2020 美团技术年货

CODE A BETTER LIFE

【后台篇】

美团 美团

目录

后台

Java 线程池实现原理及其在美团业务中的实践

作者：致远　陆晨

随着计算机行业的飞速发展，摩尔定律逐渐失效，多核 CPU 成为主流。使用多线程并行计算逐渐成为开发人员提升服务器性能的基本武器。J.U.C 提供的线程池：ThreadPoolExecutor 类，帮助开发人员管理线程并方便地执行并行任务。了解并合理使用线程池，是一个开发人员必修的基本功。

本文开篇简述线程池概念和用途，接着结合线程池的源码，帮助读者领略线程池的设计思路，最后回归实践，通过案例讲述使用线程池遇到的问题，并给出了一种动态化线程池解决方案。

一、写在前面

1.1 线程池是什么

线程池（Thread Pool）是一种基于池化思想管理线程的工具，经常出现在多线程服务器中，如 MySQL。

线程过多会带来额外的开销，其中包括创建销毁线程的开销、调度线程的开销等等，同时也降低了计算机的整体性能。线程池维护多个线程，等待监督管理者分配可并发执行的任务。这种做法，一方面避免了处理任务时创建销毁线程开销的代价，另一方面避免了线程数量膨胀导致的过分调度问题，保证了对内核的充分利用。

而本文描述线程池是 JDK 中提供的 ThreadPoolExecutor 类。

当然，使用线程池可以带来一系列好处：

- **降低资源消耗**：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
- **提高响应速度**：任务到达时，无需等待线程创建即可立即执行。
- **提高线程的可管理性**：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。
- **提供更多更强大的功能**：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池 ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

1.2 线程池解决的问题是什么

线程池解决的核心问题就是资源管理问题。在并发环境下，系统不能够确定在任意时刻中，有多少任务需要执行，有多少资源需要投入。这种不确定性将带来以下若干问题：

1. 频繁申请 / 销毁资源和调度资源，将带来额外的消耗，可能会非常巨大。
2. 对资源无限申请缺少抑制手段，易引发系统资源耗尽的风险。
3. 系统无法合理管理内部的资源分布，会降低系统的稳定性。

为解决资源分配这个问题，线程池采用了“池化”（Pooling）思想。池化，顾名思义，是为了最大化收益并最小化风险，而将资源统一在一起管理的一种思想。

Pooling is the grouping together of resources (assets, equipment, personnel, effort, etc.) for the purposes of maximizing advantage or minimizing risk to the users. The term is used in finance, computing and equipment management.——wikipedia

“池化”思想不仅仅能应用在计算机领域，在金融、设备、人员管理、工作管理等领域也有相关的应用。

在计算机领域中的表现为：统一管理 IT 资源，包括服务器、存储、和网络资源等等。通过共享资源，使用户在低投入中获益。除去线程池，还有其他比较典型的几种使用策略包括：

1. 内存池 (Memory Pooling)：预先申请内存，提升申请内存速度，减少内存碎片。
2. 连接池 (Connection Pooling)：预先申请数据库连接，提升申请连接的速度，降低系统的开销。
3. 实例池 (Object Pooling)：循环使用对象，减少资源在初始化和释放时的昂贵损耗。

在了解完“是什么”和“为什么”之后，下面我们来一起深入一下线程池的内部实现原理。

二、线程池核心设计与实现

在前文中，我们了解到：线程池是一种通过“池化”思想，帮助我们管理线程而获取并发性的工具，在 Java 中的体现是 ThreadPoolExecutor 类。那么它的的详细设计与实现是什么样的呢？我们会在本章进行详细介绍。

2.1 总体设计

Java 中的线程池核心实现类是 ThreadPoolExecutor，本章基于 JDK 1.8 的源码来分析 Java 线程池的核心设计与实现。我们首先来看一下 ThreadPoolExecutor 的 UML 类图，了解下 ThreadPoolExecutor 的继承关系。

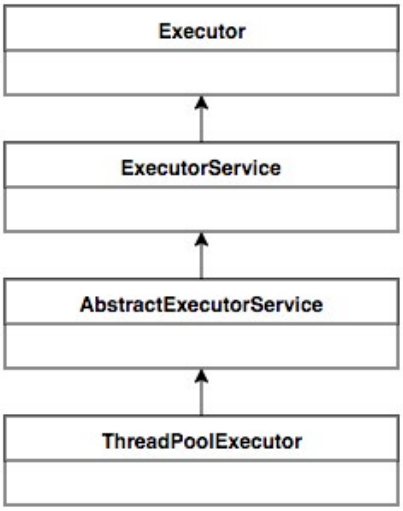


图 1　ThreadPoolExecutor UML 类图

ThreadPoolExecutor 实现的顶层接口是 Executor，顶层接口 Executor 提供了一种思想：将任务提交和任务执行进行解耦。用户无需关注如何创建线程，如何调度线程来执行任务，用户只需提供 Runnable 对象，将任务的运行逻辑提交到执行器 (Executor) 中，由 Executor 框架完成线程的调配和任务的执行部分。ExecutorService 接口增加了一些能力：(1) 扩充执行任务的能力，补充可以为一个或一批异步任务生成 Future 的方法；(2) 提供了管控线程池的方法，比如停止线程池的运行。AbstractExecutorService 则是上层的抽象类，将执行任务的流程串联了起来，保证下层的实现只需关注一个执行任务的方法即可。最下层的实现类 ThreadPoolExecutor 实现最复杂的运行部分，ThreadPoolExecutor 将会一方面维护自身的生命周期，另一方面同时管理线程和任务，使两者良好的结合从而执行并行任务。

ThreadPoolExecutor 是如何运行，如何同时维护线程和执行任务的呢？其运行机制如下图所示：

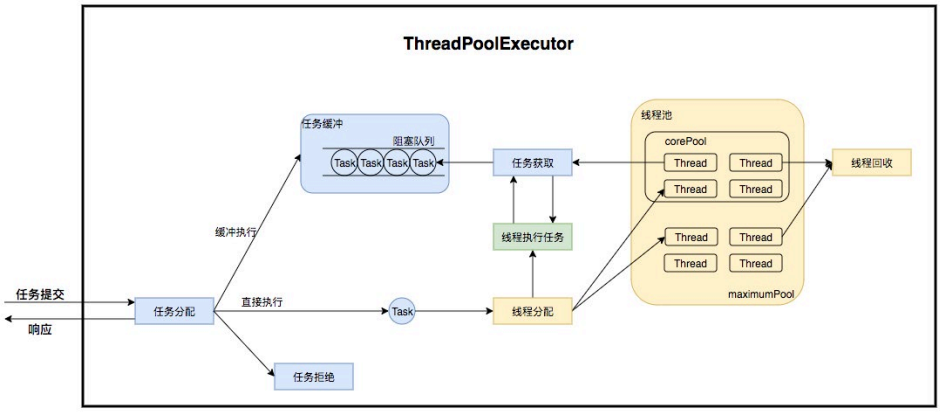


图 2　ThreadPoolExecutor 运行流程

线程池在内部实际上构建了一个生产者消费者模型，将线程和任务两者解耦，并不直接关联，从而良好的缓冲任务，复用线程。线程池的运行主要分成两部分：任务管理、线程管理。任务管理部分充当生产者的角色，当任务提交后，线程池会判断该任务后续的流转：(1) 直接申请线程执行该任务；(2) 缓冲到队列中等待线程执行；(3) 拒绝该任务。线程管理部分是消费者，它们被统一维护在线程池内，根据任务请求进行线程的分配，当线程执行完任务后则会继续获取新的任务去执行，最终当线程获取不到任务的时候，线程就会被回收。

接下来，我们会按照以下三个部分去详细讲解线程池运行机制：

1. 线程池如何维护自身状态。
2. 线程池如何管理任务。
3. 线程池如何管理线程。

2.2 生命周期管理

线程池运行的状态，并不是用户显式设置的，而是伴随着线程池的运行，由内部来维护。线程池内部使用一个变量维护两个值：运行状态 (runState) 和线程数量 (workerCount)。在具体实现中，线程池将运行状态 (runState)、线程数量 (workerCount) 两个关键参数的维护放在了一起，如下代码所示：

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

`ctl` 这个 AtomicInteger 类型，是对线程池的运行状态和线程池中有效线程的数量进行控制的一个字段，它同时包含两部分的信息：线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)，高 3 位保存 runState，低 29 位保存 workerCount，两个变量之间互不干扰。用一个变量去存储两个值，可避免在做相关决策时，出现不一致的情况，不必为了维护两者的一致，而占用锁资源。通过阅读线程池源代码也可以发现，经常出现要同时判断线程池运行状态和线程数量的情况。线程池也提供了若干方法去供用户获得线程池当前的运行状态、线程个数。这里都使用的是位运算的方式，相比于基本运算，速度也会快很多。

关于内部封装的获取生命周期状态、获取线程池线程数量的计算方法如以下代码所示：

```
private static int runStateOf(int c)     { return c & ~CAPACITY; } //计算当前运行状态
private static int workerCountOf(int c)  { return c & CAPACITY; }  //计算当前线程数量
private static int ctlOf(int rs, int wc) { return rs | wc; }   //通过状态和线程数生成 ctl
```

ThreadPoolExecutor 的运行状态有 5 种，分别为：

运行状态	状态描述
RUNNING	能接受新提交的任务，并且也能处理阻塞队列中的任务。
SHUTDOWN	关闭状态，不再接受新提交的任务，但却可以继续处理阻塞队列中已保存的任务。
STOP	不能接受新任务，也不处理队列中的任务，会中断正在处理任务的线程。
TIDYING	所有的任务都已终止了，workerCount (有效线程数) 为0。
TERMINATED	在terminated() 方法执行完后进入该状态。

其生命周期转换如下入所示：

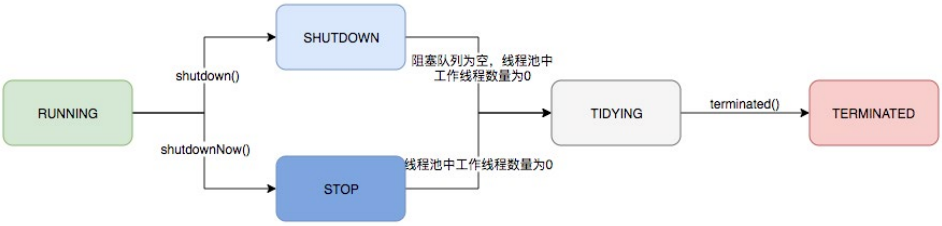


图 3　线程池生命周期

2.3 任务执行机制

2.3.1 任务调度

任务调度是线程池的主要入口，当用户提交了一个任务，接下来这个任务将如何执行都是由这个阶段决定的。了解这部分就相当于了解了线程池的核心运行机制。

首先，所有任务的调度都是由 execute 方法完成的，这部分完成的工作是：检查现在线程池的运行状态、运行线程数、运行策略，决定接下来执行的流程，是直接申请线程执行，或是缓冲到队列中执行，亦或是直接拒绝该任务。其执行过程如下：

1. 首先检测线程池运行状态，如果不是 RUNNING，则直接拒绝，线程池要保证在 RUNNING 的状态下执行任务。
2. 如果 workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务。
3. 如果 workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中。
4. 如 果 workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务。
5. 如果 workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。

其执行流程如下图所示：

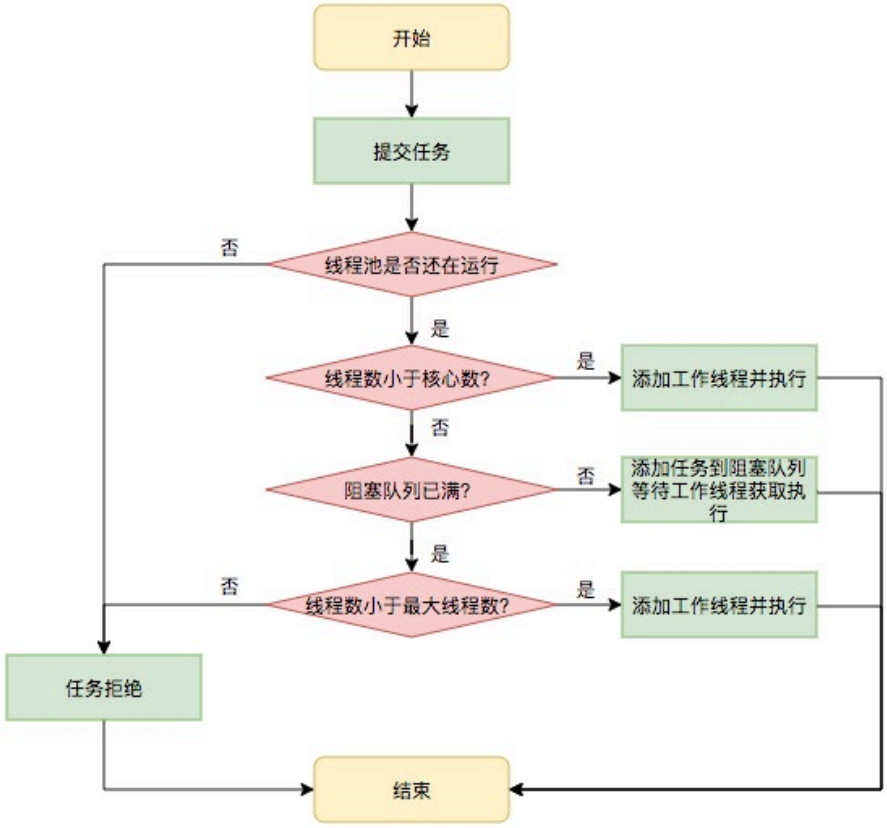


图 4　任务调度流程

2.3.2 任务缓冲

任务缓冲模块是线程池能够管理任务的核心部分。线程池的本质是对任务和线程的管理，而做到这一点最关键的思想就是将任务和线程两者解耦，不让两者直接关联，才可以做后续的分配工作。线程池中是以生产者消费者模式，通过一个阻塞队列来实现的。阻塞队列缓存任务，工作线程从阻塞队列中获取任务。

阻塞队列 (BlockingQueue) 是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，

而消费者也只从容器里拿元素。

下图中展示了线程 1 往阻塞队列中添加元素，而线程 2 从阻塞队列中移除元素：

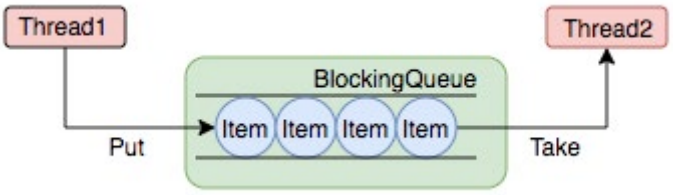


图 5　阻塞队列

使用不同的队列可以实现不一样的任务存取策略。在这里，我们可以再介绍下阻塞队列的成员：

名称	描述
ArrayBlockingQueue	一个用数组实现的有界阻塞队列，此队列按照先进先出(FIFO)的原则对元素进行排序。支持公平锁和非公平锁。
LinkedBlockingQueue	一个由链表结构组成的有界队列，此队列按照先进先出(FIFO)的原则对元素进行排序。此队列的默认长度为Integer.MAX_VALUE，所以默认创建的该队列有容量危险。
PriorityBlockingQueue	一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序。
DelayQueue	一个实现PriorityBlockingQueue实现延迟获取的无界队列，在创建元素时，可以指定多久才能从队列中获取当前元素。只有延时期满后才能从队列中获取元素。
SynchronousQueue	一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素。支持公平锁和非公平锁。SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。
LinkedTransferQueue	一个由链表结构组成的无界阻塞队列，相当于其它队列，LinkedTransferQueue队列多了transfer和tryTransfer方法。
LinkedBlockingDeque	一个由链表结构组成的双向阻塞队列。队列头部和尾部都可以添加和移除元素，多线程并发时，可以将锁的竞争最多降到一半。

2.3.3 任务申请

由上文的任务分配部分可知，任务的执行有两种可能：一种是任务直接由新创建的线程执行。另一种是线程从任务队列中获取任务然后执行，执行完任务的空闲线程会再次去从队列中申请任务再去执行。第一种情况仅出现在线程初始创建的时候，第二种是线程获取任务绝大多数的情况。

线程需要从任务缓存模块中不断地取任务执行，帮助线程从阻塞队列中获取任务，实

现线程管理模块和任务管理模块之间的通信。这部分策略由 getTask 方法实现，其执行流程如下图所示：

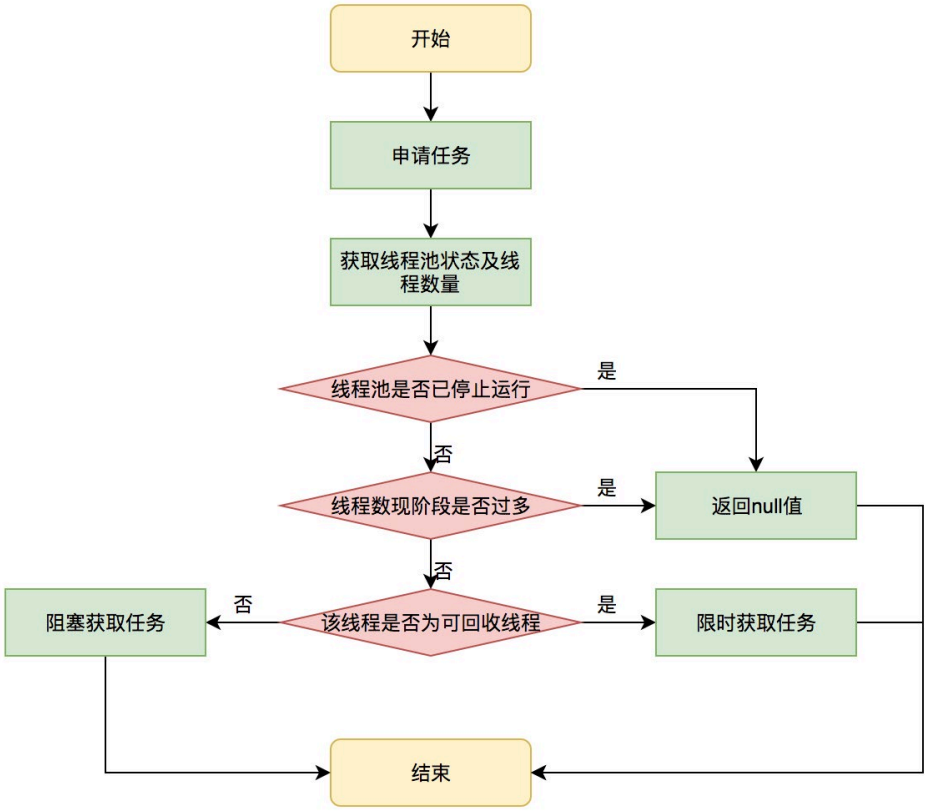


图 6　获取任务流程图

getTask 这部分进行了多次判断，为的是控制线程的数量，使其符合线程池的状态。如果线程池现在不应该持有那么多线程，则会返回 null 值。工作线程 Worker 会不断接收新任务去执行，而当工作线程 Worker 接收不到任务的时候，就会开始被回收。

2.3.4 任务拒绝

任务拒绝模块是线程池的保护部分，线程池有一个最大的容量，当线程池的任务缓存队列已满，并且线程池中的线程数目达到 maximumPoolSize 时，就需要拒绝掉该

任务，采取任务拒绝策略，保护线程池。

拒绝策略是一个接口，其设计如下：

```
public interface RejectedExecutionHandler {
    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

用户可以通过实现这个接口去定制拒绝策略，也可以选择 JDK 提供的四种已有拒绝策略，其特点如下：

	名称	描述
1	ThreadPoolExecutor.AbortPolicy	丢弃任务并抛出RejectedExecutionException异常。 这是线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐使用此拒绝策略，这样子在系统不能承载更大的并发量的时候，能够及时的通过异常发现。
2	ThreadPoolExecutor.DiscardPolicy	丢弃任务，但是不抛出异常。 使用此策略，可能会使我们无法发现系统的异常状态。建议是一些无关紧要的业务采用此策略。
3	ThreadPoolExecutor.DiscardOldestPolicy	丢弃队列最前面的任务，然后重新提交被拒绝的任务。是否要采用此种拒绝策略，还得根据实际业务是否允许丢弃老任务来认真衡量。
4	ThreadPoolExecutor.CallerRunsPolicy	由调用线程（提交任务的线程）处理该任务。这种情况是需要让所有任务都执行完毕，那么就适合大量计算的任务类型去执行，多线程仅仅是增大吞吐量的手段，最终必须要让每个任务都执行完毕。

2.4 Worker 线程管理

2.4.1 Worker 线程

线程池为了掌握线程的状态并维护线程的生命周期，设计了线程池内的工作线程 Worker。我们来看一下它的部分代码：

```
private final class Worker extends AbstractQueuedSynchronizer
implements Runnable{
    final Thread thread;//Worker 持有的线程
    Runnable firstTask;// 初始化的任务，可以为 null
}
```

Worker 这个工作线程，实现了 Runnable 接口，并持有一个线程 thread，一个初始化的任务 firstTask。thread 是在调用构造方法时通过 ThreadFactory 来创建的线程，可以用来执行任务；firstTask 用它来保存传入的第一个任务，这个任务可以有也可以为 null。如果这个值是非空的，那么线程就会在启动初期立即执行这个任务，也

就对应核心线程创建时的情况；如果这个值是 null，那么就需要创建一个线程去执行任务列表（workQueue）中的任务，也就是非核心线程的创建。

Worker 执行任务的模型如下图所示：

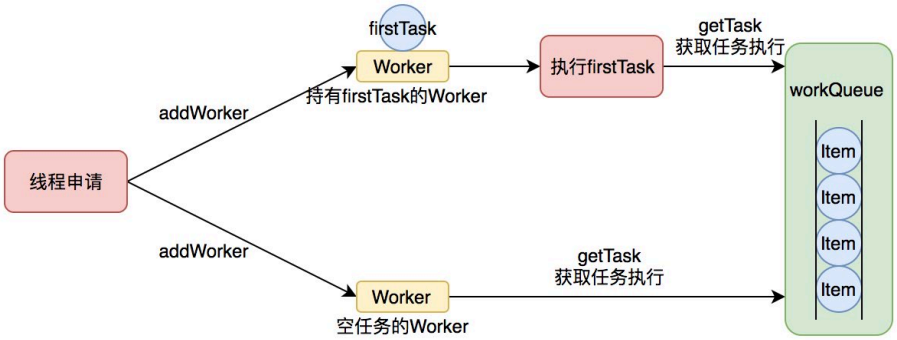


图 7　Worker 执行任务

线程池需要管理线程的生命周期，需要在线程长时间不运行的时候进行回收。线程池使用一张 Hash 表去持有线程的引用，这样可以通过添加引用、移除引用这样的操作来控制线程的生命周期。这个时候重要的就是如何判断线程是否在运行。

Worker 是通过继承 AQS，使用 AQS 来实现独占锁这个功能。没有使用可重入锁 ReentrantLock，而是使用 AQS，为的就是实现不可重入的特性去反应线程现在的执行状态。

1. lock 方法一旦获取了独占锁，表示当前线程正在执行任务中。
2. 如果正在执行任务，则不应该中断线程。
3. 如果该线程现在不是独占锁的状态，也就是空闲的状态，说明它没有在处理任务，这时可以对该线程进行中断。
4. 线程池在执行 shutdown 方法或 tryTerminate 方法时会调用 interruptIdleWorkers 方法来中断空闲的线程，interruptIdleWorkers 方法会使用 tryLock 方法来判断线程池中的线程是否是空闲状态；如果线程是空闲状态则可以安全回收。

在线程回收过程中就使用到了这种特性，回收过程如下图所示：

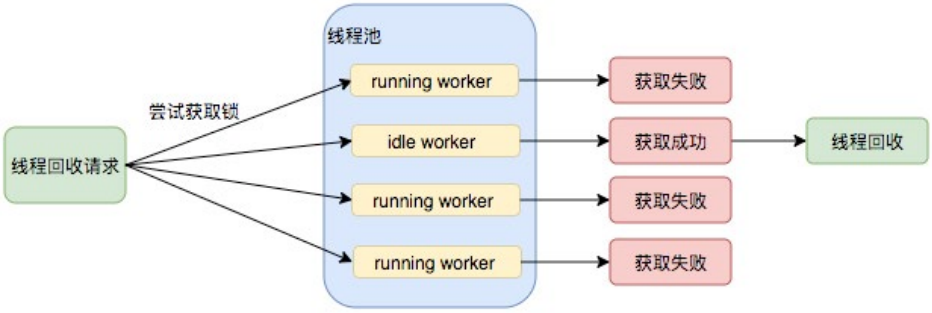


图 8　线程池回收过程

2.4.2 Worker 线程增加

增加线程是通过线程池中的 addWorker 方法，该方法的功能就是增加一个线程，该方法不考虑线程池是在哪个阶段增加的该线程，这个分配线程的策略是在上个步骤完成的，该步骤仅仅完成增加线程，并使它运行，最后返回是否成功这个结果。addWorker 方法有两个参数：firstTask、core。firstTask 参数用于指定新增的线程执行的第一个任务，该参数可以为空；core 参数为 true 表示在新增线程时会判断当前活动线程数是否少于 corePoolSize，false 表示新增线程前需要判断当前活动线程数是否少于 maximumPoolSize，其执行流程如下图所示：

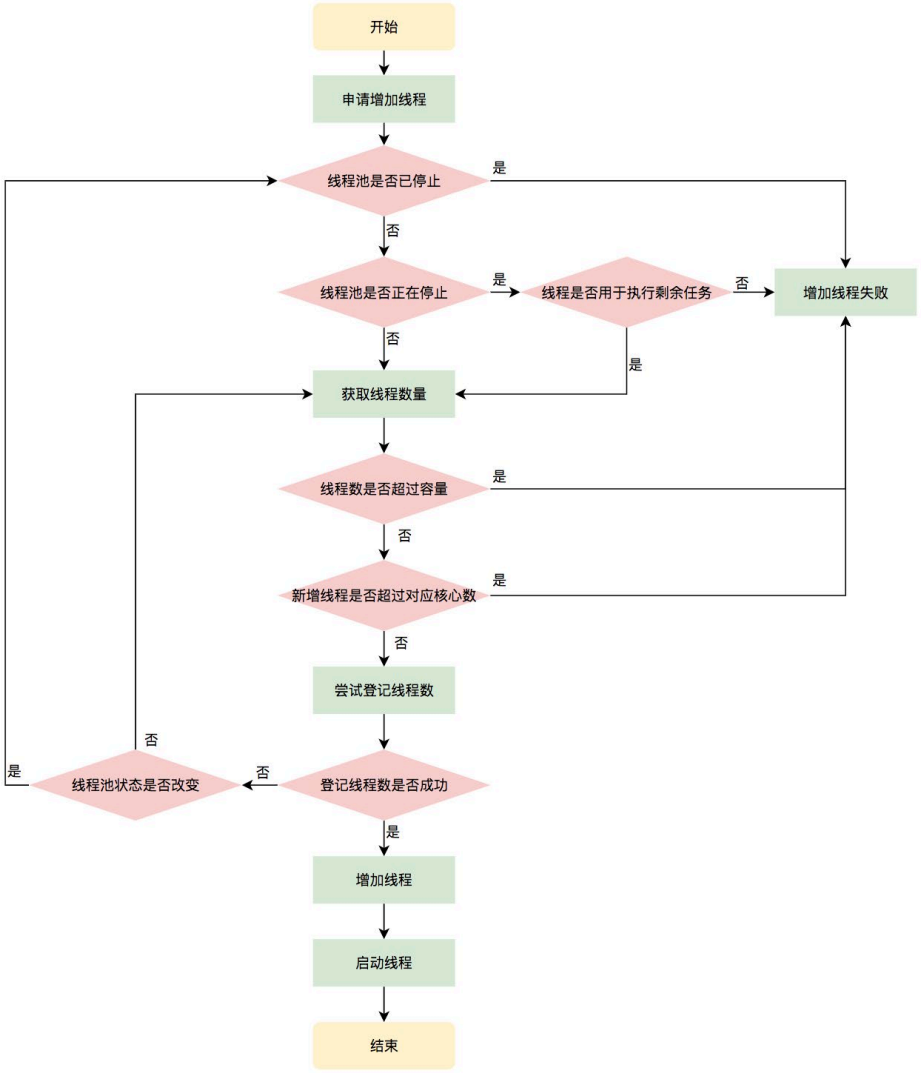


图 9　申请线程执行流程图

2.4.3 Worker 线程回收

线程池中线程的销毁依赖 JVM 自动的回收，线程池做的工作是根据当前线程池的状态维护一定数量的线程引用，防止这部分线程被 JVM 回收，当线程池决定哪些线程需要回收时，只需要将其引用消除即可。Worker 被创建出来后，就会不断地进行轮询，然后获取任务去执行，核心线程可以无限等待获取任务，非核心线程要限

时获取任务。当 Worker 无法获取到任务，也就是获取的任务为空时，循环会结束，Worker 会主动消除自身在线程池内的引用。

```
try {
  while (task != null || (task = getTask()) != null) {
    // 执行任务
  }
} finally {
  processWorkerExit(w, completedAbruptly);// 获取不到任务时，主动回收自己
}
```

线程回收的工作是在 processWorkerExit 方法完成的。

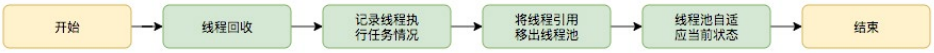


图 10　线程销毁流程

事实上，在这个方法中，将线程引用移出线程池就已经结束了线程销毁的部分。但由于引起线程销毁的可能性有很多，线程池还要判断是什么引发了这次销毁，是否要改变线程池的现阶段状态，是否要根据新状态，重新分配线程。

2.4.4 Worker 线程执行任务

在 Worker 类中的 run 方法调用了 runWorker 方法来执行任务，runWorker 方法的执行过程如下：

1. while 循环不断地通过 getTask() 方法获取任务。
2. getTask() 方法从阻塞队列中取任务。
3. 如果线程池正在停止，那么要保证当前线程是中断状态，否则要保证当前线程不是中断状态。
4. 执行任务。
5. 如果 getTask 结果为 null 则跳出循环，执行 processWorkerExit() 方法，销毁线程。

执行流程如下图所示：

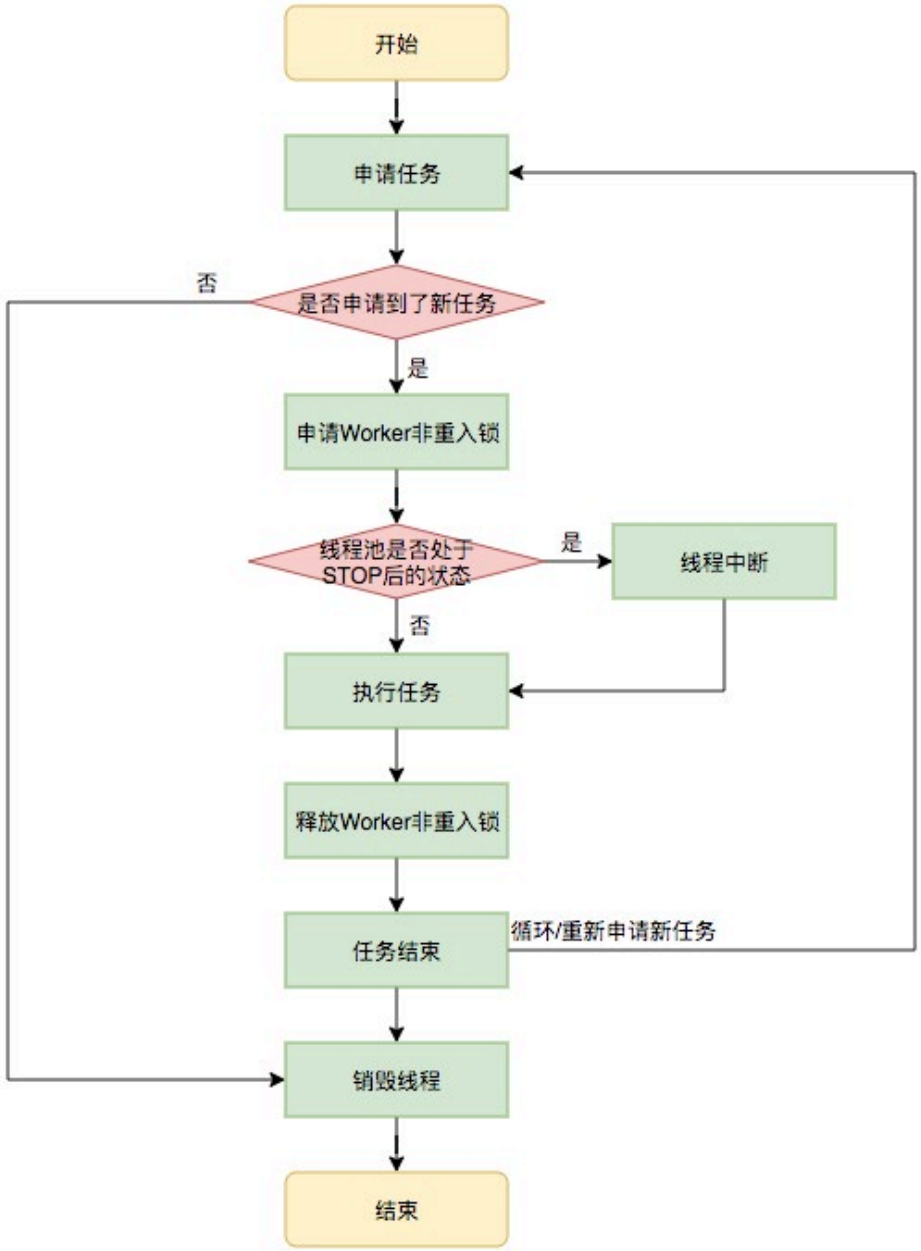


图 11　执行任务流程

三、线程池在业务中的实践

3.1 业务背景

在当今的互联网业界，为了最大程度利用 CPU 的多核性能，并行运算的能力是不可或缺的。通过线程池管理线程获取并发性是一个非常基础的操作，让我们来看两个典型的使用线程池获取并发性的场景。

场景 1：快速响应用户请求

描述：用户发起的实时请求，服务追求响应时间。比如说用户要查看一个商品的信息，那么我们需要将商品维度的一系列信息如商品的价格、优惠、库存、图片等等聚合起来，展示给用户。

分析：从用户体验角度看，这个结果响应的越快越好，如果一个页面半天都刷不出，用户可能就放弃查看这个商品了。而面向用户的功能聚合通常非常复杂，伴随着调用与调用之间的级联、多级级联等情况，业务开发同学往往会选择使用线程池这种简单的方式，将调用封装成任务并行的执行，缩短总体响应时间。另外，使用线程池也是有考量的，这种场景最重要的就是获取最大的响应速度去满足用户，所以应该不设置队列去缓冲并发任务，调高 corePoolSize 和 maxPoolSize 去尽可能创造多的线程快速执行任务。

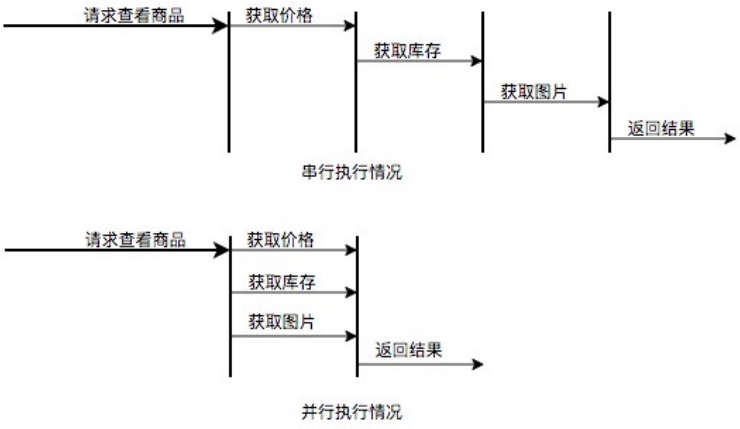


图 12　并行执行任务提升任务响应速度

场景 2：快速处理批量任务

描述：离线的大量计算任务，需要快速执行。比如说，统计某个报表，需要计算出全国各个门店中有哪些商品有某种属性，用于后续营销策略的分析，那么我们需要查询全国所有门店中的所有商品，并且记录具有某属性的商品，然后快速生成报表。

分析：这种场景需要执行大量的任务，我们也会希望任务执行的越快越好。这种情况下，也应该使用多线程策略，并行计算。但与响应速度优先的场景区别在于，这类场景任务量巨大，并不需要瞬时的完成，而是关注如何使用有限的资源，尽可能在单位时间内处理更多的任务，也就是吞吐量优先的问题。所以应该设置队列去缓冲并发任务，调整合适的 corePoolSize 去设置处理任务的线程数。在这里，设置的线程数过多可能还会引发线程上下文切换频繁的问题，也会降低处理任务的速度，降低吞吐量。

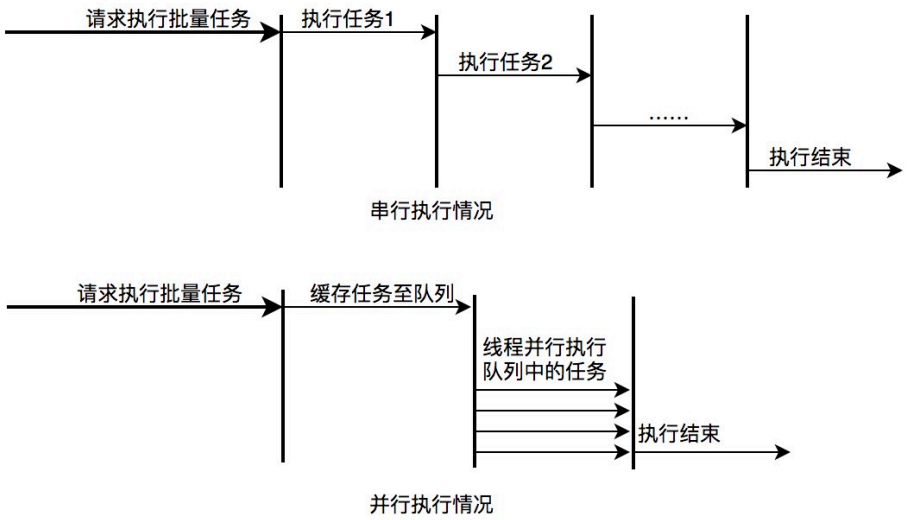


图 13　并行执行任务提升批量任务执行速度

3.2 实际问题及方案思考

线程池使用面临的核心的问题在于：**线程池的参数并不好配置**。一方面线程池的运行

机制不是很好理解，配置合理需要强依赖开发人员的个人经验和知识；另一方面，线程池执行的情况和任务类型相关性较大，IO 密集型和 CPU 密集型的任务运行起来的情况差异非常大，这导致业界并没有一些成熟的经验策略帮助开发人员参考。

关于线程池配置不合理引发的故障，公司内部有较多记录，下面举一些例子：

Case1：2018 年 XX 页面展示接口大量调用降级：

事故描述：XX 页面展示接口产生大量调用降级，数量级在几十到上百。

事故原因：该服务展示接口内部逻辑使用线程池做并行计算，由于没有预估好调用的流量，导致最大核心数设置偏小，大量抛出 RejectedExecutionException，触发接口降级条件，示意图如下：

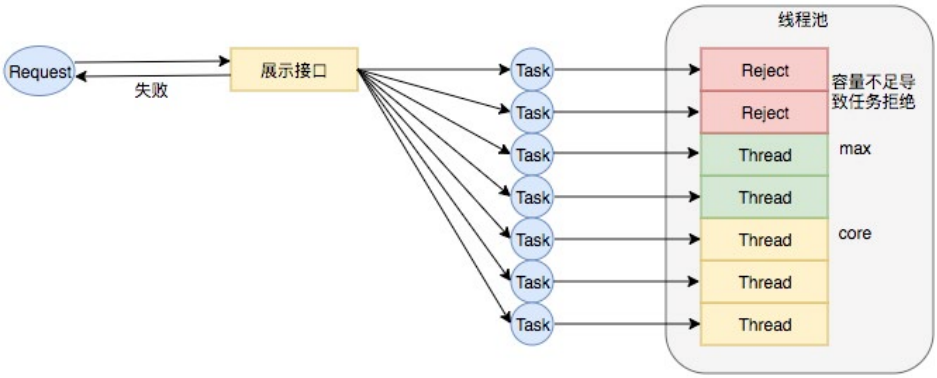


图 14　线程数核心设置过小引发 RejectExecutionException

Case2：2018 年 XX 业务服务不可用 S2 级故障

事故描述：XX 业务提供的服务执行时间过长，作为上游服务整体超时，大量下游服务调用失败。

事故原因：该服务处理请求内部逻辑使用线程池做资源隔离，由于队列设置过长，最大线程数设置失效，导致请求数量增加时，大量任务堆积在队列中，任务执行时间过长，最终导致下游服务的大量调用超时失败。示意图如下：

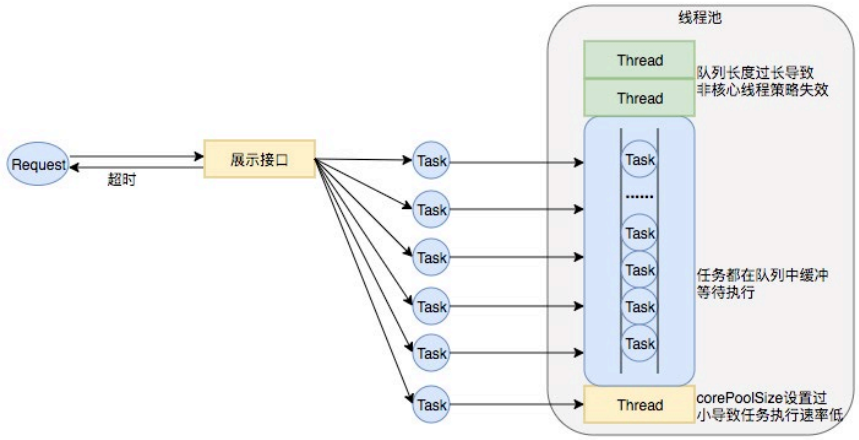


图 15　线程池队列长度设置过长、corePoolSize 设置过小导致任务执行速度低

业务中要使用线程池，而使用不当又会导致故障，那么我们怎样才能更好地使用线程池呢？针对这个问题，我们下面延展几个方向：

1. 能否不用线程池？

回到最初的问题，业务使用线程池是为了获取并发性，对于获取并发性，是否可以有什么其他的方案呢替代？我们尝试进行了一些其他方案的调研：

名称	描述	优势	劣势
Disruptor框架	线程池内部是通过一个工作队列去维护任务的执行的，它有一个根本性的缺陷：连续争用问题。也就是多个线程在申请任务时，为了合理地分配任务要付出锁资源，对比快速的任务执行来说，这部分申请的损耗是巨大的。 高性能进程间消息库LMAX使用了一个叫作环形缓冲的数据结构，用这种这个特殊的数据结构替代队列，将会避免申请任务时出现的连续争用状况。	• 避免连续争用，性能更佳	• 缺乏线程管理的能力，使用场景较少
Actor框架	Actor模型通过维护多个Actor去处理并发的任务，它放弃了直接使用线程去获取并发性，而是自己定义了一系列系统组件应该如何动作和交互的通用规则，不需要开发者直接使用线程。 通过在原生的线程或协程的级别上做了更高层次的封装，只需要开发者关心每个Actor的逻辑即可实现并发操作。由于避免了直接使用锁，很大程度解决了传统并发编程模式下大量依赖悲观锁导致的资源竞争情况。	• 无锁策略，性能更佳 • 避免直接使用线程，安全性更高	• 在Java中缺乏成熟的应用 • 内部复杂，难以排查和调试
协程框架	协程是一种用户态的轻量级线程，其拥有自己的寄存器上下文和栈，当调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。这种切换上下文的方式要小于线程的开销。在瓶颈侧重IO的情况，使用协程获得并发性要优于使用线程。	• 侧重IO情况时，性能更佳 • 与多线程策略无冲突，可结合使用	• 在Java中缺乏成熟的应用

综合考虑，这些新的方案都能在某种情况下提升并行任务的性能，然而本次重点解决的问题是如何更简易、更安全地获得的并发性。另外，Actor 模型的应用实际上甚少，只在 Scala 中使用广泛，协程框架在 Java 中维护的也不成熟。这三者现阶段都不是足够的易用，也并不能解决业务上现阶段的问题。

2. 追求参数设置合理性?

有没有一种计算公式，能够让开发同学很简易地计算出某种场景中的线程池应该是什么参数呢?

带着这样的疑问，我们调研了业界的一些线程池参数配置方案：

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ $The\ optimal\ pool\ size\ for\ keeping\ the\ processors\ at\ the\ desired\ utilization\ is:$ $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

调研了以上业界方案后，我们并没有得出通用的线程池计算方式。并发任务的执行情况和任务类型相关，IO 密集型和 CPU 密集型的任务运行起来的情况差异非常大，但这种占比是较难合理预估的，这导致很难有一个简单有效的通用公式帮我们直接计算出结果。

3. 线程池参数动态化?

尽管经过谨慎的评估，仍然不能够保证一次计算出来合适的参数，那么我们是否可以

将修改线程池参数的成本降下来，这样至少可以发生故障的时候可以快速调整从而缩短故障恢复的时间呢？基于这个思考，我们是否可以将线程池的参数从代码中迁移到分布式配置中心上，实现线程池参数可动态配置和即时生效，线程池参数动态化前后的参数修改流程对比如下：

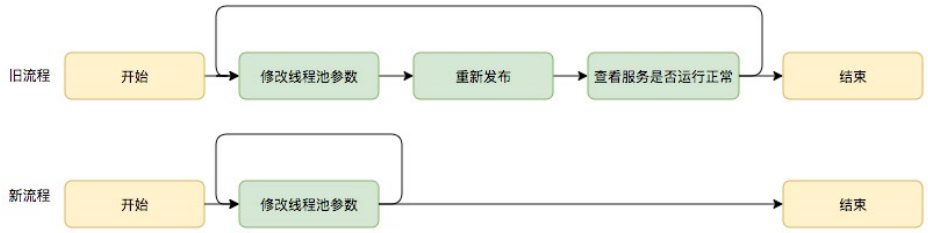


图 16　动态修改线程池参数新旧流程对比

基于以上三个方向对比，我们可以看出参数动态化方向简单有效。

3.3 动态化线程池

3.3.1 整体设计

动态化线程池的核心设计包括以下三个方面：

1. 简化线程池配置：线程池构造参数有 8 个，但是最核心的是 3 个：corePoolSize、maximumPoolSize，workQueue，它们最大程度地决定了线程池的任务分配和线程分配策略。考虑到在实际应用中我们获取并发性的场景主要是两种：（1）并行执行子任务，提高响应速度。这种情况下，应该使用同步队列，没有什么任务应该被缓存下来，而是应该立即执行。（2）并行执行大批次任务，提升吞吐量。这种情况下，应该使用有界队列，使用队列去缓冲大批量的任务，队列容量必须声明，防止任务无限制堆积。所以线程池只需要提供这三个关键参数的配置，并且提供两种队列的选择，就可以满足绝大多数的业务需求，Less is More。
2. 参数可动态修改：为了解决参数不好配，修改参数成本高等问题。在 Java 线程池留有高扩展性的基础上，封装线程池，允许线程池监听同步外部的消息，

根据消息进行修改配置。将线程池的配置放置在平台侧，允许开发同学简单的查看、修改线程池配置。

3. 增加线程池监控：对某事物缺乏状态的观测，就对其改进无从下手。在线程池执行任务的生命周期添加监控能力，帮助开发同学了解线程池状态。

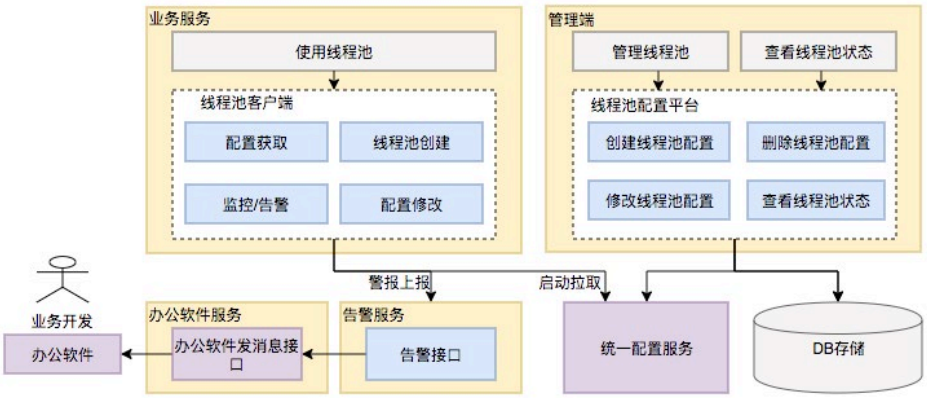


图 17　动态化线程池整体设计

3.3.2 功能架构

动态化线程池提供如下功能：

动态调参：支持线程池参数动态调整、界面化操作；包括修改线程池核心大小、最大核心大小、队列长度等；参数修改后及时生效。 **任务监控**：支持应用粒度、线程池粒度、任务粒度的 Transaction 监控；可以看到线程池的任务执行情况、最大任务执行时间、平均任务执行时间、95/99 线等。 **负载告警**：线程池队列任务积压到一定值的时候会通过大象（美团内部通讯工具）告知应用开发负责人；当线程池负载数达到一定阈值的时候会通过大象告知应用开发负责人。 **操作监控**：创建 / 修改和删除线程池都会通知到应用的开发负责人。 **操作日志**：可以查看线程池参数的修改记录，谁在什么时候修改了线程池参数、修改前的参数值是什么。 **权限校验**：只有应用开发负责人才能够修改应用的线程池参数。

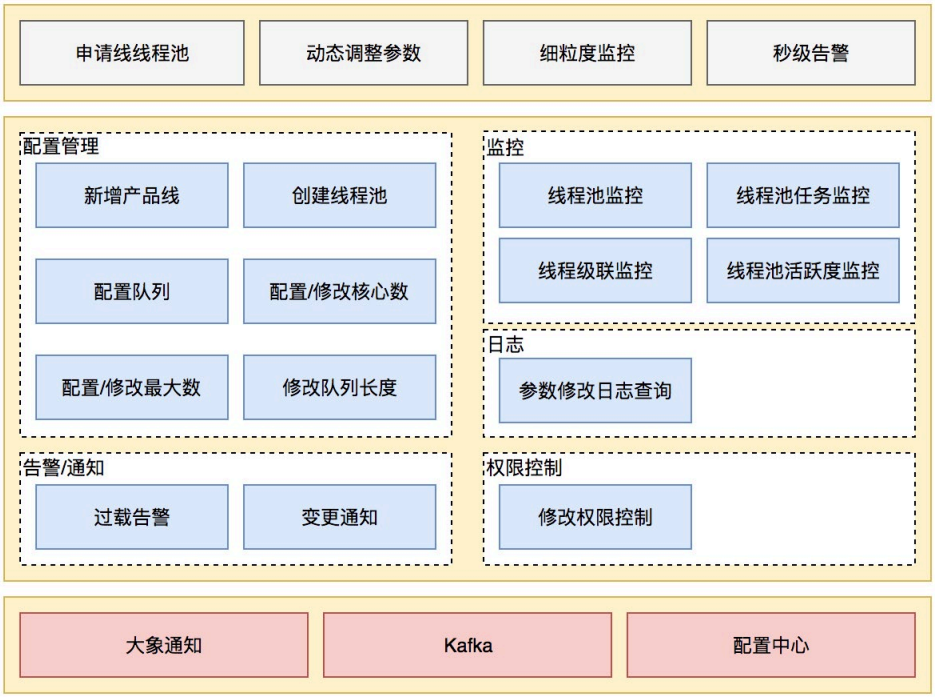


图 18　动态化线程池功能架构

参数动态化

JDK 原生线程池 ThreadPoolExecutor 提供了如下几个 public 的 setter 方法，如下图所示：

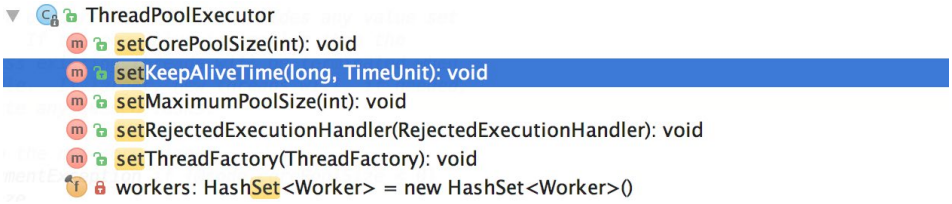


图 19　JDK 线程池参数设置接口

JDK 允许线程池使用方通过 ThreadPoolExecutor 的实例来动态设置线程池的核心策略，以 setCorePoolSize 为方法例，在运行期线程池使用方调用此方法设置 corePoolSize 之后，线程池会直接覆盖原来的 corePoolSize 值，并且基于当前

值和原始值的比较结果采取不同的处理策略。对于当前值小于当前工作线程数的情况，说明有多余的 worker 线程，此时会向当前 idle 的 worker 线程发起中断请求以实现回收，多余的 worker 在下次 idel 的时候也会被回收；对于当前值大于原始值且当前队列中有待执行任务，则线程池会创建新的 worker 线程来执行队列任务，setCorePoolSize 具体流程如下：

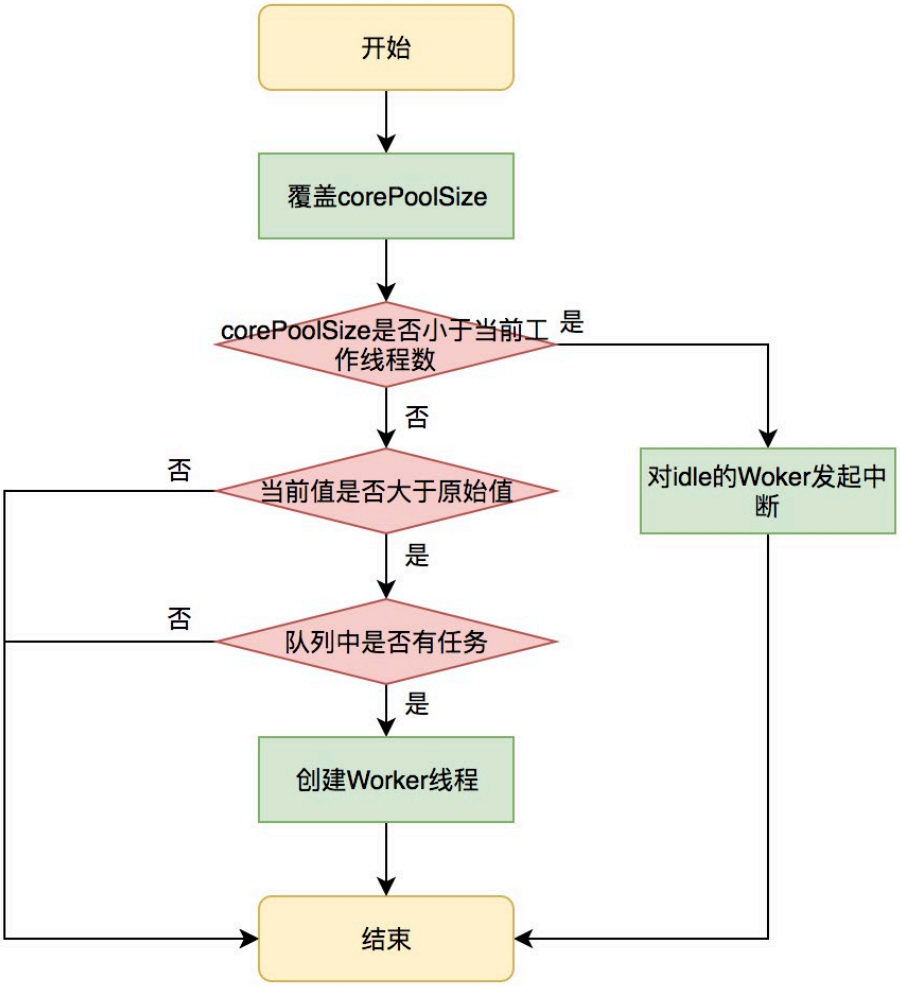


图 20　setCorePoolSize 方法执行流程

线程池内部会处理好当前状态做到平滑修改，其他几个方法限于篇幅，这里不一一

介绍。重点是基于这几个 public 方法，我们只需要维护 ThreadPoolExecutor 的实例，并且在需要修改的时候拿到实例修改其参数即可。基于以上的思路，我们实现了线程池参数的动态化、线程池参数在管理平台可配置可修改，其效果图如下图所示：

图 21　可动态修改线程池参数

用户可以在管理平台上通过线程池的名字找到指定的线程池，然后对其参数进行修改，保存后会实时生效。目前支持的动态参数包括核心数、最大值、队列长度等。除此之外，在界面中，我们还能看到用户可以配置是否开启告警、队列等待任务告警阈

值、活跃度告警等等。关于监控和告警，我们下面一节会对齐进行介绍。

线程池监控

除了参数动态化之外，为了更好地使用线程池，我们需要对线程池的运行状况有感知，比如当前线程池的负载是怎么样的？分配的资源够不够用？任务的执行情况是怎么样的？是长任务还是短任务？基于对这些问题的思考，动态化线程池提供了多个维度的监控和告警能力，包括：线程池活跃度、任务的执行 Transaction（频率、耗时）、Reject 异常、线程池内部统计信息等等，既能帮助用户从多个维度分析线程池的使用情况，又能在出现问题第一时间通知到用户，从而避免故障或加速故障恢复。

1. 负载监控和告警

线程池负载关注的核心问题是：基于当前线程池参数分配的资源够不够。对于这个问题，我们可以从事前和事中两个角度来看。事前，线程池定义了“活跃度”这个概念，来让用户在发生 Reject 异常之前能够感知线程池负载问题，线程池活跃度计算公式为：线程池活跃度 = activeCount/maximumPoolSize。这个公式代表当活跃线程数趋向于 maximumPoolSize 的时候，代表线程负载趋高。事中，也可以从两方面来看线程池的过载判定条件，一个是发生了 Reject 异常，一个是队列中有等待任务（支持定制阈值）。以上两种情况发生了都会触发告警，告警信息会通过大象推送给服务所关联的负责人。

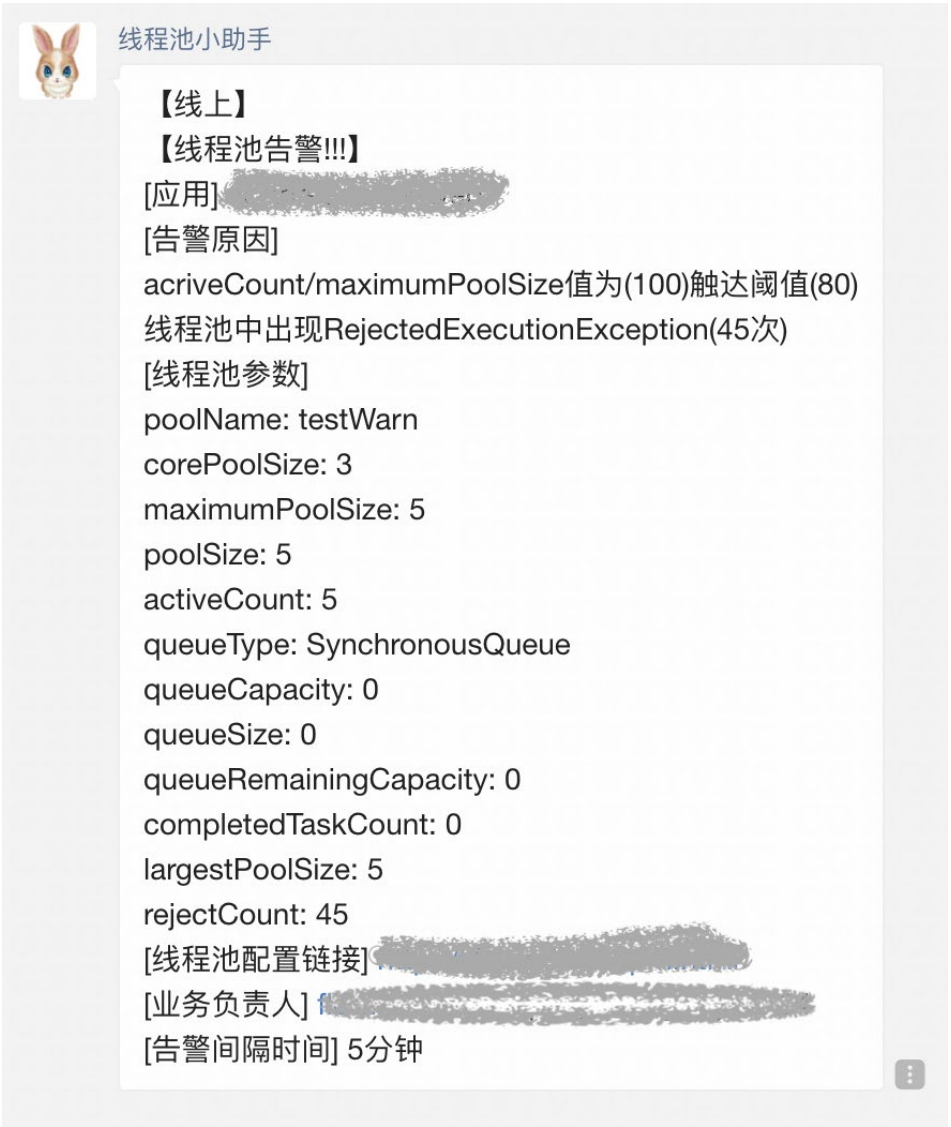


图 22　大象告警通知

2. 任务级精细化监控

在传统的线程池应用场景中，线程池中的任务执行情况对于用户来说是透明的。比如在一个具体的业务场景中，业务开发申请了一个线程池同时用于执行两种任务，一个是发消息任务、一个是发短信任务，这两类任务实际执行的频率和时长对于用户来说

没有一个直观的感受，很可能这两类任务不适合共享一个线程池，但是由于用户无法感知，因此也无从优化。动态化线程池内部实现了任务级别的埋点，且允许为不同的业务任务指定具有业务含义的名称，线程池内部基于这个名称做 Transaction 打点，基于这个功能，用户可以看到线程池内部任务级别的执行情况，且区分业务，任务监控示意图如下图所示：

图 23　线程池任务执行监控

3. 运行时状态实时查看

用户基于 JDK 原生线程池 ThreadPoolExecutor 提供的几个 public 的 getter 方法，可以读取到当前线程池的运行状态以及参数，如下图所示：

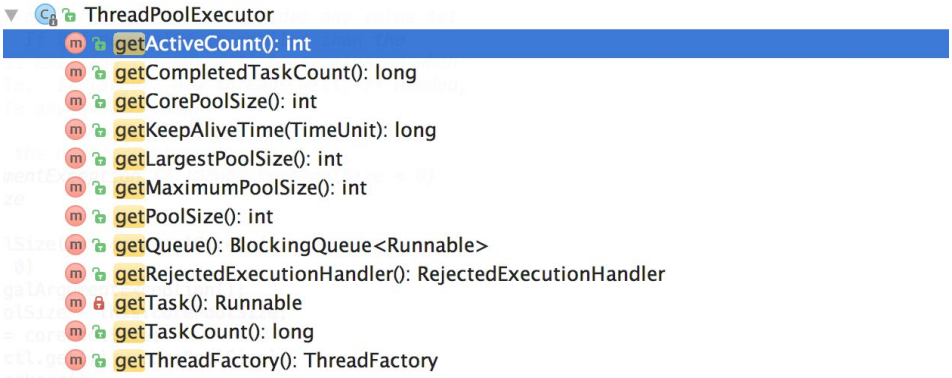
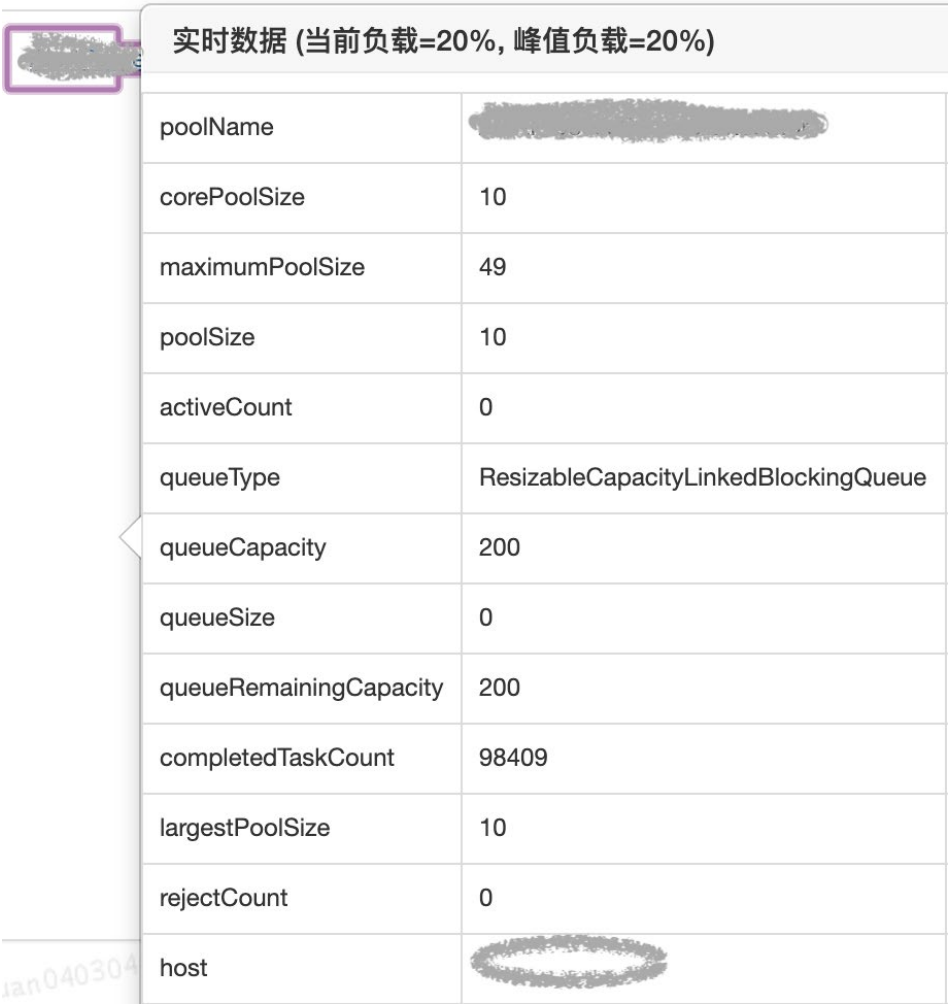


图 24　线程池实时运行情况

动态化线程池基于这几个接口封装了运行时状态实时查看的功能，用户基于这个功能可以了解线程池的实时状态，比如当前有多少个工作线程，执行了多少个任务，队列中等待的任务数等等。效果如下图所示：



实时数据 (当前负载=20%, 峰值负载=20%)

poolName	[redacted]
corePoolSize	10
maximumPoolSize	49
poolSize	10
activeCount	0
queueType	ResizableCapacityLinkedBlockingQueue
queueCapacity	200
queueSize	0
queueRemainingCapacity	200
completedTaskCount	98409
largestPoolSize	10
rejectCount	0
host	[redacted]

图 25　线程池实时运行情况

3.4 实践总结

面对业务中使用线程池遇到的实际问题，我们曾回到支持并发性问题本身来思考有没有取代线程池的方案，也曾尝试着去追求线程池参数设置的合理性，但面对业界方案具体落地的复杂性、可维护性以及真实运行环境的不确定性，我们在前两个方向上可谓“举步维艰”。最终，我们回到线程池参数动态化方向上探索，得出一个且可以解决业务问题的方案，虽然本质上还是没有逃离使用线程池的范畴，但是在成本和收益

之间，算是取得了一个很好的平衡。成本在于实现动态化以及监控成本不高，收益在于：在不颠覆原有线程池使用方式的基础之上，从降低线程池参数修改的成本以及多维度监控这两个方面降低了故障发生的概率。希望本文提供的动态化线程池思路能对大家有帮助。

四、参考资料

[1] JDK 1.8 源码
[2] 维基百科 – 线程池
[3] 更好的使用 Java 线程池
[4] 维基百科 Pooling(Resource Management)
[5] 深入理解 Java 线程池：ThreadPoolExecutor
[6]《Java 并发编程实践》

作者简介

致远，2018 年加入美团点评，美团到店综合研发中心后台开发工程师。
陆晨，2015 年加入美团点评，美团到店综合研发中心后台技术专家。

招聘信息

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

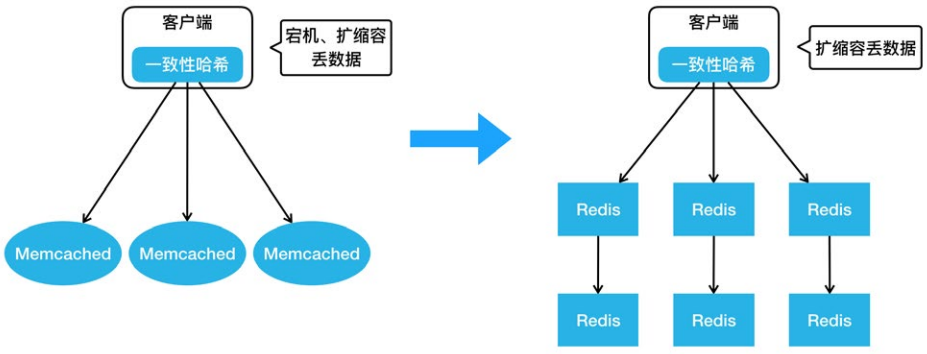
美团万亿级 KV 存储架构与实践

作者：泽斌

KV 存储作为美团一项重要的在线存储服务，承载了在线服务每天万亿级的请求量。在 2019 年 QCon 全球软件开发大会（上海站）上，美团高级技术专家齐泽斌分享了《美团点评万亿级 KV 存储架构与实践》，本文系演讲内容的整理，主要分为四个部分：第一部分讲述了美团 KV 存储的发展历程；第二部分阐述了内存 KV Squirrel 架构和实践；第三部分介绍了持久化 KV Cellar 架构和实践；最后分享了未来的发展规划和业界新趋势。

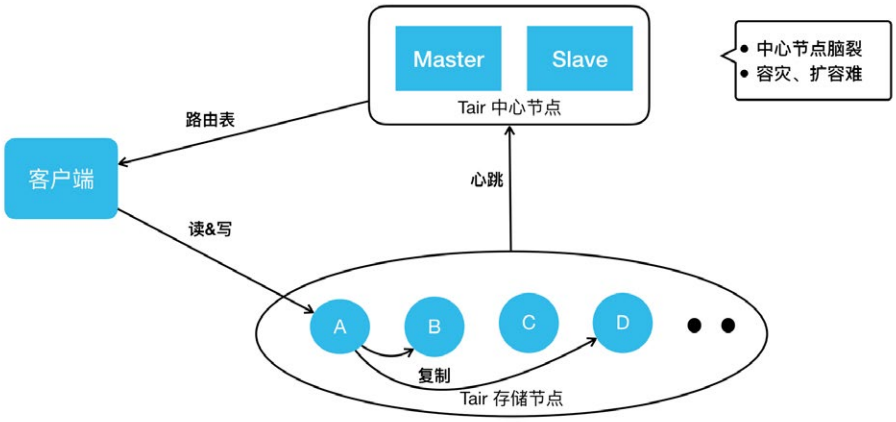
美团点评 KV 存储发展历程

美团第一代的分布式 KV 存储如下图左侧的架构所示，相信很多公司都经历过这个阶段。在客户端内做一致性哈希，在后端部署很多的 Memcached 实例，这样就实现了最基本的 KV 存储分布式设计。但这样的设计存在很明显的问题：比如在宕机摘除节点时，会丢数据，缓存空间不够需要扩容，一致性哈希也会丢失一些数据等等，这样会给业务开发带来的很多困扰。



随着 Redis 项目的成熟，美团也引入了 Redis 来解决我们上面提到的问题，进而演进出来如上图右侧这样一个架构。大家可以看到，客户端还是一样，采用了一致性哈

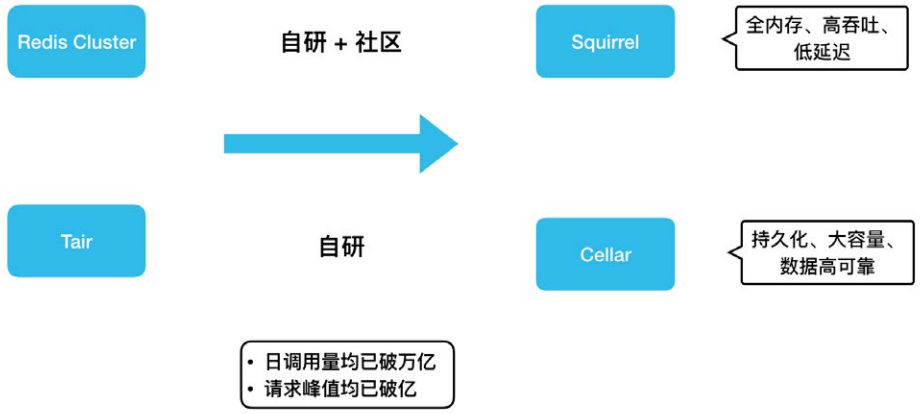
希算法，服务器端变成了 Redis 组成的主从结构。当任何一个节点宕机，我们可以通过 Redis 哨兵完成 Failover，实现高可用。但有一个问题还是没有解决，如果扩缩容的话，一致性哈希仍然会丢数据，那么这个问题该如何解决呢?



这个时候，我们发现有了一个比较成熟的 KV 存储开源项目：阿里 Tair 。2014 年，我们引入了 Tair 来满足业务 KV 存储方面的需求。Tair 开源版本的架构主要分成三部分：上图下边是存储节点，存储节点会上报心跳到它的中心节点，中心节点内部有两个配置管理节点，会监控所有的存储节点。当有任何存储节点宕机或者扩容时，它会做集群拓扑的重新构建。当客户端启动时，它会直接从中心节点拉来一个路由表。这个路由表简单来说就是一个集群的数据分布图，客户端根据路由表直接去存储节点读写。针对之前 KV 的扩容丢数据问题，它也有数据迁移机制来保证数据的完整性。

但是，我们在使用的过程中，还遇到了一些其他问题，比如中心节点虽然是主备高可用的，但实际上它没有类似于分布式仲裁的机制，所以在网络分割的情况下，它是有可能发生“脑裂”的，这个也给我们的业务造成过比较大的影响。另外，在容灾扩容时，也遇到过数据迁移影响到业务可用性的问题。另外，我们之前用过 Redis ，业务会发现 Redis 的数据结构特别丰富，而 Tair 还不支持这些数据结构。虽然我们用 Tair 解决了一些问题，但是 Tair 也无法完全满足业务需求。毕竟，在美团这样一个业务规模较大和业务复杂度较高的场景下，很难有开源系统能很好地满足我们的需

求。最终，我们决定在已应用的开源系统之上进行自研。



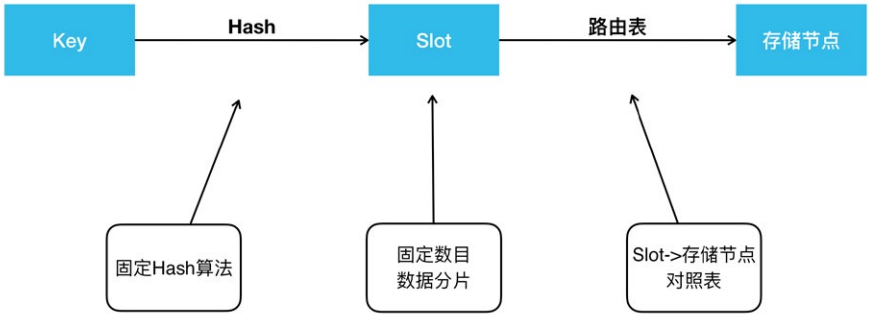
刚好在 2015 年，Redis 官方正式发布了集群版本 Redis Cluster。所以，我们紧跟社区步伐，并结合内部需求做了很多开发工作，演进出了全内存、高吞吐、低延迟的 KV 存储 Squirrel。另外，基于 Tair，我们还加入了很多自研的功能，演进出持久化、大容量、数据高可靠的 KV 存储 Cellar 。因为 Tair 的开源版本已经有四五年没有更新了，所以，Cellar 的迭代完全靠美团自研，而 Redis 社区一直很活跃。总的来说，Squirrel 的迭代是自研和社区并重，自研功能设计上也会尽量与官方架构进行兼容。后面大家可以看到，因为这些不同，Cellar 和 Squirrel 在解决同样的问题时也选取了不同的设计方案。

这两个存储其实都是 KV 存储领域不同的解决方案。在实际应用上，如果业务的数据量小，对延迟敏感，我们建议大家用 Squirrel ；如果数据量大，对延迟不是特别敏感，我们建议用成本更低的 Cellar 。目前这两套 KV 存储系统在美团内部每天的调用量均已突破万亿，它们的请求峰值也都突破了每秒亿级。

内存 KV Squirrel 架构和实践

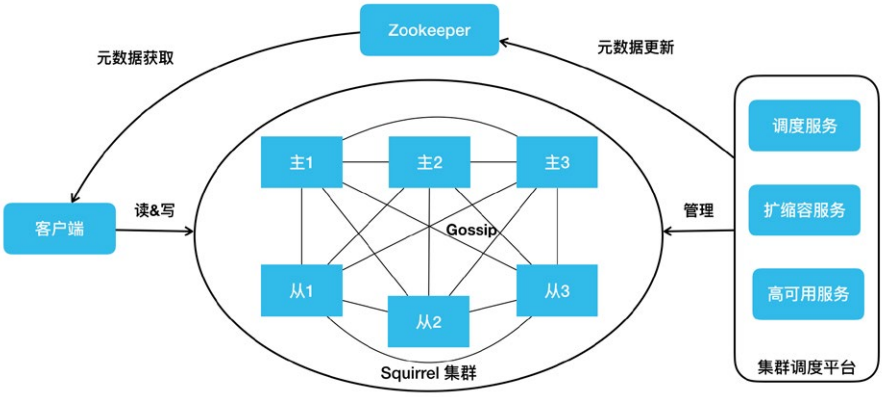
在开始之前，本文先介绍两个存储系统共通的地方。比如分布式存储的经典问题：数据是如何分布的？这个问题在 KV 存储领域，就是 Key 是怎么分布到存储节点上的。这

里 Squirrel 跟 Cellar 是一样的。当我们拿到一个 Key 后，用固定的哈希算法拿到一个哈希值，然后将哈希值对 Slot 数目取模得到一个 Slot id，我们两个 KV 现在都是预分片 16384 个 Slot 。得到 Slot id 之后，再根据路由表就能查到这个 Slot 存储在哪个存储节点上。这个路由表简单来说就是一个 Slot 到存储节点的对照表。



KV 数据分布介绍

接下来讲一下对高可用架构的认知，个人认为高可用可以从宏观和微观两个角度来看。从宏观的角度来看，高可用就是指容灾怎么做。比如说挂掉了一个节点，你该怎么做？一个机房或者说某个地域的一批机房宕机了，你该怎么做？而从微观的角度看，高可用就是怎么能保证端到端的高成功率。我们在做一些运维升级或者扩缩容数据迁移的时候，能否做到业务请求的高可用？ 本文也会从宏观和微观两个角度来分享美团做的一些高可用工作。



Squirrel 架构

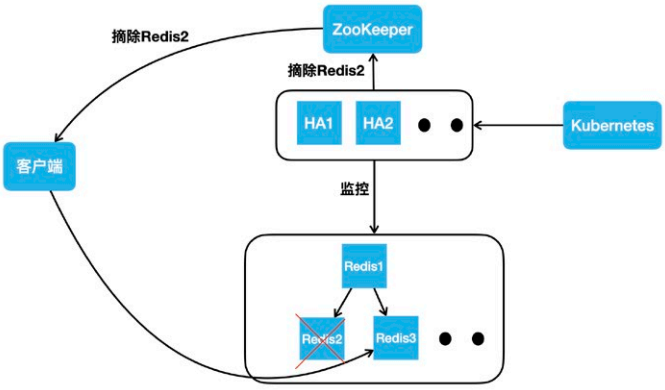
上图就是我们的 Squirrel 架构。中间部分跟 Redis 官方集群是一致的。它有主从的结构，Redis 实例之间通过 Gossip 协议去通信。我们在右边添加了一个集群调度平台，包含调度服务、扩缩容服务和高可用服务等，它会去管理整个集群，把管理结果作为元数据更新到 ZooKeeper。我们的客户端会订阅 ZooKeeper 上的元数据变更，实时获取到集群的拓扑状态，直接在 Redis 集群进行读写操作。

Squirrel 节点容灾

然后再看一下 Squirrel 容灾怎么做。对于 Redis 集群而言，节点宕机已经有完备的处理机制了。官方提供的方案，任何一个节点从宕机到被标记为 FAIL 摘除，一般需要经过 30 秒。主库的摘除可能会影响数据的完整性，所以，我们需要谨慎一些。但是对于从库呢？我们认为这个过程完全没必要。另一点，我们都知道内存的 KV 存储数据量一般都比较小。对于业务量很大的公司来说，它往往会有很多的集群。如果发生交换机故障，会影响到很多的集群，宕机之后去补副本就会变得非常麻烦。为了解决这两个问题，我们做了 HA 高可用服务。

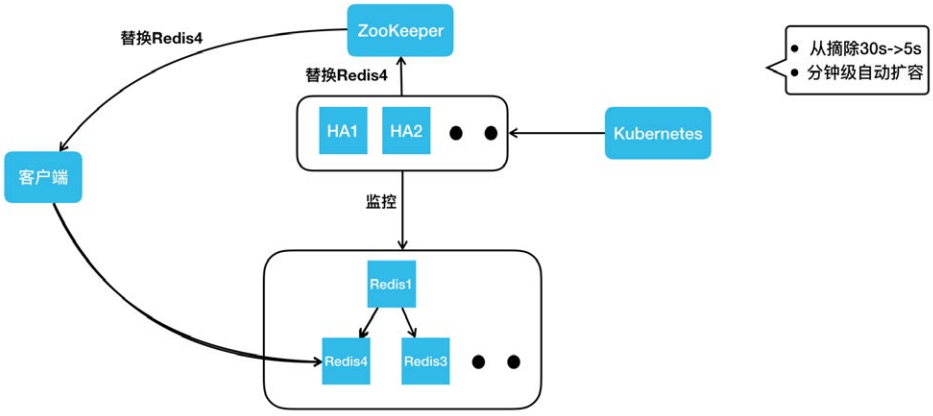
它的架构如下图所示，它会实时监控集群的所有节点。不管是网络抖动，还是发生了宕机（比如说 Redis 2），它可以实时更新 ZooKeeper，告诉 ZooKeeper 去摘除 Redis 2，客户端收到消息后，读流量就直接路由到 Redis 3 上。如果 Redis 2 只

是几十秒的网络抖动，过几十秒之后，如果 HA 节点监控到它恢复后，会把它重新加回。



Squirrel—节点容灾

如果过了一段时间，HA 判断它属于一个永久性的宕机，HA 节点会直接从 Kubernetes 集群申请一个新的 Redis 4 容器实例，把它加到集群里。此时，拓扑结构又变成了一主两从的标准结构，HA 节点更新完集群拓扑之后，就会去写 ZooKeeper 通知客户端去更新路由，客户端就能到 Redis 4 这个新从库上进行读操作。

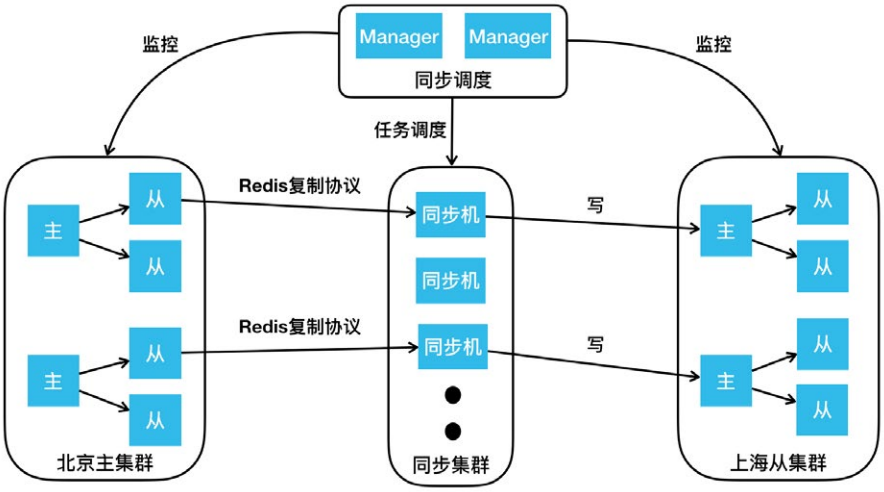


通过上述方案，我们把从库的摘除时间从 30 秒降低到了 5 秒。另外，我们通过 HA 自动申请容器实例加入集群的方式，把宕机补副本变成了一个分钟级的自动操作，不

需要任何人工的介入。

Squirrel 跨地域容灾

我们解决了单节点宕机的问题，那么跨地域问题如何解决呢？我们首先来看下跨地域有什么不同。第一，相对于同地域机房间的网络而言，跨地域专线很不稳定；第二，跨地域专线的带宽是非常有限且昂贵的。而集群内的复制没有考虑极端的网络环境。假如我们把主库部署到北京，两个从库部署在上海，同样一份数据要在北上专线传输两次，这样会造成巨大的专线带宽浪费。另外，随着业务的发展和演进，我们也在做单元化部署和异地多活架构。用官方的主从同步，满足不了我们的这些需求。基于此，我们又做了集群间的复制方案。



如上图所示，这里画出了北京的主集群以及上海的从集群，我们要做的是通过集群同步服务，把北京主集群的数据同步到上海从集群上。按照流程，首先要向我们的同步调度模块下发“在两个集群间建立同步链路”的任务，同步调度模块会根据主从集群的拓扑结构，把主从集群间的同步任务下发到同步集群，同步集群收到同步任务后会扮成 Redis 的 Slave，通过 Redis 的复制协议，从主集群上的从库拉取数据，包括 RDB 以及后续的增量变更。同步机收到数据后会把它转成客户端的写命令，写到上

海从集群的主节点里。通过这样的方式，我们把北京主集群的数据同步到了上海的从集群。同样的，我们要做异地多活也很简单，再加一个反向的同步链路，就可以实现集群间的双向同步。

接下来我们讲一下如何做好微观角度的高可用，也就是保持端到端的高成功率。对于 Squirrel ，主要讲如下三个影响成功率的问题：

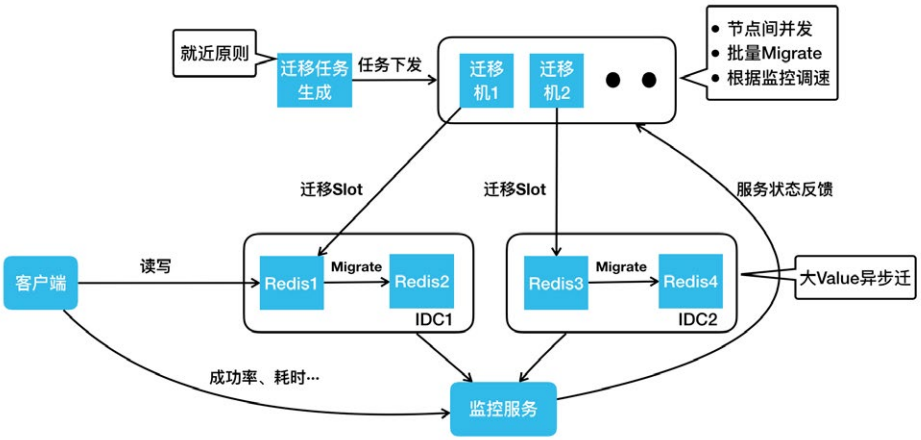
- 数据迁移造成超时抖动。
- 持久化造成超时抖动。
- 热点 Key 请求导致单节点过载。

Squirrel 智能迁移

对于数据迁移，我们主要遇到三个问题：

- Redis Cluster 虽然提供了数据迁移能力，但是对于要迁哪些 Slot，Slot 从哪迁到哪，它并不管。
- 做数据迁移的时候，大家都想越快越好，但是迁移速度过快又可能影响业务正常请求。
- Redis 的 Migrate 命令会阻塞工作线程，尤其在迁移大 Value 的时候会阻塞特别久。

为了解决这些问题，我们做了全新的迁移服务。



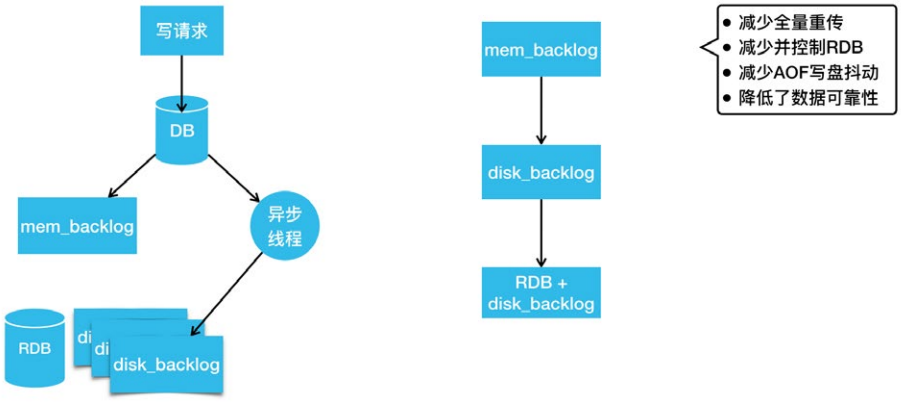
下面我们按照工作流，讲一下它是如何运行的。首先生成迁移任务，这步的核心是“就近原则”，比如说同机房的两个节点做迁移肯定比跨机房的两个节点快。迁移任务生成之后，会把任务下发到一批迁移机上。迁移机迁移的时候，有这样几个特点：

- 第一，会在集群内迁出节点间做并发，比如同时给 Redis 1、Redis 3 下发迁移命令。
- 第二，每个 Migrate 命令会迁移一批 Key。
- 第三，我们会用监控服务去实时采集客户端的成功率、耗时，服务端的负载、QPS 等，之后把这个状态反馈到迁移机上。迁移数据的过程就类似 TCP 慢启动的过程，它会把速度一直往上加，若出现请求成功率下降等情况，它的速度就会降低，最终迁移速度会在动态平衡中稳定下来，这样就达到了最快速的迁移，同时又尽可能小地影响业务的正常请求。

接下来，我们看一下大 Value 的迁移，我们实现了一个异步 Migrate 命令，该命令执行时，Redis 的主线程会继续处理其他的正常请求。如果此时有对正在迁移 Key 的写请求过来，Redis 会直接返回错误。这样最大限度保证了业务请求的正常处理，同时又不会阻塞主线程。

Squirrel 持久化重构

Redis 主从同步时会生成 RDB。生成 RDB 的过程会调用 Fork 产生一个子进程去写数据到硬盘，Fork 虽然有操作系统的 COW 机制，但是当内存用量达到 10 G 或 20 G 时，依然会造成整个进程接近秒级的阻塞。这对在线业务来说几乎是无法接受的。我们也会为数据可靠性要求高的业务去开启 AOF，而开 AOF 就可能因 IO 抖动造成进程阻塞，这也会影响请求成功率。对官方持久化机制的这两个问题，我们的解决方案是重构持久化机制。



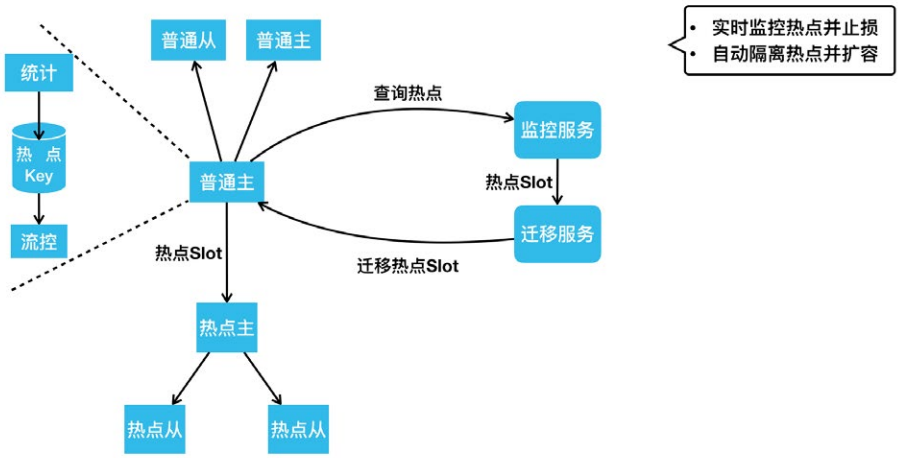
上图是我们最新版的 Redis 持久化机制，写请求会先写到 DB 里，然后写到内存 Backlog，这跟官方是一样的。同时它会把请求发给异步线程，异步线程负责把变更刷到硬盘的 Backlog 里。当硬盘 Backlog 过多时，我们会主动在业务低峰期做一次 RDB ，然后把 RDB 之前生成的 Backlog 删除。

如果这时候我们要做主从同步，去寻找同步点的时候，该怎么办？第一步还是跟官方一样，我们会从内存 Backlog 里找有没有要求的同步点，如果没有，我们会去硬盘 Backlog 找同步点。由于硬盘空间很大，硬盘 Backlog 可以存储特别多的数据，所以很少会出现找不到同步点的情况。如果硬盘 Backlog 也没有，我们就会触发一次类似于全量重传的操作，但这里的全量重传是不需要当场生成 RDB 的，它可以直接用硬盘已存的 RDB 及其之后的硬盘 Backlog 完成全量重传。通过这个设计，我们减

少了很多的全量重传。另外，我们通过控制在低峰区生成 RDB ，减少了很多 RDB 造成的抖动。同时，我们也避免了写 AOF 造成的抖动。不过，这个方案因为写 AOF 是完全异步的，所以会比官方的数据可靠性差一些，但我们认为这个代价换来了可用性的提升，这是非常值得的。

Squirrel 热点 Key

下面看一下 Squirrel 的热点 Key 解决方案。如下图所示，普通主、从是一个正常集群中的节点，热点主、从是游离于正常集群之外的节点。我们看一下它们之间怎么发生联系。

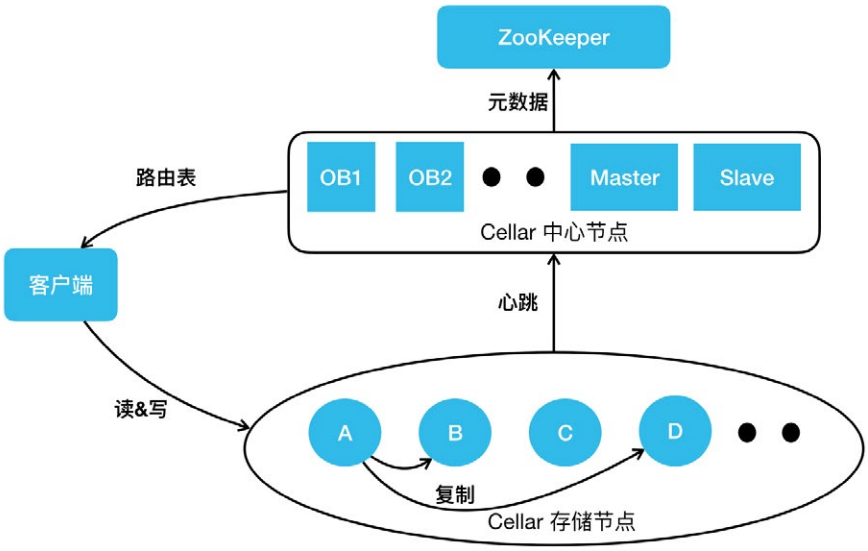


当有请求进来读写普通节点时，节点内会同时做请求 Key 的统计。如果某个 Key 达到了一定的访问量或者带宽的占用量，会自动触发流控以限制热点 Key 访问，防止节点被热点请求打满。同时，监控服务会周期性的去所有 Redis 实例上查询统计到的热点 Key。如果有热点，监控服务会把热点 Key 所在 Slot 上报到我们的迁移服务。迁移服务这时会把热点主从节点加入到这个集群中，然后把热点 Slot 迁移到这个热点主从上。因为热点主从上只有热点 Slot 的请求，所以热点 Key 的处理能力得到了大幅提升。通过这样的设计，我们可以做到实时的热点监控，并及时通过流控去

止损；通过热点迁移，我们能做到自动的热点隔离和快速的容量扩充。

持久化 KV Cellar 架构和实践

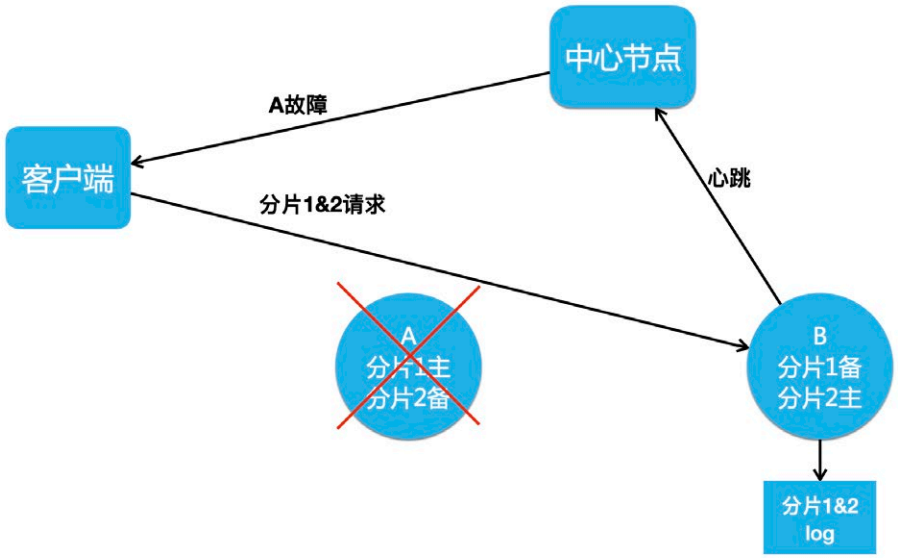
下面看一下持久化 KV Cellar 的架构和实践。下图是我们最新的 Cellar 架构图。



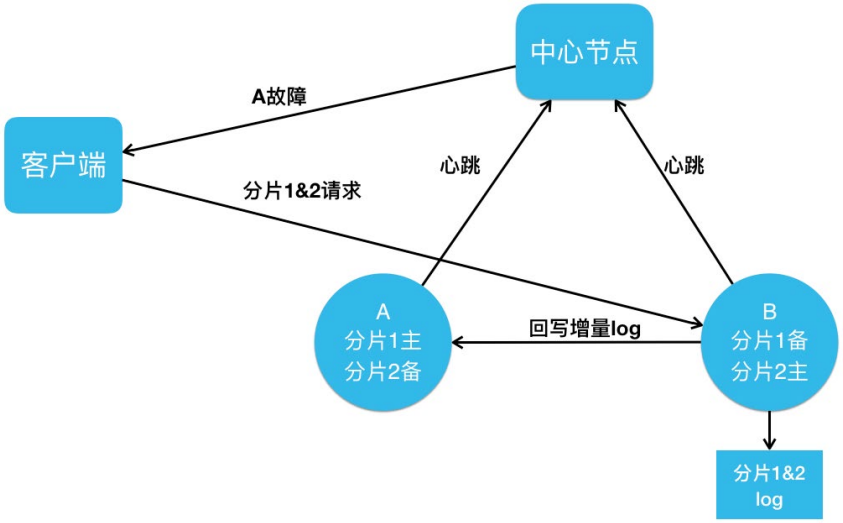
跟阿里开源的 Tair 主要有两个架构上的不同。第一个是 OB，第二个是 ZooKeeper。我们的 OB 跟 ZooKeeper 的 Observer 是类似的作用，提供 Cellar 中心节点元数据的查询服务。它可以实时与中心节点的 Master 同步最新的路由表，客户端的路由表都是从 OB 去拿。 这样做的好处主要有两点，第一，把大量的业务客户端跟集群的大脑 Master 做了天然的隔离，防止路由表请求影响集群的管理。第二，因为 OB 只供路由表查询，不参与集群的管理，所以它可以进行水平扩展，极大地提升了我们路由表的查询能力。另外，我们引入了 ZooKeeper 做分布式仲裁，解决我刚才提到的 Master、Slave 在网络分割情况下的“脑裂”问题，并且通过把集群的元数据存储到 ZooKeeper，我们保证了元数据的高可靠。

Cellar 节点容灾

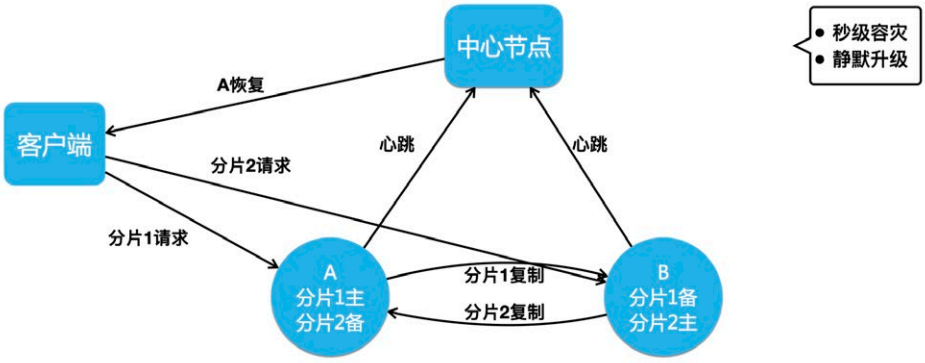
介绍完整体的架构，我们看一下 Cellar 怎么做节点容灾。一个集群节点的宕机一般是临时的，一个节点的网络抖动也是临时的，它们会很快地恢复，并重新加入集群。因为节点的临时离开就把它彻底摘除，并做数据副本补全操作，会消耗大量资源，进而影响到业务请求。所以，我们实现了 Handoff 机制来解决这种节点短时故障带来的影响。



如上图所示，如果 A 节点宕机了，会触发 Handoff 机制，这时候中心节点会通知客户端 A 节点发生了故障，让客户端把分片 1 的请求也打到 B 上。B 节点正常处理完客户端的读写请求之后，还会把本应该写入 A 节点的分片 1&2 数据写入到本地的 Log 中。



如果 A 节点宕机后 3~5 分钟，或者网络抖动 30~50 秒之后恢复了，A 节点就会上报心跳到中心节点，中心节点就会通知 B 节点：“ A 节点恢复了，你去把它不在期间的数据传给它。”这时候，B 节点就会把本地存储的 Log 回写到 A 节点上。等到 A 节点拥有了故障期间的全量数据之后，中心节点就会告诉客户端，A 节点已经彻底恢复了，客户端就可以重新把分片 1 的请求打回 A 节点。

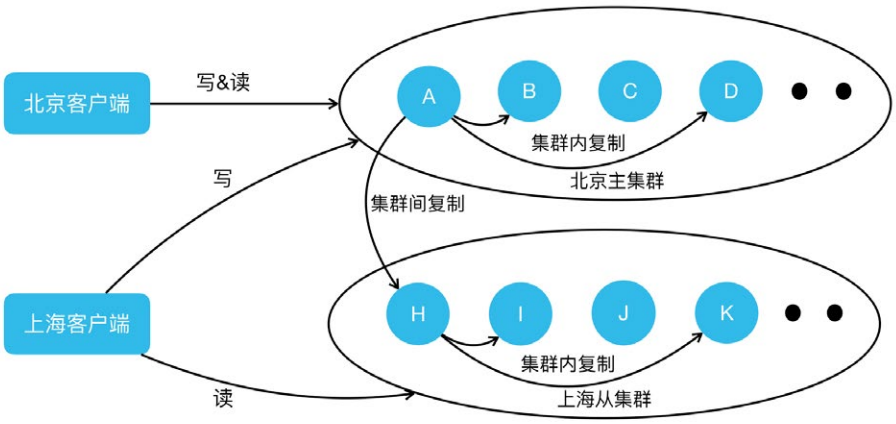


通过这样的操作，我们可以做到秒级的快速节点摘除，而且节点恢复后加回，只需补齐少量的增量数据。另外如果 A 节点要做升级，中心节点先通过主动 Handoff 把 A 节点流量切到 B 节点，A 升级后再回写增量 Log，然后切回流量加入集群。这样通

过主动触发 Handoff 机制，我们就实现了静默升级的功能。

Cellar 跨地域容灾

下面我介绍一下 Cellar 跨地域容灾是怎么做的。Cellar 跟 Squirrel 面对的跨地域容灾问题是一样的，解决方案同样也是集群间复制。以下图一个北京主集群、上海从集群的跨地域场景为例，比如说客户端的写操作到了北京的主集群 A 节点，A 节点会像正常集群内复制一样，把它复制到 B 和 D 节点上。同时 A 节点还会把数据复制一份到从集群的 H 节点。H 节点处理完集群间复制写入之后，它也会做从集群内的复制，把这个写操作复制到从集群的 I 、K 节点上。通过在主从集群的节点间建立这样一个复制链路，我们完成了集群间的数据复制，并且这个复制保证了最低的跨地域带宽占用。同样的，集群间的两个节点通过配置两个双向复制的链路，就可以达到双向同步异地多活的效果。

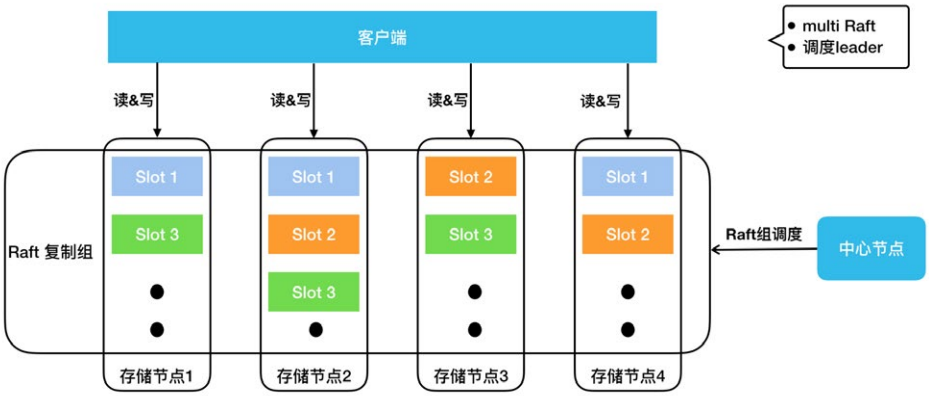


Cellar 强一致

我们做好了节点容灾以及跨地域容灾后，业务又对我们提出了更高要求：强一致存储。我们之前的数据复制是异步的，在做故障摘除时，可能因为故障节点数据还没复制出来，导致数据丢失。但是对于金融支付等场景来说，它们是不容许数据丢失的。

面对这个难题，我们该怎么解决？目前业界主流的解决方案是基于 Paxos 或 Raft 协议的强一致复制。我们最终选择了 Raft 协议。主要是因为 Raft 论文是非常详实的，是一篇工程化程度很高的论文。业界也有不少比较成熟的 Raft 开源实现，可以作为我们研发的基础，进而能够缩短研发周期。

下图是现在 Cellar 集群 Raft 复制模式下的架构图，中心节点会做 Raft 组的调度，它会决定每一个 Slot 的三副本存在哪些节点上。

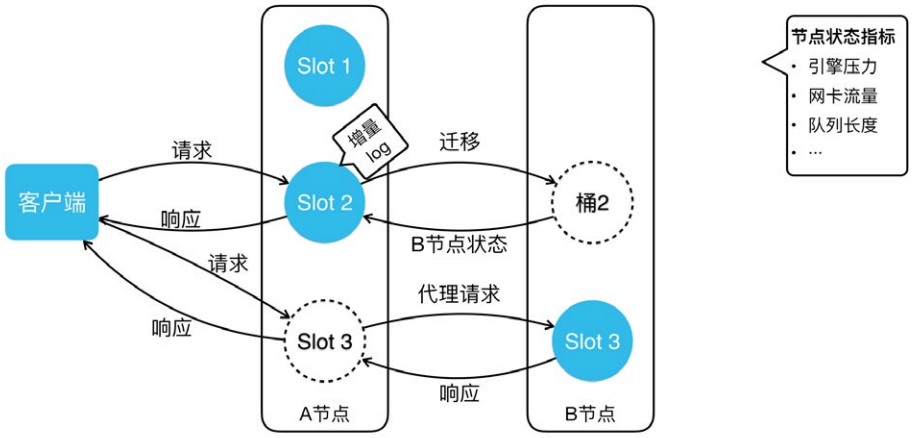


大家可以看到 Slot 1 在存储节点 1、2、4 上，Slot 2 在存储节点 2、3、4 上。每个 Slot 组成一个 Raft 组，客户端会去 Raft Leader 上进行读写。由于我们是预分配了 16384 个 Slot，所以，在集群规模很小的时候，我们的存储节点上可能会有数百甚至上千个 Slot 。这时候如果每个 Raft 复制组都有自己的复制线程、复制请求和 Log 等，那么资源消耗会非常大，写入性能会很差。所以我们做了 Multi Raft 实现，Cellar 会把同一个节点上所有的 Raft 复制组写一份 Log，用同一组线程去做复制，不同 Raft 组间的复制包也会按照目标节点做整合，以保证写入性能不会因 Raft 组过多而变差。 Raft 内部其实是有自己的选主机制，它可以控制自己的主节点，如果有任何节点宕机，它可以通过选举机制选出新的主节点。那么，中心节点是不是就不需要管理 Raft 组了吗？不是的。这里讲一个典型的场景，如果一个集群的部分节点经过几轮宕机恢复的过程，Raft Leader 在存储节点之间会变得极其不均。而为了保证数据的强一致，客户端的读写流量又必须发到 Raft Leader，这时候集群的

节点流量会很不均衡。所以我们的中心节点还会做 Raft 组的 Leader 调度。比如说 Slot 1 存储在节点 1、2、4，并且节点 1 是 Leader。如果节点 1 挂了，Raft 把节点 2 选成了 Leader。然后节点 1 恢复了并重新加入集群，中心节点这时会让节点 2 把 Leader 还给节点 1 。这样，即便经过一系列宕机和恢复，我们存储节点之间的 Leader 数目仍然能保证是均衡的。

接下来，我们看一下 Cellar 如何保证它的端到端高成功率。这里也讲三个影响成功率的问题。Cellar 遇到的数据迁移和热点 Key 问题与 Squirrel 是一样的，但解决方案不一样。这是因为 Cellar 走的是自研路径，不用考虑与官方版本的兼容性，对架构改动更大些。另一个问题是慢请求阻塞服务队列导致大面积超时，这是 Cellar 网络、工作多线程模型设计下会遇到的不同问题。

Cellar 智能迁移



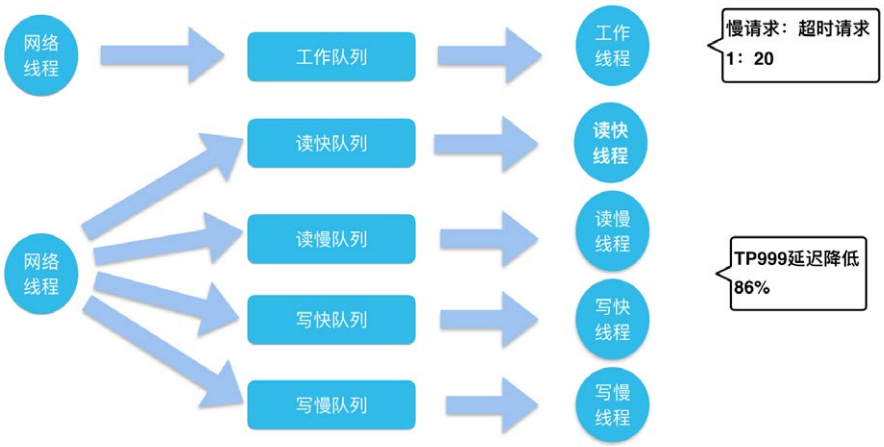
上图是 Cellar 智能迁移架构图。我们把桶的迁移分成了三个状态。第一个状态就是正常的状态，没有任何迁移。如果这时候要把 Slot 2 从 A 节点迁移到 B 节点，A 会给 Slot 2 打一个快照，然后把这个快照全量发到 B 节点上。在迁移数据的时候，B 节点的回包会带回 B 节点的状态。B 的状态包括什么？引擎的压力、网卡流量、队列长度等。A 节点会根据 B 节点的状态调整自己的迁移速度。像 Squirrel 一样，它经

过一段时间调整后，迁移速度会达到一个动态平衡，达到最快速的迁移，同时又尽可能小地影响业务的正常请求。

当 Slot 2 迁移完后，会进入图中 Slot 3 的状态。客户端这时可能还没更新路由表，当它请求到了 A 节点，A 节点会发现客户端请求错了节点，但它不会返回错误，它会把请求代理到 B 节点上，然后把 B 的响应包再返回客户端。同时它会告诉客户端，需要更新一下路由表了，此后客户端就能直接访问到 B 节点。这样就解决了客户端路由更新延迟造成的请求错误。

Cellar 快慢列队

下图上方是一个标准的线程队列模型。网络线程池接收网络流量解析出请求包，然后把请求放到工作队列里，工作线程池会从工作队列取请求来处理，然后把响应包放回网络线程池发出。

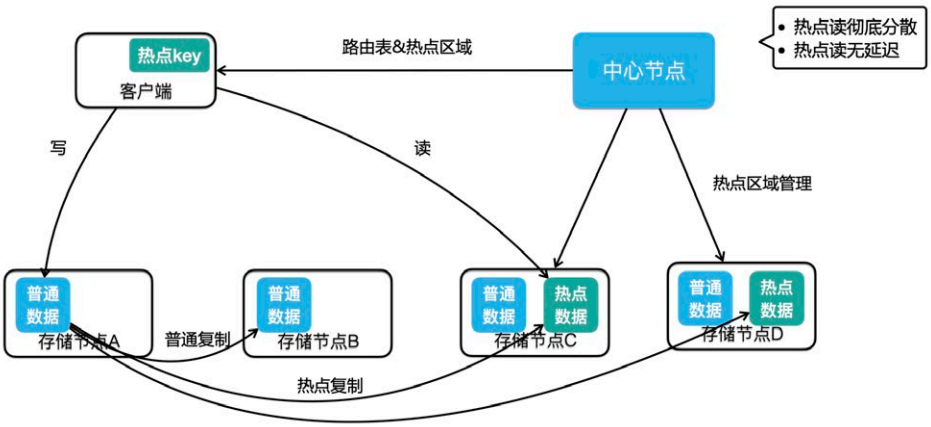


我们分析线上发生的超时案例时发现，一批超时请求当中往往只有一两个请求是引擎处理慢导致的，大部分请求，只是因为在队列等待过久导致整体响应时间过长而超时了。从线上分析来看，真正的慢请求占超时请求的比例只有 1/20。

我们的解法是什么样？很简单，拆线程池、拆队列。我们的网络线程在收到包之后，

会根据它的请求特点，是读还是写，快还是慢，分到四个队列里。读写请求比较好区分，但快慢怎么分开？我们会根据请求的 Key 个数、Value 大小、数据结构元素数等对请求进行快慢区分。然后用对应的四个工作线程池处理对应队列的请求，就实现了快慢读写请求的隔离。这样如果我有一个读的慢请求，不会影响另外三种请求的正常处理。不过这样也会带来一个问题，我们的线程池从一个变成四个，那线程数是不是变成原来的四倍？其实并不是的，我们某个线程池空闲的时候会去帮助其它的线程池处理请求。所以，我们线程池变成了四个，但是线程总数并没有变。我们线上验证中这样的设计能把服务 TP999 的延迟降低 86%，可大幅降低超时率。

Cellar 热点 Key



上图是 Cellar 热点 Key 解决方案的架构图。我们可以看到中心节点加了一个职责，多了热点区域管理，它现在不只负责正常的数据副本分布，还要管理热点数据的分布，图示这个集群在节点 C、D 放了热点区域。我们通过读写流程看一下这个方案是怎么运转的。如果客户端有一个写操作到了 A 节点，A 节点处理完成后，会根据实时的热点统计结果判断写入的 Key 是否为热点。如果这个 Key 是一个热点，那么它会在做集群内复制的同时，还会把这个数据复制有热点区域的节点，也就是图中的 C、D 节点。同时，存储节点在返回结果给客户端时，会告诉客户端，这个 Key 是热点，这时客户端内会缓存这个热点 Key。当客户端有这个 Key 的读请求时，它就会直接

去热点区域做数据的读取。通过这样的方式，我们可以做到只对热点数据做扩容，不像 Squirrel ，要把整个 Slot 迁出来做扩容。有必要的话，中心节点也可以把热点区域放到集群的所有节点上，所有的热点读请求就能均衡的分到所有节点上。另外，通过这种实时的热点数据复制，我们很好地解决了类似客户端缓存热点 KV 方案造成的一致性问题。

发展规划和业界趋势

最后，一起来看看我们项目的规划和业界的技术趋势。这部分内容会按照服务、系统、硬件三层来进行阐述。首先在服务层，主要有三点：

- 第一，Redis Gossip 协议优化。大家都知道 Gossip 协议在集群的规模变大之后，消息量会剧增，它的 Failover 时间也会变得越来越长。所以当集群规模达到 TB 级后，集群的可用性会受到很大的影响，所以我们后面会重点在这方面做一些优化。
- 第二，我们已经在 Cellar 存储节点的数据副本间做了 Raft 复制，可以保证数据强一致，后面我们会在 Cellar 的中心点内部也做一个 Raft 复制，这样就不用依赖于 ZooKeeper 做分布式仲裁、元数据存储了，我们的架构也会变得更加简单、可靠。
- 第三，Squirrel 和 Cellar 虽然都是 KV 存储，但是因为它们是基于不同的开源项目研发的，所以 API 和访问协议不同，我们之后会考虑将 Squirrel 和 Cellar 在 SDK 层做整合，虽然后端会有不同的存储集群，但业务侧可以用一套 SDK 进行访问。

在系统层面，我们正在调研并去落地一些 Kernel Bypass 技术，像 DPDK、SPDK 这种网络和硬盘的用户态 IO 技术。它可以绕过内核，通过轮询机制访问这些设备，可以极大提升系统的 IO 能力。存储作为 IO 密集型服务，性能会获得大幅的提升。

在硬件层面，像支持 RDMA 的智能网卡能大幅降低网络延迟和提升吞吐；还有像 3D XPoint 这样的闪存技术，比如英特尔新发布的 AEP 存储，其访问延迟已经比较接近

内存了，以后闪存跟内存之间的界限也会变得越来越模糊；最后，看一下计算型硬件，比如通过在闪存上加 FPGA 卡，把原本应该 CPU 做的工作，像数据压缩、解压等，下沉到卡上执行，这种硬件能在解放 CPU 的同时，也可以降低服务的响应延迟。

作者简介

泽斌，美团点评高级技术专家，2014 年加入美团。

招聘信息

美团基础技术部存储技术中心长期招聘 C/C++、Go、Java 高级 / 资深工程师和技术专家，欢迎加入美团基础技术部大家庭。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：基础技术部 – 存储技术中心）

Java 中 9 种常见的 CMS GC 问题分析与解决

作者：新宇　湘铭　祥璞

1. 写在前面

| 本文主要针对 Hotspot VM 中“CMS + ParNew”组合的一些使用场景进行总结。重点通过部分源码对根因进行分析以及对排查方法进行总结，排查过程会省略较多，另外本文专业术语较多，有一定的阅读门槛，如未介绍清楚，还请自行查阅相关材料。

| 总字数 2 万左右（不包含代码片段），整体阅读时间约 30min ，文章较长，可以选择你感兴趣的场景进行研究。

1.1 引言

自 Sun 发布 Java 语言以来，开始使用 GC 技术来进行内存自动管理，避免了手动管理带来的悬挂指针（Dangling Pointer）问题，很大程度上提升了开发效率，从此 GC 技术也一举成名。GC 有着非常悠久的历史，1960 年有着“Lisp 之父”和“人工智能之父”之称的 John McCarthy 就在论文中发布了 GC 算法，60 年以来，GC 技术的发展也突飞猛进，但不管是多么前沿的收集器也都是基于三种基本算法的组合或应用，也就是说 GC 要解决的根本问题这么多年一直都没有变过。笔者认为，在不太远的将来，GC 技术依然不会过时，比起日新月异的新技术，GC 这门古典技术更值得我们学习。

目前，互联网上 Java 的 GC 资料要么是主要讲解理论，要么就是针对单一场景的 GC 问题进行了剖析，对整个体系总结的资料少之又少。前车之鉴，后事之师，美团的几位工程师搜集了内部各种 GC 问题的分析文章，并结合个人的理解做了一些总结，希望能起到“抛砖引玉”的作用，文中若有错误之处，还请大家不吝指正。

GC 问题处理能力能不能系统性掌握？一些影响因素都是**互为因果**的问题该怎么分析？比如一个服务 RT 突然上涨，有 GC 耗时增大、线程 Block 增多、慢查询增多、CPU 负载高四个表象，到底哪个是诱因？如何判断 GC 有没有问题？使用 CMS 有哪些常见问题？如何判断根因是什么？如何解决或避免这些问题？阅读完本文，相信你将会对 CMS GC 的问题处理有一个系统性的认知，更能游刃有余地解决这些问题，下面就让我们开始吧！

1.2 概览

想要系统性地掌握 GC 问题处理，笔者这里给出一个学习路径，整体文章的框架也是按照这个结构展开，主要分四大步。

- **建立知识体系：**从 JVM 的内存结构到垃圾收集的算法和收集器，学习 GC 的基础知识，掌握一些常用的 GC 问题分析工具。
- **确定评价指标：**了解基本 GC 的评价方法，摸清如何设定独立系统的指标，以及在业务场景中判断 GC 是否存在问题的手段。
- **场景调优实践：**运用掌握的知识和系统评价指标，分析与解决九种 CMS 中常见 GC 问题场景。
- **总结优化经验：**对整体过程做总结并提出笔者的几点建议，同时将总结到的经验完善到知识体系之中。

2. GC 基础

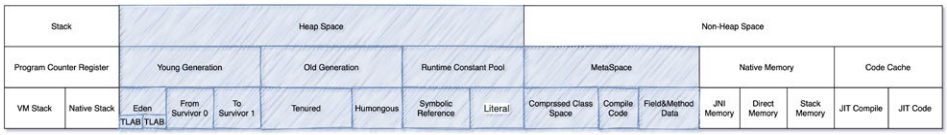
在正式开始前，先做些简要铺垫，介绍下 JVM 内存划分、收集算法、收集器等常用概念介绍，基础比较好的同学可以直接跳过这部分。

2.1 基础概念

- **GC**：GC 本身有三种语义，下文需要根据具体场景带入不同的语义：
  - **Garbage Collection**：垃圾收集技术，名词。
  - **Garbage Collector**：垃圾收集器，名词。
  - **Garbage Collecting**：垃圾收集动作，动词。
- **Mutator**：生产垃圾的角色，也就是我们的应用程序，垃圾制造者，通过 Allocator 进行 allocate 和 free。
- **TLAB**：Thread Local Allocation Buffer 的简写，基于 CAS 的独享线程（Mutator Threads）可以优先将对象分配在 Eden 中的一块内存，因为是 Java 线程独享的内存区没有锁竞争，所以分配速度更快，每个 TLAB 都是一个线程独享的。
- **Card Table**：中文翻译为卡表，主要是用来标记卡页的状态，每个卡表项对应一个卡页。当卡页中一个对象引用有写操作时，写屏障将会标记对象所在的卡表状态改为 dirty，卡表的本质是用来解决跨代引用的问题。具体怎么解决的可以参考 StackOverflow 上的这个问题 how-actually-card-table-and-writer-barrier-works，或者研读一下 cardTableRS.app 中的源码。

2.2 JVM 内存划分

从 JCP（Java Community Process）的官网中可以看到，目前 Java 版本最新已经到了 Java 16，未来的 Java 17 以及现在的 Java 11 和 Java 8 是 LTS 版本，JVM 规范也在随着迭代在变更，由于本文主要讨论 CMS，此处还是放 Java 8 的内存结构。



GC 主要工作在 Heap 区和 MetaSpace 区（上图蓝色部分），在 Direct Memory 中，如果使用的是 DirectByteBuffer，那么在分配内存不够时则是 GC 通过 `Cleaner#-`

`clean` 间接管理。

任何自动内存管理系统都会面临的步骤：为新对象分配空间，然后收集垃圾对象空间，下面我们就展开介绍一下这些基础知识。

2.3 分配对象

Java 中对象地址操作主要使用 Unsafe 调用了 C 的 allocate 和 free 两个方法，分配方法有两种：

- **空闲链表（free list）：** 通过额外的存储记录空闲的地址，将随机 IO 变为顺序 IO，但带来了额外的空间消耗。
- **碰撞指针（bump pointer）：** 通过一个指针作为分界点，需要分配内存时，仅需把指针往空闲的一端移动与对象大小相等的距离，分配效率较高，但使用场景有限。

2.4 收集对象

2.4.1 识别垃圾

- **引用计数法（Reference Counting）：** 对每个对象的引用进行计数，每当有一个地方引用它时计数器 +1、引用失效则 −1，引用的计数放到对象头中，大于 0 的对象被认为是存活对象。虽然循环引用的问题可通过 Recycler 算法解决，但是在多线程环境下，引用计数变更也要进行昂贵的同步操作，性能较低，早期的编程语言会采用此算法。
- **可达性分析，又称引用链法（Tracing GC）：** 从 GC Root 开始进行对象搜索，可以被搜索到的对象即为可达对象，此时还不足以判断对象是否存活 / 死亡，需要经过多次标记才能更加准确地确定，整个连通图之外的对象便可以作为垃圾被回收掉。目前 Java 中主流的虚拟机均采用此算法。

备注：引用计数法是可以处理循环引用问题的，下次面试时不要再这么说啦 ~ ~

2.4.2 收集算法

自从有自动内存管理出现之时就有的一些收集算法，不同的收集器也是在不同场景下进行组合。

- **Mark-Sweep（标记 - 清除）：** 回收过程主要分为两个阶段，第一阶段为追踪（Tracing）阶段，即从 GC Root 开始遍历对象图，并标记（Mark）所遇到的每个对象，第二阶段为清除（Sweep）阶段，即回收器检查堆中每一个对象，并将所有未被标记的对象进行回收，整个过程不会发生对象移动。整个算法在不同的实现中会使用三色抽象（Tricolour Abstraction）、位图标记（BitMap）等技术来提高算法的效率，存活对象较多时较高效。
- **Mark-Compact（标记 - 整理）：** 这个算法的主要目的就是解决在非移动式回收器中都会存在的碎片化问题，也分为两个阶段，第一阶段与 Mark-Sweep 类似，第二阶段则会对存活对象按照整理顺序（Compaction Order）进行整理。主要实现有双指针（Two-Finger）回收算法、滑动回收（Lisp2）算法和引线整理（Threaded Compaction）算法等。
- **Copying（复制）：** 将空间分为两个大小相同的 From 和 To 两个半区，同一时间只会使用其中一个，每次进行回收时将一个半区的存活对象通过复制的方式转移到另一个半区。有递归（Robert R. Fenichel 和 Jerome C. Yochelson 提出）和迭代（Cheney 提出）算法，以及解决了前两者递归栈、缓存行等问题的近似优先搜索算法。复制算法可以通过碰撞指针的方式进行快速地分配内存，但是也存在着空间利用率不高的缺点，另外就是存活对象比较大时复制的成本比较高。

三种算法在是否移动对象、空间和时间方面的一些对比，假设存活对象数量为 *L*、堆空间大小为 *H*，则：

	移动对象	空间开销	时间开销
Mark-Sweep	否	低（有碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，sweep 阶段与整堆大小成正比 *O(H)*
Mark-Compact	是	低（无碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，compaction 阶段与存活对象的大小成正比 *O(L)*
Copying	是	高	与存活对象大小成正比 *O(L)*

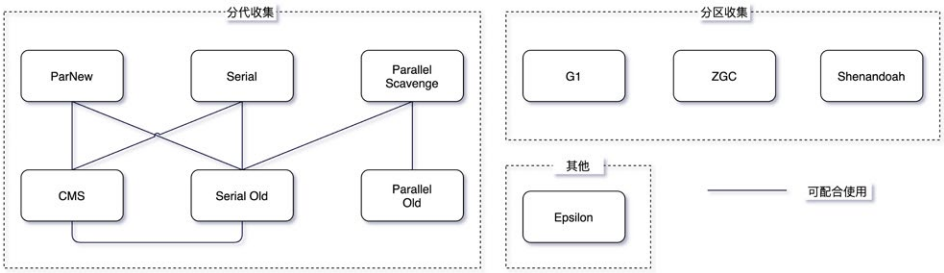
把 mark、sweep、compaction、copying 这几种动作的耗时放在一起看，大致有这样的关系：

$$\begin{cases} compaction \geq copying > mark > sweep \\ mark + sweep > copying \end{cases}$$

虽然 compaction 与 copying 都涉及移动对象，但取决于具体算法，compaction 可能要先计算一次对象的目标地址，然后修正指针，最后再移动对象。copying 则可以把这几件事情合为一体来做，所以可以快一些。另外，还需要留意 GC 带来的开销不能只看 Collector 的耗时，还得看 Allocator 。如果能保证内存没碎片，分配就可以用 pointer bumping 方式，只需要挪一个指针就完成了分配，非常快。而如果内存有碎片就得用 freelist 之类的方式管理，分配速度通常会慢一些。

2.5 收集器

目前在 Hotspot VM 中主要有分代收集和分区收集两大类，具体可以看下面的这个图，不过未来会逐渐向分区收集发展。在美团内部，有部分业务尝试用了 ZGC（感兴趣的同学可以学习下这篇文章 新一代垃圾回收器 ZGC 的探索与实践），其余基本都停留在 CMS 和 G1 上。另外在 JDK11 后提供了一个不执行任何垃圾回收动作的回收器 Epsilon（A No-Op Garbage Collector）用作性能分析。另外一个就是 Azul 的 Zing JVM，其 C4（Concurrent Continuously Compacting Collector）收集器也在业内有一定的影响力。



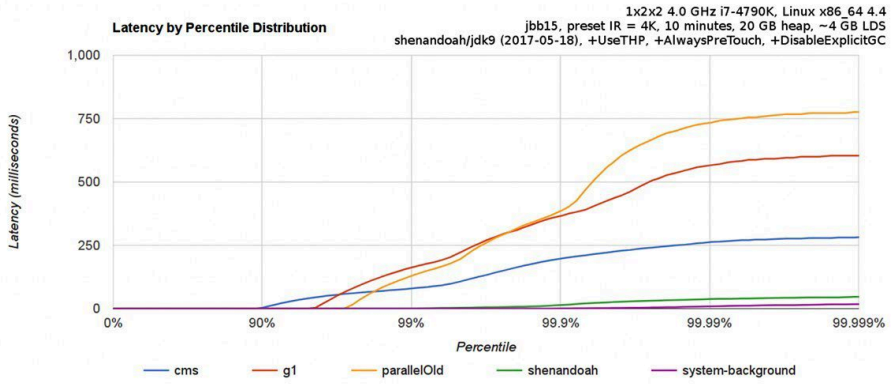
备注：值得一提的是，早些年国内 GC 技术的布道者 RednaxelaFX（江湖人称 R 大）也曾就职于 Azul，本文的一部分材料也参考了他的一些文章。

2.5.1 分代收集器

- **ParNew：**一款多线程的收集器，采用复制算法，主要工作在 Young 区，可以通过 `-XX:ParallelGCThreads` 参数来控制收集的线程数，整个过程都是 STW 的，常与 CMS 组合使用。
- **CMS：**以获取最短回收停顿时间为目标，采用“标记－清除”算法，分 4 大步进行垃圾收集，其中初始标记和重新标记会 STW ，多数应用于互联网站或者 B/S 系统的服务器端上，JDK9 被标记弃用，JDK14 被删除，详情可见 JEP 363。

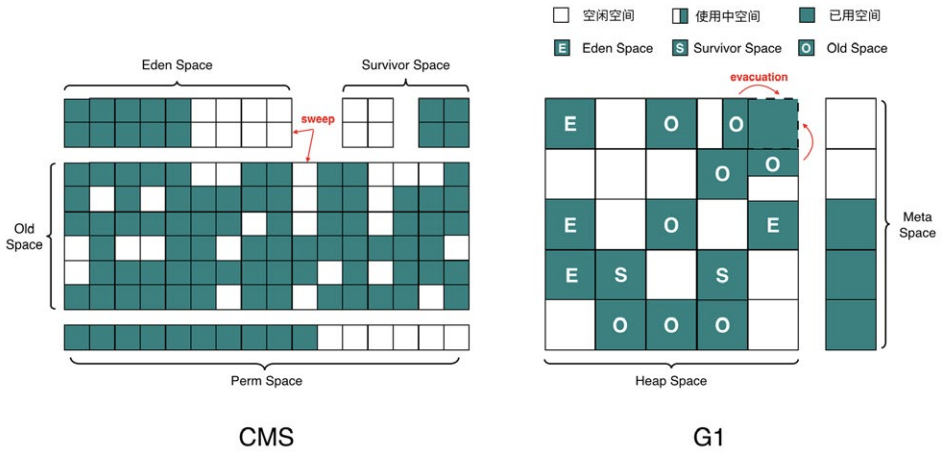
2.5.2 分区收集器

- **G1：**一种服务器端的垃圾收集器，应用在多处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能地满足垃圾收集暂停时间的要求。
- **ZGC：**JDK11 中推出的一款低延迟垃圾回收器，适用于大内存低延迟服务的内存管理和回收，SPECjbb 2015 基准测试，在 128G 的大堆下，最大停顿时间才 1.68 ms，停顿时间远胜于 G1 和 CMS。
- **Shenandoah：**由 Red Hat 的一个团队负责开发，与 G1 类似，基于 Region 设计的垃圾收集器，但不需要 Remember Set 或者 Card Table 来记录跨 Region 引用，停顿时间和堆的大小没有任何关系。停顿时间与 ZGC 接近，下图为与 CMS 和 G1 等收集器的 benchmark。



2.5.3 常用收集器

目前使用最多的是 CMS 和 G1 收集器，二者都有分代的概念，主要内存结构如下：



2.5.4 其他收集器

以上仅列出常见收集器，除此之外还有很多，如 Metronome、Stopless、Staccato、Chicken、Clover 等实时回收器，Sapphire、Compressor、Pauseless 等并发复制 / 整理回收器，Doligez-Leroy-Conthier 等标记整理回收器，由于篇幅原因，不在此一一介绍。

2.6 常用工具

工欲善其事，必先利其器，此处列出一些笔者常用的工具，具体情况大家可以自由选择，本文的问题都是使用这些工具来定位和分析的。

2.6.1 命令行终端

- 标准终端类：jps、jinfo、jstat、jstack、jmap
- 功能整合类：jcmd、vjtools、arthas、greys

2.6.2 可视化界面

- 简易：JConsole、JVisualvm、HA、GCHisto、GCViewer
- 进阶：MAT、JProfiler

命令行推荐 arthas ，可视化界面推荐 JProfiler，此外还有一些在线的平台 gceasy、heaphero、fastthread ，美团内部的 Scalpel（一款自研的 JVM 问题诊断工具，暂时未开源）也比较好用。

3. GC 问题判断

在做 GC 问题排查和优化之前，我们需要先来明确下到底是不是 GC 直接导致的问题，或者应用代码导致的 GC 异常，最终出现问题。

3.1 判断 GC 有没有问题?

3.1.1 设定评价标准

评判 GC 的两个核心指标：

- **延迟（Latency）：** 也可以理解为最大停顿时间，即垃圾收集过程中一次 STW 的最长时间，越短越好，一定程度上可以接受频次的增大，GC 技术的主要发展方向。
- **吞吐量（Throughput）：** 应用系统的生命周期内，由于 GC 线程会占用 Mutator 当前可用的 CPU 时钟周期，吞吐量即为 Mutator 有效花费的时间占

系统总运行时间的百分比，例如系统运行了 100 min，GC 耗时 1 min，则系统吞吐量为 99%，吞吐量优先的收集器可以接受较长的停顿。

目前各大互联网公司的系统基本都更追求低延时，避免一次 GC 停顿的时间过长对用户体验造成损失，衡量指标需要结合一下应用服务的 SLA，主要如下两点来判断：

$$t_{stw} \le t_{tp9999} \quad (t_{stw}\text{为单次停顿时间，}t_{tp9999}\text{为应用4个9耗时})$$

$$(1 - \frac{\sum_{k=1}^{n} c_k \bar{t}_k}{T_n}) \times 100\% \ge 99.99\% \quad (c_k\text{为一个时间周期内}gc\text{次数，}\bar{t}_k\text{为该时间周期平均停顿时间，}T_n\text{为总时间})$$

简而言之，即为**一次停顿的时间不超过应用服务的 TP9999，GC 的吞吐量不小于 99.99%**。举个例子，假设某个服务 A 的 TP9999 为 80 ms，平均 GC 停顿为 30 ms，那么该服务的最大停顿时间最好不要超过 80 ms，GC 频次控制在 5 min 以上一次。如果满足不了，那就需要调优或者通过更多资源来进行并联冗余。（大家可以先停下来，看看监控平台上面的 gc.meantime 分钟级别指标，如果超过了 6 ms 那单机 GC 吞吐量就达不到 4 个 9 了。）

备注：除了这两个指标之外还有 Footprint（资源量大小测量）、反应速度等指标，互联网这种实时系统追求低延迟，而很多嵌入式系统则追求 Footprint。

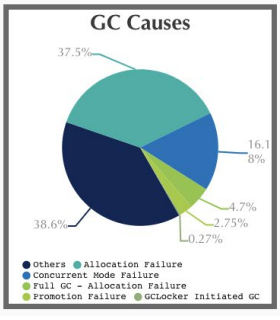
3.1.2 读懂 GC Cause

拿到 GC 日志，我们就可以简单分析 GC 情况了，通过一些工具，我们可以比较直观地看到 Cause 的分布情况，如下图就是使用 gceasy 绘制的图表：

GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Others	3	n/a	n/a	1 min 55 sec 640 ms	38.6%
Allocation Failure	1638	68.6 ms	18 sec 200 ms	1 min 52 sec 340 ms	37.5%
Concurrent Mode Failure	8	6 sec 59 ms	11 sec 960 ms	48 sec 470 ms	16.18%
Full GC - Allocation Failure	5	2 sec 816 ms	3 sec 100 ms	14 sec 80 ms	4.7%
Promotion Failure	1	8 sec 240 ms	8 sec 240 ms	8 sec 240 ms	2.75%
GCLocker Initiated GC	13	62.3 ms	130 ms	810 ms	0.27%
Total	1668	n/a	n/a	4 min 59 sec 580 ms	100.0%



如上图所示，我们很清晰的就能知道是什么原因引起的 GC，以及每次的时间花费情况，但是要分析 GC 的问题，先要读懂 GC Cause，即 JVM 什么样的条件下选择进行 GC 操作，具体 Cause 的分类可以看一下 Hotspot 源码：src/share/vm/gc/shared/gcCause.hpp 和 src/share/vm/gc/shared/gcCause.cpp 中。

