

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。

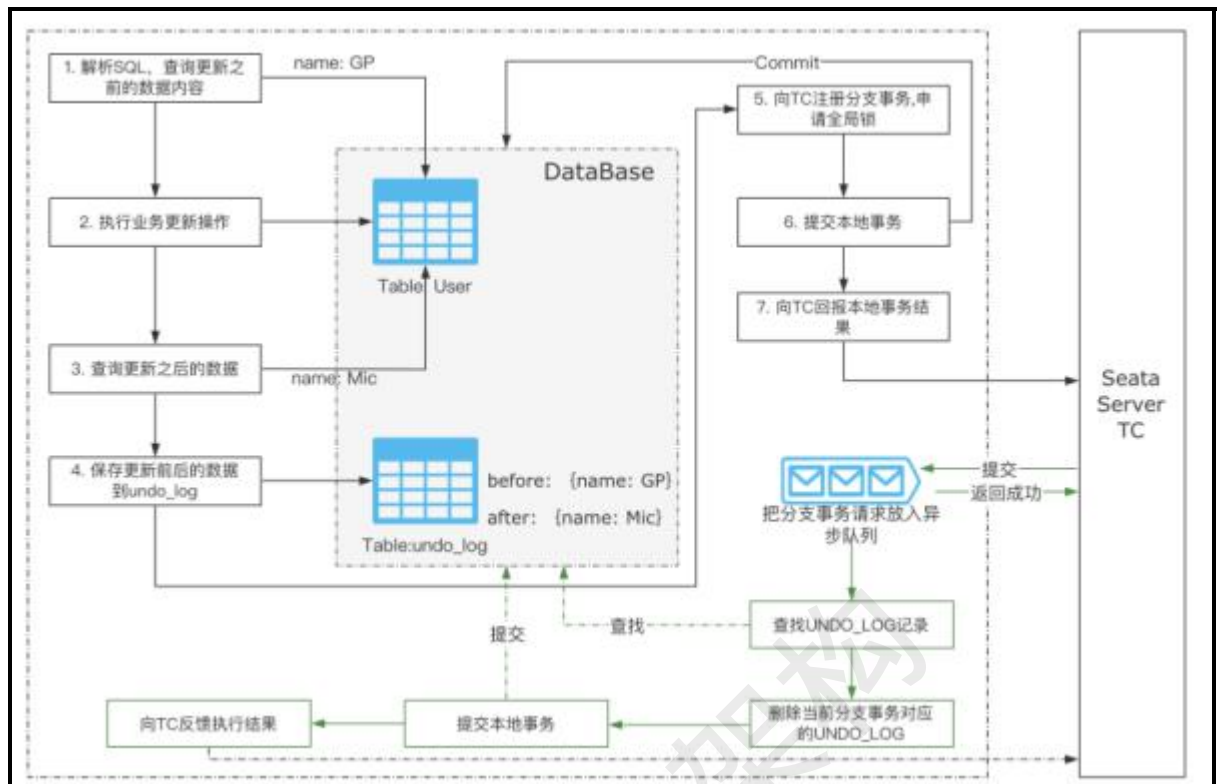
在分布式事务的实现上，对于强一致性，我们可以通过基于 **XA** 协议下的二阶段提交来实现，对于弱一致性，可以基于 **TCC** 事务模型、可靠性消息模型等方案来实现。

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。

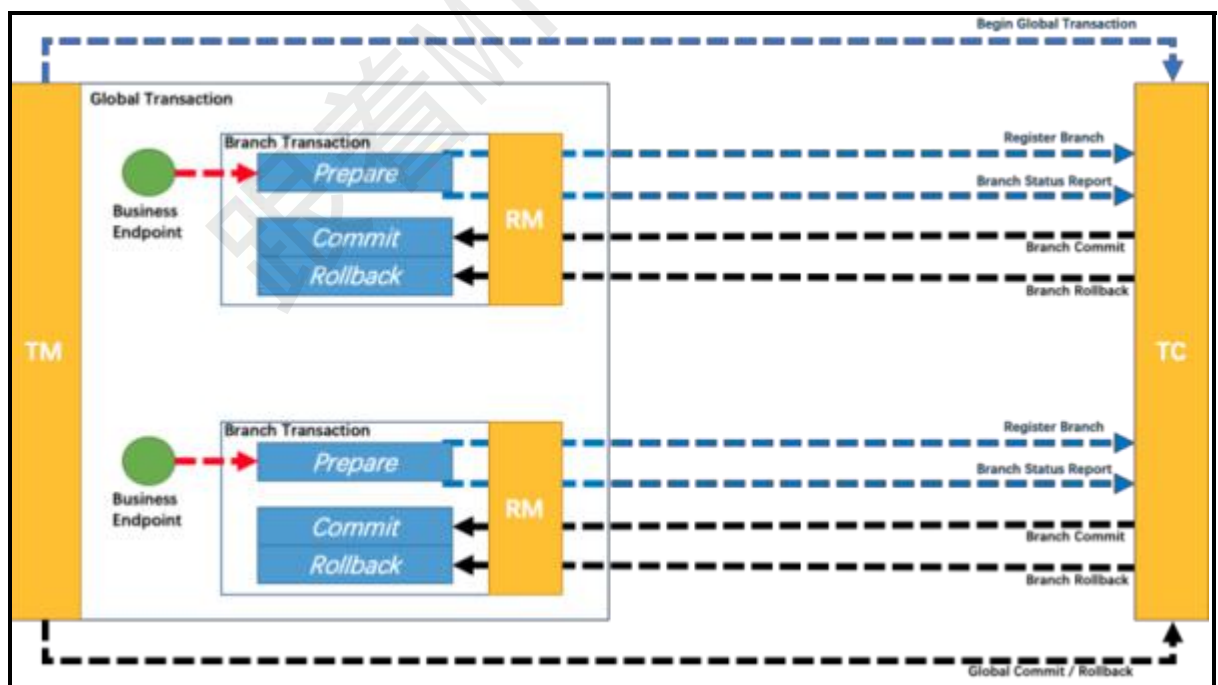
而 **Seata** 就是其中一种，它是阿里开源的分布式事务解决方案，提供了高性能且简单易用的分布式事务服务。

Seata 中封装了四种分布式事务模式，分别是：

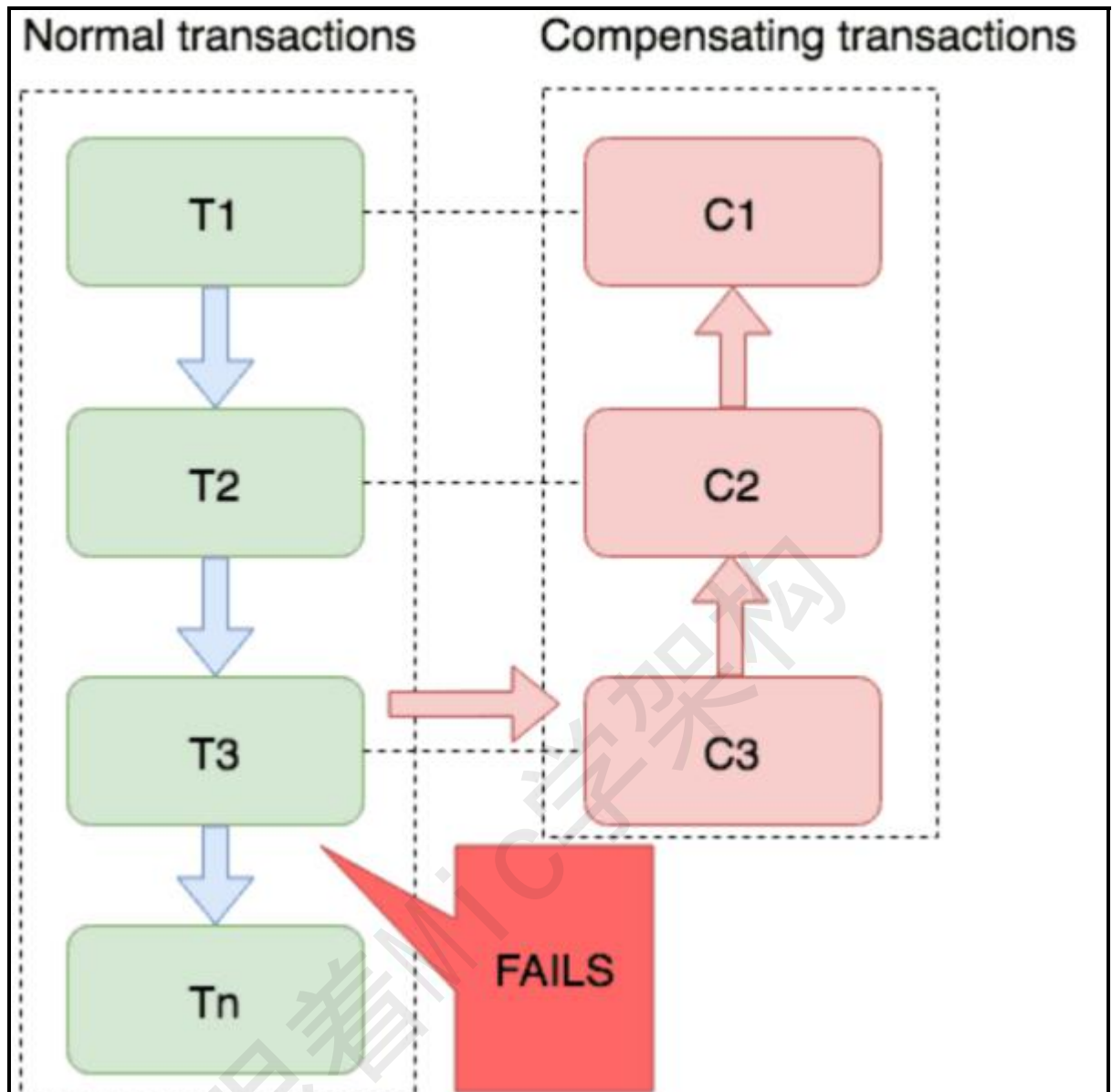
AT 模式，是一种基于本地事务+二阶段协议来实现的最终数据一致性方案，也是 **Seata** 默认的解决方案



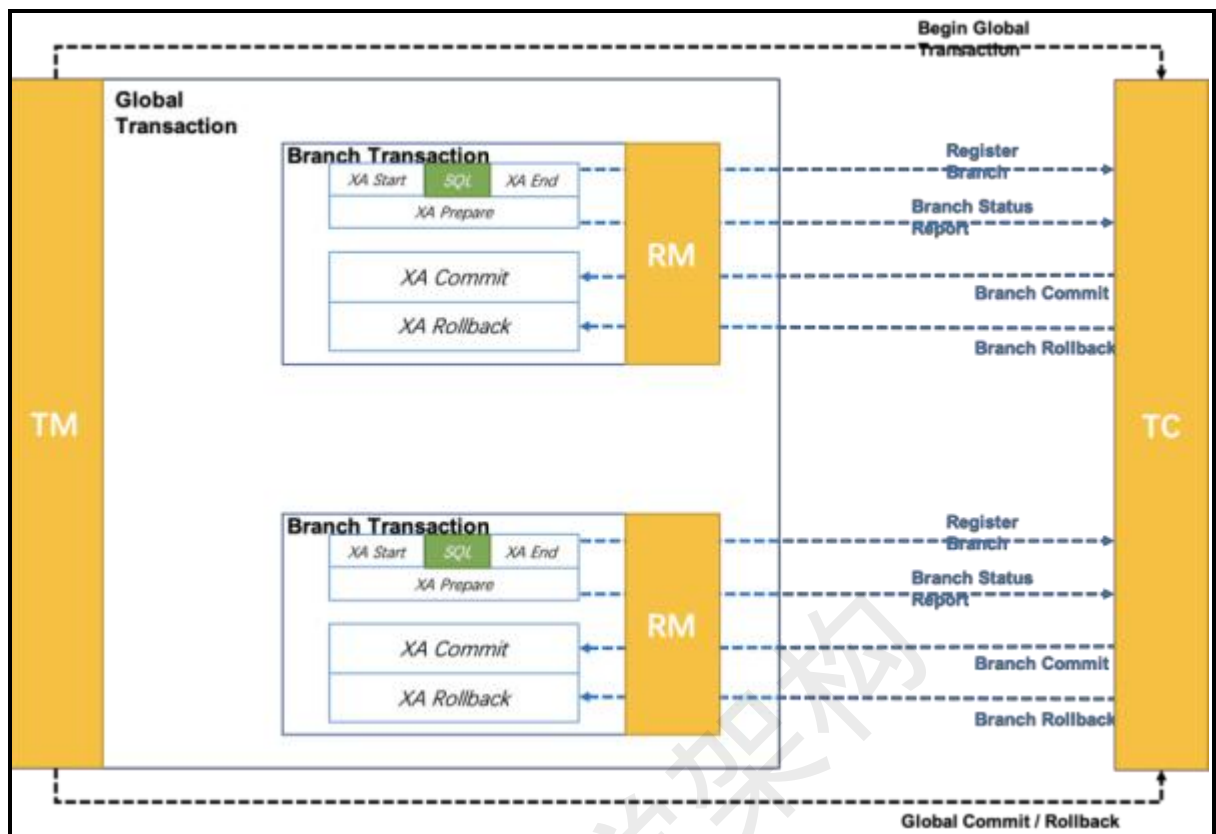
TCC 模式，TCC 事务是 Try、Confirm、Cancel 三个词语的缩写，简单理解就是把一个完整的业务逻辑拆分成三个阶段，然后通过事务管理器在业务逻辑层面根据每个分支事务的执行情况分别调用该业务的 Confirm 或者 Canceled 方法。



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，这就是约定优于配置思想的体现。

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。

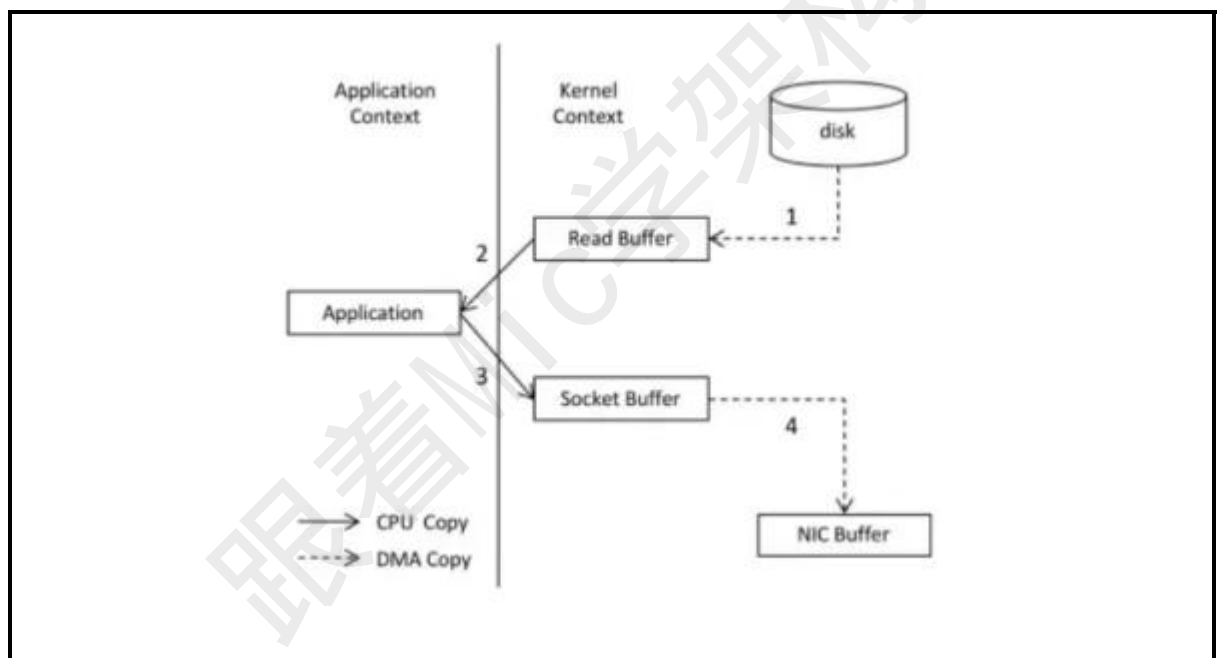
从磁盘中读取目标文件内容拷贝到内核缓冲区

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

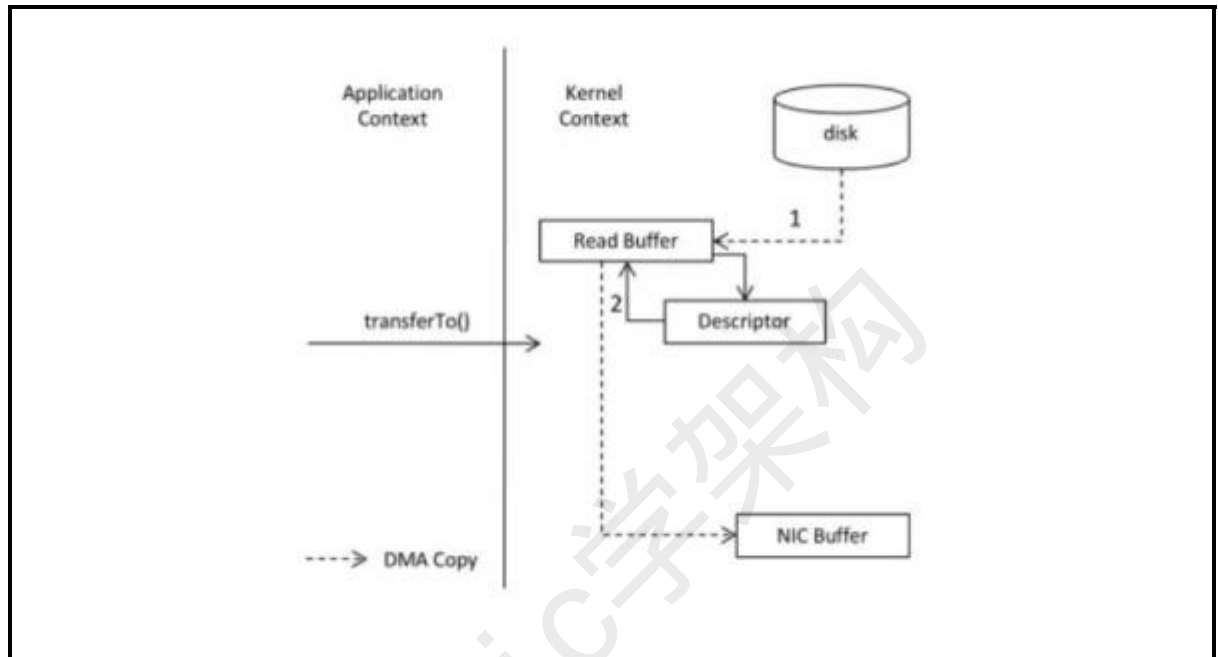
除此之外，由于用户空间和内核空间的切换会带来 CPU 的上下文切换，对于 CPU 性能也会造成性能影响。

