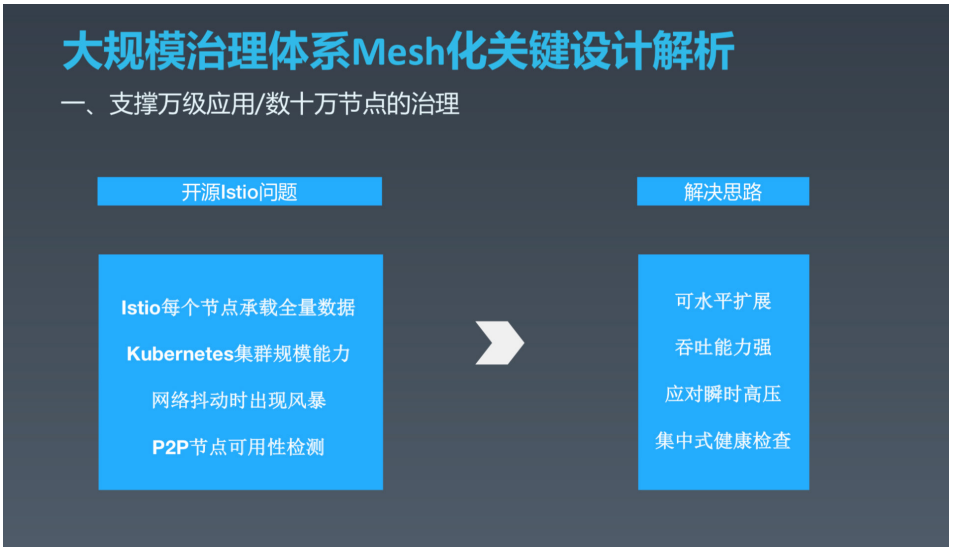
三、关键设计解析

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。

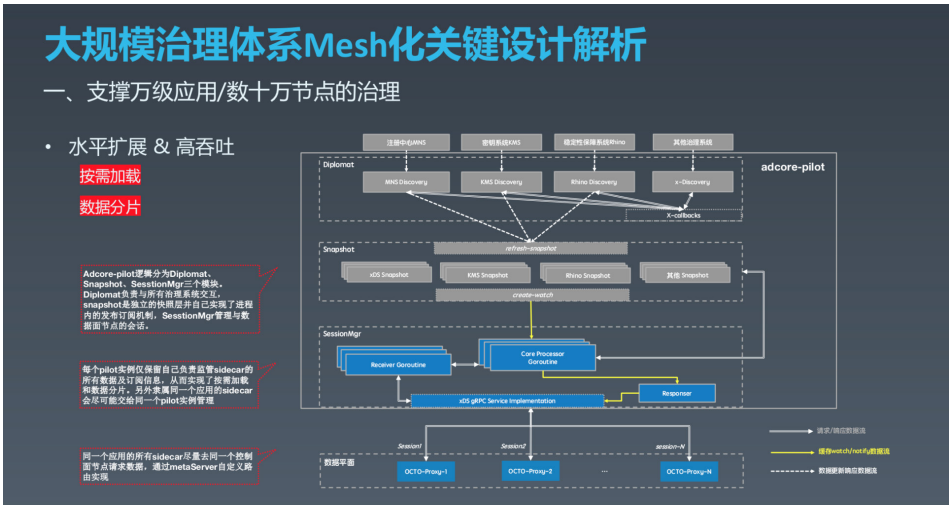
围绕这四点，便可以在系统能力、治理能力、稳定性、运营效率方面支撑美团当前多倍体量的新架构落地。

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：

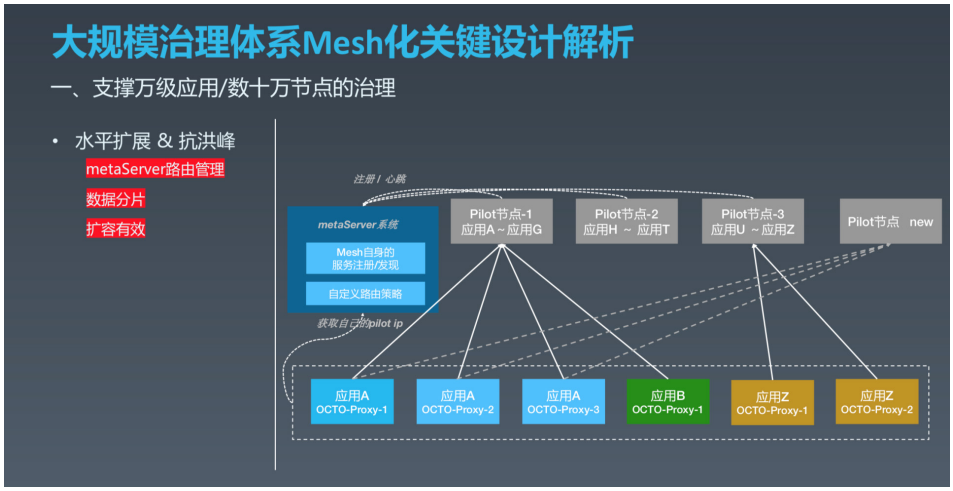
- 控制面每个节点并不承载所有治理数据，系统整体做水平扩展，在此基础上提升每个实例的整体吞吐量和性能。
- 当出现机房断网等异常情况时，可以应对瞬时流量骤增的能力。
- 只做必要的 P2P 模式健康检查，配合集中式健康检查进行百万级节点管理。



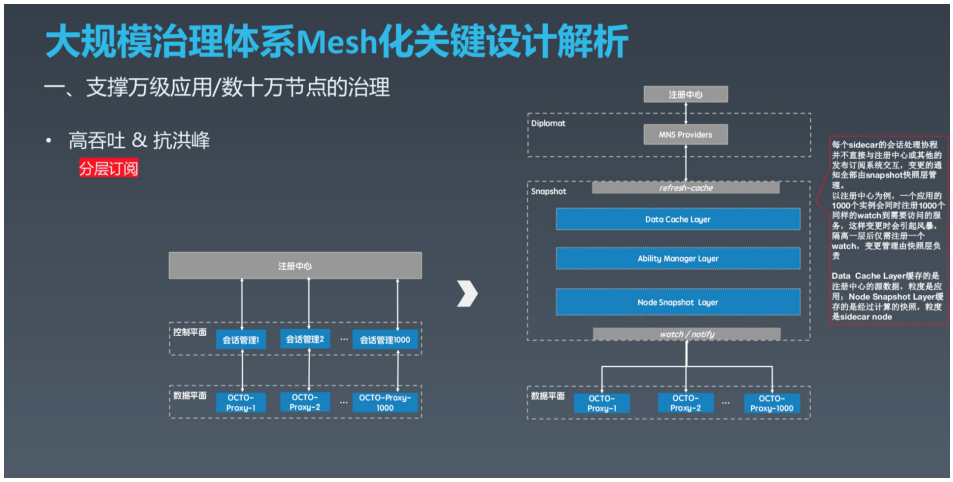
按需加载和数据分片主要由 Adcore Pilot 配合 Meta Server 实现。

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。



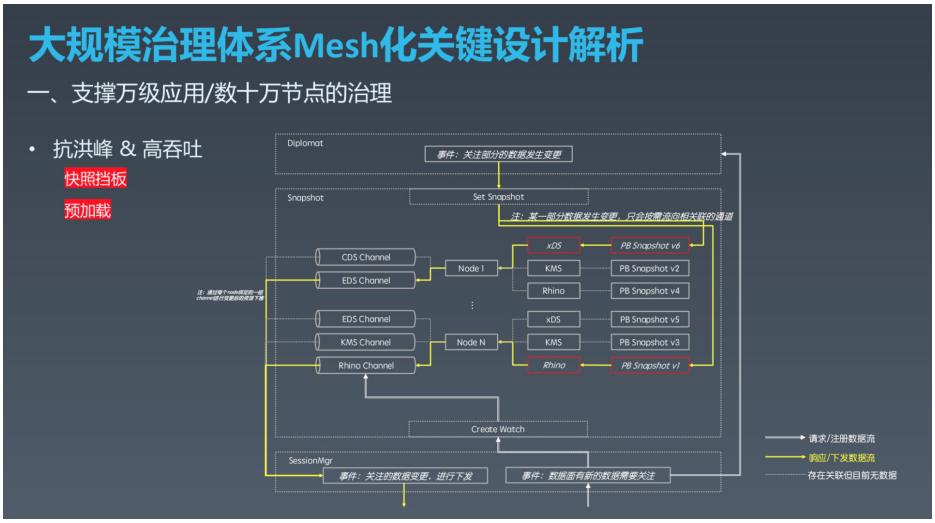
Meta Server 管控每个 Pilot 节点负责应用 OCTO Proxy 的归属关系。当 Pilot 实例启动会注册到 Meta Server，此后定时发送心跳进行续租，长时间心跳异常会自动剔除。在 Meta Server 内部实现了较为复杂的一致性哈希策略，会综合节点的应用、机房、负载等信息进行分组。当一个 Pilot 节点异常或发布时，隶属该 Pilot 的 OCTO Proxy 都会有规律的连接到接替节点，而不会全局随机连接对后端注册中心造成风暴。当异常或发布后的节点恢复后，划分出去的 OCTO Proxy 又会有规则的重新归属当前 Pilot 实例管理。对于关注节点特别多的应用 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。

