37.为什么 Netty 线程池默认大小为 CPU 核数的 2 倍

有位工作 5 年的小伙伴问我说，为什么 Netty 线程池默认大小为 CPU 核数的 2 倍，今天，我花 2 分钟时间给大家专门分享一下我对这个问题的理解。

另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，想获取的小伙伴可以在我的煮叶简介中找到。

1、分析原因

我们都知道使用多线程的本质是为了提升程序的性能，总体来说有两个最核心的指标，一个延迟，一个吞吐量。延迟指的是发出请求到收到响应的时间，吞吐量指的是 。这两个指标之间有一定的关联，因为同等条件下延迟越短吞吐量越大，但由于它们是不同的维度，一个是时间，一个是空间，并不能相互转换。

因此，提升性能最主要的目的就是要降低延迟，提高吞吐量。

那我们如何来衡量这些性能指标呢？

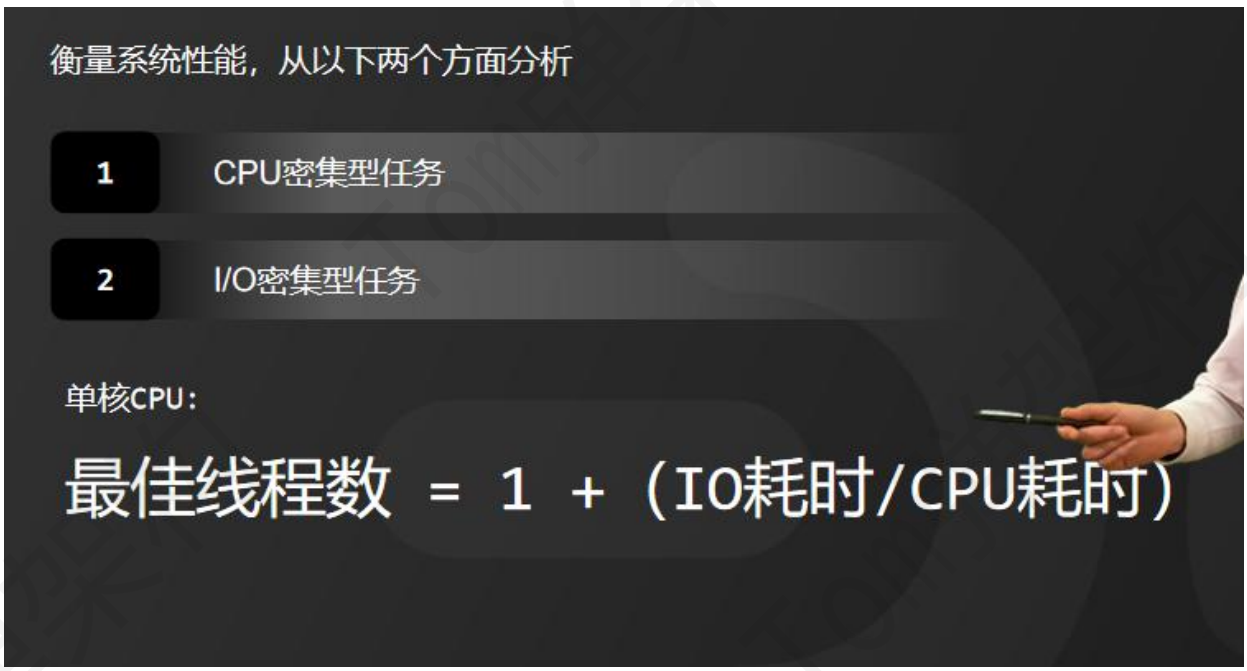
2、如何衡量性能指标

具体来说，要降低延时，就是要提高 CPU 的处理能力。而提高吞吐量，就是要提高 IO 读写效率。那么具体如何衡量系统性能，我从以下两个方面来分析：

我们可以将程序分为是 I/O 密集型任务和 CPU 密集型任务。

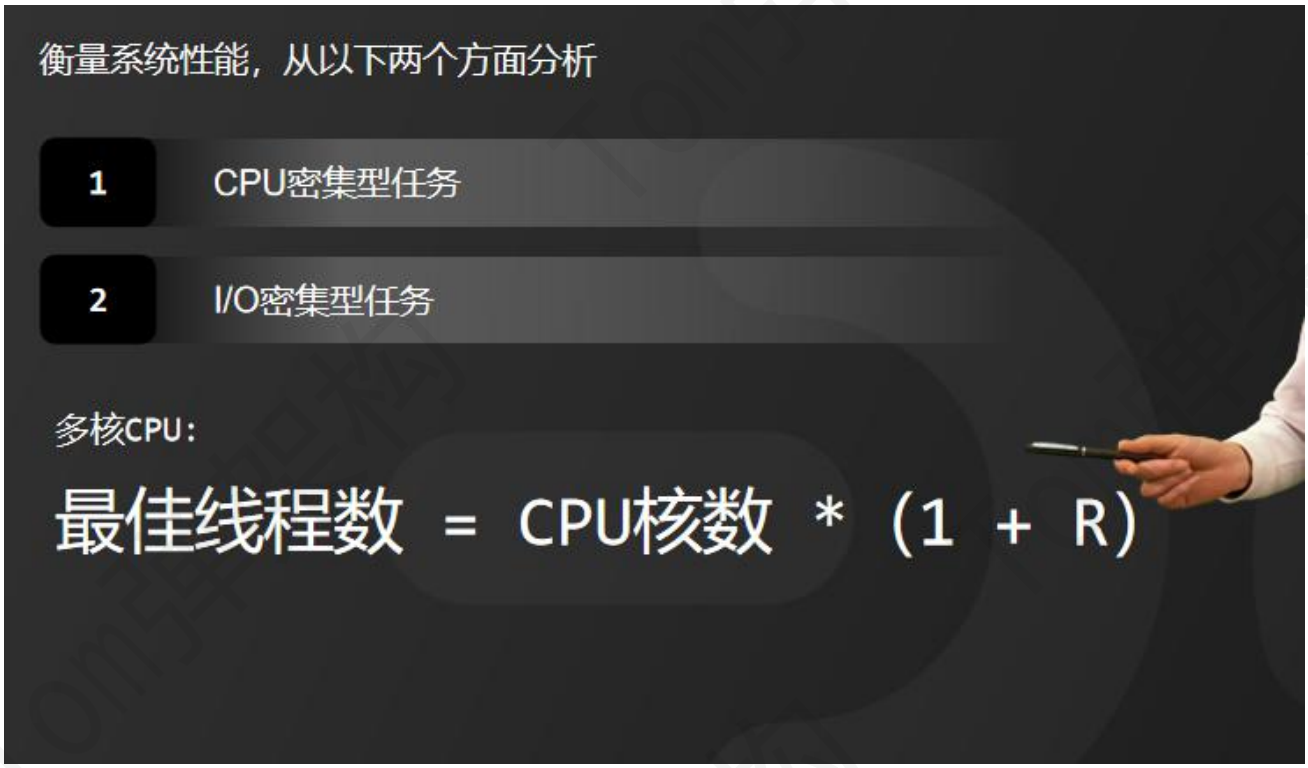
那么第 1 种情况，对于 CPU 密集型任务而言，理论上“线程的数量 = CPU 核数”就是合适的。但是，在实际应用中的线程数量一般会设置为“CPU 核数 + 1”。因为线程有可能因为内存页失效或其他原因导致阻塞，多设置一个线程可以保证 CPU 的利用率。

第 2 种情况，而对于 I/O 密集型任务而言，我们假设 CPU 计算和 I/O 操作的耗时比是 1:1，那么 2 个线程是最合适的。如果 CPU 计算和 I/O 操作的耗时比是 1:2，也就是说 3 个线程是合适的，这样 CPU 和 I/O 设备的利用率都可以达到 100%。根据这个推测，我们可以得到这样一个公式：



不过上面这个公式是针对单核 CPU，如果是多核 CPU 只需要等比扩大就可以了，假设 IO 耗时和 CPU 耗时比为 R，那么计算公式如下：

最佳线程数 = CPU 核数 *（1 + R）



而 Netty 的默认线程池个数，就是假设了 I/O 耗时和 CPU 耗时的占比是 1:1，实际上 Netty 有一个参数叫 ioRatio，默认为 50，它表示在一个轮事件循环中，单个 I/O 线程执行 I/O 事件和执行异步任务的耗时占比为 1:1。相当于 R = 1，代入上面的公式，就可以得出 Netty 默认设置的线程池大小自然就是

默认线程池大小 = CPU 核数 * (1 + 1)

也就 2 倍 CPU 核数大小。而且 Netty 的应用场景主要是 I/O 密集型任务，所以，Netty 这样设计是

有科学性的。

看到了这里，你是不是豁然开朗了呢？

3、总结与使用建议

通过前面的分析，我们已经知道了 Netty 线程池默认大小未 CPU 核数 2 倍的原因，我们在实际开发中，如何来得到一个比较准确的线程池大小呢？

我们可以提前压测，根据压测结果来进行微调。一般情况下，保证生产环境为压测环境的 75%即可。如果修改 Netty 的线程池大小，也一定要考虑 ioRatio 这个参数是否需要调整，因为 2 倍 CPU 核数的大小是假设的 I/O 耗时和 CPU 耗时为 1:1，调整线程大小之后，性能效果也不一定符合期望值。

在大部分场景下，没有必要太过于关注线程池大小怎么配置，I/O 密集型任务使用 Netty 默认配置就可以了。因为，提高吞吐量也不能只简单的只依赖线程池，还可以通过缓存、微服务拆分，优化业务逻辑、优化算法等方式来协作解决。

好了，以上就是我的分享和理解，对望能对大家有所帮助。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

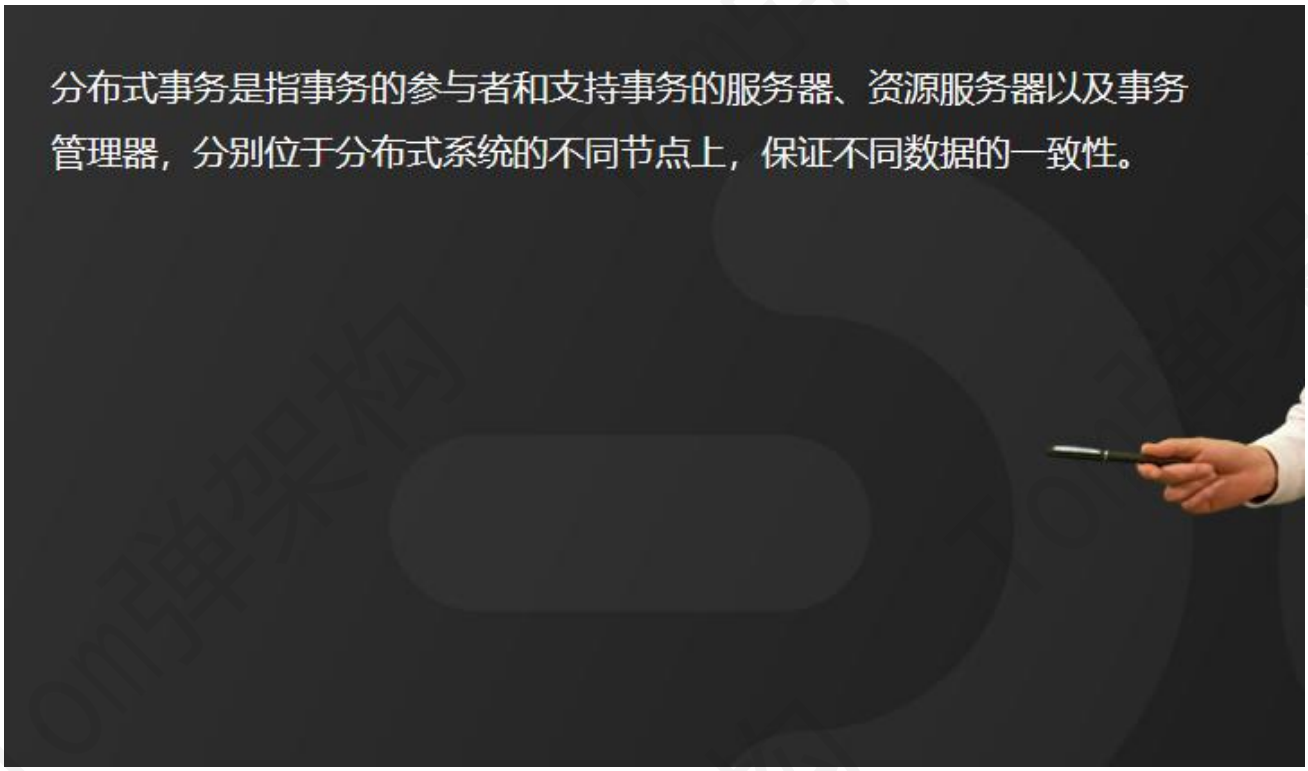
38.谈谈分布式事务的 3 种解决方案

前几天，有一位 10 多年经验的架构师在面试互联网大厂时被问到这样一个问题，说请你谈谈分布式事务的解决方案。那今天，我给大家分享一下我对这个问题的理解。

另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，想获取的小伙伴可以在我的煮叶简介中找到。

1、什么是分布式事务

分布式事务是指事务的参与者和支持事务的服务器、资源服务器以及事务管理器，分别位于分布式系统的不同节点上，保证不同数据的一致性。



比如大型电商系统中的下单场景，会涉及到扣库存、优惠折扣计算、订单 ID 生成等这些服务，通常情况下库存、折扣、订单 ID 生成的服务都位于不同的服务器和数据库中，那么下单是否成功，不仅取决于本地节点的数据库操作，还需要依赖其他服务的执行结果结果，这个时候分布式事务就是保证这些操作要么全部成功，要么全部失败。