```
const char* GCCause::to_string(GCCause::Cause cause) {
  switch (cause) {
    case _java_lang_system_gc:
      return "System.gc()";

    case _full_gc_alot:
      return "FullGCAlot";

    case _scavenge_alot:
      return "ScavengeAlot";

    case _allocation_profiler:
      return "Allocation Profiler";

    case _jvmti_force_gc:
      return "JvmtiEnv ForceGarbageCollection";

    case _gc_locker:
      return "GCLocker Initiated GC";

    case _heap_inspection:
      return "Heap Inspection Initiated GC";

    case _heap_dump:
      return "Heap Dump Initiated GC";

    case _wb_young_gc:
      return "WhiteBox Initiated Young GC";

    case _wb_conc_mark:
      return "WhiteBox Initiated Concurrent Mark";

    case _wb_full_gc:
      return "WhiteBox Initiated Full GC";

    case _no_gc:
      return "No GC";

    case _allocation_failure:
      return "Allocation Failure";
```

```
    case _tenured_generation_full:
      return "Tenured Generation Full";

    case _metadata_GC_threshold:
      return "Metadata GC Threshold";

    case _metadata_GC_clear_soft_refs:
      return "Metadata GC Clear Soft References";

    case _cms_generation_full:
      return "CMS Generation Full";

    case _cms_initial_mark:
      return "CMS Initial Mark";

    case _cms_final_remark:
      return "CMS Final Remark";

    case _cms_concurrent_mark:
      return "CMS Concurrent Mark";

    case _old_generation_expanded_on_last_scavenge:
      return "Old Generation Expanded On Last Scavenge";

    case _old_generation_too_full_to_scavenge:
      return "Old Generation Too Full To Scavenge";

    case _adaptive_size_policy:
      return "Ergonomics";

    case _g1_inc_collection_pause:
      return "G1 Evacuation Pause";

    case _g1_humongous_allocation:
      return "G1 Humongous Allocation";

    case _dcmd_gc_run:
      return "Diagnostic Command";

    case _last_gc_cause:
      return "ILLEGAL VALUE - last gc cause - ILLEGAL VALUE";

    default:
      return "unknown GCCause";
  }
  ShouldNotReachHere();
}
```

重点需要关注的几个 GC Cause：

- **System.gc()：**手动触发 GC 操作。
- **CMS：**CMS GC 在执行过程中的一些动作，重点关注 CMS Initial Mark 和 CMS Final Remark 两个 STW 阶段。
- **Promotion Failure：**Old 区没有足够的空间分配给 Young 区晋升的对象（即使总可用内存足够大）。
- **Concurrent Mode Failure：**CMS GC 运行期间，Old 区预留的空间不足以分配给新的对象，此时收集器会发生退化，严重影响 GC 性能，下面的一个案例即为这种场景。
- **GCLocker Initiated GC：**如果线程执行在 JNI 临界区时，刚好需要进行 GC，此时 GC Locker 将会阻止 GC 的发生，同时阻止其他线程进入 JNI 临界区，直到最后一个线程退出临界区时触发一次 GC。

什么时机使用这些 Cause 触发回收，大家可以看一下 CMS 的代码，这里就不讨论了，具体在 /src/hotspot/share/gc/cms/concurrentMarkSweepGeneration.cpp 中。

```
bool CMSCollector::shouldConcurrentCollect() {
  LogTarget(Trace, gc) log;

  if (_full_gc_requested) {
    log.print("CMSCollector: collect because of explicit  gc request
(or GCLocker)");
    return true;
  }

  FreelistLocker x(this);
  // ------------------------------------------------------------------
--
  // Print out lots of information which affects the initiation of
  // a collection.
  if (log.is_enabled() && stats().valid()) {
    log.print("CMSCollector shouldConcurrentCollect: ");

    LogStream out(log);
    stats().print_on(&out);
```

```
    log.print("time_until_cms_gen_full %3.7f", stats().time_until_cms_
gen_full());
    log.print("free=" SIZE_FORMAT, _cmsGen->free());
    log.print("contiguous_available=" SIZE_FORMAT, _cmsGen->contiguous_
available());
    log.print("promotion_rate=%g", stats().promotion_rate());
    log.print("cms_allocation_rate=%g", stats().cms_allocation_rate());
    log.print("occupancy=%3.7f", _cmsGen->occupancy());
    log.print("initiatingOccupancy=%3.7f", _cmsGen->initiating_
occupancy());
    log.print("cms_time_since_begin=%3.7f", stats().cms_time_since_
begin());
    log.print("cms_time_since_end=%3.7f", stats().cms_time_since_
end());
    log.print("metadata initialized %d", MetaspaceGC::should_concurrent_
collect());
  }
  // ------------------------------------------------------------------
--

  // If the estimated time to complete a cms collection (cms_
duration())
  // is less than the estimated time remaining until the cms
generation
  // is full, start a collection.
  if (!UseCMSInitiatingOccupancyOnly) {
    if (stats().valid()) {
      if (stats().time_until_cms_start() == 0.0) {
        return true;
      }
    } else {

      if (_cmsGen->occupancy() >= _bootstrap_occupancy) {
        log.print(" CMSCollector: collect for bootstrapping
statistics: occupancy = %f, boot occupancy = %f",
                  _cmsGen->occupancy(), _bootstrap_occupancy);
        return true;
      }
    }
  }
  if (_cmsGen->should_concurrent_collect()) {
    log.print("CMS old gen initiated");
    return true;
  }

  CMSHeap* heap = CMSHeap::heap();
  if (heap->incremental_collection_will_fail(true /* consult_young */))
{
    log.print("CMSCollector: collect because incremental collection
will fail ");
```

```
    return true;
  }

  if (MetaspaceGC::should_concurrent_collect()) {
    log.print("CMSCollector: collect for metadata allocation ");
    return true;
  }

  // CMSTriggerInterval starts a CMS cycle if enough time has passed.
  if (CMSTriggerInterval >= 0) {
    if (CMSTriggerInterval == 0) {
      // Trigger always
      return true;
    }

    // Check the CMS time since begin (we do not check the stats
validity
    // as we want to be able to trigger the first CMS cycle as well)
    if (stats().cms_time_since_begin() >= (CMSTriggerInterval / ((double)
MILLIUNITS))) {
      if (stats().valid()) {
        log.print("CMSCollector: collect because of trigger interval
(time since last begin %3.7f secs)",
                  stats().cms_time_since_begin());
      } else {
        log.print("CMSCollector: collect because of trigger interval
(first collection)");
      }
      return true;
    }
  }

  return false;
}
```

3.2 判断是不是 GC 引发的问题?

到底是结果（现象）还是原因，在一次 GC 问题处理的过程中，如何判断是 GC 导致的故障，还是系统本身引发 GC 问题。这里继续拿在本文开头提到的一个 Case：“GC 耗时增大、线程 Block 增多、慢查询增多、CPU 负载高等四个表象，如何判断哪个是根因?”，笔者这里根据自己的经验大致整理了四种判断方法供参考：

- **时序分析：**先发生的事件是根因的概率更大，通过监控手段分析各个指标的异常时间点，还原事件时间线，如先观察到 CPU 负载高（要有足够的时间

Gap），那么整个问题影响链就可能是：CPU 负载高 -> 慢查询增多 -> GC 耗时增大 -> 线程 Block 增多 -> RT 上涨。

- **概率分析：**使用统计概率学，结合历史问题的经验进行推断，由近到远按类型分析，如过往慢查的问题比较多，那么整个问题影响链就可能是：慢查询增多 -> GC 耗时增大 -> CPU 负载高 -> 线程 Block 增多 -> RT 上涨。
- **实验分析：**通过故障演练等方式对问题现场进行模拟，触发其中部分条件（一个或多个），观察是否会发生问题，如只触发线程 Block 就会发生问题，那么整个问题影响链就可能是：线程 Block 增多 -> CPU 负载高 -> 慢查询增多 -> GC 耗时增大 -> RT 上涨。
- **反证分析：**对其中某一表象进行反证分析，即判断表象的发不发生跟结果是否有相关性，例如我们从整个集群的角度观察到某些节点慢查和 CPU 都正常，但也出了问题，那么整个问题影响链就可能是：GC 耗时增大 -> 线程 Block 增多 -> RT 上涨。

不同的根因，后续的分析方法是完全不同的。如果是 CPU 负载高那可能需要用火焰图看下热点、如果是慢查询增多那可能需要看下 DB 情况、如果是线程 Block 引起那可能需要看下锁竞争的情况，最后如果各个表象证明都没有问题，那可能 GC 确实存在问题，可以继续分析 GC 问题了。

3.3 问题分类导读

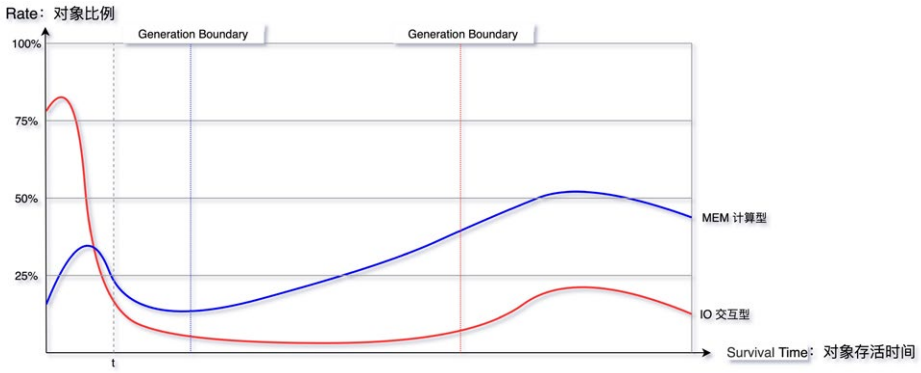
3.3.1 Mutator 类型

Mutator 的类型根据对象存活时间比例图来看主要分为两种，在弱分代假说中也提到类似的说法，如下图所示“Survival Time”表示对象存活时间，“Rate”表示对象分配比例：

- **IO 交互型：**互联网上目前大部分的服务都属于该类型，例如分布式 RPC、MQ、HTTP 网关服务等，对内存要求并不大，大部分对象在 TP9999 的时间内都会死亡，Young 区越大越好。

- **MEM 计算型：**主要是分布式数据计算 Hadoop，分布式存储 HBase、Cassandra，自建的分布式缓存等，对内存要求高，对象存活时间长，Old 区越大越好。

当然，除了二者之外还有介于两者之间的场景，本篇文章主要讨论第一种情况。对象 Survival Time 分布图，对我们设置 GC 参数有着非常重要的指导意义，如下图就可以简单推算分代的边界。



3.3.2 GC 问题分类

笔者选取了九种不同类型的 GC 问题，覆盖了大部分场景，如果有更好的场景，欢迎在评论区给出。

- **Unexpected GC：**意外发生的 GC，实际上不需要发生，我们可以通过一些手段去避免。
  - **Space Shock：**空间震荡问题，参见“场景一：动态扩容引起的空间震荡”。
  - **Explicit GC：**显示执行 GC 问题，参见“场景二：显式 GC 的去与留”。
- **Partial GC：**部分收集操作的 GC，只对某些分代 / 分区进行回收。
  - **Young GC：**分代收集里面的 Young 区收集动作，也可以叫做 Minor GC。
    - **ParNew：**Young GC 频繁，参见“场景四：过早晋升”。
  - **Old GC：**分代收集里面的 Old 区收集动作，也可以叫做 Major GC，有些也会叫做 Full GC，但其实这种叫法是不规范的，在 CMS 发生 Fore-

ground GC 时才是 Full GC，CMSScavengeBeforeRemark 参数也只是在 Remark 前触发一次 Young GC。

  - **CMS：** Old GC 频繁，参见“场景五：CMS Old GC 频繁”。
  - **CMS：** Old GC 不频繁但单次耗时大，参见“场景六：单次 CMS Old GC 耗时长”。
- **Full GC：** 全量收集的 GC，对整个堆进行回收，STW 时间会比较长，一旦发生，影响较大，也可以叫做 Major GC，参见“场景七：内存碎片 & 收集器退化”。
- **MetaSpace：** 元空间回收引发问题，参见“场景三：MetaSpace 区 OOM”。
- **Direct Memory：** 直接内存（也可以称作为堆外内存）回收引发问题，参见“场景八：堆外内存 OOM”。
- **JNI：** 本地 Native 方法引发问题，参见“场景九：JNI 引发的 GC 问题”。

3.3.3 排查难度

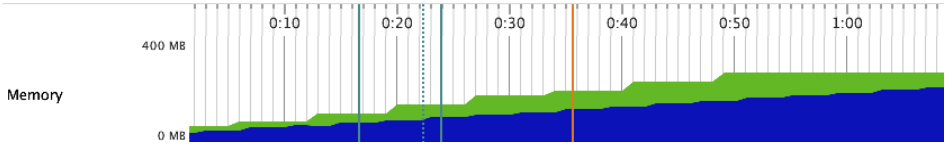
一个问题的**解决难度跟它的常见程度成反比**，大部分我们都可以通过各种搜索引擎找到类似的问题，然后用同样的手段尝试去解决。当一个问题在各种网站上都找不到相似的问题时，那么可能会有两种情况，一种这不是一个问题，另一种就是遇到一个隐藏比较深的问题，遇到这种问题可能就要深入到源码级别去调试了。以下 GC 问题场景，排查难度从上到下依次递增。

4. 常见场景分析与解决

4.1 场景一：动态扩容引起的空间震荡

4.1.1 现象

服务**刚刚启动时 GC 次数较多**，最大空间剩余很多但是依然发生 GC，这种情况我们可以通过观察 GC 日志或者通过监控工具来观察堆的空间变化情况即可。GC Cause 一般为 Allocation Failure，且在 GC 日志中会观察到经历一次 GC ，堆内各个空间的大小会被调整，如下图所示：



4.1.2 原因

在 JVM 的参数中 `-Xms` 和 `-Xmx` 设置的不一致，在初始化时只会初始 `-Xms` 大小的空间存储信息，每当空间不够用时再向操作系统申请，这样的话必然要进行一次 GC。具体是通过 `ConcurrentMarkSweepGeneration::compute_new_size()` 方法计算新的空间大小：

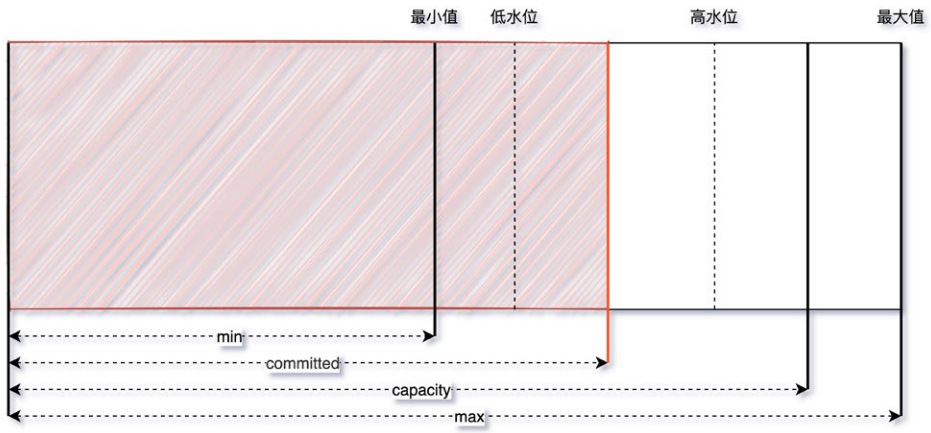
```
void ConcurrentMarkSweepGeneration::compute_new_size() {
  assert_locked_or_safepoint(Heap_lock);

  // If incremental collection failed, we just want to expand
  // to the limit.
  if (incremental_collection_failed()) {
    clear_incremental_collection_failed();
    grow_to_reserved();
    return;
  }

  // The heap has been compacted but not reset yet.
  // Any metric such as free() or used() will be incorrect.

  CardGeneration::compute_new_size();

  // Reset again after a possible resizing
  if (did_compact()) {
    cmsSpace()->reset_after_compaction();
  }
}
```

另外，如果空间剩余很多时也会进行缩容操作，JVM 通过 `-XX:MinHeapFreeRatio` 和 `-XX:MaxHeapFreeRatio` 来控制扩容和缩容的比例，调节这两个值也可以控制伸缩的时机，例如扩容便是使用 `GenCollectedHeap::expand_heap_and_allocate()` 来完成的，代码如下：

```
HeapWord* GenCollectedHeap::expand_heap_and_allocate(size_t size, bool
is_tlab) {
```

```
  HeapWord* result = NULL;
  if (_old_gen->should_allocate(size, is_tlab)) {
    result = _old_gen->expand_and_allocate(size, is_tlab);
  }
  if (result == NULL) {
    if (_young_gen->should_allocate(size, is_tlab)) {
      result = _young_gen->expand_and_allocate(size, is_tlab);
    }
  }
  assert(result == NULL || is_in_reserved(result), "result not in
heap");
  return result;
}
```

整个伸缩的模型理解可以看这个图，当 committed 的空间大小超过了低水位 / 高水位的大小，capacity 也会随之调整：



4.1.3 策略

定位：观察 CMS GC 触发时间点 Old/MetaSpace 区的 committed 占比是不是一个固定的值，或者像上文提到的观察总的内存使用率也可以。

解决：尽量**将成对出现的空间大小配置参数设置成固定的**，如 `-Xms` 和 `-Xmx`，`-XX:-MaxNewSize` 和 `-XX:NewSize`，`-XX:MetaSpaceSize` 和 `-XX:MaxMetaSpace-Size` 等。

4.1.4 小结

一般来说，我们需要保证 Java 虚拟机的堆是稳定的，确保 `-Xms` 和 `-Xmx` 设置的是一个值（即初始值和最大值一致），获得一个稳定的堆，同理在 MetaSpace 区也有类似的问题。不过在不追求停顿时间的情况下震荡的空间也是有利的，可以动态地伸缩以节省空间，例如作为富客户端的 Java 应用。

这个问题虽然初级，但是发生的概率还真不小，尤其是在一些规范不太健全的情况下。

4.2 场景二：显式 GC 的去与留

4.2.1 现象

除了扩容缩容会触发 CMS GC 之外，还有 Old 区达到回收阈值、MetaSpace 空间不足、Young 区晋升失败、大对象担保失败等几种触发条件，如果这些情况都没有发生却触发了 GC？这种情况有可能是代码中手动调用了 System.gc 方法，此时可以找到 GC 日志中的 GC Cause 确认下。那么这种 GC 到底有没有问题，翻看网上的一些资料，有人说可以添加 `-XX:+DisableExplicitGC` 参数来避免这种 GC，也有人说不能加这个参数，加了就会影响 Native Memory 的回收。先说结论，笔者这里建议保留 System.gc，那为什么要保留？我们一起来分析下。

4.2.2 原因

找到 System.gc 在 Hotspot 中的源码，可以发现增加 `-XX:+DisableExplicitGC` 参数后，这个方法变成了一个空方法，如果没有加的话便会调用 `Universe::heap()::collect` 方法，继续跟进到这个方法中，发现 System.gc 会引发一次 STW 的 Full GC，对整个堆做收集。

```
JVM_ENTRY_NO_ENV(void, JVM_GC(void))
  JVMWrapper("JVM_GC");
  if (!DisableExplicitGC) {
    Universe::heap()->collect(GCCause::_java_lang_system_gc);
  }
JVM_END
```

```
void GenCollectedHeap::collect(GCCause::Cause cause) {
  if (cause == GCCause::_wb_young_gc) {
    // Young collection for the WhiteBox API.
    collect(cause, YoungGen);
  } else {
#ifdef ASSERT
  if (cause == GCCause::_scavenge_alot) {
    // Young collection only.
    collect(cause, YoungGen);
  } else {
    // Stop-the-world full collection.
    collect(cause, OldGen);
  }
#else
    // Stop-the-world full collection.
    collect(cause, OldGen);
#endif
  }
}
```

保留 System.gc

此处补充一个知识点，**CMS GC 共分为 Background 和 Foreground 两种模式**，前者就是我们常规理解中的并发收集，可以不影响正常的业务线程运行，但 Foreground Collector 却有很大的差异，他会进行一次压缩式 GC。此压缩式 GC 使用的是跟 Serial Old GC 一样的 Lisp2 算法，其使用 Mark-Compact 来做 Full GC，一般称之为 MSC（Mark-Sweep-Compact），它收集的范围是 Java 堆的 Young 区和 Old 区以及 MetaSpace。由上面的算法章节中我们知道 compact 的代价是巨大的，那么使用 Foreground Collector 时将会带来非常长的 STW。如果在应用程序中 System.gc 被频繁调用，那就非常危险了。

去掉 System.gc

如果禁用掉的话就会带来另外一个内存泄漏问题，此时就需要说一下 DirectByteBuffer，它有着零拷贝等特点，被 Netty 等各种 NIO 框架使用，会使用到堆外内存。堆内存由 JVM 自己管理，堆外内存必须要手动释放，DirectByteBuffer 没有 Finalizer，它的 Native Memory 的清理工作是通过 `sun.misc.Cleaner` 自动完成

的，是一种基于 PhantomReference 的清理工具，比普通的 Finalizer 轻量些。

为 DirectByteBuffer 分配空间过程中会显式调用 System.gc ，希望通过 Full GC 来强迫已经无用的 DirectByteBuffer 对象释放掉它们关联的 Native Memory，下面为代码实现：

```
// These methods should be called whenever direct memory is allocated or
// freed.  They allow the user to control the amount of direct memory
// which a process may access.  All sizes are specified in bytes.
static void reserveMemory(long size) {

    synchronized (Bits.class) {
        if (!memoryLimitSet && VM.isBooted()) {
            maxMemory = VM.maxDirectMemory();
            memoryLimitSet = true;
        }
        if (size <= maxMemory - reservedMemory) {
            reservedMemory += size;
            return;
        }
    }

    System.gc();
    try {
        Thread.sleep(100);
    } catch (InterruptedException x) {
        // Restore interrupt status
        Thread.currentThread().interrupt();
    }
    synchronized (Bits.class) {
        if (reservedMemory + size > maxMemory)
            throw new OutOfMemoryError("Direct buffer memory");
        reservedMemory += size;
    }

}
```

HotSpot VM 只会在 Old GC 的时候才会对 Old 中的对象做 Reference Processing，而在 Young GC 时只会对 Young 里的对象做 Reference Processing。Young 中的 DirectByteBuffer 对象会在 Young GC 时被处理，也就是说，做 CMS GC 的话会对 Old 做 Reference Processing，进而能触发 Cleaner 对已死的 DirectByteBuffer 对象做清理工作。但如果很长一段时间里没做过 GC 或者只做了

Young GC 的话则不会在 Old 触发 Cleaner 的工作，那么就可能让本来已经死亡，但已经晋升到 Old 的 DirectByteBuffer 关联的 Native Memory 得不到及时释放。这几个实现特征使得依赖于 System.gc 触发 GC 来保证 DirectByteMemory 的清理工作能及时完成。如果打开了 `-XX:+DisableExplicitGC`，清理工作就可能得不到及时完成，于是就有发生 Direct Memory 的 OOM。

4.2.3 策略

通过上面的分析看到，无论是保留还是去掉都会有一定的风险点，不过目前互联网中的 RPC 通信会大量使用 NIO，所以笔者在这里建议保留。此外 JVM 还提供了 `-XX:+ExplicitGCInvokesConcurrent` 和 `-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses` 参数来将 System.gc 的触发类型从 Foreground 改为 Background，同时 Background 也会做 Reference Processing，这样的话就能大幅降低了 STW 开销，同时也不会发生 NIO Direct Memory OOM。

4.2.4 小结

不止 CMS，在 G1 或 ZGC 中开启 `ExplicitGCInvokesConcurrent` 模式，都会采用高性能的并发收集方式进行收集，不过还是建议在代码规范方面也要做好约束，规范好 System.gc 的使用。

P.S. HotSpot 对 System.gc 有特别处理，最主要的地方体现在一次 System.gc 是否与普通 GC 一样会触发 GC 的统计 / 阈值数据的更新，HotSpot 里的许多 GC 算法都带有自适应的功能，会根据先前收集的效率来决定接下来的 GC 中使用的参数，但 System.gc 默认不更新这些统计数据，避免用户强行 GC 对这些自适应功能的干扰（可以参考 -XX:+UseAdaptiveSizePolicyWithSystemGC 参数，默认是 false）。

4.3 场景三：MetaSpace 区 OOM

4.3.1 现象

JVM 在启动后或者某个时间点开始，**MetaSpace 的已使用大小在持续增长，同时每次 GC 也无法释放，调大 MetaSpace 空间也无法彻底解决**。

4.3.2 原因

在讨论为什么会 OOM 之前，我们先来看一下这个区里面会存什么数据，Java7 之前字符串常量池被放到了 Perm 区，所有被 intern 的 String 都会被存在这里，由于 String.intern 是不受控的，所以 `-XX:MaxPermSize` 的值也不太好设置，经常会出现 `java.lang.OutOfMemoryError: PermGen space` 异常，所以在 Java7 之后常量池等字面量（Literal）、类静态变量（Class Static）、符号引用（Symbols Reference）等几项被移到 Heap 中。而 Java8 之后 PermGen 也被移除，取而代之的是 MetaSpace。

在最底层，JVM 通过 mmap 接口向操作系统申请内存映射，每次申请 2MB 空间，这里是虚拟内存映射，不是真的就消耗了主存的 2MB，只有之后在使用的时候才会真的消耗内存。申请的这些内存放到一个链表中 VirtualSpaceList，作为其中的一个 Node。

在上层，MetaSpace 主要由 Klass Metaspace 和 NoKlass Metaspace 两大部分组成。

- **Klass MetaSpace：**就是用来存 Klass 的，就是 Class 文件在 JVM 里的运行时数据结构，这部分默认放在 Compressed Class Pointer Space 中，是一块连续的内存区域，紧接着 Heap。Compressed Class Pointer Space 不是必须有的，如果设置了 `-XX:-UseCompressedClassPointers`，或者 `-Xmx` 设置大于 32 G，就不会有这块内存，这种情况下 Klass 都会存在 NoKlass Metaspace 里。

- **NoKlass MetaSpace：**专门来存 Klass 相关的其他的内容，比如 Method，ConstantPool 等，可以由多块不连续的内存组成。虽然叫做 NoKlass Metaspace，但是也其实可以存 Klass 的内容，上面已经提到了对应场景。

具体的定义都可以在源码 shared/vm/memory/metaspace.hpp 中找到：

```
class Metaspace : public AllStatic {

  friend class MetaspaceShared;

 public:
  enum MetadataType {
    ClassType,
    NonClassType,
    MetadataTypeCount
  };
  enum MetaspaceType {
    ZeroMetaspaceType = 0,
    StandardMetaspaceType = ZeroMetaspaceType,
    BootMetaspaceType = StandardMetaspaceType + 1,
    AnonymousMetaspaceType = BootMetaspaceType + 1,
    ReflectionMetaspaceType = AnonymousMetaspaceType + 1,
    MetaspaceTypeCount
  };

 private:

  // Align up the word size to the allocation word size
  static size_t align_word_size_up(size_t);

  // Aligned size of the metaspace.
  static size_t _compressed_class_space_size;

  static size_t compressed_class_space_size() {
    return _compressed_class_space_size;
  }

  static void set_compressed_class_space_size(size_t size) {
    _compressed_class_space_size = size;
  }

  static size_t _first_chunk_word_size;
  static size_t _first_class_chunk_word_size;

  static size_t _commit_alignment;
  static size_t _reserve_alignment;
```

```
  DEBUG_ONLY(static bool   _frozen;)

  // Virtual Space lists for both classes and other metadata
  static metaspace::VirtualSpaceList* _space_list;
  static metaspace::VirtualSpaceList* _class_space_list;

  static metaspace::ChunkManager* _chunk_manager_metadata;
  static metaspace::ChunkManager* _chunk_manager_class;

  static const MetaspaceTracer* _tracer;
}
```

MetaSpace 的对象为什么无法释放，我们看下面两点：

- **MetaSpace 内存管理：**类和其元数据的生命周期与其对应的类加载器相同，只要类的类加载器是存活的，在 Metaspace 中的类元数据也是存活的，不能被回收。每个加载器有单独的存储空间，通过 ClassLoaderMetaspace 来进行管理 SpaceManager* 的指针，相互隔离的。
- **MetaSpace 弹性伸缩：**由于 MetaSpace 空间和 Heap 并不在一起，所以这块的空间可以不用设置或者单独设置，一般情况下避免 MetaSpace 耗尽 VM 内存都会设置一个 MaxMetaSpaceSize，在运行过程中，如果实际大小小于这个值，JVM 就会通过 `-XX:MinMetaspaceFreeRatio` 和 `-XX:-MaxMetaspaceFreeRatio` 两个参数动态控制整个 MetaSpace 的大小，具体使用可以看 `MetaSpaceGC::compute_new_size()` 方法（下方代码），这个方法会在 CMSCollector 和 G1CollectorHeap 等几个收集器执行 GC 时调用。这个里面会根据 `used_after_gc`，`MinMetaspaceFreeRatio` 和 `MaxMetaspaceFreeRatio` 这三个值计算出来一个新的 `_capacity_until_GC` 值（水位线）。然后根据实际的 `_capacity_until_GC` 值使用 `MetaspaceGC::inc_capacity_until_GC()` 和 `MetaspaceGC::dec_capacity_until_GC()` 进行 expand 或 shrink，这个过程也可以参照场景一中的伸缩模型进行理解。

```
void MetaspaceGC::compute_new_size() {
  assert(_shrink_factor <= 100, "invalid shrink factor");
  uint current_shrink_factor = _shrink_factor;
```

```
  _shrink_factor = 0;
  const size_t used_after_gc = MetaspaceUtils::committed_bytes();
  const size_t capacity_until_GC = MetaspaceGC::capacity_until_GC();

  const double minimum_free_percentage = MinMetaspaceFreeRatio / 100.0;
  const double maximum_used_percentage = 1.0 - minimum_free_percentage;

  const double min_tmp = used_after_gc / maximum_used_percentage;
  size_t minimum_desired_capacity =
    (size_t)MIN2(min_tmp, double(max_uintx));
  // Don't shrink less than the initial generation size
  minimum_desired_capacity = MAX2(minimum_desired_capacity,
                                  MetaspaceSize);

  log_trace(gc, metaspace)("MetaspaceGC::compute_new_size: ");
  log_trace(gc, metaspace)("    minimum_free_percentage: %6.2f
maximum_used_percentage: %6.2f",
                           minimum_free_percentage, maximum_used_
percentage);
  log_trace(gc, metaspace)("     used_after_gc       : %6.1fKB", used_
after_gc / (double) K);


  size_t shrink_bytes = 0;
  if (capacity_until_GC < minimum_desired_capacity) {
    // If we have less capacity below the metaspace HWM, then
    // increment the HWM.
    size_t expand_bytes = minimum_desired_capacity - capacity_until_GC;
    expand_bytes = align_up(expand_bytes, Metaspace::commit_alignment());
    // Don't expand unless it's significant
    if (expand_bytes >= MinMetaspaceExpansion) {
      size_t new_capacity_until_GC = 0;
      bool succeeded = MetaspaceGC::inc_capacity_until_GC(expand_bytes,
&new_capacity_until_GC);
      assert(succeeded, "Should always succesfully increment HWM when
at safepoint");

      Metaspace::tracer()->report_gc_threshold(capacity_until_GC,
                                               new_capacity_until_GC,

MetaspaceGCThresholdUpdater::ComputeNewSize);
      log_trace(gc, metaspace)("    expanding:  minimum_desired_
capacity: %6.1fKB  expand_bytes: %6.1fKB  MinMetaspaceExpansion:
%6.1fKB  new metaspace HWM:  %6.1fKB",
                               minimum_desired_capacity / (double) K,
                               expand_bytes / (double) K,
                               MinMetaspaceExpansion / (double) K,
                               new_capacity_until_GC / (double) K);
    }
```

```
    return;
  }

  // No expansion, now see if we want to shrink
  // We would never want to shrink more than this
  assert(capacity_until_GC >= minimum_desired_capacity,
         SIZE_FORMAT " >= " SIZE_FORMAT,
         capacity_until_GC, minimum_desired_capacity);
  size_t max_shrink_bytes = capacity_until_GC - minimum_desired_capacity;

  // Should shrinking be considered?
  if (MaxMetaspaceFreeRatio < 100) {
    const double maximum_free_percentage = MaxMetaspaceFreeRatio /
100.0;
    const double minimum_used_percentage = 1.0 - maximum_free_
percentage;
    const double max_tmp = used_after_gc / minimum_used_percentage;
    size_t maximum_desired_capacity = (size_t)MIN2(max_tmp, double(max_
uintx));
    maximum_desired_capacity = MAX2(maximum_desired_capacity,
                                    MetaspaceSize);
    log_trace(gc, metaspace)("    maximum_free_percentage: %6.2f
minimum_used_percentage: %6.2f",
                             maximum_free_percentage, minimum_used_
percentage);
    log_trace(gc, metaspace)("    minimum_desired_capacity: %6.1fKB
maximum_desired_capacity: %6.1fKB",
                             minimum_desired_capacity / (double) K,
maximum_desired_capacity / (double) K);

    assert(minimum_desired_capacity <= maximum_desired_capacity,
           "sanity check");

    if (capacity_until_GC > maximum_desired_capacity) {
      // Capacity too large, compute shrinking size
      shrink_bytes = capacity_until_GC - maximum_desired_capacity;
      shrink_bytes = shrink_bytes / 100 * current_shrink_factor;

      shrink_bytes = align_down(shrink_bytes, Metaspace::commit_
alignment());

      assert(shrink_bytes <= max_shrink_bytes,
             "invalid shrink size " SIZE_FORMAT " not <= " SIZE_FORMAT,
             shrink_bytes, max_shrink_bytes);
      if (current_shrink_factor == 0) {
        _shrink_factor = 10;
      } else {
        _shrink_factor = MIN2(current_shrink_factor * 4, (uint) 100);
      }
```

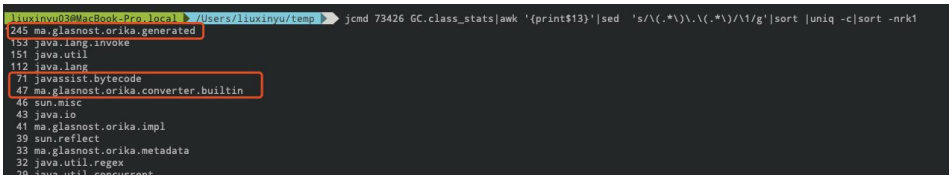
```
      log_trace(gc, metaspace)("    shrinking:  initThreshold: %.1fK
maximum_desired_capacity: %.1fK",
                               MetaspaceSize / (double) K, maximum_
desired_capacity / (double) K);
      log_trace(gc, metaspace)("    shrink_bytes: %.1fK  current_
shrink_factor: %d  new shrink factor: %d  MinMetaspaceExpansion:
%.1fK",
                               shrink_bytes / (double) K, current_shrink_
factor, _shrink_factor, MinMetaspaceExpansion / (double) K);
    }
  }

  // Don't shrink unless it's significant
  if (shrink_bytes >= MinMetaspaceExpansion &&
      ((capacity_until_GC - shrink_bytes) >= MetaspaceSize)) {
    size_t new_capacity_until_GC = MetaspaceGC::dec_capacity_until_
GC(shrink_bytes);
    Metaspace::tracer()->report_gc_threshold(capacity_until_GC,
                                             new_capacity_until_GC,

MetaspaceGCThresholdUpdater::ComputeNewSize);
  }
}
```

由场景一可知，为了避免弹性伸缩带来的额外 GC 消耗，我们会将 `-XX:MetaSpaceSize` 和 `-XX:MaxMetaSpaceSize` 两个值设置为固定的，但是这样也会导致在空间不够的时候无法扩容，然后频繁地触发 GC，最终 OOM。所以关键原因就是 ClassLoader 不停地在内存中 load 了新的 Class ，一般这种问题都发生在动态类加载等情况上。

4.3.3 策略

了解大概什么原因后，如何定位和解决就很简单了，可以 dump 快照之后通过 JProfiler 或 MAT 观察 Classes 的 Histogram（直方图）即可，或者直接通过命令即可定位，jcmd 打几次 Histogram 的图，看一下具体是哪个包下的 Class 增加较多就可以定位了。不过有时候也要结合 InstBytes、KlassBytes、Bytecodes、MethodAll 等几项指标综合来看下。如下图便是笔者使用 jcmd 排查到一个 Orika 的问题。

```
jcmd <PID> GC.class_stats|awk '{print$13}'|sed  's/\(.*\)\.\(.*\)/\1/
g'|sort |uniq -c|sort -nrk1
```



如果无法从整体的角度定位，可以添加 `-XX:+TraceClassLoading` 和 `-XX:+TraceClassUnLoading` 参数观察详细的类加载和卸载信息。

4.3.4 小结

原理理解比较复杂，但定位和解决问题会比较简单，经常会出问题的几个点有 Orika 的 classMap、JSON 的 ASMSerializer、Groovy 动态加载类等，基本都集中在反射、Javasisit 字节码增强、CGLIB 动态代理、OSGi 自定义类加载器等的技术点上。另外就是及时给 MetaSpace 区的使用率加一个监控，如果指标有波动提前发现并解决问题。

4.4 场景四：过早晋升 *

4.4.1 现象

这种场景主要发生在分代的收集器上面，专业的术语称为“Premature Promotion”。90% 的对象朝生夕死，只有在 Young 区经历过几次 GC 的洗礼后才会晋升到 Old 区，每经历一次 GC 对象的 GC Age 就会增长 1，最大通过 `-XX:MaxTenuringThreshold` 来控制。

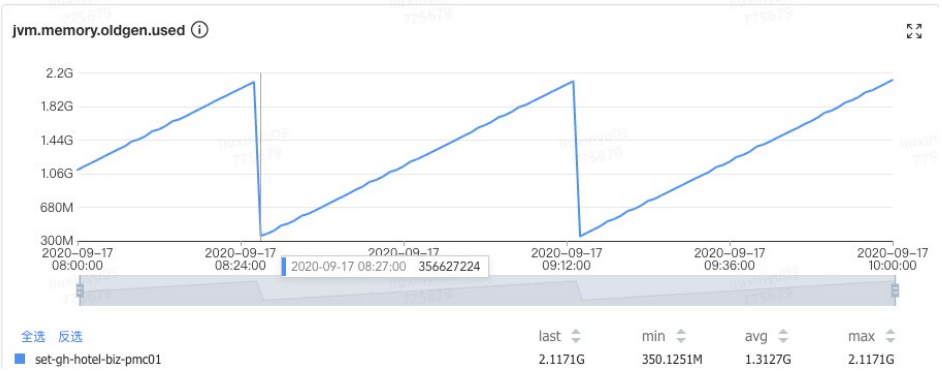
过早晋升一般不会直接影响 GC，总会伴随着浮动垃圾、大对象担保失败等问题，但这些问题不是立刻发生的，我们可以观察以下几种现象来判断是否发生了过早晋升。

分配速率接近于晋升速率，对象晋升年龄较小。

GC 日志中出现“Desired survivor size 107347968 bytes, **new threshold 1(max 6)**”等信息，说明此时经历过一次 GC 就会放到 Old 区。

Full GC 比较频繁，且经历过一次 GC 之后 Old 区的**变化比例非常大**。

比如说 Old 区触发的回收阈值是 80%，经历过一次 GC 之后下降到了 10%，这就说明 Old 区的 70% 的对象存活时间其实很短，如下图所示，Old 区大小每次 GC 后从 2.1G 回收到 300M，也就是说回收掉了 1.8G 的垃圾，只有 **300M 的活跃对象**。整个 Heap 目前是 4G，活跃对象只占了不到十分之一。



过早晋升的危害：

- Young GC 频繁，总的吞吐量下降。
- Full GC 频繁，可能会有较大停顿。

4.4.2 原因

主要的原因有以下两点：

- **Young/Eden 区过小：**过小的直接后果就是 Eden 被装满的时间变短，本应该回收的对象参与了 GC 并晋升，Young GC 采用的是复制算法，由基础篇我们知道 copying 耗时远大于 mark，也就是 Young GC 耗时本质上就是 copy 的时间（CMS 扫描 Card Table 或 G1 扫描 Remember Set 出问题的情况另说），没来及回收的对象增大了回收的代价，所以 Young GC 时间增加，同时又无法快速释放空间，Young GC 次数也跟着增加。
- **分配速率过大：**可以观察出问题前后 Mutator 的分配速率，如果有明显波动可以尝试观察网卡流量、存储类中间件慢查询日志等信息，看是否有大量数据被加载到内存中。

同时无法 GC 掉对象还会带来另外一个问题，引发动态年龄计算：JVM 通过 `-XX:-MaxTenuringThreshold` 参数来控制晋升年龄，每经过一次 GC，年龄就会加一，达到最大年龄就可以进入 Old 区，最大值为 15（因为 JVM 中使用 4 个比特来表示对象的年龄）。设定固定的 MaxTenuringThreshold 值作为晋升条件：

- MaxTenuringThreshold 如果设置得过大，原本应该晋升的对象一直停留在 Survivor 区，直到 Survivor 区溢出，一旦溢出发生，Eden + Survivor 中对象将不再依据年龄全部提升到 Old 区，这样对象老化的机制就失效了。
- MaxTenuringThreshold 如果设置得过小，过早晋升即对象不能在 Young 区充分被回收，大量短期对象被晋升到 Old 区，Old 区空间迅速增长，引起频繁的 Major GC，分代回收失去了意义，严重影响 GC 性能。

相同应用在不同时间的表现不同，特殊任务的执行或者流量成分的变化，都会导致对象的生命周期分布发生波动，那么固定的阈值设定，因为无法动态适应变化，会造成和上面问题，所以 Hotspot 会使用动态计算的方式来调整晋升的阈值。

具体动态计算可以看一下 Hotspot 源码，具体在 /src/hotspot/share/gc/shared/ageTable.cpp 的 `compute_tenuring_threshold` 方法中：

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
  //TargetSurvivorRatio 默认 50，意思是：在回收之后希望 survivor 区的占用率达到这个比例
  size_t desired_survivor_size = (size_t)((((double) survivor_capacity)*TargetSurvivorRatio)/100);
  size_t total = 0;
  uint age = 1;
  assert(sizes[0] == 0, "no objects with age zero should be recorded");
  while (age < table_size) {//table_size=16
    total += sizes[age];
    // 如果加上这个年龄的所有对象的大小之后，占用量 > 期望的大小，就设置 age 为新的晋升阈值
    if (total > desired_survivor_size) break;
    age++;
  }

  uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
  if (PrintTenuringDistribution || UsePerfData) {
```

```
    // 打印期望的 survivor 的大小以及新计算出来的阈值，和设置的最大阈值
    if (PrintTenuringDistribution) {
      gclog_or_tty->cr();
      gclog_or_tty->print_cr("Desired survivor size " SIZE_FORMAT "
bytes, new threshold %u (max %u)",
        desired_survivor_size*oopSize, result, (int)
MaxTenuringThreshold);
    }

    total = 0;
    age = 1;
    while (age < table_size) {
      total += sizes[age];
      if (sizes[age] > 0) {
        if (PrintTenuringDistribution) {
          gclog_or_tty->print_cr("- age %3u: " SIZE_FORMAT_W(10) "
bytes, " SIZE_FORMAT_W(10) " total",
                                              age,    sizes[age]*oopSize,
total*oopSize);
        }
      }
      if (UsePerfData) {
        _perf_sizes[age]->set_value(sizes[age]*oopSize);
      }
      age++;
    }
    if (UsePerfData) {
      SharedHeap* sh = SharedHeap::heap();
      CollectorPolicy* policy = sh->collector_policy();
      GCPolicyCounters* gc_counters = policy->counters();
      gc_counters->tenuring_threshold()->set_value(result);
      gc_counters->desired_survivor_size()->set_value(
        desired_survivor_size*oopSize);
    }
  }

  return result;
}
```

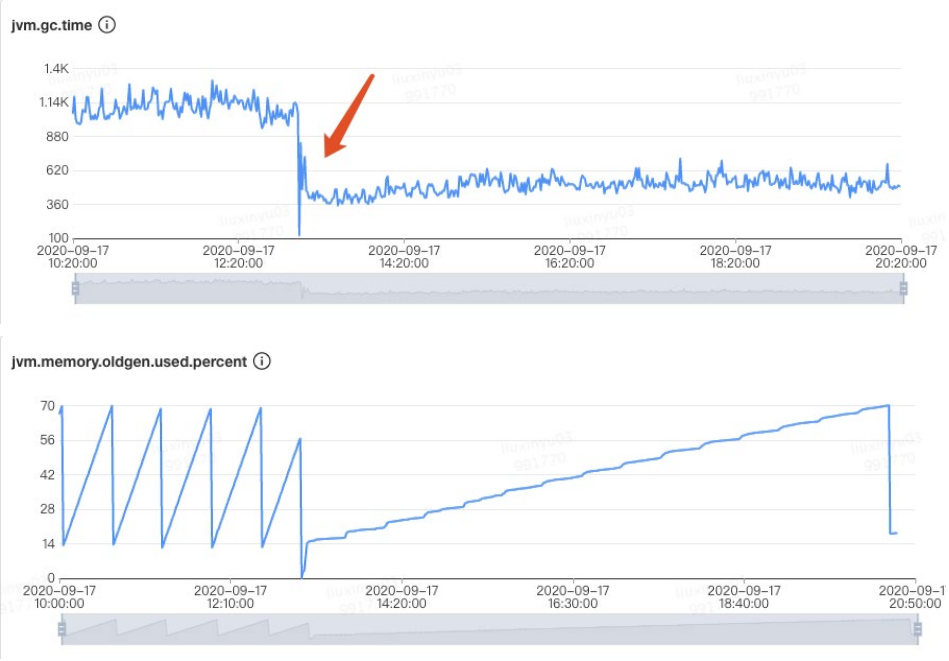
可以看到 Hotspot 遍历所有对象时，从所有年龄为 0 的对象占用的空间开始累加，如果加上年龄等于 n 的所有对象的空间之后，使用 Survivor 区的条件值（Target-SurvivorRatio / 100，TargetSurvivorRatio 默认值为 50）进行判断，若大于这个值则结束循环，将 n 和 MaxTenuringThreshold 比较，若 n 小，则阈值为 n，若 n 大，则只能去设置最大阈值为 MaxTenuringThreshold。**动态年龄触发后导致更多**

的对象进入了 Old 区，造成资源浪费。

4.4.3 策略

知道问题原因后我们就有解决的方向，如果是 **Young/Eden 区过小**，我们可以在总的 Heap 内存不变的情况下适当增大 Young 区，具体怎么增加？一般情况下 Old 的大小应当为活跃对象的 2~3 倍左右，考虑到浮动垃圾问题最好在 3 倍左右，剩下的都可以分给 Young 区。

拿笔者的一次典型过早晋升优化来看，原配置为 Young 1.2G + Old 2.8G，通过观察 CMS GC 的情况找到存活对象大概为 300~400M，于是调整 Old 1.5G 左右，剩下 2.5G 分给 Young 区。仅仅调了一个 Young 区大小参数（`-Xmn`），整个 JVM 一分钟 Young GC 从 26 次降低到了 11 次，单次时间也没有增加，总的 GC 时间从 1100ms 降低到了 500ms，CMS GC 次数也从 40 分钟左右一次降低到了 7 小时 30 分钟一次。



如果是分配速率过大：

- **偶发较大**：通过内存分析工具找到问题代码，从业务逻辑上做一些优化。
- **一直较大**：当前的 Collector 已经不满足 Mutator 的期望了，这种情况要么扩容 Mutator 的 VM，要么调整 GC 收集器类型或加大空间。

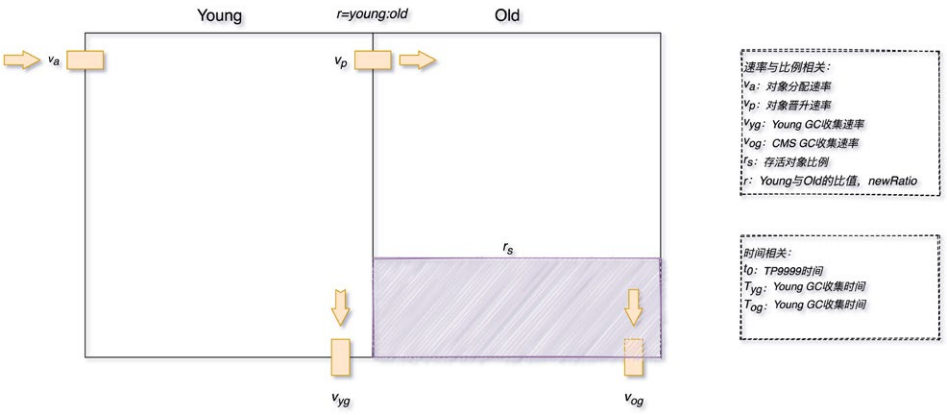
4.4.4 小结

过早晋升问题一般不会特别明显，但日积月累之后可能会爆发一波收集器退化之类的问题，所以我们还是要提前避免掉的，可以看看自己系统里面是否有这些现象，如果比较匹配的话，可以尝试优化一下。一行代码优化的 ROI 还是很高的。

如果在观察 Old 区前后比例变化的过程中，发现可以回收的比例非常小，如从 80% 只回收到了 60%，说明我们大部分对象都是存活的，Old 区的空间可以适当调大些。

4.4.5 加餐

关于在调整 Young 与 Old 的比例时，如何选取具体的 NewRatio 值，这里将问题抽象成为一个蓄水池模型，找到以下关键衡量指标，大家可以根据自己场景进行推算。



$$(1) r = f(v_a, v_p, v_{yc}, v_{oc}, r_s)$$

$$(2) T_{stw} = T_{yc} + T_{oc} = \int_0^t g(v_{yc}, v_p)dt + \int_0^t g(v_{oc})dt, t \in (0, +\infty)$$

$$(3) f(t) = \lim_{\bar{T_{yg}} \to 0, T_{oc} \to 0} \frac{\Sigma_{k=1}^{n}(t_{ak} - t_{bk} + \bar{T_{yg}}) + T_{oc}}{n} \Rightarrow f(t) = \frac{T_n}{n} < t_0$$

($\bar{T_{yg}}$为平均一次$YoungGC$时间，t_{ak}为第k次$YoungGC$结束时间，t_{bk}为第k次$YoungGC$开始时间，T_n为总运行时间)

- NewRatio 的值 r 与 va、vp、vyc、voc、rs 等值存在一定函数相关性（rs 越小 r 越大、r 越小 vp 越小，…，之前尝试使用 NN 来辅助建模，但目前还没有完全算出具体的公式，有想法的同学可以在评论区给出你的答案）。
- 总停顿时间 T 为 Young GC 总时间 Tyc 和 Old GC 总时间 Toc 之和，其中 Tyc 与 vyc 和 vp 相关，Toc 与 voc 相关。
- 忽略掉 GC 时间后，两次 Young GC 的时间间隔要大于 TP9999 时间，这样尽量让对象在 Eden 区就被回收，可以减少很多停顿。

4.5 场景五：CMS Old GC 频繁 *

4.5.1 现象

Old 区频繁的做 CMS GC，但是每次耗时不是特别长，整体最大 STW 也在可接受范围内，但由于 GC 太频繁导致吞吐下降比较多。

4.5.2 原因

这种情况比较常见，基本都是一次 Young GC 完成后，负责处理 CMS GC 的一个后台线程 concurrentMarkSweepThread 会不断地轮询，使用 `shouldConcurrentCollect()` 方法做一次检测，判断是否达到了回收条件。如果达到条件，使用 `collect_in_background()` 启动一次 Background 模式 GC。轮询的判断是使用 `sleepBeforeNextCycle()` 方法，间隔周期为 `-XX:CMSWaitDuration` 决定，默认为 2s。

具体代码在：src/hotspot/share/gc/cms/concurrentMarkSweepThread.cpp。

```
void ConcurrentMarkSweepThread::run_service() {
  assert(this == cmst(), "just checking");

  if (BindCMSThreadToCPU && !os::bind_to_processor(CPUForCMSThread)) {
```

```
    log_warning(gc)("Couldn't bind CMS thread to processor " UINTX_
FORMAT, CPUForCMSThread);
  }

  while (!should_terminate()) {
    sleepBeforeNextCycle();
    if (should_terminate()) break;
    GCIdMark gc_id_mark;
    GCCause::Cause cause = _collector->_full_gc_requested ?
      _collector->_full_gc_cause : GCCause::_cms_concurrent_mark;
    _collector->collect_in_background(cause);
  }
  verify_ok_to_terminate();
}
```

```
void ConcurrentMarkSweepThread::sleepBeforeNextCycle() {
  while (!should_terminate()) {
    if(CMSWaitDuration >= 0) {
      // Wait until the next synchronous GC, a concurrent full gc
      // request or a timeout, whichever is earlier.
      wait_on_cms_lock_for_scavenge(CMSWaitDuration);
    } else {
      // Wait until any cms_lock event or check interval not to call
shouldConcurrentCollect permanently
      wait_on_cms_lock(CMSCheckInterval);
    }
    // Check if we should start a CMS collection cycle
    if (_collector->shouldConcurrentCollect()) {
      return;
    }
    // .. collection criterion not yet met, let's go back
    // and wait some more
  }
}
```

判断是否进行回收的代码在：/src/hotspot/share/gc/cms/concurrentMarkSweep-Generation.cpp。

```
bool CMSCollector::shouldConcurrentCollect() {
  LogTarget(Trace, gc) log;

  if (_full_gc_requested) {
    log.print("CMSCollector: collect because of explicit  gc request
(or GCLocker)");
    return true;
  }
```

```
  FreelistLocker x(this);
  // ------------------------------------------------------------------
--
  // Print out lots of information which affects the initiation of
  // a collection.
  if (log.is_enabled() && stats().valid()) {
    log.print("CMSCollector shouldConcurrentCollect: ");

    LogStream out(log);
    stats().print_on(&out);

    log.print("time_until_cms_gen_full %3.7f", stats().time_until_cms_
gen_full());
    log.print("free=" SIZE_FORMAT, _cmsGen->free());
    log.print("contiguous_available=" SIZE_FORMAT, _cmsGen->contiguous_
available());
    log.print("promotion_rate=%g", stats().promotion_rate());
    log.print("cms_allocation_rate=%g", stats().cms_allocation_rate());
    log.print("occupancy=%3.7f", _cmsGen->occupancy());
    log.print("initiatingOccupancy=%3.7f", _cmsGen->initiating_
occupancy());
    log.print("cms_time_since_begin=%3.7f", stats().cms_time_since_
begin());
    log.print("cms_time_since_end=%3.7f", stats().cms_time_since_
end());
    log.print("metadata initialized %d", MetaspaceGC::should_concurrent_
collect());
  }
  // ------------------------------------------------------------------
--
  if (!UseCMSInitiatingOccupancyOnly) {
    if (stats().valid()) {
      if (stats().time_until_cms_start() == 0.0) {
        return true;
      }
    } else {

      if (_cmsGen->occupancy() >= _bootstrap_occupancy) {
        log.print(" CMSCollector: collect for bootstrapping
statistics: occupancy = %f, boot occupancy = %f",
                  _cmsGen->occupancy(), _bootstrap_occupancy);
        return true;
      }
    }
  }

  if (_cmsGen->should_concurrent_collect()) {
    log.print("CMS old gen initiated");
    return true;
```

```
  }

  // We start a collection if we believe an incremental collection
may fail;
  // this is not likely to be productive in practice because it's
probably too
  // late anyway.
  CMSHeap* heap = CMSHeap::heap();
  if (heap->incremental_collection_will_fail(true /* consult_young */))
{
    log.print("CMSCollector: collect because incremental collection
will fail ");
    return true;
  }

  if (MetaspaceGC::should_concurrent_collect()) {
    log.print("CMSCollector: collect for metadata allocation ");
    return true;
  }

  // CMSTriggerInterval starts a CMS cycle if enough time has passed.
  if (CMSTriggerInterval >= 0) {
    if (CMSTriggerInterval == 0) {
      // Trigger always
      return true;
    }

    // Check the CMS time since begin (we do not check the stats
validity
    // as we want to be able to trigger the first CMS cycle as well)
    if (stats().cms_time_since_begin() >= (CMSTriggerInterval / ((double)
MILLIUNITS))) {
      if (stats().valid()) {
        log.print("CMSCollector: collect because of trigger interval
(time since last begin %3.7f secs)",
                  stats().cms_time_since_begin());
      } else {
        log.print("CMSCollector: collect because of trigger interval
(first collection)");
      }
      return true;
    }
  }

  return false;
}
```

分析其中逻辑判断是否触发 GC，分为以下几种情况：

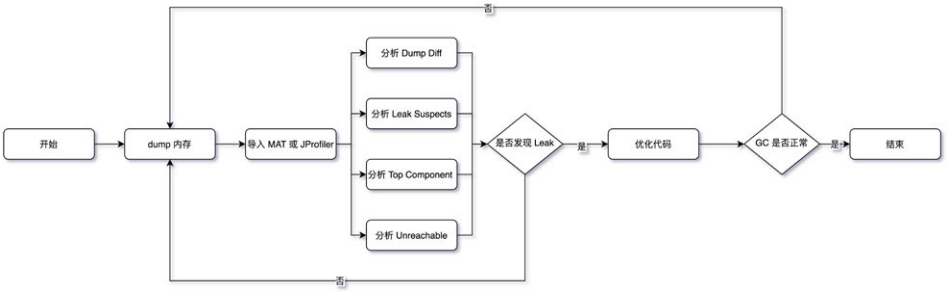
- **触发 CMS GC：**通过调用 `_collector->collect_in_background()` 进行触发 Background GC 。
  - CMS 默认采用 JVM 运行时的统计数据判断是否需要触发 CMS GC，如果需要根据 `-XX:CMSInitiatingOccupancyFraction` 的值进行判断，需要设置参数 `-XX:+UseCMSInitiatingOccupancyOnly`。
  - 如果开启了 `-XX:UseCMSInitiatingOccupancyOnly` 参数，判断当前 Old 区使用率是否大于阈值，则触发 CMS GC，该阈值可以通过参数 `-XX:CMSInitiatingOccupancyFraction` 进行设置，如果没有设置，默认为 92%。
  - 如果之前的 Young GC 失败过，或者下次 Young 区执行 Young GC 可能失败，这两种情况下都需要触发 CMS GC。
  - CMS 默认不会对 MetaSpace 或 Perm 进行垃圾收集，如果希望对这些区域进行垃圾收集，需要设置参数 `-XX:+CMSClassUnloadingEnabled`。
- **触发 Full GC：**直接进行 Full GC，这种情况到场景七中展开说明。
  - 如果 `_full_gc_requested` 为真，说明有明确的需求要进行 GC，比如调用 System.gc。
  - 在 Eden 区为对象或 TLAB 分配内存失败，导致一次 Young GC，在 `GenCollectorPolicy` 类的 `satisfy_failed_allocation()` 方法中进行判断。