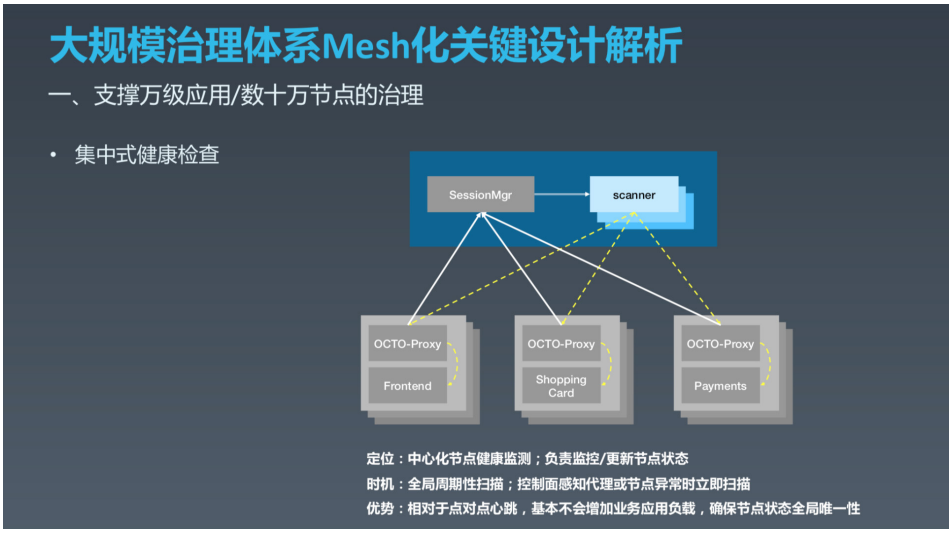
针对这个问题，我们采用分层订阅的方案解决。每个 OCTO Proxy 的会话并不直接与注册中心或其他的发布订阅系统交互，变更的通知全部由 Snapshot 快照层管理。Snapshot 内部又划分为 3 层，Data Cache 层对接并缓存注册中心及其他系统的原始数据，粒度是应用；Node Snapshot 层则是保留经过计算的节点粒度的数据；Ability Manager 层内部会做索引和映射的管理，当注册中心存在节点状态变更时，会通过索引将变更推送给关注变更的 OCTO Proxy。

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。

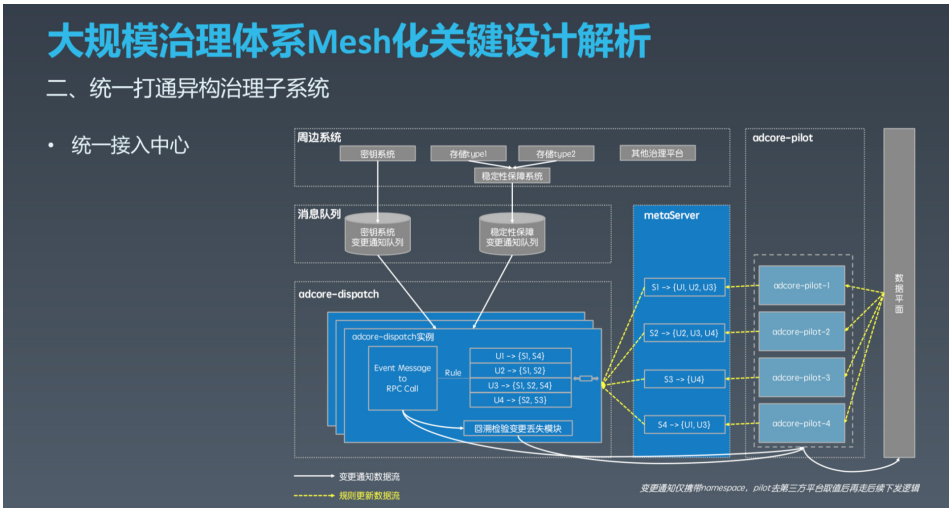
Google 的 Traffic Director 也采用了类似的设计，但大规模使用需要一些技巧：第一个是为了避免机房自治的影响而选择了同机房检测方式，第二个是为了减少中心检测机器因自己 GC 或网络异常造成误判，而采用了 Double Check 的机制。

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。

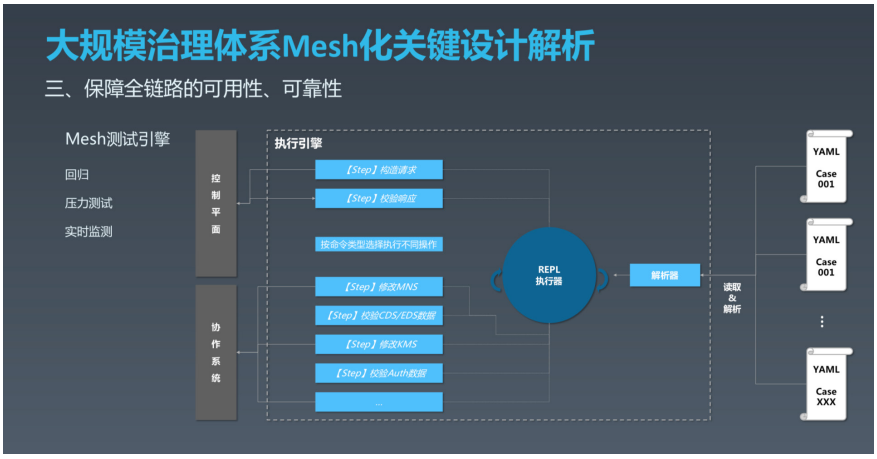
总体而言需要实现三个子目标：打通所有系统，治理能力对齐；快速应对未来新系统的接入；变更发送给关注节点。我们解法是：独立的统一接入中心，屏蔽所有异构系统的存储、发布订阅机制；Meta Server 承担实时分片规则的元数据管理。



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。

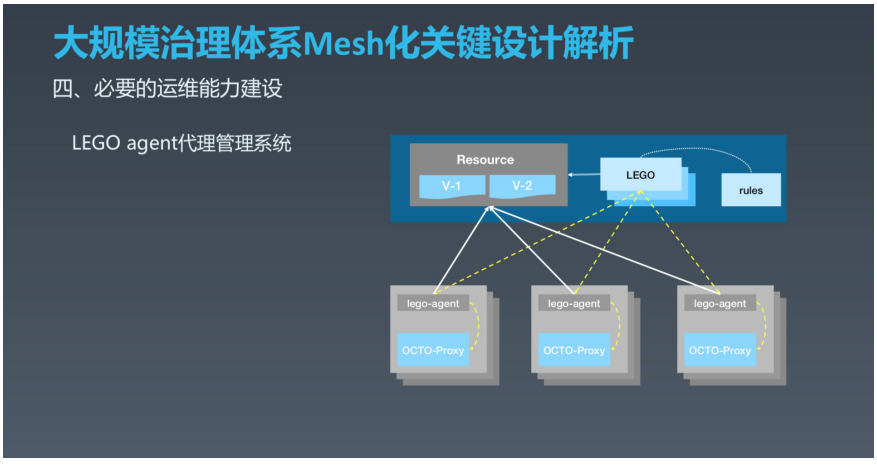
3.3 稳定性保障设计

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?

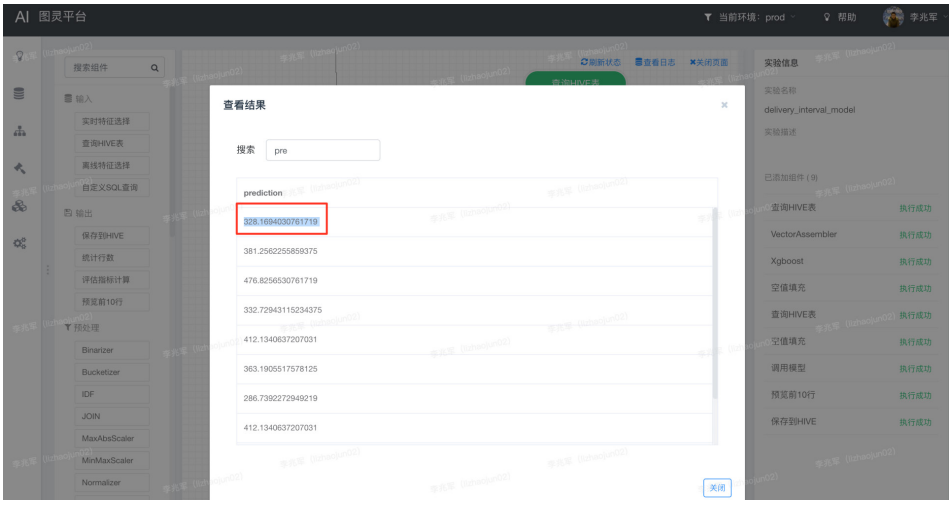
该平台对 XGBoost 模型进行过多次定向优化，在 XGBoost 模型测试时，并没有出现过本地调用（Java 引擎）与平台（Spark 引擎）计算结果不一致的情形。而且平台上运行的版本，和该同学本地使用的版本，都来源于 Dmlc 的官方版本，JNI 底层调用的应该是同一份代码，理论上，结果应该是完全一致的，但实际中却不同。

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);
```

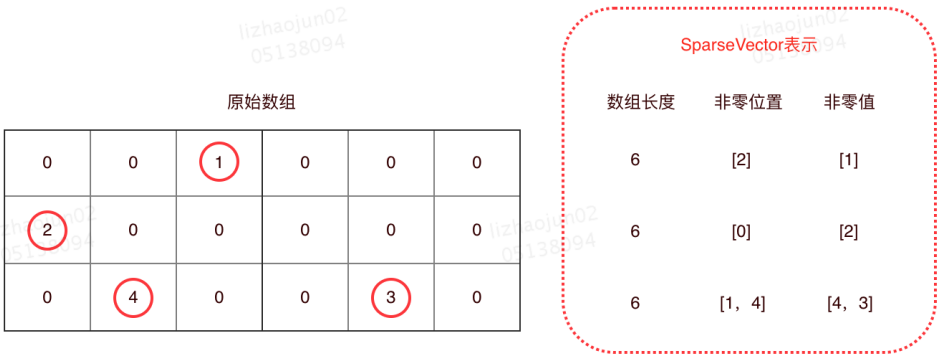
3. XGBoost on Spark 源码中缺失值引入的不稳定问题

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。

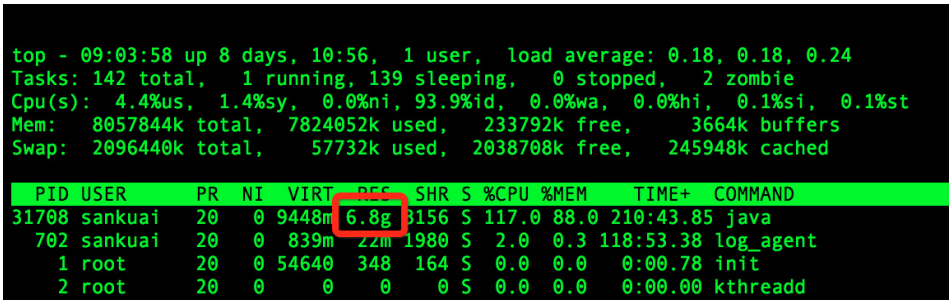
现面向数据工程，数据开发，算法工程，算法应用等领域招聘资深研发工程师 / 技术专家 / 方向负责人（机器学习平台 / 算法平台），欢迎有兴趣的同学一起加入，简历可投递至：tech@meituan.com（注明：美团配送事业部）

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 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：