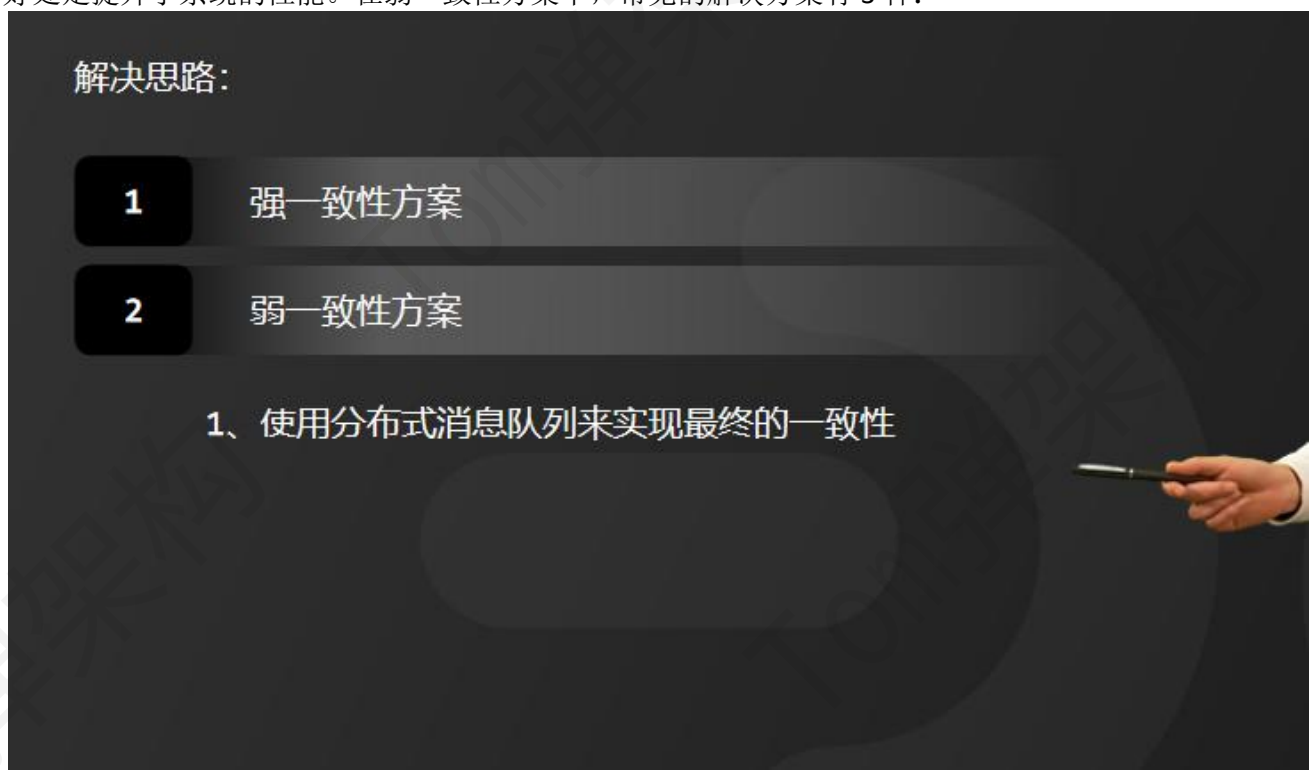
因此，本质上来说，分布式事务就是为了保证不同数据库数据的一致性。

2、分布式事务解决方案

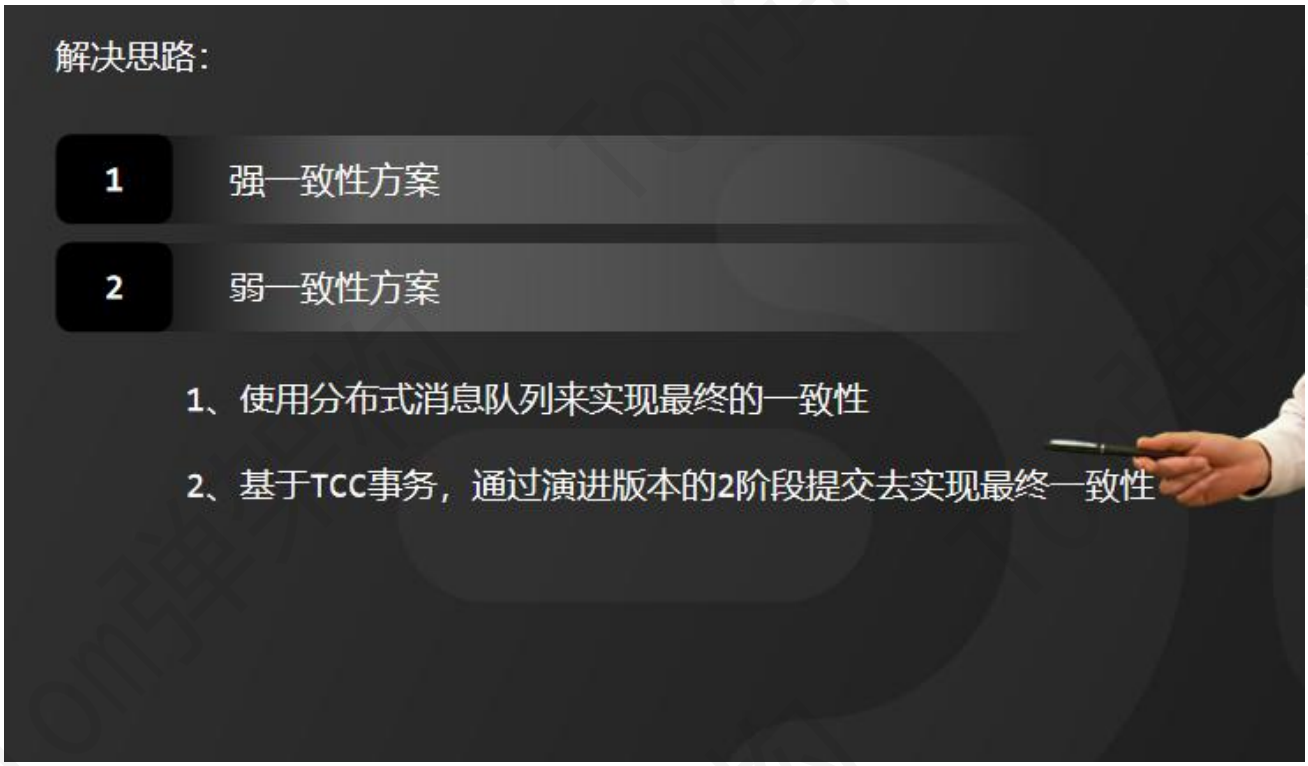
基于 CAP 定理和 Base 理论，我们可以知道，对于上述情况产生的分布式问题，我们要么采用强一致性方案，要么采用弱一致性方案。

所谓强一致性方案，是指通过第三方事务管理器来协调多个节点的事务性，保证每一个节点的事务达到同时成功或者同时失败，为了实现这样一个需求。我们可以引入 X/Open DTP 模型提供的 XA 协议，基于 2 阶段提交或者 3 阶段提交的方式去实现，但是如果全局事务管理器中的多个节点，任意一个节点在进行事务提交确认的时候，由于网络通信延迟导致了阻塞，就会影响到所有节点的事务提交，而这个阻塞过程呢，也会影响到用户的请求线程，这对于用户体验以及整体的性能影响非常大。

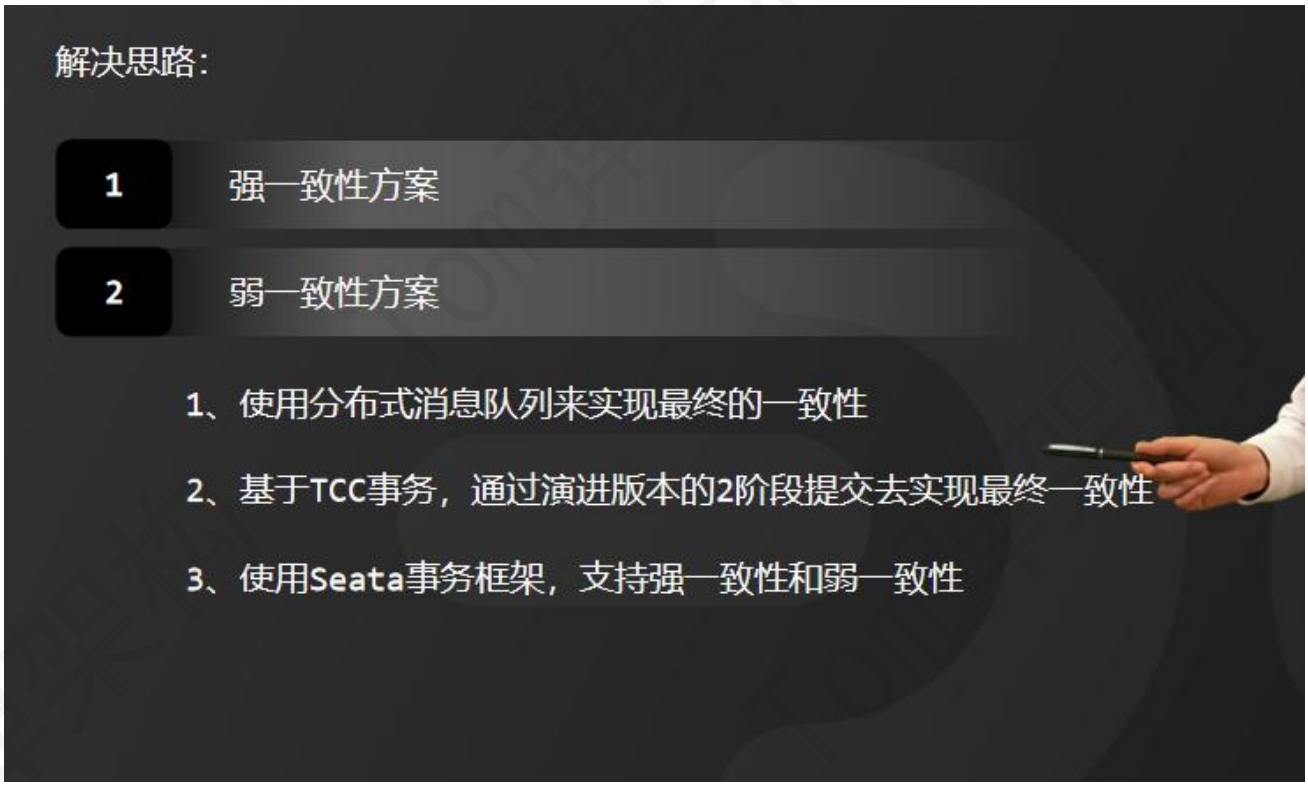
而弱一致性方案就是针对强一致性方案所衍生出来的性能和数据一致性平衡的一个方案。简单来说就是损失掉强一致性，数据在某一个时刻会存在不一致的状态，但是最终这些数据会达成一致，这样的好处是提升了系统的性能。在弱一致性方案中，常见的解决方案有 3 种：



第 1 个：使用分布式消息队列来实现最终的一致性。



第 2 个：基于 TCC 事务，通过演进版本的 2 阶段提交去实现最终一致性。



第 3 个：使用 Seata 事务框架，它提供了多种事务模型。比如说 AT、XA 、Saga、TCC 等，不同的模型提供的是强一致性或者弱一致性的支持。

以上就是我对分布式事务的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

39.synchronized 和 Lock 的区别

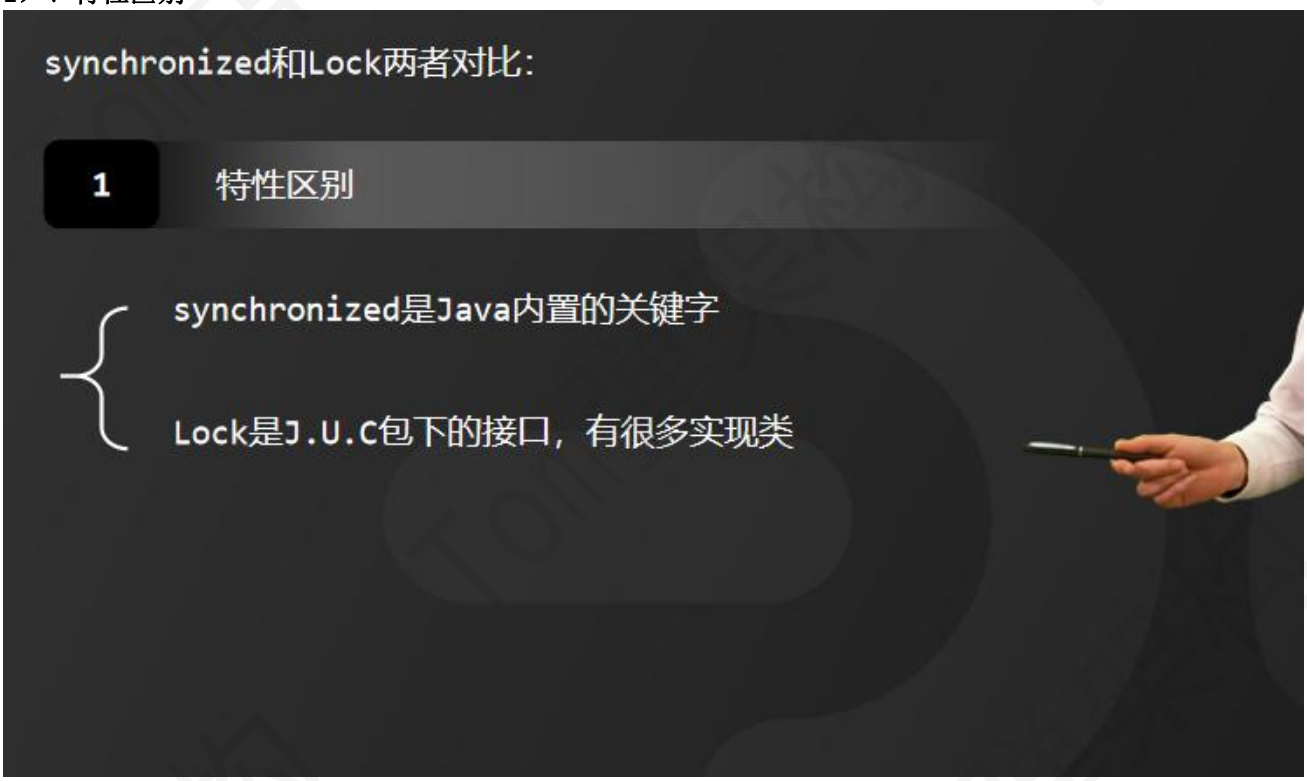
最近有多位粉丝被问到 synchronized 和 Lock，据说还是阿里一面的面试题。在分布式开发中，锁是控制线程的重要方式。Java 提供了两种锁机制 synchronized 和 Lock。接下来，我给大家分享一下我对 synchronized 和 Lock 的理解。

另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，想获取的小伙伴可以在我的煮叶简介中找到。

1、两者对比

synchronized 和 Lock 都是 Java 中用来解决线程安全问题的一个工具，那么关于 synchronized 和 Lock 的区别，我从以下 4 个方面来给大家来做一个详细的分析：

1）、特性区别



synchronized 是 Java 内置的一个线程同步关键字，

而 Lock 是 J.U.C 包下面的一个接口，它有很多实现类，比如 ReentrantLock 就是它的一个实现类。

2）、用法区别

synchronized 可以写在需要同步的对象、方法或者是特定代的码块中。主要有两种写法，比如这样：

一种是把 synchronized 修饰在方法上

```
//控制方法
public synchronized void sync(){
}
```

一种是把 synchronized 修饰在代码块上

```
Object lock = new Object();
//控制代码块
```

```
public void sync(){
synchronized(lock){

}
}
```

