

# 谈谈你对 AQS 的理解

---

AQS 是 AbstractQueuedSynchronizer 的简称，是并发编程中比较核心的组件。

在很多大厂的面试中，面试官对于并发编程的考核要求相对较高，简单来说，如果你不懂并发编程，那么你很难通过大厂高薪岗位的面试。

Hello，大家好，我是 Mic，一个工作了 14 年的程序员，今天来和大家聊聊并发编程中的 AQS 组件。

我们来看一下，关于“谈谈你对 AQS 的理解”，看看普通人和高手是如何回答的！

## 普通人的回答

---

AQS 全称是 AbstractQueuedSynchronizer，它是 J.U.C 包中 Lock 锁的底层实现，可以用它来实现多线程的同步器！

## 高手的回答

---

AQS 是多线程同步器，它是 J.U.C 包中多个组件的底层实现，如 Lock、CountDownLatch、Semaphore 等都用到了 AQS。

从本质上来说，AQS 提供了两种锁机制，分别是排它锁，和共享锁。

排它锁，就是存在多线程竞争同一共享资源时，同一时刻只允许一个线程访问该共享资源，也就是多个线程中只能有一个线程获得锁资源，比如 Lock 中的 ReentrantLock 重入锁实现就是用到了 AQS 中的排它锁功能。

共享锁也称为读锁，就是在同一时刻允许多个线程同时获得锁资源，比如 CountDownLatch 和 Semaphore 都是用到了 AQS 中的共享锁功能。

## 结尾

---

好的，关于普通人和高手对于这个问题的回答，哪个更加好呢？你们如果有更好的回答，可以在下方评论区留言。

另外，我整理了一张比较完整的并发编程知识体系的脑图，大家感兴趣的可以私信我获取。

本期的普通人 VS 高手面试系列就到这里结束了，喜欢的朋友记得一键三连，加个关注，

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

# fail-safe 机制与 fail-fast 机制分别有什么作用

---

前段时间一个小伙伴去面试，遇到这样一个问题。

“fail-safe 机制与 fail-fast 机制分别有什么作用”

他说他听到这个问题的时候，脑子里满脸问号。那么今天我们来看一下，关于这个问题，普通人和高手应该如何回答吧。

## 普通人的回答

---

额....嗯...（持续几秒后，贴一个搞笑的图，比如 5 year s latter 之类，然后再配个一脸蒙蔽。。）

## 高手的回答

---

fail-safe 和 fail-fast，是多线程并发操作集合时的一种失败处理机制。

**Fail-fast:** 表示快速失败，在集合遍历过程中，一旦发现容器中的数据被修改了，会立刻抛出 `ConcurrentModificationException` 异常，从而导致遍历失败，像这种情况（贴下面这个图）。

定义一个 `Map` 集合，使用 `Iterator` 迭代器进行数据遍历，在遍历过程中，对集合数据做变更时，就会发生 `fail-fast`。

`java.util` 包下的集合类都是快速失败机制的，常见的的使用 `fail-fast` 方式遍历的容器有 `HashMap` 和 `ArrayList` 等。

```
public static void main(String[] args) {
    Map<String, String> empName = new HashMap<String, String>();
    empName.put("name", "mic");
    empName.put("sex", "male");
    empName.put("age", "18");
    Iterator iterator = empName.keySet().iterator();
    while (iterator.hasNext()) {
        System.out.println(empName.get(iterator.next()));
        empName.put("work", "Java");
    }
}
```

>上述程序运行结果如下:

male

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1445)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1469)
    at org.example.cl09.ThreadExample.main(ThreadExample.java:14)
```

**Fail-safe**，表示失败安全，也就是在这种机制下，出现集合元素的修改，不会抛出 **ConcurrentModificationException**。

原因是采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，

在拷贝的集合上进行遍历。由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到

比如这种情况（贴下面这个图），定义了一个 **CopyOnWriteArrayList**，在对这个集合遍历过程中，对集合元素做修改后，不会抛出异常，但同时也不会打印出增加的元素。

**java.util.concurrent** 包下的容器都是安全失败的,可以在多线程下并发使用,并发修改。

常见的的使用 **fail-safe** 方式遍历的容器有 **ConcerrentHashMap** 和 **CopyOnWriteArrayList** 等。

```
public static void main(String[] args) {  
    CopyOnWriteArrayList<Integer> list  
        = new CopyOnWriteArrayList<>(new Integer[] { 1, 7, 9, 11 });  
    Iterator itr = list.iterator();  
    while (itr.hasNext()) {  
        Integer i = (Integer)itr.next();  
        System.out.println(i);  
        if (i == 7)  
            list.add(15); // 在fail-safe模式下, 这里不会被打印  
    }  
}
```

## 结尾

---

好的，fail-safe 和 fail-fast 的作用，你理解了吗？

你们是否有更好的回答方式？欢迎在评论区给我留言！

本期的普通人 VS 高手面试系列就到这里结束了，喜欢的朋友记得一键三连，加个关注，

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 谈谈你对 Seata 的理解

---

很多面试官都喜欢问一些“谈谈你对 xxx 技术的理解”，

大家遇到这种问题时，是不是完全不知道从何说起，有同感小伙伴的 call 1.

那么我们来看一下，普通人和高手是如何回答这个问题的？

### 普通人

---

**Seata** 是用来解决分布式事务问题的框架。是阿里开源的中间件。

实际项目中我没有用过，我记得 **Seata** 里面有几种事务模型，有一种 **AT** 模式、还有 **TCC** 模式。

然后 **AT** 是一种二阶段提交的事务，它是采用的最终一致性来实现数据的一致性。

## 高手

在微服务架构下，由于数据库和应用服务的拆分，导致原本一个事务单元中的多个 **DML** 操作，变成了跨进程或者跨数据库的多个事务单元的多个 **DML** 操作，

而传统的数据库事务无法解决这类的问题，所以就引出了分布式事务的概念。

分布式事务本质上要解决的就是跨网络节点的多个事务的数据一致性问题，业内常见的解决方法有两种

强一致性，就是所有的事务参与者要么全部成功，要么全部失败，全局事务协调者需要知道每个事务参与者的执行状态，再根据状态来决定数据的提交或者回滚！

最终一致性，也叫弱一致性，也就是多个网络节点的数据允许出现不一致的情况，但是在最终的某个时间点会达成数据一致。

基于 **CAP** 定理我们可以知道，强一致性方案对于应用的性能和可用性会有影响，所以对于数据一致性要求不高的场景，就会采用最终一致性算法。

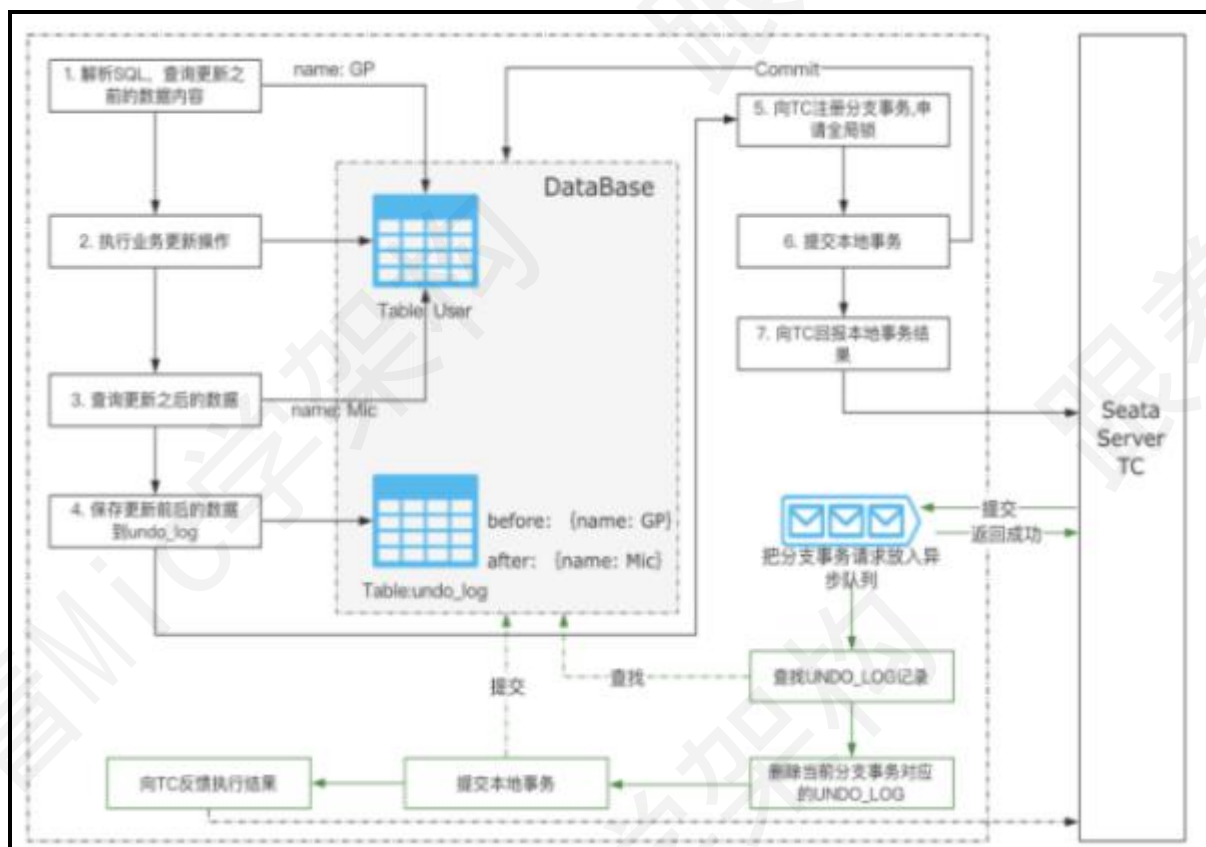
在分布式事务的实现上，对于强一致性，我们可以通过基于 **XA** 协议下的二阶段提交来实现，对于弱一致性，可以基于 **TCC** 事务模型、可靠性消息模型等方案来实现。

市面上有很多针对这些理论模型实现的分布式事务框架，我们可以在应用中集成这些框架来实现分布式事务。

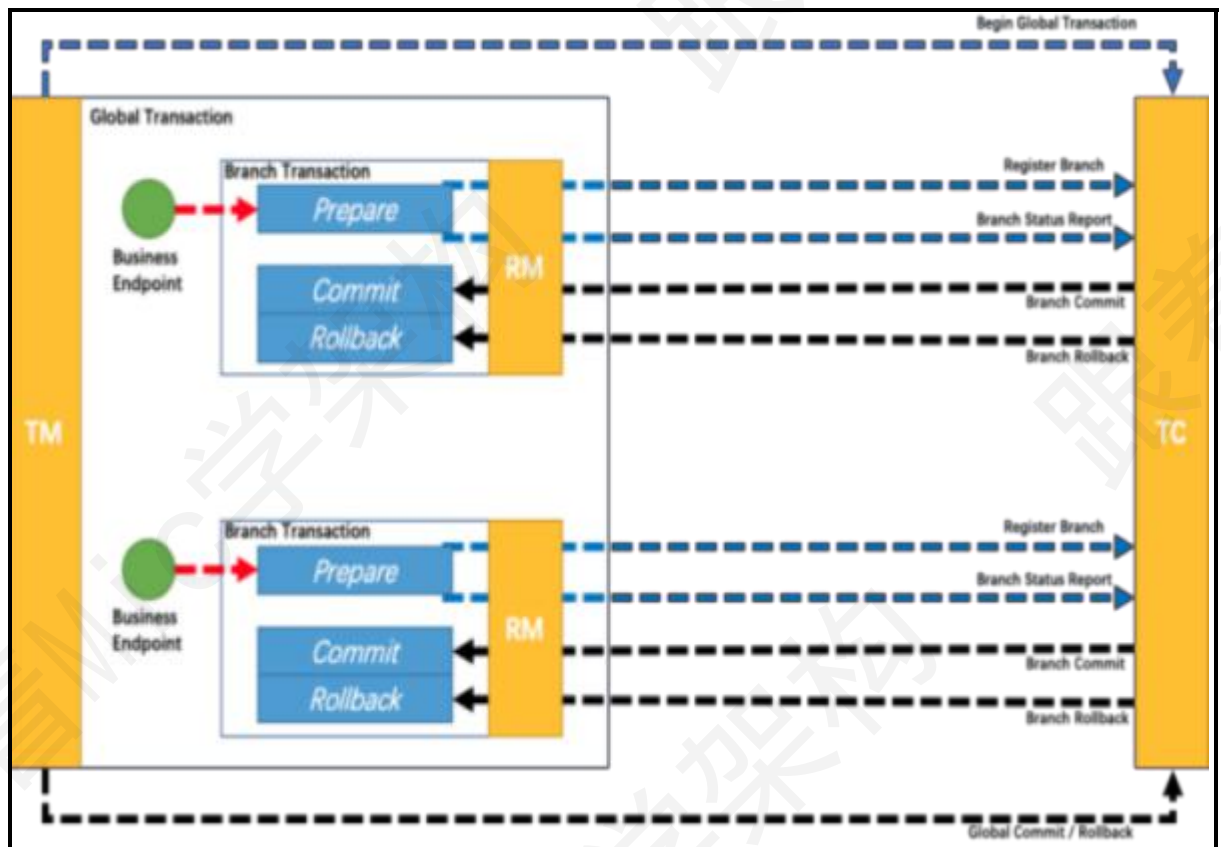
而 **Seata** 就是其中一种，它是阿里开源的分布式事务解决方案，提供了高性能且简单易用的分布式事务服务。

**Seata** 中封装了四种分布式事务模式，分别是：

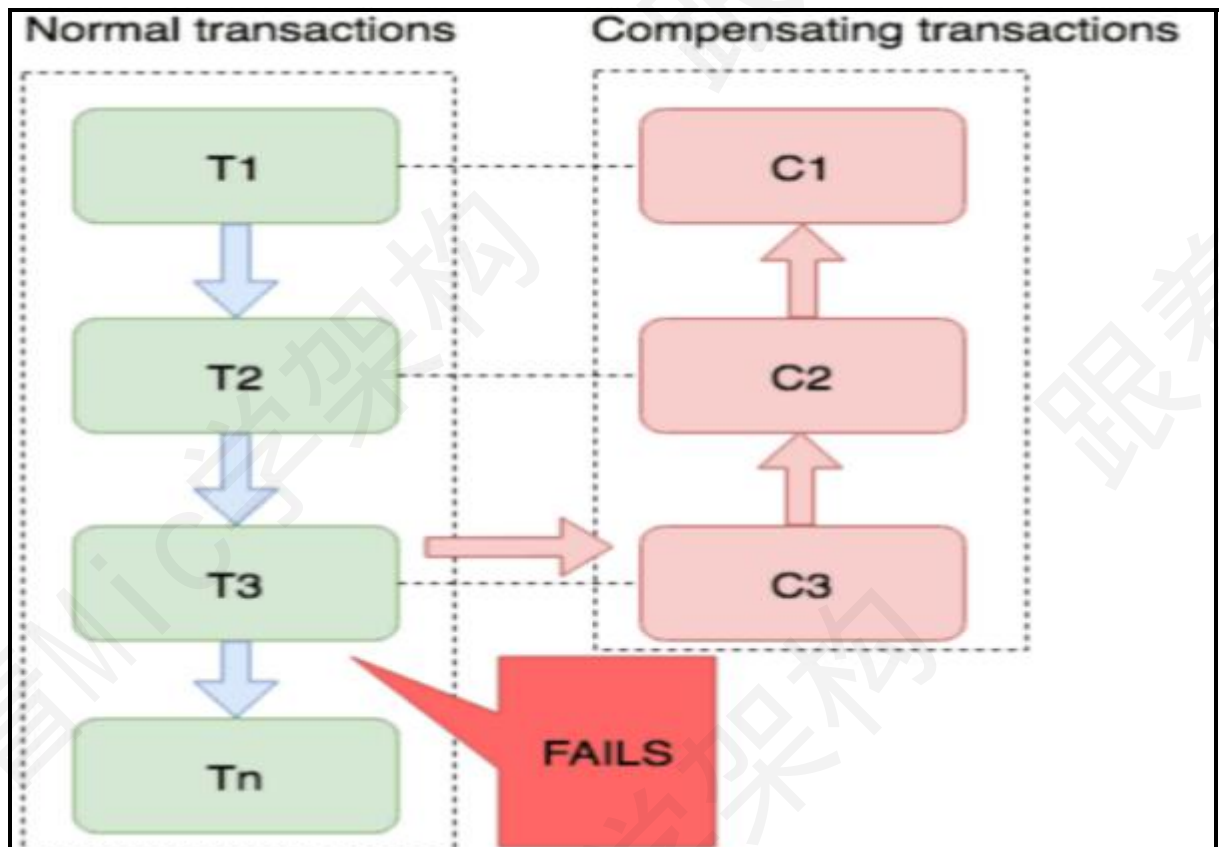
**AT** 模式，是一种基于本地事务+二阶段协议来实现的最终数据一致性方案，也是 **Seata** 默认的解决方案



TCC 模式，TCC 事务是 Try、Confirm、Cancel 三个词语的缩写，简单理解就是把一个完整的业务逻辑拆分成三个阶段，然后通过事务管理器在业务逻辑层面根据每个分支事务的执行情况分别调用该业务的 Confirm 或者 Cacel 方法。

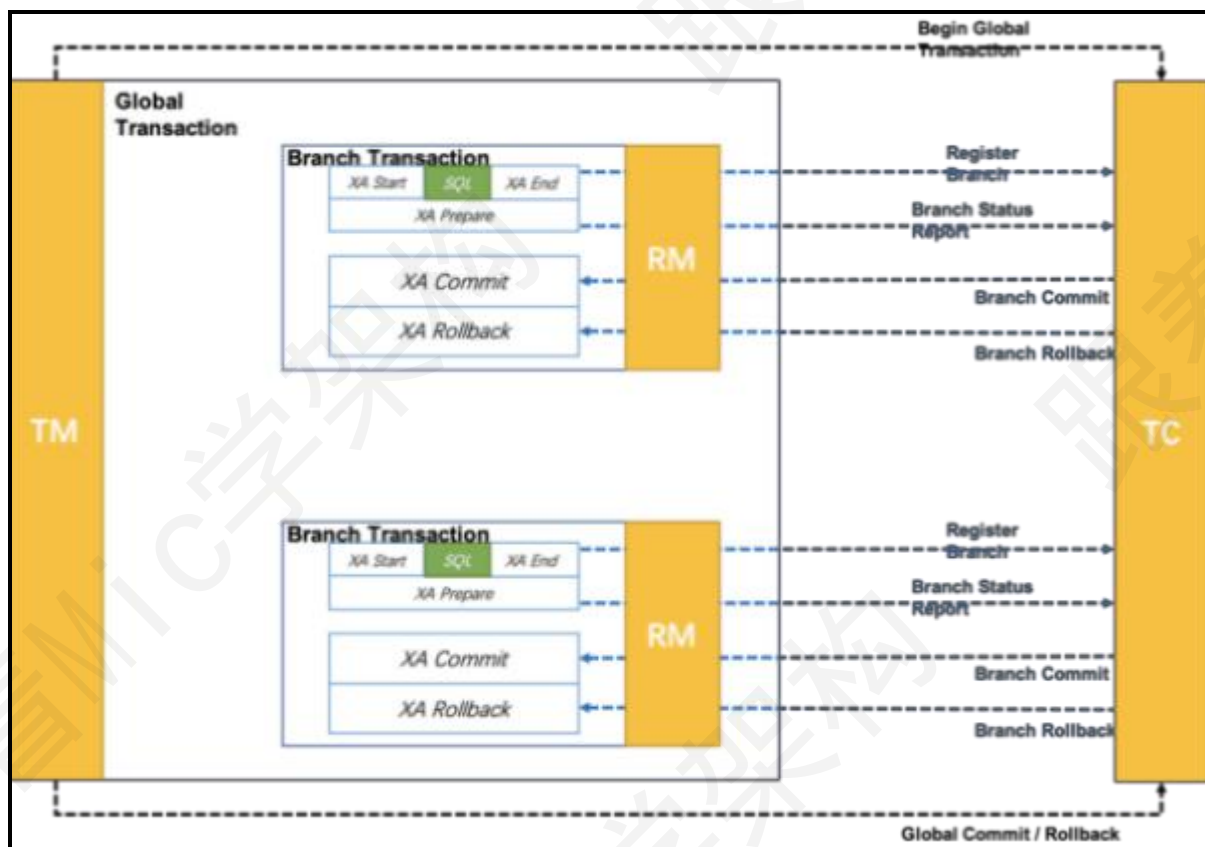


Saga 模式，Saga 模式是 SEATA 提供的长事务解决方案，在 Saga 模式中，业务流程中每个参与者都提交本地事务，当出现某一个参与者失败则补偿前面已经成功的参与者。



XA 模式，XA 可以认为是一种强一致性的事务解决方法，它利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种事务模式。





从这四种模型中不难看出，在不同的业务场景中，我们可以使用 Seata 的不同事务模型来解决不同业务场景中的分布式事务问题，因此我们可以认为 Seata 是一个一站式的分布式事务解决方案。

## 结尾

屏幕前的小伙伴们，你是否通过高手的回答找到了这类问题的回答方式呢？

面试的时候遇到这种宽泛的问题时，先不用慌，首先自己要有一个回答的思路。

按照技术的话术，就是先给自己大脑中的知识建立一个索引，然后基于索引来定位你的知识。

我对于这类问题，建立的索引一般有几个：

它是什么

它能解决什么问题

它有哪些特点和优势

它的核心原理，为什么能解决这类问题

大家对照这几个索引去回答今天的这个面试题，是不是就更清晰了？

好的，本期的普通人 VS 高手面试系列就到这里结束了，喜欢的朋友记得一键三连，加个关注，

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Spring Boot 的约定优于配置，你的理解是什么？

---

对于 Spring Boot 约定优于配置这个问题，看看普通人和高手是如何回答的？

### 普通人的回答

---

嗯，在 Spring Boot 里面，通过约定优于配置这个思想，可以让我们少写很多的配置，

然后就只需要关注业务代码的编写就行。嗯！

### 高手的回答

---

我从 4 个点方面来回答。

首先，约定优于配置是一种软件设计的范式，它的核心思想是减少软件开发人员对于配置项的维护，从而让开发人员更加聚焦在业务逻辑上。

Spring Boot 就是约定优于配置这一理念下的产物，它类似于 Spring 框架下的一个脚手架，通过 Spring Boot，我们可以快速开发基于 Spring 生态下的应用程序。

基于传统的 Spring 框架开发 web 应用，我们需要做很多和业务开发无关并且只需要做一次的配置，比如

管理 jar 包依赖

web.xml 维护

Dispatch-Servlet.xml 配置项维护

应用部署到 Web 容器

第三方组件集成到 Spring IOC 容器中的配置项维护

而在 Spring Boot 中，我们不需要再去做这些繁琐的配置，Spring Boot 已经自动帮我们完成了，这就是约定由于配置思想的体现。

Spring Boot 约定由于配置的体现有很多，比如

Spring Boot Starter 启动依赖，它能帮我们管理所有 jar 包版本

如果当前应用依赖了 spring mvc 相关的 jar，那么 Spring Boot 会自动内置 Tomcat 容器来运行 web 应用，我们不需要再去单独做应用部署。

Spring Boot 的自动装配机制的实现中，通过扫描约定路径下的 spring.factories 文件来识别配置类，实现 Bean 的自动装配。

默认加载的配置文件 application.properties 等等。

总的来说，约定优于配置是一个比较常见的软件设计思想，它的核心本质都是为了更高效以及更便捷的实现软件系统的开发和维护。

以上就是我对这个问题的理解。

## 结尾

---

好的，本期的普通人 VS 高手面试系列就到这里结束了，对于这个问题，你知道该怎么回答了吗？

另外，如果你有任何面试相关的疑问，欢迎评论区给我留言。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 滴滴二面：kafka 的零拷贝原理？

---

最近一个学员去滴滴面试，在第二面的时候遇到了这个问题：

“请你简单说一下 Kafka 的零拷贝原理”

然后那个学员努力在大脑里检索了很久，没有回答上来。

那么今天，我们基于这个问题来看看，普通人和高手是如何回答的！

### 普通人的回答

---

零拷贝是一种减少数据拷贝的机制，能够有效提升数据的效率

## 高手的回答

在实际应用中，如果我们需要把磁盘中的某个文件内容发送到远程服务器上，如图

那么它必须要经过几个拷贝的过程，如图。

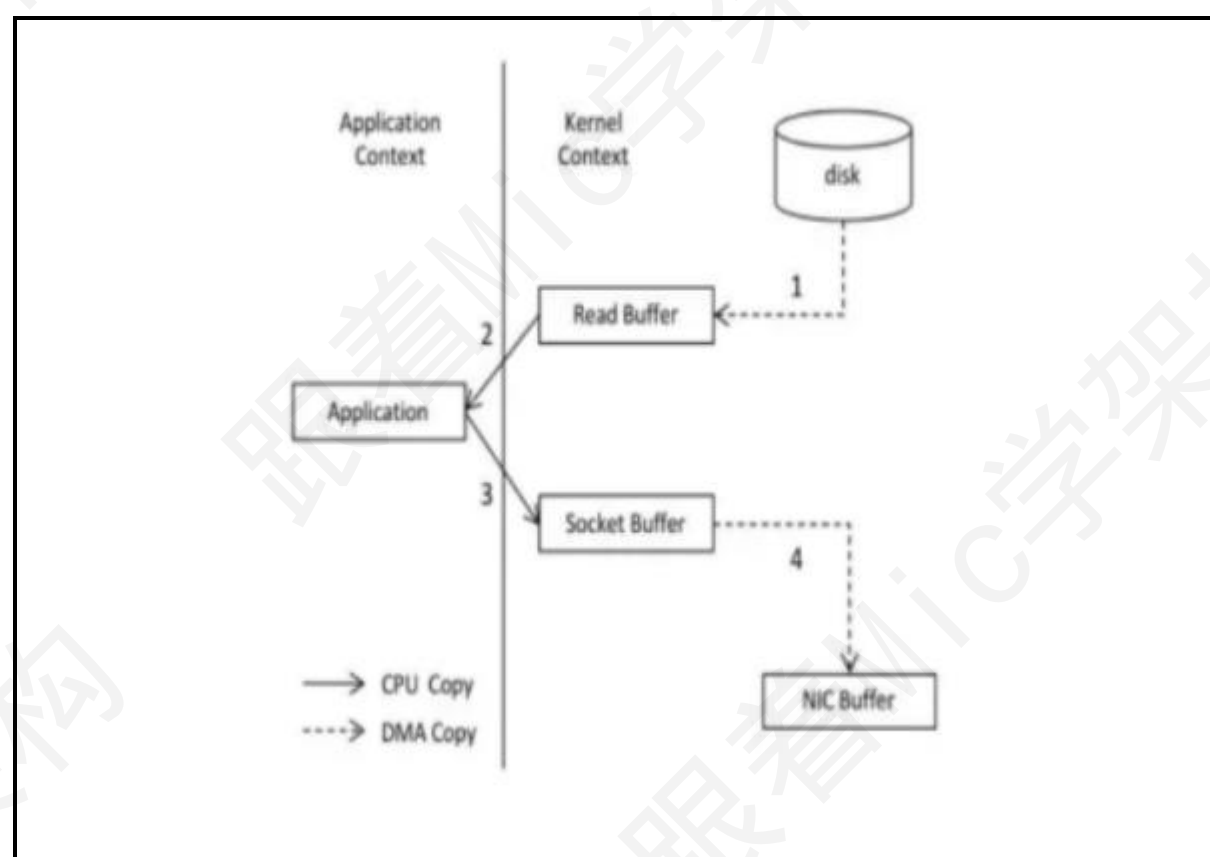
从磁盘中读取目标文件内容拷贝到内核缓冲区

CPU 控制器再把内核缓冲区的数据赋值到用户空间的缓冲区中

接着在应用程序中，调用 `write()` 方法，把用户空间缓冲区中的数据拷贝到内核下的 **Socket Buffer** 中。

最后，把在内核模式下的 **SocketBuffer** 中的数据赋值到网卡缓冲区 (**NIC Buffer**)

网卡缓冲区再把数据传输到目标服务器上。



在这个过程中我们可以发现，数据从磁盘到最终发送出去，要经历 4 次拷贝，而在这四次拷贝过程中，有两次拷贝是浪费的，分别是：

从内核空间赋值到用户空间

从用户空间再次复制到内核空间

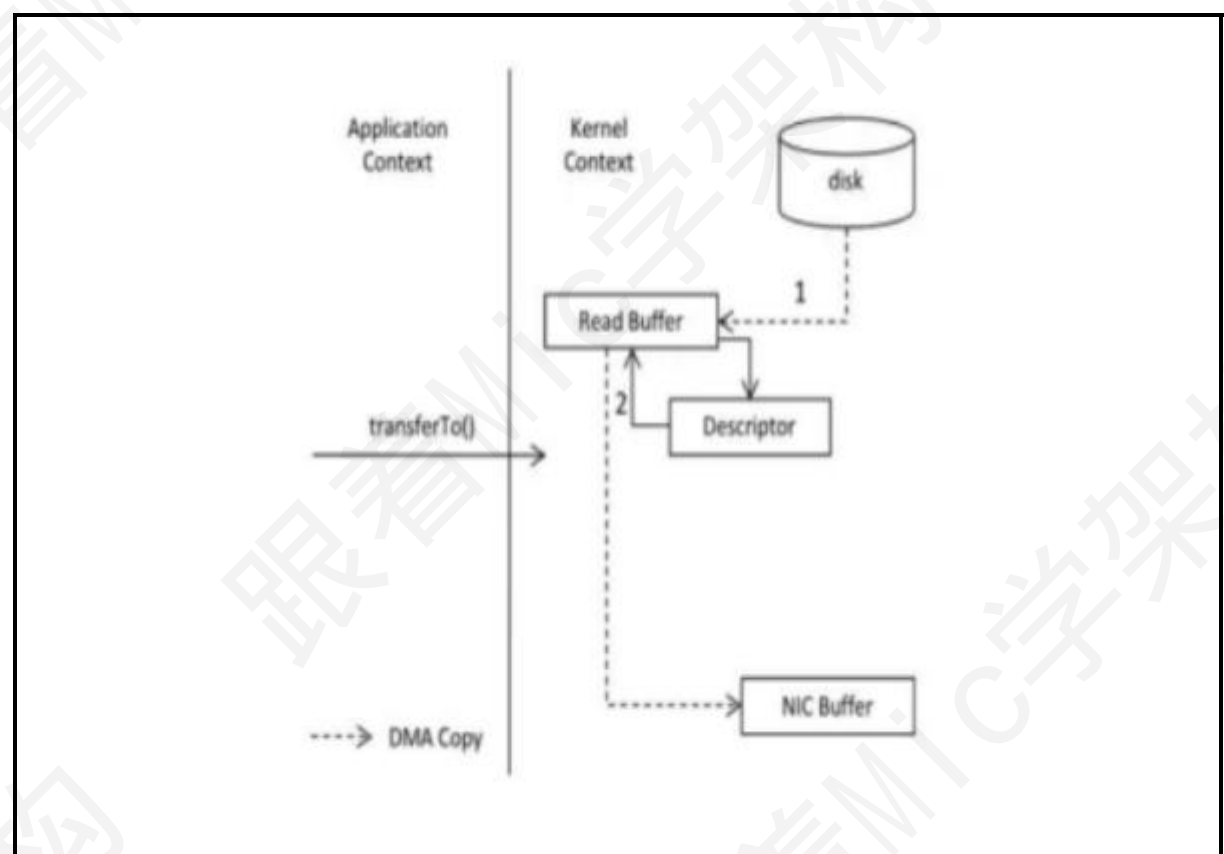
除此之外,由于用户空间和内核空间的切换会带来CPU的上下文切换,对于CPU性能也会造成性能影响。

而零拷贝,就是把这两次多余的拷贝省略掉,应用程序可以直接把磁盘中的数据从内核中直接传输给Socket,而不需要再经过应用程序所在的用户空间,如下图所示。

零拷贝通过DMA (Direct Memory Access) 技术把文件内容复制到内核空间中的Read Buffer,

接着把包含数据位置和长度信息的文件描述符加载到Socket Buffer中,DMA引擎直接可以把数据从内核空间中传递给网卡设备。

在这个流程中,数据只经历了两次拷贝就发送到了网卡中,并且减少了2次cpu的上下文切换,对于效率有非常大的提高。



所以,所谓零拷贝,并不是完全没有数据赋值,只是相对于用户空间来说,不再需要进行数据拷贝。对于前面说的整个流程来说,零拷贝只是减少了不必要的拷贝次数而已。

在程序中如何实现零拷贝呢?

在Linux中,零拷贝技术依赖于底层的 `sendfile()` 方法实现

在 Java 中，FileChannel.transferTo()方法的底层实现就是 sendfile()方法。

除此之外，还有一个 mmap 的文件映射机制

它的原理是：将磁盘文件映射到内存,用户通过修改内存就能修改磁盘文件。使用这种方式可以获得很大的 I/O 提升，省去了用户空间到内核空间复制的开销。

以上就是我对于 Kafka 中零拷贝原理的理解

## 结尾

---

好的，本期的普通人 VS 高手面试系列就到这里结束了。

本次的面试题涉及到一些计算机底层的原理，基本上也是业务程序员的知识盲区。

但我想提醒大家，做开发其实和建房子一样，要想楼层更高更稳，首先地基要打牢固。

另外，如果你有任何面试相关的疑问，欢迎评论区给我留言。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## innoDB 如何解决幻读

---

前天有个去快手面试的小伙伴私信我，他遇到了这样一个问题：“InnoDB 如何解决幻读”？

这个问题确实不是很好回答，在实际应用中，很多同学几乎都不关注数据库的事务隔离性。

所有问题基本就是 CRUD，一把梭~

那么今天，我们看一下关于“InnoDB 如何解决幻读”这个问题，普通人和高手的回答！

### 普通人

---

嗯，我印象中，幻读是通过 MVCC 机制来解决的，嗯....

MVCC 类似于一种乐观锁的机制，通过版本的方式来区分不同的并发事务，避免幻读问题！

### 高手

---

我会从三个方面来回答：

## 1、Mysql 的事务隔离级别

Mysql 有四种事务隔离级别，这四种隔离级别代表当存在多个事务并发冲突时，可能出现的脏读、不可重复读、幻读的问题。

其中 InnoDB 在 RR 的隔离级别下，解决了幻读的问题。

| 事务隔离级别                  | 脏读  | 不可重复读 | 幻读         |
|-------------------------|-----|-------|------------|
| 未提交读 (Read Uncommitted) | 可能  | 可能    | 可能         |
| 已提交读 (Read Committed)   | 不可能 | 可能    | 可能         |
| 可重复读 (Repeatable Read)  | 不可能 | 不可能   | 对InnoDB不可能 |
| 串行化 (Serializable)      | 不可能 | 不可能   | 不可能        |

## 2、什么是幻读？

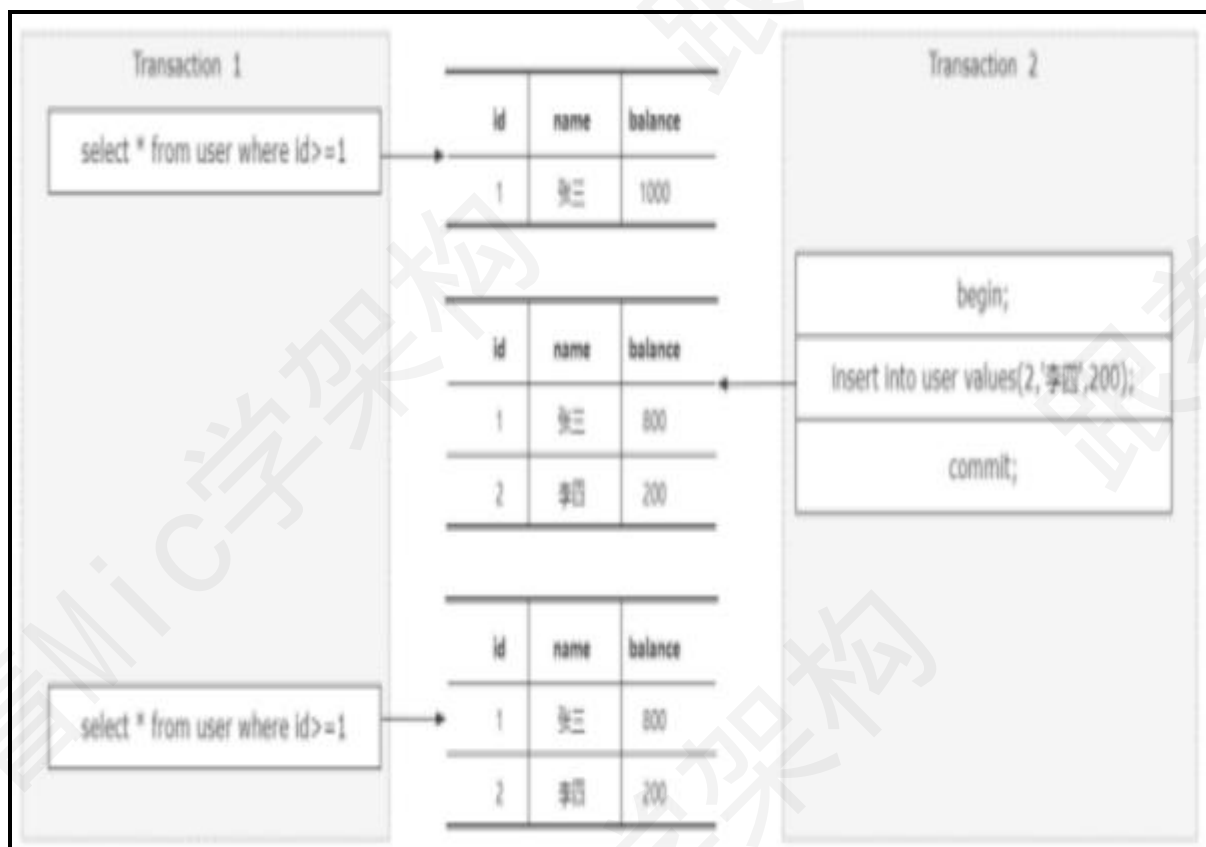
那么，什么是幻读呢？

幻读是指在同一个事务中，前后两次查询相同的范围时，得到的结果不一致（我们来看这个图）

第一个事务里面我们执行了一个范围查询，这个时候满足条件的数据只有一条

第二个事务里面，它插入了一行数据，并且提交了

接着第一个事务再去查询的时候，得到的结果比第一查询的结果多出来了一条数据。



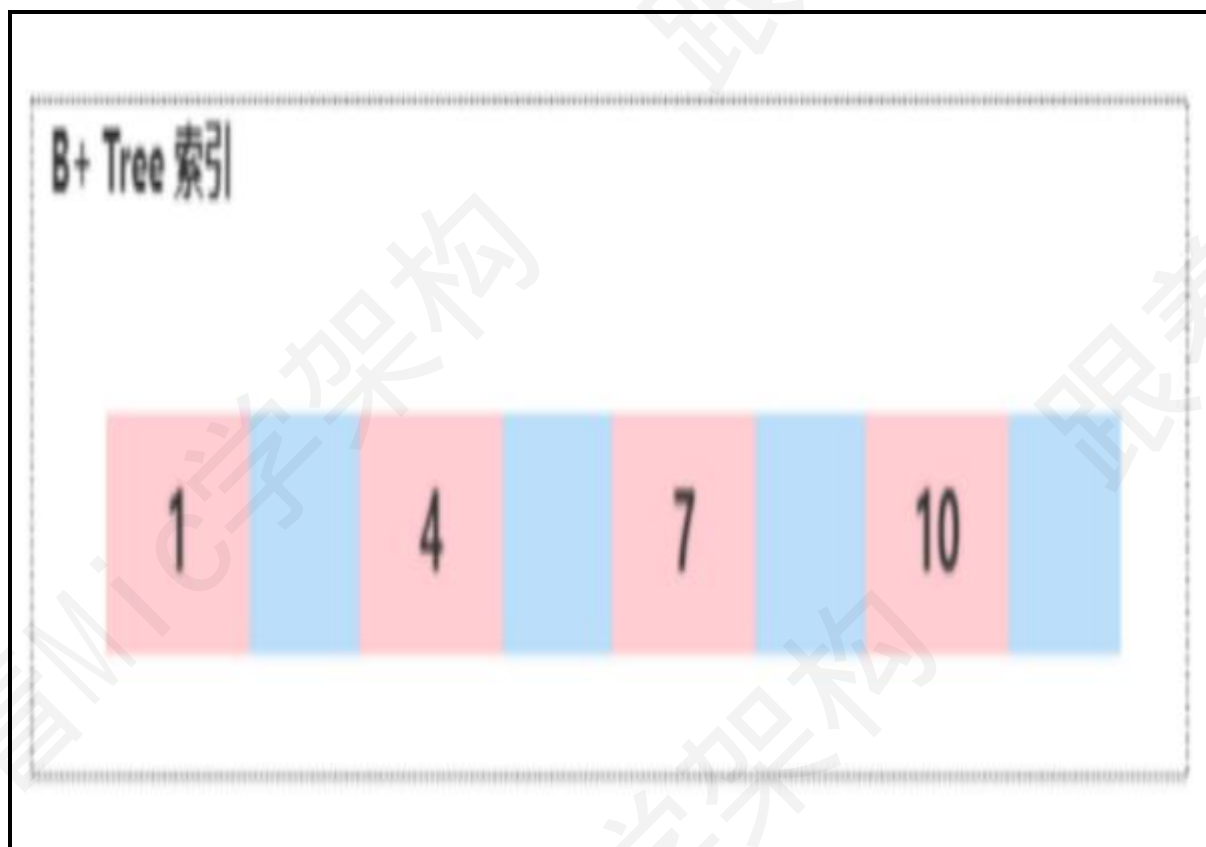
所以，幻读会带来数据一致性问题。

### 3、InnoDB 如何解决幻读的问题

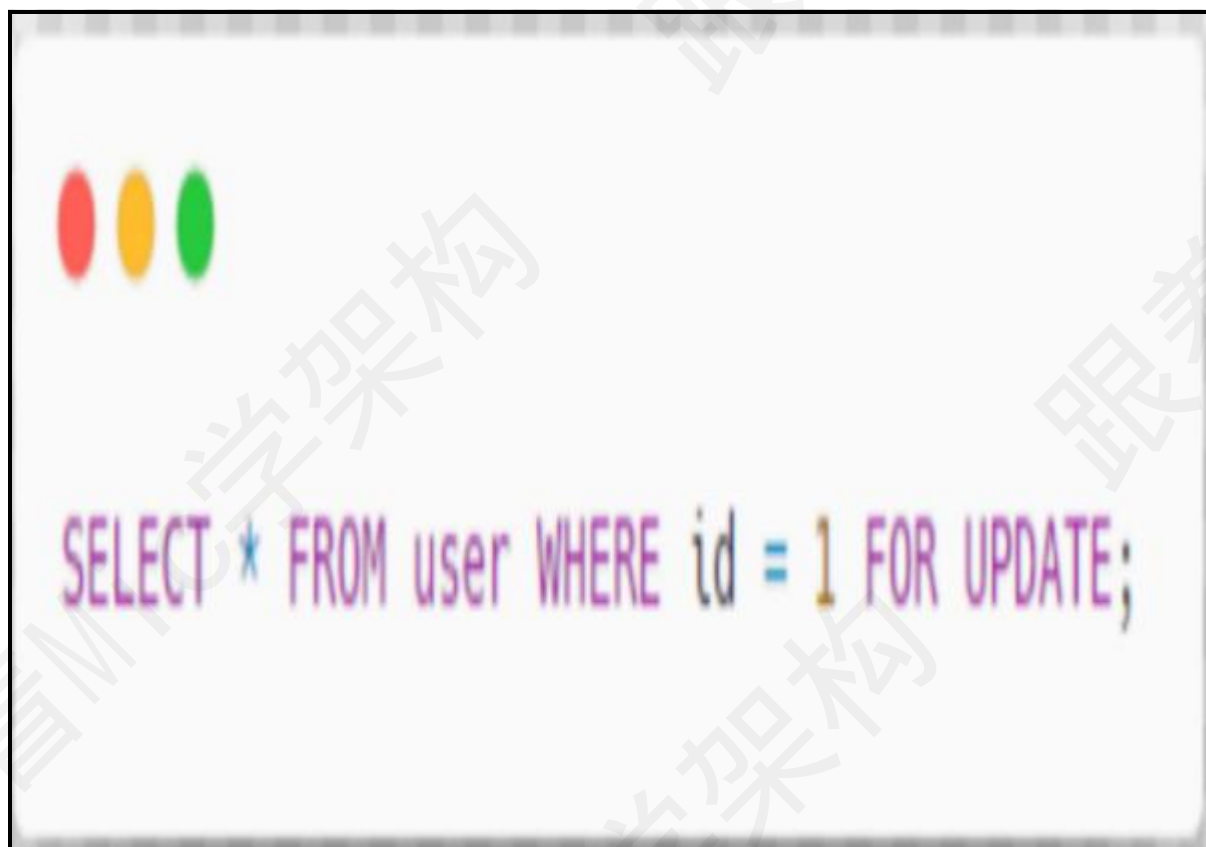
InnoDB 引入了间隙锁和 **next-key Lock** 机制来解决幻读问题，为了更清晰的说明这两种锁，我举一个例子：

假设现在存在这样（图片）这样一个 **B+Tree** 的索引结构，这个结构中有四个索引元素分别是：1、4、7、10。

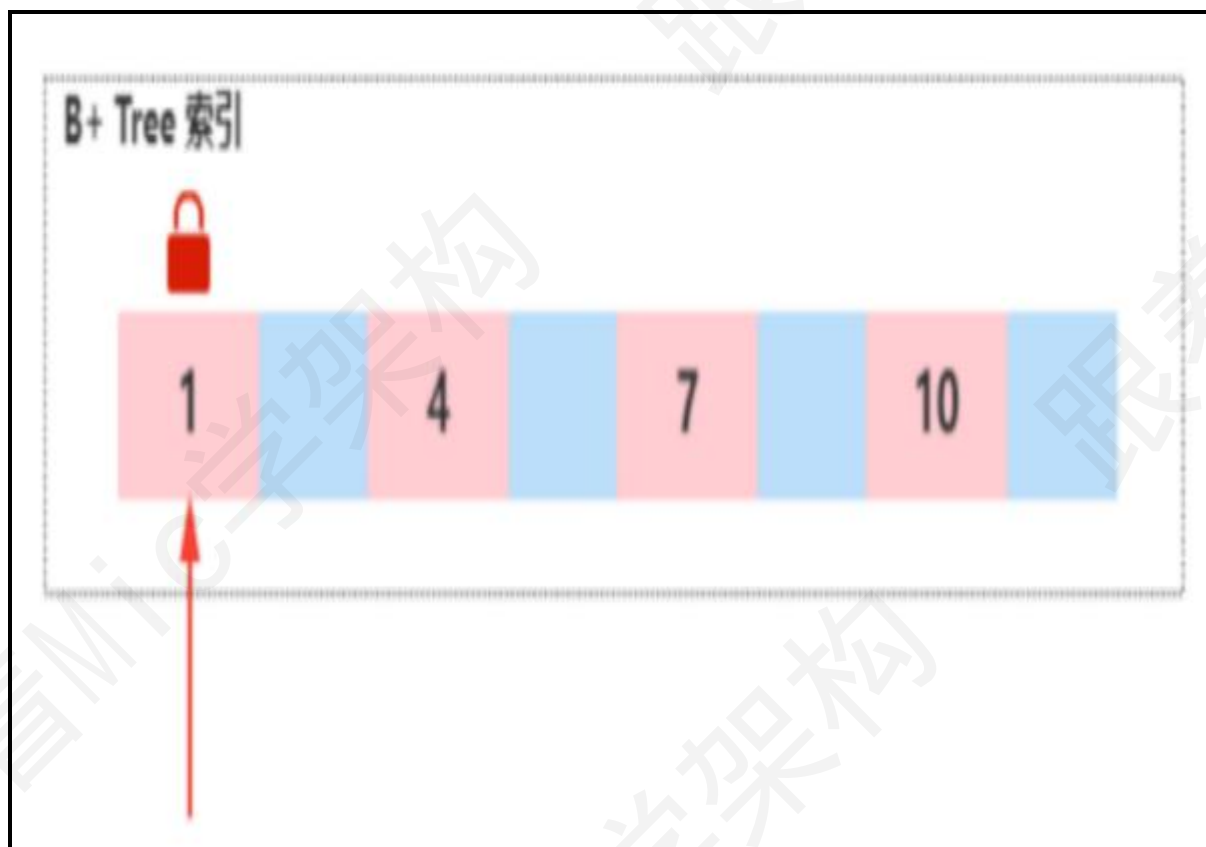




当我们通过主键索引查询一条记录，并且对这条记录通过 `for update` 加锁（请看这个图片）



这个时候，会产生一个记录锁，也就是行锁，锁定 `id=1` 这个索引（请看这个图片）。

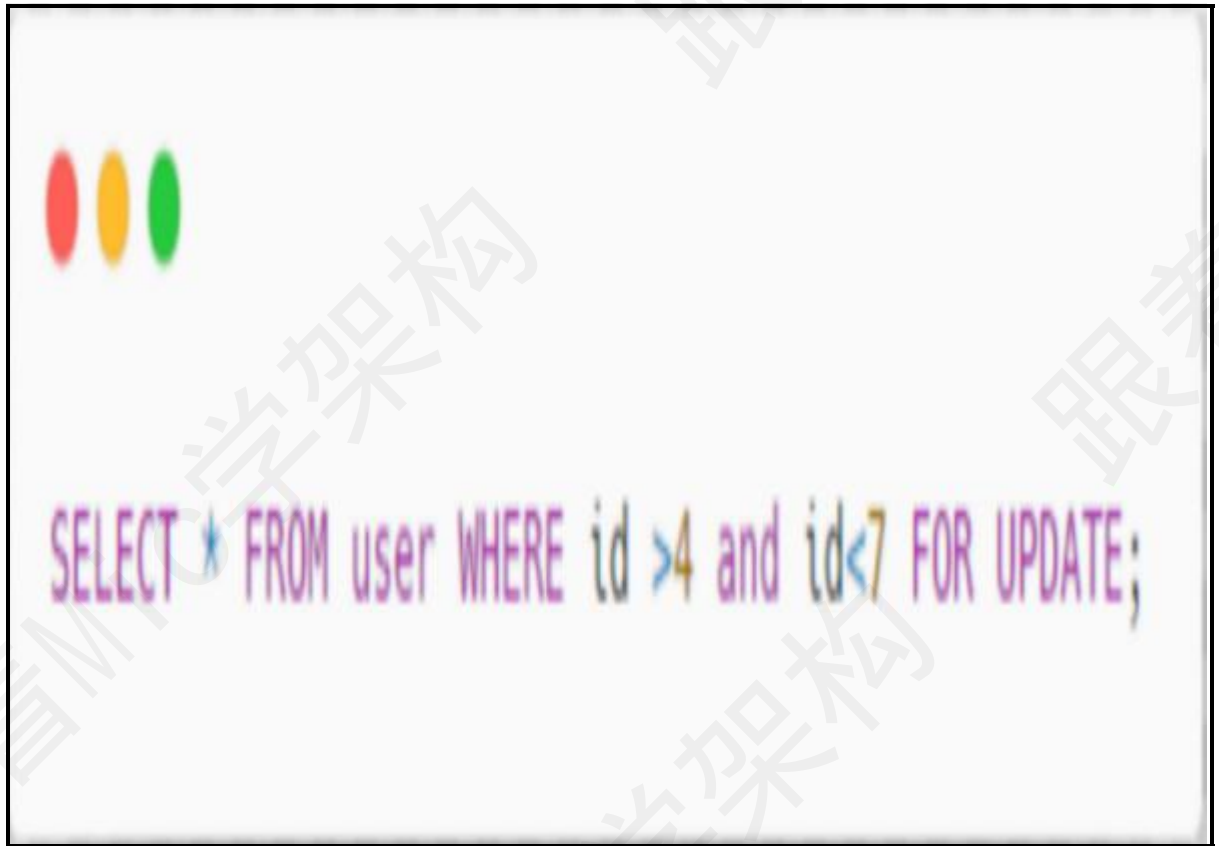


被锁定的记录在锁释放之前，其他事务无法对这条记录做任何操作。

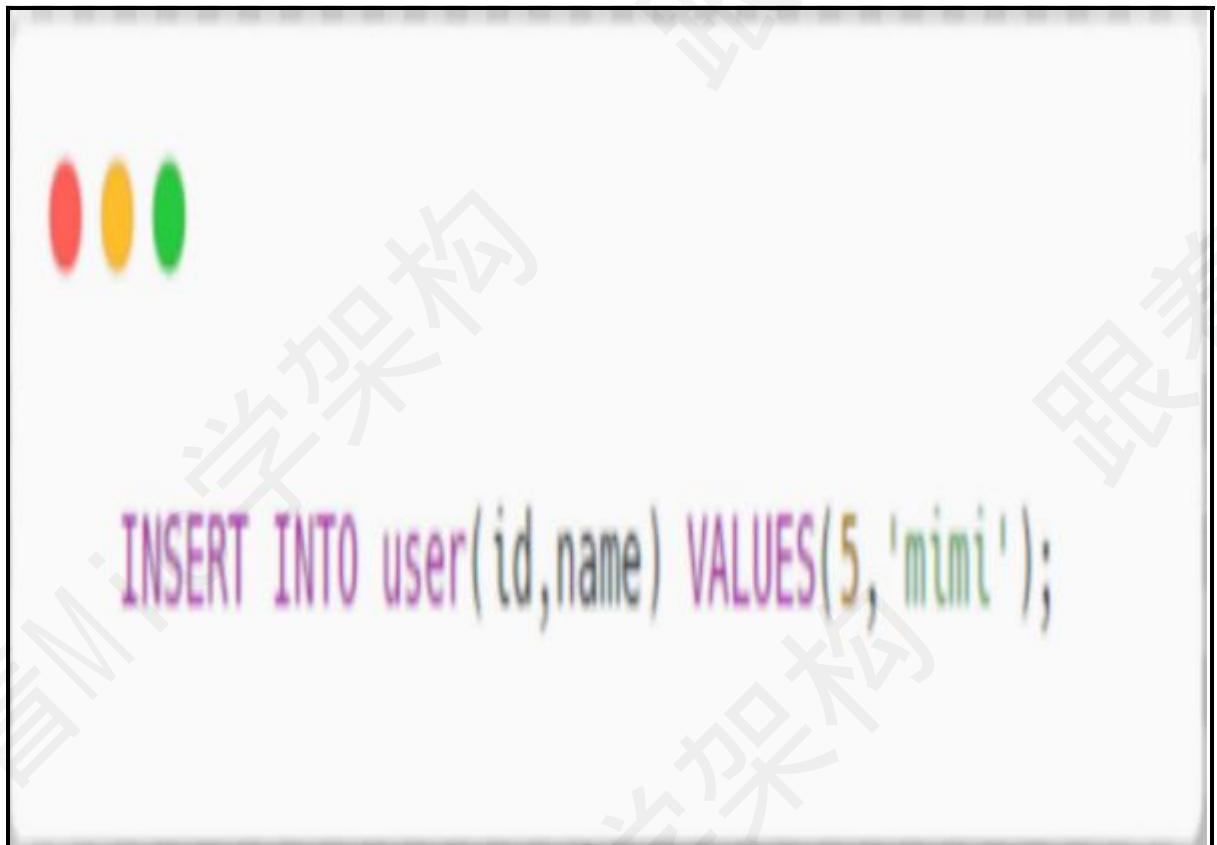
前面我说过对幻读的定义：幻读是指在同一个事务中，前后两次查询相同的范围时，得到的结果不一致！

注意，这里强调的是范围查询，

也就是说，InnoDB 引擎要解决幻读问题，必须要保证一个点，就是如果一个事务通过这样一条语句进行锁定时。



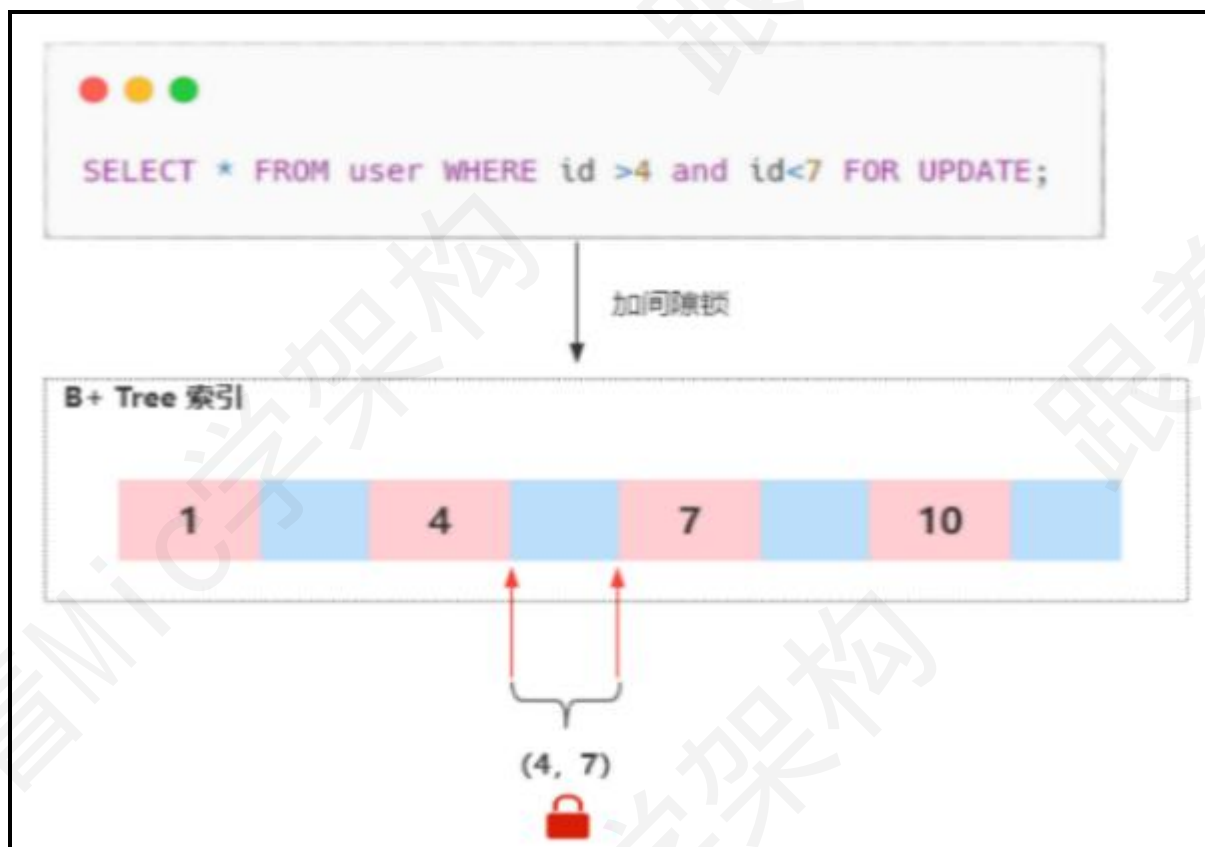
另外一个事务再执行这样一条（显示图片）insert 语句，需要被阻塞，直到前面获得锁的事务释放。



所以，在 InnoDB 中设计了一种间隙锁，它的主要功能是锁定一段范围内的索引记录

当对查询范围 `id>4 and id<7` 加锁的时候，会针对 B+树中（4，7）这个开区间范围的索引加间隙锁。

意味着在这种情况下，其他事务对这个区间的数据进行插入、更新、删除都会被锁住。

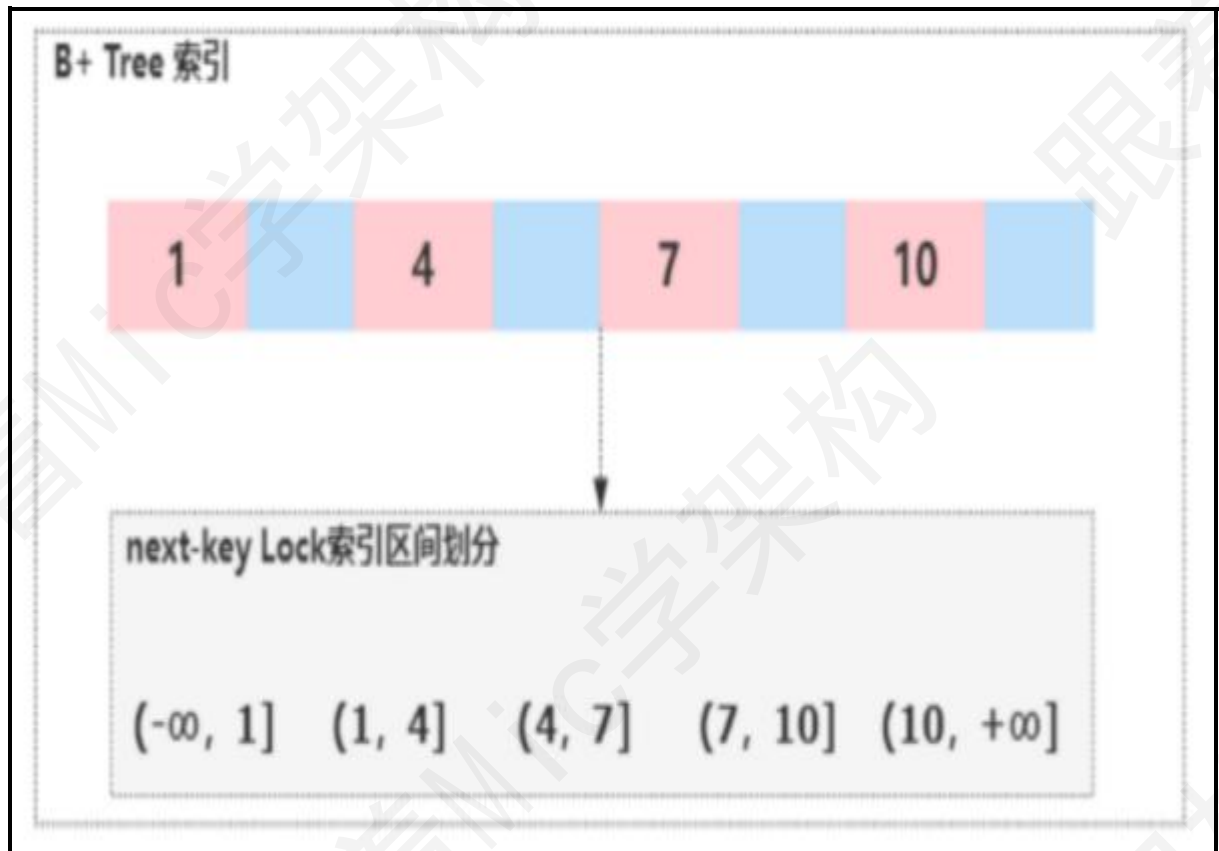


但是，还有另外一种情况，比如像这样（图片）



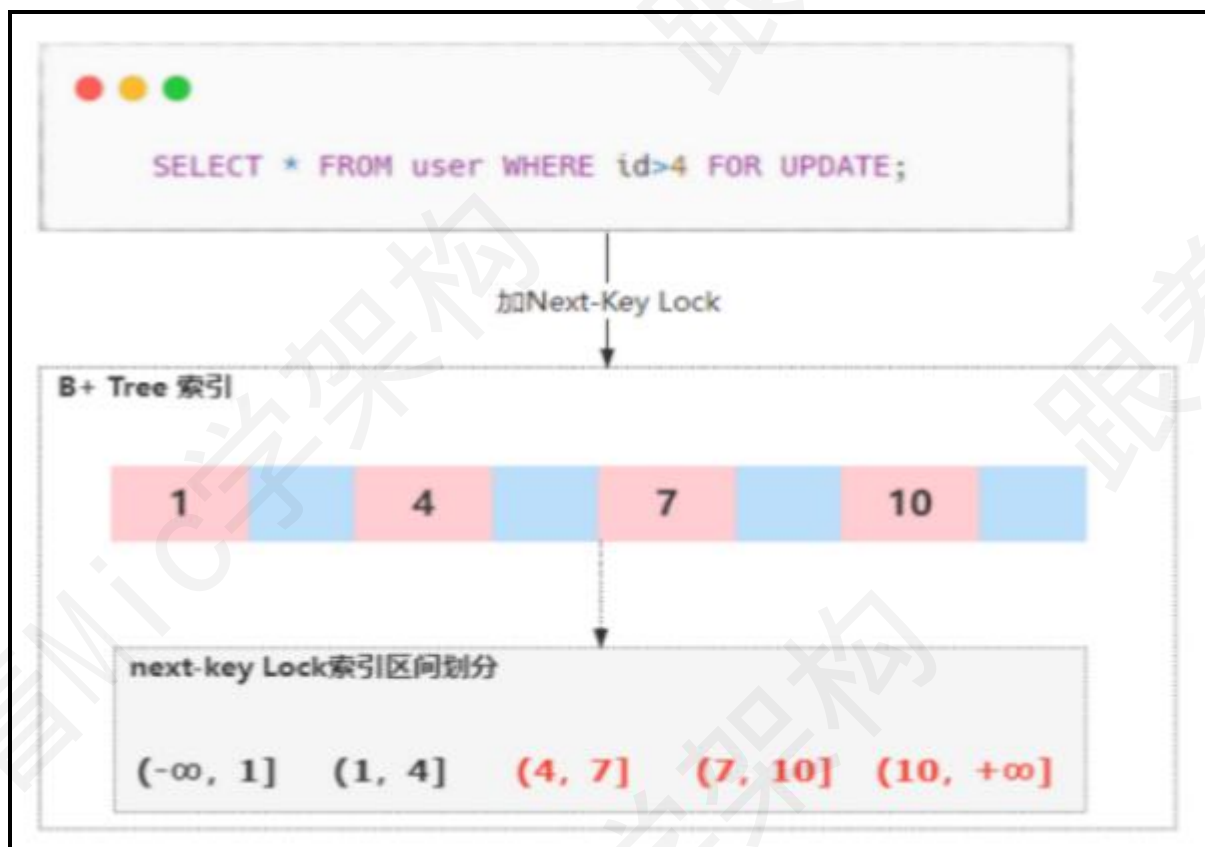
这条查询语句是针对  $id > 4$  这个条件加锁，那么它需要锁定多个索引区间，所以在 InnoDB 引入了 next-key Lock 机制。

next-key Lock 相当于间隙锁和记录锁的合集，记录锁锁定存在的记录行，间隙锁锁住记录行之间的间隙，而 next-key Lock 锁住的是两者之和。（如图所示）



每个数据行上的非唯一索引列上都会存在一把 next-key lock，当某个事务持有该数据行的 next-key lock 时，会锁住一段左开右闭区间的数据。

因此，当通过  $id > 4$  这样一种范围查询加锁时，会加 next-key Lock，锁定的区间范围是： $(4, 7], (7, 10], (10, +\infty]$



间隙锁和 next-key Lock 的区别在于加锁的范围，间隙锁只锁定两个索引之间的引用间隙，而 next-key Lock 会锁定多个索引区间，它包含记录锁和间隙锁。

当我们使用了范围查询，不仅仅命中了 Record 记录，还包含了 Gap 间隙，在这种情况下我们使用的就是临键锁，它是 MySQL 里面默认的行锁算法。

#### 4、总结

虽然 InnoDB 中通过间隙锁的方式解决了幻读问题，但是加锁之后一定会影响到并发性能，因此，如果对性能要求较高的业务场景中，可以把隔离级别设置成 RC，这个级别中不存在间隙锁。

以上就是我对于 InnoDB 如何解决幻读问题的理解！

## 结尾

好的，通过这个面试题可以发现，大厂面试对于基本功的考察还是比较严格的。

不过，不管是为了应付面试，还是为以后的职业规划做铺垫，技术能力的高低都是你在这个行业的核心竞争力。

本期的普通人 VS 高手面试系列的视频就到这里结束了，

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。



# CPU 飙高系统反应慢怎么排查？

---

面试过程中，场景类的问题更容易检测出一个开发人员的基本能力。

这不，一个小伙伴去阿里面试，第一面就遇到了关于“CPU 飙高系统反应慢怎么排查”的问题？

对于这个问题，我们来看看普通人和高手的回答！

## 普通人

---

嗯，CPU 飙高的原因可能是线程创建过多导致的

## 高手

---

好的，关于这个问题，我从四个方面来回答。

CPU 是整个电脑的核心计算资源，对于一个应用进程来说，CPU 的最小执行单元是线程。

导致 CPU 飙高的原因有几个方面

CPU 上下文切换过多，对于 CPU 来说，同一时刻下每个 CPU 核心只能运行一个线程，如果有多个线程要执行，CPU 只能通过上下文切换的方式来执行不同的线程。上下文切换需要做两件事情

保存运行线程的执行状态

让处于等待中的线程执行

这两个过程需要 CPU 执行内核相关指令实现状态保存，如果较多的上下文切换会占据大量 CPU 资源，从而使得 cpu 无法去执行用户进程中的指令，导致响应速度下降。

在 Java 中，文件 IO、网络 IO、锁等待、线程阻塞等操作都会造成线程阻塞从而触发上下文切换

CPU 资源过度消耗，也就是在程序中创建了大量的线程，或者有线程一直占用 CPU 资源无法被释放，比如死循环！

CPU 利用率过高之后，导致应用中的线程无法获得 CPU 的调度，从而影响程序的执行效率！

既然是这两个问题导致的 CPU 利用率较高，于是我们可以通过 `top` 命令，找到 CPU 利用率较高的进程，在通过 `Shift+H` 找到进程中 CPU 消耗过高的线程，这里有两种情况。

CPU 利用率过高的线程一直是同一个，说明程序中存在线程长期占用 CPU 没有释放的情况，这种情况直接通过 `jstack` 获得线程的 Dump 日志，定位到线程日志后就可以找到问题的代码。

CPU 利用率过高的线程 id 不断变化，说明线程创建过多，需要挑选几个线程 id，通过 `jstack` 去线程 dump 日志中排查。

最后有可能定位的结果是程序正常，只是在 CPU 飙高的那一刻，用户访问量较大，导致系统资源不够。

以上就是我对这个问题的理解！

## 结尾

---

从这个问题来看，面试官主要考察实操能力，以及解决问题的思路。

如果你没有实操过，但是你知道导致 CPU 飙高这个现象的原因，并说出你的解决思路，通过面试是没问题的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，

如果你在面试的时候遇到了一些比较刁钻也奇葩的问题，欢迎在评论区给我留言，我是 Mic。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## lock 和 synchronized 区别

---

今天来分享一道阿里一面的面试题，“lock 和 synchronized 的区别”。

对于这个问题，看看普通人和高手的回答！

### 普通人

---

嗯，lock 是 J.U.C 包里面提供的锁，synchronized 是 Java 中的同步关键字。

他们都可以实现多线程对共享资源访问的线程安全性。

## 高手

下面我从 3 个方面来回答

从功能角度来看，**Lock** 和 **Synchronized** 都是 Java 中用来解决线程安全问题的工具。

从特性来看，

**Synchronized** 是 Java 中的同步关键字，**Lock** 是 J.U.C 包中提供的接口，这个接口有很多实现类，其中就包括 **ReentrantLock** 重入锁

**Synchronized** 可以通过两种方式来控制锁的粒度，（贴图）