```
p-gperftools/heap.hprof_21187.0030.heap (2796 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0030.heap
p-gperftools/heap.hprof_21187.0031.heap (2896 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0031.heap
p-gperftools/heap.hprof_21187.0032.heap (2996 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0032.heap
p-gperftools/heap.hprof_21187.0033.heap (3096 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0033.heap
p-gperftools/heap.hprof_21187.0034.heap (9043 MB allocated cumulatively, 640 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0034.heap
p-gperftools/heap.hprof_21187.0035.heap (10068 MB allocated cumulatively, 729 MB currently in use)
uai/heap-gperftools/heap.hprof_21187.0035.heap
p-gperftools/heap.hprof_21187.0036.heap (11106 MB allocated cumulatively, 762 MB currently in use)
uai/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
] 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

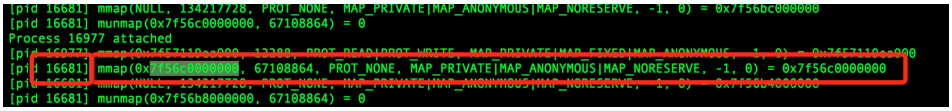
因为使用 strace 没有追踪到可疑内存申请；于是想着看看内存中的情况。就是直接使用命令 gdp –pid pid 进入 GDB 之后，然后使用命令 dump memory mem.bin startAddress endAddressdump 内存，其中 startAddress 和 endAddress 可以从 /proc/pid/smaps 中查找。然后使用 strings mem.bin 查看 dump 的内容，如下：

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

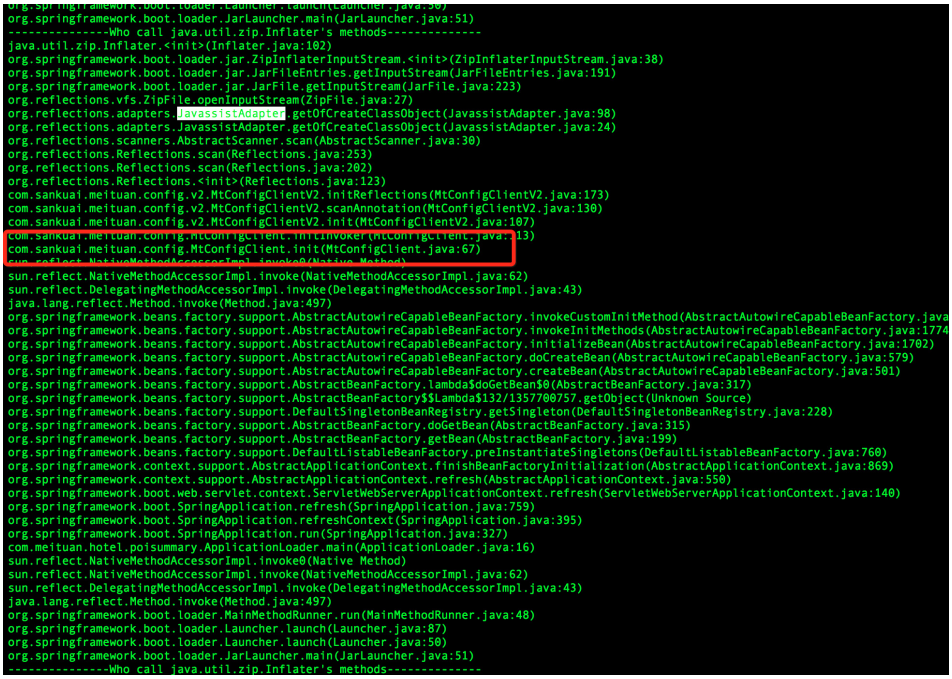
因为 strace 命令中已经显示申请内存的线程 ID。直接使用命令 jstack pid 去查看线程栈，找到对应的线程栈（注意 10 进制和 16 进制转换）如下：

```
"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?

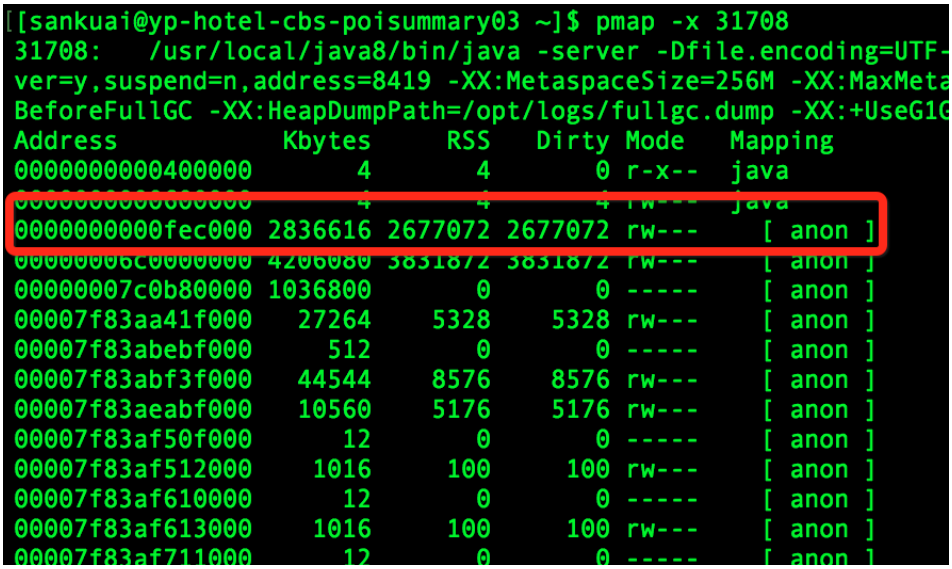
带着疑问，笔者直接看了一下 Spring Boot Loader 那一块的源码。发现 Spring Boot 对 Java JDK 的 InflaterInputStream 进行了包装并且使用了 Inflater，而 Inflater 本身用于解压 JAR 包的需要用到堆外内存。而包装之后的类 ZipInflaterInputStream 没有释放 Inflater 持有的堆外内存。于是笔者以为找到了原因，立马向 Spring Boot 社区反馈了这个 bug。但是反馈之后，笔者就发现 Inflater 这个对象本身实现了 finalize 方法，在这个方法中有调用释放堆外内存的逻辑。也就是说 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：