大家可以看一下源码中的日志打印，通过日志我们就可以比较清楚地知道具体的原因，然后就可以着手分析了。

4.5.3 策略

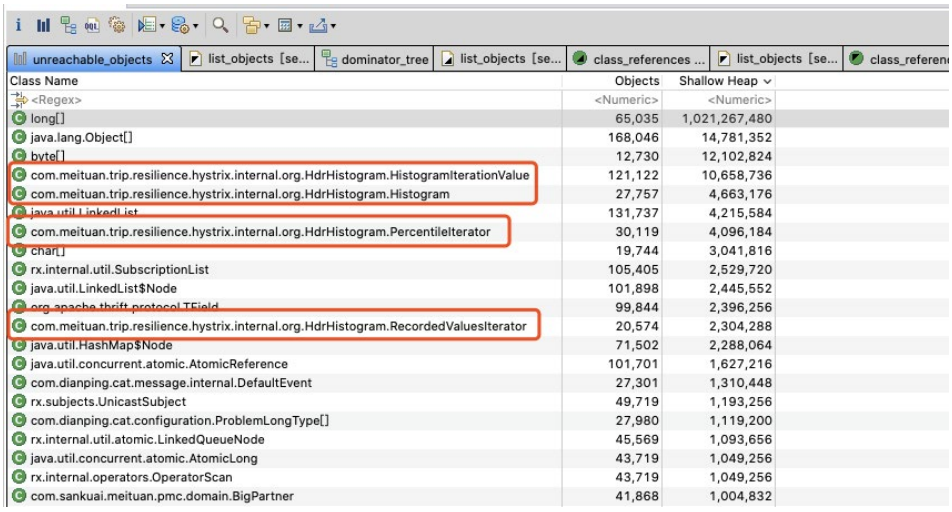
我们这里还是拿最常见的达到回收比例这个场景来说，与过早晋升不同的是这些对象确实存活了一段时间，Survival Time 超过了 TP9999 时间，但是又达不到长期存活，如各种数据库、网络链接，带有失效时间的缓存等。

处理这种常规内存泄漏问题基本是一个思路，主要步骤如下：



Dump Diff 和 Leak Suspects 比较直观就不介绍了，这里说下其它几个关键点：

- **内存 Dump：**使用 jmap、arthas 等 dump 堆进行快照时记得摘掉流量，同时**分别在 CMS GC 的发生前后分别 dump 一次**。
- **分析 Top Component：**要记得按照对象、类、类加载器、包等多个维度观察 Histogram，同时使用 outgoing 和 incoming 分析关联的对象，另外就是 Soft Reference 和 Weak Reference、Finalizer 等也要看一下。
- **分析 Unreachable：**重点看一下这个，关注下 Shallow 和 Retained 的大小。如下图所示，笔者之前一次 GC 优化，就根据 Unreachable Objects 发现了 Hystrix 的滑动窗口问题。



unreachable_objects | list_objects [se... | dominator_tree | list_objects [se... | class_references ... | list_objects [se... | class_referen

Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
long[]	65,035	1,021,267,480
java.lang.Object[]	168,046	14,781,352
byte[]	12,730	12,102,824
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.HistogramIterationValue	121,122	10,658,736
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.Histogram	27,757	4,663,176
java.util.LinkedList	131,737	4,215,584
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.PercentileIterator	30,119	4,096,184
char[]	19,744	3,041,816
rx.internal.util.SubscriptionList	105,405	2,529,720
java.util.LinkedList$Node	101,898	2,445,552
org.apache.thrift.protocol.TField	99,844	2,396,256
com.meituan.trip.resilience.hystrix.internal.org.HdrHistogram.RecordedValuesIterator	20,574	2,304,288
java.util.HashMap$Node	71,502	2,288,064
java.util.concurrent.atomic.AtomicReference	101,701	1,627,216
com.dianping.cat.message.internal.DefaultEvent	27,301	1,310,448
rx.subjects.UnicastSubject	49,719	1,193,256
com.dianping.cat.configuration.ProblemLongType[]	27,980	1,119,200
rx.internal.util.atomic.LinkedQueueNode	45,569	1,093,656
java.util.concurrent.atomic.AtomicLong	43,719	1,049,256
rx.internal.operators.OperatorScan	43,719	1,049,256
com.sankuai.meituan.pmc.domain.BigPartner	41,868	1,004,832

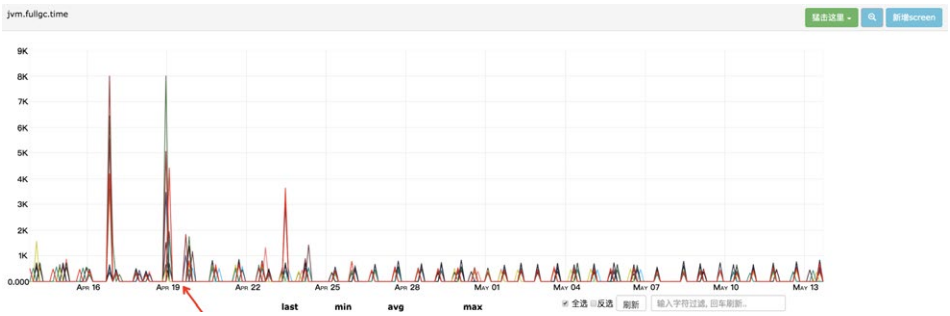
4.5.4 小结

经过整个流程下来基本就能定位问题了，不过在优化的过程中记得使用**控制变量**的方法来优化，防止一些会加剧问题的改动被掩盖。

4.6 场景六：单次 CMS Old GC 耗时长 *

4.6.1 现象

CMS GC 单次 STW 最大超过 1000ms，不会频繁发生，如下图所示最长达到了 8000ms。某些场景下会引起“雪崩效应”，这种场景非常危险，我们应该尽量避免出现。



4.6.2 原因

CMS 在回收的过程中，STW 的阶段主要是 Init Mark 和 Final Remark 这两个阶段，也是导致 CMS Old GC 最多的原因，另外有些情况就是在 STW 前等待 Mutator 的线程到达 SafePoint 也会导致时间过长，但这种情况较少，我们在此处主要讨论前者。发生收集器退化或者碎片压缩的场景请看场景七。

想要知道这两个阶段为什么会耗时，我们需要先看一下这两个阶段都会干什么。

核心代码都在 /src/hotspot/share/gc/cms/concurrentMarkSweepGeneration.cpp 中，内部有个线程 ConcurrentMarkSweepThread 轮询来校验，Old 区的垃圾回收相关细节被完全封装在 `CMSCollector` 中，调用入口就是 ConcurrentMarkSweepThread 调用的 `CMSCollector::collect_in_background` 和

`ConcurrentMarkSweepGeneration` 调用的 `CMSCollector::collect` 方法，此处我们讨论大多数场景的 `collect_in_background`。整个过程中会 STW 的主要是 initial Mark 和 Final Remark，核心代码在 `VM_CMS_Initial_Mark` / `VM_CMS_Final_Remark` 中，执行时需要将执行权交由 VMThread 来执行。

- CMS Init Mark 执行步骤，实现在 `CMSCollector::checkpointRootsInitialWork()` 和 `CMSParInitialMarkTask::work` 中，整体步骤和代码如下：

```
void CMSCollector::checkpointRootsInitialWork() {
  assert(SafepointSynchronize::is_at_safepoint(), "world should be
stopped");
  assert(_collectorState == InitialMarking, "just checking");

  // Already have locks.
  assert_lock_strong(bitMapLock());
  assert(_markBitMap.isAllClear(), "was reset at end of previous
cycle");

  // Setup the verification and class unloading state for this
  // CMS collection cycle.
  setup_cms_unloading_and_verification_state();

  GCTraceTime(Trace, gc, phases) ts("checkpointRootsInitialWork", _gc_
timer_cm);

  // Reset all the PLAB chunk arrays if necessary.
  if (_survivor_plab_array != NULL && !CMSPLABRecordAlways) {
    reset_survivor_plab_arrays();
  }

  ResourceMark rm;
  HandleMark  hm;

  MarkRefsIntoClosure notOlder(_span, &_markBitMap);
  CMSHeap* heap = CMSHeap::heap();

  verify_work_stacks_empty();
  verify_overflow_empty();

  heap->ensure_parsability(false);  // fill TLABs, but no need to
retire them
  // Update the saved marks which may affect the root scans.
  heap->save_marks();
```

```
  // weak reference processing has not started yet.
  ref_processor()->set_enqueuing_is_done(false);

  // Need to remember all newly created CLDs,
  // so that we can guarantee that the remark finds them.
  ClassLoaderDataGraph::remember_new_clds(true);

  // Whenever a CLD is found, it will be claimed before proceeding to
mark
  // the klasses. The claimed marks need to be cleared before marking
starts.
  ClassLoaderDataGraph::clear_claimed_marks();

  print_eden_and_survivor_chunk_arrays();

  {
    if (CMSParallelInitialMarkEnabled) {
      // The parallel version.
      WorkGang* workers = heap->workers();
      assert(workers != NULL, "Need parallel worker threads.");
      uint n_workers = workers->active_workers();

      StrongRootsScope srs(n_workers);

      CMSParInitialMarkTask tsk(this, &srs, n_workers);
      initialize_sequential_subtasks_for_young_gen_rescan(n_workers);
      // If the total workers is greater than 1, then multiple workers
      // may be used at some time and the initialization has been set
      // such that the single threaded path cannot be used.
      if (workers->total_workers() > 1) {
        workers->run_task(&tsk);
      } else {
        tsk.work(0);
      }
    } else {
      // The serial version.
      CLDToOopClosure cld_closure(&notOlder, true);
      heap->rem_set()->prepare_for_younger_refs_iterate(false); // Not
parallel.

      StrongRootsScope srs(1);

      heap->cms_process_roots(&srs,
                              true,   // young gen as roots
                              GenCollectedHeap::ScanningOption(roots_
scanning_options()),
                              should_unload_classes(),
                              &notOlder,
```

```
                                &cld_closure);
    }
  }

  // Clear mod-union table; it will be dirtied in the prologue of
  // CMS generation per each young generation collection.
  assert(_modUnionTable.isAllClear(),
       "Was cleared in most recent final checkpoint phase"
       " or no bits are set in the gc_prologue before the start of the
next "
       "subsequent marking phase.");

  assert(_ct->cld_rem_set()->mod_union_is_clear(), "Must be");
  // Save the end of the used_region of the constituent generations
  // to be used to limit the extent of sweep in each generation.
  save_sweep_limits();
  verify_overflow_empty();
}
```

```
void CMSParInitialMarkTask::work(uint worker_id) {
  elapsedTimer _timer;
  ResourceMark rm;
  HandleMark   hm;

  // ---------- scan from roots --------------
  _timer.start();
  CMSHeap* heap = CMSHeap::heap();
  ParMarkRefsIntoClosure par_mri_cl(_collector->_span, &(_collector->_
markBitMap));

  // ---------- young gen roots --------------
  {
    work_on_young_gen_roots(&par_mri_cl);
    _timer.stop();
    log_trace(gc, task)("Finished young gen initial mark scan work in
%dth thread: %3.3f sec", worker_id, _timer.seconds());
  }

  // ---------- remaining roots --------------
  _timer.reset();
  _timer.start();

  CLDToOopClosure cld_closure(&par_mri_cl, true);

  heap->cms_process_roots(_strong_roots_scope,
                          false,     // yg was scanned above
                          GenCollectedHeap::ScanningOption(_collector-
>CMSCollector::roots_scanning_options()),
                          _collector->should_unload_classes(),
```

```
                            &par_mri_cl,
                            &cld_closure,
                            &_par_state_string);

  assert(_collector->should_unload_classes()
         || (_collector->CMSCollector::roots_scanning_options() &
GenCollectedHeap::SO_AllCodeCache),
         "if we didn't scan the code cache, we have to be ready to
drop nmethods with expired weak oops");
  _timer.stop();
  log_trace(gc, task)("Finished remaining root initial mark scan work
in %dth thread: %3.3f sec", worker_id, _timer.seconds());
}
```

整个过程比较简单，从 GC Root 出发标记 Old 中的对象，处理完成后借助 BitMap 处理下 Young 区对 Old 区的引用，整个过程基本都比较快，很少会有较大的停顿。

- CMS Final Remark 执行步骤，实现在 `CMSCollector::checkpointRootsFinalWork()` 中，整体代码和步骤如下：

```
void CMSCollector::checkpointRootsFinalWork() {
  GCTraceTime(Trace, gc, phases) tm("checkpointRootsFinalWork", _gc_
timer_cm);

  assert(haveFreelistLocks(), "must have free list locks");
  assert_lock_strong(bitMapLock());

  ResourceMark rm;
  HandleMark   hm;

  CMSHeap* heap = CMSHeap::heap();

  if (should_unload_classes()) {
    CodeCache::gc_prologue();
  }
  assert(haveFreelistLocks(), "must have free list locks");
  assert_lock_strong(bitMapLock());

  heap->ensure_parsability(false);  // fill TLAB's, but no need to
retire them
  // Update the saved marks which may affect the root scans.
  heap->save_marks();
```

```
  print_eden_and_survivor_chunk_arrays();

  {
    if (CMSParallelRemarkEnabled) {
      GCTraceTime(Debug, gc, phases) t("Rescan (parallel)", _gc_timer_
cm);
      do_remark_parallel();
    } else {
      GCTraceTime(Debug, gc, phases) t("Rescan (non-parallel)", _gc_
timer_cm);
      do_remark_non_parallel();
    }
  }
  verify_work_stacks_empty();
  verify_overflow_empty();

  {
    GCTraceTime(Trace, gc, phases) ts("refProcessingWork", _gc_timer_
cm);
    refProcessingWork();
  }
  verify_work_stacks_empty();
  verify_overflow_empty();

  if (should_unload_classes()) {
    CodeCache::gc_epilogue();
  }
  JvmtiExport::gc_epilogue();
  assert(_markStack.isEmpty(), "No grey objects");
  size_t ser_ovflw = _ser_pmc_remark_ovflw + _ser_pmc_preclean_ovflw +
                     _ser_kac_ovflw        + _ser_kac_preclean_ovflw;
  if (ser_ovflw > 0) {
    log_trace(gc)("Marking stack overflow (benign) (pmc_pc=" SIZE_
FORMAT ", pmc_rm=" SIZE_FORMAT ", kac=" SIZE_FORMAT ", kac_preclean="
SIZE_FORMAT ")",
                          _ser_pmc_preclean_ovflw, _ser_pmc_remark_ovflw,
_ser_kac_ovflw, _ser_kac_preclean_ovflw);
    _markStack.expand();
    _ser_pmc_remark_ovflw = 0;
    _ser_pmc_preclean_ovflw = 0;
    _ser_kac_preclean_ovflw = 0;
    _ser_kac_ovflw = 0;
  }
  if (_par_pmc_remark_ovflw > 0 || _par_kac_ovflw > 0) {
     log_trace(gc)("Work queue overflow (benign) (pmc_rm=" SIZE_FORMAT
", kac=" SIZE_FORMAT ")",
                           _par_pmc_remark_ovflw, _par_kac_ovflw);
     _par_pmc_remark_ovflw = 0;
    _par_kac_ovflw = 0;
```

```
    }
     if (_markStack._hit_limit > 0) {
       log_trace(gc)(" (benign) Hit max stack size limit (" SIZE_FORMAT
")",
                              _markStack._hit_limit);
     }
     if (_markStack._failed_double > 0) {
       log_trace(gc)(" (benign) Failed stack doubling (" SIZE_FORMAT "),
current capacity " SIZE_FORMAT,
                              _markStack._failed_double, _markStack.
capacity());
     }
    _markStack._hit_limit = 0;
    _markStack._failed_double = 0;

    if ((VerifyAfterGC || VerifyDuringGC) &&
        CMSHeap::heap()->total_collections() >= VerifyGCStartAt) {
      verify_after_remark();
    }

    _gc_tracer_cm->report_object_count_after_gc(&_is_alive_closure);

    // Change under the freelistLocks.
    _collectorState = Sweeping;
    // Call isAllClear() under bitMapLock
    assert(_modUnionTable.isAllClear(),
        "Should be clear by end of the final marking");
    assert(_ct->cld_rem_set()->mod_union_is_clear(),
        "Should be clear by end of the final marking");
}
```

Final Remark 是最终的第二次标记，这种情况只有在 Background GC 执行了 InitialMarking 步骤的情形下才会执行，如果是 Foreground GC 执行的 Initial-Marking 步骤则不需要再次执行 FinalRemark。Final Remark 的开始阶段与 Init Mark 处理的流程相同，但是后续多了 Card Table 遍历、Reference 实例的清理并将其加入到 Reference 维护的 `pend_list` 中，如果要收集元数据信息，还要清理 SystemDictionary、CodeCache、SymbolTable、StringTable 等组件中不再使用的资源。

4.6.3 策略

知道了两个 STW 过程执行流程，我们分析解决就比较简单了，由于大部分问题都出在 Final Remark 过程，这里我们也拿这个场景来举例，主要步骤：

- **【方向】**观察详细 GC 日志，找到出问题时 Final Remark 日志，分析下 Reference 处理和元数据处理 real 耗时是否正常，详细信息需要通过 `-XX:+PrintReferenceGC` 参数开启。**基本在日志里面就能定位到大概是哪个方向出了问题，耗时超过 10% 的就需要关注**。

```
2019-02-27T19:55:37.920+0800: 516952.915: [GC (CMS Final Remark)
516952.915: [ParNew516952.939: [SoftReference, 0 refs, 0.0003857
secs]516952.939: [WeakReference, 1362 refs, 0.0002415 secs]516952.940:
[FinalReference, 146 refs, 0.0001233 secs]516952.940: [PhantomReference,
0 refs, 57 refs, 0.0002369 secs]516952.940: [JNI Weak Reference,
0.0000662 secs]
[class unloading, 0.1770490 secs]516953.329: [scrub symbol table,
0.0442567 secs]516953.373: [scrub string table, 0.0036072 secs][1 CMS-
remark: 1638504K(2048000K)] 1667558K(4352000K), 0.5269311 secs] [Times:
user=1.20 sys=0.03, real=0.53 secs]
```

- **【根因】**有了具体的方向我们就可以进行深入的分析，一般来说最容易出问题的地方就是 Reference 中的 FinalReference 和元数据信息处理中的 scrub symbol table 两个阶段，想要找到具体问题代码就需要内存分析工具 MAT 或 JProfiler 了，注意要 dump 即将开始 CMS GC 的堆。在用 MAT 等工具前也可以先用命令行看下对象 Histogram，有可能直接就能定位问题。
  - 对 FinalReference 的分析主要观察 `java.lang.ref.Finalizer` 对象的 dominator tree，找到泄漏的来源。经常会出现问题的几个点有 Socket 的 `SocksSocketImpl`、Jersey 的 `ClientRuntime`、MySQL 的 `ConnectionImpl` 等等。
  - scrub symbol table 表示清理元数据符号引用耗时，符号引用是 Java 代码被编译成字节码时，方法在 JVM 中的表现形式，生命周期一般与 Class 一致，当 `_should_unload_classes` 被设置为 true 时在 `CMSCollector::refProcessingWork()` 中与 Class Unload、String Table 一起被处理。

```
if (should_unload_classes()) {
    {
      GCTraceTime(Debug, gc, phases) t("Class Unloading", _gc_timer_cm);

      // Unload classes and purge the SystemDictionary.
      bool purged_class = SystemDictionary::do_unloading(_gc_timer_cm);

      // Unload nmethods.
      CodeCache::do_unloading(&_is_alive_closure, purged_class);

      // Prune dead klasses from subklass/sibling/implementor lists.
      Klass::clean_weak_klass_links(purged_class);
    }

    {
      GCTraceTime(Debug, gc, phases) t("Scrub Symbol Table", _gc_timer_cm);
      // Clean up unreferenced symbols in symbol table.
      SymbolTable::unlink();
    }

    {
      GCTraceTime(Debug, gc, phases) t("Scrub String Table", _gc_timer_
cm);
      // Delete entries for dead interned strings.
      StringTable::unlink(&_is_alive_closure);
    }
  }
```

- **【策略】**知道 GC 耗时的根因就比较好处理了，这种问题不会大面积同时爆发，不过有很多时候单台 STW 的时间会比较长，如果业务影响比较大，及时摘掉流量，具体后续优化策略如下：
  - FinalReference：找到内存来源后通过优化代码的方式来解决，如果短时间无法定位可以增加 `-XX:+ParallelRefProcEnabled` 对 Reference 进行并行处理。
  - symbol table：观察 MetaSpace 区的历史使用峰值，以及每次 GC 前后的回收情况，一般没有使用动态类加载或者 DSL 处理等，MetaSpace 的使用率上不会有什么变化，这种情况可以通过 `-XX:-CMSClassUnloadingEnabled` 来避免 MetaSpace 的处理，JDK8 会默认开启 CMSClassUnloadingEnabled，这会使得 CMS 在 CMS-Remark 阶段尝试进行类的卸载。

4.6.4 小结

正常情况进行的 Background CMS GC，出现问题基本都集中在 Reference 和 Class 等元数据处理上，在 Reference 类的问题处理方面，不管是 FinalReference，还是 SoftReference、WeakReference 核心的手段就是找准时机 dump 快照，然后用内存分析工具来分析。Class 处理方面目前除了关闭类卸载开关，没有太好的方法。

在 G1 中同样有 Reference 的问题，可以观察日志中的 Ref Proc，处理方法与 CMS 类似。

4.7 场景七：内存碎片 & 收集器退化

4.7.1 现象

并发的 CMS GC 算法，退化为 Foreground 单线程串行 GC 模式，STW 时间超长，有时会长达十几秒。其中 CMS 收集器退化后单线程串行 GC 算法有两种：

- 带压缩动作的算法，称为 MSC，上面我们介绍过，使用标记 - 清理 - 压缩，单线程全暂停的方式，对整个堆进行垃圾收集，也就是真正意义上的 Full GC，暂停时间要长于普通 CMS。
- 不带压缩动作的算法，收集 Old 区，和普通的 CMS 算法比较相似，暂停时间相对 MSC 算法短一些。

4.7.2 原因

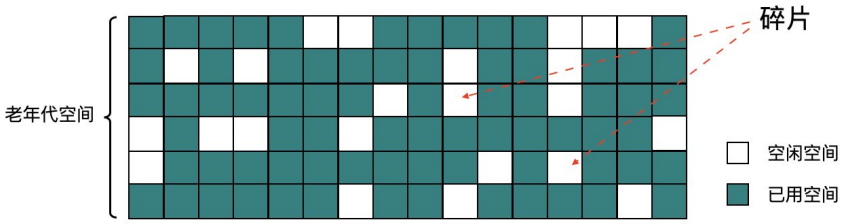
CMS 发生收集器退化主要有以下几种情况：

晋升失败（Promotion Failed）

顾名思义，晋升失败就是指在进行 Young GC 时，Survivor 放不下，对象只能放入 Old，但此时 Old 也放不下。直觉上乍一看这种情况可能会经常发生，但其实因为有 concurrentMarkSweepThread 和担保机制的存在，发生的条件是很苛刻的，除非是短时间将 Old 区的剩余空间迅速填满，例如上文中说的动态年龄判断导

致的过早晋升（见下文的增量收集担保失败）。另外还有一种情况就是内存碎片导致的 Promotion Failed，Young GC 以为 Old 有足够的空间，结果到分配时，晋级的大对象找不到连续的空间存放。

使用 CMS 作为 GC 收集器时，运行过一段时间的 Old 区如下图所示，清除算法导致内存出现多段的不连续，出现大量的内存碎片。



碎片带来了两个问题：

- **空间分配效率较低**：上文已经提到过，如果是连续的空间 JVM 可以通过使用 pointer bumping 的方式来分配，而对于这种有大量碎片的空闲链表则需要逐个访问 freelist 中的项来访问，查找可以存放新建对象的地址。
- **空间利用效率变低**：Young 区晋升的对象大小大于了连续空间的大小，那么将会触发 Promotion Failed ，即使整个 Old 区的容量是足够的，但由于其不连续，也无法存放新对象，也就是本文所说的问题。

增量收集担保失败

分配内存失败后，会判断统计得到的 Young GC 晋升到 Old 的平均大小，以及当前 Young 区已使用的大小也就是最大可能晋升的对象大小，是否大于 Old 区的剩余空间。只要 CMS 的剩余空间比前两者的任意一者大，CMS 就认为晋升还是安全的，反之，则代表不安全，不进行 Young GC，直接触发 Full GC。

显式 GC

这种情况参见场景二。

并发模式失败（Concurrent Mode Failure）

最后一种情况，也是发生概率较高的一种，在 GC 日志中经常能看到 Concurrent Mode Failure 关键字。这种是由于并发 Background CMS GC 正在执行，同时又有 Young GC 晋升的对象要放入到了 Old 区中，而此时 Old 区空间不足造成的。

为什么 CMS GC 正在执行还会导致收集器退化呢？主要是由于 CMS 无法处理浮动垃圾（Floating Garbage）引起的。CMS 的并发清理阶段，Mutator 还在运行，因此不断有新的垃圾产生，而这些垃圾不在这次清理标记的范畴里，无法在本次 GC 被清除掉，这些就是浮动垃圾，除此之外在 Remark 之前那些断开引用脱离了读写屏障控制的对象也算浮动垃圾。所以 Old 区回收的阈值不能太高，否则预留的内存空间很可能不够，从而导致 Concurrent Mode Failure 发生。

4.7.3 策略

分析到具体原因后，我们就可以针对性解决了，具体思路还是从根因出发，具体解决策略：

- **内存碎片：**通过配置 `-XX:UseCMSCompactAtFullCollection=true` 来控制 Full GC 的过程中是否进行空间的整理（默认开启，注意是 Full GC，不是普通 CMS GC），以及 `-XX: CMSFullGCsBeforeCompaction=n` 来控制多少次 Full GC 后进行一次压缩。
- **增量收集：**降低触发 CMS GC 的阈值，即参数 `-XX:CMSInitiatingOccupancyFraction` 的值，让 CMS GC 尽早执行，以保证有足够的连续空间，也减少 Old 区空间的使用大小，另外需要使用 `-XX:+UseCMSInitiatingOccupancyOnly` 来配合使用，不然 JVM 仅在第一次使用设定值，后续则自动调整。
- **浮动垃圾：**视情况控制每次晋升对象的大小，或者缩短每次 CMS GC 的时间，必要时可调节 NewRatio 的值。另外就是使用 `-XX:+CMSScavengeBeforeRemark` 在过程中提前触发一次 Young GC，防止后续晋升过多对象。

4.7.4 小结

正常情况下触发并发模式的 CMS GC，停顿非常短，对业务影响很小，但 CMS GC 退化后，影响会非常大，建议发现一次后就彻底根治。只要能定位到内存碎片、浮动垃圾、增量收集相关等具体产生原因，还是比较好解决的，关于内存碎片这块，如果 `-XX:CMSFullGCsBeforeCompaction` 的值不好选取的话，可以使用 `-XX:PrintFLSStatistics` 来观察内存碎片率情况，然后再设置具体的值。

最后就是在编码的时候也要避免需要连续地址空间的大对象的产生，如过长的字符串，用于存放附件、序列化或反序列化的 byte 数组等，还有就是过早晋升问题尽量在爆发问题前就避免掉。

4.8 场景八：堆外内存 OOM

4.8.1 现象

内存使用率不断上升，甚至开始使用 SWAP 内存，同时可能出现 GC 时间飙升，线程被 Block 等现象，**通过 top 命令发现 Java 进程的 RES 甚至超过了 `-Xmx` 的大小**。出现这些现象时，基本可以确定是出现了堆外内存泄漏。

4.8.2 原因

JVM 的堆外内存泄漏，主要有两种的原因：

- 通过 `UnSafe#allocateMemory`，`ByteBuffer#allocateDirect` 主动申请了堆外内存而没有释放，常见于 NIO、Netty 等相关组件。
- 代码中有通过 JNI 调用 Native Code 申请的内存没有释放。

4.8.3 策略

哪种原因造成的堆外内存泄漏?

首先，我们需要确定是哪种原因导致的堆外内存泄漏。这里可以使用 NMT（Native-MemoryTracking）进行分析。在项目中添加 `-XX:NativeMemoryTracking=detail` JVM 参数后重启项目（需要注意的是，打开 NMT 会带来 5%~10% 的性能损

耗)。使用命令 `jcmd pid VM.native_memory detail` 查看内存分布。重点观察 total 中的 committed，因为 jcmd 命令显示的内存包含堆内内存、Code 区域、通过 `Unsafe.allocateMemory` 和 `DirectByteBuffer` 申请的内存，但是不包含其他 Native Code（C 代码）申请的堆外内存。

如果 total 中的 committed 和 top 中的 RES 相差不大，则应为主动申请的堆外内存未释放造成的，如果相差较大，则基本可以确定是 JNI 调用造成的。

原因一：主动申请未释放

JVM 使用 `-XX:MaxDirectMemorySize=size` 参数来控制可申请的堆外内存的最大值。在 Java8 中，如果未配置该参数，默认和 `-Xmx` 相等。

NIO 和 Netty 都会取 `-XX:MaxDirectMemorySize` 配置的值，来限制申请的堆外内存的大小。NIO 和 Netty 中还有一个计数器字段，用来计算当前已申请的堆外内存大小，NIO 中是 `java.nio.Bits#totalCapacity`、Netty 中 `io.netty.util.internal.PlatformDependent#DIRECT_MEMORY_COUNTER`。

当申请堆外内存时，NIO 和 Netty 会比较计数器字段和最大值的大小，如果计数器的值超过了最大值的限制，会抛出 OOM 的异常。

NIO 中是：`OutOfMemoryError: Direct buffer memory`。

Netty 中是：`OutOfDirectMemoryError: failed to allocate capacity byte(s) of direct memory (used: usedMemory , max: DIRECT_MEMORY_LIMIT )`。

我们可以检查代码中是如何使用堆外内存的，NIO 或者是 Netty，通过反射，获取到对应组件中的计数器字段，并在项目中对该字段的数值进行打点，即可准确地监控到这部分堆外内存的使用情况。

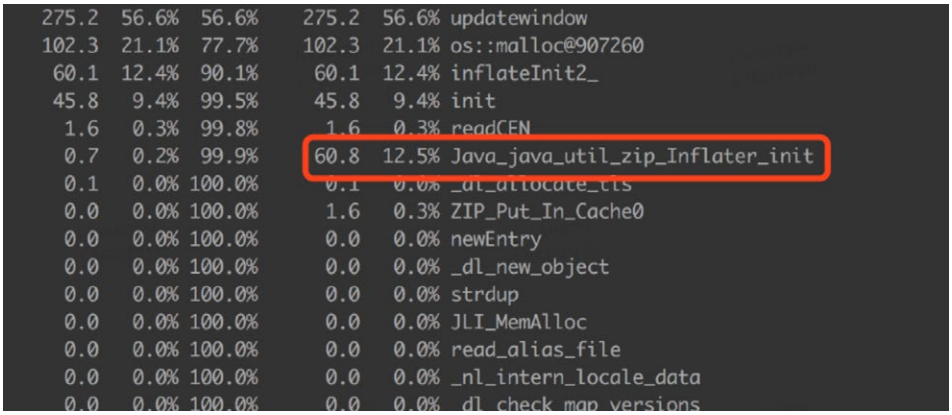
此时，可以通过 Debug 的方式确定使用堆外内存的地方是否正确执行了释放内存的代码。另外，需要检查 JVM 的参数是否有 `-XX:+DisableExplicitGC` 选项，如

果有就去掉，因为该参数会使 System.gc 失效。（场景二：显式 GC 的去与留）

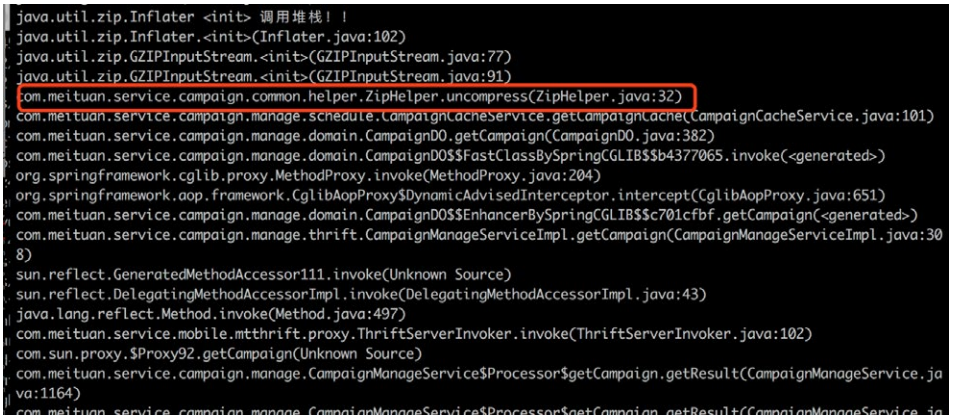
原因二：通过 JNI 调用的 Native Code 申请的内存未释放

这种情况排查起来比较困难，我们可以通过 Google perftools + Btrace 等工具，帮助我们分析出问题的代码在哪里。

gperftools 是 Google 开发的一款非常实用的工具集，它的原理是在 Java 应用程序运行时，当调用 malloc 时换用它的 libtcmalloc.so，这样就能对内存分配情况做一些统计。我们使用 gperftools 来追踪分配内存的命令。如下图所示，通过 gperftools 发现 `Java_java_util_zip_Inflater_init` 比较可疑。



接下来可以使用 Btrace，尝试定位具体的调用栈。Btrace 是 Sun 推出的一款 Java 追踪、监控工具，可以在不停机的情况下对线上的 Java 程序进行监控。如下图所示，通过 Btrace 定位出项目中的 `ZipHelper` 在频繁调用 `GZIPInputStream`，在堆外内存分配对象。



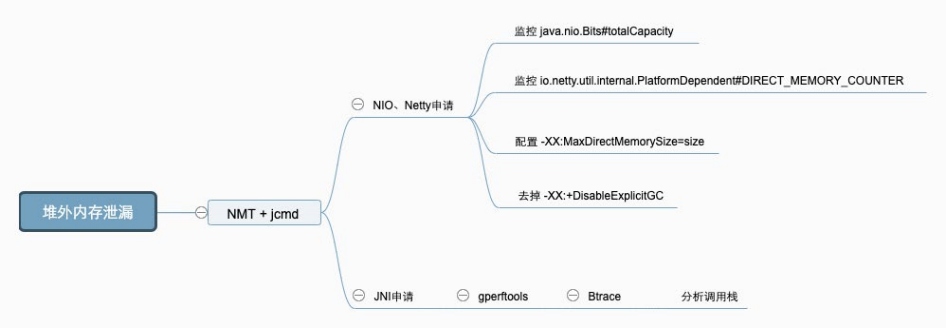
最终定位到是，项目中对 `GIPInputStream` 的使用错误，没有正确的 close()。

```
26 @    public static String uncompress(String str) throws Exception {
27          if (str == null || str.length() == 0) {
28              return str;
29          }
30          ByteArrayOutputStream out = new ByteArrayOutputStream();
31          ByteArrayInputStream in = new ByteArrayInputStream(str.getBytes( charsetName: "ISO-8859-1"));
32          GZIPInputStream gunzip = new GZIPInputStream(in);
33          byte[] buffer = new byte[1024];
34          int n;
35          while ((n = gunzip.read(buffer)) >= 0) {
36              out.write(buffer,  off: 0, n);
37          }
38          return out.toString();
39      }
40  }
```

除了项目本身的原因，还可能有外部依赖导致的泄漏，如 Netty 和 Spring Boot，详细情况可以学习下这两篇文章，Spring Boot 引起的“堆外内存泄漏”排查及经验总结、Netty 堆外内存泄露排查盛宴。

4.8.4 小结

首先可以使用 NMT + jcmd 分析泄漏的堆外内存是哪里申请，确定原因后，使用不同的手段，进行原因定位。



4.9 场景九：JNI 引发的 GC 问题

4.9.1 现象

在 GC 日志中，出现 GC Cause 为 GCLocker Initiated GC。

```
2020-09-23T16:49:09.727+0800: 504426.742: [GC (GCLocker Initiated GC)
504426.742: [ParNew (promotion failed): 209716K->6042K(1887488K),
0.0843330 secs] 1449487K->1347626K(3984640K), 0.0848963 secs] [Times:
user=0.19 sys=0.00, real=0.09 secs]
2020-09-23T16:49:09.812+0800: 504426.827: [Full GC (GCLocker Initiated
GC) 504426.827: [CMS: 1341583K->419699K(2097152K), 1.8482275 secs]
1347626K->419699K(3984640K), [Metaspace: 297780K->297780K(1329152K)],
1.8490564 secs] [Times: user=1.62 sys=0.20, real=1.85 secs]
```

4.9.2 原因

JNI（Java Native Interface）意为 Java 本地调用，它允许 Java 代码和其他语言写的 Native 代码进行交互。

JNI 如果需要获取 JVM 中的 String 或者数组，有两种方式：

- 拷贝传递。
- 共享引用（指针），性能更高。

由于 Native 代码直接使用了 JVM 堆区的指针，如果这时发生 GC，就会导致数据错误。因此，在发生此类 JNI 调用时，禁止 GC 的发生，同时阻止其他线程进入 JNI 临界区，直到最后一个线程退出临界区时触发一次 GC。

GC Locker 实验：

```
public class GCLockerTest {

  static final int ITERS = 100;
  static final int ARR_SIZE =  10000;
  static final int WINDOW = 10000000;

  static native void acquire(int[] arr);
  static native void release(int[] arr);

  static final Object[] window = new Object[WINDOW];

  public static void main(String... args) throws Throwable {
    System.loadLibrary("GCLockerTest");
    int[] arr = new int[ARR_SIZE];

    for (int i = 0; i < ITERS; i++) {
      acquire(arr);
      System.out.println("Acquired");
      try {
        for (int c = 0; c < WINDOW; c++) {
          window[c] = new Object();
        }
      } catch (Throwable t) {
        // omit
      } finally {
        System.out.println("Releasing");
        release(arr);
      }
    }
  }
}
```

```
#include <jni.h>
#include "GCLockerTest.h"

static jbyte* sink;

JNIEXPORT void JNICALL Java_GCLockerTest_acquire(JNIEnv* env, jclass
klass, jintArray arr) {
sink = (*env)->GetPrimitiveArrayCritical(env, arr, 0);
}

JNIEXPORT void JNICALL Java_GCLockerTest_release(JNIEnv* env, jclass
klass, jintArray arr) {
(*env)->ReleasePrimitiveArrayCritical(env, arr, sink, 0);
}
```

运行该 JNI 程序，可以看到发生的 GC 都是 GCLocker Initiated GC，并且注意在“Acquired”和“Released”时不可能发生 GC。

```
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
[GC (GCLocker Initiated GC)  1801127K->1269053K(4126208K), 0.1635153 secs]
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
Acquired
Releasing
[GC (GCLocker Initiated GC)  1942063K->1401284K(4126208K), 0.1379408 secs]
```

GC Locker 可能导致的不良后果有：

- 如果此时是 Young 区不够 Allocation Failure 导致的 GC，由于无法进行 Young GC，会将对象直接分配至 Old 区。
- 如果 Old 区也没有空间了，则会等待锁释放，导致线程阻塞。
- 可能触发额外不必要的 Young GC，JDK 有一个 Bug，有一定的几率，本来只该触发一次 GCLocker Initiated GC 的 Young GC，实际发生了一次 Allocation Failure GC 又紧接着一次 GCLocker Initiated GC。是因为 GCLocker Initiated GC 的属性被设为 full，导致两次 GC 不能收敛。

4.9.3 策略

- 添加 `-XX+PrintJNIGCStalls` 参数，可以打印出发生 JNI 调用时的线程，进一步分析，找到引发问题的 JNI 调用。
- JNI 调用需要谨慎，不一定可以提升性能，反而可能造成 GC 问题。
- 升级 JDK 版本到 14，避免 JDK-8048556 导致的重复 GC。

4.9.4 小结

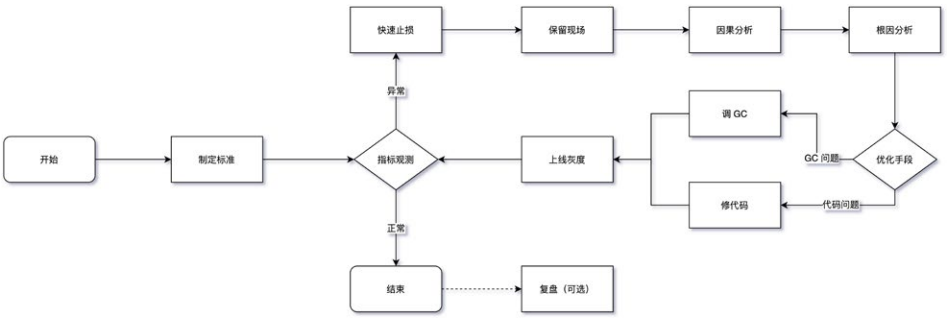
JNI 产生的 GC 问题较难排查，需要谨慎使用。

5. 总结

在这里，我们把整个文章内容总结一下，方便大家整体地理解回顾。

5.1 处理流程（SOP）

下图为整体 GC 问题普适的处理流程，重点的地方下面会单独标注，其他的基本都是标准处理流程，此处不再赘述，最后在整个问题都处理完之后有条件的话建议做一下复盘。



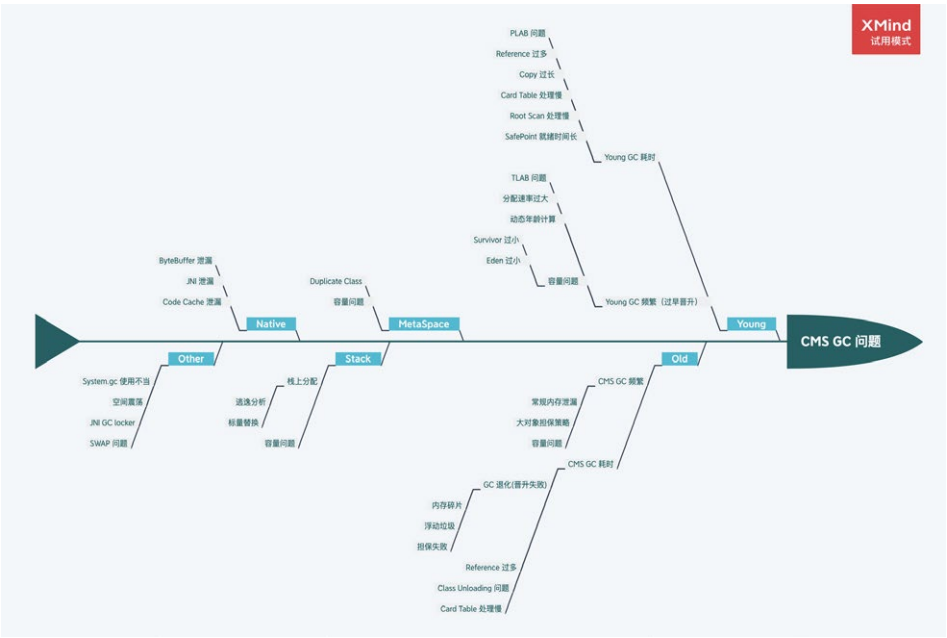
- **制定标准：**这块内容其实非常重要，但大部分系统都是缺失的，笔者过往面试的同学中只有不到一成的同学能给出自己的系统 GC 标准到底什么样，其他的都是用的统一指标模板，缺少预见性，具体指标制定可以参考 3.1 中的内容，

需要结合应用系统的 TP9999 时间和延迟、吞吐量等设定具体的指标，而不是被问题驱动。

- **保留现场：**目前线上服务基本都是分布式服务，某个节点发生问题后，如果条件允许一定不要直接操作重启、回滚等动作恢复，优先通过摘掉流量的方式来恢复，这样我们可以将堆、栈、GC 日志等关键信息保留下来，不然错过了定位根因的时机，后续解决难度将大大增加。当然除了这些，应用日志、中间件日志、内核日志、各种 Metrics 指标等对问题分析也有很大帮助。
- **因果分析：**判断 GC 异常与其他系统指标异常的因果关系，可以参考笔者在 3.2 中介绍的时序分析、概率分析、实验分析、反证分析等 4 种因果分析法，避免在排查过程中走入误区。
- **根因分析：**确实是 GC 的问题后，可以借助上文提到的工具并通过 5 why 根因分析法以及跟第三节中的九种常见的场景进行逐一匹配，或者直接参考下文的根因鱼骨图，找出问题发生根因，最后再选择优化手段。

5.2 根因鱼骨图

送上一张问题根因鱼骨图，一般情况下我们在处理一个 GC 问题时，只要能定位到问题的“病灶”，有的放矢，其实就相当于解决了 80%，如果在某些场景下不太好定位，大家可以借助这种根因分析图通过**排除法**去定位。



5.3 调优建议

- **Trade Off：**与 CAP 注定要缺一角一样，GC 优化要在延迟（Latency）、吞吐量（Throughput）、容量（Capacity）三者之间进行权衡。
- **最终手段：**GC 发生问题不是一定要对 JVM 的 GC 参数进行调优，大部分情况下是通过 GC 的情况找出一些业务问题，切记上来就对 GC 参数进行调整，当然有明确配置错误的场景除外。
- **控制变量：**控制变量法是在蒙特卡洛（Monte Carlo）方法中用于减少方差的一种技术方法，我们调优的时候尽量也要使用，每次调优过程尽可能只调整一个变量。
- **善用搜索：**理论上 99.99% 的 GC 问题基本都被遇到了，我们要学会使用搜索引擎的高级技巧，重点关注 StackOverFlow、Github 上的 Issue、以及各种论坛博客，先看看其他人是怎么解决的，会让解决问题事半功倍。能看到这篇文章，你的搜索能力基本过关了 ~
- **调优重点：**总体上来讲，我们开发的过程中遇到的问题类型也基本都符合正态分布，太简单或太复杂的基本遇到的概率很低，笔者这里将中间最重要的三个

场景添加了“*”标识，希望阅读完本文之后可以观察下自己负责的系统，是否存在上述问题。

- **GC 参数：**如果堆、栈确实无法第一时间保留，一定要保留 GC 日志，这样我们最起码可以看到 GC Cause，有一个大概的排查方向。关于 GC 日志相关参数，最基本的 `-XX:+HeapDumpOnOutOfMemoryError` 等一些参数就不再提了，笔者建议添加以下参数，可以提高我们分析问题的效率。

分类	参数	作用
基本参数	`-XX:+PrintGCDetails`、`-XX:+PrintGCDateStamps`、`-XX:+PrintGCTimeStamps`	GC 日志的基本参数
时间相关	`-XX:+PrintGCApplicationConcurrentTime`、`-XX:+PrintGCApplicationStoppedTime`	详细步骤的并行时间，STW 时间等等
年龄相关	`-XX:+PrintTenuringDistribution`	可以观察 GC 前后的对象年龄分布，方便发现过早晋升问题
空间变化	`-XX:+PrintHeapAtGC`	各个空间在 GC 前后的回收情况，非常详细
引用相关	`-XX:+PrintReferenceGC`	观察系统的软引用，弱引用，虚引用等回收情况

- **其他建议：**上文场景中没有提到，但是对 GC 性能也有提升的一些建议。
  - **主动式 GC：**也有另开生面的做法，通过监控手段监控观测 Old 区的使用情况，即将到达阈值时将应用服务摘掉流量，手动触发一次 Major GC，减少 CMS GC 带来的停顿，但随之系统的健壮性也会减少，如非必要不建议引入。
  - **禁用偏向锁：**偏向锁在只有一个线程使用到该锁的时候效率很高，但是在竞争激烈情况会升级成轻量级锁，此时就需要先**消除偏向锁，这个过程是 STW** 的。如果每个同步资源都走这个升级过程，开销会非常大，所以在已知并发激烈的前提下，一般会禁用偏向锁 `-XX:-UseBiasedLocking` 来提高性能。
  - **虚拟内存：**启动初期有些操作系统（例如 Linux）并没有真正分配物理内存给 JVM ，而是在虚拟内存中分配，使用的时候才会在物理内存中分配内存页，这样也会导致 GC 时间较长。这种情况可以添加 `-XX:+AlwaysPreTouch` 参数，让 VM 在 commit 内存时跑个循环来强制保证申请的内存真的 commit，避免运行时触发缺页异常。在一些大内存的场景下，有时候能将前几次的 GC 时间降一个数量级，但是添加这个参数后，启动的过程可能会变慢。

6. 写在最后

最后，再说笔者个人的一些小建议，遇到一些 GC 问题，如果有精力，一定要探本穷源，找出最深层次的原因。另外，在这个信息泛滥的时代，有一些被“奉为圭臬”的经验可能都是错误的，尽量养成看源码的习惯，有一句话说到“源码面前，了无秘密”，也就意味着遇到搞不懂的问题，我们可以从源码中一窥究竟，某些场景下确有奇效。但也不是只靠读源码来学习，如果硬啃源码但不理会其背后可能蕴含的理论基础，那很容易“捡芝麻丢西瓜”，“只见树木，不见森林”，让“了无秘密”变成了一句空话，我们还是要结合一些实际的业务场景去针对性地学习。

你的时间在哪里，你的成就就会在哪里。笔者也是在前两年才开始逐步地在 GC 方向上不断深入，查问题、看源码、做总结，每个 Case 形成一个小的闭环，目前初步摸到了 GC 问题处理的一些门道，同时将经验总结应用于生产环境实践，慢慢地形成一个良性循环。

本篇文章主要是介绍了 CMS GC 的一些常见场景分析，另外一些，如 CodeCache 问题导致 JIT 失效、SafePoint 就绪时间长、Card Table 扫描耗时等问题不太常见就没有花太多篇幅去讲解。Java GC 是在“分代”的思想下内卷了很多年才突破到了“分区”，目前在美团也已经开始使用 G1 来替换使用了多年的 CMS，虽然在小的堆方面 G1 还略逊色于 CMS，但这是一个趋势，短时间无法升级到 ZGC，所以未来遇到的 G1 的问题可能会逐渐增多。目前已经收集到 Remember Set 粗化、Humongous 分配、Ergonomics 异常、Mixed GC 中 Evacuation Failure 等问题，除此之外也会给出 CMS 升级到 G1 的一些建议，接下来笔者将继续完成这部分文章整理，敬请期待。

“防火”永远要胜于“救火”，**不放过任何一个异常的小指标**（一般来说，任何**不平滑的曲线**都是值得怀疑的），就有可能避免一次故障的发生。作为 Java 程序员基本都会遇到一些 GC 的问题，独立解决 GC 问题是我们必须迈过的一道坎。开篇中也提到过 GC 作为经典的技术，非常值得我们学习，一些 GC 的学习材料，如《The Garbage Collection Handbook》《深入理解 Java 虚拟机》等也是常读常新，赶紧

动起来，苦练 GC 基本功吧。

最后的最后，再多啰嗦一句，目前所有 GC 调优相关的文章，第一句讲的就是“不要过早优化”，使得很多同学对 GC 优化望而却步。在这里笔者提出不一样的观点，熵增定律（在一个孤立系统里，如果没有外力做功，其总混乱度（即熵）会不断增大）在计算机系统同样适用，**如果不主动做功使熵减，系统终究会脱离你的掌控**，在我们对业务系统和 GC 原理掌握得足够深的时候，可以放心大胆地做优化，因为我们基本可以预测到每一个操作的结果，放手一搏吧，少年！

7. 参考资料

[1]《ガベージコレクションのアルゴリズムと実装》中村 成洋 / 相川 光
[2]《The Garbage Collection Handbook》Richard Jones/ Antony Hosking / Eliot Moss
[3]《深入理解 Java 虚拟机（第 3 版）》周志明
[4]《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》
[5]《Shipilev One Page Blog》Shipilëv
[6] https://openjdk.java.net/projects/jdk/15/
[7] https://jcp.org/en/home/index
[8]《A Generational Mostly-concurrent Garbage Collector》Tony Printezis / David Detlefs
[9]《Java Memory Management White Paper》
[10]《Stuff Happens：Understanding Causation in Policy and Strategy》AA Hill

8. 作者简介

新宇：2015 年加入美团，到店住宿门票业务开发工程师。
湘铭：2018 年加入美团，到店客户平台开发工程师。
祥璞：2018 年加入美团，到店客户平台开发工程师。

9. 招聘信息

美团到店事业群住宿门票数据智能组诚招小伙伴，从供、控、选、售等层面全方位提升业务竞争力，十万级 QPS 处理，亿级数据分析，完整业务闭环，目前有海量 HC，有兴趣的请将邮件发送至 hezhiming@meituan.com，我们会在第一时间与你联系。

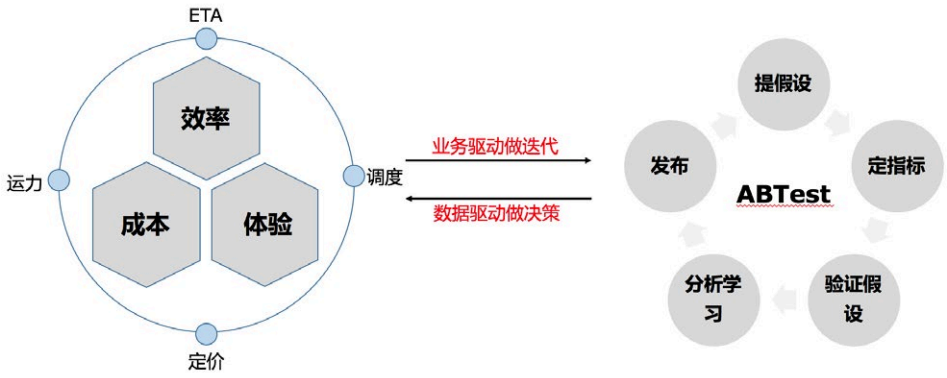
美团配送 A/B 评估体系建设实践

作者：王鹏　启政　连恒

2019 年 5 月 6 日，美团点评正式推出新品牌“美团配送”，发布了美团配送新愿景：“每天完成一亿次值得信赖的配送服务，成为不可或缺的生活基础设施。”现在，美团配送已经服务于全国 400 多万商家和 4 亿多用户，覆盖 2800 余座市县，日活跃骑手超过 70 万人，成为全球领先的分钟级配送网络。

即时配送的三要素是“效率”、“成本”、“体验”，通过精细化的策略迭代来提升效率，降低成本，提高体验，不断地扩大规模优势，从而实现正向循环。但是，策略的改变，不是由我们随便“拍脑袋”得出，而是一种建立在数据基础上的思维方式，数据反馈会告诉我们做的好不好，哪里有问题，以及衡量可以带来多少确定性的增长。而 A/B-test 就是我们精细化迭代的一个“利器”，通过为同一个迭代目标制定两个或多个版本的方案，在同一时间维度，让组成成分相同（或相似）的 A/B 群组分别采用这些版本，然后收集各群组的体验数据和业务数据，最后分析、评估出最好的版本，帮助我们作出正确的决策，使迭代朝着更好的方向去演进。基于此，构建一个适用于配送业务的 A/B 平台就应运而生了。

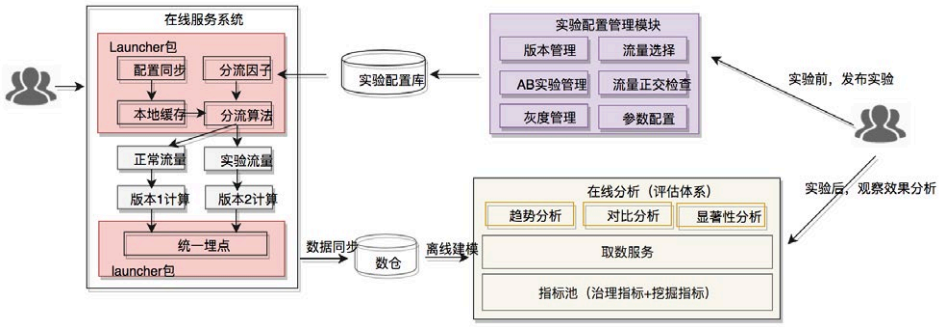
1. A/B 平台简介



如上图所示，A/B 实验可以看作一个“无尽”的学习环，我们通过提出假设、定义成功指标、检验假设（A/B 实验）、分析学习、发布、建立另一个假设，这就形成一个完整的闭环，通过多轮实验迭代，使策略趋于更优。基于上述对 A/B 实验划分的 5 个步骤，我们将 A/B 实验的完整生命周期分为三个阶段：

- 实验前，提出该实验假设，定义实验成功的指标，确定分流策略；
- 实验中，即验证假设的阶段，根据配置阶段的分流策略进行分流和埋点上报；
- 实验后，进行实验分析与学习，并基于实验报告决定是否发布。

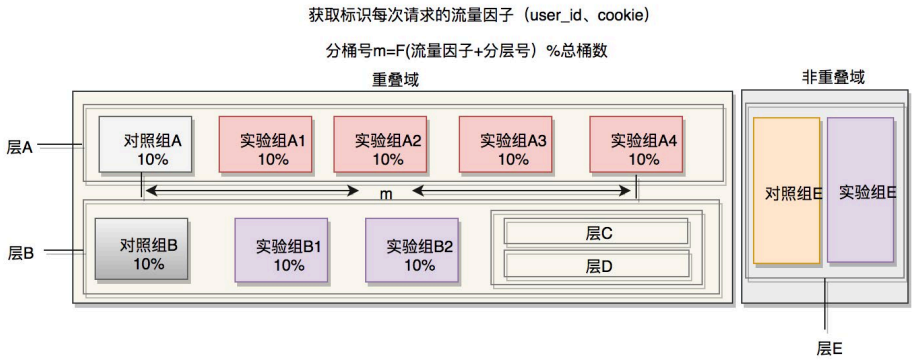
按照功能划分，我们将 A/B 平台分为三个模块，实验配置管理模块、分流以及埋点上报模块和在线分析模块，分别对应于 A/B 实验生命周期的实验前、实验中和实验后三个阶段。在实验配置模块，用户可以基于实验前提出的假设、定义的成功指标快速创建实验，并基于特定的分流策略完成分流配置；分流以及埋点上报模块，提供 JAR 包接入的形式，异步获取实验配置进行本地分流计算和埋点上报；在线分析模块，依据用户在实验配置管理模块选取的用于说明实验效果的指标、分流埋点上报模块记录的日志，自动地产生各实验的实验报告，供实验观察者使用，然后根据实验效果帮助他们作出正确的决策。具体流程如下图所示：



2. 为什么要强调评估体系建设

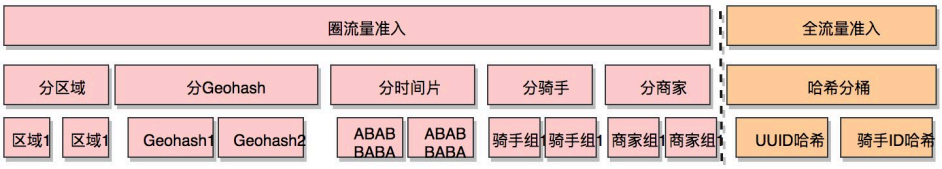
2.1 分流业务场景需要

业界的A/B平台建设基本以《Overlapping Experiment Infrastructure: More, Better, Faster Experimentation》这篇论文为蓝本进行展开，引入分层模型以及在分流算法中加入层编号因子来解决“流量饥饿”和“正交”问题，并且通过引入域的概念，支持域和层之间的相互嵌套，使分层实验模型更加灵活，进而满足多种场景下的A/B诉求。如下图所示，将流量通过Hash取模的方式即可实现流量的均匀划分。



这种是面向C端用户进行流量选择的传统A/B实验，采用上述的分流方式基于这样的假设：参与实验的流量因子是相互独立的、随机的，服从独立同分布。但是，配送业务场景下的A/B实验，涉及到用户、骑手、商家三端，请求不独立，策略之间相互影响并且受线下因素影响较大。传统A/B实验的分流方式，无法保证分出的两个群组

实验组和对照组的流量都是无差别的，无法避免因流量分配不平衡而导致的 A/B 群组差异过大问题，很容易造成对实验结果的误判。为满足不同业务场景的诉求，我们的 A/B 平台建设采取了多种分流策略，如下图所示：



针对策略之间的相互影响、请求不独立场景下的 A/B 实验，我们采取限流准入的分流方式，针对不同的实验，选取不同的分流因子。在实验前，我们通过 AA 分组，找出无差别的实验组和对照组，作为我们实验分流配置的依据，这种分流方式要求我们要有一套完整刻画流量因子的指标体系，只要刻画流量因子的指标间无统计显著性，我们就认为分出的实验组和对照组无差别。

2.2 业务决策的重要依据

在实验后的效果评估环节，通常允许实验者用自定义的指标来衡量不同策略带来的影响。但这样做会带来如下两个问题：

- 首先，由实验者来负责实验效果的评估，很难做到客观。同时也无法避免实验者仅仅选择支持自己假设的指标，来证明自己的实验结论；
- 其次，所有的策略迭代都是为业务服务，如果实验者用自定义的、与业务认知不一致的指标，来说明实验效果、推动业务灰度，这种方式往往难以被采纳。

因此，权威的评估体系对于对齐大家认知，并帮助我们在策略迭代方面作出正确的决策，尤为重要。

3.A/B 评估体系构建

A/B 评估体系的构建，要解决 A/B 平台两个核心问题：

- 第一，要有一套用于刻画流量因子（区域、骑手、商家）的权威的、完备的指标体系，帮助实验者完成实验前的 AA 分组和实验后的效果评估；
- 第二，要建立一套科学的评估方法，帮助实验者作出正确的决策。

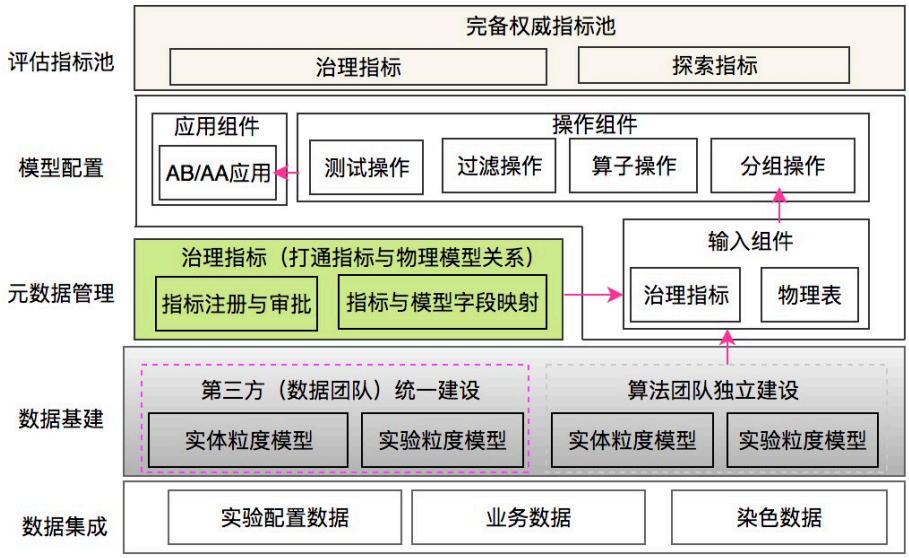
3.1 权威完备的指标体系

指标的权威性体现在：刻画分流因子，用于实验前 AA 分组和证明实验假设的指标，必须经过治理且业务认知一致，这样才能对齐认知，使得实验结果更具说服力；指标的完备性体现在：评估体系中的指标，不仅要有经过第三方独立生产治理且各业务方认知一致的治理指标，而且还要有实验者为了更全面的分析，描述实验过程，自定义的探索指标。

3.1.1 整体架构

治理指标强调的是指标的权威性和生产的规范性，而探索性指标强调的是指标的多样性和生产的灵活性。在评估体系中要实现这两类指标的统一，既要包含用于说明实验效果的治理指标，又要包含帮助实验者更好迭代实验所需的探索指标。

为实现上述的统一，指标层面要有分级运营的策略：治理指标按照业务认知一致性和算法内部认知一致性分别定级为 P0、P1，这一类指标在生产前必须要有严格的注册、评审，生产环节需要交给独立的第三方团队（数据团队）生产，保证指标的权威性，产出后打通指标与字段的映射关系，对用户屏蔽底层实现逻辑；对于探索性指标，定级为 P2，强调的是生产的灵活性和快速实现，因此，它的生产就不宜带有指标注册和评审等环节。为保证其快速实现，希望基于物理表和简单的算子配置就可以实现效果分析时即席查询使用。基于如上的问题拆解，我们进行了如下的架构设计：



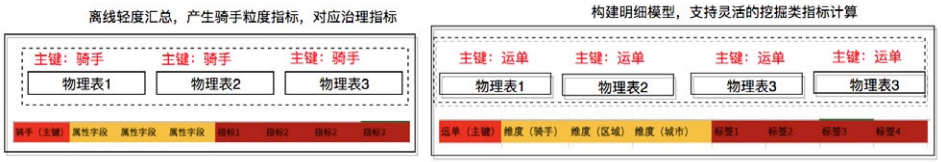
3.1.2 数据集成

为了支持监控和分析，在数据集成环节，我们集成了实验配置数据、业务数据和染色数据，以便实验者在效果评估环节不仅可以查看流量指标（PV、UV 和转化率），也可以深入探索策略变动对业务带来的影响。对于那些在实验配置环节不能确定流量是否真正参加实验的场景（例如：选择了特定区域进行实验，该区域产生的单只有满足特定条件时才能触发实验），我们不能直接通过限制确定的区域来查看业务指标。因为，此时查看的指标并不是真正参与实验的流量所对应的指标。因此在数据集成环节，我们同时将实验前的实验配置数据和实验中的染色数据（针对每个参与实验的流量，每次操作所产生的数据，都会打上实验场景、实验组以及具体的分组标记，我们该数据为染色数据）同步到数仓。在数据基建环节，将业务数据模型和染色数据模型通过流量实体作为关联条件进行关联，构建实验粒度模型。

3.1.3 数据基建

在数据基建层，我们基于指标分级运营的思路，由数据团队和算法团队分别构建实体粒度（区域、骑手、GeoHash）和实验粒度的实体宽表模型，以满足 P0/P1 指标和 P2 指标的诉求；为实现指标的规范化建设和灵活建设的统一，在物理模型和对外

提供应用的指标池之间，我们提供了元数据管理工具和模型配置工具，从而实现离线数据快速接入评估体系的指标池。由数据团队建设的实体宽表模型，对应着治理指标（P0/P1 指标），必须在生产后通过元数据管理工具完成指标与物理字段的映射，将指标的加工口径封装在数据层，对用户屏蔽物理实现，确保治理指标的一致性。由算法团队独立建设的实体宽表模型，对应着挖掘指标（P2 指标），为确保其接入评估体系指标池的灵活性和方便性，我们在数据基建环节，通过标签的形式对指标口径做部分封装，在模型配置环节完成指标逻辑的最终加工。

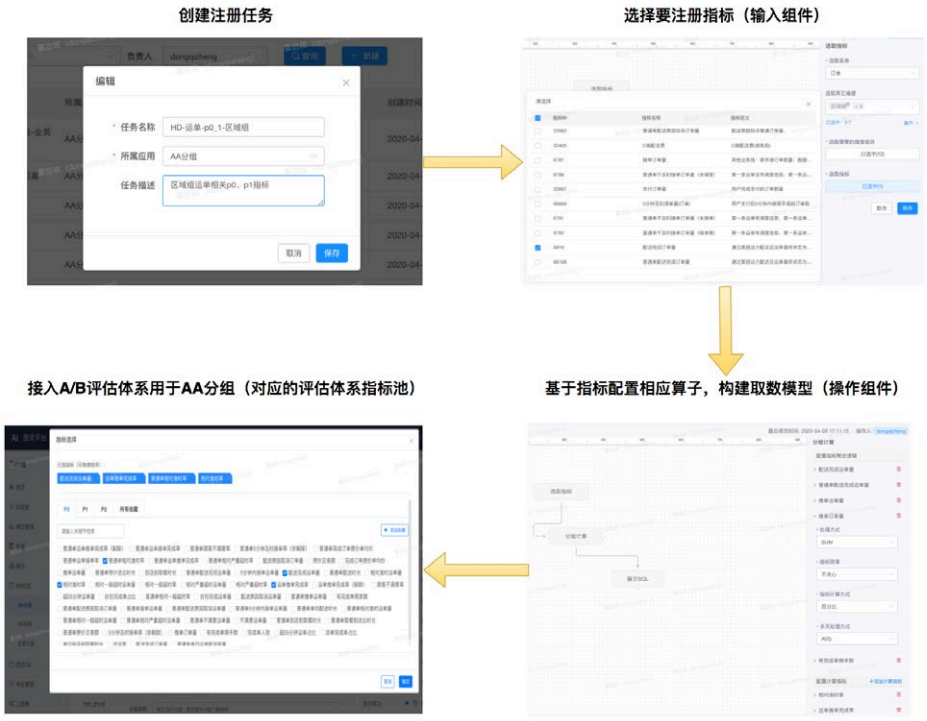


3.1.4 元数据管理

元数据管理层，是实现指标权威性的关键。治理指标在本层实现注册、评审，达到业务认知一致性和算法内部认知一致性的目的。同时，本层还完成了治理指标与数据基建层物理模型之间的绑定，为后续的模型配置建立基础。

3.1.5 模型配置

模型配置工具，是打通物理模型与评估指标池的桥梁，它通过输入组件、操作组件和应用组件，将离线数据接入到评估体系中，满足实验前 AA 分组和实验后 AB 评估的需求。首先，输入组件可以对应不同的数据源，既可以接入治理的离线指标，也可以接入特定库下的物理表。其次，操作组件提供了分组操作、算子操作、过滤操作和测试操作，通过分组操作，确定模型包含的维度；通过算子操作，将算子作用在指标或标签字段上，在取数环节实现指标的二次计算；通过过滤操作，实现数据的过滤；通过测试操作，保证模型配置质量。最后，应用组件可以将配置的模型注册到不同的应用上，针对 A/B 场景主要是 AA 分组和 AB 评估。具体接入流程如下图所示：



3.2 科学权威的评估方式

评估报告的可靠和权威性主要体现在两个方面：一是评估指标的可靠性和权威性；二是评估方式的科学性。在上一节中，我们重点讨论了如何构建可靠权威的指标体系。在这一节，我们重点讨论如何进行科学的评估。

在讨论科学评估之前，我们再重温一下 A/B 实验的定义：A/B 实验，简单来说，就是为同一个目标制定两个版本或多个版本的方案，在同一时间维度，分别让组成成分相同（相似）的 A/B 群组分别采用这些版本，收集各群组的体验数据和业务数据，最后分析、评估出最好版本，正式采用。其中 A 方案为现行的设计（称为控制组），B 方案是新的设计（称为实验组）。分析 A/B 实验的定义，要实现科学权威的评估，最重要的两点在于：

- 第一，确保在实验前分出无差别的实验组和对照组，避免因流量分配不平衡导致的 AB 群组差异过大，最终造成对于实验结果的误判；

- 第二，确保对实验结果作出准确的判断，能够准确的判断新策略相对于旧策略的优势是不是由自然波动引起的，它的这一优势能否在大规模的推广中反映出来。

无论是实验前确保实验组和对照组流量无显著性差异，还是实验后新策略较旧策略的指标变动是否具有统计上的显著性，无一例外，它们都蕴含着统计学的知识。接下来，我们重点论述一下 A/B 实验所依赖的统计学基础以及如何依据统计学理论做出科学评估。

3.2.1 假设检验

3.2.1.1 两个假设

A/B 测试是一种对比试验，我们圈定一定的流量进行实验，实验结束后，我们基于实验样本进行数据统计，进而验证实验前假设的正确性，我们得出这一有效结论的科学依据便是假设检验。假设检验是利用样本统计量估计总体参数的方法，在假设检验中，先对总体均值提出一个假设，然后用样本信息去检验这个假设是否成立。我们把提出的这个假设叫做原假设，与原假设对立的结论叫做备择假设，如果原假设不成立，就要拒绝原假设，进而接受备择假设。

3.2.1.2 两类错误

对于原假设提出的命题，我们需要作出判断，要么原假设成立，要么原假设不成立。因为基于样本对总体的推断，会面临着犯两种错误的可能：第一类错误，原假设为真，我们却拒绝了；第二类错误，原假设为伪，我们却接受了。显然，我们希望犯这两类错误的概率越小越好，但对于一定的样本量 n，不能同时做到犯这两类错误的概率很小。

在假设检验中，就有一个对两类错误进行控制的问题。一般来说，哪一类错误所带来的后果严重、危害越大，在假设检验中就应该把哪一类错误作为首要的控制目标。在假设检验中，我们都执行这样一个原则，首先控制犯第一类错误的概率。这也是为什么我们在实际应用中会把要推翻的假设作为原假设，这样得出的结论更具说服力（我

们有足够充分的证据证明原来确定的结论是错误的），所以通常会看到，我们把要证明的结论作为备择假设。

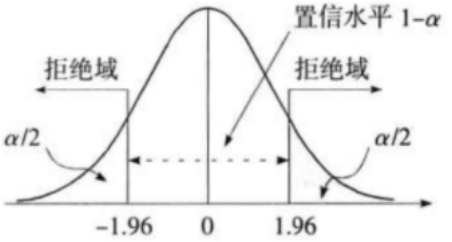
3.2.1.3 T 检验

常见的假设检验方法有 Z 检验、T 检验和卡方检验等，不同的方法有不同的适用条件和检验目标。Z 检验和 T 检验都是用来推断两个总体均值差异的显著性水平，具体选择哪种检验由样本量的大小、总体的方差是否已知决定。在样本量较小且总体的方差未知的情况下，这时只能使用样本方差代替总体方差，样本统计量服从 T 分布，应该采用 T 统计量进行检验。T 统计量具体构造公式如下图所示，其中 f 是 T 统计量的自由度，S1、S2 是样本标准差。

$$t=\frac{(\bar{x}_1-\bar{x}_2)-(\mu_1-\mu_2)}{\sqrt{\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}}}$$

$$f=\frac{\left(\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1-1}+\frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2-1}}$$

T 检验的流程是，在给定的弃真错误概率下（一般取 0.05），依据样本统计量 T 是否落在拒绝域来判断接受还是拒绝原假设。实际上在确定弃真错误概率以后，拒绝域的位置也就相应地确定了。使用 T 统计量进行判断的好处是，进行决策的界限清晰，但缺陷是决策面临的风险是笼统的。例如 T=3 落入拒绝域，我们拒绝原假设，犯弃真错误的概率为 0.05；T=2 也落入拒绝域，我们拒绝原假设，犯弃真错误的概率也是 0.05。事实上，依据不同的统计量进行决策，面临的风险也是有差别的。为了精确地反映决策的风险度，我们仍然需要 P 值来帮助业务来做决策。



3.2.1.4 利用 P 值决策

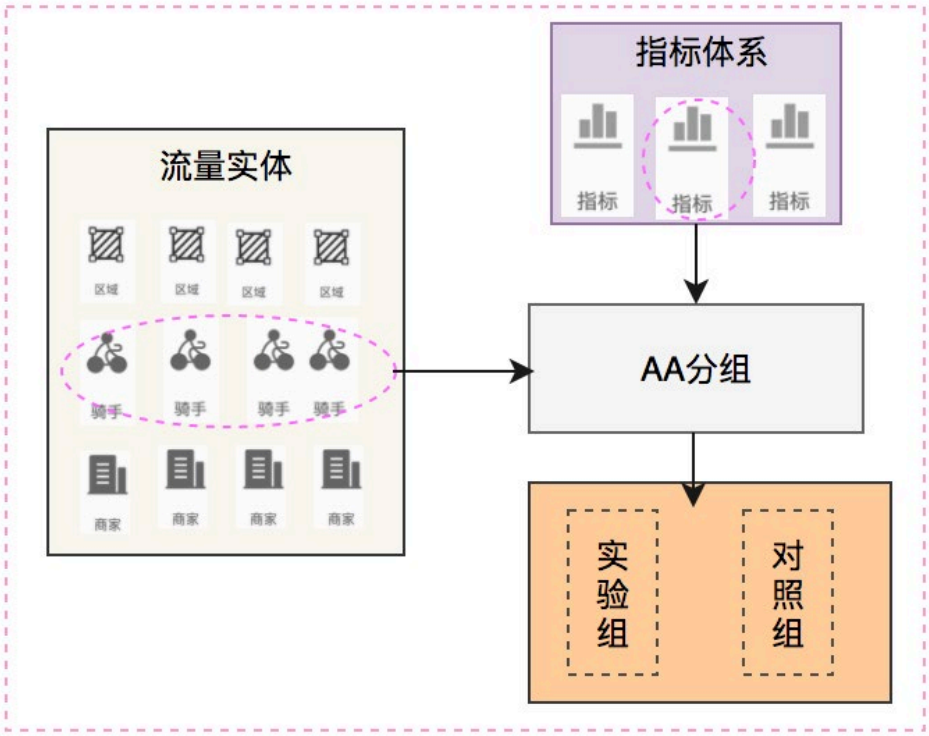
P 值是当原假设为真时，所得到的样本观察结果或更极端的结果出现的概率。如果 P 值很小，说明这种情况发生的概率很小，但是在这次试验中却出现了，根据小概率原理，我们有理由拒绝原假设，P 值越小，我们拒绝原假设的理由越充分。P 值可以理解为犯弃真错误的概率，在确定的显著性水平下（一般取 0.05），P 值小于显著性水平，则拒绝原假设。

3.2.2 基于假设检验的科学评估

围绕着科学评估要解决的两个问题，实验前，针对圈定的流量使用假设检验加上动态规划算法，确保分出无差别的实验组和对照组；实验后，基于实验前选定的用于验证假设结论的指标，构造 T 统计量并计算其对应的 P 值，依据 P 值帮我们做决策。

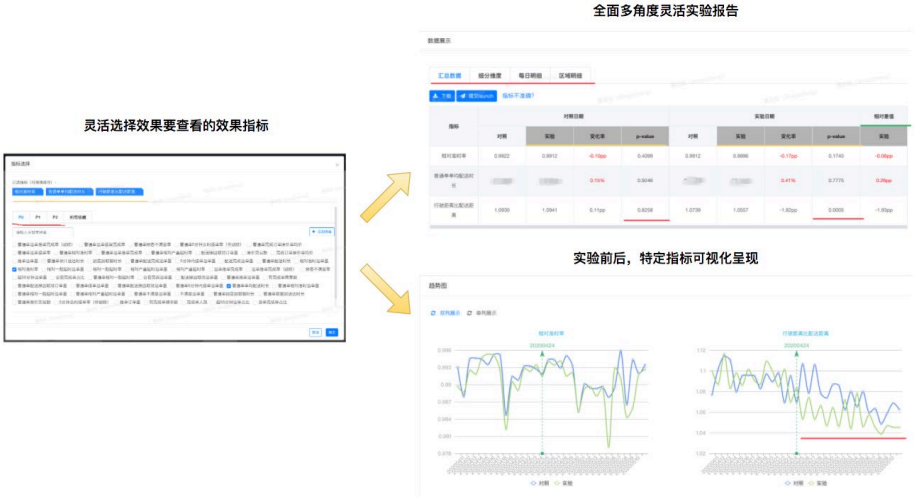
3.2.2.1 AA 分组

首先看如何解决第一个问题：避免因流量分配不平衡，A/B 组本身差异过大造成对实验结果的误判。为解决该问题，我们引入了 AA 分组：基于实验者圈定的流量，通过 AA 分组将该流量分为无显著性差异的实验组和对照组。我们这样定义无显著性差异这一约束：首先，实验者选取的用于刻画实验流量的指标，在实验组和对照组之间无统计上的显著性（即上节所描述的基于均值的假设检验）；其次，在所分出的实验组和对照组之间，这些指标的差值最小，即一个寻找最优解的过程。从实验者的实验流程看，在实验前，圈定进入该实验的流量，然后确定用于刻画实验流量的指标，最后调用 AA 分组，为其将流量分成合理的实验组和对照组。



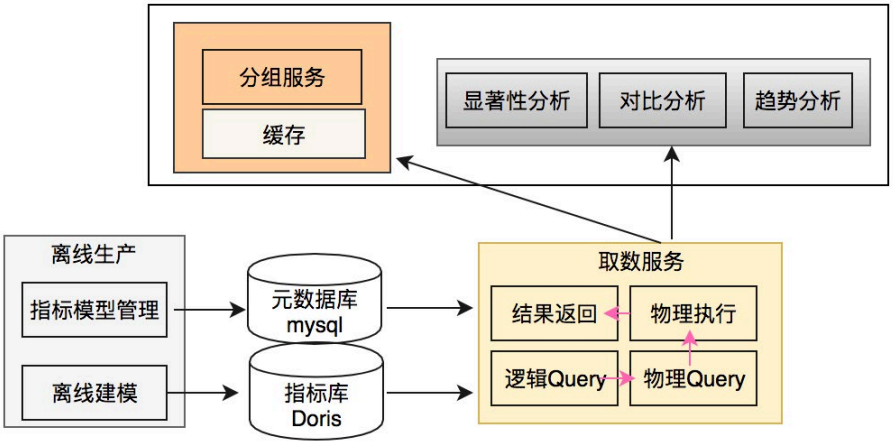
3.2.2.2 A/B 效果评估

A/B 效果评估是实验者在实验后，依据评估报告进行决策的重要依据。 因此，我们在实验后的效果评估环节，效果评估要达成三个目标即权威、灵活性和方便。首先，权威性体现在用于作出实验结论所依赖的指标都是经过治理、各方达成一致的指标，并且确保数据一致性，最终通过假设检验给出科学的实验结论，帮助实验者作出正确的判断。其次，灵活性主要体现在采用列转行的形式，按需自动生成报表告别“烟囱式”的报表开发方式。第三，方便主要体现在不仅可以查看用于说明实验效果的指标，还可以选择查看接入到评估体系里的任意指标；不仅可以查看其实验前后对比以及趋势变化，还可以做到从实验粒度到流量实体粒度的下钻。效果如下图所示：



3.2.2.3 技术实现

不管是实验前的 AA 分组，还是实验后的效果评估，我们要解决的一个核心问题就是如何灵活地“取数”，为我们的 AA 分组和 AB 效果分析提供一个灵活稳定的取数服务。因此，我们整个架构的核心就是构建稳定、灵活的取数服务，具体架构如下图所示。离线建模和指标模型管理完成数据和元数据建设，建立权威完备的指标体系；中间的取数服务作为上层各应用服务和指标体系的“桥梁”，为上层各应用服务提供其所依赖的指标。



4. 总结与展望

目前，A/B 测试已成为许多互联网公司评估其新产品策略和方法的“金标准”，在美团配送业务场景下，它被广泛应用于调度策略、定价策略、运力优化、ETA 时间预估等业务场景，为我们的策略迭代制定数据驱动型决策。特别是针对配送场景下这种策略之间相互影响，请求不独立场景下的 A/B 实验，结合配送技术团队的具体实践，跟大家分享了我们目前的解决思路。

最后再补充一点，在 A/B 测试领域，实验的流量规模应该有足够的统计能力，才能确保指标的变化有统计意义的，为了更好地达到这个目标，未来我们将通过辅助工具建设，在实验前，依据实验者所关注的指标以及敏感度给出流量规模的建议，方便实验者在实验前快速地圈定其实验所需的流量。

5. 作者简介

王鹏，2016 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、数据治理及系统化和 A/B 评估体系建设相关工作。

启政，2018 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、A/B 评估体系建设相关工作。

连恒，2016 年加入美团点评，目前在配送事业部数据团队负责众包业务数据建设、A/B 评估体系建设相关工作。

新一代垃圾回收器 ZGC 的探索与实践

作者：王东　王伟

ZGC（The Z Garbage Collector）是 JDK 11 中推出的一款低延迟垃圾回收器，它的设计目标包括：

- 停顿时间不超过 10ms；
- 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
- 支持 8MB~4TB 级别的堆（未来支持 16TB）。

从设计目标来看，我们知道 ZGC 适用于大内存低延迟服务的内存管理和回收。本文主要介绍 ZGC 在低延时场景中的应用和卓越表现，文章内容主要分为四部分：

- **GC 之痛**：介绍实际业务中遇到的 GC 痛点，并分析 CMS 收集器和 G1 收集器停顿时间瓶颈；
- **ZGC 原理**：分析 ZGC 停顿时间比 G1 或 CMS 更短的本质原因，以及背后的技术原理；
- **ZGC 调优实践**：重点分享对 ZGC 调优的理解，并分析若干个实际调优案例；
- **升级 ZGC 效果**：展示在生产环境应用 ZGC 取得的效果。

GC 之痛

很多低延迟高可用 Java 服务的系统可用性经常受 GC 停顿的困扰。GC 停顿指垃圾回收期间 STW（Stop The World），当 STW 时，所有应用线程停止活动，等待 GC 停顿结束。以美团风控服务为例，部分上游业务要求风控服务 65ms 内返回结果，并且可用性要达到 99.99%。但因为 GC 停顿，我们未能达到上述可用性目标。当时使用的是 CMS 垃圾回收器，单次 Young GC 40ms，一分钟 10 次，接口平均响应时间 30ms。通过计算可知，有（40ms + 30ms）* 10 次 / 60000ms = 1.12% 的请求的响应时间会增加 0 ~ 40ms 不等，其中 30ms * 10 次 / 60000ms = 0.5% 的请求

响应时间会增加 40ms。可见，GC 停顿对响应时间的影响较大。为了降低 GC 停顿对系统可用性的影响，我们从降低单次 GC 时间和降低 GC 频率两个角度出发进行了调优，还测试过 G1 垃圾回收器，但这三项措施均未能降低 GC 对服务可用性的影响。

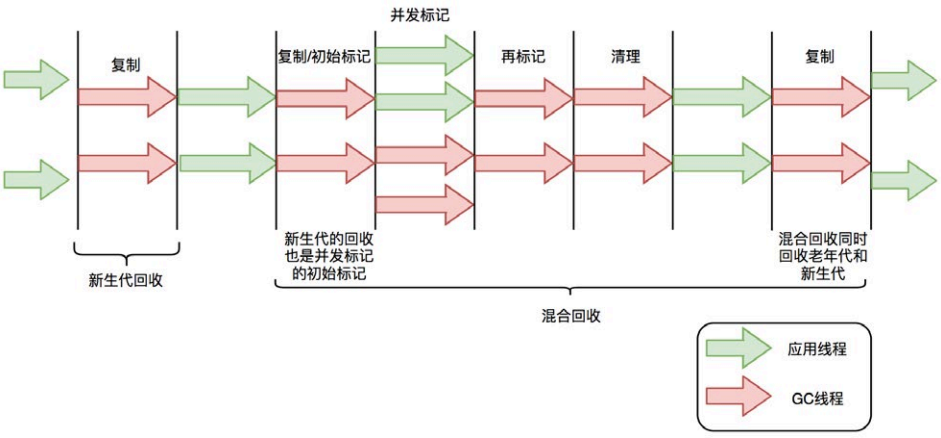
CMS 与 G1 停顿时间瓶颈

在介绍 ZGC 之前，首先回顾一下 CMS 和 G1 的 GC 过程以及停顿时间的瓶颈。CMS 新生代的 Young GC、G1 和 ZGC 都基于标记 – 复制算法，但算法具体实现的不同就导致了巨大的性能差异。

标记 – 复制算法应用在 CMS 新生代（ParNew 是 CMS 默认的新生代垃圾回收器）和 G1 垃圾回收器中。标记 – 复制算法可以分为三个阶段：

- 标记阶段，即从 GC Roots 集合开始，标记活跃对象；
- 转移阶段，即把活跃对象复制到新的内存地址上；
- 重定位阶段，因为转移导致对象的地址发生了变化，在重定位阶段，所有指向对象旧地址的指针都要调整到对象新的地址上。

下面以 G1 为例，通过 G1 中标记 – 复制算法过程（G1 的 Young GC 和 Mixed GC 均采用该算法），分析 G1 停顿耗时的主要瓶颈。G1 垃圾回收周期如下图所示：



G1 的混合回收过程可以分为标记阶段、清理阶段和复制阶段。

标记阶段停顿分析

- 初始标记阶段：初始标记阶段是指从 GC Roots 出发标记全部直接子节点的过程，该阶段是 STW 的。由于 GC Roots 数量不多，通常该阶段耗时非常短。
- 并发标记阶段：并发标记阶段是指从 GC Roots 开始对堆中对象进行可达性分析，找出存活对象。该阶段是并发的，即应用线程和 GC 线程可以同时活动。并发标记耗时相对长很多，但因为不是 STW，所以我们不太关心该阶段耗时的长短。
- 再标记阶段：重新标记那些在并发标记阶段发生变化的对象。该阶段是 STW 的。

清理阶段停顿分析

- 清理阶段清点出有存活对象的分区和没有存活对象的分区，该阶段不会清理垃圾对象，也不会执行存活对象的复制。该阶段是 STW 的。

复制阶段停顿分析

- 复制算法中的转移阶段需要分配新内存和复制对象的成员变量。转移阶段是 STW 的，其中内存分配通常耗时非常短，但对象成员变量的复制耗时有可能较长，这是因为复制耗时与存活对象数量与对象复杂度成正比。对象越复杂，复制耗时越长。

四个 STW 过程中，初始标记因为只标记 GC Roots，耗时较短。再标记因为对象数少，耗时也较短。清理阶段因为内存分区数量少，耗时也较短。转移阶段要处理所有存活的对象，耗时会较长。因此，G1 停顿时间的瓶颈主要是标记 – 复制中的转移阶段 STW。为什么转移阶段不能和标记阶段一样并发执行呢？主要是 G1 未能解决转移过程中准确定位对象地址的问题。

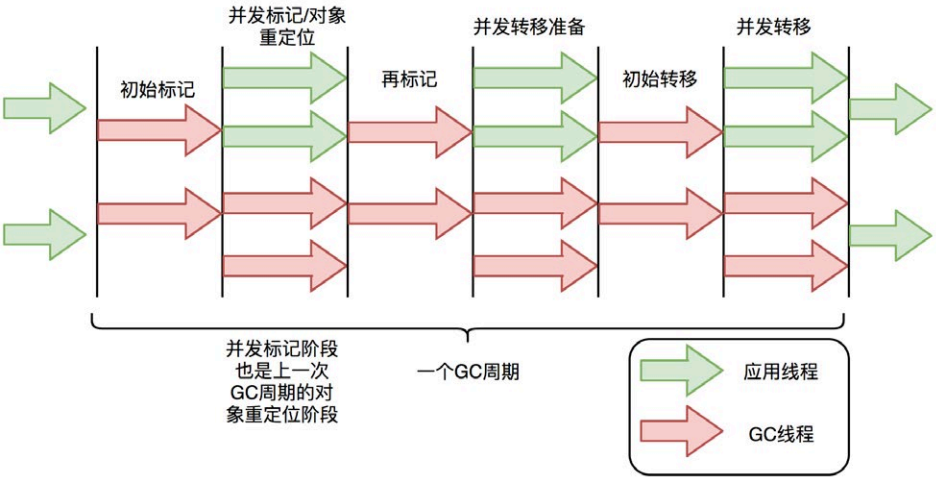
G1 的 Young GC 和 CMS 的 Young GC，其标记 – 复制全过程 STW，这里不再详细阐述。

ZGC 原理

全并发的 ZGC

与 CMS 中的 ParNew 和 G1 类似，ZGC 也采用标记－复制算法，不过 ZGC 对该算法做了重大改进：ZGC 在标记、转移和重定位阶段几乎都是并发的，这是 ZGC 实现停顿时间小于 10ms 目标的最关键原因。

ZGC 垃圾回收周期如下图所示：



ZGC 只有三个 STW 阶段：**初始标记**，**再标记**，**初始转移**。其中，初始标记和初始转移分别都只需要扫描所有 GC Roots，其处理时间和 GC Roots 的数量成正比，一般情况耗时非常短；再标记阶段 STW 时间很短，最多 1ms，超过 1ms 则再次进入并发标记阶段。即，ZGC 几乎所有暂停都只依赖于 GC Roots 集合大小，停顿时间不会随着堆的大小或者活跃对象的大小而增加。与 ZGC 对比，G1 的转移阶段完全 STW 的，且停顿时间随存活对象的大小增加而增加。

ZGC 关键技术

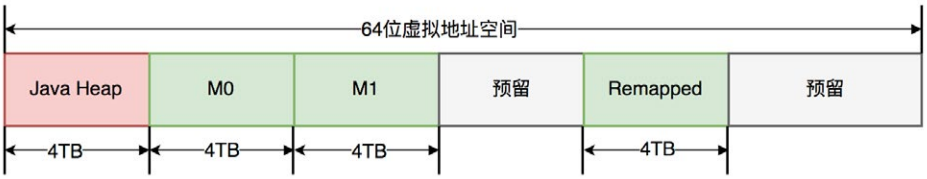
ZGC 通过着色指针和读屏障技术，解决了转移过程中准确访问对象的问题，实现了并发转移。大致原理描述如下：并发转移中“并发”意味着 GC 线程在转移对象的过

程中，应用线程也在不停地访问对象。假设对象发生转移，但对象地址未及时更新，那么应用线程可能访问到旧地址，从而造成错误。而在 ZGC 中，应用线程访问对象将触发“读屏障”，如果发现对象被移动了，那么“读屏障”会把读出来的指针更新到对象的新地址上，这样应用线程始终访问的都是对象的新地址。那么，JVM 是如何判断对象被移动过呢？就是利用对象引用的地址，即着色指针。下面介绍着色指针和读屏障技术细节。

着色指针

着色指针是一种将信息存储在指针中的技术。

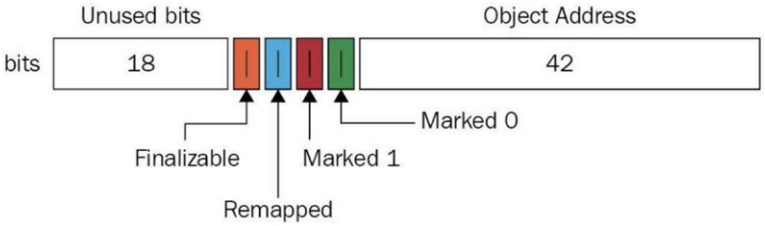
ZGC 仅支持 64 位系统，它把 64 位虚拟地址空间划分为多个子空间，如下图所示：



其中，[0~4TB) 对应 Java 堆，[4TB ~ 8TB) 称为 M0 地址空间，[8TB ~ 12TB) 称为 M1 地址空间，[12TB ~ 16TB) 预留未使用，[16TB ~ 20TB) 称为 Remapped 空间。

当应用程序创建对象时，首先在堆空间申请一个虚拟地址，但该虚拟地址并不会映射到真正的物理地址。ZGC 同时会为该对象在 M0、M1 和 Remapped 地址空间分别申请一个虚拟地址，且这三个虚拟地址对应同一个物理地址，但这三个空间在同一时间有且只有一个空间有效。ZGC 之所以设置三个虚拟地址空间，是因为它使用“空间换时间”思想，去降低 GC 停顿时间。“空间换时间”中的空间是虚拟空间，而不是真正的物理空间。后续章节将详细介绍这三个空间的切换过程。

与上述地址空间划分相对应，ZGC 实际仅使用 64 位地址空间的第 0~41 位，而第 42~45 位存储元数据，第 47~63 位固定为 0。



ZGC 将对象存活信息存储在 42~45 位中，这与传统的垃圾回收并将对象存活信息放在对象头中完全不同。

读屏障

读屏障是 JVM 向应用代码插入一小段代码的技术。当应用线程从堆中读取对象引用时，就会执行这段代码。需要注意的是，仅“从堆中读取对象引用”才会触发这段代码。

读屏障示例：

```
Object o = obj.FieldA   // 从堆中读取引用，需要加入屏障
<Load barrier>
Object p = o  // 无需加入屏障，因为不是从堆中读取引用
o.dosomething() // 无需加入屏障，因为不是从堆中读取引用
int i =  obj.FieldB  // 无需加入屏障，因为不是对象引用
```

ZGC 中读屏障的代码作用：在对象标记和转移过程中，用于确定对象的引用地址是否满足条件，并作出相应动作。

ZGC 并发处理演示

接下来详细介绍 ZGC 一次垃圾回收周期中地址视图的切换过程：

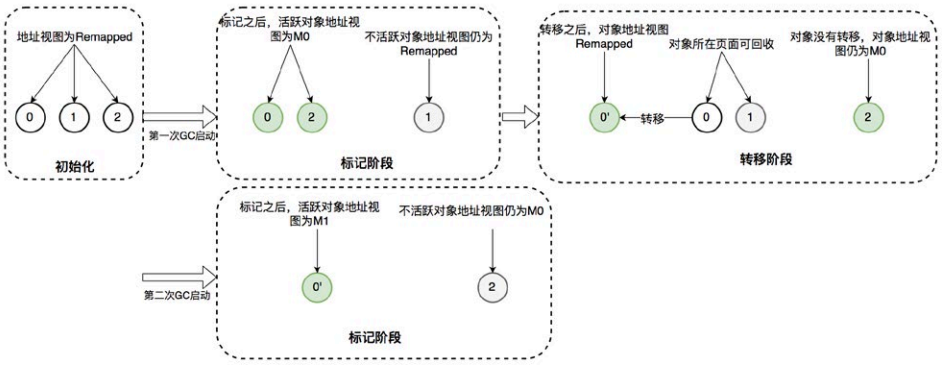
- **初始化**：ZGC 初始化之后，整个内存空间的地址视图被设置为 Remapped。程序正常运行，在内存中分配对象，满足一定条件后垃圾回收启动，此时进入标记阶段。
- **并发标记阶段**：第一次进入标记阶段时视图为 M0，如果对象被 GC 标记线程或者应用线程访问过，那么就将对象的地址视图从 Remapped 调整