```
//修饰在方法层面
public synchronized void sync(){
}

Object lock=new Object();
//修饰在代码块
public void sync(){
    synchronized(lock){
    }
}
```

一种是把 **synchronized** 关键字修饰在方法层面，

另一种是修饰在代码块上，并且我们可以通过 **Synchronized** 加锁对象的声明周期来控制锁的作用范围，比如锁对象是静态对象或者类对象，那么这个锁就是全局锁。

如果锁对象是普通实例对象，那这个锁的范围取决于这个实例的声明周期。

Lock 锁的粒度是通过它里面提供的 `lock()` 和 `unlock()` 方法决定的（贴图），包裹在这两个方法之间的代码能够保证线程安全性。而锁的作用域取决于 Lock 实例的生命周期。



Lock 比 Synchronized 的灵活性更高，Lock 可以自主决定什么时候加锁，什么时候释放锁，只需要调用 `lock()` 和 `unlock()` 这两个方法就行，同时 Lock 还提供了非阻塞的竞争锁方法 `tryLock()` 方法，这个方法通过返回 `true/false` 来告诉当前线程是否已经有其他线程正在使用锁。

Synchronized 由于是关键字，所以它无法实现非阻塞竞争锁的方法，另外，Synchronized 锁的释放是被动的，就是当 Synchronized 同步代码块执行完以后或者代码出现异常时才会释放。

Lock 提供了公平锁和非公平锁的机制，公平锁是指线程竞争锁资源时，如果已经有其他线程正在排队等待锁释放，那么当前竞争锁资源的线程无法插队。而非公平锁，就是不管是否有线程在排队等待锁，它都会尝试去竞争一次锁。Synchronized 只提供了一种非公平锁的实现。

从性能方面来看，**Synchronized** 和 **Lock** 在性能方面相差不大，在实现上会有一些区别，**Synchronized** 引入了偏向锁、轻量级锁、重量级锁以及锁升级的方式来优化加锁的性能，而 **Lock** 中则用到了自旋锁的方式来实现性能优化。

以上就是我对于这个问题的理解。

## 结尾

---

这个问题主要是考察求职对并发基础能力的掌握。

在实际应用中，线程以及线程安全性是非常重要的和常见的功能，对于这部分内容如果理解不够深刻，很容易造成生产级别的故障。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

如果在面试过程中遇到了比较刁钻和奇葩的问题，欢迎评论区给我留言！

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 线程池如何知道一个线程的任务已经执行完成

---

一个小伙伴私信了一个小米的面试题，问题是：“线程池如何知道一个线程的任务已经执行完成”？

说实话，这个问题确实很刁钻，毕竟像很多工作 5 年多的小伙伴，连线程池都没用过，怎么可能回答出来这个问题呢？

下面我们来看看普通人和高手遇到这个问题的回答思路。

### 普通人

---

嗯..（临场发挥吧）

### 高手

---

好的，我会从两个方面来回答。

在线程池内部，当我们把一个任务丢给线程池去执行，线程池会调度工作线程来执行这个任务的 **run** 方法，**run** 方法正常结束，也就意味着任务完成了。

所以线程池中的工作线程是通过同步调用任务的 `run()` 方法并且等待 `run` 方法返回后，再去统计任务的完成数量。

如果想在线程池外部去获得线程池内部任务的执行状态，有几种方法可以实现。

线程池提供了一个 `isTerminated()` 方法，可以判断线程池的运行状态，我们可以循环判断 `isTerminated()` 方法的返回结果来了解线程池的运行状态，一旦线程池的运行状态是 `Terminated`，意味着线程池中的所有任务都已经执行完了。想要通过这个方法获取状态的前提是，程序中主动调用了线程池的 `shutdown()` 方法。在实际业务中，一般不会主动去关闭线程池，因此这个方法在实用性和灵活性方面都不是很好。

在线程池中，有一个 `submit()` 方法，它提供了一个 `Future` 的返回值，我们通过 `Future.get()` 方法来获得任务的执行结果，当线程池中的任务没执行完之前，`future.get()` 方法会一直阻塞，直到任务执行结束。因此，只要 `future.get()` 方法正常返回，也就意味着传入到线程池中的任务已经执行完成了！

可以引入一个 `CountDownLatch` 计数器，它可以通过初始化指定一个计数器进行倒计时，其中有两个方法分别是 `await()` 阻塞线程，以及 `countDown()` 进行倒计时，一旦倒计时归零，所以被阻塞在 `await()` 方法的线程都会被释放。

基于这样的原理，我们可以定义一个 `CountDownLatch` 对象并且计数器为 1，接着在线程池代码块后面调用 `await()` 方法阻塞主线程，然后，当传入到线程池中的任务执行完成后，调用 `countDown()` 方法表示任务执行结束。

最后，计数器归零 0，唤醒阻塞在 `await()` 方法的线程。

```
public static void main(String[] args) throws InterruptedException {
    ExecutorService executorService= Executors.newFixedThreadPool(10);
    CountDownLatch countDownLatch=new CountDownLatch(1);
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            //开始执行任务
            try {
                Thread.sleep(3000); //模拟任务执行时间
                countDownLatch.countDown(); //任务执行结束后，计数器减1
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    //阻塞main线程| 当任务执行结束调用countDown()方法使得计数器归零后，唤醒主线程。
    countDownLatch.await();
    executorService.shutdown();
}
```

基于这个问题，我简单总结一下，不管是线程池内部还是外部，要想知道线程是否执行结束，我们必须获取线程执行结束后的状态，而线程本身没有返回值，所以只能通过阻塞-唤醒的方式来实现，`future.get` 和 `CountDownLatch` 都是这样一个原理。

以上就是我对于这个问题的回答！

## 结尾

大家可以站在面试官的角度来看高手的回答，

不难发现，高手对于技术基础的掌握程度，是非常深和全面的。这也是面试官考察这类问题的目的。

因此，Mic 提醒大家，除了日常的 **CRUD** 以外，抽出部分时间去做技术深度和广度的学习是非常有必要的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

# HashMap 是怎么解决哈希冲突的？

---

常用数据结构基本上是面试必问的问题，比如 HashMap、LinkedList、ConcurrentHashMap 等。

关于 HashMap，有个学员私信了我一个面试题说：“HashMap 是怎么解决哈希冲突的？”

关于这个问题，我们来模拟一下普通人和高手对于这个问题的回答。

## 普通人

---

嗯....HashMap 我好久之前看过它的源码，我记得好像是通过链表来解决的！

## 高手

---

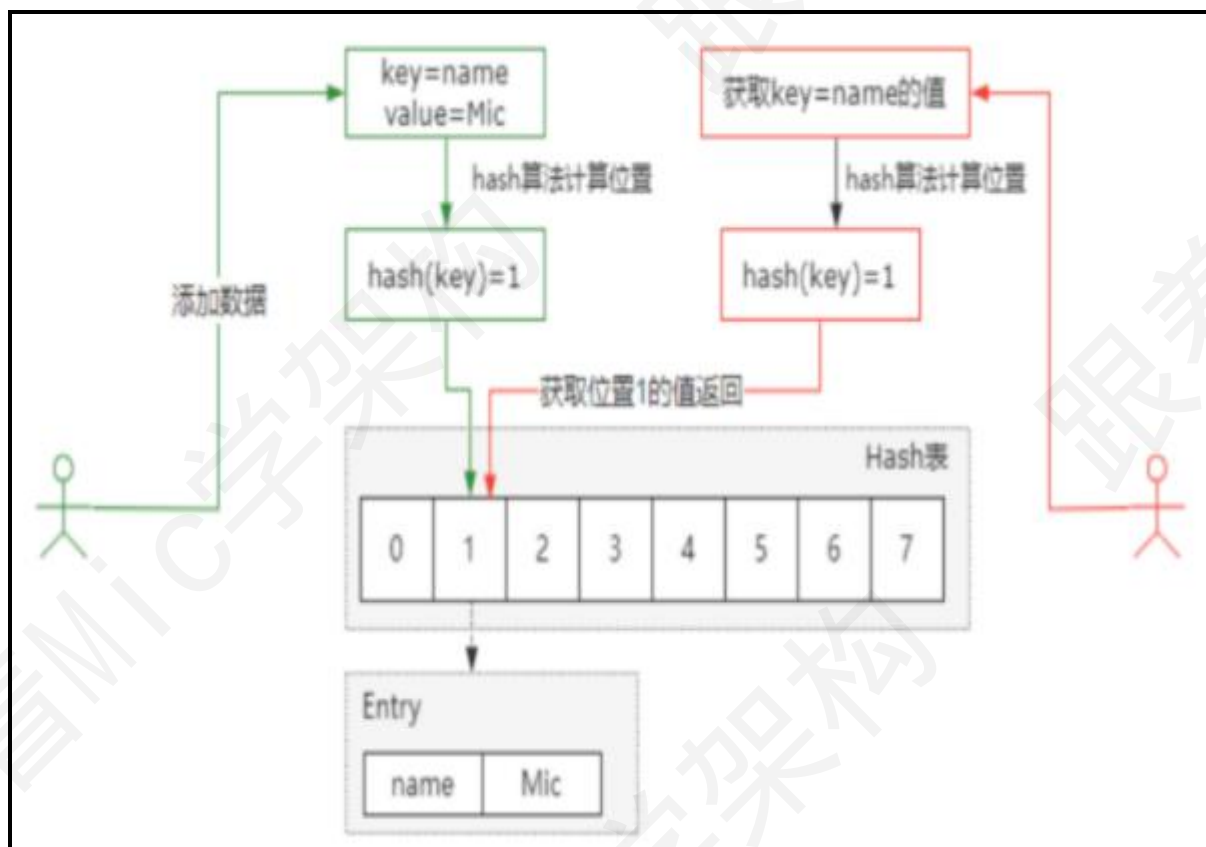
嗯，这个问题我从三个方面来回答。

要了解 Hash 冲突，那首先我们要先了解 Hash 算法和 Hash 表。

Hash 算法，就是把任意长度的输入，通过散列算法，变成固定长度的输出，这个输出结果是散列值。

Hash 表又叫做“散列表”，它是通过 key 直接访问在内存存储位置的数据结构，在具体实现上，我们通过 hash 函数把 key 映射到表中的某个位置，来获取这个位置的数据，从而加快查找速度。



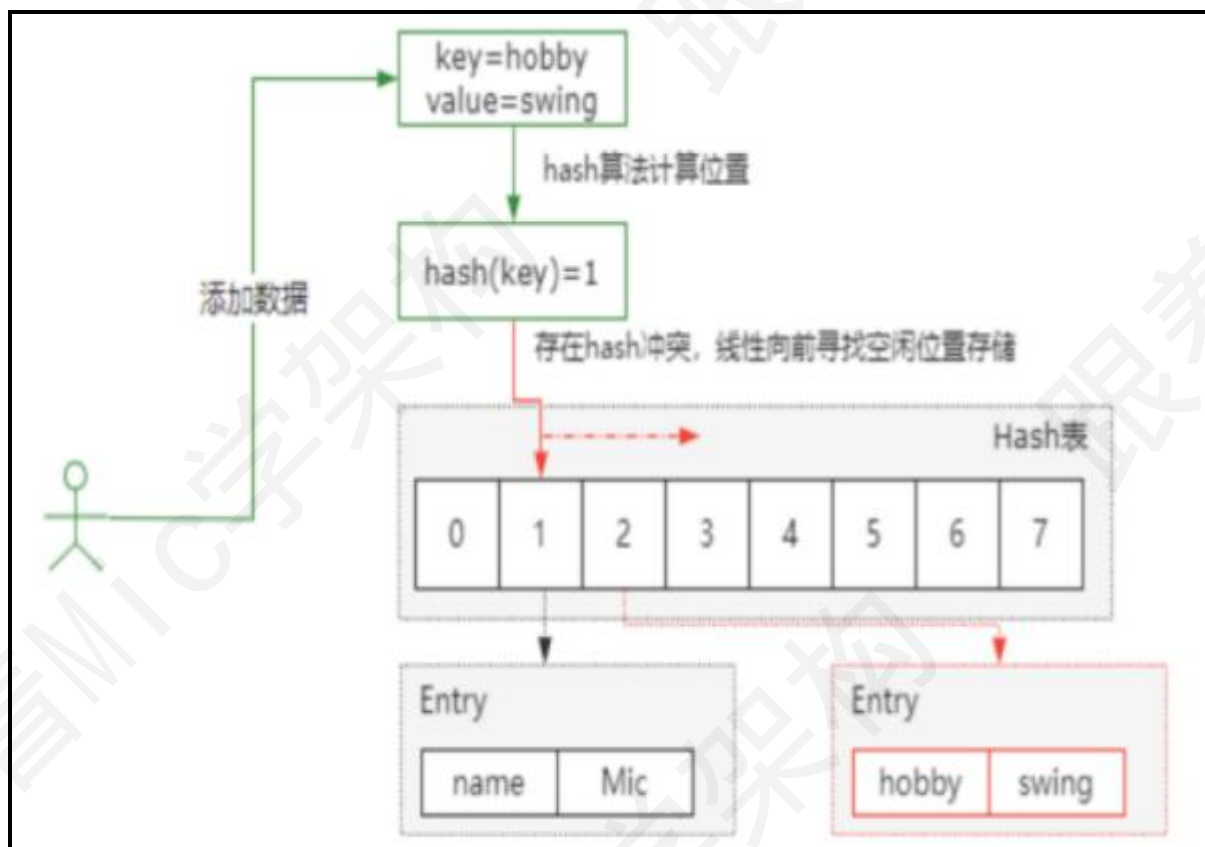


所谓 **hash 冲突**，是由于哈希算法被计算的数据是无限的，而计算后的结果范围有限，所以总会存在不同的数据经过计算后得到的值相同，这就是哈希冲突。

通常解决 **hash 冲突** 的方法有 4 种。

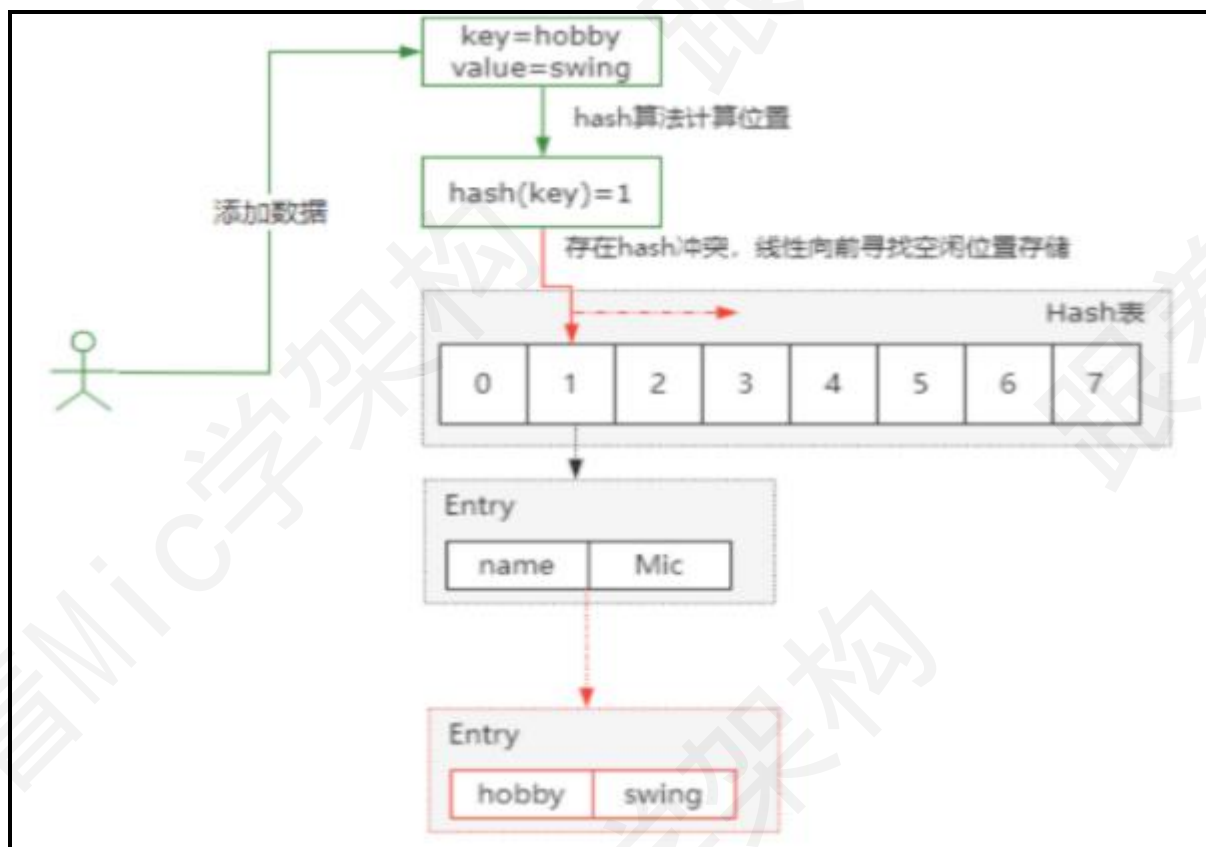
开放定址法，也称为线性探测法，就是从发生冲突的那个位置开始，按照一定的次序从 **hash** 表中找到一个空闲的位置，然后把发生冲突的元素存入到这个空闲位置中。**ThreadLocal** 就用到了线性探测法来解决 **hash 冲突** 的。

向这样一种情况，在 **hash** 表索引 1 的位置存了一个 **key=name**，当再次添加 **key=hobby** 时，**hash** 计算得到的索引也是 1，这个就是 **hash 冲突**。而开放定址法，就是按顺序向前找到一个空闲的位置来存储冲突的 **key**。



链式寻址法，这是一种非常常见的方法，简单理解就是把存在 hash 冲突的 key，以单向链表的方式来存储，比如 HashMap 就是采用链式寻址法来实现的。

向这样一种情况，存在冲突的 key 直接以单向链表的方式进行存储。



再 hash 法，就是当通过某个 hash 函数计算的 key 存在冲突时，再用另外一个 hash 函数对这个 key 做 hash，一直运算直到不再产生冲突。这种方式会增加计算时间，性能影响较大。

建立公共溢出区，就是把 hash 表分为基本表和溢出表两个部分，凡事存在冲突的元素，一律放入到溢出表中。

HashMap 在 JDK1.8 版本中，通过链式寻址法+红黑树的方式来解决 hash 冲突问题，其中红黑树是为了优化 Hash 表链表过长导致时间复杂度增加的问题。当链表长度大于 8 并且 hash 表的容量大于 64 的时候，再向链表中添加元素就会触发转化。

以上就是我对这个问题的理解！

## 结尾

这道面试题主要考察 Java 基础，面向的范围是工作 1 到 5 年甚至 5 年以上。

因为集合类的对象在项目中使用频率较高，如果对集合理解不够深刻，容易在项目中制造隐藏的 BUG。

所以，再强调一下，面试的时候，基础是很重要的考核项！！

本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 什么叫做阻塞队列的有界和无界

---

昨天一个 3 年 Java 经验的小伙伴私信我，他说现在面试怎么这么难啊！

我只是面试一个业务开发，他们竟然问我：什么叫阻塞队列的有界和无界。现在面试也太卷了吧！

如果你也遇到过类似问题，那我们来看看普通人和高手的回答吧！

### 普通人

---

有界队列就是说队列中的元素个数是有限制的，而无界对接表示队列中的元素个数没有限制！嗯！！！！

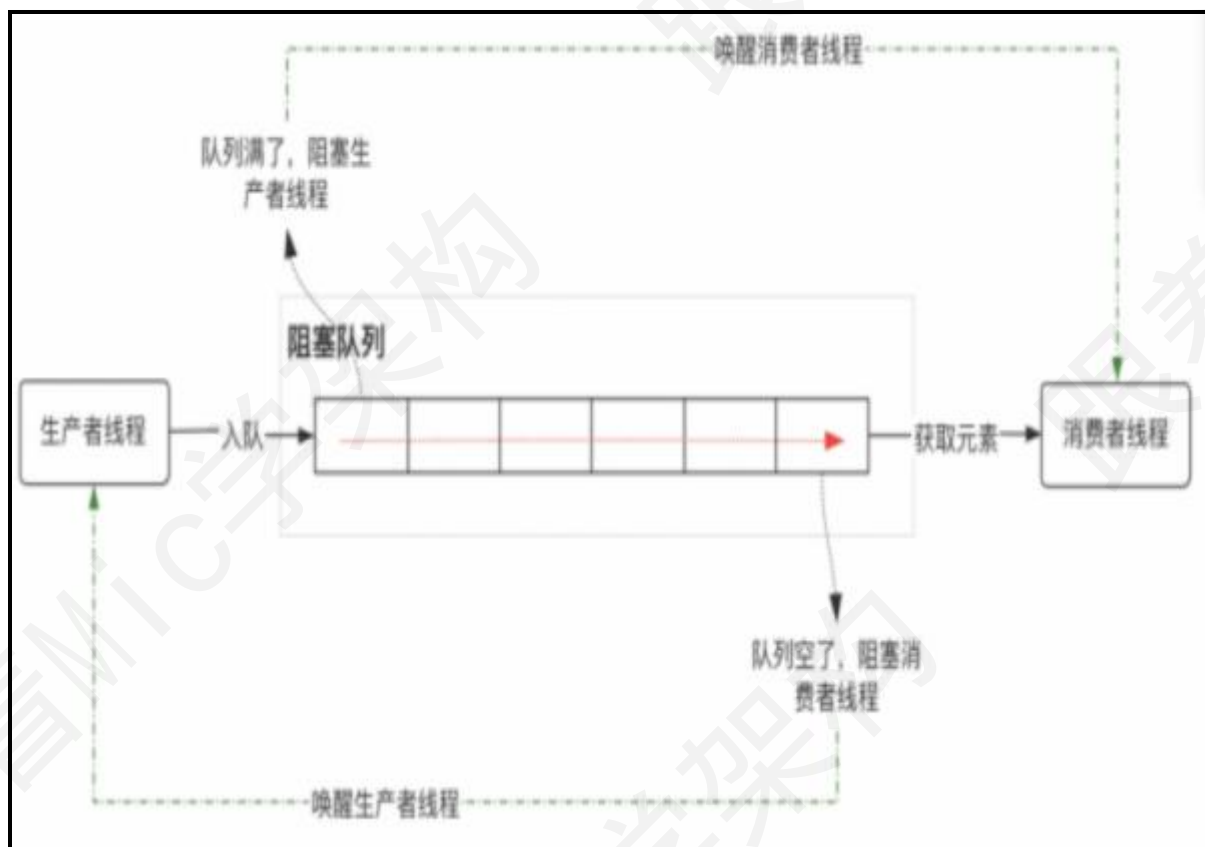
### 高手

---

，阻塞队列，是一种特殊的队列，它在普通队列的基础上提供了两个附加功能

当队列为空的时候，获取队列中元素的消费者线程会被阻塞，同时唤醒生产者线程。

当队列满了的时候，向队列中添加元素的生产者线程被阻塞，同时唤醒消费者线程。



其中，阻塞队列中能够容纳的元素个数，通常情况下是有界的，比如我们实例化一个 `ArrayBlockingList`，可以在构造方法中传入一个整形的数字，表示这个基于数组的阻塞队列中能够容纳的元素个数。这种就是有界队列。

而无界队列，就是没有设置固定大小的队列，不过它并不是像我们理解的那种元素没有任何限制，而是它的元素存储量很大，像 `LinkedBlockingQueue`，它的默认队列长度是 `Integer.Max_Value`，所以我们感知不到它的长度限制。

无界队列存在比较大的潜在风险，如果在并发量较大的情况下，线程池中可以几乎无限制的添加任务，容易导致内存溢出的问题！

以上就是我对这个问题的理解！

## 结尾

阻塞队列在生产者消费者模型的场景中使用频率比较高，比较典型的就是在线程池中，通过阻塞队列来实现线程任务的生产和消费功能。

基于阻塞队列实现的生产者消费者模型比较适合用在异步化性能提升的场景，以及做并发流量缓冲类的场景中！

在很多开源中间件中都可以看到这种模型的使用，比如在 **Zookeeper** 源码中就大量用到了阻塞队列实现的生产者消费者模型。

OK，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 **Java** 程序员，咱们下期再见。

## Dubbo 的服务请求失败怎么处理？

今天分享的面试题，几乎是 90% 以上的互联网公司都会问到的问题。

“Dubbo 的服务请求失败怎么处理”？

对于这个问题，我们来看一下普通人和高手的回答。

### 普通人

嗯...我记得，Dubbo 请求处理失败以后，好像是会重试。嗯！

### 高手

Dubbo 是一个 **RPC** 框架，它为我们的应用提供了远程通信能力的封装，同时，Dubbo 在 **RPC** 通信的基础上，逐步在向一个生态在演进，它涵盖了服务注册、动态路由、容错、服务降级、负载均衡等能力，基本上在微服务架构下面临的问题，Dubbo 都可以解决。

而对于 Dubbo 服务请求失败的场景，默认提供了重试的容错机制，也就是说，如果基于 Dubbo 进行服务间通信出现异常，服务消费者会对服务提供者集群中其他的节点发起重试，确保这次请求成功，默认的额外重试次数是 2 次。

除此之外，Dubbo 还提供了更多的容错策略，我们可以根据不同的业务场景来进行选择。

快速失败策略，服务消费者只发起一次请求，如果请求失败，就直接把错误抛出去。这种比较适合在非幂等性场景中使用

失败安全策略，如果出现服务通信异常，直接把这个异常吞掉不做任何处理

失败自动恢复策略，后台记录失败请求，然后通过定时任务来对这个失败的请求进行重发。

并行调用多个服务策略，就是把这个消息广播给服务提供者集群，只要有任何一个节点返回，就表示请求执行成功。

广播调用策略，逐个调用服务提供者集群，只要集群中任何一个节点出现异常，就表示本次请求失败

要注意的是，默认基于重试策略的容错机制中，需要注意幂等性的处理，否则在事务型的操作中，容易出现多次数据变更的问题。

以上就是我对这个问题的理解！

## 结尾

---

这类的问题，并不需要去花太多时间去背，如果你对于整个技术体系有一定的了解，你就很容易想象到最基本的处理方式。

即便是你对 Dubbo 不熟悉，也能回答一两种！

OK，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

另外，我也陆续收到了很多小伙伴的面试题，我会在后续的内容中逐步更新给大家！

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## ConcurrentHashMap 底层具体实现知道吗？实现原理是什么？

---

之前分享过一期 HashMap 的面试题，然后有个小伙伴私信我说，他遇到了一个 ConcurrentHashMap 的问题不知道怎么回答。

于是，就有了这一期的内容！！

我是 Mic，一个工作了 14 年的 Java 程序员，今天我来分享关于“ConcurrentHashMap 底层实现原理”这个问题，

看看普通人和高手是如何回答的！

### 普通人

---

嗯..ConcurrentHashMap 是用数组和链表的方式来实现的，嗯...在 JDK1.8 里面还引入了红黑树。

然后链表和红黑树是解决 hash 冲突的。嗯.....

## 高手

---

这个问题我从这三个方面来回答：（下面这三个点，打印在屏幕上）

ConcurrentHashMap 的整体架构

ConcurrentHashMap 的基本功能

ConcurrentHashMap 在性能方面的优化

ConcurrentHashMap 的整体架构（字幕提示）

这个是 ConcurrentHashMap 在 JDK1.8 中的存储结构，它是由数组、单向链表、红黑树组成。

当我们初始化一个 ConcurrentHashMap 实例时，默认会初始化一个长度为 16 的数组。由于 ConcurrentHashMap 它的核心仍然是 hash 表，所以必然会存在 hash 冲突问题。

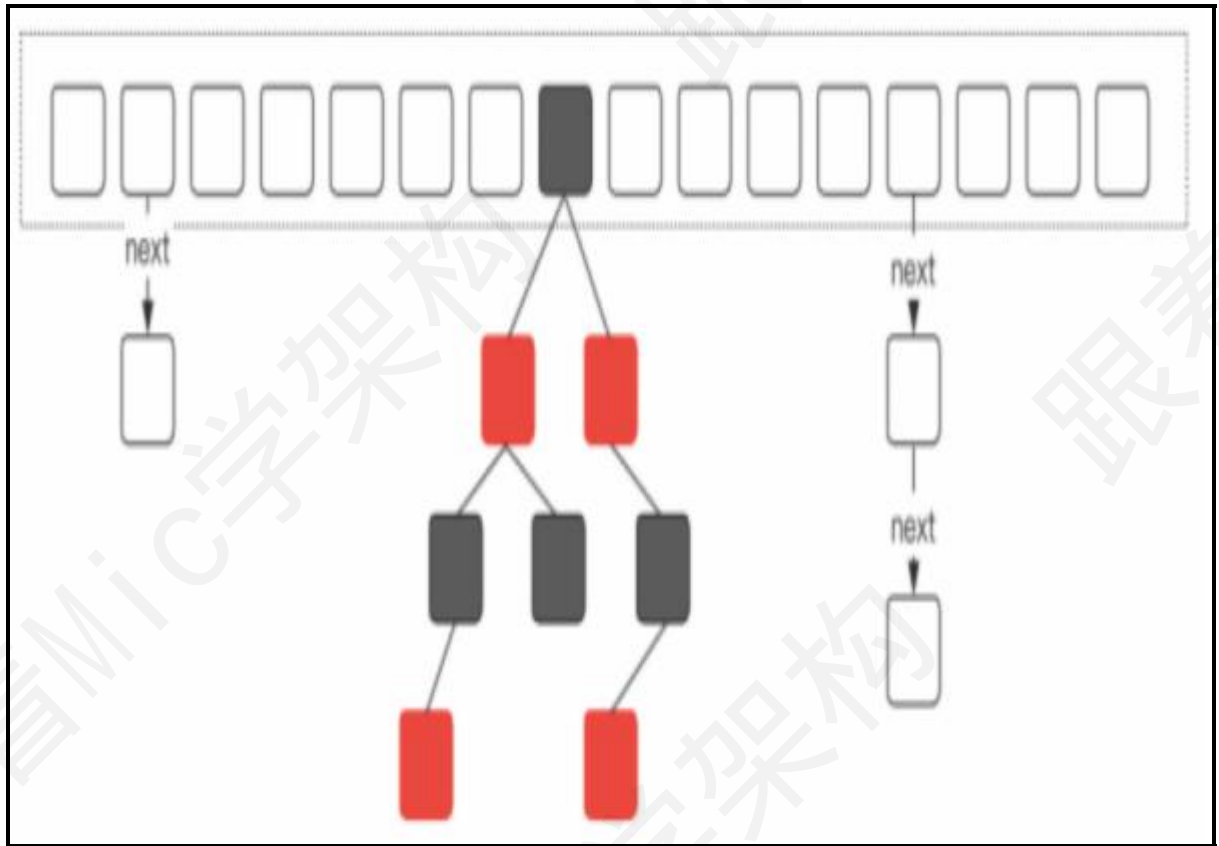
ConcurrentHashMap 采用链式寻址法来解决 hash 冲突。

当 hash 冲突比较多的时候，会造成链表长度较长，这种情况会使得 ConcurrentHashMap 中数据元素的查询复杂度变成  $O(\sim n)$ 。因此在 JDK1.8 中，引入了红黑树的机制。

当数组长度大于 64 并且链表长度大于等于 8 的时候，单项链表就会转换为红黑树。

另外，随着 ConcurrentHashMap 的动态扩容，一旦链表长度小于 8，红黑树会退化成单向链表。

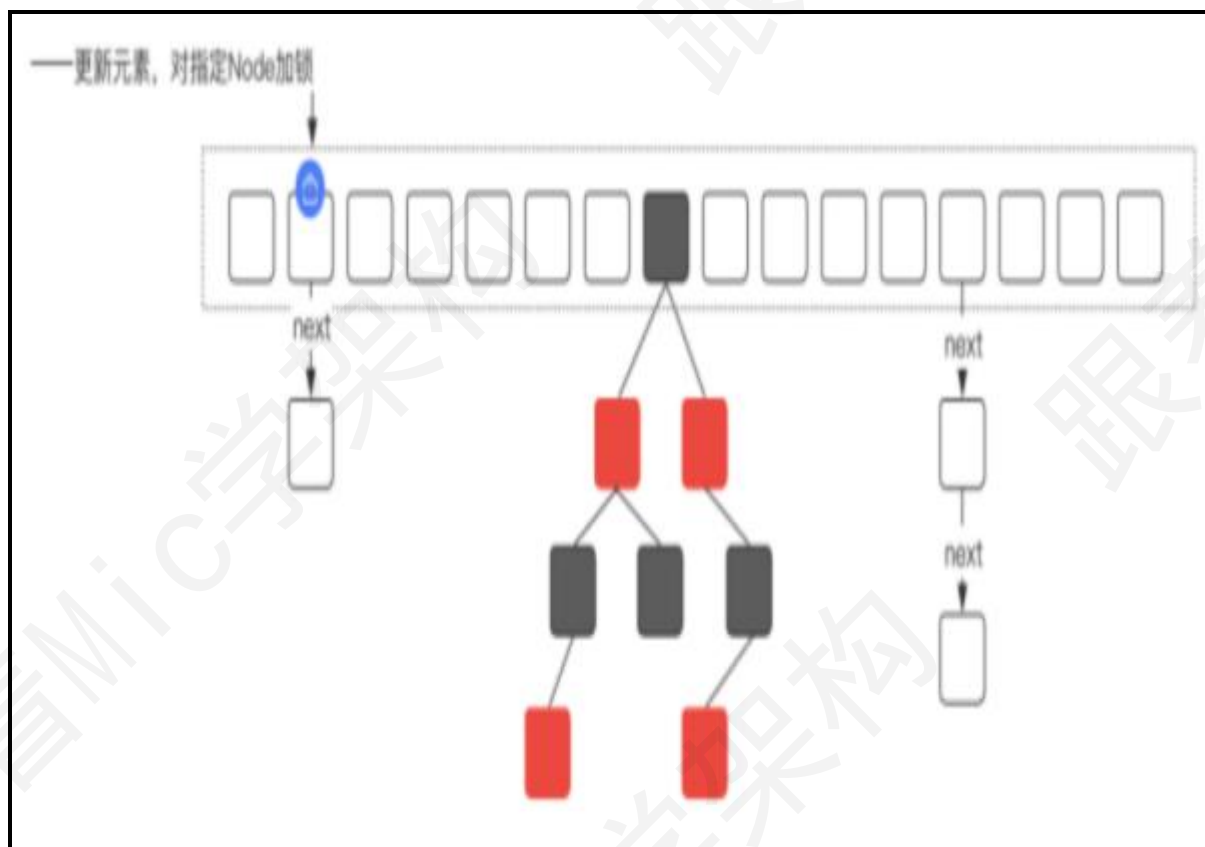




### ConcurrentHashMap 的基本功能

ConcurrentHashMap 本质上是一个 HashMap，因此功能和 HashMap 一样，但是 ConcurrentHashMap 在 HashMap 的基础上，提供了并发安全的实现。

并发安全的主要实现是通过对指定的 Node 节点加锁，来保证数据更新的安全性。



### ConcurrentHashMap 在性能方面做的优化

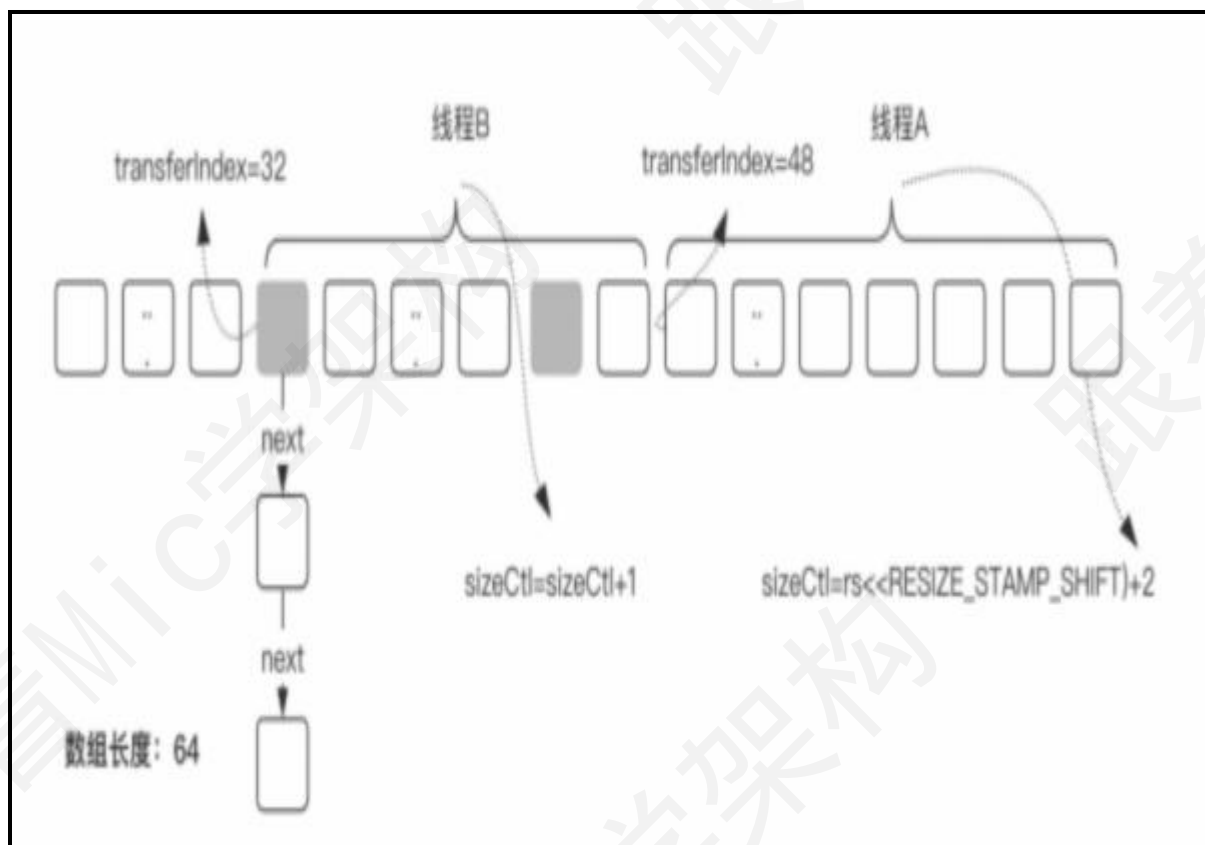
如果在并发性能和数据安全性之间做好平衡，在很多地方都有类似的设计，比如cpu的三级缓存、mysql的buffer\_pool、Synchronized的锁升级等等。

ConcurrentHashMap 也做了类似的优化，主要体现在以下几个方面：

在JDK1.8中，ConcurrentHashMap锁的粒度是数组中的某一个节点，而在JDK1.7，锁定的是Segment，锁的范围要更大，因此性能上会更低。

引入红黑树，降低了数据查询的时间复杂度，红黑树的时间复杂度是  $O(\sim \log n \sim)$ 。

当数组长度不够时，ConcurrentHashMap需要对数组进行扩容，在扩容的实现上，ConcurrentHashMap引入了多线程并发扩容的机制，简单来说就是多个线程对原始数组进行分片后，每个线程负责一个分片的数据迁移，从而提升了扩容过程中数据迁移的效率。

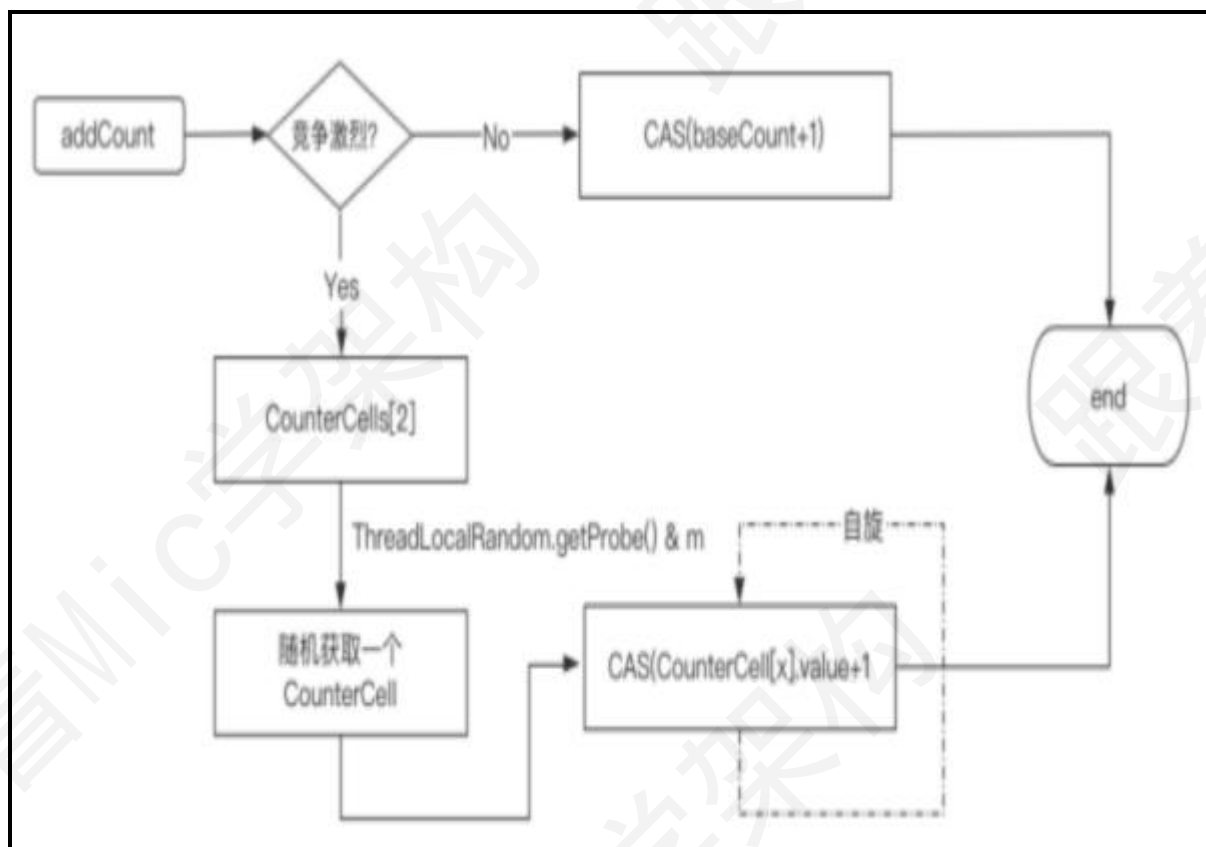


`ConcurrentHashMap` 中有一个 `size()` 方法来获取总的元素个数，而在多线程并发场景中，在保证原子性的前提下来实现元素个数的累加，性能是非常低的。

`ConcurrentHashMap` 在这个方面的优化主要体现在两个点：

当线程竞争不激烈时，直接采用 **CAS** 来实现元素个数的原子递增。

如果线程竞争激烈，使用一个数组来维护元素个数，如果要增加总的元素个数，则直接从数组中随机选择一个，再通过 **CAS** 实现原子递增。它的核心思想是引入了数组来实现对并发更新的负载。



以上就是我对这个问题的理解！

## 结尾

从高手的回答中可以看到，**ConcurrentHashMap** 里面有很多设计思想值得学习和借鉴。

比如锁粒度控制、分段锁的设计等，它们都可以应用在实际业务场景中。

很多时候大家会认为这种面试题毫无价值，当你有足够的积累之后，你会发现从这些技术底层的设计思想中能够获得

很多设计思路。

本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

另外，我也陆续收到了很多小伙伴的面试题，我会在后续的内容中逐步更新给大家！

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## b 树和 b+树的理解

数据结构与算法问题，困扰了无数的小伙伴。

很多小伙伴对数据结构与算法的认知有一个误区，认为工作中没有用到，为什么面试要问，问了能解决实际问题？

图灵奖获得者：**Niklaus Wirth** 说过：程序=数据结构+算法，也就说我们无时无刻都在和数据结构打交道。

只是作为 **Java** 开发，由于技术体系的成熟度较高，使得大部分人认为：程序应该等于框架+SQL 呀？

今天我们就来分析一道数据结构的题目：“B 树和 B+树”。

关于这个问题，我们来看看普通人和高手的回答！

## 普通人

---

嗯.我想想...嗯...Mysql 里面好像是用了 B+树来做索引的！然后...

## 高手

---

为了更清晰的解答这个问题，我打算从三个方面来回答：

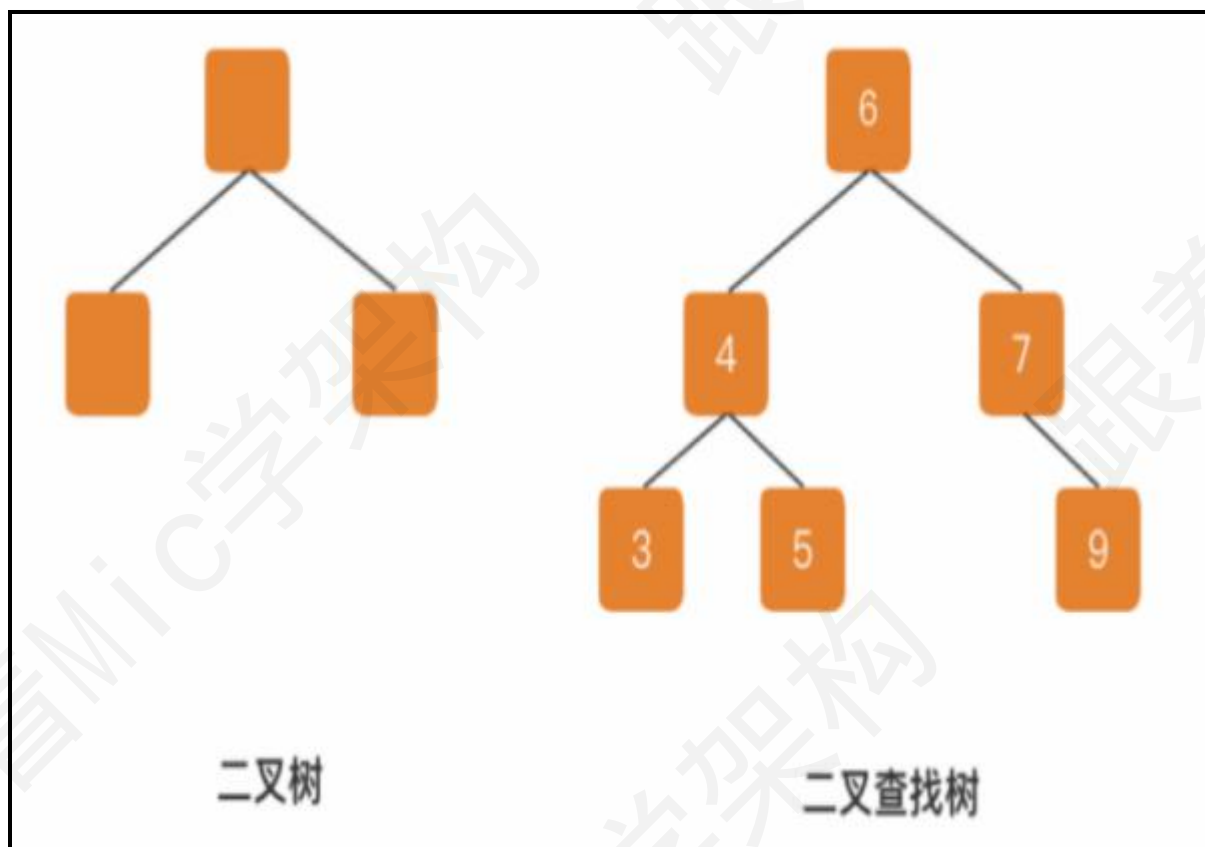
了解二叉树、AVL 树、B 树的概念

B 树和 B+树的应用场景

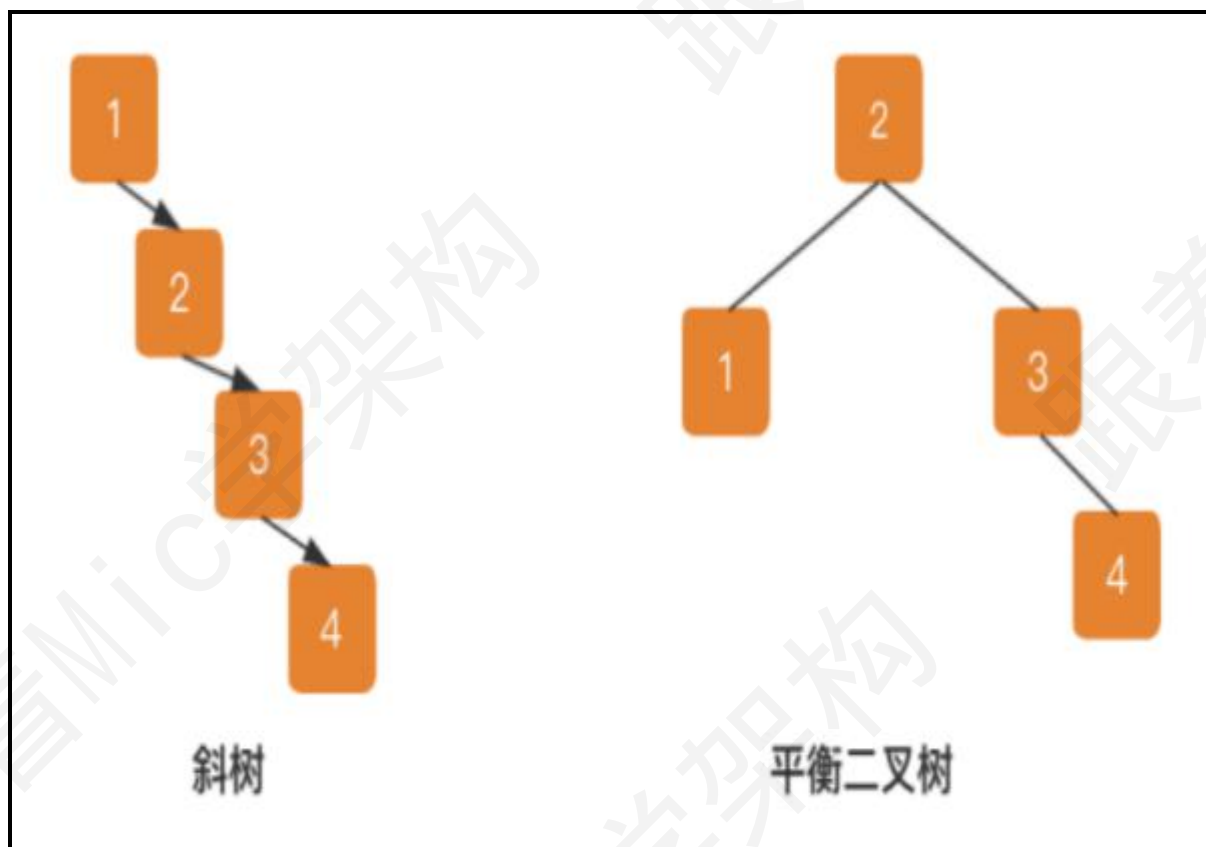
B 树是一种多路平衡查找树，为了更形象的理解，（我们来看这张图）。

二叉树，每个节点支持两个分支的树结构，相比于单向链表，多了一个分支。

二叉查找树，在二叉树的基础上增加了一个规则，左子树的所有节点的值都小于它的根节点，右子树的所有子节点都大于它的根节点。

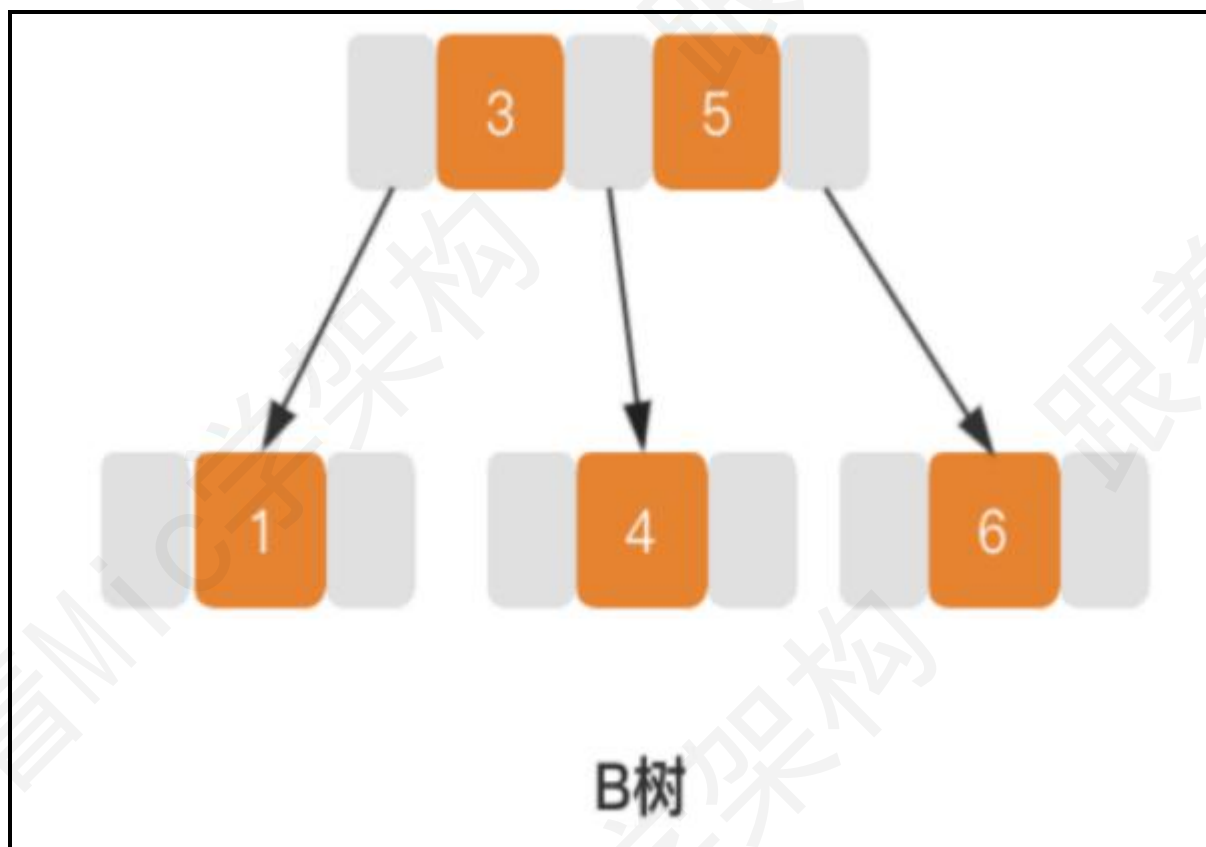


，二叉查找树会出现斜树问题，导致时间复杂度增加，因此又引入了一种平衡二叉树，它具有二叉查找树的所有特点，同时增加了一个规则：“它的左右两个子树的高度差的绝对值不超过 1”。平衡二叉树会采用左旋、右旋的方式来实现平衡。



，而 B 树是一种多路平衡查找树，它满足平衡二叉树的规则，但是它可以有多个子树，子树的数量取决于关键字的数量，比如这个图中根节点有两个关键字 3 和 5，那么它能够拥有的子路数量=关键字数+1。

因此从这个特征来看，在存储同样数据量的情况下，平衡二叉树的高度要大于 B 树。



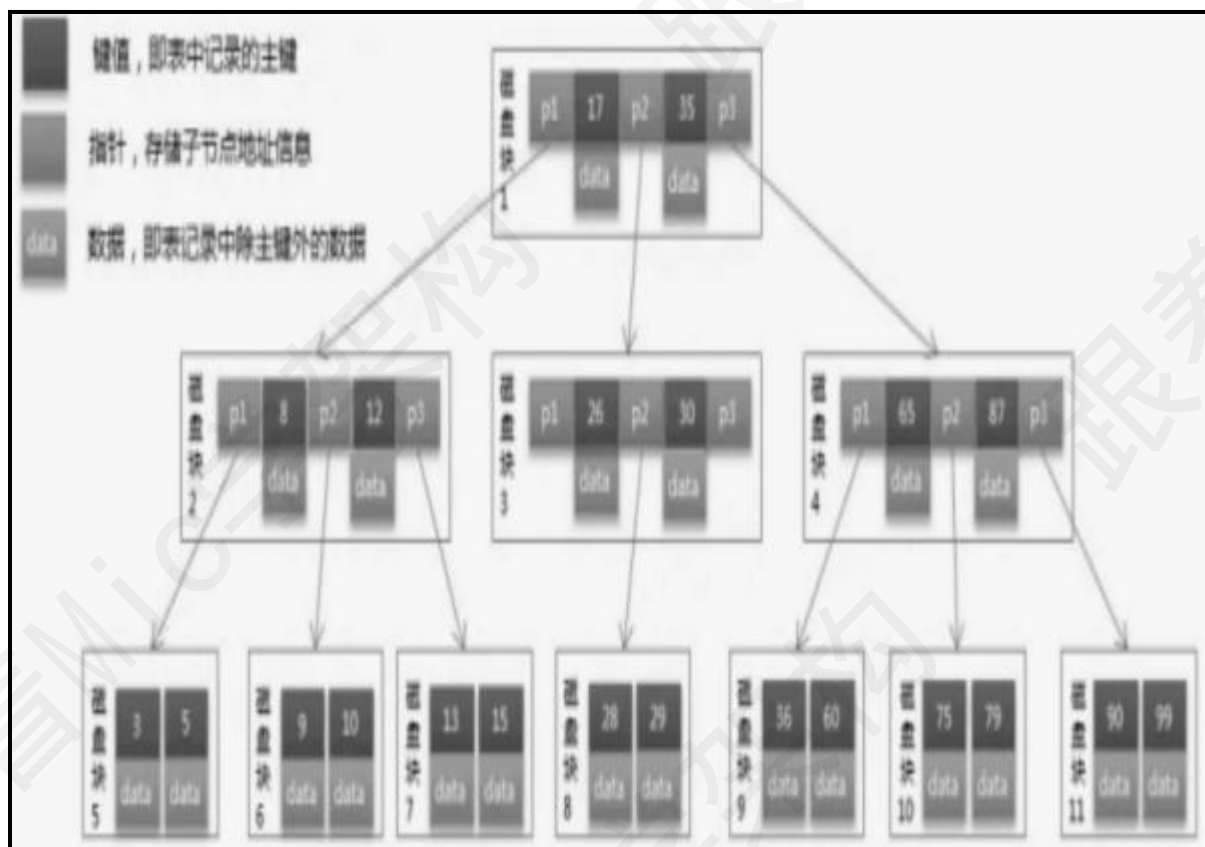
B+树，其实是在 B 树的基础上做的增强，最大的区别有两个：

B 树的数据存储在每个节点上，而 B+树中的数据是存储在叶子节点，并且通过链表的方式把叶子节点中的数据进行连接。

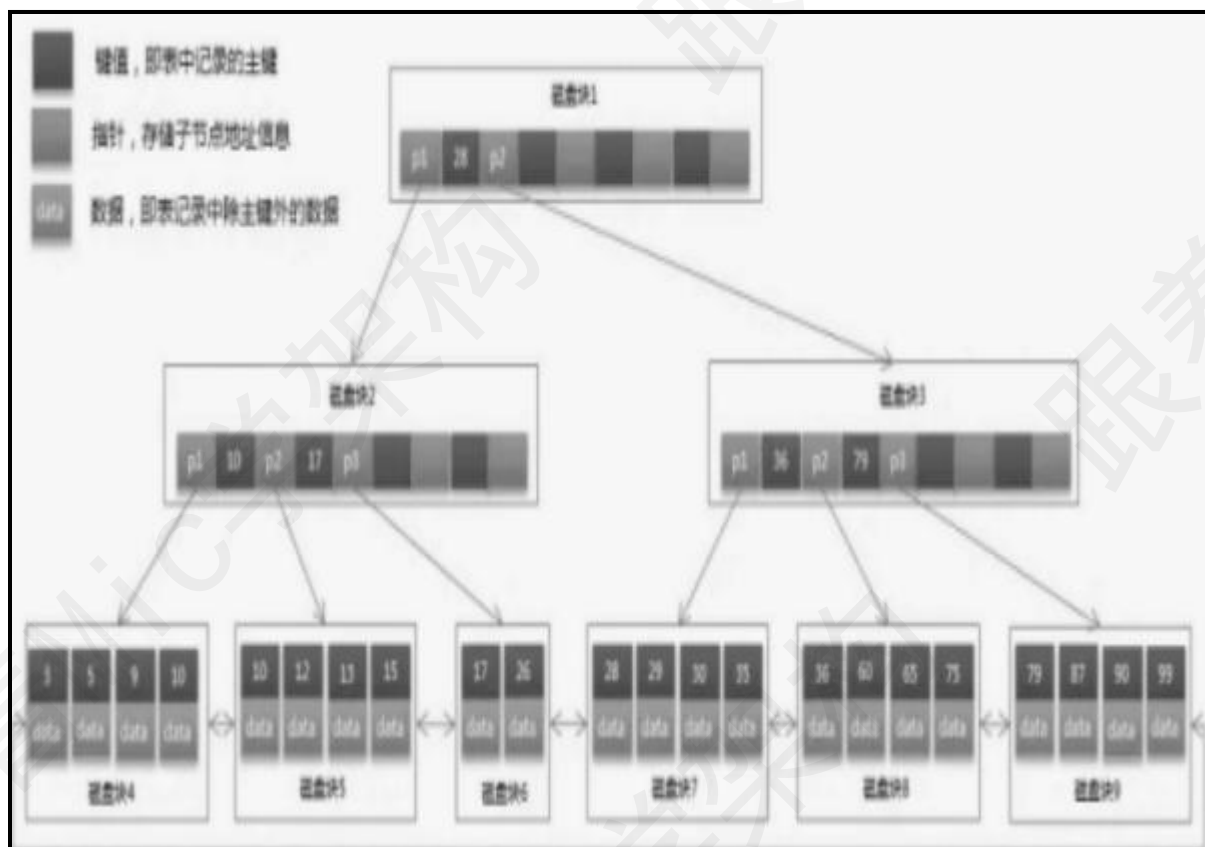
B+树的子路数量等于关键字数

（如图所示）这个是 B 树的存储结构，从 B 树上可以看到每个节点会存储数据。





(如图所示) 这个是 B+树，B+树的所有数据是存储在叶子节点，并且叶子节点的数据是用双向链表关联的。



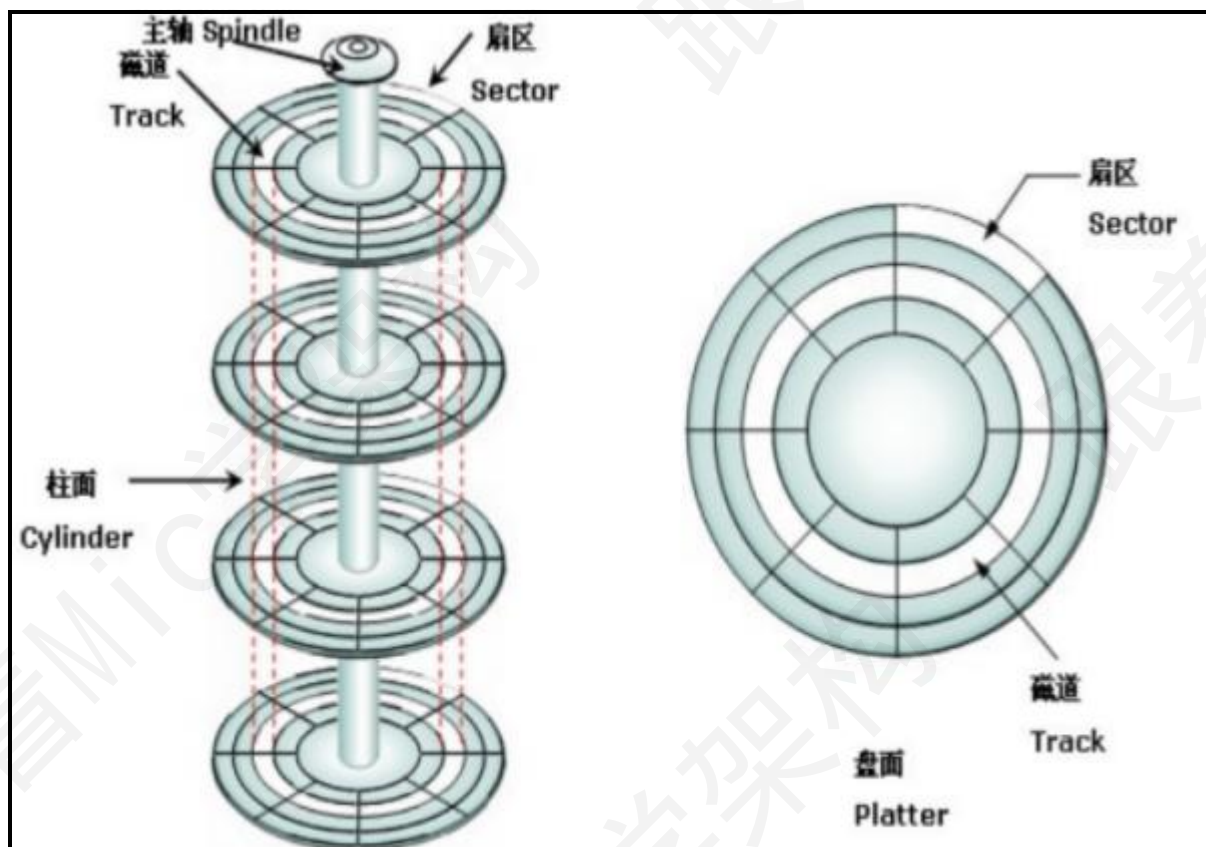
B 树和 B+树，一般都是应用在文件系统和数据库系统中，用来减少磁盘 IO 带来的性能损耗。

以 Mysql 中的 InnoDB 为例，当我们通过 `select` 语句去查询一条数据时，InnoDB 需要从磁盘上去读取数据，这个过程会涉及到磁盘 IO 以及磁盘的随机 IO（如图所示）

我们知道磁盘 IO 的性能是特别低的，特别是随机磁盘 IO。

因为，磁盘 IO 的工作原理是，首先系统会把数据逻辑地址传给磁盘，磁盘控制电路按照寻址逻辑把逻辑地址翻译成物理地址，也就是确定要读取的数据在哪个磁道，哪个扇区。

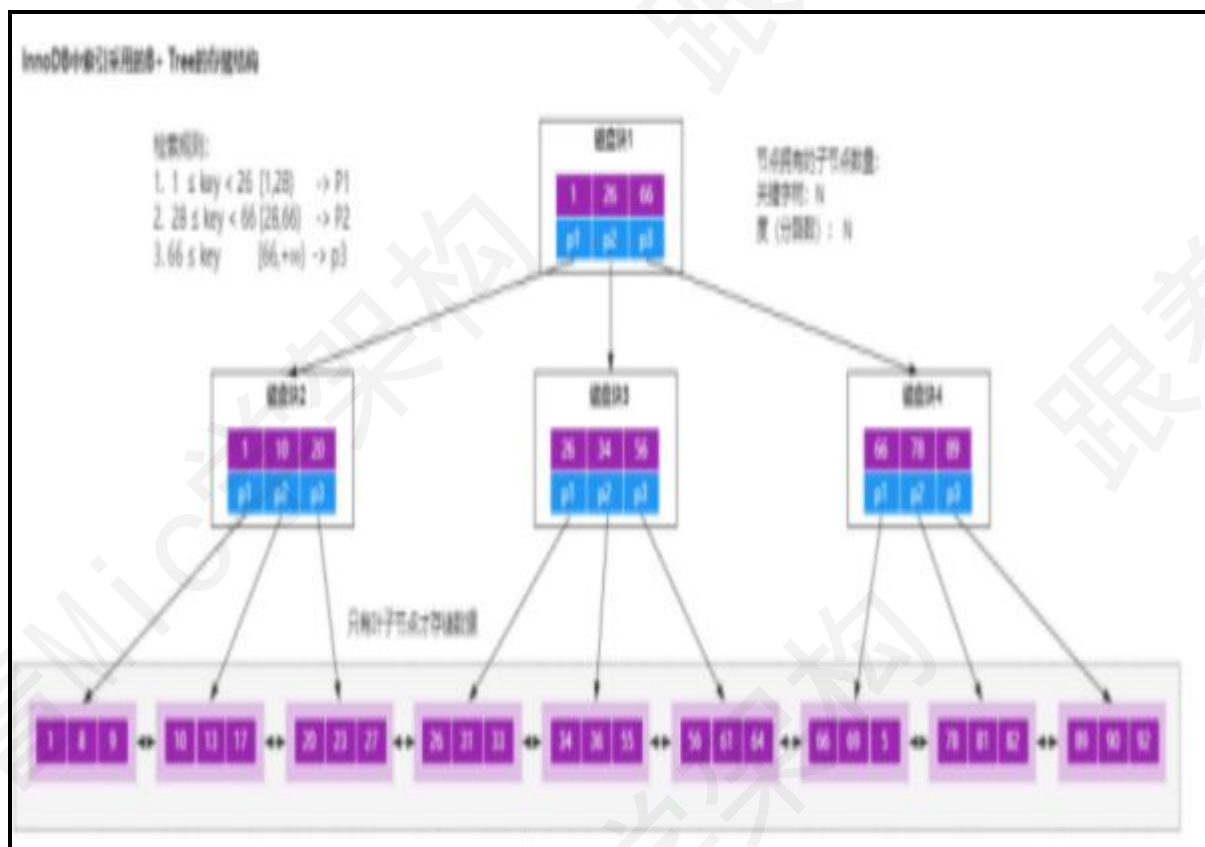
为了读取这个扇区的数据，需要把磁头放在这个扇区的上面，为了实现这一个点，磁盘会不断旋转，把目标扇区旋转到磁头下面，使得磁头找到对应的磁道，这里涉及到寻道事件以及旋转时间。



很明显，磁盘 IO 这个过程的性能开销是非常大的，特别是查询的数据量比较多的情况下。

所以在 InnoDB 中，干脆对存储在磁盘块上的数据建立一个索引，然后把索引数据以及索引列对应的磁盘地址，以 B+树的方式来存储。

如图所示，当我们需要查询目标数据的时候，根据索引从 B+树中查找目标数据即可，由于 B+树分路较多，所以只需要较少次数的磁盘 IO 就能查找到。



为什么用 B 树或者 B+树来做索引结构？原因是 AVL 树的高度要比 B 树的高度要高，而高度就意味着磁盘 IO 的数量。所以为了减少磁盘 IO 的次数，文件系统或者数据库才会采用 B 树或者 B+树。

以上就是我对 B 树和 B+树的理解！

## 结尾

数据结构在实际开发中非常常见，比如数组、链表、双向链表、红黑树、跳跃表、B 树、B+树、队列等。

在我看来，数据结构是编程中最重要的基本功之一。

学了顺序表和链表，我们就能知道查询操作比较多的场景中应该用顺序表，修改操作比较多的场景应该使用链表。

学了队列之后，就知道对于 FIFO 的场景中，应该使用队列。

学了树的结构后，会发现原来查找类的场景，还可以更进一步提升查询性能。

基本功决定大家在技术这个岗位上能够走到的高度。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

如果最近大家遇到一些场景类和方案设计类的问题，欢迎私信我，我在后续的内容中给大家做解答！

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 能谈一下 CAS 机制吗？

一个小伙伴私信我，他说遇到了一个关于 CAS 机制的问题，他以为面试官问的是 CAS 实现单点登录。

心想，这个问题我熟啊，然后就按照单点登录的思路去回答，结果面试官一直摇头。

他来和我说，到了面试结束都没明想白自己回答这么好，怎么就没有当场给我发 offer 呢？

实际上，面试官问的是并发编程中的 CAS 机制。

下面我们来看看普通人和高手对于 CAS 机制的回答吧

### 普通人

CAS，是并发编程中用来实现原子性功能的一种操作，嗯，它类似于一种乐观锁的机制，可以保证并发情况下对共享变量的值的更改的原子性。

嗯，像 `AtomicInteger` 这个类中，就用到了 CAS 机制。嗯...

### 高手

CAS 是 Java 中 `Unsafe` 类里面的方法，它的全称是 `CompareAndSwap`，比较并交换的意思。它的主要功能是能够保证在多线程环境下，对于共享变量的修改的原子性。

我来举个例子，比如说有这样一个场景，有一个成员变量 `state`，默认值是 0，

定义了一个方法 `doSomething()`，这个方法的逻辑是，判断 `state` 是否为 0，如果为 0，就修改成 1。

这个逻辑看起来没有任何问题，但是在多线程环境下，会存在原子性的问题，因为这里是一个典型的，`Read-Write` 的操作。

一般情况下，我们会在 `doSomething()` 这个方法上加同步锁来解决原子性问题。