```
[[sankuai@yp-hotel-cbs-poisummary03 ~]$ pmap -x 31708
31708:   /usr/local/java8/bin/java -server -Dfile.encoding=UTF-
ver=y,suspend=n,address=8419 -XX:MetaspaceSize=256M -XX:MaxMeta
BeforeFullGC -XX:HeapDumpPath=/opt/logs/fullgc.dump -XX:+UseG1G
Address           Kbytes     RSS   Dirty Mode   Mapping
0000000000400000       4       4       0 r-x--  java
0000000000600000       4       4       4 rw---  java
0000000000fec000 2836616 2677072 2677072 rw---    [ anon ]
00000006c0000000 4206080 3831872 3831872 rw---    [ anon ]
00000007c0b80000 1036800       0       0 -----    [ anon ]
00007f83aa41f000   27264    5328    5328 rw---    [ anon ]
00007f83abebf000     512       0       0 -----    [ anon ]
00007f83abf3f000   44544    8576    8576 rw---    [ anon ]
00007f83aeabf000   10560    5176    5176 rw---    [ anon ]
00007f83af50f000      12       0       0 -----    [ anon ]
00007f83af512000    1016     100     100 rw---    [ anon ]
00007f83af610000      12       0       0 -----    [ anon ]
00007f83af613000    1016     100     100 rw---    [ anon ]
00007f83af711000      12       0       0 -----    [ anon ]
```

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

