

怎样才能做好性能调优？



先给你讲个故事吧。多年前我加入了一家大型互联网公司，刚进入就以 996 标准，参与新品研发。公司业务发展急需互联网产品，因此我们的时间很紧张，4 个月新产品就上线了。

开始还算顺利，但不久后的一天晚上，系统突然就瘫痪了，重启之后，问题仍然无规律地重现。当时运维同事马上写了一个重启脚本，定时排队重启各个服务，但也只能做到“治标不治本”。

作为主力开发，我和公司的系统架构师一起排查问题。架构师轻车熟路地通过各种 Linux 命令在线上环境查看性能指标，也 dump 出日志文件，走查代码，最后定位到了问题，后

面就是分析原因、制定解决方案、更新版本等一系列操作。那是我第一次深刻感受到性能调优的重要性。

后来的几年里，我又陆续参与过物流、电商、游戏支付系统的研发，这些项目都存在一个共性，就是经常会运营一些大促以及抢购类活动。活动期间，系统不仅要保证处理请求业务的严谨性，还要历经短时间内高并发的考验。我也一直处于性能调优的一线。

正所谓“实践出真知”。这些年在生产环境中遇到的事故不少，很多坑一点点踩平，就走出了一条路，这个过程中我收获了很多实打实的经验，希望能分享给更多的人，我们一起学习、交流和探讨。

关于性能调优，我先来说说我的感受。Java 性能调优不像是学一门编程语言，无法通过直线式的思维来掌握和应用，它对于工程师的技术广度和深度都有着较高的要求。

互联网时代，一个简单的系统就囊括了应用程序、数据库、容器、操作系统、网络等技术，线上一旦出现性能问题，就可能要你协调多方面组件去进行优化，这就是技术广度；而很多性能问题呢，又隐藏得很深，可能因为一个小小的代码，也可能因为线程池的类型选择错误...可归根结底考验的还是我们对这项技术的了解程度，这就是技术深度。

显然，性能调优不是一件容易的事。但有没有什么方法能把这件事情做好呢？接下来跟你分享几点我的心得。

1. 扎实的计算机基础

我们调优的对象不是单一的应用服务，而是错综复杂的系统。应用服务的性能可能与操作系统、网络、数据库等组件相关，所以我们需要储备计算机组成原理、操作系统、网络协议以及数据库等基础知识。具体的性能问题往往还与传输、计算、存储数据等相关，那我们还需要储备数据结构、算法以及数学等基础知识。

2. 习惯透过源码了解技术本质

我身边有很多好学的同学，他们经常和我分享在一些技术论坛或者公众号上学到的技术。这个方式很好，因为论坛上边的大部分内容，都是生产者自己吸收消化后总结的知识点，能帮助我们快速获取、快速理解。但是只做到这个程度还不够，因为你缺失了自己的判断。怎么办呢？我们需要深入源码，通过分析来学习、总结一项技术的实现原理和优缺点，这样我们

就能更客观地去学习一项技术，还能透过源码来学习牛人的思维方式，收获更好的编码实现方式。

3. 善于追问和总结

很多同学在使用一项技术时，只是因为这项技术好用就用了，从来不问自己：为什么这项技术可以提升系统性能？对比其他技术它好在哪儿？实现的原理又是什么呢？事实上，“知其然且知所以然”才是我们积累经验的关键。知道了一项技术背后的实现原理，我们才能在遇到性能问题时，做到触类旁通。

在这个专栏里，我将从实战出发，精选高频性能问题，透过 Java 底层源码，提炼出优化思路和它背后的实现原理，最后形成一套“学完就能用的调优方法论”。这也是很多一线大厂对于高级工程师的要求，希望通过这个专栏帮助你快速进阶。

那这个专栏具体是怎么设计的呢？结合 Java 应用开发的知识点，我将内容分为七大模块，从上到下依次详解 Java 应用服务的每一层优化实战。

模块一，概述。为你建立两个标准。一个是性能调优标准，告诉你可以通过哪些参数去衡量系统性能；另一个是调优过程标准，带你了解通过哪些严格的调优策略，我们可以排查性能问题，从而解决问题。

模块二，Java 编程性能调优。JDK 是 Java 语言的基础库，熟悉 JDK 中各个包中的工具类，可以帮助你编写出高性能代码。这里我会从基础的数据类型讲起，涉及容器在实际应用场景中的调优，还有现在互联网系统架构中比较重要的网络通信调优。

模块三，多线程性能调优。目前大部分服务器都是多核处理器，多线程编程的应用广泛。为了保证线程的安全性，通常会用到同步锁，这会为系统埋下很多隐患；除此之外，还有多线程高并发带来的性能问题，这些都会在这个模块重点讲解。

模块四，JVM 性能监测及调优。Java 应用程序是运行在 JVM 之上的，对 JVM 进行调优可以提升系统性能。这里重点讲解 Java 对象的创建和回收、内存分配等。

模块五，设计模式调优。在架构设计中，我们经常会用到一些设计模式来优化架构设计。这里我将结合一些复杂的应用场景，分享设计优化案例。

模块六，数据库性能调优。数据库最容易成为整个系统的性能瓶颈，这里我会重点解析一些数据库的常用调优方法。

模块七，实战演练场。以上六个模块的内容，都是基于某个点的调优，现在是时候把你前面所学都调动起来了，这里我将带你进入综合性能问题高频出现的应用场景，学习整体调优方法。

01 | 如何制定性能调优标准?



我有一个朋友，有一次他跟我说，他们公司的系统从来没有经过性能调优，功能测试完成后就上线了，线上也没有出现过什么性能问题呀，那为什么很多系统都要去做性能调优呢？

当时我就回答了他一句，如果你们公司做的是 12306 网站，不做系统性能优化就上线，试试看会是什么情况。

如果是你，你会怎么回答呢？今天，我们就从这个话题聊起，希望能跟你一起弄明白这几个问题：我们为什么要性能调优？什么时候开始做？做性能调优是不是有标准可参考？

为什么要性能调优？

一款线上产品如果没有经过性能测试，那它就好比是一颗定时炸弹，你不知道它什么时候会出现问题，你也不清楚它能承受的极限在哪儿。

有些性能问题是时间累积慢慢产生的，到了一定时间自然就爆炸了；而更多的性能问题是由访问量的波动导致的，例如，活动或者公司产品用户量上升；当然也有可能是一款产品上线后就半死不活，一直没有大访问量，所以还没有引发这颗定时炸弹。

现在假设你的系统要做一次活动，产品经理或者老板告诉你预计有几十万的用户访问量，询问系统能否承受得住这次活动的压力。如果你不清楚自己系统的性能情况，也只能战战兢兢地回答老板，有可能大概没问题吧。

所以，要不要做性能调优，这个问题其实很好回答。所有的系统在开发完之后，多多少少都会有性能问题，我们首先要做的就是想办法把问题暴露出来，例如进行压力测试、模拟可能的操作场景等等，再通过性能调优去解决这些问题。

比如，当你在用某一款 App 查询某一条信息时，需要等待十几秒钟；在抢购活动中，无法进入活动页面等等。你看，系统响应就是体现系统性能最直接的一个参考因素。

那如果系统在线上没有出现响应问题，我们是不是就不用去做性能优化了呢？再给你讲一个故事吧。

曾经我的前前东家系统研发部门来了一位大神，为什么叫他大神，因为在他来公司的一年时间里，他只做了一件事情，就是把服务器的数量缩减到了原来的一半，系统的性能指标，反而还提升了。

好的系统性能调优不仅仅可以提高系统的性能，还能为公司节省资源。这也是我们做性能调优的最直接的目的。

什么时候开始介入调优？

解决了为什么要做性能优化的问题，那么新的问题就来了：如果需要对系统做一次全面的性能监测和优化，我们从什么时候开始介入性能调优呢？是不是越早介入越好？

其实，在项目开发的初期，我们没有必要过于在意性能优化，这样反而会让我们疲于性能优化，不仅不会给系统性能带来提升，还会影响到开发进度，甚至获得相反的效果，给系统带来新的问题。

我们只需要在代码层面保证有效的编码，比如，减少磁盘 I/O 操作、降低竞争锁的使用以及使用高效的算法等等。遇到比较复杂的业务，我们可以充分利用设计模式来优化业务代码。例如，设计商品价格的时候，往往会有许多折扣活动、红包活动，我们可以用装饰模式去设计这个业务。

在系统编码完成之后，我们就可以对系统进行性能测试了。这时候，产品经理一般会提供线上预期数据，我们在提供的参考平台上进行压测，通过性能分析、统计工具来统计各项性能指标，看是否在预期范围之内。

在项目成功上线后，我们还需要根据线上的实际情况，依照日志监控以及性能统计日志，来观测系统性能问题，一旦发现问题，就要对日志进行分析并及时修复问题。

有哪些参考因素可以体现系统的性能？

上面我们讲到了在项目研发的各个阶段性能调优是如何介入的，其中多次讲到了性能指标，那么性能指标到底有哪些呢？

在我们了解性能指标之前，我们先来了解下哪些计算机资源会成为系统的性能瓶颈。

CPU：有的应用需要大量计算，他们会长时间、不间断地占用 CPU 资源，导致其他资源无法争夺到 CPU 而响应缓慢，从而带来系统性能问题。例如，代码递归导致的无限循环，正则表达式引起的回溯，JVM 频繁的 FULL GC，以及多线程编程造成的大量上下文切换等，这些都有可能导致 CPU 资源繁忙。

内存：Java 程序一般通过 JVM 对内存进行分配管理，主要是用 JVM 中的堆内存来存储 Java 创建的对象。系统堆内存的读写速度非常快，所以基本不存在读写性能瓶颈。但是由于内存成本要比磁盘高，相比磁盘，内存的存储空间又非常有限。所以当内存空间被占满，对象无法回收时，就会导致内存溢出、内存泄露等问题。

磁盘 I/O：磁盘相比内存来说，存储空间要大很多，但磁盘 I/O 读写的速度要比内存慢，虽然目前引入的 SSD 固态硬盘已经有所优化，但仍然无法与内存的读写速度相提并论。

网络：网络对于系统性能来说，也起着至关重要的作用。如果你购买过云服务，一定经历过，选择网络带宽大小这一环节。带宽过低的话，对于传输数据比较大，或者是并发量比较大的系统，网络就很容易成为性能瓶颈。

异常: Java 应用中，抛出异常需要构建异常栈，对异常进行捕获和处理，这个过程非常消耗系统性能。如果在高并发的情况下引发异常，持续地进行异常处理，那么系统的性能就会明显地受到影响。

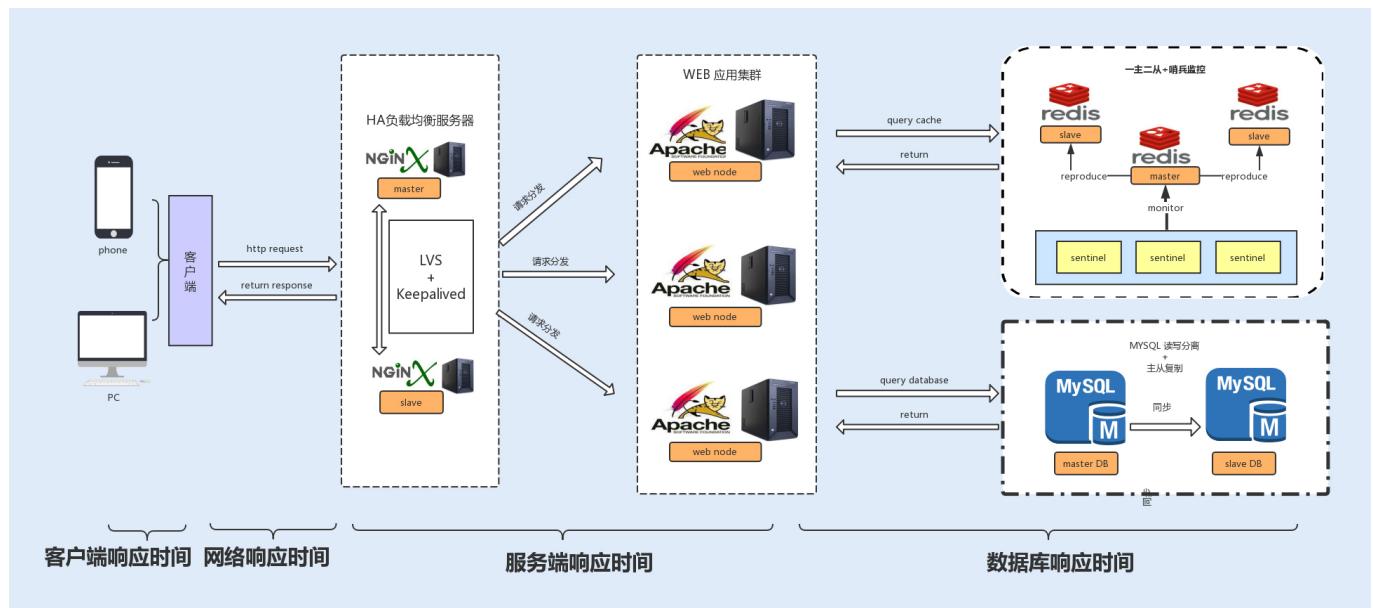
数据库: 大部分系统都会用到数据库，而数据库的操作往往是涉及到磁盘 I/O 的读写。大量的数据库读写操作，会导致磁盘 I/O 性能瓶颈，进而导致数据库操作的延迟性。对于有大量数据库读写操作的系统来说，数据库的性能优化是整个系统的核心。

锁竞争: 在并发编程中，我们经常会需要多个线程，共享读写操作同一个资源，这个时候为了保持数据的原子性（即保证这个共享资源在一个线程写的时候，不被另一个线程修改），我们就会用到锁。锁的使用可能会带来上下文切换，从而给系统带来性能开销。JDK1.6 之后，Java 为了降低锁竞争带来的上下文切换，对 JVM 内部锁已经做了多次优化，例如，新增了偏向锁、自旋锁、轻量级锁、锁粗化、锁消除等。而如何合理地使用锁资源，优化锁资源，就需要你了解更多的操作系统知识、Java 多线程编程基础，积累项目经验，并结合实际场景去处理相关问题。

了解了上面这些基本内容，我们可以得到下面几个指标，来衡量一般系统的性能。

响应时间

响应时间是衡量系统性能的重要指标之一，响应时间越短，性能越好，一般一个接口的响应时间是在毫秒级。在系统中，我们可以把响应时间自下而上细分为以下几种：



数据库响应时间: 数据库操作所消耗的时间，往往是整个请求链中最耗时的；

服务端响应时间：服务端包括 Nginx 分发的请求所消耗的时间以及服务端程序执行所消耗的时间；

网络响应时间：这是网络传输时，网络硬件需要对传输的请求进行解析等操作所消耗的时间；

客户端响应时间：对于普通的 Web、App 客户端来说，消耗时间是可以忽略不计的，但如果你的客户端嵌入了大量的逻辑处理，消耗的时间就有可能变长，从而成为系统的瓶颈。

吞吐量

在测试中，我们往往会比较注重系统接口的 TPS（每秒事务处理量），因为 TPS 体现了接口的性能，TPS 越大，性能越好。在系统中，我们也可以把吞吐量自下而上地分为两种：磁盘吞吐量和网络吞吐量。

我们先来看**磁盘吞吐量**，磁盘性能有两个关键衡量指标。

一种是 IOPS (Input/Output Per Second)，即每秒的输入输出量（或读写次数），这种是指单位时间内系统能处理的 I/O 请求数量，I/O 请求通常为读或写数据操作请求，关注的是随机读写性能。适应于随机读写频繁的应用，如小文件存储（图片）、OLTP 数据库、邮件服务器。

另一种是数据吞吐量，这种是指单位时间内可以成功传输的数据量。对于大量顺序读写频繁的应用，传输大量连续数据，例如，电视台的视频编辑、视频点播 VOD (Video On Demand)，数据吞吐量则是关键衡量指标。

接下来看**网络吞吐量**，这个是指网络传输时没有帧丢失的情况下，设备能够接受的最大数据速率。网络吞吐量不仅仅跟带宽有关系，还跟 CPU 的处理能力、网卡、防火墙、外部接口以及 I/O 等紧密关联。而吞吐量的大小主要由网卡的处理能力、内部程序算法以及带宽大小决定。

计算机资源分配使用率

通常由 CPU 占用率、内存使用率、磁盘 I/O、网络 I/O 来表示资源使用率。这几个参数好比一个木桶，如果其中任何一块木板出现短板，任何一项分配不合理，对整个系统性能的影响都是毁灭性的。

负载承受能力

当系统压力上升时，你可以观察，系统响应时间的上升曲线是否平缓。这项指标能直观地反馈给你，系统所能承受的负载压力极限。例如，当你对系统进行压测时，系统的响应时间会随着系统并发数的增加而延长，直到系统无法处理这么多请求，抛出大量错误时，就到了极限。

总结

通过今天的学习，我们知道性能调优可以使系统稳定，用户体验更佳，甚至在比较大的系统中，还能帮公司节约资源。

但是在项目的开始阶段，我们没有必要过早地介入性能优化，只需在编码的时候保证其优秀、高效，以及良好的程序设计。

在完成项目后，我们就可以进行系统测试了，我们可以将以下性能指标，作为性能调优的标准，响应时间、吞吐量、计算机资源分配使用率、负载承受能力。

回顾我自己的项目经验，有电商系统、支付系统以及游戏充值计费系统，用户级都是千万级别，且要承受各种大型抢购活动，所以我对系统的性能要求非常苛刻。除了通过观察以上指标来确定系统性能的好坏，还需要在更新迭代中，充分保障系统的稳定性。

这里，**给你延伸一个方法**，就是将迭代之前版本的系统性能指标作为参考标准，通过自动化性能测试，校验迭代发版之后的系统性能是否出现异常，这里就不仅仅是比较吞吐量、响应时间、负载能力等直接指标了，还需要比较系统资源的 CPU 占用率、内存使用率、磁盘 I/O、网络 I/O 等几项间接指标的变化。

02 | 如何制定性能调优策略?



上一讲，我在介绍性能调优重要性的时候，提到了性能测试。面对日渐复杂的系统，制定合理的性能测试，可以提前发现性能瓶颈，然后有针对性地制定调优策略。**总结一下就是“测试 - 分析 - 调优”三步走。**

今天，我们就在这个基础上，好好聊一聊“如何制定系统的性能调优策略”。

性能测试攻略

性能测试是提前发现性能瓶颈，保障系统性能稳定的必要措施。下面我先给你介绍两种常用的测试方法，帮助你从点到面地测试系统性能。

1. 微基准性能测试

微基准性能测试可以精准定位到某个模块或者某个方法的性能问题，特别适合做一个功能模块或者一个方法在不同实现方式下的性能对比。例如，对比一个方法使用同步实现和非同步实现的性能。

2. 宏基准性能测试

宏基准性能测试是一个综合测试，需要考虑到测试环境、测试场景和测试目标。

首先看测试环境，我们需要模拟线上的真实环境。

然后看测试场景。我们需要确定在测试某个接口时，是否有其他业务接口同时也在平行运行，造成干扰。如果有，请重视，因为你一旦忽视了这种干扰，测试结果就会出现偏差。

最后看测试目标。我们的性能测试是要有目标的，这里可以通过吞吐量以及响应时间来衡量系统是否达标。不达标，就进行优化；达标，就继续加大测试的并发数，探底接口的TPS（最大每秒事务处理量），这样做，可以深入了解接口的性能。除了测试接口的吞吐量和响应时间以外，我们还需要循环测试可能导致性能问题的接口，观察各个服务器的CPU、内存以及I/O使用率的变化。

以上就是两种测试方法的详解。其中值得注意的是，性能测试存在干扰因子，会使测试结果不准确。所以，**我们在做性能测试时，还要注意一些问题。**

1. 热身问题

当我们做性能测试时，我们的系统会运行得越来越快，后面的访问速度要比我们第一次访问的速度快上几倍。这是怎么回事呢？

在Java编程语言和环境中，.java文件编译成为.class文件后，机器还是无法直接运行.class文件中的字节码，需要通过解释器将字节码转换成本地机器码才能运行。为了节约内存和执行效率，代码最初被执行时，解释器会率先解释执行这段代码。

随着代码被执行的次数增多，当虚拟机发现某个方法或代码块运行得特别频繁时，就会把这些代码认定为热点代码（Hot Spot Code）。为了提高热点代码的执行效率，在运行时，虚拟机将会通过即时编译器（JIT compiler, just-in-time compiler）把这些代码编译成与

本地平台相关的机器码，并进行各层次的优化，然后存储在内存中，之后每次运行代码时，直接从内存中获取即可。

所以在刚开始运行的阶段，虚拟机会花费很长的时间来全面优化代码，后面就能以最高性能执行了。

这就是热身过程，如果在进行性能测试时，热身时间过长，就会导致第一次访问速度过慢，你就可以考虑先优化，再进行测试。

2. 性能测试结果不稳定

我们在做性能测试时发现，每次测试处理的数据集都是一样的，但测试结果却有差异。这是因为测试时，伴随着很多不稳定因素，比如机器其他进程的影响、网络波动以及每个阶段 JVM 垃圾回收的不同等等。

我们可以通过多次测试，将测试结果求平均，或者统计一个曲线图，只要保证我们的平均值是在合理范围之内，而且波动不是很大，这种情况下，性能测试就是通过的。

3. 多 JVM 情况下的影响

如果我们的服务器有多个 Java 应用服务，部署在不同的 Tomcat 下，这就意味着我们的服务器会有多个 JVM。任意一个 JVM 都拥有整个系统的资源使用权。如果一台机器上只部署单独的一个 JVM，在做性能测试时，测试结果很好，或者你调优的效果很好，但在一台机器多个 JVM 的情况下就不一定了。所以我们应该尽量避免线上环境中一台机器部署多个 JVM 的情况。

合理分析结果，制定调优策略

这里我将“三步走”中的分析和调优结合在一起讲。

我们在完成性能测试之后，需要输出一份性能测试报告，帮我们分析系统性能测试的情况。其中测试结果需要包含测试接口的平均、最大和最小吞吐量，响应时间，服务器的 CPU、内存、I/O、网络 IO 使用率，JVM 的 GC 频率等。

通过观察这些调优标准，可以发现性能瓶颈，我们再通过自下而上的方式分析查找问题。首先从操作系统层面，查看系统的 CPU、内存、I/O、网络的使用率是否存在异常，再通过命令查找异常日志，最后通过分析日志，找到导致瓶颈的原因；还可以从 Java 应用的 JVM

层面，查看 JVM 的垃圾回收频率以及内存分配情况是否存在异常，分析日志，找到导致瓶颈的原因。

如果系统和 JVM 层面都没有出现异常情况，我们可以查看应用服务业务层是否存在性能瓶颈，例如 Java 编程的问题、读写数据瓶颈等等。

分析查找问题是一个复杂而又细致的过程，某个性能问题可能是一个原因导致的，也可能是几个原因共同导致的结果。我们分析查找问题可以采用自下而上的方式，而我们解决系统性能问题，则可以采用自上而下的方式逐级优化。下面我来介绍下从应用层到操作系统层的几种调优策略。

1. 优化代码

应用层的问题代码往往因为耗尽系统资源而暴露出来。例如，我们某段代码导致内存溢出，往往是将 JVM 中的内存用完了，这个时候系统的内存资源消耗殆尽了，同时也会引发 JVM 频繁地发生垃圾回收，导致 CPU 100% 以上居高不下，这个时候又消耗了系统的 CPU 资源。

还有一些是非问题代码导致的性能问题，这种往往是比较难发现的，需要依靠我们的经验来优化。例如，我们经常使用的 `LinkedList` 集合，如果使用 `for` 循环遍历该容器，将大大降低读的效率，但这种效率的降低很难导致系统性能参数异常。

这时有经验的同学，就会改用 `Iterator`（迭代器）迭代循环该集合，这是因为 `LinkedList` 是链表实现的，如果使用 `for` 循环获取元素，在每次循环获取元素时，都会去遍历一次 `List`，这样会降低读的效率。

2. 优化设计

面向对象有很多设计模式，可以帮助我们优化业务层以及中间件层的代码设计。优化后，不仅可以精简代码，还能提高整体性能。例如，单例模式在频繁调用创建对象的场景中，可以共享一个创建对象，这样可以减少频繁地创建和销毁对象所带来的性能消耗。

3. 优化算法

好的算法可以帮助我们大大地提升系统性能。例如，在不同的场景中，使用合适的查找算法可以降低时间复杂度。

4. 时间换空间

有时候系统对查询时的速度并没有很高的要求，反而对存储空间要求苛刻，这个时候我们可以考虑用时间来换取空间。

例如，我在 03 讲就会详解的用 String 对象的 intern 方法，可以将重复率比较高的数据集存储在常量池，重复使用一个相同的对象，这样可以大大节省内存存储空间。但由于常量池使用的是 HashMap 数据结构类型，如果我们存储数据过多，查询的性能就会下降。所以在这种对存储容量要求比较苛刻，而对查询速度不作要求的场景，我们就可以考虑用时间换空间。

5. 空间换时间

这种方法是使用存储空间来提升访问速度。现在很多系统都是使用的 MySQL 数据库，较为常见的分表分库是典型的使用空间换时间的案例。

因为 MySQL 单表在存储千万数据以上时，读写性能会明显下降，这个时候我们需要将表数据通过某个字段 Hash 值或者其他方式分拆，系统查询数据时，会根据条件的 Hash 值判断找到对应的表，因为表数据量减小了，查询性能也就提升了。

6. 参数调优

以上都是业务层代码的优化，除此之外，JVM、Web 容器以及操作系统的优化也是非常关键的。

根据自己的业务场景，合理地设置 JVM 的内存空间以及垃圾回收算法可以提升系统性能。例如，如果我们业务中会创建大量的大对象，我们可以通过设置，将这些大对象直接放进老年代。这样可以减少年轻代频繁发生小的垃圾回收（Minor GC），减少 CPU 占用时间，提升系统性能。

Web 容器线程池的设置以及 Linux 操作系统的内核参数设置不合理也有可能导致系统性能瓶颈，根据自己的业务场景优化这两部分，可以提升系统性能。

兜底策略，确保系统稳定性

上边讲到的所有的性能调优策略，都是提高系统性能的手段，但在互联网飞速发展的时代，产品的用户量是瞬息万变的，无论我们的系统优化得有多好，还是会存在承受极限，所以为

了保证系统的稳定性，我们还需要采用一些兜底策略。

什么是兜底策略？

第一，限流，对系统的入口设置最大访问限制。这里可以参考性能测试中探底接口的 TPS 。同时采取熔断措施，友好地返回没有成功的请求。

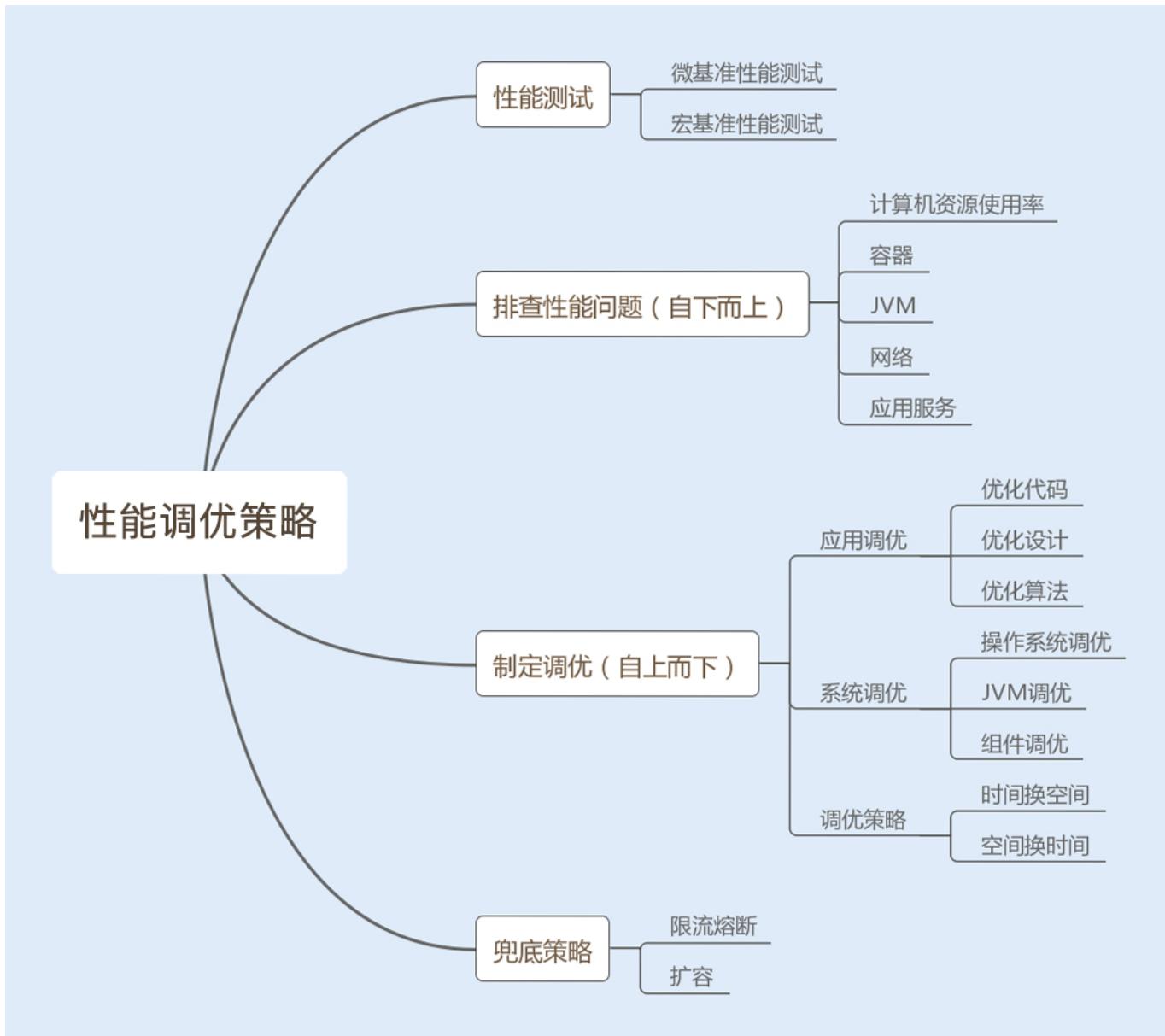
第二，实现智能化横向扩容。智能化横向扩容可以保证当访问量超过某一个阈值时，系统可以根据需求自动横向新增服务。

第三，提前扩容。这种方法通常应用于高并发系统，例如，瞬时抢购业务系统。这是因为横向扩容无法满足大量发生在瞬间的请求，即使成功了，抢购也结束了。

目前很多公司使用 Docker 容器来部署应用服务。这是因为 Docker 容器是使用 Kubernetes 作为容器管理系统，而 Kubernetes 可以实现智能化横向扩容和提前扩容 Docker 服务。

总结

学完这讲，你应该对性能测试以及性能调优有所认识了。我们再通过一张图来回顾下今天的内容。



我们将性能测试分为微基准性能测试和宏基准性能测试，前者可以精准地调优小单元的业务功能，后者可以结合内外因素，综合模拟线上环境来测试系统性能。两种方法结合，可以更立体地测试系统性能。

测试结果可以帮助我们制定性能调优策略，调优方法很多，这里就不一一赘述了。但有一个共同点就是，调优策略千变万化，但思路和核心都是一样的，都是从业务调优到编程调优，再到系统调优。

最后，给你提个醒，任何调优都需要结合场景明确已知问题和性能目标，不能为了调优而调优，以免引入新的 Bug，带来风险和弊端。

03 | 字符串性能优化不容小觑，百M内存轻松存储几十G数据



从第二个模块开始，我将带你学习 Java 编程的性能优化。今天我们就从最基础的 String 字符串优化讲起。

String 对象是我们使用最频繁的一个对象类型，但它的性能问题却是最容易被忽略的。String 对象作为 Java 语言中重要的数据类型，是内存中占据空间最大的一个对象。**高效地使用字符串，可以提升系统的整体性能。**

接下来我们就从 String 对象的实现、特性以及实际使用中的优化这三个方面入手，深入了解。

在开始之前，我想先问你一个小问题，也是我在招聘时，经常会问到面试者的一道题。虽是老生常谈了，但错误率依然很高，当然也有一些面试者答对了，但能解释清楚答案背后原理的人少之又少。问题如下：

通过三种不同的方式创建了三个对象，再依次两两匹配，每组被匹配的两个对象是否相等？
代码如下：

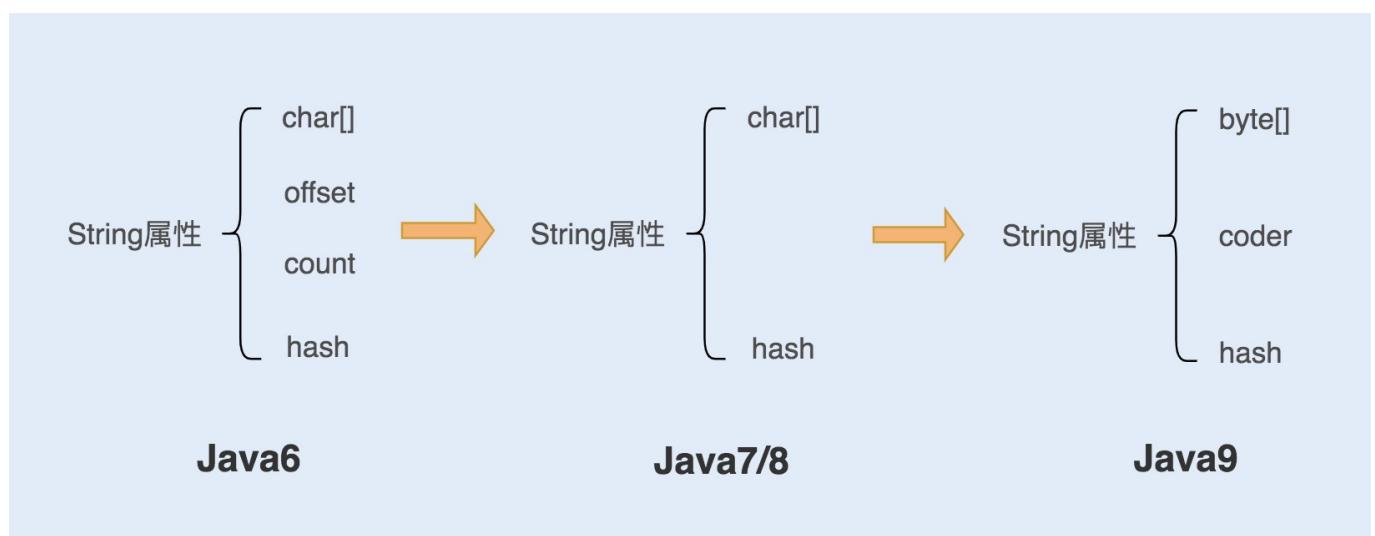
 复制代码

```
1 String str1= "abc";
2 String str2= new String("abc");
3 String str3= str2.intern();
4 assertEquals(str1==str2);
5 assertEquals(str2==str3);
6 assertEquals(str1==str3)
```

你可以先想想答案，以及这样回答的原因。希望通过今天的学习，你能拿到满分。

String 对象是如何实现的？

在 Java 语言中，Sun 公司的工程师们对 String 对象做了大量的优化，来节约内存空间，提升 String 对象在系统中的性能。一起来看看优化过程，如下图所示：



1. 在 Java6 以及之前的版本中，String 对象是对 char 数组进行了封装实现的对象，主要有四个成员变量：char 数组、偏移量 offset、字符数量 count、哈希值 hash。

String 对象是通过 offset 和 count 两个属性来定位 char[] 数组，获取字符串。这么做可以高效、快速地共享数组对象，同时节省内存空间，但这种方式很有可能会导致内存泄漏。

2. 从 Java7 版本开始到 Java8 版本，Java 对 String 类做了一些改变。String 类中不再有 offset 和 count 两个变量了。这样的好处是 String 对象占用的内存稍微少了些，同时，String.substring 方法也不再共享 char[]，从而解决了使用该方法可能导致的内存泄漏问题。

3. 从 Java9 版本开始，工程师将 char[] 字段改为了 byte[] 字段，又维护了一个新的属性 coder，它是一个编码格式的标识。

工程师为什么这样修改呢？

我们知道一个 char 字符占 16 位，2 个字节。这个情况下，存储单字节编码内的字符（占一个字节的字符）就显得非常浪费。JDK1.9 的 String 类为了节约内存空间，于是使用了占 8 位，1 个字节的 byte 数组来存放字符串。

而新属性 coder 的作用是，在计算字符串长度或者使用 indexOf () 函数时，我们需要根据这个字段，判断如何计算字符串长度。coder 属性默认有 0 和 1 两个值，0 代表 Latin-1（单字节编码），1 代表 UTF-16。如果 String 判断字符串只包含了 Latin-1，则 coder 属性值为 0，反之则为 1。

String 对象的不可变性

了解了 String 对象的实现后，你有没有发现在实现代码中 String 类被 final 关键字修饰了，而且变量 char 数组也被 final 修饰了。

我们知道类被 final 修饰代表该类不可继承，而 char[] 被 final+private 修饰，代表了 String 对象不可被更改。Java 实现的这个特性叫作 String 对象的不可变性，即 String 对象一旦创建成功，就不能再对它进行改变。

Java 这样做的好处在哪里呢？

第一，保证 String 对象的安全性。假设 String 对象是可变的，那么 String 对象将可能被恶意修改。

第二，保证 hash 属性值不会频繁变更，确保了唯一性，使得类似 HashMap 容器才能实现相应的 key-value 缓存功能。

第三，可以实现字符串常量池。在 Java 中，通常有两种创建字符串对象的方式，一种是通过字符串常量的方式创建，如 String str= "abc" ；另一种是字符串变量通过 new 形式创建，如 String str = new String("abc")。

当代码中使用第一种方式创建字符串对象时，JVM 首先会检查该对象是否在字符串常量池中，如果在，就返回该对象引用，否则新的字符串将在常量池中被创建。这种方式可以减少同一个值的字符串对象的重复创建，节约内存。

String str = new String("abc")这种方式，首先在编译类文件时，"abc"常量字符串将会放入到常量结构中，在类加载时，“abc”将会在常量池中创建；其次，在调用 new 时，JVM 命令将会调用 String 的构造函数，同时引用常量池中的"abc" 字符串，在堆内存中创建一个 String 对象；最后，str 将引用 String 对象。

这里附上一个你可能会想到的经典反例。

平常编程时，对一个 String 对象 str 赋值 “hello” ，然后又让 str 值为 “world” ，这个时候 str 的值变成了 “world” 。那么 str 值确实改变了，为什么我还说 String 对象不可变呢？

首先，我来解释下什么是对象和对象引用。Java 初学者往往对此存在误区，特别是一些从 PHP 转 Java 的同学。在 Java 中要比较两个对象是否相等，往往是用 ==，而要判断两个对象的值是否相等，则需要用 equals 方法来判断。

这是因为 str 只是 String 对象的引用，并不是对象本身。对象在内存中是一块内存地址，str 则是一个指向该内存地址的引用。所以在刚刚我们说的这个例子中，第一次赋值的时候，创建了一个 “hello” 对象，str 引用指向 “hello” 地址；第二次赋值的时候，又重新创建了一个对象 “world” ，str 引用指向了 “world” ，但 “hello” 对象依然存在于内存中。

也就是说 str 并不是对象，而只是一个对象引用。真正的对象依然还在内存中，没有被改变。

String 对象的优化

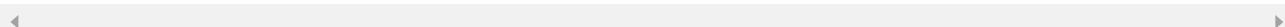
了解了 String 对象的实现原理和特性，接下来我们就结合实际场景，看看如何优化 String 对象的使用，优化的过程中又有哪些需要注意的地方。

1. 如何构建超大字符串？

编程过程中，字符串的拼接很常见。前面我讲过 String 对象是不可变的，如果我们使用 String 对象相加，拼接我们想要的字符串，是不是就会产生多个对象呢？例如以下代码：

 复制代码

```
1 String str= "ab" + "cd" + "ef";
```

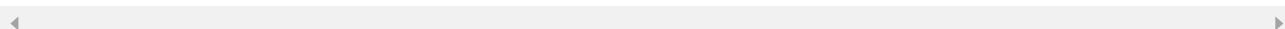


分析代码可知：首先会生成 ab 对象，再生成 abcd 对象，最后生成 abcdef 对象，从理论上来说，这段代码是低效的。

但实际运行中，我们发现只有一个对象生成，这是为什么呢？难道我们的理论判断错了？我们再来看编译后的代码，你会发现编译器自动优化了这行代码，如下：

 复制代码

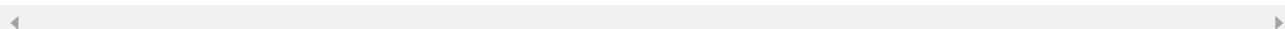
```
1 String str= "abcdef";
```



上面我介绍的是字符串常量的累计，我们再来看看字符串变量的累计又是怎样的呢？

 复制代码

```
1 String str = "abcdef";
2
3 for(int i=0; i<1000; i++) {
4     str = str + i;
5 }
```



上面的代码编译后，你可以看到编译器同样对这段代码进行了优化。不难发现，Java 在进行字符串的拼接时，偏向使用 StringBuilder，这样可以提高程序的效率。

 复制代码

```
1  
2 String str = "abcdef";  
3  
4 for(int i=0; i<1000; i++) {  
5     str = (new StringBuilder(String.valueOf(str))).append(i).toString();  
6 }
```



综上已知：即使使用 + 号作为字符串的拼接，也一样可以被编译器优化成 StringBuilder 的方式。但再细致些，你会发现在编译器优化的代码中，每次循环都会生成一个新的 StringBuilder 实例，同样也会降低系统的性能。

所以平时做字符串拼接的时候，我建议你还是要显示地使用 String Builder 来提升系统性能。

如果在多线程编程中，String 对象的拼接涉及到线程安全，你可以使用 StringBuffer。但是要注意，由于 StringBuffer 是线程安全的，涉及到锁竞争，所以从性能上来说，要比 StringBuilder 差一些。

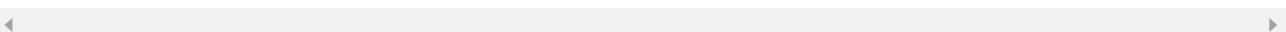
2. 如何使用 `String.intern` 节省内存？

讲完了构建字符串，我们再来讨论下 String 对象的存储问题。先看一个案例。

Twitter 每次发布消息状态的时候，都会产生一个地址信息，以当时 Twitter 用户的规模预估，服务器需要 32G 的内存来存储地址信息。

 复制代码

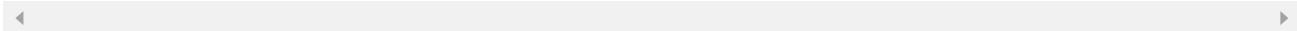
```
1 public class Location {  
2     private String city;  
3     private String region;  
4     private String countryCode;  
5     private double longitude;  
6     private double latitude;  
7 }
```



考虑到其中有很多用户在地址信息上是有重合的，比如，国家、省份、城市等，这时就可以将这部分信息单独列出一个类，以减少重复，代码如下：

 复制代码

```
1 public class SharedLocation {  
2  
3     private String city;  
4     private String region;  
5     private String countryCode;  
6  
7 }  
8  
9 public class Location {  
10  
11     private SharedLocation sharedLocation;  
12     double longitude;  
13     double latitude;  
14 }
```



通过优化，数据存储大小减到了 20G 左右。但对于内存存储这个数据来说，依然很大，怎么办呢？

这个案例来自一位 Twitter 工程师在 QCon 全球软件开发大会上的演讲，他们想到的解决方法，就是使用 `String.intern` 来节省内存空间，从而优化 `String` 对象的存储。

具体做法就是，在每次赋值的时候使用 `String` 的 `intern` 方法，如果常量池中有相同值，就会重复使用该对象，返回对象引用，这样一开始的对象就可以被回收掉。这种方式可以使重复性非常高的地址信息存储大小从 20G 降到几百兆。

 复制代码

```
1 SharedLocation sharedLocation = new SharedLocation();  
2  
3 sharedLocation.setCity(messageInfo.getCity().intern());           sharedLocation.setCount  
4 sharedLocation.setRegion(messageInfo.getCountryCode().intern());  
5  
6 Location location = new Location();  
7 location.set(sharedLocation);  
8 location.set(messageInfo.getLongitude());  
9 location.set(messageInfo.getLatitude());
```



为了更好地理解，我们再来通过一个简单的例子，回顾下其中的原理：

 复制代码

```
1 String a =new String("abc").intern();
2 String b = new String("abc").intern();
3
4 if(a==b) {
5     System.out.print("a==b");
6 }
```

输出结果：

 复制代码

```
1 a==b
```

在字符串常量中，默认会将对象放入常量池；在字符串变量中，对象是会创建在堆内存中，同时也会在常量池中创建一个字符串对象，复制到堆内存对象中，并返回堆内存对象引用。

如果调用 `intern` 方法，会去查看字符串常量池中是否有等于该对象的字符串，如果没有，就在常量池中新增该对象，并返回该对象引用；如果有，就返回常量池中的字符串引用。堆内存中原有的对象由于没有引用指向它，将会通过垃圾回收器回收。

了解了原理，我们再一起看看上边的例子。

在一开始创建 `a` 变量时，会在堆内存中创建一个对象，同时会在加载类时，在常量池中创建一个字符串对象，在调用 `intern` 方法之后，会去常量池中查找是否有等于该字符串的对象，有就返回引用。

在创建 `b` 字符串变量时，也会在堆中创建一个对象，此时常量池中有该字符串对象，就不再创建。调用 `intern` 方法则会去常量池中判断是否有等于该字符串的对象，发现有等于"abc"字符串的对象，就直接返回引用。而在堆内存中的对象，由于没有引用指向它，将会被垃圾回收。所以 `a` 和 `b` 引用的是同一个对象。

下面我用一张图来总结下 `String` 字符串的创建分配内存地址情况：



使用 `intern` 方法需要注意的一点是，一定要结合实际场景。因为常量池的实现是类似于一个 `HashTable` 的实现方式，`HashTable` 存储的数据越大，遍历的时间复杂度就会增加。如果数据过大，会增加整个字符串常量池的负担。

3. 如何使用字符串的分割方法？

最后我想跟你聊聊字符串的分割，这种方法在编码中也很常见。`Split()` 方法使用了正则表达式实现了其强大的分割功能，而正则表达式的性能是非常不稳定的，使用不恰当会引起回溯问题，很可能导致 CPU 居高不下。

所以我们应该慎重使用 `Split()` 方法，我们可以用 `String.indexOf()` 方法代替 `Split()` 方法完成字符串的分割。如果实在无法满足需求，你就在使用 `Split()` 方法时，对回溯问题加以重视就可以了。

总结

这一讲中，我们认识到做好 `String` 字符串性能优化，可以提高系统的整体性能。在这个理论基础上，Java 版本在迭代中通过不断地更改成员变量，节约内存空间，对 `String` 对象进行优化。

我们还特别提到了 `String` 对象的不可变性，正是这个特性实现了字符串常量池，通过减少同一个值的字符串对象的重复创建，进一步节约内存。

但也是因为这个特性，我们在做长字符串拼接时，需要显示使用 `StringBuilder`，以提高字符串的拼接性能。最后，在优化方面，我们还可以使用 `intern` 方法，让变量字符串对象重复使用常量池中相同值的对象，进而节约内存。

最后再分享一个个人观点。那就是千里之堤，溃于蚁穴。日常编程中，我们往往可能就是对一个小小的字符串了解不够深入，使用不够恰当，从而引发线上事故。

04 | 慎重使用正则表达式



上一讲，我在讲 String 对象优化时，提到了 Split() 方法，该方法使用的正则表达式可能引起回溯问题，今天我们就来深入了解下，这究竟是怎么回事？

开始之前，我们先来看一个案例，可以帮助你更好地理解内容。

在一次小型项目开发中，我遇到过这样一个问题。为了宣传新品，我们开发了一个小程序，按照之前评估的访问量，这次活动预计参与用户量 30W+，TPS（每秒事务处理量）最高 3000 左右。

这个结果来自我对接口做的微基准性能测试。我习惯使用 ab 工具（通过 yum -y install httpd-tools 可以快速安装）在另一台机器上对 http 请求接口进行测试。

我可以通过设置 -n 请求数 /-c 并发用户数来模拟线上的峰值请求，再通过 TPS、RT（每秒响应时间）以及每秒请求时间分布情况这三个指标来衡量接口的性能，如下图所示（图中隐藏部分为我的服务器地址）：

```
[root@192.168.1.10 ~]# ab -n100000 -c 1000 http://192.168.1.10:8080/test
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 10.11.100.146 (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Completed 100000 requests
Finished 100000 requests

Server Software:
Server Hostname:      -----
Server Port:          ----

Document Path:         -
Document Length:       36801 bytes

Concurrency Level:    1000
Time taken for tests: 30.024 seconds
Complete requests:   100000
Failed requests:     0
Write errors:         0
Total transferred:   3692000000 bytes
HTML transferred:    3680100000 bytes
Requests per second: 3330.63 #[/sec] (mean)
Time per request:    300.243 [ms] (mean)
Time per request:    0.300 [ms] (mean, across all concurrent requests)
Transfer rate:        120084.87 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0 117 384.3    11  7018
Processing:     4 171 236.6   136  6066
Waiting:        2 148 236.7   110  6061
Total:         13 288 444.7   164  7239

Percentage of the requests served within a certain time (ms)
 50% 164
 66% 202
 75% 232
 80% 261
 90% 1063
 95% 1167
 98% 1281
 99% 1396
100% 7239 (longest request)
```

就在做性能测试的时候，我发现有一个提交接口的 TPS 一直上不去，按理说这个业务非常简单，存在性能瓶颈的可能性并不大。

我迅速使用了排除法查找问题。首先将方法里面的业务代码全部注释，留一个空方法在这里，再看性能如何。这种方式能够很好地区分是框架性能问题，还是业务代码性能问题。

我快速定位到了是业务代码问题，就马上逐一查看代码查找原因。我将插入数据库操作代码加上之后，TPS 稍微下降了，但还是没有找到原因。最后，就只剩下 Split() 方法操作了，果然，我将 Split() 方法加入之后，TPS 明显下降了。

可是一个 Split() 方法为什么会影响到 TPS 呢？下面我们就来了解下正则表达式的相关内容，学完了答案也就出来了。

什么是正则表达式？

很基础，这里带你简单回顾一下。

正则表达式是计算机科学的一个概念，很多语言都实现了它。正则表达式使用一些特定的元字符来检索、匹配以及替换符合规则的字符串。

构造正则表达式语法的元字符，由普通字符、标准字符、限定字符（量词）、定位字符（边界字符）组成。详情可见下图：



正则表达式引擎

正则表达式是一个用正则符号写出的公式，程序对这个公式进行语法分析，建立一个语法分析树，再根据这个分析树结合正则表达式的引擎生成执行程序（这个执行程序我们把它称作状态机，也叫状态自动机），用于字符匹配。

而这里的正则表达式引擎就是一套核心算法，用于建立状态机。

目前实现正则表达式引擎的方式有两种：DFA 自动机（Deterministic Final Automata 确定有限状态自动机）和 NFA 自动机（Non deterministic Finite Automaton 非确定有限状态自动机）。

对比来看，构造 DFA 自动机的代价远大于 NFA 自动机，但 DFA 自动机的执行效率高于 NFA 自动机。

假设一个字符串的长度是 n ，如果用 DFA 自动机作为正则表达式引擎，则匹配的时间复杂度为 $O(n)$ ；如果用 NFA 自动机作为正则表达式引擎，由于 NFA 自动机在匹配过程中存在大量的分支和回溯，假设 NFA 的状态数为 s ，则该匹配算法的时间复杂度为 $O(ns)$ 。

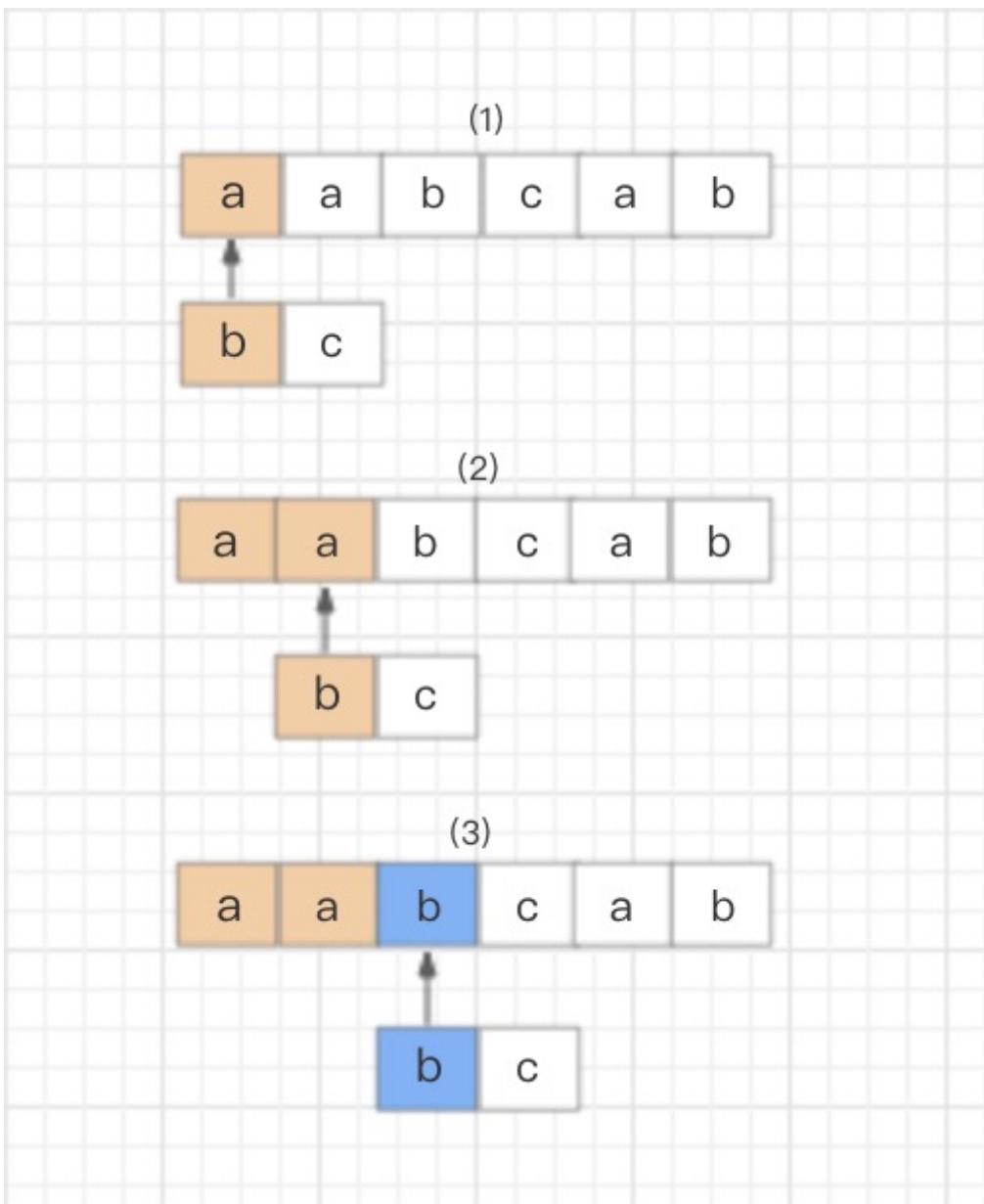
NFA 自动机的优势是支持更多功能。例如，捕获 group、环视、占有优先量词等高级功能。这些功能都是基于子表达式独立进行匹配，因此在编程语言里，使用的正则表达式库都是基于 NFA 实现的。

那么 NFA 自动机到底是怎么进行匹配的呢？我以下面的字符和表达式来举例说明。

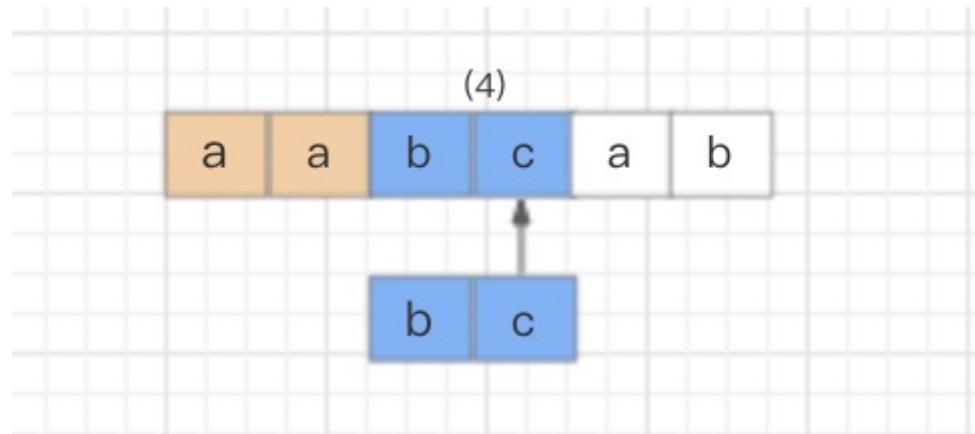
```
text= "aabcab"  
regex= "bc"
```

NFA 自动机会读取正则表达式的每一个字符，拿去和目标字符串匹配，匹配成功就换正则表达式的下一个字符，反之就继续和目标字符串的下一个字符进行匹配。分解一下过程。

首先，读取正则表达式的第一个匹配符和字符串的第一个字符进行比较， b 对 a ，不匹配；继续换字符串的下一个字符，也是 a ，不匹配；继续换下一个，是 b ，匹配。



然后，同理，读取正则表达式的第二个匹配符和字符串的第四个字符进行比较， c 对 c ，匹配；继续读取正则表达式的下一个字符，然而后面已经没有可匹配的字符了，结束。



这就是 NFA 自动机的匹配过程，虽然在实际应用中，碰到的正则表达式都要比这复杂，但匹配方法是一样的。

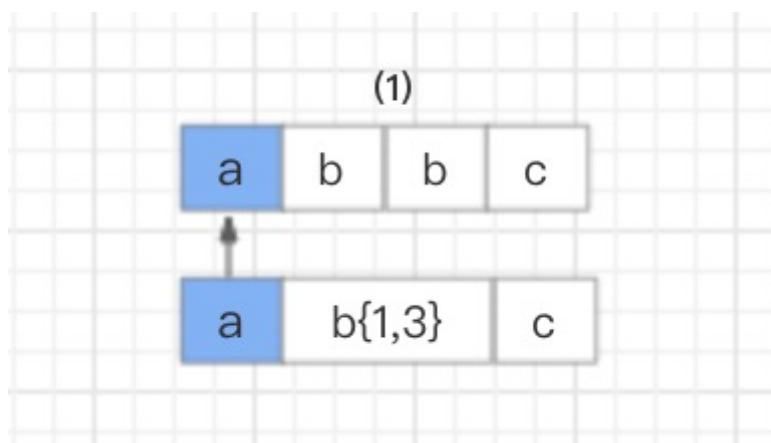
NFA 自动机的回溯

用 NFA 自动机实现的比较复杂的正则表达式，在匹配过程中经常会引起回溯问题。大量的回溯会长时间地占用 CPU，从而带来系统性能开销。我来举例说明。

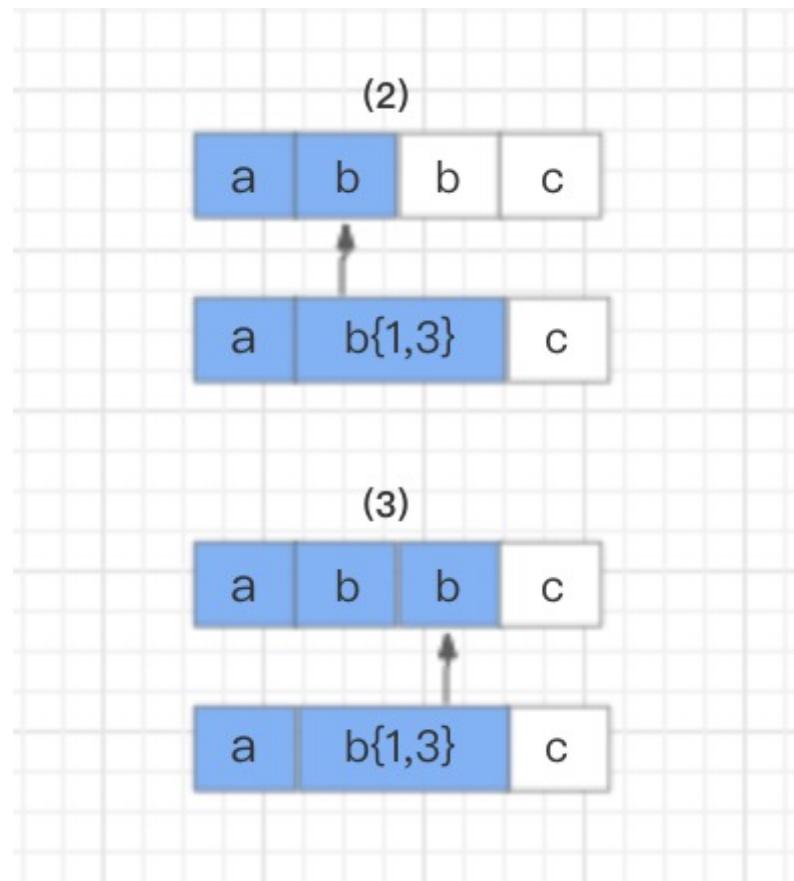
```
text= "abbc"  
regex= "ab{1,3}c"
```

这个例子，匹配目的比较简单。匹配以 a 开头，以 c 结尾，中间有 1-3 个 b 字符的字符串。NFA 自动机对其解析的过程是这样的：

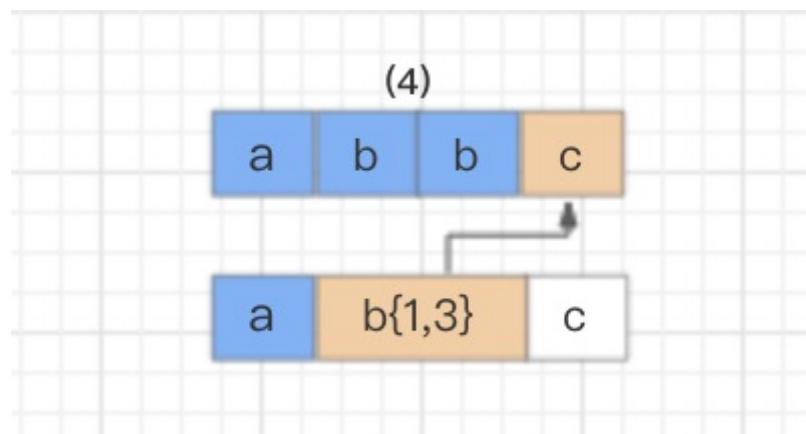
首先，读取正则表达式第一个匹配符 a 和字符串第一个字符 a 进行比较，a 对 a，匹配。



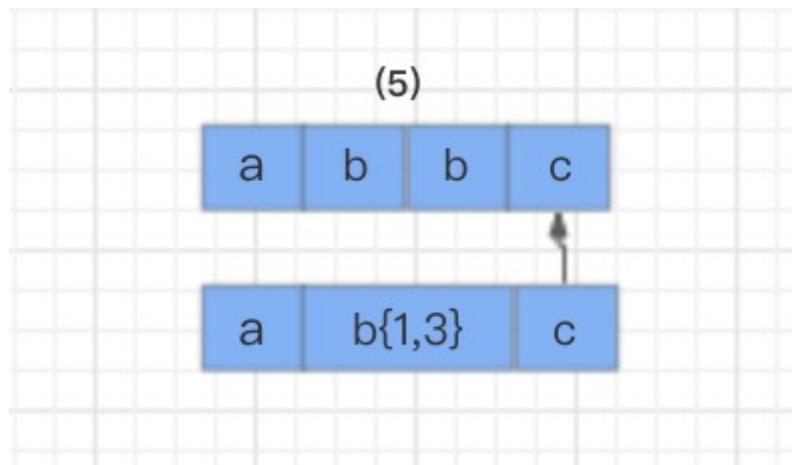
然后，读取正则表达式第二个匹配符 $b\{1,3}$ 和字符串的第二个字符 b 进行比较，匹配。但因为 $b\{1,3}$ 表示 1-3 个 b 字符串，NFA 自动机又具有贪婪特性，所以此时不会继续读取正则表达式的下一个匹配符，而是依旧使用 $b\{1,3}$ 和字符串的第三个字符 b 进行比较，结果还是匹配。



接着继续使用 $b\{1,3\}$ 和字符串的第四个字符 c 进行比较，发现不匹配了，此时就会发生回溯，已经读取的字符串第四个字符 c 将被吐出去，指针回到第三个字符 b 的位置。



那么发生回溯以后，匹配过程怎么继续呢？程序会读取正则表达式的下一个匹配符 c ，和字符串中的第四个字符 c 进行比较，结果匹配，结束。



如何避免回溯问题？

既然回溯会给系统带来性能开销，那我们如何应对呢？如果你有仔细看上面那个案例的话，你会发现 NFA 自动机的贪婪特性就是导火索，这和正则表达式的匹配模式息息相关，一起来了解一下。

1. 贪婪模式 (Greedy)

顾名思义，就是在数量匹配中，如果单独使用 +、?、* 或 {min,max} 等量词，正则表达式会匹配尽可能多的内容。

例如，上边那个例子：

```
text= "abbc"
regex= "ab{1,3}c"
```

就是在贪婪模式下，NFA 自动机读取了最大的匹配范围，即匹配 3 个 b 字符。匹配发生了一次失败，就引起了一次回溯。如果匹配结果是 "abbbc"，就会匹配成功。

```
text= "abbbc"
regex= "ab{1,3}c"
```

2. 懒惰模式 (Reluctant)

在该模式下，正则表达式会尽可能少地重复匹配字符。如果匹配成功，它会继续匹配剩余的字符串。

例如，在上面例子的字符后面加一个“？” ，就可以开启懒惰模式。

```
text= "abc"  
regex= "ab{1,3}?c"
```

匹配结果是“abc” ，该模式下 NFA 自动机首先选择最小的匹配范围，即匹配 1 个 b 字符，因此就避免了回溯问题。

3. 独占模式 (Possessive)

同贪婪模式一样，独占模式一样会最大限度地匹配更多内容；不同的是，在独占模式下，匹配失败就会结束匹配，不会发生回溯问题。

还是上边的例子，在字符后面加一个“+” ，就可以开启独占模式。

```
text= "abbc"  
regex= "ab{1,3}+bc"
```

结果是不匹配，结束匹配，不会发生回溯问题。讲到这里，你应该非常清楚了，**避免回溯的方法就是：使用懒惰模式和独占模式。**

还有开头那道“一个 split() 方法为什么会影响到 TPS”的存疑，你应该也清楚了吧？

我使用了 split() 方法提取域名，并检查请求参数是否符合规定。split() 在匹配分组时遇到特殊字符产生了大量回溯，我当时是在正则表达式后加了一个需要匹配的字符和“+” ，解决了这个问题。

 复制代码

```
1 \\\?(([A-Za-z0-9-~_=\\%]+\\&{0,1})+)
```

正则表达式的优化

正则表达式带来的性能问题，给我敲了个警钟，在这里我也希望分享给你一些心得。任何一个细节问题，都有可能导致性能问题，而这背后折射出来的是我们对这项技术的了解不够透

彻。所以我鼓励你学习性能调优，要掌握方法论，学会透过现象看本质。下面我就总结几种正则表达式的优化方法给你。

1. 少用贪婪模式，多用独占模式

贪婪模式会引起回溯问题，我们可以使用独占模式来避免回溯。前面详解过了，这里我就不在解释了。

2. 减少分支选择

分支选择类型 “(X|Y|Z)” 的正则表达式会降低性能，我们在开发的时候要尽量减少使用。如果一定要用，我们可以通过以下几种方式来优化：

首先，我们需要考虑选择的顺序，将比较常用的选择项放在前面，使它们可以较快地被匹配；

其次，我们可以尝试提取共用模式，例如，将 “(abcd|abef)” 替换为 “ab(cd|ef)” ，后者匹配速度较快，因为 NFA 自动机会尝试匹配 ab，如果没有找到，就不会再尝试任何选项；

最后，如果是简单的分支选择类型，我们可以用三次 index 代替 “(X|Y|Z)” ，如果测试的话，你就会发现三次 index 的效率要比 “(X|Y|Z)” 高出一些。

3. 减少捕获嵌套

在讲这个方法之前，我先简单介绍下什么是捕获组和非捕获组。

捕获组是指把正则表达式中，子表达式匹配的内容保存到以数字编号或显式命名的数组中，方便后面引用。一般一个 () 就是一个捕获组，捕获组可以进行嵌套。

非捕获组则是指参与匹配却不进行分组编号的捕获组，其表达式一般由 (?:exp) 组成。

在正则表达式中，每个捕获组都有一个编号，编号 0 代表整个匹配到的内容。我们可以看下面的例子：

 复制代码

```
1 public static void main( String[] args )
```

```
2 {
3     String text = "<input high=\"20\" weight=\"70\">test</input>";
4     String reg="(<input.*?>)(.*?)(</input>)"; 
5     Pattern p = Pattern.compile(reg);
6     Matcher m = p.matcher(text);
7     while(m.find()) {
8         System.out.println(m.group(0));// 整个匹配到的内容
9         System.out.println(m.group(1));//( <input.*?>)
10        System.out.println(m.group(2));//(.*?)
11        System.out.println(m.group(3));//( </input>)
12    }
13 }
```

◀ ▶

运行结果：

 复制代码

```
1 <input high="20" weight="70">test</input>
2 <input high="20" weight="70">
3 test
4 </input>
```

◀ ▶

如果你并不需要获取某一个分组内的文本，那么就使用非捕获分组。例如，使用 “(?:X)” 代替 “(X)”，我们再看下面的例子：

 复制代码

```
1 public static void main( String[] args )
2 {
3     String text = "<input high=\"20\" weight=\"70\">test</input>";
4     String reg="(?:<input.*?>)(.*?)(?:</input>)"; 
5     Pattern p = Pattern.compile(reg);
6     Matcher m = p.matcher(text);
7     while(m.find()) {
8         System.out.println(m.group(0));// 整个匹配到的内容
9         System.out.println(m.group(1));//(.*?)
10    }
11 }
```

◀ ▶

运行结果：

 复制代码

```
1 <input high=\"20\" weight=\"70\">test</input>
2 test
```

综上可知：减少不需要获取的分组，可以提高正则表达式的性能。

总结

正则表达式虽然小，却有着强大的匹配功能。我们经常用到它，比如，注册页面手机号或邮箱的校验。

但很多时候，我们又会因为它小而忽略它的使用规则，测试用例中又没有覆盖到一些特殊用例，不乏上线就中招的情况发生。

综合我以往的经验来看，如果使用正则表达式能使你的代码简洁方便，那么在做好性能排查的前提下，可以去使用；如果不能，那么正则表达式能不用就不用，以此避免造成更多的性能问题。

05 | ArrayList还是LinkedList? 使用不当性能差千倍



集合作为一种存储数据的容器，是我们日常开发中使用最频繁的对象类型之一。JDK 为开发者提供了一系列的集合类型，这些集合类型使用不同的数据结构来实现。因此，不同的集合类型，使用场景也不同。

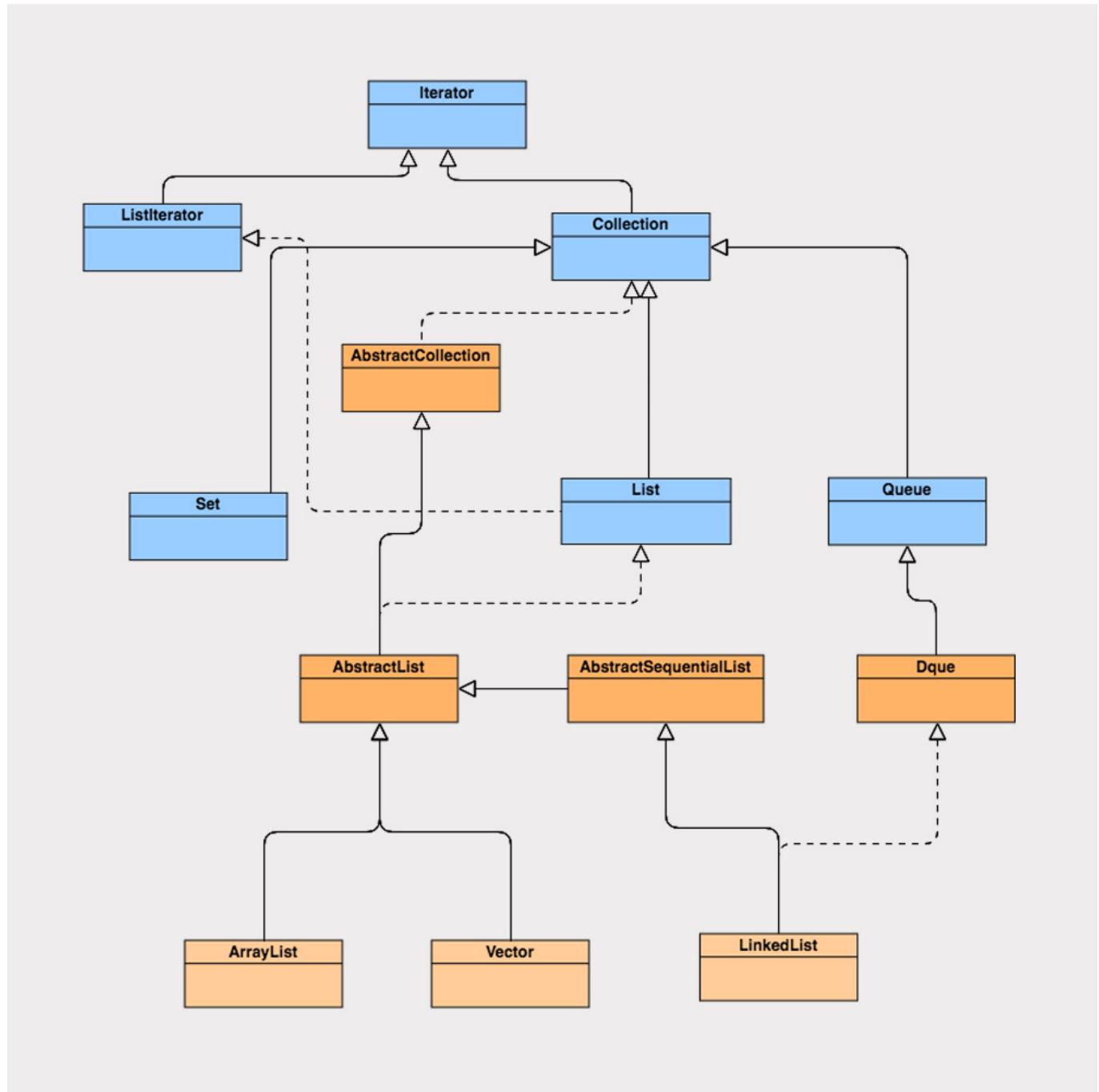
很多同学在面试的时候，经常会被问到集合的相关问题，比较常见的有 ArrayList 和 LinkedList 的区别。

相信大部分同学都能回答上：“ArrayList 是基于数组实现，LinkedList 是基于链表实现。”

而在回答使用场景的时候，我发现大部分同学的答案是：“**ArrayList 和 LinkedList 在新增、删除元素时，LinkedList 的效率要高于 ArrayList，而在遍历的时候，ArrayList 的效率要高于 LinkedList。**”这个回答是否准确呢？今天这一讲就带你验证。

初识 List 接口

在学习 List 集合类之前，我们先来通过这张图，看下 List 集合类的接口和类的实现关系：



我们可以看到 **ArrayList**、**Vector**、**LinkedList** 集合类继承了 **AbstractList** 抽象类，而 **AbstractList** 实现了 **List** 接口，同时也继承了 **AbstractCollection** 抽象类。**ArrayList**、**Vector**、**LinkedList** 又根据自我定位，分别实现了各自的功能。

ArrayList 和 Vector 使用了数组实现，这两者的实现原理差不多，LinkedList 使用了双向链表实现。基础铺垫就到这里，接下来，我们就详细地分析下 ArrayList 和 LinkedList 的源码实现。

ArrayList 是如何实现的？

ArrayList 很常用，先来几道测试题，自检下你对 ArrayList 的了解程度。

问题 1：我们在查看 ArrayList 的实现类源码时，你会发现对象数组 elementData 使用了 transient 修饰，我们知道 transient 关键字修饰该属性，则表示该属性不会被序列化，然而我们并没有看到文档中说明 ArrayList 不能被序列化，这是为什么？

问题 2：我们在使用 ArrayList 进行新增、删除时，经常被提醒“使用 ArrayList 做新增删除操作会影响效率”。那是不是 ArrayList 在大量新增元素的场景下效率就一定会变慢呢？

问题 3：如果让你使用 for 循环以及迭代循环遍历一个 ArrayList，你会使用哪种方式呢？原因是什？

如果你对这几道测试都没有一个全面的了解，那就跟我一起从数据结构、实现原理以及源码角度重新认识下 ArrayList 吧。

1.ArrayList 实现类

ArrayList 实现了 List 接口，继承了 AbstractList 抽象类，底层是数组实现的，并且实现了自增扩容数组大小。

ArrayList 还实现了 Cloneable 接口和 Serializable 接口，所以他可以实现克隆和序列化。

ArrayList 还实现了 RandomAccess 接口。你可能对这个接口比较陌生，不知道具体的用处。通过代码我们可以发现，这个接口其实是一个空接口，什么也没有实现，那 ArrayList 为什么要去实现它呢？

其实 RandomAccess 接口是一个标志接口，他标志着“只要实现该接口的 List 类，都能实现快速随机访问”。

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

2.ArrayList 属性

ArrayList 属性主要由数组长度 size、对象数组 elementData、初始化容量 default_capacity 等组成，其中初始化容量默认大小为 10。

 复制代码

```
1 // 默认初始化容量
2 private static final int DEFAULT_CAPACITY = 10;
3 // 对象数组
4 transient Object[] elementData;
5 // 数组长度
6 private int size;
```

从 ArrayList 属性来看，它没有被任何的多线程关键字修饰，但 elementData 被关键字 transient 修饰了。这就是我在上面提到的第一道测试题：transient 关键字修饰该字段则表示该属性不会被序列化，但 ArrayList 其实是实现了序列化接口，这到底是怎么回事呢？

这还得从“ArrayList 是基于数组实现”开始说起，由于 ArrayList 的数组是基于动态扩增的，所以并不是所有被分配的内存空间都存储了数据。

如果采用外部序列化法实现数组的序列化，会序列化整个数组。ArrayList 为了避免这些没有存储数据的内存空间被序列化，内部提供了两个私有方法 writeObject 以及 readObject 来自我完成序列化与反序列化，从而在序列化与反序列化数组时节省了空间和时间。

因此使用 transient 修饰数组，是防止对象数组被其他外部方法序列化。

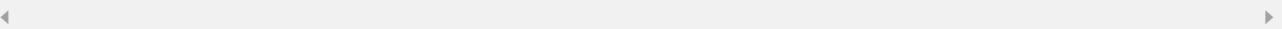
3.ArrayList 构造函数

ArrayList 类实现了三个构造函数，第一个是创建 ArrayList 对象时，传入一个初始化值；第二个是默认创建一个空数组对象；第三个是传入一个集合类型进行初始化。

当 ArrayList 新增元素时，如果所存储的元素已经超过其已有大小，它会计算元素大小后再进行动态扩容，数组的扩容会导致整个数组进行一次内存复制。因此，我们在初始化 ArrayList 时，可以通过第一个构造函数合理指定数组初始大小，这样有助于减少数组的扩容次数，从而提高系统性能。

 复制代码

```
1 public ArrayList(int initialCapacity) {  
2     // 初始化容量不为零时，将根据初始化值创建数组大小  
3     if (initialCapacity > 0) {  
4         this.elementData = new Object[initialCapacity];  
5     } else if (initialCapacity == 0) { // 初始化容量为零时，使用默认的空数组  
6         this.elementData = EMPTY_ELEMENTDATA;  
7     } else {  
8         throw new IllegalArgumentException("Illegal Capacity: "+  
9                         initialCapacity);  
10    }  
11 }  
12  
13 public ArrayList() {  
14     // 初始化默认为空数组  
15     this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;  
16 }
```



4.ArrayList 新增元素

ArrayList 新增元素的方法有两种，一种是直接将元素加到数组的末尾，另外一种是添加元素到任意位置。

 复制代码

```
1 public boolean add(E e) {  
2     ensureCapacityInternal(size + 1); // Increments modCount!!  
3     elementData[size++] = e;  
4     return true;  
5 }  
6  
7 public void add(int index, E element) {  
8     rangeCheckForAdd(index);  
9  
10    ensureCapacityInternal(size + 1); // Increments modCount!!  
11    System.arraycopy(elementData, index, elementData, index + 1,  
12                      size - index);  
13    elementData[index] = element;  
14    size++;  
15 }
```

两个方法的相同之处是在添加元素之前，都会先确认容量大小，如果容量够大，就不用进行扩容；如果容量不够大，就会按照原来数组的 1.5 倍大小进行扩容，在扩容之后需要将数组复制到新分配的内存地址。

 复制代码

```
1  private void ensureExplicitCapacity(int minCapacity) {
2      modCount++;
3
4      // overflow-conscious code
5      if (minCapacity - elementData.length > 0)
6          grow(minCapacity);
7
8  private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
9
10 private void grow(int minCapacity) {
11     // overflow-conscious code
12     int oldCapacity = elementData.length;
13     int newCapacity = oldCapacity + (oldCapacity >> 1);
14     if (newCapacity - minCapacity < 0)
15         newCapacity = minCapacity;
16     if (newCapacity - MAX_ARRAY_SIZE > 0)
17         newCapacity = hugeCapacity(minCapacity);
18     // minCapacity is usually close to size, so this is a win:
19     elementData = Arrays.copyOf(elementData, newCapacity);
20 }
```

当然，两个方法也有不同之处，添加元素到任意位置，会导致在该位置后的所有元素都需要重新排列，而将元素添加到数组的末尾，在没有发生扩容的前提下，是不会有元素复制排序过程的。

这里你就可以找到第二道测试题的答案了。如果我们在初始化时就比较清楚存储数据的大小，就可以在 ArrayList 初始化时指定数组容量大小，并且在添加元素时，只在数组末尾添加元素，那么 ArrayList 在大量新增元素的场景下，性能并不会变差，反而比其他 List 集合的性能要好。

5.ArrayList 删除元素

ArrayList 的删除方法和添加任意位置元素的方法是有些相同的。ArrayList 在每一次有效的删除元素操作之后，都要进行数组的重组，并且删除的元素位置越靠前，数组重组的开销就越大。

 复制代码

```
1 public E remove(int index) {  
2     rangeCheck(index);  
3  
4     modCount++;  
5     E oldValue = elementData(index);  
6  
7     int numMoved = size - index - 1;  
8     if (numMoved > 0)  
9         System.arraycopy(elementData, index+1, elementData, index,  
10                         numMoved);  
11    elementData[--size] = null; // clear to let GC do its work  
12  
13    return oldValue;  
14 }
```

6.ArrayList 遍历元素

由于 ArrayList 是基于数组实现的，所以在获取元素的时候是非常快捷的。

 复制代码

```
1 public E get(int index) {  
2     rangeCheck(index);  
3  
4     return elementData(index);  
5 }  
6  
7 E elementData(int index) {  
8     return (E) elementData[index];  
9 }
```

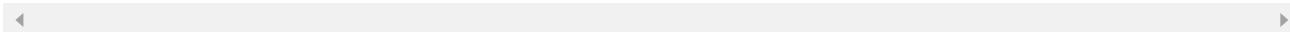
LinkedList 是如何实现的？

虽然 LinkedList 与 ArrayList 都是 List 类型的集合，但 LinkedList 的实现原理却和 ArrayList 大相径庭，使用场景也不太一样。

LinkedList 是基于双向链表数据结构实现的， LinkedList 定义了一个 Node 结构， Node 结构中包含了 3 个部分：元素内容 item、前指针 prev 以及后指针 next，代码如下。

 复制代码

```
1 private static class Node<E> {  
2     E item;  
3     Node<E> next;  
4     Node<E> prev;  
5  
6     Node(Node<E> prev, E element, Node<E> next) {  
7         this.item = element;  
8         this.next = next;  
9         this.prev = prev;  
10    }  
11 }
```



总结一下， LinkedList 就是由 Node 结构对象连接而成的一个双向链表。在 JDK1.7 之前， LinkedList 中只包含了一个 Entry 结构的 header 属性，并在初始化的时候默认创建一个空的 Entry，用来做 header，前后指针指向自己，形成一个循环双向链表。

在 JDK1.7 之后， LinkedList 做了很大的改动，对链表进行了优化。链表的 Entry 结构换成了 Node，内部组成基本没有改变，但 LinkedList 里面的 header 属性去掉了，新增了一个 Node 结构的 first 属性和一个 Node 结构的 last 属性。这样做有以下几点好处：

first/last 属性能更清晰地表达链表的链头和链尾概念；

first/last 方式可以在初始化 LinkedList 的时候节省 new 一个 Entry；

first/last 方式最重要的性能优化是链头和链尾的插入删除操作更加快捷了。

这里同 ArrayList 的讲解一样，我将从数据结构、实现原理以及源码分析等几个角度带你深入了解 LinkedList。

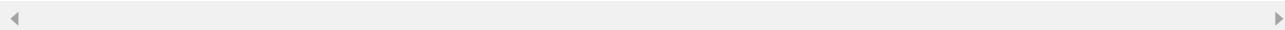
1.LinkedList 实现类

LinkedList 类实现了 List 接口、 Deque 接口，同时继承了 AbstractSequentialList 抽象类， LinkedList 既实现了 List 类型又有 Queue 类型的特点； LinkedList 也实现了 Cloneable 和 Serializable 接口，同 ArrayList 一样，可以实现克隆和序列化。

由于 `LinkedList` 存储数据的内存地址是不连续的，而是通过指针来定位不连续地址，因此，`LinkedList` 不支持随机快速访问，`LinkedList` 也就不能实现 `RandomAccess` 接口。

 复制代码

```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

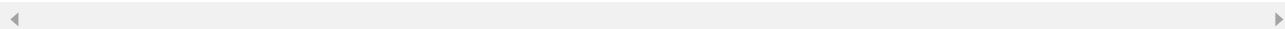


2. `LinkedList` 属性

我们前面讲到了 `LinkedList` 的两个重要属性 `first/last` 属性，其实还有一个 `size` 属性。我们可以看到这三个属性都被 `transient` 修饰了，原因很简单，我们在序列化的时候不会只对头尾进行序列化，所以 `LinkedList` 也是自行实现 `readObject` 和 `writeObject` 进行序列化与反序列化。

 复制代码

```
1 transient int size = 0;
2 transient Node<E> first;
3 transient Node<E> last;
```



3. `LinkedList` 新增元素

`LinkedList` 添加元素的实现很简洁，但添加的方式却有很多种。默认的 `add (Ee)` 方法是将添加的元素加到队尾，首先是将 `last` 元素置换到临时变量中，生成一个新的 `Node` 节点对象，然后将 `last` 引用指向新节点对象，之前的 `last` 对象的前指针指向新节点对象。

 复制代码

```
1 public boolean add(E e) {
2     linkLast(e);
3     return true;
4 }
5
6 void linkLast(E e) {
7     final Node<E> l = last;
8     final Node<E> newNode = new Node<>(l, e, null);
9     last = newNode;
10    if (l == null)
11        first = newNode;
```

```
12     else
13         l.next = newNode;
14     size++;
15     modCount++;
16 }
```

LinkedList 也有添加元素到任意位置的方法，如果我们是将元素添加到任意两个元素的中间位置，添加元素操作只会改变前后元素的前后指针，指针将会指向添加的新元素，所以相比 ArrayList 的添加操作来说，LinkedList 的性能优势明显。

 复制代码

```
1 public void add(int index, E element) {
2     checkPositionIndex(index);
3
4     if (index == size)
5         linkLast(element);
6     else
7         linkBefore(element, node(index));
8 }
9
10 void linkBefore(E e, Node<E> succ) {
11     // assert succ != null;
12     final Node<E> pred = succ.prev;
13     final Node<E> newNode = new Node<>(pred, e, succ);
14     succ.prev = newNode;
15     if (pred == null)
16         first = newNode;
17     else
18         pred.next = newNode;
19     size++;
20     modCount++;
21 }
```

4.LinkedList 删除元素

在 LinkedList 删除元素的操作中，我们首先要通过循环找到要删除的元素，如果要删除的位置处于 List 的前半段，就从前往后找；若其位置处于后半段，就从后往前找。

这样做的话，无论要删除较为靠前或较为靠后的元素都是非常高效的，但如果 List 拥有大量元素，移除的元素又在 List 的中间段，那效率相对来说会很低。

5.LinkedList 遍历元素

LinkedList 的获取元素操作实现跟 LinkedList 的删除元素操作基本类似，通过分前后半段来循环查找到对应的元素。但是通过这种方式来查询元素是非常低效的，特别是在 for 循环遍历的情况下，每一次循环都会去遍历半个 List。

所以在 LinkedList 循环遍历时，我们可以使用 iterator 方式迭代循环，直接拿到我们的元素，而不需要通过循环查找 List。

总结

前面我们已经从源码的实现角度深入了解了 ArrayList 和 LinkedList 的实现原理以及各自的特点。如果你能充分理解这些内容，很多实际应用中的相关性能问题也就迎刃而解了。

就像如果现在还有人跟你说，“ArrayList 和 LinkedList 在新增、删除元素时，LinkedList 的效率要高于 ArrayList，而在遍历的时候，ArrayList 的效率要高于 LinkedList”，你还会表示赞同吗？

现在我们不妨通过几组测试来验证一下。这里因为篇幅限制，所以我就直接给出测试结果了，对应的测试代码你可以访问 [查看和下载](#)。

1.ArrayList 和 LinkedList 新增元素操作测试

从集合头部位置新增元素

从集合中间位置新增元素

从集合尾部位置新增元素

测试结果 (花费时间)：

ArrayList > LinkedList

ArrayList < LinkedList

ArrayList < LinkedList

通过这组测试，我们可以知道 LinkedList 添加元素的效率未必必要高于 ArrayList。

由于 ArrayList 是数组实现的，而数组是一块连续的内存空间，在添加元素到数组头部的时候，需要对头部以后的数据进行复制重排，所以效率很低；而 LinkedList 是基于链表实现，在添加元素的时候，首先会通过循环查找到添加元素的位置，如果要添加的位置处于 List 的前半段，就从前往后找；若其位置处于后半段，就从后往前找。因此 LinkedList 添加元素到头部是非常高效的。

同上可知，ArrayList 在添加元素到数组中间时，同样有部分数据需要复制重排，效率也不是很高；LinkedList 将元素添加到中间位置，是添加元素最低效率的，因为靠近中间位置，在添加元素之前的循环查找是遍历元素最多的操作。

而在添加元素到尾部的操作中，我们发现，在没有扩容的情况下，ArrayList 的效率要高于 LinkedList。这是因为 ArrayList 在添加元素到尾部的时候，不需要复制重排数据，效率非常高。而 LinkedList 虽然也不用循环查找元素，但 LinkedList 中多了 new 对象以及变换指针指向对象的过程，所以效率要低于 ArrayList。

说明一下，这里我是基于 ArrayList 初始化容量足够，排除动态扩容数组容量的情况下进行的测试，如果有动态扩容的情况，ArrayList 的效率也会降低。

2. ArrayList 和 LinkedList 删除元素操作测试

从集合头部位置删除元素

从集合中间位置删除元素

从集合尾部位置删除元素

测试结果 (花费时间)：

ArrayList > LinkedList

ArrayList < LinkedList

ArrayList < LinkedList

ArrayList 和 LinkedList 删除元素操作测试的结果和添加元素操作测试的结果很接近，这是同样的原理，我在这里就不重复讲解了。

3. ArrayList 和 LinkedList 遍历元素操作测试

for(;) 循环

迭代器迭代循环

测试结果 (花费时间):

ArrayList<LinkedList

ArrayList≈LinkedList

我们可以看到， LinkedList 的 for 循环性能是最差的，而 ArrayList 的 for 循环性能是最好的。

这是因为 LinkedList 基于链表实现的，在使用 for 循环的时候，每一次 for 循环都会去遍历半个 List，所以严重影响了遍历的效率； ArrayList 则是基于数组实现的，并且实现了 RandomAccess 接口标志，意味着 ArrayList 可以实现快速随机访问，所以 for 循环效率非常高。

LinkedList 的迭代循环遍历和 ArrayList 的迭代循环遍历性能相当，也不会太差，所以在遍历 LinkedList 时，我们要切忌使用 for 循环遍历。

06 | Stream如何提高遍历集合效率？



上一讲中，我在讲 List 集合类，那我想你一定也知道集合的顶端接口 Collection。在 Java8 中，Collection 新增了两个流方法，分别是 Stream() 和 parallelStream()。

通过英文名不难猜测，这两个方法肯定和 Stream 有关，那进一步猜测，是不是和我们熟悉的 InputStream 和 OutputStream 也有关系呢？集合类中新增的两个 Stream 方法到底有什么作用？今天，我们就来深入了解下 Stream。

什么是 Stream ?

现在很多大数据量系统中都存在分表分库的情况。

例如，电商系统中的订单表，常常使用用户 ID 的 Hash 值来实现分表分库，这样是为了减少单个表的数据量，优化用户查询订单的速度。

但在后台管理员审核订单时，他们需要将各个数据源的数据查询到应用层之后进行合并操作。

例如，当我们需要查询出过滤条件下的所有订单，并按照订单的某个条件进行排序，单个数据源查询出来的数据是可以按照某个条件进行排序的，但多个数据源查询出来已经排序好的数据，并不代表合并后是正确的排序，所以我们需要在应用层对合并数据集合重新进行排序。

在 Java8 之前，我们通常是通过 for 循环或者 Iterator 迭代来重新排序合并数据，又或者通过重新定义 Collections.sorts 的 Comparator 方法来实现，这两种方式对于大数据量系统来说，效率并不是很理想。

Java8 中添加了一个新的接口类 Stream，他和我们之前接触的字节流概念不太一样，Java8 集合中的 Stream 相当于高级版的 Iterator，他可以通过 Lambda 表达式对集合进行各种非常便利、高效的聚合操作（Aggregate Operation），或者大批量数据操作（Bulk Data Operation）。

Stream 的聚合操作与数据库 SQL 的聚合操作 sorted、filter、map 等类似。我们在应用层就可以高效地实现类似数据库 SQL 的聚合操作了，而在数据操作方面，Stream 不仅可以通过串行的方式实现数据操作，还可以通过并行的方式处理大批量数据，提高数据的处理效率。

接下来我们就用一个简单的例子来体验下 Stream 的简洁与强大。

这个 Demo 的需求是过滤分组一所中学里身高在 160cm 以上的男女同学，我们先用传统的迭代方式来实现，代码如下：

 复制代码

```
1 Map<String, List<Student>> stuMap = new HashMap<String, List<Student>>();
2     for (Student stu: studentsList) {
3         if (stu.getHeight() > 160) { // 如果身高大于 160
4             if (stuMap.get(stu.getSex()) == null) { // 该性别还没分类
5                 List<Student> list = new ArrayList<Student>(); // 新建该性别学生的列表
6                 list.add(stu); // 将学生放进去列表
7             }
8             stuMap.get(stu.getSex()).add(stu);
9         }
10    }
11 }
```

```
7             stuMap.put(stu.getSex(), list); // 将列表放到 map 中
8     } else { // 该性别分类已存在
9         stuMap.get(stu.getSex()).add(stu); // 该性别分类已存在，则直接放进去即可
10    }
11 }
12 }
13 }
```



我们再使用 Java8 中的 Stream API 进行实现：

1. 串行实现

复制代码

```
1 Map<String, List<Student>> stuMap = stuList.stream().filter((Student s) -> s.getHeight()
```



2. 并行实现

复制代码

```
1 Map<String, List<Student>> stuMap = stuList.parallelStream().filter((Student s) -> s.getHeight()
```



通过上面两个简单的例子，我们可以发现，Stream 结合 Lambda 表达式实现遍历筛选功能非常得简洁和便捷。

Stream 如何优化遍历？

上面我们初步了解了 Java8 中的 Stream API，那 Stream 是如何做到优化迭代的呢？并行又是如何实现的？下面我们就透过 Stream 源码剖析 Stream 的实现原理。

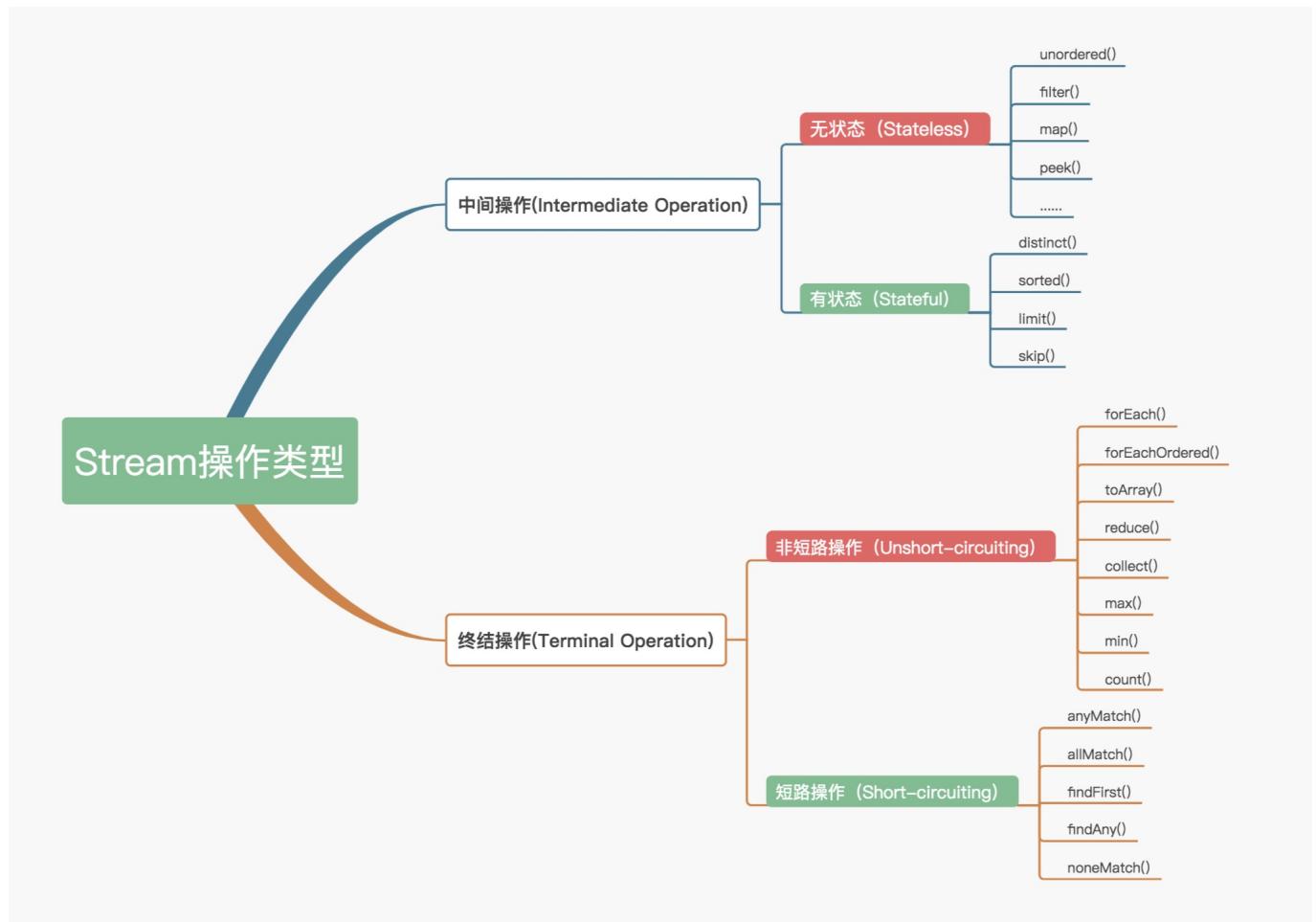
1. Stream 操作分类

在了解 Stream 的实现原理之前，我们先来了解下 Stream 的操作分类，因为他的操作分类其实是实现高效迭代大数据集合的重要原因之一。为什么这样说，分析完你就清楚了。

官方将 Stream 中的操作分为两大类：中间操作（Intermediate operations）和终结操作（Terminal operations）。中间操作只对操作进行了记录，即只会返回一个流，不会进行计算操作，而终结操作是实现了计算操作。

中间操作又可以分为无状态（Stateless）与有状态（Stateful）操作，前者是指元素的处理不受之前元素的影响，后者是指该操作只有拿到所有元素之后才能继续下去。

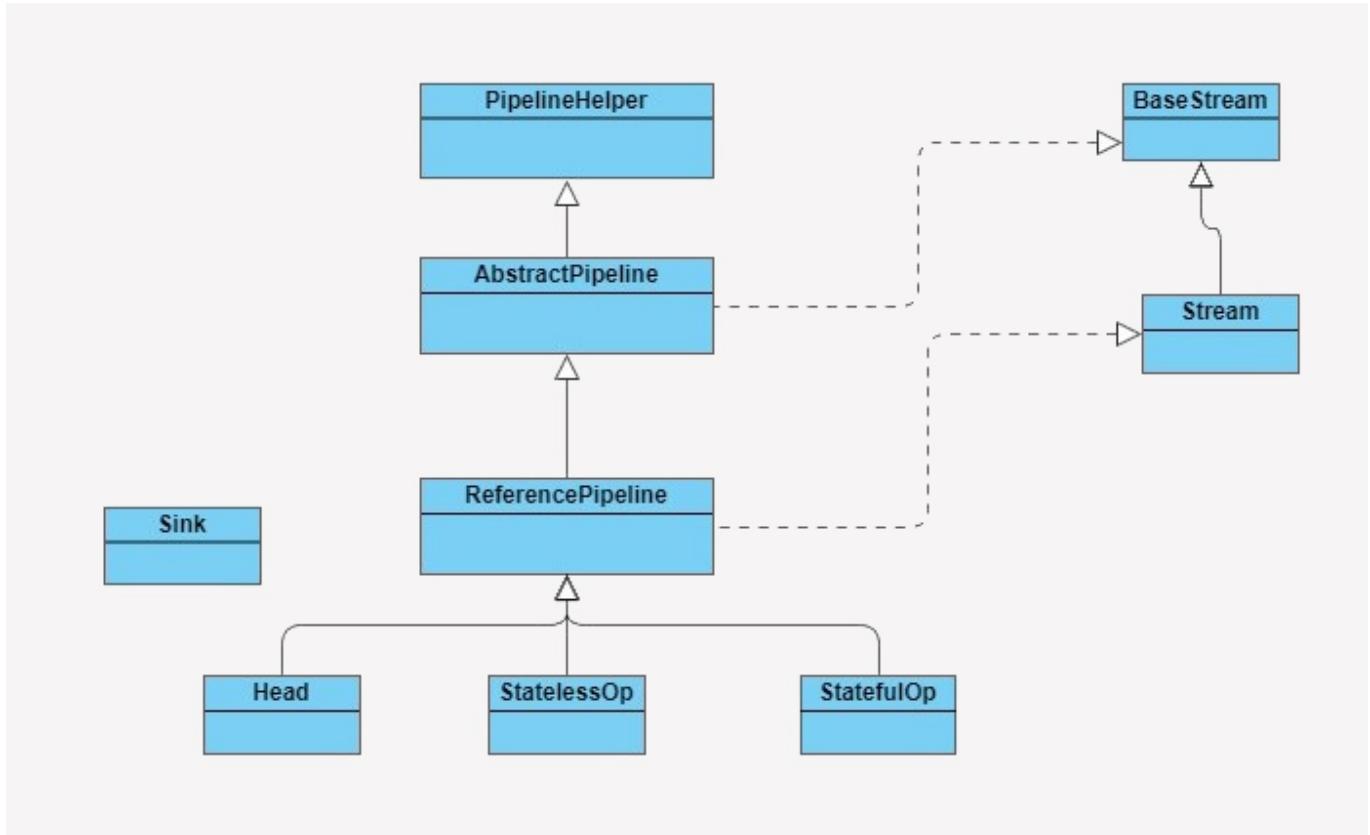
终结操作又可以分为短路（Short-circuiting）与非短路（Unshort-circuiting）操作，前者是指遇到某些符合条件的元素就可以得到最终结果，后者是指必须处理完所有元素才能得到最终结果。操作分类详情如下图所示：



我们通常还会将中间操作称为懒操作，也正是由这种懒操作结合终结操作、数据源构成的处理管道（Pipeline），实现了 Stream 的高效。

2. Stream 源码实现

在了解 Stream 如何工作之前，我们先来了解下 Stream 包是由哪些主要结构类组合而成的，各个类的职责是什么。参照下图：



BaseStream 和 Stream 为最顶端的接口类。BaseStream 主要定义了流的基本接口方法，例如，spliterator、isParallel 等；Stream 则定义了一些流的常用操作方法，例如，map、filter 等。

ReferencePipeline 是一个结构类，他通过定义内部类组装了各种操作流。他定义了 Head、StatelessOp、StatefulOp 三个内部类，实现了 BaseStream 与 Stream 的接口方法。

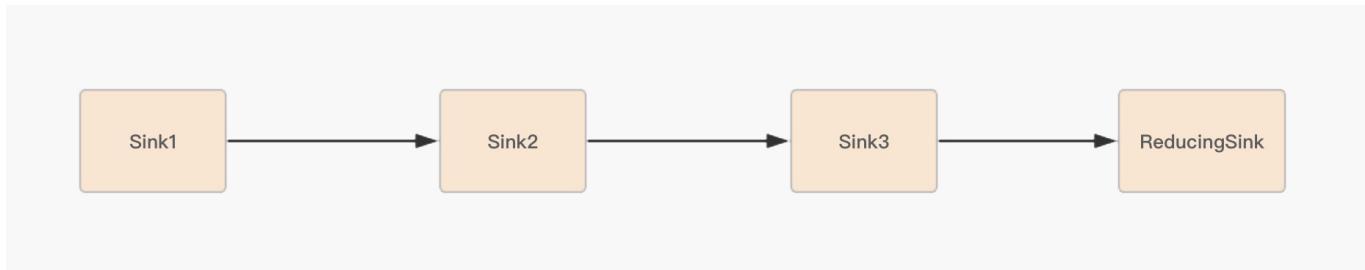
Sink 接口是定义每个 Stream 操作之间关系的协议，他包含 begin()、end()、cancellationRequested()、accept() 四个方法。ReferencePipeline 最终会将整个 Stream 流操作组装成一个调用链，而这条调用链上的各个 Stream 操作的上下关系就是通过 Sink 接口协议来定义实现的。

3.Stream 操作叠加

我们知道，一个 Stream 的各个操作是由处理管道组装，并统一完成数据处理的。在 JDK 中每次的中断操作会以使用阶段（Stage）命名。

管道结构通常是由 ReferencePipeline 类实现的，前面讲解 Stream 包结构时，我提到过 ReferencePipeline 包含了 Head、StatelessOp、StatefulOp 三种内部类。

Head 类主要用来定义数据源操作，在我们初次调用 names.stream() 方法时，会初次加载 Head 对象，此时为加载数据源操作；接着加载的是中间操作，分别为无状态中间操作 StatelessOp 对象和有状态操作 StatefulOp 对象，此时的 Stage 并没有执行，而是通过 AbstractPipeline 生成了一个中间操作 Stage 链表；当我们调用终结操作时，会生成一个最终的 Stage，通过这个 Stage 触发之前的中间操作，从最后一个 Stage 开始，递归产生一个 Sink 链。如下图所示：



下面我们再通过一个例子来感受下 Stream 的操作分类是如何实现高效迭代大数据集合的。

复制代码

```
1 List<String> names = Arrays.asList(" 张三 ", " 李四 ", " 王老五 ", " 李三 ", " 刘老四 ",  
2  
3 String maxLenStartWithZ = names.stream()  
4         .filter(name -> name.startsWith(" 张 "))  
5         .mapToInt(String::length)  
6         .max()  
7         .toString();
```

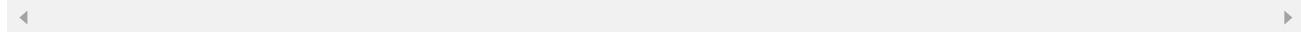
这个例子的需求是查找出一个长度最长，并且以张为姓氏的名字。从代码角度来看，你可能会认为是这样的操作流程：首先遍历一次集合，得到以“张”开头的所有名字；然后遍历一次 filter 得到的集合，将名字转换成数字长度；最后再从长度集合中找到最长的那个名字并且返回。

这里我要很明确地告诉你，实际情况并非如此。我们来逐步分析下这个方法里所有的操作是如何执行的。

首先，因为 names 是 ArrayList 集合，所以 names.stream() 方法将会调用集合类基础接口 Collection 的 Stream 方法：

复制代码

```
1 default Stream<E> stream() {
2     return StreamSupport.stream(splitterator(), false);
3 }
```



然后，Stream 方法就会调用 StreamSupport 类的 Stream 方法，方法中初始化了一个 ReferencePipeline 的 Head 内部类对象：

复制代码

```
1 public static <T> Stream<T> stream(Splitterator<T> splitterator, boolean parallel) {
2     Objects.requireNonNull(splitterator);
3     return new ReferencePipeline.Head<>(splitterator,
4                                         StreamOpFlag.fromCharacteristics(splitterato
5                                         parallel);
6 }
```



再调用 filter 和 map 方法，这两个方法都是无状态的中间操作，所以执行 filter 和 map 操作时，并没有进行任何的操作，而是分别创建了一个 Stage 来标识用户的每一次操作。

而通常情况下 Stream 的操作又需要一个回调函数，所以一个完整的 Stage 是由数据来源、操作、回调函数组成的三元组来表示。如下图所示，分别是 ReferencePipeline 的 filter 方法和 map 方法：

复制代码

```
1 @Override
2     public final Stream<P_OUT> filter(Predicate<? super P_OUT> predicate) {
3         Objects.requireNonNull(predicate);
4         return new StatelessOp<P_OUT, P_OUT>(this, StreamShape.REFERENCE,
5                                         StreamOpFlag.NOT_SIZED) {
6             @Override
7             Sink<P_OUT> opWrapSink(int flags, Sink<P_OUT> sink) {
8                 return new Sink.ChainedReference<P_OUT, P_OUT>(sink) {
9                     @Override
10                    public void begin(long size) {
11                        downstream.begin(-1);
12                    }
13
14                     @Override
15                     public void accept(P_OUT u) {
16                         if (predicate.test(u))
```

```
17             downstream.accept(u);
18         }
19     };
20 }
21 };
22 }
```

 复制代码

```
1 @Override
2 @SuppressWarnings("unchecked")
3 public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper) {
4     Objects.requireNonNull(mapper);
5     return new StatelessOp<P_OUT, R>(this, StreamShape.REFERENCE,
6                                         StreamOpFlag.NOT_SORTED | StreamOpFlag.NOT_DISTINCT);
7 }
8
9     @Override
10    Sink<P_OUT> opWrapSink(int flags, Sink<R> sink) {
11         return new Sink.ChainedReference<P_OUT, R>(sink) {
12             @Override
13             public void accept(P_OUT u) {
14                 downstream.accept(mapper.apply(u));
15             }
16         };
17     }
18 }
```

 复制代码

new StatelessOp 将会调用父类 AbstractPipeline 的构造函数，这个构造函数将前后的 Stage 联系起来，生成一个 Stage 链表：

```
1 AbstractPipeline(AbstractPipeline<?, E_IN, ?> previousStage, int opFlags) {
2     if (previousStage.linkedOrConsumed)
3         throw new IllegalStateException(MSG_STREAM_LINKED);
4     previousStage.linkedOrConsumed = true;
5     previousStage.nextStage = this;// 将当前的 stage 的 next 指针指向之前的 stage
6
7     this.previousStage = previousStage;// 赋值当前 stage 当全局变量 previousStage
8     this.sourceOrOpFlags = opFlags & StreamOpFlag.OP_MASK;
9     this.combinedFlags = StreamOpFlag.combineOpFlags(opFlags, previousStage.combinedFlags);
10    this.sourceStage = previousStage.sourceStage;
11    if (opIsStateful())
12        sourceStage.sourceAnyStateful = true;
```

```
13     this.depth = previousStage.depth + 1;
14 }
```

复制代码

```
1  @Override
2  public final Optional<P_OUT> max(Comparator<? super P_OUT> comparator) {
3      return reduce(BinaryOperator.maxBy(comparator));
4 }
```

复制代码

最后，调用 AbstractPipeline 的 wrapSink 方法，该方法会调用 opWrapSink 生成一个 Sink 链表，Sink 链表中的每一个 Sink 都封装了一个操作的具体实现。

```
1  @Override
2  @SuppressWarnings("unchecked")
3  final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {
4      Objects.requireNonNull(sink);
5
6      for ( @SuppressWarnings("rawtypes") AbstractPipeline p=AbstractPipeline.this; p
7          sink = p.opWrapSink(p.previousStage.combinedFlags, sink);
8      }
9      return (Sink<P_IN>) sink;
10 }
11 }
```

当 Sink 链表生成完成后，Stream 开始执行，通过 spliterator 迭代集合，执行 Sink 链表中的具体操作。

复制代码

```
1  @Override
2      final <P_IN> void copyInto(Sink<P_IN> wrappedSink, Spliterator<P_IN> spliterator) {
3          Objects.requireNonNull(wrappedSink);
4
5          if (!StreamOpFlag.SHORT_CIRCUIT.isKnown(getStreamAndOpFlags())) {
6              wrappedSink.begin(spliterator.getExactSizeIfKnown());
7              spliterator.forEachRemaining(wrappedSink);
8              wrappedSink.end();
9          }
10         else {
11             copyIntoWithCancel(wrappedSink, spliterator);
12         }
13     }
```

Java8 中的 Spliterator 的 forEachRemaining 会迭代集合，每迭代一次，都会执行一次 filter 操作，如果 filter 操作通过，就会触发 map 操作，然后将结果放入到临时数组 object 中，再进行下一次的迭代。完成中间操作后，就会触发终结操作 max。

这就是串行处理方式了，那么 Stream 的另一种处理数据的方式又是怎么操作的呢？

4. Stream 并行处理

Stream 处理数据的方式有两种，串行处理和并行处理。要实现并行处理，我们只需要在例子的代码中新增一个 Parallel() 方法，代码如下所示：

复制代码

```
1 List<String> names = Arrays.asList(" 张三 ", " 李四 ", " 王老五 ", " 李三 ", " 刘老四 ", "
2
3 String maxLenStartWithZ = names.stream()
4         .parallel()
5         .filter(name -> name.startsWith(" 张 "))
6         .mapToInt(String::length)
7         .max()
8         .toString();
```

Stream 的并行处理在执行终结操作之前，跟串行处理的实现是一样的。而在调用终结方法之后，实现的方式就有点不太一样，会调用 TerminalOp 的 evaluateParallel 方法进行并行处理。

 复制代码

```
1 final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
2     assert getOutputShape() == terminalOp.inputShape();
3     if (linkedOrConsumed)
4         throw new IllegalStateException(MSG_STREAM_LINKED);
5     linkedOrConsumed = true;
6
7     return isParallel()
8         ? terminalOp.evaluateParallel(this, sourceSpliterator(terminalOp.getOpFlavor()))
9         : terminalOp.evaluateSequential(this, sourceSpliterator(terminalOp.getOpFlavor()));
10 }
```



这里的并行处理指的是，Stream 结合了 ForkJoin 框架，对 Stream 处理进行了分片，SplitIterator 中的 estimateSize 方法会估算出分片的数据量。

ForkJoin 框架和估算算法，在这里我就不具体讲解了，如果感兴趣，你可以深入源码分析下该算法的实现。

通过预估的数据量获取最小处理单元的阀值，如果当前分片大小大于最小处理单元的阀值，就继续切分集合。每个分片将会生成一个 Sink 链表，当所有的分片操作完成后，ForkJoin 框架将会合并分片任何结果集。

合理使用 Stream

看到这里，你应该对 Stream API 是如何优化集合遍历有个清晰的认知了。Stream API 用起来简洁，还能并行处理，那是不是使用 Stream API，系统性能就更好呢？通过一组测试，我们一探究竟。

我们将对常规的迭代、Stream 串行迭代以及 Stream 并行迭代进行性能测试对比，迭代循环中，我们将对数据进行过滤、分组等操作。分别进行以下几组测试：

多核 CPU 服务器配置环境下，对比长度 100 的 int 数组的性能；

多核 CPU 服务器配置环境下，对比长度 1.00E+8 的 int 数组的性能；

多核 CPU 服务器配置环境下，对比长度 1.00E+8 对象数组过滤分组的性能；

单核 CPU 服务器配置环境下，对比长度 1.00E+8 对象数组过滤分组的性能。

由于篇幅有限，我这里直接给出统计结果，你也可以自己去验证一下，具体的测试代码可以在[Github](#)上查看。通过以上测试，我统计出的测试结果如下（迭代使用时间）：

常规的迭代 < Stream 并行迭代 < Stream 串行迭代

Stream 并行迭代 < 常规的迭代 < Stream 串行迭代

Stream 并行迭代 < 常规的迭代 < Stream 串行迭代

常规的迭代 < Stream 串行迭代 < Stream 并行迭代

通过以上测试结果，我们可以看到：在循环迭代次数较少的情况下，常规的迭代方式性能反而更好；在单核 CPU 服务器配置环境中，也是常规迭代方式更有优势；而在大数据循环迭代中，如果服务器是多核 CPU 的情况下，Stream 的并行迭代优势明显。所以我们在平时处理大数据的集合时，应该尽量考虑将应用部署在多核 CPU 环境下，并且使用 Stream 的并行迭代方式进行处理。

用事实说话，我们看到其实使用 Stream 未必可以使系统性能更佳，还是要结合应用场景进行选择，也就是合理地使用 Stream。

总结

纵观 Stream 的设计实现，非常值得我们学习。从大的设计方向上来说，Stream 将整个操作分解为了链式结构，不仅简化了遍历操作，还为实现了并行计算打下了基础。

从小的分类方向上来说，Stream 将遍历元素的操作和对元素的计算分为中间操作和终结操作，而中间操作又根据元素之间状态有无干扰分为有状态和无状态操作，实现了链结构中的不同阶段。

在串行处理操作中，Stream 在执行每一步中间操作时，并不会做实际的数据操作处理，而是将这些中间操作串联起来，最终由终结操作触发，生成一个数据处理链表，通过 Java8 中的 Spliterator 迭代器进行数据处理；此时，每执行一次迭代，就对所有的无状态的中间操作进行数据处理，而对有状态的中间操作，就需要迭代处理完所有的数据，再进行处理操作；最后就是进行终结操作的数据处理。

在并行处理操作中，Stream 对中间操作基本跟串行处理方式是一样的，但在终结操作中，Stream 将结合 ForkJoin 框架对集合进行切片处理，ForkJoin 框架将每个切片的处理结果 Join 合并起来。最后就是要注意 Stream 的使用场景。

07 | 深入浅出HashMap的设计与优化



在上一讲中我提到过 Collection 接口，那么在 Java 容器类中，除了这个接口之外，还定义了一个很重要的 Map 接口，主要用来存储键值对数据。

HashMap 作为我们日常使用最频繁的容器之一，相信你一定不陌生了。今天我们就从 HashMap 的底层实现讲起，深度了解下它的设计与优化。

常用的数据结构

我在 05 讲分享 List 集合类的时候，讲过 ArrayList 是基于数组的数据结构实现的，
LinkedList 是基于链表的数据结构实现的，而我今天要讲的 HashMap 是基于哈希表的数

据结构实现的。我们不妨一起来温习下常用的数据结构，这样也有助于你更好地理解后面地内容。

数组：采用一段连续的存储单元来存储数据。对于指定下标的查找，时间复杂度为 $O(1)$ ，但在数组中间以及头部插入数据时，需要复制移动后面的元素。

链表：一种在物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

链表由一系列结点（链表中每一个元素）组成，结点可以在运行时动态生成。每个结点都包含“存储数据单元的数据域”和“存储下一个结点地址的指针域”这两个部分。

由于链表不用必须按顺序存储，所以链表在插入的时候可以达到 $O(1)$ 的复杂度，但查找一个结点或者访问特定编号的结点需要 $O(n)$ 的时间。

哈希表：根据关键码值（Key value）直接进行访问的数据结构。通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做哈希函数，存放记录的数组就叫做哈希表。

树：由 n ($n \geq 1$) 个有限结点组成的一个具有层次关系的集合，就像是一棵倒挂的树。

HashMap 的实现结构

了解完数据结构后，我们再来看下 HashMap 的实现结构。作为最常用的 Map 类，它是基于哈希表实现的，继承了 AbstractMap 并且实现了 Map 接口。

哈希表将键的 Hash 值映射到内存地址，即根据键获取对应的值，并将其存储到内存地址。也就是说 HashMap 是根据键的 Hash 值来决定对应值的存储位置。通过这种索引方式，HashMap 获取数据的速度会非常快。

例如，存储键值对 (x , "aa") 时，哈希表会通过哈希函数 $f(x)$ 得到"aa"的实现存储位置。

但也会有新的问题。如果再来一个 (y , "bb")，哈希函数 $f(y)$ 的哈希值跟之前 $f(x)$ 是一样的，这样两个对象的存储地址就冲突了，这种现象就被称为哈希冲突。**那么哈希表是怎么解决的呢？方式有很多，比如，开放定址法、再哈希函数法和链地址法。**

开放定址法很简单，当发生哈希冲突时，如果哈希表未被装满，说明在哈希表中必然还有空位置，那么可以把 key 存放到冲突位置的空位置上去。这种方法存在着很多缺点，例如，查找、扩容等，所以我不建议你作为解决哈希冲突的首选。

再哈希法顾名思义就是在同义词产生地址冲突时再计算另一个哈希函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但却增加了计算时间。如果我们不考虑添加元素的时间成本，且对查询元素的要求极高，就可以考虑使用这种算法设计。

HashMap 则是综合考虑了所有因素，采用链地址法解决哈希冲突问题。这种方法是采用了数组（哈希表）+ 链表的数据结构，当发生哈希冲突时，就用一个链表结构存储相同 Hash 值的数据。

HashMap 的重要属性

从 HashMap 的源码中，我们可以发现，HashMap 是由一个 Node 数组构成，每个 Node 包含了一个 key-value 键值对。

 复制代码

```
1 transient Node<K,V>[] table;
```

◀ ▶

Node 类作为 HashMap 中的一个内部类，除了 key、value 两个属性外，还定义了一个 next 指针。当有哈希冲突时，HashMap 会用之前数组当中相同哈希值对应存储的 Node 对象，通过指针指向新增的相同哈希值的 Node 对象的引用。

 复制代码

```
1 static class Node<K,V> implements Map.Entry<K,V> {
2     final int hash;
3     final K key;
4     V value;
5     Node<K,V> next;
6
7     Node(int hash, K key, V value, Node<K,V> next) {
8         this.hash = hash;
9         this.key = key;
10        this.value = value;
11        this.next = next;
12    }
13 }
```

HashMap 还有两个重要的属性：加载因子 (loadFactor) 和边界值 (threshold)。在初始化 HashMap 时，就会涉及到这两个关键初始化参数。

 复制代码