用这种方式来控制锁的生命周期。而 Lock 控制锁的粒度是通过 lock() 和 unlock() 方法来实现的，以 ReentrantLock 为例，来看这样一段代码：

```
Lock lock = new ReentrantLock();
public void sync(){
lock.lock();     //添加锁
//TODO 线程安全的代码
lock.unlock(); //释放锁
}
```

这种方式，是可以保证 lock()方法和 unlock()方法之间的代码是线程安全的。而锁的作用域，取决于 Lock 实例的生命周期。

Lock 比 synchronized 在使用上相对来说要更加灵活一些。Lock 可以自主地去决定什么时候加锁，什么时候释放锁。只需要调用 lock()和 unlock()这两个方法就可以了。需要注意的是，为了避免死锁，一般我们 unlock()方法写在 finally 块中。

另外，Lock 还提供了非阻塞的竞争锁的方法叫 trylock()，这个方法可以通过返回 true 或者 fasle 来告诉当前线程是否已经有其他线程正在使用锁。

而 synchronized 是关键字，无法去扩展实现非阻塞竞争锁的方法。另外，synchronized 只有代码块执行结束或者代码出现异常的时候才会释放锁，因此，它对锁的释放是被动的。

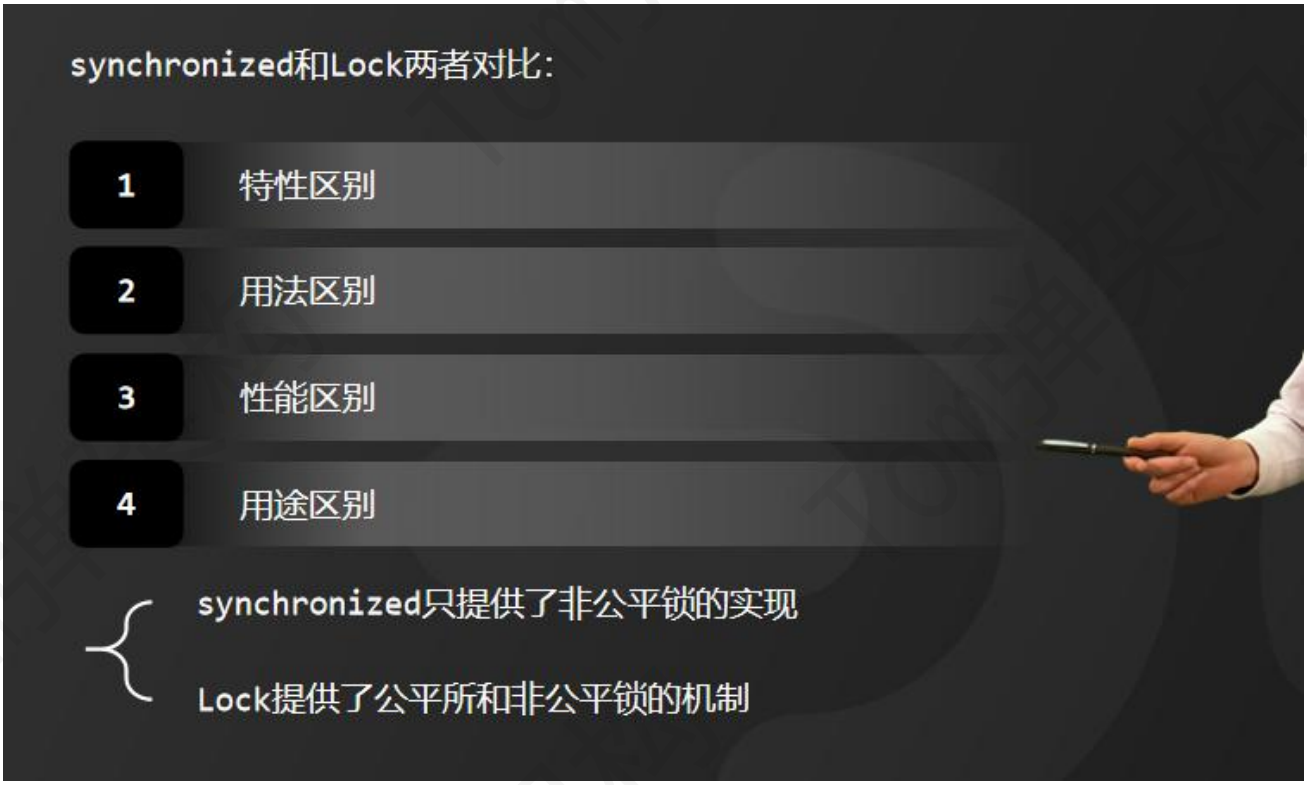
3）、性能区别

synchronized 和 Lock 在性能上差别不大。在实现上有一些区别，

synchronized 采用的是悲观锁机制，synchronized 是托管给 JVM 执行的。在 JDK1.6 以后采用了偏向锁、轻量级锁、重量级锁及锁升级的方式进行优化。

而 Lock 用的是乐观锁机制。控制锁的代码由用于自定义，也采用 CAS 自旋锁进行了优化。

4）、用途区别



二者在一般情况下没有什么区别，但是在非常复杂的同步应用中，建议使用 Lock。

因为 synchronized 只提供了非公平锁的实现，而 Lock 提供了公平所和非公平锁的机制。

公平锁是指线程竞争锁资源的时候，如果已经有其他线程正在排队或者等待锁释放，那么当前竞争锁的线程是无法去插队的。

而非公平锁就是不管是否线程再排队等待锁，它都会去尝试竞争一次锁。

以上，就是我对 synchronized 和 Lock 的理解，我还专门整理了一张表格帮助大家更好地理解，有需要这张表的小伙伴可以在我的主页简介中获取。

项	synchronized	Lock
特性	Java的关键字，在JVM层面	J.U.C包中的接口
获取	A获得锁，B等待。A阻塞，B一直等待	可尝试获得锁，线程可以不用一直等待
释放	执行完同步代码或者发生异常，被动释放	在finally中释放锁，避免死锁
状态	无法判断	可以判断
类型	可以重入，不可中断，非公平	可重入，可判断，可公平，可非公平
性能	少量同步	大量同步

我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

40.Dubbo 和 SpringCloud 的优缺点对比

其实我个人不太愿意，拿 Dubbo 和 Spring Cloud 进行对比，因为它俩最初出现并不是为了解决同一类问题。但是，国内技术是在太卷，加上微服务的盛行，很多互联网大厂也经常会问到这个问题。那么今天，我还是给大家来详细聊一聊。

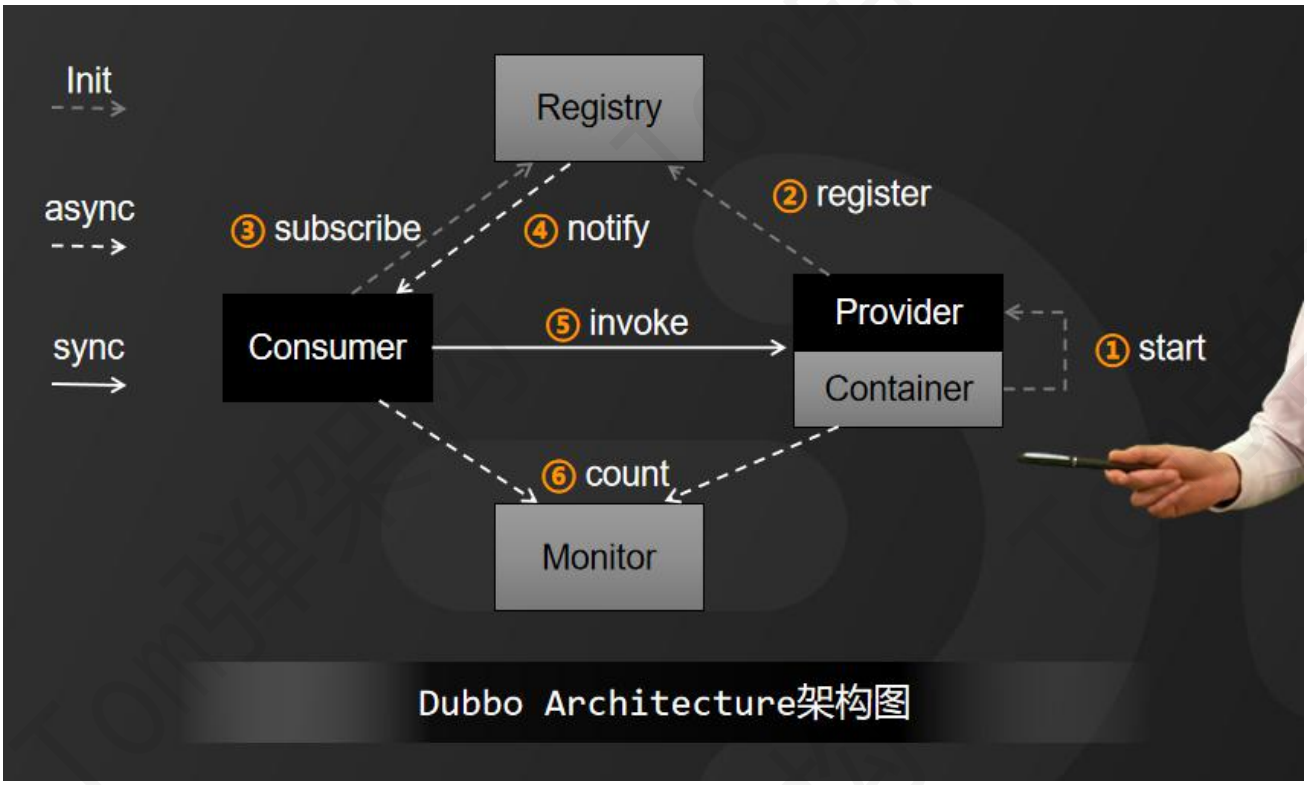
另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，想获取的小伙伴可以在我的煮叶简介中找到。

1、两者对比

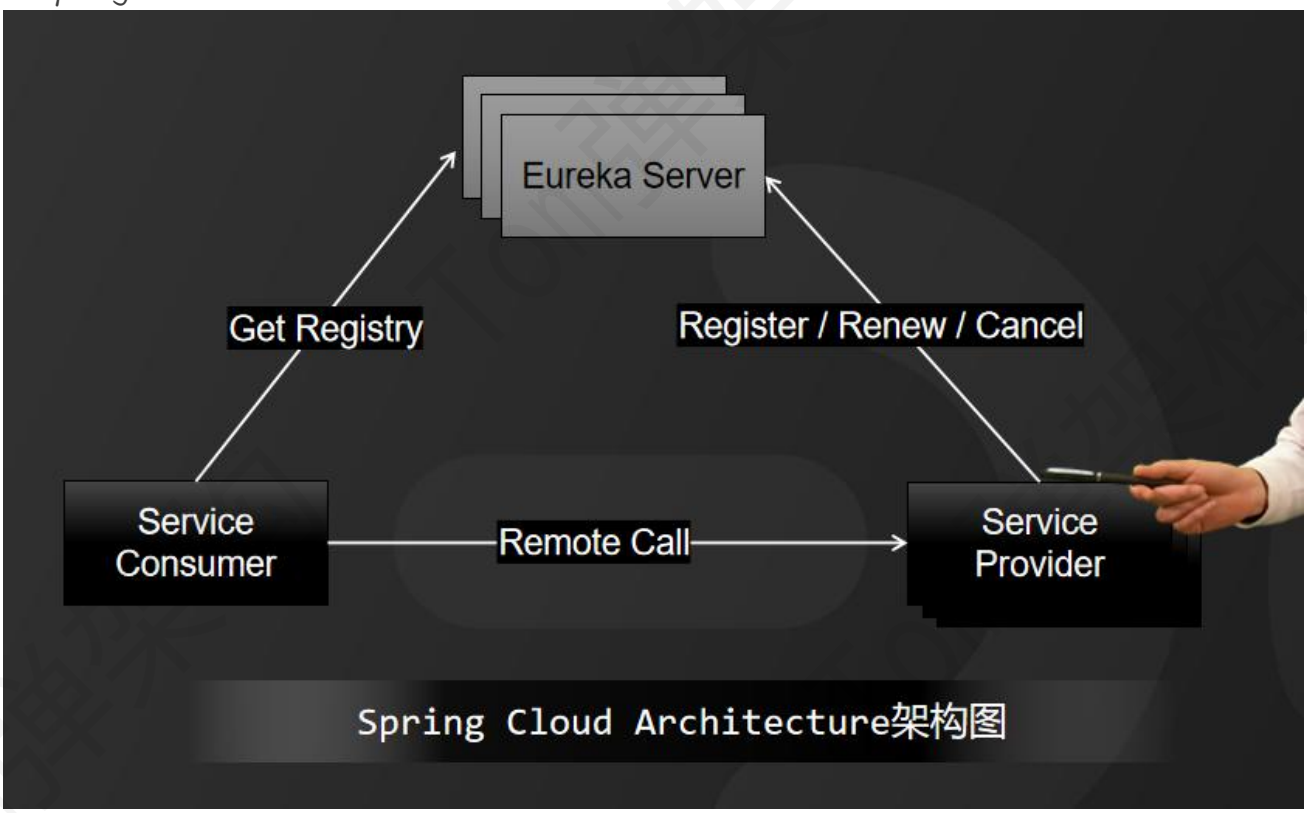
关于 Dubbo 和 Spring Cloud 的优缺点，我以奈菲（Netflix）版本为例，从以下 5 个方面来分析：