```
public class Example {  
    private int state=0;  
    public void doSomething(){  
        if(state==0){ //多线程环境中, 存在原子性问题  
            state=1;  
            //TODO  
        }  
    }  
}
```

但是，加同步锁，会带来性能上的损耗，所以，对于这类场景，我们就可以使用 CAS 机制来进行优化

这个是优化之后的代码

在 `doSomething()` 方法中，我们调用了 `unsafe` 类中的 `compareAndSwapInt()` 方法来达到同样的目的，这个方法有四个参数，

分别是：当前对象实例、成员变量 `state` 在内存地址中的偏移量、预期值 0、期望更改之后的值 1。

CAS 机制会比较 `state` 内存地址偏移量对应的值和传入的预期值 0 是否相等，如果相等，就直接修改内存地址中 `state` 的值为 1。

否则，返回 `false`，表示修改失败，而这个过程是原子的，不会存在线程安全问题。

```
public class Example {  
  
    private volatile int state=0;  
  
    private static final Unsafe unsafe = Unsafe.getUnsafe();  
    private static final long stateOffset;  
  
    static {  
        try {  
            stateOffset = unsafe.objectFieldOffset  
                (Example.class.getDeclaredField("state"));  
        } catch (Exception ex) { throw new Error(ex); }  
    }  
  
