! 写在前面的话

石杉老师课程汇总：

石杉老师架构课程

- 架构课程试看汇总
- 架构课程完整版大纲

亿级流量电商详情页系统实战（完整版）

Elasticsearch顶尖高手系列课程

互联网Java工程师面试突击第一季

互联网Java工程师面试突击第二季

互联网Java工程师面试突击第三季

本文档内容出自于 石杉的架构笔记 欢迎扫描下方二维码关注，**每天定时分享技术干货！**

石杉的架构笔记-

大白话聊聊 Java 并发面试问题之 volatile 到底是什么？

> 作者:中华石杉 原文地址

一、写在前面

前段时间把几年前带过的一个项目架构演进的过程整理了一个系列出来，参见（《亿级流量架构系列专栏总结》）。

不过很多同学看了之后，后台反馈说文章太烧脑，看的云里雾里。其实这个也正常，文章承载的信息毕竟有限，而架构的东西细节太多，想要仅仅通过文章看懂一个系统架构的设计和落地，确实难度不小。

所以接下来用大白话跟大家聊点轻松的话题，比较易于理解，而且对大家工作和面试都很有帮助。

二、场景引入，问题初现

很多同学出去面试，都会被问到一个常见的问题：说说你对 volatile 的理解？

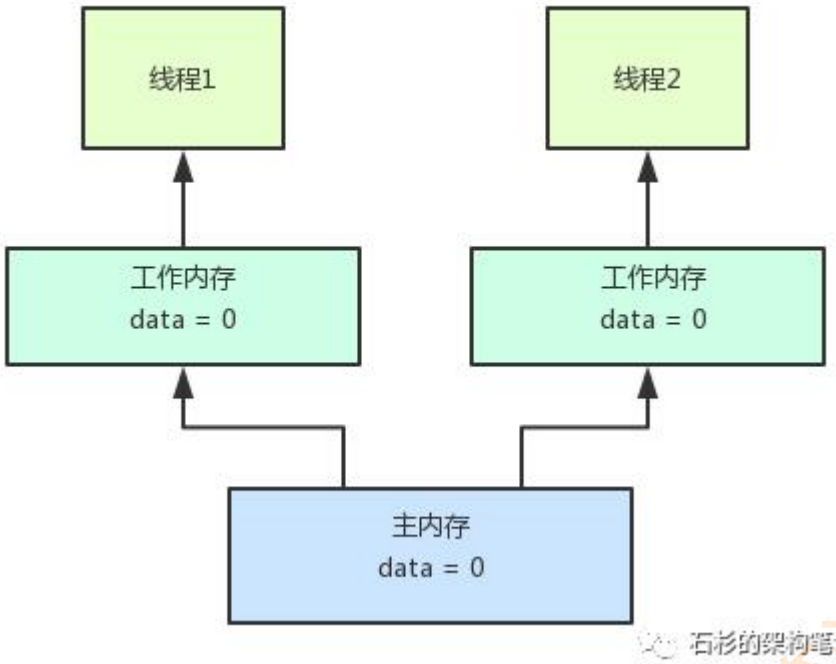
不少初出茅庐的同学可能会有点措手不及，因为可能就是之前没关注过这个。但是网上百度一下呢，不少文章写的很好，但是理论扎的太深，文字太多，图太少，让人有点难以理解。

基于上述痛点，这篇文章尝试站在年轻同学的角度，用最简单的大白话，加上多张图给大家说一下，volatile 到底是什么？

当然本文不会把理论扎的太深，因为一下子扎深了文字太多，很多同学还是会不好理解。

本文仅仅是定位在用大白话的语言将 volatile 这个东西解释清楚，而涉及到特别底层的一些原理和技术问题，以后有机会开文再写。

首先，给大家上一张图，咱们来一起看看：



如上图，这张图说的是 java 内存模型中，每个线程有自己的工作内存，同时还有一个共享的主内存。

举个例子，比如说有两个线程，他们的代码里都需要读取 data 这个变量的值，那么他们都会从主内存里加载 data 变量的值到自己的工作内存，然后才可以使用那个值。

好了，现在大家从图里看到，每个线程都把 data 这个变量的副本加载到了自己的工作内存里了，所以每个线程都可以读到 data = 0 这个值。

这样，在线程代码运行的过程中，对 data 的值都可以直接从工作内存里加载了，不需要再从主内存里加载了。

那问题来了，为啥一定要让每个线程用一个工作内存来存放变量的副本以供读取呢？我直接让线程每次都从主内存加载变量的值不行吗？

很简单！因为线程运行的代码对应的是一些指令，是由 CPU 执行的！但是 CPU 每次执行指令运算的时候，也就是执行我们写的那一大坨代码的时候，要是每次需要一个变量的值，都从主内存加载，性能会比较差！

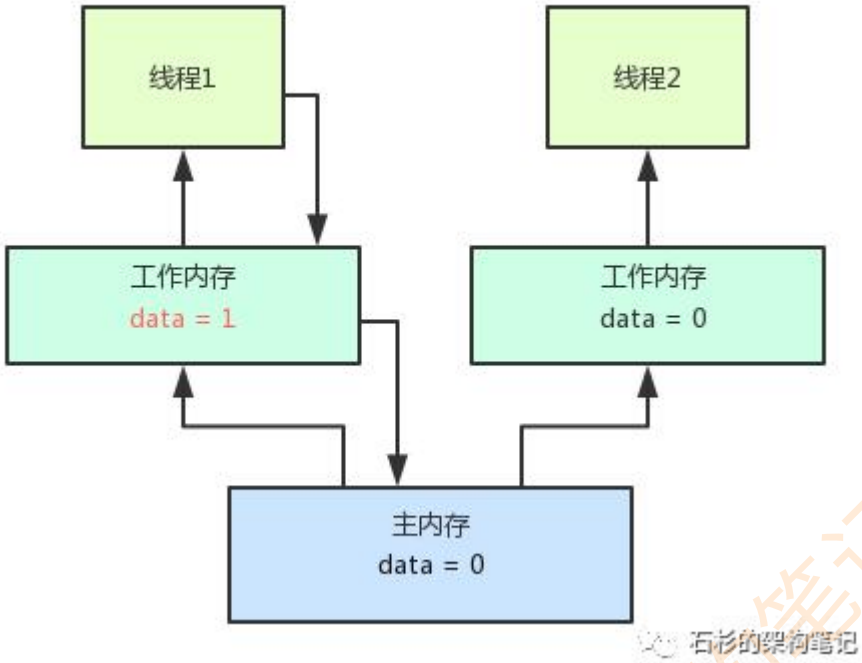
所以说后来想了一个办法，就是线程有工作内存的概念，类似于一个高速的本地缓存。

这样一来，线程的代码在执行过程中，就可以直接从自己本地缓存里加载变量副本，不需要从主内存加载变量值，性能可以提升很多！

但是大家思考一下，这样会有什么问题？

我们来设想一下，假如说线程 1 修改了 data 变量的值为 1，然后将这个修改写入自己的本地工作内存。那么此时，线程 1 的工作内存里的 data 值为 1。

然而，主内存里的 data 值还是为 0！线程 2 的工作内存里的 data 值还是 0 啊？！



这可尴尬了，那接下来，在线程 1 的代码运行过程中，他可以直接读到 data 最新的值是 1，但是线程 2 的代码运行过程中读到的 data 的值还是 0！

这就导致，线程 1 和线程 2 其实都是在操作一个变量 data，但是线程 1 修改了 data 变量的值之后，线程 2 是看不到的，一直都是看到自己本地工作内存中的一个旧的副本的值！

这就是所谓的 java 并发编程中的可见性问题：

多个线程并发读写一个共享变量的时候，有可能某个线程修改了变量的值，但是其他线程看不到！也就是对其他线程不可见！

三、volatile 的作用及背后的原理

那如果要解决这个问题怎么办呢？这时就轮到 volatile 闪亮登场了！你只要给 data 这个变量在定义的时候加一个 volatile，就直接可以完美的解决这个可见性的问题。

比如下面的这样的代码，在加了 volatile 之后，会有啥作用呢？

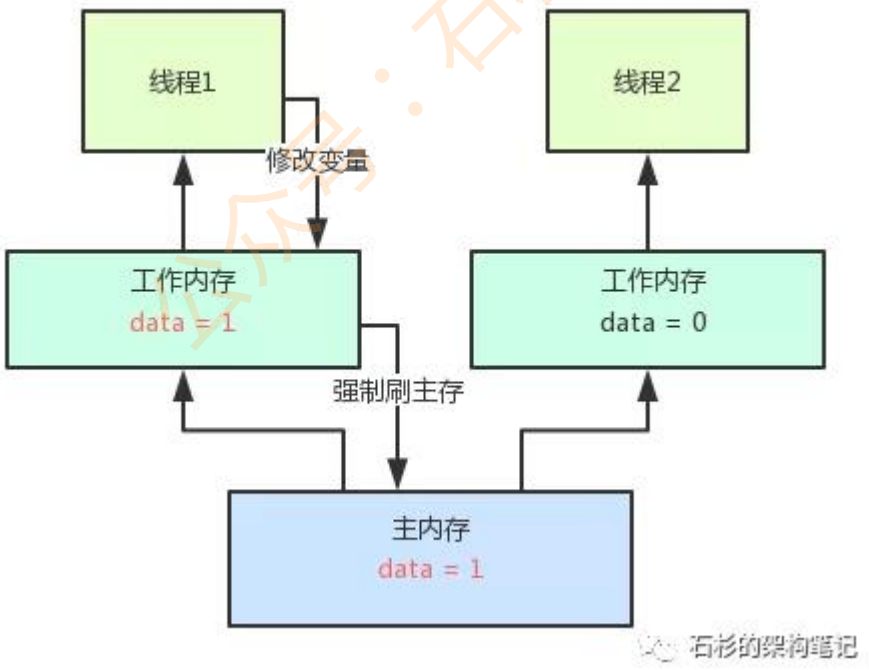
```
public class HelloWorld {

    private volatile int data = 0;

    // 线程1会读取和修改data变量值

    // 线程2会读取data变量值
}
```


完整的作用就不给大家解释了，因为我们定位就是大白话，要是把底层涉及的各种内存屏障、指令重排等概念在这里带出来，不少同学又要蒙圈了！

我们这里，就说说他最关键的几个作用是啥？

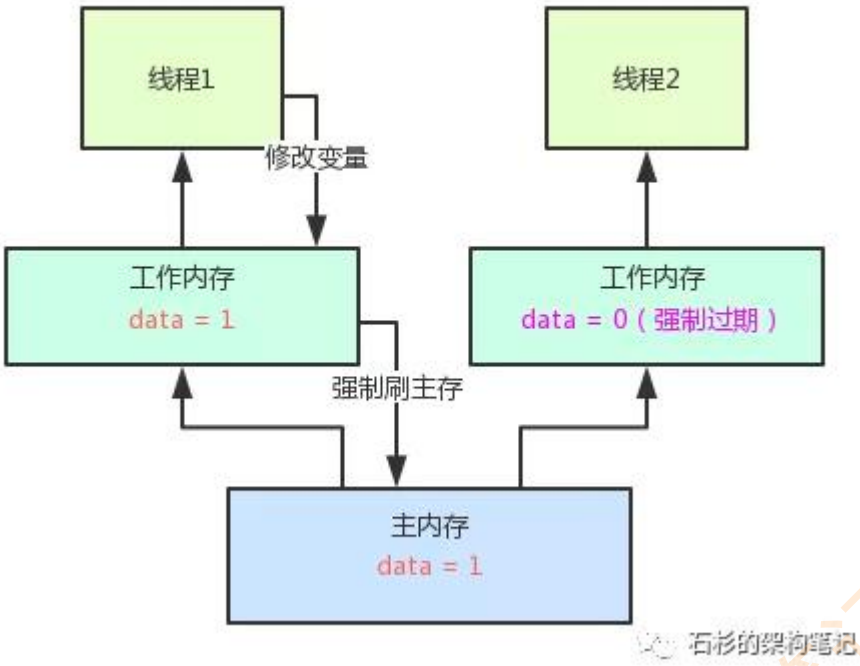
第一，一旦 data 变量定义的时候前面加了 volatile 来修饰的话，那么线程 1 只要修改 data 变量的值，就会在修改完自己本地工作内存的 data 变量值之后，强制将这个 data 变量最新的值刷回主内存，必须让主内存里的 data 变量值立马变成最新的值！

整个过程，如下图所示：



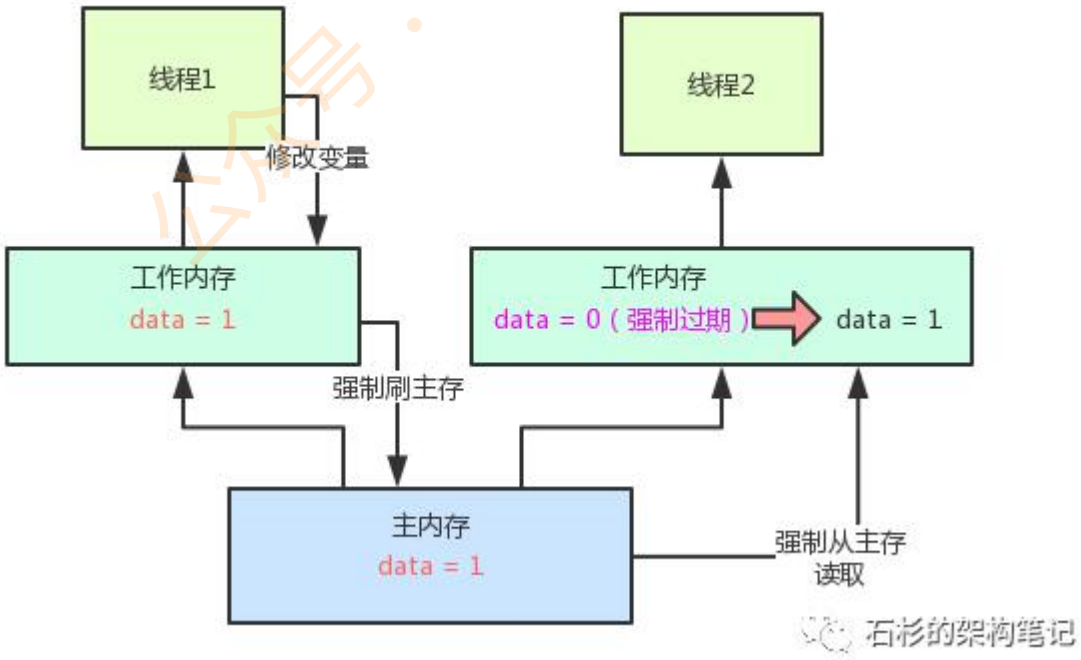
第二，如果此时别的线程的工作内存中有这个 data 变量的本地缓存，也就是一个变量副本的话，那么会强制让其他线程的工作内存中的 data 变量缓存直接失效过期，不允许再次读取和使用了！

整个过程，如下图所示：



第三，如果线程 2 在代码运行过程中再次需要读取 data 变量的值，此时尝试从本地工作内存中读取，就会发现这个 data = 0 已经过期了！

此时，他就必须重新从主内存中加载 data 变量最新的值！那么不就可以读取到 data = 1 这个最新的值了！整个过程，参见下图：



bingo！好了，volatile 完美解决了 java 并发中可见性的问题！

对一个变量加了 volatile 关键字修饰之后，只要一个线程修改了这个变量的值，立马强制刷回主内存。

接着强制过期其他线程的本地工作内存中的缓存，最后其他线程读取变量值的时候，强制重新从主内存来加载最新的值！

这样就保证，任何一个线程修改了变量值，其他线程立马就可以看见了！这就是所谓的 volatile 保证了可见性的工作原理！

四、总结 & 提醒

最后给大家提一嘴，volatile 主要作用是保证可见性以及有序性。

有序性涉及到较为复杂的指令重排、内存屏障等概念，本文没提及，但是 volatile 是不能保证原子性的！

也就是说，volatile 主要解决的是一个线程修改变量值之后，其他线程立马可以读到最新的值，是解决这个问题的，也就是可见性！

但是如果是多个线程同时修改一个变量的值，那还是可能出现多线程并发的安全问题，导致数据值修改错乱，volatile 是不负责解决这个问题的，也就是不负责解决原子性问题！

原子性问题，得依赖 synchronized、ReentrantLock 等加锁机制来解决。

大白话聊聊 Java 并发面试问题之 Java 8 如何优化 CAS 性能?

> **作者:中华石杉** 原文地址

一、前情回顾

上篇文章给大家聊了一下 volatile 的原理，具体参见：**《大白话聊聊 Java 并发面试问题之 volatile 到底是什么？》**

这篇文章给大家聊一下 java 并发包下的 CAS 相关的原子操作，以及 Java 8 如何改进和优化 CAS 操作的性能。

因为 Atomic 系列的原子类，无论在并发编程、JDK 源码、还是各种开源项目中，都经常用到。而且在 Java 并发面试中，这一块也属于比较高频的考点，所以还是值得给大家聊一聊。

二、场景引入，问题凸现

好，我们正式开始！假设多个线程需要对一个变量不停的累加 1，比如说下面这段代码：

```
public class HelloWorld {

    private int data = 0;

    // 多个线程同时对data变量执行操作：data++

}
```


实际上，上面那段代码是不 ok 的，因为多个线程直接这样并发的对一个 data 变量进行修改，是线程不安全性的行为，会导致 data 值的变化不遵照预期的值来改变。

举个例子，比如说 20 个线程分别对 data 执行一次 data++ 操作，我们以为最后 data 的值会变成 20，其实不是。

最后可能 data 的值是 18，或者是 19，都有可能，因为多线程并发操作下，就是会有这种安全问题，导致数据结果不准确。

至于为什么会不准确？那不在本文讨论的范围里，因为这个一般只要是学过 java 的同学，肯定都了解过多线程并发问题。

三、初步的解决方案：synchronized

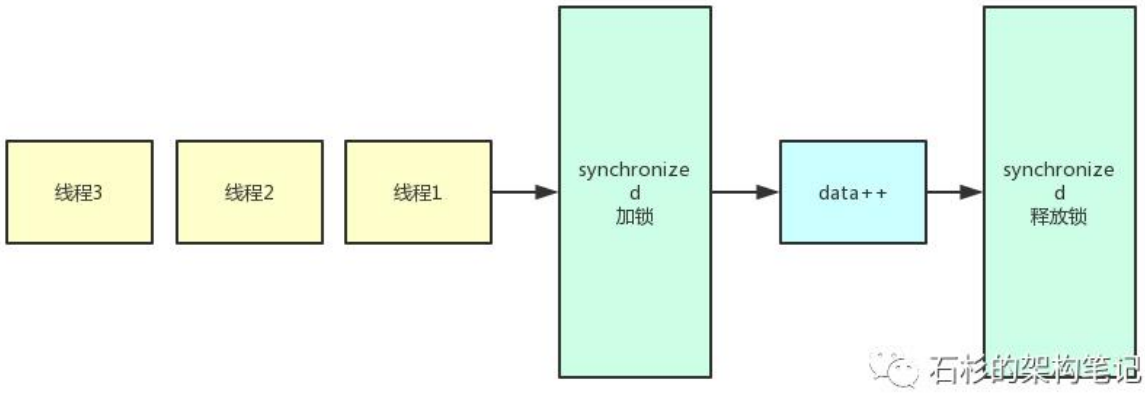
所以，对于上面的代码，一般我们会改造一下，让他通过加锁的方式变成线程安全的：

```
public class HelloWorld {

    private int data = 0;

    public synchronized void increment() {
        data++;
    }

    // 多个线程同时调用方法：increment()

}
```


这个时候，代码就是线程安全的了，因为我们加了 synchronized，也就是让每个线程要进入 increment() 方法之前先得尝试加锁，同一时间只有一个线程能加锁，其他线程需要等待锁。

通过这样处理，就可以保证换个 data 每次都会累加 1，不会出现数据错乱的问题。

老规矩！我们来看看下面的图，感受一下 synchronized 加锁下的效果和氛围，相当于 N 个线程一个一个的排队在更新那个数值。



但是，如此简单的 data++ 操作，都要加一个重磅的 synchronized 锁来解决多线程并发问题，就有点杀鸡用牛刀，大材小用了。

虽然随着 Java 版本更新，也对 synchronized 做了很多优化，但是处理这种简单的累加操作，仍然显得 “太重了”。人家 synchronized 是可以解决更加复杂的并发编程场景和问题的。

而且，在这个场景下，你要是用 synchronized，不就相当于让各个线程串行化了么？一个接一个的排队，加锁，处理数据，释放锁，下一个再进来。

四、更高效的方案：Atomic 原子类及其底层原理

对于这种简单的 data++ 类的操作，其实我们完全可以换一种做法，java 并发包下面提供了一系列的 Atomic 原子类，比如说 AtomicInteger。

他可以保证多线程并发安全的情况下，高性能的并发更新一个数值。我们来看下面的代码：

```
public class HelloWorld {

    private AtomicInteger data = new AtomicInteger(0);

    // 多个线程并发的执行：data.incrementAndGet()

}
```

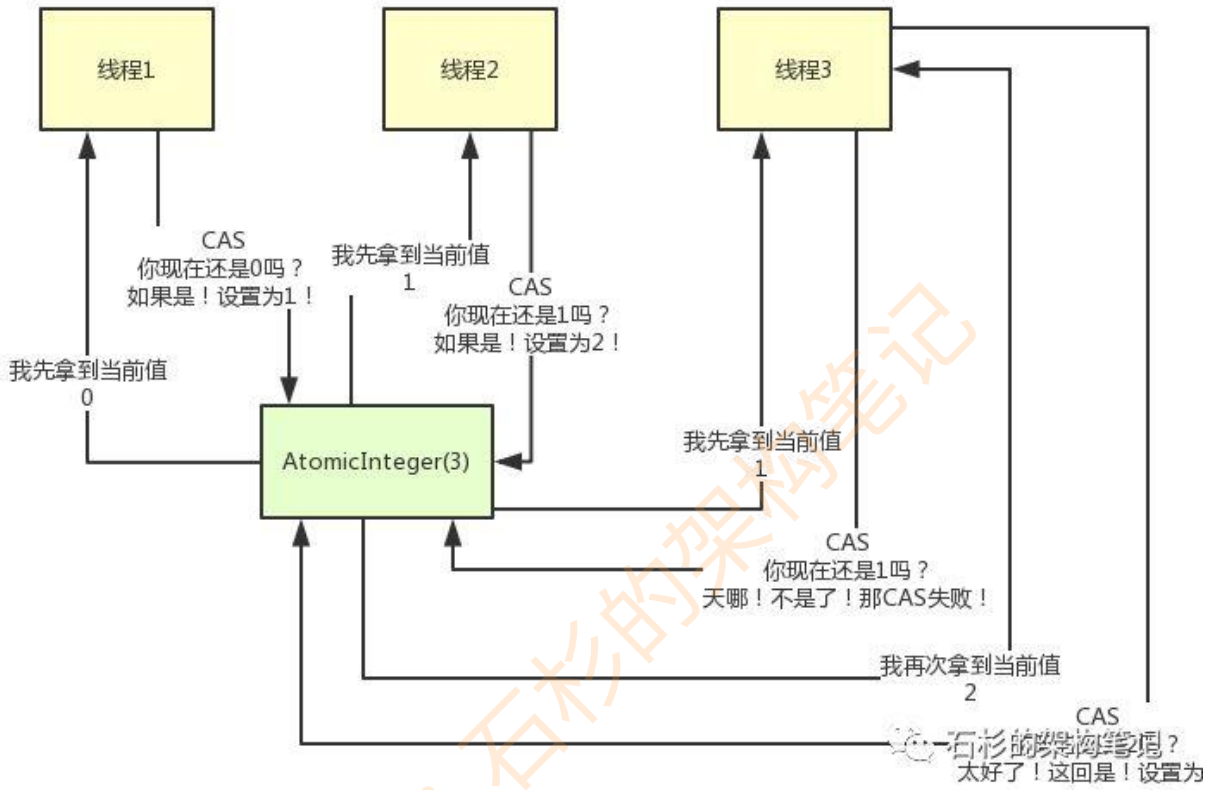

大家看上面的代码，是不是很简单！多个线程可以并发的执行 AtomicInteger 的 incrementAndGet() 方法，意思就是给我把 data 的值累加 1，接着返回累加后最新的值。

这个代码里，就没有看到加锁和释放锁这一说了吧！

实际上，Atomic 原子类底层用的不是传统意义的锁机制，而是无锁化的 CAS 机制，通过 CAS 机制保证多线程修改一个数值的安全性

那什么是 CAS 呢？他的全称是：Compare and Set，也就是先比较再设置的意思。

话不多说，先上图！



我们来看上面的图，假如说有 3 个线程并发的要修改一个 AtomicInteger 的值，他们底层的机制如下：

首先，每个线程都会先获取当前的值，接着走一个原子的 CAS 操作，原子的意思就是这个 CAS 操作一定是自己完整执行完的，不会被别人打断。

然后 CAS 操作里，会比较一下说，唉！大兄弟！现在你的值是不是刚才我获取到的那个值啊?

如果是的话，bingo！说明没人改过这个值，那你给我设置成累加 1 之后的一个值好了！

同理，如果有人在执行 CAS 的时候，发现自己之前获取的值跟当前的值不一样，会导致 CAS 失败，失败之后，进入一个无限循环，再次获取值，接着执行 CAS 操作！

好！现在我们对照着上面的图，来看一下这整个过程：

> **首先第一步，我们假设线程一咔嚓一下过来了，然后对 AtomicInteger 执行 incrementAndGet() 操作，他底层就会先获取 AtomicInteger 当前的值，这个值就是 0。**

> 此时没有别的线程跟他抢！他也不管那么多，直接执行原子的 CAS 操作，问问人家说：兄弟，你现在值还是 0 吗？

> 如果是，说明没人修改过啊！太好了，给我累加 1，设置为 1。于是 AtomicInteger 的值变为 1！

> 接着线程 2 和线程 3 同时跑了过来，因为底层不是基于锁机制，都是无锁化的 CAS 机制，所以他们俩可能会并发的同时执行 incrementAndGet() 操作。

> 然后俩人都获取到了当前 AtomicInteger 的值，就是 1

> 接着线程 2 抢先一步发起了原子的 CAS 操作！注意，CAS 是原子的，此时就他一个线程在执行！

> 然后线程 2 问：兄弟，你现在值还是 1 吗？如果是，太好了，说明没人改过，我来改成 2

> 好了，此时 AtomicInteger 的值变为了 2。关键点来了：现在线程 3 接着发起了 CAS 操作，但是他手上还是拿着之前获取到的那个 1 啊！

> 线程 3 此时会问问说：兄弟，你现在值还是 1 吗？

> 噩耗传来！！！这个时候的值是 2 啊！线程 3 哭泣了，他说，居然有人在这个期间改过值。算了，那我还是重新再获取一次值吧，于是获取到了最新的值，值为 2。

然后再次发起 CAS 操作，问问，现在值是 2 吗？是的！太好了，没人改，我抓紧改，此时 AtomicInteger 值变为 3！

上述整个过程，就是所谓 Atomic 原子类的原理，没有基于加锁机制串行化，而是基于 CAS 机制：先获取一个值，然后发起 CAS，比较这个值被人改过没？如果没有，就更改值！这个 CAS 是原子的，别人不会打断你！

通过这个机制，不需要加锁这么重量级的机制，也可以用轻量级的方式实现多个线程安全的并发的修改某个数值。

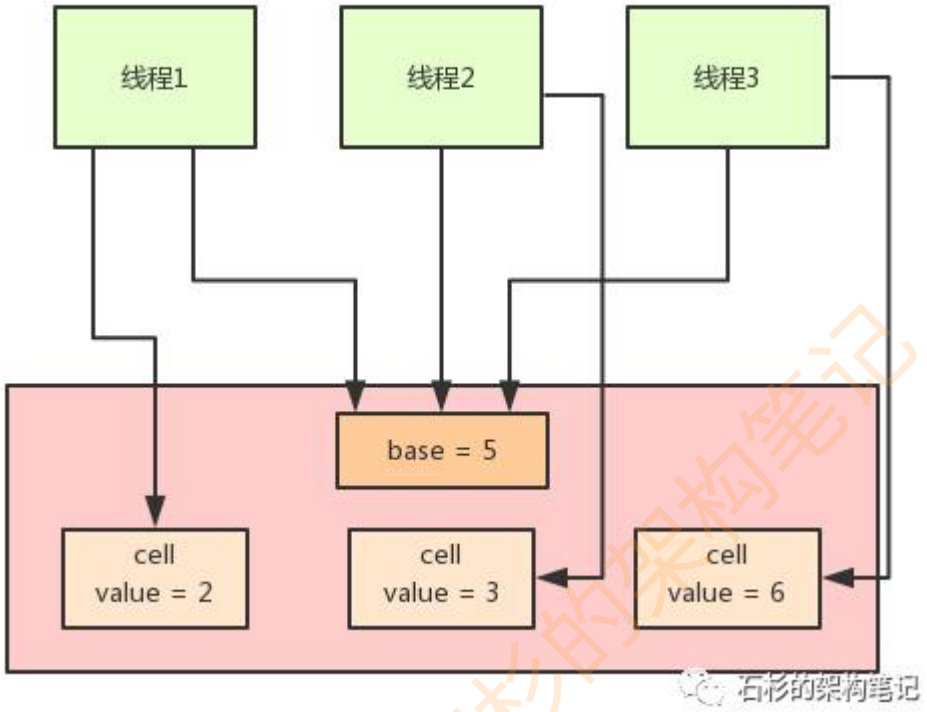
五、Java 8 对 CAS 机制的优化

但是这个 CAS 有没有问题呢？肯定是有的。比如说大量的线程同时并发修改一个 AtomicInteger，可能有很多线程会不停的自旋，进入一个无限重复的循环中。

这些线程不停地获取值，然后发起 CAS 操作，但是发现这个值被别人改过了，于是再次进入下一个循环，获取值，发起 CAS 操作又失败了，再次进入下一个循环。

在大量线程高并发更新 AtomicInteger 的时候，这种问题可能会比较明显，导致大量线程空循环，自旋转，性能和效率都不是特别好。

于是，当当当当，Java 8 推出了一个新的类，LongAdder，他就是尝试使用分段 CAS 以及自动分段迁移的方式来大幅度提升多线程高并发执行 CAS 操作的性能！



在 LongAdder 的底层实现中，首先有一个 base 值，刚开始多线程来不停的累加数值，都是对 base 进行累加的，比如刚开始累加成了 base = 5。

接着如果发现并发更新的线程数量过多，就会开始施行分段 CAS 的机制，也就是内部会搞一个 Cell 数组，每个数组是一个数值分段。

这时，让大量的线程分别去对不同 Cell 内部的 value 值进行 CAS 累加操作，这样就把 CAS 计算压力分散到了不同的 Cell 分段数值中了！

这样就可以大幅度的降低多线程并发更新同一个数值时出现的无限循环的问题，大幅度提升了多线程并发更新数值的性能和效率！

而且他内部实现了自动分段迁移的机制，也就是如果某个 Cell 的 value 执行 CAS 失败了，那么就会自动去找另外一个 Cell 分段内的 value 值进行 CAS 操作。

这样也解决了线程空旋转、自旋不停等待执行 CAS 操作的问题，让一个线程过来执行 CAS 时可以尽快的完成这个操作。

最后，如果你要从 LongAdder 中获取当前累加的总值，就会把 base 值和所有 Cell 分段数值加起来返回给你。

六、总结 & 思考 不知道大家有没有发现这种高并发访问下的分段处理机制，在很多地方都有类似的思想体现！因为高并发中的分段处理机制实际上是一个很常见和常用的并发优化手段。

在我们之前的一篇讲分布式锁的文章：**（《每秒上千订单场景下的分布式锁高并发优化实践》）**，也是用到了分段加锁以及自动分段迁移 / 合并加锁的一套机制，来大幅度几十倍的提升分布式锁的并发性能。

所以其实很多技术，思想都是有异曲同工之妙的。

大白话聊聊Java并发面试问题之谈谈你对AQS的理解?

> **作者:中华石杉** 原文地址

一、写在前面

> ! 上一篇文章聊了一下 java 并发中常用的原子类的原理和 Java 8 的优化，具体请参见文章：**《大白话聊聊 Java 并发面试问题之 Java 8 如何优化 CAS 性能？》**

这篇文章，我们来聊聊面试的时候比较有杀伤力的一个问题：聊聊你对 AQS 的理解?

之前有同学反馈，去互联网公司面试，面试官聊到并发时就问到了这个问题。当时那位同学内心估计受到了一万点伤害。。。

因为首先，很多人还真的连 AQS 是什么都不知道，可能听都没听说过。或者有的人听说过 AQS 这个名词，但是可能连具体全称怎么拼写都不知道。

更有甚者，可能会说：AQS？是不是一种思想？我们平时开发怎么来用 AQS?

总体来说，很多同学估计都对 AQS 有一种云里雾里的感觉，如果用搜索引擎查一下 AQS 是什么？看几篇文章，估计就直接放弃了，因为密密麻麻的文字，实在是看不懂！

所以，基于上述痛点，咱们这篇文章，就用最简单的大白话配合 N 多张手绘图，给大家讲清楚 AQS 到底是什么？让各位同学面试被问到这个问题时，不至于不知所措。

二、ReentrantLock 和 AQS 的关系

首先我们来看看，如果用 java 并发包下的 ReentrantLock 来加锁和释放锁，是个什么样的感觉?

这个基本学过 java 的同学应该都会吧，毕竟这个是 java 并发基本 API 的使用，应该每个人都是学过的，所以我们直接看一下代码就好了：

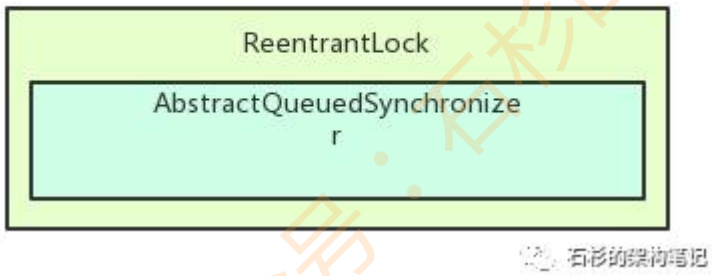
```
ReentrantLock lock = new ReentrantLock();

lock.lock(); // 加锁

// 业务逻辑代码。。。

lock.unlock(); //释放锁
```


上面那段代码应该不难理解吧，无非就是搞一个 Lock 对象，然后加锁和释放锁。

你这时可能会问，这个跟 AQS 有啥关系？关系大了去了！因为 java 并发包下很多 API 都是基于 AQS 来实现的加锁和释放锁等功能的，AQS 是 java 并发包的基础类。

举个例子，比如说 ReentrantLock、ReentrantReadWriteLock 底层都是基于 AQS 来实现的。

那么 AQS 的全称是什么呢？AbstractQueuedSynchronizer，抽象队列同步器。给大家画一个图先，看一下 ReentrantLock 和 AQS 之间的关系。



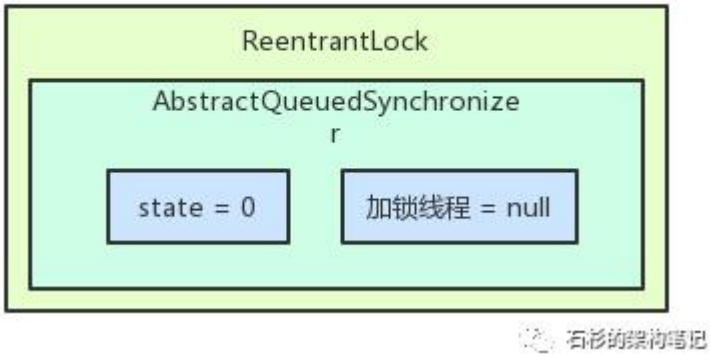
我们来看上面的图。说白了，ReentrantLock 内部包含了一个 AQS 对象，也就是 AbstractQueuedSynchronizer 类型的对象。这个 AQS 对象就是 ReentrantLock 可以实现加锁和释放锁的关键性的核心组件。

三、ReentrantLock 加锁和释放锁的底层原理

好了，那么现在如果有一个线程过来尝试用 ReentrantLock 的 lock() 方法进行加锁，会发生什么事情呢?

很简单，这个 AQS 对象内部有一个核心的变量叫做 state，是 int 类型的，代表了加锁的状态。初始状态下，这个 state 的值是 0。

另外，这个 AQS 内部还有一个关键变量，用来记录当前加锁的是哪个线程，初始化状态下，这个变量是 null。

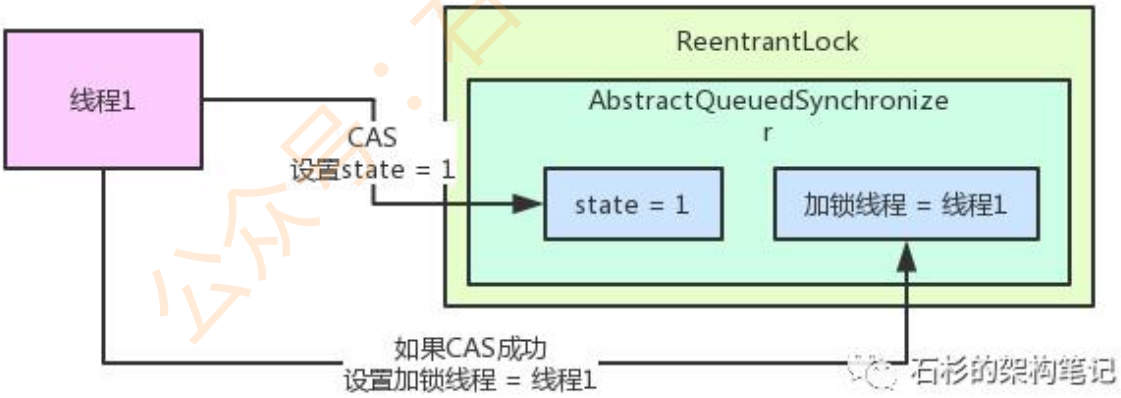


接着线程 1 跑过来调用 ReentrantLock 的 lock() 方法尝试进行加锁，这个加锁的过程，直接就是用 CAS 操作将 state 值从 0 变为 1。

如果不知道 CAS 是啥的，请看上篇文章，《大白话聊聊 Java 并发面试问题之 Java 8 如何优化 CAS 性能？》

如果之前没人加过锁，那么 state 的值肯定是 0，此时线程 1 就可以加锁成功。

一旦线程 1 加锁成功了之后，就可以设置当前加锁线程是自己。所以大家看下面的图，就是线程 1 跑过来加锁的一个过程。



其实看到这儿，大家应该对所谓的 AQS 有感觉了。说白了，就是并发包里的一个核心组件，里面有 state 变量、加锁线程变量等核心的东西，维护了加锁状态。

你会发现，ReentrantLock 这种东西只是一个外层的 API，内核中的锁机制实现都是依赖 AQS 组件的。

这个 ReentrantLock 之所以用 Reentrant 打头，意思就是他是一个可重入锁。

可重入锁的意思，就是你可以对一个 ReentrantLock 对象多次执行 lock() 加锁和 unlock() 释放锁，也就是可以对一个锁加多次，叫做可重入加锁。

大家看明白了那个 state 变量之后，就知道了如何进行可重入加锁！

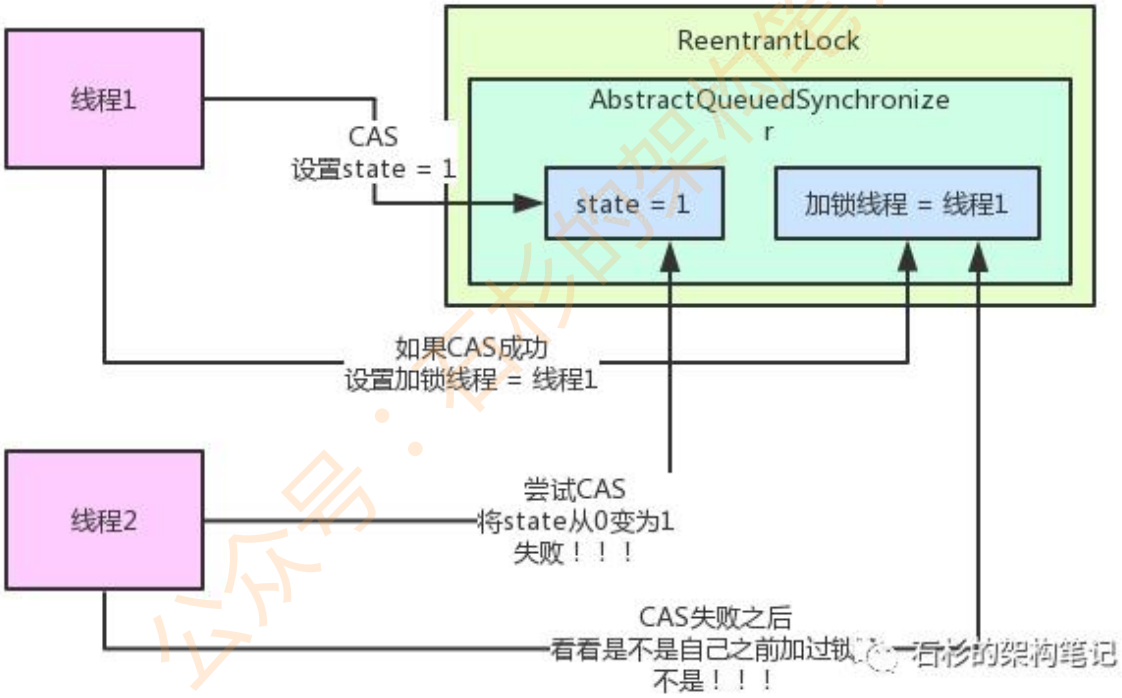
其实每次线程 1 可重入加锁一次，会判断一下当前加锁线程就是自己，那么他自己就可以可重入多次加锁，每次加锁就是把 state 的值给累加 1，别的没啥变化。

接着，如果线程 1 加锁了之后，线程 2 跑过来加锁会怎么样呢?

我们来看看锁的互斥是如何实现的？线程 2 跑过来一下看到，哎呀！state 的值不是 0 啊？所以 CAS 操作将 state 从 0 变为 1 的过程会失败，因为 state 的值当前为 1，说明已经有人加锁了！

接着线程 2 会看一下，是不是自己之前加的锁啊？当然不是了，“加锁线程” 这个变量明确记录了是线程 1 占用了这个锁，所以线程 2 此时就是加锁失败。

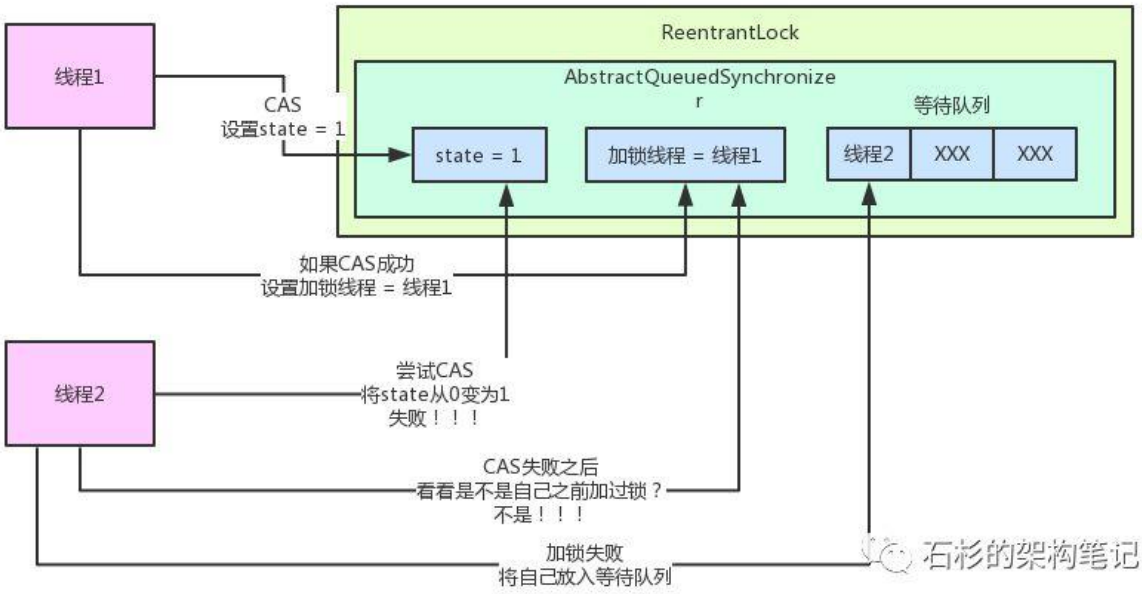
给大家来一张图，一起来感受一下这个过程：



接着，线程 2 会将自己放入 AQS 中的一个等待队列，因为自己尝试加锁失败了，此时就要将自己放入队列中来等待，等待线程 1 释放锁之后，自己就可以重新尝试加锁了

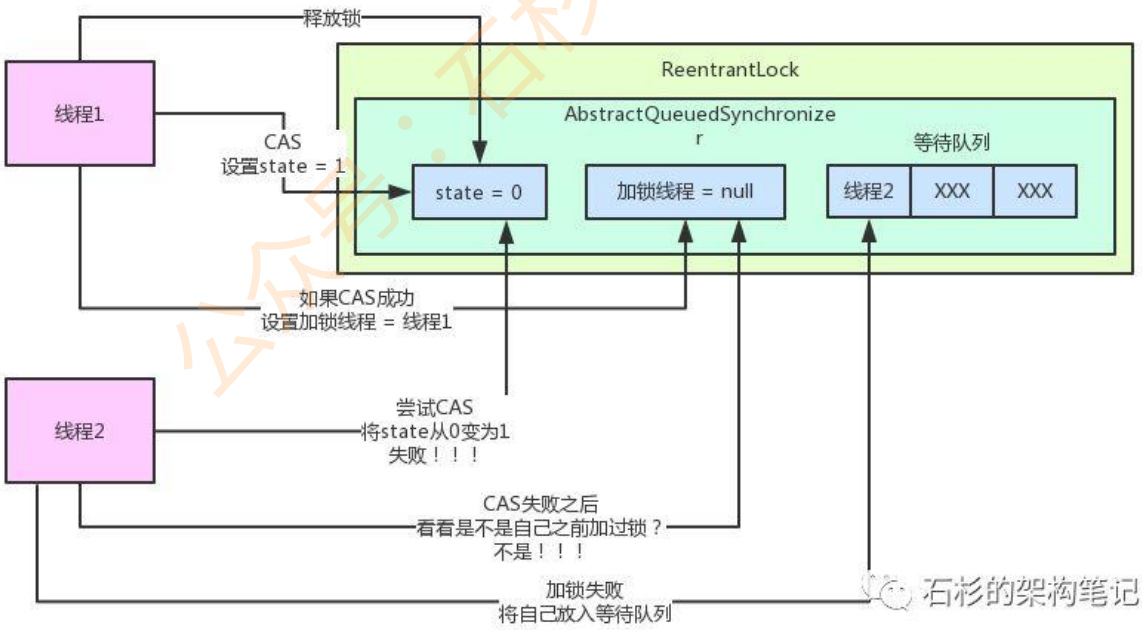
所以大家可以看到，AQS 是如此的核心！AQS 内部还有一个等待队列，专门放那些加锁失败的线程！

同样，给大家来一张图，一起感受一下：



接着，线程 1 在执行完自己的业务逻辑代码之后，就会释放锁！他释放锁的过程非常的简单，就是将 AQS 内的 state 变量的值递减 1，如果 state 值为 0，则彻底释放锁，会将 “加锁线程” 变量也设置为 null！

整个过程，参见下图：

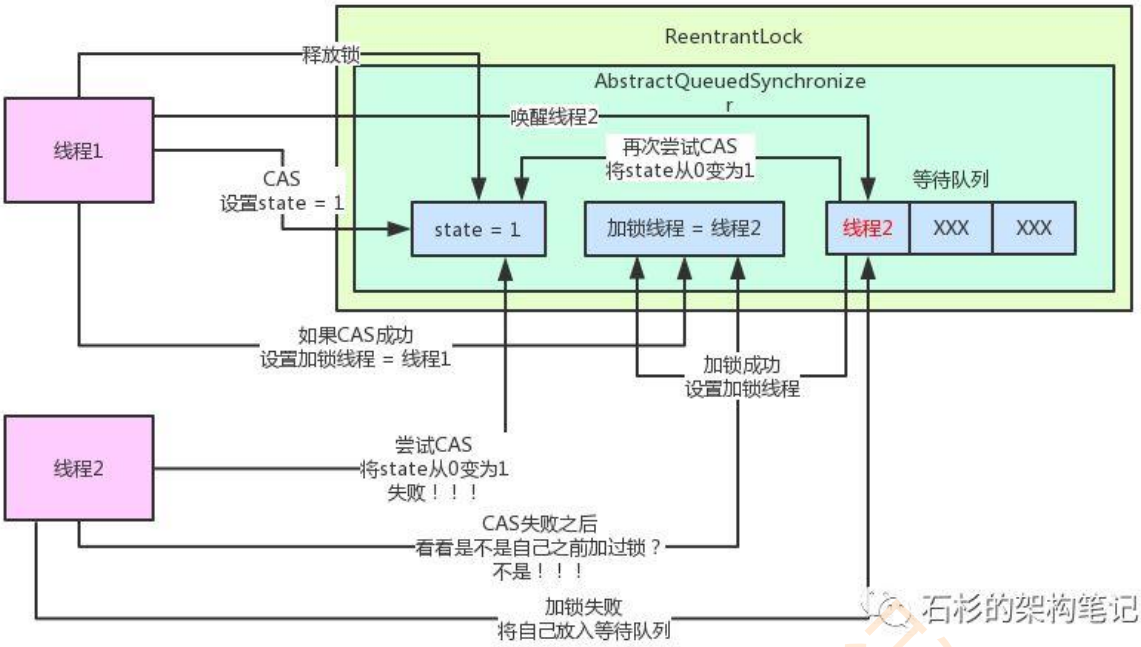


接下来，会从等待队列的队头唤醒线程 2 重新尝试加锁。

好！线程 2 现在就重新尝试加锁，这时还是用 CAS 操作将 state 从 0 变为 1，此时就会成功，成功之后代表加锁成功，就会将 state 设置为 1。

此外，还要把 “加锁线程” 设置为线程 2 自己，同时线程 2 自己就从等待队列中出队了。

最后再来一张图，大家来看看这个过程。



四、总结

OK，本文到这里为止，基本借着 ReentrantLock 的加锁和释放锁的过程，给大家讲清楚了其底层依赖的 AQS 的核心原理。

基本上大家把这篇文章看懂，以后再也不会担心面试的时候被问到：谈谈你对 AQS 的理解这种问题了。

其实一句话总结 AQS 就是一个并发包的基础组件，用来实现各种锁，各种同步组件的。它包含了 state 变量、加锁线程、等待队列等并发中的核心组件。

Java并发面试现场：聊聊公平锁与非公平锁分别是啥?

> **作者:中华石杉** 原文地址

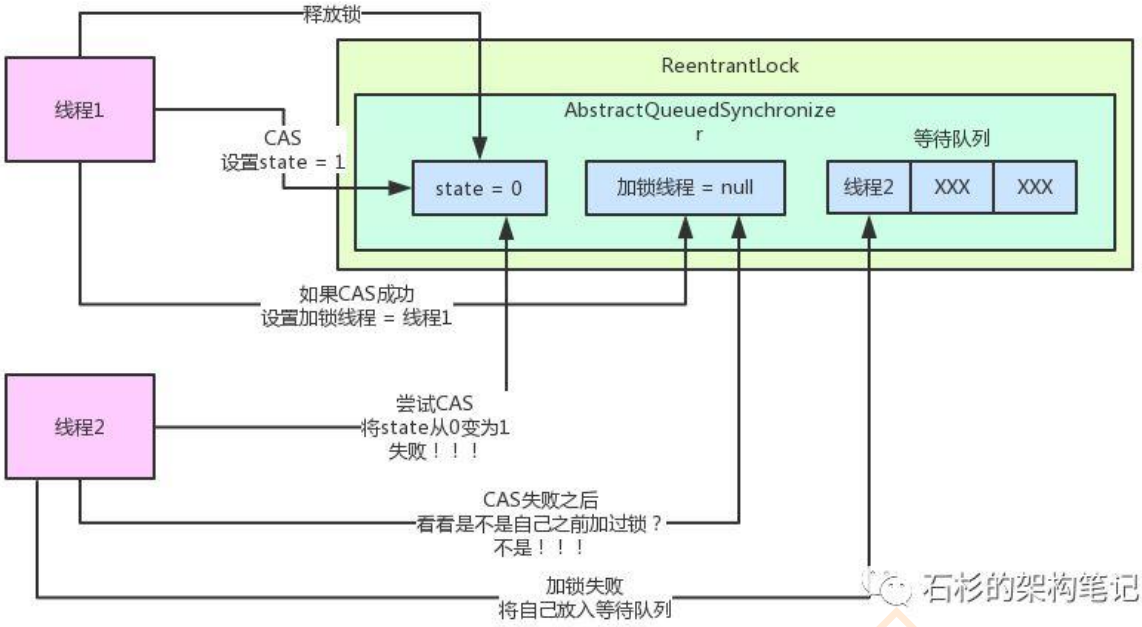
一、写在前面

上篇文章 **《大白话聊聊 Java 并发面试问题之谈谈你对 AQS 的理解？》** 聊了一下 java 并发包中的 AQS 的工作原理，也间接说明了 ReentrantLock 的工作原理。

这篇文章接着来聊一个话题，java 并发包中的公平锁与非公平锁有啥区别?

二、什么是非公平锁?

先来聊聊非公平锁是啥，现在大家先回过头来看下面这张图。



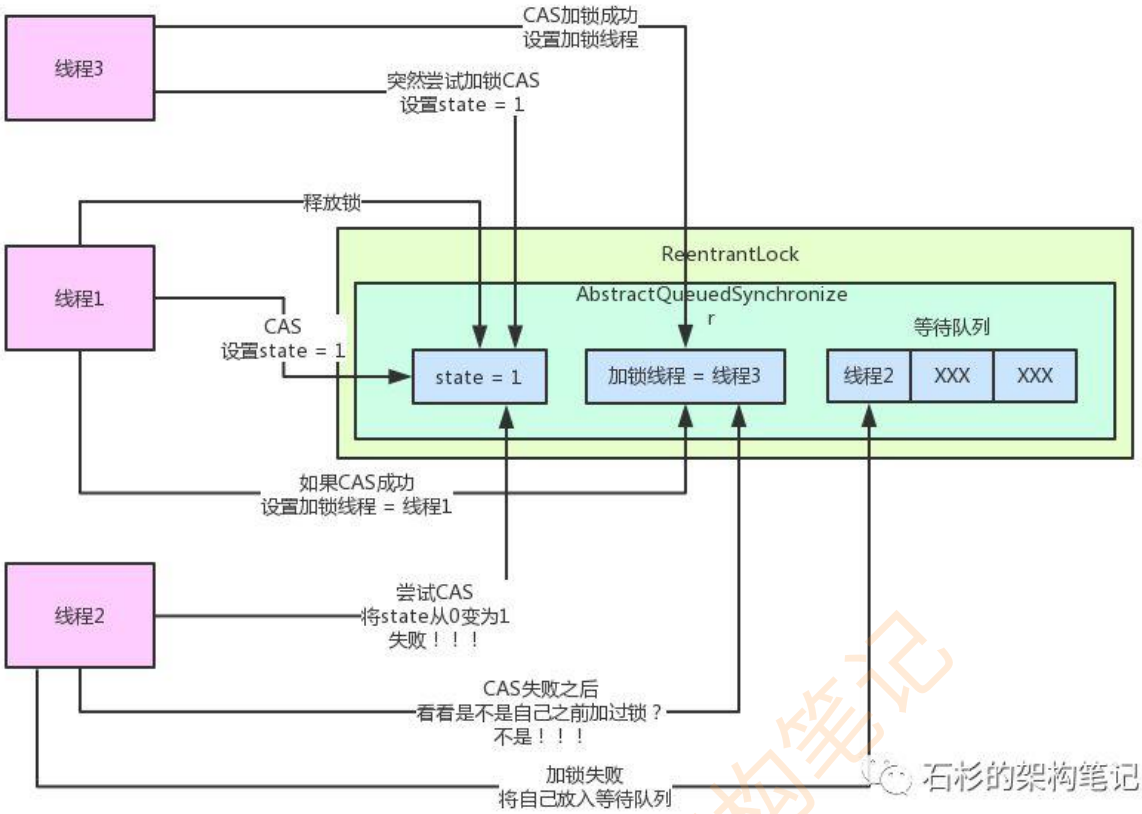
如上图，现在线程 1 加了锁，然后线程 2 尝试加锁，失败后进入了等待队列，处于阻塞中。然后线程 1 释放了锁，准备来唤醒线程 2 重新尝试加锁。

注意一点，此时线程 2 可还停留在等待队列里啊，还没开始尝试重新加锁呢！

然而，不幸的事情发生了，这时半路杀出个程咬金，来了一个线程 3！线程 3 突然尝试对 ReentrantLock 发起加锁操作，此时会发生什么事情?

很简单！线程 2 还没来得及重新尝试加锁呢。也就是说，还没来得及尝试重新执行 CAS 操作将 state 的值从 0 变为 1 呢！线程 3 冲上来直接一个 CAS 操作，尝试将 state 的值从 0 变为 1，结果还成功了！

一旦 CAS 操作成功，线程 3 就会将 “加锁线程” 这个变量设置为他自己。给大家来一张图，看看这整个过程：



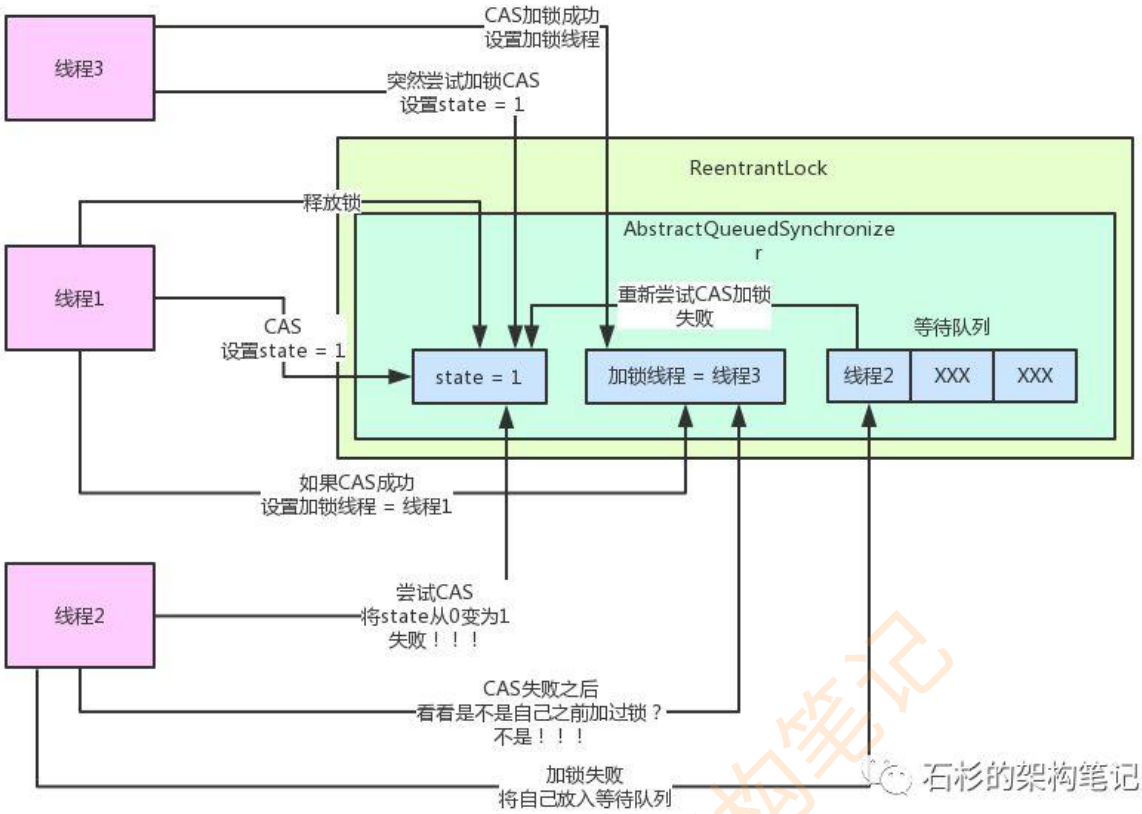
明明人家线程 2 规规矩矩的排队领锁呢，结果你线程 3 不守规矩，线程 1 刚释放锁，不分青红皂白，直接就跑过来抢先加锁了。

这就导致线程 2 被唤醒过后，重新尝试加锁执行 CAS 操作，结果毫无疑问，失败！

原因很简单啊！因为加锁 CAS 操作，是要尝试将 state 从 0 变为 1，结果此时 state 已经是 1 了，所以 CAS 操作一定会失败！

一旦加锁失败，就会导致线程 2 继续留在等待队列里不断的等着，等着线程 3 释放锁之后，再来唤醒自己，真是可怜！先来的线程 2 居然加不到锁！

同样给大家来一张图，体会一下线程 2 这无助的过程：



上述的锁策略，就是所谓的非公平锁！

如果你用默认的构造函数来创建 ReentrantLock 对象，默认的锁策略就是非公平的。

在非公平锁策略之下，不一定说先来排队的线程就就先会得到机会加锁，而是出现各种线程随意抢占的情况。

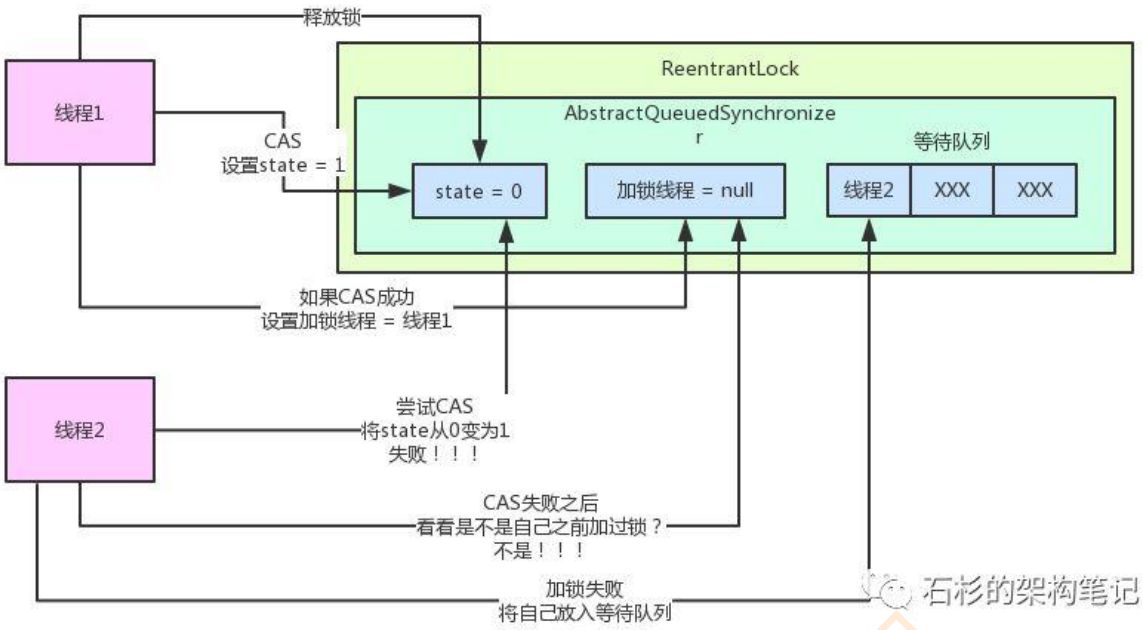
那如果要实现公平锁的策略该怎么办呢？也很简单，在构造 ReentrantLock 对象的时候传入一个 true 即可：

ReentrantLock lock = new ReentrantLock(true)

此时就是说让他使用公平锁的策略，那么公平锁具体是什么意思呢？

三、什么是公平锁？

咱们重新回到第一张图，就是线程 1 刚刚释放锁之后，线程 2 还没来得及重新加锁的那个状态。



同样，这时假设来了一个线程 3，突然杀出来，想要加锁。

如果是公平锁的策略，那么此时线程 3 不会跟个愣头青一样盲目的直接加锁。

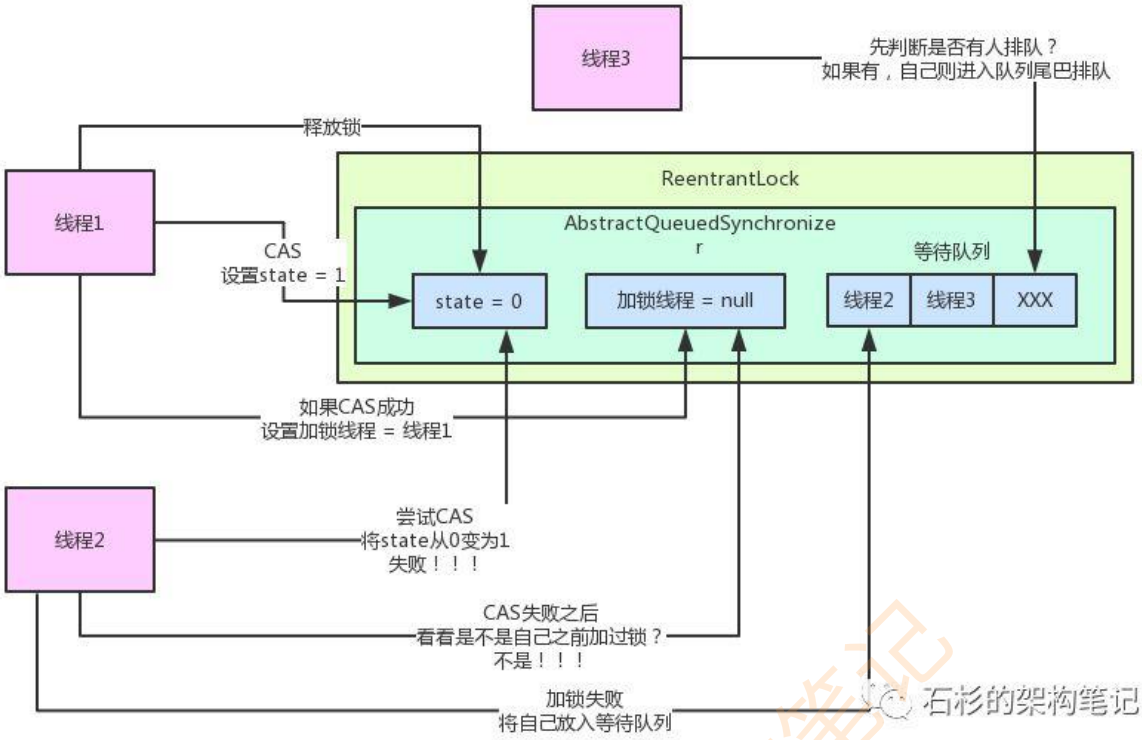
他会先判断一下：咦？AQS 的等待队列里，有没有人在排队啊？如果有人在排队的话，说明我前面有兄弟正想要加锁啊！

如果 AQS 的队列里真的有线程排着队，那我线程 3 就不能跟个二愣子一样直接抢占加锁了。

因为现在咱们是公平策略，得按照先来后到的顺序依次排队，谁先入队，谁就先从队列里出来加锁！

所以，线程 3 此时一判断，发现队列里有人排队，自己就会乖乖的排到队列后面去，而不会贸然加锁！

同样，整个过程我们用下面这张图给大家直观的展示一下：

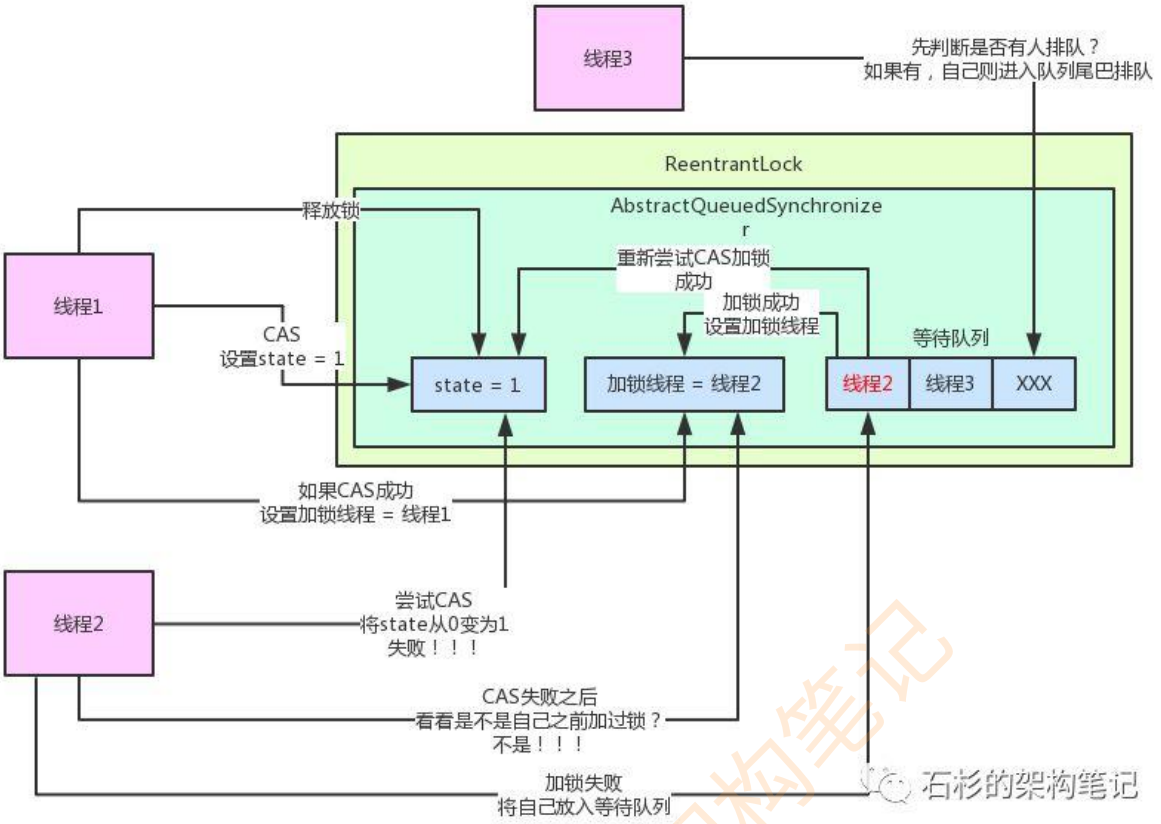


上面的等待队列中，线程 3 会按照公平原则直接进入队列尾部进行排队。

接着，线程 2 不是被唤醒了么？他就会重新尝试进行 CAS 加锁，此时没人跟他抢，他当然可以加锁成功了。

然后呢，线程 2 就会将 state 值变为 1，同时设置 “加锁线程” 是自己。最后，线程 2 自己从等待队列里出队。

整个过程，参见下图：



这个就是公平锁的策略，过来加锁的线程全部是按照先来后到的顺序，依次进入等待队列中排队的，不会盲目的胡乱抢占加锁，非常的公平。

四、小结

好了，通过画图和文字分析，相信大家都明白什么是公平锁，什么是非公平锁了！

不过要知道 java 并发包里很多锁默认的策略都是非公平的，也就是可能后来的线程先加锁，先来的线程后加锁。

而一般情况下，非公平的策略都没什么大问题，但是大家要对这个策略做到心里有数，在开发的时候，需要自己来考虑和权衡是要用公平策略还是非公平策略。

大白话聊聊Java并发面试问题之微服务注册中心的读写锁优化

> 作者:中华石杉 原文地址

一、读写锁的介绍

上一篇文章 大白话聊聊 Java 并发面试问题之公平锁与非公平锁是啥?，聊了一下 java 并发包的公平锁和非公平锁。

这篇文章来聊一下读写锁。所谓的读写锁，就是将一个锁拆分为读锁和写锁两个锁，然后你加锁的时候，可以加写锁，也可以加读锁。如下面代码所示：

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

// 加写锁
lock.writeLock().lock();
lock.writeLock().unlock();

// 加读锁
lock.readLock().lock();
lock.readLock().unlock();
```


如果有一个线程加了写锁，那么其他线程就不能加写锁了，同一时间只能允许一个线程加写锁。因为加了写锁就意味着有人要写一个共享数据，那同时就不能让其他人来写这个数据了。

同时如果有线程加了写锁，其他线程就不能加读锁了，因为既然都有人在写数据了，你其他人当然不能来读数据了！

如果有一个线程加了读锁，别的线程是可以随意同时加读锁的，因为只是有线程在读数据而已，此时别的线程也是可以来读数据的！

同理，如果一个线程加了读锁，此时其他线程是不可以加写锁的，因为既然有人在读数据，那就不能让你随意来写数据了！

好了！这个就是初步介绍一下读写锁的使用方法，相信很多同学应该之前都知道了，因为这个是 java 开发中非常基础的一块知识。

二、微服务注册中心的读写锁优化

现在进入主题，我们主要聊一下微服务注册中心里面的读写锁优化。

为什么要聊一下这个问题呢?

因为如果你出去面试，很可能被问到读写锁的问题，此时你可以自然而然的带出来，你之前了解过 Spring Cloud 微服务技术架构，同时对里面的微服务注册中心的注册表读写锁优化有一些自己的感悟和看法。

这样的话，相比于你简单的给面试官聊聊读写锁的基本概念和使用方法，要增色不少！

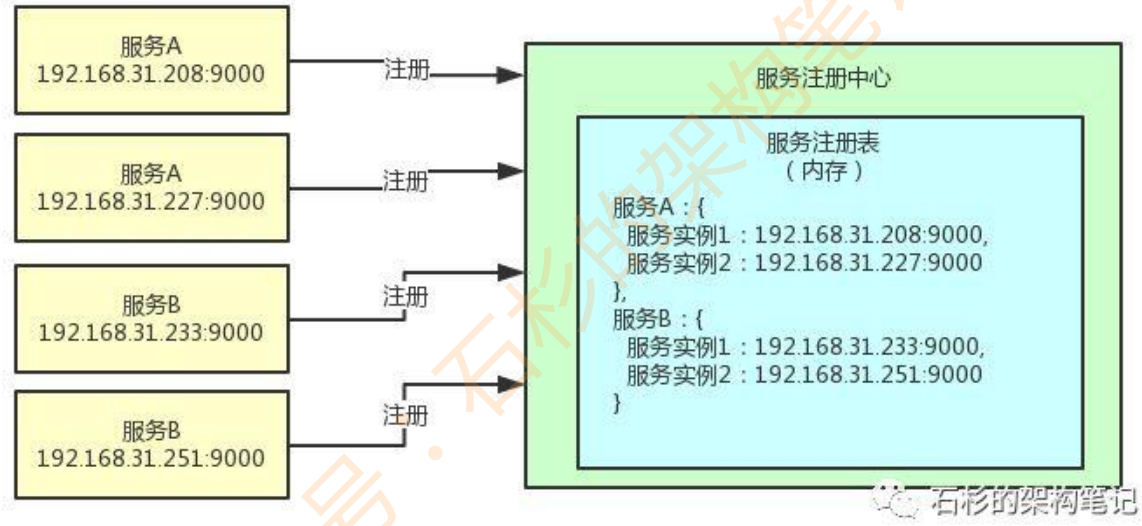
首先，大家需要了解一点微服务的整体架构知识，可以参考之前写过的一篇文章拜托，面试请不要再问我 Spring Cloud 底层原理！。

同时还需要了解一下 Spring Cloud Eureka（即微服务注册中心）的核心原理。这个可以参考之前写过的一篇文章【双 11 狂欢的背后】微服务注册中心是如何承载大型系统千万级访问的。

好，了解了这些前置知识之后，我们正式开始。

先来看看下面的图，现在我们知道一个微服务注册中心（可以是 Eureka 或者 Consul 或者你自己写的一个微服务注册中心），他肯定会在内存中有一个服务注册表的概念。

这个服务注册表中就是存放了各个微服务注册时发送过来的自己的地址信息，里面保存了每个服务有多少个服务实例，每个服务实例部署在哪台机器上监听哪个端口号，主要是这样的一些信息。



OK，那现在问题来了，这个服务注册表的数据，其实是有人读也有人写的。

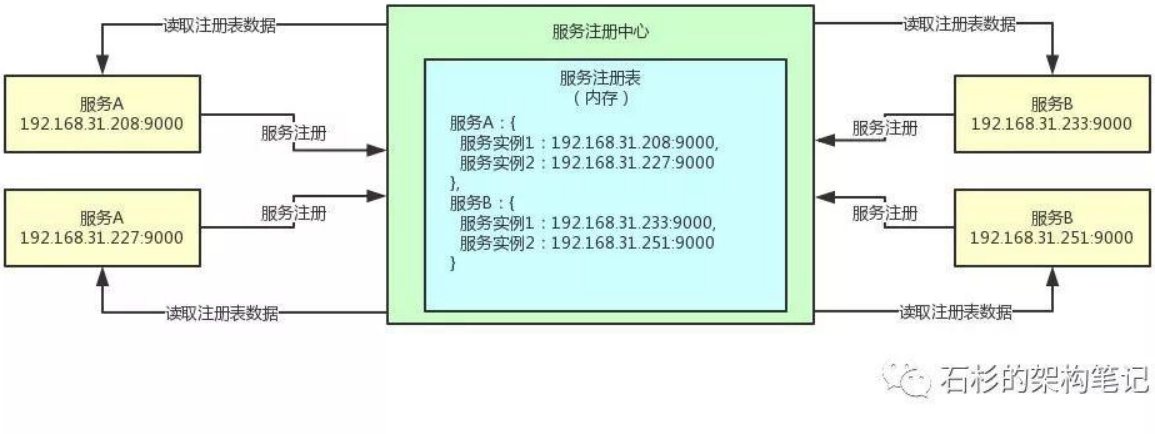
举个例子，比如有的服务启动的时候会来注册，此时就会修改服务注册表的数据，这个就是写的过程。

接着，别的服务也会来读这个服务注册表的数据，因为每个服务都需要感知到其他服务在哪些机器上部署。

所以，这个内存里的服务注册表数据，天然就是有读写并发问题的！可能会有多个线程来写，也可能会有多个线程来读！

如果你对同一份内存中的注册表数据不加任何保护措施，那么可能会有多线程并发修改共享数据的问题，可能导致数据错乱，对吧？

上述过程，大家看看下面的图，就明白了。



此时，如果对服务注册表的服务注册和读取服务注册表的方法，都加一个 synchronized 关键字，是不是就可以了呢?

或许你会想，加上 synchronized，直接让所有线程对服务注册表的读写操作，全部串行化。那不就可以保证内存中的服务注册表数据安全了吗?

下面是一段伪代码，大家来感受一下：

```
// 服务注册表
public class ServiceRegistry {

    // 这是内存中的核心的服务注册表数据
    // 采用Map数据结构来存放，这里存放了每个服务有哪些服务实例
    // 每个服务实例在哪台机器上，监听哪个端口号
    private Map<String, Map<String, InstanceInfo>> registry = ....

    // 服务注册
    public synchronized void register() {
        // 将服务实例信息加入内存的Map数据结构中
    }

    // 读取服务注册表
    public synchronized Map<String, Map<String, InstanceInfo>> getRegistry() {
        // 返回服务注册表数据
    }
}
```


在上面的代码中直接给写（服务注册）和读（读取服务注册表）两个方法，都暴力的加上了 synchronized 关键字，确实是可以保证服务注册表的数据不错乱，但是这样肯定是不太合适的。

因为这么搞的话，相当于是所有的线程读写服务注册表数据，全部串行化了。

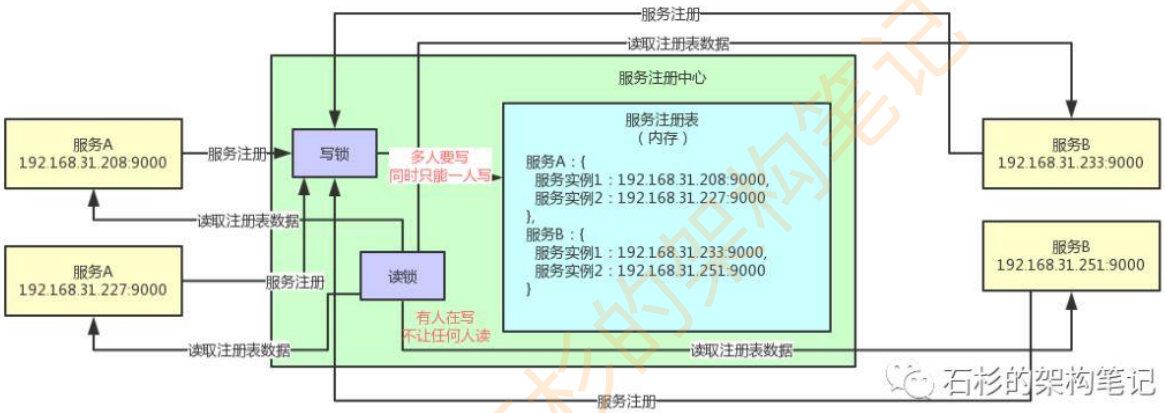
大家思考一下，我们想要的效果是什么？其实不就是在有人往服务注册表里写数据的时候，就不让其他人写了，同时也不让其他人读！

然后，有人在读服务注册表的数据的时候，其他人都可以随便同时读，但是此时不允许别人写服务注册表数据了！

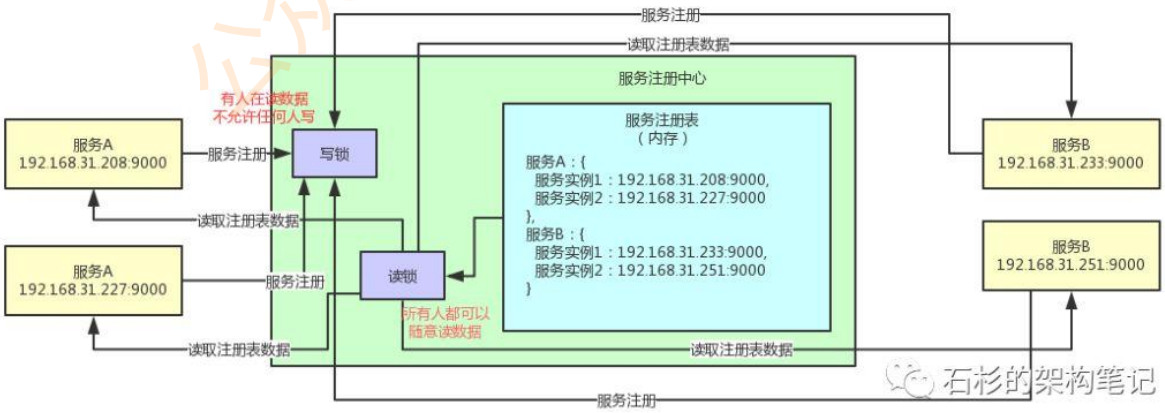
对吧，我们想要的，其实不就是这个效果吗？

想清楚了这点，我们就不应该暴力的加一个 synchronized，让所有读写线程全部串行化，那样会导致并发性非常的低。

大家看看下面的图，我们想要的第一个效果：一旦有人在写服务注册表数据，我们加个写锁，此时别人不能写，也不能读。



那么如果有人在读数据呢？此时就可以让别人都可以读，但是不允许任何人写。大家看下面的图。



关键点来了，这样做有什么好处呢？其实大部分时候都是读操作，所以使用读锁可以让大量的线程同时来读数据，不需要阻塞不需要排队，保证高并发读的性能是比较高的。

然后少量的时候是有服务上线要注册数据，写数据的场景是比较少的，此时写数据的时候，只能一个一个的加写锁然后写数据，同时写数据的时候就不允许别人来读数据了。

所以读写锁是非常适合这种读多写少的场景的。

另外，我们能不能尽量在写数据的期间还保证可以继续读数据呢？大量加读锁的时候，会阻塞人家写数据加写锁过长时间，这种情况能否避免呢？

可以的，采用多级缓存的机制，具体可以参加之前的一篇文章：**【双 11 狂欢的背后】微服务注册中心是如何承载大型系统千万级访问的》**、。里面分析了 Spring Cloud Eureka 微服务注册中心里的多级缓存机制。

最后看下上面那段伪代码如果用读写锁来优化是怎么样的？

```
// 服务注册表
public class ServiceRegistry {

    // 这是内存中的核心的服务注册表数据
    // 采用Map数据结构来存放，这里存放了每个服务有哪些服务实例
    // 每个服务实例在哪台机器上，监听哪个端口号
    private Map<String, Map<String, InstanceInfo>> registry = ....

    // 针对注册表数据准备的读写锁
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private WriteLock writeLock = lock.writeLock();
    private ReadLock readLock = lock.readLock();

    // 服务注册
    public /** synchronized */ void register() {
        writeLock.lock();
        // 将服务实例信息加入内存的Map数据结构中
        writeLock.unlock();
    }

    // 读取服务注册表
    public /** synchronized */ Map<String, Map<String, InstanceInfo>> getRegistry() {
        readLock.lock();
        // 返回服务注册表数据
        readLock.unlock();
    }
}
```


10倍请求压力来袭，你的系统会被击垮吗？

> **作者:中华石杉** 原文地址

> ！“ 上篇文章《记一次 JVM FullGC 导致线上生产系统宕机的稳定性优化》，给大家讲了一个线上系统因为 JVM FullGC 异常宕机的 case。

这篇文章，我们继续给大家聊聊另外一个线上系统在生产环境遇到的问题。

一、背景介绍

背景情况是这样：线上一个系统，在某次高峰期间 MQ 中间件故障的情况下，触发了降级机制，结果降级机制触发之后运行了一小会儿，突然系统就完全卡死，无法响应任何请求。

给大家简单介绍一下这个系统的整体架构，这个系统简单来说就是有一个非常核心的行为，就是往 MQ 里写入数据，但是这个往 MQ 里写入的数据是非常核心及关键的，绝对不容许有丢失。

所以最初就设计了一个降级机制，如果一旦 MQ 中间件故障，那么这个系统立马就会把核心数据写入本地磁盘文件。

额外提一句，如果有同学不太清楚 MQ 中间件的概念，建议看一下之前发的一篇文章《哥们，你们的系统架构中为什么要引入消息中间件？》，先对 MQ 中间件这个东西做一个基本的了解。

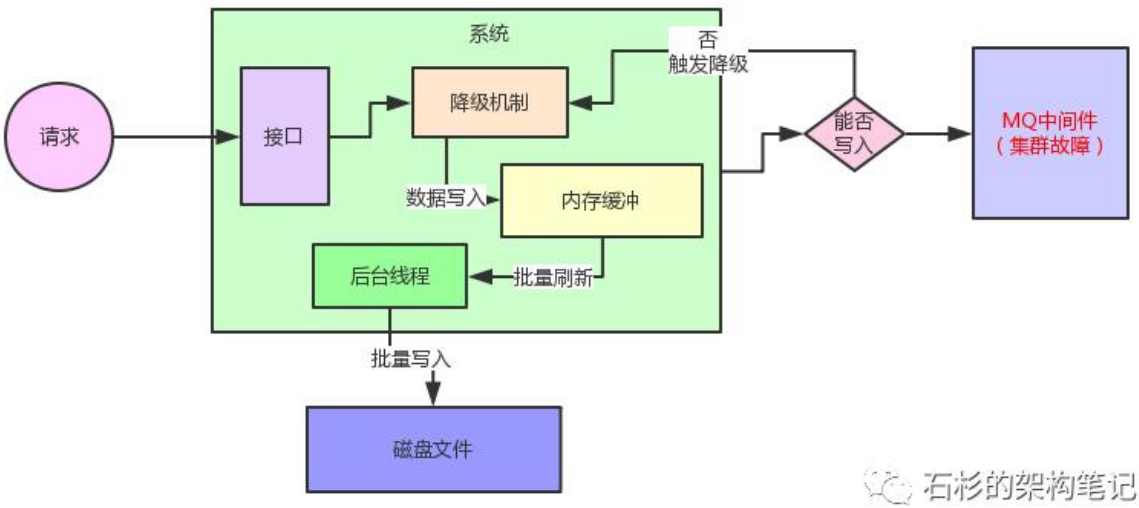
但是如果说在高峰期并发量比较高的情况下，接收到一条数据立马同步写本地磁盘文件，这个性能绝对是极其差的，会导致系统自身的吞吐量瞬间大幅度下降，这个降级机制是绝对无法在生产环境运行的，因为自己就会被高并发请求压垮。

因此当时设计的时候，对降级机制进行了一番精心的设计。

我们的核心思路是一旦 MQ 中间件故障，触发降级机制之后，系统接收到一条请求不是立马写本地磁盘，而是采用内存双缓冲 + 批量刷磁盘的机制。

简单来说，系统接收到一条消息就会立马写内存缓冲，然后开启一个后台线程把内存缓冲的数据刷新到磁盘上去。

整个过程，大家看看下面的图，就知道了。

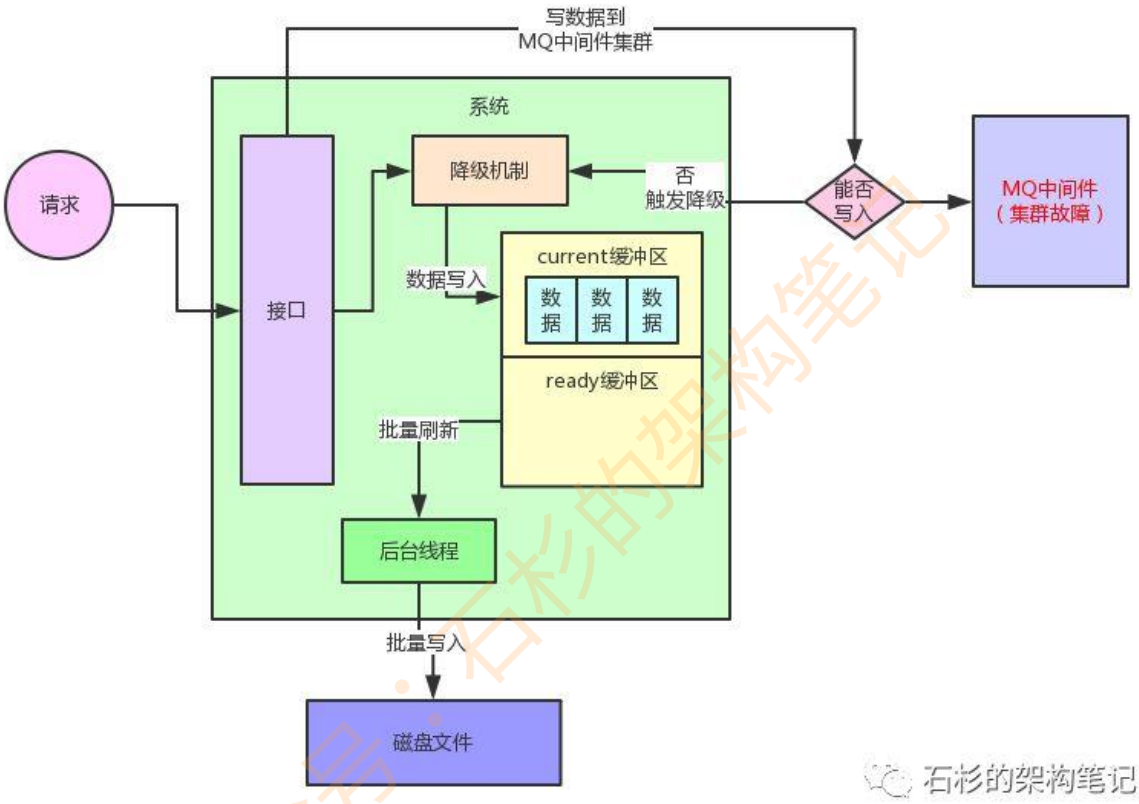


这个内存缓冲实际在设计的时候，分为了两个区域。

一个是 current 区域，用来供系统写入数据，另外一个是 ready 区域，用来供后台线程刷新数据到磁盘里去。

每一块内存区域设置的缓冲大小是 512kb，系统接收到请求就写 current 缓冲区，但是 current 缓冲区总共就 512kb 的内存空间，因此一定会写满。

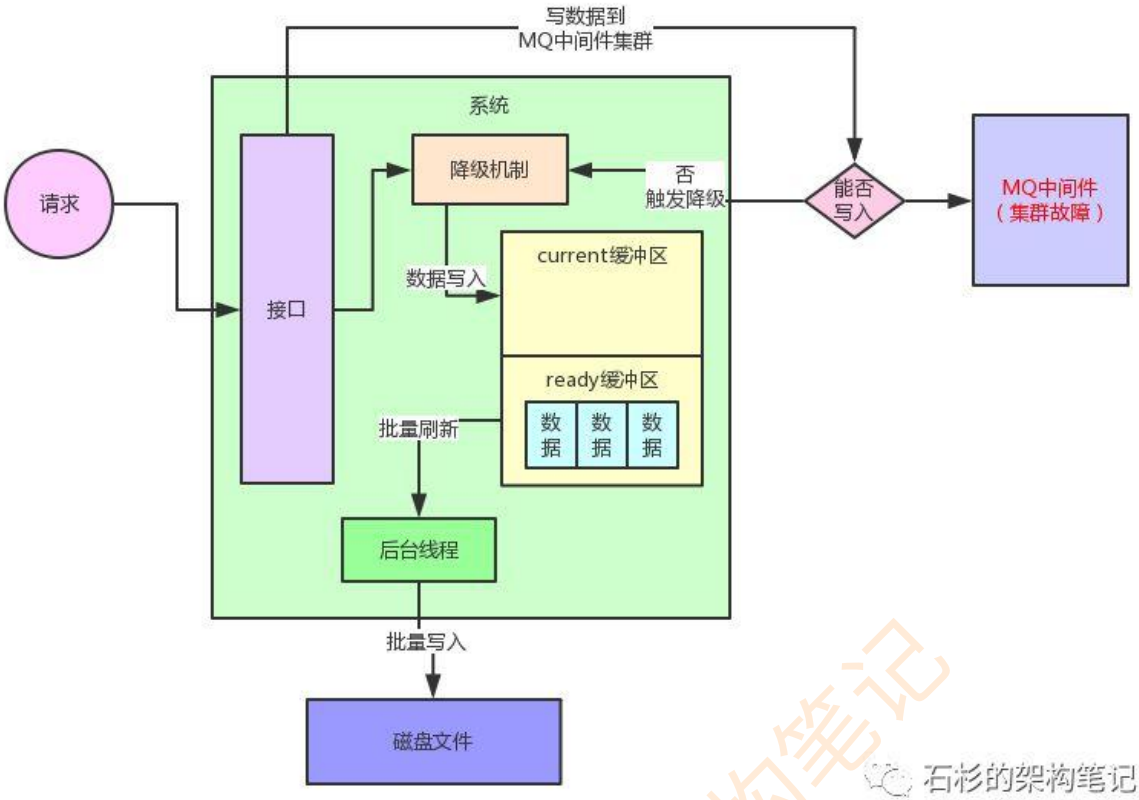
同样，大家结合下面的图，一起来看看。



current 缓冲区写满之后，就会交换 current 缓冲区和 ready 缓冲区。交换过后，ready 缓冲区承载了之前写满的 512kb 的数据。

然后 current 缓冲区此时是空的，可以继续接着系统继续将新来的数据写入交换后的新的 current 缓冲区。

整个过程如下图所示：



此时，后台线程就可以将 ready 缓冲区中的数据通过 Java NIO 的 API，直接高性能 append 方式的写入到本地磁盘文件里。

当然，这里后台线程会有一整套完善的机制，比如说一个磁盘文件有固定大小，如果达到了一定大小，自动开启一个新的磁盘文件来写入数据。

二、埋下隐患

好！通过上面一套机制，即使是高峰期，也能顺利的抗住高并发的请求，一切看起来都很美好！

但是，当时这个降级机制在开发时，我们采取的思路，为后面埋下了隐患！

当时采取的思路是：如果 current 缓冲区写满了之后，所有的线程全部陷入一个 while 循环无限等待。

等到什么时候呢？一直需要等到 ready 缓冲区的数据被刷到磁盘文件之后，清空掉 ready 缓冲区，然后跟 current 缓冲区进行交换。

这样 current 缓冲区要再次变为空的缓冲区，才可以让工作线程继续写入数据。

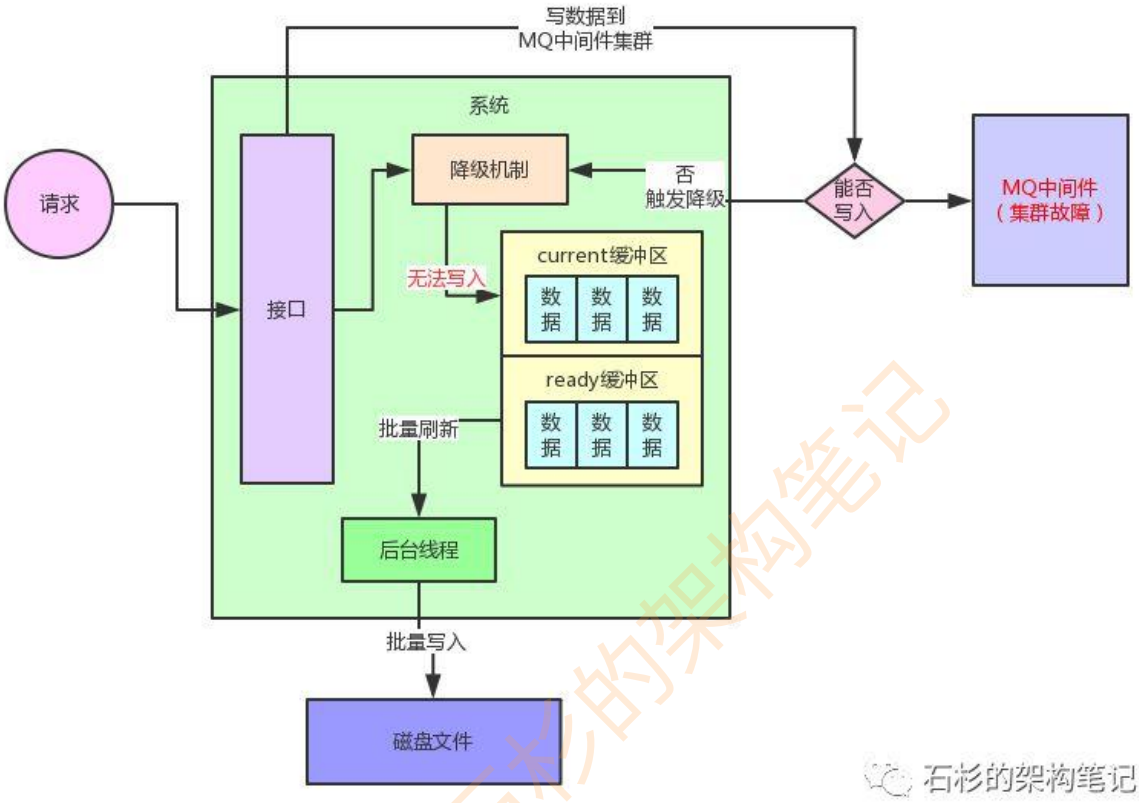
但是大家有没有考虑过一个异常的情况有可能会发生？

就是后台线程刷新 ready 缓冲区的数据到磁盘文件，实际上也是需要一点时间的。

万一在他刷新数据到磁盘文件的过程中，current 缓冲区突然也被写满了呢?

此时就会导致系统的所有工作线程无法写入 current 缓冲区，线程全部卡死。

给大家上一张图，看看这个问题!



这个就是系统的降级机制的双缓冲机制最根本的问题了，在开发好这套降级机制之后，采用正常的请求压力测试过，发现两块缓冲区在设置为 512kb 的情况下，运作良好，没有什么问题。

三、高峰请求，问题爆发

但是问题就出在高峰期上了。某一次高峰期，系统请求压力达到了平时的 10 倍以上。

当然正常流程下，高峰期的时候，写请求其实也是直接全部写到 MQ 中间件集群去的，所以哪怕你高峰期流量增加 10 倍也无所谓，MQ 集群是可以天然抗高并发的。

但是当时不幸的是，在高峰期的时候，MQ 中间件集群突然临时故障，这也是一年遇不到几次的。

这就导致这个系统突然触发了降级机制，然后就开始写入数据到内存双缓冲里面去。

要知道，此时是高峰期啊，请求量是平时正常的 10 倍！因此 10 倍的请求压力瞬间导致了一个问题的发生。

这个问题就是瞬时涌入的高并发请求一下将 current 缓冲区写满，然后两个缓冲区交换，后台线程开始刷新 ready 缓冲区的数据到磁盘文件里去。

结果因为高峰期请求涌入过快，导致 ready 缓冲区的数据还没来得及刷新到磁盘文件，此时 current 缓冲区又突然写满了。。。

这就尴尬了，线上系统瞬间开始出现异常。。。

典型的表现就是，所有机器上部署的实例全部线程都卡死，处于 wait 的状态。

四、定位问题，对症下药

于是，这套系统开始在高峰期无法响应任何请求。后来经过线上故障紧急排查、定位和抢修，才解决了这个问题。

其实说来解决方法也很简单，我们通过 jvm dump 出来快照进行分析，查看系统的线程具体是卡在哪个环节，然后发现大量线程卡死在等待 current 缓冲区的地方。

这就很明显知道原因了，解决方法就是对线上系统扩容双段缓冲的大小，从 512kb 扩容到一个缓冲区 10mb。

这样在线上高峰期的情况下，也可以稳稳的让降级机制的双缓冲机制流畅的运行，不会说瞬间高峰涌入的请求打满两块缓冲区。

因为缓冲区越大，就可以让 ready 缓冲区被 flush 到磁盘文件的过程中，current 缓冲区没那么快被打满。

但是这个线上故障反馈出来的一个教训，就是对系统设计和开发的任何较为复杂的机制，都必须要参照线上高峰期的最大流量来压力测试。只有这样，才能确保任何在系统上线的复杂机制可以经得起线上高峰期的流量的考验。

高并发场景下，如何保证生产者投递到消息中间件的消息不丢失?

> **作者:中华石杉** 原文地址

目录

- （4）confirm 机制投递消息的高延迟性
- （5）高并发下如何投递消息才能不丢失
- （6）消息中间件全链路 100% 数据不丢失能做到吗?

1 前情提示

上篇文章：**《面试大杀器：消息中间件如何实现消费吞吐量的百倍优化？》**，我们分析了 RabbitMQ 开启手动 ack 机制保证消费端数据不丢失的时候，prefetch 机制对消费者的吞吐量以及内存消耗的影响。

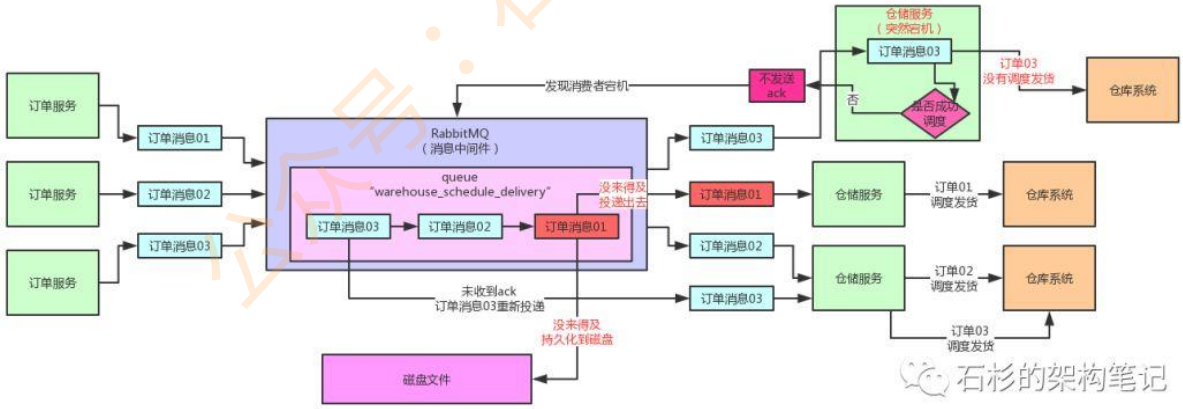
通过分析，我们知道了 prefetch 过大容易导致内存溢出，prefetch 过小又会导致消费吞吐量过低，所以在实际项目中需要慎重测试和设置。

这篇文章，我们转移到消息中间件的生产端，一起来看看如何保证投递到 MQ 的数据不丢失。

如果投递出去的消息在网络传输过程中丢失，或者在 RabbitMQ 的内存中还没写入磁盘的时候宕机，都会导致生产端投递到 MQ 的数据丢失。

而且丢失之后，生产端自己还感知不到，同时还没办法来补救。

下面的图就展示了这个问题。



2 保证投递消息不丢失的 confirm 机制

其实要解决这个问题，相信大家看过之前的消费端 ack 机制之后，也都猜到了。

很简单，就是生产端（比如上图的订单服务）首先需要开启一个 confirm 模式，接着投递到 MQ 的消息，如果 MQ 一旦将消息持久化到磁盘之后，必须也要回传一个 confirm 消息给生产端。

这样的话，如果生产端的服务接收到了这个 confirm 消息，就知道是已经持久化到磁盘了。

否则如果没有接收到 confirm 消息，那么就说明这条消息半路可能丢失了，此时你就可以重新投递消息到 MQ 去，确保消息不要丢失。

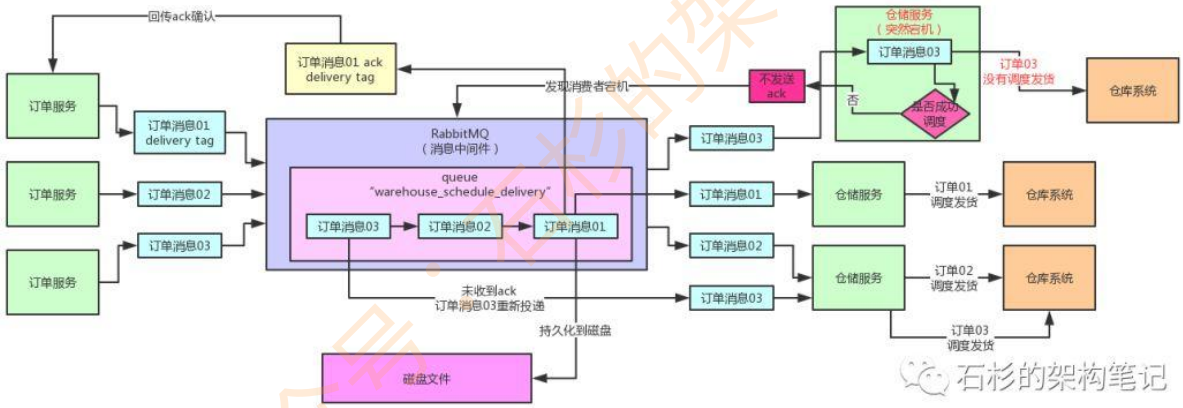
而且一旦你开启了 confirm 模式之后，每次消息投递也同样是有一个 delivery tag 的，也是起到唯一标识一次消息投递的作用。

这样，MQ 回传 ack 给生产端的时候，会带上这个 delivery tag。你就知道具体对应着哪一次消息投递了，可以删除这条消息。

此外，如果 RabbitMQ 接收到一条消息之后，结果内部出错发现无法处理这条消息，那么他会回传一个 nack 消息给生产端。此时你就会感知到这条消息可能处理有问题，你可以选择重新再次投递这条消息到 MQ 去。

或者另一种情况，如果某条消息很长时间都没给你回传 ack/nack，那可能是极端意外情况发生了，数据也丢了，你也可以自己重新投递消息到 MQ 去。

通过这套 confirm 机制，就可以实现生产端投递消息不会丢失的效果。大家来看看下面的图，一起来感受一下。



3 confirm 机制的代码实现

下面，我们再来看看 confirm 机制的代码实现：

```
// 在生产端的代码，首先对channel使用confirmSelect()方法
// 这个方法可以将当前channel设置为confirm模式
channel.confirmSelect();

channel.addConfirmListener(new ConfirmListener() {

    // 这个就是回调方法，专门用于处理MQ回传的ack的，
    // 收到这个ack消息，说明发送的消息被MQ给confirm了，持久化到磁盘了
    public void handleAck(long deliveryTag, boolean multiple) throws IOException {

        // 你可以把发送出去的、但是还没被ack的消息，先保存在内存/数据库/缓存/kv存储
        // 然后收到这个消息的ack之后，你就可以将那个消息删除了
        // deliveryTag 就唯一标识了一个消息投递
        。。。