1）、从整体架构上来看

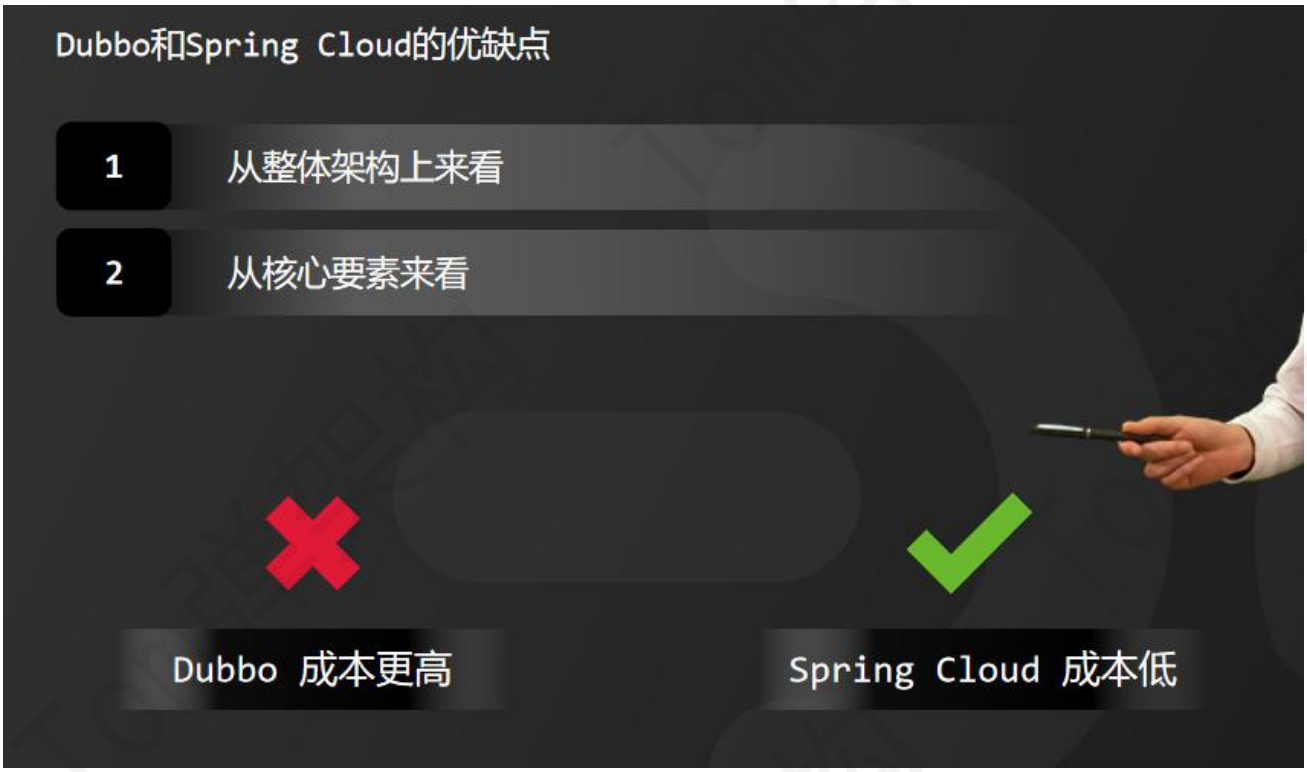
Dubbo 和 SpringCloud 的模式都比较接近，都需要服务提供方，注册中心，服务消费方。差异并不大。Dubbo 的架构图是这样的，



而 Spring Cloud 的架构图是这样的

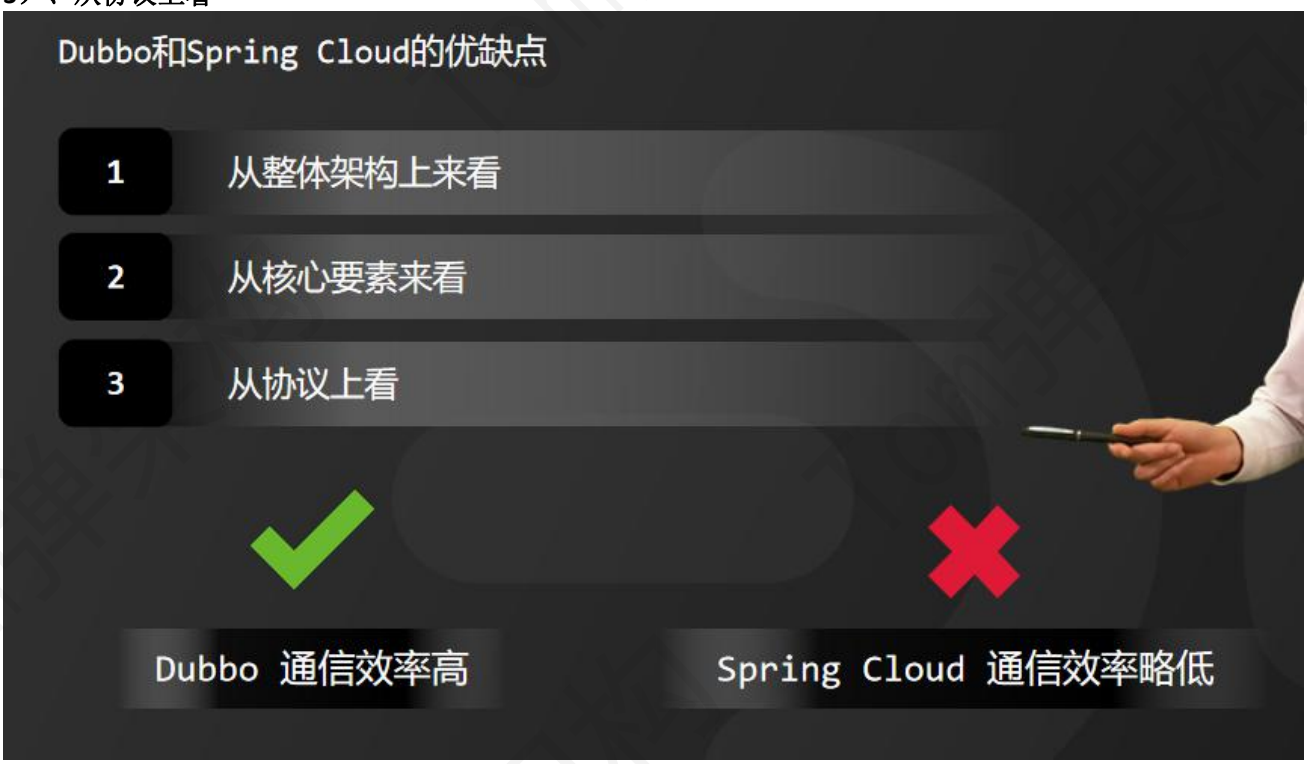


2）、从核心要素来看



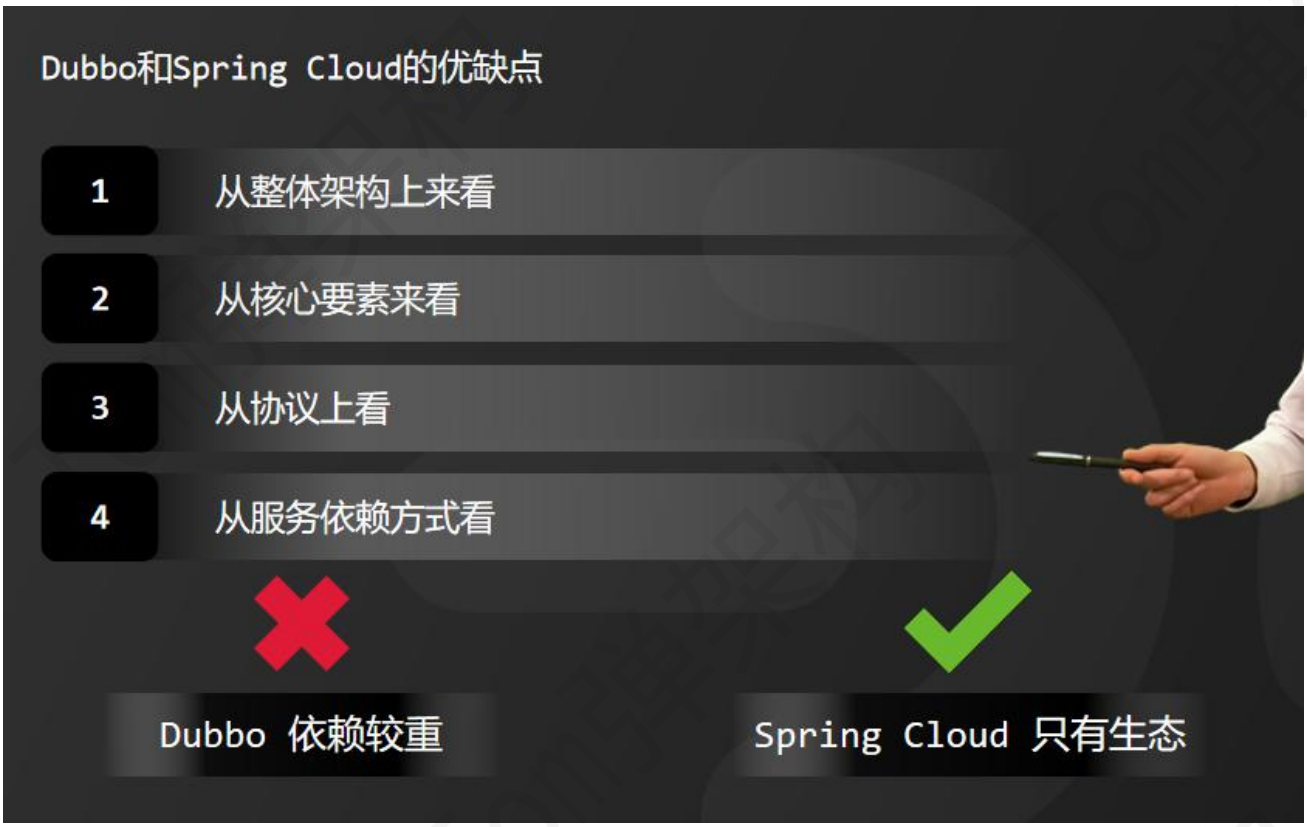
Spring Cloud 更胜一筹，在开发过程中只要整合 Spring Cloud 的子项目就可以顺利的完成各种组件的融合，而 Dubbo 需要通过实现各种 Filter 来做定制，开发成本以及技术难度略高。

3）、从协议上看



Dubbo 默认采用的是单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。Dubbo 还支持其他各种通信协议，而 Spring Cloud 使用 HTTP 协议的 REST API。因此，在通信速度上 Dubbo 略胜。

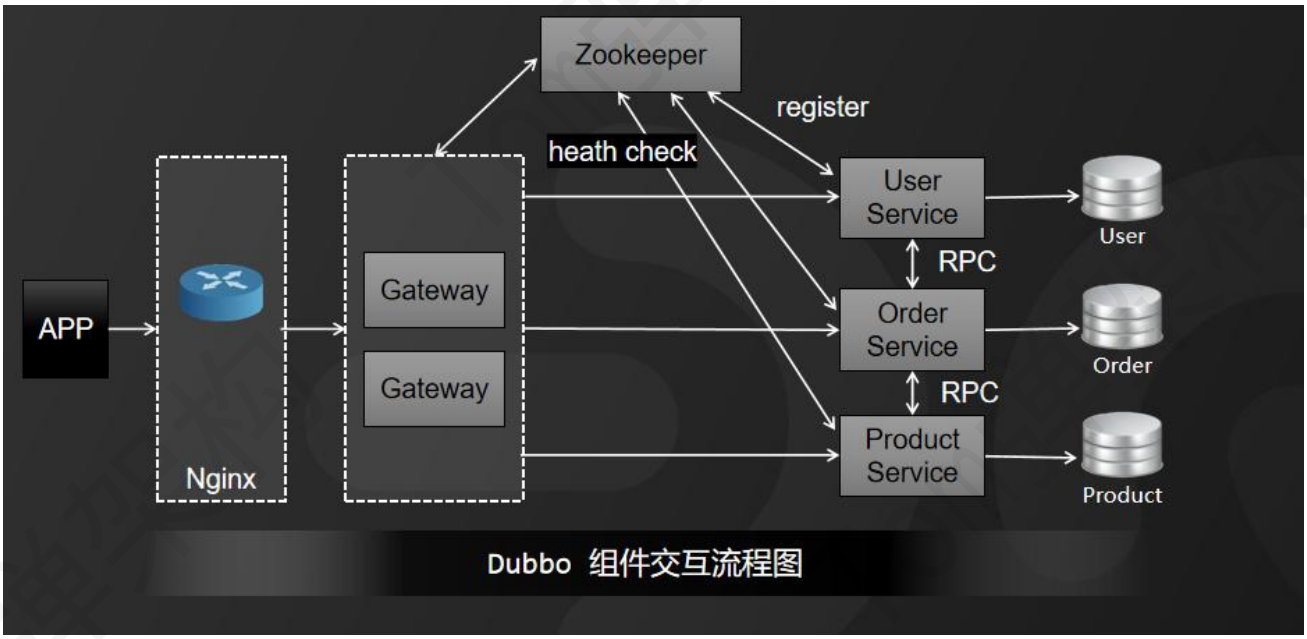
4）、从服务依赖方式看



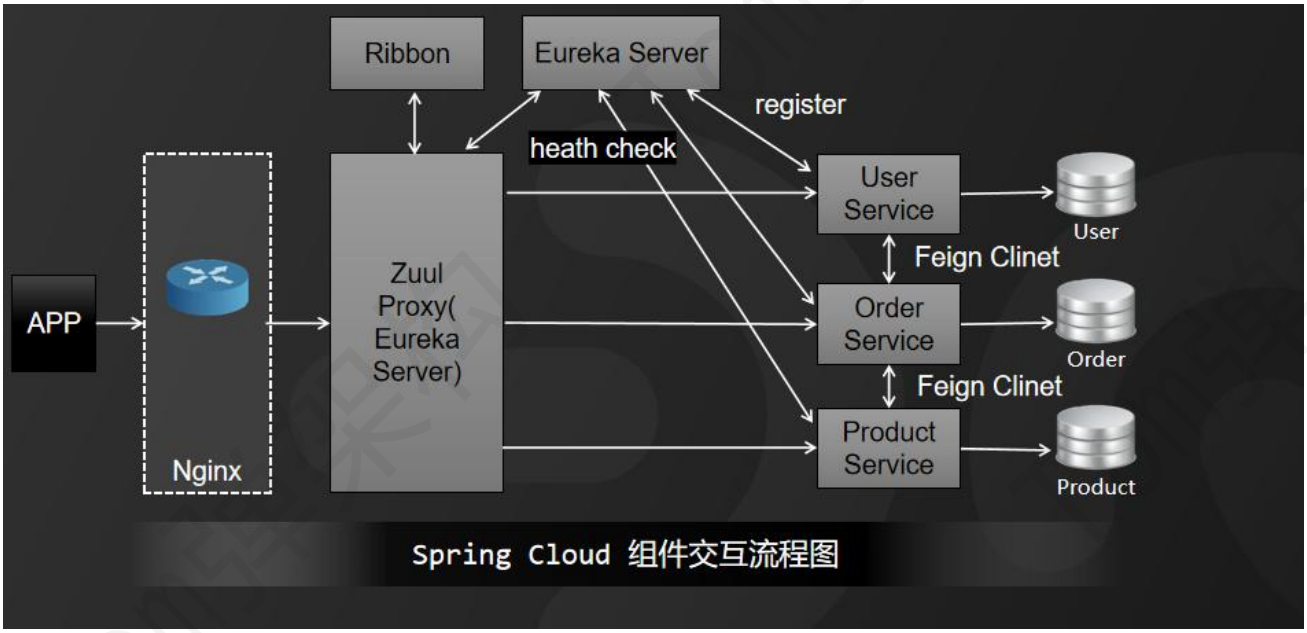
Dubbo 服务依赖比较重，需要有完善的版本管理机制，但是程序入侵少。而 Spring Cloud 是自有生态，省略了版本管理的问题，它使用 JSON 进行交互，为跨平台调用奠定了基础。