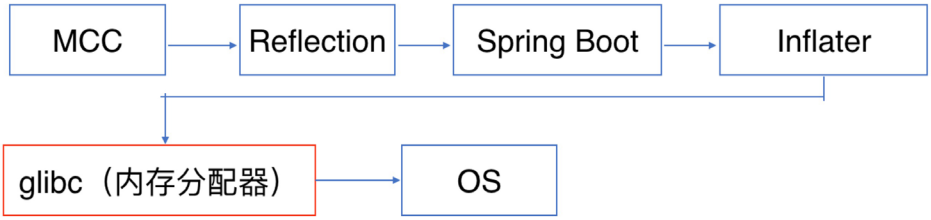
内存测试对比

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。

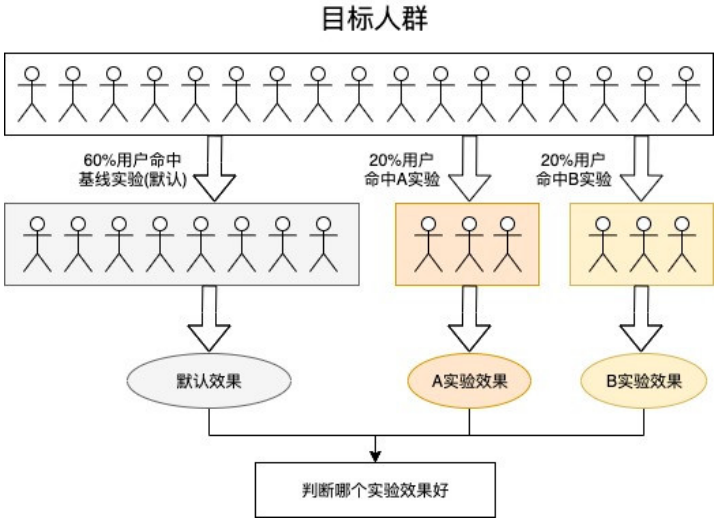


图 1　什么是 AB 实验

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。

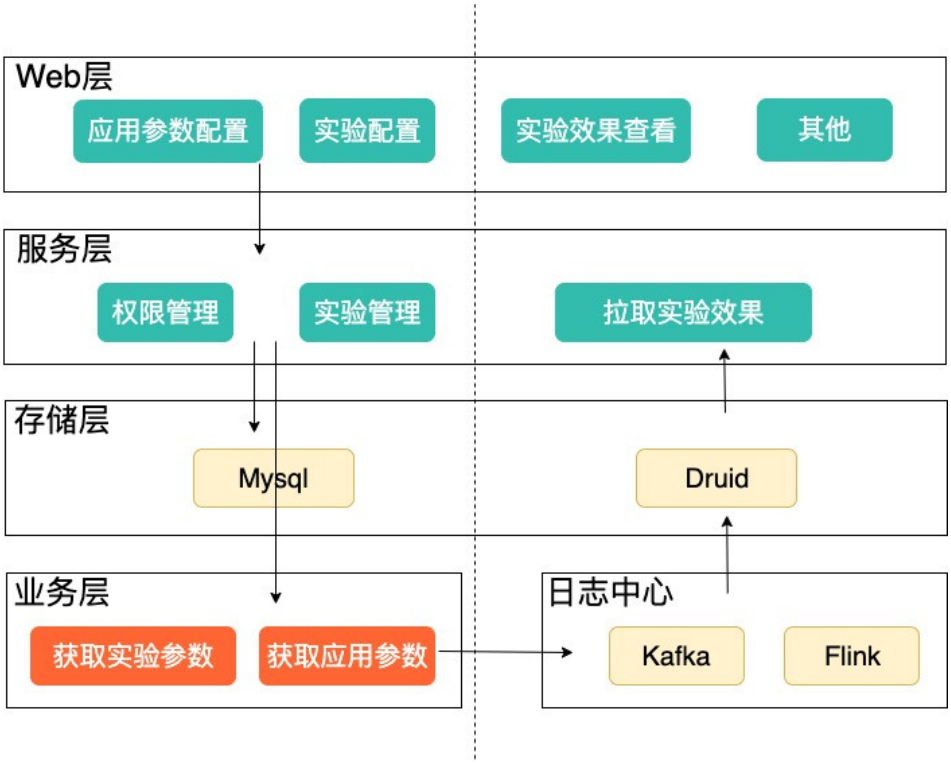


图 2　架构图

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。

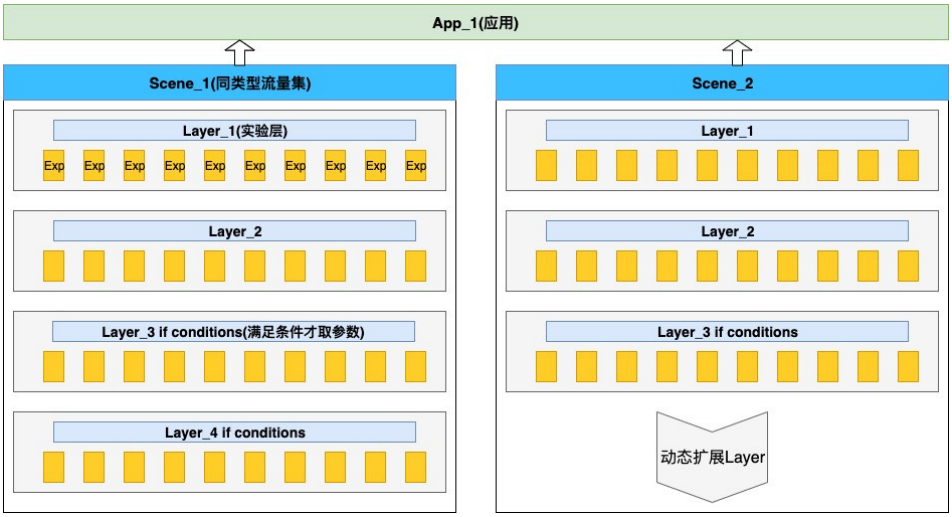


图 3　实验模型

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。

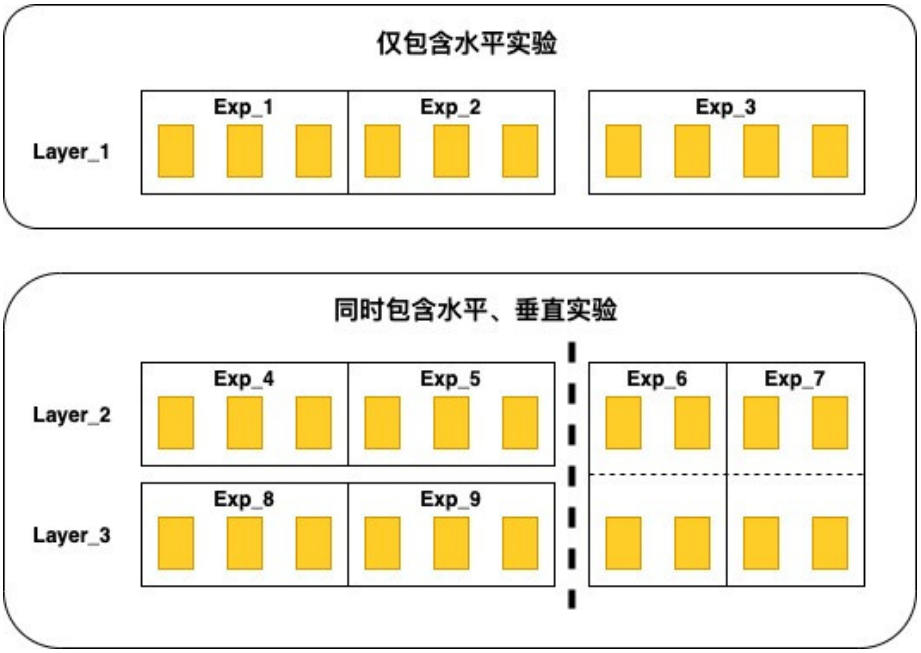


图 4　水平、垂直分流模型

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。

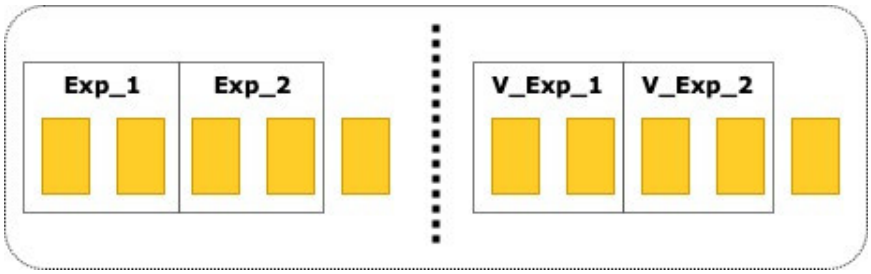


图 5　流量分配示例

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。

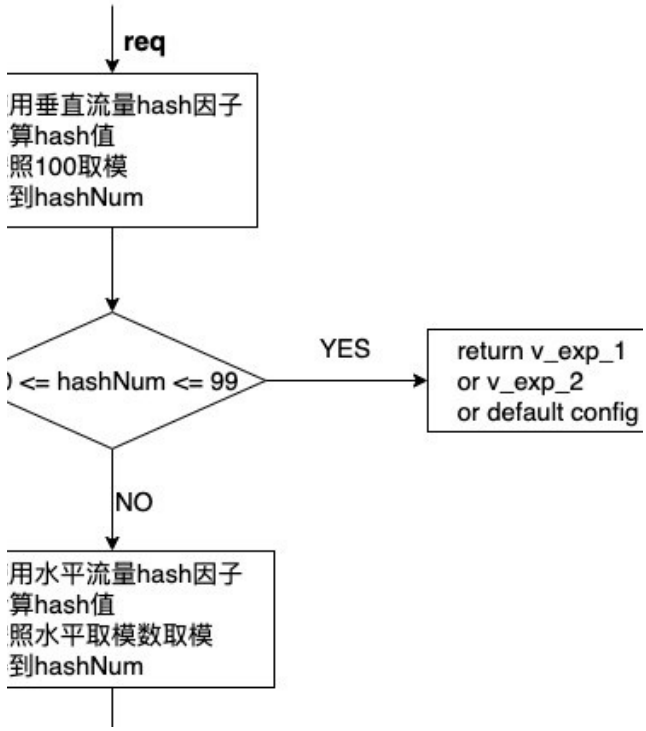


图 6　实验命中流程

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。

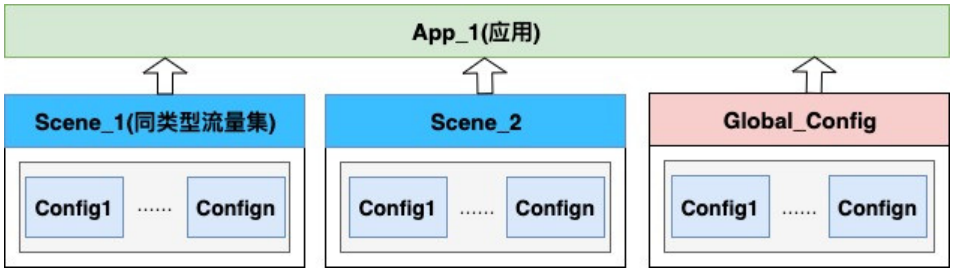


图 7　应用参数模型

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。

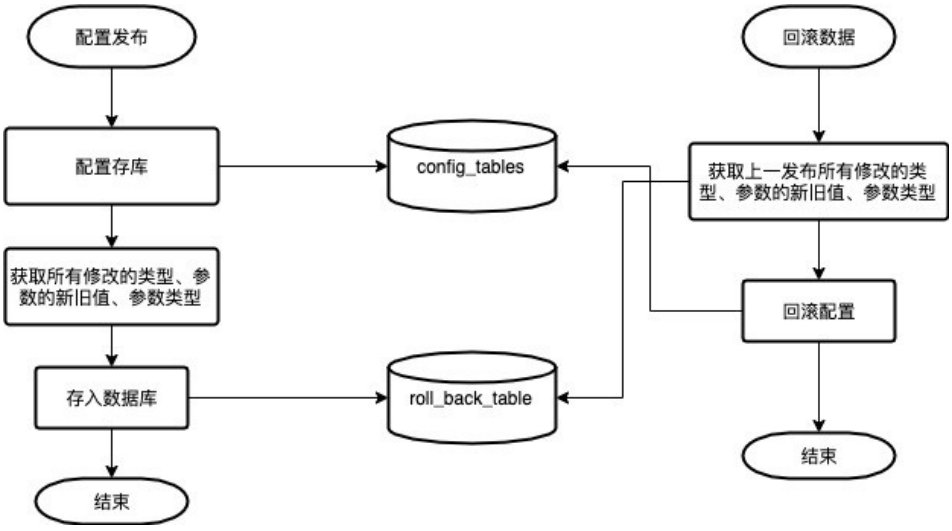


图 8　回滚模型

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。

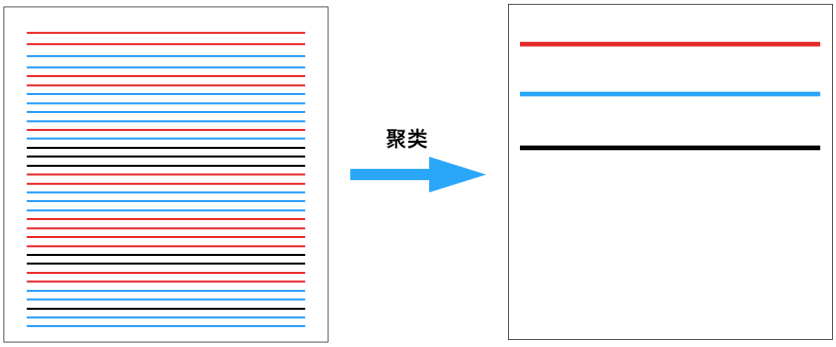


图 1

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。

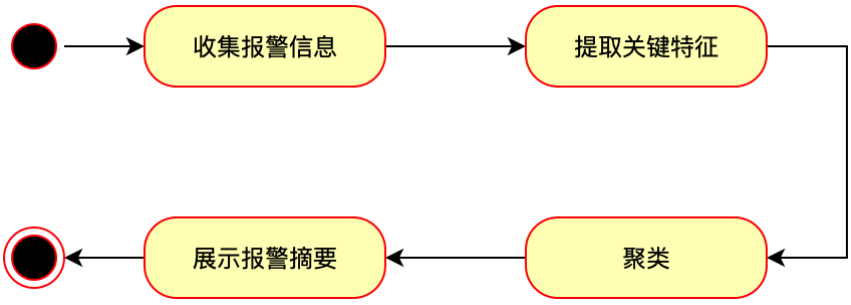


图 2

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：