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？

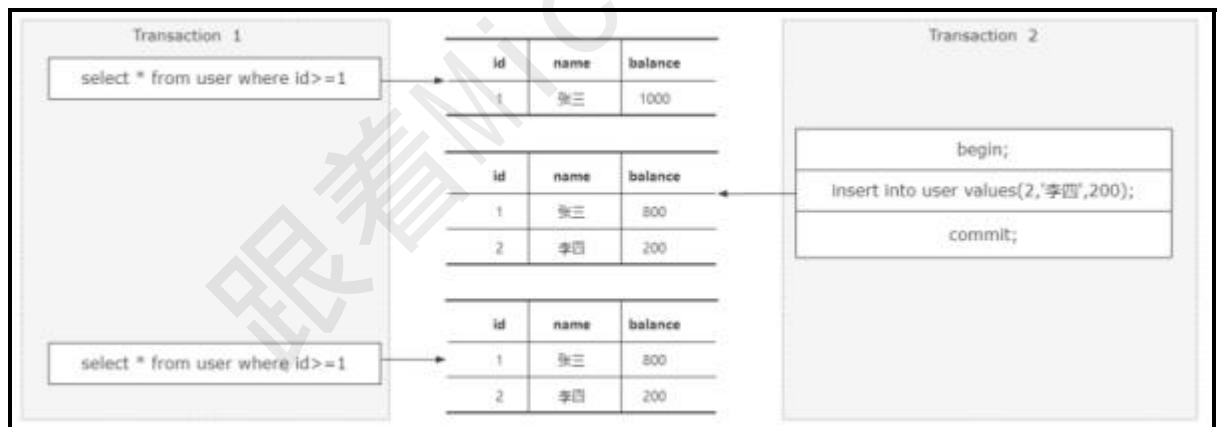
那么，什么是幻读呢？

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。

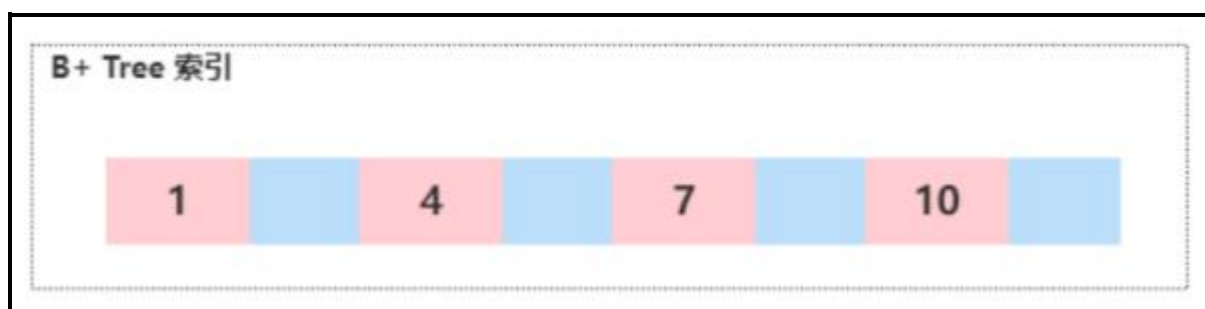


所以，幻读会带来数据一致性问题。

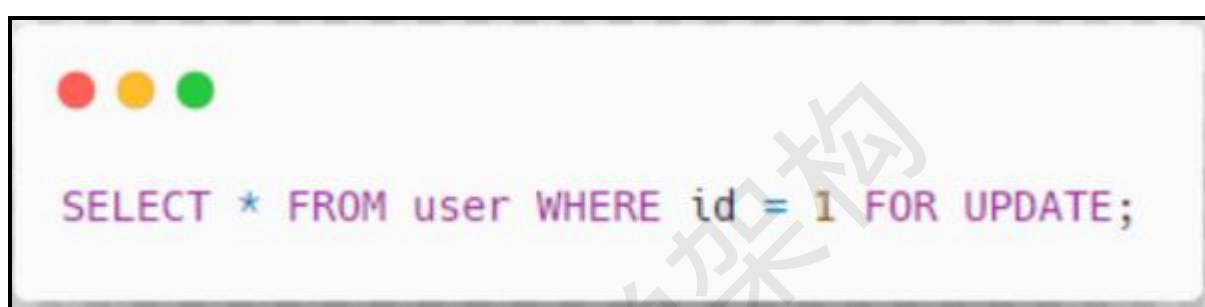
3、InnoDB 如何解决幻读的问题

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：

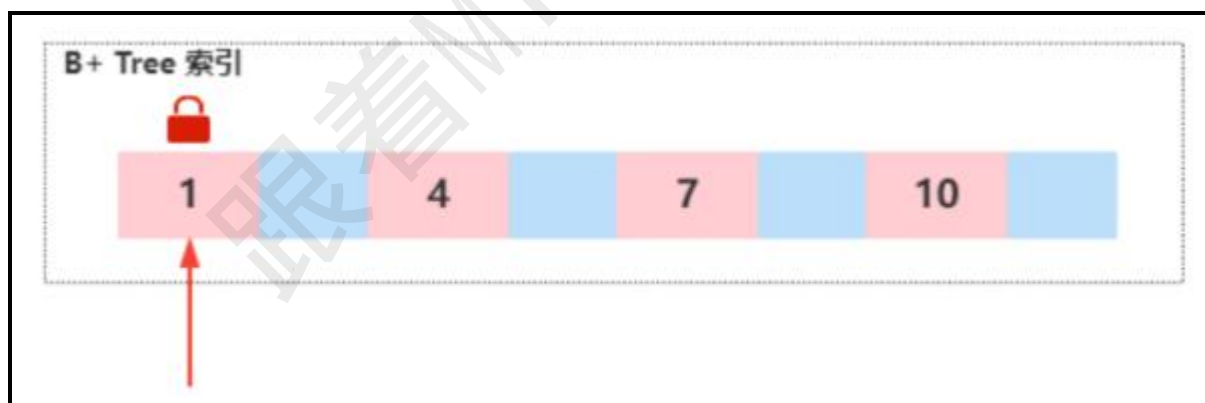
假设现在存在这样（图片）这样一个 B+ Tree 的索引结构，这个结构中有四个索引元素分别是：1、4、7、10。



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。



```
SELECT * FROM user WHERE id >4 and id<7 FOR UPDATE;
```

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。

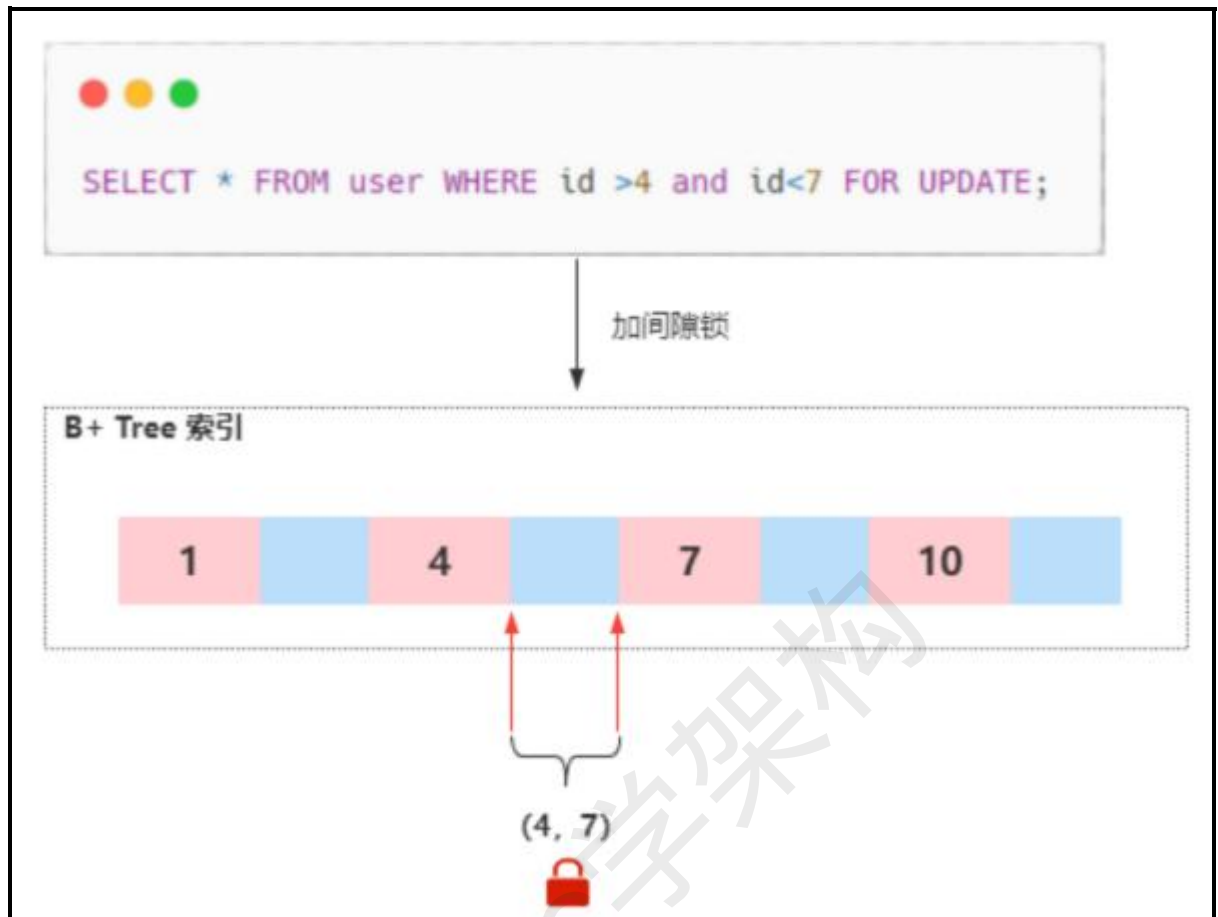


```
INSERT INTO user(id,name) VALUES(5,'mimi');
```

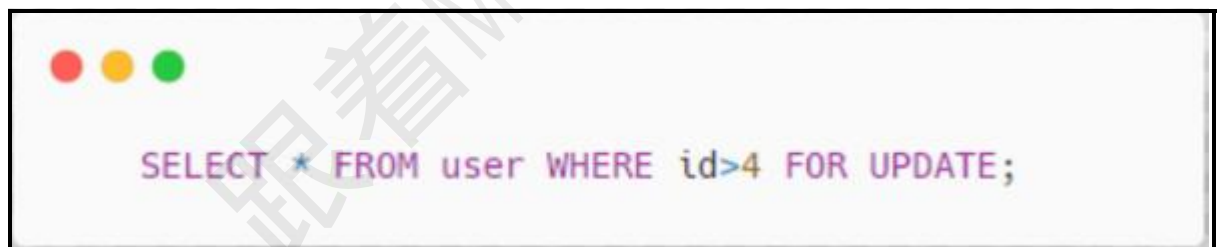
所以，在 InnoDB 中设计了一种间隙锁，它的主要功能是锁定一段范围内的索引记录

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）



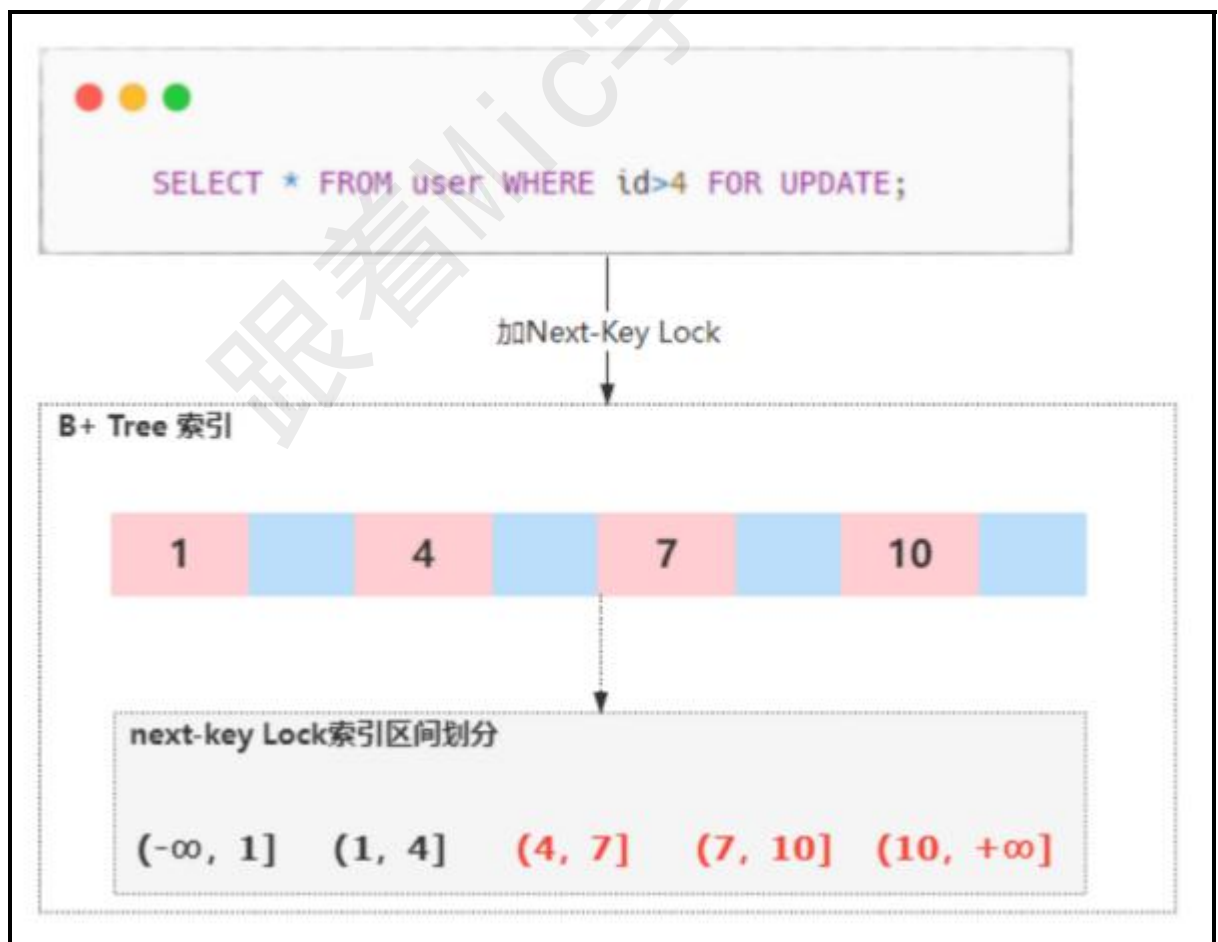
这条查询语句是针对 $id > 4$ 这个条件加锁，那么它需要锁定多个索引区间，所以在这种情况下 InnoDB 引入了 next-key Lock 机制。