5）、从组件运行流程看

Dubbo 的每个组件都是需要部署在单独的服务器上， 用来接收前端请求、聚合服务，并批量调用后台原子服务。每个 Service 层和单独的 DB 交互。



而 Spring Cloud 所有请求都统一通过 API 网关（Zuul）来访问内部服务。网关接收到请求后，从注册中心（Eureka）获取可用服务。由 Ribbon 进行均衡负载后，分发到后端的具体实例。微服务之间通过 Feign 进行通信处理业务。



但是，两者的业务部署方式相同，都需要前置一个网关来隔绝外部直接调用原子服务的风险。*Dubbo* 需要自己开发一套 *API* 网关，而 *Spring Cloud* 则可以通过 *Zuul* 配置就可以完成网关定制。所以，从使用方式上 *Spring Cloud* 更加方便。

以上就是我对 Dubbo 和 Spring Cloud 的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

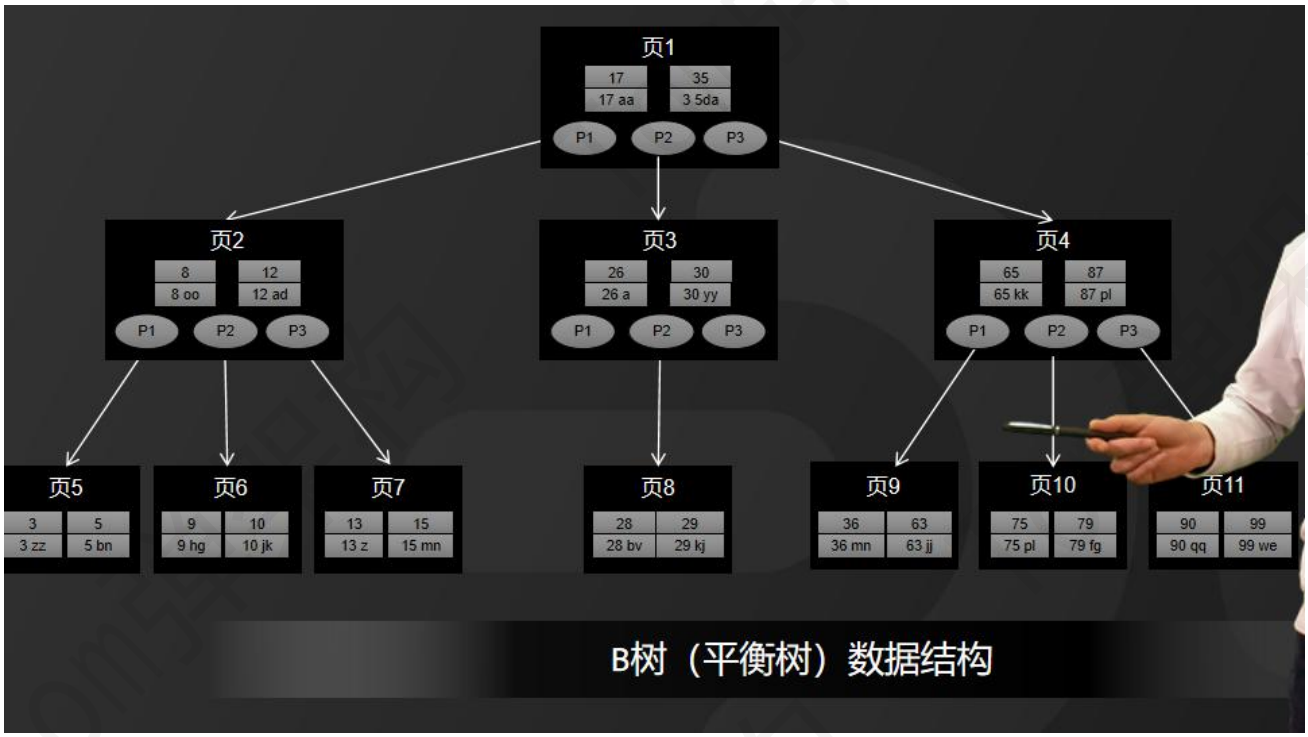
41.为什么 MySQL 索引结构采用 B+树？

一位 *6* 年经验的小伙伴去字节面试的时候被问到这样一个问题，为什么 *MySQL* 索引结构要采用 *B+*树？这位小伙伴从来就没有思考过这个问题。只因为现在都这么卷，后面还特意查了很多资料，他也希望听听我的见解。

另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，一共有 10 万字，想获取的小伙伴可以在我的煮叶简介中找到。

1、B 树和 B+树

一般来说，数据库的存储引擎都是采用 *B* 树或者 *B+*树来实现索引的存储。首先来看 *B* 树，如图所示。



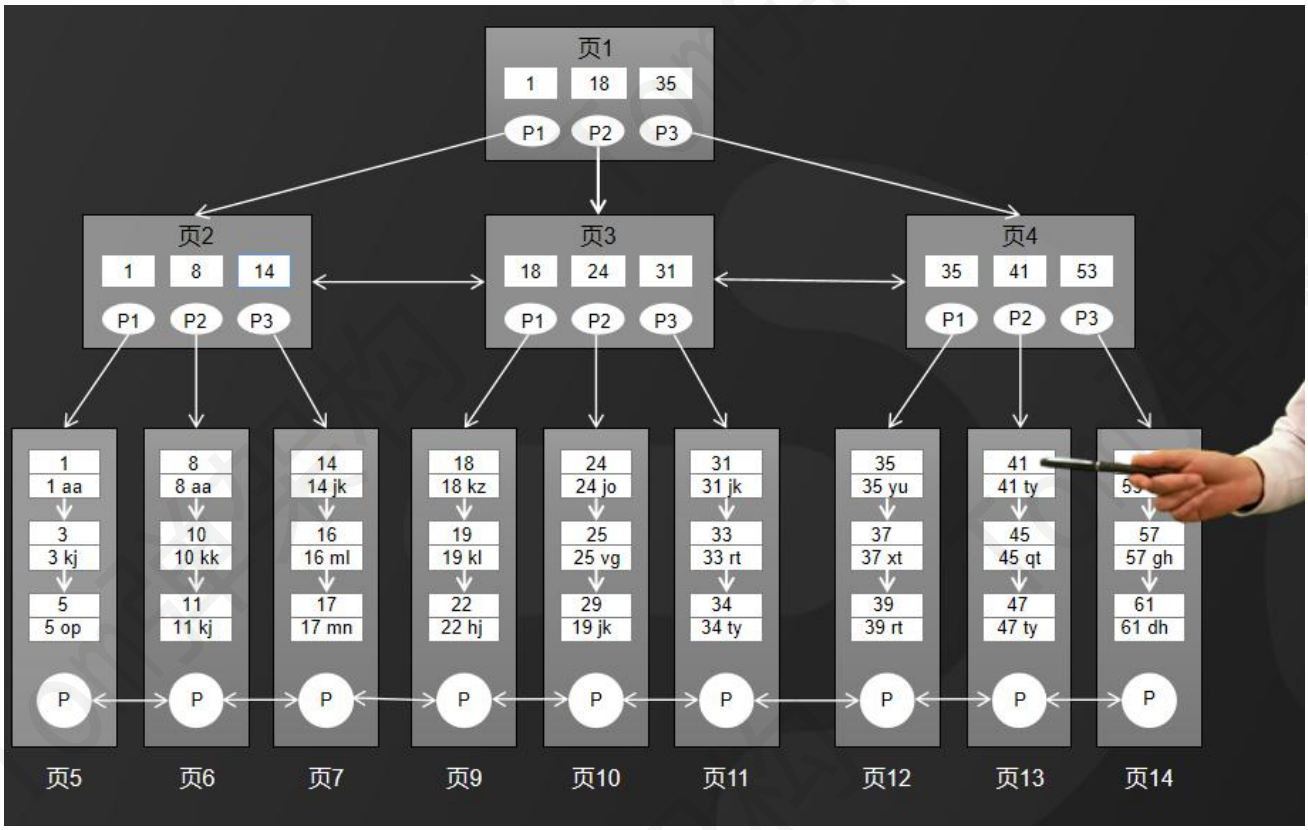
B 树是一种多路平衡树，用这种存储结构来存储大量数据，它的整个高度会相比二叉树来说，会矮很多。

而对于数据库而言，所有的数据都将会保存到磁盘上，而磁盘 I/O 的效率又比较低，特别是在随机磁盘 I/O 的情况下效率更低。

所以 高度决定了磁盘 I/O 的次数，磁盘 I/O 次数越少，对于性能的提升就越大，这也是为什么采用 B 树作为索引存储结构的原因，如图所示。

而 MySQL 的 InnoDB 存储引擎，它用了一种增强的 B 树结构，也就是 B+树来作为索引和数据的存储结构。

相比较于 B 树结构来说，B+树做了两个方面的优化，如图所示。



1、B+树的所有数据都存储在叶子节点，非叶子节点只存储索引。

2、叶子节点中的数据使用双向链表的方式进行关联。

2、原因分析

我认为，MySQL 索引结构采用 B+树，有以下 4 个原因：

1、从磁盘 I/O 效率方面来看：B+树的非叶子节点不存储数据，所以树的每一层就能够存储更多的索引数量，也就是说，B+树在层高相同的情况下，比 B 树的存储数据量更多，间接会减少磁盘 I/O 的次数。

2、从范围查询效率方面来看：在 MySQL 中，范围查询是一个比较常用的操作，而 B+树的所有存储在叶子节点的数据使用了双向链表来关联，所以 B+树在查询的时候只需查两个节点进行遍历就行，而 B 树需要获取所有节点，因此，B+树在范围查询上效率更高。

3、从全表扫描方面来看：因为，B+树的叶子节点存储所有数据，所以 B+树的全局扫描能力更强一些，因为它只需要扫描叶子节点。而 B 树需要遍历整个树。

4、从自增 ID 方面来看：基于 B+树的这样一种数据结构，如果采用自增的整型数据作为主键，还能更好的避免增加数据的时候，带来叶子节点分裂导致的大量运算的问题。

3、总结

总体来说，我认为技术方案的选型，更多的要根据具体的业务场景来决定，并不一定是说 B+树就是最好的选择，就像 MongoDB 里面采用 B 树结构，本质上来说，其实是关系型数据库和非关系型数据库的差异。

以上就是我对为什么 MySQL 索引结构采用 B+树 的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

42.谈谈你对 MySQL 事务隔离级别的理解

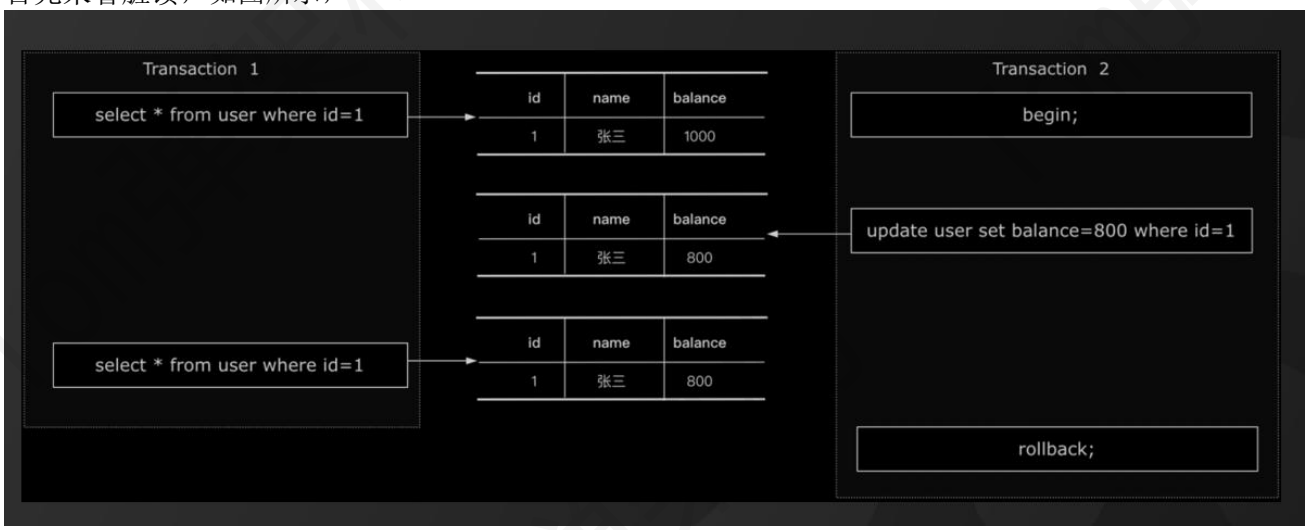
一位 5 年工作经验的粉丝，去阿里面试被问到一个关于数据库事务隔离级别的问题，当时，没有问答上来，希望给他一个参考答案。那么，今天我给大家谈谈我的理解。