为 M0。所以，在标记阶段结束之后，对象的地址要么是 M0 视图，要么是 Remapped。如果对象的地址是 M0 视图，那么说明对象是活跃的；如果对象的地址是 Remapped 视图，说明对象是不活跃的。

- **并发转移阶段**：标记结束后就进入转移阶段，此时地址视图再次被设置为 Remapped。如果对象被 GC 转移线程或者应用线程访问过，那么就将对象的地址视图从 M0 调整为 Remapped。

其实，在标记阶段存在两个地址视图 M0 和 M1，上面的过程显示只用了一个地址视图。之所以设计成两个，是为了区别前一次标记和当前标记。也即，第二次进入并发标记阶段后，地址视图调整为 M1，而非 M0。

着色指针和读屏障技术不仅应用在并发转移阶段，还应用在并发标记阶段：将对象设置为已标记，传统的垃圾回收器需要进行一次内存访问，并将对象存活信息放在对象头中；而在 ZGC 中，只需要设置指针地址的第 42~45 位即可，并且因为是寄存器访问，所以速度比访问内存更快。



ZGC 调优实践

ZGC 不是“银弹”，需要根据服务的具体特点进行调优。网络上能搜索到实战经验较少，调优理论需自行摸索，我们在此阶段也耗费了不少时间，最终才达到理想的性能。本文的一个目的是列举一些使用 ZGC 时常见的问题，帮助大家使用 ZGC 提高服务可用性。

调优基础知识

理解 ZGC 重要配置参数

以我们服务在生产环境中 ZGC 参数配置为例，说明各个参数的作用：

重要参数配置样例：

```
-Xms10G -Xmx10G
-XX:ReservedCodeCacheSize=256m -XX:InitialCodeCacheSize=256m
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC
-XX:ConcGCThreads=2 -XX:ParallelGCThreads=6
-XX:ZCollectionInterval=120 -XX:ZAllocationSpikeTolerance=5
-XX:+UnlockDiagnosticVMOptions -XX:-ZProactive
-Xlog:safepoint,classhisto*=trace,age*,gc*=info:file=/opt/logs/logs/gc-
%t.log:time,tid,tags:filecount=5,filesize=50m
```

-Xms -Xmx：堆的最大内存和最小内存，这里都设置为 10G，程序的堆内存将保持 10G 不变。**-XX:ReservedCodeCacheSize -XX:InitialCodeCacheSize**：设置 CodeCache 的大小，JIT 编译的代码都放在 CodeCache 中，一般服务 64m 或 128m 就已经足够。我们的服务因为有一定特殊性，所以设置的较大，后面会详细介绍。**-XX:+UnlockExperimentalVMOptions -XX:+UseZGC**：启用 ZGC 的配置。**-XX:ConcGCThreads**：并发回收垃圾的线程。默认是总核数的 12.5%，8 核 CPU 默认是 1。调大后 GC 变快，但会占用程序运行时的 CPU 资源，吞吐会受到影响。**-XX:ParallelGCThreads**：STW 阶段使用线程数，默认是总核数的 60%。**-XX:ZCollectionInterval**：ZGC 发生的最小时间间隔，单位秒。**-XX:ZAllocationSpikeTolerance**：ZGC 触发自适应算法的修正系数，默认 2，数值越大，越早的触发 ZGC。**-XX:+UnlockDiagnosticVMOptions -XX:-ZProactive**：是否启用主动回收，默认开启，这里的配置表示关闭。**-Xlog**：设置 GC 日志中的内容、格式、位置以及每个日志的大小。

理解 ZGC 触发时机

相比于 CMS 和 G1 的 GC 触发机制，ZGC 的 GC 触发机制有很大不同。ZGC 的核心特点是并发，GC 过程中一直有新的对象产生。如何保证在 GC 完成之前，新

产生的对象不会将堆占满，是 ZGC 参数调优的第一大目标。因为在 ZGC 中，当垃圾来不及回收将堆占满时，会导致正在运行的线程停顿，持续时间可能长达秒级之久。

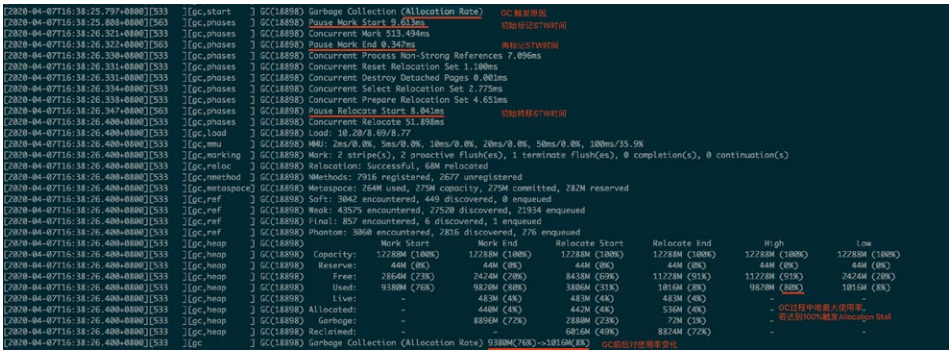
ZGC 有多种 GC 触发机制，总结如下：

- 阻塞内存分配请求触发：当垃圾来不及回收，垃圾将堆占满时，会导致部分线程阻塞。我们应当避免出现这种触发方式。日志中关键字是“Allocation Stall”。
- 基于分配速率的自适应算法：最主要的 GC 触发方式，其算法原理可简单描述为”ZGC 根据近期的对象分配速率以及 GC 时间，计算出当内存占用达到什么阈值时触发下一次 GC”。自适应算法的详细理论可参考彭成寒《新一代垃圾回收器 ZGC 设计与实现》一书中的内容。通过 ZAllocationSpikeTolerance 参数控制阈值大小，该参数默认 2，数值越大，越早的触发 GC。我们通过调整此参数解决了一些问题。日志中关键字是“Allocation Rate”。
- 基于固定时间间隔：通过 ZCollectionInterval 控制，适合应对突增流量场景。流量平稳变化时，自适应算法可能在堆使用率达到 95% 以上才触发 GC。流量突增时，自适应算法触发的时机可能会过晚，导致部分线程阻塞。我们通过调整此参数解决流量突增场景的问题，比如定时活动、秒杀等场景。日志中关键字是“Timer”。
- 主动触发规则：类似于固定间隔规则，但时间间隔不固定，是 ZGC 自行算出来的时机，我们的服务因为已经加了基于固定时间间隔的触发机制，所以通过 -ZProactive 参数将该功能关闭，以免 GC 频繁，影响服务可用性。 日志中关键字是“Proactive”。
- 预热规则：服务刚启动时出现，一般不需要关注。日志中关键字是“War-mup”。
- 外部触发：代码中显式调用 System.gc() 触发。 日志中关键字是“System.gc()”。

- 元数据分配触发：元数据区不足时导致，一般不需要关注。 日志中关键字是“Metadata GC Threshold”。

理解 ZGC 日志

一次完整的 GC 过程，需要注意的点已在图中标出。

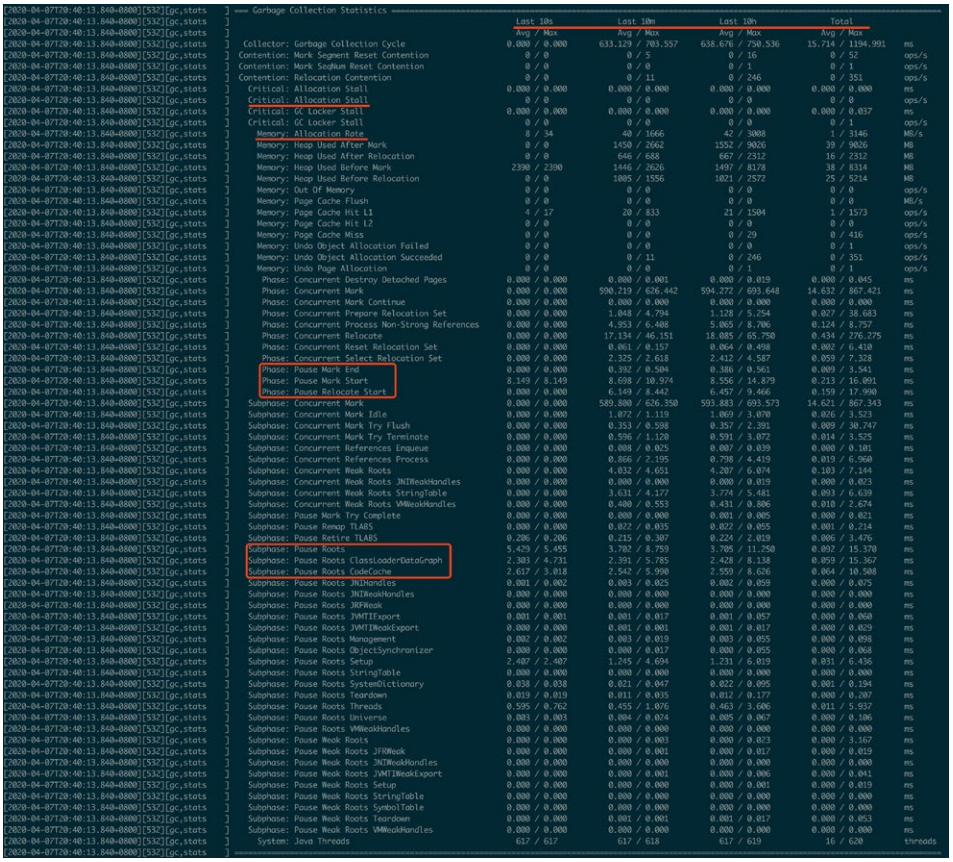


注意：该日志过滤了进入安全点的信息。正常情况，在一次 GC 过程中还穿插着进入安全点的操作。

GC 日志中每一行都注明了 GC 过程中的信息，关键信息如下：

- **Start**：开始 GC，并标明的 GC 触发的原因。上图中触发原因是自适应算法。
- **Phase-Pause Mark Start**：初始标记，会 STW。
- **Phase-Pause Mark End**：再次标记，会 STW。
- **Phase-Pause Relocate Start**：初始转移，会 STW。
- **Heap 信息**：记录了 GC 过程中 Mark、Relocate 前后的堆大小变化状况。High 和 Low 记录了其中的最大值和最小值，我们一般关注 High 中 Used 的值，如果达到 100%，在 GC 过程中一定存在内存分配不足的情况，需要调整 GC 的触发时机，更早或者更快地进行 GC。
- **GC 信息统计**：可以定时的打印垃圾收集信息，观察 10 秒内、10 分钟内、10 个小时内，从启动到现在的所有统计信息。利用这些统计信息，可以排查定位一些异常点。

日志中内容较多，关键点已用红线标出，含义较好理解，更详细的解释大家可以自行在网上查阅资料。



```
[2020-04-07T20:40:13.840+0800][532][gc,stats     ] === Garbage Collection Statistics =======================================================================================================================
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]                                                                  Last 10s              Last 10m              Last 10h                Total
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]                                                                  Avg / Max             Avg / Max             Avg / Max               Avg / Max
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Collector: Garbage Collection Cycle                          0.000 / 0.000         633.129 / 703.557     638.676 / 750.536       15.714 / 1194.991     ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ] Contention: Mark Segment Reset Contention                        0 / 0                 0 / 5                 0 / 16                  0 / 52                ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ] Contention: Mark SeqNum Reset Contention                         0 / 0                 0 / 0                 0 / 1                   0 / 1                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ] Contention: Relocation Contention                                0 / 0                 0 / 11                0 / 246                 0 / 351               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Critical: Allocation Stall                                   0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Critical: Allocation Stall                                   0 / 0                 0 / 0                 0 / 0                   0 / 0                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Critical: GC Locker Stall                                    0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.037         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Critical: GC Locker Stall                                    0 / 0                 0 / 0                 0 / 0                   0 / 1                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Allocation Rate                                    8 / 34                40 / 1666             42 / 3008               1 / 3146              MB/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Heap Used After Mark                               0 / 0                 1450 / 2662           1552 / 9026             39 / 9026             MB
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Heap Used After Relocation                         0 / 0                 646 / 688             667 / 2312              16 / 2312             MB
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Heap Used Before Mark                              2390 / 2390           1446 / 2626           1497 / 8178             38 / 8314             MB
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Heap Used Before Relocation                        0 / 0                 1005 / 1556           1021 / 2572             25 / 5214             MB
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Out Of Memory                                      0 / 0                 0 / 0                 0 / 0                   0 / 0                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Page Cache Flush                                   0 / 0                 0 / 0                 0 / 0                   0 / 0                 MB/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Page Cache Hit L1                                  4 / 17                20 / 833              21 / 1504               1 / 1573              ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Page Cache Hit L2                                  0 / 0                 0 / 0                 0 / 0                   0 / 0                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Page Cache Miss                                    0 / 0                 0 / 0                 0 / 29                  0 / 416               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Undo Object Allocation Failed                      0 / 0                 0 / 0                 0 / 0                   0 / 1                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Undo Object Allocation Succeeded                   0 / 0                 0 / 11                0 / 246                 0 / 351               ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     Memory: Undo Page Allocation                               0 / 0                 0 / 0                 0 / 1                   0 / 1                 ops/s
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Destroy Detached Pages                  0.000 / 0.000         0.000 / 0.001         0.000 / 0.019           0.000 / 0.045         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Mark                                    0.000 / 0.000         590.219 / 626.442     594.272 / 693.648       14.632 / 867.421      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Mark Continue                           0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Prepare Relocation Set                  0.000 / 0.000         1.048 / 4.794         1.128 / 5.254           0.027 / 38.683        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Process Non-Strong References           0.000 / 0.000         4.953 / 6.408         5.065 / 8.706           0.124 / 8.757         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Relocate                                0.000 / 0.000         17.134 / 46.151       18.885 / 65.750         0.434 / 276.275       ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Reset Relocation Set                    0.000 / 0.000         0.061 / 0.157         0.064 / 0.498           0.002 / 6.410         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Concurrent Select Relocation Set                   0.000 / 0.000         2.325 / 2.618         2.412 / 4.587           0.059 / 7.328         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Pause Mark End                                     0.000 / 0.000         0.392 / 0.504         0.386 / 0.561           0.009 / 3.541         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Pause Mark Start                                   8.149 / 8.149         8.698 / 10.974        8.556 / 14.879          0.213 / 16.091        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]      Phase: Pause Relocate Start                               0.000 / 0.000         6.149 / 8.442         6.457 / 9.466           0.159 / 17.990        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Mark                                    0.000 / 0.000         589.800 / 626.350     593.883 / 693.573       14.621 / 867.343      ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Mark Idle                               0.000 / 0.000         1.072 / 1.119         1.069 / 3.070           0.026 / 3.523         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Mark Try Flush                          0.000 / 0.000         0.353 / 0.598         0.357 / 2.391           0.009 / 30.747        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Mark Try Terminate                      0.000 / 0.000         0.596 / 1.120         0.591 / 3.072           0.014 / 3.525         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent References Enqueue                      0.000 / 0.000         0.008 / 0.025         0.007 / 0.039           0.000 / 0.101         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent References Process                      0.000 / 0.000         0.866 / 2.105         0.798 / 4.419           0.019 / 6.960         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Weak Roots                              0.000 / 0.000         4.032 / 4.651         4.207 / 6.074           0.103 / 7.144         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Weak Roots JNIWeakHandles               0.000 / 0.000         0.000 / 0.000         0.000 / 0.019           0.000 / 0.023         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Weak Roots StringTable                  0.000 / 0.000         3.631 / 4.177         3.774 / 5.481           0.093 / 6.639         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Concurrent Weak Roots VMWeakHandles                0.000 / 0.000         0.400 / 0.553         0.431 / 0.806           0.010 / 2.674         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Mark Try Complete                            0.000 / 0.000         0.000 / 0.000         0.001 / 0.005           0.000 / 0.021         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Remap TLABS                                  0.000 / 0.000         0.022 / 0.035         0.022 / 0.055           0.001 / 0.214         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Retire TLABS                                 0.206 / 0.206         0.215 / 0.307         0.224 / 2.019           0.006 / 3.476         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots                                        5.429 / 5.455         3.702 / 8.759         3.705 / 11.258          0.092 / 15.370        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots ClassLoaderDataGraph                   2.303 / 4.731         2.391 / 5.785         2.428 / 8.138           0.059 / 15.367        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots CodeCache                              2.617 / 3.818         2.542 / 5.990         2.559 / 8.626           0.064 / 10.508        ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots JNIHandles                             0.001 / 0.002         0.003 / 0.025         0.002 / 0.059           0.000 / 0.075         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots JNIWeakHandles                         0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots JRFWeak                                0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots JVMTIExport                            0.001 / 0.001         0.001 / 0.017         0.001 / 0.057           0.000 / 0.060         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots JVMTIWeakExport                        0.000 / 0.000         0.001 / 0.001         0.001 / 0.017           0.000 / 0.029         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots Management                             0.002 / 0.002         0.003 / 0.019         0.003 / 0.055           0.000 / 0.098         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots ObjectSynchronizer                     0.000 / 0.000         0.000 / 0.017         0.000 / 0.055           0.000 / 0.068         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots Setup                                  2.407 / 2.407         1.245 / 4.694         1.231 / 6.019           0.031 / 6.436         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots StringTable                            0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots SystemDictionary                       0.038 / 0.038         0.021 / 0.047         0.022 / 0.095           0.001 / 0.194         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots Teardown                               0.019 / 0.019         0.011 / 0.035         0.012 / 0.177           0.000 / 0.207         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots Threads                                0.595 / 0.762         0.455 / 1.076         0.463 / 3.606           0.011 / 5.937         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots Universe                               0.003 / 0.003         0.004 / 0.024         0.005 / 0.067           0.000 / 0.106         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Roots VMWeakHandles                          0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots                                   0.000 / 0.000         0.000 / 0.003         0.000 / 0.023           0.000 / 3.167         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots JFRWeak                           0.000 / 0.000         0.000 / 0.001         0.000 / 0.017           0.000 / 0.019         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots JNIWeakHandles                    0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots JVMTIWeakExport                   0.000 / 0.000         0.000 / 0.001         0.000 / 0.006           0.000 / 0.041         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots Setup                             0.000 / 0.000         0.000 / 0.000         0.000 / 0.001           0.000 / 0.019         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots StringTable                       0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots SymbolTable                       0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots Teardown                          0.000 / 0.000         0.001 / 0.001         0.001 / 0.017           0.000 / 0.053         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]   Subphase: Pause Weak Roots VMWeakHandles                     0.000 / 0.000         0.000 / 0.000         0.000 / 0.000           0.000 / 0.000         ms
[2020-04-07T20:40:13.840+0800][532][gc,stats     ]     System: Java Threads                                       617 / 617             617 / 618             617 / 619               16 / 620              threads
[2020-04-07T20:40:13.840+0800][532][gc,stats     ] =======================================================================================================================
```

理解 ZGC 停顿原因

我们在实战过程中共发现了 6 种使程序停顿的场景，分别如下：

- **GC 时，初始标记**：日志中 Pause Mark Start。
- **GC 时，再标记**：日志中 Pause Mark End。
- **GC 时，初始转移**：日志中 Pause Relocate Start。
- **内存分配阻塞**：当内存不足时线程会阻塞等待 GC 完成，关键字是”Allocation Stall”。

```
Allocation Stall (zeus-fiber-worker-common-pool-3) 364.489ms
Allocation Stall (zeus-fiber-worker-common-pool-4) 364.293ms
Allocation Stall (zeus-fiber-worker-common-pool-14) 363.021ms
Allocation Stall (MtthriftClientNioGroup-5-thread-29) 363.921ms
Allocation Stall (MtthriftClientNioGroup-5-thread-5) 362.064ms
Allocation Stall (MtthriftClientNioGroup-5-thread-17) 359.804ms
```

- **安全点**：所有线程进入到安全点后才能进行 GC，ZGC 定期进入安全点判断是否需要 GC。先进入安全点的线程需要等待后进入安全点的线程直到所有线程挂起。
- **dump 线程、内存**：比如 jstack、jmap 命令。

```
Entering safepoint region: ThreadDump
Leaving safepoint region
Total time for which application threads were stopped: 1.1186756 seconds, Stopping threads took: 0.0004182 seconds
```

```
Entering safepoint region: HeapDumper
GC(34778) Garbage Collection (Timer)
Leaving safepoint region
Total time for which application threads were stopped: 8.2160410 seconds, Stopping threads took: 0.0003233 seconds
```

调优案例

我们维护的服务名叫 Zeus，它是美团的规则平台，常用于风控场景中的规则管理。规则运行是基于开源的表达式执行引擎 Aviator。Aviator 内部将每一条表达式转化成 Java 的一个类，通过调用该类的接口实现表达式逻辑。

Zeus 服务内的规则数量超过万条，且每台机器每天的请求量几百万。这些客观条件导致 Aviator 生成的类和方法会产生很多的 ClassLoader 和 CodeCache，这些在使用 ZGC 时都成为过 GC 的性能瓶颈。接下来介绍两类调优案例。

内存分配阻塞，系统停顿可达到秒级

案例一：秒杀活动中流量突增，出现性能毛刺

日志信息：对比出现性能毛刺时间点的 GC 日志和业务日志，发现 JVM 停顿了较长时间，且停顿时 GC 日志中有大量的“Allocation Stall”日志。

分析：这种案例多出现在“自适应算法”为主要 GC 触发机制的场景中。ZGC 是一款并发的垃圾回收器，GC 线程和应用线程同时活动，在 GC 过程中，还会产生新的

对象。GC 完成之前，新产生的对象将堆占满，那么应用线程可能因为申请内存失败而导致线程阻塞。当秒杀活动开始，大量请求打入系统，但自适应算法计算的 GC 触发间隔较长，导致 GC 触发不及时，引起了内存分配阻塞，导致停顿。

解决方法：

（1）开启”基于固定时间间隔“的 GC 触发机制：-XX:ZCollectionInterval。比如调整为 5 秒，甚至更短。

（2）增大修正系数 -XX:ZAllocationSpikeTolerance，更早触发 GC。ZGC 采用正态分布模型预测内存分配速率，模型修正系数 ZAllocationSpikeTolerance 默认值为 2，值越大，越早的触发 GC，Zeus 中所有集群设置的是 5。

案例二：压测时，流量逐渐增大到一定程度后，出现性能毛刺

日志信息：平均 1 秒 GC 一次，两次 GC 之间几乎没有间隔。

分析：GC 触发及时，但内存标记和回收速度过慢，引起内存分配阻塞，导致停顿。

解决方法：增大 -XX:ConcGCThreads，加快并发标记和回收速度。ConcGC-Threads 默认值是核数的 1/8，8 核机器，默认值是 1。该参数影响系统吞吐，如果 GC 间隔时间大于 GC 周期，不建议调整该参数。

GC Roots 数量大，单次 GC 停顿时间长

案例三：单次 GC 停顿时间 30ms，与预期停顿 10ms 左右有较大差距

日志信息：观察 ZGC 日志信息统计，“Pause Roots ClassLoaderDataGraph”一项耗时较长。

分析：dump 内存文件，发现系统中有上万个 ClassLoader 实例。我们知道 Class-Loader 属于 GC Roots 一部分，且 ZGC 停顿时间与 GC Roots 成正比，GC Roots 数量越大，停顿时间越久。再进一步分析，ClassLoader 的类名表明，这些 ClassLoader 均由 Aviator 组件生成。分析 Aviator 源码，发现 Aviator 对每一个表

达式新生成类时，会创建一个 ClassLoader，这导致了 ClassLoader 数量巨大的问题。在更高 Aviator 版本中，该问题已经被修复，即仅创建一个 ClassLoader 为所有表达式生成类。

解决方法：升级 Aviator 组件版本，避免生成多余的 ClassLoader。

案例四：服务启动后，运行时间越长，单次 GC 时间越长，重启后恢复

日志信息：观察 ZGC 日志信息统计，“Pause Roots CodeCache”的耗时会随着服务运行时间逐渐增长。

分析：CodeCache 空间用于存放 Java 热点代码的 JIT 编译结果，而 CodeCache 也属于 GC Roots 一部分。通过添加 -XX:+PrintCodeCacheOnCompilation 参数，打印 CodeCache 中的被优化的方法，发现大量的 Aviator 表达式代码。定位到根本原因，每个表达式都是一个类中一个方法。随着运行时间越长，执行次数增加，这些方法会被 JIT 优化编译进入到 Code Cache 中，导致 CodeCache 越来越大。

解决方法：JIT 有一些参数配置可以调整 JIT 编译的条件，但对于我们的问题都不太适用。我们最终通过业务优化解决，删除不需要执行的 Aviator 表达式，从而避免了大量 Aviator 方法进入 CodeCache 中。

值得一提的是，我们并不是在所有这些问题都解决后才全量部署所有集群。即使开始有各种各样的毛刺，但计算后发现，有各种问题的 ZGC 也比之前的 CMS 对服务可用性影响小。所以从开始准备使用 ZGC 到全量部署，大概用了 2 周的时间。在之后的 3 个月时间里，我们边做业务需求，边跟进这些问题，最终逐个解决了上述问题，从而使 ZGC 在各个集群上达到了一个更好表现。

升级 ZGC 效果

延迟降低

TP(Top Percentile) 是一项衡量系统延迟的指标：TP999 表示 99.9% 请求都能被响

应的最小耗时；TP99 表示 99% 请求都能被响应的最小耗时。

在 Zeus 服务不同集群中，ZGC 在低延迟（TP999 < 200ms）场景中收益较大：

- **TP999**：下降 12~142ms，下降幅度 18%~74%。
- **TP99**：下降 5~28ms，下降幅度 10%~47%。

超低延迟（TP999 < 20ms）和高延迟（TP999 > 200ms）服务收益不大，原因是这些服务的响应时间瓶颈不是 GC，而是外部依赖的性能。

吞吐下降

对吞吐量优先的场景，ZGC 可能并不适合。例如，Zeus 某离线集群原先使用 CMS，升级 ZGC 后，系统吞吐量明显降低。究其原因有二：第一，ZGC 是单代垃圾回收器，而 CMS 是分代垃圾回收器。单代垃圾回收器每次处理的对象更多，更耗费 CPU 资源；第二，ZGC 使用读屏障，读屏障操作需耗费额外的计算资源。

总结

ZGC 作为下一代垃圾回收器，性能非常优秀。ZGC 垃圾回收过程几乎全部是并发，实际 STW 停顿时间极短，不到 10ms。这得益于其采用的着色指针和读屏障技术。

Zeus 在升级 JDK 11+ZGC 中，通过将风险和问题分类，然后各个击破，最终顺利实现了升级目标，GC 停顿也几乎不再影响系统可用性。

最后推荐大家升级 ZGC，Zeus 系统因为业务特点，遇到了较多问题，而风控其他团队在升级时都非常顺利。欢迎大家加入“ZGC 使用交流”群。

参考文献

ZGC 官网
彭成寒 .《新一代垃圾回收器 ZGC 设计与实现》. 机械工业出版社 , 2019.
从实际案例聊聊 Java 应用的 GC 优化
Java Hotspot G1 GC 的一些关键技术

附录

如何使用新技术

在生产环境升级 JDK 11，使用 ZGC，大家最关心的可能不是效果怎么样，而是这个新版本用的人少，网上实践也少，靠不靠谱，稳不稳定。其次是升级成本会不会很大，万一不成功岂不是白白浪费时间。所以，在使用新技术前，首先要做的是评估收益、成本和风险。

评估收益

对于 JDK 这种世界关注的程序，大版本升级所引入的新技术一般已经在理论上经过验证。我们要做的事情就是确定当前系统的瓶颈是否是新版本 JDK 可解决的问题，切忌问题未诊断清楚就采取措施。评估完收益之后再评估成本和风险，收益过大或者过小，其他两项影响权重就会小很多。

以本文开头提到的案例为例，假设 GC 次数不变（10 次 / 分钟），且单次 GC 时间从 40ms 降低 10ms。通过计算，一分钟内有 100/60000 = 0.17% 的时间在进行 GC，且期间所有请求仅停顿 10ms，GC 期间影响的请求数和因 GC 增加的延迟都有所减少。

评估成本

这里主要指升级所需要的人力成本。此项相对比较成熟，根据新技术的使用手册判断改动点。跟做其他项目区别不大，不再具体细说。

在我们的实践中，两周时间完成线上部署，达到安全稳定运行的状态。后续持续迭代 3 个月，根据业务场景对 ZGC 进行了更契合的优化适配。

评估风险

升级 JDK 的风险可以分为三类：

- **兼容性风险**：Java 程序 JAR 包依赖很多，升级 JDK 版本后程序是否能运行

起来。例如我们的服务是从 JDK 7 升级到 JDK 11，需要解决较多 JAR 包不兼容的问题。

- **功能风险**：运行起来后，是否会有一些组件逻辑变更，影响现有功能的逻辑。
- **性能风险**：功能如果没有问题，性能是否稳定，能稳定的在线上运行。

经过分类后，每类风险的应对转化成了常见的测试问题，不再属于未知风险。风险是指不确定的事情，如果不确定的事情都能转化成可确定的事情，意味着风险已消除。

升级 JDK 11

选择 JDK 11，是因为在 JDK 11 中首次支持 ZGC，而且 JDK 11 属于长期支持（Long Term Support，LTS）版本，至少会被维护三年，普通版本（如 JDK 12、JDK 13 和 JDK 14）只有 6 个月的维护周期，不建议使用。

本地测试环境安装

从两个源 OpenJDK 和 OracleJDK 下载 JDK 11，二个版本的 JDK 主要区别是长时期的免费和付费，短期内都免费。注意 JDK 11 版本中的 ZGC 不支持 Mac OS 系统，在 Mac OS 系统上使用 JDK 11 只能用其他垃圾回收器，如 G1。

生产环境安装

升级 JDK 11 不仅仅是升级自己项目的 JDK 版本，还需要编译、发布部署、运行、监控、性能内存分析工具等项目支持。美团内部的实践：

编译打包：美团发布系统支持选择 JDK 11 进行编译打包。 **线上运行 & 全量部署**：要求线上机器已安装 JDK11，有 3 种方式：

1. 新申请默认安装 JDK 11 的虚拟机：试用 JDK 11 时可用这种方式；全量部署时，如果新申请机器数量过多，可能没有足够机器资源。
2. 通过手写脚本给存量虚拟机安装 JDK 11：不推荐，业务同学过多参与到运维当中。

3. 使用容器提供的镜像部署功能，在打包镜像时安装 JDK 11：推荐方式，不需要新申请资源。

监控指标：主要是 GC 的时间和频率，我们通过美团的 CAT 监控系统支持 ZGC 数据的收集（CAT 已开源）。**性能内存分析**：线上遇到性能问题时，还需要借助 Profiling 工具，美团的性能诊断优化平台 Scalpel 已支持 JDK 11 的性能内存分析。如果你的公司没有相关工具，推荐使用 JProfier。

解决组件兼容性

我们的项目包含二十多万行代码，需要从 JDK 7 升级到 JDK 11，依赖组件众多。虽然看起来升级会比较复杂，但实际只花了两天时间即解决了兼容性问题。具体过程如下：

1. 编译，需要修改 pom 文件中的 build 配置，根据报错作修改，主要有两类：

a. 一些类被删除：比如“sun.misc.BASE64Encoder”，找到替换类 java.util.Base64 即可。

b. 组件依赖版本不兼容 JDK 11 问题：找到对应依赖组件，搜索最新版本，一般都支持 JDK 11。

2. 编译成功后，启动运行，此时仍有可能组件依赖版本问题，按照编译时的方式处理即可。

升级所修改的依赖：

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
```

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.4</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-parent</artifactId>
    <version>6.0.16.Final</version>
</dependency>
<dependency>
    <groupId>com.sankuai.inf</groupId>
    <artifactId>patriot-sdk</artifactId>
    <version>1.2.1</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.9</version>
</dependency>
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.39.Final</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

JDK 11 已经出来两年，常见的依赖组件都有兼容性版本。但是，如果是公司内部提供的公司级组件，可能会不兼容 JDK 11，需要推动相关组件进行升级。如果对方升级较为困难，可以考虑拆分功能，将依赖这些组件的功能单独部署，继续使用低版本 JDK。随着 JDK11 的卓越性能被大家悉知，相信会有更多团队会用 JDK 11 解决 GC 问题，使用者越多，各个组件升级的动力也会越大。

验证功能正确性

通过完备的单测、集成和回归测试，保证功能正确性。

作者简介

王东，美团信息安全资深工程师。
王伟，美团信息安全技术专家。

设计模式在外卖营销业务中的实践

作者：亮亮

一、前言

随着美团外卖业务的不断迭代与发展，外卖用户数量也在高速地增长。在这个过程中，外卖营销发挥了“中流砥柱”的作用，因为用户的快速增长离不开高效的营销策略。而由于市场环境和业务环境的多变，营销策略往往是复杂多变的，营销技术团队作为营销业务的支持部门，就需要快速高效地响应营销策略变更带来的需求变动。因此，设计并实现易于扩展和维护的营销系统，是美团外卖营销技术团队不懈追求的目标和必修的基本功。

本文通过自顶向下的方式，来介绍设计模式如何帮助我们构建一套易扩展、易维护的营销系统。本文会首先介绍设计模式与领域驱动设计（Domain-Driven Design，以下简称为 DDD）之间的关系，然后再阐述外卖营销业务引入业务中用到的设计模式以及其具体实践案例。

二、设计模式与领域驱动设计

设计一个营销系统，我们通常的做法是采用自顶向下的方式来解构业务，为此我们引入了 DDD。从战略层面上讲，DDD 能够指导我们完成从问题空间到解决方案的剖析，将业务需求映射为领域上下文以及上下文间的映射关系。从战术层面上，DDD 能够细化领域上下文，并形成有效的、细化的领域模型来指导工程实践。建立领域模型的一个关键意义在于，能够确保不断扩展和变化的需求在领域模型内不断地演进和发展，而不至于出现模型的腐化和领域逻辑的外溢。关于 DDD 的实践，大家可以参考此前美团技术团队推出的《领域驱动设计在互联网业务开发中的实践》一文。

同时，我们也需要在代码工程中贯彻和实现领域模型。因为代码工程是领域模型在工程实践中的直观体现，也是领域模型在技术层面的直接表述。而设计模式，可以说是连接领域模型与代码工程的一座桥梁，它能有效地解决从领域模型到代码工程的转化。

为什么说设计模式天然具备成为领域模型到代码工程之间桥梁的作用呢？其实，2003 年出版的《领域驱动设计》一书的作者 Eric Evans 在这部开山之作中就已经给出了解释。他认为，立场不同会影响人们如何看待什么是“模式”。因此，无论是领域驱动模式还是设计模式，本质上都是“模式”，只是解决的问题不一样。站在业务建模的立场上，DDD 的模式解决的是如何进行领域建模。而站在代码实践的立场上，设计模式主要关注于代码的设计与实现。既然本质都是模式，那么它们天然就具有一定的共通之处。

所谓“模式”，就是一套反复被人使用或验证过的方法论。从抽象或者更宏观的角度上看，只要符合使用场景并且能解决实际问题，模式应该既可以应用在 DDD 中，也可以应用在设计模式中。事实上，Evans 也是这么做的。他在著作中阐述了 Strategy 和 Composite 这两个传统的 GOF 设计模式是如何来解决领域模型建设的。因此，当领域模型需要转化为代码工程时，同构的模式，天然能够将领域模型翻译成代码模型。

三、设计模式在外卖营销业务中的具体案例

3.1 为什么需要设计模式

营销业务的特点

如前文所述，营销业务与交易等其他模式相对稳定的业务的区别在于，营销需求会随着市场、用户、环境的不断变化而进行调整。也正是因此，外卖营销技术团队选择了 DDD 进行领域建模，并在适用的场景下，用设计模式在代码工程的层面上实践和反映了领域模型。以此来做到在支持业务变化的同时，让领域和代码模型健康演进，避

免模型腐化。

理解设计模式

软件设计模式（Design pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码，让代码更容易被他人理解，保证代码可靠性，程序的重用性。可以理解为："世上本来没有设计模式，用的人多了，便总结出了一套设计模式。"

设计模式原则

面向对象的设计模式有七大基本原则：

- 开闭原则（Open Closed Principle，OCP）
- 单一职责原则（Single Responsibility Principle, SRP）
- 里氏代换原则（Liskov Substitution Principle，LSP）
- 依赖倒转原则（Dependency Inversion Principle，DIP）
- 接口隔离原则（Interface Segregation Principle，ISP）
- 合成 / 聚合复用原则（Composite/Aggregate Reuse Principle，CARP）
- 最少知识原则（Least Knowledge Principle，LKP）或者迪米特法则（Law of Demeter，LOD）

简单理解就是：开闭原则是总纲，它指导我们要对扩展开放，对修改关闭；单一职责原则指导我们实现类要职责单一；里氏替换原则指导我们不要破坏继承体系；依赖倒置原则指导我们要面向接口编程；接口隔离原则指导我们在设计接口的时候要精简单一；迪米特法则指导我们要降低耦合。

设计模式就是通过这七个原则，来指导我们如何做一个好的设计。但是设计模式不是一套"奇技淫巧"，它是一套方法论，一种高内聚、低耦合的设计思想。我们可以在此基础上自由的发挥，甚至设计出自己的一套设计模式。

当然，学习设计模式或者是在工程中实践设计模式，必须深入到某一个特定的业务场

景中去，再结合对业务场景的理解和领域模型的建立，才能体会到设计模式思想的精髓。如果脱离具体的业务逻辑去学习或者使用设计模式，那是极其空洞的。接下来我们将通过外卖营销业务的实践，来探讨如何用设计模式来实现可重用、易维护的代码。

3.2 “邀请下单”业务中设计模式的实践

3.2.1 业务简介

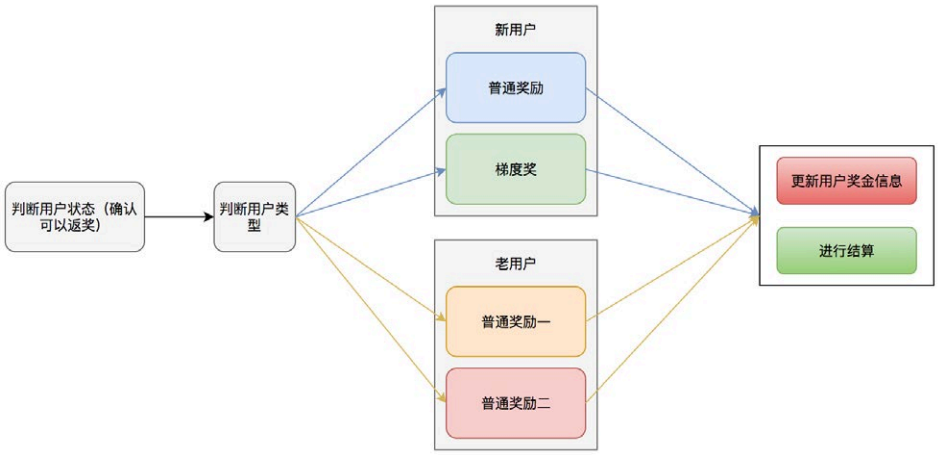
“邀请下单”是美团外卖用户邀请其他用户下单后给予奖励的平台。即用户 A 邀请用户 B，并且用户 B 在美团下单后，给予用户 A 一定的现金奖励（以下简称返奖）。同时为了协调成本与收益的关系，返奖会有多个计算策略。邀请下单后台主要涉及两个技术要点：

1. 返奖金额的计算，涉及到不同的计算规则。
2. 从邀请开始到返奖结束的整个流程。

3.2.2 返奖规则与设计模式实践

业务建模

如图是返奖规则计算的业务逻辑视图：



从这份业务逻辑图中可以看到返奖金额计算的规则。首先要根据用户状态确定用户是否满足返奖条件。如果满足返奖条件，则继续判断当前用户属于新用户还是老用户，从而给予不同的奖励方案。一共涉及以下几种不同的奖励方案：

新用户

- 普通奖励（给予固定金额的奖励）
- 梯度奖（根据用户邀请的人数给予不同的奖励金额，邀请的人越多，奖励金额越多）

老用户

- 根据老用户的用户属性来计算返奖金额。为了评估不同的邀新效果，老用户返奖会存在多种返奖机制。

计算完奖励金额以后，还需要更新用户的奖金信息，以及通知结算服务对用户的金额进行结算。这两个模块对于所有的奖励来说都是一样的。

可以看到，无论是何种用户，对于整体返奖流程是不变的，唯一变化的是返奖规则。此处，我们可参考**开闭原则**，对于返奖流程保持封闭，对于可能扩展的返奖规则进行开放。我们将返奖规则抽象为**返奖策略**，即针对不同用户类型的不同返奖方案，我们

视为不同的返奖策略，不同的返奖策略会产生不同的返奖金额结果。

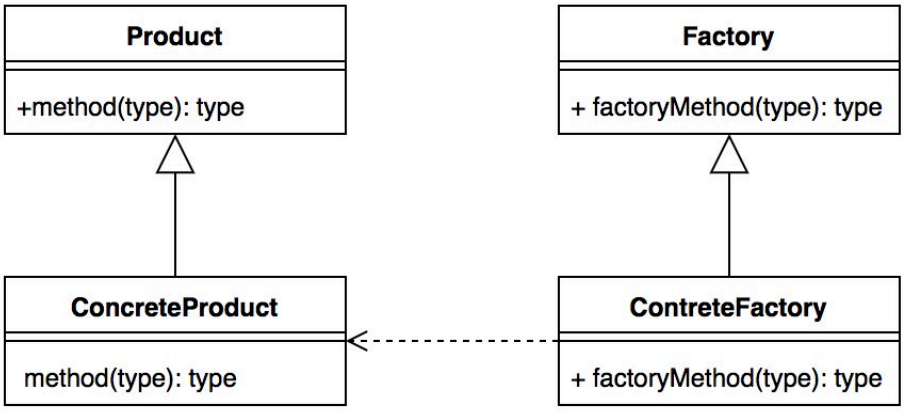
在我们的领域模型里，返奖策略是一个**值对象**，我们通过工厂的方式生产针对不同用户的奖励策略值对象。下文我们将介绍以上领域模型的工程实现，即**工厂模式**和**策略模式**的实际应用。

模式：工厂模式

工厂模式又细分为工厂方法模式和抽象工厂模式，本文主要介绍工厂方法模式。

模式定义：定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法是一个类的实例化延迟到其子类。

工厂模式通用类图如下：



我们通过一段较为通用的代码来解释如何使用工厂模式：

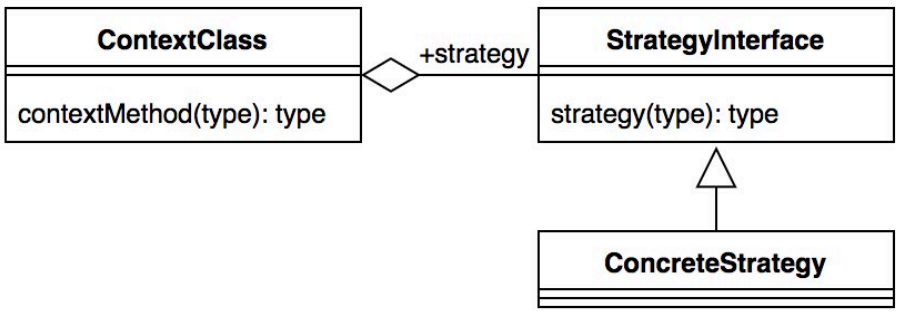
```
// 抽象的产品
public abstract class Product {
    public abstract void method();
}
// 定义一个具体的产品 （可以定义多个具体的产品）
class ProductA extends Product {
    @Override
    public void method() {}  // 具体的执行逻辑
}
// 抽象的工厂
abstract class Factory<T> {
```

```
    abstract Product createProduct(Class<T> c);
}
// 具体的工厂可以生产出相应的产品
class FactoryA extends Factory{
    @Override
    Product createProduct(Class c) {
        Product product = (Product) Class.forName(c.getName()).
newInstance();
        return product;
    }
}
```

模式：策略模式

模式定义：定义一系列算法，将每个算法都封装起来，并且它们可以互换。策略模式是一种对象行为模式。

策略模式通用类图如下：



我们通过一段比较通用的代码来解释怎么使用策略模式：

```
// 定义一个策略接口
public interface Strategy {
    void strategyImplementation();
}

// 具体的策略实现（可以定义多个具体的策略实现）
public class StrategyA implements Strategy{
    @Override
    public void strategyImplementation() {
        System.out.println("正在执行策略 A");
    }
}
```

```
// 封装策略，屏蔽高层模块对策略、算法的直接访问，屏蔽可能存在的策略变化
public class Context {
    private Strategy strategy = null;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void doStrategy() {
        strategy.strategyImplementation();
    }
}
```

工程实践

通过上文介绍的返奖业务模型，我们可以看到返奖的主流程就是选择不同的返奖策略的过程，每个返奖策略都包括返奖金额计算、更新用户奖金信息、以及结算这三个步骤。 我们可以使用工厂模式生产出不同的策略，同时使用策略模式来进行不同的策略执行。首先确定我们需要生成出 n 种不同的返奖策略，其编码如下：

```
// 抽象策略
public abstract class RewardStrategy {
    public abstract void reward(long userId);

    public void insertRewardAndSettlement(long userId, int reward) {}
; // 更新用户信息以及结算
}
// 新用户返奖具体策略 A
public class newUserRewardStrategyA extends RewardStrategy {
    @Override
    public void reward(long userId) {}  // 具体的计算逻辑，...
}

// 老用户返奖具体策略 A
public class OldUserRewardStrategyA extends RewardStrategy {
    @Override
    public void reward(long userId) {}  // 具体的计算逻辑，...
}

// 抽象工厂
public abstract class StrategyFactory<T> {
    abstract RewardStrategy createStrategy(Class<T> c);
}
```

```
// 具体工厂创建具体的策略
public class FactorRewardStrategyFactory extends StrategyFactory {
    @Override
    RewardStrategy createStrategy(Class c) {
        RewardStrategy product = null;
        try {
            product = (RewardStrategy) Class.forName(c.getName()).
newInstance();
        } catch (Exception e) {}
        return product;
    }
}
```

通过工厂模式生产出具体的策略之后，根据我们之前的介绍，很容易就可以想到使用策略模式来执行我们的策略。具体代码如下：

```
public class RewardContext {
    private RewardStrategy strategy;

    public RewardContext(RewardStrategy strategy) {
        this.strategy = strategy;
    }

    public void doStrategy(long userId) {
        int rewardMoney = strategy.reward(userId);
        insertRewardAndSettlement(long userId, int reward) {
          insertReward(userId, rewardMoney);
          settlement(userId);
       }
    }
}
```

接下来我们将工厂模式和策略模式结合在一起，就完成了整个返奖的过程：

```
public class InviteRewardImpl {
    // 返奖主流程
    public void sendReward(long userId) {
        FactorRewardStrategyFactory strategyFactory = new
FactorRewardStrategyFactory();  // 创建工厂
        Invitee invitee = getInviteeByUserId(userId);  // 根据用户 id 查询用
户信息
        if (invitee.userType == UserTypeEnum.NEW_USER) {  // 新用户返奖策略
            NewUserBasicReward newUserBasicReward = (NewUserBasicReward)
strategyFactory.createStrategy(NewUserBasicReward.class);
            RewardContext rewardContext = new
RewardContext(newUserBasicReward);
```

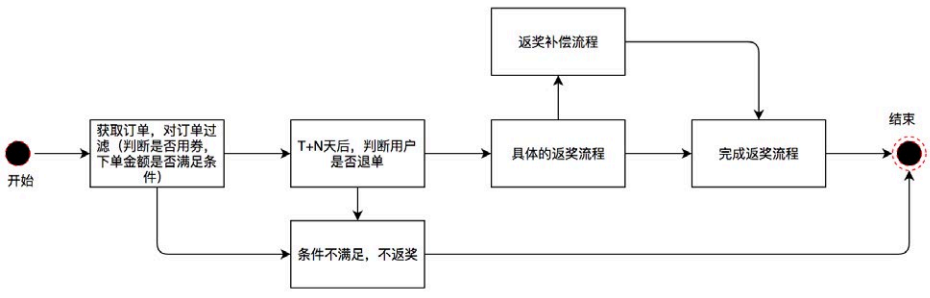
```
                rewardContext.doStrategy(userId); // 执行返奖策略
            }if(invitee.userType == UserTypeEnum.OLD_USER){}  // 老用户返奖策略，
...
        }
    }
```

工厂方法模式帮助我们直接产生一个具体的策略对象，策略模式帮助我们保证这些策略对象可以自由地切换而不需要改动其他逻辑，从而达到解耦的目的。通过这两个模式的组合，当我们系统需要增加一种返奖策略时，只需要实现 RewardStrategy 接口即可，无需考虑其他的改动。当我们需要改变策略时，只要修改策略的类名即可。不仅增强了系统的可扩展性，避免了大量的条件判断，而且从真正意义上达到了高内聚、低耦合的目的。

3.2.3 返奖流程与设计模式实践

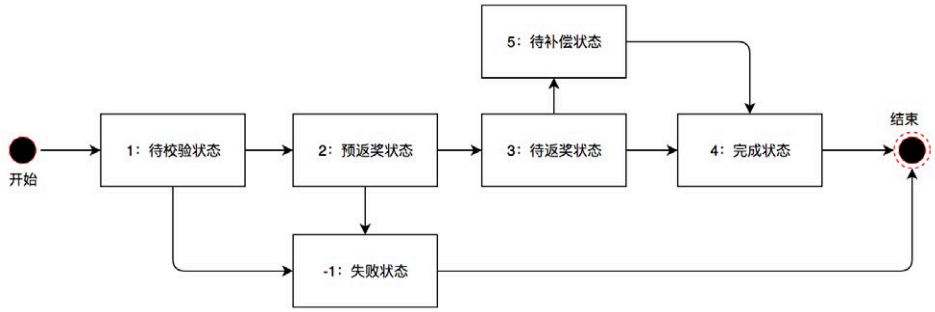
业务建模

当受邀人在接受邀请人的邀请并且下单后，返奖后台接收到受邀人的下单记录，此时邀请人也进入返奖流程。首先我们订阅用户订单消息并对订单进行返奖规则校验。例如，是否使用红包下单，是否在红包有效期内下单，订单是否满足一定的优惠金额等等条件。当满足这些条件以后，我们将订单信息放入延迟队列中进行后续处理。经过 T+N 天之后处理该延迟消息，判断用户是否对该订单进行了退款，如果未退款，对用户进行返奖。若返奖失败，后台还有返奖补偿流程，再次进行返奖。其流程如下图所示：



我们对上述业务流程进行领域建模：

1. 在接收到订单消息后，用户进入待校验状态；
2. 在校验后，若校验通过，用户进入预返奖状态，并放入延迟队列。若校验未通过，用户进入不返奖状态，结束流程；
3. T+N 天后，处理延迟消息，若用户未退款，进入待返奖状态。若用户退款，进入失败状态，结束流程；
4. 执行返奖，若返奖成功，进入完成状态，结束流程。若返奖不成功，进入待补偿状态；
5. 待补偿状态的用户会由任务定期触发补偿机制，直至返奖成功，进入完成状态，保障流程结束。

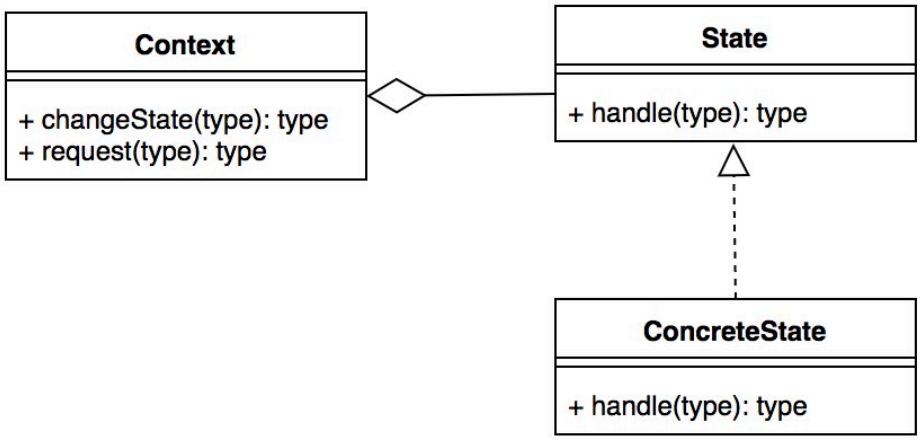


可以看到，我们通过建模将返奖流程的多个步骤映射为系统的状态。对于系统状态的表述，DDD 中常用到的概念是领域事件，另外也提及过事件溯源的实践方案。当然，在设计模式中，也有一种能够表述系统状态的代码模型，那就是状态模式。在邀请下单系统中，我们的主要流程是返奖。对于返奖，每一个状态要进行的动作和操作都是不同的。因此，使用状态模式，能够帮助我们对系统状态以及状态间的流转进行统一的管理和扩展。

模式：状态模式

模式定义：当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。

状态模式的通用类图如下图所示：



对比策略模式的类型会发现和状态模式的类图很类似，但实际上有很大的区别，具体体现在 concrete class 上。策略模式通过 Context 产生唯一一个 ConcreteStrategy 作用于代码中，而状态模式则是通过 context 组织多个 ConcreteState 形成一个状态转换图来实现业务逻辑。接下来，我们通过一段通用代码来解释怎么使用状态模式：

```
// 定义一个抽象的状态类
public abstract class State {
    Context context;
    public void setContext(Context context) {
        this.context = context;
    }
    public abstract void handle1();
    public abstract void handle2();
}
// 定义状态 A
public class ConcreteStateA extends State {
    @Override
    public void handle1() {}  // 本状态下必须要处理的事情

    @Override
    public void handle2() {
        super.context.setCurrentState(Context.contreteStateB);  // 切换到
状态 B
        super.context.handle2();  // 执行状态 B 的任务
```

```
    }
}
// 定义状态 B
public class ConcreteStateB extends State {
    @Override
    public void handle2() {}  // 本状态下必须要处理的事情，...

    @Override
    public void handle1() {
        super.context.setCurrentState(Context.contreteStateA);  // 切换到
状态 A
        super.context.handle1();  // 执行状态 A 的任务
    }
}
// 定义一个上下文管理环境
public class Context {
    public final static ConcreteStateA contreteStateA = new
ConcreteStateA();
    public final static ConcreteStateB contreteStateB = new
ConcreteStateB();

    private State CurrentState;
    public State getCurrentState() {return CurrentState;}

    public void setCurrentState(State currentState) {
        this.CurrentState = currentState;
        this.CurrentState.setContext(this);
    }

    public void handle1() {this.CurrentState.handle1();}
    public void handle2() {this.CurrentState.handle2();}
}
// 定义 client 执行
public class client {
    public static void main(String[] args) {
        Context context = new Context();
        context.setCurrentState(new ContreteStateA());
        context.handle1();
        context.handle2();
    }
}
```

工程实践

通过前文对状态模式的简介，我们可以看到当状态之间的转换在不是非常复杂的情况下，通用的状态模式存在大量的与状态无关的动作从而产生大量的无用代码。在我们

的实践中，一个状态的下游不会涉及特别多的状态装换，所以我们简化了状态模式。当前的状态只负责当前状态要处理的事情，状态的流转则由第三方类负责。其实践代码如下：

```
// 返奖状态执行的上下文
public class RewardStateContext {

    private RewardState rewardState;

    public void setRewardState(RewardState currentState) {this.
rewardState = currentState;}
    public RewardState getRewardState() {return rewardState;}
    public void echo(RewardStateContext context, Request request) {
        rewardState.doReward(context, request);
    }
}

public abstract class RewardState {
    abstract void doReward(RewardStateContext context, Request
request);
}

// 待校验状态
public class OrderCheckState extends RewardState {
    @Override
    public void doReward(RewardStateContext context, Request request)
{
        orderCheck(context, request);  // 对进来的订单进行校验，判断是否用券，
是否满足优惠条件等等
    }
}

// 待补偿状态
public class CompensateRewardState extends RewardState {
    @Override
    public void doReward(RewardStateContext context, Request request)
{
        compensateReward(context, request);  // 返奖失败，需要对用户进行返奖
补偿
    }
}

// 预返奖状态，待返奖状态，成功状态，失败状态（此处逻辑省略）
//..

public class InviteRewardServiceImpl {
    public boolean sendRewardForInvtee(long userId, long orderId) {
```

```
        Request request = new Request(userId, orderId);
        RewardStateContext rewardContext = new RewardStateContext();
        rewardContext.setRewardState(new OrderCheckState());
        rewardContext.echo(rewardContext, request);  // 开始返奖，订单校验
        // 此处的 if-else 逻辑只是为了表达状态的转换过程，并非实际的业务逻辑
        if (rewardContext.isResultFlag()) {  // 如果订单校验成功，进入预返奖
状态
            rewardContext.setRewardState(new BeforeRewardCheckState());
            rewardContext.echo(rewardContext, request);
        } else {// 如果订单校验失败，进入返奖失败流程，...
            rewardContext.setRewardState(new RewardFailedState());
            rewardContext.echo(rewardContext, request);
            return false;
        }
        if (rewardContext.isResultFlag()) {// 预返奖检查成功，进入待返奖流程，
...
            rewardContext.setRewardState(new SendRewardState());
            rewardContext.echo(rewardContext, request);
        } else {  // 如果预返奖检查失败，进入返奖失败流程，...
            rewardContext.setRewardState(new RewardFailedState());
            rewardContext.echo(rewardContext, request);
            return false;
        }
        if (rewardContext.isResultFlag()) {  // 返奖成功，进入返奖结束流程，
...
            rewardContext.setRewardState(new RewardSuccessState());
            rewardContext.echo(rewardContext, request);
        } else {  // 返奖失败，进入返奖补偿阶段，...
            rewardContext.setRewardState(new CompensateRewardState());
            rewardContext.echo(rewardContext, request);
        }
        if (rewardContext.isResultFlag()) {  // 补偿成功，进入返奖完成阶段，
...
            rewardContext.setRewardState(new RewardSuccessState());
            rewardContext.echo(rewardContext, request);
        } else {  // 补偿失败，仍然停留在当前态，直至补偿成功（或多次补偿失败后人
工介入处理）
            rewardContext.setRewardState(new CompensateRewardState());
            rewardContext.echo(rewardContext, request);
        }
        return true;
    }
}
```

状态模式的核心是封装，将状态以及状态转换逻辑封装到类的内部来实现，也很好的体现了“开闭原则”和“单一职责原则”。每一个状态都是一个子类，不管是修改还是增加状态，只需要修改或者增加一个子类即可。在我们的应用场景中，状态数量以

及状态转换远比上述例子复杂，通过“状态模式”避免了大量的 if-else 代码，让我们的逻辑变得更加清晰。同时由于状态模式的良好的封装性以及遵循的设计原则，让我们在复杂的业务场景中，能够游刃有余地管理各个状态。

3.3 点评外卖投放系统中设计模式的实践

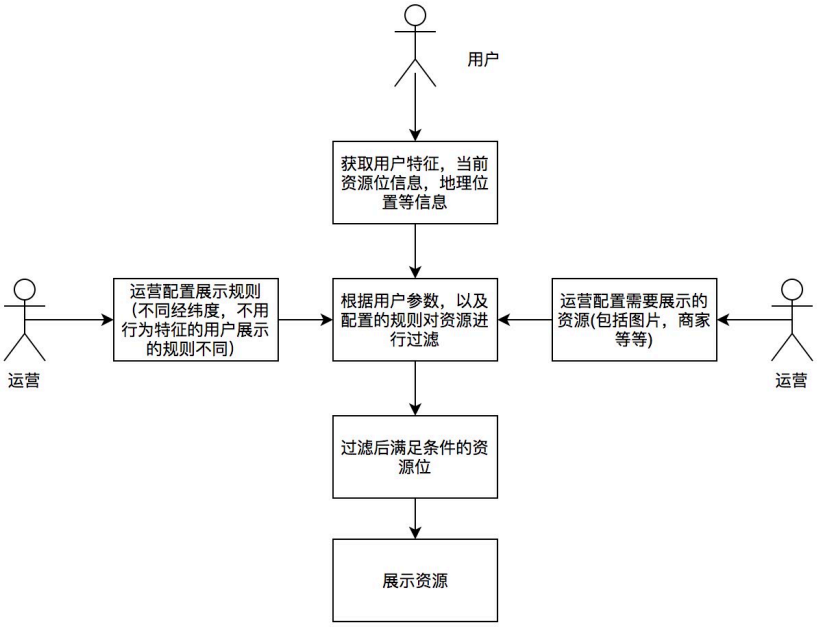
3.3.1 业务简介

继续举例，点评 App 的外卖频道中会预留多个资源位为营销使用，向用户展示一些比较精品美味的外卖食品，为了增加用户点外卖的意向。当用户点击点评首页的“美团外卖”入口时，资源位开始加载，会通过一些规则来筛选出合适的展示 Banner。

3.3.2 设计模式实践

业务建模

对于投放业务，就是要在这些资源位中展示符合当前用户的资源。其流程如下图所示：



从流程中我们可以看到，首先运营人员会配置需要展示的资源，以及对资源进行过滤的规则。我们资源的过滤规则相对灵活多变，这里体现为三点：

1. 过滤规则大部分可重用，但也会有扩展和变更。
2. 不同资源位的过滤规则和过滤顺序是不同的。
3. 同一个资源位由于业务所处的不同阶段，过滤规则可能不同。

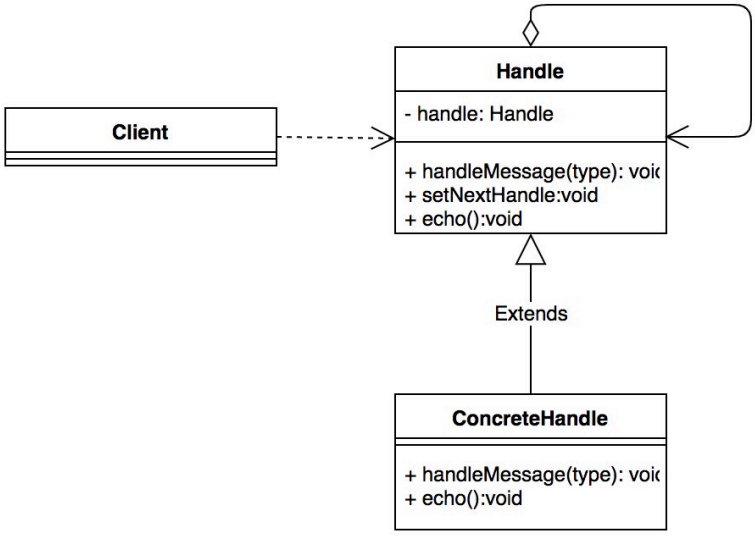
过滤规则本身是一个个的值对象，我们通过领域服务的方式，操作这些规则值对象完成资源位的过滤逻辑。下图介绍了资源位在进行用户特征相关规则过滤时的过程：

为了实现过滤规则的解耦，对单个规则值对象的修改封闭，并对规则集合组成的过滤链条开放，我们在资源位过滤的领域服务中引入了责任链模式。

模式：责任链模式

模式定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

责任链模式通用类图如下：



我们通过一段比较通用的代码来解释如何使用责任链模式：

```
// 定义一个抽象的 handle
public abstract class Handler {
    private Handler nextHandler;  // 指向下一个处理者
    private int level;  // 处理者能够处理的级别

    public Handler(int level) {
        this.level = level;
    }

    public void setNextHandler(Handler handler) {
        this.nextHandler = handler;
    }

    // 处理请求传递，注意 final，子类不可重写
    public final void handleMessage(Request request) {
```

```
        if (level == request.getRequstLevel()) {
            this.echo(request);
        } else {
            if (this.nextHandler != null) {
                this.nextHandler.handleMessage(request);
            } else {
                System.out.println(" 已经到最尽头了 ");
            }
        }
    }
    // 抽象方法，子类实现
    public abstract void echo(Request request);
}

// 定义一个具体的 handleA
public class HandleRuleA extends Handler {
    public HandleRuleA(int level) {
        super(level);
    }
    @Override
    public void echo(Request request) {
        System.out.println(" 我是处理者 1，我正在处理 A 规则 ");
    }
}

// 定义一个具体的 handleB
public class HandleRuleB extends Handler {}  //...

// 客户端实现
class Client {
    public static void main(String[] args) {
        HandleRuleA handleRuleA = new HandleRuleA(1);
        HandleRuleB handleRuleB = new HandleRuleB(2);
        handleRuleA.setNextHandler(handleRuleB);  // 这是重点，将 handleA 和
handleB 串起来
        handleRuleA.echo(new Request());
    }
}
```

工程实践

下面通过代码向大家展示如何实现这一套流程：

```
// 定义一个抽象的规则
public abstract class BasicRule<CORE_ITEM, T extends RuleContext<CORE_
ITEM>>{
    // 有两个方法，evaluate 用于判断是否经过规则执行，execute 用于执行具体的规则
内容。
```

```
    public abstract boolean evaluate(T context);
    public abstract void execute(T context) {
}

// 定义所有的规则具体实现
// 规则 1：判断服务可用性
public class ServiceAvailableRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {
    @Override
    public boolean evaluate(UserPortraitRuleContext context) {
        TakeawayUserPortraitBasicInfo basicInfo = context.getBasicInfo();
        if (basicInfo.isServiceFail()) {
              return false;
        }
        return true;
    }

    @Override
    public void execute(UserPortraitRuleContext context) {}

}
// 规则 2：判断当前用户属性是否符合当前资源位投放的用户属性要求
public class UserGroupRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {
    @Override
    public boolean evaluate(UserPortraitRuleContext context) {}

    @Override
    public void execute(UserPortraitRuleContext context) {
        UserPortrait userPortraitPO = context.getData();
        if(userPortraitPO.getUserGroup() == context.getBasicInfo().
getUserGroup().code) {
          context.setValid(true);
        } else {
          context.setValid(false);
        }
    }
}

// 规则 3：判断当前用户是否在投放城市，具体逻辑省略
public class CityInfoRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {}
// 规则 4：根据用户的活跃度进行资源过滤，具体逻辑省略
public class UserPortraitRule extends BasicRule<UserPortrait,
UserPortraitRuleContext> {}

// 我们通过 spring 将这些规则串起来组成一个一个请求链
    <bean name="serviceAvailableRule" class="com.dianping.takeaway.
ServiceAvailableRule"/>
```

```
    <bean name="userGroupValidRule" class="com.dianping.takeaway.
UserGroupRule"/>
    <bean name="cityInfoValidRule" class="com.dianping.takeaway.
CityInfoRule"/>
    <bean name="userPortraitRule" class="com.dianping.takeaway.
UserPortraitRule"/>

    <util:list id="userPortraitRuleChain" value-type="com.dianping.
takeaway.Rule">
        <ref bean="serviceAvailableRule"/>
        <ref bean="userGroupValidRule"/>
        <ref bean="cityInfoValidRule"/>
        <ref bean="userPortraitRule"/>
    </util:list>

// 规则执行
public class DefaultRuleEngine{
    @Autowired
    List<BasicRule> userPortraitRuleChain;