```
1 int threshold;  
2  
3     final float loadFactor;
```

LoadFactor 属性是用来间接设置 Entry 数组（哈希表）的内存空间大小，在初始 HashMap 不设置参数的情况下，默认 LoadFactor 值为 0.75。**为什么是 0.75 这个值呢？**

这是因为对于使用链表法的哈希表来说，查找一个元素的平均时间是 $O(1+n)$ ，这里的 n 指的是遍历链表的长度，因此加载因子越大，对空间的利用就越充分，这就意味着链表的长度越长，查找效率也就越低。如果设置的加载因子太小，那么哈希表的数据将过于稀疏，对空间造成严重浪费。

那有没有什么办法来解决这个因链表过长而导致的查询时间复杂度高的问题呢？你可以先想想，我将在后面的内容中讲到。

Entry 数组的 Threshold 是通过初始容量和 LoadFactor 计算所得，在初始 HashMap 不设置参数的情况下，默认边界值为 12。如果我们在初始化时，设置的初始化容量较小，HashMap 中 Node 的数量超过边界值，HashMap 就会调用 resize() 方法重新分配 table 数组。这将会导致 HashMap 的数组复制，迁移到另一块内存中去，从而影响 HashMap 的效率。

HashMap 添加元素优化

初始化完成后，HashMap 就可以使用 put() 方法添加键值对了。从下面源码可以看出，当程序将一个 key-value 对添加到 HashMap 中，程序首先会根据该 key 的 hashCode() 返回值，再通过 hash() 方法计算出 hash 值，再通过 putVal 方法中的 $(n - 1) \& hash$ 决定该 Node 的存储位置。

 复制代码

```
1 public V put(K key, V value) {  
2     return putVal(hash(key), key, value, false, true);  
3 }
```

 复制代码

```
1 static final int hash(Object key) {  
2     int h;  
3     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
4 }
```

 复制代码

```
1 if ((tab = table) == null || (n = tab.length) == 0)  
2     n = (tab = resize()).length;  
3 // 通过 putVal 方法中的 (n - 1) & hash 决定该 Node 的存储位置  
4 if ((p = tab[i = (n - 1) & hash]) == null)  
5     tab[i] = newNode(hash, key, value, null);  
6
```

如果你不太清楚 hash() 以及 $(n-1) \& hash$ 的算法，就请你看下面的详述。

我们先来了解下 hash() 方法中的算法。如果我们没有使用 hash() 方法计算 hashCode，而是直接使用对象的 hashCode 值，会出现什么问题呢？

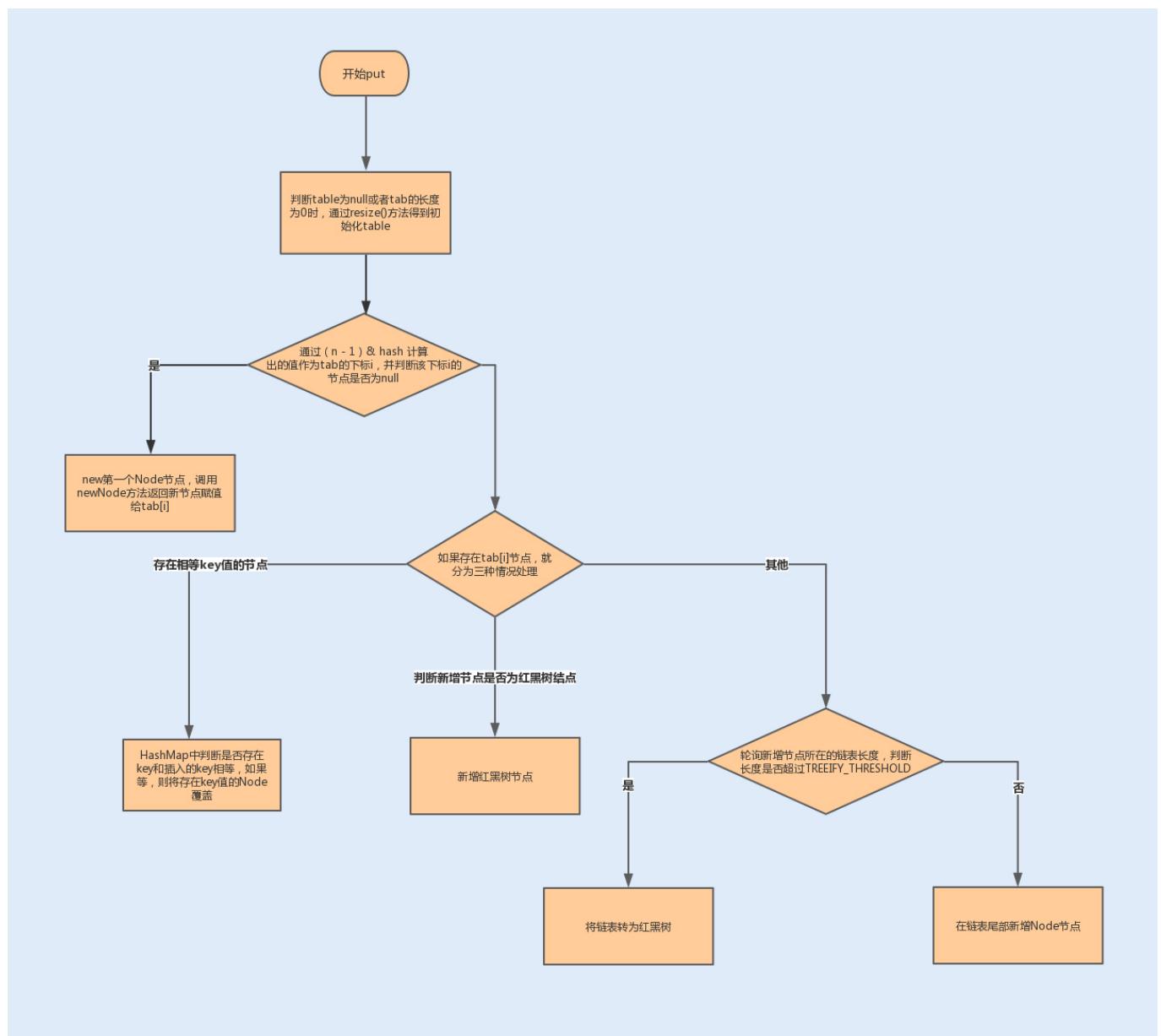
假设要添加两个对象 a 和 b，如果数组长度是 16，这时对象 a 和 b 通过公式 $(n - 1) \& hash$ 运算，也就是 $(16-1) \& a.hashCode$ 和 $(16-1) \& b.hashCode$ ，15 的二进制为 0000000000000000000000001111，假设对象 A 的 hashCode 为 1000010001110001000001111000000，对象 B 的 hashCode 为 0111011100111000101000010100000，你会发现上述与运算结果都是 0。这样的哈希结果就太让人失望了，很明显不是一个好的哈希算法。

但如果我们将 hashCode 值右移 16 位 ($h >>> 16$ 代表无符号右移 16 位)，也就是取 int 类型的一半，刚好可以将该二进制数对半切开，并且使用位异或运算（如果两个数对应的位置相反，则结果为 1，反之为 0），这样的话，就能避免上面的情况发生。这就是

hash() 方法的具体实现方式。简而言之，就是尽量打乱 hashCode 真正参与运算的低 16 位。

我再来解释下 $(n - 1) \& hash$ 是怎么设计的，这里的 n 代表哈希表的长度，哈希表习惯将长度设置为 2 的 n 次方，这样恰好可以保证 $(n - 1) \& hash$ 的计算得到的索引值总是位于 table 数组的索引之内。例如：hash=15，n=16 时，结果为 15；hash=17，n=16 时，结果为 1。

在获得 Node 的存储位置后，如果判断 Node 不在哈希表中，就新增一个 Node，并添加到哈希表中，整个流程我将用一张图来说明：



从图中我们可以看出：在 JDK1.8 中，HashMap 引入了红黑树数据结构来提升链表的查询效率。

这是因为链表的长度超过 8 后，红黑树的查询效率要比链表高，所以当链表超过 8 时，HashMap 就会将链表转换为红黑树，这里值得注意的一点是，这时的新增由于存在左旋、右旋效率会降低。讲到这里，我前面我提到的“因链表过长而导致的查询时间复杂度高”的问题，也就迎刃而解了。

以下就是 put 的实现源码：

 复制代码

```
1  final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
2                  boolean evict) {
3      Node<K,V>[] tab; Node<K,V> p; int n, i;
4      if ((tab = table) == null || (n = tab.length) == 0)
5 //1、判断当 table 为 null 或者 tab 的长度为 0 时，即 table 尚未初始化，此时通过 resize() 方法
6         n = (tab = resize()).length;
7         if ((p = tab[i = (n - 1) & hash]) == null)
8 //1.1、此处通过 (n - 1) & hash 计算出的值作为 tab 的下标 i，并另 p 表示 tab[i]，也就是该链表
9             tab[i] = newNode(hash, key, value, null);
10 //1.1.1、当 p 为 null 时，表明 tab[i] 上没有任何元素，那么接下来就 new 第一个 Node 节点，调用
11         else {
12 //2.1 下面进入 p 不为 null 的情况，有三种情况：p 为链表节点；p 为红黑树节点；p 是链表节点但长
13             Node<K,V> e; K k;
14             if (p.hash == hash &&
15                 ((k = p.key) == key || (key != null && key.equals(k))))
16 //2.1.1HashMap 中判断 key 相同的条件是 key 的 hash 相同，并且符合 equals 方法。这里判断了 p.
17
18             e = p;
19             else if (p instanceof TreeNode)
20 //2.1.2 现在开始了第一种情况，p 是红黑树节点，那么肯定插入后仍然是红黑树节点，所以我们直接强制转
21                 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
22             else {
23 //2.1.3 接下里就是 p 为链表节点的情形，也就是上述说的另外两类情况：插入后还是链表 / 插入后转红！
24                 for (int binCount = 0; ; ++binCount) {
25 // 我们需要一个计数器来计算当前链表的元素个数，并遍历链表，binCount 就是这个计数器
26
27                     if ((e = p.next) == null) {
28                         p.next = newNode(hash, key, value, null);
29                         if (binCount >= TREEIFY_THRESHOLD - 1)
30 // 插入成功后，要判断是否需要转换为红黑树，因为插入后链表长度加 1，而 binCount 并不包含新节点，
31                             treeifyBin(tab, hash);
32 // 当新长度满足转换条件时，调用 treeifyBin 方法，将该链表转换为红黑树
33                             break;
34                     }
35                     if (e.hash == hash &&
36                         ((k = e.key) == key || (key != null && key.equals(k))))
37                         break;
38                     p = e;
39                 }
40             }
```

```
41     if (e != null) { // existing mapping for key
42         V oldValue = e.value;
43         if (!onlyIfAbsent || oldValue == null)
44             e.value = value;
45         afterNodeAccess(e);
46         return oldValue;
47     }
48 }
49 ++modCount;
50 if (++size > threshold)
51     resize();
52 afterNodeInsertion(evict);
53 return null;
54 }
55
```



HashMap 获取元素优化

当 HashMap 中只存在数组，而数组中没有 Node 链表时，是 HashMap 查询数据性能最好的时候。一旦发生大量的哈希冲突，就会产生 Node 链表，这个时候每次查询元素都可能遍历 Node 链表，从而降低查询数据的性能。

特别是在链表长度过长的情况下，性能将明显降低，红黑树的使用很好地解决了这个问题，使得查询的平均复杂度降低到了 $O(\log(n))$ ，链表越长，使用黑红树替换后的查询效率提升就越明显。

我们在编码中也可以优化 HashMap 的性能，例如，重新 key 值的 hashCode() 方法，降低哈希冲突，从而减少链表的产生，高效利用哈希表，达到提高性能的效果。

HashMap 扩容优化

HashMap 也是数组类型的数据结构，所以一样存在扩容的情况。

在 JDK1.7 中，HashMap 整个扩容过程就是分别取出数组元素，一般该元素是最后一个放入链表中的元素，然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标，然后进行交换。这样的扩容方式会将原来哈希冲突的单向链表尾部变成扩容后单向链表的头部。

而在 JDK 1.8 中，HashMap 对扩容操作做了优化。由于扩容数组的长度是 2 倍关系，所以对于假设初始 `tableSize = 4` 要扩容到 8 来说就是 0100 到 1000 的变化（左移一位就是 2 倍），在扩容中只用判断原来的 hash 值和左移动的一位（`newTable` 的值）按位与操作是 0 或 1 就行，0 的话索引不变，1 的话索引变成原索引加上扩容前数组。

之所以能通过这种“与运算”来重新分配索引，是因为 hash 值本来就是随机的，而 hash 按位与上 `newTable` 得到的 0（扩容前的索引位置）和 1（扩容前索引位置加上扩容前数组长度的数值索引处）就是随机的，所以扩容的过程就能把之前哈希冲突的元素再随机分布到不同的索引中去。

总结

HashMap 通过哈希表数据结构的形式来存储键值对，这种设计的好处就是查询键值对的效率高。

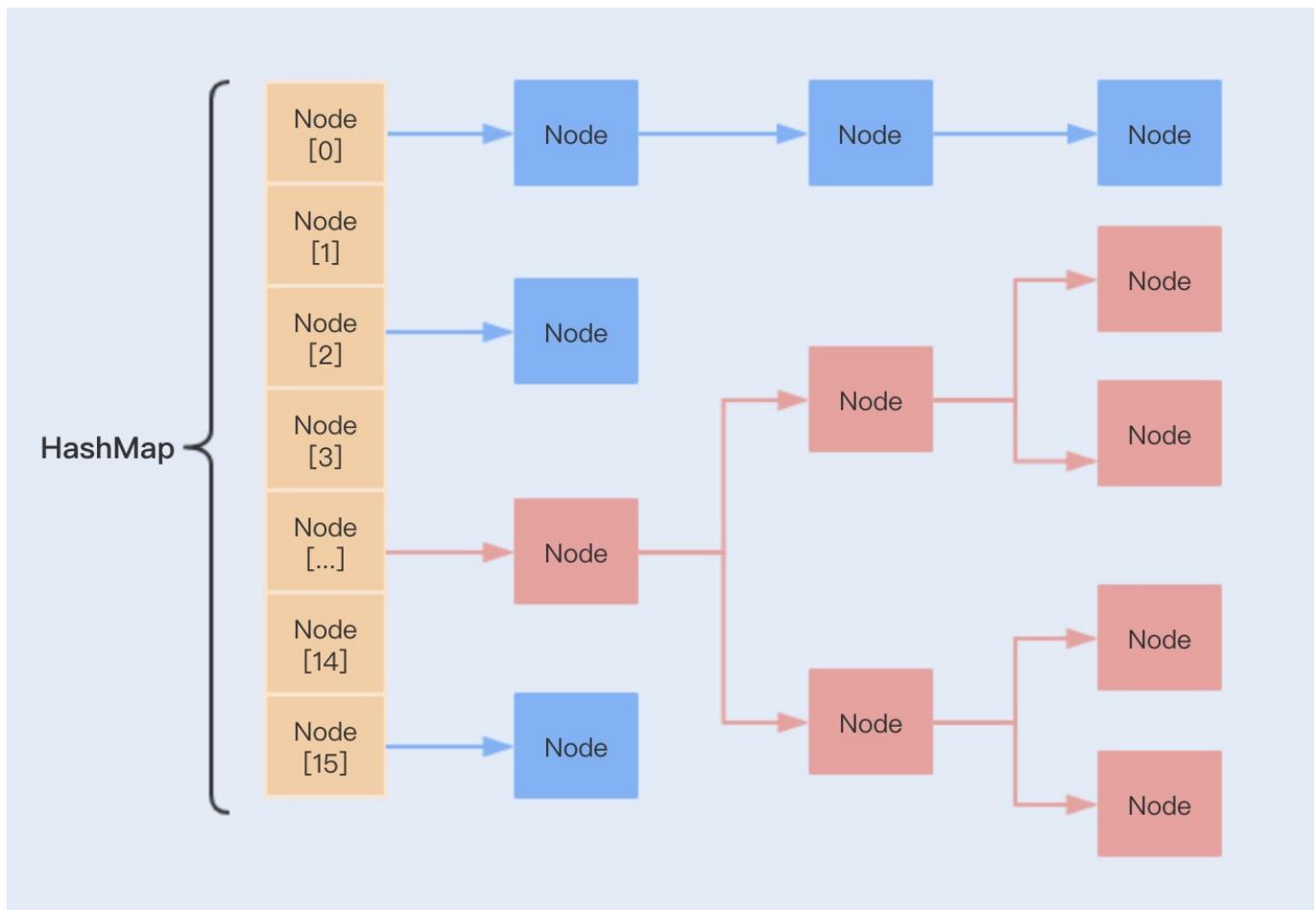
我们在使用 HashMap 时，可以结合自己的场景来设置初始容量和加载因子两个参数。当查询操作较为频繁时，我们可以适当地减少加载因子；如果对内存利用率要求比较高，我可以适当的增加加载因子。

我们还可以在预知存储数据量的情况下，提前设置初始容量（初始容量 = 预知数据量 / 加载因子）。这样做的好处是可以减少 `resize()` 操作，提高 HashMap 的效率。

HashMap 还使用了数组 + 链表这两种数据结构相结合的方式实现了链地址法，当有哈希值冲突时，就可以将冲突的键值对链成一个链表。

但这种方式又存在一个性能问题，如果链表过长，查询数据的时间复杂度就会增加。

HashMap 就在 Java8 中使用了红黑树来解决链表过长导致的查询性能下降问题。以下是 HashMap 的数据结构图：



08 | 网络通信优化之I/O模型：如何解决高并发下I/O瓶颈？



提到 Java I/O，相信你一定不陌生。你可能使用 I/O 操作读写文件，也可能使用它实现 Socket 的信息传输...这些都是我们在系统中最常遇到的和 I/O 有关的操作。

我们都知道，I/O 的速度要比内存速度慢，尤其是在现在这个大数据时代背景下，I/O 的性能问题更是尤为突出，I/O 读写已经成为很多应用场景下的系统性能瓶颈，不容我们忽视。

今天，我们就来深入了解下 Java I/O 在高并发、大数据业务场景下暴露出的性能问题，从源头入手，学习优化方法。

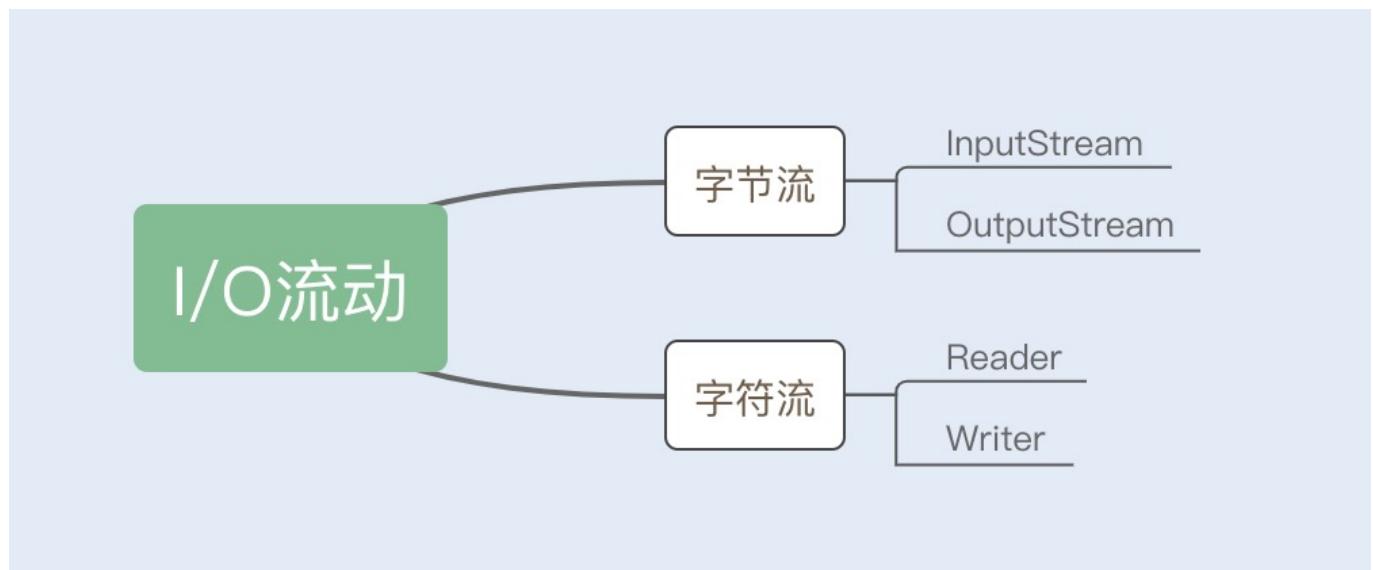
什么是 I/O

I/O 是机器获取和交换信息的主要渠道，而流是完成 I/O 操作的主要方式。

在计算机中，流是一种信息的转换。流是有序的，因此相对于某一机器或者应用程序而言，我们通常把机器或者应用程序接收外界的信息称为输入流（InputStream），从机器或者应用程序向外输出的信息称为输出流（OutputStream），合称为输入 / 输出流（I/O Streams）。

机器间或程序间在进行信息交换或者数据交换时，总是先将对象或数据转换为某种形式的流，再通过流的传输，到达指定机器或程序后，再将流转换为对象数据。因此，流就可以被看作是一种数据的载体，通过它可以实现数据交换和传输。

Java 的 I/O 操作类在包 java.io 下，其中 InputStream、OutputStream 以及 Reader、Writer 类是 I/O 包中的 4 个基本类，它们分别处理字节流和字符流。如下图所示：

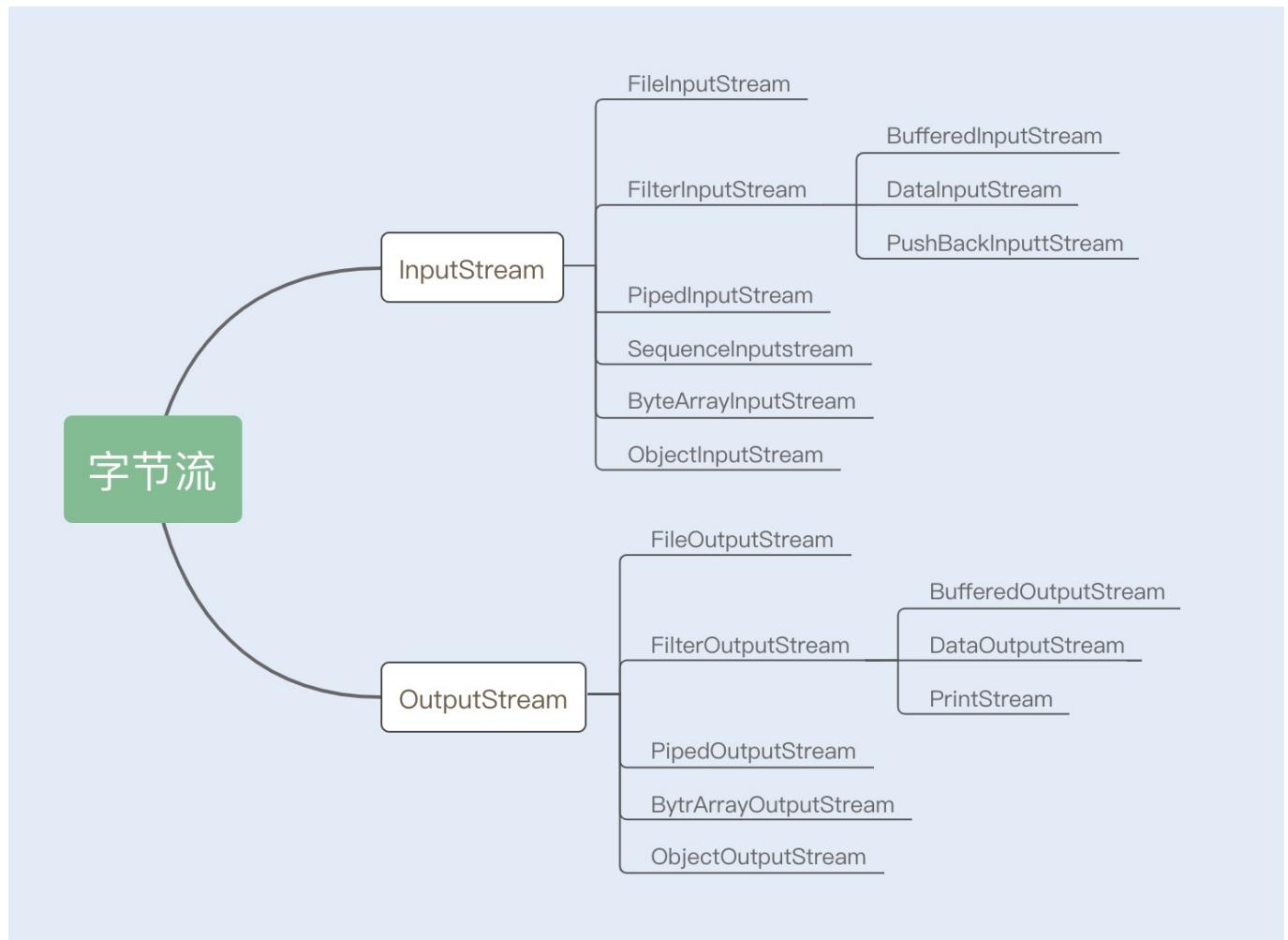


回顾我的经历，我记得在初次阅读 Java I/O 流文档的时候，我有过这样一个疑问，在这里也分享给你，那就是：“**不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？**”

我们知道字符到字节必须经过转码，这个过程非常耗时，如果我们不知道编码类型就很容易出现乱码问题。所以 I/O 流提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。下面我们就分别了解下“字节流”和“字符流”。

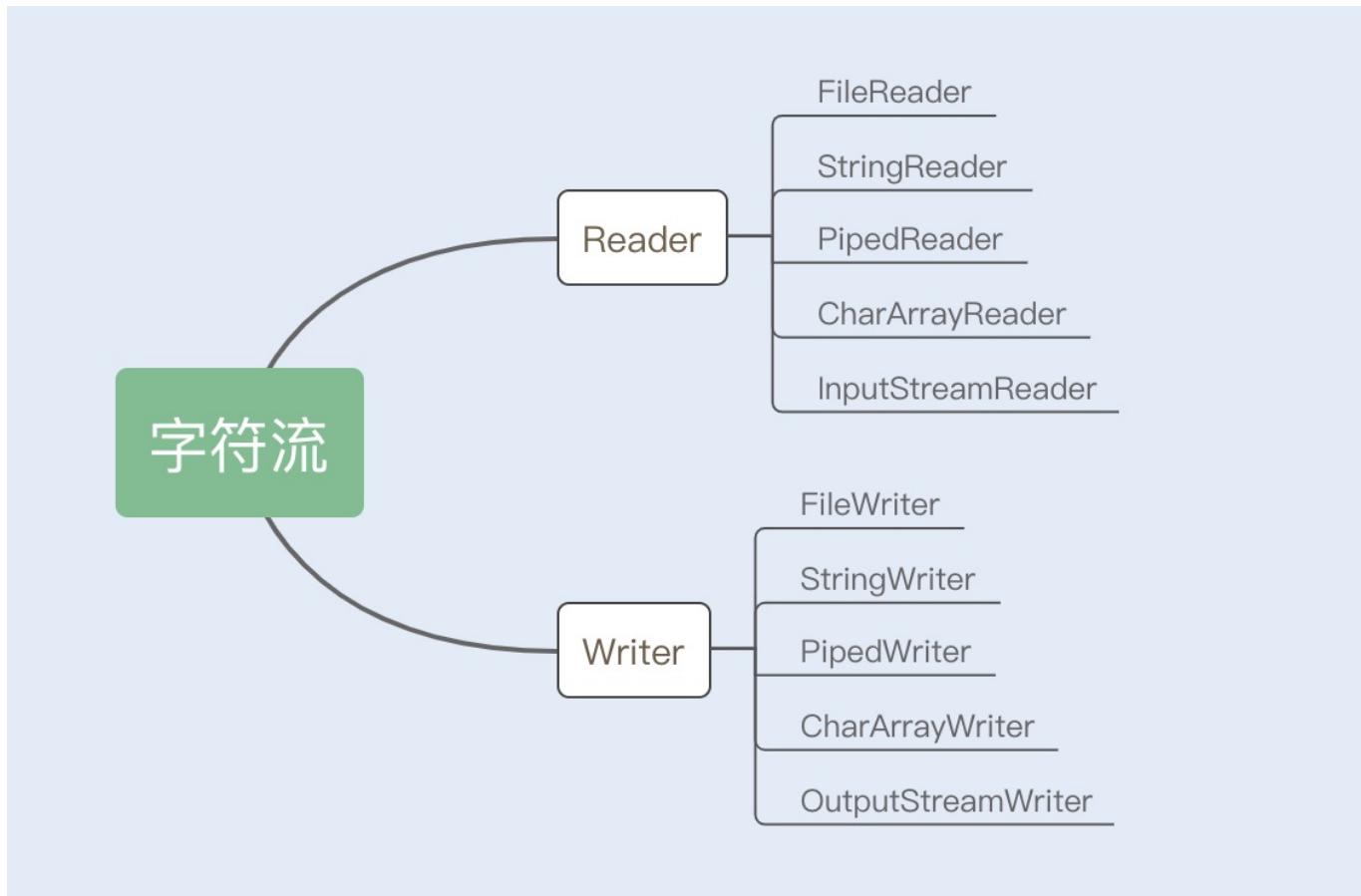
1. 字节流

InputStream/OutputStream 是字节流的抽象类，这两个抽象类又派生出了若干子类，不同的子类分别处理不同的操作类型。如果是文件的读写操作，就使用 FileInputStream/FileOutputStream；如果是数组的读写操作，就使用 ByteArrayInputStream/ByteArrayOutputStream；如果是普通字符串的读写操作，就使用 BufferedInputStream/BufferedOutputStream。具体内容如下图所示：



2. 字符流

Reader/Writer 是字符流的抽象类，这两个抽象类也派生出了若干子类，不同的子类分别处理不同的操作类型，具体内容如下图所示：

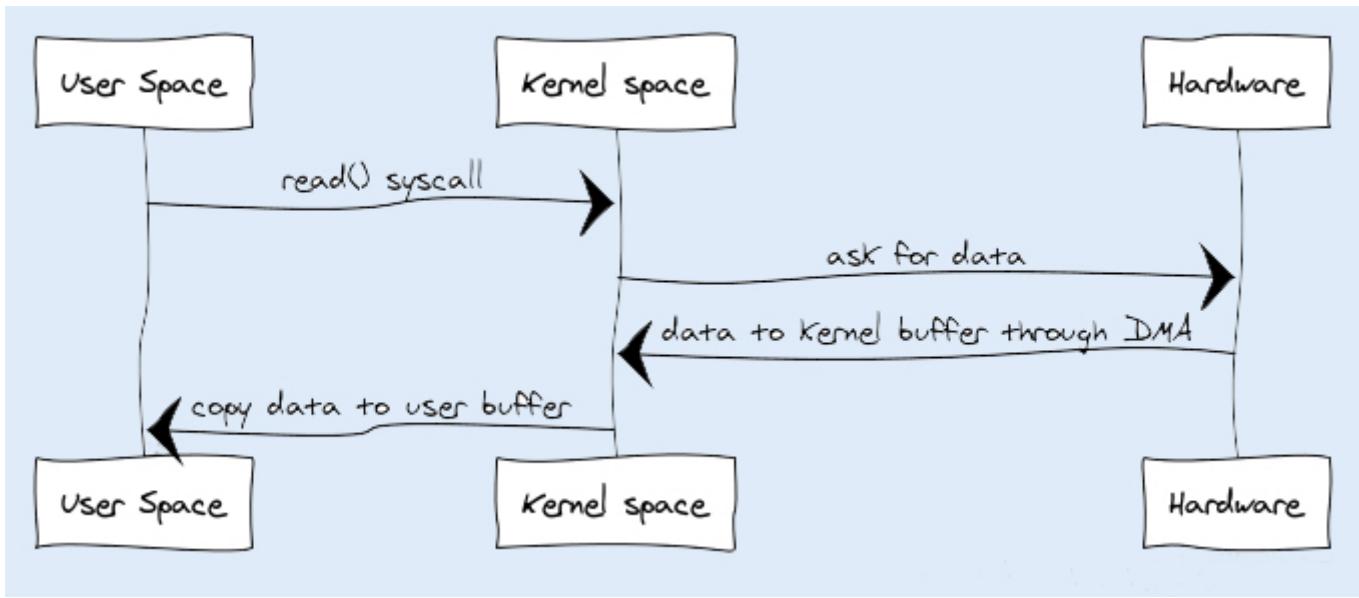


传统 I/O 的性能问题

我们知道，I/O 操作分为磁盘 I/O 操作和网络 I/O 操作。前者是从磁盘中读取数据源输入到内存中，之后将读取的信息持久化输出在物理磁盘上；后者是从网络中读取信息输入到内存，最终将信息输出到网络中。但不管是磁盘 I/O 还是网络 I/O，在传统 I/O 中都存在严重的性能问题。

1. 多次内存复制

在传统 I/O 中，我们可以通过 InputStream 从源数据中读取数据流输入到缓冲区里，通过 OutputStream 将数据输出到外部设备（包括磁盘、网络）。你可以先看下输入操作在操作系统中的具体流程，如下图所示：



JVM 会发出 `read()` 系统调用，并通过 `read` 系统调用向内核发起读请求；

内核向硬件发送读指令，并等待读就绪；

内核把将要读取的数据复制到指向的内核缓存中；

操作系统内核将数据复制到用户空间缓冲区，然后 `read` 系统调用返回。

在这个过程中，数据先从外部设备复制到内核空间，再从内核空间复制到用户空间，这就发生了两次内存复制操作。这种操作会导致不必要的数据拷贝和上下文切换，从而降低 I/O 的性能。

2. 阻塞

在传统 I/O 中，`InputStream` 的 `read()` 是一个 `while` 循环操作，它会一直等待数据读取，直到数据就绪才会返回。**这意味着如果没有数据就绪，这个读取操作将会一直被挂起，用户线程将会处于阻塞状态。**

在少量连接请求的情况下，使用这种方式没有问题，响应速度也很高。但在发生大量连接请求时，就需要创建大量监听线程，这时如果线程没有数据就绪就会被挂起，然后进入阻塞状态。一旦发生线程阻塞，这些线程将会不断地抢夺 CPU 资源，从而导致大量的 CPU 上下文切换，增加系统的性能开销。

如何优化 I/O 操作

面对以上两个性能问题，不仅编程语言对此做了优化，各个操作系统也进一步优化了 I/O。JDK1.4 发布了 `java.nio` 包（新 I/O 的缩写），NIO 的发布优化了内存复制以及阻塞导

致的严重性能问题。JDK1.7 又发布了 NIO2，提出了从操作系统层面实现的异步 I/O。下面我们就来了解下具体的优化实现。

1. 使用缓冲区优化读写流操作

在传统 I/O 中，提供了基于流的 I/O 实现，即 `InputStream` 和 `OutputStream`，这种基于流的实现以字节为单位处理数据。

NIO 与传统 I/O 不同，它是基于块（Block）的，它以块为基本单位处理数据。在 NIO 中，最为重要的两个组件是缓冲区（Buffer）和通道（Channel）。Buffer 是一块连续的内存块，是 NIO 读写数据的中转地。Channel 表示缓冲数据的源头或者目的地，它用于读取缓冲或者写入数据，是访问缓冲的接口。

传统 I/O 和 NIO 的最大区别就是传统 I/O 是面向流，NIO 是面向 Buffer。Buffer 可以将文件一次性读入内存再做后续处理，而传统的方式是边读文件边处理数据。虽然传统 I/O 后面也使用了缓冲块，例如 `BufferedInputStream`，但仍然不能和 NIO 相媲美。使用 NIO 替代传统 I/O 操作，可以提升系统的整体性能，效果立竿见影。

2. 使用 DirectBuffer 减少内存复制

NIO 的 Buffer 除了做了缓冲块优化之外，还提供了一个可以直接访问物理内存的类 `DirectBuffer`。普通的 Buffer 分配的是 JVM 堆内存，而 `DirectBuffer` 是直接分配物理内存。

我们知道数据要输出到外部设备，必须先从用户空间复制到内核空间，再复制到输出设备，而 `DirectBuffer` 则是直接将步骤简化为从内核空间复制到外部设备，减少了数据拷贝。

这里拓展一点，由于 `DirectBuffer` 申请的是非 JVM 的物理内存，所以创建和销毁的代价很高。`DirectBuffer` 申请的内存并不是直接由 JVM 负责垃圾回收，但在 `DirectBuffer` 包装类被回收时，会通过 Java Reference 机制来释放该内存块。

3. 避免阻塞，优化 I/O 操作

NIO 很多人也称之为 Non-block I/O，即非阻塞 I/O，因为这样叫，更能体现它的特点。为什么这么说呢？

传统的 I/O 即使使用了缓冲块，依然存在阻塞问题。由于线程池线程数量有限，一旦发生大量并发请求，超过最大数量的线程就只能等待，直到线程池中有空闲的线程可以被复用。而对 Socket 的输入流进行读取时，读取流会一直阻塞，直到发生以下三种情况的任意一种才会解除阻塞：

有数据可读；

连接释放；

空指针或 I/O 异常。

阻塞问题，就是传统 I/O 最大的弊端。NIO 发布后，通道和多路复用器这两个基本组件实现了 NIO 的非阻塞，下面我们就一起来了解下这两个组件的优化原理。

通道 (Channel)

前面我们讨论过，传统 I/O 的数据读取和写入是从用户空间到内核空间来回复制，而内核空间的数据是通过操作系统层面的 I/O 接口从磁盘读取或写入。

最开始，在应用程序调用操作系统 I/O 接口时，是由 CPU 完成分配，这种方式最大的问题是“发生大量 I/O 请求时，非常消耗 CPU”；之后，操作系统引入了 DMA（直接存储器存储），内核空间与磁盘之间的存取完全由 DMA 负责，但这种方式依然需要向 CPU 申请权限，且需要借助 DMA 总线来完成数据的复制操作，如果 DMA 总线过多，就会造成总线冲突。

通道的出现解决了以上问题，Channel 有自己的处理器，可以完成内核空间和磁盘之间的 I/O 操作。在 NIO 中，我们读取和写入数据都要通过 Channel，由于 Channel 是双向的，所以读、写可以同时进行。

多路复用器 (Selector)

Selector 是 Java NIO 编程的基础。用于检查一个或多个 NIO Channel 的状态是否处于可读、可写。

Selector 是基于事件驱动实现的，我们可以在 Selector 中注册 accept、read 监听事件，Selector 会不断轮询注册在其上的 Channel，如果某个 Channel 上面发生监听事件，这个 Channel 就处于就绪状态，然后进行 I/O 操作。

一个线程使用一个 Selector，通过轮询的方式，可以监听多个 Channel 上的事件。我们可以在注册 Channel 时设置该通道为非阻塞，当 Channel 上没有 I/O 操作时，该线程就不会一直等待了，而是会不断轮询所有 Channel，从而避免发生阻塞。

目前操作系统的 I/O 多路复用机制都使用了 epoll，相比传统的 select 机制，epoll 没有最大连接句柄 1024 的限制。所以 Selector 在理论上可以轮询成千上万的客户端。

下面我用一个生活化的场景来举例，看完你就更清楚 Channel 和 Selector 在非阻塞 I/O 中承担什么角色，发挥什么作用了。

我们可以把监听多个 I/O 连接请求比作一个火车站的进站口。以前检票只能让搭乘就近一趟发车的旅客提前进站，而且只有一个检票员，这时如果有其他车次的旅客要进站，就只能在站口排队。这就相当于最早没有实现线程池的 I/O 操作。

后来火车站升级了，多了几个检票入口，允许不同车次的旅客从各自对应的检票入口进站。这就相当于用多线程创建了多个监听线程，同时监听各个客户端的 I/O 请求。

最后火车站进行了升级改造，可以容纳更多旅客了，每个车次载客更多了，而且车次也安排合理，乘客不再扎堆排队，可以从一个大的统一的检票口进站了，这一个检票口可以同时检票多个车次。这个大的检票口就相当于 Selector，车次就相当于 Channel，旅客就相当于 I/O 流。

总结

Java 的传统 I/O 开始是基于 InputStream 和 OutputStream 两个操作流实现的，这种流操作是以字节为单位，如果在高并发、大数据场景中，很容易导致阻塞，因此这种操作的性能是非常差的。还有，输出数据从用户空间复制到内核空间，再复制到输出设备，这样的操作会增加系统的性能开销。

传统 I/O 后来使用了 Buffer 优化了“阻塞”这个性能问题，以缓冲块作为最小单位，但相比整体性能来说依然不尽人意。

于是 NIO 出现，它是基于缓冲块为单位的流操作，在 Buffer 的基础上，新增了两个组件“管道和多路复用器”，实现了非阻塞 I/O，NIO 适用于发生大量 I/O 连接请求的场景，这三个组件共同提升了 I/O 的整体性能。

09 | 网络通信优化之序列化：避免使用Java序列化



当前大部分后端服务都是基于微服务架构实现的。服务按照业务划分被拆分，实现了服务的解偶，但同时也带来了新的问题，不同业务之间通信需要通过接口实现调用。两个服务之间要共享一个数据对象，就需要从对象转换成二进制流，通过网络传输，传送到对方服务，再转换回对象，供服务方法调用。这个编码和解码过程我们称之为序列化与反序列化。

在大量并发请求的情况下，如果序列化的速度慢，会导致请求响应时间增加；而序列化后的传输数据体积大，会导致网络吞吐量下降。所以一个优秀的序列化框架可以提高系统的整体性能。

我们知道，Java 提供了 RMI 框架可以实现服务与服务之间的接口暴露和调用，RMI 中对数据对象的序列化采用的是 Java 序列化。而目前主流的微服务框架却几乎没有用到 Java 序列化，SpringCloud 用的是 Json 序列化，Dubbo 虽然兼容了 Java 序列化，但默认使用的是 Hessian 序列化。这是为什么呢？

今天我们就来深入了解下 Java 序列化，再对比近两年比较火的 Protobuf 序列化，看看 Protobuf 是如何实现最优序列化的。

Java 序列化

在说缺陷之前，你先得知道什么是 Java 序列化以及它的实现原理。

Java 提提供了一种序列化机制，这种机制能够将一个对象序列化为二进制形式（字节数组），用于写入磁盘或输出到网络，同时也能从网络或磁盘中读取字节数组，反序列化成对象，在程序中使用。



JDK 提供的两个输入、输出流对象 `ObjectInputStream` 和 `ObjectOutputStream`，它们只能对实现了 `Serializable` 接口的类的对象进行反序列化和序列化。

`ObjectOutputStream` 的默认序列化方式，仅对对象的非 `transient` 的实例变量进行序列化，而不会序列化对象的 `transient` 的实例变量，也不会序列化静态变量。

在实现了 `Serializable` 接口的类的对象中，会生成一个 `serialVersionUID` 的版本号，这个版本号有什么用呢？它会在反序列化过程中来验证序列化对象是否加载了反序列化的类，如果是具有相同类名的不同版本号的类，在反序列化中是无法获取对象的。

具体实现序列化的是 `writeObject` 和 `readObject`，通常这两个方法是默认的，当然我们也可以在实现 `Serializable` 接口的类中对其进行重写，定制一套属于自己的序列化与反序列化机制。

另外，Java 序列化的类中还定义了两个重写方法：`writeReplace()` 和 `readResolve()`，前者是用来在序列化之前替换序列化对象的，后者是用来在反序列化之后对返回对象进行处理的。

Java 序列化的缺陷

如果你用过一些 RPC 通信框架，你就会发现这些框架很少使用 JDK 提供的序列化。其实不用和不好用多半是挂钩的，下面我们就一起来看看 JDK 默认的序列化到底存在着哪些缺陷。

1. 无法跨语言

现在的系统设计越来越多元化，很多系统都使用了多种语言来编写应用程序。比如，我们公司开发的一些大型游戏就使用了多种语言，C++ 写游戏服务，Java/Go 写周边服务，Python 写一些监控应用。

而 Java 序列化目前只适用基于 Java 语言实现的框架，其它语言大部分都没有使用 Java 的序列化框架，也没有实现 Java 序列化这套协议。因此，如果是两个基于不同语言编写的应用程序相互通信，则无法实现两个应用服务之间传输对象的序列化与反序列化。

2. 易被攻击

Java 官网安全编码指导方针中说明：“对不信任数据的反序列化，从本质上来说是危险的，应该予以避免”。可见 Java 序列化是不安全的。

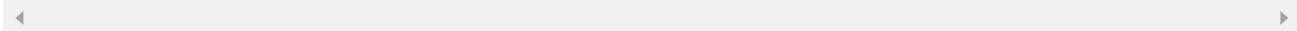
我们知道对象是通过在 `ObjectInputStream` 上调用 `readObject()` 方法进行反序列化的，这个方法其实是一个神奇的构造器，它可以将类路径上几乎所有实现了 `Serializable` 接口的对象都实例化。

这也就意味着，在反序列化字节流的过程中，该方法可以执行任意类型的代码，这是非常危险的。

对于需要长时间进行反序列化的对象，不需要执行任何代码，也可以发起一次攻击。攻击者可以创建循环对象链，然后将序列化后的对象传输到程序中反序列化，这种情况会导致 `hashCode` 方法被调用次数呈次方爆发式增长，从而引发栈溢出异常。例如下面这个案例就可以很好地说明。

 复制代码

```
1 Set root = new HashSet();
2 Set s1 = root;
3 Set s2 = new HashSet();
4 for (int i = 0; i < 100; i++) {
5     Set t1 = new HashSet();
6     Set t2 = new HashSet();
7     t1.add("foo"); // 使 t2 不等于 t1
8     s1.add(t1);
9     s1.add(t2);
10    s2.add(t1);
11    s2.add(t2);
12    s1 = t1;
13    s2 = t2;
14 }
```



2015 年 FoxGlove Security 安全团队的 breenmachine 发布过一篇长博客，主要内容是：通过 Apache Commons Collections，Java 反序列化漏洞可以实现攻击。一度横扫了 WebLogic、WebSphere、JBoss、Jenkins、OpenNMS 的最新版，各大 Java Web Server 纷纷躺枪。

其实，Apache Commons Collections 就是一个第三方基础库，它扩展了 Java 标准库里 的 Collection 结构，提供了很多强有力的数据结构类型，并且实现了各种集合工具类。

实现攻击的原理就是：Apache Commons Collections 允许链式的任意的类函数反射调用，攻击者通过“实现了 Java 序列化协议”的端口，把攻击代码上传到服务器上，再由 Apache Commons Collections 里的 TransformedMap 来执行。

那么后来是如何解决这个漏洞的呢？

很多序列化协议都制定了一套数据结构来保存和获取对象。例如，JSON 序列化、ProtocolBuf 等，它们只支持一些基本类型和数组数据类型，这样可以避免反序列化创建一些不确定的实例。虽然它们的设计简单，但足以满足当前大部分系统的数据传输需求。

我们也可以通过反序列化对象白名单来控制反序列化对象，可以重写 resolveClass 方法，并在该方法中校验对象名字。代码如下所示：

 复制代码

```
1 @Override
2 protected Class resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {
3     if (!desc.getName().equals(Bicycle.class.getName())) {
4
5         throw new InvalidClassException(
6             "Unauthorized deserialization attempt", desc.getName());
7     }
8     return super.resolveClass(desc);
9 }
```



3. 序列化后的流太大

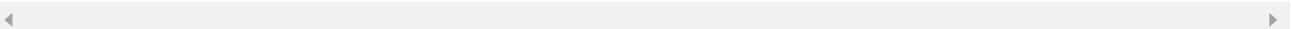
序列化后的二进制流大小能体现序列化的性能。序列化后的二进制数组越大，占用的存储空间就越多，存储硬件的成本就越高。如果我们是进行网络传输，则占用的带宽就更多，这时就会影响到系统的吞吐量。

Java 序列化中使用了 `ObjectOutputStream` 来实现对象转二进制编码，那么这种序列化机制实现的二进制编码完成的二进制数组大小，相比于 NIO 中的 `ByteBuffer` 实现的二进制编码完成的数组大小，有没有区别呢？

我们可以通过一个简单的例子来验证下：

复制代码

```
1 User user = new User();
2         user.setUserName("test");
3         user.setPassword("test");
4
5         ByteArrayOutputStream os =new ByteArrayOutputStream();
6         ObjectOutputStream out = new ObjectOutputStream(os);
7         out.writeObject(user);
8
9         byte[] testByte = os.toByteArray();
10        System.out.print("ObjectOutputStream 字节编码长度: " + testByte.length + "\n");
```



复制代码

```
1 ByteBuffer byteBuffer = ByteBuffer.allocate( 2048);
2
3         byte[] userName = user.getUserName().getBytes();
4         byte[] password = user.getPassword().getBytes();
5         byteBuffer.putInt(userName.length);
```

```
6     byteBuffer.put(userName);
7     byteBuffer.putInt(password.length);
8     byteBuffer.put(password);
9
10    byteBuffer.flip();
11    byte[] bytes = new byte[byteBuffer.remaining()];
12    System.out.print("ByteBuffer 字节编码长度: " + bytes.length+ "\n");
13
```

 复制代码

```
1 ObjectOutputStream 字节编码长度: 99
2 ByteBuffer 字节编码长度: 16
```

这里我们可以清楚地看到：Java 序列化实现的二进制编码完成的二进制数组大小，比 ByteBuffer 实现的二进制编码完成的二进制数组大小要大上几倍。因此，Java 序列化的流会变大，最终会影响到系统的吞吐量。

4. 序列化性能太差

序列化的速度也是体现序列化性能的重要指标，如果序列化的速度慢，就会影响网络通信的效率，从而增加系统的响应时间。我们再来通过上面这个例子，来对比下 Java 序列化与 NIO 中的 ByteBuffer 编码的性能：

 复制代码

```
1     User user = new User();
2     user.setUserName("test");
3     user.setPassword("test");
4
5     long startTime = System.currentTimeMillis();
6
7     for(int i=0; i<1000; i++) {
8         ByteArrayOutputStream os =new ByteArrayOutputStream();
9         ObjectOutputStream out = new ObjectOutputStream(os);
10        out.writeObject(user);
11        out.flush();
12        out.close();
13        byte[] testByte = os.toByteArray();
14        os.close();
```

```
15     }
16
17
18     long endTime = System.currentTimeMillis();
19     System.out.print("ObjectOutputStream 序列化时间: " + (endTime - startTime) + "\n"
```

 复制代码

```
1 long startTime1 = System.currentTimeMillis();
2         for(int i=0; i<1000; i++) {
3             ByteBuffer byteBuffer = ByteBuffer.allocate( 2048);
4
5             byte[] userName = user.getUserName().getBytes();
6             byte[] password = user.getPassword().getBytes();
7             byteBuffer.putInt(userName.length);
8             byteBuffer.put(userName);
9             byteBuffer.putInt(password.length);
10            byteBuffer.put(password);
11
12            byteBuffer.flip();
13            byte[] bytes = new byte[byteBuffer.remaining()];
14        }
15        long endTime1 = System.currentTimeMillis();
16        System.out.print("ByteBuffer 序列化时间: " + (endTime1 - startTime1)+ "\n");
```

 复制代码

```
1 ObjectOutputStream 序列化时间: 29
2 ByteBuffer 序列化时间: 6
```

通过以上案例，我们可以清楚地看到：Java 序列化中的编码耗时要比 ByteBuffer 长很多。

使用 Protobuf 序列化替换 Java 序列化

目前业内优秀的序列化框架有很多，而且大部分都避免了 Java 默认序列化的一些缺陷。例如，最近几年比较流行的 FastJson、Kryo、Protobuf、Hessian 等。我们完全可以找一种替换掉 Java 序列化，这里我推荐使用 Protobuf 序列化框架。

Protobuf 是由 Google 推出且支持多语言的序列化框架，目前在主流网站上的序列化框架性能对比测试报告中，Protobuf 无论是编解码耗时，还是二进制流压缩大小，都名列前茅。

Protobuf 以一个 .proto 后缀的文件为基础，这个文件描述了字段以及字段类型，通过工具可以生成不同语言的数据结构文件。在序列化该数据对象的时候，Protobuf 通过.proto 文件描述来生成 Protocol Buffers 格式的编码。

这里拓展一点，我来讲下什么是 Protocol Buffers 存储格式以及它的实现原理。

Protocol Buffers 是一种轻便高效的结构化数据存储格式。它使用 T-L-V (标识 - 长度 - 字段值) 的数据格式来存储数据，T 代表字段的正数序列 (tag) ，Protocol Buffers 将对象中的每个字段和正数序列对应起来，对应关系的信息是由生成的代码来保证的。在序列化的时候用整数值来代替字段名称，于是传输流量就可以大幅缩减；L 代表 Value 的字节长度，一般也只占一个字节；V 则代表字段值经过编码后的值。这种数据格式不需要分隔符，也不需要空格，同时减少了冗余字段名。

Protobuf 定义了一套自己的编码方式，几乎可以映射 Java/Python 等语言的所有基础数据类型。不同的编码方式对应不同的数据类型，还能采用不同的存储格式。如下图所示：

Wire Type	编码方式	编码长度	存储方式	代表的数据类型
0	Varint(负数时以 Zigzag 辅助编码)	变长(1-10个字节)	T-V	int32, int64, uint32, uint64, bool, enum sint32, sint64(负数时使用)
1	64-bit	固定8个字节		fixed64, sfixed64, double
2	Length-delimited	变长	T-L-V	string, bytes, embedded, messages, packed repeated fields
3	32-bit	固定4个字节	T-V	fixed32, sfixed32, float

对于存储 Varint 编码数据，由于数据占用的存储空间是固定的，就不需要存储字节长度 Length，所以实际上 Protocol Buffers 的存储方式是 T - V，这样就又减少了一个字节的存储空间。

Protobuf 定义的 Varint 编码方式是一种变长的编码方式，每个数据类型一个字节的最后一一位是一个标志位 (msb)，用 0 和 1 来表示，0 表示当前字节已经是最后一个字节，1 表示这个数字后面还有一个字节。

对于 int32 类型数字，一般需要 4 个字节表示，若采用 Varint 编码方式，对于很小的 int32 类型数字，就可以用 1 个字节来表示。对于大部分整数类型数据来说，一般都是小于 256，所以这种操作可以起到很好地压缩数据的效果。

我们知道 int32 代表正负数，所以一般最后一位是用来表示正负值，现在 Varint 编码方式将最后一位用作了标志位，那还如何去表示正负整数呢？如果使用 int32/int64 表示负数就需要多个字节来表示，在 Varint 编码类型中，通过 Zigzag 编码进行转换，将负数转换成无符号数，再采用 sint32/sint64 来表示负数，这样就可以大大地减少编码后的字节数。

Protobuf 的这种数据存储格式，不仅压缩存储数据的效果好，在编码和解码的性能方面也很高效。Protobuf 的编码和解码过程结合.proto 文件格式，加上 Protocol Buffer 独特的编码格式，只需要简单的数据运算以及位移等操作就可以完成编码与解码。可以说 Protobuf 的整体性能非常优秀。

总结

无论是网路传输还是磁盘持久化数据，我们都需要将数据编码成字节码，而我们平时在程序中使用的数据都是基于内存的数据类型或者对象，**我们需要通过编码将这些数据转化成二进制字节流**；如果需要接收或者再使用时，又需要通过解码将二进制字节流转换成内存数据。我们通常将这两个过程称为序列化与反序列化。

Java 默认的序列化是通过 Serializable 接口实现的，只要类实现了该接口，同时生成一个默认的版本号，我们无需手动设置，该类就会自动实现序列化与反序列化。

Java 默认的序列化虽然实现方便，但却存在安全漏洞、不跨语言以及性能差等缺陷，所以我强烈建议你避免使用 Java 序列化。

纵观主流序列化框架，FastJson、Protobuf、Kryo 是比较有特点的，而且性能以及安全方面都得到了业界的认可，我们可以结合自身业务来选择一种适合的序列化框架，来优化系统的序列化性能。

10 | 网络通信优化之通信协议：如何优化RPC网络通信？



上一讲中，我提到了微服务框架，其中 SpringCloud 和 Dubbo 的使用最为广泛，行业内也一直存在着对两者的比较，很多技术人会为这两个框架哪个更好而争辩。

我记得我们部门在搭建微服务框架时，也在技术选型上纠结良久，还曾一度有过激烈的讨论。当前 SpringCloud 炙手可热，具备完整的微服务生态，得到了很多同事的票选，但我们最终的选择却是 Dubbo，这是为什么呢？

RPC 通信是大型服务框架的核心

我们经常讨论微服务，首要应该了解的就是微服务的核心到底是什么，这样我们在做技术选型时，才能更准确地把握需求。

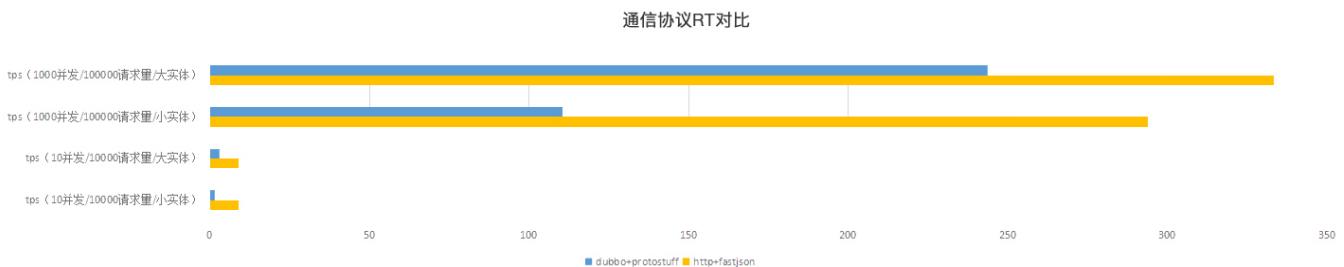
就我个人理解，**我认为微服务的核心是远程通信和服务治理**。远程通信提供了服务之间通信的桥梁，服务治理则提供了服务的后勤保障。所以，我们在做技术选型时，更多要考虑的是这两个核心的需求。

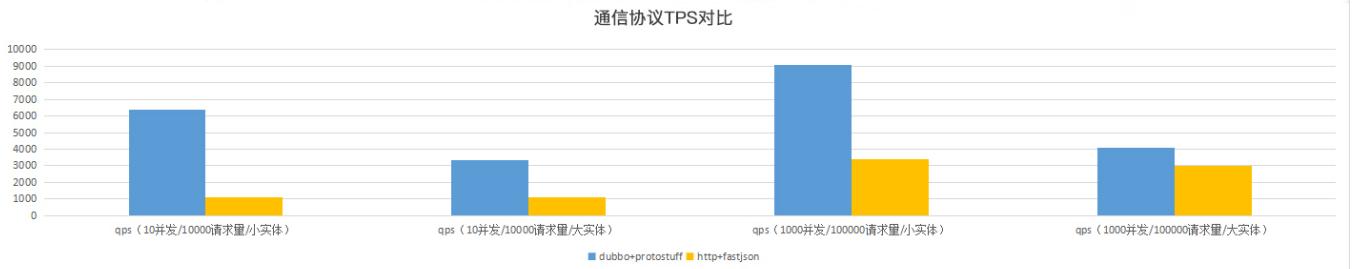
我们知道服务的拆分增加了通信的成本，特别是在一些抢购或者促销的业务场景中，如果服务之间存在方法调用，比如，抢购成功之后需要调用订单系统、支付系统、券包系统等，这种远程通信就很容易成为系统的瓶颈。所以，在满足一定的服务治理需求的前提下，对远程通信的性能需求就是技术选型的主要影响因素。

目前，很多微服务框架中的服务通信是基于 RPC 通信实现的，在没有进行组件扩展的前提下，**SpringCloud 是基于 Feign 组件实现的 RPC 通信（基于 Http+Json 序列化实现）**，**Dubbo 是基于 SPI 扩展了很多 RPC 通信框架**，包括 RMI、Dubbo、Hessian 等 RPC 通信框架（默认是 Dubbo+Hessian 序列化）。不同的业务场景下，RPC 通信的选择和优化标准也不同。

例如，开头我提到的我们部门在选择微服务框架时，选择了 Dubbo。当时的选择标准就是 RPC 通信可以支持抢购类的高并发，在这个业务场景中，请求的特点是瞬时高峰、请求量大和传入、传出参数数据包较小。而 Dubbo 中的 Dubbo 协议就很好地支持了这个请求。

以下是基于 Dubbo:2.6.4 版本进行的简单的性能测试。分别测试 Dubbo+Protobuf 序列化以及 Http+Json 序列化的通信性能（这里主要模拟单一 TCP 长连接 +Protobuf 序列化和短连接的 Http+Json 序列化的性能对比）。为了验证在数据量不同的情况下二者的性能表现，我分别准备了小对象和大对象的性能压测，通过这样的方式我们也可以间接地了解下二者在 RPC 通信方面的水平。





这个测试是我之前的积累，基于测试环境比较复杂，这里我就直接给出结果了，如果你感兴趣的话，可以留言和我讨论。

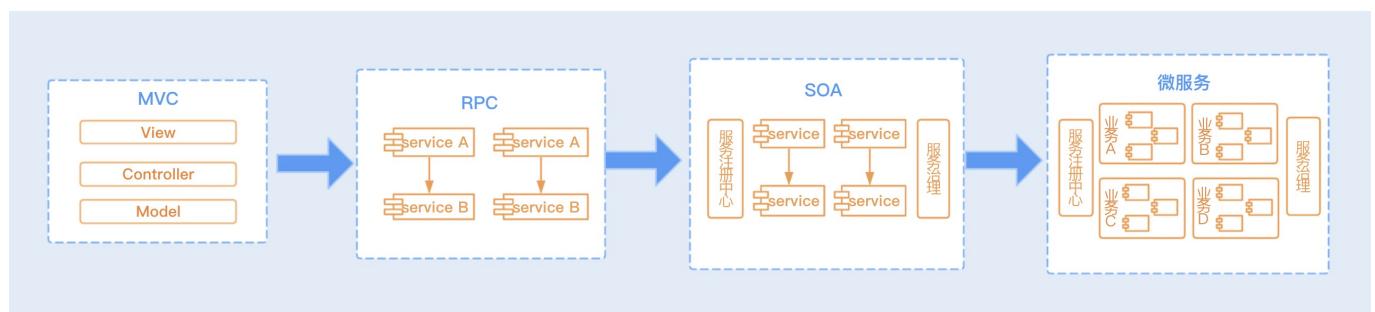
通过以上测试结果可以发现：**无论从响应时间还是吞吐量上来看，单一 TCP 长连接 + Protobuf 序列化实现的 RPC 通信框架都有着非常明显的优势。**

在高并发场景下，我们选择后端服务框架或者中间件部门自行设计服务框架时，RPC 通信是重点优化的对象。

其实，目前成熟的 RPC 通信框架非常多，如果你们公司没有自己的中间件团队，也可以基于开源的 RPC 通信框架做扩展。在正式进行优化之前，我们不妨简单回顾下 RPC。

什么是 RPC 通信

一提到 RPC，你是否还想到 MVC、SOA 这些概念呢？如果你没有经历过这些架构的演变，这些概念就很容易混淆。**你可以通过下面这张图来了解下这些架构的演变史。**



无论是微服务、SOA、还是 RPC 架构，它们都是分布式服务架构，都需要实现服务之间的互相通信，我们通常把这种通信统称为 RPC 通信。

RPC (Remote Process Call)，即远程服务调用，是通过网络请求远程计算机程序服务的通信技术。RPC 框架封装好了底层网络通信、序列化等技术，我们只需要在项目中引入各个服务的接口包，就可以实现在代码中调用 RPC 服务同调用本地方法一样。正因为这种方

便、透明的远程调用，RPC 被广泛应用于当下企业级以及互联网项目中，是实现分布式系统的核心。

RMI (Remote Method Invocation) 是 JDK 中最先实现了 RPC 通信的框架之一，RMI 的实现对建立分布式 Java 应用程序至关重要，是 Java 体系非常重要的底层技术，很多开源的 RPC 通信框架也是基于 RMI 实现原理设计出来的，包括 Dubbo 框架中也接入了 RMI 框架。接下来我们就一起了解下 RMI 的实现原理，看看它存在哪些性能瓶颈有待优化。

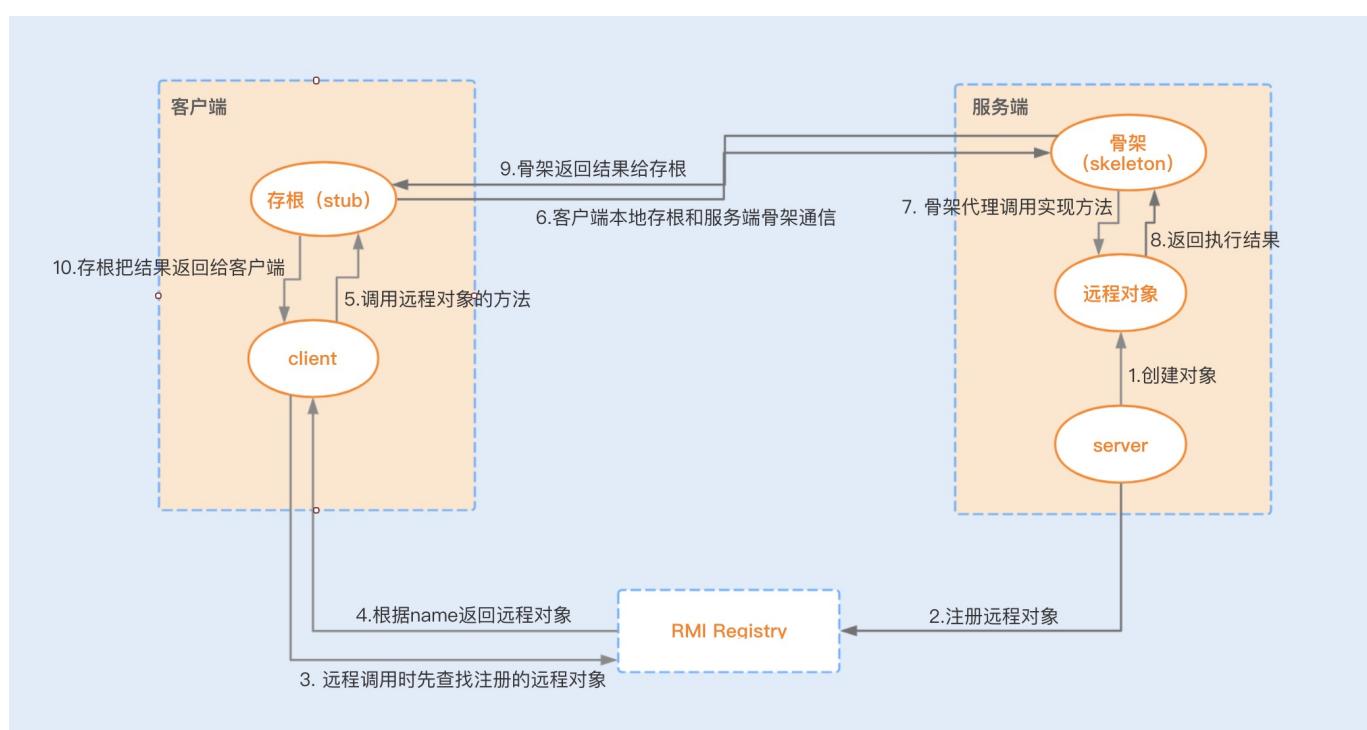
RMI : JDK 自带的 RPC 通信框架

目前 RMI 已经很成熟地应用在了 EJB 以及 Spring 框架中，是纯 Java 网络分布式应用系统的核心解决方案。RMI 实现了一台虚拟机应用对远程方法的调用可以同对本地方法的调用一样，RMI 帮我们封装好了其中关于远程通信的内容。

RMI 的实现原理

RMI 远程代理对象是 RMI 中最核心的组件，除了对象本身所在的虚拟机，其它虚拟机也可以调用此对象的方法。而且这些虚拟机可以不在同一个主机上，通过远程代理对象，远程应用可以用网络协议与服务进行通信。

我们可以通过一张图来详细地了解下整个 RMI 的通信过程：



RMI 在高并发场景下的性能瓶颈

Java 默认序列化

RMI 的序列化采用的是 Java 默认的序列化方式，我在 09 讲中详细地介绍过 Java 序列化，我们深知它的性能并不是很好，而且其它语言框架也暂时不支持 Java 序列化。

TCP 短连接

由于 RMI 是基于 TCP 短连接实现，在高并发情况下，大量请求会带来大量连接的创建和销毁，这对于系统来说无疑是非常消耗性能的。

阻塞式网络 I/O

在 08 讲中，我提到了网络通信存在 I/O 瓶颈，如果在 Socket 编程中使用传统的 I/O 模型，在高并发场景下基于短连接实现的网络通信就很容易产生 I/O 阻塞，性能将会大打折扣。

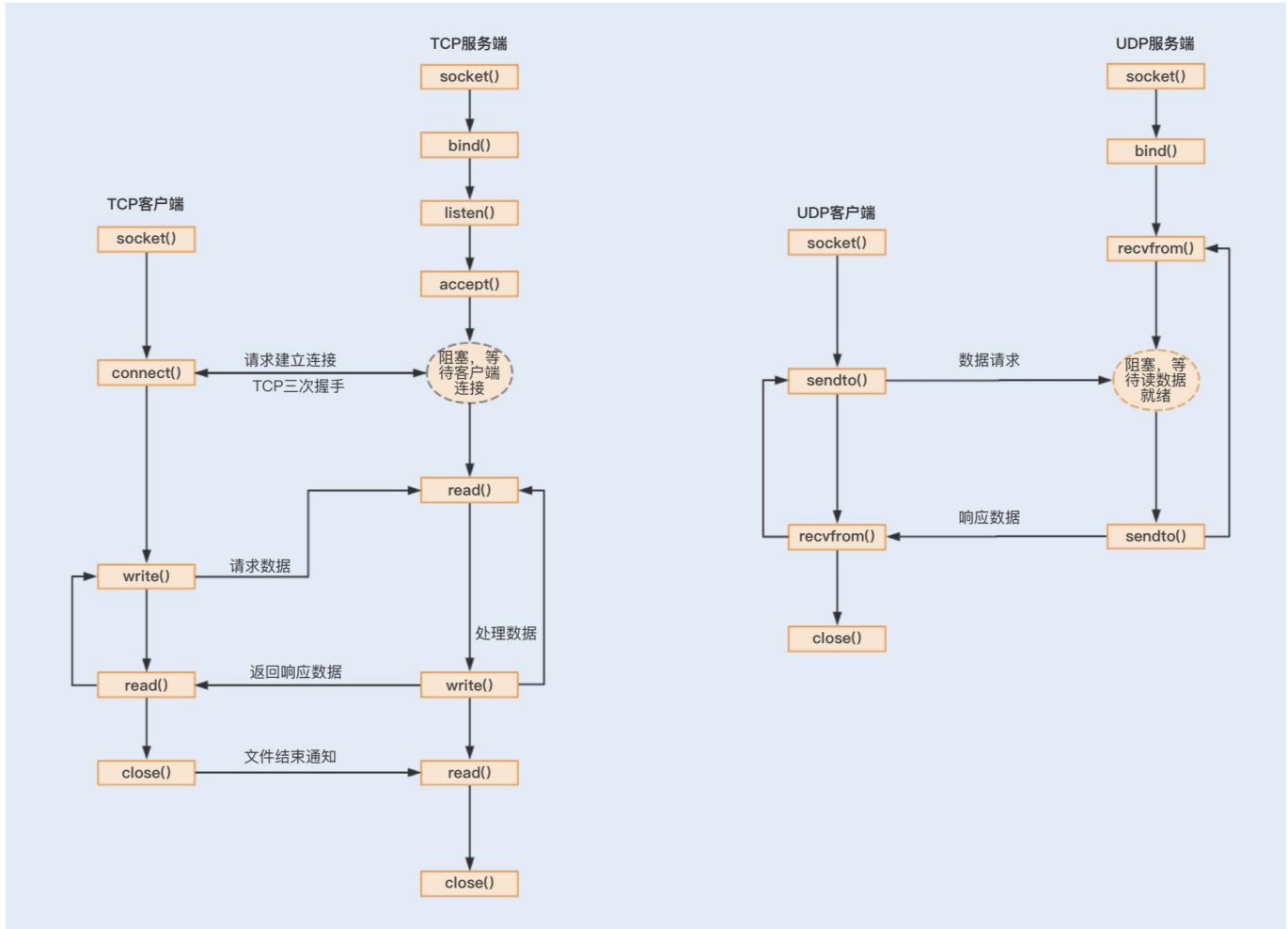
一个高并发场景下的 RPC 通信优化路径

SpringCloud 的 RPC 通信和 RMI 通信的性能瓶颈就非常相似。SpringCloud 是基于 Http 通信协议（短连接）和 Json 序列化实现的，在高并发场景下并没有优势。那么，在瞬时高并发的场景下，我们又该如何去优化一个 RPC 通信呢？

RPC 通信包括了建立通信、实现报文、传输协议以及传输数据编解码等操作，接下来我们就从每一层的优化出发，逐步实现整体的性能优化。

1. 选择合适的通信协议

要实现不同机器间的网络通信，我们先要了解计算机系统网络通信的基本原理。网络通信是两台设备之间实现数据流交换的过程，是基于网络传输协议和传输数据的编解码来实现的。其中网络传输协议有 TCP、UDP 协议，这两个协议都是基于 Socket 编程接口之上，为某类应用场景而扩展出的传输协议。通过以下两张图，我们可以大概了解到基于 TCP 和 UDP 协议实现的 Socket 网络通信是怎样的一个流程。



基于 TCP 协议实现的 Socket 通信是有连接的，而传输数据是要通过三次握手来实现数据传输的可靠性，且传输数据是没有边界的，采用的是字节流模式。

基于 UDP 协议实现的 Socket 通信，客户端不需要建立连接，只需要创建一个套接字发送数据报给服务端，这样就不能保证数据报一定会达到服务端，所以在传输数据方面，基于 UDP 协议实现的 Socket 通信具有不可靠性。UDP 发送的数据采用的是数据报模式，每个 UDP 的数据报都有一个长度，该长度将与数据一起发送到服务端。

通过对比，我们可以得出优化方法：为了保证数据传输的可靠性，通常情况下我们会采用 **TCP 协议**。如果在局域网且对数据传输的可靠性没有要求的情况下，我们也可以考虑使用 UDP 协议，毕竟这种协议的效率要比 TCP 协议高。

2. 使用单一长连接

如果是基于 TCP 协议实现 Socket 通信，我们还能做哪些优化呢？

服务之间的通信不同于客户端与服务端之间的通信。客户端与服务端由于客户端数量多，基于短连接实现请求可以避免长时间地占用连接，导致系统资源浪费。

但服务之间的通信，连接的消费端不会像客户端那么多，但消费端向服务端请求的数量却一样多，我们基于长连接实现，就可以省去大量的 TCP 建立和关闭连接的操作，从而减少系统的性能消耗，节省时间。

3. 优化 Socket 通信

建立两台机器的网络通信，我们一般使用 Java 的 Socket 编程实现一个 TCP 连接。传统的 Socket 通信主要存在 I/O 阻塞、线程模型缺陷以及内存拷贝等问题。我们可以使用比较成熟的通信框架，比如 Netty。Netty4 对 Socket 通信编程做了很多方面的优化，具体见下方。

实现非阻塞 I/O：在 08 讲中，我们提到了多路复用器 Selector 实现了非阻塞 I/O 通信。

高效的 Reactor 线程模型：Netty 使用了主从 Reactor 多线程模型，服务端接收客户端请求连接是用了一个主线程，这个主线程用于客户端的连接请求操作，一旦连接建立成功，将会监听 I/O 事件，监听到事件后会创建一个链路请求。

链路请求将会注册到负责 I/O 操作的 I/O 工作线程上，由 I/O 工作线程负责后续的 I/O 操作。利用这种线程模型，可以解决在高负载、高并发的情况下，由于单个 NIO 线程无法监听海量客户端和满足大量 I/O 操作造成的问题。

串行设计：服务端在接收消息之后，存在着编码、解码、读取和发送等链路操作。如果这些操作都是基于并行去实现，无疑会导致严重的锁竞争，进而导致系统的性能下降。为了提升性能，Netty 采用了串行无锁化完成链路操作，Netty 提供了 Pipeline 实现链路的各个操作在运行期间不进行线程切换。

零拷贝：在 08 讲中，我们提到了一个数据从内存发送到网络中，存在着两次拷贝动作，先是用户空间拷贝到内核空间，再是从内核空间拷贝到网络 I/O 中。而 NIO 提供的 ByteBuffer 可以使用 Direct Buffers 模式，直接开辟一个非堆物理内存，不需要进行字节缓冲区的二次拷贝，可以直接将数据写入到内核空间。

除了以上这些优化，我们还可以针对套接字编程提供的一些 TCP 参数配置项，提高网络吞吐量，Netty 可以基于 ChannelOption 来设置这些参数。

TCP_NODELAY：TCP_NODELAY 选项是用来控制是否开启 Nagle 算法。Nagle 算法通过缓存的方式将小的数据包组成一个大的数据包，从而避免大量的小数据包发送阻塞网络，

提高网络传输的效率。我们可以关闭该算法，优化对于时延敏感的应用场景。

SO_RCVBUF 和 SO_SNDBUF：可以根据场景调整套接字发送缓冲区和接收缓冲区的大小。

SO_BACKLOG：backlog 参数指定了客户端连接请求缓冲队列的大小。服务端处理客户端连接请求是按顺序处理的，所以同一时间只能处理一个客户端连接，当有多个客户端进来的时候，服务端就会将不能处理的客户端连接请求放在队列中等待处理。

SO_KEEPALIVE：当设置该选项以后，连接会检查长时间没有发送数据的客户端的连接状态，检测到客户端断开连接后，服务端将回收该连接。我们可以将该时间设置得短一些，来提高回收连接的效率。

4. 量身定做报文格式

接下来就是实现报文，我们需要设计一套报文，用于描述具体的校验、操作、传输数据等内容。为了提高传输的效率，我们可以根据自己的业务和架构来考虑设计，尽量实现报体小、满足功能、易解析等特性。我们可以参考下面的数据格式：

魔数(0×12345678)	版本号(1)	序列化算法	指令	数据长度	数据
4字节	1字节	1字节	1字节	4字节	N字节

字段	定义的作用
魔数	它的作用类似于协议内的标识，通过客户端与服务端魔数对比，我们就知道这组二进制数据是否属于当前通信协议，通常情况下魔数是一个固定数字。
版本号	占用 1 个字节，通常情况下是预留字段。
序列化算法	占用 1 个字节，表示对数据编码和解码的方式，比如，我在09讲介绍过的 Java 自带序列化，还有 Google 实现了 Protobuf 序列化。
指令	占用 1 个字节，服务端或者客户端每收到一种指令都会有相应的处理逻辑，最高支持 256 种指令，比如，Http 中的增删改查指令。
数据长度	占用 4 个字节，这个字段用来截取数据。
数据	用于存储编码后的数据。

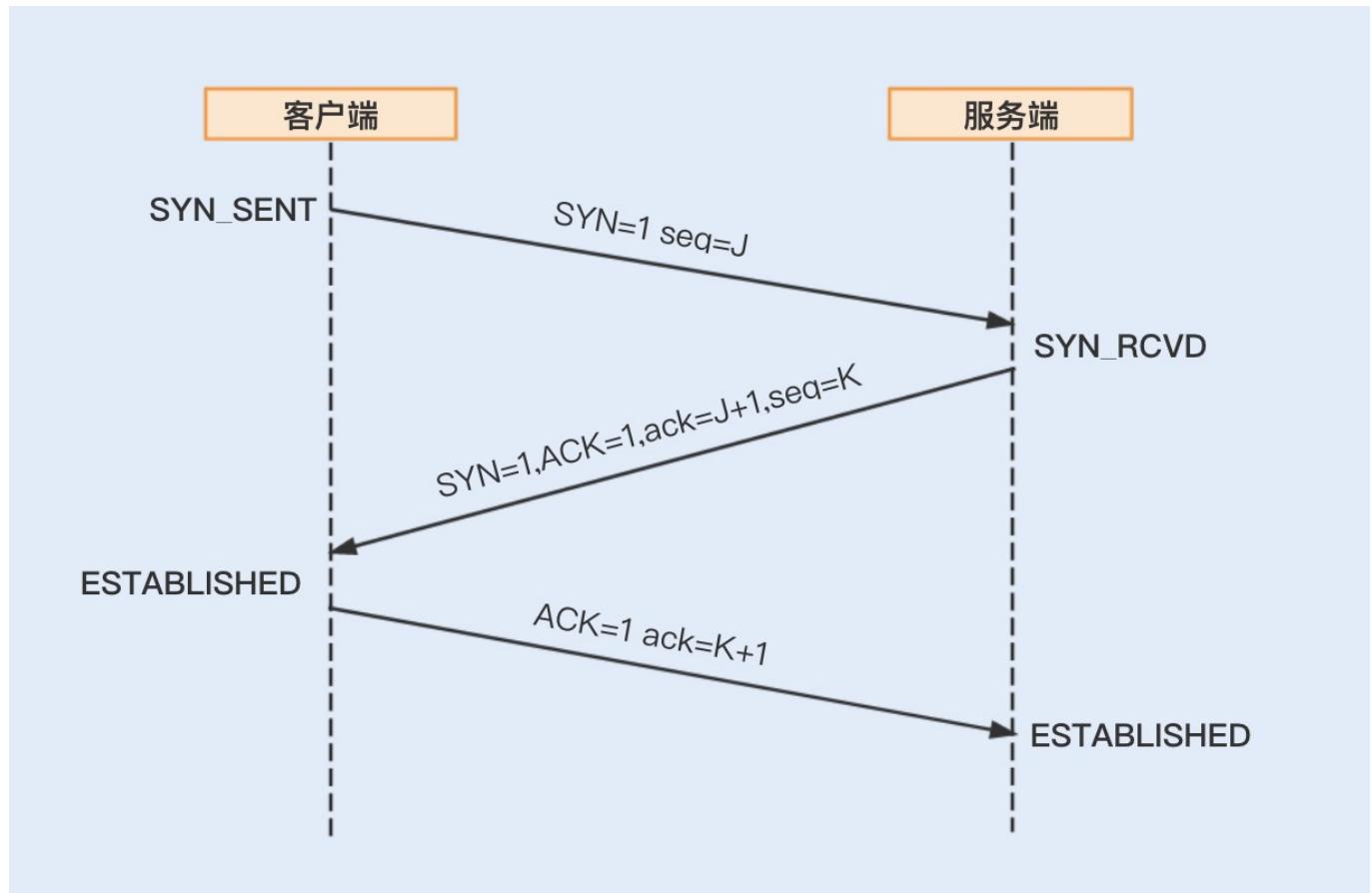
5. 编码、解码

在 09 讲中，我们分析过序列化编码和解码的过程，对于实现一个好的网络通信协议来说，兼容优秀的序列化框架是非常重要的。如果只是单纯的数据对象传输，我们可以选择性能相对较好的 Protobuf 序列化，有利于提高网络通信的性能。

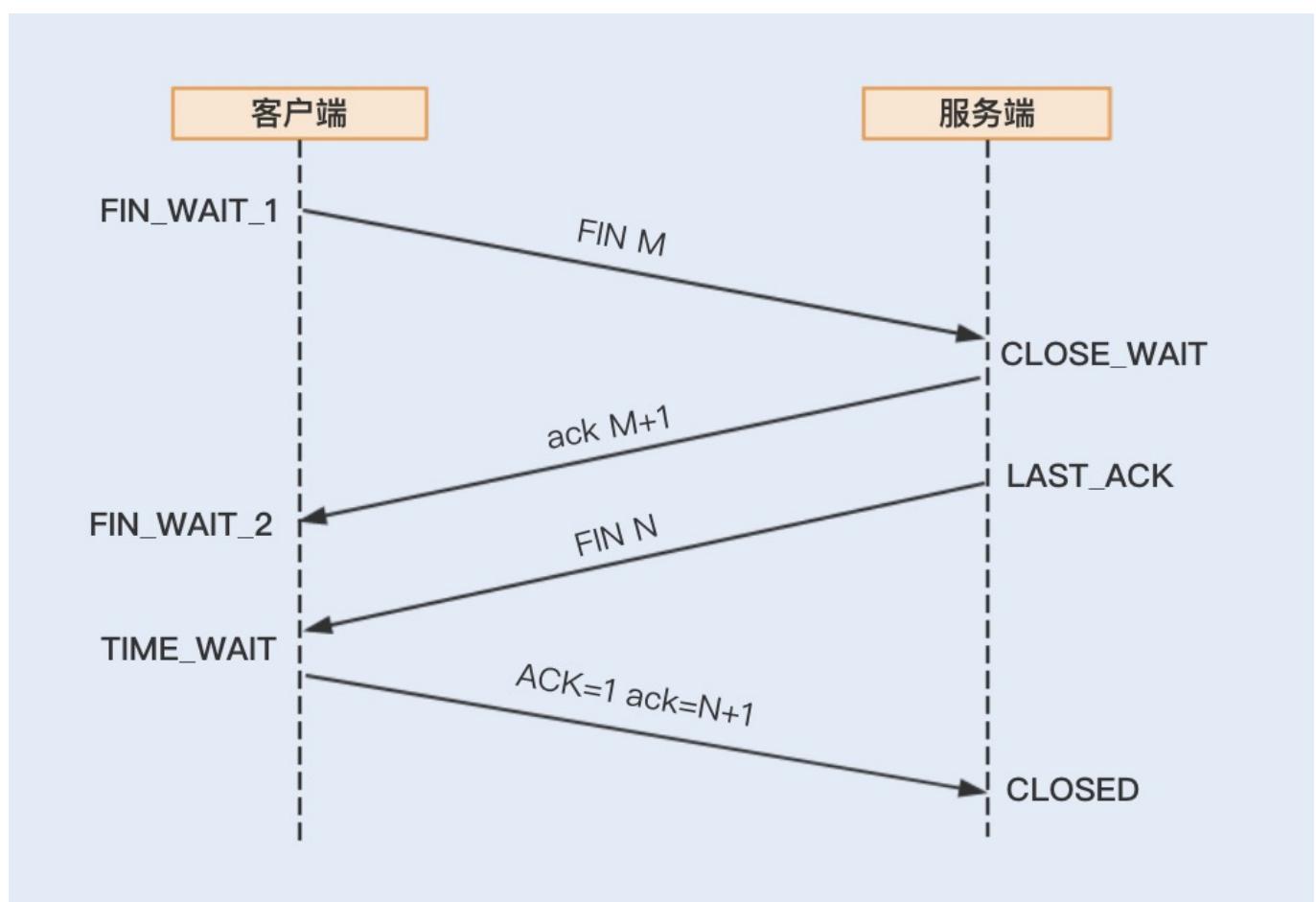
6. 调整 Linux 的 TCP 参数设置选项

如果 RPC 是基于 TCP 短连接实现的，我们可以通过修改 Linux TCP 配置项来优化网络通信。开始 TCP 配置项的优化之前，我们先来了解下建立 TCP 连接的三次握手和关闭 TCP 连接的四次握手，这样有助后面内容的理解。

三次握手



四次握手



我们可以通过 `sysctl -a | grep net.xxx` 命令运行查看 Linux 系统默认的的 TCP 参数设置，如果需要修改某项配置，可以通过编辑 `vim/etc/sysctl.conf`，加入需要修改的配置项，并通过 `sysctl -p` 命令运行生效修改后的配置项设置。通常我们会通过修改以下几个配置项来提高网络吞吐量和降低延时。

配置项	说明+优化方法
file-max/ ulimit	Linux系统默认单个进程用户open files（一个socket连接相当于一个open file）数量为1024，有时应用程序会报“java.io.IOException：打开的文件过多”的错误，就说明open files数量不够。
	我们可以通过设置ulimit来增加单个用户的open files数量，file-max表示系统总的打开文件数量，如果单个进程的open files数量已经超过file-max，就要修改file-max。
net.ipv4.tcp_keepalive_time	该配置项与Netty的SO_KEEPALIVE配置项的作用是一样的，用来检查客户端的连接状态。
	我们可以通过减少检查状态的时间，来提高连接的回收效率。
net.ipv4.tcp_max_syn_backlog	该配置项表示SYN队列的长度，默认为1024。
	加大队列长度，可以容纳更多等待连接的网络连接数。
net.ipv4.tcp_syncookies	当出现SYN等待队列溢出时，可以开启该配置项。
	启用cookies来处理，可防范少量SYN攻击。
net.ipv4.ip_local_port_range	客户端连接服务端时，需要动态分配源端口号，该配置项表示用于向外连接的端口范围。
	默认端口范围是32768到61000，我们可以扩大该范围。
net.ipv4.tcp_max_tw_buckets	当一个连接关闭时，TCP会通过四次握手来完成一次关闭连接操作。在请求量比较大的情况下，消费端会有大量TIME_WAIT状态的连接。
	由于该参数可以限制TIME_WAIT状态的连接数量，所以我们可以用减小参数的方式来应对。设置完成后，如果TIME_WAIT的数量超过参数值，TIME_WAIT将会立刻被清除并打印警告信息。
net.ipv4.tcp_tw_reuse	客户端每次连接服务端时，都会获得一个新的源端口以实现连接的唯一性。在TIME_WAIT状态的连接数量过大的情况下，会增加端口号的占用时间。
	由于处于TIME_WAIT状态的连接属于关闭连接，所以新创建的连接可以复用该端口号。

以上就是我们从不同层次对 RPC 优化的详解，除了最后的 Linux 系统中 TCP 的配置项设置调优，其它的调优更多是从代码编程优化的角度出发，最终实现了一套 RPC 通信框架的优化路径。

弄懂了这些，你就可以根据自己的业务场景去做技术选型了，还能很好地解决过程中出现的一些性能问题。

总结

在现在的分布式系统中，特别是系统走向微服务化的今天，服务间的通信就显得尤为频繁，掌握服务间的通信原理和通信协议优化，是你的一项的必备技能。

在一些并发场景比较多的系统中，我更偏向使用 Dubbo 实现的这一套 RPC 通信协议。Dubbo 协议是建立的单一长连接通信，网络 I/O 为 NIO 非阻塞读写操作，更兼容了 Kryo、FST、Protobuf 等性能出众的序列化框架，在高并发、小对象传输的业务场景中非常实用。

在企业级系统中，业务往往要比普通的互联网产品复杂，服务与服务之间可能不仅仅是数据传输，还有图片以及文件的传输，所以 RPC 的通信协议设计考虑更多的是功能性需求，在性能方面不追求极致。其它通信框架在功能性、生态以及易用、易入门等方面更具有优势。

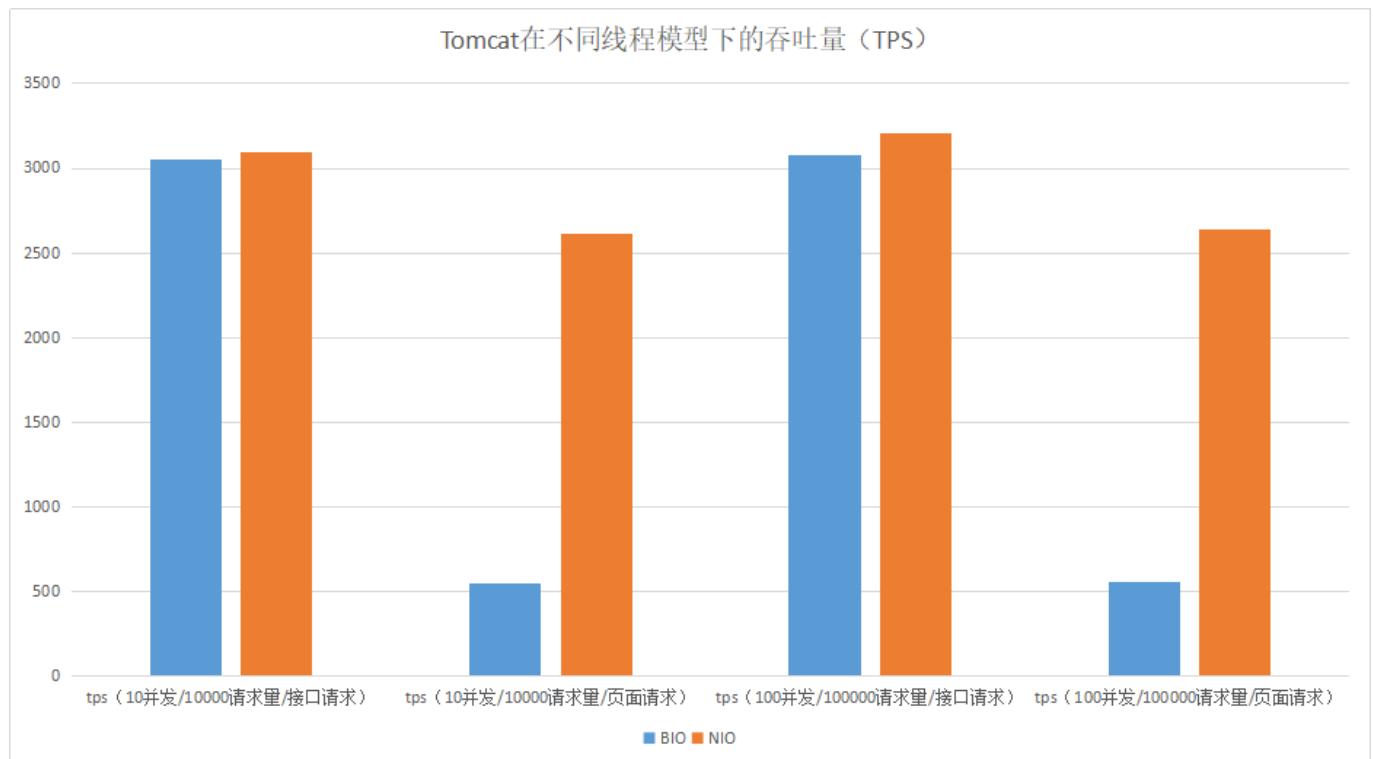
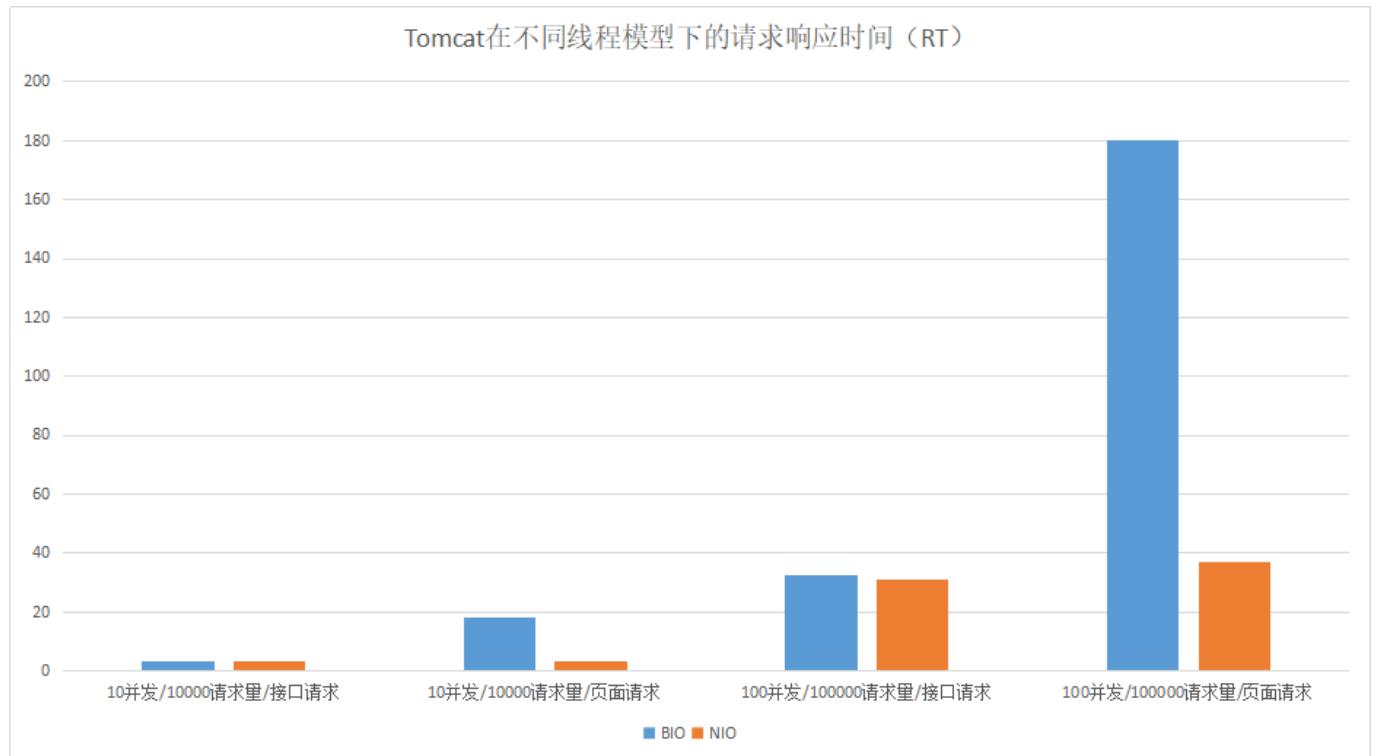
11 | 答疑课堂：深入了解NIO的优化实现原理



综合查看完近期的留言以后，我的第一篇答疑课堂就顺势诞生了。我将继续讲解 I/O 优化，对大家在 08 讲中提到的内容做重点补充，并延伸一些有关 I/O 的知识点，更多结合实际场景进行分享。话不多说，我们马上切入正题。

Tomcat 中经常被提到的一个调优就是修改线程的 I/O 模型。[Tomcat 8.5 版本之前](#)，默认情况下使用的是 BIO 线程模型，如果在高负载、高并发的场景下，可以通过设置 NIO 线程模型，来提高系统的网络通信性能。

我们可以通过一个性能对比测试来看看在高负载或高并发的情况下，BIO 和 NIO 通信性能（这里用页面请求模拟多 I/O 读写操作的请求）：



测试结果：Tomcat 在 I/O 读写操作比较多的情况下，使用 NIO 线程模型有明显的优势。

Tomcat 中看似一个简单的配置，其中却包含了大量的优化升级知识点。下面我们就从底层的网络 I/O 模型优化出发，再到内存拷贝优化和线程模型优化，深入分析下 Tomcat、

Netty 等通信框架是如何通过优化 I/O 来提高系统性能的。

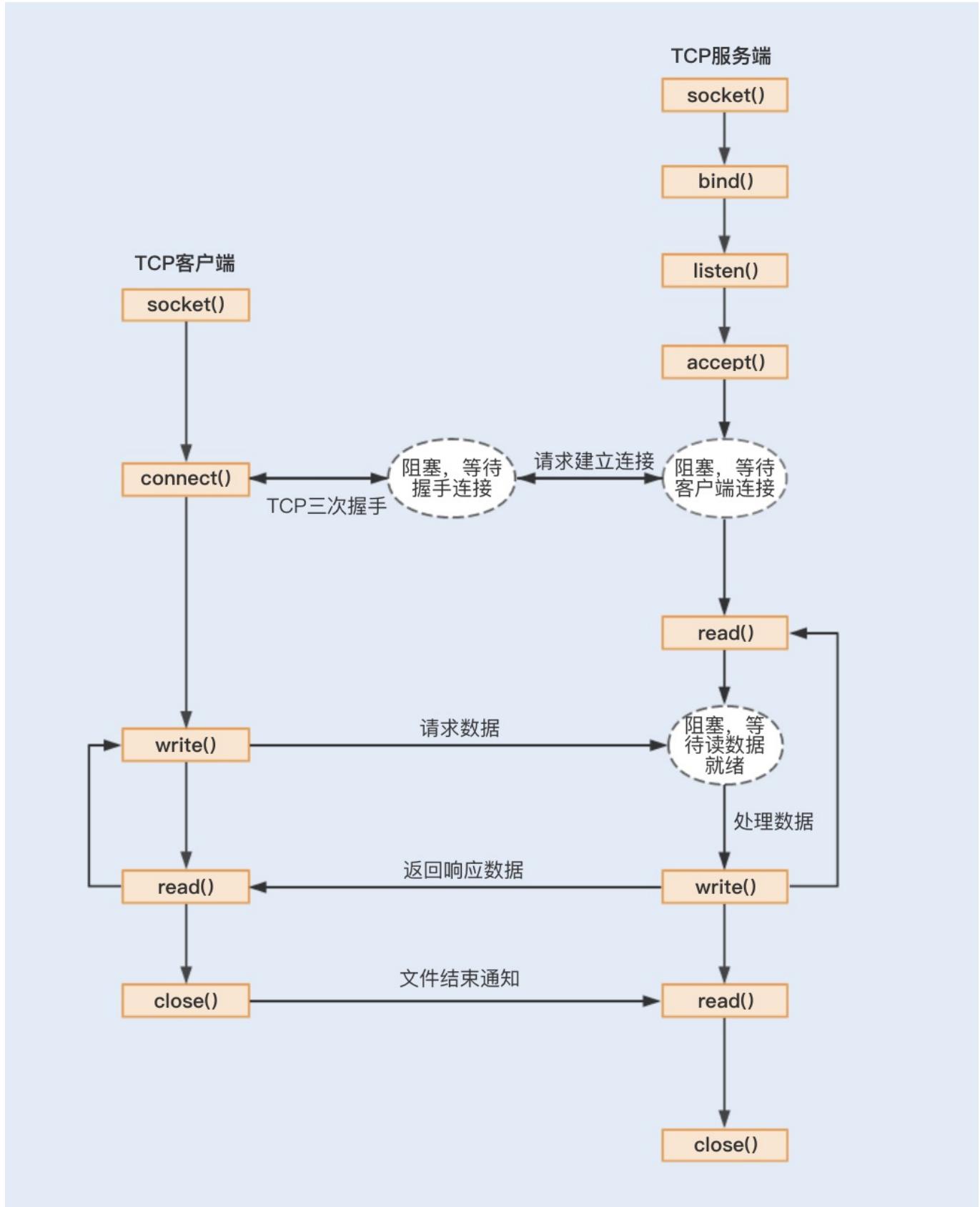
网络 I/O 模型优化

网络通信中，最底层的就是内核中的网络 I/O 模型了。随着技术的发展，操作系统内核的网络模型衍生出了五种 I/O 模型，《UNIX 网络编程》一书将这五种 I/O 模型分为阻塞式 I/O、非阻塞式 I/O、I/O 复用、信号驱动式 I/O 和异步 I/O。每一种 I/O 模型的出现，都是基于前一种 I/O 模型的优化升级。

最开始的阻塞式 I/O，它在每一个连接创建时，都需要一个用户线程来处理，并且在 I/O 操作没有就绪或结束时，线程会被挂起，进入阻塞等待状态，阻塞式 I/O 就成为了导致性能瓶颈的根本原因。

那阻塞到底发生在套接字（socket）通信的哪些环节呢？

在《Unix 网络编程》中，套接字通信可以分为流式套接字（TCP）和数据报套接字（UDP）。其中 TCP 连接是我们最常用的，一起来了解下 TCP 服务端的工作流程（由于 TCP 的数据传输比较复杂，存在拆包和装包的可能，这里我只假设一次最简单的 TCP 数据传输）：



首先，应用程序通过系统调用 `socket` 创建一个套接字，它是系统分配给应用程序的一个文件描述符；

其次，应用程序会通过系统调用 `bind`，绑定地址和端口号，给套接字命名一个名称；

然后，系统会调用 `listen` 创建一个队列用于存放客户端进来的连接；

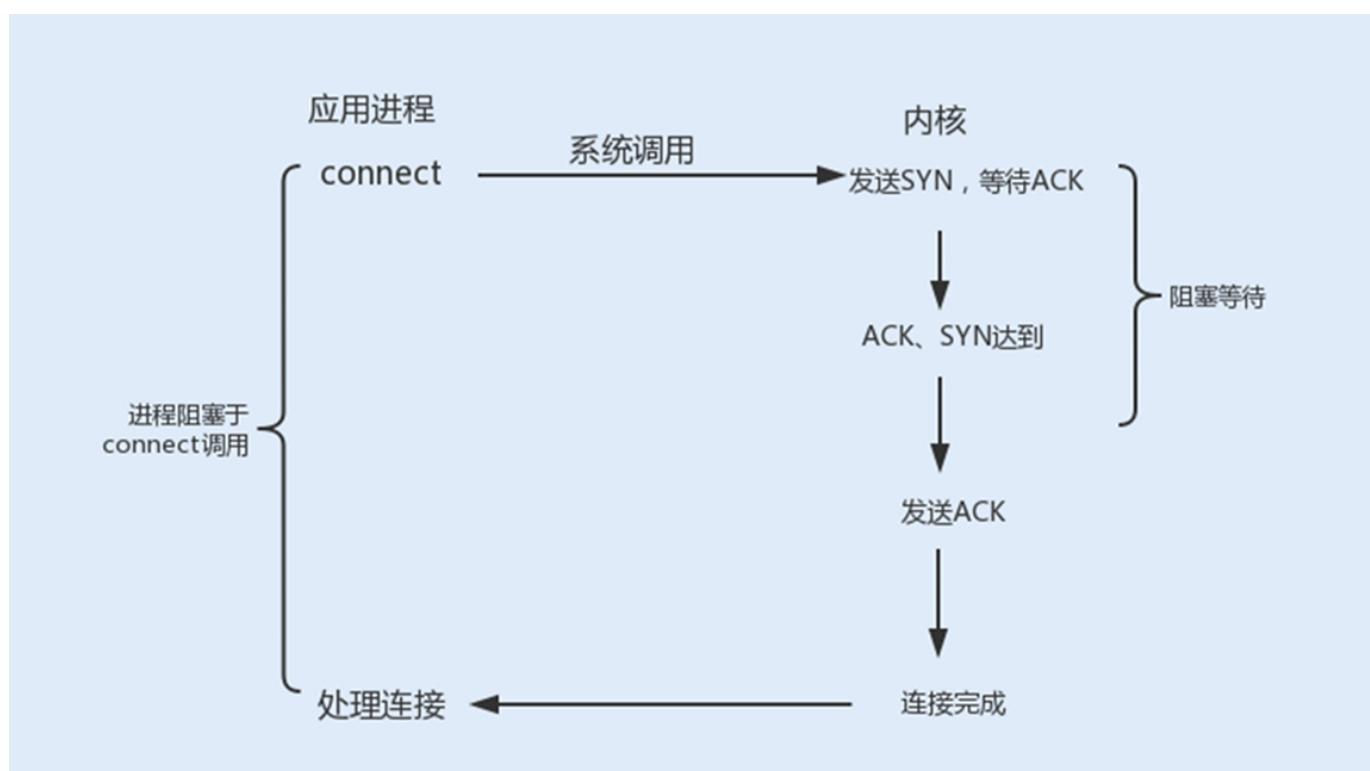
最后，应用服务会通过系统调用 accept 来监听客户端的连接请求。

当有一个客户端连接到服务端之后，服务端就会调用 fork 创建一个子进程，通过系统调用 read 监听客户端发来的消息，再通过 write 向客户端返回信息。

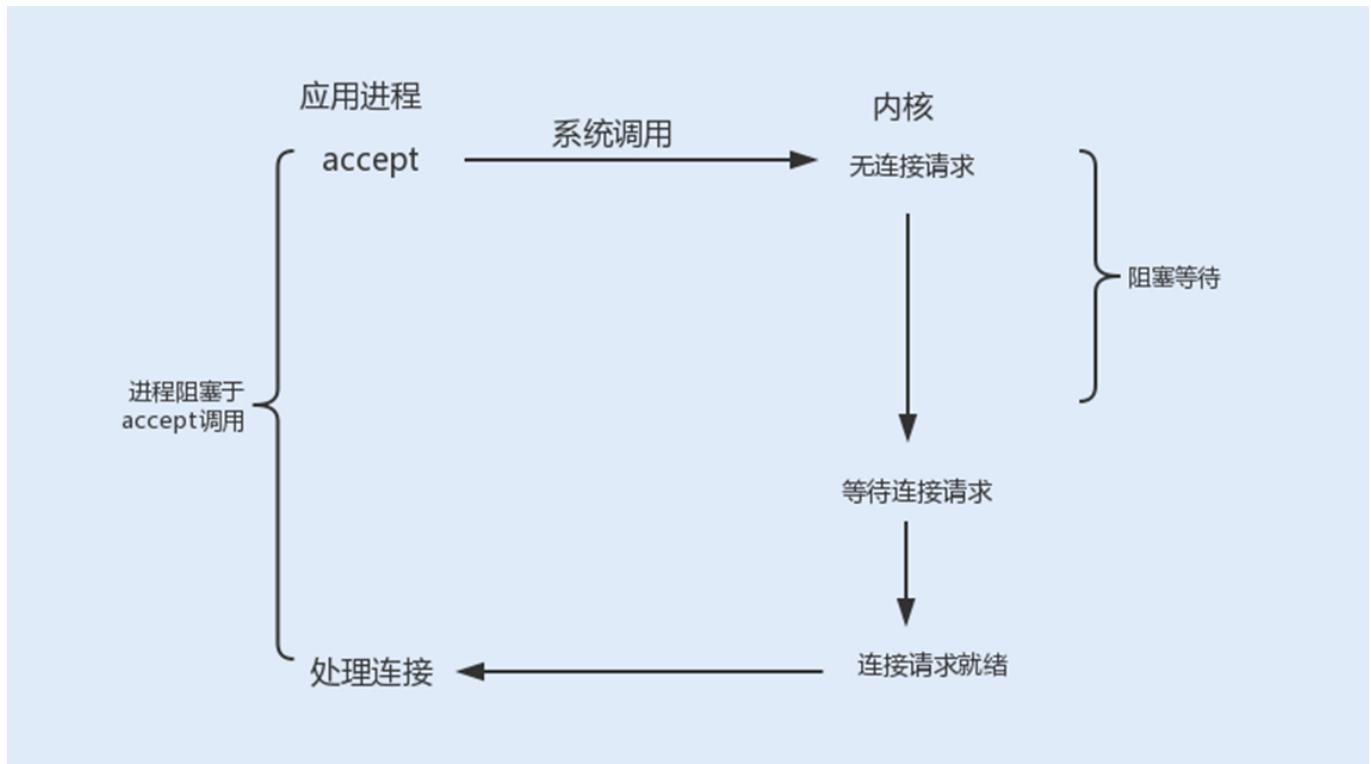
1. 阻塞式 I/O

在整个 socket 通信工作流程中，socket 的默认状态是阻塞的。也就是说，当发出一个不能立即完成的套接字调用时，其进程将被阻塞，被系统挂起，进入睡眠状态，一直等待相应的操作响应。从上图中，我们可以发现，可能存在的阻塞主要包括以下三种。

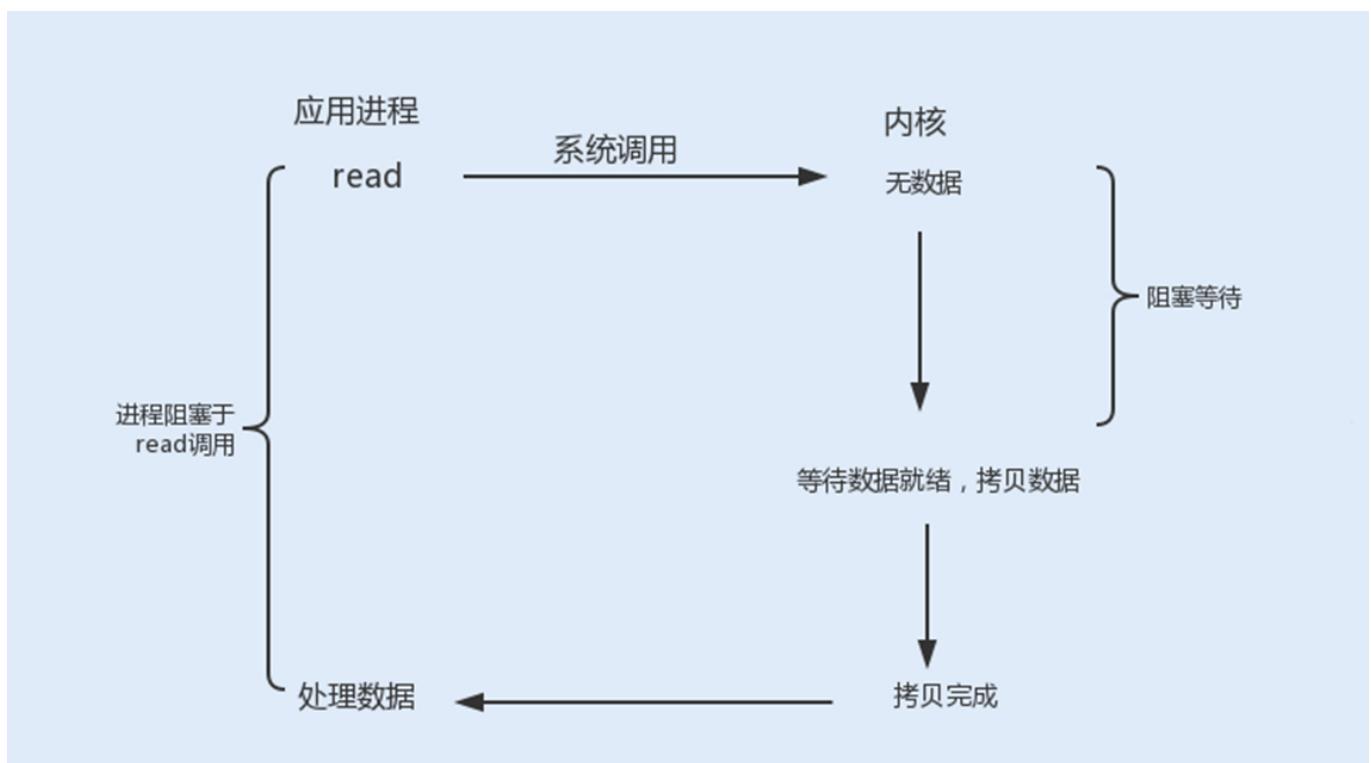
connect 阻塞：当客户端发起 TCP 连接请求，通过系统调用 connect 函数，TCP 连接的建立需要完成三次握手过程，客户端需要等待服务端发送回来的 ACK 以及 SYN 信号，同样服务端也需要阻塞等待客户端确认连接的 ACK 信号，这就意味着 TCP 的每个 connect 都会阻塞等待，直到确认连接。



accept 阻塞：一个阻塞的 socket 通信的服务端接收外来连接，会调用 accept 函数，如果没有新的连接到达，调用进程将被挂起，进入阻塞状态。



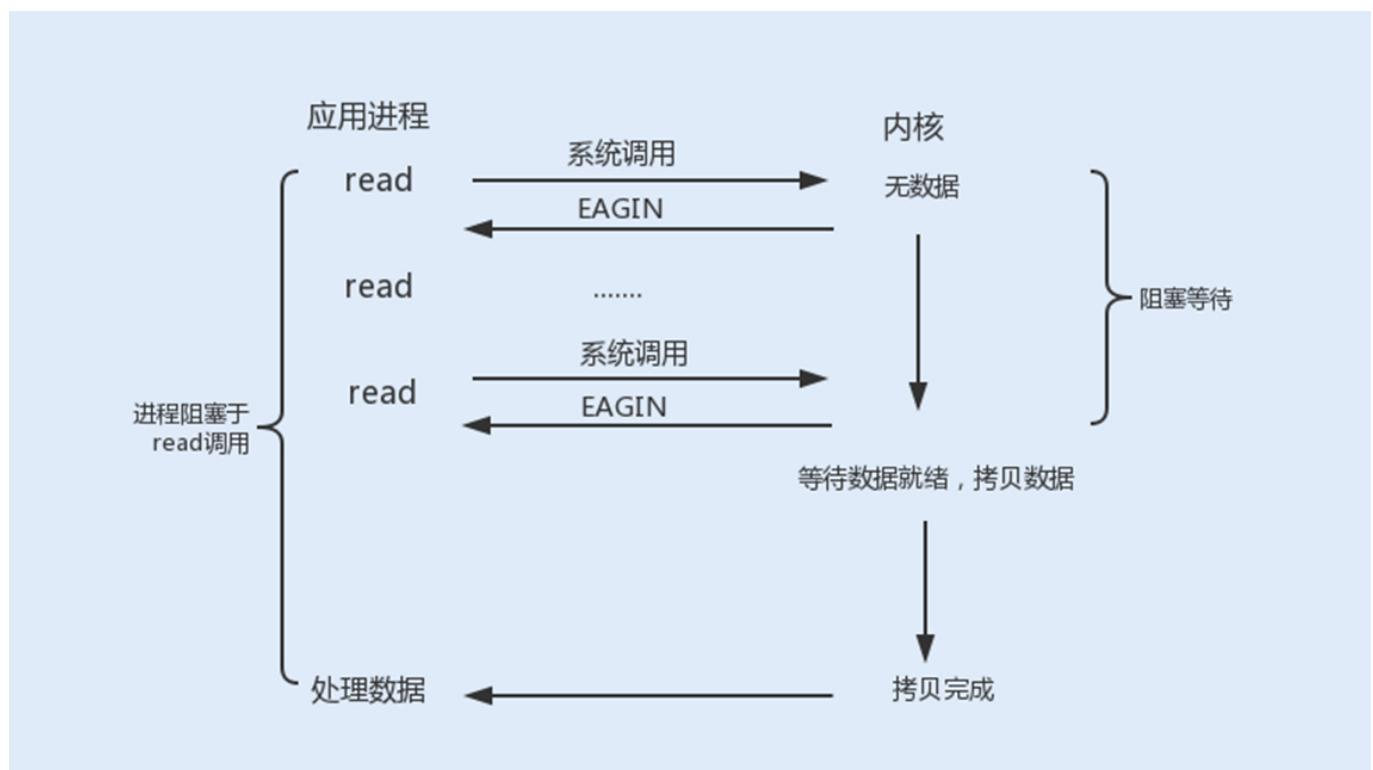
read、write 阻塞：当一个 socket 连接创建成功之后，服务端用 `fork` 函数创建一个子进程，调用 `read` 函数等待客户端的数据写入，如果没有数据写入，调用子进程将被挂起，进入阻塞状态。



2. 非阻塞式 I/O

使用 `fcntl` 可以把以上三种操作都设置为非阻塞操作。如果没有数据返回，就会直接返回一个 `EWOULDBLOCK` 或 `EAGAIN` 错误，此时进程就不会一直被阻塞。

当我们把以上操作设置为了非阻塞状态，我们需要设置一个线程对该操作进行轮询检查，这也是最传统的非阻塞 I/O 模型。



3. I/O 复用

如果使用用户线程轮查看一个 I/O 操作的状态，在大量请求的情况下，这对于 CPU 的使用率无疑是种灾难。那么除了这种方式，还有其它方式可以实现非阻塞 I/O 套接字吗？

Linux 提供了 I/O 复用函数 select/poll/epoll，进程将一个或多个读操作通过系统调用函数，阻塞在函数操作上。这样，系统内核就可以帮我们侦测多个读操作是否处于就绪状态。

select() 函数：它的用途是，在超时时间内，监听用户感兴趣的文件描述符上的可读可写和异常事件的发生。Linux 操作系统的内核将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令，返回一个文件描述符 (fd) 。

复制代码

```
1 int select(int maxfdp1,fd_set *readset,fd_set *writeset,fd_set *exceptset,const struct
```

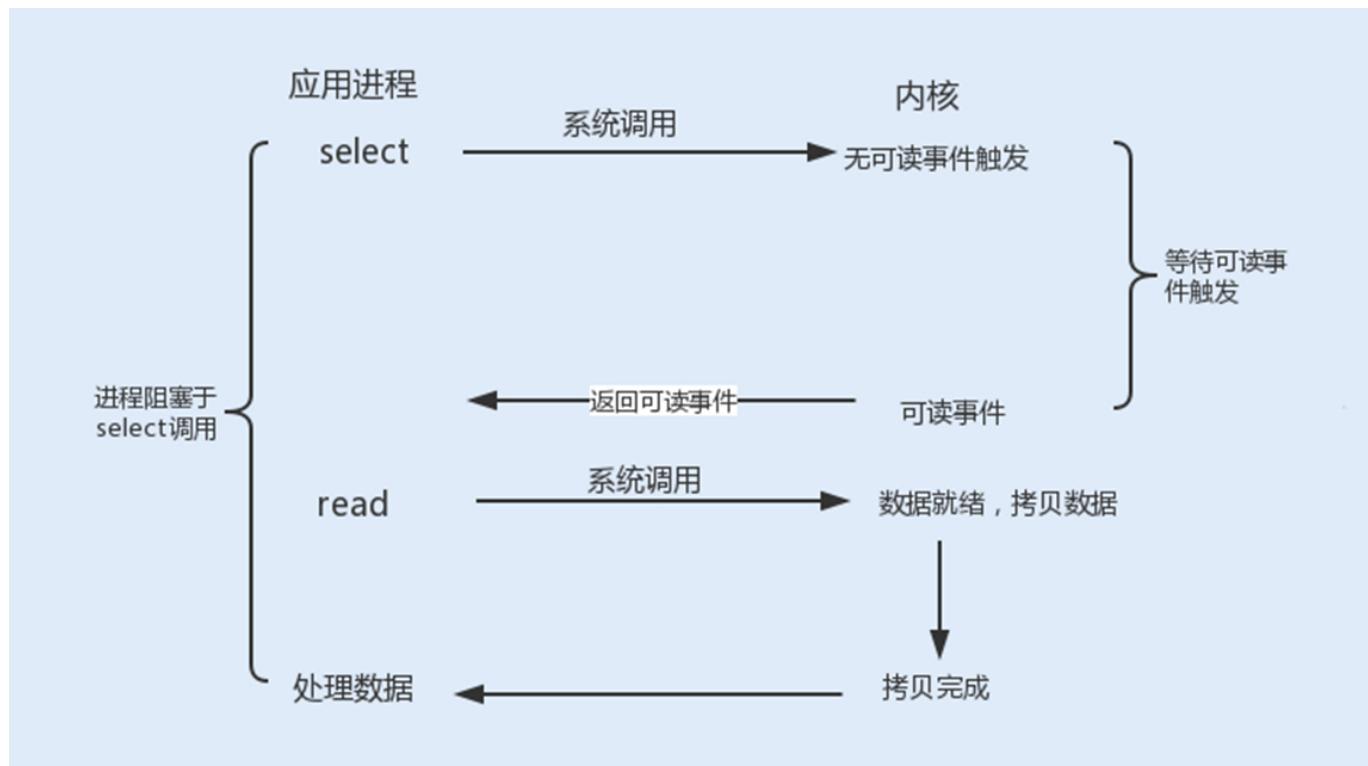
查看以上代码，select() 函数监视的文件描述符分 3 类，分别是 writefds (写文件描述符)、readfds (读文件描述符) 以及 exceptfds (异常事件文件描述符) 。

调用后 `select()` 函数会阻塞，直到有描述符就绪或者超时，函数返回。当 `select` 函数返回后，可以通过函数 `FD_ISSET` 遍历 `fdset`，来找到就绪的描述符。`fd_set` 可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

 复制代码