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 lock=new ReentrantLock();

public void sync(){
    lock.lock();    //竞争锁
    //TODO 线程安全的代码
    lock.unlock(); //释放锁
}
```

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功能和常见的功能，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！

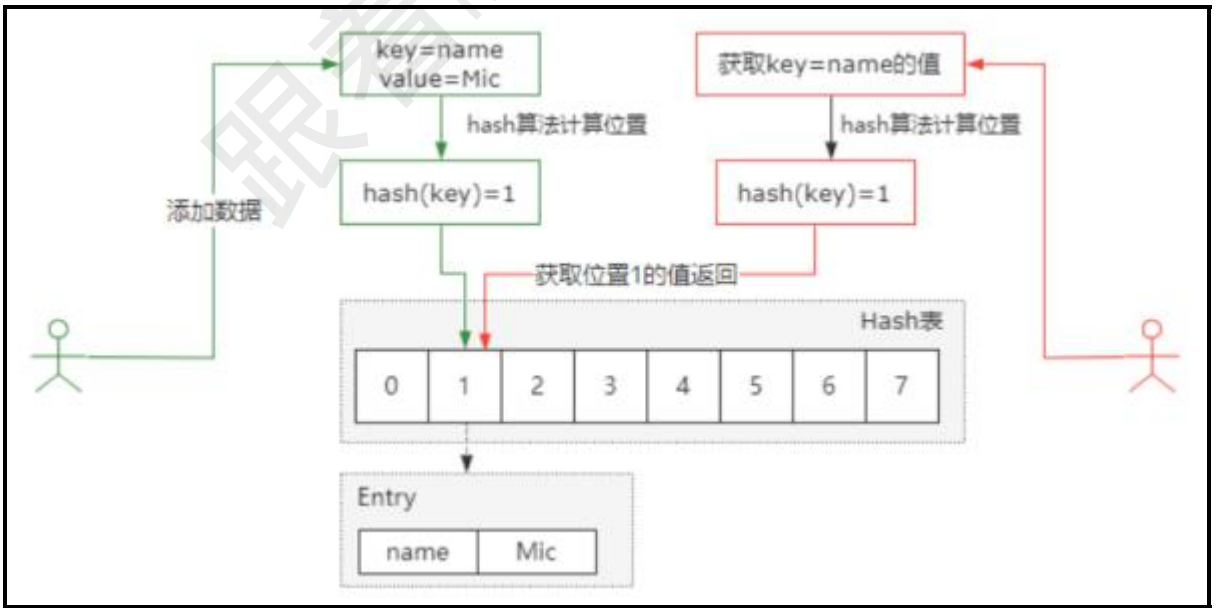
高手

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。

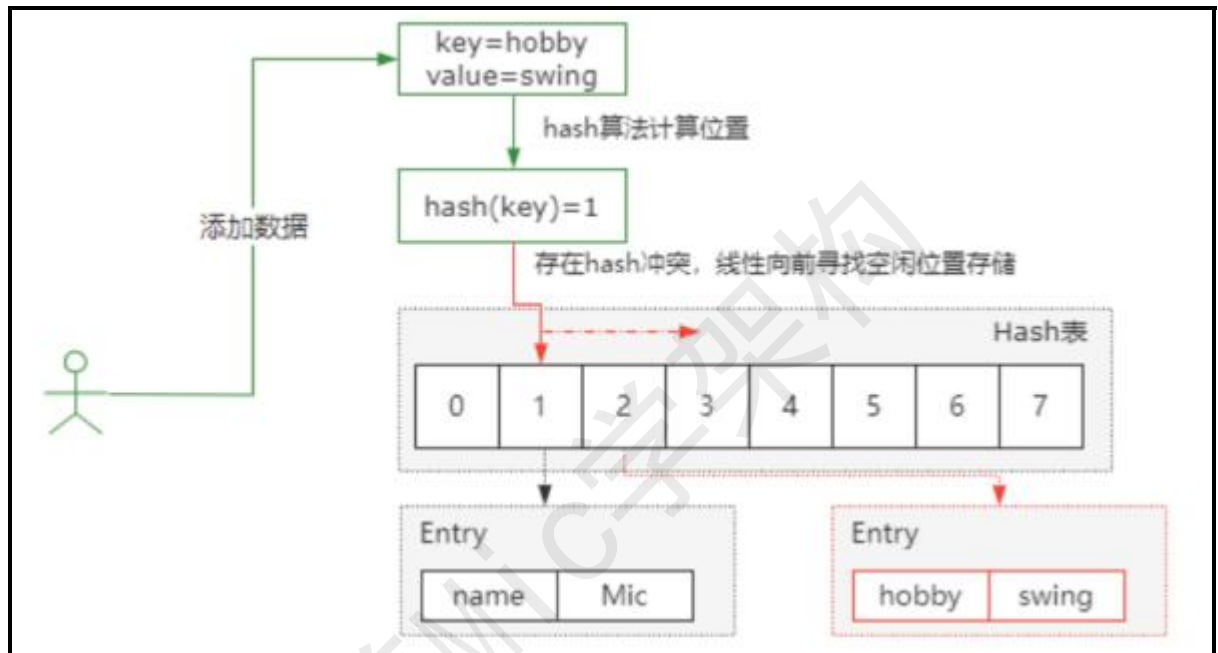


所谓 hash 冲突，是由于哈希算法被计算的数据是无限的，而计算后的结果范围有限，所以总会存在不同的数据经过计算后得到的值相同，这就是哈希冲突。

通常解决 hash 冲突的方法有 4 种。

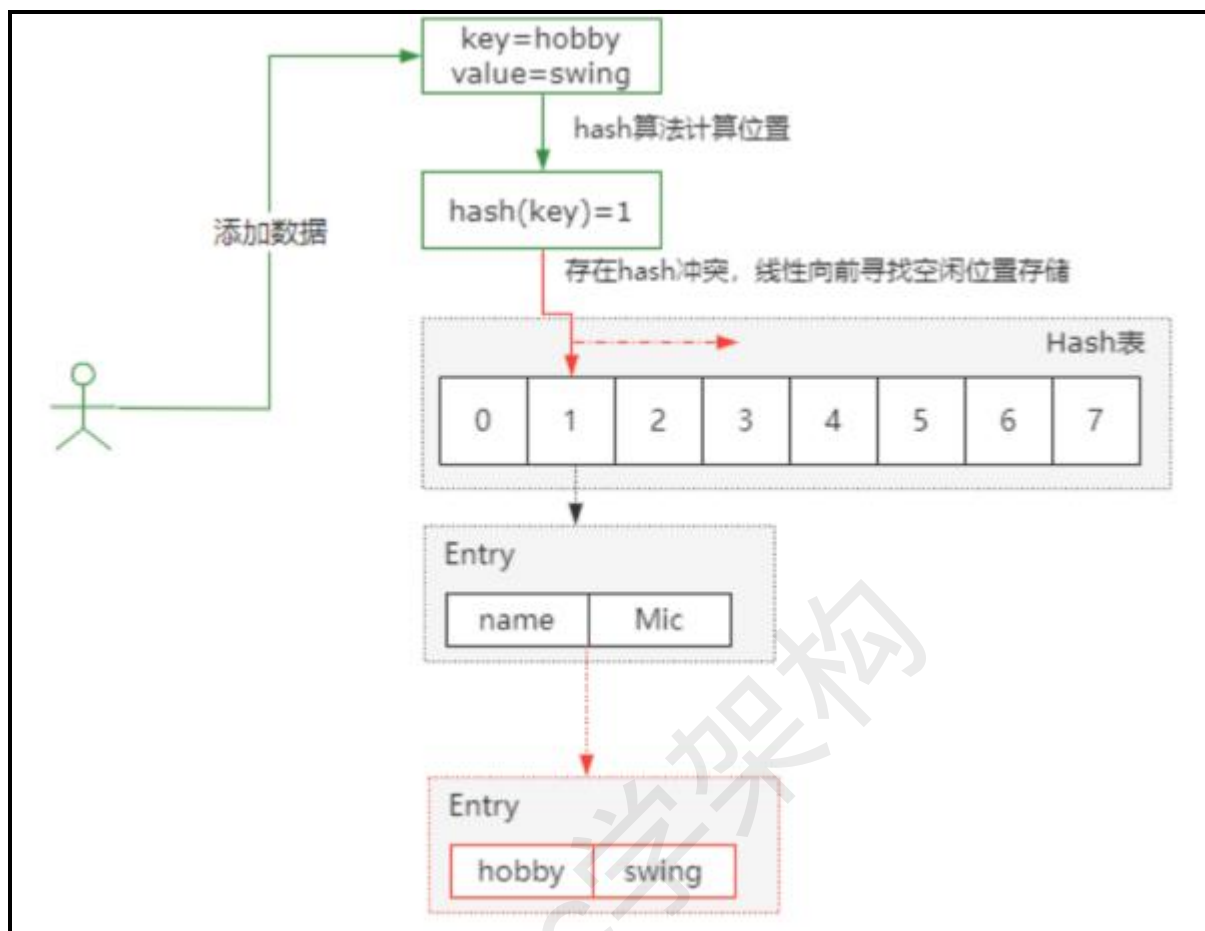
开放定址法，也称为线性探测法，就是从发生冲突的那个位置开始，按照一定的次序从 hash 表中找到一个空闲的位置，然后把发生冲突的元素存入到这个空闲位置中。ThreadLocal 就用到了线性探测法来解决 hash 冲突的。

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

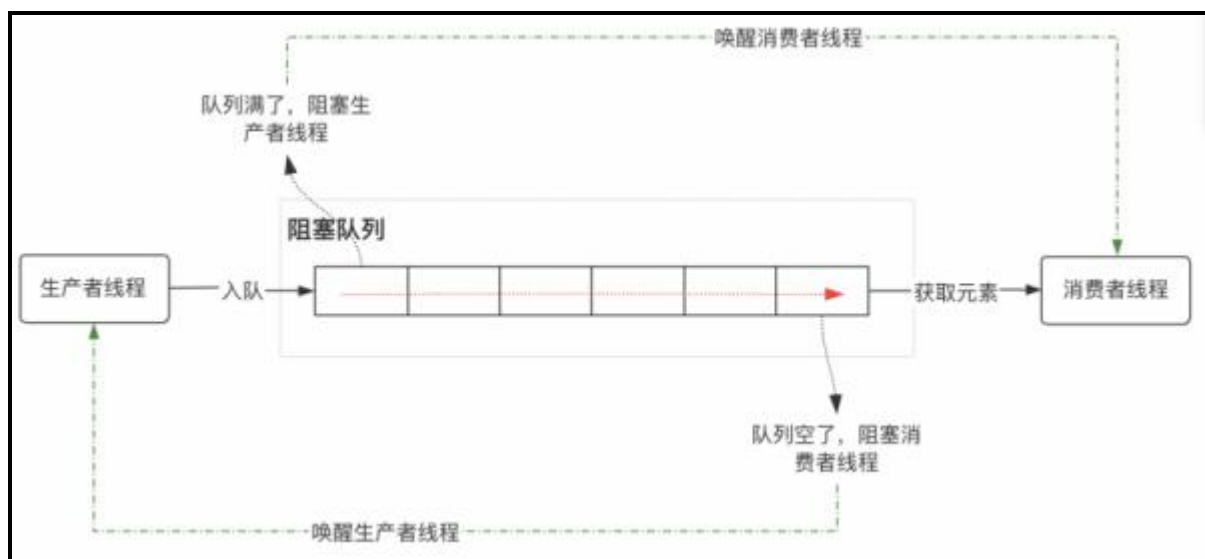
有界队列就是说队列中的元素个数是有限制的，而无界对接表示队列中的元素个数没有限制！ 嗯！！！

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。

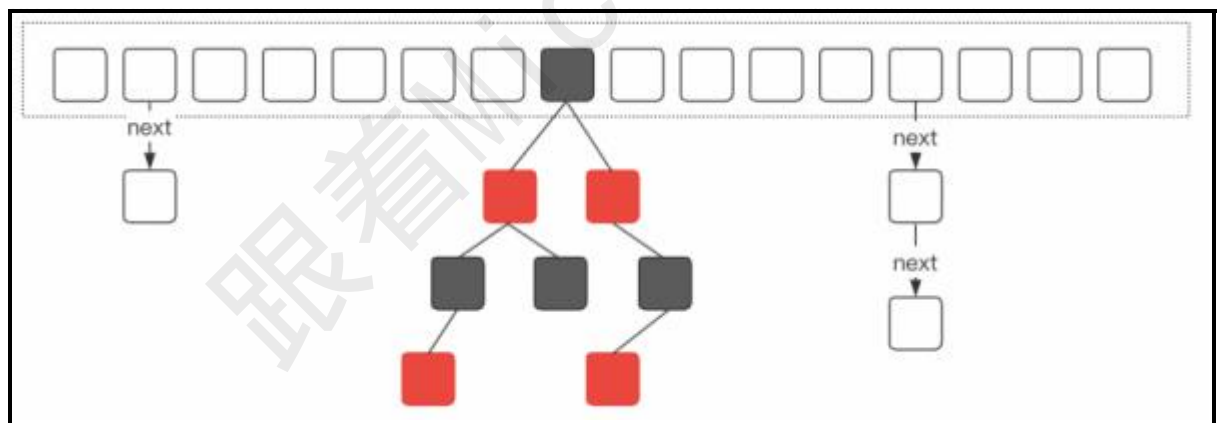
当我们初始化一个 ConcurrentHashMap 实例时，默认会初始化一个长度为 16 的数组。由于 ConcurrentHashMap 它的核心仍然是 hash 表，所以必然会存在 hash 冲突问题。

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。

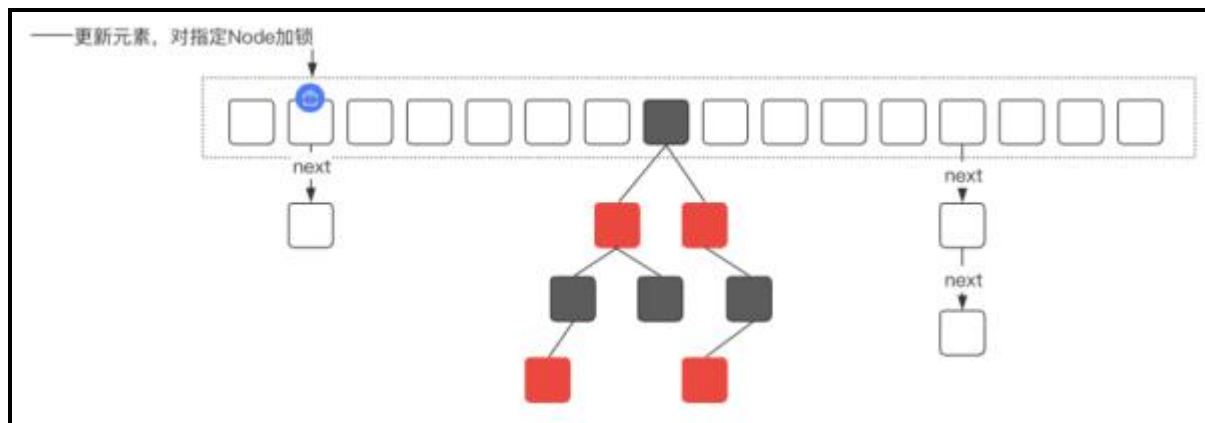
另外，随着 ConcurrentHashMap 的动态扩容，一旦链表长度小于 8，红黑树会退化成单向链表。