另外，我花了 1 个多星期把往期的面试题解析配套文档准备好了，一共有 10W 字，想获取的小伙伴可以从我的个人煮叶简介中找到。

1、脏读、幻读、不可重复读

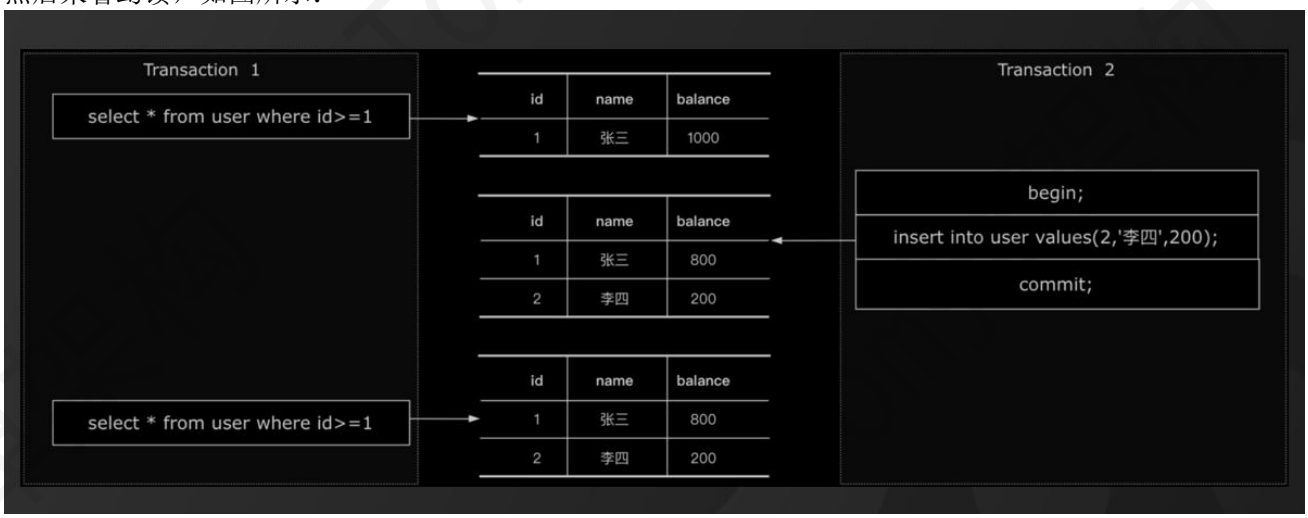
在 SQL 操作中，多个事务竞争可能会产生三种不同的现象，分别是脏读、幻读、不可重复读。

首先来看脏读，如图所示，



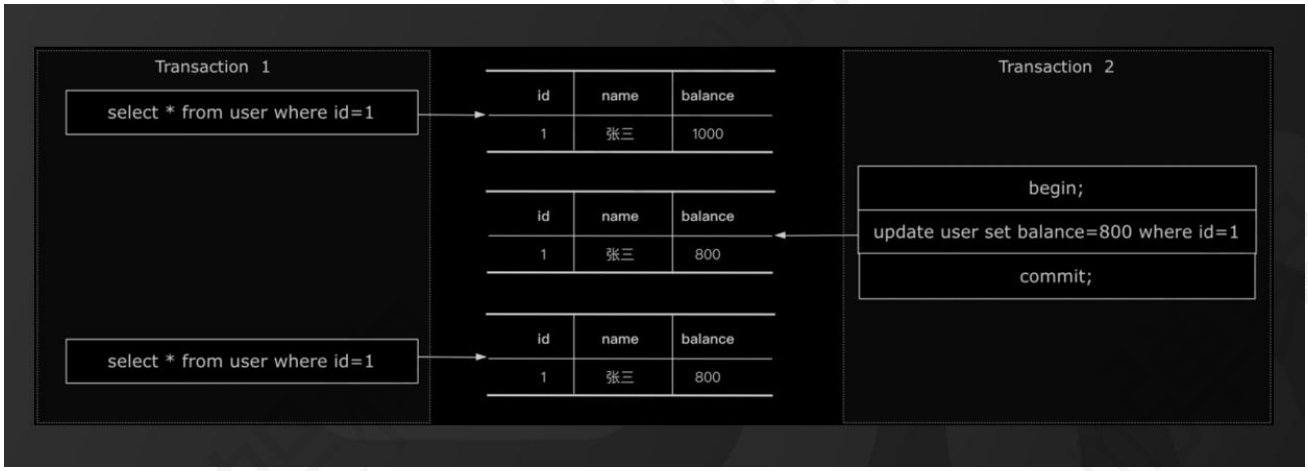
假设有两个事务 T1/T2 同时在执行，T1 事务有可能会读取到 T2 事务未提交的数据，但是未提交的事务 T2 可能会回滚，也就导致了 T1 事务读取到最终不一定存在的数据产生脏读的现象。

然后来看幻读，如图所示：



假设有两个事务 T1/T2 同时执行，事务 T1 执行范围查询或者范围修改的过程中，事务 T2 插入了一条属于事务 T1 范围内的数据并且提交了，这时候在事务 T1 查询发现多出来了一条数据，或者在 T1 事务发现这条数据没有被修改，看起来像是产生了幻觉，这种现象称为幻读。

最后来看，不可重复读，如图所示：



假设有两个事务 T1/T2 同时执行，事务 T1 在不同的时刻读取同一行数据的时候结果可能不一样，从而导致不可重复读的问题。

2、事务隔离级别

那么事务隔离级别，就是是为了解决多个并行事务竞争， 。而这脏读、幻读、不可重复读这三种现象在实际应用中，有些业务场景是不能接受这些现象存在的，所以在 SQL 标准中定义了四种隔离级别，分别是：

读未提交，在这种隔离级别下，可能会产生脏读、不可重复读、幻读。

读已提交（RC)，在这种隔离级别下，可能会产生不可重复读和幻读。

可重复读（RR），在这种隔离级别下，可能会产生幻读

串行化，在这种隔离级别下，多个并行事务串行化执行，不会产生安全性问题。

这四种隔离级别里面，只有串行化解决了全部的问题，但这种隔离级别的性能是最低的。

在 MySQL 里面，InnoDB 引擎默认的隔离级别是 RR（可重复读），因为它需要保证事务 ACID 特性中的隔离性特征。

以上就是我对 MySQL 事务隔离级别的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

43、什么是双亲委派机制？

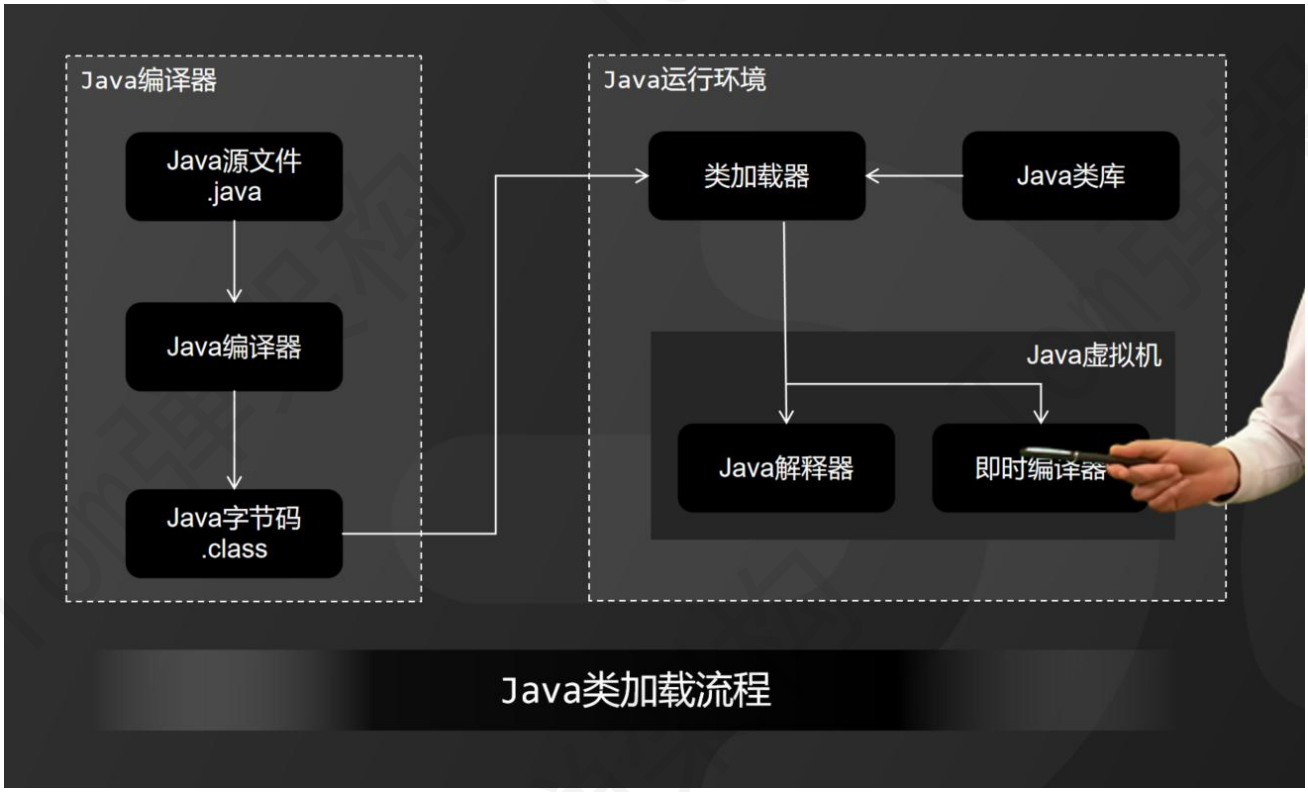
今天给大家分享一道国内的一二线互联网公司，高频次出现的面试题。比如，什么是类加载？什么是双亲委派？ 等等。首先，我们来看双亲委派，它全称是 Parent Delegation Model，直译过来可能叫做父级委托模型更容易理解。不管它叫什么，如果是你被问到这样的问题，你会不会冷场呢？

那么，今天我给大家分享一下我的理解。

我花了 1 个多星期把往期的面试题解析配套文档准备好了，一共有 10W 字，想获取的小伙伴可以从我的个人煮叶简介中找到。

1、类加载机制

要理解双亲委派，首先要理解 Java 的类加载机制。接下来，我简单介绍一下 Java 的类加载机制，如图所示：



Java 编译器将 Java 源文件编译成.class 文件，再由 JVM 加载.class 文件到内存中，JVM 装载完成后得到一个 Class 字节码对象。拿到字节码对象之后，我们就可以实例化了。

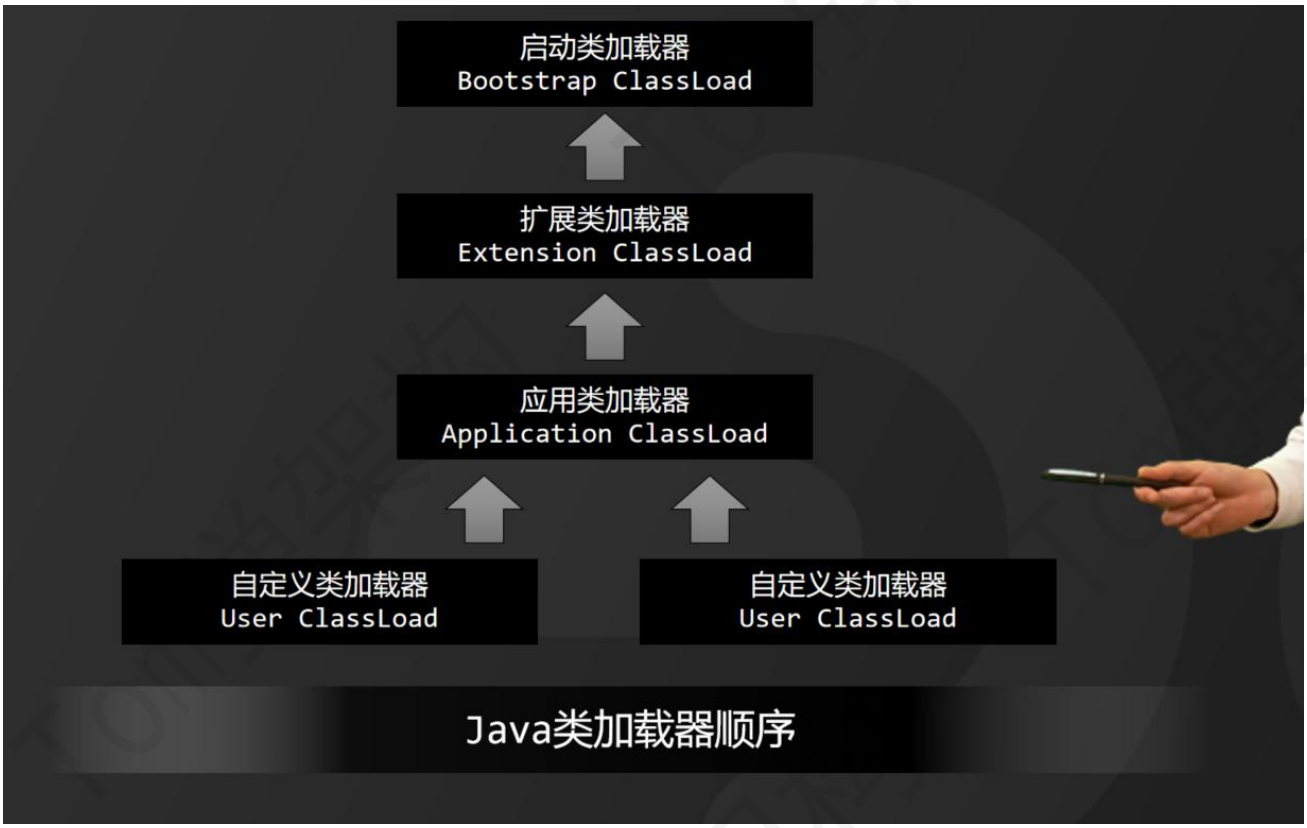
那么，类的加载过程需要使用到加载器。JVM 设计了 3 个类加载器，它们分别是：Bootstrap 类加载器、Extension 类加载器和 类加载器，这些类加载器分别加载不同作用范围的 jar 包和.class 文件。下面，我给大家详细介绍一下：

1、Bootstrap ClassLoader，主要是负责 Java 核心类库的加载，也就是 %{JDK_HOME}\lib 下的 rt.jar、resources.jar 等