    public void doSomething(){  
        if(unsafe.compareAndSwapInt(this, stateOffset, 0, 1)){  
            //TODO  
        }  
    }  
}
```

CompareAndSwap 是一个 native 方法，实际上它最终还是会面临同样的问题，就是先从内存地址中读取 state 的值，然后去比较，最后再修改。

这个过程不管是在什么层面上实现，都会存在原子性问题。

所以呢，CompareAndSwap 的底层实现中，在多核 CPU 环境下，会增加一个 Lock 指令对缓存或者总线加锁，从而保证比较并替换这两个指令的原子性。

CAS 主要用在并发场景中，比较典型的使用场景有两个。

第一个是 J.U.C 里面 Atomic 的原子实现，比如 AtomicInteger，AtomicLong。

第二个是实现多线程对共享资源竞争的互斥性质，比如在 AQS、ConcurrentHashMap、ConcurrentLinkedQueue 等都有用到。

以上就是我对这个问题的理解。

## 结尾

最近大家也发现了我的视频内容在高手回答部分的变化。

有些小伙伴说，你面试怎么还能带图来，明显作弊啊。



其实主要是最近很多的面试题都偏底层，而底层的内容涵盖的知识面比较广，大家平时几乎没有接触过。

所以，如果我想要去把这些知识传递给大家，就得做很多的图形和内容结构的设计，否则大家听完之后还是一脸懵逼。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 请说一下网络四元组

---

之前我一直在向大家征集一些刁钻的面试题，然后今天就收到了这样的一个问题。

“请你说一下网络四元组的理解”，他说他听到这个问题的时候，完全就懵了。

“这个是程序员应该懂的吗？你是让我去做啥？造火箭吗？”(声音的起伏)

好吧，关于这个问题，我们来看看普通人和高手的回答。

### 普通人

---

啥，刚刚你问了什么？四元组？四元组是什么东西？(内心戏+表情)

嗯.....四元组是？（要说出来）

我明白你意思，我不配做程序员，我自己滚~（内心戏+表情）

### 高手

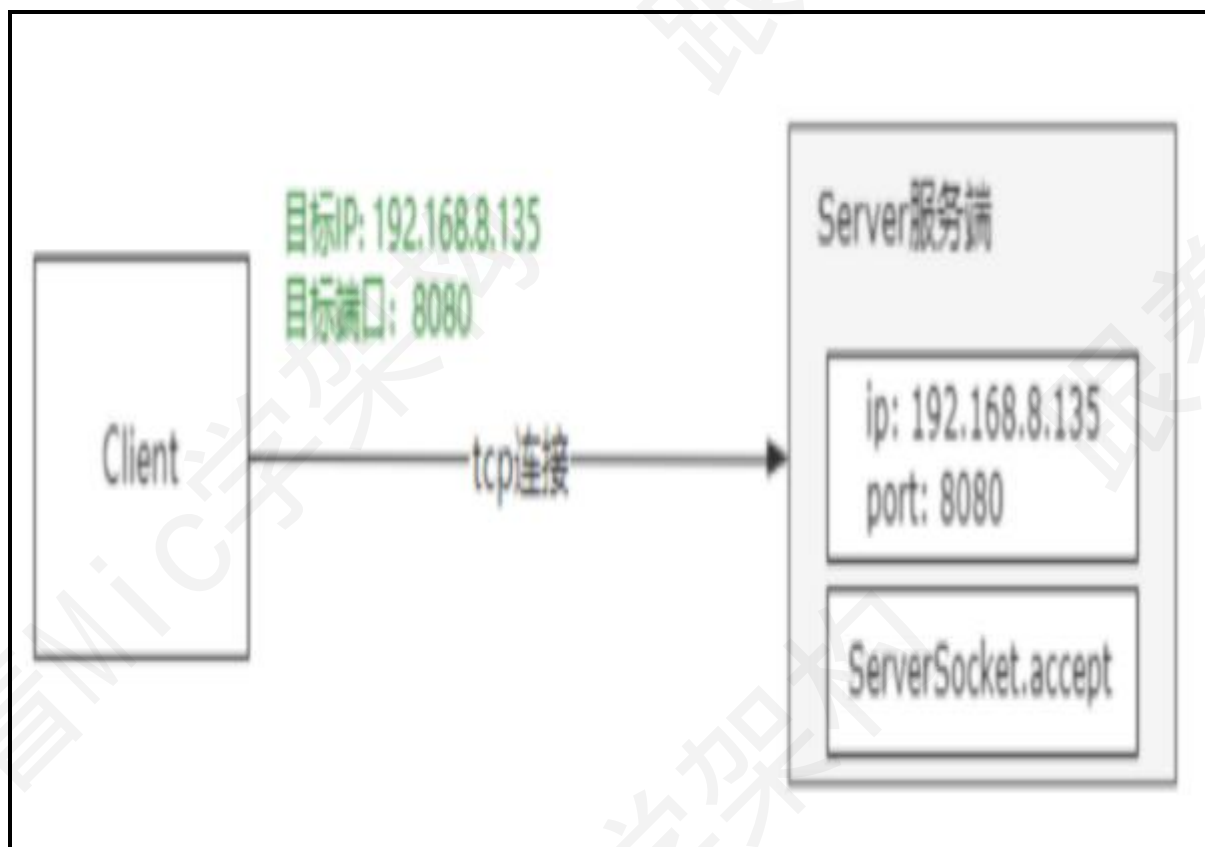
---

四元组，简单理解就是在 TCP 协议中，去确定一个客户端连接的组成要素，它包括源 IP 地址、目标 IP 地址、源端口号、目标端口号。

正常情况下，我们对于网络通信的认识可能是这样。

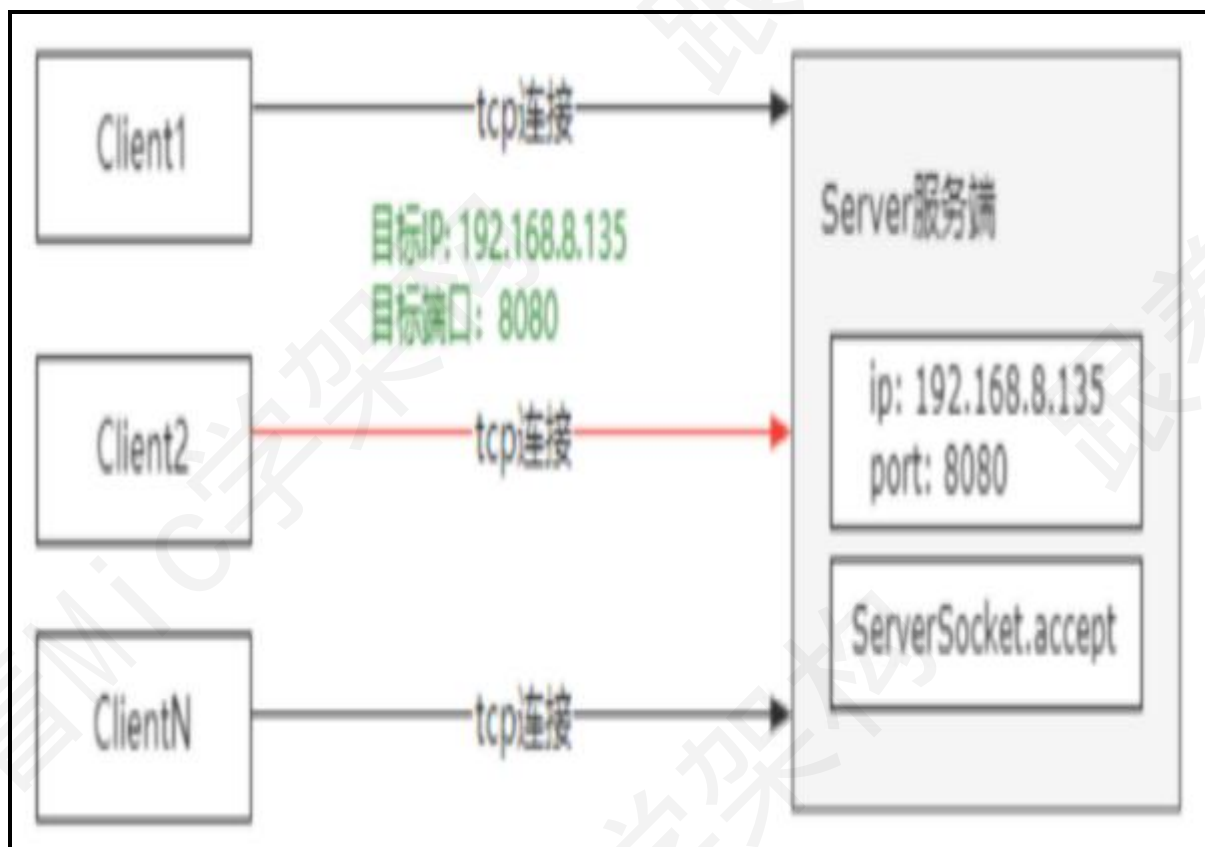
服务端通过 **ServerSocket** 建立一个对指定端口号的监听，比如 8080。客户端通过目标 ip 和端口就可以和服务端建立一个连接，然后进行数据传输。



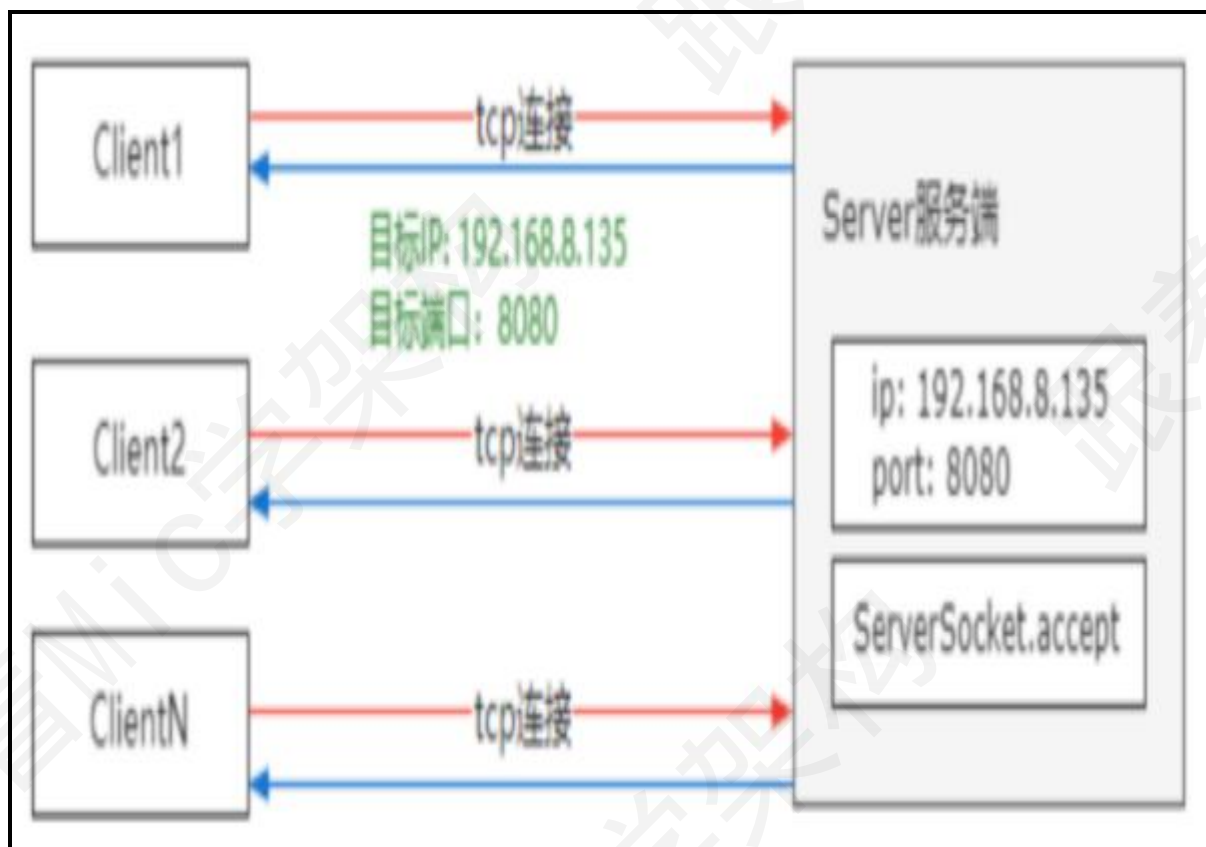


但是我们知道的是，一个 **Server** 端可以接收多个客户端的连接，比如像这种情况。

那，当多个客户端连接到服务端的时候，服务端需要去识别每一个连接。



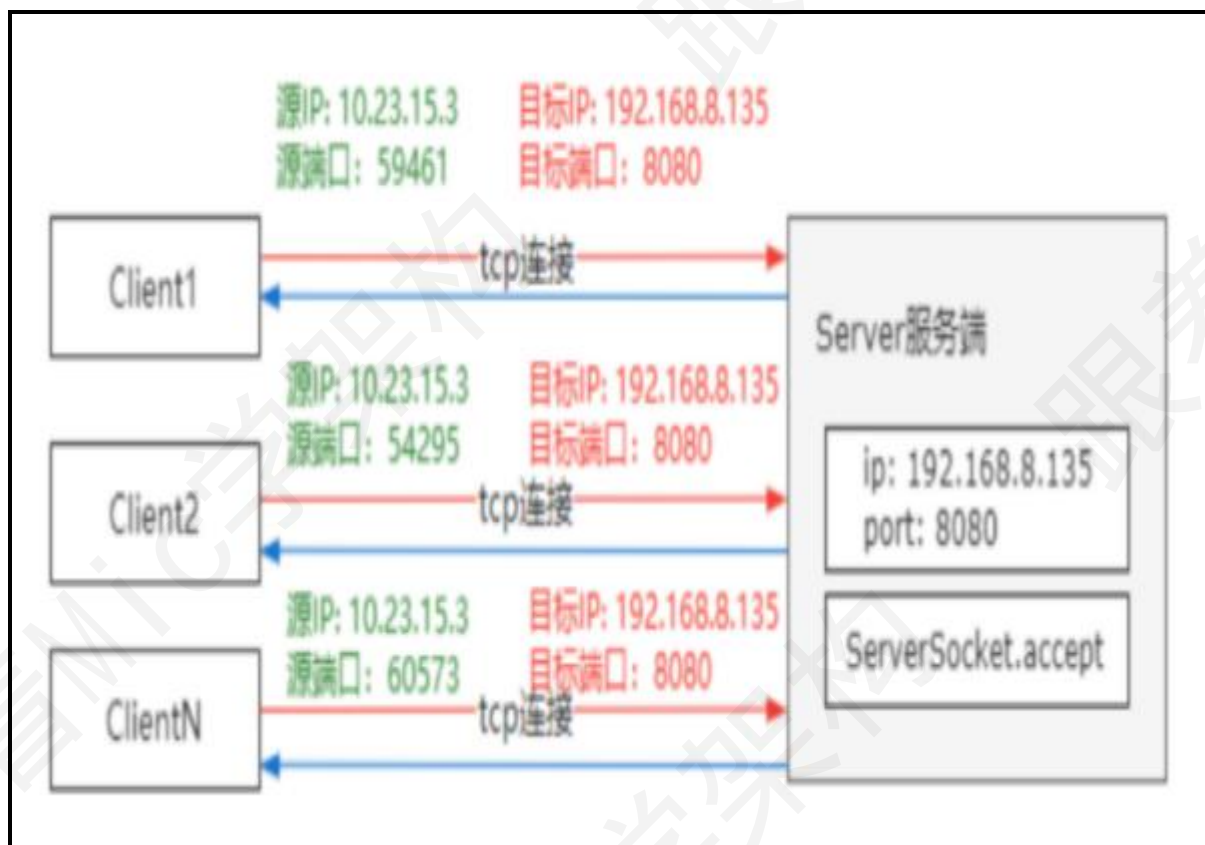
并且，TCP 是全双工协议，也就是说数据允许在连接的两个方向上同时传输，因此这里的客户端，如果是反向通信，它又变成了服务端。



所以基于这两个原因，就引入了四元组的设计，也就是说，当一个客户端和服务端建立一个 TCP 连接的时候，通过源 IP 地址、目标 IP 地址、源端口号、目标端口号来确定一个唯一的 TCP 连接。因为服务器的 IP 和端口是不变的，只要客户端的 IP 和端口彼此不同就 OK 了。

比如像这种情况，同一个客户端主机上有三个连接连到 Server 端，那么这个时候源 IP 相同，源端口号不同。此时建立的四元组就是（10.23.15.3，59461，192.168.8.135，8080）

其中，源端口号是每次建立连接的时候系统自动分配的。



以上就是我对于四元组的理解。

## 结尾

网络部分的知识，可能大家作为一个 CURD 工程师，觉得没必要去理解。

但是未来呢？至少国内没有条件允许大家做一辈子 CRUD，所以建议大家要“终局思维”来看待自己的职业规划。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 什么是服务网格？

今天继续来分享一个有趣的面试题，“什么是服务网格”？

服务网格这个概念出来很久了，从 2017 年被提出来，到 2018 年正式爆发，很多云厂商和互联网企业都在纷纷向服务网格靠拢。像蚂蚁集团、美团、百度、网易等一线互联网公司，都有服务网格的落地应用。

在我看来呢，服务网格是微服务架构的更进一步升级，它的核心目的是实现网络通信与业务逻辑的分离，使得开发人员更加专注在业务的实现上。

那么基于这个问题，我们来看看普通人和高手的回答。

## 普通人

---

嗯？

内心戏：服务网格？服务网格是什么东西？

嗯，很抱歉，这个问题我不是很清楚。

## 高手

---

服务网格，也就是 **Service Mesh**，它是专门用来处理服务通讯的基础设施层。它的主要功能是处理服务之间的通信，并且负责实现请求的可靠性传递。

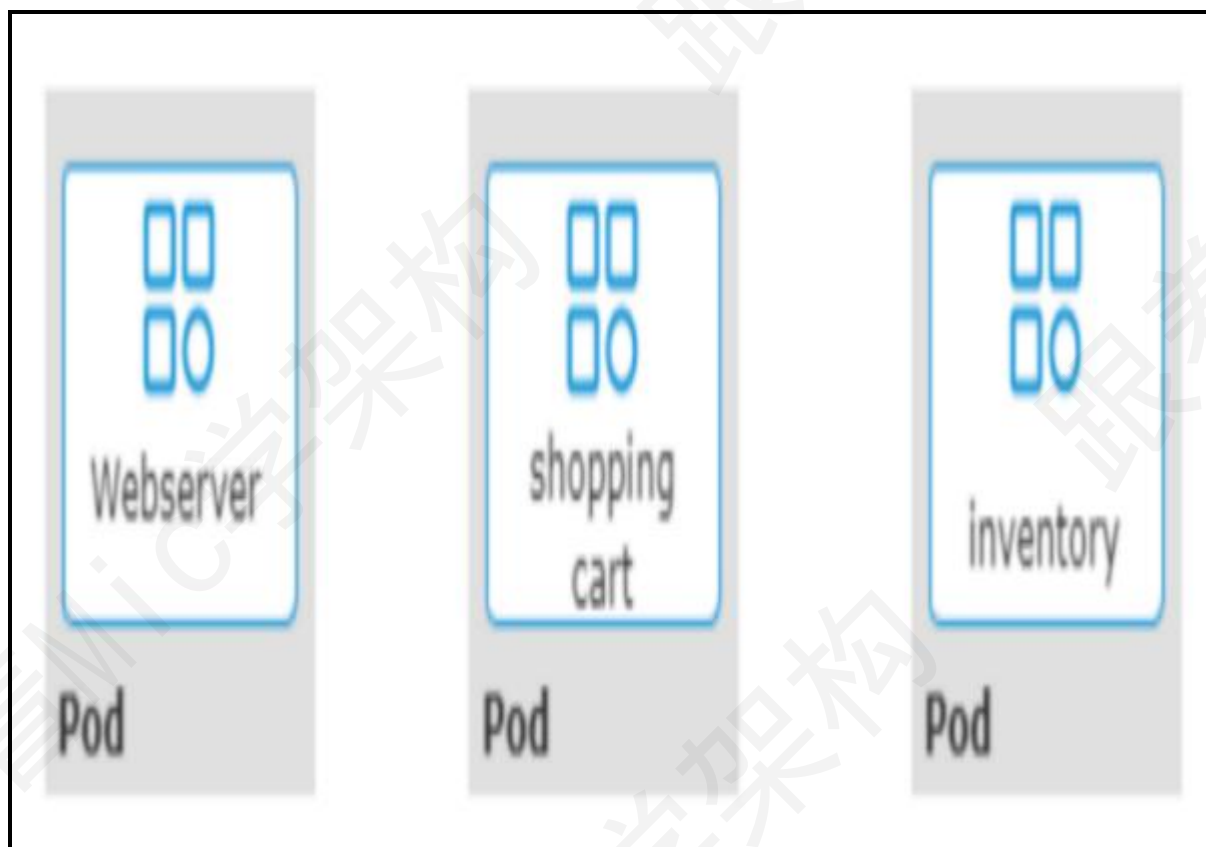
**Service Mesh**，我们通常把他称为第三代微服务架构，既然是第三代，那么意味着他是在原来的微服务架构下做的升级。

为了更好的说明 **Service Mesh**，那我就不得不说一下微服务架构部分的东西。

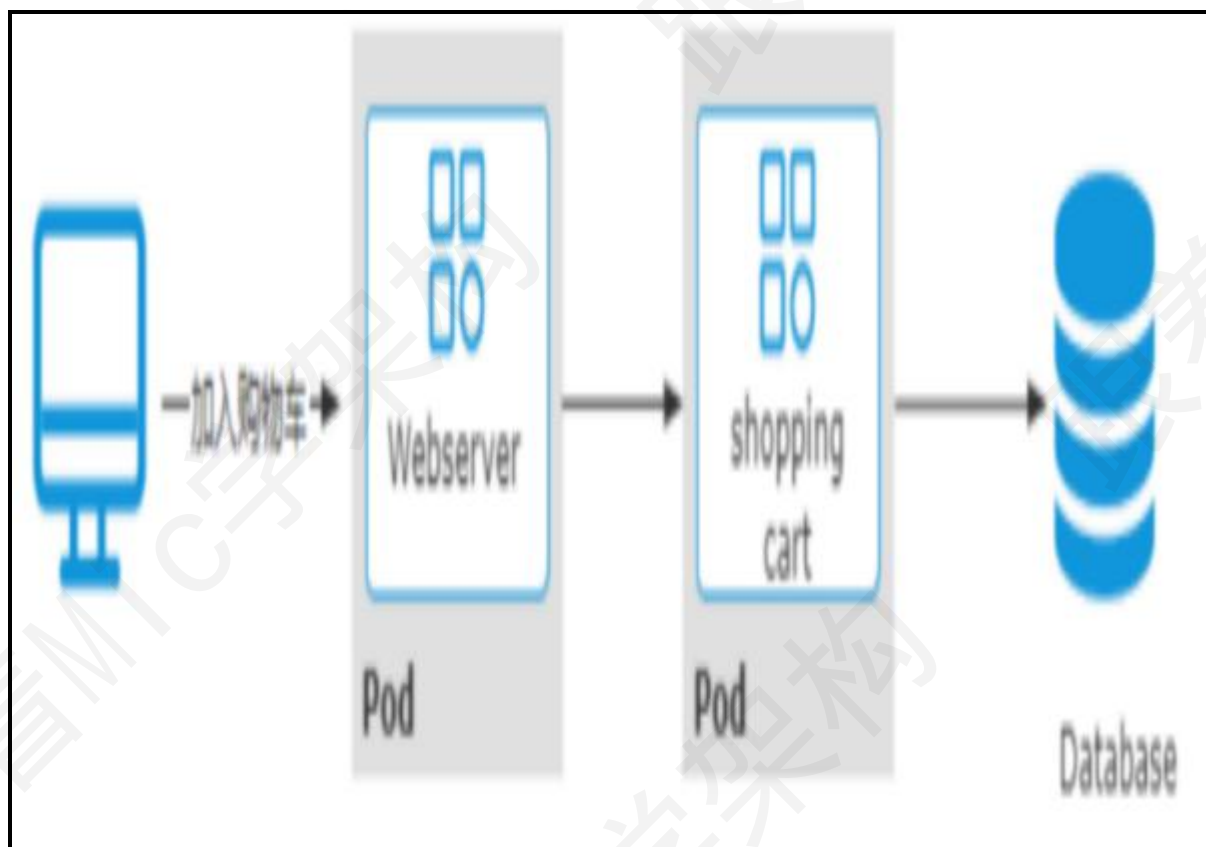
首先，当我们把一个电商系统以微服务化架构进行拆分后，会的到这样的一个架构，其中包括 **Webserver**、**payment**、**inventory** 等等。



这些微服务应用，会被部署到 Docker 容器、或者 Kubernetes 集群。由于每个服务的业务逻辑是独立的，比如 **payment** 会实现支付的业务逻辑、**order** 实现订单的处理、**Webserver** 实现客户端请求的响应等。



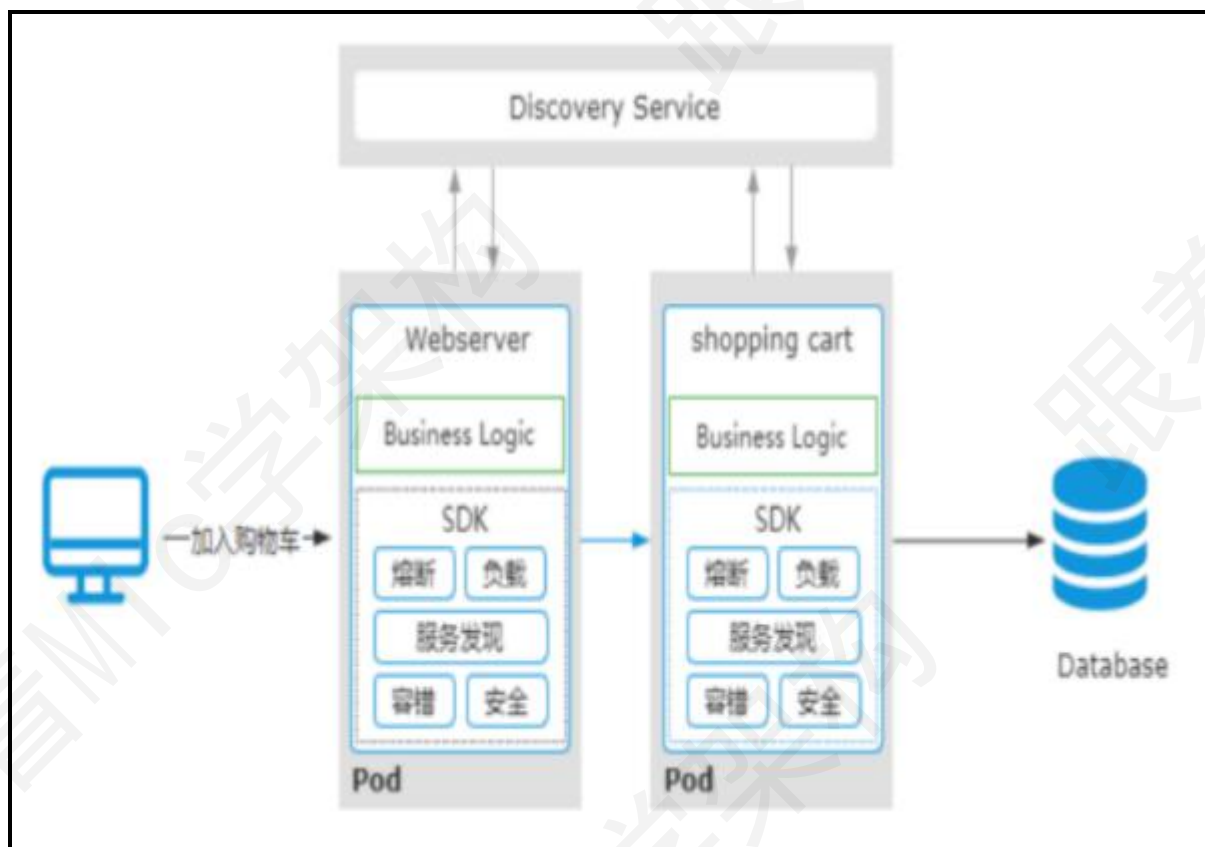
所以，服务之间必须要相互通信，才能实现功能的完整性。比如用户把一个商品加入购物车，请求会进入到 Webserver，然后转发到 shopping cart 进行处理，并存到数据库。



而在这个过程中，每个服务之间必须要知道对方的通信地址，并且当有新的节点加入进来的时候，还需要对这些通信地址进行动态维护。所以，在第一代微服务架构中，每个微服务除了要实现业务逻辑以外，还需要解决上下游寻址、通讯、以及容错等问题。

于是，在第二代微服务架构下，引入了服务注册中心来实现服务之间的寻址，并且服务之间的容错机制、负载均衡也逐步形成了独立的服务框架，比如主流的 Spring Cloud、或者 Spring Cloud Alibaba。





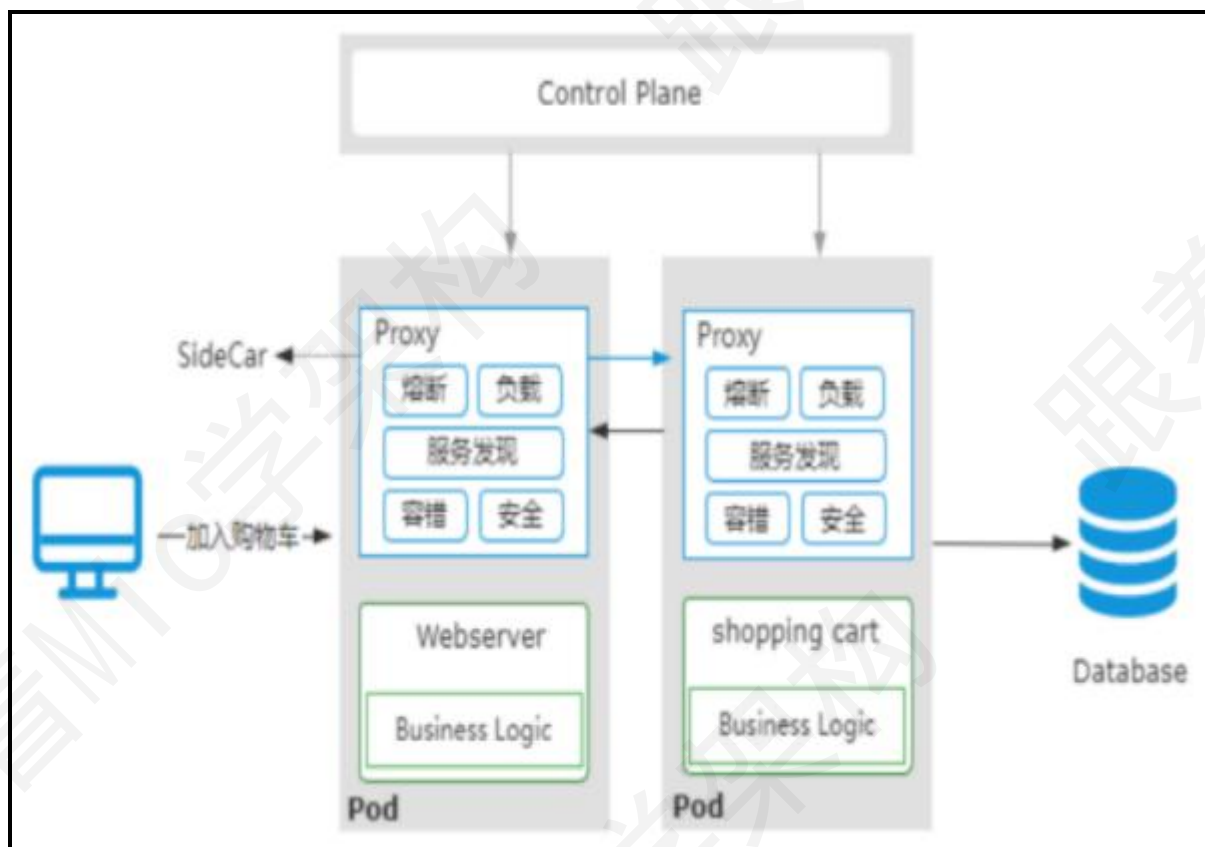
在第二代微服务架构中，负责业务开发的小伙伴不仅仅需要关注业务逻辑，还需要花大量精力去处理微服务中的一些基础性配置工作，虽然 **Spring Cloud** 已经尽可能去完成了这些事情，但对于开发人员来说，学习 **Spring Cloud**，以及针对 **Spring Cloud** 的配置和维护，仍然存在较大的挑战。另外呢，也增加了整个微服务的复杂性。

实际上，在我看来，“微服务中所有的这些服务注册、容错、重试、安全等工作，都是为了保证服务之间通信的可靠性”。

于是，就有了第三代微服务架构，**Service Mesh**。

原本模块化到微服务框架里的微服务基础能力，被进一步的从一个 **SDK** 中演进成了一个独立的代理进程-**SideCar**

**SideCar** 的主要职责就是负责各个微服务之间的通信，承载了原本第二代微服务架构中的服务发现、调用容错、服务治理等功能。使得微服务基础能力和业务逻辑迭代彻底解耦。



之所以我们称 **Service Mesh** 为服务网格，是因为在大规模微服务架构中，每个服务的通信都是由 **SideCar** 来代理的，各个服务之间的通信拓扑图，看起来就像一个网格形状。

**Istio** 是目前主流的 **Service Mesh** 开源框架。

以上就是我对服务网格的理解。

## 结尾

**Service Mesh** 架构其实就是云原生时代的微服务架构，对于大部分企业来说，仍然是处在第二代微服务架构下。

所以，很多小伙伴不一定能够知道。

不过，技术是在快速迭代的，有一句话叫“时代抛弃你的时候，连一句再见也不会说”，就像有些人在外包公司干了 10 多年

再出来面试，发现很多公司要求的技术栈，他都不会。所以，建议大家要时刻刷新自己的能力，保持竞争优势！

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 **Java** 程序员，咱们下期再见。

# Redis 和 Mysql 如何保证数据一致性

---

今天分享一道一线互联网公司高频面试题。

“Redis 和 Mysql 如何保证数据一致性”。

这个问题难倒了不少工作 5 年以上的程序员，难的不是问题本身，而是解决这个问题的思维模式。

下面来看看普通人和高手对于这个问题的回答。

## 普通人

---

嗯....

Redis 和 Mysql 的数据一致性保证是吧？我想想。

嗯，就是，Mysql 的数据发生变化以后，可以同步修改 Redis 里面的数据。

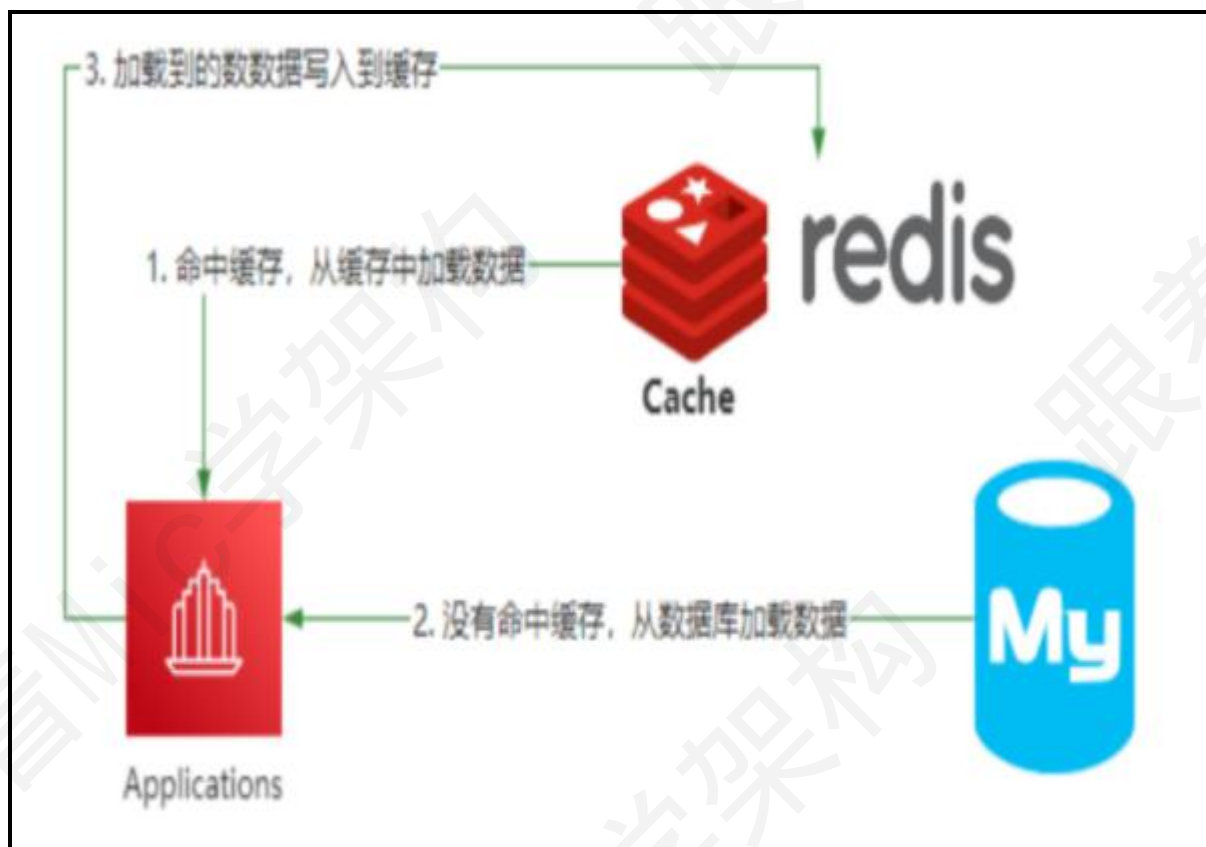
## 高手

---

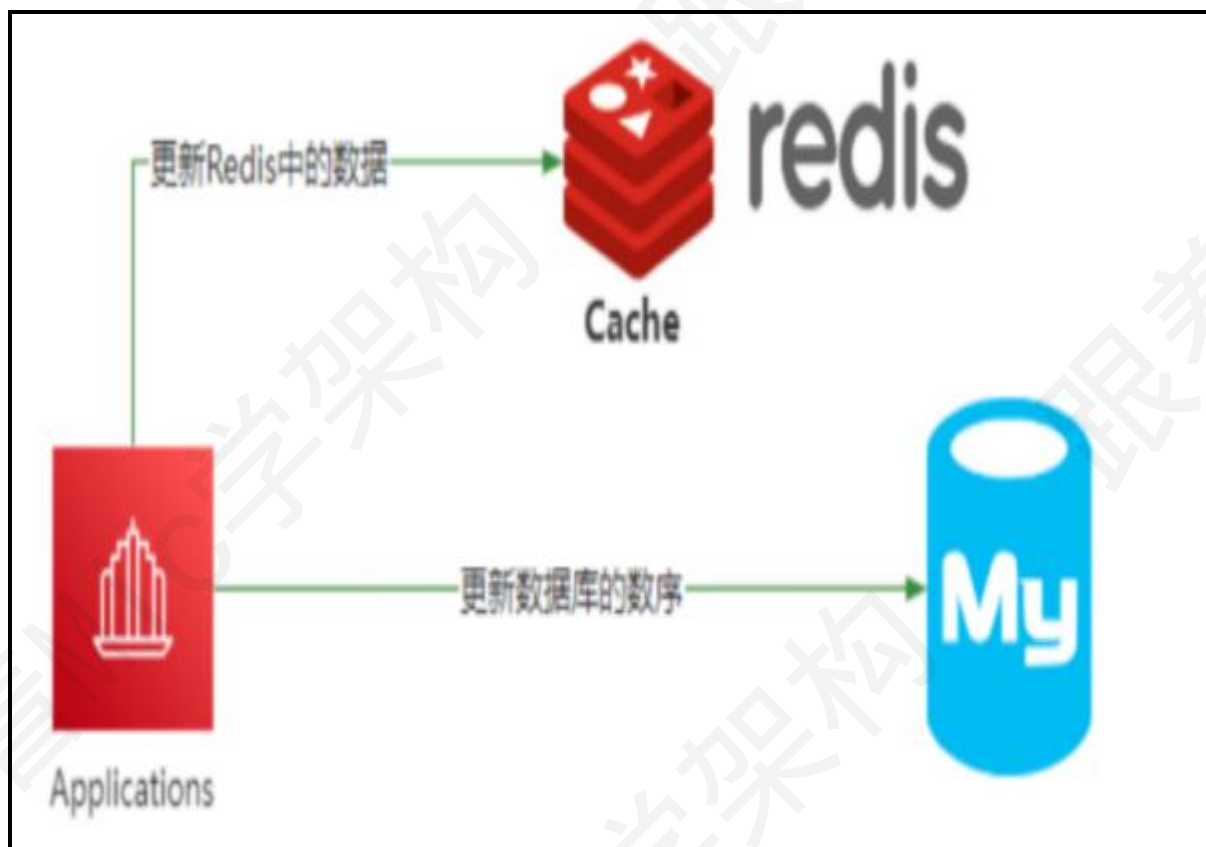
一般情况下，Redis 用来实现应用和数据库之间读操作的缓存层，主要目的是减少数据库 IO，还可以提升数据的 IO 性能。

这是它的整体架构。

当应用程序需要去读取某个数据的时候，首先会先尝试去 Redis 里面加载，如果命中就直接返回。如果没有命中，就从数据库查询，查询到数据后再把这个数据缓存到 Redis 里面。



在这样一个架构中, 会出现一个问题, 就是一份数据, 同时保存在数据库和 Redis 里面, 当数据发生变化的时候, 需要同时更新 Redis 和 Mysql, 由于更新是有先后顺序的, 并且它不像 Mysql 中的多表事务操作, 可以满足 ACID 特性。所以就会出现数据一致性问题。

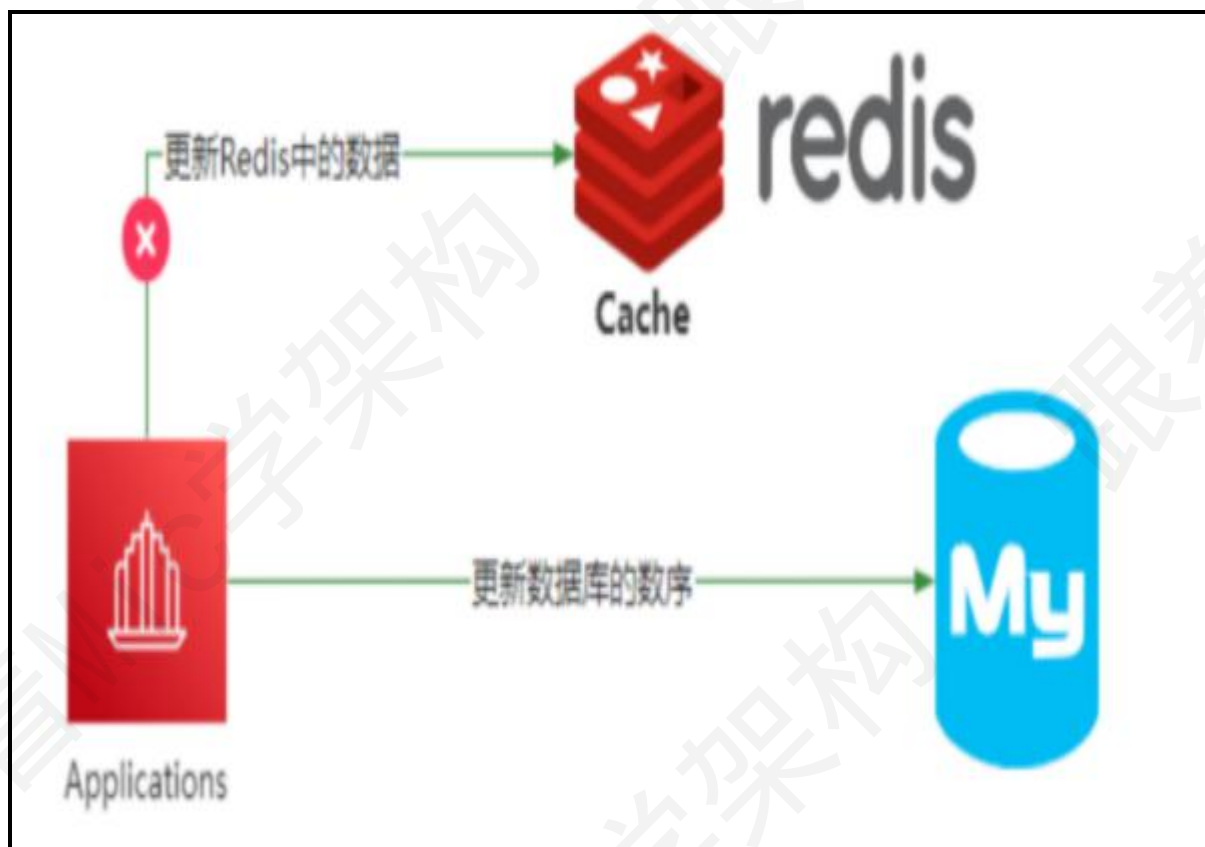


在这种情况下，能够选择的方法只有几种。

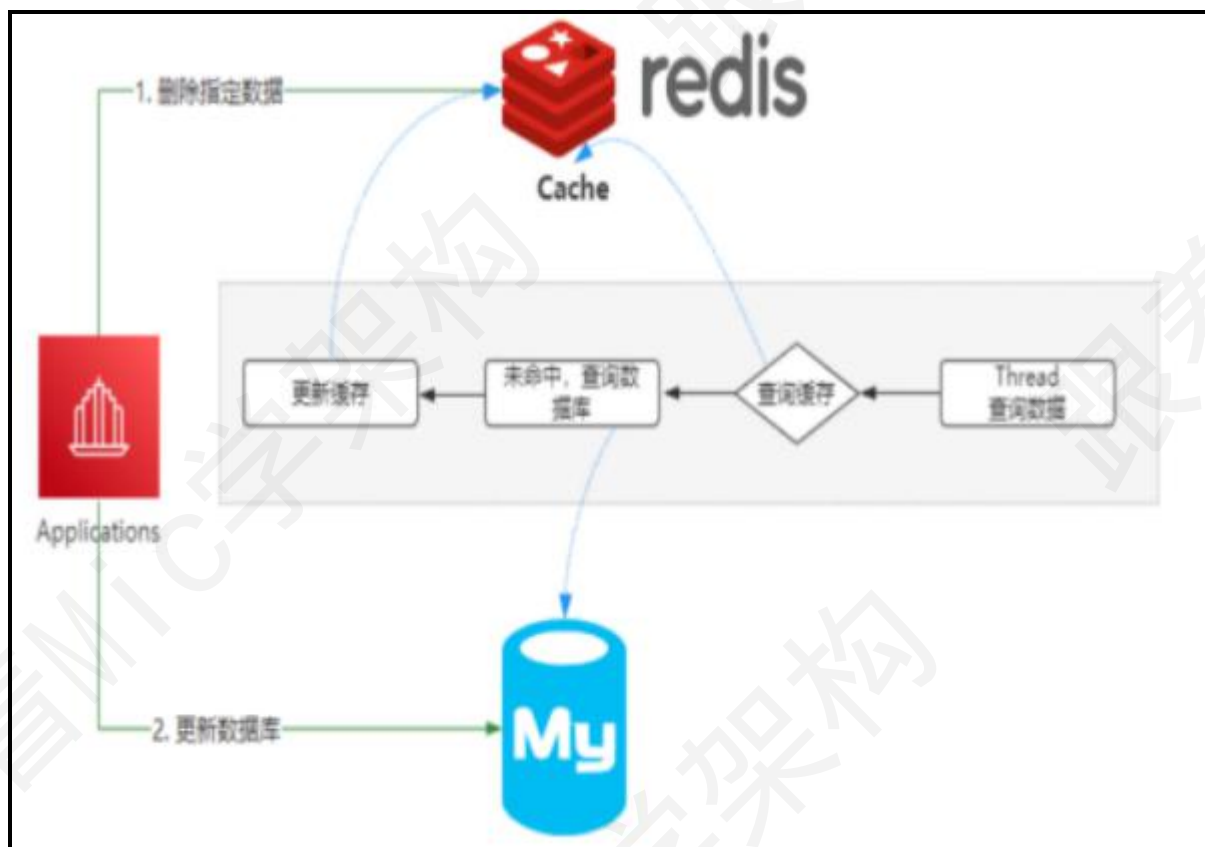
先更新数据库，再更新缓存

先删除缓存，再更新数据库

如果先更新数据库，再更新缓存，如果缓存更新失败，就会导致数据库和 Redis 中的数据不一致。

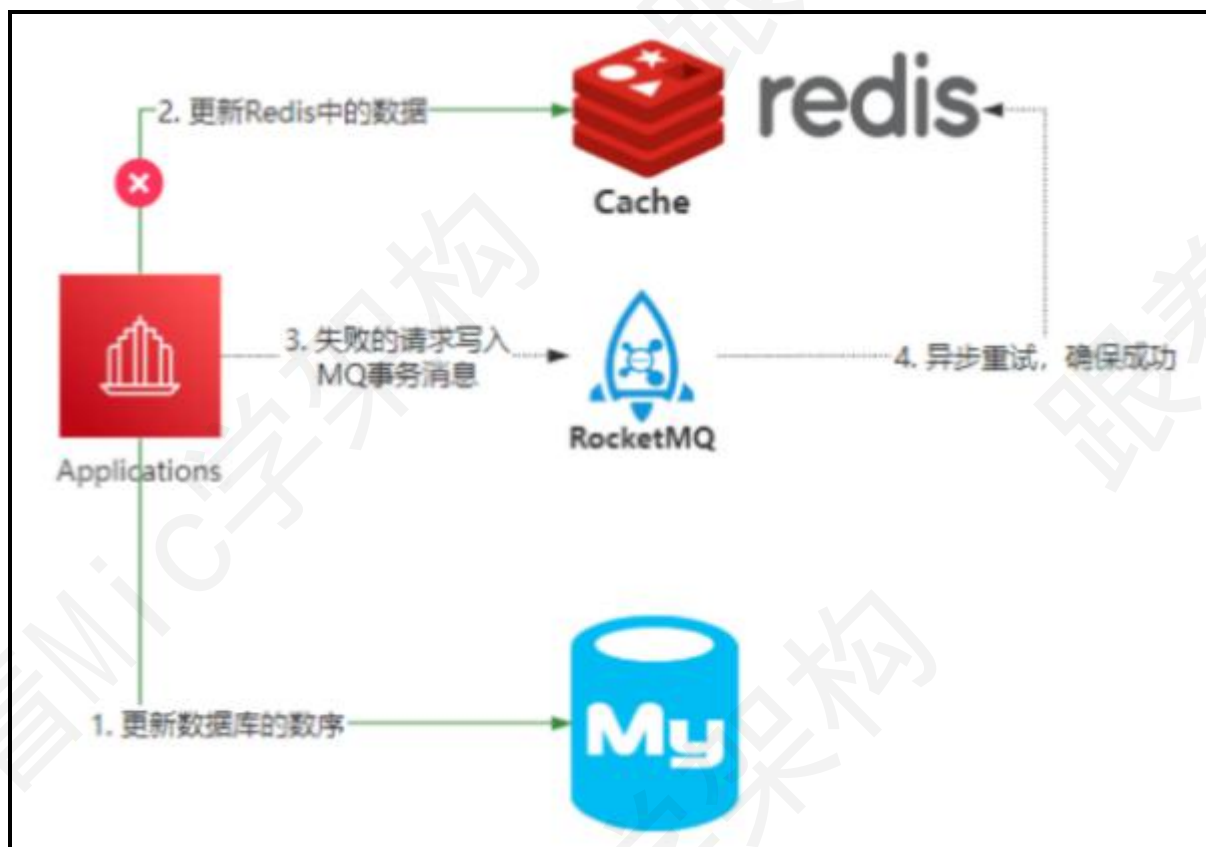


如果是先删除缓存，再更新数据库，理想情况是应用下次访问 **Redis** 的时候，发现 **Redis** 里面的数据是空的，就从数据库加载保存到 **Redis** 里面，那么数据是一致的。但是在极端情况下，由于删除 **Redis** 和更新数据库这两个操作并不是原子的，所以这个过程如果有其他线程来访问，还是会存在数据不一致问题。



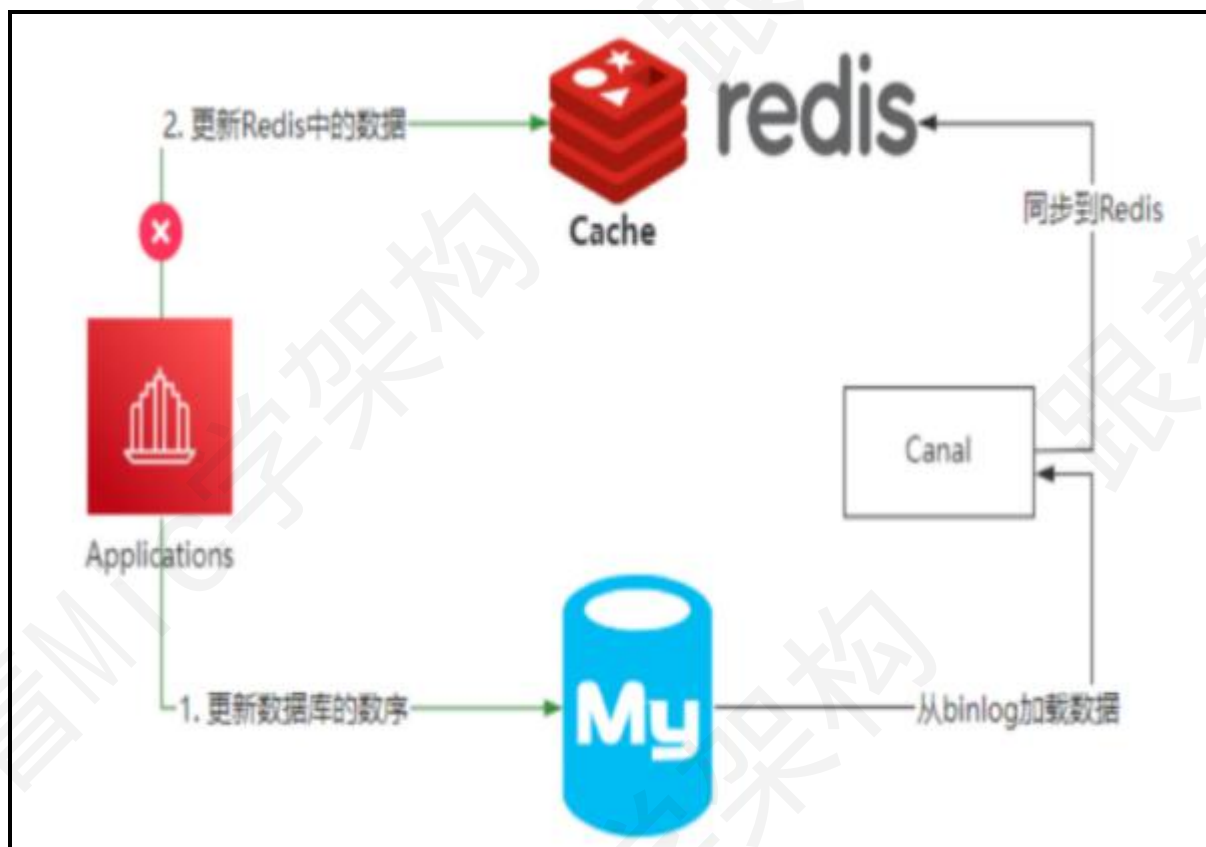
所以，如果需要在极端情况下仍然保证 Redis 和 Mysql 的数据一致性，就只能采用最终一致性方案。

比如基于 RocketMQ 的可靠性消息通信，来实现最终一致性。



还可以直接通过 Canal 组件，监控 Mysql 中 binlog 的日志，把更新后的数据同步到 Redis 里面。





因为这里是基于最终一致性来实现的，如果业务场景不能接受数据的短期不一致性，那就不能使用这个方案来做。

以上就是我对这个问题的理解。

## 结尾

在面试的时候，面试官喜欢问各种没有场景化的纯粹的技术问题，比如说：“你这个最终一致性方案”还是会存在数据不一致的问题啊？那怎么解决？

先不用慌，技术是为业务服务的，所以不同的业务场景，对于技术的选择和方案的设计都是不同的，所以这个时候，可以反问面试官，具体的业务场景是什么？

一定要知道的是，一个技术方案不可能 **cover** 住所有的场景，明白了吗？

好的，好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

另外，最近从评论区收到的面试问题，都有点太泛了，比如：说一下 **mongoDB** 啊，说一下 **kafka** 啊、说一下并发啊，这些问题都是要几个小时才能彻底说清楚，建议大家提具体一点的问题。

我是 Mic，一个工作了 14 年的 **Java** 程序员，咱们下期再见。

# Spring Boot 中自动装配机制的原理

最近一个粉丝说，他面试了 4 个公司，有三个公司问他：“Spring Boot 中自动装配机制的原理”

他回答了，感觉没回答错误，但是怎么就没给 offer 呢？

对于这个问题，看看普通人和高手该如何回答。

## 普通人

嗯...Spring Boot 里面的自动装配，就是@EnableAutoConfiguration 注解。

嗯...它可以实现 Bean 的自动管理，不需要我们手动再去配置。

## 高手

自动装配，简单来说就是自动把第三方组件的 Bean 装载到 Spring IOC 器里面，不需要开发人员再去写 Bean 的装配配置。

在 Spring Boot 应用里面，只需要在启动类加上@SpringBootApplication 注解就可以实现自动装配。

@SpringBootApplication 是一个复合注解，真正实现自动装配的注解是@EnableAutoConfiguration。

自动装配的实现主要依靠三个核心关键技术。

引入 Starter 启动依赖组件的时候，这个组件里面必须要包含@Configuration 配置类，在这个配置类里面通过@Bean 注解声明需要装配到 IOC 容器的 Bean 对象。

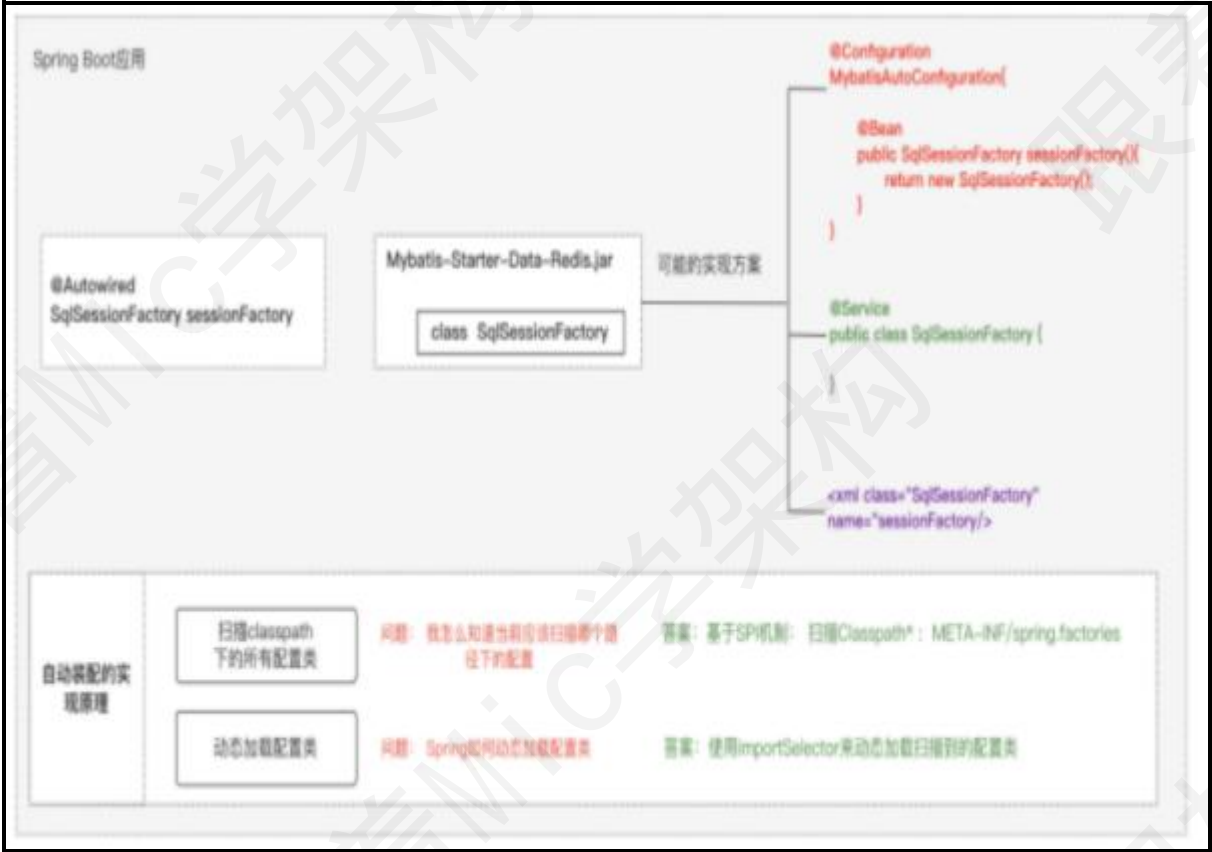
这个配置类是放在第三方的 jar 包里面，然后通过 SpringBoot 中的约定优于配置思想，把这个配置类的全路径放在 classpath:/META-INF/spring.factories 文件中。这样 SpringBoot 就可以知道第三方 jar 包里面的配置类的位置，这个步骤主要是用到了 Spring 里面的 SpringFactoriesLoader 来完成的。

SpringBoot 拿到所第三方 jar 包里面声明的配置类以后，再通过 Spring 提供的 ImportSelector 接口，实现对这些配置类的动态加载。

在我看来，SpringBoot 是约定优于配置这一理念下的产物，所以在很多的地方，都会看到这类的思想。它的出现，让开发人员更加聚焦在了业务代码的编写上，而不需要去关心和业务无关的配置。

其实，自动装配的思想，在 **SpringFramework3.x** 版本里面的 **@Enable** 注解，就有了实现的雏形。**@Enable** 注解是模块驱动的意思，我们只需要增加某个 **@Enable** 注解，就自动打开某个功能，而不需要针对这个功能去做 **Bean** 的配置，**@Enable** 底层也是帮我们去自动完成这个模块相关 **Bean** 的注入。

以上，就是我对 **Spring Boot** 自动装配机制的理解。



## 结尾

发现了吗？高手和普通人的回答，并不是回答的东西多和少。

而是让面试官看到你对于这个技术领域的理解深度和自己的见解，从而让面试官在一大堆求职者中，对你产生清晰的印象。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 死锁的发生原因和怎么避免

一个去阿里面试的小伙伴私信我说：今天被一个死锁的问题难到了。

平常我都特意看了死锁这块的内容，但是回答的时候就想不起来。

这里可能存在一个误区，认为技术是要靠记的。

大家可以想想，平时写代码的时候，这些代码是背下来的吗？

遇到一个需求的时候，能够立刻提供解决思路，这个也是记下来的吗？

所有的技术问题，都可以用一个问题来解决：“如果让你遇到这个问题，你会怎么设计”？

当你大脑一片空白时，说明你目前掌握的技术只能足够支撑你写 CURD 的能力。

好了，下面来看看普通人和高手是如何回答这个问题的。

## 普通人

---

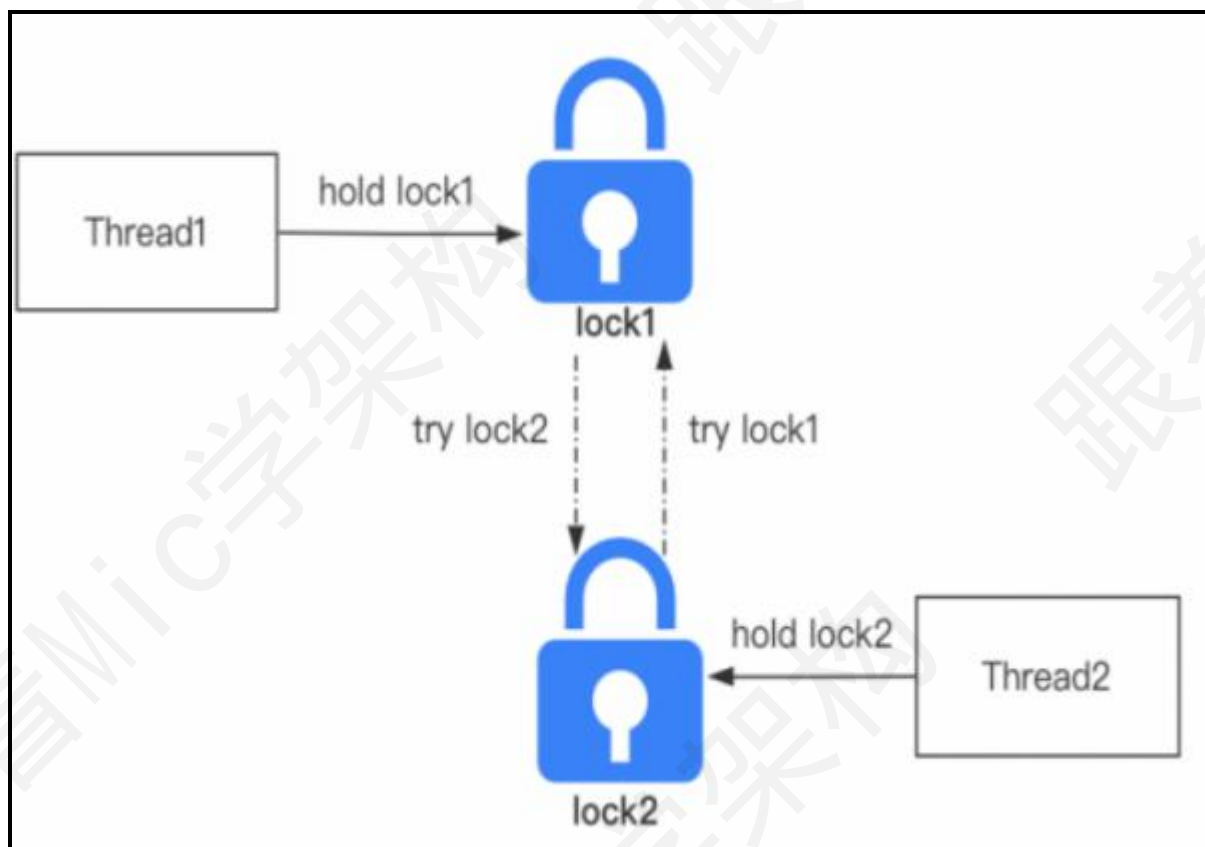
临场发挥...

## 高手

---

(如图)，死锁，简单来说就是两个或者两个以上的线程在执行的过程中，争夺同一个共享资源造成的相互等待的现象。

如果没有外部干预，线程会一直阻塞无法往下执行，这些一直处于相互等待资源的线程就称为死锁线程。



导致死锁的条件有四个，也就是这四个条件同时满足就会产生死锁。

互斥条件，共享资源 X 和 Y 只能被一个线程占用；

请求和保持条件，线程 T1 已经取得共享资源 X，在等待共享资源 Y 的时候，不释放共享资源 X；

不可抢占条件，其他线程不能强行抢占线程 T1 占有的资源；

循环等待条件，线程 T1 等待线程 T2 占有的资源，线程 T2 等待线程 T1 占有的资源，就是循环等待。

导致死锁之后，只能通过人工干预来解决，比如重启服务，或者杀掉某个线程。

所以，只能在写代码的时候，去规避可能出现的死锁问题。

按照死锁发生的四个条件，只需要破坏其中的任何一个，就可以解决，但是，互斥条件是没办法破坏的，因为这是互斥锁的基本约束，其他三方条件都有办法来破坏：

对于“请求和保持”这个条件，我们可以一次性申请所有的资源，这样就不存在等待了。

对于“不可抢占”这个条件，占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可抢占这个条件就破坏掉了。

对于“循环等待”这个条件，可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后自然就不存在循环了。

以上就是我对这个问题的理解。

## 结尾

---

发现了吗？当大家理解了死锁发生的条件，那么对于这些条件的破坏，是可以通过自己的技术积累，来设计解决方法的。

所有的技术思想和技术架构，都是由人来设计的，为什么别人能够设计？

本质上，还是技术积累后的结果！越是底层的设计，对于知识面的要求就越多。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 请说一下你对分布式锁的理解，以及分布式锁的实现

---

一个工作了 7 年的 Java 程序员，私信我关于分布式锁的问题。

一上来就两个灵魂拷问：

Redis 锁超时怎么办？

Redis 主从切换导致锁失效怎么办？

我说，别着急，这些都是小问题。

那么，关于“分布式锁的理解和实现”这个问题，我们看看普通人高手的回答。

### 普通人

---

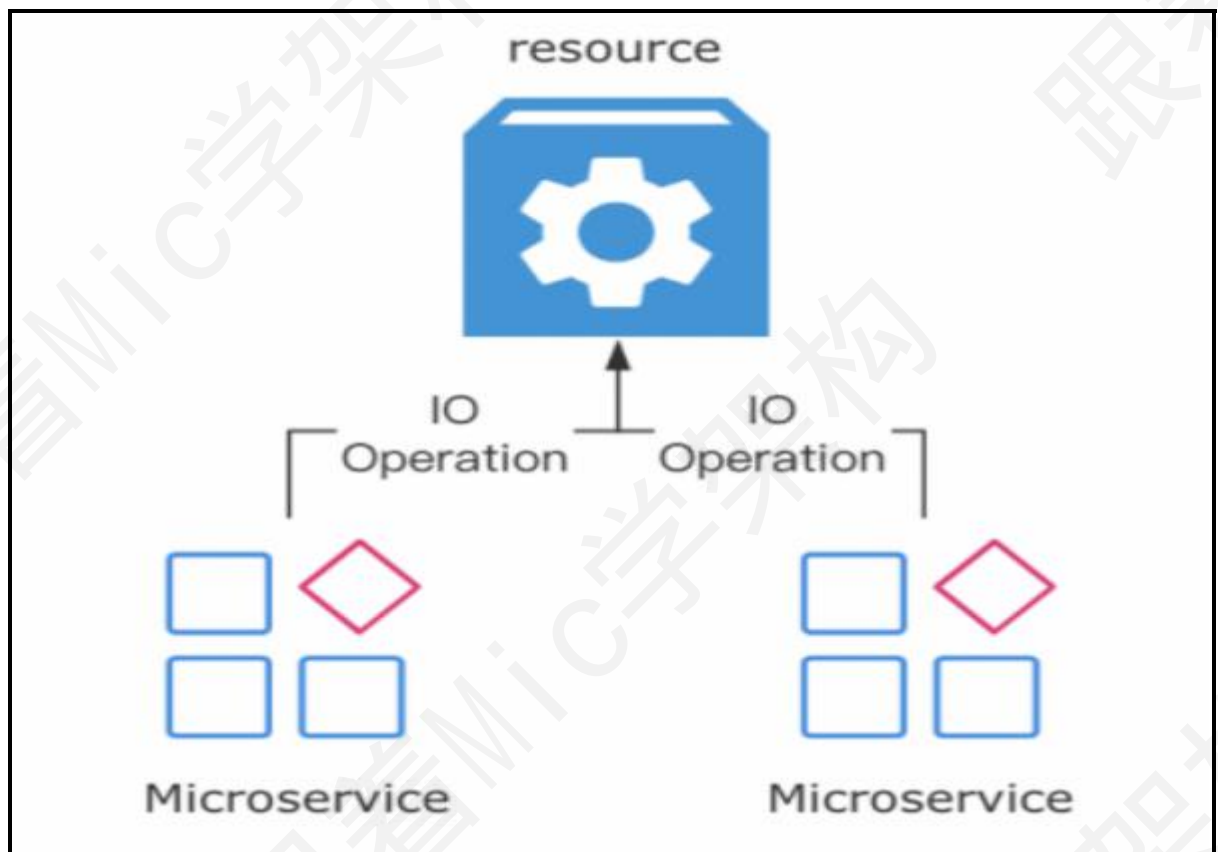
嗯，分布式锁，就是可以用来实现锁的分布性，嗯...

就是可以解决跨进程的应用对于共享资源访问的冲突问题。

可以用 Redis 来实现分布式锁。

## 高手

分布式锁，是一种跨进程的跨机器节点的互斥锁，它可以用来保证多机器节点对于共享资源访问的排他性。



我觉得分布式锁和线程锁本质上是一样的，线程锁的生命周期是单进程多线程，分布式锁的声明周期是多进程多机器节点。

在本质上，他们都需要满足锁的几个重要特性：

排他性，也就是说，同一时刻只能有一个节点去访问共享资源。

可重入性，允许一个已经获得锁的进程，在没有释放锁之前再次重新获得锁。

锁的获取、释放的方法

锁的失效机制，避免死锁的问题

所以，我认为，只要能够满足这些特性的技术组件都能够实现分布式锁。

关系型数据库，可以使用唯一约束来实现锁的排他性，

如果要针对某个方法加锁，就可以创建一个表包含方法名称字段，

并且把方法名设置成唯一的约束。

那抢占锁的逻辑就是：往表里面插入一条数据，如果已经有其他的线程获得了某个方法的锁，那这个时候插入数据会失败，从而保证了互斥性。

这种方式虽然简单啊，但是要实现比较完整的分布式锁，还需要考虑重入性、锁失效机制、没抢占到锁的线程要实现阻塞等，就会比较麻烦。

**Redis**，它里面提供了 **SETNX** 命令可以实现锁的排他性，当 **key** 不存在就返回 1，存在就返回 0。然后还可以用 **expire** 命令设置锁的失效时间，从而避免死锁问题。

当然有可能存在锁过期了，但是业务逻辑还没执行完的情况。所以这种情况，可以写一个定时任务对指定的 **key** 进行续期。

**Redisson** 这个开源组件，就提供了分布式锁的封装实现，并且也内置了一个 **Watch Dog** 机制来对 **key** 做续期。

我认为 **Redis** 里面这种分布式锁设计已经能够解决 99% 的问题了，当然如果在 **Redis** 搭建了高可用集群的情况下出现主从切换导致 **key** 失效，这个问题也有可能造成

多个线程抢占到同一个锁资源的情况，所以 **Redis** 官方也提供了一个 **RedLock** 的解决办法，但是实现会相对复杂一些。

在我看来，分布式锁应该是一个 **CP** 模型，而 **Redis** 是一个 **AP** 模型，所以在集群架构下由于数据的一致性问题导致极端情况下出现多个线程抢占到锁的情况很难避免。

那么基于 **CP** 模型又能实现分布式锁特性的组件，我认为可以选择 **Zookeeper** 或者 **etcd**，

在数据一致性方面，**zookeeper** 用到了 **zab** 协议来保证数据的一致性，**etcd** 用到了 **raft** 算法来保证数据一致性。

在锁的互斥方面，**zookeeper** 可以基于有序节点再结合 **Watch** 机制实现互斥和唤醒，**etcd** 可以基于 **Prefix** 机制和 **Watch** 实现互斥和唤醒。

以上就是我对于分布式锁的理解！

## 面试点评

我认为，回答这个问题的核心本质，还是在技术底层深度理解的基础上的思考。



可以从高手的回答中明显感受到，对于排它锁底层逻辑的理解是很深刻的，同时在技术的广度上也是有足够的积累。

所以在回答的时候，面试官可以去抓到求职者在回答这个问题的技术关键点和思维。

我认为，当具备体系化的技术能力的时候，是很容易应对各种面试官的各种刁难的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，有任何技术上的问题，职业发展有关的问题，都可以私信我，我会在第一时间回复。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## volatile 关键字有什么用？它的实现原理是什么？

---

一个工作了 6 年的 Java 程序员，在阿里二面，被问到“volatile”关键字。

然后，就没有然后了...

同样，另外一个去美团面试的工作 4 年的小伙伴，也被“volatile 关键字”。

然后，也没有然后了...

这个问题说实话，是有点偏底层，但也的确是并发编程里面比较重要的一个关键字。

下面，我们来看看普通人和高手对于这个问题的回答吧。

### 普通人

---

嗯...volatile 可以保证可见性。

### 高手

---

volatile 关键字有两个作用。

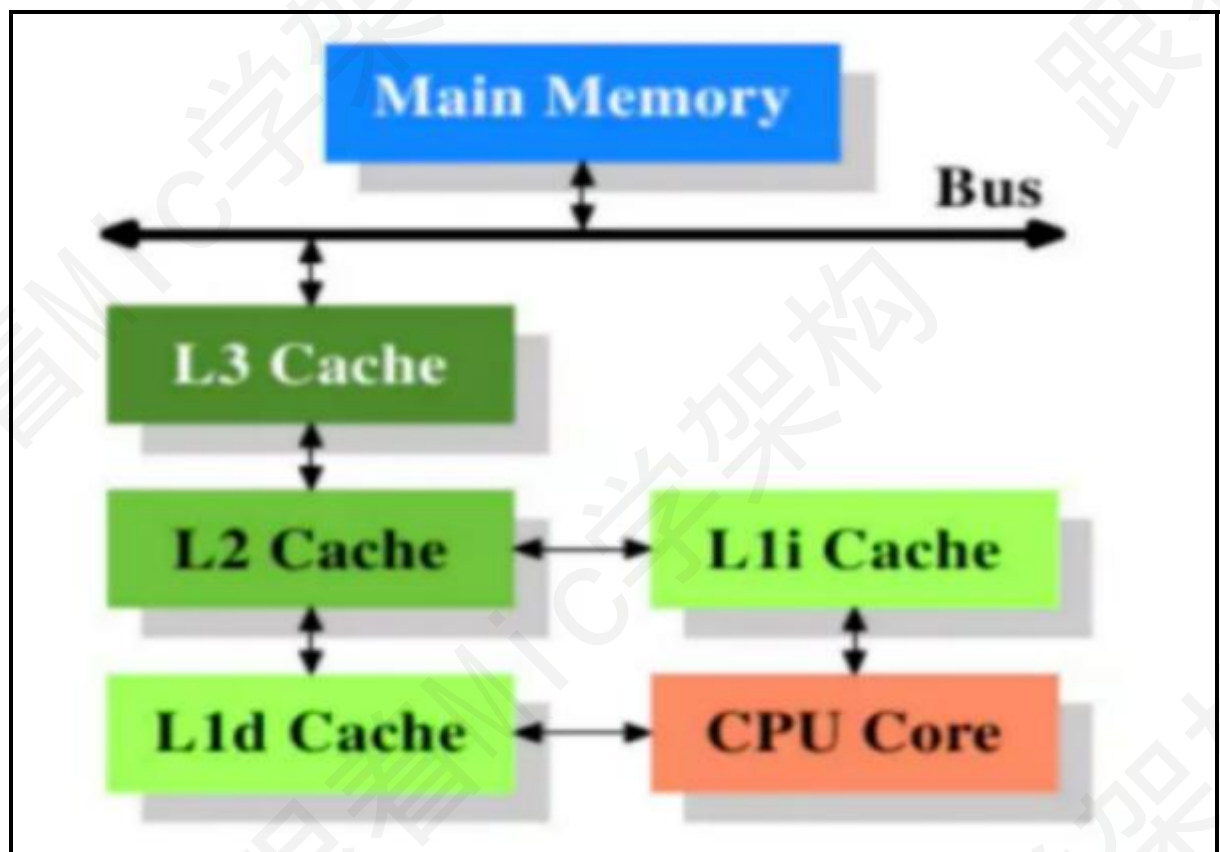
可以保证在多线程环境下共享变量的可见性。

通过增加内存屏障防止多个指令之间的重排序。

我理解的可见性，是指当某一个线程对共享变量的修改，其他线程可以立刻看到修改之后的值。

其实这个可见性问题，我认为本质上是由几个方面造成的。

CPU 层面的高速缓存，在 CPU 里面设计了三级缓存去解决 CPU 运算效率和内存 IO 效率问题，但是带来的就是缓存的一致性问题，而在多线程并行执行的情况下，缓存一致性就会导致可见性问题。



所以，对于增加了 `volatile` 关键字修饰的共享变量，JVM 虚拟机会自动增加一个 `#Lock` 汇编指令，这个指令会根据 CPU 型号自动添加总线锁或/缓存锁

我简单说一下这两种锁，

总线锁是锁定了 CPU 的前端总线，从而导致在同一时刻只能有一个线程去和内存通信，这样就避免了多线程并发造成的可见性。

缓存锁是对总线锁的优化，因为总线锁导致了 CPU 的使用效率大幅度下降，所以缓存锁只针对 CPU 三级缓存中的目标数据加锁，缓存锁是使用 MESI 缓存一致性来实现的。

指令重排序，所谓重排序，就是指指令的编写顺序和执行顺序不一致，在多线程环境下导致可见性问题。指令重排序本质上是一种性能优化的手段，它来自于几个方面。

CPU 层面，针对 MESI 协议的更进一步优化去提升 CPU 的利用率，引入了 StoreBuffer 机制，而这一种优化机制会导致 CPU 的乱序执行。当然为了避免这样的问题，CPU 提供了内存屏障指令，上层应用可以在合适的地方插入内存屏障来避免 CPU 指令重排序问题。

编译器的优化，编译器在编译的过程中，在不改变单线程语义和程序正确性的前提下，对指令进行合理重排序优化来提升性能。

所以，如果对共享变量增加了 volatile 关键字，那么在编译器层面，就不会去触发编译器优化，同时再 JVM 里面，会插入内存屏障指令来避免重排序问题。

当然，除了 volatile 以外，从 JDK5 开始，JMM 就使用了一种 Happens-Before 模型去描述多线程之间的内存可见性问题。

如果两个操作之间具备 Happens-Before 关系，那么意味着这两个操作具备可见性关系，不需要再额外去考虑增加 volatile 关键字来提供可见性保障。

以上就是我对这个问题的理解。

## 面试点评

---

在我看来，并发编程是每个程序员必须要掌握好的领域，它里面涵盖的设计思想、和并发问题的解决思路、以及作为一个并发工具，都是非常值得深度研究的。

我推荐大家去读一下《Java 并发编程深度解析与原理实战》这本书，对 Java 并发这块的内容描述得很清晰。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，有任何技术上的问题，职业发展有关的问题，都可以私信我，我会在第一时间回复。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 说说缓存雪崩和缓存穿透的理解，以及如何避免？

---

听说 10 个人去互联网公司面试，有 9 个人会被问到缓存雪崩和缓存穿透的问题。

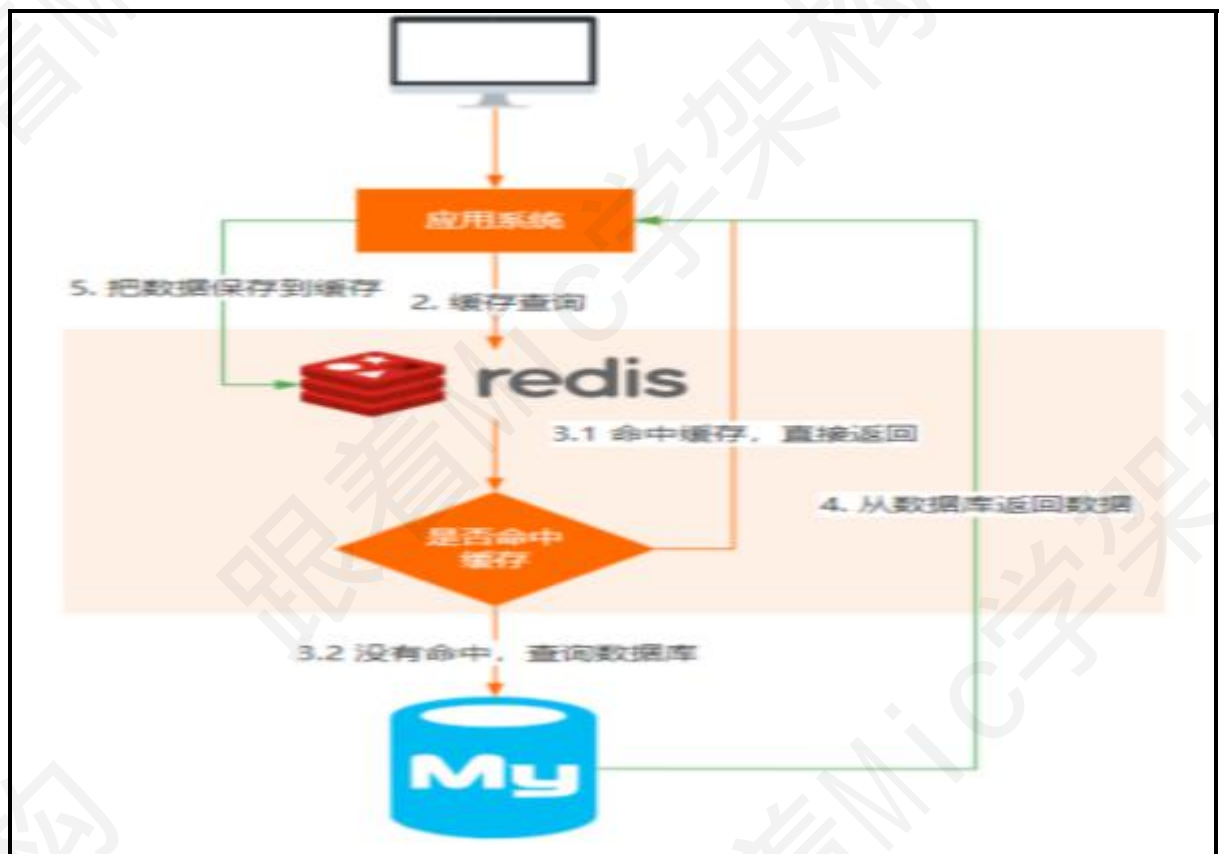
听说，这 9 个人里面，至少有 8 个人回答得不完整。

而这 8 个人里面，全都是在网上找的各种面试资料去应付的，并没有真正理解。当然，也很正常，只有大规模应用缓存的架构才会重点关注这两个问题。那么如何真正理解这两个问题的底层逻辑，我们来看普通人和高手的回答。

## 普通人

## 高手

缓存雪崩，就是存储在缓存里面的大量数据，在同一个时刻全部过期，原本缓存组件抗住的大部分流量全部请求到了数据库。导致数据库压力增加造成数据库服务器崩溃的现象。



导致缓存雪崩的主要原因，我认为有两个：

缓存中间件宕机，当然可以对缓存中间件做高可用集群来避免。