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

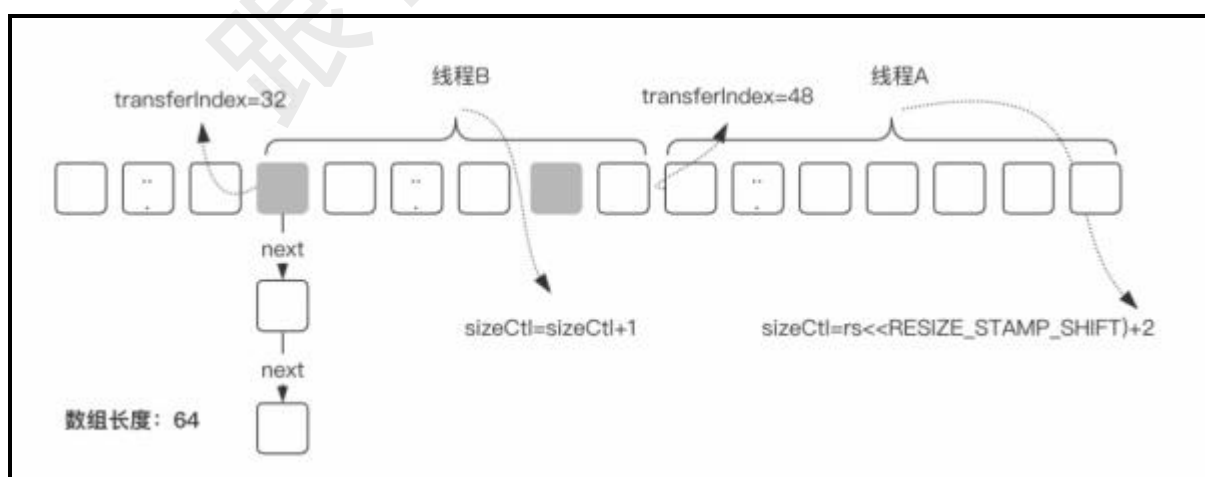
如果在并发性能和数据安全性之间做好平衡，在很多地方都有类似的设计，比如cpu的三级缓存、mysql的buffer_pool、Synchronized的锁升级等等。

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。

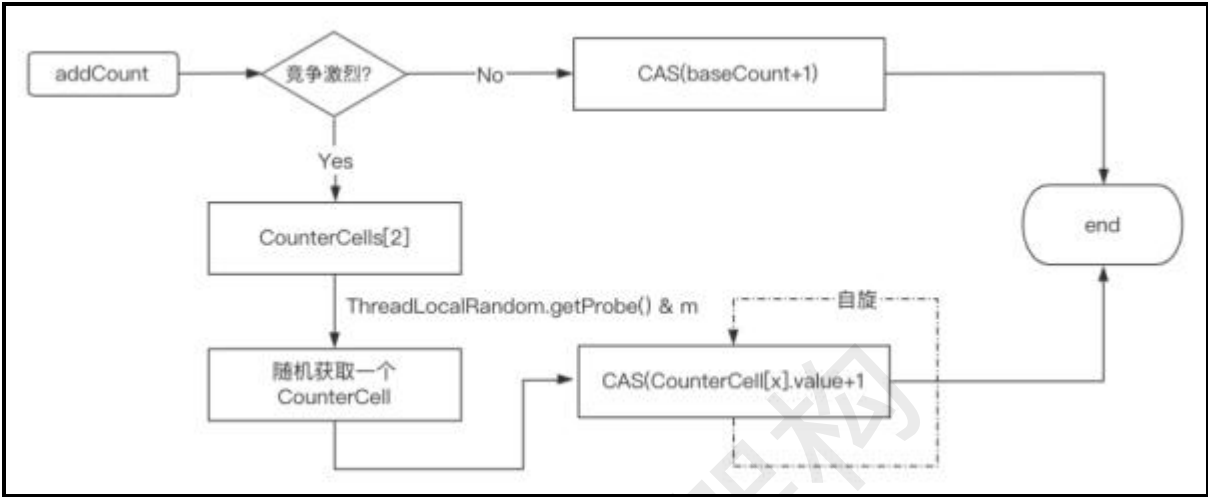


ConcurrentHashMap 中有一个size()方法来获取总的元素个数，而在多线程并发场景中，在保证原子性的前提下来实现元素个数的累加，性能是非常低的。

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：

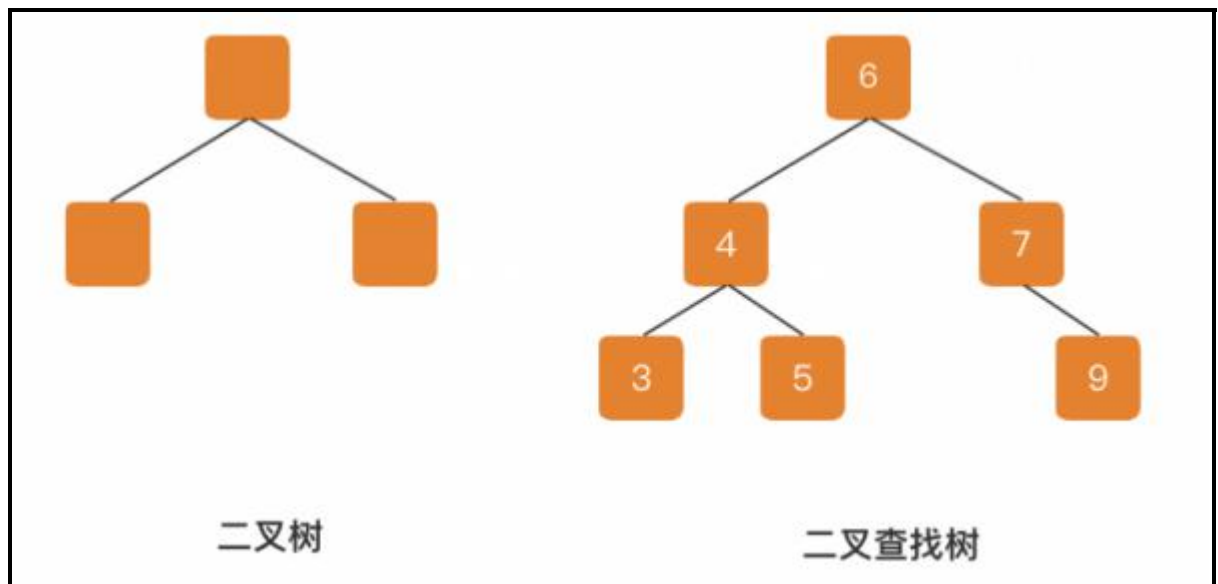
了解二叉树、AVL 树、B 树的概念

B 树和 B+树的应用场景

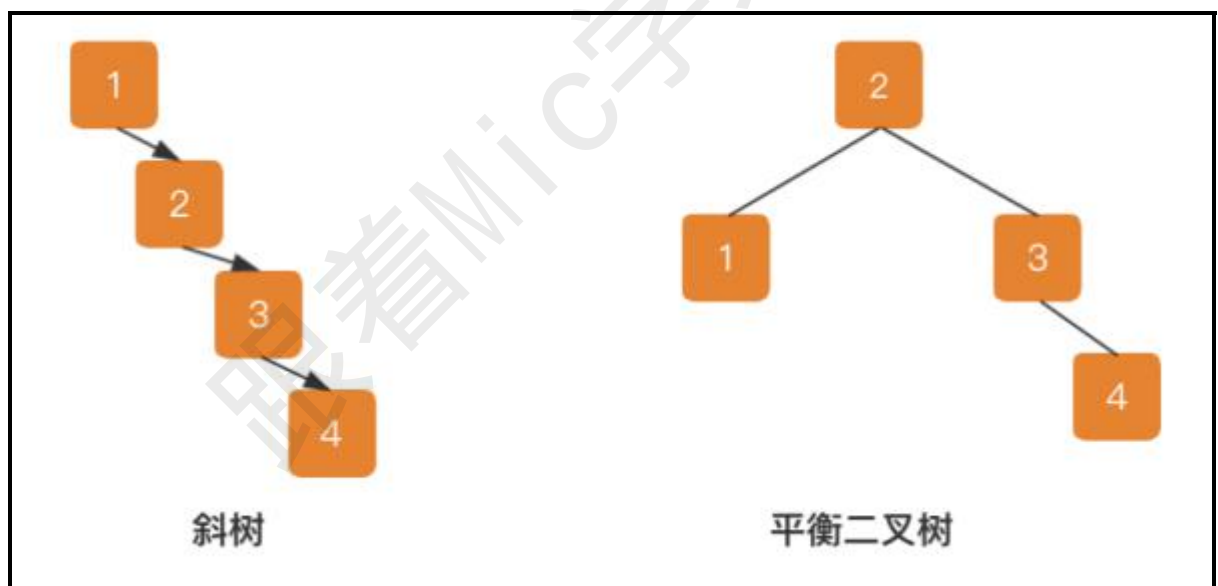
B 树是一种多路平衡查找树，为了更形象的理解，（我们来看这张图）。

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。

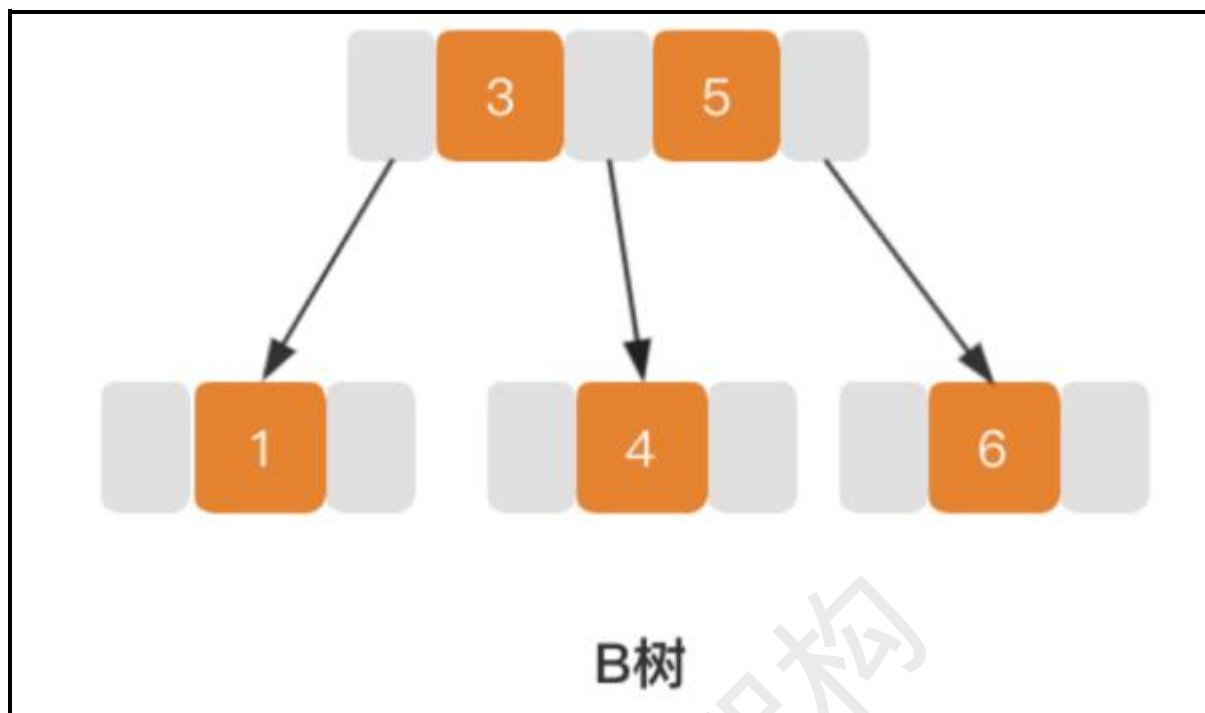


，二叉查找树会出现斜树问题，导致时间复杂度增加，因此又引入了一种平衡二叉树，它具有二叉查找树的所有特点，同时增加了一个规则：“它的左右两个子树的高度差的绝对值不超过 1”。平衡二叉树会采用左旋、右旋的方式来实现平衡。



，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。

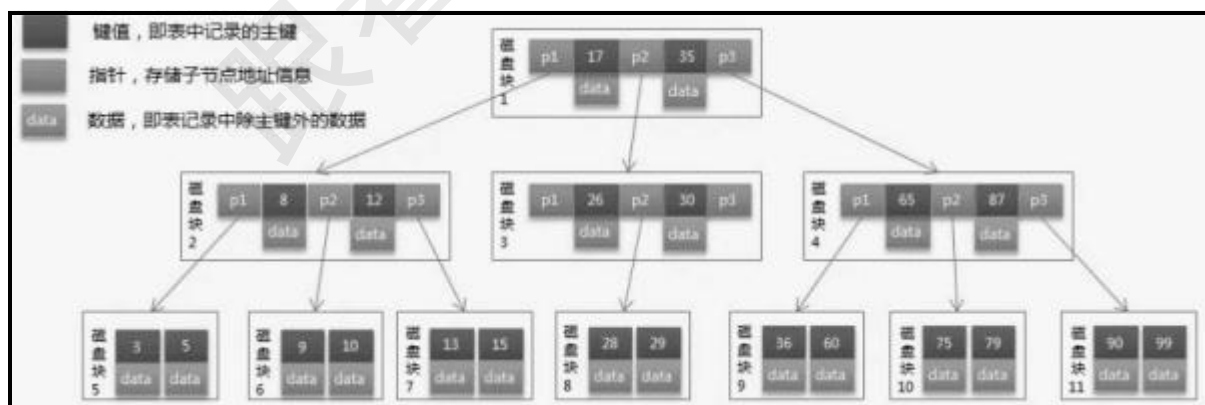


B+树，其实是在 B 树的基础上做的增强，最大的区别有两个：

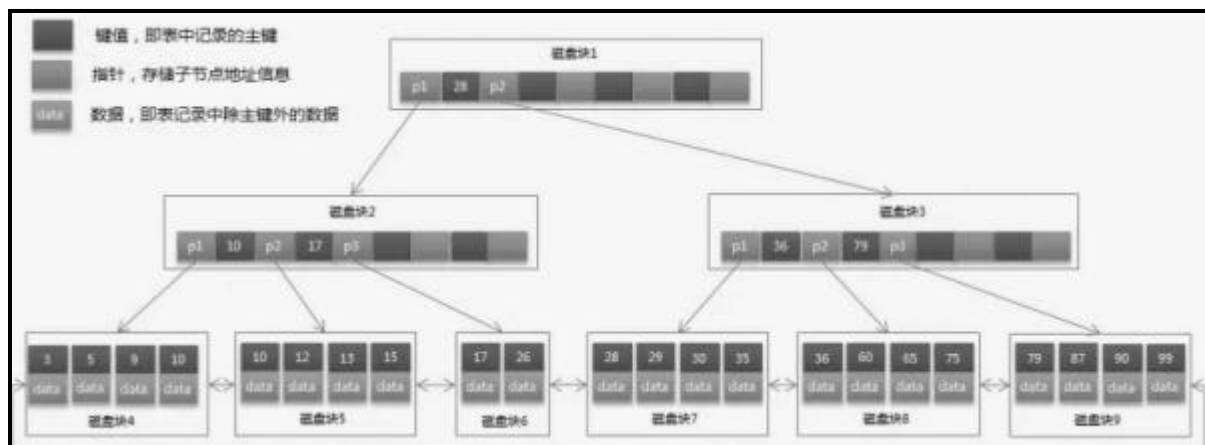
B 树的数据存储在每个节点上，而 B+树中的数据是存储在叶子节点，并且通过链表的方式把叶子节点中的数据进行连接。

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。



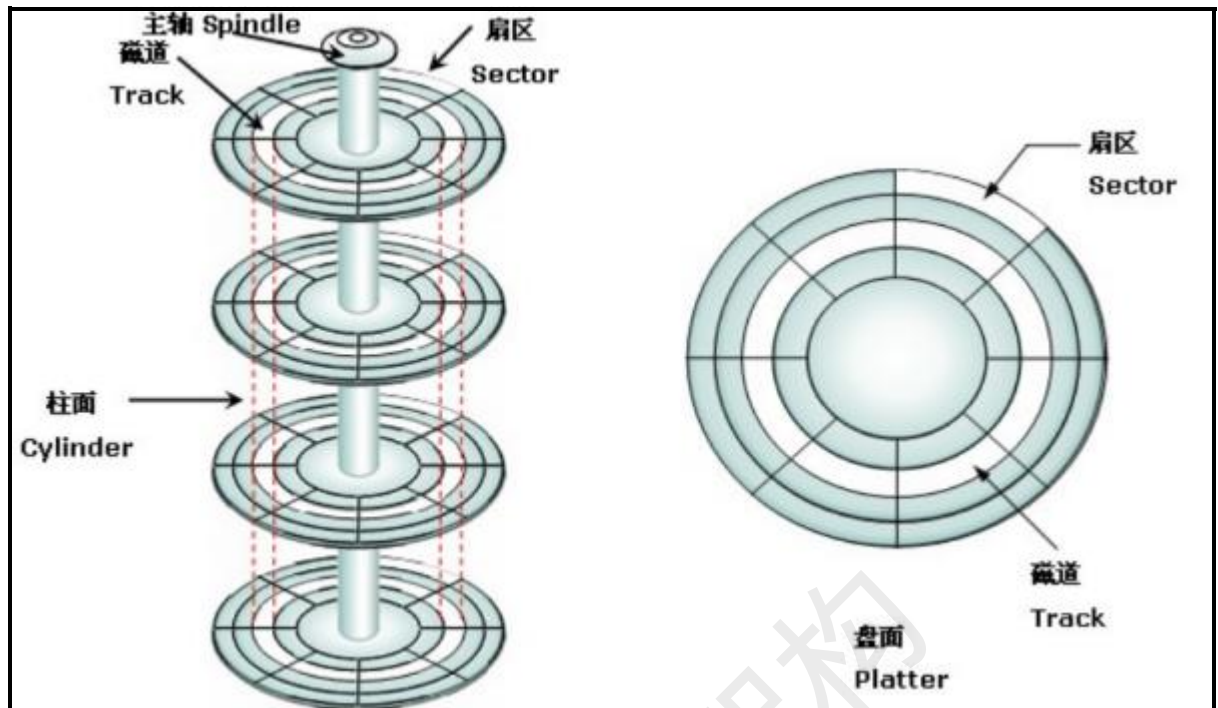
B 树和 B+树，一般都是应用在文件系统和数据库系统中，用来减少磁盘 IO 带来的性能损耗。