```
1 void FD_ZERO(fd_set *fdset);           // 清空集合
2 void FD_SET(int fd, fd_set *fdset);     // 将一个给定的文件描述符加入集合之中
3 void FD_CLR(int fd, fd_set *fdset);     // 将一个给定的文件描述符从集合中删除
4 int FD_ISSET(int fd, fd_set *fdset);    // 检查集合中指定的文件描述符是否可以读写
```

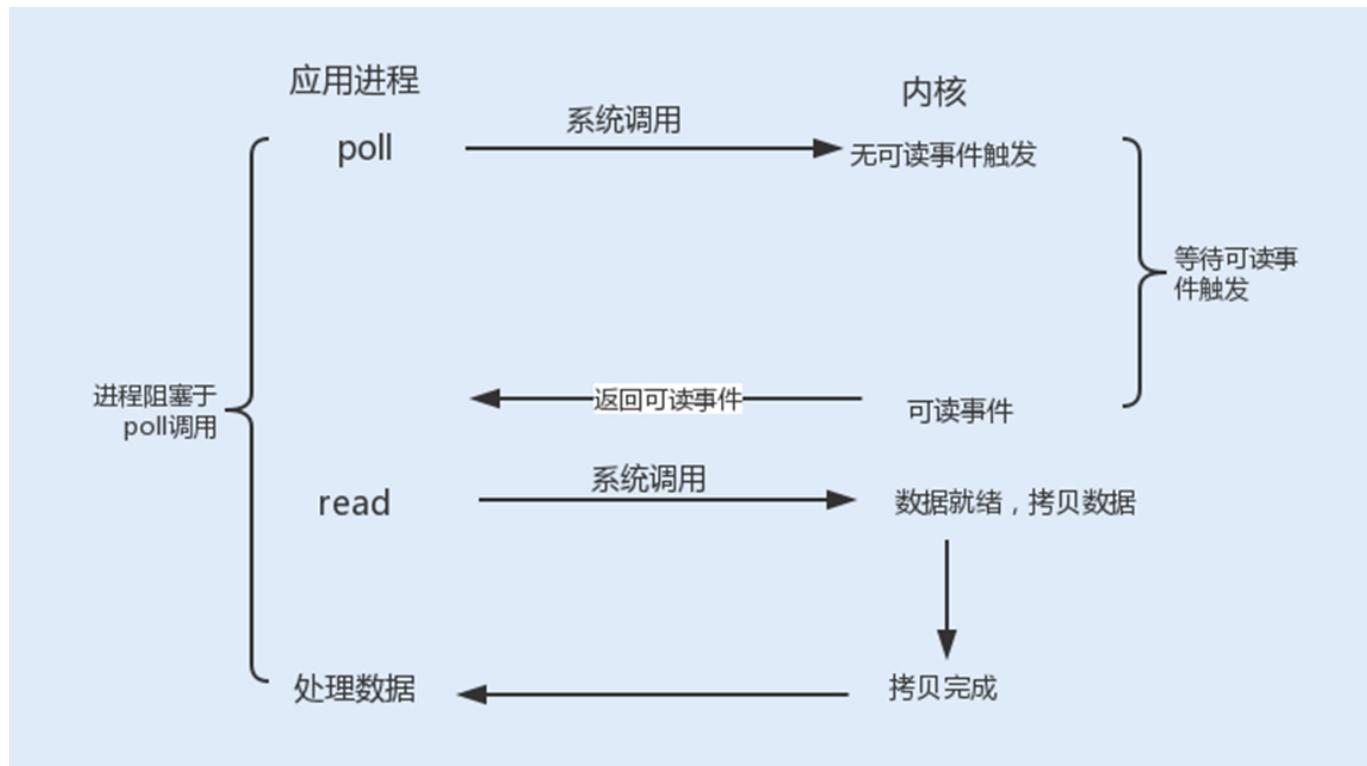


poll() 函数：在每次调用 `select()` 函数之前，系统需要把一个 `fd` 从用户态拷贝到内核态，这样就给系统带来了一定的性能开销。再有单个进程监视的 `fd` 数量默认是 1024，我们可以通过修改宏定义甚至重新编译内核的方式打破这一限制。但由于 `fd_set` 是基于数组实现的，在新增和删除 `fd` 时，数量过大会导致效率降低。

`poll()` 的机制与 `select()` 类似，二者在本质上差别不大。`poll()` 管理多个描述符也是通过轮询，根据描述符的状态进行处理，但 `poll()` 没有最大文件描述符数量的限制。

`poll()` 和 `select()` 存在一个相同的缺点，那就是包含大量文件描述符的数组被整体复制到用户态和内核的地址空间之间，而无论这些文件描述符是否就绪，他们的开销都会随着文件描

述符数量的增加而线性增大。



epoll() 函数：`select/poll` 是顺序扫描 `fd` 是否就绪，而且支持的 `fd` 数量不宜过大，因此它的使用受到了一些制约。

Linux 在 2.6 内核版本中提供了一个 `epoll` 调用，`epoll` 使用事件驱动的方式代替轮询扫描 `fd`。`epoll` 事先通过 `epoll_ctl()` 来注册一个文件描述符，将文件描述符存放到内核的一个事件表中，这个事件表是基于红黑树实现的，所以在大量 I/O 请求的场景下，插入和删除的性能比 `select/poll` 的数组 `fd_set` 要好，因此 `epoll` 的性能更胜一筹，而且不会受到 `fd` 数量的限制。

复制代码

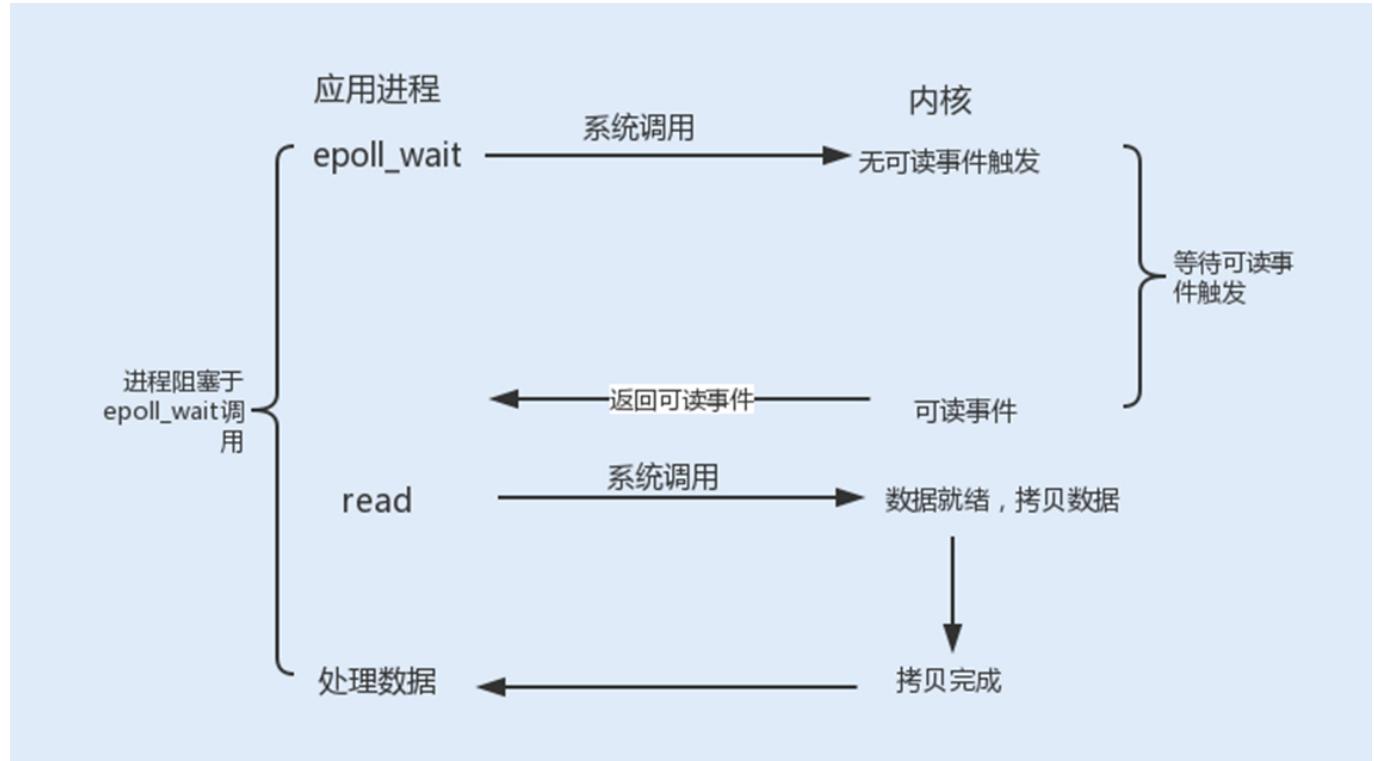
```
1 int epoll_ctl(int epfd, int op, int fd, struct epoll_event event)  
2
```

通过以上代码，我们可以看到：`epoll_ctl()` 函数中的 `epfd` 是由 `epoll_create()` 函数生成的一个 `epoll` 专用文件描述符。`op` 代表操作事件类型，`fd` 表示关联文件描述符，`event` 表示指定监听的事件类型。

一旦某个文件描述符就绪时，内核会采用类似 callback 的回调机制，迅速激活这个文件描述符，当进程调用 epoll_wait() 时便得到通知，之后进程将完成相关 I/O 操作。

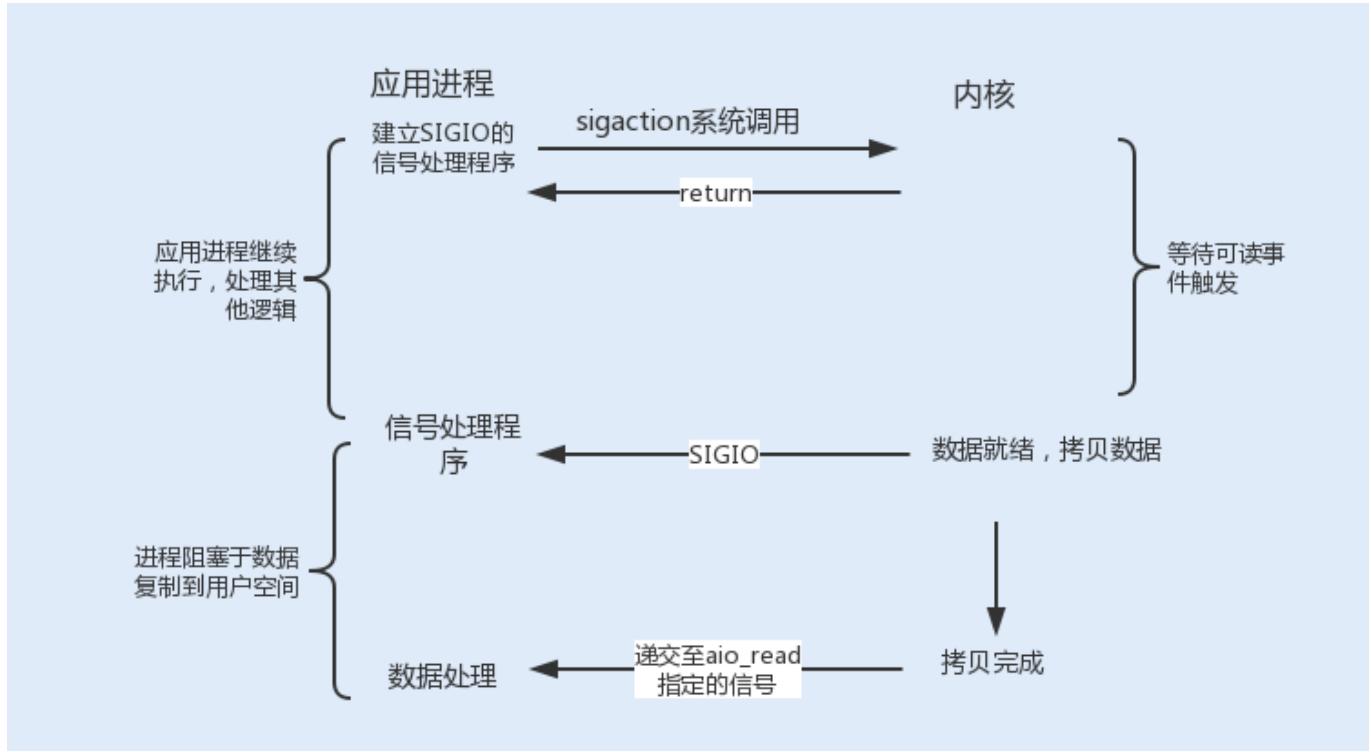
 复制代码

```
1 int epoll_wait(int epfd, struct epoll_event events,int maxevents,int timeout)
```



4. 信号驱动式 I/O

信号驱动式 I/O 类似观察者模式，内核就是一个观察者，信号回调则是通知。用户进程发起一个 I/O 请求操作，会通过系统调用 `sigaction` 函数，给对应的套接字注册一个信号回调，此时不阻塞用户进程，进程会继续工作。当内核数据就绪时，内核就为该进程生成一个 `SIGIO` 信号，通过信号回调通知进程进行相关 I/O 操作。



信号驱动式 I/O 相比于前三种 I/O 模式，实现了在等待数据就绪时，进程不被阻塞，主循环可以继续工作，所以性能更佳。

而由于 TCP 来说，信号驱动式 I/O 几乎没有被使用，这是因为 SIGIO 信号是一种 Unix 信号，信号没有附加信息，如果一个信号源有多种产生信号的原因，信号接收者就无法确定究竟发生了什么。而 TCP socket 生产的信号事件有七种之多，这样应用程序收到 SIGIO，根本无从区分处理。

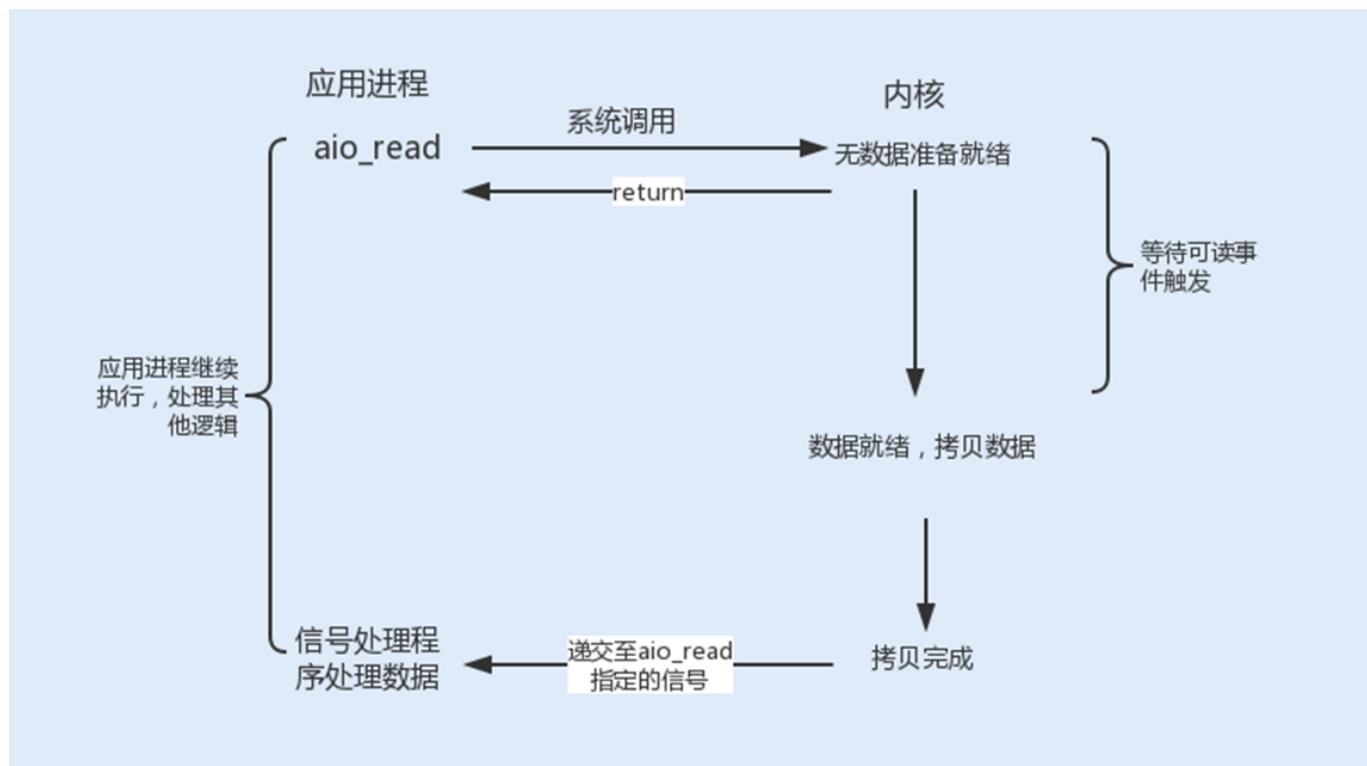
但信号驱动式 I/O 现在被用在了 UDP 通信上，我们从 10 讲中的 UDP 通信流程图中可以发现，UDP 只有一个数据请求事件，这也就意味着在正常情况下 UDP 进程只要捕获 SIGIO 信号，就调用 `recvfrom` 读取到达的数据报。如果出现异常，就返回一个异常错误。比如，NTP 服务器就应用了这种模型。

5. 异步 I/O

信号驱动式 I/O 虽然在等待数据就绪时，没有阻塞进程，但在被通知后进行的 I/O 操作还是阻塞的，进程会等待数据从内核空间复制到用户空间中。而异步 I/O 则是实现了真正的非阻塞 I/O。

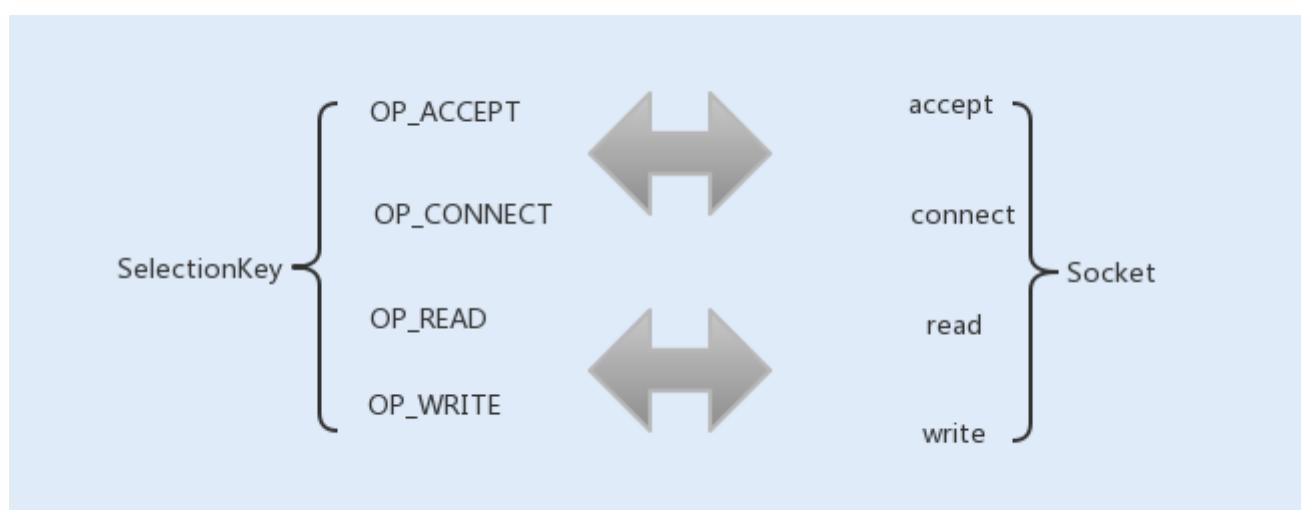
当用户进程发起一个 I/O 请求操作，系统会告知内核启动某个操作，并让内核在整个操作完成后通知进程。这个操作包括等待数据就绪和数据从内核复制到用户空间。由于程序的代

码复杂度高，调试难度大，且支持异步 I/O 的操作系统比较少见（目前 Linux 暂不支持，而 Windows 已经实现了异步 I/O），所以在实际生产环境中很少用到异步 I/O 模型。

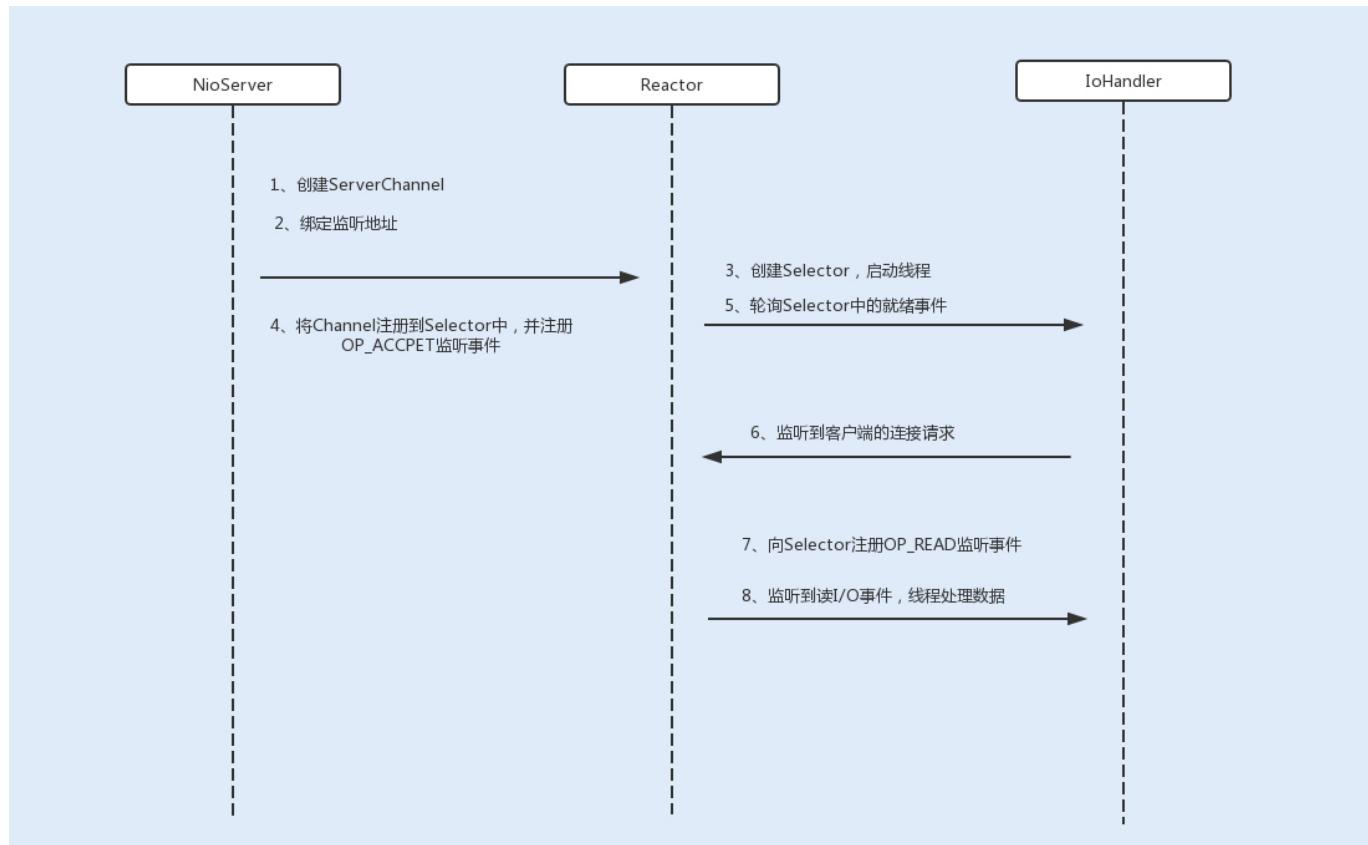


在 08 讲中，我讲到了 NIO 使用 I/O 复用器 Selector 实现非阻塞 I/O，**Selector 就是使用了这五种类型中的 I/O 复用模型**。Java 中的 Selector 其实就是 select/poll/epoll 的外包类。

我们在上面的 TCP 通信流程中讲到，Socket 通信中的 connect、accept、read 以及 write 为阻塞操作，在 Selector 中分别对应 SelectionKey 的四个监听事件 OP_ACCEPT、OP_CONNECT、OP_READ 以及 OP_WRITE。



在 NIO 服务端通信编程中，首先会创建一个 Channel，用于监听客户端连接；接着，创建多路复用器 Selector，并将 Channel 注册到 Selector，程序会通过 Selector 来轮询注册在其上的 Channel，当发现一个或多个 Channel 处于就绪状态时，返回就绪的监听事件，最后程序匹配到监听事件，进行相关的 I/O 操作。



在创建 Selector 时，程序会根据操作系统版本选择使用哪种 I/O 复用函数。在 JDK1.5 版本中，如果程序运行在 Linux 操作系统，且内核版本在 2.6 以上，NIO 中会选择 epoll 来替代传统的 select/poll，这也极大地提升了 NIO 通信的性能。

由于信号驱动式 I/O 对 TCP 通信的不支持，以及异步 I/O 在 Linux 操作系统内核中的应用还不大成熟，大部分框架都还是基于 I/O 复用模型实现的网络通信。

零拷贝

在 I/O 复用模型中，执行读写 I/O 操作依然是阻塞的，在执行读写 I/O 操作时，存在着多次内存拷贝和上下文切换，给系统增加了性能开销。

零拷贝是一种避免多次内存复制的技术，用来优化读写 I/O 操作。

在网络编程中，通常由 read、write 来完成一次 I/O 读写操作。每一次 I/O 读写操作都需要完成四次内存拷贝，路径是 I/O 设备 -> 内核空间 -> 用户空间 -> 内核空间 -> 其它 I/O 设备。

Linux 内核中的 mmap 函数可以代替 read、write 的 I/O 读写操作，实现用户空间和内核空间共享一个缓存数据。mmap 将用户空间的一块地址和内核空间的一块地址同时映射到相同的一块物理内存地址，不管是用户空间还是内核空间都是虚拟地址，最终要通过地址映射映射到物理内存地址。这种方式避免了内核空间与用户空间的数据交换。I/O 复用中的 epoll 函数中就是使用了 mmap 减少了内存拷贝。

在 Java 的 NIO 编程中，则是使用到了 Direct Buffer 来实现内存的零拷贝。Java 直接在 JVM 内存空间之外开辟了一个物理内存空间，这样内核和用户进程都能共享一份缓存数据。这是在 08 讲中已经详细讲解过的内容，你可以再去回顾下。

线程模型优化

除了内核对网络 I/O 模型的优化，NIO 在用户层也做了优化升级。NIO 是基于事件驱动模型来实现的 I/O 操作。Reactor 模型是同步 I/O 事件处理的一种常见模型，其核心思想是将 I/O 事件注册到多路复用器上，一旦有 I/O 事件触发，多路复用器就会将事件分发到事件处理器中，执行就绪的 I/O 事件操作。**该模型有以下三个主要组件：**

事件接收器 Acceptor：主要负责接收请求连接；

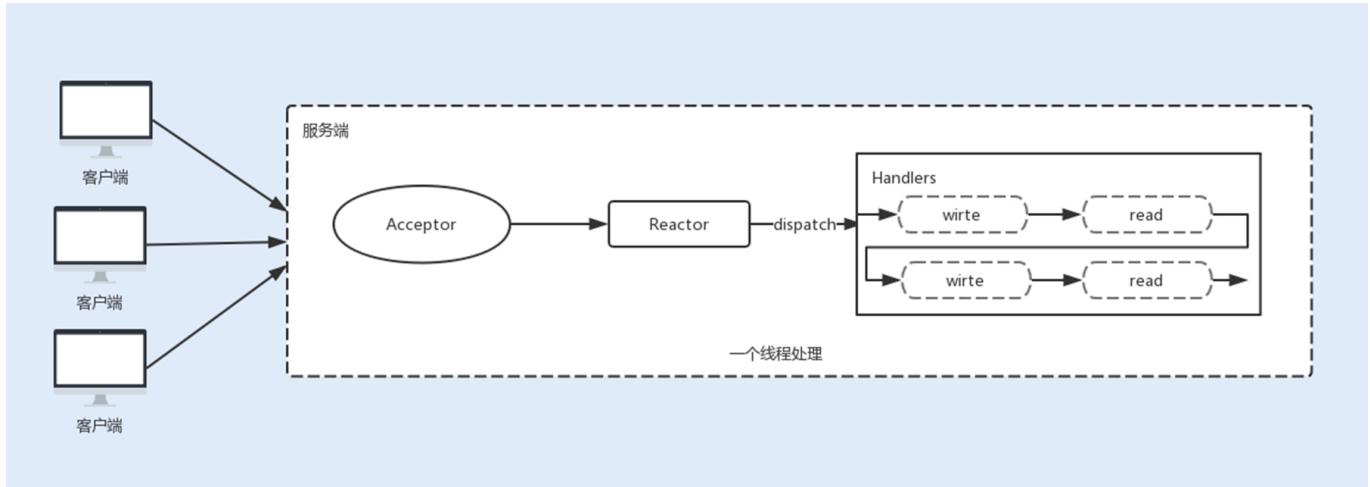
事件分离器 Reactor：接收请求后，会将建立的连接注册到分离器中，依赖于循环监听多路复用器 Selector，一旦监听到事件，就会将事件 dispatch 到事件处理器；

事件处理器 Handlers：事件处理器主要是完成相关的事件处理，比如读写 I/O 操作。

1. 单线程 Reactor 线程模型

最开始 NIO 是基于单线程实现的，所有的 I/O 操作都是在一个 NIO 线程上完成。由于 NIO 是非阻塞 I/O，理论上一个线程可以完成所有的 I/O 操作。

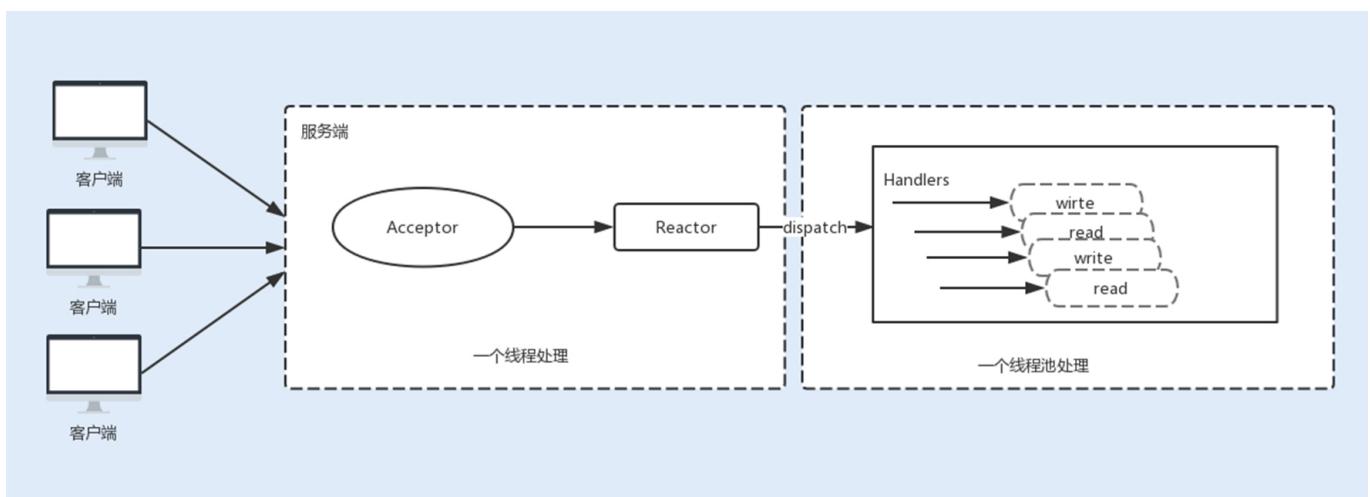
但 NIO 其实还不算真正地实现了非阻塞 I/O 操作，因为读写 I/O 操作时用户进程还是处于阻塞状态，这种方式在高负载、高并发的场景下会存在性能瓶颈，一个 NIO 线程如果同时处理上万连接的 I/O 操作，系统是无法支撑这种量级的请求的。



2. 多线程 Reactor 线程模型

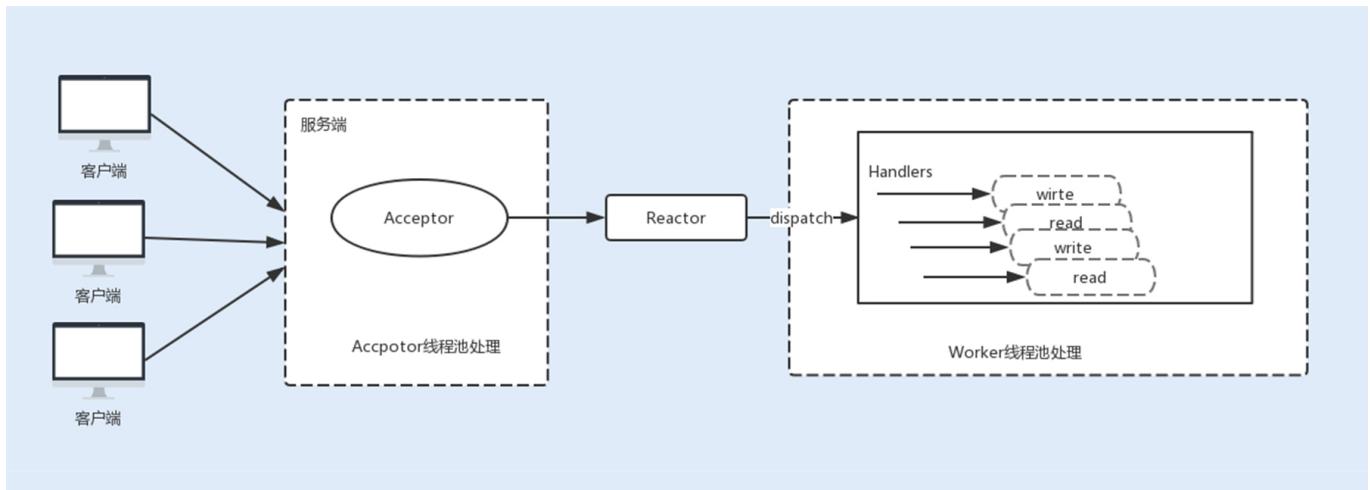
为了解决这种单线程的 NIO 在高负载、高并发场景下的性能瓶颈，后来使用了线程池。

在 Tomcat 和 Netty 中都使用了一个 Acceptor 线程来监听连接请求事件，当连接成功之后，会将建立的连接注册到多路复用器中，一旦监听到事件，将交给 Worker 线程池来负责处理。大多数情况下，这种线程模型可以满足性能要求，但如果连接的客户端再上一个量级，一个 Acceptor 线程可能会存在性能瓶颈。



3. 主从 Reactor 线程模型

现在主流通信框架中的 NIO 通信框架都是基于主从 Reactor 线程模型来实现的。在这个模型中，Acceptor 不再是一个单独的 NIO 线程，而是一个线程池。Acceptor 接收到客户端的 TCP 连接请求，建立连接之后，后续的 I/O 操作将交给 Worker I/O 线程。



基于线程模型的 Tomcat 参数调优

Tomcat 中，BIO、NIO 是基于主从 Reactor 线程模型实现的。

在 BIO 中，Tomcat 中的 Acceptor 只负责监听新的连接，一旦连接建立监听到 I/O 操作，将会交给 Worker 线程中，Worker 线程专门负责 I/O 读写操作。

在 NIO 中，Tomcat 新增了一个 Poller 线程池，Acceptor 监听到连接后，不是直接使用 Worker 中的线程处理请求，而是先将请求发送给了 Poller 缓冲队列。在 Poller 中，维护了一个 Selector 对象，当 Poller 从队列中取出连接后，注册到该 Selector 中；然后通过遍历 Selector，找出其中就绪的 I/O 操作，并使用 Worker 中的线程处理相应的请求。



你可以通过以下几个参数来设置 Acceptor 线程池和 Worker 线程池的配置项。

acceptorThreadCount：该参数代表 Acceptor 的线程数量，在请求客户端的数据量非常巨大的情况下，可以适当地调大该线程数量来提高处理请求连接的能力，默认值为 1。

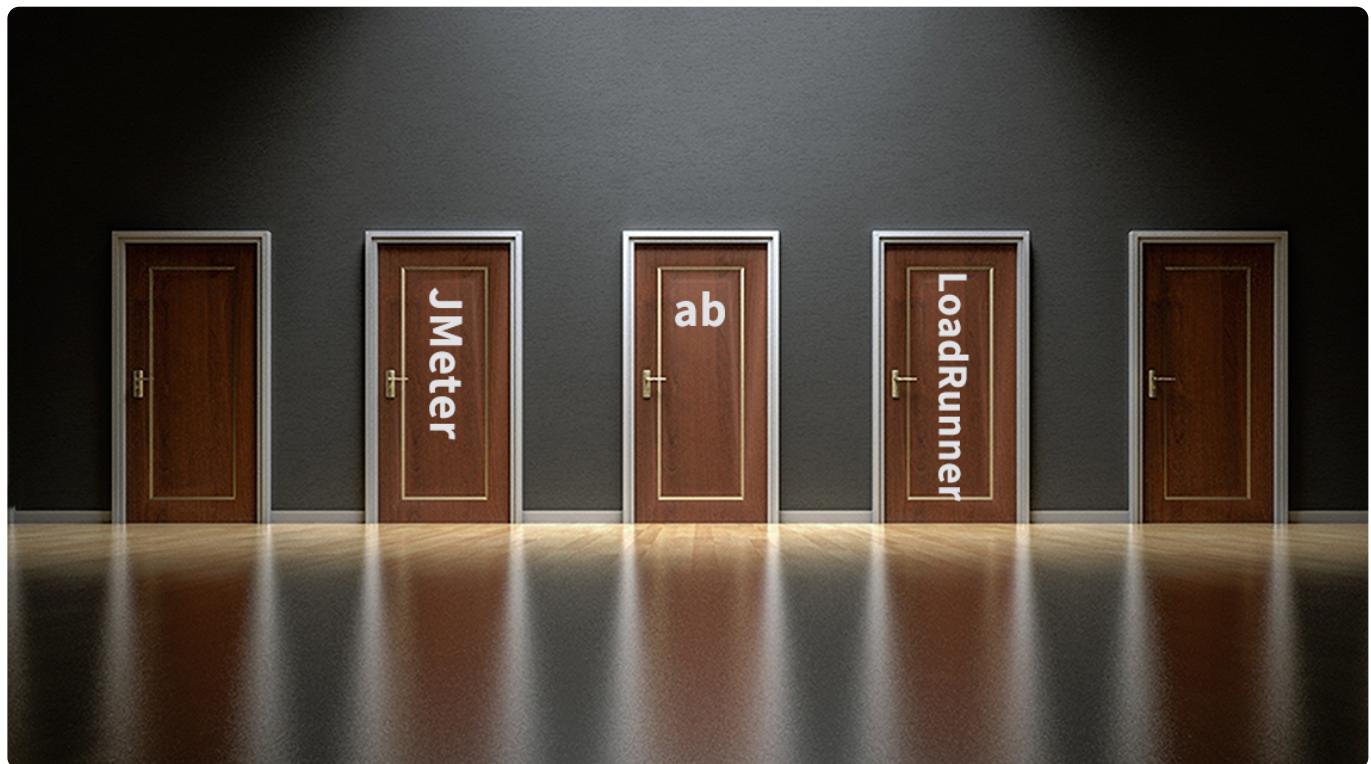
maxThreads：专门处理 I/O 操作的 Worker 线程数量，默认是 200，可以根据实际的环境来调整该参数，但不一定越大越好。

acceptCount : Tomcat 的 Acceptor 线程是负责从 accept 队列中取出该 connection , 然后交给工作线程去执行相关操作 , 这里的 acceptCount 指的是 accept 队列的大小。

当 Http 关闭 keep alive , 在并发量比较大时 , 可以适当地调大这个值。而在 Http 开启 keep alive 时 , 因为 Worker 线程数量有限 , Worker 线程就可能因长时间被占用 , 而连接在 accept 队列中等待超时。如果 accept 队列过大 , 就容易浪费连接。

maxConnections : 表示有多少个 socket 连接到 Tomcat 上。在 BIO 模式中 , 一个线程只能处理一个连接 , 一般 maxConnections 与 maxThreads 的值大小相同 ; 在 NIO 模式中 , 一个线程同时处理多个连接 , maxConnections 应该设置得比 maxThreads 要大的多 , 默认是 10000。

推荐几款常用的性能测试工具



熟练掌握一款性能测试工具，是我们必备的一项技能。他不仅可以帮助我们模拟测试场景（包括并发、复杂的组合场景），还能将测试结果转化成数据或图形，帮助我们更直观地了解系统性能。

常用的性能测试工具

常用的性能测试工具有很多，在这里我将列举几个比较实用的。

对于开发人员来说，首选是一些开源免费的性能（压力）测试软件，例如 ab (ApacheBench)、JMeter 等；对于专业的测试团队来说，付费版的 LoadRunner 是

首选。当然，也有很多公司是自行开发了一套量身定做的性能测试软件，优点是定制化强，缺点则是通用性差。

接下来，我会为你重点介绍 ab 和 JMeter 两款测试工具的特点以及常规的使用方法。

1.ab

ab 测试工具是 Apache 提供的一款测试工具，具有简单易上手的特点，在测试 Web 服务时非常实用。

ab 可以在 Windows 系统中使用，也可以在 Linux 系统中使用。这里说下在 Linux 系统中的安装方法，非常简单，只需要在 Linux 系统中输入 `yum-y install httpd-tools` 命令，就可以了。

安装成功后，输入 ab 命令，可以看到以下提示：

ab 工具用来测试 post get 接口请求非常便捷，可以通过参数指定请求数、并发数、请求参数等。例如，一个测试并发用户数为 10、请求数量为 100 的的 post 请求输入如下：

 复制代码

```
1 ab -n 100 -c 10 -p 'post.txt' -T 'application/x-www-form-urlencoded' 'http://test.api..
```

post.txt 为存放 post 参数的文档，存储格式如下：

 复制代码

```
1 username=test&password=test&sex=1
```

附上几个常用参数的含义：

-n : 总请求次数（最小默认为 1）；

-c : 并发次数（最小默认为 1 且不能大于总请求次数，例如：10 个请求，10 个并发，实际就是 1 人请求 1 次）；

-p : post 参数文档路径（-p 和 -T 参数要配合使用）；

-T : header 头内容类型（此处切记是大写英文字母 T）。

当我们测试一个 get 请求接口时，可以直接在链接的后面带上请求的参数：

 复制代码

```
1 ab -c 10 -n 100 http://www.test.api.com/test/login?userName=test&password=test
```

输出结果如下：

```
[root@localhost .ssh]# ab -c 10 -n 100 http://localhost:8061/test/login?userName=test&password=test
[2] 15970
[root@localhost .ssh]# This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost (be patient).....done

Server Software:           nginx/1.14.0
Server Hostname:          localhost
Server Port:              8061

Document Path:            /test/login?userName=test
Document Length:          53 bytes

Concurrency Level:        10
Time taken for tests:    0.106 seconds
Complete requests:       100
Failed requests:          0
Write errors:             0
Total transferred:       17200 bytes
HTML transferred:         5300 bytes
Requests per second:      942.01 [#/sec] (mean)
Time per request:         10.616 [ms] (mean)
Time per request:         1.062 [ms] (mean, across all concurrent requests)
Transfer rate:            158.23 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0.1     0.1    0     1
Processing:    3    10.6    6.0    8    38
Waiting:       3    9.4    4.6    8    26
Total:         3    10.6    6.0    8    38

Percentage of the requests served within a certain time (ms)
  50%    8
  66%   10
  75%   12
  80%   13
  90%   19
  95%   24
  98%   27
  99%   38
100%   38 (longest request)
```

以上输出中，有几项性能指标可以提供给你参考使用：

Requests per second：吞吐率，指某个并发用户数下单位时间内处理的请求数；

Time per request：上面的是用户平均请求等待时间，指处理完成所有请求数所花费的时间 / (总请求数 / 并发用户数) ；

Time per request：下面的是服务器平均请求处理时间，指处理完成所有请求数所花费的时间 / 总请求数；

Percentage of the requests served within a certain time：每秒请求时间分布情况，指在整个请求中，每个请求的时间长度的分布情况，例如有 50% 的请求响应在 8ms 内，66% 的请求响应在 10ms 内，说明有 16% 的请求在 8ms~10ms 之间。

2.JMeter

JMeter 是 Apache 提供的一款功能性比较全的性能测试工具，同样可以在 Windows 和 Linux 环境下安装使用。

JMeter 在 Windows 环境下使用了图形界面，可以通过图形界面来编写测试用例，具有易学和易操作的特点。

JMeter 不仅可以实现简单的并发性能测试，还可以实现复杂的宏基准测试。我们可以通过录制脚本的方式，在 JMeter 实现整个业务流程的测试。JMeter 也支持通过 csv 文件导入参数变量，实现用多样化的参数测试系统性能。

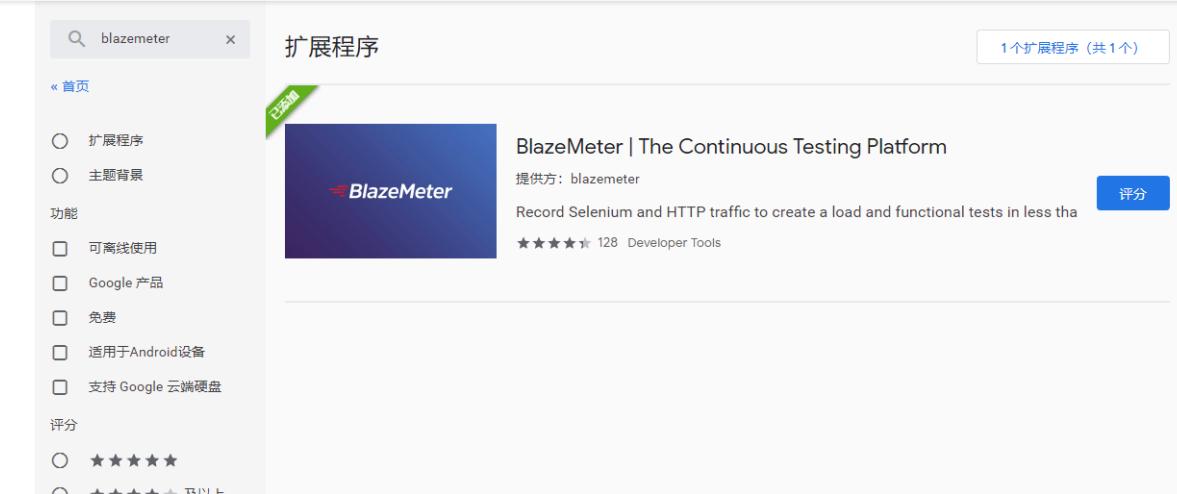
Windows 下的 JMeter 安装非常简单，在官网下载安装包，解压后即可使用。如果你需要打开图形化界面，那就进入到 bin 目录下，找到 jmeter.bat 文件，双击运行该文件就可以了。

这台电脑 > 本地磁盘 (D:) > apache-jmeter-5.0 > apache-jmeter-5.0 > bin			
	名称	修改日期	类型
	examples	2018/9/14 15:46	文件夹
	report-template	2018/9/14 15:46	文件夹
	templates	2018/9/14 15:46	文件夹
e	ApacheJMeter.jar	2018/9/14 16:25	Executable Jar File 13 KB
当	BeanShellAssertion.bshrc	2018/9/14 16:35	BSHRC 文件 2 KB
挡	BeanShellFunction.bshrc	2018/9/14 16:35	BSHRC 文件 3 KB
	BeanShellListeners.bshrc	2018/9/14 16:35	BSHRC 文件 2 KB
	BeanShellSampler.bshrc	2018/9/14 16:35	BSHRC 文件 3 KB
	create-rmi-keystore.bat	2018/9/14 15:46	Windows 批处理... 2 KB
	create-rmi-keystore.sh	2018/9/14 15:46	SH 文件 2 KB
	hc.parameters	2018/9/14 16:35	PARAMETERS 文... 2 KB
	headdump.cmd	2018/9/14 15:46	Windows 命令脚本 2 KB
	headdump.sh	2018/9/14 15:46	SH 文件 2 KB
	jaas.conf	2018/9/14 16:35	CONF 文件 2 KB
	jmeter	2018/9/14 16:05	文件 9 KB
	jmeter.bat	2018/9/14 16:05	Windows 批处理... 9 KB
	jmeter.log	2019/5/26 11:58	文本文档 0 KB
	jmeter.properties	2018/9/14 16:35	PROPERTIES 文件 54 KB
	jmeter.sh	2018/9/14 15:46	SH 文件 4 KB
t (C)	jmeter-n.cmd	2018/9/14 16:05	Windows 命令脚本 2 KB
	jmeter-n-r.cmd	2018/9/14 15:46	Windows 命令脚本 2 KB

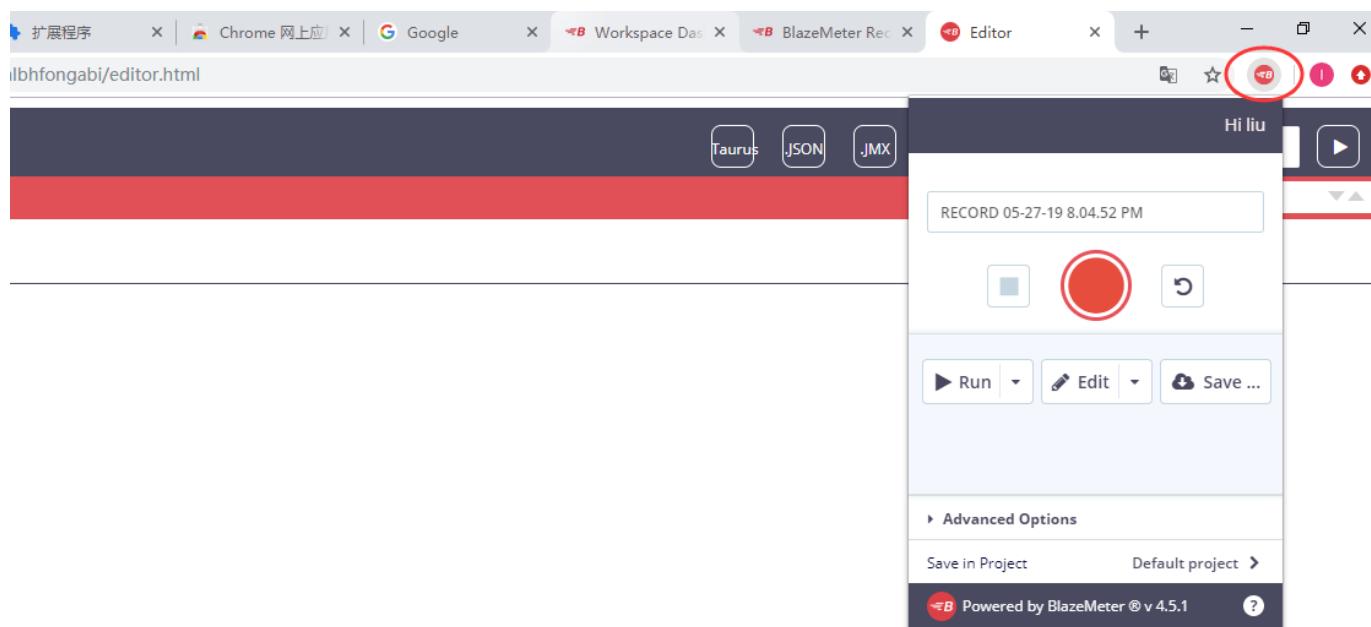
JMeter 的功能非常全面，我在这里简单介绍下如何录制测试脚本，并使用 JMeter 测试业务的性能。

录制 JMeter 脚本的方法有很多，一种是使用 Jmeter 自身的代理录制，另一种是使用 Badboy 这款软件录制，还有一种是我下面要讲的，通过安装浏览器插件的方式实现脚本的录制，这种方式非常简单，不用做任何设置。

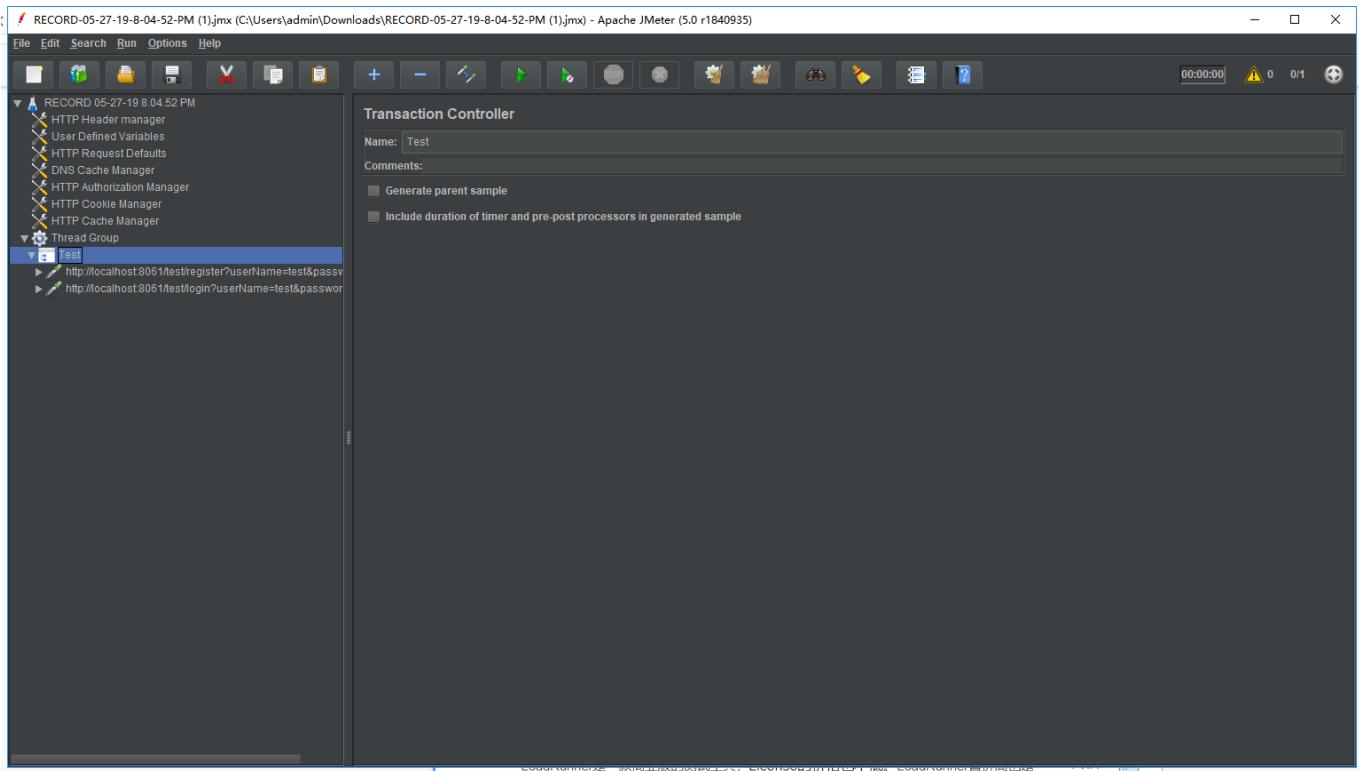
首先我们安装一个录制测试脚本的插件，叫做 BlazeMeter 插件。你可以在 Chrome 应用商店中找到它，然后点击安装，如图所示：



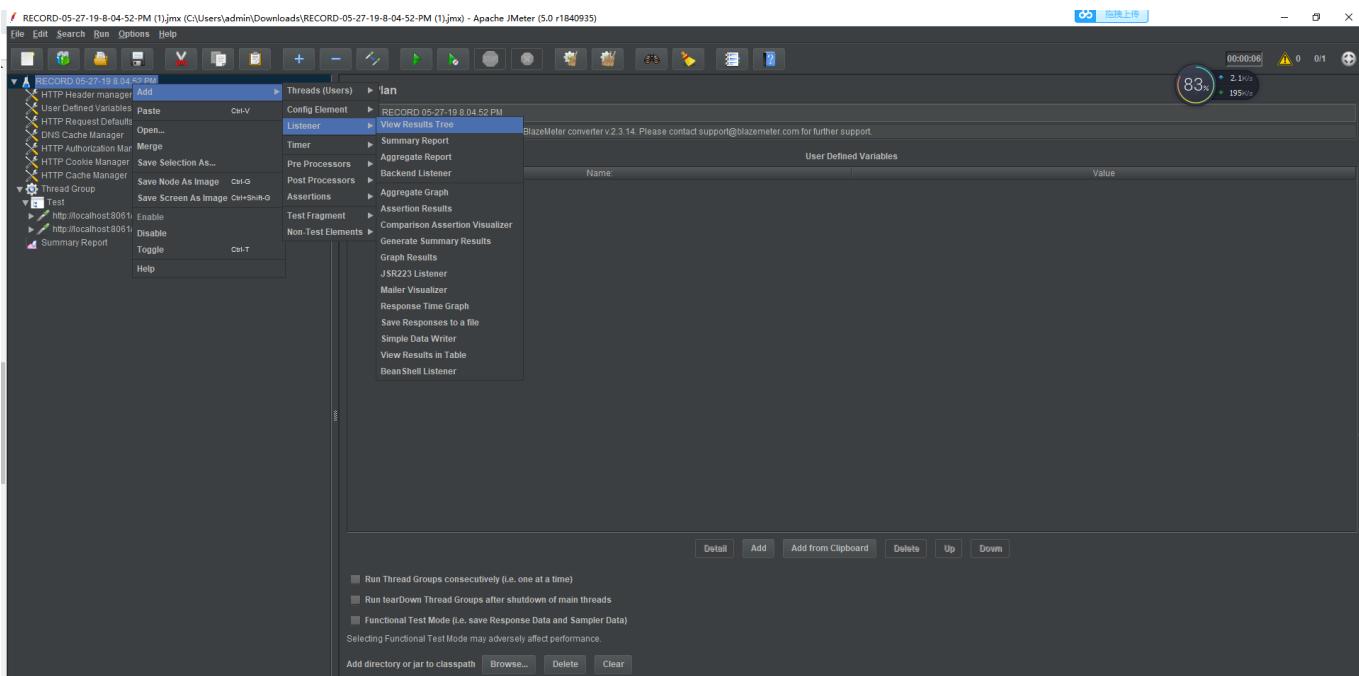
然后使用谷歌账号登录这款插件，如果不登录，我们将无法生成 JMeter 文件，安装以及登录成功后的界面如下图所示：



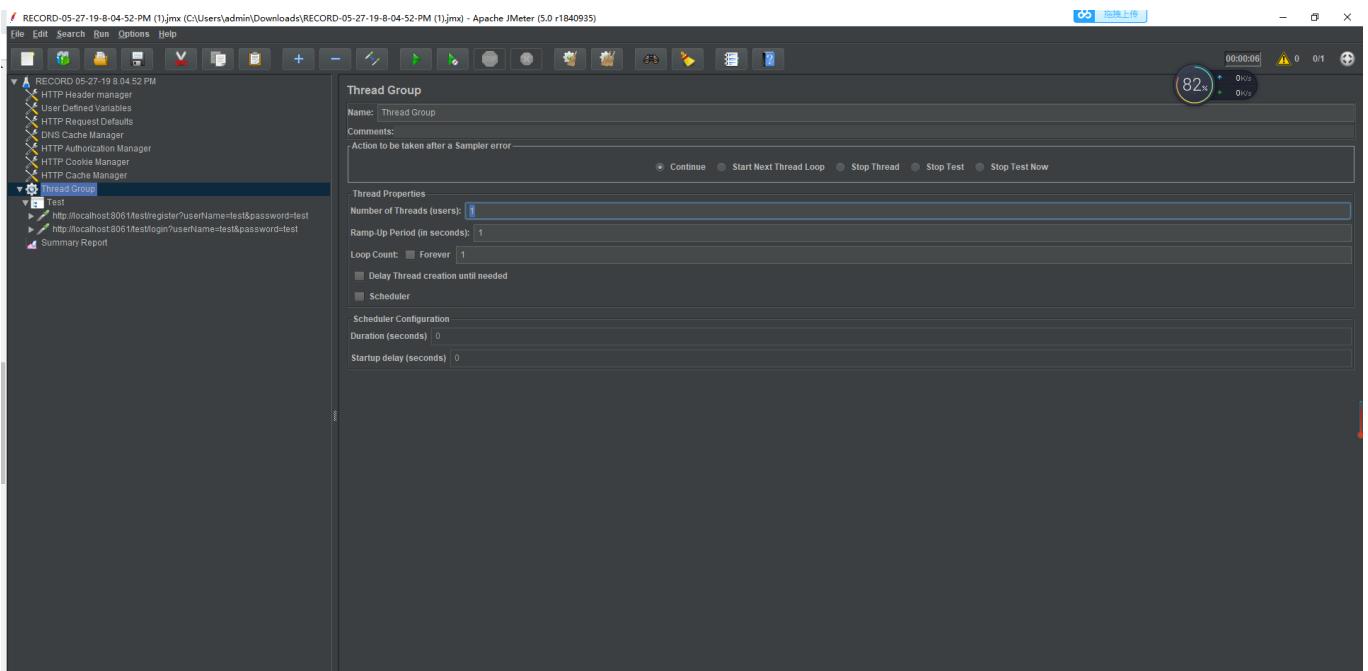
最后点击开始，就可以录制脚本了。录制成功后，点击保存为 JMX 文件，我们就可以通过 JMeter 打开这个文件，看到录制的脚本了，如下图所示：



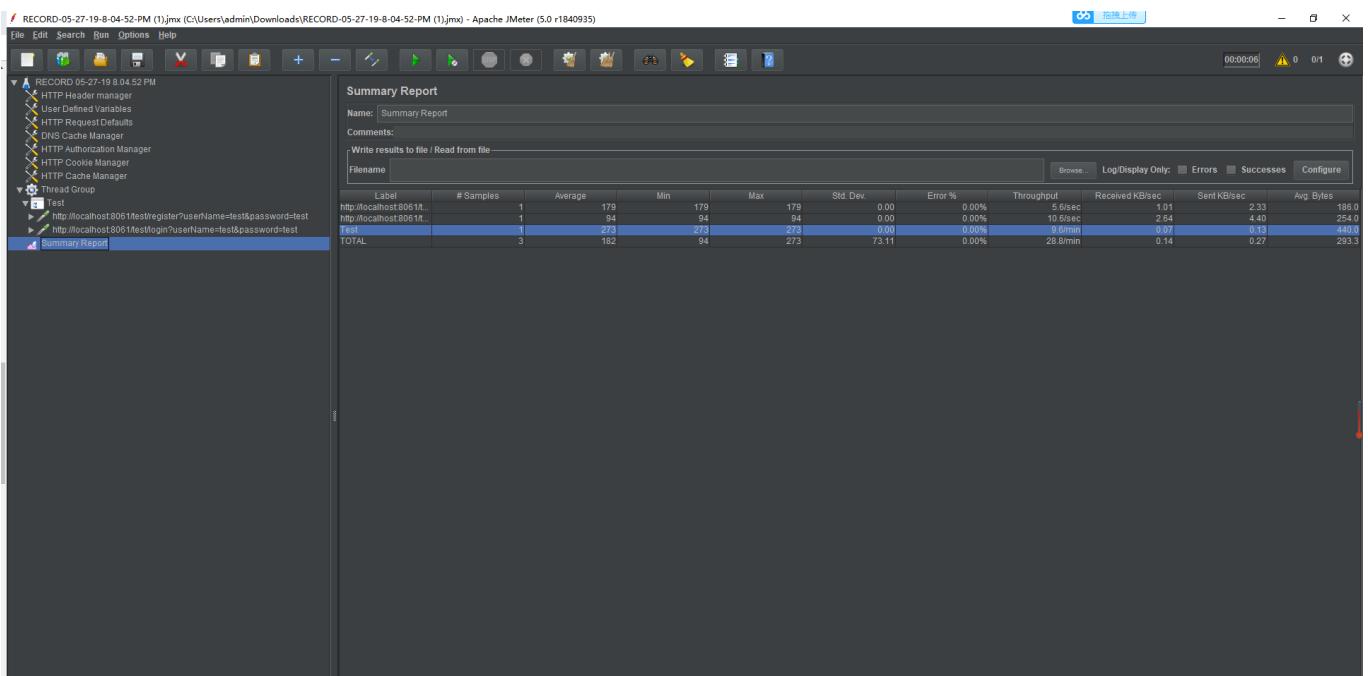
这个时候，我们还需要创建一个查看结果树，用来可视化查看运行的性能结果集合：



设置好结果树之后，我们可以对线程组的并发用户数以及循环调用次数进行设置：



设置成功之后，点击运行，我们可以看到运行的结果：



JMeter 的测试结果与 ab 的测试结果的指标参数差不多，这里我就不再重复讲解了。

3.LoadRunner

LoadRunner 是一款商业版的测试工具，并且 License 的售价不低。

作为一款专业的性能测试工具，LoadRunner 在性能压测时，表现得非常稳定和高效。相比 JMeter，LoadRunner 可以模拟出不同的内网 IP 地址，通过分配不同的 IP 地址给测试的用户，模拟真实环境下的用户。这里我就不展开详述了。

总结

三种常用的性能测试工具就介绍完了，最后我把今天的主要内容为你总结了一张图。



现在测试工具非常多，包括阿里云的 PTS 测试工具也很好用，但每款测试工具其实都有自己的优缺点。个人建议，还是在熟练掌握其中一款测试工具的前提下，再去探索其他测试工具的使用方法会更好。

12 | 多线程之锁优化（上）：深入了解Synchronized同步锁的优化方法



在并发编程中，多个线程访问同一个共享资源时，我们必须考虑如何维护数据的原子性。在 JDK1.5 之前，Java 是依靠 Synchronized 关键字实现锁功能来做到这点的。Synchronized 是 JVM 实现的一种内置锁，锁的获取和释放是由 JVM 隐式实现。

到了 JDK1.5 版本，并发包中新增了 Lock 接口来实现锁功能，它提供了与 Synchronized 关键字类似的同步功能，只是在使用时需要显示获取和释放锁。

Lock 同步锁是基于 Java 实现的，而 Synchronized 是基于底层操作系统的 Mutex Lock 实现的，每次获取和释放锁操作都会带来用户态和内核态的切换，从而增加系统性能开销。

因此，在锁竞争激烈的情况下，Synchronized 同步锁在性能上就表现得非常糟糕，它也常被大家称为重量级锁。

特别是在单个线程重复申请锁的情况下，JDK1.5 版本的 Synchronized 锁性能要比 Lock 的性能差很多。例如，在 Dubbo 基于 Netty 实现的通信中，消费端向服务端通信之后，由于接收返回消息是异步，所以需要一个线程轮询监听返回信息。而在接收消息时，就需要用到锁来确保 request session 的原子性。如果我们这里使用 Synchronized 同步锁，那么每当同一个线程请求锁资源时，都会发生一次用户态和内核态的切换。

到了 JDK1.6 版本之后，Java 对 Synchronized 同步锁做了充分的优化，甚至在某些场景下，它的性能已经超越了 Lock 同步锁。这一讲我们就来看看 Synchronized 同步锁究竟是通过了哪些优化，实现了性能地提升。

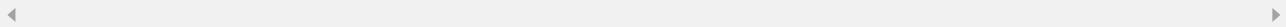
Synchronized 同步锁实现原理

了解 Synchronized 同步锁优化之前，我们先来看看它的底层实现原理，这样可以帮助我们更好地理解后面的内容。

通常 Synchronized 实现同步锁的方式有两种，一种是修饰方法，一种是修饰方法块。以下就是通过 Synchronized 实现的两种同步方法加锁的方式：

 复制代码

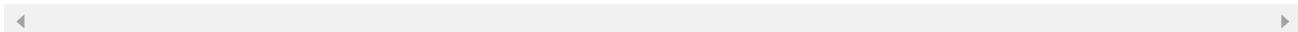
```
1 // 关键字在实例方法上，锁为当前实例
2     public synchronized void method1() {
3         // code
4     }
5
6 // 关键字在代码块上，锁为括号里面的对象
7     public void method2() {
8         Object o = new Object();
9         synchronized (o) {
10            // code
11        }
12    }
```



下面我们可以反编译看下具体字节码的实现，运行以下反编译命令，就可以输出我们想要的字节码：

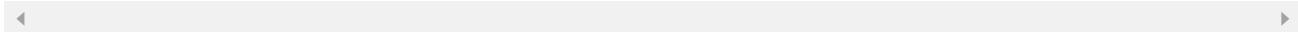
复制代码

```
1 javac -encoding UTF-8 SyncTest.java // 先运行编译 class 文件命令
```



复制代码

```
1 javap -v SyncTest.class // 再通过 javap 打印出字节文件
```

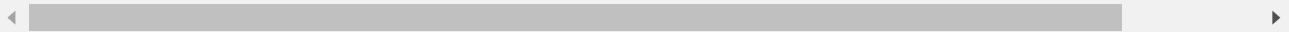


通过输出的字节码，你会发现：Synchronized 在修饰同步代码块时，是由 monitoreenter 和 monitorexit 指令来实现同步的。进入 monitoreenter 指令后，线程将持有 Monitor 对象，退出 monitoreenter 指令后，线程将释放该 Monitor 对象。

复制代码

```
1 public void method2();
2     descriptor: ()V
3     flags: ACC_PUBLIC
4     Code:
5         stack=2, locals=4, args_size=1
6             0: new           #2
7             3: dup
8             4: invokespecial #1
9             7: astore_1
10            8: aload_1
11            9: dup
12            10: astore_2
13            11: monitoreenter //monitoreenter 指令
14            12: aload_2
15            13: monitorexit  //monitorexit  指令
16            14: goto          22
17            17: astore_3
18            18: aload_2
19            19: monitorexit
20            20: aload_3
21            21: athrow
22            22: return
23     Exception table:
24         from   to   target type
25             12    14    17   any
26             17    20    17   any
27     LineNumberTable:
28         line 18: 0
29         line 19: 8
30         line 21: 12
31         line 22: 22
```

```
32     StackMapTable: number_of_entries = 2
33         frame_type = 255 /* full_frame */
34         offset_delta = 17
35         locals = [ class com/demo/io/SyncTest, class java/lang/Object, class java/lang/
36             stack = [ class java/lang/Throwable ]
37             frame_type = 250 /* chop */
38             offset_delta = 4
```

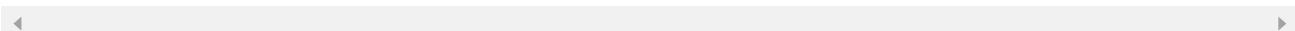


再来看以下同步方法的字节码，你会发现：当 Synchronized 修饰同步方法时，并没有发现 monitoreenter 和 monitorexit 指令，而是出现了一个 ACC_SYNCHRONIZED 标志。

这是因为 JVM 使用了 ACC_SYNCHRONIZED 访问标志来区分一个方法是否是同步方法。当方法调用时，调用指令将会检查该方法是否被设置 ACC_SYNCHRONIZED 访问标志。如果设置了该标志，执行线程将先持有 Monitor 对象，然后再执行方法。在该方法运行期间，其它线程将无法获取到该 Mointor 对象，当方法执行完成后，再释放该 Monitor 对象。

复制代码

```
1  public synchronized void method1();
2      descriptor: ()V
3      flags: ACC_PUBLIC, ACC_SYNCHRONIZED // ACC_SYNCHRONIZED 标志
4      Code:
5          stack=0, locals=1, args_size=1
6              0: return
7      LineNumberTable:
8          line 8: 0
9
```



通过以上的源码，我们再来看看 Synchronized 修饰方法是怎么实现锁原理的。

JVM 中的同步是基于进入和退出管程（Monitor）对象实现的。每个对象实例都会有一个 Monitor，Monitor 可以和对象一起创建、销毁。Monitor 是由 ObjectMonitor 实现，而 ObjectMonitor 是由 C++ 的 ObjectMonitor.hpp 文件实现，如下所示：

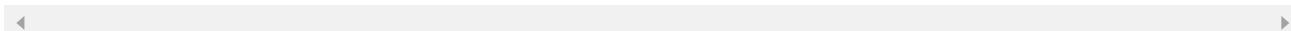
复制代码

```
1 ObjectMonitor() {
2     _header = NULL;
3     _count = 0; // 记录个数
```

```

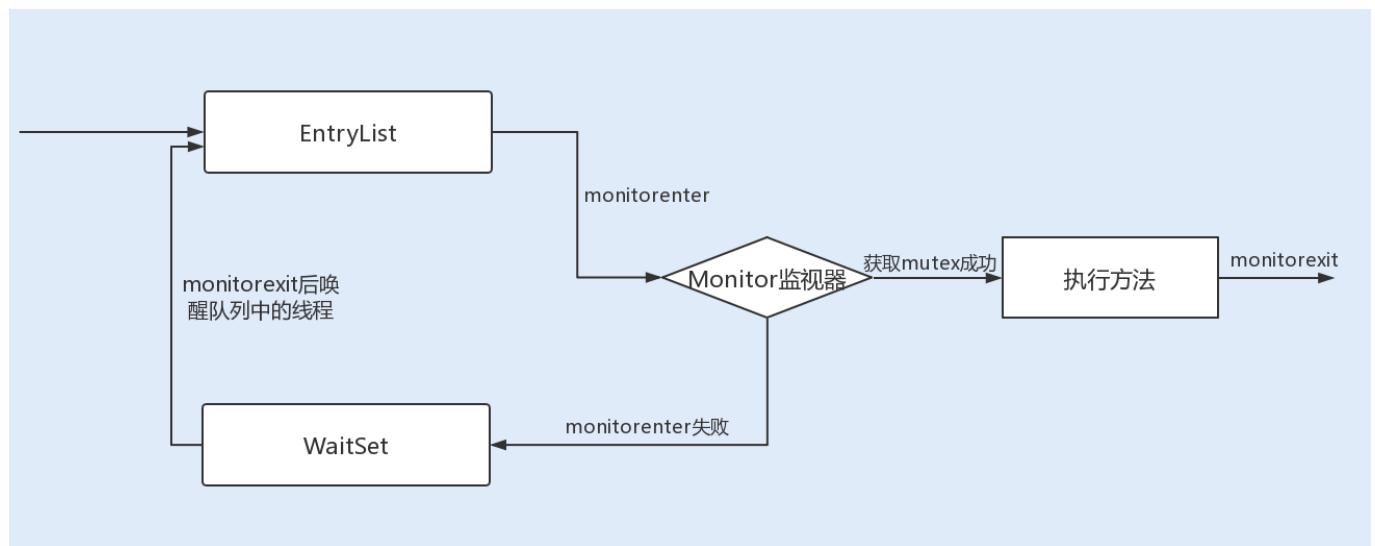
4     _waiters = 0,
5     _recursions = 0;
6     _object = NULL;
7     _owner = NULL;
8     _WaitSet = NULL; // 处于 wait 状态的线程，会被加入到 _WaitSet
9     _WaitSetLock = 0 ;
10    _Responsible = NULL ;
11    _succ = NULL ;
12    _cxq = NULL ;
13    FreeNext = NULL ;
14    _EntryList = NULL ; // 处于等待锁 block 状态的线程，会被加入到该列表
15    _SpinFreq = 0 ;
16    _SpinClock = 0 ;
17    OwnerIsThread = 0 ;
18 }

```



当多个线程同时访问一段同步代码时，多个线程会先被存放在 EntryList 集合中，处于 block 状态的线程，都会被加入到该列表。接下来当线程获取到对象的 Monitor 时，Monitor 是依靠底层操作系统的 Mutex Lock 来实现互斥的，线程申请 Mutex 成功，则持有该 Mutex，其它线程将无法获取到该 Mutex。

如果线程调用 wait() 方法，就会释放当前持有的 Mutex，并且该线程会进入 WaitSet 集合中，等待下一次被唤醒。如果当前线程顺利执行完方法，也将释放 Mutex。



看完上面的讲解，相信你对同步锁的实现原理已经有个深入的了解了。总结来说就是，同步锁在这种实现方式中，因 Monitor 是依赖于底层的操作系统实现，存在用户态与内核态之间的切换，所以增加了性能开销。

锁升级优化

为了提升性能，JDK1.6 引入了偏向锁、轻量级锁、重量级锁概念，来减少锁竞争带来的上下文切换，而正是新增的 Java 对象头实现了锁升级功能。

当 Java 对象被 Synchronized 关键字修饰成为同步锁后，围绕这个锁的一系列升级操作都将和 Java 对象头有关。

Java 对象头

在 JDK1.6 JVM 中，对象实例在堆内存中被分为了三个部分：对象头、实例数据和对齐填充。其中 Java 对象头由 Mark Word、指向类的指针以及数组长度三部分组成。

Mark Word 记录了对象和锁有关的信息。Mark Word 在 64 位 JVM 中的长度是 64bit，我们可以一起看下 64 位 JVM 的存储结构是怎么样的。如下图所示：

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

锁升级功能主要依赖于 Mark Word 中的锁标志位和释放偏向锁标志位，Synchronized 同步锁就是从偏向锁开始的，随着竞争越来越激烈，偏向锁升级到轻量级锁，最终升级到重量级锁。下面我们就沿着这条优化路径去看下具体的内容。

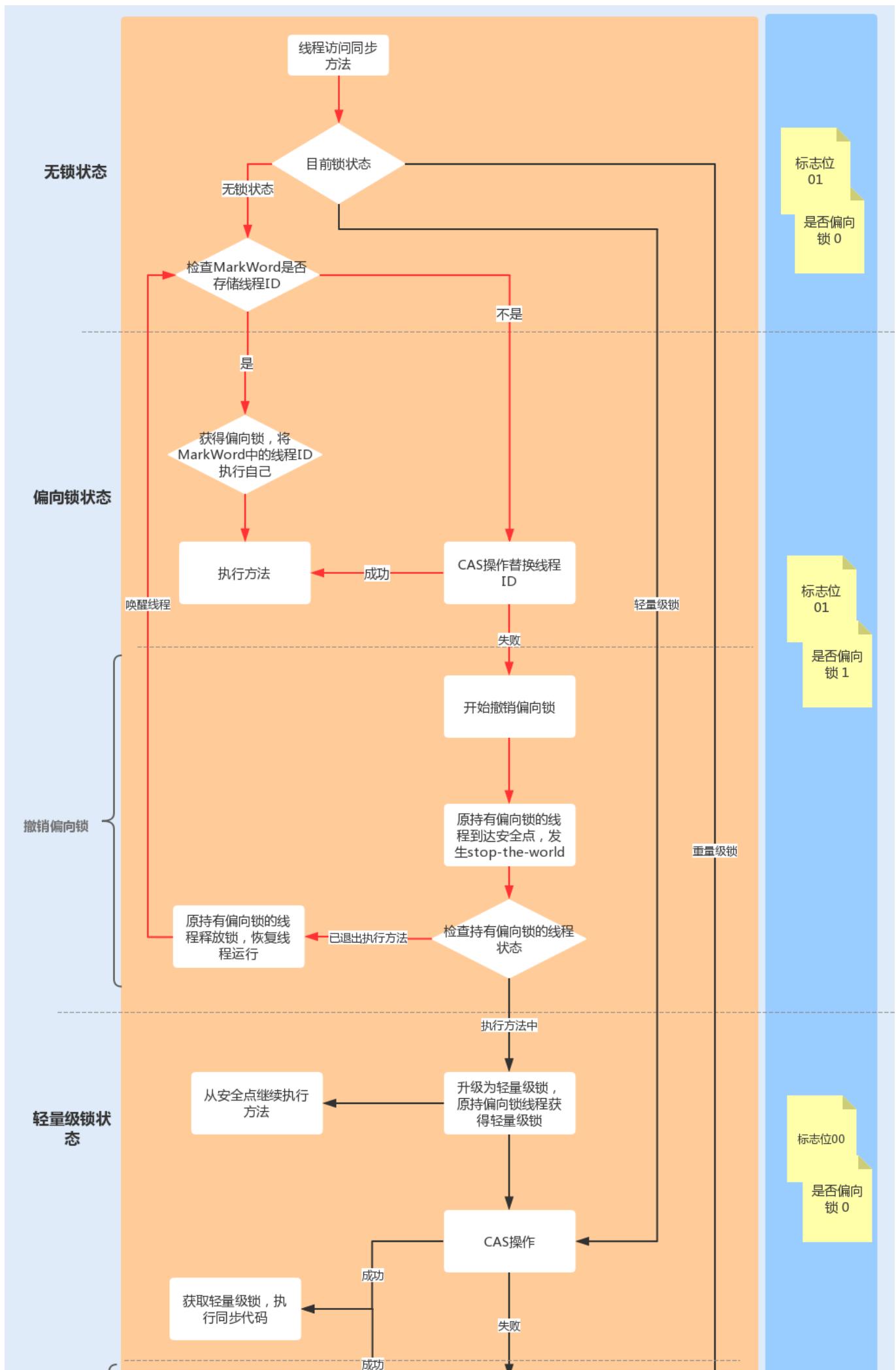
1. 偏向锁

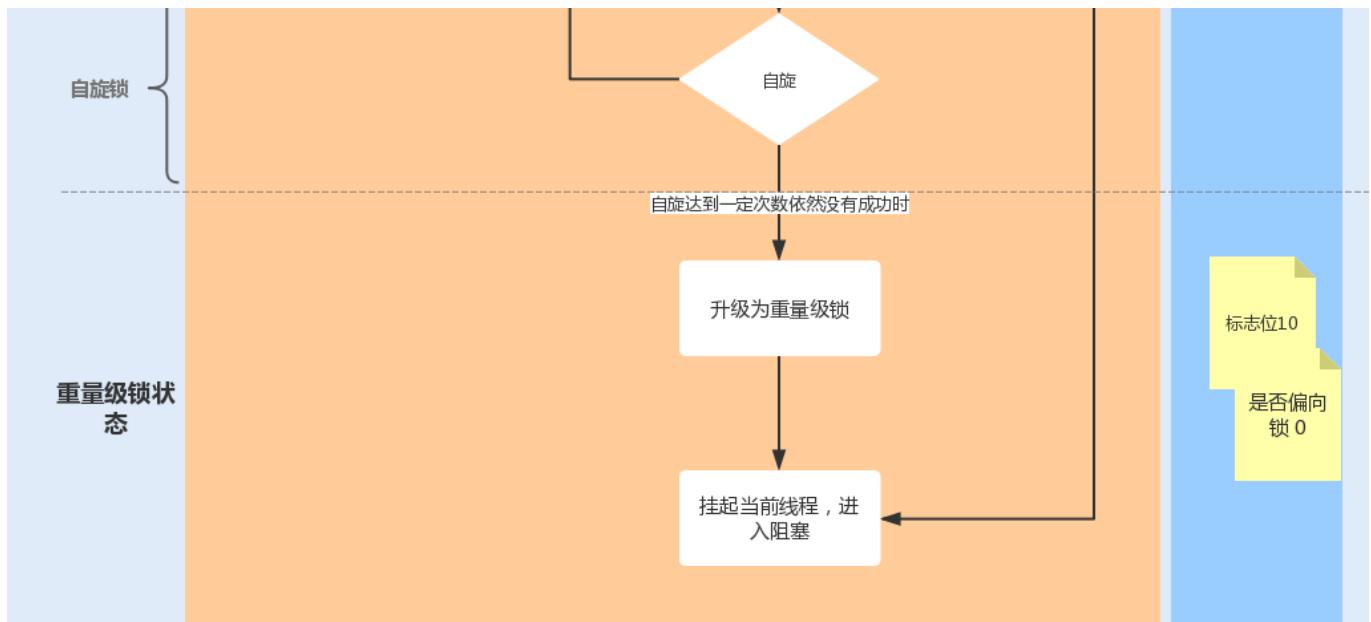
偏向锁主要用来优化同一线程多次申请同一个锁的竞争。在某些情况下，大部分时间是同一个线程竞争锁资源，例如，在创建一个线程并在线程中执行循环监听的场景下，或单线程操作一个线程安全集合时，同一线程每次都需要获取和释放锁，每次操作都会发生用户态与内核态的切换。

偏向锁的作用就是，当一个线程再次访问这个同步代码或方法时，该线程只需去对象头的 Mark Word 中去判断一下是否有偏向锁指向它的 ID，无需再进入 Monitor 去竞争对象了。当对象被当做同步锁并有一个线程抢到了锁时，锁标志位还是 01，“是否偏向锁”标志位设置为 1，并且记录抢到锁的线程 ID，表示进入偏向锁状态。

一旦出现其它线程竞争锁资源时，偏向锁就会被撤销。偏向锁的撤销需要等待全局安全点，暂停持有该锁的线程，同时检查该线程是否还在执行该方法，如果是，则升级锁，反之则被其它线程抢占。

下图中红线流程部分为偏向锁获取和撤销流程：





因此，在高并发场景下，当大量线程同时竞争同一个锁资源时，偏向锁就会被撤销，发生 stop the word 后，开启偏向锁无疑会带来更大的性能开销，这时我们可以通过添加 JVM 参数关闭偏向锁来调优系统性能，示例代码如下：

复制代码

```
1 -XX:-UseBiasedLocking // 关闭偏向锁（默认打开）
```

或

复制代码

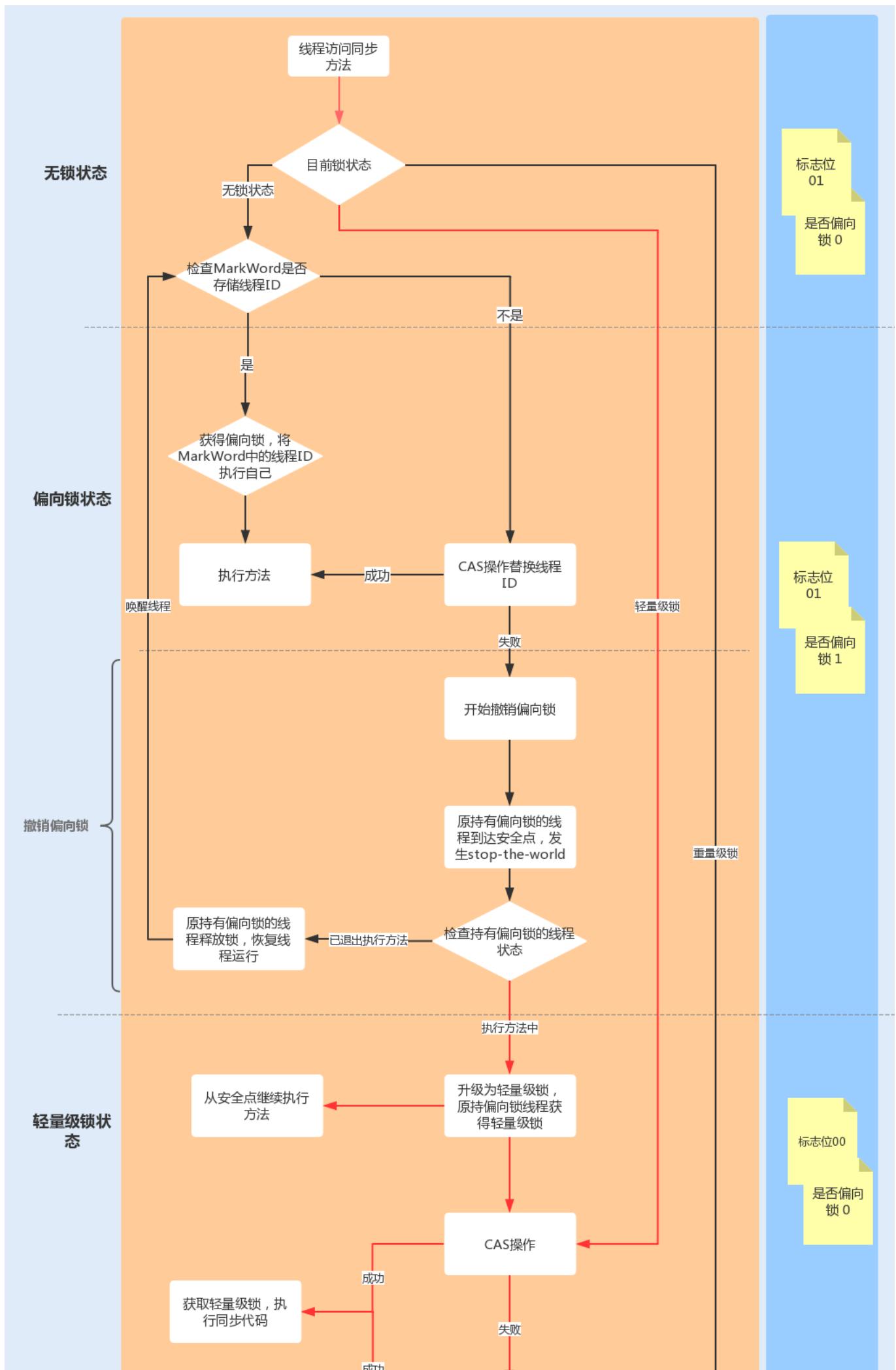
```
1 -XX:+UseHeavyMonitors // 设置重量级锁
```

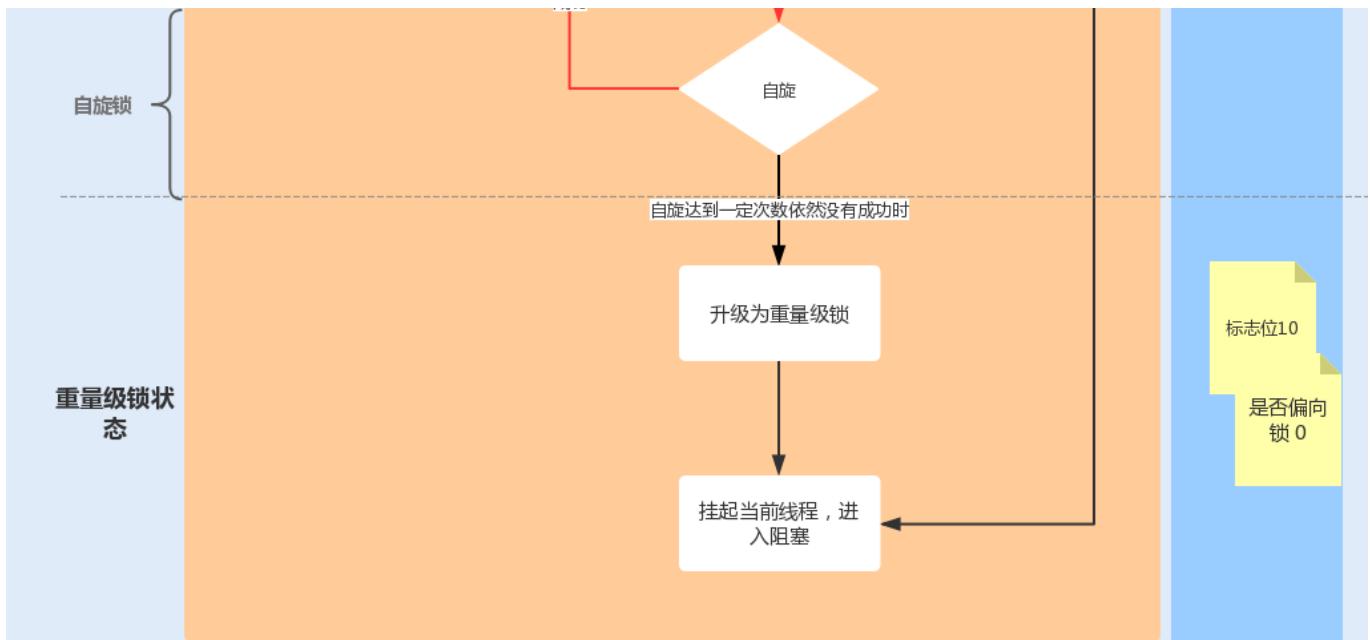
2. 轻量级锁

当有另外一个线程竞争获取这个锁时，由于该锁已经是偏向锁，当发现对象头 Mark Word 中的线程 ID 不是自己的线程 ID，就会进行 CAS 操作获取锁，如果获取成功，直接替换 Mark Word 中的线程 ID 为自己的 ID，该锁会保持偏向锁状态；如果获取锁失败，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。

轻量级锁适用于线程交替执行同步块的场景，绝大部分的锁在整个同步周期内都不存在长时间的竞争。

下图中红线流程部分为升级轻量级锁及操作流程：





3. 自旋锁与重量级锁

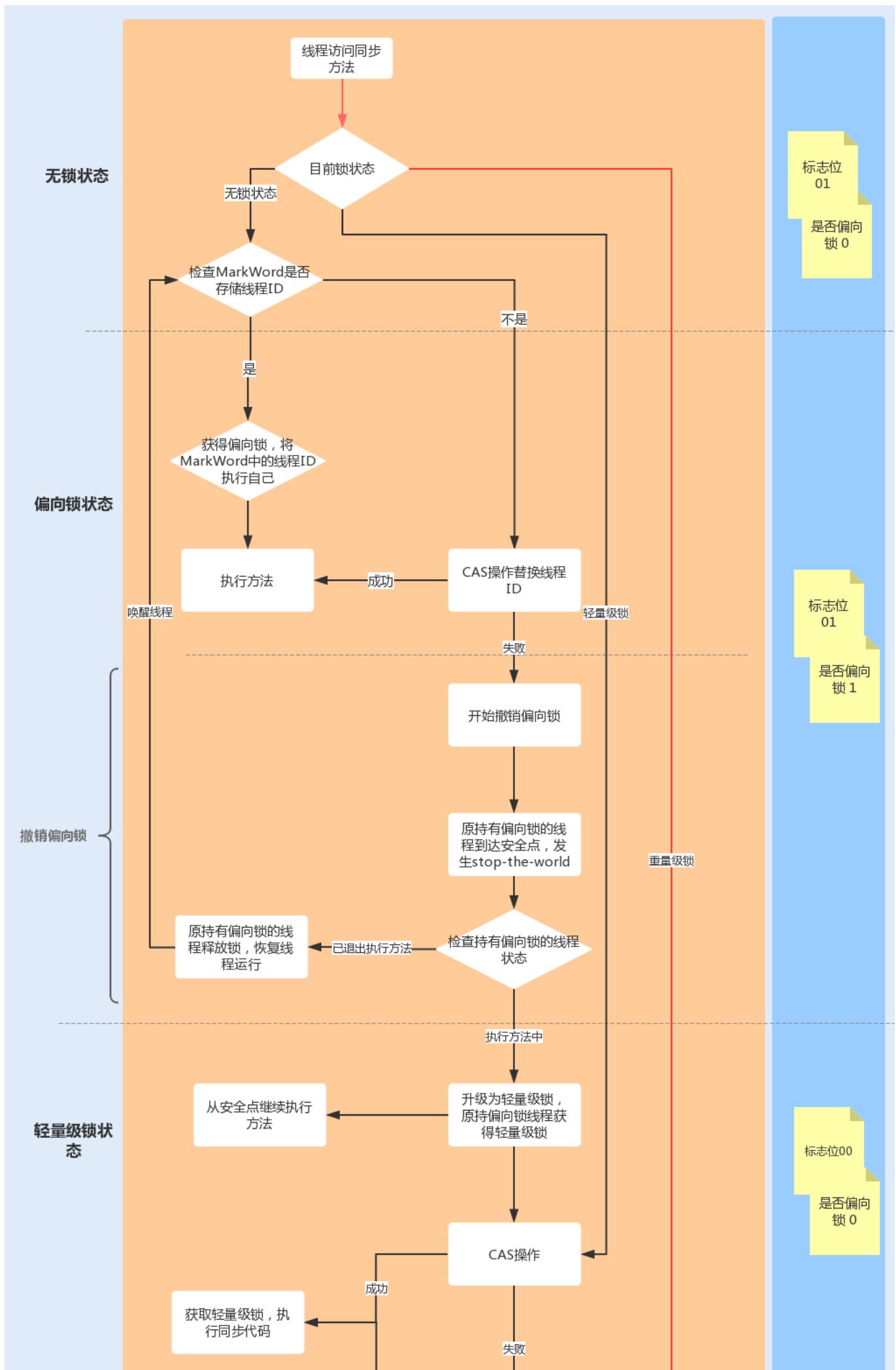
轻量级锁 CAS 抢锁失败，线程将会被挂起进入阻塞状态。如果正在持有锁的线程在很短的时间内释放资源，那么进入阻塞状态的线程无疑又要申请锁资源。

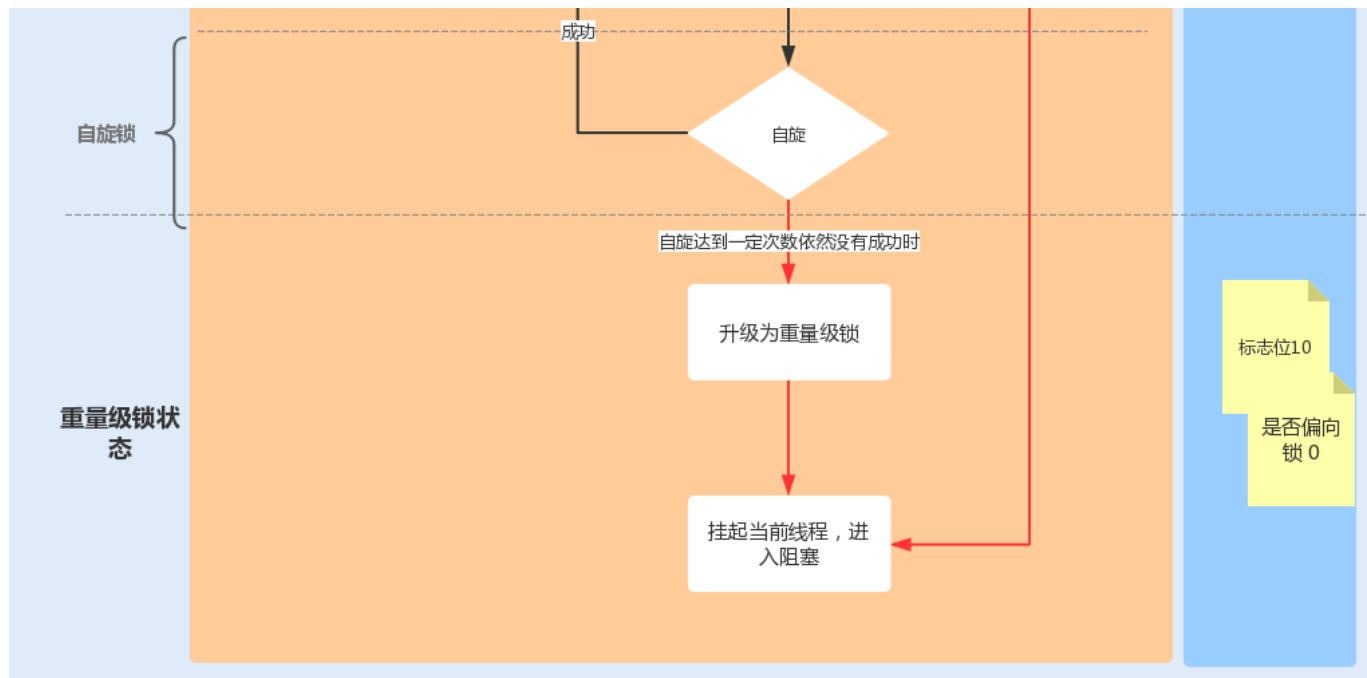
JVM 提供了一种自旋锁，可以通过自旋方式不断尝试获取锁，从而避免线程被挂起阻塞。这是基于大多数情况下，线程持有锁的时间都不会太长，毕竟线程被挂起阻塞可能会得不偿失。

从 JDK1.7 开始，自旋锁默认启用，自旋次数由 JVM 设置决定，这里我不建议设置的重试次数过多，因为 CAS 重试操作意味着长时间地占用 CPU。

自旋锁重试之后如果抢锁依然失败，同步锁就会升级至重量级锁，锁标志位改为 10。在这个状态下，未抢到锁的线程都会进入 Monitor，之后会被阻塞在 _WaitSet 队列中。

下图中红线流程部分为自旋后升级为重量级锁的流程：





在锁竞争不激烈且锁占用时间非常短的场景下，自旋锁可以提高系统性能。一旦锁竞争激烈或锁占用的时间过长，自旋锁将会导致大量的线程一直处于 CAS 重试状态，占用 CPU 资源，反而会增加系统性能开销。所以自旋锁和重量级锁的使用都要结合实际场景。

在高负载、高并发的场景下，我们可以通过设置 JVM 参数来关闭自旋锁，优化系统性能，示例代码如下：

复制代码

```

1 -XX:-UseSpinning // 参数关闭自旋锁优化（默认打开）
2 -XX:PreBlockSpin // 参数修改默认的自旋次数。JDK1.7 后，去掉此参数，由 jvm 控制

```

动态编译实现锁消除 / 锁粗化

除了锁升级优化，Java 还使用了编译器对锁进行优化。JIT 编译器在动态编译同步块的时候，借助了一种被称为逃逸分析的技术，来判断同步块使用的锁对象是否只能够被一个线程访问，而没有被发布到其它线程。

确认是的话，那么 JIT 编译器在编译这个同步块的时候不会生成 synchronized 所表示的锁的申请与释放的机器码，即消除了锁的使用。在 Java7 之后的版本就不需要手动配置了，该操作可以自动实现。

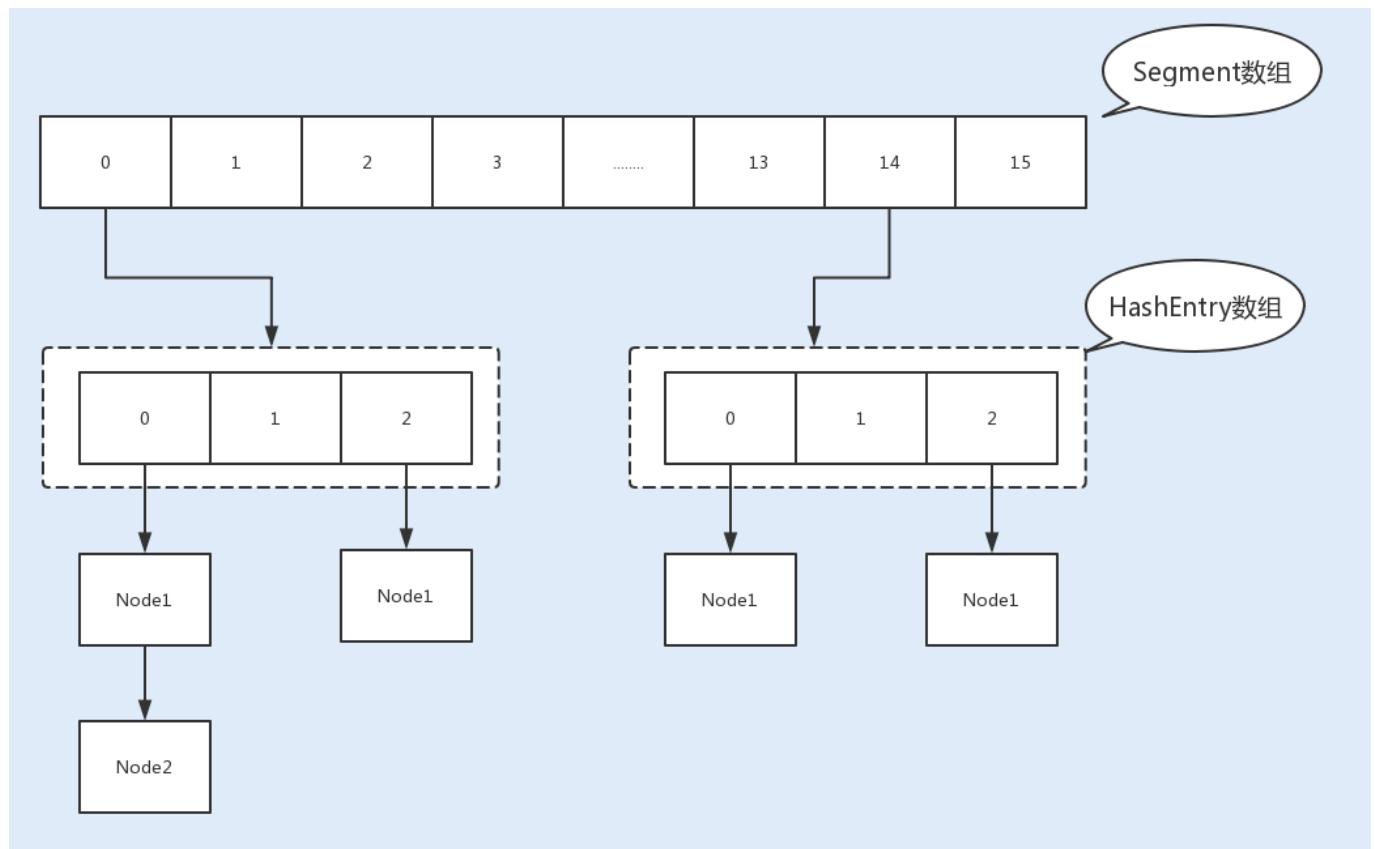
锁粗化同理，就是在 JIT 编译器动态编译时，如果发现几个相邻的同步块使用的是同一个锁实例，那么 JIT 编译器将会把这几个同步块合并为一个大的同步块，从而避免一个线程“反复申请、释放同一个锁”所带来的性能开销。

减小锁粒度

除了锁内部优化和编译器优化之外，我们还可以通过代码层来实现锁优化，减小锁粒度就是一种惯用的方法。

当我们的锁对象是一个数组或队列时，集中竞争一个对象的话会非常激烈，锁也会升级为重量级锁。我们可以考虑将一个数组和队列对象拆成多个小对象，来降低锁竞争，提升并行度。

最经典的减小锁粒度的案例就是 JDK1.8 之前实现的 ConcurrentHashMap 版本。我们知道，HashTable 是基于一个数组 + 链表实现的，所以在并发读写操作集合时，存在激烈的锁资源竞争，也因此性能会存在瓶颈。而 ConcurrentHashMap 就很巧妙地使用了分段锁 Segment 来降低锁资源竞争，如下图所示：



总结

JVM 在 JDK1.6 中引入了分级锁机制来优化 Synchronized，当一个线程获取锁时，首先对象锁将成为一个偏向锁，这样做是为了优化同一线程重复获取导致的用户态与内核态的切换问题；其次如果有多个线程竞争锁资源，锁将会升级为轻量级锁，它适用于在短时间内持有锁，且分锁有交替切换的场景；偏向锁还使用了自旋锁来避免线程用户态与内核态的频繁切换，大大地提高了系统性能；但如果锁竞争太激烈了，那么同步锁将会升级为重量级锁。

减少锁竞争，是优化 Synchronized 同步锁的关键。我们应该尽量使 Synchronized 同步锁处于轻量级锁或偏向锁，这样才能提高 Synchronized 同步锁的性能；通过减小锁粒度来降低锁竞争也是一种最常用的优化方法；另外我们还可以通过减少锁的持有时间来提高 Synchronized 同步锁在自旋时获取锁资源的成功率，避免 Synchronized 同步锁升级为重量级锁。

这一讲我们重点了解了 Synchronized 同步锁优化，这里由于字数限制，也为了你能更好地理解内容，目录中 12 讲的内容我拆成了两讲，在下一讲中，我会重点讲解 Lock 同步锁的优化方法。

13 | 多线程之锁优化（中）：深入了解Lock同步锁的优化方法



今天这讲我们继续来聊聊锁优化。上一讲我重点介绍了在JVM层实现的Synchronized同步锁的优化方法，除此之

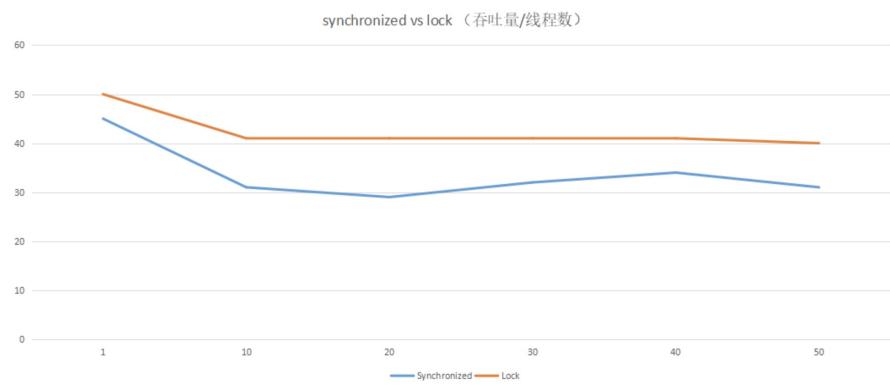
外，在 JDK1.5 之后，Java 还提供了 Lock 同步锁。那么它有什么优势呢？

相对于需要 JVM 隐式获取和释放锁的 Synchronized 同步锁，Lock 同步锁（以下简称 Lock 锁）需要的是显示获取和释放锁，这就为获取和释放锁提供了更多的灵活性。**Lock 锁的基本操作是通过乐观锁来实现的，但由于 Lock 锁也会在阻塞时被挂起，因此它依然属于悲观锁。**我们可以通过一张图来简单对比下两个同步锁，了解下各自的特点：

	Synchronized	Lock
实现方式	JVM层实现	Java底层代码实现
锁的获取	JVM隐式获取	Lock.lock(): 获取锁，如被锁定则等待。 Lock.tryLock(): 如未被锁定才获取锁。 Lock.tryLock(long timeout, TimeUnit unit): 获取锁，如已被锁定，则最多等待timeout时间后返回获 取锁状态。 Lock.lockInterruptibly(): 如当前线程未被interrupt才 获取锁。
锁的释放	JVM隐式释放	通过Lock.unlock(), 在finally中释放锁
锁的类型	非公平锁、可重入	非公平锁、公平锁、可重入
锁的状态	不可中断	可中断

从性能方面上来说，在并发量不高、竞争不激烈的情况下，Synchronized 同步锁由于具有分级锁的优势，性能上与 Lock 锁差不多；但在高负载、高并发的情况下，Synchronized 同步锁由于竞争激烈会升级到重量级锁，性能则没有 Lock 锁稳定。

我们可以通过一组简单的性能测试，直观地对比下两种锁的性能，结果见下方，代码可以在[Github](#)上下载查看。



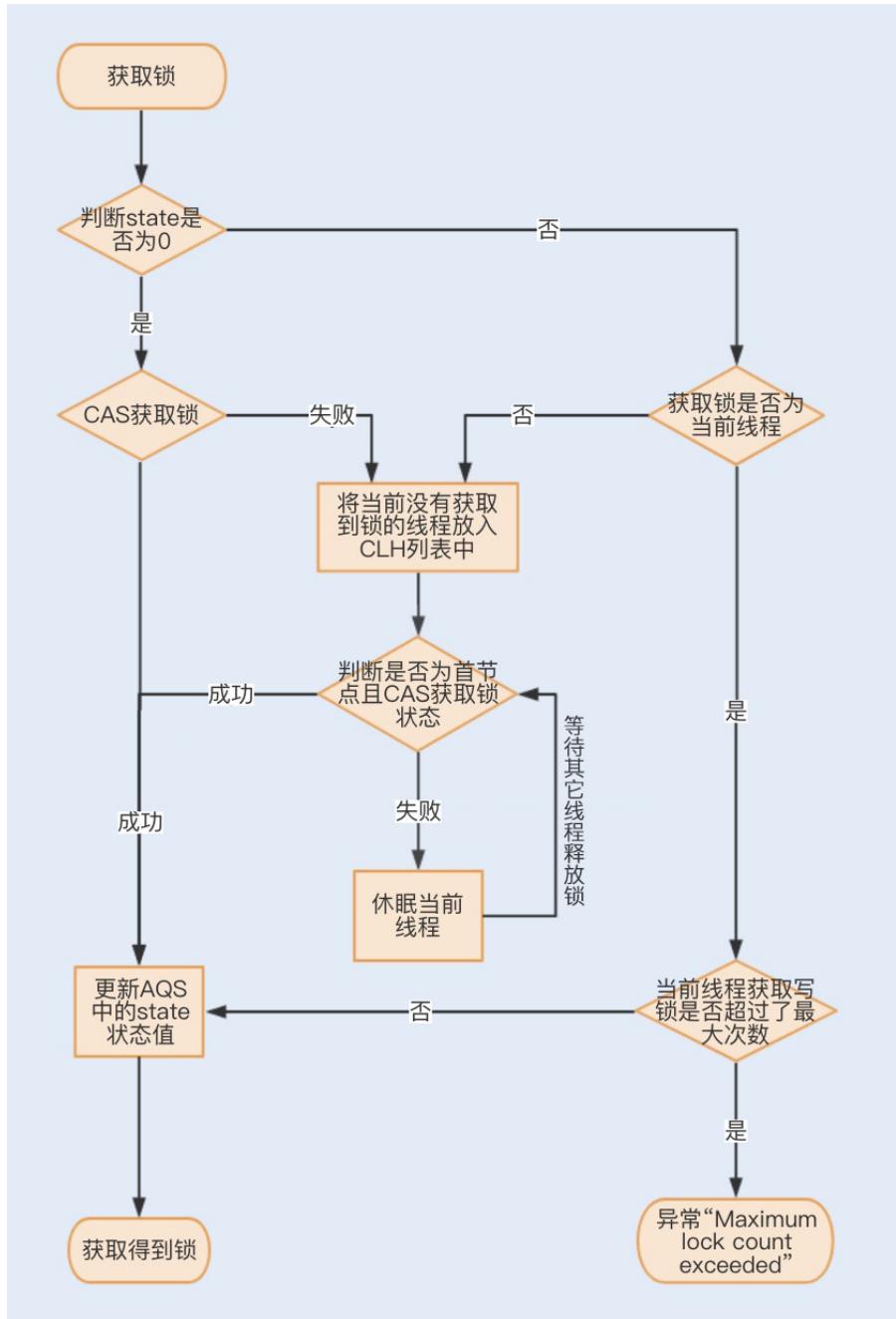
通过以上数据，我们可以发现：Lock 锁的性能相对来说更加稳定。那它与上一讲的 Synchronized 同步锁相比，实现原理又是怎样的呢？

Lock 锁的实现原理

Lock 锁是基于 Java 实现的锁，Lock 是一个接口类，常用的实现类有 ReentrantLock、ReentrantReadWriteLock (RRW) ，它们都是依赖 AbstractQueuedSynchronizer (AQS) 类实现的。

AQS 类结构中包含一个基于链表实现的等待队列 (CLH 队列) ，用于存储所有阻塞的线程，AQS 中还有一个 state 变量，该变量对 ReentrantLock 来说表示加锁状态。

该队列的操作均通过 CAS 操作实现，我们可以通过一张图来看下整个获取锁的流程。



锁分离优化 Lock 同步锁

虽然 Lock 锁的性能稳定，但也并不是所有的场景下都默认使用 ReentrantLock 独占锁来实现线程同步。

我们知道，对于同一份数据进行读写，如果一个线程在读数据，而另一个线程在写数据，那么读到的数据和最终的数据就会不一致；如果一个线程在写数据，而另一个线程也在写数据，那么线程前后看到的数据也会不一致。这个时候我们可以在读写方法中加入互斥锁，来保证任何时候只能有一个线程进行读或写操作。

在大部分业务场景中，读业务操作要远远大于写业务操作。而在多线程编程中，读操作并不会修改共享资源的数据，如果多个线程仅仅是读取共享资源，那么这种情况下其实没有必要对资源进行加锁。如果使用互斥锁，反倒会影响业务的并发性能，那么在这种场景下，有没有什么办法可以优化下锁的实现方式呢？

1. 读写锁 ReentrantReadWriteLock

针对这种读多写少的场景，Java 提供了另外一个实现 Lock 接口的读写锁 RRW。我们已知 ReentrantLock 是一个独占锁，同一时间只允许一个线程访问，而 RRW 允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同

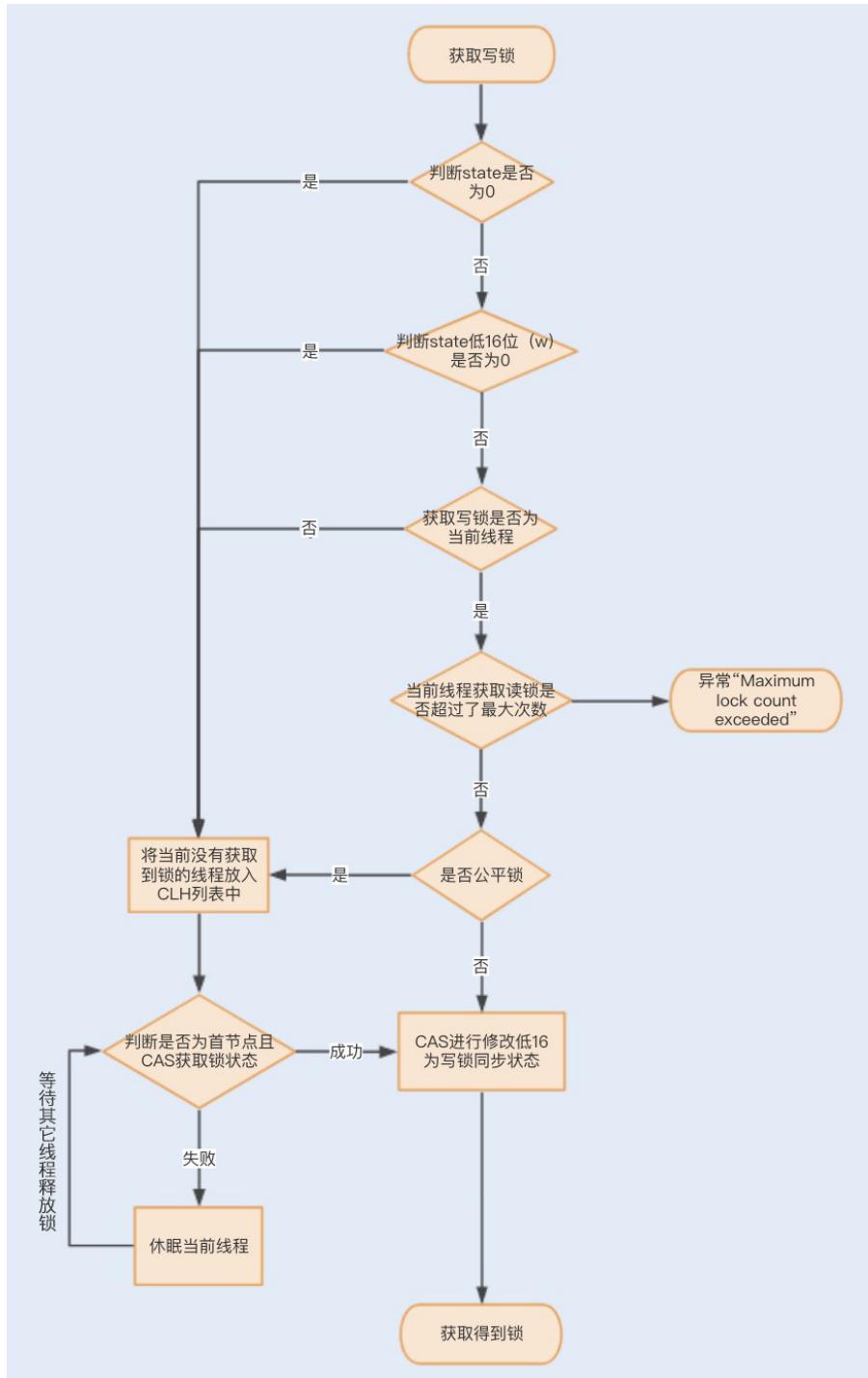
时访问。读写锁内部维护了两个锁，一个是为了读操作的 ReadLock，一个是用于写操作的 WriteLock。

那读写锁又是如何实现锁分离来保证共享资源的原子性呢？

RRW 也是基于 AQS 实现的，它的自定义同步器（继承 AQS）需要在同步状态 state 上维护多个读线程和一个写线程的状态，该状态的设计成为实现读写锁的关键。RRW 很好地使用了高低位，来实现一个整型控制两种状态的功能，读写锁将变量切分成了两个部分，高 16 位表示读，低 16 位表示写。

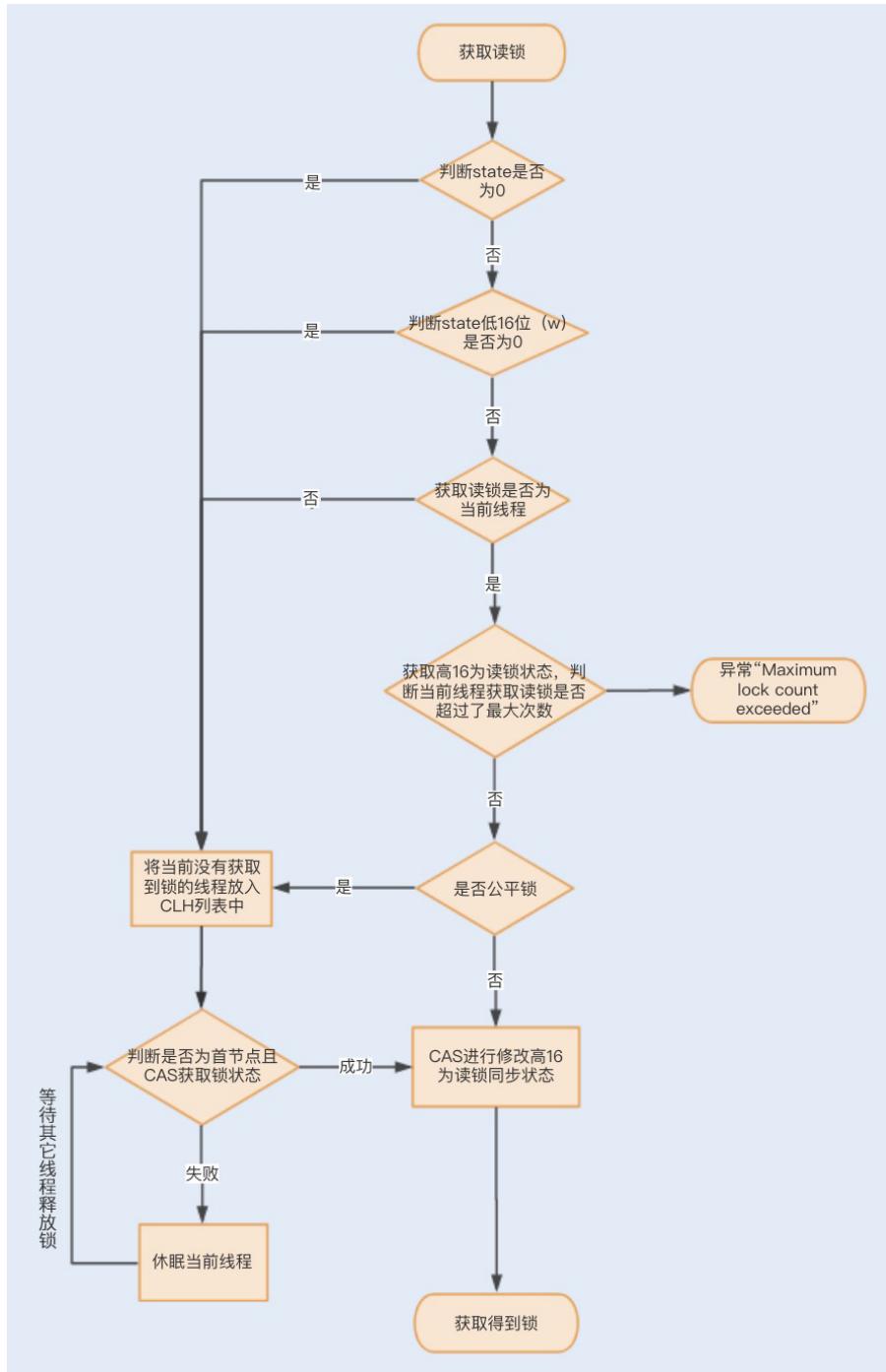
一个线程尝试获取写锁时，会先判断同步状态 state 是否为 0。如果 state 等于 0，说明暂时没有其它线程获取锁；如果 state 不等于 0，则说明有其它线程获取了锁。

此时再判断同步状态 state 的低 16 位 (w) 是否为 0，如果 w 为 0，则说明其它线程获取了读锁，此时进入 CLH 队列进行阻塞等待；如果 w 不为 0，则说明其它线程获取了写锁，此时要判断获取了写锁的是不是当前线程，若不是就进入 CLH 队列进行阻塞等待；若是，就应该判断当前线程获取写锁是否超过了最大次数，若超过，抛异常，反之更新同步状态。



一个线程尝试获取读锁时，同样会先判断同步状态 state 是否为 0。如果 state 等于 0，说明暂时没有其它线程获取锁，此时判断是否需要阻塞，如果需要阻塞，则进入 CLH 队列进行阻塞等待；如果不需要阻塞，则 CAS 更新同步状态为读状态。

如果 state 不等于 0，会判断同步状态低 16 位，如果存在写锁，则获取读锁失败，进入 CLH 阻塞队列；反之，判断当前线程是否应该被阻塞，如果不应该阻塞则尝试 CAS 同步状态，获取成功更新同步锁为读状态。

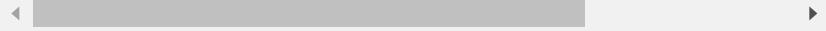


下面我们通过一个求平方的例子，来感受下 RRW 的实现，
代码如下：

 复制代码