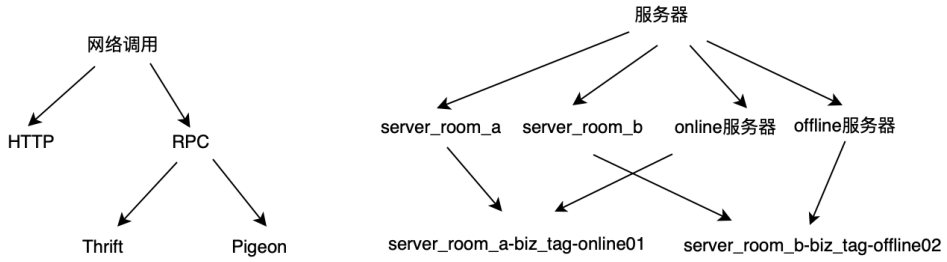


图 3　泛化层次结构示例

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：

$$\bar{d}(\mathbf{g},\mathcal{C}) := 1/|\mathcal{C}| \times \sum_{\mathbf{a}\in\mathcal{C}} d(\mathbf{g},\mathbf{a})$$

$$H(\mathcal{C}) := \min\{\, \bar{d}(\mathbf{g},\mathcal{C}) \,|\, \mathbf{g} \in \mathbf{X}_{i=1}^{n}\mathrm{Dom}(A_i),\ \forall \mathbf{a} \in \mathcal{C} : \mathbf{a} \trianglelefteq \mathbf{g} \,\}$$

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。

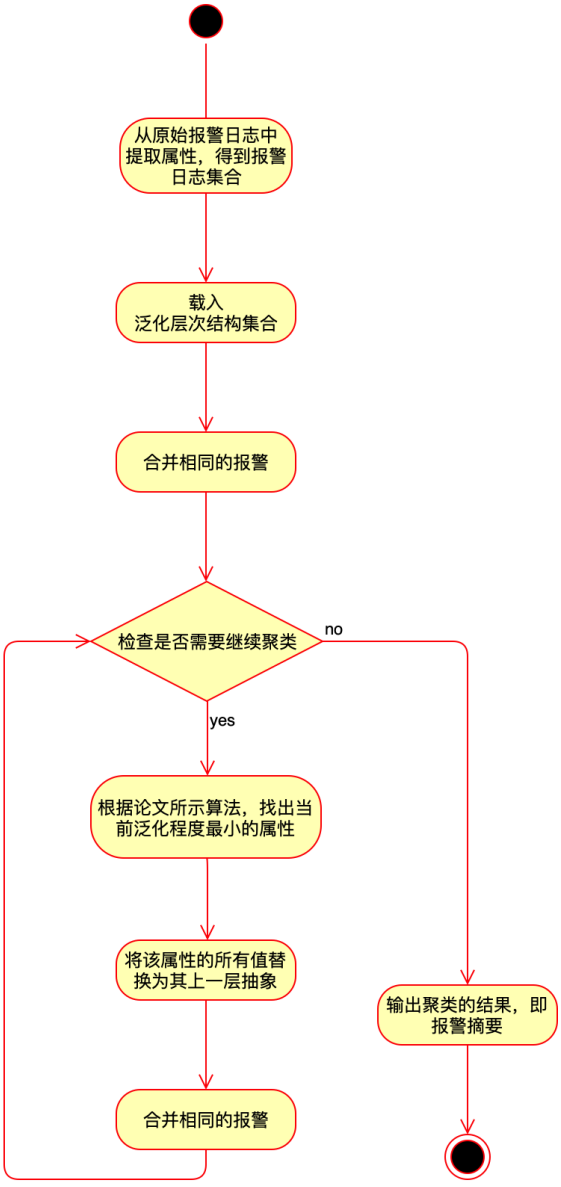


图 4　报警日志聚类流程图

（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：

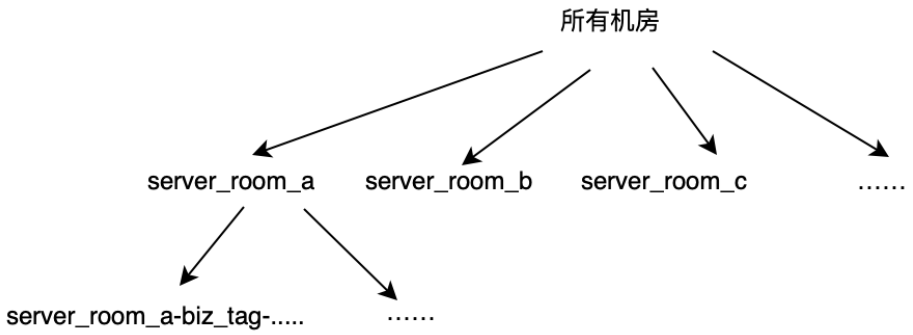


图 5　机房泛化层次结构

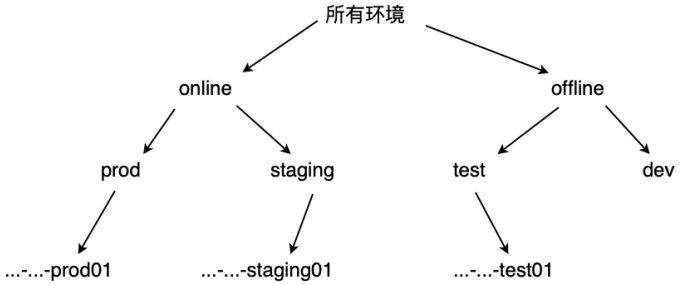


图 6　环境泛化层次结构

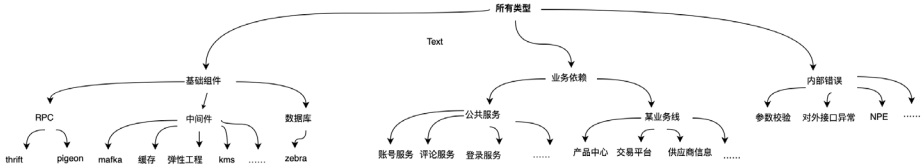


图 7　错误来源泛化层次结构

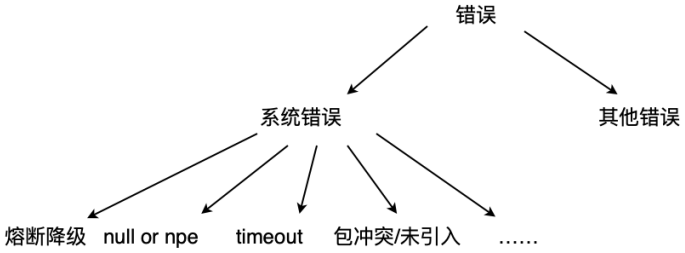


图 8　日志文本摘要泛化层次结构

“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。

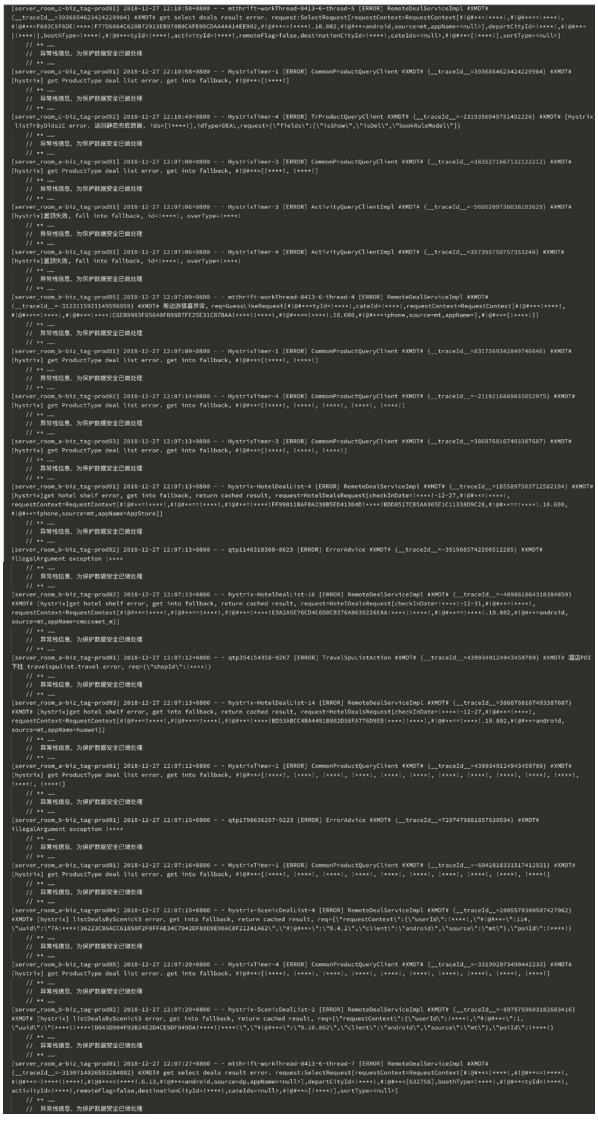


图 9　单依赖故障报警日志节选

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：

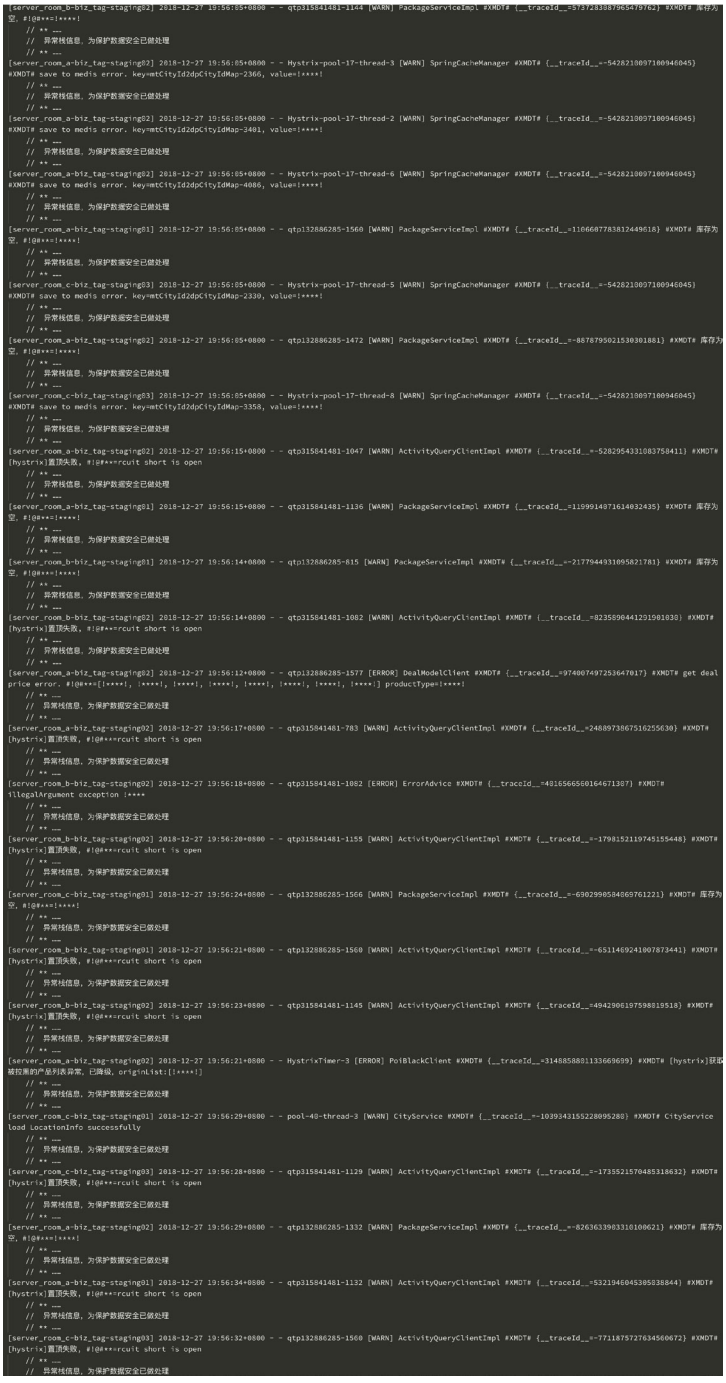


图 10　无相关的多依赖同时故障报警日志节选

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：

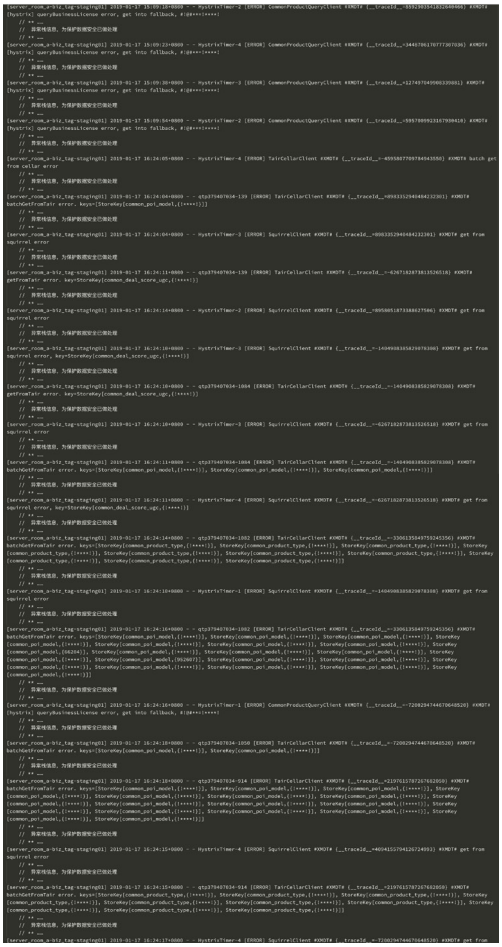


图 11　中间件与相关依赖同时故障报警日志节选