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。

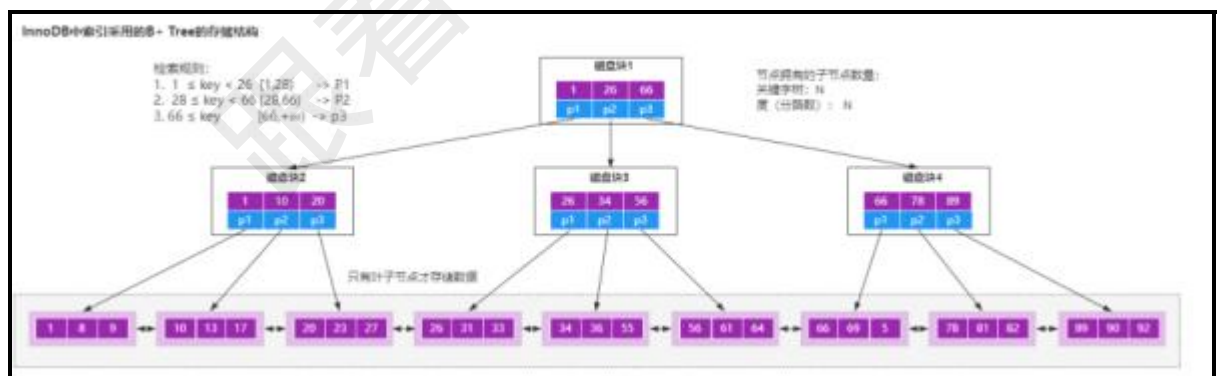
为了读取这个扇区的数据，需要把磁头放在这个扇区的上面，为了实现这一个点，磁盘会不断旋转，把目标扇区旋转到磁头下面，使得磁头找到对应的磁道，这里涉及到寻道事件以及旋转时间。



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。



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。

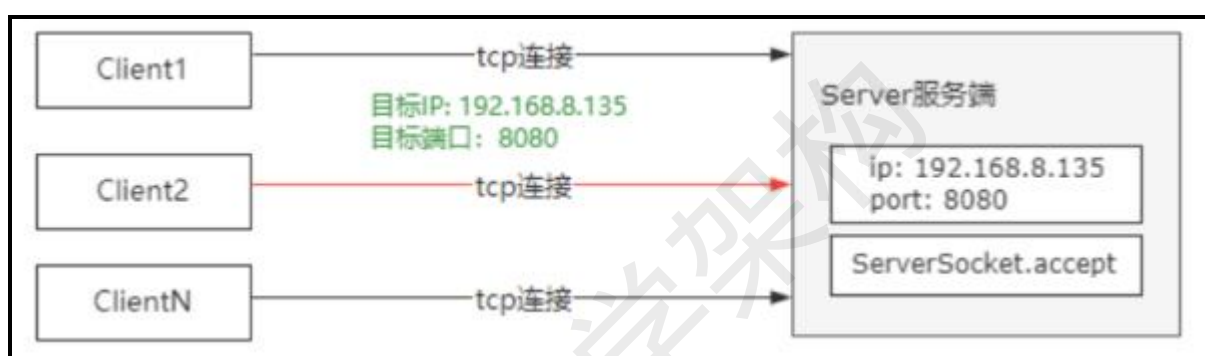
正常情况下，我们对于网络通信的认识可能是这样。

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(如图)，**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。

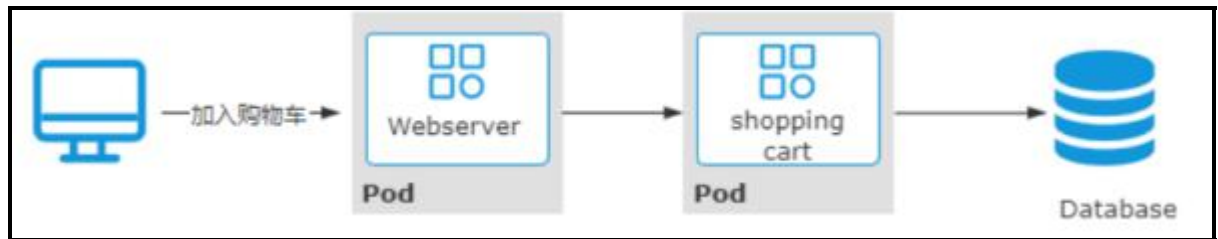
首先，当我们把一个电商系统以微服务化架构进行拆分后，会的到这样的一个架构，其中包括 **Webserver**、**payment**、**inventory** 等等。



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。

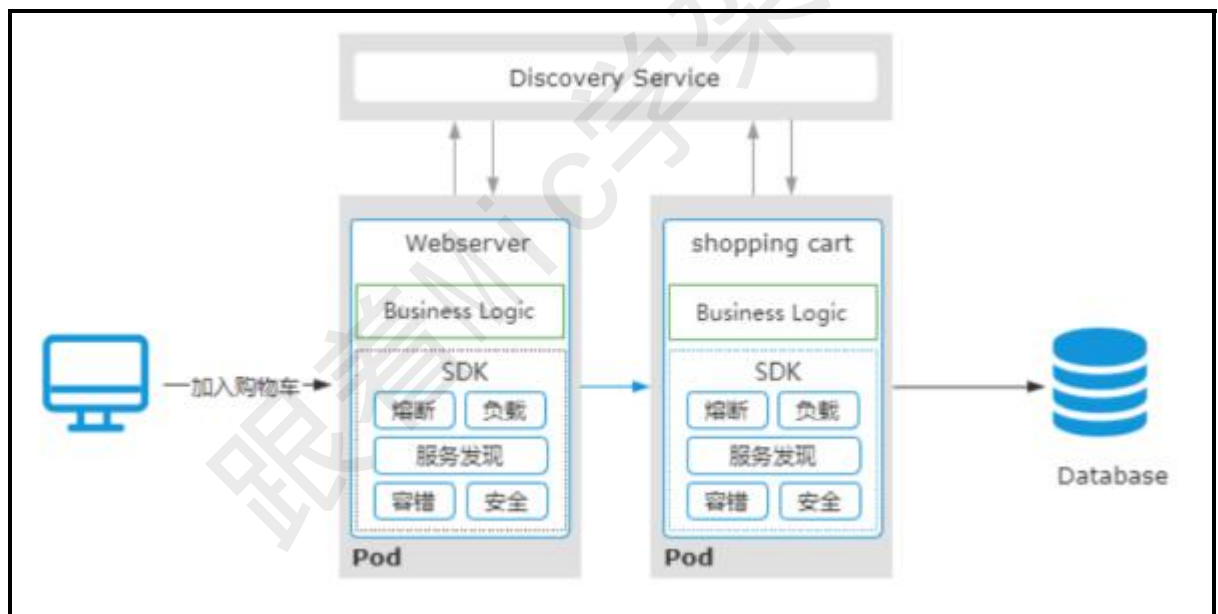


所以，服务之间必须要相互通信，才能实现功能的完整性。比如用户把一个商品加入购物车，请求会进入到 **Webserver**，然后转发到 **shopping cart** 进行处理，并存到数据库。



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**。



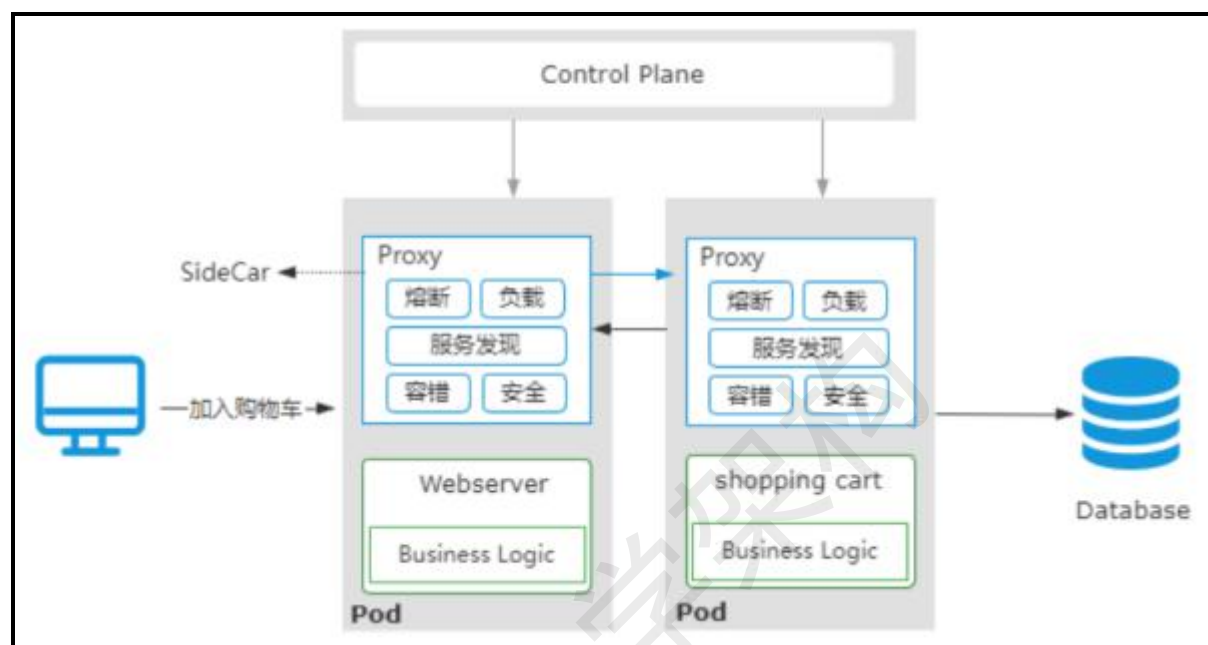
在第二代微服务架构中，负责业务开发的小伙伴不仅仅需要关注业务逻辑，还需要花大量精力去处理微服务中的一些基础性配置工作，虽然 **Spring Cloud** 已经尽可能去完成了这些事情，但对于开发人员来说，学习 **Spring Cloud**，以及针对 **Spring Cloud** 的配置和维护，仍然存在较大的挑战。另外呢，也增加了整个微服务的复杂性。

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。