```
1 public class TestRTTLock {  
2  
3     private double x, y;  
4  
5     private ReentrantReadWriteLock lock = new Reent  
6         // 读锁  
7     private Lock readLock = lock.readLock();  
8         // 写锁  
9     private Lock writeLock = lock.writeLock();  
10  
11    public double read() {  
12        // 获取读锁  
13        readLock.lock();  
14        try {  
15            return Math.sqrt(x * x + y * y)  
16        } finally {  
17            // 释放读锁  
18            readLock.unlock();  
19        }  
20    }  
21  
22    public void move(double deltaX, double deltaY)  
23        // 获取写锁  
24        writeLock.lock();  
25        try {  
26            x += deltaX;  
27            y += deltaY;  
28        } finally {  
29            // 释放写锁
```

```
30                     writeLock.unlock();
31                 }
32             }
33
34 }
```



2. 读写锁再优化之 StampedLock

RRW 被很好地应用在了读大于写的并发场景中，然而 RRW 在性能上还有可提升的空间。在读取很多、写入很少的情况下，RRW 会使写入线程遭遇饥饿（Starvation）问题，也就是说写入线程会因迟迟无法竞争到锁而一直处于等待状态。

在 JDK1.8 中，Java 提供了 StampedLock 类解决了这个问题。StampedLock 不是基于 AQS 实现的，但实现的原理和 AQS 是一样的，都是基于队列和锁状态实现的。与 RRW 不一样的是，**StampedLock 控制锁有三种模式：写、悲观读以及乐观读**，并且 StampedLock 在获取锁时会返回一个票据 stamp，获取的 stamp 除了在释放锁时需要校验，在乐观读模式下，stamp 还会作为读取共享资源后的二次校验，后面我会讲解 stamp 的工作原理。

我们先通过一个官方的例子来了解下 StampedLock 是如何使用的，代码如下：

 复制代码

```
1 public class Point {  
2     private double x, y;  
3     private final StampedLock s1 = new StampedLock();  
4  
5     void move(double deltaX, double deltaY) {  
6         // 获取写锁  
7         long stamp = s1.writeLock();  
8         try {  
9             x += deltaX;  
10            y += deltaY;  
11        } finally {  
12            // 释放写锁  
13            s1.unlockWrite(stamp);  
14        }  
15    }  
16  
17    double distanceFromOrigin() {  
18        // 乐观读操作  
19        long stamp = s1.tryOptimisticRead();  
20        // 拷贝变量  
21        double currentX = x, currentY = y;  
22        // 判断读期间是否有写操作  
23        if (!s1.validate(stamp)) {  
24            // 升级为悲观读  
25            stamp = s1.readLock();  
26            try {  
27                currentX = x;  
28                currentY = y;  
29            } finally {  
30                s1.unlockRead(stamp);  
31            }  
32        }  
33        return Math.sqrt(currentX * currentX + currentY  
34    }
```



我们可以发现：一个写线程获取写锁的过程中，首先是通过 WriteLock 获取一个票据 stamp，WriteLock 是一个独占锁，同时只有一个线程可以获取该锁，当一个线程获取该锁后，其它请求的线程必须等待，当没有线程持有读锁或者写锁的时候才可以获取到该锁。请求该锁成功后会返回一个 stamp 票据变量，用来表示该锁的版本，当释放该锁的时候，需要 unlockWrite 并传递参数 stamp。

接下来就是一个读线程获取锁的过程。首先线程会通过乐观锁 tryOptimisticRead 操作获取票据 stamp，如果当前没有线程持有写锁，则返回一个非 0 的 stamp 版本信息。线程获取该 stamp 后，将会拷贝一份共享资源到方法栈，在这之前具体的操作都是基于方法栈的拷贝数据。

之后方法还需要调用 validate，验证之前调用 tryOptimisticRead 返回的 stamp 在当前是否有其它线程持有了写锁，如果是，那么 validate 会返回 0，升级为悲观锁；否则就可以使用该 stamp 版本的锁对数据进行操作。

相比于 RRW，StampedLock 获取读锁只是使用与或操作进行检验，不涉及 CAS 操作，即使第一次乐观锁获取失

败，也会马上升级至悲观锁，这样就可以避免一直进行 CAS 操作带来的 CPU 占用性能的问题，因此 StampedLock 的效率更高。

总结

不管使用 Synchronized 同步锁还是 Lock 同步锁，只要存在锁竞争就会产生线程阻塞，从而导致线程之间的频繁切换，最终增加性能消耗。因此，**如何降低锁竞争，就成为了优化锁的关键。**

在 Synchronized 同步锁中，我们了解了可以通过减小锁粒度、减少锁占用时间来降低锁的竞争。在这一讲中，我们知道可以利用 Lock 锁的灵活性，通过锁分离的方式来降低锁竞争。

Lock 锁实现了读写锁分离来优化读大于写的场景，从普通的 RRW 实现到读锁和写锁，到 StampedLock 实现了乐观读锁、悲观读锁和写锁，都是为了降低锁的竞争，促使系统的并发性能达到最佳。

14 | 多线程之锁优化（下）：使用乐观锁优化并行操作



前两讲我们讨论了 `Synchronized` 和 `Lock` 实现的同步锁机制，这两种同步锁都属于悲观锁，是保护线程安全最直观的

方式。

我们知道悲观锁在高并发的场景下，激烈的锁竞争会造成线程阻塞，大量阻塞线程会导致系统的上下文切换，增加系统的性能开销。那有没有可能实现一种非阻塞型的锁机制来保证线程的安全呢？答案是肯定的。今天我就带你学习下乐观锁的优化方法，看看怎么使用才能发挥它最大的价值。

什么是乐观锁

开始优化前，我们先来简单回顾下乐观锁的定义。

乐观锁，顾名思义，就是说在操作共享资源时，它总是抱着乐观的态度进行，它认为自己可以成功地完成操作。但实际上，当多个线程同时操作一个共享资源时，只有一个线程会成功，那么失败的线程呢？它们不会像悲观锁一样在操作系统中挂起，而仅仅是返回，并且系统允许失败的线程重试，也允许自动放弃退出操作。

所以，乐观锁相比悲观锁来说，不会带来死锁、饥饿等活性故障问题，线程间的相互影响也远远比悲观锁要小。更为重要的是，**乐观锁没有因竞争造成的系统开销，所以在性能上也是更胜一筹。**

乐观锁的实现原理

相信你对上面的内容是有一定的了解的，下面我们来看看乐观锁的实现原理，有助于我们从根本上总结优化方法。

CAS 是实现乐观锁的核心算法，它包含了 3 个参数：V（需要更新的变量）、E（预期值）和 N（最新值）。

只有当需要更新的变量等于预期值时，需要更新的变量才会被设置为最新值，如果更新值和预期值不同，则说明已经有其它线程更新了需要更新的变量，此时当前线程不做操作，返回 V 的真实值。

1.CAS 如何实现原子操作

在 JDK 中的 concurrent 包中，atomic 路径下的类都是基于 CAS 实现的。AtomicInteger 就是基于 CAS 实现的一个线程安全的整型类。下面我们通过源码来了解下如何使用 CAS 实现原子操作。

我们可以看到 AtomicInteger 的自增方法 getAndIncrement 是用了 Unsafe 的 getAndAddInt 方法，显然 AtomicInteger 依赖于本地方法 Unsafe 类，Unsafe 类中的操作方法会调用 CPU 底层指令实现原子操作。

 复制代码

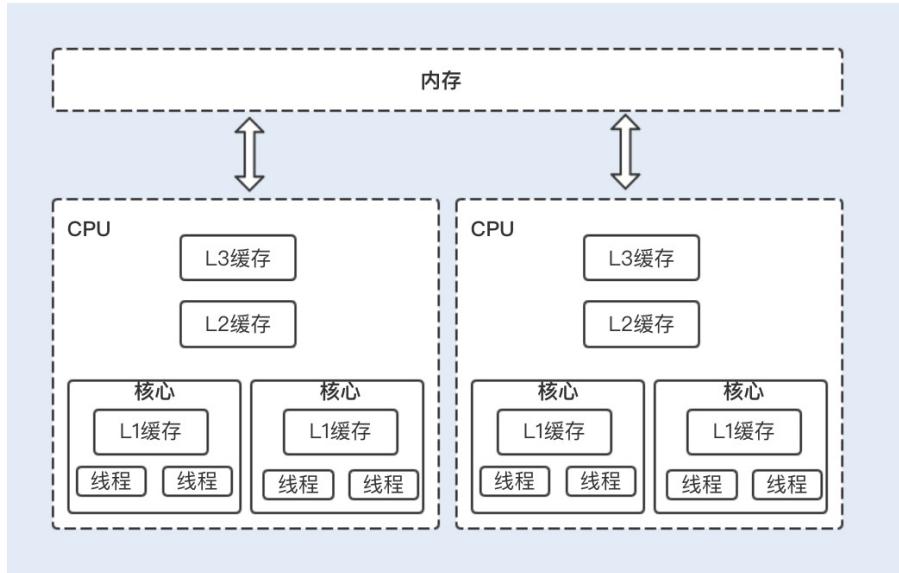
```
1 // 基于 CAS 操作更新值
2     public final boolean compareAndSet(int expect, int
3             return unsafe.compareAndSwapInt(this, valueOffset,
4         }
5 // 基于 CAS 操作增 1
6     public final int getAndIncrement() {
7         return unsafe.getAndAddInt(this, valueOffset, 1
8     }
9
10 // 基于 CAS 操作减 1
11    public final int getAndDecrement() {
12        return unsafe.getAndAddInt(this, valueOffset, -1
13    }
```



2. 处理器如何实现原子操作

CAS 是调用处理器底层指令来实现原子操作，那么处理器底层又是如何实现原子操作的呢？

处理器和物理内存之间的通信速度要远慢于处理器间的处理速度，所以处理器有自己的内部缓存。如下图所示，在执行操作时，频繁使用的内存数据会缓存在处理器的 L1、L2 和 L3 高速缓存中，以加快频繁读取的速度。



一般情况下，一个单核处理器能自我保证基本的内存操作是原子性的，当一个线程读取一个字节时，所有进程和线程看到的字节都是同一个缓存里的字节，其它线程不能访问这个字节的内存地址。

但现在的服务器通常是多处理器，并且每个处理器都是多核的。每个处理器维护了一块字节的内存，每个内核维护了一块字节的缓存，这时候多线程并发就会存在缓存不一致的问题，从而导致数据不一致。

这个时候，处理器提供了**总线锁定**和**缓存锁定**两个机制来保证复杂内存操作的原子性。

当处理器要操作一个共享变量的时候，其在总线上会发出一个 Lock 信号，这时其它处理器就不能操作共享变量了，该处理器会独享此共享内存中的变量。但总线锁定在阻塞其它处理器获取该共享变量的操作请求时，也可能会导致大量阻塞，从而增加系统的性能开销。

于是，后来的处理器都提供了缓存锁定机制，也就说当某个处理器对缓存中的共享变量进行了操作，就会通知其它处理器放弃存储该共享资源或者重新读取该共享资源。**目前最新的处理器都支持缓存锁定机制。**

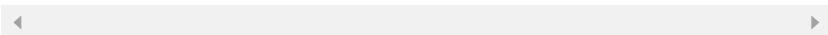
优化 CAS 乐观锁

虽然乐观锁在并发性能上要比悲观锁优越，但是在写大于读的操作场景下，CAS 失败的可能性会增大，如果不放弃此次 CAS 操作，就需要循环做 CAS 重试，这无疑会长时间地占用 CPU。

在 Java7 中，通过以下代码我们可以看到：AtomicInteger 的 getAndSet 方法中使用了 for 循环不断重试 CAS 操作，如果长时间不成功，就会给 CPU 带来非常大的执行开销。到了 Java8，for 循环虽然被去掉了，但我们反编译 Unsafe 类时就可以发现该循环其实是被封装在了 Unsafe 类中，CPU 的执行开销依然存在。

 复制代码

```
1  public final int getAndSet(int newValue) {  
2      for (;;) {  
3          int current = get();  
4          if (compareAndSet(current, newValue))  
5              return current;  
6      }  
7  }
```



在 JDK1.8 中，Java 提供了一个新的原子类 LongAdder。
**LongAdder 在高并发场景下会比 AtomicInteger 和
AtomicLong 的性能更好，代价就是会消耗更多的内存空
间。**

LongAdder 的原理就是降低操作共享变量的并发数，也就是将对单一共享变量的操作压力分散到多个变量值上，将竞争的每个写线程的 value 值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的 value 值进行 CAS 操作，最后在读取值的时候会将原子操作的共享变量与各个分散在数组的 value 值相加，返回一个近似准确的数值。

LongAdder 内部由一个 base 变量和一个 cell[] 数组组
成。当只有一个写线程，没有竞争的情况下，LongAdder
会直接使用 base 变量作为原子操作变量，通过 CAS 操作

修改变量；当有多个写线程竞争的情况下，除了占用 base 变量的一个写线程之外，其它各个线程会将修改的变量写入到自己的槽 cell[] 数组中，最终结果可通过以下公式计算得出：

$$value = base + \sum_{i=0}^n Cell[i]$$

我们可以发现，LongAdder 在操作后的返回值只是一个近似准确的数值，但是 LongAdder 最终返回的是一个准确的数值，所以**在一些对实时性要求比较高的场景下，LongAdder 并不能取代 AtomicInteger 或 AtomicLong。**

总结

在日常开发中，使用乐观锁最常见的场景就是数据库的更新操作了。为了保证操作数据库的原子性，我们常常会为每一条数据定义一个版本号，并在更新前获取到它，到了更新数据库的时候，还要判断下已经获取的版本号是否被更新过，如果没有，则执行该操作。

CAS 乐观锁在平常使用时比较受限，它只能保证单个变量操作的原子性，当涉及到多个变量时，CAS 就无能为力了，但前两讲讲到的悲观锁可以通过对整个代码块加锁来做到这点。

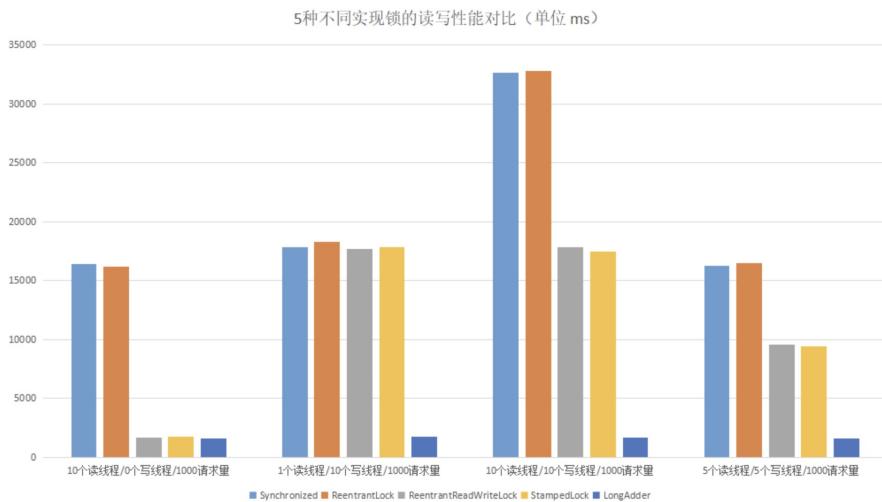
CAS 乐观锁在高并发写大于读的场景下，大部分线程的原子操作会失败，失败后的线程将会不断重试 CAS 原子操作，这样就会导致大量线程长时间地占用 CPU 资源，给系统带来很大的性能开销。在 JDK1.8 中，Java 新增了一个原子类 LongAdder，它使用了空间换时间的方法，解决了上述问题。

11~13 讲的内容，我详细地讲解了基于 JVM 实现的同步锁 Synchronized，AQS 实现的同步锁 Lock 以及 CAS 实现的乐观锁。相信你也很好奇，这三种锁，到底哪一种的性能最好，现在我们来对比下三种不同实现方式下的锁的性能。

鉴于脱离实际业务场景的性能对比测试没有意义，我们可以分别在“读多写少”“读少写多”“读写差不多”这三种场景下进行测试。又因为锁的性能还与竞争的激烈程度有关，所以除此之外，我们还将做三种锁在不同竞争级别下的性能测试。

综合上述条件，我将对四种模式下的五个锁 Synchronized、ReentrantLock、ReentrantReadWriteLock、StampedLock 以及乐观锁 LongAdder 进行压测。

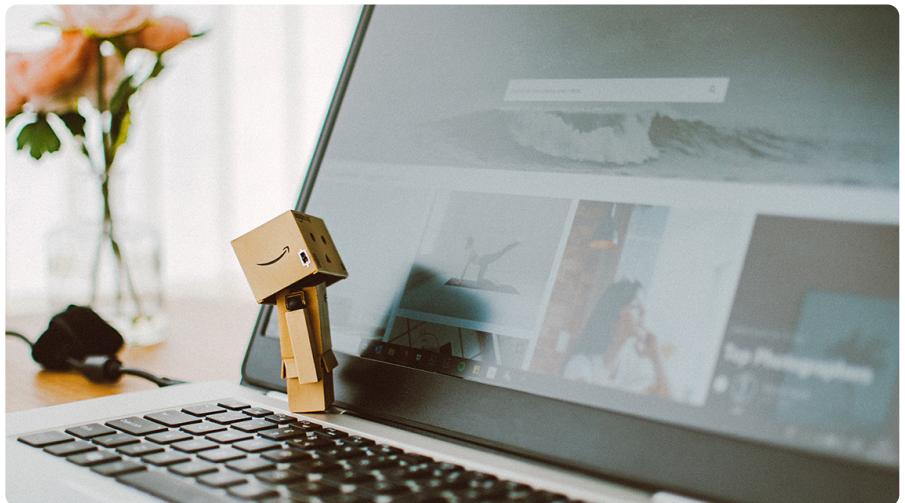
这里简要说明一下：我是在不同竞争级别的情况下，用不同的读写线程数组合出了四组测试，测试代码使用了计算并发计数器，读线程会去读取计数器的值，而写线程会操作变更计数器值，运行环境是 4 核的 i7 处理器。结果已给出，具体的测试代码可以点击[Github](#)查看下载。



通过以上结果，我们可以发现：在读大于写的场景下，读写锁 ReentrantReadWriteLock、StampedLock 以及乐观锁的读写性能是最好的；在写大于读的场景下，乐观锁的性能

是最好的，其它 4 种锁的性能则相差不多；在读和写差不多的场景下，两种读写锁以及乐观锁的性能要优于 Synchronized 和 ReentrantLock。

15 | 多线程调优（上）：哪些操作导致了上下文切换？



我们常说“实践是检验真理的唯一标准”，这句话不光在社会发展中可行，在技术学习中也同样适用。

记得我刚入职上家公司的时候，恰好赶上了一次抢购活动。这是系统重构上线后经历的第一次高并发考验，如期出现了大量超时报警，不过比我预料的要好一点，起码没有挂掉重启。

通过工具分析，我发现 cs（上下文切换每秒次数）指标已经接近了 60w，平时的话最高 5w。再通过日志分析，我发现了大量带有 wait() 的 Exception，由此初步怀疑是大量线程处理不及时导致的，进一步锁定问题是连接池大小设置不合理。后来我就模拟了生产环境配置，对连接数压测进行调节，降低最大线程数，最后系统的性能就上去了。

从实践中总结经验，我知道了在并发程序中，并不是启动更多的线程就能让程序最大限度地并发执行。线程数量设置太小，会导致程序不能充分地利用系统资源；线程数量设置太大，又可能带来资源的过度竞争，导致上下文切换带来额外的系统开销。

你看，其实很多经验就是这么一点点积累的。那么今天，我就想和你分享下“上下文切换”的相关内容，希望也能让你有所收获。

初识上下文切换

我们首先得明白，上下文切换到底是什么。

其实在单个处理器的时期，操作系统就能处理多线程并发任务。处理器给每个线程分配 CPU 时间片（Time Slice），线程在分配获得的时间片内执行任务。

CPU 时间片是 CPU 分配给每个线程执行的时间段，一般为几十毫秒。在这这么短的时间内线程互相切换，我们根本感觉不到，所以看上去就好像是同时进行的一样。

时间片决定了一个线程可以连续占用处理器运行的时长。当一个线程的时间片用完了，或者因自身原因被迫暂停运行了，这个时候，另外一个线程（可以是同一个线程或者其它进程的线程）就会被操作系统选中，来占用处理器。这种一个线程被暂停剥夺使用权，另外一个线程被选中开始或者继续运行的过程就叫做上下文切换（Context Switch）。

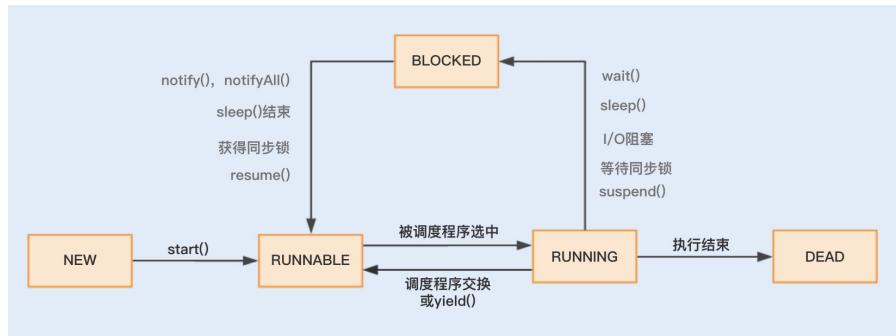
具体来说，一个线程被剥夺处理器的使用权而被暂停运行，就是“切出”；一个线程被选中占用处理器开始或者继续运行，就是“切入”。在这种切出切入的过程中，操作系统需要保存和恢复相应的进度信息，这个进度信息就是“上下文”了。

那上下文都包括哪些内容呢？具体来说，它包括了寄存器的存储内容以及程序计数器存储的指令内容。CPU 寄存器负责存储已经、正在和将要执行的任务，程序计数器负责存储 CPU 正在执行的指令位置以及即将执行的下一条指令的位置。

在当前 CPU 数量远远不止一个的情况下，操作系统将 CPU 轮流分配给线程任务，此时的上下文切换就变得更加频繁了，并且存在跨 CPU 上下文切换，比起单核上下文切换，跨核切换更加昂贵。

多线程上下文切换诱因

在操作系统中，上下文切换的类型还可以分为进程间的上下文切换和线程间的上下文切换。而在多线程编程中，我们主要面对的就是线程间的上下文切换导致的性能问题，下面我们就重点看看究竟是什么原因导致了多线程的上下文切换。开始之前，先看下 Java 线程的生命周期状态。



结合图示可知，线程主要有“新建”（NEW）、“就绪”（RUNNABLE）、“运行”（RUNNING）、“阻塞”（BLOCKED）、“死亡”（DEAD）五种状态。

在这个运行过程中，线程由 RUNNABLE 转为非 RUNNABLE 的过程就是线程上下文切换。

一个线程的状态由 RUNNING 转为 BLOCKED，再由 BLOCKED 转为 RUNNABLE，然后再被调度器选中执行，这就是一个上下文切换的过程。

当一个线程从 RUNNING 状态转为 BLOCKED 状态时，我们称为一个线程的暂停，线程暂停被切出之后，操作系统会保存相应的上下文，以便这个线程稍后再次进入 RUNNABLE 状态时能够在之前执行进度的基础上继续执行。

当一个线程从 BLOCKED 状态进入到 RUNNABLE 状态时，我们称为一个线程的唤醒，此时线程将获取上次保存的上下文继续完成执行。

通过线程的运行状态以及状态间的相互切换，我们可以了解到，多线程的上下文切换实际上就是由多线程两个运行状态的互相切换导致的。

那么在线程运行时，线程状态由 RUNNING 转为 BLOCKED 或者由 BLOCKED 转为 RUNNABLE，这又是什么诱发的呢？

我们可以分两种情况来分析，一种是程序本身触发的切换，这种我们称为自发性上下文切换，另一种是由系统或者虚拟机诱发的非自发性上下文切换。

自发性上下文切换指线程由 Java 程序调用导致切出，在多线程编程中，执行调用以下方法或关键字，常常就会引发自发性上下文切换。

sleep()

wait()

yield()

join()

park()

synchronized

lock

非自发性上下文切换指线程由于调度器的原因被迫切出。常见的有：线程被分配的时间片用完，虚拟机垃圾回收导致或

者执行优先级的问题导致。

这里重点说下“**虚拟机垃圾回收为什么会导致上下文切换**”。在 Java 虚拟机中，对象的内存都是由虚拟机中的堆分配的，在程序运行过程中，新的对象将不断被创建，如果旧的对象使用后不进行回收，堆内存将很快被耗尽。Java 虚拟机提供了一种回收机制，对创建后不再使用的对象进行回收，从而保证堆内存的可持续性分配。而这种垃圾回收机制的使用有可能会导致 stop-the-world 事件的发生，这其实就是一种线程暂停行为。

发现上下文切换

我们总说上下文切换会带来系统开销，那它带来的性能问题是不是真有这么糟糕呢？我们又该怎么去监测到上下文切换？上下文切换到底开销在哪些环节？接下来我将给出一段代码，来对比串联执行和并发执行的速度，然后一一解答这些问题。

 复制代码

```
1 public class DemoApplication {  
2     public static void main(String[] args) {  
3         // 运行多线程  
4         MultiThreadTester test1 = new MultiThreadTester();  
5         test1.Start();  
6         // 运行单线程  
7         SerialTester test2 = new SerialTester();  
8     }  
9 }
```

```
8             test2.Start();
9         }
10
11
12     static class MultiThreadTester extends ThreadCor
13         @Override
14         public void Start() {
15             long start = System.currentTimeMillis();
16             MyRunnable myRunnable1 = new MyRunn
17             Thread[] threads = new Thread[4];
18             // 创建多个线程
19             for (int i = 0; i < 4; i++) {
20                 threads[i] = new Thread(myRu
21                 threads[i].start();
22             }
23             for (int i = 0; i < 4; i++) {
24                 try {
25                     // 等待一起运行完
26                     threads[i].join();
27                 } catch (InterruptedException e) {
28                     // TODO Auto-generated e.printStackTrace();
29                 }
30             }
31             long end = System.currentTimeMillis();
32             System.out.println("multi thread e
33             System.out.println("counter: " + c
34         }
35         // 创建一个实现 Runnable 的类
36         class MyRunnable implements Runnable {
37             public void run() {
38                 while (counter < 1000000000)
39                     synchronized (this) {
40                         if(counter < 1
41                             increas
```

```
43 }  
44  
45 }  
46 }  
47 }  
48 }  
49 }  
50  
51 // 创建一个单线程  
52 static class SerialTester extends ThreadContextS  
53     @Override  
54     public void Start() {  
55         long start = System.currentTimeMillis();  
56         for (long i = 0; i < count; i++) {  
57             increaseCounter();  
58         }  
59         long end = System.currentTimeMillis();  
60         System.out.println("serial exec time: " + (end - start));  
61         System.out.println("counter: " + counter);  
62     }  
63 }  
64  
65 // 父类  
66 static abstract class ThreadContextSwitchTester  
67     public static final int count = 100000000;  
68     public volatile int counter = 0;  
69     public int getCount() {  
70         return this.counter;  
71     }  
72     public void increaseCounter() {  
73         this.counter += 1;  
74     }  
75     public abstract void Start();  
76 }  
77 }
```



执行之后，看一下两者的时间测试结果：

```
<terminated> DemoApplication [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (2019年3月2日 下午3:36:10)
multi thread exec time: 5234s
counter: 100000000
serial exec time: 858s
counter: 100000000
```

通过数据对比我们可以看到：串联的执行速度比并发的执行速度要快。这就是因为线程的上下文切换导致了额外的开销，使用 Synchronized 锁关键字，导致了资源竞争，从而引起了上下文切换，但即使不使用 Synchronized 锁关键字，并发的执行速度也无法超越串联的执行速度，这是因为多线程同样存在着上下文切换。**Redis、NodeJS 的设计就很好地体现了单线程串行的优势。**

在 Linux 系统下，可以使用 Linux 内核提供的 vmstat 命令，来监视 Java 程序运行过程中系统的上下文切换频率，cs 如下图所示：

```
[root@localhost ~]# vmstat 2
procs --memory-- --swap-- --io-- --system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
1 0 0 2130600 247876 5004968 0 0 0 1 0 1 0 0 100 0 0
0 0 0 2130352 247876 5004968 0 0 0 0 317 653 0 0 99 0 0
2 0 0 2106740 247876 5004968 0 0 0 2 1818 5202 9 1 90 0 0
2 0 0 2106848 247876 5005004 0 0 0 0 13664 42153 35 5 59 0 0
1 0 0 2106848 247876 5005004 0 0 0 0 16576 51599 33 6 61 0 0
0 0 0 2130536 247876 5004972 0 0 0 36 511 635 5 0 94 0 0
0 0 0 2130536 247876 5004972 0 0 0 0 254 579 0 0 100 0 0
0 0 0 2130536 247876 5004972 0 0 0 0 254 569 0 0 100 0 0
```

如果是监视某个应用的上下文切换，就可以使用 pidstat 命令监控指定进程的 Context Switch 上下文切换。

Time	UID	PID	cswch/s	nvcswch/s	Command
12:09:34 PM	0	998	0.00	0.00	-bash
12:09:35 PM	0	998	0.00	0.00	-bash
12:09:36 PM	0	998	0.00	0.00	-bash
12:09:37 PM	0	998	0.00	0.00	-bash
12:09:38 PM	0	998	0.00	0.00	-bash
12:09:39 PM	0	998	0.00	0.00	-bash
12:09:40 PM	0	998	0.00	0.00	-bash
12:09:41 PM	0	998	0.00	0.00	-bash
12:09:42 PM	0	998	0.00	0.00	-bash
12:09:43 PM	0	998	0.00	0.00	-bash
12:09:44 PM	0	998	0.00	0.00	-bash
12:09:45 PM	0	998	0.00	0.00	-bash
12:09:46 PM	0	998	0.00	0.00	-bash
12:09:47 PM	0	998	0.00	0.00	-bash
12:09:48 PM	0	998	0.00	0.00	-bash
12:09:49 PM	0	998	0.00	0.00	-bash
12:09:50 PM	0	998	2.00	0.00	-bash
12:09:51 PM	0	998	0.00	0.00	-bash
12:09:52 PM	0	998	0.00	0.00	-bash

由于 Windows 没有像 vmstat 这样的工具，在 Windows 下，我们可以使用 Process Explorer，来查看程序执行时，线程间上下文切换的次数。

至于系统开销具体发生在切换过程中的哪些具体环节，总结如下：

操作系统保存和恢复上下文；

调度器进行线程调度；

处理器高速缓存重新加载；

上下文切换也可能导致整个高速缓存区被冲刷，从而带来时间开销。

总结

上下文切换就是一个工作的线程被另外一个线程暂停，另外一个线程占用了处理器开始执行任务的过程。系统和 Java 程序自发性以及非自发性的调用操作，就会导致上下文切换，从而带来系统开销。

线程越多，系统的运行速度不一定越快。那么我们平时在并发量比较大的情况下，**什么时候用单线程，什么时候用多线程呢？**

一般在单个逻辑比较简单，而且速度相对来非常快的情况下，我们可以使用单线程。例如，我们前面讲到的 Redis，从内存中快速读取值，不用考虑 I/O 瓶颈带来的阻塞问题。而在逻辑相对来说很复杂的场景，等待时间相对较长又或者是需要大量计算的场景，我建议使用多线程来提高系统的整体性能。例如，NIO 时期的文件读写操作、图像处理以及大数据分析等。

16 | 多线程调优（下）：如何优化多线程上下文切换？



通过上一讲的讲解，相信你对上下文切换已经有了一定的了解了。如果是单个线程，在 CPU 调用之后，那么它基本上是不会被调度出去的。如果可运行的线程数远大于 CPU 数量，那么操作系统最终会将某个正在运行的线程调度出来，从而使其它线程能够使用 CPU，这就会导致上下文切换。

还有，在多线程中如果使用了竞争锁，当线程由于等待竞争锁而被阻塞时，JVM 通常会将这个锁挂起，并允许它被交换出去。如果频繁地发生阻塞，CPU 密集型的程序就会发生更多的上下文切换。

那么问题来了，我们知道在某些场景下使用多线程是非常必要的，但多线程编程给系统带来了上下文切换，从而增加的性能开销也是实打实存在的。那么我们该如何优化多线程上下文切换呢？这就是我今天要和你分享的话题，我将重点介绍几种常见的优化方法。

竞争锁优化

大多数人在多线程编程中碰到性能问题，第一反应多是想到了锁。

多线程对锁资源的竞争会引起上下文切换，还有锁竞争导致的线程阻塞越多，上下文切换就越频繁，系统的性能开销也就越大。由此可见，在多线程编程中，锁其实不是性能开销的根源，竞争锁才是。

第 11~13 讲中我曾集中讲过锁优化，我们知道锁的优化归根结底就是减少竞争。这讲中我们就再来总结下锁优化的一些方式。

1. 减少锁的持有时间

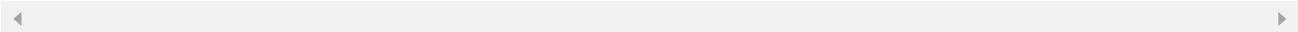
我们知道，锁的持有时间越长，就意味着有越多的线程在等待该竞争资源释放。如果是 Synchronized 同步锁资源，就不仅是带来线程间的上下文切换，还有可能会增加进程间的上下文切换。

在第 12 讲中，我曾分享过一些更具体的方法，例如，可以将一些与锁无关的代码移出同步代码块，尤其是那些开销较大的操作以及可能被阻塞的操作。

优化前

 复制代码

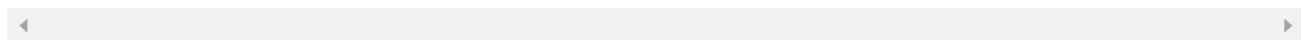
```
1 public synchronized void mySyncMethod(){  
2     businesscode1();  
3     mutexMethod();  
4     businesscode2();  
5 }
```



优化后

 复制代码

```
1 public void mySyncMethod(){
2     businesscode1();
3     synchronized(this)
4     {
5         mutexMethod();
6     }
7     businesscode2();
8 }
```



2. 降低锁的粒度

同步锁可以保证对象的原子性，我们可以考虑将锁粒度拆分得更小一些，以此避免所有线程对一个锁资源的竞争过于激烈。具体方式有以下两种：

锁分离

与传统锁不同的是，读写锁实现了锁分离，也就是说读写锁是由“读锁”和“写锁”两个锁实现的，其规则是可以共享读，但只有一个写。

这样做好处是，在多线程读的时候，读读是不互斥的，读写是互斥的，写写是互斥的。而传统的独占锁在没有区分读写锁的时候，读写操作一般是：读读互斥、读写互斥、写写互斥。所以在读远大于写的多线程场景中，锁分离避免了在高并发读情况下的资源竞争，从而避免了上下文切换。

锁分段

我们在使用锁来保证集合或者大对象原子性时，可以考虑将锁对象进一步分解。例如，我之前讲过的 Java1.8 之前版本的 ConcurrentHashMap 就使用了锁分段。

3. 非阻塞乐观锁替代竞争锁

volatile 关键字的作用是保障可见性及有序性，volatile 的读写操作不会导致上下文切换，因此开销比较小。但是，volatile 不能保证操作变量的原子性，因为没有锁的排他性。

而 CAS 是一个原子的 if-then-act 操作，CAS 是一个无锁算法实现，保障了对一个共享变量读写操作的一致性。CAS 操作中有 3 个操作数，内存值 V、旧的预期值 A 和要修改的新

值 B，当且仅当 A 和 V 相同时，将 V 修改为 B，否则什么都不做，CAS 算法将不会导致上下文切换。Java 的 Atomic 包就使用了 CAS 算法来更新数据，就不需要额外加锁。

上面我们了解了如何从编码层面去优化竞争锁，那么除此之外，JVM 内部其实也对 Synchronized 同步锁做了优化，我在 12 讲中有详细地讲解过，这里简单回顾一下。

在 JDK1.6 中，JVM 将 Synchronized 同步锁分为了偏向锁、轻量级锁、偏向锁以及重量级锁，优化路径也是按照以上顺序进行。JIT 编译器在动态编译同步块的时候，也会通过锁消除、锁粗化的方式来优化该同步锁。

wait/notify 优化

在 Java 中，我们可以通过配合调用 Object 对象的 wait() 方法和 notify() 方法或 notifyAll() 方法来实现线程间的通信。

在线程中调用 wait() 方法，将阻塞等待其它线程的通知（其它线程调用 notify() 方法或 notifyAll() 方法），在线程中调用 notify() 方法或 notifyAll() 方法，将通知其它线程从 wait() 方法处返回。

下面我们通过 wait() / notify() 来实现一个简单的生产者和消费者的案例，代码如下：

 复制代码

```
1 public class WaitNotifyTest {
2     public static void main(String[] args) {
3         Vector<Integer> pool=new Vector<Integer>();
4         Producer producer=new Producer(pool, 10);
5         Consumer consumer=new Consumer(pool);
6         new Thread(producer).start();
7         new Thread(consumer).start();
8     }
9 }
10 /**
11 * 生产者
12 * @author admin
13 *
14 */
15 class Producer implements Runnable{
16     private Vector<Integer> pool;
17     private Integer size;
18
19     public Producer(Vector<Integer> pool, Integer size) {
20         this.pool = pool;
```

```
21         this.size = size;
22     }
23
24     public void run() {
25         for(;;){
26             try {
27                 System.out.println(" 生产一个商品 ");
28                 produce(1);
29             } catch (InterruptedException e) {
30                 // TODO Auto-generated catch block
31                 e.printStackTrace();
32             }
33         }
34     }
35     private void produce(int i) throws InterruptedException{
36         while(pool.size()==size){
37             synchronized (pool) {
38                 System.out.println(" 生产者等待消费者消费商品，当前商品数量为 "+poc
39                 pool.wait();// 等待消费者消费
40             }
41         }
42         synchronized (pool) {
43             pool.add(i);
44             pool.notifyAll();// 生产成功，通知消费者消费
45         }
46     }
47 }
48
49
50 /**
51 * 消费者
52 * @author admin
53 *
54 */
55 class Consumer implements Runnable{
56     private Vector<Integer> pool;
57     public Consumer(Vector<Integer> pool) {
58         this.pool = pool;
59     }
60
61     public void run() {
62         for(;;){
63             try {
64                 System.out.println(" 消费一个商品 ");
65                 consume();
66             } catch (InterruptedException e) {
67                 // TODO Auto-generated catch block
68                 e.printStackTrace();
69             }
70         }
71     }
72 }
```

```
73     private void consume() throws InterruptedException{
74         while(pool.isEmpty()){
75             synchronized (pool) {
76                 System.out.println(" 消费者等待生产者生产商品，当前商品数量为 "+poc
77                 pool.wait();// 等待生产者生产商品
78             }
79         }
80         synchronized (pool) {
81             pool.remove(0);
82             pool.notifyAll();// 通知生产者生产商品
83
84         }
85     }
86
87 }
```



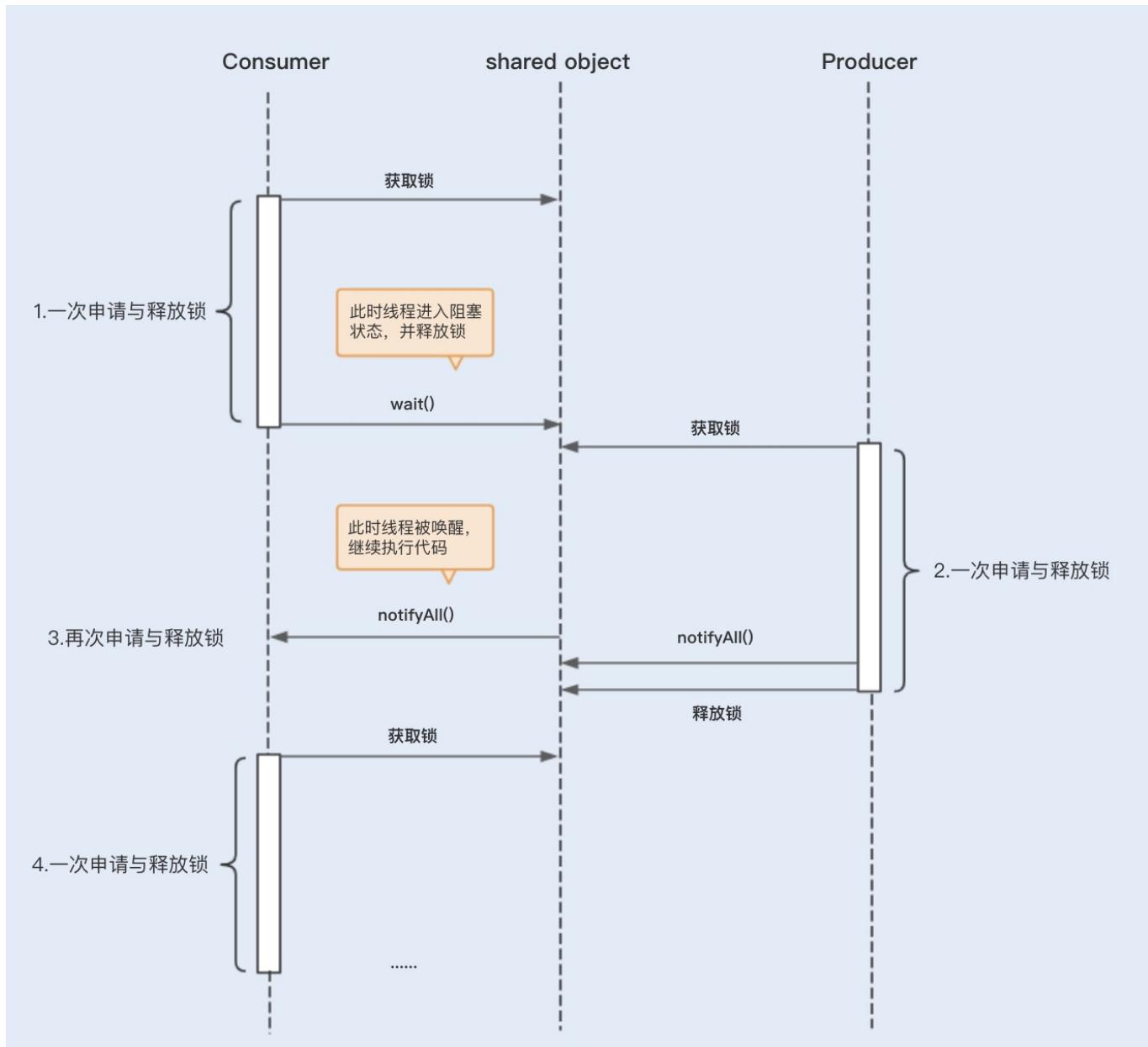
wait/notify 的使用导致了较多的上下文切换

结合以下图片，我们可以看到，在消费者第一次申请到锁之前，发现没有商品消费，此时会执行 Object.wait() 方法，这里会导致线程挂起，进入阻塞状态，这里为一次上下文切换。

当生产者获取到锁并执行 notifyAll() 之后，会唤醒处于阻塞状态的消费者线程，此时这里又发生了一次上下文切换。

被唤醒的等待线程在继续运行时，需要再次申请相对对象的内部锁，此时等待线程可能需要和其它新来的活跃线程争用内部锁，这也可能会导致上下文切换。

如果有多个消费者线程同时被阻塞，用 notifyAll() 方法，将会唤醒所有阻塞的线程。而某些商品依然没有库存，过早地唤醒这些没有库存的商品的消费线程，可能会导致线程再次进入阻塞状态，从而引起不必要的上下文切换。



优化 `wait/notify` 的使用，减少上下文切换

首先，我们在多个不同消费场景中，可以使用 `Object.notify()` 替代 `Object.notifyAll()`。因为 `Object.notify()` 只会唤醒指定线程，不会过早地唤醒其它未满足需求的阻塞线程，所以可以减少相应的上下文切换。

其次，在生产者执行完 `Object.notify()` / `notifyAll()` 唤醒其它线程之后，应该尽快地释放内部锁，以避免其它线程在唤醒之后长时间地持有锁处理业务操作，这样可以避免被唤醒的线程再次申请相应内部锁的时候等待锁的释放。

最后，为了避免长时间等待，我们常会使用 `Object.wait (long)` 设置等待超时时间，但线程无法区分其返回是由于等待超时还是被通知线程唤醒，从而导致线程再次尝试获取锁操作，增加了上下文切换。

这里我建议使用 Lock 锁结合 Condition 接口替代 Synchronized 内部锁中的 wait / notify，实现等待 / 通知。这样做不仅可以解决上述的 Object.wait(long) 无法区分的问题，还可以解决线程被过早唤醒的问题。

Condition 接口定义的 await 方法、signal 方法和 signalAll 方法分别相当于 Object.wait()、Object.notify() 和 Object.notifyAll()。

合理地设置线程池大小，避免创建过多线程

线程池的线程数量设置不宜过大，因为一旦线程池的工作线程总数超过系统所拥有的处理器数量，就会导致过多的上下文切换。更多关于如何合理设置线程池数量的内容，我将在第 18 讲中详解。

还有一种情况就是，在有些创建线程池的方法里，线程数量设置不会直接暴露给我们。比如，用 Executors.newCachedThreadPool() 创建的线程池，该线程池会复用其内部空闲的线程来处理新提交的任务，如果没有，再创建新的线程（不受 MAX_VALUE 限制），这样的线程池如果碰到大量且耗时长的任务场景，就会创建非常多的工作线程，从而导致频繁的上下文切换。因此，这类线程池就只适合处理大量且耗时短的非阻塞任务。

使用协程实现非阻塞等待

相信很多人一听到协程（Coroutines），马上想到的就是 Go 语言。协程对于大部分 Java 程序员来说可能还有点陌生，但其在 Go 中的使用相对来说已经很成熟了。

协程是一种比线程更加轻量级的东西，相比于由操作系统内核来管理的进程和线程，**协程则完全由程序本身所控制，也就是在用户态执行**。协程避免了像线程切换那样产生的上下文切换，在性能方面得到了很大的提升。协程在多线程业务上的运用，我会在第 18 讲中详述。

减少 Java 虚拟机的垃圾回收

我们在上一讲讲上下文切换的诱因时，曾提到过“垃圾回收会导致上下文切换”。

很多 JVM 垃圾回收器（serial 收集器、ParNew 收集器）在回收旧对象时，会产生内存碎片，从而需要进行内存整理，在这个过程中就需要移动存活的对象。而移动内存对象就意味着这些对象所在的内存地址会发生变化，因此在移动对象前需要暂停线程，在移动完成后需要再次唤醒该线程。因此减少 JVM 垃圾回收的频率可以有效地减少上下文切换。

总结

上下文切换是多线程编程性能消耗的原因之一，而竞争锁、线程间的通信以及过多地创建线程等多线程编程操作，都会给系统带来上下文切换。除此之外，I/O 阻塞以及 JVM 的垃圾回收也会增加上下文切换。

总的来说，过于频繁的上下文切换会影响系统的性能，所以我们应该避免它。另外，**我们还可以将上下文切换也作为系统的性能参考指标，并将该指标纳入到服务性能监控，防患于未然。**

17 | 并发容器的使用：识别不同场景下最优容器



在并发编程中，我们经常会用到容器。今天我要和你分享的话题就是：在不同场景下我们该如何选择最优容器。

并发场景下的 Map 容器

假设我们现在要给一个电商系统设计一个简单的统计商品销量 TOP 10 的功能。常规情况下，我们是用一个哈希表来存储商品和销量键值对，然后使用排序获得销量前十的商品。在这里，哈希表是实现该功能的关键。那么请思考一下，如果要你设计这个功能，你会使用哪个容器呢？

在 07 讲中，我曾详细讲过 HashMap 的实现原理，以及 HashMap 结构的各个优化细节。我说过 HashMap 的性能优越，经常被用来存储键值对。那么这里我们可以使用 HashMap 吗？

答案是不可以，我们切忌在并发场景下使用 HashMap。因为在 JDK1.7 之前，在并发场景下使用 HashMap 会出现死循环，从而导致 CPU 使用率居高不下，而扩容是导致死循环的主要原因。虽然 Java 在 JDK1.8 中修复了 HashMap 扩容导致的死循环问题，但在高并发场景下，依然会有数据丢失以及不准确的情况出现。

这时为了保证容器的线程安全，Java 实现了 Hashtable、ConcurrentHashMap 以及 ConcurrentSkipListMap 等 Map 容器。

Hashtable、ConcurrentHashMap 是基于 HashMap 实现的，对于小数据量的存取比较有优势。

ConcurrentSkipListMap 是基于 TreeMap 的设计原理实现的，略有不同的是前者基于跳表实现，后者基于红黑树实现，ConcurrentSkipListMap 的特点是存取平均时间复杂度是 $O(\log(n))$ ，适用于大数据量存取的场景，最常见的是基于跳跃表实现的数据量比较大的缓存。

回归到开始的案例再看一下，如果这个电商系统的商品总量不是特别大的话，我们可以用 Hashtable 或 ConcurrentHashMap 来实现哈希表的功能。

Hashtable ConcurrentHashMap

更精准的话，我们可以进一步对比看看以上两种容器。

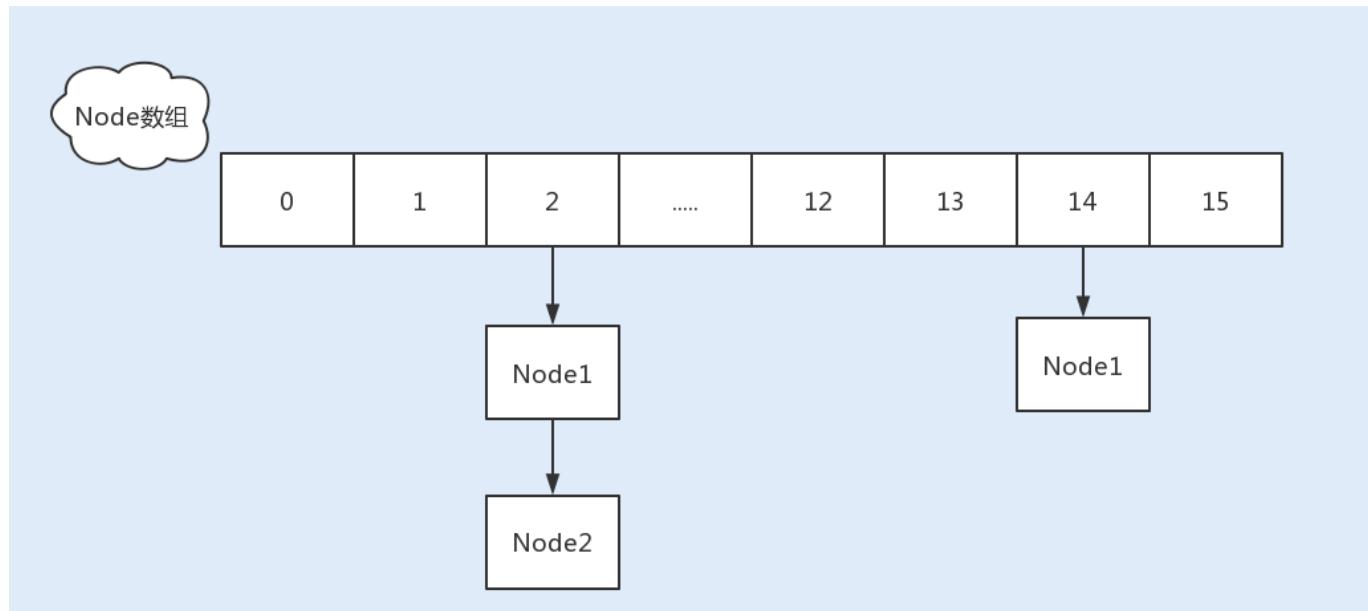
在数据不断地写入和删除，且不存在数据量累积以及数据排序的场景下，我们可以选用 Hashtable 或 ConcurrentHashMap。

Hashtable 使用 Synchronized 同步锁修饰了 put、get、remove 等方法，因此在高并发场景下，读写操作都会存在大量锁竞争，给系统带来性能开销。

相比 Hashtable，ConcurrentHashMap 在保证线程安全的基础上兼具了更好的并发性能。在 JDK1.7 中，ConcurrentHashMap 就使用了分段锁 Segment 减小了锁粒度，最终优化了锁的并发操作。

到了 JDK1.8，ConcurrentHashMap 做了大量的改动，摒弃了 Segment 的概念。由于 Synchronized 锁在 Java6 之后的性能已经得到了很大的提升，所以在 JDK1.8 中，Java 重新启用了 Synchronized 同步锁，通过 Synchronized 实现 HashEntry 作为锁粒度。这种改动将数据结构变得更加简单了，操作也更加清晰流畅。

与 JDK1.7 的 put 方法一样，JDK1.8 在添加元素时，在没有哈希冲突的情况下，会使用 CAS 进行添加元素操作；如果有冲突，则通过 Synchronized 将链表锁定，再执行接下来的操作。



综上所述，我们在设计销量 TOP10 功能时，首选 ConcurrentHashMap。

但要注意一点，虽然 ConcurrentHashMap 的整体性能要优于 Hashtable，但在某些场景中，ConcurrentHashMap 依然不能代替 Hashtable。例如，在强一致的场景中 ConcurrentHashMap 就不适用，原因是 ConcurrentHashMap 中的 get、size 等方法没有用到锁，ConcurrentHashMap 是弱一致性的，因此有可能会导致某次读无法马上获取到写入的数据。

ConcurrentHashMap ConcurrentHashMap

我们再看一个案例，我上家公司的操作系统中有这样一个功能，提醒用户手机卡实时流量不足。主要的流程是服务端先通过虚拟运营商同步用户实时流量，再通过手机端定时触发查询功能，如果流量不足，就弹出系统通知。

该功能的特点是用户量大，并发量高，写入多于查询操作。这时我们就需要设计一个缓存，用来存放这些用户以及对应的流量键值对信息。那么假设让你来实现一个简单的缓存，你会怎么设计呢？

你可能会考虑使用 ConcurrentHashMap 容器，但我在 07 讲中说过，该容器在数据量比较大的时候，链表会转换为红黑树。红黑树在并发情况下，删除和插入过程中有个平衡的过程，会牵涉到大量节点，因此竞争锁资源的代价相对比较高。

而跳跃表的操作针对局部，需要锁住的节点少，因此在并发场景下的性能会更好一些。你可能会问了，在非线程安全的 Map 容器中，我并没有看到基于跳跃表实现的 SkipListMap 呀？这是因为在非线程安全的 Map 容器中，基于红黑树实现的 TreeMap 在单线程中的性能表现得并不比跳跃表差。

因此就实现了在非线程安全的 Map 容器中，用 TreeMap 容器来存取大数据；在线程安全的 Map 容器中，用 SkipListMap 容器来存取大数据。

那么 ConcurrentSkipListMap 是如何使用跳跃表来提升容器存取大数据的性能呢？我们先来了解下跳跃表的实现原理。

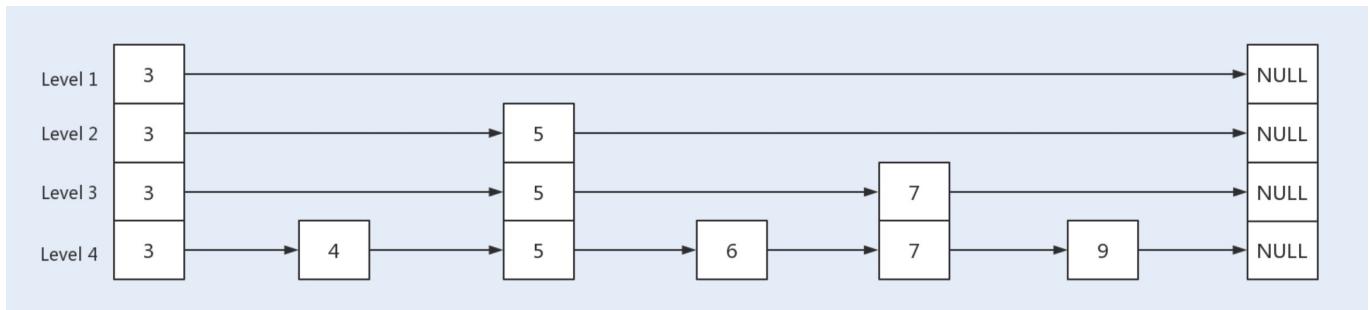
什么是跳跃表

跳跃表是基于链表扩展实现的一种特殊链表，类似于树的实现，跳跃表不仅实现了横向链表，还实现了垂直方向的分层索引。

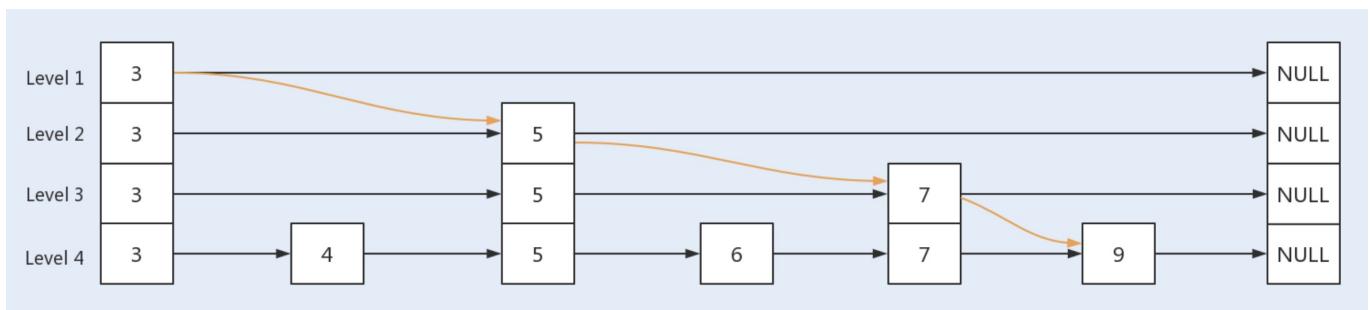
一个跳跃表由若干层链表组成，每一层都实现了一个有序链表索引，只有最底层包含了所有数据，每一层由下往上依次通过一个指针指向层相同值的元素，每层数据依次减少，等到了最顶层就只会保留部分数据了。

跳跃表的这种结构，是利用了空间换时间的方法来提高了查询效率。程序总是从最顶层开始查询访问，通过判断元素值来缩小查询范围。我们可以通过以下几张图来了解下跳跃表的具体实现原理。

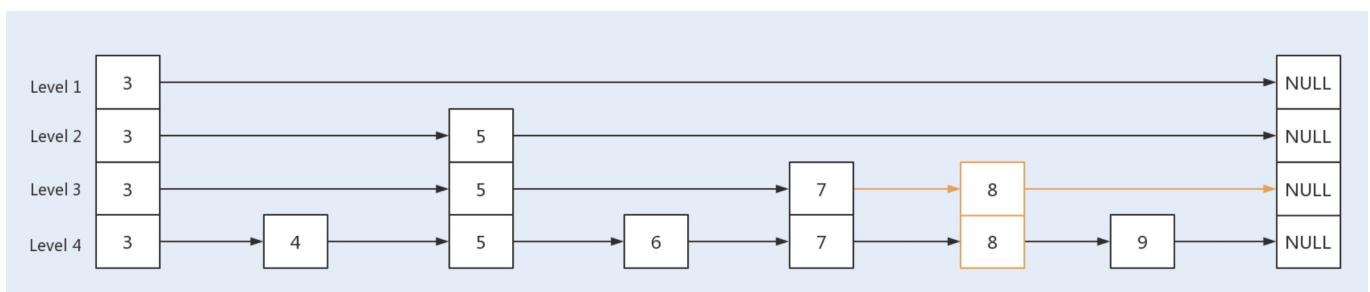
首先是一个初始化的跳跃表：



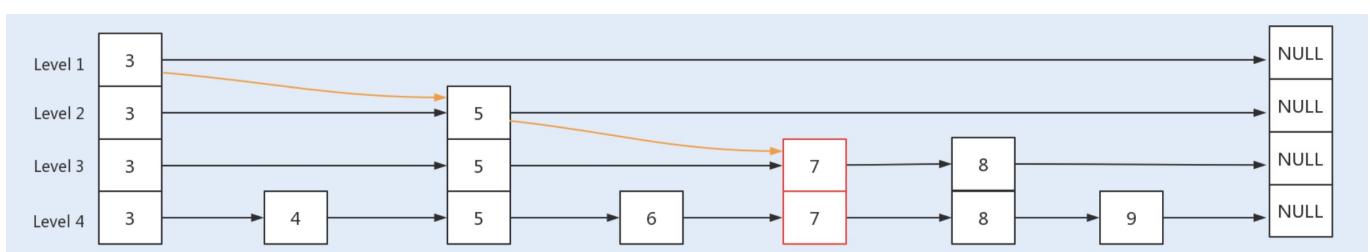
当查询 key 值为 9 的节点时，此时查询路径为：



当新增一个 key 值为 8 的节点时，首先新增一个节点到最底层的链表中，根据概率算出 level 值，再根据 level 值新建索引层，最后链接索引层的新节点。新增节点和链接索引都是基于 CAS 操作实现。



当删除一个 key 值为 7 的结点时，首先找到待删除结点，将其 value 值设置为 null；之后再向待删除结点的 next 位置新增一个标记结点，以便减少并发冲突；然后让待删结点的前驱节点直接越过本身指向的待删结点，直接指向后继结点，中间要被删除的结点最终将会被 JVM 垃圾回收处理掉；最后判断此次删除后是否导致某一索引层没有其它节点了，并视情况删除该层索引。



通过以上两个案例，我想你应该清楚了 Hashtable、ConcurrentHashMap 以及 ConcurrentSkipListMap 这三种容器的适用场景了。

如果对数据有强一致要求，则需使用 Hashtable；在大部分场景通常都是弱一致性的情况下，使用 ConcurrentHashMap 即可；如果数据量在千万级别，且存在大量增删改操作，则可以考虑使用 ConcurrentSkipListMap。

并发场景下的 List 容器

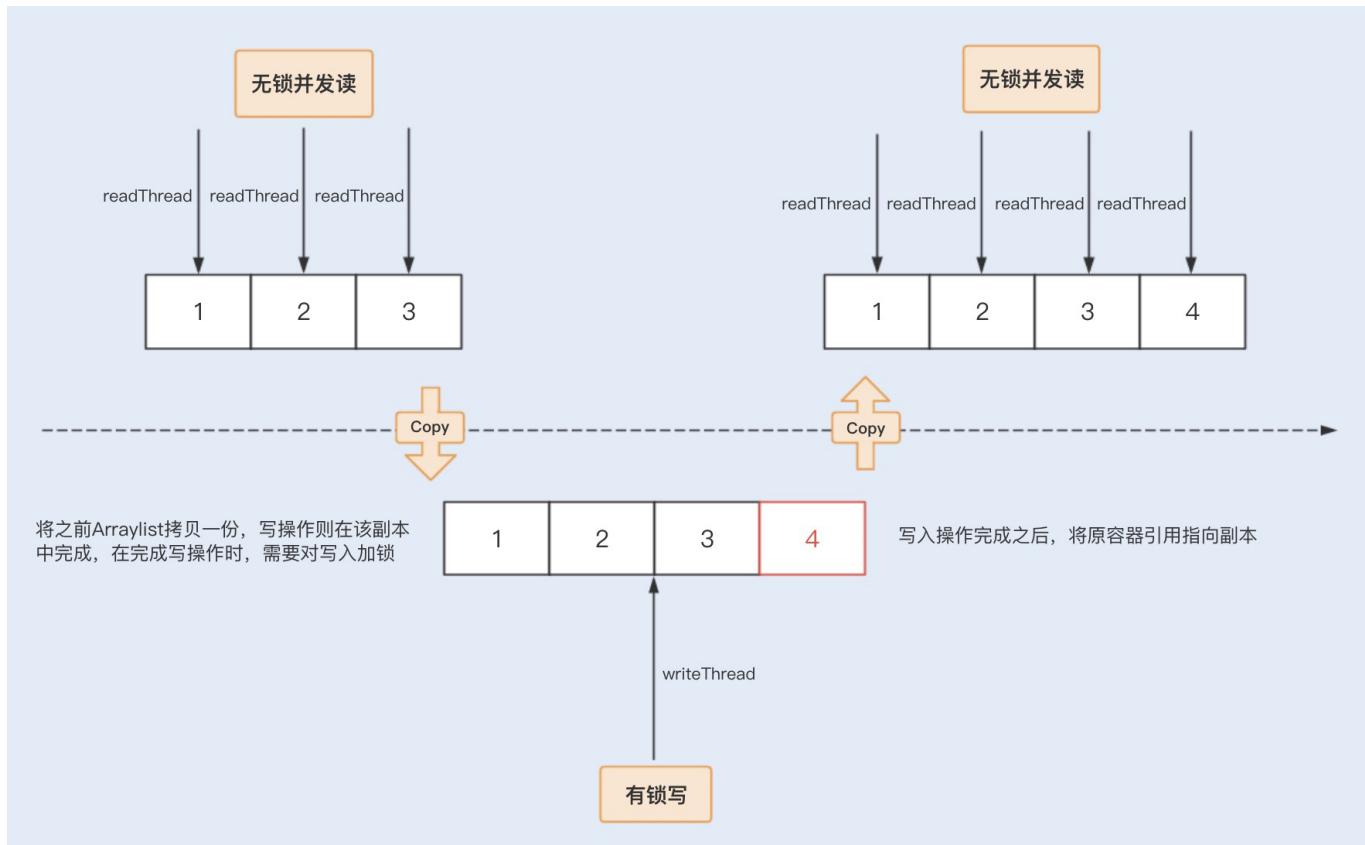
下面我们再来看一个实际生产环境中的案例。在大部分互联网产品中，都会设置一份黑名单。例如，在电商系统中，系统可能会将一些频繁参与抢购却放弃付款的用户放入到黑名单列表。想想这个时候你又会使用哪个容器呢？

首先用户黑名单的数据量并不会很大，但在抢购中需要查询该容器，快速获取到该用户是否存在于黑名单中。其次用户 ID 是整数类型，因此我们可以考虑使用数组来存储。那么 ArrayList 是否是你第一时间想到的呢？

我讲过 ArrayList 是非线程安全容器，在并发场景下使用很可能会导致线程安全问题。这时，我们就可以考虑使用 Java 在并发编程中提供的线程安全数组，包括 Vector 和 CopyOnWriteArrayList。

Vector 也是基于 Synchronized 同步锁实现的线程安全，Synchronized 关键字几乎修饰了所有对外暴露的方法，所以在读远大于写的操作场景中，Vector 将会发生大量锁竞争，从而给系统带来性能开销。

相比之下，CopyOnWriteArrayList 是 java.util.concurrent 包提供的方法，它实现了读操作无锁，写操作则通过操作底层数组的新副本实现，是一种读写分离的并发策略。我们可以通过以下图示来了解下 CopyOnWriteArrayList 的具体实现原理。



回到案例中，我们知道黑名单是一个读远大于写的操作业务，我们可以固定在某一个业务比较空闲的时间点来更新名单。

这种场景对写入数据的实时获取并没有要求，因此我们只需要保证最终能获取到写入数组中的用户 ID 就可以了，而 `CopyOnWriteArrayList` 这种并发数组容器无疑是最适合这类场景的了。

总结

在并发编程中，我们经常会使用容器来存储数据或对象。Java 在 JDK1.1 到 JDK1.8 这个漫长的发展过程中，依据场景的变化实现了同类型的多种容器。我将今天的主要内容为你总结了一张表格，希望能对你有所帮助，也欢迎留言补充。

分类	名称	特性	适用场景
Map并发容器	Hashtable	强一致性	对数据强一致性有要求的场景
	ConcurrentHashMap	基于数据+链表+红黑树实现， CAS+Synchronized实现原子性， 部分操作属于无锁操作，具有弱一致性	存取数据量小，查询操作频繁， 且对数据没有强一致要求的高并发场景
	ConcurrentSkipListMap	基于跳跃表实现，具有弱一致性	存取数据量大，增删改查操作频繁， 且对数据没有强一致要求的高并发场景
List并发容器	Vector	具有强一致性	对数据强一致性有要求的场景
	CopyOnWriteArrayList	基于复制副本用于有锁写操作，操作完成之后，Array容器重新指向新的副本容器，具有弱一致性	读远大于写操作的场景

18 | 如何设置线程池大小？



还记得我在 16 讲中说过“线程池的线程数量设置过多会导致线程竞争激烈”吗？今天再补一句，如果线程数量设置过少的话，还会导致系统无法充分利用计算机资源。那么如何设置才不会影响系统性能呢？

其实线程池的设置是有方法的，不是凭借简单的估算来决定的。[今天我们就来看看究竟有哪些计算方法可以复用](#)，线程池中各个参数之间又存在怎样的关系。

线程池原理

开始优化之前，我们先来看看线程池的实现原理，有助于你更好地理解后面的内容。

在 HotSpot VM 的线程模型中，Java 线程被一对一映射为内核线程。Java 在使用线程执行程序时，需要创建一个内核线程；当该 Java 线程被终止时，这个内核线程也会被回收。因此 Java 线程的创建与销毁将会消耗一定的计算机资源，从而增加系统的性能开销。

除此之外，大量创建线程同样会给系统带来性能问题，因为内存和 CPU 资源都将被线程抢占，如果处理不当，就会发生内存溢出、CPU 使用率超负荷等问题。

为了解决上述两类问题，Java 提供了线程池概念，对于频繁创建线程的业务场景，线程池可以创建固定的线程数量，并且在操作系统底层，轻量级进程将会把这些线程映射到内核。

线程池可以提高线程复用，又可以固定最大线程使用量，防止无限制地创建线程。当程序提交一个任务需要一个线程时，会去线程池中查找是否有空闲的线程，若有，则直接使用线程池中的线程工作，若没有，会去判断当前已创建的线程数量是否超过最大线程数量，如未超过，则创建新线程，如已超过，则进行排队等待或者直接抛出异常。

线程池框架 Executor

Java 最开始提供了 ThreadPool 实现了线程池，为了更好地实现用户级的线程调度，更有效地帮助开发人员进行多线程开发，Java 提供了一套 Executor 框架。

这个框架中包括了 ScheduledThreadPoolExecutor 和 ThreadPoolExecutor 两个核心线程池。前者是用来定时执行任务，后者是用来执行被提交的任务。鉴于这两个线程池的核心原理是一样的，下面我们就重点看看 ThreadPoolExecutor 类是如何实现线程池的。

Executors 实现了以下四种类型的 ThreadPoolExecutor：

类型	特性
newCachedThreadPool	线程池的大小不固定，可灵活回收空闲线程，若无可回收，则新建线程
newFixedThreadPool	固定大小的线程池，当有新的任务提交，线程池中如果有空闲线程，则立即执行，否则新的任务会被缓存在一个任务队列中，等待线程池释放空闲线程
newScheduledThreadPool	定时线程池，支持定时及周期性任务执行
newSingleThreadExecutor	只创建一个线程，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序（FIFO-LIFO-优先级）执行

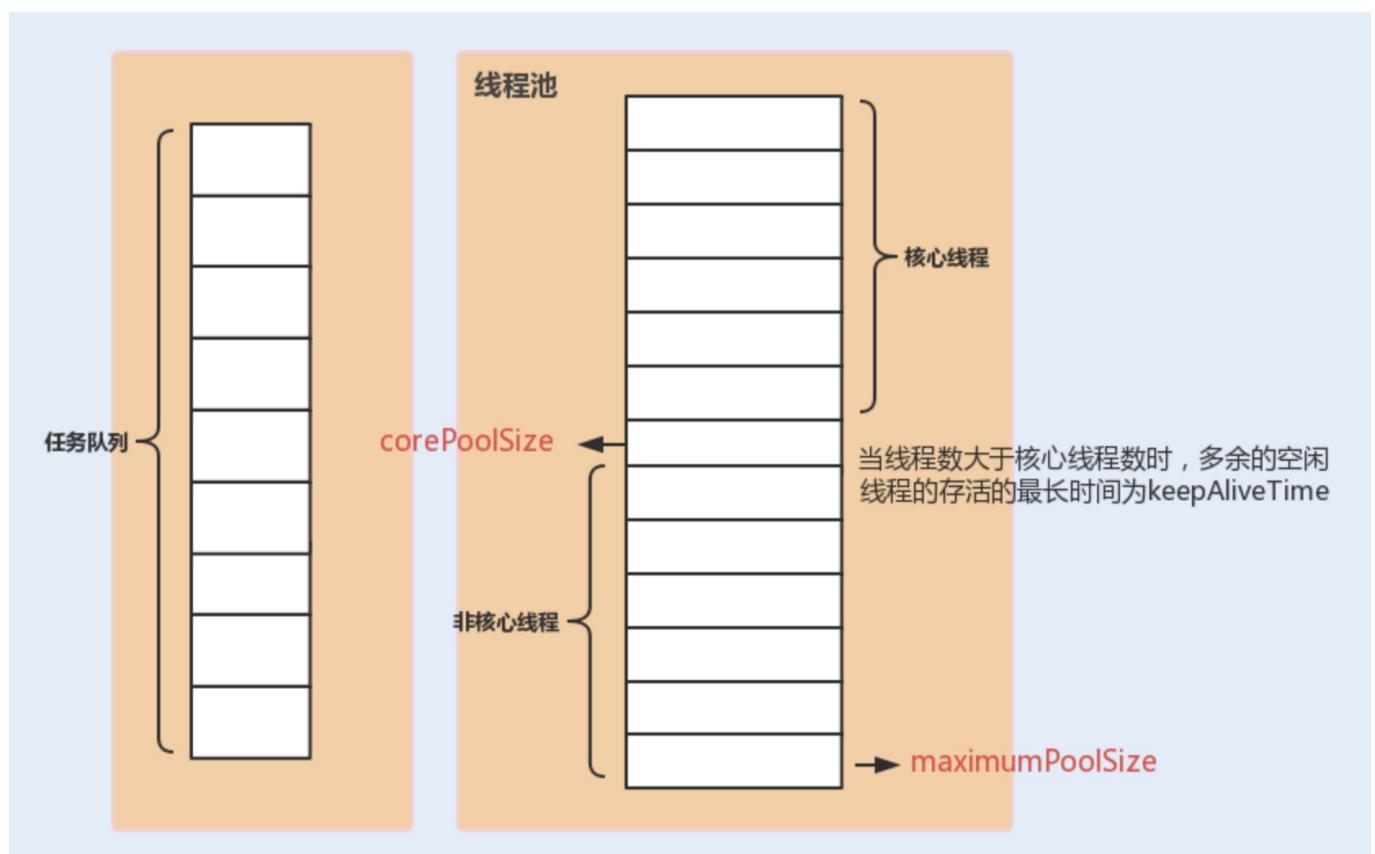
Executors 利用工厂模式实现的四种线程池，我们在使用的时候需要结合生产环境下的实际场景。不过我不太推荐使用它们，因为选择使用 Executors 提供的工厂类，将会忽略很多线程池的参数设置，工厂类一旦选择设置默认参数，就很容易导致无法调优参数设置，从而产生性能问题或者资源浪费。

这里我建议你使用 ThreadPoolExecutor 自我定制一套线程池。进入四种工厂类后，我们可以发现除了 newScheduledThreadPool 类，其它类均使用了 ThreadPoolExecutor 类进行实现，你可以通过以下代码简单看下该方法：

 复制代码

```
1 public ThreadPoolExecutor(int corePoolSize, // 线程池的核心线程数量
2                           int maximumPoolSize, // 线程池的最大线程数
3                           long keepAliveTime, // 当线程数大于核心线程数时，多余的空闲线程存活的最长时间
4                           TimeUnit unit, // 时间单位
5                           BlockingQueue<Runnable> workQueue, // 任务队列，用来储存等待执行的任务
6                           ThreadFactory threadFactory, // 线程工厂，用来创建线程，一般默
7                           RejectedExecutionHandler handler) // 拒绝策略，当提交的任务...
```

我们还可以通过下面这张图来了解下线程池中各个参数的相互关系：



通过上图，我们发现线程池有两个线程数的设置，一个为核心线程数，一个为最大线程数。在创建完线程池之后，默认情况下，线程池中并没有任何线程，等到有任务来才创建线程去执行任务。

但有一种情况排除在外，就是调用 `prestartAllCoreThreads()` 或者 `prestartCoreThread()` 方法的话，可以提前创建等于核心线程数的线程数量，这种方式被称为预热，在抢购系统中就经常被用到。

当创建的线程数等于 `corePoolSize` 时，提交的任务会被加入到设置的阻塞队列中。当队列满了，会创建线程执行任务，直到线程池中的数量等于 `maximumPoolSize`。

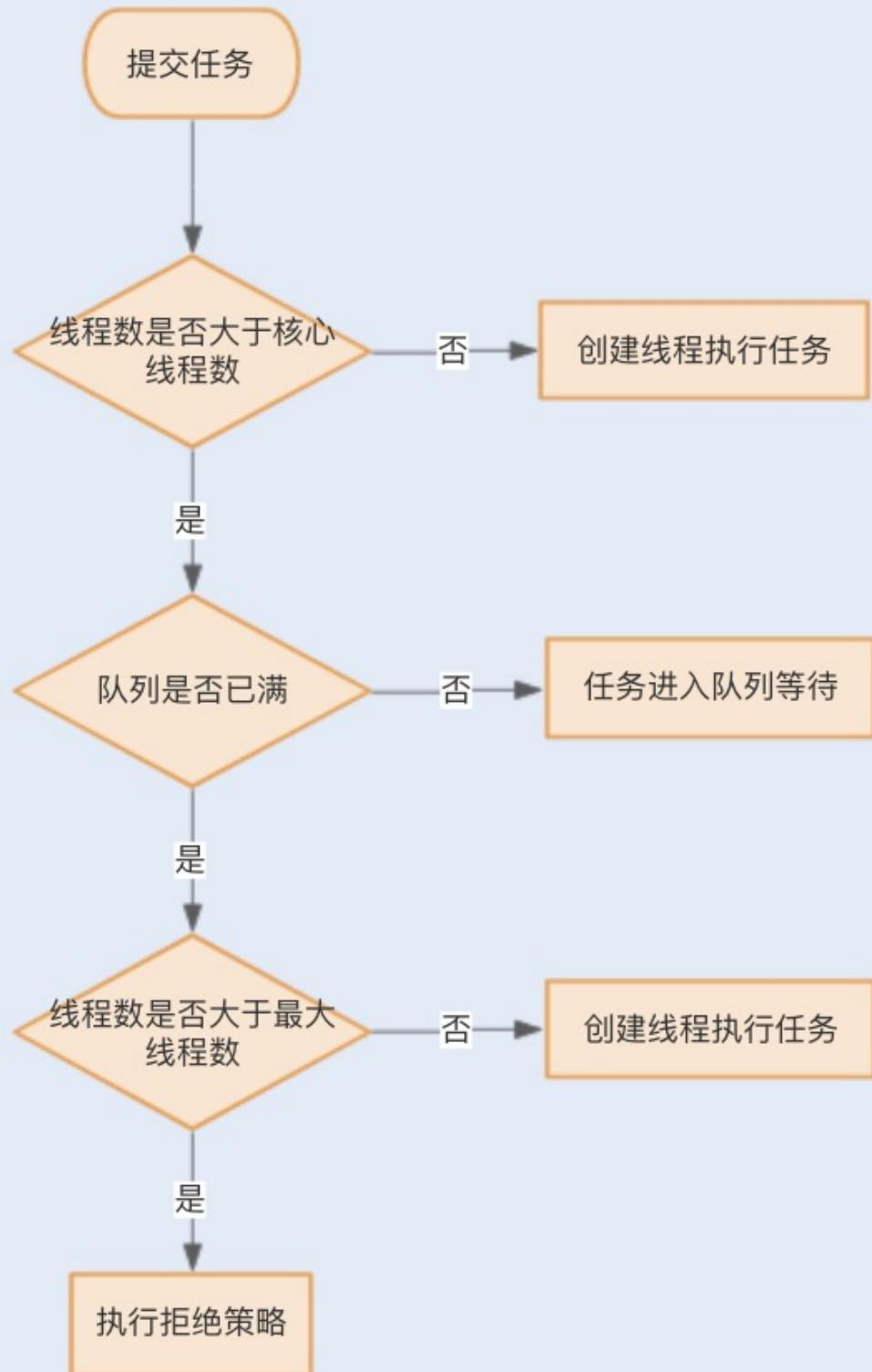
当线程数量已经等于 `maximumPoolSize` 时，新提交的任务无法加入到等待队列，也无法创建非核心线程直接执行，我们又没有为线程池设置拒绝策略，这时线程池就会抛出 `RejectedExecutionException` 异常，即线程池拒绝接受这个任务。

当线程池中创建的线程数量超过设置的 `corePoolSize`，在某些线程处理完任务后，如果等待 `keepAliveTime` 时间后仍然没有新的任务分配给它，那么这个线程将会被回收。线程池回收线程时，会对所谓的“核心线程”和“非核心线程”一视同仁，直到线程池中线程的数量等于设置的 `corePoolSize` 参数，回收过程才会停止。

即使是 `corePoolSize` 线程，在一些非核心业务的线程池中，如果长时间地占用线程数量，也可能会影响到核心业务的线程池，这个时候就需要把没有分配任务的线程回收掉。

我们可以通过 `allowCoreThreadTimeOut` 设置项要求线程池：将包括“核心线程”在内的，没有任务分配的所有线程，在等待 `keepAliveTime` 时间后全部回收掉。

我们可以通过下面这张图来了解下线程池的线程分配流程：



计算线程数量

了解完线程池的实现原理和框架，我们就可以动手实践优化线程池的设置了。

我们知道，环境具有多变性，设置一个绝对精准的线程数其实是不大可能的，但我们可以
通过一些实际操作因素来计算出一个合理的线程数，避免由于线程池设置不合理而导致的性
能问题。下面我们就来看看具体的计算方法。

一般多线程执行的任务类型可以分为 CPU 密集型和 I/O 密集型，根据不同的任务类型，我
们计算线程数的方法也不一样。

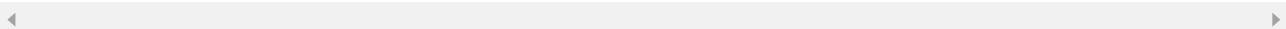
CPU 密集型任务：这种任务消耗的主要 CPU 资源，可以将线程数设置为 N (CPU 核心
数) + 1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原
因导致的任务暂停而带来的影响。一旦任务暂停， CPU 就会处于空闲状态，而在这种情况下
多出来的一个线程就可以充分利用 CPU 的空闲时间。

下面我们用一个例子来验证下这个方法的可行性，通过观察 CPU 密集型任务在不同线程数
下的性能情况就可以得出结果，你可以点击[Github](#)下载到本地运行测试：

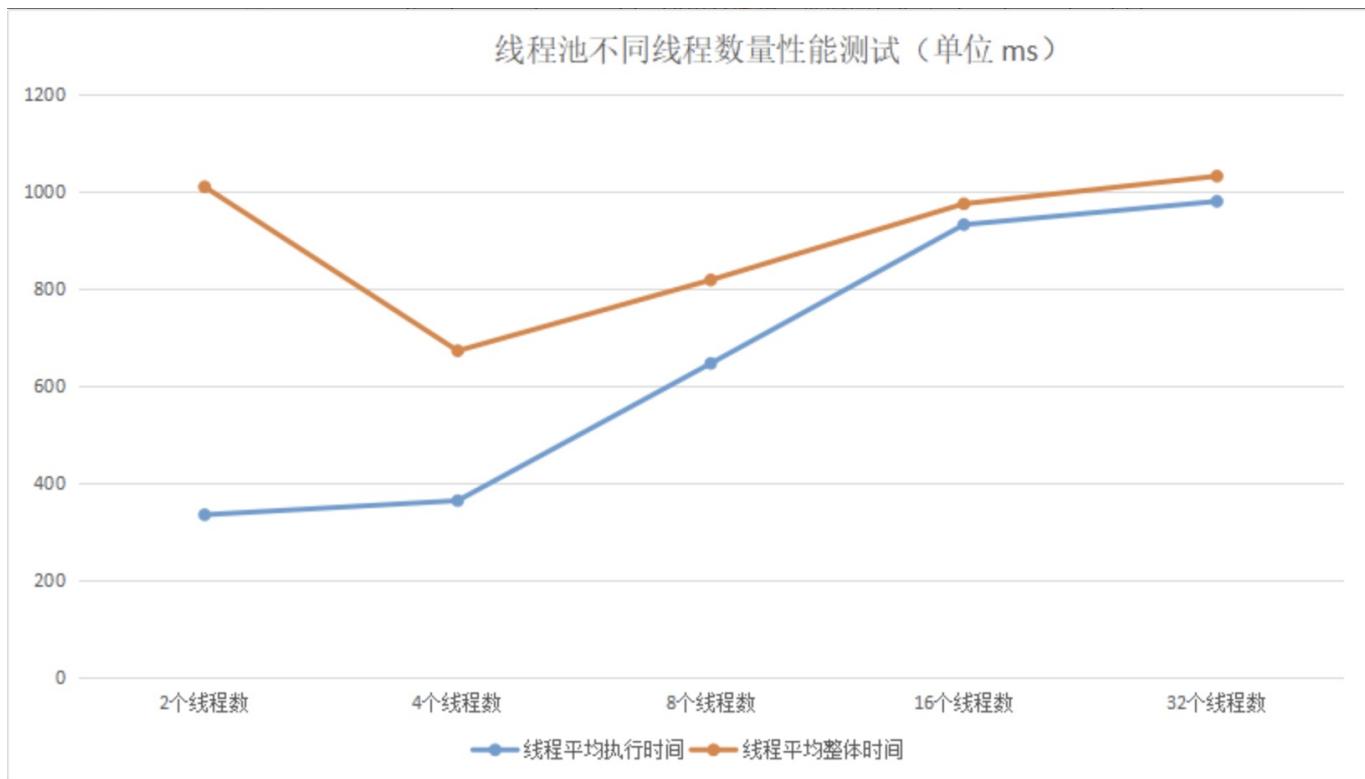
 复制代码

```
1 public class CPUPTypeTest implements Runnable {  
2  
3     // 整体执行时间，包括在队列中等待的时间  
4     List<Long> wholeTimeList;  
5     // 真正执行时间  
6     List<Long> runTimeList;  
7  
8     private long initStartTime = 0;  
9  
10    /**  
11     * 构造函数  
12     * @param runTimeList  
13     * @param wholeTimeList  
14     */  
15    public CPUPTypeTest(List<Long> runTimeList, List<Long> wholeTimeList) {  
16        initStartTime = System.currentTimeMillis();  
17        this.runTimeList = runTimeList;  
18        this.wholeTimeList = wholeTimeList;  
19    }  
20  
21    /**  
22     * 判断素数  
23     * @param number  
24     * @return  
25     */  
26    public boolean isPrime(final int number) {  
27        if (number <= 1)  
28            return false;
```

```
29
30
31         for (int i = 2; i <= Math.sqrt(number); i++) {
32             if (number % i == 0)
33                 return false;
34         }
35         return true;
36     }
37
38     /**
39      * 計算素數
40      * @param number
41      * @return
42      */
43     public int countPrimes(final int lower, final int upper) {
44         int total = 0;
45         for (int i = lower; i <= upper; i++) {
46             if (isPrime(i))
47                 total++;
48         }
49         return total;
50     }
51
52     public void run() {
53         long start = System.currentTimeMillis();
54         countPrimes(1, 1000000);
55         long end = System.currentTimeMillis();
56
57         long wholeTime = end - initStartTime;
58         long runTime = end - start;
59         wholeTimeList.add(wholeTime);
60         runTimeList.add(runTime);
61         System.out.println(" 单个线程花费时间: " + (end - start));
62     }
63 }
64 }
```



测试代码在 4 核 intel i5 CPU 机器上的运行时间变化如下：



综上可知：当线程数量太小，同一时间大量请求将被阻塞在线程队列中排队等待执行线程，此时 CPU 没有得到充分利用；当线程数量太大，被创建的执行线程同时在争取 CPU 资源，又会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。通过测试可知，4~6 个线程数是最合适的。

I/O 密集型任务：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 $2N$ 。

这里我们还是通过一个例子来验证下这个公式是否可以标准化：

复制代码

```

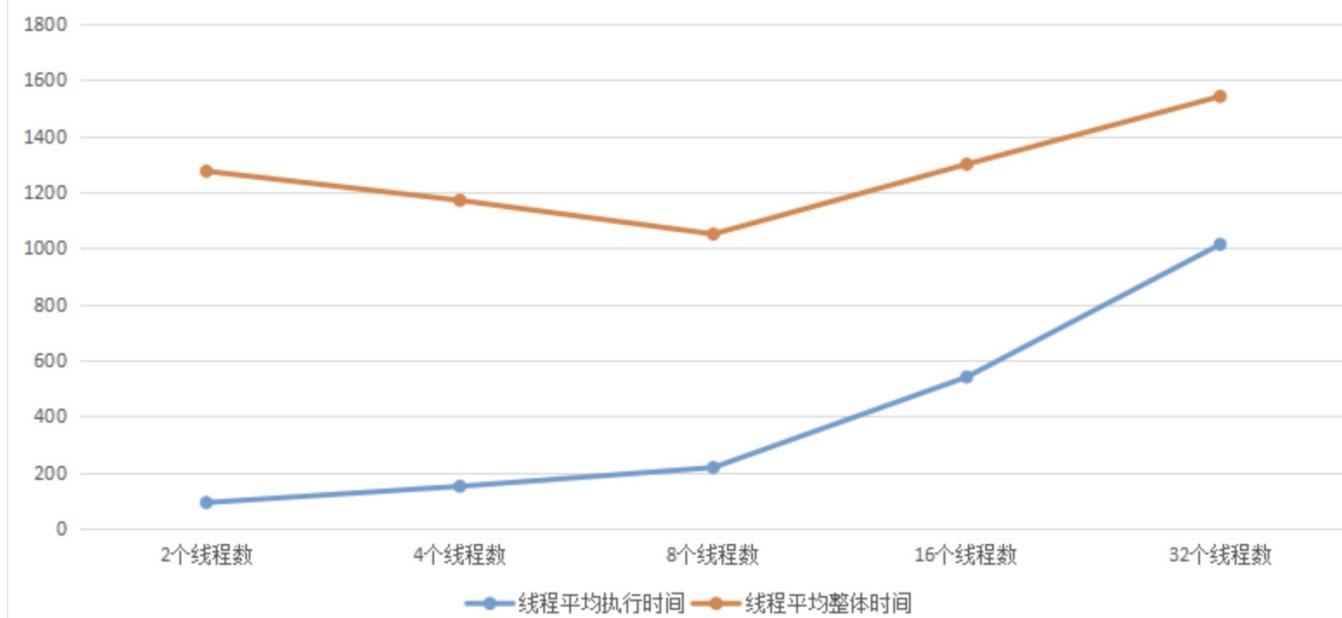
1 public class IOTypeTest implements Runnable {
2
3     // 整体执行时间，包括在队列中等待的时间
4     Vector<Long> wholeTimeList;
5     // 真正执行时间
6     Vector<Long> runTimeList;
7
8     private long initStartTime = 0;
9
10    /**
11     * 构造函数
12     * @param runTimeList
13     * @param wholeTimeList

```

```
14     */
15     public IOTypeTest(Vector<Long> runTimeList, Vector<Long> wholeTimeList) {
16         initStartTime = System.currentTimeMillis();
17         this.runTimeList = runTimeList;
18         this.wholeTimeList = wholeTimeList;
19     }
20
21     /**
22      *IO 操作
23      * @param number
24      * @return
25      * @throws IOException
26      */
27     public void readAndWrite() throws IOException {
28         File sourceFile = new File("D:/test.txt");
29         // 创建输入流
30         BufferedReader input = new BufferedReader(new FileReader(sourceFile));
31         // 读取源文件，写入到新的文件
32         String line = null;
33         while((line = input.readLine()) != null){
34             //System.out.println(line);
35         }
36         // 关闭输入输出流
37         input.close();
38     }
39
40     public void run() {
41         long start = System.currentTimeMillis();
42         try {
43             readAndWrite();
44         } catch (IOException e) {
45             // TODO Auto-generated catch block
46             e.printStackTrace();
47         }
48         long end = System.currentTimeMillis();
49
50
51         long wholeTime = end - initStartTime;
52         long runTime = end - start;
53         wholeTimeList.add(wholeTime);
54         runTimeList.add(runTime);
55         System.out.println(" 单个线程花费时间: " + (end - start));
56     }
57 }
```

备注：由于测试代码读取 2MB 大小的文件，涉及到大内存，所以在运行之前，我们需要调整 JVM 的堆内存空间：`-Xms4g -Xmx4g`，避免发生频繁的 FullGC，影响测试结果。

I/O密集型线程池不同线程数量性能测试（单位 ms）



通过测试结果，我们可以看到每个线程所花费的时间。当线程数量在 8 时，线程平均执行时间是最佳的，这个线程数量和我们的计算公式所得的结果就差不多。

看完以上两种情况下的线程计算方法，你可能还想说，在平常的应用场景中，我们常常遇不到这两种极端情况，那么碰上一些常规的业务操作，比如，通过一个线程池实现向用户定时推送消息的业务，我们又该如何设置线程池的数量呢？

此时我们可以参考以下公式来计算线程数：

复制代码

1 线程数 = $N(\text{CPU 核数}) * (1 + \text{WT}(\text{线程等待时间}) / \text{ST}(\text{线程时间运行时间}))$

◀ ▶

我们可以通过 JDK 自带的工具 VisualVM 来查看 WT/ST 比例，以下例子是基于运行纯 CPU 运算的例子，我们可以看到：

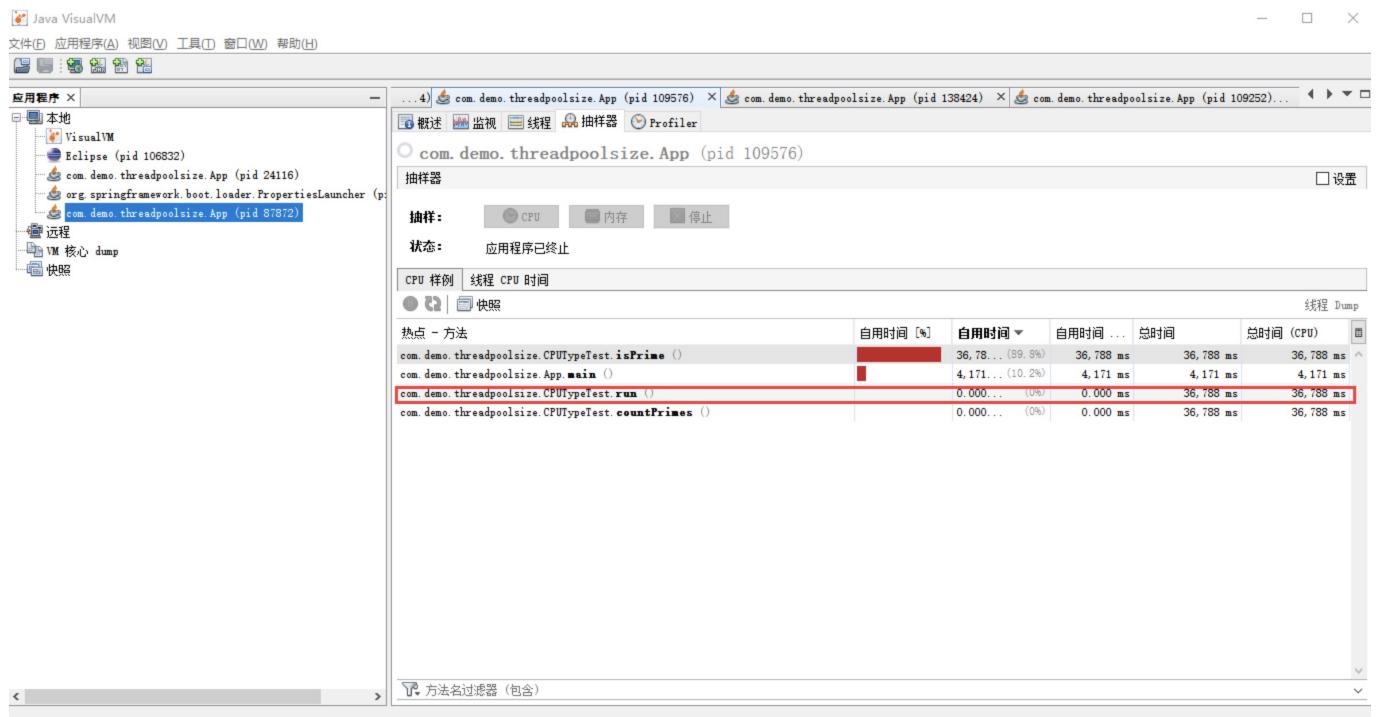
复制代码

1 WT (线程等待时间) = 36788ms [线程运行总时间] - 36788ms[ST (线程时间运行时间)] = 0

2 线程数 = $N(\text{CPU 核数}) * (1 + 0 [\text{WT}(\text{线程等待时间})] / 36788ms[\text{ST}(\text{线程时间运行时间})]) = N(\text{CPU})$

◀ ▶

这跟我们之前通过 CPU 密集型的计算公式 $N+1$ 所得出的结果差不多。



综合来看，我们可以根据自己的业务场景，从“ $N+1$ ”和“ $2N$ ”两个公式中选出一个适合的，计算出一个大概的线程数量，之后通过实际压测，逐渐往“增大线程数量”和“减小线程数量”这两个方向调整，然后观察整体的处理时间变化，最终确定一个具体的线程数量。

总结

今天我们主要学习了线程池的实现原理，Java 线程的创建和消耗会给系统带来性能开销，因此 Java 提供了线程池来复用线程，提高程序的并发效率。

Java 通过用户线程与内核线程结合的 1:1 线程模型来实现，Java 将线程的调度和管理设置在了用户态，提供了一套 Executor 框架来帮助开发人员提高效率。Executor 框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等，可以说 Executor 框架为并发编程提供了一个完善的架构体系。

在不同的业务场景以及不同配置的部署机器中，线程池的线程数量设置是不一样的。其设置不宜过大，也不宜过小，要根据具体情况，计算出一个大概的数值，再通过实际的性能测试，计算出一个合理的线程数量。

我们要提高线程池的处理能力，一定要先保证一个合理的线程数量，也就是保证 CPU 处理线程的最大化。在此前提下，我们再增大线程池队列，通过队列将来不及处理的线程缓存起

来。在设置缓存队列时，我们要尽量使用一个有界队列，以防因队列过大而导致的内存溢出问题。

19 | 如何用协程来优化多线程业务？



近一两年，国内很多互联网公司开始使用或转型 Go 语言，其中一个很重要的原因就是 Go 语言优越的性能表现，而这个优势与 Go 实现的轻量级线程 Goroutines（协程

Coroutine) 不无关系。那么 Go 协程的实现与 Java 线程的实现有什么区别呢？

线程实现模型

了解协程和线程的区别之前，我们不妨先来了解下底层实现线程几种方式，为后面的学习打个基础。

实现线程主要有三种方式：轻量级进程和内核线程一对一相互映射实现的 1:1 线程模型、用户线程和内核线程实现的 N:1 线程模型以及用户线程和轻量级进程混合实现的 N:M 线程模型。

1:1 线程模型

以上我提到的内核线程（Kernel-Level Thread, KLT）是由操作系统内核支持的线程，内核通过调度器对线程进行调度，并负责完成线程的切换。

我们知道在 Linux 操作系统编程中，往往都是通过 fork() 函数创建一个子进程来代表一个内核中的线程。一个进程调用 fork() 函数后，系统会先给新的进程分配资源，例如，存储数据和代码的空间。然后把原来进程的所有值都复制到新的进程中，只有少数值与原来进程的值（比如 PID）不同，这相当于复制了一个主进程。

采用 fork() 创建子进程的方式来实现并行运行，会产生大量冗余数据，即占用大量内存空间，又消耗大量 CPU 时间用来初始化内存空间以及复制数据。

如果是一份一样的数据，为什么不共享主进程的这一份数据呢？这时候轻量级进程（Light Weight Process，即 LWP）出现了。

相对于 fork() 系统调用创建的线程来说，LWP 使用 clone() 系统调用创建线程，该函数是将部分父进程的资源的数据结构进行复制，复制内容可选，且没有被复制的资源可以通过指针共享给子进程。因此，轻量级进程的运行单元更小，运行速度更快。LWP 是跟内核线程一对一映射的，每个 LWP 都是由一个内核线程支持。

N:1 线程模型

1:1 线程模型由于跟内核是一对一映射，所以在线程创建、切换上都存在用户态和内核态的切换，性能开销比较大。除此之外，它还存在局限性，主要就是指系统的资源有限，不能支持创建大量的 LWP。

N:1 线程模型就可以很好地解决 1:1 线程模型的这两个问题。

该线程模型是在用户空间完成了线程的创建、同步、销毁和调度，已经不需要内核的帮助了，也就是说在线程创建、同步、销毁的过程中不会产生用户态和内核态的空间切换，因此线程的操作非常快速且低消耗。

N:M 线程模型

N:1 线程模型的缺点在于操作系统不能感知用户态的线程，因此容易造成某一个线程进行系统调用内核线程时被阻塞，从而导致整个进程被阻塞。

N:M 线程模型是基于上述两种线程模型实现的一种混合线程管理模型，即支持用户态线程通过 LWP 与内核线程连接，用户态的线程数量和内核态的 LWP 数量是 N:M 的映射关系。

了解完这三个线程模型，你就可以清楚地了解到 Go 协程的实现与 Java 线程的实现有什么区别了。

JDK 1.8 Thread.java 中 Thread#start 方法的实现，实际上是通过 Native 调用 start0 方法实现的；在 Linux 下，JVM Thread 的实现是基于 pthread_create 实现的，而 pthread_create 实际上是调用了 clone() 完成系统调用创建线程的。

所以，目前 Java 在 Linux 操作系统下采用的是用户线程加轻量级线程，一个用户线程映射到一个内核线程，即 1:1 线程模型。由于线程是通过内核调度，从一个线程切换到另一个线程就涉及到了上下文切换。

而 Go 语言是使用了 N:M 线程模型实现了自己的调度器，它在 N 个内核线程上多路复用（或调度）M 个协程，协程的上下文切换是在用户态由协程调度器完成的，因此不需要陷入内核，相比之下，这个代价就很小了。

协程的实现原理

协程不只在 Go 语言中实现了，其实目前大部分语言都实现了自己的一套协程，包括 C#、erlang、python、lua、javascript、ruby 等。

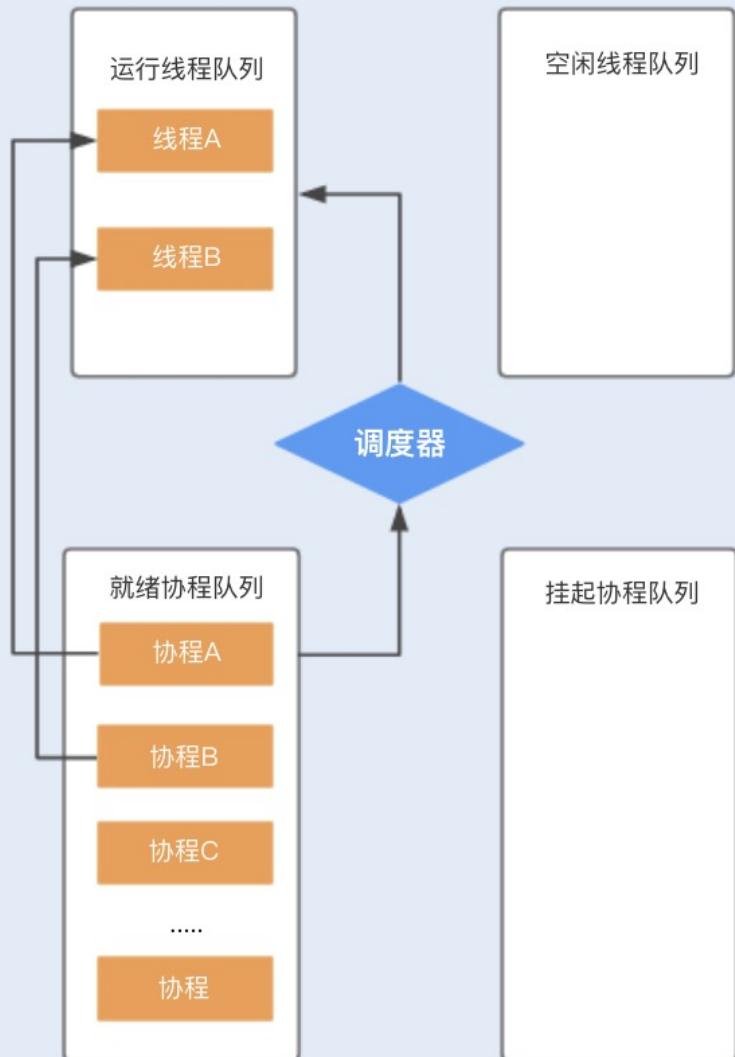
相对于协程，你可能对进程和线程更为熟悉。进程一般代表一个应用服务，在一个应用服务中可以创建多个线程，而协程与进程、线程的概念不一样，**我们可以将协程看作是一个类函数或者一块函数中的代码**，我们可以在一个主线程里面轻松创建多个协程。

程序调用协程与调用函数不一样的是，协程可以通过暂停或者阻塞的方式将协程的执行挂起，而其它协程可以继续执

行。这里的挂起只是在程序中（用户态）的挂起，同时将代码执行权转让给其它协程使用，待获取执行权的协程执行完成之后，将从挂起点唤醒挂起的协程。协程的挂起和唤醒是通过一个调度器来完成的。

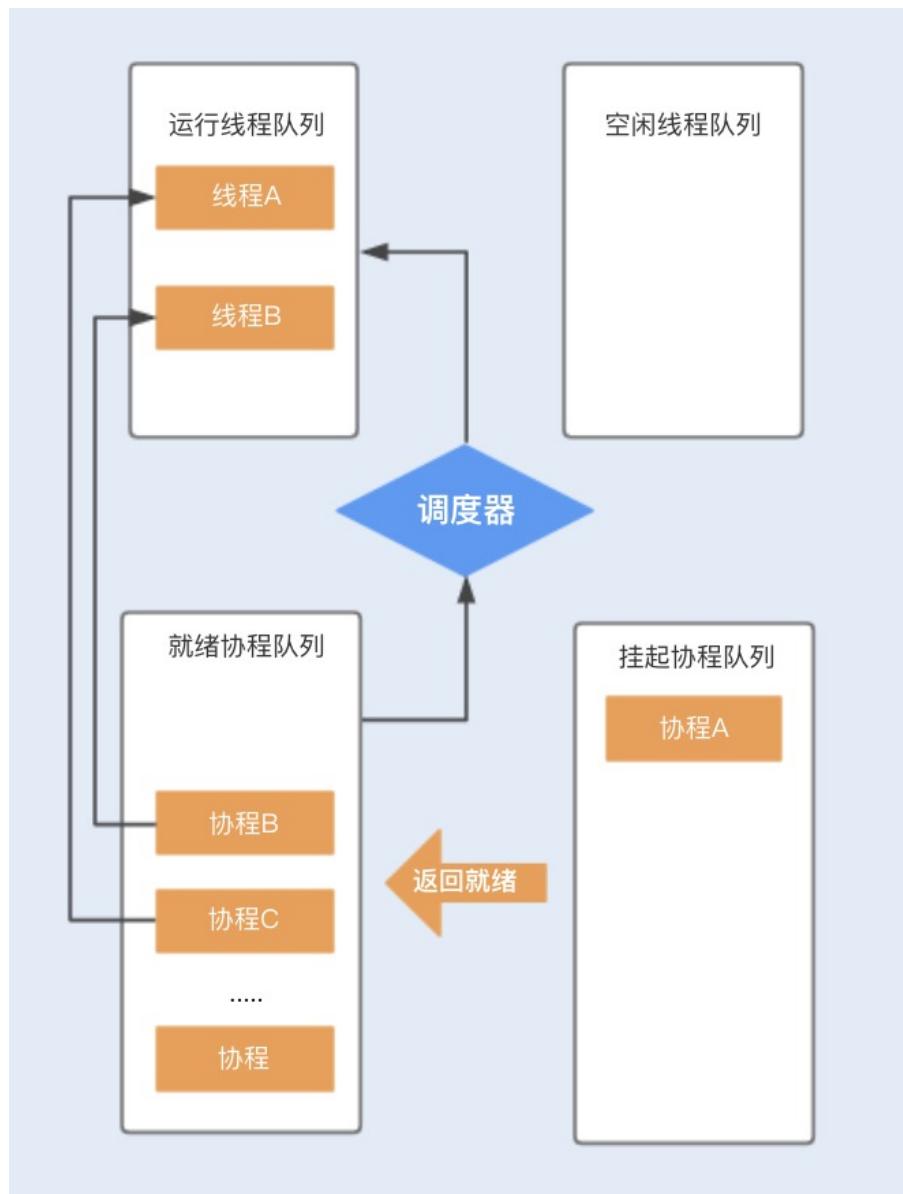
结合下图，你可以更清楚地了解到基于 N:M 线程模型实现的协程是如何工作的。

假设程序中默认创建两个线程为协程使用，在主线程中创建协程 ABCD...，分别存储在就绪队列中，调度器首先会分配一个工作线程 A 执行协程 A，另外个工作线程 B 执行协程 B，其它创建的协程将会放在队列中进行排队等待。



当协程 A 调用暂停方法或被阻塞时，协程 A 会进入到挂起队列，调度器会调用等待队列中的其它协程抢占线程 A 执行。当协程 A 被唤醒时，它需要重新进入到就绪队列中，通

通过调度器抢占线程，如果抢占成功，就继续执行协程 A，失败则继续等待抢占线程。



相比线程，协程少了由于同步资源竞争带来的 CPU 上下文切换，I/O 密集型的应用比较适合使用，特别是在网络请求中，有较多的时间在等待后端响应，协程可以保证线程不会阻塞在等待网络响应中，充分利用了多核多线程的能力。而对于 CPU 密集型的应用，由于在多数情况下 CPU 都比较繁忙，协程的优势就不是特别明显了。

Kilim 协程框架

虽然这么多的语言都实现了协程，但目前 Java 原生语言暂时还不支持协程。不过你也不用泄气，我们可以通过协程框架在 Java 中使用协程。

目前 Kilim 协程框架在 Java 中应用得比较多，通过这个框架，开发人员就可以低成本地在 Java 中使用协程了。

在 Java 中引入 [Kilim](#)，和我们平时引入第三方组件不太一样，除了引入 jar 包之外，还需要通过 Kilim 提供的织入（Weaver）工具对 Java 代码编译生成的字节码进行增强处理，比如，识别哪些方式是可暂停的，对相关的方法添加上下文处理。通常有以下四种方式可以实现这种织入操作：

在编译时使用 maven 插件；

在运行时调用 kilim.tools.Weaver 工具；

在运行时使用 `kilim.tools.Kilim` invoking 调用 Kilim 的类文件；

在 `main` 函数添加 if
(`kilim.tools.Kilim.trampoline(false,args)`) return。

Kilim 框架包含了四个核心组件，分别为：任务载体（Task）、任务上下文（Fiber）、任务调度器（Scheduler）以及通信载体（Mailbox）。



Task 对象主要用来执行业务逻辑，我们可以把这个比作多线程的 Thread，与 Thread 类似，Task 中也有一个 run 方

法，不过在 Task 中方法名为 execute，我们可以将协程里面要做的业务逻辑操作写在 execute 方法中。

与 Thread 实现的线程一样，Task 实现的协程也有状态，包括：Ready、Running、Pausing、Paused 以及 Done 总共五种。Task 对象被创建后，处于 Ready 状态，在调用 execute() 方法后，协程处于 Running 状态，在运行期间，协程可以被暂停，暂停中的状态为 Pausing，暂停后的状态为 Paused，暂停后的协程可以被再次唤醒。协程正常结束后的状态为 Done。

Fiber 对象与 Java 的线程栈类似，主要用来维护 Task 的执行堆栈，Fiber 是实现 N:M 线程映射的关键。

Scheduler 是 Kilim 实现协程的核心调度器，Scheduler 负责分派 Task 给指定的工作者线程 WorkerThread 执行，工作者线程 WorkerThread 默认初始化个数为机器的 CPU 个数。

Mailbox 对象类似一个邮箱，协程之间可以依靠邮箱来进行通信和数据共享。协程与线程最大的不同就是，线程是通过共享内存来实现数据共享，而协程是使用了通信的方式来实现了数据共享，主要就是为了避免内存共享数据而带来的线程安全问题。

协程与线程的性能比较

接下来，我们通过一个简单的生产者和消费者的案例，来对比下协程和线程的性能。可通过 [Github](#) 下载本地运行代码。

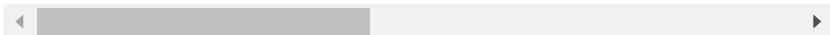
Java 多线程实现源码：

 复制代码

```
1 public class MyThread {  
2     private static Integer count = 0; //  
3     private static final Integer FULL = 10; // 最大  
4     private static String LOCK = "lock"; // 资源锁  
5  
6     public static void main(String[] args) {  
7         MyThread test1 = new MyThread();  
8  
9         long start = System.currentTimeMillis()  
10  
11         List<Thread> list = new ArrayList<Thread>();  
12         for (int i = 0; i < 1000; i++) { // 创建  
13             Thread thread = new Thread(test1);  
14             thread.start();  
15             list.add(thread);  
16         }  
17  
18         for (int i = 0; i < 1000; i++) { // 创建  
19             Thread thread = new Thread(test1);  
20             thread.start();  
21             list.add(thread);  
22         }  
23     }
```

```
24         try {
25             for (Thread thread : list) {
26                 thread.join(); // 等待所
27             }
28         } catch (InterruptedException e) {
29             e.printStackTrace();
30         }
31
32         long end = System.currentTimeMillis();
33         System.out.println("子线程执行时长: " +
34     }
35 // 生产者
36     class Producer implements Runnable {
37         public void run() {
38             for (int i = 0; i < 10; i++) {
39                 synchronized (LOCK) {
40                     while (count ==
41                         try {
42
43                     } catch
44
45                 }
46             }
47             count++;
48             System.out.prir
49             LOCK.notifyAll(
50                 }
51             }
52         }
53     }
54 // 消费者
55     class Consumer implements Runnable {
56         public void run() {
57             for (int i = 0; i < 10; i++) {
58                 synchronized (LOCK) {
```

```
59                     while (count == 0) {
60                         try {
61                             ...
62                         } catch (Exception e) {
63                             ...
64                         }
65                         count--;
66                         System.out.println("Count: " + count);
67                         LOCK.notifyAll();
68                     }
69                 }
70             }
71         }
72     }
```

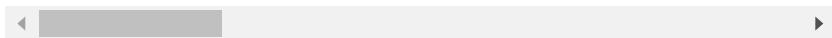


Kilim 协程框架实现源码：

复制代码

```
1 public class Coroutine {
2
3     static Map<Integer, Mailbox<Integer>> m
4
5     public static void main(String[] args) {
6
7         if (kilim.tools.Kilim.trampoline(false)) {
8             Properties props = new Properties();
9             props.setProperty("kilim.Scheduler.numThreads", "1");
10            System.setProperties(props);
11            long startTime = System.currentTimeMillis();
12            for (int i = 0; i < 1000; i++) { // 创建一个线程池
13                Mailbox<Integer> mb = new Mailbox<Integer>(i);
14                mb.start();
15            }
16        }
17    }
18}
```

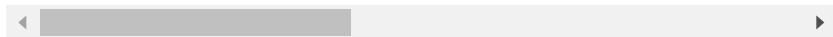
```
14             new Producer(i, mb).start();
15             mailMap.put(i, mb);
16         }
17
18         for (int i = 0; i < 1000; i++) {// 创建
19             new Consumer(mailMap.get(i)).st
20         }
21
22         Task.idledown();// 开始运行
23
24         long endTime = System.currentTimeMillis();
25
26         System.out.println( Thread.currentThread()
27     }
28
29 }
```



复制代码

```
1 // 生产者
2 public class Producer extends Task<Object> {
3
4     Integer count = null;
5     Mailbox<Integer> mb = null;
6
7     public Producer(Integer count, Mailbox<Integer>
8                     this.count = count;
9                     this.mb = mb;
10    }
11
12    public void execute() throws Pausable {
13        count = count*10;
14        for (int i = 0; i < 10; i++) {
```

```
15                         mb.put(count); // 当空间不足时，阻塞  
16                         System.out.println(Thread.currentThread().  
17                                         getName());  
18                     }  
19                 }  
20             }
```



复制代码

```
1 // 消费者  
2 public class Consumer extends Task<Object> {  
3  
4     Mailbox<Integer> mb = null;  
5  
6     public Consumer(Mailbox<Integer> mb) {  
7         this.mb = mb;  
8     }  
9  
10    /**  
11     * 执行  
12     */  
13    public void execute() throws Pausable {  
14        Integer c = null;  
15        for (int i = 0; i < 10000; i++) {  
16            c = mb.get(); // 获取消息，阻塞协程  
17  
18            if (c == null) {  
19                System.out.println(" 计数器空了");  
20            } else {  
21                System.out.println(Thread.currentThread().  
22                                         getName());  
23            }  
24        }  
25    }
```

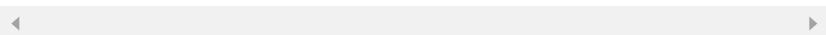
```
25         }
26 }
```



在这个案例中，我创建了 1000 个生产者和 1000 个消费者，每个生产者生产 10 个产品，1000 个消费者同时消费产品。我们可以看到两个例子运行的结果如下：

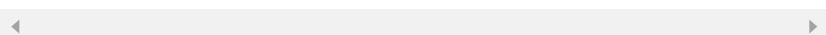
复制代码

1 多线程执行时长： 2761



复制代码

1 协程执行时长： 1050



通过上述性能对比，我们可以发现：在有严重阻塞的场景下，协程的性能更胜一筹。其实，**I/O 阻塞型场景也就是协程在 Java 中的主要应用。**

总结

协程和线程密切相关，协程可以认为是运行在线程上的代码块，协程提供的挂起操作会使协程暂停执行，而不会导致线程阻塞。

协程又是一种轻量级资源，即使创建了上千个协程，对于系统来说也不是很大的负担，但如果在程序中创建上千个线程，那系统可真就压力山大了。可以说，**协程的设计方式极大地提高了线程的使用率。**

通过今天的学习，当其他人侃侃而谈 Go 语言在网络编程中的优势时，相信你不会一头雾水。学习 Java 的我们也不要觉得，协程离我们很遥远了。协程是一种设计思想，不仅仅局限于某一门语言，况且 Java 已经可以借助协程框架实现协程了。

但话说回来，协程还是在 Go 语言中的应用较为成熟，在 Java 中的协程目前还不是很稳定，重点是缺乏大型项目的验证，可以说 Java 的协程设计还有很长的路要走。

什么是数据的强、弱一致性？



在第 17 讲讲解并发容器的时候，我提到了“强一致性”和“弱一致性”。很多同学留言表示对这个概念没有了解或者比较模糊，今天这讲加餐就来详解一下。

说到一致性，其实在系统的很多地方都存在数据一致性的相关问题。除了在并发编程中保证共享变量数据的一致性之外，还有数据库的 ACID 中的 C (Consistency 一致性)、分布式系统的 CAP 理论中的 C (Consistency 一致性)。下面我们主要讨论的就是“**并发编程中共享变量的一致性**”。

在并发编程中，Java 是通过共享内存来实现共享变量操作的，所以在多线程编程中就会涉及到数据一致性的问题。

我先通过一个经典的案例来说明下多线程操作共享变量可能出现的问题，假设我们有两个线程（线程 1 和线程 2）分别执行下面的方法，x 是共享变量：

 复制代码