    public void invokeAll(RuleContext ruleContext) {
        for(Rule rule : userPortraitRuleChain) {
            rule.evaluate(ruleContext)
        }
    }
}
```

责任链模式最重要的优点就是解耦，将客户端与处理者分开，客户端不需要了解是哪个处理者对事件进行处理，处理者也不需要知道处理的整个流程。在我们的系统中，后台的过滤规则会经常变动，规则和规则之间可能也会存在传递关系，通过责任链模式，我们将规则与规则分开，将规则与规则之间的传递关系通过 Spring 注入到 List 中，形成一个链的关系。当增加一个规则时，只需要实现 BasicRule 接口，然后将新增的规则按照顺序加入 Spring 中即可。当删除时，只需删除相关规则即可，不需要考虑代码的其他逻辑。从而显著地提高了代码的灵活性，提高了代码的开发效率，同时也保证了系统的稳定性。

四、总结

本文从营销业务出发，介绍了领域模型到代码工程之间的转化，从 DDD 引出了设计模式，详细介绍了工厂方法模式、策略模式、责任链模式以及状态模式这四种模式在

营销业务中的具体实现。除了这四种模式以外，我们的代码工程中还大量使用了代理模式、单例模式、适配器模式等等，例如在我们对 DDD 防腐层的实现就使用了适配器模式，通过适配器模式屏蔽了业务逻辑与第三方服务的交互。因篇幅原因不再进行过多的阐述。

对于营销业务来说，业务策略多变导致需求多变是我们面临的主要问题。如何应对复杂多变的需求，是我们提炼领域模型和实现代码模型时必须要考虑的内容。DDD 以及设计模式提供了一套相对完整的方法论帮助我们完成了领域建模及工程实现。其实，设计模式就像一面镜子，将领域模型映射到代码模型中，切实地提高代码的复用性、可扩展性，也提高了系统的可维护性。

当然，设计模式只是软件开发领域内多年来的经验总结，任何一个或简单或复杂的设计模式都会遵循上述的七大设计原则，只要大家真正理解了七大设计原则，设计模式对我们来说应该就不再是一件难事。但是，使用设计模式也不是要求我们循规蹈矩，只要我们的代码模型设计遵循了上述的七大原则，我们会发现原来我们的设计中就已经使用了某种设计模式。

五、参考资料

软件设计模式 – 百度百科
快速理解 – 设计模式六大原则
Software design pattern
《设计模式之禅》，秦小波，机械工业出版社
《领域驱动设计 – 软件核心复杂性应对之道》，Eric Evans，人民邮电出版社。
领域驱动设计在互联网业务开发中的实践

六、作者简介

吴亮亮，2017 年加入美团外卖，美团外卖营销后台团队开发工程师。

美团命名服务的挑战与演进

作者：舒超　张翔

本文根据美团基础架构部技术专家舒超在 2019 ArchSummit（全球架构师峰会）上的演讲内容整理而成。

命名服务主要解决微服务拆分后带来的服务发现、路由隔离等需求，是服务治理的基石。美团命名服务（以下简称 MNS）作为服务治理体系 OCTO 的核心模块，目前承载美团上万项服务，日均调用达到万亿级别。为了更好地支撑美团各项飞速发展的业务，MNS 开始从 1.0 向 2.0 演进。本文将围绕本次演进的初衷、实现方案以及落地的效果等方面进行展开，同时本文还介绍了命名服务作为一个技术中台组件，对业务的重要价值以及推动业务升级的一些成果。希望本文对大家能够有所启发。

一、MNS 1.0 简介

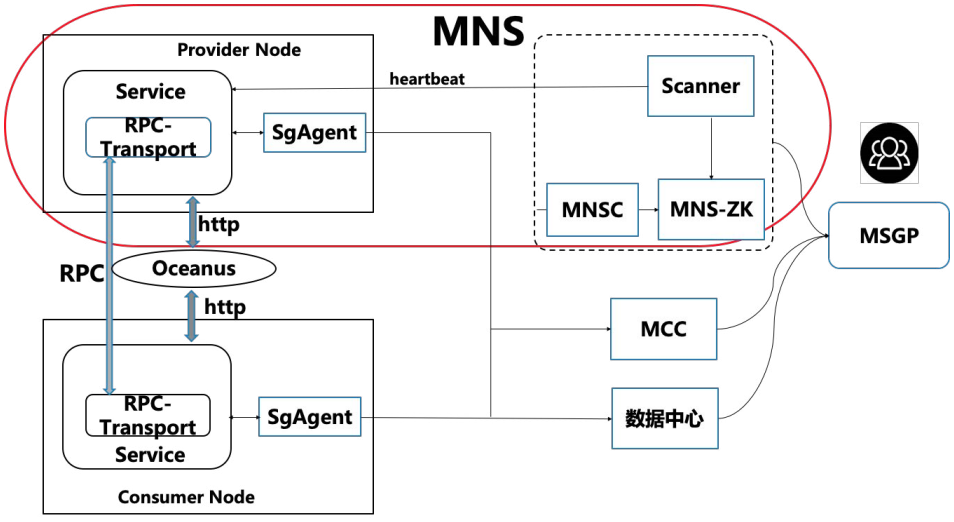


图 1　MNS 1.0 架构

从架构上看，MNS 1.0 主要分为三层：首先是嵌入业务内部的 SDK，用作业务自定义调用；然后是驻守在每个机器上的 SgAgent，以代理的方式将一些易变的、消耗性能的计算逻辑与业务进程分离开来，从而降低 SDK 对业务的侵入，减少策略变动对业务的干扰；远端是集中式的组件，包括健康检查模块 Scanner，鉴权缓存模块 MNSC，以及基于 ZooKeeper（以下简称 ZK）打造的一致性组件 MNS-ZK，作为通知和存储模块。在层级之间设立多级缓存，利用“边缘计算”思想拆分逻辑，简化数据，尽量将路由分配等工作均摊到端上，从而降低中心组件负载。更多详情大家可参考《美团大规模微服务通信框架及治理体系 OCTO 核心组件开源》一文中的 OCTO-NS 部分。

在体量方面，MNS 1.0 已经接入了美团所有的在线应用，涉及上万项服务、数十万个节点，并覆盖了美团所有的业务线，日均调用达万亿级别，目前我们已将其开源。

总的来讲，作为公司级核心服务治理组件，MNS 1.0 在架构上带有明显的 CP 属性，在保持架构简洁、流程清晰的同时，还支持了业务大量迭代的需求，较好地帮助公司业务实现了服务化、标准化的目标。

二、MNS 1.0 遇到的问题和挑战

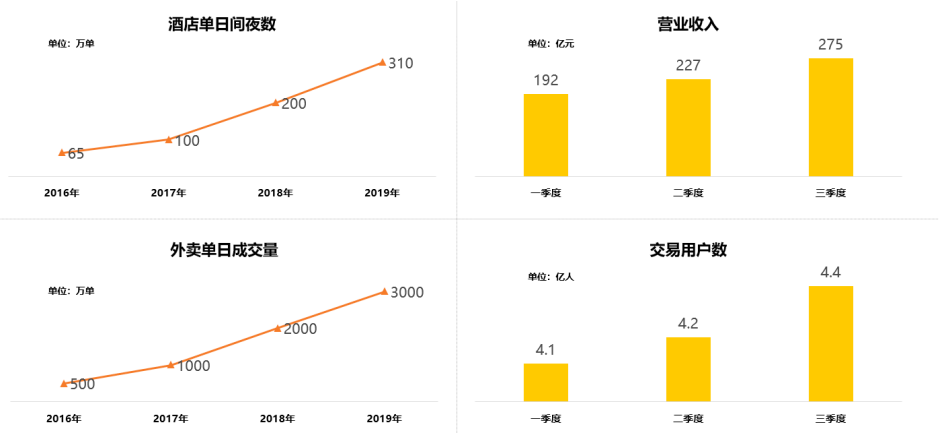


图 2　近三年美团业务增长数据

但是随着美团业务的快速增长，公司的服务数、节点数、服务信息量、服务变动频次等维度都在快速增长，有些服务甚至呈现出跨数量级的增长。在这样的情况下，命名服务也面临着一些新的问题和挑战：

- **可用性**：公司业务持续高速增长，很多都需要进行跨地域的部署。在 MNS 1.0 架构下，地域之间的专线如果断连，会发生一个地域命名服务整体不可用的情况；其次，强一致组件存在单点问题，Leader 出现故障，整个集群中断服务；另外，在多客户端和大数据传输的命名服务场景下，CP 系统恢复困难，RTO 达到小时级别。MNS 1.0 曾经出现过后端集中式组件单机连接超过十万且活跃链接数过半的情况，出现问题之后，现场恢复的负荷可想而知，这些都给命名系统的可用性带来很大的风险。
- **扩展性**：相对于需要支持的业务数量，MNS 1.0 整体平行扩展能力不足。MNS-ZK 的单集群规模上限不超过 300（实际只能达到 250 左右，这与 ZooKeeper 内部的 myid 等机制有关），否则同步性能会急剧恶化；其次，集群写入不可扩展，参与写入节点越多性能越差；另外，服务信息快照在固定的时间片内持续增长，增加 IO 压力的同时也延长了迁移的同步时间。而扩展性上的短板，导致 MNS 1.0 难以应对突发的流量洪峰，易出现“雪崩”。
- **性能**：写入操作受 CP 属性限制，串行性能较低。MNS-ZK 整体到 7000 左右的写量就已接近上限。其次，在热点服务场景下，数据分发较慢，这跟数据粒度较粗也有一定关系。而数据粒度粗、量大，也会在组件间传输消息时，导致临时对象频繁生成，引起 GC。另外，MNS 1.0 的后端集群负载还存在均衡性问题，一是因为原架构中缺乏集中管控服务，无法进行动态的集群与数据拆分伸缩，二是因为强一致属性下，集群节点间基于 Session 的迁移现场较重，很容易因一个节点挂掉而引起连锁反应。

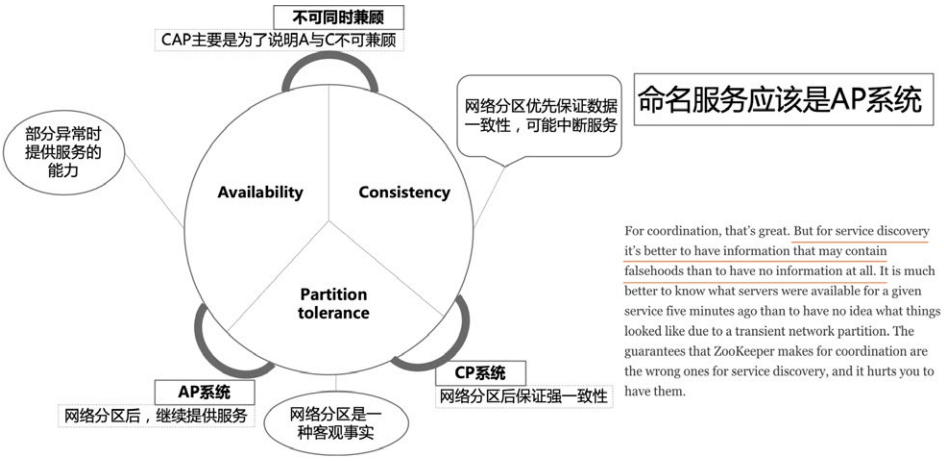


图 3　命名服务应该是 AP 系统

从可用性、扩展性、性能等三个方面，MNS 1.0 暴露出很多的问题，究其根源，原来的命名服务作为一个 CP 系统，为获得“数据一致性”而牺牲了部分情况下的可用性。其实，对于命名服务而言，一致性并没有这么重要，最多是调用方在一定时间内调多了或调漏了服务方而已，这个后果在不可靠网络的大前提下，即使命名服务没有出现问题，也可能经常会出现。借助端的自我检查调整机制，其影响可以说微乎其微。而另一方面，如果是一个机房或一个地域的调用方和服务方没有问题，但是因为这个区域和命名服务主集群断开了链接，从而导致本域内不能互调，这个就显得不太合理了。

所以总结一下，我们认为，**命名服务的本质是建立和增强服务本身的连通性，而不是主动去破坏连通性**。命名服务本质上应该是一个 AP 系统。

其实，业界对命名服务 AP/CP 模式都有相应实现和应用，很多企业使用 CP 模式，原因可能有以下几点：

- 架构行为简单：CP 系统在出现分区时行为比较简单，冷冻处理，粗暴但相对不容易出错。
- 启动门槛低：一些成熟的开源一致性组件，比如 ZK、etcd 都是 CP 系统，能够开箱即用，在数据规模可控的情况下，基本能够满足企业的需求。

另外，我们了解到，一些公司使用特殊的方式弱化了这个问题，比如将 CP 系统进一步拆分到更小的域中（比如一个 IDC），缩小分区的粒度，而全局有多个 CP 系统自治。当然，这个可能跟调用方服务方的跨度限制或者说调用配套部署有关系，但也有可能带来更复杂的问题（比如 CP 系统之间的数据同步），这里就不做详细的讨论了。

除去 MNS 1.0 本身的架构缺陷，我们还需要面临另一个问题，当初在项目启动时，云原生尚处于起步阶段，而如今，一些基于云原生理念兴起的网络基础设施，尤其是 Service Mesh 在美团快速发展，也需要 MNS 进行改造去适配新的流量通道和管控组件，这也是此次 MNS 2.0 演进的目标之一。

综上，我们以 AP 化、Mesh 化为主要目标，正式开始了从 MNS 1.0 向 MNS 2.0 的演进。

三、MNS 2.0

1. 整体架构

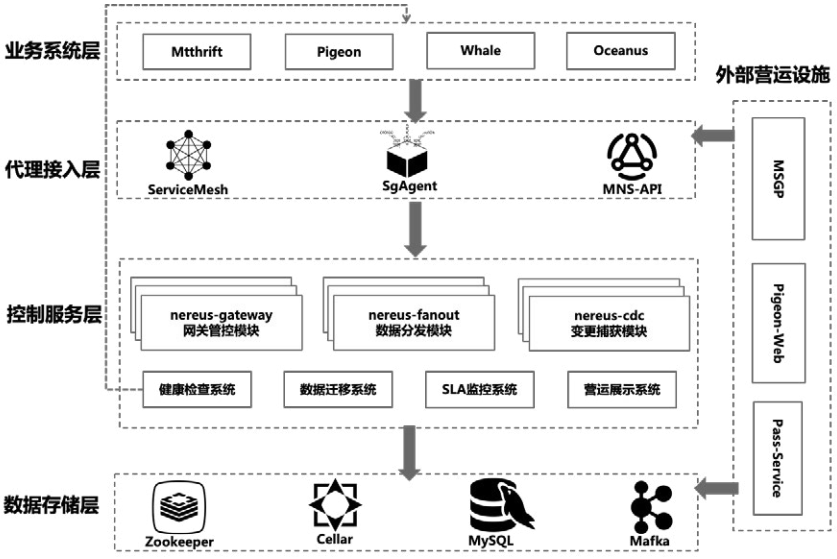


图 4　MNS 2.0 整体架构

MNS 2.0 的整体架构自上而下主要分为四层：

- 业务系统层：这一层与 MNS 1.0 架构类似，依然是嵌入到业务系统中的 SDK 或框架，但是不再感知服务列表和路由计算，从而变得更加的轻薄。
- 代理接入层：这一层主要变化是加入了 Service Mesh 的 Sidecar 和 MNS-API。前者将命名服务的部分链路接入 Mesh；后者为 MNS 增加了更丰富的 HTTP 调用，帮助一些没有使用 SDK 或框架的业务快速接入到命名服务。整个代理接入层的改造使得 MNS 对接入业务更加亲和。
- 控制服务层：新增注册中心控制服务，这也是 MNS 2.0 的核心。主要分为以下三个模块：
  - 网关管控模块：提供集中式的鉴权、限流 / 熔断、统计等 SOA 服务化的管控能力，同时避免海量代理组件直连存储层。
  - 数据分发模块：数据的通道，包括注册数据的上传、订阅数据的下发，可进行精细化数据拆分和平行扩展来适配热点服务。
  - 变更捕获模块：服务注册发现的审计，包括对第三方系统进行事件通知和回调，支持多元化的服务数据营运需求。
  - 另外，控制服务层还包括全链路 SLA 监控等新的子模块，以及健康检查 Scanner 这样的传统 MNS 组件。
- **数据存储层**：我们进一步丰富了命名服务的存储和分发介质，提高了数据层的整体性能。主力存储使用 K/V 存储系统（美团 Cellar 系统）替代 MNS-ZK，有效地提高了数据的吞吐能力，支持网络分区后的数据读写以及宕机灾备，同时保留 ZK 做一些轻量级的 Notify 功能。新增的关系型数据库和消息队列（美团 Mafka 系统），配合控制层的变更捕获模块，提供更方便的数据挖掘结构和外部扇出。

旁路于上面 4 层的是外部营运设施，主要是业务端的可视化 Portal，用户可以在上面对自身服务进行监控和操作，美团的基础研发部门也可以在上面进行一些集中式的管控。

综上所述，MNS 2.0 整体架构在兼容 1.0 的前提下，重点变化是：新增了控制服务层并对底层存储介质进行拆解。下面，我们来看一下服务注册发现功能在 MNS 2.0 中的实现流程：

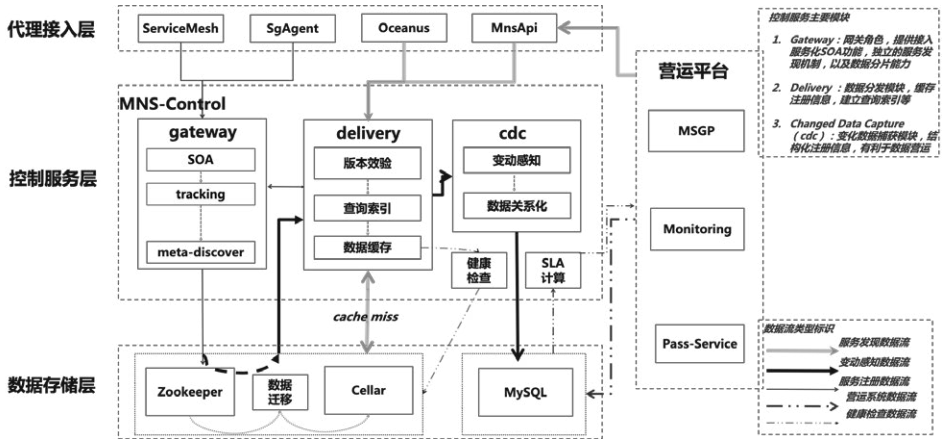


图 5　MNS 2.0 中的服务注册发现流程

1. **服务注册**：代理接入层透传业务注册请求，经过控制服务层的管控模块（Gateway）一系列 SOA 和审计流程后，写入注册数据到存储层。
2. **数据感知**：控制服务监听数据变动，服务注册写入新信息后，分发模块（Delivery）更新内存中的缓存，数据流经过捕获模块（CDC）将注册信息关系化后存储到 DB。
3. **服务发现**：代理层请求经过控制服务的管控模块（Gateway）效验后，从分发模块（Delivery）的缓存中批量获取服务端注册信息。Cache Miss 场景时从存储层读取数据。
4. **外部交互层**：外部系统当下通过代理层接入整个 MNS 体系，避免直连存储带来的各类风险问题。未来，营运平台可直接使用准实时 DB 数据，以 OLAP 方式进行关系化数据的分析。

接下来，我们一起来看下 MNS2.0 的主要演进成果。

2. 演进成果

2.1 流量洪峰 & 平行扩展

流量洪峰对于不同领域而言有不同的时段。对于 O2O 领域比如美团来说，就是每天中午的外卖高峰期，然后每天晚上也会有酒旅入住的高峰等等。当然，基于其它的业务场景，也会有不同的高峰来源。比如通过“借势营销”等运营手段，带来的高峰量可能会远超预期，进而对服务造成巨大的压力。

图 6　流量洪峰

MNS 1.0 受制于 MNS-ZK 集群数量上限和强一致性的要求，无法做到快速、平行扩展。MNS 2.0 的数据存储层重心是保证数据安全读写和分布式协调，在扩展能力层面不应该对其有太多的要求。MNS 2.0 的平行扩展能力主要体现在控制服务层，其具体功能包括以下两个方面：

集群分片：控制服务提供全量注册数据的分片能力，解决命名服务单独进行大集群部署时不能进行业务线隔离的风险。MNS-Control 网关模块分为 Master 和 Shard 等两类角色，协作提供大集群分片能力。

- **Master**：维护服务注册信息与各个分片集群（Shard）的映射关系，向代理层组件提供该类 Meta 数据。Master 接收各个 Shard 集群事件，新增、清理 Shard 中维护的注册信息。

- **Shard**：数据分片自治向代理组件提供服务注册 / 发现功能，实现按业务线隔离命名服务，同时按照 Master 发出的指令，调整维护的注册信息内容。
- **Services**：业务服务通过代理组件接入 MNS，代理组件启动时通过与 Master 的交互，获知归属的 Shard 集群信息以及 Fallback 处理措施。此后 Agent 只与业务线 Shard 进行命名数据交互，直到 Shard 集群不可用，重新执行“自发现”流程。
- **Fallback 措施**：设置一个提供全量服务信息的默认 Shard 集群，当业务线 Shard 异常时。一方面，Agent 重启“自发现”流程，同时将命名请求重定向到默认的 Shard 集群，直到业务线 Shard 恢复后，流量再切回。

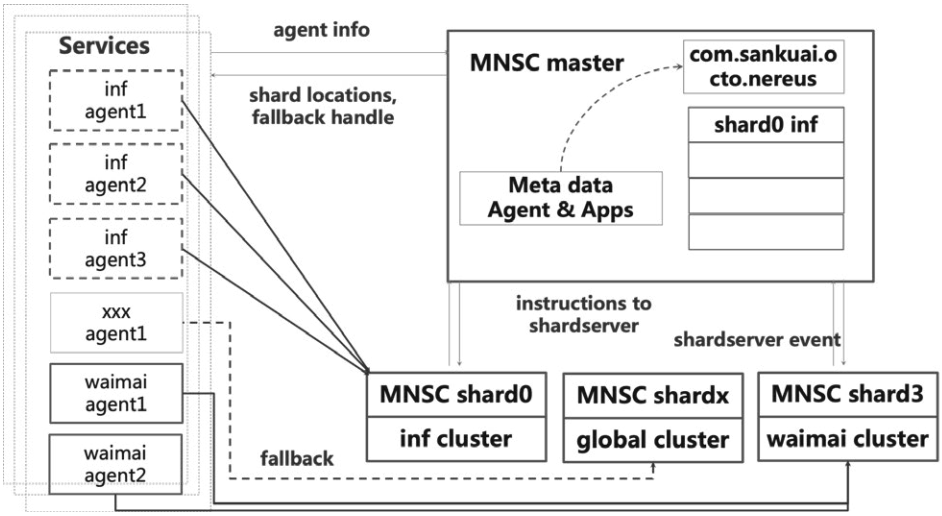


图 7　控制服务数据分片示意图

网络分区可用：MNS 1.0 阶段，网络分区对可用性的影响巨大，分区后 MNS-ZK 直接拒绝服务。新架构将存储迁移到 KV 系统后，在网络专线抖动等极端情况下，各区域依然能正常提供数据读取功能。同时，我们与公司存储团队共建 C++ SDK 的地域就近读写功能。一方面，提高跨域服务注册发现的性能，另一方面，实现了网络分区后，各区域内命名服务可读、可写的目标，提高了系统的可用性，整个命名服务对网络的敏感度降低。

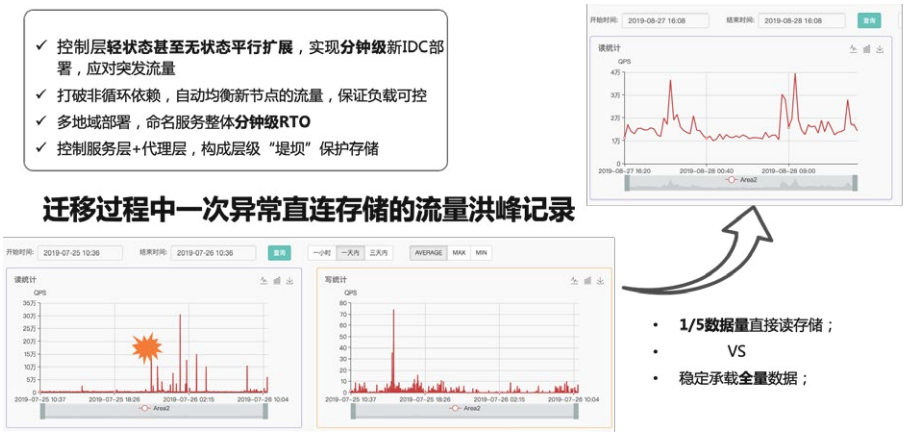
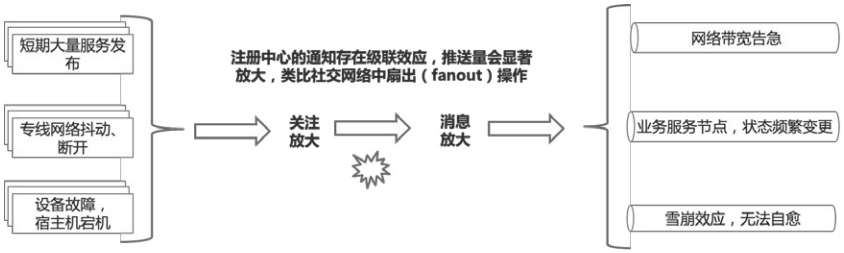


图 8　命名服务的平行扩展

目前，控制服务层在平行扩缩容时间和集群原地恢复时间等方面，都达到了分钟级，集群也没有节点上限，从而能够有效应对突发的流量，防止服务出现“雪崩”。

2.2 推送风暴 & 性能提升

另一个典型的场景是推送“风暴”，在服务集中发布、出现网络抖动等情况下，会导致命名服务出现”一横一纵”两种类型的放大效应：横向是“关注放大”，类似于社交网络中某大 V 消息需要分发给众多的粉丝，服务越核心，放大的效果越明显。纵向是“级联放大”，命名服务的上下游会逐级进行拷贝发送，甚至一级上下游会针对一个消息有多次的交互（Notify+Pull）。



$$\text{推送规模} = N\text{服务变化} \times 1\text{次放大} \times 2\text{次放大}$$

$$1000N = N \times 100 \times 10$$

图 9　命名服务领域的消息放大现象

“关注放大”和“级联放大”本身都是无法避免的，这是由系统属性决定，而我们能做的就是从两方面去平滑其带来的影响：

正面提升核心模块性能，增强吞吐、降低延迟

1. 结构化聚合注册信息：控制服务的数据分发模块，在内存中存储结构化的服务注册信息，提供批量数据的读取能力；降低多次网络 RTT 传输单个数据以及数据结构转化带来的高耗时。 2. 高并发的吞吐能力：控制服务通过无锁编程处理关键路径的竞争问题，借助应用侧协程机制，提供高并发、低延迟的数据分发功能。
2. 与存储团队共建，实现 KV 系统就近区域读写，提高命名服务的服务注册（数据写入）性能。

另外，包括前面提到的控制服务集群的平行扩展能力，其实也是整体性能提升的一种方式。

侧面疏通，区分冷热数据，降低推送的数据量，提高效能

自然界中普遍存在“80/20 法则”，命名服务也不例外。服务注册信息的结构体中元素，80% 的改动主要是针对 20% 的成员，比较典型的就是服务状态。因此，我们将单个整块的服务信息结构体，拆分为多个较小的结构体分离存储；当数据变动发生时，按需分发对应的新结构体，能够降低推送的数据量，有效减少网络带宽的占用，避免代理组件重复计算引起的 CPU 开销，数据结构变小后，内存开销也得到显著降低。

那是否需要做到完全的“按需更新”，仅推送数据结构中变动的元素呢？我们认为，完全的“按需更新”需要非常精细的架构设计，并会引入额外的计算存储开销。比如，我们需要将变动成员分开存储以实现细粒度 Watcher，或用专门服务识别变动元素然后进行推送。

同时，还要保证不同成员变动时，每次都要推送成功，否则就会出现不一致等问题。在组件计算逻辑所需的数据发生变化时，也会带来更多的改动。在命名服务这样的大

型分布式系统中，“复杂”、“易变”就意味着稳定性的下降和风险的上升。所以根据“80/20 法则”，我们进行尺度合理的冷热数据分离，这是在方案有效性和风险性上进行权衡后的结果。

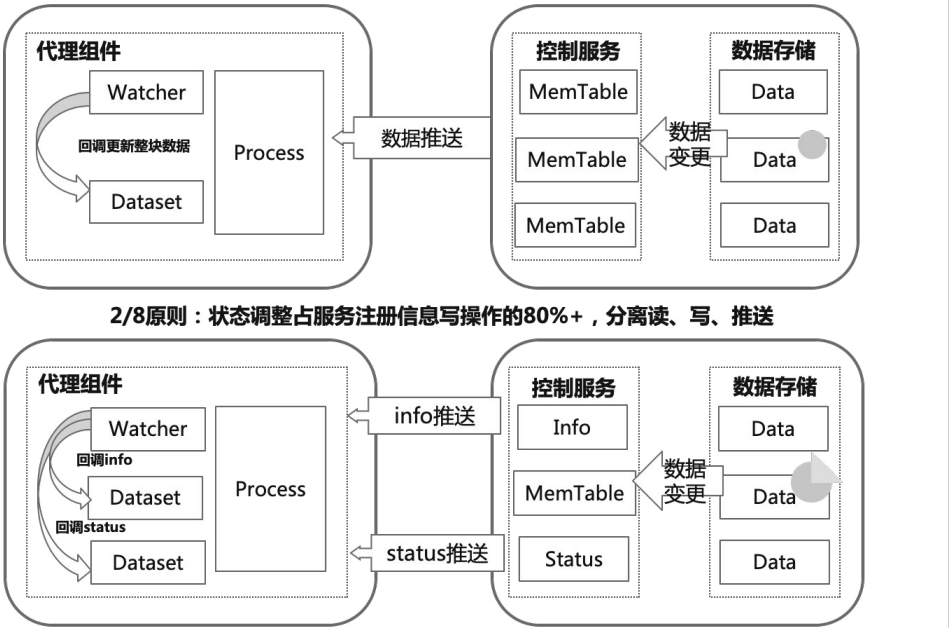


图 10　冷热数据分拆推送

经过改造，MNS 2.0 相比 MNS 1.0 的吞吐能力提升 8 倍以上，推送成功率从 96% 提升到 99%+，1K 大小服务列表服务发现的平均耗时，从 10s 降低到 1s，TP999 从 90s 下降到 10s，整体优化效果非常明显。

2.3 融入 Service Mesh

在 MNS 2.0 中，我们将代理服务 SgAgent 部分注册发现功能合并到了 Mesh 数据面，其流程如下图所示：

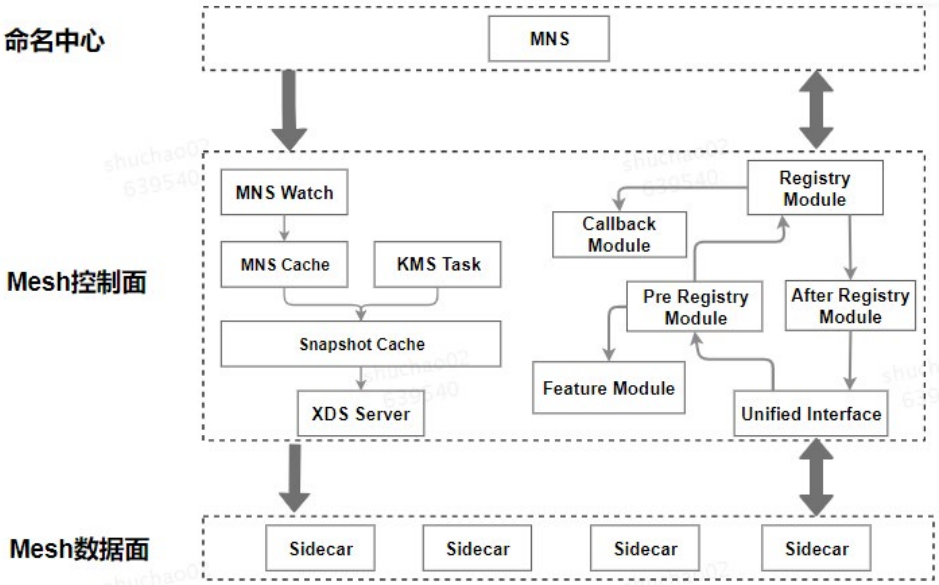


图 11　命名服务与 Service Mesh 的融合

关于美团服务治理功能与 Service Mesh 结合的技术细节，这部分的内容，我们今年会单独做一个专题来进行分享，感兴趣的同学可以关注“美团技术团队”微信公众号，敬请期待。

2.4 无损迁移

除了上面说到的 MNS 2.0 的这些重点演进成果之外，我们还想谈一下整个命名服务的迁移过程。像 MNS 这样涉及多个组件、部署在公司几十万个机器节点上、支撑数万业务系统的大规模分布式系统，如何进行平滑的数据迁移而不中断业务正常服务，甚至不让业务感知到，是 MNS 2.0 设计的一个重点。围绕业务服务无感知、具备快速回滚能力、新 / 老体系互备数据不丢失等要求，我们设计了如下图所示的迁移流程：

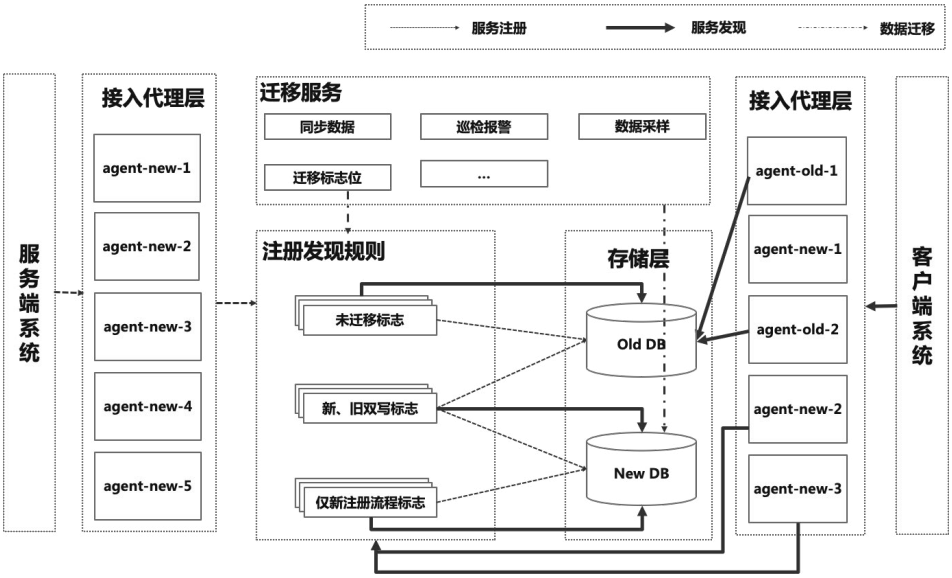


图 12　MNS 2.0 的无损迁移

MNS 2.0 整体以服务为粒度进行迁移操作，设置标志位说明服务所处的状态，由接入代理层组件识别该标志做出相应的处理。标志位包括：

1. 未迁移标志：服务注册 / 发现走 MNS 1.0 的流程，注册信息存储到重构前的 MNS-ZK 中。
2. 迁移中标志：服务注册并行走 MNS 1.0 和 MNS 2.0 流程，数据同时写入新旧两个地方，服务发现执行 MNS 2.0 流程。
3. 已迁移标志：服务注册 / 发现全部走 MNS 2.0 流程，注册信息仅存储到 MNS 2.0 的数据层。该阶段无法进行平滑的回滚，是项目长期验证后最终的迁移状态。

上述可以总结为：**聚焦单个服务，阶段性迁移服务发现流量，从而达到类似系统新功能发布时“灰度上线”的能力**。当然，这个策略还涉及一些细节在其中，比如，分开存储在双写时需要重点去保证异常情况下的最终一致性等等，鉴于篇幅原因，这里就不详细展开讨论了，而且业界针对这种情况都有一些成熟的做法。

另外，我们通过优化命名服务发布系统的发版形式，实现自动化的流量灰度策略，降低了人力成本，同时构建了自动化的迁移工具、巡检工具，高效地进行自动化的数据迁移工作，能够快速巡检迁移后的链路风险，保障新 MNS 2.0 的稳定上线。

2.5 演进总结

在美团命名服务这样的大型分布式系统优化过程中，具体到架构和功能设计层面，我们也做了不少的权衡和取舍，前面我们也提到，放弃了增量式的精准变动信息推送方式。在考虑性能的前提下，也没有使用数据多维度营运能力更好的 MySQL 作为主要存取介质等等。取舍的核心原则是：改造目标时强调业务收益，落地过程中减少业务感知。

目前，MNS 2.0 主要成果如下：

- 重构了 5 个已有的核心组件，研发了 2 个全新的系统，完成公司 PaaS 层数十个服务的功能的适配改造，目前已成功迁移全公司 80% 以上的服务，且在迁移过程中无重大事故。
- RTO 从数小时降到分钟级，RPO 为 0；全链路推送耗时 TP999 从 90s 降到了 10s，推送成功率从 96% 提升到 99% 以上，基本完成了百万级别服务节点治理能力的建设。
- 集群数据按照业务线等多维度进行拆分，建立了基于分级染色的 SLA 指标和定期巡检机制，同时根据公司实际场景增加了众多的服务风控审计功能，保障了业务安全。
- 云原生方面，支持了 Service Mesh，注册发现流程融入了 Mesh 底层基础设施。

四、命名服务对业务的赋能

命名服务本身作为基础的技术中台设施，在坚持“以客户为中心”，升级自身架构的同时，也从如下几个方面对美团的多个业务进行赋能。

4.1 单元化 & 泳道

单元化（SET 化）是业界比较流行的容灾扩展方案，关于美团单元化的详细内容，可参考 OCTO 团队在本次 ArchSummit 中的另一个专题分享《SET 化技术与美团点评实践解密》。这里主要是从命名服务对单元化支撑的角度去解答这个问题。

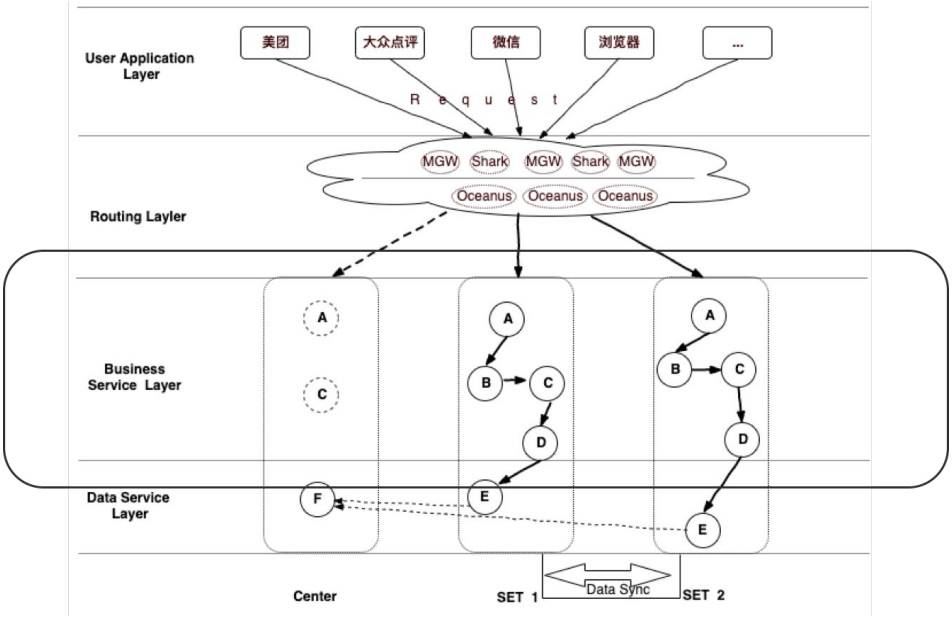


图 13　命名服务对单元化的支持

如上图所示，业务多种来源的外网流量在通过网关进入内网后，会借助命名服务提供的能力（SDK/Agent），并按照业务自定义的核心数据维度和机器属性，给流量打上单元化标签，然后路由到标签匹配的下一跳，从而实现了单元间流量隔离。一个单元内部，从服务节点到各种存储组件，都依赖于命名服务提供的单元识别和路由能力来完成隔离，所以命名服务在单元化中主要起底层支撑的作用。目前单元化在美团的重点业务，比如外卖、配送已经建设的比较完备，通过一定的单元冗余度，能在一个单元出现问题时，切换到另一个可用的镜像单元，显著提高了业务整体可用性。

接下来，我们再来看一下泳道场景。目前泳道在美团这边主要用于业务做完代码 UT 之后的线下集成测试阶段，同时结合容器，实现一个即插即用的上下游调用环境去验

证逻辑。命名服务在其中起到的作用与单元化类似，根据泳道发布平台对机器的配置，自动编排上下游的调用关系。如下图所示：

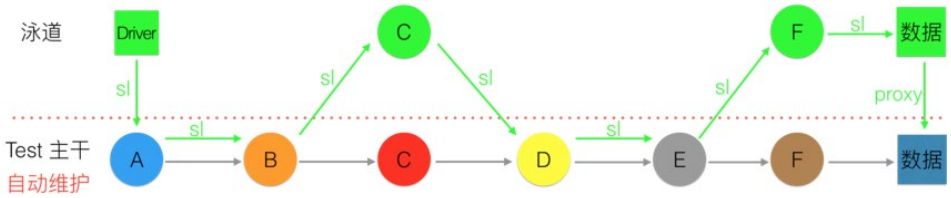


图 14　泳道流量示意图

当下一跳存在泳道节点时，测试流量进入泳道。反之，测试流量回流到主干。每个节点重复上述过程。

4.2 平滑发布 & 弹性伸缩

命名服务另外一个重要场景是服务的平滑发布，我们与发布平台配合，控制发布流量的自动摘除与恢复，实现服务发布操作的自动化与透明化。具体流程如下图所示：

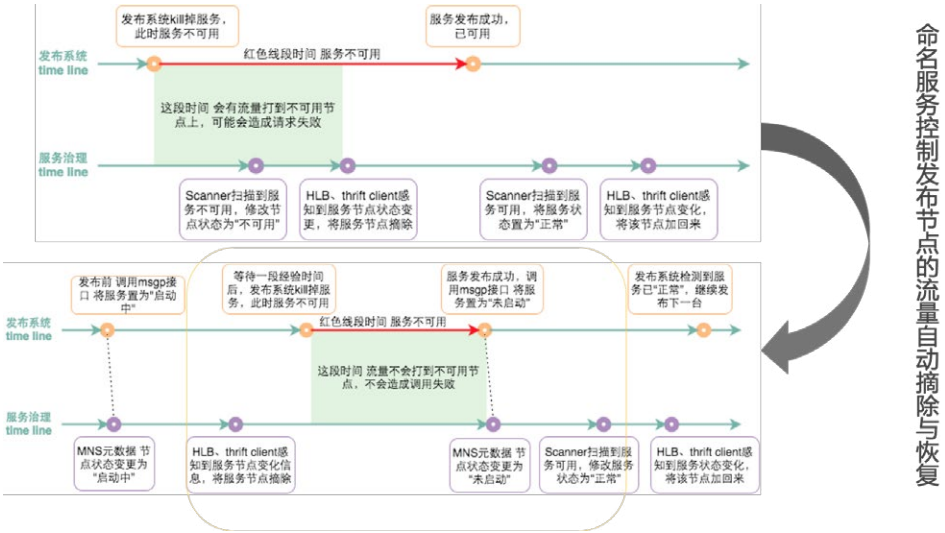


图 15　命名服务支持平滑发布

早期的发布流程，存在异常调用的风险，上线过程中存在一段服务实例不可用的“时间窗口”，此时流量再进入可能导致调用失败，然后会报错。为保证发布的稳定性，需要操作人员手动进行流量排空，流程上效率低、易出错，浪费业务团队的时间。针对这项业务痛点，命名服务向发布系统提供针对服务实例的流量管控能力，实现进程重启前自动摘除并清空来源请求，升级完毕后，提供自动流量恢复的平滑发布功能。智能地解决发版过程中的异常调用问题，提高公司的服务上线效率，降低了业务团队的运维成本。

弹性伸缩是容器一个很大的卖点。但是在伸缩过程中有很多问题需要考虑，比如扩缩容策略，包括调用亲和度配置，路由均衡等等。命名服务和容器化合作，提供业务的上下游调用关系、分组设置信息、容器下线流量无损摘除等服务，同时保障伸缩服务注册及时、高效。如下图所示：

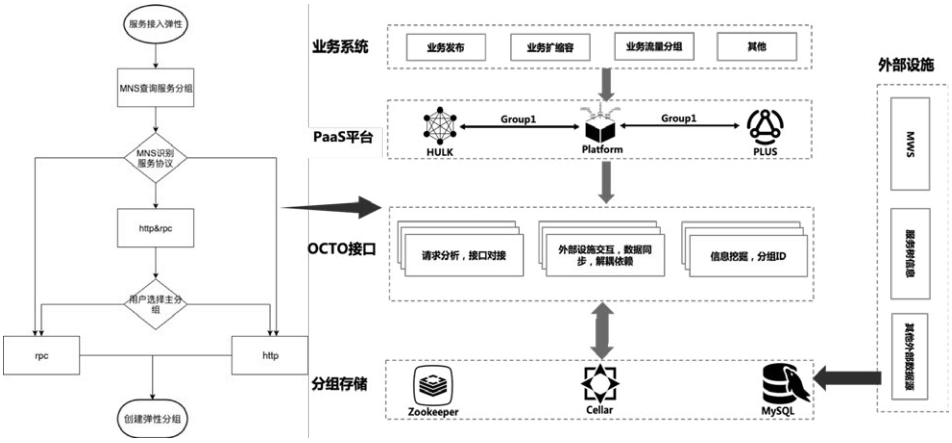


图 16　命名服务支持弹性伸缩

4.3 服务数据

MNS 服务数据对业务的赋能主要分为两个部分，一是将自身运行状态以业务可理解的 SLA 暴露出来，方便业务评估命名服务健康状况。对于命名服务来说，SLA 指标主要有推送成功率和推送耗时两种（其实，推送耗时也可以看做成功率的一个衡量维度，这里暂时不做太详细的区分）。

MNS 1.0 精准量化运行状况的困难在于，一方面 MNS 的发布 / 订阅机制重度依赖 ZK，受限于 ZK 的自身实现和链路上众多不同组件的异构性，及时获取完整链路的推送行为数据很困难。另一方面，由于节点数众多，全量采集服务发现数据，对公司的监控上报系统以及采集之后的计算也是很大的负担。

在 MNS 2.0 中，我们首先在架构上显著弱化了 ZK 的地位，它基本上只作为一致性通知组件。我们自研的 MNS-Control 可以充分 DIY 埋点场景，保证了主要推送行为抓取的可控性。其次，我们采用了分级采样计算，在全面梳理了公司现有的服务节点数比例后，将典型的服务列表数划分为几个档次，比如 1000 个节点的服务为一档，100 个又为一档，并创建相应的非业务哨兵服务，然后在不同机房选取适量的采样机器定期注册 + 拉取来评估整体的运行情况。这样既做到了上报量的精简，又兼顾了上报数据的有效性。详细操作流程，如下图所示：

图 17　MNS 2.0 SLA 采集

控制层周期性修改采样服务中的服务数据，触发服务发现的数据推送流程。参与指标统计的机器节点，本地代理进程获取到注册信息推送后，上报送达时间到运维平台并由其写入存储层。 控制层对数据库中的数据进行聚合计算，最后上报监控系统展示指标数据。此外，通过与监控团队合作，解决全量部署的 Agent 埋点监控问题。

命名服务存储的服务信息有很高的业务价值，从中可以知道服务的部署情况、发布频次以及上下游拓扑信息等等。所以“赋能”的第二个部分在于，借助这些信息，我们可以挖掘出业务服务部署运维上不合理的地方或风险点，不断推动优化，从来避免一

些不必要的损失。目前已经在推动的项目包括：

- 单进程多端口改造：业务使用这种方式去区分不同的调用端口，从而造成端口资源的浪费，而且调用下游还要感知部署信息，这种机器属性粒度细节的暴露在云原生时代是不太恰当的。我们协助业务将调用行为改成单端口多接口的形式，在协议内部去区分调用逻辑。
- 大服务列表的拆分：随着业务的发展，单个服务的节点数呈爆发式增长，甚至出现了一个服务有接近 7W 的节点数的情况，对发布、监控、服务治理等底层设施产生很大的压力。我们通过和业务的沟通，发现大列表产生的原因主要有两点：1. 单机性能不够，只能以实例抗；2. 服务内容不够聚焦。换句话说，因为服务还不够“微”，所以导致逻辑越来越庞大，有需求的调用方和自身实例相应增加，代码风险也在增大。因此，我们和业务一起梳理分析架构和调用，将核心共用模块与业务逻辑群分拆出来，减少冗余调用，最终使一些大服务节点数量从数万降低到数千，实现了数量级的下降。
- 上下游均衡部署：这个比较容易理解，结合调用端与服务端比例、服务方机器性能以及调用失败率等信息，可以作为服务业务在各机房间均衡调整机器数量的参考。另一方面，我们也在减少基本就近调用策略的粒度，目前只保留了“同 IDC”和“同城”两种，去掉了之前的“同中心策略”，减轻业务服务部署的心智负担。后期，随着机房数量的降低和同地域机房间延时的可控，同城路由可能是最终的方案。

在架构上，MNS 2.0 依赖 DB 和其它一些数仓介质，进行多种维度的数据上卷和下钻，并结合一些定制的风险策略逻辑去帮助业务发现和规避问题，目前这个事情也在进行中。

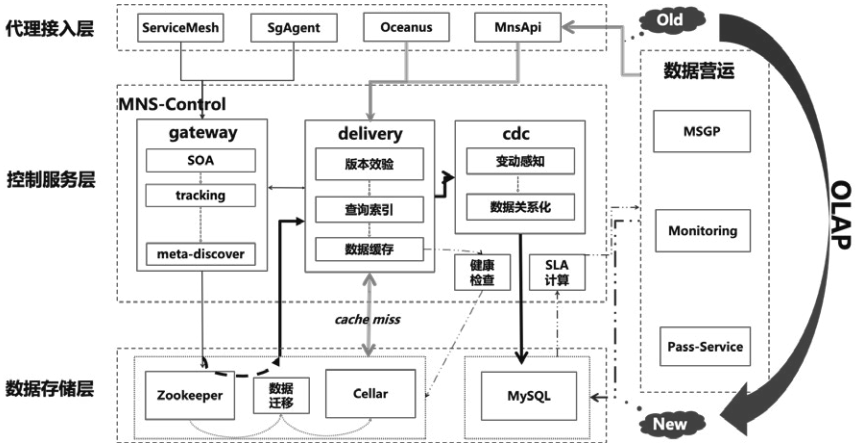


图 18　MNS 2.0 数据赋能

五、未来展望

未来，美团命名服务主要会朝两个方向发展：

- 进一步收集、挖掘服务数据的价值，打造服务数据平台，赋能业务升级。从单纯实现业务注册发现路由等需求，到通过数据反向推动业务架构流程改造升级，这也是美团核心价值观“以客户为中心”的一种体现。
- 深度结合 Service Mesh 等云原生基础设施的演进。云原生理念及相关设施架构的快速发展，必然会造成传统服务治理组件架构上流程的变动，深度拥抱云原生，并享受基础功能下沉带来的“红利”，是命名服务一个比较确定的方向。

作者简介

舒超，美团资深技术专家，OCTO 服务治理团队研发核心成员。
张翔，美团高级工程师，美团 OCTO 服务治理团队研发核心成员。

招聘信息

美团 OCTO 服务治理团队诚招 C++/Java 高级工程师、技术专家。我们致力于研发公司级、业界领先的基础架构组件，研发范围涵盖分布式框架、命名服务、Service Mesh 等技术领域。欢迎有兴趣的同学投送简历至 tech@meituan.com（邮件备注：美团 OCTO 团队）。

美团 MySQL 数据库巡检系统的设计与应用

作者：王琦

巡检工作是保障系统平稳有效运行必不可少的一个环节，目的是能及时发现系统中存在的隐患。我们生活中也随处可见各种巡检，比如电力巡检、消防检查等，正是这些巡检工作，我们才能在稳定的环境下进行工作、生活。巡检对于数据库或者其他 IT 系统来说也同样至关重要，特别是在降低风险、提高服务稳定性方面起到了非常关键作用。

本文介绍了美团 MySQL 数据库巡检系统的框架和巡检内容，希望能够帮助大家了解什么是数据库巡检，美团的巡检系统架构是如何设计的，以及巡检系统是如何保障 MySQL 服务稳定运行的。

一、背景

为了保障数据库的稳定运行，以下核心功能组件必不可少：

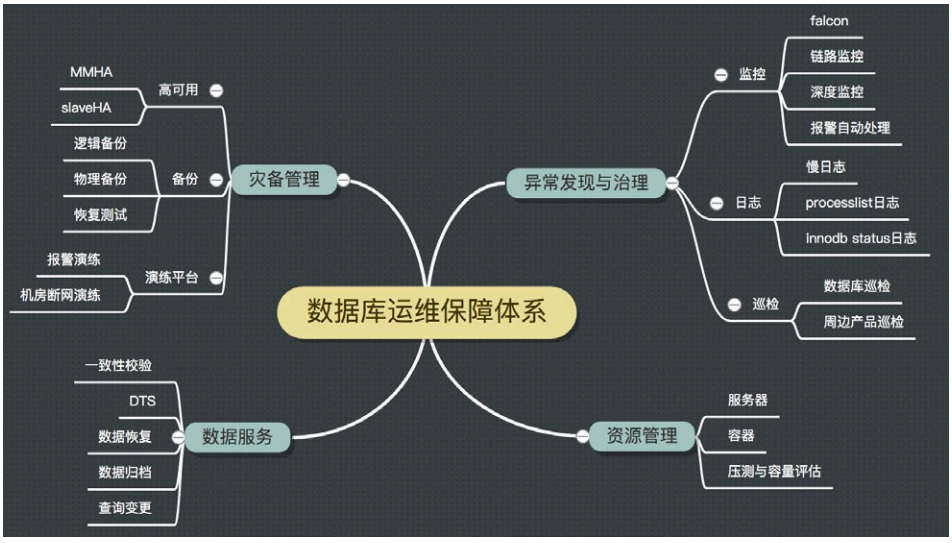


图 1　数据库运维保障核心功能组件

其中，数据库巡检作为运维保障体系最重要的环节之一，能够帮助我们发现数据库存在的隐患，提前治理，做到防患于未然。对于大规模集群而言，灵活健壮的自动化巡检能力，至关重要。

任何系统都会经历一个原始的阶段，最早的巡检是由中控机 + 定时巡检脚本 + 前端展示构成的。但是，随着时间的推移，老巡检方案逐渐暴露出了一些问题：

- 巡检定时任务执行依赖中控机，存在单点问题；
- 巡检结果分散在不同的库表，无法进行统计；
- 巡检脚本没有统一开发标准，不能保证执行的成功率；
- 每个巡检项都需要单独写接口取数据，并修改前端用于巡检结果展示，比较繁琐；
- 巡检发现的隐患需要 DBA 主动打开前端查看，再进行处理，影响整体隐患的治理速度；
- ……

所以我们需要一个灵活、稳定的巡检系统来帮助我们解决这些痛点，保障数据库的稳定。

二、设计原则

巡检系统的设计原则，我们从以下三个方面进行考虑：

稳定：巡检作为保证数据库稳定的工具，它自身的稳定性也必须有所保证；

高效：以用户为中心，尽量化繁为简，降低用户的使用成本，让新同学也能迅速上手治理和管理隐患；提高新巡检部署效率，随着架构、版本、基础模块等运维环境不断变化，新的巡检需求层出不穷，更快的部署等于更早的保障；

可运营：用数据做基础，对巡检隐患进行运营，包括推进隐患治理，查看治理效率、趋势、薄弱点等。

三、系统架构

美团 MySQL 数据库巡检系统架构图设计如下。接下来，我们按照架构图从下到上的顺序来对巡检系统主要模块进行简单的介绍：

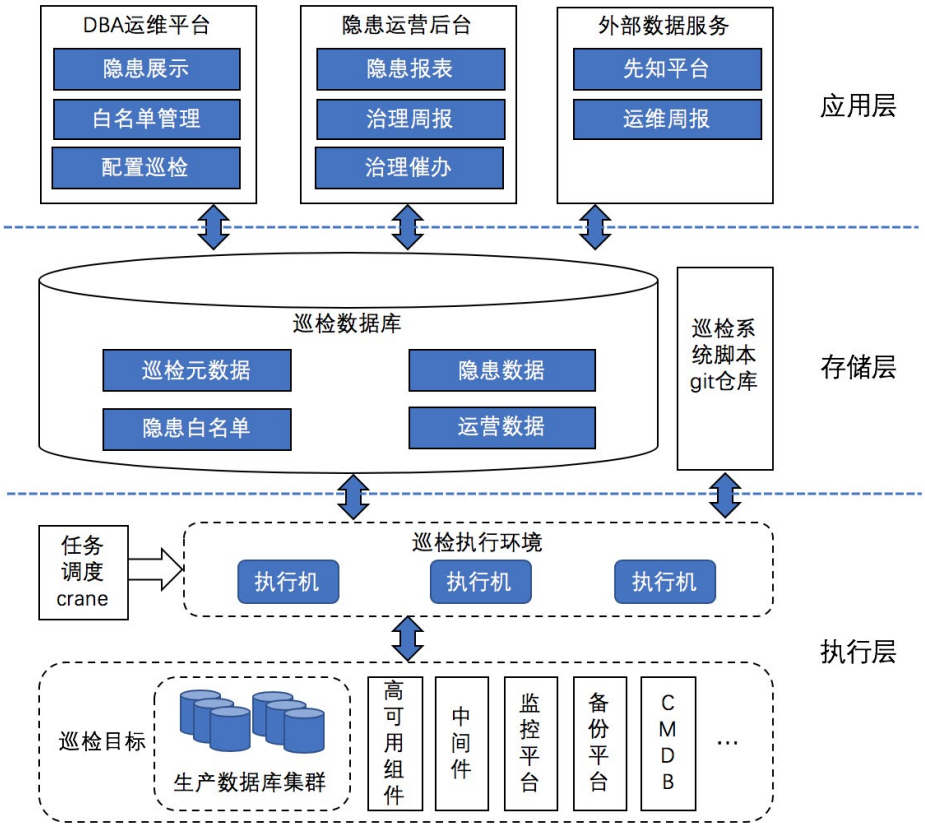


图 2　美团 MySQL 数据库巡检系统架构图

1. 执行层

巡检执行环境：由多台巡检执行机组成，巡检任务脚本会同时部署在所有执行机上。执行机会定时从巡检 Git 仓库拉取最新的脚本，脚本使用 Python Virtualenv + Git 进行管理，方便扩充新的执行机。

任务调度：巡检任务使用了美团基础架构部研发的分布式定时任务系统 Crane 进行

调度，解决传统定时任务单点问题。Crane 会随机指派某一台执行机执行任务，假如这台执行机出现故障，会指派其他执行机重新执行任务。一般一个巡检任务对应着一个巡检项，巡检任务会针对特定的巡检目标根据一定的规则来判断是否存在隐患。

巡检目标：除了对生产数据库进行巡检以外，还会对高可用组件、中间件等数据库周边产品进行巡检，尽可能覆盖所有会引发数据库故障的风险点。

2. 存储层

巡检数据库：主要用来保存巡检相关数据。为了规范和简化流程，我们将巡检发现的隐患保存到数据库中，提供了通用的入库函数，能够实现以下功能：

- 自动补齐隐患负责人、隐患发现时间等信息；
- 入库操作幂等；
- 支持半结构化的巡检结果入库，不同巡检的隐患结果包括不同的属性，比如巡检 A 的隐患有“中间件类型”，巡检 B 有“主库 CPU 核数”，以上不同结构的数据均可解析入库；
- 针对表粒度的隐患项，如果分库分表的表出现隐患，会自动合并成一个逻辑表隐患入库。

巡检脚本 Git 仓库：用来管理巡检脚本。为了方便 DBA 添加巡检，在系统建设过程中，我们增加了多个公共函数，用来降低开发新巡检的成本，也方便将老的巡检脚本迁移到新的体系中。

3. 应用层

集成到数据库运维平台：作为隐患明细展示、配置巡检展示、管理白名单等功能的入口。为了提高隐患治理效率。我们做了以下设计。

- 隐患明细展示页面会标注每个隐患出现的天数，便于追踪隐患出现原因。
- 配置新的巡检展示时必须要同时制定隐患解决方案，确保隐患治理有章可循，避免错误的治理方式导致“错上加错”。

隐患运营后台：这个模块主要目的是推进隐患的治理。

- 运营报表，帮助管理者从全局角度掌握隐患治理进展，报表包括隐患趋势、存量分布、增量分布、平均治理周期等核心内容，进而由上到下推动隐患治理；报表数据同样是通过 crane 定时任务计算获得。
- 隐患治理催办功能，用来督促 DBA 处理隐患。催办内容中会带有隐患具体内容、出现时长、处理方案等。催办形式包括大象消息、告警，具体选用哪种形式可根据巡检关键程度做相应配置。

外部数据服务：主要是将巡检隐患数据提供给美团内部其他平台或项目使用，让巡检数据发挥更大的价值。

- 对接先知平台（美团 SRE 团队开发的主要面向 RD 用户的风险发现和运营平台），平台接收各服务方上报的隐患数据，以 RD 视角从组织架构维度展示各服务的风险点，并跟进 RD 处理进度。巡检系统会把需要 RD 参与治理的隐患，比如大表、无唯一键表等，借助先知平台统一推送给 RD 进行治理。
- 运维周报，主要面向业务线 RD 负责人和业务线 DBA，以静态报告形式展示业务线数据库运行情况以及存在的问题，巡检隐患是报告内容之一。

四、巡检项目

巡检项目根据负责方分为 DBA 和 RD，DBA 主要负责处理数据库基础功能组件以及影响服务稳定性的隐患。RD 主要负责库表设计缺陷、数据库使用不规范等引起的业务故障或性能问题的隐患。也存在需要他们同时参与治理的巡检项，比如“磁盘可用空间预测”等。目前巡检项目共 64 个，类目分布情况如下图所示：

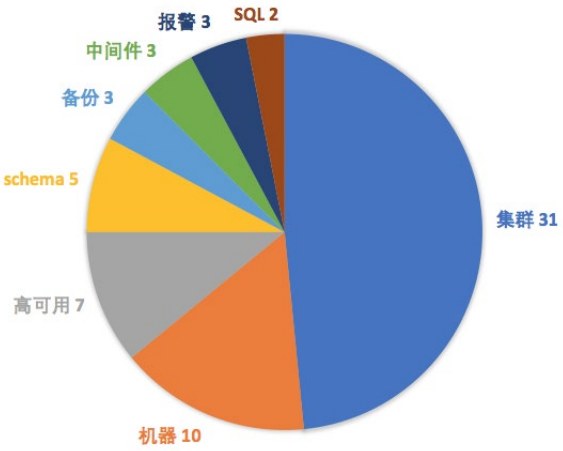


图 3　巡检项类目分布

集群：主要检查集群拓扑、核心参数等集群层面的隐患；**机器**：主要检查服务器硬件层面的隐患；**Schema/SQL**：检查表结构设计、数据库使用、SQL 质量等方面的隐患；**高可用 / 备份 / 中间件 / 报警**：主要检查相关核心功能组件是否存在隐患。

下面，我们通过列举几个巡检任务来对巡检项做简单的说明：

负责方	类目	巡检项	说明
DBA	备份	恢复测试结果检查	用来发现异常备份集。数据备份的重要性不言而喻，为了保证备份的有效性，备份系统会对每一个备份集做恢复测试，测试失败的情况会记录到巡检系统。
		Binlog无法衔接最近一次备份	通过最近一次备份和集群最早的Binlog日志情况，来判断备份和Binlog是否衔接，衔接不上会导致无法增加从库或基于时间点的数据恢复等。
	机器	磁盘可用空间预测	通过监控平台采集到的磁盘空间情况，预测磁盘空间是否会在近期使用耗尽。
	集群	集群GTID集异常	GTID即全局事务ID，在MySQL中每执行一个事务，就会分配一个GTID，并且从库会同步主库每一个GTID的操作，在没有延迟的情况下，集群所有实例的GTID集应该是一致的。由于MySQL主从复制原理，如果实例间GTID集不一致可能会导致主从切换失败，巡检通过比对每个实例的GTID集来判断是否存在异常。
		集群下多实例共宿主机	这种情况下，假如宿主机宕机，会导致集群下多台实例同时挂掉，严重影响数据库服务的稳定。虽然实例初始化时加入了相关策略主动避免这个问题，但不排除极特殊情况仍会造成这个隐患。
	高可用	集群高可用功能未开启	通常人工维护、或其他异常情况下，集群高可用功能会被关闭，巡检会检查CMDB中高可用开关来判断是否存在该隐患。
		同地域没有候选主库	这种情况下，主库宕机会提升其他地域实例作为主库，会导致写入延时增加，可能会影响业务。巡检会检查CMDB中集群实例的机房部署来判断是否存在该隐患。
RD	Schema	数值类型溢出	MySQL表的整型数据类型字段能够保存的数值是有上限的，超出最大值的数据无法写入表中。巡检系统根据一段时间采集到的整型类型字段值，判断该字段是否即将超过最大值。除了自增字段以外，还会对非自增的字段值进行预测。
		大表	表体积过大会出现性能问题，并且会增加维护成本。通过扫描所有数据库集群中表大小的情况，并结合监控平台采集到的表读写流量情况，找出问题大表。

五、成果

美团 MySQL 巡检系统已稳定运行近一年时间，基于新巡检体系上线的巡检项 49 个。通过巡检体系持续运行，在团队的共同努力下，我们共治理了 8000+ 核心隐患，近 3 个月隐患治理周期平均不超过 4 天，将隐患总数持续保持在极小的量级，有效地保障了数据库的稳定。

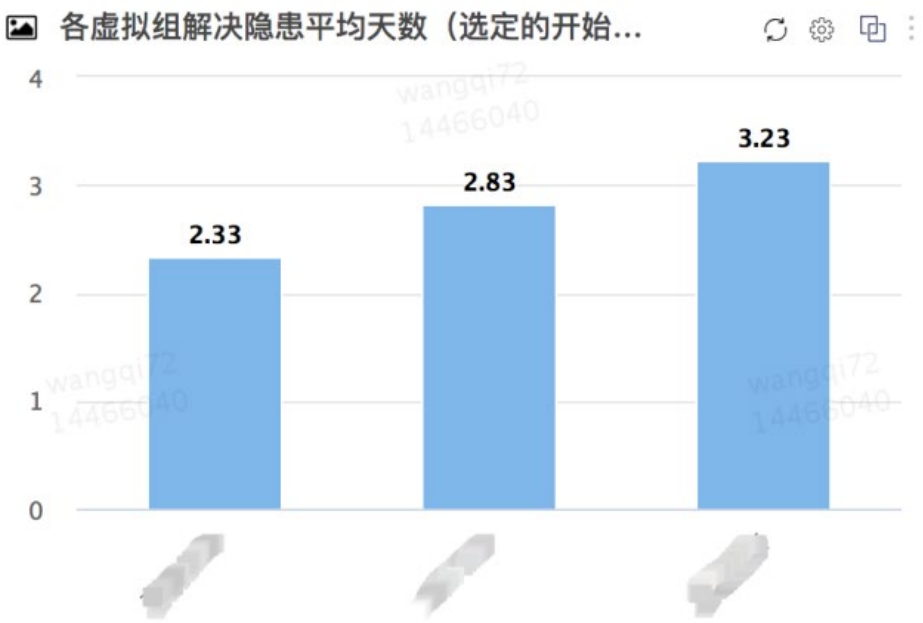


图 4　隐患运营 – 团队内各虚拟小组隐患平均治理周期

下面的隐患趋势图，展示了近一年中隐患的个数，数量突然增长是由于新的巡检项上线。从整体趋势上看，隐患存量有非常明显的下降。

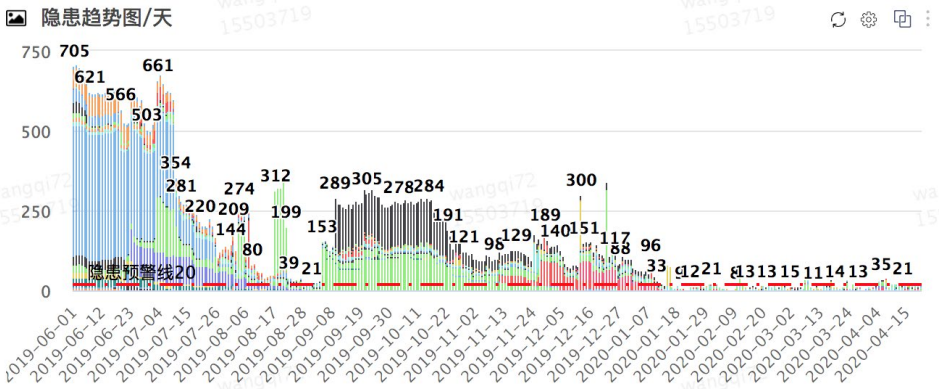


图 5　隐患运营 – 隐患总量趋势情况

除了推动内部隐患治理之外，我们还通过对接先知平台，积极推动 RD 治理隐患数量超过 5000 个。

图 6　对接先知 – 推动 RD 治理隐患

为了提升用户体验，我们在提升准确率方面也做了重点的投入，让每一个巡检在上线前都会经过严格的测试和校验。

对比其他先知接入方，DBA 上报隐患在总量、转化率、反馈率几个指标上都处于较高水平，可见我们上报的隐患风险也得到了 RD 的认可。

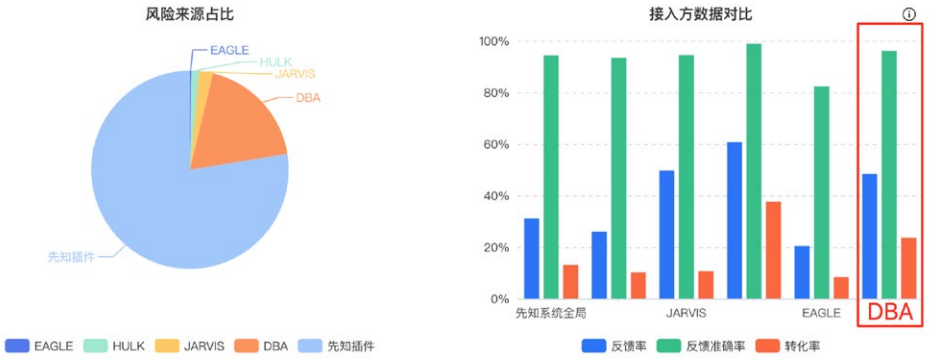


图 7　对接先知 – 各接入方上报隐患情况

指标说明：

- 反馈率 = 截止到当前时刻反馈过的风险事件数量 / 截止到当前时刻产生的风险事件总量 * 100%；
- 反馈准确率 = 截止到当前时刻反馈准确的风险事件数量 / 截止到当前时刻反馈过的风险事件总量 * 100%；
- 转化率 = 截止到当前时刻用户反馈准确且需要处理的风险事件数量 / 截止到当前时刻产生的风险事件总量 * 100%。

六、未来规划

除了继续完善补充巡检项以外，未来巡检系统还会在以下几个方向继续探索迭代：

- 提高自动化能力，完善 CI 和审计；
- 加强运营能力，进一步细化每个隐患的重要程度，辅助决策治理优先级；
- 隐患自动修复。

作者简介

王琦，基础架构部 DBA 组成员，2018 年加入美团，负责 MySQL 数据库运维 / 数据库巡检系统 / 监控 / 自动化运维周报 / 运维数据集市建设等工作。

Kubernetes 如何改变美团的云基础设施?

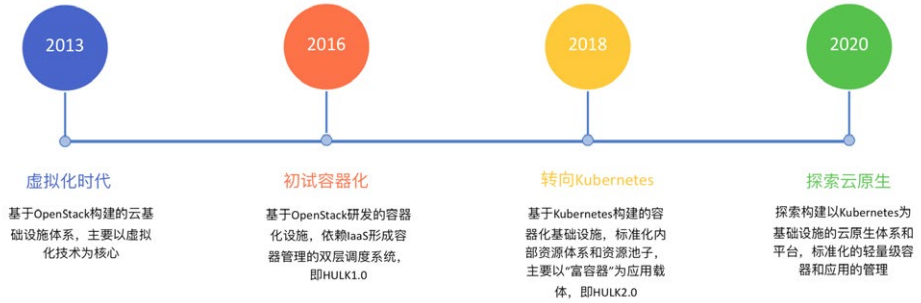
作者：王国梁

本文根据美团基础架构部王国梁在 KubeCon 2020 云原生开源峰会 Cloud Native + Open Source Virtual Summit China 2020 上的演讲内容整理而成。

一、背景与现状

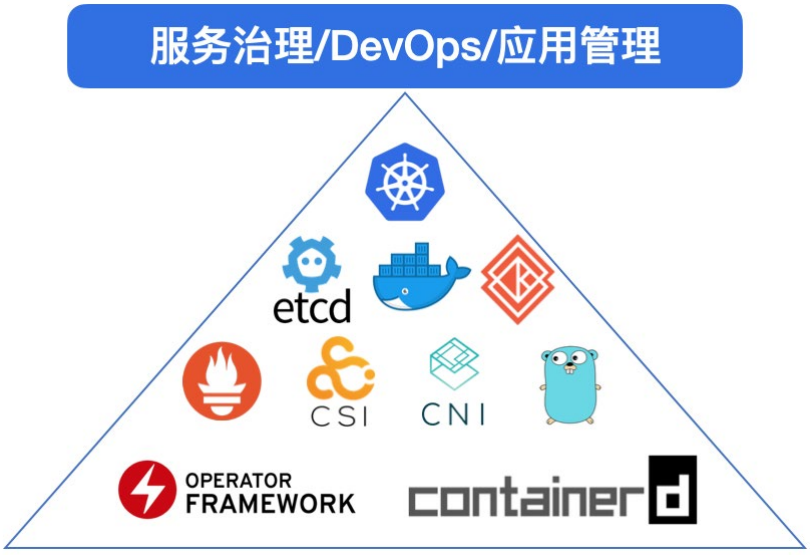
Kubernetes 是让容器应用进入大规模工业生产环境的开源系统，也是集群调度领域的事实标准，目前已被业界广泛接受并得到了大规模的应用。Kubernetes 已经成为美团云基础设施的管理引擎，它带来的不仅仅是高效的资源管理，同时也大幅降低了成本，而且为美团云原生架构的推进打下了坚实的基础，支持了 Serverless、云原生分布式数据库等一些平台完成容器化和云原生化的建设。

从 2013 年开始，美团就以虚拟化技术为核心构建了云基础设施平台；2016 年，开始探索容器技术并在内部进行落地，在原有 OpenStack 的资源管理能力之上构建了 Hulk1.0 容器平台；2018 年，美团开始打造以 Kubernetes 技术为基础的 Hulk2.0 平台；2019 年年底，我们基本完成了美团云基础设施的容器化改造；2020 年，我们坚信 Kubernetes 才是未来的云基础设施标准，又开始探索云原生架构落地和演进。

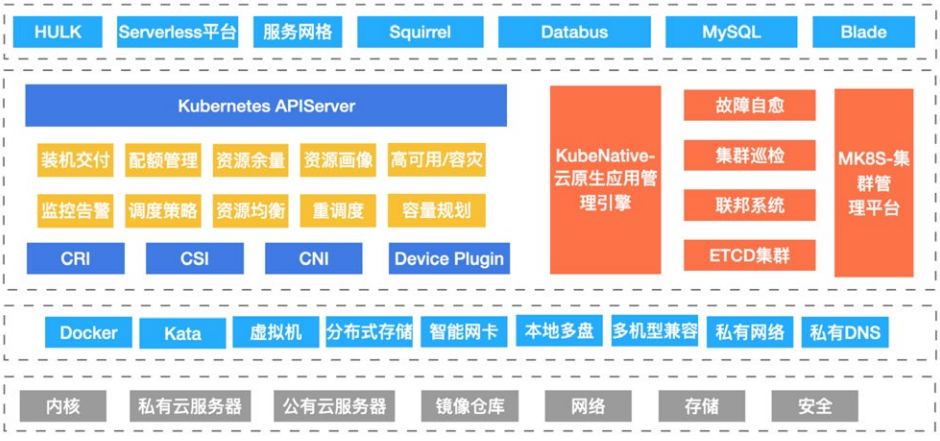


当前，我们构建了以 Kubernetes、Docker 等技术为代表的云基础设施，支持整个美团的服务和应用管理，容器化率达到 98% 以上，目前已有数十个大小 Kuberne-

tes 集群，数万的管理节点以及几十万的 Pod。不过出于容灾考虑，我们最大单集群设置为 5K 个节点。



下图是当前我们基于 Kubrnetes 引擎的调度系统架构，构建了以 Kubernetes 为核心的统一的资源管理系统，服务于各个 PaaS 平台和业务。除了直接支持 Hulk 容器化之外，也直接支持了 Serverless、Blade 等平台，实现了 PaaS 平台的容器化和云原生化。



二、OpenStack 到 Kubernetes 转变的障碍和收益

对于一个技术栈比较成熟的公司而言，整个基础设施的转变并不是一帆风顺的，在 OpenStack 云平台时期，我们面临的主要问题包括以下几个方面：

1. 架构复杂，运维和维护比较困难：OpenStack 的整个架构中计算资源的管理模块是非常庞大和复杂，问题排查和可靠性一直是很大的问题。
2. 环境不一致问题突出：环境不一致问题是容器镜像出现之前业界的通用问题，不利于业务的快速上线和稳定性。
3. 虚拟化本身资源占用多：虚拟化本身大概占用 10% 的宿主机资源消耗，在集群规模足够大的时候，这是一块非常大的资源浪费。
4. 资源交付和回收周期长，不易灵活调配：一方面是整个虚拟机创建流程冗长；另一方面各种初始化和配置资源准备耗时长且容易出错，所以就导致整个机器资源从申请到交付周期长，快速的资源调配是个难题。
5. 高低峰明显，资源浪费严重：随着移动互联网的高速发展，公司业务出现高低峰的时间越来越多，为了保障服务稳定不得不按照最高的资源需求来准备资源，这就导致低峰时资源空闲严重，进而造成浪费。

2.1 容器化的过程和障碍

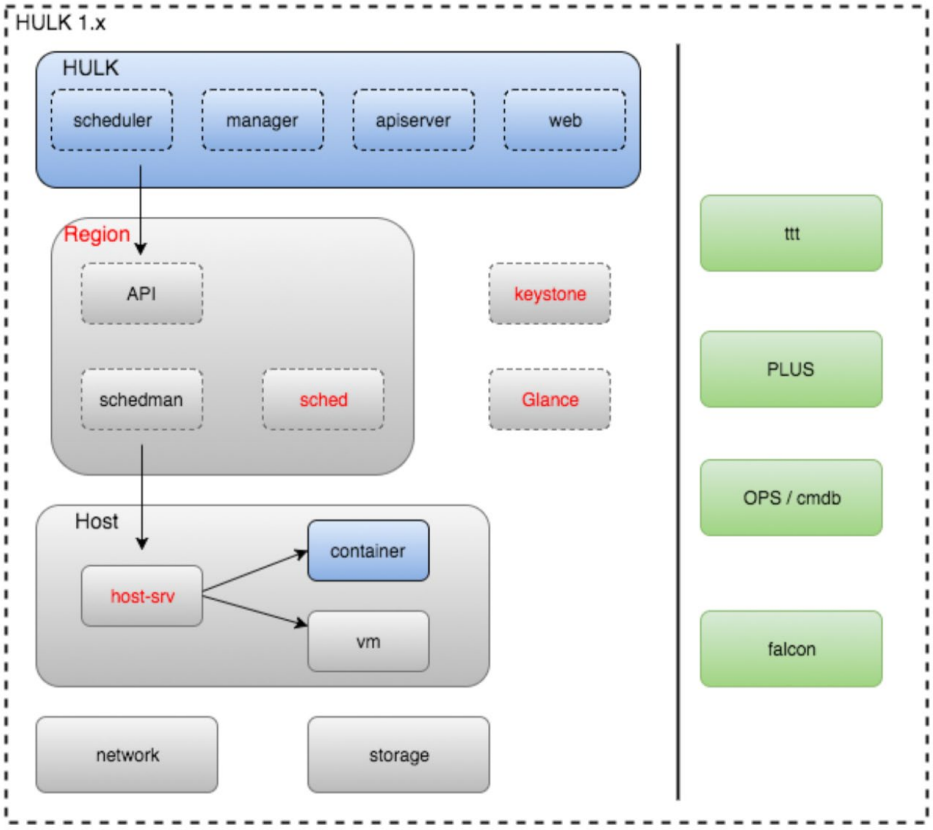
为了解决虚拟机存在的问题，美团开始探索更加轻量级的容器技术的落地，也就是 Hulk1.0 项目。不过基于当时的资源环境和架构，Hulk1.0 是以原有的 OpenStack 为基础资源管理层实现的容器平台，OpenStack 提供底层的宿主机的资源管理能力，解决了业务对弹性资源的需求，并且整个资源交付周期从分钟级别降低到了秒级。

但是，随着 Hulk1.0 的推广和落地，也暴露出一些新的问题：

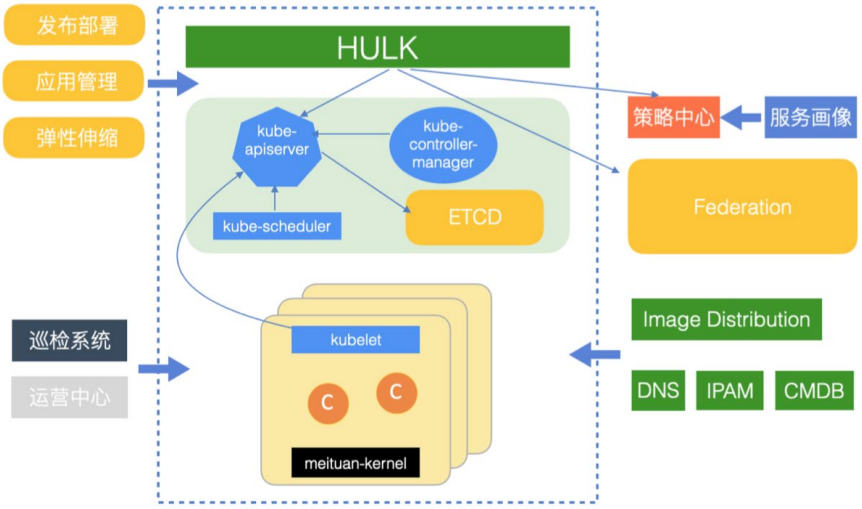
1. 稳定性差：因为复用了 OpenStack 的底层资源管理能力，整个扩容过程包括两层的资源调度，且数据同步流程复杂，机房的隔离性也比较差，经常出现一个机房出现问题，其他机房的扩缩容也受到影响。
2. 能力欠缺：由于涉及的系统多，并且是跨部门协作，故障节点的迁移和恢复能

力不易实现，资源类型也比较单一，整个故障排查和沟通效率低下。

3. 扩展性差：Hulk1.0 的控制层面对底层资源的管理能力受限，无法根据场景和需求快速扩展。
4. 性能：业务对于扩缩容和弹性资源的交付速度需求进一步提高，且容器技术的弱隔离性导致业务的服务受到的干扰增多，负面反馈增加。



上述的问题经过一段时间的优化和改善，始终不能彻底解决。在这种情况下，我们不得不重新思考整个容器平台的架构合理性，而此时 Kubernetes 已逐步被业界认可和应用，它清晰的架构和先进的设计思路让我们看到了希望。所以我们基于 Kubernetes 构建了新的容器平台，在新的平台中 Hulk 完全基于原生的 Kubernetes API，通过 Hulk API 来对接内部的发布部署系统，这样两层 API 将整个架构解耦开来，领域明确，应用管理和资源管理可以独立迭代，Kubernetes 强大的编排和资源管理能力凸显。



容器化的核心思路是让 Kubernetes 做好资源层面的管理，而通过上层的控制层解决对美团应用管理系统和运维系统的依赖问题，保持 Kubernetes 的原生兼容性，减少后续的维护成本，并完成了快速收敛资源管理的需求。同时，也减少了用户基于新平台的资源申请的学习成本，这点非常重要，也是后续我们能快速大规模迁移基础设施资源的“基础”。

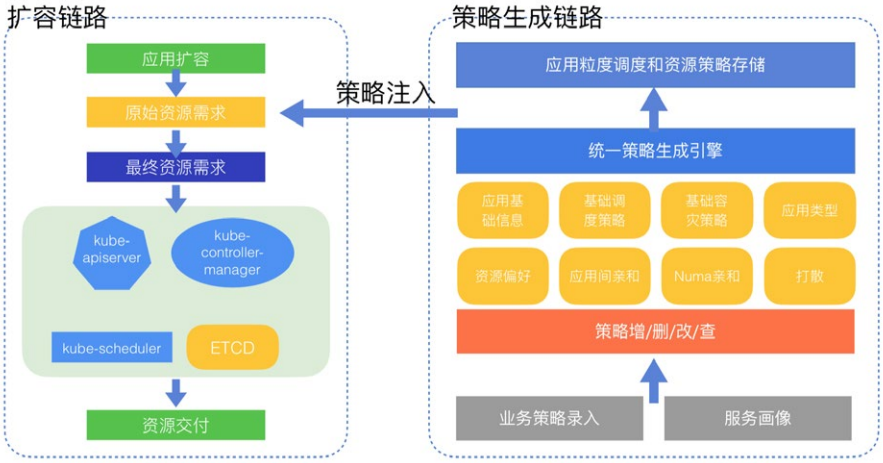
2.2 容器化过程的挑战和应对策略

2.2.1 复杂灵活、动态和可配置的调度策略

美团产品众多，业务线和应用特点五花八门，所以相应的，我们对于资源类型和调度策略的需求也是非常多。例如，有些业务需要特定的资源类型（SSD、高内存、高 IO 等等），有些业务需要特定的打散策略（例如机房、服务依赖等），所以如何很好地应对这些多样化的需求，就是一个很大的问题。

为了解决这些问题，我们为扩容链路增加了策略引擎，业务可以对自己的应用 APPKEY 自定义录入策略需求，同时基于大数据分析的服务画像，也会根据业务特点和公司的应用管理策略为业务策略推荐，最终这些策略会保存到策略中心。在扩容过程中，我们会自动为应用的实例打上对应的需求标签，并最终在 Kubenretes 中生

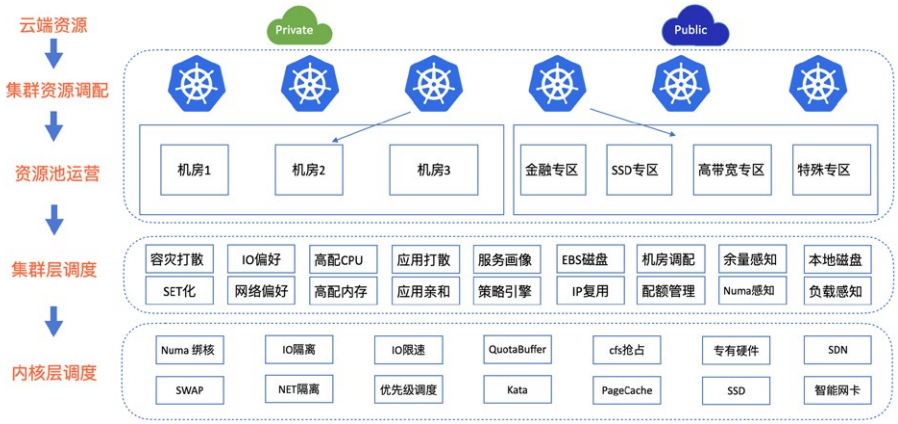
效，完成预期的资源交付。



2.2.2 精细化的资源调度和运营

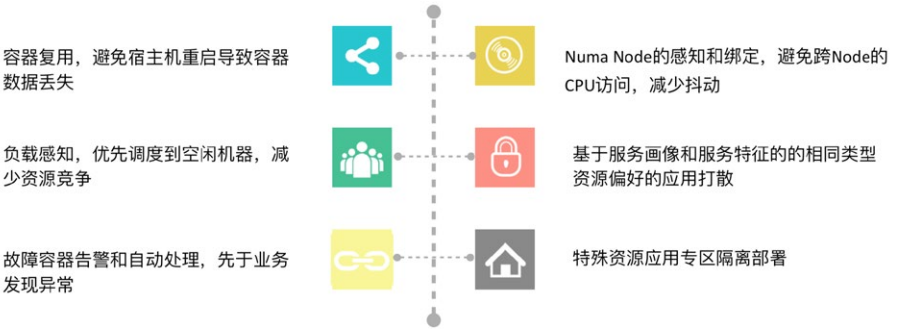
精细化的资源调度和运营，之所以做精细化运营主要是出于两点考虑：业务的资源需求场景复杂，以及资源不足的情况较多。

我们依托私有云和公有云资源，部署多个 Kubenretes 集群，这些集群有些是承载通用业务，有些是为特定应用专有的集群，在集群维度对云端资源进行调配，包括机房的划分、机型的区分等。在集群之下，我们又根据不同的业务需要，建设不同业务类型的专区，以便做到资源池的隔离来应对业务的需要。更细的维度，我们针对应用层面的资源需求、容灾需求以及稳定性等做集群层的资源调度，最后基于底层不同硬件以及软件，实现 CPU、MEM 和磁盘等更细粒度的资源隔离和调度。



2.2.3 应用稳定性的提升和治理

不管是 VM，还是最初的容器平台，在应用稳定性方面一直都存在问题。为此，我们需要在保障应用的 SLA 上做出更多的努力。



2.2.3.1 容器复用

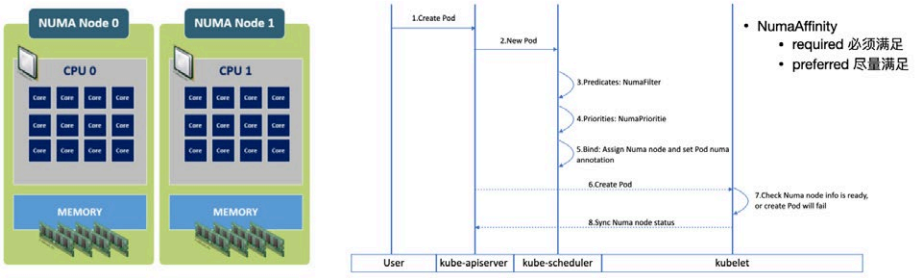
在生产环境中，宿主机的发生重启是一种非常常见的场景，可能是主动重启也可能是被动，但用户角度来看，宿主机重启意味着用户的一些系统数据就可能丢失，代价还是比较高的。我们需要避免容器的迁移或重建，直接重启恢复。但我们都知道，在 Kubernetes 中，对于 Pod 中的容器的重启策略有以下几种：Always、OnFailure 和 Never，宿主机重启后容器会重新被创建。

为了解决这个问题，我们为容器的重启策略类型增加了 Reuse 策略。流程如下：

1. kubelet 在 SyncPod 时，重启策略如果是 Reuse 则会获取对应 Pod 已退出状态的 App 容器，如果存在则拉起最新的 App 容器（可能有多个），如果不存在则直接新建。
2. 更新 App 容器映射的 pauseID 为新的 pause 容器 ID，这样就建立了 Pod 下新的 pause 容器和原先 App 容器的映射。
3. 重新拉起 App 容器即可完成 Pod 状态同步，最终即使宿主机重启或内核升级，容器数据也不会丢失。

2.2.3.2 Numa 感知与绑定

用户的另一个痛点与容器性能和稳定性相关。我们不断收到业务反馈，同样配置的容器性能存在不小的差异，主要表现为部分容器请求延迟很高，经过我们测试和深入分析发现：这些容器存在跨 Numa Node 访问 CPU，在我们将容器的 CPU 使用限制在同一个 Numa Node 后问题消失。所以，对于一些延迟敏感型的业务，我们要保证应用性能表现的一致性和稳定性，需要做到在调度侧感知 Numa Node 的使用情况。



为了解决这个问题，我们在 Node 层采集了 Numa Node 的分配情况，在调度器层

增加了对 Numa Node 的感知和调度，并保证资源使用的均衡性。对于一些强制需要绑定 Node 的敏感型应用，如果找不到合适的 Node 则扩容失败；对于一些不需要绑定 Numa Node 的应用，则可以选择尽量满足的策略。

2.2.3.3 其他稳定性优化

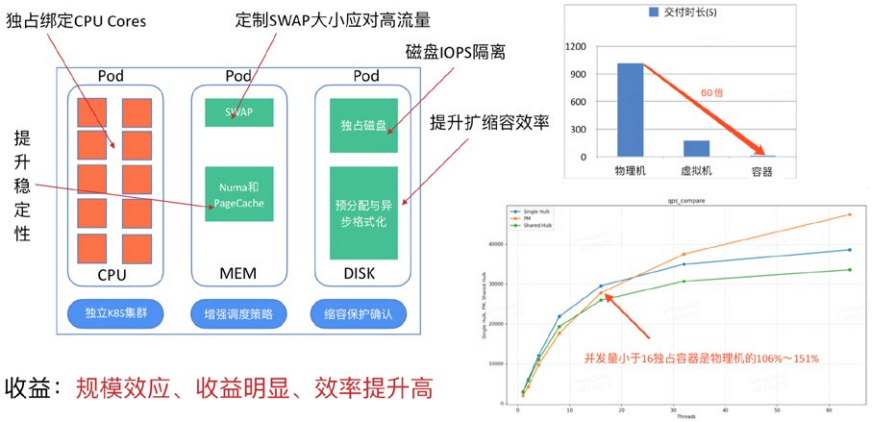
1. 在调度层面，我们为调度器增加了负载感知和基于服务画像应用特征的打散和优选策略。
2. 在故障容器发现和处理上，基于特征库落地的告警自愈组件，能够秒级发现 – 分析 – 处理告警。
3. 对于一些有特殊资源需求，例如高 IO、高内存等应用采用专区隔离，避免对其他应用造成影响。

2.2.4 平台型业务容器化

相信做过 ToB 业务的同学应该都了解，任何产品都存在大客户方案，那么对于美团这样的公司，内部也会存在这种情况。平台型业务的容器化有个特点是：实例数多，以千或万计，所以资源成本就比较高；业务地位比较高，一般都是非常核心的业务，对性能和稳定性要求很高。所以，如果想要通过“一招鲜”的方式解决此类业务的问题，就有些不切实际。

这里，我们以 MySQL 平台为例，数据库业务对于稳定性、性能和可靠性要求非常高，业务自己又主要以物理机为主，所以成本压力非常大。针对数据库的容器化，我们主要是从宿主机端的资源分配定制和优化为切入口。

1. 针对 CPU 资源分配，采用独占 CPU 集合的方式，避免 Pod 之间发生争抢。
2. 通过允许自定义 SWAP 大小来应对短暂的高流量，并关闭 Numa Node 和 PageCache 来提升稳定性。
3. 在磁盘分配中采用 Pod 独占磁盘进行 IOPS 的隔离，以及通过预分配和格式化磁盘来提升扩容的速度，提升资源交付效率。
4. 调度支持独有的打散策略和缩容确认，规避缩容风险。



最终，我们将数据库的交付效率提升了 60 倍，并且在大多数情况下性能比之前的物理机器还要好。

2.2.5 业务资源优先级保障

对于一个企业而言，基于成本考虑，资源一直会处于不足的状态，那么如何保障资源的供给和分配就显得非常重要。

1. 业务预算配额确定资源供给，通过专区来做专有资源专用。
2. 建设弹性资源池和打通公有云来应对突发资源需求。
3. 按照业务和应用类型的优先级保障资源使用，确保核心业务先拿到资源。
4. 多个 Kubenretes 集群和多机房来做容灾，应对集群或机房的故障。

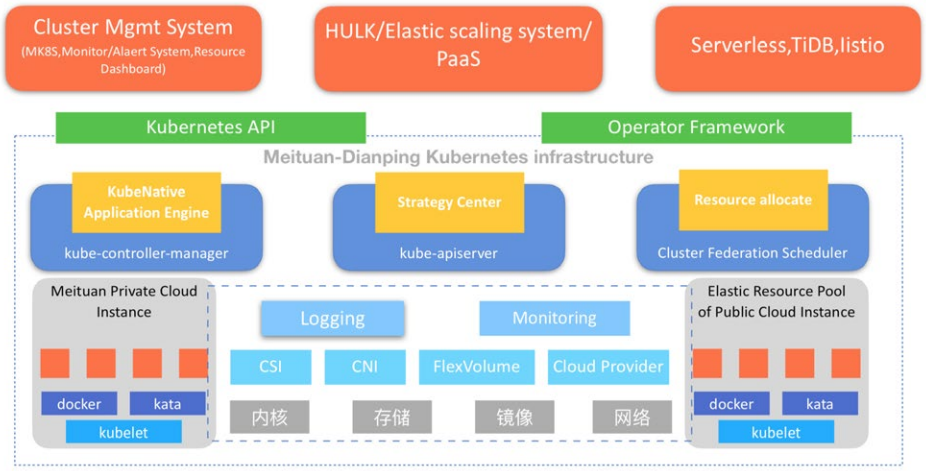
2.2.6 云原生架构的落地

在迁移到 Kubernetes 之后，我们进一步实现了云原生架构的落地。

为了解决云原生应用管理的障碍，我们设计实现了美团特色的云原生应用管理引擎——KubeNative，将应用的配置和信息管理对平台透明化，业务平台只需要创建原生的 Pod 资源即可，不需要关注应用的信息同步和管理细节，并支持各 PaaS 平台自己来扩展控制层面的能力，运行自己的 Operator。

下图就是目前我们整个的云原生应用管理架构，已支持 Hulk 容器平台、Serverless

以及 TiDB 等平台的落地。



2.3 基础设施迁移后的收益

1. 完成全公司业务 98% 的容器化，显著提升了资源管理的效率和业务稳定性。
2. Kubernetes 稳定性 99.99% 以上。
3. Kubernetes 成为美团内部集群管理平台的标准。

三、运营大规模 Kubernetes 集群的挑战和应对策略

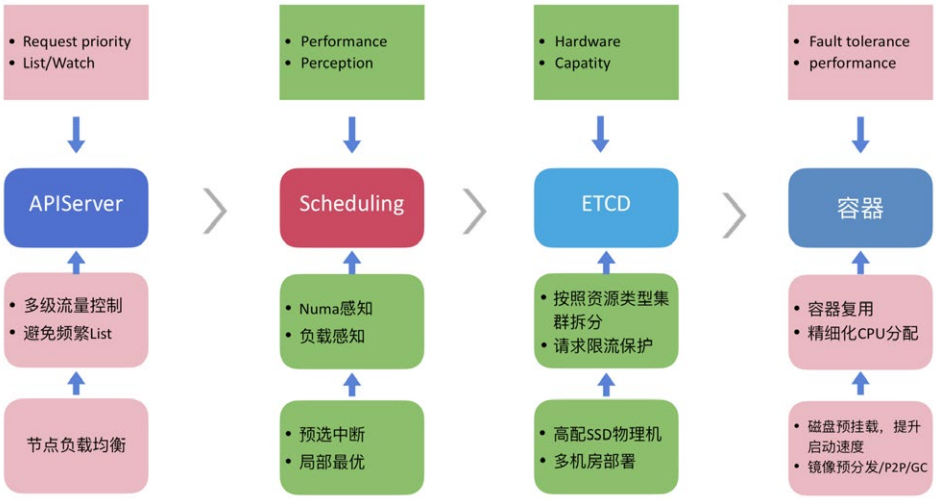
在整个基础设施迁移过程中，除了解决历史遗留问题和系统建设，随着 Kubernetes 集群规模和数量快速增长，我们遇到的新的挑战是：如何稳定、高效地运营大规模 Kubernetes 集群。我们在这几年的 Kubernetes 运营中，也逐渐摸索出了一套验证可行的运营经验。

3.1 核心组件优化与升级

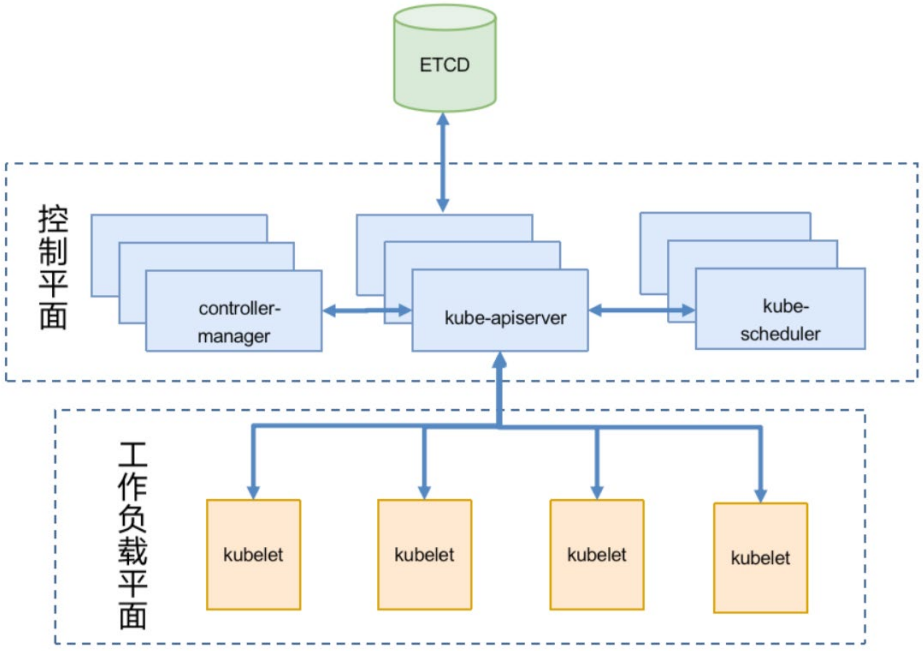
我们最初使用的 Kubernetes 是 1.6 版本，性能和稳定性是比较差的，当我们达到 1K 节点的时候就逐渐出现问题，达到 5K 节点时基本集群不可用。例如，调度性能非常差，集群吞吐量也比较低，偶尔还发生“雪崩”的情况，扩缩容链路耗时也在变长。

针对核心组件的分析和优化，这里从 kube-apiserver、kube-scheduler、etcd 以及容器等四个方面来概括下。

1. 针对 kube-apiserver，为了减少重启过程长时间地发生 429 请求重试，我们实现了多级的流量控制，将不可用窗口从 15min 降低为 1min，并通过减少和避免外部系统的 List 操作降低集群负载，通过内部的 VIP 来做节点的负载均衡，保障控制节点的稳定性。
2. 在 kube-scheduler 层，我们增强了调度的感知策略，调度效果相比之前更稳定；对调度性能的优化提出的预选中断和局部最优策略也已合并到社区，并成为通用的策略。
3. 针对 etcd 的运营，通过拆分出独立的 Event 集群降低主库的压力，并且基于高配的 SSD 物理机器部署可以达到日常 5 倍的高流量访问。
4. 在容器层面，容器复用提升了容器的故障容忍能力，并通过精细化的 CPU 分配提升应用稳定性；通过容器的磁盘预挂载提升 Node 的故障恢复速度。

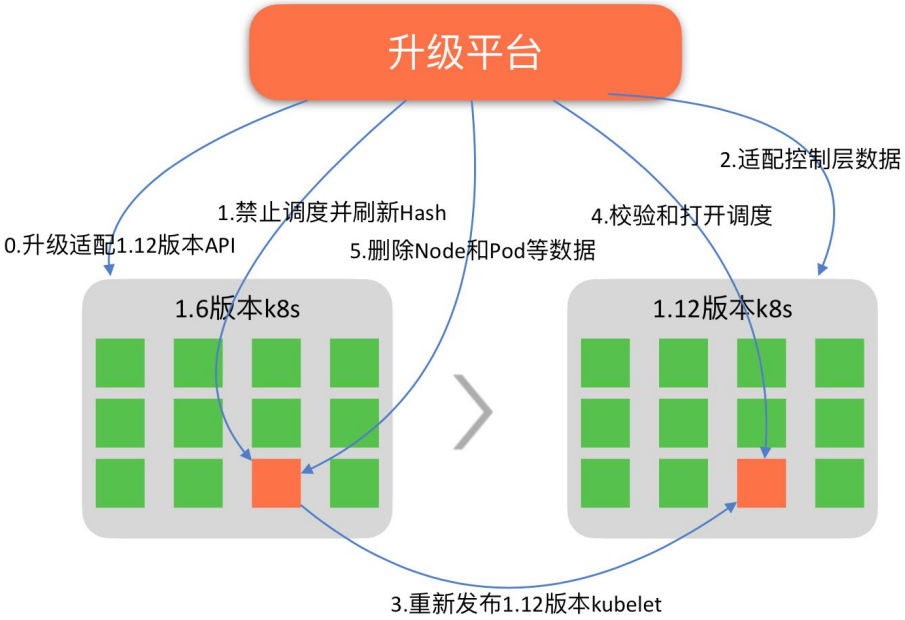


另外，社区版本的迭代是非常快的，高版本在稳定性和特性支持上更好，不可避免我们需要进行版本的升级，但如何确保升级成功是一个很大的挑战，尤其是我们在没有足够的 Buffer 资源进行资源腾挪情况下。



集群升级，业界通用的方案是直接基于原有集群升级，方案存在以下几点问题：

1. 升级版本有限制，不能跨大版本升级：只能一点点从低版本升级到高版本，耗时费力，而且成功率低。
2. 控制平面升级风险不可控：尤其是有 API 变更的时候，会覆盖之前的数据，甚至是不可回滚的。
3. 用户有感知，容器需要新建，成本和影响较高：这个是比较痛的点，无可避免会发生容器新建。



为此，我们深入研究了 Kubernetes 对容器层面的控制方式，设计实现了一种能够平滑将容器数据从低版本集群迁移到高版本集群的方案，将集群升级细化为 Node 粒度的逐个宿主机上容器的原地热升级，随时可以暂停和回滚。新方案主要是通过外部工具将 Node 和 Pod 数据从低版本集群迁移到高版本集群，并解决 Pod 对象和容器的兼容性问题。核心思路是两点：通过低版本兼容高版本的 API，通过刷新容器的 Hash 保障 Pod 下的容器不会被新；通过工具实现 Pod 和 Node 资源数据从低版本集群迁移到高版本集群。

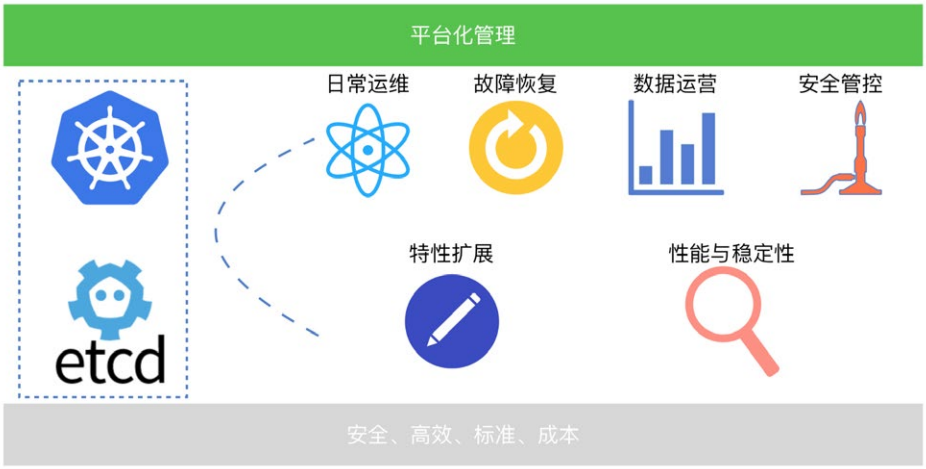
该方案亮点主要包括以下 4 个方面：

1. 大规模生产环境的集群升级不再是难题。
2. 解决了现有技术方案风险不可控的问题，风险降到了宿主机级别，升级更为安全。
3. 通用性强，可做到任意版本的升级，且方案生命周期长。
4. 优雅地解决了升级过程中容器新建问题，真正做到了原地热升级。

3.2 平台化与运营效率

大规模的集群运营是非常有挑战的事情，满足业务的快速发展和用户需求也是对团队极大的考验，我们需要从不同纬度的考虑集群的运营和研发能力。

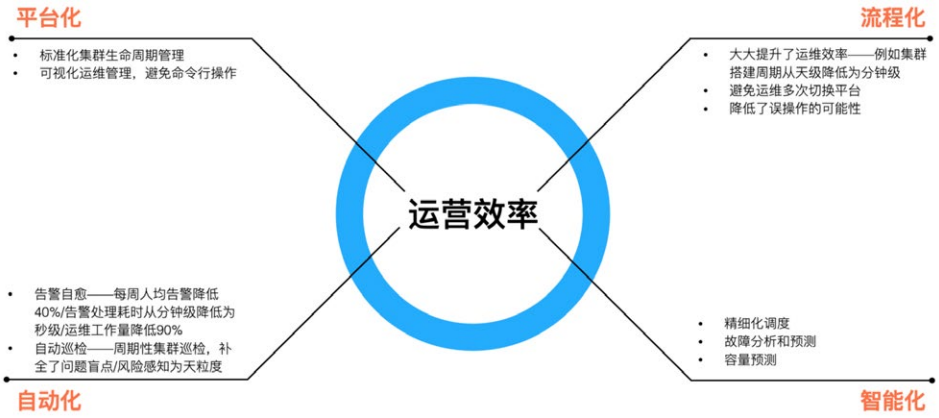
在 Kubernetes 与 etcd 集群的整个运营和运维能力建设上，我们关注的目标是安全运营、高效运维、标准化管理以及节约成本。所以针对 Kubernetes 与 etcd 集群，我们已经完成了平台化的管理运营，覆盖了特性扩展、性能与稳定性、日常运维、故障恢复、数据运营以及安全管控等 6 个方面。



对于一个非公有云业务的 Kubernetes 团队，人力还是非常有限的，除了集群的日常运营还有研发任务，所以我们对于运营效率的提升非常关注。我们将日常运维逐步的沉淀转换，构建了一套美团内部的 Kubernetes 集群管理平台。

1. 将集群的管理标准化、可视化，避免了黑白屏的运维操作。
2. 通过告警自愈和自动巡检将问题处理收敛掉，所以虽然我们有大几十个集群，但我们的运维效率还是比较高的，值班同学很少需要关注。
3. 全部的运维操作流程化，不仅提升了运维效率，人为操作导致的故障的概率也减小了。
4. 通过运营数据的分析进一步做了资源的精细化调度和故障预测，进一步提前

发现风险，提升了运营的质量。

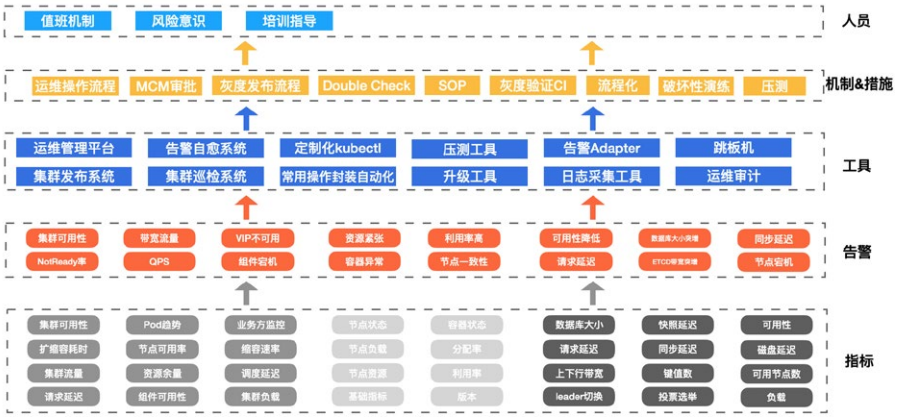


3.3 风险控制和可靠性保障

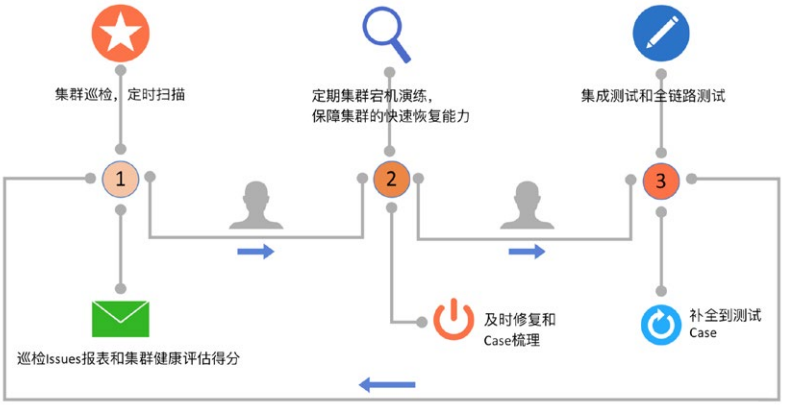
规模大、覆盖业务广，任何的集群故障都会直接影响到服务的稳定性甚至用户的体验，在经历了多次运维故障和安全压力下，我们形成了一套可复制的风险控制和可靠性保障策略。

在整个风险管控链路中，我们分为指标、告警、工具、机制 & 措施和人员 5 个层面：

1. 指标数据采集，从节点、集群、组件以及资源层面采集核心指标作为数据源。
2. 风险推送，覆盖核心指标的多级、多维度的告警机制。
3. 在工具支持上，通过主动、被动以及流程化等减少误操作风险。
4. 机制保障上，打通测试、灰度验证、发布确认以及演练等降低疏忽大意的情况。
5. 人是风险的根本，这块我们一直也在努力建设和轮值，确保问题的响应。



在可靠性验证和运营方面，我们笃信需要把功夫用在评审，通过集群巡检来评估集群的健康情况，并推送报表；定期的宕机演练保障真实故障能够快速恢复，并将日常问题补全到全链路测试中，形成闭环。



四、总结与未来展望

4.1 经验心得

1. Kubernetes 的落地完全兼容社区的 Kubernetes API；只会做插件化的扩展，并尽量不改控制层面的原有行为。
2. 对社区的一些特性，取长补短，并且有预期的升级，不盲目升级和跟进社区版本，尽量保持每年度的一个核心稳定版本。

3. 落地以用户痛点为突破口，业务是比较实际的，为什么需要进行迁移？业务会怕麻烦、不配合，所以推进要找到业务痛点，从帮助业务的角度出发，效果就会不一样。
4. 内部的集群管理运营的价值展现也是很重要的一环，让用户看到价值，业务看到潜在的收益，他们会主动来找你。

在容器时代，不能只看 Ku1. bernetes 本身，对于企业内的基础设施，“向上”和“向下”的融合和兼容问题也很关键。“向上”是面向业务场景为用户提供对接，因为容器并不能直接服务于业务，它还涉及到如何部署应用、服务治理、调度等诸多层面。“向下”，即容器与基础设施相结合的问题，这里更多的是兼容资源类型、更强大的隔离性、更高的资源使用效率等都是关键问题。

4.2 未来展望

1. 统一调度：VM 会少量长期存在一段时间，但如果同时维护两套基础设施产品成本是非常高的，所以我们也在落地 Kubernetes 来统一管理 VM 和容器。
2. VPA：探索通过 VPA 来进一步提升整个资源的使用效率。
3. 云原生应用管理：当前，我们已将云原生应用管理在生产环境落地，未来我们会进一步扩大云原生应用的覆盖面，不断提升研发效率。
4. 云原生架构落地：推进各个中间件、存储系统、大数据以及搜索业务合作落地各个领域的云原生系统。

作者介绍

国梁，美团点评技术专家，现负责美团点评 Kubernetes 集群的整体运营和维护以及云原生技术落地支持。

招聘信息

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

基本功 | Java 即时编译器原理解析及实践

作者：昊天　珩智　薛超

一、导读

常见的编译型语言如 C++，通常会把代码直接编译成 CPU 所能理解的机器码来运行。而 Java 为了实现“一次编译，处处运行”的特性，把编译的过程分成两部分，首先它会先由 javac 编译成通用的中间形式——字节码，然后再由解释器逐条将字节码解释为机器码来执行。所以在性能上，Java 通常不如 C++ 这类编译型语言。

为了优化 Java 的性能，JVM 在解释器之外引入了即时（Just In Time）编译器：当程序运行时，解释器首先发挥作用，代码可以直接执行。随着时间推移，即时编译器逐渐发挥作用，把越来越多的代码编译优化成本地代码，来获取更高的执行效率。解释器这时可以作为编译运行的降级手段，在一些不可靠的编译优化出现问题时，再切换回解释执行，保证程序可以正常运行。

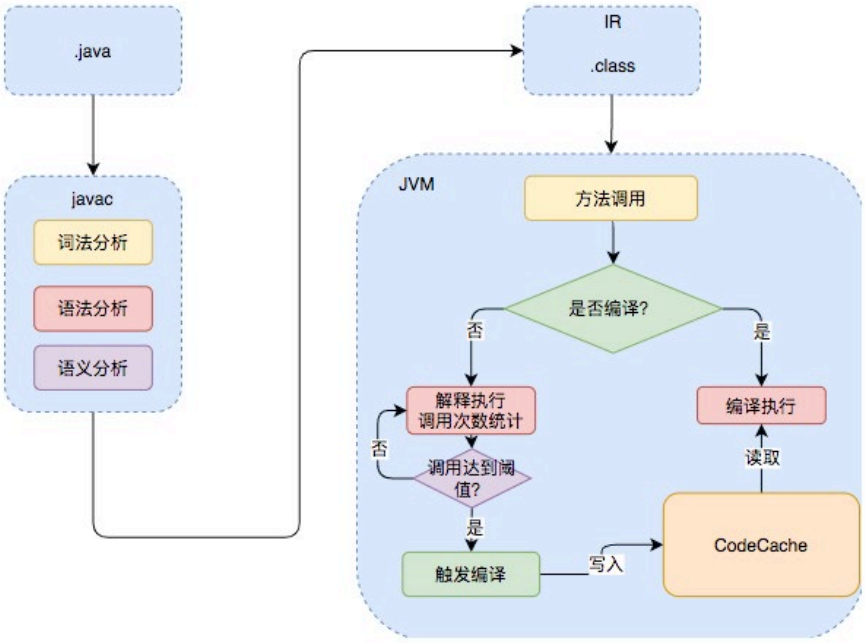
即时编译器极大地提高了 Java 程序的运行速度，而且跟静态编译相比，即时编译器可以选择性地编译热点代码，省去了很多编译时间，也节省很多的空间。目前，即时编译器已经非常成熟了，在性能层面甚至可以和编译型语言相比。不过在这个领域，大家依然在不断探索如何结合不同的编译方式，使用更加智能的手段来提升程序的运行速度。

二、Java 的执行过程

Java 的执行过程整体可以分为两个部分，第一步由 javac 将源码编译成字节码，在这个过程中会进行词法分析、语法分析、语义分析，编译原理中这部分的编译称为前端编译。接下来无需编译直接逐条将字节码解释执行，在解释执行的过程中，虚拟机同时对程序运行的信息进行收集，在这些信息的基础上，编译器会逐渐发挥作用，它

会进行后端编译——把字节码编译成机器码，但不是所有的代码都会被编译，只有被 JVM 认定为的热点代码，才可能被编译。

怎么样才会被认为是热点代码呢？ JVM 中会设置一个阈值，当方法或者代码块的在一定时间内的调用次数超过这个阈值时就会被编译，存入 codeCache 中。当下次执行时，再遇到这段代码，就会从 codeCache 中读取机器码，直接执行，以此来提升程序运行的性能。整体的执行过程大致如下图所示：



1. JVM 中的编译器

JVM 中集成了两种编译器，Client Compiler 和 Server Compiler，它们的作用也不同。Client Compiler 注重启动速度和局部的优化，Server Compiler 则更加关注全局的优化，性能会更好，但由于会进行更多的全局分析，所以启动速度会变慢。两种编译器有着不同的应用场景，在虚拟机中同时发挥作用。

Client Compiler

HotSpot VM 带有一个 Client Compiler C1 编译器。这种编译器启动速度快，但是性能比较 Server Compiler 来说会差一些。C1 会做三件事：

- 局部简单可靠的优化，比如字节码上进行的一些基础优化，方法内联、常量传播等，放弃许多耗时较长的全局优化。
- 将字节码构造成高级中间表示（High-level Intermediate Representation，以下称为 HIR），HIR 与平台无关，通常采用图结构，更适合 JVM 对程序进行优化。
- 最后将 HIR 转换成低级中间表示（Low-level Intermediate Representation，以下称为 LIR），在 LIR 的基础上会进行寄存器分配、窥孔优化（局部的优化方式，编译器在一个基本块或者多个基本块中，针对已经生成的代码，结合 CPU 自己指令的特点，通过一些认为可能带来性能提升的转换规则或者通过整体的分析，进行指令转换，来提升代码性能）等操作，最终生成机器码。

Server Compiler

Server Compiler 主要关注一些编译耗时较长的全局优化，甚至会还会根据程序运行的信息进行一些不可靠的激进优化。这种编译器的启动时间长，适用于长时间运行的后台程序，它的性能通常比 Client Compiler 高 30% 以上。目前，Hotspot 虚拟机中使用的 Server Compiler 有两种：C2 和 Graal。

C2 Compiler

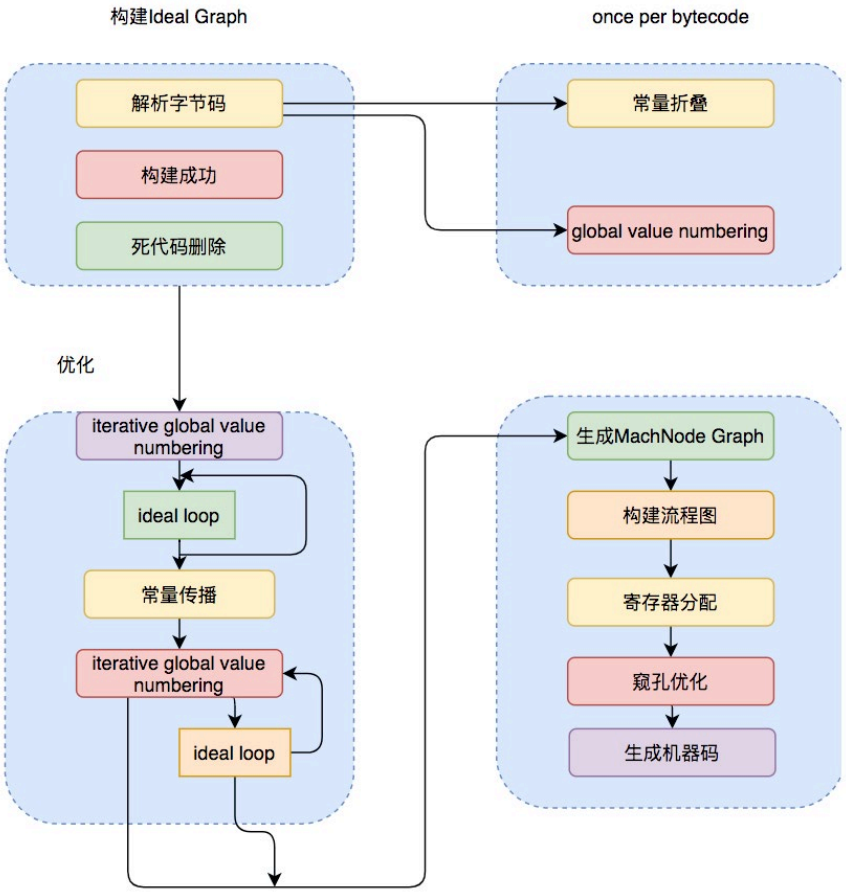
在 Hotspot VM 中，默认的 Server Compiler 是 C2 编译器。

C2 编译器在进行编译优化时，会使用一种控制流与数据流结合的图数据结构，称为 Ideal Graph。 Ideal Graph 表示当前程序的数据流向和指令间的依赖关系，依靠这种图结构，某些优化步骤（尤其是涉及浮动代码块的那些优化步骤）变得不那么复杂。

Ideal Graph 的构建是在解析字节码的时候，根据字节码中的指令向一个空的 Graph

中添加节点，Graph 中的节点通常对应一个指令块，每个指令块包含多条相关联的指令，JVM 会利用一些优化技术对这些指令进行优化，比如 Global Value Numbering、常量折叠等，解析结束后，还会进行一些死代码剔除的操作。生成 Ideal Graph 后，会在这个基础上结合收集的程序运行信息来进行一些全局的优化，这个阶段如果 JVM 判断此时没有全局优化的必要，就会跳过这部分优化。

无论是否进行全局优化，Ideal Graph 都会被转化为一种更接近机器层面的 MachNode Graph，最后编译的机器码就是从 MachNode Graph 中得的，生成机器码前还会有一些包括寄存器分配、窥孔优化等操作。关于 Ideal Graph 和各种全局的优化手段会在后面的章节详细介绍。Server Compiler 编译优化的过程如下图所示：



Graal Compiler

从 JDK 9 开始，Hotspot VM 中集成了一种新的 Server Compiler，Graal 编译器。相比 C2 编译器，Graal 有这样几种关键特性：

- 前文有提到，JVM 会在解释执行的时候收集程序运行的各种信息，然后编译器会根据这些信息进行一些基于预测的激进优化，比如分支预测，根据程序不同分支的运行概率，选择性地编译一些概率较大的分支。Graal 比 C2 更加青睐这种优化，所以 Graal 的峰值性能通常要比 C2 更好。
- 使用 Java 编写，对于 Java 语言，尤其是新特性，比如 Lambda、Stream 等更加友好。
- 更深层次的优化，比如虚函数的内联、部分逃逸分析等。

Graal 编译器可以通过 Java 虚拟机参数 -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 启用。当启用时，它将替换掉 HotSpot 中的 C2 编译器，并响应原本由 C2 负责的编译请求。

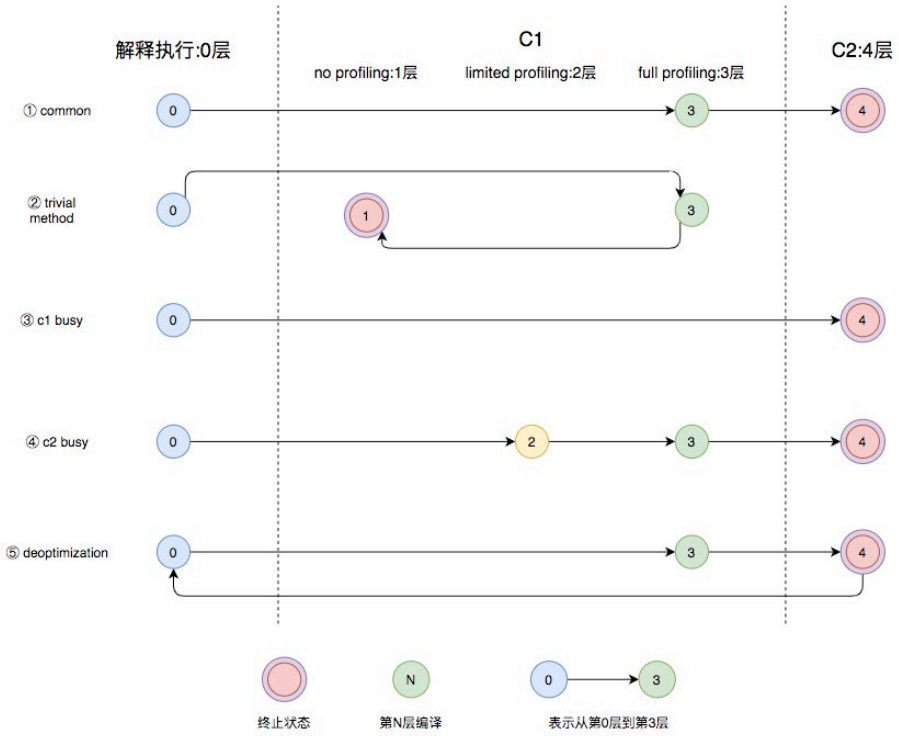
2. 分层编译

在 Java 7 以前，需要研发人员根据服务的性质去选择编译器。对于需要快速启动的，或者一些不会长期运行的服务，可以采用编译效率较高的 C1，对应参数 -client。长期运行的服务，或者对峰值性能有要求的后台服务，可以采用峰值性能更好的 C2，对应参数 -server。Java 7 开始引入了分层编译的概念，它结合了 C1 和 C2 的优势，追求启动速度和峰值性能的一个平衡。分层编译将 JVM 的执行状态分为了五个层次。五个层级分别是：

1. 解释执行。
2. 执行不带 profiling 的 C1 代码。
3. 执行仅带方法调用次数以及循环回边执行次数 profiling 的 C1 代码。
4. 执行带所有 profiling 的 C1 代码。
5. 执行 C2 代码。

profiling 就是收集能够反映程序执行状态的数据。其中最基本的统计数据就是方法的调用次数，以及循环回边的执行次数。

通常情况下，C2 代码的执行效率要比 C1 代码的高出 30% 以上。C1 层执行的代码，按执行效率排序从高至低则是 1 层 >2 层 >3 层。这 5 个层次中，1 层和 4 层都是终止状态，当一个方法到达终止状态后，只要编译后的代码并没有失效，那么 JVM 就不会再次发出该方法的编译请求的。服务实际运行时，JVM 会根据服务运行情况，从解释执行开始，选择不同的编译路径，直到到达终止状态。下图中就列举了几种常见的编译路径：



- 图中第①条路径，代表编译的一般情况，热点方法从解释执行到被 3 层的 C1 编译，最后被 4 层的 C2 编译。
- 如果方法比较小（比如 Java 服务中常见的 getter/setter 方法），3 层的 profiling 没有收集到有价值的数据，JVM 就会断定该方法对于 C1 代码和 C2

代码的执行效率相同，就会执行图中第②条路径。在这种情况下，JVM 会在 3 层编译之后，放弃进入 C2 编译，直接选择用 1 层的 C1 编译运行。

- 在 C1 忙碌的情况下，执行图中第③条路径，在解释执行过程中对程序进行 profiling ，根据信息直接由第 4 层的 C2 编译。
- 前文提到 C1 中的执行效率是 1 层 >2 层 >3 层，第 3 层一般要比第 2 层慢 35% 以上，所以在 C2 忙碌的情况下，执行图中第④条路径。这时方法会被 2 层的 C1 编译，然后再被 3 层的 C1 编译，以减少方法在 3 层的执行时间。
- 如果编译器做了一些比较激进的优化，比如分支预测，在实际运行时发现预测出错，这时就会进行反优化，重新进入解释执行，图中第⑤条执行路径代表的就是反优化。

总的来说，C1 的编译速度更快，C2 的编译质量更高，分层编译的不同编译路径，也就是 JVM 根据当前服务的运行情况来寻找当前服务的最佳平衡点的一个过程。从 JDK 8 开始，JVM 默认开启分层编译。

3. 即时编译的触发

Java 虚拟机根据方法的调用次数以及循环回边的执行次数来触发即时编译。循环回边是一个控制流图中的概念，程序中可以简单理解为往回跳转的指令，比如下面这段代码：

循环回边