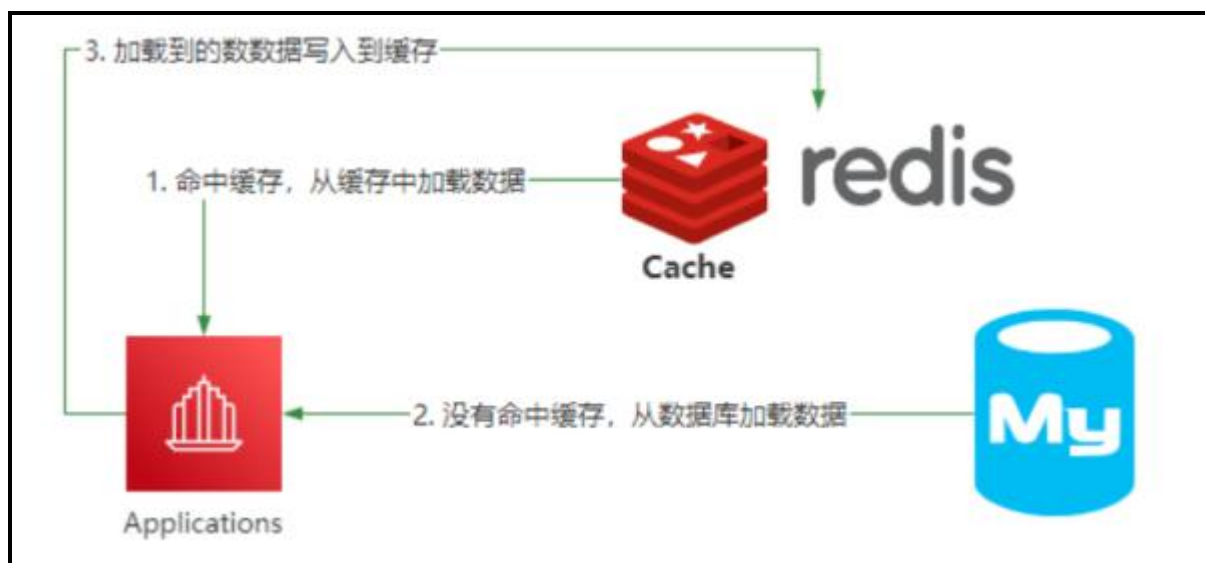
嗯，就是，Mysql 的数据发生变化以后，可以同步修改 Redis 里面的数据。

高手

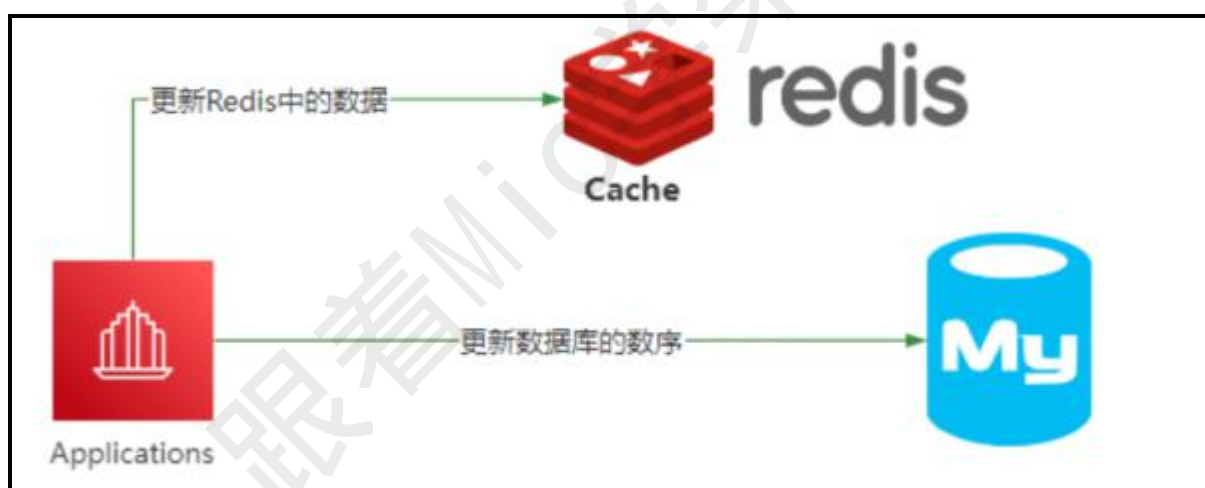
一般情况下，Redis 用来实现应用和数据库之间读操作的缓存层，主要目的是减少数据库 IO，还可以提升数据的 IO 性能。

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。

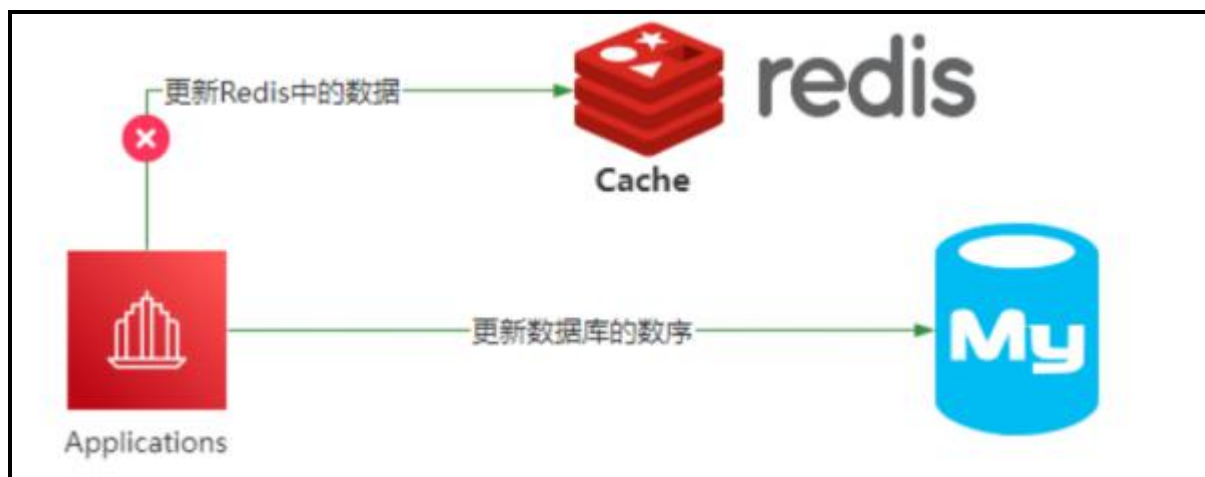


在这种情况下，能够选择的方法只有几种。

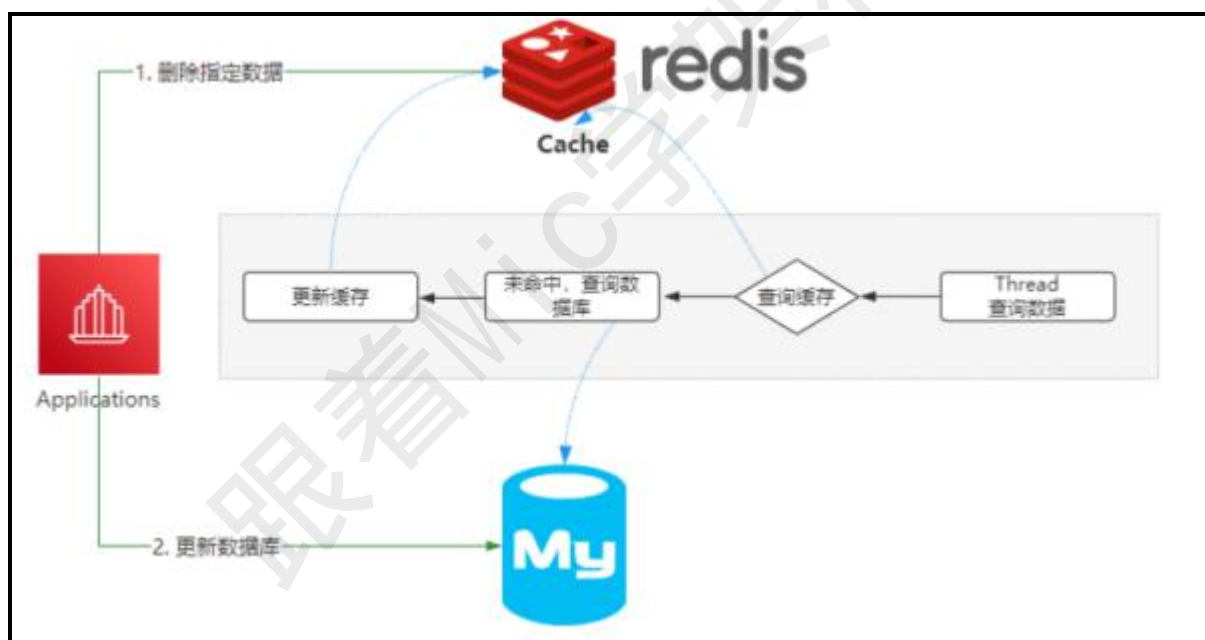
先更新数据库，再更新缓存

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。

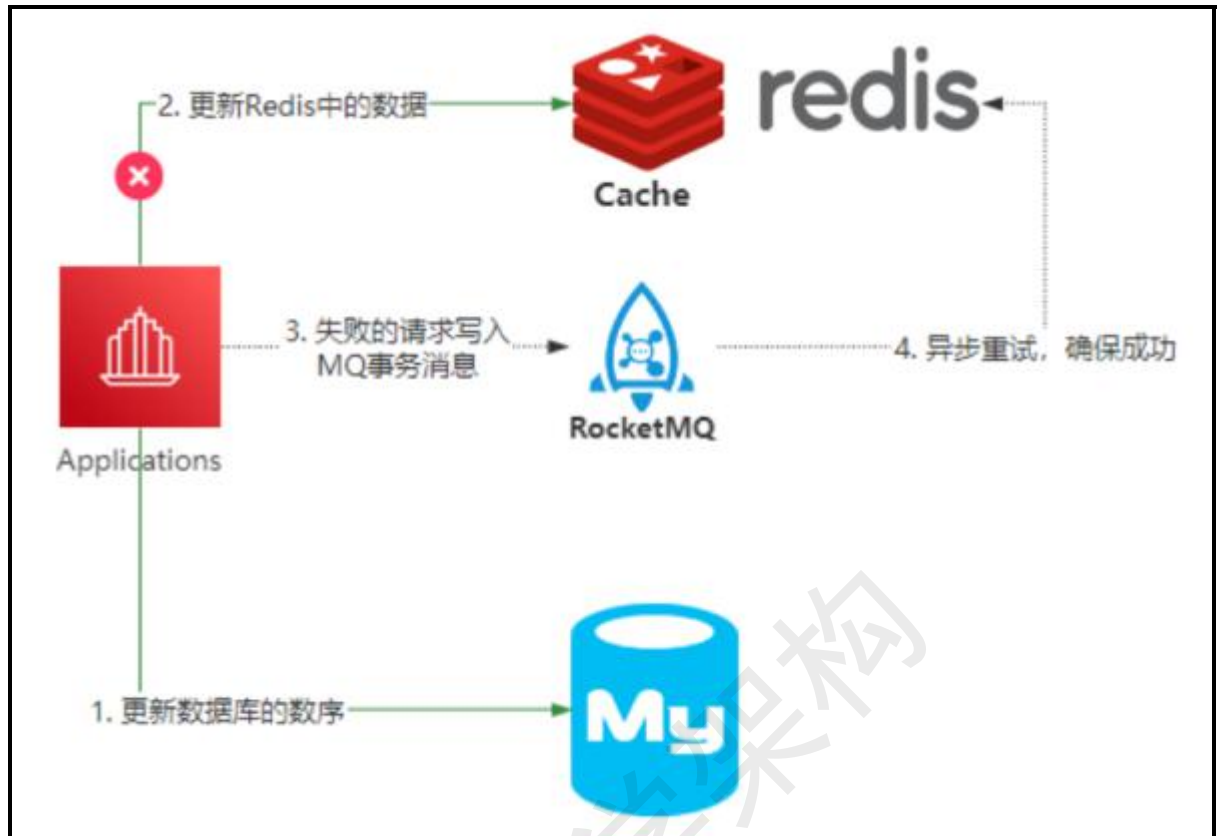


如果是先删除缓存，再更新数据库，理想情况是应用下次访问 Redis 的时候，发现 Redis 里面的数据是空的，就从数据库加载保存到 Redis 里面，那么数据是一致的。但是在极端情况下，由于删除 Redis 和更新数据库这两个操作并不是原子的，所以这个过程如果有其他线程来访问，还是会存在数据不一致问题。

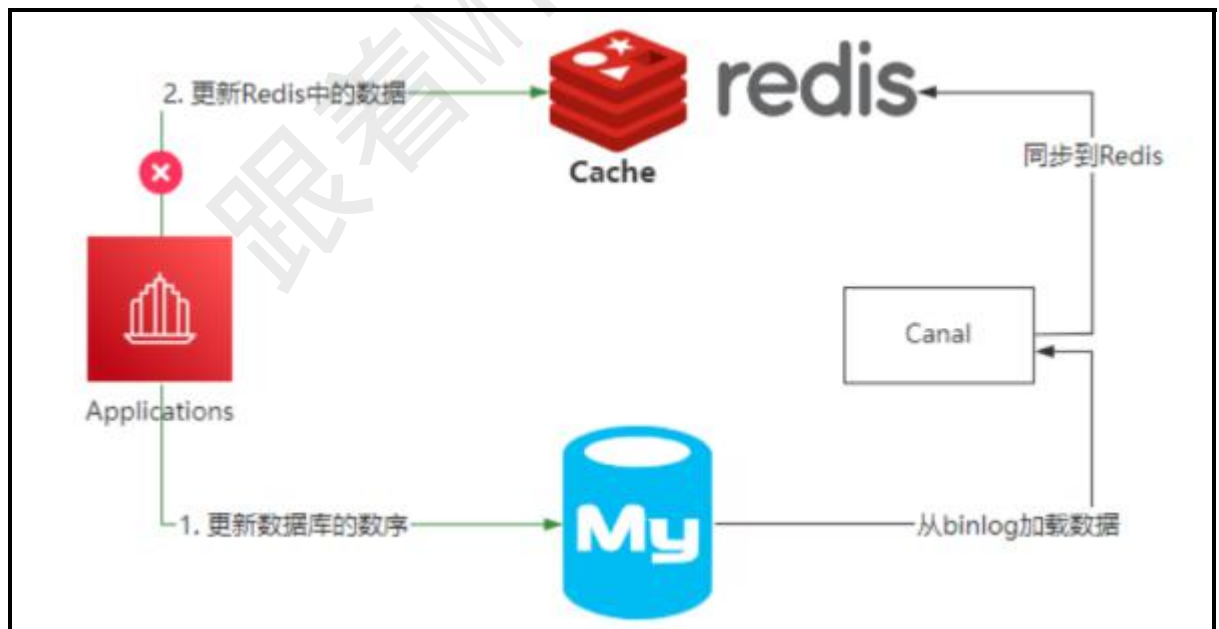


所以，如果需要在极端情况下仍然保证 Redis 和 Mysql 的数据一致性，就只能采用最终一致性方案。

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。

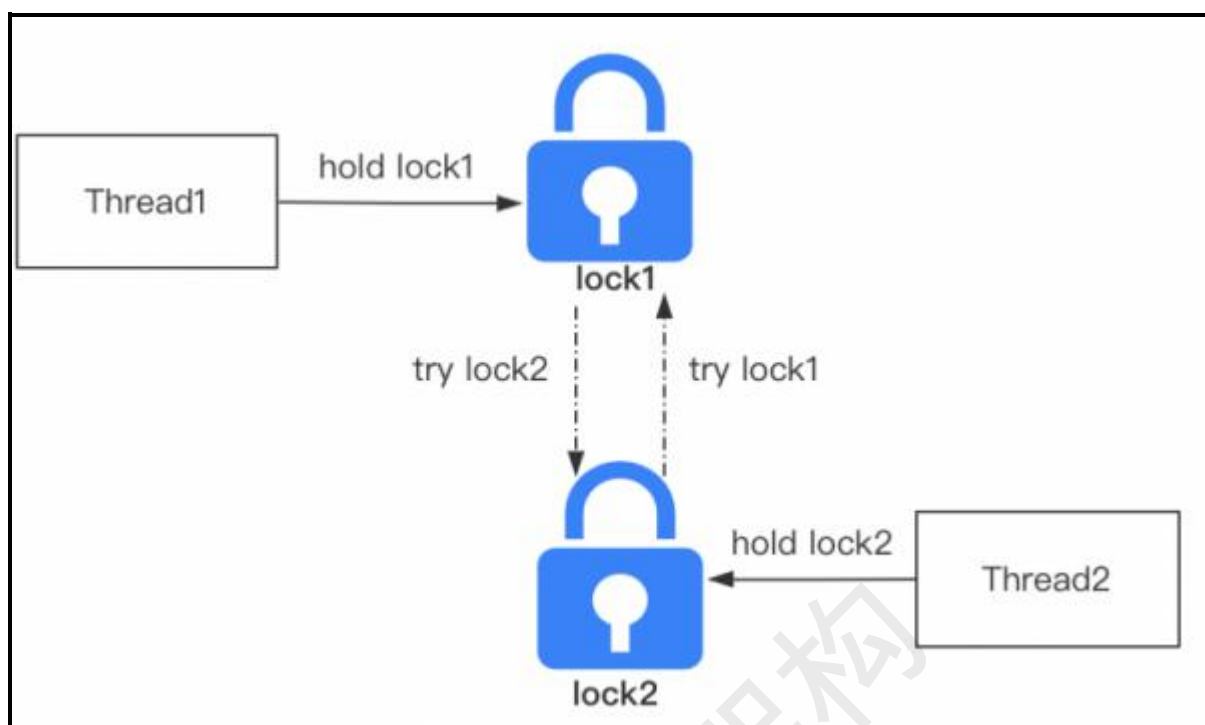
普通人

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

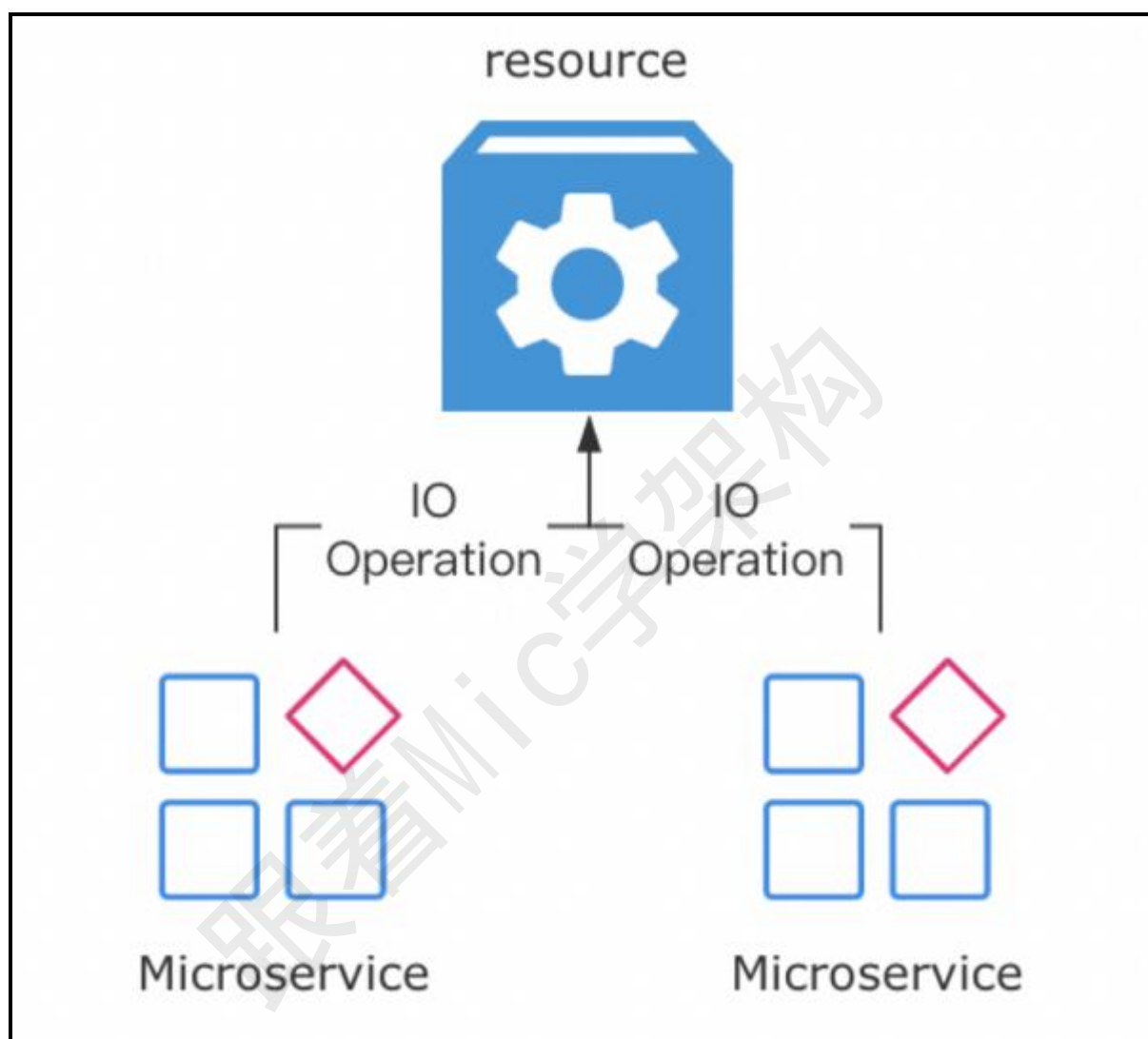
嗯，分布式锁，就是可以用来实现锁的分布性，嗯...

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，同时再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时候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。

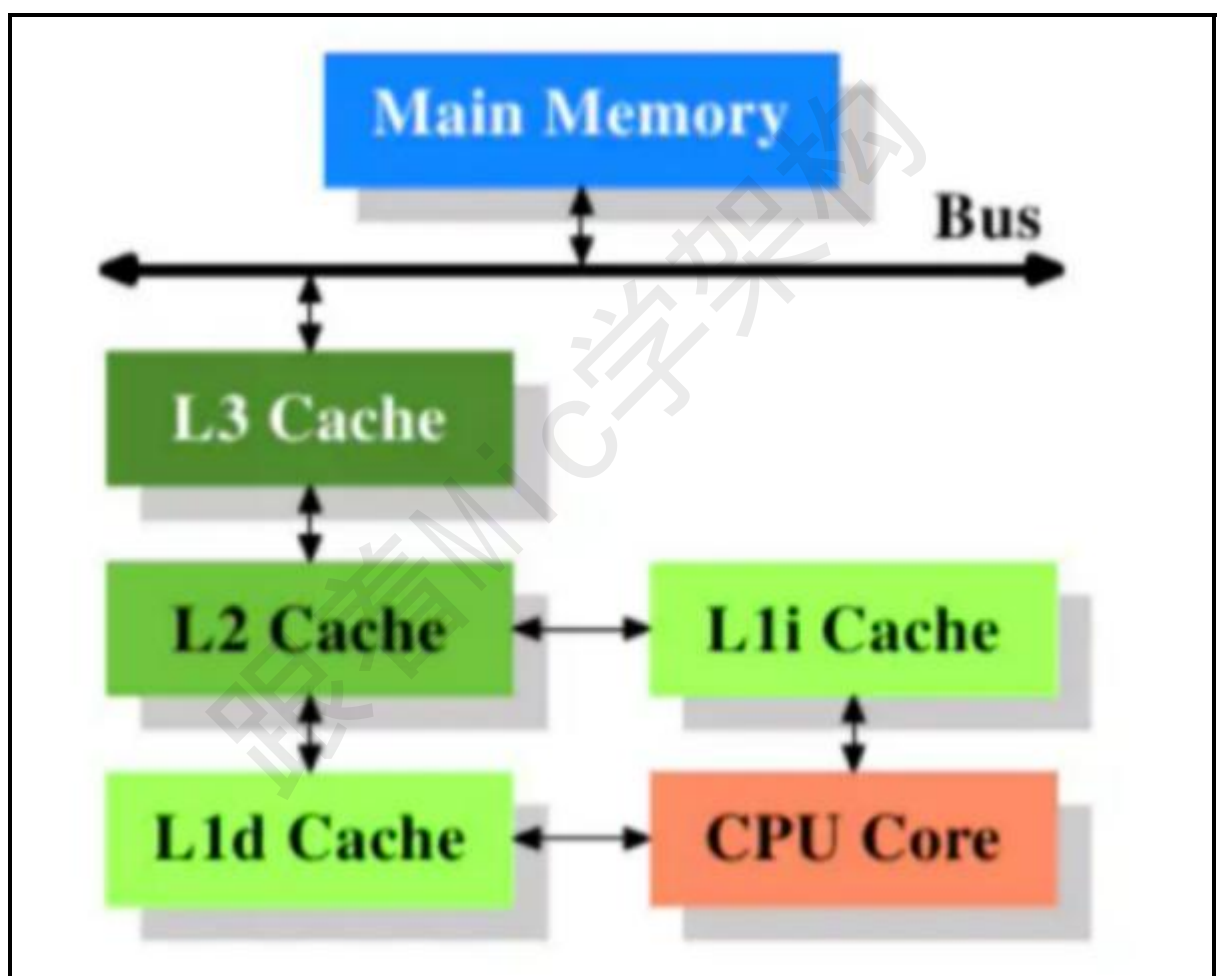
可以保证在多线程环境下共享变量的可见性。