缓存中大部分 key 都设置了相同的过期时间，导致同一时刻这些 key 都过期了。对于这样的情况，可以在失效时间上增加一个 1 到 5 分钟的随机值。

缓存穿透问题，表示是短时间内有大量的不存在的 **key** 请求到应用里面，而这些不存在的 **key** 在缓存里面又找不到，从而全部穿透到了数据库，造成数据库压力。

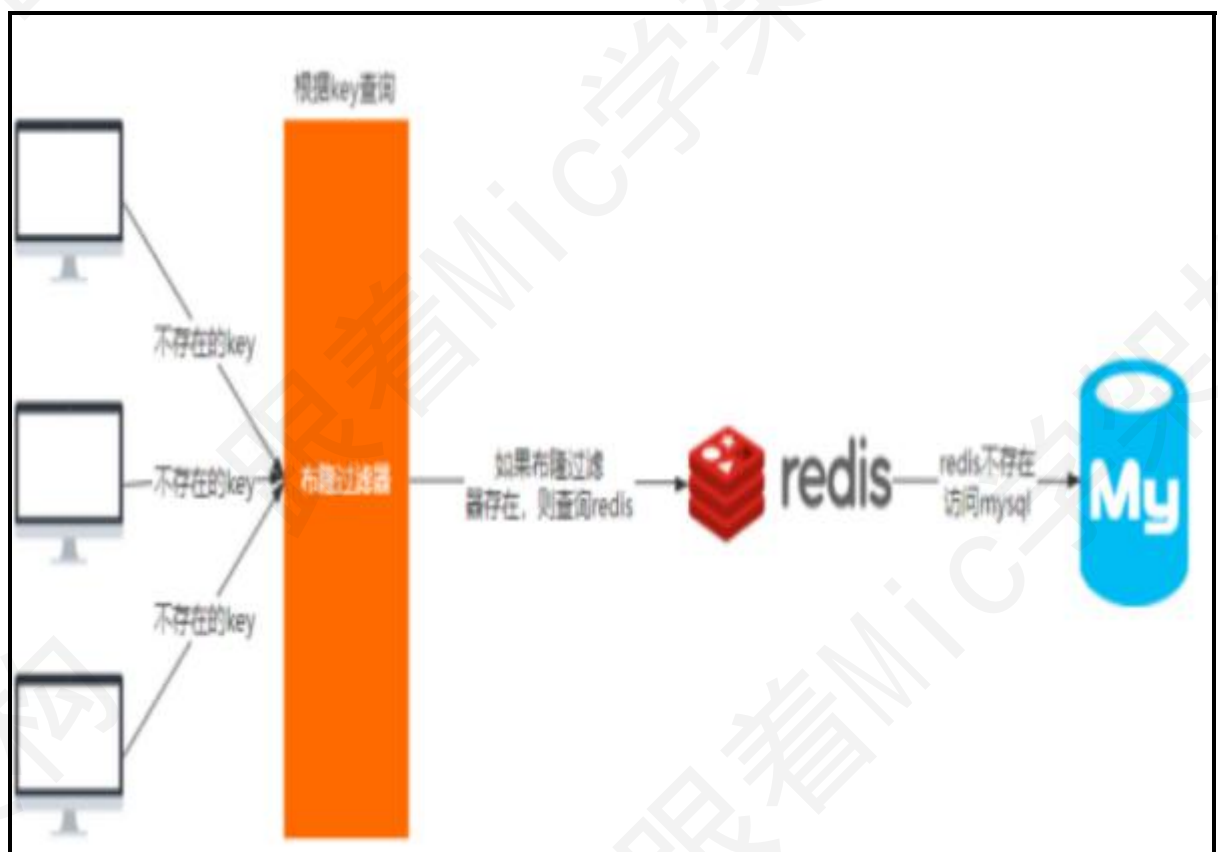
我认为这个场景的核心问题是针对缓存的一种攻击行为，因为在正常的业务里面，即便是出现了这样的情况，由于缓存的不断预热，影响不会很大。

而攻击行为就需要具备时间的持续性，而只有 **key** 确实在数据库里面也不存在的情况下，才能达到这个目的，所以，我认为有两个方法可以解决：

把无效的 **key** 也保存到 **Redis** 里面，并且设置一个特殊的值，比如“null”，这样的话下次再来访问，就不会去查数据库了。

但是如果攻击者不断用随机的不存在 **key** 来访问，也还是会存在问题，所以可以用布隆过滤器来实现，在系统启动的时候把目标数据全部缓存到布隆过滤器里面，当攻击者用不存在的 **key** 来请求的时候，先到布隆过滤器里面查询，如果不存在，那意味着这个 **key** 在数据库里面也不存在。

布隆过滤器还有一个好处，就是它采用了 **bitmap** 来进行数据存储，占用的内存空间很少。



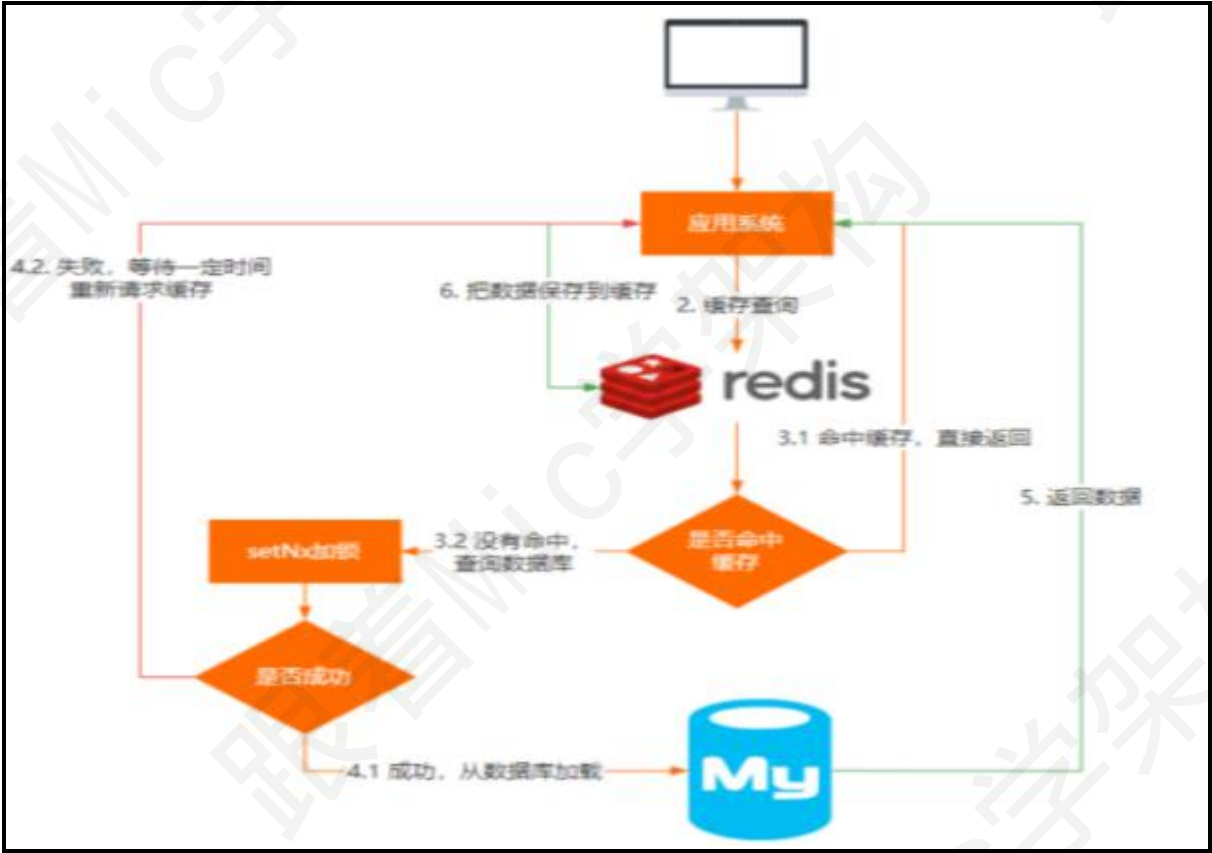
不过，在我看来，您提出来的这个问题，有点过于放大了它带来的影响。

首先，在一个成熟的系统里面，对于比较重要的热点数据，必然会有一个专门缓存系统来维护，同时它的过期时间的维护必然和其他业务的 key 有一定的差别。而且非常重要的场景，我们还会设计多级缓存系统。

其次，即便是触发了缓存雪崩，数据库本身的容灾能力也并没有那么脆弱，数据库的主从、双主、读写分离这些策略都能够很好的缓解并发流量。

最后，数据库本身也有最大连接数的限制，超过限制的请求会被拒绝，再结合熔断机制，也能够很好的保护数据库系统，最多就是造成部分用户体验不好。

另外，在程序设计上，为了避免缓存未命中导致大量请求穿透到数据库的问题，还可以在访问数据库这个环节加锁。虽然影响了性能，但是对系统是安全的。



总而言之，我认为解决的办法很多，具体选择哪种方式，还是看具体的业务场景。

以上就是我对这个问题的理解。

## 面试点评

我发现现在很多面试，真的是为了面试而面试，要么就是在网上摘题，要么就是不断的问一些无关痛痒的问题。

至于最终面试官怎么判断你是否合适，咱也不知道，估计就是有些小伙伴说的，看长相，看眼缘！

我认为一个合格的面试官，他必须要具备非常深厚的技术功底。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，有任何技术上的问题，职业发展有关的问题，都可以私信我，我会在第一时间回复。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 讲一下 wait 和 notify 这个为什么要在 synchronized 代码块中？

一个工作七年的小伙伴，竟然不知道“wait”和“notify”为什么要在 Synchronized 代码块里面。

好吧，如果屏幕前的你也不知道，请在公屏上刷“不知道”。

对于这个问题，我们来看看普通人和高手的回答。

### 普通人

### 高手

wait 和 notify 用来实现多线程之间的协调，wait 表示让线程进入到阻塞状态，notify 表示让阻塞的线程唤醒。

wait 和 notify 必然是成对出现的，如果一个线程被 wait() 方法阻塞，那么必然需要另外一个线程通过 notify() 方法来唤醒这个被阻塞的线程，从而实现多线程之间的通信。

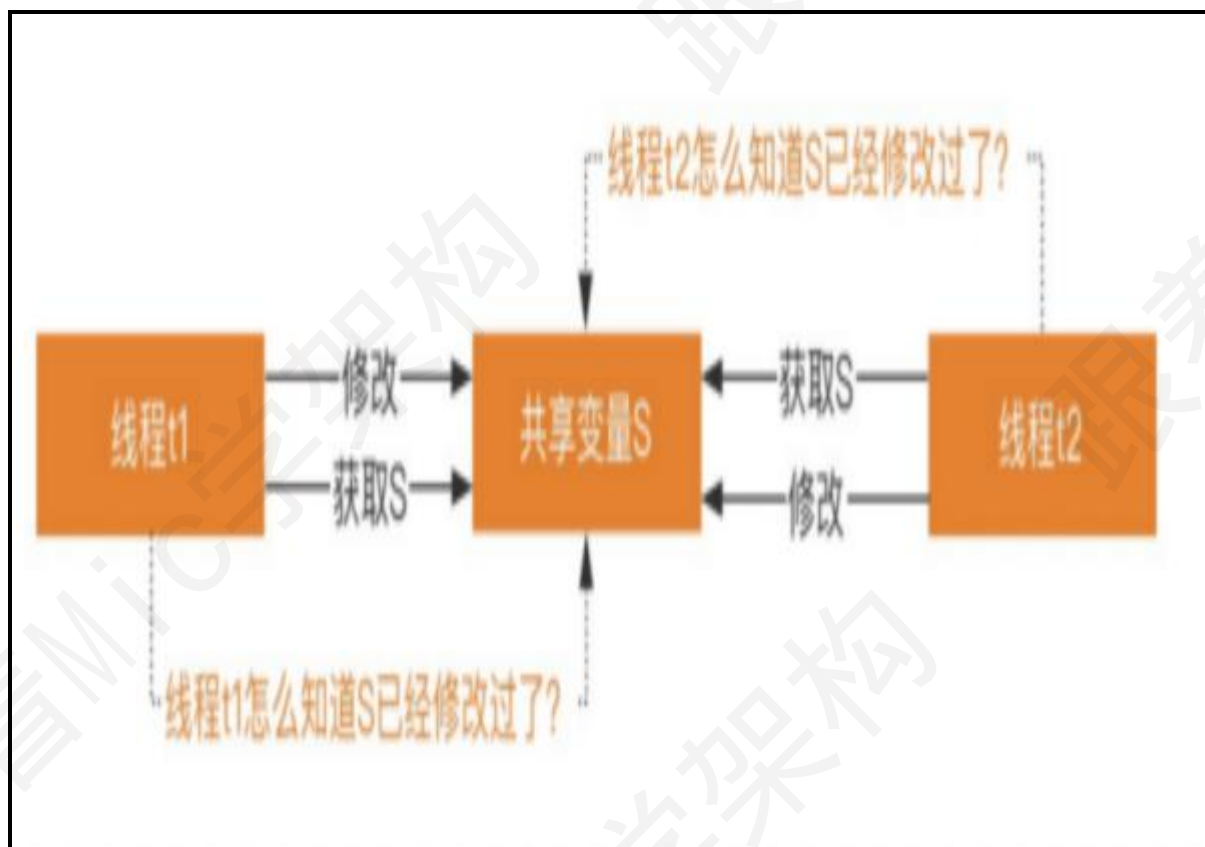
在多线程里面，要实现多个线程之间的通信，除了管道流以外，只能通过共享变量的方法来实现，也就是线程 t1 修改共享变量 s，线程 t2 获取修改后的共享变量 s，从而完成数据通信。

但是多线程本身具有并行执行的特性，也就是在同一时刻，多个线程可以同时执行。在这种情况下，线程 t2 在访问共享变量 s 之前，必须要知道线程 t1 已经修改过了共享变量 s，否则就需要等待。

同时，线程 t1 修改过了共享变量 S 之后，还需要通知在等待中的线程 t2。

所以要在这种特性下要去实现线程之间的通信，就必须要有个竞争条件控制线程在什么条件下等待，什么条件下唤醒。





而 **Synchronized** 同步关键字就可以实现这样一个互斥条件，也就是在通过共享变量来实现多个线程通信的场景里面，参与通信的线程必须要竞争到这个共享变量的锁资源，才有资格对共享变量做修改，修改完成后就释放锁，那么其他的线程就可以再次来竞争同一个共享变量的锁来获取修改后的数据，从而完成线程之前的通信。

所以这也是为什么 **wait/notify** 需要放在 **Synchronized** 同步代码块中的原因，有了 **Synchronized** 同步锁，就可以实现对多个通信线程之间的互斥，实现条件等待和条件唤醒。

另外，为了避免 **wait/notify** 的错误使用，jdk 强制要求把 **wait/notify** 写在同步代码块里面，否则会抛出 **IllegalMonitorStateException**

最后，基于 **wait/notify** 的特性，非常适合实现生产者消费者的模型，比如说用 **wait/notify** 来实现连接池就绪前的等待与就绪后的唤醒。

以上就是我对 **wait/notify** 这个问题的理解。

## 面试点评

这个是一个典型的经典面试题。



其实考察的就是 **Synchronized**、**wait/notify** 的设计原理和实现原理。

由于 **wait/notify** 在业务开发整几乎不怎么用到，所以大部分人回答不出来。

其实并发这块内容理论上来说所有程序员都应该要懂，不管是它的应用价值，还是设计理念，非常值得学习和借鉴。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，有任何技术上的问题，职业发展有关的问题，都可以私信我，我会在第一时间回复。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## ThreadLocal 是什么？它的实现原理呢？

---

一个工作了 4 年的小伙伴，又私信了我一个并发编程里面的问题。

他说他要抓狂了，每天 **CRUD**，也没用到过 **ThreadLocal** 啊，怎么就不能问我怎么写 **CRUD** 呢？

我反问他，如果只问你项目和业务，那有些 4 年的小伙伴他要求月薪 **30K**，有些只要求月薪 **15K**，

那请问，凭什么每个月要多出 **15k** 给你？我花 **30k** 招两个 **15k** 的，不能写 **CRUD** 吗？

好吧，我们来看看 **ThreadLocal** 这个问题，普通人和高手的回答。

### 普通人

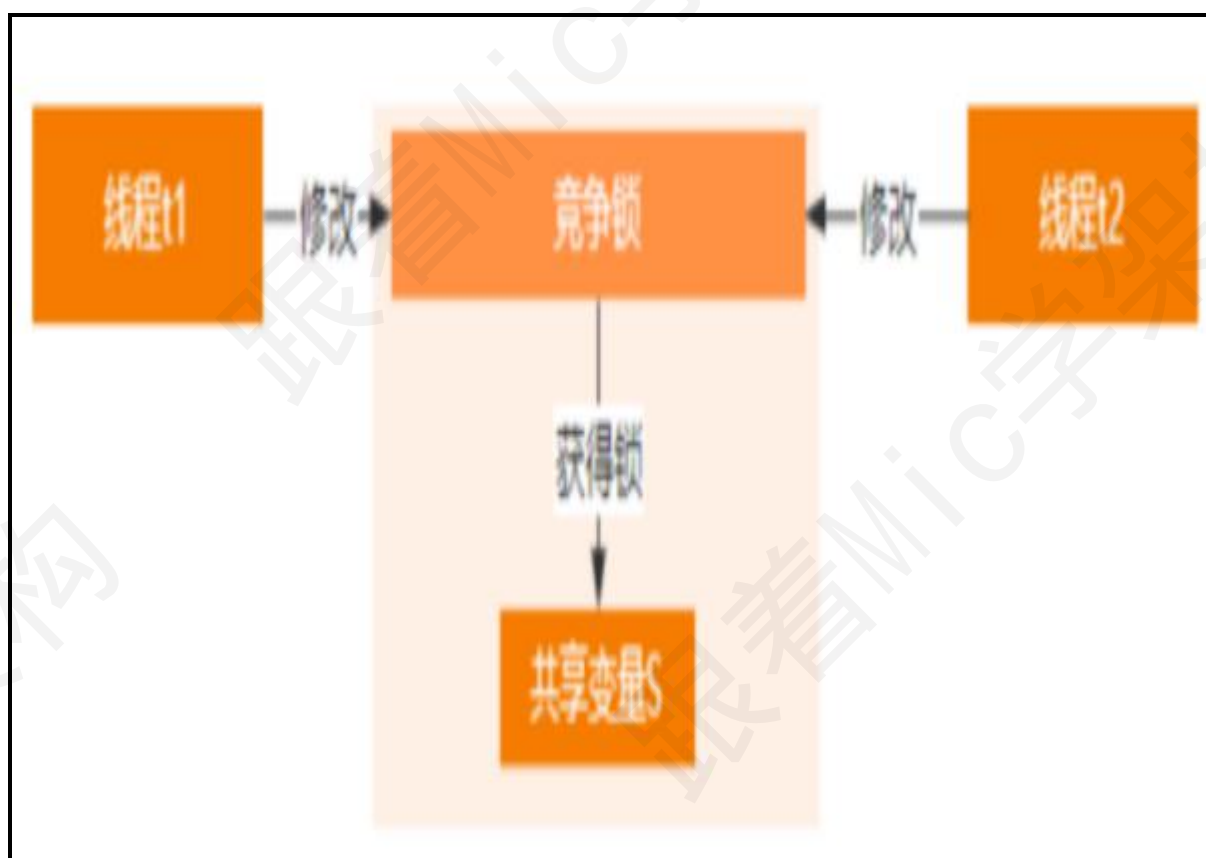
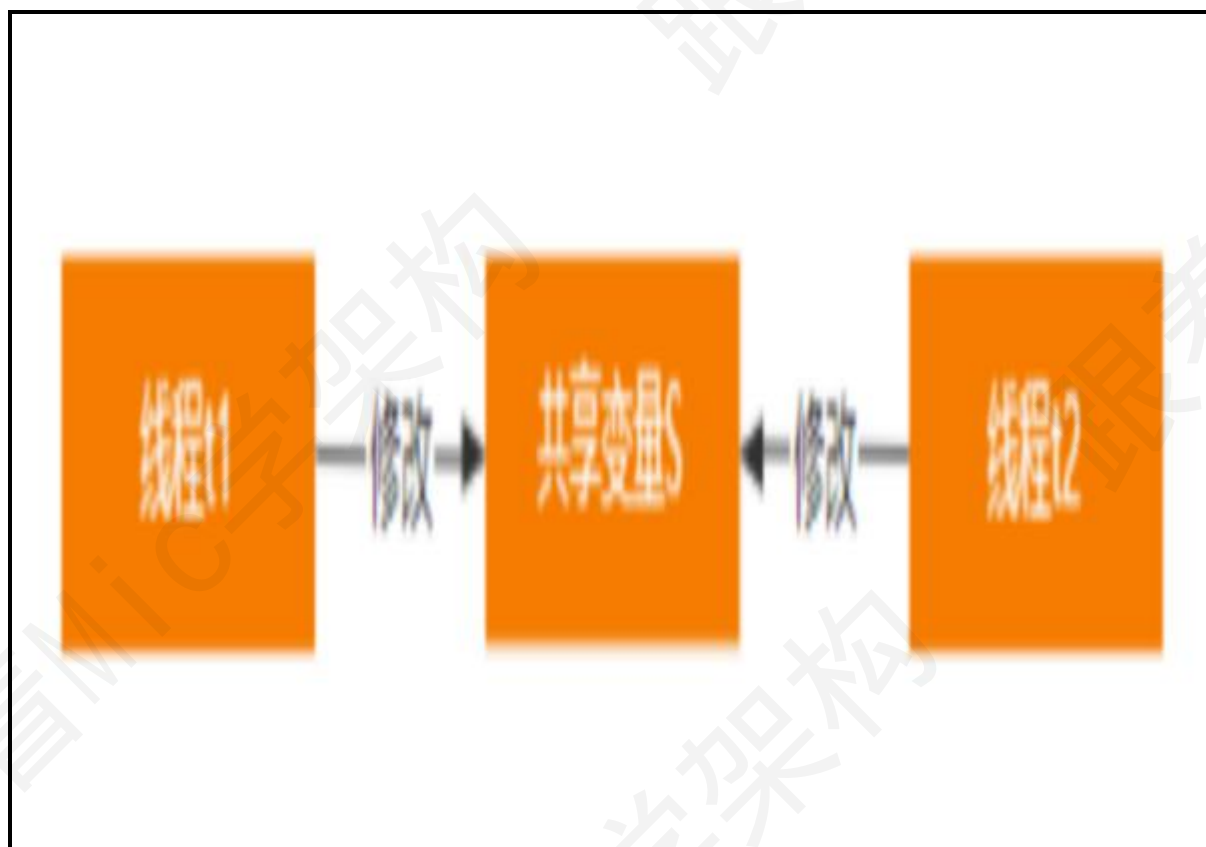
### 高手

---

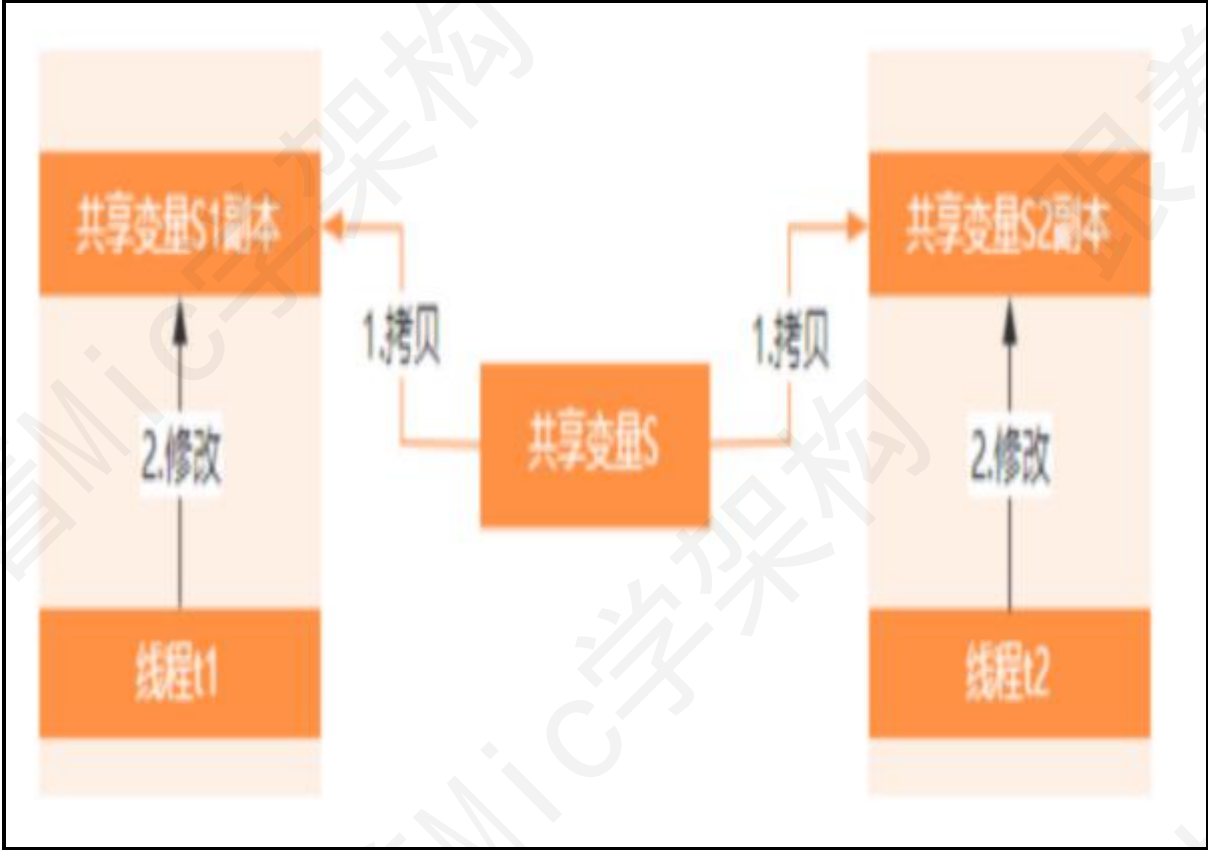
好的，这个问题我从三个方面来回答。

**ThreadLocal** 是一种线程隔离机制，它提供了多线程环境下对于共享变量访问的安全性。

在多线程访问共享变量的场景中（出现下面第一个图），一般的解决办法是对共享变量加锁（出现下面第二个图），从而保证在同一时刻只有一个线程能够对共享变量进行更新，并且基于 **Happens-Before** 规则里面的监视器锁规则，又保证了数据修改后对其他线程的可见性。



但是加锁会带来性能的下降，所以 `ThreadLocal` 用了一种空间换时间的设计思想，也就是说在每个线程里面，都有一个容器来存储共享变量的副本，然后每个线程只对自己的变量副本来做更新操作，这样既解决了线程安全问题，又避免了多线程竞争加锁的开销。



`ThreadLocal` 的具体实现原理是，在 `Thread` 类里面有一个成员变量 `ThreadLocalMap`，它专门来存储当前线程的共享变量副本，后续这个线程对于共享变量的操作，都是从这个 `ThreadLocalMap` 里面进行变更，不会影响全局共享变量的值。

以上就是我对这个问题的理解。

## 面试点评

`ThreadLocal` 使用场景比较多，比如在数据库连接的隔离、对于客户端请求会话的隔离等等。

在 `ThreadLocal` 中，除了空间换时间的设计思想以外，还有一些比较好的设计思想，比如线性探索解决 `hash` 冲突，数据预清理机制、弱引用 `key` 设计尽可能避免内存泄漏等。

这些思想在解决某些类似的业务问题时，都是可以直接借鉴的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，这些面试题我都整理成了笔记，大家有需要的可以私信获取。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 基于数组的阻塞队列 ArrayBlockingQueue 原理

---

今天来分享一道“饿了么”的高级工程师的面试题。

“基于数组的阻塞队列 ArrayBlockingQueue”的实现原理。

关于这个问题，我们来看看普通人和高手的回答。

### 普通人

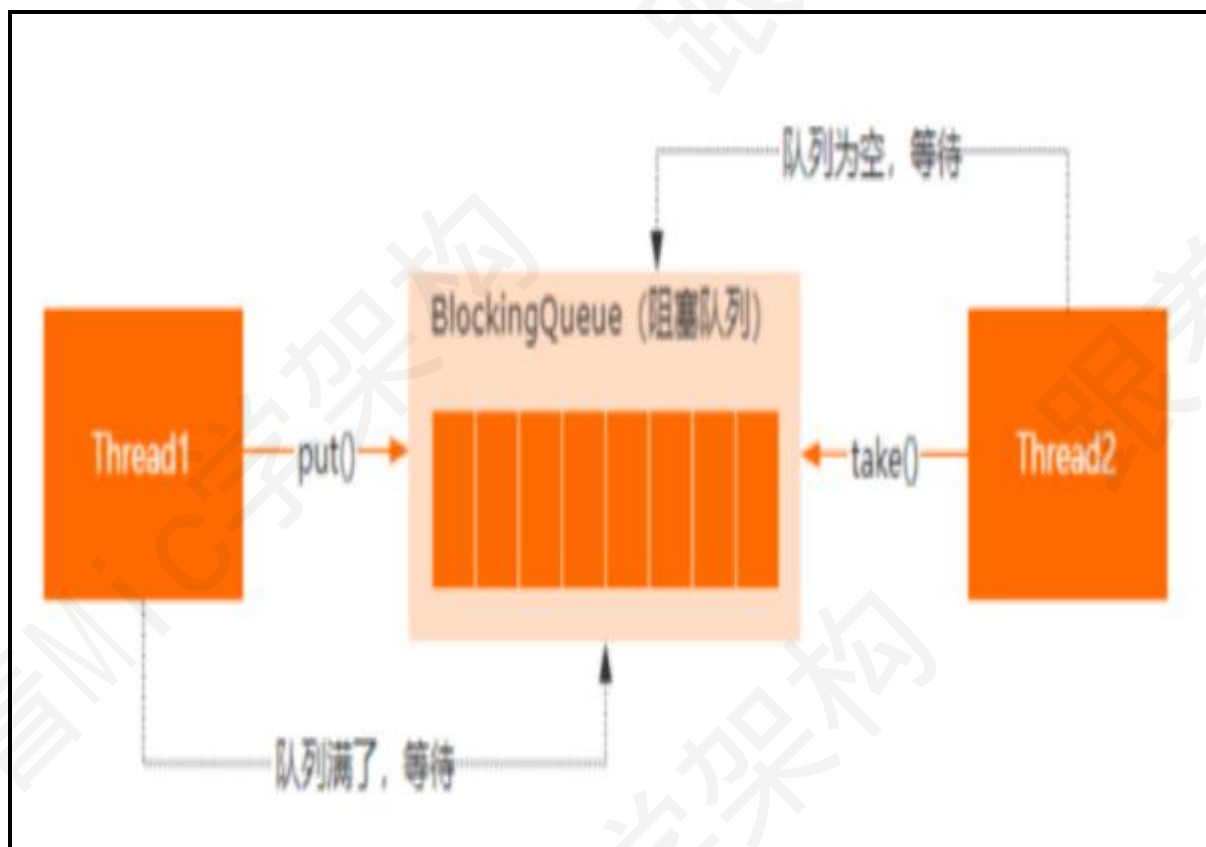
### 高手

---

阻塞队列（BlockingQueue）是在队列的基础上增加了两个附加操作，

在队列为空的时候，获取元素的线程会等待队列变为非空。

当队列满时，存储元素的线程会等待队列可用。



由于阻塞队列的特性，可以非常容易实现生产者消费者模型，也就是生产者只需要关心数据的生产，消费者只需要关注数据的消费，所以如果队列满了，生产者就等待，同样，队列空了，消费者也需要等待。

要实现这样的一个阻塞队列，需要用到两个关键的技术，队列元素的存储、以及线程阻塞和唤醒。

而 `ArrayBlockingQueue` 是基于数组结构的阻塞队列，也就是队列元素是存储在一个数组结构里面，并且由于数组有长度限制，为了达到循环生产和循环消费的目的，`ArrayBlockingQueue` 用到了循环数组。

而线程的阻塞和唤醒，用到了 `J.U.C` 包里面的 `ReentrantLock` 和 `Condition`。`Condition` 相当于 `wait/notify` 在 `JUC` 包里面的实现。

以上就是我对这个问题的理解。

## 面试点评

对于原理类的问题，有些小伙伴找不到切入点，不知道该怎么回答。

所谓的原理，通常说的是工作原理，比如对于 `ArrayBlockingQueue` 这个问题。

它的作用是在队列的基础上提供了阻塞添加和获取元素的能力，那么它的工作原理就是指用了什么设计方法或者技术来实现这样的功能，我们只要把这个部分说清楚就可以了。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，这些面试题我都整理成了笔记，大家有需要的可以私信获取。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 什么是聚集索引和非聚集索引

一个去阿里面试并且第一面就挂了的粉丝私信我，被数据库里面几个问题难倒了，他说面试官问了事务隔离级别、MVCC、聚集索引/非聚集索引、B 树、B+树这些，没回答好。

大厂面试基本上是这样，由点到面去展开，如果你对这个技术理解不够全面，很容易就会被看出来。

ok，关于“什么是聚集索引和非聚集索引”这个问题，看看普通人和高手的回答。

### 普通人

嗯，聚集索引就是通过主键来构建的索引结构。

而非聚集索引就是除了主键以外的其他索引。

### 高手

简单来说，聚集索引就是基于主键创建的索引，除了主键索引以外的其他索引，称为非聚集索引，也叫做二级索引。

由于在 InnoDB 引擎里面，一张表的数据对应的物理文件本身就是按照 B+树来组织的一种索引结构，而聚集索引就是按照每张表的主键来构建一颗 B+树，然后叶子节点里面存储了这个表的每一行数据记录。

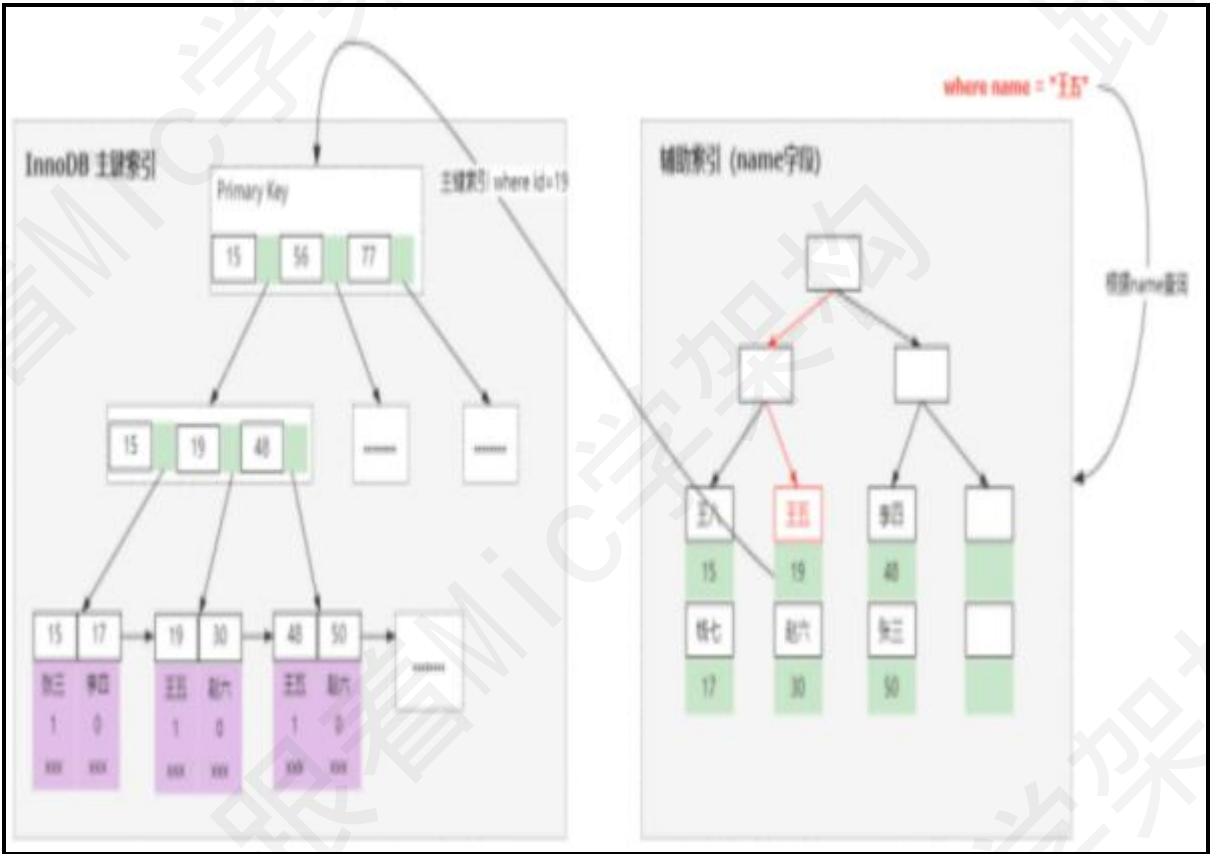
所以基于 InnoDB 这样的特性，聚集索引并不仅仅是一种索引类型，还代表着一种数据的存储方式。

同时也意味着每个表里面必须要有一个主键，如果没有主键，InnoDB 会默认选择或者添加一个隐藏列作为主键索引来存储这个表的数据行。一般情况是建议使用自增 id 作为主键，这样的话 id 本身具有连续性使得对应的数据也会按照顺序

存储在磁盘上，写入性能和检索性能都很高。否则，如果使用 `uuid` 这种随机 `id`，那么在频繁插入数据的时候，就会导致随机磁盘 `IO`，从而导致性能较低。

需要注意的是，`InnoDB` 里面只能存在一个聚集索引，原因很简单，如果存在多个聚集索引，那么意味着这个表里面的数据存在多个副本，造成磁盘空间的浪费，以及数据维护的困难。

由于在 `InnoDB` 里面，主键索引表示的是一种数据存储结构，所以如果是基于非聚集索引来查询一条完整的记录，最终还是需要访问主键索引来检索。



## 面试点评

这个问题要回答好，还真不容易。涉及到 `Mysql` 里面索引的实现原理。

但是如果回答好了，就能够很好的反馈求职者的技术功底，那通过面试就比较容易了。

好的，本期的普通人 `VS` 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，这些面试题我都整理成了笔记，大家有需要的可以私信获取。

我是 `Mic`，一个工作了 `14` 年的 `Java` 程序员，咱们下期再见。

# 什么是双亲委派？

---

Hi，大家好。

今天我们来分享一道关于 **Java** 类加载方面的面试题。在国内的一二线互联网公司面试的时候，面试官通常是使用这方面的问题来暖场，但往往造成的是冷场~

比如，什么是双亲委派？什么是类加载？`new String()`生成了几个对象等等。

双亲委派的英文是 **parent delegation model**，我认为从真正的实现逻辑来看，正确的翻译应该是父委托模型。

不管它叫什么，我们先来看看遇到这个问题应该怎么回答。

## 普通人的回答

## 高手的回答

---

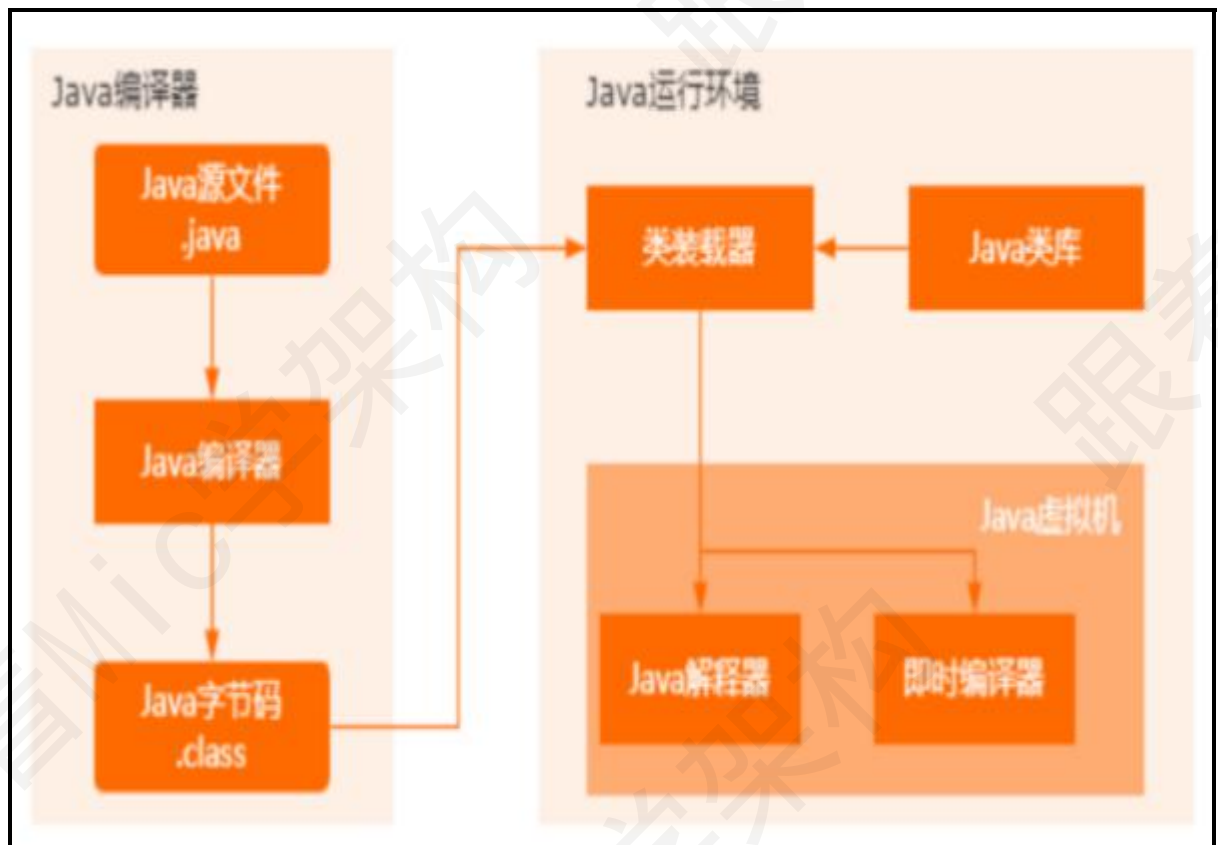
关于这个问题，需要从几个方面来回答。

首先，我简单说一下类的加载机制，就是我们自己写的 **java** 源文件到最终运行，必须要经过编译和类加载两个阶段。

编译的过程就是把 **java** 文件编译成 **class** 文件。

类加载的过程，就是把 **class** 文件装载到 **JVM** 内存中，装载完成以后就会得到一个 **Class** 对象，我们就可以使用 **new** 关键字来实例化这个对象。





而类的加载过程，需要涉及到类加载器。

JVM 在运行的时候，会产生 3 个类加载器，这三个类加载器组成了一个层级关系

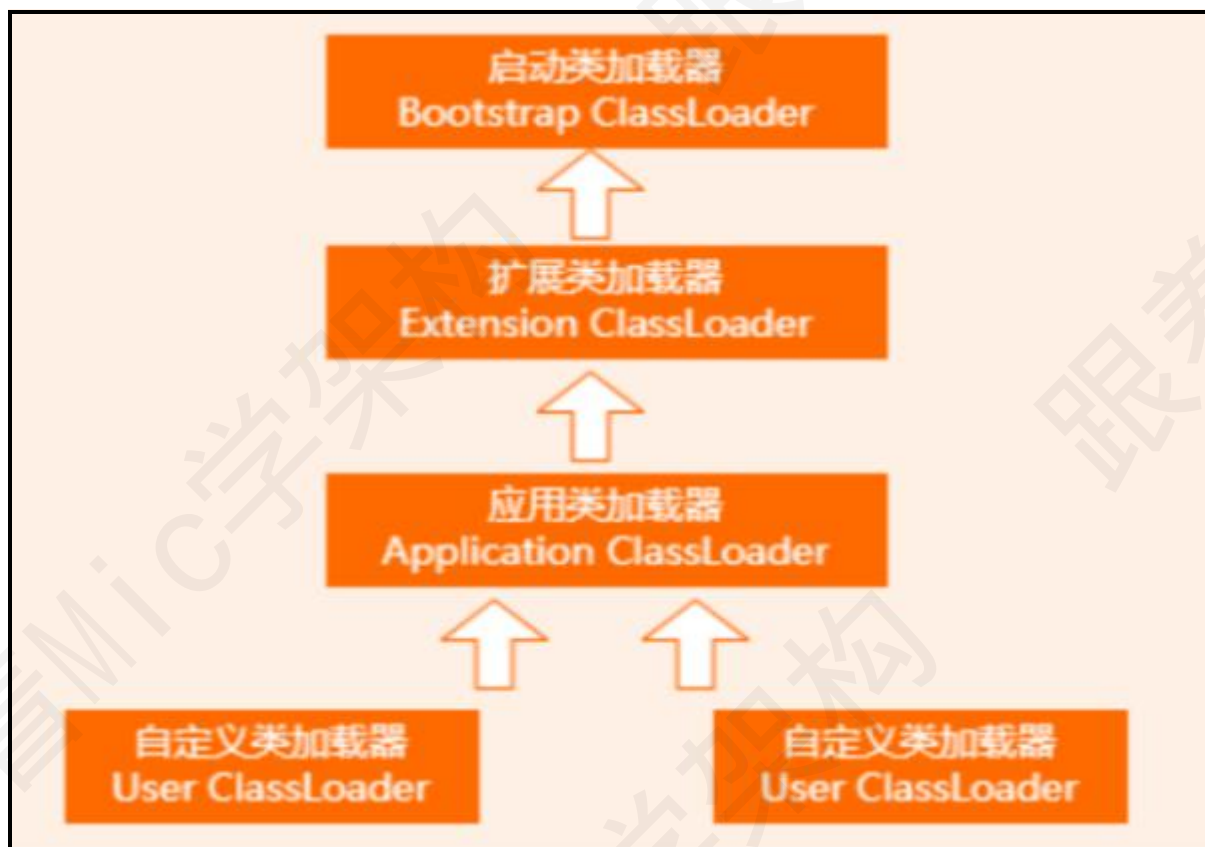
每个类加载器分别去加载不同作用范围的 jar 包，比如

**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

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

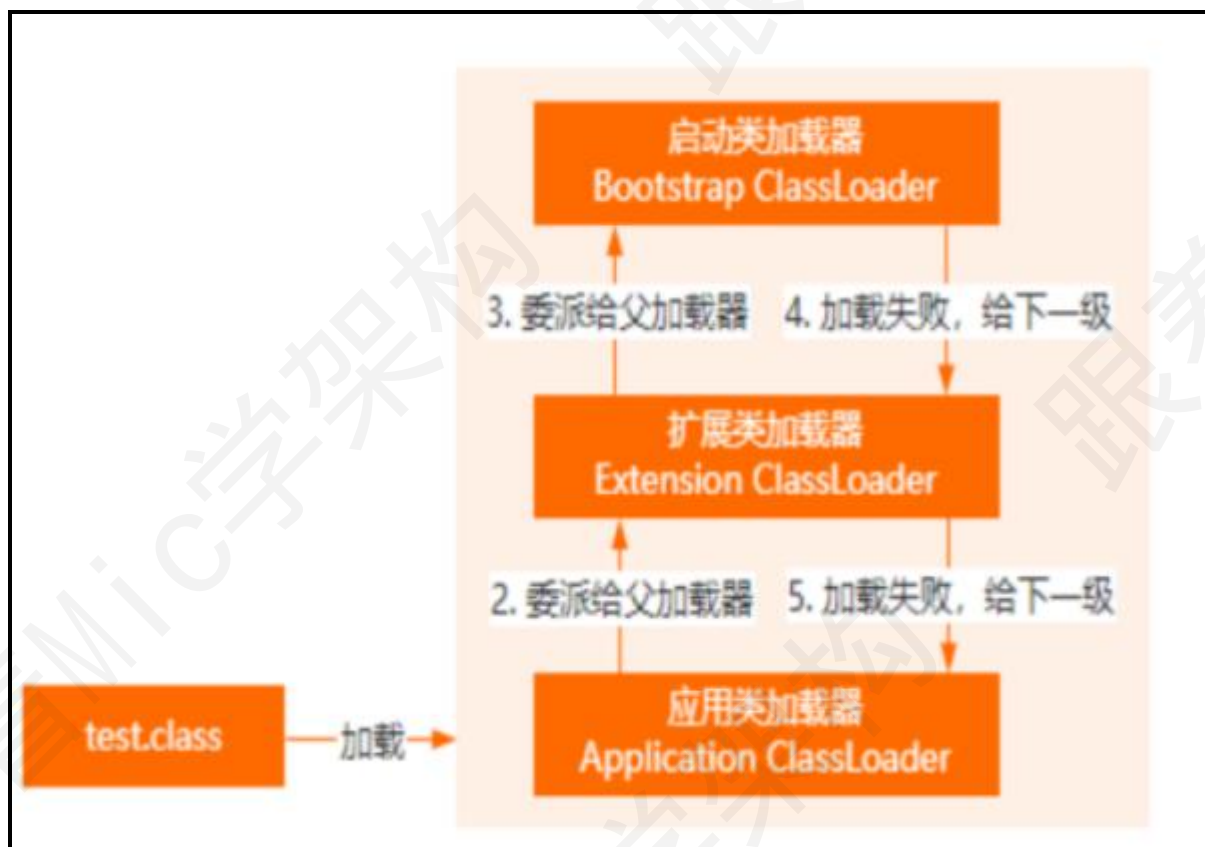
**Application ClassLoader**，主要负责当前应用里面的 `classpath` 下的所有 jar 包和类文件

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。



所谓的父委托模型，就是按照类加载器的层级关系，逐层进行委派。

比如当需要加载一个 `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`。



这样设计的好处，我认为有几个。

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`，最终还是会交给启动类加载器。再加上每个类加载器的作用范围，那么自己写的 `java.lang.String` 就没办法去覆盖类库中类。

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，因为如果父加载器已经加载过了，那么子类就没必要去加载了。

以上就是我对这个问题的理解。

## 面试点评

**JVM** 虚拟机一定面试必问的领域，因为我们自己写的程序运行在 **JVM** 上，一旦出现问题，你不理解，就无法排查。

就像一个修汽车的工人，他不知道汽车的工作原理，不懂发动机，那他是无法做好这份工作的。

好的，本期的普通人 **VS** 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，这些面试题我都整理成了笔记，大家有需要的可以私信获取。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 怎么理解线程安全？

---

Hi，大家好，我是 Mic

一个工作了 4 年的小伙伴，遇到了一个非常抽象的面试题，说说你对线程安全性的理解。

这类问题，对于临时刷面试题来面试的小伙伴，往往是致命的。

一个是不知道从何说起，也就是语言组织比较困难。

其次就是，如果对于线程安全性没有一定程度的理解，一般很难说出你的理解。

ok，我们来看看这个问题的回答。

### 普通人

### 高手

---

简单来说，在多个线程访问某个方法或者对象的时候，不管通过任何的方式调用以及线程如何去交替执行。

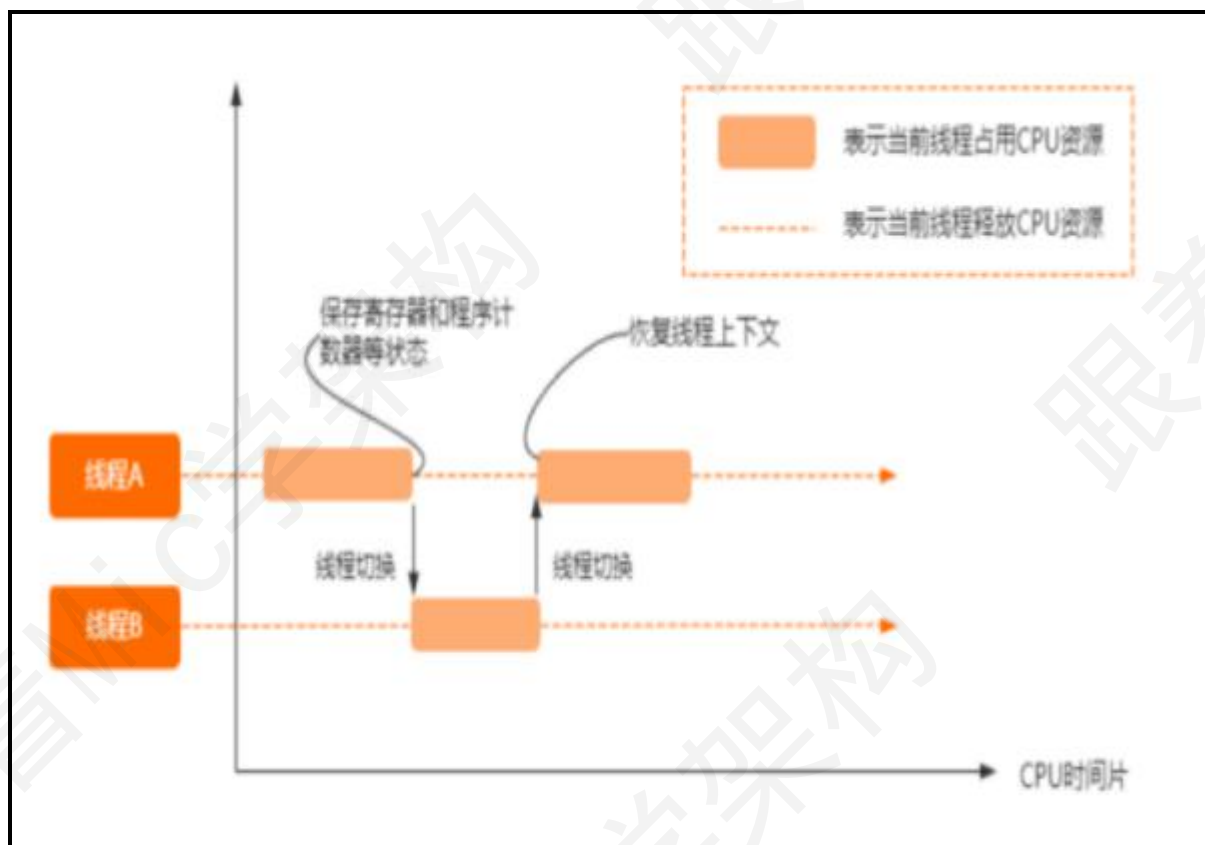
在程序中不做任何同步干预操作的情况下，这个方法或者对象的执行/修改都能按照预期的结果来反馈，那么这个类就是线程安全的。

实际上，线程安全问题的具体表现体现在三个方面，原子性、有序性、可见性。

原子性呢，是指当一个线程执行一系列程序指令操作的时候，它应该是不可中断的，因为一旦出现中断，站在多线程的视角来看，这一系列的程序指令会出现前后执行结果不一致的问题。

这个和数据库里面的原子性是一样的，简单来说就是一段程序只能由一个线程完整的执行完成，而不能存在多个线程干扰。

CPU 的上下文切换，是导致原子性问题的核心，而 JVM 里面提供了 Synchronized 关键字来解决原子性问题。



可见性，就是说的多线程环境下，由于读和写是发生在不同的线程里面，有可能出现某个线程对共享变量的修改，对其他线程不是实时可见的。

导致可见性问题的原因有很多，比如 CPU 的高速缓存、CPU 的指令重排序、编译器的指令重排序。

有序性，指的是程序编写的指令顺序和最终 CPU 运行的指令顺序可能出现不一致的现象，这种现象也可以称为指令重排序，所以有序性也会导致可见性问题。

可见性和有序性可以通过 JVM 里面提供了一个 **Volatile** 关键字来解决。

在我看来，导致有序性、原子性、可见性问题的本质，是计算机工程师为了最大化提升 CPU 利用率导致的。比如为了提升 CPU 利用率，设计了三级缓存、设计了 **StoreBuffer**、设计了缓存行这种预读机制、在操作系统里面，设计了线程模型、在编译器里面，设计了编译器的深度优化机制。

一上就是我对这个问题的理解。

## 面试点评

从高手的回答中，可以很深刻的感受到，他对于计算机底层原理和线程安全性相关的底层实现是理解得很透彻的。

对我来说，这个人去写程序代码，不用担心他滥用线程导致一些不可预测的线程安全性问题了，这就是这个面试题的价值。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，喜欢的朋友记得点赞和收藏。

另外，这些面试题我都整理成了笔记，大家有需要的可以私信获取。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 如果让你设计一个秒杀系统，怎么设计？

普通人

高手

我认为秒杀系统的核心有两个

过滤掉 90% 以上的无效流量

解决库存超卖的问题

面试点评

## 请你说一下数据库优化

热点账户处理（如平台账户汇总手续费）

1. 新建在途资金表 2. 之前业务中操作平台账户加资金改为插入在途资金表一条记录 3. 定时汇总在途资金表，每条在途资金一条流水，但是修改资金一次 update

中台热点账户只记流水不记余额单独的线程通过流水记余额余额不时时更新

<https://gper.club/articles/7e7e7f7ff7g5egc7g68>

## 如果问你项目的重点和难点，该如何回答呢？

<https://zhuanlan.zhihu.com/p/265070762>

# 请简述一下伪共享的概念以及如何避免

一个工作 5 年的小伙伴，2 个星期时间，去应聘了 10 多家公司，结果每次在技术面试环节，被技术原理难倒了。

他现在很尴尬，简历投递出去就像石沉大海，基本没什么面试邀约。

技术能力的提升也不是一两天的事情，所以他现在特别焦虑。

于是，我从他遇到过的面试题里面抽出来一道，给大家分享一下。

“请简述一下伪共享的概念以及避免的方法”

对于这个问题，看看高手该怎么回答。

## 普通人

临场发挥

## 高手

对于这个问题，要从几个方面来回答。

首先，计算机工程师为了提高 CPU 的利用率，平衡 CPU 和内存之间的速度差异，在 CPU 里面设计了三级缓存。

CPU 在向内存发起 IO 操作的时候，一次性会读取 64 个字节的数据作为一个缓存行，缓存到 CPU 的高速缓存里面。

在 Java 中一个 long 类型是 8 个字节，意味着一个缓存行可以存储 8 个 long 类型的变量。

这个设计是基于空间局部性原理来实现的，也就是说，如果一个存储器的位置被引用，那么将来它附近的位置也会被引用。

所以缓存行的设计对于 CPU 来说，可以有效的减少和内存的交互次数，从而避免了 CPU 的 IO 等待，以提升 CPU 的利用率。

正是因为这种缓存行的设计，导致如果多个线程修改同一个缓存行里面的多个独立变量的时候，基于缓存一致性协议，就会无意中影响了彼此的性能，这就是伪共享的问题。

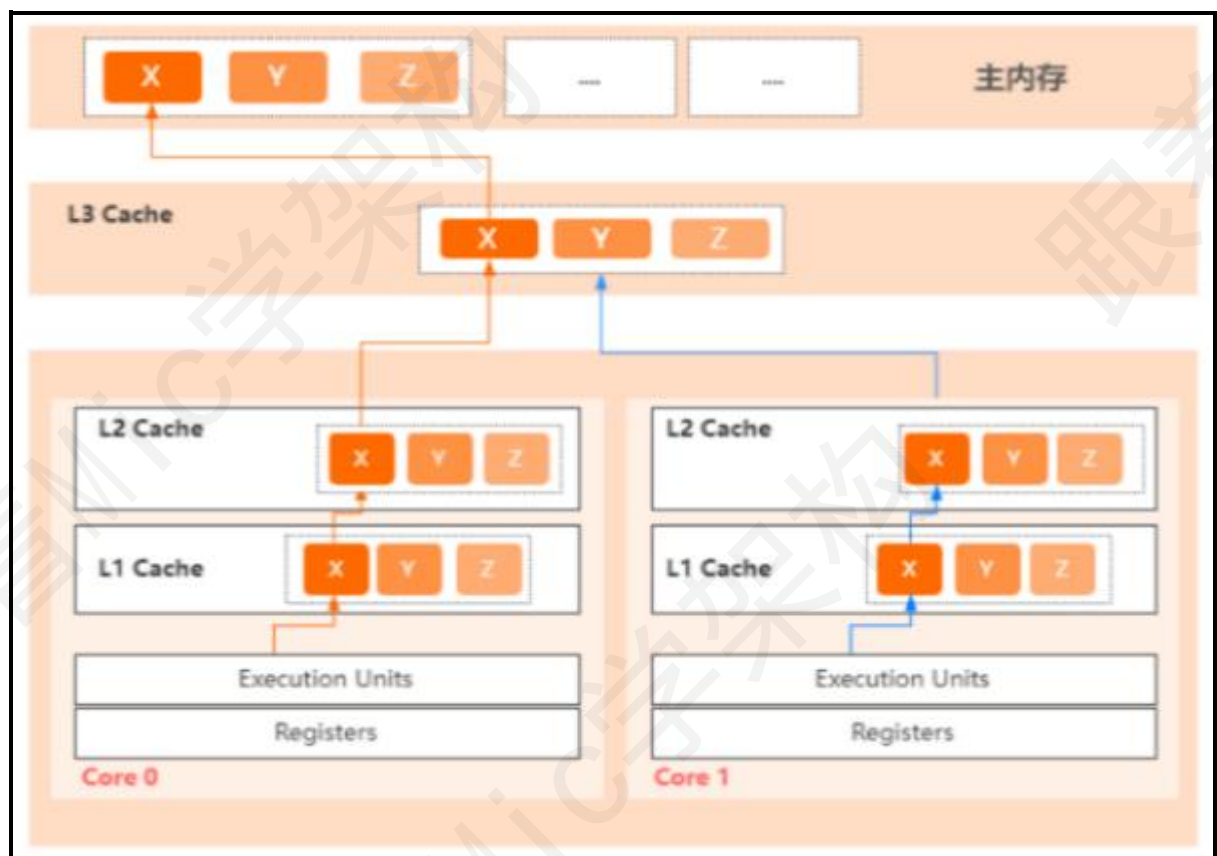
像这样一种情况，CPU0 上运行的线程想要更新变量 X、CPU1 上的线程想要更新变量 Y，而 X/Y/Z 都在同一个缓存行里面。

每个线程都需要去竞争缓存行的所有权对变量做更新，基于缓存一致性协议。



一旦运行在某个 CPU 上的线程获得了所有权并执行了修改，就会导致其他 CPU 中的缓存行失效。

这就是伪共享问题的原理。



因为伪共享会问题导致缓存锁的竞争，所以在并发场景中的程序执行效率一定会收到较大的影响。

这个问题的解决办法有两个：

使用对齐填充，因为一个缓存行大小是 64 个字节，如果读取的目标数据小于 64 个字节，可以增加一些无意义的成员变量来填充。

在 Java8 里面，提供了 `@Contended` 注解，它也是通过缓存行填充来解决伪共享问题的，被 `@Contended` 注解声明的类或者字段，会被加载到独立的缓存行上。

已上就是我对这个问题的理解！

## 面试点评

在 Netty 里面，有大量用到对齐填充的方式来避免伪共享问题。

所以这并不是一个所谓超纲的问题，在我看来，多线程也好、数据结构算法也好、还是 JVM，这个一个



合格的 **Java** 程序员必须要掌握的基础。

我们习惯了在框架里面写代码，却忽略了各种成熟框架已经让 **Java** 程序员变得越来越普通。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

有任何不懂的技术面试题，欢迎随时私信我

我是 Mic，一个工作了 14 年的 **Java** 程序员，咱们下期再见。

## 为什么要使用 **Spring** 框架？

---

一个工作了 4 年的小伙伴，他说他从线下培训就开始接触 **Spring**，到现在已经快 5 年时间了。

从来没有想过，为什么要使用 **Spring** 框架。

结果在面试的时候，竟然遇到一个这样的问题。

大脑一时间短路了，来求助我，这类问题应该怎么去回答。

下面我们来看看普通人和高手的回答

### 普通人

### 高手

---

**Spring** 是一个轻量级应用框架，它提供了 **IoC** 和 **AOP** 这两个核心的功能。

它的核心目的是为了简化企业级应用程序的开发，使得开发者只需要关心业务需求，不需要关心 **Bean** 的管理，

以及通过切面增强功能减少代码的侵入性。

从 **Spring** 本身的特性来看，我认为有几个关键点是我们选择 **Spring** 框架的原因。

轻量：**Spring** 是轻量的，基本的版本大约 2MB。

**IOC/DI**：**Spring** 通过 **IOC** 容器实现了 **Bean** 的生命周期的管理，以及通过 **DI** 实现依赖注入，从而实现了对象依赖的松耦合管理。

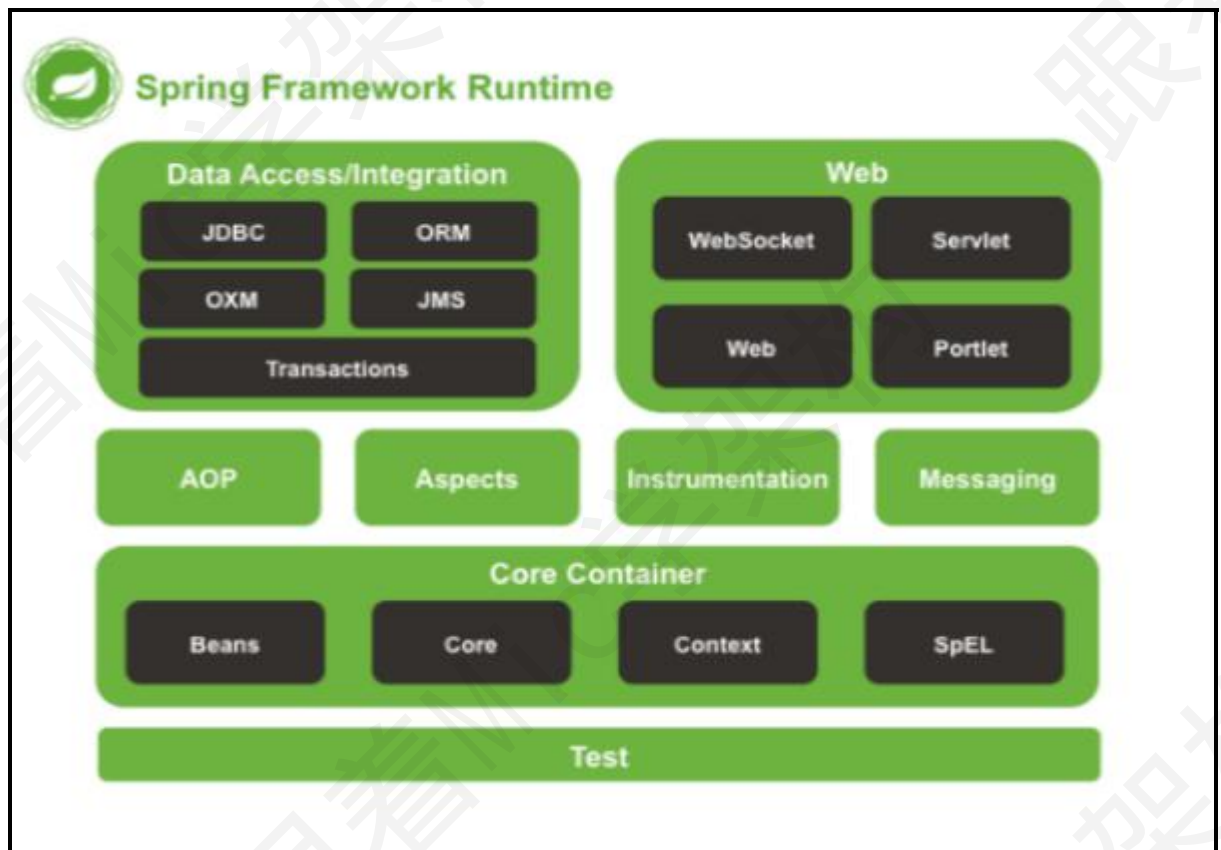
面向切面的编程(**AOP**)：**Spring** 支持面向切面的编程，从而把应用业务逻辑和系统服务分开。

**MVC** 框架：**Spring MVC** 提供了功能更加强大且更加灵活的 **Web** 框架支持

事务管理：Spring 通过 AOP 实现了事务的统一管理，对应用开发中的事务处理提供了非常灵活的支持

最后，Spring 从第一个版本发布到现在，它的生态已经非常庞大了。在业务开发领域，Spring 生态几乎提供了

非常完善的支持，更重要的是社区的活跃度和技术的成熟度都非常高，以上就是我对这个问题的理解。



## 面试点评

任何一个技术框架，一定是为了解决某些特定的问题，只是大家忽视了这个点。

为什么要用，再往高一点来说，其实就是技术选型，能回答这个问题，

意味着面对业务场景或者技术问题的解决方案上，会有自己的见解和思考。

所以，我自己也喜欢在面试的时候问这一类的问题。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

有任何不懂的技术面试题，欢迎随时私信我

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

# Spring 中事务的传播行为有哪些？

---

一个工作了 2 年的粉丝，私信了一个比较简单的问题。

说：“Spring 中事务的传播行为有哪些？”

他说他能记得一些，但是在项目中基本上不需要配置，所以一下就忘记了。

结果导致面试被拒绝，有点遗憾！

ok，关于这个问题，看看普通人和高手的回答。

## 普通人

## 高手

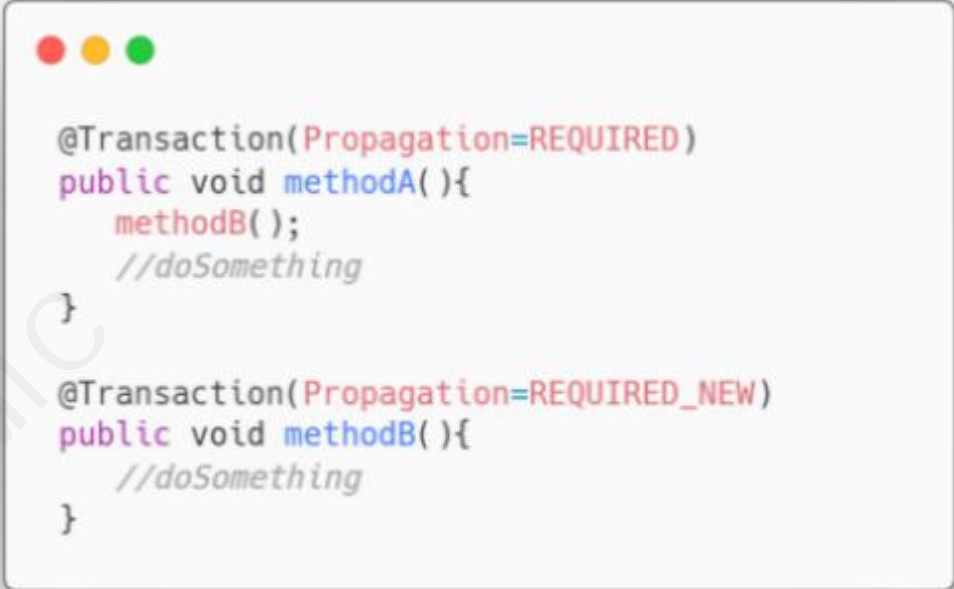
---

对于这个问题，需要从几个方面去回答。

首选，所谓的事务传播行为，就是多个声明了事务的方法相互调用的时候，这个事务应该如何传播。

比如说，`methodA()`调用 `methodB()`，两个方法都显示的开启了事务。

那么 `methodB()` 是开启一个新事务，还是继续在 `methodA()` 这个事务中执行？就取决于事务的传播行为。