    }

    public void handleNack(long deliveryTag, boolean multiple) throws IOException {

        // 在这里，如果收到消息的nack通知，那么可以选择重新投递消息
        // 当然，为了避免其他的一些意外，你也可以自己轮询遍历所有未ack的消息
        // 达到一定的超时时间之后，自己主动重新投递消息也是可以的
        。。。
    }
});
```


4 confirm 机制投递消息的高延迟性

这里有一个很关键的点，就是一旦启用了 confirm 机制投递消息到 MQ 之后，MQ 是不保证什么时候会给你一个 ack 或者 nack 的。

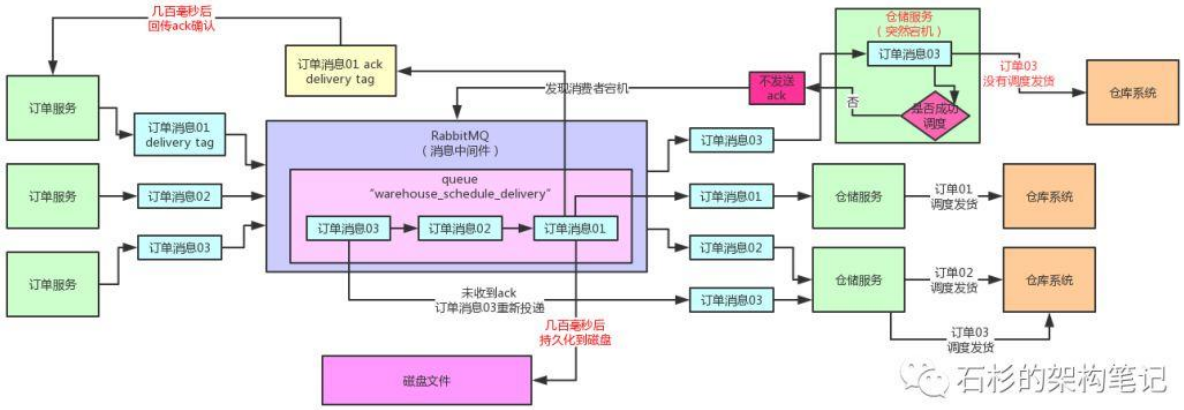
因为 RabbitMQ 自己内部将消息持久化到磁盘，本身就是通过异步批量的方式来进行的。

正常情况下，你投递到 RabbitMQ 的消息都会先驻留在内存里，然后过了几百毫秒的延迟时间之后，再一次性批量把多条消息持久化到磁盘里去。

这样做，是为了兼顾高并发写入的吞吐量和性能的，因为要是你来一条消息就写一次磁盘，那么性能会很差，每次写磁盘都是一次 fsync 强制刷入磁盘的操作，是很耗时的。

所以正是因为这个原因，你打开了 confirm 模式之后，很可能你投递出去一条消息，要间隔几百毫秒之后，MQ 才会把消息写入磁盘，接着你才会收到 MQ 回传过来的 ack 消息，这个就是所谓 confirm 机制投递消息的高延迟性。

大家看看下面的图，一起来感受一下。



5 高并发下如何投递消息才能不丢失

大家可以考虑一下，在生产端高并发写入 MQ 的场景下，你会面临两个问题：

- 1、你每次写一条消息到 MQ，为了等待这条消息的 ack，必须把消息保存到一个存储里。

并且这个存储不建议是内存，因为高并发下消息是很多的，每秒可能都几千甚至上万的消息投递出去，消息的 ack 要等几百毫秒的话，放内存可能有内存溢出的风险。

- 2、绝对不能以同步写消息 + 等待 ack 的方式来投递，那样会导致每次投递一个消息都同步阻塞等待几百毫秒，会导致投递性能和吞吐量大幅度下降。

针对这两个问题，相对应的方案其实也呼之欲出了。

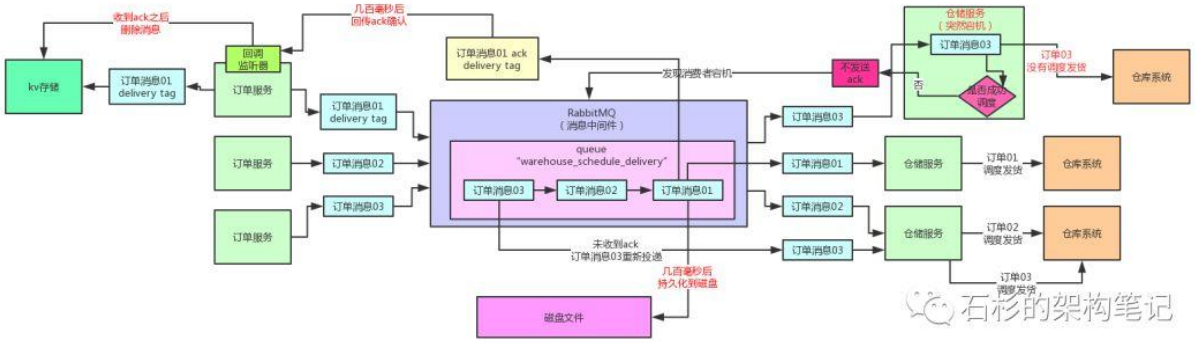
首先，用来临时存放未 ack 消息的存储需要承载高并发写入，而且我们不需要什么复杂的运算操作，这种存储首选绝对不是 MySQL 之类的数据库，而建议采用 kv 存储。kv 存储承载高并发能力极强，而且 kv 操作性能很高。

其次，投递消息之后等待 ack 的过程必须是异步的，也就是类似上面那样的代码，已经给出了一个初步的异步回调的方式。

消息投递出去之后，这个投递的线程其实就可以返回了，至于每个消息的异步回调，是通过在 channel 注册一个 confirm 监听器实现的。

收到一个消息 ack 之后，就从 kv 存储中删除这条临时消息；收到一个消息 nack 之后，就从 kv 存储提取这条消息然后重新投递一次即可；也可以自己对 kv 存储里的消息做监控，如果超过一定时长没收到 ack，就主动重发消息。

大家看看下面的图，一起来体会一下：



6 消息中间件全链路 100% 数据不丢失能做到吗?

到此为止，我们已经把生产端和消费端如何保证消息不丢失的相关技术方案结合 RabbitMQ 这种中间件都给大家分析过了。

其实，架构思想是通用的， 无论你用的是哪一种 MQ 中间件，他们提供的功能是不太一样的，但是你都需要考虑如下几点：

生产端如何保证投递出去的消息不丢失：消息在半路丢失，或者在 MQ 内存中宕机导致丢失，此时你如何基于 MQ 的功能保证消息不要丢失?

MQ 自身如何保证消息不丢失：起码需要让 MQ 对消息是有持久化到磁盘这个机制。

消费端如何保证消费到的消息不丢失：如果你处理到一半消费端宕机，导致消息丢失，此时怎么办?

目前来说，我们初步的借着 RabbitMQ 举例，已经把从前到后一整套技术方案的原理、设计和实现都给大家分析了一遍了。

但是此时真的能做到 100% 数据不丢失吗？恐怕未必，大家再考虑一下个特殊的场景。

生产端投递了消息到 MQ，而且持久化到磁盘并且回传 ack 给生产端了。

但是此时 MQ 还没投递消息给消费端，结果 MQ 部署的机器突然宕机，而且因为未知的原因磁盘损坏了，直接在物理层面导致 MQ 持久化到磁盘的数据找不回来了。

这个大家千万别以为是开玩笑的，大家如果留意留意行业新闻，这种磁盘损坏导致数据丢失的是真的有的。

那么此时即使你把 MQ 重启了，磁盘上的数据也丢失了，数据是不是还是丢失了?

你说，我可以用 MQ 的集群机制啊，给一个数据做多个副本，比如后面我们就会给大家分析 RabbitMQ 的镜像集群机制，确实可以做到数据多副本。

但是即使数据多副本，**一定可以做到 100% 数据不丢失?**

比如说你的机房突然遇到地震，结果机房里的机器全部没了，数据是不是还是全丢了?

说这个，并不是说要抬杠。而是告诉大家，技术这个东西，100% 都是理论上的期望。

应该说，我们凡事都朝着 100% 去做，但是理论上是不可能完全做到 100% 保证的，可能就是做到 99.9999% 的可能性数据不丢失，但是还是有千万分之一的概率会丢失。

当然，从实际的情况来说，能做到这种地步，其实基本上已经基本数据不会丢失了。

从团队自研的百万并发中间件系统的内核设计看 Java 并发性能优化

> **作者:中华石杉** 原文地址

目录

> ! “ 这篇文章，给大家聊聊一个百万级并发的中间件系统的内核代码里的锁性能优化。

很多同学都对 Java 并发编程很感兴趣，学习了很多相关的技术和知识。比如 volatile、Atomic、synchronized 底层、读写锁、AQS、并发包下的集合类、线程池，等等。

1、大部分人对 Java 并发仍停留在理论阶段

很多同学对 Java 并发编程的知识，可能看了很多的书，也通过不少视频课程进行了学习。

但是，大部分人可能还是停留在理论的底层，主要是了解理论，基本对并发相关的技术很少实践和使用，更很少做过复杂的中间件系统。

实际上，真正把这些技术落地到中间件系统开发中去实践的时候，是会遇到大量的问题，需要对并发相关技术的底层有深入的理解和掌握。

然后，结合自己实际的业务场景来进行对应的技术优化、机制优化，才能实现最好的效果。

因此，本文将从笔者曾经带过的一个高并发中间件项目的内核机制出发，来看看一个实际的场景中遇到的并发相关的问题。

同时，我们也将一步步通过对应的伪代码演进，来分析其背后涉及到的并发的性能优化思想和实践，最后来看看优化之后的效果。

2、中间件系统的内核机制：双缓冲机制

这个中间件项目整体就不做阐述了，因为涉及核心项目问题。我们仅仅拿其中涉及到的一个内核机制以及对应的场景来给大家做一下说明。

其实这个例子是大量的开源中间件系统、大数据系统中都有涉及到的一个场景，就是：核心数据写磁盘文件。

比如，大数据领域里的 hadoop、hbase、elasitcsearch，Java 中间件领域里的 redis、mq，这些都会涉及到核心数据写磁盘文件的问题。

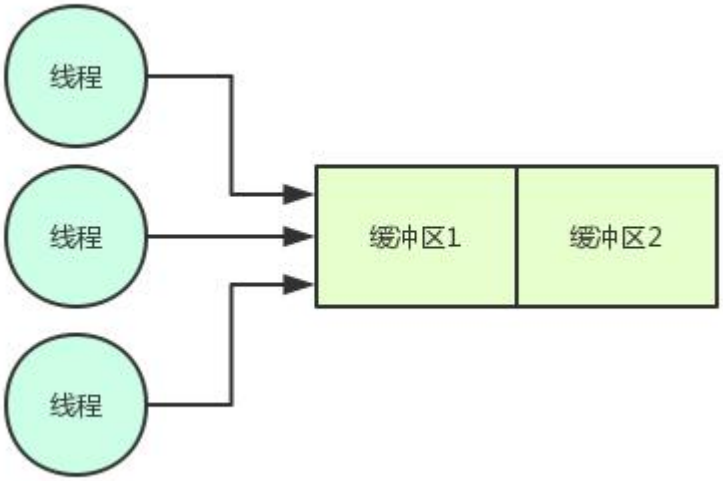
而很多大型互联网公司自研的中年间系统，同样也会有这个场景。只不过不同的中间件系统，他的作用和目标是不一样的，所以在核心数据写磁盘文件的机制设计上，是有一些区别的。

那么我们公司自研的中间件项目，简单来说，需要实现的一个效果是：开辟两块内存空间，也就是经典的内存双缓冲机制。

然后核心数据进来全部写第一块缓冲区，写满了之后，由一个线程进行那块缓冲区的数据批量刷到磁盘文件的工作，其他线程同时可以继续写另外一块缓冲区。

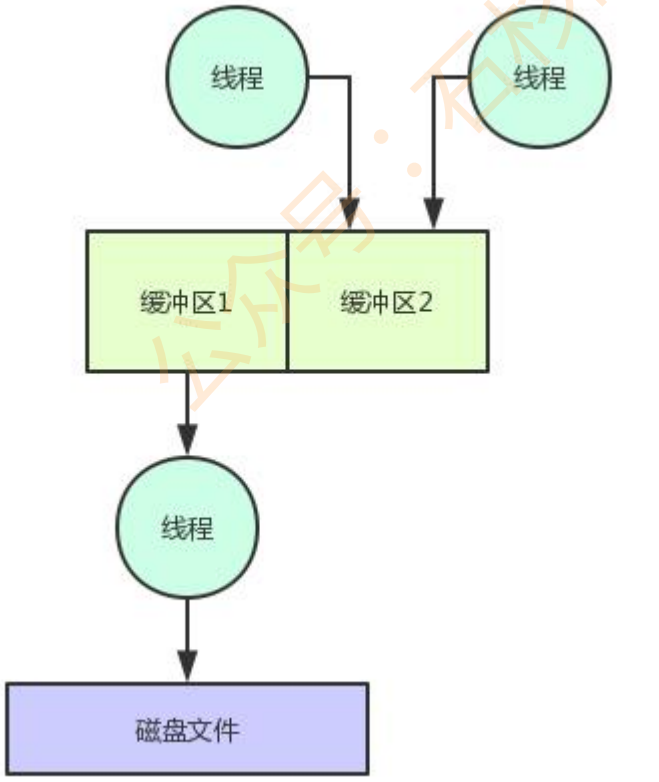
我们想要实现的就是这样的一个效果。这样的话，一块缓冲区刷磁盘的同时，另外一块缓冲区可以接受其他线程的写入，两不耽误。核心数据写入是不会断的，可以持续不断的写入这个中间件系统中。

我们来看看下面的那张图，也来了解一下这个场景。



如上图，首先是很多线程需要写缓冲区 1，然后是缓冲区 1 写满之后，就会由写满的那个线程把缓冲区 1 的数据刷入磁盘文件，其他线程继续写缓冲区 2。

这样，数据批量刷磁盘和持续写内存缓冲，两个事儿就不会耽误了，这是中间件系统设计中极为常用的一个机制，大家看下面的图。



3、百万并发的技术挑战

先给大家说一下这个中间件系统的背景：这是一个服务某个特殊场景下的中间件系统，整体是集群部署。

然后每个实例部署的都是高配置机器，定位是单机承载并发达到万级甚至十万级，整体集群足以支撑百万级并发，因此对单机的写入性能和吞吐要求极为高。

在超高并发的要求之下，上图中的那个内核机制的设计就显得尤为重要了。弄的不好，就容易导致写入并发性能过差，达不到上述的要求。

此外在这里多提一句，类似的这种机制在很多其他的系统里都有涉及。比如之前一篇文章：【高并发优化实践】10 倍请求压力来袭，你的系统会被击垮吗？，那里面讲的一个系统也有类似机制。

只不过不同的是，那篇文章是用这个机制来做 MQ 集群整体故障时的容灾降级机制，跟本文的高并发中间件系统还有点不太一样，所以在设计上考虑的一些细节也是不同的。

而且，之前那篇文章的主题是讲这种内存双缓冲机制的一个线上问题：瞬时超高并发下的系统卡死问题。

4、内存数据写入的锁机制以及串行化问题

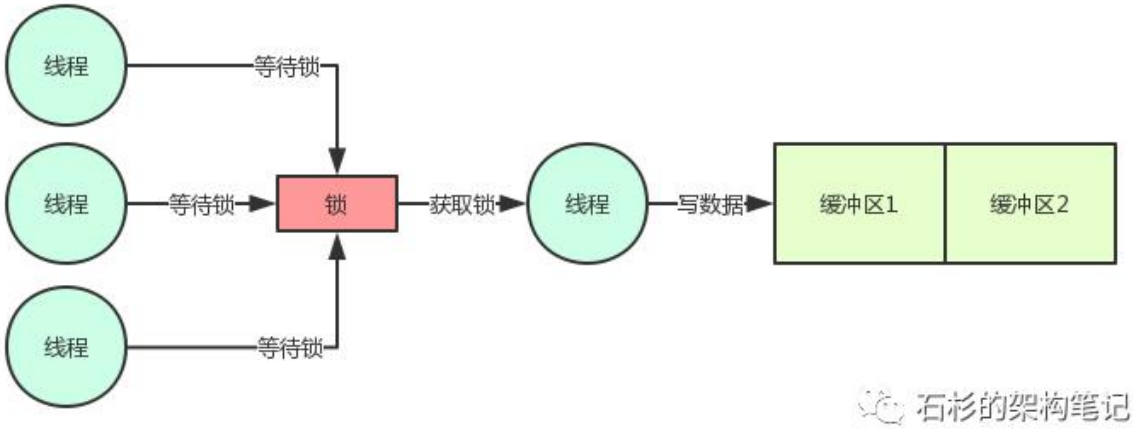
首先我们先考虑第一个问题，你多个线程会并发写同一块内存缓冲，这个肯定有问题啊！

因为内存共享数据并发写入的时候，必须是要加锁的，否则必然会有并发安全问题，导致内存数据错乱。

所以在这里，我们写了下面的伪代码，先考虑一下线程如何写入内存缓冲。

```
// 任何一个线程，想要写内存缓冲，必须要先加锁才可以
synchronized(lock) {

    // 这行代码，意思就是对双段缓冲写入数据，此时会写入缓冲区1中
    doubleBuffer.write(data);
}
```


好了，这行代码弄好之后，对应着下面的这幅图，大家看一下。



看到这里，就遇到了 Java 并发的第一个性能问题了，你要知道高并发场景下，大量线程会并发写内存的，你要是直接这样加一个锁，必然会导致所有线程都是串行化。

即一个线程加锁，写数据，然后释放锁。接着下一个线程干同样的事情。这种串行化必然导致系统整体的并发性能和吞吐量会大幅度降低的。

5、内存缓冲分片机制 + 分段枷锁机制

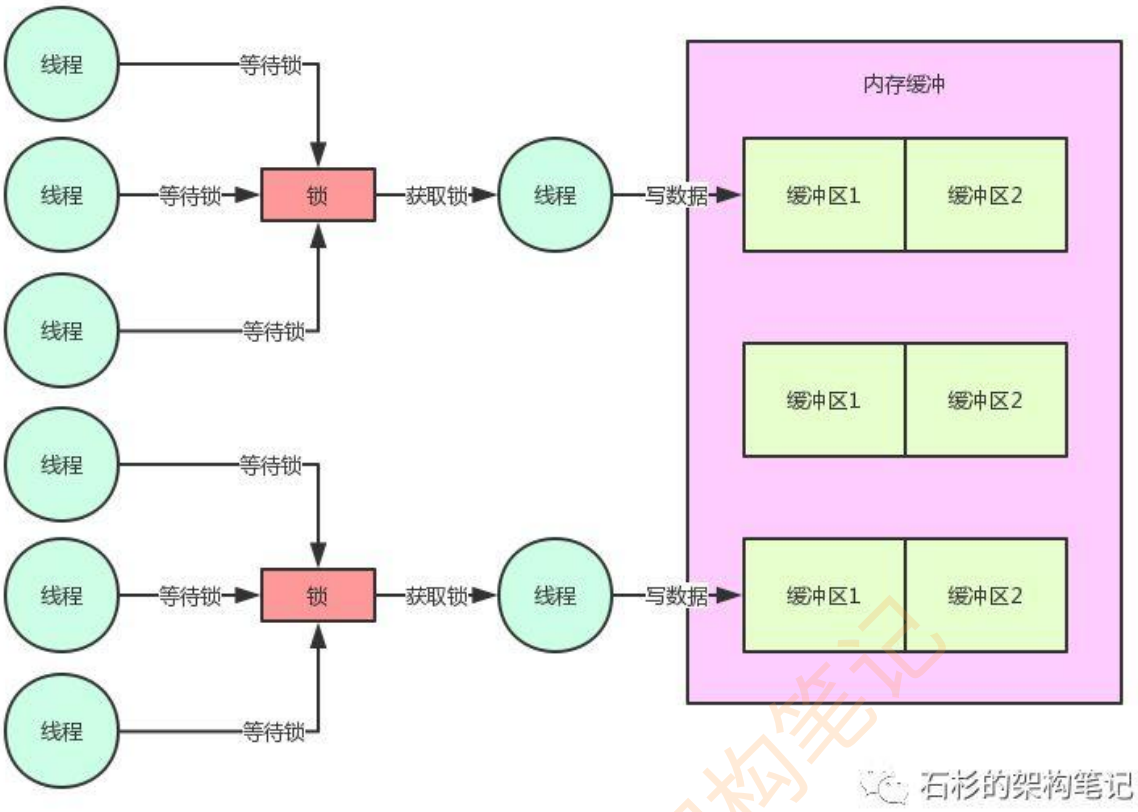
因此在这里必须要对内存双缓冲机制引入分段加锁机制，也就是将内存缓冲切分为多个分片，每个内存缓冲分片就对应一个锁。

这样的话，你完全可以根据自己的系统压测结果，调整内存分片数量，提升锁的数量，进而允许大量线程高并发写入内存。

我们看下面的伪代码，对这块就实现了内存缓冲分片机制：

```
// 任何一个线程，想要写内存缓冲，必须要先加锁才可以
// 但是可以从一个内存缓冲分片集合中随机挑选一个，自动负载均衡
DoubleBuffer doubleBufferSlice = doubleBuffers.chooseRandom();

// 这里加锁，仅仅对一个内存缓冲分片加锁，这样就可以大幅度提升锁的数量
// 同时大幅度降低线程对同一把锁的争用
synchronized(doubleBufferSlice) {

    // 这行代码，意思就是对双段缓冲写入数据，此时会写入缓冲区1中
    doubleBufferSlice.write(data);
}
```


好！我们再来看看，目前为止的图是什么样子的：



这里因为每个线程仅仅就是加锁，写内存，然后释放锁。

所以，每个线程持有锁的时间是很短很短的，单个内存分片的并发写入经过压测，达到每秒几百甚至上千是没问题的，因此线上系统我们是单机开辟几十个到上百个内存缓冲分片的。

经过压测，这足以支撑每秒数万的并发写入，如果将机器资源使用的极限，每秒十万并发也是可以支持的。

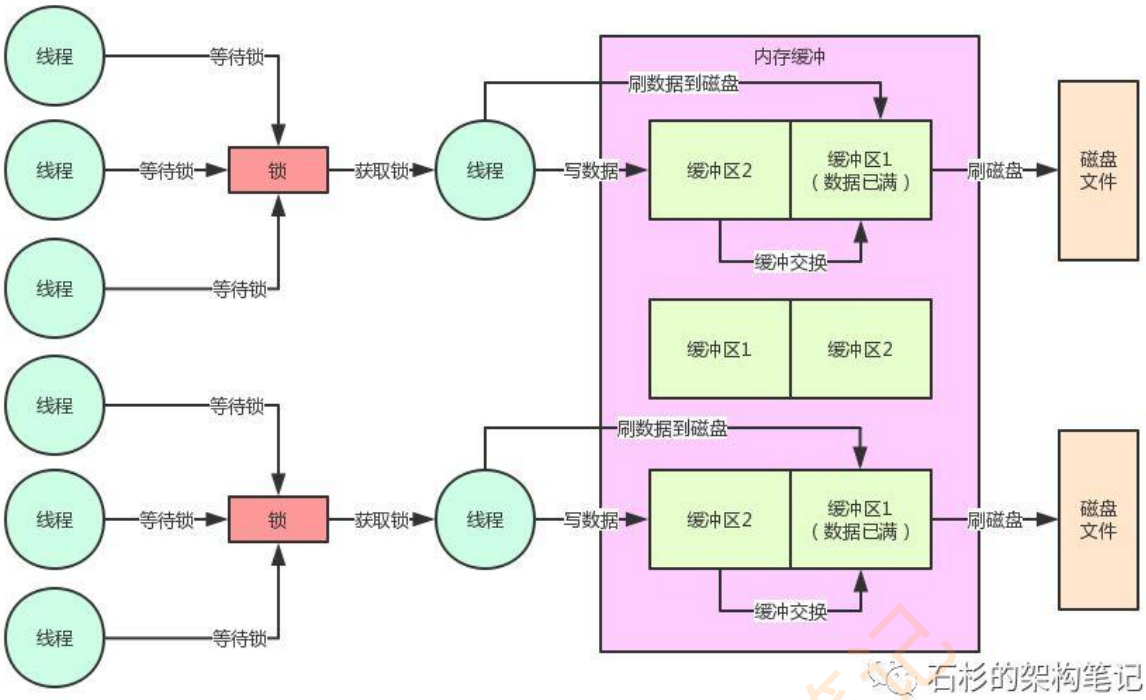
6、缓冲区写满时的双缓冲交换

那么当一块缓冲区写满的时候，是不是就必须要交换两块缓冲区？接着需要有一个线程来将写满的缓冲区数据刷写到磁盘文件中？

此时的伪代码，大家考虑一下，是不是如下所示：

```
// 任何一个线程，想要写内存缓冲，必须要先加锁才可以
// 但是可以从一个内存缓冲分片集合中随机挑选一个，自动负载均衡
DoubleBuffer doubleBufferSlice = doubleBuffers.chooseRandom();

// 这里加锁，仅仅对一个内存缓冲分片加锁，这样就可以大幅度提升锁的数量
// 同时大幅度降低线程对同一把锁的争用
synchronized(doubleBufferSlice) {

    // 这行代码，意思就是对双段缓冲写入数据，此时会写入缓冲区1中
    doubleBufferSlice.write(data);

    // 判断一下假如一块缓冲区是否写满了
    // 每块缓冲区大小都是限定好的，比如就只能放20MB的数据
    // 写到那么多的数据，就是写满了
    if(doubleBufferSlice.isFull()) {

        // 这个方法就是交换一下两块缓冲区，让缓冲区2提供写入服务
        doubleBufferSlice.exhange();

        // 下面的代码就是将缓冲区1的数据刷写到磁盘文件里去
        doubleBufferSlice.flush();
    }
}
```


同样，我们通过下面的图来看看这个机制的实现：



7、且慢！刷写磁盘不是会导致锁持有时间过长吗?

且慢，各位同学，如果按照上面的伪代码思路，一定会有一个问题：要是一个线程，他获取了锁，开始写内存数据。

然后，发现内存满了，接着直接在持有锁的过程中，还去执行数据刷磁盘的操作，这样是有问题的。

要知道，数据刷磁盘是很慢的，根据数据的多少，搞不好要几十毫秒，甚至几百毫秒。

这样的话，岂不是一个线程会持有锁长达几十毫秒，甚至几百毫秒?

这当然不行了，后面的线程此时都在等待获取锁然后写缓冲区 2，你怎么能一直占有锁呢?

一旦你按照这个思路来写代码，必然导致高并发场景下，一个线程持有锁上百毫秒。刷数据到磁盘的时候，后续上百个工作线程全部卡在等待锁的那个环节，啥都干不了，严重的情况下，甚至又会导致系统整体呈现卡死的状态。

8、内存 + 磁盘并行写机制

所以此时正确的并发优化代码，应该是发现内存缓冲区 1 满了，然后就交换两个缓冲区。

接着直接就释放锁，释放锁了之后再由这个线程将数据刷入磁盘中，刷磁盘的过程是不会占用锁的，然后后续的线程都可以继续获取锁，快速写入内存，接着释放锁。

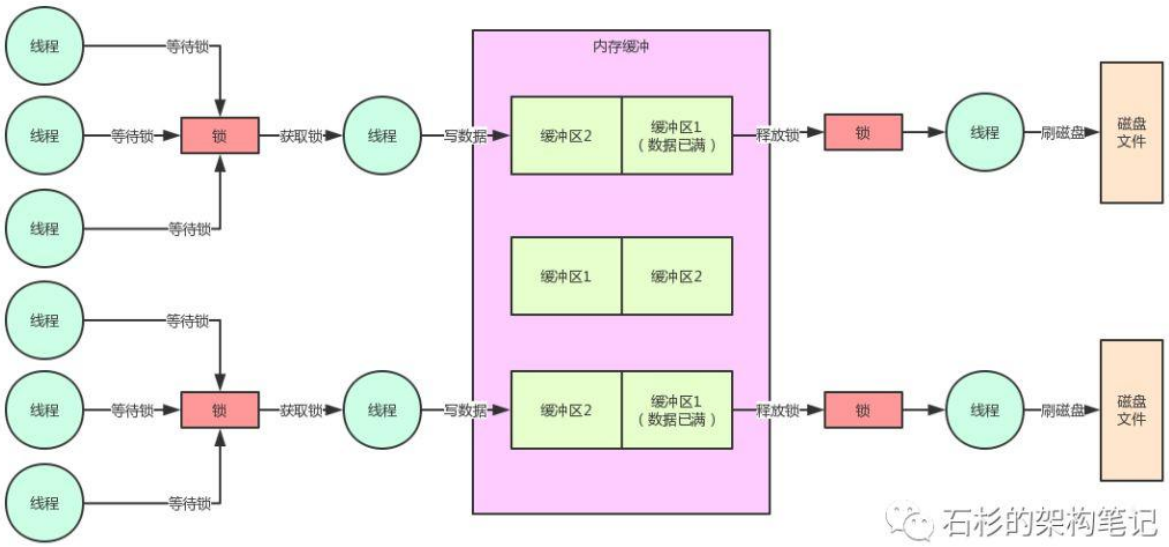
大家先看看下面的伪代码的优化：

```
// 任何一个线程，想要写内存缓冲，必须要先加锁才可以
// 但是可以从一个内存缓冲分片集合中随机挑选一个，自动负载均衡
DoubleBuffer doubleBufferSlice = doubleBuffers.chooseRandom();

// 这里加锁，仅仅对一个内存缓冲分片加锁，这样就可以大幅度提升锁的数量
// 同时大幅度降低线程对同一把锁的争用
synchronized(doubleBufferSlice) {

    // 这行代码，意思就是对双段缓冲写入数据，此时会写入缓冲区1中
    doubleBufferSlice.write(data);

    // 判断一下假如一块缓冲区是否写满了
    // 每块缓冲区大小都是限定好的，比如就只能放20MB的数据
    // 写到那么多的数据，就是写满了
    if(doubleBufferSlice.isFull()) {

        // 这个方法就是交换一下两块缓冲区，让缓冲区2提供写入服务
        doubleBufferSlice.exhange();
    }
}

// 下面的代码就是将缓冲区1的数据刷写到磁盘文件里去
// 这里优化的核心就在于，将刷磁盘的操作，放在synchronized加锁代码块外面做
// 这样这个耗时的操作就不会占用锁了
doubleBufferSlice.flush();
```


按照上面的伪代码的优化，此时磁盘的刷写和内存的写入，完全可以并行同时进行。

因为这里核心的要点就在于大幅度降低了锁占用的时间，这是 java 并发锁优化的一个非常核心的思路。

大家看下面的图，一起来感受一下：



9、为什么必须要用双缓冲机制?

其实看到这里，大家可能或多或少都体会到了一些双缓冲机制的设计思想了，如果只用单块内存缓冲的话，那么从里面读数据刷入磁盘的过程，也需要占用锁，而此时想要获取锁写入内存缓冲的线程是获取不到锁的。

所以假如只用单块缓冲，必然导致读内存数据，刷入磁盘的过程，长时间占用锁。进而导致大量线程卡在锁的获取上，无法获取到锁，然后无法将数据写入内存。**这就是必须要在这里使用双缓冲机制的核心原因。**

10、总结

最后做一下总结，本文从笔者团队自研的百万并发量级中间件系统的内核机制出发，给大家展示了 Java 并发中加锁的时候：

- 如何利用双缓冲机制
- 内存缓冲分片机制
- 分段加锁机制
- 磁盘 + 内存并行写入机制
- 高并发场景下大幅度优化多线程对锁的串行化争用问题
- 长时间占用锁的问题

其实在很多开源的优秀中间件系统中，都有很多类似的 Java 并发优化的机制，主要就是应对高并发的场景下大幅度的提升系统的并发性能以及吞吐量。大家如果感兴趣，也可以去了解阅读一下相关的底层源码。

如果 20 万用户同时访问一个热点缓存，如何优化你的缓存架构?

> 作者:中华石杉 原文地址

目录

（1）为什么要用缓存集群

这篇文章，咱们来聊聊热点缓存的架构优化问题。

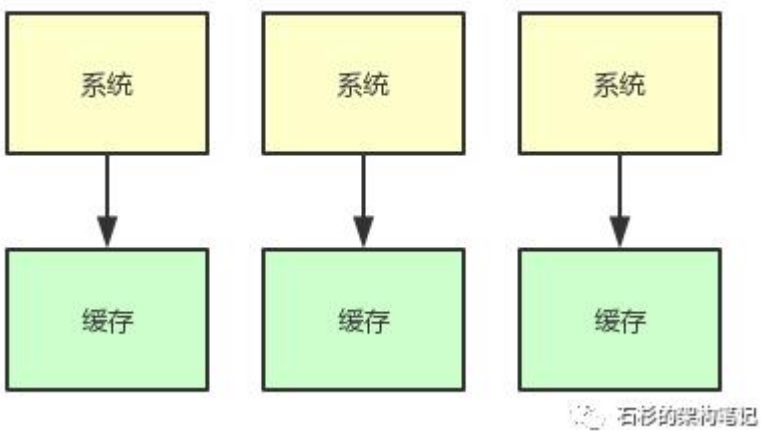
其实使用缓存集群的时候，最怕的就是热 key、大 value 这两种情况，那啥叫热 key 大 value 呢?

简单来说，热 key，就是你的缓存集群中的某个 key 瞬间被数万甚至十万的并发请求打爆。

大 value，就是你的某个 key 对应的 value 可能有 GB 级的大小，导致查询 value 的时候导致网络相关的故障问题。

这篇文章，我们就来聊聊热 key 问题。先来看看下面的一幅图。

简单来说，假设你手头有个系统，他本身是集群部署的，然后后面有一套缓存集群，这个集群不管你用 redis cluster，还是 memcached，或者是公司自研缓存集群，都可以。



那么，这套系统用缓存集群干什么呢?

很简单了，在缓存里放一些平时不怎么变动的数据，然后用户在查询大量的平时不怎么变动的数据的时候，不就可以直接从缓存里走了吗?

缓存集群的并发能力是很强的，而且读缓存的性能是很高的。

举个例子，假设你每秒有 2 万请求，但是其中 90% 都是读请求，那么每秒 1.8 万请求都是在读一些不太变化的数据，而不是写数据。

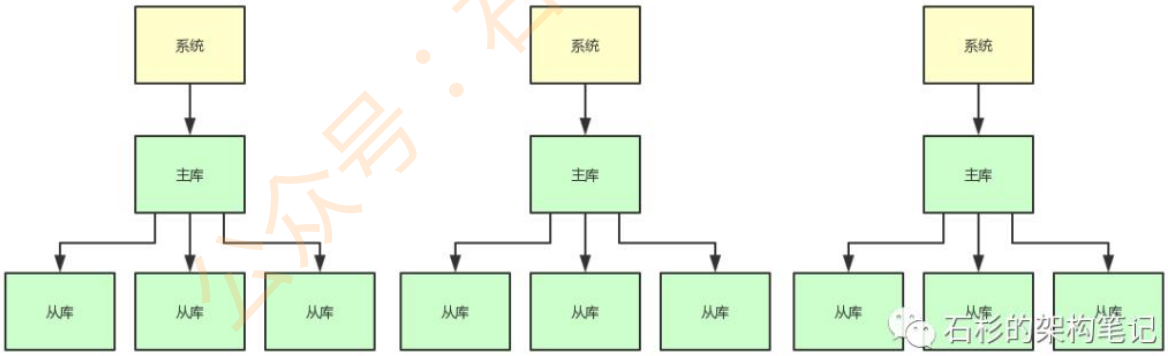
那此时你把数据都放在数据库里，然后每秒发送 2 万请求到数据库上读写数据，你觉得合适吗?

当然不太合适了，如果你要用数据库承载每秒 2 万请求的话，那么不好意思，你很可能就得搞分库分表 + 读写分离。

比如你得分 3 个主库，承载每秒 2000 的写入请求，然后每个主库挂 3 个从库，一共 9 个从库承载每秒 1.8 万的读请求。

这样的话，你可能就需要一共是 12 台高配置的数据库服务器，这是很耗费钱的，成本非常高，而且很不合适。

大家看看下面的图，来体会下这种情况。

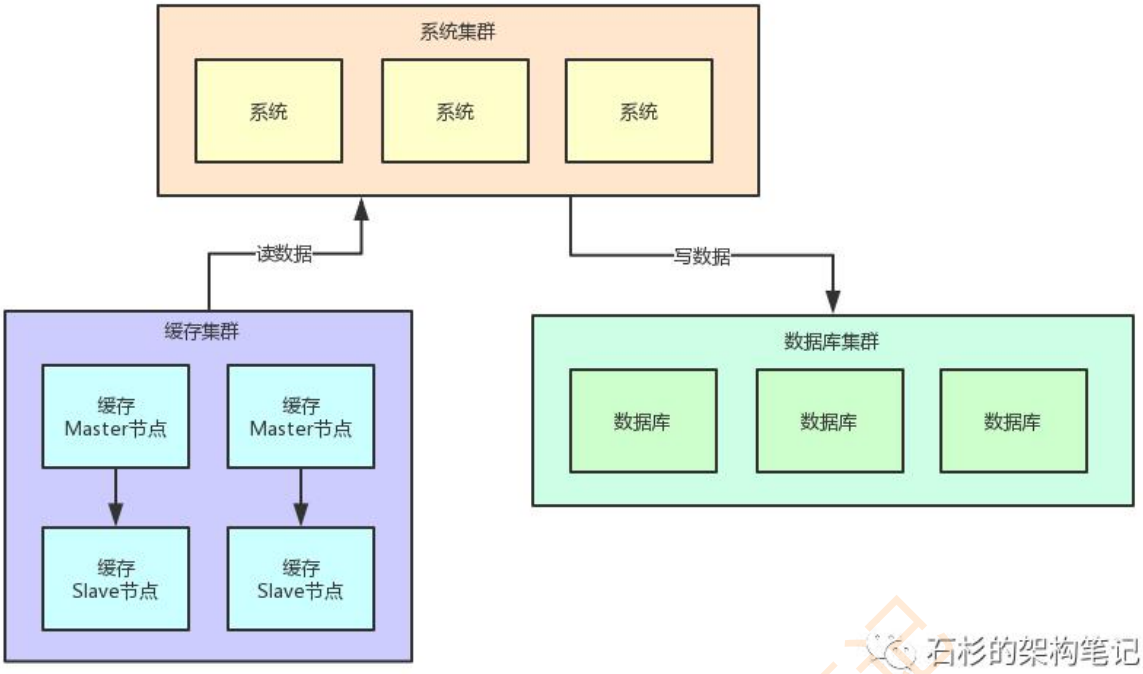


所以，此时你完全就可以把平时不太变化的数据放在缓存集群里，缓存集群可以采用 2 主 2 从，主节点用来写入缓存，从节点用来读缓存。

以缓存集群的性能，2 个从节点完全可以用来承载每秒 1.8 万的大量读了，然后 3 个数据库主库就是承载每秒 2000 的写请求和少量其他读请求就可以了。

大家看看下面的图，你耗费的机器瞬间变成了 4 台缓存机器 + 3 台数据库机器 = 7 台机器，是不是比之前的 12 台机器减少了很大的资源开销?

没错，缓存其实在系统架构里是非常重要的组成部分。很多时候，对于那些很少变化但是大量高并发读的数据，通过缓存集群来抗高并发读，是非常合适的。



这里所有的机器数量、并发请求量都是一个示例，大家主要是体会一下这个意思就好，其目的主要是给一些不太熟悉缓存相关技术的同学一点背景性的阐述，让这些同学能够理解在系统里用缓存集群承载读请求是什么意思。

（2）20 万用户同时访问一个热点缓存的问题

好了，背景是已经给大家解释清楚了，那么现在就可以给大家说说今天重点要讨论的问题：热点缓存。

我们来做一个假设，你现在有 10 个缓存节点来抗大量的读请求。正常情况下，读请求应该是均匀的落在 10 个缓存节点上的，对吧！

这 10 个缓存节点，每秒承载 1 万请求是差不多的。

然后我们再做一个假设，你一个节点承载 2 万请求是极限，所以一般你就限制一个节点正常承载 1 万请求就 ok 了，稍微留一点 buffer 出来。

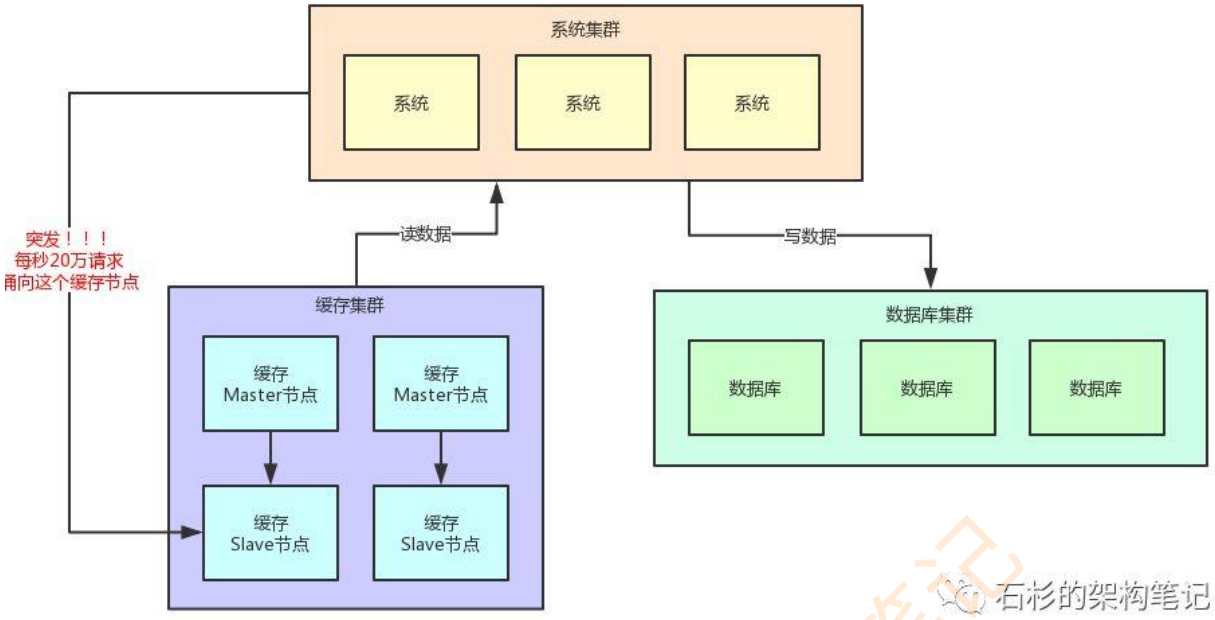
好，所谓的热点缓存问题是什么意思呢?

很简单，就是突然因为莫名的原因，出现大量的用户访问同一条缓存数据。

举个例子，某个明星突然宣布跟某某结婚，这个时候是不是会引发可能短时间内每秒都是数十万的用户去查看这个明星跟某某结婚的那条新闻?

那么假设那条新闻就是一个缓存，然后对应就是一个缓存 key，就存在一台缓存机器上，此时瞬时假设有 20 万请求奔向那一台机器上的一个 key。

此时会如何？我们看看下面的图，来体会一下这种绝望的感受。



这个时候很明显了，我们刚才假设的是一个缓存 Slave 节点最多每秒就是 2 万的请求，当然实际缓存单机承载 5 万~ 10 万读请求也是可能的，我们这里就是一个假设。

结果此时，每秒突然奔过来 20 万请求到这台机器上，会怎么样？

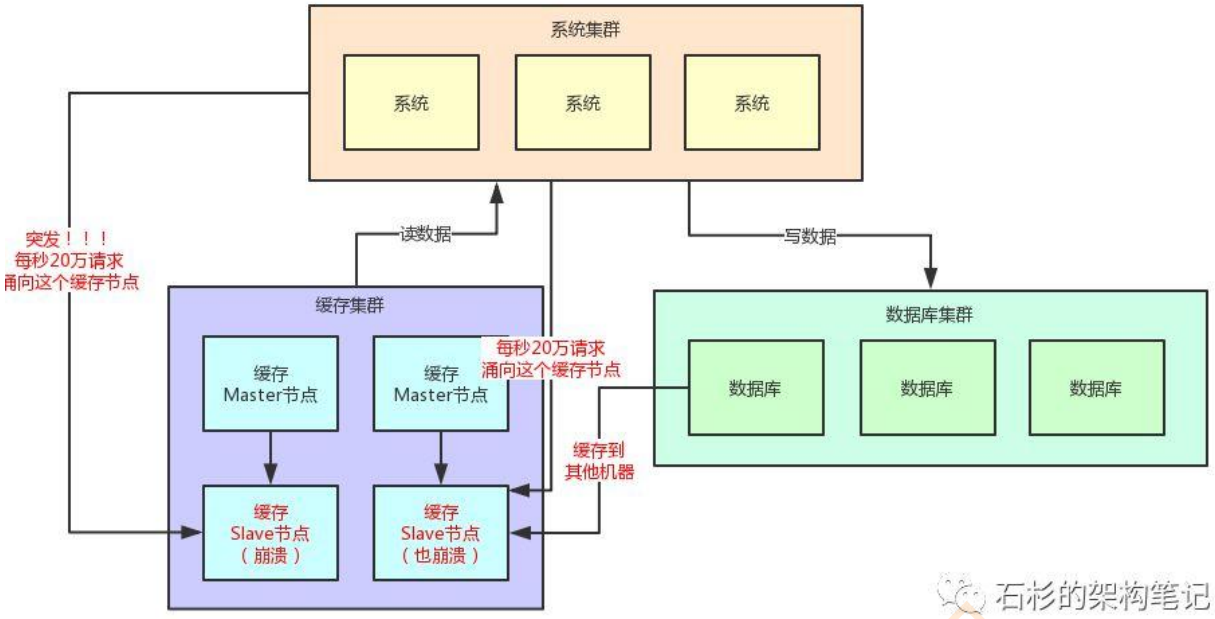
很简单，上面图里那台被 20 万请求指向的缓存机器会过度操劳而宕机的。

那么如果缓存集群开始出现机器的宕机，此时会如何？

接着，读请求发现读不到数据，会从数据库里提取原始数据，然后放入剩余的其他缓存机器里去。但是接踵而来的每秒 20 万请求，会再次压垮其他的缓存机器。

以此类推，最终导致缓存集群全盘崩溃，引发系统整体宕机。

咱们看看下面的图，再感受一下这个恐怖的现场。



（3）基于流式计算技术的缓存热点自动发现

其实这里关键的一点，就是对于这种热点缓存，你的系统需要能够在热点缓存突然发生的时候，直接发现他，然后瞬间立马实现毫秒级的自动负载均衡。

那么我们就先来说说，你如何自动发现热点缓存问题？

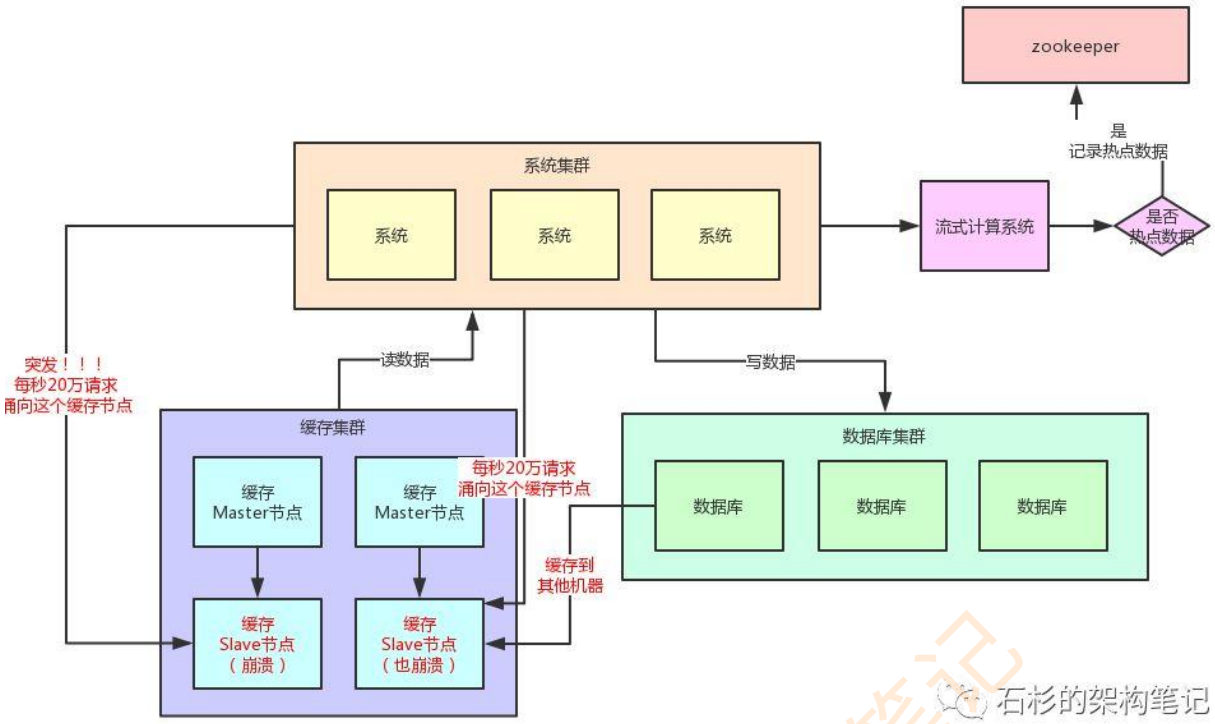
首先你要知道，一般出现缓存热点的时候，你的每秒并发肯定是很高的，可能每秒都几十万甚至上百万的请求量过来，这都是有可能的。

所以，此时完全可以基于大数据领域的流式计算技术来进行实时数据访问次数的统计，比如storm、spark streaming、flink，这些技术都是可以的。

然后一旦在实时数据访问次数统计的过程中，比如发现一秒之内，某条数据突然访问次数超过了 1000，就直接立马把这条数据判定为是热点数据，可以将这个发现出来的热点数据写入比如zookeeper 中。

当然，你的系统如何判定热点数据，可以根据自己的业务还有经验值来就可以了。

大家看看下面这张图，看看整个流程是如何进行的。

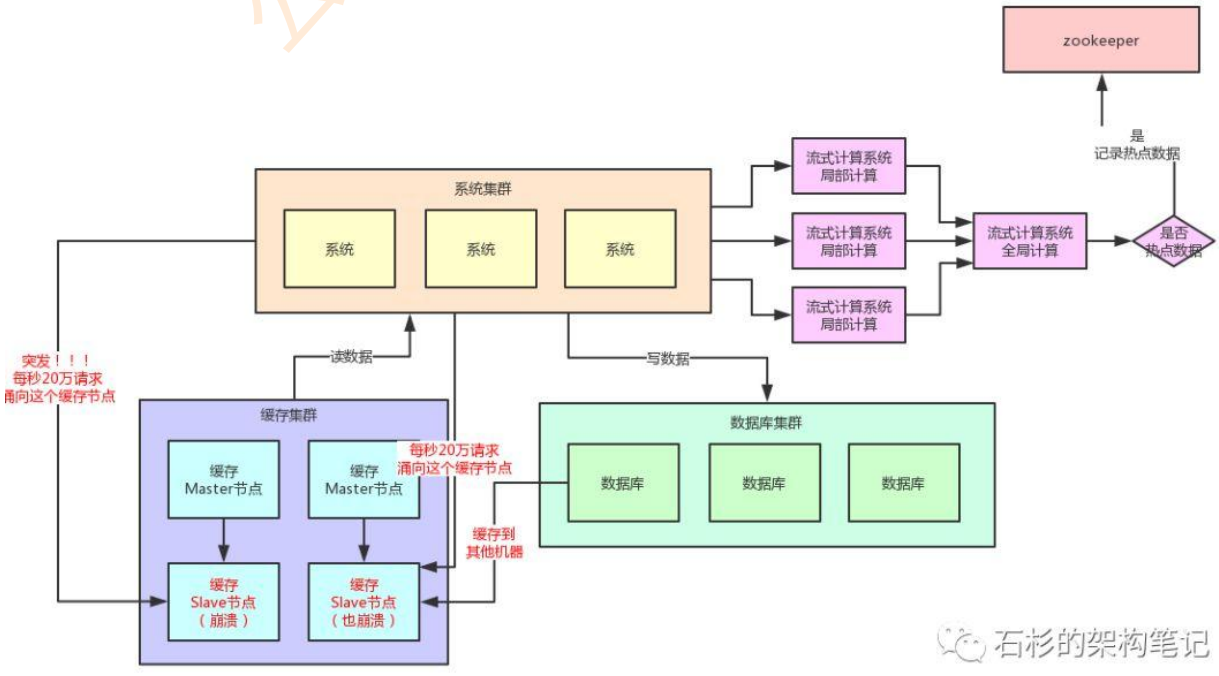


当然肯定有人会问，那你的流式计算系统在进行数据访问次数统计的时候，会不会也存在说单台机器被请求每秒几十万次的问题呢?

答案是否，因为流式计算技术，尤其是 storm 这种系统，他可以做到同一条数据的请求过来，先分散在很多机器里进行本地计算，最后再汇总局部计算结果到一台机器进行全局汇总。

所以几十万请求可以先分散在比如 100 台机器上，每台机器统计了这条数据的几千次请求。

然后 100 条局部计算好的结果汇总到一台机器做全局计算即可，所以基于流式计算技术来进行统计是不会有热点问题的。



（4）热点缓存自动加载为 JVM 本地缓存

我们自己的系统可以对 zookeeper 指定的热点缓存对应的 znode 进行监听，如果有变化他立马就可以感知到了。

此时系统层就可以立马把相关的缓存数据从数据库加载出来，然后直接放在自己系统内部的本地缓存里即可。

这个本地缓存，你用 ehcache、hashmap，其实都可以，一切都看自己的业务需求，主要说的就是将缓存集群里的集中式缓存，直接变成每个系统自己本地实现缓存即可，每个系统自己本地是无法缓存过多数据的。

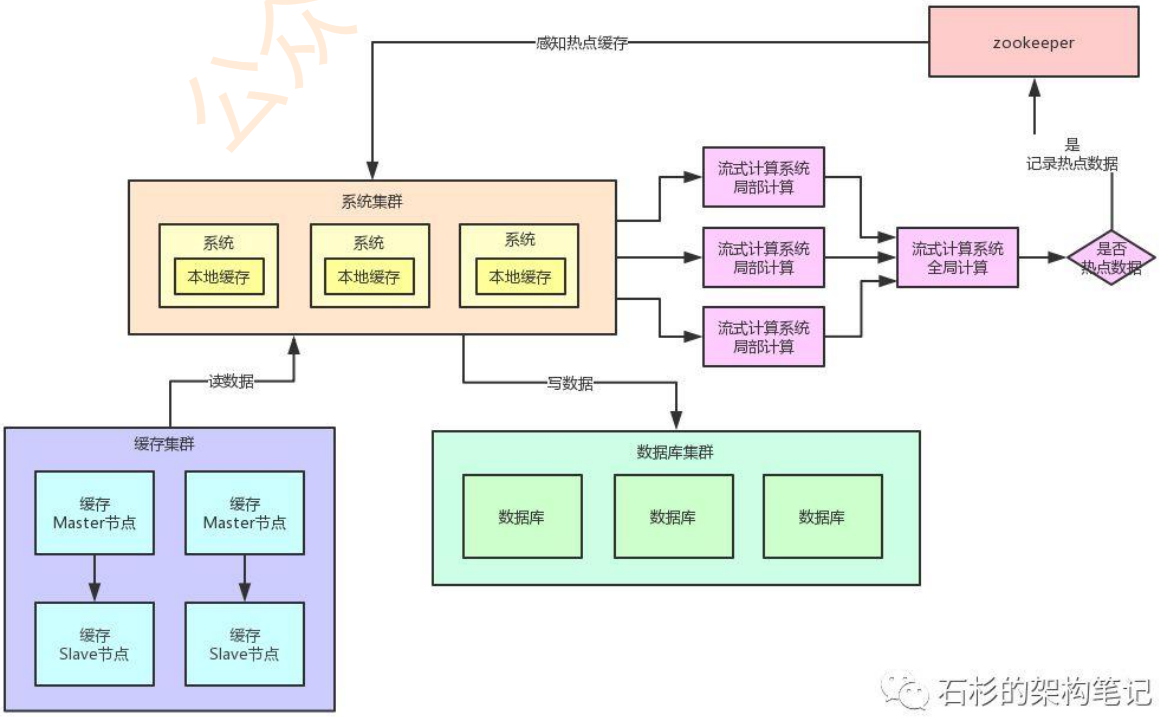
因为一般这种普通系统单实例部署机器可能就一个 4 核 8G 的机器，留给本地缓存的空间是很少的，所以用来放这种热点数据的本地缓存是最合适的，刚刚好。

假设你的系统层集群部署了 100 台机器，那么好了，此时你 100 台机器瞬间在本地都会有一份热点缓存的副本。

然后接下来对热点缓存的读操作，直接系统本地缓存读出来就给返回了，不用再走缓存集群了。

这样的话，也不可能允许每秒 20 万的读请求到达缓存机器的一台机器上读一个热点缓存了，而是变成 100 台机器每台机器承载数千请求，那么那数千请求就直接从机器本地缓存返回数据了，这是没有问题的。

我们再来画一幅图，一起来看看这个过程：



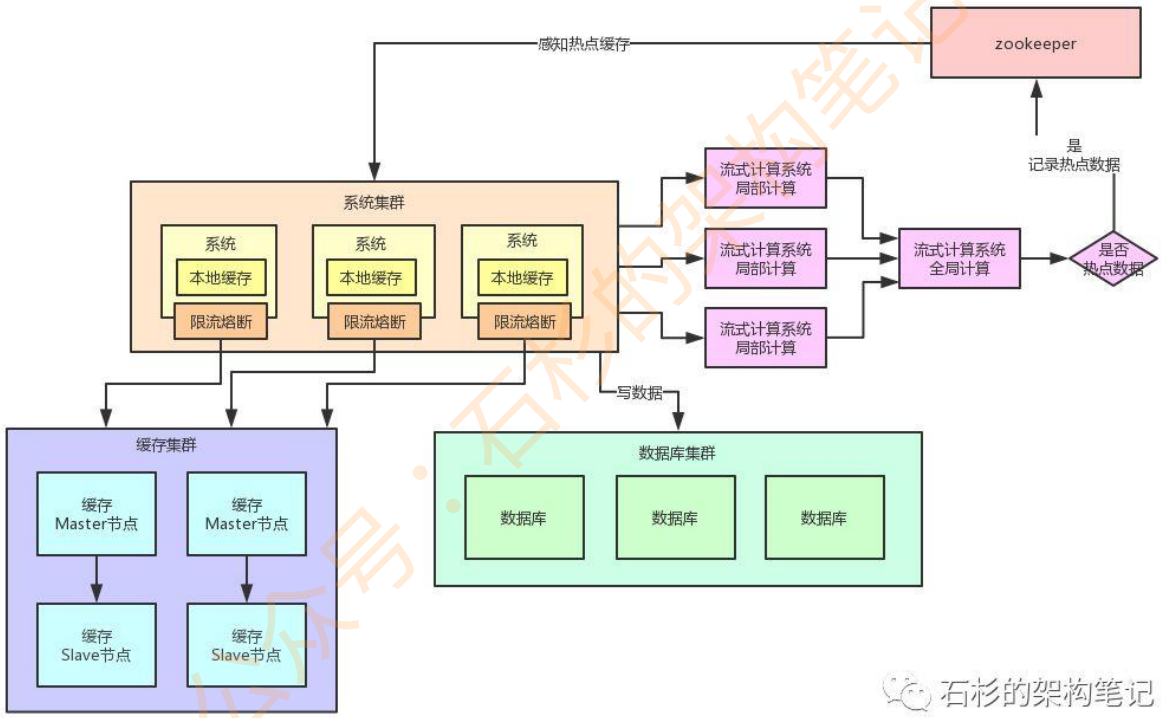
（5）限流熔断保护

除此之外，在每个系统内部，其实还应该专门加一个对热点数据访问的限流熔断保护措施。

每个系统实例内部，都可以加一个熔断保护机制，假设缓存集群最多每秒承载 4 万读请求，那么你一共有 100 个系统实例。

你自己就该限制好，每个系统实例每秒最多请求缓存集群读操作不超过 400 次，一超过就可以熔断掉，不让请求缓存集群，直接返回一个空白信息，然后用户稍后会自行再次重新刷新页面之类的。

通过系统层自己直接加限流熔断保护措施，可以很好的保护后面的缓存集群、数据库集群之类的不要被打死，我们来看看下面的图。



（6）本文总结

具体要不要在系统里实现这种复杂的缓存热点优化架构呢？这个还要看你们自己的系统有没有这种场景了。

如果你的系统有热点缓存问题，那么就要实现类似本文的复杂热点缓存支撑架构。

但是如果没有的话，那么也别过度设计，其实你的系统可能根本不需要这么复杂的架构。

如果是后者，那么大伙儿就权当看看本文，来了解一下对应的架构思想好了 ^_^

支撑日活百万用户的高并发系统，应该如何设计其数据库架构？

> 作者:中华石杉 原文地址

目录：

> ！“ 这篇文章，我们来聊一下对于一个支撑日活百万用户的高并系统，他的数据库架构应该如何设计？

看到这个题目，很多人第一反应就是：

分库分表啊！

但是实际上，数据库层面的分库分表到底是用来干什么的，他的不同的作用如何应对不同的场景，我觉得很多同学可能都没搞清楚。

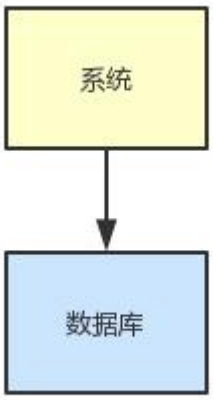
（1）用一个创业公司的发展作为背景引入

假如我们现在是一个小创业公司，注册用户就 20 万，每天活跃用户就 1 万，每天单表数据量就 1000，然后高峰期每秒钟并发请求最多就 10。

天哪！就这种系统，随便找一个有几年工作经验的高级工程师，然后带几个年轻工程师，随便干干都可以做出来。

因为这样的系统，实际上主要就是在前期快速的进行业务功能的开发，搞一个单块系统部署在一台服务器上，然后连接一个数据库就可以了。

接着大家就是不停的在一个工程里填充进去各种业务代码，尽快把公司的业务支撑起来，如下图所示。



结果呢，没想到我们运气这么好，碰上个优秀的 CEO 带着我们走上了康庄大道！

公司业务发展迅猛，过了几个月，注册用户数达到了 2000 万！每天活跃用户数 100 万！每天单表新增数据量达到 50 万条！高峰期每秒请求量达到 1 万！

同时公司还顺带着融资了两轮，估值达到了惊人的几亿美金！一只朝气蓬勃的幼年独角兽的节奏！

好吧，现在大家感觉压力已经有点大了，为啥呢？

因为每天单表新增 50 万条数据，一个月就多 1500 万条数据，一年下来单表会达到上亿条数据。

经过一段时间的运行，现在咱们单表已经两三千万条数据了，勉强还能支撑着。

但是，眼见着系统访问数据库的性能怎么越来越差呢，单表数据量越来越大，拖垮了一些复杂查询 SQL 的性能啊！

然后高峰期请求现在是每秒 1 万，咱们的系统在线上部署了 20 台机器，平均每台机器每秒支撑 500 请求，这个还能抗住，没啥大问题。

但是数据库层面呢？

如果说此时你还是一台数据库服务器在支撑每秒上万的请求，负责任的告诉你，每次高峰期会出现下述问题：

- 你的数据库服务器的磁盘 IO、网络带宽、CPU 负载、内存消耗，都会达到非常高的情况，数据库所在服务器的整体负载会非常重，甚至都快不堪重负了
- 高峰期时，本来你单表数据量就很大，SQL 性能就不太好，这时加上你的数据库服务器负载太高导致性能下降，就会发现你的 SQL 性能更差了

- 最明显的一个感觉，就是你的系统在高峰期各个功能都运行的很慢，用户体验很差，点一个按钮可能要几十秒才出来结果
- 如果你运气不太好，数据库服务器的配置不是特别的高的话，弄不好你还会经历数据库宕机的情况，因为负载太高对数据库压力太大了

（2）多台服务器分库支撑高并发读写

首先我们先考虑第一个问题，数据库每秒上万的并发请求应该如何来支撑呢?

要搞清楚这个问题，先得明白一般数据库部署在什么配置的服务器上。

通常来说，假如你用普通配置的服务器来部署数据库，那也起码是 16 核 32G 的机器配置。

这种非常普通的机器配置部署的数据库，一般线上的经验是：不要让其每秒请求支撑超过 2000，一般控制在 2000 左右。

控制在这个程度，一般数据库负载相对合理，不会带来太大的压力，没有太大的宕机风险。

所以首先第一步，就是在上万并发请求的场景下，部署个 5 台服务器，每台服务器上都部署一个数据库实例。

然后每个数据库实例里，都创建一个一样的库，比如说订单库。

此时在 5 台服务器上都有一个订单库，名字可以类似为：db_order_01，db_order_02，等等。

然后每个订单库里，都有一个相同的表，比如说订单库里有订单信息表，那么此时 5 个订单库里都有一个订单信息表。

比如 db_order_01 库里就有一个 tb_order_01 表，db_order_02 库里就有一个 tb_order_02 表。

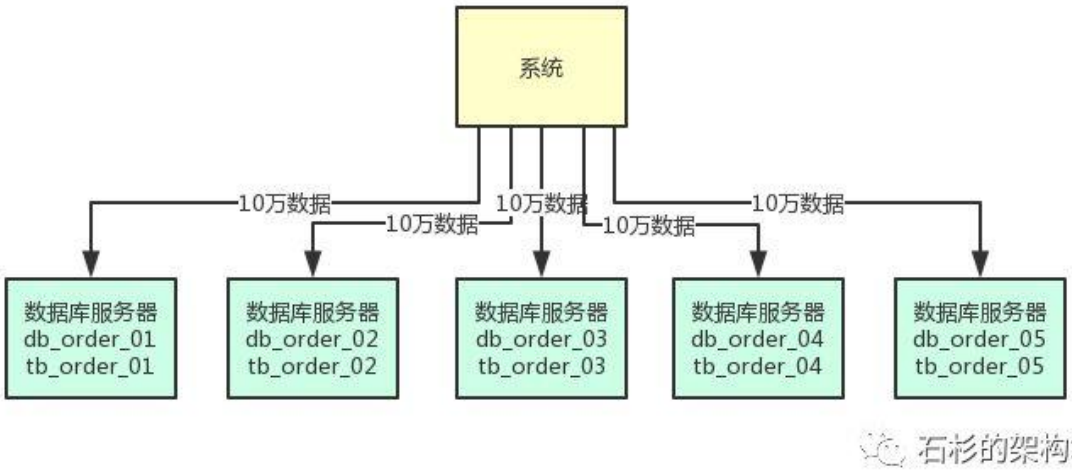
这就实现了一个基本的分库分表的思路，原来的一台数据库服务器变成了 5 台数据库服务器，原来的一个库变成了 5 个库，原来的一张表变成了 5 个表。

然后你在写入数据的时候，需要借助数据库中间件，比如 sharding-jdbc，或者是 mycat，都可以。

你可以根据比如订单 id 来 hash 后按 5 取模，比如每天订单表新增 50 万数据，此时其中 10 万条数据会落入 db_order_01 库的 tb_order_01 表，另外 10 万条数据会落入 db_order_02 库的 tb_order_02 表，以此类推。

这样就可以把数据均匀分散在 5 台服务器上了，查询的时候，也可以通过订单 id 来 hash 取模，去对应的服务器上的数据库里，从对应的表里查询那条数据出来即可。

依据这个思路画出的图如下所示，大家可以看看。



做这一步有什么好处呢?

第一个好处，原来比如订单表就一张表，这个时候不就成了 5 张表了么，那么每个表的数据就变成 1/5 了。

假设订单表一年有 1 亿条数据，此时 5 张表里每张表一年就 2000 万数据了。

那么假设当前订单表里已经有 2000 万数据了，此时做了上述拆分，每个表里就只有 400 万数据了。

而且每天新增 50 万数据的话，那么每个表才新增 10 万数据，这样是不是初步缓解了单表数据量过大影响系统性能的问题?

另外就是每秒 1 万请求到 5 台数据库上，每台数据库就承载每秒 2000 的请求，是不是一下子把每台数据库服务器的并发请求降低到了安全范围内?

这样，降低了数据库的高峰期负载，同时还保证了高峰期的性能。

（3）大量分表来保证海量数据下的查询性能

但是上述的数据库架构还有一个问题，那就是单表数据量还是过大，现在订单表才分为了 5 张表，那么如果订单一年有 1 亿条，每个表就有 2000 万条，这也还是太大了。

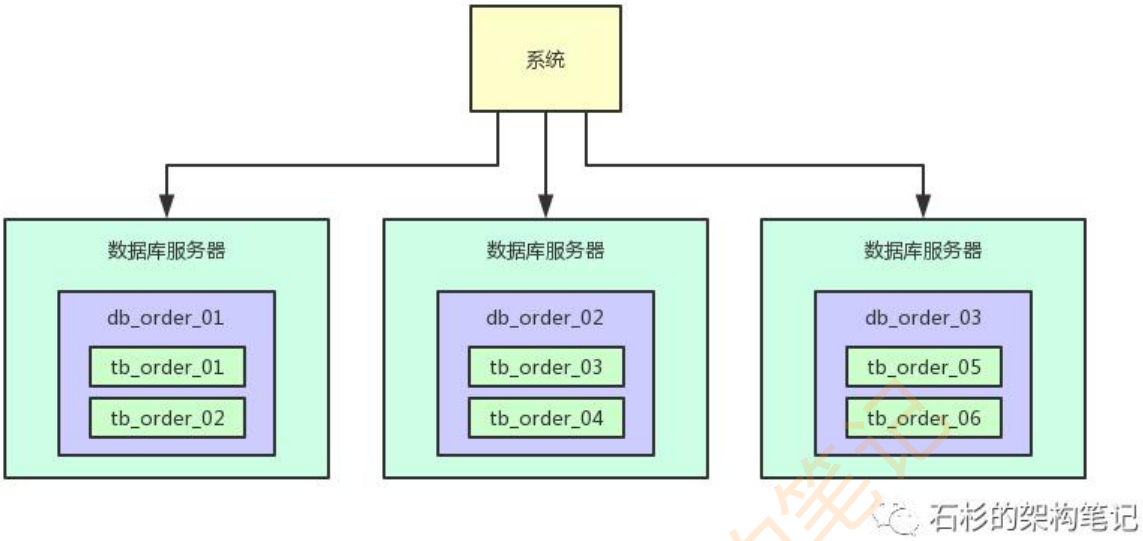
所以还应该继续分表，大量分表。

比如可以把订单表一共拆分为 1024 张表，这样 1 亿数据量的话，分散到每个表里也就才 10 万量级的数据量，然后这上千张表分散在 5 台数据库里就可以了。

在写入数据的时候，需要做两次路由，先对订单 id hash 后对数据库的数量取模，可以路由到一台数据库上，然后再对那台数据库上的表数量取模，就可以路由到数据库上的一个表里了。

通过这个步骤，就可以让每个表里的数据量非常小，每年 1 亿数据增长，但是到每个表里才 10 万条数据增长，这个系统运行 10 年，每个表里可能才百万级的数据量。

这样可以一次性为系统未来的运行做好充足的准备，看下面的图，一起来感受一下：



（4）读写分离来支撑按需扩容以及性能提升

这个时候整体效果已经挺不错了，大量分表的策略保证可能未来 10 年，每个表的数据量都不会太大，这可以保证单表内的 SQL 执行效率和性能。

然后多台数据库的拆分方式，可以保证每台数据库服务器承载一部分的读写请求，降低每台服务器的负载。

但是此时还有一个问题，假如说每台数据库服务器承载每秒 2000 的请求，然后其中 400 请求是写入，1600 请求是查询。

也就是说，增删改的 SQL 才占到了 20% 的比例，80% 的请求是查询。

此时假如说随着用户量越来越大，假如说又变成每台服务器承载 4000 请求了。

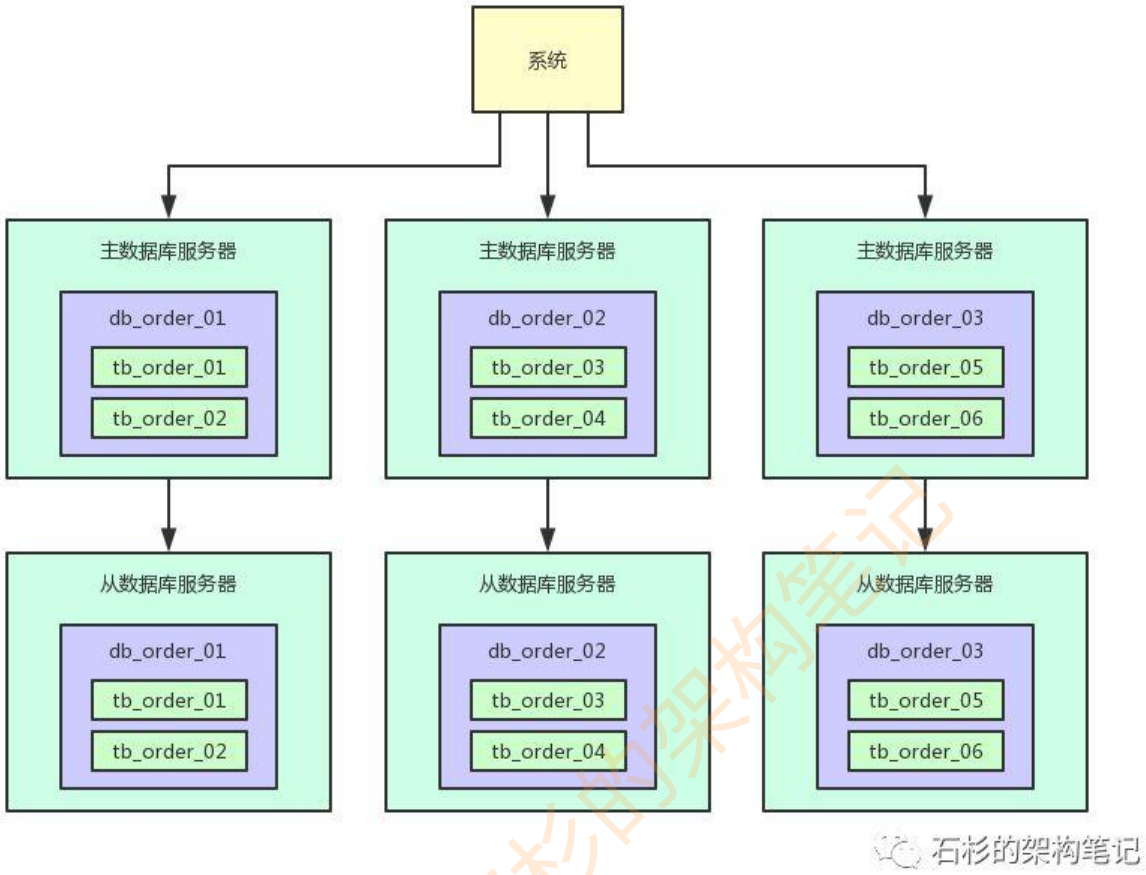
那么其中 800 请求是写入，3200 请求是查询，如果说你按照目前的情况来扩容，就需要增加一台数据库服务器.

但是此时可能就会涉及到表的迁移，因为需要迁移一部分表到新的数据库服务器上去，是不是很麻烦?

其实完全没必要，数据库一般都支持读写分离，也就是做主从架构。

写入的时候写入主数据库服务器，查询的时候读取从数据库服务器，就可以让一个表的读写请求分开落地到不同的数据库上去执行。

这样的话，假如写入主库的请求是每秒 400，查询从库的请求是每秒 1600，那么图大概如下所示。



写入主库的时候，会自动同步数据到从库上去，保证主库和从库数据一致。

然后查询的时候都是走从库去查询的，这就通过数据库的主从架构实现了读写分离的效果了。

现在的好处就是，假如说现在主库写请求增加到 800，这个无所谓，不需要扩容。然后从库的读请求增加到了 3200，需要扩容了。

这时，你直接给主库再挂载一个新的从库就可以了，两个从库，每个从库支撑 1600 的读请求，不需要因为读请求增长来扩容主库。

实际上线上生产你会发现，读请求的增长速度远远高于写请求，所以读写分离之后，大部分时候就是扩容从库支撑更高的读请求就可以了。

而且另外一点，对同一个表，如果你既写入数据（涉及加锁），还从该表查询数据，可能会牵扯到锁冲突等问题，无论是写性能还是读性能，都会有影响。

所以一旦读写分离之后，对主库的表就仅仅是写入，没任何查询会影响他，对从库的表就仅仅是查询。

（5）高并发下的数据库架构设计总结

其实从大的一个简化的角度来说，高并发的场景下，数据库层面的架构肯定是需要经过精心的设计的。

尤其是涉及到分库来支撑高并发的请求，大量分表保证每个表的数据量别太大，读写分离实现主库和从库按需扩容以及性能保证。

这篇文章就是从一个大的角度来梳理了一下思路，各位同学可以结合自己公司的业务和项目来考虑自己的系统如何做分库分表应该怎么做。

另外就是，具体的分库分表落地的时候，需要借助数据库中间件来实现分库分表和读写分离，大家可以自己参考 sharding-jdbc 或者 mycat 的官网即可，里面的文档都有详细的使用描述。

支撑百万连接的系统应该如何设计其高并发架构?

> **作者:中华石杉** 原文地址

目录

> ! “ 这篇文章，给大家聊聊：如果你设计一个系统需要支撑百万用户连接，应该如何来设计其高并发请求处理架构?

（1）到底什么是连接?

假如说现在你有一个系统，他需要连接很多很多的硬件设备，这些硬件设备都要跟你的系统来通信。

那么，怎么跟你的系统通信呢?

首先，他一定会跟你的系统建立连接，然后会基于那个连接发送请求给你的系统。

接着你的系统会返回响应给那个系统，最后是大家一起把连接给断开，释放掉网络资源。

所以我们来看一下下面的那个图，感受一下这个所谓的连接到底是个什么概念。

（2）为什么每次发送请求都要建立连接?

但是大家看着上面的那个图，是不是感觉有一个很大的问题。

什么问题呢？那就是为啥每次发送请求，都必须要建立一个连接，然后再断开一个连接?

要知道，网络连接的建立和连接涉及到多次网络通信，本质是一个比较耗费资源的过程。

所以说咱们完全没必要每次发送请求都要建立一次连接，断开一次连接。

我们完全可以建立好一个连接，然后设备就不停的发送请求过来，系统就通过那个连接返回响应。

大家完全可以多次通过一个连接发送请求和返回响应，这就是所谓的长连接。

也就是说，如果你一个连接建立之后，然后发送请求，接着就断开，那这个连接维持的时间是很短的，这个就是所谓的短连接。

那如果一个设备跟你的系统建立好一个连接，然后接着就不停的通过这个连接发送请求接收响应，就可以避免不停的创建连接和断开连接的开销了。

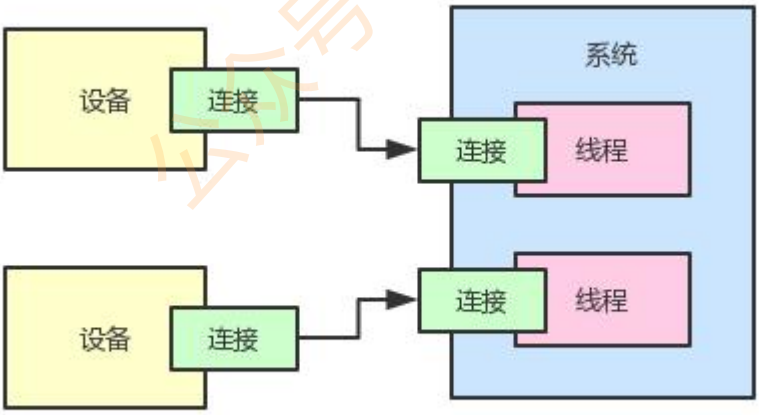
大家看下面的图，体验一下这个过程。在图里面，两次连接之间，有很多次发送请求和接收响应的过程，这样就可以利用一个连接但是进行多次通信了。

（3）长连接模式下需要耗费大量线程资源

但是现在问题又来了，长连接的模式确实是不错的，但是如果说每个设备都要跟系统长期维持一个连接，那么对于系统来说就需要搞一个线程，这个线程需要去维护一个设备的长连接，然后通过这个连接跟一个设备不停的通信，接收人家发送过来的请求，返回响应给人家。

大家看下面的图，每个设备都要跟系统维持一个连接，那么对于每个设备的连接，系统都会有一个独立的线程来维护这个连接。

因为你必须要有一个线程不停的尝试从网络连接中读取请求，接着要处理请求，最后还要返回响应给设备。



那么这种模式有什么缺点呢？

缺点是很显而易见的，假如说此时你有上百万个设备要跟你的系统进行连接，假设你的系统做了集群部署一共有 100 个服务实例，难道每个服务实例要维持 1 万个连接支撑跟 1 万个设备的通信？

如果这样的话，每个服务实例不就是要维持 1 万个线程来维持 1 万个连接了吗？大家觉得这个事儿靠谱吗？

根据线上的生产经验，一般 4 核 8G 的标准服务用的虚拟机，自己开辟的工作线程在一两百个就会让 CPU 负载很高了，最佳的建议就是在几十个工作线程就差不多。

所以要是期望每个服务实例来维持上万个线程，那几乎是不可能的，所以这种模式最大的问题就在于这里，没法支撑大量连接。

（4）Kafka 遇到的问题：应对大量客户端连接

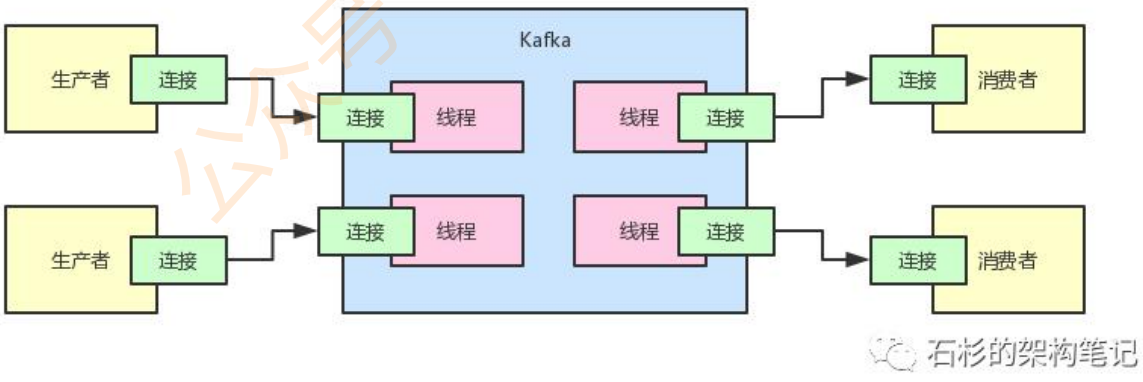
实际上，对于大名鼎鼎的消息系统 Kafka 来说，他也是会面对同样的问题，因为他需要应对大量的客户端连接。

有很多生产者和消费者都要跟 Kafka 建立类似上面的长连接，然后基于一个连接，一直不停的通信。

举个例子，比如生产者需要通过一个连接，不停的发送数据给 Kafka。然后 Kafka 也要通过这个连接不停的返回响应给生产者。

消费者也需要通过一个连接不停的从 Kafka 获取数据，Kafka 需要通过这个连接不停的返回数据给消费者。

大家看下面的图，感受一下 Kafka 的生产现场。



那假如 Kafka 就简单的按照这个架构来处理，如果你的公司里有几万几十万个的生产者或者消费者的服务实例，难道 Kafka 集群就要为了几万几十万个连接来维护这么多的线程吗？

同样，这是不现实的，因为线程是昂贵的资源，不可能在集群里使用那么多的线程。

（5）Kafka 的架构实践：Reactor 多路复用

针对这个问题，大名鼎鼎的 Kafka 采用的架构策略是 Reactor 多路复用模型。

简单来说，就是搞一个 acceptor 线程，基于底层操作系统的支持，实现连接请求监听。

如果有某个设备发送了建立连接的请求过来，那么那个线程就把这个建立好的连接交给 processor 线程。

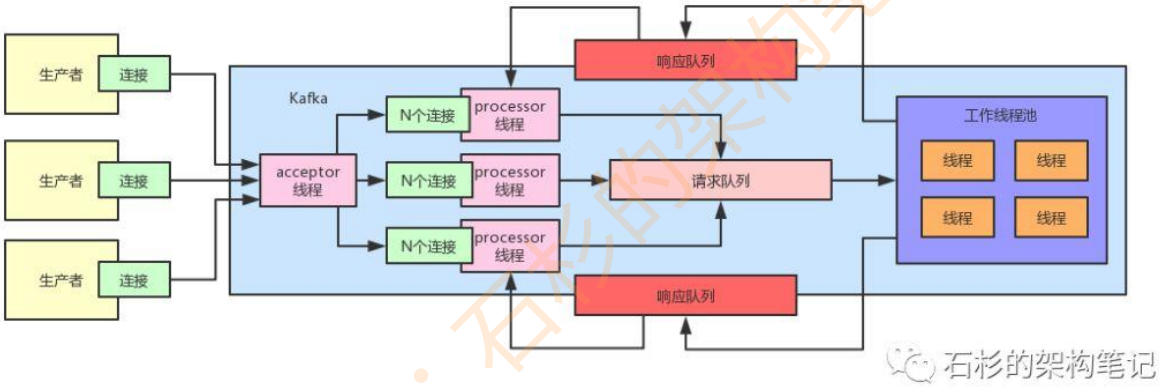
每个 processor 线程会被分配 N 多个连接，一个线程就可以负责维持 N 多个连接，他同样会基于底层操作系统的支持监听 N 多连接的请求。

如果某个连接发送了请求过来，那么这个 processor 线程就会把请求放到一个请求队列里去。

接着后台有一个线程池，这个线程池里有工作线程，会从请求队列里获取请求，处理请求，接着将请求对应的响应放到每个 processor 线程对应的一个响应队列里去。

最后，processor 线程会把自己的响应队列里的响应发送回给客户端。

说了这么多，还是来一张图，大家看下面的图，就可以理解上述整个过程了。



（6）优化后的架构是如何支撑大量连接的?

那么上面优化后的那套架构，是如何支撑大量连接的呢?

其实很简单。这里最关键的一个因素，就是 processor 线程是一个人维持 N 个线程，基于底层操作系统的特殊机制的支持，一个人可以监听 N 个连接的请求。

这是极为关键的一个步骤，就仅此一个步骤就可以让一个线程支持多个连接了，不需要一个连接一个线程来支持。

而且那个 processor 线程仅仅是接收请求和发送响应，所有的请求都会入队列排队，交给后台线程池来处理。

比如说按照 100 万连接来计算，如果有 100 台机器来处理，按照老的模式，每台机器需要维持 1 万个线程来处理 1 万个连接。

但是如果按照这种多路复用的模式，可能就比如 10 个 processor + 40 个线程的线程池，一共 50 个线程就可以上万连接。

在这种模式下，每台机器有限的线程数量可以抗住大量的连接。

因此实际上我们在设计这种支撑大量连接的系统的时候，完全可以参考这种架构，设计成多路复用的模式，用几十个线程处理成千上万个连接，最终实现百万连接的处理架构。

拜托！面试请不要再问我Spring Cloud底层原理

> **作者:中华石杉** 原文地址

目录

概述

毫无疑问，Spring Cloud 是目前微服务架构领域的翘楚，无数的书籍博客都在讲解这个技术。不过大多数讲解还停留在对 Spring Cloud 功能使用的层面，其底层的很多原理，很多人可能并不知晓。因此本文将通过大量的手绘图，给大家谈谈 Spring Cloud 微服务架构的底层原理。 实际上，Spring Cloud 是一个全家桶式的技术栈，包含了很多组件。本文先从其最核心的几个组件入手，来剖析一下其底层的工作原理。也就是 Eureka、Ribbon、Feign、Hystrix、Zuul 这几个组件。

一、业务场景介绍

先来给大家说一个业务场景，假设咱们现在开发一个电商网站，要实现支付订单的功能，流程如下：

- 创建一个订单之后，如果用户立刻支付了这个订单，我们需要将订单状态更新为 “已支付”
- 扣减相应的商品库存
- 通知仓储中心，进行发货

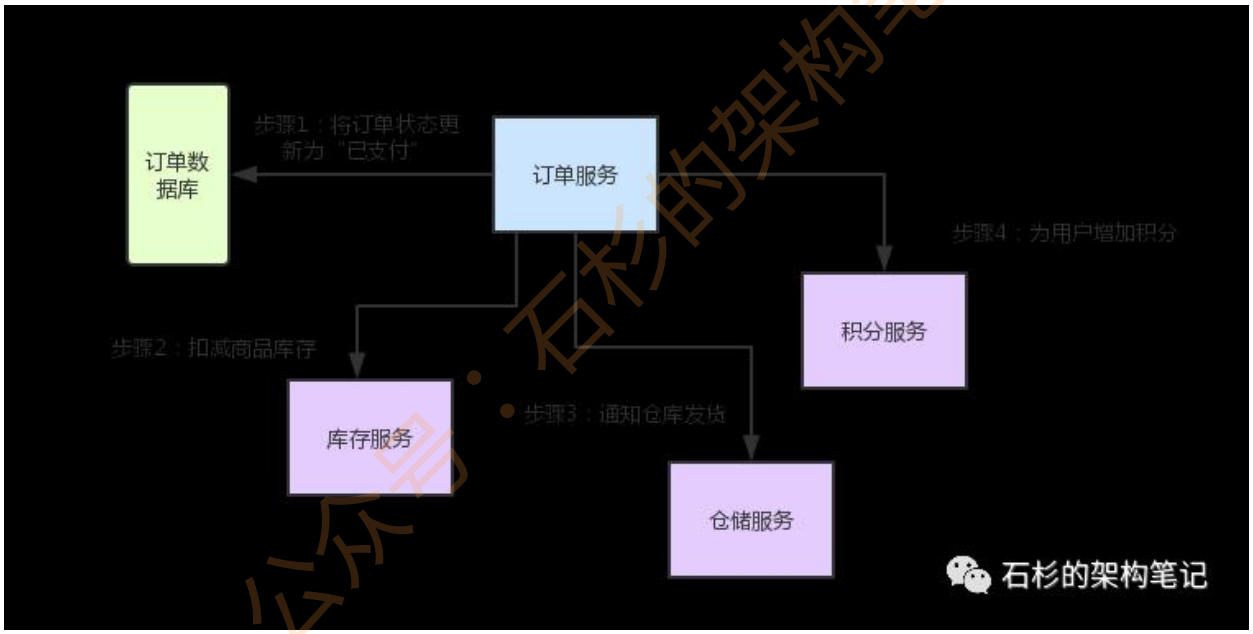
- 给用户的这次购物增加相应的积分

针对上述流程，我们需要有订单服务、库存服务、仓储服务、积分服务。整个流程的大体思路如下：

- 用户针对一个订单完成支付之后，就会去找订单服务，更新订单状态
- 订单服务调用库存服务，完成相应功能
- 订单服务调用仓储服务，完成相应功能
- 订单服务调用积分服务，完成相应功能

至此，整个支付订单的业务流程结束

下图这张图，清晰表明了各服务间的调用过程：



好！有了业务场景之后，咱们就一起来看看 Spring Cloud 微服务架构中，这几个组件如何相互协作，各自发挥的作用以及其背后的原理。

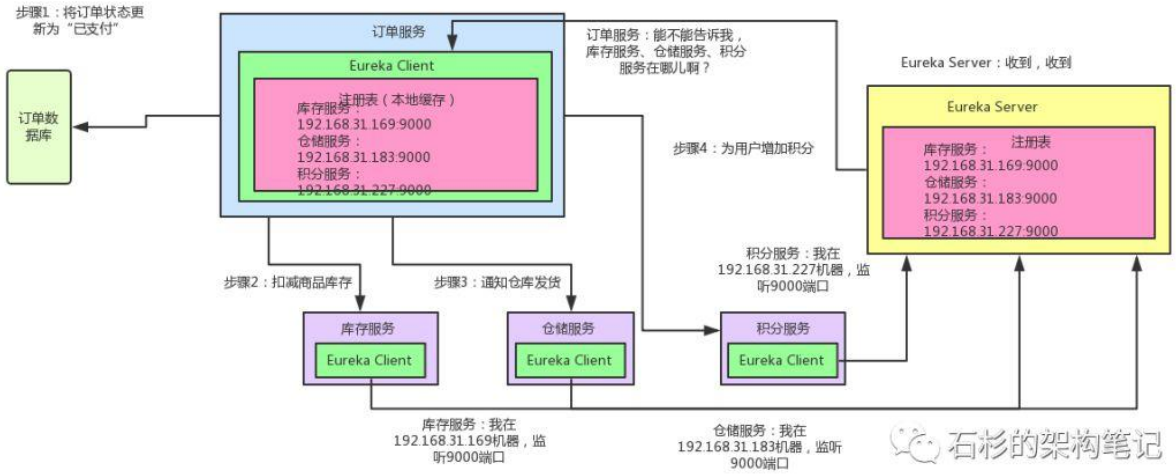
二、Spring Cloud 核心组件：Eureka

咱们来考虑第一个问题：订单服务想要调用库存服务、仓储服务，或者是积分服务，怎么调用？

订单服务压根儿就不知道人家库存服务在哪台机器上啊！他就算想要发起一个请求，都不知道发送给谁，有心无力！

这时候，就轮到 Spring Cloud Eureka 出场了。Eureka 是微服务架构中的注册中心，专门负责服务的注册与发现。

咱们来看看下面的这张图，结合图来仔细剖析一下整个流程：



如上图所示，库存服务、仓储服务、积分服务中都有一个 Eureka Client 组件，这个组件专门负责将这个服务的信息注册到 Eureka Server 中。说白了，就是告诉 Eureka Server，自己在哪台机器上，监听着哪个端口。而 Eureka Server 是一个注册中心，里面有一个注册表，保存了各服务所在的机器和端口号

订单服务里也有一个 Eureka Client 组件，这个 Eureka Client 组件会找 Eureka Server 问一下：库存服务在哪台机器啊？监听着哪个端口啊？仓储服务呢？积分服务呢？然后就可以把这些相关信息从 Eureka Server 的注册表中拉取到自己本地缓存起来。

这时如果订单服务想要调用库存服务，不就可以找自己本地的 Eureka Client 问一下库存服务在哪台机器？监听哪个端口吗？收到响应后，紧接着就可以发送一个请求过去，调用库存服务扣减库存的那个接口！同理，如果订单服务要调用仓储服务、积分服务，也是如法炮制。

总结一下：

- Eureka Client：负责将这个服务的信息注册到 Eureka Server 中
- Eureka Server：注册中心，里面有一个注册表，保存了各个服务所在的机器和端口号

三、Spring Cloud 核心组件：Feign

现在订单服务确实知道库存服务、积分服务、仓库服务在哪里了，同时也监听着哪些端口号了。但是新问题又来了：难道订单服务要自己写一大堆代码，跟其他服务建立网络连接，然后构造一个复杂的请求，接着发送请求过去，最后对返回的响应结果再写一大堆代码来处理吗？

这是上述流程翻译的代码片段，咱们一起来看看，体会一下这种绝望而无助的感受！！！

友情提示，前方高能：

```java
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpPost httpPost = new HttpPost("http://192.168.31.169:9000/");

List<NameValuePair> parameters = new ArrayList<NameValuePair>();
parameters.add(new BasicNameValuePair("scope", "project"));
parameters.add(new BasicNameValuePair("q", "java"));

UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(parameters);
httpPost.setEntity(formEntity);
httpPost.setHeader(
"User-Agent",
"Mozilla/5.0 (Windows NT 6.3; Win64; x64)"
);

CloseableHttpResponse response = null;
response = httpclient.execute(httpPost);
if (response.getStatusLine().getStatusCode() == 200) {
    String content = EntityUtils.toString(response.getEntity(), "UTF-8");
    System.out.println(content);
}
if (response != null) {
    response.close();
}
httpclient.close();
```


看完上面那一大段代码，有没有感到后背发凉、一身冷汗？实际上你进行服务间调用时，如果每次都手写代码，代码量比上面那段要多至少几倍，所以这个事儿压根儿就不是地球人能干的。

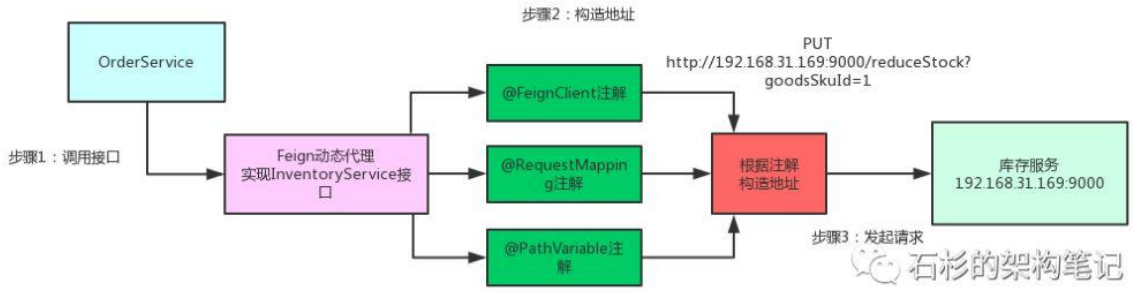
既然如此，那怎么办呢？别急，Feign 早已为我们提供好了优雅的解决方案。来看看如果用 Feign 的话，你的订单服务调用库存服务的代码会变成啥样？

```java
@FeignClient("inventory-service")
public class InventoryService {
    @RequestMapping(value = "/reduceStock/{goodsSkuId}", method = HttpMethod.PUT)
    @ResponseBody
    public ResultCode reduceStock(@PathVariable("goodsSkuId") Long goodsSkuId);
}

@Service
public class OrderService {
    @Autowired
    private InventoryService inventoryService;

    public ResultCode payOrder() {
        // 步骤1：更新本地数据库订单状态为“已支付”
        orderDAO.updateStatus(id, OrderStatus.PAYED);
        // 步骤2：调用库存服务，扣减商品库存
        inventoryService.reduceStock(goodsSkuId);
    }
}
```


看完上面的代码什么感觉？是不是感觉整个世界都干净了，又找到了活下去的勇气！没有底层的建立连接、构造请求、解析响应的代码，直接就是用注解定义一个 FeignClient 接口，然后调用那个接口就可以了。人家 Feign Client 会在底层根据你的注解，跟你指定的服务建立连接、构造请求、发起踌求、获取响应、解析响应，等等。这一系列脏活累活，人家 Feign 全给你干了。

那么问题来了，Feign 是如何做到这么神奇的呢？很简单，Feign 的一个关键机制就是使用了动态代理。咱们一起来看看下面的图，结合图来分析：

- 首先，如果你对某个接口定义了 @FeignClient 注解，Feign 就会针对这个接口创建一个动态代理
- 接着你要是调用那个接口，本质就是会调用 Feign 创建的动态代理，这是核心中的核心
- Feign 的动态代理会根据你在接口上的 @RequestMapping 等注解，来动态构造出你要请求的服务的地址
- 最后针对这个地址，发起请求、解析响应



四、Spring Cloud 核心组件：Ribbon

说完了 Feign，还没完。现在新的问题又来了，如果人家库存服务部署在了 5 台机器上，如下所示：

- 192.168.169:9000
- 192.168.170:9000
- 192.168.171:9000
- 192.168.172:9000
- 192.168.173:9000

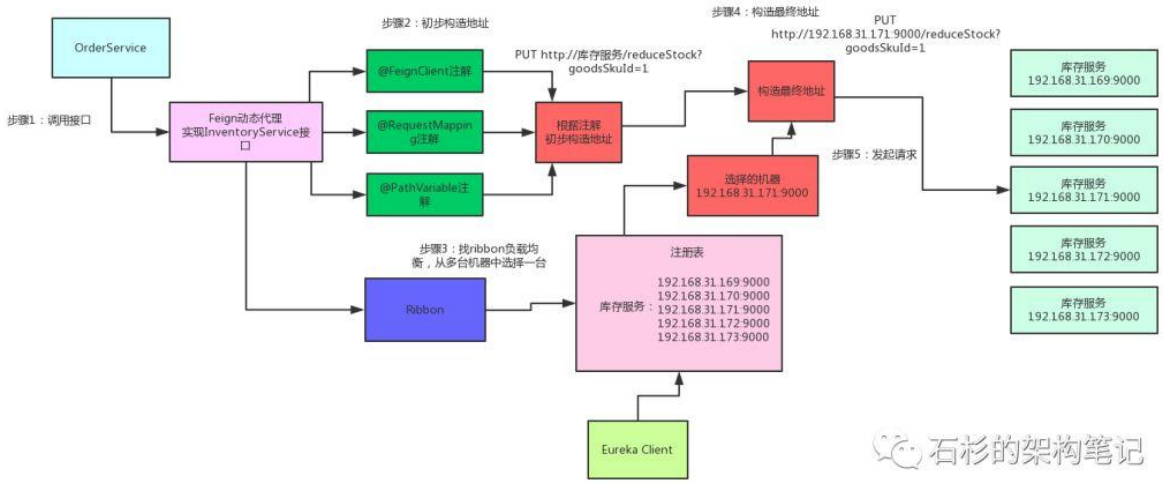
这下麻烦了！人家 Feign 怎么知道该请求哪台机器呢?

- 这时 Spring Cloud Ribbon 就派上用场了。Ribbon 就是专门解决这个问题的。它的作用是负载均衡，会帮你在每次请求时选择一台机器，均匀的把请求分发到各个机器上
- Ribbon 的负载均衡默认使用的最经典的 Round Robin 轮询算法。这是啥？简单来说，就是如果订单服务对库存服务发起 10 次请求，那就先让你请求第 1 台机器、然后是第 2 台机器、第 3 台机器、第 4 台机器、第 5 台机器，接着再来一个循环，第 1 台机器、第 2 台机器。。。以此类推。

此外，Ribbon 是和 Feign 以及 Eureka 紧密协作，完成工作的，具体如下：

- 首先 Ribbon 会从 Eureka Client 里获取到对应的服务注册表，也就知道了所有的服务都部署在了哪些机器上，在监听哪些端口号。
- 然后 Ribbon 就可以使用默认的 Round Robin 算法，从中选择一台机器
- Feign 就会针对这台机器，构造并发起请求。

对上述整个过程，再来一张图，帮助大家更深刻的理解：



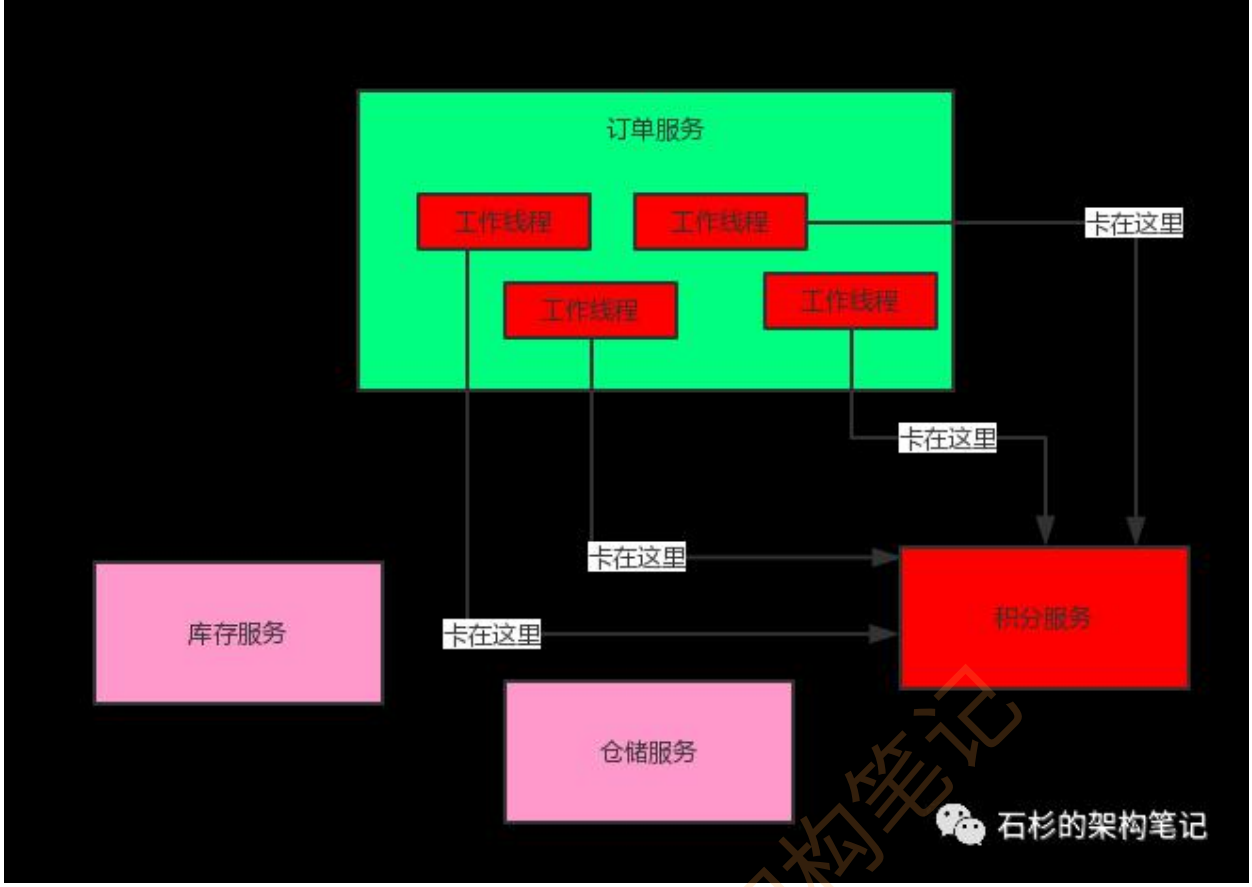
五、Spring Cloud 核心组件：Hystrix

在微服务架构里，一个系统会有很多的服务。以本文的业务场景为例：订单服务在一个业务流程里需要调用三个服务。现在假设订单服务自己最多只有 100 个线程可以处理请求，然后呢，积分服务不幸的挂了，每次订单服务调用积分服务的时候，都会卡住几秒钟，然后抛出一个超时异常。

咱们一起来分析一下，这样会导致什么问题？

- 如果系统处于高并发的场景下，大量请求涌过来的时候，订单服务的 100 个线程都会卡在请求积分服务这块。导致订单服务没有一个线程可以处理请求
- 然后就会导致别人请求订单服务的时候，发现订单服务也挂了，不响应任何请求了

上面这个，就是微服务架构中恐怖的服务雪崩问题，如下图所示：



如上图，这么多服务互相调用，要是不做任何保护的话，某一个服务挂了，就会引起连锁反应，导致别的服务也挂。比如积分服务挂了，会导致订单服务的线程全部卡在请求积分服务这里，没有一个线程可以工作，瞬间导致订单服务也挂了，别人请求订单服务全部会卡住，无法响应。

但是我们思考一下，就算积分服务挂了，订单服务也可以不用挂啊！为什么？

- 我们结合业务来看：支付订单的时候，只要把库存扣减了，然后通知仓库发货就 OK 了
- 如果积分服务挂了，大不了等他恢复之后，慢慢人肉手工恢复数据！为啥一定要因为一个积分服务挂了，就直接导致订单服务也挂了呢？不可以接受！

现在问题分析完了，如何解决？

这时就轮到 Hystrix 闪亮登场了。Hystrix 是隔离、熔断以及降级的一个框架。啥意思呢？说白了，Hystrix 会搞很多个小小的线程池，比如订单服务请求库存服务是一个线程池，请求仓储服务是一个线程池，请求积分服务是一个线程池。每个线程池里的线程就仅仅用于请求那个服务。

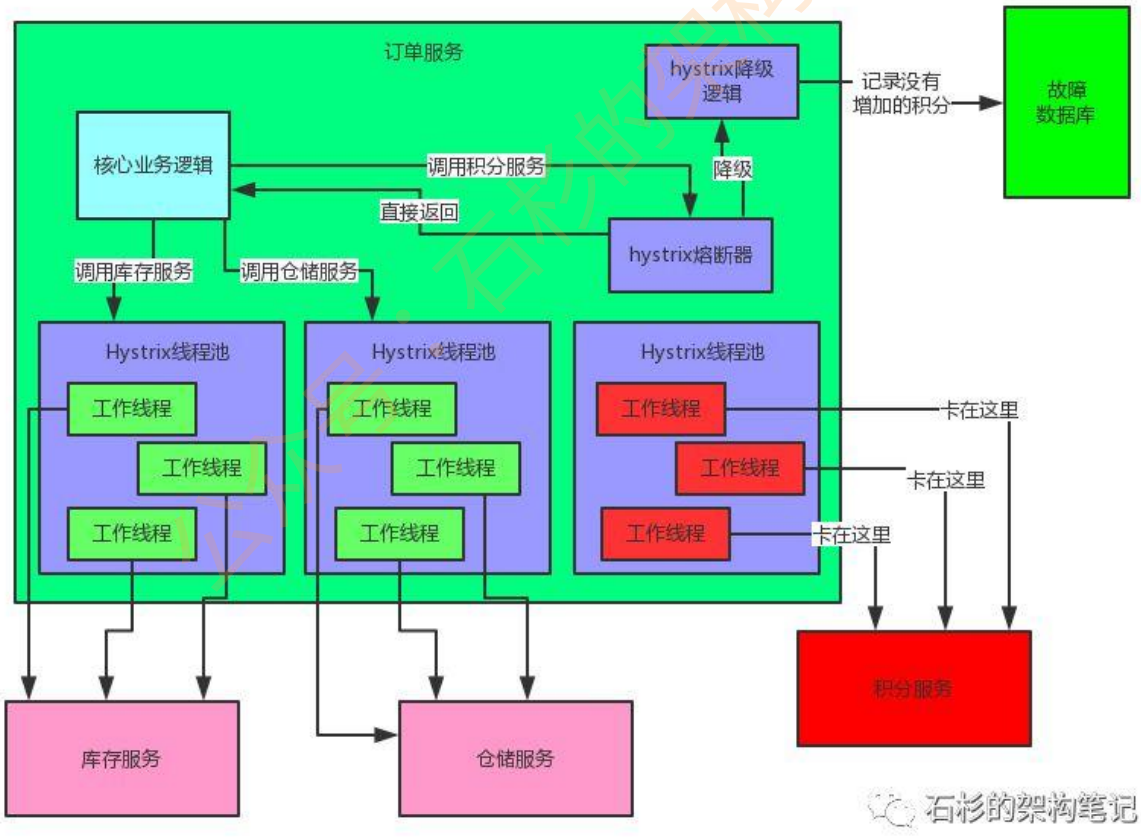
打个比方：现在很不幸，积分服务挂了，会咋样？

当然会导致订单服务里的那个用来调用积分服务的线程都卡死不能工作了啊！但是由于订单服务调用库存服务、仓储服务的这两个线程池都是正常工作的，所以这两个服务不会受到任何影响。

这个时候如果别人请求订单服务，订单服务还是可以正常调用库存服务扣减库存，调用仓储服务通知发货。只不过调用积分服务的时候，每次都会报错。但是如果积分服务都挂了，每次调用都要去卡住几秒钟干啥呢？有意义吗？当然没有！所以我们直接对积分服务熔断不就得了，比如在 5 分钟内请求积分服务直接就返回了，不要去走网络请求卡住几秒钟，这个过程，就是所谓的熔断！

那人家又说，兄弟，积分服务挂了你就熔断，好歹你干点儿什么啊！别啥都不干就直接返回啊？没问题，咱们就来个降级：每次调用积分服务，你就在数据库里记录一条消息，说给某某用户增加了多少积分，因为积分服务挂了，导致没增加成功！这样等积分服务恢复了，你可以根据这些记录手工加一下积分。这个过程，就是所谓的降级。

为帮助大家更直观的理解，接下来用一张图，梳理一下 Hystrix 隔离、熔断和降级的全流程：



六、Spring Cloud 核心组件：Zuul

说完了 Hystrix，接着给大家说说最后一个组件：Zuul，也就是微服务网关。这个组件是负责网络路由的。不懂网络路由？行，那我给你说说，如果没有 Zuul 的日常工作会怎样？

假设你后台部署了几百个服务，现在有个前端兄弟，人家请求是直接从浏览器那儿发过来的。打个比方：人家要请求一下库存服务，你难道还让人家记着这服务的名字叫做 inventory-service？部署在 5 台机器上？就算人家肯记住这一个，你后台可有几百个服务的名称和地址呢？难不成人家请求一个，就得记住一个？你要这样玩儿，那真是友谊的小船，说翻就翻！

上面这种情况，压根儿是不现实的。所以一般微服务架构中都必然会设计一个网关在里面，像 android、ios、pc 前端、微信小程序、H5 等等，不用去关心后端有几百个服务，就知道有一个网关，所有请求都往网关走，网关会根据请求中的一些特征，将请求转发给后端的各个服务。

而且有一个网关之后，还有很多好处，比如可以做统一的降级、限流、认证授权、安全，等等。

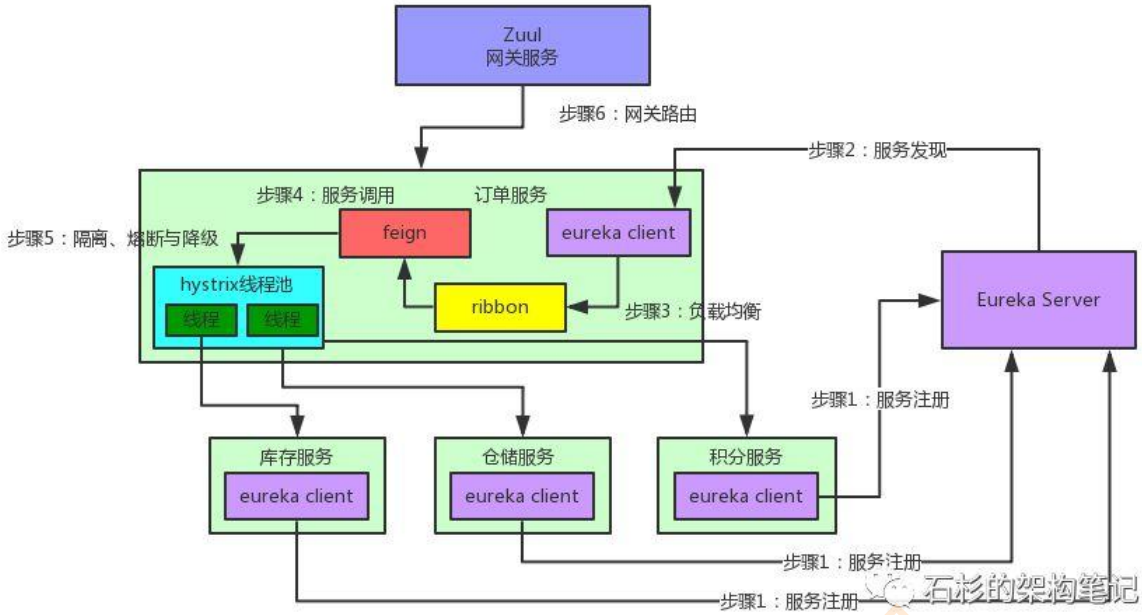
七、总结

最后再来总结一下，上述几个 Spring Cloud 核心组件，在微服务架构中，分别扮演的角色：

- Eureka：各个服务启动时，Eureka Client 都会将服务注册到 Eureka Server，并且 Eureka Client 还可以反过来从 Eureka Server 拉取注册表，从而知道其他服务在哪里
- Ribbon：服务间发起请求的时候，基于 Ribbon 做负载均衡，从一个服务的多台机器中选择一台
- Feign：基于 Feign 的动态代理机制，根据注解和选择的机器，拼接请求 URL 地址，发起请求
- Hystrix：发起请求是通过 Hystrix 的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题
- Zuul：如果前端、移动端要调用后端系统，统一从 Zuul 网关进入，由 Zuul 网关转发请求给对应的服务

以上就是我们通过一个电商业务场景，阐述了 Spring Cloud 微服务架构几个核心组件的底层原理。

文字总结还不够直观？没问题！我们将 Spring Cloud 的 5 个核心组件通过一张图串联起来，再来直观的感受一下其底层的架构原理：



如有收获，请帮忙转发，您的鼓励是作者最大的动力，谢谢！

【双11狂欢的背后】微服务注册中心如何承载大型系统的千万级访问?

> **作者:中华石杉** 原文地址

目录：

一、问题起源

Spring Cloud 架构体系中，Eureka 是一个至关重要的组件，它扮演着微服务注册中心的角色，所有的服务注册与服务发现，都是依赖 Eureka 的。

不少初学 Spring Cloud 的朋友在落地公司生产环境部署时，经常会问：

- Eureka Server 到底要部署几台机器?
- 我们的系统那么多服务，到底会对 Eureka Server 产生多大的访问压力?
- Eureka Server 能不能抗住一个大型系统的访问压力?

如果你也有这些疑问，别着急！咱们这就一起去看看，Eureka 作为微服务注册中心的核心原理

下面这些问题，大家先看看，有个大概印象。带着这些问题，来看后面的内容，效果更佳

- Eureka 注册中心使用什么样的方式来储存各个服务注册时发送过来的机器地址和端口号?
- 各个服务找 Eureka Server 拉取注册表的时候，是什么样的频率?
- 各个服务是如何拉取注册表的?
- 一个几百服务，部署上千台机器的大型分布式系统，会对 Eureka Server 造成多大的访问压力?
- Eureka Server 从技术层面是如何抗住日千万级访问量的?

先给大家说一个基本的知识点，各个服务内的 Eureka Client 组件，默认情况下，每隔 30 秒会发送一个请求到 Eureka Server，来拉取最近有变化的服务信息

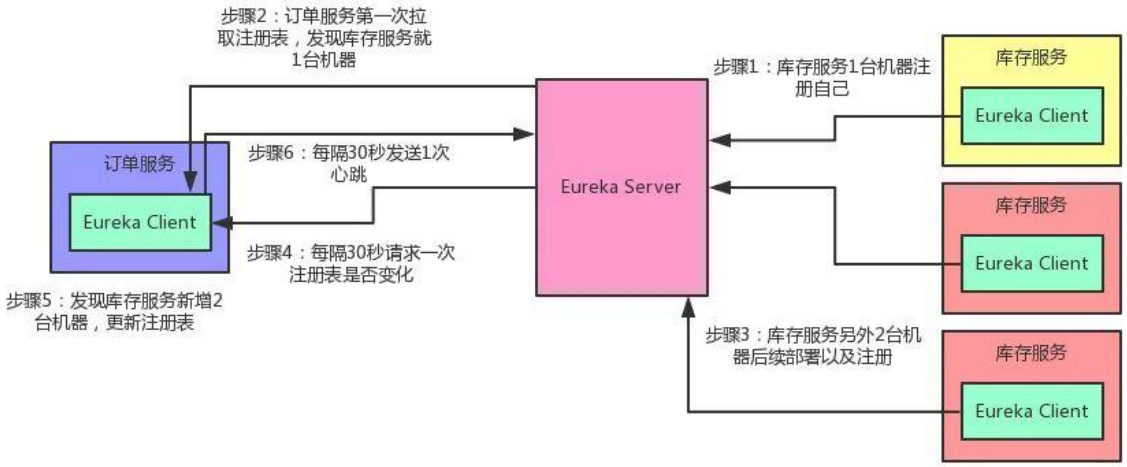
举个例子：

- 库存服务原本部署在 1 台机器上，现在扩容了，部署到了 3 台机器，并且均注册到了 Eureka Server 上。
- 然后订单服务的 Eureka Client 会每隔 30 秒去找 Eureka Server 拉取最近注册表的变化，看看其他服务的地址有没有变化。

除此之外，Eureka 还有一个心跳机制，各个 Eureka Client 每隔 30 秒会发送一次心跳到 Eureka Server，通知人家说，哥们，我这个服务实例还活着！

如果某个 Eureka Client 很长时间没有发送心跳给 Eureka Server，那么就说明这个服务实例已经挂了。

光看上面的文字，大家可能没什么印象。老规矩！咱们还是来一张图，一起来直观的感受一下这个过程。



二、Eureka Server 设计精妙的注册表存储结构

现在咱们假设手头有一套大型的分布式系统，一共 100 个服务，每个服务部署在 20 台机器上，机器是 4 核 8G 的标准配置。

也就是说，相当于你一共部署了 100 * 20 = 2000 个服务实例，有 2000 台机器。

每台机器上的服务实例内部都有一个 Eureka Client 组件，它会每隔 30 秒请求一次 Eureka Server，拉取变化的注册表。

此外，每个服务实例上的 Eureka Client 都会每隔 30 秒发送一次心跳请求给 Eureka Server。

那么大家算算，Eureka Server 作为一个微服务注册中心，每秒钟要被请求多少次？一天要被请求多少次?

- 按标准的算法，每个服务实例每分钟请求 2 次拉取注册表，每分钟请求 2 次发送心跳
- 这样一个服务实例每分钟会请求 4 次，2000 个服务实例每分钟请求 8000 次
- 换算到每秒，则是 8000 / 60 = 133 次左右，我们就大概估算为 Eureka Server 每秒会被请求 150 次
- 那一天的话，就是 8000 * 60 * 24 = 1152 万，也就是每天千万级访问量

好！经过这么一个测算，大家是否发现这里的奥秘了?

- 首先，对于微服务注册中心这种组件，在一开始设计它的拉取频率以及心跳发送频率时，就已经考虑到了一个大型系统的各个服务请求时的压力，每秒会承载多大的请求量。
- 所以各服务实例每隔 30 秒发起请求拉取变化的注册表，以及每隔 30 秒发送心跳给 Eureka Server，其实这个时间安排是有其用意的。

按照我们的测算，一个上百个服务，几千台机器的系统，按照这样的频率请求 Eureka Server，日请求量在千万级，每秒的访问量在 150 次左右。

即使算上其他一些额外操作，我们姑且就算每秒钟请求 Eureka Server 在 200 次~300 次吧。

所以通过设置一个适当的拉取注册表以及发送心跳的频率，可以保证大规模系统里对 Eureka Server 的请求压力不会太大。

关键问题来了，Eureka Server 是如何保证轻松抗住这每秒数百次请求，每天千万级请求的呢?

要搞清楚这个，首先得清楚 Eureka Server 到底是用什么来存储注册表的？三个字，看源码

接下来咱们就一起进入 Eureka 源码里一探究竟：

```java
// 这是代表内存注册表的类
public abstract class AbstractInstanceRegistry implements InstanceRegistry {

    private final CocurrentHashMap<String, Map<String, Lease<InstanceInfo>>> registry
        = new ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>();
}
```

- 如上图所示，图中的这个名字叫做 registry 的 CocurrentHashMap，就是注册表的核心结构。看完之后忍不住先赞叹一下，精妙的设计！
- 从代码中可以看到，Eureka Server 的注册表直接基于纯内存，即在内存里维护了一个数据结构。
- 各个服务的注册、服务下线、服务故障，全部会在内存里维护和更新这个注册表。
- 各个服务每隔 30 秒拉取注册表的时候，Eureka Server 就是直接提供内存里存储的有变化的注册表数据给他们就可以了。
- 同样，每隔 30 秒发起心跳时，也是在这个纯内存的 Map 数据结构里更新心跳时间。

一句话概括：维护注册表、拉取注册表、更新心跳时间，全部发生在内存里！这是 Eureka Server 非常核心的一个点。

搞清楚了这个，咱们再来分析一下 registry 这个东西的数据结构，大家千万别被它复杂的外表唬住了，沉下心来，一层层的分析！

- 首先，这个 ConcurrentHashMap 的 key 就是服务名称，比如 “inventory-service”，就是一个服务名称。
- value 则代表了一个服务的多个服务实例。
- 举例：比如 “inventory-service” 是可以有 3 个服务实例的，每个服务实例部署在一台机器上。

再来看看作为 value 的这个 Map：