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。

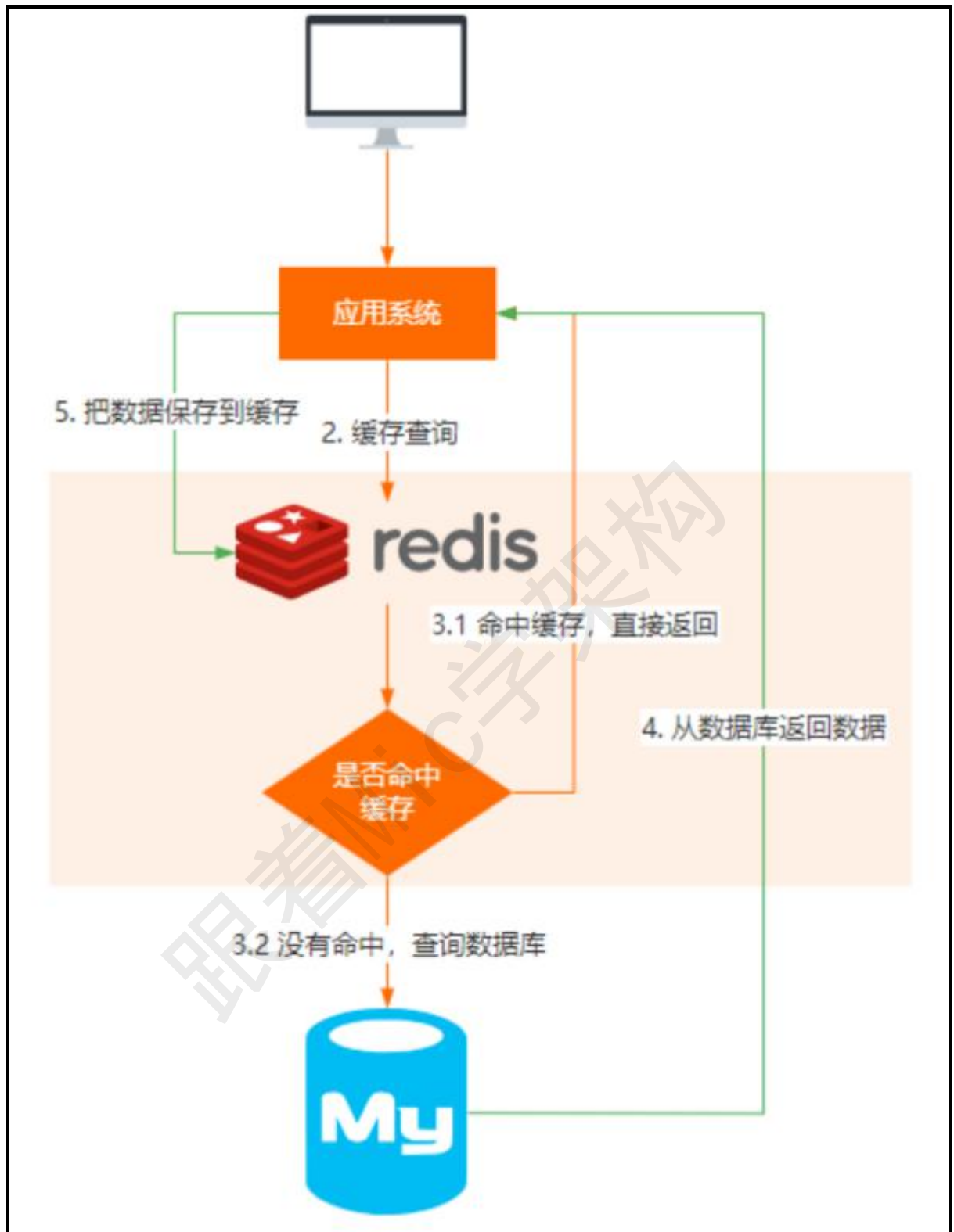
普通人

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关键字请求到应用里面，而这些不存在的关键字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，而只有关键字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：

把无效的关键字也保存到 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的关键字来访问，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关键字来请求的时候，先到布隆过滤器里面查询，如果不存在，那意味着这个关键字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。



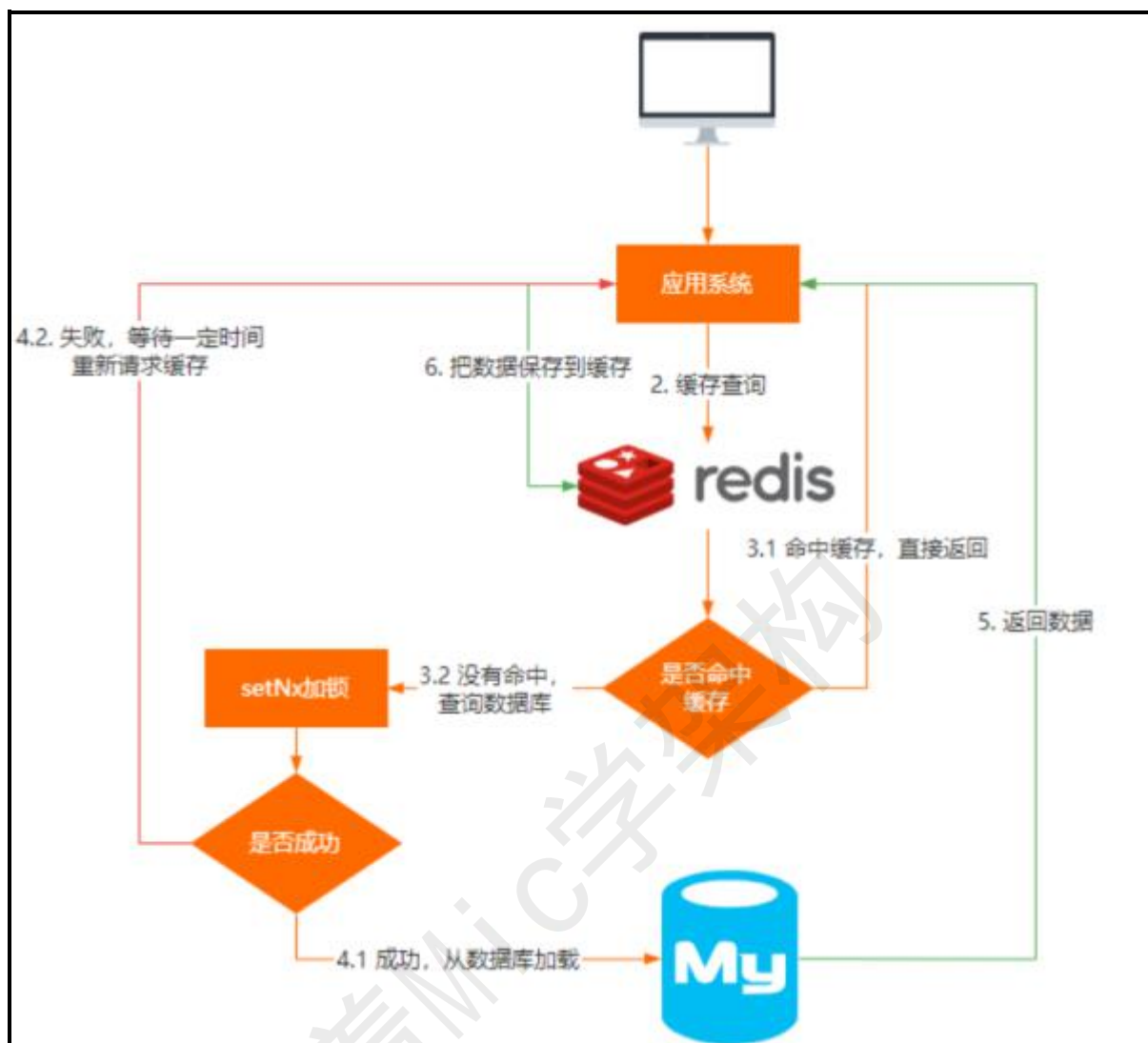
不过，在我看来，您提出来的这个问题，有点过于放大了它带来的影响。

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关键字有一定的差别。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。

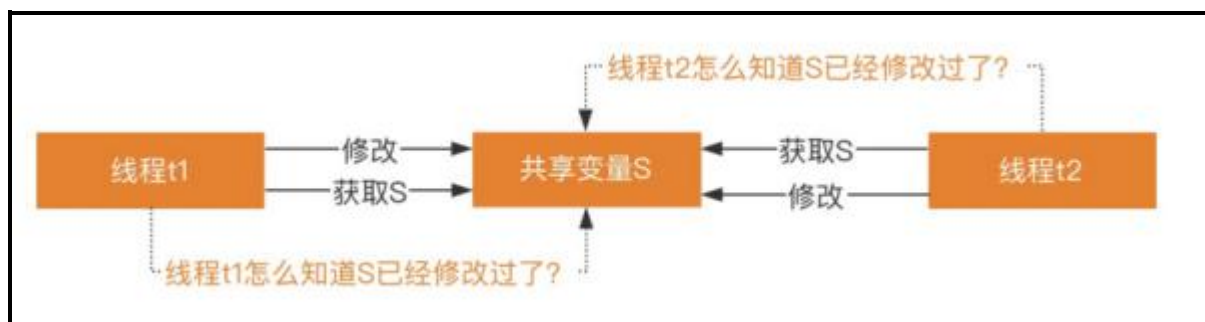
wait 和 notify 必然是成对出现的，如果一个线程被 wait() 方法阻塞，那么必然需要另外一个线程通过 notify() 方法来唤醒这个被阻塞的线程，从而实现多线程之间的通信。

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，就必须要有有一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。

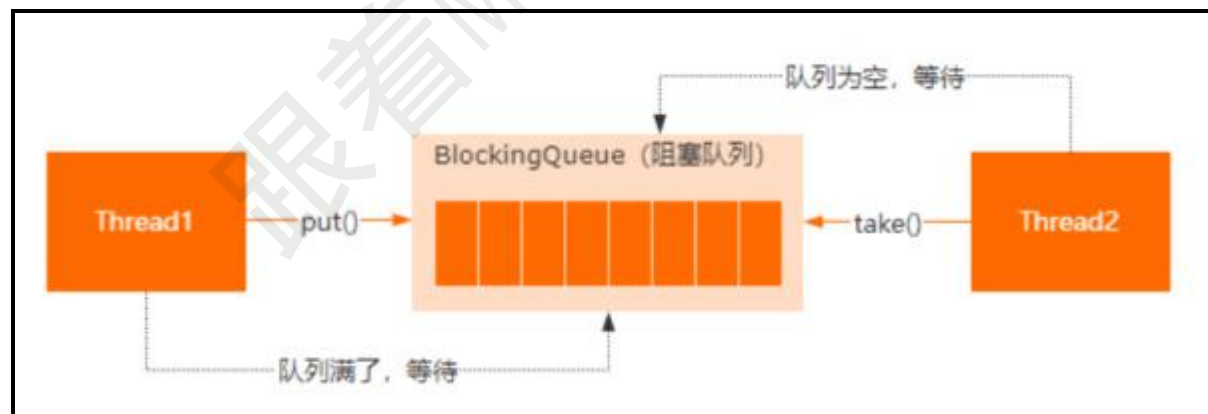
“基于数组的阻塞队列 ArrayBlockingQueue”的实现原理。

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。

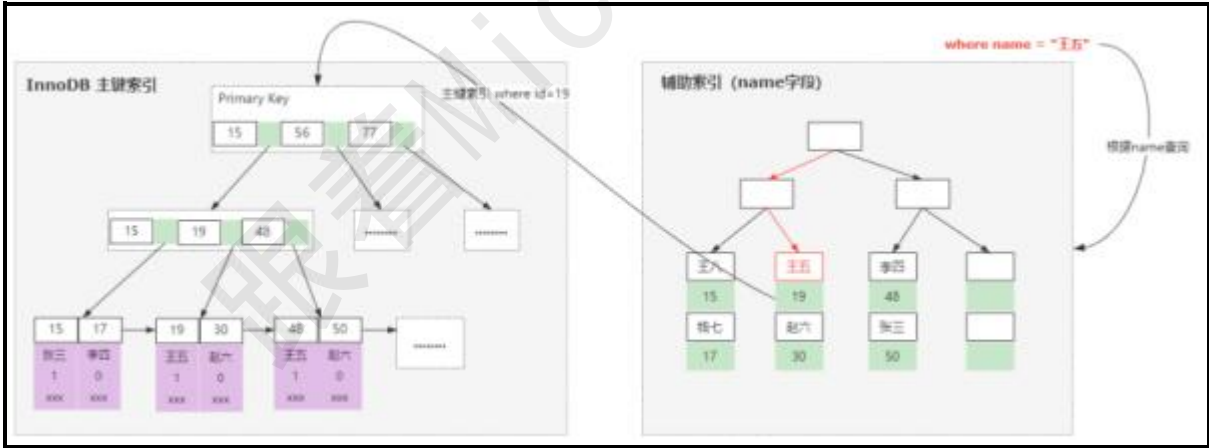