```
@Transaction(Propagation=REQUIRED)
public void methodA(){
    methodB();
    //doSomething
}

@Transaction(Propagation=REQUIRED_NEW)
public void methodB(){
    //doSomething
}
```

在 Spring 中，定义了 7 种事务传播行为。

**REQUIRED:** 默认的 Spring 事物传播级别，如果当前存在事务，则加入这个事务，如果不存在事务，就新建一个事务。

**REQUIRED\_NEW:** 不管是否存在事务，都会新开一个事务，新老事务相互独立。外部事务抛出异常回滚不会影响内部事务的正常提交。

**NESTED:** 如果当前存在事务，则嵌套在当前事务中执行。如果当前没有事务，则新建一个事务，类似于 **REQUIRED\_NEW**。

**SUPPORTS:** 表示支持当前事务，如果当前不存在事务，以非事务的方式执行。

**NOT\_SUPPORTED:** 表示以非事务的方式来运行，如果当前存在事务，则把当前事务挂起。

**MANDATORY:** 强制事务执行，若当前不存在事务，则抛出异常。

**NEVER:** 以非事务的方式执行，如果当前存在事务，则抛出异常。

Spring 事务传播级别一般不需要定义，默认就是 **PROPAGATION\_REQUIRED**，除非在嵌套事务的情况下需要重点了解。

以上就是我对这个问题的理解！

## 面试点评

---

这个问题其实只需要理解事务传播行为的本质以及为什么需要考虑到事务传播的问题。

就可以直接基于自身的技术积累来推演出答案，无非就是基于可能的策略进行穷举，怎么也能推演出 5 种吧。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

有任何不懂的技术面试题，欢迎随时私信我

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Dubbo 是如何动态感知服务下线的？

---

什么情况？一个工作了 5 年的 Java 程序员，竟然无法回答出这个问题？

“Dubbo 是如何动态感知服务下线的”？

好吧，对于这个问题，看看普通人和高手的回答！

### 普通人

### 高手

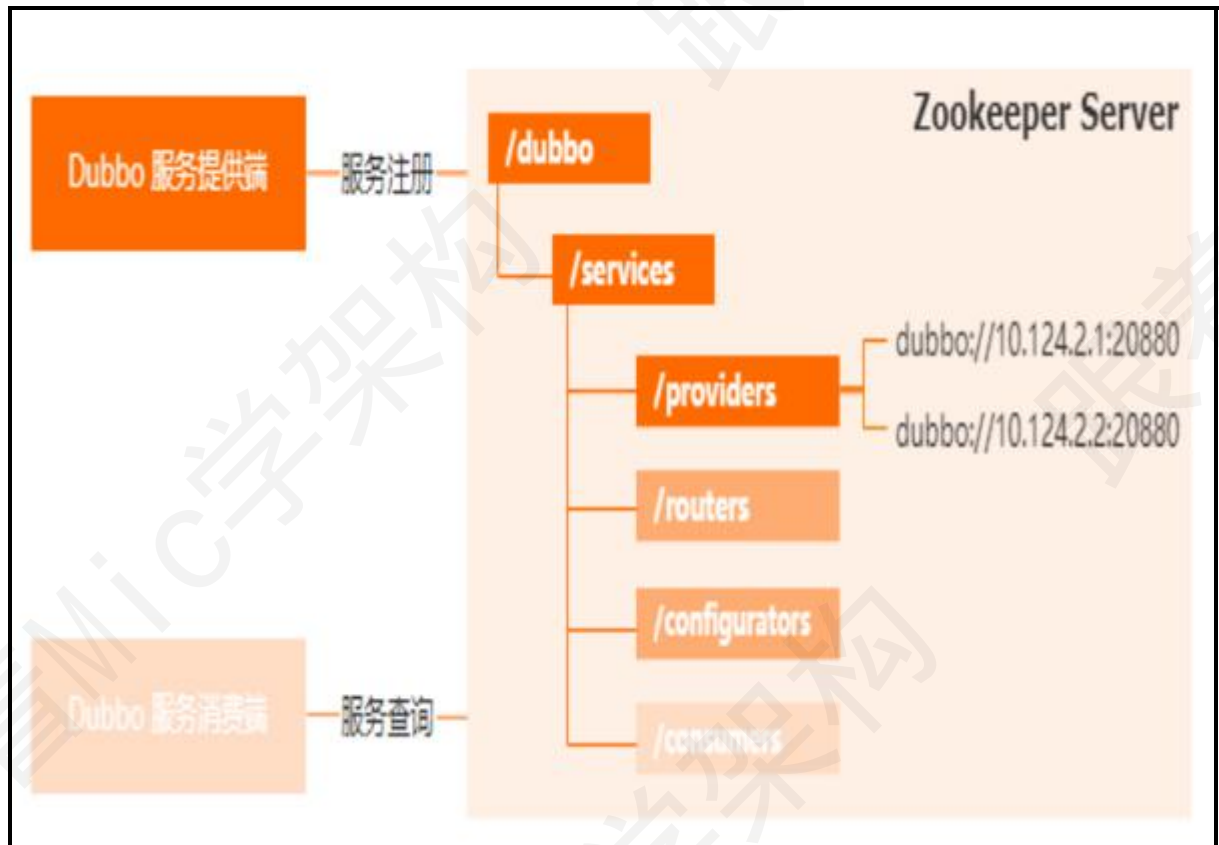
---

好的，面试官，关于这个问题，我从几个方面来回答。

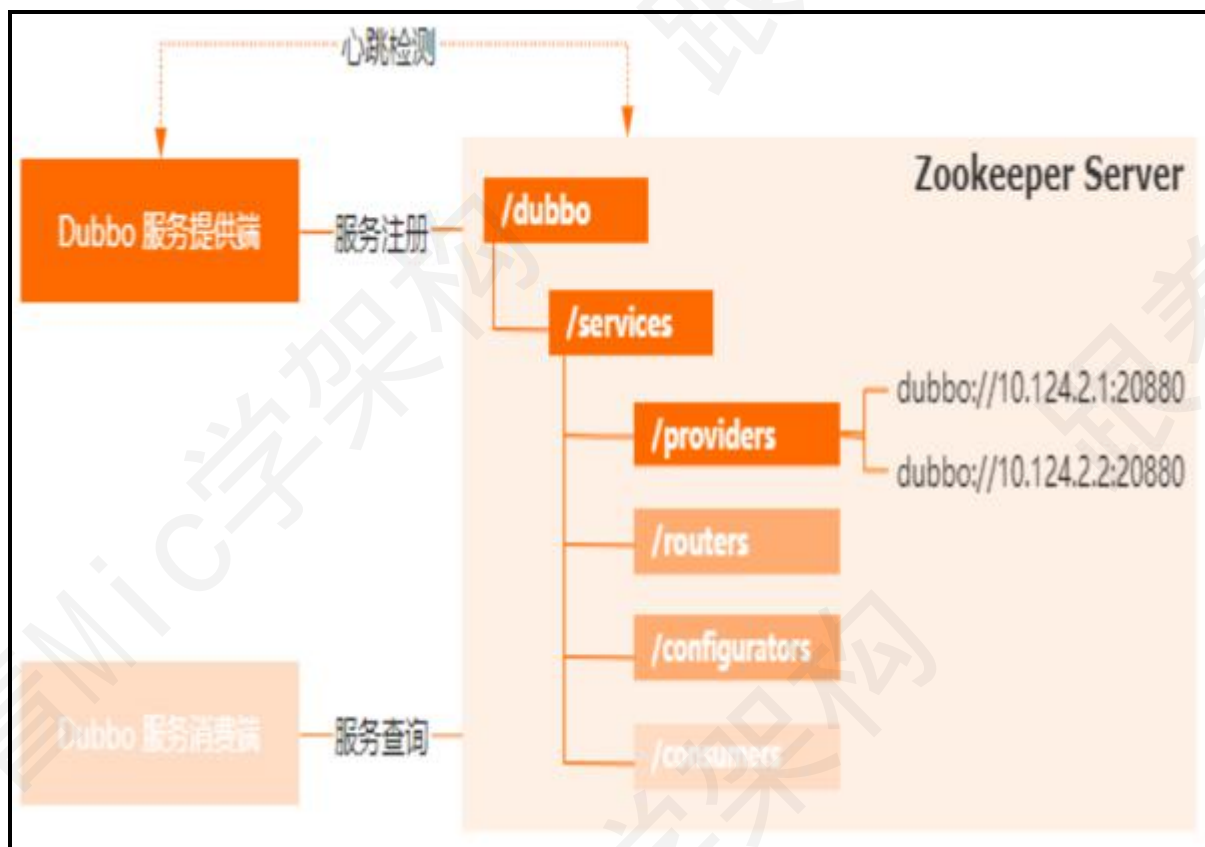
首先，Dubbo 默认采用 Zookeeper 实现服务的注册与服务发现，简单来说啊，就是多个 Dubbo 服务之间的通信地址，是使用 Zookeeper 来维护的。

在 Zookeeper 上，会采用树形结构的方式来维护 Dubbo 服务提供端的协议地址，

Dubbo 服务消费端会从 Zookeeper Server 上去查找目标服务的地址列表，从而完成服务的注册和消费的功能。



Zookeeper 会通过心跳检测机制，来判断 Dubbo 服务提供端的运行状态，来决定是否应该把这个服务从地址列表剔除。

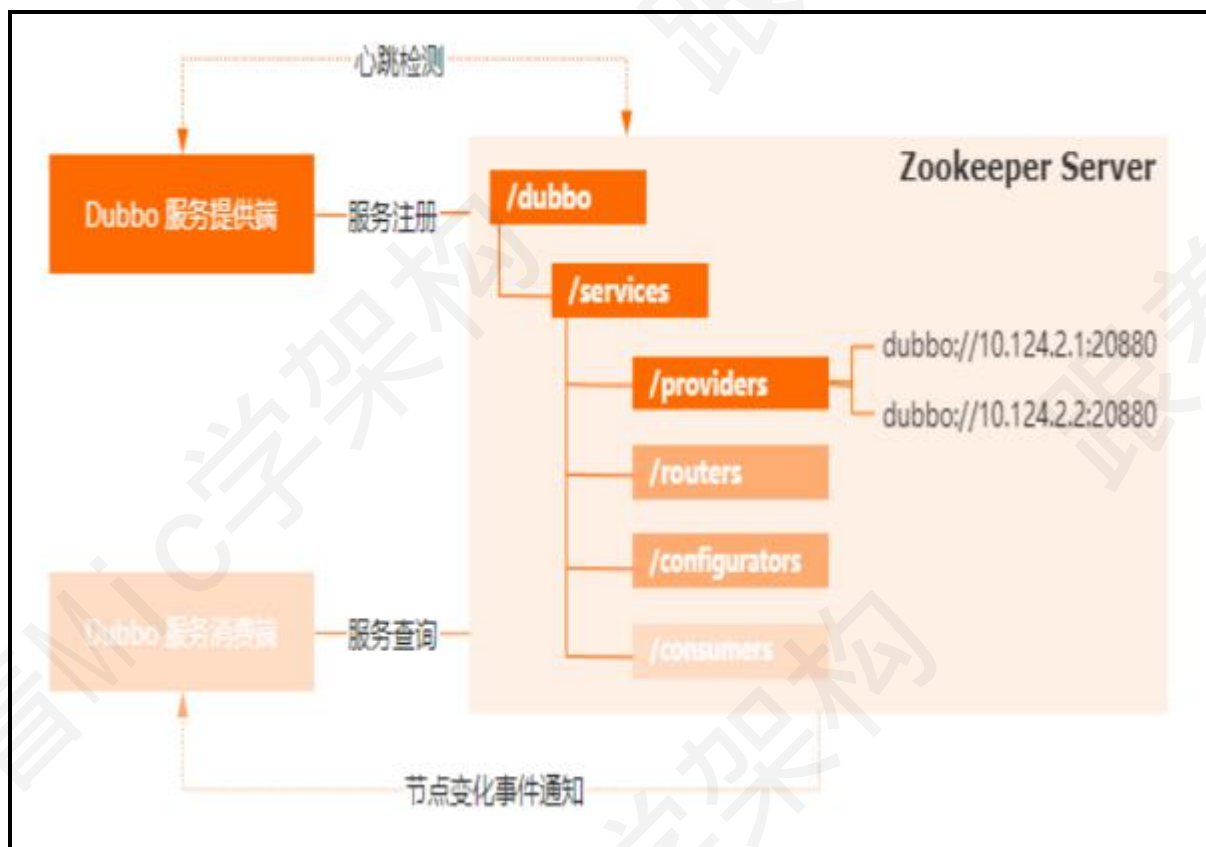


当 Dubbo 服务提供方出现故障导致 Zookeeper 剔除了这个服务的地址，那么 Dubbo 服务消费端需要感知到地址的变化，从而避免后续的请求发送到故障节点，导致请求失败。

也就是说 Dubbo 要提供服务下线的动态感知能力。

这个能力是通过 Zookeeper 里面提供的 Watch 机制来实现的，简单来说呢，Dubbo 服务消费端会使用 Zookeeper 里面的 Watch 来针对 Zookeeper Server 端的 `/providers` 节点注册监听，

一旦这个节点下的子节点发生变化，Zookeeper Server 就会发送一个事件通知 Dubbo Client 端。



Dubbo Client 端收到事件以后，就会把本地缓存的这个服务地址删除，这样后续就不会把请求发送到失败的节点上，完成服务下线感知。

以上就是我对这个问题的理解！

## 面试点评

Dubbo 是目前非常主流的开源 RPC 框架，在很多的企業都有使用。

理解这个 RPC 底层的工作原理很有必要，它能帮助开发者提高开发问题的解决效率。

还是想多说一句，在 Java 这个岗位上，如果想走得更远，一定要花苦功夫。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

有任何不懂的技术面试题，欢迎随时私信我

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Spring 中 Bean 的作用域有哪些？

一个工作 3 年的小伙子，去面试被问到 Spring 里面的问题。



这个问题比较简单，但是他却没有回答上来。

虽然他可以通过搜索引擎找到答案，但是如果没有理解，下次面试还是不会！

这个面试题是：“Spring 中的 Bean，作用域有哪些？”

对于这个问题，看看普通人和高手的回答。

## 普通人

## 高手

---

好的，这个问题可以从几个方面来回答。

首先呢，Spring 框架里面的 IOC 容器，可以非常方便的去帮助我们管理应用里面的 Bean 对象实例。

我们只需要按照 Spring 里面提供的 xml 或者注解等方式去告诉 IOC 容器，哪些 Bean 需要被 IOC 容器管理就行了。

其次呢，既然是 Bean 对象实例的管理，那意味着这些实例，是存在生命周期，也就是所谓的作用域。

理论上来说，常规的生命周期只有两种：

singleton，也就是单例，意味着在整个 Spring 容器中只会存在一个 Bean 实例。

prototype，翻译成原型，意味着每次从 IOC 容器去获取指定 Bean 的时候，都会返回一个新的实例对象。

但是在基于 Spring 框架下的 Web 应用里面，增加了一个会话纬度来控制 Bean 的生命周期，主要有三个选择

request，针对每一次 http 请求，都会创建一个新的 Bean

session，以 session 会话为纬度，同一个 session 共享同一个 Bean 实例，不同的 session 产生不同的 Bean 实例

globalSession，针对全局 session 纬度，共享同一个 Bean 实例

以上就是我对这个问题的理解。

## 面试点评

---

“技术框架的本质是去解决特定问题的，所以如果能够站在技术的角度去思考 Spring”

当遇到这种问题的时候，就可以像这个高手的回答一样，能够基于场景来推断出答案。

就像我们现在写 CRUD 代码，它已经变成了一种基本能力去让我们完成复杂业务逻辑的开发。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

有任何不懂的技术面试题，欢迎随时私信我

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Zookeeper 中的 Watch 机制的原理？

---

一个工作了 7 年的粉丝，遇到了一个 Zookeeper 的问题。

因为接触过 Zookeeper 这个技术，不知道该怎么回答。

我说一个工作了 7 年的程序员，没有接触过主流技术，这不正常。

于是我问了他工资以后，我理解了！

好吧，关于“Zookeeper 中 Watch 机制的实现原理”，看看普通人和高手的回答。

### 普通人

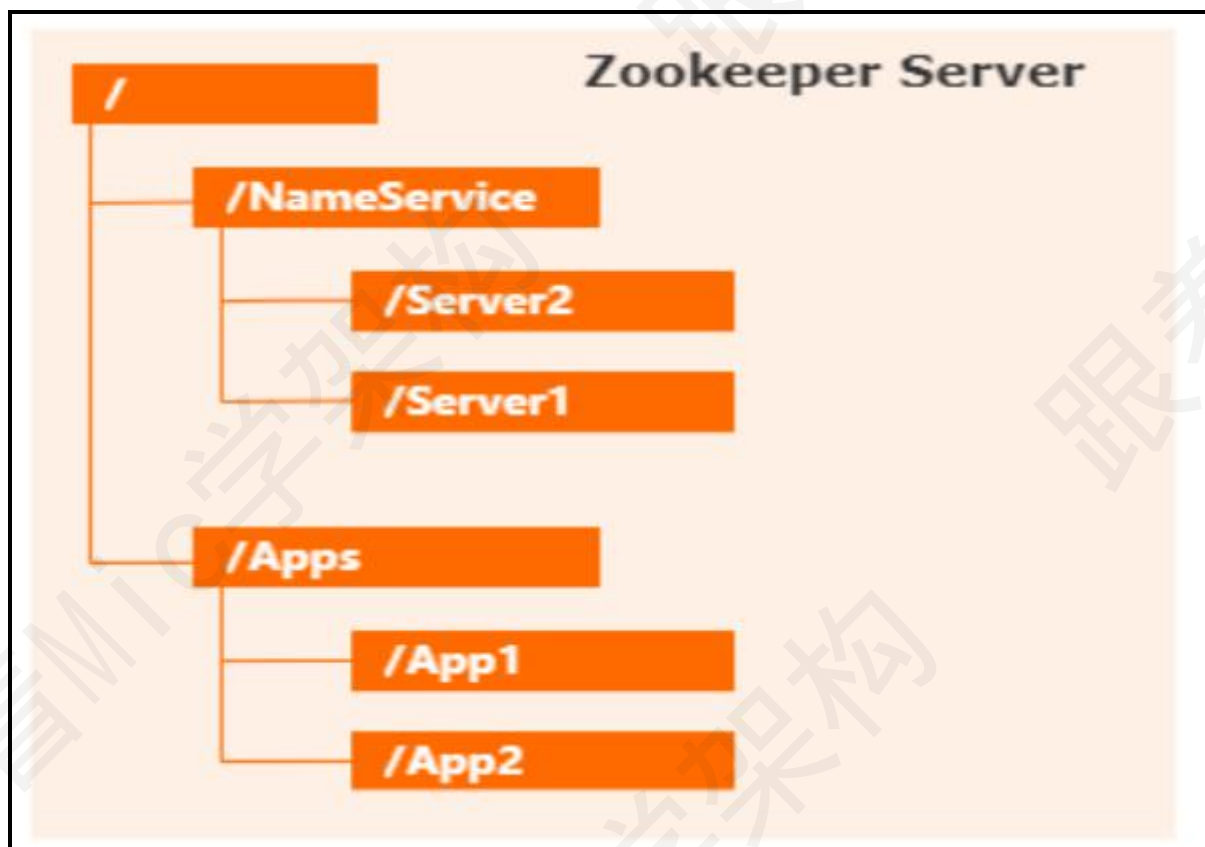
### 高手

---

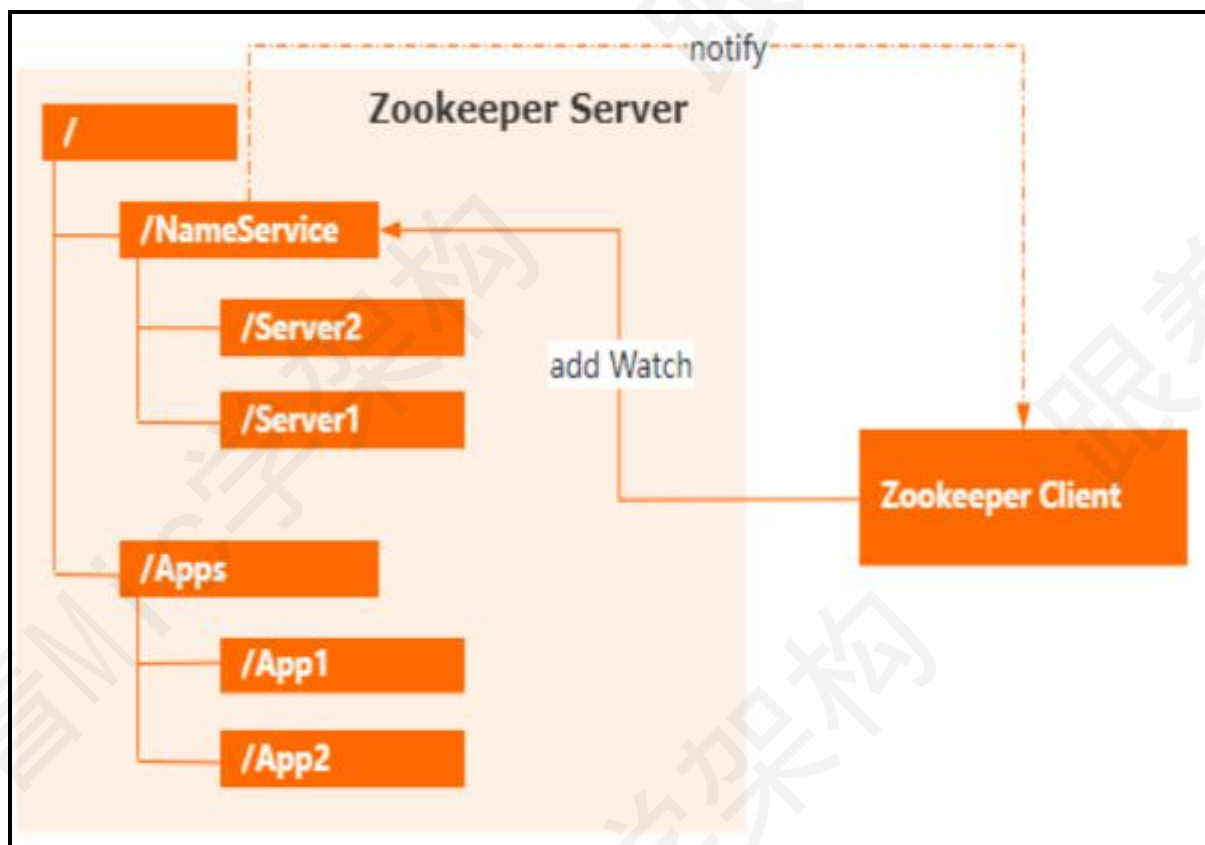
好的，这个问题我打算从两个方面来回答。

Zookeeper 是一个分布式协调组件，为分布式架构下的多个应用组件提供了顺序访问控制能力。

它的数据存储采用了类似于文件系统的树形结构，以节点的方式来管理存储在 Zookeeper 上的数据。



Zookeeper 提供了一个 Watch 机制，可以让客户端感知到 Zookeeper Server 上存储的数据变化，这样一种机制可以让 Zookeeper 实现很多的场景，比如配置中心、注册中心等。



Watch 机制采用了 Push 的方式来实现，也就是说客户端和 Zookeeper Server 会建立一个长连接，一旦监听的指定节点发生了变化，就会通过这个长连接把变化的事件推送给客户端。

Watch 的具体流程分为几个部分：

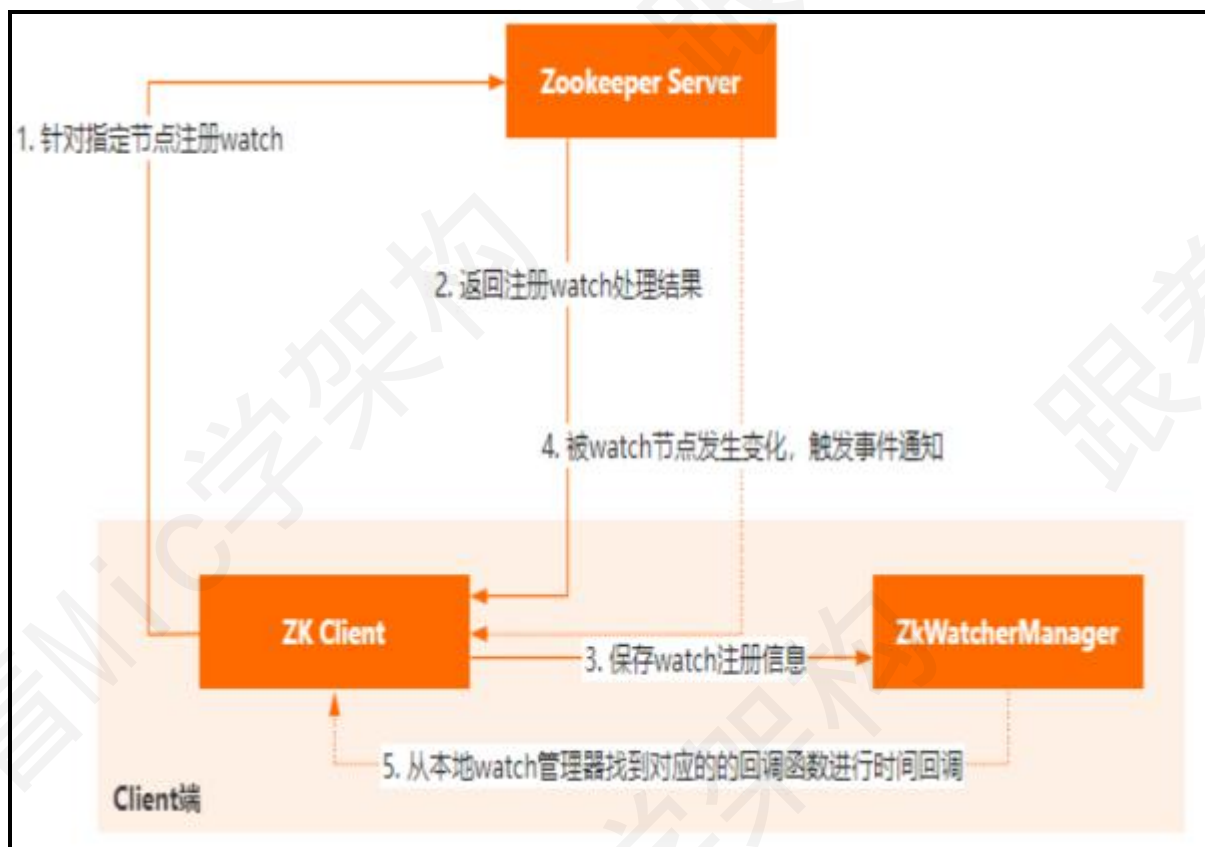
首先，是客户端通过指定命令比如 `exists`、`get`，对特定路径增加 watch

然后服务端收到请求以后，用 `HashMap` 保存这个客户端会话以及对应关注的节点路径，同时客户端也会使用 `HashMap`

存储指定节点和事件回调函数的对应关系。

当服务端指定被 watch 的节点发生变化后，就会找到这个节点对应的会话，把变化的事件和节点信息发给这个客户端。

客户端收到请求以后，从 `ZkWatcherManager` 里面对应的回调方法进行调用，完成事件变更的通知。



以上就是我对这个问题的理解！

## 面试点评

这个面试题呢，我认为考察的价值也很大，其实对于服务端的数据变更通知，无非就是 **pull** 和 **push** 两种方案，而这道题里面涉及到的技术点就是 **push** 的实现。

在业务开发里面，也可能会涉及到类似的场景，比如消息通知，扫码登录等。

如果你了解这些思想，那在解决这类问题的时候，会变得更加从容。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Spring 中有哪些方式可以把 Bean 注入到 IOC 容器？

今天收到一个工作 4 年的粉丝的面试题。

问题是：“Spring 中有哪些方式可以把 Bean 注入到 IOC 容器”。

他说这道题是所有面试题里面回答最好的，但是看面试官的表情，好像不太对。

我问他怎么回答的，他说：“接口注入”、“Setter 注入”、“构造器注入”。

为什么不对？来看看普通人和高手的回答。

## 普通人

## 高手

好的，把 Bean 注入到 IOC 容器里面的方式有 7 种方式

使用 xml 的方式来声明 Bean 的定义，Spring 容器在启动的时候会加载并解析这个 xml，把 bean 装载到 IOC 容器中。

使用 @ComponentScan 注解来扫描声明了 @Controller、@Service、@Repository、@Component 注解的类。

使用 @Configuration 注解声明配置类，并使用 @Bean 注解实现 Bean 的定义，这种方式其实是 xml 配置方式的一种演变，是 Spring 迈入到无配置化时代的里程碑。

使用 @Import 注解，导入配置类或者普通的 Bean

使用 FactoryBean 工厂 bean，动态构建一个 Bean 实例，Spring Cloud OpenFeign 里面的动态代理实例就是使用 FactoryBean 来实现的。

实现 ImportBeanDefinitionRegistrar 接口，可以动态注入 Bean 实例。这个在 Spring Boot 里面的启动注解有用到。

实现 ImportSelector 接口，动态批量注入配置类或者 Bean 对象，这个在 Spring Boot 里面的自动装配机制里面有用到。

以上就是我对这个问题的理解。

## 面试点评

工作了 4 年，IOC 和 DI 都没有搞清楚，作为面试官，想给你放水都放不了啊。

这道题目也很有意义，要想更加优雅的去解决一些实际业务问题，首先得有足够多的工具积累。

你可曾想过，**Bean** 的注入竟然有这么多方式，而且还有些方式是没听过的呢？

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Redis 存在线程安全问题吗？为什么？

---

一个工作了 5 年的粉丝私信我。

他说自己准备了半年时间，想如蚂蚁金服，结果第一面就挂了，非常难过。

问题是：“Redis 存在线程安全问题吗？”

关于这个问题，看看普通人和高手的回答。

### 普通人

### 高手

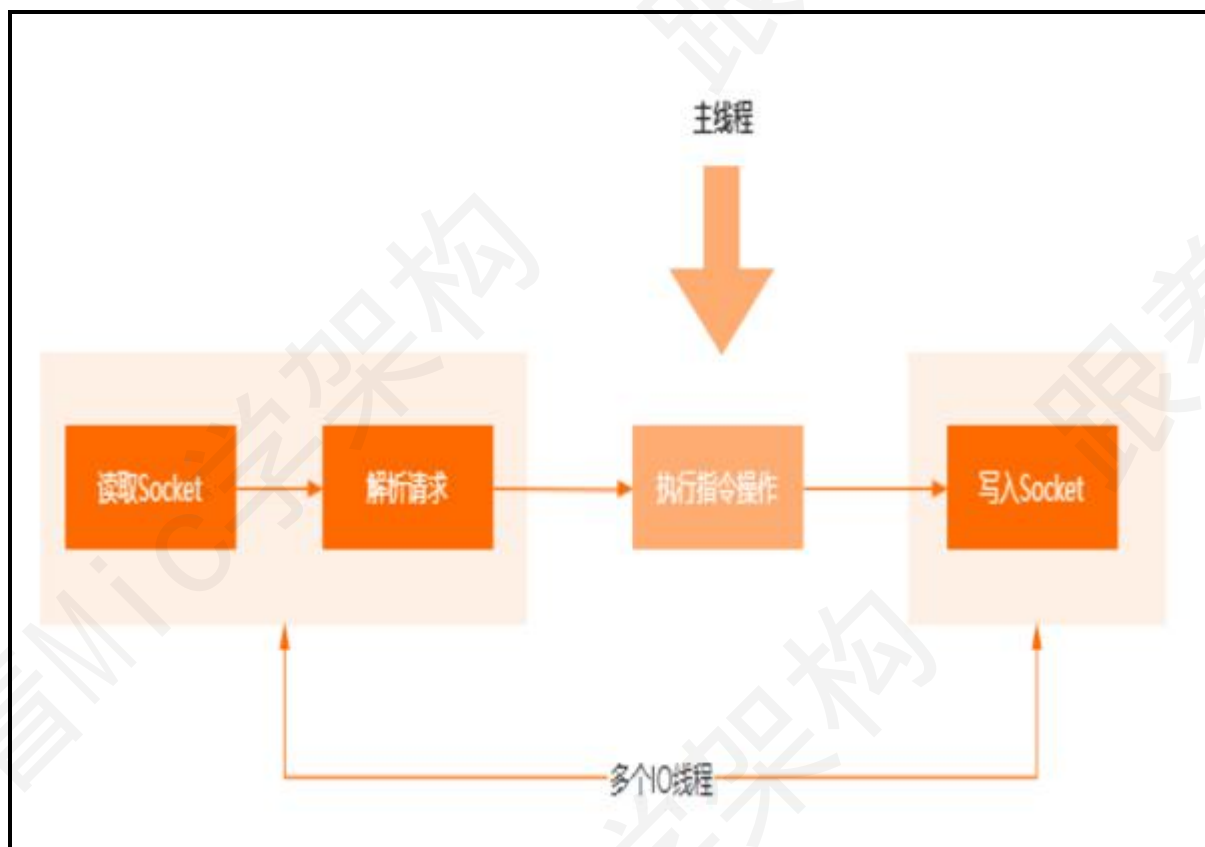
---

好的，关于这个问题，我从两个方面来回答。

第一个，从 Redis 服务端层面。

Redis Server 本身是一个线程安全的 K-V 数据库，也就是说在 Redis Server 上执行的指令，不需要任何同步机制，不会存在线程安全问题。

虽然 Redis 6.0 里面，增加了多线程的模型，但是增加的多线程只是用来处理网络 IO 事件，对于指令的执行过程，仍然是由主线程来处理，所以不会存在多个线程通知执行操作指令的情况。



为什么 Redis 没有采用多线程来执行指令，我认为有几个方面的原因。

Redis Server 本身可能出现的性能瓶颈点无非就是网络 IO、CPU、内存。但是 CPU 不是 Redis 的瓶颈点，所以没必要使用多线程来执行指令。

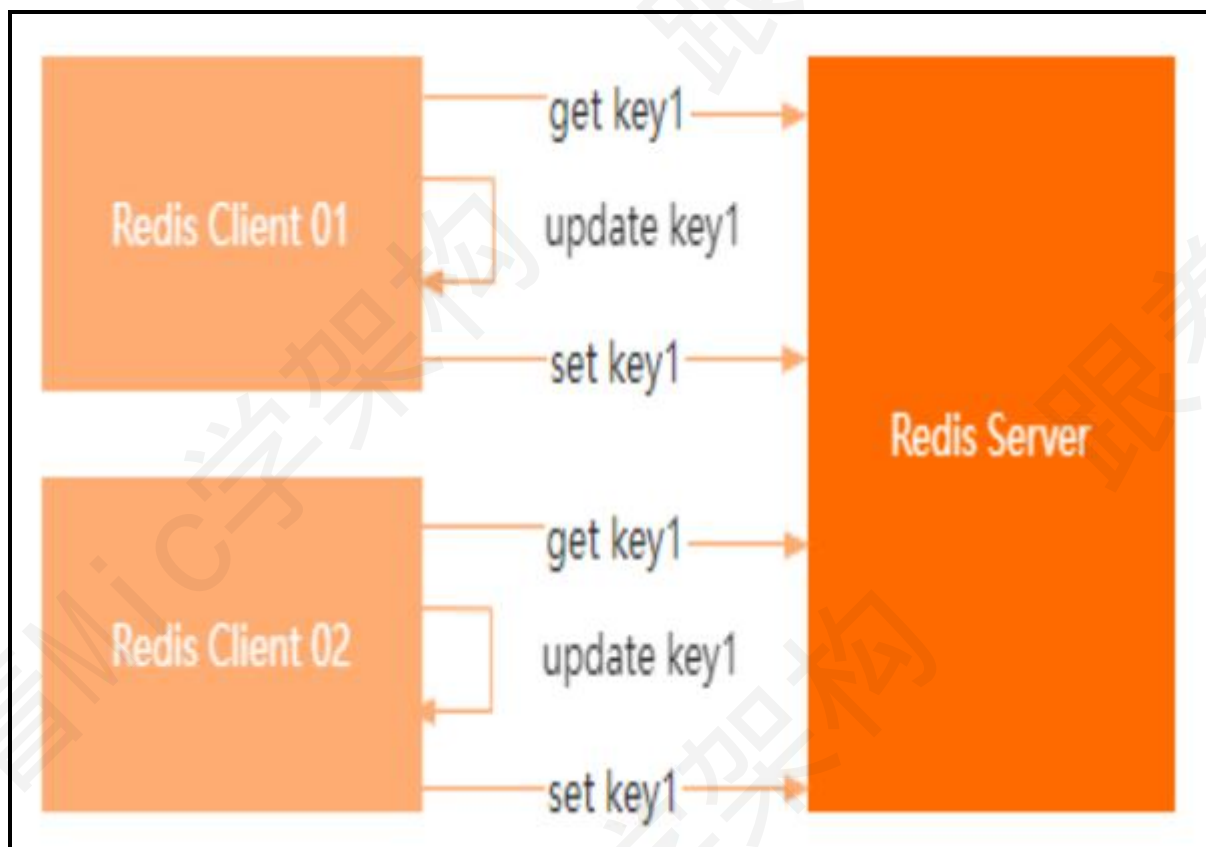
如果采用多线程，意味着对于 redis 的所有指令操作，都必须考虑到线程安全问题，也就是说需要加锁来解决，这种方式带来的性能影响反而更大。

第二个，从 Redis 客户端层面。

虽然 Redis Server 中的指令执行是原子的，但是如果有多个 Redis 客户端同时执行多个指令的时候，就无法保证原子性。

假设两个 redis client 同时获取 Redis Server 上的 key1，同时进行修改和写入，因为多线程环境下的原子性无法被保障，以及多进程情况下的共享资源访问的竞争问题，使得数据的安全性无法得到保障。





当然，对于客户端层面的线程安全性问题，解决方法有很多，比如尽可能的使用 Redis 里面的原子指令，或者对多个客户端的资源访问加锁，或者通过 Lua 脚本来实现多个指令的操作等等。

以上就是我对这个问题的理解。

## 面试点评

关于线程安全性问题，是一个非常重要，非常重要的知识。

虽然我们在实际开发中很少去主动使用线程，但是在项目中线程无处不在，比如 Tomcat 就是用多线程来处理请求的

如果对线程安全不了解，那么很容易出现各种生产事故和莫名其妙的问题。

这也是为什么大厂一定会问多线程并发的原因。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

# Spring 中 BeanFactory 和 FactoryBean 的区别

---

一个工作了六年多的粉丝，胸有成竹的去京东面试。

然后被 Spring 里面的一个问题卡住，唉，我和他说，6 年啦，Spring 都没搞明白？

那怎么去让面试官给你通过呢？

这个问题是：Spring 中 BeanFactory 和 FactoryBean 的区别。

好吧，对于这个问题看看普通人和高手的回答。

## 普通人

## 高手

---

关于这个问题，我从几个方面来回答。

首先，Spring 里面的核心功能是 IOC 容器，所谓 IOC 容器呢，本质上就是一个 Bean 的容器或者是一个 Bean 的工厂。

它能够根据 xml 里面声明的 Bean 配置进行 bean 的加载和初始化，然后 BeanFactory 来生产我们需要的各种各样的 Bean。

所以我对 BeanFactory 的理解了有两个。

BeanFactory 是所有 Spring Bean 容器的顶级接口，它为 Spring 的容器定义了一套规范，并提供像 `getBean` 这样的方法从容器中获取指定的 Bean 实例。

BeanFactory 在产生 Bean 的同时，还提供了解决 Bean 之间的依赖注入的能力，也就是所谓的 DI。

FactoryBean 是一个工厂 Bean，它是一个接口，主要的功能是动态生成某一个类型的 Bean 的实例，也就是说，我们可以自定义一个 Bean 并且加载到 IOC 容器里面。

它里面有一个重要的方法叫 `getObject()`，这个方法里面就是用来实现动态构建 Bean 的过程。

Spring Cloud 里面的 OpenFeign 组件，客户端的代理类，就是使用了 FactoryBean 来实现的。

以上就是我对这个问题的理解。

## 面试点评

---

这个问题，只要稍微看过 Spring 框架的源码，怎么都能回答出来。

关键在于你是否愿意逼自己去学习一些工作中不常使用的技术来提升自己。

在我看来，薪资和能力是一种等价交换，在市场经济下，能力一般又想获得更高薪资，很显然不可能！

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 简述一下你对线程池的理解？

---

到底是什么面试题，

让一个工作了 4 年的精神小伙，只是去参加了一场技术面试，

就被搞得精神萎靡。郁郁寡欢！

这一切的背后到底是道德的沦丧，还是人性的扭曲。

让我们一起揭秘一下这道面试题。

关于，“简述你对线程池的理解”，看看普通人和高手的回答。

### 普通人

### 高手

---

关于这个问题，我会从几个方面来回答。

首先，线程池本质上是一种池化技术，而池化技术是一种资源复用的思想，比较常见的有连接池、内存池、对象池。

而线程池里面复用的是线程资源，它的核心设计目标，我认为有两个：

减少线程的频繁创建和销毁带来的性能开销，因为线程创建会涉及到 CPU 上下文切换、内存分配等工作。

线程池本身会有参数来控制线程创建的数量，这样就可以避免无休止的创建线程带来的资源利用率过高的问题，

起到了资源保护的作用。

其次，我简单说一下线程池里面的线程复用技术。因为线程本身并不是一个受控的技术，也就是说线程的生命周期时由任务运行的状态决定的，无法人为控制。

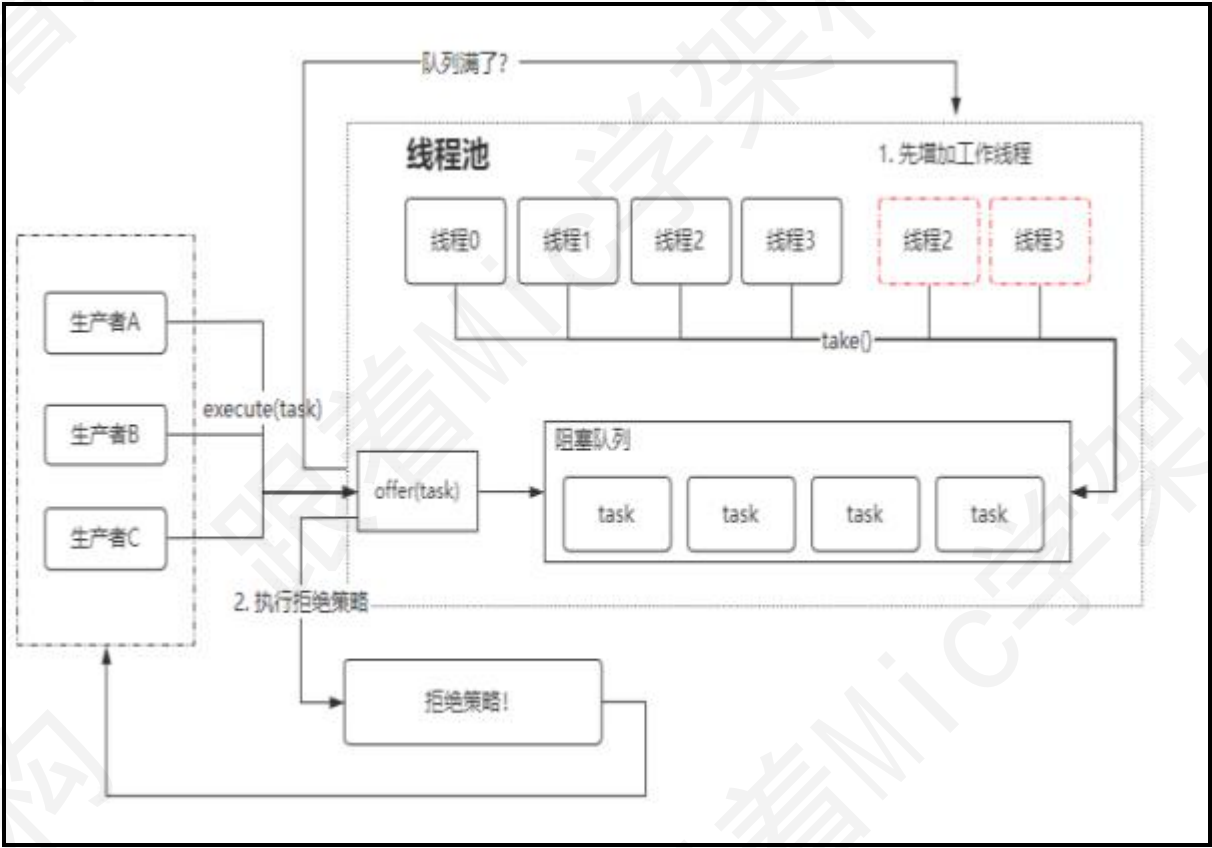
（图片）所以为了实现线程的复用，线程池里面用到了阻塞队列，简单来说就是线程池里面的工作线程处于一直运行状态，它会从阻塞队列中去获取待执行的任务，一旦队列空了，那这个工作线程就会被阻塞，直到下次有新的任务进来。

也就是说，工作线程是根据任务的情况实现阻塞和唤醒，从而达到线程复用的目的。

最后，线程池里面的资源限制，是通过几个关键参数来控制的，分别是核心线程数、最大线程数。

核心线程数表示默认长期存在的工作线程，而最大线程数是根据任务的情况动态创建的线程，主要是提高阻塞队列中任务的

处 理 效 率 。



以上就是我对这个问题的理解！

## 面试点评

我当时在阅读线程池的源码的时候，被里面的各种设计思想惊艳到了。

比如动态扩容和缩容的思想、线程的复用思想、以及线程回收的方法等等。

我发现越是简单的东西，反而越不简单。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了

更多的面试资料和面试技巧，可以私信我获取。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见！

## 如何理解 Spring Boot 中的 Starter?

一个工作了 3 年的 Java 程序员，遇到一个 Spring Boot 的问题。

他对这个问题有一些了解，但是回答得不是很好，希望参考我的高手回答。

这个问题是：“如何理解 Spring Boot 中的 Starter”。

对于这个问题，看看普通人和高手的回答。

### 普通人

### 高手

**Starter** 是 Spring Boot 的四大核心功能特性之一，除此之外，Spring Boot 还有自动装配、**Actuator** 监控等特性。

Spring Boot 里面的这些特性，都是为了让开发者在开发基于 Spring 生态下的企业级应用时，只需要关心业务逻辑，

减少对配置和外部环境的依赖。

其中，**Starter** 是启动依赖，它的主要作用有几个。

**Starter** 组件以功能为纬度，来维护对应的 jar 包的版本依赖，

使得开发者可以不需要去关心这些版本冲突这种容易出错的细节。

**Starter** 组件会把对应功能的所有 jar 包依赖全部导入进来，避免了开发者自己去引入依赖带来的麻烦。

**Starter** 内部集成了自动装配的机制，也就说在程序中依赖对应的 **starter** 组件以后，

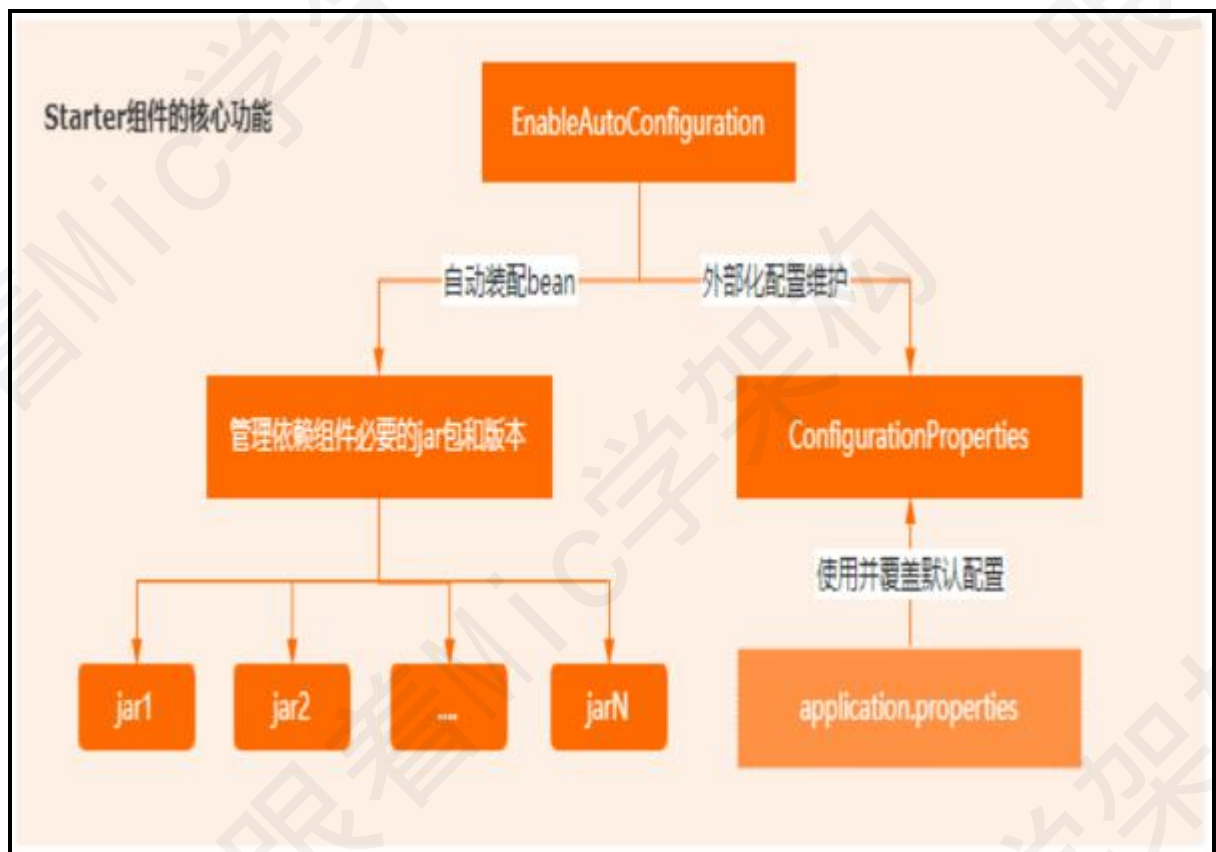
这个组件会自动集成到 Spring 生态下，并且对于相关 **Bean** 的管理，也是基于自动装配机制来完成。

依赖 **Starter** 组件后，这个组件对应的功能所需要维护的外部化配置，会自动集成到 **Spring Boot** 里面，

我们只需要在 `application.properties` 文件里面进行维护就行了，比如 **Redis** 这个 **starter**，只需要在 `application.properties`

文件里面添加 `redis` 的连接信息就可以直接使用了。

在我看来，**Starter** 组件几乎完美的体现了 **Spring Boot** 里面约定优于配置的理念。



另外，**Spring Boot** 官方提供了很多的 **Starter** 组件，比如 **Redis**、**JPA**、**MongoDB** 等等。

但是官方并不一定维护了所有中间件的 **Starter**，所以对于不存在的 **Starter**，第三方组件一般会自己去维护一个。

官方的 **starter** 和第三方的 **starter** 组件，最大的区别在于命名上。

官方维护的 **starter** 的以 `spring-boot-starter` 开头的前缀。

第三方维护的 **starter** 是以 `spring-boot-starter` 结尾的后缀

这也是一种约定优于配置的体现。

## 官方维护的starter

spring-boot-starter-xxx

## 第三方维护的starter

xxx-spring-boot-starter

以上就是我对这个问题的理解。

## 面试点评

---

在技术的学习过程中，我认为“为什么是”比“是什么”更重要。

以这种方式来学习，带来的好处就是对技术理解会更加深刻。

这道题考察的就是“为什么是”，不难，关键在于自己的理解。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，

如果你喜欢这个视频，记得点赞和收藏。

如果想获得一对一的面试指导以及面试资料，可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见！

## IO 和 NIO 有什么区别？

---

IO 问题一直是面试的重灾区之一

但又是非常重要而且面试必问的知识点



一个工作了 7 年的粉丝私信我，他去面试了 4 家互联网公司，有三个公司问他网络 IO 的问题，另外一个公司问了 Netty，结果都没回答上来。好吧，对于“IO 和 NIO 的区别”，看看普通人和高手的回答。

## 普通人

## 高手

好的，关于这个问题，我会从下面几个方面来回答。

首先，I/O，指的是 IO 流，它可以实现数据从磁盘中的读取以及写入。

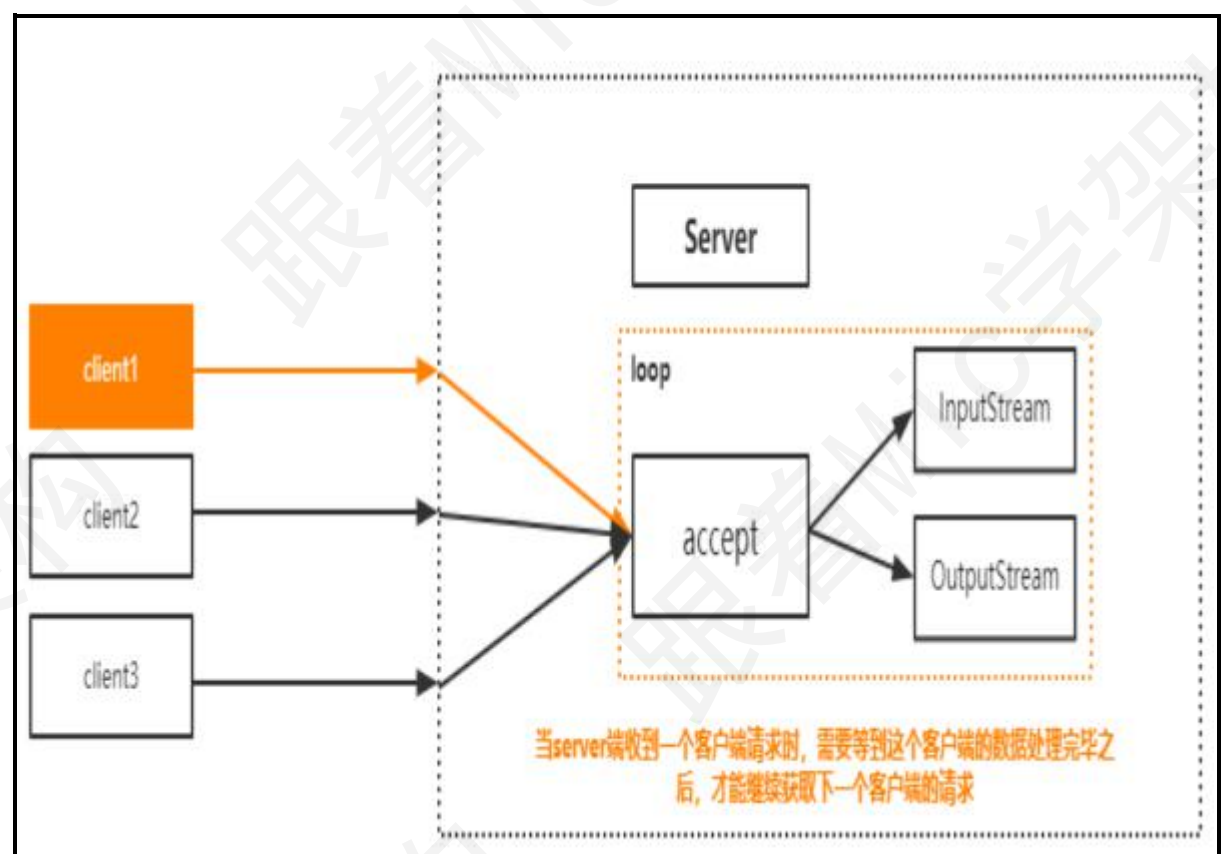
实际上，除了磁盘以外，内存、网络都可以作为 I/O 流的数据来源和目的地。

在 Java 里面，提供了字符流和字节流两种方式来实现数据流的操作。

其次，当程序是面向网络进行数据的 IO 操作的时候，Java 里面提供了 Socket 的方式来实现。

通过这种方式可以实现数据的网络传输。

基于 Socket 的 IO 通信，它是属于阻塞式 IO，也就是说，在连接以及 IO 事件未就绪的情况下，当前的连接会处于阻塞等待的状态。





如果一旦某个连接处于阻塞状态，那么后续的连接都得等待。所以服务端能够处理的连接数量非常有限。

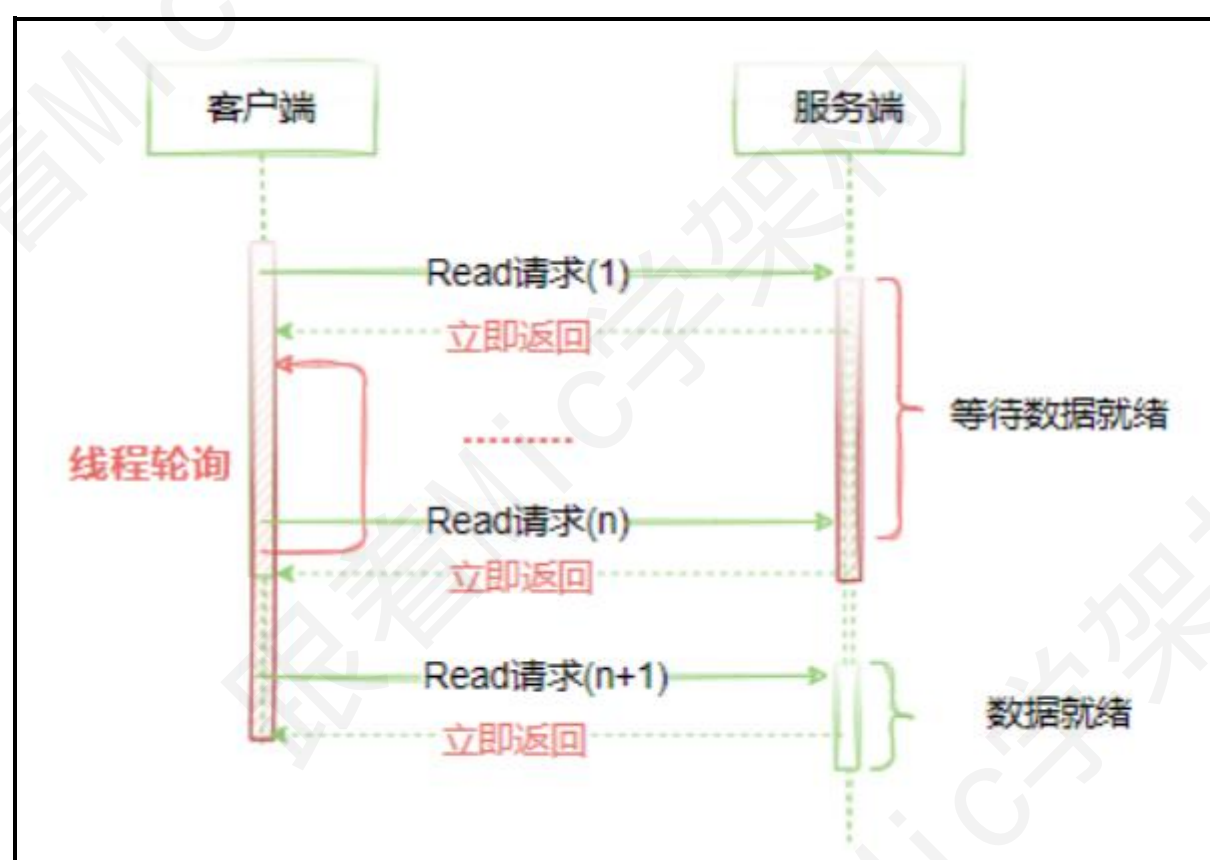
NIO，是 JDK1.4 里面新增的一种 NEW IO 机制，相比于传统的 IO，NIO 在效率上做了很大的优化，并且新增了几个核心组件。

Channel、Buffer、Selectors。

另外，还提供了非阻塞的特性，所以，对于网络 IO 来说，NIO 通常也称为 No-Block IO，非阻塞 IO。

也就是说，通过 NIO 进行网络数据传输的时候，如果连接未就绪或者 IO 事件未就绪的情况下，服务端不会阻塞当前连接，而是继续去轮询后续的连接来处理。

所以在 NIO 里面，服务端能够并行处理的链接数量更多。



因此，总的来说，IO 和 NIO 的区别，站在网络 IO 的视角来说，前者是阻塞 IO，后者是非阻塞 IO。

以上就是我对这个问题的理解。

## 面试点评

在互联网时代，网络 IO 是最基础的技术。

无论是微服务架构中的服务通信、还是应用系统和中间件之间的网络通信。

都在体现网络 IO 的重要性。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，

如果你喜欢这个视频，记得点赞和收藏。

如果想获得一对一的面试指导以及面试资料，可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见！

## 什么是幂等？如何解决幂等性问题？

---

一个在传统行业工作了 7 年的粉丝私信我。

他最近去很多互联网公司面试，遇到的很多技术和概念都没听过。

其中就有一道题是：“什么是幂等、如何解决幂等性问题”？

他说，这个概念听都没听过，怎么可能回答出来。

好的，对于这个问题，看看普通人和高手的回答。

### 普通人

### 高手

---

好的。

所谓幂等，其实它是一个数学上的概念，在计算机编程领域中，幂等是指一个方法被多次重复执行的时候产生的影响和第一次执行的影响相同。

之所以要考虑到幂等性问题，是因为在网络通信中，存在两种行为可能会导致接口被重复执行。

用户的重复提交或者用户的恶意攻击，导致这个请求会被多次重复执行。

在分布式架构中，为了避免网络通信导致的数据丢失，在服务之间进行通信的时候都会设计超时重试的机制，而这种机制有可能导致服务端接口被重复调用。

所以在程序设计中，对于数据变更类操作的接口，需要保证接口的幂等性。

而幂等性的核心思想，其实就是保证这个接口的执行结果只影响一次，后续即便再次调用，也不能对数据产生影响，所以基于这样一个诉求，常见的解决方法有很多。

使用数据库的唯一约束实现幂等，比如对于数据插入类的场景，比如创建订单，因为订单号肯定是唯一的，所以如果是多次调用就会触发数据库的唯一约束异常，从而避免一个请求创建多个订单的问题。

使用 redis 里面提供的 setNX 指令，比如对于 MQ 消费的场景，为了避免 MQ 重复消费导致数据多次被修改的问题，可以在接受到 MQ 的消息时，把这个消息通过 setNx 写入到 redis 里面，一旦这个消息被消费过，就不会再次消费。

使用状态机来实现幂等，所谓的状态机是指一条数据的完整运行状态的转换流程，比如订单状态，因为它的状态只会向前变更，所以多次修改同一条数据的时候，一旦状态发生变更，那么对这条数据修改造成的影响只会发生一次。

当然，除了这些方法以外，还可以基于 token 机制、去重表等方法来实现，但是不管是什么方法，无非就是两种，

要么就是接口只允许调用一次，比如唯一约束、基于 redis 的锁机制。

要么就是对数据的影响只会触发一次，比如幂等性、乐观锁

以上就是我对这个问题的理解。

## 面试点评

---

技术这个行业的发展是很快的，如果自己的技术能力和认知跟不上变化。

那基本上可以说是被时代淘汰了，所以保持持续学习是非常重要的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了

喜欢我的作品的小伙伴记得点赞和收藏。

如果你在面试的时候遇到一些不懂的问题，可以随时来私信我

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 如何中断一个正在运行的线程？

---

一个去京东面试的工作了 5 年的粉丝来找我说：

Mic 老师，你说并发编程很重要，果然我今天又挂在一道并发编程的面试题上了。

我问他问题是什么，他说：“如何中断一个正在运行中的线程？”。

我说这个问题很多工作 2 年的人都知道~

好吧，对于这个问题，来看看普通人和高手的回答。

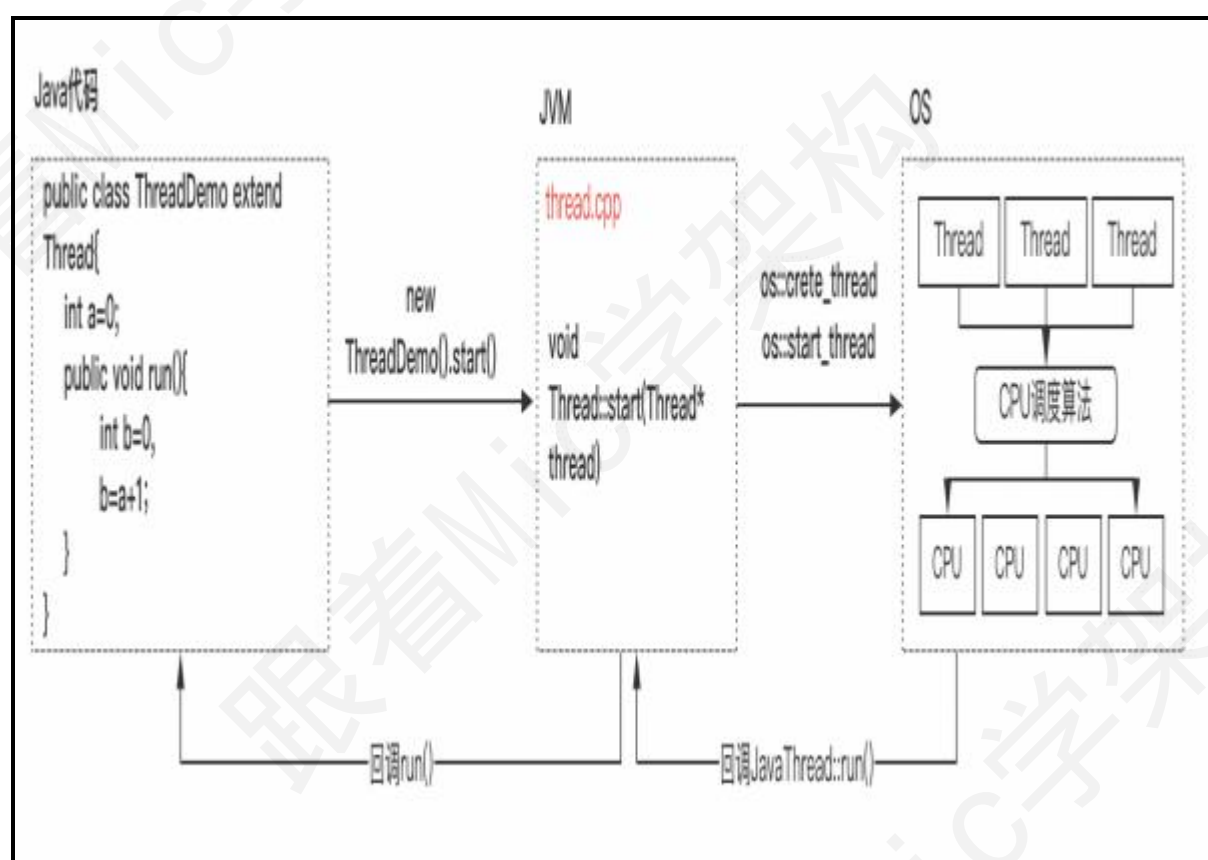
## 普通人

## 高手

关于这个问题，我从几个方面来回答。

首先，线程是系统级别的概念，在 **Java** 里面实现的线程，最终的执行和调度都是由操作系统来决定的，**JVM** 只是对操作系统层面的线程做了一层包装而已。

所以我们在 **Java** 里面通过 **start** 方法启动一个线程的时候，只是告诉操作系统这个线程可以被执行，但是最终交给 **CPU** 来执行是操作系统的调度算法来决定的。



因此，理论上来说，要在 **Java** 层面去中断一个正在运行的线程，只能像类似于 **Linux** 里面的 `kill` 命令结束进程的方式一样，强制终止。

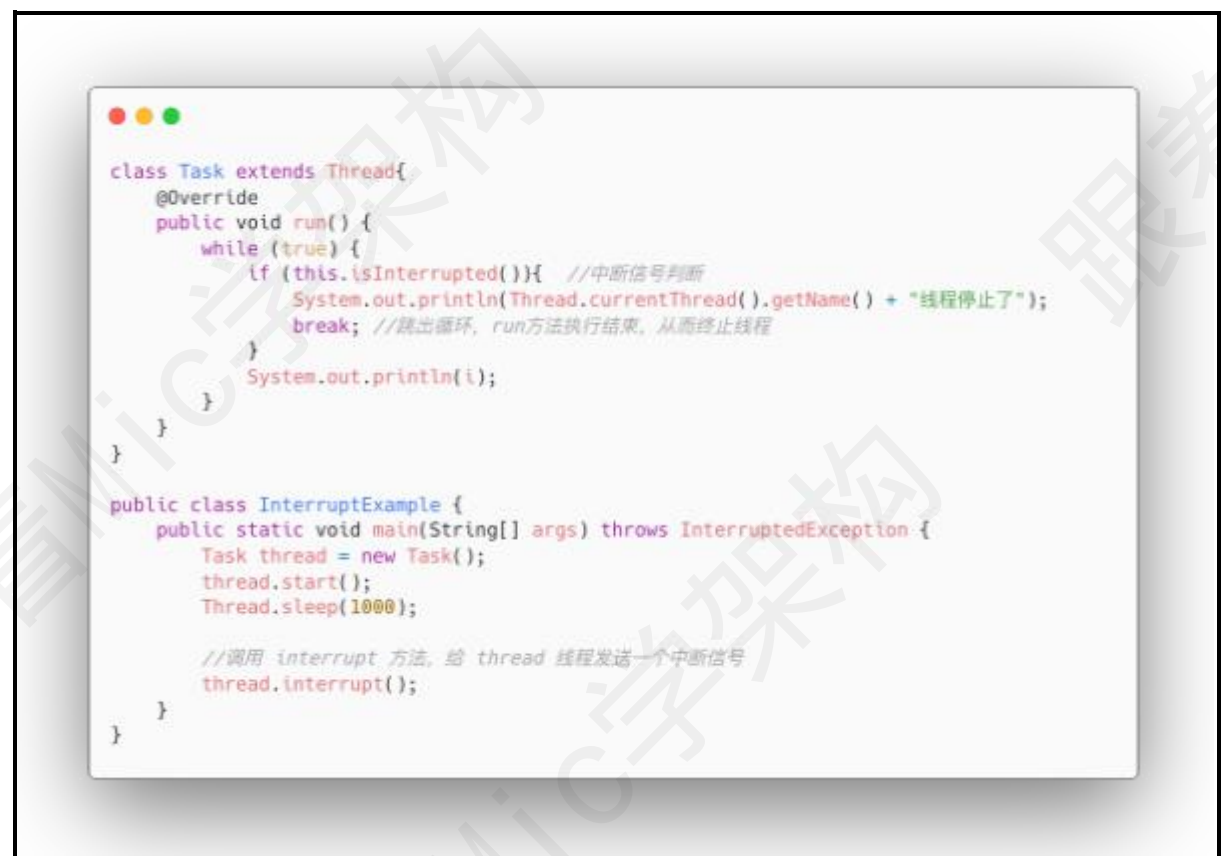
所以，**Java Thread** 里面提供了一个 `stop` 方法可以强行终止，但是这种方式是不安全的，因为有可能线程的任务还没有，导致出现运行结果不正确的问题。

要想安全的中断一个正在运行的线程，只能在线程内部埋下一个钩子，外部程序通过这个钩子来触发线程的中断命令。

因此，在 **Java Thread** 里面提供了一个 `interrupt()` 方法，这个方法配合 `isInterrupted()` 方法使用，就可以实现安全的中断机制。

这种实现方法并不是强制中断，而是告诉正在运行的线程，你可以停止了，不过是否要中断，取决于正在运行的线程，所以它能够保证线程运行结果的安全性。

以上就是我对这个问题的理解！



## 面试点评：

这个问题，很多工作了 5 年以上的小伙伴都不一定清楚。

我想说的是，一味的专注在 CRUD 这种自动化的重复性工作中除了前面 3 年时间会有很多的成长以外，后续的时间基本上就是在做重复的劳动。

和别人拉开差距恰恰是工作之外的 8 个小时。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了

如果觉得作品不错，记得点赞和关注。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## JVM 如何判断一个对象可以被回收

Hi，我是 Mic。

今天分享一道一线互联网公司必问的面试题。

”JVM 如何判断一个对象可以被回收“

关于这个问题，来看看普通人和高手的回答。

## 高手

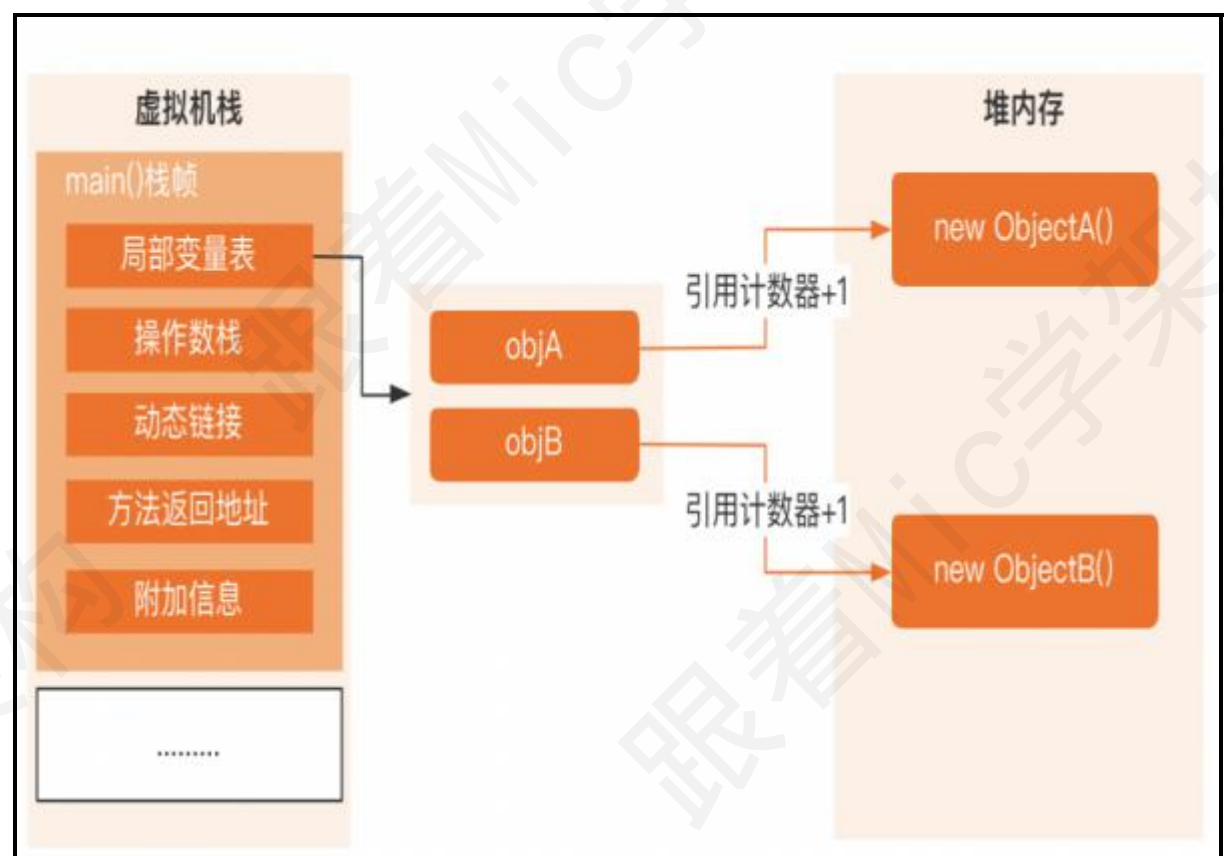
好的，面试官。

在 **JVM** 里面，要判断一个对象是否可以被回收，最重要的是判断这个对象是否还在被使用，只有没被使用的对象才能回收。

引用计数器，也就是为每一个对象添加一个引用计数器，用来统计指向当前对象的引用次数，

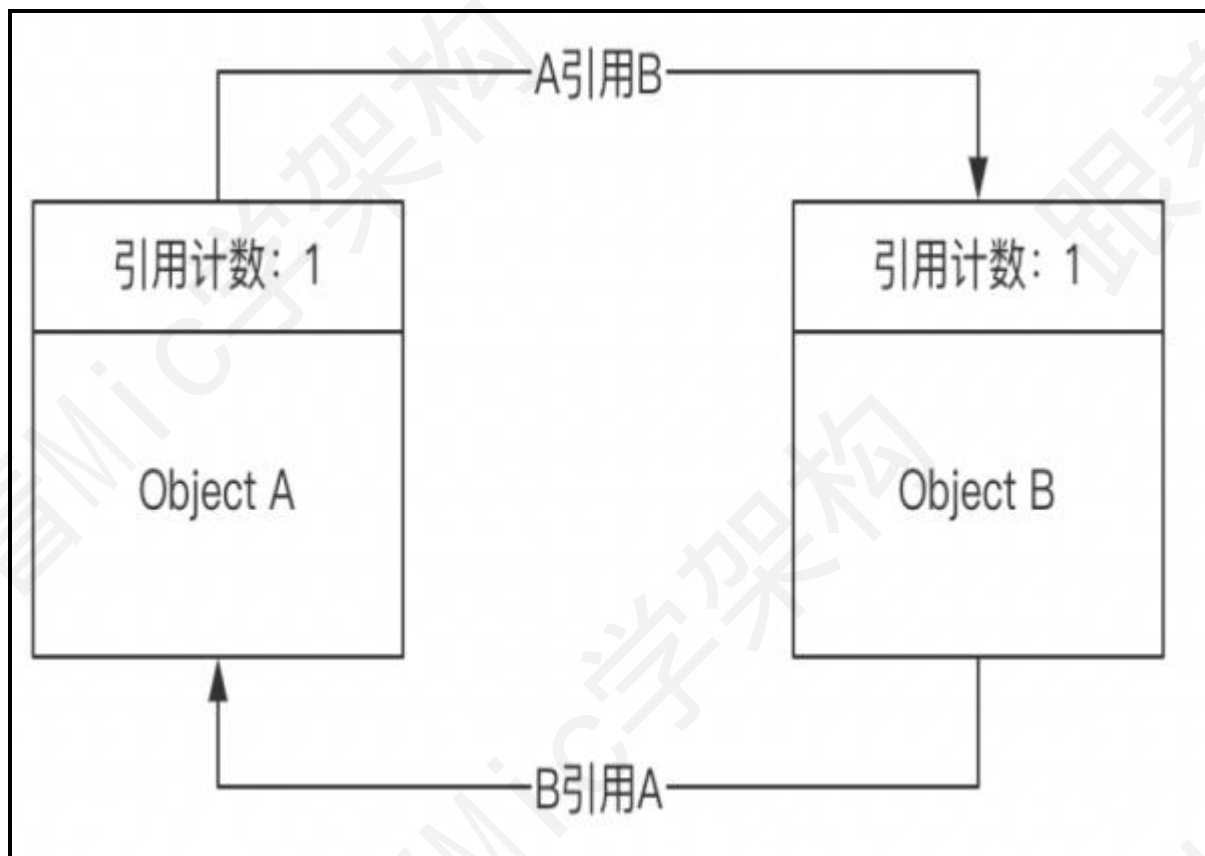
如果当前对象存在应用的更新，那么就对这个引用计数器进行增加，一旦这个引用计数器变成 0，就意味着它可以被回收了。

这种方法需要额外的空间来存储引用计数器，但是它的实现很简单，而且效率也比较高。



不过主流的 JVM 都没有采用这种方式，因为引用计数器在处理一些复杂的循环引用或者相互依赖的情况时，

可能会出现一些不再使用但又无法回收的内存，造成内存泄露的问题。



可达性分析，它的主要思想是，首先确定一系列肯定不能回收的对象作为 GC root，

比如虚拟机栈里面的引用对象、本地方法栈引用的对象等，然后以 GC ROOT 作为起始节点，

从这些节点开始向下搜索，去寻找它的直接和间接引用的对象，当遍历完之后如果发现有一些对象不可到达，

那么就认为这些对象已经没有用了，需要被回收。

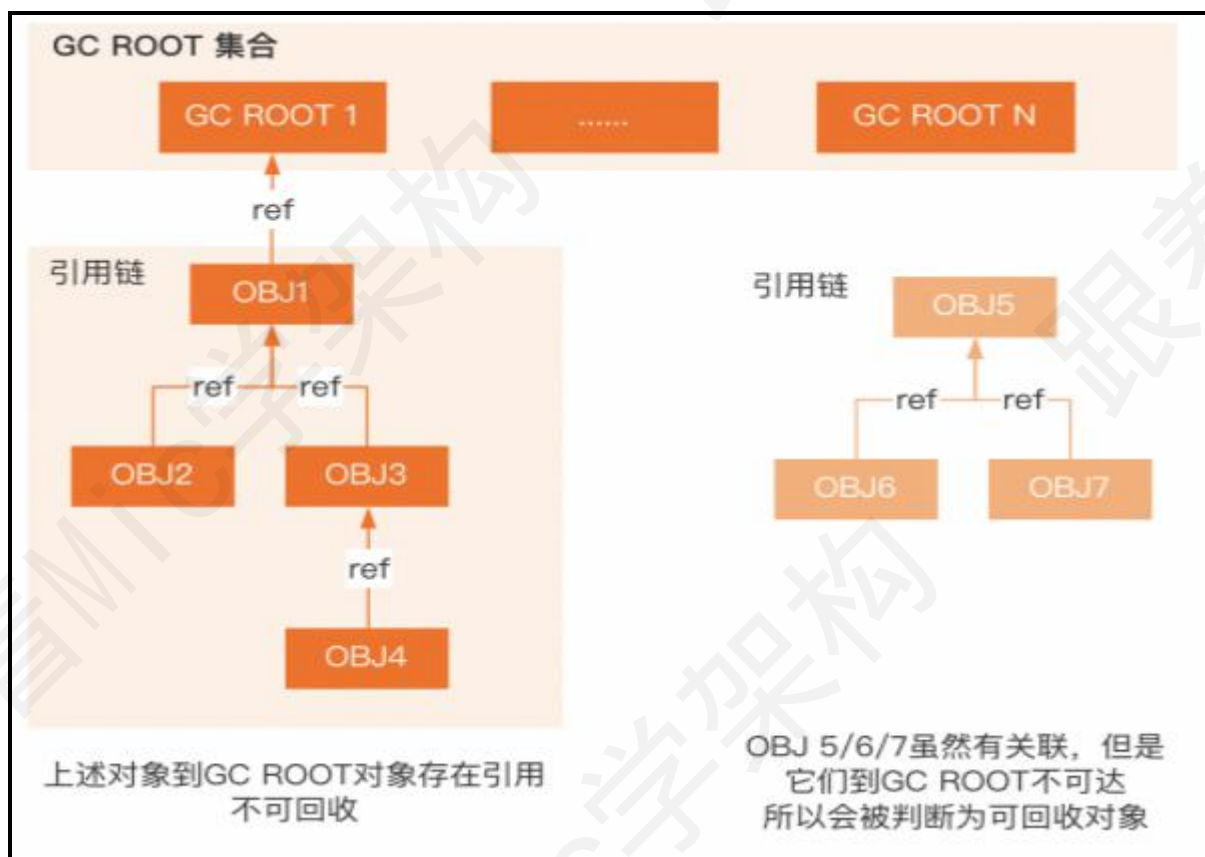
在垃圾回收的时候，JVM 会首先找到所有的 GC root，这个过程会暂停所有用户线程，

也就是 stop the world，然后再从 GC Roots 这些根节点向下搜索，可达的对象保留，不可达的就会回收掉。

可达性分析是目前主流 JVM 使用的算法。



以上就是我对这个问题的理解。



## 面试点评

很多粉丝和我说，很多东西看完以后过一段时间就忘记了，问我是怎么记下来的。

我和他说，技术这些东西不需要记，你唯一能做的就是减少碎片化的学习，多花一点时间在系统学习上，只有体系化的知识是不会忘记的。

可是，搭建体系化知识的过程要比碎片化的点状学习痛苦不止一万倍。

技术的沉淀是没有捷径的，只能花苦功夫去学习。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了

喜欢我的作品的小伙伴记得点赞和收藏。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 说说你对 Spring MVC 的理解

一个工作了 7 年的粉丝，他说在面试之前，Spring 这块的内容准备得很充分。



而且各种面试题也刷了，结果在面试的时候，面试官问：“说说你对 Spring MVC 的理解”。

这个问题一下给他整不会了，就是那种突然不知道怎么组织语言，最后因为回答比较混乱没通过面试。

ok，对于这个问题，我们来看看普通人和高手的回答。

## 普通人

## 高手

好的，关于这个问题，我会从几个方面来回答。

首先，Spring MVC 是属于 Spring Framework 生态里面的一个模块，它是在 Servlet 基础上构建并且使用 MVC 模式设计的一个 Web 框架，

主要的目的是简化传统 Servlet+JSP 模式下的 Web 开发方式。

其次，Spring MVC 的整体架构设计对 Java Web 里面的 MVC 架构模式做了增强和扩展，主要有几个方面。

把传统 MVC 框架里面的 Controller 控制器做了拆分，分成了前端控制器 DispatcherServlet 和后端控制器 Controller。

把 Model 模型拆分成业务层 Service 和数据访问层 Repository。

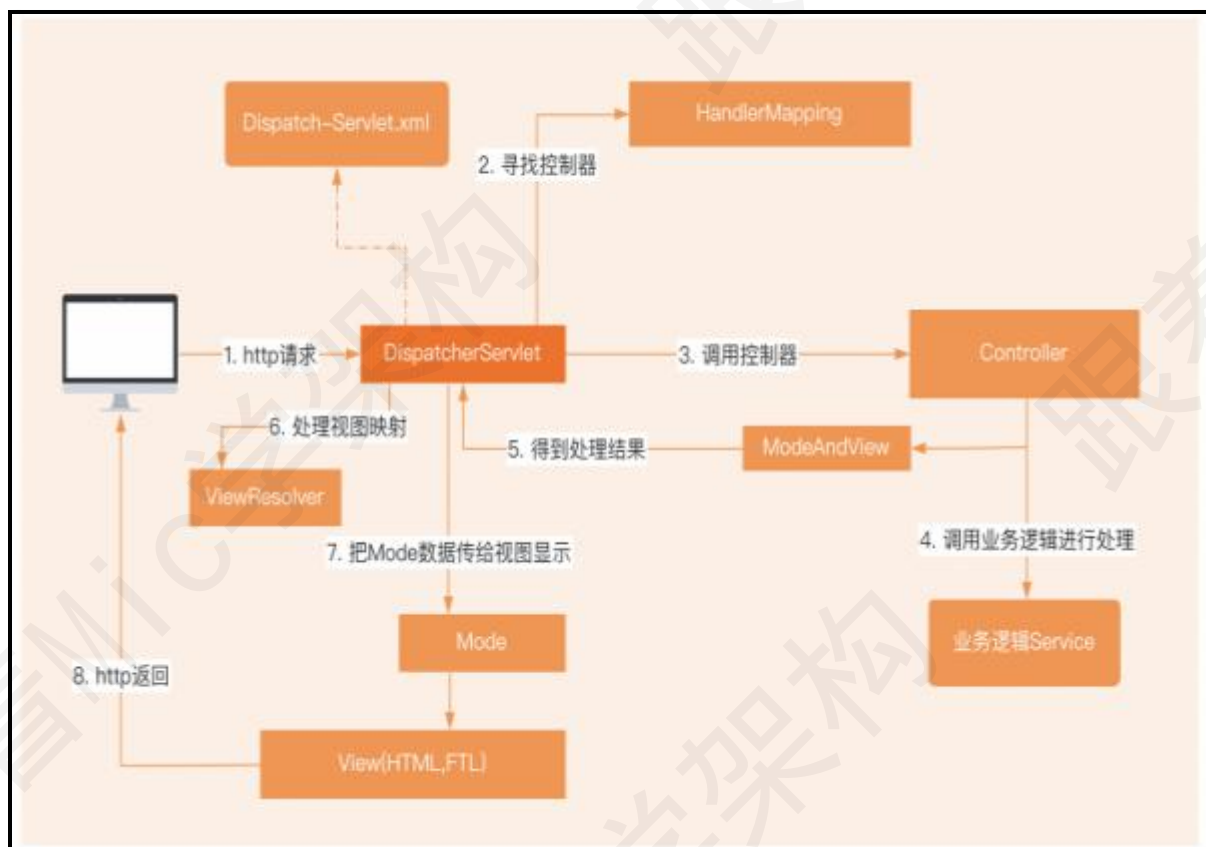
在视图层，可以支持不同的视图，比如 Freemark、velocity、JSP 等等。

所以，Spring MVC 天生就是为了 MVC 模式而设计的，因此在开发 MVC 应用的时候会更加方便和灵活。

Spring MVC 的具体工作流程是，浏览器的请求首先会经过 SpringMVC 里面的核心控制器 DispatcherServlet，它负责对请求进行分发到对应的 Controller。

Controller 里面处理完业务逻辑之后，返回 ModelAndView。

然后 DispatcherServlet 寻找一个或者多个 ViewResolver 视图解析器，找到 ModelAndView 指定的视图，并把数据显示到客户端。



以上就是我对 Spring MVC 的理解。

## 面试点评

我培训过 3W 多名 Java 架构师,我发现他们对技术的理解只是停留在使用层面,并没有深层次的思考这些技术框架的底层设计,导致他们在到了工作 5 年以后。想转架构的时候,缺少顶层设计能力和抽象思维。

好的,本期的普通人 VS 高手面试系列的视频就到这里结束了

喜欢我的作品的小伙伴记得点赞和收藏。

我是 Mic,一个工作了 14 年的 Java 程序员,咱们下期再见。

## 请说一下 Mysql 索引的优点和缺点?

Hi, 我是 Mic

今天分享的这道面试题,让一个工作 4 年的小伙子去大众点评拿了 60W 年薪。

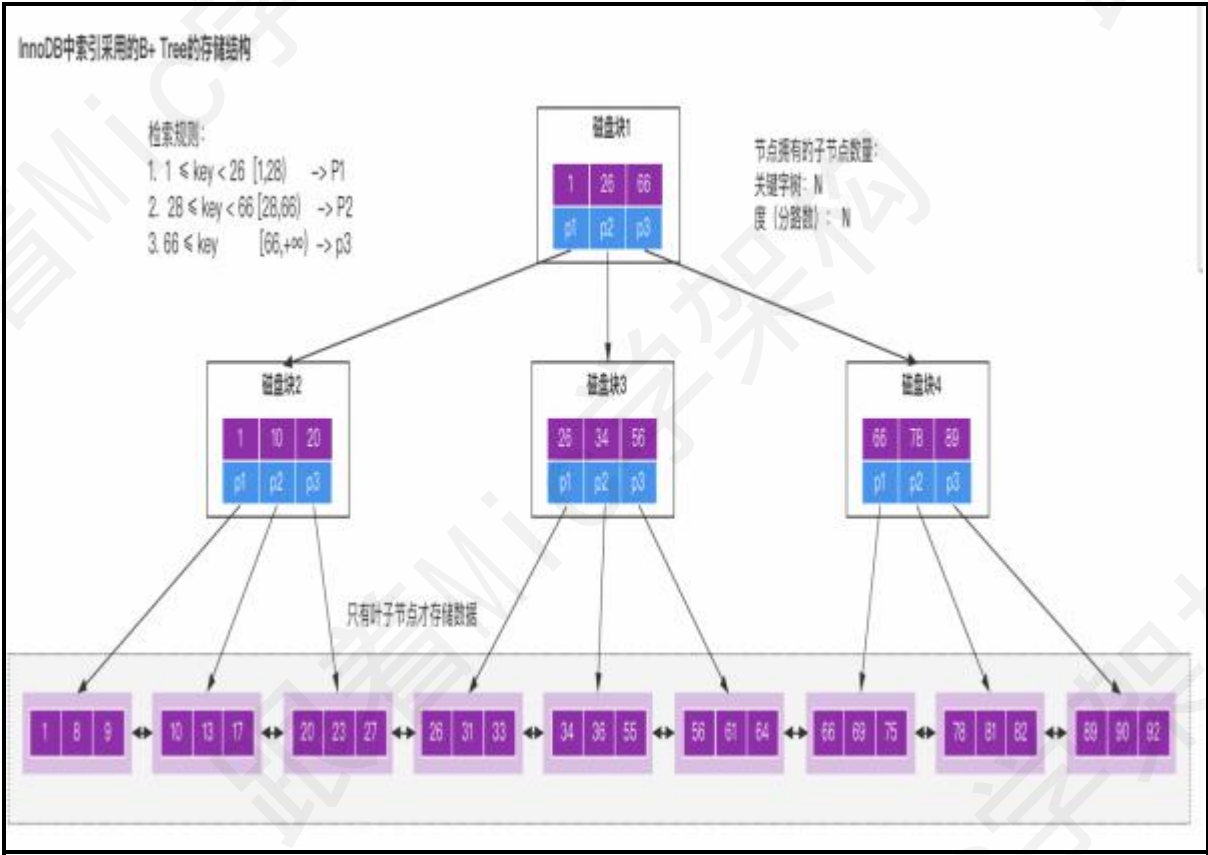
这道面试题是:“请你说一下 Mysql 索引的优点和缺点“

关于这道题，看看普通人和高手的回答

## 普通人

## 高手

（图片）索引，是一种能够帮助 **Mysql** 高效从磁盘上检索数据的一种数据结构。  
在 **Mysql** 中的 **InnoDB** 引擎中，采用了 **B+** 树的结构来实现索引和数据的存储



在我看来，**Mysql** 里面的索引的优点有很多

通过 **B+** 树的结构来存储数据，可以大大减少数据检索时的磁盘 **IO** 次数，从而提升数据查询的性能

**B+** 树索引在进行范围查找的时候，只需要找到起始节点，然后基于叶子节点的链表结构往下读取即可，查询效率较高。

通过唯一索引约束，可以保证数据表中每一行数据的唯一性

当然，索引的不合理使用，也会有带来很多的缺点。

数据的增加、修改、删除，需要涉及到索引的维护，当数据量较大的情况下，索引的维护会带来较大的性能开销。

一个表中允许存在一个聚簇索引和多个非聚簇索引，但是索引数不能创建太多，否则造成的索引维护成本过高。

创建索引的时候，需要考虑到索引字段值的分散性，如果字段的重复数据过多，创建索引反而会带来性能降低。

在我看来，任何技术方案都会有两面性，大部分情况下，技术方案的选择更多的是看中它的优势和当前问题的匹配度。

以上就是我对这个问题的理解。

## 面试点评

---

行业竞争加剧，再加上现在大环境不好，各个一二线大厂都在裁员。

带来的问题就是，人才筛选难度增加，找工作越来越难。

这道题目考察的是求职者对于 **Mysql** 的理解程度，不算难，但能卡主很多人。

好的，本期的普通人 **VS** 高手面试系列的视频就到这里结束了，

喜欢的朋友记得点赞和收藏。

有任何工作和学习上的问题，可以随时私信我。

我是 Mic，一个工作了 14 年的 **Java** 程序员，咱们下期再见。

## new String("abc")到底创建了几个对象？

---

一个工作了 6 年的粉丝和我说，

最近面试感觉越来越难的，基本上都会问技术底层原理，甚至有些还会问到操作系统层面的知识。

我说，现在各个一线大厂有很多优秀的程序员毕业了，再加上市场大环境不好对程序员的需求量也在减少。

如果技术底子不好，确实找工作会很困难。

今天分享的问题是：“new String(“abc”)到底创建了几个对象？

关于这个问题，看看普通人和高手的回答。

## 普通人

## 高手

---

好的，面试官。

首先，这个代码里面有一个 `new` 关键字，这个关键字是在程序运行时，根据已经加载的系统类 `String`，在堆内存里面实例化的一个字符串对象。

然后，在这个 `String` 的构造方法里面，传递了一个“abc”字符串，因为 `String` 里面的字符串成员变量是 `final` 修饰的，所以它是一个字符串常量。

接下来，JVM 会拿字面量“abc”去字符串常量池里面试图去获取它对应的 `String` 对象引用，如果拿不到，就会在堆内存里面创建一个“abc”的 `String` 对象

并且把引用保存到字符串常量池里面。

后续如果再有字面量“abc”的定义，因为字符串常量池里面已经存在了字面量“abc”的引用，所以只需要从常量池获取对应的引用就可以了，不需要再创建。

所以，对于这个问题，我认为的答案是

如果 `abc` 这个字符串常量不存在，则创建两个对象，分别是 `abc` 这个字符串常量，

以及 `new String` 这个实例对象。

如果 `abc` 这字符串常量存在，则只会创建一个对象

## 面试点评

---

从高手的回答中可以看到，必须要对 JVM 里面的运行时内存划分以及对 JVM 常量池的理解足够深刻。

现在技术的面试也偏向于体系化的考察，不再是点状式的提问了。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，

喜欢的朋友记得点赞和收藏。

有任何工作和学习上的问题，可以随时私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

# 常见的限流算法有哪些？

---

一个在传统行业工作了 5 年的程序员。

去一个互联网公司面试，遇到了一个限流的问题。

因为之前完全没接触过分布式这块的项目，所以根本就回答不上来，向我来求助。

其中有一个问题是：“请你说一下常见的限流算法”。

对于这个问题，看看普通人和高手的回答。

## 普通人

## 高手

---

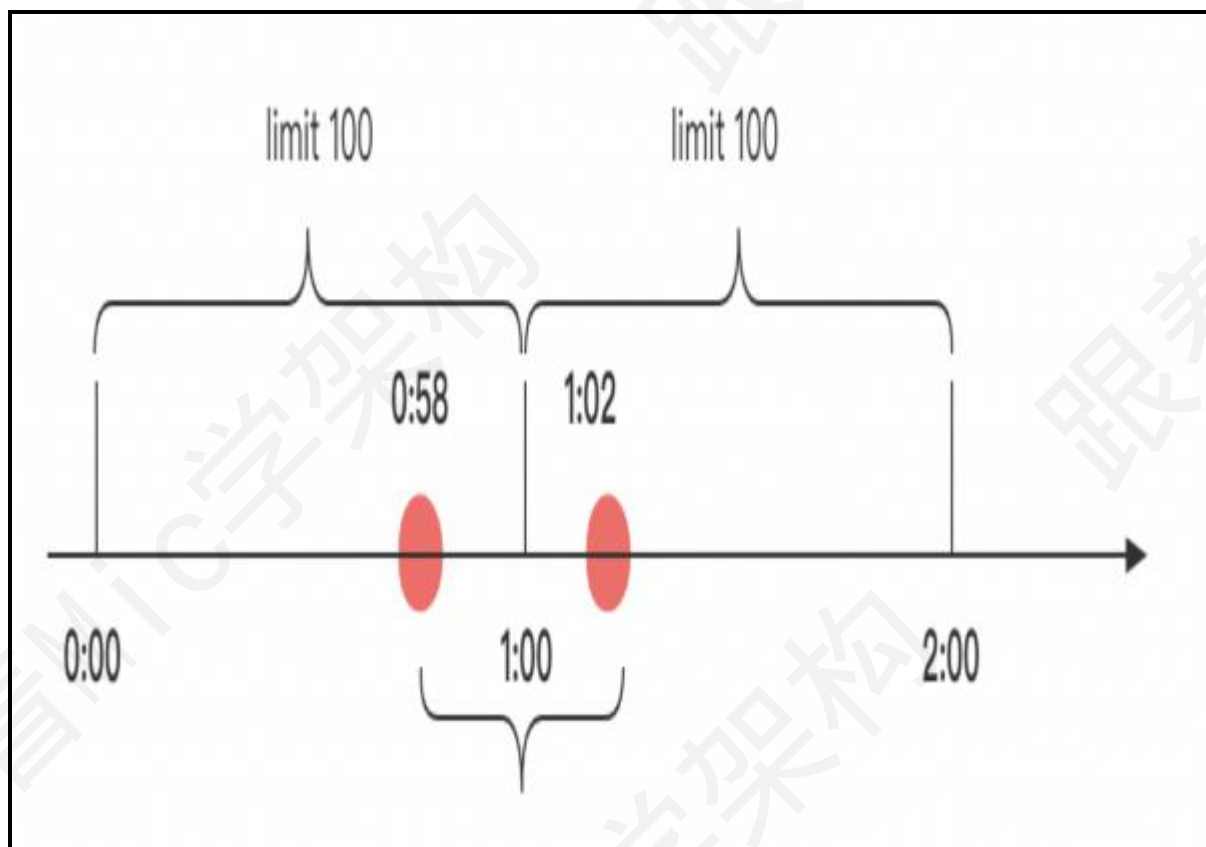
好的，关于这个问题，我会从几个方面来回答。

首先，限流算法是一种系统保护策略，主要是避免在流量高峰导致系统被压垮，造成系统不可用的问题。

常见的限流算法有 5 种。

计数器限流，一般用在单一维度的访问频率限制上，比如短信验证码每隔 60s 只能发送一次，或者接口调用次数等

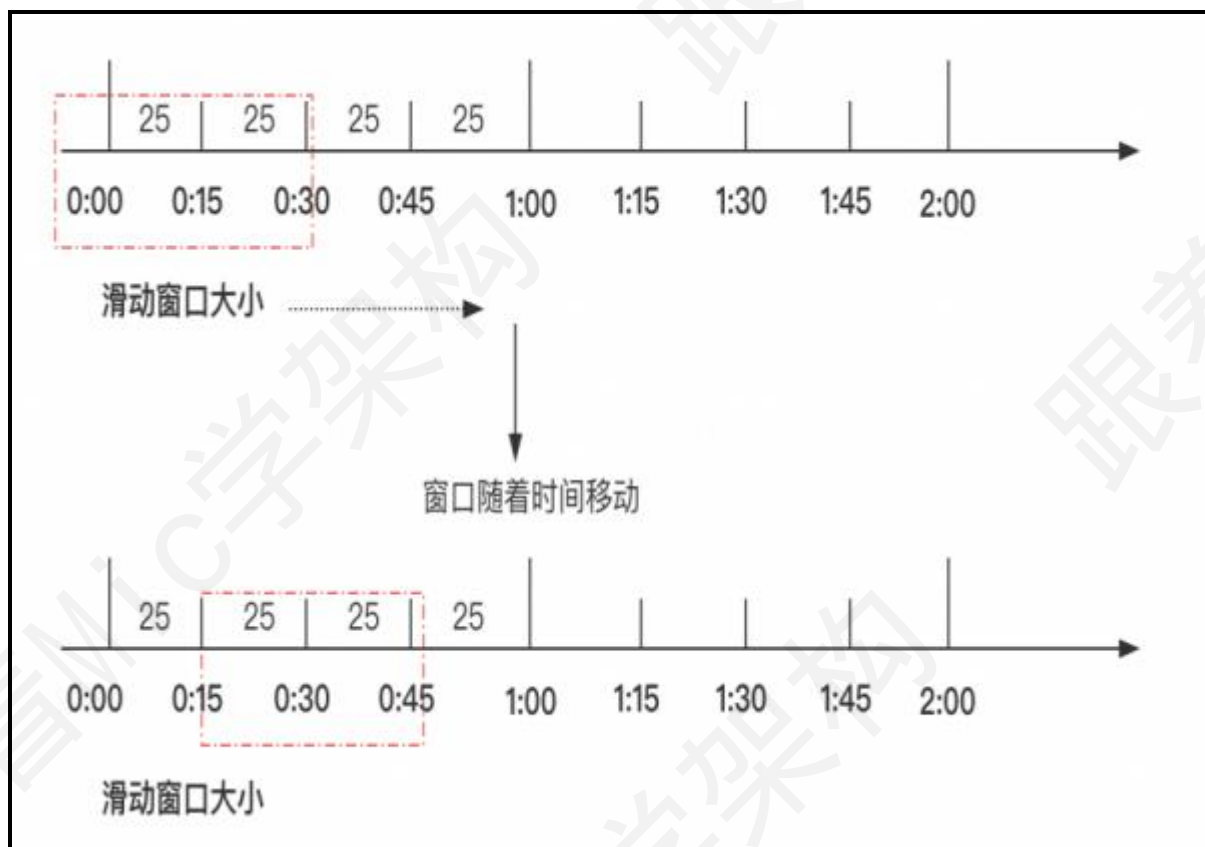
它的实现方法很简单，每调用一次就加 1，处理结束以后减一。



滑动窗口限流，本质上也是一种计数器，只是通过以时间为维度的可滑动窗口设计，来减少了临界值带来的并发超过阈值的问题。

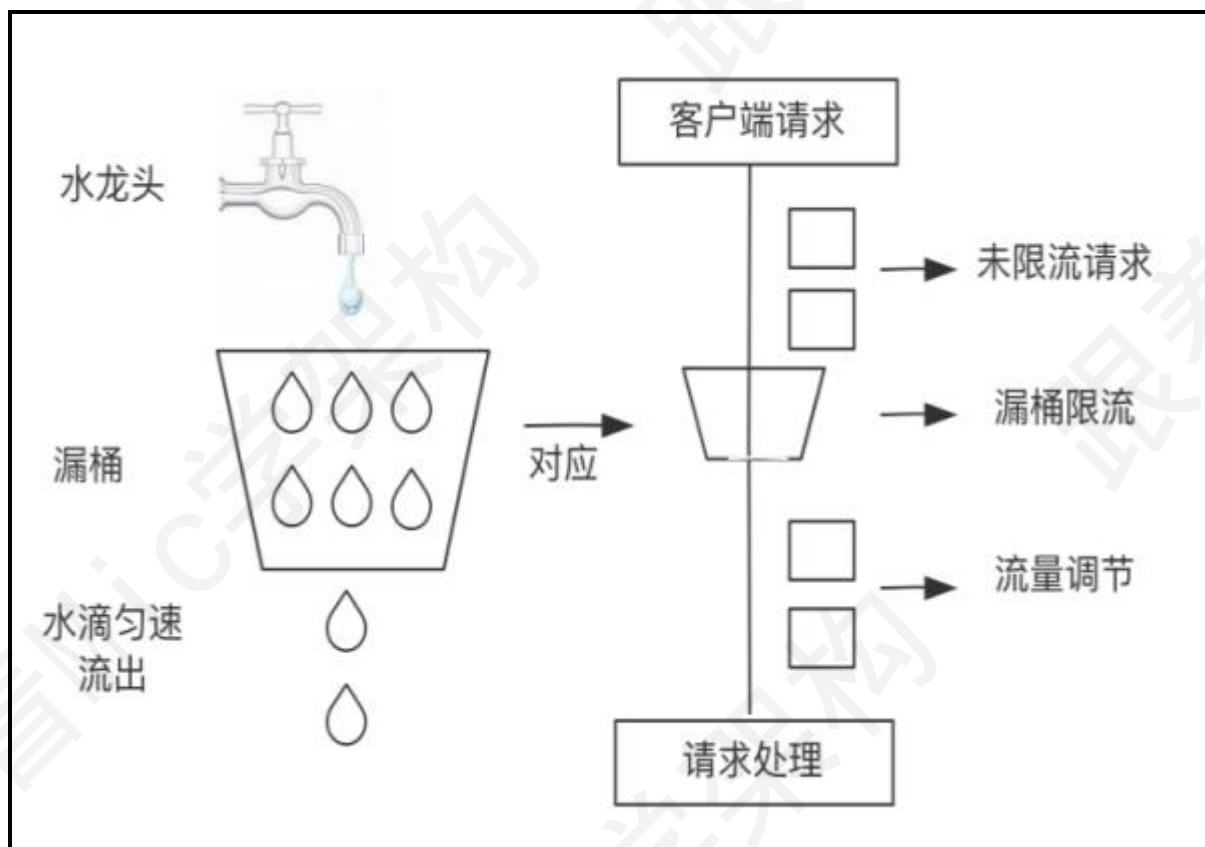
每次进行数据统计的时候，只需要统计这个窗口内每个时间刻度的访问量就可以了。

Spring Cloud 里面的熔断框架 Hystrix，以及 Spring Cloud Alibaba 里面的 Sentinel 都采用了滑动窗口来做数据统计。



(如图)漏桶算法，它是一种恒定速率的限流算法，不管请求量是多少，服务端的处理效率是恒定的。基于 MQ 来实现的生产者消费者模型，其实算是一种漏桶限流算法。





令牌桶算法，相对漏桶算法来说，它可以处理突发流量的问题。

它的核心思想是，令牌桶以恒定速率去生成令牌保存到令牌桶里面，桶的大小是固定的，令牌桶满了以后就不再生成令牌。

每个客户端请求进来的时候，必须要从令牌桶获得一个令牌才能访问，否则排队等待。

在流量低峰的时候，令牌桶会出现堆积，因此当出现瞬时高峰的时候，有足够多的令牌可以获取，因此令牌桶能够允许瞬时流量的处理。

网关层面的限流、或者接口调用的限流，都可以使用令牌桶算法，像 Google 的 Guava，和 Redisson 的限流，都用到了令牌桶算法

在我看来，限流的本质是实现系统保护，最终选择什么样的算法，一方面取决于统计的精准度，另一方面考虑限流维度和场景的需求。

以上就是我对这个问题的理解

## 面试点评

英国生物学家 Charles Darwin(查尔斯.达尔文)说过。

最终能够在社会上生存下来的人，不是强者，也不是智者，而是能够适应改变的人。技术开发虽然是谋生手段，但是技术能力的高低决定了职业发展的高度。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了，

喜欢的朋友记得点赞和收藏。

有任何工作和学习上的问题，可以随时私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 什么是可重入，什么是可重入锁？它用来解决什么问题？

---

一个工作了 3 年的粉丝，去一个互联网公司面试，结果被面试官怼了。

面试官说：“这么简单的问题你都不知道？没法聊了，回去等通知吧”。

这个问题是：“什么是可重入锁，以及它的作用是什么？”

对于这个问题，来看看普通人和高手的回答吧

### 普通人

### 高手

---

好的。

可重入是多线程并发编程里面一个比较重要的概念，

简单来说，就是在运行的某个函数或者代码，因为抢占资源或者中断等原因导致函数或者代码的运行中断，

等待中断程序执行结束后，重新进入到这个函数或者代码中运行，并且运行结果不会受到影响，那么这个函数或者代码就是可重入的。

(如图)而可重入锁，简单来说就是一个线程如果抢占到了互斥锁资源，在锁释放之前再去竞争同一把锁的时候，不需要等待，只需要记录重入次数。

在多线程并发编程里面，绝大部分锁都是可重入的，比如 **Synchronized**、**ReentrantLock** 等，但是也有不支持重入的锁，比如 **JDK8** 里面提供的读写锁 **StampedLock**。