```
public void nlp(Object obj) {
  int sum = 0;
  for (int i = 0; i < 200; i++) {
    sum += i;
  }
}
```

上面这段代码经过编译生成下面的字节码。其中，偏移量为 18 的字节码将往回跳至偏移量为 4 的字节码中。在解释执行时，每当运行一次该指令，Java 虚拟机便会将该方法的循环回边计数器加 1。

字节码

```
public void nlp(java.lang.Object);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: sipush        200
       8: if_icmpge     21
      11: iload_1
      12: iload_2
      13: iadd
      14: istore_1
      15: iinc          2, 1
      18: goto          4
      21: return
```

在即时编译过程中，编译器会识别循环的头部和尾部。上面这段字节码中，循环体的头部和尾部分别为偏移量为 11 的字节码和偏移量为 15 的字节码。编译器将在循环体结尾增加循环回边计数器的代码，来对循环进行计数。

当方法的调用次数和循环回边的次数的和，超过由参数 -XX:CompileThreshold 指定的阈值时（使用 C1 时，默认值为 1500；使用 C2 时，默认值为 10000），就会触发即时编译。

开启分层编译的情况下，-XX:CompileThreshold 参数设置的阈值将会失效，触发编译会由以下的条件来判断：

- 方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数。
- 方法调用次数大于由参数 -XX:TierXMINInvocationThreshold 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 -XX:TierXCompileThreshold 指定的阈值乘以系数时。

分层编译触发条件公式

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s
&& i + b > TierXCompileThreshold * s)
i 为调用次数，b 是循环回边次数
```

上述满足其中一个条件就会触发即时编译，并且 JVM 会根据当前的编译方法数以及编译线程数动态调整系数 s。

三、编译优化

即时编译器会对正在运行的服务进行一系列的优化，包括字节码解析过程中的分析，根据编译过程中代码的一些中间形式来做局部优化，还会根据程序依赖图进行全局优化，最后才会生成机器码。

1. 中间表达形式（Intermediate Representation）

在编译原理中，通常把编译器分为前端和后端，前端编译经过词法分析、语法分析、语义分析生成中间表达形式（Intermediate Representation，以下称为 IR），后端会对 IR 进行优化，生成目标代码。

Java 字节码就是一种 IR，但是字节码的结构复杂，字节码这样代码形式的 IR 也不适合做全局的分析优化。现代编译器一般采用图结构的 IR，静态单赋值（Static Single Assignment，SSA）IR 是目前比较常用的一种。这种 IR 的特点是每个变量只能被赋值一次，而且只有当变量被赋值之后才能使用。举个例子：

SSA IR

```
Plain Text
{
  a = 1;
  a = 2;
  b = a;
}
```

上述代码中我们可以轻易地发现 a = 1 的赋值是冗余的，但是编译器不能。传统的编

译器需要借助数据流分析，从后至前依次确认哪些变量的值被覆盖掉。不过，如果借助了 SSA IR，编译器则可以很容易识别冗余赋值。

上面代码的 SSA IR 形式的伪代码可以表示为：

SSA IR

```
Plain Text
{
  a_1 = 1;
  a_2 = 2;
  b_1 = a_2;
}
```

由于 SSA IR 中每个变量只能赋值一次，所以代码中的 a 在 SSA IR 中会分成 a_1、a_2 两个变量来赋值，这样编译器就可以很容易通过扫描这些变量来发现 a_1 的赋值后并没有使用，赋值是冗余的。

除此之外，SSA IR 对其他优化方式也有很大的帮助，例如下面这个死代码删除（Dead Code Elimination）的例子：

DeadCodeElimination

```
public void DeadCodeElimination{
  int a = 2;
  int b = 0
  if(2 > 1){
    a = 1;
  } else{
    b = 2;
  }
  add(a,b)
}
```

可以得到 SSA IR 伪代码：

DeadCodeElimination

```
a_1 = 2;
b_1 = 0
```

```
if true:
  a_2 = 1;
else
  b_2 = 2;
add(a,b)
```

编译器通过执行字节码可以发现 b_2 赋值后不会被使用，else 分支不会被执行。经过死代码删除后就可以得到代码：

DeadCodeElimination

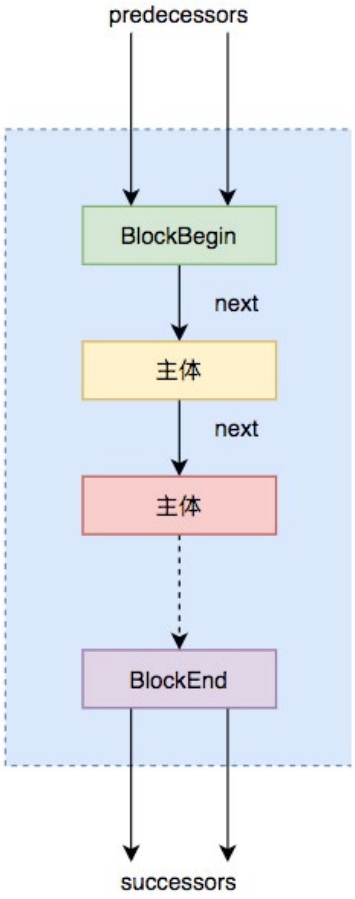
```
public void DeadCodeElimination{
  int a = 1;
  int b = 0;
  add(a,b)
}
```

我们可以将编译器的每一种优化看成一个图优化算法，它接收一个 IR 图，并输出经过转换后的 IR 图。编译器优化的过程就是一个个图节点的优化串联起来的。

C1 中的中间表达形式

前文提及 C1 编译器内部使用高级中间表达形式 HIR，低级中间表达形式 LIR 来进行各种优化，这两种 IR 都是 SSA 形式的。

HIR 是由很多基本块（Basic Block）组成的控制流图结构，每个块包含很多 SSA 形式的指令。基本块的结构如下图所示：



其中，predecessors 表示前驱基本块（由于前驱可能是多个，所以是 BlockList 结构，是多个 BlockBegin 组成的可扩容数组）。同样，successors 表示多个后继基本块 BlockEnd。除了这两部分就是主体块，里面包含程序执行的指令和一个 next 指针，指向下一个执行的主体块。

从字节码到 HIR 的构造最终调用的是 GraphBuilder，GraphBuilder 会遍历字节码构造所有代码基本块储存为一个链表结构，但是这个时候的基本块只有 BlockBegin，不包括具体的指令。第二步 GraphBuilder 会用一个 ValueStack 作为操作数栈和局部变量表，模拟执行字节码，构造出对应的 HIR，填充之前空的基本块，这里给出简单字节码块构造 HIR 的过程示例，如下所示：

字节码构造 HIR

```
         字节码                        Local Value              operand stack
HIR
      5: iload_1                      [i1,i2]                   [i1]
      6: iload_2                      [i1,i2]                   [i1,i2]
                                      ....................................
.........   i3: i1 * i2
      7: imul
      8: istore_3                     [i1,i2, i3]               [i3]
```

可以看出，当执行 iload_1 时，操作数栈压入变量 i1，执行 iload_2 时，操作数栈压入变量 i2，执行相乘指令 imul 时弹出栈顶两个值，构造出 HIR i3 : i1 * i2，生成的 i3 入栈。

C1 编译器优化大部分都是在 HIR 之上完成的。当优化完成之后它会将 HIR 转化为 LIR，LIR 和 HIR 类似，也是一种编译器内部用到的 IR，HIR 通过优化消除一些中间节点就可以生成 LIR，形式上更加简化。

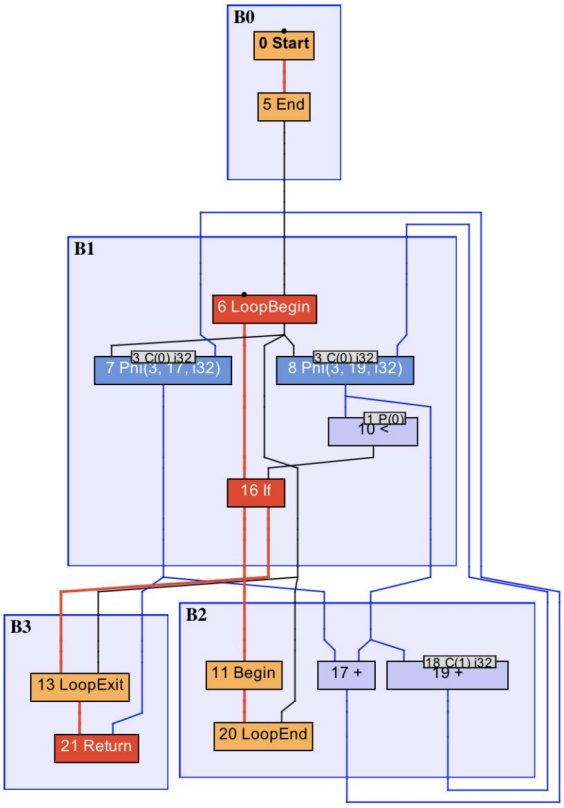
Sea-of-Nodes IR

C2 编译器中的 Ideal Graph 采用的是一种名为 Sea-of-Nodes 中间表达形式，同样也是 SSA 形式的。它最大特点是去除了变量的概念，直接采用值来进行运算。为了方便理解，可以利用 IR 可视化工具 Ideal Graph Visualizer（IGV），来展示具体的 IR 图。比如下面这段代码：

example

```
public static int foo(int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += i;
  }
  return sum;
}
```

对应的 IR 图如下所示：



图中若干个顺序执行的节点将被包含在同一个基本块之中，如图中的 B0、B1 等。B0 基本块中 0 号 Start 节点是方法入口，B3 中 21 号 Return 节点是方法出口。红色加粗线条为控制流，蓝色线条为数据流，而其他颜色的线条则是特殊的控制流或数据流。被控制流边所连接的是固定节点，其他的则是浮动节点（浮动节点指只要能满足数据依赖关系，可以放在不同位置的节点，浮动节点变动的这个过程称为 Schedule）。

这种图具有轻量级的边结构。 图中的边仅由指向另一个节点的指针表示。节点是 Node 子类的实例，带有指定输入边的指针数组。这种表示的优点是改变节点的输入边很快，如果想要改变输入边，只要将指针指向 Node，然后存入 Node 的指针数组就可以了。

依赖于这种图结构，通过收集程序运行的信息，JVM 可以通过 Schedule 那些浮动节点，从而获得最好的编译效果。

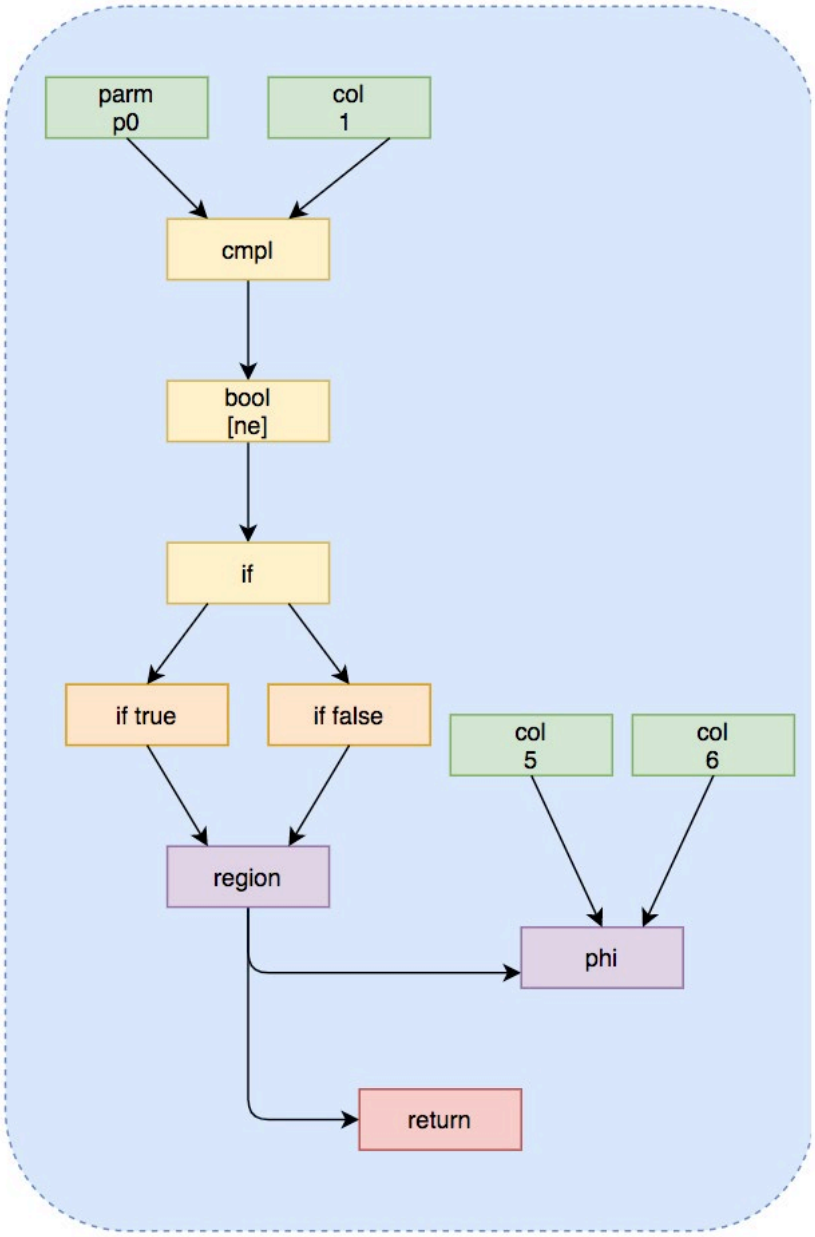
Phi And Region Nodes

Ideal Graph 是 SSA IR。 由于没有变量的概念，这会带来一个问题，就是不同执行路径可能会对同一变量设置不同的值。例如下面这段代码 if 语句的两个分支中，分别返回 5 和 6。此时，根据不同的执行路径，所读取到的值很有可能不同。

example

```
int test(int x) {
int a = 0;
  if(x == 1) {
    a = 5;
  } else {
    a = 6;
  }
  return a;
}
```

为了解决这个问题，就引入一个 Phi Nodes 的概念，能够根据不同的执行路径选择不同的值。于是，上面这段代码可以表示为下面这张图：



Phi Nodes 中保存不同路径上包含的所有值，Region Nodes 根据不同路径的判断条件，从 Phi Nodes 取得当前执行路径中变量应该赋予的值，带有 Phi 节点的 SSA 形式的伪代码如下：

Phi Nodes

```
int test(int x) {
  a_1 = 0;
  if(x == 1){
    a_2 = 5;
  }else {
    a_3 = 6;
  }
  a_4 = Phi(a_2,a_3);
  return a_4;
}
```

Global Value Numbering

Global Value Numbering（GVN）是一种因为 Sea-of-Nodes 变得非常容易的优化技术。

GVN 是指为每一个计算得到的值分配一个独一无二的编号，然后遍历指令寻找优化的机会，它可以发现并消除等价计算的优化技术。如果一段程序中出现了多次操作数相同的乘法，那么即时编译器可以将这些乘法合并为一个，从而降低输出机器码的大小。如果这些乘法出现在同一执行路径上，那么 GVN 还将省下冗余的乘法操作。在 Sea-of-Nodes 中，由于只存在值的概念，因此 GVN 算法将非常简单：即时编译器只需判断该浮动节点是否与已存在的浮动节点的编号相同，所输入的 IR 节点是否一致，便可以将这两个浮动节点归并成一个。比如下面这段代码：

GVN

```
a = 1;
b = 2;
c = a + b;
d = a + b;
e = d;
```

GVN 会利用 Hash 算法编号，计算 a = 1 时，得到编号 1，计算 b = 2 时得到编号 2，计算 c = a + b 时得到编号 3，这些编号都会放入 Hash 表中保存，在计算 d = a + b 时，会发现 a + b 已经存在 Hash 表中，就不会再进行计算，直接从 Hash 表中

取出计算过的值。最后的 e = d 也可以由 Hash 表中查到而进行复用。

可以将 GVN 理解为在 IR 图上的公共子表达式消除（Common Subexpression Elimination，CSE）。两者区别在于，GVN 直接比较值的相同与否，而 CSE 是借助词法分析器来判断两个表达式相同与否。

2. 方法内联

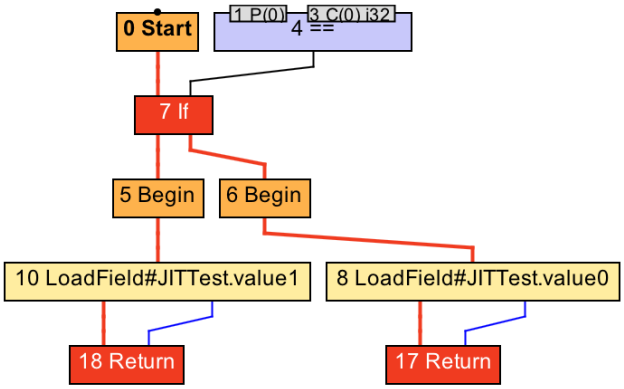
方法内联，是指在编译过程中遇到方法调用时，将目标方法的方法体纳入编译范围之中，并取代原方法调用的优化手段。JIT 大部分的优化都是在内联的基础上进行的，方法内联是即时编译器中非常重要的一环。

Java 服务中存在大量 getter/setter 方法，如果没有方法内联，在调用 getter/setter 时，程序执行时需要保存当前方法的执行位置，创建并压入用于 getter/setter 的栈帧、访问字段、弹出栈帧，最后再恢复当前方法的执行。内联了对 getter/setter 的方法调用后，上述操作仅剩字段访问。在 C2 编译器 中，方法内联在解析字节码的过程中完成。当遇到方法调用字节码时，编译器将根据一些阈值参数决定是否需要内联当前方法的调用。如果需要内联，则开始解析目标方法的字节码。比如下面这个示例（来源于网络）：

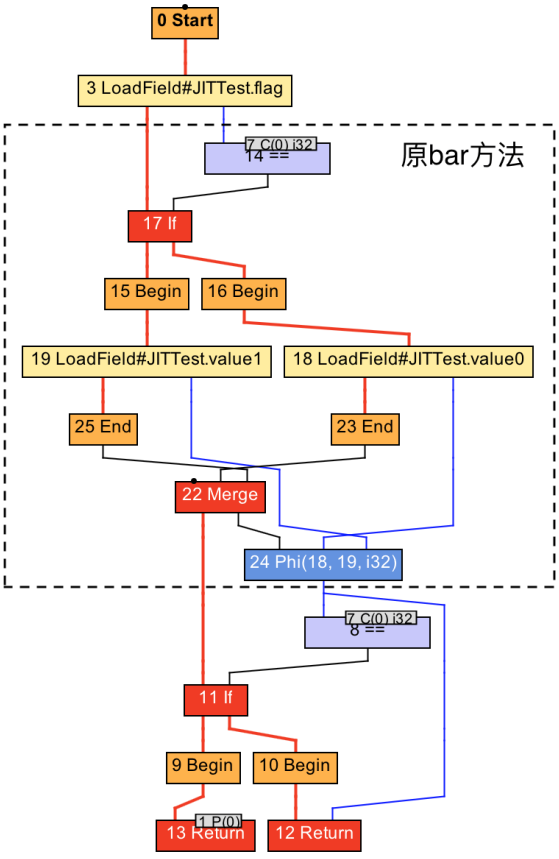
方法内联的过程

```
public static boolean flag = true;
public static int value0 = 0;
public static int value1 = 1;

public static int foo(int value) {
    int result = bar(flag);
    if (result != 0) {
        return result;
    } else {
        return value;
    }
}

public static int bar(boolean flag) {
    return flag ? value0 : value1;
}
```

bar 方法的 IR 图：



内联后的 IR 图：



内联不仅将被调用方法的 IR 图节点复制到调用者方法的 IR 图中，还要完成其他操作。

被调用方法的参数替换为调用者方法进行方法调用时所传入参数。上面例子中，将 bar 方法中的 1 号 P(0) 节点替换为 foo 方法 3 号 LoadField 节点。

调用者方法的 IR 图中，方法调用节点的数据依赖会变成被调用方法的返回。如果存在多个返回节点，会生成一个 Phi 节点，将这些返回值聚合起来，并作为原方法调用节点的替换对象。图中就是将 8 号 == 节点，以及 12 号 Return 节点连接到原 5 号 Invoke 节点的边，然后指向新生成的 24 号 Phi 节点中。

如果被调用方法将抛出某种类型的异常，而调用者方法恰好有该异常类型的处理器，并且该异常处理器覆盖这一方法调用，那么即时编译器需要将被调用方法抛出异常的路径，与调用者方法的异常处理器相连接。

方法内联的条件

编译器的大部分优化都是在方法内联的基础上。所以一般来说，内联的方法越多，生成代码的执行效率越高。但是对于即时编译器来说，内联的方法越多，编译时间也就越长，程序达到峰值性能的时刻也就比较晚。

可以通过虚拟机参数 -XX:MaxInlineLevel 调整内联的层数，以及 1 层的直接递归调用（可以通过虚拟机参数 -XX:MaxRecursiveInlineLevel 调整）。一些常见的内联相关的参数如下表所示：

参数名	默认值	说明
-XX:InlineSmallCode	2000	如果目标方法已被编译，且其生成的机器码大小超过该值，则无法内联
-XX:MaxTrivialSize	6	如果方法的字节码大小小于该值，则直接内联
-XX:MinInliningThreshold	250	如果目标方法的调用次数低于该值，则无法内联
-XX:InlineFrequencyCount	100	如果方法调用指令执行次数超过该值，则认为是热点方法
-XX:MaxInlineSize	35	如果非热点方法的字节码大小超过该值，则无法内联
-XX:FreqInlineSize	325	如果热点方法的字节码大小超过该值，则无法内联
-XX:LineNodeCountInliningCutof	40000	编译过程中IR节点数目的上限

虚函数内联

内联是 JIT 提升性能的主要手段，但是虚函数使得内联是很难的，因为在内联阶段并不知道他们会调用哪个方法。例如，我们有一个数据处理的接口，这个接口中的一个方法有三种实现 add、sub 和 multi，JVM 是通过保存虚函数表 Virtual Method Table（以下称为 VMT）存储 class 对象中所有的虚函数，class 的实例对象保存着一个 VMT 的指针，程序运行时首先加载实例对象，然后通过实例对象找到 VMT，通过 VMT 找到对应方法的地址，所以虚函数的调用比直接指向方法地址的 classic call 性能上会差一些。很不幸的是，Java 中所有非私有的成员函数的调用都是虚调用。

C2 编译器已经足够智能，能够检测这种情况并会对虚调用进行优化。比如下面这段代码例子：

virtual call

```
public class SimpleInliningTest
{
    public static void main(String[] args) throws InterruptedException
{
        VirtualInvokeTest obj = new VirtualInvokeTest();
        VirtualInvoke1 obj1 = new VirtualInvoke1();
        for (int i = 0; i < 100000; i++) {
            invokeMethod(obj);
            invokeMethod(obj1);
```

```
        }
        Thread.sleep(1000);
    }

    public static void invokeMethod(VirtualInvokeTest obj) {
        obj.methodCall();
    }

    private static class VirtualInvokeTest {
        public void methodCall() {
            System.out.println("virtual call");
        }
    }

    private static class VirtualInvoke1 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
}
```

经过 JIT 编译器优化后，进行反汇编得到下面这段汇编代码：

```
 0x0000000113369d37: callq  0x00000001132950a0  ; OopMap{off=476}
                                                ;*invokevirtual
methodCall  // 代表虚调用
                                                ; -
SimpleInliningTest::invokeMethod@1 (line 18)
                                                ;   {optimized virtual_
call}  // 虚调用已经被优化
```

可以看到 JIT 对 methodCall 方法进行了虚调用优化 optimized virtual_call。经过优化后的方法可以被内联。但是 C2 编译器的能力有限，对于多个实现方法的虚调用就“无能为力”了。

比如下面这段代码，我们增加一个实现：

多实现的虚调用

```
public class SimpleInliningTest
{
    public static void main(String[] args) throws InterruptedException
{
```

```
        VirtualInvokeTest obj = new VirtualInvokeTest();
        VirtualInvoke1 obj1 = new VirtualInvoke1();
        VirtualInvoke2 obj2 = new VirtualInvoke2();
        for (int i = 0; i < 100000; i++) {
            invokeMethod(obj);
            invokeMethod(obj1);
        invokeMethod(obj2);
        }
        Thread.sleep(1000);
    }

    public static void invokeMethod(VirtualInvokeTest obj) {
        obj.methodCall();
    }

    private static class VirtualInvokeTest {
        public void methodCall() {
            System.out.println("virtual call");
        }
    }

    private static class VirtualInvoke1 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
    private static class VirtualInvoke2 extends VirtualInvokeTest {
        @Override
        public void methodCall() {
            super.methodCall();
        }
    }
}
```

经过反编译得到下面的汇编代码：

代码块

```
 0x000000011f5f0a37: callq  0x000000011f4fd2e0  ; OopMap{off=28}
                                                ;*invokevirtual
methodCall  // 代表虚调用
                                                ; -
SimpleInliningTest::invokeMethod@1 (line 20)
                                                ;   {virtual_call}  // 虚
调用未被优化
```

可以看到多个实现的虚调用未被优化，依然是 virtual_call。

Graal 编译器针对这种情况，会去收集这部分执行的信息，比如在一段时间，发现前面的接口方法的调用 add 和 sub 是各占 50% 的几率，那么 JVM 就会在每次运行时，遇到 add 就把 add 内联进来，遇到 sub 的情况再把 sub 函数内联进来，这样这两个路径的执行效率就会提升。在后续如果遇到其他不常见的情况，JVM 就会进行去优化的操作，在那个位置做标记，再遇到这种情况时切换回解释执行。

3. 逃逸分析

逃逸分析是“一种确定指针动态范围的静态分析，它可以分析在程序的哪些地方可以访问到指针”。Java 虚拟机的即时编译器会对新建的对象进行逃逸分析，判断对象是否逃逸出线程或者方法。即时编译器判断对象是否逃逸的依据有两种：

1. 对象是否被存入堆中（静态字段或者堆中对象的实例字段），一旦对象被存入堆中，其他线程便能获得该对象的引用，即时编译器就无法追踪所有使用该对象的代码位置。
2. 对象是否被传入未知代码中，即时编译器会将未被内联的代码当成未知代码，因为它无法确认该方法调用会不会将调用者或所传入的参数存储至堆中，这种情况，可以直接认为方法调用的调用者以及参数是逃逸的。

逃逸分析通常是在方法内联的基础上进行的，即时编译器可以根据逃逸分析的结果进行诸如锁消除、栈上分配以及标量替换的优化。下面这段代码的就是对象未逃逸的例子：

```
pulbic class Example{
    public static void main(String[] args) {
      example();
    }
    public static void example() {
      Foo foo = new Foo();
      Bar bar = new Bar();
      bar.setFoo(foo);
    }
  }
```

```
  class Foo {}

  class Bar {
    private Foo foo;
    public void setFoo(Foo foo) {
      this.foo = foo;
    }
  }
}
```

在这个例子中，创建了两个对象 foo 和 bar，其中一个作为另一个方法的参数提供。该方法 setFoo() 存储对收到的 Foo 对象的引用。如果 Bar 对象在堆上，则对 Foo 的引用将逃逸。但是在这种情况下，编译器可以通过逃逸分析确定 Bar 对象本身不会对逃逸出 example() 的调用。这意味着对 Foo 的引用也不能逃逸。因此，编译器可以安全地在栈上分配两个对象。

锁消除

在学习 Java 并发编程时会了解锁消除，而锁消除就是在逃逸分析的基础上进行的。

如果即时编译器能够证明锁对象不逃逸，那么对该锁对象的加锁、解锁操作没就有意义。因为线程并不能获得该锁对象。在这种情况下，即时编译器会消除对该不逃逸锁对象的加锁、解锁操作。实际上，编译器仅需证明锁对象不逃逸出线程，便可以进行锁消除。由于 Java 虚拟机即时编译的限制，上述条件被强化为证明锁对象不逃逸出当前编译的方法。不过，基于逃逸分析的锁消除实际上并不多见。

栈上分配

我们都知道 Java 的对象是在堆上分配的，而堆是对所有对象可见的。同时，JVM 需要对所分配的堆内存进行管理，并且在对象不再被引用时回收其所占据的内存。如果逃逸分析能够证明某些新建的对象不逃逸，那么 JVM 完全可以将其分配至栈上，并且在 new 语句所在的方法退出时，通过弹出当前方法的栈桢来自动回收所分配的内存空间。这样一来，我们便无须借助垃圾回收器来处理不再被引用的对象。不过 Hotspot 虚拟机，并没有进行实际的栈上分配，而是使用了标量替换这一技术。所谓

的标量，就是仅能存储一个值的变量，比如 Java 代码中的基本类型。与之相反，聚合量则可能同时存储多个值，其中一个典型的例子便是 Java 的对象。编译器会在方法内将未逃逸的聚合量分解成多个标量，以此来减少堆上分配。下面是一个标量替换的例子：

标量替换

```
public class Example{
  @AllArgsConstructor
  class Cat{
    int age;
    int weight;
  }
  public static void example(){
    Cat cat = new Cat(1,10);
    addAgeAndWeight(cat.age,Cat.weight);
  }
}
```

经过逃逸分析，cat 对象未逃逸出 example() 的调用，因此可以对聚合量 cat 进行分解，得到两个标量 age 和 weight，进行标量替换后的伪代码：

```
public class Example{
  @AllArgsConstructor
  class Cat{
    int age;
    int weight;
  }
  public static void example(){
    int age = 1;
    int weight = 10;
    addAgeAndWeight(age,weight);
  }
}
```

部分逃逸分析

部分逃逸分析也是 Graal 对于概率预测的应用。通常来说，如果发现一个对象逃逸出了方法或者线程，JVM 就不会去进行优化，但是 Graal 编译器依然会去分析当前程序的执行路径，它会在逃逸分析基础上收集、判断哪些路径上对象会逃逸，哪

些不会。然后根据这些信息，在不会逃逸的路径上进行锁消除、栈上分配这些优化手段。

4. Loop Transformations

在文章中介绍 C2 编译器的部分有提及到，C2 编译器在构建 Ideal Graph 后会进行很多的全局优化，其中就包括对循环的转换，最重要的两种转换就是循环展开和循环分离。

循环展开

循环展开是一种循环转换技术，它试图以牺牲程序二进制码大小为代价来优化程序的执行速度，是一种用空间换时间的优化手段。

循环展开通过减少或消除控制程序循环的指令，来减少计算开销，这种开销包括增加指向数组中下一个索引或者指令的指针算数等。如果编译器可以提前计算这些索引，并且构建到机器代码指令中，那么程序运行时就可以不必进行这种计算。也就是说有些循环可以写成一些重复独立的代码。比如下面这个循环：

循环展开

```
public void loopRolling(){
  for(int i = 0;i<200;i++){
    delete(i);
  }
}
```

上面的代码需要循环删除 200 次，通过循环展开可以得到下面这段代码：

循环展开

```
public void loopRolling(){
  for(int i = 0;i<200;i+=5){
    delete(i);
    delete(i+1);
    delete(i+2);
    delete(i+3);
    delete(i+4);
  }
}
```

这样展开就可以减少循环的次数，每次循环内的计算也可以利用 CPU 的流水线提升效率。当然这只是一个示例，实际进行展开时，JVM 会去评估展开带来的收益，再决定是否进行展开。

循环分离

循环分离也是循环转换的一种手段。它把循环中一次或多次的特殊迭代分离出来，在循环外执行。举个例子，下面这段代码：

循环分离

```
int a = 10;
for(int i = 0;i<10;i++){
  b[i] = x[i] + x[a];
  a = i;
}
```

可以看出这段代码除了第一次循环 a = 10 以外，其他的情况 a 都等于 i−1。所以可以把特殊情况分离出去，变成下面这段代码：

循环分离

```
b[0] = x[0] + 10;
for(int i = 1;i<10;i++){
  b[i] = x[i] + x[i-1];
}
```

这种等效的转换消除了在循环中对 a 变量的需求，从而减少了开销。

5. 窥孔优化与寄存器分配

前文提到的窥孔优化是优化的最后一步，这之后就会程序就会转换成机器码，窥孔优化就是将编译器所生成的中间代码（或目标代码）中相邻指令，将其中的某些组合替换为效率更高的指令组，常见的比如强度削减、常数合并等，看下面这个例子就是一个强度削减的例子：

强度削减

```
y1=x1*3  经过强度削减后得到  y1=(x1<<1)+x1
```

编译器使用移位和加法削减乘法的强度，使用更高效率的指令组。

寄存器分配也是一种编译的优化手段，在 C2 编译器中普遍的使用。它是通过把频繁使用的变量保存在寄存器中，CPU 访问寄存器的速度比内存快得多，可以提升程序的运行速度。

寄存器分配和窥孔优化是程序优化的最后一步。经过寄存器分配和窥孔优化之后，程序就会被转换成机器码保存在 codeCache 中。

四、实践

即时编译器情况复杂，同时网络上也很少有实战经验，以下是我们团队的一些调整经验。

1. 编译相关的重 * 要参数

- -XX:+TieredCompilation：开启分层编译，JDK8 之后默认开启
- -XX:+CICompilerCount=N：编译线程数，设置数量后，JVM 会自动分配线程数，C1:C2 = 1:2
- -XX:TierXBackEdgeThreshold：OSR 编译的阈值
- -XX:TierXMinInvocationThreshold：开启分层编译后各层调用的阈值
- -XX:TierXCompileThreshold：开启分层编译后的编译阈值
- -XX:ReservedCodeCacheSize：codeCache 最大大小
- -XX:InitialCodeCacheSize：codeCache 初始大小

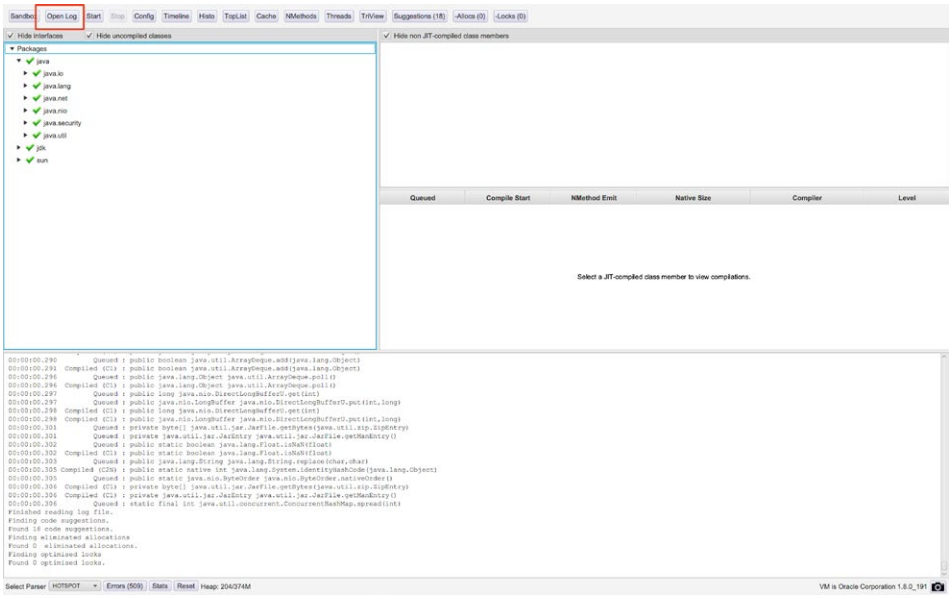
-XX:TierXMinInvocationThreshold 是开启分层编译的情况下，触发编译的阈值参数，当方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数，或者当方法调用次数大于由参数 -XX:TierXMINInvocationThreshold 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 -XX:TierX-

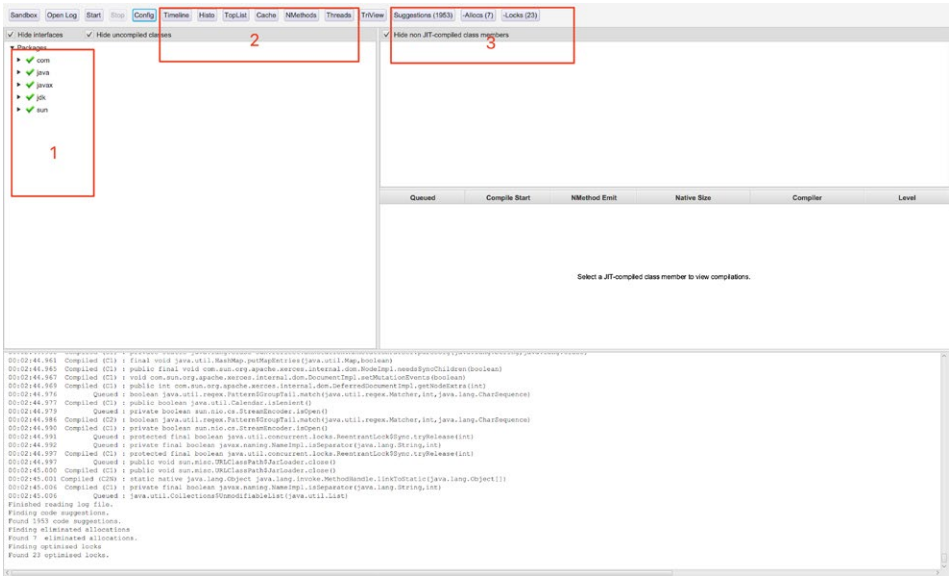
CompileThreshold 指定的阈值乘以系数时，便会触发 X 层即时编译。分层编译开启下会乘以一个系数，系数根据当前编译的方法和编译线程数确定，降低阈值可以提升编译方法数，一些常用但是不能编译的方法可以编译优化提升性能。

由于编译情况复杂，JVM 也会动态调整相关的阈值来保证 JVM 的性能，所以不建议手动调整编译相关的参数。除非一些特定的 Case，比如 codeCache 满了停止了编译，可以适当增加 codeCache 大小，或者一些非常常用的方法，未被内联到，拖累了性能，可以调整内敛层数或者内联方法的大小来解决。

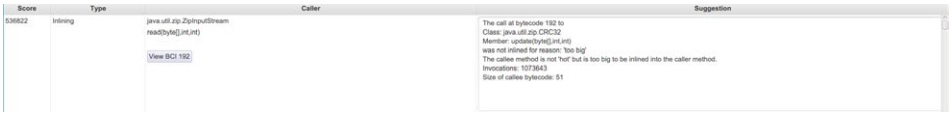
2. 通过 JITwatch 分析编译日志

通过增加 -XX:+UnlockDiagnosticVMOptions -XX:+PrintCompilation -XX-:+PrintInlining -XX:+PrintCodeCache -XX:+PrintCodeCacheOnCompilation -XX:+TraceClassLoading -XX:+LogCompilation -XX:LogFile=LogPath 参数可以输出编译、内联、codeCache 信息到文件。但是打印的编译日志多且复杂很难直接从其中得到信息，可以使用 JITwatch 的工具来分析编译日志。JITwatch 首页的 Open Log 选中日志文件，点击 Start 就可以开始分析日志。





如上图所示，区域 1 中是整个项目 Java Class 包括引入的第三方依赖；区域 2 是功能区 Timeline 以图形的形式展示 JIT 编译的时间轴，Histo 是直方图展示一些信息，TopList 里面是编译中产生的一些对象和数据的排序，Cache 是空闲 codeCache 空间，NMethod 是 Native 方法，Threads 是 JIT 编译的线程；区域 3 是 JITwatch 对日志分析结果的展示，其中 Suggestions 中会给出一些代码优化的建议，举个例子，如下图中：



我们可以看到在调用 ZipInputStream 的 read 方法时，因为该方法没有被标记为热点方法，同时又“太大了”，导致无法被内联到。使用 -XX:CompileCommand 中 inline 指令可以强制方法进行内联，不过还是建议谨慎使用，除非确定某个方法内联会带来不少的性能提升，否则不建议使用，并且过多使用对编译线程和 codeCache 都会带来不小的压力。

区域 3 中的 -Allocs 和 -Locks 逃逸分析后 JVM 对代码做的优化，包括栈上分配、锁消除等。

3. 使用 Graal 编译器

由于 JVM 会去根据当前的编译方法数和编译线程数对编译阈值进行动态的调整，所以实际服务中对这一部分的调整空间是不大的，JVM 做的已经足够多了。

为了提升性能，在服务中尝试了最新的 Graal 编译器。只需要使用 -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 就可以启动 Graal 编译器来代替 C2 编译器，并且响应 C2 的编译请求，不过要注意的是，Graal 编译器与 ZGC 不兼容，只能与 G1 搭配使用。

前文有提到过，Graal 是一个用 Java 写的即时编译器，它从 Java 9 开始便被集成自 JDK 中，作为实验性质的即时编译器。Graal 编译器就是脱身于 GraalVM，GraalVM 是一个高性能的、支持多种编程语言的执行环境。它既可以在传统的 OpenJDK 上运行，也可以通过 AOT（Ahead-Of-Time）编译成可执行文件单独运行，甚至可以集成至数据库中运行。

前文提到过数次，Graal 的优化都基于某种假设（Assumption）。当假设出错的情况下，Java 虚拟机会借助去优化（Deoptimization）这项机制，从执行即时编译器生成的机器码切换回解释执行，在必要情况下，它甚至会废弃这份机器码，并在重新收集程序 profile 之后，再进行编译。

这些中激进的手段使得 Graal 的峰值性能要好于 C2，而且在 Scale、Ruby 这种语言 Graal 表现更加出色，Twitter 目前已经在服务中大量的使用 Graal 来提升性能，企业版的 GraalVM 使得 Twitter 服务性能提升了 22%。

使用 Graal 编译器后性能表现

在我们的线上服务中，启用 Graal 编译后，TP9999 从 60ms -> 50ms ，下降 10ms，下降幅度达 16.7%。

运行过程中的峰值性能会更高。可以看出对于该服务，Graal 编译器带来了一定的性能提升。

Graal 编译器的问题

Graal 编译器的优化方式更加激进，因此在启动时会进行更多的编译，Graal 编译器本身也需要被即时编译，所以服务刚启动时性能会比较差。

考虑的解决办法：JDK 9 开始提供工具 jaotc，同时 GraalVM 的 Native Image 都是可以通过静态编译，极大地提升服务的启动速度的方式，但是 GraalVM 会使用自己的垃圾回收，这是一种很原始的基于复制算法的垃圾回收，相比 G1、ZGC 这些优秀的新型垃圾回收器，它的性能并不好。同时 GraalVM 对 Java 的一些特性支持也不够，比如基于配置的支持，比如反射就需要把所有需要反射的类配置一个 JSON 文件，在大量使用反射的服务，这样的配置会是很大的工作量。我们也在做这方面的调研。

五、总结

本文主要介绍了 JIT 即时编译的原理以及在美团一些实践的经验，还有最前沿的即时编译器的使用效果。作为一项解释型语言中提升性能的技术，JIT 已经比较成熟了，在很多语言中都有使用。对于 Java 服务，JVM 本身已经做了足够多，但是我们还应该不断深入了解 JIT 的优化原理和最新的编译技术，从而弥补 JIT 的劣势，提升 Java 服务的性能，不断追求卓越。

六、参考文献

《深入理解 Java 虚拟机》
《Proceedings of the Java™ Virtual Machine Research and Technology Symposium》Monterey, California, USA April 23 – 24, 2001
《Visualization of Program Dependence Graphs》Thomas W ü rthinger
《深入拆解 Java 虚拟机》郑宇迪
JIT 的 Profile 神器 JITWatch

作者简介

珩智，昊天，薛超，均来自美团 AI 平台 / 搜索与 NLP 部。

招聘信息

美团搜索与 NLP 部，长期招聘搜索、对话、NLP 算法工程师，坐标北京 / 上海，感兴趣的同学可投递简历至：tech@meituan.com（邮件标题请注明：搜索与 NLP 部）。

MyBatis 版本升级引发的线上告警回顾及原理分析

作者：凯伦

背景

某天晚上，美团到店事业群某项系统服务正在进行常规需求的上线。因为在内部的 Plus 系统发布时，提示 inf-bom 版本需要升级，于是我们就将 inf-bom 版本从 1.3.9.6 升级至 1.4.2.1，如下图 1 所示：

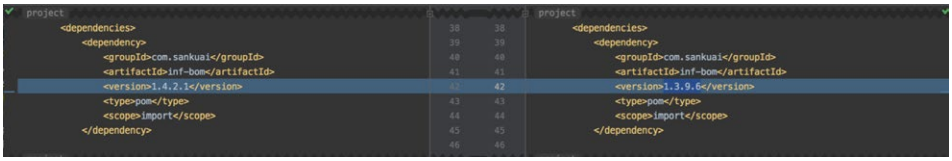


图 1　版本升级

不过，当服务上线后，开始陆续出现了一些更新系统交互日志方面的报警，这属于系统的辅助流程，报警如下方代码所示。我们发现都是跟 MyBatis 相关的报警，说明在进行类型转换的时候，系统产生了强转错误。

```
更新开票请求返回日志 , id:{#######},
response:{{"code":XXX,"data":{"callType":3,"code":XXX,"msg":"XXXX",
"shopId":XXXXX,"taxPlateDockType":"XXXXXXX"},"msg":"XXXXX",
"success":XXXX}}
nested execption is org.apache.ibatis.type.TypeException: Could not set
parameters for mapping: ParameterMapping{property='updateTime', mode=IN,
javaType=class java.lang.String,
jdbcTyp=null,resultMapId='null',jdbcTypeName='null',expression='null'}.
Cause org.apache.ibatis.type.TypeException,Error setting non null
parameter #2 with JdbcType null. Try setting a
different Jdbc Type for this parameter or a different configuration
property.Cause java.lang.ClassCastException:java.time.LocalDateTime
cannot be cast to java.lang.String
```

因为报警这一块代码，属于历史功能，如果失败并不会影响主流程。但在定位期间，如果频繁报警的话，就会造成一定的干扰。因此，我们马上采取了回滚操作，将 inf-

bom 的版本回滚至历史版本，直至报警消失，然后再进行问题的定位和分析。以下章节就是我们对报警原因的定位及原因详细分析的介绍，希望这些思路能够对大家有所启发和帮助。

报警原因定位

在回滚完毕后，我们开始具体分析报警产生的主要原因，于是进行了以下几步的排查。

第一步，查看了报警的 Mapper 方法，如下代码段所示。这个是接收返回参数，根据主键 id，更新具体响应内容和时间的代码，入参有 3 个，类型分别为 long、String 和 LocalDateTime。

```
int updateResponse(@Param("id")long id, @Param("response")String
response, @Param("updateTime")LocalDateTime updateTime);
```

第二步，我们查看了 Mapper 方法对应的 XML 文件，如下代码段所示，对应的 parameterType 类型是 String，而实际参数的类型包括 long、String 以及 LocalDateTime。

```
<update id="updateResponse" parameterType="java.lang.String">
UPDATE invoice_log
  SET response = #{response}, update_time = #{updateTime}
WHERE id = #{id}
</update>
```

第三步，我们查看了 MyBatis 上线前后的版本，报警的内容是：MyBatis 在处理 SQL 语句时，发现不能将 LocalDateTime 转型为 String，这一段逻辑在上线前是可以正常运行的，并且上线的业务逻辑对这段历史代码无改动。因此，我们猜测是因为 inf-bom 的升级，从而导致 MyBatis 的版本发生了变化，对某些历史功能不再支持了。MyBatis 版本上线前后的变化如下表所示：

inf-bom版本	MyBatis版本
1.3.9.6	3.2.3
1.4.2.1	3.4.6

表 1　MyBatis 版本升级前后对比

第四步，我们通过第三步可以得到，在这次 inf-bom 的版本升级中，MyBatis 的版本直接升了两个大版本，因此我们可以基本将原因猜测为 MyBatis 升级跨度较大，导致部分历史功能没有兼容支持，从而引起线上 SQL 的更新报错。

第五步，为了具体验证第四步的想法，我们通过 UT 的方式，将 MyBatis 的版本不断从 3.4.6 往下降，直至没有报错的位置。最终的定位是：当 MyBatis 版本为 3.2.3 时，线上代码是正常可用的，但只要升一个版本，也就是自 3.2.4 开始，就开始不兼容目前的用法。不过，我们当时的思路并不是很好，应该从小版本逐个往上升或者使用二分法，可以加速定位版本的效率。

最后，我们定位到了产生报警的根本问题。总的来说，MyBatis 版本由 inf-bom 引入而来，inf-bom 从 3.2.3 升级到了 3.4.6 版本，而 MyBatis 自 3.2.4 开始就不支持目前系统内的 SQL Mapper 的用法，因此在升级后，线上就出现了频繁报警的问题。

问题已经定位，但是还有很多事情我们需要弄清楚。为什么版本升级后就不兼容历史的用法？具体是哪一块内容不兼容？背后的原理又是什么？下文，我们会详细进行分析。

详细分析

MyBatis 升级 3.2.4 版本的官方 Release 公告

首先，从报错的原因上来看，请注意这句话：“Caused by: java.lang.ClassCastException: java.lang.LocalDateTime cannot be cast to java.lang.String.”MyBatis 在构建 SQL 语句时，发现时间字段类型 LocalDateTime 不能强制转为 String 类型。而这个 SQL 对应的 XML 配置在 3.2.3 的版本是可以正常使用的，那么我们先从 MyBatis 的 Release Log 上查看 3.2.4 版本到底发生了什么变化。

> An special remark about this feature. Previous versions ignored the “parameterType” attribute and used the actual parameter to calculate bindings. This version builds the binding information during startup and the “parameterType” attribute is used if present (though it is still optional), so in case you had a wrong value for it you will have to change it.

从官网的 Release Log 可以看到，MyBatis 在 3.2.4 以前的版本，会忽略 XML 中的 parameterType 这个属性，并且使用真实的变量类型进行值的处理。但在 3.2.4 及以后的版本中，这个属性就被启用了，如果出现类型不匹配的话，就会出现转型失败的报错。这也提示我们开发者，在升级版本时，需要检查系统内的 XML 配置，使类型进行匹配，或者不设置该属性，让 MyBatis 自行进行计算。

根据以上内容，我们可以了解到，在版本升级后，MyBatis 在构建 SQL 语句，在获取字段值时的逻辑发生了变化。接下来我们将通过一个简单的示例，来了解一下 MyBatis 在获取字段值这一块的具体代码流程是怎样的，以 3.2.3 版本为例。

以版本 3.2.3 为例，MyBatis 构建 SQL 语句过程的原理分析

我们看一下配置，首先定义一个通过主键 id 获取学生信息的方法，仿造系统内的历史代码，我们将 parameterType 定义为 java.lang.String，这和方法对应的参数 int 并不相同。

```
public StudentEntity getStudentById(@Param("id") int id);
<select id="getStudentById" parameterType="java.lang.String"
resultType="entity.StudentEntity">
SELECT id,name,age FROM student WHERE id = #{id}
</select>
```

MyBatis 框架要做的事情，就是在运行 getStudentById(2) 的时候，将 #{id} 进行替换，使 SQL 语句变成 SELECT id,name,age FROM student WHERE id = 2。MyBatis 要将 SQL 语句完整替换成带参数值的版本，需要经历框架初始化以及实际运行时动态替换这两个部分。因为 MyBatis 的代码非常多，接下来我们主要阐释和本次案例相关的内容。

在框架初始化阶段，主要包括以下流程，如下图 2 所示：

图 2　框架初始化流程

在框架初始化阶段，有一些组件会被构建，逐一做个简单的介绍：

- **SqlSession**：作为 MyBatis 工作的主要顶层 API，表示和数据库交互的会话，完成必要的数据库增删改查功能。
- **数据库增删改查功能**：负责根据用户传递的 parameterObject，动态地生成 SQL 语句，将信息封装到 BoundSql 对象中，并返回。
- **Configuration**：MyBatis 所有的配置信息都维持在 Configuration 对象之中。

接下来，我们主要关注 SqlSource，这个类会负责生成 SQL 语句，这也是本次案例中，3.2.3 和 3.2.4 差异比较大的一个地方。下面，我们会介绍一些源码。

在构建 Configuration 的过程中，会涉及到构建对应每一条 SQL 语句对应的 MappedStatement，parameterTypeClass 就是根据我们在 XML 配置中写的 parameterType 转换而来，值为 java.lang.String，在构建 SqlSource 时，传入这个参数。如下图 3 所示：

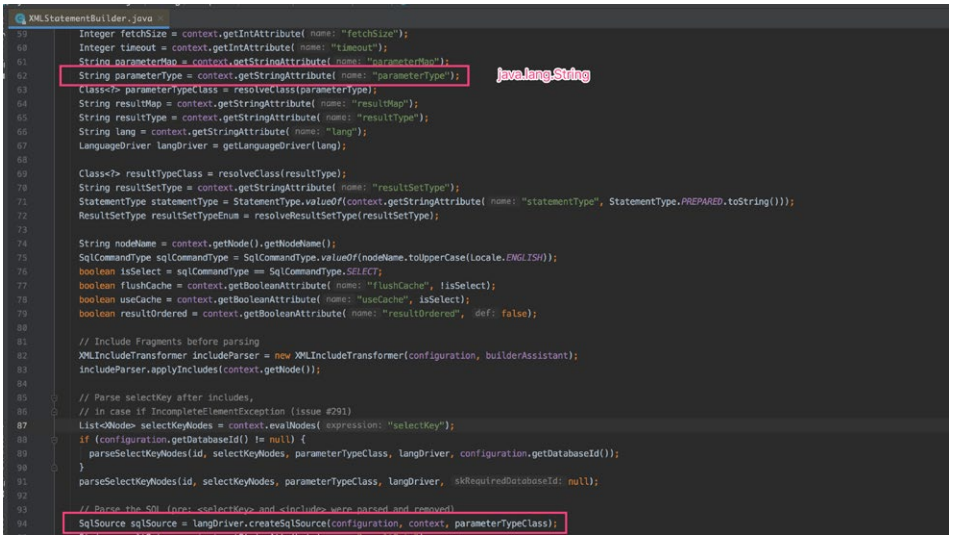


图 3　SqlSource 依赖参数

在 SqlSource 的构建中，parameterType 参数其实是被忽略不用的，并没有继续往下传递，这跟官方的描述是一致的。因为 3.2.4 之前这个 parameterType 属性被

忽略了，然后就创建了 DynamicSqlSource，这个类主要是用于处理 MyBatis 动态 SQL 的类。如下图 4 所示：

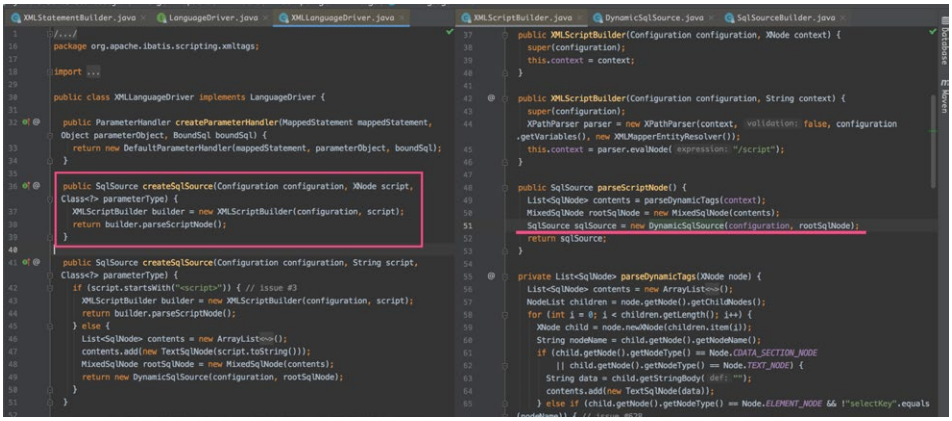


图 4　SqlSource 构建

在框架初始化的阶段，需要介绍的内容，在 3.2.3 版本已经介绍完毕。当执行 getStudentById 方法时，MyBatis 的流程如下图 5 所示。因受限于图片长度，我们对布局进行了一些调整：

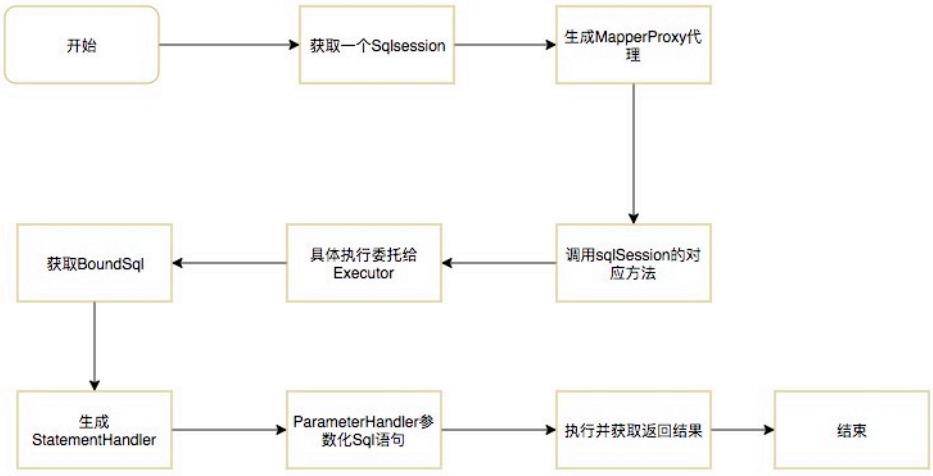


图 5　运行流程

在具体执行阶段，也涉及到一些组件，我们需要做简单的了解：

- **SqlSession**：作为 MyBatis 工作的主要顶层 API，表示和数据库交互的会话，完成必要数据库增删改查功能。
- **Executor**：MyBatis 执行器，这是 MyBatis 调度的核心，负责 SQL 语句的生成和查询缓存的维护。
- **BoundSql**：表示动态生成的 SQL 语句以及相应的参数信息。
- **StatementHandler**：封装了 JDBC Statement 操作，负责对 JDBC statement 的操作，如设置参数、将 Statement 结果集转换成 List 集合等等。
- **ParameterHandler**：负责对用户传递的参数转换成 JDBC Statement 所需要的参数。
- **TypeHandler**：负责 Java 数据类型和 JDBC 数据类型之间的映射和转换。

我们主要关注获取 BoundSql 以及参数化语句的流程，这也是 3.2.3 和 3.2.4 差异比较大的一个地方。在进入 Executor 的 Query 方法后，会首先通过对应的 MappedStatement 来获取 BoundSql，用来帮助我们动态生成 SQL 语句，里面绑定了对应的 SQL 以及参数映射关系。在构建框架阶段，我们使用的 SqlSource 是 DynamicSqlSource，通过这个类来生成获取 BoundSql，如下图 6 所示：

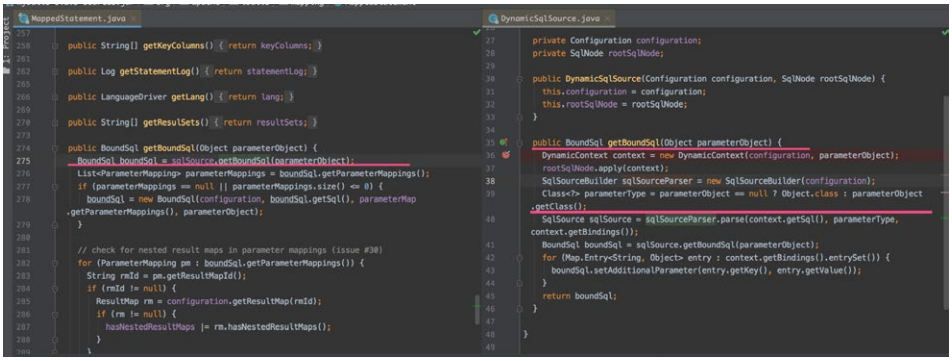


图 6　获取 BoundSql

通过图 6 的代码，我们可以得知，parameterType 在初始化阶段未被使用，而是在 SQL 执行时获取到的，但获取到的类型是 parameterObject 对应的类型，这个类是用来记录 Mapper 方法上对应的参数。如下图 7 所示，它并非在 SQL 配置文件中标注的 java.lang.String。

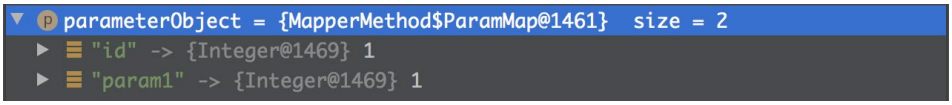


图 7　parameterObject 类型

然后我们通过 SqlSourceBuilder 的 parse 方法对 SQL 以及获取到的类型进行再次处理，其中的流程代码比较长。在这个过程中，我们主要去构建 SQL 的参数和 Java 类型的绑定关系，MyBatis 依赖这个绑定关系，使用对应的 TypeHandler 去进行值的转换。

调用链路是 `SqlSourceParser.parse -> 内部类 ParameterMappingTokenHandler.handleToken -> 私有方法 buildParameterMapping`，如下图 8 中的代码所示。因为当前的 parameterType 为 MapperMethod$ParamMap，经过了多个 if 判断，判定当前 property id 的 propertyType 为 Object.class 类型。接下来，构建 SQL 的参数和 Java 类型的绑定关系 ParameterMapping，再进行返回。

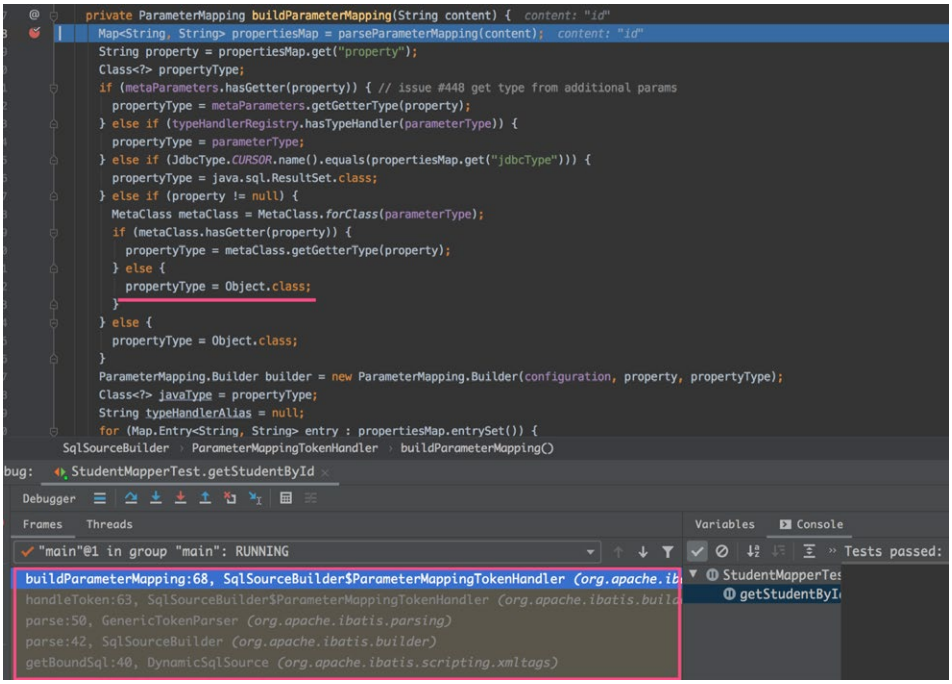


图 8　buildParameterMapping 过程

构建完成的 ParameterMapping 的结构如下图 9 中的代码所示，参数 id 对应的 javaType 类型为 java.lang.Object，对应的 TypeHander 处理器为 UnknownTypeHandler，也就是未找到合适的 TypeHandler 的兜底选项。

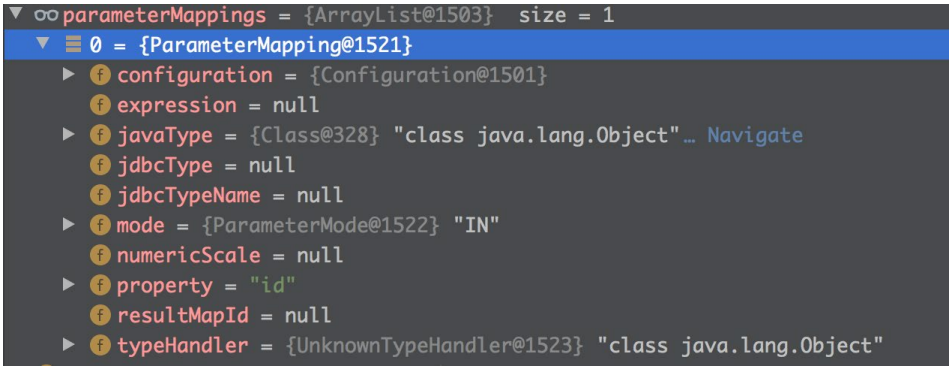


图 9　ParameterMapping 结构

接下来，流程就会流转到Executor，在`org.apache.ibatis.executor.SimpleExecutor#doQuery`进行查询时，会根据当前的SQL类型，生成对应的StatementHandler。因为我们目前都是用的预编译SQL，因此生成的statementHandler就是PreparedStatementHandler，熟悉JDBC的小伙伴应该马上可以猜到对应的语句是什么类型了。然后，我们对这句SQL语句进行填充，如下图10中的代码所示。我们会通过PreparedStatementHandler的parameterize方法对Statement进行参数化，也就是进行填充。

```
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {  handler: RoutingStatementHandler@1552
  Statement stmt;  stmt: "com.mysql.jdbc.JDBC42PreparedStatement@662b4c69: SELECT id,name,age FROM student WHERE id = ** NOT SPECIFIED **"
  Connection connection = getConnection(statementLog);  connection: "com.mysql.jdbc.JDBC4Connection@7ea9e1e2"
  stmt = handler.prepare(connection);  connection: "com.mysql.jdbc.JDBC4Connection@7ea9e1e2"
  handler.parameterize(stmt);  handler: RoutingStatementHandler@1552  stmt: "com.mysql.jdbc.JDBC42PreparedStatement@662b4c69: SELECT id,name,age FROM st
  return stmt;
}
```

图10　PrepareStatement处理过程

在PreparedStatementHandler进行参数化时，会将参数化的职责交给DefaultParameterHandler处理。如下图11中的代码所示，我们主要关注红线部分，首先会获取ParameterMapping对应的TypeHander，如前文所述，获取到的是UnknownTypeHandler，然后会通过setParameter方法，将参数id替换成对应的值。

```
public void setParameters(PreparedStatement ps) throws SQLException {
  ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());
  List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
  if (parameterMappings != null) {
    MetaObject metaObject = parameterObject == null ? null : configuration.newMetaObject(parameterObject);
    for (int i = 0; i < parameterMappings.size(); i++) {
      ParameterMapping parameterMapping = parameterMappings.get(i);
      if (parameterMapping.getMode() != ParameterMode.OUT) {
        Object value;
        String propertyName = parameterMapping.getProperty();
        if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params
          value = boundSql.getAdditionalParameter(propertyName);
        } else if (parameterObject == null) {
          value = null;
        } else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
          value = parameterObject;
        } else {
          value = metaObject == null ? null : metaObject.getValue(propertyName);
        }
        TypeHandler typeHandler = parameterMapping.getTypeHandler();
        JdbcType jdbcType = parameterMapping.getJdbcType();
        if (value == null && jdbcType == null) jdbcType = configuration.getJdbcTypeForNull();
        typeHandler.setParameter(ps,  i: i + 1, value, jdbcType);
      }
    }
  }
}
```

在 Typehandler 的流程里，首先会进入 BaseTypeHandler，然后在具体设置时，会进入子类的方法。在 UnknownTypeHandler，首先会再次对参数 parameter 进行解析，判断最正确的 TypeHandler 类型，如下图 12 中的代码所示：

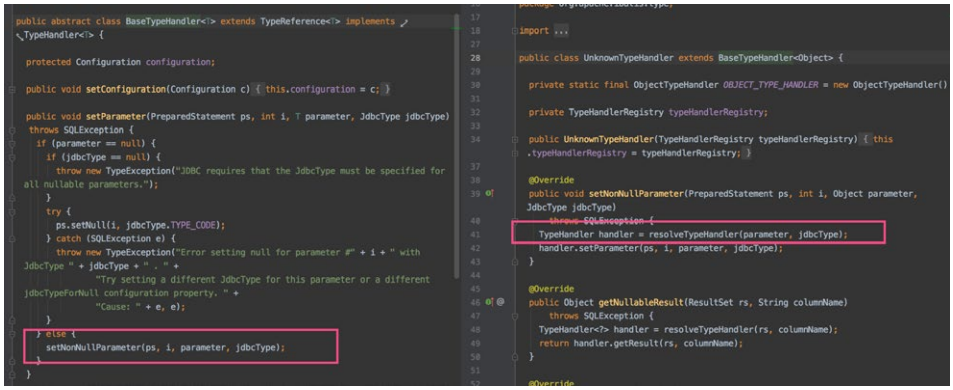


图 12　获取可用 TypeHandler

在 resolveTypeHandler 方法中，因为已知了参数值的类型，通过 Integer 这个 class 在 typeHandlerRegistry 中寻找对应的 TypeHandler，TypeHandlerRegistry 是 MyBatis 启动时内置好的，代表 Java 对象类型和 TypeHandler 的映射关系，有兴趣的同学可以进入这个类详细看下。在这个例子中，我们会直接获取到 IntegerHandler，如下图 13 中的代码所示：

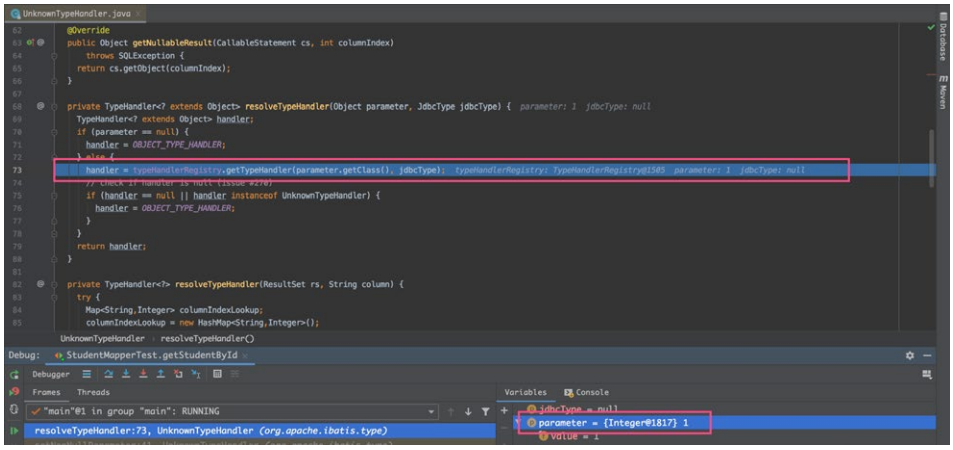


图 13　获取 IntegerHandler

在获取到 IntegerHandler 后，我们就可以使用 IntegerTypeHandler 的 setInt 方法，对 SQL 语句中的参数进行替换。如图 14 中的代码所示，SQL 语句被成功替换：

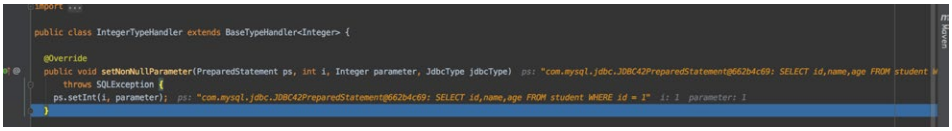


图 14　IntegerHander 值替换

后续就是执行 SQL 并处理返回结果，这就不在本文的讨论范围内了。从上文的分析中，我们可以了解到，在 3.2.3 及以下版本，MyBatis 会忽略 parameterType，在真正进行 SQL 转换时，重新根据 SQL 方法入参类型，然后计算合适的 TypeHandler 处理器，所以本案例中的代码在 3.2.3 版本时，它在运行时是正常的。

以版本 3.2.4 为例，相比版本 3.2.3，MyBatis 构建 SQL 语句过程的变化分析

在前一章节中，我们得知 MyBatis 在运行 SQL 阶段重新计算参数对应的 TypeHandler，然后进行 SQL 参数的替换。那么，在版本 3.2.4 中，MyBatis 做了什么改动，从而导致了原有的使用方式变得不可用呢？从官方的 Release Log 来看，版本 3.2.4 做了这样的一个改动。

> This version builds the binding information during startup and the “parameterType” attribute is used

这个意思是说：parameterType 会在框架初始化阶段阶段就被使用到。我们将分析的重点放在构建阶段，因为负责处理绑定关系的 BoundSql 由配置阶段的 SqlSource 生成，我们主要查看 SqlSource 的构建，在 3.2.4 中发生了什么变化。如图 15 所示，与 3.2.3 不同，3.2.4 首先判断了是否为动态 SQL，在非动态 SQL 情况下，才会将 parameterType java.lang.String 作为参数，传入 SqlSource 的构造方法。

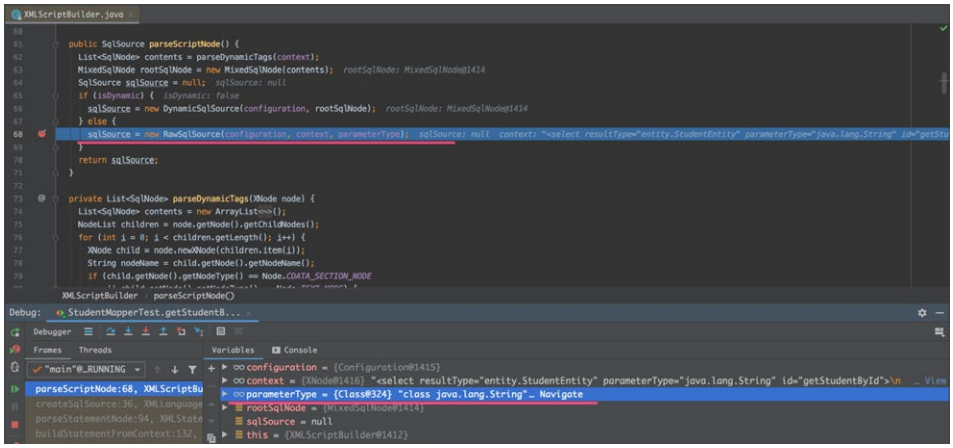


图 15　生成 SqlSource

而后续流程与 3.2.3 一致，因为 parameter 类型为 java.lang.String，在构建 parameterMapping 时，使用的类型就是 java.lang.String。

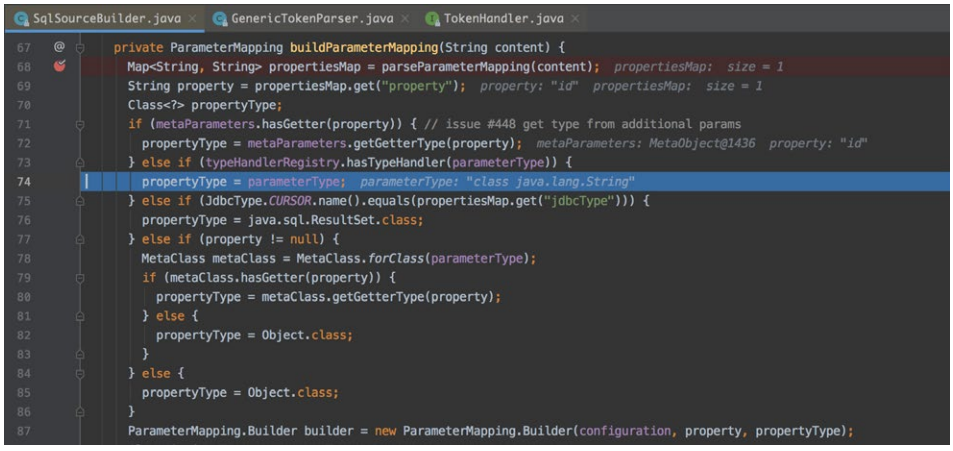


图 16　构建 ParameterMapping 与 3.2.3 版本的差异

因为在框架初始化阶段，SqlSource 的 ParameterMapping 中 id 对应的类型就是 java.lang.String，这就导致在进行 SQL 语句的替换时，获取到的 TypeHandler 是 StringTypeHandler，如下图 17 所示：

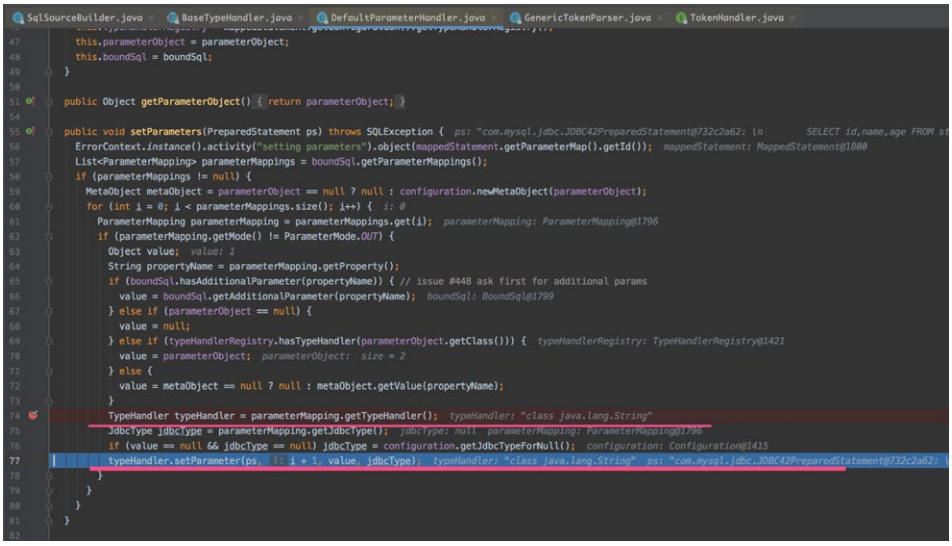


图 17　整数类型的参数获取到了 StringTypeHandler

后面的报错原因就比较好理解了，在调用 StringTypeHandler 的 setString 方法时，报出了`java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String`的错误。

总结

我们总结一下这个案例因：

MyBatis 3.2.3 版本支持 parameterType 和实际参数类型不匹配，在执行 SQL 阶段，动态计算值处理器类型。在大版本升级 2 个版本号后，parameterType 实际的类型开始生效，使用对应这个类型的 TypeHandler 对 SQL 进行参数替换，会导致 Mapper 方法中的参数和 XML 中的 parameterType 不匹配时，进而会出现类型转换报错。

这一段排查的经历，对自己后续编写代码及在系统上线时也有一些启发，主要包括以下几个方面：

- 在 inf-bom 升级时，需要线下进行全面回归，要避免框架存在不兼容的用法，不然的话，就容易导致线上错误。
- 开发同学可以检查自己系统内的 MyBatis 版本，如果是 3.2.4 以下，需要全面检查下现在的 Mapper 文件里对于 parameterType 的使用和 Mapper 方法中实际的参数类型是否一致，避免升级到 3.2.4 及以上版本时发生转型报错。如果有不匹配的情况存在，需要进行修正或者不使用 parameterType，让 MyBatis 在运行 SQL 时自动计算对应的类型。
- 可以考虑使用 MyBatis-Generator 来自动生成 XML 和 Mapper 文件，毕竟是专业团队在维护，稳定性相对来说会更好一些，同时能够避免手动修改 XML 文件带来的误操作。
- 可以主动关注强依赖的一些开源框架的 Release Log，不要错过了重要的信息。

参考资料

带你一步一步手撕 MyBatis 源码加手绘流程图——构建部分
带你一步一步手撕 MyBatis 源码加手绘流程图——执行部分
MyBatis 源码解析（三）—缓存篇
面试官问你 MyBatis SQL 是如何执行的？把这篇文章甩给他
源码分析（1.4 万字）| MyBatis 接口没有实现类为什么可以执行增删改查
MyBatis/MyBatis-3/Comparing changes

作者简介

凯伦，2016 年校招加入美团，后端开发工程师。

招聘信息

电子发票技术团队负责美团发票平台建设和相关创新探索工作，我们的宗旨是提升美团用户在美团各业务场景下的开票体验，同时赋能商家提升发票管理效率。欢迎志同道合的小伙伴加入，通过技术的力量为亿万用户提升电子发票服务体验。感兴趣的同学可投递简历至：tech@meituan.com（邮件标题注明：电子发票业务）

复杂环境下落地 Service Mesh 的挑战与实践

作者：继东　薛晨　业祥　张昀

导读

在私有云集群环境下建设 Service Mesh ，往往需要对现有技术架构做较大范围的改造，同时会面临诸如兼容困难、规模化支撑技术挑战大、推广困境多等一系列复杂性问题。本文会系统性地讲解在美团在落地 Service Mesh 过程中，我们面临的一些挑战及实践经验，希望能对大家有所启发或者帮助。

一、美团服务治理建设进展

1.1 服务治理发展史

首先讲一下 OCTO，此前美团技术团队公众号也分享过很相关的文章，它是美团标准化的服务治理基础设施，现应用于美团所有事业线。OCTO 的治理生态非常丰富，性能及易用性表现也很优异，可整体概括为 3 个特征：

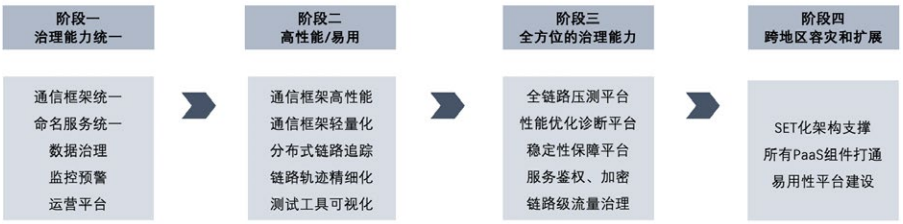
1. 属于公司级的标准化基础设施。技术栈高度统一，覆盖了公司 90% 以上的应用，日均调用量达数万亿次。
2. 经历过较大规模的技术考验。覆盖数万个服务、数十万个节点。
3. 治理能力丰富。协同周边治理生态，实现了 SET 化、链路级复杂路由、全链路压测、鉴权加密、限流熔断等治理能力。

回顾美团服务治理体系的发展史，历程整体上划分为四个阶段：

1. **第一阶段是基础治理能力统一**。实现通信框架及注册中心的统一，由统一的治理平台支撑节点管理、流量管理、监控预警等运营能力。
2. **第二阶段重点提升性能及易用性**。4 核 4GB 环境下使用 1KB 数据进行 echo 测试，QPS 从 2 万提升至接近 10 万，99 分位线 1ms；也建设了分布式链

路追踪、分阶段耗时精细埋点等功能。

3. **第三阶段是全方位丰富治理能力**。落地了全链路压测平台、性能诊断优化平台、稳定性保障平台、鉴权加密等一系列平台，也实现了链路级别的流量治理，如全链路灰度发布等。
4. **第四阶段是建设了跨地域的容灾及扩展能力**。在每天数千万订单量级下实现了单元化，也实现了所有 PaaS 层组件及核心存储系统的打通。



1.2 服务治理体系的困境

目前，美团已具备了较完善的治理体系，但仍有较多的痛点及挑战。大的背景是公司业务蓬勃发展，业务愈发多元化，治理也愈发精细化，这带来了较多新的问题：

1. 业务与中间件强耦合，制约彼此迭代。当中间件引入 Bug，可能成百上千、甚至数千个业务需要做配合升级，中间件的新特性也依赖业务升级后才能使用，成本很高。
2. 中间件版本碎片化严重。发布出去的组件基本托管在业务侧，很难统一进行管控，这也频繁造成业务多类的问题。
3. 异构体系融合难。新融入公司的技术体系往往与美团不兼容，治理体系打通的成本很高，难度也很大。此前，美团与大众点评打通治理，不包含业务迁移，就历时 1 年半的时间；近期，摩拜使用的 gRPC 框架也无法与系统进行通信，但打通迫在眉睫。
4. 非 Java 语言治理体系能力弱，多个主流语言无官方 SDK。多元业务场景下，未来多语言也是个趋势，比如在机器学习领域，Python 语言不太可能被其他语言完全代替。

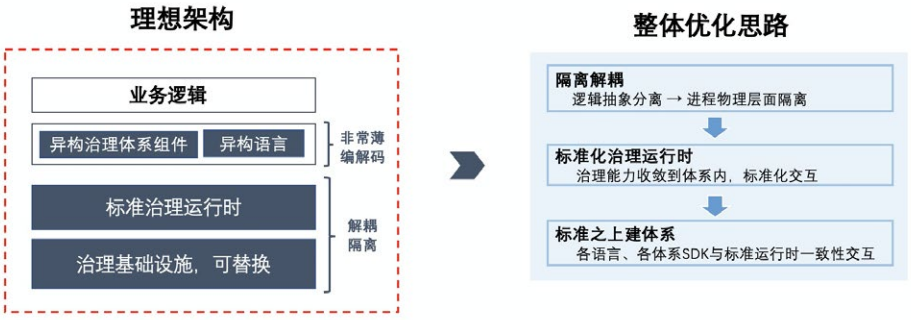
二、服务治理体系优化的思路与挑战

2.1 优化思路

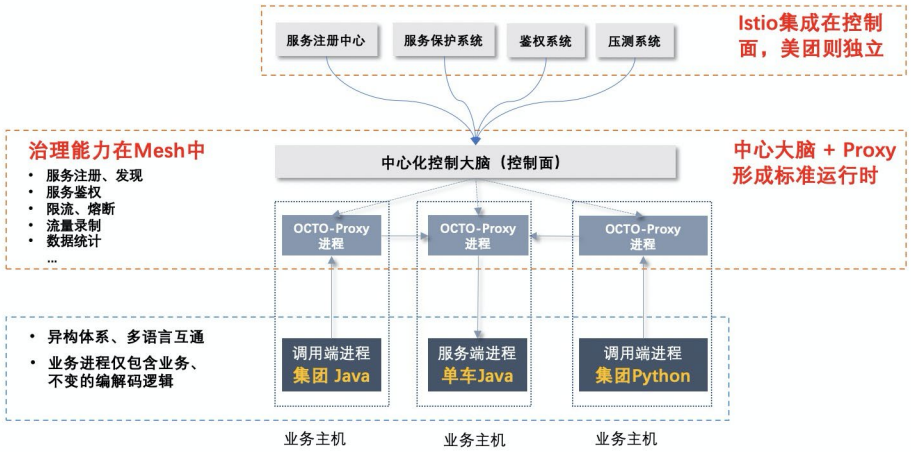
总结来看，OCTO 在服务层实现了统一抽象来支撑业务发展，但它并未解决这层架构可以独立演进的问题。

1.2 节中问题 1 与问题 2 的本质是“耦合”，问题 3 与问题 4 的本质是“缺乏标准服务治理运行时”。在理想的架构中，异构语言、异构治理体系可以共用统一的标准治理运行时，仅在业务使用的 SDK 部分有轻微差异。

所以，我们整体的优化思路分为三步：**隔离解耦，在隔离出的基础设施层建设标准化治理运行时，标准之上建体系**。



上述解决方案所对应的新架构模式下，各业务进程会附属一个 Proxy 进程，SDK 发出以及接收的流量均会被附属的 Proxy 拦截。像限流、路由等治理功能均由 Proxy 和中心化的控制大脑完成，并由控制面对接所有治理子系统集成。这种模式下 SDK 很轻薄，异构的语言、异构的治理体系就很容易互通，从而实现了物理解耦，业界将这种模式称为 Service Mesh（其中 Proxy 被称为数据面、中心化控制大脑被称为控制面）。



2.2 复杂性挑战

美团在实践中所面临的复杂性划主要包括以下 4 类：

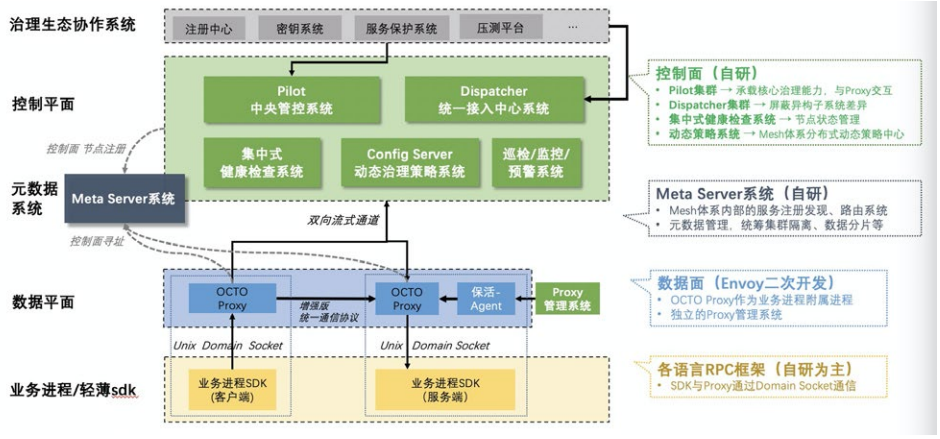
1. **兼容性**：技术改造涉及范围较大，一方面需要通过保证现有通信方式及平台使用方式不变，从而来保障业务研发效率，另一方面也要解决运行载体多样性、运维体系兼容等问题。
2. **异构性**：第一是多语言互通问题；第二是打通治理体系内的众多治理子系统，像服务鉴权、注册中心等系统的存储及发布订阅机制都是不同的；第三是快速打通新融入公司的异构治理体系。
3. **大规模支撑**：出于性能方面考虑，开源 Istio 等产品不宜直接应用于大规模的生产环境，美团控制面需具备百万级链接下高吞吐、低延迟、高精确的系统能力。
4. **重交易型业务容错性低**：交易型业务场景下，业务对 Service Mesh 的性能、稳定性往往持怀疑态度；美团基础架构团队也强调在业务价值导向下，基于实际业务价值进行运营推广，而不是采用从上至下的偏政策性推广方式。

三、美团落地 Service Mesh 的解决方案

3.1 整体架构

美团采用数据面基于 Envoy 二次开发、控制面自研为主、SDK 协同升级的方案（内部项目名是 OCTO Mesh ）。架构简介如下：

- 各语言轻薄的 SDK 与 Proxy 通过 UDS（Unix Domain Socket）交互，主要出发点是考虑到相比透明流量劫持，UDS 性能与可运维性更好。
- 控制面与 Proxy 通过双向流通信，控制面与治理生态的多个子系统交互，并将计算处理过的治理数据及策略数据下发给 Proxy 执行，协同配合完成路由、限流等所有核心治理功能。
- 控制面内部的 5 个模块都是自研的独立服务。
  - Pilot 承载核心治理能力，与 Proxy 直接交互。
  - Dispatcher 负责屏蔽异构子系统差异。
  - 集中式健康检查管理节点状态。
  - Config Server 管理 Mesh 体系内相关的策略，并将 Pilot 有状态的部分尽量迁移出来。
  - 监控及巡检系统负责提升稳定性。
- 自研了的 Meta Server 系统实现 Mesh 体系内部的节点注册和寻址，通过管理控制面与数据面的链接关系，也实现了按事业群隔离、水平扩展等能力。



3.2 兼容性解决方案

兼容性的目标及特征用一句话来总结就是：业务接入无感知。为此，我们做了以下三件事情：

(1) 与现有基础设施及治理体系兼容

- 将 Service Mesh 与 OCTO 深度打通，确保各治理子系统的使用方式都不变。
- 运行载体方面，同时支持容器、虚拟机、物理机。
- 打通运维体系，保证服务治理基础设施处于可管理、可监测的状态。

(2) 协议兼容

- 服务间调用往往是多对多的关系，一般调用端与服务端无法同时升级，为支持 Mesh 与非 Mesh 的互通，增强后的协议对业务完全透明。
- 与语义相关的所有内容（比如异常等），均在 SDK 与 Proxy 之间达成共识，保证兼容。
- 无法在控制面及数据面中实现的能力，在 SDK 中执行并通过上下文传递给 Proxy，保障功能完全对齐，当然这种情况应该尽量避免的。

(3) Mesh 与非 Mesh 模式的无缝切换

- 基于 UDS 通信必然需要业务升级一次 SDK 版本，我们在 2020 年初时预先发布早做部署，确保当前大部分业务已经升级到新版本，但默认仍是不开启 Mesh 的状态。
- 在可视化平台上面通过开关操作，几乎无成本实现从 Mesh 模式与非 Mesh 模式的切换，并具备实时生效的能力。

3.3 异构性解决方案

异构性的目标及特征用一句话总结就是：标准化服务治理运行时。具体可拆分为 3 个子目标：

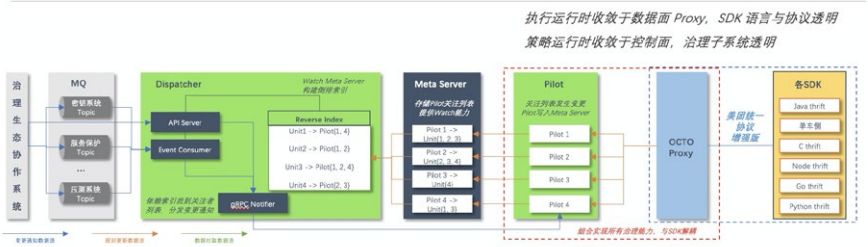
- 标准化美团内部 6 种语言的治理体系。
- 架构层面由控制面统一对接各个治理子系统，屏蔽注册中心、鉴权、限流等系

统具体实现机制的异构性。

- 支持摩拜及未来新融入公司的异构治理体系与公司整体的快速融合。

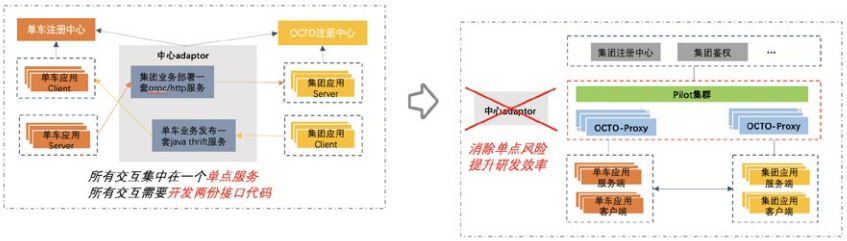
针对上述 3 个子目标，我们所采取的方案如下：

- 将数据面 + 控制面定义为标准化的服务治理运行时，在标准运行时内打通所有治理能力。
- 建设统一接入中心系统 Dispatcher ，并由其对接并屏蔽治理子系统的异构性，从而实现外部系统的差异对 Pilot 透明；下图中 Dispatcher 与 Pilot 直接交互，Meta Server 的作用是避免广播降低冗余。
- 重构或从零建设 SDK，目前使用的 6 种语言 SDK 均已落地并使用。
- 异构语言、异构体系均使用增强的统一协议交互，实现互通。



通过 Service Mesh 实现体系融合的前后对比如下：

- 引入 Service Mesh 前，单车向公司的流量以及公司向单车的流量，均是由中间的 adaptor 单点服务承接。除稳定性有较大隐患外，所有交互逻辑均需要开发两份代码，效率较差。
- 引入 Service Mesh 后，在一套服务治理设施内打通并直接交互，消除了中心 adaptor 带来的稳定性及研发效率方面的缺陷；此外整个打通在 1 个月内完成，异构体系融合效率很高。



通过上述方案，针对异构性方面取得了较好的效果：

- 标准化 6 种语言治理体系，非 Java 语言的核心治理能力基本对齐 Java；新语言也很容易融入，提供的官方 Python 语言、Golang 语言的通信框架新版本（依托于 OCTO Mesh），开发成本均控制在 1 个月左右。
- 支持异构治理子系统通过统一接入中心快速融入，架构简洁、扩展性强。
- 支持异构治理体系快速融合并在单车侧落地，异构治理体系打通成本也从 1.5 年降低到 1 个月。

3.4 规模化解决方案

3.4.1 开源 Istio 问题分析

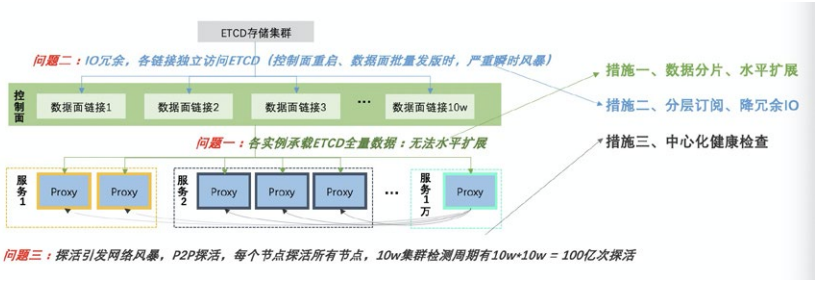
规模化的目标及特征用一句话总结是：**具备支撑数万服务、百万节点体量的系统能力，并支持水平扩展**。挑战主要有 3 个：

- 美团体量是最流行开源产品 Istio 上限的上千倍。
- 极高的实时性、准确性要求；配置下发错误或丢失会直接引发流量异常。
- 起步较早，业界参考信息很少。

经过对 Istio 架构进行深入分析，我们发现核心问题聚焦在以下 3 个瓶颈点：

- 每个控制面实例有 ETCD 存储系统的全部数据，无法水平扩展。
- 每个 Proxy 链接相当于独立与 ETCD 交互，而同一个服务的 Proxy 请求内容都相同，独立交互有大量的 I/O 冗余。当 Proxy 批量发版或网络抖动时，瞬时风暴很容易压垮控制面及 ETCD。

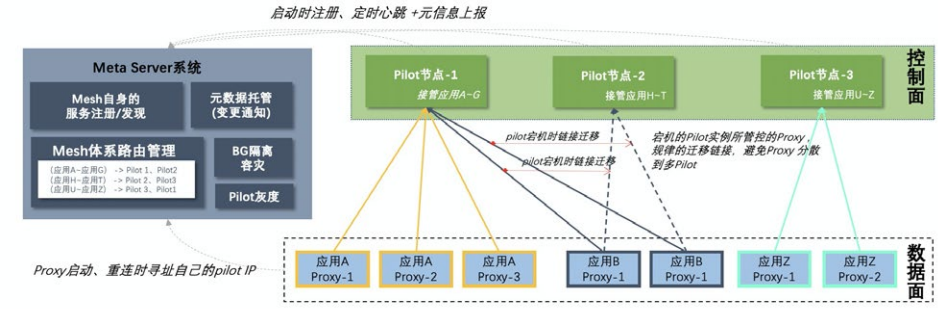
- 每个节点都会探活所有其他节点。10 万节点规模的集群，1 个检测周期有 100 亿次探活，会引发网络风暴。