```
1 // 代码 1
2 public class Example {
3     int x = 0;
4     public void count() {
5         x++;                         //1
6         System.out.println(x)//2
7     }
8 }
```



线程1 调用 count	线程2 调用count
x++;	x++;

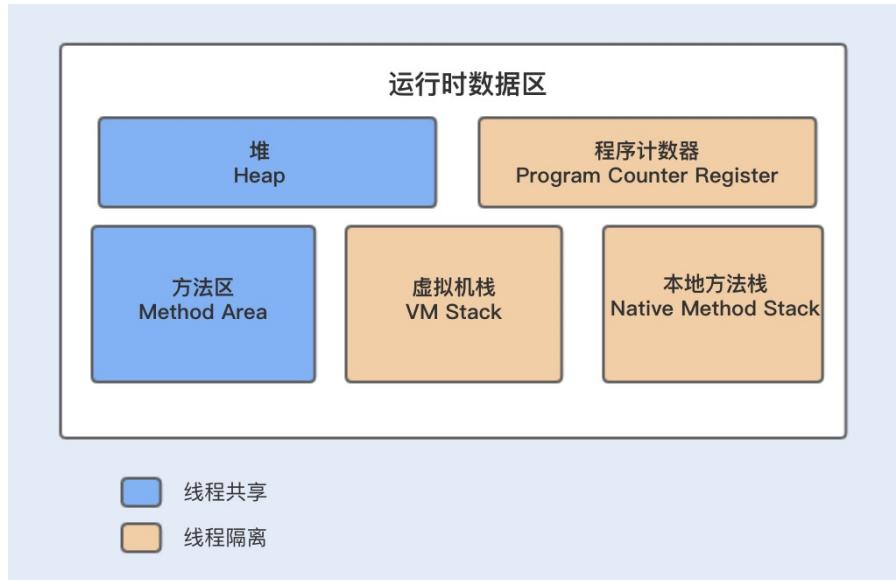
如果两个线程同时运行，两个线程的变量的值可能会出现以下三种结果：

结果1	结果2	结果3
1,1	2,1	1,2

Java 存储模型

2,1 和 1,2 的结果我们很好理解，那为什么会出现以上 1,1 的结果呢？

我们知道，Java 采用共享内存模型来实现多线程之间的信息交换和数据同步。在解释为什么会出现这样的结果之前，我们先通过下图来简单了解下 Java 的内存模型（第 21 讲还会详解），程序在运行时，局部变量将会存放在虚拟机栈中，而共享变量将会被保存在堆内存中。



由于局部变量是跟随线程的创建而创建，线程的销毁而销毁，所以存放在栈中，由上图我们可知，Java 栈数据不是所有线程共享的，所以不需要关心其数据的一致性。

共享变量存储在堆内存或方法区中，由上图可知，堆内存和方法区的数据是线程共享的。而堆内存中的共享变量在被不同线程操作时，会被加载到自己的工作内存中，也就是 CPU 中的高速缓存。

CPU 缓存可以分为一级缓存 (L1)、二级缓存 (L2) 和三级缓存 (L3)，每一级缓存中所储存的全部数据都是下一级缓存的一部分。当 CPU 要读取一个缓存数据时，首先会从

一级缓存中查找；如果没有找到，再从二级缓存中查找；如果还是没有找到，就从三级缓存或内存中查找。

如果是单核 CPU 运行多线程，多个线程同时访问进程中的共享数据，CPU 将共享变量加载到高速缓存后，不同线程在访问缓存数据的时候，都会映射到相同的缓存位置，这样即使发生线程的切换，缓存仍然不会失效。

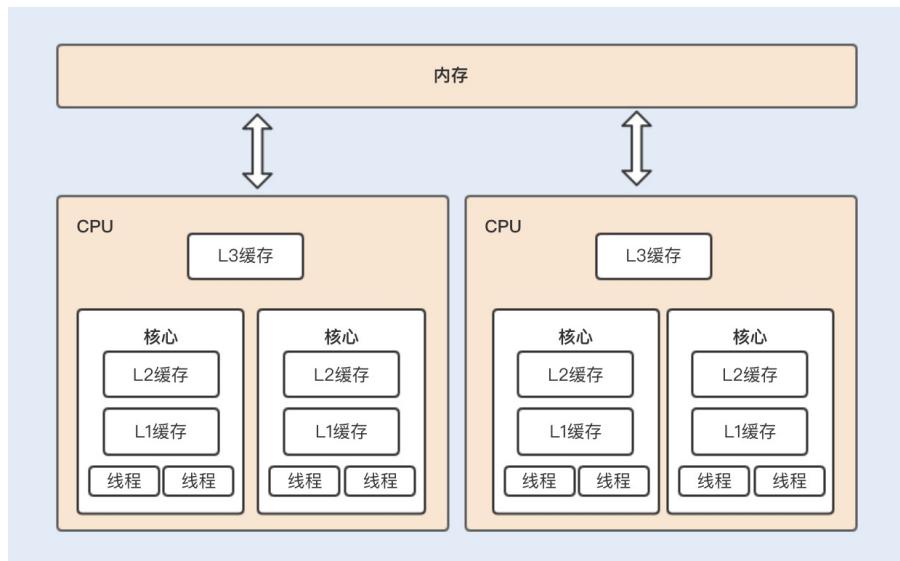
如果是多核 CPU 运行多线程，每个核都有一个 L1 缓存，如果多个线程运行在不同的内核上访问共享变量时，每个内核的 L1 缓存将会缓存一份共享变量。

假设线程 A 操作 CPU 从堆内存中获取一个缓存数据，此时堆内存中的缓存数据值为 0，该缓存数据会被加载到 L1 缓存中，在操作后，缓存数据的值变为 1，然后刷新到堆内存中。

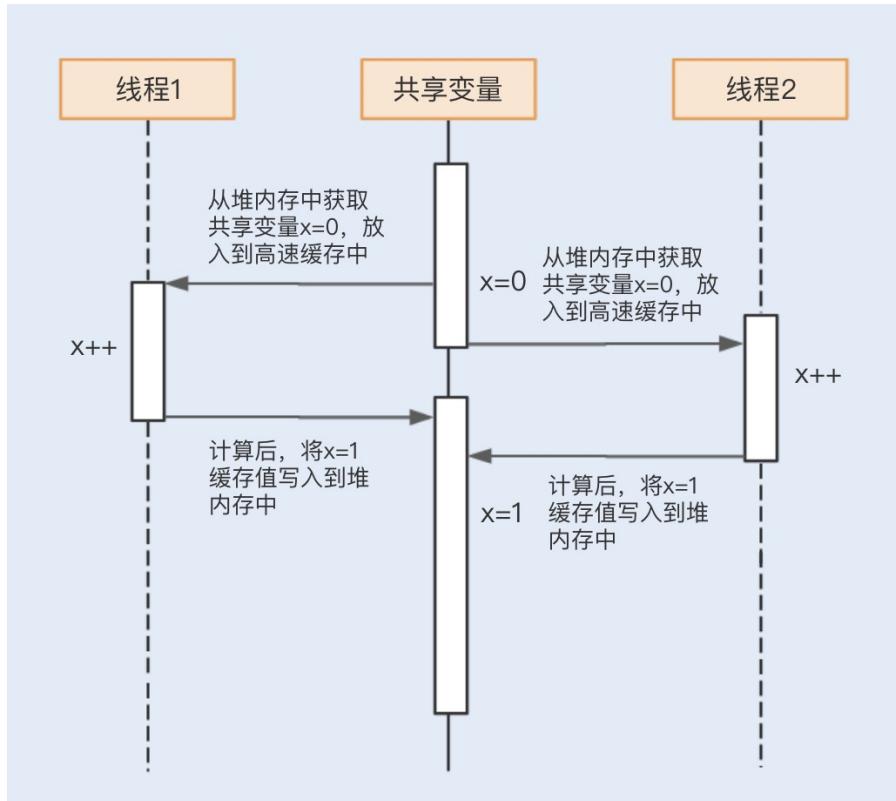
在正好刷新到堆内存中之前，又有另外一个线程 B 将堆内存中为 0 的缓存数据加载到了另外一个内核的 L1 缓存中，此时线程 A 将堆内存中的数据刷新到了 1，而线程 B 实际拿到的缓存数据的值为 0。

此时，内核缓存中的数据和堆内存中的数据就不一致了，且线程 B 在刷新缓存到堆内存中的时候也将覆盖线程 A 中修

改的数据。这时就产生了数据不一致的问题。



了解完内存模型之后，结合以上解释，我们就可以回过头来看看第一段代码中的运行结果是如何产生的了。看到这里，相信你可以理解图中 1,1 的运行结果了。



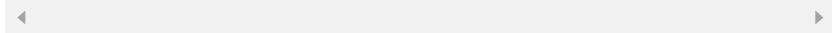
重排序

除此之外，在 Java 内存模型中，还存在重排序的问题。请看以下代码：

复制代码

```
1 // 代码 1
2 public class Example {
3     int x = 0;
4     boolean flag = false;
5     public void writer() {
```

```
6         x = 1;           //1
7         flag = true;      //2
8     }
9
10    public void reader() {
11        if (flag) {          //3
12            int r1 = x;       //4
13            System.out.println(r1==x)
14        }
15    }
16 }
```

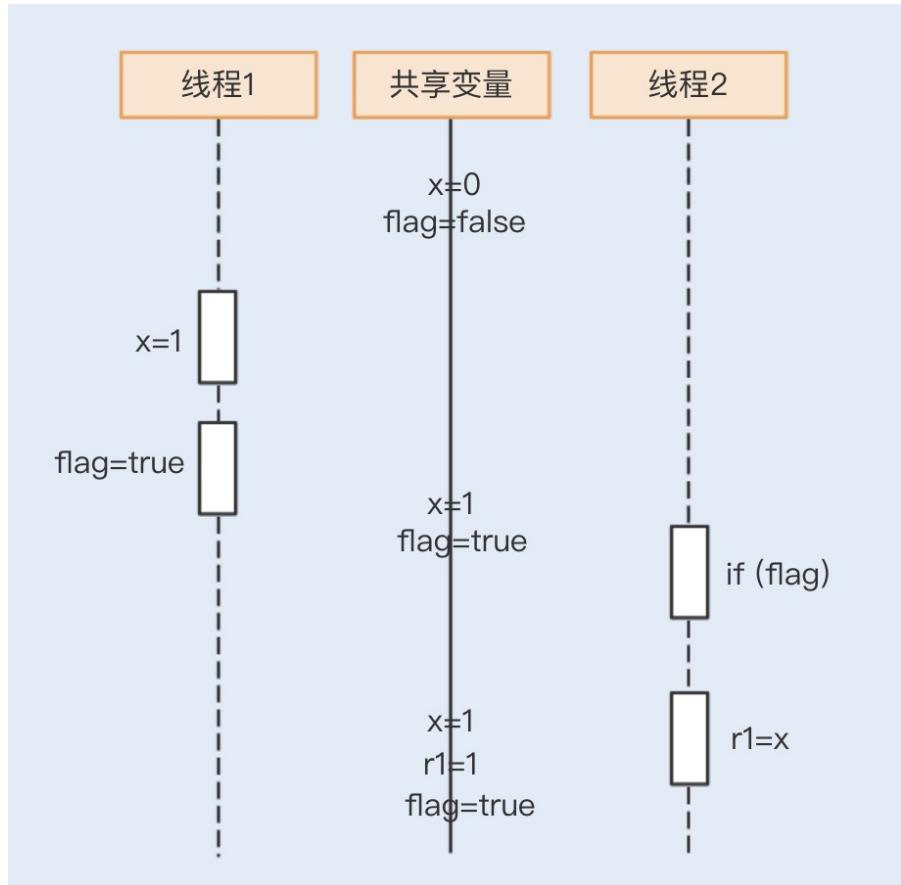


线程1 调用writer	线程2 调用reader
x = 1;	if (flag)
flag = true	int r1 = x;

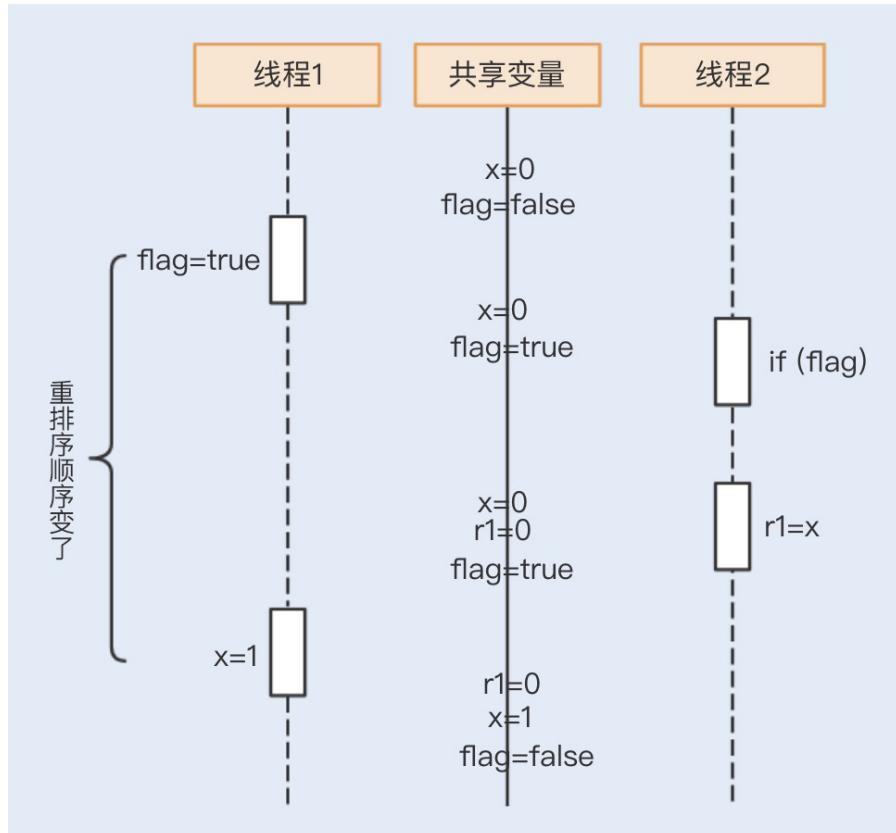
如果两个线程同时运行，线程 2 中的变量的值可能会出现以下两种可能：

结果1	结果2
r1=0	r1=1

现在一起来看看 $r1=1$ 的运行结果，如下图所示：



那 `r1=0` 又是怎么获取的呢？我们再来看一个时序图：

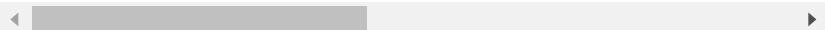


在不影响运算结果的前提下，编译器有可能会改变顺序代码的指令执行顺序，特别是在一些可以优化的场景。

例如，在以下案例中，编译器为了尽可能地减少寄存器的读取、存储次数，会充分复用寄存器的存储值。如果没有进行重排序优化，正常的执行顺序是步骤 1\2\3，而在编译期间进行了重排序优化之后，执行的步骤有可能就变成了步骤 1/3/2 或者 2/1/3，这样就能减少一次寄存器的存取次数。

 复制代码

```
1 int x = 1; // 步骤 1: 加载 x 变量的内存地址到寄存器中, 加载  
2 boolean flag = true; // 步骤 2 加载 flag 变量的内存地址到  
3 int y = x + 1; // 步骤 3 重新加载 a 变量的内存地址到寄存器中
```



在 JVM 中，重排序是十分重要的一环，特别是在并发编程中。可 JVM 要是能对它们进行任意排序的话，也可能会给并发编程带来一系列的问题，其中就包括了一致性的问题。

Happens-before 规则

为了解决这个问题，Java 提出了 Happens-before 规则来规范线程的执行顺序：

程序次序规则：在单线程中，代码的执行是有序的，虽然可能会存在运行指令的重排序，但最终执行的结果和顺序执行的结果是一致的；

锁定规则：一个锁处于被一个线程锁定占用状态，那么只有当这个线程释放锁之后，其它线程才能再次获取锁操作；

volatile 变量规则：如果一个线程正在写 volatile 变量，其它线程读取该变量会发生在写入之后；

线程启动规则：Thread 对象的 start() 方法先行发生于此线程的其它每一个动作；

线程终结规则：线程中的所有操作都先行发生于对此线程的终止检测；

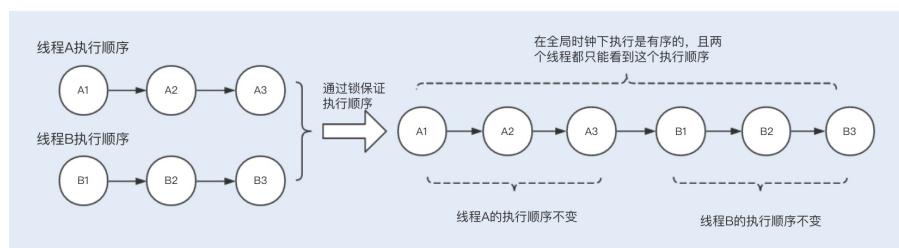
对象终结规则：一个对象的初始化完成先行发生于它的 finalize() 方法的开始；

传递性：如果操作 A happens-before 操作 B，操作 B happens-before 操作 C，那么操作 A happens-before 操作 C；

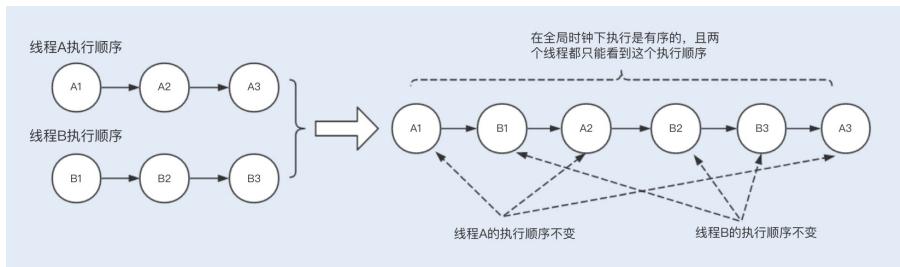
线程中断规则：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生。

结合这些规则，我们可以将一致性分为以下几个级别：

严格一致性（强一致性）：所有的读写操作都按照全局时钟下的顺序执行，且任何时刻线程读取到的缓存数据都是一样的，Hashtable 就是严格一致性；



顺序一致性：多个线程的整体执行可能是无序的，但对于单个线程而言执行是有序的，要保证任何一次读都能读到最近一次写入的数据，`volatile` 可以阻止指令重排序，所以修饰的变量的程序属于顺序一致性；



弱一致性：不能保证任何一次读都能读到最近一次写入的数据，但能保证最终可以读到写入的数据，单个写锁 + 无锁读，就是弱一致性的一种实现。

20 | 磨刀不误砍柴工：欲知JVM调优先了解JVM内存模型



从今天开始，我将和你一起探讨 Java 虚拟机（JVM）的性能调优。JVM 算是面试中的高频问题了，通常情况下总会有人问到：请你讲解下 JVM 的内存模型，JVM 的性能调优做过吗？

为什么 JVM 在 Java 中如此重要？

首先你应该知道，运行一个 Java 应用程序，我们必须要先安装 JDK 或者 JRE 包。这是因为 Java 应用在编译后会变成字节码，然后通过字节码运行在 JVM 中，而 JVM 是 JRE 的核心组成部分。

JVM 不仅承担了 Java 字节码的分析 (JIT compiler) 和执行 (Runtime) , 同时也内置了自动内存分配管理机制。这个机制可以大大降低手动分配回收机制可能带来的内存泄露和内存溢出风险 , 使 Java 开发人员不需要关注每个对象的内存分配以及回收 , 从而更专注于业务本身。

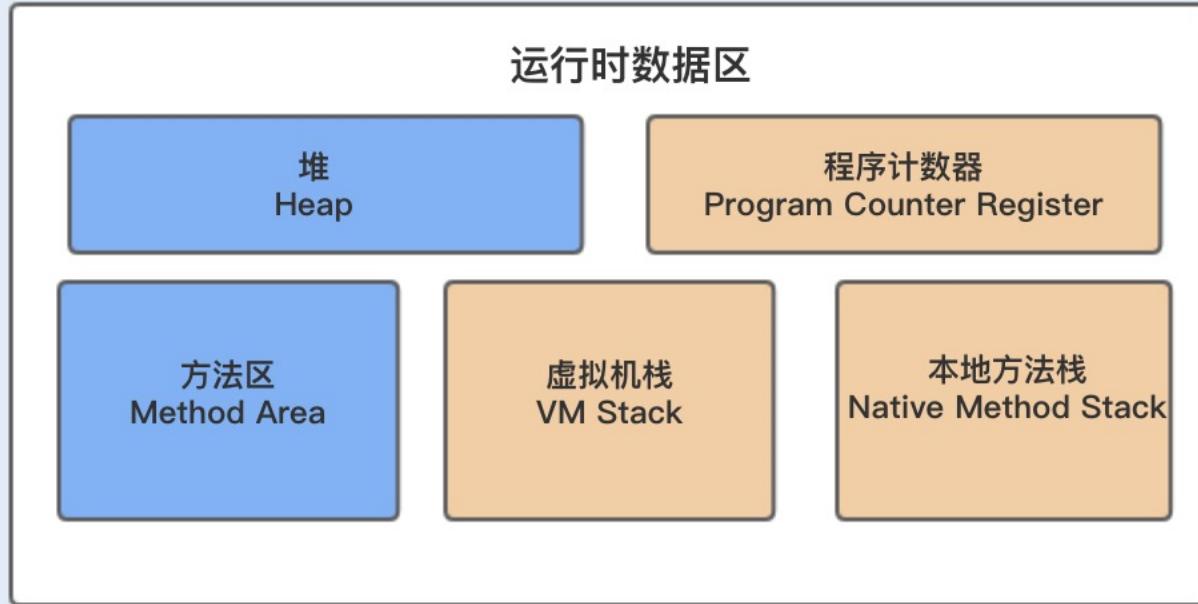
从了解内存模型开始

JVM 自动内存分配管理机制的好处很多 , 但实则是把双刃剑。这个机制在提升 Java 开发效率的同时 , 也容易使 Java 开发人员过度依赖于自动化 , 弱化对内存的管理能力 , 这样系统就很容易发生 JVM 的堆内存异常 , 垃圾回收 (GC) 的方式不合适以及 GC 次数过于频繁等问题 , 这些都将直接影响到应用服务的性能。

因此 , 要进行 JVM 层面的调优 , 就需要深入了解 JVM 内存分配和回收原理 , 这样在遇到问题时 , 我们才能通过日志分析快速地定位问题 ; 也能在系统遇到性能瓶颈时 , 通过分析 JVM 调优来优化系统性能。这也是整个模块四的重点内容 , 今天我们就从 JVM 的内存模型学起 , 为后续的学习打下一个坚实的基础。

JVM 内存模型的具体设计

我们先通过一张 JVM 内存模型图 , 来熟悉下其具体设计。在 Java 中 , JVM 内存模型主要分为堆、程序计数器、方法区、虚拟机栈和本地方法栈。



线程共享

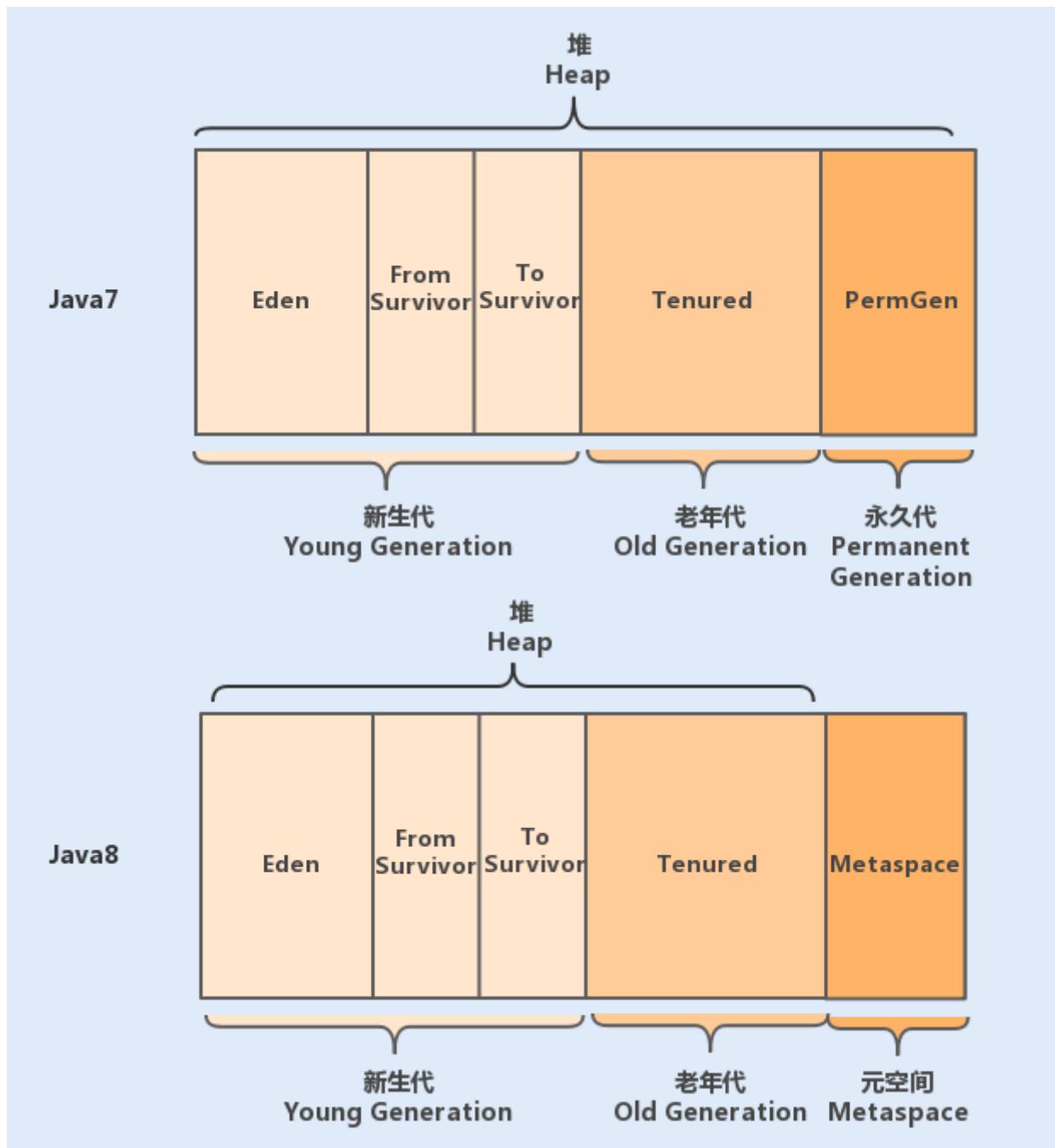
线程隔离

JVM 的 5 个分区具体是怎么实现的呢？我们一一分析。

1. 堆 (Heap)

堆是 JVM 内存中最大的一块内存空间，该内存被所有线程共享，几乎所有对象和数组都被分配到了堆内存中。堆被划分为新生代和老年代，新生代又被进一步划分为 Eden 和 Survivor 区，最后 Survivor 由 From Survivor 和 To Survivor 组成。

在 Java6 版本中，永久代在非堆内存区；到了 Java7 版本，永久代的静态变量和运行时常量池被合并到了堆中；而到了 Java8，永久代被元空间取代了。结构如下图所示：



2. 程序计数器 (Program Counter Register)

程序计数器是一块很小的内存空间，主要用来记录各个线程执行的字节码的地址，例如，分支、循环、跳转、异常、线程恢复等都依赖于计数器。

由于 Java 是多线程语言，当执行的线程数量超过 CPU 数量时，线程之间会根据时间片轮询争夺 CPU 资源。如果一个线程的时间片用完了，或者是其它原因导致这个线程的 CPU 资源被提前抢夺，那么这个退出的线程就需要单独的一个程序计数器，来记录下一条运行的指令。

3. 方法区 (Method Area)

很多开发者都习惯将方法区称为“永久代”，其实这两者并不是等价的。

HotSpot 虚拟机使用永久代来实现方法区，但在其它虚拟机中，例如，Oracle 的 JRockit、IBM 的 J9 就不存在永久代一说。因此，方法区只是 JVM 中规范的一部分，可以说，在 HotSpot 虚拟机中，设计人员使用了永久代来实现了 JVM 规范的方法区。

方法区主要是用来存放已被虚拟机加载的类相关信息，包括类信息、运行时常量池、字符串常量池。类信息又包括了类的版本、字段、方法、接口和父类等信息。

JVM 在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。在加载类的时候，JVM 会先加载 class 文件，而在 class 文件中除了有类的版本、字段、方法和接口等描述信息外，还有一项信息是常量池 (Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。

字面量包括字符串 (String a= “b”)、基本类型的常量 (final 修饰的变量)，符号引用则包括类和方法的全限定名 (例如 String 这个类，它的全限定名就是 Java/lang/String)、字段的名称和描述符以及方法的名称和描述符。

而当类加载到内存中后，JVM 就会将 class 文件常量池中的内容存放到运行时的常量池中；在解析阶段，JVM 会把符号引用替换为直接引用 (对象的索引值)。

例如，类中的一个字符串常量在 class 文件中时，存放在 class 文件常量池中的；在 JVM 加载完类之后，JVM 会将这个字符串常量放到运行时常量池中，并在解析阶段，指定该字符串对象的索引值。运行时常量池是全局共享的，多个类共用一个运行时常量池，class 文件中常量池多个相同的字符串在运行时常量池只会存在一份。

方法区与堆空间类似，也是一个共享内存区，所以方法区是线程共享的。假如两个线程都试图访问方法区中的同一个类信息，而这个类还没有装入 JVM，那么此时就只允许一个线程去加载它，另一个线程必须等待。

在 HotSpot 虚拟机、Java7 版本中已经将永久代的静态变量和运行时常量池转移到了堆中，其余部分则存储在 JVM 的非堆内存中，而 Java8 版本已经将方法区中实现的永久代去掉了，并用元空间 (class metadata) 代替了之前的永久代，并且元空间的存储位置是本地内存。之前永久代的类的元数据存储在了元空间，永久代的静态变量 (class static

variables) 以及运行时常量池 (runtime constant pool) 则跟 Java7 一样，转移到了堆中。

那你可能又有疑问了，Java8 为什么使用元空间替代永久代，这样做有什么好处呢？

官方给出的解释是：

移除永久代是为了融合 HotSpot JVM 与 JRockit VM 而做出的努力，因为 JRockit 没有永久代，所以不需要配置永久代。

永久代内存经常不够用或发生内存溢出，爆出异常 `java.lang.OutOfMemoryError: PermGen`。这是因为在 JDK1.7 版本中，指定的 PermGen 区大小为 8M，由于 PermGen 中类的元数据信息在每次 FullGC 的时候都可能被收集，回收率都偏低，成绩很难令人满意；还有，为 PermGen 分配多大的空间很难确定，PermSize 的大小依赖于很多因素，比如，JVM 加载的 class 总数、常量池的大小和方法的大小等。

4. 虚拟机栈 (VM stack)

Java 虚拟机栈是线程私有的内存空间，它和 Java 线程一起创建。当创建一个线程时，会在虚拟机栈中申请一个线程栈，用来保存方法的局部变量、操作数栈、动态链接方法和返回地址等信息，并参与方法的调用和返回。每一个方法的调用都伴随着栈帧的入栈操作，方法的返回则是栈帧的出栈操作。

5. 本地方法栈 (Native Method Stack)

本地方法栈跟 Java 虚拟机栈的功能类似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈则用于管理本地方法的调用。但本地方法并不是用 Java 实现的，而是由 C 语言实现的。

JVM 的运行原理

看到这里，相信你对 JVM 内存模型已经有个充分的了解了。接下来，我们通过一个案例来了解下代码和对象是如何分配存储的，Java 代码又是如何在 JVM 中运行的。

 复制代码

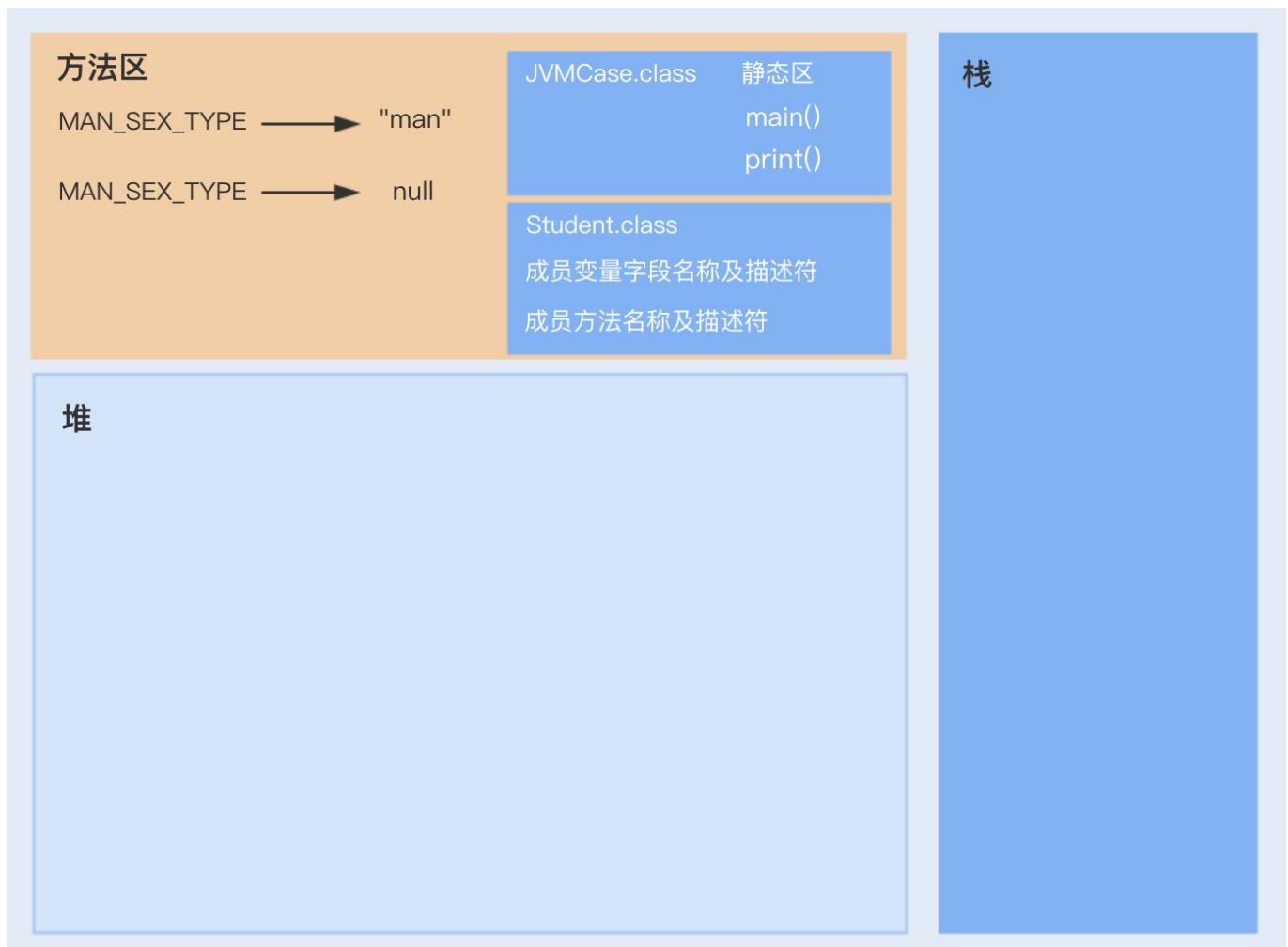
```
1 public class JVMCase {  
2  
3     // 常量
```

```
4     public final static String MAN_SEX_TYPE = "man";
5
6     // 静态变量
7     public static String WOMAN_SEX_TYPE = "woman";
8
9     public static void main(String[] args) {
10
11         Student stu = new Student();
12         stu.setName("nick");
13         stu.setSexType(MAN_SEX_TYPE);
14         stu.setAge(20);
15
16         JVMCase jvmcase = new JVMCase();
17
18         // 调用静态方法
19         print(stu);
20         // 调用非静态方法
21         jvmcase.sayHello(stu);
22     }
23
24
25     // 常规静态方法
26     public static void print(Student stu) {
27         System.out.println("name: " + stu.getName() + "; sex:" + stu.getSexType
28     }
29
30
31     // 非静态方法
32     public void sayHello(Student stu) {
33         System.out.println(stu.getName() + "say: hello");
34     }
35 }
36
37 class Student{
38     String name;
39     String sexType;
40     int age;
41
42     public String getName() {
43         return name;
44     }
45     public void setName(String name) {
46         this.name = name;
47     }
48
49     public String getSexType() {
50         return sexType;
51     }
52     public void setSexType(String sexType) {
53         this.sexType = sexType;
54     }
55     public int getAge() {
```

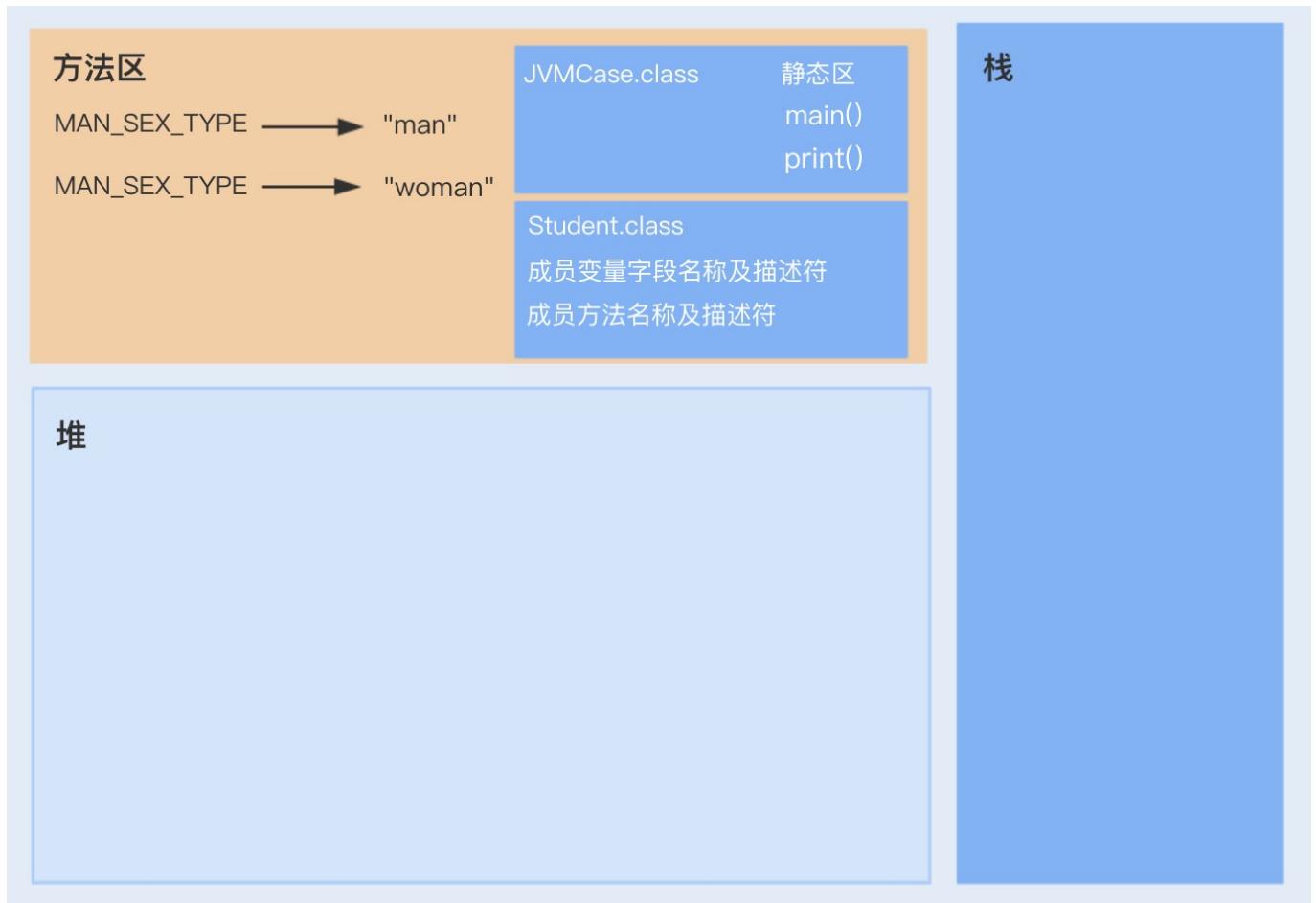
```
56             return age;
57     }
58     public void setAge(int age) {
59         this.age = age;
60     }
61 }
```

当我们通过 Java 运行以上代码时，JVM 的整个处理过程如下：

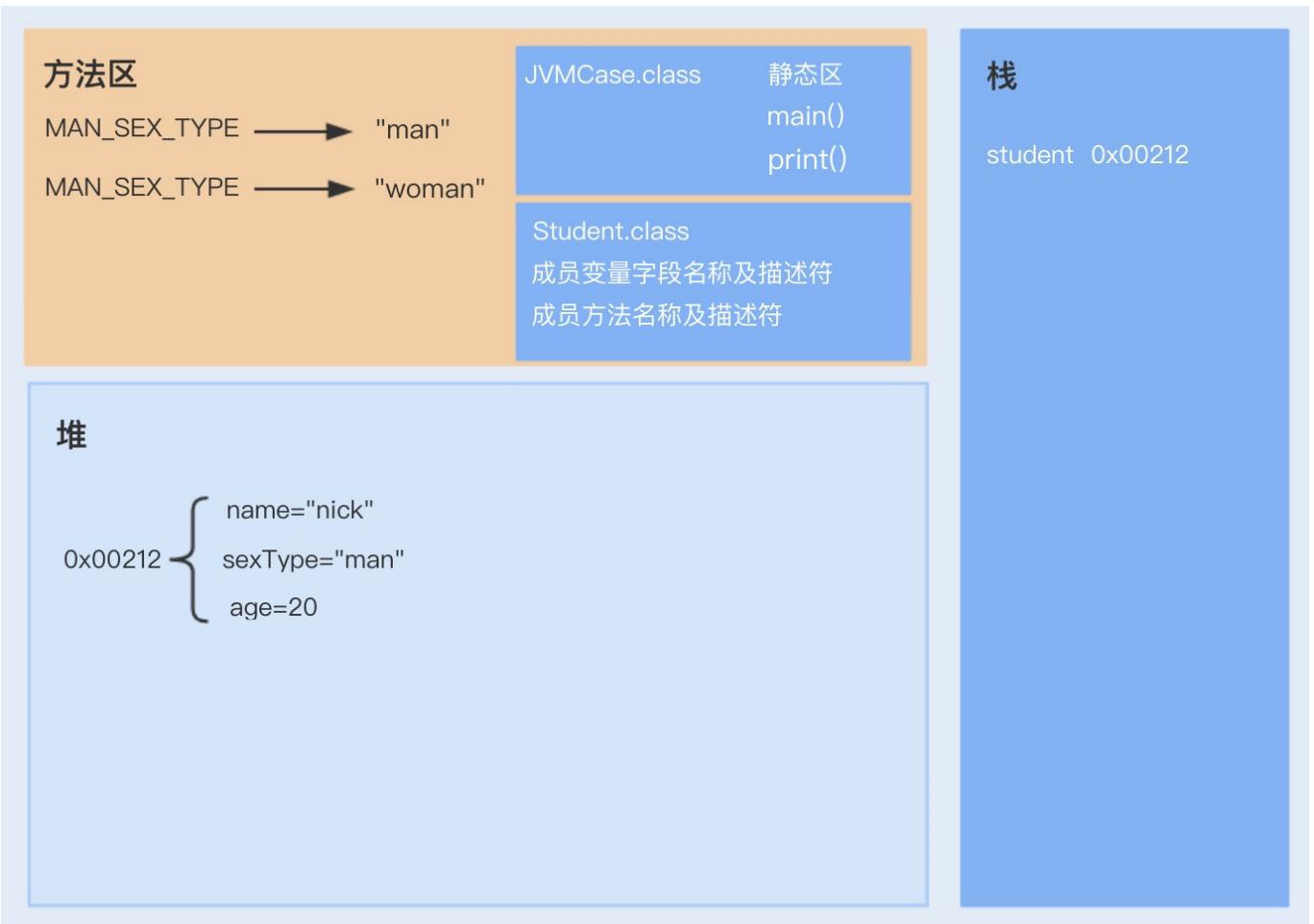
- 1.JVM 向操作系统申请内存，JVM 第一步就是通过配置参数或者默认配置参数向操作系统申请内存空间，根据内存大小找到具体的内存分配表，然后把内存段的起始地址和终止地址分配给 JVM，接下来 JVM 就进行内部分配。
- 2.JVM 获得内存空间后，会根据配置参数分配堆、栈以及方法区的内存大小。
- 3.class 文件加载、验证、准备以及解析，其中准备阶段会为类的静态变量分配内存，初始化为系统的初始值（这部分我在第 21 讲还会详细介绍）。



4. 完成上一个步骤后，将会进行最后一个初始化阶段。在这个阶段中，JVM 首先会执行构造器 <clinit> 方法，编译器会在.java 文件被编译成.class 文件时，收集所有类的初始化代码，包括静态变量赋值语句、静态代码块、静态方法，收集在一起成为 <clinit>() 方法。



5. 执行方法。启动 main 线程，执行 main 方法，开始执行第一行代码。此时堆内存中会创建一个 student 对象，对象引用 student 就存放在栈中。



6. 此时再次创建一个 `JVMCase` 对象，调用 `sayHello` 非静态方法，`sayHello` 方法属于对象 `JVMCase`，此时 `sayHello` 方法入栈，并通过栈中的 `student` 引用调用堆中的 `Student` 对象；之后，调用静态方法 `print`，`print` 静态方法属于 `JVMCase` 类，是从静态方法中获取，之后放入到栈中，也是通过 `student` 引用调用堆中的 `student` 对象。



了解完实际代码在 JVM 中分配的内存空间以及运行原理，相信你会更加清楚内存模型中各个区域的职责分工。

总结

这讲我们主要深入学习了最基础的内存模型设计，了解其各个分区的作用及实现原理。

如今，JVM 在很大程度上减轻了 Java 开发人员投入到对象生命周期的管理精力。在使用对象的时候，JVM 会自动分配内存给对象，在不使用的时候，垃圾回收器会自动回收对象，释放占用的内存。

21 | 深入JVM即时编译器JIT，优化Java编译



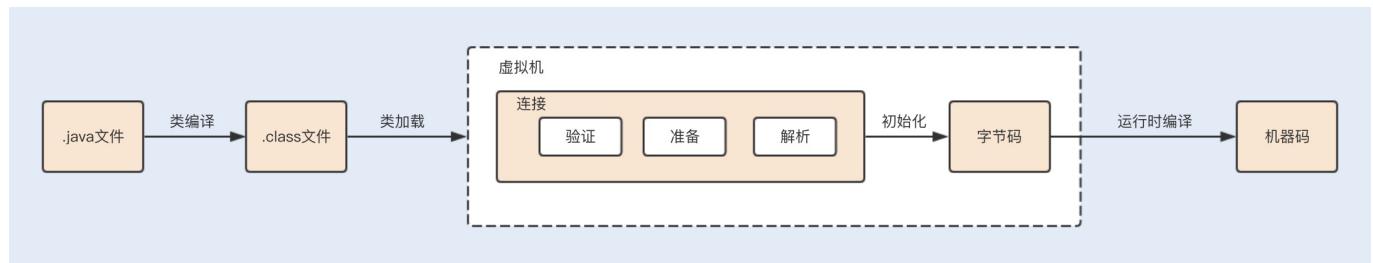
说到编译，我猜你一定会想到 .java 文件被编译成 .class 文件的过程，这个编译我们一般称为前端编译。Java 的编译和运行过程非常复杂，除了前端编译，还有运行时编译。由于机器无法直接运行 Java 生成的字节码，所以在运行时，JIT 或解释器会将字节码转换成机器码，这个过程就叫运行时编译。

类文件在运行时被进一步编译，它们可以变成高度优化的机器代码，由于 C/C++ 编译器的所有优化都是在编译期间完成的，运行期间的性能监控仅作为基础的优化措施则无法进行，例如，调用频率预测、分支频率预测、裁剪未被选择的分支等，而 Java 在运行时的再次编译，就可以进行基础的优化措施。因此，JIT 编译器可以说是 JVM 中运行时编译最重要的部分之一。

然而许多 Java 开发人员对 JIT 编译器的了解并不多，不深挖其工作原理，也不深究如何检测应用程序的即时编译情况，线上发生问题后很难做到从容应对。今天我们就来学习运行时编译如何实现对 Java 代码的优化。

类编译加载执行过程

在这之前，我们先了解下 Java 从编译到运行的整个过程，为后面的学习打下基础。请看下图：



类编译

在编写好代码之后，我们需要将 .java 文件编译成 .class 文件，才能在虚拟机上正常运行代码。文件的编译通常是由 JDK 中自带的 Javac 工具完成，一个简单的 .java 文件，我们可以通过 javac 命令来生成 .class 文件。

下面我们通过 javap ([第 12 讲](#) 讲过如何使用 javap 反编译命令行) 反编译来看看一个 class 文件结构中主要包含了哪些信息：

魔数 →

```

Classfile /C:/Users/admin/eclipse-workspace/demo/coroutine/src/main/java/com/demo/coroutine/TestClass.class
Last modified 2019-7-1; size 842 bytes
MD5 checksum 72d5dd9cd7400e37546b0e25a28809f7
Compiled from "TestClass.java"
public class com.demo.coroutine.TestClass
  minor version: 0
  major version: 52
  flags: ACC_PUELIC, ACC_SUPER
Constant pool:
#1 = Methodref      #15.#31      // java/lang/Object."<init>":()V
#2 = String          #32        // test3
#3 = Fieldref        #6.#33      // com/demo/coroutine/TestClass.c:Ljava/lang/String;
#4 = Class           #34        // java/lang/StringBuilder
#5 = Methodref        #4.#31      // java/lang/StringBuilder."<init>":()V
#6 = Class           #35        // com/demo/coroutine/TestClass
#7 = String          #36        // test1
#8 = Methodref        #4.#37      // java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
#9 = Fieldref        #6.#38      // com/demo/coroutine/TestClass.b:Ljava/lang/String;
#10 = Methodref       #4.#39      // java/lang/StringBuilder.toString():Ljava/lang/String;
#11 = Methodref       #6.#40      // com/demo/coroutine/TestClass.splice:(Ljava/lang/String);
#12 = Fieldref        #41.#42      // java/lang/System.out:Ljava/io/PrintStream;
#13 = Methodref       #43.#44      // java/io/PrintStream.print:(Ljava/lang/String;)V
#14 = String          #45        // test2
#15 = Class           #46        // java/lang/Object
#16 = Utf8            a
#17 = Utf8            Ljava/lang/String;
#18 = Utf8            ConstantValue
#19 = Utf8            b
#20 = Utf8            c
#21 = Utf8            <init>
#22 = Utf8            ()V
#23 = Utf8            Code
#24 = Utf8            LineNumberTable
#25 = Utf8            splice
#26 = Utf8            ()Ljava/lang/String;
#27 = Utf8            print
#28 = Utf8            <clinit>
#29 = Utf8            SourceFile
#30 = Utf8            TestClass.java
#31 = NameAndType    #21:#22      // "<init>":()V
#32 = Utf8            test3
#33 = NameAndType    #20:#17      // c:Ljava/lang/String;
#34 = Utf8            java/lang/StringBuilder
#35 = Utf8            com/demo/coroutine/TestClass
#36 = Utf8            test1
#37 = NameAndType    #47:#48      // append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
#38 = NameAndType    #19:#17      // b:Ljava/lang/String;
#39 = NameAndType    #49:#26      // toString():Ljava/lang/String;
#40 = NameAndType    #25:#26      // splice:(Ljava/lang/String;
#41 = Class          #50        // java/lang/System
#42 = NameAndType    #51:#52      // out:Ljava/io/PrintStream;
#43 = Class          #53        // java/io/PrintStream
#44 = NameAndType    #27:#54      // print:(Ljava/lang/String;)V
#45 = Utf8            test2
#46 = Utf8            java/lang/Object
#47 = Utf8            append
#48 = Utf8            (Ljava/lang/String;)Ljava/lang/StringBuilder;
#49 = Utf8            toString
#50 = Utf8            java/lang/System
#51 = Utf8            out
#52 = Utf8            Ljava/io/PrintStream;
#53 = Utf8            java/io/PrintStream
#54 = Utf8            (Ljava/lang/String;)V
{
  public static final java.lang.String a;
    descriptor: Ljava/lang/String;
    flags: ACC_PUELIC, ACC_STATIC, ACC_FINAL
  ConstantValue: String test1

  public static java.lang.String b;
    descriptor: Ljava/lang/String;
    flags: ACC_PUELIC, ACC_STATIC

  public com.demo.coroutine.TestClass();
    descriptor: ()V
    flags: ACC_PUELIC
    Code:
      stack=2, locals=1, args_size=1
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: aload_0
        5: ldc           #2                  // String test3
        7: putfield       #3                  // Field c:Ljava/lang/String;
        10: return

      LineNumberTable:
        line 3: 0
        line 6: 4

  public java.lang.String splice();
    descriptor: ()Ljava/lang/String;
    flags: ACC_PUELIC
    Code:
      stack=2, locals=1, args_size=1
        0: new             #4                  // class java/lang/StringBuilder
        3: dup
        4: invokespecial #5                  // Method java/lang/StringBuilder."<init>":()V
        7: ldc           #7                  // String test1
        9: invokevirtual #8                  // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
        12: getstatic      #9                  // Field b:Ljava/lang/String;
        15: invokevirtual #8                  // Method java/lang/StringBuilder.append:

```

```

10: invokevirtual #8           // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   18: invokevirtual #10          // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   21: areturn
LineNumberTable:
  line 9: 0

public void print();
  descriptor: ()V
  flags: ACC_PUBLIC
Code:
  stack=3, locals=2, args_size=1
    0: aload_0
    1: invokevirtual #11          // Method splice:()Ljava/lang/String;
    4: astore_1
    5: getstatic     #12          // Field java/lang/System.out:Ljava/io/PrintStream;
    8: new          #4           // class java/lang/StringBuilder
   11: dup
   12: invokespecial #5          // Method java/lang/StringBuilder."<init>":()V
   15: aload_0
   16: getfield      #3           // Field c:Ljava/lang/String;
   19: invokevirtual #8          // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   22: aload_1
   23: invokevirtual #8          // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   26: invokevirtual #10          // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   29: invokevirtual #13          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
   32: return
LineNumberTable:
  line 13: 0
  line 14: 5
  line 15: 32
}

static {};
  descriptor: ()V
  flags: ACC_STATIC
Code:
  stack=1, locals=0, args_size=0
    0: ldc           #14          // String test2
    2: putstatic     #9           // Field b:Ljava/lang/String;
    5: return
LineNumberTable:
  line 5: 0
}
SourceFile: "TestClass.java"

```

方法表集合

} 属性表集合

看似一个简单的命令执行，前期编译的过程其实是非常复杂的，包括词法分析、填充符号表、注解处理、语义分析以及生成 class 文件，这个过程我们不用过多关注。只要从上图中知道，**编译后的字节码文件主要包括常量池和方法表集合这两部分**就可以了。

常量池主要记录的是类文件中出现的字面量以及符号引用。字面常量包括字符串常量（例如 String str= “abc” ，其中“abc”就是常量），声明为 final 的属性以及一些基本类型（例如，范围在 -127-128 之间的整型）的属性。符号引用包括类和接口的全限定名、类引用、方法引用以及成员变量引用（例如 String str= “abc” ，其中 str 就是成员变量引用）等。

方法表集合中主要包含一些方法的字节码、方法访问权限（ public、protect、private 等）、方法名索引（与常量池中的方法引用对应）、描述符索引、JVM 执行指令以及属性集合等。

类加载

当一个类被创建实例或者被其它对象引用时，虚拟机在没有加载过该类的情况下，会通过类加载器将字节码文件加载到内存中。

不同的实现类由不同的类加载器加载，JDK 中的本地方法类一般由根加载器（Bootstrap loader）加载进来，JDK 中内部实现的扩展类一般由扩展加载器（ExtClassLoader）实现加载，而程序中的类文件则由系统加载器（AppClassLoader）实现加载。

在类加载后，class 类文件中的常量池信息以及其它数据会被保存到 JVM 内存的方法区中。

类连接

类在加载进来之后，会进行连接、初始化，最后才会被使用。在连接过程中，又包括验证、准备和解析三个部分。

验证：验证类符合 Java 规范和 JVM 规范，在保证符合规范的前提下，避免危害虚拟机安全。

准备：为类的静态变量分配内存，初始化为系统的初始值。对于 final static 修饰的变量，直接赋值为用户的定义值。例如，private final static int value=123，会在准备阶段分配内存，并初始化值为 123，而如果是 private static int value=123，这个阶段 value 的值仍然为 0。

解析：将符号引用转为直接引用的过程。我们知道，在编译时，Java 类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。类结构文件的常量池中存储了符号引用，包括类和接口的全限定名、类引用、方法引用以及成员变量引用等。如果要使用这些类和方法，就需要把它们转化为 JVM 可以直接获取的内存地址或指针，即直接引用。

类初始化

类初始化阶段是类加载过程的最后阶段，在这个阶段中，JVM 首先将执行构造器 <clinit> 方法，编译器会在将 .java 文件编译成 .class 文件时，收集所有类初始化代码，包括静态变量赋值语句、静态代码块、静态方法，收集在一起成为 <clinit>() 方法。

初始化类的静态变量和静态代码块为用户自定义的值，初始化的顺序和 Java 源码从上到下的顺序一致。例如：

 复制代码

```
1 private static int i=1;
2 static{
```

```
3     i=0;
4 }
5 public static void main(String [] args){
6     System.out.println(i);
7 }
8
```

 复制代码

```
1 0
```

此时运行结果为：

```
1 static{
2     i=0;
3 }
4 private static int i=1;
5 public static void main(String [] args){
6     System.out.println(i);
7 }
```

 复制代码

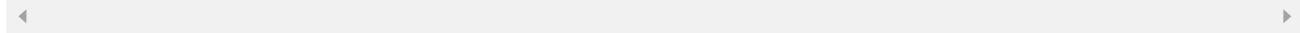
```
1 1
```

子类初始化时会首先调用父类的 <clinit>() 方法，再执行子类的 <clinit>() 方法，运行以下代码：

```
1 public class Parent{
```

 复制代码

```
2 public static String parentStr= "parent static string";
3 static{
4     System.out.println("parent static fields");
5     System.out.println(parentStr);
6 }
7 public Parent(){
8     System.out.println("parent instance initialization");
9 }
10 }
11
12 public class Sub extends Parent{
13     public static String subStr= "sub static string";
14     static{
15         System.out.println("sub static fields");
16         System.out.println(subStr);
17     }
18
19     public Sub(){
20         System.out.println("sub instance initialization");
21     }
22
23     public static void main(String[] args){
24         System.out.println("sub main");
25         new Sub();
26     }
27 }
```



运行结果：

复制代码

```
1 parent static fields
2 parent static string
3 sub static fields
4 sub static string
5 sub main
6 parent instance initialization
7 sub instance initialization
```



JVM 会保证 <clinit>() 方法的线程安全，保证同一时间只有一个线程执行。

JVM 在初始化执行代码时，如果实例化一个新对象，会调用 <init> 方法对实例变量进行初始化，并执行对应的构造方法内的代码。

即时编译

初始化完成后，类在调用执行过程中，执行引擎会把字节码转为机器码，然后在操作系统中才能执行。**在字节码转换为机器码的过程中，虚拟机中还存在着一道编译，那就是即时编译。**

最初，虚拟机中的字节码是由解释器（Interpreter）完成编译的，当虚拟机发现某个方法或代码块的运行特别频繁的时候，就会把这些代码认定为“热点代码”。

为了提高热点代码的执行效率，在运行时，即时编译器（JIT）会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，然后保存到内存中。

即时编译器类型

在 HotSpot 虚拟机中，内置了两个 JIT，分别为 C1 编译器和 C2 编译器，这两个编译器的编译过程是不一样的。

C1 编译器是一个简单快速的编译器，主要的关注点在于局部性的优化，适用于执行时间较短或对启动性能有要求的程序，例如，GUI 应用对界面启动速度就有一定要求。

C2 编译器是为长期运行的服务器端应用程序做性能调优的编译器，适用于执行时间较长或对峰值性能有要求的程序。根据各自的适配性，这两种即时编译也被称为 Client Compiler 和 Server Compiler。

在 Java7 之前，需要根据程序的特性来选择对应的 JIT，虚拟机默认采用解释器和其中一个编译器配合工作。

Java7 引入了分层编译，这种方式综合了 C1 的启动性能优势和 C2 的峰值性能优势，我们也可以通过参数 “-client” “-server” 强制指定虚拟机的即时编译模式。**分层编译将 JVM 的执行状态分为了 5 个层次：**

第 0 层：程序解释执行，默认开启性能监控功能（Profiling），如果不开启，可触发第二层编译；

第 1 层：可称为 C1 编译，将字节码编译为本地代码，进行简单、可靠的优化，不开启 Profiling；

第 2 层：也称为 C1 编译，开启 Profiling，仅执行带方法调用次数和循环回边执行次数 profiling 的 C1 编译；

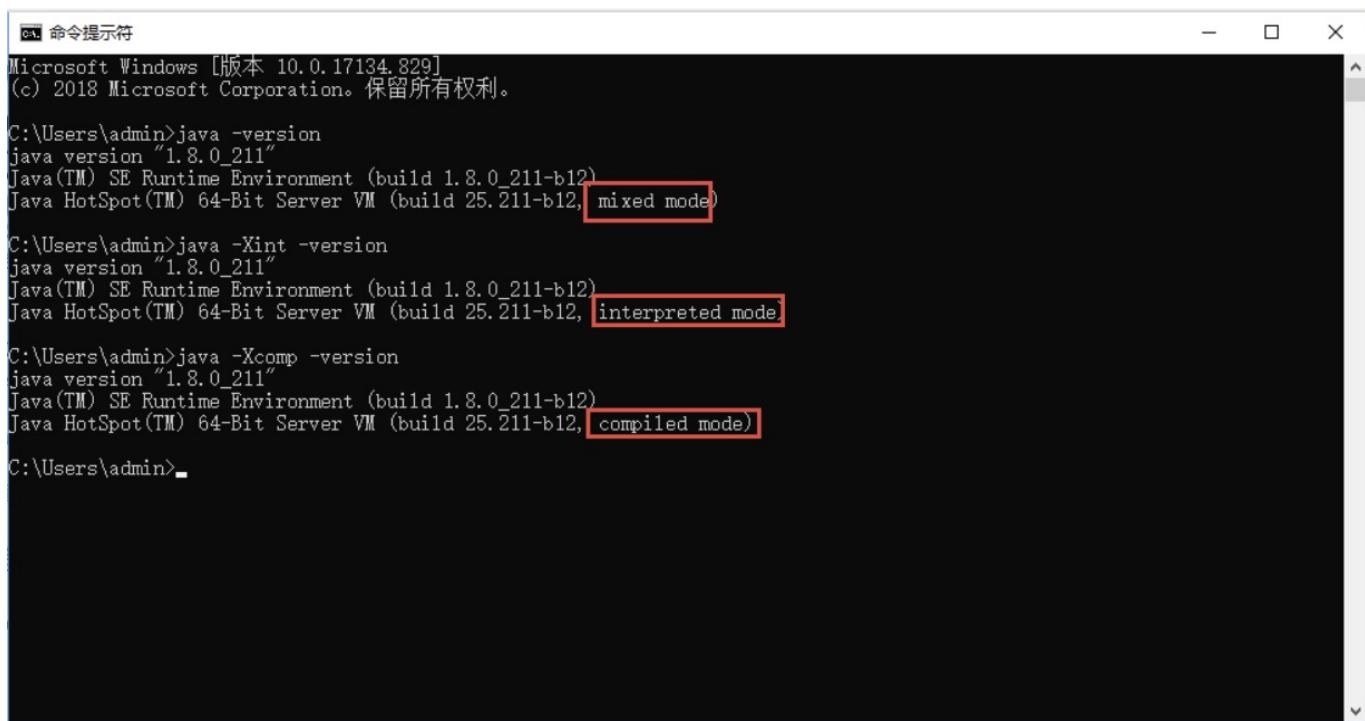
第 3 层：也称为 C1 编译，执行所有带 Profiling 的 C1 编译；

第 4 层：可称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

在 Java8 中，默认开启分层编译，-client 和 -server 的设置已经是无效的了。如果只想开启 C2，可以关闭分层编译（-XX:-TieredCompilation），如果只想用 C1，可以在打开分层编译的同时，使用参数：-XX:TieredStopAtLevel=1。

除了这种默认的混合编译模式，我们还可以使用 “-Xint” 参数强制虚拟机运行于只有解释器的编译模式下，这时 JIT 完全不介入工作；我们还可以使用参数 “-Xcomp” 强制虚拟机运行于只有 JIT 的编译模式下。

通过 java -version 命令行可以直接查看到当前系统使用的编译模式。如下图所示：



```
命令提示符
Microsoft Windows [版本 10.0.17134.829]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\admin>java -version
java version "1.8.0_211"
Java(TM) SE Runtime Environment (build 1.8.0_211-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, mixed mode)

C:\Users\admin>java -Xint -version
java version "1.8.0_211"
Java(TM) SE Runtime Environment (build 1.8.0_211-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, interpreted mode)

C:\Users\admin>java -Xcomp -version
java version "1.8.0_211"
Java(TM) SE Runtime Environment (build 1.8.0_211-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, compiled mode)

C:\Users\admin>
```

热点探测

在 HotSpot 虚拟机中的热点探测是 JIT 优化的条件，热点探测是基于计数器的热点探测，采用这种方法的虚拟机会为每个方法建立计数器统计方法的执行次数，如果执行次数超过一定的阈值就认为它是“热点方法”。

虚拟机为每个方法准备了两类计数器：方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发 JIT 编译。

方法调用计数器：用于统计方法被调用的次数，方法调用计数器的默认阈值在 C1 模式下是 1500 次，在 C2 模式下是 10000 次，可通过 -XX: CompileThreshold 来设定；而在分层编译的情况下，-XX: CompileThreshold 指定的阈值将失效，此时将会根据当前待编译的方法数以及编译线程数来动态调整。当方法计数器和回边计数器之和超过方法计数器阈值时，就会触发 JIT 编译器。

回边计数器：用于统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”（Back Edge），该值用于计算是否触发 C1 编译的阈值，在不开启分层编译的情况下，C1 默认为 13995，C2 默认为 10700，可通过 -XX: OnStackReplacePercentage=N 来设置；而在分层编译的情况下，-XX: OnStackReplacePercentage 指定的阈值同样会失效，此时将根据当前待编译的方法数以及编译线程数来动态调整。

建立回边计数器的主要目的是为了触发 OSR（On StackReplacement）编译，即栈上编译。在一些循环周期比较长的代码段中，当循环达到回边计数器阈值时，JVM 会认为这段是热点代码，JIT 编译器就会将这段代码编译成机器语言并缓存，在该循环时间段内，会直接将执行代码替换，执行缓存的机器语言。

编译优化技术

JIT 编译运用了一些经典的编译优化技术来实现代码的优化，即通过一些例行检查优化，可以智能地编译出运行时的最优性能代码。今天我们主要来学习以下两种优化手段：

1. 方法内联

调用一个方法通常要经历压栈和出栈。调用方法是将程序执行顺序转移到存储该方法的内存地址，将方法的内容执行完后，再返回到执行该方法前的位置。

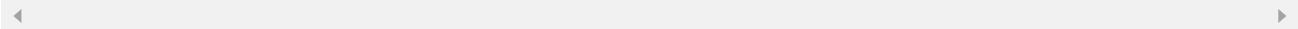
这种执行操作要求在执行前保护现场并记忆执行的地址，执行后要恢复现场，并按原来保存的地址继续执行。因此，方法调用会产生一定的时间和空间方面的开销。

那么对于那些方法体代码不是很大，又频繁调用的方法来说，这个时间和空间的消耗会很大。**方法内联的优化行为就是把目标方法的代码复制到发起调用的方法之中，避免发生真实的方法调用。**

例如以下方法：

 复制代码

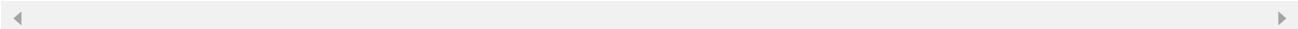
```
1 private int add1(int x1, int x2, int x3, int x4) {  
2     return add2(x1, x2) + add2(x3, x4);  
3 }  
4 private int add2(int x1, int x2) {  
5     return x1 + x2;  
6 }
```



最终会被优化为：

 复制代码

```
1 private int add1(int x1, int x2, int x3, int x4) {  
2     return x1 + x2+ x3 + x4;  
3 }
```



JVM 会自动识别热点方法，并对它们使用方法内联进行优化。我们可以通过 -XX:CompileThreshold 来设置热点方法的阈值。但要强调一点，热点方法不一定会被 JVM 做内联优化，如果这个方法体太大了，JVM 将不执行内联操作。而方法体的大小阈值，我们也可以通过参数设置来优化：

经常执行的方法，默认情况下，方法体大小小于 325 字节的都会进行内联，我们可以通过 -XX:MaxFreqInlineSize=N 来设置大小值；

不是经常执行的方法，默认情况下，方法大小小于 35 字节才会进行内联，我们也可以通过 -XX:MaxInlineSize=N 来重置大小值。

之后我们就可以通过配置 JVM 参数来查看到方法被内联的情况：

 复制代码

```
1 -XX:+PrintCompilation // 在控制台打印编译过程信息  
2 -XX:+UnlockDiagnosticVMOptions // 解锁对 JVM 进行诊断的选项参数。默认是关闭的，开启后支持一些  
3 -XX:+PrintInlining // 将内联方法打印出来
```

 复制代码

```
1     public static void main(String[] args) {  
2         for(int i=0; i<1000000; i++) {// 方法调用计数器的默认阈值在 C1 模式下是 150  
3             add1(1,2,3,4);  
4         }  
5     }
```

我们可以看到运行结果中，显示了方法内联的日志：

```
963  91 %    4      com.test.decorator.App::main @ 5 (23 bytes)  
                  @ 9  com.test.decorator.App::add1 (12 bytes)  inline (hot)  
                  @ 2  com.test.decorator.App::add2 (4 bytes)  inline (hot)  
                  @ 7  com.test.decorator.App::add2 (4 bytes)  inline (hot)  
963  89 %    3      com.test.decorator.App::main @ -2 (23 bytes)  made not entrant  
964  91 %    4      com.test.decorator.App::main @ -2 (23 bytes)  made not entrant
```

热点方法的优化可以有效提高系统性能，一般我们可以通过以下几种方式来提高方法内联：

通过设置 JVM 参数来减小热点阈值或增加方法体阈值，以便更多的方法可以进行内联，但这种方法意味着需要占用更多地内存；

在编程中，避免在一个方法中写大量代码，习惯使用小方法体；

尽量使用 final、private、static 关键字修饰方法，编码方法因为继承，会需要额外的类型检查。

2. 逃逸分析

逃逸分析 (Escape Analysis) 是判断一个对象是否被外部方法引用或外部线程访问的分析技术，编译器会根据逃逸分析的结果对代码进行优化。

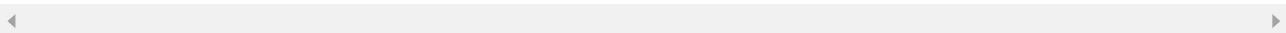
栈上分配

我们知道，在 Java 中默认创建一个对象是在堆中分配内存的，而当堆内存中的对象不再使用时，则需要通过垃圾回收机制回收，这个过程相对分配在栈中的对象的创建和销毁来说，更消耗时间和性能。这个时候，逃逸分析如果发现一个对象只在方法中使用，就会将对象分配在栈上。

以下是通过循环获取学生年龄的案例，方法中创建一个学生对象，我们现在通过案例来看看打开逃逸分析和关闭逃逸分析后，堆内存对象创建的数量对比。

 复制代码

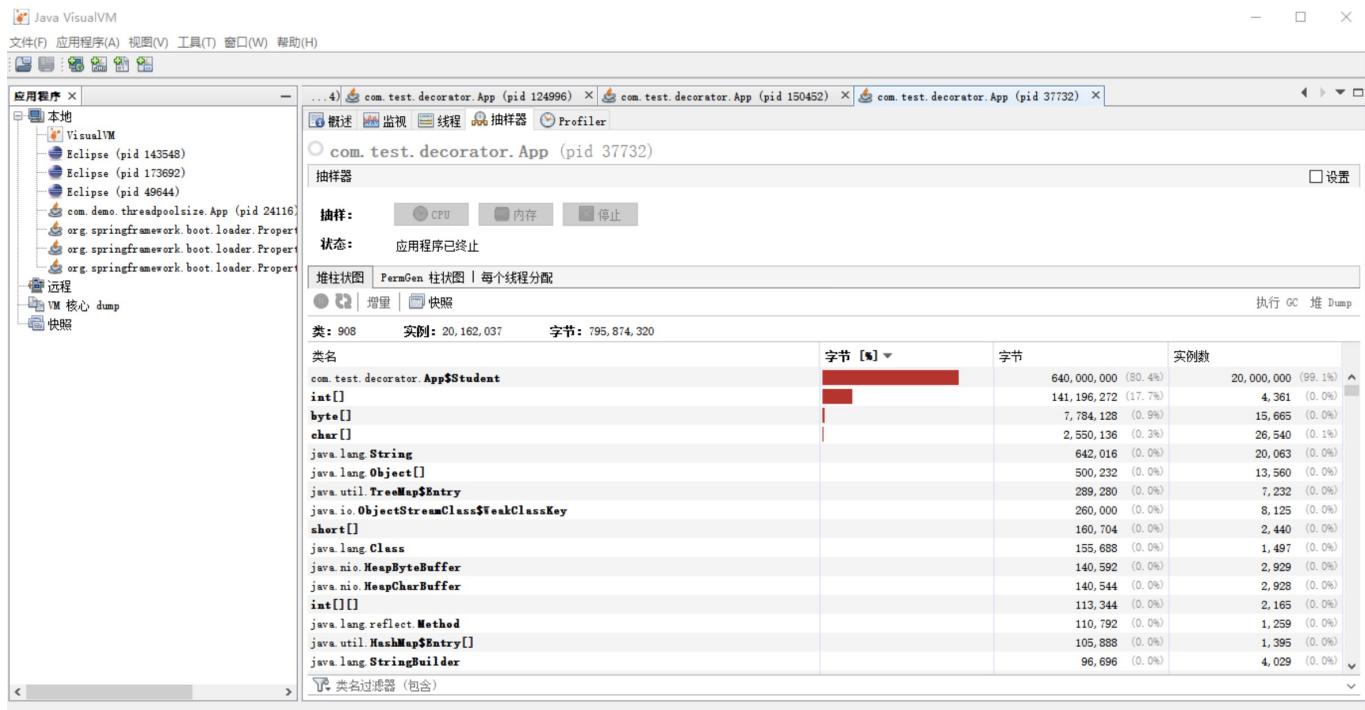
```
1 public static void main(String[] args) {
2     for (int i = 0; i < 200000 ; i++) {
3         getAge();
4     }
5 }
6
7 public static int getAge(){
8     Student person = new Student(" 小明 ",18,30);
9     return person.getAge();
10 }
11
12 static class Student {
13     private String name;
14     private int age;
15
16     public Student(String name, int age) {
17         this.name = name;
18         this.age = age;
19     }
20
21     public String getName() {
22         return name;
23     }
24
25     public void setName(String name) {
26         this.name = name;
27     }
28
29     public int getAge() {
30         return age;
31     }
32
33     public void setAge(int age) {
34         this.age = age;
35     }
36 }
```



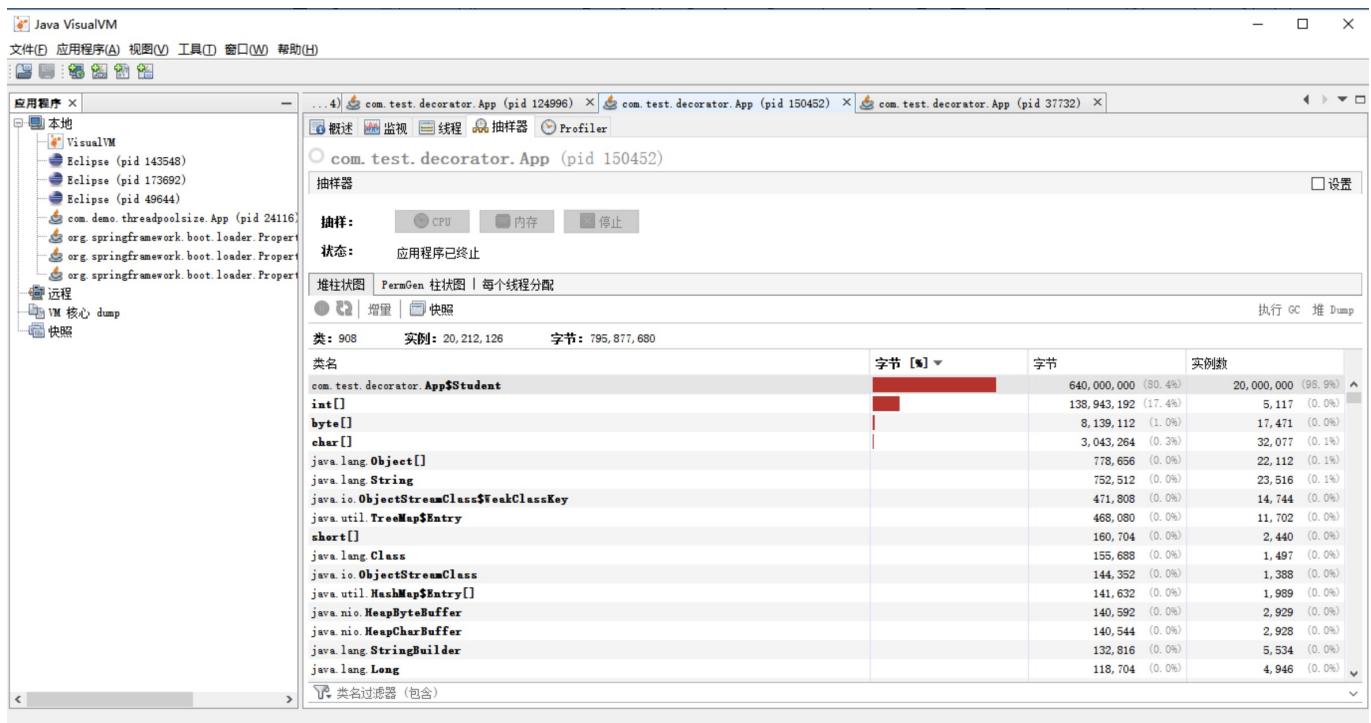
然后，我们分别设置 VM 参数：`Xmx1000m -Xms1000m -XX:-DoEscapeAnalysis -XX:+PrintGC` 以及 `-Xmx1000m -Xms1000m -XX:+DoEscapeAnalysis -XX:+PrintGC`，通过之前讲过的 VisualVM 工具，查看堆中创建的对象数量。

然而，运行结果却没有达到我们想要的优化效果，也许你怀疑是 JDK 版本的问题，然而我分别在 1.6~1.8 版本都测试过了，效果还是一样的：

(`-server -Xmx1000m -Xms1000m -XX:-DoEscapeAnalysis -XX:+PrintGC`)



(`-server -Xmx1000m -Xms1000m -XX:+DoEscapeAnalysis -XX:+PrintGC`)



这其实是因为由于 HotSpot 虚拟机目前的实现导致栈上分配实现比较复杂，可以说，在 HotSpot 中暂时没有实现这项优化。随着即时编译器的发展与逃逸分析技术的逐渐成熟，相信不久的将来 HotSpot 也会实现这项优化功能。

锁消除

在非线程安全的情况下，尽量不要使用线程安全容器，比如 StringBuffer。由于 StringBuffer 中的 append 方法被 Synchronized 关键字修饰，会使用到锁，从而导致性能下降。

但实际上，在以下代码测试中，StringBuffer 和 StringBuilder 的性能基本没什么区别。这是因为在局部方法中创建的对象只能被当前线程访问，无法被其它线程访问，这个变量的读写肯定不会有竞争，这个时候 JIT 编译会对这个对象的方法锁进行锁消除。

复制代码

```

1  public static String getString(String s1, String s2) {
2      StringBuffer sb = new StringBuffer();
3      sb.append(s1);
4      sb.append(s2);
5      return sb.toString();
6  }

```

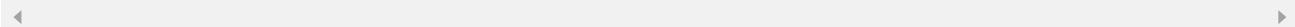
标量替换

逃逸分析证明一个对象不会被外部访问，如果这个对象可以被拆分的话，当程序真正执行的时候可能不创建这个对象，而直接创建它的成员变量来代替。将对象拆分后，可以分配对象的成员变量在栈或寄存器上，原本的对象就无需分配内存空间了。这种编译优化就叫做标量替换。

我们用以下代码验证：

 复制代码

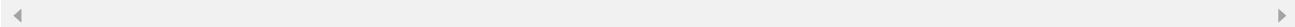
```
1 public void foo() {  
2     TestInfo info = new TestInfo();  
3     info.id = 1;  
4     info.count = 99;  
5     ...//to do something  
6 }
```



逃逸分析后，代码会被优化为：

 复制代码

```
1  
2 public void foo() {  
3     id = 1;  
4     count = 99;  
5     ...//to do something  
6 }
```



我们可以通过设置 JVM 参数来开关逃逸分析，还可以单独开关同步消除和标量替换，在 JDK1.8 中 JVM 是默认开启这些操作的。

 复制代码

```
1 -XX:+DoEscapeAnalysis 开启逃逸分析 (jdk1.8 默认开启, 其它版本未测试)  
2 -XX:-DoEscapeAnalysis 关闭逃逸分析  
3  
4 -XX:+EliminateLocks 开启锁消除 (jdk1.8 默认开启, 其它版本未测试)  
5 -XX:-EliminateLocks 关闭锁消除  
6
```

- 7 -XX:+EliminateAllocations 开启标量替换（jdk1.8 默认开启，其它版本未测试）
- 8 -XX:-EliminateAllocations 关闭就可以了

总结

今天我们主要了解了 JDK1.8 以及之前的类的编译和加载过程，Java 源程序是通过 Javac 编译器编译成 .class 文件，其中文件中包含的代码格式我们称之为 Java 字节码（bytecode）。

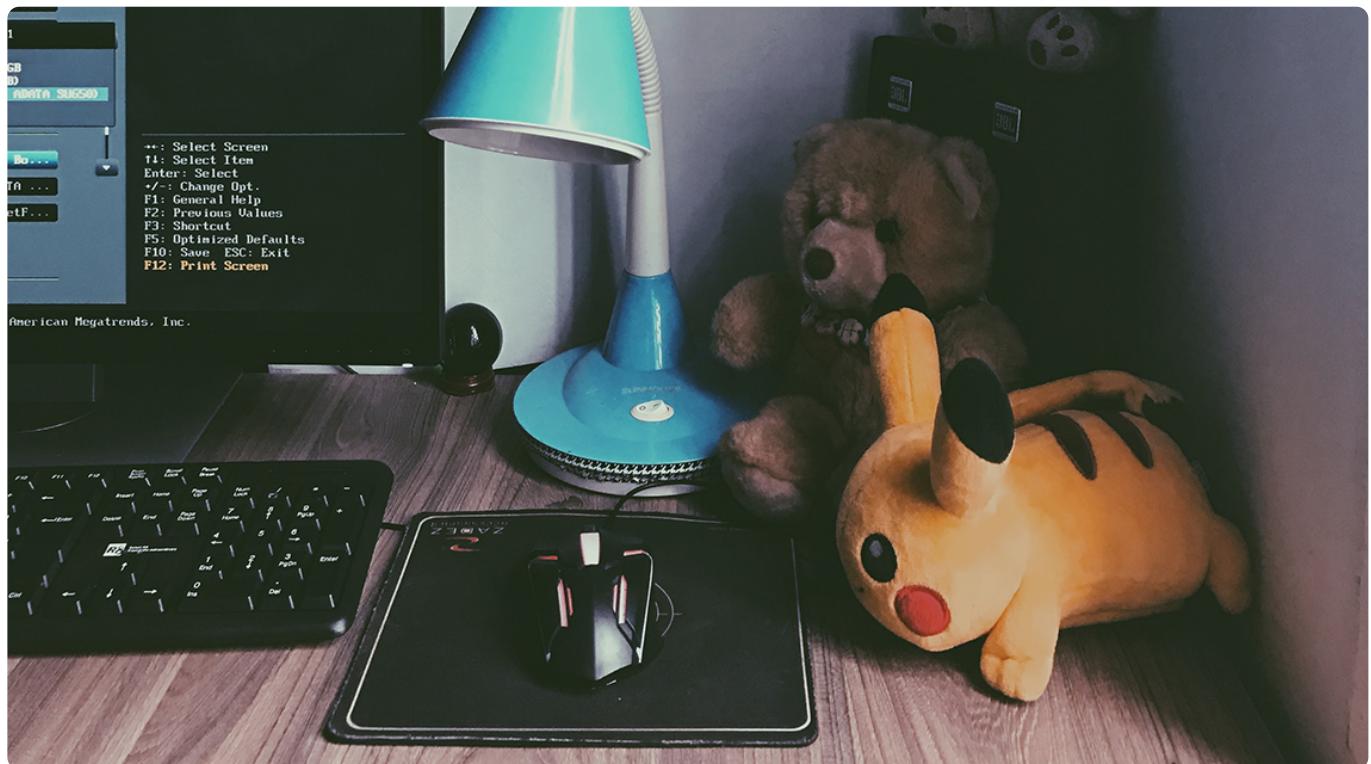
这种代码格式无法直接运行，但可以被不同平台 JVM 中的 Interpreter 解释执行。由于 Interpreter 的效率低下，JVM 中的 JIT 会在运行时有选择性地将运行次数较多的方法编译成二进制代码，直接运行在底层硬件上。

在 Java8 之前，HotSpot 集成了两个 JIT，用 C1 和 C2 来完成 JVM 中的即时编译。虽然 JIT 优化了代码，但收集监控信息会消耗运行时的性能，且编译过程会占用程序的运行时间。

到了 Java9，AOT 编译器被引入。和 JIT 不同，AOT 是在程序运行前进行的静态编译，这样就可以避免运行时的编译消耗和内存消耗，且 .class 文件通过 AOT 编译器是可以编译成 .so 的二进制文件的。

到了 Java10，一个新的 JIT 编译器 Graal 被引入。Graal 是一个以 Java 为主要编程语言、面向 Java bytecode 的编译器。与用 C++ 实现的 C1 和 C2 相比，它的模块化更加明显，也更容易维护。Graal 既可以作为动态编译器，在运行时编译热点方法；也可以作为静态编译器，实现 AOT 编译。

22 | 如何优化垃圾回收机制？



我们知道，在 Java 开发中，开发人员是无需过度关注对象的回收与释放的，JVM 的垃圾回收机制可以减轻不少工作量。但完全交由 JVM 回收对象，也会增加回收性能的不确定性。在一些特殊的业务场景下，不合适的垃圾回收算法以及策略，都有可能导致系统性能下降。

面对不同的业务场景，垃圾回收的调优策略也不一样。例如，在对内存要求苛刻的情况下，需要提高对象的回收效率；在 CPU 使用率高的情况下，需要降低高并发时垃圾回收的频率。可以说，垃圾回收的调优是一项必备技能。

这讲我们就把这项技能的学习进行拆分，看看回收（后面简称 GC）的算法有哪些，体现 GC 算法好坏的指标有哪些，又如何根据自己的业务场景对 GC 策略进行调优？

垃圾回收机制

掌握 GC 算法之前，我们需要先弄清楚 3 个问题。第一，回收发生在哪？第二，对象在什么时候可以被回收？第三，如何回收这些对象？

1. 回收发生在哪？

JVM 的内存区域中，程序计数器、虚拟机栈和本地方法栈这 3 个区域是线程私有的，随着线程的创建而创建，销毁而销毁；栈中的栈帧随着方法的进入和退出进行入栈和出栈操作，每个栈帧中分配多少内存基本是在类结构确定下来的时候就已知的，因此这三个区域的内存分配和回收都具有确定性。

那么垃圾回收的重点就是关注堆和方法区中的内存了，堆中的回收主要是对象的回收，方法区的回收主要是废弃常量和无用的类的回收。

2. 对象在什么时候可以被回收？

那 JVM 又是怎样判断一个对象是可以被回收的呢？**一般一个对象不再被引用，就代表该对象可以被回收。** 目前有以下两种算法可以判断该对象是否可以被回收。

引用计数算法：这种算法是通过一个对象的引用计数器来判断该对象是否被引用了。每当对象被引用，引用计数器就会加 1；每当引用失效，计数器就会减 1。当对象的引用计数器的值为 0 时，就说明该对象不再被引用，可以被回收了。这里强调一点，虽然引用计数算法的实现简单，判断效率也很高，但它存在着对象之间相互循环引用的问题。

可达性分析算法：GC Roots 是该算法的基础，GC Roots 是所有对象的根对象，在 JVM 加载时，会创建一些普通对象引用正常对象。这些对象作为正常对象的起始点，在垃圾回收时，会从这些 GC Roots 开始向下搜索，当一个对象到 GC Roots 没有任何引用链相连时，就证明此对象是不可用的。目前 HotSpot 虚拟机采用的就是这种算法。

以上两种算法都是通过引用来判断对象是否可以被回收。在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为了以下四种：

引用类型	功能特点
强引用 (Strong Reference)	被强引用关联的对象永远不会被垃圾收集器回收掉
软引用 (Soft Reference)	软引用关联的对象，只有当系统将要发生内存溢出时，才会去回收软引用引用的对象
弱引用 (Weak Reference)	只被弱引用关联的对象，只要发生垃圾收集事件，就会被回收
虚引用 (Phantom Reference)	被虚引用关联的对象的唯一作用是能在这个对象被回收器回收时收到一个系统通知

3. 如何回收这些对象？

了解完 Java 程序中对象的回收条件，那么垃圾回收线程又是如何回收这些对象的呢？JVM 垃圾回收遵循以下两个特性。

自动性：Java 提供了一个系统级的线程来跟踪每一块分配出去的内存空间，当 JVM 处于空闲循环时，垃圾收集器线程会自动检查每一块分配出去的内存空间，然后自动回收每一块空闲的内存块。

不可预期性：一旦一个对象没有被引用了，该对象是否立刻被回收呢？答案是不可预期的。我们很难确定一个没有被引用的对象是不是会被立刻回收掉，因为有可能当程序结束后，这个对象仍在内存中。

垃圾回收线程在 JVM 中是自动执行的，Java 程序无法强制执行。我们唯一能做的就是通过调用 `System.gc` 方法来“建议”执行垃圾收集器，但是是否可执行，什么时候执行？仍然不可预期。

GC 算法

JVM 提供了不同的回收算法来实现这一套回收机制，通常垃圾收集器的回收算法可以分为以下几种：

回收算法类型	优点	缺点
标记-清除算法 (Mark-Sweep)	不需要移动对象，简单高效	标记-清除过程效率低，GC产生内存碎片
复制算法 (Copying)	简单高效，不会产生内存碎片	内存使用率低，且有可能产生频繁复制问题
标记-整理算法 (Mark-Compact)	综合了前两种算法的优点	仍需要移动局部对象
分代收集算法 (Generational Collection)	分区回收	对于长时间存活对象的场景回收效果不明显，甚至起到反作用

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现，JDK1.7 update14 之后 Hotspot 虚拟机所有的回收器整理如下（以下为服务端垃圾收集器）：

回收器类型	回收算法	特点	设置参数
Serial New / Serial Old回收器	复制算法/标记-清除算法	单线程复制回收，简单高效，但会暂停程序导致停顿	-XX:+UseSerialGC (年轻代、老年代回收器为：Serial New、Serial Old)
ParNew New / ParNew Old回收器	复制算法/标记-整理算法	多线程复制回收，降低了停顿时间，但容易增加上下文切换	-XX:+UseParNewGC (年轻代、老年代回收器为：ParNew New、Serial Old, JDK1.8中无效) -XX:+UseParallelOldGC (年轻代、老年代回收器为：Parallel Scavenge、Parallel Old)
Parallel Scavenge回收器	复制算法	并行回收器，追求高吞吐量，高效利用CPU	-XX:+UseParallelGC (年轻代、老年代回收器为：Parallel Scavenge、Serial Old) -XX:ParallelGCThreads=4 (设置并发线程)
CMS回收器	标记-清理算法	老年代回收器，高并发、低停顿，追求最短GC回收停顿时间，CPU占用比较高，响应时间快，停顿时间短	-XX:+UseConcMarkSweepGC (年轻代、老年代回收器为：ParNew New、CMS (Serial Old作为备用))
G1回收器	标记-整理+复制算法	高并发、低停顿，可预测停顿时间	-XX:+UseG1GC (年轻代、老年代回收器为：G1、G1) -XX:MaxGCPauseMillis=200 (设置最大暂停时间)

其实在 JVM 规范中并没有明确 GC 的运作方式，各个厂商可以采用不同的方式实现垃圾收集器。我们可以通过 **JVM 工具查询当前 JVM 使用的垃圾收集器类型**，首先通过 ps 命令查询出经常 ID，再通过 jmap -heap ID 查询出 JVM 的配置信息，其中就包括垃圾收集器的设置类型。

```
[root@localhost ~]# jmap -heap 29438
Attaching to process ID 29438, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
MinHeapFreeRatio          = 0
MaxHeapFreeRatio           = 100
MaxHeapSize                = 8589934592 (8192.0MB)
NewSize                    = 429391872 (409.5MB)
MaxNewSize                 = 1717567488 (1638.0MB)
OldSize                    = 1718091776 (1638.5MB)
NewRatio                   = 4
SurvivorRatio              = 4
MetaspaceSize              = 21807104 (20.796875MB)
CompressedClassSpaceSize   = 1073741824 (1024.0MB)
MaxMetaspaceSize           = 17592186044415 MB
G1HeapRegionSize           = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
capacity = 1405091840 (1340.0MB)
used     = 1374612800 (1310.9329223632812MB)
free     = 30479040 (29.06707763671875MB)
97.8308151017374% used
From Space:
capacity = 3670016 (3.5MB)
used     = 3620912 (3.4531707763671875MB)
free     = 49104 (0.0468292236328125MB)
98.66202218191964% used
To Space:
capacity = 12582912 (12.0MB)
used     = 0 (0.0MB)
free     = 12582912 (12.0MB)
0.0% used
PS Old Generation
capacity = 1718091776 (1638.5MB)
used     = 21777496 (20.768638610839844MB)
free     = 1696314280 (1617.7313613891602MB)
1.267539738226417% used

19957 interned strings occupying 2608464 bytes.
[root@localhost ~]#
```

GC 性能衡量指标

一个垃圾收集器在不同场景下表现出的性能也不一样，那么如何评价一个垃圾收集器的性能好坏呢？我们可以借助一些指标。

吞吐量：这里的吞吐量是指应用程序所花费的时间和系统总运行时间的比值。我们可以按照这个公式来计算 GC 的吞吐量：系统总运行时间 = 应用程序耗时 + GC 耗时。如果系统运行了 100 分钟，GC 耗时 1 分钟，则系统吞吐量为 99%。**GC 的吞吐量一般不能低于 95%。**

停顿时间：指垃圾收集器正在运行时，应用程序的暂停时间。对于串行回收器而言，停顿时间可能会比较长；而使用并发回收器，由于垃圾收集器和应用程序交替运行，程序的停顿时间就会变短，但其效率很可能不如独占垃圾收集器，系统的吞吐量也很可能会降低。

垃圾回收频率：多久发生一次指垃圾回收呢？通常垃圾回收的频率越低越好，增大堆内存空间可以有效降低垃圾回收发生的频率，但同时也意味着堆积的回收对象越多，最终也会增加回收时的停顿时间。所以我们只要适当地增大堆内存空间，保证正常的垃圾回收频率即可。

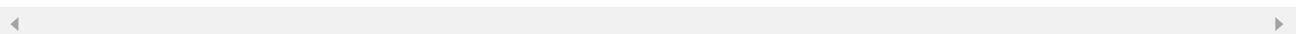
查看 & 分析 GC 日志

已知了性能衡量指标，现在我们需要通过工具查询 GC 相关日志，统计各项指标的信息。

首先，我们需要通过 JVM 参数预先设置 GC 日志，通常有以下几种 JVM 参数设置：

复制代码

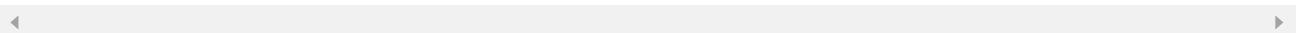
- 1 -XX:+PrintGC 输出 GC 日志
- 2 -XX:+PrintGCDetails 输出 GC 的详细日志
- 3 -XX:+PrintGCTimeStamps 输出 GC 的时间戳（以基准时间的形式）
- 4 -XX:+PrintGCDateStamps 输出 GC 的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）
- 5 -XX:+PrintHeapAtGC 在进行 GC 的前后打印出堆的信息
- 6 -Xloggc:.../logs/gc.log 日志文件的输出路径



这里使用如下参数来打印日志：

复制代码

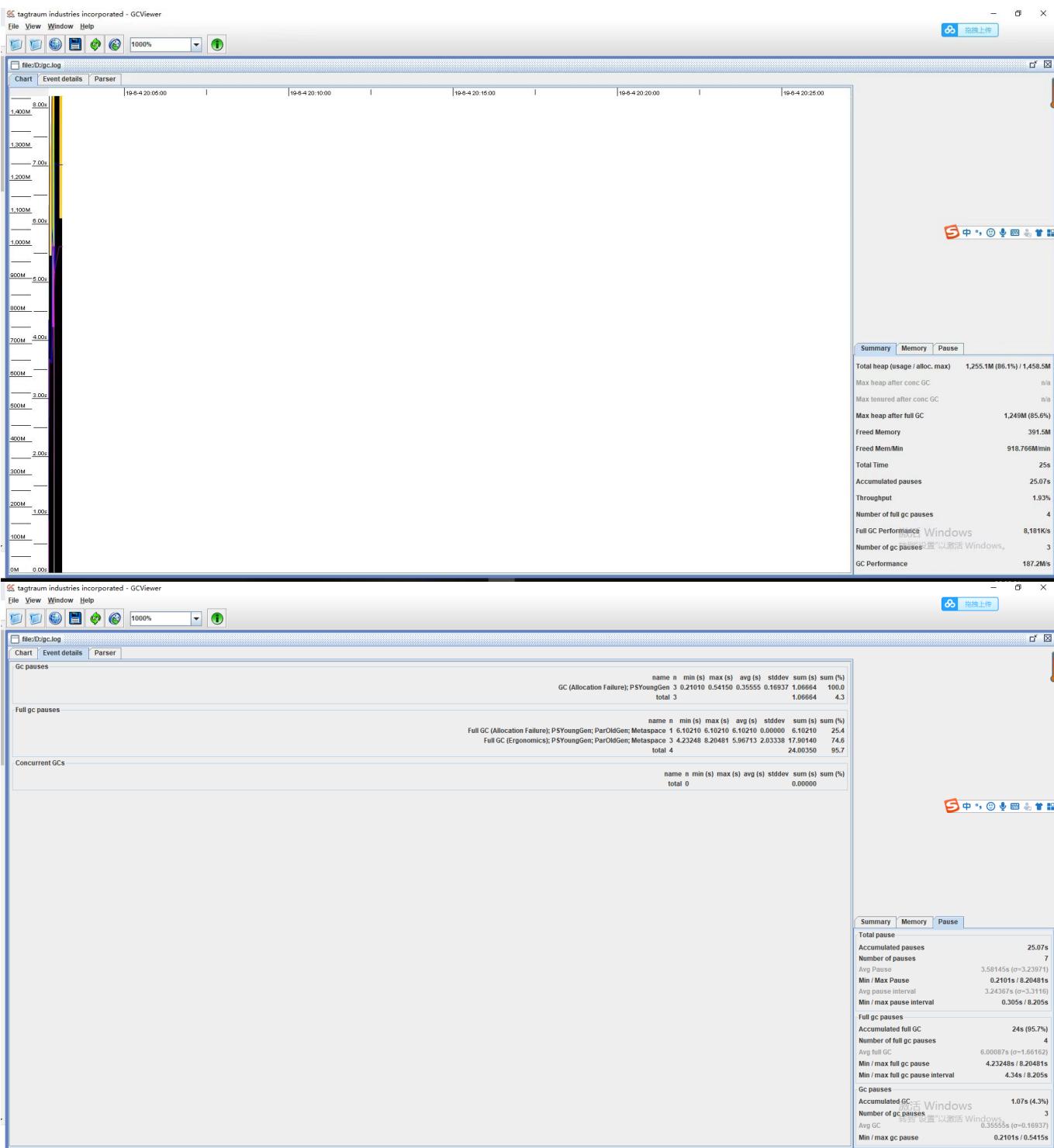
- 1 -XX:+PrintGCDateStamps -XX:+PrintGCDetails -Xloggc:./gclogs



打印后的日志为：

```
1 Java HotSpot(TM) 64-Bit Server VM (25.181-b13) for windows-amd64 JRE (1.8.0_181-b13), built on Jul 7 2018 04:01:33 by "java_re" with MS VC++ 10.0 (VS2010)
2 Memory: 4k page, physical 16696608k(4055756k free), swap 54445344k(33996432k free)
3 Command-line flags: -XX:InitialHeapSize=1048576000 -XX:MaxHeapSize=1572864000 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPoint
4 0.640: [GC (Allocation Failure)] [PSYoungGen: 234769K->42492K(298496K)] 234769K->170029K(981504K), 0.2100955 secs] [Times: user=0.67 sys=0.13, real=0.21 secs]
5 0.945: [GC (Allocation Failure)] [PSYoungGen: 298492K->42472K(469504K)] 426029K->366523K(1152512K), 0.3150446 secs] [Times: user=0.97 sys=0.16, real=0.32 secs]
6 1.432: [GC (Allocation Failure)] [PSYoungGen: 469480K->42472K(469504K)] 793531K->713297K(1152512K), 0.5414956 secs] [Times: user=1.25 sys=0.27, real=0.54 secs]
7 1.973: [Full GC (Ergonomics)] [PSYoungGen: 42472K->0K(469504K)] [ParOldGen: 670825K->657410K(1024000K)] 713297K->657410K(1493504K), [Metaspace: 4887K->4887K(10567
8 7.557: [Full GC (Ergonomics)] [PSYoungGen: 427008K->0K(469504K)] [ParOldGen: 657410K->950235K(1024000K)] 1084418K->950235K(1493504K), [Metaspace: 5305K->5305K(105
9 11.897: [Full GC (Ergonomics)] [PSYoungGen: 334950K->254985K(469504K)] [ParOldGen: 950235K->1023975K(1024000K)] 1285186K->1278961K(1493504K), [Metaspace: 5305K->5
10 20.102: [Full GC (Allocation Failure)] [PSYoungGen: 254985K->254985K(469504K)] [ParOldGen: 1023975K->1023898K(1024000K)] 1278961K->1278884K(1493504K), [Metaspace: 
11 Heap
12 PSYoungGen total 469504K, used 274663K [0x00000000e0c00000, 0x0000000100000000, 0x0000000100000000)
13 eden space 427008K, 64% used [0x00000000e0c00000, 0x00000000f1839d38, 0x00000000fad00000)
14 from space 42496K, 0% used [0x00000000fad00000, 0x00000000fad40000, 0x00000000fd680000)
15 to space 42496K, 0% used [0x00000000fd680000, 0x00000000fd680000, 0x0000000010000000)
16 ParOldGen total 1024000K, used 1023898K [0x00000000a2400000, 0x00000000e0c00000, 0x00000000e0c00000)
17 object space 1024000K, 99% used [0x00000000a2400000, 0x00000000e0bee910, 0x00000000e0c00000)
18 Metaspace used 5338K, capacity 5420K, committed 5504K, reserved 1056768K
19 class space used 565K, capacity 592K, committed 640K, reserved 1048576K
20
```

上图是运行很短时间的 GC 日志，如果是长时间的 GC 日志，我们很难通过文本形式去查看整体的 GC 性能。此时，我们可以通过[GCView](#)工具打开日志文件，图形化界面查看整体的 GC 性能，如下图所示：

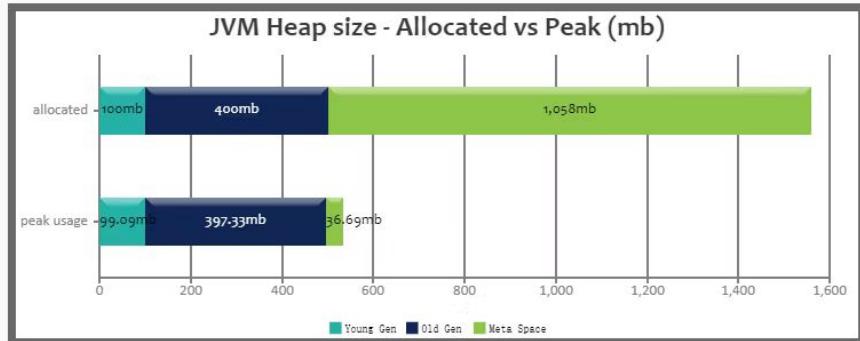


通过工具，我们可以看到吞吐量、停顿时间以及 GC 的频率，从而可以非常直观地了解到 GC 的性能情况。

这里我再推荐一个比较好用的 GC 日志分析工具，[GCEasy](#)是一款非常直观的 GC 日志分析工具，我们可以将日志文件压缩之后，上传到 GCEasy 官网即可看到非常清楚的 GC 日志分析结果：

JVM Heap Size

Generation	Allocated	Peak
Young Generation	100 mb	99.09 mb
Old Generation	400 mb	397.33 mb
Meta Space	1.03 gb	36.69 mb
Young + Old + Meta space	1.52 gb	524.64 mb



🔍 Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

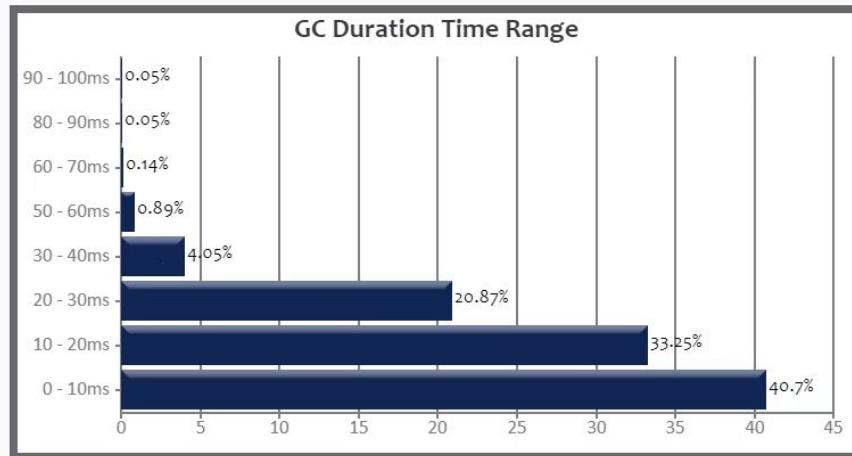
➊ Throughput: 56.414%

➋ Latency:

Avg Pause GC Time	18.5 ms
Max Pause GC Time	90.0 ms

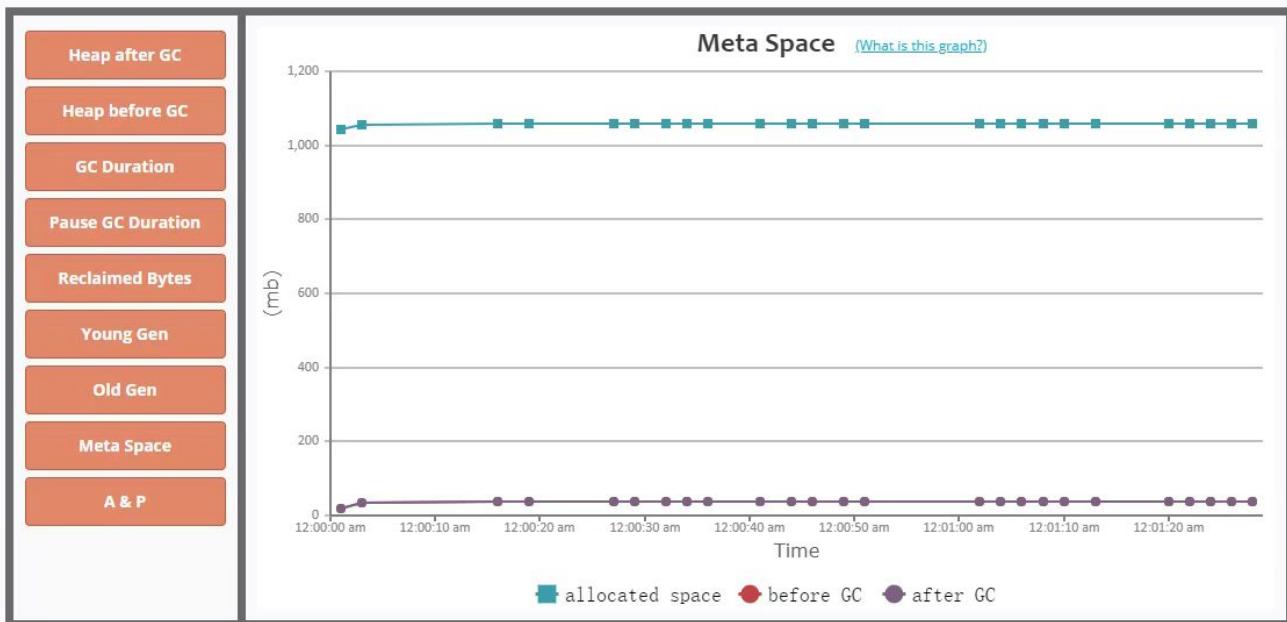
GC Pause Duration Time Range:

Duration (ms)	No. of GCs	Percentage
0 - 10	864	40.7%
10 - 20	706	33.25%
20 - 30	443	20.87%
30 - 40	86	4.05%
50 - 60	19	0.89%
60 - 70	3	0.14%
80 - 90	1	0.05%
90 - 100	1	0.05%

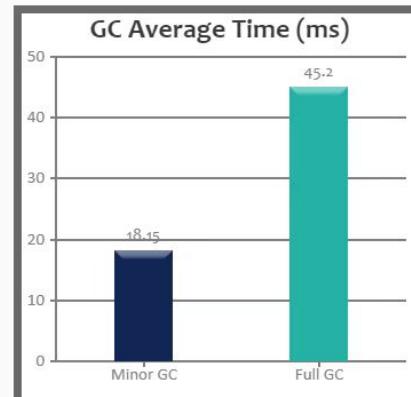
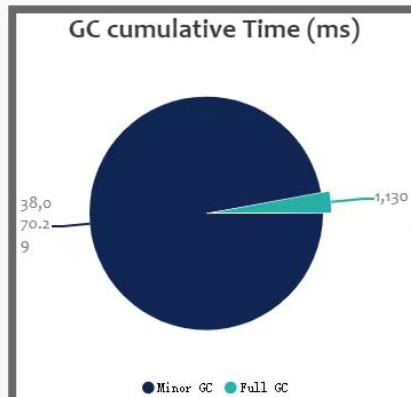
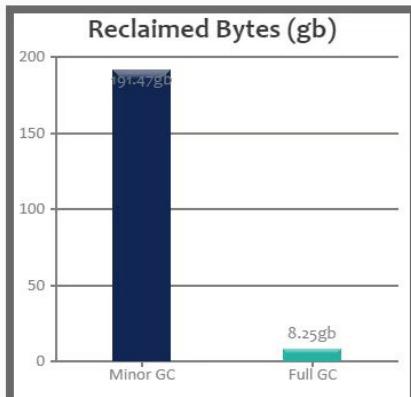


Interactive Graphs

(All graphs are zoomable)



GC Statistics [?](#)



Total GC stats

Total GC count ?	2123
Total reclaimed bytes ?	199.71 gb
Total GC time ?	39 sec 200 ms
Avg GC time ?	18.5 ms
GC avg time std dev	10.6 ms
GC min/max time	0 / 90.0 ms
GC Interval avg time ?	42.0 ms

Minor GC stats

Minor GC count	2098
Minor GC reclaimed ?	191.47 gb
Minor GC total time	38 sec 70 ms
Minor GC avg time ?	18.1 ms
Minor GC avg time std dev	10.1 ms
Minor GC min/max time	0 / 60.0 ms
Minor GC Interval avg ?	42.0 ms

Full GC stats

Full GC Count	25
Full GC reclaimed ?	8.25 gb
Full GC total time	1 sec 130 ms
Full GC avg time ?	45.2 ms
Full GC avg time std dev	13.9 ms
Full GC min/max time	30.0 ms / 90.0 ms
Full GC Interval avg ?	3 sec 650 ms

GC Pause Statistics

Pause Count	2123
Pause total time	39 sec 200 ms
Pause avg time <small>?</small>	18.5 ms
Pause avg time std dev	0.0
Pause min/max time	0 / 90.0 ms

⚙ Object Stats

(These are perfect [micro-metrics](#) to include in your performance reports)

Total created bytes <small>?</small>	200.04 gb
Total promoted bytes <small>?</small>	8.56 gb
Avg creation rate <small>?</small>	2.22 gb/sec
Avg promotion rate <small>?</small>	97.45 mb/sec

📄 Command Line Flags ?