```
public static synchronized void lock1(){  
    //ThreadX 获取到了lock1中的Synchronized锁,  
    // 再次调用另外一个加同步锁的lock2()方法  
    lock2();  
}  
  
public static synchronized void lock2(){  
    //doSomething  
}
```

锁的可重入性，主要解决的问题是避免线程死锁的问题。

因为一个已经获得同步锁 X 的线程，在释放锁 X 之前再去竞争锁 X 的时候，相当于会出现自己要等待自己释放锁，这很显然是无法成立的。

以上就是我对这个问题的理解。

## 面试点评

关于这个问题，其实是考察求职者的基础知识。

互联网大厂对基础的考察会特别深，有必要的話还是需要在工作之外去多花一点时间研究。

并且，对于 3 年工作经验，考察这类问题也不算过分。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 请你简单说一下 Mysql 的事务隔离级别

一个工作了 6 年的粉丝，去阿里面试，在第一面的时候被问到“Mysql 的事务隔离级别”。

他竟然没有回答上来，一直在私信向我诉苦。

我说，你只能怪年轻时候的你，那个时候不够努力导致现在的你技术水平不够。

好吧，关于这个问题，看看普通人和高手的回答。

## 普通人

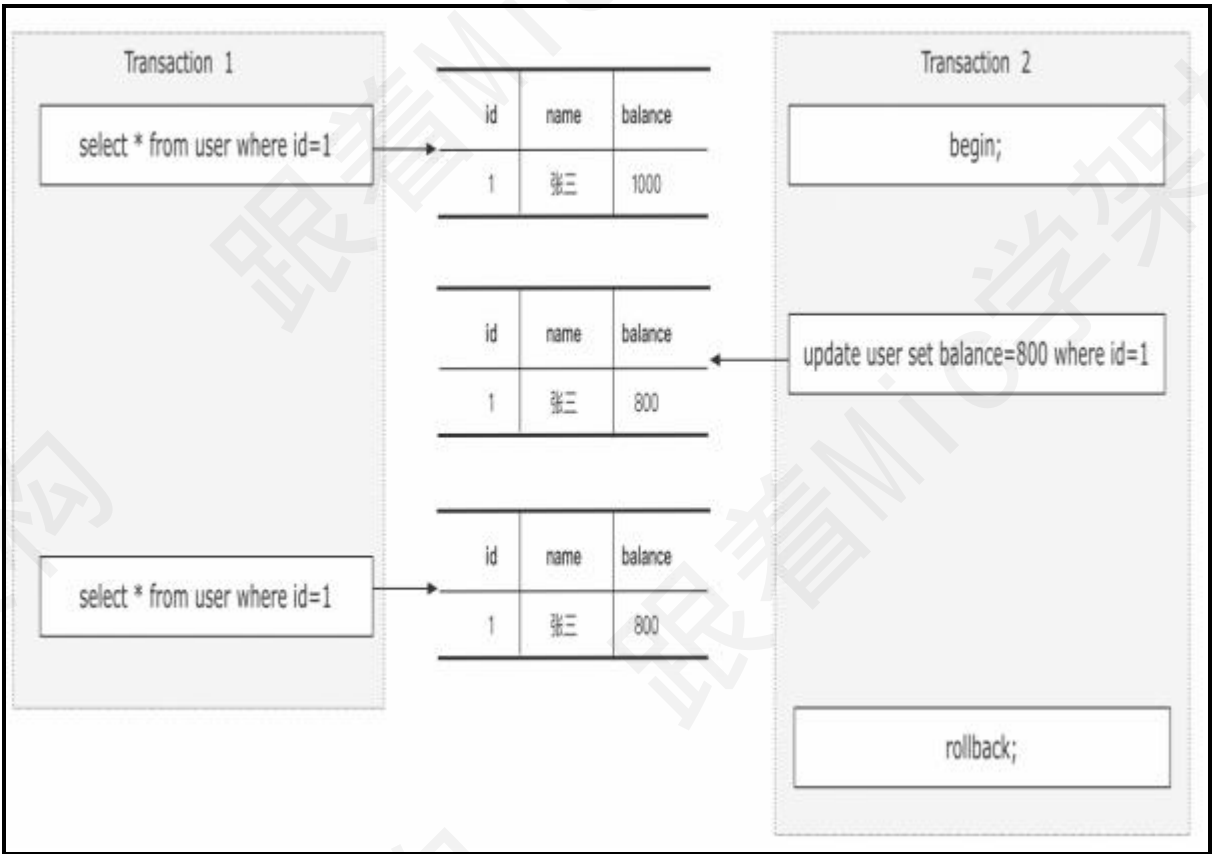
## 高手

好的，关于这个问题，我会从几个方面来回答。

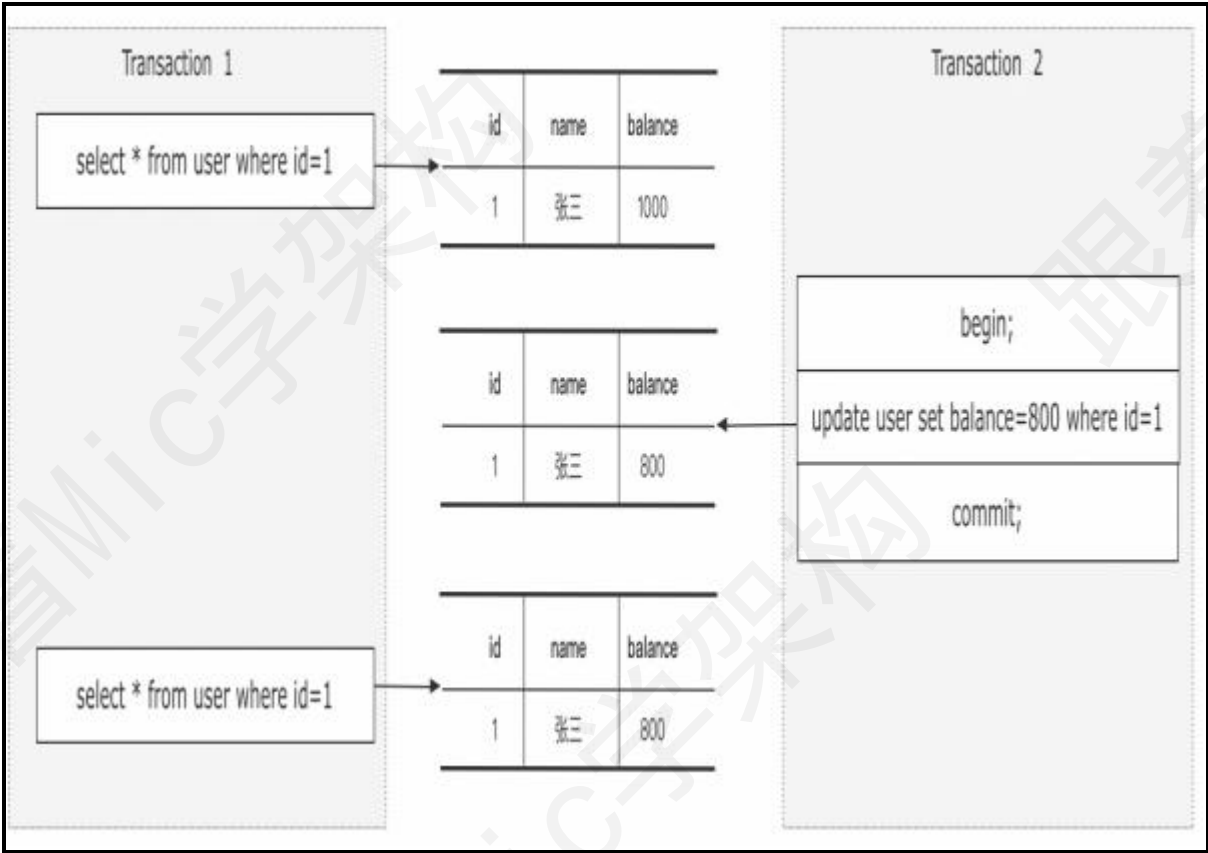
首先，事务隔离级别，是为了解决多个并行事务竞争导致的数据安全问题的一种规范。

具体来说，多个事务竞争可能会产生三种不同的现象。

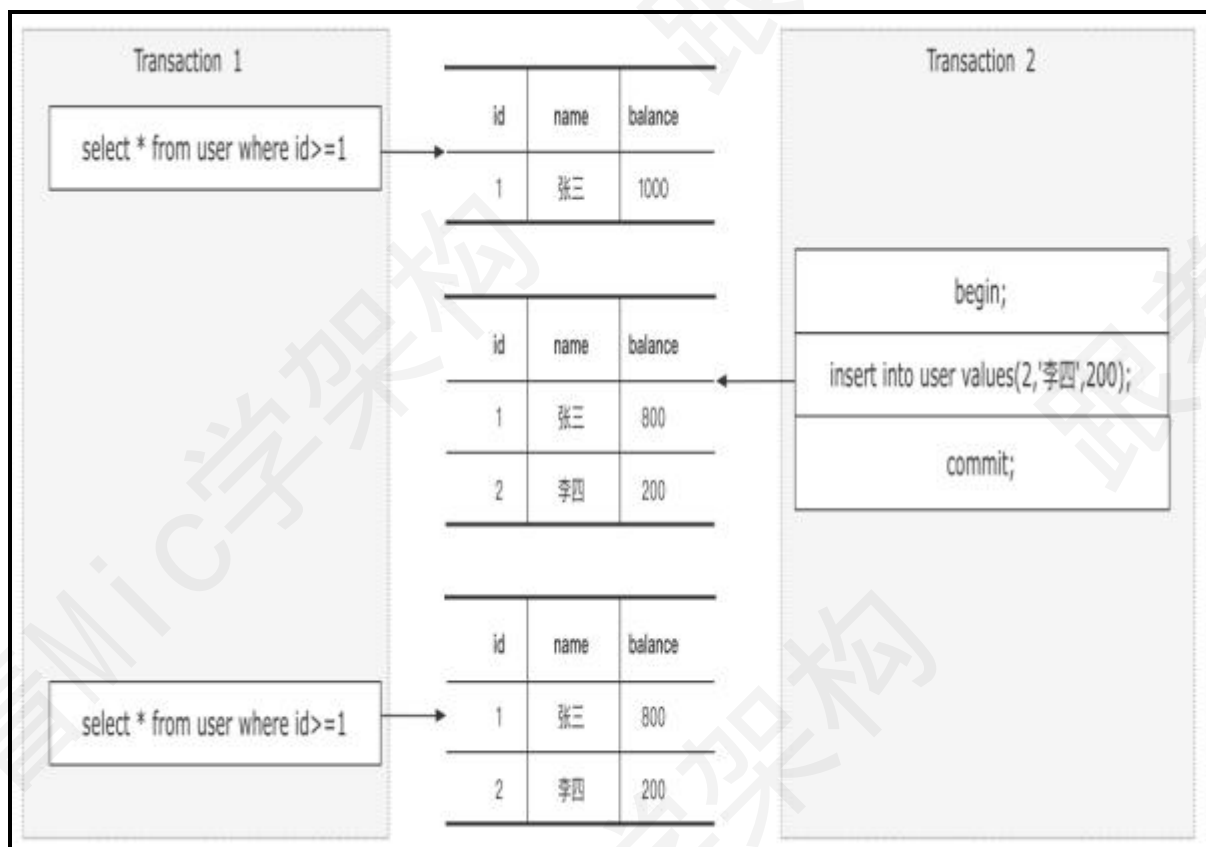
假设有两个事务 T1/T2 同时在执行，T1 事务有可能会读取到 T2 事务未提交的数据，但是未提交的事务 T2 可能会回滚，也就导致了 T1 事务读取到最终不一定存在的数据产生脏读的现象。



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。



，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。



而这三种现象在实际应用中，可能有些场景不能接受某些现象的存在，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，但也意味着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。

以上就是我对这个问题的理解。

## 面试点评

关于这个问题，很多用 Mysql5 年甚至更长时间的程序员都不一定非常清楚知道。

这其实是不正常的，因为虽然 InnoDB 默认隔离级别能解决 99% 以上的问题，但是有些公司的某些业务可能会修改隔离级别。

而如果你不知道，就很可能在程序中出现莫名其妙的问题。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 请说一下 ReentrantLock 的实现原理？

一个工作了 3 年的粉丝私信我，在面试的时候遇到了这样一个问题。

“请说一下 ReentrantLock 的实现原理”，他当时根据自己的理解零零散散的说了一些。

但是似乎没有说到关键点，让我出一期视频说一下回答思路。

好吧，关于这个问题，我们来看看普通人和高手的回答。

### 普通人

### 高手

好的，面试官，关于这个问题，我会从这几个方面来回答。

什么是 ReentrantLock

ReentrantLock 的特性

ReentrantLock 的实现原理

首先，ReentrantLock 是一种可重入的排它锁，主要用来解决多线程对共享资源竞争的问题。

它的核心特性有几个：

它支持可重入，也就是获得锁的线程在释放锁之前再次去竞争同一把锁的时候，不需要加锁就可以直接访问。

它支持公平和非公平特性

它提供了阻塞竞争锁和非阻塞竞争锁的两种方法，分别是 lock() 和 tryLock()。

然后，**ReentrantLock** 的底层实现有几个非常关键的技术。

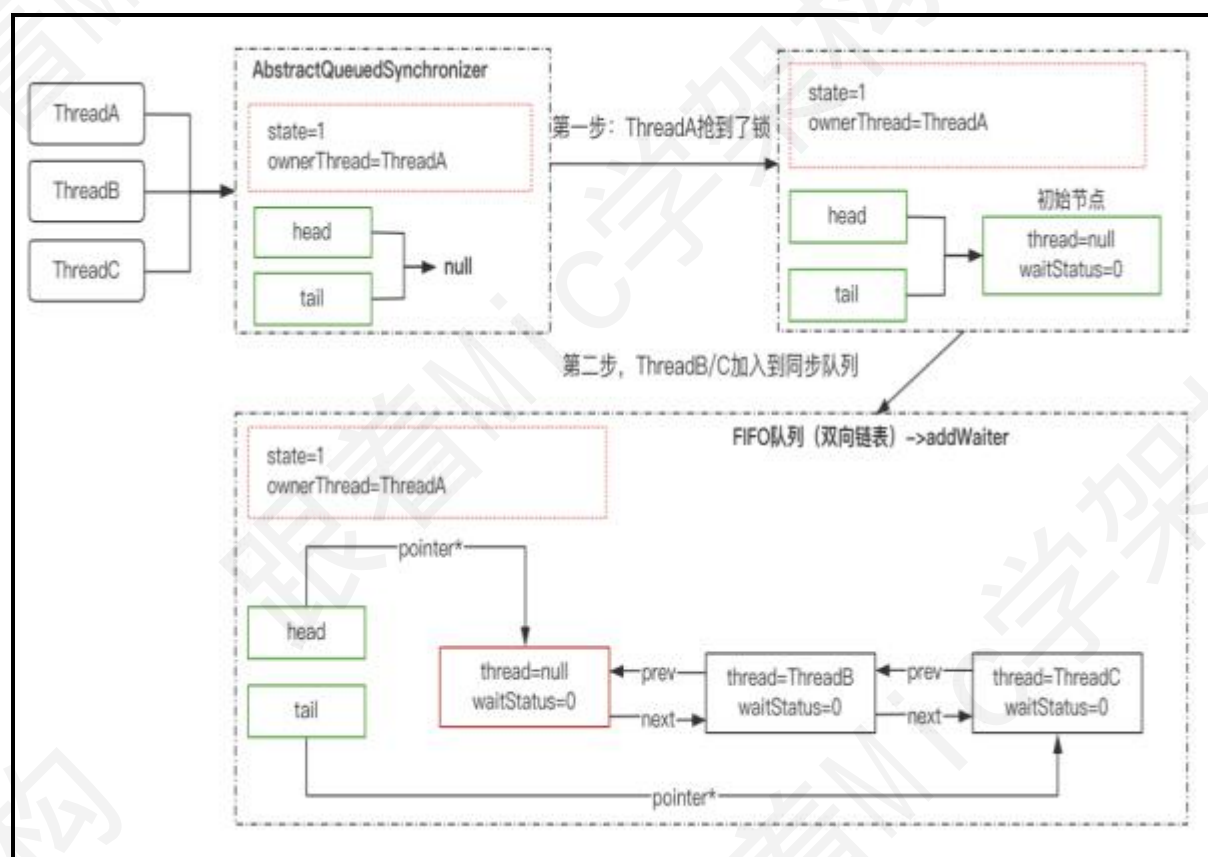
锁的竞争，**ReentrantLock** 是通过互斥变量，使用 **CAS** 机制来实现的。

没有竞争到锁的线程，使用了 **AbstractQueuedSynchronizer** 这样一个队列同步器来存储，底层是通过双向链表来实现的。当锁被释放之后，会从 **AQS** 队列里面的头部唤醒下一个等待锁的线程。

公平和非公平的特性，主要是体现在竞争锁的时候，是否需要判断 **AQS** 队列存在等待中的线程。

最后，关于锁的重入特性，在 **AQS** 里面有一个成员变量来保存当前获得锁的线程，当同一个线程下次再来竞争锁的时候，就不会去走锁竞争的逻辑，而是直接增加重入次数。

以上就是我对这个问题的理解。



## 面试点评

这道题很简单，但是要回答好，有两个关键点。

大家必须要理解 **ReentrantLock** 的整个设计思想



表达一定要清晰有条理

还是那句话，虽然基础，但很重要。地基的深度决定了楼层的高度。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Mybatis 中#{ }和\${ }的区别是什么？

一个工作 2 年的粉丝，被问到 Mybatis 里面的基础问题。

他跑过来调戏我，说 Mic 老师，你要是能把这个问题回答到一定高度，请我和一个月奶茶。

这个问题是：“Mybatis 里面#{ }和\${ }的区别是什么”

下面看看普通人和高手对这个问题的回答。

### 普通人

### 高手

好的，关于这个问题我从几个方面来回答。

首先，Mybatis 提供到的#号占位符和\$号占位符，都是实现动态 SQL 的一种方式，通过这两种方式把参数传递到 XML 之后，

在执行操作之前，Mybatis 会对这两种占位符进行动态解析。

#号占位符，等同于 jdbc 里面的 ? 号占位符。

它相当于向 PreparedStatement 中的预处理语句中设置参数，

而 PreparedStatement 中的 sql 语句是预编译的，SQL 语句中使用了占位符，规定了 sql 语句的结构。

并且在设置参数的时候，如果有特殊字符，会自动进行转义。

所以#号占位符可以防止 SQL 注入。