```markup
Map<String, Lease<InstanceInfo>>
```

这个 Map 的 key 就是服务实例的 id

- value 是一个叫做 Lease 的类，它的泛型是一个叫做 InstanceInfo 的东东，你可能会问，这俩又是什么鬼?

- 首先说下 InstanceInfo，其实啊，我们见名知义，这个 InstanceInfo 就代表了服务实例的具体信息，比如机器的 ip 地址、hostname 以及端口号。

- 而这个 Lease，里面则会维护每个服务最近一次发送心跳的时间

三、Eureka Server 端优秀的多级缓存机制

假设 Eureka Server 部署在 4 核 8G 的普通机器上，那么基于内存来承载各个服务的请求，每秒钟最多可以处理多少请求呢?

- 根据之前的测试，单台 4 核 8G 的机器，处理纯内存操作，哪怕加上一些网络的开销，每秒处理几百请求也是轻松加愉快的。

- 而且 Eureka Server 为了避免同时读写内存数据结构造成的并发冲突问题，还采用了多级缓存机制来进一步提升服务请求的响应速度。

- 在拉取注册表的时候：

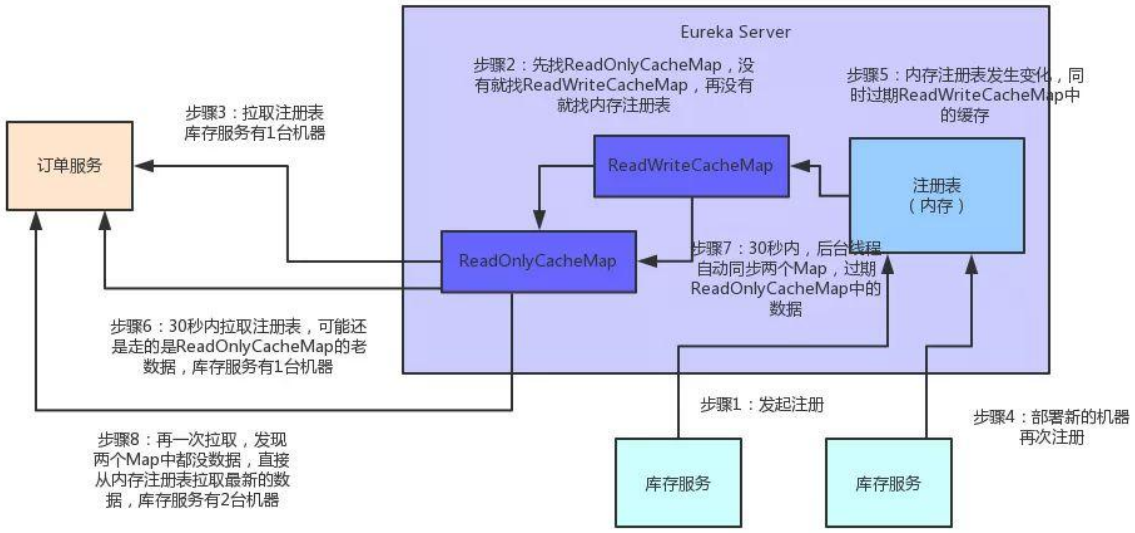
    - 首先从 ReadOnlyCacheMap 里查缓存的注册表。
    - 若没有，就找 ReadWriteCacheMap 里缓存的注册表。
    - 如果还没有，就从内存中获取实际的注册表数据。

- 在注册表发生变更的时候：

    - 会在内存中更新变更的注册表数据，同时过期掉 ReadWriteCacheMap。
    - 此过程不会影响 ReadOnlyCacheMap 提供人家查询注册表。
    - 一段时间内（默认 30 秒），各服务拉取注册表会直接读 ReadOnlyCacheMap
    - 30 秒过后，Eureka Server 的后台线程发现 ReadWriteCacheMap 已经清空了，也会清空 ReadOnlyCacheMap 中的缓存
    - 下次有服务拉取注册表，又会从内存中获取最新的数据了，同时填充各个缓存。

- 多级缓存机制的优点是什么?

    - 尽可能保证了内存注册表数据不会出现频繁的读写冲突问题。
    - 并且进一步保证对 Eureka Server 的大量请求，都是快速从纯内存走，性能极高。
    - 为方便大家更好的理解，同样来一张图，大家跟着图再来回顾一下这整个过程：



四、总结

- 通过上面的分析可以看到，Eureka 通过设置适当的请求频率（拉取注册表 30 秒间隔，发送心跳 30 秒间隔），可以保证一个大规模的系统每秒请求 Eureka Server 的次数在几百次。
- 同时通过纯内存的注册表，保证了所有的请求都可以在内存处理，确保了极高的性能
- 另外, 多级缓存机制，确保了不会针对内存数据结构发生频繁的读写并发冲突操作，进一步提升性能。
- 上述就是 Spring Cloud 架构中，Eureka 作为微服务注册中心可以承载大规模系统每天千万级访问量的原理。

如有收获，请帮忙转发，您的鼓励是作者最大的动力，谢谢！

性能优化之道】每秒上万并发下的 Spring Cloud 参数优化实战

> **作者:中华石杉** 原文地址

目录

- 6、总结全文，回眸再看

一、写在前面

相信不少朋友都在自己公司使用 Spring Cloud 框架来构建微服务架构，毕竟现在这是非常火的一门技术。

如果只是用户量很少的传统 IT 系统，使用 Spring Cloud 可能还暴露不出什么问题。

如果是较多用户量，高峰每秒高达上万并发请求的互联网公司的系统，使用 Spring Cloud 技术就有一些问题需要注意了。

二、场景引入，问题初现

先不空聊原理、理论，来讲一个真实的例子，这是我的一个朋友在创业互联网公司发生过的真实案例。

> ! 朋友 A 的公司做互联网类的创业，组建了一个小型研发团队，上来就用了 Spring Cloud 技术栈来构建微服务架构的系统。 !> 段时间没日没夜的加班，好不容易核心业务系统给做出来了，平时正常 QA 测试没发现什么大毛病，感觉性能还不错，一切都很完美。

然后系统就这么上线了，一开始用户规模很小，注册用户量小几十万，日活几千用户。

每天都有新的数据进入数据库的表中，就这么日积月累的，没想到数据规模居然慢慢吞吞增长到了单表几百万。

这个时候呢，看起来也没太大的毛病，就是有用户反映，系统有些操作，会感觉卡顿几秒钟，会刷不出来页面。

这是为啥呢?

- 核心原因是单表数据量大了一些，达到了几百万。
- 有个别服务，跑的 SQL 比较复杂，一大堆的多表关联
- 并且还没有设计好索引，或者是设计了索引，但无奈一些小弟写了上百行的大 SQL，SQL 实在太复杂了，那么一个 SQL 跑出来好几秒肯定是正常的。

如果大家对微服务框架有点了解的话，应该知道，比如 Feign + Ribbon 组成的服务调用框架，是有接口调用超时这一说的，有一些参数可以设置接口调用的超时时间。

如果你调用一个接口，好几秒刷不出来，人家就超时异常返回，用户就刷不出来页面了。

三、扬汤止沸，饮鸩止渴

一般碰到这种事情，一大坨屎一样的 SQL 摆在那儿，写 SQL 的人过一个月自己都看不懂了，80% 的工程师看着都不愿意去花时间重写和优化。

一是修改的人力成本太高，二是谁敢负担这责任呢?

系统跑的好好的，就是慢了点而已，结果你硬是乱改一通，重构，把系统核心业务流程搞挂了怎么办?

所以，那些兄弟第一反应是：增加超时时间啊！接口慢点可以，但是别超时不响应啊！

咱们让接口执行个几秒把结果返回，用户不就可以刷出来页面了！不用重构系统了啊！轻松 + 愉快！

如何增加呢？很简单，看下面的参数就知道了：

```
ribbon:
  ConnectTimeout: 30000
  ReadTimeout: 30000

hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 60000
```


大家如果看过之前的文章，应该知道，Spring Cloud 里一般会用 hystrix 的线程池来执行接口调用的请求。

如果忘了这一点的，可以回头看看 **《拜托，面试请不要再问我 Spring Cloud 底层原理！》**。

所以设置超时一般设置两个地方，feign 和 ribbon 那块的超时，还有 hystrix 那块的超时。其中后者那块的超时一般必须大于前者。

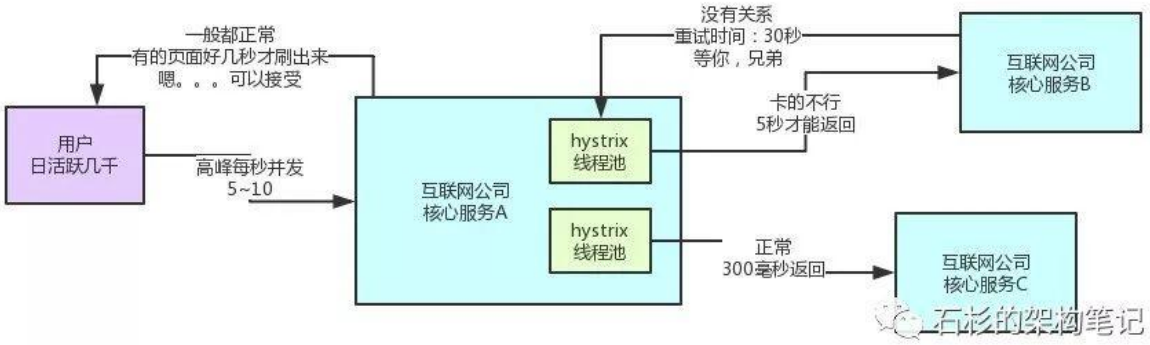
Spring Cloud 玩儿的好的兄弟，可千万别看着这些配置发笑，因为我确实见过不少 Spring Cloud 玩儿的没那么溜的哥们，真的就这么干了。

好了，日子在继续。。。

优化了参数后，看上去效果不错，用户虽然觉得有的页面慢是慢点，但是起码过几秒能刷出来。

这个时候，日活几千的用户量，压根儿没什么并发可言，高峰期每秒最多一二十并发请求罢了。

大家看看下面这张图，感受一下现场氛围：



四、问题爆发，洪水猛兽

随着时间的推移，公司业务高速发展……

那位兄弟的公司，在系统打磨成熟，几万用户试点都 ok 之后，老板立马拿到一轮几千万的融资。

公司上上下下意气风发啊！紧接着就是组建运营团队，地推团队，全国大范围的推广。

总之就是三个字：推！推！推！

这一推不打紧！研发人员在后台系统发现，自己的用户量蹭蹭蹭的直线增长。

注册用户增长了几十倍，突破了千万级别，日活用户也翻了几十倍，在活动之类的高峰期，居然达到了上百万的日活用户量！

幸福的烦恼。。。

为什么这么说？因为用户量上来后，悲剧的事情就发生了。

高峰期每秒的并发请求居然达到了近万的程度，研发团队的兄弟们哪里敢怠慢！在这个过程中，先是紧张的各种扩容服务，一台变两台，两台变四台。

然后数据库主从架构挂上去，读写分离是必须的，否则单个数据库服务器哪能承载那么大的请求！多搞几个从库，扛一下大量的读请求，这样基本就扛住了。

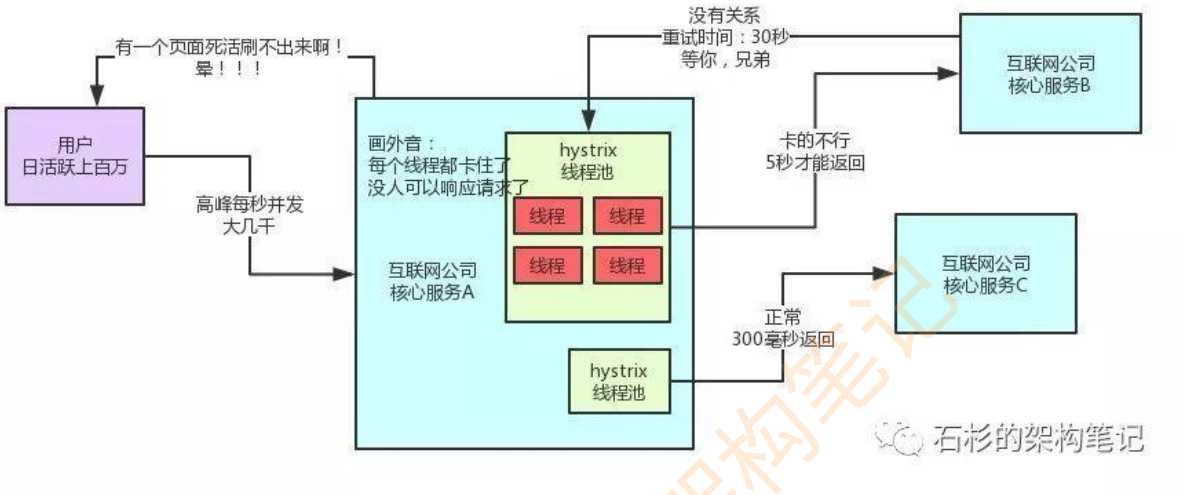
正准备坐下来喝口茶、松口气，更加悲剧的事情就发生了。

在这个过程中，那些兄弟经常会发现高峰期，系统的某个功能页面，突然就整个 hang 死了，就是没法再响应任何请求！所有用户刷新这个页面全部都是无法响应！

这是为什么呢？原因很简单啊！一个服务 A 的实例里，专门调用服务 B 的那个线程池里的线程，总共可能就几十个。每个线程调用服务 B 都会卡住 5 秒钟。

那如果每秒钟过来几百个请求这个服务实例呢？一下子那个线程池里的线程就全部 hang 死了，没法再响应任何请求了。

大家来看看下面这张图，再直观的感受一下这个无助的过程！



这个时候咋办？兄弟们只能祭出程序员最古老的法宝，重启机器！

遇到页面刷不出来，只能重启机器，相当于短暂的初始化了一下机器内的资源。

然后接着运行一段时间，又卡死，再次重启！真是令人崩溃啊！用户们的体验是极差的，老板的心情是愤怒的！

画外音：

其实这个问题本身不大，但如果对 Spring Cloud 没有高并发场景的真实经验，确实可能会跟这帮兄弟一样，搞出些莫名其妙的问题。

比如这个公司，明明应该去优化服务接口性能，结果硬是调大了超时时间。结果导致并发量高了，对那个服务的调用直接 hang 死，系统的核心页面刷不出来，影响用户体验了，这怪谁呢？

五、追本溯源，治标治本

没法子了，那帮兄弟们只能找人求助。下面就是作者全程指导他们完成系统优化的过程。

第一步

关键点，优化图中核心服务 B 的性能。互联网公司，核心业务逻辑，面向 C 端用户高并发的请求，不要用上百行的大 SQL，多表关联，那样单表几百万行数据量的话，会导致一下执行好几

秒。

其实最佳的方式，就是对数据库就执行简单的单表查询和更新，然后复杂的业务逻辑全部放在 java 系统中来执行，比如一些关联，或者是计算之类的工作。

这一步干完了之后，那个核心服务 B 的响应速度就已经优化成几十毫秒了，是不是很开心？从几秒变成了几十毫秒！

第二步

那个超时的时间，也就是上面那段 ribbon 和 hystrix 的超时时间设置。

奉劝各位同学，不要因为系统接口的性能过差而懒惰，搞成几秒甚至几十秒的超时，一般超时定义在 1 秒以内，是比较通用以及合理的。

为什么这么说?

因为一个接口，理论的最佳响应速度应该在 200ms 以内，或者慢点的接口就几百毫秒。

如果一个接口响应时间达到 1 秒 +，建议考虑用缓存、索引、NoSQL 等各种你能想到的技术手段，优化一下性能。

否则你要是胡乱设置超时时间是几秒，甚至几十秒，万一下游服务偶然出了点问题响应时间长了点呢？那你这个线程池里的线程立马全部卡死！

具体 hystrix 的线程池以及超时时间的最佳生产实践，请见下一篇文章：《微服务架构如何保障双 11 狂欢下的 99.99% 高可用》

这两步解决之后，其实系统表现就正常了，核心服务 B 响应速度很快，而且超时时间也在 1 秒以内，不会出现 hystrix 线程池频繁卡死的情况了。

第三步

事儿还没完，你要真觉得两步就搞定了，那还是经验不足。

如果你要是超时时间设置成了 1 秒，如果就是因为偶然发生的网络抖动，导致接口某次调用就是在 1.5 秒呢？这个是经常发生的，因为网络的问题，接口调用偶然超时。

所以此时配合着超时时间，一般都会设置一个合理的重试，如下所示：

```
ribbon:
  ConnectTimeout: 1000
  ReadTimeout: 1000
  OkToRetryOnAllOperations: true
  MaxAutoRetries: 1
  MaxAutoRetriesNextServer: 1
```


设置这段重试之后，Spring Cloud 中的 Feign + Ribbon 的组合，在进行服务调用的时候，如果发现某台机器超时请求失败，会自动重试这台机器，如果还是不行会换另外一台机器重试。

这样由于偶尔的网络请求造成的超时，不也可以通过自动重试避免了？

第四步

其实事儿还没完，如果把重试参数配置了，结果你居然就放手了，那还是没对人家负责任啊！

你的系统架构中，只要涉及到了重试，那么必须上接口的幂等性保障机制。

否则的话，试想一下，你要是对一个接口重试了好几次，结果人家重复插入了多条数据，该怎么办呢？

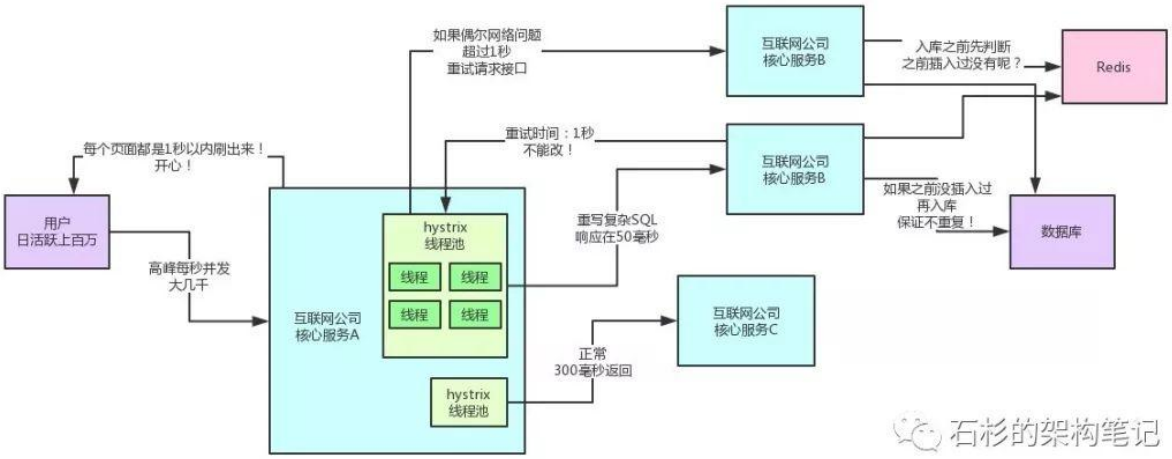
其实幂等性保证本身并不复杂，根据业务来，常见的方案：

- 可以在数据库里建一个唯一索引，插入数据的时候如果唯一索引冲突了就不会插入重复数据
- 或者是通过 redis 里放一个唯一 id 值，然后每次要插入数据，都通过 redis 判断一下，那个值如果已经存在了，那么就不要插入重复数据了。

类似这样的方案还有一些。总之，要保证一个接口被多次调用的时候，不能插入重复的数据。

六、总结全文，回眸再看

有图有真相！老规矩，最后给大家上一张图，最终优化后的系统表现大概是长下面这样子的。



如有收获，请帮忙转发，您的鼓励是作者最大的动力，谢谢！

微服务架构如何保障双11狂欢下的99.99%高可用

> **作者:中华石杉** 原文地址

目录

一、概述

上一篇文章讲了一个朋友公司使用 Spring Cloud 架构遇到问题的一个真实案例，虽然不是什么大的技术问题，但如果对一些东西理解的不深刻，还真会犯一些错误。

如果没看过上一篇文章的朋友，建议先看看：**【双 11 狂欢的背后】微服务注册中心如何承载大型系统的千万级访问?** 因为本文的案例背景会基于上一篇文章。

这篇文章我们来聊聊在微服务架构中，到底如何保证整套系统的高可用?

排除掉一些基础设施的故障，比如说 Redis 集群挂了，Elasticsearch 集群故障了，MySQL 宕机。

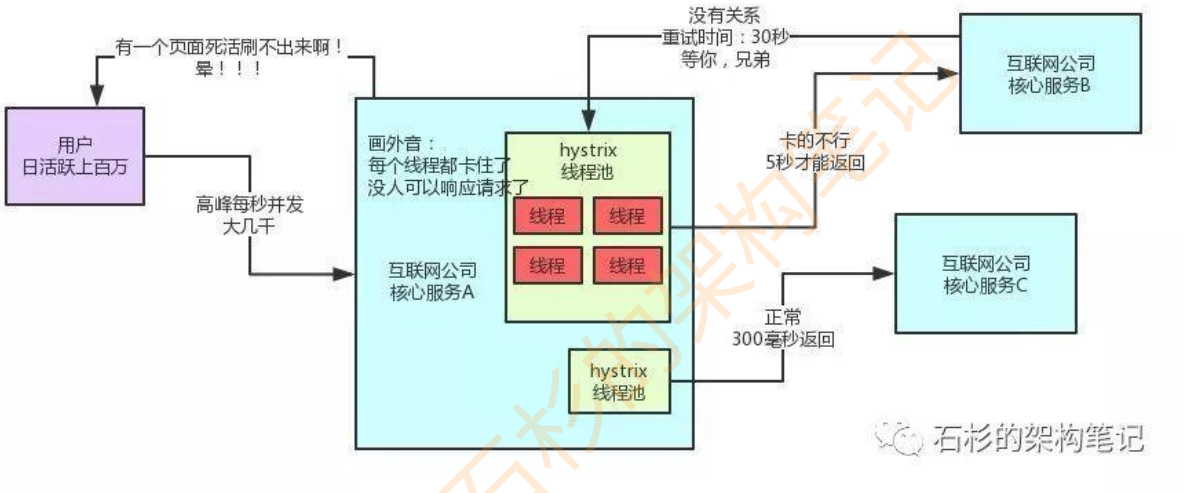
微服务架构本身最最核心的保障高可用的措施，就是两点：

- 一个是基于 Hystrix 做资源隔离以及熔断；
- 另一个是做备用降级方案。
- 如果资源隔离和降级都做的很完善，那么在双 11 这种高并发场景下，也许可能会出现个别的服务故障，但是绝不会蔓延到整个系统全部宕机。

这里大家如果忘了如何基于 hystrix 做资源隔离、熔断以及降级的话，可以回顾一下之前的文章： **拜托！面试请不要再问我 Spring Cloud 底层原理**

二、业务场景介绍

大家首先回顾一下下面这张图，这是上篇文章中说到的一个公司的系统。



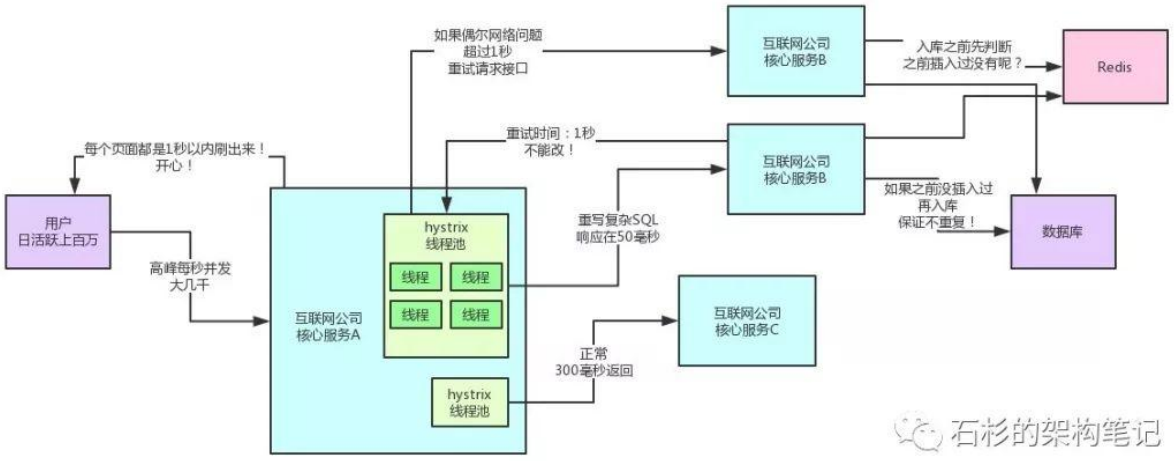
如上图，核心服务 A 调用了核心服务 B 和 C，在核心服务 B 响应过慢时，会导致核心服务 A 的某个线程池全部卡死。

但是此时因为你用了 hystrix 做了资源隔离，所以核心服务 A 是可以正常调用服务 C 的，那么就可以保证用户起码是可以使用 APP 的部分功能的，只不过跟服务 B 关联的页面刷不出来，功能无法使用罢了。

当然这种情况在生产系统中，是绝对不被允许的，所以大家不要让上述情况发生。

在上一篇文章中，我们最终把系统优化成了下图这样：

- 要保证一个 hystrix 线程池可以轻松处理每秒钟的请求
- 同时还有合理的超时时间设置，避免请求太慢卡死线程。



三、线上经验—如何设置 Hystrix 线程池大小

好，现在问题来了，在生产环境中，我们到底应该如何设置服务中每个 hystrix 线程池的大小?

下面是我们在线上经过了大量系统优化后的生产经验总结：

假设你的服务 A，每秒钟会接收 30 个请求，同时会向服务 B 发起 30 个请求，然后每个请求的响应时长经验值大概在 200ms，那么你的 hystrix 线程池需要多少个线程呢?

计算公式是：30（每秒请求数量） * 0.2（每个请求的处理秒数） + 4（给点缓冲 buffer） = 10（线程数量）。

如果对上述公式存在疑问，不妨反过来推算一下，为什么 10 个线程可以轻松抗住每秒 30 个请求?

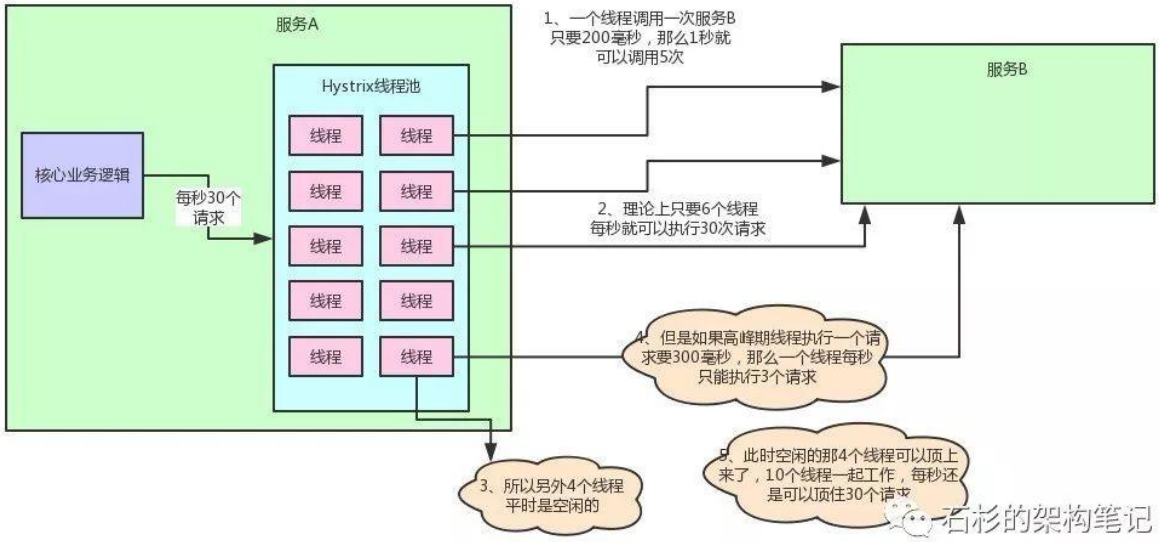
一个线程 200 毫秒可以执行完一个请求，那么一个线程 1 秒可以执行 5 个请求，理论上，只要 6 个线程，每秒就可以执行 30 个请求。

也就是说，线程里的 10 个线程中，就 6 个线程足以抗住每秒 30 个请求了。剩下 4 个线程都在玩儿，空闲着。

那为啥要多搞 4 个线程呢？很简单，因为你要留一点 buffer 空间。

万一在系统高峰期，系统性能略有下降，此时不少请求都耗费了 300 多毫秒才执行完，那么一个线程每秒只能处理 3 个请求了，10 个线程刚刚好勉强可以 hold 住每秒 30 个请求。所以你必须多考虑留几个线程。

老规矩，给大家来一张图，直观的感受一下整个过程。



四、线上经验—如何设置请求超时时间

线程数量 OK 了，那么请求的超时时间设置为多少？答案是 300 毫秒。

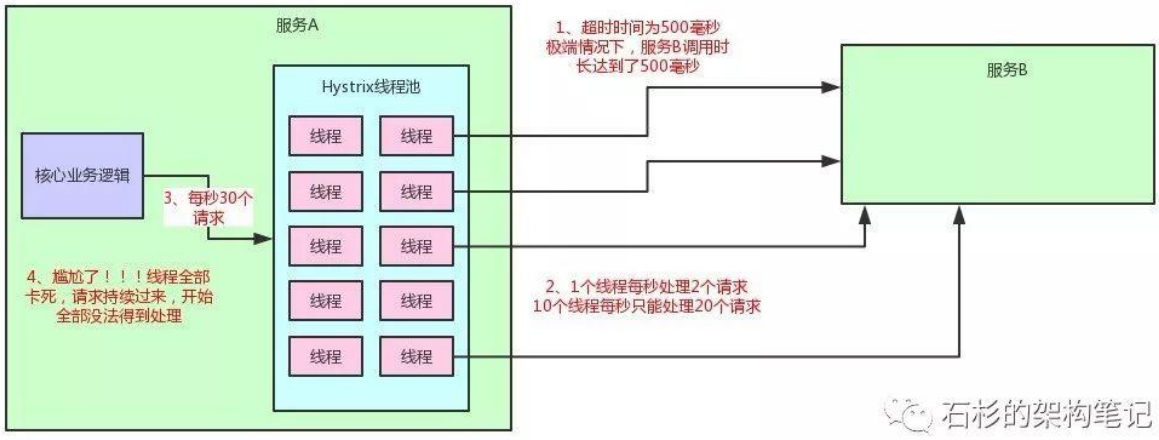
为啥呢？很简单啊，如果你的超时时间设置成了 500 毫秒，想想可能会有什么后果？

考虑极端情况，如果服务 B 响应变慢，要 500 毫秒才响应，你一个线程每秒最多只能处理 2 个请求了，10 个线程只能处理 20 个请求。

而每秒是 30 个请求过来，结局会如何？

咱们回看一下第一张图就知道了，大量的线程会全部卡死，来不及处理那么多请求，最后用户会刷不出来页面。

还是有点不理解？再给你一张图，让你感受一下这个不合理的超时时间导致的问题！



如果你的线程池大小和超时时间没有配合着设置好，很可能会导致服务 B 短暂的性能波动，瞬间导致服务 A 的线程池卡死，里面的线程要卡顿一段时间才能继续执行下一个请求。

哪怕一段时间后，服务 B 的接口性能恢复到 200 毫秒以内了，服务 A 的线程池里卡死的状况也要好一会儿才能恢复过来。

你的超时时间设置的越不合理，比如设置的越长，设置到了 1 秒、2 秒，那么这种卡死的情况就需要越长的时间来恢复。

所以说，此时你的超时时间得设置成 300 毫秒，保证一个请求 300 毫秒内执行不完，立马超时返回。

这样线程池里的线程不会长时间卡死，可以有条不紊的处理多出来的请求，大不了就是 300 毫秒内处理不完立即超时返回，但是线程始终保持可以运行的状态。

这样当服务 B 的接口性能恢复到 200 毫秒以内后，服务 A 的线程池里的线程很快就可以恢复。

这就是生产系统上的 hystrix 参数设置优化经验，你需要考虑到各种参数应该如何设置。

否则的话，很可能会出现上文那样的情况，用了高大上的 Spring Cloud 架构，结果跟黑盒子一样，莫名其妙系统故障，各种卡死，宕机什么的。

好了，我们继续。如果现在这套系统每秒有 6000 请求，然后核心服务 A 一共部署了 60 台机器，每台机器就是每秒会收到 100 个请求，那么此时你的线程池需要多少个线程?

很简单，10 个线程抗 30 个请求，30 个线程抗 100 请求，差不多了吧。

这个时候，你应该知道服务 A 的线程池调用服务 B 的线程池分配多少线程了吧？超时时间如何设置应该也知道了！

其实这个东西不是固定死的，但是你要知道他的计算方法。

根据服务的响应时间、系统高峰 QPS、有多少台机器，来计算出来，线程池的大小以及超时时间！

五、服务降级

设置完这些后，就应该要考虑服务降级的事了。

如果你的某个服务挂了，那么你的 hystrix 会走熔断器，然后就会降级，你需要考虑到各个服务的降级逻辑。

举一些常见的例子：

- 如果查询数据的服务挂了，你可以查本地的缓存
- 如果写入数据的服务挂了，你可以先把这个写入操作记录日志到比如 mysql 里，或者写入 MQ 里，后面再慢慢恢复

- 如果 redis 挂了，你可以查 mysql
- 如果 mysql 挂了，你可以把操作日志记录到 es 里去，后面再慢慢恢复数据。

具体用什么降级策略，要根据业务来定，不是一成不变的。

六、总结

最后总结一下，排除那些基础设施的故障，你要玩儿微服务架构的话，需要保证两点：

- 首先你的 hystrix 资源隔离以及超时这块，必须设置合理的参数，避免高峰期，频繁的 hystrix 线程卡死
- 其次，针对个别的服务故障，要设置合理的降级策略，保证各个服务挂了，可以合理的降级，系统整体可用！

如有收获，请帮忙转发，您的鼓励是作者最大的动力，谢谢！

尴尬了！Spring Cloud 微服务注册中心 Eureka 2.x 停止维护了咋办?

> **作者:中华石杉** 原文地址

目录

1、Eureka 官方宣布 2.x 不再开源

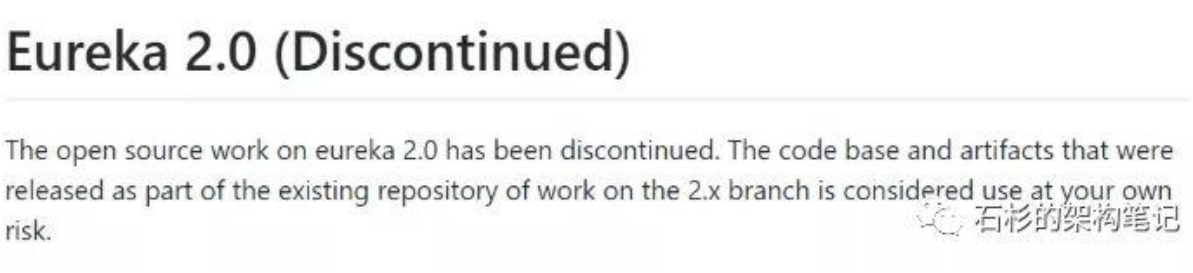
之前写过一篇文章：**拜托！面试请不要再问我Spring Cloud底层原理**，文章介绍了 Spring Cloud 微服务技术体系的一些基础知识和架构原理。

如果对 Spring Cloud 微服务技术体系有一定了解了之后，肯定就知道 Spring Cloud 最开始原生支持和推荐的服务注册中心是国外的一个视频网站 Netflix 开源的 Eureka。

这个 Eureka 呢，又分成了所谓的 1.x 版本和 2.x 版本，之前在国内比较常用在生产环境中的都是 Eureka 的 1.x 版本。

然后 Netflix 这个公司本身一直在做 Eureka 2.x 版本，结果做着做着，大家万众瞩目很期待的时候。。。

2018 年 7 月，人家官方就突然宣布 Eureka 2.x 停止开源计划了，具体如下：



用中文给大家翻译一下，这里的意思就是说：Eureka 2.0 的开源工作已经停止了，如**果你要用 Eureka 2.x 版本的代码来部署到生产环境的话，一切后果请自负。**

大概就是这个意思，就是不打算把这个事儿做大做强下去了。

当然现在其实 Eureka 1.x 的版本也有不少公司在生产环境用，而且基本也还算能用的状态，基本功能还算正常，应付很多常规的场景也足够了。

但是现实就是这个声明发出来，让大伙都心里一凉，怎么感觉这个这个 Eureka 有点不太靠谱了呢，咱还敢继续用么，没错，很多小伙伴就是这感觉。

2、互联网大厂的基础架构：自研服务注册中心 这里给大家说一句题外话，BAT、TMD 等一线互联网公司，包括一些有一定研发实力的中大型互联网公司，都是自研了微服务技术架构中的服务注册中心。或者是基于开源的 Eureka 之类的项目来做二次开发，自行优化里面的架构，解决遇到的问题。

所以对于有基础架构团队的公司而言，这个问题相对来说还没那么严重。

因为大厂的基础架构团队，完全可以把常见的开源服务注册中心的源码都深入看一遍，然后经过大量严谨的测试找到各个开源技术的优点和缺点。最后决定是从 0 开始自研一个服务注册中心，还是说基于某个开源的技术来进行二次开发和优化。

比如说 Eureka 1.x 作为一个服务注册中心，**有一个非常典型的架构问题。**

虽然他可以部署集群架构，但是集群中每个 Eureka 实例都是对等的。每个 Eureka 实例都包含了全部的服务注册表，每个 Eureka 实例接收到了服务注册 / 下线等请求的时候，会同步转发给集群中其他的 Eureka 实例，实现集群数据同步。

大家看下面的图，大概就是一个示意。

石杉的架构笔记

那么这里就有一个问题了：如果是支持超大规模的服务集群，这样的模式能行么？

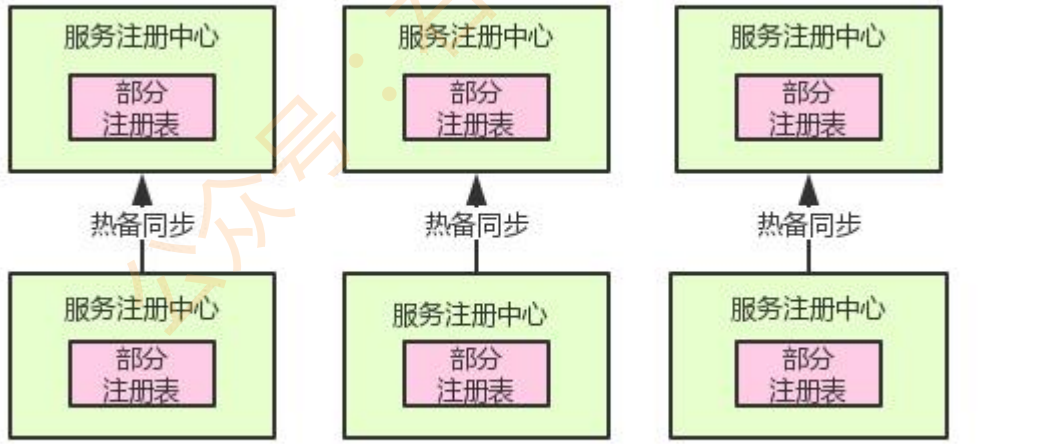
每台机器的内存是有限的，集群里的服务数量越来越多，可能有几十万个服务实例在运行，那么服务注册表越来越大，最后超过单机内存支撑的极限怎么办？

这个时候如果自己研发服务注册中心，就可以参考大数据领域的 Hadoop 的架构思想。

Hadoop 的设计思想是把注册表分片存储，分布式存储在多台机器上，每台机器存储部分注册表数据。

然后每个 Server 可以加上一个从节点做热备份，避免单机挂掉导致注册 表数据丢失。

我们来看看架构图，如下所示：



石杉的架构笔记

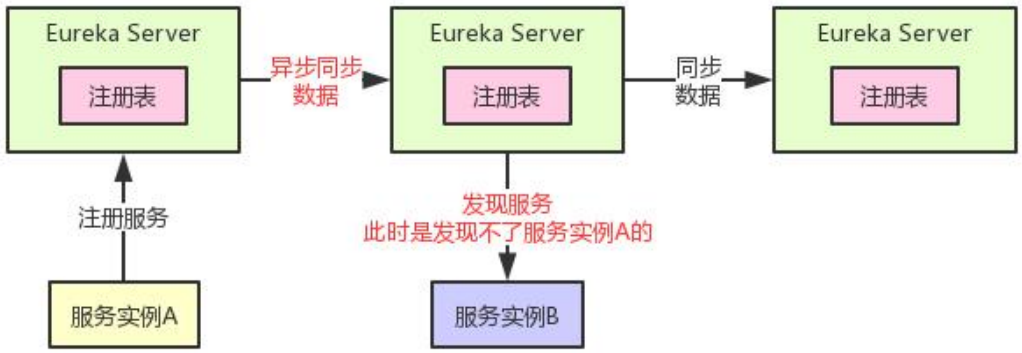
实际在生产环境使用 Eureka 的时候，你还会碰到很多现实的问题。

比如说上面讲了，Eureka 本身是基于简单的同步机制实现集群架构的，但是这里在集群之间进行同步的时候，其实是异步进行的，采用的是最终一致性的协议。

这就可能会导致说，你某个服务注册到了一个 Eureka Server 实例上去，但是他需要异步复制到其他的 Eureka Server，这中间是需要时间的。

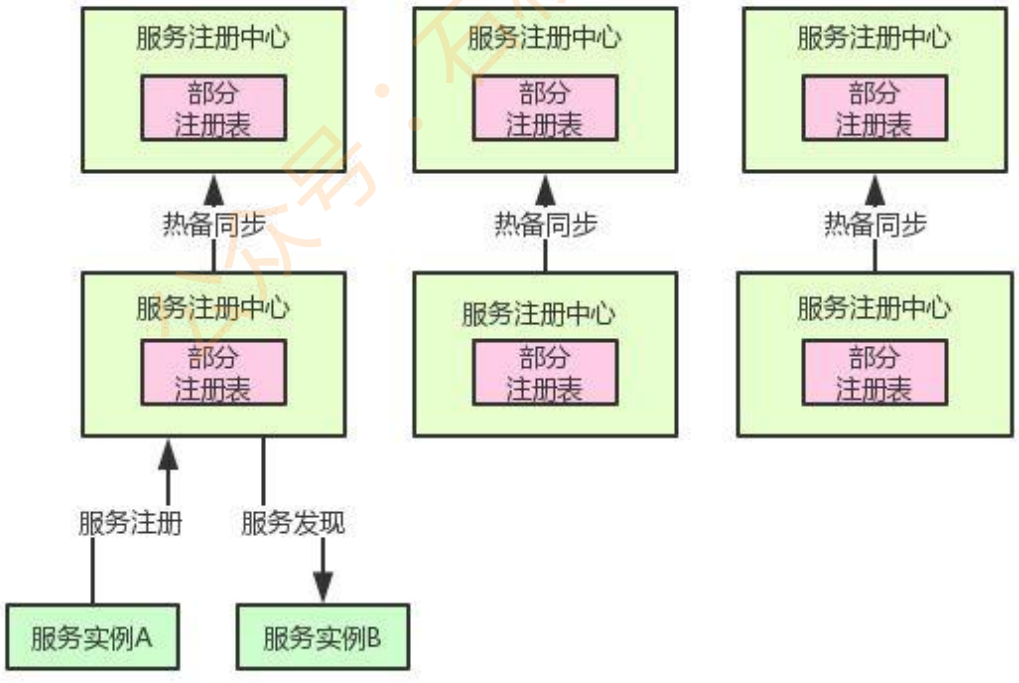
所以可能导致其他的 Eureka Server 是看不到那个刚新注册的服务实例的。

大家看下面的图，就示意了这个问题：



但是如果是采取了类似 Hadoop 的那种数据分片思想的话，一个注册表数据分片就在一台机器上，由这台机器负责提供服务的注册和发现，那么此时就可以实现强一致的效果。

也就是说，只要你注册了，立马就会被别人发现，如下图。



这里只是说其中几个例子罢了，改造开源系统的思路是很多的，实际上大厂完全可以对开源技术做很多的自研、定制和改造，解决线上的生产问题，让服务注册中心朝着他们心里期望的效果去发展，所以对他们来说其实问题并不大。

3、中小公司的其他选择：Consul

只是对于很多中小型公司而言，可能本身没有基础架构团队的支撑，或者是没有过多的人力物力投入到自研中间件、开源系统二次开发中去。

那么此时就可以考虑选择其他的开源服务注册中心的技术了，比如 Spring Cloud 同样支持的 Consul 就是目前很多公司的选择。

这儿咱们简单介绍一下 Consul，后面可以考虑再写文章介绍介绍 Consul 的架构原理和使用什么的，大家看一下，可以作为一个服务注册中心技术选型的参考：

服务注册与发现

- Consul 当然是可以作为服务注册中心的了，可以用做微服务架构的服务注册和发现。
- 同时这里可以先给大家说一下，Consul 的服务注册机制选择的是基于 Daft 协议的强一致，没有像 Eureka 那样使用最终一致的效果。

健康检查

- Consul 可以支持非常强大的健康检查的功能，啥叫健康检查?
- 简单来说就是不停的发请求给你的服务检查他到底死了没有，目前是否还健康，这个就是叫做健康检查。

kv 存储

- Consul 不光支持服务注册和发现，居然还可以支持简单的 kv 存储。
- 他可以让你用 key-value 对的形式存放一些信息以及提取查询，是不是很神奇?

安全的服务通信

- Consul 支持让你的服务之间进行授权来限制哪些服务可以通信和连接

多数据中心支持

其实说实话，在做技术选型的时候，非常关键的一点，就是看社区是否活跃。

所以虽然上面说了很多，但是其实大家完全可以看一眼下面的 Eureka Github 和 Consul Github 的更新活跃度的对比。

我们明显会发现，Eureka 1.x 版本最近的更新都在几个月前甚至几年前，但是 Consul 最近的更新很多都是几天前的。

所以本身 Spring Cloud 官方技术栈也是支持 Consul 的，Eureka 开源社区慢慢不再活跃之后，自然很多中小公司开始选择使用功能更加强大，而且社区更新也更加活跃的 Consul 作为服务注册

中心了，这也是一个不错的选择。

如何优化 Spring Cloud 微服务注册中心架构?

> **作者:中华石杉** 原文地址

目录

> ! “ 上一篇文章：尴尬了！Spring Cloud 服务注册中心 Eureka 2.x 停止维护了咋办？，我们给大家说了一下 Spring Cloud 服务注册中心的一些问题。

如果用 Eureka 作为其注册中心的话，很多同学都觉得心里没底，所以现在很多公司都开始使用 Consul 作为其注册中心。

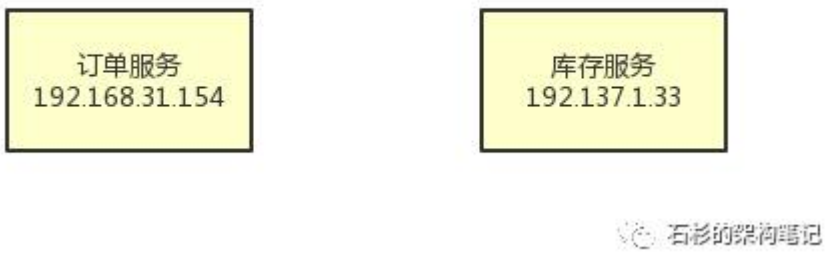
那么这篇文章我们就来给大家说说：Consul 这种服务注册中心的架构是如何设计的?

1、再回顾：什么是服务注册中心?

先回顾一下什么叫做服务注册中心?

顾名思义，假设你有一个分布式系统，里面包含了多个服务，部署在不同的机器上，然后这些不同机器上的服务之间要互相调用。

举个现实点的例子吧，比如电商系统里的订单服务需要调用库存服务，如下图所示。

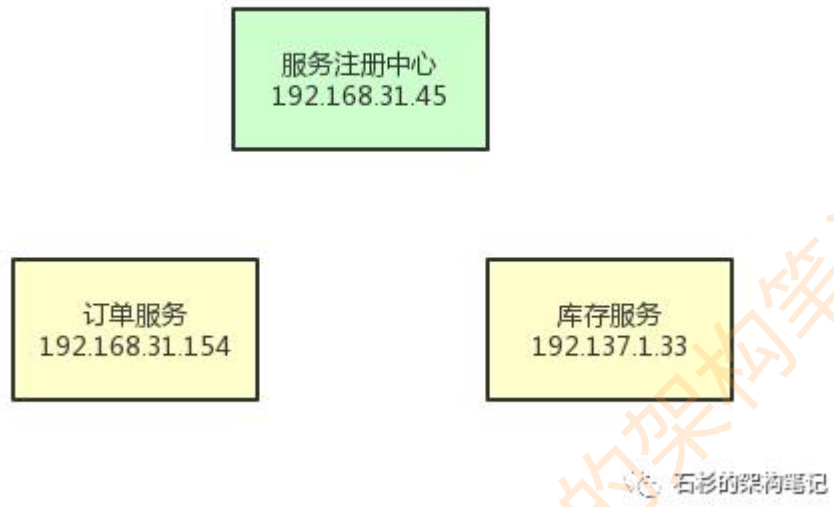


现在的问题在于，订单服务在 192.168.31.154 这台机器上，库存服务在 192.137.1.33 这台机器上。

现在订单服务是想要调用库存服务，但是他并不知道库存服务在哪台机器上啊！毕竟人家都是在不同机器上的。

所以这个时候就需要服务注册中心出场了，这个时候你的系统架构中需要引入独立部署在一台机器上的服务注册中心，如下图所示。

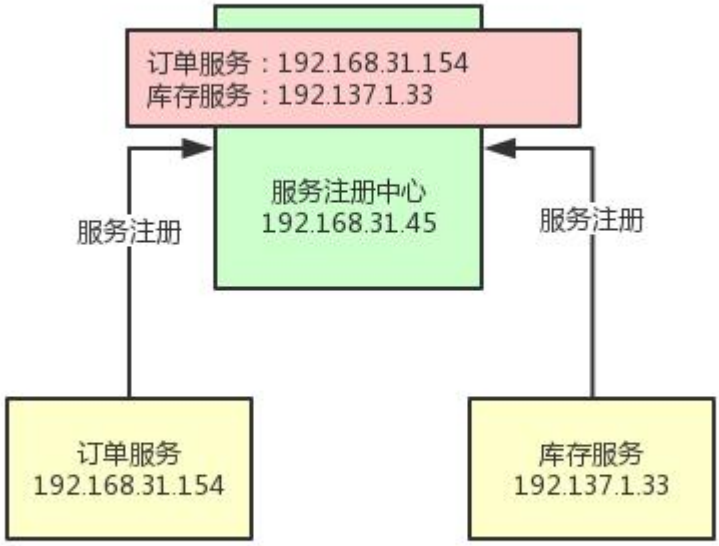
然后订单服务、库存服务之类的兄弟，都需要配置上服务注册中心部署在哪台机器上，比如192.168.31.45 这台机器。



接着订单服务、库存服务他们自己启动的时候，就得发送请求到到服务注册中心上去进行服务注册。

也就是说，得告诉服务注册中心，自己是哪个服务，然后自己部署在哪台机器上。

然后服务注册中心会把大家注册上来的信息放在注册表里，如下图。

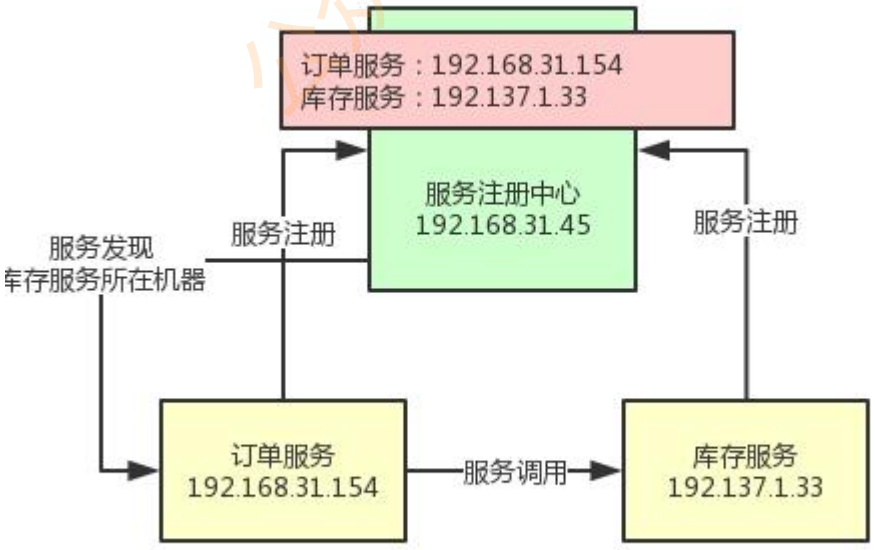


接着订单服务假如想要调用库存服务，那么就找服务注册中心问问：能不能告诉我库存服务部署在哪台机器上？

服务注册中心是知道这个信息的，所以就会告诉订单服务：库存服务部署在 192.1371.133 这台机器上，你就给这台机器发送请求吧。

然后，订单服务就可以往库存服务的那台机器发送请求了，完成了服务间的调用。

整个过程，如下图所示：



上述就是服务注册中心的作用、地位以及意义，现在大家应该知道服务注册中心的作用了吧。

好！接着我们就来看看 Consul 作为服务注册中心，他的架构设计原理是什么？

2、Consul 服务注册中心的整体架构

如果要基于 Consul 作为服务注册中心，那么首先必须在每个服务所在的机器上部署一个 Consul Agent，作为一个服务所在机器的代理。

然后还得在多台机器上部署 Consul Server，这就是核心的服务注册中心。

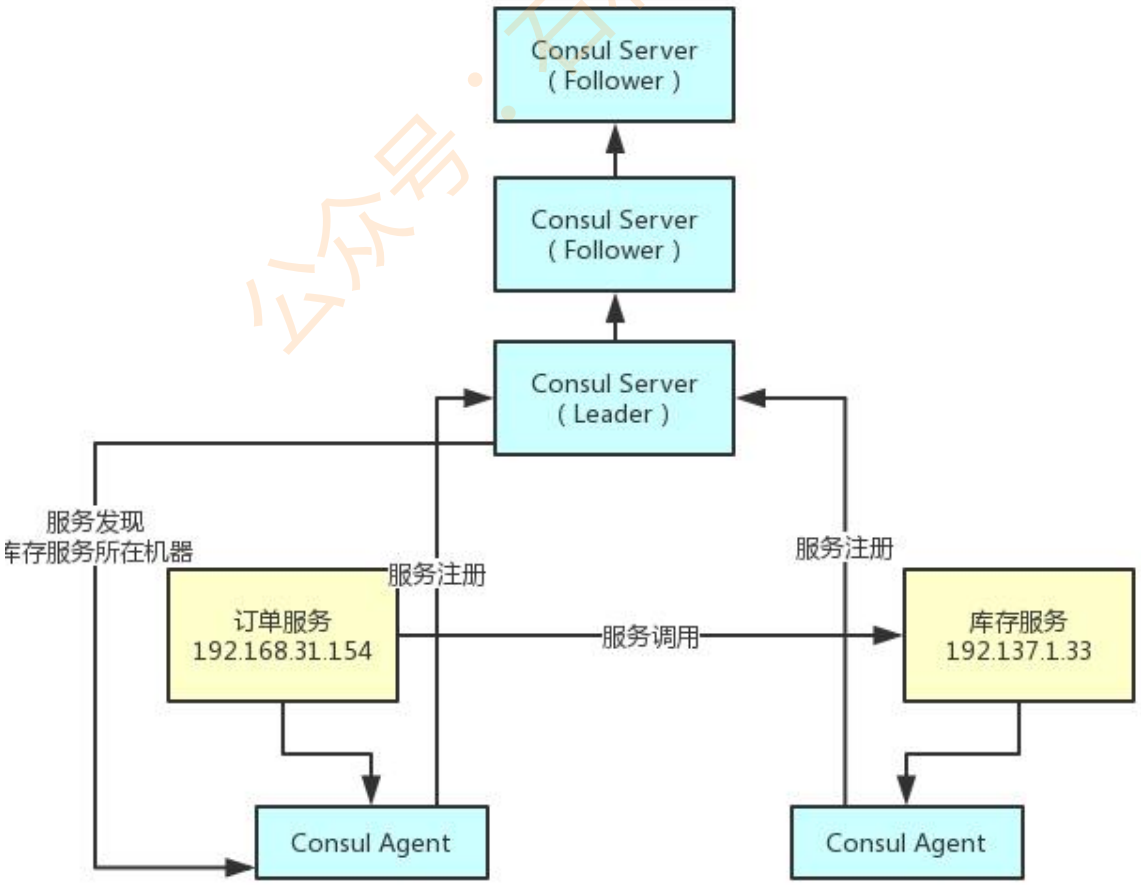
这个 Consul Agent 可以用来收集你的服务信息然后发送给 Consul Server，还会对你的服务不停的发送请求检查他是否健康。

然后你要发现别的服务的时候，Consul Agent 也会帮你转发请求给 Consul Server，查询其他服务所在机器。

Consul Server 一般要求部署 3~5 台机器，以保证高可用以及数据一致性。

他们之间会自动实现数据同步，而且 Consul Server 集群会自动选举出一台机器作为 leader，其他的 Consul Server 就是 follower。

咱们看下面的图，先感受一下这个 Consul 他整体的架构。



3、Consul 如何通过 Raft 协议实现强一致性?

首先上篇文章：尴尬了！Spring Cloud 微服务注册中心 Eureka 2.x 停止维护了咋办？已经说了，Eureka 服务注册中心是不保证数据一致性的。

这样的话，很可能你注册的服务，其他人是发现不了的，或者很迟才能发现。

OK，那么这里就来讨论一下 Consul 是如何实现数据一致性的。

首先，大家知道 Consul Server 是部署集群的，而且他会选举出来一台 Server 作为 Leader。

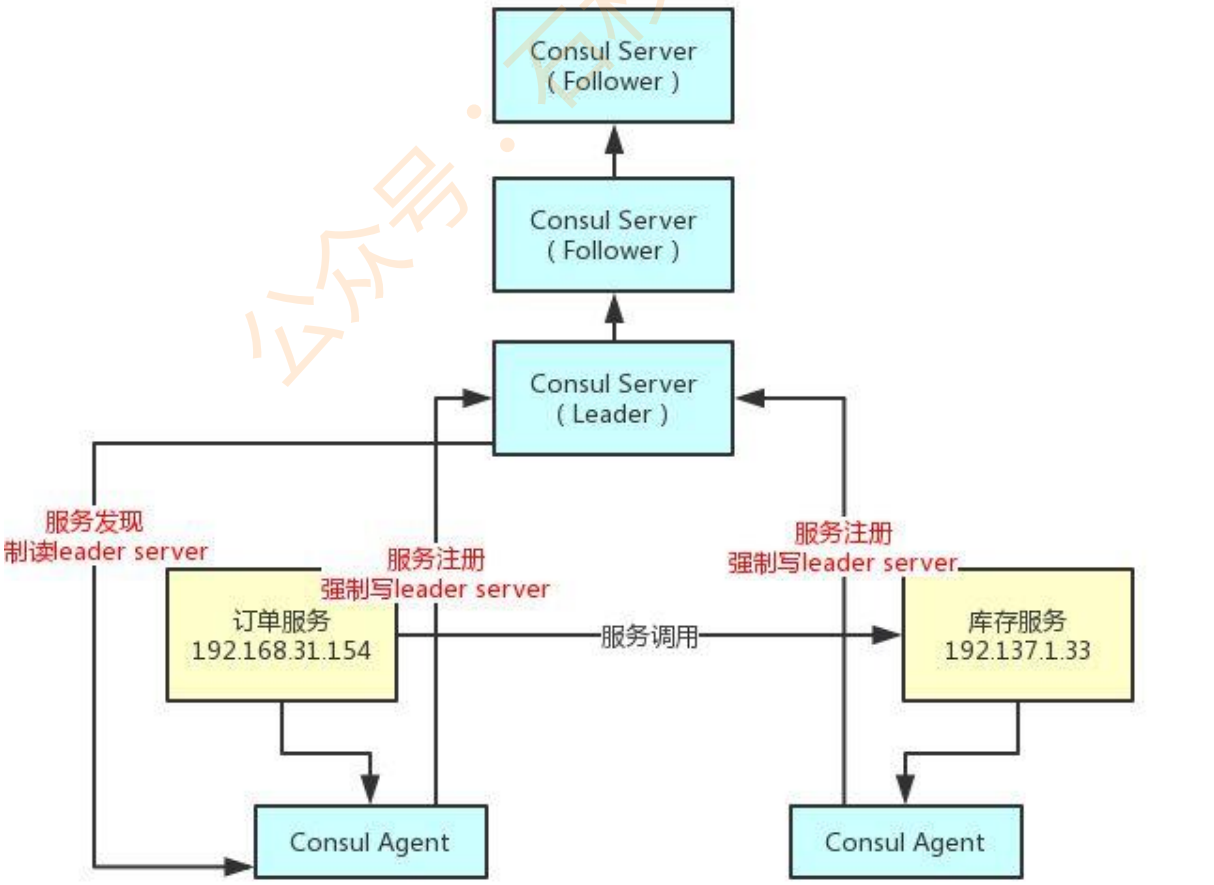
接下来各个服务发送的注册请求都会落地给 Leader，由 Leader 同步给其他 Follower。

所以首先第一点，Leader Server 是绝对有最新的服务注册信息的，是不是?

比如库存服务发起注册了，那么 Leader Server 上一定有库存服务的注册信息。

接着如果比如订单服务要发现库存服务的话，这个查询请求会发送给 Leader Server。

这样服务注册和发现，都是通过一台 Leader Server 来进行的，就可以保证服务注册数据的强一致性了，大家看下图。



接着大家想，假如说库存服务在注册的时候数据刚写到 Leader Server，结果 Leader Server 就宕机了，这时候怎么办？

那么此时这条注册数据就丢失了，订单服务就没法发现那个库存服务了。没关系，**这里 Consul 会基于 Raft 协议来解决这个问题。**

首先，库存服务注册到 Leader Server 的时候，会采取 Raft 协议，要求必须让 Leader Server 把这条注册数据复制给大部分的 Follower Server 才算成功。

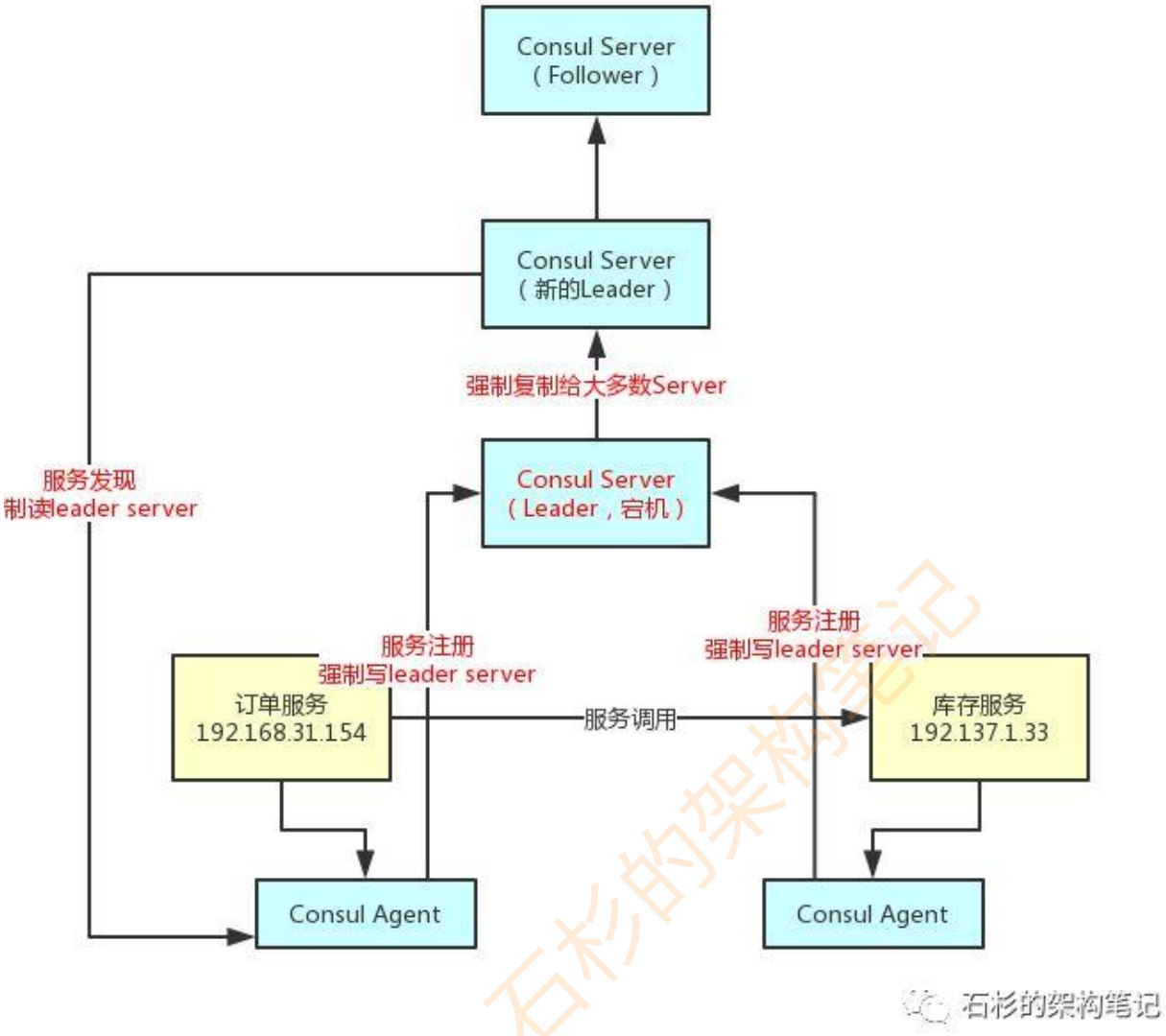
这就保证了，如果你认为自己注册成功了，那么必然是多台 Consul Server 都有这条注册数据了。

如果你刚发送给 Leader Server 他自己就宕机了，那么这次注册会认为失败。

此时，Consul Server 集群会重新选举一个 Leader Server 出来，你需要再次重新注册。

这样就可以保证你注册成功的数据绝对不会丢，然后别人发现服务的时候一定可以从 Leader Server 上获取到最新的强一致的注册数据。

整个过程，如下图所示：



上面的图就可以看到，只要你注册的时候基于 Raft 协议强制同步到大多数 Server，哪怕是 Leader 挂了，也会选举新的 Leader。

这样就可以让别人从新的 Leader Server 来发现你这个服务，所以数据是绝对强一致的。

4、Consul 如何通过 Agent 实现分布式健康检查?

最后说说 Consul 是如何通过各个服务机器上部署 Agent 来实现分布式健康检查的。

集中式的心跳机制，比如传统的 Eureka，是让各个服务都必须每隔一定时间发送心跳到 Eureka Server。

如果一段时间没收到心跳，那么就认为这个服务宕机了。

但是这种集中式的心跳机制会对 Eureka Server 造成较大的心跳请求压力，实际上平时 Eureka Server 接收最多的请求之一就是成千上万服务发送过来的心跳请求。

所以 Consul 在这块进行了架构优化，引入了 Agent 概念。

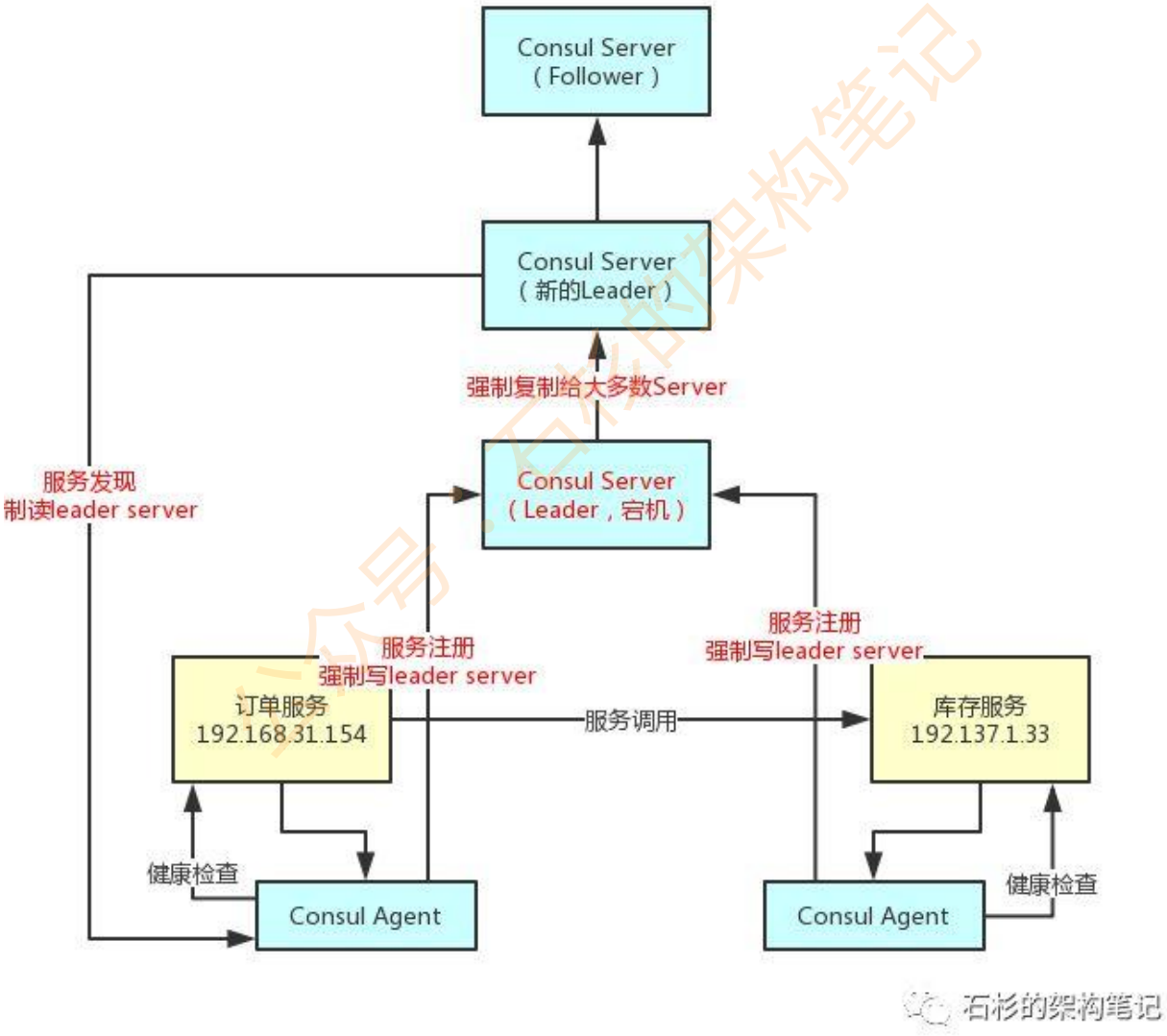
每个机器上的 Consul Agent 会不断的发送请求检查服务是否健康，是否宕机。如果服务宕机了，那么就会通知 Consul Server。

怎么样？是不是发现各个服务自己不用再发送心跳请求去 Server 了？减小了 Server 这部分的压力吧？

没错，这就是 Consul 基于 Agent 实现的分布式健康检查机制，可以大幅度的减小 Server 端的压力。

这样一来，哪怕你就部署个三五台机器，可以轻松支持成千上万个服务。

咱们再来一张图，一起来看看：



拜托，面试请不要再问我TCC分布式事务的实现原理！

> 作者:中华石杉 原文地址

目录

一、写在前面

之前网上看到很多写分布式事务的文章，不过大多都是将分布式事务各种技术方案简单介绍一下。很多朋友看了不少文章，还是不知道分布式事务到底怎么回事，在项目里到底如何使用。

所以咱们这篇文章，就用大白话 + 手工绘图，并结合一个电商系统的案例实践，来给大家讲清楚到底什么是 TCC 分布式事务。

首先说一下，这里可能会牵扯到一些 Spring Cloud 的原理，如果有不太清楚的同学，可以参考之前的文章：**《拜托，面试请不要再问我 Spring Cloud 底层原理！》**。

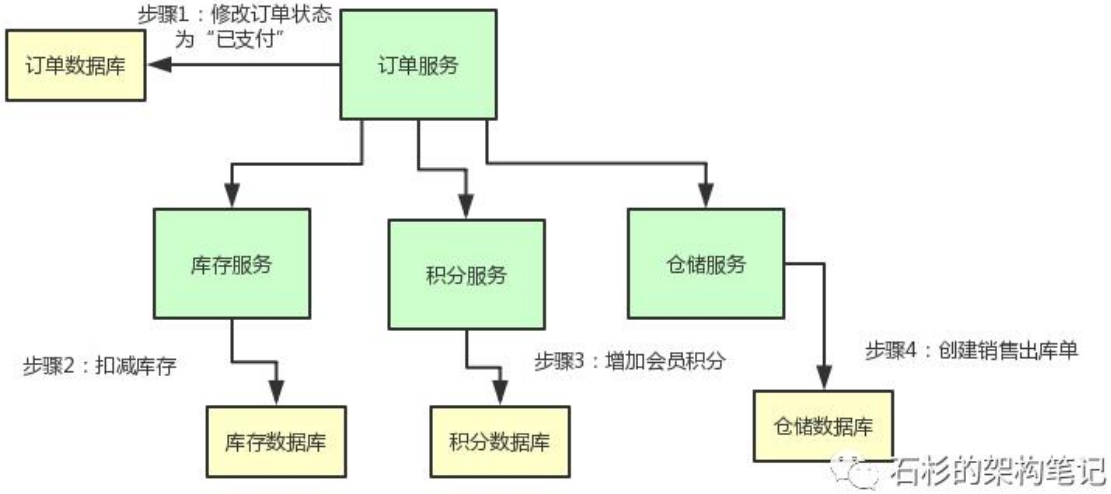
二、业务场景介绍

咱们先来看看业务场景，假设你现在有一个电商系统，里面有一个支付订单的场景。

那对一个订单支付之后，我们需要做下面的步骤：

- 更改订单的状态为 “已支付”
- 扣减商品库存
- 给会员增加积分
- 创建销售出库单通知仓库发货

这是一系列比较真实的步骤，无论大家有没有做过电商系统，应该都能理解。



三、进一步思考

好，业务场景有了，现在我们要更进一步，实现一个 TCC 分布式事务的效果。

什么意思呢？也就是说，订单服务 - 修改订单状态，库存服务 - 扣减库存，积分服务 - 增加积分，仓储服务 - 创建销售出库单。

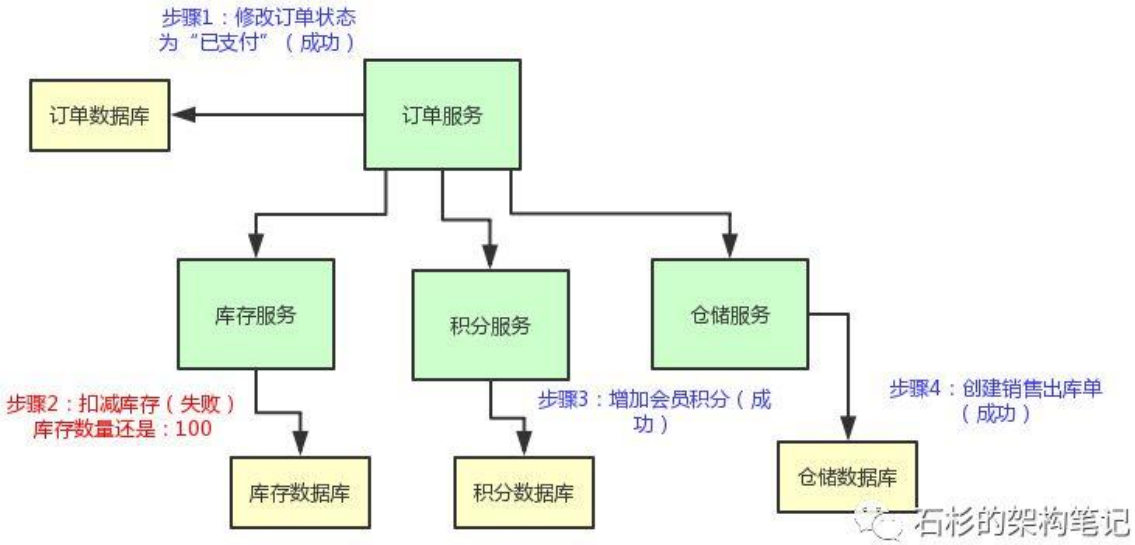
上述这几个步骤，要么一起成功，要么一起失败，必须是一个整体性的事务。

举个例子，现在订单的状态都修改为 “已支付” 了，结果库存服务扣减库存失败。那个商品的库存原来是 100 件，现在卖掉了 2 件，本来应该是 98 件了。

结果呢？由于库存服务操作数据库异常，导致库存数量还是 100。这不是在坑人么，当然不能允许这种情况发生了！

但是如果你不用 TCC 分布式事务方案的话，就用个 Spring Cloud 开发这么一个微服务系统，很有可能会干出这种事儿来。

我们来看看下面的这个图，直观的表达了上述的过程。

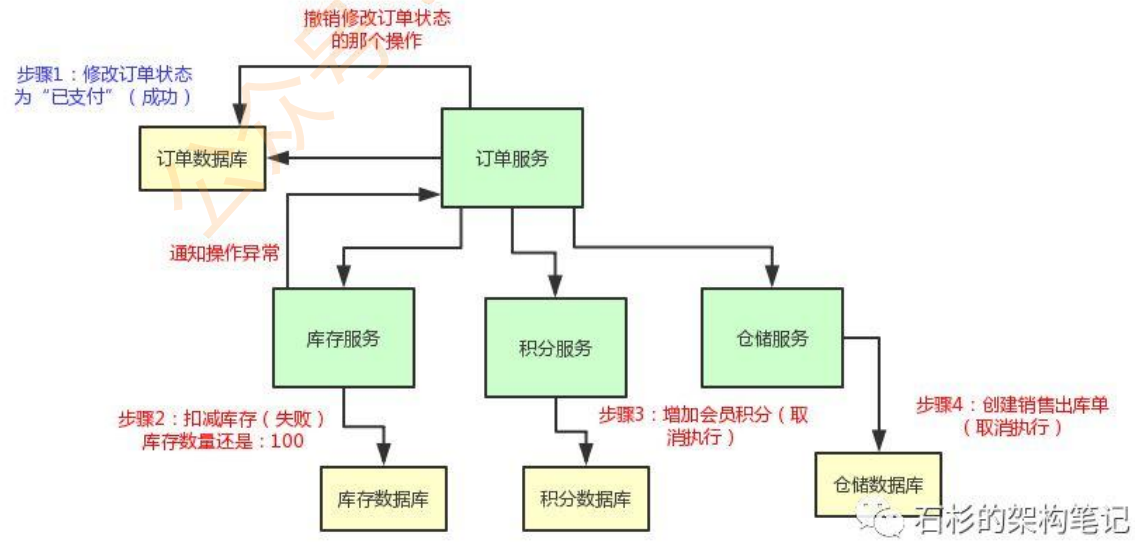


所以说，我们有必要使用 TCC 分布式事务机制来保证各个服务形成一个整体性的事务。

上面那几个步骤，要么全部成功，如果任何一个服务的操作失败了，就全部一起回滚，撤销已经完成的操作。

比如说库存服务要是扣减库存失败了，那么订单服务就得撤销那个修改订单状态的操作，然后得停止执行增加积分和通知出库两个操作。

说了那么多，老规矩，给大家上一张图，大伙儿顺着图来直观的感受一下。



四、落地实现 TCC 分布式事务

那么现在到底要如何来实现一个 TCC 分布式事务，使得各个服务，要么一起成功？要么一起失败呢？

大家稍安勿躁，我们这就来一步一步的分析一下。咱们就以一个 Spring Cloud 开发系统作为背景来解释。

1、TCC 实现阶段一：Try

首先，订单服务那儿，他的代码大致来说应该是这样子的：

```
public class OrderService {
    // 库存服务
    @Autowired
    private InventoryService inventoryService;

    // 积分服务
    @Autowried
    private CreditService creditService;

    // 仓储服务
    @Autowried
    private WmsService wmsService;

    // 对这个订单完成支付
    public void pay() {
        // 对本地的订单数据库修改订单状态为“已支付”
        orderDAO.updateStatus(OrderStatus.PAYED);

        // 调用库存服务扣减库存
        inventoryService.reduceStock();

        // 调用积分服务增加积分
        creditService.addCredit();

        // 调用仓储服务通知发货
        wmsService.saleDelivery();
    }
}
```

如果你之前看过 Spring Cloud 架构原理那篇文章，同时对 Spring Cloud 有一定的了解的话，应该是可以理解上面那段代码的。

其实就是订单服务完成本地数据库操作之后，通过 Spring Cloud 的 Feign 来调用其他的各个服务罢了。

但是光是凭借这段代码，是不足以实现 TCC 分布式事务的啊？！兄弟们，别着急，我们对这个订单服务修改点儿代码好不好。

首先，上面那个订单服务先把自己的状态修改为：OrderStatus.UPDATING。

这是啥意思呢？也就是说，在 pay() 那个方法里，你别直接把订单状态修改为已支付啊！你先把订单状态修改为 UPDATING，也就是修改中的意思。

这个状态是个没有任何含义的这么一个状态，代表有人正在修改这个状态罢了。

然后呢，库存服务直接提供的那个 reduceStock() 接口里，也别直接扣减库存啊，你可以是冻结掉库存。

举个例子，本来你的库存数量是 100，你别直接 100 - 2 = 98，扣减这个库存！

你可以把可销售的库存：100 - 2 = 98，设置为 98 没问题，然后在一个单独的冻结库存的字段里，设置一个 2。也就是说，有 2 个库存是给冻结了。

积分服务的 addCredit() 接口也是同理，别直接给用户增加会员积分。你可以先在积分表里的一个预增加积分字段加入积分。

比如：用户积分原本是 1190，现在要增加 10 个积分，别直接 1190 + 10 = 1200 个积分啊！

你可以保持积分为 1190 不变，在一个预增加字段里，比如说 prepare_add_credit 字段，设置一个 10，表示有 10 个积分准备增加。

仓储服务的 saleDelivery() 接口也是同理啊，你可以先创建一个销售出库单，但是这个销售出库单的状态是 “UNKNOWN”。

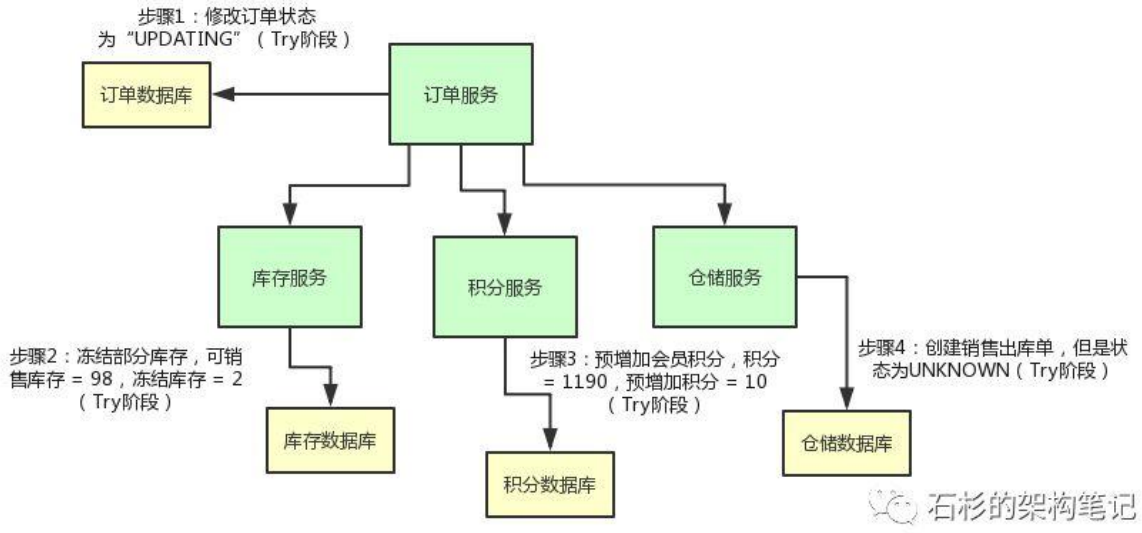
也就是说，刚刚创建这个销售出库单，此时还不确定他的状态是什么呢！

上面这套改造接口的过程，其实就是所谓的 TCC 分布式事务中的第一个 T 字母代表的阶段，也就是 Try 阶段。

总结上述过程，如果你要实现一个 TCC 分布式事务，首先你的业务的主流程以及各个接口提供的业务含义，不是说直接完成那个业务操作，而是完成一个 Try 的操作。

这个操作，一般都是锁定某个资源，设置一个预备类的状态，冻结部分数据，等等，大概都是这类操作。

咱们来一起看看下面这张图，结合上面的文字，再来捋一捋这整个过程。



2、TCC 实现阶段二：Confirm

然后就分成两种情况了，第一种情况是比较理想的，那就是各个服务执行自己的那个 Try 操作，都执行成功了，bingo！

这个时候，就需要依靠 TCC 分布式事务框架来推动后续的执行了。

这里简单提一句，如果你要玩儿 TCC 分布式事务，必须引入一款 TCC 分布式事务框架，比如国内开源的 ByteTCC、himly、tcc-transaction。

否则的话，感知各个阶段的执行情况以及推进执行下一个阶段的这些事情，不太可能自己手写实现，太复杂了。

如果你在各个服务里引入了一个 TCC 分布式事务的框架，订单服务里内嵌的那个 TCC 分布式事务框架可以感知到，各个服务的 Try 操作都成功了。

此时，TCC 分布式事务框架会控制进入 TCC 下一个阶段，第一个 C 阶段，也就是 Confirm 阶段。

为了实现这个阶段，你需要在各个服务里再加入一些代码。

比如说，订单服务里，你可以加入一个 Confirm 的逻辑，就是正式把订单的状态设置为 “已支付” 了，大概是类似下面这样子：

```
public class OrderServiceConfirm {
    public void pay() {
        orderDAO.updateStatus(OrderStatus.PAYED);
    }
}
```


库存服务也是类似的，你可以有一个 InventoryServiceConfirm 类，里面提供一个 reduceStock() 接口的 Confirm 逻辑，这里就是将之前冻结库存字段的 2 个库存扣掉变为 0。

这样的话，可销售库存之前就已经变为 98 了，现在冻结的 2 个库存也没了，那就正式完成了库存的扣减。

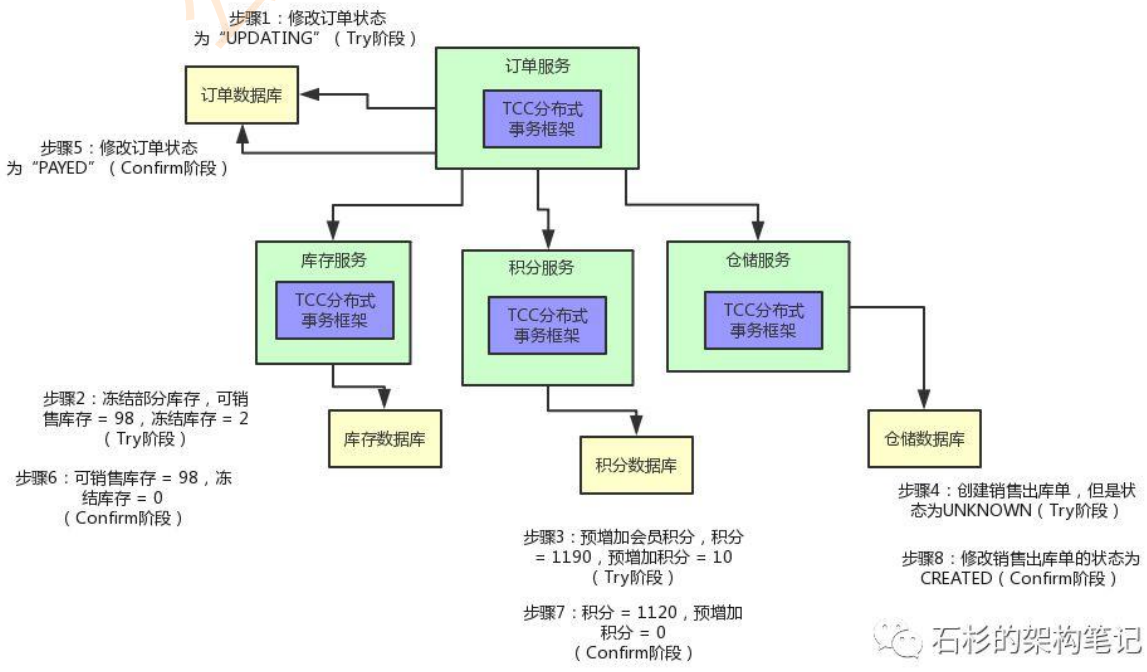
积分服务也是类似的，可以在积分服务里提供一个 CreditServiceConfirm 类，里面有一个 addCredit() 接口的 Confirm 逻辑，就是将预增加字段的 10 个积分扣掉，然后加入实际的会员积分字段中，从 1190 变为 1120。

仓储服务也是类似，可以在仓储服务中提供一个 WmsServiceConfirm 类，提供一个 saleDelivery()接口的 Confirm 逻辑，将销售出库单的状态正式修改为 “已创建”，可以供仓储管理人员查看和使用，而不是停留在之前的中间状态“UNKNOWN” 了。

好了，上面各种服务的 Confirm 的逻辑都实现好了，一旦订单服务里面的 TCC 分布式事务框架感知到各个服务的 Try 阶段都成功了以后，就会执行各个服务的 Confirm 逻辑。

订单服务内的 TCC 事务框架会负责跟其他各个服务内的 TCC 事务框架进行通信，依次调用各个服务的 Confirm 逻辑。然后，正式完成各个服务的所有业务逻辑的执行。

同样，给大家来一张图，顺着图一起来看看整个过程。



3、TCC 实现阶段三：Cancel

好，这是比较正常的一种情况，那如果是异常的一种情况呢?

举个例子：在 Try 阶段，比如积分服务吧，他执行出错了，此时会怎么样?

那订单服务内的 TCC 事务框架是可以感知到的，然后他会决定对整个 TCC 分布式事务进行回滚。

也就是说，会执行各个服务的第二个 C 阶段，Cancel 阶段。

同样，为了实现这个 Cancel 阶段，各个服务还得加一些代码。

首先订单服务，他得提供一个 OrderServiceCancel 的类，在里面有一个 pay() 接口的 Cancel 逻辑，就是可以将订单的状态设置为 “CANCELED”，也就是这个订单的状态是已取消。

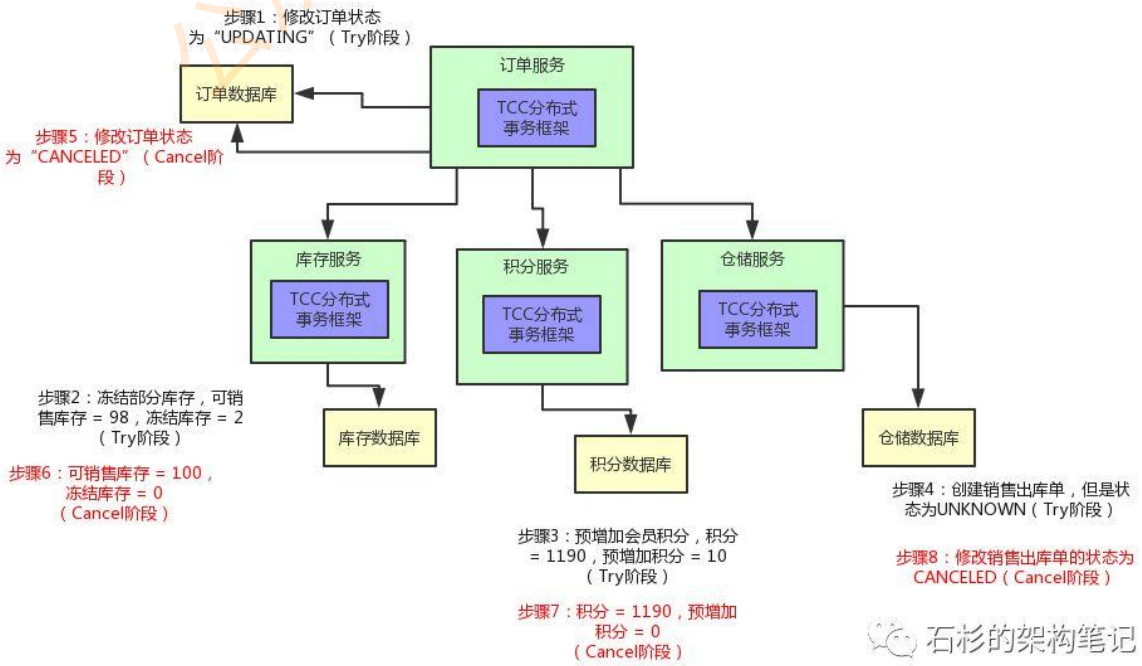
库存服务也是同理，可以提供 reduceStock() 的 Cancel 逻辑，就是将冻结库存扣减掉 2，加回到可销售库存里去，98 + 2 = 100。

积分服务也需要提供 addCredit() 接口的 Cancel 逻辑，将预增加积分字段的 10 个积分扣减掉。

仓储服务也需要提供一个 saleDelivery()接口的 Cancel 逻辑，将销售出库单的状态修改为 “CANCELED” 设置为已取消。

然后这个时候，订单服务的 TCC 分布式事务框架只要感知到了任何一个服务的 Try 逻辑失败了，就会跟各个服务内的 TCC 分布式事务框架进行通信，然后调用各个服务的 Cancel 逻辑。

大家看看下面的图，直观的感受一下。



五、总结与思考

好了，兄弟们，聊到这儿，基本上大家应该都知道 TCC 分布式事务具体是怎么回事了！

总结一下，你要玩儿 TCC 分布式事务的话：

首先需要选择某种 TCC 分布式事务框架，各个服务里就会有这个 TCC 分布式事务框架在运行。

然后你原本的一个接口，要改造为 3 个逻辑，Try-Confirm-Cancel。、

- 先是服务调用链路依次执行 Try 逻辑
- 如果都正常的话，TCC 分布式事务框架推进执行 Confirm 逻辑，完成整个事务
- 如果某个服务的 Try 逻辑有问题，TCC 分布式事务框架感知到之后就会推进执行各个服务的 Cancel 逻辑，撤销之前执行的各种操作

这就是所谓的 TCC 分布式事务。

TCC 分布式事务的核心思想，说白了，就是当遇到下面这些情况时，

- 某个服务的数据库宕机了
- 某个服务自己挂了
- 那个服务的 redis、elasticsearch、MQ 等基础设施故障了
- 某些资源不足了，比如说库存不够这些

先来 Try 一下，不要把业务逻辑完成，先试试看，看各个服务能不能基本正常运转，能不能先冻结我需要的资源。

如果 Try 都 ok，也就是说，底层的数据库、redis、elasticsearch、MQ 都是可以写入数据的，并且你保留好了需要使用的一些资源（比如冻结了一部分库存）。

接着，再执行各个服务的 Confirm 逻辑，基本上 Confirm 就可以很大概率保证一个分布式事务的完成了。

那如果 Try 阶段某个服务就失败了，比如说底层的数据库挂了，或者 redis 挂了，等等。

此时就自动执行各个服务的 Cancel 逻辑，把之前的 Try 逻辑都回滚，所有服务都不要执行任何设计的业务逻辑。保证大家要么一起成功，要么一起失败。

写到这里，本文差不多该结束了。等一等，你有没有想到一个问题?

如果有一些意外的情况发生了，比如说订单服务突然挂了，然后再次重启，TCC 分布式事务框架是如何保证之前没执行完的分布式事务继续执行的呢?

所以，TCC 事务框架都是要记录一些分布式事务的活动日志的，可以在磁盘上的日志文件里记录，也可以在数据库里记录。保存下来分布式事务运行的各个阶段和状态。

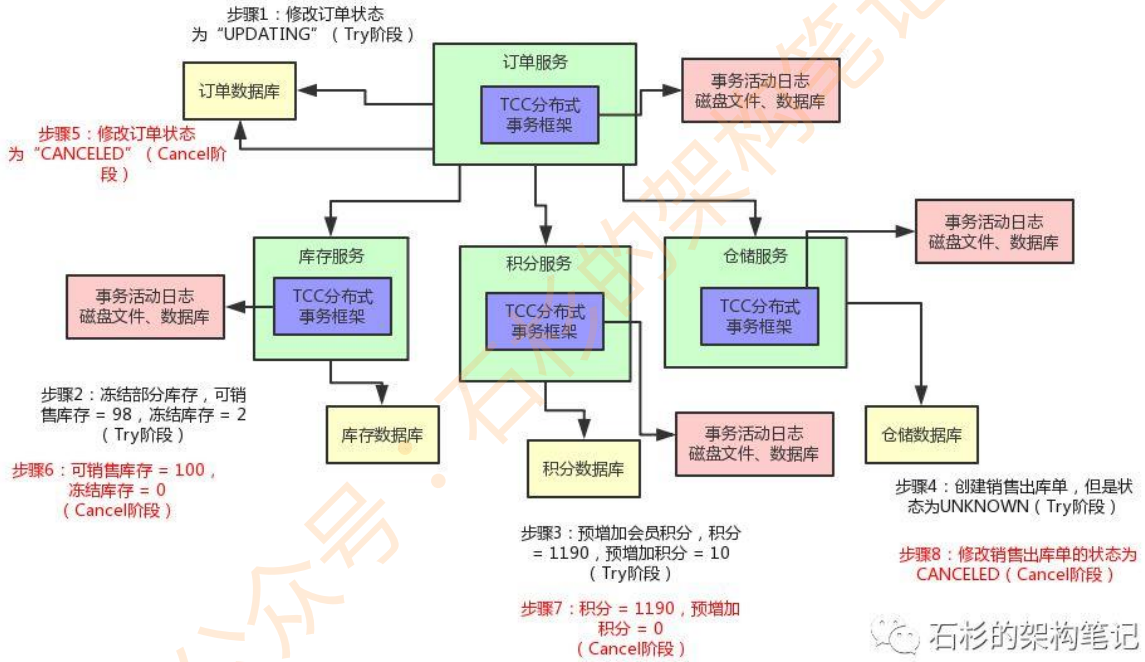
问题还没完，万一某个服务的 Cancel 或者 Confirm 逻辑执行一直失败怎么办呢?

那也很简单，TCC 事务框架会通过活动日志记录各个服务的状态。

举个例子，比如发现某个服务的 Cancel 或者 Confirm 一直没成功，会不停的重试调用他的 Cancel 或者 Confirm 逻辑，务必要他成功!

当然了，如果你的代码没有写什么 bug，有充足的测试，而且 Try 阶段都基本尝试了一下，那么其实一般 Confirm、Cancel 都是可以成功的!

最后，再给大家来一张图，来看看给我们的业务，加上分布式事务之后的整个执行流程：



不少大公司里，其实都是自己研发 TCC 分布式事务框架的，专门在公司内部使用，比如我们就是这样。

不过如果自己公司没有研发 TCC 分布式事务框架的话，那一般就会选用开源的框架。

这里笔者给大家推荐几个比较不错的框架，都是咱们国内自己开源出去的：ByteTCC，tcc-transaction，himly。

大家有兴趣的可以去他们的 github 地址，学习一下如何使用，以及如何跟 Spring Cloud、Dubbo 等服务框架整合使用。

只要把那些框架整合到你的系统里，很容易就可以实现上面那种奇妙的 TCC 分布式事务的效果了。

【坑爹呀！】最终一致性分布式事务如何保障实际生产中99.99%高可用?

> 作者:中华石杉 原文地址

目录

一、写在前面

上一篇文章咱们聊了聊 TCC 分布式事务，对于常见的微服务系统，大部分接口调用是同步的，也就是一个服务直接调用另外一个服务的接口。

这个时候，用 TCC 分布式事务方案来保证各个接口的调用，要么一起成功，要么一起回滚，是比较合适的。

但是在实际系统的开发过程中，可能服务间的调用是异步的。

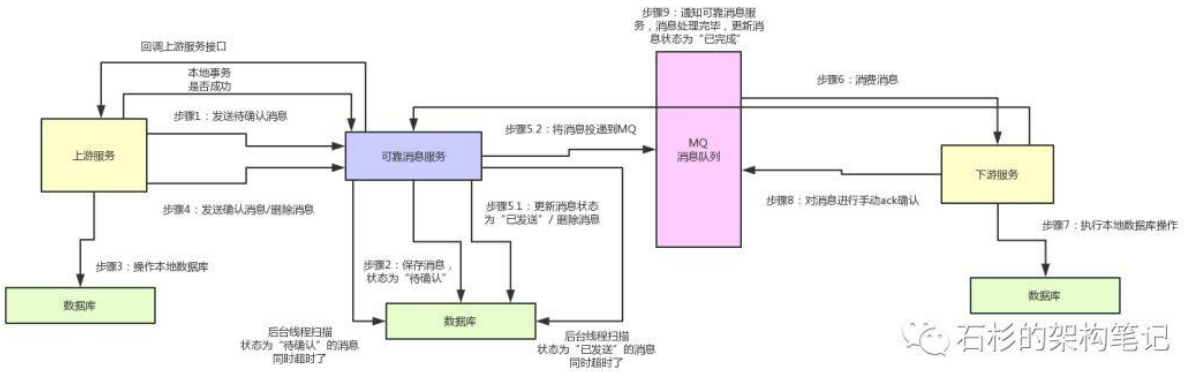
也就是说，一个服务发送一个消息给 MQ，即消息中间件，比如 RocketMQ、RabbitMQ、Kafka、ActiveMQ 等等。

然后，另外一个服务从 MQ 消费到一条消息后进行处理。这就成了基于 MQ 的异步调用了。

那么针对这种基于 MQ 的异步调用，如何保证各个服务间的分布式事务呢?

也就是说，我希望的是基于 MQ 实现异步调用的多个服务的业务逻辑，要么一起成功，要么一起失败。

这个时候，就要用上可靠 **消息最终一致性方案**，来实现分布式事务。



大家看看上面那个图，其实如果不考虑各种高并发、高可用等技术挑战的话，单从“可靠消息”以及“最终一致性”两个角度来考虑，这种分布式事务方案还是比较简单的。

二、可靠消息最终一致性方案的核心流程

（1）上游服务投递消息

如果要实现可靠消息最终一致性方案，一般你可以自己写一个可靠消息服务，实现一些业务逻辑。

首先，上游服务需要发送一条消息给可靠消息服务。

这条消息说白了，你可以认为是对下游服务一个接口的调用，里面包含了对应的一些请求参数。

然后，可靠消息服务就得把这条消息存储到自己的数据库里去，状态为“待确认”。

接着，上游服务就可以执行自己本地的数据库操作，根据自己的执行结果，再次调用可靠消息服务的接口。

如果本地数据库操作执行成功了，那么就找可靠消息服务确认那条消息。如果本地数据库操作失败了，那么就找可靠消息服务删除那条消息。

此时如果是确认消息，那么可靠消息服务就把数据库里的消息状态更新为“已发送”，同时将消息发送给 MQ。

这里有一个很关键的点，就是更新数据库里的消息状态和投递消息到 MQ。这俩操作，你得放在一个方法里，而且得开启本地事务。

啥意思呢?

- 如果数据库里更新消息的状态失败了，那么就抛异常退出了，就别投递到 MQ；
- 如果投递 MQ 失败报错了，那么就要抛异常让本地数据库事务回滚。
- 这俩操作必须得一起成功，或者一起失败。

如果上游服务是通知删除消息，那么可靠消息服务就得删除这条消息。

（2）下游服务接收消息

下游服务就一直等着从 MQ 消费消息好了，如果消费到了消息，那么就操作自己本地数据库。

如果操作成功了，就反过来通知可靠消息服务，说自己处理成功了，然后可靠消息服务就会把消息的状态设置为“已完成”。

（3）如何上游服务对消息的 100% 可靠投递?

上面的核心流程大家都看完：一个很大的问题就是，如果在上述投递消息的过程中各个环节出现了问题该怎么办?

我们如何保证消息 100% 的可靠投递，一定会从上游服务投递到下游服务？别着急，下面我们来逐一分析。

如果上游服务给可靠消息服务发送待确认消息的过程出错了，那没关系，上游服务可以感知到调用异常的，就不用执行下面的流程了，这是没问题的。

如果上游服务操作完本地数据库之后，通知可靠消息服务确认消息或者删除消息的时候，出现了问题。

比如：没通知成功，或者没执行成功，或者是可靠消息服务没成功的投递消息到 MQ。这一系列步骤出了问题怎么办?

其实也没关系，因为在这些情况下，那条消息在可靠消息服务的数据库里的状态会一直是 “待确认”。

此时，我们在可靠消息服务里开发一个后台定时运行的线程，不停的检查各个消息的状态。

如果一直是 “待确认” 状态，就认为这个消息出了点什么问题。

此时的话，就可以回调上游服务提供的一个接口，问问说，兄弟，这个消息对应的数据库操作，你执行成功了没啊?

如果上游服务答复说，我执行成功了，那么可靠消息服务将消息状态修改为 “已发送”，同时投递消息到 MQ。

如果上游服务答复说，没执行成功，那么可靠消息服务将数据库中的消息删除即可。

通过这套机制，就可以保证，可靠消息服务一定会尝试完成消息到 MQ 的投递。

（4）如何保证下游服务对消息的 100% 可靠接收?

那如果下游服务消费消息出了问题，没消费到？或者是下游服务对消息的处理失败了，怎么办?

其实也没关系，在**可靠消息服务里开发一个后台线程，不断的检查消息状态**。

如果消息状态一直是 “已发送”，始终没有变成 “已完成”，那么就说明下游服务始终没有处理成功。

此时可靠消息服务就可以再次尝试重新投递消息到 MQ，让下游服务来再次处理。

只要下游服务的接口逻辑 **实现幂等性**，保证多次处理一个消息，不会插入重复数据即可。

（5）如何基于 RocketMQ 来实现可靠消息最终一致性方案？

在上面的通用方案设计里，完全依赖可靠消息服务的各种自检机制来确保：

- 如果上游服务的数据库操作没成功，下游服务是不会收到任何通知
- 如果上游服务的数据库操作成功了，可靠消息服务死活都会确保将一个调用消息投递给下游服务，而且一定会确保下游服务务必成功处理这条消息。

通过这套机制，保证了基于 MQ 的异步调用 / 通知的服务间的分布式事务保障。

其实阿里开源的 RocketMQ，就实现了可靠消息服务的所有功能，核心思想跟上面类似。

只不过 RocketMQ 为了保证高并发、高可用、高性能，做了较为复杂的架构实现，非常的优秀。

有兴趣的同学，自己可以去查阅 RocketMQ 对分布式事务的支持。

三、可靠消息最终一致性方案的高可用保障生产实践

（1）背景引入

其实上面那套方案和思想，很多同学应该都知道是怎么回事儿，我们也主要就是铺垫一下这套理论思想。

在实际落地生产的时候，如果没有高并发场景的，完全可以参照上面的思路自己基于某个 MQ 中间件开发一个可靠消息服务。

如果有高并发场景的，可以用 RocketMQ 的分布式事务支持，上面的那套流程都可以实现。

今天给大家分享的一个核心主题，就是**这套方案如何保证 99.99% 的高可用**。

其实大家应该发现了这套方案里保障高可用性最大的一个依赖点，就是 **MQ 的高可用性**。

任何一种 MQ 中间件都有一整套的高可用保障机制，无论是 RabbitMQ、RocketMQ 还是 Kafka。

所以在大公司里使用可靠消息最终一致性方案的时候，我们通常对可用性的保障都是依赖于公司基础架构团队对 MQ 的高可用保障。

也就是说，大家应该相信兄弟团队，99.99% 可以保障 MQ 的高可用，绝对不会因为 MQ 集群整体宕机，而导致公司业务系统的分布式事务全部无法运行。

但是现实是很残酷的，很多中小型的公司，甚至是一些中大型公司，或多或少都遇到过 MQ 集群整体故障的场景。

MQ 一旦完全不可用，就会导致业务系统的各个服务之间无法通过 MQ 来投递消息，导致业务流程中断。

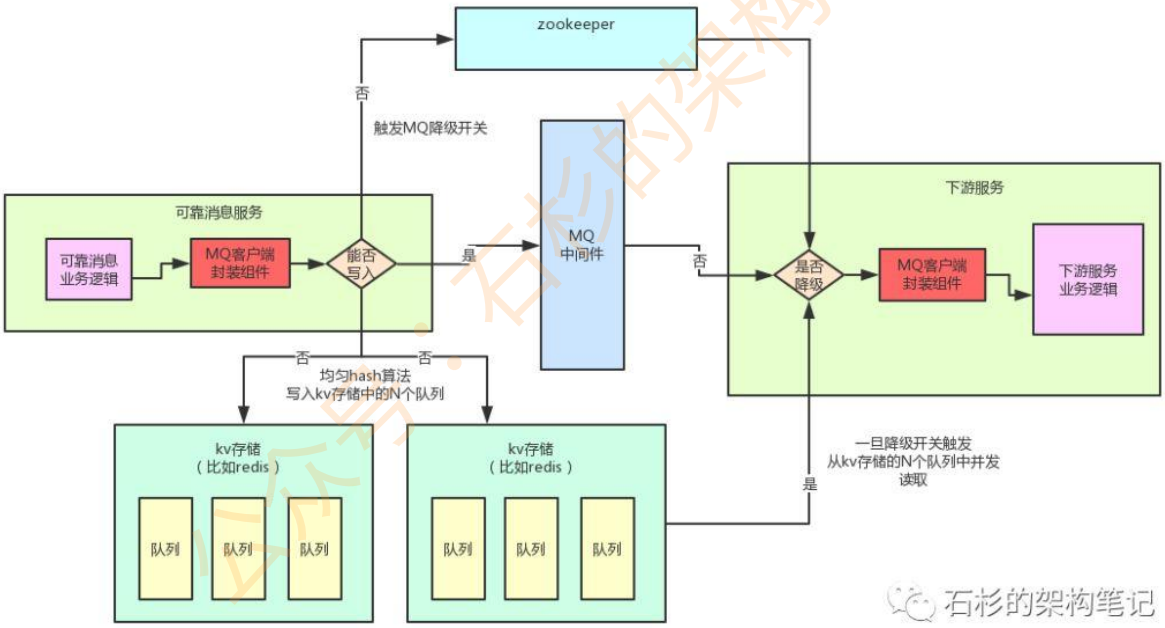
比如最近就有一个朋友的公司，也是做电商业务的，就遇到了 MQ 中间件在自己公司机器上部署的集群整体故障不可用，导致依赖 MQ 的分布式事务全部无法跑通，业务流程大量中断的情况。

这种情况，就需要针对这套分布式事务方案实现一套高可用保障机制。

(2）基于 KV 存储的队列支持的高可用降级方案

大家来看看下面这张图，这是我曾经指导过朋友的一个公司针对可靠消息最终一致性方案设计的一套高可用保障降级机制。

这套机制不算太复杂，可以非常简单有效的保证那位朋友公司的高可用保障场景，一旦 MQ 中间件出现故障，立马自动降级为备用方案。



（1）自行封装 MQ 客户端组件与故障感知

首先第一点，你要做到自动感知 MQ 的故障接着自动完成降级，那么必须动手对 MQ 客户端进行封装，发布到公司 Nexus 私服上去。

然后公司需要支持 MQ 降级的业务服务都使用这个自己封装的组件来发送消息到 MQ，以及从 MQ 消费消息。

在你自己封装的 MQ 客户端组件里，你可以根据写入 MQ 的情况来判断 MQ 是否故障。

比如说，如果连续 10 次重试尝试投递消息到 MQ 都发现异常报错，网络无法联通等问题，说明 MQ 故障，此时就可以自动感知以及自动触发降级开关。

(2）基于 kv 存储中队列的降级方案

如果 MQ 挂掉之后，要是希望继续投递消息，那么就必须得找一个 MQ 的替代品。

举个例子，比如我那位朋友的公司是没有高并发场景的，消息的量很少，只不过可用性要求高。此时就可以类似 redis 的 kv 存储中的队列来进行替代。

由于 redis 本身就支持队列的功能，还有类似队列的各种数据结构，所以你可以将消息写入 kv 存储格式的队列数据结构中去。

ps：关于 redis 的数据存储格式、支持的数据结构等基础知识，请大家自行查阅了，网上一大堆

但是，这里有几个大坑，一定要注意一下。

> **第一个，任何 kv 存储的集合类数据结构，建议不要往里面写入数据量过大，否则会导致大 value 的情况发生，引发严重的后果。 因此绝不能在 redis 里搞一个 key，就拼命往这个数据结构中一直写入消息，这是肯定不行的。**

> **第二个，绝对不能往少数 key 对应的数据结构中持续写入数据，那样会导致热 key 的产生，也就是某几个 key 特别热。大家要知道，一般 kv 集群，都是根据 key 来 hash 分配到各个机器上的，你要是老写少数几个 key，会导致 kv 集群中的某台机器访问过高，负载过大。**

基于以上考虑，下面是笔者当时设计的方案：

- 根据他们每天的消息量，在 kv 存储中固定划分上百个队列，有上百个 key 对应。
- 这样保证每个 key 对应的数据结构中不会写入过多的消息，而且不会频繁的写少数几个 key。
- 一旦发生了 MQ 故障，可靠消息服务可以对每个消息通过 hash 算法，均匀的写入固定好的上百个 key 对应的 kv 存储的队列中。

同时此时需要通过 zk 触发一个降级开关，整个系统在 MQ 这块的读和写全部立马降级。

（3）下游服务消费 MQ 的降级感知

下游服务消费 MQ 也是通过自行封装的组件来做的，此时那个组件如果从 zk 感知到降级开关打开了，首先会判断自己是否还能继续从 MQ 消费到数据?

如果不能了，就开启多个线程，并发的从 kv 存储的各个预设好的上百个队列中不断的获取数据。

每次获取到一条数据，就交给下游服务的业务逻辑来执行。

通过这套机制，就实现了 MQ 故障时候的自动故障感知，以及自动降级。如果系统的负载和并发不是很高的话，用这套方案大致是没没问题的。

因为在生产落地的过程中，包括大量的容灾演练以及生产实际故障发生时的表现来看，都是可以有效的保证 MQ 故障时，业务流程继续自动运行的。

（4）故障的自动恢复

如果降级开关打开之后，自行封装的组件需要开启一个线程，每隔一段时间尝试给 MQ 投递一个消息看看是否恢复了。

如果 MQ 已经恢复可以正常投递消息了，此时就可以通过 zk 关闭降级开关，然后可靠消息服务继续投递消息到 MQ，下游服务在确认 kv 存储的各个队列中已经没有数据之后，就可以重新切换为从 MQ 消费消息。

（5）更多的业务细节

其实上面说的那套方案主要是一套通用的降级方案，但是具体的落地是要结合各个公司不同的业务细节来决定的，很多细节多没法在文章里体现。

比如说你们要不要保证消息的顺序性？是不是涉及到需要根据业务动态，生成大量的 key？等等。

此外，这套方案实现起来还是有一定的成本的，所以建议大家尽可能还是 push 公司的基础架构团队，保证 MQ 的 99.99% 可用性，不要宕机。

其次就是根据大家公司的实际对高可用需求来决定，如果感觉 MQ 偶尔宕机也没事，可以容忍的话，那么也不用实现这种降级方案。

但是如果公司领导认为 MQ 中间件宕机后，一定要保证业务系统流程继续运行，那么还是要考虑一些高可用的降级方案，比如本文提到的这种。

最后再说一句，真要是一些公司涉及到每秒几万几十万的高并发请求，那么对 MQ 的降级方案会设计的更加的复杂，那就远远不是这么简单可以做到的。

拜托，面试请不要再问我 Redis 分布式锁的实现原理

> **作者:中华石杉** 原文地址

目录

一、写在前面

现在面试，一般都会聊聊分布式系统这块的东西。通常面试官都会从服务框架（Spring Cloud、Dubbo）聊起，一路聊到分布式事务、分布式锁、ZooKeeper 等知识。

所以咱们这篇文章就来聊聊分布式锁这块知识，具体的来看看 Redis 分布式锁的实现原理。

说实话，如果在公司里落地生产环境用分布式锁的时候，一定是会用开源类库的，比如 Redis 分布式锁，一般就是用 Redisson 框架就好了，非常的简便易用。

大家如果有兴趣，可以去看看 Redisson 的官网，看看如何在项目中引入 Redisson 的依赖，然后基于 Redis 实现分布式锁的加锁与释放锁。

下面给大家看一段简单的使用代码片段，先直观的感受一下：

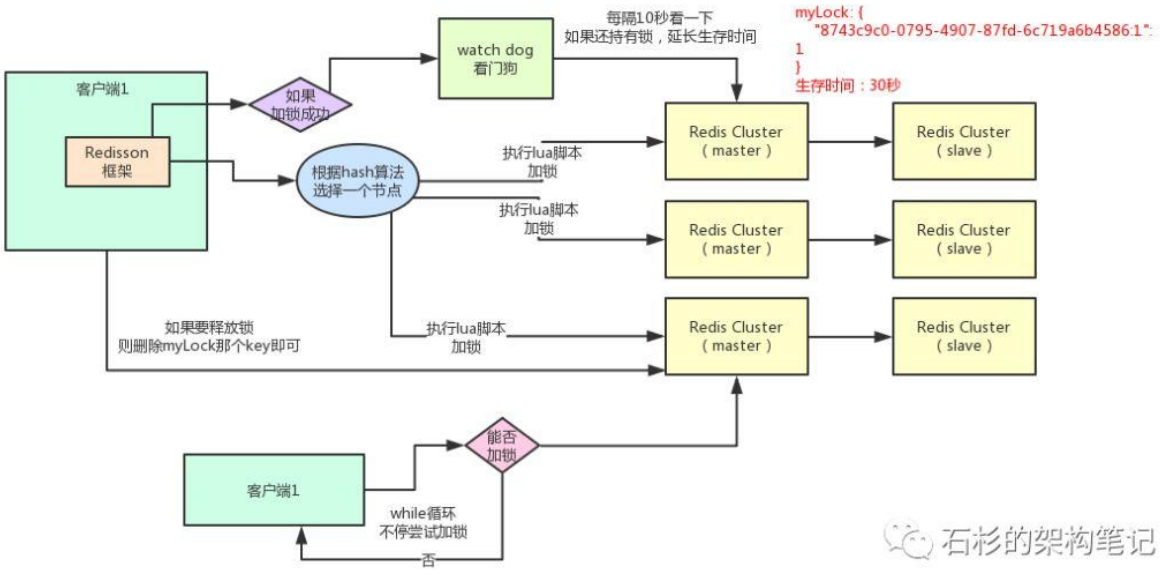
```
RLock lock = redisson.getLock("myLock");
lock.lock();
lock.unlock();
```


怎么样，上面那段代码，是不是感觉简单的不行！

此外，人家还支持 redis 单实例、redis 哨兵、redis cluster、redis master-slave 等各种部署架构，都可以给你完美实现。

二、Redisson 实现 Redis 分布式锁的底层原理

好的，接下来就通过一张手绘图，给大家说说 Redisson 这个开源框架对 Redis 分布式锁的实现原理。



（1）加锁机制

咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个 redis cluster 集群，他首先会根据 hash 节点选择一台机器。

这里注意，仅仅只是选择一台机器！这点很关键！

紧接着，就会发送一段 lua 脚本到 redis 上，那段 lua 脚本如下所示：