2、Extension ClassLoader，主要负责%{JDK_HOME}\lib\ext 目录下的 jar 包和 class 文件

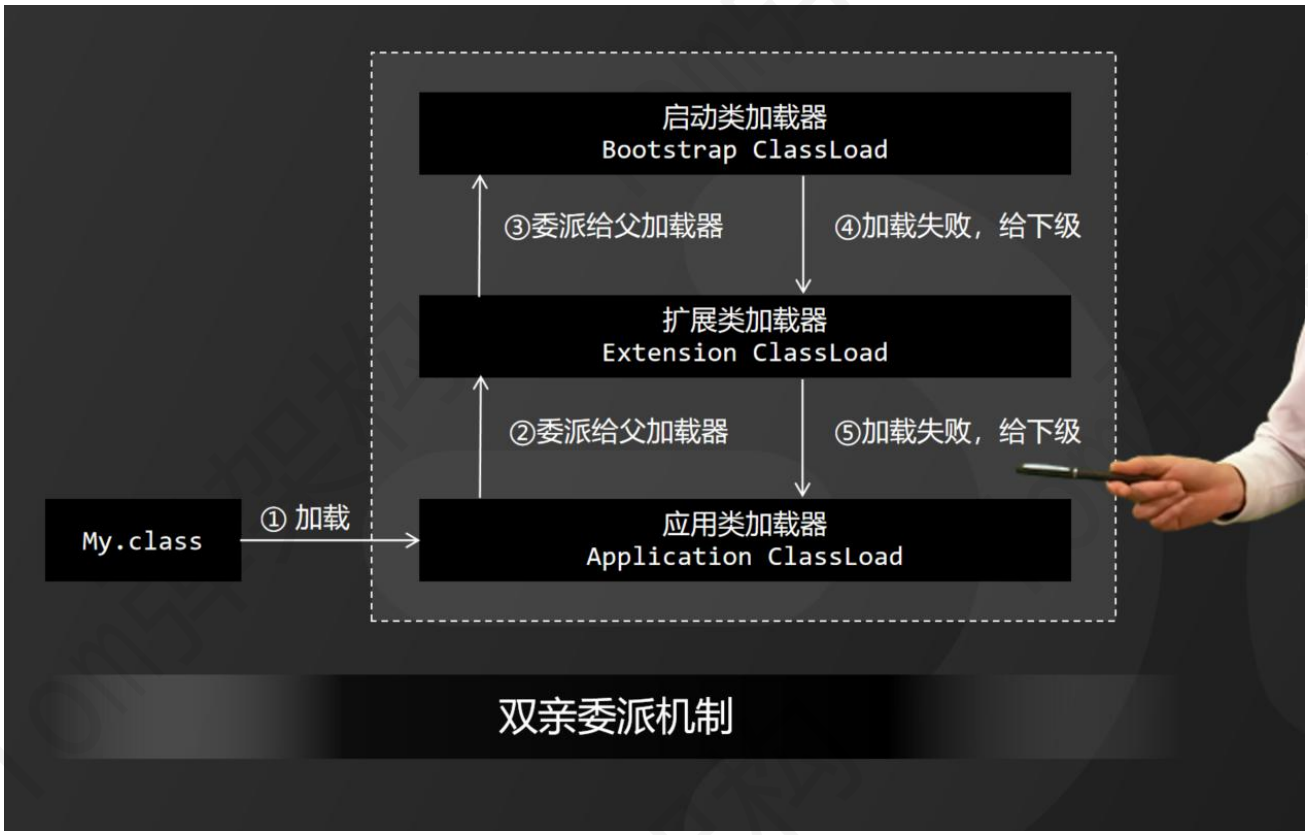
3、Application ClassLoader，主要负责当前应用里面的 classpath 下的所有 jar 包和类文件

除了系统自己提供的类加载器以外，还可以通过 ClassLoader 类实现自定义加载器，去满足一些特殊场景的需求。



2、双亲委派机制

所谓双亲委派机制，或者叫父级委托模型，就是指按照类加载器的层级关系，逐层进行委派。如图所示：



比如，我们需要加载一个 class 文件的时候，首先会把这个 class 的查询和加载委派给父加载器去执行，如果父加载器都无法加载，再尝试自己来加载这个 class。

3、总结

我认为，双亲委派机制的设计有 2 个好处：

1. 保证安全性，因为这种层级关系实际上代表的是一种优先级，也就是所有的类的加载，优先给 Bootstrap ClassLoader。那对于核心类库中的类，就没办法去破坏，比如自己写一个 java.lang.String，最终还是会交给 Bootstrap 类加载器加载。再加上每个类加载器的都有不同的作用范围，这就意味着自己写的 java.lang.String 就没办法去覆盖核心类库中类。
2. 避免重复加载，我认为这种层级关系的设计，可以避免重复加载导致程序混乱的问题。如果父加载器已经加载过了，那么子加载器就没必要去加载了。

好了，以上就是我对双亲委派机制的理解。我是被编程耽误的文艺 Tom，如果我的分享对你　有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

44、Netty 是什么，为什么要使用 Netty？

最近，也不知道什么原因，经常有粉丝问我关于 Netty 的问题。难道是大厂面试更卷了，开始关注更加底层的框架了？先不深究什么原因了，今天，我给大家分享一下什么是 Netty，它能解决什么问题？

另外，我花了 1 个多星期，准备了一份 10W 字的面试题解析配套文档，想获取的小伙伴可以从我的个人煮叶简介中找到。

1、Netty 是什么

一句话总结，Netty 就是一个基于 Java NIO 封装的高性能的网络通信框架。我从以下三个方面给大家归纳一下：

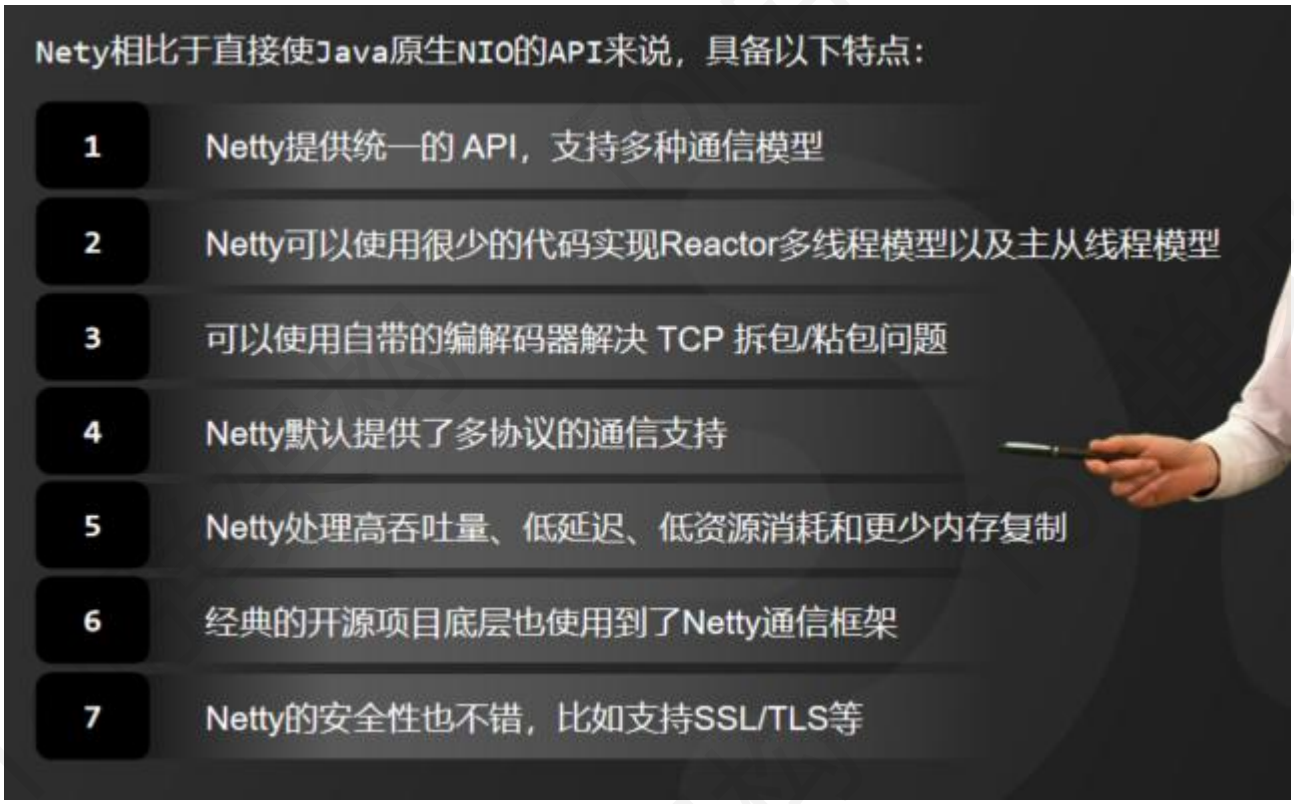
1、Netty 提供了比 NIO 更简单易用的 API，我们可以利用这些封装好的 API 快速开发自己的网络通信程序。

2、Netty 在 NIO 的基础上还做了很多优化，比如零拷贝机制、、内存池管理等等，因此，总体运行性能会比原生的 NIO 更高。

3、Netty 内置支持了多种通信协议，如 HTTP、WebSocket 等，并且针对数据通信的拆包、黏包问题，Netty 还内置了解决方案。

2、为什么要用 Netty？

Nety 相比于直接使 Java 原生 NIO 的 API 来说，选择 Netty 具备以下优势：



1、Netty 提供统一的 API，支持多种通信模型，如阻塞、非阻塞， 以及 epoll、poll 等模型。
2、Netty 可以使用很少的代码实现 Reactor 多线程模型以及主从线程模型。
3、可以使用自带的编解码器解决 TCP 拆包/粘包问题。
4、Netty 默认提供了多协议的通信支持。
5、Netty 处理高吞吐量、低延迟、低资源消耗，比 Java 原生 NIO 的 API 更有优势。
6、经典的开源项目底层也使用到了 Netty 通信框架， 比如 Zookeeper、Dubbo、RocketMQ 等等，经历了大型项目的使用和考验更加成熟稳定。
7、Netty 对安全性支持也不错，比如支持 SSL/TLS 等。

好了，以上就是我对 Netty 的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

45、Netty 中提供了哪些线程模型？

最近，我更新了一些 Netty 相关的内容，于是有很多粉丝开始私信问我一些关于 Netty 的问题。今天，给大家分享一个大家问得比较多问题，Netty 中提供了哪些线程模型？

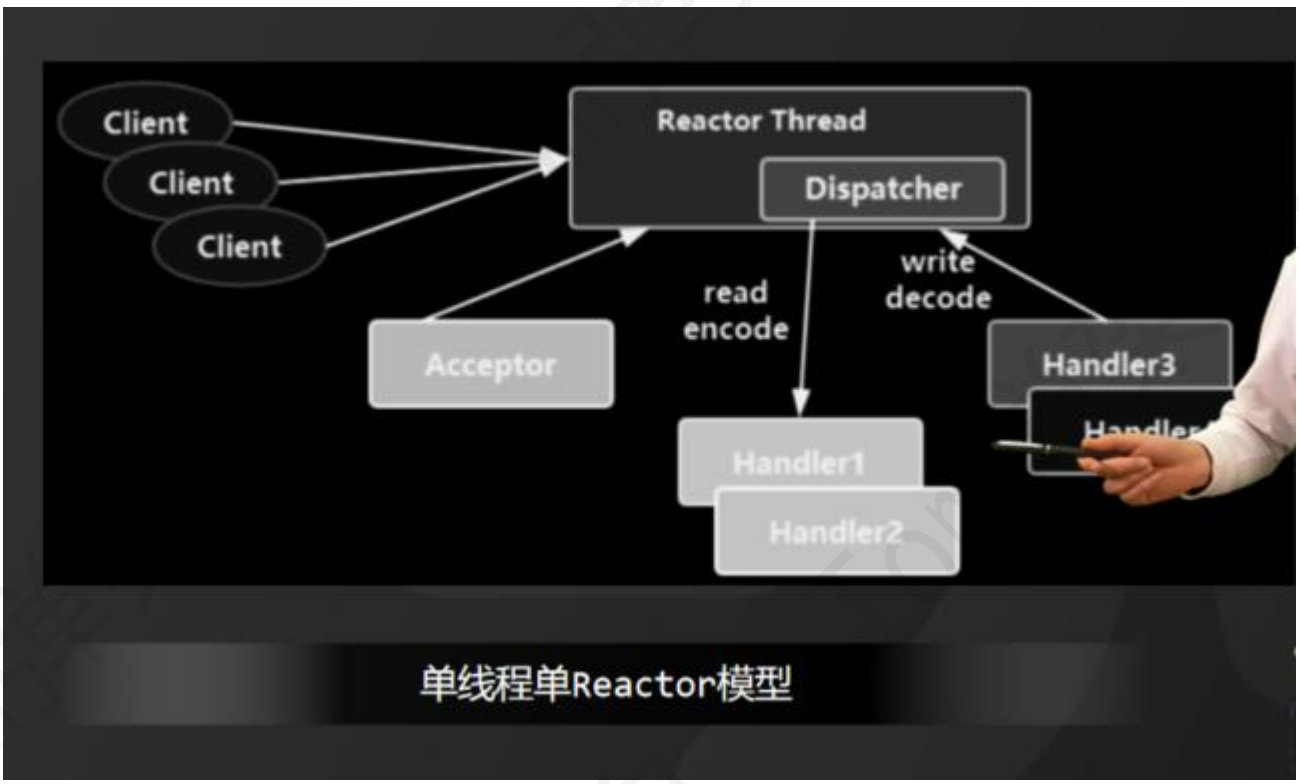
另外，我花了 1 个多星期，准备了一份 10W 字的面试题解析配套文档，想获取的小伙伴可以从我的个人煮叶简介中找到。

说到线程模型，又不得不说 Netty 中的 Reactor，Reactor 直译过来叫做反应堆，它是 Netty 支持异步多线程的核心组件。常见的 Reactor 线程模型有三种，分别是：Reactor 单线程模型、Reactor 多线程模型、主从 Reactor 多线程模型；

1、单线程单 Reactor 模型

在 Reactor 模型有三个重要的组件：

1、Reactor ： 主要负责将 I/O 事件发派给对应的 Handler
2、Acceptor ： 用于处理客户端连接请求
3、Handlers ： 执行非阻塞的 I/O 读写任务
首先来看单线程单 Reactor 模型，如图所示：