由于在 InnoDB 引擎里面，一张表的数据对应的物理文件本身就是按照 B+树来组织的一种索引结构，而聚集索引就是按照每张表的主键来构建一颗 B+树，然后叶子节点里面存储了这个表的每一行数据记录。

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

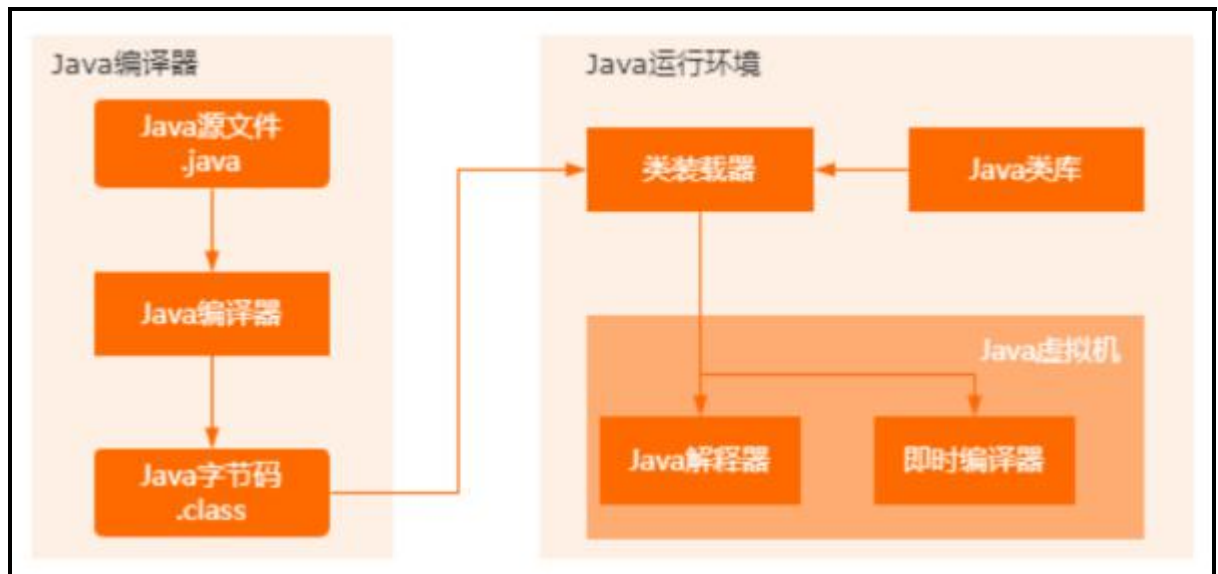
高手的回答

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

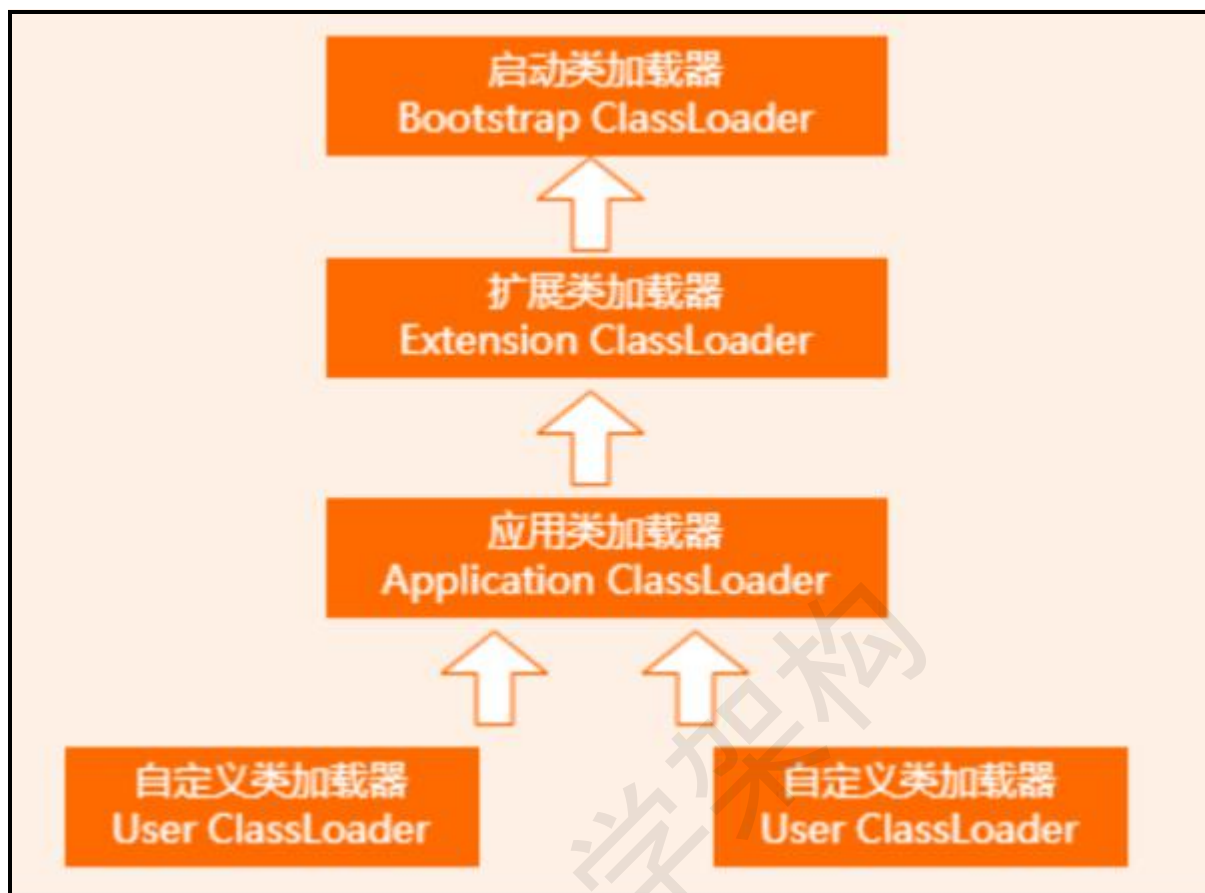
每个类加载器分别去加载不同作用范围的 jar 包，比如

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

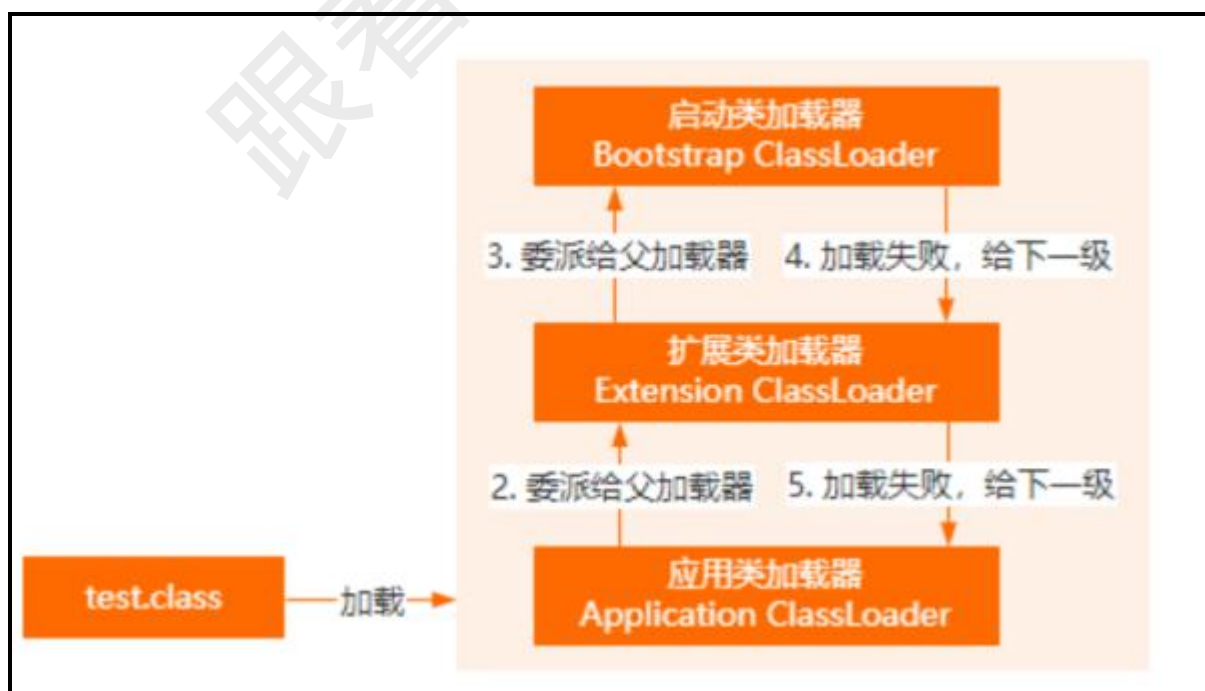
Application ClassLoader，主要负责当前应用里面的 `classpath` 下的所有 jar 包和类文件

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。

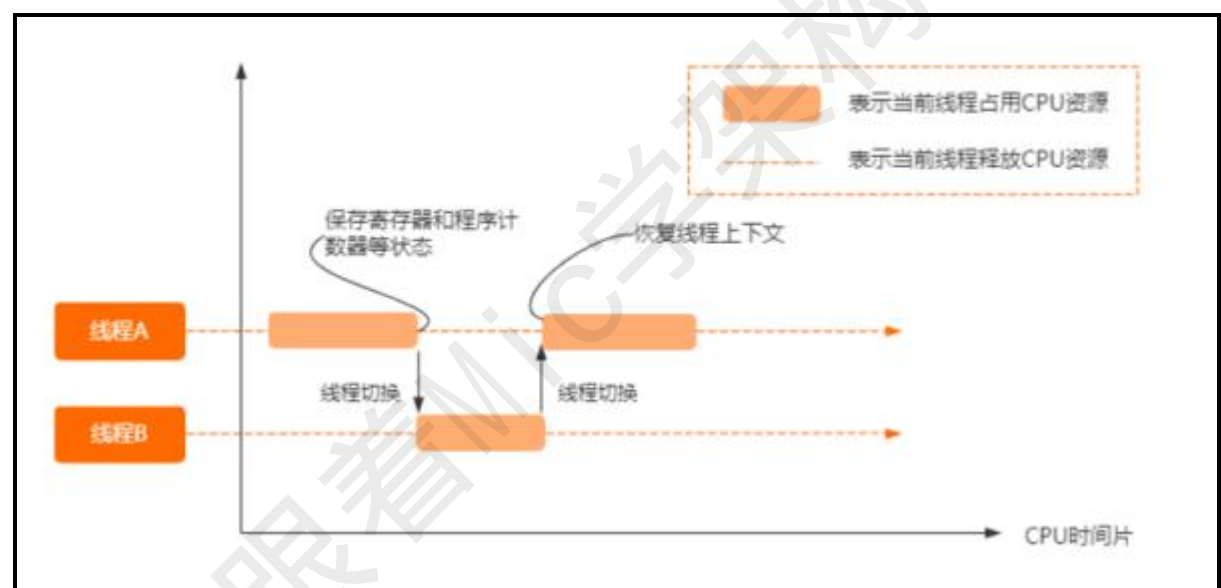
在程序中不做任何同步干预操作的情况下，这个方法或者对象的执行/修改都能按照预期的结果来反馈，那么这个类就是线程安全的。

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，就是说在多线程环境下，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。

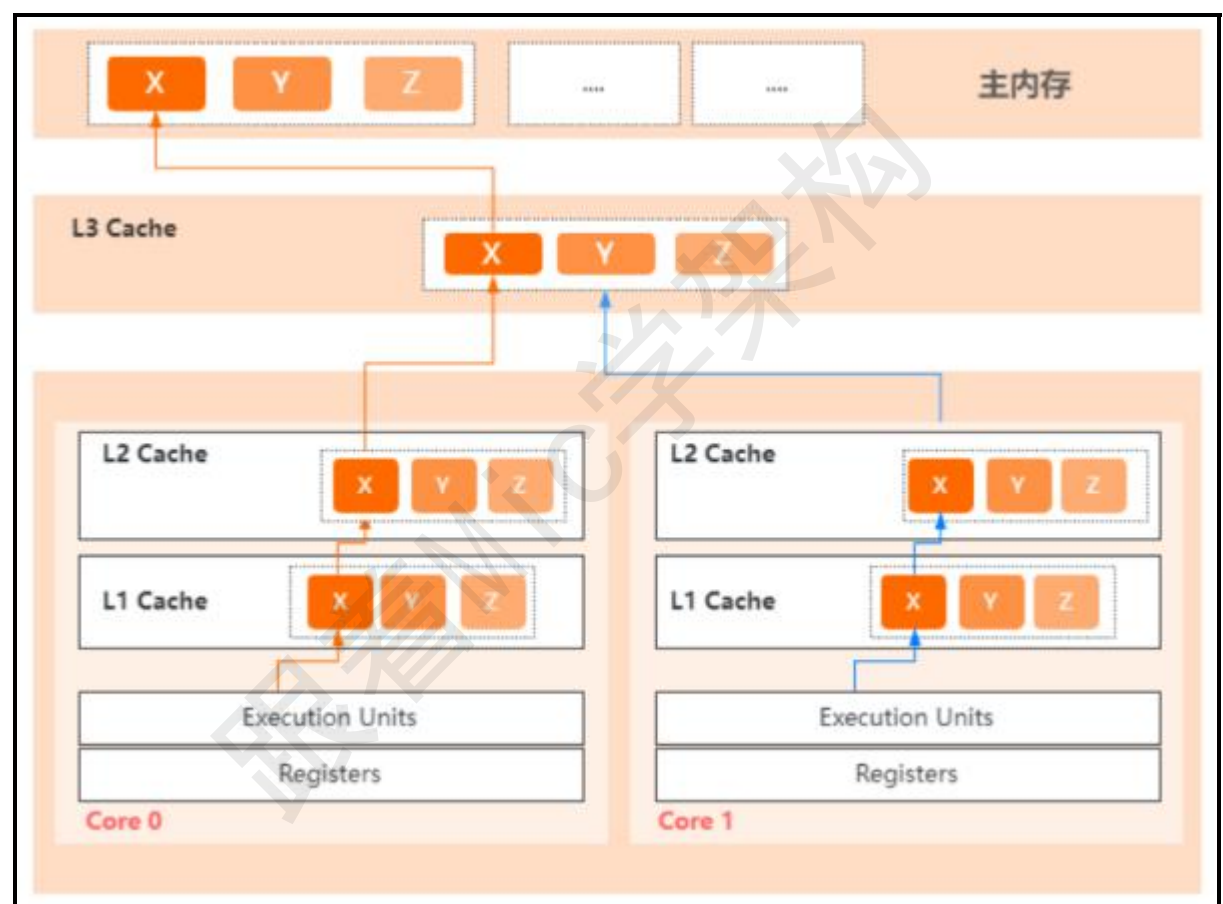
正是因为这种缓存行的设计，导致如果多个线程修改同一个缓存行里面的多个独立变量的时候，基于缓存一致性协议，就会无意中影响了彼此的性能，这就是伪共享的问题。

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。

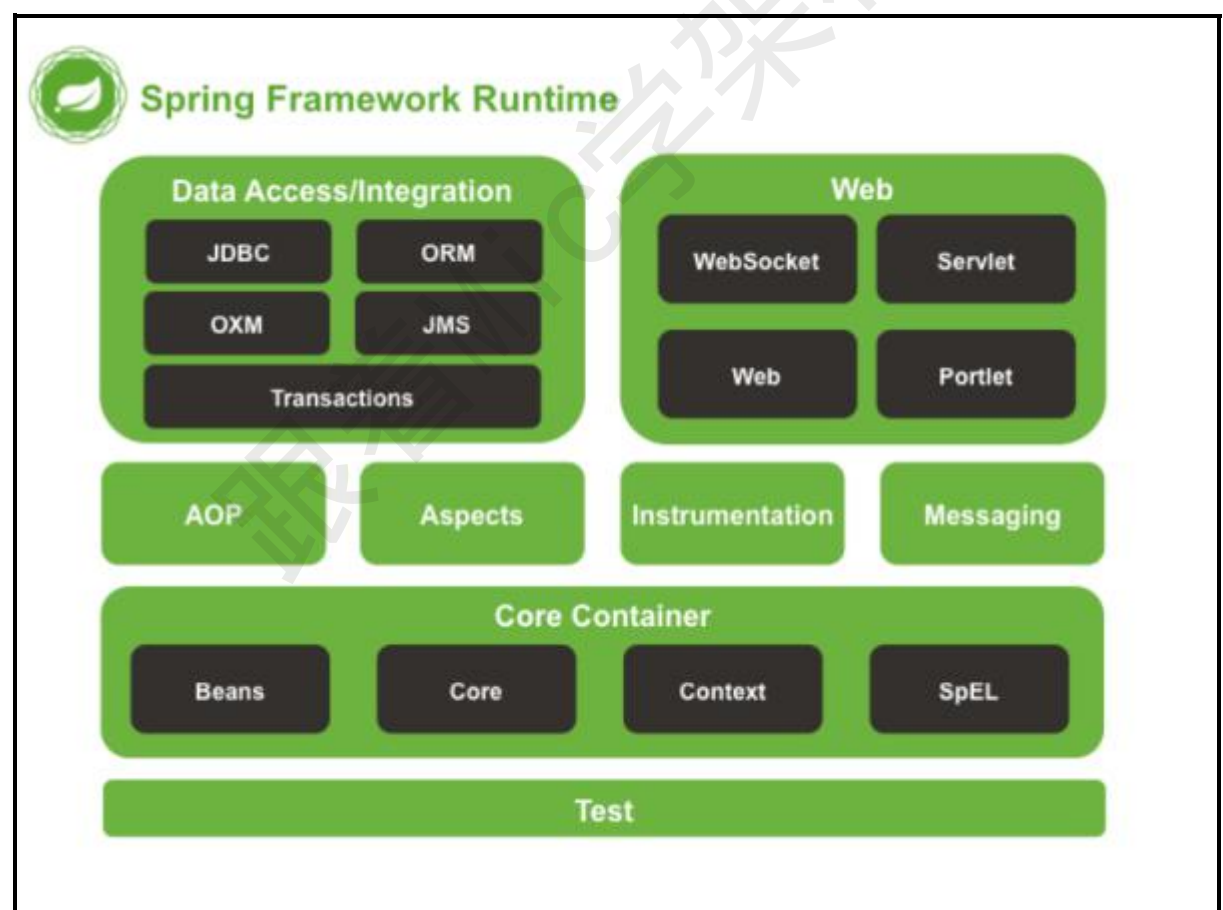
面向切面的编程(AOP)：Spring 支持面向切面的编程，从而把应用业务逻辑和系统服务分开。

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

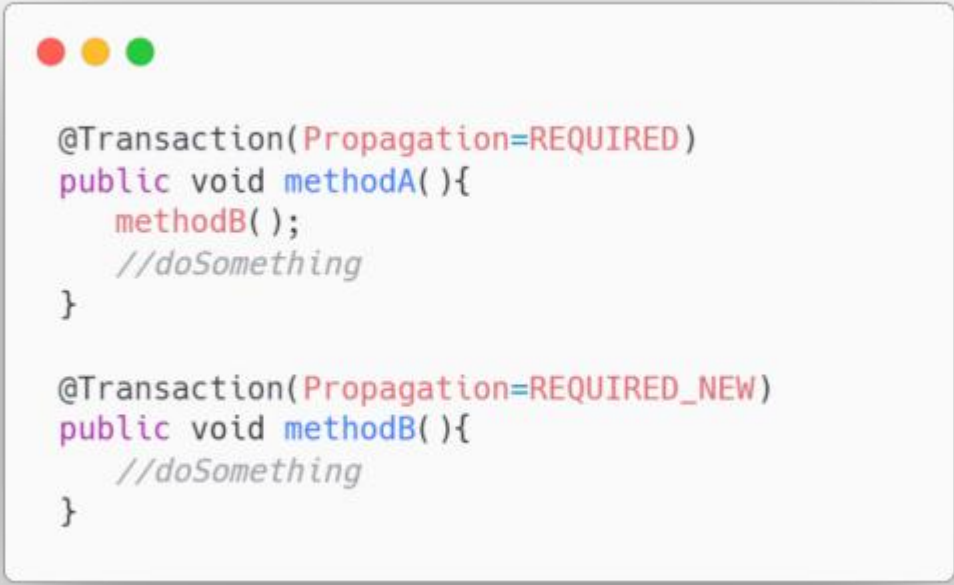
高手

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。

跟着Mic学架构