```
"if (redis.call('exists', KEYS[1]) == 0) then " +
    "redis.call('hset', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"return redis.call('pttl', KEYS[1]);"
```


为啥要用 lua 脚本呢？

因为一大坨复杂的业务逻辑，可以通过封装在 lua 脚本中发送给 redis，保证这段复杂业务逻辑执行的原子性。

那么，这段 lua 脚本是什么意思呢？

KEYS[1] 代表的是你加锁的那个 key，比如说：

```
RLock lock = redisson.getLock("myLock");
```

这里你自己设置了加锁的那个锁 key 就是 “myLock”。

ARGV[1] 代表的就是锁 key 的默认生存时间，默认 30 秒。

ARGV[2] 代表的是加锁的客户端的 ID，类似于下面这样：

```
8743c9c0-0795-4907-87fd-6c719a6b4586:1
```

给大家解释一下，第一段 if 判断语句，就是用 “exists myLock” 命令判断一下，如果你要加锁的那个锁 key 不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的命令：

```
hset myLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
```

通过这个命令设置一个 hash 数据结构，这行命令执行后，会出现一个类似下面的数据结构：

```
myLock:
{
    “8743c9c0-0795-4907-87fd-6c719a6b4586:1”: 1
}
```


上述就代表 “8743c9c0-0795-4907-87fd-6c719a6b4586:1” 这个客户端对 “myLock” 这个锁 key 完成了加锁。

接着会执行 “pexpire myLock 30000” 命令，设置 myLock 这个锁 key 的生存时间是 30 秒。

好了，到此为止，ok，加锁完成了。

（2）锁互斥机制

那么在这个时候，如果客户端 2 来尝试加锁，执行了同样的一段 lua 脚本，会咋样呢?

很简单，第一个 if 判断会执行 “exists myLock”，发现 myLock 这个锁 key 已经存在了。

接着第二个 if 判断，判断一下，myLock 锁 key 的 hash 数据结构中，是否包含客户端 2 的 ID，但是明显不是的，因为那里包含的是客户端 1 的 ID。

所以，客户端 2 会获取到 pttl myLock 返回的一个数字，这个数字代表了 myLock 这个锁 key 的剩余生存时间。比如还剩 15000 毫秒的生存时间。

此时客户端 2 会进入一个 while 循环，不停的尝试加锁。

（3）watch dog 自动延期机制

客户端 1 加锁的锁 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢?

简单！只要客户端 1 一旦加锁成功，就会启动一个 watch dog 看门狗，他是一个后台线程，会每隔 10 秒检查一下，如果客户端 1 还持有锁 key，那么就会不断的延长锁 key 的生存时间。

（4）可重入加锁机制

那如果客户端 1 都已经持有了这把锁了，结果可重入的加锁会怎么样呢?

比如下面这种代码：

```
RLock lock = redisson.getLock("myLock");
lock.lock();

// 一大坨代码

lock.lock();
// 一大坨代码
lock.unlock();

lock.unlock();
```


这时我们来分析一下上面那段 lua 脚本。

第一个 if 判断肯定不成立，“exists myLock” 会显示锁 key 已经存在了。

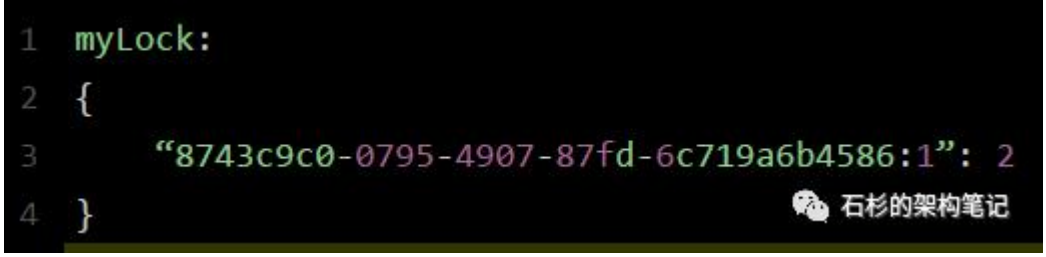
第二个 if 判断会成立，因为 myLock 的 hash 数据结构中包含的那个 ID，就是客户端 1 的那个 ID，也就是 “8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

```
incrby myLock  8743c9c0-0795-4907-87fd-6c71a6b4586:1 1
```

通过这个命令，对客户端 1 的加锁次数，累加 1。

此时 myLock 数据结构变为下面这样：



大家看到了吧，那个 myLock 的 hash 数据结构中的那个客户端 ID，就对应着加锁的次数

（5）释放锁机制

如果执行 `lock.unlock()` ，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对 myLock 数据结构中的那个加锁次数减 1。

如果发现加锁次数是 0 了，说明这个客户端已经不再持有锁了，此时就会用：

`“del myLock”` 命令，从 redis 里删除这个 key。

然后呢，另外的客户端 2 就可以尝试完成加锁了。

这就是所谓的分布式锁的开源 Redisson 框架的实现机制。

一般我们在生产系统中，可以用 Redisson 框架提供的这个类库来基于 redis 进行分布式锁的加锁与释放锁。

（6）上述 Redis 分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个 redis master 实例，写入了 myLock 这种锁 key 的 value，此时会异步复制给对应的 master slave 实例。

但是这个过程中一旦发生 redis master 宕机，主备切换，redis slave 变为了 redis master。

接着就会导致，客户端 2 来尝试加锁的时候，在新的 redis master 上完成了加锁，而客户端 1 也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是 redis cluster，或者是 redis master-slave 架构的主从异步复制导致的 redis 分布式锁的最大缺陷：在 redis master 实例宕机的时候，可能导致多个客户端同时完成加锁。

三、未完待续

下一篇文章，给大家分享一下电商系统中，大促销的活动场景下，每秒上千订单的时候如何对 Redis 分布式锁进行高并发的优化。

敬请关注：

《每秒上千订单场景下的分布式锁高并发优化实践！》

每秒上千订单场景下的分布式锁高并发优化实践！

> **作者:中华石杉** 原文地址

“上一篇文章我们聊了聊 Redisson 这个开源框架对 Redis 分布式锁的实现原理，如果有不了解的兄弟可以看一下：**《拜托，面试请不要再问我 Redis 分布式锁实现原理》**。

今天就给大家聊一个有意思的话题：每秒上千订单场景下，如何对分布式锁的并发能力进行优化？

背景引入

首先，我们一起来看看这个问题的背景？

前段时间有个朋友在外面面试，然后有一天找我聊说：有一个国内不错的电商公司，面试官给他出了一个场景题：

假如下单时，用分布式锁来防止库存超卖，但是是每秒上千订单的高并发场景，如何对分布式锁进行高并发优化来应对这个场景？

他说他当时没答上来，因为没做过没什么思路。其实我当时听到这个面试题心里也觉得有点意思，因为如果是我来面试候选人的话，应该会给的范围更大一些。

比如，让面试的同学聊一聊电商高并发秒杀场景下的库存超卖解决方案，各种方案的优缺点以及实践，进而聊到分布式锁这个话题。

因为库存超卖问题是有很多种技术解决方案的，比如悲观锁，分布式锁，乐观锁，队列串行化，Redis 原子操作，等等吧。

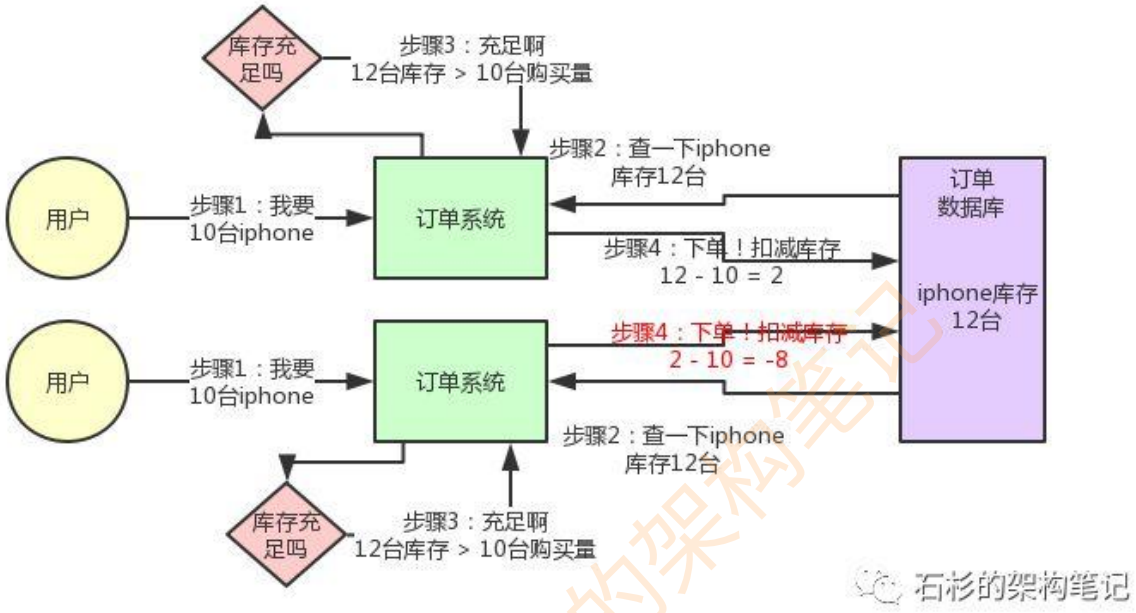
但是既然那个面试官兄弟限定死了用分布式锁来解决库存超卖，我估计就是想问一个点：在高并发场景下如何优化分布式锁的并发性能。

我觉得，面试官提问的角度还是可以接受的，因为在实际落地生产的时候，分布式锁这个东西保证了数据的准确性，但是他天然并发能力有点弱。

刚好我之前在自己项目的其他场景下，确实是做过高并发场景下的分布式锁优化方案，因此正好是借着这个朋友的面试题，把分布式锁的高并发优化思路，给大家来聊一聊。

库存超卖现象是怎么产生的?

先来看看如果不用分布式锁，所谓的电商库存超卖是啥意思？大家看看下面的图：



这个图，其实很清晰了，假设订单系统部署两台机器上，不同的用户都要同时买 10 台 iphone，分别发了一个请求给订单系统。

接着每个订单系统实例都去数据库里查了一下，当前 iphone 库存是 12 台。

俩大兄弟一看，乐了，12 台库存大于了要买的 10 台数量啊！

于是乎，每个订单系统实例都发送 SQL 到数据库里下单，然后扣减了 10 个库存，其中一个将库存从 12 台扣减为 2 台，另外一个将库存从 2 台扣减为 - 8 台。

现在完了，库存出现了负数！泪奔啊，没有 20 台 iphone 发给两个用户啊！这可如何是好。

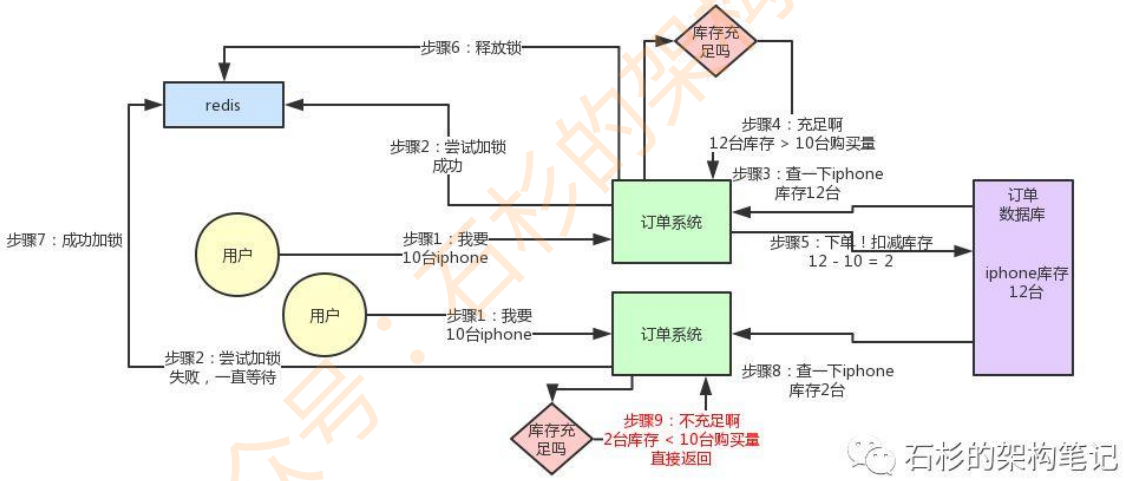
用分布式锁如何解决库存超卖问题?

我们用分布式锁如何解决库存超卖问题呢？其实很简单，回忆一下上次我们说的那个分布式锁的实现原理：

同一个锁 key，同一时间只能有一个客户端拿到锁，其他客户端会陷入无限的等待来尝试获取那个锁，只有获取到锁的客户端才能执行下面的业务逻辑。

```
// 大家都获取iphone库存的那个锁key
RLock lock = redisson.getLock("iphone_stock");

// 只有一个客户端可以成功加锁，继续往下运行
lock.lock();

// 执行数据库业务逻辑，先查出来iphone的库存，跟购买数量比较一下，如果库存充足就下单扣减库存
Long goodsStock = stockDAO.getByGoodsId(iphoneGoodsId);

if(goodsStock > purhcaseCount) {
    orderDAO.create(order);
    stockDAO.reduceStock(iphoneGoodsId, purchaseCount);
}

// 释放锁，其他客户端可以尝试加锁
lock.unlock();
```


代码大概就是上面那个样子，现在我们来分析一下，为啥这样做可以避免库存超卖？



大家可以顺着上面的那个步骤序号看一遍，马上就明白了。

从上图可以看到，只有一个订单系统实例可以成功加分布式锁，然后只有他一个实例可以查库存、判断库存是否充足、下单扣减库存，接着释放锁。

释放锁之后，另外一个订单系统实例才能加锁，接着查库存，一下发现库存只有 2 台了，库存不足，无法购买，下单失败。不会将库存扣减为 - 8 的。

有没有其他方案可以解决库存超卖问题？

当然有啊！比如悲观锁，分布式锁，乐观锁，队列串行化，异步队列分散，Redis 原子操作，等等，很多方案，我们对库存超卖有自己的一整套优化机制。

但是前面说过了，这篇文章就聊一个分布式锁的并发优化，不是聊库存超卖的解决方案，所以**库存超卖只是一个业务场景而已。**

以后有机会笔者会写一篇文章，讲讲电商库存超卖问题的解决方案，这篇文章先 focus 在一个分布式锁并发优化上，希望大家明白这个用意和背景，避免有的兄弟没看清楚又吐槽。

而且建议大家即使对文章里的内容有异议，公众号后台给我留言跟我讨论一下，技术，就是要多交流，打开思路，碰撞思维。

分布式锁的方案在高并发场景下

好，现在我们来看看，分布式锁的方案在高并发场景下有什么问题?

问题很大啊！兄弟，不知道你看出来了没有。分布式锁一旦加了之后，对同一个商品的下单请求，会导致所有客户端都必须对同一个商品的库存锁 key 进行加锁。

比如，对 iphone 这个商品的下单，都必对 “iphone_stock” 这个锁 key 来加锁。这样会导致对同一个商品的下单请求，就必须串行化，一个接一个的处理。

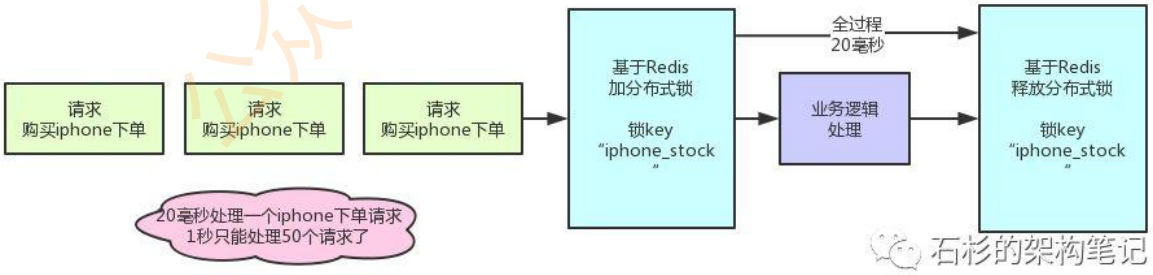
大家再回去对照上面的图反复看一下，应该能想明白这个问题。

假设加锁之后，释放锁之前，查库存 -> 创建订单 -> 扣减库存，这个过程性能很高吧，算他全过程 20 毫秒，这应该不错了。

那么 1 秒是 1000 毫秒，只能容纳 50 个对这个商品的请求依次串行完成处理。

比如一秒钟来 50 个请求，都是对 iphone 下单的，那么每个请求处理 20 毫秒，一个一个来，最后 1000 毫秒正好处理完 50 个请求。

大家看一眼下面的图，加深一下感觉。



所以看到这里，大家起码也明白了，简单的使用分布式锁来处理库存超卖问题，存在什么缺陷。

缺陷就是同一个商品多用户同时下单的时候，会基于分布式锁串行化处理，导致没法同时处理同一个商品的大量下单的请求。

这种方案，要是应对那种低并发、无秒杀场景的普通小电商系统，可能还可以接受。

因为如果并发量很低，每秒就不到 10 个请求，没有瞬时高并发秒杀单个商品的场景的话，其实也很少会对同一个商品在一秒内瞬间下 1000 个订单，因为小电商系统没那场景。

如何对分布式锁进行高并发优化?

好了，终于引入正题了，那么现在怎么办呢?

面试官说，我现在就卡死，库存超卖就是用分布式锁来解决，而且一秒对一个 iphone 下上千订单，怎么优化?

现在按照刚才的计算，你一秒钟只能处理针对 iphone 的 50 个订单。

其实说出来也很简单，相信很多人看过 java 里的 ConcurrentHashMap 的源码和底层原理，应该知道里面的核心思路，就是分段加锁!

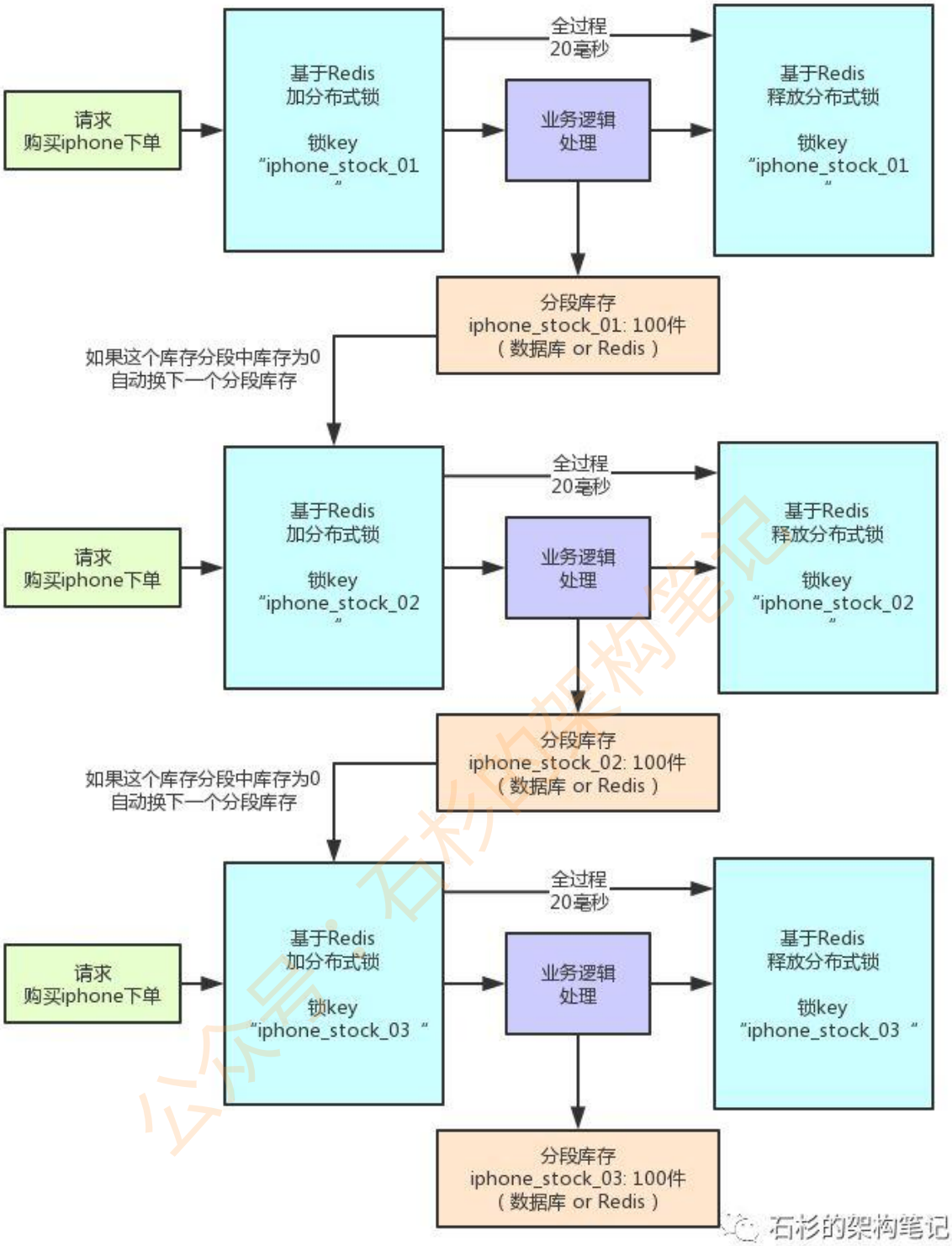
把数据分成很多个段，每个段是一个单独的锁，所以多个线程过来并发修改数据的时候，可以并发的修改不同段的数据。不至于说，同一时间只能有一个线程独占修改 ConcurrentHashMap 中的数据。

另外，Java 8 中新增了一个 LongAdder 类，也是针对 Java 7 以前的 AtomicLong 进行的优化，解决的是 CAS 类操作在高并发场景下，使用乐观锁思路，会导致大量线程长时间重复循环。

LongAdder 中也是采用了类似的分段 CAS 操作，失败则自动迁移到下一个分段进行 CAS 的思路。

其实分布式锁的优化思路也是类似的，之前我们是在另外一个业务场景下落地了这个方案到生产中，不是在库存超卖问题里用的。

但是库存超卖这个业务场景不错，很容易理解，所以我们就用这个场景来说一下。大家看看下面的图:



其实这就是分段加锁。你想，假如你现在 iphone 有 1000 个库存，那么你完全可以给拆成 20 个库存段，要是你愿意，可以在数据库的表里建 20 个库存字段，比如 stock_01，stock_02，类似这样的，也可以在 redis 之类的地方放 20 个库存 key。

总之，就是把你的 1000 件库存给他拆开，每个库存段是 50 件库存，比如 stock_01 对应 50 件库存，stock_02 对应 50 件库存。

接着，每秒 1000 个请求过来了，好！此时其实可以是自己写一个简单的随机算法，每个请求都是随机在 20 个分段库存里，选择一个进行加锁。

bingo！这样就好了，同时可以有最多 20 个下单请求一起执行，每个下单请求锁了一个库存分段，然后在业务逻辑里面，就对数据库或者是 Redis 中的那个分段库存进行操作即可，包括查库存 -> 判断库存是否充足 -> 扣减库存。

这相当于什么呢？相当于一个 20 毫秒，可以并发处理掉 20 个下单请求，那么 1 秒，也就可以依次处理掉 20 * 50 = 1000 个对 iphone 的下单请求了。

一旦对某个数据做了分段处理之后，**有一个坑大家一定要注意**：就是如果某个下单请求，咔嚓加锁，然后发现这个分段库存里的库存不足了，此时咋办？

这时你得自动释放锁，然后立马换下一个分段库存，再次尝试加锁后尝试处理。这个过程一定要实现。

分布式锁并发优化方案有没有什么不足？

不足肯定是有的，最大的不足，大家发现没有，很不方便啊！实现太复杂了。

- 首先，你得对一个数据分段存储，一个库存字段本来好好的，现在要分为 20 个分段库存字段；
- 其次，你在每次处理库存的时候，还得自己写随机算法，随机挑选一个分段来处理；
- 最后，如果某个分段中的数据不足了，你还得自动切换到下一个分段数据去处理。

这个过程都是要手动写代码实现的，还是有点工作量，挺麻烦的。

不过我们确实在一些业务场景里，因为用到了分布式锁，然后又必须要进行锁并发的优化，又进一步用到了分段加锁的技术方案，效果当然是很好的了，一下子并发性能可以增长几十倍。

该优化方案的后续改进

以我们本文所说的库存超卖场景为例，你要是这么玩，会把自己搞的很痛苦！

再次强调，**我们这里的库存超卖场景，仅仅只是作为演示场景而已，以后有机会，再单独聊聊高并发秒杀系统架构下的库存超卖的其他解决方案**。

上篇文章的补充说明

本文最后做个说明，笔者收到一些朋友留言，说有朋友在技术群里看到上篇文章之后，吐槽了一通上一篇文章**（《拜托，面试请不要再问我 Redis 分布式锁的实现原理》）**，说是那个 Redis 分布式锁的实现原理把人给带歪了。

在这儿得郑重说明一下，上篇文章，明确说明了是 Redisson 那个开源框架对 Redis 锁的实现原理，并不是我个人 YY 出来的那一套原理。

实际上 Redisson 作为一款优秀的开源框架，我觉得他整体对分布式锁的实现是 OK 的，虽然有一些缺陷，但是生产环境可用。

另外，有的兄弟可能觉得那个跟 Redis 官网作者给出的分布式锁实现思路不同，所以就吐槽，说要遵循 Redis 官网中的作者的分布式锁实现思路。

其实我必须指出，Redis 官网中给出的仅仅是 Redis 分布式锁的实现思路而已，记住，那是思路！思路跟落地生产环境的技术方案之间是有差距的。

比如说 Redis 官网给出的分布式锁实现思路，并没有给出到分布式锁的自动续期机制、锁的互斥自等待机制、锁的可重入加锁与释放锁的机制。但是 Redisson 框架对分布式锁的实现是实现了一整套机制的。

所以重复一遍，那仅仅是思路，如果你愿意，你完全可以基于 Redis 官网的思路自己实现一套生产级的分布式锁出来。

另外 Redis 官网给出的 RedLock 算法，一直是我个人并不推崇在生产使用的。

因为那套算法中可能存在一些逻辑问题，在国外是引发了争议的，连 Redis 作者自己都在官网中给出了因为他的 RedLock 算法而引发争议的文章，当然他自己是不太同意的。

但是这个事儿，就搞成公说公有理，婆说婆有理了。具体请参加官网原文：

`Martin Kleppmann analyzed Redlock here. I disagree with the analysis and posted my reply to his analysis here。`

因此下回有机会，我会通过大量手绘图的形式，给大家写一下 Redis 官方作者自己提出的 RedLock 分布式锁的算法，以及该算法基于 Redisson 框架如何落地生产环境使用，到时大家可以再讨论。

【七张图】彻底讲清楚ZooKeeper分布式锁的实现原理

> **作者:中华石杉** **原文地址**

一、写在前面

之前写过一篇文章（**《拜托，面试请不要再问我 Redis 分布式锁的实现原理》**），给大家说了一下 Redisson 这个开源框架是如何实现 Redis 分布式锁原理的，这篇文章再给大家聊一下 ZooKeeper 实现分布式锁的原理。

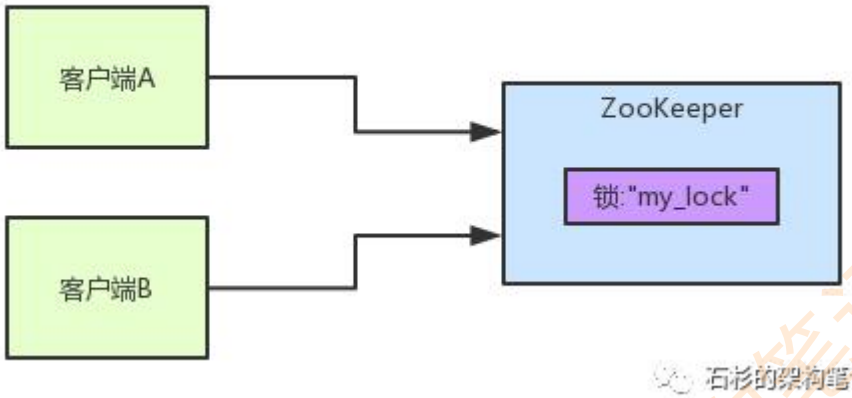
同理，我是直接基于比较常用的 Curator 这个开源框架，聊一下这个框架对 ZooKeeper（以下简称 zk）分布式锁的实现。

一般除了大公司是自行封装分布式锁框架之外，建议大家用这些开源框架封装好的分布式锁实现，这是一个比较快捷省事儿的方式。

二、ZooKeeper 分布式锁机制

接下来我们一起来看看，多客户端获取及释放 zk 分布式锁的整个流程及背后的原理。

首先大家看看下面的图，如果现在有两个客户端一起要争抢 zk 上的一把分布式锁，会是个什么场景?



如果大家对 zk 还不太了解，建议先百度一下，快速了解一些基本概念，比如 zk 有哪些节点类型等等。

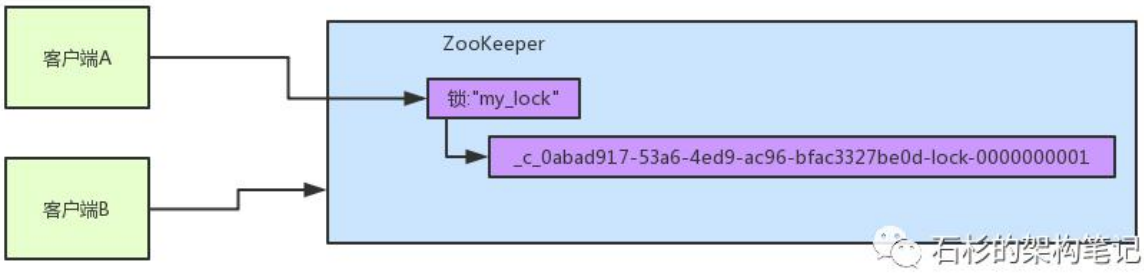
参见上图。zk 里有一把锁，这个锁就是 zk 上的一个节点。然后呢，两个客户端都要来获取这个锁，具体是怎么来获取呢?

咱们就假设客户端 A 抢先一步，对 zk 发起了加分布式锁的请求，这个加锁请求是用到了 zk 中的一个特殊的概念，叫做 “临时顺序节点”。

简单来说，就是直接在 "my_lock" 这个锁节点下，创建一个顺序节点，这个顺序节点有 zk 内部自行维护的一个节点序号。

比如说，第一个客户端来搞一个顺序节点，zk 内部会给起个名字叫做：xxx-000001。然后第二个客户端来搞一个顺序节点，zk 可能会起个名字叫做：xxx-000002。大家注意一下，**最后一个数字都是依次递增的**，从 1 开始逐次递增。zk 会维护这个顺序。

所以这个时候，假如说客户端 A 先发起请求，就会搞出来一个顺序节点，大家看下面的图，Curator 框架大概会弄成如下的样子：



大家看，客户端 A 发起一个加锁请求，先会在你要加锁的 node 下搞一个临时顺序节点，这一大坨长长的名字都是 Curator 框架自己生成出来的。

然后，那个最后一个数字是 "1"。大家注意一下，因为客户端 A 是第一个发起请求的，所以给他搞出来的顺序节点的序号是 "1"。

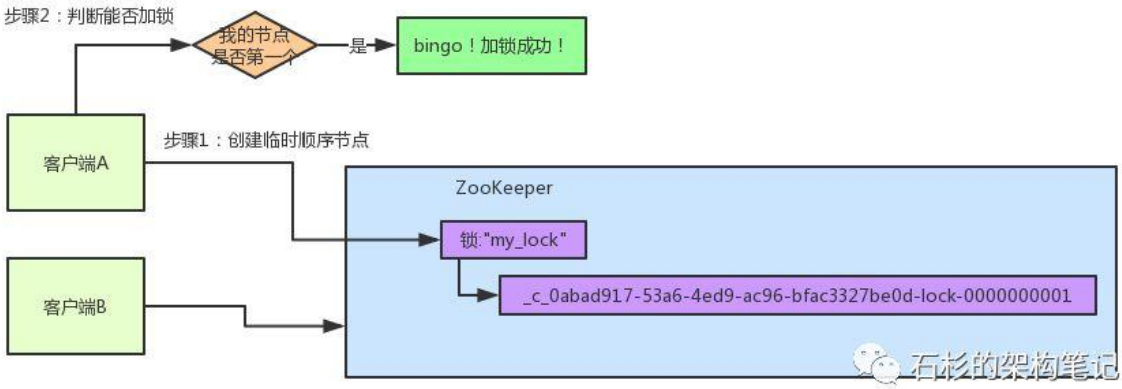
接着客户端 A 创建完一个顺序节点。还没完，他会查一下 "my_lock" 这个锁节点下的所有子节点，并且这些子节点是按照序号排序的，这个时候他大概会拿到这么一个集合：

```
[
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
]
```


接着客户端 A 会走一个关键性的判断，就是说：唉！兄弟，这个集合里，我创建的那个顺序节点，是不是排在第一个啊?

如果是的话，那我就可以加锁了啊！因为明明我就是第一个来创建顺序节点的人，所以我就是第一个尝试加分布式锁的人啊！

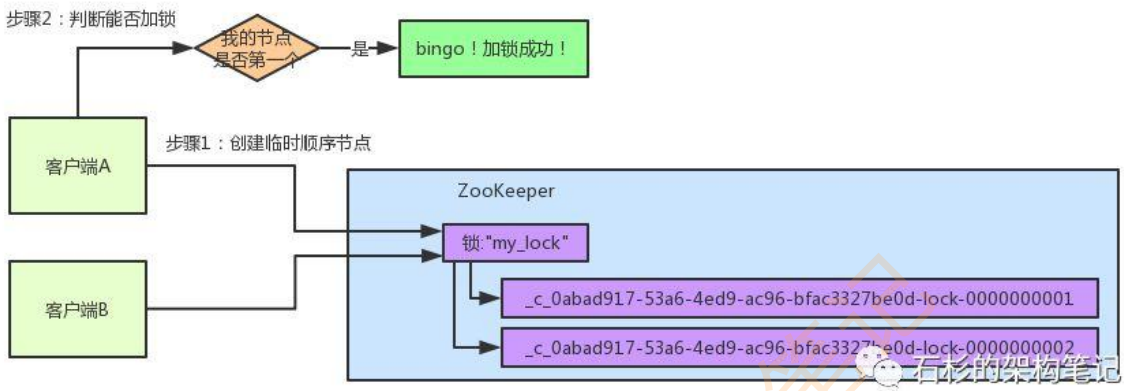
bingo！加锁成功！大家看下面的图，再来直观的感受一下整个过程。



接着假如说，客户端 A 都加完锁了，客户端 B 过来想要加锁了，这个时候他会干一样的事儿：先是在 "my_lock" 这个锁节点下创建一个**临时顺序节点**，此时名字会变成类似于：

```
_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002
```


大家看看下面的图：



客户端 B 因为是第二个来创建顺序节点的，所以 zk 内部会维护序号为 "2"。

接着客户端 B 会走加锁判断逻辑，查询 "my_lock" 锁节点下的所有子节点，按序号顺序排列，此时他看到的类似于：

```
[
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001",
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002"
]
```


同时检查自己创建的顺序节点，是不是集合中的第一个？

明显不是啊，此时第一个是客户端 A 创建的那个顺序节点，序号为 "01" 的那个。所以加锁失败！

加锁失败以后，客户端 B 就会通过 ZK 的 API，对他的上一个顺序节点加一个监听器。zk 天然就可以实现对某个节点的监听。

我们举例说明，客户端 B 的顺序节点是：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002"
```

他的上一个顺序节点，不就是下面这个吗？

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```

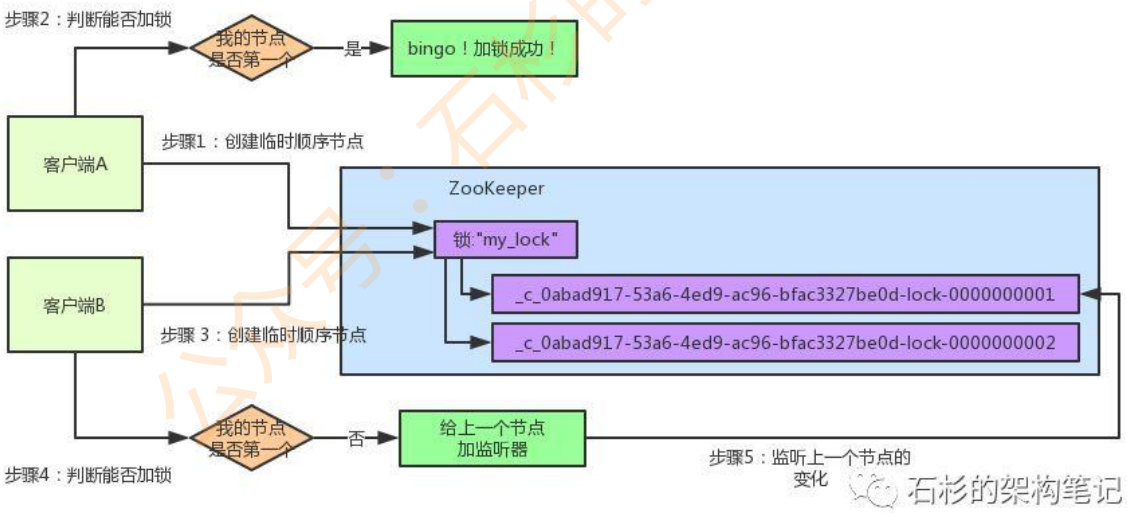

也就是客户端 A 创建的那个顺序节点！

所以，客户端 B 会对：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```


这个节点加一个监听器，监听这个节点是否被删除等变化！

说了那么多，老规矩，给大家来一张图，直观的感受一下：



接着，客户端 A 加锁之后，可能处理了一些代码逻辑，然后就会释放锁。那么，释放锁是个什么过程呢？

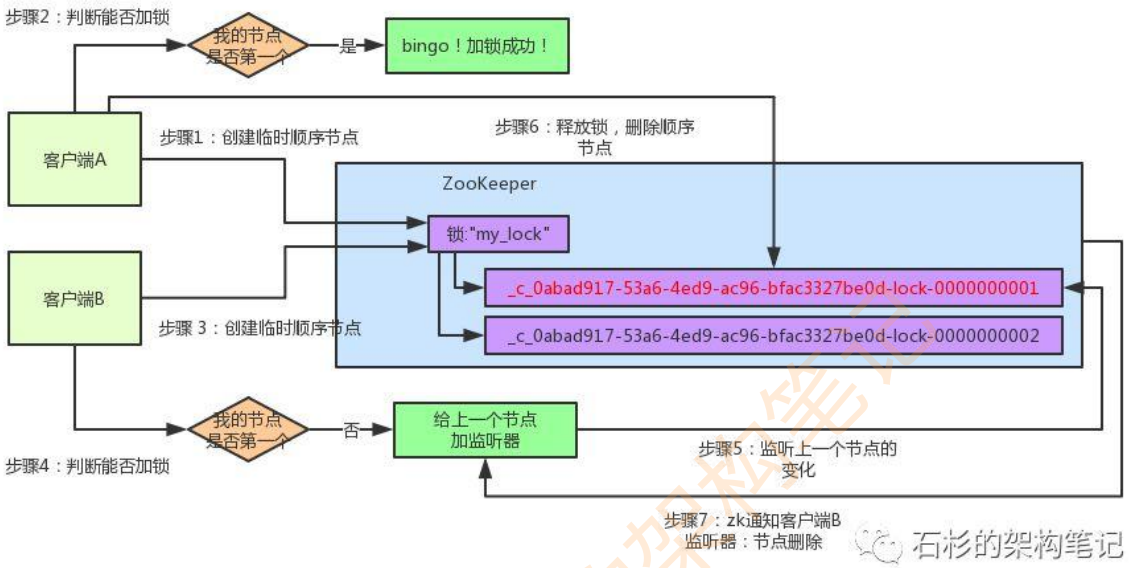
其实很简单，就是把自己在 zk 里创建的那个顺序节点，也就是：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```

这个节点给删除。

删除了那个节点之后，zk 会负责通知监听这个节点的监听器，也就是客户端 B 之前加的那个监听器，说：兄弟，你监听的那个节点被删除了，有人释放了锁。

我们一起来看看下面的图，体会一下这个过程：



此时客户端 B 的监听器感知到了上一个顺序节点被删除，也就是排在他之前的某个客户端释放了锁。

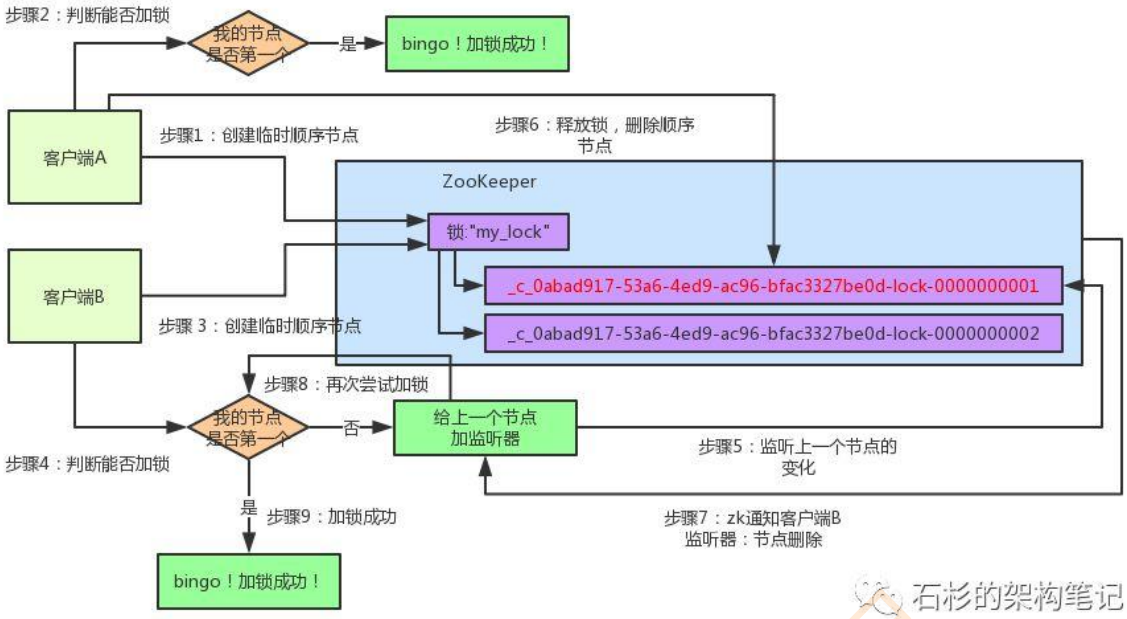
此时，就会通知客户端 B 重新尝试去获取锁，也就是获取 "my_lock" 节点下的子节点集合，此时为：

```
[
_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002
]
```


集合里此时只有客户端 B 创建的唯一的一个顺序节点了！

然后呢，客户端 B 一判断，**发现自己居然是集合中的第一个顺序节点**，bingo！可以加锁了！直接完成加锁，运行后续的业务代码即可，运行完了之后再次释放锁。

最后，再给大家来一张图，大伙儿顺着图仔细的捋一捋这整个过程：



三、总结

其实如果有客户端 C、客户端 D 等 N 个客户端争抢一个 zk 分布式锁，原理都是类似的。

- 大家都是上来直接创建一个锁节点下的一个接一个的临时顺序节点
- 如果自己不是第一个节点，就对自己上一个节点加监听器
- 只要上一个节点释放锁，自己就排到前面去了，相当于是一个排队机制。

而且用临时顺序节点的另外一个用意就是，如果某个客户端创建临时顺序节点之后，不小心自己宕机了也没关系，zk 感知到那个客户端宕机，会自动删除对应的临时顺序节点，相当于自动释放锁，或者是自动取消自己的排队。

最后，咱们来看下用 Curator 框架进行加锁和释放锁的一个过程：

```
// 定义锁节点名称
InterProcessMutex lock =
    InterProcessMutex(client, "/locks/my_lock");

lock.acquire(); // 加锁

// 业务代码逻辑...

lock.release(); // 释放锁
```

其实用开源框架就是这点好，方便。这个 Curator 框架的 zk 分布式锁的加锁和释放锁的实现原理，其实就是上面我们说的那样子。

但是如果你要手动实现一套那个代码的话。还是有点麻烦的，要考虑到各种细节，异常处理等等。所以大家如果考虑用 zk 分布式锁，可以参考下本文的思路。

兄弟，用大白话给你讲小白都能看懂的分布式系统容错架构

> **作者:中华石杉** 原文地址

目录

> “ 这篇文章，我们将用非常浅显易懂的语言，跟大家聊聊大规模分布式系统的容错架构设计。虽然定位是有 “分布式”、“容错架构” 等看起来略显复杂的字眼，但是咱们还是按照老规矩：大白话 + 手绘数张彩图，逐步递进，让每个同学都能看懂这种复杂架构的设计思想。

1、TB 级数据放在一台机器上：难啊！

咱们就用分布式存储系统举例，来聊一下容错架构的设计。

首先，我们来瞧瞧，到底啥是分布式存储系统呢?

其实特别的简单，咱们就用数据库里的一张表来举例。

比如你手头有个数据库，数据库里有一张特别大的表，里面有几十亿，甚至上百亿的数据。

更进一步说，假设这一张表的数据量多达几十个 TB，甚至上百个 TB，这时你觉得咋样?

当然是内心感到恐慌和无助了，因为如果你用 MySQL 之类的数据库，单台数据库服务器上的磁盘可能都不够放这一张表的数据！

咱们就来看看下面的这张图，来感受一下。

2、到底啥是分布式存储?

所以，假如你手头有一个超大的数据集，几百 TB！那你还是别考虑传统的数据库技术来存放了。

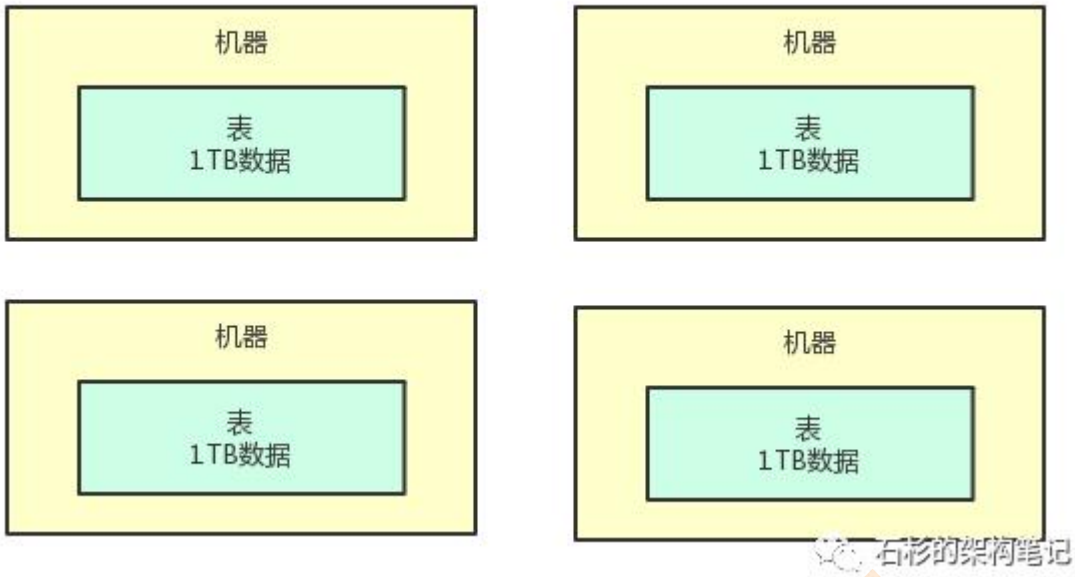
因为用一台数据库服务器可能根本都放不下，所以我们考虑一下分布式存储技术？对了！这才是解决这个问题的办法。

咱们完全可以搞多台机器嘛！比如搞 20 台机器，每台机器上就放 1/20 的数据。

举个例子，比如总共 20TB 的数据，在每台机器上只要把 1TB 就可以了，1TB 应该还好吧？每台机器都可以轻松加愉快的放下这么多数据了。

所以说，把一个超大的数据集拆分成多片，给放到多台机器上去，这就是所谓的分布式存储。

咱们再看看下面的图。



3、那么啥又是分布式存储系统呢?

那分布式存储系统是啥呢?

分布式存储系统，当然就是负责把一个超大数据集拆分成多块，然后放到多台机器上来存储，接着统一管理这些分散在多台机器上存储的数据的一套系统。

比如说经典的 hadoop 就是这类系统，然后 fastdfs 也是类似的。

如果你可以脑洞打开，从思想本质共通的层面出发，那你会发现，其实类似 elasticsearch、redis cluster 等等系统，他本质都是如此。

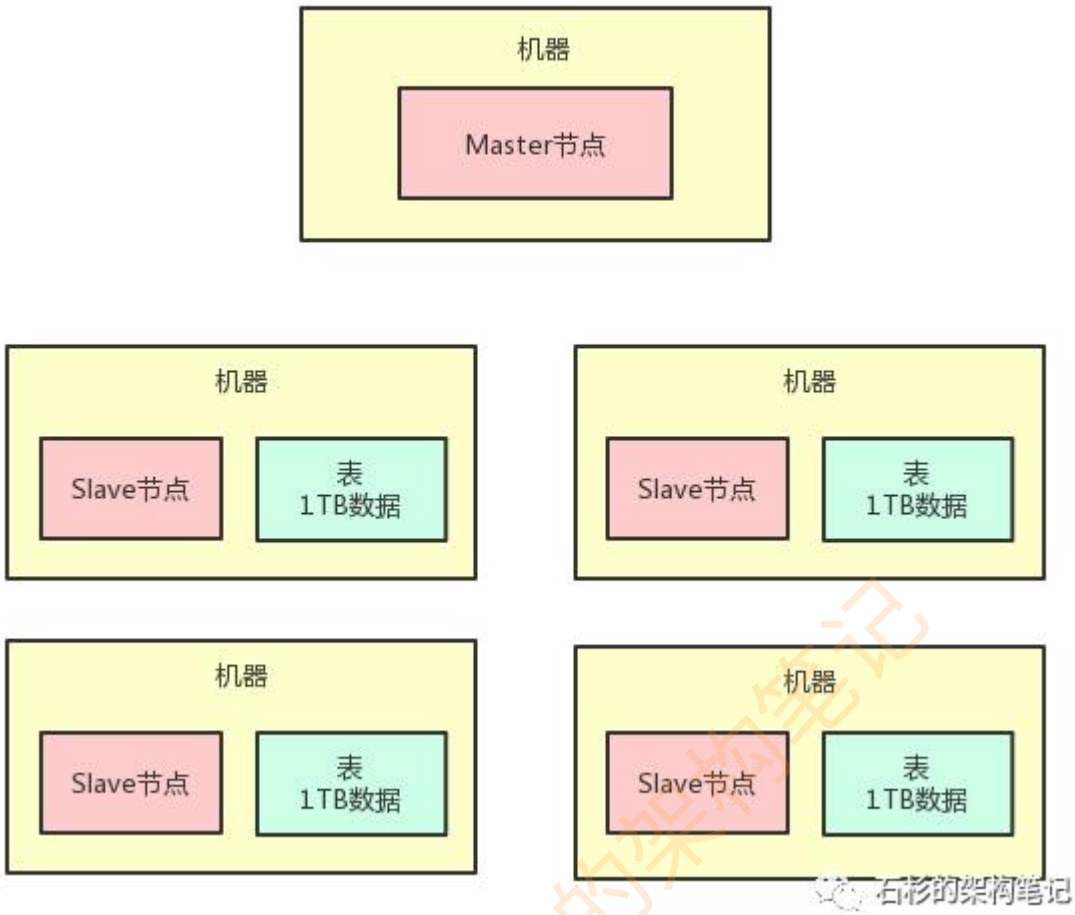
这些都是基于分布式的系统架构，把超大数据拆分成多片给你存放在多台机器上。

咱们这篇文章是从分布式系统架构层面出发，不拘泥于任何一种技术，所以姑且可以设定：这套分布式存储系统，有两种进程。

一个进程是 Master 节点，就在一台机器上，负责统一管控分散在多台机器上的数据。

另外一批进程叫做 Slave 节点，每台机器上都有一个 Slave 节点，负责管理那台机器上的数据，跟 Master 节点进行通信。

咱们看看下面的图，通过图再来直观的看看上面的描述。



4、天哪！某台机器宕机了咋办？

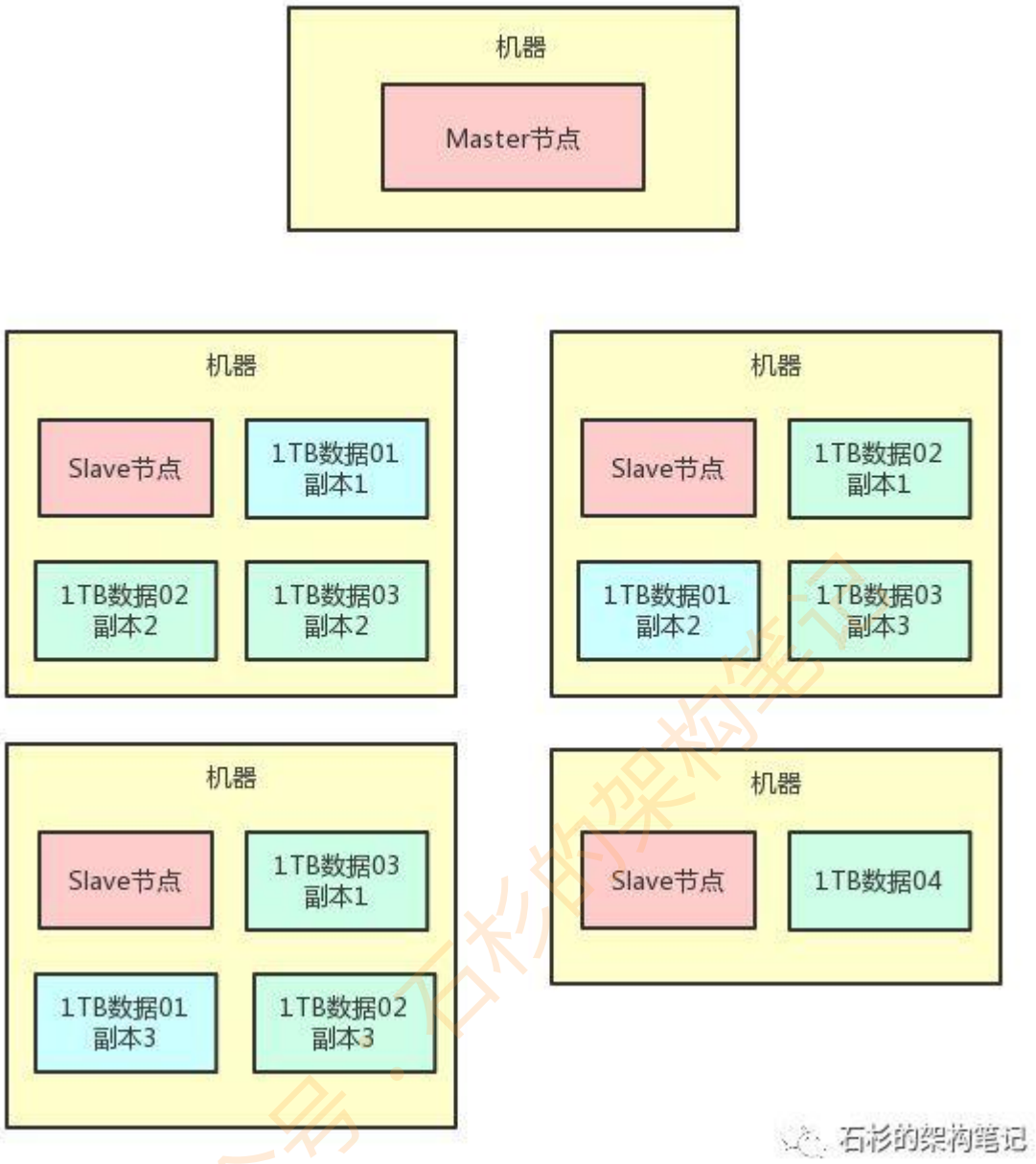
这个时候又有一个问题了，那么万一上面那 20 台机器上，其中 1 台机器宕机了咋整呢？

这就尴尬了，兄弟，这会导致本来完整的一份 20TB 的数据，最后有 19TB 还在了，有 1TB 的数据就搞丢了，因为那台机器宕机了啊。

所以说你当然不能允许这种情况的发生，这个时候就必须做一个数据副本的策略。

比如说，我们完全可以给每一台机器上的那 1TB 的数据做 2 个副本的冗余，放在别的机器上，然后呢，万一说某一台机器宕机，没事啊，因为其他机器上还有他的副本。

我们来看看这种多副本冗余的架构设计图。

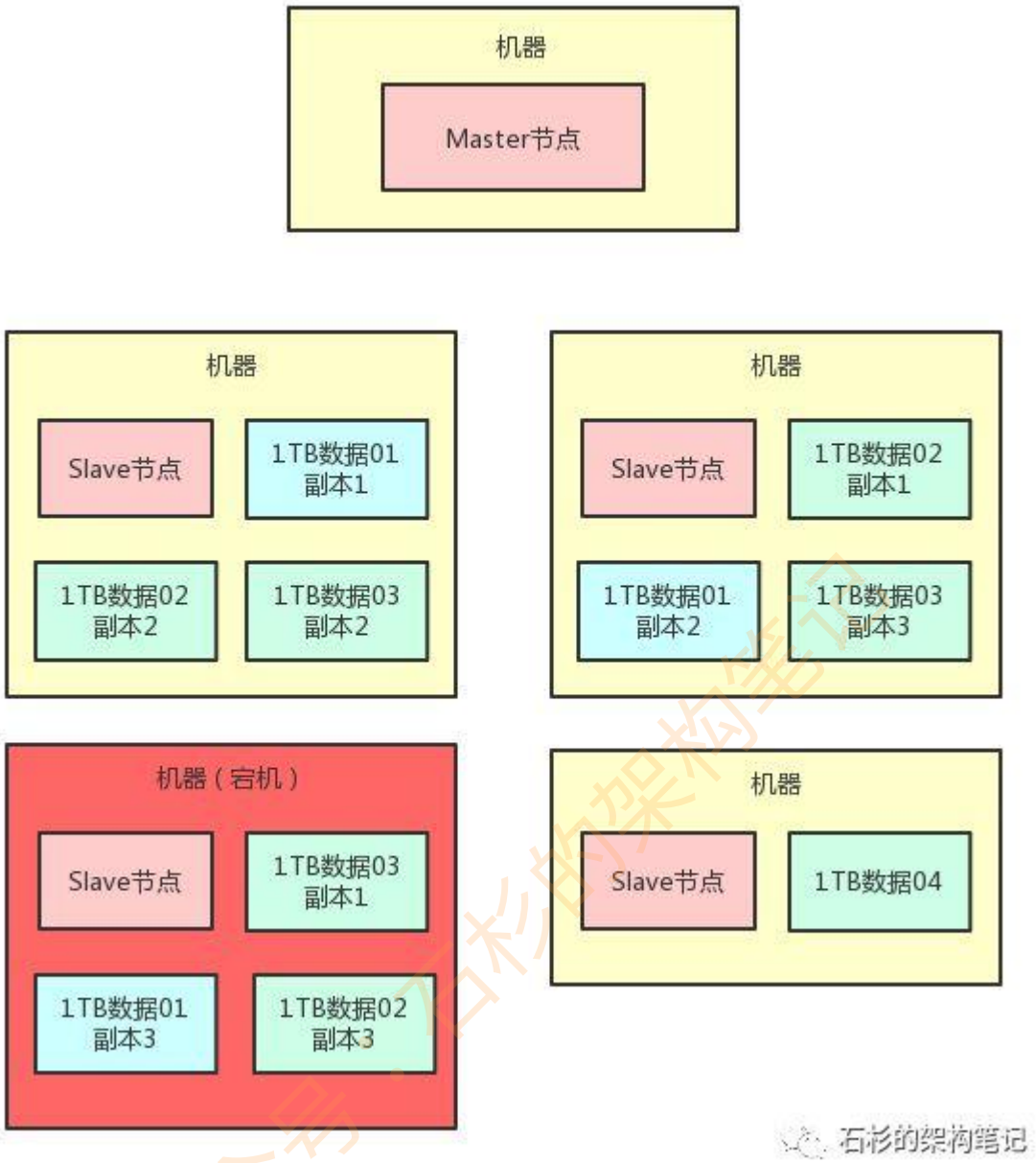


上面那个图里的浅蓝色的“1TB 数据 01”，代表的是 20TB 数据集中的第一个 1TB 数据分片。

图中可以看到，他就有 3 个副本，分别在三台机器中都有浅蓝色的方块，代表了他的三个副本。

这样的话，一份数据就有了 3 个副本了。其他的数据也是类似。

这个时候我们假设有一台机器宕机了，比如下面这台机器宕机，必然会导致“1TB 数据 01”这个数据分片的其中一个数据副本丢失。如下图所示：



那这个时候要紧吗？不要紧，因为“1TB 数据 01”这个数据分片，他还有另外 2 个副本在存活的两台机器上呢！

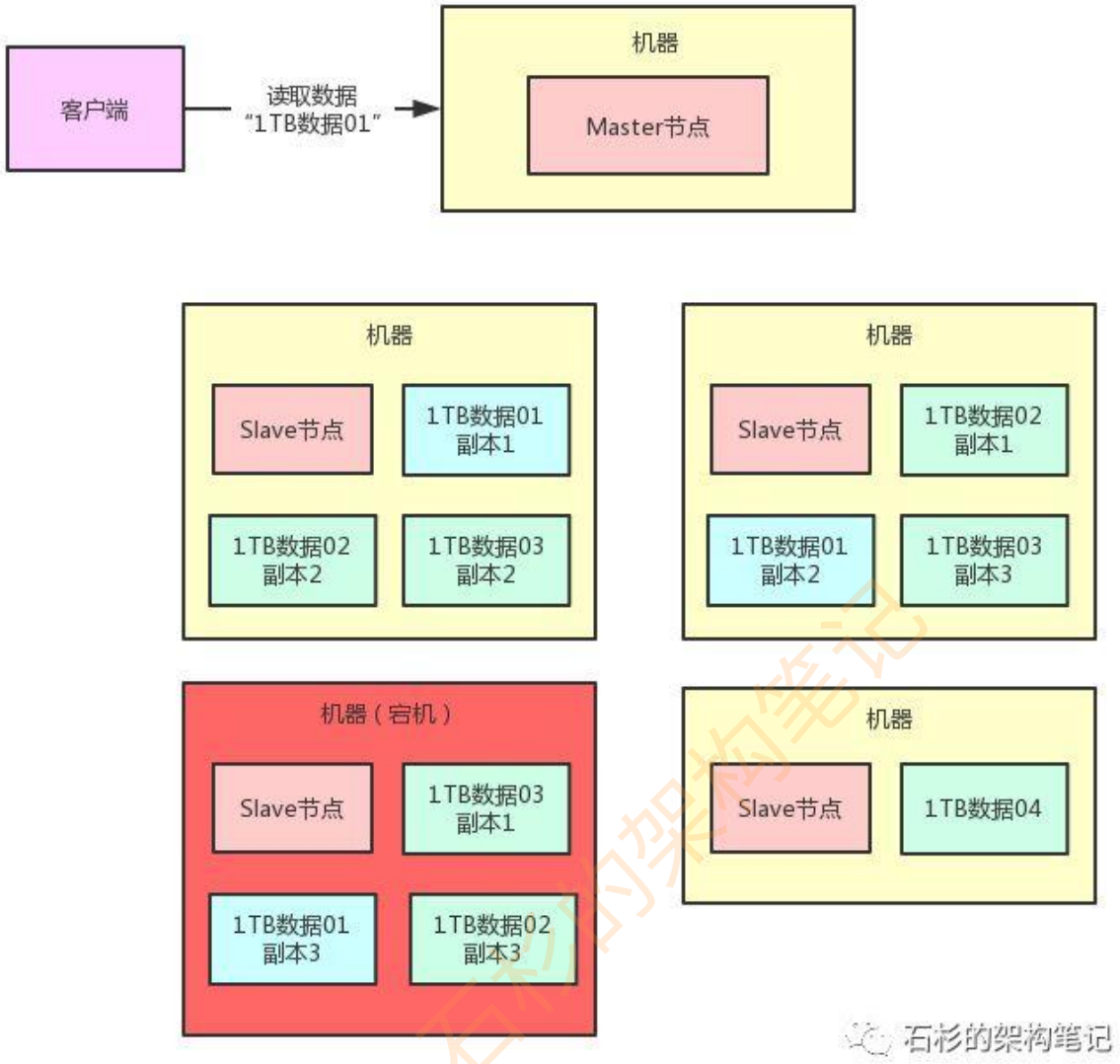
所以如果有人要读取数据，完全可以从另外两台机器上随便挑一个副本来读取就可以了，数据不会丢的，不要紧张，大兄弟。

5、Master 节点如何感知到数据副本消失？

现在有一个问题，比如说有个兄弟要读取“1TB 数据 01”这个数据分片，那么他就会找 Master 节点，说：

“你能不能告诉我“1TB 数据 01”这个数据分片人在哪里啊？在哪台机器上啊？我需要读他啊！”

我们来看看下面的图。



那么这个时候，Master 节点就需要从 “1TB 数据 01” 的 3 个副本里选择一个出来，告诉人家说：

“兄弟，在哪台哪台机器上，有 1 个副本，你可以去那台机器上读 “1TB 数据 01” 的一个副本就 ok 了。”

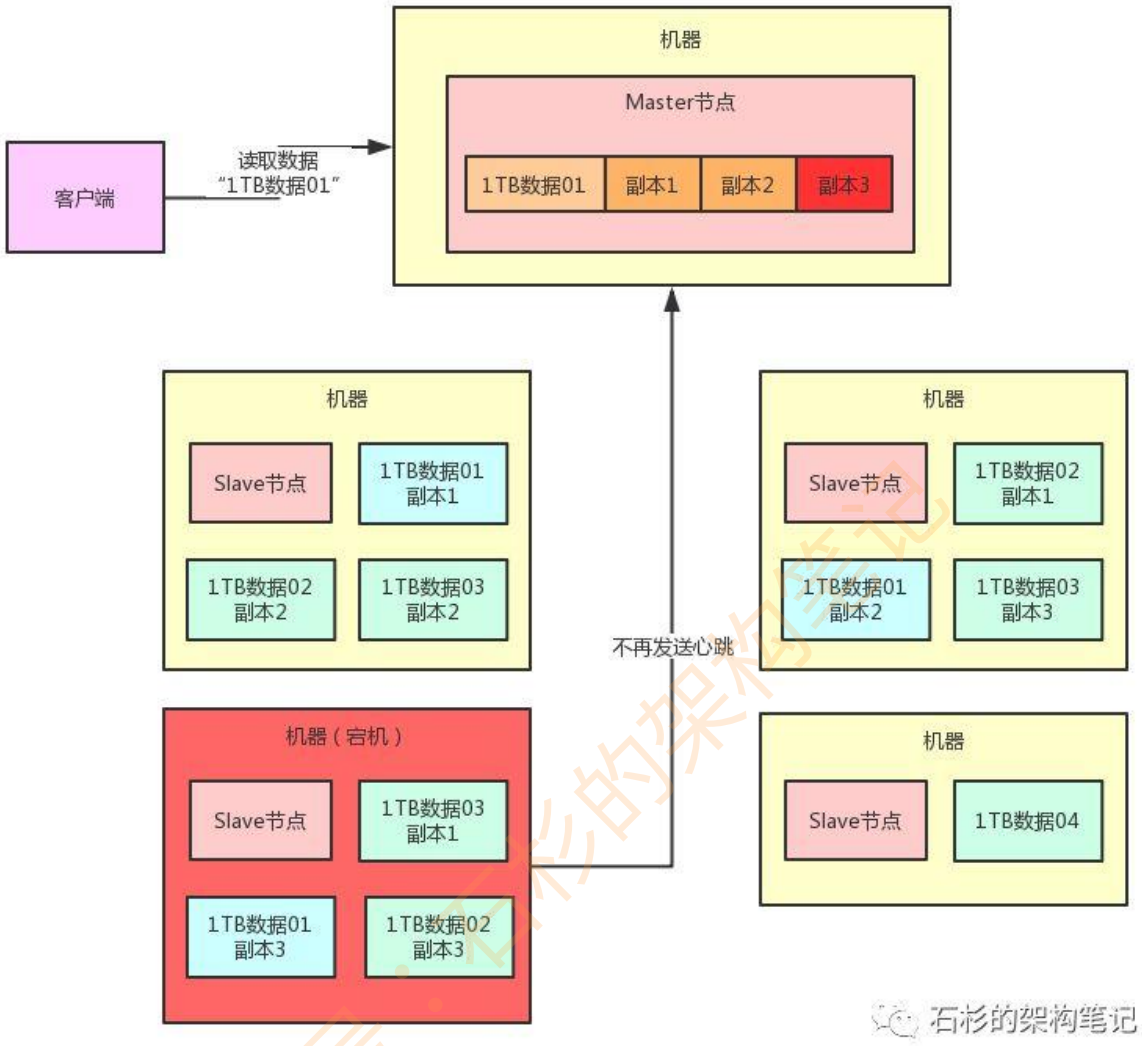
但是现在的问题是，Master 节点此时还不知道 “1TB 数据 01” 的副本 3 已经丢失了，那万一 Master 节点还是通知人家去读取一个已经丢失的副本 3，肯定是不可以的。

所以，我们怎么才能让 Master 节点知道副本 3 已经丢失了呢?

其实也很简单，每台机器上负责管理数据的 Slave 节点，都每隔几秒（比如说 1 秒）给 Master 节点发送一个心跳。

那么，一旦 Master 节点发现一段时间（比如说 30 秒内）没收到某个 Slave 节点发送过来的心跳，此时就会认为这个 Slave 节点所在机器宕机了，那台机器上的数据副本都丢失了，然后 Master 节点就不会告诉别人去读那个丢失的数据副本。

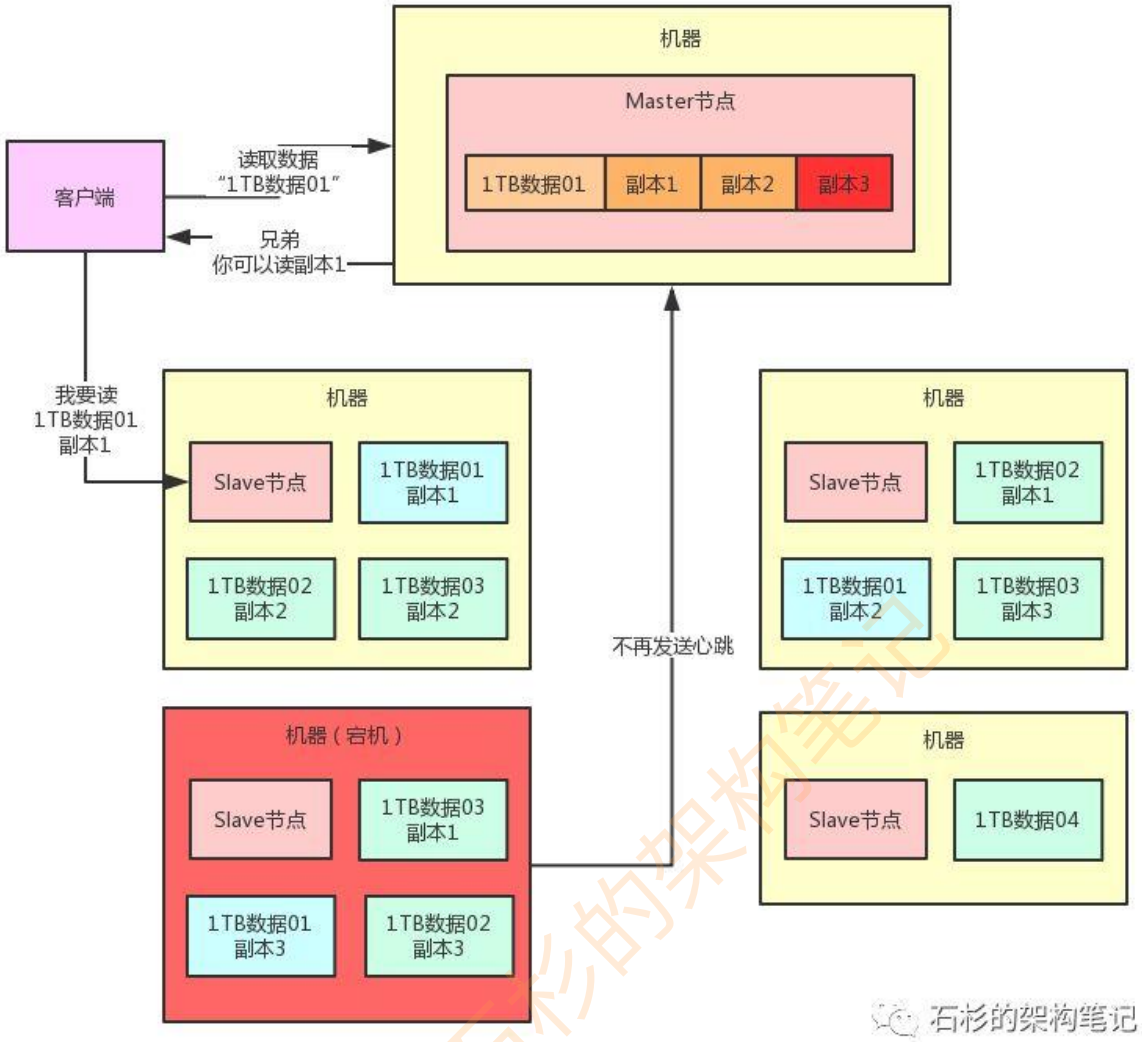
大家看看下面的图，一旦 Slave 节点宕机，Master 节点收不到心跳，就会认为那台机器上的副本 3 就已经丢失了，此时绝对不会让别人去读那台宕机机器上的副本 3。



那么此时，Master 节点就可以通知人家去读 “1TB 数据 01” 的副本 1 或者副本 2，哪个都行，因为那两个副本其实还是在的。

举个例子，比如可以通知客户端去读副本 1，此时客户端就可以找那台机器上的 Slave 节点说要读取那个副本 1。

整个过程如下图所示。



6、复制副本保持足够副本数量

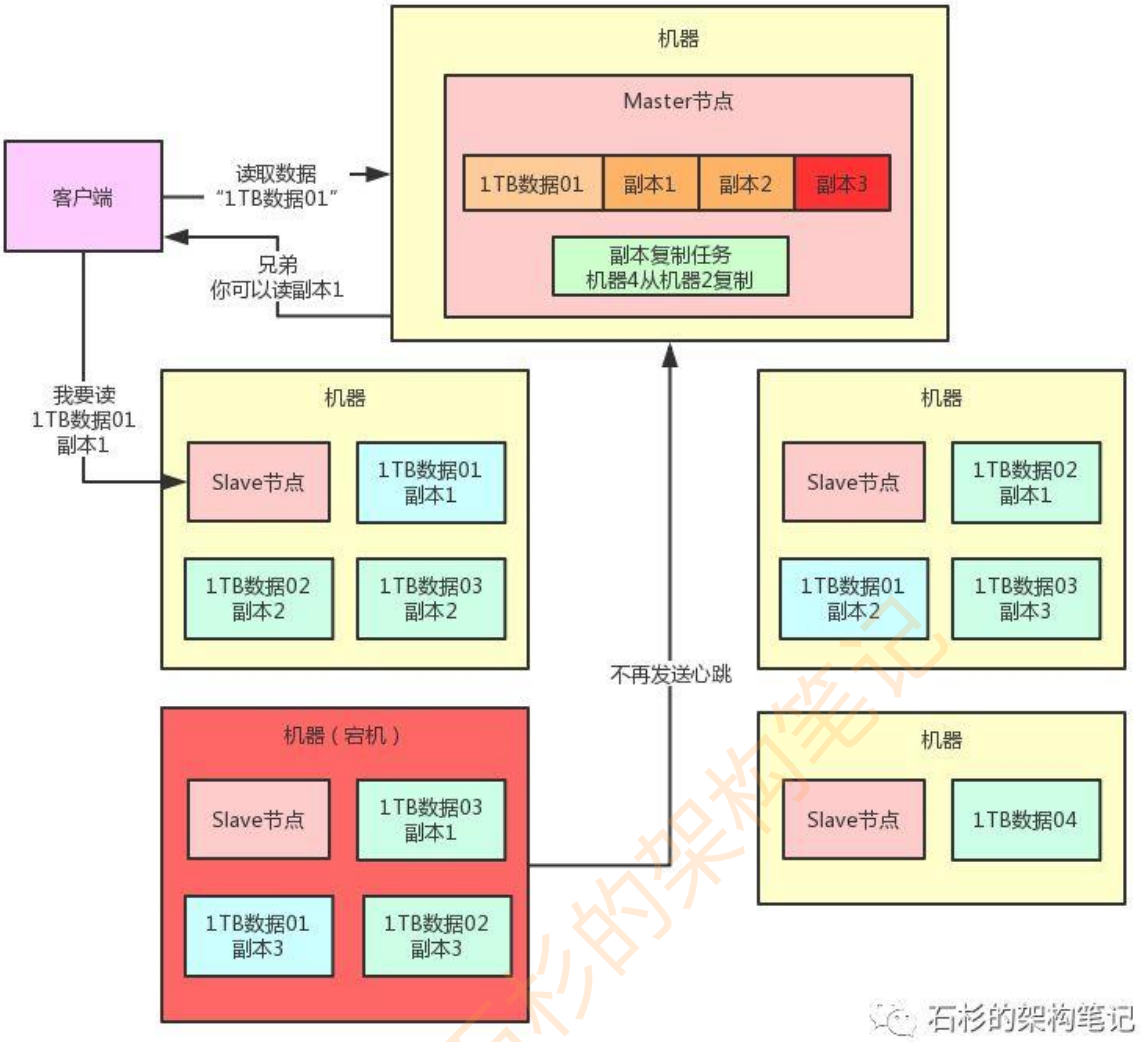
这个时候又有另外一个问题，那就是 “1TB 数据 01” 这个数据分片此时只有副本 1 和副本 2 这两个副本了，这就不足够 3 个副本啊。

因为我们预设的是每个数据分片都得有 3 个副本的。大家想想，此时如何给这个数据分片增加 1 个副本呢?

很简单，Master 节点一旦感知到某台机器宕机，就能感知到某个数据分片的副本数量不足了。

此时，就会生成一个副本复制的任务，挑选另外一台机器来从有副本的机器去复制一个副本。

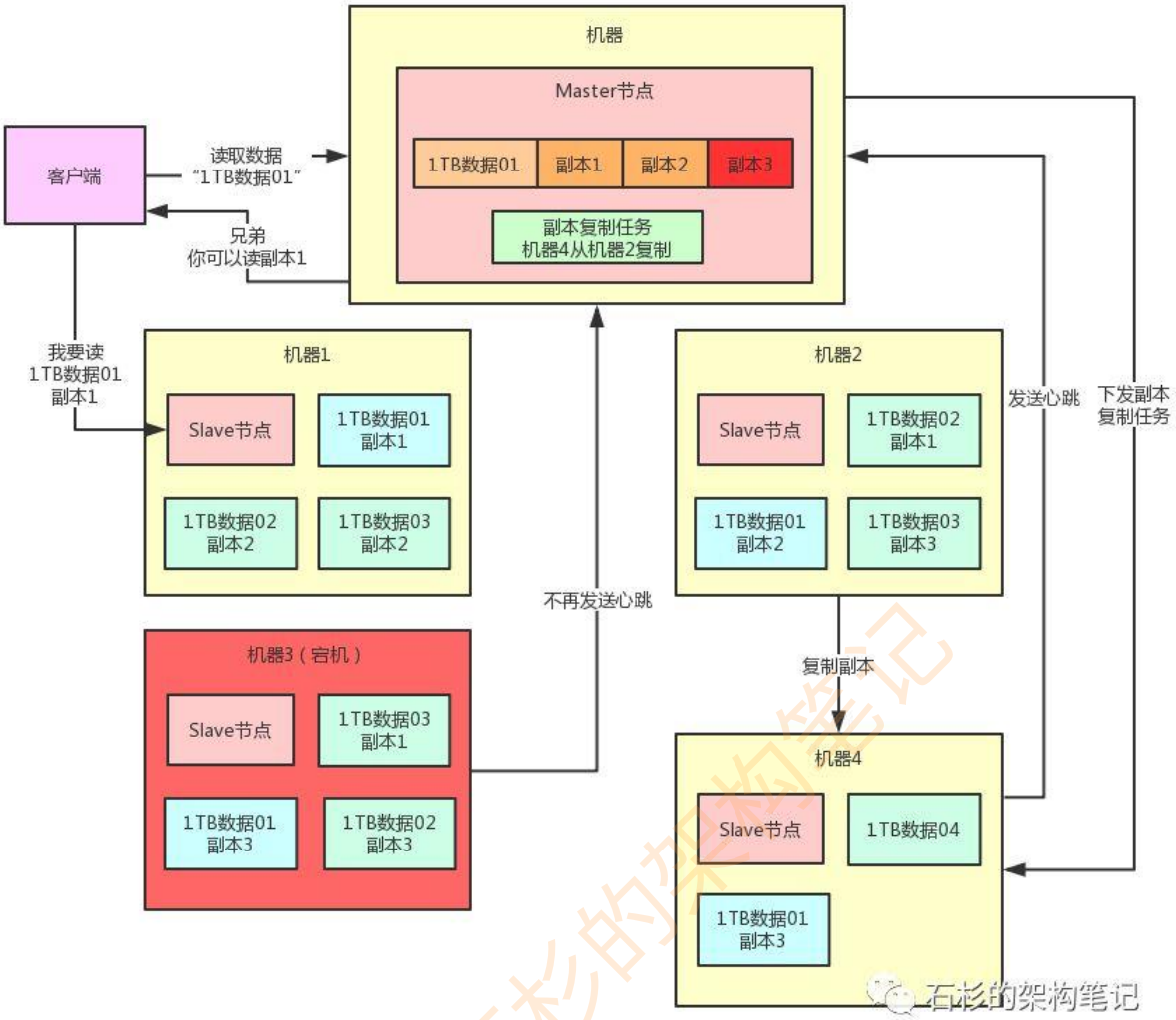
比如看下面的图，可以挑选第四台机器从第二台机器去复制一个副本。



但是，现在这个复制任务是有了，我们怎么让机器 4 知道呢？

其实也很简单，机器 4 不是每秒都会发送一次心跳么？当机器 4 发送心跳过去的时候，Master 节点就通过心跳响应把这个复制任务下发给机器 4，让机器 4 从机器 2 复制一个副本好了。

同样，我们来一张图，看看这个过程：



看上图，现在机器 4 上是不是又多了一个 “1TB 数据 01” 的副本 3？ 那么 “1TB 数据 01” 这个数据分片是不是又变成 3 个副本了?

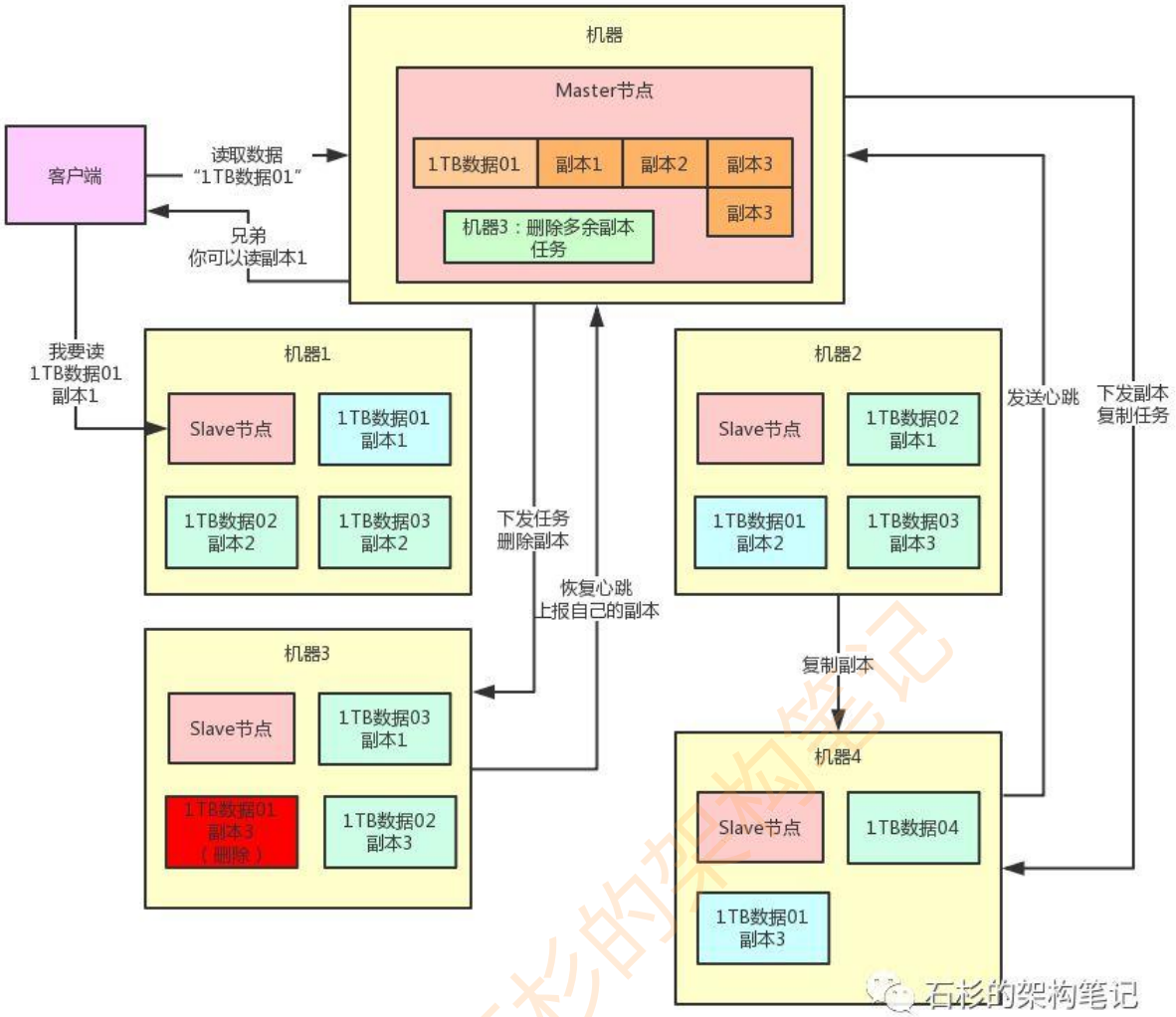
7、删除多余副本

那反过来，如果说此时机器 3 突然恢复了，他上面也有一个 “1TB 数据 01” 的副本 3，相当于此时 “1TB 数据 01” 就有 4 个副本了，副本不就多余了吗?

没关系，一旦 Master 节点感知到机器 3 复活，会发现副本数量过多，此时会生成一个删除副本任务。

他会在机器 3 发送心跳的时候，下发一个删除副本的指令，让机器 3 删除自己本地多余的副本就可以了。这样，就可以保持副本数量只有 3 个。

一样的，大家来看看下面的图。



8、全文总结 好了，到这里，通过超级大白话的讲解，还有十多张图的渐进式演进说明，相信大家以前即使不了解分布式系统，都绝对能理解一个**分布式系统的完整的数据容错架构**是如何设计的了。

实际上，这种数据分片存储 、多副本冗余、宕机感知、自动副本迁移、多余副本删除，这套机制，对于 hadoop、elasticsearch 等很多系统来说，都是类似的。

所以笔者在这里强烈建议大家，一定好好吸收一下这种分布式系统、中间件系统底层数据容错架构的思想。

这样，以后学习类似的一些技术的时候，对他们的原理、思想都会感到一种似曾相识的感觉。

拜托，面试请不要再问我分布式搜索引擎的架构原理

> **作者:中华石杉** 原文地址

目录

- （1）倒排索引到底是啥?
- （2）什么叫分布式搜索引擎?
- （3）ElasticSearch 的数据结构
- （4）Shard 数据分片机制
- （5）Replica 多副本数据冗余机制
- （6）全文总结

> “ 这篇文章，我们来聊一下最近这一两年行业内 Java 高级工程师面试的时候尤为常见的一个问题：谈谈你对分布式搜索引擎的理解，聊聊他的架构原理?

很多同学可能从来没接触过这个东西，所以本文我们就以现在最火最流行的 Elasticsearch 为例，来聊一下分布式搜索引擎的核心架构原理。

（1）倒排索引到底是啥?

要了解分布式搜索引擎，先了解一下搜索这个事儿吧，搜索这个技术领域里最入门级别的一个概念就是倒排索引。

我们先简单说一下倒排索引是个什么东西。

假如说你现在不用搜索引擎，单纯使用数据库来存放和搜索一些数据，比如说放了一些论坛的帖子数据吧，那么这个数据的格式大致如下：

id	title	content
1	Java 好用吗?	Java 是非常非常好的一门语言。。。。
2	大家一起来学 Java	我这儿有一些很好的 Java 学习资源，比如说。。。
3	一次 Java 面试经验	去年这个时候，我学了 Java，今年开始了面试。。。

那么这个时候，比如我们要是用数据库来进行搜索包含 “Java” 这个关键字的所有帖子，大致 SQL 如下：

```
select * from articles where title like "%Java%" or content like "%java%"
```


咱们姑且不论这个数据库层面也有支持全文检索的一些特殊索引类型，或者数据库层面是怎么执行的，这个不是本文讨论的重点，你就看看数据库的数据格式以及搜索的方式就好了。

但是如果你通过搜索引擎类的技术来存放帖子的内容，他是可以建立倒排索引的。

也就是说，你把上述的几行数据放到搜索引擎里，这个倒排索引的数据大致看起来如下：

关键词	id
Java	[1, 2, 3]
语言	[1]
面试	[3]
资源	[2]

所谓的倒排索引，就是把你的数据内容先分词，每句话分成一个一个的关键词，然后记录好每个关键词对应出现在了哪些 id 标识的数据里。

那么你要搜索包含 “Java” 关键词的帖子，直接扫描这个倒排索引，在倒排索引里找到 “Java” 这个关键词对应的那些数据的 id 就好了。

然后你可以从其他地方根据这几个 id 找到对应的数据就可以了，这个就是倒排索引的数据格式以及搜索的方式，上面这种利用倒排索引查找数据的方式，也被称之为全文检索。

（2）什么叫做分布式搜索引擎?

其实要知道什么叫做分布式搜索引擎，你首先得知道，假如我们就用一台机器部署一个搜索引擎系统，然后利用上述的那种倒排索引来存储数据，同时支持一些全文检索之类的搜索功能，那么会有什么问题?

其实还是很简单，假如说你现在要存储 1TB 的数据，那么放在一台机器还是可以的。

但是如果你要存储超过 10TB，100TB，甚至 1000TB 的数据呢？你用一台机器放的下吗?

当然是放不下的了，你的机器磁盘空间是不够的。

大家看一下下面的图：

所以这个时候，你就得用分布式搜索引擎了，也就是要使用多台机器来部署搜索引擎集群。

比如说，假设你用的是 Elasticsearch（后面简写为：ES）。

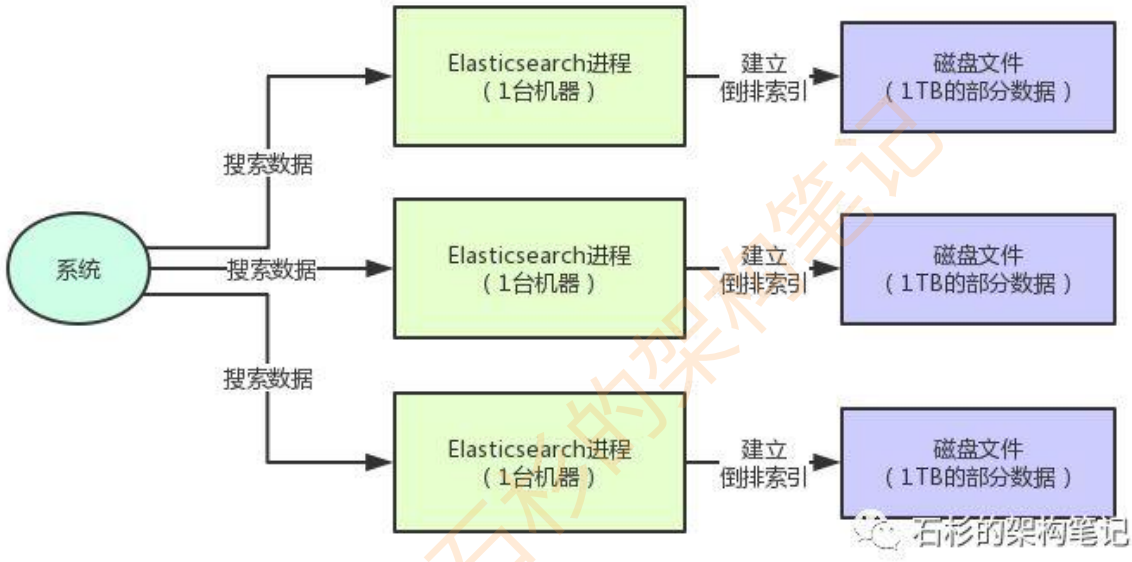
现在你总共有 3TB 的数据，那么你搞 3 台机器，每台机器上部署一个 ES 进程，管理那台机器上的 1TB 数据就可以了。

这样不就可以把 3TB 的数据分散在 3 台机器上来存储了？这不就是索引数据的分布式存储吗？

而且，你在搜索数据的时候，不就可以利用 3 台机器来对分布式存储后的数据进行搜索了？每台机器上的 ES 进程不都可以对一部分数据搜索？这不就是分布式的搜索？

是的，这就是所谓的分布式搜索引擎：把大量的索引数据拆散成多块，每台机器放一部分，然后利用多台机器对分散之后的数据进行搜索，所有操作全部是分布在多台机器上进行，形成了完整的分布式的架构。

同样，我们来看下面的图，直观的感受一下。



（3）Elasticsearch 的数据结构

如果你要是使用 Elasticsearch 这种分布式搜索引擎，必须要熟悉他的一些专业的技术名词，描述他的一些数据结构。

比如说 “index” 这个东西，他是索引的意思，其实他有点类似于数据库里的一张表，大概对应表的那个概念。

比如你搞一个专门存放帖子的索引，然后他有 id、title、content 几个 field，这个 field 大致就是他的一个字段。

然后还有一个概念，就是 document，这个就代表了 index 中的一条数据。

下面就是一个 document，这个 document 可以写到 index 里去，算是 index 里的一条数据。

而且写到 es 之后，这条数据的内容就会拆分为倒排索引的数据格式来存储。

id	title	content
1	Java 好用吗?	Java 是非常非常好的一门语言。。。。

（4）Shard 数据分片机制

那么这个时候大家考虑一下，比如说你有一个 index，专门存放论坛里的帖子，现在论坛里的帖子有 1 亿，占用了 1TB 的磁盘空间，这个还好说。

如果这个帖子有 10 亿，100 亿，占用了 10TB、甚至 100TB 的磁盘空间呢?

那你这个 index 的数据还能在一台机器上存储吗？答案明显是不能的。

这个时候，你必须得支持这个 index 的数据分布式存储在多台机器上，利用多台机器的磁盘空间来承载这么大的数据量。

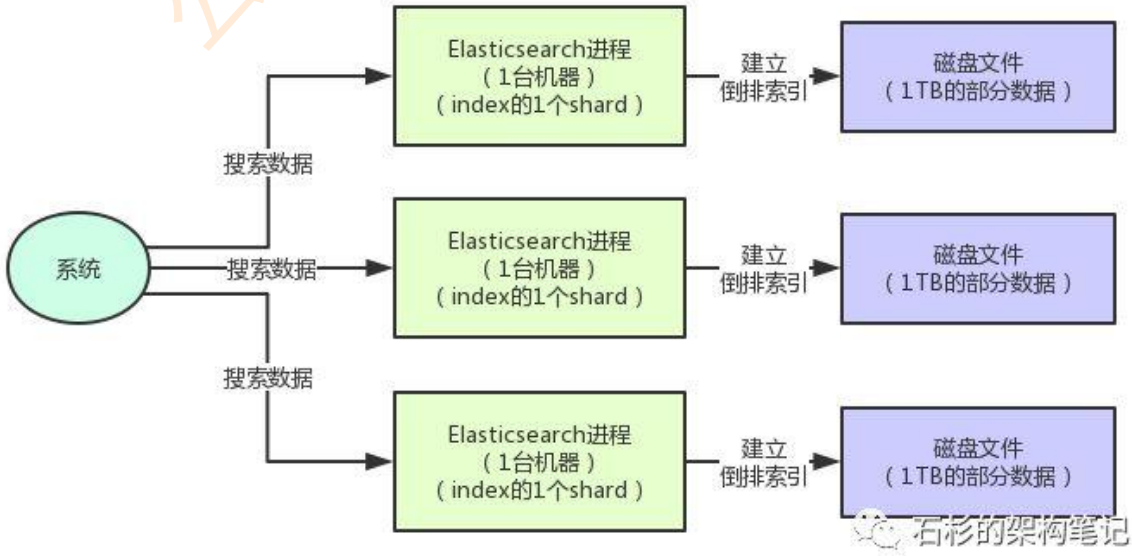
而且，需要保证每台机器上对这个 index 存储的数据量不要太大，因为控制单台机器上这个 index 的数据量，可以保证他的搜索性能更高。

所以这里就引入了一个概念：Shard 数据分片结构。每个 index 你都可以指定创建多少个 shard，每个 shard 就是一个数据分片，会负责存储这个 index 的一部分数据。

比如说 index 里有 3 亿帖子，占据 3TB 数据。然后这个 index 你设置了 3 个 shard。

那么每个 shard 就可以包含一个 1TB 大小的数据分片，每个 shard 在集群里的一台机器上，这样就形成了利用 3 台机器来分布式存储一个 index 的数据的效果了。

大家看下面的图：



现在 index 里的 3TB 数据分布式存储在了 3 台机器上，每台机器上有一个 shard，每个 shard 负责管理这个 index 的其中 1TB 数据的分片。

而且，另外一个好处是，假设我们要对这个 index 的 3TB 数据运行一个搜索，是不是可以发送请求到 3 台机器上去?

3 台机器上的 shard 直接可以分布式的并行对一部分数据进行搜索，起到一个分布式搜索的效果，大幅度提升海量数据的搜索性能和吞吐量。

（5）Replica 多副本数据冗余机制

但是现在有一个问题，假如说 3 台机器中的其中一台宕机了，此时怎么办呢?

是不是这个 index 的 3TB 数据的 1/3 就丢失了？因为上面有 1TB 的数据分片没了。

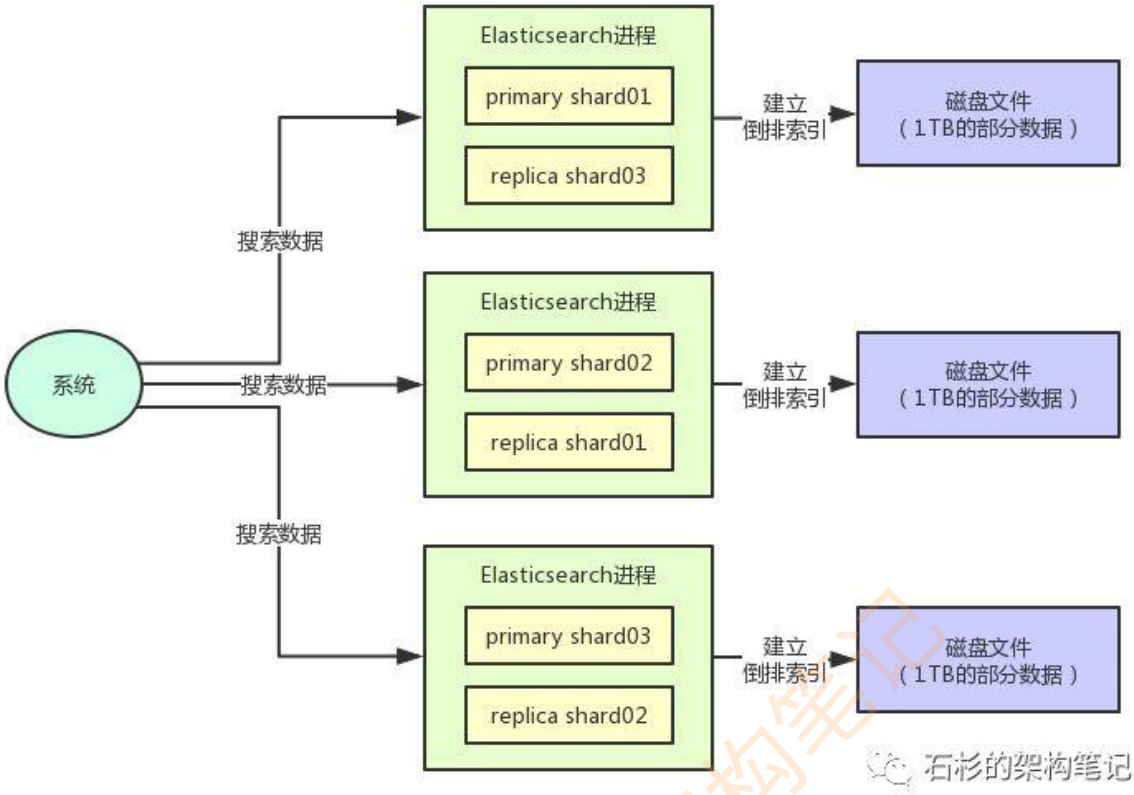
所以说，还需要为了实现高可用使用 Replica 多副本数据冗余机制。

在 Elasticsearch 里，就是支持对每个 index 设置一个 replica 数量的，也就是每个 shard 对应的 replica 副本的数量。

比如说你现在一个 index 有 3 个 shard，你设置对每个 shard 做 1 个 replica 副本，那么此时每个 shard 都会有一个 replica shard。

这个初始的 shard 就是 primary shard，而且 primary shard 和 replica shard 是绝对不会放在一台机器上的，避免一台机器宕机直接一个 shard 的副本也同时丢失了。

我们再来看下面的图，感受一下：



在上述的 replica 机制下，每个 primary shard 都有一个 replica shard 在别的机器上，任何一台机器宕机，都可以保证数据不会丢失，分布式搜索引擎继续可用。

Elasticsearch 默认是支持每个 index 是 5 个 primary shard，每个 primary shard 有 1 个 replica shard 作为副本。

（6）文末总结

好了，本文到这儿就结束了，再来给大伙简单小结。

我们从搜索引擎的倒排索引开始，到单机无法承载海量数据，再到分布式搜索引擎的存储和搜索。

然后我们以优秀的分布式搜索引擎 ES 为例，阐述了 ES 的数据结构，shard 数据分片机制，replica 多副本机制，解释了一下分布式搜索引擎的架构原理。

最后还是强调一下，在 Java 面试尤其是高级 Java 面试中，对于分布式搜索引擎技术的考察越来越重，所以这块技术的重要性，还是不容小觑的！

高阶Java开发必备：分布式系统的唯一id生成算法你了解吗?

作者:中华石杉 原文地址

> “ 之前一篇文章，我们聊了一下分库分表相关的一些基础知识，具体可以参见：《支撑日活百万用户的高并发系统，应该如何设计其数据库架构？》。

这篇文章，我们就接着分库分表的知识，来具体聊一下全局唯一 id 如何生成。

在分库分表之后你必然要面对的一个问题，就是 id 咋生成?

因为要是一个表分成多个表之后，每个表的 id 都是从 1 开始累加自增长，那肯定不对啊。

举个例子，你的订单表拆分为了 1024 张订单表，每个表的 id 都从 1 开始累加，这个肯定有问题了!

你的系统就没办法根据表主键来查询订单了，比如 id = 50 这个订单，在每个表里都有!

所以此时就需要分布式架构下的全局唯一 id 生成的方案了，在分库分表之后，对于插入数据库中的核心 id，不能直接简单使用表自增 id，要全局生成唯一 id，然后插入各个表中，保证每个表内的某个 id，全局唯一。

比如说订单表虽然拆分为了 1024 张表，但是 id = 50 这个订单，只会存在于一个表里。

那么如何实现全局唯一 id 呢？有以下几种方案。

（1）方案一：独立数据库自增 id

这个方案就是说你的系统每次要生成一个 id，都是往一个独立库的一个独立表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个 id。拿到这个 id 之后再往对应的分库分表里去写入。

比如说你有一个 auto_id 库，里面就一个表，叫做 auto_id 表，有一个 id 是自增长的。

那么你每次要获取一个全局唯一 id，直接往这个表里插入一条记录，获取一个全局唯一 id 即可，然后这个全局唯一 id 就可以插入订单的分库分表中。

这个方案的好处就是方便简单，谁都会用。缺点就是单库生成自增 id，要是高并发的话，就会有瓶颈的，因为 auto_id 库要是承载个每秒几万并发，肯定是不现实的了。

（2）方案二：uuid

这个每个人都应该知道吧，就是用 UUID 生成一个全局唯一的 id。

好处就是每个系统本地生成，不要基于数据库来了

不好之处就是，uuid 太长了，作为主键性能太差了，不适合用于主键。

如果你是要随机生成个什么文件名了，编号之类的，你可以用 uuid，但是作为主键是不能用 uuid 的。

（3）方案三：获取系统当前时间

这个方案的意思就是获取当前时间作为全局唯一的 id。

但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。

一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个 id，如果业务上你觉得可以接受，那么也是可以的。

你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号，比如说订单编号：时间戳 + 用户 id + 业务含义编码。

####（4）方案四：snowflake 算法的思想分析

snowflake 算法，是 twitter 开源的分布式 id 生成算法。

其核心思想就是：使用一个 64 bit 的 long 型的数字作为全局唯一 id，这 64 个 bit 中，其中 1 个 bit 是不用的，然后用其中的 41 bit 作为毫秒数，用 10 bit 作为工作机器 id，12 bit 作为序列号。

给大家举个例子吧，比如下面那个 64 bit 的 long 型数字，大家看看

```
0 | 0001100    10100010    10111110    10001001    01011100    00 | 10001 | 1    1001 | 0000    00000000
```


上面第一个部分，是 1 个 bit：0，这个是无意义的

上面第二个部分是 41 个 bit：表示的是时间戳

上面第三个部分是 5 个 bit：表示的是机房 id，10001

上面第四个部分是 5 个 bit：表示的是机器 id，1 1001

上面第五个部分是 12 个 bit：表示的序号，就是某个机房某台机器上这一毫秒内同时生成的 id 的序号，0000 00000000

- 1 bit：是不用的，为啥呢?

> **因为二进制里第一个 bit 为如果是 1，那么都是负数，但是我们生成的 id 都是正数，所以第一个 bit 统一都是 0**

- 41 bit：表示的是时间戳，单位是毫秒。

> **41 bit 可以表示的数字多达 2^41 - 1，也就是可以标识 2 ^ 41 - 1 个毫秒值，换算成年就是表示 69 年的时间。**

- 10 bit：记录工作机器id，代表的是这个服务最多可以部署在2^10台机器上，也就是1024台机器。

> **但是10 bit里5个bit代表机房id，5个bit代表机器id。意思就是最多代表2 ^ 5个机房（32个机房），每个机房里可以代表2 ^ 5个机器（32台机器）。**

- 12 bit：这个是用来记录同一个毫秒内产生的不同id。

> **12 bit可以代表的最大正整数是2 ^ 12 - 1 = 4096，也就是说可以用这个12bit代表的数字来区分同一个毫秒内的4096个不同的id**

简单来说，你的某个服务假设要生成一个全局唯一id，那么就可以发送一个请求给部署了snowflake算法的系统，由这个snowflake算法系统来生成唯一id。

这个snowflake算法系统首先肯定是知道自己所在的机房和机器的，比如机房id = 17，机器id = 12。

接着snowflake算法系统接收到这个请求之后，首先就会用二进制位运算的方式生成一个64 bit的long型id，64个bit中的第一个bit是无意义的。

接着41个bit，就可以用当前时间戳（单位到毫秒），然后接着5个bit设置上这个机房id，还有5个bit设置上机器id。

最后再判断一下，当前这台机房的这台机器上这一毫秒内，这是第几个请求，给这次生成id的请求累加一个序号，作为最后的12个bit。

最终一个64个bit的id就出来了，类似于：

```
0 | 0001100    10100010    10111110    10001001    01011100    00 | 10001 | 1    1001 | 0000    00000000
```

这个算法可以保证说，一个机房的一台机器上，在同一毫秒内，生成了一个唯一的 id。可能一个毫秒内会生成多个 id，但是有最后 12 个 bit 的序号来区分开来。

下面我们简单看看这个 snowflake 算法的一个代码实现，这就是个示例，大家如果理解了这个意思之后，以后可以自己尝试改造这个算法。

总之就是用一个 64bit 的数字中各个 bit 位来设置不同的标志位，区分每一个 id。

（5）snowflake 算法的代码实现

```php
public class IdWorker {

    private long workerId; // 这个就是代表了机器id
    private long datacenterId; // 这个就是代表了机房id
    private long sequence; // 这个就是代表了一毫秒内生成的多个id的最新序号

    public IdWorker(long workerId, long datacenterId, long sequence) {

        // sanity check for workerId
        // 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0
        if (workerId > maxWorkerId || workerId < 0) {

            throw new IllegalArgumentException(
                String.format("worker Id can't be greater than %d or less t
        }

        if (datacenterId > maxDatacenterId || datacenterId < 0) {

            throw new IllegalArgumentException(
                String.format("datacenter Id can't be greater than %d or le
        }

        this.workerId = workerId;
        this.datacenterId = datacenterId;
        this.sequence = sequence;
    }

    private long twepoch = 1288834974657L;

    private long workerIdBits = 5L;
    private long datacenterIdBits = 5L;
```

```
// 这个是二进制运算，就是5 bit最多只能有31个数字，也就是说机器id最多只能是32以内
private long maxWorkerId = -1L ^ (-1L << workerIdBits);

// 这个是一个意思，就是5 bit最多只能有31个数字，机房id最多只能是32以内
private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);
private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;
private long datacenterIdShift = sequenceBits + workerIdBits;
private long timestampLeftShift = sequenceBits + workerIdBits + datacen
private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId(){
    return workerId;
}

public long getDatacenterId() {
    return datacenterId;
}

public long getTimestamp() {
    return System.currentTimeMillis();
}

// 这个是核心方法，通过调用nextId()方法，让当前这台机器上的snowflake算法程序生成一
public synchronized long nextId() {

    // 这儿就是获取当前时间戳，单位是毫秒
    long timestamp = timeGen();

    if (timestamp < lastTimestamp) {
        System.err.printf(
            "clock is moving backwards. Rejecting requests until %d.",
        throw new RuntimeException(
            String.format("Clock moved backwards. Refusing to generate
                          lastTimestamp - timestamp));
    }


    // 下面是说假设在同一个毫秒内，又发送了一个请求生成一个id
    // 这个时候就得把seqence序号给递增1，最多就是4096
    if (lastTimestamp == timestamp) {
```

```
            // 这个意思是说一个毫秒内最多只能有4096个数字，无论你传递多少进来，
            //这个位运算保证始终就是在4096这个范围内，避免你自己传递个sequence超过了4
            sequence = (sequence + 1) & sequenceMask;

            if (sequence == 0) {
                timestamp = tilNextMillis(lastTimestamp);
            }

        } else {
            sequence = 0;
        }

        // 这儿记录一下最近一次生成id的时间戳，单位是毫秒
        lastTimestamp = timestamp;

        // 这儿就是最核心的二进制位运算操作，生成一个64bit的id
        // 先将当前时间戳左移，放到41 bit那儿；将机房id左移放到5 bit那儿；将机器id左移
        // 最后拼接起来成一个64 bit的二进制数字，转换成10进制就是个long型
        return ((timestamp - twepoch) << timestampLeftShift) |
                (datacenterId << datacenterIdShift) |
                (workerId << workerIdShift) | sequence;
    }

    private long tilNextMillis(long lastTimestamp) {

        long timestamp = timeGen();

        while (timestamp <= lastTimestamp) {
            timestamp = timeGen();
        }
        return timestamp;
    }

    private long timeGen(){
        return System.currentTimeMillis();
    }

    //---------------测试---------------
    public static void main(String[] args) {

        IdWorker worker = new IdWorker(1,1,1);

        for (int i = 0; i < 30; i++) {
```

```
                System.out.println(worker.nextId());
            }
        }
    }
```

(6）snowflake 算法一个小小的改进思路

其实在实际的开发中，这个 snowflake 算法可以做一点点改进。

因为大家可以考虑一下，我们在生成唯一 id 的时候，一般都需要指定一个表名，比如说订单表的唯一 id。

所以上面那 64 个 bit 中，代表机房的那 5 个 bit，可以使用业务表名称来替代，比如用 00001 代表的是订单表。

因为其实很多时候，机房并没有那么多，所以那 5 个 bit 用做机房 id 可能意义不是太大。

这样就可以做到，snowflake 算法系统的每一台机器，对一个业务表，在某一毫秒内，可以生成一个唯一的 id，一毫秒内生成很多 id，用最后 12 个 bit 来区分序号对待。

用小白都能看懂的大白话告诉你：什么是分布式计算系统！

> **作者:中华石杉** [原文地址]

目录

> ! 这篇文章聊一个话题：什么是分布式计算系统?