3.4.2 措施一：横向数据分片

针对 Istio 控制面各实例承载全集群数据的问题，对应的措施是通过横向逻辑数据分片支持扩展性，具体方案设计如下：

- Proxy 启动时会去向 Meta Server 系统请求需要连接的 Pilot IP，Meta Server 将相同服务的 Proxy 尽量落到同一个控制面节点（内部策略更为复杂，还要考虑地域、负载等情况），这样每个 Pilot 实例按需加载而不必承载所有数据。
- 控制面节点异常或发布更新时，其所管理的 Proxy 也会有规律的迁移，恢复后在一定时间后还会接管其负责的 Proxy，从而实现了会话粘滞，也就实现逻辑上面的数据分片。

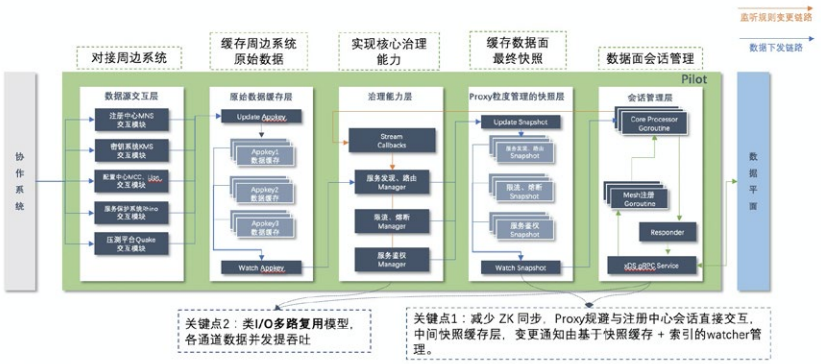


通过管理链接关系实现了按事业群隔离、按服务灰度等平台能力，最关键的还是解决了 Mesh 体系水平扩展的问题。

3.4.3 措施二：纵向分层订阅

针对 Istio 独立管理各 Proxy 链接的 I/O 冗余问题，对应的措施是通过分层订阅减少冗余 I/O。Proxy 不直接与存储等系统对接，而是在中间经过一系列的处理，关键点有两个：

- 关键点 1：基于快照缓存 + 索引的机制来减少 ZK watcher 同步。以注册中心为例，常规实现方式下，如果每个 Proxy 关注 100 个节点，1 万个节点就会注册 100 万个 watcher，相同服务的 Proxy 所关注内容是相同的，另外不同服务 Proxy 所关注的也有很多交集，其中包含大量的冗余。分层订阅模式下，Proxy 不与注册中心直接交互，通过中间的快照缓存与分层，确保每个 Pilot 实例中 ZK 相同路径的监听最多只用 1 个 watcher，获取到 watcher 通知后，Pilot 根据内部的快照缓存 + 索引向所有关注者分发，大大降低了冗余。
- 关键点 2：治理能力层及会话管理层实现了类似于 I/O 多路复用能力，通过并发提升吞吐。

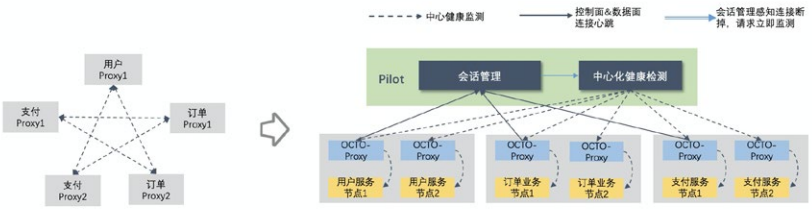


结果方面有效应对了网络抖动或批量发版的瞬间风暴压力，压测单 Pilot 实例可以承载 6 万以上的链接，时延 TP99 线 < 2.3ms、数据零丢失。

3.4.4 措施三：集中式健康检测

针对大规模集群内指数级膨胀的节点间健康监测次数，对应的措施是摒弃了 P2P 检测模式，我们参考并优化了 Google 的 Traffic Drector 中心化管理的健康检测模式。这种模式下检测次数大大减少，一个周期内 10 万节点集群的检测次数，从 100 亿次下降到 10 万次。

此外，当 Pilot 感知到 Proxy 异常时，会立即通知中心化健康检测系统启动检测，而不是等待检测周期窗口的到来，这可以有效提升业务调用的成功率。

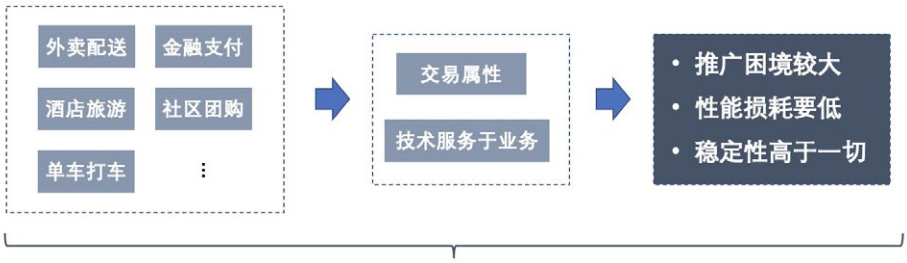


3.5 交易型场景困境下的解决方案

3.5.1 业务属性分析

美团内部业务线较多，包括外卖、配送、酒店、旅游、单车、团购等，其中绝大多数业务都带有交易属性，交易链路上一个流量异常就可能影响到订单。业务系统对新技术领域的探索往往比较慎重，期望在新技术充分验证后再启动试点，所以除小语种及亟待与公司打通的单车业务外，推广的难度是非常大的。此外，基础架构部秉承“以客户为中心”的原则，研发、运维、测试人员均是我们的“客户”，所以技术升级会重点从业务价值入手，并非简单依靠从上至下的政策推动力。

所以，我们对外的承诺是：**通信足够快、系统足够稳定、接入足够平滑高效**。



3.5.2 精细化运营体系建设

针对推广的困境，我们首先做了两件事情：

- 寻找具备强诉求的业务试点，客观来说，美团技术栈内这类业务数量非常有限。
- 寻求标杆核心业务试点，充分验证后推广给其他业务，但效果并不理想，与业务稳定性的诉求并不匹配。

针对上述困境，我们进行深度思考后建立了一个精细化的运营体系：

- 服务接入 Mesh 前。基于 SOA 分级将服务划分为非核心与核心两类，先针对非核心服务以及所有服务的线下环境进行重点突破，实现了在广泛的业务场景下，全面且充分的验证系统能力。
- 服务接入 Mesh 中。运营系统通过校验 SDK 版本、运行时环境等信息，自动筛选出满足条件的服务，业务同学只需要在平台上做（1）开启开关、（2）选择节点（3）指定 Mesh 流量比例三个步骤，就完成了到 Mesh 模式的切换，不需代码改造也不需发布服务，整个过程基本在 1 分钟左右完成；此外，通过与 IM 工具深度联动，提升了推广与数据运营的效率。
- 服务接入 Mesh 后。一方面，业务侧包括架构侧的运营有详细的数据指标做对比参考；另一方面，运营系统支持预先设置稳定性策略并做准实时的检测，当某个接入服务 Mesh 模式异常时，即时自动切换回非 Mesh 模式。

运营体系具备“接入过程无感”、“精细化流量粒度灰度”、“异常自动回滚恢复”三个核心能力，在运营体系建设后推广运营较为顺利，目前线上接入的 600+ 服务、线下接入的 3500+ 服务中，90% 以上是依托运营平台接入 Mesh 的。

3.5.3 通信性能优化

在性能损耗优化这个方向，除使用 UDS 规避网络栈外，我们也通过增量聚合下发、序列化优化两个措施减少不必要的解包，提升了通信性能。

经过压测，去除非核心功能在 2 核 4G 环境用 1KB 数据做 echo 测试，QPS 在 34000 以上，一跳平均延迟 0.207ms，时延 TP99 线 0.4ms 左右。

3.5.4 流量多级保护

美团落地 Service Mesh 在稳定性保障方面建设投入较多，目前尚无 Service Mesh 引发的故障，具体包含三个方面：

- 首先做了流量多级保护
  - 一方面，当 Proxy 不可用时，流量会自动 fallback 到非 Mesh 模式；另一方面，支持最精细支持按单节点的 1/1000 比例灰度。下图是具体的交互流程，当然，这两个特性与 Service Mesh 的最终形态是冲突的，只是作为系统建设初期优先保证业务稳定性的过渡性方案，长期来看必然是要去除的（包括美团一些核心服务已经完全去除）。
  - 基于 FD 迁移 + SDK 配合协议交互，实现 Proxy 无损热重启的能力。
- 控制面下发错误配置比停发配置的后果更为严重，我们建设了应用层面及系统层面的周期巡检，从端到端的应用视角验证正确性，避免或减少因变更引发的异常。
- 系统交互方面，通过限流、熔断对中心化控制面做服务保护；系统内柔性可用，当控制面全部异常时，缓存机制也能协助 Proxy 在一定时间内可用。

四、总结

本文系统性的介绍美团在 Service Mesh 落地进程中面临的“兼容性”、“异构性”、“规模化”、“交易属性业务容错性低”这四类复杂性挑战，针对上述挑战，我们也详细介绍了大规模私有云集群场景下的优化思考及实践方案。

基于上述实践，目前美团线上落地服务数超过 600，线下服务数超过 3500+，初步验证了模式的可行性。短期价值方面，我们支持了摩拜等异构治理体系的快速融合、多语言治理能力的统一；长期价值仍需在实践中继续探索与验证，但在标准化服务治理运行时并与业务解耦、中心化管控下更丰富的治理能力输出两个方面，是非常值得期待的。

作者简介

继东、薛晨、业祥、张昀，均来自美团基础技术部 – 基础架构部。

招聘信息

基础技术部 – 基础架构部 – 中间件研发中心 – 服务框架组涉及领域主要包括 RPC 通信框架、Service Mesh、服务治理门户、Set 化、流程引擎系统 Gravity 等。感兴趣的同学可投递简历至：tech@meituan.com（邮件主题请注明：基础架构部）。

C++ 服务编译耗时优化原理及实践

作者：周磊

一、背景

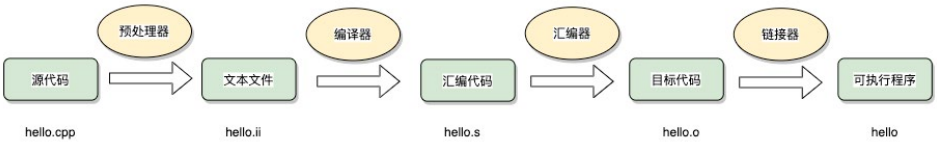
大型 C++ 工程项目，都会面临编译耗时较长的问题。不管是开发调试迭代、准入测试，亦或是持续集成阶段，编译行为无处不在，降低编译时间对提高研发效率来说具有非常重要意义。

美团搜索与 NLP 部为公司提供基础的搜索平台服务，出于性能的考虑，底层的基础服务通过 C++ 语言实现，其中我们负责的深度查询理解服务（DeepQueryUnderstanding，下文简称 DQU）也面临着编译耗时较长这个问题，整个服务代码在优化前编译时间需要二十分钟左右（32 核机器并行编译），已经影响到了团队开发迭代的效率。基于这样的背景，我们针对 DQU 服务的编译问题进行了专项优化。在这个过程中，我们也积累了一些优化的知识和经验，在这里分享给大家。

二、编译原理及分析

2.1 编译原理介绍

为了更好地理解编译优化方案，在介绍优化方案之前，我们先简单介绍一下编译原理，通常我们在进行 C++ 开发时，编译的过程主要包含下面四个步骤：



预处理器：宏定义替换，头文件展开，条件编译展开，删除注释。

- gcc -E 选项可以得到预处理后的结果，扩展名为 .i 或 .ii。

- C/C++ 预处理不做任何语法检查，不仅是因为它不具备语法检查功能，也因为预处理命令不属于 C/C++ 语句（这也是定义宏时不要加分号的原因），语法检查是编译器要做的事情。
- 预处理之后，得到的仅仅是真正的源代码。

编译器：生成汇编代码，得到汇编语言程序（把高级语言翻译为机器语言），该种语言程序中的每条语句都以一种标准的文本格式确切的描述了一条低级机器语言指令。

- gcc -S 选项可以得到编译后的汇编代码文件，扩展名为 .s。
- 汇编语言为不同高级语言的不同编译器提供了通用的输出语言。

汇编器：生成目标文件。

- gcc -c 选项可以得到汇编后的结果文件，扩展名为 .o。
- .o 文件，是按照的二进制编码方式生成的文件。

链接器：生成可执行文件或库文件。

- **静态库**：指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了，其后缀名一般为“.a”。
- **动态库**：在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可执行文件比较小，动态库一般后缀名为“.so”。
- **可执行文件**：将所有的二进制文件链接起来融合成一个可执行程序，不管这些文件是目标二进制文件还是库二进制文件。

2.2 C++ 编译特点

（1）每个源文件独立编译

C/C++ 的编译系统和其他高级语言存在很大的差异，其他高级语言中，编译单元是整个 Module，即 Module 下所有源码，会在同一个编译任务中执行。而在 C/C++ 中，编译单元是以文件为单位。每个 .c/.cc/.cxx/.cpp 源文件是一个独立的编译单

元，导致编译优化时只能基于本文件内容进行优化，很难跨编译单元提供代码优化。

（2）每个编译单元，都需要独立解析所有包含的头文件

如果 N 个源文件引用到了同一个头文件，则这个头文件需要解析 N 次（对于 Thrift 文件或者 Boost 头文件这类动辄几千上万行的头文件来说，简直就是“鬼故事”）。

如果头文件中有模板（STL/Boost），则该模板在每个 cpp 文件中使用时都会做一次实例化，N 个源文件中的 std::vector 会实例化 N 次。

（3）模板函数实例化

在 C++ 98 语言标准中，对于源代码中出现的每一处模板实例化，编译器都需要去做实例化的工作；而在链接时，链接器还需要移除重复的实例化代码。显然编译器遇到一个模板定义时，每次都去进行重复的实例化工作，进行重复的编译工作。此时，如果能够让编译器避免此类重复的实例化工作，那么可以大大提高编译器的工作效率。在 C++ 0x 标准中一个新的语言特性 – 外部模板的引入解决了这个问题。

在 C++ 98 中，已经有一个叫做显式实例化（Explicit Instantiation）的语言特性，它的目的是指示编译器立即进行模板实例化操作（即强制实例化）。而外部模板语法就是在显式实例化指令的语法基础上进行修改得到的，通过在显式实例化指令前添加前缀 extern，从而得到外部模板的语法。

① 显式实例化语法：template class vector。 ② 外部模板语法：extern template class vector。

一旦在一个编译单元中使用了外部模板声明，那么编译器在编译该编译单元时，会跳过与该外部模板声明匹配的模板实例化。

（4）虚函数

编译器处理虚函数的方法是：给每个对象添加一个指针，存放了指向虚函数表的地址，虚函数表存储了该类（包括继承自基类）的虚函数地址。如果派生类重写了虚函数的新定义，该虚函数表将保存新函数的地址，如果派生类没有重新定义虚函数，该

虚函数表将保存函数原始版本的地址。如果派生类定义了新的虚函数，则该函数的地址将被添加到虚函数表中。

调用虚函数时，程序将查看存储在对象中的虚函数表地址，转向相应的虚函数表，使用类声明中定义的第几个虚函数，程序就使用数组的第几个函数地址，并执行该函数。

使用虚函数后的变化：

① 对象将增加一个存储地址的空间（32 位系统为 4 字节，64 位为 8 字节）。② 每个类编译器都创建一个虚函数地址表。③ 对每个函数调用都需要增加在表中查找地址的操作。

（5）编译优化

GCC 提供了为了满足用户不同程度的的优化需要，提供了近百种优化选项，用来对编译时间，目标文件长度，执行效率这个三维模型进行不同的取舍和平衡。优化的方法不一而足，总体上将有以下几类：

① 精简操作指令。② 尽量满足 CPU 的流水操作。③ 通过对程序行为地猜测，重新调整代码的执行顺序。④ 充分使用寄存器。⑤ 对简单的调用进行展开等等。

如果全部了解这些编译选项，对代码针对性的优化还是一项复杂的工作，幸运的是 GCC 提供了从 O0-O3 以及 Os 这几种不同的优化级别供大家选择，在这些选项中，包含了大部分有效的编译优化选项，并且可以在这个基础上，对某些选项进行屏蔽或添加，从而大大降低了使用的难度。

- O0：不做任何优化，这是默认的编译选项。
- O 和 O1：对程序做部分编译优化，编译器会尝试减小生成代码的尺寸，以及缩短执行时间，但并不执行需要占用大量编译时间的优化。
- O2：是比 O1 更高级的选项，进行更多的优化。GCC 将执行几乎所有的不包含时间和空间折中的优化。当设置 O2 选项时，编译器并不进行循环展开以及函数内联优化。与 O1 比较而言，O2 优化增加了编译时间的基础上，提高了

生成代码的执行效率。

- O3：在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。
- Os：主要是对代码大小的优化，通常各种优化都会打乱程序的结构，让调试工作变得无从着手。并且会打乱执行顺序，依赖内存操作顺序的程序需要做相关处理才能确保程序的正确性。

编译优化有可能带来的问题：

① **调试问题**：正如上面所提到的，任何级别的优化都将带来代码结构的改变。例如：对分支的合并和消除，对公用子表达式的消除，对循环内 load/store 操作的替换和更改等，都将会使目标代码的执行顺序变得面目全非，导致调试信息严重不足。

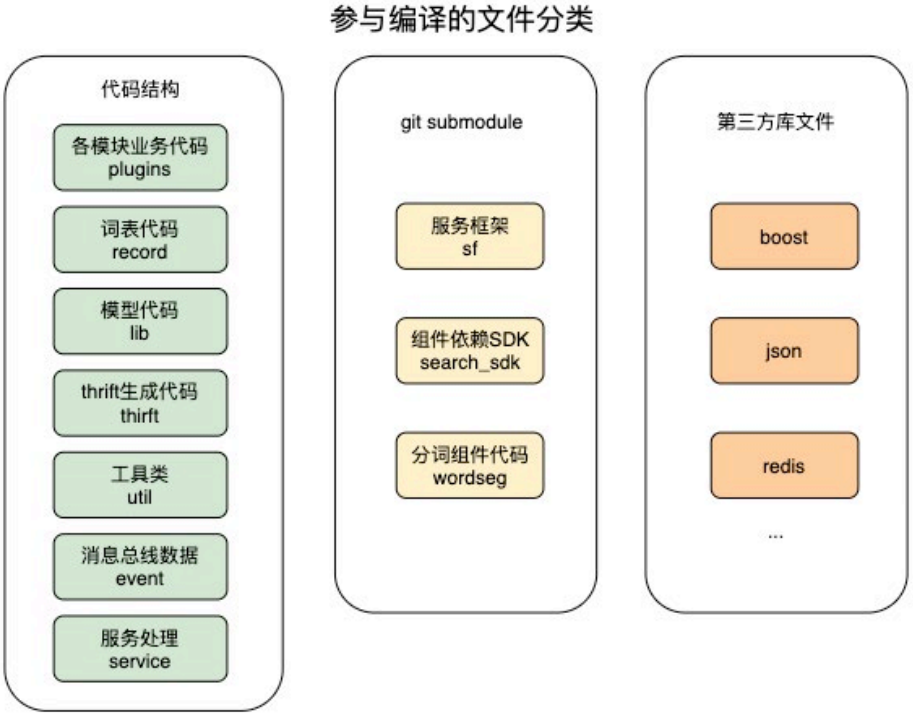
② **内存操作顺序改变问题**：在 O2 优化后，编译器会对影响内存操作的执行顺序。例如：-fschedule-insns 允许数据处理时先完成其他的指令；-fforce-mem 有可能导致内存与寄存器之间的数据产生类似脏数据的不一致等。对于某些依赖内存操作顺序而进行的逻辑，需要做严格的处理后才能进行优化。例如，采用 Volatile 关键字限制变量的操作方式，或者利用 Barrier 迫使 CPU 严格按照指令序执行。

(6)C/C++ 跨编译单元的优化只能交给链接器

当链接器进行链接的时候，首先决定各个目标文件在最终可执行文件里的位置。然后访问所有目标文件的地址重定义表，对其中记录的地址进行重定向（加上一个偏移量，即该编译单元在可执行文件上的起始地址）。然后遍历所有目标文件的未解决符号表，并且在所有的导出符号表里查找匹配的符号，并在未解决符号表中所记录的位置上填写实现地址，最后把所有的目标文件的内容写在各自的位置上，就生成一个可执行文件。链接的细节比较复杂，链接阶段是单进程，无法并行加速，导致大项目链接极慢。

三、服务问题分析

DQU 是美团搜索使用的查询理解平台，内部包含了大量的模型、词表、在代码结构上，包含 20 多个 Thrift 文件，使用大量 Boost 处理函数，同时引入了 SF 框架，公司第三方组件 SDK 以及分词三个 Submodule，各个模块采用动态库编译加载的方式，模块之间通过消息总线做数据的传输，消息总线是一个大的 Event 类，这样这个类就包含了各个模块需要的数据类型的定义，所以各个模块都会引入 Event 头文件，不合理的依赖关系造成这个文件被改动，几乎所有的模块都会重新编译。



每个服务所面临的编译问题都有各自的特点，但是遇到问题的本质原因是类似的，结合编译的过程和原理，我们从预编译展开、头文件依赖以及编译过程耗时 3 个方面对 DQU 服务编译问题进行了分析。

3.1 编译展开分析

编译展开分析就是通过 C++ 的预编译阶段保留的 .ii 文件，查看通过展开后的编译文件大小，具体可以通过在 cmake 中指定编译选型“-save-temps”保留编译中间文件。

```
set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS} -ggdb -Og -fPIC -w
-Wl,--export-dynamic -Wno-deprecated -fpermissive -save-temps")
```

编译耗时的最直接原因就是编译文件展开之后比较大，通过编译展开后的文件大小和内容，通过预编译展开分析能看到文件展开后的文件有 40 多万行，发现有大量的 Boost 库引用及头文件引用造成的展开文件比较大，影响到编译的耗时。通过这个方式能够找到各个文件编译耗时的共性，下图是编译展开后文件大小截图。

```
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/entity_recognition/entity_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/query_correct/querycorrect_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/query_preprocess/query_preprocess_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:50 ./plugins/query_async_call/query_entity_link.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:42 ./services/query_analysis_server.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/platform/query_classification_pack/query_classification_pack.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:49 ./plugins/location_recognition/location_handler.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:44 ./plugins/query_chunk/query_chunk_platform_business.ii
-rw-rw-r-- 1 sankuai sankuai  12M Jun 12 18:51 ./plugins/query_async_call/async_query_manager.ii
```

3.2 头文件依赖分析

头文件依赖分析是从引用头文件数量的角度来看代码是否合理的一种分析方式，我们实现了一个脚本，用来统计头文件的依赖关系，并且分析输出头文件依赖引用计数，用来辅助判断头文件依赖关系是否合理。

(1) 头文件引用总数结果统计

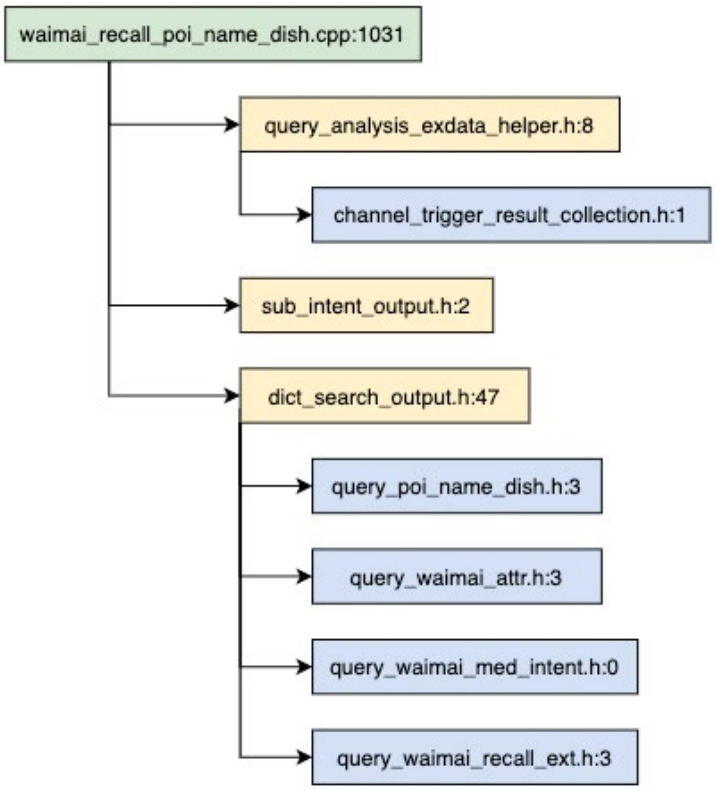
通过工具统计出编译源文件直接和间接依赖的头文件的总个数，用来从头文件引入数量上分析问题。

```
('total', 208934)
('querycorrect_handler.cpp', '\t', 1359)
('recall_strategy.cpp', '\t', 1280)
('query_dqu_hotel_pack.cpp', '\t', 1275)
('processors_handler.cpp', '\t', 1266)
('query_rank_pack.cpp', '\t', 1228)
('query_entity_link.cpp', '\t', 1215)
('query_floating_red_pack.cpp', '\t', 1202)
('async_query_manager.cpp', '\t', 1183)
```

(2) 单个头文件依赖关系统计

通过工具分析头文件依赖关系，生成依赖关系拓扑图，能够直观的看到依赖不合理的地方。

图中包含引用层次关系，以及引用头文件个数。



3.3 编译耗时结果分段统计

编译耗时分段统计是从结果上看各个文件的编译耗时以及各个编译阶段的耗时情况，这个是直观的一个结果，正常情况下，是和文件展开大小以及头文件引用个数是正相关的，cmake 通过指定环境变量能打印出编译和链接阶段的耗时情况，通过这个数据能直观的分析出耗时情况。

```
set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE "${CMAKE_COMMAND} -E
time")
set_property(GLOBAL PROPERTY RULE_LAUNCH_LINK "${CMAKE_COMMAND} -E
time")
```

编译耗时结果输出：

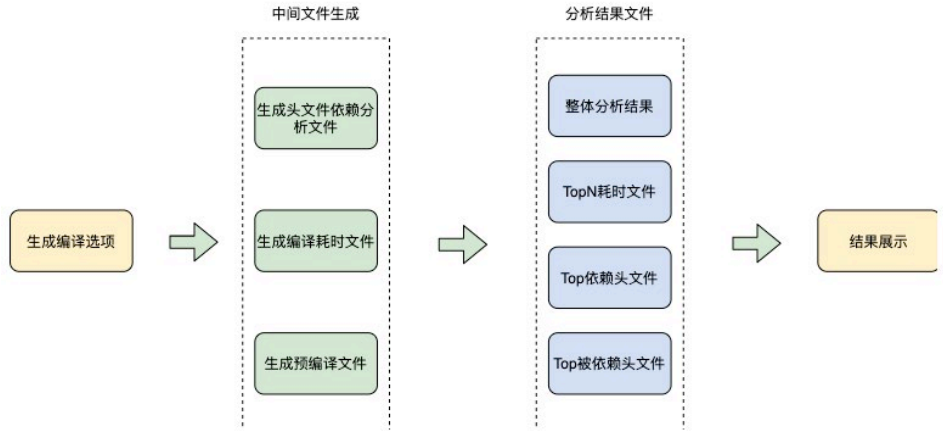
```
Elapsed time: 17 s. (time), 0 s. (clock)
[ 32%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/router_config.cpp.o
Elapsed time: 4 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/rewrite_graph.cpp.o
Elapsed time: 14 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/rewrite_pack_info.cpp.o
Elapsed time: 62 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/util.cpp.o
Elapsed time: 12 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/channel_trigger_common.cpp.o
Elapsed time: 0 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/query_analysis_config_manager.cpp.o
Elapsed time: 7 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/movie_name.cpp.o
Elapsed time: 1 s. (time), 0 s. (clock)
[ 33%] Building CXX object common/platform/CMakeFiles/platform_lib.dir/common/channel_utils.cpp.o
Elapsed time: 32 s. (time), 0 s. (clock)
```

3.4 分析工具建设

通过上面的工具分析能拿到几个编译数据：

① 头文件依赖关系及个数。 ② 预编译展开大小及内容。 ③ 各个文件编译耗时。 ④ 整体链接耗时。 ⑤ 可以计算出编译并行度。

通过这几个数据的输入我们考虑可以做个自动化分析工具，找出优化点以及界面化展示。基于这个目的，我们建设了全流程自动化分析工具，能够自动分析耗时共性问题以及 TopN 耗时文件。分析工具处理流程如下图所示：



(1) 整体统计分析效果

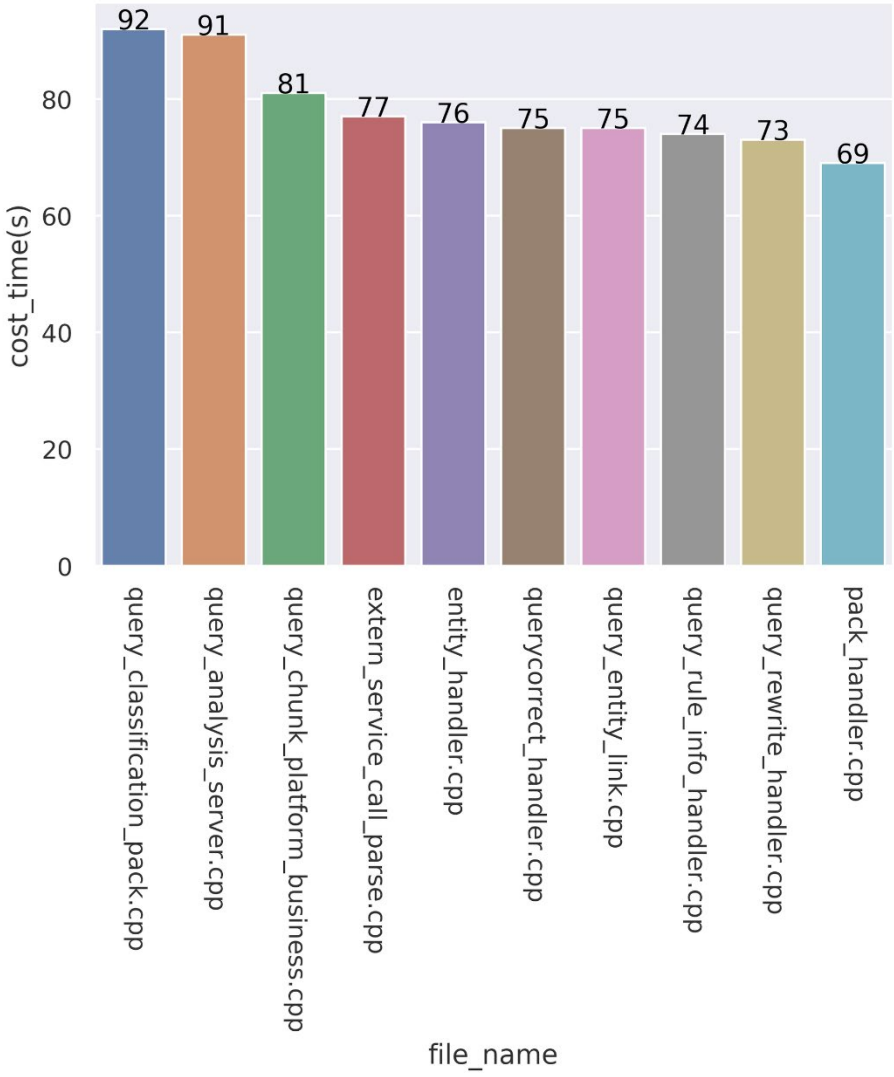
	cost_time(s)	file_compile_size(M)	file_name	include_h_nums	top_h_files_info
0	92	10.863007	query_classification_pack.cpp	1434	[(channel_utils.h, 763)]
1	91	10.788443	query_analysis_server.cpp	1020	[(signal_common_resource.h, 488)]
2	81	10.540730	query_chunk_platform_business.cpp	1038	[(record_search.h, 422)]
3	77	10.198677	extern_service_call_parse.cpp	677	[(platform_query_analysis_event.h, 217)]
4	76	10.832691	entity_handler.cpp	1099	[(record_factory.h, 419)]
5	75	10.839204	querycorrect_handler.cpp	1375	[(resourcemanager.h, 475)]
6	75	10.544925	query_entity_link.cpp	1205	[(address_ner.h, 509)]
7	74	6.678957	query_rule_info_handler.cpp	1022	[(record_search.h, 422)]
8	73	9.101572	query_rewrite_handler.cpp	981	[(resourcemanager.h, 475)]
9	69	9.786721	pack_handler.cpp	1453	[(channel_utils.h, 763)]
10	69	10.463621	location_handler.cpp	818	[(landmark_feature_utils.h, 200)]
11	66	8.549806	query_analysis_event.cpp	728	[(query_analysis_event.h, 139)]
12	66	10.535795	async_query_manager.cpp	1203	[(address_ner.h, 509)]
13	63	10.558627	query_classification_handler.cpp	1204	[(channel_utils.h, 763)]
14	61	10.508329	query_travel_intention.cpp	1153	[(record_search.h, 422)]
15	58	10.313571	query_substring_handler.cpp	802	[(resourcemanager.h, 475)]

具体字段说明：

① cost_time 编译耗时，单位是秒。② file_compile_size，编译中间文件大小，单位是 M。③ file_name，文件名称。④ include_h_nums，引入头文件个数，单位是个。⑤ top_h_files_info，引入最多的 TopN 头文件。

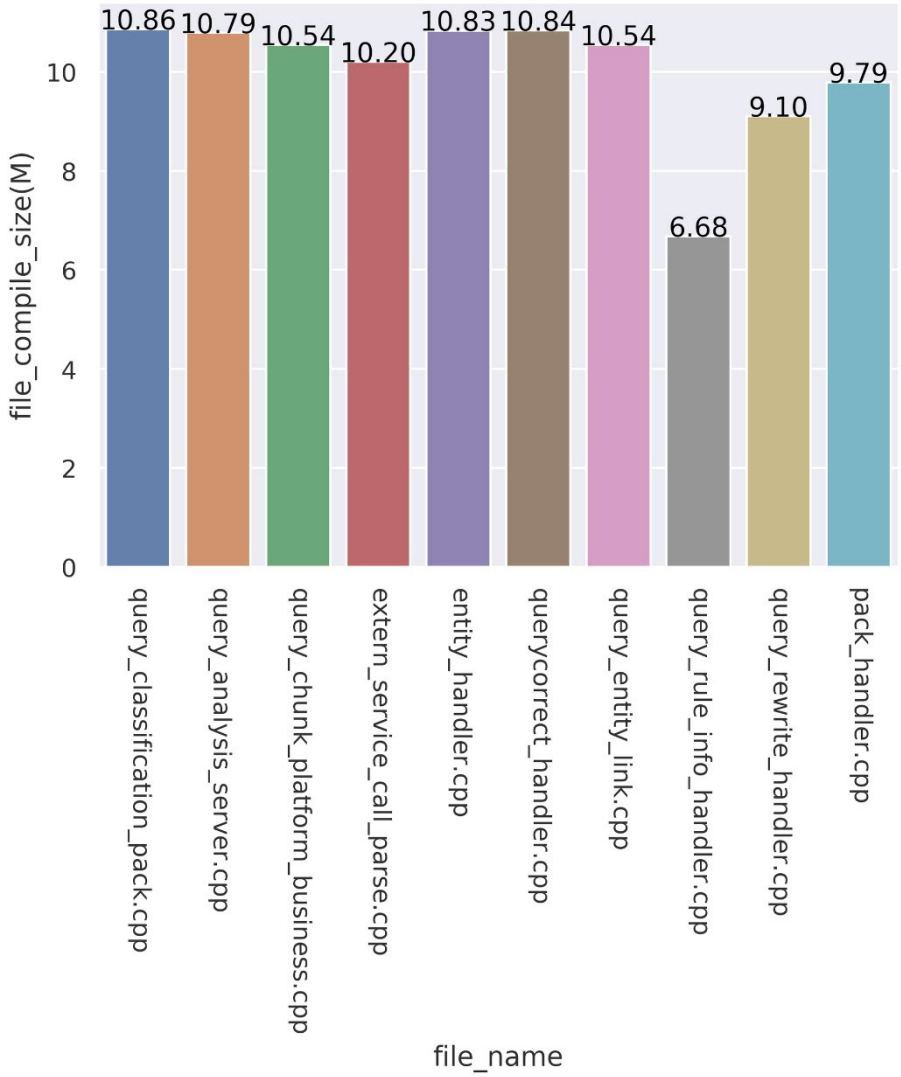
(2) Top10 编译耗时文件统计

用来展示统计编译耗时最久的 TopN 文件，N 可以自定义指定。

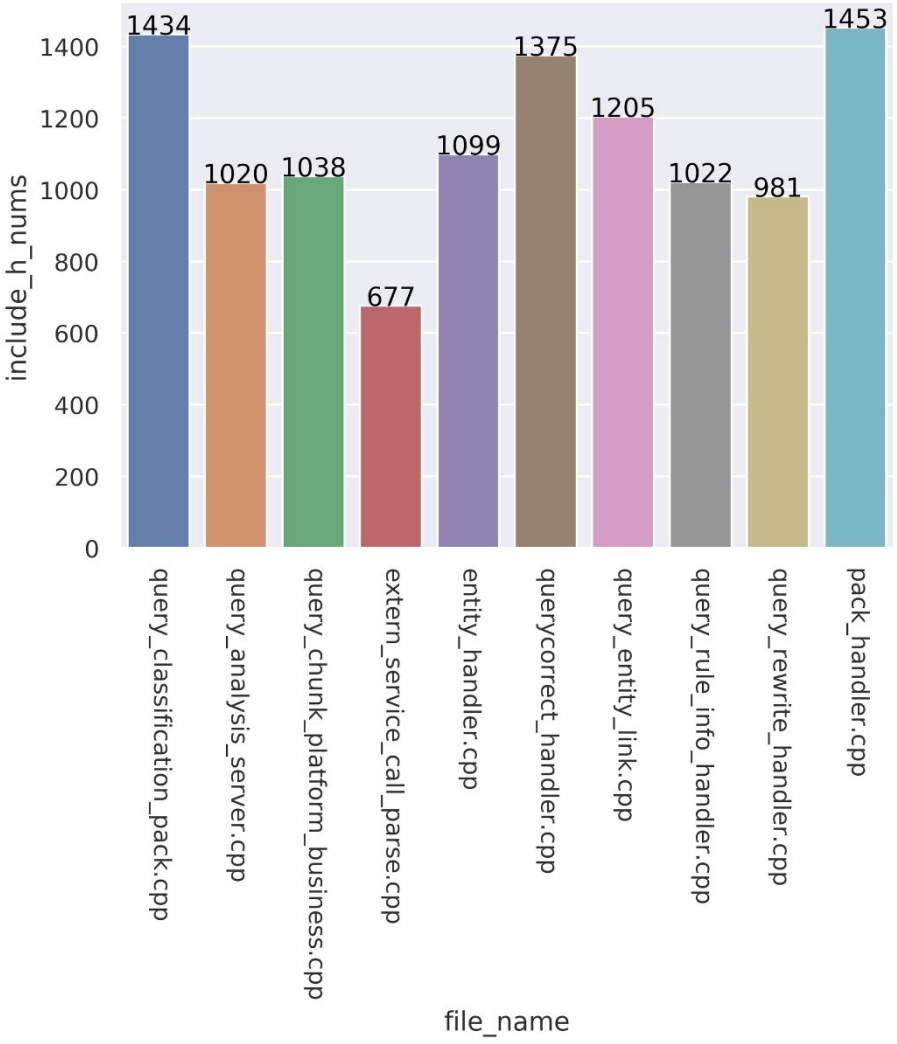


(3) Top10 编译中间文件大小统计

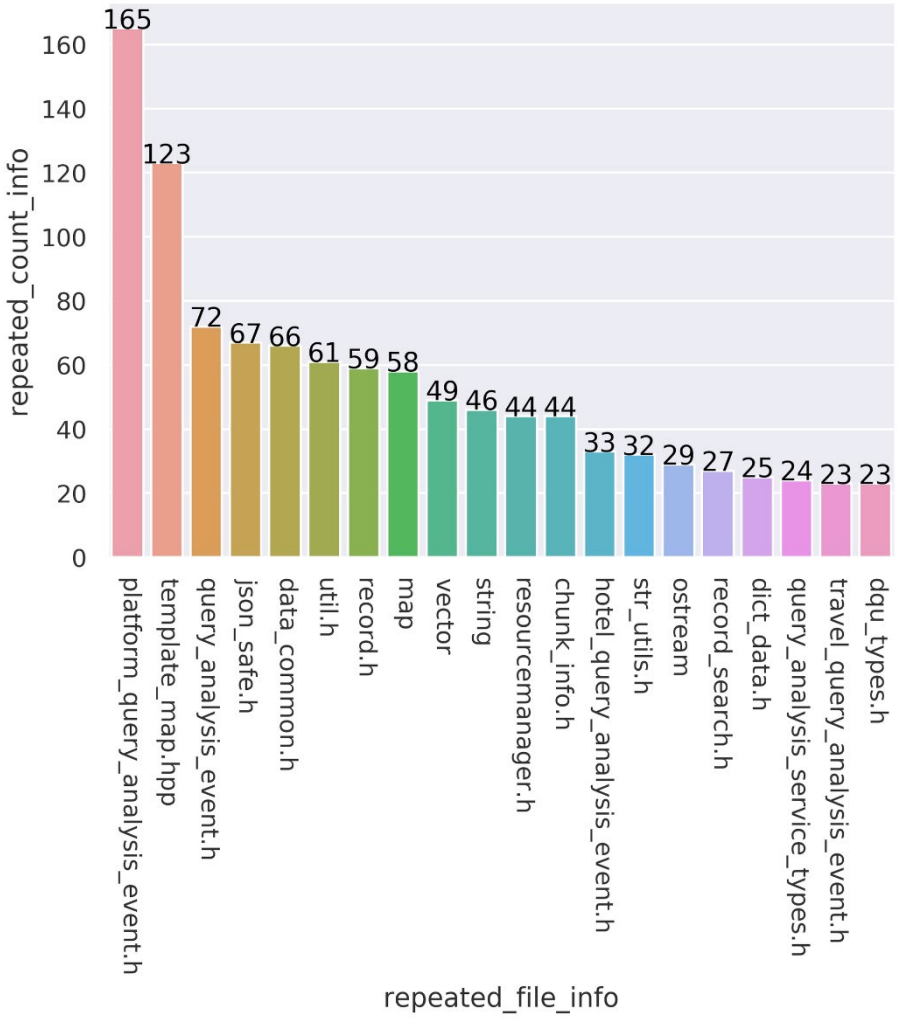
通过统计和展示编译文件大小，用来判断这块是否符合预期，这个是和编译耗时一一对应的。



（4）Top10 引入最多头文件的头文件统计



(5) Top10 头文件重复次数统计



目前，这个工具支持一键化生成编译耗时分析结果，其中几个小工具，比如依赖文件个数工具已经集成到公司的上线集成测试流程中，通过自动化工具检查代码改动对编译耗时的影响，工具的建设还在不断迭代优化中，后续会集成到公司的 MCD 平台中，可以自动分析来定位编译耗时长的问题，解决其它部门编译耗时问题。

四、优化方案与实践

通过运用上述相关工具，我们能够发现 Top10 编译耗时文件的共性，比如都依赖消息总线文件 platform_query_analysis_enent.h，这个文件又直接间接引入 2000 多个头文件，我们重点优化了这类文件，通过工具的编译展开，找出了 Boost 使用、模板类展开、Thrift 头文件展开等共性问题，并针对这些问题做专门的优化。此外，我们也使用了一些业内通用的编译优化方案，并取得了不错的效果。下面详细介绍我们采用的各种优化方案。

4.1 通用编译加速方案

业内有不少通用编译加速工具（方案），无需侵入代码就能提高编译速度，非常值得尝试。

（1）并行编译

在 Linux 平台上一般使用 GNU 的 Make 工具进行编译，在执行 make 命令时可以加上 `-j` 参数增加编译并行度，如 `make -j 4` 将开启 4 个任务。在实践中我们并不将该参数写死，而是通过 `$(nproc)` 方法动态获取编译机的 CPU 核数作为编译并发度，从而最大限度利用多核的性能优势。

（2）分布式编译

使用分布式编译技术，比如利用 Distcc 和 Dmucs 构建大规模、分布式 C++ 编译环境，Linux 平台利用网络集群进行分布式编译，需要考虑网络时延与网络稳定性。分布式编译适合规模较大的项目，比如单机编译需要数小时甚至数天。DQU 服务从代码规模以及单机编译时长来说，暂时还不需要使用分布式的方式来加速，具体细节可以参考 Distcc 官方文档说明。

（3）预编译头文件

PCH（Precompiled Header），该方法预先将常用头文件的编译结果保存起来，这样编译器在处理对应的头文件引入时可以直接使用预先编译好的结果，从而加快整个

编译流程。PCH 是业内十分常用的加速编译的方法，且大家反馈效果非常不错。在我们的项目中，由于涉及到很多 Shared Library 的编译生成，而 Shared Library 相互之间无法共享 PCH，因此没有取得预想效果。

（4）CCache

CCache（Compiler Cache 是一个编译缓存工具，其原理是将 cpp 的编译结果保存在文件缓存中，以后编译时若对应文件无变动可直接从缓存中获取编译结果。需要注意的是，Make 本身也有一定缓存功能，当目标文件已编译（且依赖无变化）时，若源文件时间戳无变化也不会再次编译；但 CCache 是按文件内容做的缓存，且同一机器的多个项目可以共享缓存，因此适用面更大。

（5）Module 编译

如果你的项目是用 C++ 20 进行开发的，那么恭喜你，Module 编译也是一个优化编译速度的方案，C++20 之前的版本会把每一个 cpp 当做一个编译单元处理，会存在引入的头文件被多次解析编译的问题。而 Module 的出现就是解决这一问题，Module 不再需要头文件（只需要一个模块文件，不需要声明和实现两个文件），它会将你的（.ixx 或者 .cppm）模块实体直接编译，并自动生成一个二进制接口文件。import 和 include 预处理不同，编译好的模块下次 import 的时候不会重复编译，可以大幅度提高编译器的效率。

（6）自动依赖分析

Google 也推出了开源的 Include-What-You-Use 工具（简称 IWYU），基于 Clang 的 C/C++ 工程冗余头文件检查工具。IWYU 依赖 Clang 编译套件，使用该工具可以扫描出文件依赖问题，同时该工具还提供脚本解决头文件依赖问题，我们尝试搭建了这套分析工具，这个工具也提供自动化头文件解决方案，但是由于我们的代码依赖比较复杂，有动态库、静态库、子仓库等，这个工具提供的优化功能不能直接使用，其它团队如果代码结构比较简单的话，可以考虑使用这个工具分析优化，会生成如下结果文件，指导哪些头文件需要删除。

```
>>> Fixing #includes in '/opt/meituan/zhoulei/query_analysis/src/common/
qa/record/brand_record.h'
@@ -1,9 +1,10 @@

 #ifndef _MTINTENTION_DATA_BRAND_RECORD_H_
 #define _MTINTENTION_DATA_BRAND_RECORD_H_
-#include "qa/data/record.h"
-#include "qa/data/template_map.hpp"
-#include "qa/data/template_vector.hpp"
-#include <boost/serialization/version.hpp>
+#include <boost/serialization/version.hpp>  // for BOOST_CLASS_
VERSION
+#include <string>                       // for string
+#include <vector>                       // for vector
+
+#include "qa/data/file_buffer.h"        // for REG_TEMPLATE_FILE_
HANDLER
```

4.2 代码优化方案与实践

(1) 前置类型声明

通过分析头文件引用统计，我们发现项目中被引用最多的是总线类型 Event，而该类型中又放置了各种业务需要的成员，示例如下：

```
#include “a.h”
#include "b.h"
class Event {
// 业务A, B, C ...
  A1 a1;
  A2 a2;
      // ...
  B1 b1;
  B2 b2;
  // ...
};
```

这导致 Event 中包含了数量庞大的头文件，在头文件展开后，文件大小达到 15M；而各种业务都会需要使用 Event，自然会严重拖累编译性能。

我们通过前置类型声明来解决这个问题，即不引入对应类型的头文件，只做前置声明，在 Event 中只使用对应类型的指针，如下所示：

```
class A2;
// ...
class Event {
// 业务A, B, C ...
  shared_ptr<A1> a1;
  shared_ptr<A2> a2;
  // ...
  shared_ptr<B1> b1;
  shared_ptr<B2> b2;
  // ...
};
```

只有在真正使用对应成员变量时，才需要引入对应头文件；这样真正做到了按需引入头文件。

(2) 外部模板

由于模板被使用时才会实例化这一特性，相同的实例可以出现在多个文件对象中。编译器要对每一处模板进行实例化，链接器还要移除重复的实例化代码。当在广泛使用模板的项目中，编译器会产生大量的冗余代码，这会极大地增加编译时间和链接时间。C++ 11 新标准中可以通过外部模板来避免。

```
// util.h
template <typename T>
void max(T) { ... }
// A.cpp
extern template void max<int>(int);
#include "util.h"
template void max<int>(int); // 显式地实例化
void test1()
{
    max(1);
}
```

在编译 A.cpp 的时候，实例化出一个 max(int) 版本的函数。

```
// B.cpp
#include "util.h"
extern template void max<int>(int); // 外部模板的声明
void test2()
{
    max(2);
}
```

在编译 B.cpp 的时候，就不再生成 max(int) 实例化代码，这样就节省了前面提到的实例化，编译以及链接的耗时了。

（3）多态替换模板使用

我们的项目重度使用词典相关操作，如加载词典、解析词典、匹配词典（各种花式匹配），这些操作都是通过 Template 模板扩展支持各种不同类型的词典。据统计，词典的类型超过 150 个，这也造成模板展开的代码量膨胀。

```
template <class R>
class Dict {
public:
  // 匹配 key 和 condition，赋值给 record
  bool match(const string &key, const string &condition, R &record);
// 对每种类型的 Record 都会展开一次
private:
  map<string, R> dict;
};
```

幸运的是，我们词典的绝大部分操作都可以抽象出几类接口，因此可以只实现针对基类的操作：

```
class Record {  // 基类
public:
  virtual bool match(const string &condition);  // 派生类需实现
};

class Dict {
public:
  shared_ptr<Record> match(const string &key, const string &condition);
// 使用方传入派生类的指针即可
private:
  map<string, shared_ptr<Record>> dict;
};
```

通过继承和多态，我们有效避免了大量的模板展开。需要注意的是，使用指针作为 Map 的 Value 会增加内存分配的压力，推荐使用 Tcmalloc 或 Jemalloc 替换默认的 Ptmalloc 优化内存分配。

（4）替换 Boost 库

Boost 是一个广泛使用的基础库，涵盖了大量常用函数，十分方便、好用，然而也存在一些不足之处。一个显著缺点是其实现采用了 hpp 的形式，即声明和实现均放在头文件中，这会造成预编译展开后十分巨大。

```
// 字符串操作是常用功能，仅仅引入该头文件展开大小就超过 4M
#include <boost/algorithm/string.hpp>
// 与此相对的，引入多个 STL 的头文件，展开后仅仅只有 1M
#include <vector>
#include <map>
// ...
```

在我们项目中主要使用的 Boost 函数不超过二十个，部分可以在 STL 中找到替代，部分我们手动做了实现，使得项目从重度依赖 Boost 转变成绝大部分达到 Boost-Free，大大降低了编译的负担。

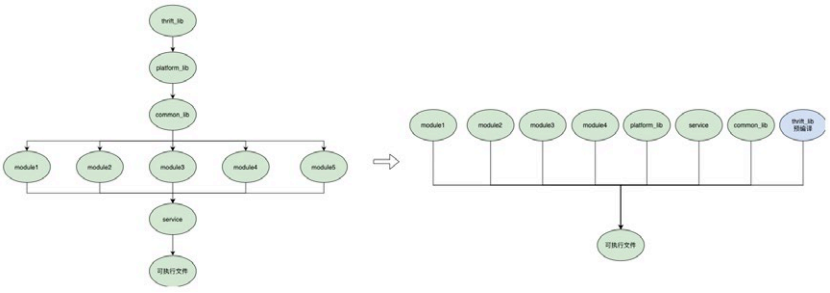
（5）预编译

代码中有一些平常改动比较少，但是对编译耗时产生一定的影响，比如 Thrift 生成的文件，模型库文件以及 Common 目录下的通用文件，我们采取提起预编译成动态库，减少后续文件的编译耗时，也解决了部分编译依赖。

（6）解决编译依赖，提高编译并行度

在我们项目中有大量模块级别的动态库文件需要编译，cmake 文件指定的编译依赖关系在一定程度上限制了编译并行度的执行。

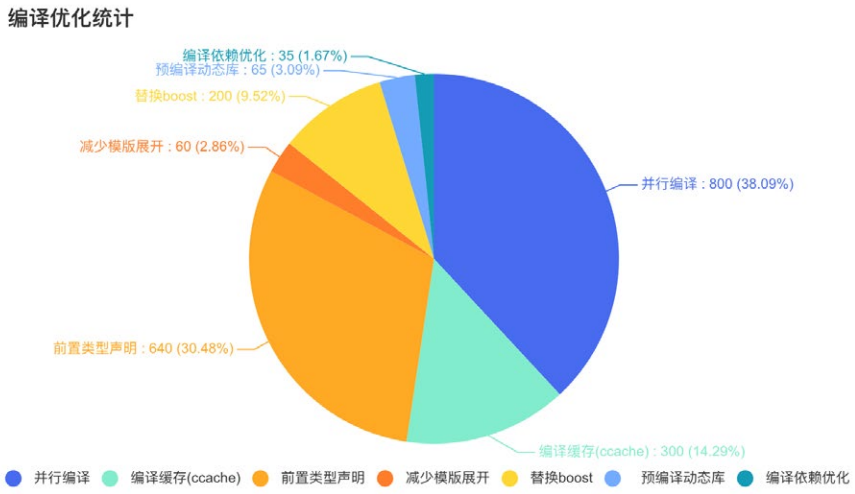
比如下面这个场景，通过合理设置库文件依赖关系，可以提高编译并行度。



4.3 优化效果

我们通过 32C、64G 内存机器做了编译耗时优化前后的效果对比，统计结果如下：

序号	优化方案	优化效果	说明
1	并行编译	800秒	根据CPU核数，提高编译并行度，16换32并行度
2	编译缓存（CCache）	300秒	首次编译无效果。CCache的优化和缓存命中情况有关，以普通更改部分cpp或者头文件测试方式
3	前置类型声明	640秒	通过前置声明，解决头文件重复依赖问题
4	减少模板展开	60秒	通过外部模板和多态替换模板来解决
5	替换Boost	200秒	减少Boost库使用
6	预编译动态库	65秒	提前编译Thrift文件打包成动态库
7	编译依赖优化	35秒	调整makefile，提高编译并行度
	总体效果	2100秒	首次编译从优化前的2200秒优化到500秒，非首次编译优化到100秒左右



4.4 守住优化成果

编译优化是一件“逆水行舟”的事情，开发人员总是倾向于不断增加新的功能、新的

库乃至新的框架，而要删除旧代码、旧库、下线旧框架总是困难重重（相信一线开发人员一定深有体会）。因此，如何守住之前取得的优化成果也是至关重要的。我们在实践中有以下几点体会：

- 代码审核是困难的（引起编译耗时增加的改动，往往无法通过审核代码直观地发现）。
- 工具、流程才值得依赖。
- 关键在于控制增量。

我们发现，cpp 文件的编译耗时，和其预编译展开文件（.ii）大小呈正相关（绝大部分情况下）；对每一个上线版本，将其所有 cpp 文件的预编译展开大小记录下来，就形成了其编译指纹（CF，Compile Fingerprint）。通过比较相邻两个版本的 CF，就能较准确的知道新版带来的编译耗时主要由哪些改动引入，并可以进一步分析耗时上涨是否合理，是否有优化空间。

我们将该种方式制作成脚本工具并引入上线流程，从而能够很清楚的了解每次代码发版带来的编译性能影响，并有效地帮助我们守住前期的优化成果。

五、总结

DQU 项目是美团搜索业务环节中重要的一环，该系统需要对接 20+RPC、数十个模型、加载超过 300 个词典，使用内存数十 G，日均响应请求超过 20 亿的大型 C++ 服务。在业务高速迭代的情况，冗长的编译时间为开发同学带来较大的困扰，一定程度上制约了开发效率。最终我们通过编译优化分析工具建设，结合采用了通用编译优化加速方案和代码层面的优化，将 DQU 的编译时间缩短了 70%，并通过引 CCache 等手段，使得本地开发的编译，能够在 100s 内完成，给开发团队节省了大量的时间。

在取得阶段性成果之后，我们总结整个问题解决的过程，并沉淀出一些分析方法、工具以及流程规范。这些工具在后续的开发迭代过程中，能够快速有效地检测新的

代码变更带来的编译时间变化，并成为了我们的上线流程检查中的一环检测标准。这一点与我们以往一次性的或者针对性的编译优化，产生了很大的区别。毕竟代码的维护是一个持久的过程，系统化的解决这一问题，不只是需要有效的方法和便捷的工具，更需要一个标准化的，规范化的上线流程来保持成果。希望本文对大家能有所帮助。

参考文献

[1]《编译原理透视 · 图解编译原理》
[2] CCache
[3] 分布式编译
[4] 头文件预编译
[5] 头文件预编译
[6] C++ Templates
[7] Include-what-you-use

作者简介

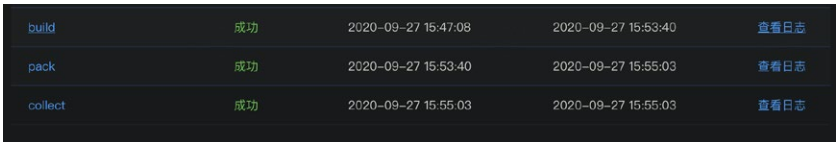
本文作者周磊、识瀚、朱超、王鑫、刘亮、昌术、李超、云森、永超等，均来自美团 AI 平台搜索与 NLP 部。

速度与压缩比如何兼得？压缩算法在构建部署中的优化

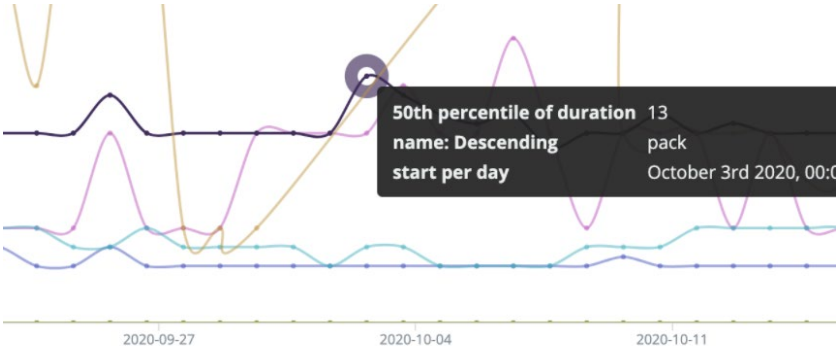
作者：宏达

背景

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



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



pack 作为几乎所有需要部署的服务必需步骤，它目前的耗时基本上仅低于编译和构建镜像，因此，为了提高整体构建的效率，我们准备对 pack 打包压缩的步骤进行一轮优化工作。

方案对比

准备场景数据

发布项的包大小分析

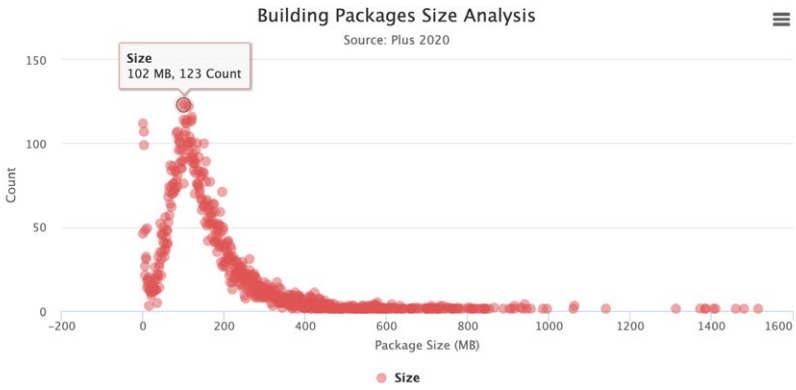
为了尽可能地模拟构建部署中的应用场景，我们将 2020 年的部分**构建包数据**进行了整理分析，其中压缩后的包大小如下图所示，钟形曲线说明了整体的包体积呈正态分布，并且有着较明显的长尾效应。压缩后体积主要在 200M 以内，压缩前的大小大致在 516.0MB 以内。

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

这样选择的主要原因如下：

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

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



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

目录大小为 1.2G，也可以比较清晰地对比出不同方案之间的差距。

gzip

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

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

文件格式

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

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

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

Brotli

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

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

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

具体测试数据见下文。

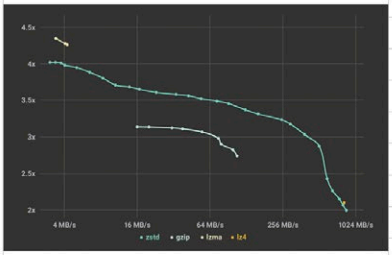
Zstd

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

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

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

官方 Benchmark 数据对比



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

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

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

LZ4

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

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

比并不是它的强项，它默认支持 1-9 的压缩参数，我们分别进行了测试。

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

Pigz

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

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

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

ISA-L Acceleration Version

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

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

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

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

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

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

备注：截图来自 crc32_ieee_by16_10.asm

使用

ISA-L 实现的压缩优化级别支持 [0,3]，3 为压缩比最大的版本，综合考虑我们采用

了最大的压缩比，虽然压缩比 2.346 略低于 gzip 不过影响不大。在 2019 年 6 月发布的 v2.27 版本里，ISA-L 加了多线程的 Feature，因此在后续的测试中，我们采用了多线程并发的参数，效果提升比较显著。

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

压缩时间

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

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

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

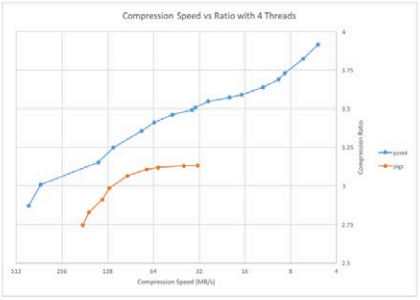
Pzstd

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

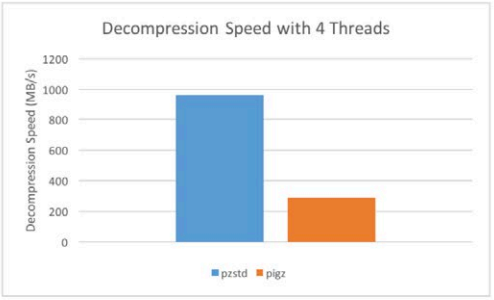
Pzstd 是 C++11 实现的并行版本的 Zstandard（Zstd 也在这之后加入了多线程的支持），类似于 Pigz 的工具。 它提供了与 Zstandard 格式兼容的压缩和解压缩功能，可以利用多个 CPU 核心。 它将输入分成相等大小的块，并将每个块独立压缩为 Zstandard 帧。 然后将这些帧连接在一起以产生最终的压缩输出。 Pzstd 同样支持文件的并行解压缩。 解压缩使用 Zstandard 压缩的文件时，PZstandard 在一个线程中执行 IO，而在另一个线程中进行解压缩。

下图是和 Pigz 的压缩和解压缩速度对比，来自官方 Github 仓库（机器配置为：

Intel Core i7 @ 3.1 GHz, 4 threads），效果比 Pigz 还要出色，具体对比数据见下文：



压缩速度



解压缩速度

Pied Piper (Middle-out compression)

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

备注：Pied Piper 是美剧《硅谷》中虚拟出来的公司和算法。

兼容性

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

选型

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

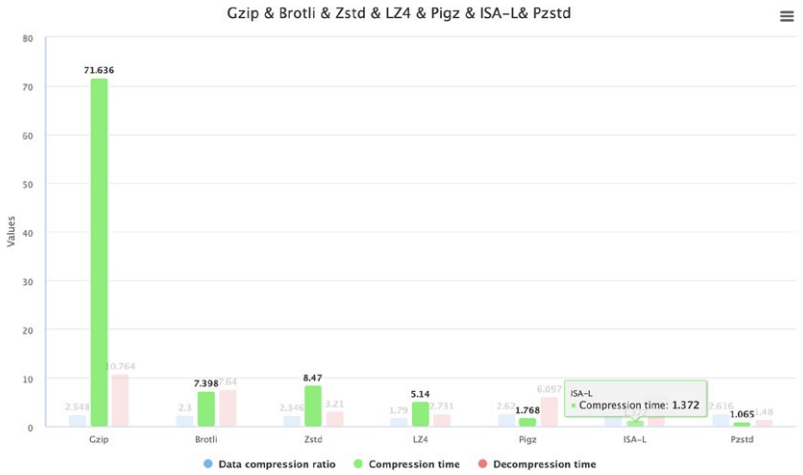
对比数据

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

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

压缩时间对比

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



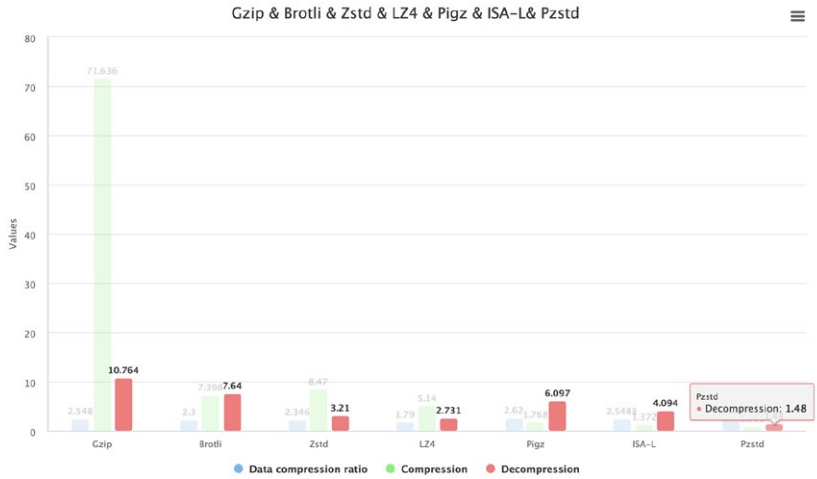
解压缩时间对比

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

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

下面的时间对比是分别使用各自的解压方案的对比：

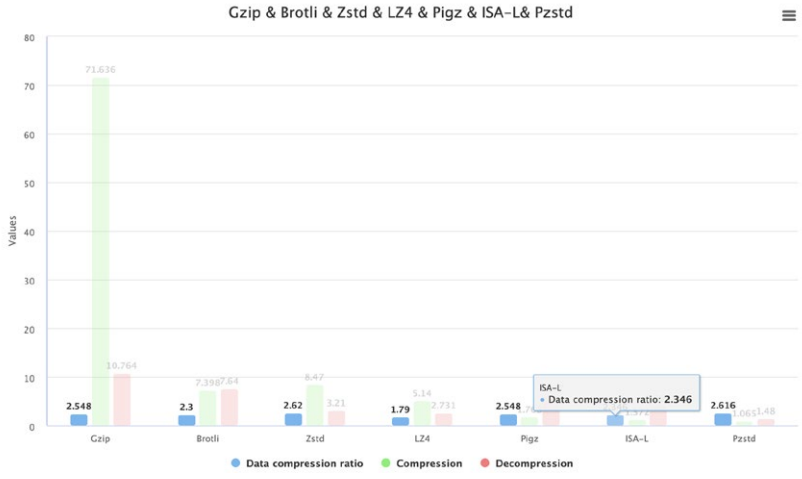
- Pzstd 解压速度最快，相比 Gzip 节省了 86.241% 的时间。
- Zstd 算法的解压缩效率其次，大约可以节省 70.169% 的解压时间。
- ISA-L 可节省 61.9658% 的时间。
- Pigz 可节省 43.357% 的解压时间。
- Brotli 解压可以节省 29.02% 的时间。



压缩比的对比

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

而 ISA-L 的实现在压缩比上有一些牺牲，不过并没有差距很大。



优劣分析总结

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

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

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

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

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

评分策略

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

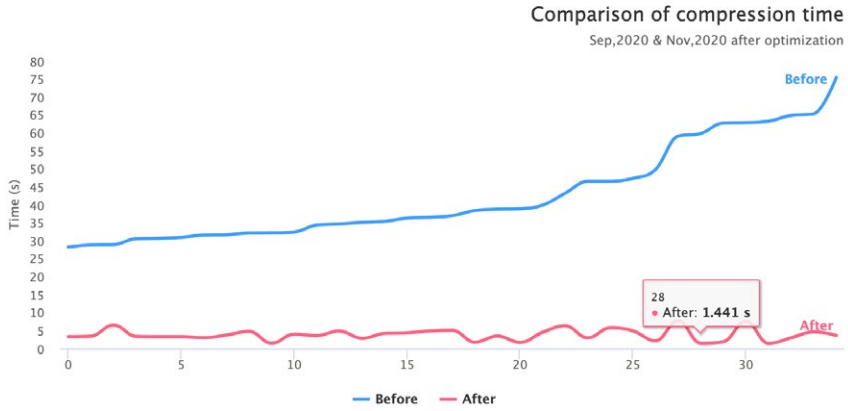
而对于部署来说，可能收益并不是十分明显，反而加重了一些维护和运维成本，所以我们暂时没有把 Pzstd 的方案放到较高的优先级。

选型的策略主要有基于如下原则：

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

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

优化效果

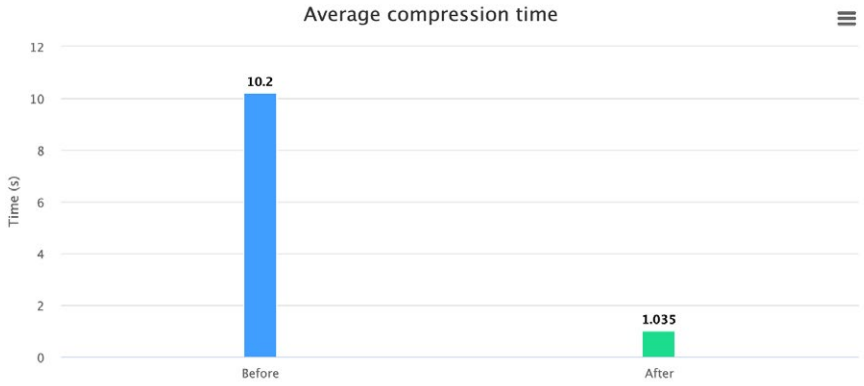


为了方便结果的展示，我们过滤出了部分打包时间较长的发布项展示出来（这些耗时很久的项目往往十分影响用户的使用体验，而且总体的占比在 10% 左右），这部分任务优化的时间从 27s 到 72s 不等，过去越是项目大的项目压缩时间越长，如今压缩时间都可以稳定在 10s 以内，而且是在我们的机器同时执行多个任务的情况下。

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

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

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



压缩平均节省了 90% 的时间。

写在后面

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

作者简介

宏达，美团基础技术部研发工程师。

团队简介

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

附录

机器环境

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

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

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

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

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

```
/proc/cpuinfo 文件中也可以查看 CPU 支持的指令集

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

参考文献

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

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

招聘信息

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

美团 OCTO 万亿级数据中心计算引擎技术解析

作者：继东　业祥　成达　张昀

美团自研的 OCTO 数据中心（简称 Watt）日均处理万亿级数据量，该系统具备较好的扩展能力及实时性，千台实例集群周运维成本低于 10 分钟。本文将详细阐述 Watt 计算引擎的演进历程及架构设计，同时详细介绍其全面提升计算能力、吞吐能力、降低运维成本所采用的各项技术方案。希望能给大家一些启发或者帮助。

一、OCTO 数据中心简介

1.1 系统介绍

1.1.1 OCTO 系统介绍

OCTO 是美团标准化的服务治理基础设施，目前几乎覆盖公司所有业务的治理与运营。OCTO 提供了服务注册发现、数据治理、负载均衡、容错、灰度发布等治理功能，致力于提升研发效率，降低运维成本，并提升应用的稳定性。OCTO 最新演进动态细节可参考《美团下一代服务治理系统 OCTO2.0 的探索与实践》一文。

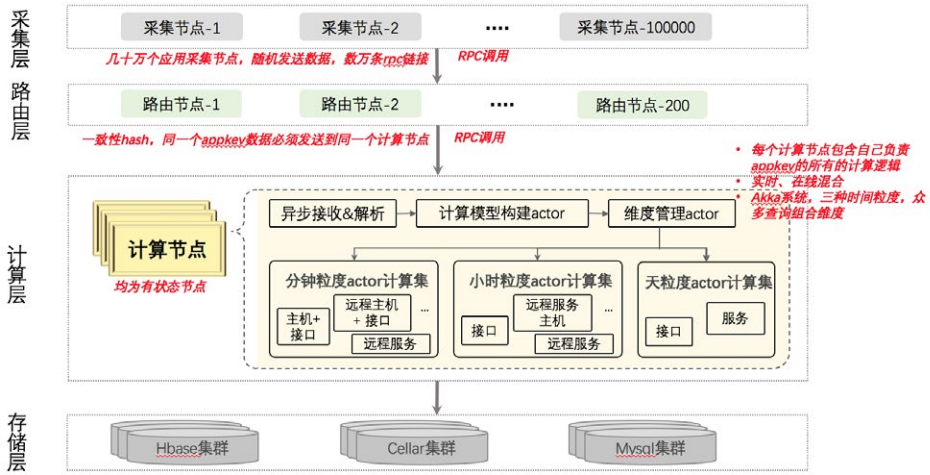
1.1.2 OCTO 数据中心业务介绍

OCTO 数据中心为业务提供了立体化的数字驱动服务治理能力，提供了多维度的精确时延 TP（Top Percent，分位数，最高支持 6 个 9）、QPS、成功率等一系列核心指标，粒度方面支持秒级、分钟级、小时级、天级，检索维度支持多种复杂查询（如指定调用端 IP + 服务端各接口的指标，指定主机 + 接口的指标等）。这些功能有效地帮助开发人员在复杂的分布式调用关系拓扑内出现异常时，能快速定位到问题，也有助于研发人员全方位掌控系统的稳定性状况。

目前 Watt 承载日均超万亿条数据的 10 余个维度精确准实时统计。而伴随着数据量的迅猛增长，其整个系统架构也经历了全面的技术演进。

1.1.3 OCTO 原架构介绍

OCTO 计算引擎在重构之前，也面临诸多的问题，其原始架构设计如下：



1. **采集层**：每个业务应用实例部署一个采集代理，代理通过将采集数据用批量 RPC 的方式发送给路由节点。
2. **路由层**：每个路由节点随机收到各服务的数据后，将同一服务的所有数据，用类似 IP 直连的方式通过 RPC 发送到计算层的同一个计算节点。同服务数据汇总到同计算节点才能进行特定服务各个维度的聚合计算。
3. **计算层**：每个计算节点采用 Akka 模型，节点同时负责分钟、小时、天粒度的数据计算集。每个计算集里面又有 10 个子计算 actor，每个子计算 actor 对应的是一个维度。采用“先计算指标，再存储数据”的准实时模式。
4. **存储层**：准实时数据使用 HBase 存储，元数据及较大数据采用 KV 存储（美团内部系统 Cellar）及 MySQL 存储。

1.2 问题、目标与挑战

1.2.1 原架构面临的问题

1. **计算节点有状态，异常无法自动化迁移**。计算层部署的每个节点平均负责 200+ 服务的统计。一个节点 OOM 或宕机时，其管理的 200 个服务的数据

会丢失或波动，报警等依托数据的治理功能也会异常。此外计算节点 OOM 时也不宜自动迁移受影响的服务，需人工介入处理（异常的原因可能是计算节点无法承载涌入的数据量，简单的迁移易引发“雪崩”），每周投入的运维时间近 20 小时。

2. **系统不支持水平扩展**。计算节点的压力由其管理的服务调用量、服务内维度复杂度等因素决定。大体量的服务需单独分配高配机器，业务数据膨胀计算节点能力不匹配时，只能替换更高性能的机器。
3. **系统整体稳定性不高**。数据传输采用 RPC，单计算节点响应慢时，易拖垮所有路由层的节点并引发系统“雪崩”。
4. **热点及数据倾斜明显，策略管理复杂**。按服务划分热点明显，存在一个服务数据量比整个计算节点 200 个服务总数多的情况，部分机器的 CPU 利用率不到 10%，部分利用率在 90%+。改变路由策略易引发新的热点机器，大体量服务数据增长时需要纵向扩容解决。旧架构时人工维护 160 余个大服务到计算节点的映射关系供路由层使用。

旧架构日承载数据量约 3000 亿，受上述缺陷影响，系统会频繁出现告警丢失、误告警、数据不准、数据延迟几小时、服务发布后 10 分钟后才能看到流量等多种问题。此外，数据体量大的服务也不支持部分二级维度的数据统计。

1.2.2 新架构设计的目标

基于上述问题总结与分析，我们新架构整体的目标如下：

1. 高吞吐、高度扩展能力。具备 20 倍 + 的水平扩展能力，支持日 10 万亿数据的处理能力。
2. 数据高度精确。解决节点宕机后自愈、解决数据丢失问题。
3. 高可靠、高可用。无计算单点，所有计算节点无状态；1/3 计算节点宕机无影响；具备削峰能力。
4. 延时低。秒级指标延迟 TP99<10s；分钟指标延迟 TP99<2min。

1.2.3 新架构面临的挑战

在日计算量万亿级别的体量下，实现上述挑战如下：

1. 数据倾斜明显的海量数据，数据指标的维度多、指标多、时间窗口多，服务间体量差异达百万倍。
2. TP 分位数长尾数据是衡量系统稳定性最核心的指标，所有数据要求非采样拟合，实现多维度下精确的分布式 TP 数据。
3. 架构具备高稳定性，1/3 节点宕机不需人工介入。
4. 每年数据膨胀至 2.4 倍 +，计算能力及吞吐能力必须支持水平扩展。
5. TP 数据是衡量服务最核心的指标之一，但在万亿规模下，精确的准实时多维度分布式 TP 数据是一个难题，原因详细解释下：常规的拆分计算后聚合是无法计算精确 TP 数据的，如将一个服务按 IP（一般按 IP 划分数据比较均匀）划分到 3 个子计算节点计算后汇总，会面临如下问题：
   - 假设计算得出 IP1 的 TP99 是 100ms、QPS 为 50；IP2 的 TP99 是 10ms、QPS 为 50；IP3 的 TP99 是 1ms，QPS 为 50。那么该服务整体 TP99 是（100ms x 50 + 10ms x 50 + 1ms x 50）/ (50 + 50 + 50) = 37ms 吗？ 并非如此，该服务的真实 TP99 可能是 100ms，在没有全量样本情况下无法保证准确的 TP 值。
   - 假设不需要服务全局精确的时延 TP 数据，也不可以忽略该问题。按上述方式拆分并合并后，服务按接口维度计算的 TP 数据也失去了准确性。进一步说，只要不包含 IP 查询条件的所有 TP 数据都失真了。分位数这类必须建立在全局样本之上才能有正确计算的统计值。

二、计算引擎技术设计解析

2.1 方案选型

大数据计算应用往往基于实时或离线计算技术体系建设，但 Flink、Spark、OLAP 等技术栈在日超万亿级别量级下，支持复杂维度的准实时精确 TP 计算，对资源的消耗非常较大，总结如下：

技术体系	特定场景不适宜原因分析
Hadoop 离线任务	（1）时延不满足，未纳入考虑
Storm 实时计算	（1）内存密集型，时间窗口大 （2）批处理实时性受较大影响
Flink 实时流计算	（1）大窗口排序（2）大量shuffle和sort，降低DAG优化（3）同一条数据需做广播管理
Spark 实时流计算	（1）大窗口排序，微批架构吞吐有限（2）热点明显散列管理较复杂（3）集群资源较难控制
OLAP 存储引擎	（1）高实时性要求下精确TP计算较难支持

2.2 系统设计思路

1. 解决稳定性问题，思路是（1）将计算节点无状态化（2）基于心跳机制自动剔除异常节点并由新节点承载任务（3）消息队列削峰。
2. 解决海量数据计算问题，思路是（1）在线 & 离线计算隔离，两者的公共子计算前置只计算一次（2）高并发高吞吐能力设计（3）理论无上限的水平扩展能力设计。
3. 解决热点问题，思路是（1）优化计算量分配算法，如均衡 Hash（2）理论无上限的水平扩展能力设计。
4. 解决水平扩展问题，思路（1）是将单节点无法承载的计算模式改为局部分布式子计算并汇总，但这种方式可能会对数据准确性造成较大影响，数据统计领域精确的分布式分位数才是最难问题，另外多维条件组织对分布式改造也相当不利。（备注：其中在 1.2.3 第五条有详细的解释）
5. 解决海量数据分布式多维精确 TP 计算，采用局部分布式计算，然后基于拓扑树组织数据计算流，在前置的计算结果精度不丢失的前提下，多阶段逐级降维得到所有的计算结果。

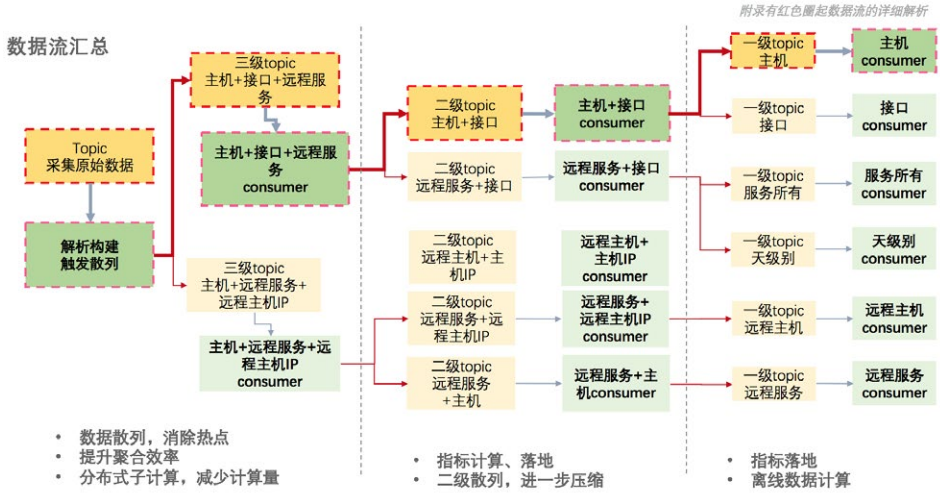
2.3 技术方案详细解析

2.3.1 数据流解析

系统根据待统计的维度构建了一棵递推拓扑树，如下图所示。其中黄色的部分代表消

息队列（每个矩形代表一个 topic），绿色部分代表一个计算子集群（消费前置 topic 多个 partition，统计自己负责维度的数据指标并存储，同时聚合压缩向后继续发）。除“原始采集数据 topic 外”，其他 topic 和 consumer 所在维度对应数据的检索条件（如标红二级 topic ：主机 + 接口，代表同时指定主机和接口的指标查询数据），红色箭头代表数据流向。

拓扑树形结构的各个子集群所对应的维度标识集合，必为其父计算集群对应维度标识集合的真子集（如下图最上面链路，“主机 + 接口 + 远程服务”3 个维度一定包含“主机 + 接口”两个维度。“主机 + 接口”两个维度包含“主机”维度）。集群间数据传输，采用批量聚合压缩后在消息队列媒介上通信完成，在计算过程中实现降维。



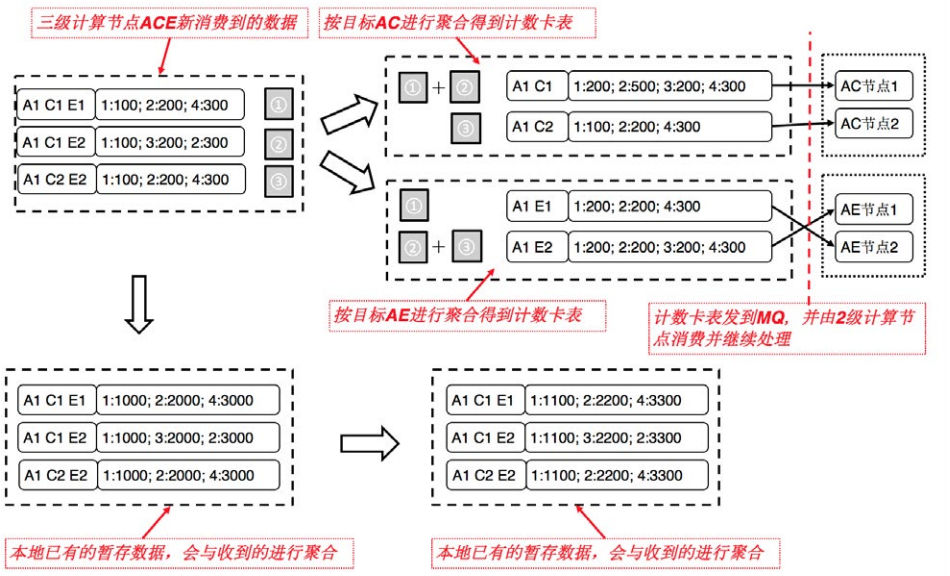
2.3.2 计算模式解析

下面详细介绍数据拓扑树中分布式子集群的计算模式：

1. 首先，维度值相同的所有数据会在本层级集群内落到同一计算节点。每个计算子集群中的各计算节点，从消息队列消费得到数据并按自身维度进行聚合（前置集群已经按当前集群维度指定分发，所以聚合率很高），得到若干计数卡表（计数卡表即指定维度的时延、错误数等指标与具体计数的映射 Map）。

2. 其次，将聚合后的计数卡表与现有的相同维度合并计算，并在时间窗口存储指标。
3. 若计算集群有后续的子计算集群，则基于后继集群的目标维度，根据目标维度属性做散列计算，并将相同散列码的计数卡表聚合压缩后发到后继 partition。
4. 离线的天级计算复用了三级维度、二级维度的多项结果，各环节前面计算的结果为后面多个计算集群复用，任何一个服务的数据都是在分布式计算。此外，整个计算过程中维护着技术卡表的映射关系，对于 TP 数据来说就是精确计算的，不会失真。

整个计算过程中，前置计算结果会被多个直接及间接后续子计算复用（如三级聚合计算对二级和一级都是有效的），在很大程度上减少了计算量。



2.3.3 关键技术总结

1. 高吞吐 & 扩展性关键设计

- 去计算热点：组织多级散列数据流，逐级降维。
- 降计算量：前置子计算结果复用，分布式多路归并。

- 降网络 IO，磁盘 IO：优化 PB 序列化算法，自管理 MQ 批量。
- 去存储热点：消除 HBase Rowkey 热点。
- 无锁处理：自研线程分桶的流批处理模型，全局无锁。
- 全环节水平扩展：计算、传输、存储均水平扩展。

2. 系统高可靠、低运维、数据准确性高于 5 个 9 关键设计

- 无状态化 + 快速自愈：节点无状态化，宕机秒级感知自愈。
- 异常实时感知：异常节点通过心跳摘除，异常数据迁移回溯。
- 故障隔离容灾：各维度独立隔离故障范围；多机房容灾。
- 逐级降维过程中数据不失真，精确的 TP 计算模式。

3. 提升实时性关键设计

- 流式拓扑模型，分布式子计算结果复用，计算量逐级递减。
- 无锁处理：自研线程分桶的流批处理模型，全局无锁。
- 异常实时监测自愈：计算节点异常时迅速 Rebalance，及时自愈。
- 秒级计算流独立，内存存储。

三、优化效果

1. 目前日均处理数据量超万亿，系统可支撑日 10 万亿 + 量级并具备良好的扩展能力；秒峰值可处理 5 亿 + 数据；单服务日吞吐上限提升 1000 倍 +，单服务可以支撑 5000 亿数据计算。
2. 最大时延从 4 小时 + 降低到 2min−，秒级指标时延 TP99 达到 6s；平均时延从 4.7 分 + 降低到 1.5 分 −，秒级指标平均时延 6s。
3. 上线后集群未发生雪崩，同时宕机 1/3 实例 2 秒内自动化自愈。
4. 支持多维度的准实时精确 TP 计算，最高支持到 TP 6 个 9；支持所有服务所有维度统计。
5. 千余节点集群运维投入从周 20 小时 + 降低至 10 分 −。

四、总结

本文提供了一种日均超万亿规模下多维度精确 TP 计算的准实时数据计算引擎方案，适用于在超大规模数字化治理体系建设中，解决扩展性、实时性、精确性、稳定性、运维成本等问题。美团基础研发平台欢迎业界同行一起交流、探讨。

五、作者简介

继东，业祥，成达，张昀，均来自基础架构服务治理团队，研发工程师。

招聘信息

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

Intel PAUSE 指令变化影响到 MySQL 的性能，该如何解决？

作者：春林

MySQL 得益于其开源属性、成熟的商业运作、良好的社区运营以及功能的不断迭代与完善，已经成为互联网关系型数据库的标配。可以说，X86 服务器、Linux 作为基础设施，跟 MySQL 一起构建了互联网数据存储服务的基石，三者相辅相成。本文将分享一个工作中的实践案例：因 Intel PAUSE 指令周期的迭代，引发了 MySQL 的性能瓶颈，美团 MySQL DBA 团队如何基于这三者来一步步进行分析、定位和优化。希望这些思路能对大家有所启发。

1. 背景

在 2017 年，Intel 发布了新一代的服务器平台 Purley，并将 Intel Xeon Scalable Processor（至强可扩展处理器）重新划分为：Platinum（铂金）、Gold（金）、Silver（银）、Broze（铜）等四个等级。产品定位和框架也变得更加清晰。

因美团线上海量数据交易和存储等后端服务依赖大量高性能服务器的支撑。随着线上部分 Grantly 平台 E 系列服务器生命周期的临近，以及产品本身的发展和迭代。从 2019 年开始，RDS（关系型数据库服务）后端存储（MySQL）开始大量上线 Purley 平台的 Skylake CPU 服务器，其中包含 Silver 4110 等。

Silver 4110 相比上一代 E5-2620 V4，支持更高的内存频率、更多的内存通道、更大的 L2 Cache、更快的总线传输速率等。Intel 官方数据显示 Silver 4110 的性能比上一代 E5-2620 V4 提升了 10%。

然而，随着线上 Skylake 服务器数量的增加，以及越来越多的业务接入。美团 MySQL DBA 团队发现部分 MySQL 实例性能与预期并不相符，有时甚至出现较大程度的下降。经过持续的性能问题分析，我们定位到 Skylake 服务器存在性能瓶颈：

- CPU 负载相对较高。
- TPS 等吞吐量下降。

接下来，我们将从 Intel CPU、ut_delay 函数、PAUSE 指令三方面入手，进行剖析定位，并探索相关优化方案。

2. 性能问题分析

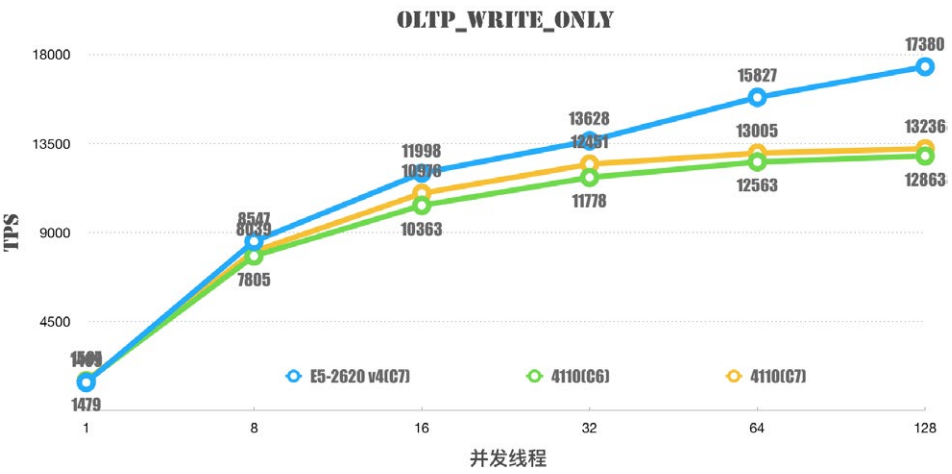
2.1 Grantly 与 Purley CPU 性能差异

首先，基于上述两代平台的 CPU（Grantly 和 Purley），通过基准测试，横向对比在不同 OS 下的性能表现。

通过基准测试数据，总结如下：

1. 在 oltp_write_only（只写）的场景下 Purley 4110 的性能下降较为明显。 2. 同为 Purley 4110，CentOS 7 比 CentOS 6 oltp_write_only（只写）性能有提升。

我们通过二维折线图，来展示性能之间的差异：



在上图中，同为 Purley 4110，CentOS 7 比 CentOS 6 性能有提升。具体提升原因，因不涉及本文重点内容，所以不在这里详细展开了。

> New MCS-based Locking Mechanism
>
> Red Hat Enterprise Linux 7.1 introduces a new locking mechanism, MCS locks. This new locking mechanism significantly reduces spinlock overhead in large systems, which makes spinlocks generally more efficient in Red Hat Enterprise Linux 7.1.

红帽官网 Release Notes 显示，从内核 3.10.0-229 开始，引入了新的加锁机制，MCS 锁。可以降低 spinlock 的开销，从而更高效地运行。普通 spinlock 在多 CPU Core 下，同时只能有一个 CPU 获取变量，并自旋，而缓存一致性协议为了保证数据的正确，会对所有 CPU Cache Line 状态、数据，同步、失效等操作，导致性能下降。而 MSC 锁实现每个 CPU 都有自己的“spinlock”本地变量，只在本地自旋。避免 Cache Line 同步等，从而提升了相关性能。不过，社区对于 spinlock 的优化争议还是比较大的，后续又有大牛基于 MSC 实现了 qspinlock，并在 4.x 的版本上 patch 了。具体实现可以参看：MCS locks and qspinlocks。

在大致了解 CentOS 7 性能的迭代后，接下来我们深入分析一下 Skylake CPU 4110 导致性能下降的缘由。

3.CPU 性能跟踪

3.1 定位热点函数

具体定位 4110 性能瓶颈，分如下几步：

1. 首先，通过 perf top 来跟踪一下 Linux CPU 性能开销。
2. 然后，通过 perf record 记录函数 CPU 周期的消耗占比。
3. 最后，通过火焰图来验证定位热点函数。

```
  Children      Self  Shared Object       Symbol
+   77.13%     0.00%  mysqld              [.] handle_connection
+   77.12%     0.15%  mysqld              [.] do_command
+   74.41%     0.35%  mysqld              [.] dispatch_command
+   71.34%     0.02%  mysqld              [.] mysqld_stmt_execute
+   71.22%     0.15%  mysqld              [.] Prepared_statement::execute_loop
+   70.08%     0.01%  mysqld              [.] Prepared_statement::execute
+   68.54%     0.22%  mysqld              [.] mysql_execute_command
+   25.24%     0.05%  mysqld              [.] trans_commit_stmt
+   25.17%     0.11%  mysqld              [.] ha_commit_trans
+   21.73%     0.05%  mysqld              [.] Sql_cmd_update::execute
+   21.67%     0.02%  mysqld              [.] Sql_cmd_update::try_single_table_update
+   20.38%     0.14%  mysqld              [.] mysql_update
+   19.46%     0.08%  mysqld              [.] MYSQL_BIN_LOG::commit
+   18.80%     0.02%  mysqld              [.] MYSQL_BIN_LOG::ordered_commit
-   18.61%    17.89%  mysqld              [.] ut_delay
   - 2.77% start_thread
      - 1.89% pfs_spawn_thread
           handle_connection
           do_command
           dispatch_command
+   13.50%     0.03%  [kernel]            [k] system_call_fastpath
+   11.22%     0.00%  libpthread-2.17.so  [.] start_thread
+   11.15%     0.00%  mysqld              [.] row_update_for_mysql
+   10.97%     0.00%  mysqld              [.] mtr_t::commit
+   10.83%     0.02%  mysqld              [.] row_upd
+   10.80%     0.57%  mysqld              [.] mtr_t::Command::execute
+   10.36%     0.00%  mysqld              [.] row_upd_step
+    9.65%     0.03%  mysqld              [.] handler::ha_update_row
+    9.21%     0.06%  mysqld              [.] Sql_cmd_delete::mysql_delete
+    9.06%     0.05%  mysqld              [.] Sql_cmd_insert::execute
+    9.06%     0.00%  mysqld              [.] Sql_cmd_delete::execute
+    8.99%     0.68%  mysqld              [.] mtr_t::Command::prepare_write
+    8.70%     0.00%  mysqld              [.] pfs_spawn_thread
+    8.67%     0.11%  mysqld              [.] Sql_cmd_insert::mysql_insert
+    8.21%     0.14%  mysqld              [.] ha_innobase::update_row
+    8.17%     0.50%  mysqld              [.] btr_cur_search_to_nth_level
+    7.93%     0.02%  mysqld              [.] MYSQL_BIN_LOG::change_stage
+    7.31%     0.03%  mysqld              [.] QUICK_RANGE_SELECT::get_next
+    7.21%     0.04%  mysqld              [.] handler::multi_range_read_next
+    7.14%     0.10%  mysqld              [.] ha_innobase::index_read
     6.93%     0.02%  mysqld              [.] handler::read_range_first
+    6.85%     0.01%  mysqld              [.] handler::ha_write_row
+    6.85%     0.18%  mysqld              [.] row_search_mvcc
     6.80%     0.00%  mysqld              [.] handler::ha_index_read_map
```

可以看到，其中占 CPU 消耗占比较大为：ut_delay 函数。

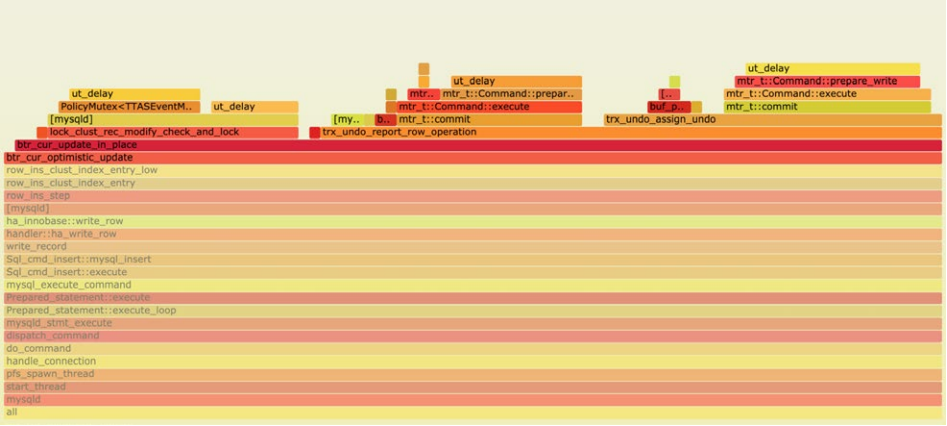
我们继续深挖一下函数链调用关系：

```
# Children      Self  Command  Shared Object
Symbol

# ........  ........  .......  ..................  ..................
.....................................................................
.....................................................................
.....................
#
    93.54%     0.00%  mysqld   libpthread-2.17.so   [.] start_thread
```

```
            |
            ---start_thread
               |
               |--77.07%--pfs_spawn_thread
               |          |
               |           --77.05%--handle_connection
               |                     |
               |                      --76.97%--do_command
               |                                |
               |                                |--74.30%--dispatch_
command
               |                                |          |
               |                                |
|--71.16%--mysqld_stmt_execute
               |                                |          |          |
               |                                |          |
--70.74%--Prepared_statement::execute_loop
               |                                |          |
|
               |                                |          |
|--69.53%--Prepared_statement::execute
               |                                |          |
|          |
               |                                |          |
|          |--67.90%--mysql_execute_command
               |                                |          |
|          |          |
               |                                |          |
|          |          |--23.43%--trans_commit_stmt
               |                                |          |
|          |          |          |
               |                                |          |
|          |          |           --23.30%--ha_commit_trans
               |                                |          |
|          |          |                     |
               |                                |          |
|          |          |                     |--18.86%--MYSQL_BIN_
LOG::commit
               |                                |          |
|          |          |                     |          |
               |                                |          |
|          |          |                     |           --18.18%--MYSQL_
BIN_LOG::ordered_commit
               |                                |          |
|          |          |                     |                     |
               |                                |
|                     |          |          |                     |
|--8.02%--MYSQL_BIN_LOG::change_stage
               |                                |          |
```

```
|          |          |                      |                    |
|
|               |                                  |          |
|          |          |                      |                    |
|--2.35%--__lll_unlock_wake
|               |                                  |          |
|          |          |                      |                    |
|          |
|               |                                  |          |
|          |          |                      |                    |
|           --2.24%--system_call_fastpath
|               |                                  |          |
|          |          |                      |                    |
|                     |
|               |                                  |          |
|          |          |                      |                    |
|                      --2.24%--sys_futex
|               |                                  |          |
|          |          |                      |                    |
|                                  |
|               |                                  |          |
|          |          |                      |                    |
|                                  --2.23%--do_futex
|               |                                  |          |
|          |          |                      |                    |
|                                             |
|               |                                  |          |
|          |          |                      |                    |
|                                              --2.14%--futex_wake
|               |                                  |          |
|          |          |                      |                    |
|                                                        |
|               |                                  |          |
|          |          |                      |                    |
|                                                         --1.38%--wake_
up_q
|               |                                  |          |
|          |          |                      |                    |
|                                                                    |
|               |                                  |
|                     |          |
|                     |                      |          |
--1.33%--try_to_wake_up
                ...
```

将上述调用通过火焰图进行直观展示：



现在基本可以确定，所有的函数调用，最后大部分的消耗都在 ut_delay 上。

3.2 ut_delay 和 PAUSE 之间的关联与性能影响

3.2.1 MySQL ut_delay 实现

接下来，我们继续看一下 MySQL 源码中 ut_delay 函数的功能：

```
/**********************************************************//**
Runs an idle loop on CPU. The argument gives the desired delay
in microseconds on 100 MHz Pentium + Visual C++.
@return dummy value */
ulint
ut_delay(
/*=====*/
  ulint delay)  /*!< in: delay in microseconds on 100 MHz Pentium */
{
  ulint i, j;

  UT_LOW_PRIORITY_CPU();

  j = 0;

  for (i = 0; i < delay * 50; i++) {
    j += i;
    UT_RELAX_CPU();
  }

  UT_RESUME_PRIORITY_CPU();

  return(j);
```

```
}
...