```
String sql = "UPDATE Employees set age=? WHERE id=?";  
PreparedStatement stmt = conn.prepareStatement(sql);  
  
//Bind values into the parameters.  
stmt.setInt(1, 18); // This would set age  
stmt.setInt(2, 101); // This would set ID
```

而使用\$的方式传参，相当于直接把参数拼接到了原始的 SQL 里面，Mybatis 不会对它进行特殊处理。



```
SELECT id,name FROM ${table} WHERE id =${id};
```

```
// 假设传递两个参数分别是 table=student 和 id=1  
// 就会得到下面这个语句
```

```
SELECT id,name FROM student WHERE id =1;
```

所以\$和#最大的区别在于，前者是动态参数，后者是占位符，动态参数无法防止 SQL 注入的问题，所以在实际应用中，应该尽可能的使用#号占位符。

另外，\$符号的动态传参，可以适合应用在一些动态 SQL 场景中，比如动态传递表名、动态设置排序字段等。

以上就是我对这个问题的理解。

## 面试点评

一些小的细节如果不注意，就有可能造成巨大的经济损失。

比如现如今还是会有一些网站出现 SQL 注入导致信息泄露的问题。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## Mysql 为什么使用 B+Tree 作为索引结构

一个工作 8 年的粉丝私信了我一个问题。

他说这个问题是去阿里面试的时候被问到的，自己查了很多资料也没搞明白，希望我帮他解答。

问题是：“Mysql 为什么使用 B+Tree 作为索引结构”

关于这个问题，看看普通人和高手的回答。

## 普通人

## 高手

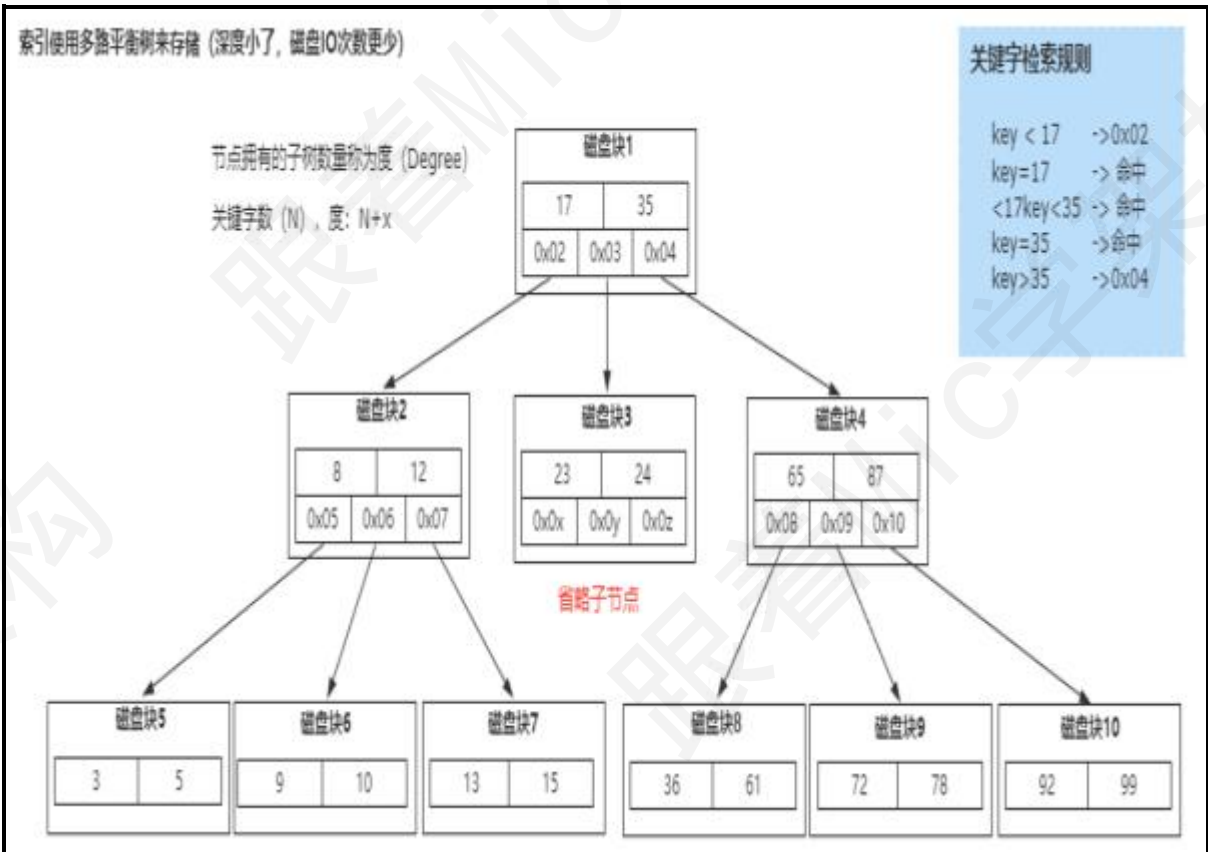
关于这个问题，我从几个方面来回答。

首先，常规的数据库存储引擎，一般都是采用 B 树或者 B+树来实现索引的存储。

因为 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。

而对于数据库来说，所有的数据必然都是存储在磁盘上的，而磁盘 IO 的效率实际上是很低的，特别是在随机磁盘 IO 的情况下效率更低。

所以树的高度能够决定磁盘 IO 的次数，磁盘 IO 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。

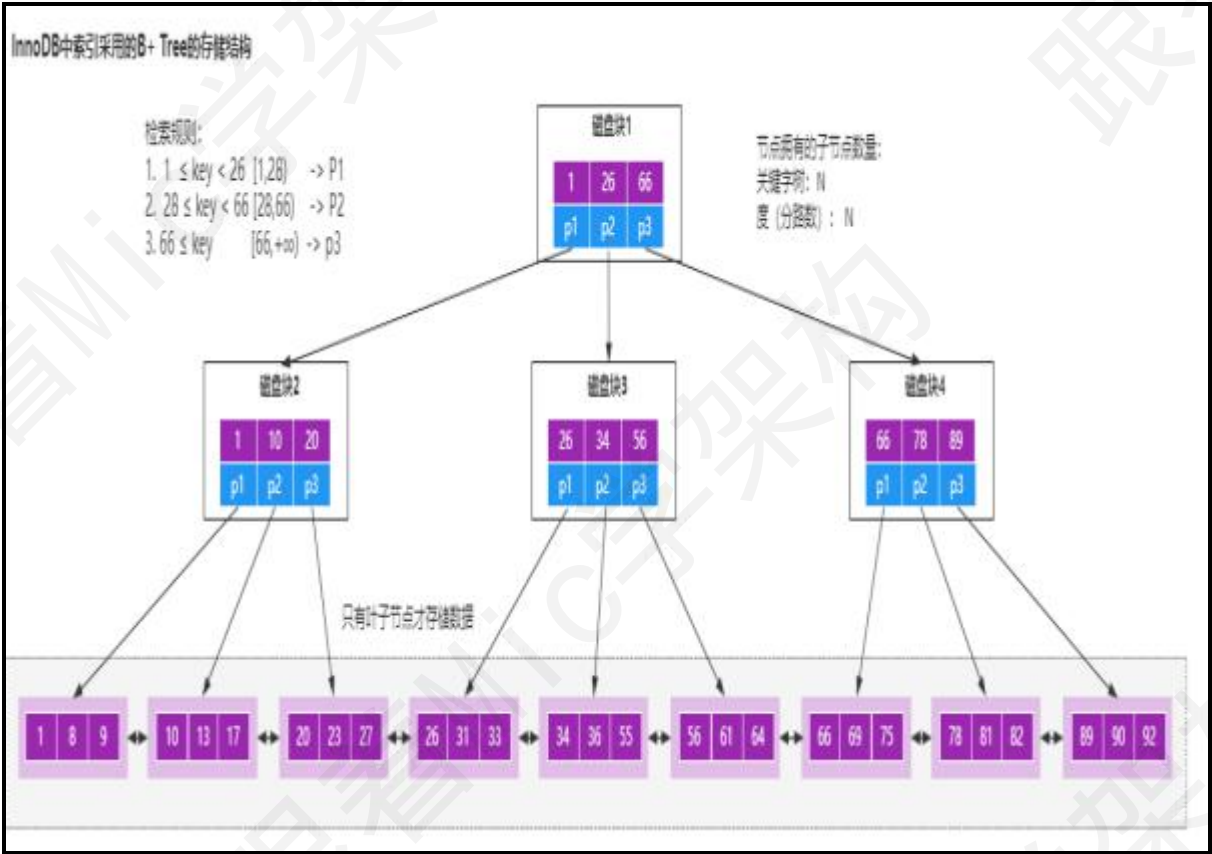


但是在 Mysql 的 InnoDB 存储引擎里面，它用了一种增强的 B 树结构，也就是 B+树来作为索引和数据的存储结构。

相比较于 B 树结构，B+树做了几个方面的优化。

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。

叶子节点中的数据使用双向链表的方式进行关联。



使用 B+树来实现索引的原因，我认为有几个方面。

B+树非叶子节点不存储数据，所以每一层能够存储的索引数量会增加，意味着 B+树在层高相同的情况下存储的数据量要比 B 树要多，使得磁盘 IO 次数更少。

在 Mysql 里面，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，所以 B+树在范围查询上效率更高。

在数据检索方面，由于所有的数据都存储在叶子节点，所以 B+树的 IO 次数会更加稳定一些。

因为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。但是 B 树需要遍历整个树。

另外，基于 B+ 树这样一种结构，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。

总的来说，我认为技术方案的选型，更多的是去解决当前场景下的特定问题，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。

以上就是我对这个问题的理解。

## 面试点评

---

对于“为什么要选择 xx 技术”的问题，其实很好回答。

只要你对这个技术本身的特性足够了解，那么自然就知道为什么要这么设计。

就像，我们在业务开发中，知道什么时候使用 List，什么时候使用 Map，道理是一样的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见。

## 数据库连接池有什么用？它有哪些关键参数？

---

一个工作 5 年的粉丝找到我，他说参加美团面试，遇到一个基础题没回答上来。

这个问题是：“数据库连接池有什么用？以及它有哪些关键参数”？

我说，这个问题都不知道，那你项目里面的连接池配置怎么设置的？你们猜他怎么回答。懂得懂得啊。

好的，关于这个问题，我们来看看普通人和高手的回答。

### 普通人

### 高手

---

关于这个问题，我从这几个方面来回答。

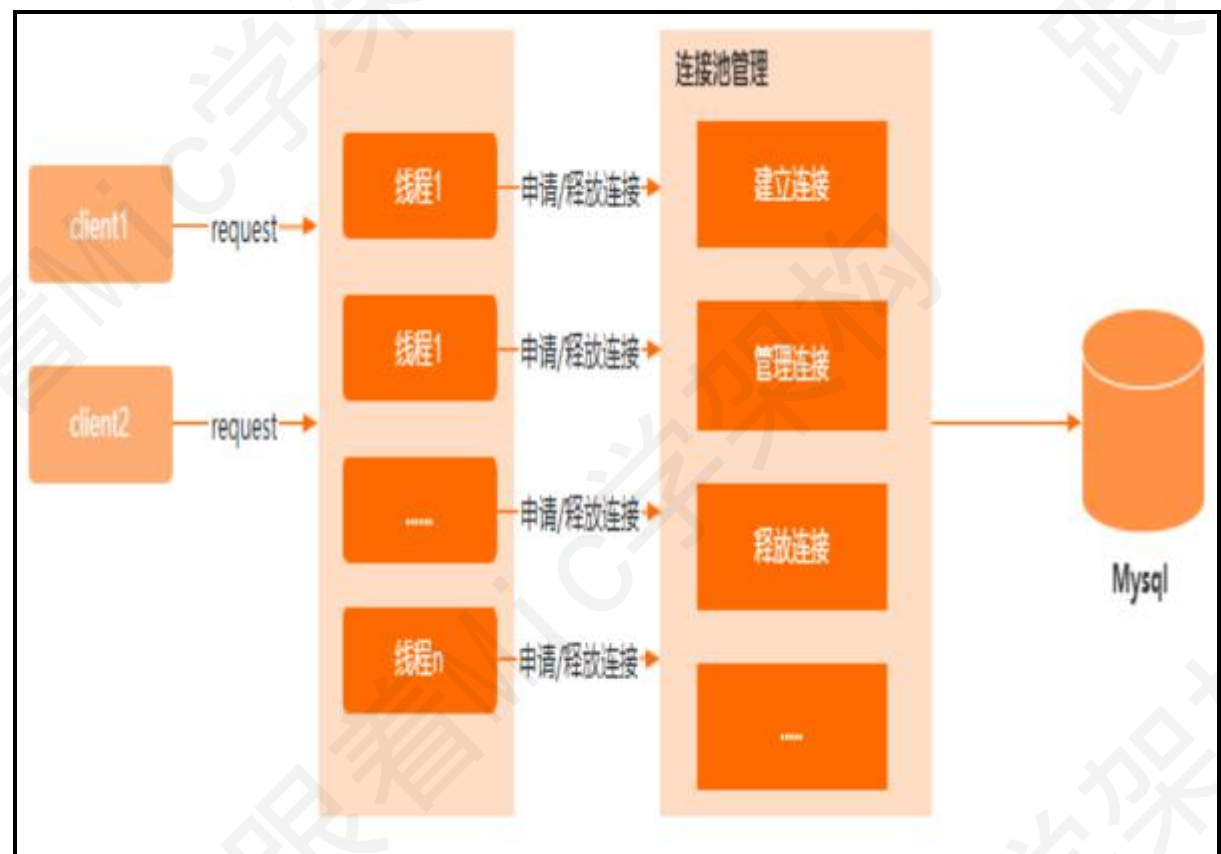
首先，数据库连接池是一种池化技术，池化技术的核心思想是实现资源的复用，避免资源重复创建销毁的开销。

而在数据库的应用场景里面，应用程序每次向数据库发起 **CRUD** 操作的时候，都需要创建连接

在数据库访问量较大的情况下，频繁的创建连接会带来较大的性能开销。

而连接池的核心思想，就是应用程序在启动的时候提前初始化一部分连接保存到连接池里面，当应用需要使用连接的时候，直接从连接池获取一个已经建立好的链接。

连接池的设计，避免了每次连接的建立和释放带来的开销。



连接池的参数有很多，不过关键参数就几个：

首先是，连接池初始化的时候会有几个关键参数：

初始化连接数，表示启动的时候初始多少个连接保存到连接池里面。

最大连接数，表示同时最多能支持多少连接，如果连接数不够，后续要获取连接的线程会阻塞。

最大空闲连接数，表示没有请求的时候，连接池中要保留的最大空闲连接。

最小空闲连接，当连接数小于这个值的时候，连接池需要再创建连接来补充到这个值。

然后，就是在使用连接的时候的关键参数：

最大等待时间，就是连接池里面的连接用完了以后，新的请求要等待的时间，超过这个时间就会提示超时异常。

无效连接清除，清理连接池里面的无效连接，避免使用这个连接操作的时候出现错误。

不同的连接池框架，除了核心的参数以外，还有很多业务型的参数，比如是否要检测连接 `sql` 的有效性、连接初始化 `SQL` 等等，这些配置参数可以在使用的时候去查询 `api` 文档就可以知道。

以上就是我对这个问题的理解。

## 面试点评

---

这个问题更进一步去问，就会问到最大连接数、最小连接数应该如何设置？

连接池的实现原理啊等等。

所以建议各位粉丝还是要有一个系统化的学习。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

如果有任何面试问题、职业发展问题、学习问题，都可以私信我。

我是 Mic，一个工作了 14 年的 `Java` 程序员，咱们下期再见。

## TCP 协议为什么要设计三次握手？

---

一个工作 5 年的粉丝，最近去面试了很多公司，每次都被各种技术原理题问得语无伦次。

由于找了快 1 个月时间的工作，有点焦虑，来向我求助。

我能做的只是保证每天更新一个面试题，然后问他印象最深刻的一个面试题是什么，他说。

“TCP 协议为什么要设计三次握手”。

这个问题的高手回答，我整理成了文档，大家可以在我主页加 V 领取。

好的，关于这个问题，我们来看看普通人和高手的回答。

### 普通人

### 高手

---



关于这个问题，我会从下面 3 个方面来回答。

**TCP** 协议，是一种可靠的，基于字节流的，面向连接的传输层协议。

可靠性体现在 **TCP** 协议通信双方的数据传输是稳定的，即便是在网络不好的情况下，**TCP** 都能够保证数据传输到目标端，而这个可靠性是基于数据包确认机制来实现的。

**TCP** 通信双方的数据传输是通过字节流来实现传输的

面向连接，是说数据传输之前，必须要建立一个连接，然后基于这个连接进行数据传输

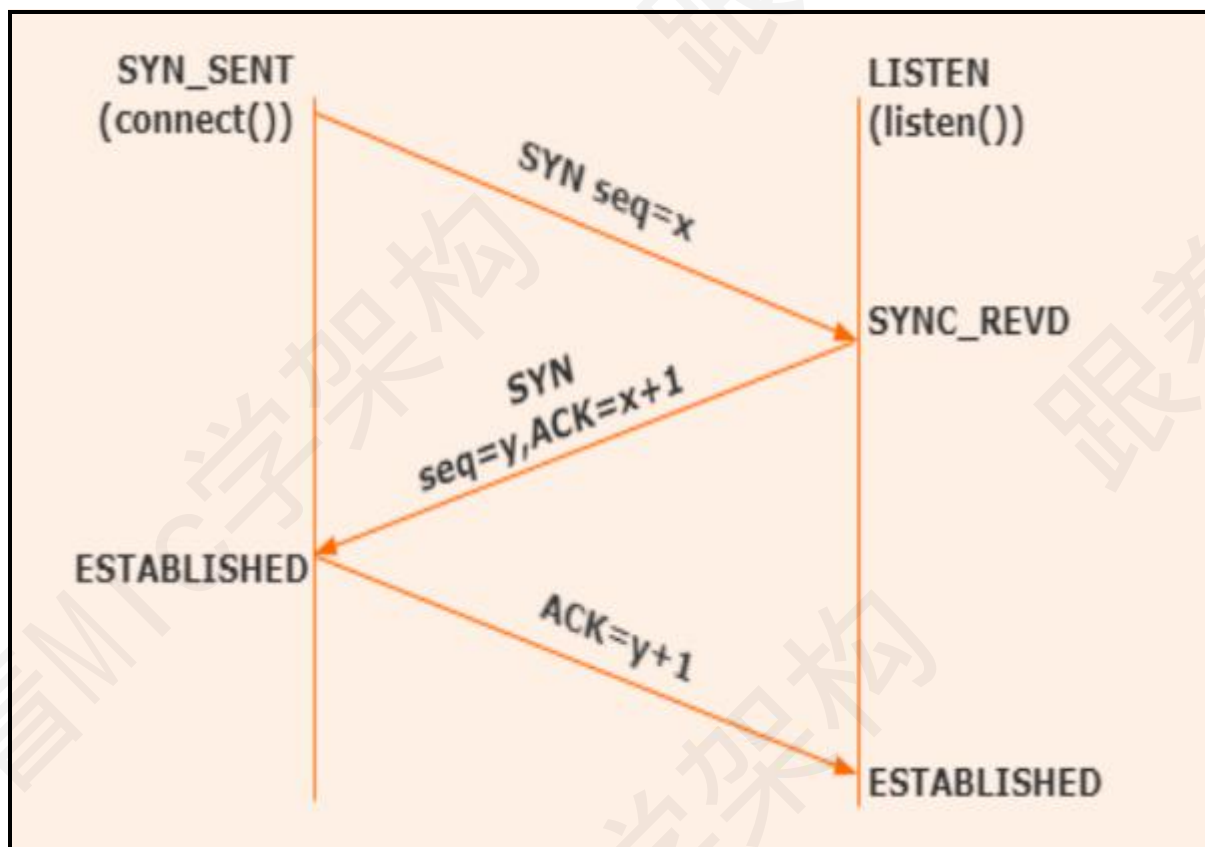
因为 **TCP** 是面向连接的协议，所以在进行数据通信之前，需要建立一个可靠的连接，**TCP** 采用了三次握手的方式来实现连接的建立。

所谓的三次握手，就是通信双方一共需要发送三次请求，才能确保这个连接的建立。

客户端向服务端发送连接请求并携带同步序列号 **SYN**。

服务端收到请求后，发送 **SYN** 和 **ACK**，这里的 **SYN** 表示服务端的同步序列号，**ACK** 表示对前面收到请求的一个确认，表示告诉客户端，我收到了你的请求。

客户端收到服务端的请求后，再次发送 **ACK**，这个 **ACK** 是针对服务端连接的一个确认，表示告诉服务端，我收到了你的请求。



之所以 TCP 要设计三次握手，我认为有三个方面的原因：

TCP 是可靠性通信协议，所以 TCP 协议的通信双方都必须要维护一个序列号，去标记已经发送出去的数据包，哪些是已经被对方签收的。而三次握手就是通信双方相互告知序列号的起始值，为了确保这个序列号被收到，所以双方都需要有一个确认的操作。

TCP 协议需要在一个不可靠的网络环境下实现可靠的数据传输，意味着通信双方必须要通过某种手段来实现一个可靠的数据传输通道，而三次通信是建立一个通道的最小值。当然还可以四次、五次，只是没必要浪费这个资源。

防止历史的重复连接初始化造成的混乱问题，比如说在网络比较差的情况下，客户端连续多次发送建立连接的请求，假设只有两次握手，那么服务端只能选择接受或者拒绝这个连接请求，但是服务端不知道这次请求是不是之前因为网络堵塞而过期的请求，也就是说服务端不知道当前客户端的连接是有效还是无效。

以上就是我对这个问题的理解。

## 面试点评

网络通信这块内容还是比较重要的，面对一些线上网络故障排查的时候，

可以快速的去帮助我们定位问题，并找到解决办法。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了

我是 Mic，一个工作了 14 年的 Java 程序员，咱们下期再见！

## 请简单说一下你对受检异常和非受检异常的理解

---

Hi，我是 Mic

今天给大家分享一道阿里一面的面试题。

这道题目比较基础，但是确难倒了很多。

关于“受检异常和非受检异常的理解”

我们来看看普通人和高手的回答。

另外，高手部分的回答已经整理成了文档，有需要的小伙伴可以主页加 V 领取

### 普通人

### 高手

---

##

好的。

所谓的受检异常，表示在编译的时候强制检查的异常，这种异常需要显示的通过 try/catch 来捕捉，或者通过 throws 抛出去，否则从程序无法通过编译。



而非受检异常，表示在编译器可以不需要强制检查的异常，这种异常不需要显示去捕捉。

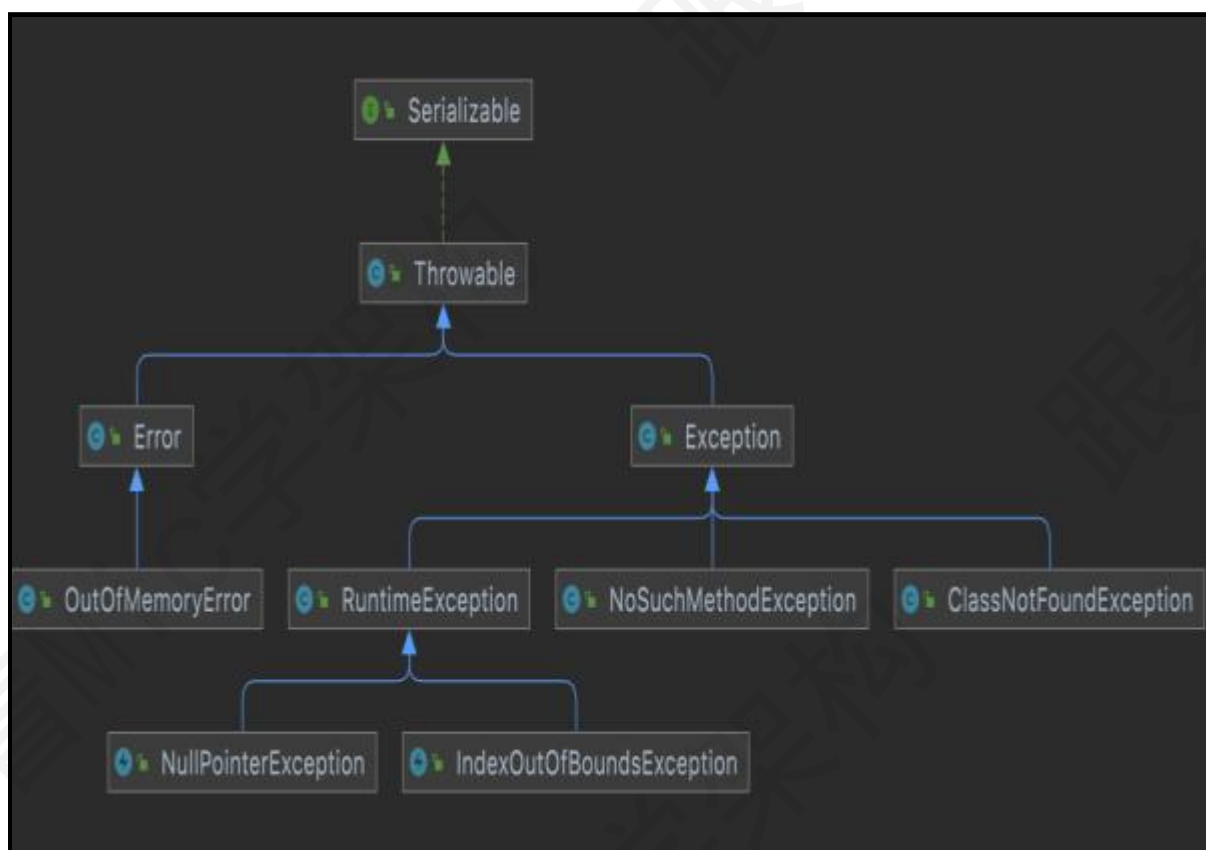
在 Java 里面，所有的异常都是继承自 `java.lang.Throwable` 类，`Throwable` 有两个直接子类，`Error` 和 `Exception`。

`Error` 用来表示程序底层或者硬件有关的错误，这种错误和程序本身无关，比如常见的 `OOM` 异常。这种异常和程序本身无关，所以不需要检查，属于非受检异常。

`Exception` 表示程序中的异常，可能是由于程序不严谨导致的，比如 `NullPointerException`。

`Exception` 下面派生了 `RuntimeException` 和其他异常，其中 `RuntimeException` 运行时异常，也是属于非受检异常。

所以，除了 `Error` 和 `RuntimeException` 及派生类以外，其他异常都是属于受检异常，比如 `IOException`、`SQLException`。



之所以在 **Java** 中要设计一些强制检查的异常，我认为主要原因是考虑到程序的正确性、稳定性和可靠性。

比如数据库异常、文件读取异常，这些异常是程序无法提前预料到的，但是一旦出现问题，就会造成资源被占用导致程序出现问题。

所以这些异常我们需要主动捕获，一旦出现问题，我们可以做出相应的处理，比如关闭数据库连接、文件流的释放等。

以上就是我对这个问题的理解！

## 面试点评

这个问题并不难，但是在实际工作中，如何用好异常又显得很重要。

从高手的回答中可以明显看到他对异常的理解层次是比较深的，分别介绍了受检和非受检异常，

以及在 **Java** 中这两种异常是如何分类，最后说明了这两种异常的价值。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

喜欢我的作品的小伙伴记得点赞和收藏加关注。

我是 Mic，一个工作 14 年的 **Java** 程序员，咱们下期再见！

# 为什么引入偏向锁、轻量级锁，介绍下升级流程

---

Hi，我是 Mic

一个工作了 7 年的粉丝来找我，他说最近被各种锁搞晕了。

比如，共享锁、排它锁、偏向锁、轻量级锁、自旋锁、重量级锁、间隙锁、临键锁、意向锁、读写锁、乐观锁、悲观锁、表锁、行锁。

然后前两天去面试，被问到偏向锁、轻量级锁，结果没回答上来。

ok，关于 **Synchronized** 锁升级的原理，看看普通人和高手的回答。

另外，高手的回答已经整理成了文档，有需要的小伙伴可以主页加 V 领取

## 普通人

## 高手

---

好的，面试官。

**Synchronized** 在 jdk1.6 版本之前，是通过重量级锁的方式来实现线程之间锁的竞争。

之所以称它为重量级锁，是因为它的底层底层依赖操作系统的 **Mutex Lock** 来实现互斥功能。

**Mutex** 是系统方法，由于权限隔离的关系，应用程序调用系统方法时需要切换到内核态来执行。

这里涉及到用户态向内核态的切换，这个切换会带来性能的损耗。



在 jdk1.6 版本中，**synchronized** 增加了锁升级的机制，来平衡数据安全性和性能。简单来说，就是线程去访问 **synchronized** 同步代码块的时候，**synchronized** 根据

线程竞争情况，会先尝试在不加重量级锁的情况下去保证线程安全性。所以引入了偏向锁和轻量级锁的机制。

偏向锁，就是直接把当前锁偏向于某个线程，简单来说就是通过 **CAS** 修改偏向锁标记，这种锁适合同一个线程多次去申请同一个锁资源并且没有其他线程竞争的场景。

轻量级锁也可以称为自旋锁，基于自适应自旋的机制，通过多次自旋重试去竞争锁。自旋锁优点在于它避免避免了用户态到内核态的切换带来的性能开销。

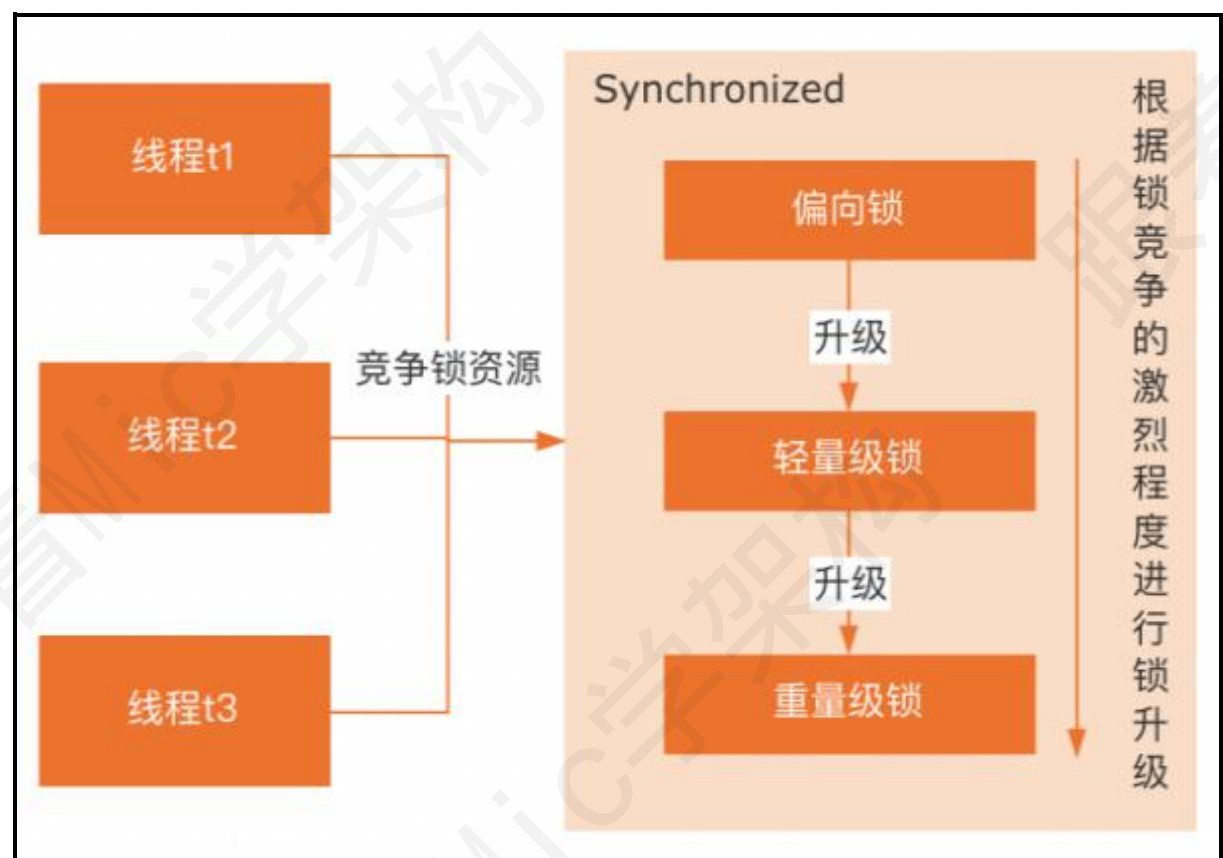
**Synchronized** 引入了锁升级的机制之后，如果有线程去竞争锁：

首先，**synchronized** 会尝试使用偏向锁的方式去竞争锁资源，如果能够竞争到偏向锁，表示加锁成功直接返回。如果竞争锁失败，说明当前锁已经偏向向了其他线程。

需要将锁升级到轻量级锁，在轻量级锁状态下，竞争锁的线程根据自适应自旋次数去尝试抢占锁资源，如果在轻量级锁状态下还是没有竞争到锁，

就只能升级到重量级锁，在重量级锁状态下，没有竞争到锁的线程就会被阻塞，线程状态是 **Blocked**。

处于锁等待状态的线程需要等待获得锁的线程来触发唤醒。



总的来说，**Synchronized** 的锁升级的设计思想，在我看来本质上是一种性能和安全性的平衡，也就是如何在不加锁的情况下能够保证线程安全性。

这种思想在编程领域比较常见，比如 **Mysql** 里面的 **MVCC** 使用版本链的方式来解决多个并行事务的竞争问题。

以上就是我对这个问题的理解。

## 面试点评

锁在程序中是非常常见的内容，我们几乎每天与锁打交道，比如 **Mysql** 里面的行锁、表锁。

因此它的重要性也不言而喻。

我们从高手的回答中可以明显的看到高手对 **Synchronized** 的理解层次是非常高的。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。



喜欢我的作品的小伙伴记得点赞和收藏加关注。

我是 Mic，一个工作 14 年的 Java 程序员，咱们下期再见！

## 介绍下 Spring IoC 的工作流程

---

Hi，我是 Mic

一个工作了 4 年的粉丝，在面试的时候遇到一个这样的问题。

“介绍一下 Spring IOC 的工作流程”

他说回答得不是很好，希望我能帮他梳理一下。

这个问题高手部分的回答已经整理成了文档，可以在主页加 V 领取。

关于这个问题，我们来看看普通人和高手的回答。

### 普通人

### 高手

---

好的，这个问题我会从几个方面来回答。

IOC 是什么

Bean 的声明方式

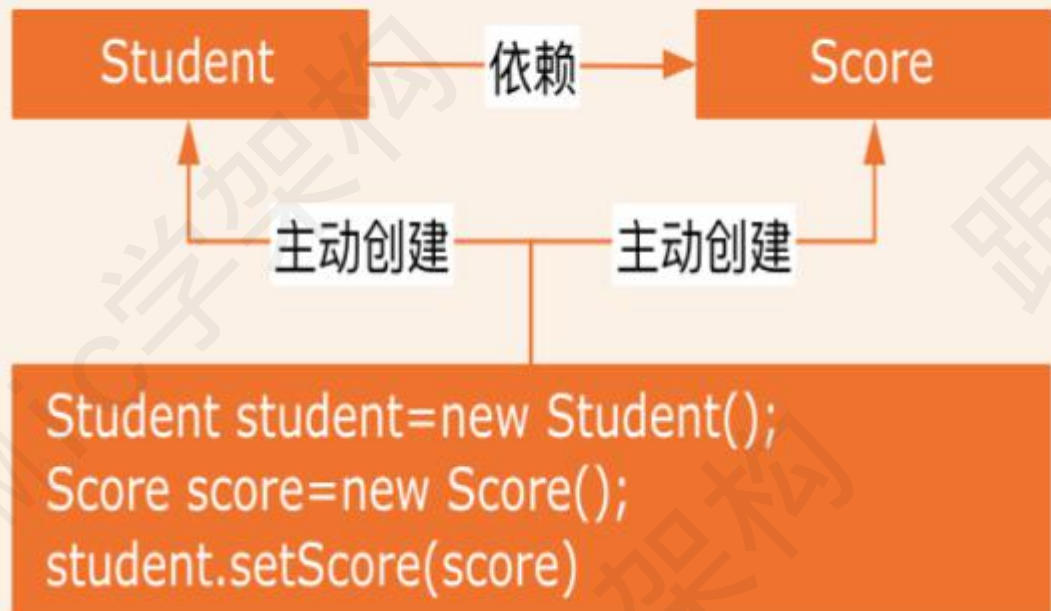
IOC 的工作流程

IOC 的全称是 Inversion Of Control,也就是控制反转，它的核心思想是把对象的管理权限交给容器。

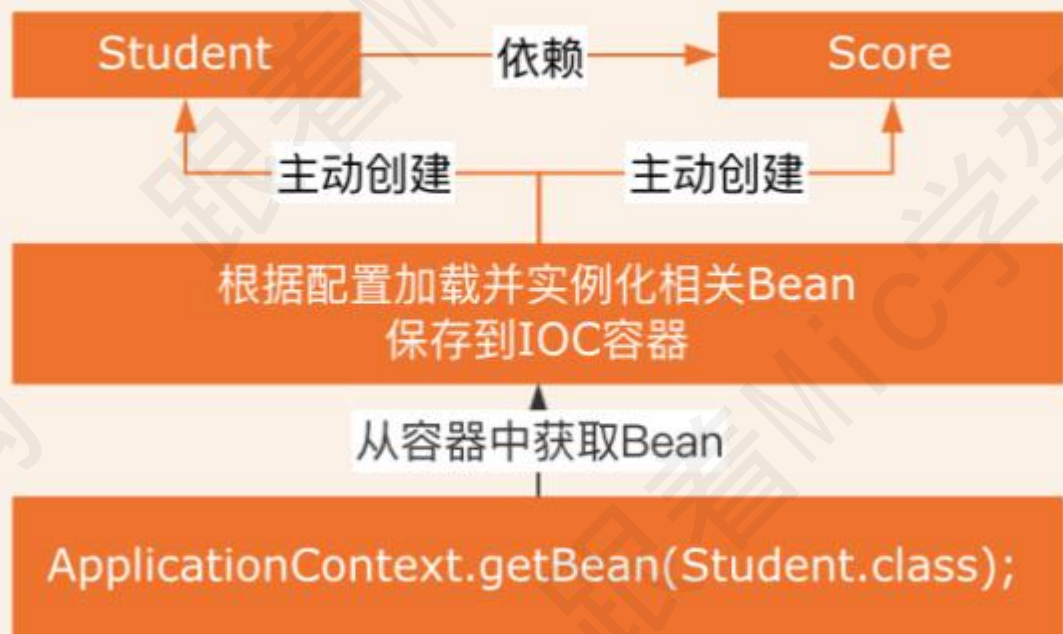
应用程序如果需要使用到某个对象实例，直接从 IOC 容器中去获取就行，这样设计的好处是降低了程序里面对象与对象之间的耦合性。

使得程序的整个体系结构变得更加灵活。

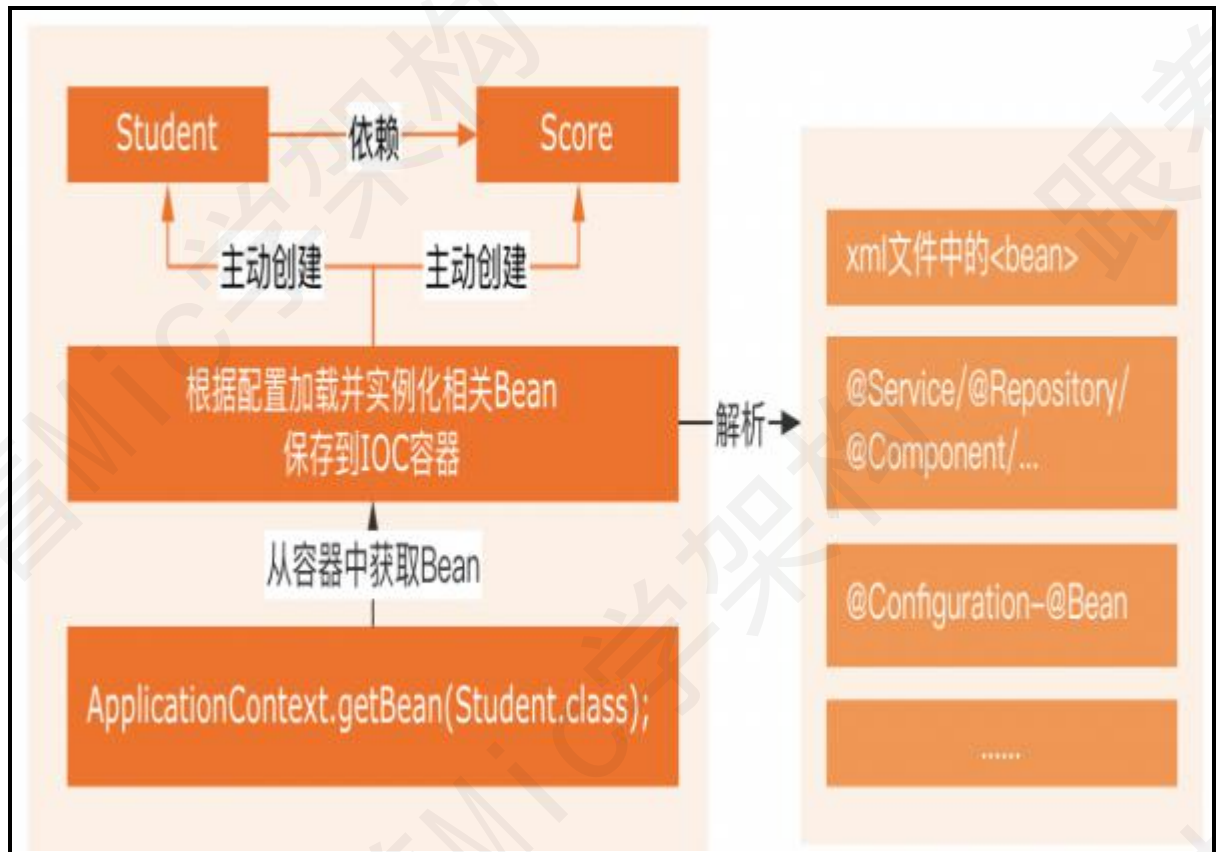
## 传统应用程序



## IOC控制反转



Spring 里面很多方式去定义 Bean，比如 XML 里面的<bean>标签、@Service、@Component、@Repository、@Configuration 配置类中的@Bean 注解等等。Spring 在启动的时候，会去解析这些 Bean 然后保存到 IOC 容器里面。

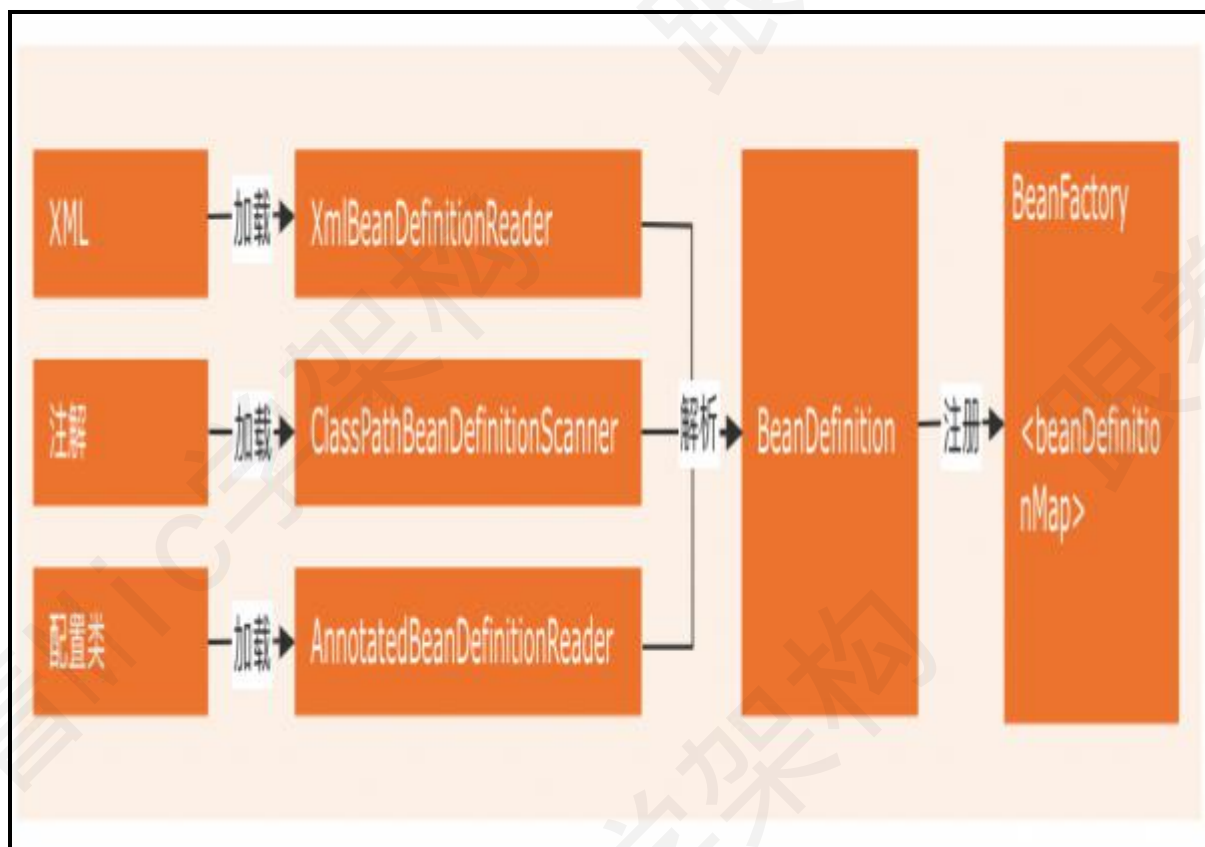


Spring IOC 的工作流程大致可以分为两个阶段。

第一个阶段，就是 IOC 容器的初始化

这个阶段主要是根据程序中定义的 XML 或者注解等 Bean 的声明方式

通过解析和加载后生成 BeanDefinition，然后把 BeanDefinition 注册到 IOC 容器。



通过注解或者 xml 声明的 bean 都会解析得到一个 **BeanDefinition** 实体，实体中包含这个 bean 中定义的基本属性。

最后把这个 **BeanDefinition** 保存到一个 **Map** 集合里面，从而完成了 IOC 的初始化。

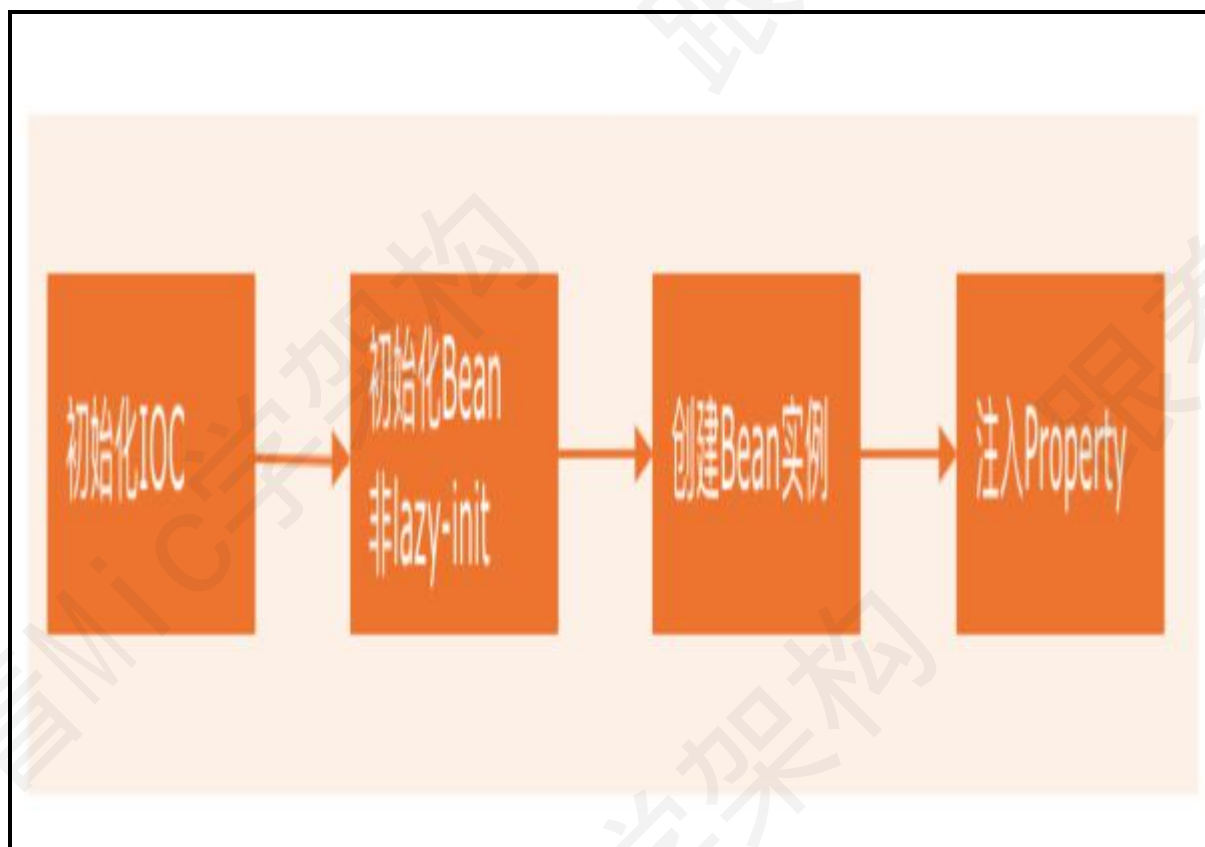
IoC 容器的作用就是对这些注册的 **Bean** 的定义信息进行处理和维护，它 IoC 容器控制反转的核心。

第二个阶段，完成 **Bean** 初始化及依赖注入

然后进入到第二个阶段，这个阶段会做两件事情

通过反射针对没有设置 **lazy-init** 属性的单例 **bean** 进行初始化。

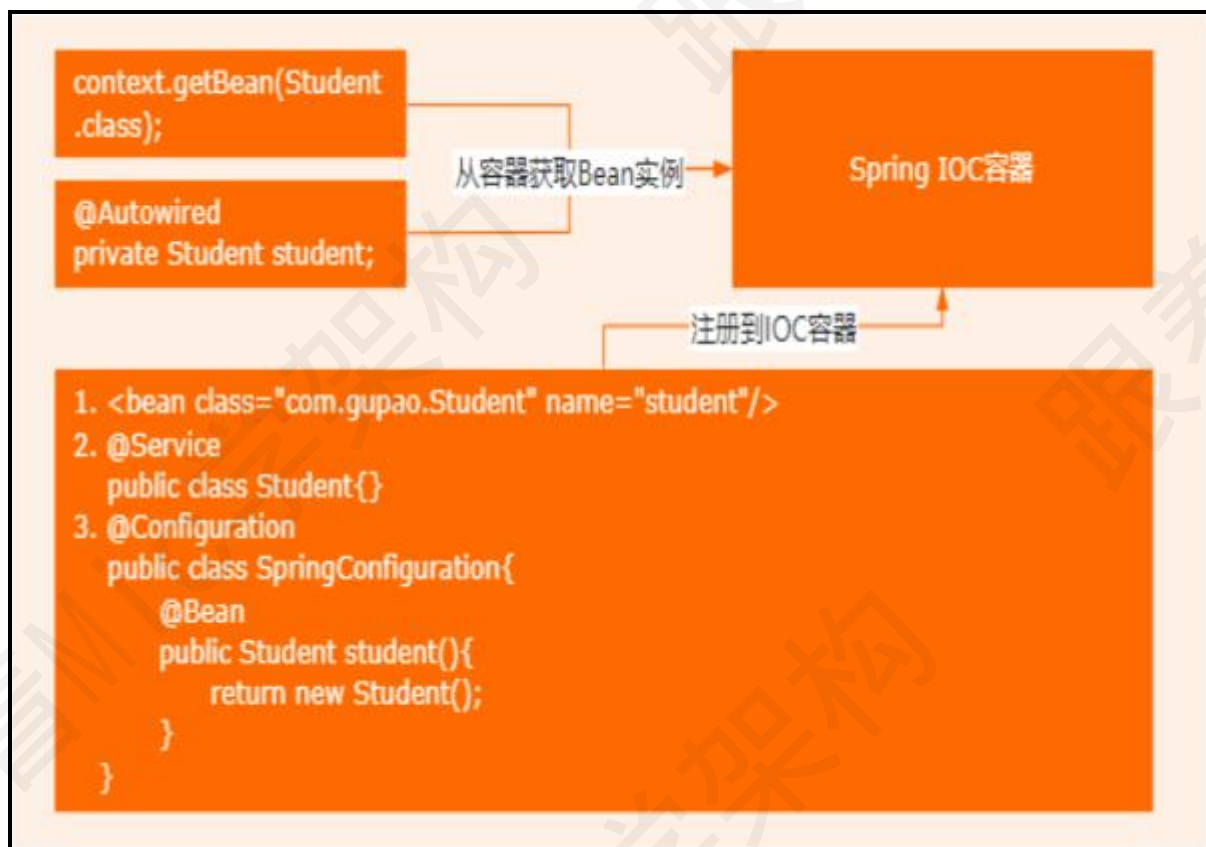
完成 **Bean** 的依赖注入。



### 第三个阶段，Bean 的使用

通常我们会通过 `@Autowired` 或者 `BeanFactory.getBean()` 从 IOC 容器中获取指定的 bean 实例。

另外，针对设置 `lazy-init` 属性以及非单例 bean 的实例化，是在每次获取 bean 对象的时候，调用 bean 的初始化方法来完成实例化的，并且 Spring IOC 容器不会去管理这些 Bean。



以上就是我对这个问题的理解。

## 面试点评

对于工作原理或者工作流程性的问题，大家一定要注意回答的结构和节奏。

否则面试官会觉得很混乱，无法理解，导致面试的效果大打折扣。

高手的回答逻辑非常清晰，大家可以参考。

好的，本期的普通人 VS 高手面试系列的视频就到这里结束了。

喜欢我的作品的小伙伴记得点赞和收藏加关注。

我是 Mic，一个工作 14 年的 Java 程序员，咱们下期再见！