（1）从一个新闻门户网站案例引入

现在很多同学经常会看到一些名词，比如分布式服务框架，分布式系统，分布式存储系统，分布式消息系统。

但是有些经验尚浅的同学，可能都很容易被这些名词给搞晕。所以这篇文章就对 ***“分布式计算系统” **这个概念做一个科普类的分析。

如果你要理解啥是分布式计算，就必须先得理解啥是分布式存储，现在我们从一个小例子来引入。

比如说现在你有一个网站，咱们假设是一个新闻门户网站好了。每天是不是会有可能上千万用户会涌入进来看你的新闻？

好的，那么他们会怎么看新闻呢？

其实很简单，首先他们会点击一些板块，比如 “体育板块”，“娱乐板块”。

然后，点击一些新闻标题，比如 “20 年来最刺激的一场比赛即将拉开帷幕”，接着还可能会发表一些评论，或者点击对某个好的新闻进行收藏。

那么你的这些用户干的这些事儿有一个专业的名词，叫做 “用户行为”。

因为在你的网站或者 APP 上，用户一定会进行各种操作，点击各种按钮，发表一些信息，这些都是各种行为，统称为 “用户行为”。

好了，现在假如说新闻门户网站的 boss 说想要做一个功能，在网站里每天做一个排行榜，统计出来每天每个版块被点击的次数，包括最热门的一些新闻。

然后呢，在网站后台系统里需要有一些报表，要让他看到不同的编辑产出的文章的点击量汇总，做一个编辑的绩效排名，还有很多类似的事情。

这些事情叫什么呢？你可以认为是基于用户行为数据进行分析和统计，产出各种各样的数据统计分析报表和结果，供网站的用户、管理人员来查看。

这也有一个专业的名词，叫做 **“用户行为分析”。**

（2）推算一下你需要分析多少条数据？

好，咱么继续。如果你要对用户行为进行分析，那你是不是首先需要收集这些用户行为的数据？

比如说有个哥儿们现在点了一下 “体育” 板块，你需要在网页前端或者是 APP 上立马发送一条日志到后台，记录清楚“id 为 117 的用户点击了一下 id 位 003 的板块”。

同样，这个东西也有一个专业的名词，叫做 “用户行为日志”。

那你可以来计算一下，这些用户行为如果采用日志的方式收集，每天大概会产生多少条数据？

假设每天 1000 万人访问你的新闻网站，平均每个人做出 30 个点击、评论以及收藏等行为，那么就是 3 亿条用户行为日志。

假设每条用户行为日志的大小是 100 个字节，因为可能包含了很多很多的字段，比如他是在网页点击的，还是在手机 APP 上点击的，手机 APP 是用的什么操作系统，android 还是 IOS，类似这样的字段是很多的。

那么你就有每天大概 28GB 左右的数据，这里一共包含 3 亿条。

假如对这 3 亿条数据，你就自己写个 Java 程序，从一个超大的 28GB 的大日志文件里，一条一条读取日志来统计分析和计算，一直到把 3 亿条数据都计算完毕，你觉得会花费多少时间?

不可想象，根据你的计算逻辑复杂度来说，搞不好要花费几十个小时的时间。

所以你觉得这种大数据场景下的分析，这么玩儿靠谱么？不靠谱。

（3）黄金搭档：分布式存储 + 分布式计算

所以这个时候，你就可以首先采用分布式存储的方式，把那 3 亿条数据分散存放在比如 30 台机器上，每台机器大概就放 1000 万条数据，大概就 1GB 的数据量。

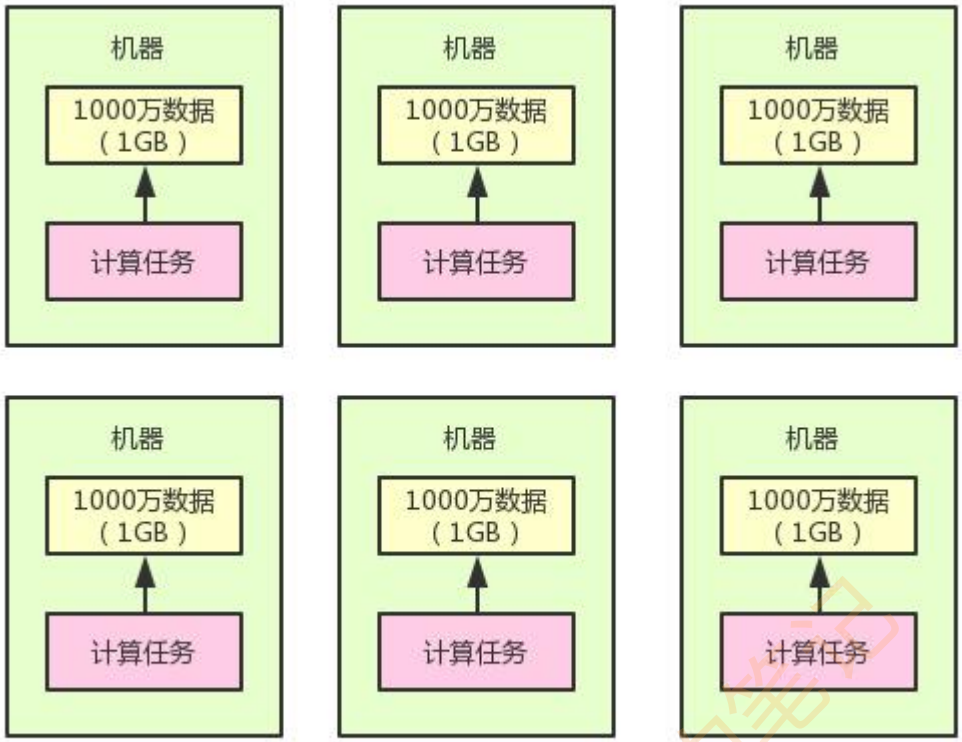
大家看看下面的图：

接着你就可以上分布式计算了，你可以把统计分析数据的计算任务，拆分成 30 个计算任务，每个计算任务都分发到一台机器上去运行。

也就是说，就专门针对机器本地的 1GB 数据，那 1000 万条数据进行分析和计算。

这样的好处就是可以依托 30 台机器的资源并行的进行数据的统计和分析，这也就是所谓的分布式计算了。

每台机器的计算结果出来之后，就可以进行综合性的汇总，然后就可以拿到最终的一个分析结果，大家看下图。



假设之前你的 3 亿条数据都在一个 30GB 的大文件里，然后你一个 Java 程序一条一条慢慢读慢慢计算，需要耗费 30 小时。

那么现在把计算任务并行到了 30 台机器上去，就可以提升 30 倍的计算速度，是不是就只需要 1 小时就可以完成计算了？

所以这个就是所谓的分布式计算，他一般是针对超大数据集，也就是现在很流行的大数据进行计算的。

首先需要将超大数据集拆分成很多数据块分散在多台机器上，然后把计算任务分发到各个机器上去，利用多台机器的 CPU、内存等计算资源来进行计算。

这种分布式计算的方式，对于超大数据集的计算可以提升几十倍甚至几百倍的效率，其实这个理论和概念，也是大数据技术的基础。

比如现在最流行的大数据技术栈里，Hadoop HDFS 就是用做分布式存储的，他可以把一个超大文件拆分为很多小的数据块放在很多机器上。

而像 Spark 就是分布式计算系统，他可以把计算任务分发到各个机器上，对各个数据块进行并行计算。

以上就是用大白话 + 画图，给小白同学们科普了一下分布式计算系统的相关知识，相信大家看了之后，对分布式计算系统，应该有一个初步的认识了。

Java同学找工作最懵圈的问题：到底啥是分布式系统开发经验?

> 作者:中华石杉 原文地址

目录

前言

现在有很多 Java 技术方向的同学在找工作的时候，肯定都会去招聘网站上找职位投递简历。

但是在很多职位 JD 上往往会有这样的一个要求：熟悉分布式系统理论、设计和开发，具备复杂分布式系统构建经验。

之前不少同学后台留言问过我：这个分布式系统的设计和开发经验，到底指的是什么？那么这篇文章就给大家来解释一下这个问题。

1 从单块系统说起

要说分布式系统是什么东西，那么就得先从单块系统开始说起。

很多同学应该都知道，如果你在一些中小型的传统软件公司里工作，那么很有可能现在在做的系统是如下这个样子。

所有的代码都在一个工程里，最多可能就是通过 maven 等构件工具拆分了一下代码工程模块，不同的模块可以放在不同的工程代码里。

在部署的时候，可能就是直接在线上的几台机器里直接放到里面的 tomcat 下来运行。

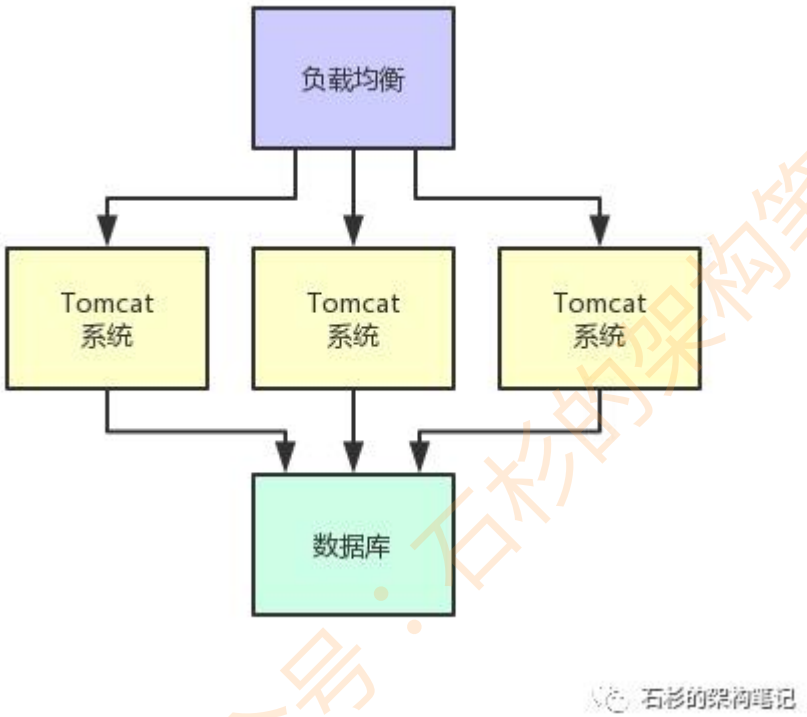
然后在 web 服务器前面可能会有一层负载均衡服务器，比如用 nginx 或者是其他的负载均衡设备。

很多流量很小的企业内部系统，比如 OA、CRM、财务等系统，甚至可能就直接在一台机器的 tomcat 下部署一下。

然后直接配置一下域名解析，就可以让这个系统的可能几十个，或者几百个用户通过访问域名来使用这个软件了。

至于说系统的依赖大概来说很可能只有一个，那就是 MySQL、Oracle 等关系型数据库，可能会在某台机器上专门部署一个数据库，让应用系统来使用。

大家看看下面的图，体会一下这种单体架构。



这种系统在很多中小型公司里现在还是比较多的，就是典型的单块系统，所有代码在一个工程，部署在一个 tomcat 里即可，这里包含了系统所有的功能。

你哪怕就部署一台机器，这个系统也可以运行，只不过为了所谓的 “高可用”，可能一般会部署两台机器，前面加一层负载均衡设备，这样其中一个机器挂了，另外一个机器上还有一个系统可以用。

2 团队越来越大，业务越来越复杂

其实上面说的那种单块系统，如果是一个 10 人以内的小团队大家一起维护和开发一个用户数量不多，请求量不大的系统，也是没问题的，还挺方便的，对吧。

你搞一个代码仓库，然后就一份代码，每个人都在自己本地写代码，最后把代码合并一下，做做测试，然后就直接部署基于 Tomcat 来就可以了。

但是问题就在于说，如果你的团队超过了 10 个人，比如有 20 个人，甚至几十个人，上百个人要一起协作开发这个系统，然后里面的业务逻辑特别多，可能功能模块多达几百个。这个时候就麻烦了，你要是还用那种单块系统的模式，那肯定是很痛苦的。

因为几十个人维护一个单块系统，大家在一个工程里写代码，大量的冲突以及代码合并都会让人崩溃。

而且部署的时候会有各种冲突，比如某个功能模块要上线了，但是他必须得把整个单块系统所有的功能都回归测试一遍才敢上线。

因为大家的代码都在一个工程里，都是耦合在一起的，你修改了代码，必须全部测试一遍才能保证系统正常。

所以说这个时候，就必须想办法把系统改造成分布式系统了。

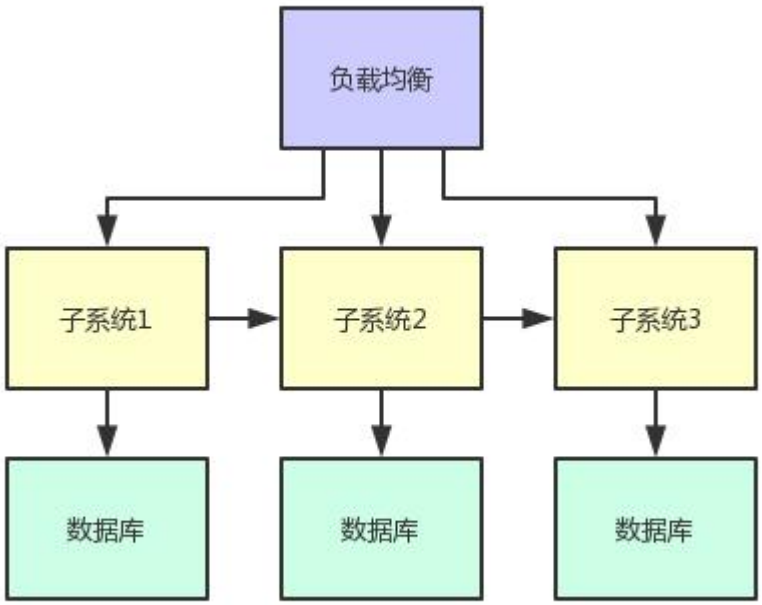
3 分布式出现：庞大系统分而治之

这个时候就可以尝试把一个大的系统拆分为很多小的系统，甚至很多小的服务，然后几个人组成一个小组就专门维护其中一个小系统，或者每个人维护一个小服务。

简单来说，就是分而治之，这样每个人可以专注维护自己的代码。

然后不同的小系统自己开发、测试和上线，都不会跟别人耦合在一起，可以自己独立进行，非常的方便，大大简化了大规模系统的开发成本。

不同的子系统之间，就是通过接口互相来回调用，每个子系统都有自己的数据库，大家看下面的图。



4 分布式系统所带来的技术问题

那么大家这个时候可以思考一下，如果你的公司是采用这种分布式系统的方式来构建公司的一个大规模系统的，那么这个时候会涉及到哪些技术问题？

（1）分布式服务框架

你如果要让不同的子系统或者服务之间互相通信，首先必须有一套分布式服务框架。

也就是各个服务可以互相感知到对方在哪里，可以发送请求过去，可以通过 HTTP 或者 RPC 的方式。

在这里，最常见的技术就是 dubbo 以及 spring cloud，当然大厂一般都是自己有服务框架

（2）分布式事务

一旦你的系统拆分为了多个子系统之后，那么一个贯穿全局的分布式事务应该怎么来实现？

这个你需要了解 TCC、最终一致性、2PC 等分布式事务的实现方案和开源技术。

（3）分布式锁

不同的系统之间如果需要在全局加锁获取某个资源的锁定，此时应该怎么来做？

毕竟大家不是在一个 JVM 里了，不可能用 synchronized 来在多个子系统之间实现锁吧，是不是？

（4）分布式缓存

如果你原来就是个单块系统，那么你其实是可以在单个 JVM 里进行本地缓存就可以了，比如搞一个 HashMap 来缓存一些数据。

但是现在你有很多个子系统，他们如果要共享一个缓存，你应该怎么办？是不是需要引入 Redis 等缓存系统？

（5）分布式消息系统

在单块系统内，就一个 JVM 进程内部，你可以用类似 LinkedList 之类的数据结构作为一个本地内存里的队列。

但是多个子系统之间要进行消息队列的传递呢？那是不是要引入类似 RabbitMQ 之类的分布式消息中间件？

（6）分布式搜索系统

如果在单块系统内，你可以比如在本地就基于 Lucene 来开发一个全文检索模块，但是如果是分布式系统下的很多子系统，你还能直接基于 Lucene 吗？

明显不行，你需要在系统里引入一个外部的分布式搜索系统，比如 Elasticsearch。

（7）其他很多的技术

比如说分布式配置中心、分布式日志中心、分布式监控告警中心、分布式会话，等等，都是分布式系统场景下你需要使用和了解的一些技术。

因为沿用单块系统时代的那些技术已经不行了，比如说你单块系统的时候，直接在本地用一个 properties 文件存放自己的配置即可，日志也写到本地即可。

但是分布式时代呢？

你那么多的子系统，怎么共享同一份配置？怎么把各个系统的日志聚合写到一个地方来查看？

单块系统的时候，你一个 web 应用直接基于 Servlet API 提供的 Session 会话功能即可，那么分布式时代呢，你有 N 多个子系统如果要共享会话该怎么做？

5 一句话总结：什么是分布式系统设计和开发经验？

其实分析完了之后，大家应该就大概知道了，招聘 JD 上写这个分布式系统的设计和开发经验，其实他是一个很大的主题，里面包含很多的内容。

你的系统一旦分布式了之后，通信、缓存、消息、事务、锁、配置、日志、监控、会话，等等各种原来单块系统场景下很容易解决的问题，都会变得很复杂，需要引入大量外部的技术。

所以你有没有参与过类似这样的一个大的分布式系统？你有没有基于各种技术解决过分布式系统场景下的各种技术问题？这就是人家希望和要求的分布式系统设计和开发的经验。如果大家还没接触过，建议多去学习一下。

6 补充说明：中间件系统及大数据系统

最后给大家说明一点，一般这种招聘 JD，如果是 Java 岗位要求分布式相关的经验，其实主要还是上面说的那些东西，他面向的是分布式的业务系统的构建。

但是其实分布式系统本身是一个非常复杂的话题，因为刚才说的只是一个分布式业务系统要依赖哪些技术来进行构建。

但是其实比如 Kafka、Rocket 等中间件，本身他也是分布式的，你要搞明白他们自己是如何实现分布式的，又是一个非常复杂的话题。

此外，像 hadoop、spark、hbase 等大数据系统，本身也都是世界上最最复杂的分布式系统，这又涉及到大数据领域的话题了，以后有机会可以单独聊聊。

哥们，你们的系统架构中为什么要引入消息中间件？

> **作者:中华石杉** 原文地址

这篇文章开始，我们把消息中间件这块高频的面试题给大家说一下，也会涵盖一些 MQ 中间件常见的技术问题。

假如面试官看你简历里写了 MQ 中间件的使用经验，很可能会有如下问题：

- 你们公司生产环境用的是什么消息中间件？
- 为什么要在系统里引入消息中间件？
- 引入消息中间件之后会有什么好处以及坏处？

好，我们一个个的来分析！

你们公司生产环境用的是什么消息中间件？

你们公司生产环境用的是什么消息中间件？ 这个首先你可以说下你们公司选用的是什么消息中间件，比如用的是 RabbitMQ，然后可以初步给一些你对不同 MQ 中间件技术的选型分析。

举个例子：比如说 ActiveMQ 是老牌的消息中间件，国内很多公司过去运用的还是非常广泛的，功能很强大。

但是问题在于没法确认 ActiveMQ 可以支撑互联网公司的高并发、高负载以及高吞吐的复杂场景，在国内互联网公司落地较少。而且使用较多的是一些传统企业，用 ActiveMQ 做异步调用和系统解耦。

然后你可以说说 RabbitMQ，他的好处在于可以支撑高并发、高吞吐、性能很高，同时有非常完善便捷的后台管理界面可以使用。

另外，他还支持集群化、高可用部署架构、消息高可靠支持，功能较为完善。

而且经过调研，国内各大互联网公司落地大规模 RabbitMQ 集群支撑自身业务的 case 较多，国内各种中小型互联网公司使用 RabbitMQ 的实践也比较多。

除此之外，RabbitMQ 的开源社区很活跃，较高频率的迭代版本，来修复发现的 bug 以及进行各种优化，因此综合考虑过后，公司采取了 RabbitMQ。

但是 RabbitMQ 也有一点缺陷，就是他自身是基于 erlang 语言开发的，所以导致较为难以分析里面的源码，也较难进行深层次的源码定制和改造，毕竟需要较为扎实的 erlang 语言功底才可以。

然后可以聊聊 RocketMQ，是阿里开源的，经过阿里的生产环境的超高并发、高吞吐的考验，性能卓越，同时还支持分布式事务等特殊场景。

而且 RocketMQ 是基于 Java 语言开发的，适合深入阅读源码，有需要可以站在源码层面解决线上生产问题，包括源码的二次开发和改造。

另外就是 Kafka。Kafka 提供的消息中间件的功能明显较少一些，相对上述几款 MQ 中间件要少很多。

但是 Kafka 的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数据计算等场景来设计。

因此 Kafka 在大数据领域中配合实时计算技术（比如 Spark Streaming、Storm、Flink）使用的较多。但是在传统的 MQ 中间件使用场景中较少采用。

PS：如果大家对上述一些 MQ 技术还没在自己电脑部署过，没写几个 helloworld 体验一下的话，建议先上各个技术的官网找到 helloworld demo，自己跑一遍玩玩

二 为什么在你们系统架构中要引入消息中间件?

回答这个问题，其实就是让你先说说消息中间件的常见使用场景。

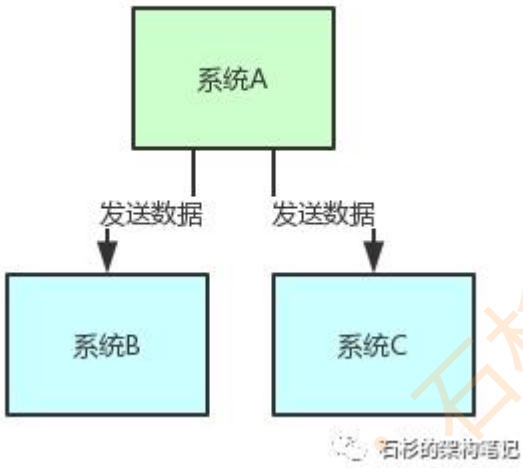
然后结合你们自身系统对应的使用场景，说一下在你们系统中引入消息中间件是解决了什么问题。

1）系统解耦

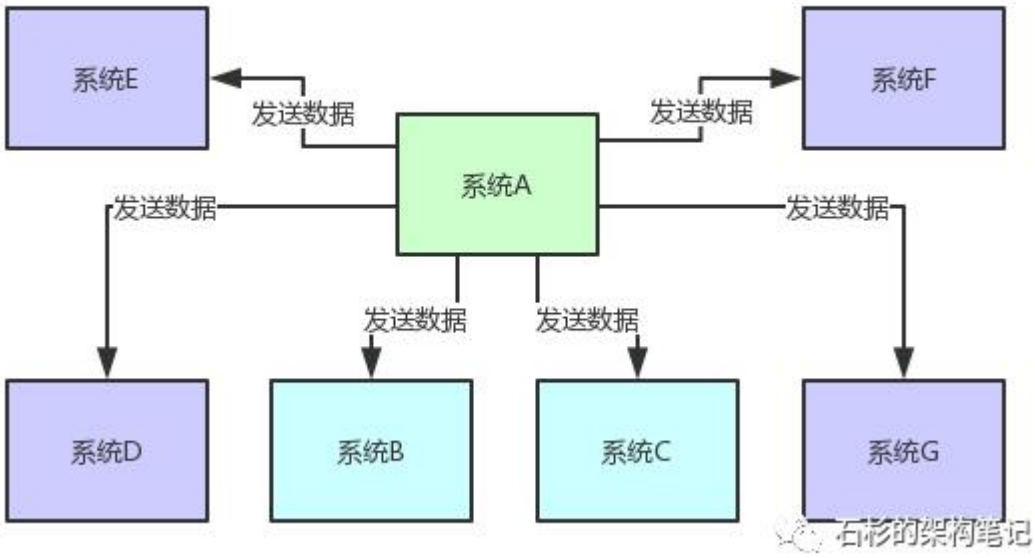
假设你有个系统 A，这个系统 A 会产出一个核心数据，现在下游有系统 B 和系统 C 需要这个数据。

那简单，系统 A 就是直接调用系统 B 和系统 C 的接口发送数据给他们就好了。

整个过程，如下图所示。



但是现在要是来了系统 D、系统 E、系统 F、系统 G，等等，十来个其他系统慢慢的都需要这份核心数据呢？如下图所示。



大家可别以为这是开玩笑，一个大规模系统，往往会拆分为几十个甚至上百个子系统，每个子系统又对应 N 多个服务，这些系统与系统之间有着错综复杂的关系网络。

如果某个系统产出一份核心数据，可能下游无数的其他系统都需要这份数据来实现各种业务逻辑。

此时如果你要是采取上面那种模式来设计系统架构，那么绝对你负责系统 A 的同学要被烦死了。

先是来一个人找他要求发送数据给一个新的系统 H，系统 A 的同学要修改代码然后在那个代码里加入调用新系统 H 的流程。

一会那个系统 B 是个陈旧老系统要下线了，告诉系统 A 的同学：别给我发送数据了，接着系统 A 再次修改代码不再给这个系统 B。

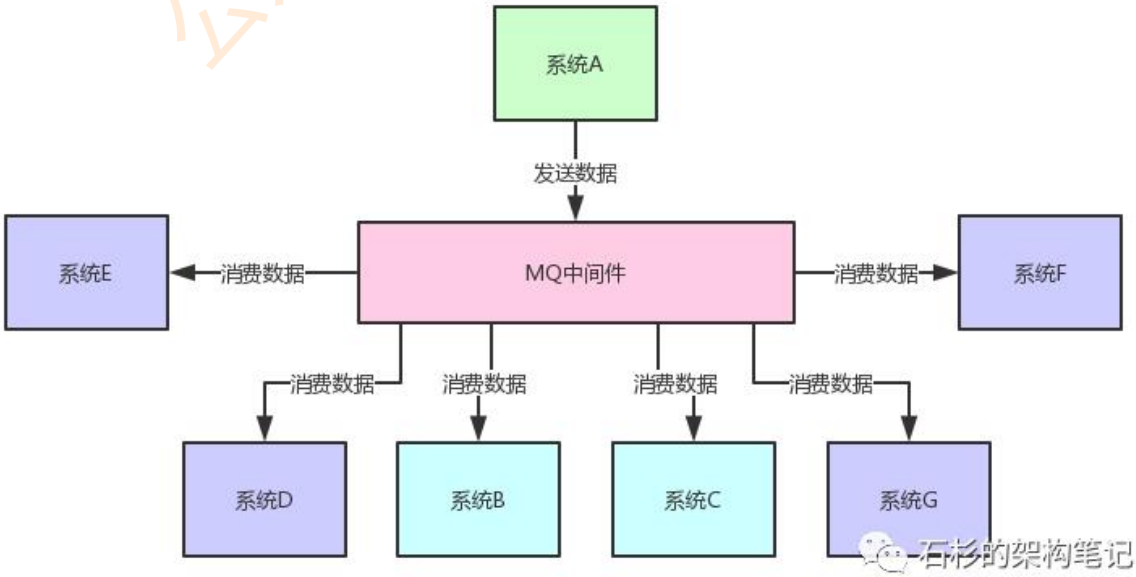
然后如果要是某个下游系统突然宕机了呢？系统 A 的调用代码里是不是会抛异常？那系统 A 的同学会收到报警说异常了，结果他还要去 care 是下游哪个系统宕机了。

所以在实际的系统架构设计中，如果全部采取这种系统耦合的方式，在某些场景下绝对是不合适的，系统耦合度太严重。

并且互相耦合起来并不是核心链路的调用，而是一些非核心的场景（比如上述的数据消费）导致了系统耦合，这样会严重的影响上下游系统的开发和维护效率。

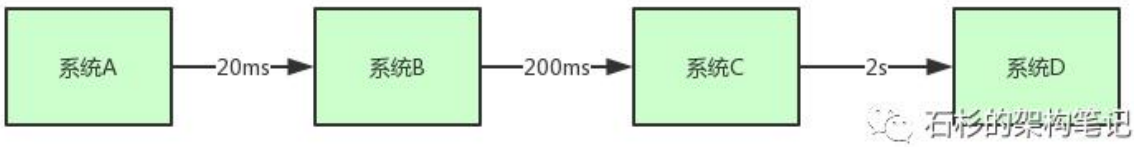
因此在上述系统架构中，就可以**采用 MQ 中间件来实现系统解耦。**

系统 A 就把自己的一份核心数据发到 MQ 里，下游哪个系统感兴趣自己去消费即可，不需要了就取消数据的消费，如下图所示。



2）异步调用

假设你有一个系统调用链路，是系统 A 调用系统 B，一般耗时 20ms；系统 B 调用系统 C，一般耗时 200ms；系统 C 调用系统 D，一般耗时 2s，如下图所示。



现在最大的问题就是：用户一个请求过来巨慢无比，因为走完一个链路，需要耗费 20ms + 200ms + 2000ms（2s） = 2220ms，也就是 2 秒多的时间。

但是实际上，链路中的系统 A 调用系统 B，系统 B 调用系统 C，这两个步骤起来也就 220ms。

就因为引入了系统 C 调用系统 D 这个步骤，导致最终链路执行时间是 2 秒多，直接将链路调用性能降低了 10 倍，这就是导致链路执行过慢的罪魁祸首。

那此时我们可以思考一下，是不是可以将系统 D 从链路中抽离出去做成异步调用呢？其实很多的业务场景是可以允许异步调用的。

举个例子，你平时点个外卖，咔嚓一下子下订单然后付款了，此时账户扣款、创建订单、通知商家给你准备菜品。

接着，是不是需要找个骑手给你送餐？那这个找骑手的过程，是需要一套复杂算法来实现调度的，比较耗时。

但是其实稍微晚个几十秒完成骑手的调度都是 ok 的，因为实际并不需要在你支付的一瞬间立马给你找好骑手，也没那个必要。

那么我们是不是就可以把找骑手给你送餐的这个步骤从链路中抽离出去，做成异步化的，哪怕延迟个几十秒，但是只要在一定时间范围内给你找到一个骑手去送餐就可以了。

这样是不是就可以让你下订单点外卖的速度变得超快？支付成功之后，直接创建好订单、账户扣款、通知商家立马给你准备做菜就 ok 了，这个过程可能就几百毫秒。

然后后台异步化的耗费可能几十秒通过调度算法给你找到一个骑手去送餐，但是这个步骤不影响我们快速下订单。

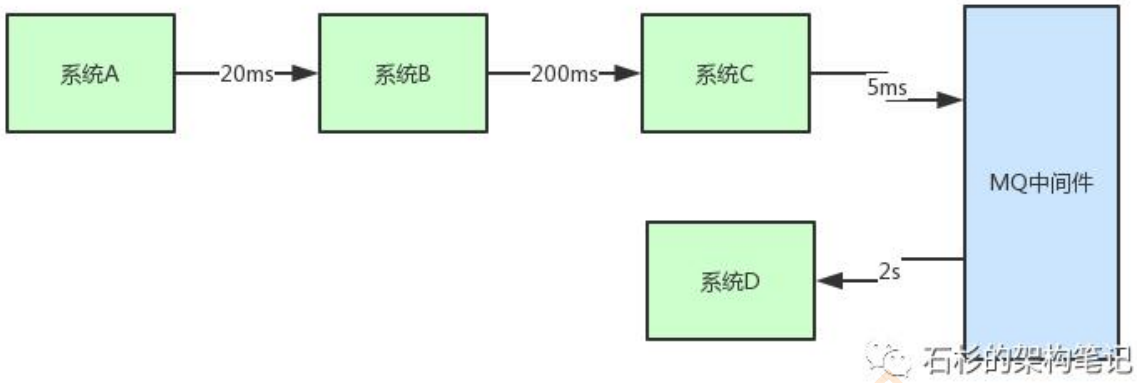
当然我们不是说那些大家熟悉的外卖平台的技术架构就一定是这么实现的，只不过是用一个生活中常见的例子给大家举例说明而已。

所以上面的链路也是同理，如果业务流程支持异步化的话，是不是就可以考虑把系统 C 对系统 D 的调用抽离出去做成异步化的，不要放在链路中同步依次调用。

这样，实现思路就是系统 A -> 系统 B -> 系统 C，直接就耗费 220ms 后直接成功了。

然后系统 C 就是发送个消息到 MQ 中间件里，由系统 D 消费到消息之后慢慢的异步来执行这个耗时 2s 的业务处理。通过这种方式直接将核心链路的执行性能提升了 10 倍。

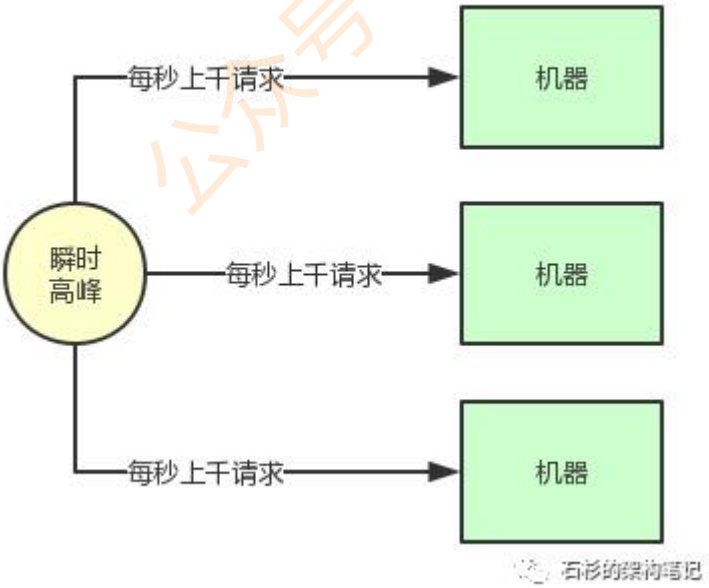
整个过程，如下图所示。



3）流量削峰

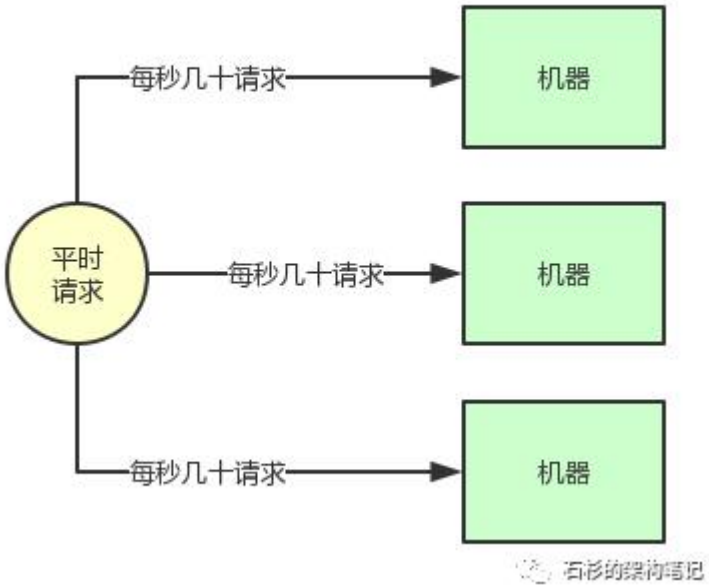
假设你有一个系统，平时正常的时候每秒可能就几百个请求，系统部署在 8 核 16G 的机器的上，正常处理都是 ok 的，每秒几百请求是可以轻松抗住的。

但是如下图所示，**在高峰期一下子来了每秒钟几千请求，瞬时出现了流量高峰，此时你的选择是要搞 10 台机器，抗住每秒几千请求的瞬时高峰吗？**



那如果瞬时高峰每天就那么半个小时，接着直接就降低为了每秒就几百请求，如果你线上部署了很多台机器，那么每台机器就处理每秒几十个请求就可以了，这不是有点浪费机器资源吗？

大部分时候，每秒几百请求，一台机器就足够了，但是为了抗那每天瞬时的高峰，硬是部署了 10 台机器，每天就那半个小时有用，别的时候都是浪费资源的。

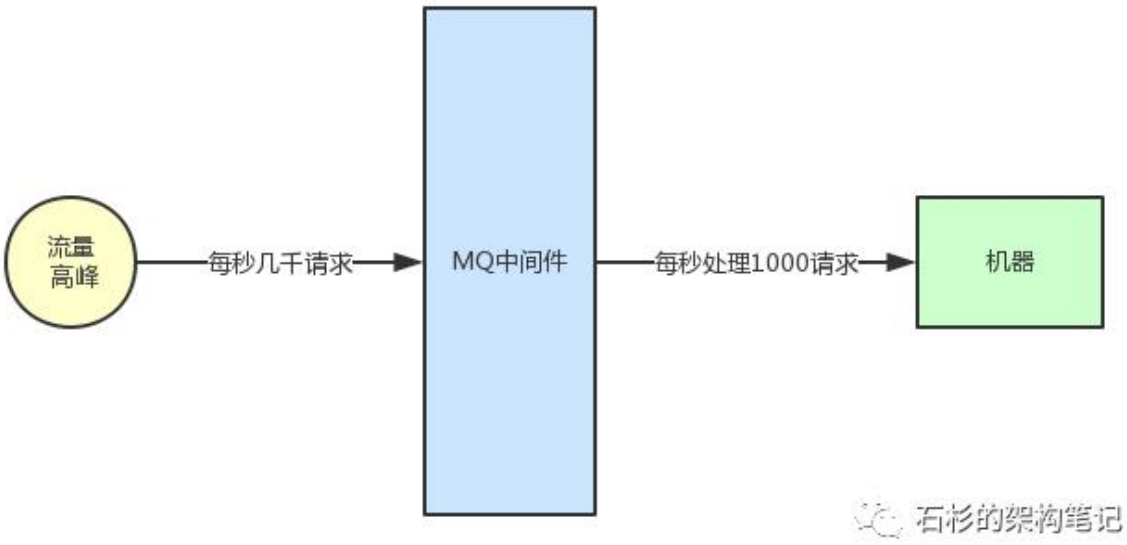


但是如果你就部署一台机器，那会导致瞬时高峰时，一下子压垮你的系统，因为绝对无法抗住每秒几千的请求高峰。

此时我们就可以用 MQ 中间件来进行流量削峰。所有机器前面部署一层 MQ，平时每秒几百请求大家都可以轻松接收消息。

一旦到了瞬时高峰期，一下涌入每秒几千的请求，就可以积压在 MQ 里面，然后那一台机器慢慢的处理和消费。

等高峰期过了，再消费一段时间，MQ 里积压的数据就消费完毕了。



这个就是很典型的一个 MQ 的用法，用有限的机器资源承载高并发请求，如果业务场景允许异步削峰，高峰期积压一些请求在 MQ 里，然后高峰期过了，后台系统在一定时间内消费完毕不再积压的话，那就很适合用这种技术方案。

下集预告：

关于第三个问题：引入消息中间件之后会有什么好处以及坏处。

我们将会在下一篇文章：**《兄弟，那你说说系统架构引入消息中间件有什么缺点呢？》** 详细阐述，敬请关注。

哥们，那你说说系统架构引入消息中间件有什么缺点?

> **作者:中华石杉** 原文地址

一 前情回顾

上篇文章**哥们，你们的系统架构中为什么要引入消息中间件？**，给大家讲了讲消息中间件引入系统架构的作用，主要是解决哪些问题的。

其比较常见的实践场景是：

- 复杂系统的解耦
- 复杂链路的异步调用
- 瞬时高峰的削峰处理

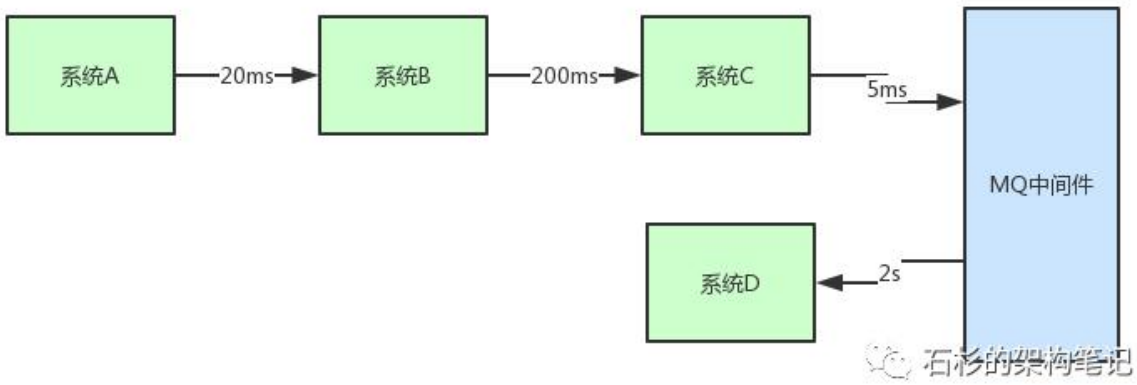
二 正式开始

> ! 这篇文章给大家讲讲，如果你在系统架构里引入了消息中间件之后，会有哪些缺点?

1 系统可用性降低

首先是你的系统整体可用性绝对会降低，给你举个例子，我们就拿之前的一幅图来说明。

比如说一个核心链路里面，系统 A -> 系统 B -> 系统 C，然后系统 C 是通过 MQ 异步调用系统 D 的。



看起来很好，你用这个 MQ 异步化的手段解决了一个核心链路执行性能过差的问题。

但是你有没有考虑另外一个问题，就是万一你依赖的那个 MQ 中间件突然挂掉了怎么办？这个还真的不是异想天开，MQ、Redis、MySQL 这些组件都有可能会挂掉。

一旦你的 MQ 挂了，就导致你的系统的核心业务流程中断了。本来你要是不引入 MQ 中间件，那其实就是一些系统之间的调用，但是现在你引入了 MQ，就导致你多了一个依赖。一旦多了一个依赖，就会导致你的可用性降低。

因此，一旦引入了 MQ 中间件，你就必须去考虑这个 MQ 是如何部署的，如何保证高可用性。

甚至在复杂的高可用的场景下，你还要考虑如果 MQ 一旦挂了以后，你的系统有没有备用兜底的技术方案，可以保证系统继续运行下去。

之前写过两篇文章，都涉及到了** MQ 挂掉之后的高可用保障方案**。

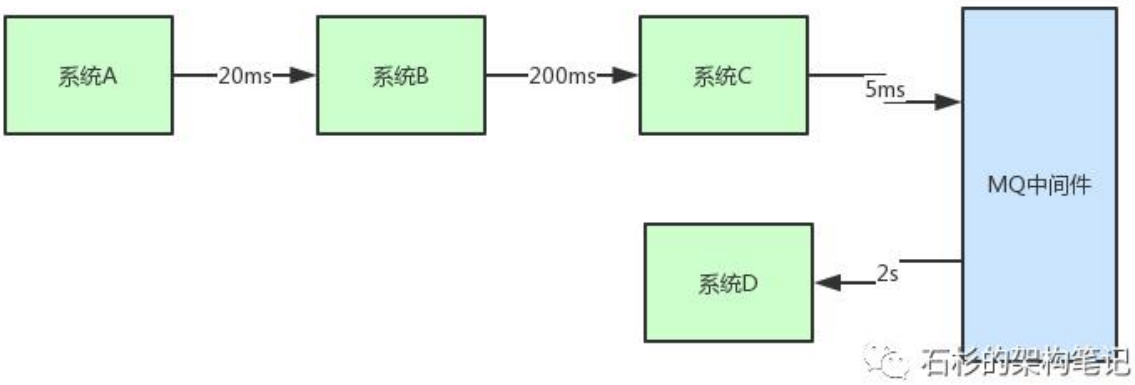
大伙如果感兴趣，可以参考一下：

- **《最终一致性分布式事务的 99.99% 高可用保障生产实践》**
- **《亿级流量系统架构之如何设计全链路 99.99% 高可用架构》**

通过这两篇文章，具体看看我们在各种场景下遇到 MQ 故障采取的高可用降级方案。

2 系统稳定性降低

还是上面那张图，大家再来看一下。



不知道大家有没有发现一个问题，这个链路除了 MQ 中间件挂掉这个可能存在的隐患之外，可能还有一些其他的技术问题。

比如说，莫名其妙的，系统 C 发了一个消息到 MQ，结果那个消息因为网络故障等问题，就丢失了。这就导致系统 D 没有收到那条消息。

这可就惨了，这样会导致系统 D 没完成自己该做的任务，此时可能整个系统会出现业务错乱，数据丢失，严重的 bug，用户体验很差等各种问题。

这还只是其中之一，万一说系统 C 给 MQ 发送消息，不小心一抽风重复发了一条一模一样的，导致消息重复了，这个时候该怎么办?

可能会导致系统 D 一下子把一条数据插入了两次，导致数据错误，**脏数据的产生**，最后一样会导致各种问题。

或者说如果系统 D 突然宕机了几个小时，导致无法消费消息，结果大量的消息在 MQ 中间件里积压了很久，这个时候怎么办?

即使系统 D 恢复了，也需要慢慢的消费数据来进行处理。

所以这就是引入 MQ 中间件的第二个大问题，系统稳定性可能会下降，故障会增多，各种各样乱七八糟的问题都可能产生。

而且一旦产生了一个问题，就会导致系统整体出问题。就需要为了解决各种 MQ 引发的技术问题，采取很多的技术方案。

关于这个，我们后面会用专门的文章聊聊 MQ 中间件的这些问题的解决方案，包括：

- 消息高可靠传递（0 丢失）
- 消息幂等性传递（绝对不重复）
- 百万消息积压的线上故障处理

3 分布式一致性问题

引入消息中间件，还有分布式一致性的问题。

举个例子，比如说系统 C 现在处理自己本地数据库成功了，然后发送了一个消息给 MQ，系统 D 也确实是消费到了。

但是结果不幸的是，系统 D 操作自己本地数据库失败了，那这个时候咋办？

系统 C 成功了，系统 D 失败了，会导致系统整体数据不一致了啊。

所以此时又需要使用可靠消息最终一致性的分布式事务方案来保障。

关于这个，可以参考之前的一篇文章：

《最终一致性分布式事务的 99.99% 高可用保障生产实践》

我们在里面详细阐述了系统之间异步调用场景下，如何采用分布式事务方案保证其数据一致性。

三 总结

最后，我们来做一个简单的小结。

在面试中要答好这个问题，首先一定要熟悉 MQ 这个技术的优缺点。了解清楚把他引入系统是为了解决哪些问题的，但是他自身又会带来哪些问题。

此外，对于引入 MQ 以后，是否对他自身可能引发的问题有一些方案的设计，来保证你的系统高可用、高可靠的运行，保证数据的一致性。这个也有做好相应的准备。

哥们，消息中间件在你们项目里是如何落地的？

> **作者:中华石杉** 原文地址

一、前情回顾

之前给大家聊了一下，面试时如果遇到消息中间件这个话题，面试官上来可能问的两个问题：

- 你们的系统架构中为什么要引入消息中间件？
- 系统架构中引入消息中间件有什么缺点？

关于这两个问题的回答，可以参见之前的两篇文章：

- **哥们，你们的系统架构中为什么要引入消息中间件?**
- **哥们，那你说说系统架构引入消息中间件有什么缺点?**

在问完这两个问题之后，不同风格的面试官可能会展开不同的发问。

针对那种工作年限比较长的资深的同学，可能会开始就候选人所在公司使用的消息中间件，深入里面的技术细节，比如让你聊聊 RocketMQ 的架构原理和核心源码?

但是另外一种面试风格，会先从你们的项目和业务入手进行考察，比如像下面这样：

- 消息中间件在你们生产项目里具体是哪个业务场景下落地的?
- 这个业务场景有什么技术挑战?
- 为什么必须要在这个业务场景里用消息中间件技术?
- 具体使用消息中间件的时候是怎么来用的?

好！这篇文章，咱们从第二种风格来聊聊。

二、业务场景介绍

我们会落地到某个具体业务系统的某个场景下，看看如何使用消息中间件，然后其效果是什么。

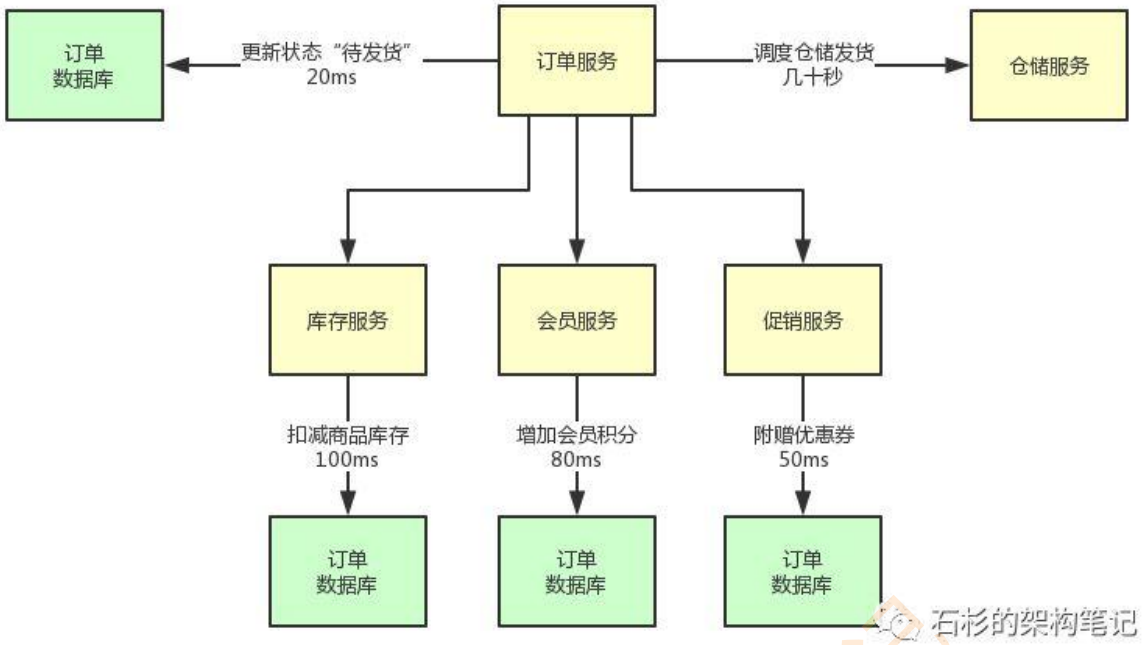
业务场景的话，咱们就用大家都很熟悉的电商业务为例，这里为了便于理解，对其做了一定的抽象和简化。

大家还是来考虑一个下订单的业务流程，比如你下个订单，此时需要干几件事情：

- 更新订单状态为 “待发货”（耗时 20ms）
- 扣减商品库存（耗时 100ms）
- 增加会员积分（耗时 80ms）
- 附赠优惠券（耗时 50ms）
- 仓储调度发货（耗时几十秒）

说明一下：上述环节，为了便于大家理解，做了简化。实际真正复杂的电商系统里，整体环节和业务流程会比这个复杂很多倍，而且耗时也绝对不是上面那么简单的。

老规矩！我们还是通过一张手绘图，来看看这整个的业务流程：



如上图，这个下订单的业务流程中：

更新订单状态（20ms） + 扣减商品库存（100ms） + 增加会员积分（80ms） + 附赠优惠券（50ms） = 250ms。

也就是说，仅仅是这 4 个流程的话，也就 200 多毫秒的耗时。

200 多毫秒的耗时，对用户下单体验来说是非常快速的，几乎就是一瞬间就完成了，不会感到过多的停顿，也就是一下子就可以看到自己下单成功了。

但是，如果加上那个调度仓储发货呢?

那个环节需要读取大量的数据、使用多仓库 / 多货位的调度算法、还要跟 C/S 架构的仓储系统进行网络通信，因此我们这里假设这个环节可能会耗时数十秒。

一旦加上那个调度仓储发货的环节到这个下单流程里，就可能导致用户要等页面卡顿几十秒后才会看到下单成功的提示，这个用户体验就相当的差了。

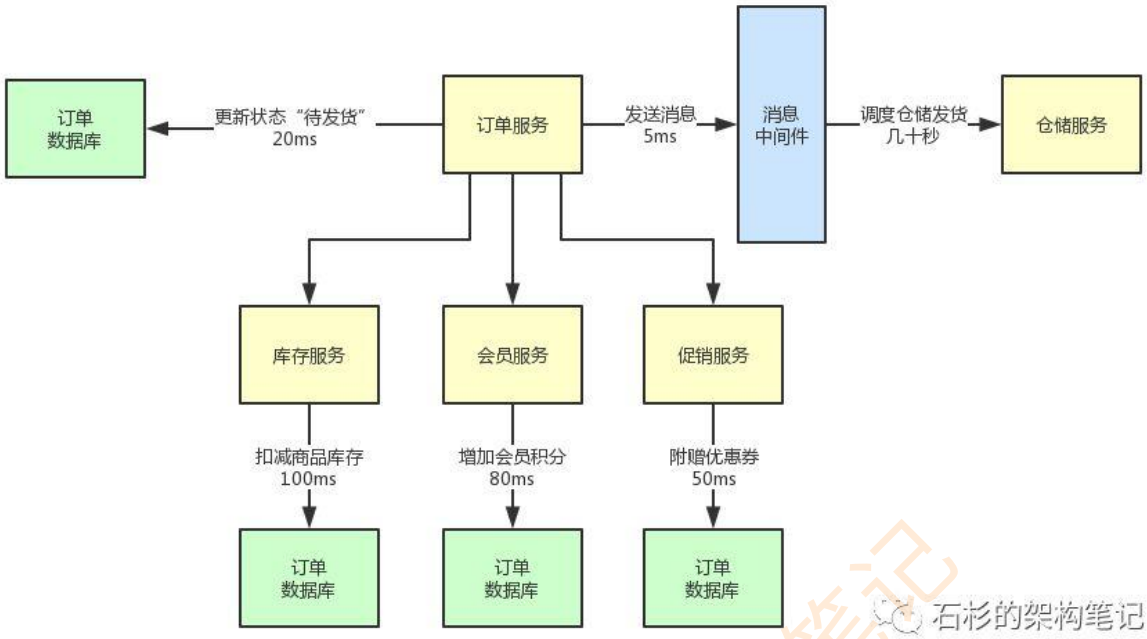
按照之前一篇文章**《哥们，你们的系统架构中为什么要引入消息中间件？》**的说法。对于这种场景，完全适合使用消息中间件来进行异步化调用。

也就是说，订单服务对仓储调度发货，仅仅是发送一个消息到 MQ 里，然后仓储服务消费消息之后再慢慢的执行调度算法，然后分配商品发货任务给对应的仓库即可。

这样的话，就可以把耗时几十秒的仓储调度发货的环节，从下单流程里摘除出去了。进而保证下单流程就仅仅是耗时 200 多毫秒而已。

至于那个耗时几十秒的仓储调度发货环节，我们通过异步的方式慢慢执行即可，不会影响用户下单的体验。

以上过程，我们同样来一张图，大家直观的感受一下：



三、初步落地

好！接下来我们就假设大家在实际生产中还没用过消息中间件，咱们从 0 开始，看看如何落地？

对于已经在生产中使用过消息中间件的小伙伴，不妨也看看，权当复习，温故知新！

我们以 RabbitMQ 为例，假如你用的消息中间件是 RabbitMQ，那么我们对这个消息中间件应该如何安装和部署呢？

很简单，RabbitMQ 的官方文档里提供了非常详细的安装部署步骤，你可以在自己的笔记本电脑本地安装，也可以在公司的服务器上部署。

现在假设你已经参考了官方文档并安装完成，那么接下来在代码层面应该怎么来引入 RabbitMQ 以及在系统里实现收发消息呢？

下面通过一些 HelloWorld 级别的代码和一些简单的示例图，给大家演示一下 RabbitMQ 是如何收发消息的。

对于很多在实际生产中使用过 MQ 的同学，这些代码可能对实际生产中使用过 MQ 的同学，显得太简单了。

不过考虑到很多初学者可能连用都没有用过 MQ，甚至是才听说消息中间件不久，所以笔者认为这些 demo 代码以及手工绘图，还是很有必要。

```
// 下面的代码是存在于订单服务中

// 订单服务需要通过这段代码发送一个消息到RabbitMQ的queue中，
// 然后通知仓储服务调度发货

// 定义一个RabbitMQ中的queue的名称
// 这个队列名字，我们的中文含义是：仓储调度发货
// 所以代表这个队列中收发的消息都是关于仓储调度发货的，比如一个订单需要进行发货
String QUEUE_NAME = "warehouse_schedule_delivery";

// 这个ConnectionFactory就是抽象了底层的Socket网络连接，对我们来说更加简便易用
ConnectionFactory factory = new ConnectionFactory();

// 这个地方设置一下你要连接的rabbitmq部署所在的机器即可
factory.setHost("你的rabbitmq部署的机器");
```



```
// 这个地方设置一下你要连接的rabbitmq部署所在的机器即可
factory.setHost("你的rabbitmq部署的机器");

// 这儿用try语法包裹一下开启Connection连接，以及打开Channel通道
// 这样就不用自己在代码里关闭这些重量级的通信资源了，java语法会自动保证关闭他们
try (
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
) {

    // 这个定义了一个RabbitMQ中的queue，如果queue不存在就自动给你创建一个
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);

    // 这个发送到RabbitMQ的queue里的，可以是你的订单信息
    // 这样仓储服务从RabbitMQ的queue里消费的这个订单消息之后，就可以基于仓储数据调度发货了
    String message = "订单信息";

    // 这个就是使用RabbitMQ提供的Channel API，推送一条消息到指定的RabbitMQ的queue里去
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
    System.out.println(" [x] 订单服务发送消息 '" + message + "'");
}
```

```
/**
只要你能把一个单机版本的RabbitMQ安装好，那么上面的代码就可以让你推送一条
订单消息到指定的“仓储 调度发货”的queue里去。接着我们再给大家用一段HelloWorld代码
演示一下仓储服务是如何从RabbitMQ中消费数据的呢？**/

// 以下代码你可以放在仓储服务里来运行，他会不停的从RabbitMQ的queue里来消费仓储调度发货的消息
String QUEUE_NAME = "warehouse_schedule_delivery";

// 下面4行代码，跟上面的意思是一样的
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

// 这儿也尝试创建一下queue
// 如果queue不存在的话就创建一下，避免仓储服务消费出现问题
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
System.out.println(" [*] 仓储服务开始准备消息仓储调度发货的消息......");

// 这个本质其实就是定义一个回调接口
// 只要从RabbitMQ消费到一条订单消息，就会调用这个里面的代码逻辑
// 在这个里面就可以处理订单消息
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    // 这个里面，我们对消费到的订单消息，其实就是简单的打印一下罢了
    // 不然按照真实的业务流程，应该是要对订单进行调度发货的
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] 仓储服务接收到消息，准备执行调度发货的流程 '" + message + "'");
};
```

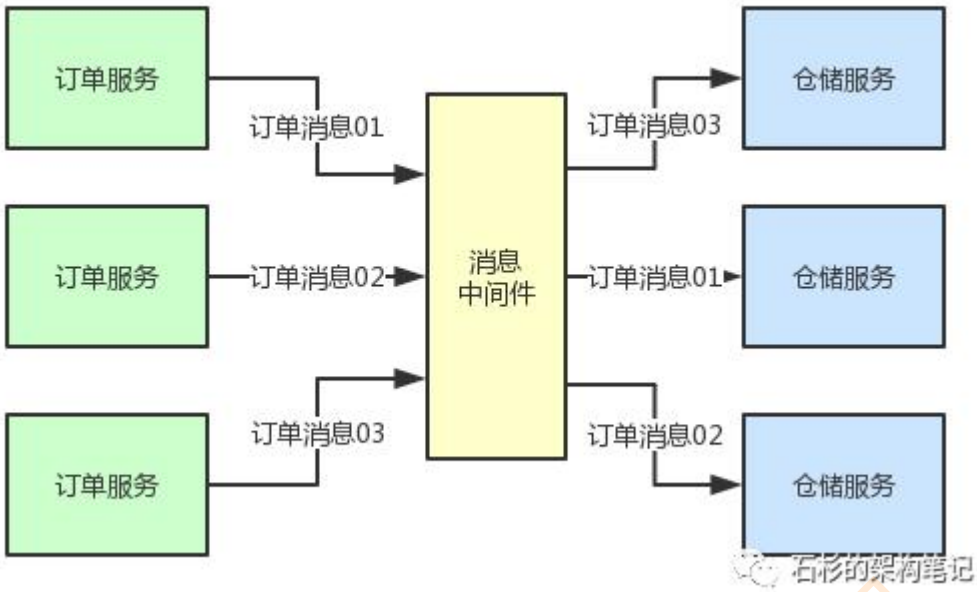


```
// 这个意思就是说，对“warehouse_schedule_delivery”这个queue进行消息监听
// 如果那个queue里有消息，就消费出来
// 然后消费出来的消息给谁处理呢？当然是这里指定的deliveryCallback回调接口了！
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
```


好！看完了代码，这个时候，我们可以通过一张图来想象一下两个服务之间的通信。

订单服务你可以启动多个，不同的订单服务都可以往一个 RabbitMQ 的 queue 里推送消息。

仓储服务你也可以启动多个，多个仓储服务会采用 round-robin 的轮询算法，每个服务实例都可以从 RabbitMQ queue 里消费到一部分的消息。



上面的图里，订单服务在 MQ 专业术语中叫做 **“生产者”**，英文是 “Producer”，意思就是这个服务是专门负责生产消息投递到 MQ 的。

仓储服务在 MQ 专业术语中叫做 **“消费者”**，英文是 “Consumer”，意思就是这个服务专门是负责从 MQ 消费消息然后处理的。

这个时候，这套异步通信的架构就可以跑起来了。

好了，到目前为止，虽然这个代码还存在不少问题，但是没关系，大体上我们已经给一些不太熟悉 MQ 技术的同学，从一个比较形象易于理解简化后的电商业务场景出发，通过 HelloWorld 级别的示例代码和手工绘图，将 MQ 这个技术落地跑起来了。

更进一步，各位同学完全可以参照这个文章里的案例，思考一下：自己负责的项目里，有没有类似的业务场景可以使用 MQ 的?

然后想办法在自己的项目里落地使用 MQ 的技术来做一下异步化，提升核心流程的性能。

这样未来在跳槽面试的时候，才可以做到游刃有余，有自己的一套东西可以说

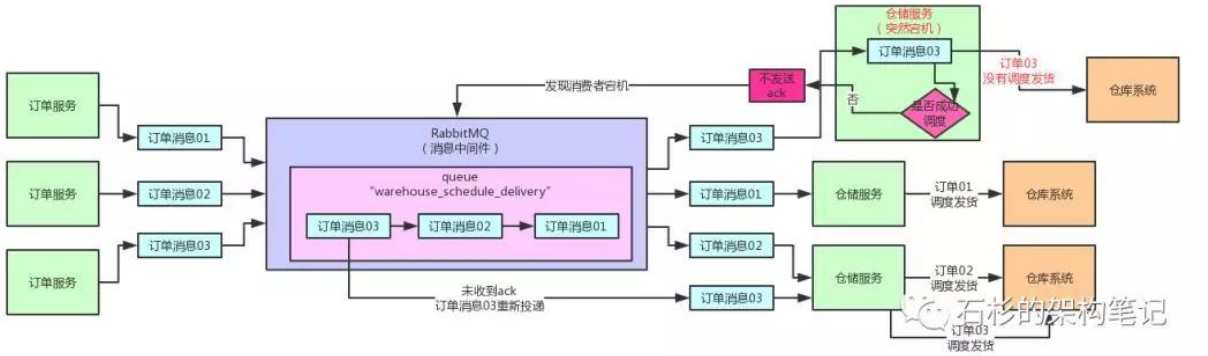
消息中间件集群崩溃，如何保证百万生产数据不丢失?

> **作者:中华石杉** 原文地址

> ! “上一篇讲消息中间件的文章**扎心！线上服务宕机时，如何保证数据100%不丢失?**，初步给大家介绍了一个在生产环境中可能遇到的问题，就是你的消费者服务可能会宕机，一旦宕机，你就需要考虑是否会导致没处理完的消息丢失。

这篇文章，再给不太熟悉 MQ 技术的同学，介绍另外一个生产环境中可能会遇到的问题。

目前为止，你的 RabbitMQ 部署在线上服务器了，对吧？然后订单服务和仓储服务都可以基于 RabbitMQ 来收发消息，同时仓储服务宕机，不会导致消息丢失。



好，我们来看下目前为止的架构图。

那如果此时出现一个问题，就是说订单服务投递了订单消息到RabbitMQ里去，RabbitMQ暂时放在了自己的内存中，还没来得及投递给下游的仓储服务呢，此时RabbitMQ突然宕机了，会怎么样？

答案其实很简单，默认情况下，按照我们目前的代码和配置，这个数据就会丢失了。

所以在这里而言，就牵扯到了RabbitMQ的一个较为重要的概念：消息的持久化，用英文来说就是durable机制。

然后这里又有一个引申的概念，如果按照我们之前的代码和配置，默认情况下，RabbitMQ一旦宕机就再次重启，就会丢失我们之前创建的queue。所以首先得先让queue是持久化的。

使用下面的代码，就可以把我们的“warehouse_schedule_delivery”这个queue，也就是仓储调度发货的queue，设置为持久化的。

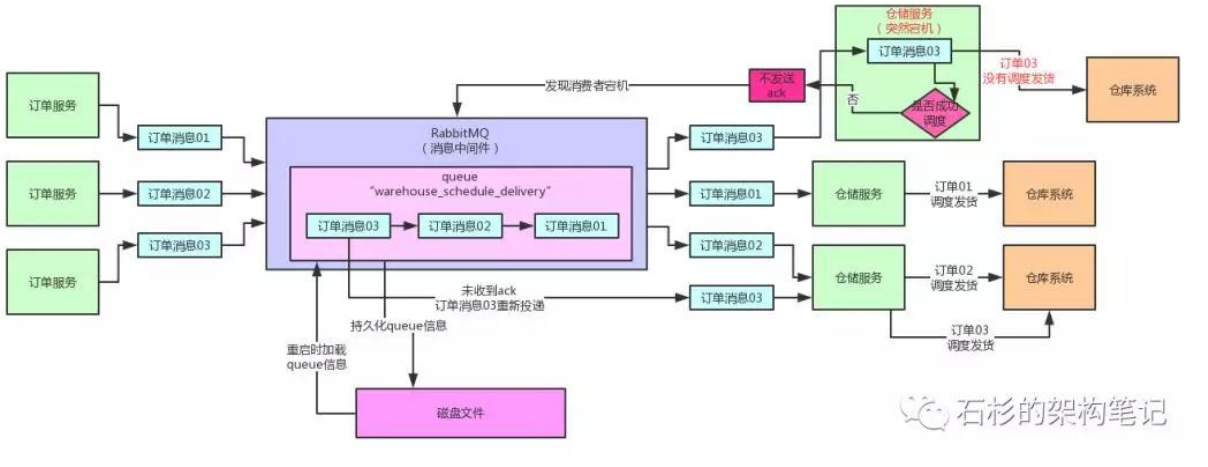
这样，即使RabbitMQ宕机后重启，也会恢复之前创建好的这个queue。

```
channel.queueDeclare("warehouse_schedule_delivery", true, false, false, null);
```

大家看到上面那行定义和创建queue的代码么？核心在于第二个参数，第二个参数是true。

他的意思就是说，这个创建的queue是durable的，也就是支持持久化的。

RabbitMQ会把这queue的相关信息持久化的存储到磁盘上去，这样RabbitMQ重启后，就可以恢复持久化的queue。



OK，现在你的queue的信息可以持久化了，RabbitMQ宕机重启后会自动恢复queue。但是，你的queue里的message数据呢?

queue里都是订单服务发送过去的订单消息数据，如果RabbitMQ还没来得及投递queue里的订单消息到仓储服务，结果RabbitMQ就宕机了。

那此时RabbitMQ重启之后，他可以恢复queue的信息，但是queue的message数据是没法恢复了。

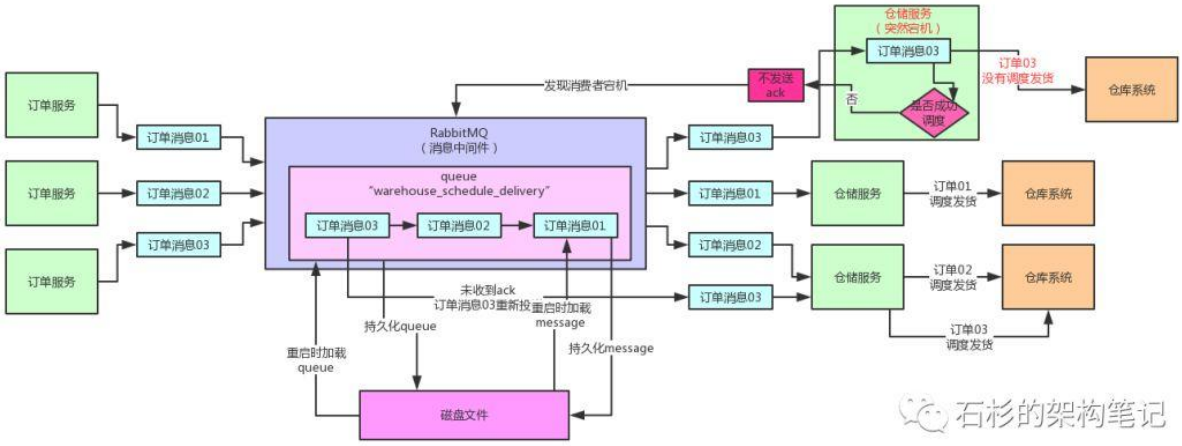
所以此时还有一个重要的点，就是在你的订单服务发送消息到RabbitMQ的时候，需要定义这条消息也是durable，即持久化的。

```
channel.basicPublish("",  "warehouse_schedule_delivery",
MessageProperties.PERSISTENT_TEXT_PLAIN,  message.getBytes());
```

通过上面的方式来发送消息，就可以让发送出去的消息是持久化的。

一旦标记了消息是持久化之后，就会让RabbitMQ把消息持久化写入到磁盘上去，此时如果RabbitMQ还没投递数据到仓储服务，结果就突然宕机了。那么再次重启的时候，就会把磁盘上持久化的消息给加载出来。

整个过程，如下图所示：



但是这里要注意一点，**RabbitMQ的消息持久化，是不承诺100%的消息不丢失的。**

因为有可能RabbitMQ接收到了消息，但是还没来得及持久化到磁盘，他自己就宕机了，这个时候消息还是会丢失的。

如果要完全100%保证写入RabbitMQ的数据必须落地磁盘，不会丢失，需要依靠其他的机制。

下次有机会再继续给不太熟悉MQ技术的同学，来讲解这里的东西。

如何保证消息中间件全链路数据100%不丢失（1）

> **作者:中华石杉** 原文地址

1 背景引入

这篇文章，我们来聊聊在线上生产环境使用消息中间件技术的时候，从前到后的全链路到底如何保证数据不能丢失。

这个问题，在互联网公司面试的时候高频出现，而且也是非常现实的生产环境问题。

如果你的简历中写了自己熟悉 MQ 技术（RabbitMQ、RocketMQ、Kafka），而且在项目里有使用的经验，那么非常实际的一个生产环境问题就是：投递消息到 MQ，然后从 MQ 消费消息来处理的这个过程，数据到底会不会丢失。

面试官此时会问：如果数据会丢失的话，你们项目生产部署的时候，是通过什么手段保证基于 MQ 传输的数据 100% 不会丢失的？麻烦结合你们线上使用的消息中间件来具体说说你们的技术方案。

这个其实就是非常区分面试候选人技术水平的一个问题。

实际上相当大比例的普通工程师，哪怕是在一些中小型互联网公司里工作过的，也就是基于公司部署的 MQ 集群简单的使用一下罢了，可能代码层面就是基本的发送消息和消费消息，基本没考虑太多的技术方案。

但是实际上，对于 MQ、缓存、分库分表、NoSQL 等各式各类的技术以及中间件在使用的时候，都会有对应技术相关的一堆生产环境问题。

那么针对这些问题，就必须要有相对应的一整套技术方案来保证系统的健壮性、稳定性以及高可用性。

所以其实中大型互联网公司的面试官在面试候选人的时候，如果考察对 MQ 相关技术的经验和掌握程度，十有八九都会抛出这个使用 MQ 时一定会涉及的数据丢失问题。因为这个问题，能够非常好的区分候选人的技术水平。

所以这篇文章，我们就来具体聊聊基于 RabbitMQ 这种消息中间件的背景下，从投递消息到 MQ，到从 MQ 消费消息出来，这个过程中有哪些数据丢失的风险和可能。

然后我们再一起来看看，应该如何结合 MQ 自身提供的一些技术特性来保证数据不丢失?

2 前情回顾

首先给大伙一点提醒，有些新同学可能还对 MQ 相关技术不太了解，建议看一下之前的 MQ 系列文章，看看 MQ 的基本使用和原理：

- **《哥们，你们的系统架构中为什么要引入消息中间件？》**
- **《哥们，那你说说系统架构引入消息中间件有什么缺点？》**
- **哥们，消息中间件在你们项目里是如何落地的?**

另外，其实之前我们有过 2 篇文章是讨论消息中间件的数据不丢失问题的。

我们分别从消费者突然宕机可能导致数据丢失，以及集群突然崩溃可能导致的数据丢失两个角度讨论了一下数据如何不丢失。

只不过仅仅那两个方案还无法保证全链路数据不丢失，但是大家如果没看过的建议也先回过头看看：

-**《扎心！线上服务宕机时，如何保证数据 100% 不丢失？》**

-**《消息中间件集群崩溃，如何保证百万生产数据不丢失？》**

总之，希望对 MQ 不太熟悉的同学，先把前面那些系列文章熟悉一下，然后再来一起系统性的研究一下 MQ 数据如何做到 100% 不丢失。

3 目前已有的技术方案

经过之前几篇文章的讨论，目前我们已经初步知道，第一个会导致数据丢失的地方，就是消费者获取到消息之后，没有来得及处理完毕，自己直接宕机了。

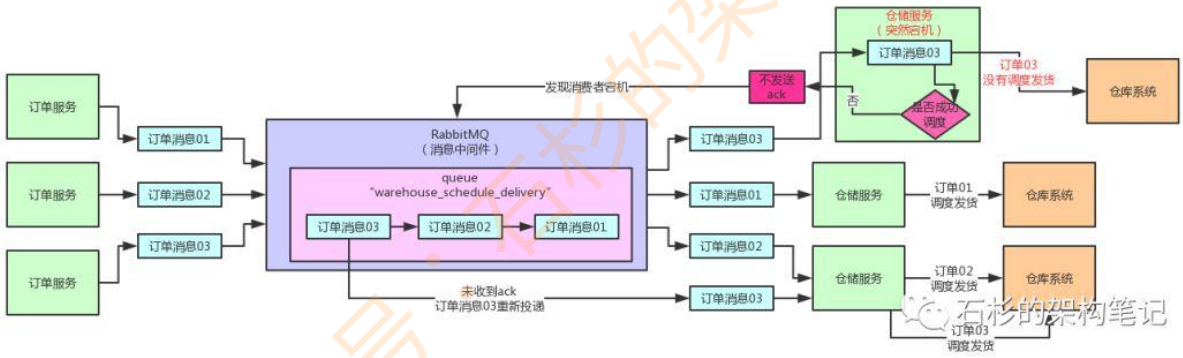
此时 RabbitMQ 的自动 ack 机制会通知 MQ 集群这条消息已经处理好了，MQ 集群就会删除这条消息。

那么这条消息不就丢失了么？不会有任何一个消费者处理到这条消息了。

所以之前我们详细讨论过，通过在消费者服务中调整为手动 ack 机制，来确保消息一定是已经成功处理完了，才会发送 ack 通知给 MQ 集群。

否则没发送 ack 之前消费者服务宕机，此时 MQ 集群会自动感知到，然后重发消息给其他的消费者服务实例。

《扎心！线上服务宕机时，如何保证数据 100% 不丢失？》 这篇文章，详细讨论了这个问题，手动 ack 机制之下的架构图如下所示：



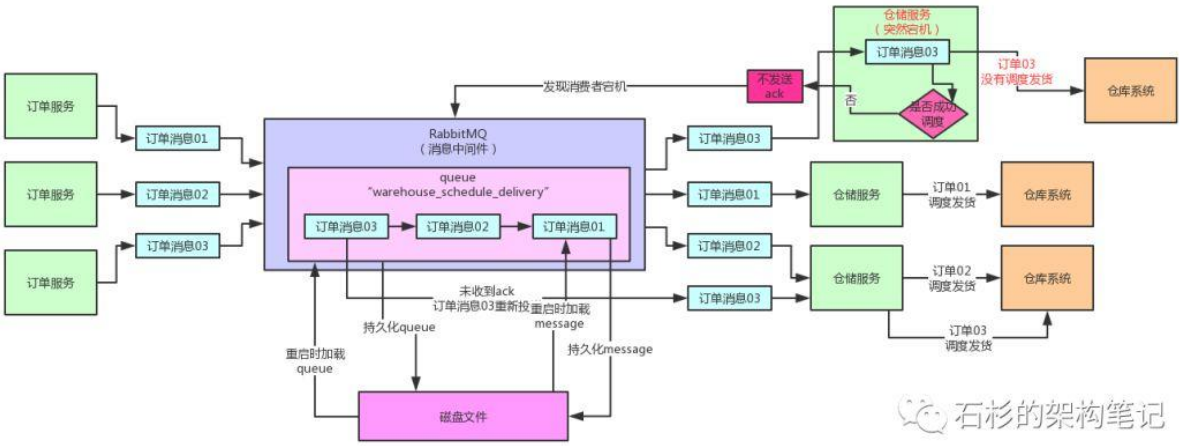
当时除了这个数据丢失问题之外，还有另外一个问题，就是 MQ 集群自身如果突然宕机，是不是会导致数据丢失？

默认情况下是肯定会的，因为 queue 和 message 都没采用持久化的方式来投递，所以 MQ 集群重启会导致部分数据丢失。

所以**《消息中间件集群崩溃，如何保证百万生产数据不丢失？》** 这篇文章，我们分析了如何采用持久化的方式来创建 queue，同时采用持久化的方式来投递消息到 MQ 集群，这样 MQ 集群会将消息持久化到磁盘上去。

此时如果消息还没来得及投递给消费者服务，然后 MQ 集群突然宕机了，数据是不会丢失的，因为 MQ 集群重启之后会自动从磁盘文件里加载出来没投递出去的消息，然后继续投递给消费者服务。

同样，该方案沉淀下来的系统架构图，如下所示：



4 数据 100% 不丢失了吗?

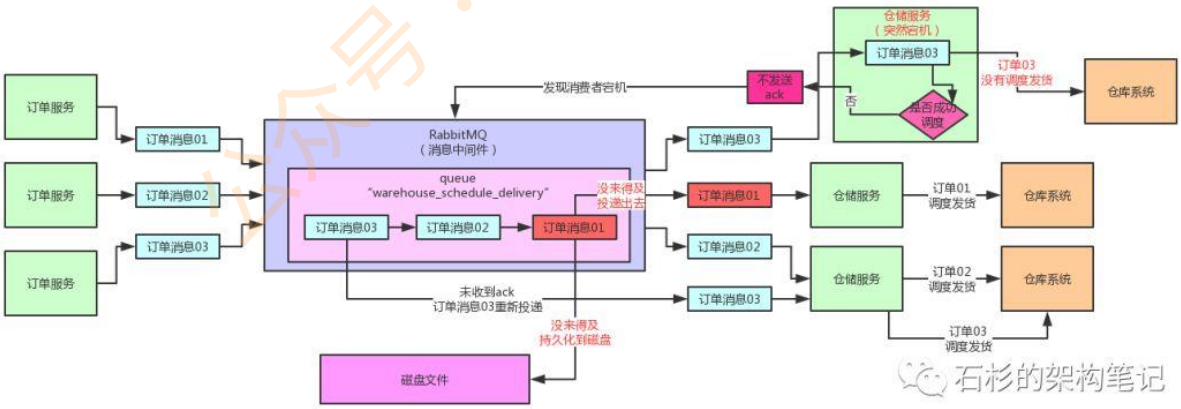
大家想一想，到目前为止，咱们的架构一定可以保证数据不丢失了吗?

其实，现在的架构，还是有一个数据可能会丢失的问题。

那就是上面作为生产者的订单服务把消息投递到 MQ 集群之后，暂时还驻留在 MQ 的内存里，还没来得及持久化到磁盘上，同时也还没来得及投递到作为消费者的仓储服务。

此时要是 MQ 集群自身突然宕机，咋办呢?

尴尬了吧，驻留在内存里的数据是一定会丢失的，我们来看看下面的图示。



5 按需制定技术方案

现在，我们需要考虑的技术方案是：订单服务如何保证消息一定已经持久化到磁盘?

实际上，作为生产者的订单服务把消息投递到 MQ 集群的过程是很容易丢数据的。

比如说网络出了点什么故障，数据压根儿没传输过去，或者就是上面说的消息刚刚被 MQ 接收但是还驻留在内存里，没落地到磁盘上，此时 MQ 集群宕机就会丢数据。

所以首先，我们得考虑一下作为生产者的订单服务要如何利用 RabbitMQ 提供的相关功能来实现一个技术方案。

这个技术方案需要保证：只要订单服务发送出去的消息确认成功了，此时 MQ 集群就一定已经将消息持久化到磁盘了。

我们必须实现这样的一个效果，才能保证投递到 MQ 集群的数据是不会丢失的。

6 需要研究的技术细节

这里我们需要研究的技术细节是：仓储服务手动 ack 保证数据不丢失的实现原理。

之前，笔者就收到很多同学提问：

- 仓储服务那块到底是如何基于手动 ack 就可以实现数据不丢失的?
- RabbitMQ 底层实现的细节和原理到底是什么?
- 为什么仓储服务没发送 ack 就宕机了，RabbitMQ 可以自动感知到他宕机了，然后自动重发消息给其他的仓储服务实例呢?

这些东西背后的实现原理和底层细节，到底是什么?

大伙儿稍安勿躁，接下来，咱们会通过一系列文章，仔细探究一下这背后的原理。

互联网面试必杀：如何保证消息中间件全链路数据100%不丢失（2）

> **作者:中华石杉** 原文地址

目录

1 前情提示

上一篇文章**《互联网面试必杀：如何保证消息中间件全链路数据 100% 不丢失（1）》**，我们初步介绍了之前制定的那些消息中间件数据不丢失的技术方案遗留的问题。

一个最大的问题，就是生产者投递出去的消息，可能会丢失。

丢失的原因有很多，比如消息在网络传输到一半的时候因为网络故障就丢了，或者是消息投递到 MQ 的内存时，MQ 突发故障宕机导致消息就丢失了。

针对这种生产者投递数据丢失的问题，RabbitMQ 实际上是提供了一些机制的。

比如，有一种重量级的机制，就是**事务消息机制**。采用类事务的机制把消息投递到 MQ，可以保证消息不丢失，但是性能极差，经过测试性能会呈现几百倍的下降。

所以说现在一般是不会用这种过于重量级的机制，而是会用**轻量级的 confirm 机制。**

但是我们这篇文章还不能直接讲解生产者保证消息不丢失的 confirm 机制，因为这种 confirm 机制实际上是采用了类似消费者的 ack 机制来实现的。

所以，要深入理解 confirm 机制，我们得先从这篇文章开始，深入的分析一下消费者手动 ack 机制保证消息不丢失的底层原理。

2 ack 机制回顾

其实手动 ack 机制非常的简单，必须要消费者确保自己处理完毕了一个消息，才能手动发送 ack 给 MQ，MQ 收到 ack 之后才会删除这个消息。

如果消费者还没发送 ack，自己就宕机了，此时 MQ 感知到他的宕机，就会重新投递这条消息给其他的消费者实例。

通过这种机制保证消费者实例宕机的时候，数据是不会丢失的。

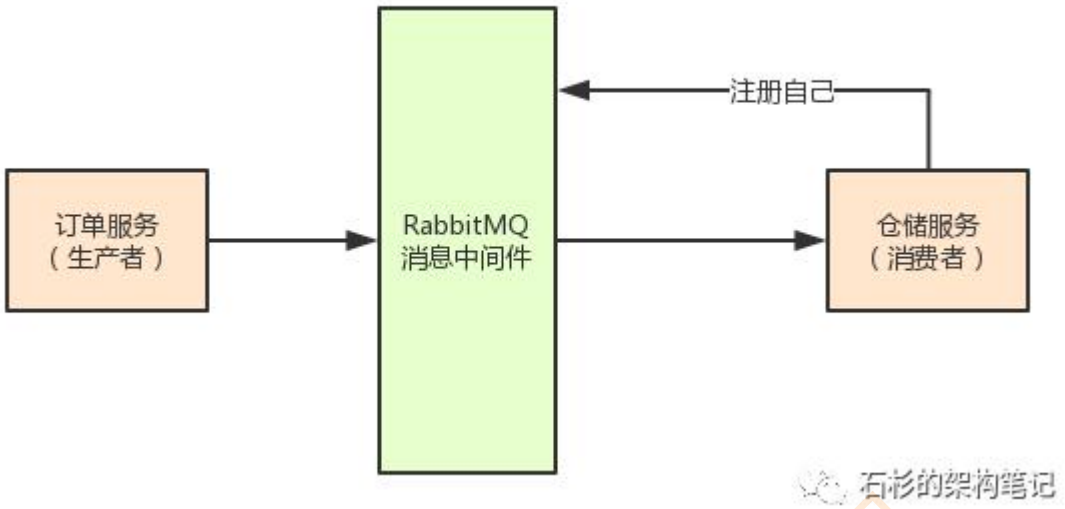
再次提醒一下大家，如果还对手动 ack 机制不太熟悉的同学，可以回头看一下之前的一篇文章：**《扎心！线上服务宕机时，如何保证数据 100% 不丢失？》**。然后这篇文章，我们将继续深入探讨一下 ack 机制的实现原理。

3 ack 机制实现原理：delivery tag

如果你写好了一个消费者服务的代码，让他开始从 RabbitMQ 消费数据，这时这个消费者服务实例就会自己注册到 RabbitMQ。

所以，RabbitMQ 其实是知道有哪些消费者服务实例存在的。

大家看看下面的图，直观的感受一下：



接着，RabbitMQ 就会通过自己内部的一个 “basic.delivery” 方法来投递消息到仓储服务里去，让他消费消息。

投递的时候，会给这次消息的投递带上一个重要的东西，就是 “delivery tag”，你可以认为是本次消息投递的一个唯一标识。

这个所谓的唯一标识，有点类似于一个 ID，比如说消息本次投递到一个仓储服务实例的唯一 ID。通过这个唯一 ID，我们就可以定位一次消息投递。

所以这个 delivery tag 机制不要看很简单，实际上他是后面要说的很多机制的核心基础。

而且这里要给大家强调另外一个概念，就是每个消费者从 RabbitMQ 获取消息的时候，都是通过一个 channel 的概念来进行的。

大家回看一下下面的消费者代码片段，我们必须是先对指定机器上部署的 RabbitMQ 建立连接，然后通过这个连接获取一个 channel。

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

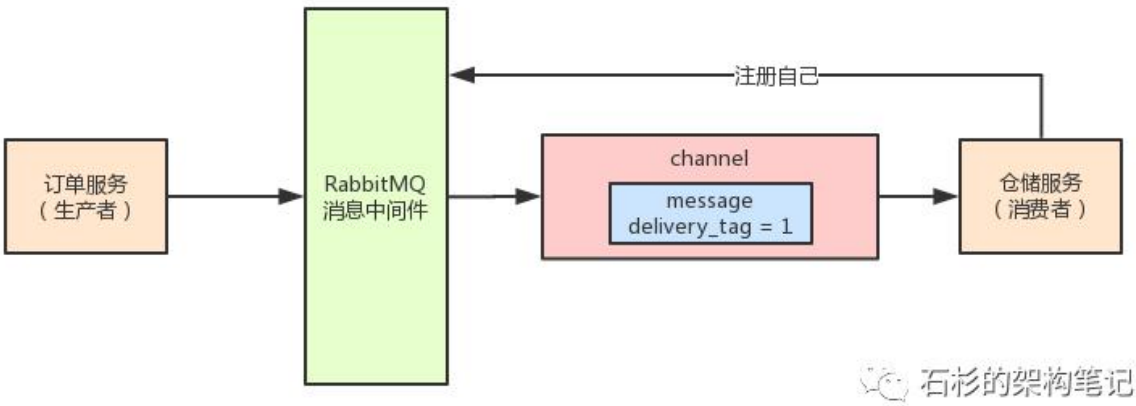

而且如果大家还有点印象的话，我们在仓储服务里对消息的消费、ack 等操作，全部都是基于这个 channel 来进行的，channel 又有点类似于是我们跟 RabbitMQ 进行通信的这么一个句柄，比如看看下面的代码：

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    try {

        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] 仓储服务接收到消息，准备执行调度发货的流程 '" + message + "'");

    } catch() {

        // 异常处理。。。

    } finally {
        System.out.println(" [x] 订单完成调度发货");

        // 这就是核心代码，手动在代码里对RabbitMQ进行ack
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    }
};

channel.basicConsume(QUEUE_NAME, false, deliverCallback, consumerTag -> { });
```


> 另外这里提一句：之前写那篇文章讲解手动 ack 保证数据不丢失的时候，有很多人提出疑问：为什么上面代码里直接是 try finally，如果代码有异常，那还是会直接执行 finally 里的手动 ack？其实很简单，自己加上 catch 就可以了。

好的，咱们继续。你大概可以认为这个 channel 就是进行数据传输的一个管道吧。对于每个 channel 而言，一个 “delivery tag” 就可以唯一的标识一次消息投递，这个 delivery tag 大致而言就是一个不断增长的数字。

大家来看看下面的图，相信会很好理解的：



如果采用手动 ack 机制，实际上仓储服务每次消费了一条消息，处理完毕完成调度发货之后，就会发送一个 ack 消息给 RabbitMQ 服务器，这个 ack 消息是会带上自己本次消息的 delivery

tag 的。

咱们看看下面的 ack 代码，是不是带上了一个 delivery tag?

```php
channel.basicAck(
        delivery.getEnvelope().getDeliveryTag(),
        false);
```

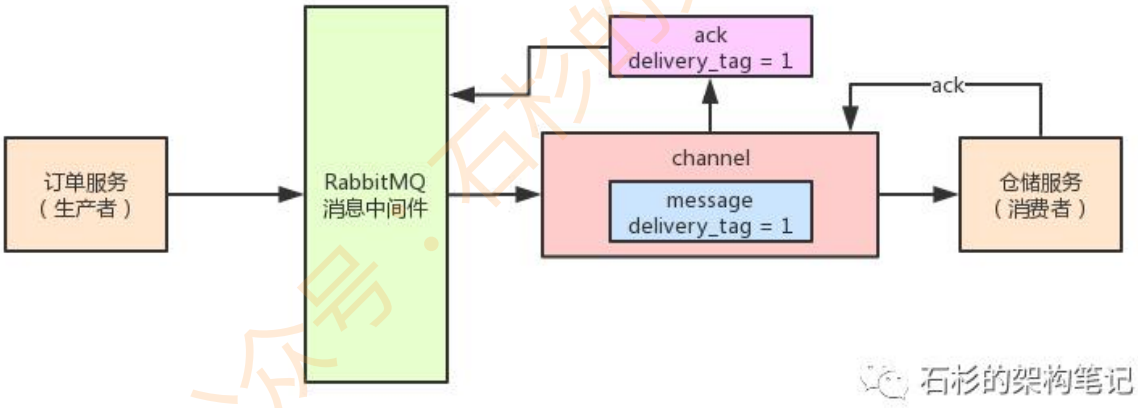
然后，RabbitMQ 根据哪个 channel 的哪个 delivery tag，不就可以唯一定位一次消息投递了?

接下来就可以对那条消息删除，标识为已经处理完毕。

这里大家必须注意的一点，就是 delivery tag 仅仅在一个 channel 内部是唯一标识消息投递的。

所以说，你 ack 一条消息的时候，必须是通过接受这条消息的同一个 channel 来进行。

大家看看下面的图，直观的感受一下。



其实这里还有一个很重要的点，就是我们可以设置一个参数，然后就批量的发送 ack 消息给 RabbitMQ，这样可以提升整体的性能和吞吐量。

比如下面那行代码，把第二个参数设置为 true 就可以了。

```php
channel.basicAck(
            delivery.getEnvelope().getDeliveryTag(),
            true);
```

看到这里，大家应该对这个 ack 机制的底层原理有了稍微进一步的认识了。起码是知道 delivery tag 是啥东西了，他是实现 ack 的一个底层机制。

然后，我们再来简单回顾一下自动 ack、手动 ack 的区别。

实际上默认用自动 ack，是非常简单的。RabbitMQ 只要投递一个消息出去给仓储服务，那么他立马就把这个消息给标记为删除，因为他是不管仓储服务到底接收到没有，处理完没有的。

所以这种情况下，性能很好，但是数据容易丢失。

如果手动 ack，那么就是必须等仓储服务完成商品调度发货以后，才会手动发送 ack 给 RabbitMQ，此时 RabbitMQ 才会认为消息处理完毕，然后才会标记消息为删除。

这样在发送 ack 之前，仓储服务宕机，RabbitMQ 会重发消息给另外一个仓储服务实例，保证数据不丢。

4 RabbitMQ 如何感知到仓储服务实例宕机

之前就有同学提出过这个问题，但是其实要搞清楚这个问题，其实不需要深入的探索底层，只要自己大致的思考和推测一下就可以了。

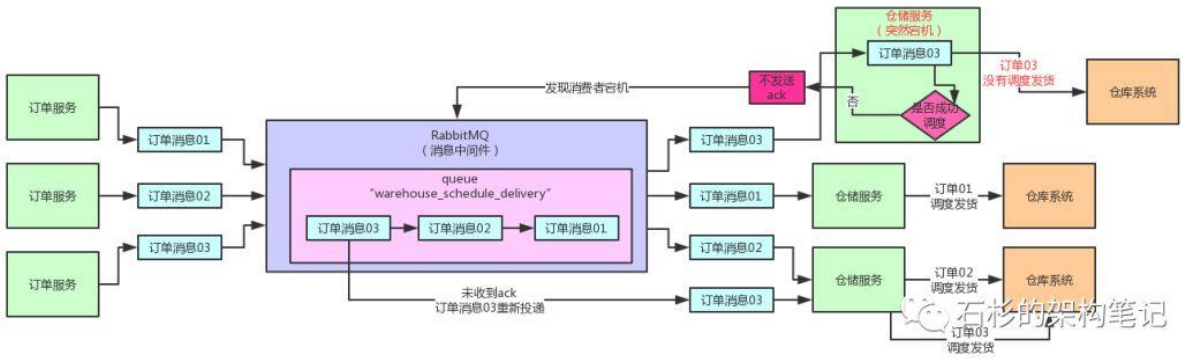
如果你的仓储服务实例接收到了消息，但是没有来得及调度发货，没有发送 ack，此时他宕机了。

我们想一想就知道，RabbitMQ 之前既然收到了仓储服务实例的注册，因此他们之间必然是建立有某种联系的。

一旦某个仓储服务实例宕机，那么 RabbitMQ 就必然会感知到他的宕机，而且对发送给他的还没 ack 的消息，都发送给其他仓储服务实例。

所以这个问题以后有机会我们可以深入聊一聊，在这里，大家其实先建立起来这种认识即可。

我们再回头看看下面的架构图：



5 仓储服务处理失败时的消息重发

首先，我们来看看下面一段代码：

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    try {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] 仓储服务接收到消息，准备执行调度发货的流程 '" + message + "'");
    } catch() {

        // 异常处理。。。

    } finally {
        System.out.println(" [x] 订单完成调度发货");

        // 这就是核心代码，手动在代码里对RabbitMQ进行ack
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    }
};
```


假如说某个仓储服务实例处理某个消息失败了，此时会进入 catch 代码块，那么此时我们怎么办呢？难道还是直接 ack 消息吗？

当然不是了，你要是还是 ack，那会导致消息被删除，但是实际没有完成调度发货。

这样的话，数据不是还是丢失了吗？因此，合理的方式是使用 nack 操作。

就是通知 RabbitMQ 自己没处理成功消息，然后让 RabbitMQ 将这个消息再次投递给其他的仓储服务实例尝试去完成调度发货的任务。

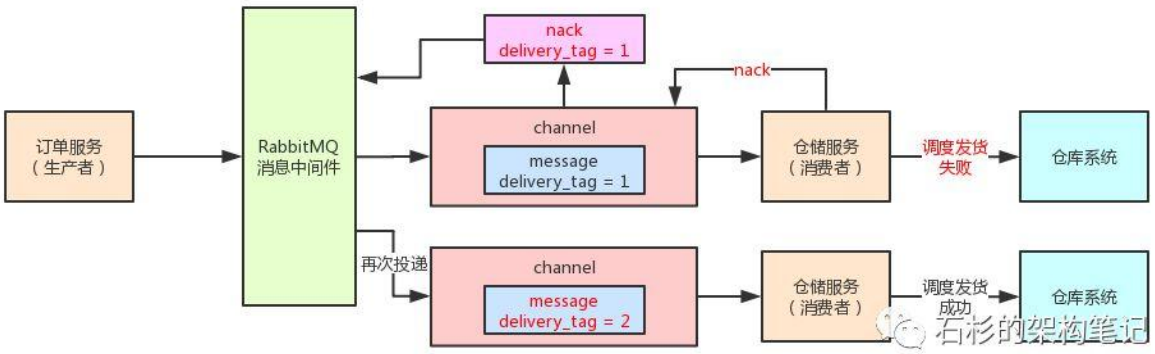
我们只要在 catch 代码块里加入下面的代码即可：

```php
    channel.basicNack(
                delivery.getEnvelope().getDeliveryTag(),
                true);
```

注意上面第二个参数是 true，意思就是让 RabbitMQ 把这条消息重新投递给其他的仓储服务实例，因为自己没处理成功。

你要是设置为 false 的话，就会导致 RabbitMQ 知道你处理失败，但是还是删除这条消息，这是不对的。

同样，我们还是来一张图，大家一起来感受一下：



6 阶段总结

这篇文章对之前的 ack 机制做了进一步的分析，包括底层的 delivery tag 机制，以及消息处理失败时的消息重发。

通过 ack 机制、消息重发等这套机制的落地实现，就可以保证一个消费者服务自身突然宕机、消息处理失败等场景下，都不会丢失数据。

面试大杀器：消息中间件如何实现消费吞吐量的百倍优化？

> 作者:中华石杉 原文地址

目录

1 前情提示

上一篇文章：**互联网面试必杀：如何保证消息中间件全链路数据 100% 不丢失（2）**，我们分析了 ack 机制的底层实现原理（delivery tag 机制），还有消除处理失败时的 nack 机制如何触发消息重发。

通过这个，已经让大家进一步对消费端保证数据不丢失的方案的理解更进一层了。

这篇文章，我们将会对 ack 底层的 delivery tag 机制进行更加深入的分析，让大家理解的更加透彻一些。

面试时，如果被问到消息中间件数据不丢失问题的时候，可以更深入到底层，给面试官进行分析。

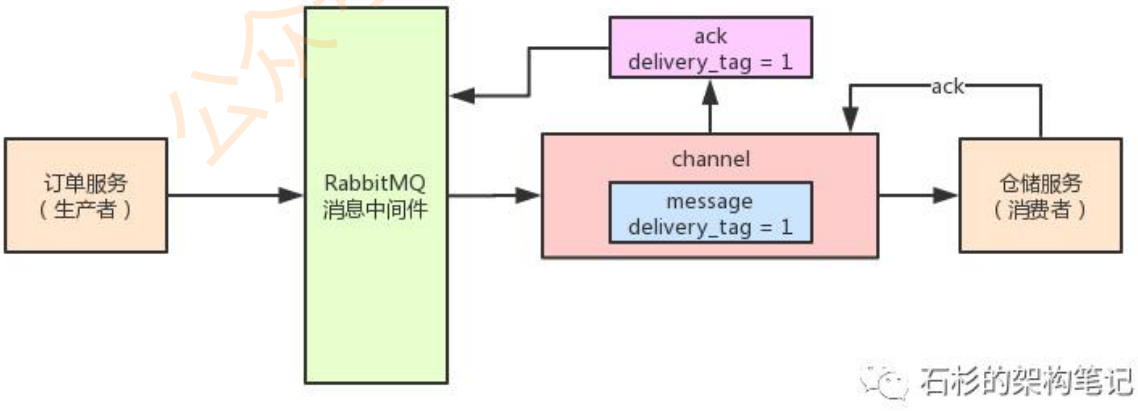
2 unack 消息的积压问题

首先，我们要给大家介绍一下 RabbitMQ 的 prefetch count 这个概念。

大家看过上篇文章之后应该都知道了，对每个 channel（其实对应了一个消费者服务实例，你大体可以这么来认为），RabbitMQ 投递消息的时候，都是会带上本次消息投递的一个 delivery tag 的，唯一标识一次消息投递。

然后，我们进行 ack 时，也会带上这个 delivery tag，基于同一个 channel 进行 ack，ack 消息里会带上 delivery tag 让 RabbitMQ 知道是对哪一次消息投递进行了 ack，此时就可以对那条消息进行删除了。

大家先来看一张图，帮助大家回忆一下这个 delivery tag 的概念。



所以大家可以考虑一下，对于每个 channel 而言（你就认为是针对每个消费者服务实例吧，比如一个仓储服务实例），其实都有一些处于 unack 状态的消息。

比如 RabbitMQ 正在投递一条消息到 channel，此时消息肯定是 unack 状态吧？

然后仓储服务接收到一条消息以后，要处理这条消息需要耗费时间，此时消息肯定是 unack 状态吧？

同时，即使你执行了 ack 之后，你要知道这个 ack 他默认是异步执行的，尤其如果你开启了批量 ack 的话，更是有一个延迟时间才会 ack 的，此时消息也是 unack 吧？

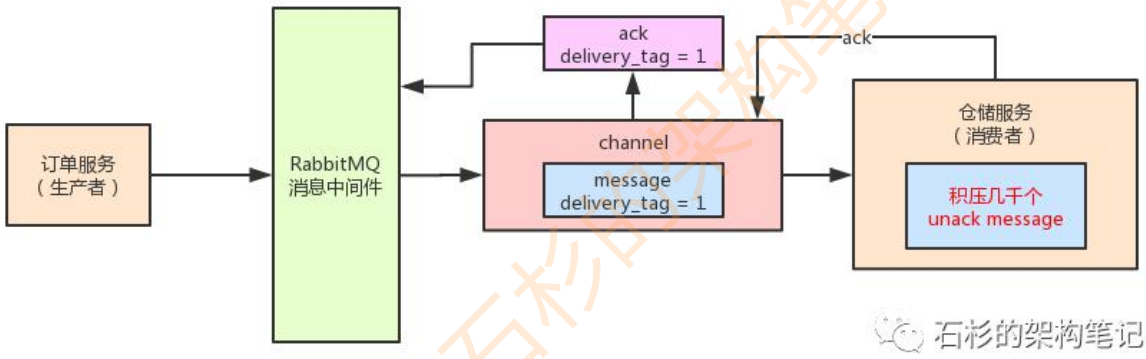
那么大家考虑一下，RabbitMQ 他能够无限制的不停给你的消费者服务实例推送消息吗？

明显是不能的，如果 RabbitMQ 给你的消费者服务实例推送的消息过多过快，比如都有几千条消息积压在某个消费者服务实例的内存中。

那么此时这几千条消息都是 unack 的状态，一直积压着，是不是有可能会导致消费者服务实例的内存溢出？内存消耗过大？甚至内存泄露之类的问题产生？

所以说，RabbitMQ 是必须要考虑一下消费者服务的处理能力的。

大家看看下面的图，感受一下如果消费者服务实例的内存中积压消息过多，都是 unack 的状态，此时会怎么样。



3 如何解决 unack 消息的积压问题

正是因为这个原因，RabbitMQ 基于一个 prefetch count 来控制这个 unack message 的数量。

你可以通过 “channel.basicQos(10)” 这个方法来设置当前 channel 的 prefetch count。

举个例子，比如你要是设置为 10 的话，那么意味着当前这个 channel 里，unack message 的数量不能超过 10 个，以此来避免消费者服务实例积压 unack message 过多。

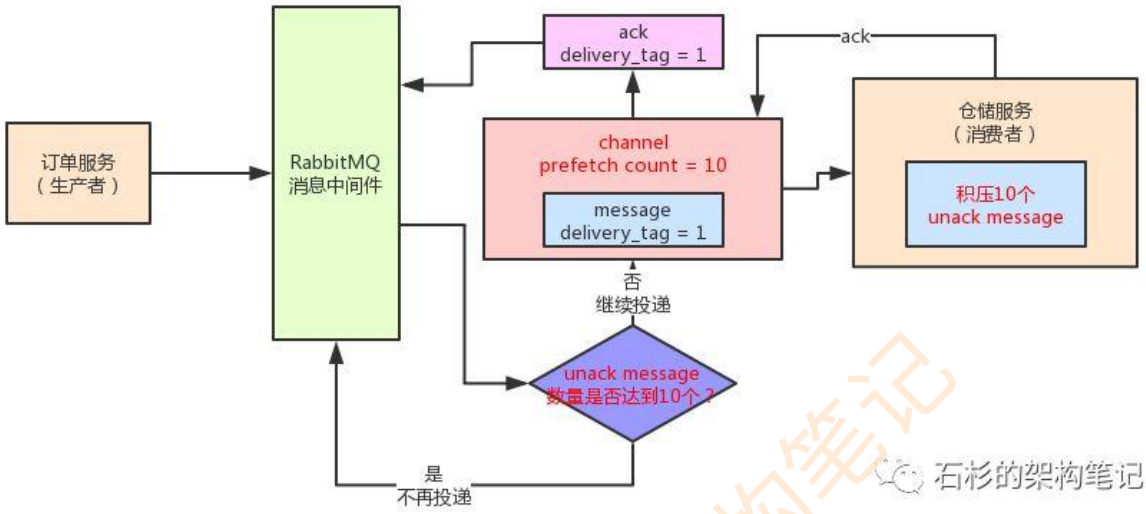
这样的话，就意味着 RabbitMQ 正在投递到 channel 过程中的 unack message，以及消费者服务在处理中的 unack message，以及异步 ack 之后还没完成 ack 的 unack message，所有这些 message 加起来，一个 channel 也不能超过 10 个。

如果你要简单粗浅的理解的话，也大致可以理解为这个 prefetch count 就代表了一个消费者服务同时最多可以获取多少个 message 来处理。所以这里也点出了 prefetch 这个单词的意思。

prefetch 就是预抓取的意思，就意味着你的消费者服务实例预抓取多少条 message 过来处理，但是最多只能同时处理这么多消息。

如果一个 channel 里的 unack message 超过了 prefetch count 指定的数量，此时 RabbitMQ 就会停止给这个 channel 投递消息了，必须要等待已经投递过去的消息被 ack 了，此时才能继续投递下一个消息。

老规矩，给大家上一张图，我们一起来看看这个东西是啥意思。



4 高并发场景下的内存溢出问题

好！现在大家对 ack 机制底层的另外一个核心机制：prefetch 机制也有了一个深刻的理解了。

此时，咱们就应该来考虑一个问题了。就是如何来设置这个 prefetch count 呢？这个东西设置的过大或者过小有什么影响呢？

其实大家理解了上面的图就很好理解这个问题了。

假如说我们把 prefetch count 设置的很大，比如说 3000，5000，甚至 100000，就这样特别大的值，那么此时会如何呢？

这个时候，在高并发大流量的场景下，可能就会导致消费者服务的内存被快速的消耗掉。

因为假如说现在 MQ 接收到的流量特别的大，每秒都上千条消息，而且此时你的消费者服务的 prefetch count 还设置的特别大，就会导致可能一瞬间你的消费者服务接收到了达到 prefetch count 指定数量的消息。

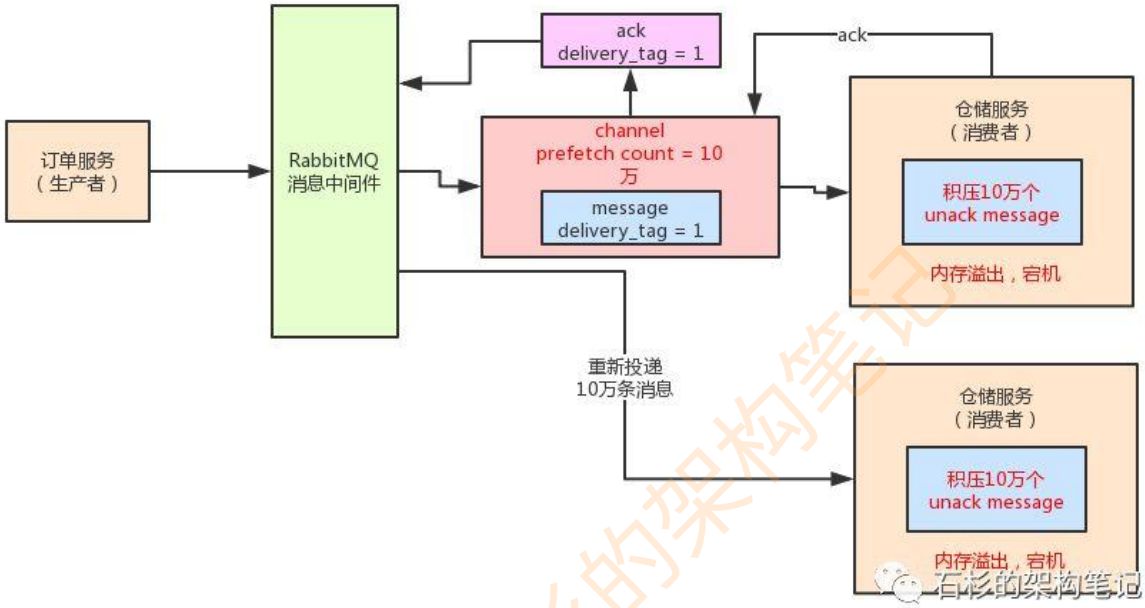
打个比方，比如一下子你的消费者服务内存里积压了 10 万条消息，都是 unack 的状态，反正你的 prefetch count 设置的是 10 万。

那么对一个 channel，RabbitMQ 就会最多容忍 10 万个 unack 状态的消息，在高并发下也就最多可能积压 10 万条消息在消费者服务的内存里。

那么此时导致的结果，就是消费者服务直接被击垮了，内存溢出，OOM，服务宕机，然后大量 unack 的消息会被重新投递给其他的消费者服务，此时其他消费者服务一样的情况，直接宕机，最后造成雪崩效应。

所有的消费者服务因为扛不住这么大的数据量，全部宕机。

大家来看看下面的图，自己感受一下现场的氛围。



5 低吞吐量问题

那么如果反过来呢，我们要是把 prefetch count 设置的很小会如何呢?

比如说我们把 prefetch count 设置为 1？此时就必然会导致消费者服务的吞吐量极低。因为你即使处理完一条消息，执行 ack 了也是异步的。

给你举个例子，假如说你的 prefetch count = 1，RabbitMQ 最多投递给你 1 条消息处于 unack 状态。

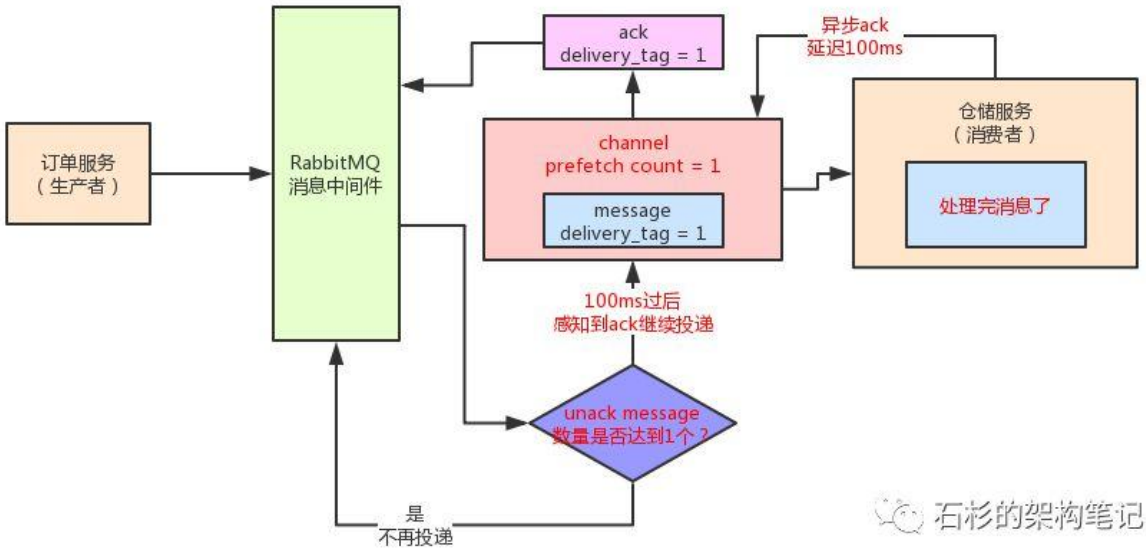
此时比如你刚处理完这条消息，然后执行了 ack 的那行代码，结果不幸的是，ack 需要异步执行，也就是需要 100ms 之后才会让 RabbitMQ 感知到。

那么 100ms 之后 RabbitMQ 感知到消息被 ack 了，此时才会投递给你下一条消息！

这就尴尬了，在这 100ms 期间，你的消费者服务是不是啥都没干啊?

这不就直接导致了你的消费者服务处理消息的吞吐量可能下降 10 倍，甚至百倍，千倍，都有这种可能！

大家看看下面的图，感受一下低吞吐量的现场。



6 合理的设置 prefetch count

所以鉴于上面两种极端情况，RabbitMQ 官方给出的建议是 prefetch count 一般设置在 100~300 之间。

也就是一个消费者服务最多接收到 100~300 个 message 来处理，允许处于 unack 状态。

这个状态下可以兼顾吞吐量也很高，同时也不容易造成内存溢出的问题。

但是其实在我们的实践中，这个 prefetch count 大家完全是可以自己去压测一下的。

比如说慢慢调节这个值，不断加大，观察高并发大流量之下，吞吐量是否越来越大，而且观察消费者服务的内存消耗，会不会 OOM、频繁 FullGC 等问题。

7 阶段性总结

其实通过最近几篇文章，基本上已经把消息中间件的消费端如何保证数据不丢失这个问题剖析的较为深入和透彻了。

如果你是基于 RabbitMQ 来做消息中间件的话，消费端的代码里，必须考虑三个问题：手动 ack、处理失败的 nack、prefetch count 的合理设置

这三个问题背后涉及到了各种机制：

- 自动 ack 机制
- delivery tag 机制
- ack 批量与异步提交机制

- 消息重发机制
- 手动 nack 触发消息重发机制
- prefetch count 过大导致内存溢出问题
- prefetch count 过小导致吞吐量过低

这些底层机制和问题，咱们都一步步分析清楚了。

所以到现在，单论消费端这块的数据不丢失技术方案，相信大家在面试的时候就可以有一整套自己的理解和方案可以阐述了。

接下来下篇文章开始，我们就来具体聊一聊：**消息中间件的生产端如何保证数据不丢失。**

Java进阶必备：优雅的告诉面试官消息中间件该如何实现高可用架构?

> **作者:中华石杉** 原文地址

目录

（1）背景引入

> ! 这篇文章，我们来聊一下消息中间件高可用架构的一些原理。

对于一个合格的高级 Java 工程师而言，你肯定会碰到在系统里用到 MQ 的场景，那么这个时候你需要基于你的业务场景和需求，考虑在使用 MQ 的时候可能遇到的一些技术问题。

接着，你必须得针对这些技术问题设计一套完整的技术方案。

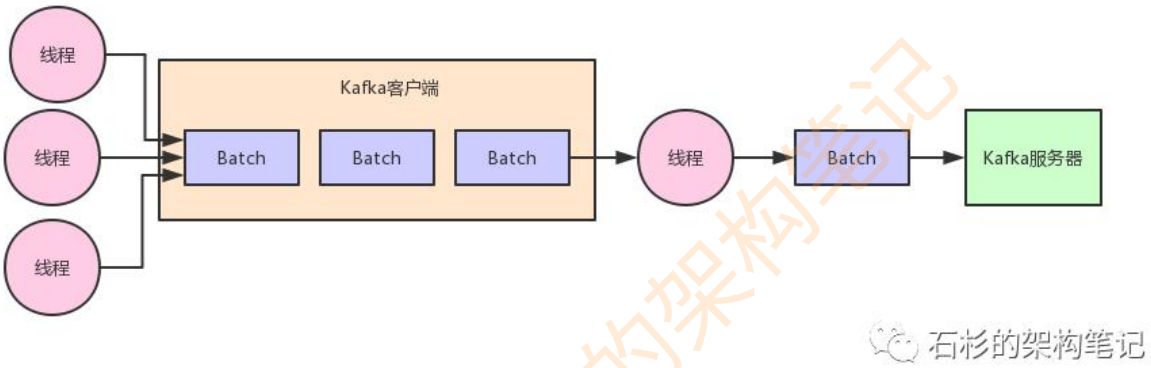
你需要从消息的订阅模式、消息的生产到消费全链路不丢数据、消息中间件本身如何保证高可用，等各个角度切入，来考虑好你的系统和 MQ 对接之后的完整技术方案。

所以，本文就来聊聊消息中间件高可用的架构原理。

（2）先来思考一下消息中间件的可用性问题

咱们先抛开各种具体的技术，就来思考一下，啥是 MQ 的可用性问题？

大家看看下面的图，其实道理很简单，假如你的 MQ 就部署在一台机器上，那么正常情况下，生产者都会发送消息到 MQ 去，然后让消费者获取到。



但是万一天有不测风云，MQ 部署的那台机器，因为一些莫名的原因，MQ 自己本身的进程挂掉了，或者是那台机器直接就宕机了，那么此时怎么办呢？

很尴尬，是不是，结果是很明显的，生产者没法发送数据出去，然后消费者也没法获取到数据了。

然后整个系统不就完蛋了？因为系统的核心流程根本无法跑通了，对不对？

MQ 宕机就直接导致你的系统本身也故障了，然后可能会导致你的公司对外的 APP、网站等产品就无法运作了，用户无法使用你们公司的服务了。

如果你们公司是电商平台、外卖平台、社交平台。那么来这么一出，不是会导致公司损失惨重？

如果你的系统持续几个小时无法被人使用，本来你公司电商平台一天营收可以达到 1 亿，结果现在导致几个小时内无法下单购买商品，最后当天营收就 5000 万，那么你的公司是不是直接活生生损失了 5000 万？

这个真的不是开玩笑的，如果大家留意互联网行业的新闻的话和小道消息的话，就应该知道近几年一些大型互联网公司都出现过类似的情况，损失惨重，咱们做码农的就得被祭天了是不是？

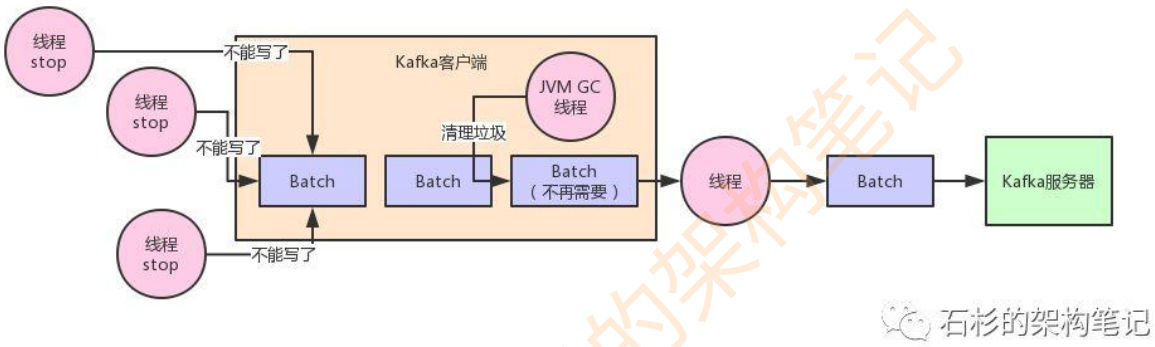
（3）集群化部署 + 数据多副本冗余

好，问题来了！现在你感觉一个 MQ 中间件应该如何实现高可用呢?