#   define UT_RELAX_CPU() asm ("pause" )
#   define UT_RELAX_CPU() __asm__ __volatile__ ("pause")
```

可以了解到，MySQL 自旋会调用 PAUSE 指令，从而提升 spin-wait loop 的性能。

3.2.2 PAUSE 指令周期的演变

我们可以看下 Intel 官网，也描述了在新平台架构 PAUSE 的改动：

> Pause Latency in Skylake Microarchitecture
>
> The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.
>
> ...
>
> The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake microarchitecture it has been extended to as many as 140 cycles.
>
> The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have

negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.
As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.
...

- 上一代架构中（Grantly 平台 E 系列）PAUSE 的周期时长为 10 cycles，新一代的 Skylake 架构中则为 140 cycles。
- 如果程序中使用固定次数的 PAUSE 循环来实现一段时间的延迟，以此阻塞程序执行，可能引发非预期的延迟。
- 由于 PAUSE 周期增加，对于 PAUSE 敏感的应用会有一定的性能损失。

衡量程序执行性能的简化公式：

ExecutionTime(T)=InstructionCount*TimePerCycle*CPI

即：程序执行时间 = 程序总指令数 x 每 CPU 时钟周期时间 x 每指令执行所需平均时钟周期数。

MySQL 内部自旋，就是通过固定次数的 PAUSE 循环实现。可知，PAUSE 指令周期的增加，那么执行自旋的时间也会增加，即程序执行的时间也会相对增加，对系统整体的吞吐量就会有影响。

显然，Intel 文档已说明不同平台、不同架构 CPU PAUSE 定义的周期是不一样的。

下面，我们通过一个测试用例来大致验证、对比一下新老架构 CPU 执行 PAUSE 的 cycles：

```
 #include <stdio.h>
#define TIMES 5

static inline unsigned long long rdtsc(void)
{
    unsigned long low, high;
```

```
    asm volatile("rdtsc" : "=a" (low), "=d" (high) );
    return ((low) | (high) << 32);
}

void pause_test()
{
    int i = 0;
    for (i = 0; i < TIMES; i++) {
        asm(
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"\
                "pause\n"
                ::
                :);
    }
}

unsigned long pause_cycle()
{
    unsigned long start, finish, elapsed;
    start = rdtsc();
    pause_test();
    finish = rdtsc();
    elapsed = finish - start;
    printf("Pause 的 cycles 约为 :%ld\n", elapsed / 100);
    return 0;
}

int main()
{
    pause_cycle();
    return 0;
}
```

其运行结果统计如下：

CPU	Cycles	备注
4110	122	• 4110 、5118 跟E5-2620 V4 相差超过10倍。 • 5218 跟E5-2620 V4 相差约4倍。
5118	119	
5218	35	
4210	32	
E5-2620 V4	8	

- 4110 和 5118 PAUSE 周期较大，均为 100 多，它们属于 Purley 第一代架构：Skylake。
- 4210 和 5218 PAUSE 相比前一代有提升，是因为它们同属 Purley 第二代架构：Cascadelake，该代 CPU PAUSE 指令有优化。

3.2.3 Intel 提升 PAUSE 猜想

Intel 提高 PAUSE 指令周期的原因，推测可能是减少自旋锁冲突的概率，以及降低功耗；但反而导致 PAUSE 执行时间变长，降低了整体的吞吐量。

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor read to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions.

3.3 PAUSE 导致写瓶颈分析

接下来，我们深入分析一下 PAUSE 指令导致 MySQL 写瓶颈的原因。

首先，通过 MySQL 内部统计信息，查看一下 InnoDB 信号量监控数据：

```
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 153720
--Thread 139868617205504 has waited at row0row.cc line 1075 for 0.00
seconds the semaphore:
X-lock on RW-latch at 0x7f4298084250 created in file buf0buf.cc line 1425
a writer (thread id 139869284108032) has reserved it in mode  SX
number of readers 0, waiters flag 1, lock_word: 10000000
Last time read locked in file not yet reserved line 0
Last time write locked in file /mnt/workspace/percona-server-5.7-redhat-
binary-rocks-new/label_exp/min-centos-7-x64/test/rpmbuild/BUILD/percona-
server-5.7.26-29/percona-server-5.7.26-29/storage/innobase/buf/buf0flu.cc
line 1216
OS WAIT ARRAY INFO: signal count 441329
RW-shared spins 0, rounds 1498677, OS waits 111991
RW-excl spins 0, rounds 717200, OS waits 9012
RW-sx spins 47596, rounds 366136, OS waits 4100
Spin rounds per wait: 1498677.00 RW-shared, 717200.00 RW-excl, 7.69 RW-sx
```

可见写操作并阻塞在：storage/innobase/buf/buf0flu.cc 第 1216 行调用上。

跟踪一下发生等待的源码：buf0flu.cc line 1216：

```
    if (flush_type == BUF_FLUSH_LIST
        && is_uncompressed
        && !rw_lock_sx_lock_nowait(rw_lock, BUF_IO_WRITE)) {    // 加锁前，
判断锁冲突

        if (!fsp_is_system_temporary(bpage->id.space())) {
        /* avoiding deadlock possibility involves
         doublewrite buffer, should flush it, because
         it might hold the another block->lock. */
        buf_dblwr_flush_buffered_writes(
          buf_parallel_dblwr_partition(bpage,
                flush_type));
      } else {
        buf_dblwr_sync_datafiles();
      }
      rw_lock_sx_lock_gen(rw_lock, BUF_IO_WRITE);        //  加 sx 锁
    }
...
 #define rw_lock_sx_lock_nowait(M, P)        \
  rw_lock_sx_lock_low((M), (P), __FILE__, __LINE__)
...

rw_lock_sx_lock_func(                                    // 加 sx 锁函数
/*=================*/
```

```
  rw_lock_t*  lock, /*!< in: pointer to rw-lock */
  ulint   pass, /*!< in: pass value; != 0, if the lock will
        be passed to another thread to unlock */
  const char* file_name,/*!< in: file name where lock requested */
  ulint   line) /*!< in: line where requested */

{
  ulint   i = 0;
  sync_array_t* sync_arr;
  ulint   spin_count = 0;
  uint64_t  count_os_wait = 0;
  ulint   spin_wait_count = 0;

  ut_ad(rw_lock_validate(lock));
  ut_ad(!rw_lock_own(lock, RW_LOCK_S));

lock_loop:

  if (rw_lock_sx_lock_low(lock, pass, file_name, line)) {

    if (count_os_wait > 0) {
      lock->count_os_wait +=
        static_cast<uint32_t>(count_os_wait);
      rw_lock_stats.rw_sx_os_wait_count.add(count_os_wait);
    }

    rw_lock_stats.rw_sx_spin_round_count.add(spin_count);
    rw_lock_stats.rw_sx_spin_wait_count.add(spin_wait_count);

    /* Locking succeeded */
    return;

  } else {

    ++spin_wait_count;

    /* Spin waiting for the lock_word to become free */
    os_rmb;
    while (i < srv_n_spin_wait_rounds
           && lock->lock_word <= X_LOCK_HALF_DECR) {

      if (srv_spin_wait_delay) {
        ut_delay(ut_rnd_interval(
            0, srv_spin_wait_delay));                          // 加锁失败,
调用 ut_delay
      }

      i++;
    }
```

```
    spin_count += i;

    if (i >= srv_n_spin_wait_rounds) {

      os_thread_yield();

    } else {

      goto lock_loop;
    }
...
ulong srv_n_spin_wait_rounds  = 30;
ulong srv_spin_wait_delay = 6;
```

上述源码可知，MySQL 锁等待是通过调用 ut_delay 做空循环实现的。

InnoDB 层有三种锁：S（共享锁）、X（排他锁）和 SX（共享排他锁）。 SX 与 SX、X 是互斥锁。加 SX 不会影响读，只会阻塞写。所以在大量写入操作时，会造成大量的锁等待，即大量的 PAUSE 指令。

分析到这里，我们总结一下影响吞吐量的两个因素：

- 自旋的时长，在 MySQL5.7 以及之前版本的源码定位为：spin_wait_delay * 50。
- Intel CPU PAUSE 的指令周期。

接下来，我们就从这两方面入手，评估优化空间以及效果。

4. 针对 PAUSE 指令和 spin 参数优化与探索

4.1 MySQL spin 参数优化

4.1.1 MySQL 5.7 spin 参数优化

我们可以基于现有 MySQL 版本、硬件等方面，来寻找优化点。

MySQL 针对 spin 控制这块有个参数可以调整，根据参数特点进行相关优化：

innodb_spin_wait_delay

innodb_spin_wait_delay 的单位，是 100MHZ 的奔腾处理器处理 1 毫秒的时间，默认 innodb_spin_wait_delay 配置成 6，表示最多在 100MHZ 的奔腾处理器上自旋 6 毫秒。

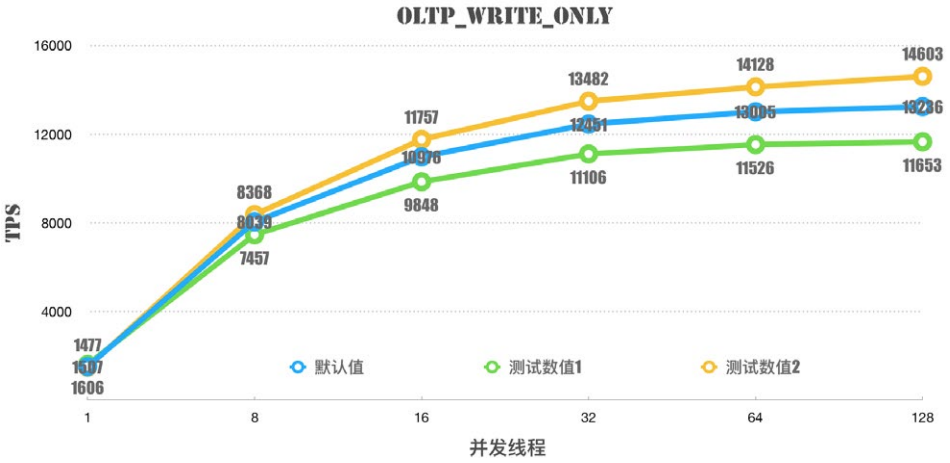
innodb_sync_spin_loops

当 innodb 线程获取 mutex 资源而得不到满足时，会最多进行 innodb_sync_spin_loops 次尝试获取 mutex 资源。

其中 innodb_spin_wait_delay 参数对 PAUSE 运行时长是有影响的。针对此参数，我们进行调优测试。

具体项	默认	测试数值 1	测试数值 2
innodb_spin_wait_delay	6	12	3
innodb_sync_spin_loops	30	30	20

同样，针对上述参数优化，我们通过基准测试来对比性能和效果：



可以总结为：

- innodb_spin_wait_delay 的调整对 TPS、QPS 一定影响，其值趋于小，则 MySQL 性能有提升。反之，下降。
- innodb_spin_wait_delay 参数调整性能优化效果有限，性能提升的幅度还是无法满足线上业务需求。

4.2 MySQL8.0 spin 新特性移植

4.2.1 spin_wait_pause_multiplier 移植

针对 Skylake CPU，PAUSE 造成的吞吐量下降，我们对 MySQL 5.7 spin 控制参数 innodb_spin_wait_delay 的调优并未取得明显效果。

于是，我们将目光投向了 MySQL 8.0 的新特性：MySQL 8.0 针对 PAUSE，源码中新增了 spin_wait_pause_multiplier 参数，来替换之前写死的循环次数。

4.2.2 spin_wait_pause_multiplier 实现

MySQL 8.0 源码中，之前循环 50 次的逻辑修改成了可以调整循环次数的参数：spin_wait_pause_multiplier。

```
ulint ut_delay(ulint delay) {
  ulint i, j;
```

```
  /* We don't expect overflow here, as ut::spin_wait_pause_multiplier
is limited
  to 100, and values of delay are not larger than @@innodb_spin_wait_
delay
  which is limited by 1 000. Anyway, in case an overflow happened,
the program
  would still work (as iterations is unsigned). */
  const ulint iterations = delay * ut::spin_wait_pause_multiplier;
  UT_LOW_PRIORITY_CPU();

  j = 0;

  for (i = 0; i < iterations; i++) {
    j += i;
    UT_RELAX_CPU();
  }

  UT_RESUME_PRIORITY_CPU();

  return (j);
}
...
namespace ut {
ulong spin_wait_pause_multiplier = 50;
}
```

4.2.3 移植 spin_wait_pause_multiplier patch 优化

既然 MySQL 8.0 参数 spin_wait_pause_multiplier 可以控制 PAUSE 执行的时长，那么就可以减少该值，从而降低整体 PAUSE 影响。

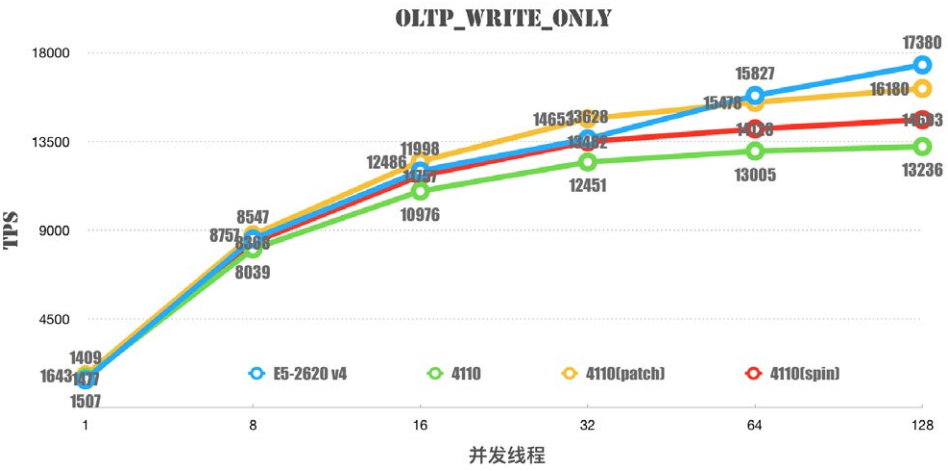
了解 MySQL 8.0 相关代码后，我们将该 patch 移植到线上的稳定版本：

```
MySQ >select version();
+-------------------+
| version()         |
+-------------------+
| 5.7.26-29-mt-log |
+-------------------+
1 row in set (0.00 sec)

MySQL>show global variables like '%spin%';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| innodb_spin_wait_delay            | 6     |
```

```
| innodb_spin_wait_pause_multiplier | 5     |
| innodb_sync_spin_loops            | 30    |
+-----------------------------------+-------+
3 rows in set (0.00 sec)
```

由上述可知，Silver 4110 的 PAUSE cycles 是 E5-2620 v4 的 14 倍左右。基于此，将 innodb_spin_wait_pause_multiplier 值调整为默认值的 1/14，取稍大值：5。即将该参数由原默认的 50 调整为 5。

最后，还是通过二维折线图来对比该 patch 调优后的基准测试数据：



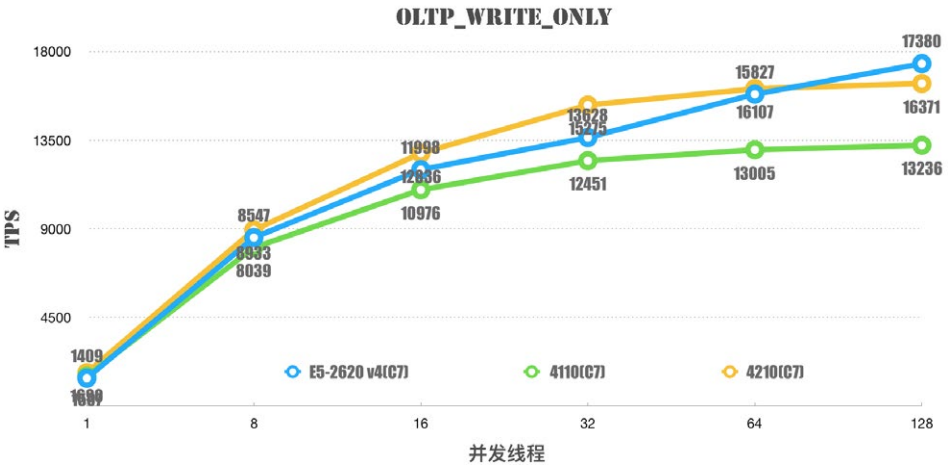
- Silver 4110 移植 spin_wait_pause_multiplier patch，并调整优化后，4110（patch）性能有了较大的提升。
- Silver 4110（patch）相对调优 innodb_spin_wait_delay 性能上更优。
- Silver 4110（patch）并发线程大于 64 的只写场景，性能略低于 E5-2620 V4，其他均优。
- 按照真实的线上读写比例，4110（patch）可以将吞吐量恢复到原先的性能水平。

4.3 PAUSE 指令周期优化

上述章节中，我们测出 Cascadelake CPU PAUSE 周期下降了。在跟 Intel 技术专家确认后得知：从 Purley 的第二代产品 Cascadelake 开始，Intel 将 PAUSE 的

指令周期降低到了 44。(估计 Intel 也发现了第一代增加 PAUSE 周期后的性能瓶颈问题。)

我们针对第二代 CPU 产品继续做基准测试，来看一下性能表现：



接着用 perf diff 来对比一下 4110 和 4210 在 ut_delay 上的开销：

```
# Baseline  Delta Abs  Shared Object                      Symbol
# ........  .........  .................................  .....................................
#
    15.62%     -8.01%  mysqld                             [.] ut_delay
               +1.27%  [kernel.kallsyms]                  [k] native_safe_halt
     0.06%     +0.74%  [kernel.kallsyms]                  [k] reschedule_interrupt
     0.80%     +0.59%  [kernel.kallsyms]                  [k] native_queued_spin_lock_slowpath
     1.25%     +0.58%  libpthread-2.17.so                 [.] __pthread_mutex_cond_lock
     1.49%     +0.43%  libpthread-2.17.so                 [.] pthread_mutex_lock
     6.92%     +0.37%  mysqld                             [.] 0x000000000036a168
               +0.28%  [kernel.kallsyms]                  [k] mlx5_eq_int
     0.58%     +0.23%  mysqld                             [.] mtr_t::Command::prepare_write
     0.94%     +0.23%  [kernel.kallsyms]                  [k] __schedule
               +0.21%  [kernel.kallsyms]                  [k] mlx5e_poll_tx_cq
     0.74%     +0.21%  [kernel.kallsyms]                  [k] _raw_spin_lock_irqsave
               +0.20%  [kernel.kallsyms]                  [k] call_function_interrupt
     0.29%     +0.19%  [kernel.kallsyms]                  [k] native_write_msr_safe
     0.02%     +0.16%  [kernel.kallsyms]                  [k] scheduler_ipi
               +0.15%  [kernel.kallsyms]                  [k] smp_call_function_many
               +0.15%  [kernel.kallsyms]                  [k] mlx5e_napi_poll
               +0.14%  [kernel.kallsyms]                  [k] eq_update_ci.isra.4
     0.05%     +0.13%  [kernel.kallsyms]                  [k] llist_add_batch
     0.03%     +0.13%  [kernel.kallsyms]                  [k] __x2apic_send_IPI_mask
     0.47%     +0.13%  [kernel.kallsyms]                  [k] _raw_qspin_lock
               +0.12%  [kernel.kallsyms]                  [k] mlx5_cmd_comp_handler
```

- 可以看到 4210 比 4110 占比下降了 8%。
- 由于 PAUSE 指令周期还是数倍于 E5 系列 CPU，4210 在高负载下，

PAUSE 的开销对 MySQL 吞吐量还是有较大的影响。而在 128 并发线程以下，性能相比 4110 有了较大的提升。按理，可以满足线上业务需求（该测试结果跟移植 spin_wait_pause_multiplier patch 性能测试数据曲线一致）。

5. 总结

最后针对本篇内容，我们可以做个简单的总结：

Intel 在新平台 CPU 产品调大了 PAUSE 指令周期，在高并发 spinlock 竞争激烈场景下，可能会造成程序性能较大损耗（特别是执行固定 PAUSE 次数的程序）。 针对 Skylake 架构 CPU（比如：4110 等）PAUSE 指令周期较长引起性能问题的优化方法如下：

将 MySQL 8.0 innodb_spin_wait_pause_multiplier patch 移植到线上稳定版本（或升级到 MySQL 8.0），通过降低 PAUSE 执行时长，来提升吞吐量。 如果是 OS 为 CentOS 6，可以升级到 CentOS 7，CentOS 7 本身 spinlock 优化，对 MySQL 性能也有一定提升。 最简单、直接的方法可以替换为 Cascadelake 架构 CPU。

针对 Cascadelake 架构 CPU，由于 Intel 本身在 PAUSE 周期已经优化，性能上已经做了修复。当然也可以采用上述优化方案，让性能提升一个台阶。

6. 作者简介

春林，2017 年加入美团，主要负责 MySQL 运维开发和优化工作。

招聘信息

美团 DBA 团队招聘各类人才，Base 北京、上海均可。我们致力于为公司提供稳定、可靠、高效的在线存储服务，打造业界领先的数据库团队。这里有数万各类架构的 MySQL 实例，每天提供万亿级的 OLTP 访问请求。真正的海量、分布式、高并发环境。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：美团 DBA 团队）

美团内部讲座｜周烜：华东师范大学的数据库系统研究

作者：周烜

【Top Talk/ 大咖说】由美团技术学院和科研合作部主办，面向全体技术同学，定期邀请美团资深技术专家、业界大咖、高校学者及畅销书作者，为大家分享最佳实践、互联网热门话题、学术界前沿技术进展等内容，帮助美团同学开拓视野、提升认知。2020 年 10 月 27 日，Top Talk 邀请到了华东师范大学周烜老师，请他带来题为《华东师范大学的数据库系统研究》的分享。本文系周烜老师分享报告的文字版，希望能对大家有所帮助或者启发。

/ 报告嘉宾 /

周烜

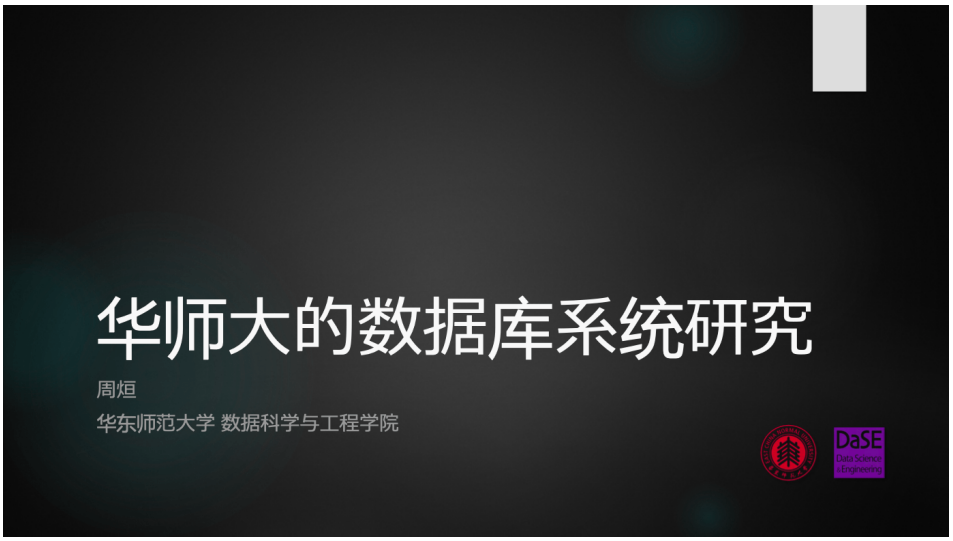
华东师范大学教授，数据科学与工程学院副院长

2001年本科毕业于复旦大学，2005年在新加坡国立大学取得博士学位，2005年至2010年期间先后在德国L3S研究中心和澳大利亚联邦科工组织从事科研工作，随后在中国人民大学信息学院任教6年，于2017年3月加入华东师范大学。研究兴趣包括数据库系统和信息检索技术。曾参与和负责多个国内外的科研项目和工业合作项目，积累了丰富的数据管理系统研发经验。研究成果被发表于众多国际一流的学术会议和期刊。凭借在分布式数据库领域的成果转化获得国家科技进步二等奖和教育部科技进步一等奖。入选教育部"新世纪优秀人才"支持计划。

/ 报告摘要 /

华东师范大学是国内为数不多长期坚持数据库内核技术研究的高校，在学术界和工业界均建立了较好的声誉。本次讲座将分享华东师范大学数据库团队近期的一些科研思路和研究成果。首先分析驱动数据库技术发展的主要因素，谈一谈未来有价值的研究方向。再聊一聊团队近来取得的一些有趣的研究成果，领域包括新硬件的数据库适配、分布式事务处理、HTAP、系统实现模块化（Modularization）等等。

00 引言



今天我代表华东师范大学的数据库团队，来分享一下对数据库这种技术或者这种产品的一些研究心得以及当前的研究成果。其实，高校跟企业实际上是处在两个不同的生态领域当中，高校更关注关于研究理论的一些问题，但是企业更多的关注企业本身的产品以及用户，所以两者面向的目标是不太一样的。但是经过我们在华师大这么多年的一些摸索，特别是我们自己研究上的一些摸索，以及跟企业合作的经历，我们觉得实际上高校和企业应该一起来做我们称为数据库系统或者基础软件的研究，因为只有这样才能够更好的推动这个行业本身的发展。

我的报告内容会分成两个部分。首先分享一下我们对数据库这种技术发展动态的看法，我们团队在数据库这个领域也做了很多年，包括我之前在中国人民大学也是做数据库系统的，我在这个领域里面有可能不到 20 年的积累。我希望能够提出我们的看法，并且能够得到大家的一些反馈，纯粹是做一种探讨，因为对技术的发展方向的探索，没有标准的答案。我觉得大家应该集思广益，共同去探讨，才能把这个问题看得更清楚，这是讲座前面一部分的内容。

后面一部分的内容我会聚焦到我们的一些研究成果上，这些研究成果可能就会比较技

术细节了，会更适合搞技术、数据库、系统的这些同学，但是我会尽量把讲座的形式变得更大众化一点，尽量用更通俗的方式去给大家介绍。我希望通过这种细节的介绍，也能够让大家了解一下，我们在设计系统的时候，通常一个工程师、一个研究者或者一个学者，他的思路大概是怎么样的，我不能代表所有的人，但是因为我们这个团队是一个典型的做系统的团队，所以这个思路可能仅代表了一部分做系统的学者的思路。

01 数据库系统的形态变化

1.1 什么引起了数据库系统的形态变化?

首先问大家一个问题，什么引起了数据库系统的形态变化？我们知道数据库系统实际上是一个有很长历史的系统，是现代软件开发的一个核心部件。任何应用都离不开某种数据库，但是我们其实也可以看到，如果你有一定的经验，比如大概 10 年的工作经验，你会看到数据库系统的形态是在发生变化的，10 年前用的很流行的东西，现在不见得普遍被采用。这个系统虽然有很长很长的历史，也很成熟了，但是它的形态还是在发生变化的。

什么在驱动数据库系统的形态变化？这是一个很重要的问题，也是比较有趣的问题，但并不是一个好回答或者能够全面的回答的问题。我这里就直接抛出我们对这个问题的一些看法，我们觉得数据库系统的形态变化，主要来自三个方面的推动力：

- 第一个方面是应用需求的变化。我们的软件产品一直在变得越来越丰富，应对的场景越来越多，实际上应用需求是在变化的，它的变化提出了不同的要求需求，它在促使数据库系统的形态在发生改变。
- 第二个方面在学术界其实比较少去触及的，就是软件开发模式的变化。数据库是 70 年代 80 年代设计的，面对的是当时的软件开发模式，我们知道在九十年代以后，软件开发模式实际上发生了很大的改变。软件开发的模式的变化，也在引起数据库使用方式的改变，也在推进数据库系统的演变。
- 第三个方面就是硬件平台的革新，硬件在改变处理器、存储器件，整个平台以

前是一个大型的计算机，大型机、中型机，然后现在其实就是云平台，这些东西的变化实际上是在影响数据库本身的形态的改变。

我们看到的推动力主要就是这三个。我们认为未来推动数据库变化的原因也不出于这三个，我们可以通过这个东西去预知未来应该朝什么方面去推进我们的数据库技术的进步。下面我就大概做一个简单的展开。

1.1.1 应用需求的变化

首先第一点就是应用需求的变化。我们认为这实际上是在推动底层技术，像数据库系统这种技术变化的一个主要原因。

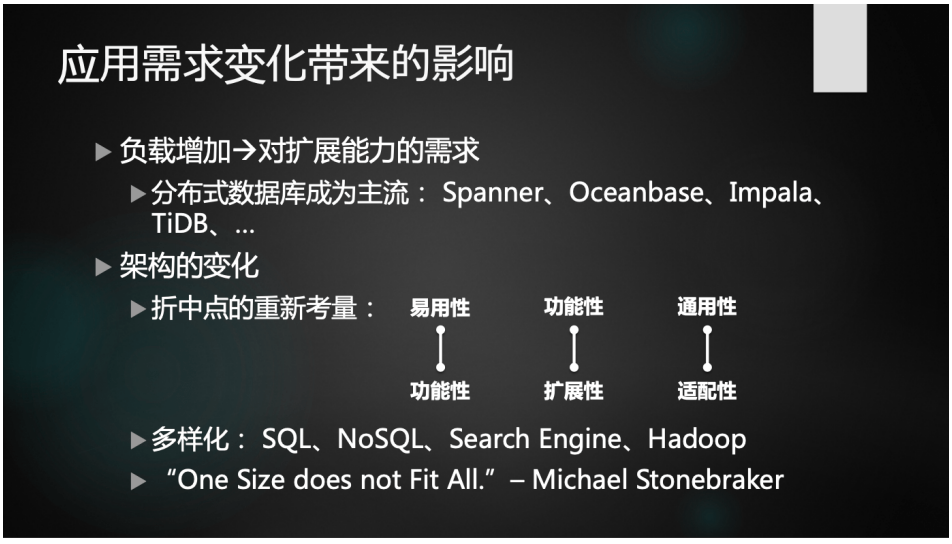
通过上面这三幅图大家能够很容易去理解数据库诞生的年代，那个时候磁盘作为一个存储介质，刚刚在市场上推广，磁盘取代了磁带，数据的随机访问变的可能了。那个时候对数据管理功能的需求一下子就增长了，当时出现了各种各样的数据库，包括网状数据库，包括后面的关系数据库，那个时候是我们叫前互联网时代就出来了。但那个时候应用的规模并不大，数据库的用户量一般，终端用户其实很少，对于一个银行来讲就是那些银行职员在使用数据库，普通用户在银行排队，他们不是终端的使用者。

后来进入到互联网时代之后，我们发现终端的使用者一下子就爆炸性的增长了。现在我们每个人有一个手机，随时都在使用手机上的 App，然后这个 App 他随时都会把请求发送给后台的数据库。我们看到应用规模在最近 20 年有一个很大的增长，如果

往后看的话，未来我们认为增长有可能还会继续。我们的工业互联网、物联网使用的话，我们的终端会变得更多，他可能对数据管理系统的压力会变得更大。

所以我们看到应用不断的扩张，给数据库这种底层系统有一个持续增长的压力，要应对这样的压力，以前对数据库的设计，它逐渐就变得不太实用，必须去革新，必须去改变它。

这个是我们看到的一个最主要的推动力。但除了应用规模的扩展，当然就还有一个应用领域的扩展，最开始的数据库它就是金融领域或者电信领域去使用，并不会在互联网、销售或者传媒等等这些领域去大规模的使用，但是我们发现 IT 的渗透到各行各业之后，它的应用范围也在增加，应用范围增加对传统的系统来讲是不友好的，或者是说你的传统数据库系统对这些应用是不友好的，所以这个也在推动它的一个变革。



应用需求带来什么变化？我们回顾历史的话，我们看到分布式数据库现在变得越来越被大家所需要，大家看到了谷歌的一些分布式数据库的产品，它现在作为一种标杆的产品，然后国内也有一些分布式数据库的场景，包括美团在内，听说美团内部也在研制自己的分布式数据库，实际上是在对需求的负载增加的一个应对。

应对它实际上并不是一个很简单的事情，如果要增加你的数据库的扩展性，有时候你

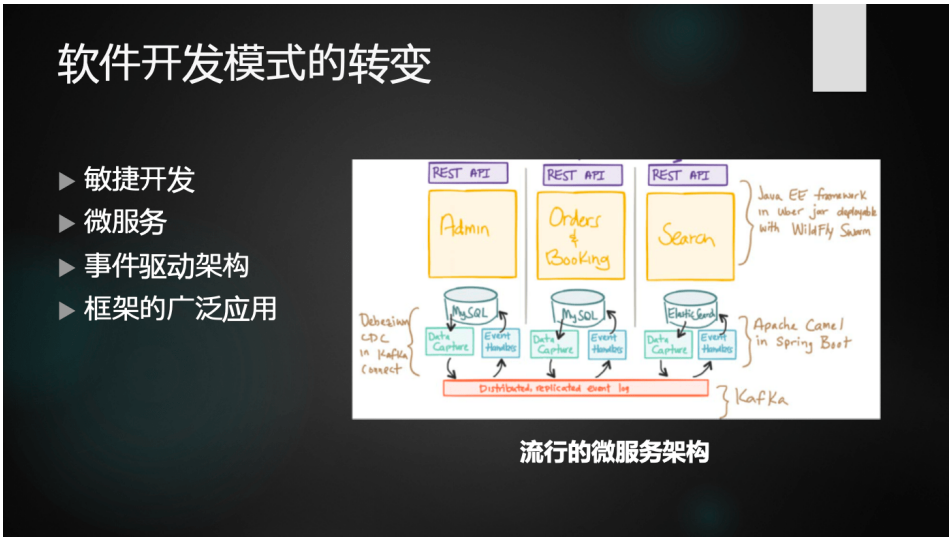
必须要重新去设计你的数据库的架构。我们通常做系统的同学应该了解，其实在做系统的时候，你需要做很多折中的考量的，有些东西是不能兼顾的，比如说你的功能性和应用性对吧？一个东西特别简单去使用的话，它功能性有时候就是比较简单，它复杂的功能处理不了，或者是你的功能很复杂的时候，你的扩展性又上不去。

在系统设计的时候你必须去做一种权衡，去获得一部分这方面的能力，你就必须丢失其他的一些能力。当你需要它的扩展性非常强的时候，你有可能就要去重新考量它，你要去丢弃什么东西，你要去忍受什么其他的一些东西，然后这个时候就产生架构的变化，这样的话我们就会发现有新的系统出来，比如说以前的是 SQL，我们现在有讲 NoSQL 大家认为它的扩展性容易做的更好一些，然后会有其他形态的一些数据管理产品。

针对这个问题，学界工业界都有很多的讨论，Michael Stonebrake 大概 10 多年前发表的一些言论，就说“One size does not fit all”，你不太能指望某一个系统能够能够处理所有的应用需求。因为不同的应用需求，你可能必须要做不同的折中、重新的考量，你只能顾此，你顾此只能失彼，所以这样就产生了一个多样化的形态。

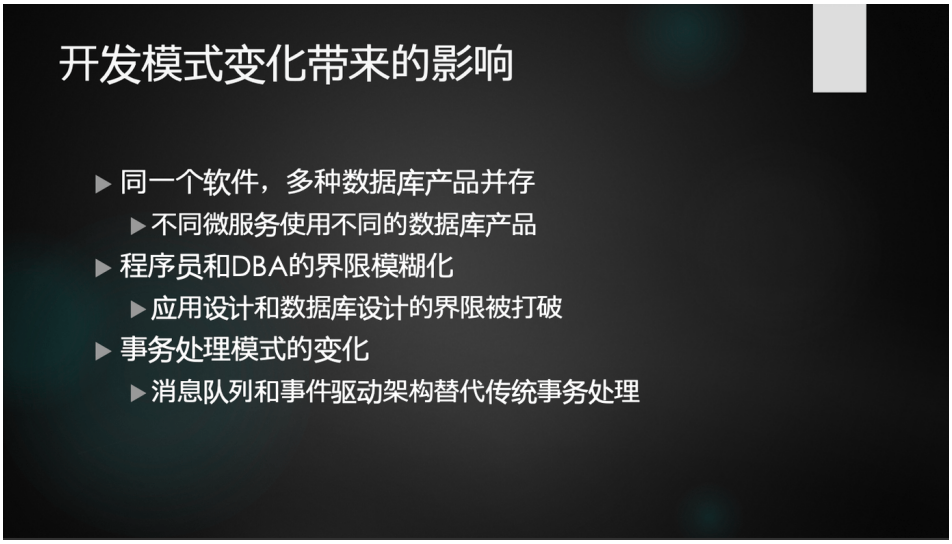
我们现在看到的数据库，如果你们在使用的话，你会面临很多选择，你到底用 MySQL 还是 MongoDB 对吧？你的分析的时候你要用什么？用 Hadoop 还是用传统的数仓 MPP 产品，有不同的需求，可能有不同的考量。这个我们看到应用需求它的变化，它的增加，它在实际上对这个系统起到了一个很重要的推动的这个作用，这是我展开的第一点。

1.1.2 软件开发模式的转变



然后，第二点就是我们发现软件开发模式的变化，也在促进数据库本身的改变。就像我刚才提到的，现在用到的很多软件开发的模式跟以前不一样。以前关系数据库这样的产品刚刚被广泛应用的时候，当时的软件开发它是以数据为中心的，一个数据库设计出来，有好多应用都会去用它，它是一个 Shared 底层系统。那个时候数据库的设计过程，它是相对独立的，DBA 根据 App 开发人员的需求、用户的需求，以 DBA 的方式去设计数据库，按照对数据模型的理解，把它设计得非常的规整，要满足各种各样的范式，然后给不同的应用去用。但现在我们发现一旦用到微服务这种新的开发模式的时候，很多时候这种横向的分割变得不是那么重要了。原本数据是一层、应用逻辑是一层，这两层之间的解耦是很重要的，但现在不是了。

现在是微服务的形式，是纵向的切割，把业务整个分成一块一块的，每一块里面都有单独的数据库，有单独的功能设计，这弱化了数据库跟应用之间的界限，强化了应用里面不同模块的界限。这样的话，每个模块可以用不同的数据库产品，比如说一个设备用 MySQL，另外一个设备可能用 MongoDB，第三个设备可能用 ES，这些模块之间的数据有同步有交互，可以用一些比如像事件驱动的架构，像 Kafka 这种 MQ (Message Queue) 去连接在一起，形成一个总体的架构。



这种设计跟以前的数据库是不太一样的，对于数据库本身的要求也不太一样。不同的 Service 会用不同形态的数据库产品，根据需求或者根据软件开发者的习惯，去采用各种缓存、消息队列等，去把这些东西给嫁接在一起。这样的形态对数据库有不同的要求，所以 NoSQL 被很多人接受。

在某些软件开发的场景下，NoSQL 就是比关系数据库使用起来更简单。然后事务的处理方式也变得很不一样，现在的消息队列在事务处理中，它的权重非常的高，而不是完全依赖于传统数据库内部支持事务处理的模式。这个是我们也看到这样的一些变化，这个是软件开发模式带来的一些改变。

1.1.3 硬件平台的革新

第三个方面就是平台的革新。其实我们不会对传统的数据库的硬件平台灵活性有太多的关注，但现在数据库产品面向的基本上都是云平台，不是以前的 IBM 大型机了，不是 Oracle 那个时候的硬件平台。我们面对的是一个云平台，云平台自身是在发展的。以后的云平台其实可以预见得到，它不单单是现在我们看到的，是由一个个虚拟机或者是一个个容器组成的一个计算平台，还有可能就像一个大型的计算机一样，只是这种大型计算机它的资源非常的丰富，需要调用什么样的资源都可以获得，需要更多的内存、CPU、存储，都可以直接的获取。

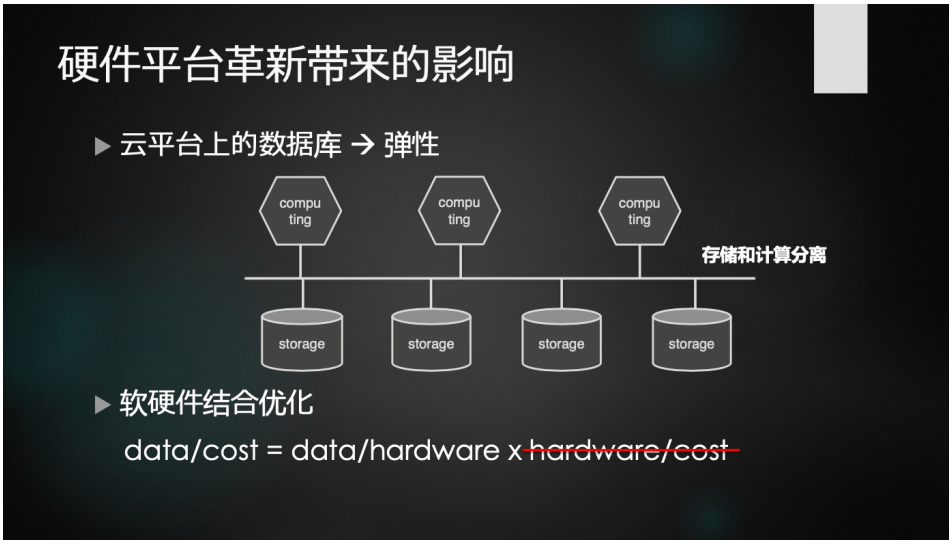
云平台通过比较高效的网络方式把这些资源全部连在一起，然后给用户的接口也很简单，很多维护功能是在云平台内部去实现的。数据库要扎根在这样的一个云平台上面，其实对数据库系统就会有新的要求。以前的数据库，大家都记得有几种架构可以选择，叫 Shared Everything、Shared Nothing、Shared Disk 这样的一些架构。

但是我觉得在“云”层面上，数据库其实不再是那么简单去划分的，就是说数据库系统的产品，必须要做到具有很好的弹性。任何的资源在短缺的时候，可以通过云的这

种方式很快的把资源调度过来，从而增加数据库的能力。对用户的话，只是提供一种数据库的服务，用户用多少？就提供多少。这样的一个云架构下的数据库，形态可能跟以前要有一些变化。除了这种云的体系之外，其实还有一些新硬件出现，硬件种类也变得越来越多。不同的种类的硬件，在不同的条件下，算力也在不断增加。当然除了计算，还有存储也发生很多变化，把这些放到云的里面去，云的资源变得更加丰富，对数据库的要求也会变得更高。

因此，我们看到这种硬件平台的变革，它实际上在推动数据库的一些发展。现在大家很常见的就是这种计算与存储分离的这种架构的数据库。我把数据库本身这种系统，它的计算层跟存储层完全的分割开，计算层可以自己扩展，存储层也可以自己扩展，两层就通过云这种高速的、互联的通道能够连接在一起，这种实际上就是针对云的一个特殊的处理。我们知道存储是便宜的，所以在存储需要扩展的时候，没有必要在扩展存储的基础上去加 CPU 的资源，因为 CPU 比较贵。如果分开扩展的话，这样确实会在成本上有极大的提升。

未来各种资源加进去之后，它都有可能有扩展的需求。比如说缓存，新的存储器，新的内存加进去之后，有可能需要它跟底层的存储分开，进行一个隔离，再分别去扩展，这都是有可能的。但现在的数据库产品其实面对这种扩展能力，实际上是非常有限的，存储和计算分开扩展到一定程度，实际上它的能力就达到一个峰值了。那么怎样去推动它进一步的这种弹性的增加，实际上是一个挺难的问题，但是也是挺有趣的问题。



新硬件对数据库产品的影响，这里有一个简单的公式，公式的左边叫做 Data/Cost，单位数据处理，单位代价上面可以处理多少数据。这个是我们需要提升的，因为可以想象以后的数据量会越来越大，如果不把单位价格上面能处理的数据这个值提升的话，应对数据的能力就没办法提升。数据越多，需要花费的资源或者代价就越多，这个是我们不希望看到的。希望左边这个式子中 Data/Cost 的值随着技术的进步，它可以逐渐的提升。

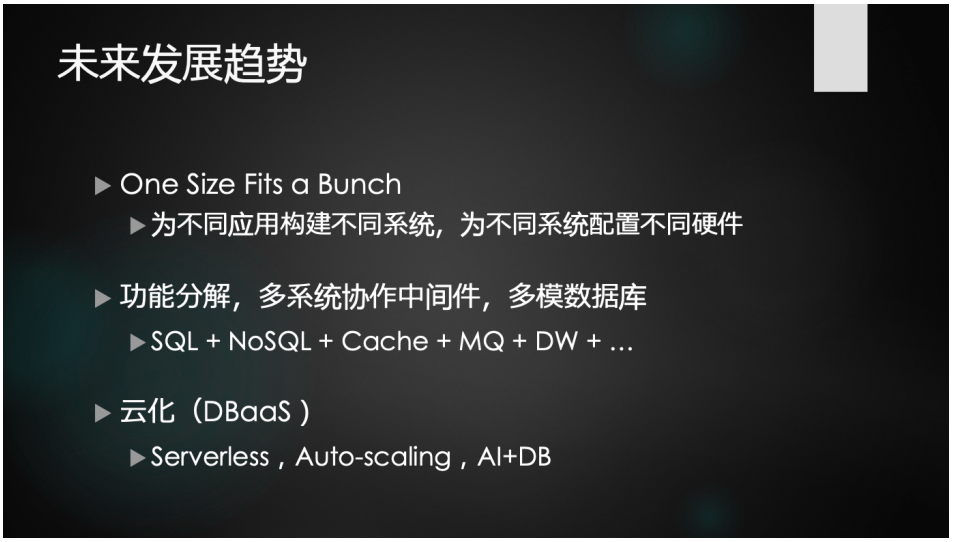
然后把左边这个式子它分解一下，分解成 Data/Hardware 和 Hardware/Cost 的一个乘积。这其实是很简单的一个因素分解，大家能够看得很明白。我们发现后面这个式子 Hardware/Cost，实际上它的增长在逐渐的趋缓甚至停滞，主要的一个问题就是单位价格能够买到的硬件资源，要在这上面做更进一步的提升，会变得非常的困难。

最后，如果想实现 Data/Cost 的提升的话，只能去提升左边这个式子 Data/Hardware，这个是未来一个很明显的趋势。怎么样能够提升 Data/Hardware？就是单位硬件下处理数据的能力要怎样提升？我们认为只有两种途径，第一种是硬件定制化，面对不同的应用需求，需要为这种应用需求做特殊的硬件。其实我们能看到像 GPU、TPU 这种出现，其实就已经在揭示这种规律了，专用硬件效率总是要比通用

硬件好的。然后软件是一样的，专用的软件的效率肯定比通用的软件好。

这个趋势我觉得可以从长期来看，短期可能并不是那么的明显，但长期来看的话，这个过程应该是不可阻挡的，也就是说我们可能会面临要去为应用去定制系统，要为专门的这种系统配置适合它的硬件。

1.2 数据库系统的未来发展趋势



我们刚才讲了三点，第一点就是应用的变化在推动系统的演进；第二个是软件开发模式的变化实际上也在带来系统的功能的一种变革；最后是硬件平台。因此，我们觉得未来数据库发展的趋势：

- 第一个是“One size fits a bunch”，为不同应用构建不同系统，为不同系统配置不同硬件。就像 Michael Stonebraker 说的那样，数据库不是一种系统就能应对所有的应用了。我觉得 Stenberg 当时是针对的是应用负载的增加带来的问题而提出的观点，但是硬件的发展瓶颈到来的时候，同样会引领我们认同这个观点，“One size does not fit all”，不可能为所有应用产生一个系统，应该是一类应用对应一套系统的，这个是我们看到的一个很明显的发展的趋势。

- 第二个发展的趋势是现在的系统变多了之后，它需要协同，会有各种中间件，还会有各种形态的数据库，我们需要把它协同起来，这些数据库形态太多，对程序开发人员，维护人员都是一个 Disaster，代价会变大，怎么更好地把它协同起来，这也是未来一个发展的方向，我们会看到这些系统相互之间会变得越来越配合，越来越融合。
- 第三个就是云平台成为一个主流的硬件平台之后，我们看到数据库会朝云这个方向有更深入这种发展，它会更适合云这种形态，它的弹性，自我维护、自我修复的能力是会进一步提升的。这个是我们对未来发展的展望。

02 华师大的数据库系统研究

2.1 研究团队

然后介绍一下我们现在这个团队，华东师范大学数据科学工程学院大概有 20 多个老师，大概 10 个老师是从事数据库内核的研究的。整个团队的历史是超过 20 年的，我们近 10 年其实做了非常多的系统内核研发的工作，跟业界的很多的公司也有合作。我们的学生其实也做了很多工程性的工作。

去年我们有拿到一个国家科技进步二等奖，这个是基于我们当时和某银行一起做的一款数据库产品，是我们基于 OceanBase 的一个早期开源的版本上实现的一个系统，具体的这种内容我就不做过多的介绍，这个系统实际上在某银行得到了比较深入的应用，是我们比较引以为豪的一个研究成果。

2.2 研究成果

我们团队的研究其实还是蛮广泛的，我们在事务型数据库和分析型数据库其实都有研究，但我们更多的精力还是集中在事务型数据库上面。然后接下来，我会介绍一些典型的研究成果，让大家了解一下我们的研究是在做一些什么事情，主要分三个部分，第一个是分布式事务，第二个是数据库系统解耦合，第三个是新硬件。

2.2.1 分布式事务

分布式事务是一个几十年来大家都在探讨的话题，实际上也是有一定的争论：分布式事务到底合不合用？我们在使用事务处理这种功能的时候，是不是应该去规避分布式事务？还是我们应该进一步去增强数据库支撑分布式事务的能力，让程序员不要刻意去规避分布式思维？这实际上是一个疑问，目前没有一个明确的答案。

如果分布式事务确实是不行的，那我们就应该做一些其他方面的处理，来弥补分布式事务本身的缺陷，这样做有两种方式：

- 第一种是干脆不要数据库提供分布式事务功能，将事务处理推给应用，根据应用的特点去规避这种分布式事务的一些缺陷。
- 第二种是尽可能提升集中式事务处理的能力，集中式事务能够达到和分布式同样的效果，就不再需要分布式事务了。

我们认为现在以 NoSQL 为代表的这些系统的推动者，实际上是持这样的观点的。比如典型的 NoSQL 客户系统 MongoDB，它的一般的事务处理或者数据库的访问，都是 Single Document 一个文档一个文档去处理的。实际上，就是如果真要进行复杂的事务处理，那就到上层应用去处理，就用最定制化的方式去应对这种事务，它的效

率可以比较高。

另外一种观点认为分布式事务本身应该是可行的，我们应该提升数据库处理分布式事务的能力，解放开发者。

这样的观点就是那些推动 NewSQL 这一类系统的人所持的观点，比如说谷歌的 Spanner、TiDB、OceanBase。那么想要把分布式事务做好，怎样去优化，提升分布式事务的能力？首先要处理异常，分布式事务最害怕的一个问题就是出现异常，出现异常之后，如果是一个分布式事务，一旦事务锁掉一个节点上的数据，另一个节点出现故障的话，就会很麻烦。那为了处理异常，然后以 Spanner 为代表，使用了很多高可用的系统架构，用 Paxos/Raft 创建这种在云平台，在这种廉价计算机上同样能够有高可用能力的基础设施，在这个上面我可以去规避分布式事务所遇到的这种异常的问题。

但除了异常问题之外，实际上还有分布式事务的扩展性的问题。虽然分布式事务可以比较放心地应用于可靠的、高可用的系统上，但是它的性能会比较差。因此，我们必须要优化它的性能，为此学术界也做了很多尝试，对于团队来讲，最近几年，我们也做过一些研究和尝试。接下来我就大概讲一下我们的大概思路。

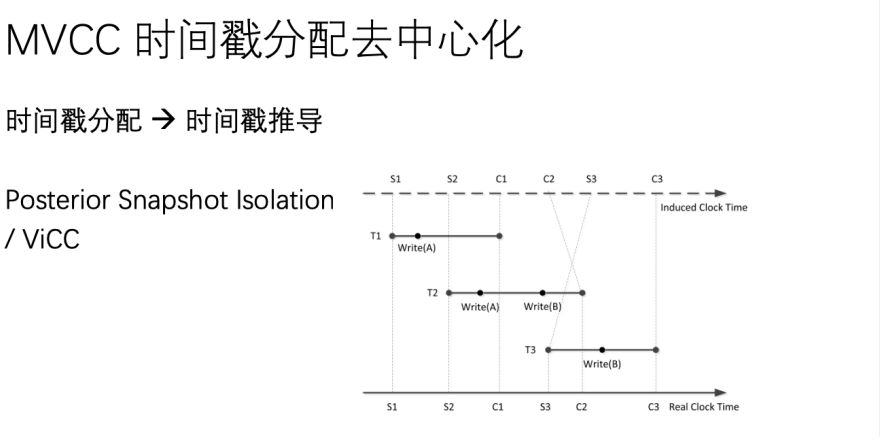
首先，我们要了解到底是什么限制了分布式事务的扩展能力，大家是比较公认的一种观点是制约分布式事务扩展的主要是事物之间的这种阻塞 Blocking。你可以想象一下，当一个事物它去访问一个数据，特别是修改一个数据之后，它会加锁，然后在加锁的过程中，又要去跟其他的节点进行各种通信。

其实这种通信有时候是很耗时的，有时候甚至要跟异地的节点，比如说要做高可用，就需要跟异地的节点建立通信。在加锁的过程中去通信，由于通信的过程很长，然后就会把加锁的时长变得特别的长，阻塞就会变得很严重。一旦事务处理的阻塞时间增加，它的事物的吞吐有可能会受到很严重的影响，特别对于有一些热点的数据出现的话，不是时长增加一倍，性能就降低一倍的，有时候时长增加一倍，性能可能会降低若干倍，这是一个很麻烦的问题。

所以说，真的想要解决分布式的扩展能力的话，在已经有高可用的前提下，我们最重要的目标就是要降低阻塞。怎样降低阻塞？其实可以用到很多的技术。比如说 MVCC/OCC、MVCC 是多版本的数据管理，它可以降低阻塞，这个很直观，就是我一个数据有多个版本，当我的一个事务去改动数据的时候，我直接产生一个新的版本，这样就不用去阻止别人读你的旧版本。

因为你有多个版本，新版本在产生的过程中，你没有办法去读，可以去读旧版本不用阻塞，这就是 MVCC 的使用。然后 OCC 也是一样的，就是 OCC 就是乐观并发控制，也就是说默认不需要加锁，到最后再来检测这个事务是不是执行正确，如果执行不正确推翻就行了。然后还有一些锁的优化，通过各种各样的技术，来把阻塞的现象把它尽量的减少，从而来提升分布式事务的扩展能力。

MVCC 时间戳分配去中心化



在 MVCC 上我们做过一些工作。当然 MVCC 有一个时间戳分配的问题，就是说它来判断一个事务或者一个数据或者一个数据的版本，它是不是应该由某一个事务去读取的话，它要通过一些时间戳的判断来做。时间戳的分配很麻烦，按道理来说，它是应该有一个中心的时钟，大家都去中心的时钟去拿时间戳，这样就可以保证事务处理正确无误。但通常如果有一个中心的时钟的话，那扩展性就受限了，比如谷歌的 Spanner 用一些原子钟去规避这个问题。然后我们做的一个研究就是去中心化，去

优化了 SI 的隔离级别。

SI 是一种典型的 MVCC 的隔离算法或者并发控制的算法，它是需要时间戳的。我们做了一个去中心化。想法是我在给事务分配时间戳的时候，不是在事务开始的时候分配，而是在事务要结束的时候，根据它跟其他事务的关系、冲突情况，来给它指定一个合适的时间戳，所以叫后验时间戳，Posterior SI。这种方式使得我们可以不需要用一个中心的时间戳去做这个事情，但这个东西做起来其实蛮复杂的，我们大概 5 年前做了这件事情，最后有一些实验没有实现在现实的系统里面去，因为实现相对来说比较复杂，而实际应用中使用一个统一的中心的时间戳，基本的应用还是可以满足的。

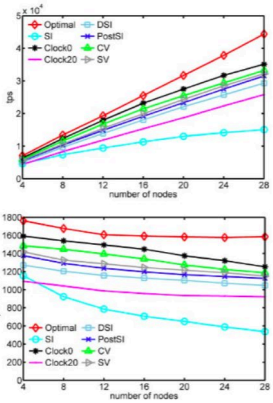


Fig. 7. TPC-C Performance (20% distributed txns)

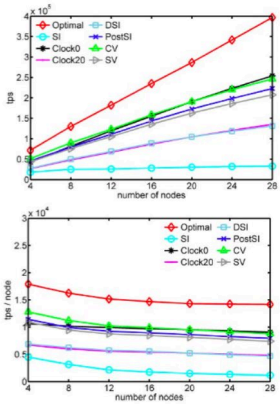
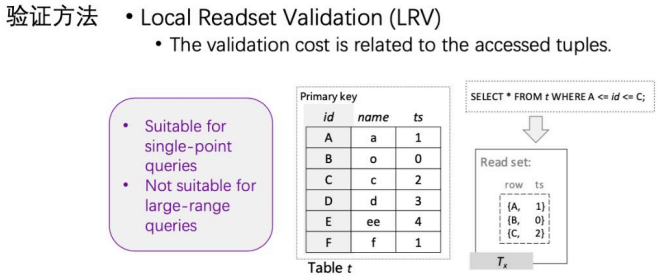
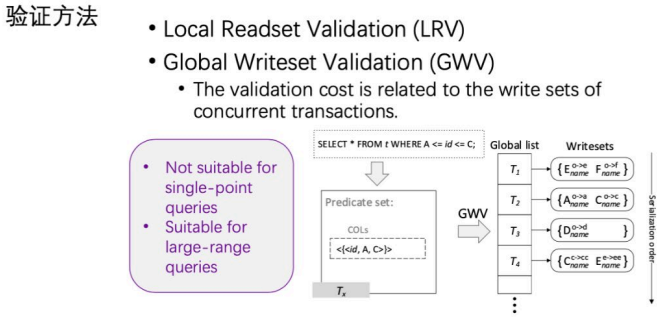


Fig. 9. SmallBank Performance (20% distributed txns)

我们做了很多的实验，实验结果可以表明我们的方法，当扩展到一定程度的时候，这种时间戳的分配是不会成为一个扩展性的瓶颈的。

降低 OCC 的阻塞时间



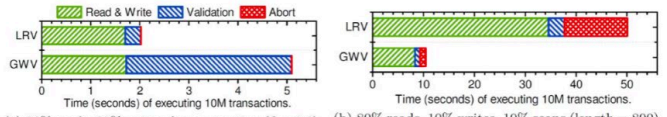


我们也做了一些 OCC 的工作。OCC 在事务访问数据的时候，就放开让事务去访问，访问完了事务要结束的时候会做一个验证叫 Validation，做完验证，再决定这个事务是提交还是回滚。这种方式也是降低事务之间阻塞的一种方法。其实这种事务的最后的正确性验证，有时候会挺耗时间的，所以在这个上面做了一个优化。

我们认为做正确性验证的方式有两种，一种主要的方式叫 Local Readset Validation，每个事务把它读过的数据记录下来，最后再去查读过的数据有没有被改动过。这种方式的缺点是当读取的东西特别多的时候，它的代价就会相当的大。

然后，还有一种方式叫 Global Writeset Validation，这种方式就是我不去记录每个事务读过的数据，只记录现在有多少正在运行的事务改动了那些数据，也就是说记录的是那些被改动的数据。然后读的时候，观察它的范围有没有包括改的数据，如果包括了验证就失败。这种方式对读取数据内容比较多的事务是友好的，但对那种小的、短的事务并没有那么友好。

- Local Readset Validation (LRV)
- Global Writeset Validation (GWV)

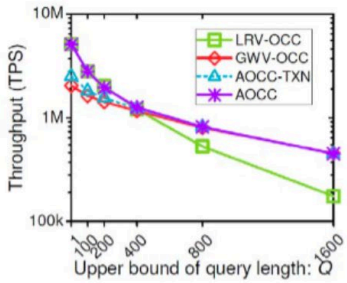


(a) 80% reads, 20% writes, low contention ($\theta = 0.6$), 32 worker threads.

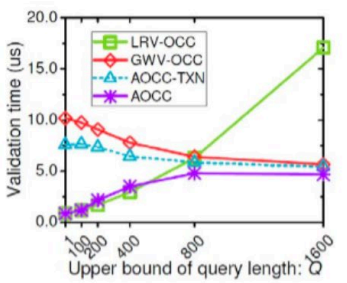
(b) 80% reads, 10% writes, 10% scans (length = 800), high contention ($\theta = 0.9$), 32 worker threads

Adaptive Optimistic Concurrency Control (AOCC) that combines LRV and GWV works well in any workload.

所以我们就做了一个叫 AOCC，Adaptive OCC，就是把这两种方式给结合起来，我们会判断一个事务的运行情况，如果读取的数据很多，就用 Writeset Validation；如果读取得很少，就用 Local Readset Validation，这样的话就把两种方式的优点结合起来。

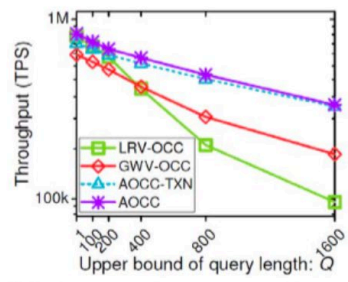


(a) Transaction throughput.

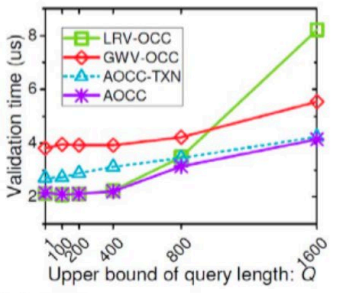


(b) Validation time of each transaction.

YCSB



(a) Transaction throughput.



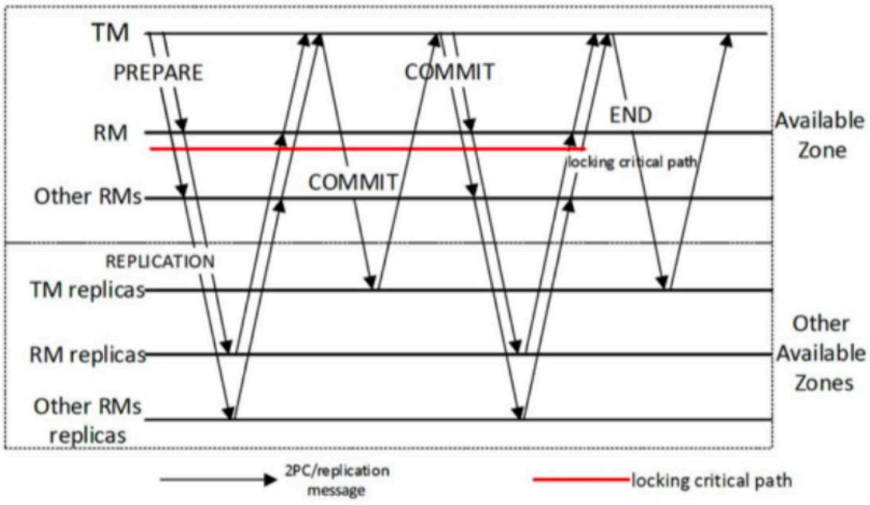
(b) Validation time of each transaction.

TPCC

我们做了一些实验，实际上结果确实证明这种方式没有极端的情况的缺陷。因为以前的那种就是读集 Receipt Validation 和 Receipt，德行在在各自的极端情况下都会呈现出特别差的一个性能。但是我们这种方法实际上它是比较均衡的。

跨区域高可用系统的锁时长

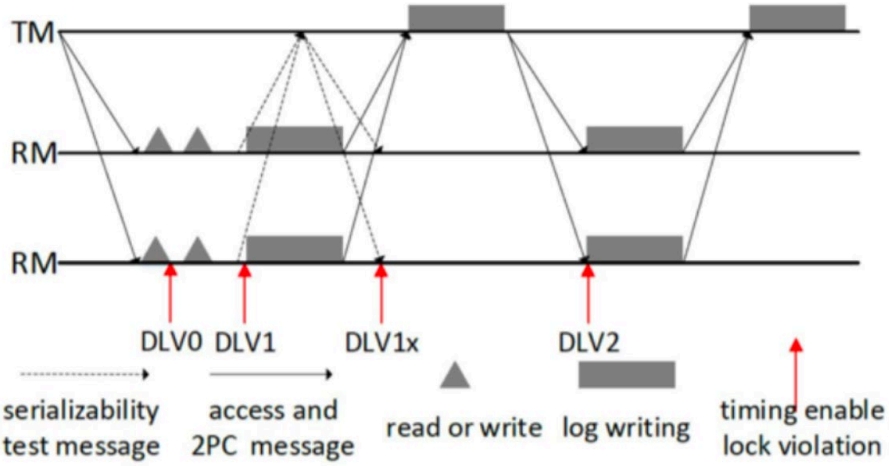
然后第三个工作也是关于分布式事务的，就是我们做了类似于 Spanner 这样的系统，跨区域的高可用的系统。



就像我刚才提到的，一旦加锁，加锁的过程中，出现了跨区域数据的通信，这个持锁的时间就会特别的长。这里是一个例子，在跨区域高可用的系统上做的一个两阶段提交。可以看到红色的线是一个加锁的过程，这个过程已经是在一个正常的事务里面最短的加锁过程了，它是在事务提交开始之前加锁，一直到事务提交完成之后释放锁。

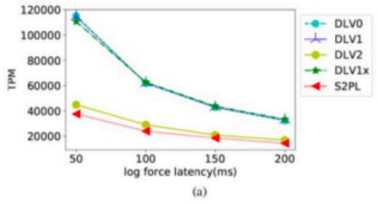
对一个普通的事务来说，它本身就是要加锁的，但这个加锁的时间可以看到，在加锁过程中，Prepare 阶段会有大量的本地节点跟异地节点之间的同步，然后在 Commit 阶段，同样的也有大量的本地节点跟异地节点之间的同步，这样的一个同步是很耗时间的，如果在这个时间上去做加锁的话，一旦遇到热点的数据访问，这个事务处理的性能就会极度的下降。所以在这样的条件下，我们就想可不可以用提前释放锁的方式

去规避加锁，缩短加锁的长度，直接降低阻塞概率。

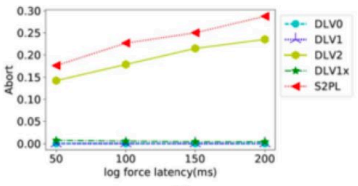


然后我们就设计了叫 DLV，LV 的意思是 Lock Violation，实际上就是提前释放锁。DLV 的话我们叫 Distributed Lock Violation，同样是一个两阶段提交的一个协议。我们就看在什么地方释放所，它的效率是好的。我们选择了四个时间点：

- 第一个是在 prepare 阶段之前访问数据的时候，访问一个数据就放一个数据锁，相当于就不加锁，这是一种最极端的方式。但这种方式到后面的回滚率会非常的高，只能通过验证的方式来判断事务是否正确的执行，因此遇到死锁的情况也会很多，然后事务不正确的情况也很多。
- 第二个时间点就是我们叫 DLV1，就是在事务基本上数据访问的差不多了，但是协调节点还不太清楚，就是说所有的事务节点是不是已经完全做完了不太清楚，但是所有的事务节点各自都认为它自己做完的时候，这个时候释放锁。
- 第三个时间点就是多加了一个协调的过程，协调的节点会跟所有的事务处理的节点做一个通信，通信完了之后，它认为这个时候所有节点都做完了，这个时候释放锁。
- 第四个时间点就是两阶段提交的第一个阶段结束之后，再释放锁，这个是最安全的。

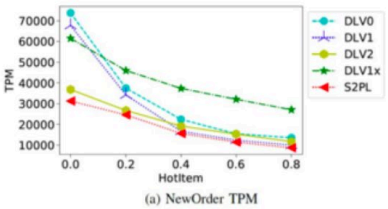


(a)

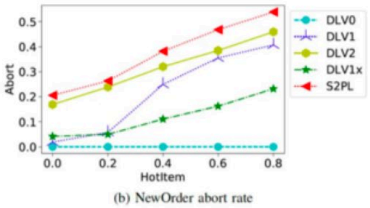


(b)

Fig. 12: Impact of Geo-distance on Performance



(a) NewOrder TPM



(b) NewOrder abort rate

Fig. 14: Impact of Hot Spot on Performance (TPC-C)

然后我们就实验了这些不同的加锁和释放锁的方式，得到的一个结论，左边这两张图（横轴是远程通信的时长）说明两个计算机中心离得越远，它通信时间越长，然后用这种传统的方式，会看到时长越长，它的性能就会越差。在高冲突的情况下，分布式事务处理性能就会比较差。但是如果使用提前释放锁的方式，性能就是绿色蓝色的线，表示着它的性能会有一个比较大的提升，这个就说明提前释放锁是有用的。

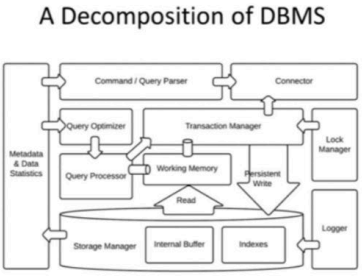
但什么时候提前释放锁最合适呢？右边这个图我们做的一个实验最后的结论是第三个时间点就是 DLV1x 这种方式，协调节点跟事务处理阶段有一个短通信，这个是本地通信，不是异地通信，通信之后确认所有节点都做完了，这个时候释放锁，这样的负面作用是最少的，而且它的加锁时间也会很短，这种方式它的效率是最高的。

2.2.2 数据库系统解耦合

1. 关系数据库的内核实现极具工程挑战

- 代码耦合度高（Monolithic）。
- 测试困难，特别对于分布式系统。
- 导致开发成本高，功能演进缓慢。

2. 为了实现 One Size Fits a Bunch



我刚才介绍了三项研究，都是关于分布式事务处理的，然后我接下来再讲一讲我们在数据库系统解耦上面的一些研究，这也是非常有趣的一个问题。什么叫数据库系统的解耦？实际上教科书会把数据库系统拆成一个一个的模块，比如说这个是 collector 就是跟应用对接的连接器，还有 Query Optimize、Query Evaluator 或者叫 Query Processor，就是查询处理查询优化的模块，New Storage Manager、New Transaction Manager，还有各种日志、Lock、Manager 等等，这个是我们教科书上面对数据库一个模块的切分。

但实际上当我们去真正的看一个数据库系统的实现，就会发现这些模块之间实际上没有切分的那么干净的，而且有时候是实际上模块之间很高耦合的，很紧的耦合在一起。对于一个刚开始做数据库的人会觉得跟我们学的东西会不完全一样。然后很少的人真的去探究为什么会是这样。我们在实现数据库系统的时候，实际上这种高耦合的系统架构给我们带来了很大的困扰。

我们当时在某银行改那个 OceanBase 的系统时，改动一个数据类型，我记得好像花了好几个月的时间，很多人去做这个事情。这实际上一听上去会让人比较诧异，但实际上你去看系统的实现，它就是这么回事。一个数据类型好多地方都会用到，必须把每一个地方都清除掉，这个时候就必须花很多时间去读代码去理解去测试的。其实我们觉得如果一个系统的耦合度能够变低，模块之间能够分得很清楚，实际上对系统工程来讲是有很大收益的。

我们回顾刚才讲的一个数据库的发展趋势，叫“One size fits a bunch”，也就是说我们认为以后系统会变得很定制化。如果一个系统的模块化做得很好的话，去定制去改动这个系统也会变得很简单。一旦一个新的硬件出现的时候，我们要去使用新的硬件去对这个系统进行优化，会变得更简单。

其实一个数据库系统的实现，是有需要去做进一步的解耦合，这里面其实有很多问题。我们去做了一些探索，但其实是非常有限的探索，我觉得这个工作其实可以有更多的人去做。我们做的探索，就是说想把并发控制直接从数据库的存储层抽离出来，然后让存储的代码跟并发控制的代码尽量的互不相关。

B-tree’ s Search Function in Textbook

```
search(root, key) {
  while !is_leaf(node) {          //寻找叶子结点
    child_node = get_child(node, key)
  return find(node, key)          //在叶子结点中寻找特定键值
}
```

这是一个 B-Tree 的 Search Function 的例子，在教科书里面，关于 B-Tree 的 Search，你可能会看到这样一个代码，非常简单。

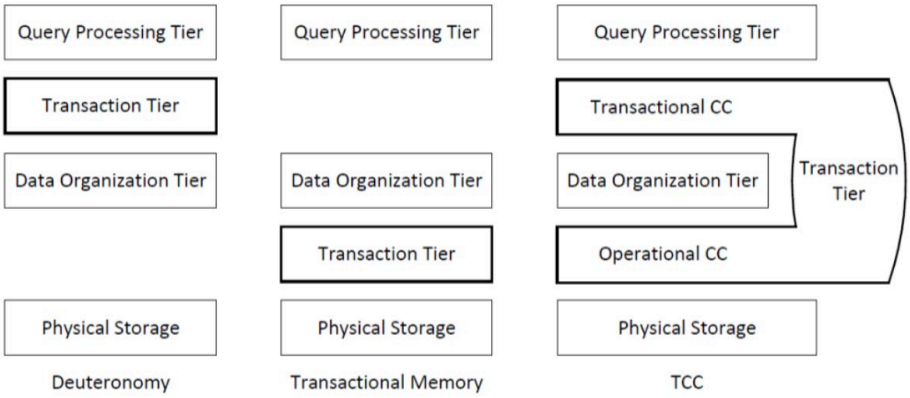
B-tree’ s Search Function in Real World:

```
search(root, key, value) {
  Lock(root, key)   //谓词锁，用来禁止幻象
  node = root
  Latch(node, S)                  //施加共享门栓
  while !is_leaf(node) {
    child_node = get_child(node, key)
    Latch(child_node, S)          //对叶子节点施加门闩
    UnLatch(node)                 //叶子节点获得门闩后
                                  //释放父节点门闩
    if FindSMO(child_node) {//检查结构修改标记,
                                  //防止有并发插入操作分裂页面
      Unlatch(child_node)         //如果有并发插入操作分裂页面,
                                  //等待分裂结束后重新查找
      GetSMOLatch()               //通过获得结构修改门闩等待分裂结束
      ReleaseSMOLatch()           //一旦获得结构修改门闩
                                  //说明分裂已经结束，这时立即释放
      retry
    }
    node = child_node
  }
  return find(node, key)
}
```

但实际上一个 B-Tree 的 Search 没有那么简单，可以看到这里面有好多的东西，这还只是一个例子。如果对开源系统比较熟悉的话，一般的一个开源系统的 B-Tree，差不多要将近十万行代码，非常复杂。

这个代码为什么这么复杂？可以看到 B-Tree 里面有很多的锁，有比如 Latch、Lock 之类的很多东西，它实际上是在做并发控制。当然并发控制只是导致代码复杂的原因之一，但还有其他的原因，并发控制把这个代码变得远远的复杂于 B-tree 本

身功能的程度。其实这就是数据库解耦合的一个动机，如果可以把耦合度解开，并发控制可以交给一个单独的模块去做，B-Tree 的代码就可以像第一个例子那样写，事情就变得很简单。

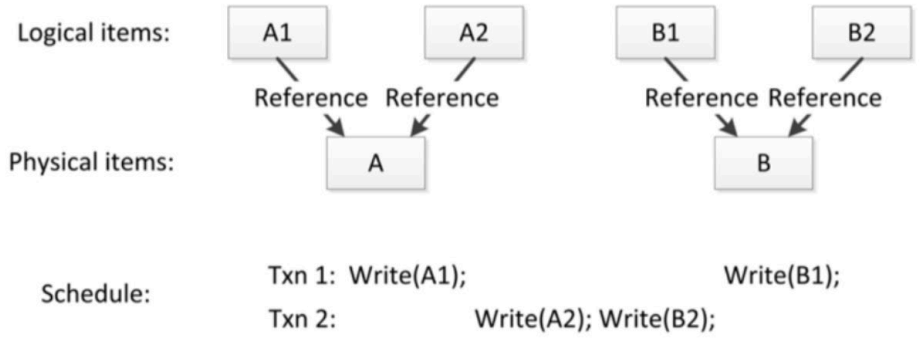


于是，我们就想如果把 CC 就是并发控制从数据库这种体系里面解耦出来应该怎么做？有很多种方式，最左边的这种其实是比较传统的方式，这种方式实际上并没有让数据库存储层变得更简单，只是在存储地上面做了一个事务处理层，这是一个比较浅的做法。中间的这种就是一个很暴力的做法，它是在物理的存储上面加一个 Transaction Tier，然后在上面做存储做运算。

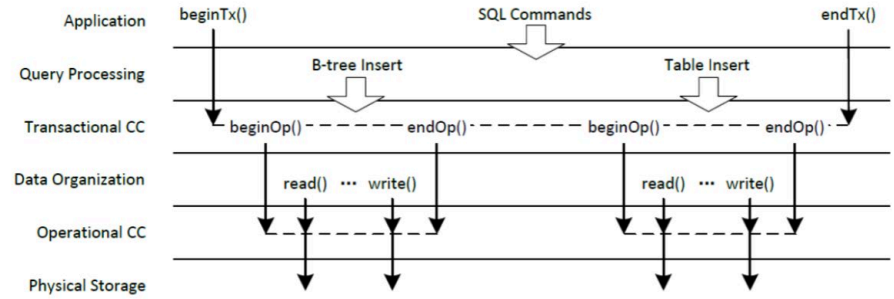
大家应该听说过 Transactional Memory，就是事务内存，这种就是直接用事务内存做事务处理，是一种很暴力的方式。最右边是我们提出来的方式，把并发控制的层次分成了两层，一种是我们叫操作层的并发控制，一种是事务层的并发控制，把它们合在一起变成一个新的模块。实践下来肯定是我们这种方式效果明显地更好，这种很暴力的事务内存的方式，实际上性能是不可以接受的。

我们其实看到现在有一些做存储的同学，他有一些比较天真的想法，他认为把事务做到存储的最底层，然后上面就不需要关心事务了，实际上那是不行的。事务是跟系统的功能是有很多耦合的因子在里面的，不能完全把它抛弃掉。然后最左边这种方式的结果是不彻底的，实现 B-tree 的时候还是会挺复杂的。然后我们提出的这种方式，

可以清楚的把 CC 给抽离出来。



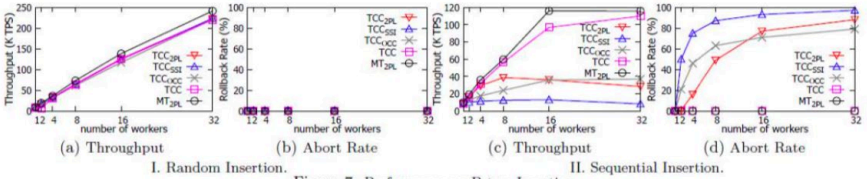
这个事情其实并不是那么的简单，上面是一个很简单的例子，A 和 B 是两个物理数据，然后在数据库的数据结构里面，定义两个引用（Reference），A1 和 A2，B1 和 B2，A1 和 A2 都是指向 A 的，B1 和 B2 都是指向 B 的。上面的事务层实际上是对 A1、A2、B1、B2 进行访问的。这个时候如果只是通过逻辑层去决定事物跟事物是否冲突的话，是会出错的。因为这里的逻辑层跟物理层，有一个重复引用的关系，可以看到下面这个事务处理的 Schedule，从 A1、A2、B1、B2 这种方式去看，好像这两个事务这样处理是没问题的，它是有这种可串行化的能力的，但实际上并没有，因为 A1 和 A2 指向的是同样一个数据。



这种实际上就是说如果真的要去把这个事务抽离出来的时候，会有很多的问题需要去解决，我这里没有办法深入地探讨，总之我们做了这样一个尝试，我们叫 Transpar-

ent Concurrency Control（TCC），就是透明的并发控制的一种模式。

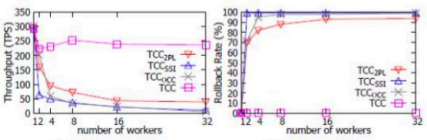
我们把这个事务层抽成两层，一个是事务层次的并发控制，一个是操作层次的并发控制，让这两种东西能够配合起来使用。然后用户去编写程序的时候，他可以不要去关心操作层面的并发控制，他直接去写他的 B-Tree 就行了。但是写好 B-Tree 之后，在事务层次上的这种并发控制，需要提供一些语义的信息说明哪种操作跟哪种操作之间实际上是不会冲突的，这样整个事务处理过程就可以保证正确性和高效性。



(a) Throughput (b) Abort Rate (c) Throughput (d) Abort Rate

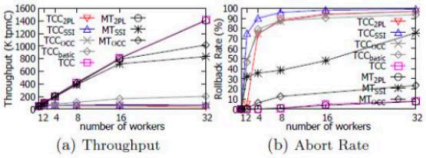
I. Random Insertion. II. Sequential Insertion.

Figure 7: Performance on B-tree Insertion.



(a) Throughput (b) Abort Rate

Figure 8: Performance on a Corner Case.

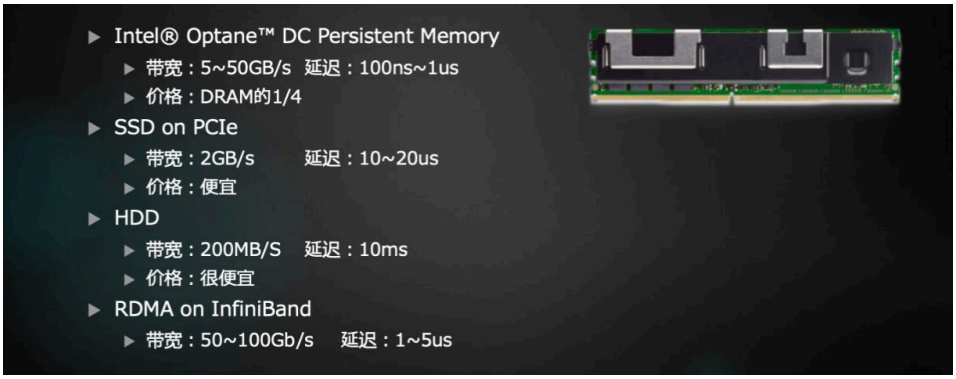


(a) Throughput (b) Abort Rate

Figure 9: Performance on Revised New-Order Transactions.

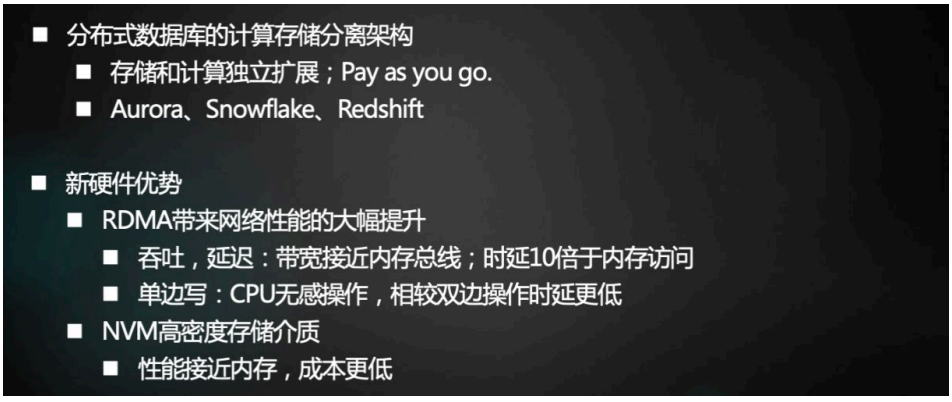
我们做了一些实验，去验证这样的一个解耦。右边的这些图，这种圆圈的线代表原始数据库实现的性能，可以看到我们的方式（TCC）的性能在很多情况下可以接近原始数据库的性能，这是解耦之后数据库的表现。很多时候解耦之后的表现可以接近原始数据库的性能，所以我们觉得这种解耦实际上还是可行的。但如果真的要把它用到一个现实的系统当中，其实并没有那么简单。这是我介绍的第二个研究工作，就是我们在数据库解耦上面的一些有趣的发现。

2.2.3 新硬件

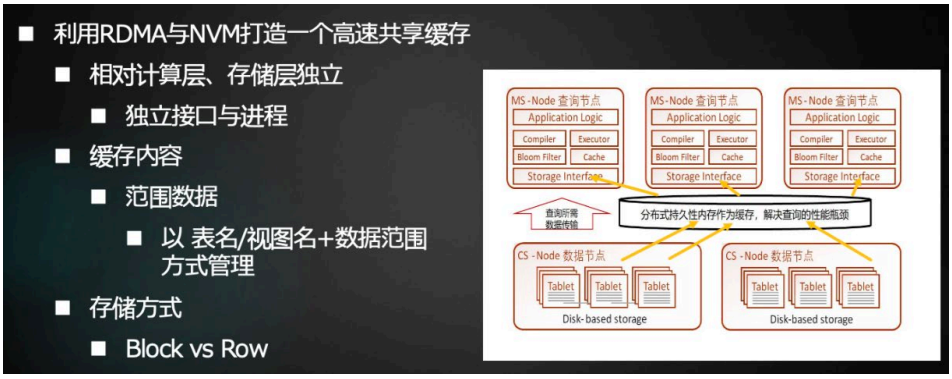


最后我再谈一谈新硬件，就是这种非易失内存。英特尔的傲腾是现在市面上唯一的一个真正的非易失内存产品，图中是产品的相关指标。对于存储器件的话，我们一般看两个指标，一个是带宽，一个是延迟。

可以看到它的带宽和延迟都是远远超过 SSD，当然更加超过这种硬盘。它的价格会比 SSD 和硬盘要昂贵不少，但相对于内存而言，它还是便宜的，我们可以预测它后面会越来越便宜，它跟 SSD 之间的一个价格的差异会变得越来越小，所以以后它有可能会取代 SSD 这种固态硬盘，但是不会太早。这种新的存储，它的性能更好，又比内存的造价低，所以它以后在系统当中肯定是有很重要的一个位置的。现在有了这种硬件之后，我们需要讨论在数据库系统里面，这个硬件到底起什么作用，一个新的数据库的架构应该怎么去使用它？怎么定位它的价值和位置？



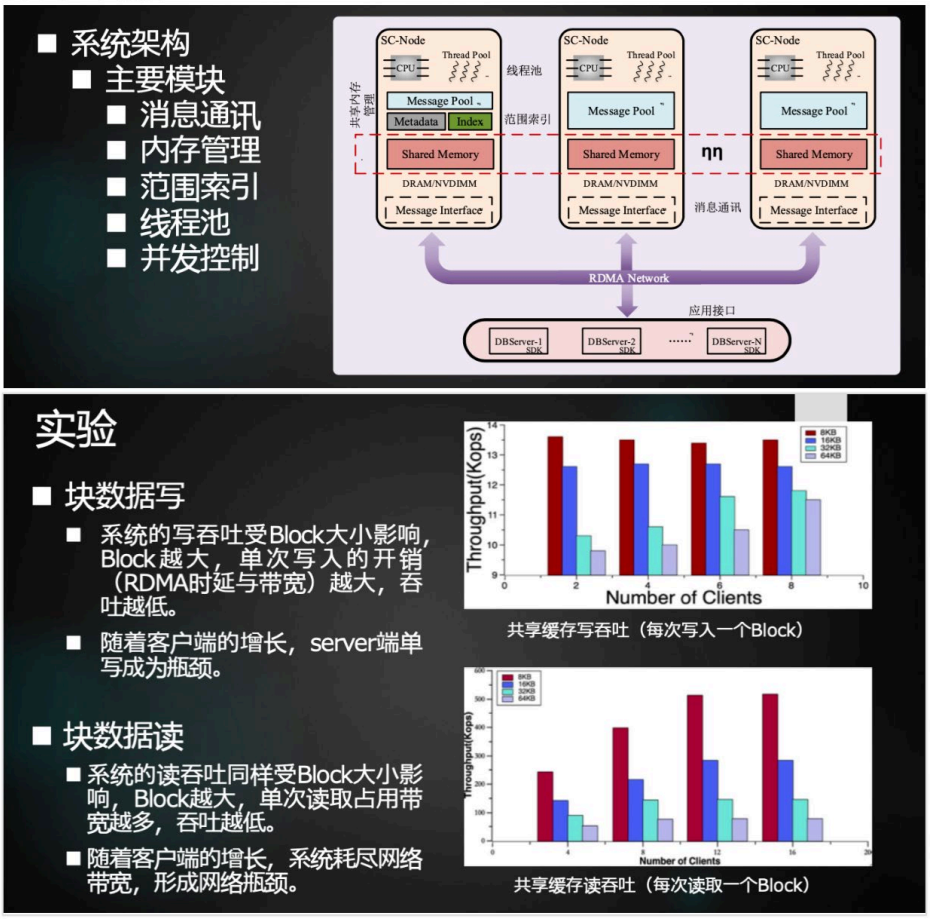
我们团队对这个东西讨论了很长的时间，最后有一个这样的设计，首先非易失内存这个东西，当然要用到数据库里面，数据库有各种形态的数据库，不同的形态的数据库使用方式是不一样的，但我们最后把它定位在云的数据库上面，因为我们知道云是未来的最主要的架构。在云的数据库上面怎么去用这个东西，我们觉得它跟 RDMA 的使用应该结合在一起。



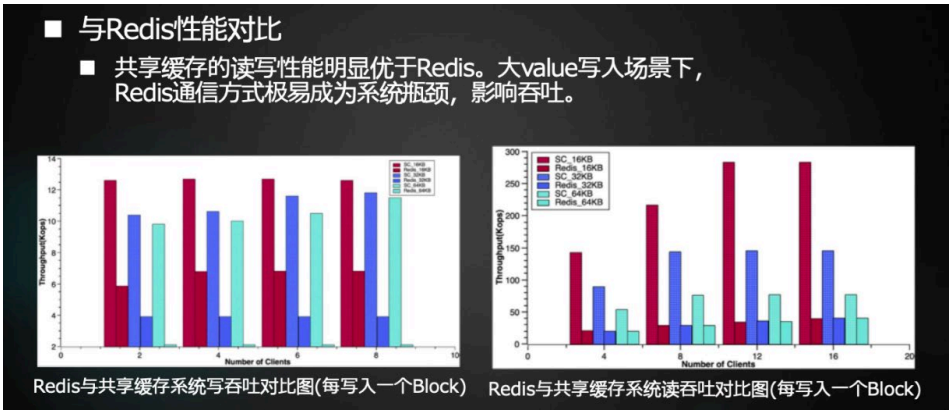
现在的云数据库变成一种计算节点跟存储节点是相对分开的架构方式。然后一旦 NVM 加进来之后，我们希望它成为计算节点跟存储节点之间的一个缓存。我们觉得现在暂时不能用它来做全量数据的存储，因为它的价格实在是比较昂贵，很多冷的数据，完全没有必要存在这样昂贵的存储里面，所以它作为一种缓存，比较合适的。

另外一方面，它作为一个缓存，不应该是一个割裂的节点，因为我们去看它的性能指标，可以看到实际上这种非易失内存的吞吐、延迟，和在高速网络上的 RDMA 的吞吐和延迟是比较接近的。如果比较接近的话，这个器件是通过 RDMA 的远程去访问，还是通过本地访问的速度差异有可能并不会很大。如果这个速度的差异并不会很大的话，我们实际上是可以把多个节点的 NVM 联合在一起，作为共享的缓存，缓存共享有非常多的优势，省去了很多缓存数据同步的代价，然后还可以让系统的负载均衡变得更好。

我们决定去设计这样一个系统架构。这个是 NVM，我们叫存储节点和计算节点之间的一个缓存。



我们这个事情正在做的过程中，所以目前为止我们是实现了一个分布式的缓存，可以作为共享缓存来用。我们开始讨论它应该是作为数据库而言，是行级别的缓存还是块级别的缓存，最后我们的选择是块级别的缓存，主要的原因还是因为实现起来更简单。我们先试一试，如果做到行级别的缓存的话，是有很多的工作量的，我们后期可能还会去尝试。



然后缓存的基本的测试，我们觉得我们的实现基本已经到位了，就是它的带宽的瓶颈基本上压到了 RDMA 访问带宽的瓶颈，如果要对它进行读写的话，它的瓶颈基本上就是 RDMA 远程访问的瓶颈，然后它的性能是远远高于像 Redis 这样的一个系统的，我们希望用这个缓存把它放在数据库里面，去提升这种云数据库的一个性能，但这个过程我们还在实现当中，我们有一些初步的结果，它是有一些效果的，特别是面对底层是 SSD 或者磁盘这样的系统的时候。我们希望后面有更明确结果的时候，再给大家介绍。

03 相关论文列表

Jinwei Guo, et al.: Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. PVLDB 2019

Huan Zhou, et al.: Plover: Parallel Logging for Replication Systems. FCS 2019

Ningnan Zhou, et al.: Transparent Concurrency Control: Decoupling Concurrency Control from DBMS. arXiv 2019

Donghui Wang, et al.: Fast Quorum-Based Log Replication and Replay for Fast Databases. DASFAA (1) 2019: 209-226

Tao Zhu, et al.: Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage. USENIX ATC 2018

Jiahao Wang, et al.: Range Optimistic Concurrency Control for a Composite OLTP and Bulk Processing Workload. ICDE 2018

Jinwei Guo, et al.: Efficient Snapshot Isolation in Paxos-Replicated Database Systems. DASFAA 2018

Xuan Zhou, et al.: Posterior Snapshot Isolation. ICDE 2017

这是我们实验室的一些代表性论文，不是很全，我刚才讲的部分技术并不在列，因为

还没有公开发表。如果感兴趣的话，大家可以阅读一下。

写在后面

华东师范大学周烜教授也是 2020–2021 年度美团科研课题合作学者。当前美团技术团队与超过 30 位来自国内外高校和科研院所的学者建立了科研课题合作。美团科研合作计划，基于美团在生活服务领域全场景里提炼出的科研命题，面向学术界征集前沿解决方案。

我们致力于与学术界“一起解决真实世界的问题”，愿与学术界共同推动产学研成果落地。2021 年将更加精彩纷呈，敬请期待。


CODE A BETTER LIFE
一行代码 亿万生活
美团
技术团队
长按二维码关注我们