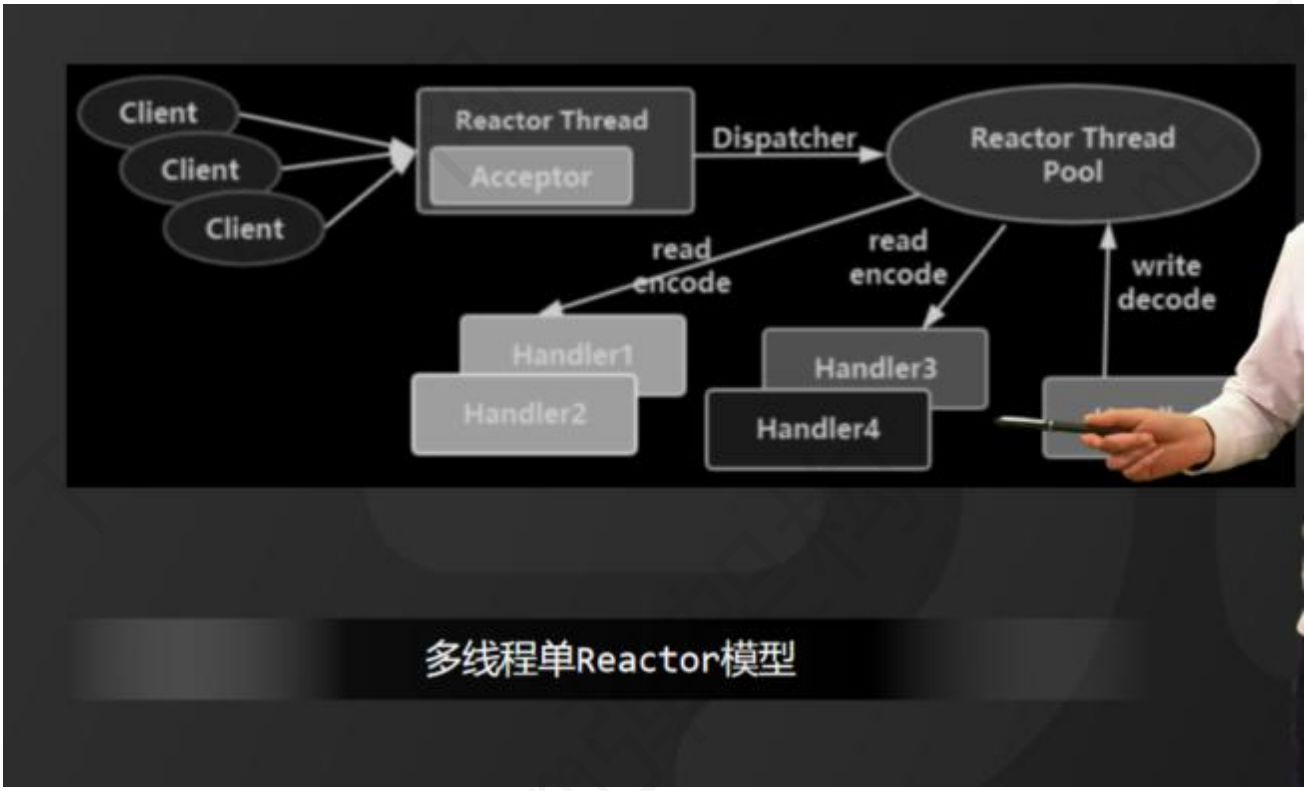


单线程 Reactor 这种实现方式存在缺点，因为，Handler 的执行是串行的，如果其中一个 Handler 处理线程阻塞，将导致其他的业务处理也会阻塞。而 Handler 和 Reactor 在同一个线程中的执行，这也将导致无

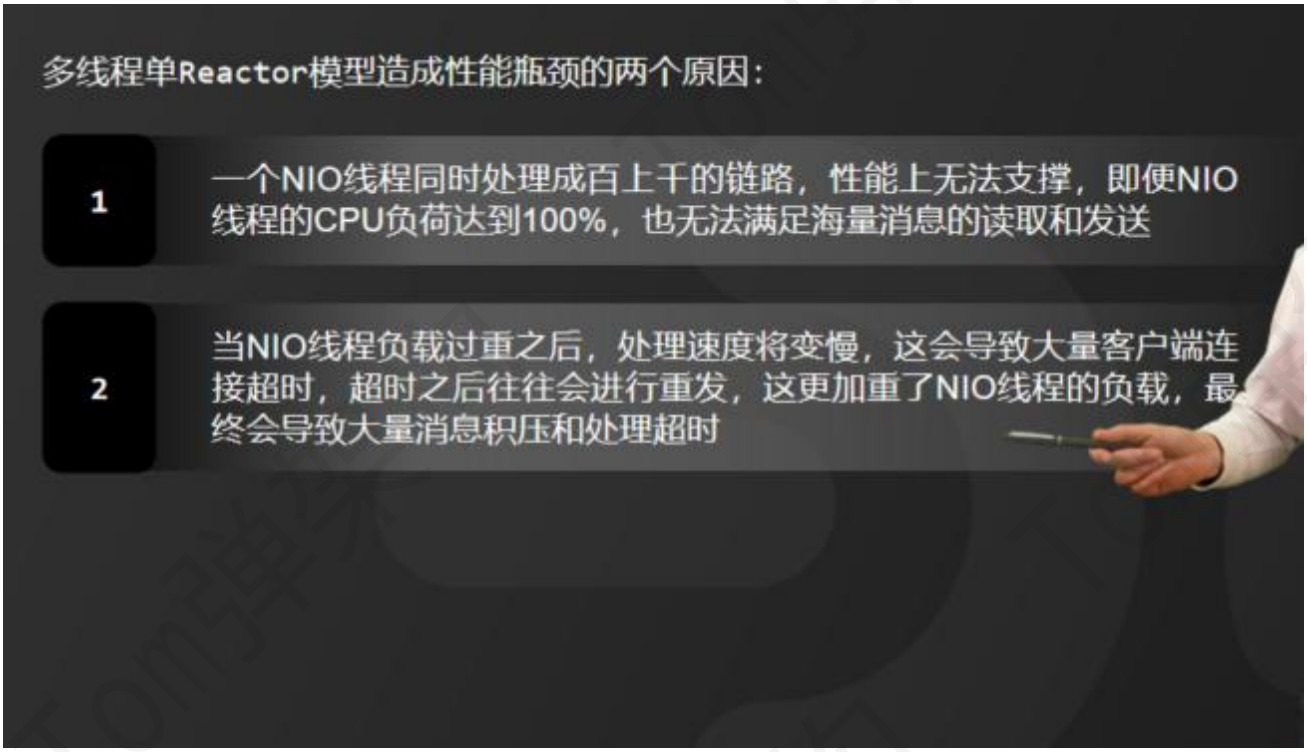
法接收新的请求。

2、多线程单 Reactor 模型

为了解决单线程 Reactor 的问题，有人提出使用多线程的方式来处理业务逻辑，也就是在业务处理的地方加入线程池，实现异步处理，这样将 Reactor 和 Handler 就放在不同的线程中来执行，这就是多线程单 Reactor 模型。



但是，问题又来了，在多线程单 Reactor 模型中，所有的 I/O 操作是由一个 Reactor 来完成，而 Reactor 运行在单个线程中，它需要处理包括 accept()/read()/write()/connect()等操作，在并发量小的情况下影响不大。一旦并发量上来，出现高负载、高并发或大数据量的应用场景时，容易成为瓶颈，主要有以下 2 个原因：

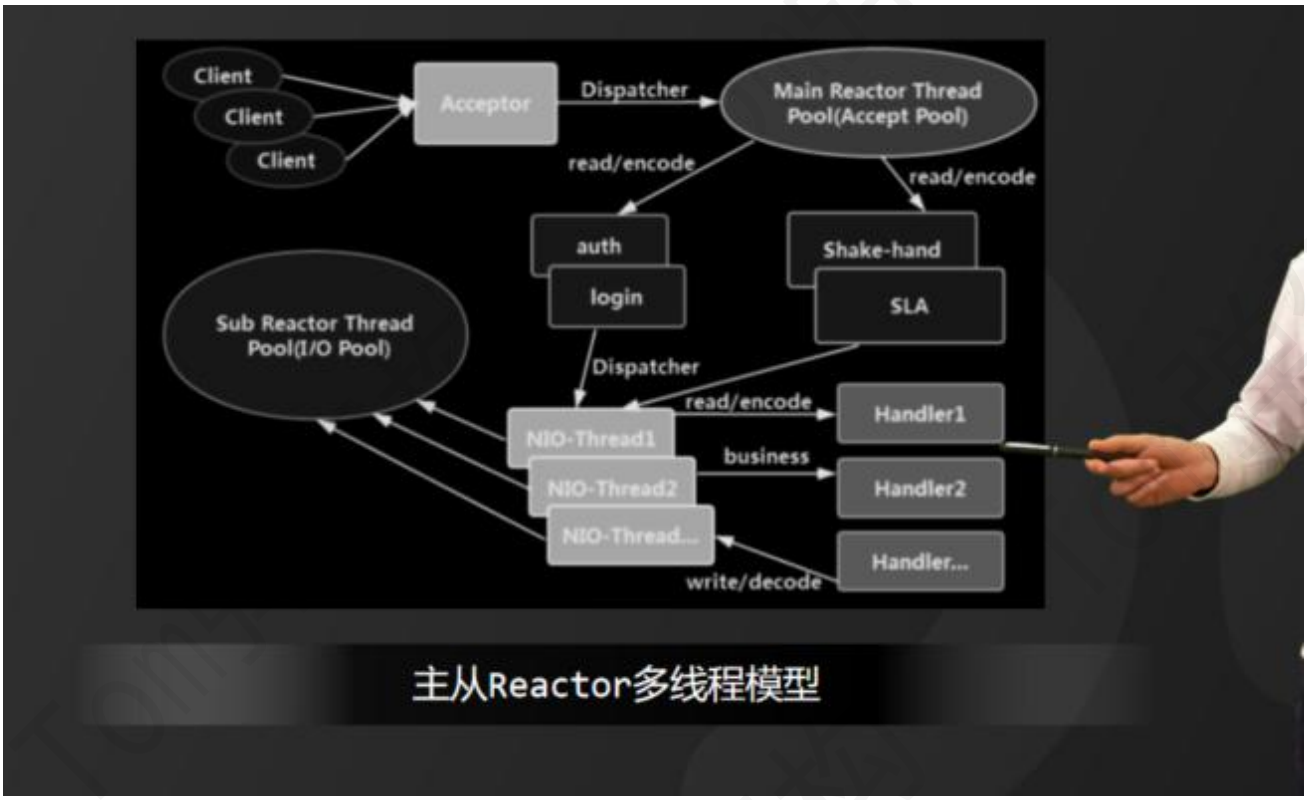


1、一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的读取和发送；

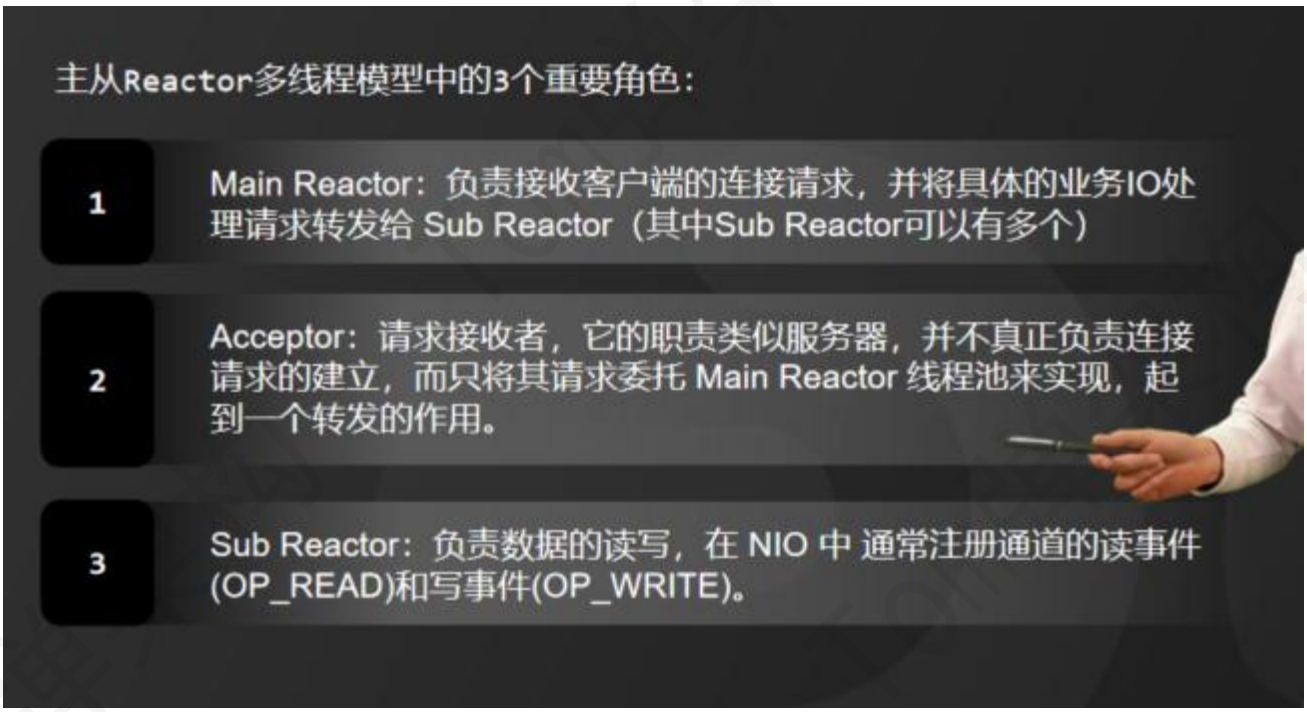
2、当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈；

所以，我们还可以更进一步优化，引入了主从 Reactor 多线程模式：

3、主从 Reactor 多线程模型



如图所示，在主从 Reactor 多线程模型中，由三个主要的角色：



Main Reactor：负责接收客户端的连接请求，并将具体的业务 IO 处理请求转发给 Sub Reactor（其中 Sub Reactor 可以有多个）。

Acceptor：请求接收者，它的职责类似服务器，并不真正负责连接请求的建立，而只将其请求委托 Main Reactor 线程池来实现，起到一个转发的作用。

Sub Reactor：负责数据的读写，在 NIO 中 通常注册通道的读事件(OP_READ)和写事件(OP_WRITE)。

好了，以上就是我对 Netty 线程模型的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！

46、Netty 中有哪些核心组件？

最近又有粉丝问我这样一个问题，说 Netty 中最核心的组件有哪些？它们都起什么作用？今天，给大家详细聊一聊

另外，我花了 1 个多星期，准备了一份 10W 字的面试题解析配套文档，想获取的小伙伴可以从我的个人煮叶简介中找到。

1、组件分层

我把 Netty 的核心组件分为三层，分别是网络通信层、事件调度层和服务编排层。

2、网络通信层

在网络通信层有三个核心组件：Bootstrap、ServerBootStrap、Channel

Bootstrap：负责客户端启动并用来链接远程 Netty Server；
ServerBootStrap：负责服务端监听，用来监听指定端口；
Channel：相当于完成网络通信的载体。

3、事件调度层

事件调度器有两个核心组件：EventLoopGroup 与 EventLoop

EventLoopGroup：本质上是一个线程池，主要负责接收 I/O 请求，并分配线程执行处理请求。

EventLoop：相当于线程池中的线程

4、服务编排层

在服务编排层有三个核心组件 ChannelPipeline、ChannelHandler、ChannelHandlerContext

ChannelPipeline：负责将多个 ChannelHandler 链接在一起

ChannelHandler：针对 I/O 的数据处理器，数据接收后，通过指定的 Handler 进行处理。

ChannelHandlerContext：用来保存 ChannelHandler 的上下文信息

以上就是我对 Netty 核心组件的理解。我是被编程耽误的文艺 Tom，如果我的分享对你有帮助，请动动手指一键三连分享给更多的人。关注我，面试不再难！