这里的方式有很多种，比如说数据多副本冗余，集群镜像同步机制，我们就抛开具体的技术来从本质层面思考一下 MQ 集群实现高可用的几种方式。

先来看下面的一张图，假设我们写到 MQ 的数据都被多副本冗余了，也就是你写的每一条消息都被复制到了其他的机器上去了。

那么此时任何一台机器宕机，似乎都不会影响我们跟 MQ 继续通信，而且写出去的数据似乎也都还在。



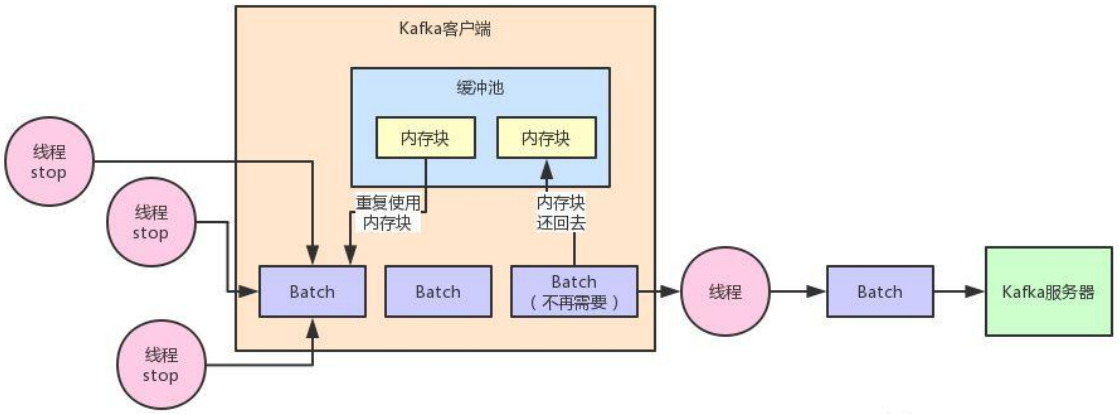
上面的图里，MQ 采用集群模式部署到了 2 台机器上去，然后生产者给其中一台机器写入一条消息，该机器自动同步复制给另外一台机器。

此时数据在 2 台机器上，就有 2 个副本了，那么如果第一台机器宕机了，会影响我们吗?

答案是：不会。

因为数据本身是多副本冗余的，此时消费者完全可以从第二台机器消费到这条消息，并且生产者还可以继续给第二台机器写入消息，数据没丢失。

而且，系统根本不用中断流程，还可以继续运行，我们看下面的图。



这种感觉是不是很棒？实际上这种 MQ 集群化部署架构以及数据多副本冗余机制，是非常常见的一种高可用架构。

Kafka 这个极为优秀的消息中间件，就是采用的这种架构保证高可用、数据容错性。

（4）多副本同步复制强制要求

但是这里你要思考另外几个问题，第一个就是：你在写数据到其中一台机器的时候，是不是得要求，必须得让那台机器复制数据到另外一台机器了，保证集群里一定有这条数据双副本了，才可以认为本次写成功了？

没错，假如你要是不能保证这一点，比如你就写数据给了其中一台机器，然后他还没来得及复制给另外一台机器呢，直接第一台机器就宕机了。

此时虽然你可以继续基于第二台机器发送消息和消费消息，但是你刚才发送的一条消息就丢失了。

大家看下面的图来理解一下这个场景。

所以对于采用这种机制的时候，你必须得让生产者通过一些参数的设置，保证说写一条消息到某台机器，他必须同步这条消息到另外一台机器成功，集群里有双副本了，然后此时才可以认为这条消息写成功了。

但凡刚写一台机器他就宕机，还没来得及复制到另外一台机器的话，本次写应该报错失败，然后你应该重试再次写入数据到 MQ 集群里去。

大家看看下面的图。只要你一次写成功了，他就保证肯定已经同步数据为双副本了，此时哪怕一台机器宕机，数据不会丢失，生产和消费都可以有条不紊的继续进行。

（5）多机器承载多副本强制要求

第二个问题，假如说现在你的集群中本来有两台机器，现在宕机了其中的一台，只有一台机器了，你还能允许你的生产者对唯一的一台机器继续写入数据吗？

答案是：否。

因为如果集群里只有一台机器可以承载写入，那么万一剩余的一台机器又宕机了呢？是不是还是会导致数据丢失，集群完蛋？

所以说，你的生产者同理应该基于参数设置一下，集群里必须有超过 2 台机器可以接收你的数据副本复制。

否则如果只有 1 台机器可以接受你的数据副本复制的话，那么还是算了。

大家看看下面的图，感受一下那个场景。

假设集群里有 3 台机器，那么其中一台宕机了，你后续再写入另外一台的时候，判断一下集群里还有剩余两台机器，足以保证数据双副本的高可用性和容错性，所以可以继续正常的写入数据到 MQ 集群里去。

实际上，上面说的那一整套的机制，在 Kafka 里都可以采用，他有对应的一些参数可以配置数据有几个副本，包括你每次写入必须复制到几台机器才可以算成功，否则就要重新发送，以及你的集群剩余机器必须可以承载几个副本才能继续写入数据。

通过这一整套方案的设计和基于具体技术的落地，才可以保证在集群化部署的情况下，集群必须有几台机器承载多副本，同时数据写入之后必须是保证多副本冗余的。

此时，任何机器宕机，数据都不会丢失，还可以正常让系统继续运行。

（6）架构原理与技术无关性

其实本文对消息中间件的集群高可用架构的探讨，是完全脱离于某个具体技术的，非常朴素的从本质的原理层面来讨论这个话题。

具体的 RabbitMQ、Kafka、RocketMQ 等各种不同的消息中间件，对这种高可用架构的实现，都有一定的相似想通性，但是也都有各自不同的技术实现，以及相对应的区别。

后面我们再通过不同的文章，以各种 MQ 中间件的具体技术实现举例来讨论一下相关的架构是如何落地的。

面试官：消息中间件如何实现每秒几十万的高并发写入？

> 作者:中华石杉 原文地址

目录

> ! “ 这篇文章来聊一下 Kafka 的一些架构设计原理，这也是互联网公司面试时非常高频的技术考点。

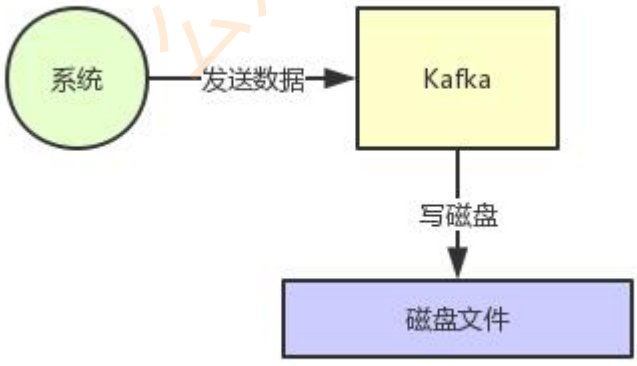
Kafka 是高吞吐低延迟的高并发、高性能的消息中间件，在大数据领域有极为广泛的运用。配置良好的 Kafka 集群甚至可以做到每秒几十万、上百万的超高并发写入。

那么 Kafka 到底是如何做到这么高的吞吐量和性能的呢？这篇文章我们来一点一点说一下。

1、页缓存技术 + 磁盘顺序写

首先 Kafka 每次接收到数据都会往磁盘上去写，如下图所示。

那么在这里我们不禁有一个疑问了，如果把数据基于磁盘来存储，频繁的往磁盘文件里写数据，这个性能会不会很差？大家肯定都觉得磁盘写性能是极差的。



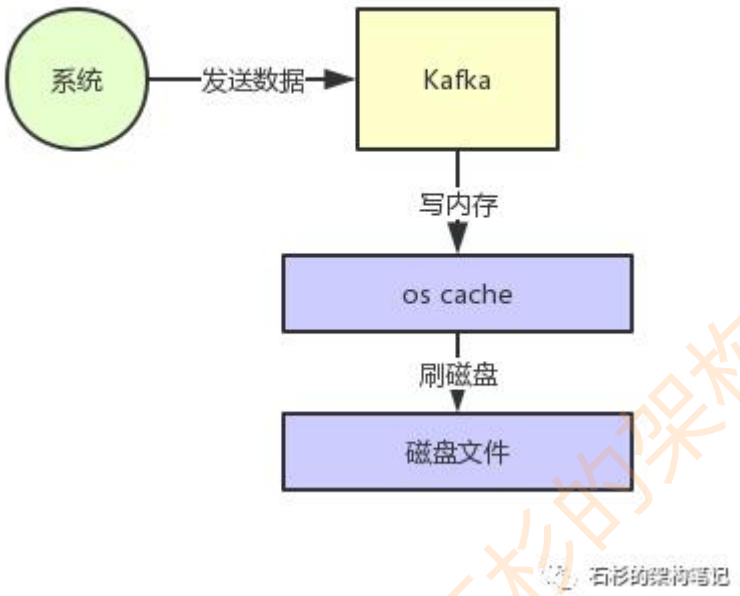
没错，要是真的跟上面那个图那么简单的话，那确实这个性能是比较差的。

但是实际上 Kafka 在这里有极为优秀和出色的设计，就是为了保证数据写入性能，首先 Kafka 是基于操作系统的页缓存来实现文件写入的。

操作系统本身有一层缓存，叫做 page cache，是在内存里的缓存，我们也可以称之为 os cache，意思就是操作系统自己管理的缓存。

你在写入磁盘文件的时候，可以直接写入这个 os cache 里，也就是仅仅写入内存中，接下来由操作系统自己决定什么时候把 os cache 里的数据真的刷入磁盘文件中。

仅仅这一个步骤，就可以将磁盘文件写性能提升很多了，因为其实这里相当于是在写内存，不是在写磁盘，大家看下图。



接着另外一个就是 kafka 写数据的时候，非常关键的一点，他是以磁盘顺序写的方式来写的。也就是说，仅仅将数据追加到文件的末尾，不是在文件的随机位置来修改数据。

普通的机械磁盘如果你要是随机写的话，确实性能极差，也就是随便找到文件的某个位置来写数据。

但是如果你是追加文件末尾按照顺序的方式来写数据的话，那么这种磁盘顺序写的性能基本上可以跟写内存的性能本身也是差不多的。

所以大家就知道了，上面那个图里，Kafka 在写数据的时候，一方面基于了 os 层面的 page cache 来写数据，所以性能很高，本质就是在写内存罢了。

另外一个，他是采用磁盘顺序写的方式，所以即使数据刷入磁盘的时候，性能也是极高的，也跟写内存是差不多的。

基于上面两点，kafka 就实现了写入数据的超高性能。

那么大家想想，假如说 kafka 写入一条数据要耗费 1 毫秒的时间，那么是不是每秒就是可以写入 1000 条数据?

但是假如 kafka 的性能极高，写入一条数据仅仅耗费 0.01 毫秒呢？那么每秒是不是就可以写入 10 万条数？

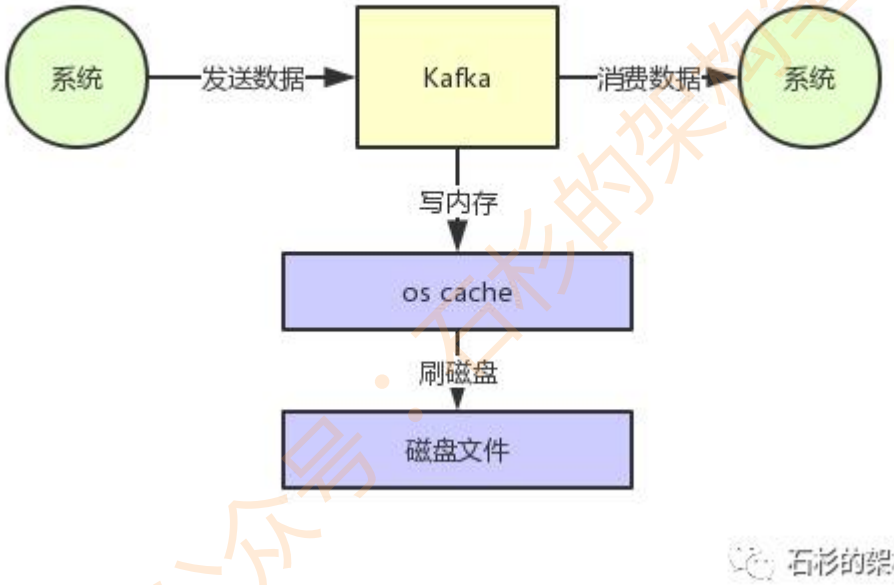
所以要保证每秒写入几万甚至几十万条数据的核心点，就是尽最大可能提升每条数据写入的性能，这样就可以在单位时间内写入更多的数据量，提升吞吐量。

2、零拷贝技术

说完了写入这块，再来谈谈消费这块。

大家应该都知道，从 Kafka 里我们经常要消费数据，那么消费的时候实际上就是要从 kafka 的磁盘文件里读取某条数据然后发送给下游的消费者，如下图所示。

那么这里如果频繁的从磁盘读数据然后发给消费者，性能瓶颈在哪里呢？

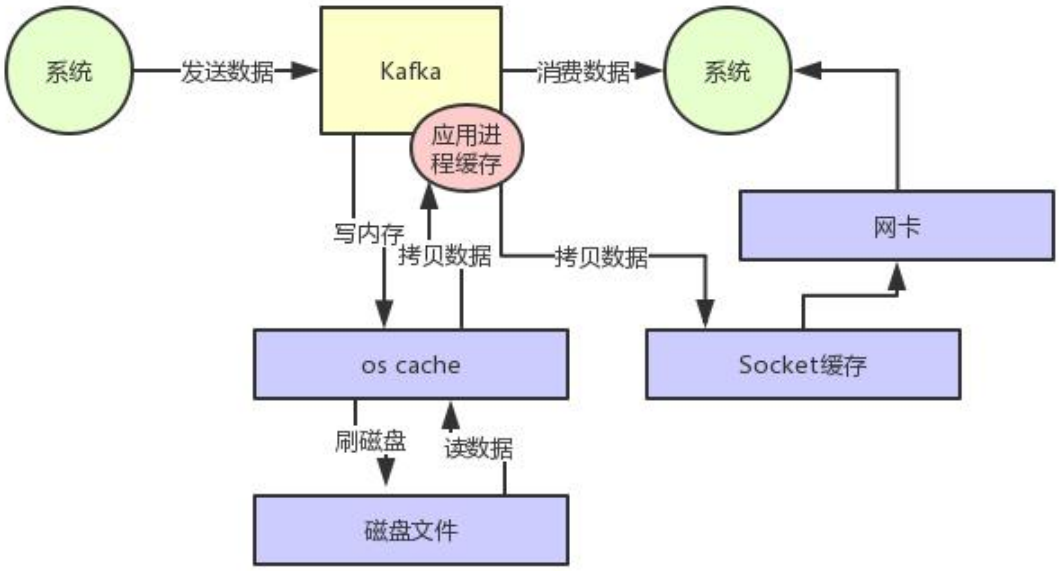


假设要是 kafka 什么优化都不做，就是很简单的从磁盘读数据发送给下游的消费者，那么大概过程如下所示：

先看看要读的数据在不在 os cache 里，如果不在的话就从磁盘文件里读取数据后放入 os cache。

接着从操作系统的 os cache 里拷贝数据到应用程序进程的缓存里，再从应用程序进程的缓存里拷贝数据到操作系统层面的 Socket 缓存里，最后从 Socket 缓存里提取数据后发送到网卡，最后发送出去给下游消费。

整个过程，如下图所示：



大家看上图，很明显可以看到有两次没必要的拷贝吧！

一次是从操作系统的 cache 里拷贝到应用进程的缓存里，接着又从应用程序缓存里拷贝回操作系统的 Socket 缓存里。

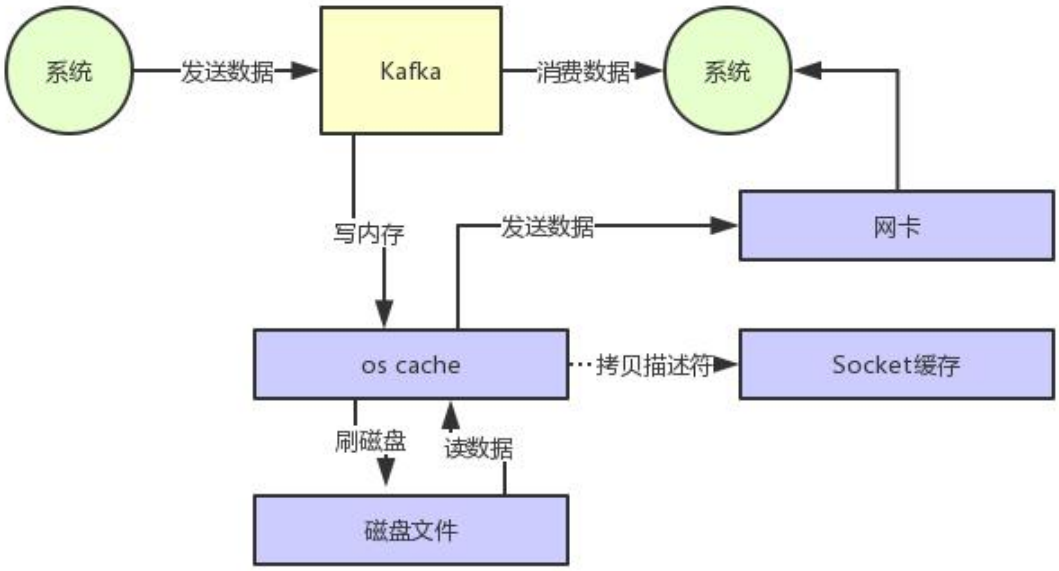
而且为了进行这两次拷贝，中间还发生了好几次上下文切换，一会儿是应用程序在执行，一会儿上下文切换到操作系统来执行。

所以这种方式来读取数据是比较消耗性能的。

Kafka 为了解决这个问题，在读数据的时候是引入**零拷贝技术。**

也就是说，直接让操作系统的 cache 中的数据发送到网卡后传输给下游的消费者，中间跳过了两次拷贝数据的步骤，Socket 缓存中仅仅会拷贝一个描述符过去，不会拷贝数据到 Socket 缓存。

大家看下图，体会一下这个精妙的过程：



通过零拷贝技术，就不需要把 os cache 里的数据拷贝到应用缓存，再从应用缓存拷贝到 Socket 缓存了，两次拷贝都省略了，所以叫做零拷贝。

对 Socket 缓存仅仅就是拷贝数据的描述符过去，然后数据就直接从 os cache 中发送到网卡上去了，这个过程大大的提升了数据消费时读取文件数据的性能。

而且大家会注意到，在从磁盘读数据的时候，会先看看 os cache 内存中是否有，如果有的话，其实读数据都是直接读内存的。

如果 kafka 集群经过良好的调优，大家会发现大量的数据都是直接写入 os cache 中，然后读数据的时候也是从 os cache 中读。

相当于是 Kafka 完全基于内存提供数据的写和读了，所以这个整体性能会极其的高。

说个题外话，下回有机会给大家说一下 Elasticsearch 的架构原理，其实 ES 底层也是大量基于 os cache 实现了海量数据的高性能检索的，跟 Kafka 原理类似。

3、最后的总结

通过这篇文章对 kafka 底层的页缓存技术的使用，磁盘顺序写的思路，以及零拷贝技术的运用，大家应该就明白 Kafka 每台机器在底层对数据进行写和读的时候采取的是什么样的思路，为什么他的性能可以那么高，做到每秒几十万的吞吐量。

这种设计思想对我们平时自己设计中间件的架构，或者是出去面试的时候，都有很大的帮助。

面试官：请谈谈写入消息中间件的数据，如何保证不丢失?

> 作者:中华石杉 原文地址

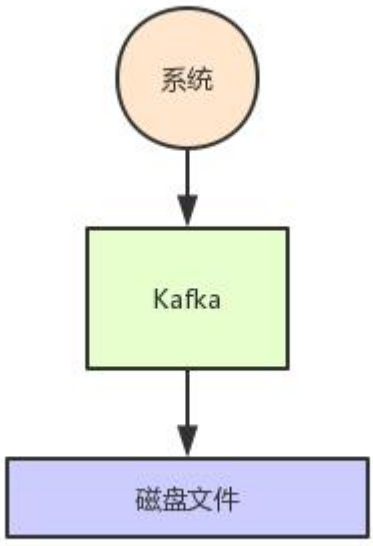
目录

（1）背景引入

> ! 这篇文章，给大家聊一下写入 Kafka 的数据该如何保证其不丢失?

看过之前的文章**《面试官：消息中间件如何实现每秒几十万的高并发写入》**的同学，应该都知道写入 Kafka 的数据是会落地写入磁盘的。

我们暂且不考虑写磁盘的具体过程，先大致看看下面的图，这代表了 Kafka 的核心架构原理。



石杉的架构笔记

（2）Kafka 分布式存储架构

那么现在问题来了，如果每天产生几十 TB 的数据，难道都写一台机器的磁盘上吗？这明显是不靠谱的啊！

所以说，这里就得考虑数据的分布式存储了，其实关于消息中间件的分布式存储以及高可用架构，之前的一篇文章《面试一线互联网大厂？那这道题目你必须得会！》也分析过了，但是这里，我们结合 Kafka 的具体情况来说说。

在 Kafka 里面，有一个核心的概念叫做 “Topic”，这个 topic 你就姑且认为是一个数据集合吧。

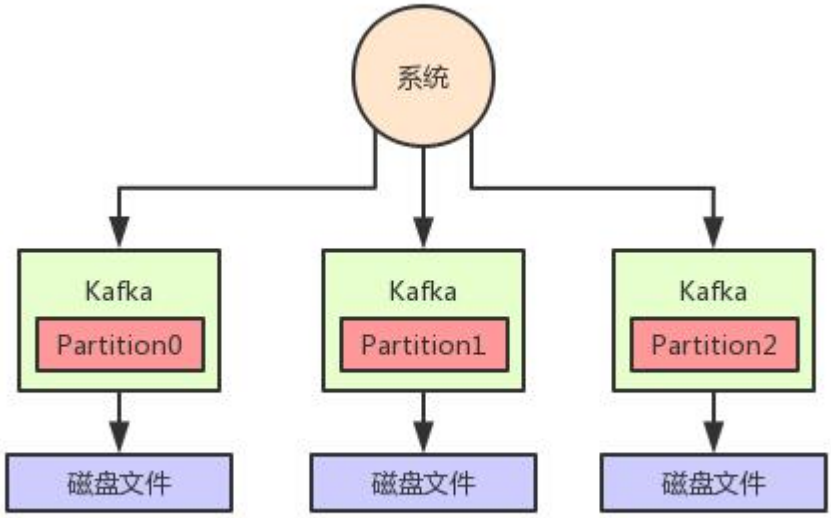
举个例子，如果你现在有一份网站的用户行为数据要写入 Kafka，你可以搞一个 topic 叫做 “user_access_log_topic”，这里写入的都是用户行为数据。

然后如果你要把电商网站的订单数据的增删改变更记录写 Kafka，那可以搞一个 topic 叫做 “order_tb_topic”，这里写入的都是订单表的变更记录。

然后假如说咱们举个例子，就说这个用户行为 topic 吧，里面如果每天写入几十 TB 的数据，你觉得都放一台机器上靠谱吗？

明显不太靠谱，所以 Kafka 有一个概念叫做 Partition，就是把一个 topic 数据集合拆分为多个数据分区，你可以认为是多个数据分片，每个 Partition 可以在不同的机器上，储存部分数据。

这样，不就可以把一个超大的数据集合分布式存储在多台机器上了吗？大家看下图，一起来体会一下。



（3）Kafka 高可用架构

但是这个时候，我们又会遇到一个问题，就是万一某台机器宕机了，这台机器上的那个 partition 管理的数据不就丢失了吗?

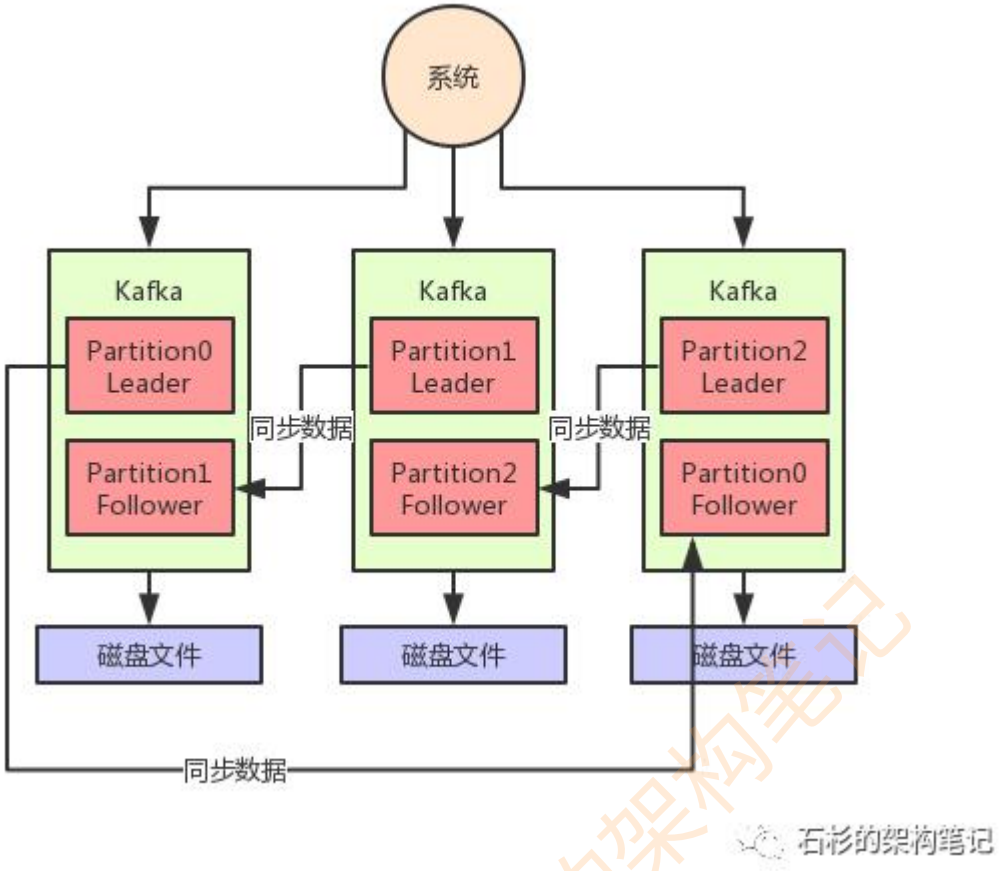
所以说，我们还得做多副本冗余，每个 Partition 都可以搞一个副本放在别的机器上，这样某台机器宕机，只不过是 Partition 其中一个副本丢失。

如果某个 Partition 有多副本的话，Kafka 会选举其中一个 Parititon 副本作为 Leader，然后其他的 Partition 副本是 Follower。

只有 Leader Partition 是对外提供读写操作的，Follower Partition 就是从 Leader Partition 同步数据。

一旦 Leader Partition 宕机了，就会选举其他的 Follower Partition 作为新的 Leader Partition 对外提供读写服务，这不就实现了高可用架构了?

大家看下面的图，看看这个过程。



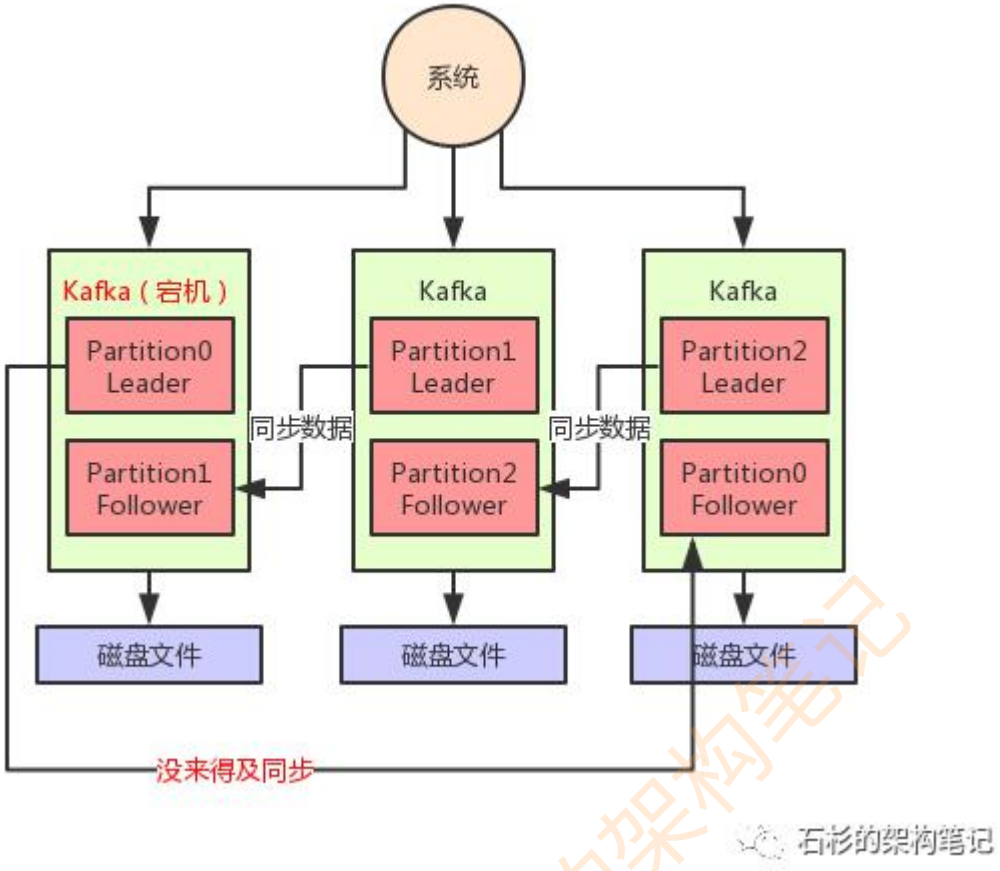
（4）Kafka 写入数据丢失问题

现在我们来看看，什么情况下 Kafka 中写入数据会丢失呢?

其实也很简单，大家都知道写入数据都是往某个 Partition 的 Leader 写入的，然后那个 Partition 的 Follower 会从 Leader 同步数据。

但是万一 1 条数据刚写入 Leader Partition，还没来得及同步给 Follower，此时 Leader Partiton 所在机器突然就宕机了呢?

大家看下图：



如上图，这个时候有一条数据是没同步到 Partition0 的 Follower 上去的，然后 Partition0 的 Leader 所在机器宕机了。

此时就会选举 Partition0 的 Follower 作为新的 Leader 对外提供服务，然后用户是不是就读不到刚才写入的那条数据了？

因为 Partition0 的 Follower 上是没有同步到最新的一条数据的。

这个时候就会造成数据丢失的问题。

（5）Kafka 的 ISR 机制是什么？

现在我们先留着这个问题不说具体怎么解决，先回过头来看一个 Kafka 的核心机制，就是 ISR 机制。

这个机制简单来说，就是会自动给每个 Partition 维护一个 ISR 列表，这个列表里一定会有 Leader，然后还会包含跟 Leader 保持同步的 Follower。

也就是说，只要 Leader 的某个 Follower 一直跟他保持数据同步，那么就会存在于 ISR 列表里。

但是如果 Follower 因为自身发生一些问题，导致不能及时的从 Leader 同步数据过去，那么这个 Follower 就会被认为是 “out-of-sync”，从 ISR 列表里踢出去。

所以大家先得明白这个 ISR 是什么，说白了，就是 Kafka 自动维护和监控哪些 Follower 及时的跟上了 Leader 的数据同步。

（6）Kafka 写入的数据如何保证不丢失?

所以如果要让写入 Kafka 的数据不丢失，你需要要求几点：

- 每个 Partition 都至少得有 1 个 Follower 在 ISR 列表里，跟上了 Leader 的数据同步
- 每次写入数据的时候，都要求至少写入 Partition Leader 成功，同时还有至少一个 ISR 里的 Follower 也写入成功，才算这个写入是成功了
- 如果不满足上述两个条件，那就一直写入失败，让生产系统不停的尝试重试，直到满足上述两个条件，然后才能认为写入成功
- 按照上述思路去配置相应的参数，才能保证写入 Kafka 的数据不会丢失

好！现在咱们来分析一下上面几点要求。

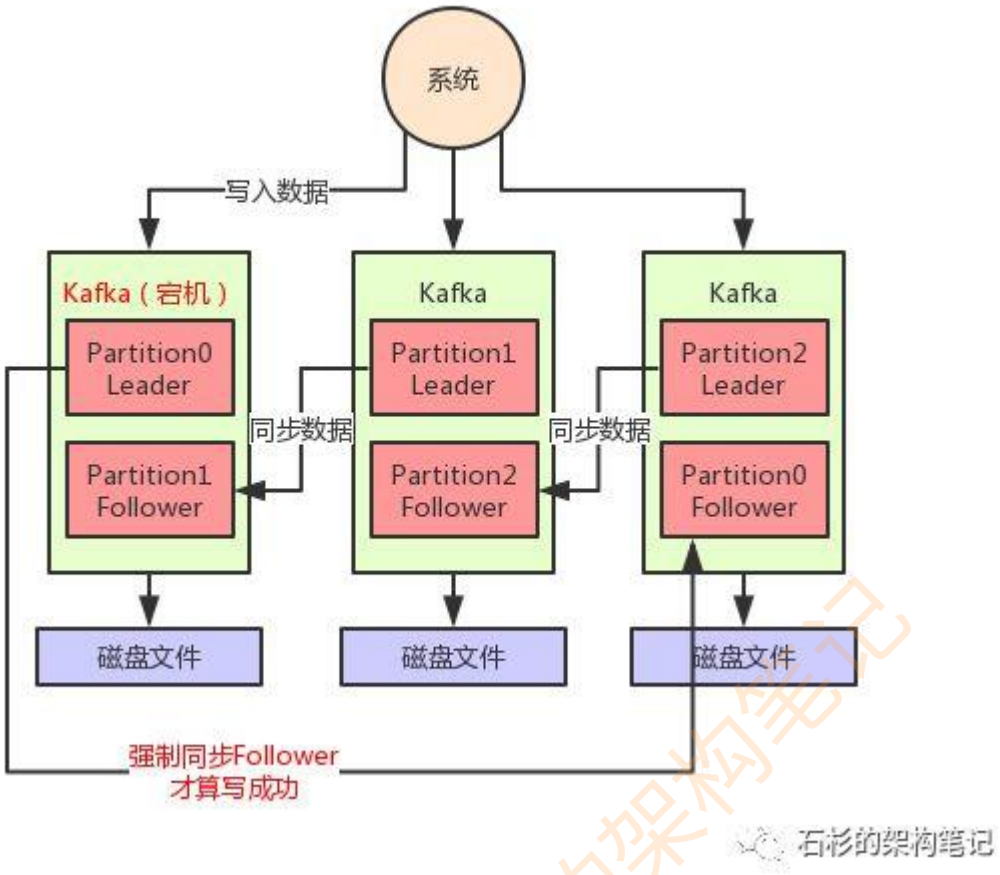
第一条，必须要求至少一个 Follower 在 ISR 列表里。

那必须的啊，要是 Leader 没有 Follower 了，或者是 Follower 都没法及时同步 Leader 数据，那么这个事儿肯定就没法弄下去了。

第二条，每次写入数据的时候，要求 leader 写入成功以外，至少一个 ISR 里的 Follower 也写成功。

大家看下面的图，这个要求就是保证说，每次写数据，必须是 leader 和 follower 都写成功了，才能算是写成功，保证一条数据必须有两个以上的副本。

这个时候万一 leader 宕机，就可以切换到那个 follower 上去，那么 Follower 上是有刚写入的数据的，此时数据就不会丢失了。



如上图所示，假如现在 leader 没有 follower 了，或者是刚写入 leader，leader 立马就宕机，还没来得及同步给 follower。

在这种情况下，写入就会失败，然后你就让生产者不停的重试，直到 kafka 恢复正常满足上述条件，才能继续写入。

这样就可以让写入 kafka 的数据不丢失。

（7）总结

最后总结一下，其实 kafka 的数据丢失问题，涉及到方方面面。

譬如生产端的缓存问题，包括消费端的问题，同时 kafka 自己内部的底层算法和机制也可能导致数据丢失。

但是平时写入数据遇到比较大的一个问题，就是 leader 切换时可能导致数据丢失。所以本文仅仅是针对这个问题说了一下生产环境解决这个问题的方案。

面试官：如果让你设计一个消息中间件，如何将其网络通信性能优化10倍以上？

作者:中华石杉 原文地址

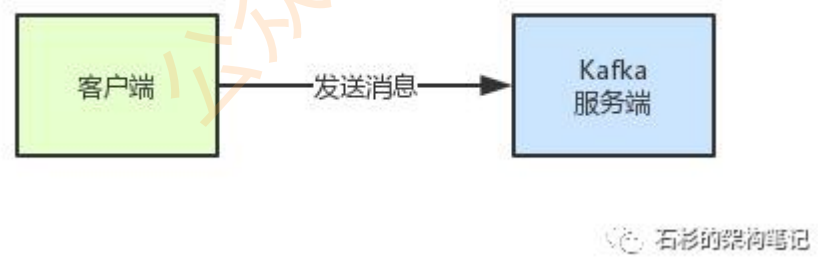
目录

> “ 这篇文章，给大家聊一个消息中间件相关的技术话题，对于一个优秀的消息中间件而言，客户端与服务端通信的时候，对于这个网络通信的机制应该如何设计，才能保证性能最优呢？甚至通过优秀的设计，让性能提升 10 倍以上。

我们本文就以 Kafka 为例来给大家分析一下，Kafka 在客户端与服务端通信的时候，底层的一些网络通信相关的机制如何设计以及如何进行优化的。

1、客户端与服务端的交互

假如我们用 kafka 作为消息中间件，势必会有客户端作为生产者向他发送消息，这个大家应该都可以理解。



对于 Kafka 来说，他本身是支持分布式的消息存储的，什么意思呢?

比如说现在你有一个 “Topic”，一个“Topic” 你就可以理解为一个消息数据的逻辑上的集合。

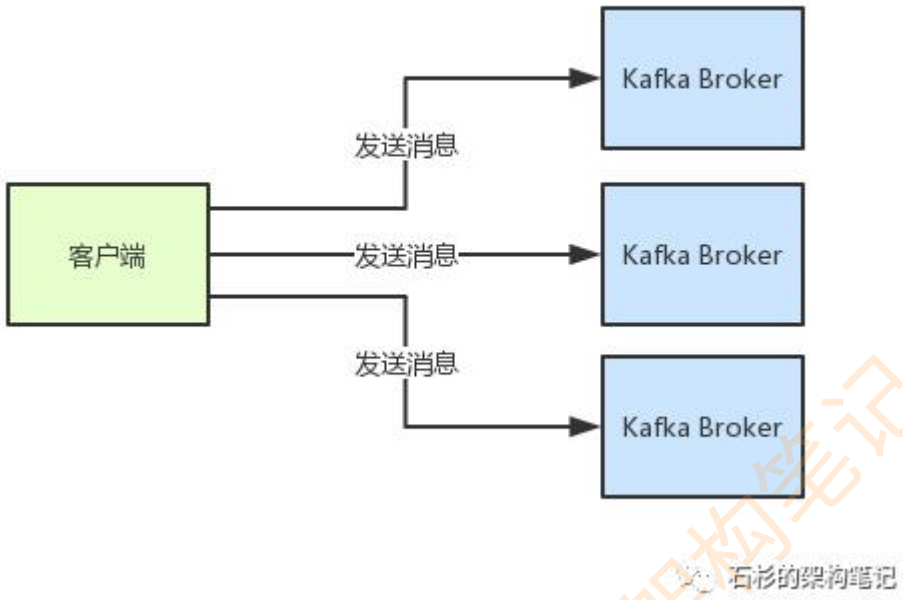
比如现在你要把所有的订单数据都发送到一个 “Topic” 里去，那么这个 “Topic” 就叫做“OrderTopic”，里面都放的是订单数据。

接着这个 “Topic” 的数据可能量很大很大，不可能放在一台机器上吧?

所以呢，我们就可以分散存储在多台 Kafka 的机器上，每台机器存储一部分的数据即可。

这就是 Kafka 的分布式消息存储的机制，每个 Kafka 服务端叫做一个 Broker，负责管理一台机器上的数据。

一起来看看下面的图：

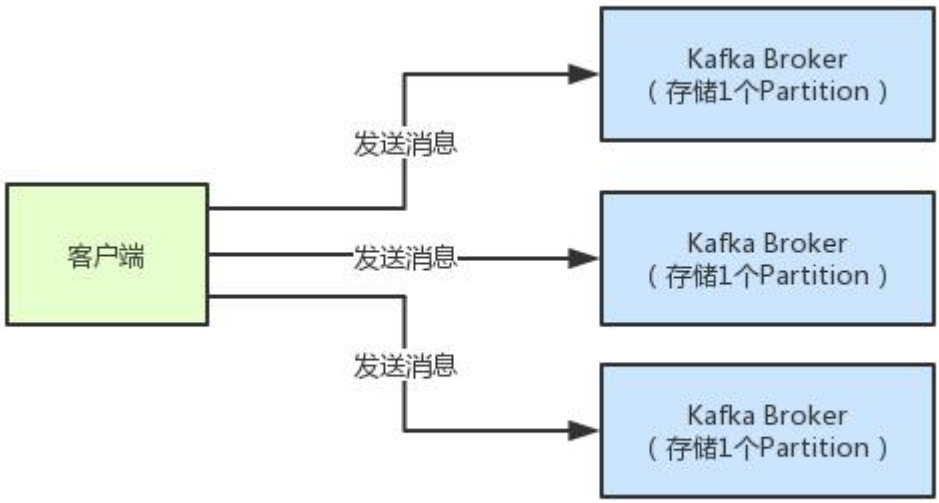


一个 “Topic” 可以拆分为多个 “Partition”，每个“Partition” 存储一部分数据，每个 Partition 都可以放在不同的 Kafka Broker 机器上，这样就实现了数据分散存储在多台机器上的效果了。

然后客户端在发送消息到 Kafka Broker 的时候，比如说你限定了 “OrderTopic” 的订单数据拆分为 3 个 “Partition”，那么 3 个“Partition” 分别放在一个 Kafka Broker 上，那么也就是要把所有的订单数据分发到三个 Kafka Broker 上去。

此时就会默认情况下走一个负载均衡的策略，举个例子，假设订单数据一共有 3 万条，就会给每个 Partition 分发 1 万条订单消息，这样订单数据均匀分散在了 3 台 Broker 机器上。

整个过程，如下图所示：



2、频繁网络通信带来的性能低下问题

好了，现在问题来了，客户端在发送消息给 Kafka Broker 的时候，比如说现在要发送一个订单到 Kafka 上去，此时他是怎么发送过去呢?

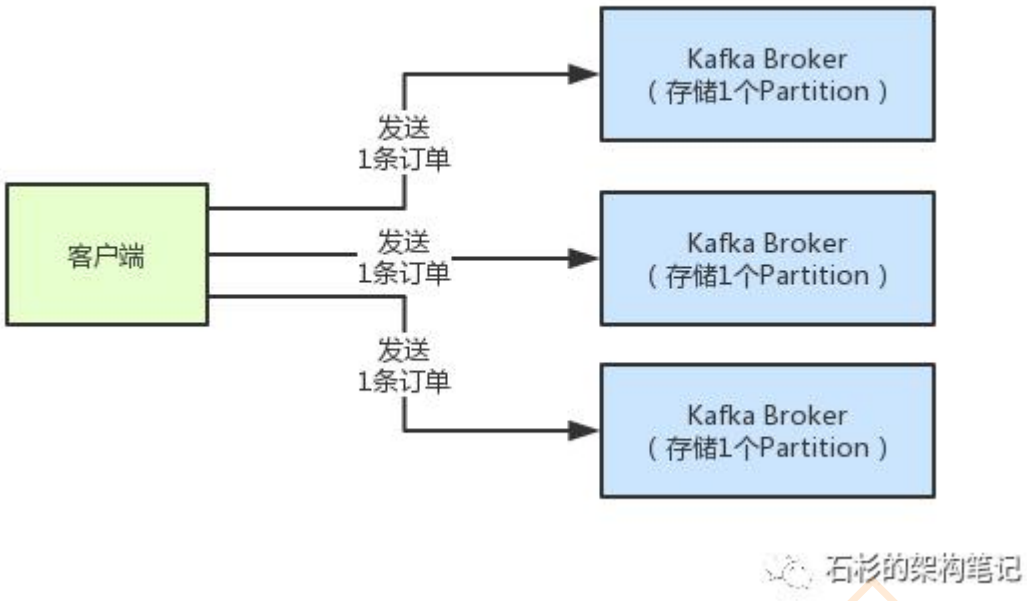
是直接一条订单消息就对应一个网络请求，发送到一台 Broker 上去吗?

如果是这样做的话，那势必会导致频繁的跟一台 broker 进行网络通信，频繁的网络通信，每次都涉及到复杂的网络连接、传输的流程，那么进而会导致客户端性能的低下。

给大家举个例子，比如说每次通过一个网络通信发送一条订单到 broker，需要耗时 10ms。

那么如果一个订单就一次网络通信发送到 broker，每秒最多就是发送 100 个订单了，大家想想，是不是这个道理?

但是假如说你每秒有 10000 个订单要发送，此时就会造成你的发送性能远远跟不上你的需求，也就是性能的低下，看起来你的系统发送订单到 kafka 的速度就是特别的慢。



3、batch 机制：多条消息打包成一个 batch

所以首先针对这个问题，kafka 做的第一个优化，就是实现了 batch 机制。

这个意思就是说，他会在客户端放一个内存缓冲区，每次你写一条订单先放到内存缓冲区里去，然后在内存缓冲区里，会把多个订单给打包起来成为一个 batch。

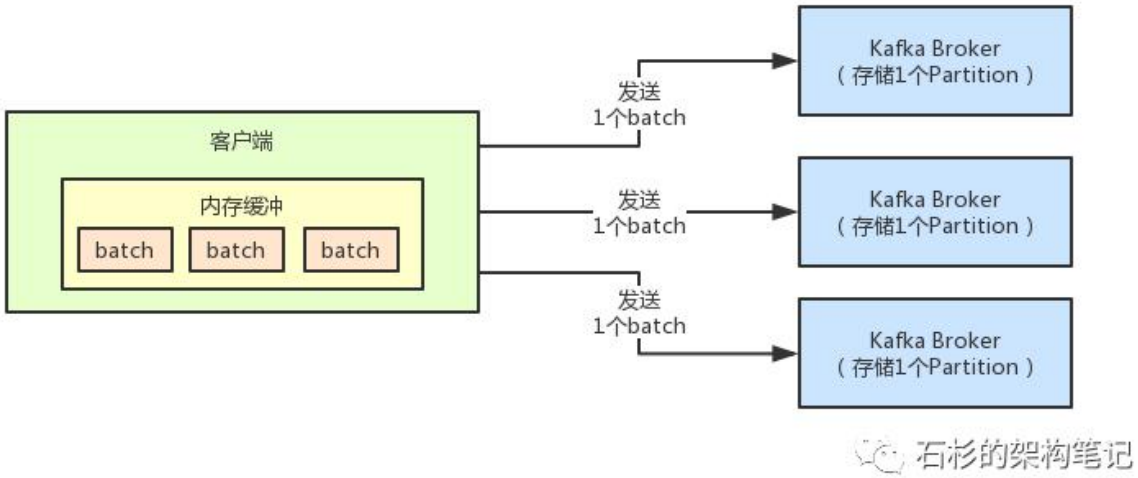
比如说默认 kafka 规定的 batch 的大小是 16kb，那么意思就是，你默认就是多条订单凑满 16kb 的大小，就会成为一个 batch，然后他就会把这个 batch 通过网络通信发送到 broker 上去。

假如说一个 batch 发送到 broker，同样也是耗费 10ms 而已，但是一个 batch 里可以放入 100 条订单，那么 1 秒是不是可以发送 100 个 batch?

此时，1 秒是不是就可以发送 10000 条订单出去了?

而且在打包消息形成 batch 的时候，是有讲究的，你必须是发送到同一个 Topic 的同一个 Partition 的消息，才会进入一个 batch。

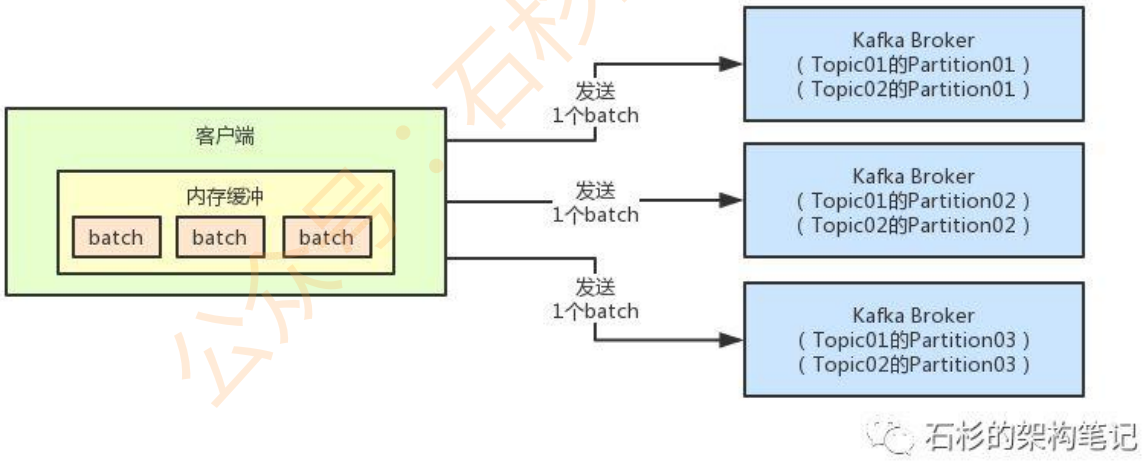
这个 batch 里就代表要发送到同一个 Partition 的多条消息，这样后续才能通过一个网络请求，就把这个 batch 发送到 broker，对应写入一个 Parititon 中。



4、request 机制：多个 batch 打包成一个 request

事情到这里就结束了吗？还没有！

比如现在我们要是手头有两个 Topic，每个 Topic 都有 3 个 Partition，那么每个 Broker 是不是就会存放 2 个 Partition？其中 1 个 Partition 是 Topic01 的，1 个 Partition 是 Topic02 的。



现在假如说针对 Topic01 的 Partition02 形成了一个 batch，针对 Topic02 的 Partition02 也形成了一个 batch，但是这两个 batch 其实都是发往同一个 Broker 的，如上图的第二个 Broker。

此时，还是一个网络请求发送一个 batch 过去吗？

其实就完全没必要了，完全此时可以把多个发往同一个 Broker 的 batch 打包成一个 request，然后一个 request 通过一次网络通信发送到 那个 Broker 上去。

假设一次网络通信还是 10ms，那么这一次网络通信就发送了 2 个 batch 过去。

通过这种多个 batch 打包成一个 request 一次性发往 Broker 的方式，又进一步提升了网络通信的效率和性能。

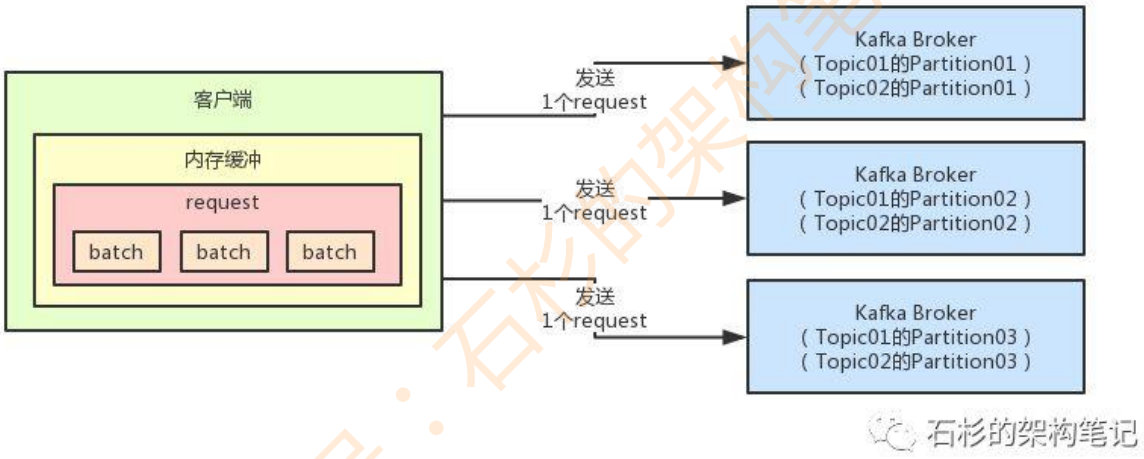
其实 batch 机制 + request 机制，都是想办法把很多数据打包起来，然后一次网络通信尽量多发送一些数据出去，这样可以提升单位时间内发送数据的数量。

这个单位时间内发送数据的数量，也就是所谓的 “吞吐量”，也就是单位时间内可以发送多少数据到 broker 上去。

比如说每秒钟可以发送 3 万条消息过去，这就是代表了客户端的 “吞吐量” 有多大。

因此，通过搞清楚这个原理，就可以学习到这种非常优秀的设计思想。而且在面试的时候，如果跟面试官聊到 kafka，也可以跟面试官侃侃 kafka 底层，是如何有效的提升网络通信性能的。

最后再来一张图，作为全文总结。



简历写了会Kafka，面试官90%会让你讲讲acks参数对消息持久化的影响

> **作者:中华石杉** 原文地址

目录

- （4）ISR 到底指的什么东西?
- （5）acks 参数的含义?
- （6）最后的思考

（0）写在前面

面试大厂时，一旦简历上写了 Kafka，几乎必然会被问到一个问题：说说 acks 参数对消息持久化的影响?

这个 acks 参数在 kafka 的使用中，是非常核心以及关键的一个参数，决定了很多东西。

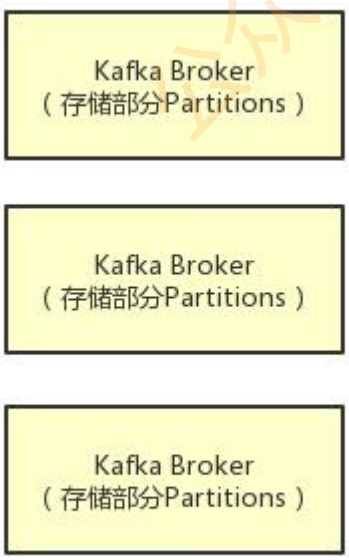
所以无论是为了面试还是实际项目使用，大家都值得看一下这篇文章对 Kafka 的 acks 参数的分析，以及背后的原理。

（1）如何保证宕机的时候数据不丢失?

如果要想理解这个 acks 参数的含义，首先就得搞明白 kafka 的高可用架构原理。

比如下面的图里就是表明了对于每一个 Topic，我们都可以设置他包含几个 Partition，每个 Partition 负责存储这个 Topic 一部分的数据。

然后 Kafka 的 Broker 集群中，每台机器上都存储了一些 Partition，也就存放了 Topic 的一部分数据，这样就实现了 Topic 的数据分布式存储在一个 Broker 集群上。



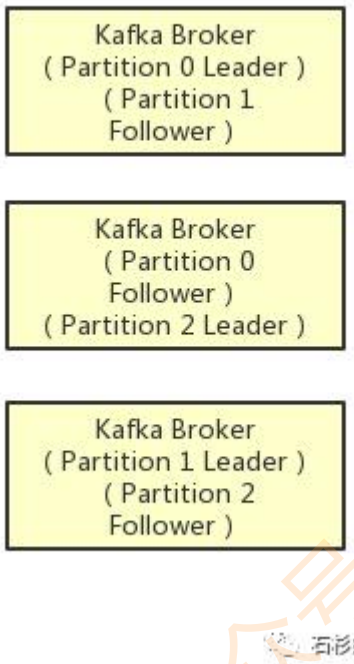
但是有一个问题，万一 一个 Kafka Broker 宕机了，此时上面存储的数据不就丢失了吗?

没错，这就是一个比较大的问题了，分布式系统的数据丢失问题，是他首先必须要解决的，一旦说任何一台机器宕机，此时就会导致数据的丢失。

（2）多副本冗余的高可用机制

所以如果大家去分析任何一个分布式系统的原理，比如说 zookeeper、kafka、redis cluster、elasticsearch、hdfs，等等，其实他都有自己内部的一套多副本冗余的机制，多副本冗余几乎是现在任何一个优秀的分布式系统都一般要具备的功能。

在 kafka 集群中，每个 Partition 都有多个副本，其中一个副本叫做 leader，其他的副本叫做 follower，如下图。



如上图所示，假设一个 Topic 拆分为了 3 个 Partition，分别是 Partition0，Partiton1，Partition2，此时每个 Partition 都有 2 个副本。

比如 Partition0 有一个副本是 Leader，另外一个副本是 Follower，Leader 和 Follower 两个副本是分布在不同机器上的。

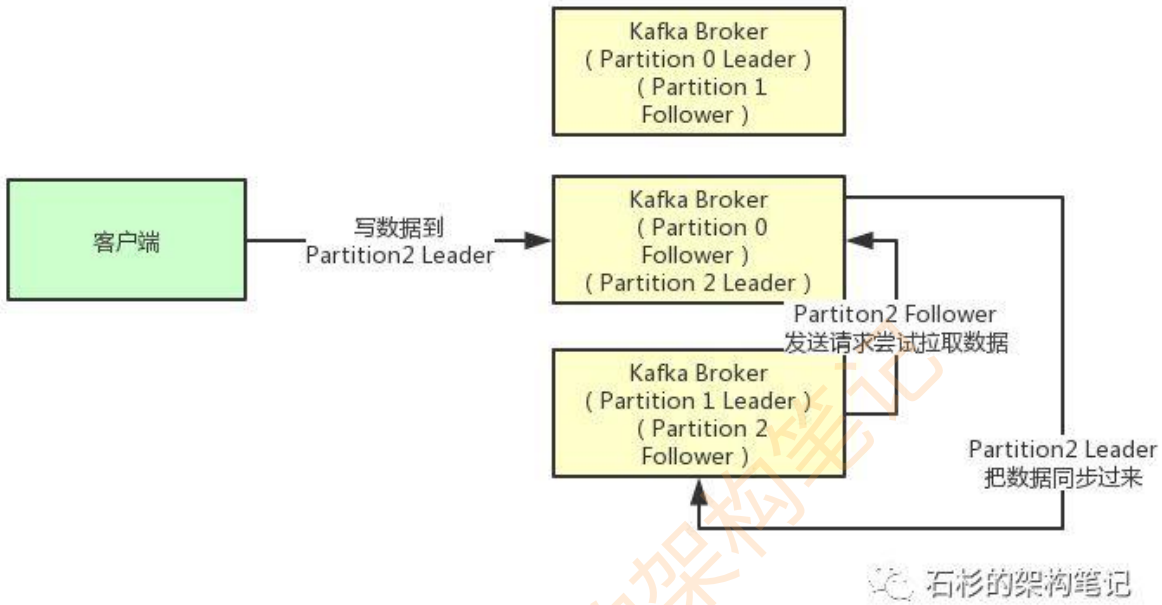
这样的多副本冗余机制，可以保证任何一台机器挂掉，都不会导致数据彻底丢失，因为起码还是有副本在别的机器上的。

（3）多副本之间数据如何同步？

接着我们就来看看多个副本之间数据是如何同步的？其实任何一个 Partition，只有 Leader 是对外提供读写服务的

也就是说，如果有一个客户端往一个 Partition 写入数据，此时一般就是写入这个 Partition 的 Leader 副本。

然后 Leader 副本接收到数据之后，Follower 副本会不停的给他发送请求尝试去拉取最新的数据，拉取到自己本地后，写入磁盘中。如下图所示：



（4）ISR 到底指的是什么东西?

既然大家已经知道了 Partiton 的多副本同步数据的机制了，那么就可以来看看 ISR 是什么了。

ISR 全称是 “In-Sync Replicas”，也就是保持同步的副本，他的含义就是，跟 Leader 始终保持同步的 Follower 有哪些。

大家可以想一下，如果说某个 Follower 所在的 Broker 因为 JVM FullGC 之类的问题，导致自己卡顿了，无法及时从 Leader 拉取同步数据，那么是不是会导致 Follower 的数据比 Leader 要落后很多?

所以这个时候，就意味着 Follower 已经跟 Leader 不再处于同步的关系了。但是只要 Follower 一直及时从 Leader 同步数据，就可以保证他们是处于同步的关系的。

所以每个 Partition 都有一个 ISR，这个 ISR 里一定会有 Leader 自己，因为 Leader 肯定数据是最新的，然后就是那些跟 Leader 保持同步的 Follower，也会在 ISR 里。

（5）acks 参数的含义

铺垫了那么多的东西，最后终于可以进入主题来聊一下 acks 参数的含义了。

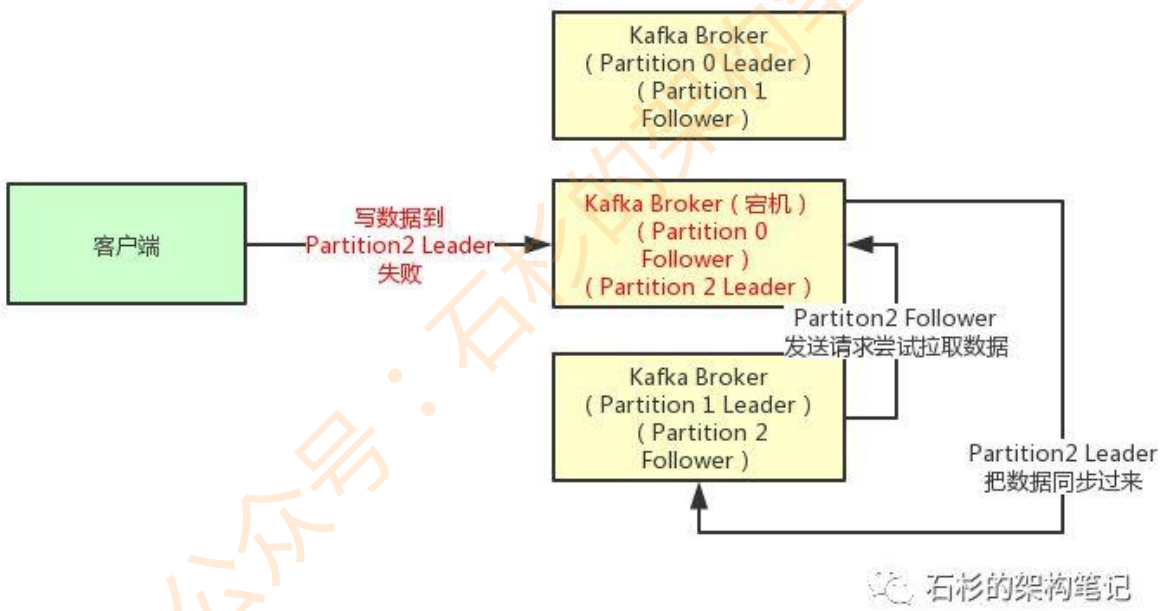
如果大家没看明白前面的那些副本机制、同步机制、ISR 机制，那么就无法充分的理解 acks 参数的含义，这个参数实际上决定了很多重要的东西。

首先这个 acks 参数，是在 KafkaProducer，也就是生产者客户端里设置的

也就是说，你往 kafka 写数据的时候，就可以来设置这个 acks 参数。然后这个参数实际上有三种常见的值可以设置，分别是：0、1 和 all。

第一种选择是把 acks 参数设置为 0，意思就是我的 KafkaProducer 在客户端，只要把消息发送出去，不管那条数据有没有在哪怕 Partition Leader 上落到磁盘，我就不管他了，直接就认为这个消息发送成功了。

如果你采用这种设置的话，那么你必须注意的一点是，可能你发送出去的消息还在半路。结果呢，Partition Leader 所在 Broker 就直接挂了，然后结果你的客户端还认为消息发送成功了，此时就会导致这条消息就丢失了。

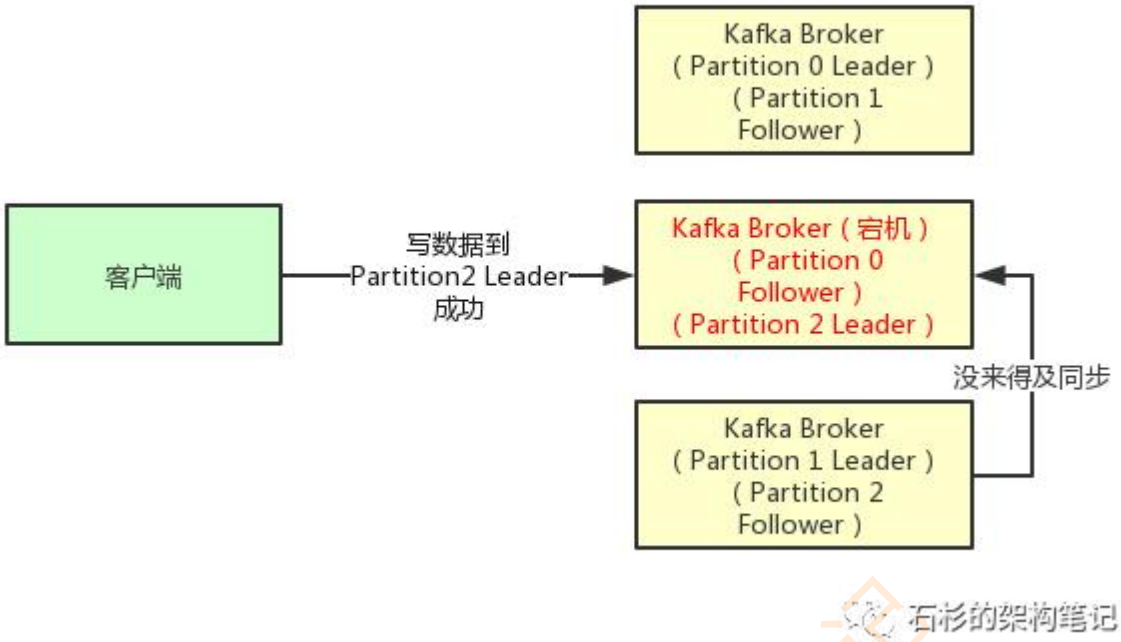


第二种选择是设置 acks = 1，意思就是说只要 Partition Leader 接收到消息而且写入本地磁盘了，就认为成功了，不管他其他的 Follower 有没有同步过去这条消息了。

这种设置其实是 kafka 默认的设置，大家请注意，划重点！这是默认的设置

也就是说，默认情况下，你要是不管 acks 这个参数，只要 Partition Leader 写成功就算成功。

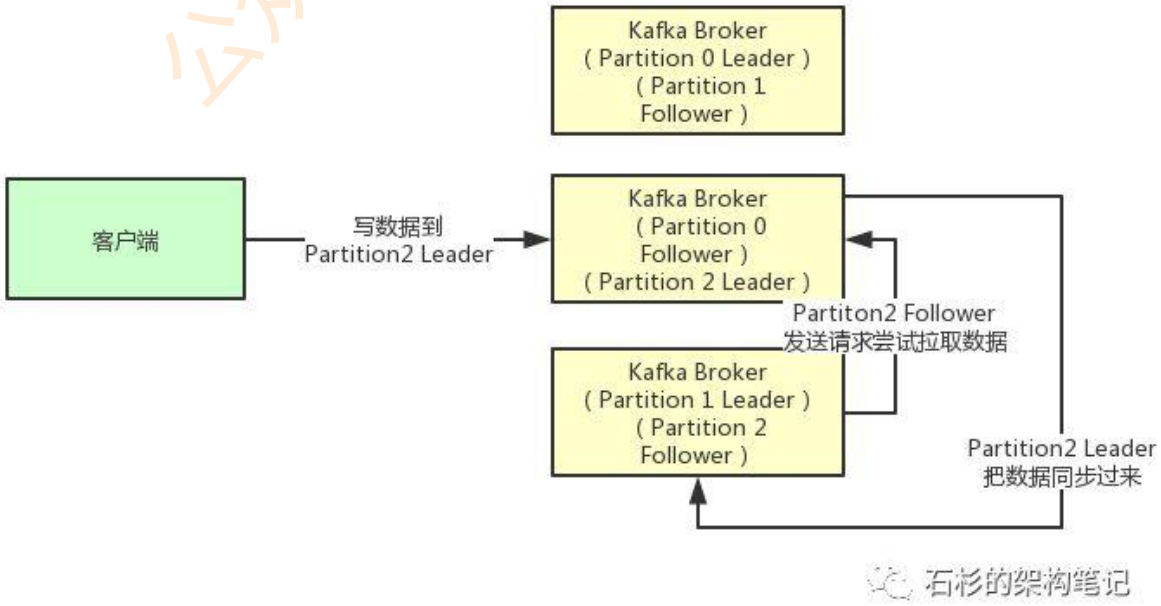
但是这里有一个问题，万一 Partition Leader 刚刚接收到消息，Follower 还没来得及同步过去，结果 Leader 所在的 broker 宕机了，此时也会导致这条消息丢失，因为人家客户端已经认为发送成功了。



最后一种情况，就是设置 acks=all，这个意思就是说，Partition Leader 接收到消息之后，还必须要求 ISR 列表里跟 Leader 保持同步的那些 Follower 都要把消息同步过去，才能认为这条消息是写入成功了。

如果说 Partition Leader 刚接收到了消息，但是结果 Follower 没有收到消息，此时 Leader 宕机了，那么客户端会感知到这个消息没发送成功，他会重试再次发送消息过去。

此时可能 Partition 2 的 Follower 变成 Leader 了，此时 ISR 列表里只有最新的这个 Follower 转变成的 Leader 了，那么只要这个新的 Leader 接收消息就算成功了。



（6）最后的思考

acks=all 就可以代表数据一定不会丢失了吗?

当然不是，如果你的 Partition 只有一个副本，也就是一个 Leader，任何 Follower 都没有，你认为 acks=all 有用吗?

当然没用了，因为 ISR 里就一个 Leader，他接收完消息后宕机，也会导致数据丢失。

所以说，这个 acks=all，必须跟 ISR 列表里至少有 2 个以上的副本配合使用，起码是有一个 Leader 和一个 Follower 才可以。

这样才能保证说写一条数据过去，一定是 2 个以上的副本都收到了才算是成功，此时任何一个副本宕机，不会导致数据丢失。

所以希望大家把这篇文章好好理解一下，对大家出去面试，或者工作中用 kafka 都是很好的一个帮助。

不了解这些“高级货”，活该你面试当炮灰。。。

> **作者:中华石杉** 原文地址

目录

> ! “ 今天聊一个非常硬核的技术知识，给大家分析一下 CopyOnWrite 思想是什么，以及在 Java 并发包中的具体体现，包括在 Kafka 内核源码中是如何运用这个思想来优化并发性能的。 这个 CopyOnWrite 在面试的时候，很可能成为面试官的一个杀手锏把候选人给一击必杀，也很有可能成为候选人拿下 Offer 的独门秘籍，是相对高级的一个知识。

1、读多写少的场景下引发的问题?

大家可以设想一下现在我们的内存里有一个 ArrayList，这个 ArrayList 默认情况下肯定是线程不安全的，要是多个线程并发读和写这个 ArrayList 可能会有问题。

好，问题来了，我们应该怎么让这个 ArrayList 变成线程安全的呢?

有一个非常简单的办法，对这个 ArrayList 的访问都加上线程同步的控制。

比如说一定要在 synchronized 代码段来对这个 ArrayList 进行访问，这样的话，就能同一时间就让一个线程来操作它了，或者是用 ReadWriteLock 读写锁的方式来控制，都可以。

我们假设就是用 ReadWriteLock 读写锁的方式来控制对这个 ArrayList 的访问。

这样多个读请求可以同时执行从 ArrayList 里读取数据，但是读请求和写请求之间互斥，写请求和写请求也是互斥的。

大家看看，代码大概就是类似下面这样：

```php
public Object  read() {
    lock.readLock().lock();
    // 对ArrayList读取
    lock.readLock().unlock();
}
public void write() {
    lock.writeLock().lock();
    // 对ArrayList写
    lock.writeLock().unlock();
}
```

大家想想，类似上面的代码有什么问题呢?

最大的问题，其实就在于写锁和读锁的互斥。假设写操作频率很低，读操作频率很高，是写少读多的场景。

那么偶尔执行一个写操作的时候，是不是会加上写锁，此时大量的读操作过来是不是就会被阻塞住，无法执行?

这个就是读写锁可能遇到的最大的问题。

2、引入 CopyOnWrite 思想解决问题

这个时候就要引入 CopyOnWrite 思想来解决问题了。

他的思想就是，不用加什么读写锁，锁统统给我去掉，有锁就有问题，有锁就有互斥，有锁就可能导致性能低下，你阻塞我的请求，导致我的请求都卡着不能执行。

那么他怎么保证多线程并发的安全性呢?

很简单，顾名思义，利用 “CopyOnWrite” 的方式，这个英语翻译成中文，大概就是“写数据的时候利用拷贝的副本来执行”。

你在读数据的时候，其实不加锁也没关系，大家左右都是一个读罢了，互相没影响。

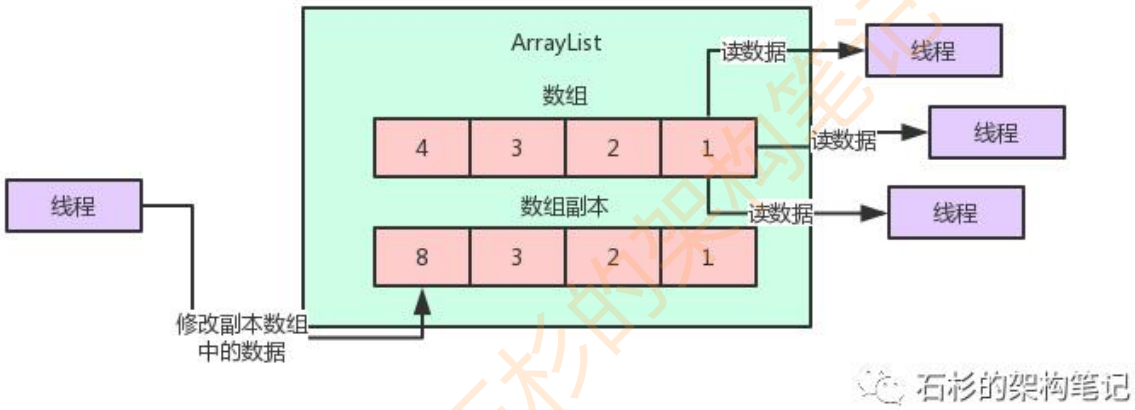
问题主要是在写的时候，写的时候你既然不能加锁了，那么就得采用一个策略。

假如说你的 ArrayList 底层是一个数组来存放你的列表数据，那么这时比如你要修改这个数组里的数据，你就必须先拷贝这个数组的一个副本。

然后你可以在这个数组的副本里写入你要修改的数据，但是在这个过程中实际上你都是在操作一个副本而已。

这样的话，读操作是不是可以同时正常的执行？这个写操作对读操作是没有任何的影响的吧！

大家看下面的图，一起来体会一下这个过程：



关键问题来了，那那个写线程现在把副本数组给修改完了，现在怎么才能让读线程感知到这个变化呢？

关键点来了，划重点！这里要配合上 volatile 关键字的使用。

笔者之前写过文章，给大家解释过 volatile 关键字的使用，核心就是让一个变量被写线程给修改之后，立马让其他线程可以读到这个变量引用的最近的值，这就是 volatile 最核心的作用。

所以一旦写线程搞定了副本数组的修改之后，那么就可以用 volatile 写的方式，把这个副本数组赋值给 volatile 修饰的那个数组的引用变量了。

只要一赋值给那个 volatile 修饰的变量，立马就会对读线程可见，大家都能看到最新的数组了。

下面是 JDK 里的 CopyOnWriteArrayList 的源码。

大家看看写数据的时候，他是怎么拷贝一个数组副本，然后修改副本，接着通过 volatile 变量赋值的方式，把修改好的数组副本给更新回去，立马让其他线程可见的。

```
// 这个数组是核心的，因为用volatile修饰了
    // 只要把最新的数组对他赋值，其他线程立马可以看到最新的数组
    private transient volatile Object[] array;
    public boolean add(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            int len = elements.length;
            // 对数组拷贝一个副本出来
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            // 对副本数组进行修改，比如在里面加入一个元素
            newElements[len] = e;
            // 然后把副本数组赋值给volatile修饰的变量
            setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }
```

然后大家想，因为是通过副本来进行更新的，万一要是多个线程都要同时更新呢？那搞出来多个副本会不会有问题？

当然不能多个线程同时更新了，这个时候就是看上面源码里，加入了 lock 锁的机制，也就是同一时间只有一个线程可以更新。

那么更新的时候，会对读操作有任何的影响吗？

绝对不会，因为读操作就是非常简单的对那个数组进行读而已，不涉及任何的锁。而且只要他更新完毕对 volatile 修饰的变量赋值，那么读线程立马可以看到最新修改后的数组，这是 volatile 保证的。


```
    private E get(Object[] a, int index) {
        // 最简单的对数组进行读取
        return (E) a[index];
    }
```

这样就完美解决了我们之前说的读多写少的问题。

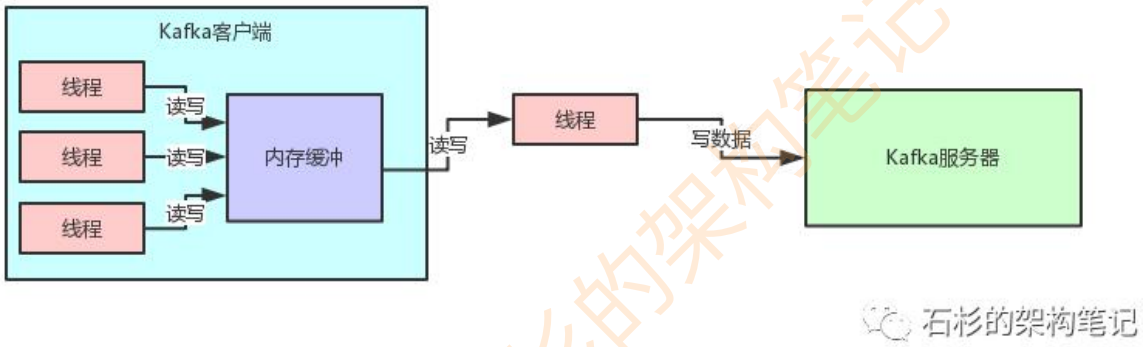
如果用读写锁互斥的话，会导致写锁阻塞大量读操作，影响并发性能。

但是如果用了 CopyOnWriteArrayList，就是用空间换时间，更新的时候基于副本更新，避免锁，然后最后用 volatile 变量来赋值保证可见性，更新的时候对读线程没有任何的影响！

3、CopyOnWrite 思想在 Kafka 源码中的运用

在 Kafka 的内核源码中，有这么一个场景，客户端在向 Kafka 写数据的时候，会把消息先写入客户端本地的内存缓冲，然后在内存缓冲里形成一个 Batch 之后再一次性发送到 Kafka 服务器上去，这样有助于提升吞吐量。

话不多说，大家看下图：



这个时候 Kafka 的内存缓冲用的是什么数据结构呢？大家看源码：

```php
private final ConcurrentMap<topicpartition, deque<="" span="">
        batches = new CopyOnWriteMap<TopicPartition, Deque>();
```

这个数据结构就是核心的用来存放写入内存缓冲中的消息的数据结构，要看懂这个数据结构需要对很多 Kafka 内核源码里的概念进行解释，这里先不展开。

但是大家关注一点，他是自己实现了一个 CopyOnWriteMap，这个 CopyOnWriteMap 采用的就是 CopyOnWrite 思想。

我们来看一下这个 CopyOnWriteMap 的源码实现：

```php
// 典型的volatile修饰普通Map
private volatile Mapmap;
@Override
public synchronized V put(K k, V v) {
    // 更新的时候先创建副本，更新副本，然后对volatile变量赋值写回去
```

```
        Mapcopy = new HashMap(this.map);
        V prev = copy.put(k, v);
        this.map = Collections.unmodifiableMap(copy);
        return prev;
    }
    @Override
    public V get(Object k) {
        // 读取的时候直接读volatile变量引用的map数据结构，无需锁
        return map.get(k);
    }
```

所以 Kafka 这个核心数据结构在这里之所以采用 CopyOnWriteMap 思想来实现，就是因为这个 Map 的 key-value 对，其实没那么频繁更新。

也就是 TopicPartition-Deque 这个 key-value 对，更新频率很低。

但是他的 get 操作却是高频的读取请求，因为会高频的读取出来一个 TopicPartition 对应的 Deque 数据结构，来对这个队列进行入队出队等操作，所以对于这个 map 而言，高频的是其 get 操作。

这个时候，Kafka 就采用了 CopyOnWrite 思想来实现这个 Map，避免更新 key-value 的时候阻塞住高频的读操作，实现无锁的效果，优化线程并发的性能。

相信大家看完这个文章，对于 CopyOnWrite 思想以及适用场景，包括 JDK 中的实现，以及在 Kafka 源码中的运用，都有了一个切身的体会了。

如果你能在面试时说清楚这个思想以及他在 JDK 中的体现，并且还能结合知名的开源项目 Kafka 的底层源码进一步向面试官进行阐述，面试官对你的印象肯定大大的加分。

【架构设计的艺术】Kafka如何通过精妙的架构设计优化JVM GC问题?

> **作者:中华石杉** 原文地址

目录

- 4、总结一下

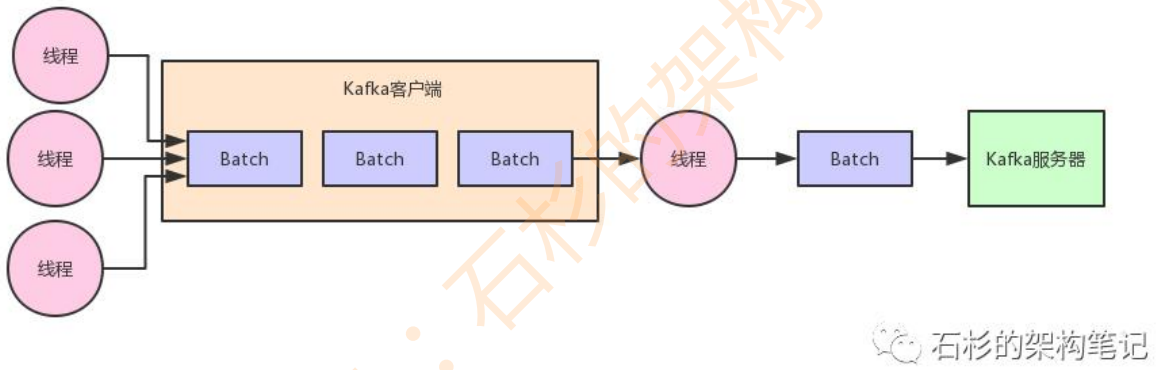
> “ 这篇文章，同样给大家聊一个硬核的技术知识，我们通过 Kafka 内核源码中的一些设计思想，来看你设计 Kafka 架构的技术大牛，是怎么优化 JVM 的 GC 问题的?

1、Kafka 的客户端缓冲机制

首先，先得给大家明确一个事情，那就是在客户端发送消息给 kafka 服务器的时候，一定是有一个内存缓冲机制的。

也就是说，消息会先写入一个内存缓冲中，然后直到多条消息组成了一个 Batch，才会一次网络通信把 Batch 发送过去。

整个过程如下图所示：



2、内存缓冲造成的频繁 GC 问题

那么这种内存缓冲机制的本意，其实就是把多条消息组成一个 Batch，一次网络请求就是一个 Batch 或者多个 Batch。

这样每次网络请求都可以发送很多数据过去，避免了一条消息一次网络请求。从而提升了吞吐量，即单位时间内发送的数据量。

但是问题来了，大家可以思考一下，一个 Batch 中的数据，会取出来然后封装在底层的网络包里，通过网络发送出去到达 Kafka 服务器。

那么然后呢？这个 Batch 里的数据都发送过去了，现在 Batch 里的数据应该怎么处理?

你要知道，这些 Batch 里的数据此时可还在客户端的 JVM 的内存里啊！那么此时从代码实现层面，一定会尝试避免任何变量去引用这些 Batch 对应的数据，然后尝试触发 JVM 自动回收掉这些内存垃圾。

这样不断的让 JVM 回收垃圾，就可以不断的清理掉已经发送成功的 Batch 了，然后就可以不断的腾出来新的内存空间让后面新的数据来使用。

这种想法很好，但是实际线上运行的时候一定会有问题，最大的问题，就是 JVM GC 问题。

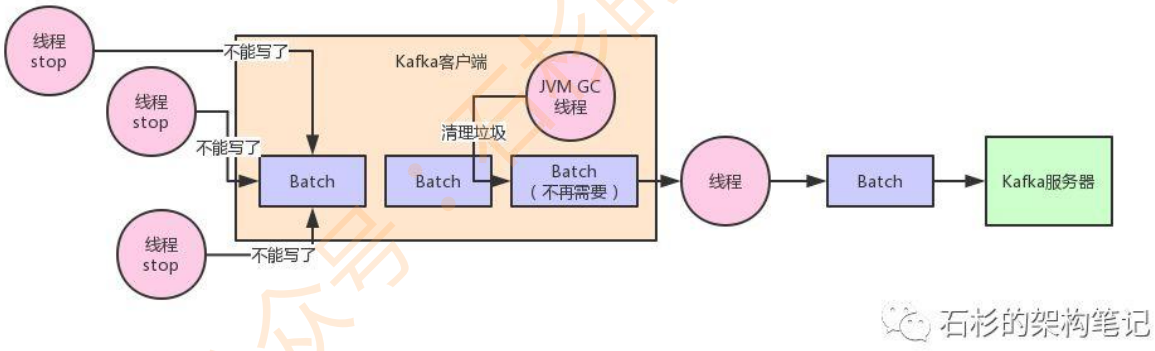
大家都知道一点，JVM GC 在回收内存垃圾的时候，他会有一个 “Stop the World” 的过程，也就是垃圾回收线程运行的时候，会导致其他工作线程短暂的停顿，这样可以便于他自己安安静静的回收内存垃圾。

这个也很容易想明白，毕竟你要是在回收内存垃圾的时候，你的工作线程还在不断的往内存里写数据，制造更多的内存垃圾，那你让人家 JVM 怎么回收垃圾？

这就好比在大马路上，如果地上有很多垃圾，现在要把垃圾都扫干净，最好的办法是什么？大家都让开，把马路空出来，然后清洁工就是把垃圾清理干净。

但是如果清洁工在清扫垃圾的时候，结果一帮人在旁边不停的嗑瓜子扔瓜子壳，吃西瓜扔西瓜皮，不停的制造垃圾，你觉得清洁工内心啥感受？当然是很愤慨了，照这么搞，地上的垃圾永远的都搞不干净了！

通过了上面的语言描述，我们再来一张图，大家看看就更加清楚了



现在 JVM GC 是越来越先进，从 CMS 垃圾回收器到 G1 垃圾回收器，核心的目标之一就是不断的缩减垃圾回收的时候，导致其他工作线程停顿的时间。

所以现在越是新款的垃圾回收器导致工作线程停顿的时间越短，但是再怎么短，他也还是存在啊！

所以说，如何尽可能在自己的设计上避免 JVM 频繁的 GC 就是一个非常考验水平的事儿了。

3、Kafka 设计者实现的缓冲池机制

在 Kafka 客户端内部，对这个问题实现了一个非常优秀的机制，就是缓冲池的机制

简单来说，就是每个 Batch 底层都对应一块内存空间，这个内存空间就是专门用来存放写入进去的消息的。

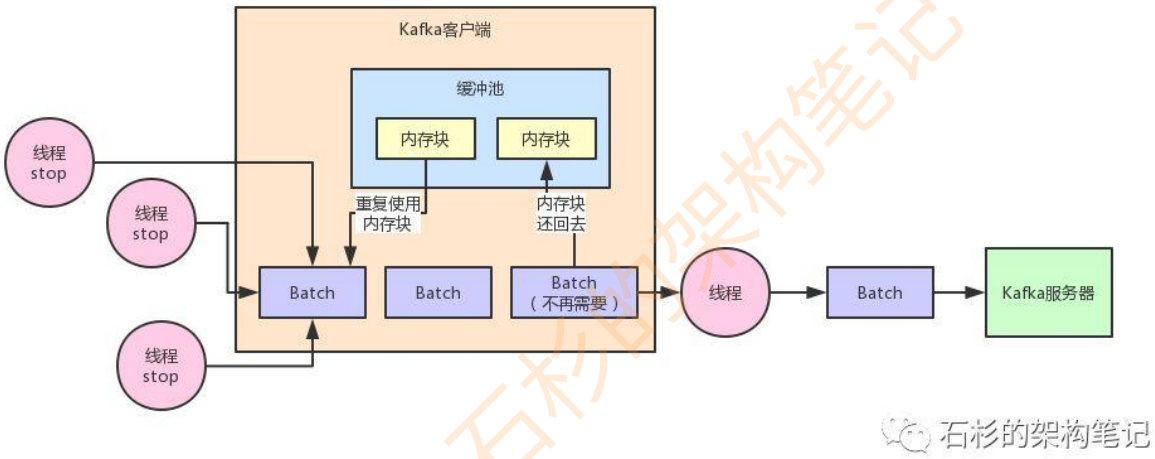
然后呢，当一个 Batch 被发送到了 kafka 服务器，这个 Batch 的数据不再需要了，就意味着这个 Batch 的内存空间不再使用了。

此时这个 Batch 底层的内存空间不要交给 JVM 去垃圾回收，而是把这块内存空间给放入一个缓冲池里。

这个缓冲池里放了很多块内存空间，下次如果你又有一个新的 Batch 了，那么不就可以直接从这个缓冲池里获取一块内存空间就 ok 了？

然后如果一个 Batch 发送出去了之后，再把内存空间给人家还回来不就好了？以此类推，循环往复。

同样，听完了上面的文字描述，再来一张图，看完这张图相信大伙儿就明白了：



一旦使用了这个缓冲池机制之后，就不涉及到频繁的大量内存的 GC 问题了。

为什么呢？因为他可以上来就占用固定的内存，比如 32MB。然后把 32MB 划分为 N 多个内存块，比如说一个内存块是 16KB，这样的话这个缓冲池里就会有很多的内存块。

然后你需要创建一个新的 Batch，就从缓冲池里取一个 16KB 的内存块就可以了，然后这个 Batch 就不断的写入消息，但是最多就是写 16KB，因为 Batch 底层的内存块就 16KB。

接着如果 Batch 被发送到 Kafka 服务器了，此时 Batch 底层的内存块就直接还回缓冲池就可以了。

下次别人再要构建一个 Batch 的时候，再次使用缓冲池里的内存块就好了。这样就可以利用有限的内存，对他不停的反复重复的利用。因为如果你的 Batch 使用完了以后是把内存块还回到缓冲池中去，那么就不涉及到垃圾回收了。

如果没有频繁的垃圾回收，自然就避免了频繁导致的工作线程的停顿了，JVM GC 问题是不是就得到了大幅度的优化？

没错，正是这个设计思想让 Kafka 客户端的性能和吞吐量都非常的高，这里蕴含了大量的优秀的机制。

那么此时有人说了，如果我现在把一个缓冲池里的内存资源都占满了，现在缓冲池里暂时没有内存块了，怎么办呢?

很简单，阻塞你的写入操作，不让你继续写入消息了。把你给阻塞住，不停的等待，直到有内存块释放出来，然后再继续让你写入消息。

4、总结一下

这篇文章我们从 Kafka 内存缓冲机制的设计思路开始，一直分析到了 JVM GC 问题的产生原因以及恶劣的影响。

接着谈到了 Kafka 优秀的缓冲池机制的设计思想以及他是如何解决这个问题的，分析了很多 Kafka 作者在设计的时候展现出的优秀的技术设计思想和能力。

希望大家多吸取这里的精华，在以后面试或者工作的时候，可以把这些优秀的思想纳为己用。

互联网公司的面试官是如何360°无死角考察候选人的？（上篇）

> 作者:中华石杉 原文地址

一、写在前面

最近收到不少读者反馈，说自己在应聘一些中大型互联网公司的 Java 工程师岗位时遇到了不少困惑。

这些同学说自己也做了精心准备，网上搜集了不少 Java 面试题，然而实际去互联网公司面试才发现，人家问的，和你准备的对不上号，这就很尴尬了。。。

因此，从这篇文章开始，笔者准备写一个长期连载的系列文章：《Java 进阶面试系列》。主要跟大家聊聊中大型互联网公司 Java 面试中的一些热门、高频的技术问题。

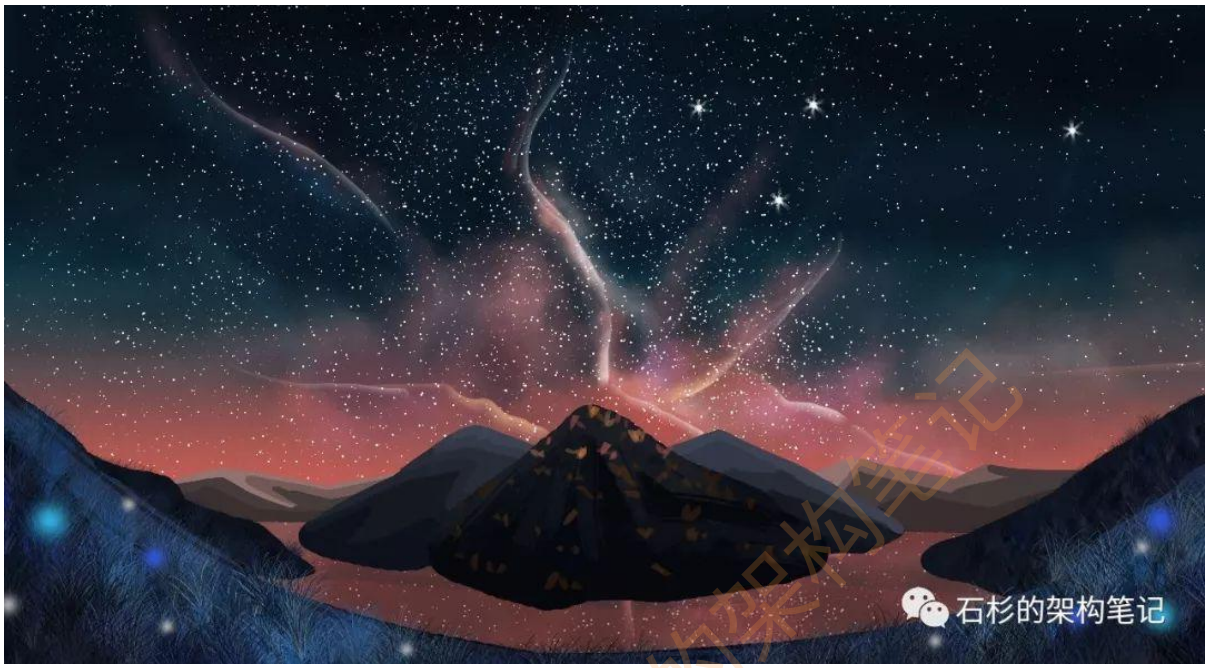
希望这个系列的文章，能在年后金三银四的跳槽季中，助各位小伙伴一臂之力

二、先来体验一个真实的面试连环炮

好，闲话不多说，我们进入正题！

本系列文章，我们将会从消息中间件、分布式缓存、分布式搜索、分布式架构、海量数据、NoSQL、高并发、高可用、高性能、数据库、JVM 虚拟机等各方面聊一下面试中的高频技术问题。

现在一些中大型互联网公司的面试官，在面试候选人时，一般都会采取连环炮的策略来深挖一个候选人的技术水平。



举个例子，比如说看你简历上写了熟悉消息中间件（MQ 技术）。那么可能我们就会有一个类似下面这样的连环炮式发问:

- 说说你们公司线上生产环境用的是什么消息中间件?
- 那你们线上系统是有哪些技术挑战，为什么必须要在系统里引入消息中间件?
- 你们的消息中间件技术选型为什么是 RabbitMQ?
- 为什么不用 RocketMQ 或者是 Kafka？技术选型的依据是什么?
- 你们怎么保证消息中间件的高可用性？避免消息中间件故障后引发系统整体故障?
- 使用消息中间件技术的时候，你们怎么保证投递出去的消息一定不会丢失?
- 你们怎么保证投递出去的消息只有一条且仅仅一条，不会出现重复的数据?
- 如果消费了重复的消息怎么保证数据的准确性?
- 你们线上业务用消息中间件的时候，是否需要保证消息的顺序性?
- 如果不需要保证消息顺序，为什么不需要？假如我有一个场景要保证消息的顺序，你们应该如何保证?
- 下游消费系统如果宕机了，导致几百万条消息在消息中间件里积压，此时怎么处理?
- 你们线上是否遇到过消息积压的生产故障？如果没遇到过，你考虑一下如何应对?
- 你们用的是 RabbitMQ？那你说说 RabbitMQ 的底层架构原理，逻辑架构、物理架构以及数据持久化机制?
- 你们 RabbitMQ 的最高峰 QPS 每秒是多少？线上如何部署的，部署了多少台机器，机器的配置如何?

- 你们用的是 Kafka？那你说说 Kafka 的底层架构原理，磁盘上数据如何存储的，整体分布式架构是如何实现的？
- 再说说 Kafka 是如何保证数据的高容错性的？零拷贝等技术是如何运用的？高吞吐量下如何优化生产者和消费者的性能？
- 看过 Kafka 的源码没有。如果看过，说说你对 Kafka 源码的理解？
- 你们用的是 RocketMQ？RocketMQ 很大的一个特点是对分布式事务的支持，你说说他在分布式事务支持这块机制的底层原理？
- RocketMQ 的源码看过么，聊聊你对 RocketMQ 源码的理解？
- 如果让你来动手实现一个分布式消息中间件，整体架构你会如何设计实现？

上面仅仅是 MQ 相关技术问题的一部分，实际上，一个比较好的面试官的问题，就是从**技术面、技术点、项目实践**几块来抽丝剥茧的发问。

三、技术广度的考察

首先考察候选人技术面的完整性，因为工作中是需要具备一定的技术视野的，不能说光知道消息中间件，但是分布式缓存却一无所知。

类似于以前高考的时候，你语文特别好，结果物理特别差，那也是不太合适的。

所以工程师首先要避免自己的技术短板，尤其是三到五年经验的同学，已经彻底度过了自己人生的职场生涯的初期小白入门菜鸟阶段。

所以，务必在工作三到五年的时候，保证自己的技术绝对没有任何短板，整体技术栈要或多或少都知道一些，不能出现盲区。

比如，我现在问你，你们公司有没有什么业务场景是可以用 NoSQL 的？现在国内各个公司用 NoSQL 的技术都有哪些选型？具体 NoSQL 可以解决什么问题？

如果你一问三不知，这就是典型的技术短板，你至少需要大概知道，每个技术一般在什么情况下用，怎么来用，解决的是什么问题。

因此，上面说的消息中间件、分布式缓存、海量数据、分布式搜索、NoSQL、分布式架构、高并发、高可用、高性能这些技术。并不是说真的要求工作几年的同学都要精通到源码层面。

而是说你工作几年以后，应该有一定的技术广度，开阔的技术视野。

四、底层技术的考察

现在很多互联网大厂都会有基本功的考察，举个例子，Java 虚拟机的核心原理、内存模型、垃圾回收、线上 FullGC 卡顿性能优化、线上 OOM 内存溢出问题你处理。

Java 并发中的 volatile、锁优化、AQS 源码；

Netty 背后的 IO、网络相关的知识。

其实这种底层技术，是线上高负载大型系统的架构设计和开发，必须要具备的。

因为底层技术不扎实，很多中间件或其他高阶的技术，都无法深入理解其原理。

而且很多时候，解决线上系统的生产故障，都需要这些技术。因此，底层技术的掌握是一个优秀工程师必须具备的素养。

五、技术深度的考察

此外，我们一定会深入考察候选人平时工作中熟悉的以及常用的一些技术。

举个例子，比如你项目里用了 Redis 或者是 Elasticsearch。

只要你用过了，而且是你某个项目里的核心技术，那么一定会用连环炮式的发问，深入各种细节、底层、生产环境可能遇到的技术挑战。

总之，就是要用压力测试出来你在这块技术水平掌握的到底有多深，实践经验有多强。

一个好的面试官，自己本身技术功底扎实，是可以对一个技术问出一连串的连环炮的，就比如上面的那个消息中间件的连环炮发问。

而且只要面试官在一个技术上的深度超过候选人，那么通过不断加深的发问，是可以考察出来一个候选人在自己最熟悉的技术领域的技术深度的。

举个例子，比如说你对一个技术的掌握是否达到了源码级别?

是否对某个框架，或者是中间件深入的理解底层的源码实现，从源码级别说清楚他的架构原理?

是否对这个技术有过线上的高可用部署，承载过高并发流量的访问?

是否对这个技术在线上生产环境解决过各种各样的复杂技术挑战?

是否基于这个技术落地到你的业务系统中，设计出各种复杂的系统架构?

通过这种连环炮，可以非常好的考察出某个候选人对技术深度的掌握。

技术深度的考察是中大型互联网公司面试官对一个高级 / 资深的候选人必须考察的。

因为如果一个人工作 5 年以上，来应聘高级职位的话，那我们绝对是要求他对至少一个技术领域有着较为深入的研究的。

比如说起码你得深入阅读过某个热门技术的核心源码，有一定的技术功底，可以解决一些复杂的线上故障。

技术广度决定了你可以利用各种技术来做项目，但是技术深度决定了你的技术功底。

你未来学新东西有多快，线上系统出了故障你能否快速定位和解决，你能否基于对技术的深刻理解为公司的项目设计和开发出复杂而且优秀的架构出来，这都取决于技术深度。

六、总结 & 预告

小结一下，本文我们用一个面试连环炮，引出了平时中大型互联网公司面试官是如何发问的。

然后从技术广度、底层技术、技术深度几个角度说了一下，我们一般如何考察候选人的技术。

接下来这篇文章：**《互联网公司的面试官是如何 360° 无死角考察候选人的？（下篇）》**

将会从项目经验的考察、系统设计的考察、候选人与岗位的匹配、多轮面试官的协作考察出发。

继续告诉大家互联网公司是如何全方位、无死角的考察候选人的。

知己知彼、百战不殆，面试也是如此。所以我们在开始《Java 进阶面试系列》之前，以这两篇文章作为序篇。

你只有真正了解了面试官的选拔标准，考察范围，才能更好的进行针对性的准备，成为行走的“offer 收割机”。

互联网公司的面试官是如何360°无死角考察候选人的？（下篇）

> **作者:中华石杉** 原文地址

一、写在前面

上一篇文章：**互联网公司的面试官是如何360°无死角考察候选人的？（上篇）**

用一个面试连环炮引出了平时中大型互联网公司的面试官是如何发问的。

紧接着从技术广度、底层技术、技术深度几个角度说了一下，我们一般是如何来考察候选人的技术。

本文是下篇，将会从**项目经验、系统设计、履历 / 学历 / 素质、候选人与岗位**的匹配、多轮面试官的协作这些方面，继续告诉大家，互联网公司是如何全方位、无死角来考察候选人的。

二、项目经验的考察

项目经验，绝对是面试官必须考察的，很可能上来就是让你先画一下项目整体架构图，说一下你们项目用了哪些技术以及核心的业务思路。

然后从项目入手，考察你项目里各个技术掌握的如何，通过连环炮对你掌握最好的技术进行深入考察，对一些高阶技术的考察，直接下探到底层。

举个例子，如果你说你们公司里用了 dubbo 作为服务框架，那么会问问你 dubbo 底层的通信框架是什么？Netty？Mina？

然后再问问你底层的 NIO 是啥？网络通信里的长连接和短连接是啥？

你是否看过 dubbo 的源码？dubbo 源码中你印象深刻的对并发技术的运用是什么？

一些面试官喜欢从项目展开问各种技术，也有一些面试官上来直接从你简历上的技术开始发问，从技术深入到项目。这就看个人喜好了。

当然无论如何，最后总会聊到项目的一些业务细节，好的面试官会掌握一个原则：**死扣细节**。

提问时，必须要深入到你把某个业务细节讲清楚，以及结合这个业务细节到底是如何落地和设计技术方案的，如何使用各种技术在业务中的。

比如说

- **你说你用了Redis，那就会进一步问你，你哪个业务用了Redis？那个业务的流程请你叙述一下？**
- **在Redis里你们具体是选用了哪种数据结构存放什么数据？数据的过期时间是什么？如果缓存过期了，你的数据兜底方案是什么，到哪儿去回查？**
- **你的key如何设计的，为什么要这么设计？你的这个业务把数据放在了Redis里，是其他哪个业务来查Redis？为什么要这样子做？如果不用Redis会怎么样？**

这只是一个例子，实际上各种技术都可以在项目里深扣细节。这就能考察出，你对这个技术的实践到底有多深，经历过多么复杂的线上业务的实践，能 hold 住一个技术解决线上系统中的哪些问题。

总之，从项目里，我们可以看出你是否负责过复杂业务架构下的分布式系统的设计和开发？

你们的系统是否是线上高并发大流量高负载场景的挑战，你是否经历过这种技术挑战？

你们的系统是否承载过亿级别海量数据的存储以及高性能读写的挑战，你是否解决过这些问题？

此外，从项目考察中，还可以直接看出你的整体能力技术定位。你是仅仅负责过一个模块呢？还是负责过一个子系统？

或者是作为架构师负责过一个完整的项目群，带过几十人的团队，设计过大规模复杂的系统架构?

所以说，你到底把控过什么样的项目，具备什么样的能力，从你负责过的项目里，直接可以看出来。

如果你来面试的是中级的岗位，那么可能我们觉得你技术整体 ok，独立负责过核心模块的开发，同时对各种技术都有一定的实践经验，就 OK 了。

如果你面的是高级 / 资深的岗位，那么我们会看看你是否带领一个小团队独立负责过一个有一定复杂度和难度的完整系统的架构设计和开发。

如果你面试的是架构师的岗位，那我们肯定是要求你在一个公司里主导过很多人协作完成的大型而且复杂的项目群。

并且我们要求你对一个大型系统架构有深度的思考和整体的把控，而且这个项目要有足够的技术挑战，大用户量、高并发、海量数据，等等。

因此，项目考察，是重中之重。很多同学平时积累了不少的技术学习，但是有一个很大的问题是，项目经验和实践太少。

这些同学可能确实没经历过复杂系统的架构设计的历练，所以非常容易在项目经验考察这块出现问题，被面试官判定为技术不错，但是经验缺乏。

三、系统设计的考察

这个也是很多互联网大厂的面试官，在考察一些高级工程师及以上的同学，喜欢发问的。

一般会用自己公司或者团队里的一些业务场景拿出来，或者是普遍性的一些业务场景，然后来问你如何针对这个业务场景设计系统架构?

举几个例子：

- 如何设计一个电商秒杀系统架构?
- 如何设计一个消息推送系统架构?
- 双 11 大促的时候如何设计系统的动态扩容 / 缩容的机制?

类似诸如此类的一些场景式的系统设计考察。其实这个主要是用一些你可能没接触过的场景，来现场考察一下你的架构设计思维。

尤其是针对上面说的高级 / 资深、架构类的岗位，我们尤其会注重现场考察你没接触过的业务场景的架构设计。

因为毕竟你来了以后，肯定要让你接触全新的业务，然后立马给出合理而且靠谱的架构设计方案，在新的公司来落地你的经验。

很多同学平时不太注意积累系统设计的能力，导致出去面试的时候，人家一问场景系统设计问题，直接发蒙了。

所以，平时应该对公司里各种业务场景多思考，自己设定一些挑战，比如假设你公司的请求量暴增 100 倍，数据量暴增 100 倍，你的系统架构应该如何设计？

多给自己设立挑战，然后去尝试着思考设计，才能积累出系统设计的思维和能力来。

四、基本功的考察

很多大厂都会考察候选人的基本功，尤其是数据结构和算法。比如现场手写一些常见的算法题。

很多同学很容易倒在基本功这块，一些基础的数据结构和算法题都不会写，那就是有点问题了。

这里强调一下，这个东西并不是应届生专用的，其实也代表了一个工程师，甚至一个架构师的基本技术素养问题。

因此建议大家平时还是要注重基本功的保持，平时写写算法题，熟悉一下数据结构，能保持自己的技术素养不会掉落。

否则数据结构和算法都不熟悉，对复杂系统的技术细节把控基本也就没法做到，因为很多复杂分布式系统的源码里，到处是自己写的数据结构和复杂算法。

五、履历背景 / 学历背景 / 过往经验 / 综合素质

最后一定会综合看一下一个候选人整体的背景，比如你的履历背景。

履历背景

- 你过去是外包公司出身？还是传统 IT 公司出身？或者是一些小型互联网公司？或者是一二线大互联网公司出身？
- 另外你的学历如何？是大专？普通本科？211 / 985 本科？普通硕士？211/985 大学的硕士 or 博士？
- 你过去做的都是一些内部系统，比如 OA 系统，财务系统？或者都是 C 端系统，有上千万用户量的系统？或者你过去做的都是某种偏门的项目，比如爬虫之类的？
- 你的沟通表达能力如何？性格是否踏实和 nice，不浮躁？你是否有团队协作精神？

这些综合性的东西，其实都会在我们的整体考察范围之内，都会纳入考虑范围内，最后决定要不要发 offer。

六、候选人与岗位需求的匹配

其实按照上述流程考察下来以后，会经历多轮面试，基本一次好的面试就可以综合考察出一个候选人的完整情况了。

这个候选人的技术面是否完整，是否有几个技术领域有足够的深度?