```
-XX:InitialHeapSize=524288000 -XX:MaxHeapSize=524288000 -XX:MaxNewSize=104857600 -XX:NewSize=104857600 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps  
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
```

GC 调优策略

找出问题后，就可以进行调优了，下面介绍几种常用的 GC 调优策略。

1. 降低 Minor GC 频率

通常情况下，由于新生代空间较小，Eden 区很快被填满，就会导致频繁 Minor GC，因此我们可以通过增大新生代空间来降低 Minor GC 的频率。

可能你会有这样的疑问，扩容 Eden 区虽然可以减少 Minor GC 的次数，但不会增加单次 Minor GC 的时间吗？如果单次 Minor GC 的时间增加，那也很难达到我们期待的优化效果呀。

我们知道，单次 Minor GC 时间是由两部分组成：T1（扫描新生代）和 T2（复制存活对象）。假设一个对象在 Eden 区的存活时间为 500ms，Minor GC 的时间间隔是 300ms，那么正常情况下，Minor GC 的时间为：T1 + T2。

当我们增大新生代空间，Minor GC 的时间间隔可能会扩大到 600ms，此时一个存活 500ms 的对象就会在 Eden 区中被回收掉，此时就不存在复制存活对象了，所以再发生

Minor GC 的时间为：两次扫描新生代，即 $2T_1$ 。

可见，扩容后，Minor GC 时增加了 T_1 ，但省去了 T_2 的时间。通常在虚拟机中，复制对象的成本要远高于扫描成本。

如果在堆内存中存在较多的长期存活的对象，此时增加年轻代空间，反而会增加 Minor GC 的时间。如果堆中的短期对象很多，那么扩容新生代，单次 Minor GC 时间不会显著增加。因此，单次 Minor GC 时间更多取决于 GC 后存活对象的数量，而非 Eden 区的大小。

2. 降低 Full GC 的频率

通常情况下，由于堆内存空间不足或老年代对象太多，会触发 Full GC，频繁的 Full GC 会带来上下文切换，增加系统的性能开销。我们可以使用哪些方法来降低 Full GC 的频率呢？

减少创建大对象：在平常的业务场景中，我们习惯一次性从数据库中查询出一个大对象用于 web 端显示。例如，我之前碰到过一个一次性查询出 60 个字段的业务操作，这种大对象如果超过年轻代最大对象阈值，会被直接创建在老年代；即使被创建在了年轻代，由于年轻代的内存空间有限，通过 Minor GC 之后也会进入到老年代。这种大对象很容易产生较多的 Full GC。

我们可以将这种大对象拆解出来，首次只查询一些比较重要的字段，如果还需要其它字段辅助查看，再通过第二次查询显示剩余的字段。

增大堆内存空间：在堆内存不足的情况下，增大堆内存空间，且设置初始化堆内存为最大堆内存，也可以降低 Full GC 的频率。

选择合适的 GC 回收器

假设我们有这样一个需求，要求每次操作的响应时间必须在 500ms 以内。这个时候我们一般会选择响应速度较快的 GC 回收器，CMS (Concurrent Mark Sweep) 回收器和 G1 回收器都是不错的选择。

而当我们的需求对系统吞吐量有要求时，就可以选择 Parallel Scavenge 回收器来提高系统的吞吐量。

总结

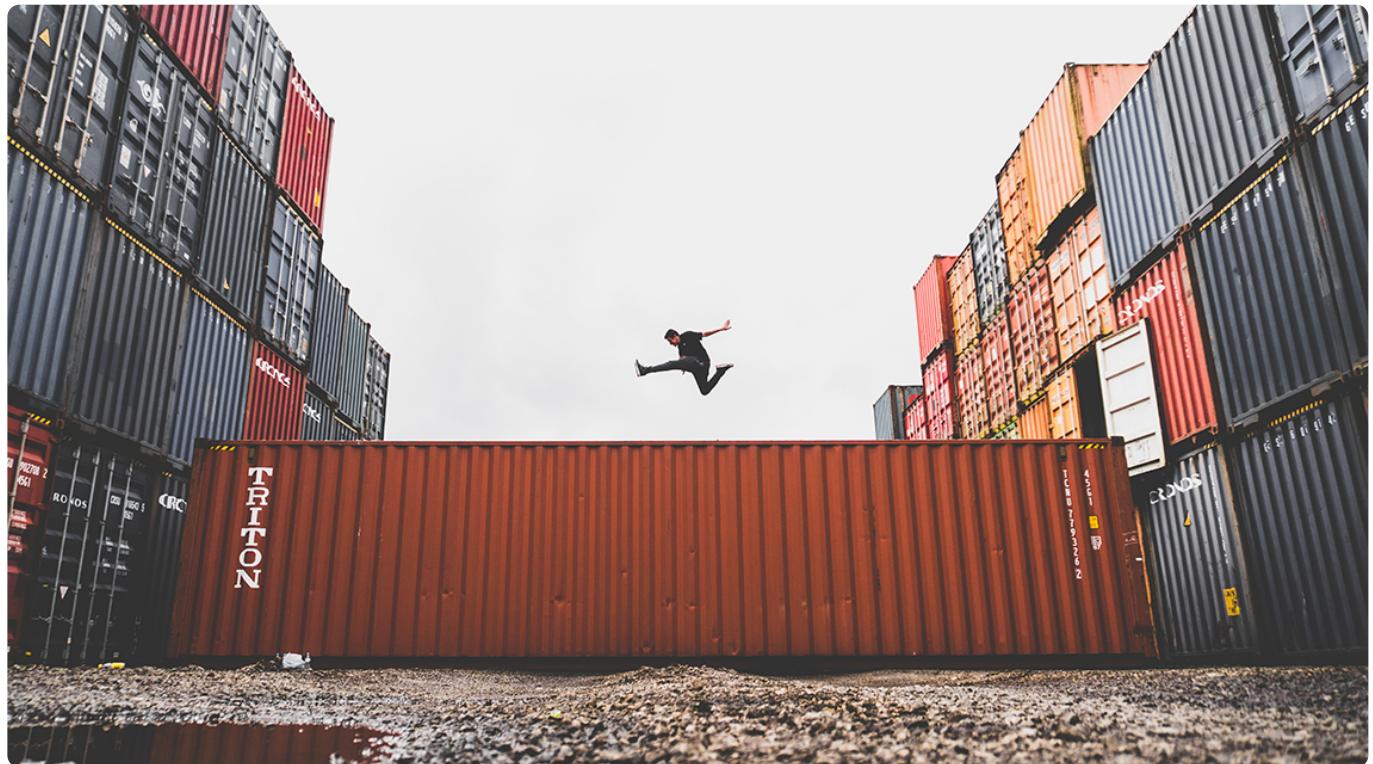
今天的内容比较多，最后再强调几个重点。

垃圾收集器的种类很多，我们可以将其分成两种类型，一种是响应速度快，一种是吞吐量高。通常情况下，CMS 和 G1 回收器的响应速度快，Parallel Scavenge 回收器的吞吐量高。

在 JDK1.8 环境下，默认使用的是 Parallel Scavenge（年轻代）+ Serial Old（老年代）垃圾收集器，你可以通过文中介绍的查询 JVM 的 GC 默认配置方法进行查看。

通常情况，JVM 是默认垃圾回收优化的，在没有性能衡量标准的前提下，尽量避免修改 GC 的一些性能配置参数。如果一定要改，那就必须基于大量的测试结果或线上的具体性能来进行调整。

23 | 如何优化JVM内存分配？



JVM 调优是一个系统而又复杂的过程，但我们知道，在大多数情况下，我们基本不用去调整 JVM 内存分配，因为一些初始化的参数已经可以保证应用服务正常稳定地工作了。

但所有的调优都是有目标性的，JVM 内存分配调优也一样。没有性能问题的时候，我们自然不会随意改变 JVM 内存分配的参数。那有了问题呢？**有了什么样的性能问题我们需要对其进行调优呢？又该如何调优呢？**这就是我今天要分享的内容。

JVM 内存分配性能问题

谈到 JVM 内存表现出的性能问题时，你可能会想到一些线上的 JVM 内存溢出事故。但这些方面的事故往往是应用程序创建对象导致的内存回收对象难，一般属于代码编程问题。

但其实很多时候，在应用服务的特定场景下，JVM 内存分配不合理带来的性能表现并不会像内存溢出问题这么突出。可以说如果你没有深入到各项性能指标中去，是很难发现其中隐藏的性能损耗。

JVM 内存分配不合理最直接的表现就是频繁的 GC，这会导致上下文切换等性能问题，从而降低系统的吞吐量、增加系统的响应时间。因此，**如果你在线上环境或性能测试时，发现频繁的 GC，且是正常的对象创建和回收，这个时候就需要考虑调整 JVM 内存分配了**，从而减少 GC 所带来的性能开销。

对象在堆中的生存周期

了解了性能问题，那需要做的势必就是调优了。但先别急，在了解 JVM 内存分配的调优过程之前，我们先来看看一个新创建的对象在堆内存中的生存周期，为后面的学习打下基础。

在[第 20 讲](#)中，我讲过 JVM 内存模型。我们知道，在 JVM 内存模型的堆中，堆被划分为新生代和老年代，新生代又被进一步划分为 Eden 区和 Survivor 区，最后 Survivor 由 From Survivor 和 To Survivor 组成。

当我们新建一个对象时，对象会被优先分配到新生代的 Eden 区中，这时虚拟机会给对象定义一个对象年龄计数器（通过参数 -XX:MaxTenuringThreshold 设置）。

同时，也有另外一种情况，当 Eden 空间不足时，虚拟机将会执行一个新生代的垃圾回收（Minor GC）。这时 JVM 会把存活的对象转移到 Survivor 中，并给对象的年龄 +1。对象在 Survivor 中同样也会经历 MinorGC，每经过一次 MinorGC，对象的年龄将会 +1。

当然了，内存空间也是有设置阈值的，可以通过参数 -XX:PerTenureSizeThreshold 设置直接被分配到老年代的最大对象，这时如果分配的对象超过了设置的阈值，对象就会直接被分配到老年代，这样做的好处就是可以减少新生代的垃圾回收。

查看 JVM 堆内存分配

我们知道了一个对象从创建至回收到堆中的过程，接下来我们再来了解下 JVM 堆内存是如何分配的。在默认不配置 JVM 堆内存大小的情况下，JVM 根据默认值来配置当前内存大

小。我们可以通过以下命令来查看堆内存配置的默认值：

 复制代码

```
1 java -XX:+PrintFlagsFinal -version | grep HeapSize
2 jmap -heap 17284
```

```
[root@localhost ~]# java -XX:+PrintFlagsFinal -version | grep HeapSize
  uintx ErgoHeapSizeLimit          = 0                               {product}
  uintx HeapSizePerGCThread       = 87241520                      {product}
  uintx InitialHeapSize           := 130023424                     {product}
  uintx LargePageHeapSizeThreshold = 134217728                     {product}
  uintx MaxHeapSize               := 2051014656                    {product}
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)

[root@localhost ~]# ps -ef|grep java
root    17284 10321  0 17:19 pts/1    00:00:11 java -jar heapTest-0.0.1-SNAPSHOT.jar
root    17741 17723  0 19:07 pts/2    00:00:00 grep --color=auto java
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# jmap -heap 17284
Attaching to process ID 17284, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize           = 2051014656 (1956.0MB)
  NewSize               = 42991616 (41.0MB)
  MaxNewSize             = 683671552 (652.0MB)
  oldSize                = 87031808 (83.0MB)
  NewRatio              = 2
  SurvivorRatio          = 8
  Metaspacesize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspacesize       = 17592186044415 MB
  G1HeapRegionSize        = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
  capacity = 254803968 (243.0MB)
  used     = 19794256 (18.877273559570312MB)
  free     = 235009712 (224.1227264404297MB)
  7.768425333156507% used
From Space:
  capacity = 8388608 (8.0MB)
  used     = 0 (0.0MB)
  free     = 8388608 (8.0MB)
  0.0% used
To Space:
  capacity = 9961472 (9.5MB)
  used     = 0 (0.0MB)
  free     = 9961472 (9.5MB)
  0.0% used
PS Old Generation
  capacity = 70254592 (67.0MB)
  used     = 17130840 (16.337242126464844MB)
  free     = 53123752 (50.662757873535156MB)
  24.383943472335588% used
```

通过命令，我们可以获得在这台机器上启动的 JVM 默认最大堆内存为 1953MB，初始化大小为 124MB。

在 JDK1.7 中，默认情况下年轻代和老年代的比例是 1:2，我们可以通过-XX:NewRatio 重置该配置项。年轻代中的 Eden 和 To Survivor、From Survivor 的比例是 8:1:1，我们可以通过 -XX:SurvivorRatio 重置该配置项。

在 JDK1.7 中如果开启了 -XX:+UseAdaptiveSizePolicy 配置项，JVM 将会动态调整 Java 堆中各个区域的大小以及进入老年代的年龄，-XX:NewRatio 和 -XX:SurvivorRatio 将会失效，而 JDK1.8 是默认开启 -XX:+UseAdaptiveSizePolicy 配置项的。

还有，在 JDK1.8 中，不要随便关闭 UseAdaptiveSizePolicy 配置项，除非你已经对初始化堆内存 / 最大堆内存、年轻代 / 老年代以及 Eden 区 /Survivor 区有非常明确的规划了。否则 JVM 将会分配最小堆内存，年轻代和老年代按照默认比例 1:2 进行分配，年轻代中的 Eden 和 Survivor 则按照默认比例 8:2 进行分配。这个内存分配未必是应用服务的最佳配置，因此可能会给应用服务带来严重的性能问题。

JVM 内存分配的调优过程

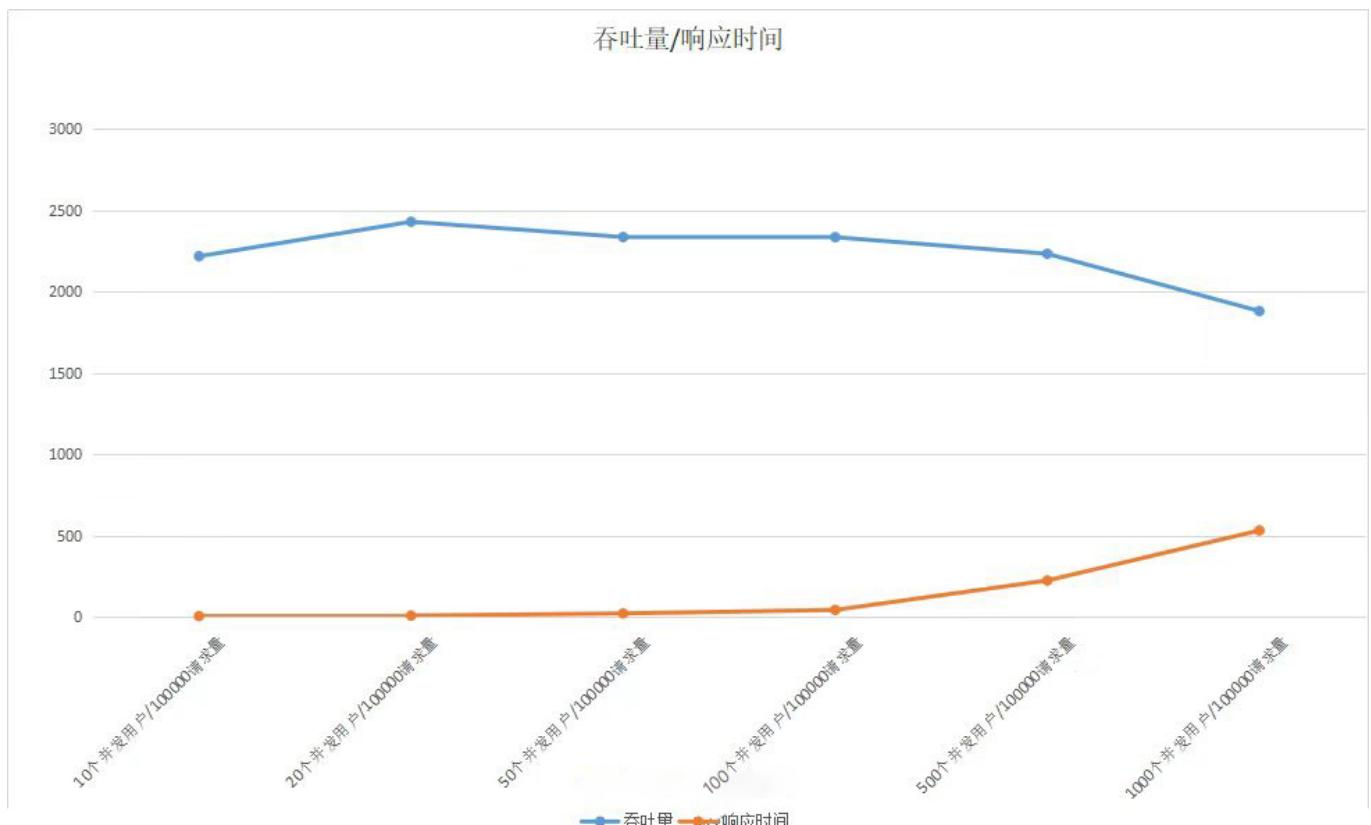
我们先使用 JVM 的默认配置，观察应用服务的运行情况，下面我将结合一个实际案例来讲述。现模拟一个抢购接口，假设需要满足一个 5W 的并发请求，且每次请求会产生 20KB 对象，我们可以通过千级并发创建一个 1MB 对象的接口来模拟万级并发请求产生大量对象的场景，具体代码如下：

 复制代码

```
1      @RequestMapping(value = "/test1")
2      public String test1(HttpServletRequest request) {
3          List<Byte[]> temp = new ArrayList<Byte[]>();
4
5          Byte[] b = new Byte[1024*1024];
6          temp.add(b);
7
8          return "success";
9      }
```

AB 压测

分别对应用服务进行压力测试，以下是请求接口的吞吐量和响应时间在不同并发用户数下的变化情况：



可以看到，当并发数量到了一定值时，吞吐量就上不去了，响应时间也迅速增加。那么，在 JVM 内部运行又是怎样的呢？

分析 GC 日志

此时我们可以通过 GC 日志查看具体的回收日志。我们可以通过设置 VM 配置参数，将运行期间的 GC 日志 dump 下来，具体配置参数如下：

复制代码

```
1 -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:/log/heapTest.log
```

以下是一个滚动条示例，展示了更多的配置项说明。

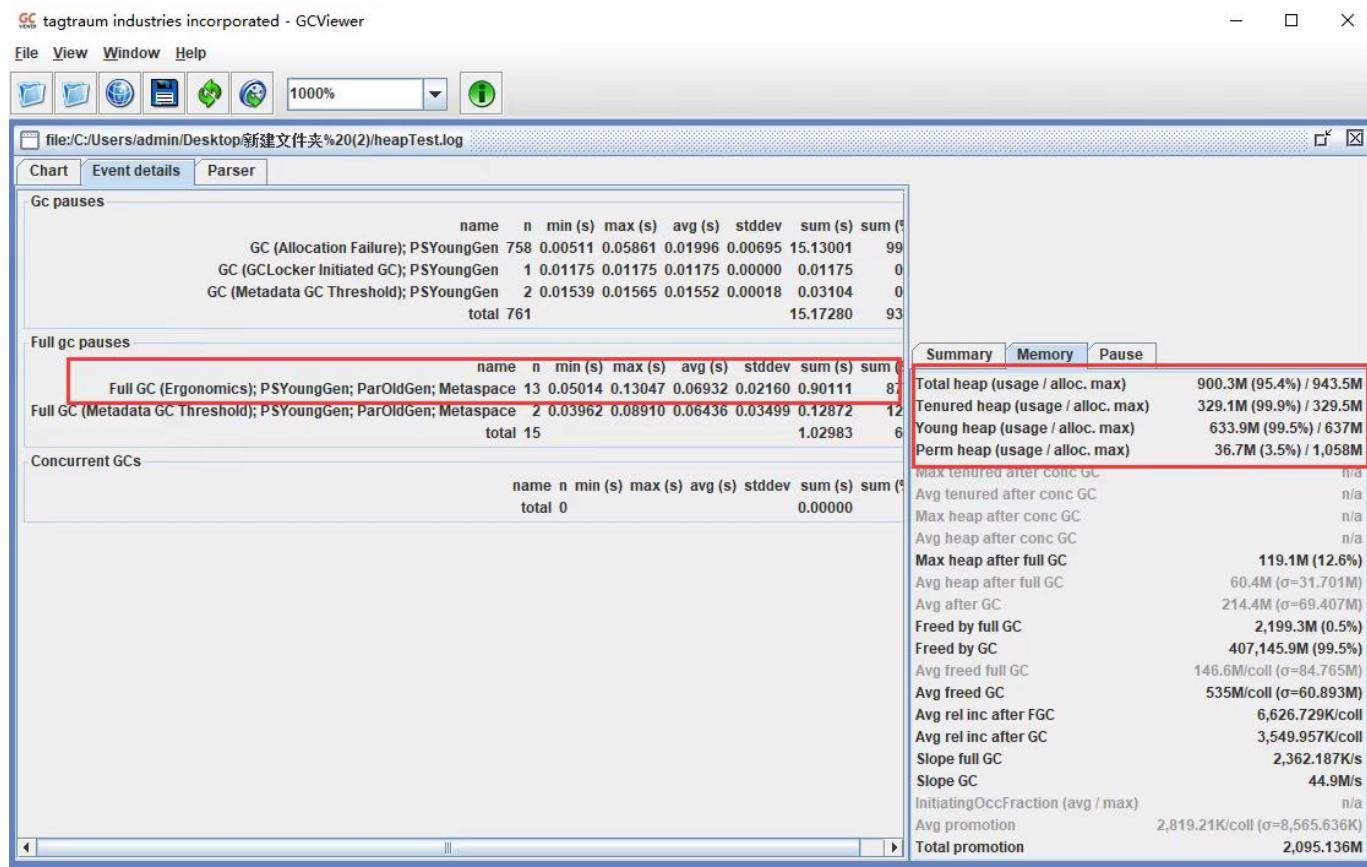
以下是各个配置项的说明：

-XX:PrintGCTimeStamps：打印 GC 具体时间；

-XX:PrintGCDetails：打印出 GC 详细日志；

-Xloggc: path : GC 日志生成路径。

收集到 GC 日志后，我们就可以使用[第 22 讲](#)中介绍过的 GCViewer 工具打开它，进而查看到具体的 GC 日志如下：



主页面显示 FullGC 发生了 13 次，右下角显示年轻代和老年代的内存使用率几乎达到了 100%。而 FullGC 会导致 stop-the-world 的发生，从而严重影响到应用服务的性能。此时，我们需要调整堆内存的大小来减少 FullGC 的发生。

参考指标

我们可以将某些指标的预期值作为参考指标，上面的 GC 频率就是其中之一，那么还有哪些指标可以为我们提供一些具体的调优方向呢？

GC 频率：高频的 FullGC 会给系统带来非常大的性能消耗，虽然 MinorGC 相对 FullGC 来说好了许多，但过多的 MinorGC 仍会给系统带来压力。

内存：这里的内存指的是堆内存大小，堆内存又分为年轻代内存和老年代内存。首先我们要分析堆内存大小是否合适，其实是分析年轻代和老年代的比例是否合适。如果内存不足或分配不均匀，会增加 FullGC，严重的将导致 CPU 持续爆满，影响系统性能。

吞吐量：频繁的 FullGC 将会引起线程的上下文切换，增加系统的性能开销，从而影响每次处理的线程请求，最终导致系统的吞吐量下降。

延时：JVM 的 GC 持续时间也会影响到每次请求的响应时间。

具体调优方法

调整堆内存空间减少 FullGC：通过日志分析，堆内存基本被用完了，而且存在大量 FullGC，这意味着我们的堆内存严重不足，这个时候我们需要调大堆内存空间。

 复制代码

```
1 java -jar -Xms4g -Xmx4g heapTest-0.0.1-SNAPSHOT.jar
```

以下是各个配置项的说明：

-Xms：堆初始大小；

-Xmx：堆最大值。

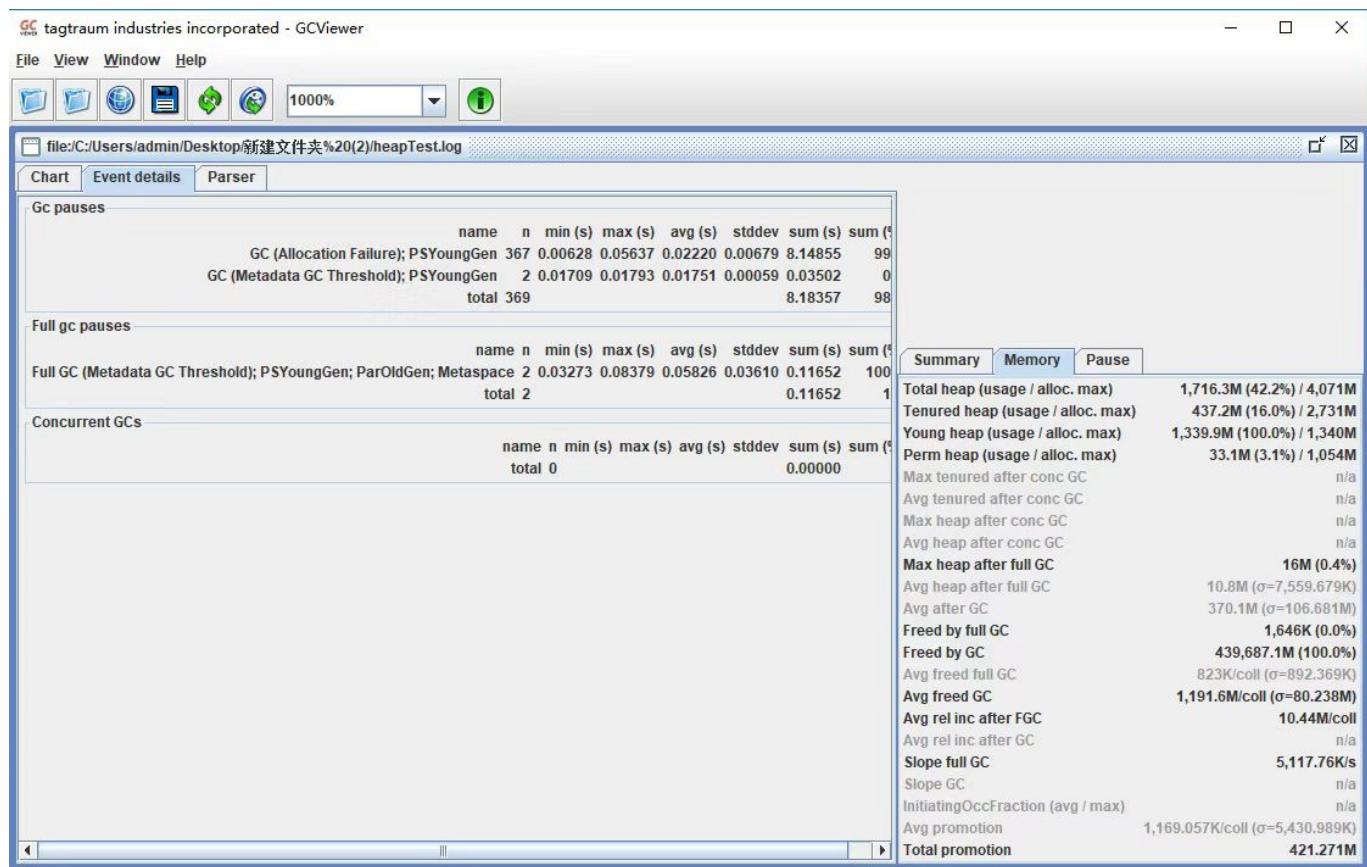
调大堆内存之后，我们再来测试下性能情况，发现吞吐量提高了 40% 左右，响应时间也降低了将近 50%。

```
Concurrency Level:      1000
Time taken for tests:  37.677 seconds
Complete requests:     100000
Failed requests:       0
Write errors:          0
Total transferred:    13900000 bytes
HTML transferred:     700000 bytes
Requests per second:   2654.12 [#/sec] (mean)
Time per request:     376.773 [ms] (mean)
Time per request:     0.377 [ms] (mean, across all concurrent requests)
Transfer rate:         360.28 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  232 773.0      0    7021
Processing:     1   134 120.9     102    1729
Waiting:        1   125 102.3      99    1729
Total:          1   366 783.4     117    8033

Percentage of the requests served within a certain time (ms)
  50%    117
  66%    183
  75%    246
  80%    328
  90%   1092
  95%   1220
  98%   3096
  99%   3224
 100%  8033 (longest request)
```

再查看 GC 日志，发现 FullGC 频率降低了，老年代的使用率只有 16% 了。



调整年轻代减少 MinorGC：通过调整堆内存大小，我们已经提升了整体的吞吐量，降低了响应时间。那还有优化空间吗？我们还可以将年轻代设置得大一些，从而减少一些 MinorGC（[第 22 讲](#)有通过降低 Minor GC 频率来提高系统性能的详解）。

复制代码

```
1 java -jar -Xms4g -Xmx4g -Xmn3g heapTest-0.0.1-SNAPSHOT.jar
```

再进行 AB 压测，发现吞吐量上去了。

```

Document Path:          /test2
Document Length:        7 bytes

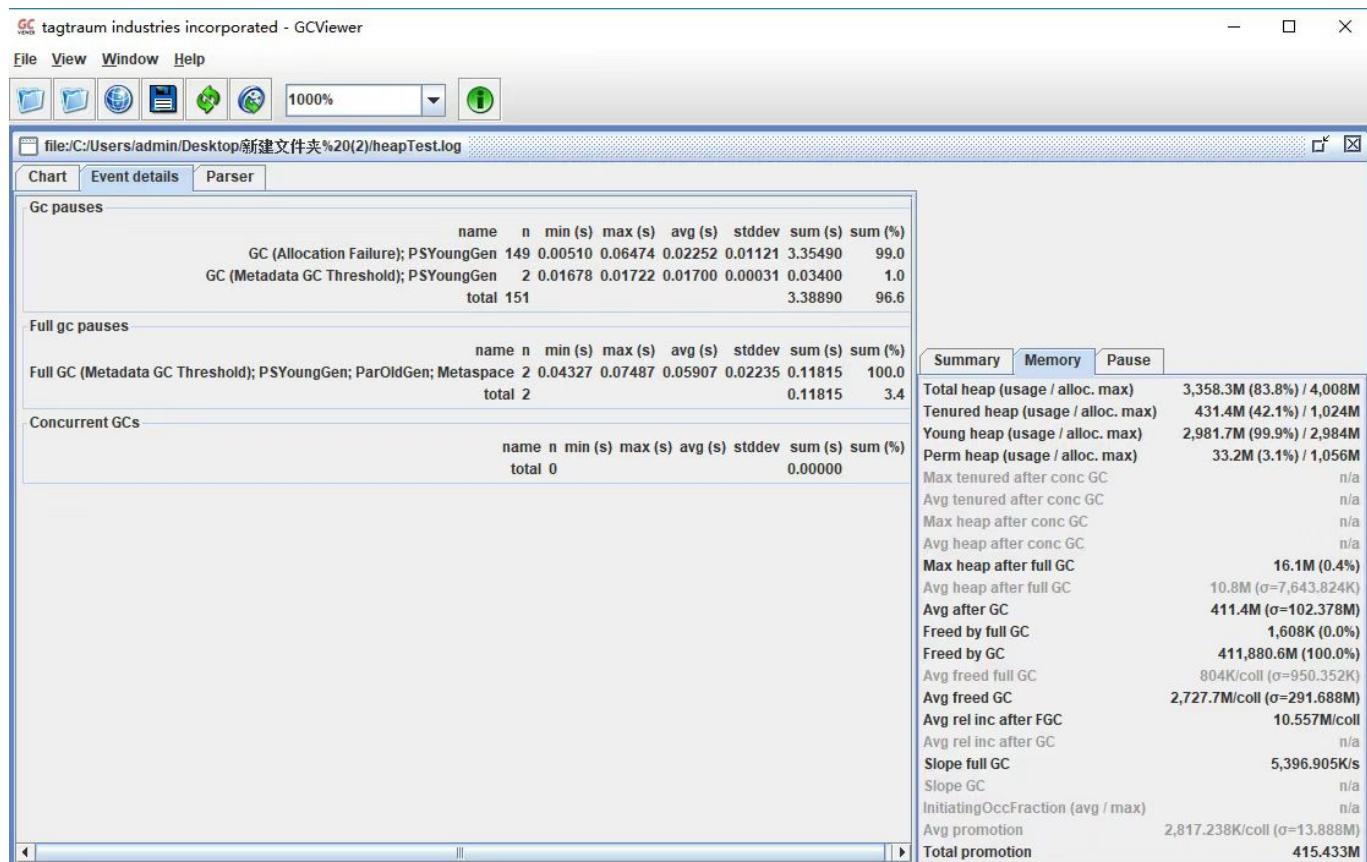
Concurrency Level:      1000
Time taken for tests:   34.157 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      13900000 bytes
HTML transferred:       7000000 bytes
Requests per second:    2927.68 [#/sec] (mean)
Time per request:       341.568 [ms] (mean)
Time per request:       0.342 [ms] (mean, across all concurrent requests)
Transfer rate:          397.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  208 812.5      0 15040
Processing:     1  126 117.2      97 1766
Waiting:        1  120 106.9      94 1683
Total:          1  334 824.0     114 15225

Percentage of the requests served within a certain time (ms)
 50%   114
 66%   166
 75%   216
 80%   266
 90%  1076
 95%  1184
 98%  3066
 99%  3160
100% 15225 (longest request)

```

再查看 GC 日志，发现 MinorGC 也明显降低了，GC 花费的总时间也减少了。



设置 Eden、Survivor 区比例：在 JVM 中，如果开启 AdaptiveSizePolicy，则每次 GC 后都会重新计算 Eden、From Survivor 和 To Survivor 区的大小，计算依据是 GC 过程中

统计的 GC 时间、吞吐量、内存占用量。这个时候 SurvivorRatio 默认设置的比例会失效。

在 JDK1.8 中，默认是开启 AdaptiveSizePolicy 的，我们可以通过 -XX:-UseAdaptiveSizePolicy 关闭该项配置，或显示运行 -XX:SurvivorRatio=8 将 Eden、Survivor 的比例设置为 8:2。大部分新对象都是在 Eden 区创建的，我们可以固定 Eden 区的占用比例，来调优 JVM 的内存分配性能。

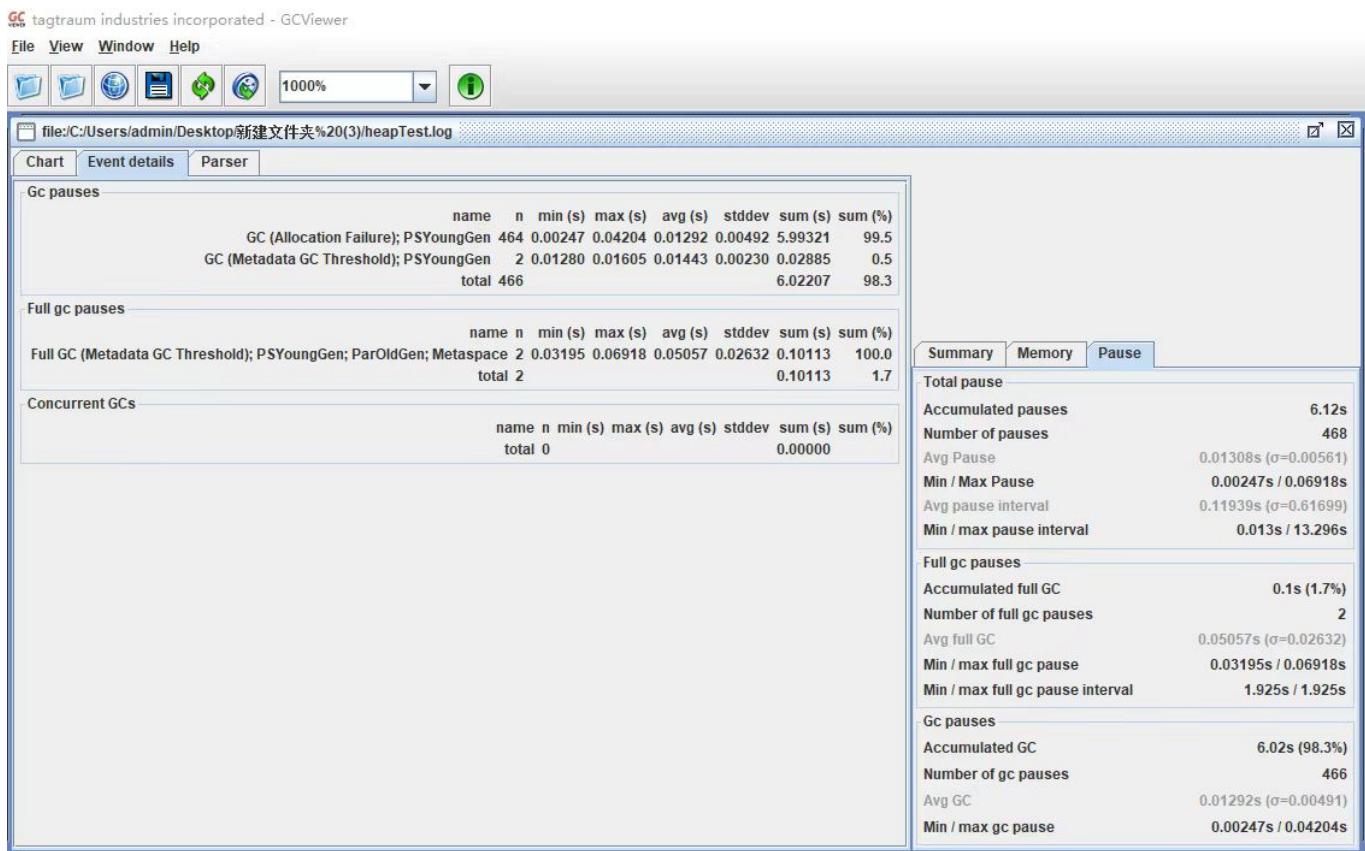
再进行 AB 性能测试，我们可以看到吞吐量提升了，响应时间降低了。

```
Document Path:          /test2
Document Length:        7 bytes

Concurrency Level:      1000
Time taken for tests:  33.322 seconds
Complete requests:     100000
Failed requests:        0
Write errors:           0
Total transferred:     13900000 bytes
HTML transferred:      700000 bytes
Requests per second:   3001.03 [#/sec] (mean)
Time per request:      333.219 [ms] (mean)
Time per request:      0.333 [ms] (mean, across all concurrent requests)
Transfer rate:          407.37 [Kbytes/sec] received

Connection Times (ms)
                  min  mean[+/-sd] median   max
Connect:          0    222  909.2      0   31044
Processing:       1    101   79.2      82    975
Waiting:          1     94   64.5      80    975
Total:            1    323  913.2      94   31135

Percentage of the requests served within a certain time (ms)
 50%    94
 66%   133
 75%   171
 80%   210
 90%  1069
 95%  1155
 98%  3063
 99%  3136
100% 31135 (longest request)
```



总结

JVM 内存调优通常和 GC 调优是互补的，基于以上调优，我们可以继续对年轻代和堆内存的垃圾回收算法进行调优。这里可以结合上一讲的内容，一起完成 JVM 调优。

虽然分享了一些 JVM 内存分配调优的常用方法，但我还是建议你在进行性能压测后如果没有发现突出的性能瓶颈，就继续使用 JVM 默认参数，起码在大部分的场景下，默认配置已经可以满足我们的需求了。但满足不了也不要慌张，结合今天所学的内容去实践一下，相信你会有新的收获。

24 | 内存持续上升，我该如何排查问题？



我想你肯定遇到过内存溢出，或是内存使用率过高的问题。碰到内存持续上升的情况，其实我们很难从业务日志中查看到具体的问题，那么面对多个进程以及大量业务线程，我们该如何精准地找到背后的原因呢？

常用的监控和诊断内存工具

工欲善其事，必先利其器。平时排查内存性能瓶颈时，我们往往需要用到一些 Linux 命令行或者 JDK 工具来辅助我们监测系统或者虚拟机内存的使用情况，下面我就来介绍几种好用且常用的工具。

Linux 命令行工具之 top 命令

top 命令是我们在 Linux 下最常用的命令之一，它可以实时显示正在执行进程的 CPU 使用率、内存使用率以及系统负载等信息。其中上半部分显示的是系统的统计信息，下半部分显示的是进程的使用率统计信息。

```
[root@localhost ~]# top
top - 20:22:09 up 164 days, 7:02, 1 user, load average: 0.00, 0.01, 0.05
Tasks: 108 total, 1 running, 102 sleeping, 5 stopped, 0 zombie
%CPU(s): 0.1 us, 0.1 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 8010548 total, 3720256 free, 2534832 used, 1755460 buff/cache
KiB Swap: 4194300 total, 3692892 free, 501408 used. 4856596 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
28376 root      20   0  581600  7612 1296 S  1.3  0.1 587:54.76 containerd
12109 root      20   0  12.1g  1.3g 14180 S  0.3 16.8 10:31.92 java
15402 systemd+ 20   0 2860420  91372 5416 S  0.3  1.1 673:40.37 mysqld
  1 root      20   0 191068  2552 1332 S  0.0  0.0 13:37.42 systemd
  2 root      20   0      0      0      0 S  0.0  0.0 0:00.06 kthreadd
  3 root      20   0      0      0      0 S  0.0  0.0 0:30.36 ksoftirqd/0
  5 root      0 -20      0      0      0 S  0.0  0.0 0:00.00 kworker/0:0H
  7 root      rt  0      0      0      0 S  0.0  0.0 0:03.95 migration/0
  8 root      20   0      0      0      0 S  0.0  0.0 0:00.00 rCU_bh
  9 root      20   0      0      0      0 S  0.0  0.0 7:43.57 rCU_sched
 10 root     rt  0      0      0      0 S  0.0  0.0 1:51.56 watchdog/0
 11 root     rt  0      0      0      0 S  0.0  0.0 1:44.91 watchdog/1
 12 root     rt  0      0      0      0 S  0.0  0.0 0:02.78 migration/1
 13 root      20   0      0      0      0 S  0.0  0.0 0:04.37 ksoftirqd/1
 15 root      0 -20      0      0      0 S  0.0  0.0 0:00.00 kworker/1:0H
```

除了简单的 top 之外，我们还可以通过 top -Hp pid 查看具体线程使用系统资源情况：

```
[root@localhost ~]# top -Hp 1593
top - 14:27:02 up 171 days, 1:09, 5 users, load average: 0.00, 0.03, 0.18
Threads: 133 total, 0 running, 133 sleeping, 0 stopped, 0 zombie
%CPU(s): 1.5 us, 1.5 sy, 0.0 ni, 96.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 8010548 total, 3401292 free, 1722052 used, 2887204 buff/cache
KiB Swap: 4194300 total, 4184712 free, 9588 used. 5689024 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
1593 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1594 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:02.97 java
1595 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:18.27 java
1596 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:18.30 java
1597 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:18.01 java
1598 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:18.50 java
1599 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:17.00 java
1600 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1601 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1602 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1603 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:03.20 java
1604 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:03.43 java
1605 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:01.56 java
1606 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1607 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:01.16 java
1610 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.16 java
1611 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.18 java
1612 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.00 java
1613 root      20   0 4761292  1.2g 13512 S  0.0 15.1 0:00.09 java
```

Linux 命令行工具之 vmstat 命令

vmstat 是一款指定采样周期和次数的功能性监测工具，我们可以看到，它不仅可以统计内存的使用情况，还可以观测到 CPU 的使用率、swap 的使用情况。但 vmstat 一般很少用来查看内存的使用情况，而是经常被用来观察进程的上下文切换。

```
[root@localhost conf]# vmstat 1 3
procs -----memory----- ---swap-- -----io---- -system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
1 0 500040 2259324 156404 1634120 0 0 0 2 0 0 0 0 0 100 0 0
0 0 500040 2259184 156404 1634120 0 0 0 0 454 923 0 0 100 0 0
0 0 500040 2259076 156404 1634120 0 0 0 20 469 999 0 0 100 0 0
```

r : 等待运行的进程数；

b : 处于非中断睡眠状态的进程数；

swpd : 虚拟内存使用情况；

free : 空闲的内存；

buff : 用来作为缓冲的内存数；

si : 从磁盘交换到内存的交换页数量；

so : 从内存交换到磁盘的交换页数量；

bi : 发送到块设备的块数；

bo : 从块设备接收到的块数；

in : 每秒中断数；

cs : 每秒上下文切换次数；

us : 用户 CPU 使用时间；

sy : 内核 CPU 系统使用时间；

id : 空闲时间；

wa : 等待 I/O 时间；

st : 运行虚拟机窃取的时间。

Linux 命令行工具之 pidstat 命令

pidstat 是 Sysstat 中的一个组件，也是一款功能强大的性能监测工具，我们可以通过命令：yum install sysstat 安装该监控组件。之前的 top 和 vmstat 两个命令都是监测进程的内存、CPU 以及 I/O 使用情况，而 pidstat 命令则是深入到线程级别。

通过 pidstat -help 命令，我们可以查看到有以下几个常用的参数来监测线程的性能：

```
[root@localhost conf]# pidstat -help
Usage: pidstat [ options ] [ <interval> [ <count> ] ]
options are:
[ -d ] [ -h ] [ -I ] [ -l ] [ -r ] [ -s ] [ -t ] [ -u [ <username> ] ] [ -u ]
[ -v ] [ -w ] [ -c <command> ] [ -p { <pid> [,...] | SELF | ALL } ]
[ -T { TASK | CHILD | ALL } ]
```

常用参数：

-u：默认的参数，显示各个进程的 CPU 使用情况；

-r：显示各个进程的内存使用情况；

-d：显示各个进程的 I/O 使用情况；

-w：显示每个进程的上下文切换情况；

-p：指定进程号；

-t：显示进程中线程的统计信息。

我们可以通过相关命令（例如 ps 或 jps）查询到相关进程 ID，再运行以下命令来监测该进程的内存使用情况：

```
[root@localhost conf]# jps
28557 Bootstrap
12109 charge-server-0.0.1-SNAPSHOT.jar
3726 Jps
16079 cloud-web-0.0.1-SNAPSHOT.jar
[root@localhost conf]#
[root@localhost conf]#
[root@localhost conf]#
[root@localhost conf]# pidstat -p 28557 -r 1 3
Linux 3.10.0-514.el7.x86_64 (localhost)        06/26/2019        _x86_64_          (4 CPU)
03:54:59 PM      UID      PID  minfllt/s  majfllt/s    VSZ     RSS   %MEM  Command
03:55:00 PM      0      28557      0.00      0.00 5759424 684004   8.54  java
03:55:01 PM      0      28557      0.00      0.00 5759424 684004   8.54  java
03:55:02 PM      0      28557      0.00      0.00 5759424 684004   8.54  java
Average:         0      28557      0.00      0.00 5759424 684004   8.54  java
```

其中 pidstat 的参数 -p 用于指定进程 ID，-r 表示监控内存的使用情况，1 表示每秒的意思，3 则表示采样次数。

其中显示的几个关键指标的含义是：

Minfllt/s：任务每秒发生的次要错误，不需要从磁盘中加载页；

Majfllt/s：任务每秒发生的主要错误，需要从磁盘中加载页；

VSZ：虚拟地址大小，虚拟内存使用 KB；

RSS：常驻集合大小，非交换区内存使用 KB。

如果我们要继续查看该进程下的线程内存使用率，则在后面添加 -t 指令即可：

[root@localhost conf]# pidstat -p 28557 -r 1 3 -t Linux 3.10.0-514.el7.x86_64 (localhost) 06/26/2019 _x86_64_ (4 CPU)									
03:58:24 PM	UID	TGID	TID	minflt/s	majflt/s	VSZ	RSS	%MEM	Command
03:58:25 PM	0	28557	-	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28557	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28558	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28559	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28560	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28561	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28562	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28563	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28564	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28565	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28566	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28567	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28568	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28569	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28570	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28571	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28572	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28574	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28575	0.00	0.00	5759424	684004	8.54	java
03:58:25 PM	0	-	28576	0.00	0.00	5759424	684004	8.54	java

我们知道，Java 是基于 JVM 上运行的，大部分内存都是在 JVM 的用户内存中创建的，所以除了通过以上 Linux 命令来监控整个服务器内存的使用情况之外，我们更需要知道 JVM 中的内存使用情况。JDK 中就自带了很多命令工具可以监测到 JVM 的内存分配以及使用情况。

JDK 工具之 jstat 命令

jstat 可以监测 Java 应用程序的实时运行情况，包括堆内存信息以及垃圾回收信息。我们可以运行 jstat -help 查看一些关键参数信息：

```
[root@localhost conf]# jstat -help
Usage: jstat -help|-options
        jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

Definitions:
  <option>      An option reported by the -options option
  <vmid>        Virtual Machine Identifier. A vmid takes the following form:
                 <lvmid>[@<hostname>[:<port>]]
                 where <lvmid> is the local vm identifier for the target
                 Java virtual machine, typically a process id; <hostname> is
                 the name of the host running the target Java virtual machine;
                 and <port> is the port number for the rmiregistry on the
                 target host. See the jvmsstat documentation for a more complete
                 description of the Virtual Machine Identifier.

  <lines>        Number of samples between header lines.
  <interval>     Sampling interval. The following forms are allowed:
                 <n>["ms"|"s"]
                 where <n> is an integer and the suffix specifies the units as
                 milliseconds("ms") or seconds("s"). The default units are "ms".
  <count>        Number of samples to take before terminating.
  -J<flag>       Pass <flag> directly to the runtime system.
```

再通过 jstat -option 查看 jstat 有哪些操作：

```
[root@localhost conf]# jstat -options
-class
-compiler
-gc
-gccapacity
-gccause
-gcmetacapacity
-gcnew
-gcnewcapacity
-gcold
-gcoldcapacity
-gcutil
-printcompilation
```

- class : 显示 ClassLoad 的相关信息；
- compiler : 显示 JIT 编译的相关信息；
- gc : 显示和 gc 相关的堆信息；
- gccapacity : 显示各个代的容量以及使用情况；
- gcmetacapacity : 显示 Metaspace 的大小；
- gcnew : 显示新生代信息；
- gcnewcapacity : 显示新生代大小和使用情况；
- gcold : 显示老年代和永久代的信息；
- gcoldcapacity : 显示老年代的大小；
- gcutil : 显示垃圾收集信息；
- gccause : 显示垃圾回收的相关信息（通 -gcutil），同时显示最后一次或当前正在发生的垃圾回收的诱因；
- printcompilation : 输出 JIT 编译的方法信息。

它的功能比较多，在这里我例举一个常用功能，如何使用 jstat 查看堆内存的使用情况。我们可以用 jstat -gc pid 查看：

```
[root@localhost conf]# jstat -gc 28557
SOC   S1C    S0U    S1U    EC      EU     OC      OU      MC      MU      CCSC    CCSU    YGC    111    YGCT    FGC      FGCT      GCT
512.0 512.0  0.0  176.0 64512.0 41367.8 128512.0 63272.4 67200.0 64600.4 8320.0 7832.7 111  1.266  3  0.227  1.493
[root@localhost conf]#
```

- S0C : 年轻代中 To Survivor 的容量（单位 KB）；
- S1C : 年轻代中 From Survivor 的容量（单位 KB）；
- S0U : 年轻代中 To Survivor 目前已使用空间（单位 KB）；
- S1U : 年轻代中 From Survivor 目前已使用空间（单位 KB）；

EC : 年轻代中 Eden 的容量 (单位 KB) ;

EU : 年轻代中 Eden 目前已使用空间 (单位 KB) ;

OC : Old 代的容量 (单位 KB) ;

OU : Old 代目前已使用空间 (单位 KB) ;

MC : Metaspace 的容量 (单位 KB) ;

MU : Metaspace 目前已使用空间 (单位 KB) ;

YGC : 从应用程序启动到采样时年轻代中 gc 次数 ;

YGCT : 从应用程序启动到采样时年轻代中 gc 所用时间 (s) ;

FGC : 从应用程序启动到采样时 old 代 (全 gc) gc 次数 ;

FGCT : 从应用程序启动到采样时 old 代 (全 gc) gc 所用时间 (s) ;

GCT : 从应用程序启动到采样时 gc 用的总时间 (s)。

JDK 工具之 jstack 命令

这个工具在模块三的[答疑课堂](#)中介绍过，它是一种线程堆栈分析工具，最常用的功能就是使用 jstack pid 命令查看线程的堆栈信息，通常会结合 top -Hp pid 或 pidstat -p pid -t 一起查看具体线程的状态，也经常用来排查一些死锁的异常。

```
"Catalina-utility-1" #12 prio=1 os_prio=0 tid=0x00007f17e524e000 nid=0x64a waiting on condition [0x00007f17d11bb000]
  java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for  <0x00000000c062fac8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)
    at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:1088)
    at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:809)
    at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1074)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
    at java.lang.Thread.run(Thread.java:748)
```

每个线程堆栈的信息中，都可以查看到线程 ID、线程的状态 (wait、sleep、running 等状态) 以及是否持有锁等。

JDK 工具之 jmap 命令

在[第 23 讲](#)中我们使用过 jmap 查看堆内存初始化配置信息以及堆内存的使用情况。那么除了这个功能，我们其实还可以使用 jmap 输出堆内存中的对象信息，包括产生了哪些对象，对象数量多少等。

我们可以用 jmap 来查看堆内存初始化配置信息以及堆内存的使用情况：

```
[root@localhost conf]# jmap -heap 28557
Attaching to process ID 28557, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
MinHeapFreeRatio          = 0
MaxHeapFreeRatio           = 100
MaxHeapSize                = 2051014656 (1956.0MB)
NewSize                    = 42991616 (41.0MB)
MaxNewSize                 = 683671552 (652.0MB)
OldSize                    = 87031808 (83.0MB)
NewRatio                   = 2
SurvivorRatio              = 8
MetaspaceSize              = 21807104 (20.796875MB)
CompressedClassSpaceSize   = 1073741824 (1024.0MB)
MaxMetaspaceSize           = 17592186044415 MB
G1HeapRegionSize           = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
capacity = 65011712 (62.0MB)
used     = 1833000 (1.7480850219726562MB)
free     = 63178712 (60.251914978027344MB)
2.819491970923639% used
From Space:
capacity = 524288 (0.5MB)
used     = 0 (0.0MB)
free     = 524288 (0.5MB)
0.0% used
To Space:
capacity = 524288 (0.5MB)
used     = 0 (0.0MB)
free     = 524288 (0.5MB)
0.0% used
PS old Generation
capacity = 168296448 (160.5MB)
used     = 43367672 (41.35863494873047MB)
free     = 124928776 (119.14136505126953MB)
25.76861990575107% used
```

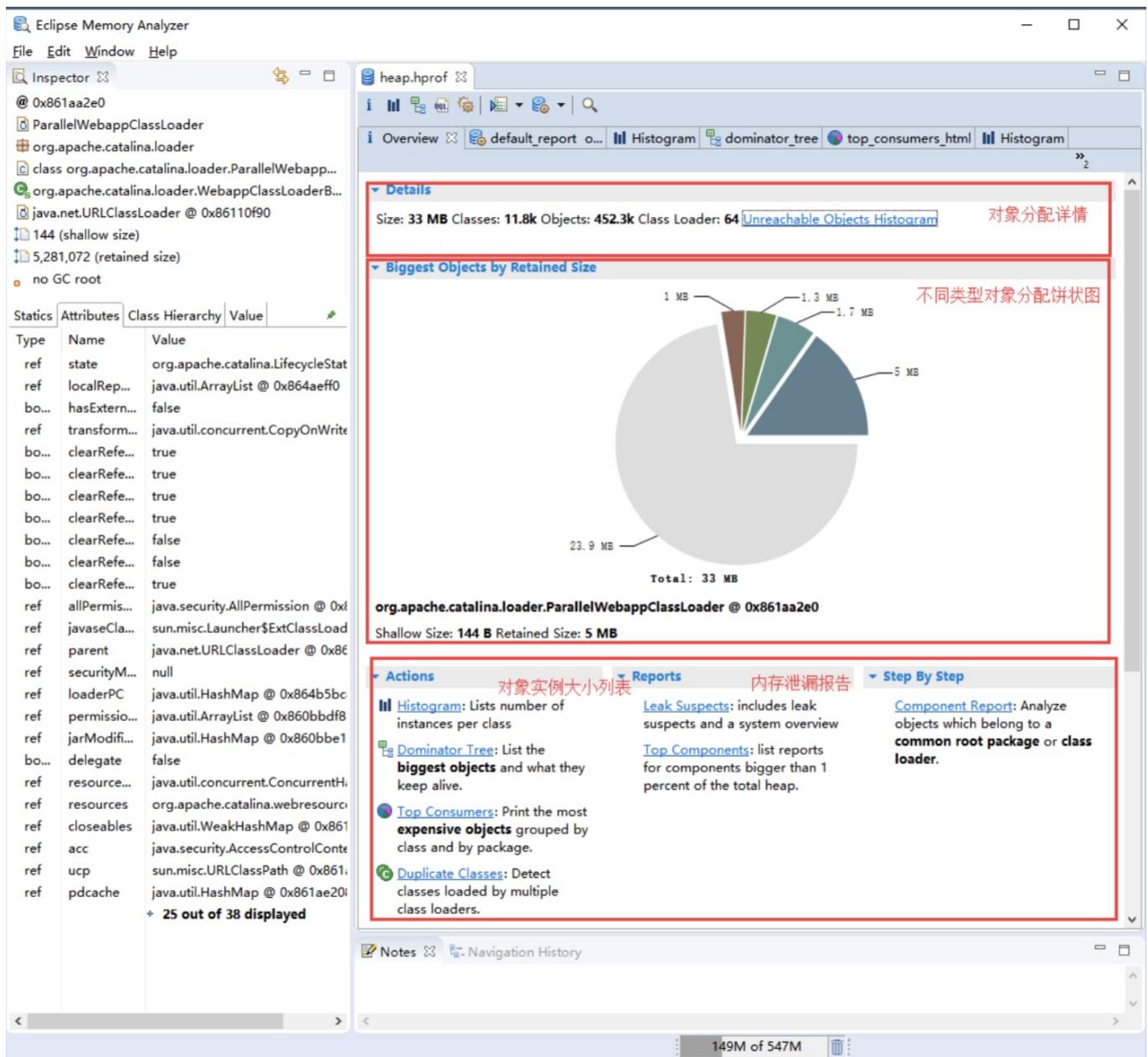
我们可以使用 jmap -histo[:live] pid 查看堆内存中的对象数目、大小统计直方图，如果带上 live 则只统计活对象：

num	#instances	#bytes	class name
1:	83066	14996568	[C
2:	13371	7238584	[B
3:	81282	1950768	java.lang.String
4:	18414	1620432	java.lang.reflect.Method
5:	12487	1385408	java.lang.Class
6:	42827	1370464	java.util.concurrent.ConcurrentHashMap\$Node
7:	18995	1097368	[Ljava.lang.Object;
8:	7945	742904	[I
9:	46134	738144	java.lang.Object
10:	6322	708064	java.net.SocksSocketImpl
11:	18695	598240	java.util.HashMap\$Node
12:	5948	526928	[Ljava.util.HashMap\$Node;
13:	12708	508320	java.util.LinkedHashMap\$Entry
14:	266	375600	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
15:	7658	367584	org.aspectj.weaver.reflect.ShadowMatchImpl
16:	6438	360528	java.util.LinkedHashMap
17:	6316	303168	java.net.SocketInputStream
18:	6316	303168	java.net.SocketOutputStream
19:	12359	296616	java.util.ArrayList
20:	12952	290912	[Ljava.lang.Class;
21:	6797	271880	java.lang.ref.Finalizer
22:	7658	245056	org.aspectj.weaver.patterns.Exposedstate
23:	9426	226224	org.apache.catalina.loader.ResourceEntry
24:	6526	208832	java.net.InetAddress\$InetAddressHolder
25:	6441	206112	java.io.FileDescriptor
26:	4153	205656	[Ljava.lang.String;
27:	6317	202144	java.net.Socket
28:	3821	183408	org.apache.tomcat.util.buf.ByteChunk
29:	3417	164016	org.apache.tomcat.util.buf.CharChunk
30:	3317	159216	org.apache.tomcat.util.buf.MessageBytes
31:	6522	156528	java.net.Inet4Address
32:	2913	139824	java.util.HashMap

我们可以通过 jmap 命令把堆内存的使用情况 dump 到文件中：

```
[root@localhost conf]#
[root@localhost conf]# jmap -dump:format=b,file=/tmp/heap.hprof 28557
Dumping heap to /tmp/heap.hprof ...
Heap dump file created
[root@localhost conf]#
```

我们可以将文件下载下来，使用 [MAT](#) 工具打开文件进行分析：



下面我们用一个实战案例来综合使用下刚刚介绍的几种工具，具体操作一下如何分析一个内存泄漏问题。

实战演练

我们平时遇到的内存溢出问题一般分为两种，一种是由于大峰值下没有限流，瞬间创建大量对象而导致的内存溢出；另一种则是由于内存泄漏而导致的内存溢出。

使用限流，我们一般就可以解决第一种内存溢出问题，但其实很多时候，内存溢出往往是内存泄漏导致的，这种问题就是程序的 BUG，我们需要及时找到问题代码。

下面我模拟了一个内存泄漏导致的内存溢出案例，我们来实践一下。

我们知道，ThreadLocal 的作用是提供线程的私有变量，这种变量可以在一个线程的整个生命周期中传递，可以减少一个线程在多个函数或类中创建公共变量来传递信息，避免了复杂度。但在使用时，如果 ThreadLocal 使用不恰当，就可能导致内存泄漏。

这个案例的场景就是 ThreadLocal，下面我们创建 100 个线程。运行以下代码，系统一会儿就发送了内存溢出异常：

 复制代码

```
1 final static ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(100, 100, 1, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());// 创建线程池，通过线程池，保证创建的线程存活
2
3
4     final static ThreadLocal<Byte[]> localVariable = new ThreadLocal<Byte[]>();// 声明一个线程局部变量
5
6     @RequestMapping(value = "/test0")
7     public String test0(HttpServletRequest request) {
8         poolExecutor.execute(new Runnable() {
9             public void run() {
10                 Byte[] c = new Byte[4096*1024];
11                 localVariable.set(c);// 为线程添加变量
12
13             }
14         });
15         return "success";
16     }
17
18     @RequestMapping(value = "/test1")
19     public String test1(HttpServletRequest request) {
20         List<Byte[]> temp1 = new ArrayList<Byte[]>();
21
22         Byte[] b = new Byte[1024*20];
23         temp1.add(b);// 添加局部变量
24
25         return "success";
26     }
}
```

在启动应用程序之前，我们可以通过 HeapDumpOnOutOfMemoryError 和 HeapDumpPath 这两个参数开启堆内存异常日志，通过以下命令启动应用程序：

 复制代码

```
1 java -jar -Xms1000m -Xmx4000m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof
```

首先，请求 test0 链接 10000 次，之后再请求 test1 链接 10000 次，这个时候我们请求 test1 的接口报异常了。

```
2019-06-27 11:41:37.508 INFO 32645 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-06-27 11:41:37.818 INFO 32645 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8011 (http) with context th ''
2019-06-27 11:41:37.822 INFO 32645 --- [main] com.demo.heapTest.App : Started App in 3.006 seconds (JVM running for 3.468)
2019-06-27 11:42:23.218 INFO 32645 --- [nio-8011-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing DispatcherServlet 'dispatcherServlet'
2019-06-27 11:42:23.226 INFO 32645 --- [nio-8011-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 8 ms
java.lang.OutOfMemoryError: Java heap space
Dumping heap to /usr/local/tomcat/heapdump.hprof...
Heap dump file created [1839744103 bytes in 4.027 secs]
Exception in thread "pool-1-thread-27" java.lang.OutOfMemoryError: Java heap space
at com.demo.heapTest.AuthorizeController$1.run(AuthorizeController.java:32)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
Exception in thread "pool-1-thread-32" java.lang.OutOfMemoryError: Java heap space
at com.demo.heapTest.AuthorizeController$1.run(AuthorizeController.java:32)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
Exception in thread "pool-1-thread-29" java.lang.OutOfMemoryError: Java heap space
at com.demo.heapTest.AuthorizeController$1.run(AuthorizeController.java:32)
```

通过日志，我们很好分辨这是一个内存溢出异常。我们首先通过 Linux 系统命令查看进程在整个系统中内存的使用率是多少，最简单就是 top 命令了。

```
[root@localhost ~]# top
top - 14:03:55 up 171 days, 46 min, 5 users, load average: 1.10, 0.37, 0.16
Tasks: 136 total, 1 running, 130 sleeping, 5 stopped, 0 zombie
%CPU(s): 59.6 us, 0.2 sy, 0.0 ni, 40.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 8010548 total, 159672 free, 4968440 used, 2882436 buff/cache
KiB Swap: 4194300 total, 4184712 free, 9588 used. 2442524 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR   S %CPU %MEM     TIME+ COMMAND
 1444 root      20   0  8017836  4.2g 13520  S 239.3 55.5  2:47.66 java
28702 root      20   0  790740  70904 25816  S 0.3  0.9 549:54.43 dockerd
  1 root      20   0  191108  3772 2356  S 0.0  0.0 2:28.92 systemd
  2 root      20   0      0      0  0  S 0.0  0.0 0:00.04 kthreadd
  3 root      20   0      0      0  0  S 0.0  0.0 4:04.86 ksoftirqd/0
  5 root      0 -20      0      0  0  S 0.0  0.0 0:00.00 kworker/0:0H
  6 root      20   0      0      0  0  S 0.0  0.0 11:00.21 kworker/u8:0
  7 root      rt  0      0      0  0  S 0.0  0.0 0:02.25 migration/0
  8 root      20   0      0      0  0  S 0.0  0.0 0:00.00 rcu_bh
  9 root      20   0      0      0  0  S 0.0  0.0 88:28.25 rcu_sched
 10 root      rt  0      0      0  0  S 0.0  0.0 1:53.36 watchdog/0
 11 root      rt  0      0      0  0  S 0.0  0.0 1:46.63 watchdog/1
 12 root      rt  0      0      0  0  S 0.0  0.0 0:01.88 migration/1
 13 root      20   0      0      0  0  S 0.0  0.0 0:04.23 ksoftirqd/1
 15 root      0 -20      0      0  0  S 0.0  0.0 0:00.00 kworker/1:0H
 16 root      rt  0      0      0  0  S 0.0  0.0 1:44.16 watchdog/2
 17 root      rt  0      0      0  0  S 0.0  0.0 0:01.79 migration/2
 18 root      20   0      0      0  0  S 0.0  0.0 0:01.62 ksoftirqd/2
 20 root      0 -20      0      0  0  S 0.0  0.0 0:00.00 kworker/2:0H
 21 root      rt  0      0      0  0  S 0.0  0.0 1:46.23 watchdog/3
 22 root      rt  0      0      0  0  S 0.0  0.0 0:01.73 migration/3
 23 root      20   0      0      0  0  S 0.0  0.0 0:02.73 ksoftirqd/3
 25 root      0 -20      0      0  0  S 0.0  0.0 0:00.00 kworker/3:0H
```

从 top 命令查看进程的内存使用情况，可以发现在机器只有 8G 内存且只分配了 4G 内存给 Java 进程的情况下，Java 进程内存使用率已经达到了 55%，再通过 top -Hp pid 查看具体线程占用系统资源情况。

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1593	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1594	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:02.97	java
1595	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:18.27	java
1596	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:18.30	java
1597	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:18.01	java
1598	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:18.50	java
1599	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:17.02	java
1600	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1601	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1602	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1603	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:03.21	java
1604	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:03.44	java
1605	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:01.58	java
1606	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1607	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:01.65	java
1610	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.22	java
1611	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.25	java
1612	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.00	java
1613	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.13	java
1614	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.16	java
1615	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.02	java
1616	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.01	java
1617	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.01	java
1618	root	20	0	4761292	1.2g	13512	S	0.0	15.1	0:00.01	java

再通过 jstack pid 查看具体线程的堆栈信息，可以发现该线程一直处于 TIMED_WAITING 状态，此时 CPU 使用率和负载并没有出现异常，我们可以排除死锁或 I/O 阻塞的异常问题了。

```

at java.lang.Thread.run(Thread.java:748)
"nioBlockingSelector.BlockPoller-1" #15 daemon prio=5 os_prio=0 tid=0x00007f17e5012800 nid=0x64d runnable [0x00007f17ca7ec000]
  java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
    at sun.nio.ch.EPollSelectorImpl.doselect(EPollSelectorImpl.java:93)
    at sun.nio.ch.SelectorImpl.lockAndDoselect(SelectorImpl.java:86)
    - locked <0x00000000c8d9b9f8> (a sun.nio.ch.Util$3)
    - locked <0x00000000c8d9b9e8> (a java.util.Collections$UnmodifiableSet)
    - locked <0x00000000c8d9b8c0> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at org.apache.tomcat.util.net.NioBlockingSelector$BlockPoller.run(NioBlockingSelector.java:304)

```

我们再通过 jmap 查看堆内存的使用情况，可以发现，老年代的使用率几乎快占满了，而且内存一直得不到释放：

```
[root@localhost ~]# jmap -heap 1593
Attaching to process ID 1593, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
MinHeapFreeRatio          = 0
MaxHeapFreeRatio          = 100
MaxHeapSize                = 1073741824 (1024.0MB)
NewSize                    = 357564416 (341.0MB)
MaxNewSize                 = 357564416 (341.0MB)
Oldsize                    = 716177408 (683.0MB)
NewRatio                   = 2
SurvivorRatio              = 8
Metaspacesize              = 21807104 (20.796875MB)
CompressedClassSpaceSize   = 1073741824 (1024.0MB)
MaxMetaspaceSize           = 17592186044415 MB
G1HeapRegionSize            = 0 (0.0MB)

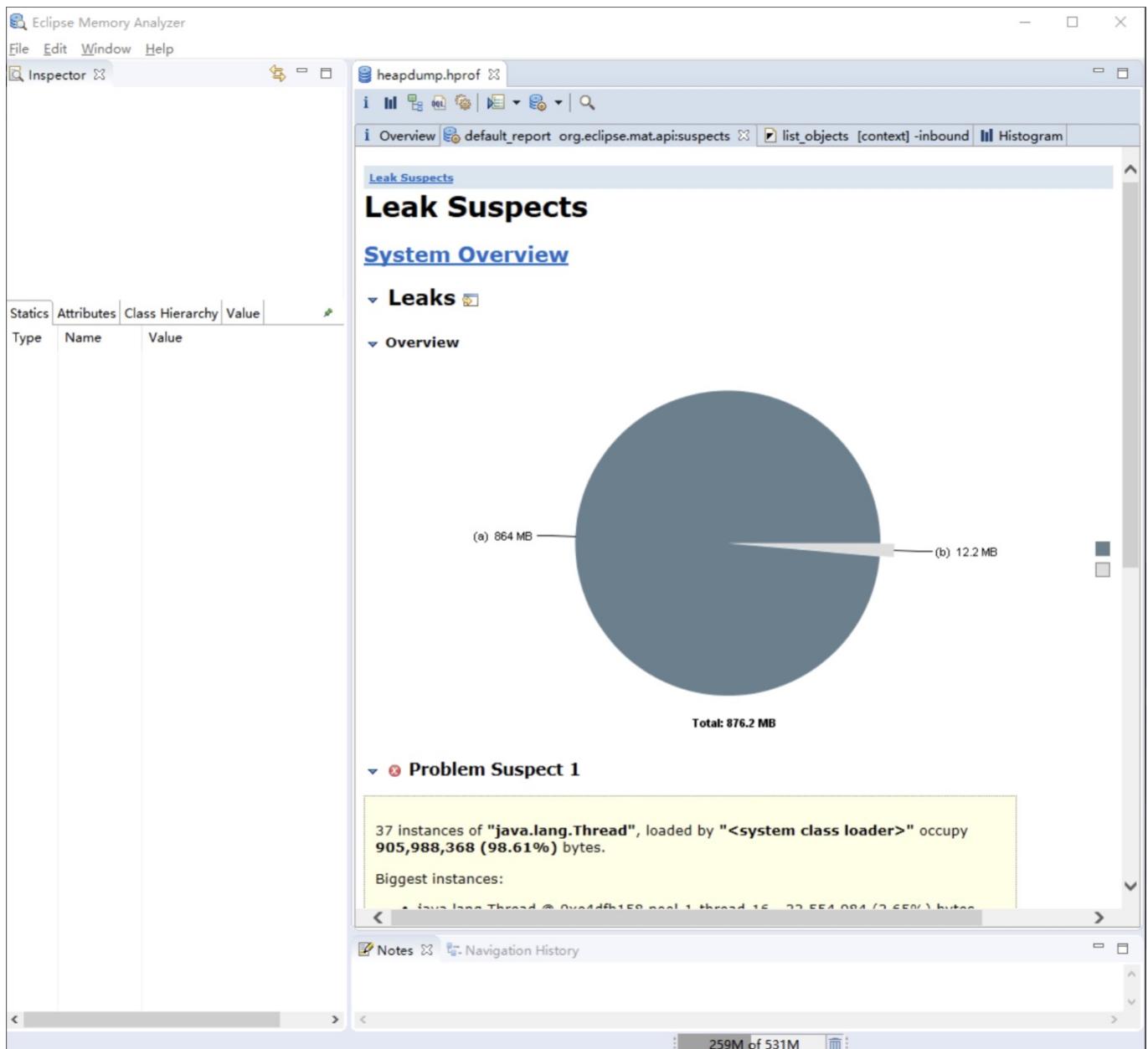
Heap Usage:
PS Young Generation
Eden Space:
capacity = 268435456 (256.0MB)
used     = 247003168 (235.56057739257812MB)
free     = 21432288 (20.439422607421875MB)
92.01585054397583% used
From Space:
capacity = 36175872 (34.5MB)
used     = 0 (0.0MB)
free     = 36175872 (34.5MB)
0.0% used
To Space:
capacity = 52428800 (50.0MB)
used     = 0 (0.0MB)
free     = 52428800 (50.0MB)
0.0% used
PS old Generation
capacity = 716177408 (683.0MB)
used     = 685996376 (654.2171249389648MB)
free     = 30181032 (28.782875061035156MB)
95.78581624289383% used
```

通过以上堆内存的情况，我们基本可以判断系统发生了内存泄漏。下面我们就需要找到具体是什么对象一直无法回收，什么原因导致了内存泄漏。

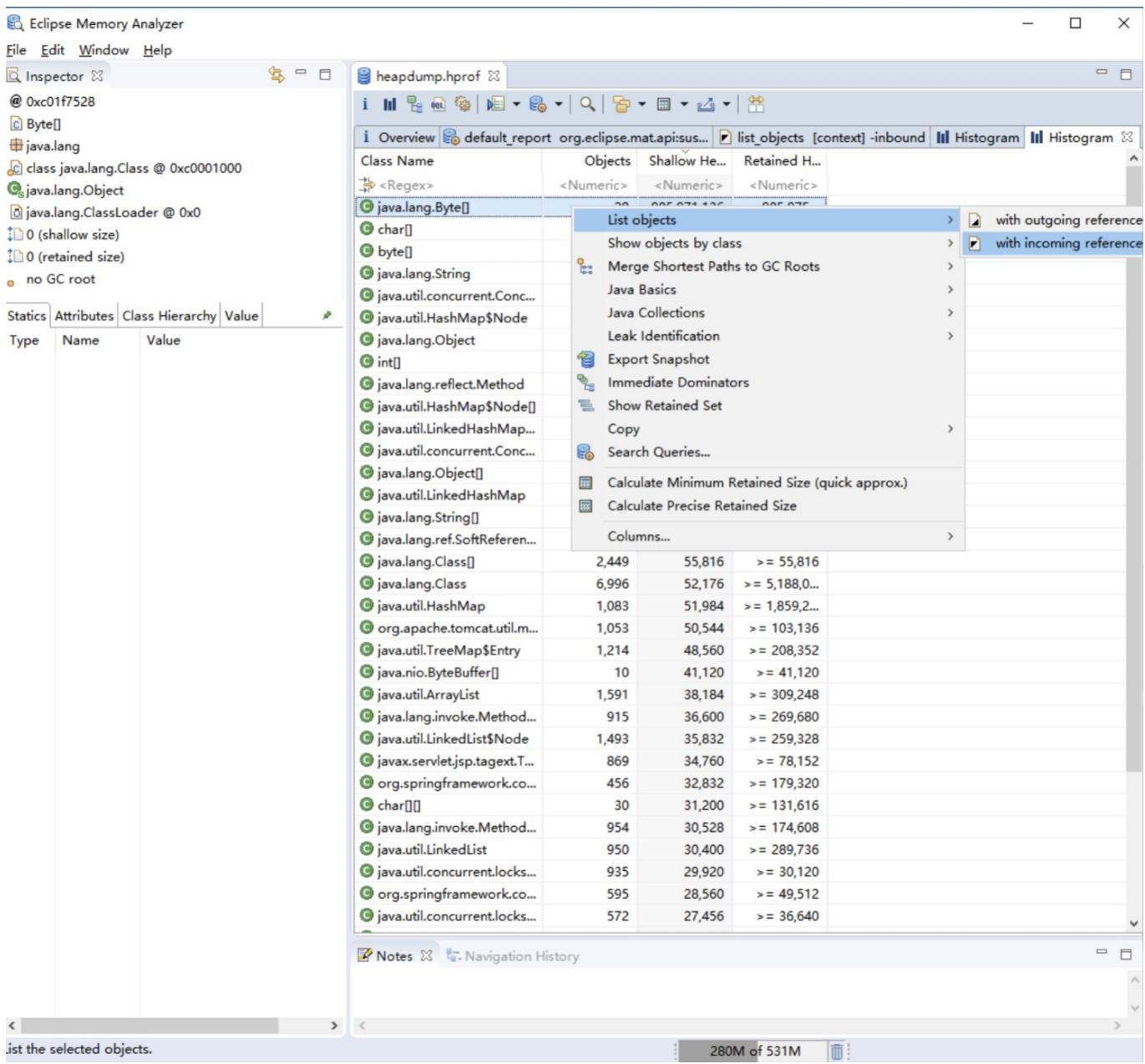
我们需要查看具体的堆内存对象，看看是哪个对象占用了堆内存，可以通过 jstat 查看存活对象的数量：

num	#instances	#bytes	class name
1:	28	905971136	[Ljava.lang.Byte;
2:	37282	6869656	[C
3:	1488	935304	[B
4:	36312	871488	java.lang.String
5:	7505	832472	java.lang.Class
6:	19401	620832	java.util.concurrent.ConcurrentHashMap\$Node
7:	7052	389152	[Ljava.lang.Object;
8:	10071	322272	java.util.HashMap\$Node
9:	2932	264944	[I
10:	2951	258320	[Ljava.util.HashMap\$Node;
11:	6201	248040	java.util.LinkedHashMap\$Entry
12:	2800	246400	java.lang.reflect.Method
13:	14859	237744	java.lang.Object
14:	121	190864	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
15:	2665	149240	java.util.LinkedHashMap
16:	1420	83200	[Ljava.lang.String;
17:	1737	69480	java.lang.ref.SoftReference
18:	2660	63840	java.util.ArrayList
19:	2877	63320	[Ljava.lang.Class;
20:	1255	60240	java.util.HashMap
21:	961	53816	java.lang.invoke.MemberName
22:	642	51360	java.lang.reflect.Constructor
23:	1053	50544	org.apache.tomcat.util.modeler.AttributeInfo
24:	1214	48560	java.util.TreeMap\$Entry
25:	110	41360	java.lang.Thread

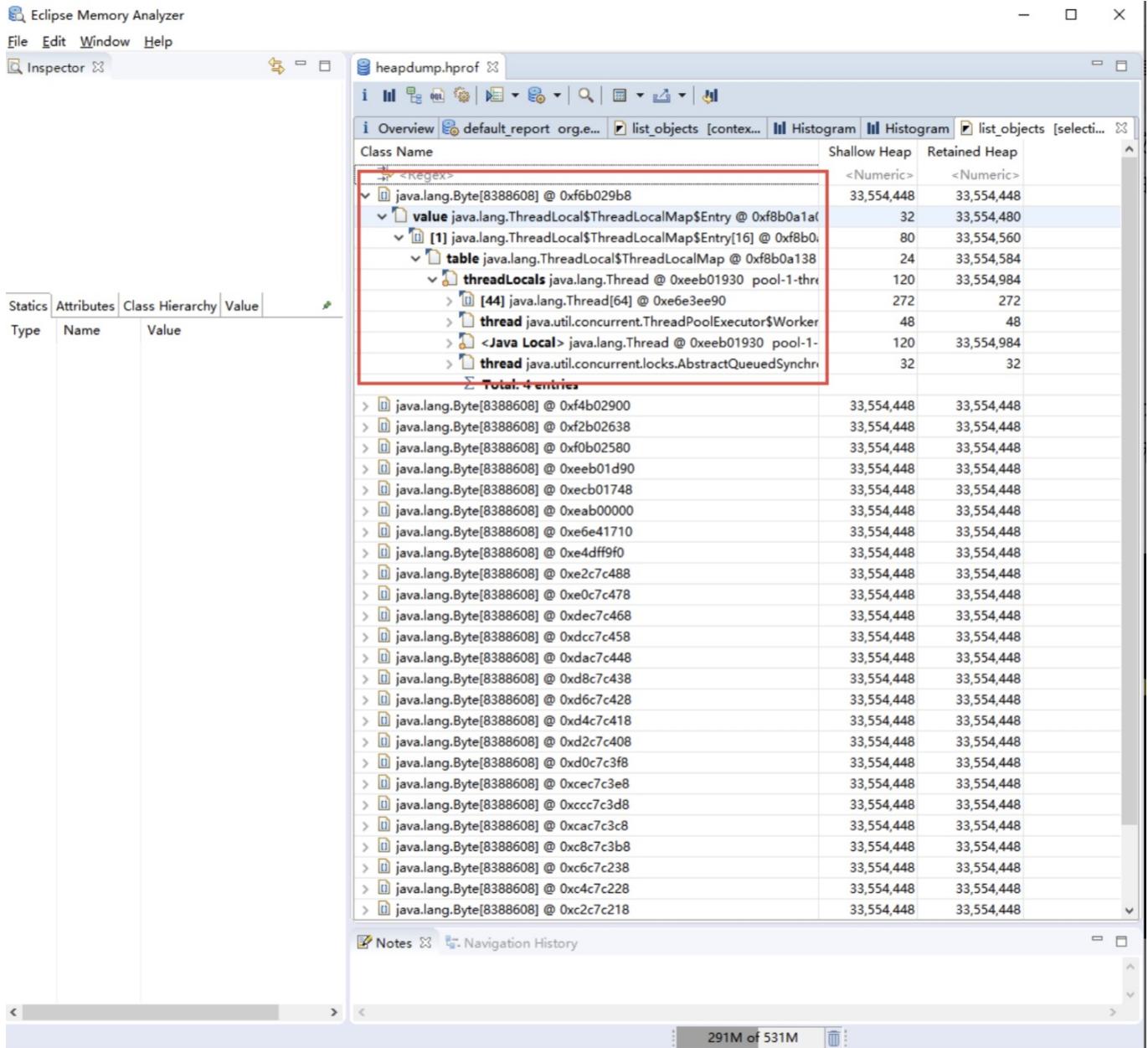
Byte 对象占用内存明显异常，说明代码中 Byte 对象存在内存泄漏，我们在启动时，已经设置了 dump 文件，通过 MAT 打开 dump 的内存日志文件，我们可以发现 MAT 已经提示了 byte 内存异常：



再点击进入到 Histogram 页面，可以查看到对象数量排序，我们可以看到 Byte[] 数组排在了第一位，选中对象后右击选择 with incomming reference 功能，可以查看到具体哪个对象引用了这个对象。



在这里我们就可以很明显地查看到是 ThreadLocal 这块的代码出现了问题。



总结

在一些比较简单的业务场景下，排查系统性能问题相对来说简单，且容易找到具体原因。但在一些复杂的业务场景下，或是一些开源框架下的源码问题，相对来说就很难排查了，有时候通过工具只能猜测到可能是某些地方出现了问题，而实际排查则要结合源码做具体分析。

可以说没有捷径，排查线上的性能问题本身不是一件很简单的事情，除了将今天介绍的这些工具融会贯通，还需要我们不断地去累积经验，真正做到性能调优。

26 | 单例模式：如何创建单一对象优化系统性能？



从这一讲开始，我们将一起探讨设计模式的性能调优。在《Design Patterns: Elements of Reusable Object-Oriented Software》一书中，有 23 种设计模式的描述，其中，单例设计模式是最常用的设计模式之一。无论是在开源框架，还是在我们的日常开发中，单例模式几乎无处不在。

什么是单例模式？

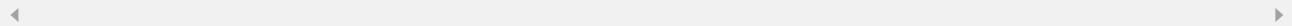
它的核心在于，单例模式可以保证一个类仅创建一个实例，并提供一个访问它的全局访问点。

该模式有三个基本要点：一是这个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

结合这三点，我们来实现一个简单的单例：

 复制代码

```
1 // 饿汉模式
2 public final class Singleton {
3     private static Singleton instance=new Singleton();// 自行创建实例
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 通过该函数向整个系统提供实例
6         return instance;
7     }
8 }
```



由于在一个系统中，一个类经常会被使用在不同的地方，**通过单例模式，我们可以避免多次创建多个实例，从而节约系统资源。**

饿汉模式

我们可以发现，以上第一种实现单例的代码中，使用了 `static` 修饰了成员变量 `instance`，所以该变量会在类初始化的过程中被收集进类构造器即 `<clinit>` 方法中。在多线程场景下，JVM 会保证只有一个线程能执行该类的 `<clinit>` 方法，其它线程将会被阻塞等待。

等到唯一的一次 `<clinit>` 方法执行完成，其它线程将不会再执行 `<clinit>` 方法，转而执行自己的代码。也就是说，`static` 修饰了成员变量 `instance`，在多线程的情况下能保证只实例化一次。

这种方式实现的单例模式，在类加载阶段就已经在堆内存中开辟了一块内存，用于存放实例化对象，所以也称为**饿汉模式**。

饿汉模式实现的单例的优点是，可以保证多线程情况下实例的唯一性，而且 `getInstance` 直接返回唯一实例，性能非常高。

然而，在类成员变量比较多，或变量比较大的情况下，这种模式可能会在没有使用类对象的情况下，一直占用堆内存。试想下，如果一个第三方开源框架中的类都是基于饿汉模式实现

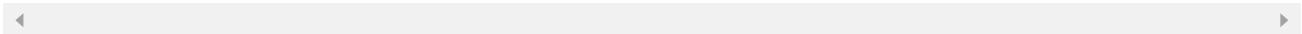
的单例，这将会初始化所有单例类，无疑是灾难性的。

懒汉模式

懒汉模式就是为了避免直接加载类对象时提前创建对象的一种单例设计模式。该模式使用懒加载方式，只有当系统使用到类对象时，才会将实例加载到堆内存中。通过以下代码，我们可以简单地了解下懒加载的实现方式：

 复制代码

```
1 // 懒汉模式
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 通过该函数向整个系统提供实例
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             instance = new Singleton();// 实例化对象
8         }
9         return instance;// 返回已存在的对象
10    }
11 }
```



以上代码在单线程下运行是没有问题的，但要运行在多线程下，就会出现实例化多个类对象的情况。这是怎么回事呢？

当线程 A 进入到 if 判断条件后，开始实例化对象，此时 instance 依然为 null；又有线程 B 进入到 if 判断条件中，之后也会通过条件判断，进入到方法里面创建一个实例对象。

所以我们需要对该方法进行加锁，保证多线程情况下仅创建一个实例。这里我们使用 Synchronized 同步锁来修饰 getInstance 方法：

 复制代码

```
1 // 懒汉模式 + synchronized 同步锁
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static synchronized Singleton getInstance(){// 加同步锁，通过该函数向整个系统提
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             instance = new Singleton();// 实例化对象
8         }
9         return instance;// 返回已存在的对象
}
```

```
10     }
11 }
```

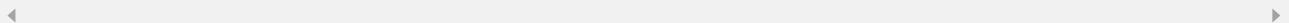


但我们前面讲过，同步锁会增加锁竞争，带来系统性能开销，从而导致系统性能下降，因此这种方式也会降低单例模式的性能。

还有，每次请求获取类对象时，都会通过 `getInstance()` 方法获取，除了第一次为 `null`，其它每次请求基本都是不为 `null` 的。在没有加同步锁之前，是因为 `if` 判断条件为 `null` 时，才导致创建了多个实例。基于以上两点，我们可以考虑将同步锁放在 `if` 条件里面，这样就可以减少同步锁资源竞争。

复制代码

```
1 // 懒汉模式 + synchronized 同步锁
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             synchronized (Singleton.class){
8                 instance = new Singleton();// 实例化对象
9             }
10        }
11        return instance;// 返回已存在的对象
12    }
13 }
```



看到这里，你是不是觉得这样就可以了呢？答案是依然会创建多个实例。这是因为当多个线程进入到 `if` 判断条件里，虽然有同步锁，但是进入到判断条件里面的线程依然会依次获取到锁创建对象，然后再释放同步锁。所以我们还需要在同步锁里面再加一个判断条件：

复制代码

```
1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
6         if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返
7             synchronized (Singleton.class){// 同步锁
8                 if(null == instance){// 第二次判断
9                     instance = new Singleton();
10                }
11            }
12        }
13        return instance;
14    }
15 }
```

```
9         instance = new Singleton()// 实例化对象
10    }
11   }
12 }
13 return instance;// 返回已存在的对象
14 }
15 }
```



以上这种方式，通常被称为 Double-Check，它可以大大提高支持多线程的懒汉模式的运行性能。那这样做是不是就能保证万无一失了呢？还会有什么问题吗？

其实这里又跟 Happens-Before 规则和重排序扯上关系了，这里我们先来简单了解下 Happens-Before 规则和重排序。

我们在第二期[加餐](#)中分享过，编译器为了尽可能地减少寄存器的读取、存储次数，会充分复用寄存器的存储值，比如以下代码，如果没有进行重排序优化，正常的执行顺序是步骤 1\2\3，而在编译期间进行了重排序优化之后，执行的步骤有可能就变成了步骤 1/3/2，这样就能减少一次寄存器的存取次数。

复制代码

```
1 int a = 1;// 步骤 1: 加载 a 变量的内存地址到寄存器中, 加载 1 到寄存器中, CPU 通过 mov 指令把
2 int b = 2;// 步骤 2 加载 b 变量的内存地址到寄存器中, 加载 2 到寄存器中, CPU 通过 mov 指令把 2
3 a = a + 1;// 步骤 3 重新加载 a 变量的内存地址到寄存器中, 加载 1 到寄存器中, CPU 通过 mov 指令
```



在 JMM 中，重排序是十分重要的一环，特别是在并发编程中。如果 JVM 可以对它们进行任意排序以提高程序性能，也可能会给并发编程带来一系列的问题。例如，我上面讲到的 Double-Check 的单例问题，假设类中有其它的属性也需要实例化，这个时候，除了要实例化单例类本身，还需要对其它属性也进行实例化：

复制代码

```
1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     public List<String> list = null;//list 属性
5     private Singleton(){
6         list = new ArrayList<String>();
7     }// 构造函数
```

```
8     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
9         if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返回
10            synchronized (Singleton.class){// 同步锁
11                if(null == instance){// 第二次判断
12                    instance = new Singleton();// 实例化对象
13                }
14            }
15        }
16        return instance;// 返回已存在的对象
17    }
18 }
```



在执行 `instance = new Singleton();` 代码时，正常情况下，实例过程这样的：

给 `Singleton` 分配内存；

调用 `Singleton` 的构造函数来初始化成员变量；

将 `Singleton` 对象指向分配的内存空间（执行完这步 `singleton` 就为非 `null` 了）。

如果虚拟机发生了重排序优化，这个时候步骤 3 可能发生在步骤 2 之前。如果初始化线程刚好完成步骤 3，而步骤 2 没有进行时，则刚好有另一个线程到了第一次判断，这个时候判断为非 `null`，并返回对象使用，这个时候实际没有完成其它属性的构造，因此使用这个属性就很可能会导致异常。在这里，`Synchronized` 只能保证可见性、原子性，无法保证执行的顺序。

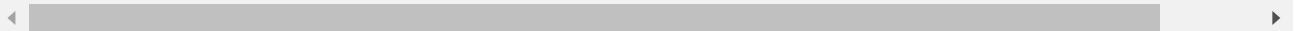
这个时候，就体现出 Happens-Before 规则的重要性了。通过字面意思，你可能会误以为是前一个操作发生在后一个操作之前。然而真正的意思是，前一个操作的结果可以被后续的操作获取。这条规则规范了编译器对程序的重排序优化。

我们知道 `volatile` 关键字可以保证线程间变量的可见性，简单地说就是当线程 A 对变量 X 进行修改后，在线程 A 后面执行的其它线程就能看到变量 X 的变动。除此之外，`volatile` 在 JDK1.5 之后还有一个作用就是阻止局部重排序的发生，也就是说，`volatile` 变量的操作指令都不会被重排序。所以使用 `volatile` 修饰 `instance` 之后，Double-Check 懒汉单例模式就万无一失了。

复制代码

```
1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {
```

```
3 private volatile static Singleton instance= null;// 不实例化
4 public List<String> list = null;//list 属性
5 private Singleton(){
6     list = new ArrayList<String>();
7 } // 构造函数
8 public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
9     if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返回
10         synchronized (Singleton.class){// 同步锁
11             if(null == instance){// 第二次判断
12                 instance = new Singleton();// 实例化对象
13             }
14         }
15     }
16     return instance;// 返回已存在的对象
17 }
18 }
```



通过内部类实现

以上这种同步锁 +Double-Check 的实现方式相对来说，复杂且加了同步锁，那有没有稍微简单一点儿的可以实现线程安全的懒加载方式呢？

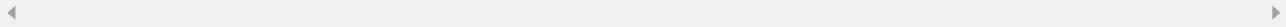
我们知道，在饿汉模式中，我们使用了 `static` 修饰了成员变量 `instance`，所以该变量会在类初始化的过程中被收集进类构造器即 `<clinit>` 方法中。在多线程场景下，JVM 会保证只有一个线程能执行该类的 `<clinit>` 方法，其它线程将会被阻塞等待。这种方式可以保证内存的可见性、顺序性以及原子性。

如果我们在 `Singleton` 类中创建一个内部类来实现成员变量的初始化，则可以避免多线程下重复创建对象的情况发生。这种方式，只有在第一次调用 `getInstance()` 方法时，才会加载 `InnerSingleton` 类，而只有在加载 `InnerSingleton` 类之后，才会实例化创建对象。具体实现如下：

复制代码

```
1 // 懒汉模式 内部类实现
2 public final class Singleton {
3     public List<String> list = null;// list 属性
4
5     private Singleton() {// 构造函数
6         list = new ArrayList<String>();
7     }
8
9     // 内部类实现
```

```
10     public static class InnerSingleton {  
11         private static Singleton instance=new Singleton();// 自行创建实例  
12     }  
13  
14     public static Singleton getInstance() {  
15         return InnerSingleton.instance;// 返回内部类中的静态变量  
16     }  
17 }
```



总结

单例的实现方式其实有很多，但总结起来就两种：饿汉模式和懒汉模式，我们可以根据自己的需求来做选择。

如果我们在程序启动后，一定会加载到类，那么用饿汉模式实现的单例简单又实用；如果我们是写一些工具类，则优先考虑使用懒汉模式，因为很多项目可能会引用到 jar 包，但未必会使用到这个工具类，懒汉模式实现的单例可以避免提前被加载到内存中，占用系统资源。

27 | 原型模式与享元模式：提升系统性能的利器



原型模式和享元模式，前者是在创建多个实例时，对创建过程的性能进行调优；后者是用减少创建实例的方式，来调优系统性能。这么看，你会不会觉得两个模式有点相互矛盾呢？

其实不然，它们的使用是分场景的。在有些场景下，我们需要重复创建多个实例，例如在循环体中赋值一个对象，此时我们就可以采用原型模式来优化对象的创建过程；而在有些场景下，我们则可以避免重复创建多个实例，在内存中共享对象就好了。

今天我们就来看看这两种模式的适用场景，了解了这些你就可以更高效地使用它们提升系统性能了。

原型模式

我们先来了解下原型模式的实现。原型模式是通过给出一个原型对象来指明所创建的对象的类型，然后使用自身实现的克隆接口来复制这个原型对象，该模式就是用这种方式来创建出更多同类型的对象。

使用这种方式创建新的对象的话，就无需再通过 new 实例化来创建对象了。这是因为 Object 类的 clone 方法是一个本地方法，它可以直接操作内存中的二进制流，所以性能相对 new 实例化来说，更佳。

实现原型模式

我们现在通过一个简单的例子来实现一个原型模式：

 复制代码

```
1 // 实现 Cloneable 接口的原型抽象类 Prototype
2 class Prototype implements Cloneable {
3     // 重写 clone 方法
4     public Prototype clone(){
5         Prototype prototype = null;
6         try{
7             prototype = (Prototype)super.clone();
8         }catch(CloneNotSupportedException e){
9             e.printStackTrace();
10        }
11        return prototype;
12    }
13 }
14 // 实现原型类
15 class ConcretePrototype extends Prototype{
16     public void show(){
17         System.out.println(" 原型模式实现类 ");
18     }
19 }
20
21 public class Client {
22     public static void main(String[] args){
23         ConcretePrototype cp = new ConcretePrototype();
24         for(int i=0; i< 10; i++){
25             ConcretePrototype clonecp = (ConcretePrototype)cp.clone();
26             clonecp.show();
27         }
28     }
29 }
```

要实现一个原型类，需要具备三个条件：

实现 Cloneable 接口：Cloneable 接口与序列化接口的作用类似，它只是告诉虚拟机可以安全地在实现了这个接口的类上使用 clone 方法。在 JVM 中，只有实现了 Cloneable 接口的类才可以被拷贝，否则会抛出 CloneNotSupportedException 异常。

重写 Object 类中的 clone 方法：在 Java 中，所有类的父类都是 Object 类，而 Object 类中有一个 clone 方法，作用是返回对象的一个拷贝。

在重写的 clone 方法中调用 super.clone()：默认情况下，类不具备复制对象的能力，需要调用 super.clone() 来实现。

从上面我们可以看出，原型模式的主要特征就是使用 clone 方法复制一个对象。通常，有些人会误以为 Object a=new Object();Object b=a; 这种形式就是一种对象复制的过程，然而这种复制只是对象引用的复制，也就是 a 和 b 对象指向了同一个内存地址，如果 b 修改了，a 的值也就跟着被修改了。

我们可以通过一个简单的例子来看看普通的对象复制问题：

 复制代码

```
1 class Student {  
2     private String name;  
3  
4     public String getName() {  
5         return name;  
6     }  
7  
8     public void setName(String name) {  
9         this.name= name;  
10    }  
11 }  
12 }  
13 public class Test {  
14  
15     public static void main(String args[]) {  
16         Student stu1 = new Student();  
17         stu1.setName("test1");  
18  
19         Student stu2 = stu1;  
20         stu1.setName("test2");  
21  
22         System.out.println(" 学生 1:" + stu1.getName());  
23         System.out.println(" 学生 2:" + stu2.getName());  
24     }  
25 }
```

```
25 }
```

 复制代码

```
1 学生 1:test1  
2 学生 2:test2
```

然而，实际上是：

 复制代码

```
1 学生 2:test2  
2 学生 2:test2
```

通过 `clone` 方法复制的对象才是真正的对象复制，`clone` 方法赋值的对象完全是一个独立的对象。刚刚讲过了，`Object` 类的 `clone` 方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。我们可以用 `clone` 方法再实现一遍以上例子。

 复制代码

```
1 // 学生类实现 Cloneable 接口  
2 class Student implements Cloneable{  
3     private String name; // 姓名  
4  
5     public String getName() {  
6         return name;  
7     }  
8  
9     public void setName(String name) {  
10        this.name= name;  
11    }  
12    // 重写 clone 方法  
13    public Student clone() {  
14        Student student = null;  
15        try {  
16            student = (Student) super.clone();  
17        } catch (CloneNotSupportedException e) {
```

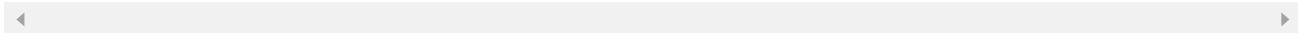
```
18     e.printStackTrace();
19 }
20     return student;
21 }
22 }
23 }
24 public class Test {
25
26     public static void main(String args[]) {
27         Student stu1 = new Student(); // 创建学生 1
28         stu1.setName("test1");
29
30         Student stu2 = stu1.clone(); // 通过克隆创建学生 2
31         stu2.setName("test2");
32
33         System.out.println(" 学生 1:" + stu1.getName());
34         System.out.println(" 学生 2:" + stu2.getName());
35     }
36 }
```



运行结果：

复制代码

```
1 学生 1:test1
2 学生 2:test2
```



深拷贝和浅拷贝

在调用 `super.clone()` 方法之后，首先会检查当前对象所属的类是否支持 `clone`，也就是看该类是否实现了 `Cloneable` 接口。

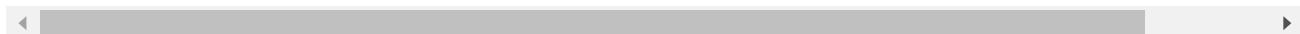
如果支持，则创建当前对象所属类的一个新对象，并对该对象进行初始化，使得新对象的成员变量的值与当前对象的成员变量的值一模一样，但对于其它对象的引用以及 `List` 等类型的成员属性，则只能复制这些对象的引用了。所以简单调用 `super.clone()` 这种克隆对象方式，就是一种浅拷贝。

所以，当我们在使用 `clone()` 方法实现对象的克隆时，就需要注意浅拷贝带来的问题。我们再通过一个例子来看看浅拷贝。

 复制代码

```
1 // 定义学生类
2 class Student implements Cloneable{
3     private String name; // 学生姓名
4     private Teacher teacher; // 定义老师类
5
6     public String getName() {
7         return name;
8     }
9
10    public void setName(String name) {
11        this.name = name;
12    }
13
14    public Teacher getTeacher() {
15        return teacher;
16    }
17
18    public void setName(Teacher teacher) {
19        this.teacher = teacher;
20    }
21 // 重写克隆方法
22    public Student clone() {
23        Student student = null;
24        try {
25            student = (Student) super.clone();
26        } catch (CloneNotSupportedException e) {
27            e.printStackTrace();
28        }
29        return student;
30    }
31
32 }
33
34 // 定义老师类
35 class Teacher implements Cloneable{
36     private String name; // 老师姓名
37
38     public String getName() {
39         return name;
40     }
41
42     public void setName(String name) {
43         this.name= name;
44     }
45
46 // 重写克隆方法，堆老师类进行克隆
47    public Teacher clone() {
48        Teacher teacher= null;
49        try {
50            teacher= (Teacher) super.clone();
51        } catch (CloneNotSupportedException e) {
```

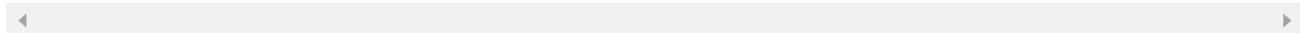
```
52         e.printStackTrace();
53     }
54     return student;
55 }
56
57 }
58 public class Test {
59
60     public static void main(String args[]) {
61         Teacher teacher = new Teacher (); // 定义老师 1
62         teacher.setName(" 刘老师 ");
63         Student stu1 = new Student(); // 定义学生 1
64         stu1.setName("test1");
65         stu1.setTeacher(teacher);
66
67         Student stu2 = stu1.clone(); // 定义学生 2
68         stu2.setName("test2");
69         stu2.getTeacher().setName(" 王老师 ");// 修改老师
70         System.out.println(" 学生 " + stu1.getName + " 的老师是:" + stu1.getTeacher().get
71         System.out.println(" 学生 " + stu1.getName + " 的老师是:" + stu2.getTeacher().get
72     }
73 }
```



运行结果：

复制代码

```
1 学生 test1 的老师是：王老师
2 学生 test2 的老师是：王老师
```



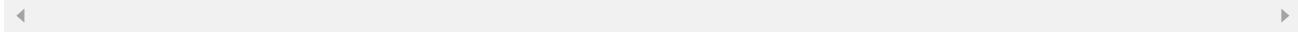
观察以上运行结果，我们可以发现：在我们给学生 2 修改老师的时候，学生 1 的老师也跟着被修改了。这就是浅拷贝带来的问题。

我们可以通过深拷贝来解决这种问题，**其实深拷贝就是基于浅拷贝来递归实现具体的每个对象**，代码如下：

复制代码

```
1     public Student clone() {
2         Student student = null;
3         try {
4             student = (Student) super.clone();
5             Teacher teacher = this.teacher.clone(); // 克隆 teacher 对象
6         }
```

```
6     student.setTeacher(teacher);
7 } catch (CloneNotSupportedException e) {
8     e.printStackTrace();
9 }
10    return student;
11 }
```



适用场景

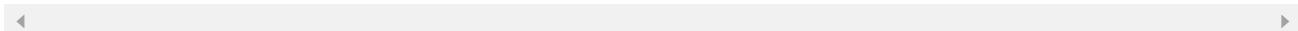
前面我详讲了原型模式的实现原理，那到底什么时候我们要用它呢？

在一些重复创建对象的场景下，我们就可以使用原型模式来提高对象的创建性能。例如，我在开头提到的，循环体内创建对象时，我们就可以考虑用 clone 的方式来实现。

例如：

复制代码

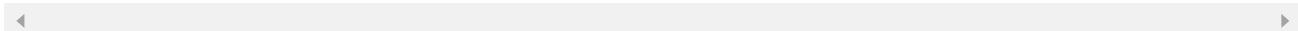
```
1 for(int i=0; i<list.size(); i++){
2     Student stu = new Student();
3     ...
4 }
5
```



我们可以优化为：

复制代码

```
1 Student stu = new Student();
2 for(int i=0; i<list.size(); i++){
3     Student stu1 = (Student)stu.clone();
4     ...
5 }
```



除此之外，原型模式在开源框架中的应用也非常广泛。例如 Spring 中，@Service 默认都是单例的。用了私有全局变量，若不想影响下次注入或每次上下文获取 bean，就需要用到原型模式，我们可以通过以下注解来实现，@Scope("prototype")。

享元模式

享元模式是运用共享技术有效地最大限度地复用细粒度对象的一种模式。该模式中，以对象的信息状态划分，可以分为内部数据和外部数据。内部数据是对象可以共享出来的信息，这些信息不会随着系统的运行而改变；外部数据则是在不同运行时被标记了不同的值。

享元模式一般可以分为三个角色，分别为 Flyweight（抽象享元类）、ConcreteFlyweight（具体享元类）和 FlyweightFactory（享元工厂类）。抽象享元类通常是一个接口或抽象类，向外界提供享元对象的内部数据或外部数据；具体享元类是指具体实现内部数据共享的类；享元工厂类则是主要用于创建和管理享元对象的工厂类。

实现享元模式

我们还是通过一个简单的例子来实现一个享元模式：

 复制代码

```
1 // 抽象享元类
2 interface Flyweight {
3     // 对外状态对象
4     void operation(String name);
5     // 对内对象
6     String getType();
7 }
```

◀ ▶

 复制代码

```
1 // 具体享元类
2 class ConcreteFlyweight implements Flyweight {
3     private String type;
4
5     public ConcreteFlyweight(String type) {
6         this.type = type;
7     }
8
9     @Override
10    public void operation(String name) {
11        System.out.printf("[类型 (内在状态)] - [%s] - [名字 (外在状态)] - [%s]\n", type, n
12    }
13
14    @Override
15    public String getType() {
16        return type;
17    }
18 }
```

```
18 }
```

复制代码

```
1 // 享元工厂类
2 class FlyweightFactory {
3     private static final Map<String, Flyweight> FLYWEIGHT_MAP = new HashMap<>(); // 享元池
4
5     public static Flyweight getFlyweight(String type) {
6         if (FLYWEIGHT_MAP.containsKey(type)) { // 如果在享元池中存在对象，则直接获取
7             return FLYWEIGHT_MAP.get(type);
8         } else { // 在响应池不存在，则新创建对象，并放入到享元池
9             ConcreteFlyweight flyweight = new ConcreteFlyweight(type);
10            FLYWEIGHT_MAP.put(type, flyweight);
11            return flyweight;
12        }
13    }
14 }
```

复制代码

```
1 public class Client {
2
3     public static void main(String[] args) {
4         Flyweight fw0 = FlyweightFactory.getFlyweight("a");
5         Flyweight fw1 = FlyweightFactory.getFlyweight("b");
6         Flyweight fw2 = FlyweightFactory.getFlyweight("a");
7         Flyweight fw3 = FlyweightFactory.getFlyweight("b");
8         fw1.operation("abc");
9         System.out.printf("[结果 (对象对比)] - [%s]\n", fw0 == fw2);
10        System.out.printf("[结果 (内在状态)] - [%s]\n", fw1.getType());
11    }
12 }
```

复制代码

输出结果：

```
1 [类型 (内在状态)] - [b] - [名字 (外在状态)] - [abc]
2 [结果 (对象对比)] - [true]
3 [结果 (内在状态)] - [b]
```

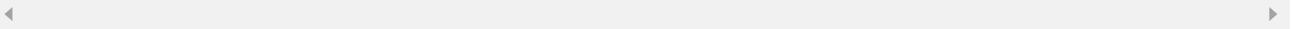
观察以上代码运行结果，我们可以发现：如果对象已经存在于享元池中，则不会再创建该对象了，而是共用享元池中内部数据一致的对象。这样就减少了对象的创建，同时也节省了同样内部数据的对象所占用的内存空间。

适用场景

享元模式在实际开发中的应用也非常广泛。例如 Java 的 String 字符串，在一些字符串常量中，会共享常量池中字符串对象，从而减少重复创建相同值对象，占用内存空间。代码如下：

 复制代码

```
1 String s1 = "hello";
2 String s2 = "hello";
3 System.out.println(s1==s2); //true
```



还有，在日常开发中的应用。例如，线程池就是享元模式的一种实现；将商品存储在应用服务的缓存中，那么每当用户获取商品信息时，则不需要每次都从 redis 缓存或者数据库中获取商品信息，并在内存中重复创建商品信息了。

总结

通过以上讲解，相信你对原型模式和享元模式已经有了更清楚的了解了。两种模式无论是在开源框架，还是在实际开发中，应用都十分广泛。

在不得已需要重复创建大量同一对象时，我们可以使用原型模式，通过 clone 方法复制对象，这种方式比用 new 和序列化创建对象的效率要高；在创建对象时，如果我们可以共用对象的内部数据，那么通过享元模式共享相同的内部数据的对象，就可以减少对象的创建，实现系统调优。

28 | 如何使用设计模式优化并发编程？



在我们使用多线程编程时，很多时候需要根据业务场景设计一套业务功能。其实，在多线程编程中，本身就存在很多成熟的功能设计模式，学好它们，用好它们，那就是如虎添翼了。今天我就带你了解几种并发编程中常用的设计模式。

线程上下文设计模式

线程上下文是指贯穿线程整个生命周期的对象中的一些全局信息。例如，我们比较熟悉的 Spring 中的 ApplicationContext 就是一个关于上下文的类，它在整个系统的生命周期中保存了配置信息、用户信息以及注册的 bean 等上下文信息。

这样的解释可能有点抽象，我们不妨通过一个具体的案例，来看看到底在什么的场景下才需要上下文呢？

在执行一个比较长的请求任务时，这个请求可能会经历很多层的方法调用，假设我们需要将最开始的方法的中间结果传递到末尾的方法中进行计算，一个简单的实现方式就是在每个函数中新增这个中间结果的参数，依次传递下去。代码如下：

 复制代码

```
1 public class ContextTest {
2
3     // 上下文类
4     public class Context {
5         private String name;
6         private long id
7
8         public long getId() {
9             return id;
10        }
11
12        public void setId(long id) {
13            this.id = id;
14        }
15
16        public String getName() {
17            return this.name;
18        }
19
20        public void setName(String name) {
21            this.name = name;
22        }
23    }
24
25    // 设置上下文名字
26    public class QueryNameAction {
27        public void execute(Context context) {
28            try {
29                Thread.sleep(1000L);
30                String name = Thread.currentThread().getName();
31                context.setName(name);
32            } catch (InterruptedException e) {
33                e.printStackTrace();
34            }
35        }
36    }
37
38    // 设置上下文 ID
39    public class QueryIdAction {
40        public void execute(Context context) {
```

```
41             try {
42                 Thread.sleep(1000L);
43                 long id = Thread.currentThread().getId();
44                 context.setId(id);
45             } catch (InterruptedException e) {
46                 e.printStackTrace();
47             }
48         }
49     }
50
51     // 执行方法
52     public class ExecutionTask implements Runnable {
53
54         private QueryNameAction queryNameAction = new QueryNameAction();
55         private QueryIdAction queryIdAction = new QueryIdAction();
56
57         @Override
58         public void run() {
59             final Context context = new Context();
60             queryNameAction.execute(context);
61             System.out.println("The name query successful");
62             queryIdAction.execute(context);
63             System.out.println("The id query successful");
64
65             System.out.println("The Name is " + context.getName() + " and id is " + context.getId());
66         }
67     }
68
69     public static void main(String[] args) {
70         IntStream.range(1, 5).forEach(i -> new Thread(new ContextTest().new ExecutionTask()).start());
71     }
72 }
```

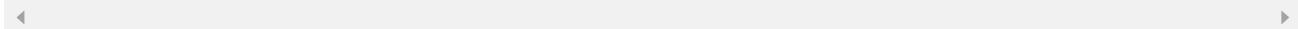


执行结果：

复制代码

```
1 The name query successful
2 The name query successful
3 The name query successful
4 The name query successful
5 The id query successful
6 The id query successful
7 The id query successful
8 The id query successful
9 The Name is Thread-1 and id 11
10 The Name is Thread-2 and id 12
11 The Name is Thread-3 and id 13
```

```
12 The Name is Thread-0 and id 10
```



然而这种方式太笨拙了，每次调用方法时，都需要传入 Context 作为参数，而且影响一些中间公共方法的封装。

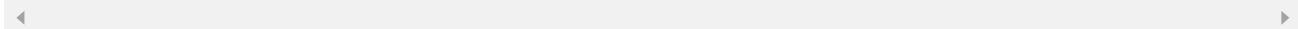
那能不能设置一个全局变量呢？如果是在多线程情况下，需要考虑线程安全，这样的话就又涉及到了锁竞争。

除了以上这些方法，其实我们还可以使用 ThreadLocal 实现上下文。ThreadLocal 是线程本地变量，可以实现多线程的数据隔离。**ThreadLocal 为每一个使用该变量的线程都提供一份独立的副本，线程间的数据是隔离的，每一个线程只能访问各自内部的副本变量。**

ThreadLocal 中有三个常用的方法：set、get、initialValue，我们可以通过以下一个简单的例子来看看 ThreadLocal 的使用：

复制代码

```
1 private void testThreadLocal() {  
2     Thread t = new Thread() {  
3         ThreadLocal<String> mStringThreadLocal = new ThreadLocal<String>();  
4  
5         @Override  
6         public void run() {  
7             super.run();  
8             mStringThreadLocal.set("test");  
9             mStringThreadLocal.get();  
10        }  
11    };  
12  
13    t.start();  
14 }
```



接下来，我们使用 ThreadLocal 来重新实现最开始的上下文设计。你会发现，我们在两个方法中并没有通过变量来传递上下文，只是通过 ThreadLocal 获取了当前线程的上下文信息：

复制代码

```
1 public class ContextTest {
```

```
2     // 上下文类
3     public static class Context {
4         private String name;
5         private long id;
6
7         public long getId() {
8             return id;
9         }
10
11        public void setId(long id) {
12            this.id = id;
13        }
14
15        public String getName() {
16            return this.name;
17        }
18
19        public void setName(String name) {
20            this.name = name;
21        }
22    }
23
24    // 复制上下文到 ThreadLocal 中
25    public final static class ActionContext {
26
27        private static final ThreadLocal<Context> threadLocal = new ThreadLocal<
28            @Override
29            protected Context initialValue() {
30                return new Context();
31            }
32        };
33
34        public static ActionContext getActionContext() {
35            return ContextHolder.actionContext;
36        }
37
38        public Context getContext() {
39            return threadLocal.get();
40        }
41
42        // 获取 ActionContext 单例
43        public static class ContextHolder {
44            private final static ActionContext actionContext = new ActionCo
45        }
46    }
47
48    // 设置上下文名字
49    public class QueryNameAction {
50        public void execute() {
51            try {
52                Thread.sleep(1000L);
53                String name = Thread.currentThread().getName();
54            }
55        }
56    }
57
58    // 打印线程名
59    public void printName() {
60        System.out.println(Thread.currentThread().getName());
61    }
62
63    // 打印线程名
64    public void printName() {
65        System.out.println(Thread.currentThread().getName());
66    }
67
68    // 打印线程名
69    public void printName() {
70        System.out.println(Thread.currentThread().getName());
71    }
72
73    // 打印线程名
74    public void printName() {
75        System.out.println(Thread.currentThread().getName());
76    }
77
78    // 打印线程名
79    public void printName() {
80        System.out.println(Thread.currentThread().getName());
81    }
82
83    // 打印线程名
84    public void printName() {
85        System.out.println(Thread.currentThread().getName());
86    }
87
88    // 打印线程名
89    public void printName() {
90        System.out.println(Thread.currentThread().getName());
91    }
92
93    // 打印线程名
94    public void printName() {
95        System.out.println(Thread.currentThread().getName());
96    }
97
98    // 打印线程名
99    public void printName() {
100       System.out.println(Thread.currentThread().getName());
101   }
102 }
```

```
54                     ActionContext.getActionContext().getContext().setName(name);
55             } catch (InterruptedException e) {
56                 e.printStackTrace();
57             }
58         }
59     }
60
61     // 设置上下文 ID
62     public class QueryIdAction {
63         public void execute() {
64             try {
65                 Thread.sleep(1000L);
66                 long id = Thread.currentThread().getId();
67                 ActionContext.getActionContext().getContext().setId(id);
68             } catch (InterruptedException e) {
69                 e.printStackTrace();
70             }
71         }
72     }
73
74     // 执行方法
75     public class ExecutionTask implements Runnable {
76         private QueryNameAction queryNameAction = new QueryNameAction();
77         private QueryIdAction queryIdAction = new QueryIdAction();
78
79         @Override
80         public void run() {
81             queryNameAction.execute();// 设置线程名
82             System.out.println("The name query successful");
83             queryIdAction.execute();// 设置线程 ID
84             System.out.println("The id query successful");
85
86             System.out.println("The Name is " + ActionContext.getActionContext().getName());
87         }
88     }
89
90     public static void main(String[] args) {
91         IntStream.range(1, 5).forEach(i -> new Thread(new ContextTest().new ExecutionTask()).start());
92     }
93 }
```

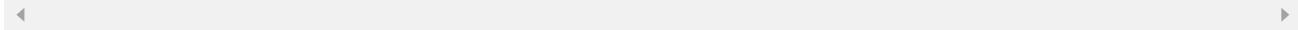


运行结果：

复制代码

```
1 The name query successful
2 The name query successful
3 The name query successful
```

```
4 The name query successful
5 The id query successful
6 The id query successful
7 The id query successful
8 The id query successful
9 The Name is Thread-2 and id 12
10 The Name is Thread-0 and id 10
11 The Name is Thread-1 and id 11
12 The Name is Thread-3 and id 13
```



Thread-Per-Message 设计模式

Thread-Per-Message 设计模式翻译过来的意思就是每个消息一个线程的意思。例如，我们在处理 Socket 通信的时候，通常是一个线程处理事件监听以及 I/O 读写，如果 I/O 读写操作非常耗时，这个时候便会影响到事件监听处理事件。

这个时候 Thread-Per-Message 模式就可以很好地解决这个问题，一个线程监听 I/O 事件，每当监听到一个 I/O 事件，则交给另一个处理线程执行 I/O 操作。下面，我们还是通过一个例子来学习下该设计模式的实现。

复制代码

```
1 //IO 处理
2 public class ServerHandler implements Runnable{
3     private Socket socket;
4
5     public ServerHandler(Socket socket) {
6         this.socket = socket;
7     }
8
9     public void run() {
10        BufferedReader in = null;
11        PrintWriter out = null;
12        String msg = null;
13        try {
14            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
15            out = new PrintWriter(socket.getOutputStream(),true);
16            while ((msg = in.readLine()) != null && msg.length()!=0) {// 当连接成功后在此
17                System.out.println("server received : " + msg);
18                out.print("received~\n");
19                out.flush();
20            }
21        } catch (Exception e) {
22            e.printStackTrace();
23        } finally {
```

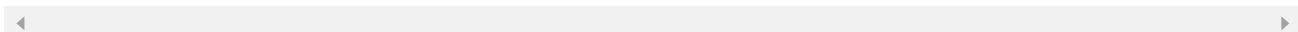
```
24         try {
25             in.close();
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29         try {
30             out.close();
31         } catch (Exception e) {
32             e.printStackTrace();
33         }
34         try {
35             socket.close();
36         } catch (IOException e) {
37             e.printStackTrace();
38         }
39     }
40 }
41 }
```



复制代码

```
1 //Socket 启动服务
2 public class Server {
3
4     private static int DEFAULT_PORT = 12345;
5     private static ServerSocket server;
6
7     public static void start() throws IOException {
8         start(DEFAULT_PORT);
9     }
10
11     public static void start(int port) throws IOException {
12         if (server != null) {
13             return;
14         }
15
16         try {
17             // 启动服务
18             server = new ServerSocket(port);
19             // 通过无线循环监听客户端连接
20             while (true) {
21
22                 Socket socket = server.accept();
23                 // 当有新的客户端接入时，会执行下面的代码
24                 long start = System.currentTimeMillis();
25                 new Thread(new ServerHandler(socket)).start();
26
27                 long end = System.currentTimeMillis();
28             }
29         }
30     }
31 }
```

```
29                     System.out.println("Spend time is " + (end - start));
30                 }
31             } finally {
32                 if (server != null) {
33                     System.out.println(" 服务器已关闭。");
34                     server.close();
35                 }
36             }
37         }
38     }
39
40     public static void main(String[] args) throws InterruptedException{
41
42         // 运行服务端
43         new Thread(new Runnable() {
44             public void run() {
45                 try {
46                     Server.start();
47                 } catch (IOException e) {
48                     e.printStackTrace();
49                 }
50             }
51         }).start();
52     }
53 }
54 }
55 }
```



以上，我们是完成了一个使用 Thread-Per-Message 设计模式实现的 Socket 服务端的代码。但这里是有一个问题的，你发现了吗？

使用这种设计模式，如果遇到大的高并发，就会出现严重的性能问题。如果针对每个 I/O 请求都创建一个线程来处理，在有大量请求同时进来时，就会创建大量线程，而此时 JVM 有可能会因为无法处理这么多线程，而出现内存溢出的问题。

退一步讲，即使是不会有大量线程的场景，每次请求过来都需要创建和销毁线程，这对系统来说，也是一笔不小的性能开销。

面对这种情况，**我们可以使用线程池来代替线程的创建和销毁**，这样就可以避免创建大量线程而带来的性能问题，是一种很好的调优方法。

Worker-Thread 设计模式

这里的 Worker 是工人的意思，代表在 Worker Thread 设计模式中，会有一些工人（线程）不断轮流处理过来的工作，当没有工作时，工人则会处于等待状态，直到有新的工作进来。除了工人角色，Worker Thread 设计模式中还包括了流水线和产品。

这种设计模式相比 Thread-Per-Message 设计模式，可以减少频繁创建、销毁线程所带来的性能开销，还有无限制地创建线程所带来的内存溢出风险。

我们可以假设一个场景来看下该模式的实现，通过 Worker Thread 设计模式来完成一个物流分拣的作业。

假设一个物流仓库的物流分拣流水线上有 8 个机器人，它们不断从流水线上获取包裹并对其进行包装，送其上车。当仓库中的商品被打包好后，会投放到物流分拣流水线上，而不是直接交给机器人，机器人会再从流水线中随机分拣包裹。代码如下：

 复制代码

```
1 // 包裹类
2 public class Package {
3     private String name;
4     private String address;
5
6     public String getName() {
7         return name;
8     }
9
10    public void setName(String name) {
11        this.name = name;
12    }
13
14    public String getAddress() {
15        return address;
16    }
17
18    public void setAddress(String address) {
19        this.address = address;
20    }
21
22    public void execute() {
23        System.out.println(Thread.currentThread().getName()+" executed "+this);
24    }
25 }
```



 复制代码

```
1 // 流水线
2 public class PackageChannel {
3     private final static int MAX_PACKAGE_NUM = 100;
4
5     private final Package[] packageQueue;
6     private final Worker[] workerPool;
7     private int head;
8     private int tail;
9     private int count;
10
11    public PackageChannel(int workers) {
12        this.packageQueue = new Package[MAX_PACKAGE_NUM];
13        this.head = 0;
14        this.tail = 0;
15        this.count = 0;
16        this.workerPool = new Worker[workers];
17        this.init();
18    }
19
20    private void init() {
21        for (int i = 0; i < workerPool.length; i++) {
22            workerPool[i] = new Worker("Worker-" + i, this);
23        }
24    }
25
26    /**
27     * push switch to start all of worker to work
28     */
29    public void startWorker() {
30        Arrays.asList(workerPool).forEach(Worker::start);
31    }
32
33    public synchronized void put(Package packagereq) {
34        while (count >= packageQueue.length) {
35            try {
36                this.wait();
37            } catch (InterruptedException e) {
38                e.printStackTrace();
39            }
40        }
41        this.packageQueue[tail] = packagereq;
42        this.tail = (tail + 1) % packageQueue.length;
43        this.count++;
44        this.notifyAll();
45    }
46
47    public synchronized Package take() {
48        while (count <= 0) {
49            try {
50                this.wait();
51            } catch (InterruptedException e) {
```

```
52             e.printStackTrace();
53         }
54     }
55     Package request = this.packageQueue[head];
56     this.head = (this.head + 1) % this.packageQueue.length;
57     this.count--;
58     this.notifyAll();
59     return request;
60 }
61
62 }
```

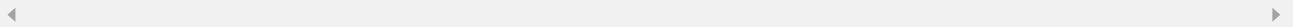
复制代码

```
1 // 机器人
2 public class Worker extends Thread{
3     private static final Random random = new Random(System.currentTimeMillis());
4     private final PackageChannel channel;
5
6     public Worker(String name, PackageChannel channel) {
7         super(name);
8         this.channel = channel;
9     }
10
11    @Override
12    public void run() {
13        while (true) {
14            channel.take().execute();
15
16            try {
17                Thread.sleep(random.nextInt(1000));
18            } catch (InterruptedException e) {
19                e.printStackTrace();
20            }
21        }
22    }
23
24 }
```

复制代码

```
1 public class Test {
2     public static void main(String[] args) {
3         // 新建 8 个工人
4         final PackageChannel channel = new PackageChannel(8);
5         // 开始工作
6         channel.startWorker();
```

```
7 // 为流水线添加包裹
8     for(int i=0; i<100; i++) {
9         Package packagereq = new Package();
10        packagereq.setAddress("test");
11        packagereq.setName("test");
12        channel.put(packagereq);
13    }
14 }
15 }
```



我们可以看到，这里有 8 个工人在不断地分拣仓库中已经包装好的商品。

总结

平时，如果需要传递或隔离一些线程变量时，我们可以考虑使用上下文设计模式。在数据库读写分离的业务场景中，则经常会用到 ThreadLocal 实现动态切换数据源操作。但在使用 ThreadLocal 时，我们需要注意内存泄漏问题，在之前的[第 25 讲](#)中，我们已经讨论过这个问题了。

当主线程处理每次请求都非常耗时时，就可能出现阻塞问题，这时候我们可以考虑将主线程业务分工到新的业务线程中，从而提高系统的并行处理能力。而 Thread-Per-Message 设计模式以及 Worker-Thread 设计模式则都是通过多线程分工来提高系统并行处理能力的设计模式。

29 | 生产者消费者模式：电商库存设计优化



生产者消费者模式，在之前的一些案例中，我们是有使用过的，相信你有一定的了解。这个模式是一个十分经典的多线程并发协作模式，生产者与消费者是通过一个中间容器来解决强耦合关系，并以此来实现不同的生产与消费速度，从而达到缓冲的效果。

使用生产者消费者模式，可以提高系统的性能和吞吐量，今天我们就来看看该模式的几种实现方式，还有其在电商库存中的应用。

Object 的 wait/notify/notifyAll 实现生产者消费者

在第 16 讲中，我就曾介绍过使用 Object 的 wait/notify/notifyAll 实现生产者消费者模式，这种方式是基于 Object 的 wait/notify/notifyAll 与对象监视器（Monitor）实现线程间的等待和通知。

还有，在第 12 讲中我也详细讲解过 Monitor 的工作原理，借此我们可以得知，这种方式实现的生产者消费者模式是基于内核来实现的，有可能会导致大量的上下文切换，所以性能并不是最理想的。

Lock 中 Condition 的 await/signal/signalAll 实现生产者消费者

相对 Object 类提供的 wait/notify/notifyAll 方法实现的生产者消费者模式，我更推荐使用 java.util.concurrent 包提供的 Lock && Condition 实现的生产者消费者模式。

在接口 Condition 类中定义了 await/signal/signalAll 方法，其作用与 Object 的 wait/notify/notifyAll 方法类似，该接口类与显示锁 Lock 配合，实现对线程的阻塞和唤醒操作。

我在第 13 讲中详细讲到了显示锁，显示锁 ReentrantLock 或 ReentrantReadWriteLock 都是基于 AQS 实现的，而在 AQS 中有一个内部类 ConditionObject 实现了 Condition 接口。

我们知道 AQS 中存在一个同步队列（CLH 队列），当一个线程没有获取到锁时就会进入到同步队列中进行阻塞，如果被唤醒后获取到锁，则移除同步队列。

除此之外，AQS 中还存在一个条件队列，通过 addWaiter 方法，可以将 await() 方法调用的线程放入到条件队列中，线程进入等待状态。当调用 signal 以及 signalAll 方法后，线程将会被唤醒，并从条件队列中删除，之后进入到同步队列中。条件队列是通过一个单向链表实现的，所以 Condition 支持多个等待队列。

由上可知，Lock 中 Condition 的 await/signal/signalAll 实现的生产者消费者模式，是基于 Java 代码层实现的，所以在性能和扩展性方面都更有优势。

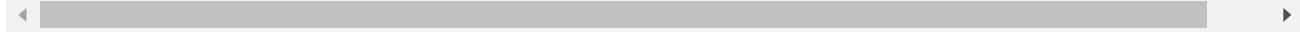
下面来看一个案例，我们通过一段代码来实现一个商品库存的生产和消费。

 复制代码

```
1 public class LockConditionTest {
```

```
2  
3     private LinkedList<String> product = new LinkedList<String>();  
4  
5     private int maxInventory = 10; // 最大库存  
6  
7     private Lock lock = new ReentrantLock();// 资源锁  
8  
9     private Condition condition = lock.newCondition();// 库存非满和非空条件  
10  
11    /**  
12     * 新增商品库存  
13     * @param e  
14     */  
15    public void produce(String e) {  
16        lock.lock();  
17        try {  
18            while (product.size() == maxInventory) {  
19                condition.await();  
20            }  
21  
22            product.add(e);  
23            System.out.println(" 放入一个商品库存，总库存为: " + product.size());  
24            condition.signalAll();  
25  
26        } catch (Exception ex) {  
27            ex.printStackTrace();  
28        } finally {  
29            lock.unlock();  
30        }  
31    }  
32  
33    /**  
34     * 消费商品  
35     * @return  
36     */  
37    public String consume() {  
38        String result = null;  
39        lock.lock();  
40        try {  
41            while (product.size() == 0) {  
42                condition.await();  
43            }  
44  
45            result = product.removeLast();  
46            System.out.println(" 消费一个商品，总库存为: " + product.size());  
47            condition.signalAll();  
48  
49        } catch (Exception e) {  
50            e.printStackTrace();  
51        } finally {  
52            lock.unlock();  
53        }  
54    }
```

```
54             return result;
55     }
56
57     /**
58      * 生产者
59      * @author admin
60      *
61      */
62
63     private class Producer implements Runnable {
64
65         public void run() {
66             for (int i = 0; i < 20; i++) {
67                 produce("商品 " + i);
68             }
69         }
70     }
71
72     /**
73      * 消费者
74      * @author admin
75      *
76      */
77
78     private class Customer implements Runnable {
79
80         public void run() {
81             for (int i = 0; i < 20; i++) {
82                 consume();
83             }
84         }
85     }
86
87     public static void main(String[] args) {
88
89         LockConditionTest lc = new LockConditionTest();
90         new Thread(lc.new Producer()).start();
91         new Thread(lc.new Customer()).start();
92         new Thread(lc.new Producer()).start();
93         new Thread(lc.new Customer()).start();
94
95     }
96 }
```



看完案例，请你思考下，我们对此还有优化的空间吗？

从代码中应该不难发现，生产者和消费者都在竞争同一把锁，而实际上两者没有同步关系，由于 Condition 能够支持多个等待队列以及不响应中断，所以我们可以将生产者和消费者的等待条件和锁资源分离，从而进一步优化系统并发性能，代码如下：

 复制代码

```
1     private LinkedList<String> product = new LinkedList<String>();
2     private AtomicInteger inventory = new AtomicInteger(0); // 实时库存
3
4     private int maxInventory = 10; // 最大库存
5
6     private Lock consumerLock = new ReentrantLock(); // 资源锁
7     private Lock productLock = new ReentrantLock(); // 资源锁
8
9     private Condition notEmptyCondition = consumerLock.newCondition(); // 库存满和空条件
10    private Condition notFullCondition = productLock.newCondition(); // 库存满和空条件
11
12    /**
13     * 新增商品库存
14     * @param e
15     */
16    public void produce(String e) {
17        productLock.lock();
18        try {
19            while (inventory.get() == maxInventory) {
20                notFullCondition.await();
21            }
22
23            product.add(e);
24
25            System.out.println(" 放入一个商品库存，总库存为: " + inventory.in
26
27            if(inventory.get()<maxInventory) {
28                notFullCondition.signalAll();
29            }
30
31        } catch (Exception ex) {
32            ex.printStackTrace();
33        } finally {
34            productLock.unlock();
35        }
36
37        if(inventory.get()>0) {
38            try {
39                consumerLock.lockInterruptibly();
40                notEmptyCondition.signalAll();
41            } catch (InterruptedException e1) {
42                // TODO Auto-generated catch block
43                e1.printStackTrace();
44            }finally {
```

```
45                     consumerLock.unlock();
46                 }
47             }
48         }
49     }
50
51     /**
52      * 消费商品
53      * @return
54      */
55     public String consume() {
56         String result = null;
57         consumerLock.lock();
58         try {
59             while (inventory.get() == 0) {
60                 notEmptyCondition.await();
61             }
62
63             result = product.removeLast();
64             System.out.println(" 消费一个商品，总库存为: " + inventory.decrem
65
66             if(inventory.get()>0) {
67                 notEmptyCondition.signalAll();
68             }
69         } catch (Exception e) {
70             e.printStackTrace();
71         } finally {
72             consumerLock.unlock();
73         }
74
75         if(inventory.get()<maxInventory) {
76
77             try {
78                 productLock.lockInterruptibly();
79                 notFullCondition.signalAll();
80             } catch (InterruptedException e1) {
81                 // TODO Auto-generated catch block
82                 e1.printStackTrace();
83             }finally {
84                 productLock.unlock();
85             }
86         }
87         return result;
88     }
89     /**
90      * 生产者
91      * @author admin
92      *
93      */
94     private class Producer implements Runnable {
95
96         public void run() {
```

```
97             for (int i = 0; i < 20; i++) {
98                 produce(" 商品 " + i);
99             }
100        }
101    }
102
103    /**
104     * 消费者
105     * @author admin
106     *
107     */
108    private class Customer implements Runnable {
109
110        public void run() {
111            for (int i = 0; i < 20; i++) {
112                consume();
113            }
114        }
115    }
116
117    public static void main(String[] args) {
118
119        LockConditionTest2 lc = new LockConditionTest2();
120        new Thread(lc.new Producer()).start();
121        new Thread(lc.new Customer()).start();
122
123    }
124 }
```



我们分别创建 `productLock` 以及 `consumerLock` 两个锁资源，前者控制生产者线程并行操作，后者控制消费者线程并发运行；同时也设置两个条件变量，一个是 `notEmptyCondition`，负责控制消费者线程状态，一个是 `notFullCondition`，负责控制生产者线程状态。这样优化后，可以减少消费者与生产者的竞争，实现两者并发执行。

我们这里是基于 `LinkedList` 来存取库存的，虽然 `LinkedList` 是非线程安全，但我们新增是操作头部，而消费是操作队列的尾部，理论上来说没有线程安全问题。而库存的实际数量 `inventory` 是基于 `AtomicInteger` (CAS 锁) 线程安全类实现的，既可以保证原子性，也可以保证消费者和生产者之间是可见的。

BlockingQueue 实现生产者消费者

相对前两种实现方式，`BlockingQueue` 实现是最简单明了的，也是最容易理解的。

因为 BlockingQueue 是线程安全的，且从队列中获取或者移除元素时，如果队列为空，获取或移除操作则需要等待，直到队列不为空；同时，如果向队列中添加元素，假设此时队列无可用空间，添加操作也需要等待。所以 BlockingQueue 非常适合用来实现生产者消费者模式。还是以一个案例来看下它的优化，代码如下：

 复制代码