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。
千钊，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：

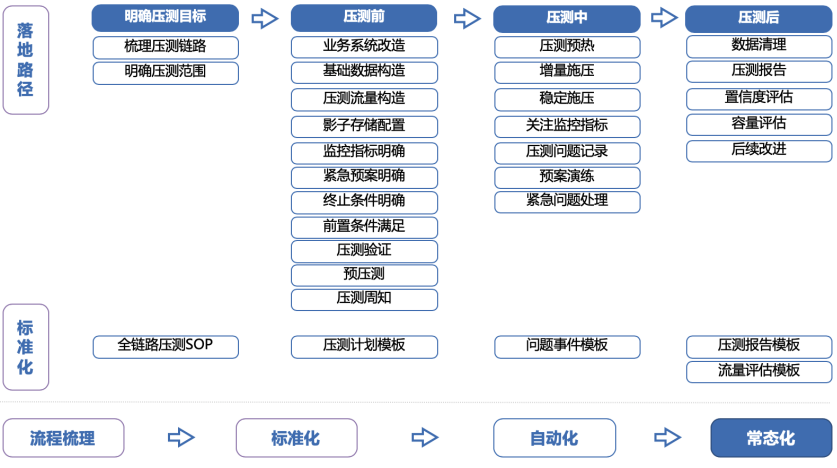


图 1　自动化落地整体思路

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

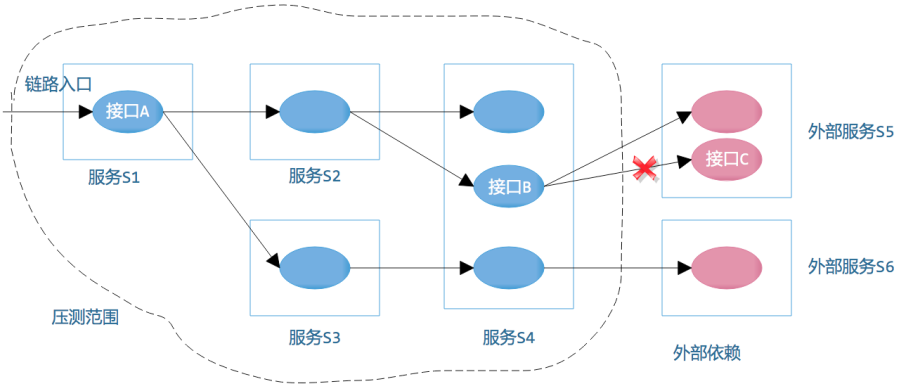


图 4　链路治理示意图

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

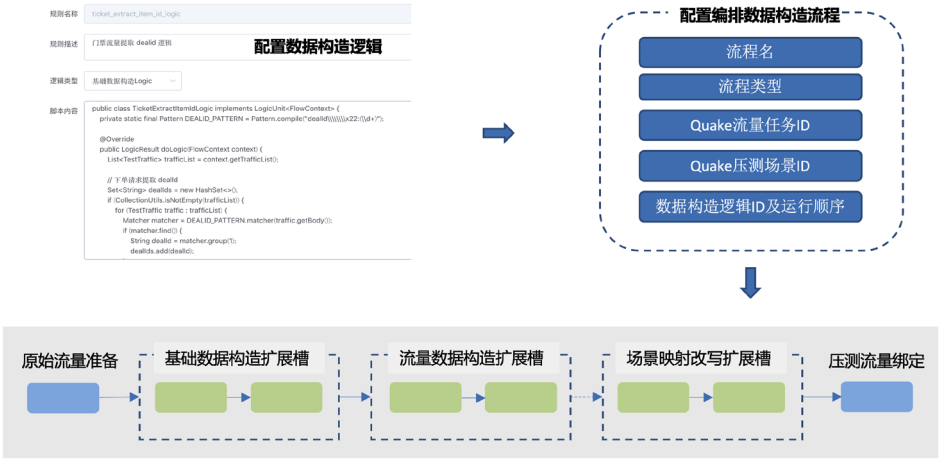


图 6　数据构造

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

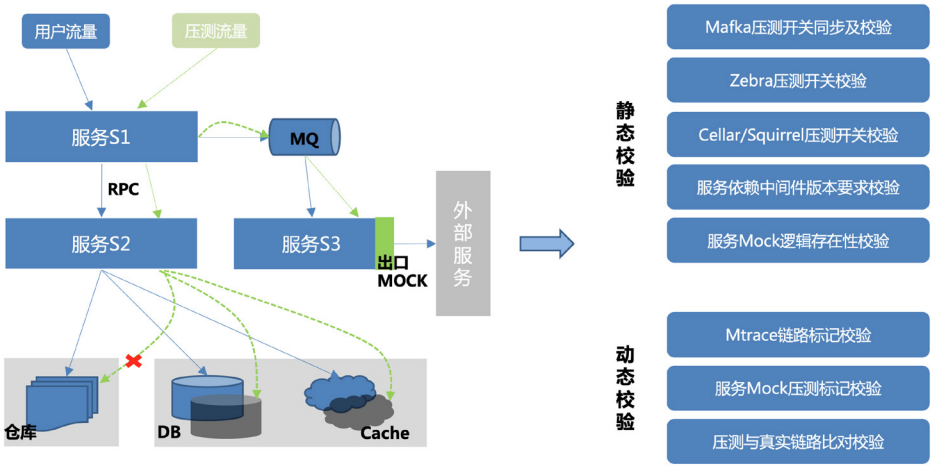


图 7　美团服务压测验证示意

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。

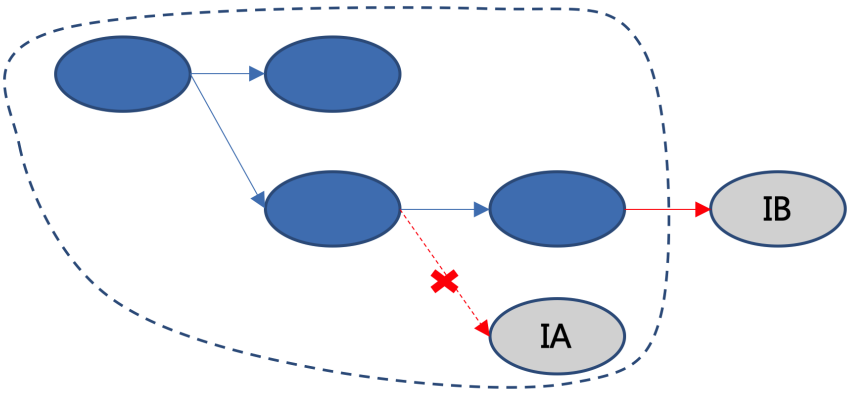


图 8　MTrace 链路标记校验示意

- 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。

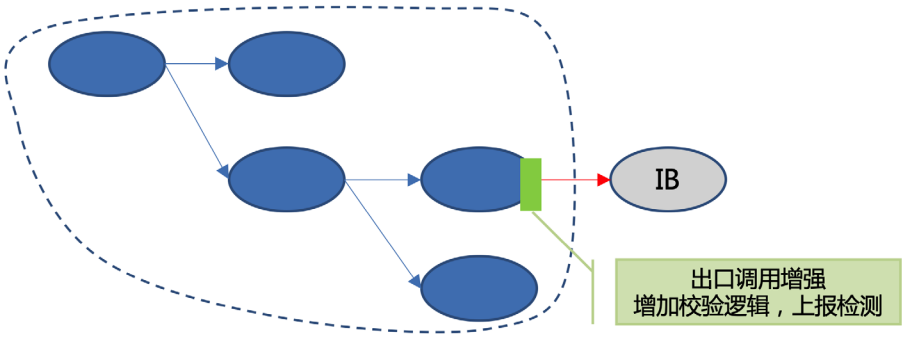


图 9　服务 Mock 压测校验示意

- 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。

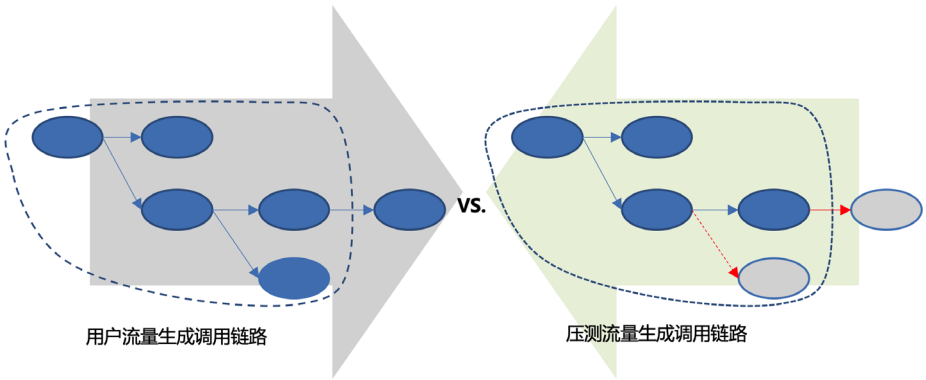


图 10　压测与真实链路对比示意

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。

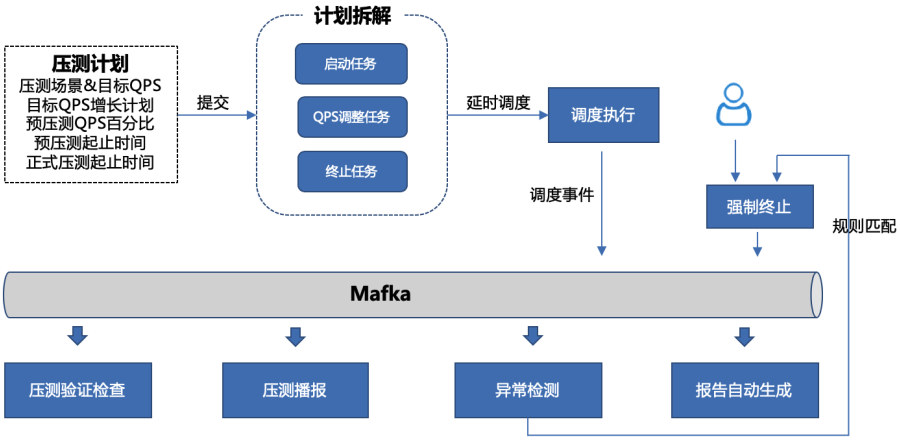


图 11　压测计划执行

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

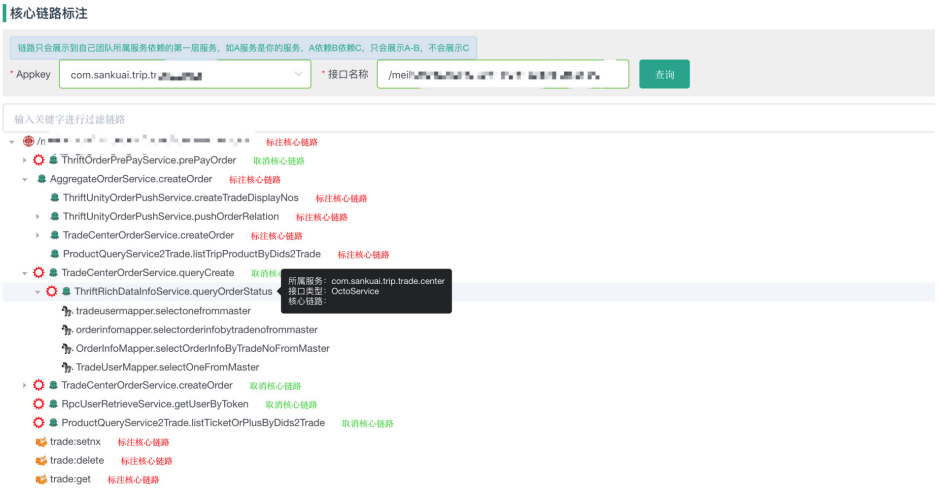
以下以实际压测的过程来做个案例分享。

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。

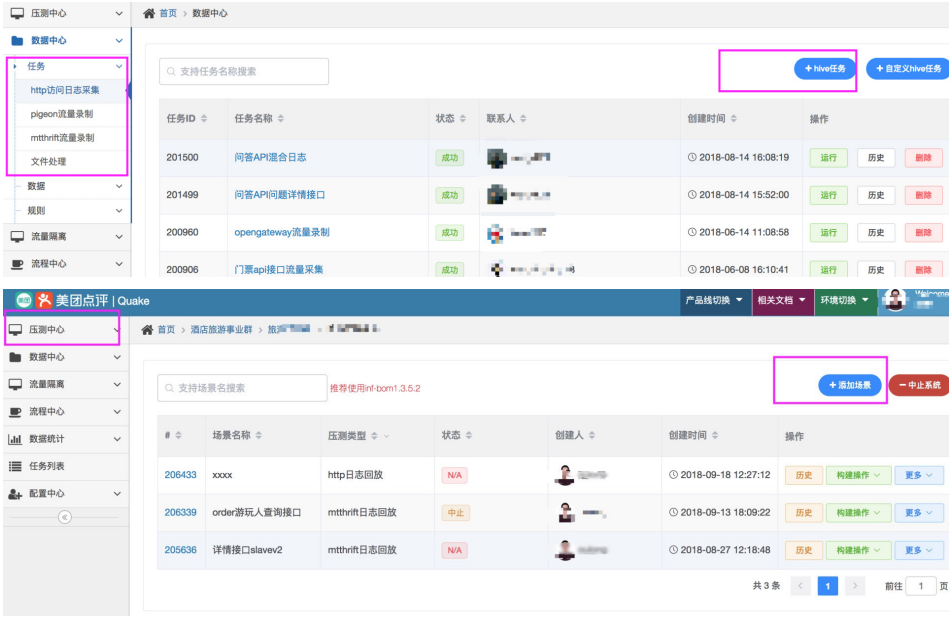