候选人做过什么样的项目，项目的实践经验如何，把控过多大的团队和多大的项目，

对全新业务场景的系统设计能力如何，基本功如何，综合背景和素质如何。这些东西，基本上都可以很好的考察出来了。

此时就会将一个候选人跟岗位的需求进行匹配，比如说你要招聘的是一个资深 Java 的岗位，需要他过来开发的是公司里较为核心的子系统。

然后呢，你公司的技术栈是 dubbo、zk、kafka、redis，等等

你们公司每秒有上万的并发访问压力，数据量一亿以上，线上系统偶尔故障，比如高并发下 zk 突然报错异常，导致系统业务中断，然后需要带 4 个初级和中级的兄弟一起开发。

这时，你考察完一个候选人，就知道他的技术能力是否匹配这个岗位，技术深度能否 cover 住线上系统常见的一些故障。

能否在线上故障的时候，立马有足够的源码功底分析、定位和解决问题。是否有过往类似足够的高并发和海量数据的项目经验。

是否带过几个人独立把控过一个核心系统的架构设计和开发，过去的公司背景咋样，学历咋样，综合素质咋样。

这个候选人和岗位需求是否匹配，基本上就出来了。

七、多轮面试官的分工协作

上面列举了大量的技术考察的内容，实际上很难说是一轮面试官直接完成的。

因此，一般我们都是分成多轮面试官协作考察。但是根据不同的公司，不同轮的面试官的职责会稍微有一些不一样。

比如说一面面试官可能主要就是考察一下技术内容，包括技术面以及连环炮发问考察技术深度，以及算法功底，不太涉及项目。

二面面试官可能会着重考察**项目经验，系统设计，同时对技术深度**也会继续考察。

三面面试官可能会从你把控过的**项目规模、带的团队规模、团队管理能力、规范和流程设计能力、整体工作履历背景和经验、软素质（沟通表达、团队协作、价值观，等等）**来考察你。

上面说的只是一种分法，一个公司内的不同团队的分工可能是不一样的。

也有的可能是一二面都是考察技术面和技术深度，不涉及项目，三面来考察你的项目经验，四面来考察你的一些综合素质。

或者可能你面的职位很高，比如是总架构师之类的职位，也许还有 CTO 或者技术 VP 出来面试你第五轮。

但是不管如何分，整体考察的内容都是上面的那套东西以及那个流程和过程。

八、Java 进阶面试系列的开始

下一篇文章，我们就开始以连载的形式，跟大家聊聊**中大型互联网公司 Java 面试时高频考察的技术点以及相关的面试技巧**

【行走的 Offer 收割机】记一位朋友斩获 BAT 技术专家 Offer 的面试经历

> **作者:中华石杉** 原文地址

概述

之前写过两篇文章：

- 互联网公司的面试官是如何360°无死角考察候选人的？（上篇）
- 互联网公司面试官是如何360°无死角考察候选人的？（下篇）

通过这两篇文章，我们给大家聊了聊国内中大型互联网公司，在 Java 面试时一些高频的技术问题。

本文我们通过一篇真实的一线面经，带大家去体验一下 BAT 等互联网公司的面试现场氛围！

背景介绍

> **PS：面试者是笔者以前的下属，多年的好朋友。这是他今年早些时候出去面试，拿到 BAT 等多家一线互联网公司技术专家 Offer 的面试经历。**

先介绍一下这位朋友的个人经历：

- 本科毕业，接近 10 年工作经验。跳槽之前，在国内某大型互联网公司里带一个 8 人左右的技术团队。
- 由于公司业务发展较为平缓，所以职业上升机会较少。
- 朋友对其负责的系统架构和技术已经非常熟悉，薪资上也较难有大幅度的增长，至于晋升更高的级别，短期内也不容易。

因此，在仔细思考一番之后，决定出来看看机会，能否在带团队的规模、技术以及薪资上实现一个突破。

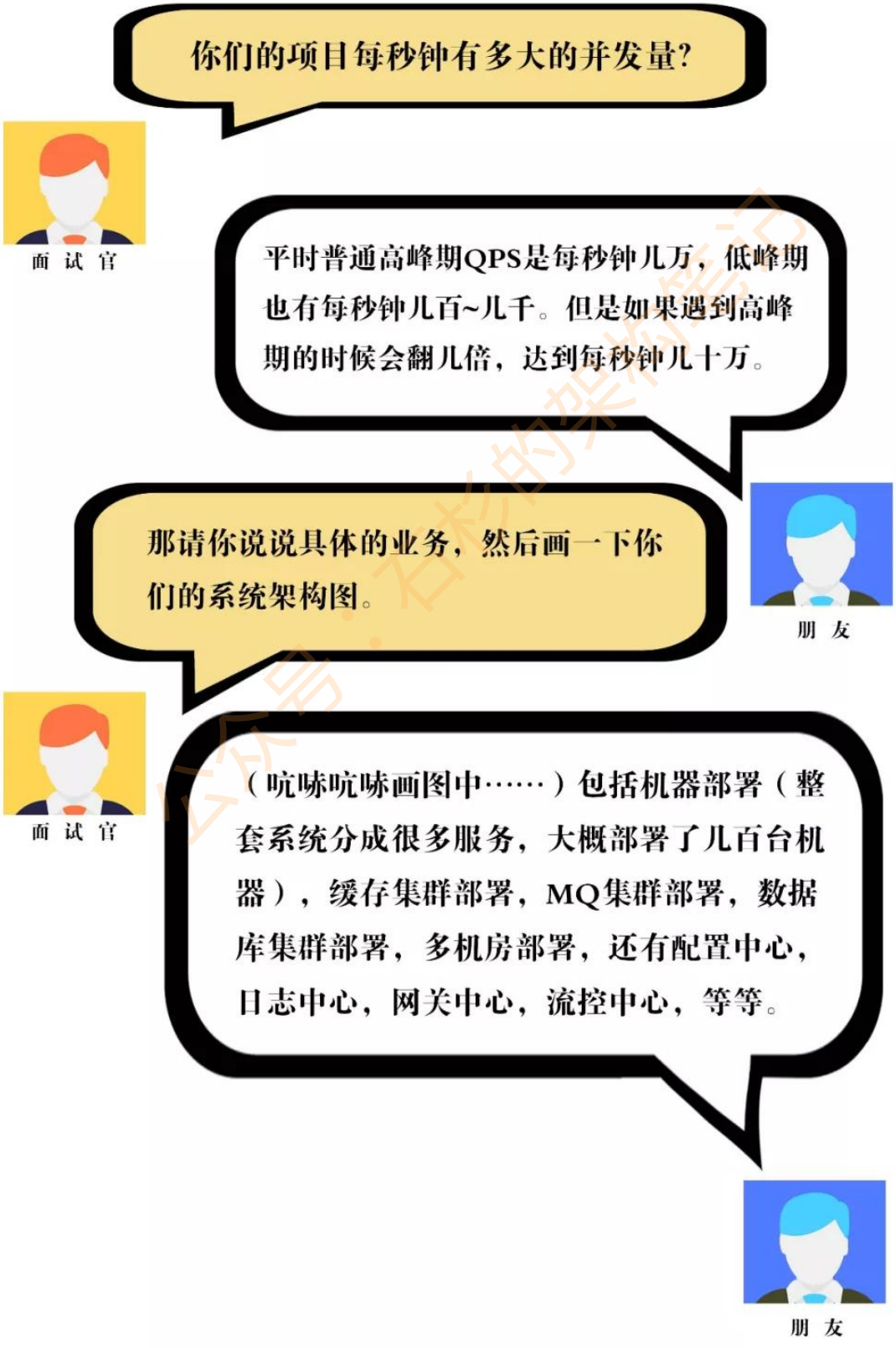
一面

一面是一个猎头给朋友推的一个职位，BAT 中某一个大厂的某个团队，具体就不说是哪个部门了。

一面就直接过去当面聊了一次，大概从下午 2 点聊到了下午 4 点多，时间很长，炮火相当猛烈。

一面面试官也是专家职级，上来就是先聊项目，针对项目中的各种细节仔细问，就项目展开，而且极其注重细节。

下面的内容，是根据朋友面试之后的回忆，整理出的部分问题。



面试同样是通过互联网公司最喜欢的连环炮形式发问。比如在面试过程中，聊到了缓存。连环炮如下：

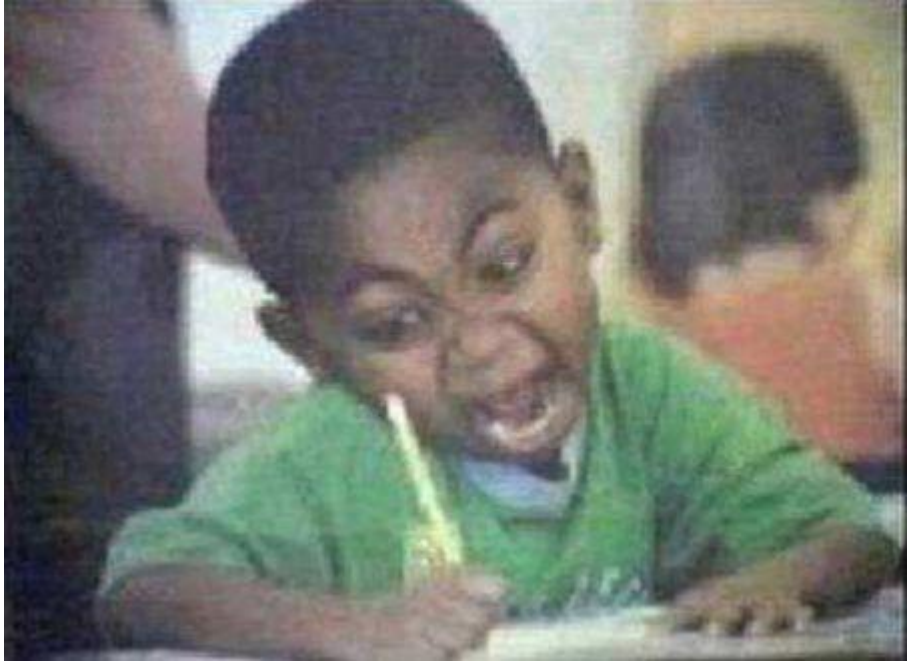
接着，面试官继续深扣了很多细节

面试官：

- 那请说一下，这些请求具体是落在哪些接口上？
- 哪些数据是数据库和缓存双写一份的？
- 双写一致性如何保证？保证一致性的同时如何保证高并发和性能？
- 缓存线上是如何部署的？给了多大的总内存？命中率有多高？
- 缓存抗了多少 QPS？数据流回源会有多少 QPS？
- 是否某个 key 出现了热点缓存导致缓存集群中某个机器的负载过高？如何解决的？
- 是否出现超大 value 打满网卡的问题？如何规避这个问题？
- 线上是否出过缓存集群事故？如果出现了你们怎么解决有什么高可用保障预案？
- 平时如何监控缓存集群的 QPS 和容量？如果要扩容该怎么扩？能否平滑扩容？扩容会导致系统需要停机吗？
- 聊聊 Redis 的集群原理？扩容的时候会不会导致数据丢失？key 寻址算法都了解哪些？
- 你了解一致性 hash 算法吗？画个图说说 Redis 线程模型和内存模型？

朋友：

纸笔翻飞，大脑高度运转，一个接一个的回答。。。



如上所述，所有问题，全部结合项目，落地到生产中，同时注重聊技术的很多细节，包括技术的一些原理。

像缓存这样的连环炮提问法，面试官还用来问了 MQ、MySQL 分库分表、高可用、JVM、多线程并发，等各种问题。

简单总结：

- 一面其实关注了技术广度，同时结合项目死扣各种细节。
- 另外也兼顾了一定的技术深度，会就一个技术往深了问下去。

总体来说，一面还算顺利，毕竟都是结合项目来问的，各种细节平时朋友进行架构设计时，都会仔细考虑过。

而且朋友也做过线上的高并发系统，踩过很多坑，所以这些问题基本都回答的不错。

但是这里给大家提醒一句，一般某个同学出去面试，回来之后其他人问他面试经验，一般都是问：都有啥面试题？面试官是怎么问的？

说实话，大家看了上面那些问题，可能会觉得说，哦，其实我也可以答出来，没什么特别的。

但其实并不是这样，如果只是拿高级岗位的 Offer，你的技术会占很大比重。

但是如果要拿专家岗位的 Offer，你到底有没有线上真实的高负载的系统架构经验，非常重要。

同样的问题，普通人会回答的很普通，但是经历过真实几十亿流量请求的人一定会说出大量经验总结、教训以及采坑。

而且对整套复杂的大型系统到底是如何抗住高并发的，会了然于胸，熟悉所有的细节。

所以针对一面，一般就是结合项目，深挖细扣，看你到底有多少水平，做过多复杂的系统。

这块说实话，做过就是做过，没做过就是没做过，是不可能作假的。

很多同学可能自己平时也看过很多书和博客，但是看书和博客只是基础，如果没有真实的线上生产环境的历练，是肯定不够的。毕竟实践出真知！

二面

一面就顺利通过了，紧接着安排了第二轮面试。

二面面试官应该是这个团队的 leader，P8 级别的，如果进去，应该就是朋友未来的顶头上司。

据朋友讲，二面面试官态度非常好，很和蔼，看来一面面试官反馈之后，这个 team 对朋友还是比较重视的。

（1）技术深度

二面内容就从广度变成深度了，面试官技术实力很深厚，应该是有十几年经验。对相关技术深挖了很多东西。

同样，二面也聊到了缓存相关的问题。问了朋友具体了解过哪些缓存技术，redis、memcached，还有阿里开源的 tair，哪个了解过内核原理?

朋友之前看过一些 redis 的内核，就聊了聊 redis 内核的一些数据结构和实现原理。包括集群、持久化在内核层面的一些东西。

此外在 MQ 这块，朋友正好对 kafka 做过深入的研究，就聊了聊 Kafka 的源码。

比如 KafkaController 在故障转移这块的源码，日志存储、网络通信的一些细节。如何保证磁盘读写的高性能，零拷贝那块的底层实现，leader 和 follower 之间的数据是如何同步的，都是从源码层面来聊。

此外，还聊了 dubbo 的源码以及 mysql 内核层面的东西。

（2）系统设计、工程素养、带团队

同时二面非常重视考察系统设计能力、工程素养、带团队的能力。

比如面试官就这个部门负责的一块业务，出了一个相关的系统设计题目。

题目细节记不清楚了，大体内容是给出具体的用户量、业务场景、并发量、数据量，然后让你整体负责这个系统的架构设计。

朋友需要阐述自己的整体设计思路，从哪些点来考虑，存在着哪些技术挑战，并且现场画出来具体的架构设计图。

工程素养这块，让朋友聊了聊平时如何做的技术设计、技术评审、编码规范、测试、上线、回滚、灰度、压测、监控等等。

带团队，让朋友说一下，如何招人、面试标准、如何搭建团队的人才梯度，等等。

（3）架构演进

此外，还会问一下，整个系统架构是如何一步一步进行演进的。

从 0 到 1 的时候是什么架构？从 1 到 10 的时候是什么架构？从 10 到 100 的时候是什么架构？这块就是看看你的整体架构能力，以及技术规划能力。

说到这里，笔者提一句，如果出去面试，尤其是去 BAT 等大型互联网公司面试，必须精心准备。

包括你的项目的每个细节，你解决过的各种线上问题和坑，你简历里的技术是否达到一定的深度，你平时其他的工程、设计能力，这些都一定要精心准备一下。

绝对不要裸面！绝对不要裸面！绝对不要裸面！

重要的事情说三遍！裸面必败，而且如果一问三不知，那么给人的印象就是很差的。

如果要冲着心仪的大公司去，最起码精心准备 1 个月以上，大家务必记住这一点，这也是朋友这次的一个重要心得，准备充分了，才能有备无患。

三面

二面之后，又等了大概一两周。。。

因为越往上面，领导级别越高，平时越忙，有时人家可能出差开会去了，不过等了一两周，那边总算约上了三面。

三面是总监级别的，不太确定是走的 M 线还是 P 线。如果是 P 线，那么一定是 P9，但是观察面试风格应该是 M 线的总监。

这一面，聊技术其实并不多，更多的是跟朋友聊过往的各种公司的经历和项目经验，具体负责过哪些比较有挑战的大型的系统。

另外，考察了各种软素质。比如说责任心、抗压能力、自我驱动，让朋友举例说明自己过去的一些事情，来证明软素质。

同时还会聊聊职业价值观，是否愿意加班，等等吧。最后也聊了聊朋友的职场期望，包括这个团队是干什么的，未来的发展方向之类的。

朋友觉得最重要的还是前面两面，其实这一面，只要人品端正，平时干活儿认真负责，一般的都没什么太大的问题。

终面

接着又过了一两个礼拜，因为当时二面面试官，也就是那个未来可能成为朋友 leader 的人，对朋友还是比较看重的，私下还短信联系了一段时间，就怕朋友跑去别的公司了。他告诉朋友说是因为 HR 那边太忙了，所以终面还未安排上。

关于 HR 面，朋友印象真是相当之深刻，为什么呢？因为 HR 是直接电话聊的，没过去了，过去实在太折腾，而且二面面试官也是去打了招呼。

HR 当时居然是晚上 11 点打来的电话，人家刚刚加班开会结束，就打来了电话，真是不得不佩服其敬业精神！

而且这位 HR 是相当专业的，如果是普通的 HR 其实随便聊聊就行了，但是这边的 HR 问了很多问题，大概聊了 1 个小时左右。

主要是跟朋友聊了一些价值观的东西，比如之前觉得做过最难的事情是啥，怎么克服的，当时啥心态。

还有就是为啥要离职，没有发展空间？那当时没考虑过公司内部 transfer（转岗）吗？为啥不好 transfer？你的绩效平时怎么样？你觉得你跟同事相处的怎么样？

终面内容，总结起来，其实还是一句话，你人品正就好了，一般都问题不大，老老实实的踏实回答。

后来 HR 面了过后，那边的薪资确实给到位了，达到了朋友的期望薪资。

但是那边给的规划是未来可以带的团队人数也就是 10 人以内，而且不是配发集团股票，是配发的正在快速发展的这个团队的期权。

所以朋友当时纠结了一下，但还是先答应了，于是 offer 就发了过来。

后记

本来朋友想的是，如果没有别的更好的机会，那么这个机会也可以考虑，毕竟薪资上还是可以的。

但是当时包括 TMD（头条、美团、滴滴）这边，也都有人内推朋友过去试试，所以当时也面了其他的几个一线互联网公司。

其实如果经历了 BAT 这种互联网公司的几轮技术面试洗礼，那么去国内任何一个公司都没什么问题了，所以当时面试也都很顺利，驾轻就熟。

同样，朋友也不出意外的拿到了那些一线互联网公司的 offer。

经过一番对比，朋友最终没有选择去最初面试的那个 BAT 中的某个大厂，而是去了上面说的那几个超级独角兽公司中的其中一个。

原因是这家超级独角兽公司给出的薪资超出期望之外，而且领导对朋友同样非常重视，配发了大量的期权，承诺可以独立带 20 + 人的团队。

而朋友更看重的是这个超级独角兽公司未来的潜力。

- 第一，公司发展速度快，人员扩张迅猛，所以给到的带团队的机会非常好，能带更大的团队，比朋友当前带的团队规模大了一倍多。
- 第二，虽然 BAT 的那家大厂同样配发了期权，但是这家超级独角兽的期权未来潜力可能更大。事实证明，的确如此。
- 所以综合考虑了之后，朋友最终还是根据自己的职业发展选择了独角兽公司，没有再回到 BAT 行列中。

扎心！线上服务宕机时，如何保证数据100%不丢失?

> **作者:中华石杉** 原文地址

一、写在前面

上篇文章**哥们，消息中间件在你们项目里是如何落地的?**，我们用一个简单易懂的电商场景给大家引入说明了一个消息中间件的使用场景。

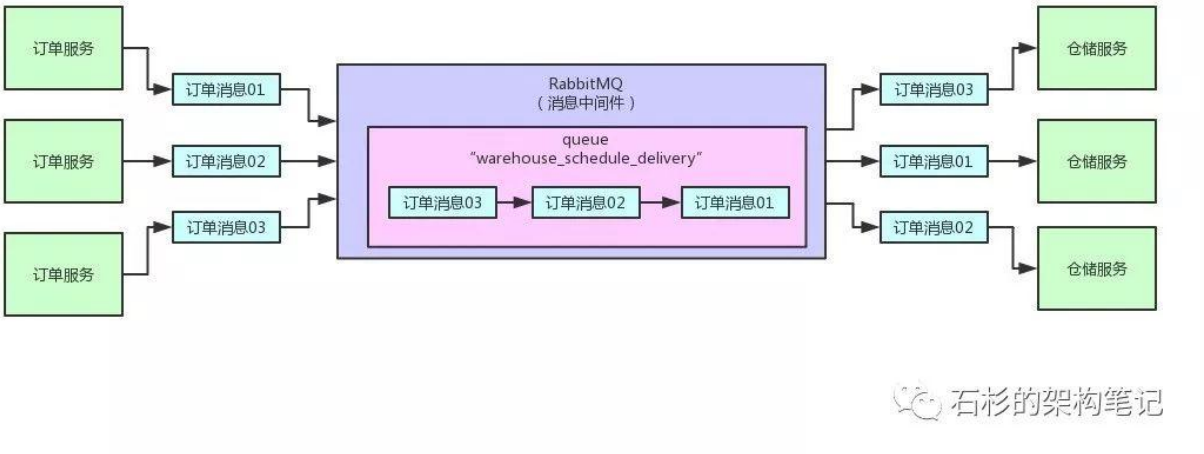
同时，我们还基于 RabbitMQ 的 HelloWorld 级别的代码，给出了订单服务和仓储服务如何基于 MQ 中间件收发消息的示例。

二、业务场景回顾

这篇文章，我们来稍微深入探讨一些 MQ 中间件使用中的基础技术问题。

首先回顾一下上篇文章做出来的一个架构图，看看订单服务和消息服务是如何基于 MQ 来收发消息的。

我们稍微把这个图细化一点，简单来说就是多个订单服务实例给 queue 推送消息，多个仓储服务每个消费一部分消息。如下图所示：



三、意外宕机，问题凸现

假如你线上对 MQ 技术的使用就到此为止了，那么基本可以跟 offer 说拜拜了。。。

因为如果是我的话，作为一个面试官就没法继续往下问了。你这个 MQ 的使用以及理解的深度仅此而已的话，那基本就是刚刚对 MQ 技术入门的程度。

如果面试官要继续问，完全可以问下面的问题：

- 那你说说如果仓储服务作为消费者服务，刚收到了一个订单消息，但是在完成消息的处理之前，也就是还没对订单完成仓储调度发货，结果这个仓储服务突然就宕机了，这个时候会发生什么事情?

所以说，大家还是要对这个技术了解的稍微深入一点点，否则随便被问几个问题就完蛋了。

大伙儿先来看看下面的图，感受一下车祸现场。

RabbitMQ 这个中间件默认的一个行为，就是只要仓储服务收到一个订单消息，RabbitMQ 就会立马把这条订单消息给标记为删除，这个行为叫做**自动 ack**，也就是投递完成一条消息就自动确认这个消息处理完毕了。

但是接着如果此时仓储服务收到了一个订单消息，但是还没来得及对仓库系统完成商品的调度发货，结果直接就宕机了。

此时，明显这个订单消息就丢失了啊，因为 RabbitMQ 那里已经没有了。。。

这会导致什么样的尴尬体验呢？就是一个用户支付了 8999 元，对一个 iphone8 下了订单，结果呢，死等活等了好几天，就是不见网站上显示他的 iphone8 在发货。

搞了半天，原因就是他的那个 iphone8 的订单在仓储服务那里，还没来得及调度发货直接就宕机了，导致这个订单消息就一直丢失了，始终没有给这个用户通知仓库系统进行发货。

这个问题，是不是很尴尬？所以说，技术问题是会严重影响企业的核心业务流程的！

各位小伙伴，还记得上一讲咱们的仓储服务消费消息的代码中，**有一行关键的代码：**

```
channel.basicConsume(
    QUEUE_NAME, true, deliverCallback, consumerTag -> { }
);
```


这行代码对 `channel.basicConsume()` 方法，传入的第二个参数：true，其实就是一个关键的参数。

这个 true 就代表了一个核心的含义，他的意思是，RabbitMQ 只要把一个消息投递到仓储服务手上，立马就标记这个消息删除了。

但是在这个默认的配置之下，要是仓储服务收到一个订单消息，结果还没来得及完成耗时几十秒的仓储调度发货的业务逻辑，结果突然宕机了，那么这个订单消息就永久性丢失了！

找了半天，原来问题的症结在这里啊！大家是不是明白了，上一篇文章最后为什么我会说，这个代码目前为止还有很多的问题。

所以这个时候，我们如果希望不要因为仓储服务的突然宕机导致一条订单消息丢失，就需要改造一下仓储服务消费消息的代码了。

首先，我们需要把那个参数从 true 改为 false，如下代码所示：

```
channel.basicConsume(
    QUEUE_NAME, false, deliverCallback, consumerTag -> {}
);
```

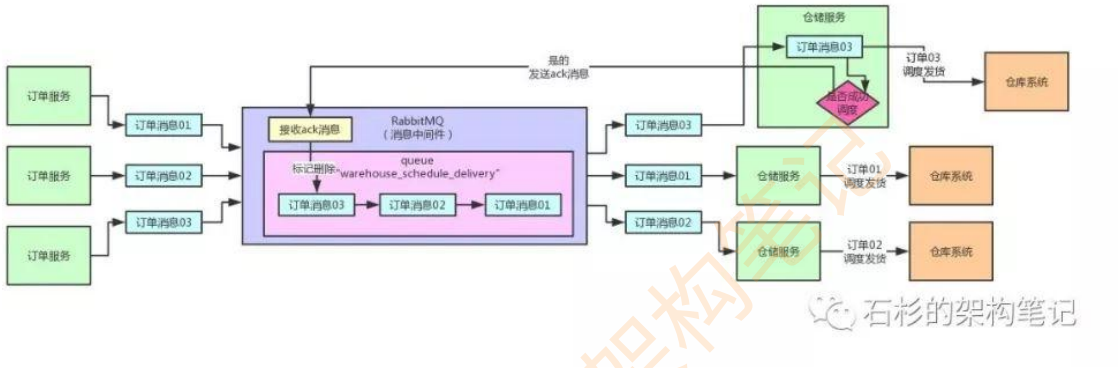

只要修改为 false 之后，RabbitMQ 就不会盲目的投递消息到仓储服务，立马就删除消息了，说白了就是关闭 autoAck 的行为，不要自作主张的认为消息处理成功了。

接着，我们需要改造一下处理订单消息的代码，如下代码所示。

这段代码，说白了，就是在对订单完成了调度发货之后，在 finally 代码块中手动执行了 ack 操作，说我自己已经完成了耗时几十秒的业务逻辑的处理，现在可以手动 ack 通知 RabbitMQ，这个消息处理完毕了。

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    try {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] 仓储服务接收到消息，准备执行调度发货的流程 '" + message + "'");
    } finally {
        System.out.println(" [x] 订单完成调度发货");
        // 这就是核心代码，手动在代码里对RabbitMQ进行ack
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    }
};
```


此时整个架构运行流程大致看起来跟下面的图那样子。



架构流程改成上面那样后，就意味着只有完成了仓储调度发货的代码业务逻辑，确保仓库系统收到通知之后，仓储服务才会在代码中手动发送 ack 消息给 RabbitMQ。

此时，RabbitMQ 收到了这个 ack 消息，才会标记对应的订单消息被删除了。

如果说在仓储服务收到了订单消息，但是还没来得及完成仓储调度发货的业务逻辑，那也就绝对不会执行这条订单消息的 ack 操作，然后 RabbitMQ 也就不会收到这条订单消息的 ack 通知。

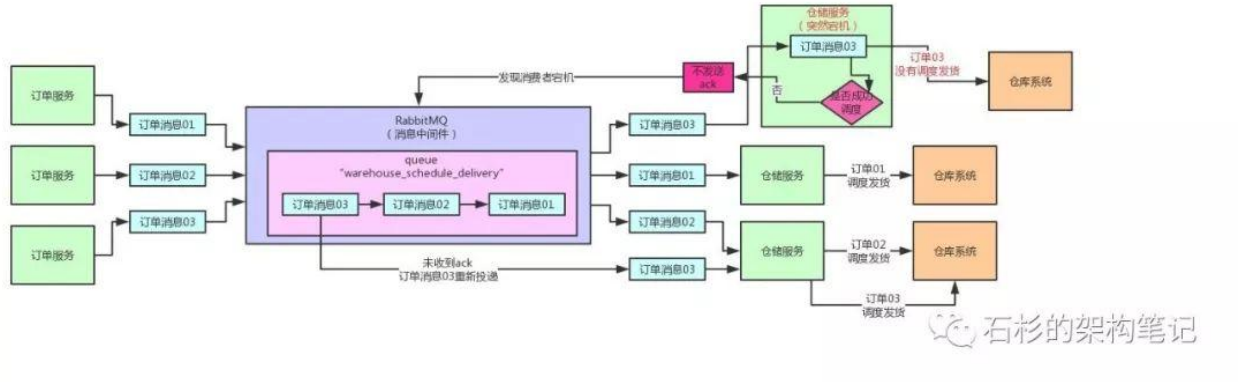
一旦 RabbitMQ 发现代表消费者的某个仓储服务实例突然宕机了，而这个仓储服务收到的一些订单消息还没来得及处理，没给自己发送那些消息的 ack 通知。

此时，RabbitMQ 会自动对这条订单消息重发推送给其他在运行中的仓储服务实例，让其他的仓储服务实例去处理这条订单消息。

这样的话，就可以保证这条订单消息不会因为某个仓储服务实例的宕机而丢失，他会确保必须由某个仓储服务实例完成这条订单消息的调度发货处理，然后才会删除那条订单消息。

四、总结 tips

最后再来一张图，大家直观的感受一下：



好了，各位同学，这篇文章是不是相对稍微深入一点点，让大家了解到了一些使用 MQ 技术时候要考虑的一些问题?

实际上无论是 RocketMQ、Kafka 还是 RabbitMQ，都有类似的 autoAck 或者是手动 ack 的机制。

线上生产环境中运行时，你必须要考虑到消费者服务可能宕机的问题。

如果消费者服务没处理完消息就自己宕机了，那么一定会导致部分消息的丢失，进而影响核心业务流程的运转。

因此大家在线上使用 MQ 时，一定要充分考虑这些潜在问题，同时结合具体的 MQ 提供的一些 API、参数来进行合理设置，确保消息不要随意丢失。

一次JVM FullGC的背后，竟隐藏着惊心动魄的线上生产事故！

> **作者:中华石杉** 原文地址

> ! “ 这篇文章给大家聊一次线上生产系统事故的解决经历，其背后代表的是线上生产系统的 JVM FullGC 可能引发的严重故障。

一、业务场景介绍

先简单说说线上生产系统的一个背景，因为仅仅是文章作为案例来讲，所以弱化大量的业务背景。

简单来说，这是一套分布式系统，系统 A 需要将一个非常核心以及关键的数据通过网络请求，传输给另外一个系统 B。

所以这里其实就考虑到了一个问题，如果系统 A 刚刚将核心数据传递给了系统 B，结果系统 B 莫名其妙宕机了，岂不是会导致数据丢失?

所以在这个分布式系统的架构设计中，采取了非常经典的一个 **Quorum 算法**。

这个算法简单来说，就是系统 B 必须要部署奇数个节点，比如说至少部署 3 台机器，或者是 5 台机器，7 台机器，类似这样子。

然后系统 A 每次传输一个数据给系统，都必须要对系统 B 部署的全部机器都发送请求，将一份数据传输给系统 B 部署的所有机器。

要判定系统 A 对系统 B 的一次数据写是成功的，要求系统 A 必须在指定时间范围内对超过 Quorum 数量的系统 B 所在机器传输成功。

举个例子，假设系统 B 部署了 3 台机器，那么他的 Quorum 数量就是：3 / 2 + 1 = 2，也就是说系统 B 的 Quorum 数量就是：所有机器数量 / 2 + 1。

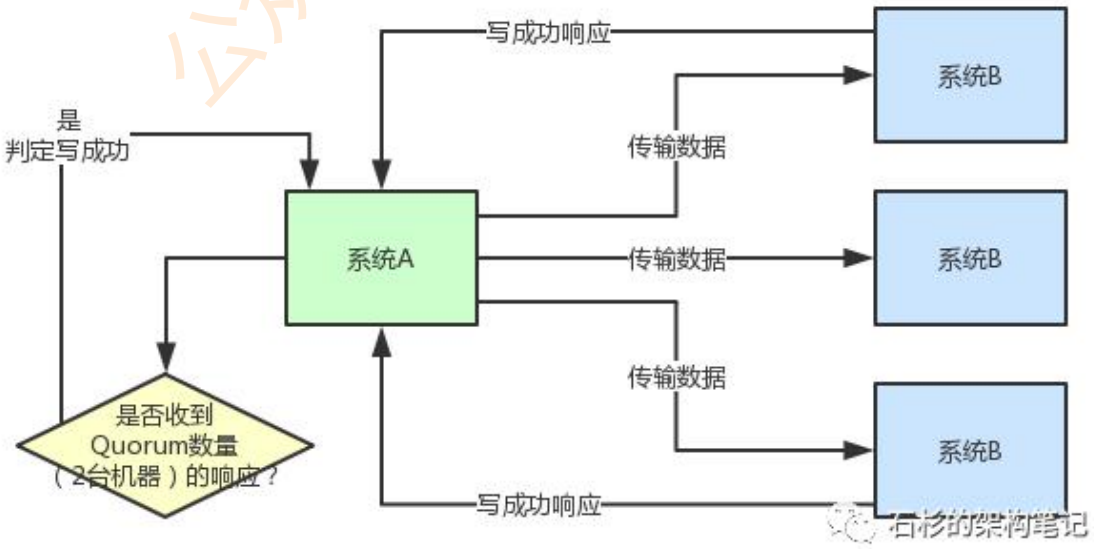
所以系统 A 要判定一个核心数据是否写成功，如果系统 B 一共部署了 3 台机器的话，那么系统 A 必须在指定时间内收到 2 台系统 B 所在机器返回的写成功的响应。

此时系统 A 才能认为这条数据对系统 B 是写成功了。这个就是所谓的 Quorum 机制。

也就是说，分布式架构下，系统之间传输数据，一个系统要确保自己给另外一个系统传输的数据不会丢失，必须要在指定时间内，收到另外一个系统 Quorum（大多数）数量的机器响应说写成功。

这套机制实际上在很多分布式系统、中间件系统中都有非常广泛的使用，我们线上的分布式系统也是采用了这个 Quorum 机制在两个系统之间传输数据。

给大家上一张图，一起来看一下这套架构长啥样。



如上图所示，图中很清晰的展示了系统 A 和系统 B 之间传输一份数据时的 Quorum 机制。

接下来，我们用代码给大家展示一下，上面的 Quorum 写机制在代码层面大概是什么样子的。

PS：因为实际这套机制涉及大量的底层网络传输、通信、容错、优化的东西，所以下面代码经过了大幅度简化，仅仅表达出了一个核心的意思。

```
// 根据系统B所在的机器数量计算出他对应的quorum数量（n / 2 + 1）
int quorum = countQuorum();

// 从配置文件加载预设值好的超时时间
// 必须要在这个指定时间范围内等到quorum数量的机器返回结果
long timeoutThreshold = configuration.get();

// 根据当前时间计算出来预期的过期时间
long expireTime = currentTime() + timeoutThreshold;

// 这段代码会通过多线程异步的方式，把一份数据传递给系统B的所有机器
SystemB.write(data);

// 因为给系统B的各个机器传输数据是开了子线程异步执行的方式
// 所以上面代码执行完毕后，立马就可以执行到这里来，代码会往下走

// 这里会进入一个无限的while循环
// 因为给系统B的各个机器传输数据可以是异步的，但是要同步的等待Quorum数量的机器返回结果
```

```
// 这里会进入一个无限的while循环
// 因为给系统B的各个机器传输数据可以是异步的，但是要同步的等待Quorum数量的机器返回结果
while(true) {

    // 这里就是检查一下系统B有几台机器返回了结果
    // 如果说系统B目前返回结果的机器（不管是成功还是失败），超过了quorum的数量
    // 此时while循环就可以中断
    if(countAvailableResponse() > quorum) {
        break;
    }

    // 获取当前时间
    long now = currentTime();

    // 判断当前时间是否超过了预设值的timeout超时时间
    // 如果超时时间超过了timeout
    // 说明指定时间范围内，没有等到Quorum数量的机器返回响应结果
    // 此时按照整体架构设计，系统A必须直接退出
    // 因为系统A严重依赖系统B，一旦系统B无法正常响应，系统A也必须连带停机
    if(now > expireTime) {
        Logger.error("系统B集群故障，无法传输数据，系统A自动退出");
        System.exit(1);
    }

    // 如果暂时还没超出截止时间，则休眠1秒钟
    // 然后继续下一轮循环来检查系统B返回的结果数量
    Thread.sleep(1000);
}
```


上面就是经过大幅精简后的代码，不过核心的意思是表达清晰了。大家可以仔细看两遍，其实还是很容易弄懂的。

这段代码其实含义很简单，说白了就是异步开启线程发送数据给系统 B 所有的机器，同时进入一个 while 循环等待系统 B 的 Quorum 数量的机器返回响应结果。

如果超过指定超时时间还没收到预期数量的机器返回结果，那么就判定系统 B 部署的集群出现故障，接着让系统 A 直接退出，相当于系统 A 宕机。

整个代码，就是这么个意思！

二、问题凸现

光是看代码其实没啥难的，但是问题就在于线上运行的时候，可不是跟你写代码的时候想的一样简单。

有一次线上生产系统运行的过程中，整体系统负载都很平稳，本来是不应该有什么问题，但是结果突然收到报警，说系统 A 突然宕机了。

然后就开始进行排查，左排查右排查，发现系统 B 集群都好好的，不应该有问题。

然后再查查系统 A，发现系统 A 别的地方也没什么问题。

最后结合系统 A 自身的日志，以及系统 A 的 JVM FullGC 进行垃圾回收的日志，我们才算是搞清楚了具体的故障原因。

三、定位问题

其实原因非常的简单，就是系统 A 在线上运行一段时间后，会偶发性的进行长时间 Stop the World 的 JVM FullGC，也就是大面积垃圾回收。

但是，此时会造成系统 A 内部的工作线程大量的卡顿，不再工作。要等 JVM FullGC 结束之后，工作线程才会恢复运作。

我们来看下面那个代码片段：

```
/**
  如果在下面这行代码运行之前，突然发生严重的JVM FullGC，
  此时会导致可能会卡顿几十秒，甚至几分钟之后，才会恢复运行，执行下面的代码

  比如现在是晚上20:00:00，结果此时突发JVM FullGC，工作线程卡顿，
  JVM FullGC一共执行了40秒。40秒过后，才执行到了下面的代码

  此时，我们通过currentTime()获取到的当前时间是：20:00:40
**/
long now = currentTime();

/**
  但关键问题是，expireTime可能不过就是20:00:20而已，
  比如20:00:00开始进入while循环，指定的timeout时间是20秒，
  所以expireTime就是20:00:20

  但是，因为JVM FullGC造成的卡顿，导致执行到下面代码的时候，
  now = 20:00:40，明显now是大于expireTime的。

  此时会导致系统A不做其他任何处理，直接就进入if语句代码，退出。
  而这就造成了JVM FullGC偶发性的导致系统A莫名其妙的宕机
**/
if(now > expireTime) {
    Logger.error("系统B集群故障，无法传输数据，系统A自动退出");
    System.exit(1);
}
```

但是这种系统 A 的莫名宕机是不正确的，因为如果没有 JVM FullGC，本来上面那个 if 语句是不会成立的。

他会停顿 1 秒钟进入下一轮 while 循环，接着就可以收到系统 B 返回的 Quorum 数量的结果，这个 while 循环就可以中断，继续运行了。

结果因为出现了 JVM FullGC 卡顿了几十秒，导致莫名其妙就触发了 if 判断的执行，系统 A 莫名其妙就退出宕机了。

所以，线上的 JVM FullGC 导致的系统长时间卡顿，真是造成系统不稳定运行的隐形杀手之一啊！

四、解决问题

至于上述代码稳定性的优化，也很简单。我们只要在代码里加入一些东西，监控一下上述代码中是否发生了 JVM FullGC。

如果发生了 JVM FullGC，就自动延长 expireTime 就可以了。

比如下面代码的改进：

```
// 设置一个观测jvm gc是否发生的时间
long startWatch = currentTime();

if(countAvailableResponse() > quorum) {
    break;
}

long now = currentTime();

if(now > expireTime) {

    // 这里就是通过当前时间和startWatch时间做一个对比，
    // 如果说超过了1秒钟，那么就一定是发生了JVM FullGC。
    // 因为理论上来说，startWatch到这段代码这里应该是微秒级别的
    if(currentTime() - startWatch > 1000) {

        // 如果发生了JVM FullGC，那么就直接把expireTime给
        // 加上JVM FullGC卡顿的时间，相当于是延长了expireTime
        expireTime += currentTime() - startWatch;

        // 直接进入下一轮循环来判断
        continue;
    }

    Logger.error("系统B集群故障，无法传输数据，系统A自动退出");
    System.exit(1);
}
```


通过上述代码的改进，就可以有效的优化线上系统的稳定性，保证其在 JVM FullGC 发生的情况下，也不会随意出现异常宕机退出的情况了。

【来自一线的血泪总结】你的系统上线时是否踩过这些坑?

> 作者:中华石杉 原文地址

> “ 之前我们写了很多线上生产实践类的文章，本文将对这些文章做一个简单的小结，同时也帮助大家回顾一下，希望大家能够跟着本文，温故知新，结合自己公司的业务项目，实际的落地这些方案，在项目上线时避开一些大坑。

首先我们来看微服务这块：**【双 11 狂欢的背后】微服务注册中心如何承载大型系统的千万级访问?**

这篇文章 focus 在 Spring Cloud 的注册中心，分析了注册表的底层存储结构、心跳机制、多级缓存机制！

更进一步，在了解了这些后，你再也不用去纠结 Eureka Server 到底要部署几台机器！再也不用担心你的 Eureka Server 能不能抗住一个大型系统的访问压力！再也不用自问自答，系统那么多服务，会对 Eureka Server 产生多大的访问压力！

接下来，是关于 Spring Cloud 的参数优化，本文基于一次真实的线上事故告诉你，面对高峰期每秒上万的并发请求，公司部署的 Spring Cloud 微服务架构应该做哪些优化?

当用户调用接口，好几秒都没响应，仅仅调大超时时间就够了吗?

如果你的公司遇到类似的业务场景和并发量，不妨看看，也许能让你避开雷区！

参见文章：**【性能优化之道】每秒上万并发下的 Spring Cloud 参数优化实战**

聊完高并发，怎能不聊聊高可用？同样，我们通过一篇文章，基于大量的一线生产经验总结，阐述了双 11 这样的高并发场景中，如何优化关键参数，从而最大限度的保障你的微服务架构系统的高可用。

设置哪些参数？设置参数为多少？为什么要这么设置？图文结合，步步为营，让你看完之后，能迅速在自己公司落地实践！

参见文章：微服务架构如何保障双 11 狂欢下的 99.99% 高可用?

好！说完了 Spring Cloud 微服务架构的一些落地实践案例，咱们来看看分布式事务和分布式锁在实际项目中的生产实践。

实际生产中，各个服务间的调用很可能是异步的。所以我们首先聊了聊基于 MQ 的异步调用如何保证各个服务间的分布式事务！详细阐述了用来实现分布式事务的可靠消息最终一致性方案的核心流程。

然后更进一步，深入剖析并指出了保障可靠消息最终一致性方案高可用的关键因素。

最后，通过一个真实的案例，给出了实际的保障 99.99% 高可用的解决方案，并且指出了其中可能存在的一些大坑。

参考文章：**【坑爹呀！】最终一致性分布式事务如何保障实际生产中 99.99% 高可用?**

接下来是分布式锁，我们通过一道真实的面试题引入：每秒上千订单场景下，如何对分布式锁的并发能力进行优化？

这个问题，无论是面试中，还是实际工作中，都是一个关于分布式锁的一个比较典型的问题。

如果有不清楚的同学，可以再好好复习一下，答案就在下面的文章中。

参考文章：**每秒上千订单场景下的分布式锁高并发优化实践**

接下来也是一个实际的高并发生产实践的问题，具体来说，是对于内存双缓冲 + 批量刷磁盘机制在 10 倍高并发访问场景下的优化实践。

通过一个实际的生产问题，看看从设计方案、埋下隐患、爆发问题、对症下药，这一整个心路历程。

参考文章：**【高并发优化实践】10 倍请求压力来袭，你的系统会被击垮吗？**

最后，我们来看看一次 JVM 垃圾回收导致的线上血案！感受一下造成系统不稳定的隐形杀手：JVM FullGC。

参考文章：**一次 JVM FullGC 的背后，竟隐藏着惊心动魄的线上生产事故！**

老司机生产实践经验：线上系统的 JVM 内存是越大越好吗？

> **作者:中华石杉** **原文地址**

> ! “ 这篇文章，给大家聊一个生产环境的实践经验：线上系统部署的时候，JVM 堆内存大小是越大越好吗？ 先说明白一个前提，本文主要讨论的是 Kafka 和 Elasticsearch 两种分布式系统的线上部署情况，不是普通的 Java 应用系统。

1、是否依赖 Java 系统自身内存处理数据？

先说明一点，不管是我们自己开发的 Java 应用系统，还是一些中间件系统，在实现的时候都需要选择是否基于自己 Java 进程的内存来处理数据。

大家应该都知道，Java、Scala 等编程语言底层依赖的都是 JVM，那么只要是使用 JVM，就可以考虑在 JVM 进程的内存中来放置大量的数据。

还是给大家举个例子，大家应该还记得之前聊过消息中间件系统。

比如说系统 A 可以给系统 B 发送一条消息，那么中间需要依赖一个消息中间件，系统 A 要先把消息发送到消息中间件，然后系统 B 从这个消息中间件消费到这条消息。

大家看下面的示意图。

大家应该都知道，一条消息发送到消息中间件之后，有一种处理方式，就是把这条数据先缓冲在自己的 JVM 内存里。

然后过一段时间之后，再从自己的内存刷新到磁盘上去，这样可以持久化保存这条消息，如下图。

2、依赖 Java 系统自身内存有什么缺陷

如果用类似上述的方式，依赖 Java 系统自身内存处理数据，比如说设计一个内存缓冲区，来缓冲住高并发写入的大量消息，那么是有其缺陷的。

最大的缺陷，其实就是 JVM 的 GC 问题，这个 GC 就是垃圾回收，这里简单说一下他是怎么回事。

大家可以想一下，如果一个 Java 进程里老是塞入很多的数据，这些数据都是用来缓冲在内存里的，但是过一会儿这些数据都会写入磁盘。

那么写入磁盘之后，这些数据还需要继续放在内存里吗?

明显是不需要的了，此时就会依托 JVM 垃圾回收机制，把内存里那些不需要的数据给回收掉，释放掉那些内存空间腾出来。

但是 JVM 垃圾回收的时候，有一种情况叫做 stop the world，就是他会停止你的工作线程，就专门让他进行垃圾回收。

这个时候，他在垃圾回收的时候，有可能你的这个中间件系统就运行不了了。

比如你发送请求给他，他可能都没法响应给你，因为他的接收请求的工作线程都停了，现在人家后台的垃圾回收线程正在回收垃圾对象。

大家看下图。

虽然说现在 JVM 的垃圾回收器一直在不断的演进和发展，从 CMS 到 G1，尽可能的在降低垃圾回收的时候的影响，减少工作线程的停顿。

但是你要是完全依赖 JVM 内存来管理大量的数据，那在垃圾回收的时候，或多或少总是有影响的。

所以特别是对于一些大数据系统，中间件系统，这个 JVM 的 GC（Garbage Collector，垃圾回收）问题，真是最头疼的一个问题。

3、优化为依赖 OS Cache 而不是 JVM

所以类似 Kafka、Elasticsearch 等分布式中间件系统，虽然也是基于 JVM 运行的，但是他们都选择了依赖 OS Cache 来管理大量的数据。

也就是说，是操作系统管理的内存缓冲，而不是依赖 JVM 自身内存来管理大量的数据。

具体来说，比如说 Kafka 吧，如果你写一条数据到 Kafka，他实际上会直接写入磁盘文件。

但是磁盘文件在写入之前其实会进入 os cache，也就是操作系统管理的内存空间，然后过一段时间，操作系统自己会选择把他的 os cache 的数据刷入磁盘。

然后后续在消费数据的时候，其实也会优先从 os cache（内存缓冲）里来读取数据。

相当于写数据和读数据都是依托于 os cache 来进行的，完全依托操作系统级别的内存区域来进行，读写性能都很高。

此外，还有另外一个好处，就是不要依托自身 JVM 来缓冲大量的数据，这样可以避免复杂而且耗时的 JVM 垃圾回收操作。

大家看下面的图，其实就是一个典型的 Kafka 的运行流程。

然后比如 Elasticsearch，他作为一个现在最流行的分布式搜索系统，也是采用类类似的机制。

大量的依赖 os cache 来缓冲大量的数据，然后在进行搜索和查询的时候，也可以优先从 os cache（内存区域）中读取数据，这样就可以保证非常高的读写性能。

4、老司机经验之谈：

依赖 os cache 的系统 JVM 内存越大越好?

所以现在就可以进入我们的主题了，那么比如就以上述说的 kafka、elasticsearch 等系统而言，在线上生产环境部署的时候，你知道他们是大量依赖于 os cache 来缓冲大量数据的。

那么，给他们分配 JVM 堆内存大小的时候是越大越好吗?

明显不是的，假如说你有一台机器，有 32GB 的内存，现在你如果在搞不清楚状况的情况下，要是傻傻的认为还是给 JVM 分配越大内存越好，此时比如给了 16G 的堆内存空间给 JVM，那么 os cache 剩下的内存，可能就不到 10GB 了，因为本身其他的程序还要占用几个 GB 的内存。

那如果是这样的话，就会导致你在写入磁盘的时候，os cache 能容纳的数据量很有限。

比如说一共有 20G 的数据要写入磁盘，现在就只有 10GB 的数据可以放在 os cache 里，然后另外 10GB 的数据就只能放在磁盘上。

此时在读取数据的时候，那么起码有一半的读取请求，必须从磁盘上去读了，没法从 os cache 里读，谁让你 os cache 里就只能放的下 10G 的一半大小的数据啊，另外一半都在磁盘里，这也是没办法的，如下图。

那此时你有一半的请求都是从磁盘上在读取数据，必然会导致性能很差。

所以很多人在用 Elasticsearch 的时候就是这样的一个问题，老是觉得 ES 读取速度慢，几个亿的数据写入 ES，读取的时候要好几秒。

那能不花费好几秒吗？你要是 ES 集群部署的时候，给 JVM 内存过大，给 os cache 留了几个 GB 的内存，导致几亿条数据大部分都在磁盘上，不在 os cache 里，最后读取的时候大量读磁盘，耗费个几秒钟是很正常的。

5、正确的做法：

针对场景合理给 os cache 更大内存

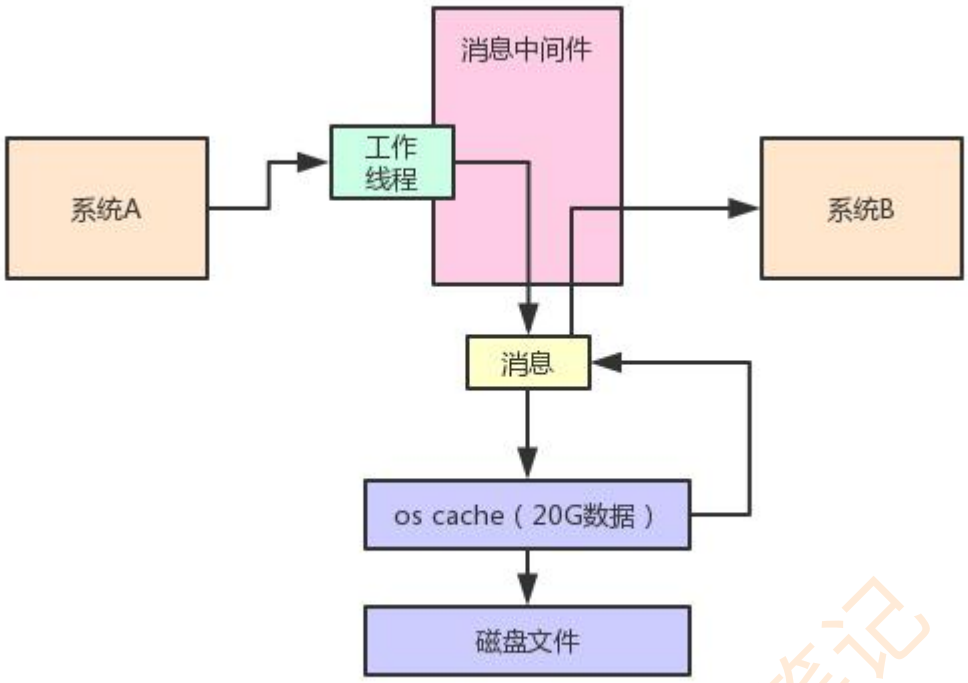
所以说，针对类似 Kafka、Elasticsearch 这种生产系统部署的时候，应该要给 JVM 比如 6GB 或者几个 GB 的内存就可以了。

因为他们可能不需要耗费过大的内存空间，不依赖 JVM 内存管理数据，当然具体是设置多少，需要你精准的压测和优化。

但是对于这类系统，应该给 os cache 留出来足够的内存空间，比如 32GB 内存的机器，完全可以给 os cache 留出来 20 多 G 的内存空间，那么此时假设你这台机器总共就写入了 20GB 的数据，就可以全部驻留在 os cache 里了。

然后后续在查询数据的时候，不就可以全部从 os cache 里读取数据了，完全依托内存来走，那你的性能必然是毫秒级的，不可能出现几秒钟才完成一个查询的情况。

整个过程，如下图所示：



所以说，建议大家在线上生产系统引入任何技术的时候，都应该先对这个技术的原理，甚至源码进行深入的理解，知道他具体的工作流程是什么，然后针对性的合理设计生产环境的部署方案，保证最佳的生产性能。

【嗅探底层】你知道Synchronized作用是同步加锁，可你知道它在JVM中是如何实现的吗?

> **作者:中华石杉** 原文地址

> **本文系公众号读者投稿 作者：李瑞杰 目前任职于阿里巴巴，资深 JVM 研究人员**

> ! 友情提示： 本文内容涉及 JVM 底层，文章烧脑，请谨慎阅读!

我们可以利用 synchronized 关键字来对程序进行加锁。它既可以用来声明一个 synchronized 代码块，也可以直接标记静态方法或者实例方法。

当谈到 synchronized 时，我们有必要了解字节码中的 monitorenter 和 monitorexit 指令。

这两种指令均会消耗操作数栈上的一个引用类型的元素（也就是 synchronized 关键字括号里的引用），作为所要加锁解锁的锁对象。

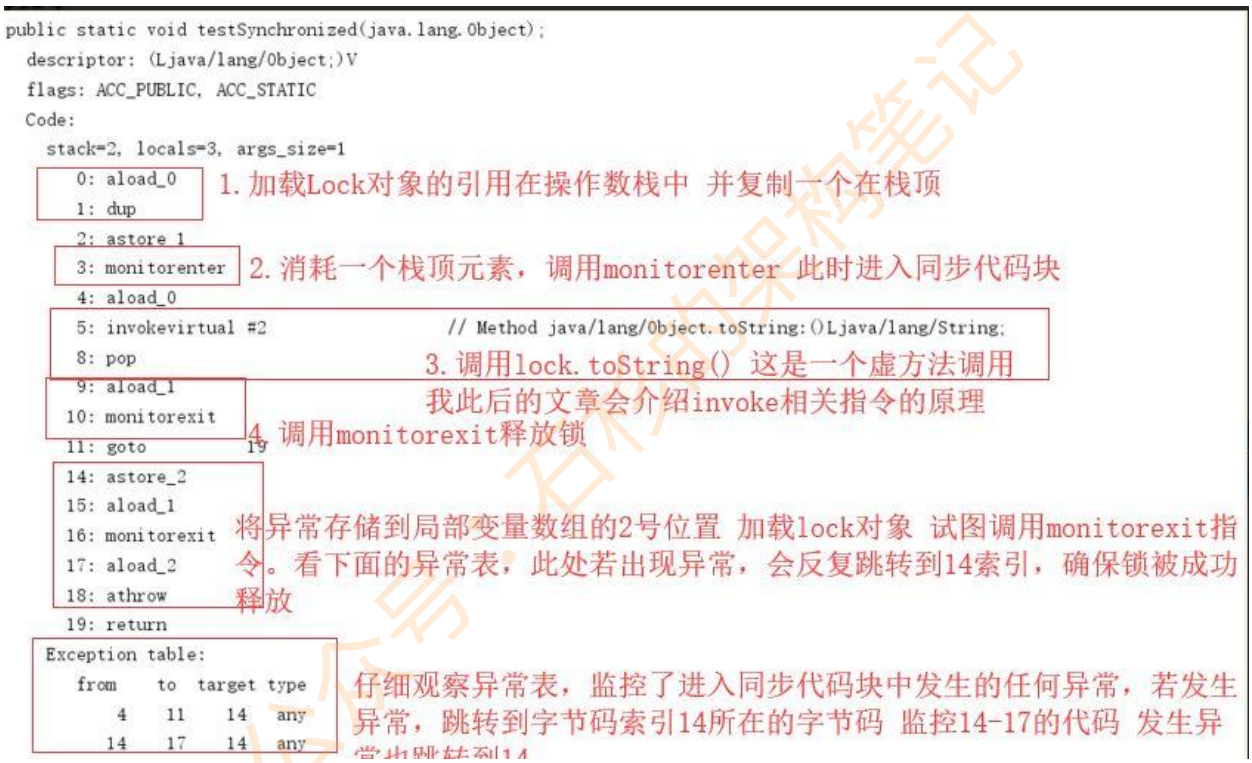
下面我们将深入了解 Synchronized 在 JVM 底层的实现原理。

考察以下的代码：

```
public static void testSynchronized(Object lock) {
    synchronized (lock) {
        lock.toString();
    }
}
```

查看这个代码编译后的字节码，我就直接用下面这张图解释了。

ps: 截图截得不太好，下面有点没截到，大家凑合看看：



你可能会留意到，上面的字节码中包含一个 monitorenter 指令以及多个 monitorexit 指令。

这是因为 Java 虚拟机需要确保所获得的锁在正常执行路径以及异常执行路径上都能够被解锁。

大家可以看我的注释，自己思考一下，应该都能看懂。

应该注意，如果用 synchronized 标记方法，你会看到字节码中方法的访问标记包括 ACC_SYNCHRONIZED。

该标记表示在进入该方法时，Java 虚拟机需要进行 monitorenter 操作。

而在退出该方法时，不管是正常返回，还是向调用者抛异常，Java 虚拟机均需要进行 monitorexit 操作。

用两张图一看就懂。

```
public synchronized void testSync() {
    //do something
}
```

```
public synchronized void testSync();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=0, locals=1, args_size=1
       0: return
    LineNumberTable:
      line 12: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       1     0  this   Lcom/jvm/research/SynchronizedImpl;
```

可以看到，在 0 号字节码处就返回了。

这里有人可能问了，这里没有调用 monitorenter 和 monitorexit 指令啊？怎么实现的加锁?

要注意，这里 monitorenter 和 monitorexit 操作所对应的锁对象是隐式的。

对于实例方法来说，这两个操作对应的锁对象是 this；对于静态方法来说，这两个操作对应的锁对象则是所在类的 Class 实例。

我们先来介绍 Synchronized 的重入的实现机理。

可以认为每个锁对象拥有一个锁计数器和一个指向持有该锁的线程的指针。

当执行 monitorenter 时，如果目标锁对象的计数器为 0，那么说明它没有被其他线程所持有。

Java 虚拟机会将该锁对象的持有线程设置为当前线程，并且将其计数器加 1。

在目标锁对象的计数器不为 0 的情况下，如果锁对象的持有线程是当前线程，那么 Java 虚拟机可以将其计数器加 1，否则需要等待，直至持有线程释放该锁。

当执行 monitorexit 时，Java 虚拟机则需将锁对象的计数器减 1。计数器为 0，代表锁已被释放。

这就是锁的重入的实现机理。

说完了这个实现机理，我们来探究具体的锁实现。

首先谈谈重量级锁，重量级锁是 Java 虚拟机中最为基础的锁实现。

在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。在 Linux 中，这是通过 pthread 库的互斥锁来实现的。

此外，这些操作将涉及系统调用，需要从操作系统的用户态切换至内核态，其开销非常之大。

为了尽量避免昂贵的线程阻塞、唤醒操作，Java 虚拟机会在线程进入阻塞状态之前，以及被唤醒后竞争不到锁的情况下，进入自旋状态，在处理器上空跑并且轮询锁是否被释放。

如果此时锁恰好被释放了，那么当前线程便无须进入阻塞状态，而是直接获得这把锁。

下面我将介绍自适应自旋的概念，刚才说了自旋是什么，但是自旋很耗费资源，所以我们可以根据以往自旋等待时是否能够获得锁，来动态调整自旋的时间（循环数目）。

所以 Synchronized 是否公平这个问题可以休矣，为什么呢?

处于阻塞状态的线程，并没有办法立刻竞争被释放的锁。然而，处于自旋状态的线程，则很有可能优先获得这把锁。所以 Synchronized 不是公平的。

我们再介绍轻量级锁，针对多个线程在不同的时间段请求同一把锁，也就是说没有锁竞争。

针对这种情形，Java 虚拟机采用了轻量级锁，来避免重量级锁的阻塞以及唤醒。

在介绍轻量级锁的原理之前，我们先来了解一下 **Java 虚拟机是怎么区分轻量级锁和重量级锁的。**

简单的说，对象头中有一个标记字段。它的最后两位便被用来表示该对象的锁状态，其中：

- 00 代表轻量级锁
- 01 代表无锁（或偏向锁）
- 10 代表重量级锁
- 11 则跟垃圾回收算法的标记有关。

当进行加锁操作时，Java 虚拟机会判断是否已经是重量级锁。

如果不是，它会在当前线程的当前栈桢中划出一块空间，作为该锁的锁记录，并且将锁对象的标记字段复制到该锁记录中。

然后，Java 虚拟机会尝试用 CAS 操作替换锁对象的标记字段。

各位有兴趣可以了解一下 JVM 的 CAS 在 X86 机器上的实现，是汇编指令 lock cmpxhcg。

这里我简单介绍一下，CAS 是一个原子操作，它会比较目标地址的值是否和期望值相等，如果相等，则替换为一个新的值。

假设当前锁对象的标记字段为 X...XYZ，Java 虚拟机会比较该字段是否为 X...X01。

如果是，则替换为刚才分配的锁记录的地址。由于内存对齐的缘故，它的最后两位为 00。此时，该线程已成功获得这把锁，可以继续执行了。

如果不是 X...X01，那么有两种可能：

- 第一，该线程重复获取同一把锁。此时，Java 虚拟机会将 0 加入锁记录，以代表该锁被重复获取。
- 第二，其他线程持有该锁。此时，Java 虚拟机会将这把锁膨胀为重量级锁，并且阻塞当前线程。

你可以将一个线程的所有锁记录想象成一个栈结构，每次加锁压入一条锁记录，解锁弹出一条锁记录，当前锁记录指的便是栈顶的锁记录。

当进行解锁操作时，如果当前锁记录的值为 0，则代表重复进入同一把锁，直接返回即可。

若当前锁记录不是 0，Java 虚拟机会尝试用 CAS 操作，比较锁对象的标记字段的值是否为当前锁记录的地址。

如果是，则替换为锁记录中的值，也就是锁对象原本的标记字段。此时，该线程已经成功释放这把锁。

如果不是，则意味着这把锁已经被膨胀为重量级锁。此时，Java 虚拟机会进入重量级锁的释放过程，唤醒因竞争该锁而被阻塞了的线程。

下面我们介绍偏向锁，偏向锁针对的是从始至终只有一个线程请求某一把锁。是轻量级锁的更进一步的乐观情况。

在线程进行加锁时，如果该锁对象支持偏向锁，那么 Java 虚拟机会通过 CAS 操作，将当前线程的地址记录在锁对象的标记字段之中，并且将标记字段的最后三位设置为 101。

这里介绍一下 epoch 的概念，每个类中维护一个 epoch 值，你可以理解为这个类所有实例对象的第几代偏向锁。

当设置偏向锁时，Java 虚拟机需要将该 epoch 值复制到锁对象的标记字段中。我们规定，你加的偏向锁的代数高，是可以把代数低的 PK 下去的。

接下来我给你讲的过程，你就知道为什么要这么设计了。

我们先从偏向锁的撤销讲起。

当请求加锁的线程和锁对象标记字段保持的线程地址不匹配时（而且 epoch 即代数必须相等，如若不等，那么当前线程可以将该锁重偏向至自己，因为新的 epoch 的代数肯定要高于以前的代数），Java 虚拟机需要撤销该偏向锁。

这个撤销过程非常麻烦，它要求持有偏向锁的线程到达安全点，再将偏向锁替换成轻量级锁。

在宣布某个类的偏向锁失效时，Java 虚拟机实则将该类的 epoch 值加 1，表示之前那一代的偏向锁已经失效。而新设置的偏向锁则需要使用类中的最新 epoch 代数来加锁。

为了保证当前持有偏向锁并且已加锁的线程不至于因此丢锁，Java 虚拟机需要遍历所有线程的 Java 栈，找出该类已加锁的实例，并且将它们标记字段中的 epoch 值加 1。该操作需要所有线程处于安全点状态。

所以有专家近年来提出，偏向锁在锁竞争激烈的情况下，非但不能优化性能，反而可能伤害应用性能。

如果总撤销数超过另一个阈值（对应 Java 虚拟机参数 -XX:BiasedLockingBulkRevokeThreshold，默认值为 40），那么 Java 虚拟机会认为这个类已经不再适合偏向锁。

此时，Java 虚拟机会撤销该类实例的偏向锁，并且在之后的加锁过程中直接为该类实例设置轻量级锁。

【非广告，纯干货】英语差的程序员如何才能无障碍阅读官方文档?

> **作者:中华石杉** 原文地址

目录

> ! 这篇文章不聊技术，我们来聊一个某种程度上比技术更重要的话题：一个英语比较渣的程序员，到底应该如何做，才能达到无障碍阅读英文官方文档呢?

首先声明一点，现在很多公众号会用类似的标题给一些学英语的机构打广告，这也无可厚非。但是强调一下，这篇文章绝对不是广告。

笔者写作本文的动机，主要是因为很多程序员朋友，尤其是刚入行不久的程序员小兄弟，留言说自己的英语底子太差了，但是又想阅读相关英文官方文档。毕竟，官方文档是入门、熟悉、掌握一个技术，最权威的第一手资料。

但是这些同学因为英语基础差，往往读起来效率低下、举步维艰，最后不得不放弃。

所以我们就来聊聊，英语较差的程序员兄弟，应该如何提升自己阅读英文官方文档的能力。这都是笔者在公司指导一些下属阅读官方文档的思考和感悟。

（1）笔者英文基础介绍

简单说一下我自己的英文基础，大概 2000 年初的时候，我就在国外留学读研，就英语的听说读写四大块能力而言，口语虽然带有中国人的口音（这是各国人几乎不可避免的），但是听力和口语跟老外正常工作和生活中的交流，是没问题的。

阅读能力，则是在国外期间大量锻炼出来，近十多年都保持着很高的水准。

如果拿 redis、elasticsearch、spark 之类的英文官方文档给我看，基本上看英文文档和看中文文档是差不多的速度和流畅度。

而且，因为少年的时候进行过速读能力的锻炼，所以阅读英文文档的速度十分的快，大概是普通人的 5 倍~ 10 倍的速度。

所以，这里基于自身经历以及给其他朋友的一些英文指导，聊聊我对程序员看英文文档的一些看法。

（2）为啥程序员需要阅读官方文档?

首先说一下，我们为啥建议程序员一定要去自己阅读英文官方文档呢?

很简单一个道理，假设现在某技术很火，比如最近大家在聊微服务架构中的 service mesh，这个最流行的开源技术就是 istio。

好，从国外一个技术开始火，一直到有大量的中文资料出来，这个过程各位如果观察一下就会发现，大致需要 1~2 年的时间。

假如你看不懂 istio 的英文官方文档，就要一直等着一些技术的中文资料，那么可能需要等个 1 年，才能等到一本国外翻译过来的书籍。

然后再等个 2 年，才有很多中国人自己写的相关的技术书籍，然后网上的博客之类的才会开始变得很多。

所以，你看不懂英文官方文档，那么自身对最新流行技术的掌握，大致比最优秀的一批国内工程师，至少要晚个 2 ~ 3 年。

对于技术而言，2 ~ 3 年的落后，肯定是不短的一个时间，这会导致你不能成为第一批吃螃蟹的人。

这只是一方面。此外，即使你去看一些国外的书翻译过来的，你难道不觉得那种翻译的书很多语言较为生硬，理解起来有点别扭么？

我本人是从来不看翻译的书籍的，虽然我觉得翻译英文书籍是一个非常值得尊敬的职业，但是我自己个人而言，觉得效果更好的还是直接看官方文档。

因为官方文档里的描述，让人读起来觉得非常顺畅和舒服。而且老外很多语言表达是相当优雅的，并且他对这个技术的理解比其他人深刻的多。

举个例子，如果你想学习 Redis，直接去看 Redis 作者写的文档。他作为 Redis 的作者，可以说是这个世界上对该技术理解最深的人。而理解的越深，就更容易把他用通俗易懂的语言描述出来，易于别人理解。

再其次，如果你对技术的学习，仅仅就是读一些书籍的话，要知道，书籍都是按照一个版本来写的，比如 1.0 版本，那如果你按照书里学了 1.0 版本的东西，结果人家官网更新到了 2.0 版本呢？

这个时候版本升级，跟书里的东西都不一样了，你不是傻眼了？

所以说，直接看英文官方文档，首先可以让你对最新的技术第一批掌握，最早上车。

其次，可以直接跟上人家的版本更新，每个小版本的发布人家都是有 release 的，里面你都可以看到他有哪些细小或者巨大的变动。

（3）如何才能无障碍阅读英文文档？

就我个人角度而言，我认为最终提升自己阅读英文文档的能力，还是脚踏实地。没什么捷径，就两个关键点：积累和坚持。

你要做的事情，就一个：每天至少抽半个小时，就挑选一个技术的英文文档，强迫自己，从头开始读，一点一点读。

这个时间不需要定太久，就半小时，太久了你坚持不下来。

半小时，就是你玩几局王者，吃几次鸡的时间，你少吃一次鸡，少坑两次队友，人家不会怪你。

给大家举个例子，我随便从 istio 这个技术的官方文档里抽取了一小段出来，我们就来看看这一小段好了。

> **This page provides an overview of how traffic management works in Istio, including the benefits of its traffic management principles. It assumes that you've already read What is Istio? and are familiar with Istio's high-level architecture.**

我自己读这段英文的感觉，就跟中文一样。但是对英文基础不好的码农来说，刚开始尝试读文档，会发现很多单词都不认识。

比如说 “this page provides” ，这个你只要有过初中英文水平的都会看懂，大致意思是 “这一页提供了”，或者中文方式的口语化一些就是 “现在这篇文章主要是讲一下”，这样一个意思。

但是你接着会看到 “overview”，很多人就不懂了，overview 是什么？

这个时候你肯定会查词典，这个很正常。这个单词就是一个 “概览，概述” 的意思。

然后你就应该自己准备一个生词本。记录下来自己每天学习到的每个生僻的单词和对应的中文意思。

接着会看到 “traffic management”，其实就是“流量管理” 的意思，你可能不理解，那就查字典，积累生僻单词，以此类推。

刚开始，半个小时你可能就读一小段，几十个单词，因为大量的单词你都不认识。

但是这是一个积累的过程，实际上语言的学习是很困难的，积累几个月开始有点小感觉，积累一两年有所小成，积累三五年就大成了。

（4）坚持！坚持！坚持！

大家可以每天都读半小时英文文档，而且每天积累生词、并快速过一遍之前的旧词。

下一次如果你又读到了相同的生词，可能还是不能反应过来。没关系，再次强化巩固，三次四次五次。。。

最后，直到你一眼看上去，立马反应出其意思，这中间几乎没有思考停顿。那么恭喜你，你的大脑已经接纳他了，此时可以把他从生词本里删除了。

而且，一个单词你潜移默化的用的越多，你的印象就会越深刻。用进废退，这个是咱们大脑生理结构决定的。

想一想，像 what、when、where 这些单词，是不是已经成了你脑中的一部分，因为你已经无形中使用它们太多次了。

慢慢的，你会发现，越是到了后面，你开始慢慢能读懂一句完整的话了，慢慢能读懂一段话了，再慢慢一篇英文文章，都能看懂了。

其实如果你真的看过几十种技术的官方文档，你会发现技术领域的官方文档，常见的英文词汇可能也就那么几千个。

所以你如果坚持一直读英文文档的话，你会发现每天都在通过实战锤炼你的阅读能力，而且经常一个单词在很多地方反复的出现，这就给了你反复强化记忆的一个机会，通过各种地方，多次看见他，你的记忆会非常非常的深刻。

这样你一直坚持，几年后，那几千常见技术英文词汇，会跟中文一样烙印在你的脑子里了，想想刚提到的 when、where、what，想想是不是这个道理。

哪怕你参加了一些英文学习的培训课程，那个会提供很多技巧给你，但是最终你也需要类似这里说的每天坚持看，大量的实战阅读，大量的重复记忆来进行强化。

这里关键最难做到的一点，就是你要把每天看英文文档变成一种习惯，日积月累，水滴石穿。

几个月、1 年、几年过后，你会发现，阅读英文文档就跟喝水一样。

那个时候，你通过几年的坚持，已经对数千个技术文档里常见的高频词汇进行了大量的训练和强化记忆。那时你读英文文档，基本跟你读中文的书籍是一样的感觉了。

这里笔者再次给英语阅读有困难的新鸟老鸟都强调一下，也算打打鸡血：

> **“重剑无锋、大巧不工”，英文阅读，跟技术积累一样，是内功修炼的过程，绝没有所谓的一步登天。 请各位记住：坚持！是一个人最难得的天赋。 纵观各路翘楚，足球 C 罗、篮球科比，论天资绝不是最出色的，但是依靠多年的坚持，强大的意志力。在各自领域同样取得了非凡成就。**

（5）来个约定吧

上面就是关于英文文档阅读的经验介绍，这里面有自身的感悟，也有在公司指导下属积累的经验和收获的成功案例。

最后，和大伙儿来个约定：希望大家能照着这个方法坚持两年，持之以恒、风雨无阻。

在最想放弃的时候，扪心自问：

我真的不想体验一把英文文档读起来像看中文一样行云流水的快感吗？

【非广告，纯干货】中小公司的Java工程师应该如何逆袭冲进BAT？

> **作者:中华石杉** [原文地址]

目录

（1）80% Java 工程师都有的迷茫

> **（2）你的技术为啥十年八年都无法进步？ 先来搞清楚一个问题，你的技术到底为什么十年八年都无法进步?**

拆解一下，你的能力集中在哪几块：

这篇文章，跟大家聊一聊很多很多很多人问我的一个问题：中小公司的 Java 工程师应该如何规划准备，才能跳槽进入 BAT 这类一线互联网公司?

之所以我用了三个 “很多” 来形容这个问题，是因为实在这个问题太普遍了，因为国内 Java 工程师至少好几十万，但是在国内互联网大厂里干过的码农可能也就十分之一，或者五分之一的比例。

所以，其实这个也是符合 28 法则的，少部分人在大厂里干过，发展的很好。但是大部分人还是在中小型公司，或者外包类传统 IT 公司里工作。

这些同学可能对自己的技术成长，职业发展感到非常的迷茫，自己有点追求，也想去一下大厂，但是又不知道怎么规划。

开始本文之前，同样强调一点：像这种讲职业规划发展的文章，有可能很多同学会以为是广告。但是这篇文章，笔者特意声明：纯干货、非广告。

因为我个人在国内几个最大的互联网公司先后有着十余年工作经历，面试和招聘过大量各种水平的开发人员。包括初、中、高级开发，技术专家，高级技术专家，都面过。

同样，也指导过很多同学的职业发展规划，看过大量的同学不顺利的职业发展，所以打算从我个人的角度来聊聊这个问题：中小公司的同学应该如何一步一步实现逆袭进入 BAT。

我相信以下情形很多同学应该都有类似体会：一直徘徊在各种中小公司里开发一些没技术难度的 Java 系统，主要就是 CRUD。

哪怕是用了用 MQ、缓存、分库分表，但是也没什么并发量，数据量也不算特别大，自己的技术成长极为缓慢。

然后就是三五年，七八年，甚至十多年，职业发展和技术水平都停滞在这个状态，无法有更进一步的发展。

随着现在寒冬到来，到处裁员，中年码农的危机，加不动班，体力越来越差，孩子压力越来越大，对自己何去何从很迷茫。

有一些同学是一直徘徊在那种中小型互联网公司里碰到上述情况，有一些同学是在一些外包类的 IT 公司里碰到上述情况。

（2）你的技术为啥十年八年都无法进步?

先来搞清楚一个问题，你的技术到底为什么十年八年都无法进步?

拆解一下，你的能力集中在哪几块：

> ! 技术广度

- 对 MQ、缓存、NoSQL、大数据、高并发、高可用、微服务，等一系列的相关技术都有一定的了解，熟悉常见功能
- 在自己的项目里落地使用过，有一定的技术使用经验

> **这可以解释为技术广度。**

> ! 技术深度

- 读过 Kafka 的底层源码?
- 对消息中间件的架构设计思想有深刻的理解?
- 对分布式事务框架 / 中间件的架构设计有过研究?
- 在每秒百万并发场景下做过底层系统的深入优化和故障处理?

如果你有类似这种过人之处，那么你才能说你有某些技术深度。

> ! 项目经验

- 你有没有整体负责过几亿注册用户，几千万日活用户的大规模、高并发、分布式、高可用、高复杂度的系统架构设计?
- 或者你负责的一直都是那种公司内部使用的，几十个人用的 OA 系统，CRM 系统?

这些就是你的项目经验

> ! 团队管理

- 你在互联网公司里带过 20 的团队?
- 或者你在一个传统 IT 公司里带过 3 个人的小组?

这都是你的团队管理经验。

拆解过后，再来看看，如果你在一些小型互联网公司，或者是做一些传统软件开发，为什么技术无法进步?

其实道理很简单，可能你的公司推出了一款 APP，但是不好意思，用户量总共就 100 万，日活用户就 10 万人。

那你觉得你的系统有技术挑战吗？没有。

既然没有技术挑战，你把系统搞那么复杂干嘛？或者你的架构师搞那么复杂干嘛？不需要。

大家简单做一做，主要 crud 写一下功能，最多现在 spring cloud 流行了，上一下拆成微服务的就够了。

然后这套系统就稳定支撑你公司的业务了，那你接触不到很大的技术挑战，所以技术进入停滞状态，不是很正常么?

或者你做一些传统的软件开发，比如说建筑类软件，办公自动化软件，类似这种的。总共就几十个人用一个系统，或者几百人用，那你就更是如此了。

可能都不需要 spring cloud，直接单块系统，单机部署，就是在里面填充业务代码就好了。

所以根本原因，就是很多同学平时的工作环境，他没有什么技术挑战，所以只要把系统技术做的简单一些，低成本就可以支撑公司业务了，那既然这样，当然技术就进展很缓慢了。

然后可能你工作了八年十年，技术广度还可以，对流行的技术自己都看过一些书，简单用过，玩过 demo。

你的项目经验积累了不少，但是都是一些各个传统领域的系统业务理解较为深刻，没有极高技术挑战的项目经验。

有的人工作时间长，可能就是带过一些人，有过一些带团队的经验，能管人。

大概就是如此了，每次换工作，还是只能换类似的公司，干类似的技术，依然没有进步，依然是类似的项目经验。

所以大伙儿先梳理清楚，迷茫的根源究竟在哪里。

（3）追求卓越，自己设立技术挑战

通常来说，我个人站在公司角度是很反对架构的过度设计的，因为平白浪费很多时间，而且很多架构过度复杂没有必要。

但是如果是站在个人的职业发展角度而言，那么你的 leader 必须要有对技术追求卓越的思维。或者你是 leader 的话，就得有对你的团队技术追求卓越的品质。

什么叫追求卓越呢?

举个例子，现在你开发了一款办公自动化系统，服务了某个公司，几百人在用，那么技术一般，就是一个单块系统，直接 Spring MVC + Spring + MyBatis 就搞定了。大家都做着没意思。

> **好，现在 leader 为了大家的幸福和未来，咬咬牙说：**

Leader

兄弟们，现在系统满足公司的发展了，但是我们不如来大胆的追求一下卓越。

兄弟们

领导你是啥意思啊？？

Leader

这样，咱们首先为了提高系统的开发效率，避免30个兄弟开发一个单块系统效率太低，我们来实践一把最流行的微服务架构吧。

咱们一起来把系统重构成微服务的架构，把spring cloud整套东西都用进去。

兄弟们

（认真听着）

Leader

咱们先得做技术调研，小A你来研究研究Spring Cloud核心技术组件，小B你来研究研究配置中心，小C你来研究研究服务链路追踪，等等。

大家分头行动，都开始学起来新技术。但是呢，咱们平时已经很忙了，要是占用工作时间搞这个，老板会骂人，大家看，每个人每天额外加班抽2小时一起来搞一下，怎么样？

兄弟们

领导，为了大家的幸福，那肯定得赶紧上新技术啊，大家都学点新东西。

最后大家辛苦了2个月，一起把系统重构成了整套的微服务架构，每个人都学到了东西，领导也学到了微服务整体架构设计的能力。

Leader

兄弟们，还想不想继续为未来的幸福努力一下？

兄弟们

一切都听领导安排。

Leader

现在这破系统就几百人用，忒没意思了，咱们来大胆想象，假如说以后这个系统要做成SaaS云产品，会有几百个公司来用，有几万人，甚至几十万人同时使用一套后台系统。大伙想想，那时会怎么样？

兄弟们

贫穷限制了我的想象力。。。。

Leader

那小A你去根据现在系统的访问量估算一下，如果有几十万人用，系统每天的并发量会有多少，数据库能不能支撑住，需要用哪些高并发的技术来支撑？

小B，你去调研一下，如果有几十万人用，我们会存储多少数据量，性能会有多差，怎么支撑海量数据存储？然后看看用什么技术来支撑一下？

兄弟们

好，领导一句话，上刀山、下火海。

几个月后，大家研发了一套系统，完成了测试，系统集成了缓存集群、MQ 集群、分库分表技术，还有很多其他的一些东西。

这个时候领导就想办法了，能不能跟老板建议一下，咱们就把产品做成 SaaS 云的模式呢？然后是不是可以就把这套系统给部署到生产环境里去？

到此为止，就通过一个例子，给大家阐述了一下，自己在公司里，如何通过想办法不断的追求系统的卓越，提高研发效率，支撑也许可能会存在的更高的并发量，更多的数据量，尽可能把系统做的更好一些。

多想想为了解决自己设想的一些技术挑战，如何尽可能把一些业界常用的技术都学习一下，比如缓存、消息、分布式、微服务、大数据，等等。

然后都尽可能进行相关的实践，积累相关的项目经验。

实际每个人在落地的这个过程的时候，方式肯定是不一样的：有的人也许人微言轻，只能对自己负责的模块设想一些技术挑战，然后只能自己在本地拉一个公司代码分支，尝试对这些分支加入一些技术，自己练习思考。

还有的人，可能是个小 leader，无法左右公司产品发展方向，但是可以尽可能跟上级沟通，阐述自己对系统架构追求卓越的一些构想。

然后，争取到一些时间，尽可能把系统多融入一些技术，给做的好一些。

每个人都有每个人的方式，但是归根到底一句话：如果你本身工作没有技术挑战，那么尽可能多给自己设立一些挑战，多学一些技术，多做一些尝试和实践。

这总是可以尽可能帮助你在一定程度上提高技术，扩展技术知识的。

在这个阶段，我见过最多的人犯的最大的一个错误就是：觉得自己这样倒腾一些技术是没用的，没有实际的真正的经验。

然后他们着急忙慌，心浮气躁，自怨自艾，总想着必须得先进一个好的公司，才能锻炼技术。

实际上，这是一种很浮躁的想法，你要进好的公司锻炼，你必须先打磨一下自己的技术，然后才能有资本去一家更好的公司。

（4）幻想一步登天？那只是你的黄粱美梦

很多人多学了一些技术，有了一些经验，很容易开始有点膨胀，老是想着一步登天，一下子就进入 BAT。

关于这个，其实是有类似的一些成功经历，比如有的人是大专学历，通过自己的努力学习，加上一些机缘巧合，直接一下子就从中小公司跳入了 BAT。

但是就现实情况来看，不是每个人都一定可以一步登天，复制这个经历的。

在你学习了一些技术，同时自己多做了一些尝试，积累了一定的经验之后，此时应该做的是：做最坏的打算，抱最好的希望。

你完全可以去试试 BAT 的面试，TMD 的面试，尽可能去争取机会，但是如果没面上也无所谓。

你可以降低期望，人只要跟自己比就好了。

比如说你现在在某小型的传统外包软件公司，那么接下来如果你能面进一家小型创业互联网公司，有个几百万用户量，日活用户有几十万，比之前的公司技术挑战多一些，用的技术也更多一些，那么你就可以去了。

只要你每一步跳槽，都比之前好，都让自己有进步，那么整体的大方向就是没错的。

也许你先进一个创业型互联网公司，然后下一家就可以进入一个市值几十亿美金的上市互联网公司，再下一步就可以进入 BAT。

在这个阶段我见过很多人犯的最大的错误就是：老是觉得自己刚学了一点东西，就必须立马进大公司。

这些同学往往心态着急的不行，而忽略了自己的学历、背景、经验导致了起点较低。能立马进 BAT 当然很好，但是有时候机缘巧合缘分没到，进不去也正常。

在这个阶段最需要做的，就是跟自己比，别跟别人比，只要每一次跳槽都比上一次好，公司更大，薪资更高，职位更高，技术挑战更大，就可以了。

（5）不断提升自己，最后进入 BAT

一旦你开始做到跳槽进入一家比之前更好的公司，有更高的技术挑战，那么公司本身的技术挑战就会促使你快速成长，还是举个例子吧。

比如说你本来就在做传统软件的开发，用的都是单块系统涉及的一些技术，就是简单的 spring mvc、spring、mybatis 等技术做 CRUD 的业务开发。

但是呢，你通过追求卓越，自己业余不停的学习技术，然后对自己负责的一些模块多设立了一些技术挑战，自己构思了很多更高挑战的场景下，可以给自己的模块加入哪些更高阶的技术。

接着你带着自己学习的一些技术，还有积累的一些实践经验和思考，进入了一家创业型互联网公司。

这家公司的好处就在于，互联网公司技术氛围更好，比如 zookeeper、redis、rocketmq、sharding-jdbc，等各种技术，在公司生产环境的系统里，都有落地和使用。

那么你此时是不是就不用停留于一些技术挑战的构思，可以开始真正做一些有点技术挑战的工作了。

但是，此时你还是需要继续不停的学习技术，学习更多的架构上需要的技术，深入的学习技术，同时积累实践经验。

然后带着这份工作经历，同时加上你不断加强的技术学习，你进入了一家比如 30 亿美金估值的独角兽公司。

这家公司有 2000 万用户，日活用户达到百万级，高峰并发量可以过万，每天数据库里日增数据量达到了数十万。

此时你开始真正接触一些所谓的：高并发、高可用、高性能、海量数据的实际处理。

基于你开发的业务系统，你开始更多的实践，同时你还对各种涉及到的技术有了更加深入的研究，比如对一些核心中间件系统进行了源码级别的阅读和研究。

最后你终于等到一个机会，BAT 里某家公司让你去面试，经历了四五轮面试之后，对方给了你一个 offer，是年薪 40 万的高级 Java 工程师的职位。

然后你进去之后，可以在最顶尖的互联网公司里学习开发流程、规范、架构，接触到最大规模的用户量，每天都有解决不完的技术挑战，在这个过程中，你又可以继续成长。

最后可能你再次跳槽，就可以进入 TMD 中某一家，拿下技术专家的 offer，在大公司里拿下技术专家的职位，带一个团队，达到人生第一个巅峰。

接着你再跳槽，可能一些创业公司就开始挖你去做一些技术管理层。

大家别以为这个仅仅是笔者捏造的一个故事，在笔者指导过的同学中，确实有同学按照这个路线，实现了人生的逆袭！

（6）最后的寄语

最后，送大家一句话：**九层之台，始于垒土；千里之行，始于足下。**

这里面最难的就是开始的那一步，也就是大量的人都停留在一些完全没太多技术含量的技术工作的情况下，这个时候是最难熬的。

其实只要能把第一步走好，自己拼命的积累技术，努力跟其他工程师竞争，技术远超跟自己同情况的其他工程师，那么你就有机会率先脱离这种困境，开始慢慢第二步，第三步。

到了后面，就是让公司的技术挑战 push 你不断努力和进步，最后进入 BAT 这类一线互联网公司。

【码农打怪升级之路】行走江湖，你需要解锁哪些技能包？

> **作者:中华石杉** 原文地址

> ！“ 年后就是金三银四跳槽季，相信很多同学都摩拳擦掌，跃跃欲试，之前我们也聊过一些关于程序员职场跳槽和发展的文章。

今天，我们就来做个简单的年前总结，帮各位在战略层面梳理一下思路，我们来看看在码农打怪升级的路上，有哪些需要解锁的技能，有哪些需要避开的雷区！

首先，作为码农的立身之本，首当其中的肯定是技术的考察。对于中大型的互联网公司，面试的热门技术大概包括消息中间件、分布式缓存、分布式搜索、NoSQL、海量数据、高并发、高可用、数据库、JVM、数据结构和算法。

上述问题，不一定一次面试都会涉及，但你作为面试准备，肯定要都有所了解，不能存在技术盲点。

你最好能在广泛掌握上述技术的基础上，深入研究过其中的一两个技术，比如你深入阅读过 kafka、mq 的源码，甚至在其基础上做过二次开发，这个会成为你的技术亮点。

这些东西是你面试高级 / 资深 Java 工程师时，在技术方面需要有的知识储备，并且这些技术绝不会说抽离出来单独的进行面试。而是结合你的项目痛点，步步深挖。

到底你的技术解决了项目中的哪些问题，不用这个技术会有什么后果，这些，才是面试官真正想要听到的东西。

之前咱们写过一篇文章《互联网公司的面试官是如何 360° 无死角考察候选人的？（上篇）》，里面对这些东西都有详细的阐述。

大家可以随着上面的总结，再结合这篇文章，重新的温习一下。

此外，笔者有一套较为详细的针对 Java 进阶面试的免费学习资料：**《Java 面试突击第一季》，大家也可以参考学习一下，直接在公众号里回复：“学习”领取。**

如果充分消化吸收，对各位同学的技术广度，也是一个质的飞跃。

除了技术广度、技术深度的考察，还有非常关键的一点，就是你的项目经验。因为从你 hold 住的项目，就能看出你具备什么样的能力。

如果你面试的是中级岗

那么可能你技术整体 ok，独立负责过核心模块的开发，同时对各种技术都有一定的实践经验，就可以了。

如果你面的是高级 / 资深岗

那么会你是否带领一个小团队独立负责过一个有一定复杂度和难度的完整系统的架构设计和开发。

如果你面试的是架构师的岗位

那么你必须在一个公司里主导过很多人协作完成的大型而且复杂的项目群的开发。并且要求你对一个大型系统架构有深度的思考和整体的把控，而且这个项目要有足够的技术挑战，大用户量、高并发、海量数据，等等。

所以，项目经验，重中之重。大家平时一定要注意项目经验的积累。

对于做过的项目中出现过的痛点，在线上踩过的坑，对这些问题的解决方案，都可以予以记录。

这样在面试官面前，你会很自信，答的口若悬河，毕竟是亲身经历过的东西。

除此以外，对于高级工程师 / 技术专家的岗位，还有一个重要的考察点：系统设计能力，这个很可能面试官直接抛出自己公司的业务，当场叫你设计方案。

这个如何准备呢？ 其实就是在平时自己做的项目中，不断 push 自己，假想自己的项目有 10 倍 100 倍并发量，然后 push 自己去思考去实践，去解决这些问题。

只有这样，面对一个全新的业务的系统设计，你才有思路和面试官侃侃而谈，不至于说大眼瞪小眼。

另外，社招一个比较容易忽视的问题，就是对于数据结构和算法的掌握。

笔者不止一次的收到读者留言，说自己面试大厂倒在了一两道算法题目上，非常可惜。

其实关于这个，也没什么秘诀，就是平时的积累。

首先，社招的算法题目不会考的太难，大家可以去刷题网站 LeetCode 或者 Lintcode。

后者是中文，可能看起来更加友好，每天坚持刷一道题目，简单和中等难度的就行。

因为对于社招的技术考察，主要还是项目经验、线上问题解决，算法的话一般就是掌握基本的算法就 OK 了。

但是，如果你连二分查找、快速排序、反转链表这些东西都写的磕磕碰碰，bug 一堆的话，那么留给面试官的印象会很差。面试官甚至会怀疑你的计算机基础功底。

所以这方面，平时还是得坚持练习，对于基本的各种算法和数据结构，能够达到在白板上手写出来并且没有 bug，那就算是真正过关了。

之前咱们写过一篇文章**《互联网公司的面试官是如何 360° 无死角考察候选人的？（下篇）》**，对上述内容有一个更加详细的阐述。

【非广告，纯干货】三四十岁的大龄程序员，应该如何保持自己的职场竞争力？

> **作者:中华石杉** **原文地址**

目录

- 4、最后的寄语

> “ 这篇文章，给大家聊聊 Java 工程师的职业发展规划的一些思考，同时也给不少 20 多岁、30 多岁，但是对自己的职业未来很迷茫的同学一些建议。

笔者希望通过此文，帮大家梳理一下程序员的职业发展方向，让大家知道自己从 20 多岁的初出茅庐，到 40 岁的大龄码农，应该如何规划属于咱们程序员的半生。

首先，咱们通过倒推的方式，看看在一个程序员 40 岁的时候，你凭什么来捍卫自己的核心竞争力？

那如果要搞清楚这个问题，又得从一个反面来看看，大部分的程序员在职业发展过程中犯了哪些错误，结果导致在 40 岁的时候没有竞争力了呢？

一、40 岁回首往事：自己竟没有任何核心优势

最尴尬的事情，莫过于一个程序员在 10 多年，甚至 20 年的从业经历中，一直没好好考虑过如何构建自己的核心竞争力。

如果长年如此，会导致他就跟着公司慢吞吞的走着，就像温水煮青蛙，直到 40 岁的时候，突然发现自己几乎一无所长。

举个例子，比如从技术角度而言，这些兄弟可能发现自己不是什么技术专家，也不是架构师，没有任何一个技术领域有足够的深度。

他们甚至可能都没好好读过什么技术的源码，很多最新的技术，比如大数据、人工智能、微服务、互联网，等等，都没跟上。

而这些朋友 10 多年来，一直做的事情可能就是重复以下：带着几个小弟，做传统的软件系统，然后整天就是研究各种软件的需求，设计一些简单的架构。

然后使用的技术都是比较过时的，一直是一些增删改查的事情，可能涉及到一些其他的技术，但是那些技术很多都是非主流的，或者是不流行的。

最后，你发现自己 10 多年工作下来，跟小年轻相比，唯一的优势好像就是做的那种 CRUD 的项目比较多，经验丰富一些罢了。

没错，我见过相当多的兄弟，在 30 多岁，乃至 40 岁的时候，就是上述那种情况，唯一的优势就是难度不高的项目经验比较多，带过几个人，仅此而已。

甚至有些兄弟虽然是一些中小公司的 “架构师” 、 “技术总监” 的 title，但是其实本质做的事儿也跟上面是差不多的。

但是呢，这帮兄弟实际上来说薪资未必就很低，因为随着在公司呆的久了，很多公司虽然知道你也许没特别大的技术能力，但是老板也认可你其实对公司的业务还算蛮熟悉的。

而且你做过的项目比较多，年龄较大，有资历，做事稳重，能带几个小弟，可以给公司撑起来一片天空。

此时，公司还是会给你不错的薪资。类似上面情况的兄弟，30 多岁时，可能薪资也会有 30 多 k~40k 那样子。

但是也有很多的兄弟，没得到公司老板那么大的认可，自己可能也一直没想太多，所以在 30 多岁的时候，可能公司就给你 28k，30k 的薪资，认可你是一个资深的骨干。

而这个时候，对大龄程序员来说，学习很多新的技术也有点有心无力，毕竟家里老婆孩子都在炕头上。你说加班加点吧，也有点加不动了，毕竟年龄上去了，各种慢性病一大把，精力不如往日，主要就是利用自己的一点经验把控公司的项目。

因此上述那个场景，就是很多大龄程序员的情况。

二、公司遇到危机时 40 岁大龄程序员会怎么样?

现在咱们换个角度，从一些中小公司的老板的角度来考虑一下这些大龄程序员，假如说公司业务还算稳定，营收还算正常，利润也算稳定，那么这些大龄程序员对老板来说是有价值的。

原因上面已经说过了，你毕竟工作了这么多年，业务还算熟悉，做过很多项目，从需求分析到系统设计，再到带小弟开发、测试和部署上线，这条流程你门儿清，那其实还是可以给你个 30k 薪资用你干活儿的。

但是假如公司现在遇到了一些危机，比如因为行业环境等原因，公司经营不善了，业务开始萎缩，利润开始下降，这个时候你说老板会怎么办?

其实很简单，站在老板的角度，假如手里本来有 10 个大龄程序员作为骨干，此时完全可以拿掉其中的 8 个。

老板这时就留下 2 个大龄程序员，让他们为了保住饭碗，加班加点给公司撑住剩下的业务。

而大龄程序员的工资可是公司里最高的那一批人，把他们拿掉 8 个，是不是一下子节省了很大的成本?

如果公司还有业务需要支撑，完全可以找几个工资才 10k 的小年轻程序员进来把活儿顶着，跟着 2 个大龄程序员继续做就行了。大家想想，是不是这么回事?

在遇到困难的时候，工资高、年龄大、加不动班的大龄程序员，很有可能就会率先 “被” 牺牲掉，用来节约公司成本。

然后呢，换上几个薪资低、年龄小、可以天天加班到凌晨、还有充足的精力学习各种新技术的程序员，给公司顶上。

那么接下来，这些大龄程序员出去找工作会遇到哪些问题呢?

很简单，这些兄弟本身技术并没有什么特长，项目也没什么难度，而且很多最新的技术还没跟上没学习。

此外，这些兄弟年龄也大了，上有老下有小，还加不动班，而且你要的薪资还不低。其他公司一看，有什么理由用 30 多 k 的薪资来聘用你呢?

你的技术、精力都不行，所谓的项目经验，仅仅是上一家公司的一些项目的经验，对别的公司可能没什么太大的作用。

如果你是另外一家公司的老板，你会不会聘用这样的人？我想说到这里，大家都明白这里的问题所在了。

（1）成为技术大牛，掌握公司的核心技术

看完上面的内容，大家都知道问题所在了，那么接下来我们来探讨一下：

什么样的大龄程序员，他可以在 40 岁的时候还得到各种公司的青睐呢?

第一种方式，就是掌握自己行业里的各种技术，哪怕走出了这个公司，也是其他公司疯抢的对象。

给大家举个例子，比如你一直在互联网行业工作，且一直在各种知名大公司，前后负责和经历过多家大型互联网公司的核心系统架构设计。

对于一个互联网系统，如果要支撑 1 亿用户，有哪些技术挑战，需要怎么来设计，你都经历过这些过程。

每秒支撑 10 万量级并发请求该怎么做，系统要能够支撑百亿级的数据存储又该怎么设计。类似这些东西，你都了然于胸。那么这个时候，你就是拥有了核心技术。

哪怕这个公司不要你了，你完全会被其他公司给争抢过去，因为很多公司都需要这种有过大规模系统、高并发经验、海量数据经验的架构师。

千军易得、一将难求。招聘很多薪资二三十 K 的高级工程师，负责把一个模块做好，很容易。

但是你要招一个能够把控全局，hold 住你公司一个复杂大系统全场的高级架构师，很不好招，这样的人很值钱，很多公司都需要。

这个时候你虽然 40 岁，但是人家认可你，因为你有核心的技术能力，核心的架构能力，你是公司技术的最后一道防线，很多岗位非你不可。

或者换个例子，比如大数据领域，现在你工作 10 多年，技术功底极为深厚，你完全可以对大数据领域的各种系统，比如 hadoop、spark、hbase、kylin、elasticsearch 等等，做非常底层的优化。

遇到任何问题，你都可以从源码级别来推断解决，而且可以修改开源项目源码，进行二次开发。

那么这时，你就是拥有核心技术的。未来大数据必然会发展的越来越好，因为各个行业都有大量的数据，很多公司都需要有最好的专家来解决自己公司的大数据问题。

因此，即使你 40 岁了，但是你有多年积累的核心技术能力，任何公司都需要你这样的顶尖大牛坐镇，解决各种技术问题。

所以如果要往这个方向去走，建议大家一定要从 20 多岁开始，好好规划自己的技术学习和职业发展。

大家一定要想好，自己要学什么技术，要往多深了学，要在什么样的公司里积累经验，踏踏实实走好每一步。

到 40 岁的时候，虽然大龄，但是你积累了足够的技术能力，你的核心技术会像 “九阳神功” 一样为你护体，让你依旧驰骋职场、炙手可热。

（2）深挖业务，成为业务领域专家级人物

第二种路线，你可以考虑成为某个业务领域的专家级人物。

你可以在一些业务有绝对难度和深度，业务极度复杂，而且行业持续发展，业务领域的知识持续保持价值的领域，一直深挖。

比如说电信、金融、银行、保险、财务、ERP，等等，诸如此类。

在这些领域的公司里面，你可以在不停的做技术类工作的同时，也注意积累这个领域的业务知识。

像一个电信系统、ERP 系统，往大了做，业务都是极度复杂的，包含了大量的东西。

如果你能持之以恒，对各种业务知识、业务系统都深入挖掘，那么最后变成这个领域的业务专家，也是很有价值的。

为什么这么说呢？因为对于很多特定行业，可能做一个系统不需要那么高大上的技术，但是绝对需要最好的业务专家来进行把控，才能设计出那种对应于某一个特定行业，且业务极其复杂的系统。

所以很多同学，假如是走传统行业的系统开发方向，可以考虑注重更多的业务积累，未来成为顶尖的业务专家。

这样，哪怕你 40 岁的时候，这个行业也需要你这样的 “老专家” 在公司里继续支撑着业务发展。

（3）带领团队：走上技术管理之路

另外一条路，就是走技术管理的道路，这个相信大家都理解。

作为任何公司的老板而言，都需要技术管理人员，他需要懂技术，但是不一定特别的精通，但是一定要有极强的团队管理的能力。

作为团队管理者，你需要有能力把控公司大的技术方向，还需要有足够的行业人脉和资源，招聘搭建合适的技术团队。

此外，你还要能够制定技术团队的工作流程和规范，进行团队之间职责的定义和分配，让各个团队有效协作运行，最后成功的支撑公司的业务发展。

这个管理，说起来就两个字，但其实背后的学问极大，要把几十个甚至几百个不同的人拧成一股绳，保持强大的战斗力，为公司做好支撑工作，其实这里难度很高，非常考验一个管理人员的水平。

如果你能从带几个人的小主管开始，到带几十人的技术 leader，再到带几百人的高级技术管理人员。

假如你能往这个方向去发展，那么其实在你 40 岁的时候，你也有对应的竞争力，因为很多老板都需要你丰富的技术管理经验来把控好公司的技术团队。

当然说实话，能真的做技术管理做的好的人，很少。尤其是咱们技术出身的同学，一般来说都比较内向腼腆，不太善于交际。

所以对大多数的普通工程师而言，个人还是比较建议上面的技术专家或者业务专家路线，这里面机会更多，而且对大多数人都更加的适合。

（4）转型其他职位或者行业

如果有人说，我对技术没那么大热情要成为专家，我也很讨厌整天捣鼓业务，我更没天赋成为技术管理人员，那怎么办呢？

那你可以考虑写代码写到 30 多岁的时候，搞点别的。

比如说考虑转型做产品经理？考虑做互联网运营？或者考虑做软件销售？

也就是说，你可以考虑带着一定的技术背景，往技术行业里的其他岗位去转型，在其他岗位上积累经验，成为不可或缺的人物，这个也可以。

更有甚者，在北上广干了 10 年技术，积累了一定的资金之后，在老家买好房子，然后回老家去做生意，比如开个餐饮店什么的。

这个也是一条路子，但这种就是因人而异了，毕竟每个人的人脉资源都不同。

四、最后的寄语

最后一句话总结：不管你选择哪条路，首先想清楚，你要成为什么样的人。

你需要仔细想想，在 40 岁时，你 10 多年的工作经验，将会积累了哪些核心竞争优势。

然后再仔细琢磨琢磨，这些优势是否是那种刚毕业的小伙子绝对无法替代的，其他公司的资深岗位是不是必须要有你这样的人。

考虑清楚了这些，剩下的就是朝着自己的目标，坚持不懈的走下去。这中间你可能会迷茫，甚至会怀疑当初的决定。但还是那句话：相信自己、勿忘初心，自己选的路，含着泪也要走完！

最不希望看到的一种情况，就是 30 多岁了，还仅仅会 “用” 各种技术，十几种二十多种技术，什么都会用。但就是没什么技术深度，没什么竞争力，就一些使用经验比较多了而已。

如果是这样，那人家刚毕业的小年轻，可能 1 到 2 年时间就学会了这些技术怎么用，也许就可以在关键时刻替代你。

这些小伙子无非就是经验不足罢了，但是人家可以拼命加班弥补，并且人家要求的薪资比你低多了。

最后，希望每个程序员兄弟都有一个好的未来和人生，程序员这条路充满艰难险阻，各种挑战，但也充满机会，需要不停的学习进步，与大家共勉！

【架构师成长必备】如何阅读一个开源项目的源码?

> **作者:中华石杉** 原文地址

目录

- 7、最后寄语：用几年时间锻造自己的核心技术

前言

> 这篇文章，给大家简单介绍一下很多同学都非常关心的一个问题：如何阅读一个开源项目的源码。我相信很多同学都希望能够去阅读一些源码来提升自己的技术水平，毕竟在面试的时候，很多大厂都经常会扣到非常深入的底层源码。

1 从最简单的源码开始：别幻想一步登天

其实开源项目有很多种，比如说有 Spring 这种框架类的，还有比如数据库连接池、log4j 等这种工具类的。

当然还有特别重型的中间件类的，比如说 RocketMQ、Kafka、Redis。更有甚者也有上百万行代码的大数据类的，比如 Hadoop、Spark。

所以如果很多同学想要读源码的话，面临的第一个问题：不知道从何下手。

那么是不是说只要随便挑选一个开源技术的源码，采用愚公移山的精神，直接硬着头皮去读，坚持就是胜利，铁杵一定就能磨成针吗?

不是的！其实很多同学始终都没掌握到阅读源码的顺序、技巧和方法，所以导致尝试看过一些源码，却还是看不懂。

首先你要明白一个前提，比如说 Kafka 的作者，Hadoop 的作者，他们本身都是有很多年经验，技术功底极为扎实，都是技术大牛的人，站在一个很高的角度去设计和开发出来了这些极为出色的分布式系统。

那么如果你的技术实力达不到他们的水平，你觉得你直接去读他们写出来的源码，就能看懂吗?

那估计是很难的，因为里面蕴含的各种底层技术细节，分布式架构设计思想，还有复杂的算法和机制，都不是你能理解的。

所以建议大家第一点，想看源码，先挑一个最最简单的，适合自己技术水平的去看。

给大家举个例子，比如说你平时常用的一些源码都有什么？显而易见，每个人都会用 Spring Web MVC、Spring、MyBatis、Spring Boot，等等。

其实这些开源框架的源码也不能说就简单了，他们同样蕴含了开源作者深厚的技术功底在里面。

但是你要考虑一点，这些开源项目已经相对来说是普通人可以优先触碰的了。因为他们不是分布式系统，不涉及到复杂的架构，网络通信，IO，等技术细节。

他们大多就是依赖一些底层的 Java 基础技术，比如说动态代理、Servlet、HTTP 协议、JDBC 等等。

而他们依赖的那些基础，大多数普通工程师都是掌握的，你完全可以优先尝试去阅读一些这种开源框架类的源码。

2 循序渐进：先搞定底层依赖的技术

好，现在假如说你经过了几个月的努力，把一些开源框架的源码，比如上面说的 SSM 三大框架的源码都看过了，现在你的技术实力有了进一步的提升。

这些提升，主要体现在对开源项目的设计思想，组件设计，组件交互，还有框架封装，等等，都有了进一步的理解。

接下来，你就可以尝试去读一些更难一点的源码。

给大家举个例子，假设你这个时候去阅读 Kafka 的源码。没问题。但是这里有一些是你需要注意的地方，**Kafka 的底层是重度依赖 ZooKeeper 的。**

如果你不把 ZooKeeper 给掌握精通的话，会导致 Kafka 你也难以理解。

所以这个时候你得先把底层依赖的技术给搞定，那么你就得回过头去先阅读 ZooKeeper 的源码，把 ZK 这个技术先给搞精通一些。

同理，如果你在研究 ZK 的时候，发现他底层有一些技术是你掌握不好的，比如你发现他大量运用了 Java 并发包下的东西。

因此如果你对 Java 并发包掌握的不够好，那么建议你去把 Java 并发包下的源码先仔细研究一下。

通过这种方式，你可以自行追踪到自己还不熟悉的很多底层技术，然后一个一个击破，把这些底层依赖的技术的源码你可以先研究透彻一些。

然后，你再一步一步往上层的技术去研究，这样看那些复杂技术的源码就会轻松很多了。

3 一定要以 Hello World 作为入口来阅读

阅读源码有一个非常非常有用的技巧，那就是你别下载了源码到本地 IDE 里然后直接胡乱的翻看，那是不行的。

一般建议就是基于一个开源技术写一个最最基本的 HelloWorld 程序，就是一个入门级的程序，然后把他的核心功能给跑通。

举个例子，假如说你要阅读 ZooKeeper 的源码，那么你先写一个 ZK 的 HelloWorld 程序。

比如说先连接，然后创建一个 znode，对 znode 注册一个监听。接着触发这个监听，接着再关闭连接，就这样的一个简单的程序。

然后就可以打断点，跟踪这个 Hello World 级别的源码一步一步调试追踪，他是如何发起和建立连接的，底层的代码流程是什么样的。

4 抓大放小，边写注释边画图

在看源码的过程中，很多人会被核心流程中混杂的一些特殊业务逻辑的处理给搞懵。

给大家举个例子，看下面的代码，是一段随手写出来演示的：