```
1 public class BlockingQueueTest {  
2  
3     private int maxInventory = 10; // 最大库存  
4  
5     private BlockingQueue<String> product = new LinkedBlockingQueue<>(maxInventory)  
6  
7     /**  
8      * 新增商品库存  
9      * @param e  
10     */  
11    public void produce(String e) {  
12        try {  
13            product.put(e);  
14            System.out.println(" 放入一个商品库存，总库存为: " + product.size());  
15        } catch (InterruptedException e1) {  
16            // TODO Auto-generated catch block  
17            e1.printStackTrace();  
18        }  
19    }  
20  
21    /**  
22     * 消费商品  
23     * @return  
24     */  
25    public String consume() {  
26        String result = null;  
27        try {  
28            result = product.take();  
29            System.out.println(" 消费一个商品，总库存为: " + product.size());  
30        } catch (InterruptedException e) {  
31            // TODO Auto-generated catch block  
32            e.printStackTrace();  
33        }  
34  
35        return result;  
36    }  
37  
38    /**  
39     * 生产者  
40     * @author admin  
41     *  
42     */  
43    private class Producer implements Runnable {
```

```
44
45     public void run() {
46         for (int i = 0; i < 20; i++) {
47             produce("商品 " + i);
48         }
49     }
50
51 }
52
53 /**
54 * 消费者
55 * @author admin
56 *
57 */
58 private class Customer implements Runnable {
59
60     public void run() {
61         for (int i = 0; i < 20; i++) {
62             consume();
63         }
64     }
65 }
66
67 public static void main(String[] args) {
68
69     BlockingQueueTest lc = new BlockingQueueTest();
70     new Thread(lc.new Producer()).start();
71     new Thread(lc.new Customer()).start();
72     new Thread(lc.new Producer()).start();
73     new Thread(lc.new Customer()).start();
74
75 }
76 }
```



在这个案例中，我们创建了一个 `LinkedBlockingQueue`，并设置队列大小。之后我们创建一个消费方法 `consume()`，方法里面调用 `LinkedBlockingQueue` 中的 `take()` 方法，消费者通过该方法获取商品，当队列中商品数量为零时，消费者将进入等待状态；我们再创建一个生产方法 `produce()`，方法里面调用 `LinkedBlockingQueue` 中的 `put()` 方法，生产方通过该方法往队列中放商品，如果队列满了，生产者就将进入等待状态。

生产者消费者优化电商库存设计

了解完生产者消费者模式的几种常见实现方式，接下来我们就具体看看该模式是如何优化电商库存设计的。

电商系统中经常会有抢购活动，在这类促销活动中，抢购商品的库存实际是存在库存表中的。为了提高抢购性能，我们通常会将库存存放在缓存中，通过缓存中的库存来实现库存的精确扣减。在提交订单并付款之后，我们还需要再去扣除数据库中的库存。如果遇到瞬时高并发，我们还都去操作数据库的话，那么在单表单库的情况下，数据库就很可能会出现性能瓶颈。

而我们库存表如果要实现分库分表，势必会增加业务的复杂度。试想一个商品的库存分别在不同库的表中，我们在扣除库存时，又该如何判断去哪个库中扣除呢？

如果随意扣除表中库存，那么就会出现有些表已经扣完了，有些表中还有库存的情况，这样的操作显然是不合理的，此时就需要额外增加逻辑判断来解决问题。

在不分库分表的情况下，为了提高订单中扣除库存业务的性能以及吞吐量，我们就可以采用生产者消费者模式来实现系统的性能优化。

创建订单等于生产者，存放订单的队列则是缓冲容器，而从队列中消费订单则是数据库扣除库存操作。其中存放订单的队列可以极大限度地缓冲高并发给数据库带来的压力。

我们还可以基于消息队列来实现生产者消费者模式，如今 RabbitMQ、RocketMQ 都实现了事务，我们只需要将订单通过事务提交到 MQ 中，扣除库存的消费方只需要通过消费 MQ 来逐步操作数据库即可。

总结

使用生产者消费者模式来缓冲高并发数据库扣除库存压力，类似这样的例子其实还有很多。

例如，我们平时使用消息队列来做高并发流量削峰，也是基于这个原理。抢购商品时，如果所有的抢购请求都直接进入判断是否有库存和冻结缓存库存等逻辑业务中，由于这些逻辑业务操作会增加资源消耗，就可能会压垮应用服务。此时，为了保证系统资源使用的合理性，我们可以通过一个消息队列来缓冲瞬时的高并发请求。

生产者消费者模式除了可以做缓冲优化系统性能之外，它还可以应用在处理一些执行任务时间比较长的场景中。

例如导出报表业务，用户在导出一种比较大的报表时，通常需要等待很长时间，这样的用户体验是非常差的。通常我们可以固定一些报表内容，比如用户经常需要在今天导出昨天的销

量报表，或者在月初导出上个月的报表，我们就可以提前将报表导出到本地或内存中，这样用户就可以在很短的时间内直接下载报表了。

30 | 装饰器模式：如何优化电商系统中复杂的商品价格策略？



开始今天的学习之前，我想先请你思考一个问题。假设现在有这样一个需求，让你设计一个装修功能，用户可以动态选择不同的装修功能来装饰自己的房子。例如，水电装修、天花板以及粉刷墙等属于基本功能，而设计窗帘装饰窗户、设计吊顶装饰房顶等未必是所有用户都需要的，这些功能则需要实现动态添加。还有就是一旦有新的装修功能，我们也可以实现动态添加。如果你要你来负责，你会怎么设计呢？

此时你可能会想了，通常给一个对象添加功能，要么直接修改代码，在对象中添加相应的功能，要么派生对应的子类来扩展。然而，前者每次都需要修改对象的代码，这显然不是理想的面向对象设计，即便后者是通过派生对应的子类来扩展，也很难满足复杂的随意组合功能需求。

面对这种情况，使用装饰器模式应该再合适不过了。它的优势我想你多少知道一点，我在这里总结一下。

装饰器模式能够实现为对象动态添加装修功能，它是从一个对象的外部来给对象添加功能，所以有非常灵活的扩展性，我们可以在对原来的代码毫无修改的前提下，为对象添加新功能。除此之外，装饰器模式还能够实现对象的动态组合，借此我们可以很灵活地给动态组合的对象，匹配所需要的功能。

下面我们就通过实践，具体看看该模式的优势。

什么是装饰器模式？

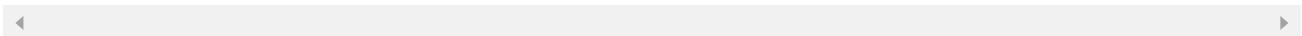
在这之前，我先简单介绍下什么是装饰器模式。装饰器模式包括了以下几个角色：接口、具体对象、装饰类、具体装饰类。

接口定义了具体对象的一些实现方法；具体对象定义了一些初始化操作，比如开头设计装修功能的案例中，水电装修、天花板以及粉刷墙等都是初始化操作；装饰类则是一个抽象类，主要用来初始化具体对象的一个类；其它的具体装饰类都继承了该抽象类。

下面我们就通过装饰器模式来实现下装修功能，代码如下：

 复制代码

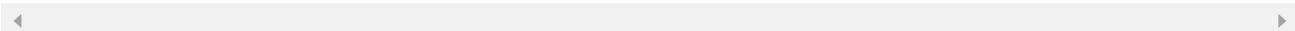
```
1 /**
2  * 定义一个基本装修接口
3  * @author admin
4  *
5 */
6 public interface IDecorator {
7
8     /**
9      * 装修方法
10     */
11    void decorate();
12
13 }
```



 复制代码

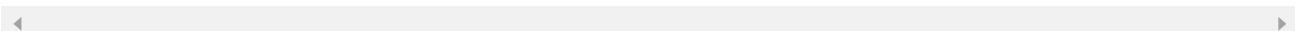
```
1 /**
```

```
2 * 装修基本类
3 * @author admin
4 *
5 */
6 public class Decorator implements IDecorator{
7
8     /**
9      * 基本实现方法
10     */
11    public void decorate() {
12        System.out.println(" 水电装修、天花板以及粉刷墙。。。");
13    }
14
15 }
```



复制代码

```
1 /**
2 * 基本装饰类
3 * @author admin
4 *
5 */
6 public abstract class BaseDecorator implements IDecorator{
7
8     private IDecorator decorator;
9
10    public BaseDecorator(IDecorator decorator) {
11        this.decorator = decorator;
12    }
13
14    /**
15     * 调用装饰方法
16     */
17    public void decorate() {
18        if(decorator != null) {
19            decorator.decorate();
20        }
21    }
22 }
```



复制代码

```
1 /**
2 * 窗帘装饰类
3 * @author admin
4 *
5 */
```

```
6 public class CurtainDecorator extends BaseDecorator{  
7  
8     public CurtainDecorator(IDecorator decorator) {  
9         super(decorator);  
10    }  
11  
12    /**  
13     * 窗帘具体装饰方法  
14     */  
15    @Override  
16    public void decorate() {  
17        System.out.println(" 窗帘装饰。。。");  
18        super.decorate();  
19    }  
20  
21 }
```

 复制代码

```
1 public static void main( String[] args )  
2 {  
3     IDecorator decorator = new Decorator();  
4     IDecorator curtainDecorator = new CurtainDecorator(decorator);  
5     curtainDecorator.decorate();  
6  
7 }
```

 复制代码

```
1 窗帘装饰。。。  
2 水电装修、天花板以及粉刷墙。。。
```

通过这个案例，我们可以了解到：如果我们想要在基础类上添加新的装修功能，只需要基于抽象类 `BaseDecorator` 去实现继承类，通过构造函数调用父类，以及重写装修方法实现装修窗帘的功能即可。在 `main` 函数中，我们通过实例化装饰类，调用装修方法，即可在基础装修的前提下，获得窗帘装修功能。

基于装饰器模式实现的装修功能的代码结构简洁易读，业务逻辑也非常清晰，并且如果我们需要扩展新的装修功能，只需要新增一个继承了抽象装饰类的子类即可。

在这个案例中，我们仅实现了业务扩展功能，接下来，我将通过装饰器模式优化电商系统中的商品价格策略，实现不同促销活动的灵活组合。

优化电商系统中的商品价格策略

相信你一定不陌生，购买商品时经常会用到的限时折扣、红包、抵扣券以及特殊抵扣金等，种类很多，如果换到开发视角，实现起来就更复杂了。

例如，每逢双十一，为了加大商城的优惠力度，开发往往要设计红包 + 限时折扣或红包 + 抵扣券等组合来实现多重优惠。而在平时，由于某些特殊原因，商家还会赠送特殊抵扣券给购买用户，而特殊抵扣券 + 各种优惠又是另一种组合方式。

要实现以上这类组合优惠的功能，最快、最普遍的实现方式就是通过大量 if-else 的方式来实现。但这种方式包含了大量的逻辑判断，致使其他开发人员很难读懂业务，并且一旦有新的优惠策略或者价格组合策略出现，就需要修改代码逻辑。

这时，刚刚介绍的装饰器模式就很适合用在这里，其相互独立、自由组合以及方便动态扩展功能的特性，可以很好地解决 if-else 方式的弊端。下面我们就用装饰器模式动手实现一套商品价格策略的优化方案。

首先，我们先建立订单和商品的属性类，在本次案例中，为了保证简洁性，我只建立了几个关键字段。以下几个重要属性关系为，主订单包含若干详细订单，详细订单中记录了商品信息，商品信息中包含了促销类型信息，一个商品可以包含多个促销类型（本案例只讨论单个促销和组合促销）：

 复制代码

```
1 /**
2  * 主订单
3  * @author admin
4 *
5 */
6 public class Order {
7
8     private int id; // 订单 ID
9     private String orderNo; // 订单号
10    private BigDecimal totalPayMoney; // 总支付金额
```

```
11     private List<OrderDetail> list; // 详细订单列表
12 }
```

 复制代码

```
1 /**
2  * 详细订单
3  * @author admin
4  *
5 */
6 public class OrderDetail {
7     private int id; // 详细订单 ID
8     private int orderId;// 主订单 ID
9     private Merchandise merchandise; // 商品详情
10    private BigDecimal payMoney; // 支付单价
11 }
```

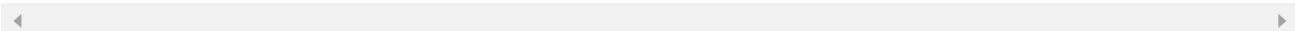
 复制代码

```
1 /**
2  * 商品
3  * @author admin
4  *
5 */
6 public class Merchandise {
7
8     private String sku;// 商品 SKU
9     private String name; // 商品名称
10    private BigDecimal price; // 商品单价
11    private Map<PromotionType, SupportPromotions> supportPromotions; // 支持促销类型
12 }
```

 复制代码

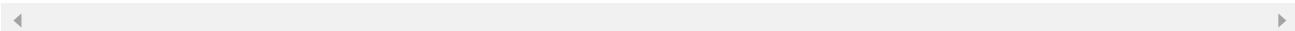
```
1 /**
2  * 促销类型
3  * @author admin
4  *
5 */
6 public class SupportPromotions implements Cloneable{
7
8     private int id;// 该商品促销的 ID
9     private PromotionType promotionType;// 促销类型 1\优惠券 2\红包
10    private int priority; // 优先级
```

```
11     private UserCoupon userCoupon; // 用户领取该商品的优惠券
12     private UserRedPacket userRedPacket; // 用户领取该商品的红包
13
14     // 重写 clone 方法
15     public SupportPromotions clone(){
16         SupportPromotions supportPromotions = null;
17         try{
18             supportPromotions = (SupportPromotions)super.clone();
19         }catch(CloneNotSupportedException e){
20             e.printStackTrace();
21         }
22         return supportPromotions;
23     }
24 }
```



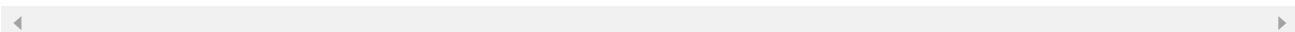
复制代码

```
1 /**
2 * 优惠券
3 * @author admin
4 *
5 */
6 public class UserCoupon {
7
8     private int id; // 优惠券 ID
9     private int userId; // 领取优惠券用户 ID
10    private String sku; // 商品 SKU
11    private BigDecimal coupon; // 优惠金额
12 }
```



复制代码

```
1 /**
2 * 红包
3 * @author admin
4 *
5 */
6 public class UserRedPacket {
7
8     private int id; // 红包 ID
9     private int userId; // 领取用户 ID
10    private String sku; // 商品 SKU
11    private BigDecimal redPacket; // 领取红包金额
12 }
```



接下来，我们再建立一个计算支付金额的接口类以及基本类：

 复制代码

```
1 /**
2  * 计算支付金额接口类
3  * @author admin
4  *
5 */
6 public interface IBaseCount {
7
8     BigDecimal countPayMoney(OrderDetail orderDetail);
9
10 }
```

◀ ▶

 复制代码

```
1 /**
2  * 支付基本类
3  * @author admin
4  *
5 */
6 public class BaseCount implements IBaseCount{
7
8     public BigDecimal countPayMoney(OrderDetail orderDetail) {
9         orderDetail.setPayMoney(orderDetail.getMerchandise().getPrice());
10        System.out.println("商品原单价金额为：" + orderDetail.getPayMoney());
11
12        return orderDetail.getPayMoney();
13    }
14
15 }
```

◀ ▶

然后，我们再建立一个计算支付金额的抽象类，由抽象类调用基本类：

 复制代码

```
1 /**
2  * 计算支付金额的抽象类
3  * @author admin
4  *
5 */
6 public abstract class BaseCountDecorator implements IBaseCount{
7
8     private IBaseCount count;
```

```
9
10    public BaseCountDecorator(IBaseCount count) {
11        this.count = count;
12    }
13
14    public BigDecimal countPayMoney(OrderDetail orderDetail) {
15        BigDecimal payTotalMoney = new BigDecimal(0);
16        if(count!=null) {
17            payTotalMoney = count.countPayMoney(orderDetail);
18        }
19        return payTotalMoney;
20    }
21 }
```

◀ ▶

然后，我们再通过继承抽象类来实现我们所需要的修饰类（优惠券计算类、红包计算类）：

 复制代码

```
1 /**
2  * 计算使用优惠券后的金额
3  * @author admin
4  *
5 */
6 public class CouponDecorator extends BaseCountDecorator{
7
8     public CouponDecorator(IBaseCount count) {
9         super(count);
10    }
11
12    public BigDecimal countPayMoney(OrderDetail orderDetail) {
13        BigDecimal payTotalMoney = new BigDecimal(0);
14        payTotalMoney = super.countPayMoney(orderDetail);
15        payTotalMoney = countCouponPayMoney(orderDetail);
16        return payTotalMoney;
17    }
18
19    private BigDecimal countCouponPayMoney(OrderDetail orderDetail) {
20
21        BigDecimal coupon = orderDetail.getMerchandise().getSupportPromotions();
22        System.out.println(" 优惠券金额: " + coupon);
23
24        orderDetail.setPayMoney(orderDetail.getPayMoney().subtract(coupon));
25        return orderDetail.getPayMoney();
26    }
27 }
```

◀ ▶

 复制代码

```
1 /**
2  * 计算使用红包后的金额
3  * @author admin
4  *
5 */
6 public class RedPacketDecorator extends BaseCountDecorator{
7
8     public RedPacketDecorator(IBaseCount count) {
9         super(count);
10    }
11
12    public BigDecimal countPayMoney(OrderDetail orderDetail) {
13        BigDecimal payTotalMoney = new BigDecimal(0);
14        payTotalMoney = super.countPayMoney(orderDetail);
15        payTotalMoney = countCouponPayMoney(orderDetail);
16        return payTotalMoney;
17    }
18
19    private BigDecimal countCouponPayMoney(OrderDetail orderDetail) {
20
21        BigDecimal redPacket = orderDetail.getMerchandise().getSupportPromotion();
22        System.out.println(" 红包优惠金额: " + redPacket);
23
24        orderDetail.setPayMoney(orderDetail.getPayMoney().subtract(redPacket));
25        return orderDetail.getPayMoney();
26    }
27 }
```



最后，我们通过一个工厂类来组合商品的促销类型：

 复制代码

```
1 /**
2  * 计算促销后的支付价格
3  * @author admin
4  *
5 */
6 public class PromotionFactory {
7
8     public static BigDecimal getPayMoney(OrderDetail orderDetail) {
9
10         // 获取给商品设定的促销类型
11         Map<PromotionType, SupportPromotions> supportPromotionslist = orderDetail.getMerchandise().getSupportPromotions();
12
13         // 初始化计算类
14         IBaseCount baseCount = new BaseCount();
15         if(supportPromotionslist!=null && supportPromotionslist.size()>0) {
```

```
16             for(PromotionType promotionType: supportPromotionslist.keySet())
17                 baseCount = protmotion(supportPromotionslist.get(promot:
18             }
19         }
20     return baseCount.countPayMoney(orderDetail);
21 }
22
23 /**
24 * 组合促销类型
25 * @param supportPromotions
26 * @param baseCount
27 * @return
28 */
29 private static IBaseCount protmotion(SupportPromotions supportPromotions, IBaseC
30     if(supportPromotions.getPromotionType()==PromotionType.COUPON) {
31         baseCount = new CouponDecorator(baseCount);
32     }else if(supportPromotions.getPromotionType()==PromotionType.REDPACKED)
33         baseCount = new RedPacketDecorator(baseCount);
34     }
35     return baseCount;
36 }
37
38 }
```

 复制代码

```
1 public static void main( String[] args ) throws InterruptedException, IOException
2 {
3     Order order = new Order();
4     init(order);
5
6     for(OrderDetail orderDetail: order.getList()) {
7         BigDecimal payMoney = PromotionFactory.getPayMoney(orderDetail);
8         orderDetail.setPayMoney(payMoney);
9         System.out.println(" 最终支付金额: " + orderDetail.getPayMoney());
10    }
11 }
```

 复制代码

- 1 商品原单价金额为: 20
- 2 优惠券金额: 3
- 3 红包优惠金额: 10
- 4 最终支付金额: 7

运行结果：

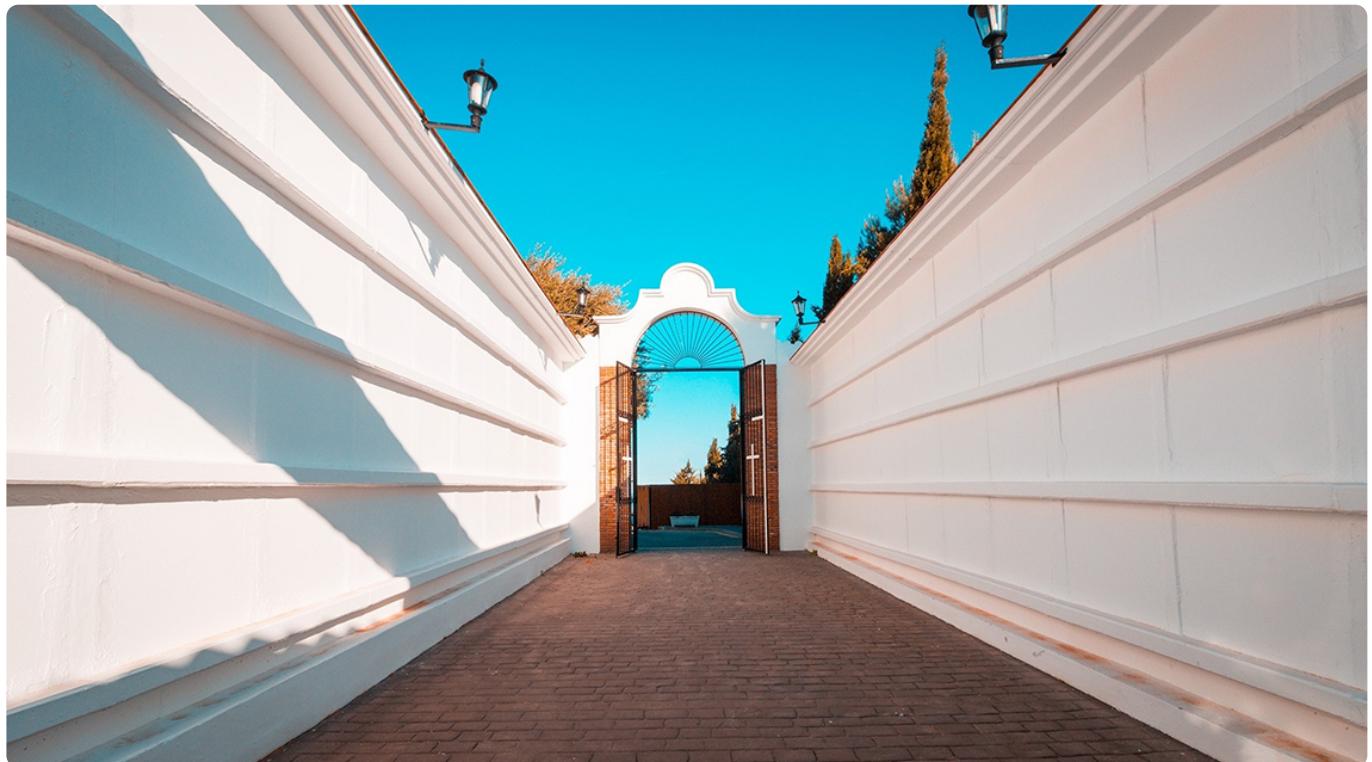
以上源码可以通过 [Github](#) 下载运行。通过以上案例可知：使用装饰器模式设计的价格优惠策略，实现各个促销类型的计算功能都是相互独立的类，并且可以通过工厂类自由组合各种促销类型。

总结

这讲介绍的装饰器模式主要用来优化业务的复杂度，它不仅简化了我们的业务代码，还优化了业务代码的结构设计，使得整个业务逻辑清晰、易读易懂。

通常，装饰器模式用于扩展一个类的功能，且支持动态添加和删除类的功能。在装饰器模式中，装饰类和被装饰类都只关心自身的业务，不相互干扰，真正实现了解耦。

32 | MySQL调优之SQL语句：如何写出高性能SQL语句？



从今天开始，我将带你一起学习 MySQL 的性能调优。MySQL 数据库是互联网公司使用最为频繁的数据库之一，不仅仅因为它开源免费，MySQL 卓越的性能、稳定的服务以及活跃的社区都成就了它的核心竞争力。

我们知道，应用服务与数据库的交互主要是通过 SQL 语句来实现的。在开发初期，我们更加关注的是使用 SQL 实现业务功能，然而系统上线后，随着生产环境数据的快速增长，之前写的很多 SQL 语句就开始暴露出性能问题。

在这个阶段中，**我们应该尽量避免一些慢 SQL 语句的实现**。但话说回来，SQL 语句慢的原因千千万，除了一些常规的慢 SQL 语句可以直接规避，其它的一味去规避也不是办法，我

们还要学会如何去分析、定位到其根本原因，并总结一些常用的 SQL 调优方法，以备不时之需。

那么今天我们就重点看看慢 SQL 语句的几种常见诱因，从这点出发，找到最佳方法，开启高性能 SQL 语句的大门。

慢 SQL 语句的几种常见诱因

1. 无索引、索引失效导致慢查询

如果在一张几千万数据的表中以一个没有索引的列作为查询条件，大部分情况下查询会非常耗时，这种查询毫无疑问是一个慢 SQL 查询。所以对于大数据量的查询，我们需要建立适合的索引来优化查询。

虽然我们很多时候建立了索引，但在一些特定的场景下，索引还有可能会失效，所以索引失效也是导致慢查询的主要原因之一。针对这点的调优，我会在第 34 讲中详解。

2. 锁等待

我们常用的存储引擎有 InnoDB 和 MyISAM，前者支持行锁和表锁，后者只支持表锁。

如果数据库操作是基于表锁实现的，试想下，如果一张订单表在更新时，需要锁住整张表，那么其它大量数据库操作（包括查询）都将处于等待状态，这将严重影响到系统的并发性能。

这时，InnoDB 存储引擎支持的行锁更适合高并发场景。但在使用 InnoDB 存储引擎时，我们要特别注意行锁升级为表锁的可能。在批量更新操作时，行锁就很可能会升级为表锁。

MySQL 认为如果对一张表使用大量行锁，会导致事务执行效率下降，从而可能造成其它事务长时间锁等待和更多的锁冲突问题发生，致使性能严重下降，所以 MySQL 会将行锁升级为表锁。还有，行锁是基于索引加的锁，如果我们在更新操作时，条件索引失效，那么行锁也会升级为表锁。

因此，基于表锁的数据库操作，会导致 SQL 阻塞等待，从而影响执行速度。在一些更新操作（insert\update\delete）大于或等于读操作的情况下，MySQL 不建议使用 MyISAM 存储引擎。

除了锁升级之外，行锁相对表锁来说，虽然粒度更细，并发能力提升了，但也带来了新的问题，那就是死锁。因此，在使用行锁时，我们要注意避免死锁。关于死锁，我还会在第 35 讲中详解。

3. 不恰当的 SQL 语句

使用不恰当的 SQL 语句也是慢 SQL 最常见的诱因之一。例如，习惯使用 <SELECT *>，<SELECT COUNT(*)> SQL 语句，在大数据表中使用 <LIMIT M,N> 分页查询，以及对非索引字段进行排序等等。

优化 SQL 语句的步骤

通常，我们在执行一条 SQL 语句时，要想知道这个 SQL 先后查询了哪些表，是否使用了索引，这些数据从哪里获取到，获取到数据遍历了多少行数据等等，我们可以通过 EXPLAIN 命令来查看这些执行信息。这些执行信息被统称为执行计划。

1. 通过 EXPLAIN 分析 SQL 执行计划

假设现在我们使用 EXPLAIN 命令查看当前 SQL 是否使用了索引，先通过 SQL EXPLAIN 导出相应的执行计划如下：

1	EXPLAIN SELECT * FROM `order` where id < 10;																													
2	<table border="1"><thead><tr><th>信息</th><th>Result 1</th><th>概况</th><th>状态</th></tr></thead><tbody><tr><td><table border="1"><thead><tr><th>id</th><th>select_type</th><th>table</th><th>partitions</th><th>type</th><th>possible_keys</th><th>key</th><th>key_len</th><th>ref</th><th>rows</th><th>filtered</th><th>Extra</th></tr></thead><tbody><tr><td>1</td><td>SIMPLE</td><td>order</td><td>(Null)</td><td>ALL</td><td>PRIMARY</td><td>(Null)</td><td>(Null)</td><td>(Null)</td><td>581</td><td>33.33</td><td>Using where</td></tr></tbody></table></td></tr></tbody></table>	信息	Result 1	概况	状态	<table border="1"><thead><tr><th>id</th><th>select_type</th><th>table</th><th>partitions</th><th>type</th><th>possible_keys</th><th>key</th><th>key_len</th><th>ref</th><th>rows</th><th>filtered</th><th>Extra</th></tr></thead><tbody><tr><td>1</td><td>SIMPLE</td><td>order</td><td>(Null)</td><td>ALL</td><td>PRIMARY</td><td>(Null)</td><td>(Null)</td><td>(Null)</td><td>581</td><td>33.33</td><td>Using where</td></tr></tbody></table>	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	1	SIMPLE	order	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	581	33.33	Using where
信息	Result 1	概况	状态																											
<table border="1"><thead><tr><th>id</th><th>select_type</th><th>table</th><th>partitions</th><th>type</th><th>possible_keys</th><th>key</th><th>key_len</th><th>ref</th><th>rows</th><th>filtered</th><th>Extra</th></tr></thead><tbody><tr><td>1</td><td>SIMPLE</td><td>order</td><td>(Null)</td><td>ALL</td><td>PRIMARY</td><td>(Null)</td><td>(Null)</td><td>(Null)</td><td>581</td><td>33.33</td><td>Using where</td></tr></tbody></table>	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	1	SIMPLE	order	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	581	33.33	Using where						
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra																			
1	SIMPLE	order	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	581	33.33	Using where																			

下面对图示中的每一个字段进行一个说明，从中你也能收获到很多零散的知识点。

id：每个执行计划都有一个 id，如果是一个联合查询，这里还将有多个 id。

select_type：表示 SELECT 查询类型，常见的有 SIMPLE（普通查询，即没有联合查询、子查询）、PRIMARY（主查询）、UNION（UNION 中后面的查询）、SUBQUERY（子查询）等。

table：当前执行计划查询的表，如果给表起别名了，则显示别名信息。

partitions：访问的分区表信息。

type：表示从表中查询到行所执行的方式，查询方式是 SQL 优化中一个很重要的指标，结果值从好到差依次是：system > const > eq_ref > ref > range > index > ALL。

```

1 EXPLAIN SELECT * FROM `order` where id = 2;
2
3
4

```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	const	PRIMARY	PRIMARY	4	const	1	100	(Null)

system/const：表中只有一行数据匹配，此时根据索引查询一次就能找到对应的数据。如果是 B + 树索引，我们知道此时索引构造成了多个层级的树，当查询的索引在树的底层时，查询效率就越低。const 表示此时索引在第一层，只需访问一层便能得到数据。

```

1 EXPLAIN SELECT * FROM `order` a, order_detail b where a.id = b.order_id;
2
3
4

```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	b	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where
1	SIMPLE	a	(Null)	eq_ref	PRIMARY	PRIMARY	4	demo.b.or 1	1	100	(Null)

eq_ref：使用唯一索引扫描，常见于多表连接中使用主键和唯一索引作为关联条件。

```

1 EXPLAIN SELECT * FROM `order` where order_no = "2";
2
3
4

```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ref	idx_order	idx_order	5	const	1	100	(Null)

ref：非唯一索引扫描，还可见于唯一索引最左原则匹配扫描。

```

1 EXPLAIN SELECT * FROM `order` where id > 2;
2
3
4

```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	range	PRIMARY	PRIMARY	4	(Null)	580	100	Using where

range：索引范围扫描，比如，<，>，between 等操作。

```

1 EXPLAIN SELECT id FROM `order`;
2
3
4

```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	index	(Null)	idx_order	5	(Null)	581	100	Using index

index：索引全表扫描，此时遍历整个索引树。

```
1 EXPLAIN SELECT * FROM `order` WHERE pay_money = 0;
2
3
4
```

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	581	10	Using where

ALL：表示全表扫描，需要遍历全表来找到对应的行。

possible_keys：可能使用到的索引。

key：实际使用到的索引。

key_len：当前使用的索引的长度。

ref：关联 id 等信息。

rows：查找到记录所扫描的行数。

filtered：查找到所需记录占总扫描记录数的比例。

Extra：额外的信息。

2. 通过 Show Profile 分析 SQL 执行性能

上述通过 EXPLAIN 分析执行计划，仅仅是停留在分析 SQL 的外部的执行情况，如果我们想要深入到 MySQL 内核中，从执行线程的状态和时间来分析的话，这个时候我们就可以选择 Profile。

Profile 除了可以分析执行线程的状态和时间，还支持进一步选择 ALL、CPU、MEMORY、BLOCK IO、CONTEXT SWITCHES 等类型来查询 SQL 语句在不同系统资源上所消耗的时间。以下是相关命令的注释：

 复制代码

```
1 SHOW PROFILE [type [, type] ... ]
2 [FOR QUERY n]
3 [LIMIT row_count [OFFSET offset]]
4
5 type 参数:
6 | ALL: 显示所有开销信息
7 | BLOCK IO: 阻塞的输入输出次数
8 | CONTEXT SWITCHES: 上下文切换相关开销信息
9 | CPU: 显示 CPU 的相关开销信息
10 | IPC: 接收和发送消息的相关开销信息
11 | MEMORY : 显示内存相关的开销，目前无用
12 | PAGE FAULTS : 显示页面错误相关开销信息
13 | SOURCE : 列出相应操作对应的函数名及其在源码中的调用位置 (行数)
```



值得注意的是，MySQL 是在 5.0.37 版本之后才支持 Show Profile 功能的，如果你不太确定的话，可以通过 select @@have_profiling 查询是否支持该功能，如下图所示：

A screenshot of the MySQL Workbench interface. At the top, there's a toolbar with a left arrow, a right arrow, and a magnifying glass icon. Below the toolbar is a menu bar with 'File', 'Edit', 'View', 'Tools', 'Help'. The main area has tabs for 'Information Schema', 'Result 1', 'Summary', and 'Status'. The 'Result 1' tab is selected, showing the query 'select @@have_profiling;'. The results table below it has four columns: 'Column1', 'Column2', 'Column3', and 'Column4'. There is one row with the value 'YES' in the first column.

最新的 MySQL 版本是默认开启 Show Profile 功能的，但在之前的旧版本中是默认关闭该功能的，你可以通过 set 语句在 Session 级别开启该功能：

A screenshot of the MySQL Workbench interface. At the top, there's a toolbar with a left arrow, a right arrow, and a magnifying glass icon. Below the toolbar is a menu bar with 'File', 'Edit', 'View', 'Tools', 'Help'. The main area has tabs for 'Information Schema', 'Result 1', 'Summary', and 'Status'. The 'Result 1' tab is selected, showing the query 'select @@profiling;'. The results table below it has four columns: 'Column1', 'Column2', 'Column3', and 'Column4'. There is one row with the value '1' in the first column.

Show Profiles 只显示最近发给服务器的 SQL 语句，默认情况下是记录最近已执行的 15 条记录，我们可以重新设置 profiling_history_size 增大该存储记录，最大值为 100。

```
1 SELECT COUNT(*) FROM `order`;  
2 show profiles;
```

信息	Result 1	概况	状态
Query_ID	Duration	Query	
1161	0.002482	SHOW STATUS	
1162	0.0020535	SHOW STATUS	
1163	0.001888	SHOW STATUS	
1164	0.0019845	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATION FROM `INFORMATION_SCHEMA`.`PROCESSLIST` GROUP BY QUERY_ID ORDER BY SUM_DURATION DESC LIMIT 1	
1165	0.00190325	SELECT STATE AS `Status`, ROUND(SUM(DURATION),7) AS SUM_DURATION FROM `INFORMATION_SCHEMA`.`PROCESSLIST` GROUP BY STATE ORDER BY SUM_DURATION DESC LIMIT 1	
1166	0.000205	SET PROFILING = 1	
1167	0.001532	SHOW STATUS	
1168	0.0014975	SHOW STATUS	
1169	0.00044325	SELECT COUNT(*) FROM `order`	
1170	0.00127325	SHOW STATUS	
1171	0.00124975	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATION FROM `INFORMATION_SCHEMA`.`PROCESSLIST` GROUP BY QUERY_ID ORDER BY SUM_DURATION DESC LIMIT 1	
1172	0.0010205	SELECT STATE AS `Status`, ROUND(SUM(DURATION),7) AS SUM_DURATION FROM `INFORMATION_SCHEMA`.`PROCESSLIST` GROUP BY STATE ORDER BY SUM_DURATION DESC LIMIT 1	
1173	0.0003905	SET PROFILING = 1	
1174	0.00254275	SHOW STATUS	
1175	0.001943	SHOW STATUS	

获取到 Query_ID 之后，我们再通过 Show Profile for Query ID 语句，就能够查看到对应 Query_ID 的 SQL 语句在执行过程中线程的每个状态所消耗的时间了：

```
1 show profile for query 1198;
```

信息	Result 1	概况	状态
Status		Duration	
▶ starting		0.00009	
checking permissions		0.000011	
Opening tables		0.000031	
init		0.000011	
System lock		0.000014	
optimizing		0.000009	
statistics		0.00002	
preparing		0.000017	
executing		0.000006	
Sending data		0.000053	
end		0.000008	
query end		0.000007	
waiting for handler comm		0.00001	
query end		0.000009	
closing tables		0.000011	
freeing items		0.000052	
cleaning up		0.000016	

通过以上分析可知：SELECT COUNT(*) FROM `order`; SQL 语句在 Sending data 状态所消耗的时间最长，这是因为在该状态下，MySQL 线程开始读取数据并返回到客户端，此时有大量磁盘 I/O 操作。

常用的 SQL 优化

在使用一些常规的 SQL 时，如果我们通过一些方法和技巧来优化这些 SQL 的实现，在性能上就会比使用常规通用的实现方式更加优越，甚至可以将 SQL 语句的性能提升到另一个数量级。

1. 优化分页查询

通常我们是使用 <LIMIT M,N> + 合适的 order by 来实现分页查询，这种实现方式在没有任何索引条件支持的情况下，需要做大量的文件排序操作（file sort），性能将会非常糟糕。如果有对应的索引，通常刚开始的分页查询效率会比较理想，但越往后，分页查询的性能就越差。

这是因为我们在使用 LIMIT 的时候，偏移量 M 在分页越靠后的时候，值就越大，数据库检索的数据也就越多。例如 LIMIT 10000,10 这样的查询，数据库需要查询 10010 条记录，

最后返回 10 条记录。也就是说将会有 10000 条记录被查询出来没有被使用到。

我们模拟一张 10 万数量级的 order 表，进行以下分页查询：

 复制代码

```
1 select * from `demo`.`order` order by order_no limit 10000, 20;
```

通过 EXPLAIN 分析可知：该查询使用到了索引，扫描行数为 10020 行，但所用查询时间为 0.018s，相对来说时间偏长了。

```
1 select * from `demo`.`order` order by order_no limit 10000, 20;
2
3
```

信息 Explain 1 Result 1 概况 状态											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	index	(Null)	idx_order	5	(Null)	10020	100	(Null)

```
1 select * from `demo`.`order` order by order_no limit 10000, 20;
2
3
```

信息 Explain 1 Result 1 概况 状态

```
select * from `demo`.`order` order by order_no limit 10000, 20
OK
时间: 0.018s
```

利用子查询优化分页查询

以上分页查询的问题在于，我们查询获取的 10020 行数据结果都返回给我们了，我们能否先查询出所需要的 20 行数据中的最小 ID 值，然后通过偏移量返回所需要的 20 行数据给我们呢？我们可以通过索引覆盖扫描，使用子查询的方式来实现分页查询：

 复制代码

```
1 select * from `demo`.`order` where id > (select id from `demo`.`order` order by order_no
```

通过 EXPLAIN 分析可知：子查询遍历索引的范围跟上一个查询差不多，而主查询扫描了更多的行数，但执行时间却减少了，只有 0.004s。这就是因为返回行数只有 20 行了，执行效率得到了明显的提升。

```
1 select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1) limit 20;
2
```

信息	Explain 1	Result 1	概况	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶ 1	PRIMARY	order	(Null)	range	PRIMARY	PRIMARY	4	(Null)	90409	100	Using where
2	SUBQUERY	order	(Null)	index	(Null)	idx_order	5	(Null)	10001	100	Using index

```
1 select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1) limit 20;
2
```

信息	Explain 1	Result 1	概况	状态
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1) limit 20				
OK				
时间: 0.004s				

2. 优化 SELECT COUNT(*)

COUNT() 是一个聚合函数，主要用来统计行数，有时候也用来统计某一列的行数量（不统计 NULL 值的行）。我们平时最常用的就是 COUNT(*) 和 COUNT(1) 这两种方式了，其实两者没有明显的区别，在拥有主键的情况下，它们都是利用主键列实现了行数的统计。

但 COUNT() 函数在 MyISAM 和 InnoDB 存储引擎所执行的原理是不一样的，通常在没有任何查询条件下的 COUNT(*)，MyISAM 的查询速度要明显快于 InnoDB。

这是因为 MyISAM 存储引擎记录的是整个表的行数，在 COUNT(*) 查询操作时无需遍历表计算，直接获取该值即可。而在 InnoDB 存储引擎中就需要扫描表来统计具体的行数。而当带上 where 条件语句之后，MyISAM 跟 InnoDB 就没有区别了，它们都需要扫描表来进行行数的统计。

如果对一张大表经常做 SELECT COUNT(*) 操作，这肯定是不明智的。那么我们该如何对大表的 COUNT() 进行优化呢？

使用近似值

有时候某些业务场景并不需要返回一个精确的 COUNT 值，此时我们可以使用近似值来代替。我们可以使用 EXPLAIN 对表进行估算，要知道，执行 EXPLAIN 并不会真正去执行查询，而是返回一个估算的近似值。

增加汇总统计

如果需要一个精确的 COUNT 值，我们可以额外新增一个汇总统计表或者缓存字段来统计需要的 COUNT 值，这种方式在新增和删除时有一定的成本，但却可以大大提升 COUNT()

的性能。

3. 优化 SELECT *

我曾经看过很多同事习惯在只查询一两个字段时，都使用 select * from table where xxx 这样的 SQL 语句，这种写法在特定的环境下会存在一定的性能损耗。

MySQL 常用的存储引擎有 MyISAM 和 InnoDB，其中 InnoDB 在默认创建主键时会创建主键索引，而主键索引属于聚族索引，即在存储数据时，索引是基于 B + 树构成的，具体的行数据则存储在叶子节点。

而 MyISAM 默认创建的主键索引、二级索引以及 InnoDB 的二级索引都属于非聚族索引，即在存储数据时，索引是基于 B + 树构成的，而叶子节点存储的是主键值。

假设我们的订单表是基于 InnoDB 存储引擎创建的，且存在 order_no、status 两列组成的组合索引。此时，我们需要根据订单号查询一张订单表的 status，如果我们使用 select * from order where order_no='xxx' 来查询，则先会查询组合索引，通过组合索引获取到主键 ID，再通过主键 ID 去主键索引中获取对应行所有列的值。

如果我们使用 select order_no, status from order where order_no='xxx' 来查询，则只会查询组合索引，通过组合索引获取到对应的 order_no 和 status 的值。如果你对这些索引还不够熟悉，请重点关注之后的第 34 讲，那一讲会详述数据库索引的相关内容。

总结

在开发中，我们要尽量写出高性能的 SQL 语句，但也无法避免一些慢 SQL 语句的出现，或因为疏漏，或因为实际生产环境与开发环境有所区别，这些都是诱因。面对这种情况，我们可以打开慢 SQL 配置项，记录下都有哪些 SQL 超过了预期的最大执行时间。首先，我们可以通过以下命令行查询是否开启了记录慢 SQL 的功能，以及最大的执行时间是多少：

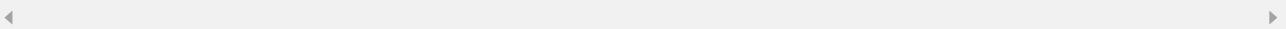
 复制代码

```
1 Show variables like 'slow_query%';
2 Show variables like 'long_query_time';
```

如果没有开启，我们可以通过以下设置来开启：

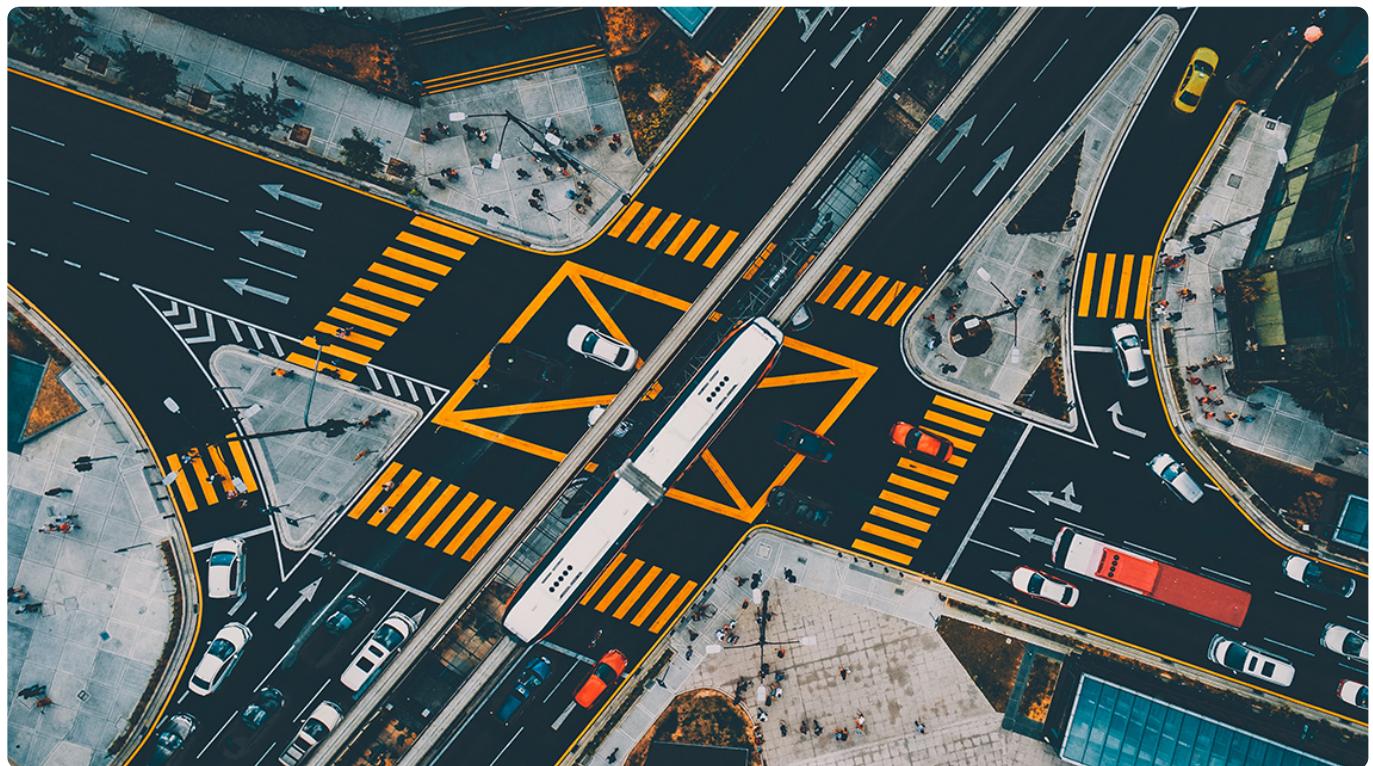
 复制代码

```
1 set global slow_query_log='ON'; // 开启慢 SQL 日志
2 set global slow_query_log_file='/var/lib/mysql/test-slow.log';// 记录日志地址
3 set global long_query_time=1;// 最大执行时间
```



除此之外，很多数据库连接池中间件也有分析慢 SQL 的功能。总之，我们要在编程中避免低性能的 SQL 操作出现，除了要具备一些常用的 SQL 优化技巧之外，还要充分利用一些 SQL 工具，实现 SQL 性能分析与监控。

33 | MySQL调优之事务：高并发场景下的数据库事务调优



数据库事务是数据库系统执行过程中的一个逻辑处理单元，保证一个数据库操作要么成功，要么失败。谈到他，就不得不提 ACID 属性了。数据库事务具有以下四个基本属性：原子性（Atomicity）、一致性（Consistent）、隔离性（Isolation）以及持久性（Durable）。正是这些特性，才保证了数据库事务的安全性。而在 MySQL 中，鉴于 MyISAM 存储引擎不支持事务，所以接下来的内容都是在 InnoDB 存储引擎的基础上进行讲解的。

我们知道，在 Java 并发编程中，可以多线程并发执行程序，然而并发虽然提高了程序的执行效率，却给程序带来了线程安全问题。事务跟多线程一样，为了提高数据库处理事务的吞

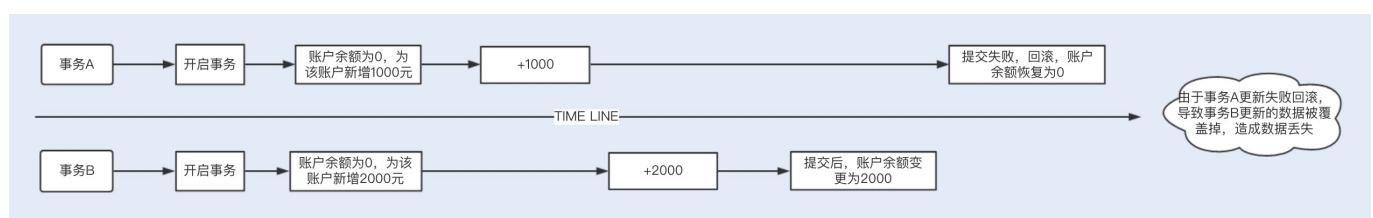
吐量，数据库同样支持并发事务，而在并发运行中，同样也存在着安全性问题，例如，修改数据丢失，读取数据不一致等。

在数据库事务中，事务的隔离是解决并发事务问题的关键，今天我们就重点了解下事务隔离的实现原理，以及如何优化事务隔离带来的性能问题。

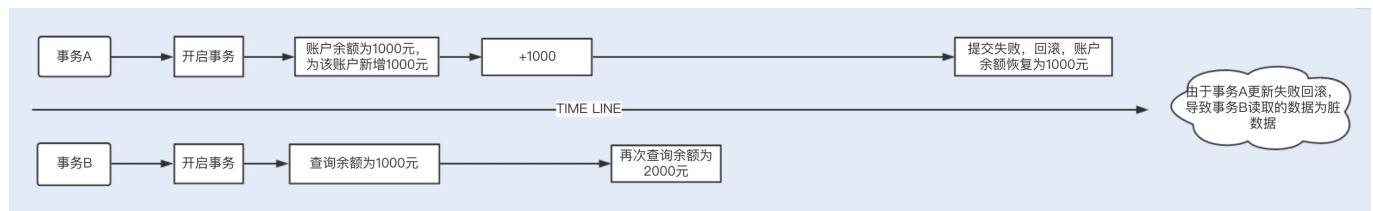
并发事务带来的问题

我们可以通过以下几个例子来了解下并发事务带来的几个问题：

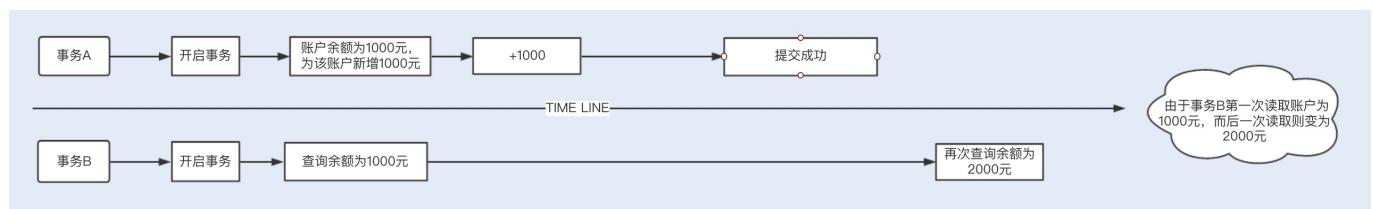
1. 数据丢失



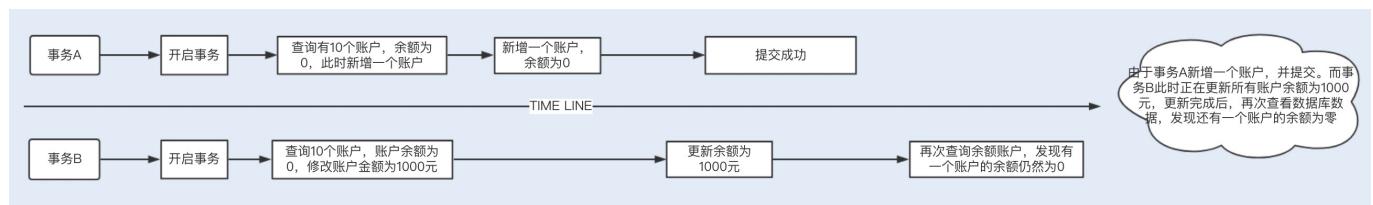
2. 脏读



3. 不可重复读



4. 幻读



事务隔离解决并发问题

以上 4 个并发事务带来的问题，其中，数据丢失可以基于数据库中的悲观锁来避免发生，即在查询时通过在事务中使用 select xx for update 语句来实现一个排他锁，保证在该事务结束之前其他事务无法更新该数据。

当然，我们也可以基于乐观锁来避免，即将某一字段作为版本号，如果更新时的版本号跟之前的版本一致，则更新，否则更新失败。剩下 3 个问题，其实是数据库读一致性造成的，需要数据库提供一定的事务隔离机制来解决。

我们通过加锁的方式，可以实现不同的事务隔离机制。在了解事务隔离机制之前，我们不妨先来了解下 MySQL 都有哪些锁机制。

InnoDB 实现了两种类型的锁机制：共享锁（S）和排他锁（X）。共享锁允许一个事务读数据，不允许修改数据，如果其他事务要再对该行加锁，只能加共享锁；排他锁是修改数据时加的锁，可以读取和修改数据，一旦一个事务对该行数据加锁，其他事务将不能再对该数据加任务锁。

熟悉了以上 InnoDB 行锁的实现原理，我们就可以更清楚地理解下面的内容。

在操作数据的事务中，不同的锁机制会产生以下几种不同的事务隔离级别，不同的隔离级别分别可以解决并发事务产生的几个问题，对应如下：

未提交读（Read Uncommitted）：在事务 A 读取数据时，事务 B 读取和修改数据加了共享锁。这种隔离级别，会导致脏读、不可重复读以及幻读。

已提交读（Read Committed）：在事务 A 读取数据时增加了共享锁，一旦读取，立即释放锁，事务 B 读取修改数据时增加了行级排他锁，直到事务结束才释放锁。也就是说，事务 A 在读取数据时，事务 B 只能读取数据，不能修改。当事务 A 读取到数据后，事务 B 才能修改。这种隔离级别，可以避免脏读，但依然存在不可重复读以及幻读的问题。

可重复读（Repeatable Read）：在事务 A 读取数据时增加了共享锁，事务结束，才释放锁，事务 B 读取修改数据时增加了行级排他锁，直到事务结束才释放锁。也就是说，事务 A 在没有结束事务时，事务 B 只能读取数据，不能修改。当事务 A 结束事务，事务 B 才能修改。这种隔离级别，可以避免脏读、不可重复读，但依然存在幻读的问题。

可序列化 (Serializable)：在事务 A 读取数据时增加了共享锁，事务结束，才释放锁，事务 B 读取修改数据时增加了表级排他锁，直到事务结束才释放锁。可序列化解决了脏读、不可重复读、幻读等问题，但隔离级别越来越高的同时，并发性会越来越低。

InnoDB 中的 RC 和 RR 隔离事务是基于多版本并发控制 (MVVC) 实现高性能事务。一旦数据被加上排他锁，其他事务将无法加入共享锁，且处于阻塞等待状态，如果一张表有大量的请求，这样的性能将是无法支持的。

MVVC 对普通的 Select 不加锁，如果读取的数据正在执行 Delete 或 Update 操作，这时读取操作不会等待排它锁的释放，而是直接利用 MVVC 读取该行的数据快照（数据快照是指在该行的之前版本的数据，而数据快照的版本是基于 undo 实现的，undo 是用来做事务回滚的，记录了回滚的不同版本的行记录）。**MVVC 避免了对数据重复加锁的过程，大大提高了读操作的性能。**

锁具体实现算法

我们知道，InnoDB 既实现了行锁，也实现了表锁。行锁是通过索引实现的，如果不通过索引条件检索数据，那么 InnoDB 将对表中所有的记录进行加锁，其实就是升级为表锁了。

行锁的具体实现算法有三种：record lock、gap lock 以及 next-key lock。record lock 是专门对索引项加锁；gap lock 是对索引项之间的间隙加锁；next-key lock 则是前面两种的组合，对索引项以其之间的间隙加锁。

只在可重复读或以上隔离级别的特定操作才会取得 gap lock 或 next-key lock，在 Select、Update 和 Delete 时，除了基于唯一索引的查询之外，其他索引查询时都会获取 gap lock 或 next-key lock，即锁住其扫描的范围。

优化高并发事务

通过以上讲解，相信你对事务、锁以及隔离级别已经有了一个透彻的了解了。清楚了问题，我们就可以聊聊高并发场景下的事务到底该如何调优了。

1. 结合业务场景，使用低级别事务隔离

在高并发业务中，为了保证业务数据的一致性，操作数据库时往往会展开到不同级别的事务隔离。隔离级别越高，并发性能就越低。

那换到业务场景中，我们如何判断用哪种隔离级别更合适呢？我们可以通过两个简单的业务来说下其中的选择方法。

我们在修改用户最后登录时间的业务场景中，这里对查询用户的登录时间没有特别严格准确性要求，而修改用户登录信息只有用户自己登录时才会修改，不存在一个事务提交的信息被覆盖的可能。所以我们允许该业务使用最低隔离级别。

而如果是账户中的余额或积分的消费，就存在多个客户端同时消费一个账户的情况，此时我们应该选择 RR 级别来保证一旦有一个客户端在对账户进行消费，其他客户端就不可能对该账户同时进行消费了。

2. 避免行锁升级表锁

前面讲了，在 InnoDB 中，行锁是通过索引实现的，如果不通过索引条件检索数据，行锁将会升级到表锁。我们知道，表锁是会严重影响到整张表的操作性能的，所以我们应该避免他。

3. 控制事务的大小，减少锁定的资源量和锁定时间长度

你是否遇到过以下 SQL 异常呢？在抢购系统的日志中，在活动区间，我们经常可以看到这种异常日志：

 复制代码

```
1 MySQLQueryInterruptedException: Query execution was interrupted
```

由于在抢购提交订单中开启了事务，在高并发时对一条记录进行更新的情况下，由于更新记录所在的事务还可能存在其他操作，导致一个事务比较长，当有大量请求进入时，就可能导致一些请求同时进入到事务中。

又因为锁的竞争是不公平的，当多个事务同时对一条记录进行更新时，极端情况下，一个更新操作进去排队系统后，可能会一直拿不到锁，最后因超时被系统打断踢出。

在用户购买商品时，首先我们需要查询库存余额，再新建一个订单，并扣除相应的库存。这一系列操作是处于同一个事务的。

以上业务若是在两种不同的执行顺序下，其结果都是一样的，但在事务性能方面却不一样：

执行顺序1	执行顺序2
1. 开启事务 2. 查询库存，判断库存是否满足 3. 新建订单 4. 扣除库存 5. 提交或回滚	1. 开启事务 2. 查询库存，判断库存是否满足 3. 扣除库存 4. 新建订单 5. 提交或回滚

这是因为，虽然这些操作在同一个事务，但锁的申请在不同时间，只有当其他操作都执行完，才会释放所有锁。因为扣除库存是更新操作，属于行锁，这将会影响到其他操作该数据的事务，所以我们应该尽量避免长时间地持有该锁，尽快释放该锁。

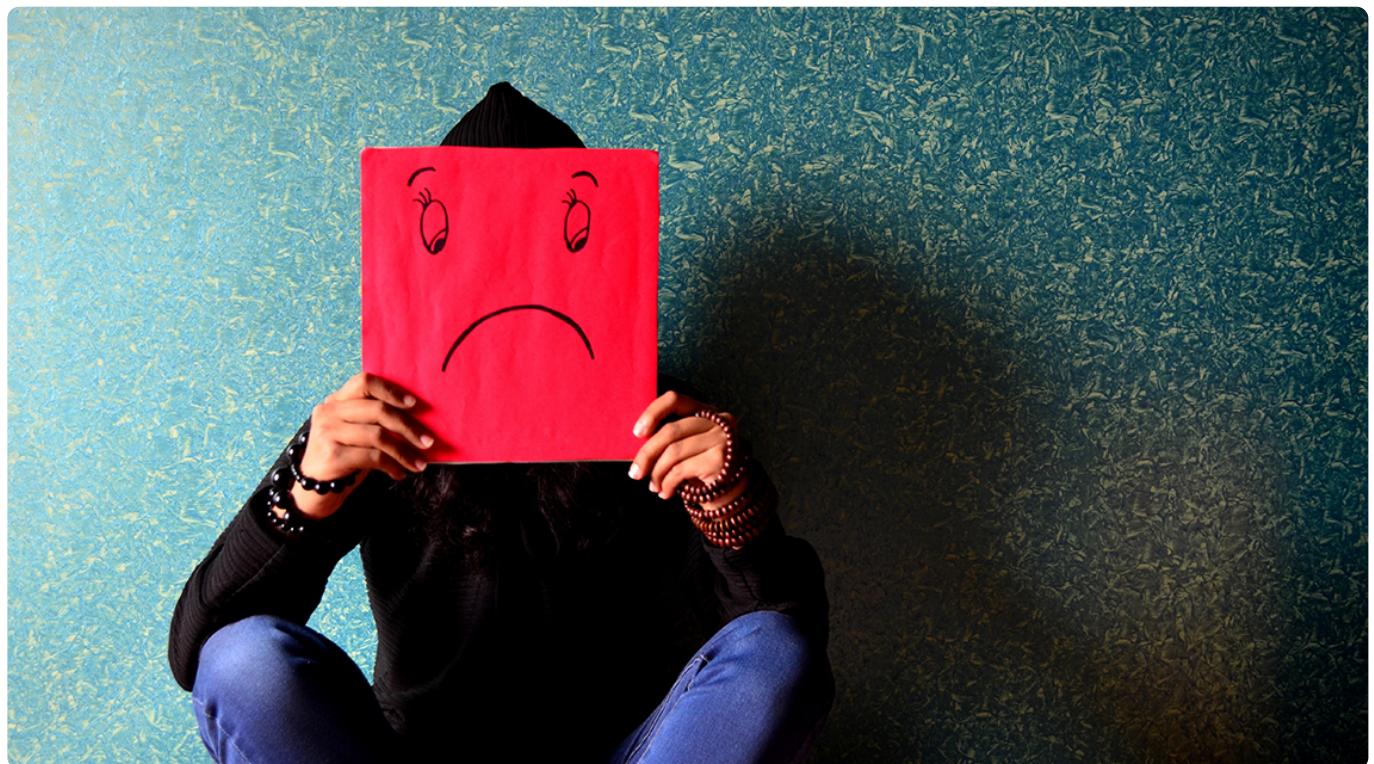
又因为先新建订单和先扣除库存都不会影响业务，所以我们可以将扣除库存操作放到最后，也就是使用执行顺序 1，以此尽量减小锁的持有时间。

总结

其实 MySQL 的并发事务调优和 Java 的多线程编程调优非常类似，都是可以通过减小锁粒度和减少锁的持有时间进行调优。**在 MySQL 的并发事务调优中，我们尽量在可以使用低事务隔离级别的业务场景中，避免使用高事务隔离级别。**

在功能业务开发时，开发人员往往会为了追求开发速度，习惯使用默认的参数设置来实现业务功能。例如，在 service 方法中，你可能习惯默认使用 transaction，很少再手动变更事务隔离级别。但要知道，transaction 默认是 RR 事务隔离级别，在某些业务场景下，可能并不合适。因此，我们还是要结合具体的业务场景，进行考虑。

34 | MySQL调优之索引：索引的失效与优化



不知道你是否跟我有过同样的经历，那就是作为一个开发工程师，经常被 DBA 叫过去“批评”，而最常见的就是申请创建新的索引或发现慢 SQL 日志了。

记得之前有一次迭代一个业务模块的开发，涉及到了一个新的查询业务，需要根据商品类型、订单状态筛选出需要的订单，并以订单时间进行排序。由于 sku 的索引已经存在了，我在完成业务开发之后，提交了一个创建 status 的索引的需求，理由是 SQL 查询需要使用到这两个索引：

```
select * from order where status = 1 and sku=10001 order by  
create_time asc
```

然而，DBA 很快就将这个需求驳回了，并给出了重建一个 sku、status 以及 create_time 组合索引的建议，查询顺序也改成了 sku=10001 and status=1。当时我是知道为什么要重建组合索引，但却无法理解为什么要添加 create_time 这列进行组合。

从执行计划中，我们可以发现使用到了索引，那为什么 DBA 还要求将 create_time 这一列加入到组合索引中呢？这个问题我们在[第 32 讲](#)中提到过，相信你也已经知道答案了。通过故事我们可以发现索引知识在平时开发时的重要性，然而它又很容易被我们忽略，所以今天我们就来详细聊一聊索引。

MySQL 索引存储结构

索引是优化数据库查询最重要的方式之一，它是在 MySQL 的存储引擎层中实现的，所以每一种存储引擎对应的索引不一定相同。我们可以通过下面这张表格，看看不同的存储引擎分别支持哪种索引类型：

索引类型	MyISAM引擎	InnoDB引擎	Memory引擎
B+Tree索引	yes	yes	yes
HASH索引	no	no	yes
R-Tree索引	yes	no	no
Full-Text索引	yes	no	no

B+Tree 索引和 Hash 索引是我们比较常用的两个索引数据存储结构，B+Tree 索引是通过 B+ 树实现的，是有序排列存储，所以在排序和范围查找方面都比较有优势。如果你对 B+Tree 索引不够了解，可以通过该[链接](#)了解下它的数据结构原理。

Hash 索引相对简单些，只有 Memory 存储引擎支持 Hash 索引。Hash 索引适合 key-value 键值对查询，无论表数据多大，查询数据的复杂度都是 O(1)，且直接通过 Hash 索引查询的性能比其它索引都要优越。

在创建表时，无论使用 InnoDB 还是 MyISAM 存储引擎，默认都会创建一个主键索引，而创建的主键索引默认使用的是 B+Tree 索引。不过虽然这两个存储引擎都支持 B+Tree 索引，但它们在具体的数据存储结构方面却有所不同。

InnoDB 默认创建的主键索引是聚族索引（Clustered Index），其它索引都属于辅助索引（Secondary Index），也被称为二级索引或非聚族索引。接下来我们通过一个简单的例

子，说明下这两种索引在存储数据中的具体实现。

首先创建一张商品表，如下：

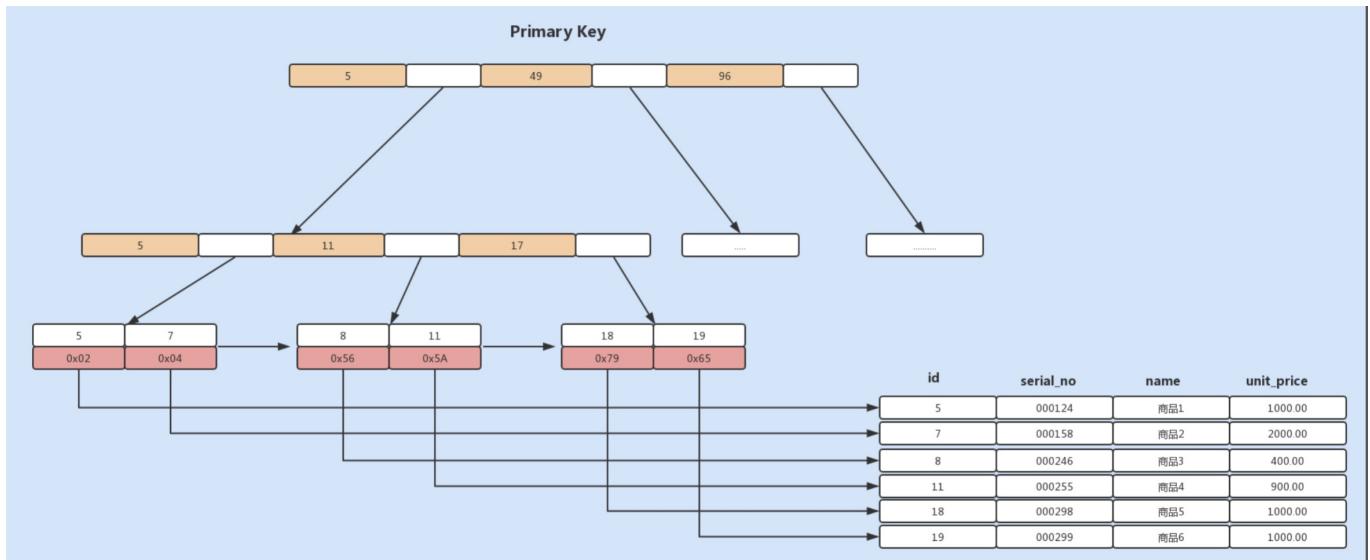
 复制代码

```
1 CREATE TABLE `merchandise` (
2   `id` int(11) NOT NULL,
3   `serial_no` varchar(20) DEFAULT NULL,
4   `name` varchar(255) DEFAULT NULL,
5   `unit_price` decimal(10, 2) DEFAULT NULL,
6   PRIMARY KEY (`id`) USING BTREE
7 ) CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

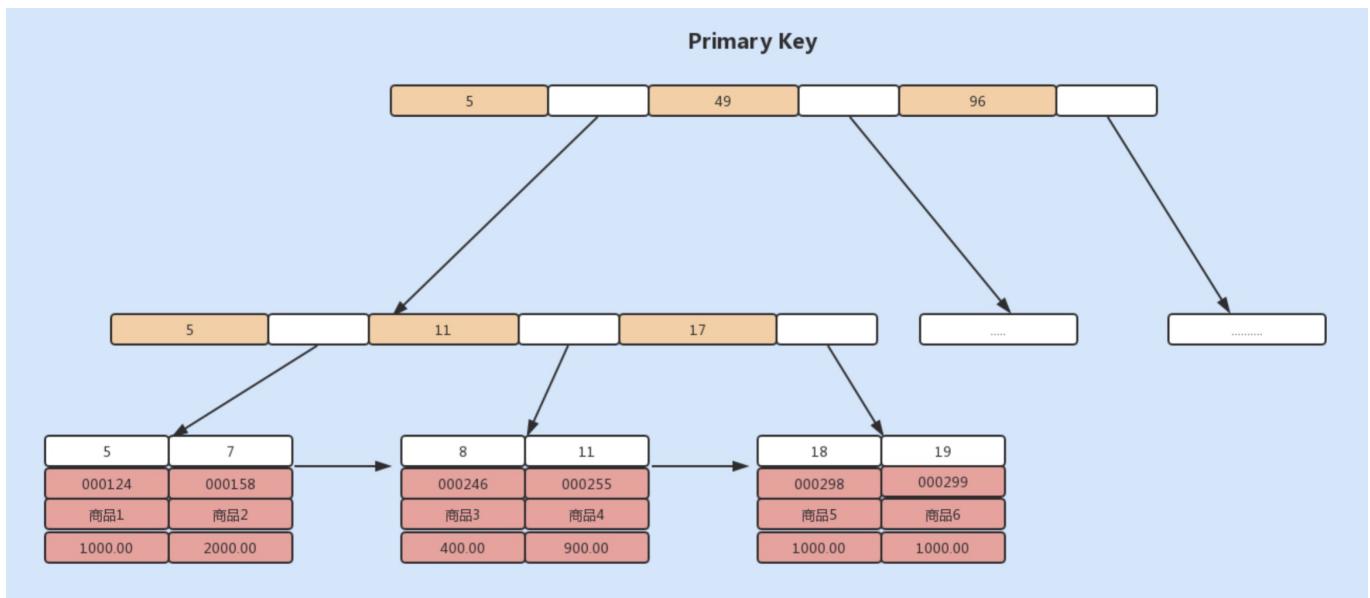
然后新增了以下几行数据，如下：

id	serial_no	name	unit_price
5	000124	商品1	1000.00
7	000158	商品2	2000.00
8	000246	商品3	400.00
11	000255	商品4	900.00
14	000298	商品5	1000.00

如果我们使用的是 MyISAM 存储引擎，由于 MyISAM 使用的是辅助索引，索引中每一个叶子节点仅仅记录的是每行数据的物理地址，即行指针，如下图所示：



如果我们使用的是 InnoDB 存储引擎，由于 InnoDB 使用的是聚族索引，聚族索引中的叶子节点则记录了主键值、事务 id、用于事务和 MVVC 的回流指针以及所有的剩余列，如下图所示：



基于上面的图示，如果我们需要根据商品编码查询商品，我们就需要将商品编码 serial_no 列作为一个索引列。此时创建的索引是一个辅助索引，与 MyISAM 存储引擎的主键索引的存储方式是一致的，但叶子节点存储的就不是行指针了，而是主键值，并以此来作为指向行的指针。这样的好处就是当行发生移动或者数据分裂时，不用再维护索引的变更。

如果我们使用主键索引查询商品，则会按照 B+ 树的索引找到对应的叶子节点，直接获取到行数据：

```
select * from merchandise where id=7
```

如果我们使用商品编码查询商品，即使用辅助索引进行查询，则会先检索辅助索引中的 B+ 树的 serial_no，找到对应的叶子节点，获取主键值，然后再通过聚族索引中的 B+ 树检索到对应的叶子节点，然后获取整行数据。这个过程叫做回表。

在了解了索引的实现原理后，我们再来详细了解下平时建立和使用索引时，都有哪些调优方法呢？

1. 覆盖索引优化查询

假设我们只需要查询商品的名称、价格信息，我们有什么方式来避免回表呢？我们可以建立一个组合索引，即商品编码、名称、价格作为一个组合索引。如果索引中存在这些数据，查询将不会再次检索主键索引，从而避免回表。

从辅助索引中查询得到记录，而不需要通过聚族索引查询获得，MySQL 中将其称为覆盖索引。使用覆盖索引的好处很明显，我们不需要查询出包含整行记录的所有信息，因此可以减少大量的 I/O 操作。

通常在 InnoDB 中，除了查询部分字段可以使用覆盖索引来优化查询性能之外，统计数量也会用到。例如，在[第 32 讲](#)我们讲 SELECT COUNT(*) 时，如果不存在辅助索引，此时会通过查询聚族索引来统计行数，如果此时正好存在一个辅助索引，则会通过查询辅助索引来统计行数，减少 I/O 操作。

通过 EXPLAIN，我们可以看到 InnoDB 存储引擎使用了 idx_order 索引列来统计行数，如下图所示：

1	select count(*) from `demo`.`order`
2	
	
id	select_type
1	SIMPLE
table	table
order	(Null)
partitions	partitions
(Null)	(Null)
type	type
index	index
possible_keys	possible_keys
(Null)	(Null)
key	key
idx_order	idx_order
key_len	key_len
6	6
ref	ref
(Null)	(Null)
rows	rows
180819	180819
filtered	filtered
100	100
Extra	Using index

2. 自增字段作主键优化查询

上面我们讲了 InnoDB 创建主键索引默认为聚族索引，数据被存放在了 B+ 树的叶子节点上。也就是说，同一个叶子节点内的各个数据是按主键顺序存放的，因此，每当有一条新的数据插入时，数据库会根据主键将其插入到对应的叶子节点中。

如果我们使用自增主键，那么每次插入的新数据就会按顺序添加到当前索引节点的位置，不需要移动已有的数据，当页面写满，就会自动开辟一个新页面。因为不需要重新移动数据，因此这种插入数据的方法效率非常高。

如果我们使用非自增主键，由于每次插入主键的索引值都是随机的，因此每次插入新的数据时，就可能会插入到现有数据页中间的某个位置，这将不得不移动其它数据来满足新数据的插入，甚至需要从一个页面复制数据到另外一个页面，我们通常将这种情况称为页分裂。页分裂还有可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率。

因此，在使用 InnoDB 存储引擎时，如果没有特别的业务需求，建议使用自增字段作为主键。

3. 前缀索引优化

前缀索引顾名思义就是使用某个字段中字符串的前几个字符建立索引，那我们为什么需要使用前缀来建立索引呢？

我们知道，索引文件是存储在磁盘中的，而磁盘中最小分配单元是页，通常一个页的默认大小为 16KB，假设我们建立的索引的每个索引值大小为 2KB，则在一个页中，我们能记录 8 个索引值，假设我们有 8000 行记录，则需要 1000 个页来存储索引。如果我们使用该索引查询数据，可能需要遍历大量页，这显然会降低查询效率。

减小索引字段大小，可以增加一个页中存储的索引项，有效提高索引的查询速度。在一些大字符串的字段作为索引时，使用前缀索引可以帮助我们减小索引项的大小。

不过，前缀索引是有一定的局限性的，例如 `order by` 就无法使用前缀索引，无法把前缀索引用作覆盖索引。

4. 防止索引失效

当我们习惯建立索引来实现查询 SQL 的性能优化后，是不是就万事大吉了呢？当然不是，有时候我们看似使用到了索引，但实际上并没有被优化器选择使用。

对于 Hash 索引实现的列，如果使用到范围查询，那么该索引将无法被优化器使用到。也就是说 Memory 引擎实现的 Hash 索引只有在 “=” 的查询条件下，索引才会生效。我们将

order 表设置为 Memory 存储引擎，分析查询条件为 id<10 的 SQL，可以发现没有使用到索引。

1	EXPLAIN SELECT * FROM `order` where id < 10;
2	

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	581	33.33	Using where

如果是以 % 开头的 LIKE 查询将无法利用节点查询数据：

1	EXPLAIN SELECT * FROM `order` where order_no like '%1'
2	

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	581	11.11	Using where

当我们在使用复合索引时，需要使用索引中的最左边的列进行查询，才能使用到复合索引。例如我们在 order 表中建立一个复合索引 idx_user_order_status(order_no, status, user_id)，如果我们使用 order_no、order_no+status、order_no+status+user_id 以及 order_no+user_id 组合查询，则能利用到索引；而如果我们用 status、status+user_id 查询，将无法使用到索引，这也是我们经常听过的最左匹配原则。

1	EXPLAIN SELECT * FROM `order` where order_no='1' and user_id=1
2	

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ref	idx_order, idx_user_order	idx_order	5	const	1	10	Using where

1	EXPLAIN SELECT * FROM `order` where status=1 and user_id=1
2	

信息	Result 1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	581	1	Using where

如果查询条件中使用 or，且 or 的前后条件中有一个列没有索引，那么涉及的索引都不会被使用到。

```
1 EXPLAIN SELECT * FROM `order` where order_no = '1' or create_date = '';  
2
```

信息	Result 1	概况	状态									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	order	(Null)	ALL	idx_order	(Null)	(Null)	(Null)	581	19	Using where	

所以，你懂了吗？作为一名开发人员，如果没有熟悉 MySQL，特别是 MySQL 索引的基础知识，很多时候都将被 DBA 批评到怀疑人生。

总结

在大多数情况下，我们习惯使用默认的 InnoDB 作为表存储引擎。在使用 InnoDB 作为存储引擎时，创建的索引默认为 B+ 树数据结构，如果是主键索引，则属于聚簇索引，非主键索引则属于辅助索引。基于主键查询可以直接获取到行信息，而基于辅助索引作为查询条件，则需要进行回表，然后再通过主键索引获取到数据。

如果只是查询一列或少部分列的信息，我们可以基于覆盖索引来避免回表。覆盖索引只需要读取索引，且由于索引是顺序存储，对于范围或排序查询来说，可以极大地减少磁盘 I/O 操作。

除了了解索引的具体实现和一些特性，我们还需要注意索引失效的情况发生。如果觉得这些规则太多，难以记住，我们就要养成经常检查 SQL 执行计划的习惯。

35 | 记一次线上SQL死锁事故：如何避免死锁？



之前我参与过一个项目，在项目初期，我们是没有将读写表分离的，而是基于一个主库完成读写操作。在业务量逐渐增大的时候，我们偶尔会收到系统的异常报警信息，DBA 通知我们数据库出现了死锁异常。

按理说业务开始是比较简单的，就是新增订单、修改订单、查询订单等操作，那为什么会出现死锁呢？经过日志分析，我们发现是作为幂等性校验的一张表经常出现死锁异常。我们和 DBA 讨论之后，初步怀疑是索引导致的死锁问题。后来我们在开发环境中模拟了相关操作，果然重现了该死锁异常。

接下来我们就通过实战来重现下该业务死锁异常。首先，创建一张订单记录表，该表主要用于校验订单重复创建：

 复制代码

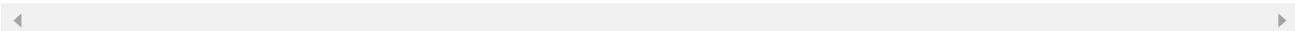
```
1 CREATE TABLE `order_record` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `order_no` int(11) DEFAULT NULL,
4   `status` int(4) DEFAULT NULL,
5   `create_date` datetime(0) DEFAULT NULL,
6   PRIMARY KEY (`id`) USING BTREE,
7   INDEX `idx_order_status`(`order_no`, `status`) USING BTREE
8 ) ENGINE = InnoDB
```



为了能重现该问题，我们先将事务设置为手动提交。这里要注意一下，MySQL 数据库和 Oracle 提交事务不太一样，MySQL 数据库默认情况下是自动提交事务，我们可以通过以下命令行查看自动提交事务是否开启：

 复制代码

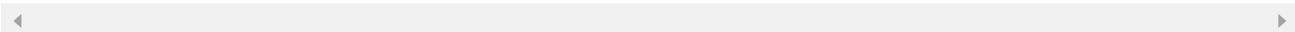
```
1 mysql> show variables like 'autocommit';
2 +-----+-----+
3 | Variable_name | Value |
4 +-----+-----+
5 | autocommit     | ON      |
6 +-----+-----+
7 1 row in set (0.01 sec)
```



下面就操作吧，先将 MySQL 数据库的事务提交设置为手动提交，通过以下命令行可以关闭自动提交事务：

 复制代码

```
1 mysql> set autocommit = 0;
2 Query OK, 0 rows affected (0.00 sec)
```



订单在做幂等性校验时，先是通过订单号检查订单是否存在，如果不存在则新增订单记录。知道具体的逻辑之后，我们再来模拟创建产生死锁的运行 SQL 语句。首先，我们模拟新建

两个订单，并按照以下顺序执行幂等性校验 SQL 语句（垂直方向代表执行的时间顺序）：

事务A	事务B
BEGIN;	BEGIN;
SELECT id FROM `order_record` where `order_no` = 4 for update; //检查是否存在 order_no 等于 4 的订单	
	SELECT id FROM `order_record` where `order_no` = 5 for update; //检查是否存在 order_no 等于 5 的订单
INSERT INTO `order_record`(`order_no`, `status`, `create_date`) VALUES (4, 1, '2019-07-13 10:57:03'); //如果没有，则插入信息	
此时，锁等待中.....	
	INSERT INTO `order_record`(`order_no`, `status`, `create_date`) VALUES (5, 1, '2019-07-13 10:57:03'); //如果没有，则插入信息
	此时，锁等待中.....
COMMIT; (未完成)	COMMIT; (未完成)

此时，我们会发现两个事务已经进入死锁状态。我们可以在 information_schema 数据库中查询到具体的死锁情况，如下图所示：

```
mysql> use information_schema
Database changed
mysql> select * from innodb_lock_waits;
+-----+-----+-----+-----+
| requesting_trx_id | requested_lock_id | blocking_trx_id | blocking_lock_id |
+-----+-----+-----+-----+
| 369579            | 369579:144:5:1  | 369576          | 369576:144:5:1 |
+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> select * from innodb_locks;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| lock_id        | lock_trx_id | lock_mode | lock_type | lock_table      | lock_index | lock_space | lock_page | lock_rec | lock_data |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 369579:144:5:1 | 369579       | X         | RECORD    | `demo`.`order_record` | idx_order  |           144 |           5 |         1 | supremum pseudo-record |
| 369576:144:5:1 | 369576       | X         | RECORD    | `demo`.`order_record` | idx_order  |           144 |           5 |         1 | supremum pseudo-record |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

看到这，你可能会想，为什么 SELECT 要加 for update 排他锁，而不是使用共享锁呢？试想下，如果是两个订单号一样的请求同时进来，就有可能出现幻读。也就是说，一开始事务 A 中的查询没有该订单号，后来事务 B 新增了一个该订单号的记录，此时事务 A 再新增一

一条该订单号记录，就会创建重复的订单记录。面对这种情况，我们可以使用锁间隙算法来防止幻读。

死锁是如何产生的？

上面我们说到了锁间隙，在[第 33 讲](#)中，我已经讲过了并发事务中的锁机制以及行锁的具体实现算法，不妨回顾一下。

行锁的具体实现算法有三种：record lock、gap lock 以及 next-key lock。record lock 是专门对索引项加锁；gap lock 是对索引项之间的间隙加锁；next-key lock 则是前面两种的组合，对索引项以其之间的间隙加锁。

只在可重复读或以上隔离级别的特定操作才会取得 gap lock 或 next-key lock，在 Select、Update 和 Delete 时，除了基于唯一索引的查询之外，其它索引查询时都会获取 gap lock 或 next-key lock，即锁住其扫描的范围。主键索引也属于唯一索引，所以主键索引是不会使用 gap lock 或 next-key lock。

在 MySQL 中，gap lock 默认是开启的，即 innodb_locks_unsafe_for_binlog 参数值是 disable 的，且 MySQL 中默认的是 RR 事务隔离级别。

当我们执行以下查询 SQL 时，由于 order_no 列为非唯一索引，此时又是 RR 事务隔离级别，所以 SELECT 的加锁类型为 gap lock，这里的 gap 范围是 $(4, +\infty)$ 。

```
SELECT id FROM demo.order_record WHERE order_no = 4 FOR  
UPDATE;
```

执行查询 SQL 语句获取的 gap lock 并不会导致阻塞，而当我们执行以下插入 SQL 时，会在插入间隙上再次获取插入意向锁。插入意向锁其实也是一种 gap 锁，它与 gap lock 是冲突的，所以当其它事务持有该间隙的 gap lock 时，需要等待其它事务释放 gap lock 之后，才能获取到插入意向锁。

以上事务 A 和事务 B 都持有间隙 $(4, +\infty)$ 的 gap 锁，而接下来的插入操作为了获取到插入意向锁，都在等待对方事务的 gap 锁释放，于是就造成了循环等待，导致死锁。

```
INSERT INTO demo.order_record(order_no, status, create_date)
VALUES (5, 1, '2019-07-13 10:57:03');
```

我们可以通过以下锁的兼容矩阵图，来查看锁的兼容性：

	Gap	Insert Intention	Record	Next-Key
Gap	兼容	冲突	兼容	兼容
Insert Intention	冲突	兼容	兼容	冲突
Record	兼容	兼容	冲突	冲突
Next-Key	兼容	兼容	冲突	冲突
备注	横向是已经持有的锁，纵向是正在请求的锁。			

避免死锁的措施

知道了死锁问题源自哪儿，就可以找到合适的方法来避免它了。

避免死锁最直观的方法就是在两个事务相互等待时，当一个事务的等待时间超过设置的某一阈值，就对这个事务进行回滚，另一个事务就可以继续执行了。这种方法简单有效，在 InnoDB 中，参数 `innodb_lock_timeout` 是用来设置超时时间的。

另外，我们还可以将 `order_no` 列设置为唯一索引列。虽然不能防止幻读，但我们可以利用它的唯一性来保证订单记录不重复创建，这种方式唯一的缺点就是当遇到重复创建订单时会抛出异常。

我们还可以使用其它的方式来代替数据库实现幂等性校验。例如，使用 Redis 以及 ZooKeeper 来实现，运行效率比数据库更佳。

其它常见的 SQL 死锁问题

这里再补充一些常见的 SQL 死锁问题，以便你遇到时也能知道其原因，从而顺利解决。

我们知道死锁的四个必要条件：互斥、占有且等待、不可强占用、循环等待。只要系统发生死锁，这些条件必然成立。所以在一些经常需要使用互斥共用一些资源，且有可能循环等待的业务场景中，要特别注意死锁问题。

接下来，我们再来了解一个出现死锁的场景。

我们讲过，InnoDB 存储引擎的主键索引为聚簇索引，其它索引为辅助索引。如果使用辅助索引来更新数据库，就需要使用聚簇索引来更新数据库字段。如果两个更新事务使用了不同的辅助索引，或一个使用了辅助索引，一个使用了聚簇索引，就都有可能导致锁资源的循环等待。由于本身两个事务是互斥，也就构成了以上死锁的四个必要条件了。

我们还是以上面的这个订单记录表来重现下聚簇索引和辅助索引更新时，循环等待锁资源导致的死锁问题：

事务A	事务B
BEGIN;	BEGIN;
UPDATE `order_record` SET status = 1 WHERE `order_no` = 4 ;	UPDATE `order_record` SET status = 1 WHERE id = 4 ;

出现死锁的步骤：

事务A	事务B
首先获取idx_order_status非聚族索引	
	获取主键索引的行锁
根据非聚族索引获取的主键，获取主键索引的行锁	
	更新status列时，需要获取idx_order_status 非聚族索引

综上可知，在更新操作时，我们应该尽量使用主键来更新表字段，这样可以有效避免一些不必要的死锁发生。

总结

数据库发生死锁的概率并不是很大，一旦遇到了，就一定要彻查具体原因，尽快找出解决方案，老实说，过程不简单。我们只有先对 MySQL 的 InnoDB 存储引擎有足够的了解，才能剖析出造成死锁的具体原因。

例如，以上我例举的两种发生死锁的场景，一个考验的是我们对锁算法的了解，另外一个考验则是我们对聚簇索引和辅助索引的熟悉程度。

解决死锁的最佳方式当然就是预防死锁的发生了，我们平时编程中，可以通过以下一些常规手段来预防死锁的发生：

1. 在编程中尽量按照固定的顺序来处理数据库记录，假设有两个更新操作，分别更新两条相同的记录，但更新顺序不一样，有可能导致死锁；
2. 在允许幻读和不可重复读的情况下，尽量使用 RC 事务隔离级别，可以避免 gap lock 导致的死锁问题；
3. 更新表时，尽量使用主键更新；
4. 避免长事务，尽量将长事务拆解，可以降低与其它事务发生冲突的概率；
5. 设置锁等待超时参数，我们可以通过 `innodb_lock_wait_timeout` 设置合理的等待超时阈值，特别是在一些高并发的业务中，我们可以尽量将该值设置得小一些，避免大量事务等待，占用系统资源，造成严重的性能开销。

36 | 什么时候需要分表分库？



在当今互联网时代，海量数据基本上是每一个成熟产品的共性，特别是在移动互联网产品中，几乎每天都在产生数据，例如，商城的订单表、支付系统的交易明细以及游戏中的战报等等。

对于一个日活用户在百万数量级的商城来说，每天产生的订单数量可能在百万级，特别在一些活动促销期间，甚至上千万。

假设我们基于单表来实现，每天产生上百万的数据量，不到一个月的时间就要承受上亿的数据，这时单表的性能将会严重下降。因为 MySQL 在 InnoDB 存储引擎下创建的索引都是

基于 B+ 树实现的，所以查询时的 I/O 次数很大程度取决于树的高度，随着 B+ 树的树高增高，I/O 次数增加，查询性能也就越差。

当我们面对一张海量数据的表时，通常有分区、NoSQL 存储、分表分库等优化方案。

分区的底层虽然也是基于分表的原理实现的，即有多个底层表实现，但分区依然是在单库下进行的，在一些需要提高并发的场景中的优化空间非常有限，且一个表最多只能支持 1024 个分区。面对日益增长的海量数据，优化存储能力有限。不过在一些非海量数据的大表中，我们可以考虑使用分区来优化表性能。

分区表是由多个相关的底层表实现的，这些底层表也是由句柄对象表示，所以我们也可以直接访问各个分区，存储引擎管理分区的各个底层表和管理普通表一样（所有的底层表都必须使用相同的存储引擎），分区表的索引只是在各个底层表上各自加上一个相同的索引，从存储引擎的角度来看，底层表和一个普通表没有任何不同，存储引擎也无须知道这是一个普通表，还是一个分区表的一部分。

而 NoSQL 存储是基于键值对存储，虽然查询性能非常高，但在一些方面仍然存在短板。例如，不是关系型数据库，不支持事务以及稳定性方面相对 RDBMS 差一些。虽然有些 NoSQL 数据库也实现了事务，宣传具有可靠的稳定性，但目前 NoSQL 还是主要用作辅助存储。

什么时候要分表分库？

分析完了分区、NoSQL 存储优化的应用，接下来我们就看看这讲的重头戏——分表分库。

在我看来，能不分表分库就不要分表分库。在单表的情况下，当业务正常时，我们使用单表即可，而当业务出现了性能瓶颈时，我们首先考虑用分区的方式来优化，如果分区优化之后仍然存在后遗症，此时我们再来考虑分表分库。

我们知道，如果在单表单库的情况下，当数据库表的数据量逐渐累积到一定的数量时（5000W 行或 100G 以上），操作数据库的性能会出现明显下降，即使我们使用索引优化或读写库分离，性能依然存在瓶颈。此时，如果每日数据增长量非常大，我们就应该考虑分表，避免单表数据量过大，造成数据库操作性能下降。

面对海量数据，除了单表的性能比较差以外，我们在单表单库的情况下，数据库连接数、磁盘 I/O 以及网络吞吐等资源都是有限的，并发能力也是有限的。所以，在一些大数据量且高并发的业务场景中，我们就需要考虑分表分库来提升数据库的并发处理能力，从而提升应用的整体性能。

如何分表分库？

通常，分表分库分为垂直切分和水平切分两种。

垂直分库是指根据业务来分库，不同的业务使用不同的数据库。例如，订单和消费券在抢购业务中都存在着高并发，如果同时使用一个库，会占用一定的连接数，所以我们可以将数据库分为订单库和促销活动库。

而垂直分表则是指根据一张表中的字段，将一张表划分为两张表，其规则就是将一些不经常使用的字段拆分到另一张表中。例如，一张订单详情表有一百多个字段，显然这张表的字段太多了，一方面不方便我们开发维护，另一方面还可能引起跨页问题。这时我们就可以拆分该表字段，解决上述两个问题。

水平分表则是将表中的某一列作为切分的条件，按照某种规则（Range 或 Hash 取模）来切分为更小的表。

水平分表只是在一个库中，如果存在连接数、I/O 读写以及网络吞吐等瓶颈，我们就需要考虑将水平切换的表分布到不同机器的库中，这就是水平分库分表了。

结合以上垂直切分和水平切分，**我们一般可以将数据库分为：单库单表 - 单库多表 - 多库多表**。在平时的业务开发中，我们应该优先考虑单库单表；如果数据量比较大，且热点数据比较集中、历史数据很少访问，我们可以考虑表分区；如果访问热点数据分散，基本上所有的数据都会访问到，我们可以考虑单库多表；如果并发量比较高、海量数据以及每日新增数据量巨大，我们可以考虑多库多表。

这里还需要注意一点，我刚刚强调过，能不分表分库，就不要分表分库。这是因为一旦分表，我们可能会涉及到多表的分页查询、多表的 JOIN 查询，从而增加业务的复杂度。而一旦分库了，除了跨库分页查询、跨库 JOIN 查询，还会存在跨库事务的问题。这些问题无疑会增加我们系统开发的复杂度。

分表分库之后面临的问题

然而，分表分库虽然存在着各种各样的问题，但在一些海量数据、高并发的业务中，分表分库仍是最常用的优化手段。所以，我们应该充分考虑分表分库操作后所面临的一些问题，接下我们就一起看看都有哪些应对之策。

为了更容易理解这些问题，我们将对一个订单表进行分库分表，通过详细的业务来分析这些问题。

假设我们有一张订单表以及一张订单详情表，每天的数据增长量在 60W 单，平时还会有一些促销类活动，订单增长量在千万单。为了提高系统的并发能力，我们考虑将订单表和订单详情表做分库分表。除了分表，因为用户一般查询的是最近的订单信息，所以热点数据比较集中，我们还可以考虑用表分区来优化单表查询。

通常订单的分库分表要么基于订单号 Hash 取模实现，要么根据用户 ID Hash 取模实现。订单号 Hash 取模的好处是数据能均匀分布到各个表中，而缺陷则是一个用户查询所有订单时，需要去多个表中查询。

由于订单表用户查询比较多，此时我们应该考虑使用用户 ID 字段做 Hash 取模，对订单表进行水平分表。如果需要考虑高并发时的订单处理能力，我们可以考虑基于用户 ID 字段 Hash 取模实现分库分表。这也是大部分公司对订单表分库分表的处理方式。

1. 分布式事务问题

在提交订单时，除了创建订单之外，我们还需要扣除相应的库存。而订单表和库存表由于垂直分库，位于不同的库中，这时我们需要通过分布式事务来保证提交订单时的事务完整性。

通常，我们解决分布式事务有两种通用的方式：两阶事务提交（2PC）以及补偿事务提交（TCC）。有关分布式事务的内容，我将在第 41 讲中详细介绍。

通常有一些中间件已经帮我们封装好了这两种方式的实现，例如 Spring 实现的 JTA，目前阿里开源的分布式事务中间件 Fescar，就很好地实现了与 Dubbo 的兼容。

2. 跨节点 JOIN 查询问题

用户在查询订单时，我们往往需要通过表连接获取到商品信息，而商品信息表可能在另外一个库中，这就涉及到了跨库 JOIN 查询。

通常，我们会冗余表或冗余字段来优化跨库 JOIN 查询。对于一些基础表，例如商品信息表，我们可以在每一个订单分库中复制一张基础表，避免跨库 JOIN 查询。而对于一两个字段的查询，我们也可以将少量字段冗余在表中，从而避免 JOIN 查询，也就避免了跨库 JOIN 查询。

3. 跨节点分页查询问题

我们知道，当用户在订单列表中查询所有订单时，可以通过用户 ID 的 Hash 值来快速查询到订单信息，而运营人员在后台对订单表进行查询时，则是通过订单付款时间来进行查询的，这些数据都分布在不同的库以及表中，此时就存在一个跨节点分页查询的问题了。

通常一些中间件是通过在每个表中先查询出一定的数据，然后在缓存中排序后，获取到对应的分页数据。这种方式在越往后面的查询，就越消耗性能。

通常我们建议使用两套数据来解决跨节点分页查询问题，一套是基于分库分表的用户单条或多条查询数据，一套则是基于 Elasticsearch、Solr 存储的订单数据，主要用于运营人员根据其它字段进行分页查询。为了不影响提交订单的业务性能，我们一般使用异步消息来实现 Elasticsearch、Solr 订单数据的新增和修改。

4. 全局主键 ID 问题

在分库分表后，主键将无法使用自增长来实现了，在不同的表中我们需要统一全局主键 ID。因此，我们需要单独设计全局主键，避免不同表和库中的主键重复问题。

使用 UUID 实现全局 ID 是最方便快捷的方式，即随机生成一个 32 位 16 进制数字，这种方式可以保证一个 UUID 的唯一性，水平扩展能力以及性能都比较高。但使用 UUID 最大的缺陷就是，它是一个比较长的字符串，连续性差，如果作为主键使用，性能相对来说会比较差。

我们也可以基于 Redis 分布式锁实现一个递增的主键 ID，这种方式可以保证主键是一个整数且有一定的连续性，但分布式锁存在一定的性能消耗。

我们还可以基于 Twitter 开源的分布式 ID 生产算法——snowflake 解决全局主键 ID 问题，snowflake 是通过分别截取时间、机器标识、顺序计数的位数组成一个 long 类型的主键 ID。这种算法可以满足每秒上万个全局 ID 生成，不仅性能好，而且低延时。

5. 扩容问题

随着用户的订单量增加，根据用户 ID Hash 取模的分表中，数据量也在逐渐累积。此时，我们需要考虑动态增加表，一旦动态增加表了，就会涉及到数据迁移问题。

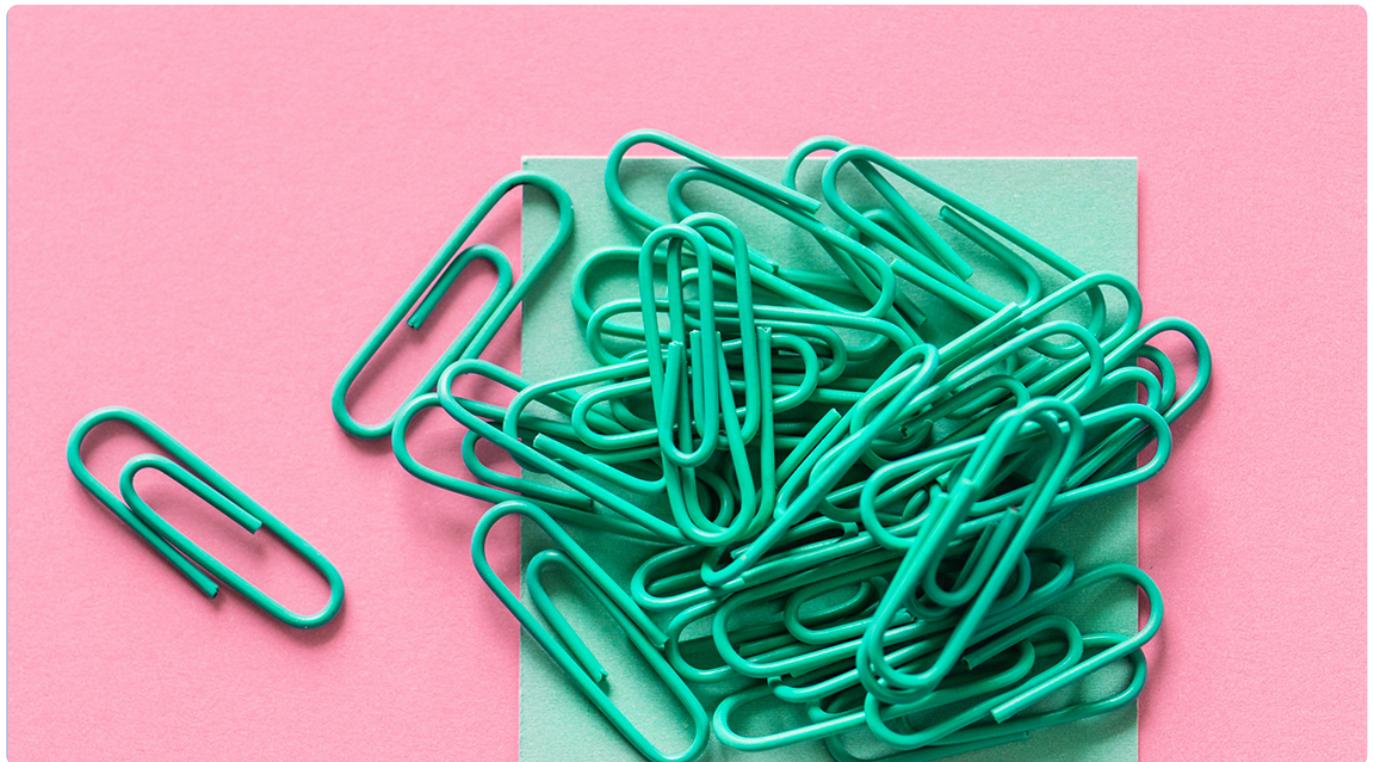
我们在最开始设计表数据量时，尽量使用 2 的倍数来设置表数量。当我们需要扩容时，也同样按照 2 的倍数来扩容，这种方式可以减少数据的迁移量。

总结

在业务开发之前，我们首先要根据自己的业务需求来设计表。考虑到一开始的业务发展比较平缓，且开发周期比较短，因此在开发时间比较紧的情况下，我们尽量不要考虑分表分库。但是我们可以将分表分库的业务接口预留，提前考虑后期分表分库的切分规则，把该冗余的字段提前冗余出来，避免后期分表分库的 JOIN 查询等。

当业务发展比较迅速的时候，我们就要评估分表分库的必要性了。一旦需要分表分库，就要结合业务提前规划切分规则，尽量避免消耗性能的跨表跨库 JOIN 查询、分页查询以及跨库事务等操作。

37 | 电商系统表设计优化案例分析



如果在业务架构设计初期，表结构没有设计好，那么后期随着业务以及数据量的增多，系统就很容易出现瓶颈。如果表结构扩展性差，业务耦合度将会越来越高，系统的复杂度也将随之增加。这一讲我将以电商系统中的表结构设计为例，为你详讲解在设计表时，我们都需要考虑哪些因素，又是如何通过表设计来优化系统性能。

核心业务

要懂得一个电商系统的表结构设计，我们必须先得熟悉一个电商系统中都有哪些基本核心业务。这部分的内容，只要你有过网购经历，就很好理解。

一般电商系统分为平台型和自营型电商系统。平台型电商系统是指有第三方商家入驻的电商平台，第三方商家自己开设店铺来维护商品信息、库存信息、促销活动、客服售后等，典型的代表有淘宝、天猫等。而自营型电商系统则是指没有第三方商家入驻，而是公司自己运营的电商平台，常见的有京东自营、苹果商城等。

两种类型的电商系统比较明显的区别是卖家是 C 端还是 B 端，很显然，平台型电商系统的复杂度要远远高于自营型电商系统。为了更容易理解商城的业务，我们将基于自营型电商系统来讨论表结构设计优化，这里以苹果商城为例。

一个电商系统的核心业务肯定就是销售商品了，围绕销售商品，我们可以将核心业务分为以下几个主要模块：

1. 商品模块

商品模块主要包括商品分类以及商品信息管理，商品分类则是我们常见的大分类了，有人喜欢将分类细化为多个层级，例如，第一个大类是手机、电视、配件等，配件的第二个大类又分为耳机、充电宝等。为了降低用户学习系统操作的成本，我们应该尽量将层级减少。

当我们通过了分类查询之后，就到了商品页面，一个商品 Item 包含了若干商品 SKU。商品 Item 是指一种商品，例如 iPhone9，就是一个 Item，商品 SKU 则是指具体属性的商品，例如金色 128G 内存的 iPhone9。

2. 购物车模块

购物车主要是用于用户临时存放欲购买的商品，并可以在购物车中统一下单结算。购物车一般分为离线购物车和在线购物车。离线购物车则是用户选择放入到购物车的商品只保存在本地缓存中，在线购物车则是会同步这些商品信息到服务端。

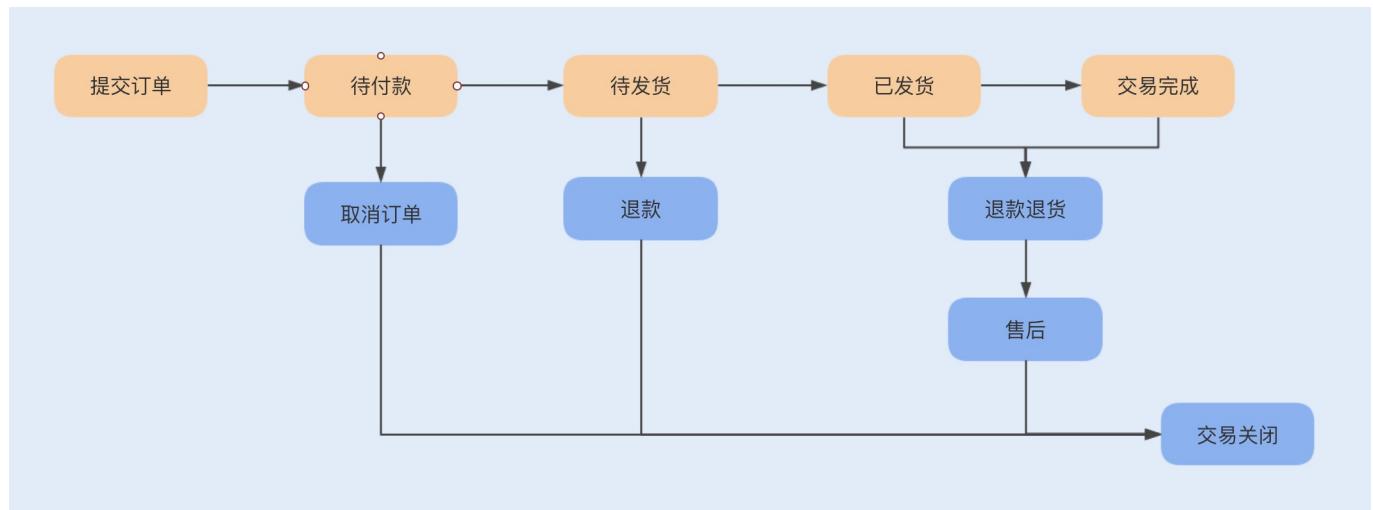
目前大部分商城都是支持两种状态的购物车，当用户没有登录商城时，主要是离线购物车在记录用户的商品信息，当用户登录商城之后，用户放入到购物车中的商品都会同步到服务端，以后在手机和电脑等不同平台以及不同时间都能查看到自己放入购物车的商品。

3. 订单模块

订单是盘活整个商城的核心功能模块，如果没有订单的产出，平台将难以维持下去。订单模块管理着用户在平台的交易记录，是用户和商家交流购买商品状态的渠道，用户可以随时更

改一个订单的状态，商家则必须按照业务流程及时订单的更新状态，告知用户已购买商品的具体状态。

通常一个订单分为以下几个状态：待付款、待发货、待收货、待评价、交易完成、用户取消、仅退款、退货退款状态。一个订单的流程见下图：



4. 库存模块

这里主要记录的是商品 SKU 的具体库存信息，主要功能包括库存交易、库存管理。库存交易是指用户购买商品时实时消费库存，库存管理主要包括运营人员对商品的生产或采购入库、调拨。

一般库存信息分为商品 SKU、仓位、实时库存、锁定库存、待退货库存、活动库存。

现在大部分电商都实现了华南华北的库存分区，所以可能存在同一个商品 SKU 在华北没有库存，而在华南存在库存的情况，所以我们需要有仓位这个字段，用来区分不同地区仓库的同一个商品 SKU。

实时库存则是指商品的实时库存，锁定库存则表示用户已经提交订单到实际扣除库存或订单失效的这段时间里锁定的库存，待退货库存、活动库存则分别表示订单退款时的库存数量以及每次活动时的库存数量。

除了这些库存信息，我们还可以为商品设置库存状态，例如虚拟库存状态、实物库存状态。如果一个商品不需要设定库存，可以任由用户购买，我们则不需要在每次用户购买商品时都去查询库存、扣除库存，只需要设定商品的库存状态为虚拟库存即可。

5. 促销活动模块

促销活动模块是指消费券、红包以及满减等促销功能，这里主要包括了活动管理和交易管理。前者主要负责管理每次发放的消费券及红包有效期、金额、满足条件、数量等信息，后者则主要负责管理用户领取红包、消费券等信息。

业务难点

了解了以上那些主要模块的具体业务之后，我们就可以更深入地去评估从业务落实到系统实现，可能存在的难点以及性能瓶颈了。

1. 不同商品类别存在差异，如何设计商品表结构？

我们知道，一个手机商品的详细信息跟一件衣服的详细信息差别很大，手机的 SKU 包括了颜色、运行内存、存储内存等，而一件衣服则包含了尺码、颜色。

如果我们要将这些商品都存放在一张表中，要么就使用相同字段来存储不同的信息，要么就新增字段来维护各自的信息。前者会导致程序设计复杂化、表宽度大，从而减少磁盘单页存储行数，影响查询性能，且维护成本高；后者则会导致一张表中字段过多，如果有新的商品类型出现，又需要动态添加字段。

比较好的方式是通过一个公共表字段来存储一些具有共性的字段，创建单独的商品类型表，例如手机商品一个表、服饰商品一个表。但这种方式也有缺点，那就是可能会导致表非常多，查询商品信息的时候不够灵活，不好实现全文搜索。

这时候，我们可以基于一个公共表来存储商品的公共信息，同时结合搜索引擎，将商品详细信息存储到键值对数据库，例如 ElasticSearch、Solr 中。

2. 双十一购物车商品数量大增，购物车系统出现性能瓶颈怎么办？

在用户没有登录系统的情况下，我们是通过 cookie 来保存购物车的商品信息，而在用户登录系统之后，购物车的信息会保存到数据库中。

在双十一期间，大部分用户都会提前将商品加入到购物车中，在加入商品到购物车的这段操作中，由于时间比较长，操作会比较分散，所以对数据库的写入并不会造成太大的压力。但在购买时，由于多数属于抢购商品，用户对购物车的访问则会比较集中了，如果都去数据库中读取，那么数据库的压力就可想而知了。

此时我们应该考虑冷热数据方案来存储购物车的商品信息，用户一般都会首选最近放入购物车的商品，这些商品信息则是热数据，而较久之前放入购物车中的商品信息则是冷数据，我们需要提前将热数据存放在 Redis 缓存中，以便提高系统在活动期间的并发性能。例如，可以将购物车中近一个月的商品信息都存放到 Redis 中，且至少为一个分页的信息。

当在缓存中没有查找到购物车信息时，再去数据库中查询，这样就可以大大降低数据库的压力。

3. 订单表海量数据，如何设计订单表结构？

通常我们的订单表是系统数据累计最快的一张表，无论订单是否真正付款，只要订单提交了就会在订单表中创建订单。如果公司的业务发展非常迅速，那么订单表的分表分库就只是迟早的事儿了。

在没有分表之前，订单的主键 ID 都是自增的，并且关联了一些其它业务表。一旦要进行分表分库，就会存在主键 ID 与业务耦合的情况，而且分表后新自增 ID 与之前的 ID 也可能会发生冲突，后期做表升级的时候我们将会面临巨大的工作量。如果我们确定后期做表升级，建议提前使用 snowflake 来生成主键 ID。

如果订单表要实现水平分表，那我们基于哪个字段来实现分表呢？

通常我们是通过计算用户 ID 字段的 Hash 值来实现订单的分表，这种方式可以优化用户购买端对订单的操作性能。如果我们需要对订单表进行水平分库，那就还是基于用户 ID 字段来实现。

在分表分库之后，对于我们的后台订单管理系统来说，查询订单就是一个挑战了。通常后台都是根据订单状态、创建订单时间进行查询的，且需要支持分页查询以及部分字段的 JOIN 查询，如果需要在分表分库的情况下进行这些操作，无疑是一个巨大的挑战了。

对于 JOIN 查询，我们一般可以通过冗余一些不常修改的配置表来实现。例如，商品的基础信息，我们录入之后很少修改，可以在每个分库中冗余该表，如果字段信息比较少，我们可以直接在订单表中冗余这些字段。

而对于分页查询，通常我们建议冗余订单信息到大数据中。后台管理系统通过大数据来查询订单信息，用户在提交订单并且付款之后，后台将会同步这条订单到大数据。用户在 C 端

修改或运营人员在后台修改订单时，会通过异步方式通知大数据更新该订单数据，这种方式可以解决分表分库后带来的分页查询问题。

4. 抢购业务，如何解决库存表的性能瓶颈？

在平时购买商品时，我们一般是直接去数据库检查、锁定库存，但如果是在促销活动期间抢购商品，我们还是直接去数据库检查、更新库存的话，面对高并发，系统无疑会产生性能瓶颈。

一般我们会将促销活动的库存更新到缓存中，通过缓存来查询商品的实时库存，并且通过分布式锁来实现库存扣减、锁定库存。分布式锁的具体实现，我会在第 41 讲中详讲。

5. 促销活动也存在抢购场景，如何设计表？

促销活动中的优惠券和红包交易，很多时候跟抢购活动有些类似。

在一些大型促销活动之前，我们一般都会定时发放各种商品的优惠券和红包，用户需要点击领取才能使用。所以在一定数量的优惠券和红包放出的同时，也会存在同一时间抢购这些优惠券和红包的情况，特别是一些热销商品。

我们可以参考库存的优化设计方式，使用缓存和分布式锁来查询、更新优惠券和红包的数量，通过缓存获取数量成功以后，再通过异步方式更新数据库中优惠券和红包的数量。

总结

这一讲，我们结合电商系统实战练习了如何进行表设计，可以总结为以下几个要点：

在字段比较复杂、易变动、不方便统一的情况下，建议使用键值对来代替关系数据库表存储；

在高并发情况下的查询操作，可以使用缓存代替数据库操作，提高并发性能；

数据量叠加比较快的表，需要考虑水平分表或分库，避免单表操作的性能瓶颈；

除此之外，我们应该通过一些优化，尽量避免比较复杂的 JOIN 查询操作，例如冗余一些字段，减少 JOIN 查询；创建一些中间表，减少 JOIN 查询。

38 | 数据库参数设置优化，失之毫厘差之千里



MySQL 是一个灵活性比较强的数据库系统，提供了很多可配置参数，便于我们根据应用和服务器硬件来做定制化数据库服务。如果现在让你回想，你可能觉得在开发的过程中很少去调整 MySQL 的配置参数，但我今天想说的是我们很有必要去深入了解它们。

我们知道，数据库主要是用来存取数据的，而存取数据涉及到了磁盘 I/O 的读写操作，所以数据库系统主要的性能瓶颈就是 I/O 读写的瓶颈了。MySQL 数据库为了减少磁盘 I/O 的读写操作，应用了大量内存管理来优化数据库操作，包括内存优化查询、排序以及写入操作。

也许你会想，我们把内存设置得越大越好，数据刷新到磁盘越快越好，不就对了吗？其实不然，内存设置过大，同样会带来新的问题。例如，InnoDB 中的数据和索引缓存，如果设置过大，就会引发 SWAP 页交换。还有数据写入到磁盘也不是越快越好，我们期望的是在高并发时，数据能均匀地写入到磁盘中，从而避免 I/O 性能瓶颈。

SWAP 页交换：SWAP 分区在系统的物理内存不够用的时候，就会把物理内存中的一部分空间释放出来，以供当前运行的程序使用。被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间的数据被临时保存到 SWAP 分区中，等到那些程序要运行时，再从 SWAP 分区中恢复保存的数据到内存中。

所以，这些参数的设置跟我们的应用服务特性以及服务器硬件有很大的关系。MySQL 是一个高定制化的数据库，我们可以根据需求来调整参数，定制性能最优的数据库。

不过想要了解这些参数的具体作用，我们先得了解数据库的结构以及不同存储引擎的工作原理。

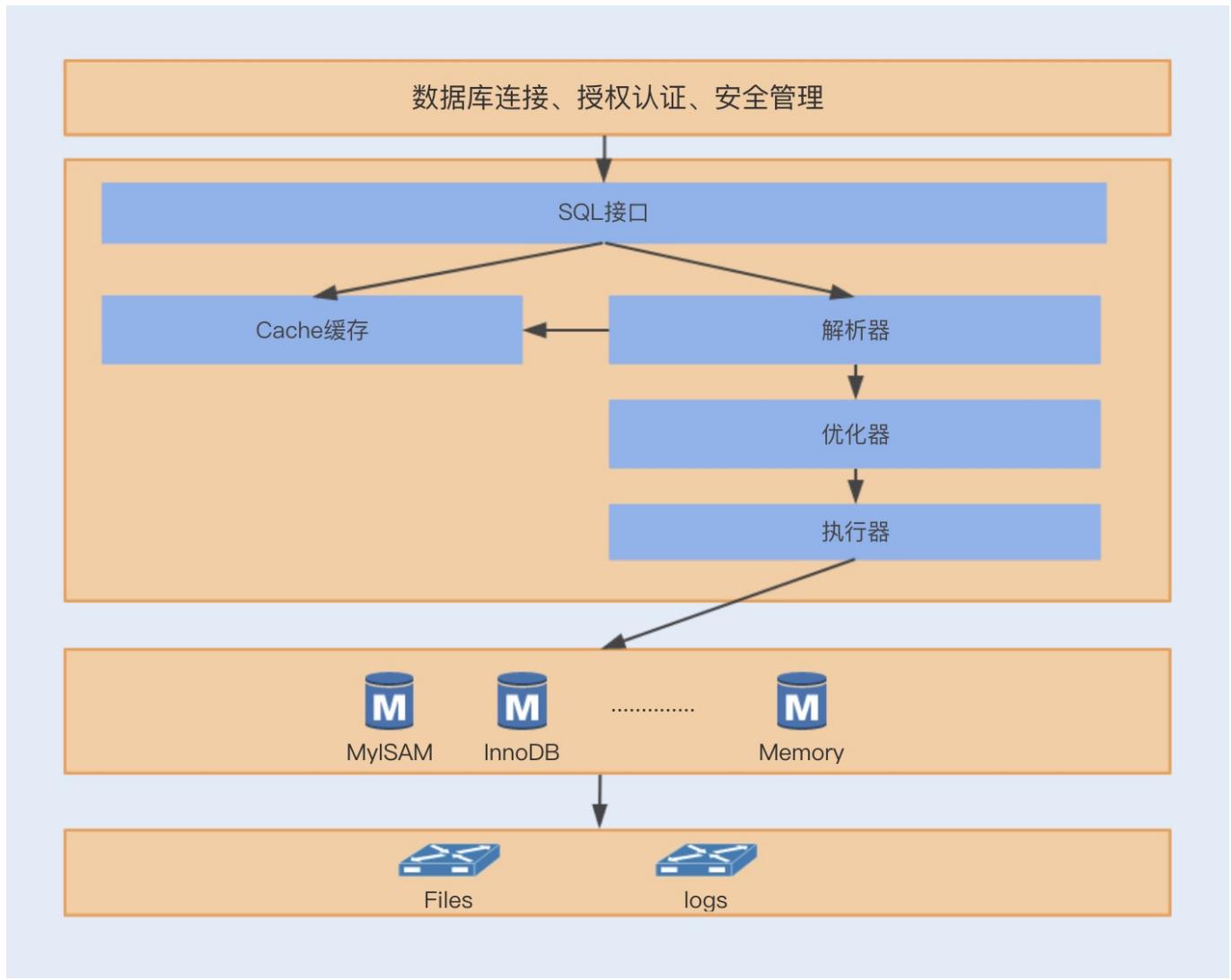
MySQL 体系结构

我们一般可以将 MySQL 的结构分为四层，最上层为客户端连接器，主要包括了数据库连接、授权认证、安全管理等，该层引用了线程池，为接入的连接请求提高线程处理效率。

第二层是 Server 层，主要实现 SQL 的一些基础功能，包括 SQL 解析、优化、执行以及缓存等，其中与我们这一讲主要相关的就是缓存。

第三层包括了各种存储引擎，主要负责数据的存取，这一层涉及到的 Buffer 缓存，也和这一讲密切相关。

最下面一层是数据存储层，主要负责将数据存储在文件系统中，并完成与存储引擎的交互。



接下来我们再来了解下，当数据接收到一个 SQL 语句时，是如何处理的。

1. 查询语句

一个应用服务需要通过第一层的连接和授权认证，再将 SQL 请求发送至 SQL 接口。SQL 接口接收到请求之后，会先检查查询 SQL 是否命中 Cache 缓存中的数据，如果命中，则直接返回缓存中的结果；否则，需要进入解析器。

解析器主要对 SQL 进行语法以及词法分析，之后，便会进入到优化器中，优化器会生成多种执行计划方案，并选择最优方案执行。

确定了最优执行计划方案之后，执行器会检查连接用户是否有该表的执行权限，有则查看 Buffer 中是否存在该缓存，存在则获取锁，查询表数据；否则重新打开表文件，通过接口调用相应的存储引擎处理，这时存储引擎就会进入到存储文件系统中获取相应的数据，并返回结果集。

2. 更新语句

数据库更新 SQL 的执行流程其实跟查询 SQL 差不多，只不过执行更新操作的时候多了记录日志的步骤。在执行更新操作时 MySQL 会将操作的日志记录到 binlog (归档日志) 中，这个步骤所有的存储引擎都有。而 InnoDB 除了要记录 binlog 之外，还需要多记录一个 redo log (重做日志)。

redo log 主要是为了解决 crash-safe 问题而引入的。我们知道，当数据库在存储数据时发生异常重启，我们需要保证存储的数据要么存储成功，要么存储失败，也就是不会出现数据丢失的情况，这就是 crash-safe 了。

我们在执行更新操作时，首先会查询相关的数据，之后通过执行器执行更新操作，并将执行结果写入到内存中，同时记录更新操作到 redo log 的缓存中，此时 redo log 中的记录状态为 prepare，并通知执行器更新完成，随时可以提交事务。执行器收到通知后会执行 binlog 的写入操作，此时的 binlog 是记录在缓存中的，写入成功后会调用引擎的提交事务接口，更新记录状态为 commit。之后，内存中的 redo log 以及 binlog 都会刷新到磁盘文件中。

内存调优

基于以上两个 SQL 执行过程，我们可以发现，在执行查询 SQL 语句时，会涉及到两个缓存。第一个缓存是刚进来时的 Query Cache，它缓存的是 SQL 语句和对应的结果集。这里的缓存是以查询 SQL 的 Hash 值为 key，返回结果集为 value 的键值对，判断一条 SQL 是否命中缓存，是通过匹配查询 SQL 的 Hash 值来实现的。

很明显，Query Cache 可以优化查询 SQL 语句，减少大量工作，特别是减少了 I/O 读取操作。我们可以通过以下几个主要的设置参数来优化查询操作：

参数	功能
have_query_cache	表示是否支持 Query Cache
query_cache_limit	表示 Query Cache 存放的单条 Query 最大结果集，默认值为 1M，结果集大小超过该值的 Query 不会被 Cache
query_cache_min_res_unit	表示 Query Cache 每个结果集存放的最小内存大小，默认为 4k
query_cache_size	表示系统中用于 Query Cache 的内存大小
query_cache_type	表示系统是否打开了 Query Cache 功能，可以设置为 ON、OFF、DEMAND（DEMAND 表示只有在查询语句中使用 SQL_CACHE 和 SQL_NO_CACHE 来控制是否需要缓存）

我们可以通过设置合适的 query_cache_min_res_unit 来减少碎片，这个参数最合适的大小和应用程序查询结果的平均大小直接相关，可以通过以下公式计算所得：

$$(\text{query_cache_size} - \text{Qcache_free_memory}) / \text{Qcache_queries_in_cache}$$

Qcache_free_memory 和 Qcache_queries_in_cache 的值可以通过以下命令查询：

 复制代码