应用改造与压测配置

- 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。

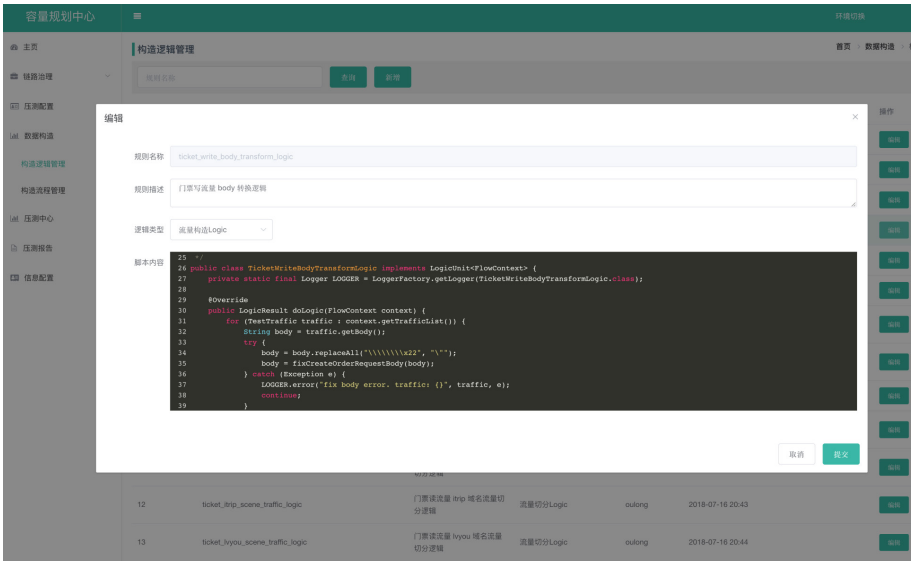
Quake 准备

- 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”。

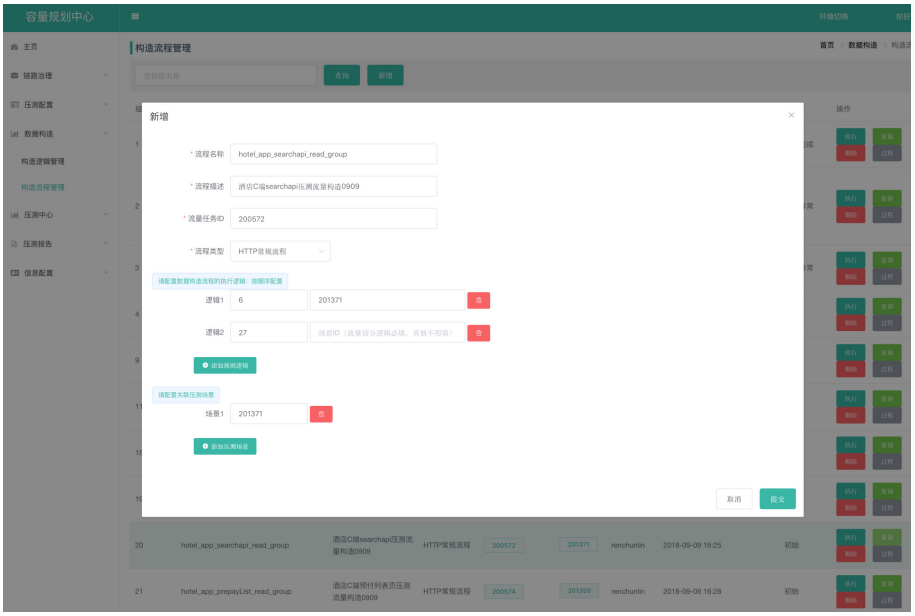


数据构造

- 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。

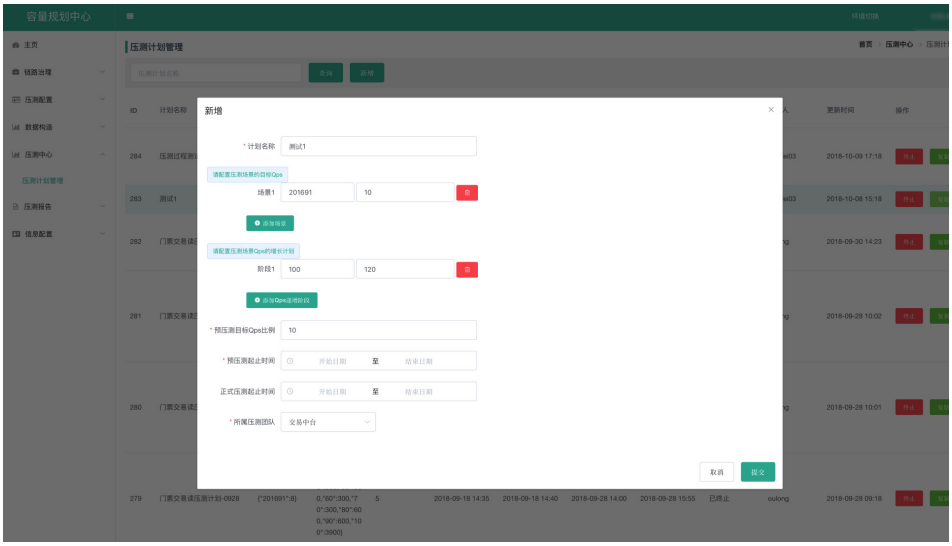


- 配置数据构造流程。

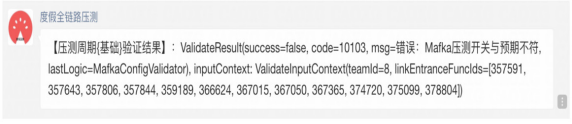


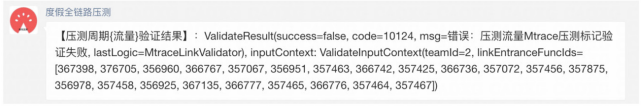
压测实施

- 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。

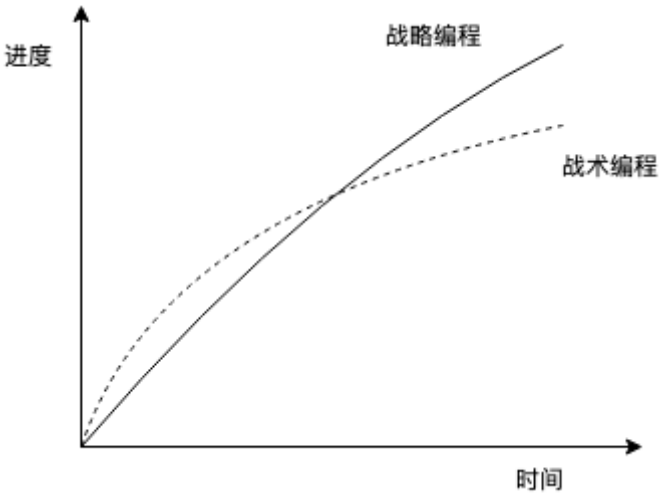
最后，代码能够描述程序的工作流程和结果，却很难描述开发人员的思路，而注释和文档可以。此外，通过注释和文档，开发人员在不阅读实现代码的情况下，就可以理解程序的功能，注释间接促成了代码抽象。好的注释能够帮助解决软件复杂性问题，尤其是认知负担和不可知问题（Unknown Unknowns）。

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。

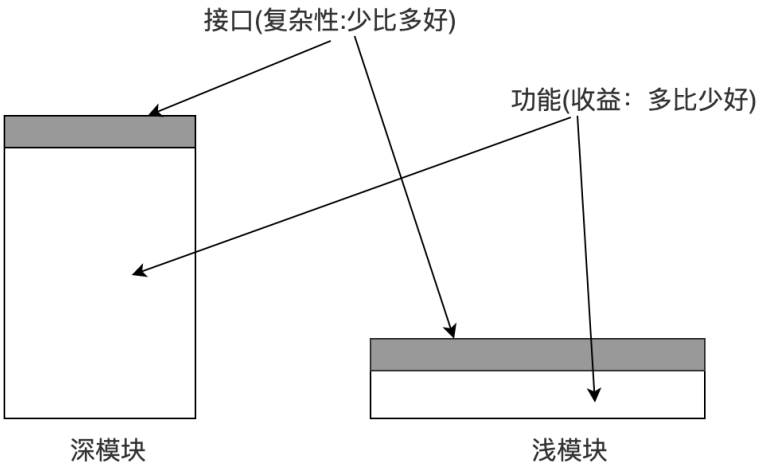
4.2 设计两次

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。

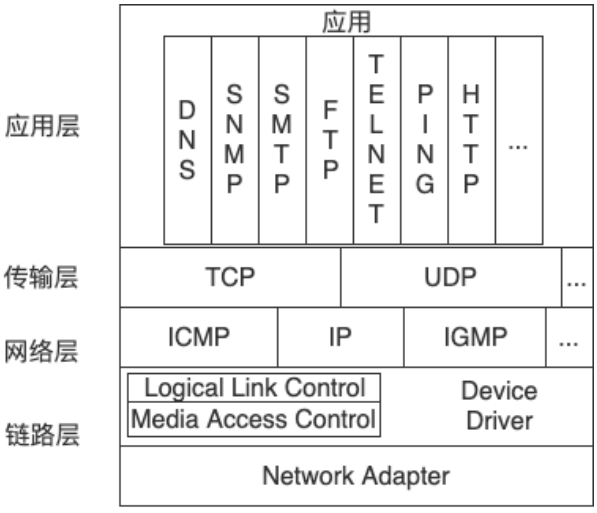
降低复杂度的一个原则就是尽可能减少需要处理异常的可能性。而最佳实践就是确保错误终结，例如删除一个并不存在的文件，与其上报文件不存在的异常，不如什么都不做。确保文件不存在就好了，上层逻辑不但不会被影响，还会因为不需要处理额外的异常而变得简单。

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 FileInputStream、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）。

微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