```
1 show status like 'Qcache%'
```

Query Cache 虽然可以优化查询操作，但也仅限于不常修改的数据，如果一张表数据经常进行新增、更新和删除操作，则会造成 Query Cache 的失效率非常高，从而导致频繁地清除 Cache 中的数据，给系统增加额外的性能开销。

这也会导致缓存命中率非常低，我们可以通过以上查询状态的命令查看 Qcache_hits，该值表示缓存命中率。如果缓存命中率特别低的话，我们还可以通过 query_cache_size = 0 或者 query_cache_type 来关闭查询缓存。

经过了 Query Cache 缓存之后，还会使用到存储引擎中的 Buffer 缓存。不同的存储引擎，使用的 Buffer 也是不一样的。这里我们主要讲解两种常用的存储引擎。

1. MyISAM 存储引擎参数设置调优

MyISAM 存储引擎使用 key buffer 缓存索引块，MyISAM 表的数据块则没有缓存，它是直接存储在磁盘文件中的。

我们可以通过 key_buffer_size 设置 key buffer 缓存的大小，而它的大小并不是越大越好。正如我前面所讲的，key buffer 缓存设置过大，实际应用却不太大的话，就容易造成内存浪费，而且系统也容易发生 SWAP 页交换，一般我是建议将服务器内存中可用内存的 1/4 分配给 key buffer。

如果要更准确地评估 key buffer 的设置是否合理，我们还可以通过缓存使用率公式来计算：

$$1 - ((\text{key_blocks_unused} * \text{key_cache_block_size}) / \text{key_buffer_size})$$

key_blocks_unused 表示未使用的缓存簇（blocks）数

key_cache_block_size 表示 key_buffer_size 被分割的区域大小

key_blocks_unused * key_cache_block_size 则表示剩余的可用缓存空间
(一般来说，缓存使用率在 80% 作用比较合适)。

2. InnoDB 存储引擎参数设置调优

InnoDB Buffer Pool (简称 IBP) 是 InnoDB 存储引擎的一个缓冲池，与 MyISAM 存储引擎使用 key buffer 缓存不同，它不仅存储了表索引块，还存储了表数据。查询数据时，IBP 允许快速返回频繁访问的数据，而无需访问磁盘文件。InnoDB 表空间缓存越多，MySQL 访问物理磁盘的频率就越低，这表示查询响应时间更快，系统的整体性能也有所提高。

我们一般可以通过多个设置参数来调整 IBP，优化 InnoDB 表性能。

innodb_buffer_pool_size

IBP 默认的内存大小是 128M，我们可以通过参数 innodb_buffer_pool_size 来设置 IBP 的大小，IBP 设置得越大，InnoDB 表性能就越好。但是，将 IBP 大小设置得过大也不好，可能会导致系统发生 SWAP 页交换。所以我们需要在 IBP 大小和其它系统服务所需内存大小之间取得平衡。MySQL 推荐配置 IBP 的大小为服务器物理内存的 80%。

我们也可以通过计算 InnoDB 缓冲池的命中率来调整 IBP 大小：

$(1 - \text{innodb_buffer_pool_reads} / \text{innodb_buffer_pool_read_request}) * 100$

但如果我们将 IBP 的大小设置为物理内存的 80% 以后，发现命中率还是很低，此时我们就应该考虑扩充内存来增加 IBP 的大小。

innodb_buffer_pool_instances

InnoDB 中的 IBP 缓冲池被划分为了多个实例，对于具有数千兆字节的缓冲池的系统来说，将缓冲池划分为单独的实例可以减少不同线程读取和写入缓存页面时的争用，从而提高系统的并发性。该参数项仅在将 innodb_buffer_pool_size 设置为 1GB 或更大时才会生效。

在 windows 32 位操作系统中，如果 innodb_buffer_pool_size 的大小超过 1.3GB，innodb_buffer_pool_instances 默认大小就为 innodb_buffer_pool_size/128MB；否则，默认为 1。

而在其它操作系统中，如果 innodb_buffer_pool_size 大小超过 1GB，innodb_buffer_pool_instances 值就默认为 8；否则，默认为 1。

为了获取最佳效率，建议指定 innodb_buffer_pool_instances 的大小，并保证每个缓冲池实例至少有 1GB 内存。通常，建议 innodb_buffer_pool_instances 的大小不超过 innodb_read_io_threads + innodb_write_io_threads 之和，建议实例和线程数量比例为 1:1。

innodb_read_io_threads / innodb_write_io_threads

在默认情况下，MySQL 后台线程包括了主线程、IO 线程、锁线程以及监控线程等，其中读写线程属于 IO 线程，主要负责数据库的读取和写入操作，这些线程分别读取和写入 innodb_buffer_pool_instances 创建的各个内存页面。MySQL 支持配置多个读写线程，即通过 innodb_read_io_threads 和 innodb_write_io_threads 设置读写线程数量。

读写线程数量值默认为 4，也就是总共有 8 个线程同时在后台运行。

innodb_read_io_threads 和 innodb_write_io_threads 设置的读写线程数量，与 innodb_buffer_pool_instances 的大小有关，两者的协同优化是提高系统性能的一个关键因素。

在一些内存以及 CPU 内核超大型的数据库服务器上，我们可以在保证足够大的 IBP 内存的前提下，通过以下公式，协同增加缓存实例数量以及读写线程。

```
( innodb_read_io_threads + innodb_write_io_threads ) =  
innodb_buffer_pool_instances
```

如果我们仅仅是将读写线程根据缓存实例数量对半来分，即读线程和写线程各为实例大小的一半，肯定是不合理的。例如我们的应用服务读取数据库的数据多于写入数据库的数据，那么增加写入线程反而没有优化效果。我们一般可以通过 MySQL 服务器保存的全局统计信息，来确定系统的读取和写入比率。

我们可以通过以下查询来确定读写比率：

 复制代码

```
1 SHOW GLOBAL STATUS LIKE 'Com_select';// 读取数量  
2  
3 SHOW GLOBAL STATUS WHERE Variable_name IN ('Com_insert', 'Com_update', 'Com_replace', '(
```



如果读大于写，我们应该考虑将读线程的数量设置得大一些，写线程数量小一些；否则，反之。

innodb_log_file_size

除了以上 InnoDB 缓存等因素之外，InnoDB 的日志缓存大小、日志文件大小以及日志文件持久化到磁盘的策略都影响着 InnoDB 的性能。InnoDB 中有一个 redo log 文件，InnoDB 用它来存储服务器处理的每个写请求的重做活动。执行的每个写入查询都会在日志文件中获得重做条目，以便在发生崩溃时可以恢复更改。

当日志文件大小已经超过我们参数设置的日志文件大小时，InnoDB 会自动切换到另外一个日志文件，由于重做日志是一个循环使用的环，在切换时，就需要将新的日志文件脏页的缓存数据刷新到磁盘中（触发检查点）。

理论上来说，innodb_log_file_size 设置得越大，缓冲池中需要的检查点刷新活动就越少，从而节省磁盘 I/O。那是不是将这个日志文件设置得越大越好呢？如果日志文件设置得太

大，恢复时间就会变长，这样不便于 DBA 管理。在大多数情况下，我们将日志文件大小设置为 1GB 就足够了。

innodb_log_buffer_size

这个参数决定了 InnoDB 重做日志缓冲池的大小，默认值为 8MB。如果高并发中存在大量的事务，该值设置得太小，就会增加写入磁盘的 I/O 操作。我们可以通过增大该参数来减少写入磁盘操作，从而提高并发时的事务性能。

innodb_flush_log_at_trx_commit

这个参数可以控制重做日志从缓存写入文件刷新到磁盘中的策略，默认值为 1。

当设置该参数为 0 时，InnoDB 每秒钟就会触发一次缓存日志写入到文件中并刷新到磁盘的操作，这有可能在数据库崩溃后，丢失 1s 的数据。

当设置该参数为 1 时，则表示每次事务的 redo log 都会直接持久化到磁盘中，这样可以保证 MySQL 异常重启之后数据不会丢失。

当设置该参数为 2 时，每次事务的 redo log 都会直接写入到文件中，再将文件刷新到磁盘。

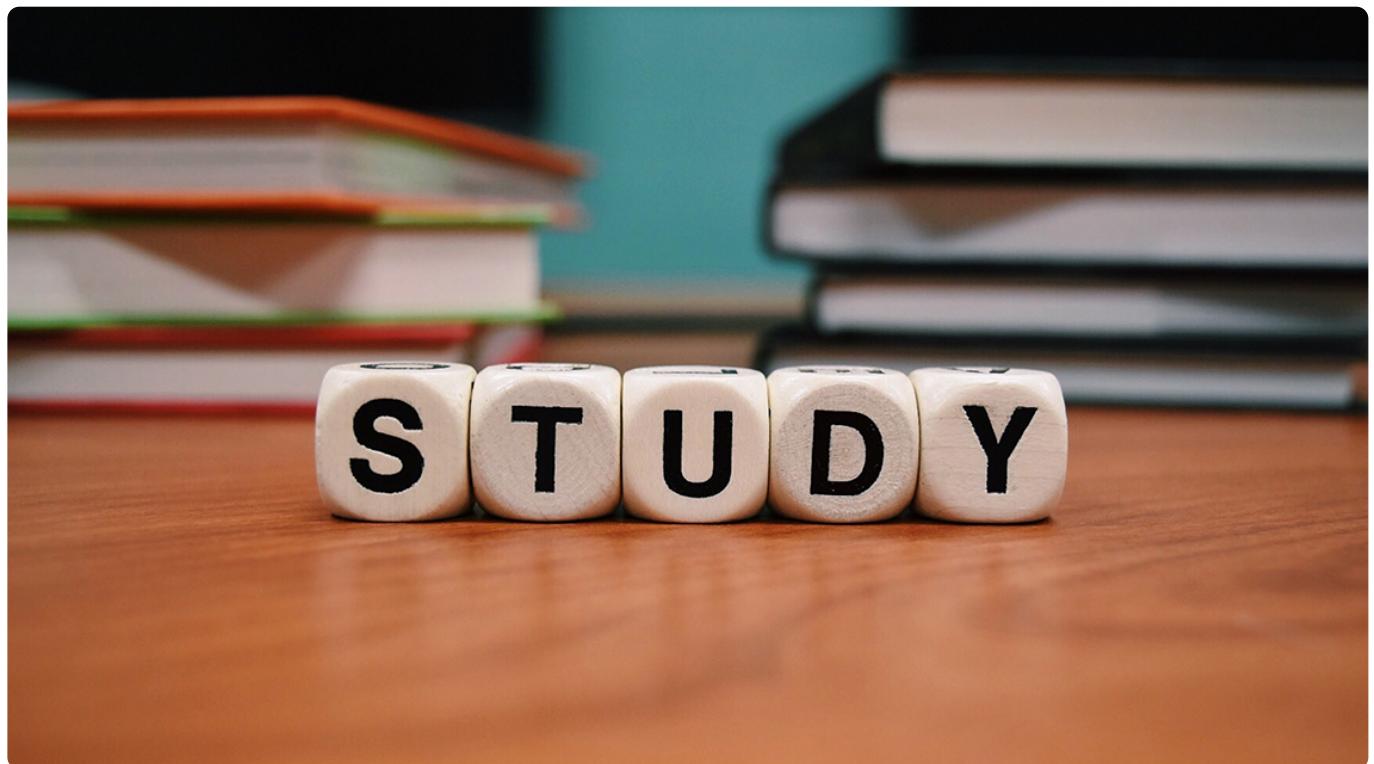
在一些对数据安全性要求比较高的场景中，显然该值需要设置为 1；而在一些可以容忍数据库崩溃时丢失 1s 数据的场景中，我们可以将该值设置为 0 或 2，这样可以明显地减少日志同步到磁盘的 I/O 操作。

总结

MySQL 数据库的参数设置非常多，今天我们仅仅是了解了与内存优化相关的参数设置。除了这些参数设置，我们还有一些常用的提高 MySQL 并发的相关参数设置，总结如下：

参数	调优
max_connections	控制允许连接到 MySQL 数据库的最大连接数量， 默认为 151。我们查看状态变量 connection_errors_max_connections 的值大于零或遇到 MySQL: ERROR 1040: Too manyconnections 时，应该考虑增加连接数。
back_log	TCP 连接请求排队等待栈， 并发量比较大的情况下， 可以适当调大该参数， 增加短时间内处理连接请求数量。
thread_cache_size	MySQL 接收到客户端的连接时， 需要生成线程用于处理连接。当连接断开时， 线程并不会立刻销毁， 而是对线程进行缓存， 便于下一个连接使用， 减少线程的创建和销毁。我们可以查看状态变量 Threads_created 是否过大， 如果该状态变量值过大， 说明 MySQL 一直在创建处理连接的线程， 我们就可以适当调大 thread_cache_size。

39 | 答疑课堂：MySQL中InnoDB的知识点串讲

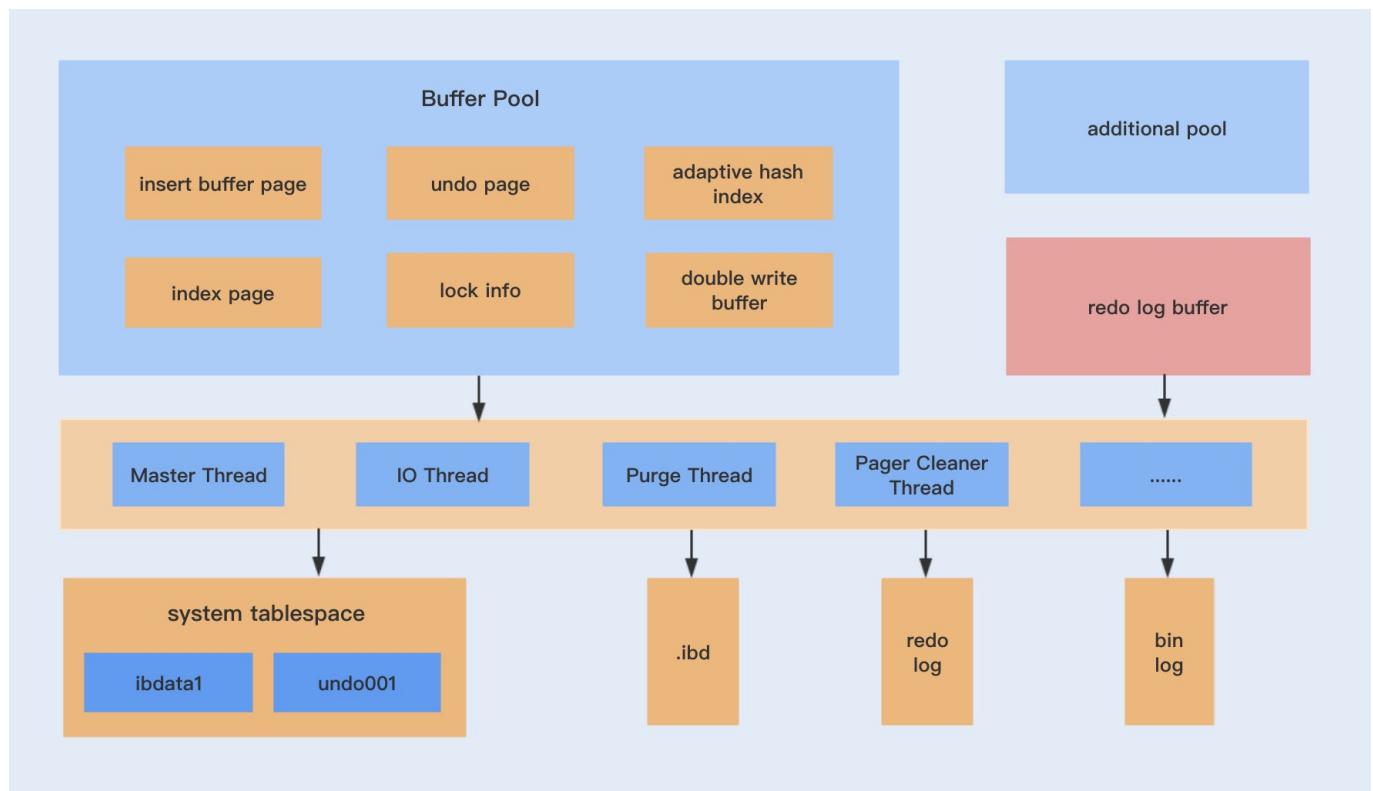


模块六有关数据库调优的内容到本周也正式结束了，今天我们一起串下 MySQL 中 InnoDB 的知识点。InnoDB 存储引擎作为我们最常用到的存储引擎之一，充分熟悉它的的实现和运行原理，有助于我们更好地创建和维护数据库表。

InnoDB 体系架构

InnoDB 主要包括了内存池、后台线程以及存储文件。内存池又是由多个内存块组成的，主要包括缓存磁盘数据、redo log 缓冲等；后台线程则包括了 Master Thread、IO Thread 以及 Purge Thread 等；由 InnoDB 存储引擎实现的表的存储结构文件一般包括表结构文

件 (.frm)、共享表空间文件 (ibdata1)、独占表空间文件 (ibd) 以及日志文件 (redo 文件等) 等。



1. 内存池

我们知道，如果客户端从数据库中读取数据是直接从磁盘读取的话，无疑会带来一定的性能瓶颈，缓冲池的作用就是提高整个数据库的读写性能。

客户端读取数据时，如果数据存在于缓冲池中，客户端就会直接读取缓冲池中的数据，否则再去磁盘中读取；对于数据库中的修改数据，首先是修改在缓冲池中的数据，然后再通过 Master Thread 线程刷新到磁盘上。

理论上来说，缓冲池的内存越大越好。我们在[第 38 讲](#)中详细讲过了缓冲池的大小配置方式以及调优。

缓冲池中不仅缓存索引页和数据页，还包括了 undo 页，插入缓存、自适应哈希索引以及 InnoDB 的锁信息等等。

InnoDB 允许多个缓冲池实例，从而减少数据库内部资源的竞争，增强数据库的并发处理能力，[第 38 讲](#)还讲到了缓冲池实例的配置以及调优。

InnoDB 存储引擎会先将重做日志信息放入到缓冲区中，然后再刷新到重做日志文件中。

2. 后台线程

Master Thread 主要负责将缓冲池中的数据异步刷新到磁盘中，除此之外还包括插入缓存、undo 页的回收等，IO Thread 是负责读写 IO 的线程，而 Purge Thread 主要用于回收事务已经提交了的 undo log，Pager Cleaner Thread 是新引入的一个用于协助 Master Thread 刷新脏页到磁盘的线程，它可以减轻 Master Thread 的工作压力，减少阻塞。

3. 存储文件

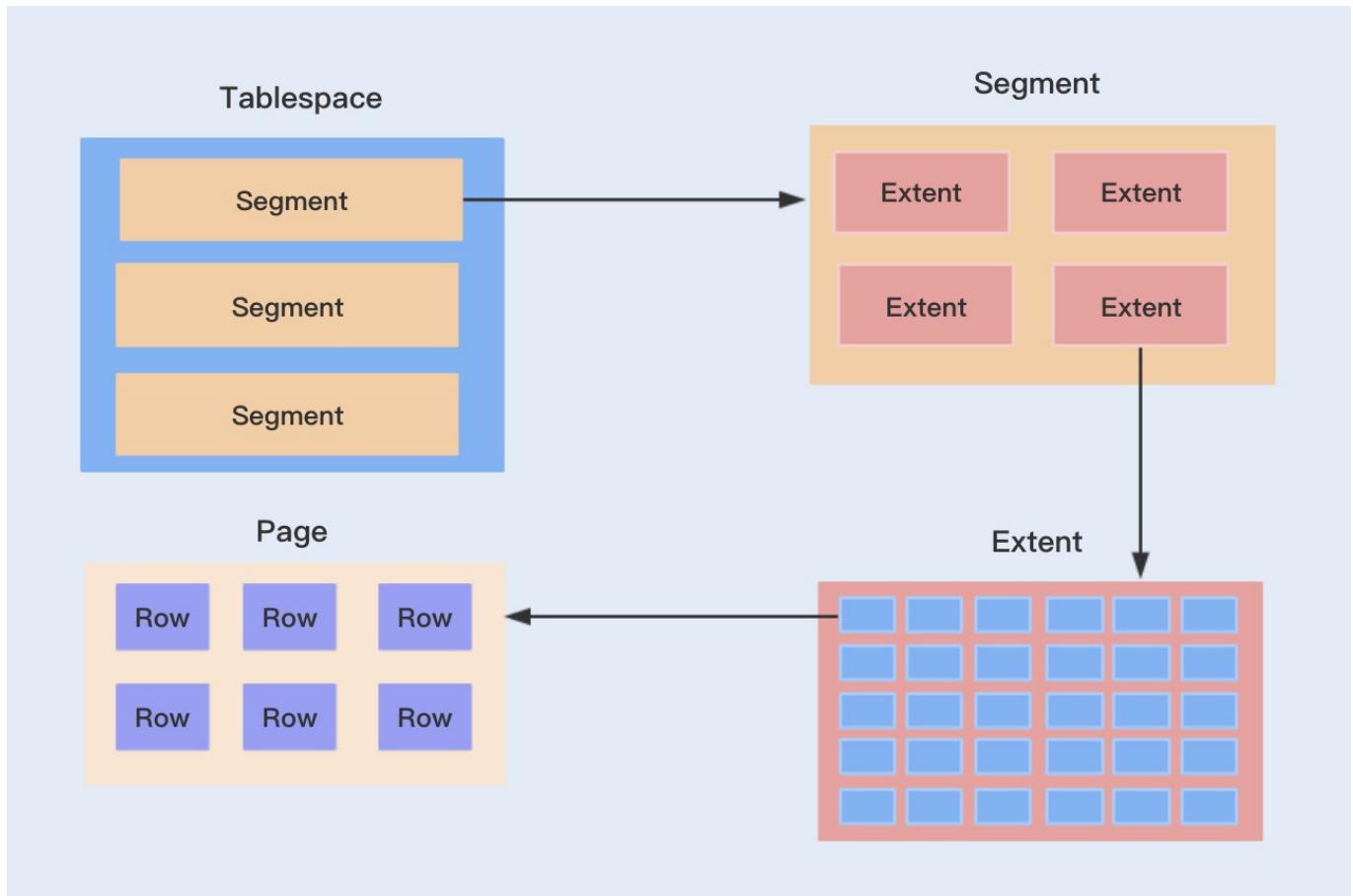
在 MySQL 中建立一张表都会生成一个.frm 文件，该文件是用来保存每个表的元数据信息的，主要包含表结构定义。

在 InnoDB 中，存储数据都是按表空间进行存放的，默认为共享表空间，存储的文件即为共享表空间文件（ibdata1）。若设置了参数 innodb_file_per_table 为 1，则会将存储的数据、索引等信息单独存储在一个独占表空间，因此也会产生一个独占表空间文件（ibd）。如果你对共享表空间和独占表空间的理解还不够透彻，接下来我会详解。

而日志文件则主要是重做日志文件，主要记录事务产生的重做日志，保证事务的一致性。

InnoDB 逻辑存储结构

InnoDB 逻辑存储结构分为表空间（Tablespace）、段（Segment）、区（Extent）、页（Page）以及行（row）。



1. 表空间 (Tablespace)

InnoDB 提供了两种表空间存储数据的方式，一种是共享表空间，一种是独占表空间。InnoDB 默认会将其所有的表数据存储在一个共享表空间中，即 ibdata1。

我们可以通过设置 `innodb_file_per_table` 参数为 1 (1 代表独占方式) 开启独占表空间模式。开启之后，每个表都有自己独立的表空间物理文件，所有的数据以及索引都会存储在该文件中，这样方便备份以及恢复数据。

2. 段 (Segment)

表空间是由各个段组成的，段一般分为数据段、索引段和回滚段等。我们知道，InnoDB 默认是基于 B + 树实现的数据存储。

这里的索引段则是指的 B + 树的非叶子节点，而数据段则是 B + 树的叶子节点。而回滚段则指的是回滚数据，之前我们在讲事务隔离的时候就介绍到了 MVCC 利用了回滚段实现了多版本查询数据。

3. 区 (Extent) / 页 (Page)

区是表空间的单元结构，每个区的大小为 1MB。而页是组成区的最小单元，页也是 InnoDB 存储引擎磁盘管理的最小单元，每个页的大小默认为 16KB。为了保证页的连续性，InnoDB 存储引擎每次从磁盘申请 4-5 个区。

4. 行 (Row)

InnoDB 存储引擎是面向列的 (row-oriented)，也就是说数据是按行进行存放的，每个页存放的行记录也是有硬性定义的，最多允许存放 16KB/2-200 行，即 7992 行记录。

InnoDB 事务之 redo log 工作原理

InnoDB 是一个事务性的存储引擎，而 InnoDB 的事务实现是基于事务日志 redo log 和 undo log 实现的。redo log 是重做日志，提供再写入操作，实现事务的持久性；undo log 是回滚日志，提供回滚操作，保证事务的一致性。

redo log 又包括了内存中的日志缓冲 (redo log buffer) 以及保存在磁盘的重做日志文件 (redo log file)，前者存储在内存中，容易丢失，后者持久化在磁盘中，不会丢失。

InnoDB 的更新操作采用的是 Write Ahead Log 策略，即先写日志，再写入磁盘。当一条记录更新时，InnoDB 会先把记录写入到 redo log buffer 中，并更新内存数据。我们可以通过参数 `innodb_flush_log_at_trx_commit` 自定义 commit 时，如何将 redo log buffer 中的日志刷新到 redo log file 中。

在这里，我们需要注意的是 InnoDB 的 redo log 的大小是固定的，分别有多个日志文件采用循环方式组成一个循环闭环，当写到结尾时，会回到开头循环写日志。我们可以通过参数 `innodb_log_files_in_group` 和 `innodb_log_file_size` 配置日志文件数量和每个日志文件的大小。

Buffer Pool 中更新的数据未刷新到磁盘中，该内存页我们称之为脏页。最终脏页的数据会刷新到磁盘中，将磁盘中的数据覆盖，这个过程与 redo log 不一定有关系。

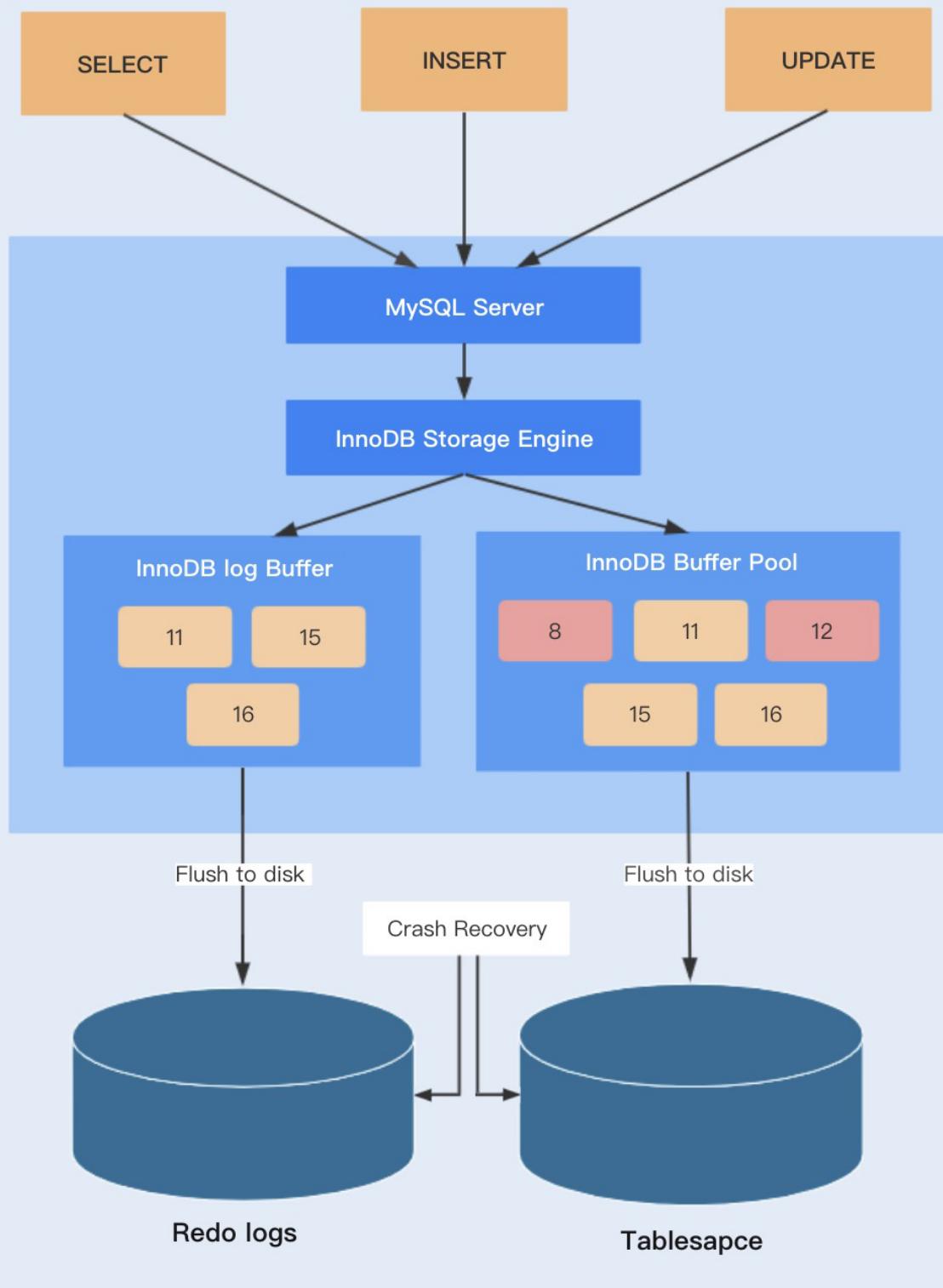
只有当 redo log 日志满了的情况下，才会主动触发脏页刷新到磁盘，而脏页不仅只有 redo log 日志满了的情况才会刷新到磁盘，以下几种情况同样会触发脏页的刷新：

系统内存不足时，需要将一部分数据页淘汰掉，如果淘汰的是脏页，需要先将脏页同步到磁盘；

MySQL 认为空闲的时间，这种情况没有性能问题；

MySQL 正常关闭之前，会把所有的脏页刷入到磁盘，这种情况也没有性能问题。

在生产环境中，如果我们开启了慢 SQL 监控，你会发现偶尔会出现一些用时稍长的 SQL。这是因为脏页在刷新到磁盘时可能会给数据库带来性能开销，导致数据库操作抖动。



LRU 淘汰策略

以上我们了解了 InnoDB 的更新和插入操作的具体实现原理，接下来我们再来了解下读的实现和优化方式。

InnoDB 存储引擎是基于集合索引实现的数据存储，也就是除了索引列以及主键是存储在 B + 树之外，其它列数据也存储在 B + 树的叶子节点中。而这里的索引页和数据页都会缓存在缓冲池中，在查询数据时，只要在缓冲池中存在该数据，InnoDB 就不用每次都去磁盘中读取页，从而提高数据库的查询性能。

虽然缓冲池是一个很大的内存区域，但由于存放了各种类型的数据，加上存储数据量之大，缓冲池无法将所有的数据都存储在其中。因此，缓冲池需要通过 LRU 算法将最近且经常查询的数据缓存在其中，而不常查询的数据就淘汰出去。

InnoDB 对 LRU 做了一些优化，我们熟悉的 LRU 算法通常是将最近查询的数据放到 LRU 列表的首部，而 InnoDB 则是将数据放在一个 midpoint 位置，通常这个 midpoint 为列表长度的 5/8。

这种策略主要是为了避免一些不常查询的操作突然将热点数据淘汰出去，而热点数据被再次查询时，需要再次从磁盘中获取，从而影响数据库的查询性能。

如果我们的热点数据比较多，我们可以通过调整 midpoint 值来增加热点数据的存储量，从而降低热点数据的淘汰率。

总结

以上 InnoDB 的实现和运行原理到这里就介绍完了。回顾模块六，前三讲我主要介绍了数据库操作的性能优化，包括 SQL 语句、事务以及索引的优化，接下来我又讲到了数据库表优化，包括表设计、分表分库的实现等等，最后我还介绍了一些数据库参数的调优。

总的来讲，作为开发工程师，我们应该掌握数据库几个大的知识点，然后再深入到数据库内部实现的细节，这样才能避免经常写出一些具有性能问题的 SQL，培养调优数据库性能的能力。

41 | 如何设计更优的分布式锁？



从这一讲开始，我们就正式进入最后一个模块的学习了，综合性实战的内容来自我亲身经历过的一些案例，其中用到的知识点会相对综合，现在是时候跟我一起调动下前面所学了！

去年双十一，我们的游戏商城也搞了一波活动，那时候我就发现在数据库操作日志中，出现最多的一个异常就是 `InterruptedException` 了，几乎所有的异常都是来自一个校验订单幂等性的 SQL。

因为校验订单幂等性是提交订单业务中第一个操作数据库的，所以幂等性校验也就承受了比较大的请求量，再加上我们还是基于一个数据库表来实现幂等性校验的，所以出现了一些请

求事务超时，事务被中断的情况。其实基于数据库实现的幂等性校验就是一种分布式锁的实现。

那什么是分布式锁呢，它又是用来解决哪些问题的呢？

在 JVM 中，在多线程并发的情况下，我们可以使用同步锁或 Lock 锁，保证在同一时间内，只能有一个线程修改共享变量或执行代码块。但现在我们的服务基本都是基于分布式集群来实现部署的，对于一些共享资源，例如我们之前讨论过的库存，在分布式环境下使用 Java 锁的方式就失去作用了。

这时，我们就需要实现分布式锁来保证共享资源的原子性。除此之外，分布式锁也经常用来避免分布式中的不同节点执行重复性的工作，例如一个定时发短信的任务，在分布式集群中，我们只需要保证一个服务节点发送短信即可，一定要避免多个节点重复发送短信给同一个用户。

因为数据库实现一个分布式锁比较简单易懂，直接基于数据库实现就行了，不需要再引入第三方中间件，所以这是很多分布式业务实现分布式锁的首选。但是数据库实现的分布式锁在一定程度上，存在性能瓶颈。

接下来我们一起了解下如何使用数据库实现分布式锁，其性能瓶颈到底在哪，有没有其它实现方式可以优化分布式锁。

数据库实现分布式锁

首先，我们应该创建一个锁表，通过创建和查询数据来保证一个数据的原子性：

 复制代码

```
1 CREATE TABLE `order` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `order_no` int(11) DEFAULT NULL,
4   `pay_money` decimal(10, 2) DEFAULT NULL,
5   `status` int(4) DEFAULT NULL,
6   `create_date` datetime(0) DEFAULT NULL,
7   `delete_flag` int(4) DEFAULT NULL,
8   PRIMARY KEY (`id`) USING BTREE,
9   INDEX `idx_status`(`status`) USING BTREE,
10  INDEX `idx_order`(`order_no`) USING BTREE
11 ) ENGINE = InnoDB
```

其次，如果是校验订单的幂等性，就要先查询该记录是否存在数据库中，查询的时候要防止幻读，如果不存在，就插入到数据库，否则，放弃操作。

 复制代码

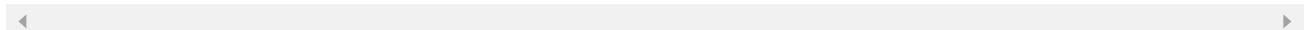
```
1 select id from `order` where `order_no` = 'xxxx' for update
```



最后注意下，除了查询时防止幻读，我们还需要保证查询和插入是在同一个事务中，因此我们需要申明事务，具体的实现代码如下：

 复制代码

```
1     @Transactional
2     public int addOrderRecord(Order order) {
3         if(orderDao.selectOrderRecord(order)==null){
4             int result = orderDao.addOrderRecord(order);
5             if(result>0){
6                 return 1;
7             }
8         }
9         return 0;
10    }
```



到这，我们订单幂等性校验的分布式锁就实现了。我想你应该能发现为什么这种方式会存在性能瓶颈了。我们在[第 34 讲](#)中讲过，在 RR 事务级别，select 的 for update 操作是基于间隙锁 gap lock 实现的，这是一种悲观锁的实现方式，所以存在阻塞问题。

因此在高并发情况下，当有大量的请求进来时，大部分的请求都会进行排队等待。为了保证数据库的稳定性，事务的超时时间往往又设置得很小，所以就会出现大量事务被中断的情况。

除了阻塞等待之外，因为订单没有删除操作，所以这张锁表的数据将会逐渐累积，我们需要设置另外一个线程，隔一段时间就去删除该表中的过期订单，这就增加了业务的复杂度。

除了这种幂等性校验的分布式锁，有一些单纯基于数据库实现的分布式锁代码块或对象，是需要在锁释放时，删除或修改数据的。如果在获取锁之后，锁一直没有获得释放，即数据没

有被删除或修改，这将会引发死锁问题。

Zookeeper 实现分布式锁

除了数据库实现分布式锁的方式以外，我们还可以基于 Zookeeper 实现。Zookeeper 是一种提供“分布式服务协调”的中心化服务，正是 Zookeeper 的以下两个特性，分布式应用程序才可以基于它实现分布式锁功能。

顺序临时节点：Zookeeper 提供一个多层级的节点命名空间（节点称为 Znode），每个节点都用一个以斜杠（/）分隔的路径来表示，而且每个节点都有父节点（根节点除外），非常类似于文件系统。

节点类型可以分为持久节点（PERSISTENT）、临时节点（EPHEMERAL），每个节点还能被标记为有序性（SEQUENTIAL），一旦节点被标记为有序性，那么整个节点就具有顺序自增的特点。一般我们可以组合这几类节点来创建我们所需要的节点，例如，创建一个持久节点作为父节点，在父节点下面创建临时节点，并标记该临时节点为有序性。

Watch 机制：Zookeeper 还提供了另外一个重要特性，Watcher（事件监听器）。ZooKeeper 允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper 服务端会将事件通知给用户。

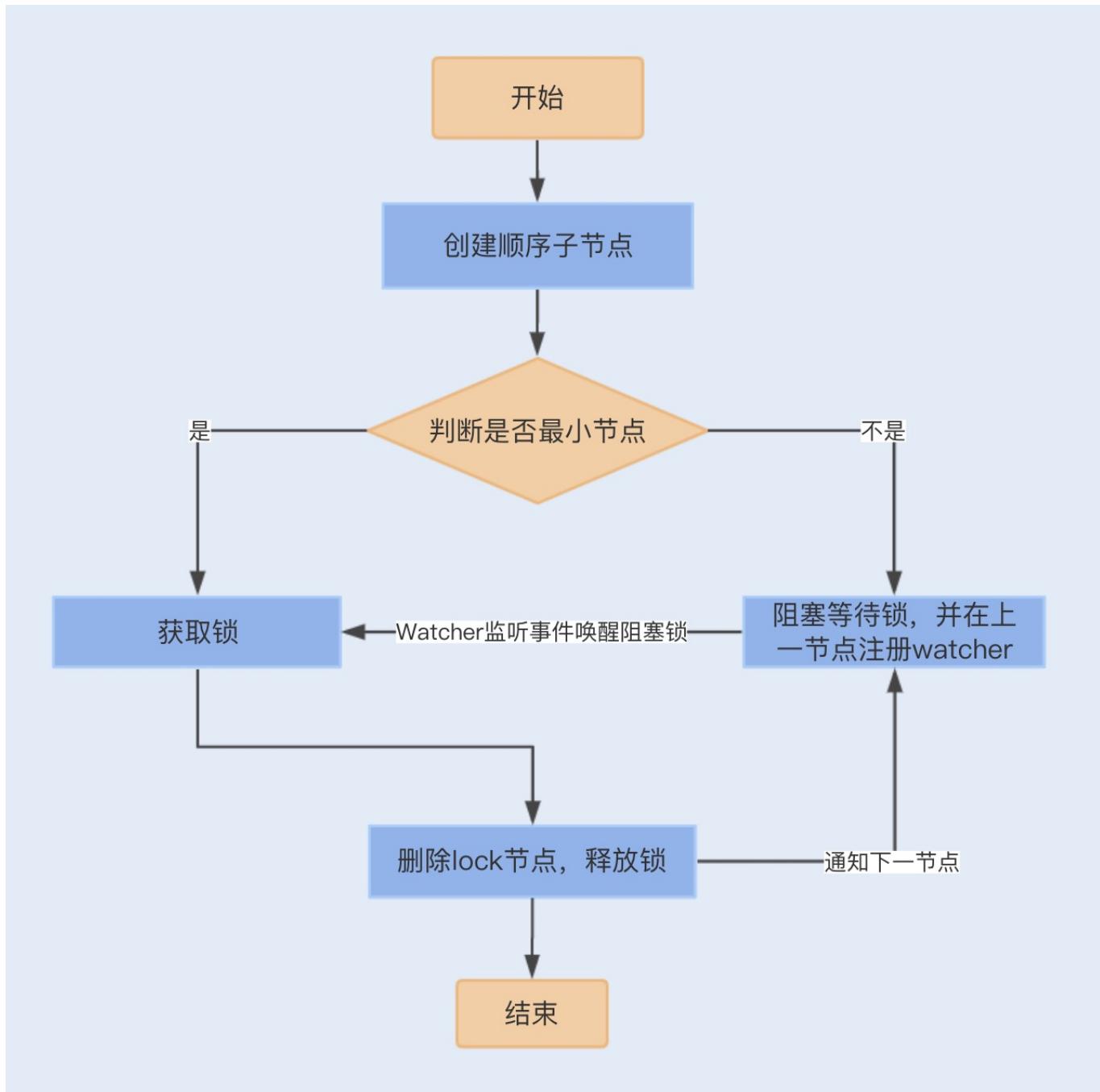
我们熟悉了 Zookeeper 的这两个特性之后，就可以看看 Zookeeper 是如何实现分布式锁的了。

首先，我们需要建立一个父节点，节点类型为持久节点（PERSISTENT），每当需要访问共享资源时，就会在父节点下建立相应的顺序子节点，节点类型为临时节点（EPHEMERAL），且标记为有序性（SEQUENTIAL），并且以临时节点名称 + 父节点名称 + 顺序号组成特定的名字。

在建立子节点后，对父节点下面的所有以临时节点名称 name 开头的子节点进行排序，判断刚刚建立的子节点顺序号是否是最小的节点，如果是最小节点，则获得锁。

如果不是最小节点，则阻塞等待锁，并且获得该节点的上一顺序节点，为其注册监听事件，等待节点对应的操作获得锁。

当调用完共享资源后，删除该节点，关闭 zk，进而可以触发监听事件，释放该锁。



以上实现的分布式锁是严格按照顺序访问的并发锁。一般我们还可以直接引用 Curator 框架来实现 Zookeeper 分布式锁，代码如下：

复制代码

```

1 InterProcessMutex lock = new InterProcessMutex(client, lockPath);
2 if ( lock.acquire(maxWait, waitUnit) )
3 {
4     try
5     {
6         // do some work inside of the critical section here
7     }
8     finally
9     {
10        lock.release();

```

```
11     }
12 }
```

Zookeeper 实现的分布式锁，例如相对数据库实现，有很多优点。Zookeeper 是集群实现，可以避免单点问题，且能保证每次操作都可以有效地释放锁，这是因为一旦应用服务挂掉了，临时节点会因为 session 连接断开而自动删除掉。

由于频繁地创建和删除结点，加上大量的 Watch 事件，对 Zookeeper 集群来说，压力非常大。且从性能上来说，其与接下来我要讲的 Redis 实现的分布式锁相比，还是存在一定的差距。

Redis 实现分布式锁

相对于前两种实现方式，基于 Redis 实现的分布式锁是最为复杂的，但性能是最佳的。

大部分开发人员利用 Redis 实现分布式锁的方式，都是使用 SETNX+EXPIRE 组合来实现，在 Redis 2.6.12 版本之前，具体实现代码如下：

 复制代码

```
1 public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId)
2
3     Long result = jedis.setnx(lockKey, requestId); // 设置锁
4     if (result == 1) { // 获取锁成功
5         // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁
6         jedis.expire(lockKey, expireTime); // 通过过期时间删除锁
7         return true;
8     }
9     return false;
10 }
```

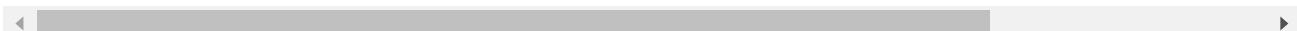
这种方式实现的分布式锁，是通过 setnx() 方法设置锁，如果 lockKey 存在，则返回失败，否则返回成功。设置成功之后，为了能在完成同步代码之后成功释放锁，方法中还需要使用 expire() 方法给 lockKey 值设置一个过期时间，确认 key 值删除，避免出现锁无法释放，导致下一个线程无法获取到锁，即死锁问题。

如果程序在设置过期时间之前、设置锁之后出现崩溃，此时如果 lockKey 没有设置过期时间，将会出现死锁问题。

在 Redis 2.6.12 版本后 SETNX 增加了过期时间参数：

 复制代码

```
1  private static final String LOCK_SUCCESS = "OK";
2  private static final String SET_IF_NOT_EXIST = "NX";
3  private static final String SET_WITH_EXPIRE_TIME = "PX";
4
5  /**
6   * 尝试获取分布式锁
7   * @param jedis Redis 客户端
8   * @param lockKey 锁
9   * @param requestId 请求标识
10  * @param expireTime 超期时间
11  * @return 是否获取成功
12  */
13 public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String req
14
15     String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_
16
17     if (LOCK_SUCCESS.equals(result)) {
18         return true;
19     }
20     return false;
21
22 }
```



我们也可以通过 Lua 脚本来实现锁的设置和过期时间的原子性，再通过 jedis.eval() 方法运行该脚本：

 复制代码

```
1 // 加锁脚本
2 private static final String SCRIPT_LOCK = "if redis.call('setnx', KEYS[1], ARGV[1])
3 // 解锁脚本
4 private static final String SCRIPT_UNLOCK = "if redis.call('get', KEYS[1]) == ARGV[1]
```



虽然 SETNX 方法保证了设置锁和过期时间的原子性，但如果我们设置的过期时间比较短，而执行业务时间比较长，就会存在锁代码块失效的问题。我们需要将过期时间设置得足够

长，来保证以上问题不会出现。

这个方案是目前最优的分布式锁方案，但如果是在 Redis 集群环境下，依然存在问题。由于 Redis 集群数据同步到各个节点时是异步的，如果在 Master 节点获取到锁后，在没有同步到其它节点时，Master 节点崩溃了，此时新的 Master 节点依然可以获取锁，所以多个应用服务可以同时获取到锁。

Redlock 算法

Redisson 由 Redis 官方推出，它是一个在 Redis 的基础上实现的 Java 驻内存数据网格 (In-Memory Data Grid)。它不仅提供了一系列的分布式的 Java 常用对象，还提供了许多分布式服务。Redisson 是基于 netty 通信框架实现的，所以支持非阻塞通信，性能相对于我们熟悉的 Jedis 会好一些。

Redisson 中实现了 Redis 分布式锁，且支持单点模式和集群模式。在集群模式下，Redisson 使用了 Redlock 算法，避免在 Master 节点崩溃切换到另外一个 Master 时，多个应用同时获得锁。我们可以通过一个应用服务获取分布式锁的流程，了解下 Redlock 算法的实现：

在不同的节点上使用单个实例获取锁的方式去获得锁，且每次获取锁都有超时时间，如果请求超时，则认为该节点不可用。当应用服务成功获取锁的 Redis 节点超过半数 ($N/2+1$, N 为节点数) 时，并且获取锁消耗的实际时间不超过锁的过期时间，则获取锁成功。

一旦获取锁成功，就会重新计算释放锁的时间，该时间是由原来释放锁的时间减去获取锁所消耗的时间；而如果获取锁失败，客户端依然会释放获取锁成功的节点。

具体的代码实现如下：

1. 首先引入 jar 包：

 复制代码

```
1 <dependency>
2   <groupId>org.redisson</groupId>
3   <artifactId>redisson</artifactId>
4   <version>3.8.2</version>
5 </dependency>
```

1. 实现 Redisson 的配置文件:

 复制代码

```
1 @Bean
2 public RedissonClient redissonClient() {
3     Config config = new Config();
4     config.useClusterServers()
5         .setScanInterval(2000) // 集群状态扫描间隔时间, 单位是毫秒
6         .addNodeAddress("redis://127.0.0.1:7000").setPassword("1")
7         .addNodeAddress("redis://127.0.0.1:7001").setPassword("1")
8         .addNodeAddress("redis://127.0.0.1:7002")
9         .setPassword("1");
10    return Redisson.create(config);
11 }
```

◀ ▶

1. 获取锁操作:

 复制代码

```
1 long waitTimeout = 10;
2 long leaseTime = 1;
3 RLock lock1 = redissonClient1.getLock("lock1");
4 RLock lock2 = redissonClient2.getLock("lock2");
5 RLock lock3 = redissonClient3.getLock("lock3");
6
7 RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3);
8 // 同时加锁: lock1 lock2 lock3
9 // 红锁在大部分节点上加锁成功就算成功, 且设置总超时时间以及单个节点超时时间
10 redLock.tryLock(waitTimeout,leaseTime,TimeUnit.SECONDS);
11 ...
12 redLock.unlock();
```

◀ ▶

总结

实现分布式锁的方式有很多，有最简单的数据库实现，还有 Zookeeper 多节点实现和缓存实现。我们可以分别对这三种实现方式进行性能压测，可以发现在同样的服务器配置下，Redis 的性能是最好的，Zookeeper 次之，数据库最差。

从实现方式和可靠性来说，Zookeeper 的实现方式简单，且基于分布式集群，可以避免单点问题，具有比较高的可靠性。因此，在对业务性能要求不是特别高的场景中，我建议使用 Zookeeper 实现的分布式锁。

42 | 电商系统的分布式事务调优



今天的分享也是从案例开始。我们团队曾经遇到过一个非常严重的线上事故，在一次 DBA 完成单台数据库线上补丁后，系统偶尔会出现异常报警，我们的开发工程师很快就定位到了数据库异常问题。

具体情况是这样的，当玩家购买道具之后，扣除通宝时出现了异常。这种异常在正常情况下发生之后，应该是整个购买操作都需要撤销，然而这次异常的严重性就是在于玩家购买道具成功后，没有扣除通宝。

究其原因是由于购买的道具更新的是游戏数据库，而通宝是在用户账户中心数据库，在一次购买道具时，存在同时操作两个数据库的情况，属于一种分布式事务。而我们的工程师在完

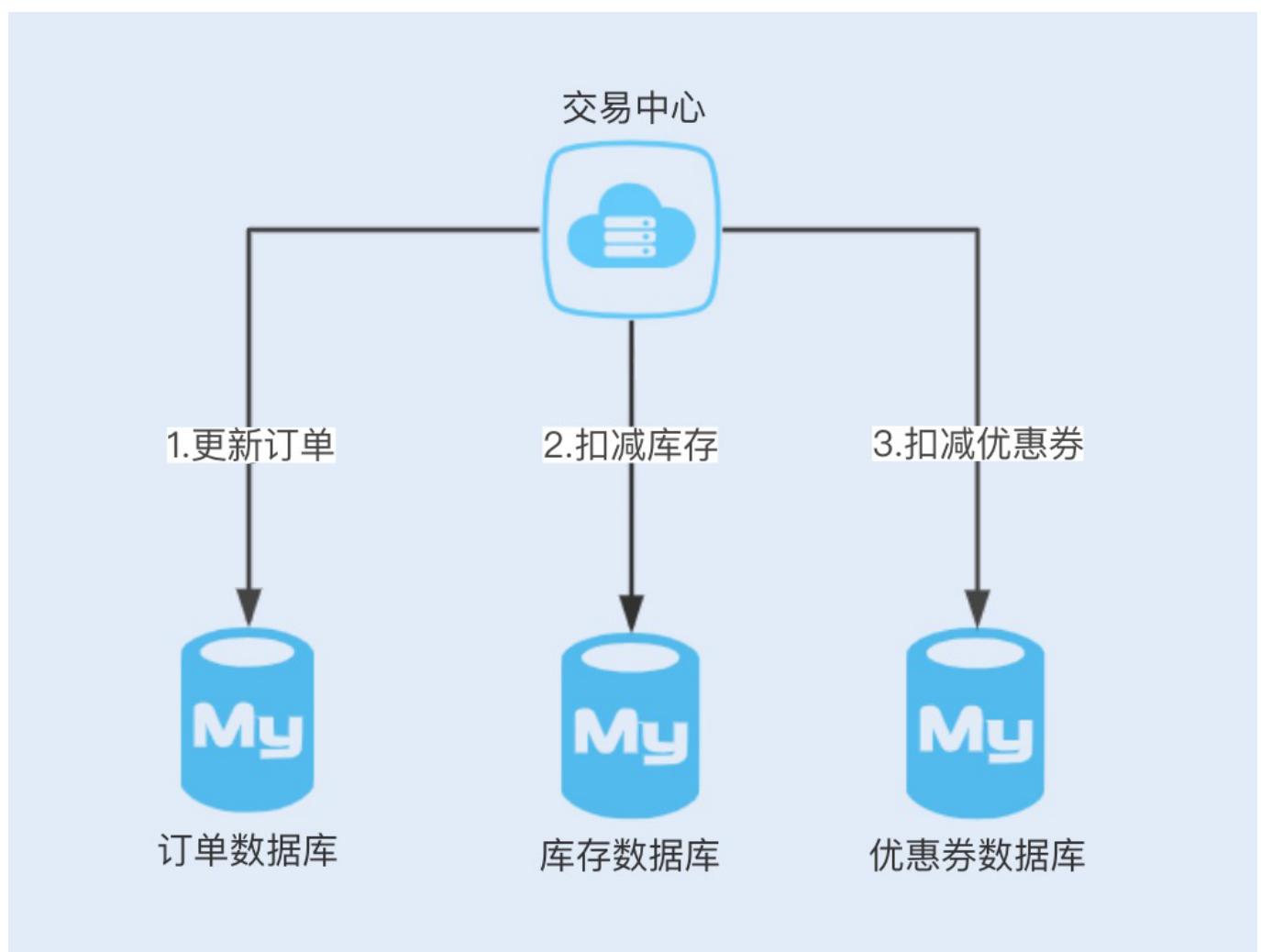
成玩家获得道具和扣除余额的操作时，没有做到事务的一致性，即在扣除通宝失败时，应该回滚已经购买的游戏道具。

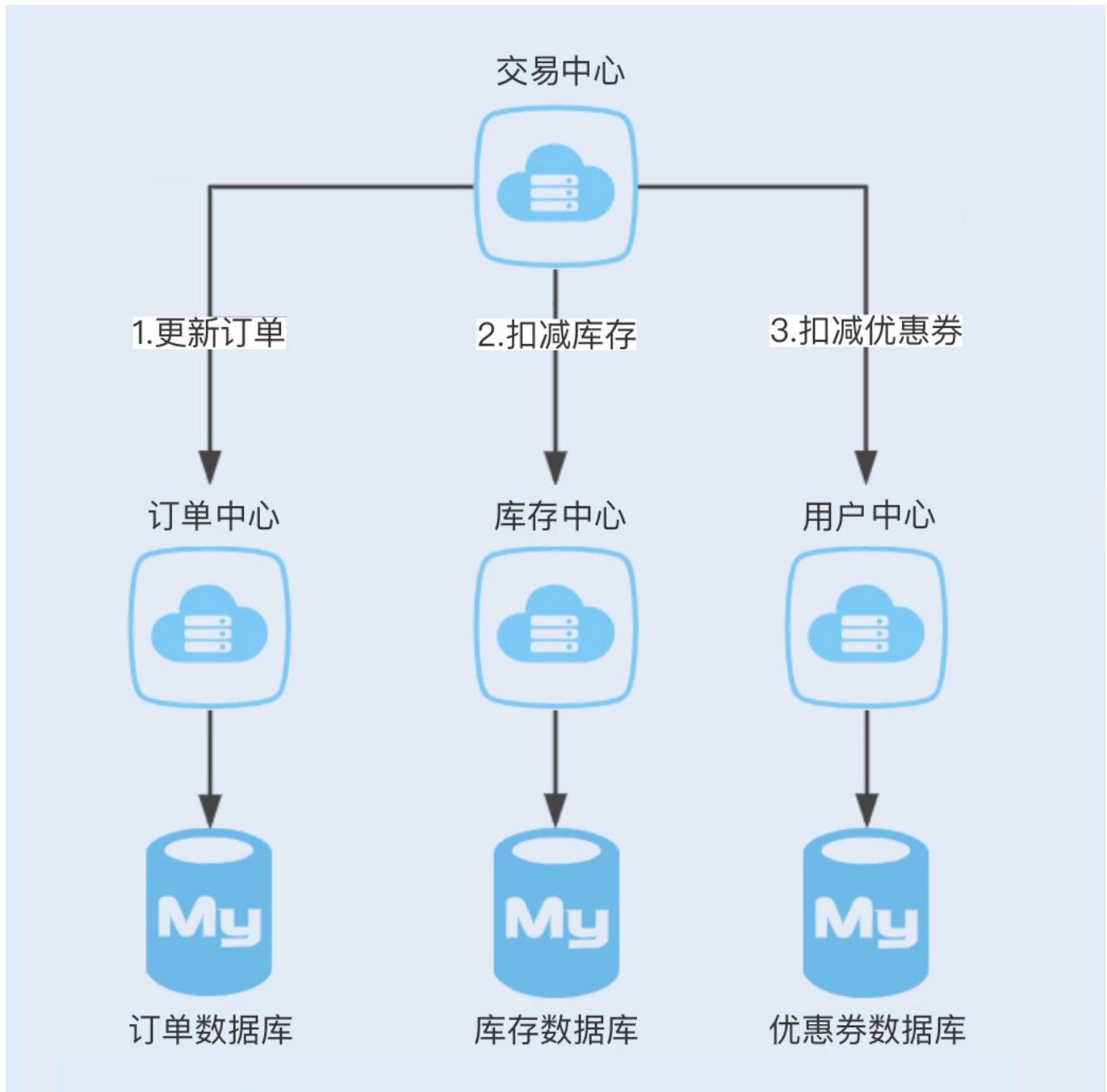
从这个案例中，我想你应该意识到了分布式事务的重要性。

如今，大部分公司的服务基本都实现了微服务化，首先是业务需求，为了解耦业务；其次是为了减少业务与业务之间的相互影响。

电商系统亦是如此，大部分公司的电商系统都是分为了不同服务模块，例如商品模块、订单模块、库存模块等等。事实上，分解服务是一把双刃剑，可以带来一些开发、性能以及运维上的优势，但同时也会增加业务开发的逻辑复杂度。其中最为突出的就是分布式事务了。

通常，存在分布式事务的服务架构部署有以下两种：同服务不同数据库，不同服务不同数据库。我们以商城为例，用图示说明下这两种部署：





通常，我们都是基于第二种架构部署实现的，那我们应该如何实现在这种服务架构下，有关订单提交业务的分布式事务呢？

分布式事务解决方案

我们讲过，在单个数据库的情况下，数据事务操作具有 ACID 四个特性，但如果在一个事务中操作多个数据库，则无法使用数据库事务来保证一致性。

也就是说，当两个数据库操作数据时，可能存在一个数据库操作成功，而另一个数据库操作失败的情况，我们无法通过单个数据库事务来回滚两个数据操作。

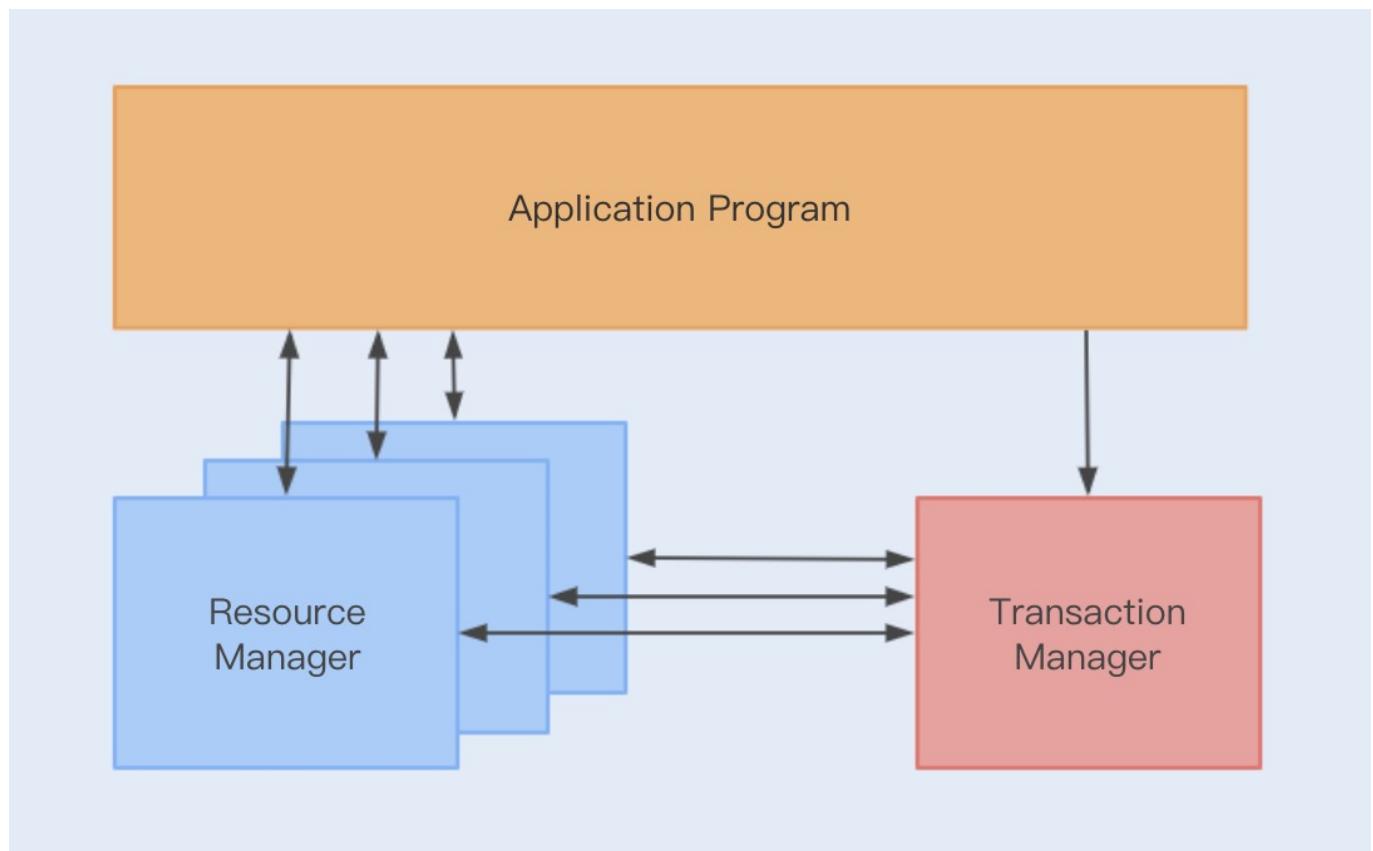
而分布式事务就是为了解决在同一个事务下，不同节点的数据库操作数据不一致的问题。在一个事务操作请求多个服务或多个数据库节点时，要么所有请求成功，要么所有请求都失败回滚回去。通常，分布式事务的实现有多种方式，例如 XA 协议实现的二阶提交（2PC）、三阶提交（3PC），以及 TCC 补偿性事务。

在了解 2PC 和 3PC 之前，我们有必要先来了解下 XA 协议。XA 协议是由 X/Open 组织提出的一个分布式事务处理规范，目前 MySQL 中只有 InnoDB 存储引擎支持 XA 协议。

1. XA 规范

在 XA 规范之前，存在着一个 DTP 模型，该模型规范了分布式事务的模型设计。

DTP 规范中主要包含了 AP、RM、TM 三个部分，其中 AP 是应用程序，是事务发起和结束的地方；RM 是资源管理器，主要负责管理每个数据库的连接数据源；TM 是事务管理器，负责事务的全局管理，包括事务的生命周期管理和资源的分配协调等。



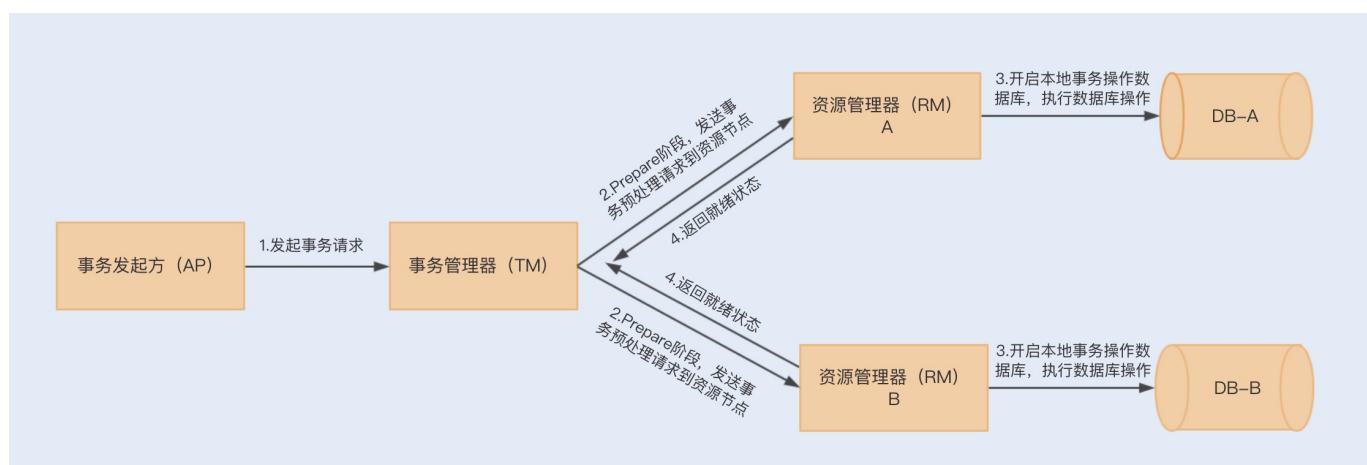
XA 则规范了 TM 与 RM 之间的通信接口，在 TM 与多个 RM 之间形成一个双向通信桥梁，从而在多个数据库资源下保证 ACID 四个特性。

这里强调一下，JTA 是基于 XA 规范实现的一套 Java 事务编程接口，是一种两阶段提交事务。我们可以通过[源码](#)简单了解下 JTA 实现的多数据源事务提交。

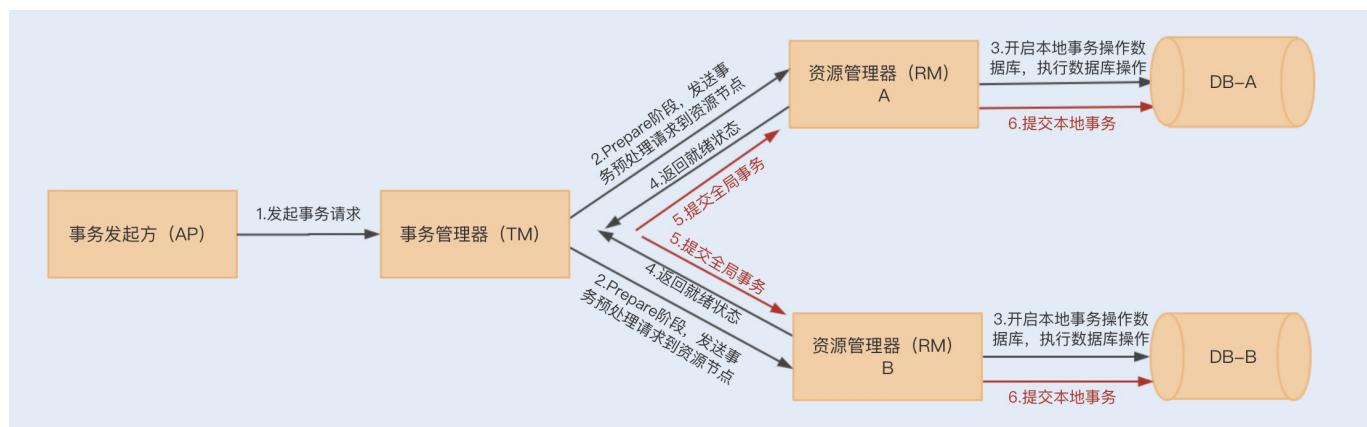
2. 二阶提交和三阶提交

XA 规范实现的分布式事务属于二阶提交事务，顾名思义就是通过两个阶段来实现事务的提交。

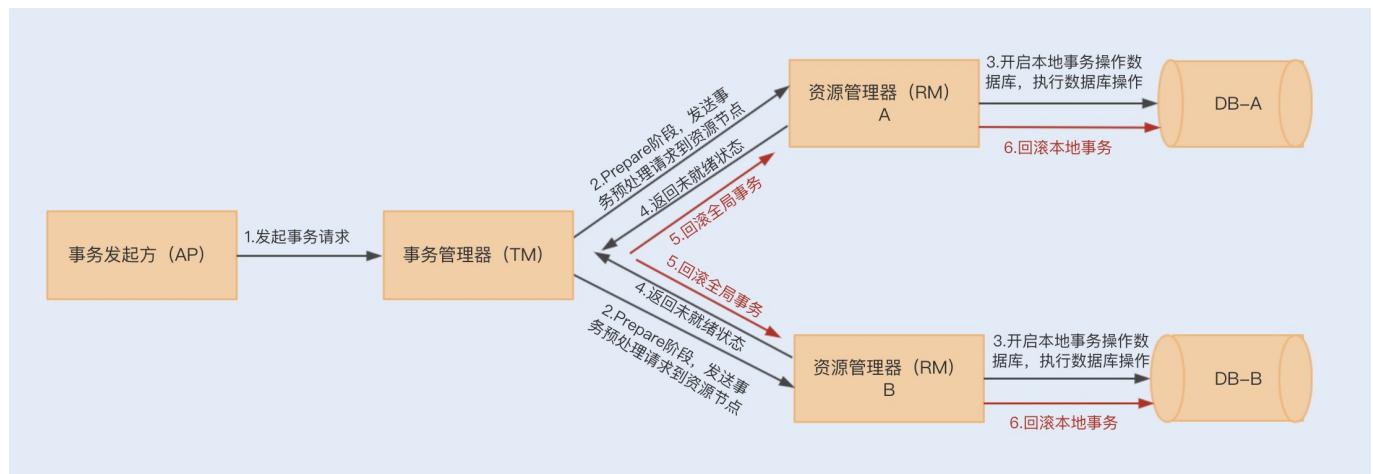
在第一阶段，应用程序向事务管理器（TM）发起事务请求，而事务管理器则会分别向参与的各个资源管理器（RM）发送事务预处理请求（Prepare），此时这些资源管理器会打开本地数据库事务，然后开始执行数据库事务，但执行完成后并不会立刻提交事务，而是向事务管理器返回已就绪（Ready）或未就绪（Not Ready）状态。如果各个参与节点都返回状态了，就会进入第二阶段。



到了第二阶段，如果资源管理器返回的都是就绪状态，事务管理器则会向各个资源管理器发送提交（Commit）通知，资源管理器则会完成本地数据库的事务提交，最终返回提交结果给事务管理器。



在第二阶段中，如果任意资源管理器返回了未就绪状态，此时事务管理器会向所有资源管理器发送事务回滚（Rollback）通知，此时各个资源管理器就会回滚本地数据库事务，释放资源，并返回结果通知。



但事实上，二阶事务提交也存在一些缺陷。

第一，在整个流程中，我们会发现各个资源管理器节点存在阻塞，只有当所有的节点都准备完成之后，事务管理器才会发出进行全局事务提交的通知，这个过程如果很长，则会有很多节点长时间占用资源，从而影响整个节点的性能。

一旦资源管理器挂了，就会出现一直阻塞等待的情况。类似问题，我们可以通过设置事务超时时间来解决。

第二，仍然存在数据不一致的可能性，例如，在最后通知提交全局事务时，由于网络故障，部分节点有可能收不到通知，由于这部分节点没有提交事务，就会导致数据不一致的情况出现。

而三阶事务（3PC）的出现就是为了减少此类问题的发生。

3PC 把 2PC 的准备阶段分为了准备阶段和预处理阶段，在第一阶段只是询问各个资源节点是否可以执行事务，而在第二阶段，所有的节点反馈可以执行事务，才开始执行事务操作，最后在第三阶段执行提交或回滚操作。并且在事务管理器和资源管理器中都引入了超时机制，如果在第三阶段，资源节点一直无法收到来自资源管理器的提交或回滚请求，它就会在超时之后，继续提交事务。

所以 3PC 可以通过超时机制，避免管理器挂掉所造成的时间阻塞问题，但其实这样还是无法解决在最后提交全局事务时，由于网络故障无法通知到一些节点的问题，特别是回滚通知，这样会导致事务等待超时从而默认提交。

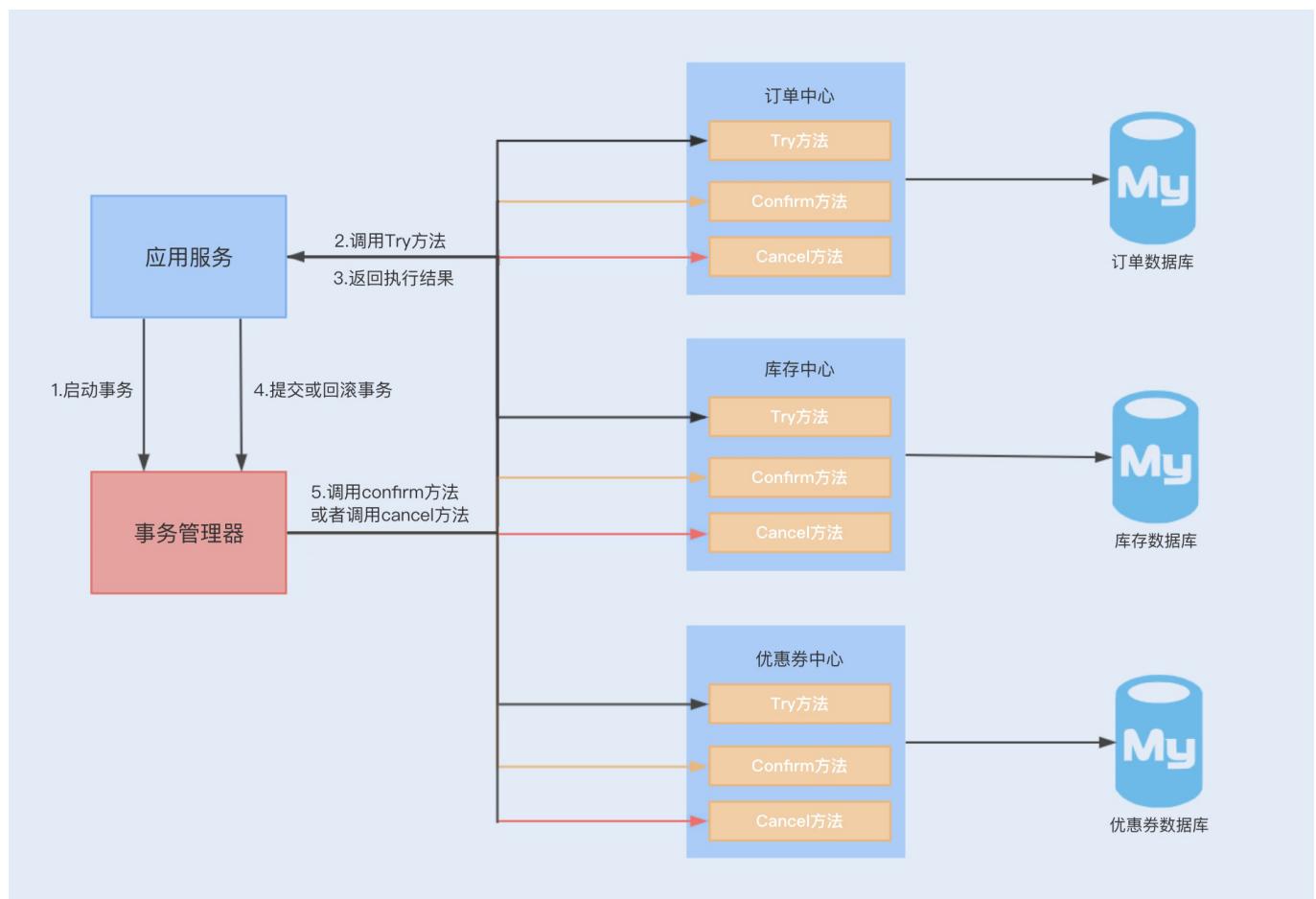
3. 事务补偿机制 (TCC)

以上这种基于 XA 规范实现的事务提交，由于阻塞等性能问题，有着比较明显的低性能、低吞吐的特性。所以在抢购活动中使用该事务，很难满足系统的并发性能。

除了性能问题，JTA 只能解决同一服务下操作多数据源的分布式事务问题，换到微服务架构下，可能存在同一个事务操作，分别在不同服务上连接数据源，提交数据库操作。

而 TCC 正是为了解决以上问题而出现的一种分布式事务解决方案。TCC 采用最终一致性的方式实现了一种柔性分布式事务，与 XA 规范实现的二阶事务不同的是，TCC 的实现是基于服务层实现的一种二阶事务提交。

TCC 分为三个阶段，即 Try、Confirm、Cancel 三个阶段。



Try 阶段：主要尝试执行业务，执行各个服务中的 Try 方法，主要包括预留操作；

Confirm 阶段：确认 Try 中的各个方法执行成功，然后通过 TM 调用各个服务的 Confirm 方法，这个阶段是提交阶段；

Cancel 阶段：当在 Try 阶段发现其中一个 Try 方法失败，例如预留资源失败、代码异常等，则会触发 TM 调用各个服务的 Cancel 方法，对全局事务进行回滚，取消执行业务。

以上执行只是保证 Try 阶段执行时成功或失败的提交和回滚操作，你肯定会想到，如果在 Confirm 和 Cancel 阶段出现异常情况，那 TCC 该如何处理呢？此时 TCC 会不停地重试调用失败的 Confirm 或 Cancel 方法，直到成功为止。

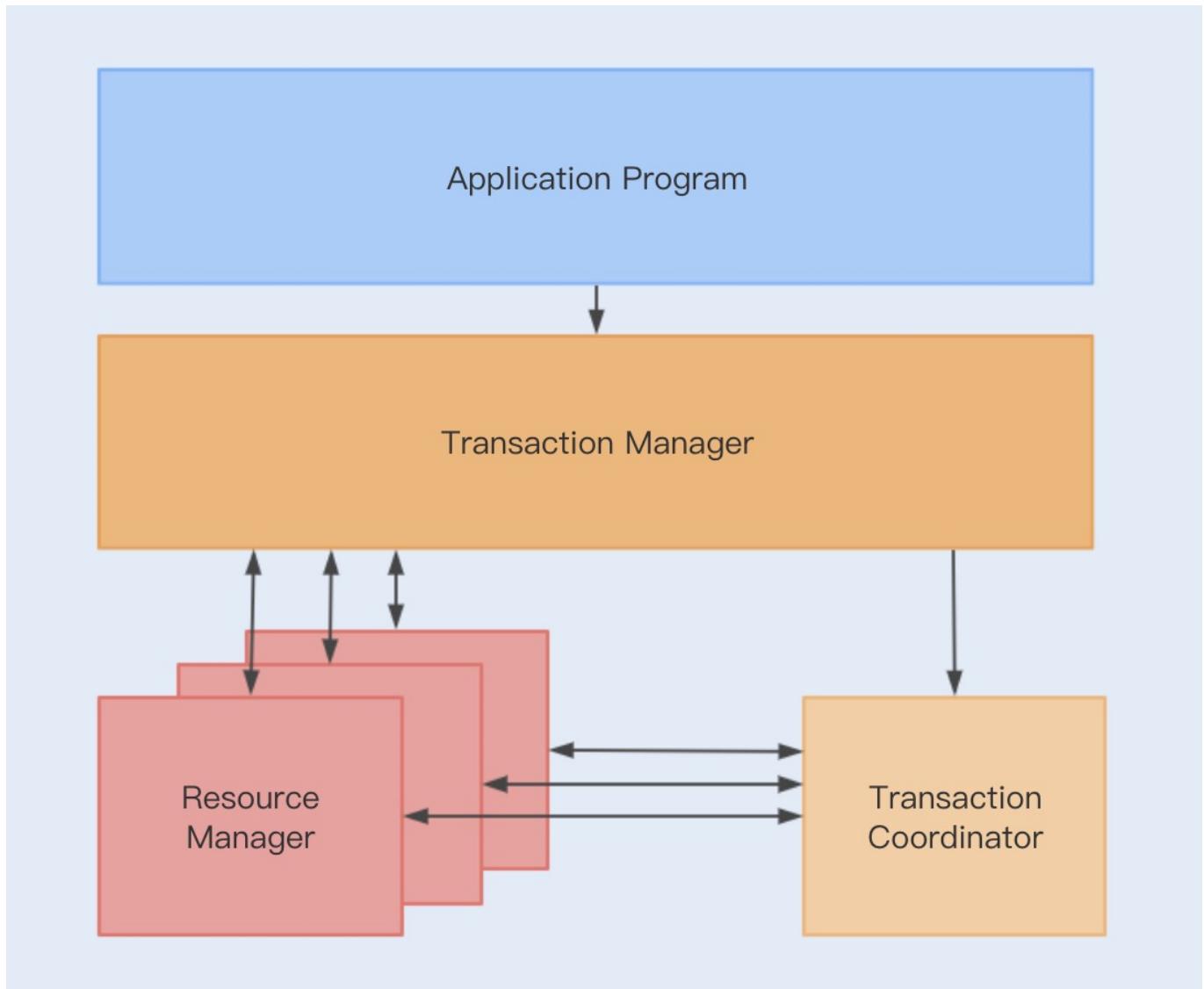
但 TCC 补偿性事务也有比较明显的缺点，那就是对业务的侵入性非常大。

首先，我们需要在业务设计的时候考虑预留资源；然后，我们需要编写大量业务性代码，例如 Try、Confirm、Cancel 方法；最后，我们还需要为每个方法考虑幂等性。这种事务的实现和维护成本非常高，但综合来看，这种实现是目前大家最常用的分布式事务解决方案。

4. 业务无侵入方案——Seata(Fescar)

Seata 是阿里去年开源的一套分布式事务解决方案，开源一年多已经有一万多 star 了，可见受欢迎程度非常之高。

Seata 的基础建模和 DTP 模型类似，只不过前者是将事务管理器分得更细了，抽出一个事务协调器（Transaction Coordinator 简称 TC），主要维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。而 TM 则负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。如下图所示：



按照[Github](#)中的说明介绍，整个事务流程为：

TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID；

XID 在微服务调用链路的上下文中传播；

RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖；

TM 向 TC 发起针对 XID 的全局提交或回滚决议；

TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求。

Seata 与其它分布式最大的区别在于，它在第一提交阶段就已经将各个事务操作 commit 了。Seata 认为在一个正常的业务下，各个服务提交事务的概率是成功的，这种事务提交操作可以节约两个阶段持有锁的时间，从而提高整体的执行效率。

那如果在第一阶段就已经提交了事务，那我们还谈何回滚呢？

Seata 将 RM 提升到了服务层，通过 JDBC 数据源代理解析 SQL，把业务数据在更新前后的数据镜像组织成回滚日志，利用本地事务的 ACID 特性，将业务数据的更新和回滚日志的写入在同一个本地事务中提交。

如果 RM 决议要全局回滚，会通知 RM 进行回滚操作，通过 XID 找到对应的回滚日志记录，通过回滚记录生成反向更新 SQL，进行更新回滚操作。

以上我们可以保证一个事务的原子性和一致性，但隔离性如何保证呢？

Seata 设计通过事务协调器维护的全局写排它锁，来保证事务间的写隔离，而读写隔离级别则默认为未提交读的隔离级别。

总结

在同服务多数据源操作不同数据库的情况下，我们可以使用基于 XA 规范实现的分布式事务，在 Spring 中有成熟的 JTA 框架实现了 XA 规范的二阶事务提交。事实上，二阶事务除了性能方面存在严重的阻塞问题之外，还有可能导致数据不一致，我们应该慎重考虑使用这种二阶事务提交。

在跨服务的分布式事务下，我们可以考虑基于 TCC 实现的分布式事务，常用的中间件有 TCC-Transaction。TCC 也是基于二阶事务提交原理实现的，但 TCC 的二阶事务提交是提到了服务层实现。TCC 方式虽然提高了分布式事务的整体性能，但也给业务层带来了非常大的工作量，对应用服务的侵入性非常强，但这是大多数公司目前所采用的分布式事务解决方案。

Seata 是一种高效的分布式事务解决方案，设计初衷就是解决分布式带来的性能问题以及侵入性问题。但目前 Seata 的稳定性有待验证，例如，在 TC 通知 RM 开始提交事务后，TC 与 RM 的连接断开了，或者 RM 与数据库的连接断开了，都不能保证事务的一致性。

43 | 如何使用缓存优化系统性能?



缓存是我们提高系统性能的一项必不可少的技术，无论是前端、还是后端，都应用到了缓存技术。前端使用缓存，可以降低多次请求服务的压力；后端使用缓存，可以降低数据库操作的压力，提升读取数据的性能。

今天我们将从前端到服务端，系统了解下各个层级的缓存实现，并分别了解下各类缓存的优缺点以及应用场景。

前端缓存技术

如果你是一位 Java 开发工程师，你可能会想，我们有必要去了解前端的技术吗？

不想当将军的士兵不是好士兵，作为一个技术人员，不想做架构师的开发不是好开发。作为架构工程师的话，我们就很有必要去了解前端的知识点了，这样有助于我们设计和优化系统。前端做缓存，可以缓解服务端的压力，减少带宽的占用，同时也可以提升前端的查询性能。

1. 本地缓存

平时使用拦截器（例如 Fiddler）或浏览器 Debug 时，我们经常会发现一些接口返回 304 状态码 + Not Modified 字符串，如下图中的 Web 首页。

Name	Value
time.geekbang.org	
jweixin-1.3.2.js	
font_372689_nw1guejwd2qjs	
aliplayer-min.js	
manifest.4613910b6b1cf3f68ee5js	
vendor.8d9ec681098078579b00.js	
app.bed13e4d38d8bde91000.js	
analytics.js	
font_372689_784f52l4cke.woff2	
hm.js?022f847c4e3acd44d4a2481d9187f1e6	
collect?v=1&_v=j79&a=802424559&t=pageview&s=1&dl=...03082599-6&_gid=17	
auth	
hm.gif?cc=1&ck=1&cl=24-bit&ds=1920x1080&vl=937&et=...788&ct=!!&tt=%E6%	
20.430b81497c50833f6ec3.js	
topList	
hot words	

Headers tab details:

- Request URL: https://time.geekbang.org/
- Request Method: GET
- Status Code: 304 Not Modified
- Remote Address: 39.106.233.176:443
- Referrer Policy: origin

Response Headers:

- Connection: keep-alive
- Date: Sat, 24 Aug 2019 03:39:23 GMT
- ETag: "5d5fd77-1003"
- Last-Modified: Fri, 23 Aug 2019 12:35:03 GMT
- Set-Cookie: SERVERID=3431a294a18c59fc8f5805662e2bd51e|1566617963|1566609990; Path=/
- Strict-Transport-Security: max-age=15768000

Request Headers:

如果我们对前端缓存技术不了解，就很容易对此感到困惑。浏览器常用的一种缓存就是这种基于 304 响应状态实现的本地缓存了，通常这种缓存被称为协商缓存。

协商缓存，顾名思义就是与服务端协商之后，通过协商结果来判断是否使用本地缓存。

一般协商缓存可以基于请求头部中的 If-Modified-Since 字段与返回头部中的 Last-Modified 字段实现，也可以基于请求头部中的 If-None-Match 字段与返回头部中的 ETag 字段来实现。

两种方式的实现原理是一样的，前者是基于时间实现的，后者是基于一个唯一标识实现的，相对来说后者可以更加准确地判断文件内容是否被修改，避免由于时间篡改导致的不可靠问题。下面我们再来了解下整个缓存的实现流程：

当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 ETag 唯一标识，这个唯一标识的值是根据当前请求的资源生成的；

当浏览器再次请求访问服务器中的该资源时，会在 Request 头部加上 If-None-Match 字段，该字段的值就是 Response 头部加上 ETag 唯一标识；

服务器再次收到请求后，会根据请求中的 If-None-Match 值与当前请求的资源生成的唯一标识进行比较，如果值相等，则返回 304 Not Modified，如果不相等，则在 Response 头部加上新的 ETag 唯一标识，并返回资源；

如果浏览器收到 304 的请求响应状态码，则会从本地缓存中加载资源，否则更新资源。

本地缓存中除了这种协商缓存，还有一种就是强缓存的实现。

强缓存指的是只要判断缓存没有过期，则直接使用浏览器的本地缓存。如下图中，返回的是 200 状态码，但在 size 项中标识的是 memory cache。

Name	Status	Type	Initiator	Size	Time	Waterfall
app.bed13e4d38d8bde91000.js	304	script	[index]	222 B	49...	
analytics.js	200	script	[index]:22	(memory cache)	0...	
font_372689_784f5214cke.woff2	200	font	[index]	(memory cache)	0...	
hm.js?022f847c4e3acd44d4a2481d9187f1e6	304	script	[index]:30	208 B	61...	
collect?v=1&_v=j79&a=802424559&t=pageview&_s=1&dl=...03082...	200	gif	analytics.js:16	63 B	16...	
auth	200	xhr	vendor.8d9ec68...js:6	690 B	14...	
hm.gif?cc=1&cl=1&cl=24-bit&ds=1920x1080&vl=937&et=...788&ct...	200	gif	hm.js?022f847...:21	299 B	51...	
20.430b81497c50833f6ec3.js	304	script	manifest.4613910...js:1	350 B	14...	
topList	200	xhr	vendor.8d9ec68...js:6	8.9 KB	10...	
hot_words	200	xhr	vendor.8d9ec68...js:6	716 B	53...	
logo_pc@2x.90583da.png	304	png	[index]	345 B	18...	
favicon-32x32.jpg	200	jpeg	Other	3.2 KB	23...	
dd8cbc79f017d1b01f643c7ea929d79e.png	304	png	vendor.8d9ec68...js:1	341 B	27...	
4aebe8fb752fa21a0fd989a45d9847c3.png	304	png	vendor.8d9ec68...js:1	325 B	29...	
9c223ccae33c5245a3009857582f1df1.png	304	png	vendor.8d9ec68...js:1	332 B	34...	
b683240befccbcaa86da8f382d3a11ajpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	308 B	37...	
2d5e1c53fd7aa2a8f7663db0dae0ee12.jpg?x-oss-process=image/resi...	304	jpeg	vendor.8d9ec68...js:1	284 B	37...	
2aa01da86c8fb550b32c79f8ca4f67b0.jpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	292 B	36...	
ca39e9384b7793c4713dcc762da5f5c0.jpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	289 B	36...	

强缓存是利用 Expires 或者 Cache-Control 这两个 HTTP Response Header 实现的，它们都用来表示资源在客户端缓存的有效期。

Expires 是一个绝对时间，而 Cache-Control 是一个相对时间，即一个过期时间大小，与协商缓存一样，基于 Expires 实现的强缓存也会因为时间问题导致缓存管理出现问题。我建议使用 Cache-Control 来实现强缓存。具体的实现流程如下：

当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 Cache-Control，Cache-Control 中设置了过期时间大小；

浏览器再次请求访问服务器中的该资源时，会先通过请求资源的时间与 Cache-Control 中设置的过期时间大小，来计算出该资源是否过期，如果没有，则使用该缓存，否则请求服务器；

服务器再次收到请求后，会再次更新 Response 头部的 Cache-Control。

2. 网关缓存

除了以上本地缓存，我们还可以在网关中设置缓存，也就是我们熟悉的 CDN。

CDN 缓存是通过不同地点的缓存节点缓存资源副本，当用户访问相应的资源时，会调用最近的 CDN 节点返回请求资源，这种方式常用于视频资源的缓存。

服务层缓存技术

前端缓存一般用于缓存一些不常修改的常量数据或一些资源文件，大部分接口请求的数据都缓存在了服务端，方便统一管理缓存数据。

服务端缓存的初衷是为了提升系统性能。例如，数据库由于并发查询压力过大，可以使用缓存减轻数据库压力；在后台管理中的一些报表计算类数据，每次请求都需要大量计算，消耗系统 CPU 资源，我们可以使用缓存来保存计算结果。

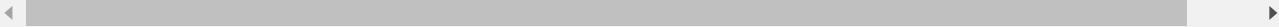
服务端的缓存也分为进程缓存和分布式缓存，在 Java 中进程缓存就是 JVM 实现的缓存，常见的有我们经常使用的容器类，`ArrayList`、`ConcurrentHashMap` 等，分布式缓存则是基于 Redis 实现的缓存。

1. 进程缓存

对于进程缓存，虽然数据的存取会更加高效，但 JVM 的堆内存数量是有限的，且在分布式环境下很难同步各个服务间的缓存更新，所以**我们一般缓存一些数据量不大、更新频率较低的数据**。常见的实现方式如下：

 复制代码

```
1 // 静态常量
2 public final static String url = "https://time.geekbang.org";
3 //list 容器
4 public static List<String> cacheList = new Vector<String>();
5 //map 容器
6 private static final Map<String, Object> cacheMap= new ConcurrentHashMap<String, Object>
```



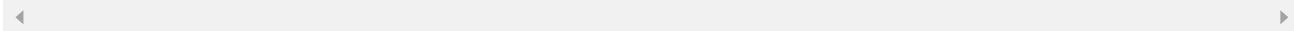
除了 Java 自带的容器可以实现进程缓存，我们还可以基于 Google 实现的一套内存缓存组件 Guava Cache 来实现。

Guava Cache 适用于高并发的多线程缓存，它和 ConcurrentHashMap 一样，都是基于分段锁实现的并发缓存。

Guava Cache 同时也实现了数据淘汰机制，当我们设置了缓存的最大值后，当存储的数据超过了最大值时，它就会使用 LRU 算法淘汰数据。我们可以通过以下代码了解下 Guava Cache 的实现：

 复制代码

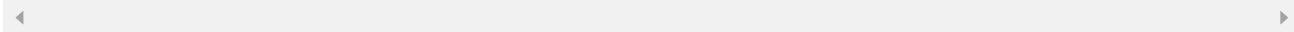
```
1 public class GuavaCacheDemo {  
2     public static void main(String[] args) {  
3         Cache<String, String> cache = CacheBuilder.newBuilder()  
4             .maximumSize(2)  
5             .build();  
6         cache.put("key1", "value1");  
7         cache.put("key2", "value2");  
8         cache.put("key3", "value3");  
9         System.out.println(" 第一个值: " + cache.getIfPresent("key1"));  
10        System.out.println(" 第二个值: " + cache.getIfPresent("key2"));  
11        System.out.println(" 第三个值: " + cache.getIfPresent("key3"));  
12    }  
13 }
```



运行结果：

 复制代码

```
1 第一个值: null  
2 第二个值: value2  
3 第三个值: value3
```



那如果我们的数据量比较大，且数据更新频繁，又是在分布式部署的情况下，想要使用 JVM 堆内存作为缓存，这时我们又该如何去实现呢？

Ehcache 是一个不错的选择，Ehcache 经常在 Hibernate 中出现，主要用来缓存查询数据结果。Ehcache 是 Apache 开源的一套缓存管理类库，是基于 JVM 堆内存实现的缓存，同时具备多种缓存失效策略，支持磁盘持久化以及分布式缓存机制。

2. 分布式缓存

由于高并发对数据一致性的要求比较严格，我一般不建议使用 Ehcache 缓存有一致性要求的数据。对于分布式缓存，我们建议使用 Redis 来实现，Redis 相当于一个内存数据库，由于是纯内存操作，又是基于单线程串行实现，查询性能极高，读速度超过了 10W 次 / 秒。

Redis 除了高性能的特点之外，还支持不同类型的数据结构，常见的有 string、list、set、hash 等，还支持数据淘汰策略、数据持久化以及事务等。

两种缓存讲完了，接下来我们看看其中可能出现的问题。

数据库与缓存数据一致性问题

在查询缓存数据时，我们会先读取缓存，如果缓存中没有该数据，则会去数据库中查询，之后再放入到缓存中。

当我们的数据被缓存之后，一旦数据被修改（修改时也是删除缓存中的数据）或删除，我们就需要同时操作缓存和数据库。这时，就会存在一个数据不一致的问题。

例如，在并发情况下，当 A 操作使得数据发生删除变更，那么该操作会先删除缓存中的数据，之后再去删除数据库中的数据，此时若是还没有删除成功，另外一个请求查询操作 B 进来了，发现缓存中已经没有了数据，则会去数据库中查询，此时发现有数据，B 操作获取之后又将数据存放在了缓存中，随后数据库的数据又被删除了。此时就出现了数据不一致的情况。

那如果先删除数据库，再删除缓存呢？

我们可以试一试。在并发情况下，当 A 操作使得数据发生删除变更，那么该操作会先删除了数据库的操作，接下来删除缓存，失败了，那么缓存中的数据没有被删除，而数据库的数据已经被删除了，同样会存在数据不一致的问题。

所以，我们还是需要先做缓存删除操作，再去完成数据库操作。那我们又该如何避免高并发下，数据更新删除操作所带来的数据不一致的问题呢？

通常的解决方案是，如果我们需要使用一个线程安全队列来缓存更新或删除的数据，当 A 操作变更数据时，会先删除一个缓存数据，此时通过线程安全的方式将缓存数据放入到队列中，并通过一个线程进行数据库的数据删除操作。

当有另一个查询请求 B 进来时，如果发现缓存中没有该值，则会先去队列中查看该数据是否正在被更新或删除，如果队列中有该数据，则阻塞等待，直到 A 操作数据库成功之后，唤醒该阻塞线程，再去数据库中查询该数据。

但其实这种实现也存在很多缺陷，例如，可能存在读请求被长时间阻塞，高并发时低吞吐量等问题。所以我们在考虑缓存时，如果数据更新比较频繁且对数据有一定的一致性要求，我通常不建议使用缓存。

缓存穿透、缓存击穿、缓存雪崩

对于分布式缓存实现大数据的存储，除了数据不一致的问题以外，还有缓存穿透、缓存击穿、缓存雪崩等问题，我们平时实现缓存代码时，应该充分、全面地考虑这些问题。

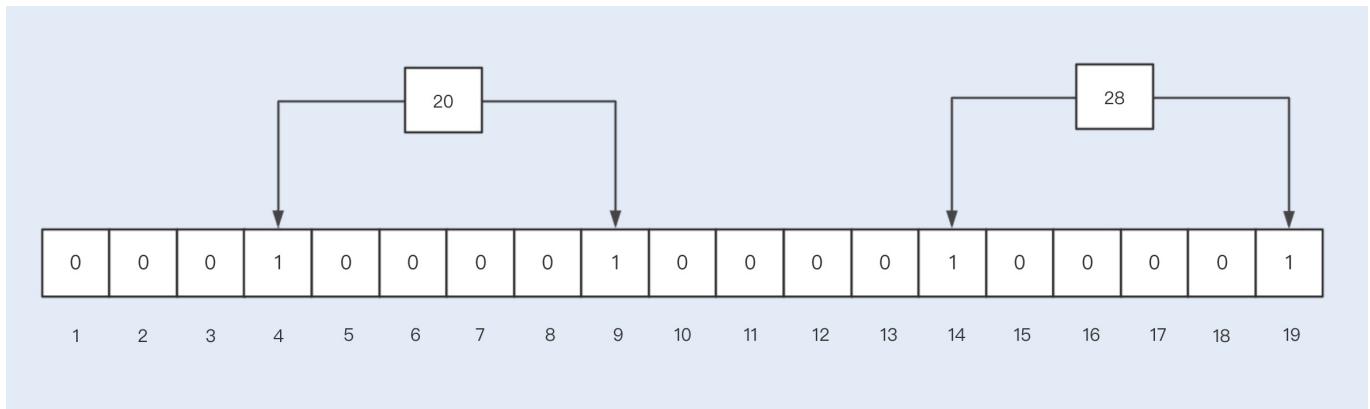
缓存穿透是指大量查询没有命中缓存，直接去到数据库中查询，如果查询量比较大，会导致数据库的查询流量大，对数据库造成压力。

通常有两种解决方案，一种是将第一次查询的空值缓存起来，同时设置一个比较短的过期时间。但这种解决方案存在一个安全漏洞，就是当黑客利用大量没有缓存的 key 攻击系统时，缓存的内存会被占满溢出。

另一种则是使用布隆过滤算法（BloomFilter），该算法可以用于检查一个元素是否存在，返回结果有两种：可能存在或一定不存在。这种情况很适合用来解决故意攻击系统的缓存穿透问题，在最初缓存数据时也将 key 值缓存在布隆过滤器的 BitArray 中，当有 key 值查询时，对于一定不存在的 key 值，我们可以直接返回空值，对于可能存在的 key 值，我们会去缓存中查询，如果没有值，再去数据库中查询。

BloomFilter 的实现原理与 Redis 中的 BitMap 类似，首先初始化一个 m 长度的数组，并且每个 bit 初始化值都是 0，当插入一个元素时，会使用 n 个 hash 函数来计算出 n 个不同的值，分别代表所在数组的位置，然后再将这些位置的值设置为 1。

假设我们插入两个 key 值分别为 20,28 的元素，通过两次哈希函数取模后的值分别为 4,9 以及 14,19，因此 4,9 以及 14,19 都被设置为 1。



那为什么说 BloomFilter 返回的结果是可能存在和一定不存在呢？

假设我们查找一个元素 25，通过 n 次哈希函数取模后的值为 1,9,14。此时在 BitArray 中肯定是不存在的。而当我们查找一个元素 21 的时候， n 次哈希函数取模后的值为 9,14，此时会返回可能存在的结果，但实际上也是不存在的。

BloomFilter 不允许删除任何元素的，为什么？假设以上 20,25,28 三个元素都存在于 BitArray 中，取模的位置值分别为 4,9、1,9,14 以及 14,19，如果我们要删除元素 25，此时需要将 1,9,14 的位置都置回 0，这样就影响 20,28 元素了。

因此，BloomFilter 是不允许删除任何元素的，这样会导致已经删除的元素依然返回可能存在的结果，也会影响 BloomFilter 判断的准确率，解决的方法则是重建一个 BitArray。

那什么缓存击穿呢？在高并发情况下，同时查询一个 key 时，key 值由于某种原因突然失效（设置过期时间或缓存服务宕机），就会导致同一时间，这些请求都去查询数据库了。这种情况经常出现在查询热点数据的场景中。通常我们会在查询数据库时，使用排斥锁来实现有序地请求数据库，减少数据库的并发压力。

缓存雪崩则与缓存击穿差不多，区别就是失效缓存的规模。雪崩一般是指发生大规模的缓存失效情况，例如，缓存的过期时间同一时间过期了，缓存服务宕机了。对于大量缓存的过期时间同一时间过期的问题，我们可以采用分散过期时间来解决；而针对缓存服务宕机的情况，我们可以采用分布式集群来实现缓存服务。

总结

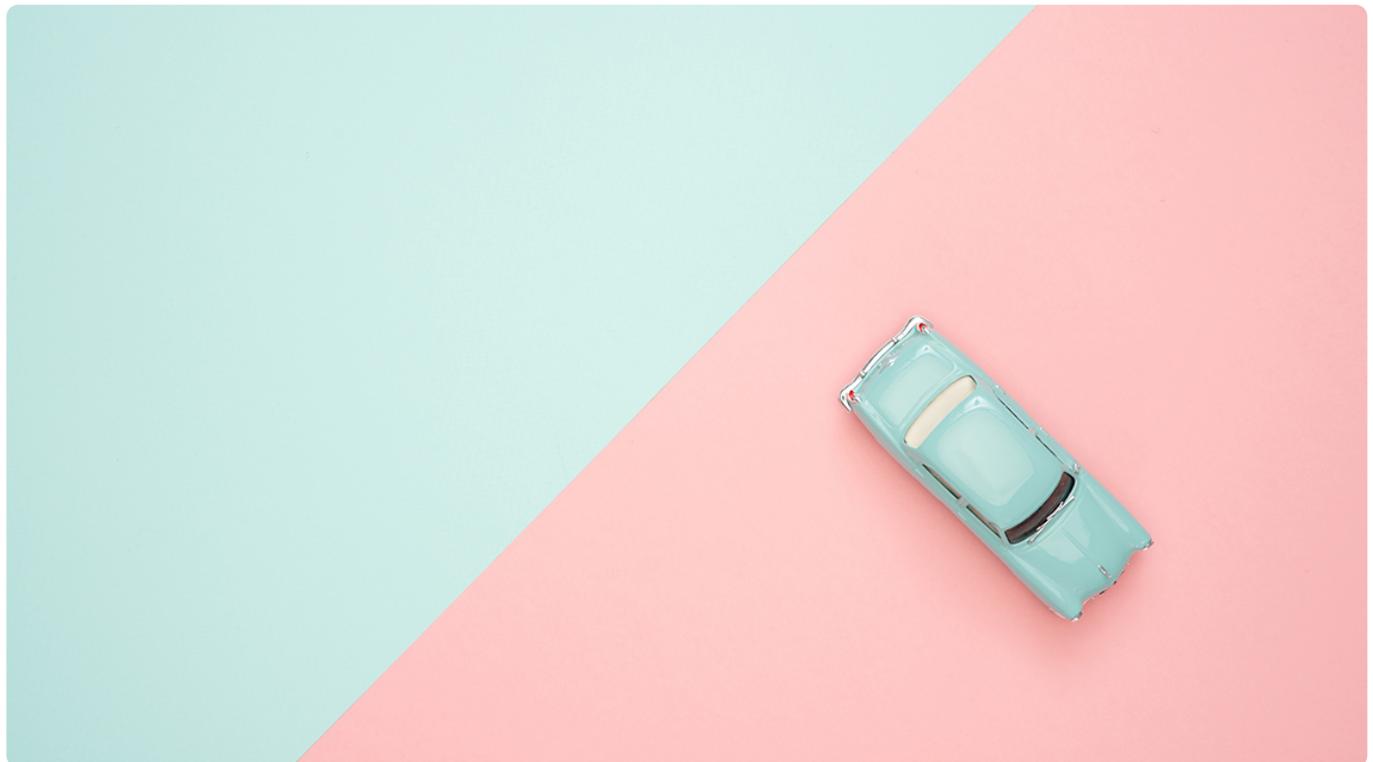
从前端到后端，对于一些不常变化的数据，我们都可以将其缓存起来，这样既可以提高查询效率，又可以降低请求后端的压力。对于前端来说，一些静态资源文件都是会被缓存在浏览器端，除了静态资源文件，我们还可以缓存一些常量数据，例如商品信息。

服务端的缓存，包括了 JVM 的堆内存作为缓存以及 Redis 实现的分布式缓存。如果是一些不常修改的数据，数据量小，且对缓存数据没有严格的一致性要求，我们就可以使用堆内存缓存数据，这样既实现简单，查询也非常高效。如果数据量比较大，且是经常被修改的数据，或对缓存数据有严格的一致性要求，我们就可以使用分布式缓存来存储。

在使用后端缓存时，我们应该注意数据库和缓存数据的修改导致的数据不一致问题，如果对缓存与数据库数据有非常严格的一致性要求，我就不建议使用缓存了。

同时，我们应该针对大量请求缓存的接口做好预防工作，防止查询缓存的接口出现缓存穿透、缓存击穿和缓存雪崩等问题。

44 | 记一次双十一抢购性能瓶颈调优



每年的双十一都是很多研发部门最头痛的节日，由于这个节日比较特殊，公司一般都会准备大量的抢购活动，相应的瞬时高并发请求对系统来说是个不小的考验。

还记得我们公司商城第一次做双十一抢购活动，优惠力度特别大，购买量也很大，提交订单的接口 TPS 一度达到了 10W。在首波抢购时，后台服务监控就已经显示服务器的各项指标都超过了 70%，CPU 更是一直处于 400%（4 核 CPU），数据库磁盘 I/O 一直处于 100%

状态。由于瞬时写入日志量非常大，导致我们的后台服务监控在短时间内，无法实时获取到最新的请求监控数据，此时后台开始出现一系列的异常报警。

更严重的系统问题是出现在第二波的抢购活动中，由于第一波抢购时我们发现后台服务的压力比较大，于是就横向扩容了服务，但却没能缓解服务的压力，反而在第二波抢购中，我们的系统很快就出现了宕机。

这次活动暴露出来的问题很多。首先，由于没有限流，超过预期的请求量导致了系统卡顿；其次，基于 Redis 实现的分布式锁分发抢购名额的功能抛出了大量异常；再次，就是我们误判了横向扩容服务可以起到的作用，其实第一波抢购的性能瓶颈是在数据库，横向扩容服务反而又增加了数据库的压力，起到了反作用；最后，就是在服务挂掉的情况下，丢失了异步处理的业务请求。

接下来我会以上面的这个案例为背景，重点讲解抢购业务中的性能瓶颈该如何调优。

抢购业务流程

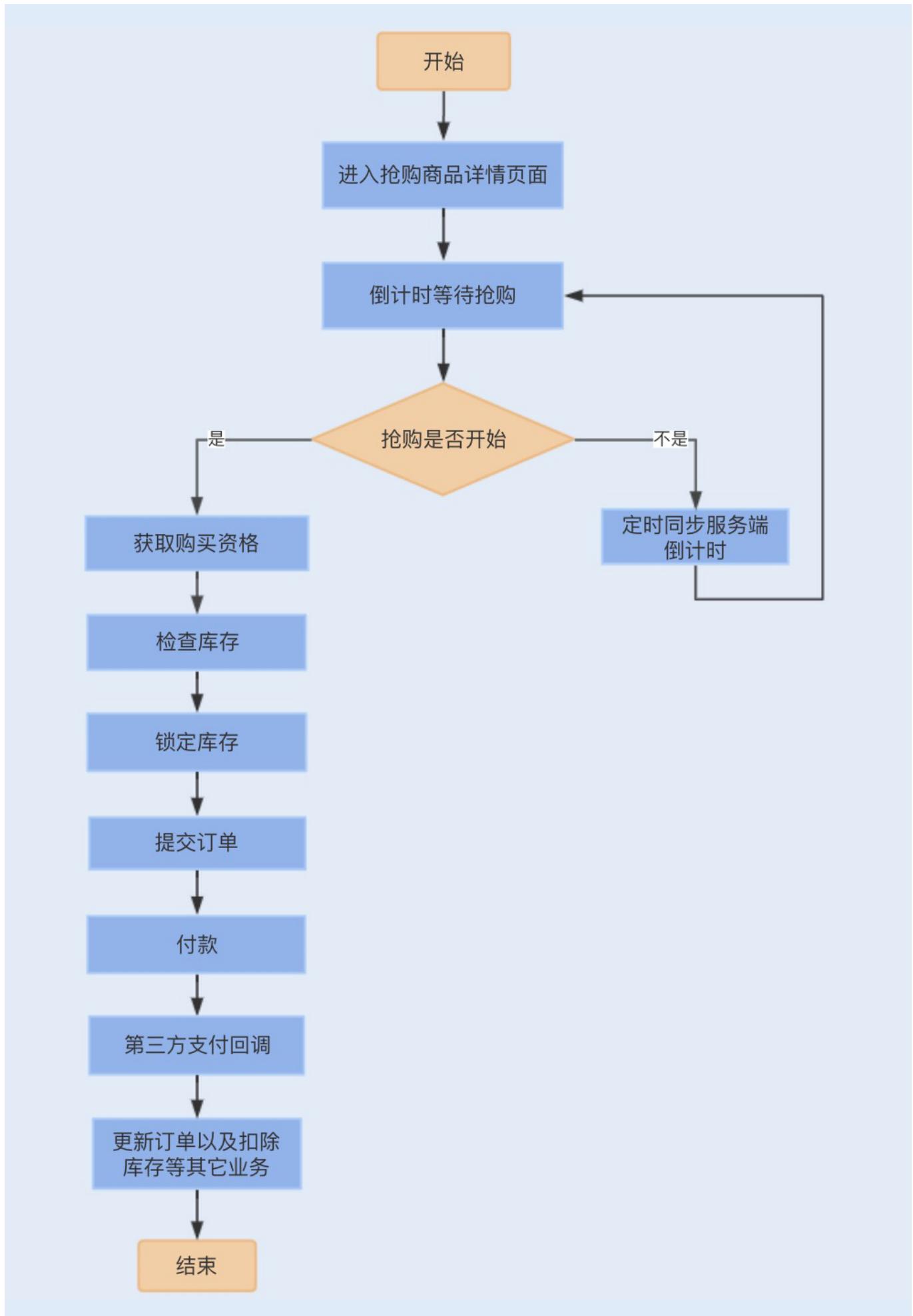
在进行具体的性能问题讨论之前，我们不妨先来了解下一个常规的抢购业务流程，这样方便我们更好地理解一个抢购系统的性能瓶颈以及调优过程。

用户登录后会进入到商品详情页面，此时商品购买处于倒计时状态，购买按钮处于置灰状态。

当购买倒计时结束后，用户点击购买商品，此时用户需要排队等待获取购买资格，如果没有获取到购买资格，抢购活动结束，反之，则进入提交页面。

用户完善订单信息，点击提交订单，此时校验库存，并创建订单，进入锁定库存状态，之后，用户支付订单款。

当用户支付成功后，第三方支付平台将产生支付回调，系统通过回调更新订单状态，并扣除数据库的实际库存，通知用户购买成功。



抢购系统中的性能瓶颈

熟悉了一个常规的抢购业务流程之后，我们再来看看抢购中都有哪些业务会出现性能瓶颈。

1. 商品详情页面

如果你有过抢购商品的经验，相信你遇到过这样一种情况，在抢购马上到来的时候，商品详情页面几乎是无法打开的。

这是因为大部分用户在抢购开始之前，会一直疯狂刷新抢购商品页面，尤其是倒计时一分钟内，查看商品详情页面的请求量会猛增。此时如果商品详情页面没有做好，就很容易成为整个抢购系统中的第一个性能瓶颈。

类似这种问题，我们通常的做法是提前将整个抢购商品页面生成为一个静态页面，并 push 到 CDN 节点，并且在浏览器端缓存该页面的静态资源文件，通过 CDN 和浏览器本地缓存这两种缓存静态页面的方式来实现商品详情页面的优化。

2. 抢购倒计时

在商品详情页面中，存在一个抢购倒计时，这个倒计时是服务端时间的，初始化时间需要从服务端获取，并且在用户点击购买时，还需要服务端判断抢购时间是否已经到了。

如果商品详情每次刷新都去后端请求最新的时间，这无疑将会把整个后端服务拖垮。我们可以改成初始化时间从客户端获取，每隔一段时间主动去服务端刷新同步一次倒计时，这个时间段是随机时间，避免集中请求服务端。这种方式可以避免用户主动刷新服务端的同步时间接口。

3. 获取购买资格

可能你会好奇，在抢购中我们已经通过库存数量限制用户了，那为什么会出现一个获取购买资格的环节呢？

我们知道，进入订单详情页面后，需要填写相关的订单信息，例如收货地址、联系方式等，在这样一个过程中，很多用户可能还会犹豫，甚至放弃购买。如果把这个环节设定为一定能购买成功，那我们就只能让同等库存的用户进来，一旦用户放弃购买，这些商品可能无法再次被其他用户抢购，会大大降低商品的抢购销量。

增加购买资格的环节，选择让超过库存的用户量进来提交订单页面，这样就可以保证有足够的提交订单的用户量，确保抢购活动中商品的销量最大化。

获取购买资格这步的并发量会非常大，还是基于分布式的，通常我们可以通过 Redis 分布式锁来控制购买资格的发放。

4. 提交订单

由于抢购入口的请求量会非常大，可能会占用大量带宽，为了不影响提交订单的请求，我建议将提交订单的子域名与抢购子域名区分开，分别绑定不同网络的服务器。

用户点击提交订单，需要先校验库存，库存足够时，用户先扣除缓存中的库存，再生成订单。如果校验库存和扣除库存都是基于数据库实现的，那么每次都去操作数据库，瞬时的并发量就会非常大，对数据库来说会存在一定的压力，从而会产生性能瓶颈。与获取购买资格一样，我们同样可以通过分布式锁来优化扣除消耗库存的设计。

由于我们已经缓存了库存，所以在提交订单时，库存的查询和冻结并不会给数据库带来性能瓶颈。但在这之后，还有一个订单的幂等校验，为了提高系统性能，我们同样可以使用分布式锁来优化。

而保存订单信息一般都是基于数据库表来实现的，在单表单库的情况下，碰到大量请求，特别是在瞬时高并发的情况下，磁盘 I/O、数据库请求连接数以及带宽等资源都可能会出现性能瓶颈。此时我们可以考虑对订单表进行分库分表，通常我们可以基于 userid 字段来进行 hash 取模，实现分库分表，从而提高系统的并发能力。

5. 支付回调业务操作

在用户支付订单完成之后，一般会有第三方支付平台回调我们的接口，更新订单状态。

除此之外，还可能存在扣减数据库库存的需求。如果我们的库存是基于缓存来实现查询和扣减，那提交订单时的扣除库存就只是扣除缓存中的库存，为了减少数据库的并发量，我们会在用户付款之后，在支付回调的时候去选择扣除数据库中的库存。

此外，还有订单购买成功的短信通知服务，一些商城还提供了累计积分的服务。

在支付回调之后，我们可以通过异步提交的方式，实现订单更新之外的其它业务处理，例如库存扣减、积分累计以及短信通知等。通常我们可以基于 MQ 实现业务的异步提交。

性能瓶颈调优

了解了各个业务流程中可能存在的性能瓶颈，我们再来讨论下商城基于常规优化设计之后，还可能出现的一些性能问题，我们又该如何做进一步调优。

1. 限流实现优化

限流是我们常用的兜底策略，无论是倒计时请求接口，还是抢购入口，系统都应该对它们设置最大并发访问数量，防止超出预期的请求集中进入系统，导致系统异常。

通常我们是在网关层实现高并发请求接口的限流，如果我们使用了 Nginx 做反向代理的话，就可以在 Nginx 配置限流算法。Nginx 是基于漏桶算法实现的限流，这样做的好处是能够保证请求的实时处理速度。

Nginx 中包含了两个限流模块：[ngx http limit conn module](#) 和 [ngx http limit req module](#)，前者是用于限制单个 IP 单位时间内的请求数量，后者是用来限制单位时间内所有 IP 的请求数量。以下分别是两个限流的配置：

 复制代码

```
1 limit_conn_zone $binary_remote_addr zone=addr:10m;
2
3 server {
4     location / {
5         limit_conn addr 1;
6     }
}
```

◀ ▶

 复制代码

```
1 http {
2     limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
3     server {
4         location / {
5             limit_req zone=one burst=5 nodelay;
6         }
7     }
}
```

◀ ▶

在网关层，我们还可以通过 lua 编写 OpenResty 来实现一套限流功能，也可以通过现成的 Kong 安装插件来实现。除了网关层的限流之外，我们还可以基于服务层实现接口的限流，通过 Zuul RateLimit 或 Guava RateLimiter 实现。

2. 流量削峰

瞬间有大量请求进入到系统后台服务之后，首先是要通过 Redis 分布式锁获取购买资格，这个时候我们看到了大量的“`JedisConnectionException Could not get connection from pool`”异常。

这个异常是一个 Redis 连接异常，由于我们当时的 Redis 集群是基于哨兵模式部署的，哨兵模式部署的 Redis 也是一种主从模式，我们在写 Redis 的时候都是基于主库来实现的，在高并发操作一个 Redis 实例就很容易出现性能瓶颈。

你可能会想到使用集群分片的方式来实现，但对于分布式锁来说，集群分片的实现只会增加性能消耗，这是因为我们需要基于 Redission 的红锁算法实现，需要对集群的每个实例进行加锁。

后来我们使用 Redission 插件替换 Jedis 插件，由于 Jedis 的读写 I/O 操作还是阻塞式的，方法调用都是基于同步实现，而 Redission 底层是基于 Netty 框架实现的，读写 I/O 是非阻塞 I/O 操作，且方法调用是基于异步实现。

但在瞬时并发非常大的情况下，依然会出现类似问题，此时，我们可以考虑在分布式锁前面新增一个等待队列，减缓抢购出现的集中式请求，相当于一个流量削峰。当请求的 key 值放入到队列中，请求线程进入阻塞状态，当线程从队列中获取到请求线程的 key 值时，就会唤醒请求线程获取购买资格。

3. 数据丢失问题

无论是服务宕机，还是异步发送给 MQ，都存在请求数据丢失的可能。例如，当第三方支付回调系统时，写入订单成功了，此时通过异步来扣减库存和累计积分，如果应用服务刚好挂掉了，MQ 还没有存储到该消息，那即使我们重启服务，这条请求数据也将无法还原。

重试机制是还原丢失消息的一种解决方案。在以上的回调案例中，我们可以在写入订单时，同时在数据库写入一条异步消息状态，之后再返回第三方支付操作成功结果。在异步业务处理请求成功之后，更新该数据库表中的异步消息状态。

假设我们重启服务，那么系统就会在重启时去数据库中查询是否有未更新的异步消息，如果有，则重新生成 MQ 业务处理消息，供各个业务方消费处理丢失的请求数据。

总结

减少抢购中操作数据库的次数，缩短抢购流程，是抢购系统设计和优化的核心点。

抢购系统的性能瓶颈主要是在数据库，即使我们对服务进行了横向扩容，当流量瞬间进来，数据库依然无法同时响应处理这么多的请求操作。我们可以对抢购业务表进行分库分表，通过提高数据库的处理能力，来提升系统的并发处理能力。

除此之外，我们还可以分散瞬时的高并发请求，流量削峰是最常用的方式，用一个队列，让请求排队等待，然后有序且有限地进入到后端服务，最终进行数据库操作。当我们的队列满了之后，可以将溢出的请求放弃，这就是限流了。通过限流和削峰，可以有效地保证系统不宕机，确保系统的稳定性。