```php
checkUser();

fetchFromPeers();

countMetrics();
```

大家可以看到，上面就三行代码，从方法名称就可以看出来，先是做了一个权限检查之类的操作，然后是核心业务逻辑去抓取数据，最后是做了一些 metric 指标统计。

那么很多同学看源码的时候，就喜欢把每一行代码都看懂，最后不停的点到很深层的地方去，把自己给绕晕了。最后淹死在源码的海洋里。。。

其实这个是不对的，这就是没有掌握源码阅读的一大典型原则：

抓大放小。

比如上面的三行代码，你应该直接跳过第一行和第三行，连看都别去看，直接进入第二行核心逻辑。

也就是说，你只需要抓最核心的代码流程就可以了，那些无关紧要的代码，千万别有强迫症点进去反复看，那样绝对会让你对源码从入门到放弃。

所以，**再次强调！强调！强调！重要的事情说三遍**。阅读源码，你一定要有粗大的神经，反复告诉自己，刚开始先把握代码的主流程即可。

很多细节看不懂直接跳过去，别有强迫症让自己看明白每个细节。

此外，大家一定要形成一个习惯，在看源码的过程中尽量多自己对源码写一些注释。

你应该结合自己的理解，尽可能把自己对源码阅读过程中的思考都写成注释写在源码里。

这个习惯可以促使你一边阅读一边思考，而且有自己注释的源码，是你宝贵的财富。

此外，还有一个非常重要的点，那就是一定要多画图。

你可以尝试在阅读的过程中，提取源码运行的核心流程，一边读源码，一边自己画在图上，可以用那种画图软件来作图即可。

大家记住，人脑对图片的敏感度，是远高于对文字或者代码的，这个是大脑机制决定的。

笔者公众号写的很多篇文章，里面对各种技术的讲解，无一不是通过大量的画图。相比于冗长的文字描述，图片会让人容易理解接受的多。

通过画图，能帮助你抽象和总结出源码的核心流程，以后如果你要回顾和复习，直接看图即可。

5 反复三遍：真正理解源码

另外一个要注意的点，源码这个东西，是多看几遍理解的就会越深刻。

因为你看第一遍，按照上面说的抓大放小的思路，可能很多东西就直接略过去了，因为刚开始你看不懂一些非核心代码在干什么。

但是第一遍看完以后，通过写注释，自己动手画图，对一个开源项目的核心流程、架构以及原理都有了一定的理解了。

此时再去读第二遍源码，再过一遍，你会发现之前很多看不懂的细节都能看懂了。然后再看第三遍源码，你会发现大多数的代码自己都能看懂了。

所以说任何一个源码，都是要至少反复看三遍的过程，不是看一遍就可以完成的。

6 借力打力：参考源码分析书籍及博客

其实现在有很多对热门开源项目进行源码分析的书籍以及博客，你大致可以认为就是一些技术比较牛的兄弟自己看了源码之后，写出来的一些分析和感悟。

但是那毕竟是别人的东西，如果你上来就直接看源码分析书籍或者博客，那么不一定可以看懂，因为文字的信息传递未必能很好的让你理解有些复杂的东西。

所以比较建议的方式，就是先自己尝试看几遍，有了一定的理解之后，此时可以借助源码分析书籍或者是博客，参考其他技术牛的同学对这个源码理解，结合自己之前的一些思考，综合起来进行分析，相信一定会大有裨益。

你会发现人家的一些理解可以很好的补充你没想明白的一些问题，或者是忽略的一些细节。

不过，需要提醒的一点，网上不少博客，包括一些书籍，他们写出的一些源码分析，可能是错误的。

所以，尽信书不如无书，你需要带着一定的纠错眼光。在和你的理解相悖时，不一定就是你错了。

7 最后寄语：用几年时间锻造自己的核心技术

其实上面那个过程说起来很简单，做起来非常的困难。

因为在上面任何一个步骤，阅读的过程中你都有大量的东西是不会的，而且会觉得很难，甚至经常有想放弃的冲动。

毕竟人的大脑天生就是会对困难的事情产生抗拒感，这是本能，天生就是对舒服、放松的事情有向往。

但是只有那些能克服人的动物本能，惰性本能，迎难而上，坚韧不拔的同学，才能真正攻克各种技术难题。

让自己的大脑不停的开动，不停的思考上面那个过程，也许你要持续一年才能有个小的开悟，持续三年才能有一定的心得，持续五年甚至八年，才能说真的融汇贯通，打通任督二脉，成为技术大牛。

但是坚持这个事情同样是很可怕的，一旦你坚持做到了，那么你将锻造出来自己最硬核的技术实力，远远不是普通人，或者刚毕业的年轻同学可以追上你的。技术深度、技术功底，这是每一个工程师最最硬核的技术实力。

希望各位同学可以从现在开始，尝试着用笔者分享的技巧阅读源码。跳出舒适区，去拥抱更大的舒适区。

真正体验一下读透源码之后，根据报错日志，从源码层面精确定位项目问题、精确制导线上bug，感受一下这种上帝视角解决问题的快感吧！

亿级流量系统架构之如何支撑百亿级数据的存储与计算

> **作者:中华石杉** 原文地址

> ! “本文聊一下笔者几年前所带的团队负责的多个项目中的其中一个，用这个项目来聊聊一个亿级流量系统架构演进的过程。

一、背景引入

首先简单介绍一下项目背景，公司对合作商家提供一个付费级产品，这个商业产品背后涉及到数百人的研发团队协作开发，包括各种业务系统来提供很多强大的业务功能，同时在整个平台中包含了一个至关重要的核心数据产品，这个数据产品的定位是全方位支持用户的业务经营和快速决策。

这篇文章就聊聊这个数据产品背后对应的一套大型商家数据平台，看看这个平台在分布式、高并发、高可用、高性能、海量数据等技术挑战下的架构演进历程。

因为整套系统规模过于庞大，涉及研发人员很多，持续时间很长，文章难以表述出其中各种详细的技术细节以及方案，因此本文主要从整体架构演进的角度来阐述。

至于选择这个商家数据平台项目来聊架构演进过程，是因为这个平台基本跟业务耦合度较低，不像我们负责过的 C 端类的电商平台以及其他业务类平台有那么重的业务在里面，文章可以专注阐述技术架构的演进，不需要牵扯太多的业务细节。

此外，这个平台项目在笔者带的团队负责过的众多项目中，相对算比较简单的，但是前后又涉及到各种架构的演进过程，因此很适合通过文字的形式来展现出来。

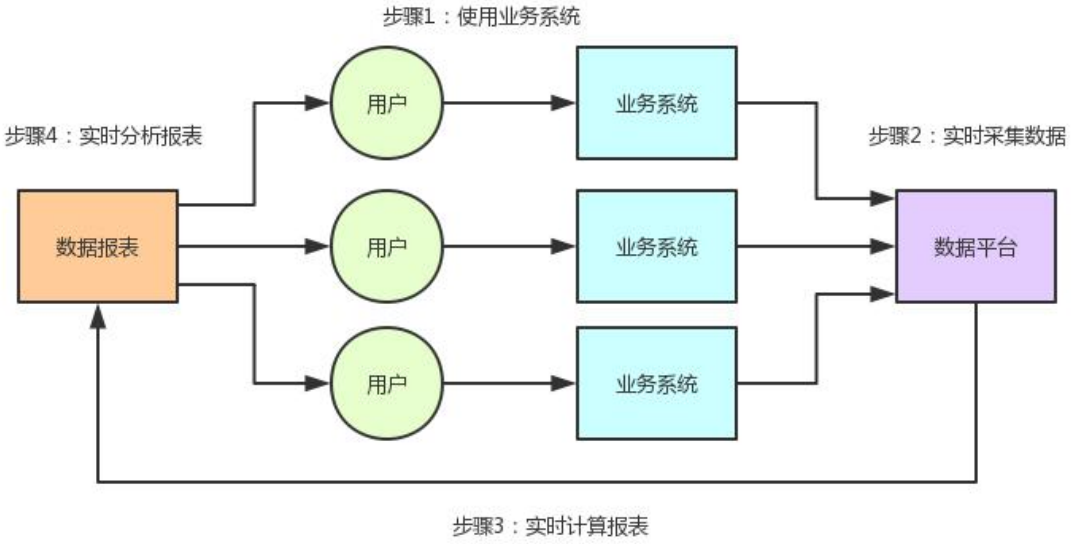
二、商家数据平台的业务流程

下面几点，是这个数据产品最核心的业务流程：

- 每天从用户使用的大量业务系统中实时的采集过来各种业务数据
- 接着存储在自己的数据中心里
- 然后实时的运算大量的几百行~ 上千行的 SQL 来生成各种数据报表
- 最后就可以提供这些数据报表给用户来分析。

基本上用户在业务系统使用过程中，只要数据一有变动，立马就反馈到各种数据报表中，用户立马就可以看到数据报表中的各种变化，进而快速的指导自己的决策和管理。

整个过程，大家看看下面的图就明白了。

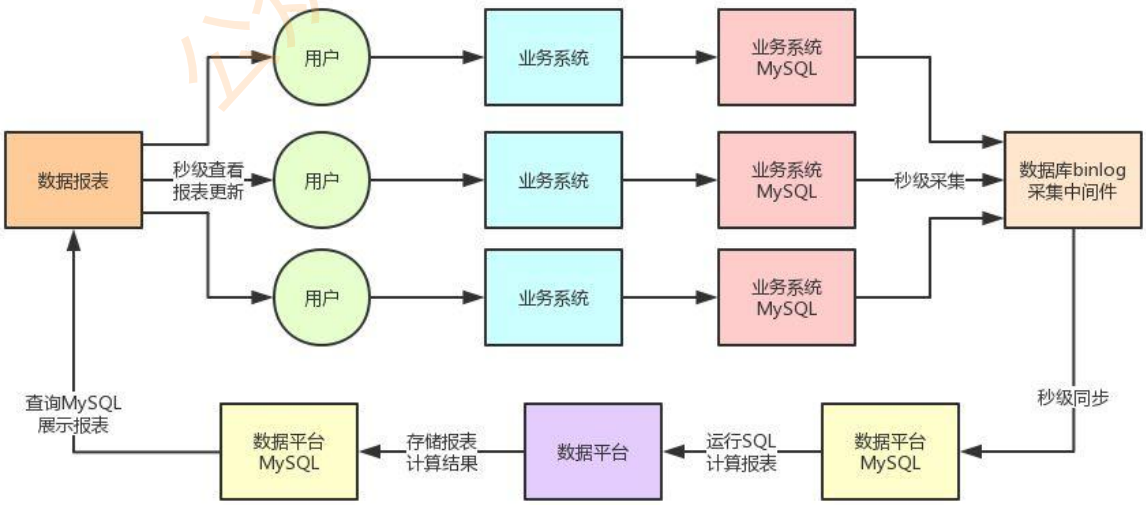


三、从 0 到 1 的过程中上线的最 low 版本

看着上面那张图好像非常的简单，是不是?

看整个过程，似乎数据平台只要想个办法把业务系统的数据采集过来，接着放在 MySQL 的各种表里，直接咔嚓一下运行 100 多个几百行的大 SQL，然后 SQL 运行结果再写到另外一些 MySQL 的表里作为报表数据，接着用户直接点击报表页面查询 MySQL 里的报表数据，就可以了！

其实任何一个系统从 0 到 1 的过程，都是比较 low 的，刚开始为了快速开发出来这个数据平台，还真的就是用了这种架构来开发，大家看下面的图。



其实在刚开始业务量很小，请求量很小，数据量很小的时候，上面那种架构也没啥问题，还挺简单的。

我们直接基于自己研发的数据库 binlog 采集中间件（这个是另外一套复杂系统了，不在本文讨论的范围里，以后有机会可以聊聊），感知各个业务系统的数据库中的数据变更，毫秒级同步到数据平台自己的 MySQL 库里；

接着数据平台里做一些定时调度任务，每隔几秒钟就运行上百个复杂大 SQL，计算各种报表的数据并将结果存储到 MySQL 库中；

最后用户只要对报表刷新一下，立马就可以从 MySQL 库里查到最新的报表数据。

基本上在无任何技术挑战的前提下，这套简易架构运行的会很顺畅，效果很好。然而，事情往往不是我们想的那么简单的，因为大家都知道国内那些互联网巨头公司最大的优势和资源之一，就是有丰富以及海量的 C 端用户以及 B 端的合作商家。

对 C 端用户，任何一个互联网巨头推出一个新的 C 端产品，很可能迅速就是上亿用户量；

对 B 端商家，任何一个互联网巨头如果打 B 端市场，凭借巨大的影响力以及合作资源，很可能迅速就可以聚拢数十万，乃至上百万的付费 B 端用户。

因此，很不幸，接下来的一两年内，这套系统将要面临业务的高速增长带来的巨大技术挑战和压力。

四、海量数据存储和计算的技术挑战

其实跟很多大型系统遇到的第一个技术挑战一样，这套系统遇到的第一个大问题，就是海量数据的存储。

你一个系统刚开始上线也许就几十个商家用，接着随着你们产品的销售持续大力推广，可能几个月内就会聚拢起来十万级别的用户。

这些用户每天都会大量的使用你提供的产品，进而每天都会产生大量的数据，大家可以想象一下，在数十万规模的商家用户使用场景下，每天你新增的数据量大概会是几千万条数据，**记住，这可是每天新增的数据！**这将会给上面你看到的那个很 low 的架构带来巨大的压力。

如果你在负责上面那套系统，结果慢慢的发现，每天都要涌入 MySQL 几千万条数据，这种现象是令人感到崩溃的，因为你的 MySQL 中的单表数据量会迅速膨胀，很快就会达到单表几亿条数据，甚至是数十亿条数据，然后你对那些怪兽一样的大表运行几百行乃至上千行的 SQL？其中包含了 N 层嵌套查询以及 N 个各种多表连接？

我跟你打赌，如果你愿意试一下，你会发现你的数据平台系统直接卡死，因为一个大 SQL 可能都要几个小时才能跑完。然后 MySQL 的 cpu 负载压力直接 100%，弄不好就把 MySQL 数据库服务器给搞宕机了。

所以这就是第一个技术挑战，数据量越来越大，SQL 跑的越来越慢，MySQL 服务器压力越来越大。

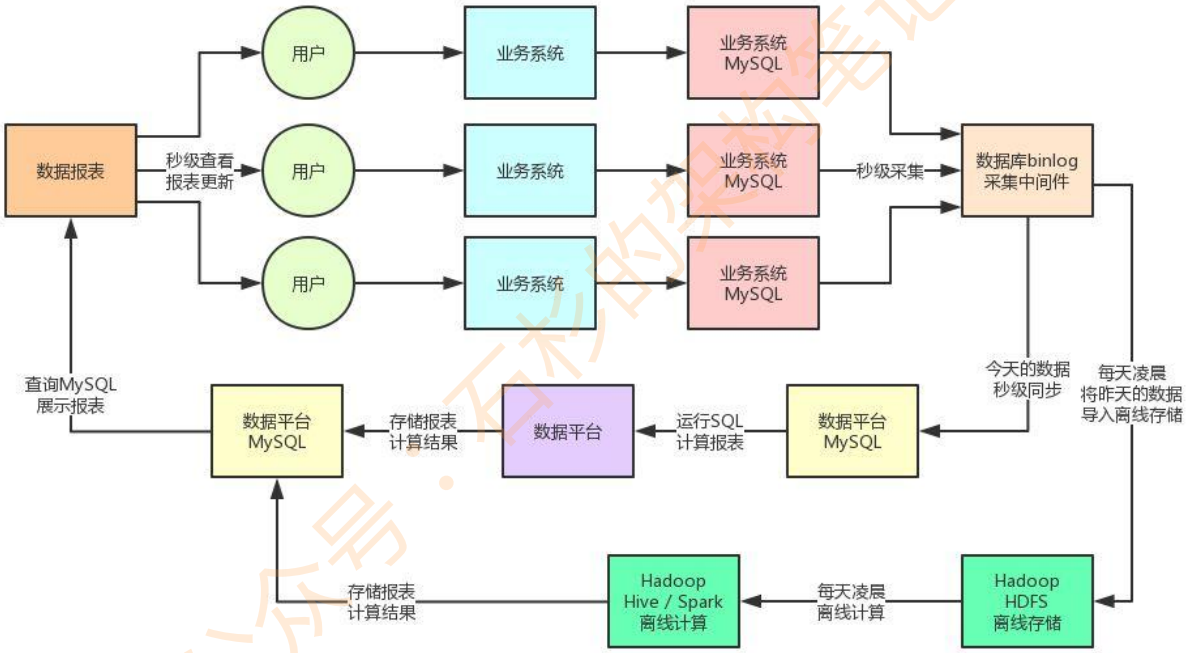
我们当时而言，已经看到了业务的快速增长，因此绝对要先业务一步来重构系统架构，不能让上述情况发生，**第一次架构重构，势在必行！**

五、离线计算与实时计算的拆分

其实在几年前我们做这个项目的时候，大数据技术已经在国内开始运用的不错了，而且尤其在一些大型互联网公司内，我们基本上都运用大数据技术支撑过很多生产环境的项目了，在大数据这块技术的经验积累，也是足够的。

针对这个数据产品的需求，我们完全可以做到，将昨天以及昨天以前的数据都放在大数据存储中，进行离线存储和离线计算，然后只有今天的数据是实时的采集的。

因此在这种技术挑战下，**第一次架构重构的核心要义，就是将离线计算与实时计算进行拆分。**



大家看上面那张图，新的架构之下，分为了离线与实时两条计算链路。

一条是离线计算链路：每天凌晨，我们将业务系统 MySQL 库中的昨天以前的数据，作为离线数据导入 Hadoop HDFS 中进行离线存储，然后凌晨就基于 Hive / Spark 对离线存储中的数据进行离线计算。

如果有同学不清楚大数据的知识，可以参加我之前写的一篇文章：《兄弟，用大白话告诉你小白都能听懂的 Hadoop 架构原理》。Hadoop 与 Spark 作为世界上最优秀、运用最广泛的大数据技术，天然适合 PB 级海量数据的分布式存储和分布式计算。

在离线计算链路全面采用大数据相关技术来支撑过后，完美解决了海量数据的存储，哪怕你一天进来上亿条数据都没事，分布式存储可以随时扩容，同时基于分布式计算技术天然适合海量数据的离线计算。

即使是每天凌晨耗费几个小时将昨天以前的数据完成计算，这个也没事，因为凌晨一般是没人看这个数据的，所以主要在人家早上 8 点上班以前，完成数据计算就可以了。

另外一条是实时计算链路：每天零点过后，当天最新的数据变更，全部还是走之前的老路子，秒级同步业务库的数据到数据平台存储中，接着就是数据平台系统定时运行大量的 SQL 进行计算。同时在每天零点的时候，还会从数据平台的存储中清理掉昨天的数据，仅仅保留当天一天的数据而已。

实时计算链路最大的改变，就是仅仅在数据平台的本地存储中保留当天一天的数据而已，这样就大幅度降低了要放在 MySQL 中的数据量了。

举个例子：比如一天就几千万条数据放在 MySQL 里，那么单表数据量被维持在了千万的级别上，此时如果对 SQL 对应索引以及优化到极致之后，勉强还是可以在几十秒内完成所有报表的计算。

六、持续增长的数据量和计算压力

但是如果仅仅只是做到上面的架构，还是只能暂时性的缓解系统架构的压力，因为业务还在加速狂飙，继续增长。

你老是期望单日的数据量在千万级别，怎么可能？业务是不会给你这个机会的。很快就可以预见到单日数据量将会达到几亿，甚至十亿的级别。

如果一旦单日数据量达到了数十亿的级别，单表数据量上亿，你再怎么优化 SQL 性能，有无法保证 100 多个几百行的复杂 SQL 可以快速的运行完毕了。

到时候又会回到最初的问题，SQL 计算过慢会导致数据平台核心系统卡死，甚至给 MySQL 服务器过大压力，CPU 100% 负载后宕机。

而且此外还有另外一个问题，那就是单个 MySQL 数据库服务器的存储容量是有限的，如果一旦单日数据量达到甚至超过了单台 MySQL 数据库服务器的存储极限，那么此时也会导致单台 MySQL 数据库无法容纳所有的数据了，这也是一个很大的问题！

第二次架构重构，势在必行！

七、大数据领域的实时计算技术的缺陷

在几年前做这个项目的背景下，当时可供选择的大数据领域的实时计算技术，主要还是 Storm，算是比较成熟的一个技术，另外就是 Spark 生态里的 Spark Streaming。当时可没有什么现在较火的 Flink、Druid 等技术。

在仔细调研了一番过后发现，根本没有任何一个大数据领域的实时计算技术可以支撑这个需求。

因为 Storm 是不支持 SQL 的，而且即使勉强你让他支持了，他的 SQL 支持也会很弱，完全不可能运行几百行甚至上千行的复杂 SQL 在这种流式计算引擎上的执行。

Spark Streaming 也是同理，当时功能还是比较弱小的，虽然可以支持简单 SQL 的执行，但是完全无法支持这种复杂 SQL 的精准运算。

因此很不幸的是，在当时的技术背景下，遇到的这个实时数据运算的痛点，没有任何开源的技术是可以解决的。**必须得自己根据业务的具体场景，从 0 开始定制开发自己的一套数据平台系统架构。**

八、分库分表解决数据扩容问题

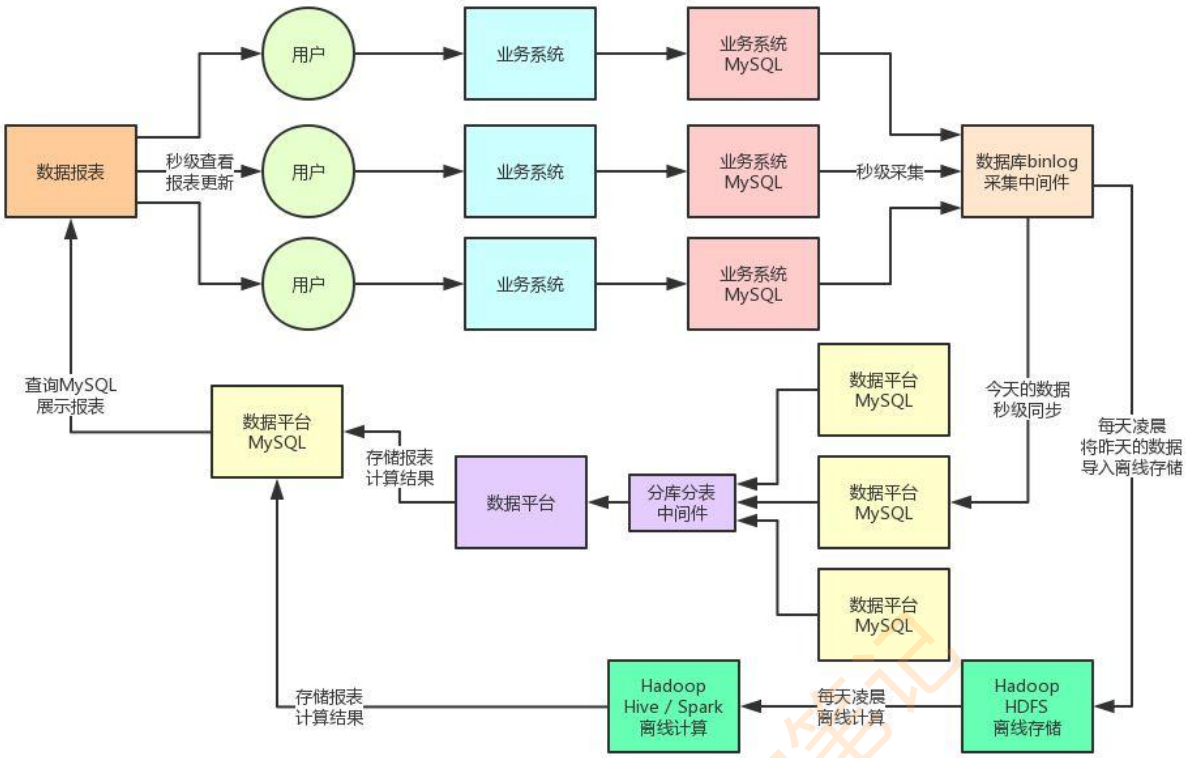
首先我们要先解决第一个痛点，就是一旦单台数据库服务器无法存储下当日的数据，该怎么办?

第一个首选的方案当然就是分库分表了。我们需要将一个库拆分为多库，不用的库放在不同的数据库服务器上，同时每个库里放多张表。

采用这套分库分表架构之后，可以做到每个数据库服务器放一部分的数据，而且随着数据量日益增长，可以不断地增加更多的数据库服务器来容纳更多的数据，做到按需扩容。

同时，每个库里单表分为多表，这样可以保证单表数据量不会太大，控制单表的数据量在几百万的量级，基本上性能优化到极致的 SQL 语句跑起来效率还是不错的，秒级出结果是可以做到的。

同样，给大家来一张图，大家直观的感受一下：



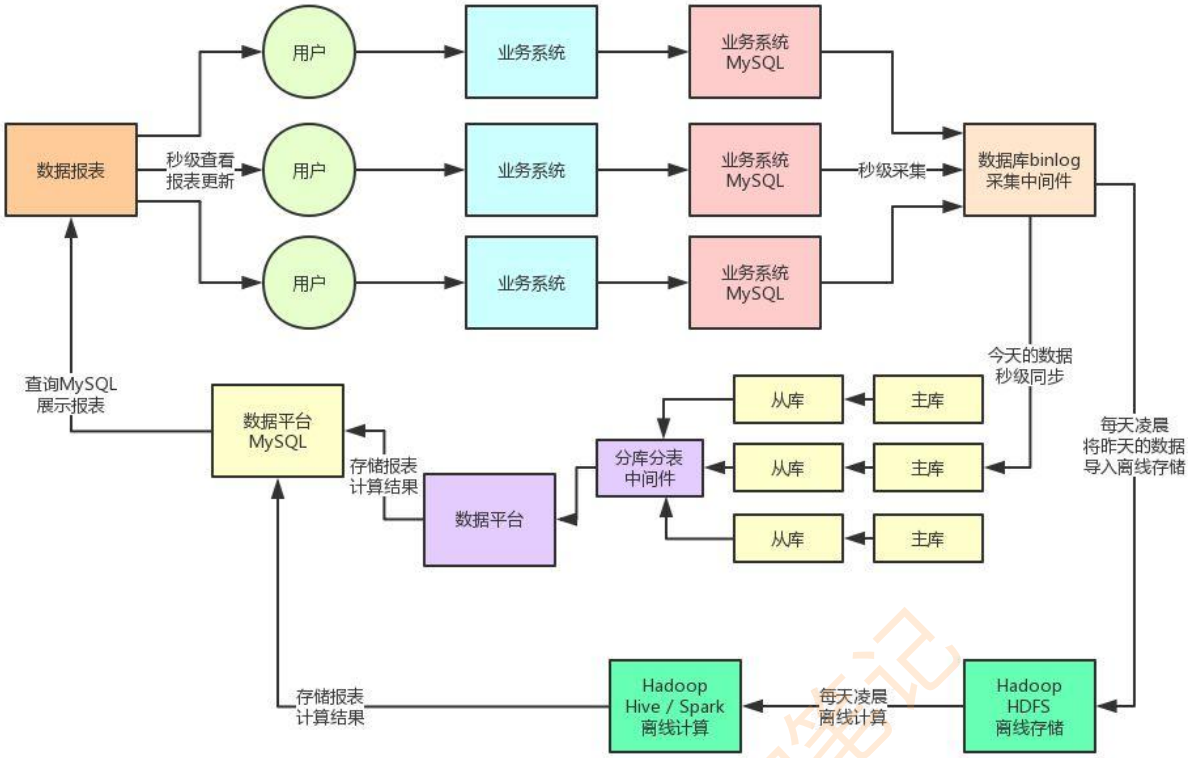
九、读写分离降低数据库服务器的负载

此时分库分表之后，又面临着另外一个问题，就是现在如果对每个数据库服务器又是写入又是读取的话，会导致数据库服务器的 CPU 负载和 IO 负载非常的高！

为什么这么说呢？因为在此时写数据库的每秒并发已经达到几千了，同时还频繁的运行那种超大 SQL 来查询数据，数据库服务器的 CPU 运算会极其的繁忙。

因此我们将 MySQL 做了读写分离的部署，每个主数据库服务器都挂了多个从数据库服务器，写只能写入主库，查可以从从库来查。

大家一起来看看下面这张图：



十、自研的滑动窗口动态计算引擎

但是光是做到这一点还是不够的，因为其实在生产环境发现，哪怕单表数据量限制在了几百万的级别，你运行几百个几百行复杂 SQL，也要几十秒甚至几分钟的时间，这个时效性对付费级的产品已经有点无法接受，**产品提出的极致性能要求是，秒级！**

因此对上述系统架构，我们再次做了架构的优化，在数据平台中嵌入了自己纯自研的滑动窗口计算引擎，核心思想如下：

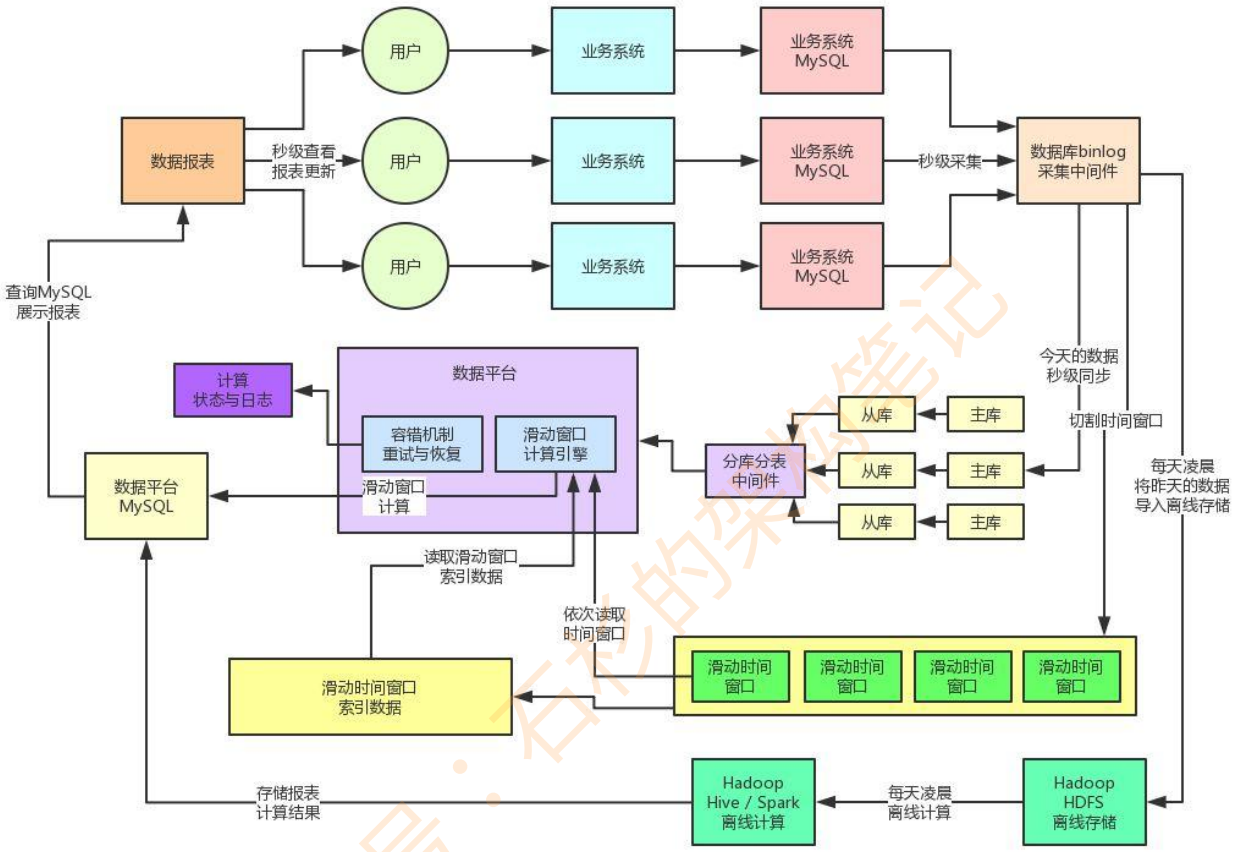
- 在数据库 binlog 采集中间件采集的过程中，要将数据的变更切割为一个一个的滑动时间窗口，每个滑动时间窗口为几秒钟，对每个窗口内的数据打上那个窗口的标签
- 同时需要维护一份滑动时间窗口的索引数据，包括每个分片的数据在哪个窗口里，每个窗口的数据的一些具体的索引信息和状态
- 接着数据平台中的核心计算引擎，不再是每隔几十秒就运行大量 SQL 对当天所有的数据全部计算一遍了，而是对一个接一个的滑动时间窗口，根据窗口标签提取出那个窗口内的数据进行计算，计算的仅仅是最近一个滑动时间窗口内的数据
- 接着对这个滑动时间窗口内的数据，可能最多就千条左右吧，运行所有的复杂 SQL 计算出这个滑动时间窗口内的报表数据，然后将这个窗口数据计算出的结果，与之前计算出来的其他窗口内的计算结果进行合并，最后放入 MySQL 中的报表内

此外，这里需要考虑到一系列的生产级机制，包括滑动时间窗口如果计算失败怎么办？如果一个滑动时间窗口计算过慢怎么办？滑动窗口计算过程中系统宕机了如何在重启之后自动恢复计

算？等等

通过这套滑动窗口的计算引擎，我们直接将系统计算性能提升了几十倍，基本上每个滑动窗口的数据只要几秒钟就可以完成全部报表的计算，相当于一下子把最终呈现给用户的实时数据的时效性提升到了几秒钟，而不是几十秒。

同样，大家看看下面的图。



十一、离线计算链路的性能优化

实时计算链路的性能问题通过自研滑动窗口计算引擎来解决了，但是离线计算链路此时又出现了性能问题。。。

因为每天凌晨从业务库中离线导入的是历史全量数据，接着需要在凌晨针对百亿量级的全量数据，运行很多复杂的上千行复杂 SQL 来进行运算，当数据量达到百亿之后，这个过程耗时很长，有时候要从凌晨一直计算到上午。

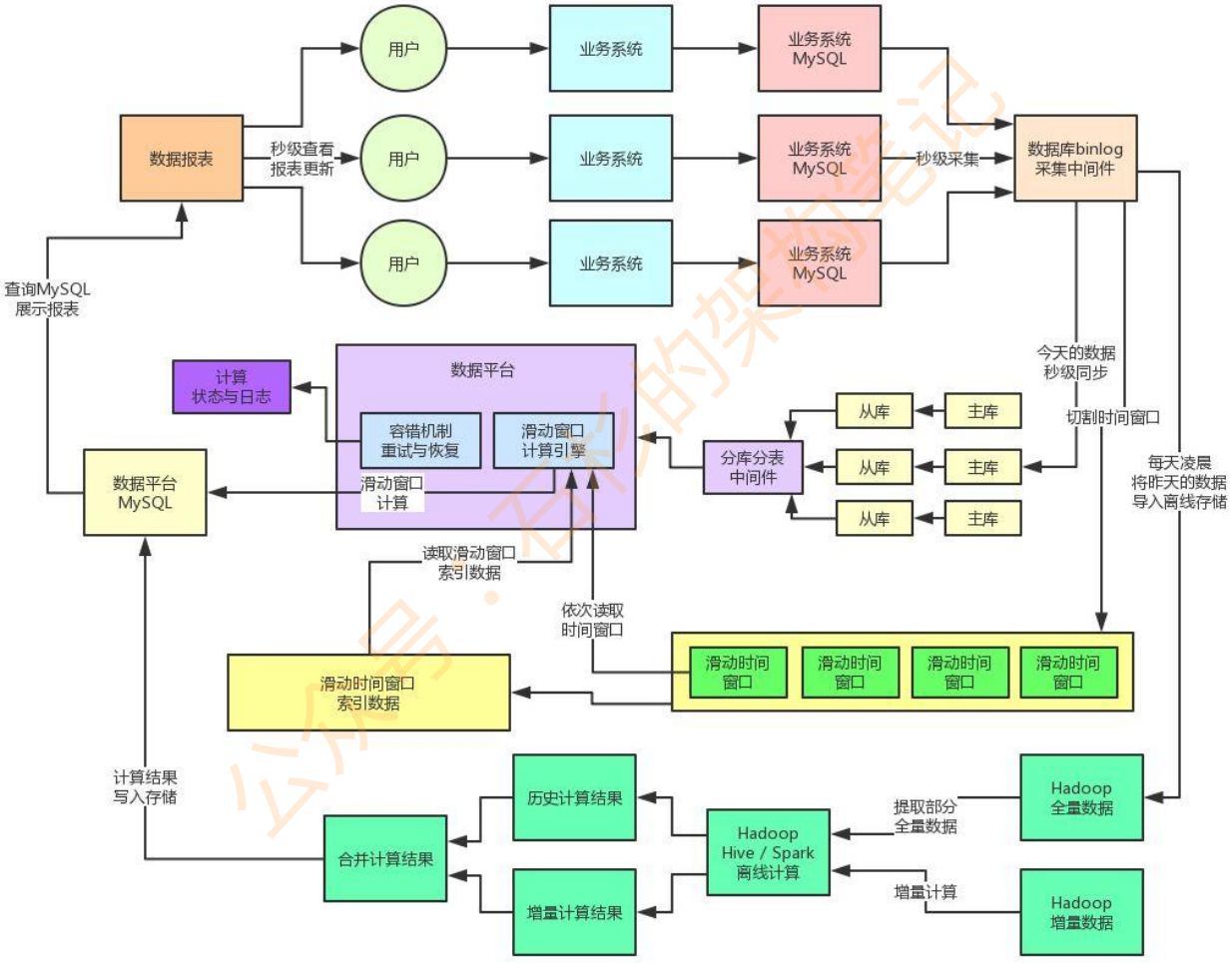
关键问题就在于，离线计算链路，每天都是导入全量数据来进行计算，这就很坑了。

之所以这么做，是因为从业务库同步数据时，每天都涉及到数据的更新操作，而 hadoop 里的数据是没法跟业务库那样来进行更新的，因此最开始都是每天导入全量历史数据，作为一个最新快照来进行全量计算。

在这里，我们对离线计算链路进行了优化，**主要就是全量计算转增量计算**：每天数据在导入 hadoop 之后，都会针对数据的业务时间戳来分析和提取出来每天变更过的增量数据，将这些增量数据放入独立的增量数据表中。

同时需要根据具体的业务需求，自动分析数据计算的基础血缘关系，有可能增量数据需要与部分全量数据混合才能完成计算，此时可能会提取部分全量历史数据，合并完成计算。计算完成之后，将计算结果与历史计算结果进行合并。

在完成这个全量计算转增量计算的过程之后，离线计算链路在凌晨基本上百亿级别的数据量，只要对昨天的增量数据花费一两个小时完成计算之后，就可以完成离线计算的全部任务，性能相较于全量计算提升至少十倍以上。



十二、阶段性总结

到此为止，就是这套系统在最初一段时间做出来的一套架构，不算太复杂，还有很多缺陷，不完美，但是在当时的业务背景下效果相当的不错。

在这套架构对应的早期业务背景下，每天新增数据大概是亿级左右，但是分库分表之后，单表数据量在百万级别，单台数据库服务器的高峰期写入压力在 2000/s，查询压力在 100/s，数据

库集群承载的总高峰写入压力在 1 万 / s，查询压力在 500/s，有需要还可以随时扩容更多的数据库服务器，承载更多的数据量，更高的写入并发与查询并发。

而且，因为做了读写分离，因此每个数据库服务器的 CPU 负载和 IO 负载都不会在高峰期打满，避免数据库服务器的负载过高。

而基于滑动时间窗口的自研计算引擎，可以保证当天更新的实时数据主要几秒钟就可以完成一个微批次的计算，反馈到用户看到的数据报表中。

同时这套引擎自行管理着计算的状态与日志，如果出现某个窗口的计算失败、系统宕机、计算超时，等各种异常的情况，这个套引擎可以自动重试与恢复。

此外，昨天以前的海量数据都是走 Hadoop 与 Spark 生态的离线存储与计算。经过性能优化之后，每天凌晨花费一两个小时，算好昨天以前所有的数据即可。

最后实时与离线的计算结果在同一个 MySQL 数据库中融合，此时用户如果对业务系统做出操作，实时数据报表在几秒后就会刷新，如果要看昨天以前的数据可以随时选择时间范围查看即可，暂时性是满足了业务的需求。

早期的几个月里，日增上亿数据，离线与实时两条链路中的整体数据量级达到了百亿级别，无论是存储扩容，还是高效计算，这套架构基本是撑住了。

十三、下一阶段的展望

这个大型系统架构演进实践是一个系列的文章，将会包含很多篇文章，因为一个大型的系统架构演进的过程，会持续很长时间，做出很多次的架构升级与重构，不断的解决日益增长的技术挑战，最终完美的抗住海量数据、高并发、高性能、高可用等场景。

下一篇文章会说说下一步是如何将数据平台系统重构为一套高可用高容错的分布式系统架构的，来解决单点故障、单系统 CPU 负载过高、自动故障转移、自动数据容错等相关的问题。包括之后还会有多篇文章涉及到我们自研的更加复杂的支撑高并发、高可用、高性能、海量数据的平台架构。

十四、上篇文章的答疑

上一篇文章写了一个分布式锁的高并发优化的文章，具体参见：**每秒上千订单场景下的分布式锁高并发优化实践！**。收到了大家很多的提问，其实最终都是一个问题：

针对那篇文章里的用分布式锁的分段加锁的方式，解决库存超卖问题，那如果一个分段的库存不满足要购买的数量，怎么办？

第一，我当时文章里提了一句，可能没写太详细，如果一个分段库存不足，要锁其他的分段，进行合并扣减，如果你做分段加锁，那就是这样的，很麻烦。

如果大家去看看 Java 8 里的 LongAdder 的源码，他的分段加锁的优化，也是如此的麻烦，要做段迁移。

第二，我在那篇文章里反复强调了一下，不要对号入座，因为实际的电商库存超卖问题，有很多其他的技术手段，我们就用的是其他的方案，不是这个方案，以后有机会给大家专门讲如何解决电商库存超卖问题。

那篇文章仅仅是用那个例作为一个业务案例而已，阐述一下分布式锁的并发问题，以及高并发的优化手段，方便大家来理解那个意思，仅此而已。

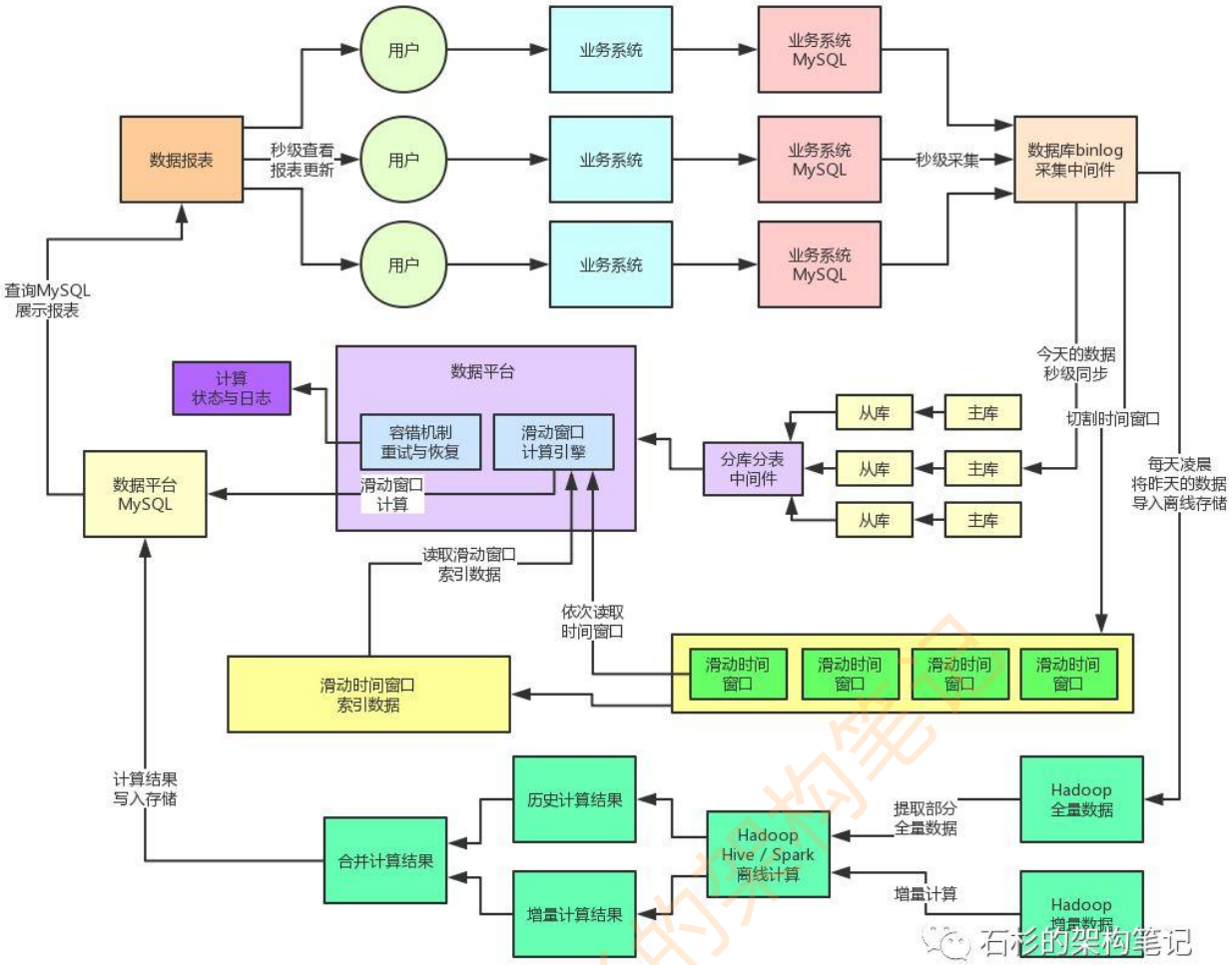
第三，最后再强调一下，大家关注分段加锁的思想就好，切记不要对号入座，不要关注过多在库存超卖业务上了。

亿级流量系统架构之如何设计高容错分布式计算系统

> **作者:中华石杉** 原文地址

一、写在前面 上篇文章**亿级流量系统架构之如何支撑百亿级数据的存储与计算**，聊了一下商家数据平台第一个阶段的架构演进。通过离线与实时计算链路的拆分，离线计算的增量计算优化，实时计算的滑动时间窗口计算引擎，分库分表 + 读写分离，等各种技术手段，支撑住了百亿量级的数据量的存储与计算。

我们先来回看一下当时的那个架构图，然后继续聊聊这套架构在面对高并发、高可用、高性能等各种技术挑战下，应该如何继续演进。



二、active-standby 高可用架构

大家看看上面的那个架构图，有没有发现里面有一个比较致命的问题？就是如何避免系统单点故障！

在最初的部署架构下，因为数据平台系统对 CPU、内存、磁盘的要求很高，所以我们是单机部署在一台较高配置的虚拟机上的，16 核 CPU、64G 内存、SSD 固态硬盘。这个机器的配置是可以保证数据平台系统在高负载之下正常运行的。

但是如果仅仅是单机部署数据平台系统的话，会导致致命的单点故障问题，也就是如果单台机器上部署的数据平台系统宕机的话，就会立马导致整套系统崩溃。

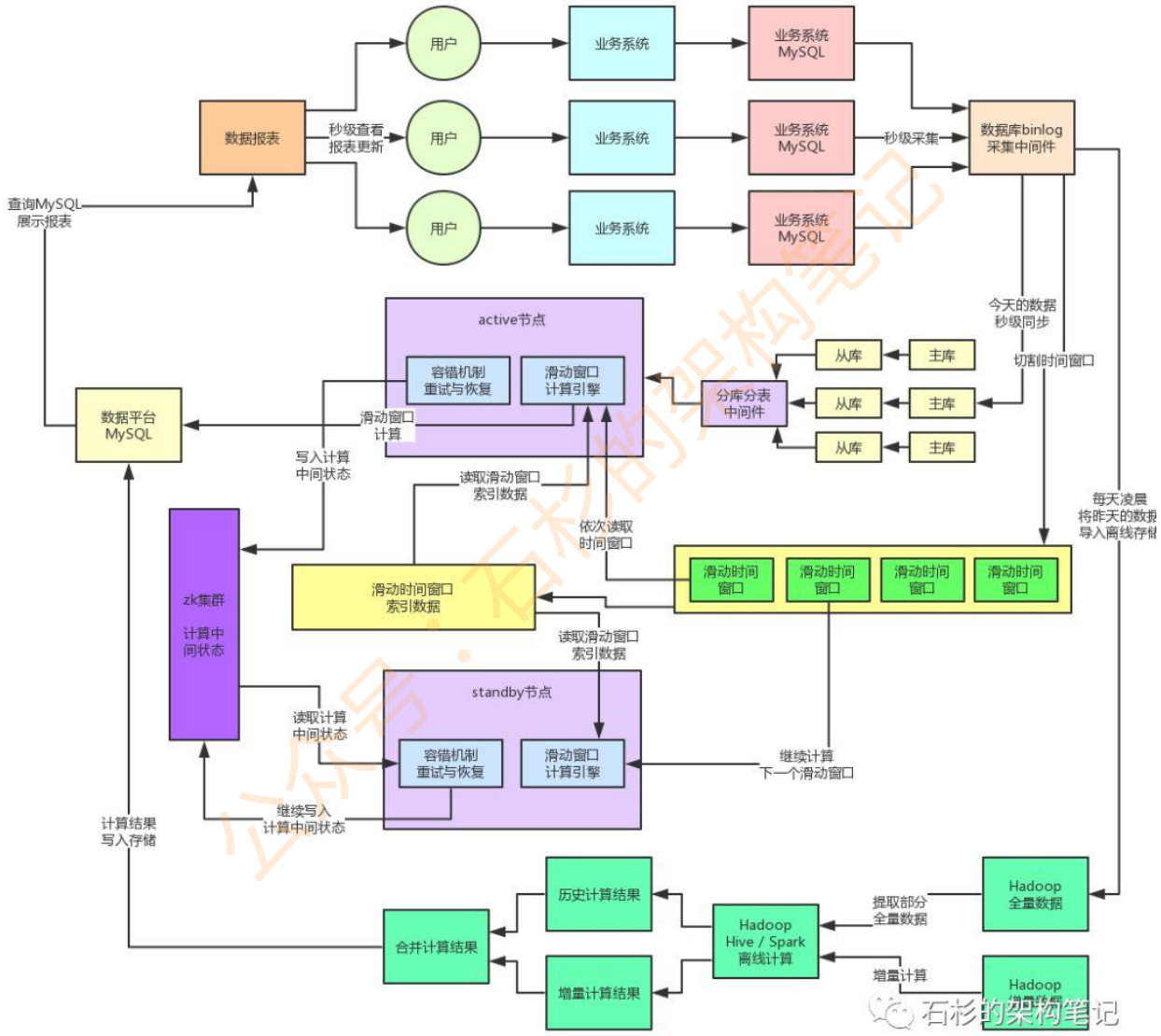
因此在初期的阶段，我们对数据平台实现了 active-standby 的高可用架构，也就是一共部署在两台机器上，但是同一时间只有一台机器是会运行的，但是另外一台机器是备用的。处于 active 状态的系统会将滑动窗口计算引擎的计算状态和结果写入 zookeeper 中，作为元数据存储起来。

关于元数据基于 zookeeper 来存储，我们是充分参考了开源的 Storm 流式计算引擎的架构实现，因为 Storm 作为一个非常优秀的分布式流式计算系统，同样需要高并发的读写大量的计算中间状态和数据，他就是基于 zookeeper 来进行存储的。

本身 zookeeper 的读写性能非常的高，而且 zookeeper 集群自身就可以做到非常高的可用性，同时还提供了大量的分布式系统需要的功能支持，包括分布式锁、分布式协调、master 选举、主备切换等等。

因此基于 zookeeper 我们实现了 active-standby 的主备自动切换，如果 active 节点宕机，那么 standby 节点感知到，会自动切花为 active，同时自动读取他们共享的一个计算引擎的中间状态，然后继续恢复之前的计算。

大家看下面的图，一起感受一下。



在完成上述的 active-standby 架构之后，肯定是消除掉了系统的单点故障了，保证了基本的可用性。而且在实际的线上生产环境中表现还不错，一年系统总有个几次会出现故障，但是每次都能自动切换 standby 机器稳定运行。

这里随便给大家举几个生产环境机器故障的例子，因为部署在公司的云环境中，用的都是虚拟机，可能遇到的坑爹故障包括但不限于下面几种情况：

- 虚拟机所在的宿主机挂了

- 虚拟机的网络出现故障
- 负载过高导致磁盘坏了

所以在线上高负载环境中，**永远别寄希望于机器永远不宕机**，你要随时做好准备，机器会挂！系统必须做好充分的故障预测、高可用架构以及故障演练，保证各种场景下都可以继续运行。

三、Master-Slave 架构的分布式计算系统

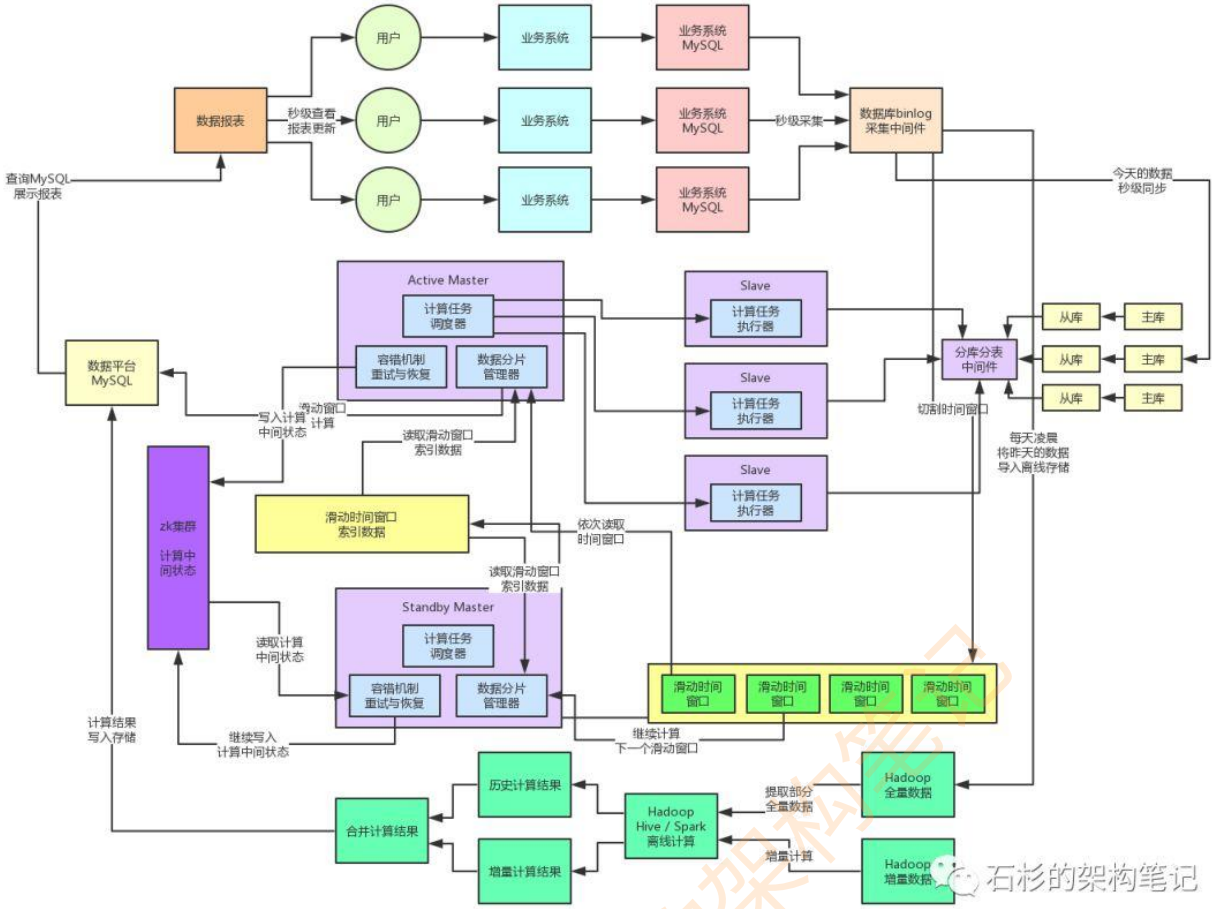
但是此时另外一个问题又来了，大家考虑一个问题，数据平台系统其实最核心的任务就是对一个一个的时间窗口中的数据进行计算，但是随着每天的日增数据量越来越多，每个时间窗口内的数据量也会越来越大，同时会导致数据平台系统的计算负载越来越高。

在线上生产环境表现出来的情况就是，数据平台系统部署机器的 CPU 负载越来越高，高峰期很容易会 100%，机器压力较大。**新一轮的系统重构，势在必行。**

首先我们将数据平台系统彻底重构和设计为一套分布式的计算系统，将任务调度与任务计算两个职责进行分离，有一个专门的 Master 节点负责读取切分好的数据分片（也就是所谓的时间窗口，一个窗口就是一个数据分片），然后将各个数据分片的计算任务分发给多个 Slave 节点。

Slave 节点的任务就是专门接收一个一个的计算任务，每个计算任务就是对一个数据分片执行一个几百行到上千行的复杂 SQL 语句来产出对应的数据分析结果。

同时对 Master 节点，我们为了避免其出现单点故障，所以还是沿用了之前的 Active-Standby 架构，Master 节点是在线上部署一主一备的，平时都是 active 节点运作，一旦宕机，standby 节点会切换为 active 节点，然后自动调度运行各个计算任务。



这套架构部署上线之后，效果还是很不错的，因为 Master 节点其实就是读取数据分片，然后为每个数据分片构造计算任务，接着就是将计算任务分发给各个 Slave 节点进行计算。

Master 节点几乎没有太多复杂的任务，部署一台高配置的机器就绝对没问题。

负载主要在 Slave 节点，而 Slave 节点因为部署了多台机器，每台机器就是执行部分计算任务，所以很大程度上降低了单台 Slave 节点的负载，而且只要有需要，随时可以对 Slave 集群进行扩容部署更多的机器，这样无论计算任务有多繁忙，都可以不断的扩容，保证单台 Slave 机器的负载不会过高。

四、弹性计算资源调度机制

在解决了单台机器计算负载压力过高的问题之后，我们又遇到了下一个问题，就是在线上生产环境中偶尔会发现某个计算任务耗时过长，导致某台 Slave 机器积压了大量的计算任务一直迟迟得不到处理。

这个问题的产生，其实主要是由于系统的高峰和低谷的数据差异导致的。

大家可以想想，在高峰期，瞬时涌入的数据量很大，很可能某个数据分片包含的数据量过大，达到普通数据分片的几倍甚至几十倍，这是原因之一。

还有一个原因，因为截止到目前为止的计算操作，其实还是基于几百行到上千行的复杂 SQL 落地到 MySQL 从库中去执行计算的。

因此，在高峰期可能 MySQL 从库所在数据库服务器的 CPU 负载、IO 负载都会非常的高，导致 SQL 执行性能下降数倍，这个时候数据分片里的数据量又大，执行的又慢，很容易就会导致某个计算任务执行时间过长。

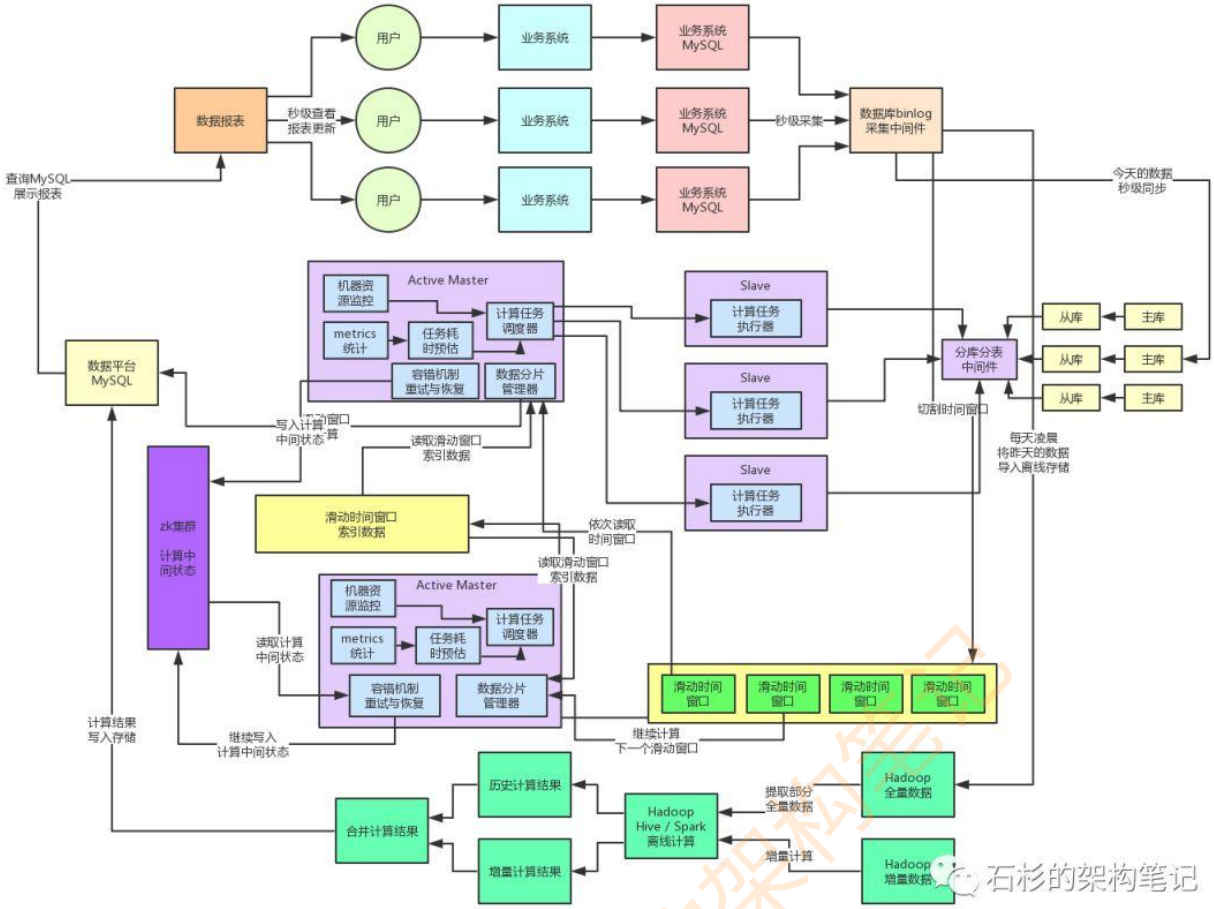
最后一个造成负载不均衡的原因，就是每个计算任务对应一个数据分片和一个 SQL，但是不同的 SQL 执行效率不同，有的 SQL 可能只要 200 毫秒就可以结束，有的 SQL 要 1 秒，所以不同的 SQL 执行效率不同，造成了不同的计算任务的执行时间的不同。

因此，我们又专门在 Master 节点中加入了计算任务 metrics 上报、计算任务耗时预估、任务执行状态监控、机器资源管理、弹性资源调度等机制。

实现的一个效果大致就是：

- Master 节点会实时感知到各个机器的计算任务执行情况、排队负载压力、资源使用等情况。
- 同时还会收集各个机器的计算任务的历史 metrics
- 接着会根据计算任务的历史 metrics、预估当前计算任务的耗时、综合考虑当前各 Slave 机器的负载，来将任务分发给负载较低的 Slave 机器。

通过这套机制，我们充分保证了线上 Slave 集群资源的均衡利用，不会出现单台机器负载过高，计算任务排队时间过长的情况，**经过生产环境的落地实践以及一些优化之后，该机制运行良好。**



五、分布式系统高容错机制

其实一旦将系统重构为分布式系统架构之后，就可能会出现各种各样的问题，此时就需要开发一整套的容错机制。

大体说起来的话，这套系统目前在线上生产环境可能产生的问题包括但不限于：

- 某个 Slave 节点在执行过程中突然宕机
- 某个计算任务执行时间过长
- 某个计算任务执行失败

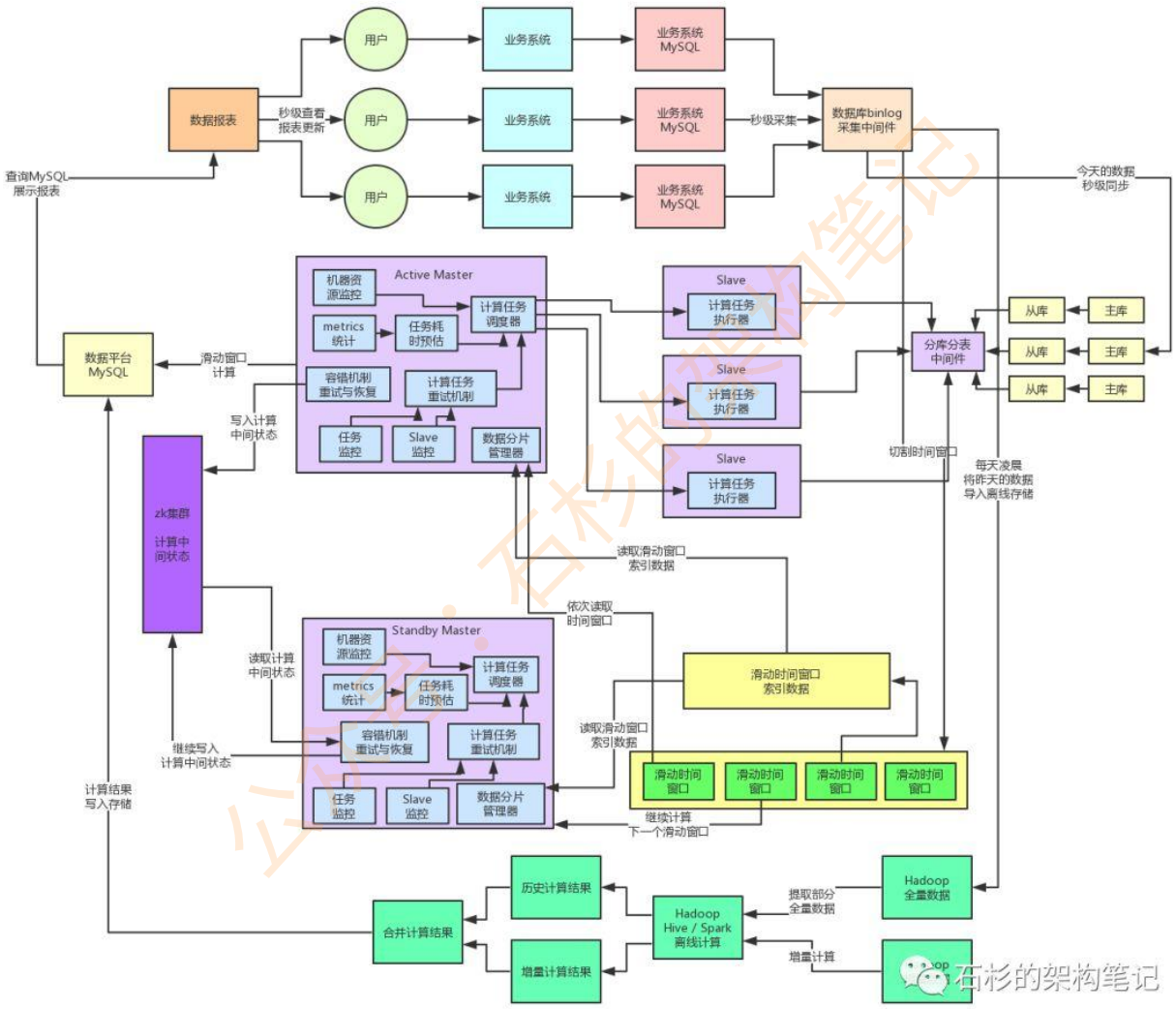
因此，Master 节点内需要实现一套针对 Slave 节点计算任务调度的容错机制，大体思路如下：

- Master 节点会监控各个计算任务的执行状态，同时也会监控各个 Slave 节点的运行状态
- 如果说某个 Slave 宕机了，那么此时 Master 就会将那个 Slave 没执行完的计算任务重新分配给其他的 Slave 节点
- 如果说某个 Slave 的计算任务执行失败了，同时重试几次之后还是失败，那么 Master 会将这个计算任务重新分配给其他的 Slave 节点来执行

- 如果说某个计算任务在多个 Slave 中无法成功计算的话，此时会将这个计算任务储存在一个延时内存队列中，间隔一段时间过后，比如说等待高峰期故去，然后再重新尝试执行这个计算任务

- 如果某个计算任务等待很长时间都没成功执行，可能是 hang 死了，那么 Master 节点会更新这个计算任务的版本号，然后分配计算任务给其他的 Slave 节点来执行。

- 之所以要更新版本号，是为了避免说，新分配的 Slave 执行完毕写入结果之后，之前的那个 Slave hang 死了一段时间恢复了，接着将计算结果写入存储覆盖正确的结果。用版本号机制可以避免这种情况的发生。



六、阶段性总结

系统架构到这个程度为止，其实在当时而言是运行的相当不错的，每日亿级的请求以及数据场景下，这套系统架构都能承载的很好，如果写数据库并发更高可以随时加更多的主库，如果读并发过高可以随时加更多的从库，同时单表数据量过大了就分更多的表，Slave 计算节点也可以随时按需扩容。

计算性能也是可以在这个请求量级和数据量级下保持很高的水准，因为数据分片计算引擎（滑动窗口）可以保证计算性能在秒级完成。同时各个 Slave 计算节点的负载都可以通过弹性资源调度机制保持的非常的均衡。

另外整套分布式系统还实现了高可用以及高容错的机制，Master 节点是 Active-Standby 架构可以自动故障转移，Slave 节点任何故障都会被 Master 节点感知到同时自动重试计算任务。

七、下一个阶段的展望

其实如果仅仅只是每天亿级的流量请求过来，这套架构是可以撑住了，但是问题是，随之接踵而来的，就是每天请求流量开始达到数十亿次甚至百亿级的请求量，此时上面那套架构又开始支撑不住了，需要继续重构和演进系统架构。

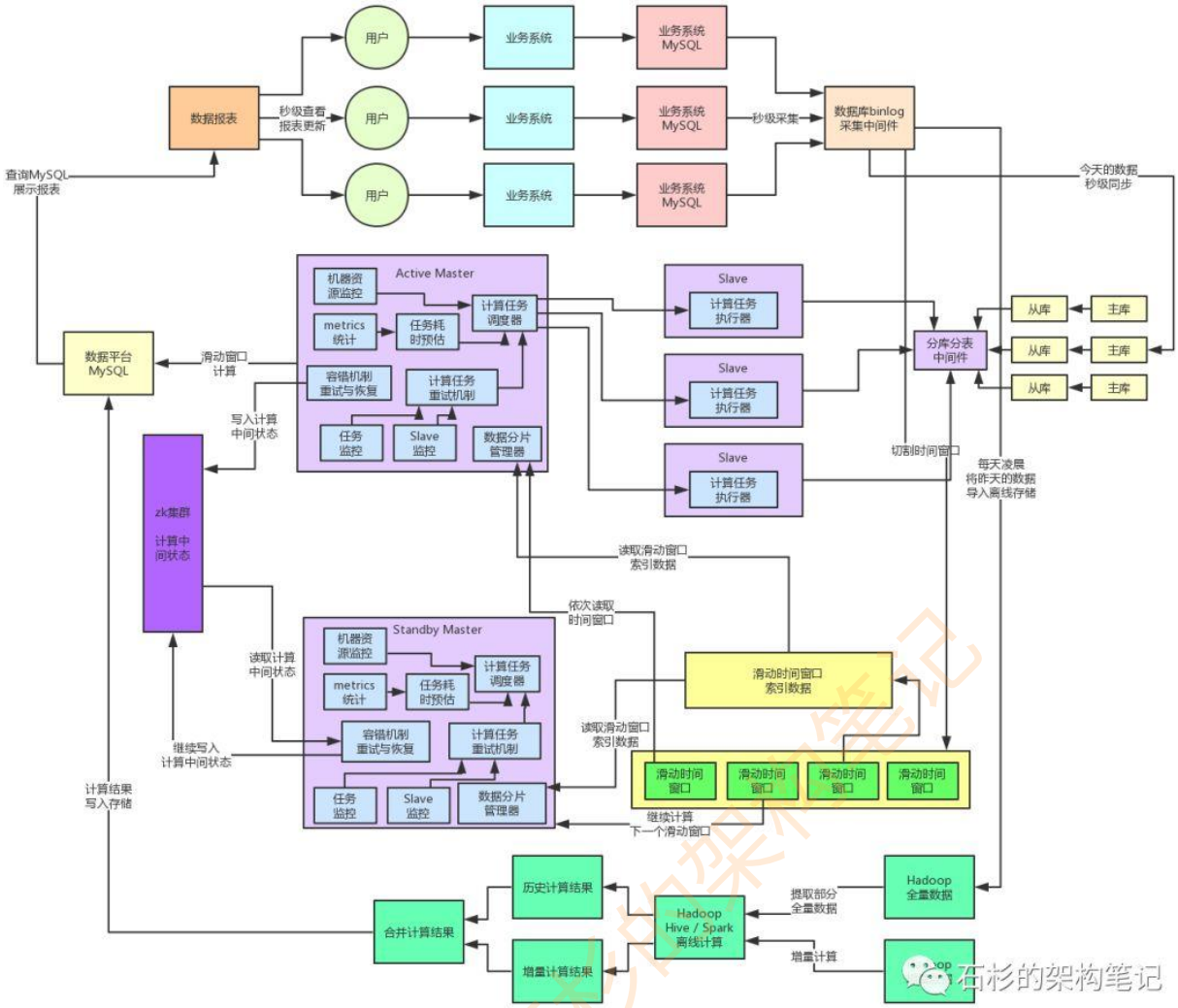
亿级流量系统架构之如何设计承载百亿流量的高性能架构

> 作者:中华石杉 原文地址

一、往期回顾

上篇文章《大型系统架构演进之如何设计高容错分布式计算系统》，主要聊了一下将单块系统重构为分布式系统，以此来避免单台机器的负载过高。同时引申出来了弹性资源调度、分布式容错机制等相关的东西。

这篇文章我们继续来聊聊这个系统后续的重构演进过程，先来看下目前的系统架构图，一起来回顾一下。



二、百亿流量的高并发技术挑战

上篇文章说到，如果仅仅只是每天亿级流量的话，其实基本上目前的系统架构就足够支撑了，但是呢，我们面临的可不仅仅是亿级流量那么简单。我们面对的是日益增多和复杂的各种业务系统，我们面对的是不断增加的系统用户，我们面对的是即将迎来每天百亿级的高并发流量。

给大家先说下当时的系统部署情况，数据库那块一共部署了 8 主 8 从，也就是 16 台数据库服务器，每个库都是部署在独立的数据库服务器上的，而且全部用的是物理机，机器的配置，如果没记错的话，应该是 32 核 + 128G+SSD 固态硬盘。

为啥要搞这么多物理机，而且全部都是高配置呢？不知道大家发现没有，目前为止，我们最大的依赖就是 MySQL！

之前给大家解释过，在当时的背景下，我们要对涌入的亿级海量数据，实时的运行数百个复杂度为几百行到上千行的大 SQL，几秒钟就要出分析结果。

这个是没有任何一个开源系统可以做到的，Storm 不行，Spark Streaming 也不行，因此必须得基于 MySQL 纯自研一套数据平台架构出来，支撑这个需求场景。

所以，只有 MySQL 是可以支撑如此复杂的 SQL 语句完美运行的，因此我们在早期必须严重依赖于 MySQL 作为数据的存储和计算，将源源不断涌入的数据放在 MySQL 中进行存储，接着基于数据分片计算的架构来高性能的运行复杂大 SQL 基于 MySQL 来进行计算。

所以大家就知道了，MySQL 目前为止是这套系统的命脉。在当时的场景下，每台数据库服务器都要抗住每秒 2000 左右的并发请求，高峰期的 CPU 负载、IO 负载其实都非常高，而且主库和从库的延迟在高峰期已经有点严重，会达到秒级了。

在我们的生产系统的实际线上运行情况下，单台 MySQL 数据库服务器，我们一般是不会让他的高峰期并发请求超过 2000/s 的，因为一旦达到每秒几千的请求，根据当时线上的资源负载情况来看，很可能 MySQL 服务器负载过高会宕机。

所以此时就有一个很尴尬的问题了，假如说每天亿级流量的场景下，需要用 8 主 8 从这么多高配置的数据库服务器来抗，那如果是几十亿流量呢？甚至如果是百亿流量呢？难道不停的增加更多的高配置机器吗？

要知道，这种高配置的数据库服务器，如果是物理机的话，是非常昂贵的！

之前给大家简单介绍过项目背景，这整套大型系统组成的商业级平台，涉及到 N 多个系统，这个数据产品只是一个子产品而已，不可能为了这么一个产品，投入大量的预算通过不停的砸高配置的机器来撑住更高的并发写入。

我们必须用技术的手段来重构系统架构，尽量用有限的机器资源，通过最优秀的架构来抗住超高的并发写入压力！

三、计算与存储分离的架构

这个架构里的致命问题之一，就是数据的存储和计算混在了一个地方，都在同一个 MySQL 库里！

大家想想，在一个单表里放上千万数据，然后你每次运行一个复杂 SQL 的时候，SQL 里都是通过索引定位到表中他要计算的那个数据分片。这样搞合适吗？

答案显然是否定的！因为表里的数据量很大，但是你每次实际 SQL 运算只要对其中很小很小的一部分数据计算就可以了，实际上我们在生产环境中实践过后发现，如果你在一个大表运行一个复杂 SQL，哪怕通过各种索引保证定位到的数据量很少，因为表数据量过大，也是会导致性能直线下降的。

因此第一件事情，先将数据的存储和计算这两件事情拆开。

我们当时的思路如下：

- 数据直接写入一个存储，仅仅只是简单的写入即可

- 然后在计算的时候从数据存储中提取你需要的那个数据分片里的可能就一两千条数据，写入另外一个专用于计算的临时表中，那个临时表内就这一两千条数据

- 然后运行你的各种复杂 SQL 即可。

bingo！一旦将数据存储和计算两个事情拆开，架构里可以发挥的空间就大多了。

首先你的数据存储只要支撑高并发的写入，日百亿流量的话，高峰每秒并发会达到几十万，撑住这就可以了。然后支持计算引擎通过简单的操作从数据存储里提取少量数据就 OK。

太好了，这个数据存储就可以 PASS 掉 MySQL 了，就这点儿需求，你还用 MySQL 干什么？兄弟！

当时我们经过充分的技术调研和选型之后，选择了公司自研的分布式 KV 存储系统，这套 KV 存储系统是完全分布式的，高可用，高性能，轻量级，支持海量数据，而且之前经历过公司线上流量的百亿级请求量的考验，绝对没问题。主要支持高并发的写入数据以及简单的查询操作，完全符合我们的需求。

这里给大家提一句，其实业内很多类似场景会选择 hbase，所以大家如果没有公司自研的优秀 kv 存储的话，可以用选用 hbase 也是没问题的，只不过 hbase 有可能生产环境会有点坑，需要大家对 hbase 非常精通，合理避坑和优化。

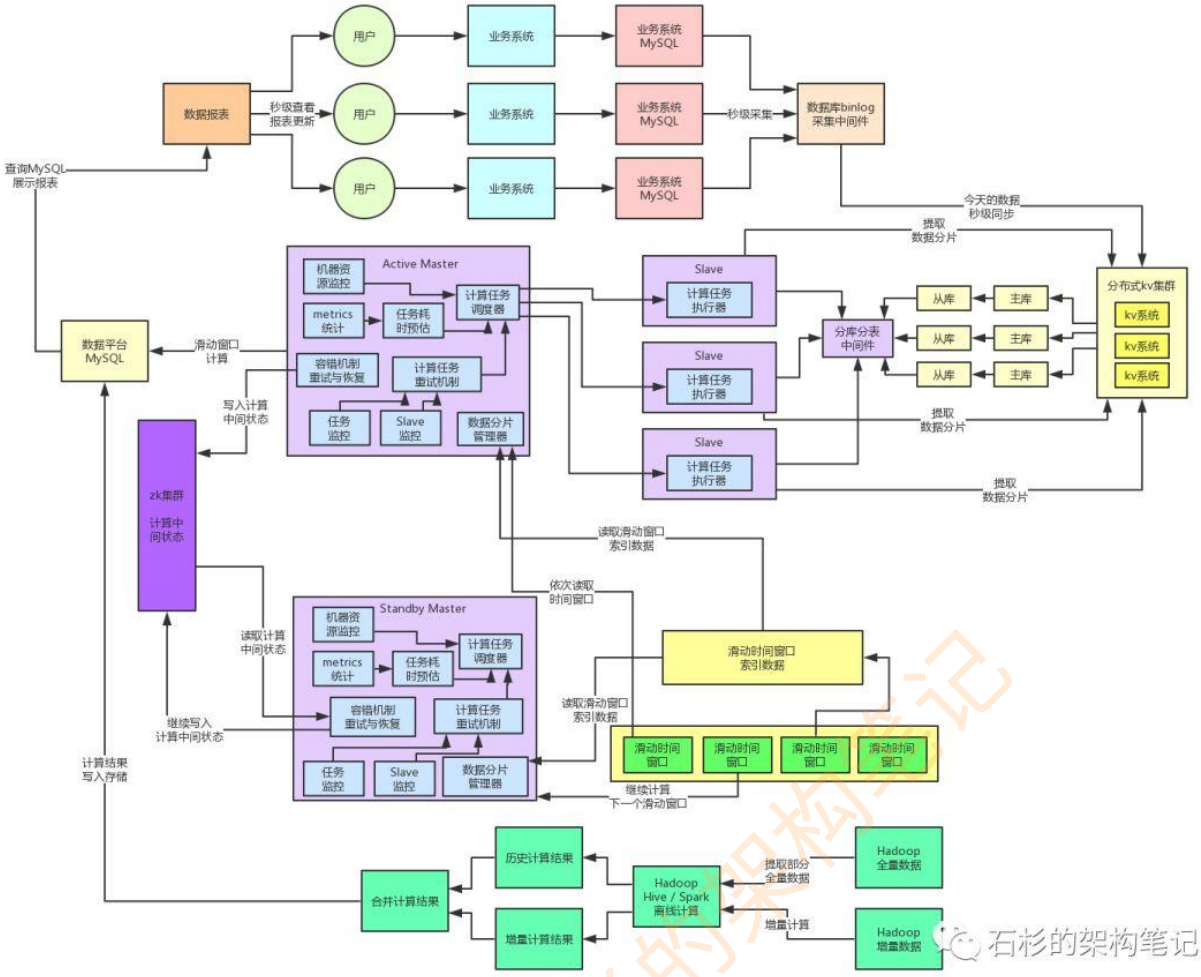
轻量级的分布式 kv 系统，一般设计理念都是支持一些简单的 kv 操作，大量的依托于内存缓存热数据来支持高并发的写入和读取，因为不需要支持 MySQL 里的那些事务啊、复杂 SQL 啊之类的重量级的机制。

因此在同等的机器资源条件下，kv 存储对高并发的支撑能力至少是 MySQL 的数倍甚至数十倍。

就好比说，大家应该都用过 Redis，Redis 普通配置的单机器撑个每秒几万并发都是 ok 的，其实就是这个道理，他非常的轻量级，转为高并发而生。

然后，我们还是可以基于 MySQL 中的一些临时表来存放 kv 存储中提取出来的数据分片，利用 MySQL 对复杂 SQL 语法的支持来进行计算就可以了。也就是说，我们在这个架构里，把 kv 系统作为存储，把 MySQL 用做少量数据的计算。

此时我们在系统架构中引入了分布式 kv 系统来作为我们的数据存储，每天的海量数据都存放在这里就可以了，然后我们的 Slave 计算引擎每次计算，都是根据那个数据分片从 kv 存储中提取对应的数据出来放入 MySQL 内的一个临时表，接着就是对那个临时表内的一两千条数据分片运行各种复杂 SQL 进行计算即可。



大家看上面的图，此时通过这一步计算与存储架构的分离，我们选用了适合支撑高并发的 kv 集群来抗住每天百亿级的流量写入。然后基于 MySQL 作为临时表放入少量数据来进行运算。这一个步骤就直接把高并发请求可以妥妥的抗住了。

而且分布式 kv 存储本来就可以按需扩容，如果并发越来越高，只要扩容增加机器就可以了。**此时，就完成了架构的一个关键的重构步骤。**

四、自研纯内存 SQL 计算引擎 下一步，我们就要对架构追求极致！因为此时我们面临的一个痛点就在于说，其实仅仅只是将 MySQL 作为一个临时表来计算了，主要就是用他的复杂 SQL 语法的支持。

但是问题是，对 MySQL 的并发量虽然大幅度降低了，可是还并不算太低。因为大量的数据分片要计算，还是需要频繁的读写 MySQL。

此外，每次从 kv 存储里提取出来了数据，还得放到 MySQL 的临时表里，还得发送 SQL 去 MySQL 里运算，这还是多了几个步骤的时间开销。

因为当时面临的另外一个问题是，每天请求量大，意味着数据量大，数据量大意味着时间分片的计算任务负载还是较重。

总是这么依赖 MySQL，还要额外维护一大堆的各种临时表，可能多达几百个临时表，你要维护，要注意他的表结构的修改，还有分库分表的一些运维操作，这一切都让依赖 MySQL 这个事

儿显得那么的多余和麻烦。

因此，我们做出决定，为了让架构的维护性更高，而且将性能优化到极致，**我们要自己研发纯内存的 SQL 计算引擎。**

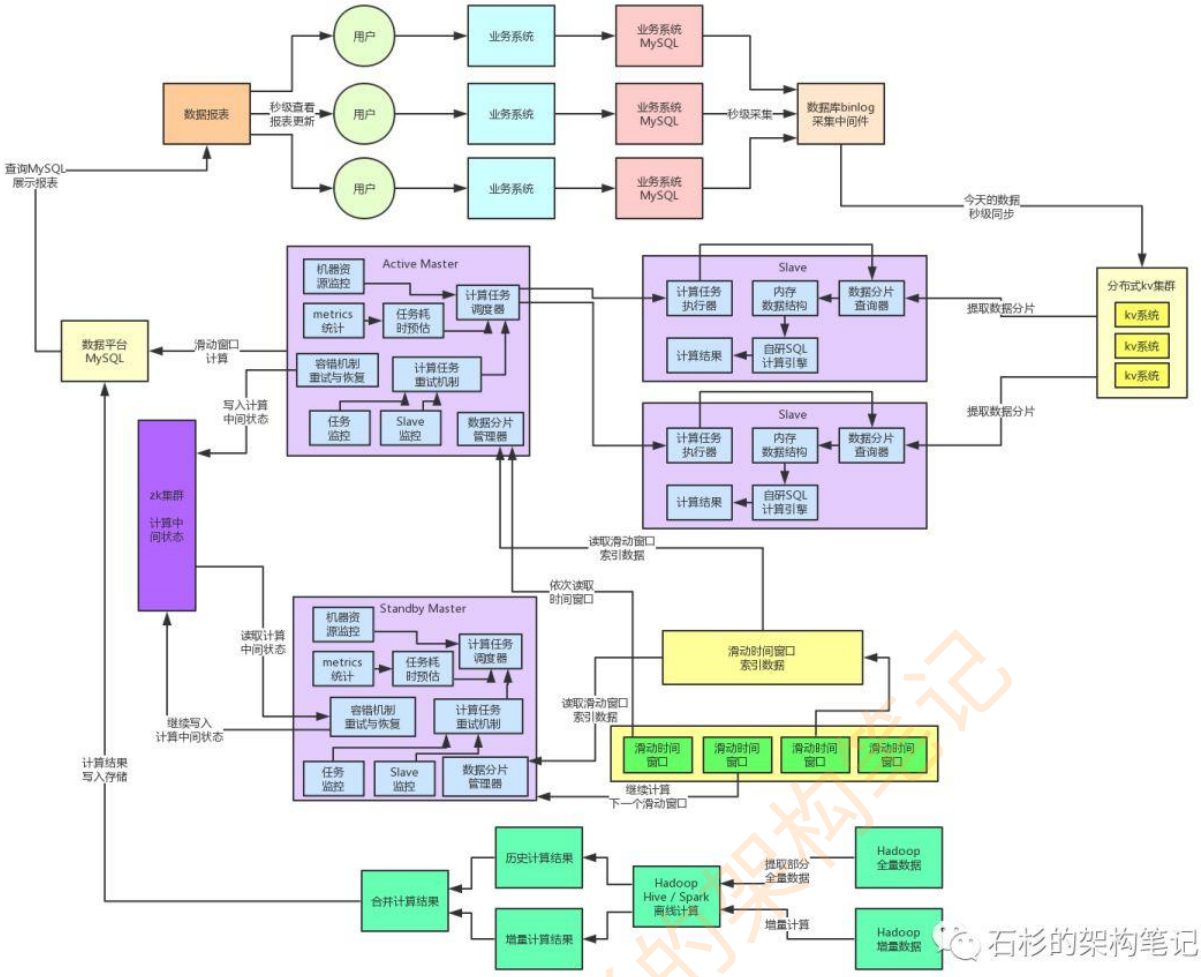
其实如果你要自研一个可以支持 MySQL 那么复杂 SQL 语法的内存 SQL 计算引擎，还是有点难度和麻烦的。但是在我们仔细研究了业务需要的那几百个 SQL 之后，发现其实问题没那么的复杂。

因为其实一般的数据分析类的 SQL，主要就是一些常见的功能，没有那么多的怪、难、偏的 SQL 语法。

因此我们将线上的 SQL 都分析过一遍之后，就针对性的研发出了仅仅支持特定少数语法的 SQL 引擎，包括了嵌套查询组件、多表关联组件、分组聚合组件、多字段排序组件、少数几个常用函数，等等。

接着就将系统彻底重构为不再依赖 MySQL，每次从 kv 存储中提取一个数据分片之后，直接放入内存中，然后用我们自研的 SQL 计算引擎来在纯内存里针对一个数据分片执行各种复杂的 SQL。

这个纯内存操作的性能，那就不用多说了，大家应该都能想象到了，基本上纯内存的 SQL 执行，都是毫秒级的，基本上一个时间分片的运算全部降低到毫秒级了。性能进一步得到了大幅度的提升，而且从此不再依赖 MySQL 了，不需要维护复杂的分库分表等等东西。



这套架构上线之后，彻底消除了对 MySQL 的依赖，理论上，无论多大的流量过来，都可以通过立马扩容 kv 集群以及扩容 Slave 计算集群来解决，不需要依赖 MySQL 的分库分表、几百张临时表等比较耗费人力、麻烦而且坑爹的方案了。而且这种纯内存的计算架构直接把计算性能提升到了毫秒级。

而且消除对 MySQL 的依赖有另外一个好处，数据库的机器总是要高配置的，但是 Slave 机器主要 4 核 8G 的普通虚拟机就够了，**分布式系统的本质就是尽量利用大量的廉价普通机器就可以完成高效的存储和计算。**

因此在百亿流量的负载之下，我们 Slave 机器部署了几十台机器就足够了，那总比你部署几十台昂贵的高配置 MySQL 物理机来的划算多了！

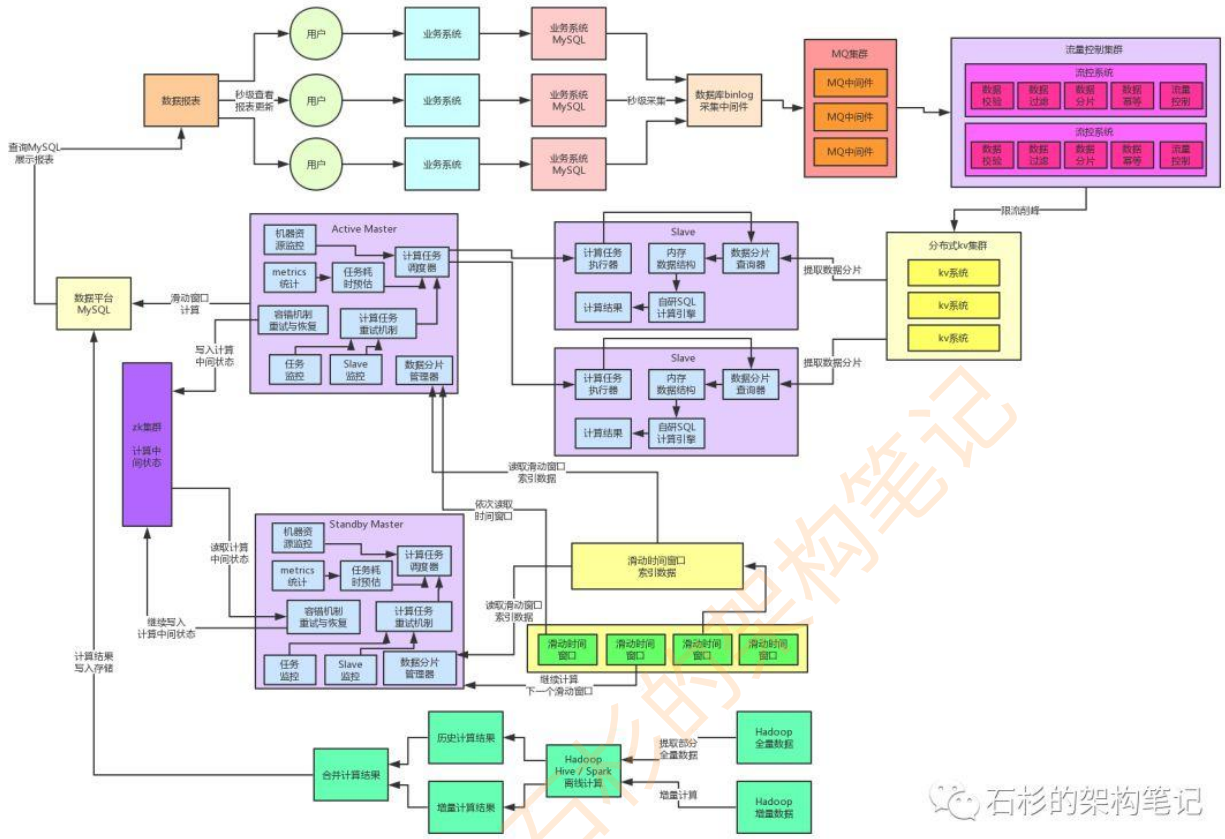
五、MQ 削峰以及流量控制

其实如果对高并发架构稍微了解点的同学都会发现，这个系统的架构中，针对高并发的写入这块，还有一个比较关键的组件要加入，就是 MQ。

因为我们如果应对的是高并发的非实时响应的写入请求的话，完全可以使用 MQ 中间件先抗住海量的请求，接着做一个中间的流量分发系统，将流量异步转发到 kv 存储中去，同时这个流量分发系统可以对高并发流量进行控制。

比如说如果瞬时高并发的写入真的导致后台系统压力过大，那么就可以由流量分发系统自动根据我们设定的阈值进行流量控制，避免高并发的压力打垮后台系统。

而且在这个流控系统中，我们其实还做了很多的细节性的优化，比如说数据校验、过滤无效数据、切分数据分片、数据同步的幂等机制、100% 保证数据落地到 kv 集群的机制保障，等等。



公司的 MQ 集群天然都支撑过大流量写入以及高并发请求，因此 MQ 集群那个层面抗住高并发并不是什么问题，再高的并发按需扩容就可以了，然后我们自己的流控系统也是集群部署的，线上采用的是 4 核 8G 的虚拟机，因为这个机器不需要太高的配置。

流控系统，基本线上我们一般保持在每台机器承载每秒小三千左右的并发请求，百亿流量场景下，高峰每秒并发在每秒小几十万的级别，因此这个流控集群部署到几十台机器就足够了。

而公司的 kv 集群也是天然支撑过大流量高并发写入的，因此 kv 集群按需扩容，抗住高并发带流量的写入也不是什么问题，而且这里其实我们因为在自身架构层面做了大量的优化（存储与计算分离的关键点），因此 kv 集群的定位基本就是 online storage，一个在线存储罢了。

通过合理、巧妙的设计 key 以及 value 的数据类型，使得我们对 kv 集群的读写请求都是优化成最最简单的 key-value 的读写操作，天然保证高并发读写是没问题的。

另外稍微给大家一点点的剧透，后面讲到全链路 99.99% 高可用架构的时候，这个流控集群会发挥巨大的作用，他是承上启下的一个效果，前置的 MQ 集群故障的高可用保障，以及后置的 KV 集群故障的高可用保障，都是依靠流控集群来实现的。

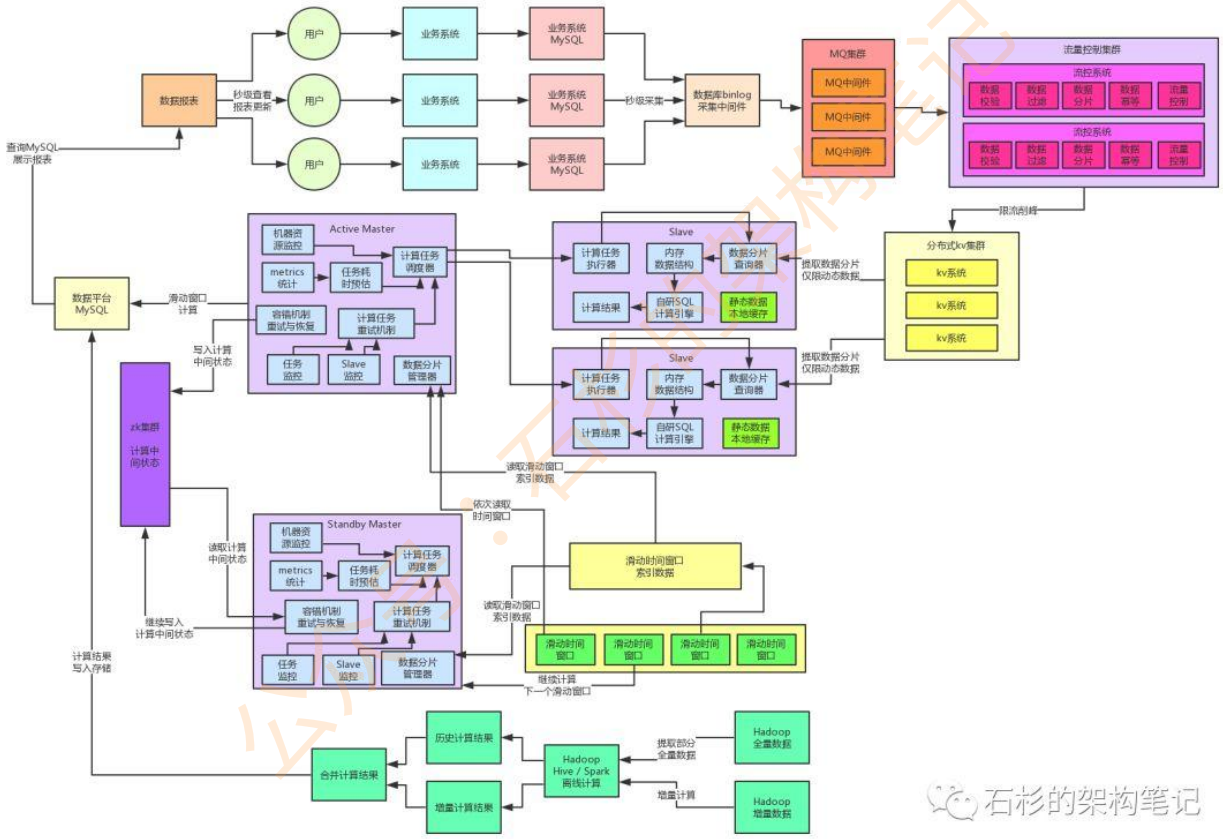
六、数据的动静分离架构

在完成上述重构之后，我们又对核心的自研内存 SQL 计算引擎做了进一步的优化。因为实际生产环境运行过程中，我们发现了一个问题：就是每次如果 Slave 节点都是对一个数据分片提取相关联的各种数据出来然后进行计算，其实是没必要的！

给大家举个例子，如果你的 SQL 要对一些表进行关联计算，里面涉及到了一些大部分时候静态不变的数据，那些表的数据一般很少改变，因此没必要每次都走网络请求从 kv 存储里提取那部分数据。

我们其实完全可以在 Slave 节点对这种静态数据做个轻量级的 cache，然后只有数据分片里对应的动态改变的数据才从 kv 存储来提取数据。

通过这个数据的动静分离架构，我们基本上把 Slave 节点对 kv 集群的网络请求降低到了最少，性能提升到了最高。大家看下面的图。



七、阶段性总结

这套架构到此为止，基本上就演进的比较不错了，因为超高并发写入、极速高性能计算、按需任意扩容，等各种特性都可以支持到了，基本上从写入到计算，这两个步骤，是没什么太大的瓶颈了。

而且通过自研内存 SQL 计算引擎的方案，将我们的实时计算性能提升到了毫秒级的标准，基本已经达到极致。

八、下一步展望

下一步，我们就要看看这个架构中的左侧，还有一个 MySQL 呢！

首先是实时计算链路和离线计算链路，都会导入大量的计算结果到那个 MySQL 中。

其次面向数十万甚至上百万的 B 端商家时，如果是实时展示数据分析结果的话，一般页面上会有定时的 JS 脚本，每隔几秒钟就会发送请求过来加载最新的数据计算结果。

因此实际上那个专门面向终端用户的 MySQL 也会承受极大的数据量的压力，高并发写入的压力以及高并发查询的压力。

亿级流量系统架构之如何设计每秒十万查询的高并发架构

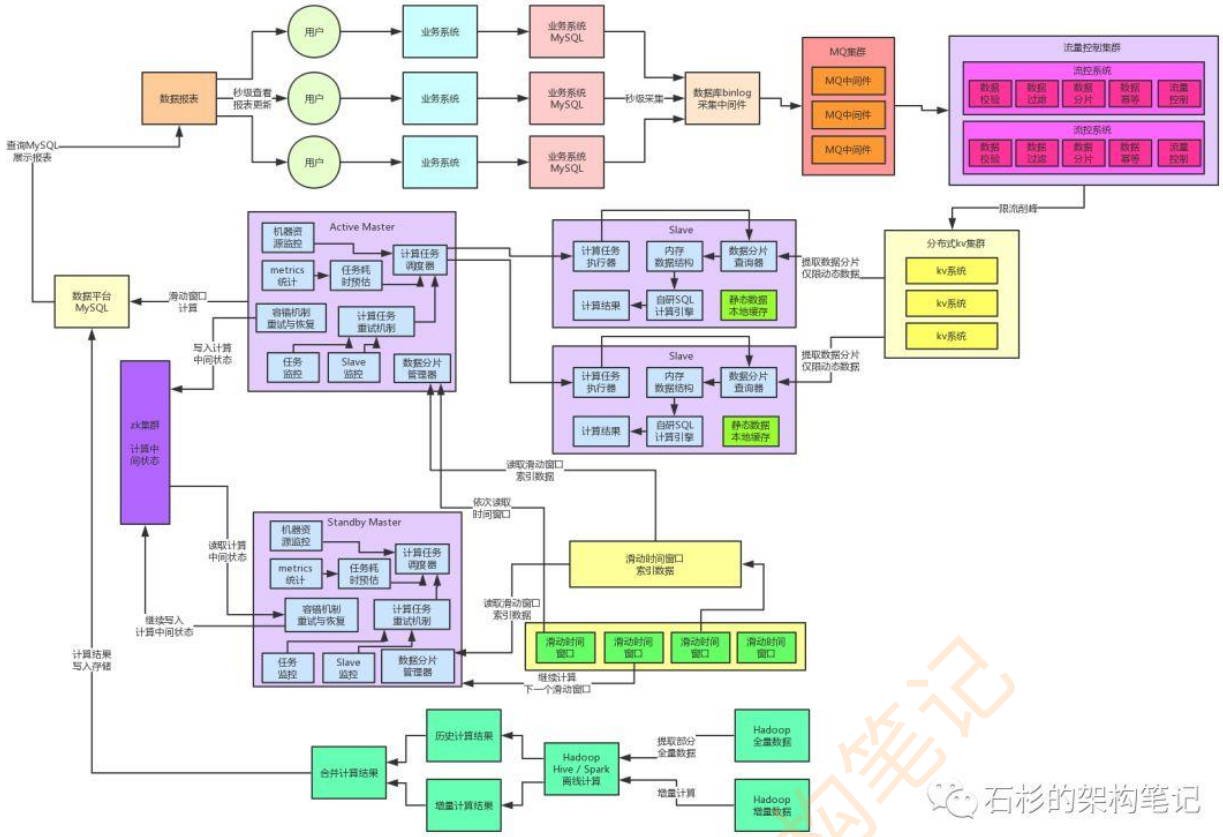
> **作者:中华石杉** 原文地址

一 前情回顾

上篇文章**《大型系统架构演进之如何设计承载百亿流量的高性能架构》**聊了一下系统架构中，百亿流量级别高并发写入场景下，如何承载这种高并发写入，同时如何在高并发写入的背景下还能保证系统的超高性能计算。

这篇文章咱们继续来聊一下，百亿级别的海量数据场景下还要支撑每秒十万级别的高并发查询，这个架构该如何演进和设计?

咱们先来看看目前系统已经演进到了什么样的架构，**大家看看下面的图:**



首先回顾一下，整个架构右侧部分演进到的那个程度，其实已经非常的不错了，因为百亿流量，每秒十万级并发写入的场景，使用 MQ 限流削峰、分布式 KV 集群给抗住了。

接着使用了计算与存储分离的架构，各个 Slave 计算节点会负责提取数据到内存中，基于自研的 SQL 内存计算引擎完成计算。同时采用了数据动静分离的架构，静态数据全部缓存，动态数据自动提取，保证了尽可能把网络请求开销降低到最低。

另外，通过自研的分布式系统架构，包括数据分片和计算任务分布式执行、弹性资源调度、分布式高容错机制、主备自动切换机制，都能保证整套系统的任意按需扩容，高性能、高可用的的运行。

下一步，咱们来研究研究架构里的左侧部分

二 日益膨胀的离线计算结果

其实大家会注意到，在左侧还有一个 MySQL，那个 MySQL 就是用来承载实时计算结果和离线计算结果放在里面汇总的。

终端的商家用户就可以随意的查询 MySQL 里的数据分析结果，支撑自己的决策，他可以看当天的数据分析报告，也可以看历史上任何一段时期内的数据分析报告。

但是那个 MySQL 在早期可能还好一些，因为其实存放在这个 MySQL 里的数据量相对要小一些，毕竟是计算后的一些结果罢了。但是到了中后期，这个 MySQL 可是也岌岌可危了。

给大家举一个例子，离线计算链路里，如果每天增量数据是 1000 万，那么每天计算完以后的结果大概只有 50 万，每天 50 万新增数据放入 MySQL，其实还是可以接受的。

但是如果每天增量数据是 10 亿，那么每天计算完以后的结果大致会是千万级，你可以算他是计算结果有 5000 万条数据吧，每天 5000 万增量数据写入左侧的 MySQL 中，你觉得是啥感觉?

可以给大家说说系统当时的情况，基本上就是，单台 MySQL 服务器的磁盘存储空间很快就要接近满掉，而且单表数据量都是几亿、甚至十亿的级别。

这种量级的单表数据量，你觉得用户查询数据分析报告的时候，体验能好么？基本当时一次查询都是几秒钟的级别。很慢。

更有甚者，出现过用户一次查询要十秒的级别，甚至几十秒，上分钟的级别。很崩溃，用户体验很差，远远达不到付费产品的级别。

所以解决了右侧的存储和计算的问题之后，左侧的查询的问题也迫在眉睫。新一轮的重构，势在必行!

三 分库分表 + 读写分离

首先就是老一套，分库分表 + 读写分离，这个基本是基于 MySQL 的架构中，必经之路了，毕竟实施起来难度不是特别的高，而且速度较快，效果比较显著。

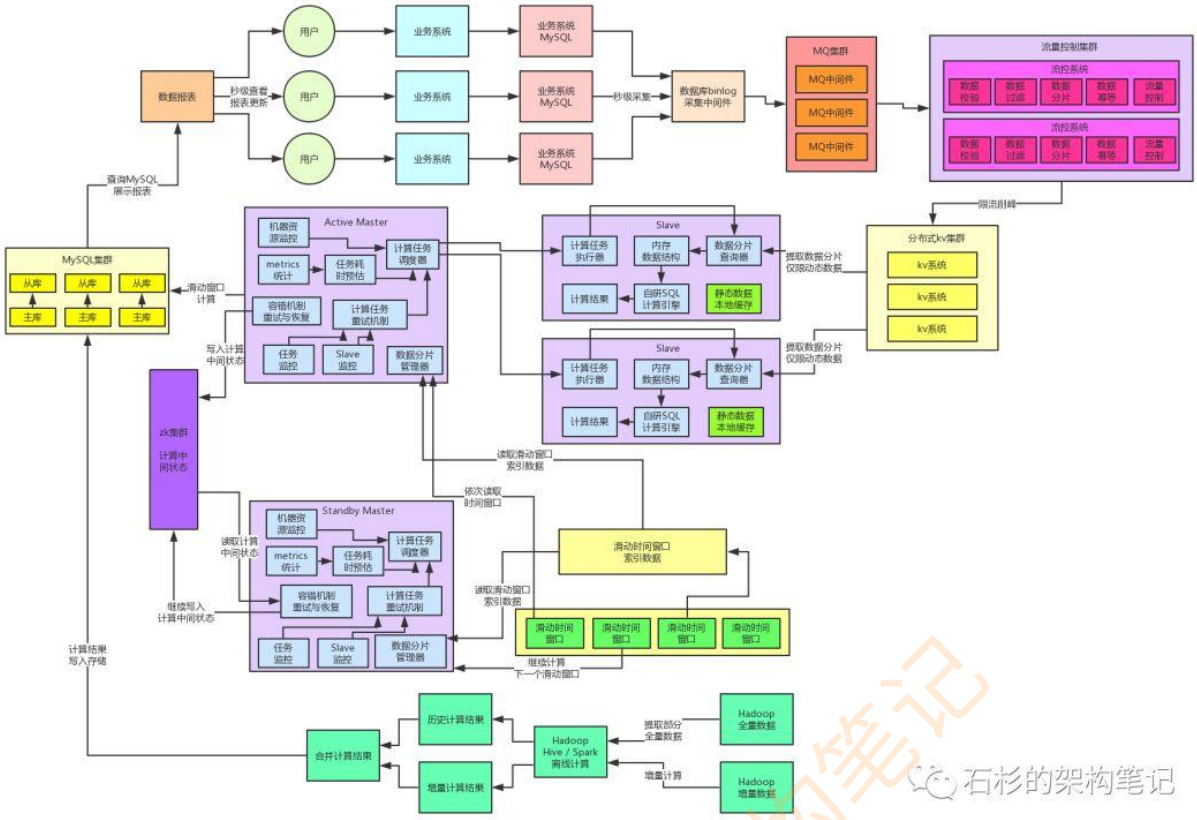
整个的思路和之前第一篇文章**《大型系统架构演进之如何支撑百亿级数据的存储与计算》** 讲的基本一致。

说白了，就是分库后，每台主库可以承载部分写入压力，单库的写并发会降低；其次就是单个主库的磁盘空间可以降低负载的数据量，不至于很快就满了；

而分表之后，单个数据表的数据量可以降低到百万级别，这个是支撑海量数据以及保证高性能的最佳实践，基本两三百万的单表数据量级还是合理的。

然后读写分离之后，就可以将单库的读写负载压力分离到主库和从库多台机器上去，主库就承载写负载，从库就承载读负载，这样避免单库所在机器的读写负载过高，导致 CPU 负载、IO 负载、网络负载过高，最后搞得数据库机器宕机。

首先这么重构一下数据库层面的架构之后，效果就好的多了。因为单表数据量降低了，那么用户查询的性能得到很大的提升，基本可以达到 1 秒以内的效果。



四 每秒 10 万查询的高并发挑战

上面那套初步的分库分表 + 读写分离的架构确实支撑了一段时间，但是慢慢的那套架构又暴露出来了弊端出来了，因为商家用户都是开了数据分析页面之后，页面上有 js 脚本会每隔几秒钟就发送一次请求到后端来加载最新的数据分析结果。

此时就有一个问题了，渐渐的查询 MySQL 的压力越来越大，基本上可预见的范围是朝着每秒 10 级别去走。

但是我们分析了一下，其实 99% 的查询，都是页面 JS 脚本自动发出刷新当日数据的查询。只有 1% 的查询是针对昨天以前的历史数据，用户手动指定查询范围后来查询的。

但是现在的这个架构之下，我们是把当日实时数据计算结果（代表了热数据）和历史离线计算结果（代表了冷数据）都放在一起的，所以大家可以想象一下，热数据和冷数据放在一起，然后对热数据的高并发查询占到了 99%，那这样的架构还合理吗？

当然不合理，我们需要再次重构系统架构。

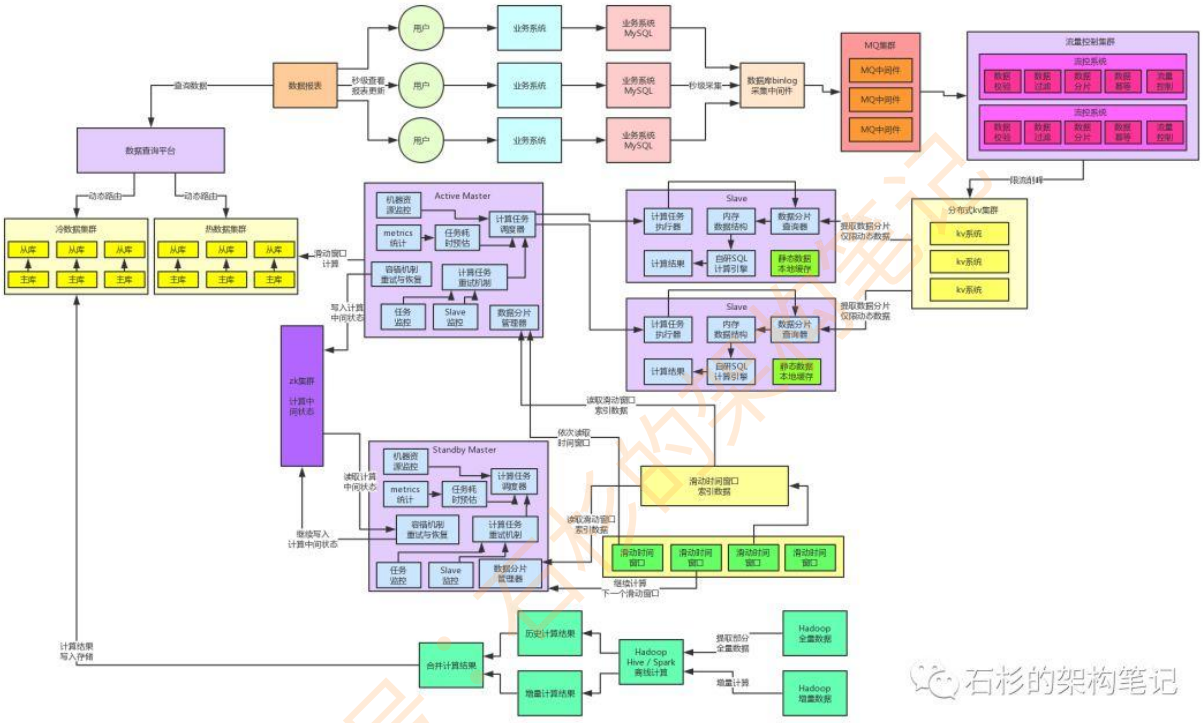
五 数据的冷热分离架构

针对上述提到的问题，很明显要做的一个架构重构就是冷热数据分离。也就是说，将今日实时计算出来的热数据放在一个 MySQL 集群里，将离线计算出来的冷数据放在另外一个 MySQL 集群里。

然后开发一个数据查询平台，封装底层的多个 MySQL 集群，根据查询条件动态路由到热数据存储或者是冷数据存储。

通过这个步骤的重构，我们就可以有效的将热数据存储中单表的数据量降低到更少更少，有的单表数据量可能就几十万，因为将离线计算的大量数据结果从表里剥离出去了，放到另外一个集群里去。此时大家可想而知，效果当然是更好了。

因为热数据的单表数据量减少了很多，当时的一个最明显的效果，就是用户 99% 的查询都是针对热数据存储发起的，性能从原来的 1 秒左右降低到了 200 毫秒以内，用户体验提升，大家感觉更好了。



六 自研 Elasticsearch+HBase + 纯内存的查询引擎

架构演进到这里，看起来好像还不错，但是其实问题还是很多。因为到了这个阶段，系统遇到了另外一个较为严重的问题：冷数据存储，如果完全用 MySQL 来承载是很不靠谱的。冷数据的数据量是日增长不断增加，而且增速很快，每天都新增几千万。

因此你的 MySQL 服务器将会面临不断的需要扩容的问题，而且如果为了支撑这 1% 的冷数据查询请求，不断的扩容增加高配置的 MySQL 服务器，大家觉得靠谱么？

肯定是不合适的！

要知道，大量分库分表后，MySQL 大量的库和表维护起来是相当麻烦的，修改个字段？加个索引？这都是一场麻烦事儿。

此外，因为对冷数据的查询，一般都是针对大量数据的查询，比如用户会选择过去几个月，甚至一年的数据进行分析查询，此时如果纯用 MySQL 还是挺灾难性的。

因为当时明显发现，针对海量数据场景下，一下子查询分析几个月或者几年的数据，性能是极差的，还是很容易搞成几秒甚至几十秒才出结果。

因此针对这个冷数据的存储和查询的问题，**我们最终选择了自研一套基于 NoSQL 来存储，然后基于 NoSQL + 内存的 SQL 计算引擎**。

具体来说，我们会将冷数据全部采用 ES+HBase 来进行存储，ES 中主要存放要对冷数据进行筛选的各种条件索引，比如日期以及各种维度的数据，然后 HBase 中会存放全量的数据字段。

因为 ES 和 HBase 的原生 SQL 支持都不太好，因此我们直接自研了另外一套 SQL 引擎，专门支持这种特定的场景，就是基本没有多表关联，就是对单个数据集进行查询和分析，然后支持 NoSQL 存储 + 内存计算。

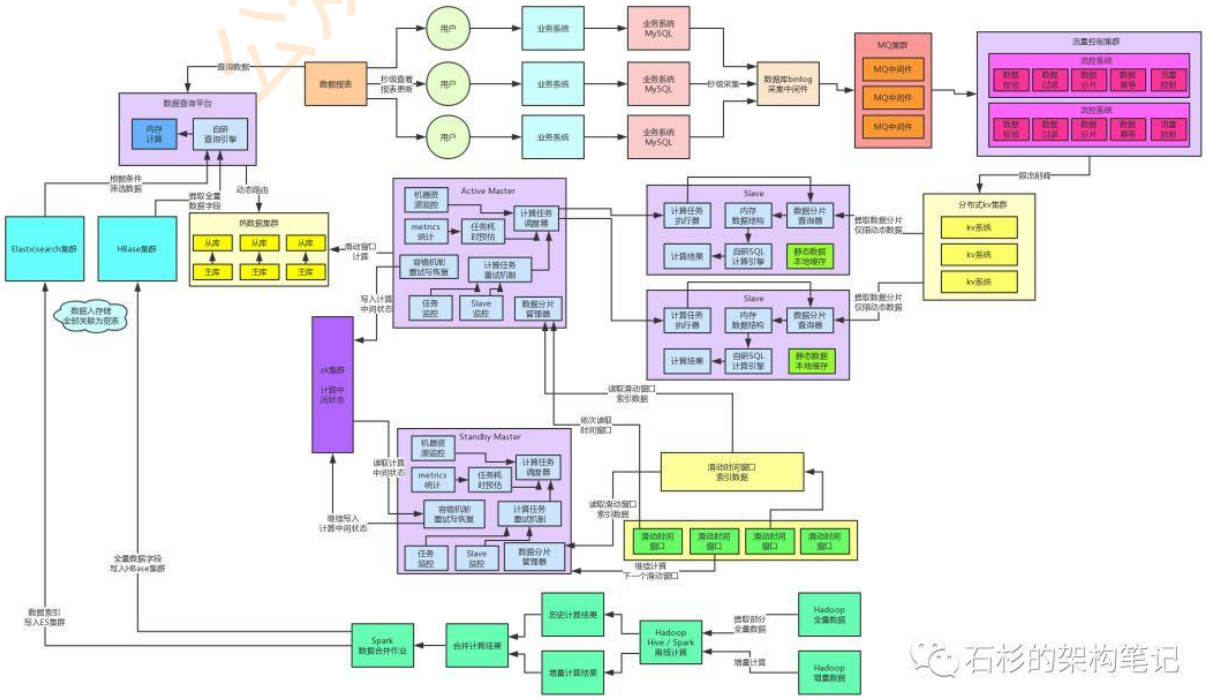
这里有一个先决条件，就是如果要做到对冷数据全部是单表类的数据集查询，必须要在冷数据进入 NoSQL 存储的时候，全部基于 ES 和 HBase 的特性做到多表入库关联，进数据存储就全部做成大宽表的状态，将数据关联全部上推到入库时完成，而不是在查询时进行。

对冷数据的查询，我们自研的 SQL 引擎首先会根据各种 where 条件先走 ES 的分布式高性能索引查询，ES 可以针对海量数据高性能的检索出来需要的那部分数据，这个过程用 ES 做是最合适的。

接着就是将检索出来的数据对应的完整的各个数据字段，从 HBase 里提取出来，拼接成完成的数据。

然后就是将这份数据集放在内存里，进行复杂的函数计算、分组聚合以及排序等操作。

上述操作，全部基于自研的针对这个场景的查询引擎完成，**底层基于 Elasticsearch、HBase、纯内存来实现**。



七 实时数据存储引入缓存集群

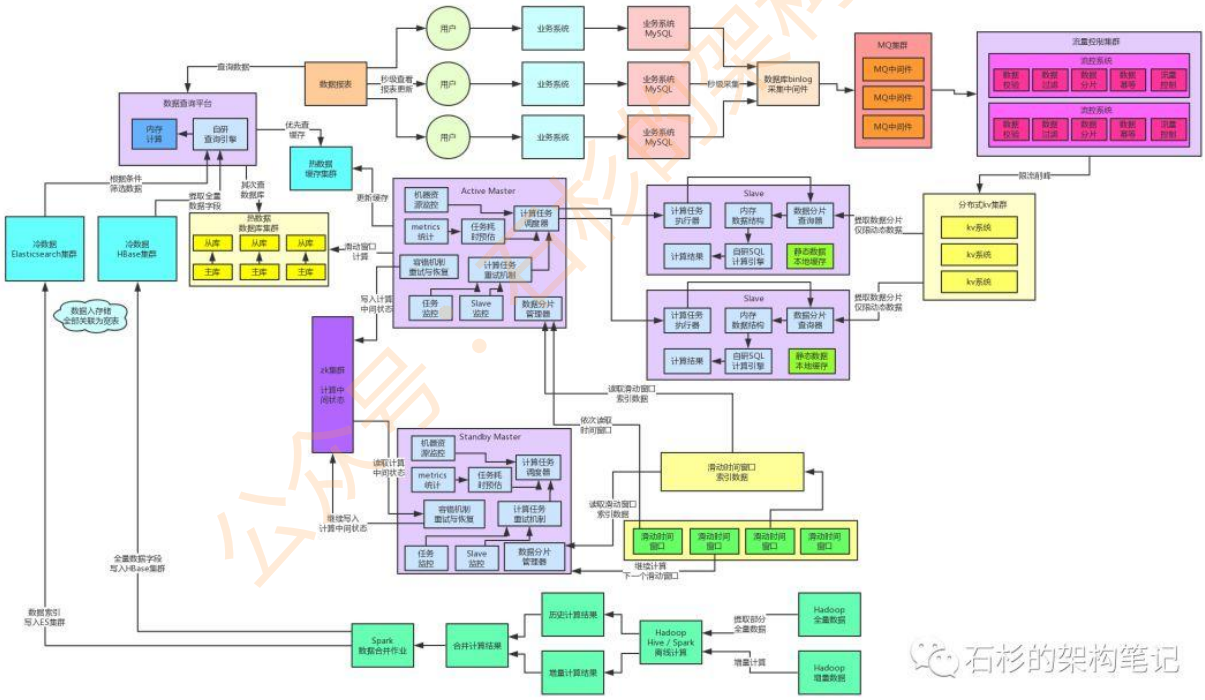
好了，到此为止，冷数据的海量数据存储、高性能查询的问题，就解决了。接着回过头来看看当日实时数据的查询，其实实时数据的每日计算结果不会太多，而且写入并发不会特别特别的高，每秒上万也就差不多了。

因此这个背景下，就是用 MySQL 分库分表来支撑数据的写入、存储和查询，都没问题。

但是有一个小问题，就是说每个商家的实时数据其实不是频繁的变更的，在一段时间内，可能压根儿没变化，因此不需要高并发请求，每秒 10 万级别的全部落地到数据库层面吧？要全都落地到数据库层面，那可能要给每个主库挂载很多从库来支撑高并发读。

因此这里我们引入了一个缓存集群，实时数据每次更新后写入的时候，都是写数据库集群同时还写缓存集群的，是双写的方式。

然后查询的时候是优先从缓存集群来走，此时基本上 90% 以上的高并发查询都走缓存集群了，然后只有 10% 的查询会落地到数据库集群。



八 阶段性总结

好了，到此为止，这个架构基本左边也都重构完毕：

> ! 热数据基于缓存集群 + 数据库集群来承载高并发的每秒十万级别的查询

> 冷数据基于 ES+HBase + 内存计算的自研查询引擎来支撑海量数据存储以及高性能查询。

经实践，整个效果非常的好。用户对热数据的查询基本多是几十毫秒的响应速度，对冷数据的查询基本都是 200 毫秒以内的响应速度。

九 下一阶段的展望

其实架构演进到这里已经很不容易了，因为看似这么一张图，里面涉及到无数的细节和技术方案的落地，需要一个团队耗费至少 1 年的时间才能做到这个程度。

但是接下来，我们要面对的，就是高可用的问题，因为付费级的产品，我们必须要保证超高的可用性，99.99% 的可用性，甚至是 99.999% 的可用性。

但是越是复杂的系统，越容易出现问题，对应的高可用架构就越是复杂无比，因此下篇文章，我们聊聊：**《亿级流量系统架构之如何设计全链路 99.99% 高可用架构》**。

亿级流量系统架构之如何设计全链路99.99%高可用架构

> **作者:中华石杉** **原文地址**

一、前情回顾

上篇文章**《亿级流量系统架构之如何设计每秒十万查询的高并发架构》**，聊了一下系统架构中的查询平台。

我们采用冷热数据分离：

> 冷数据基于 HBase+Elasticsearch + 纯内存自研的查询引擎，解决了海量历史数据的高性能毫秒级的查询

> 热数据基于缓存集群 + MySQL 集群做到了当日数据的几十毫秒级别的查询性能。

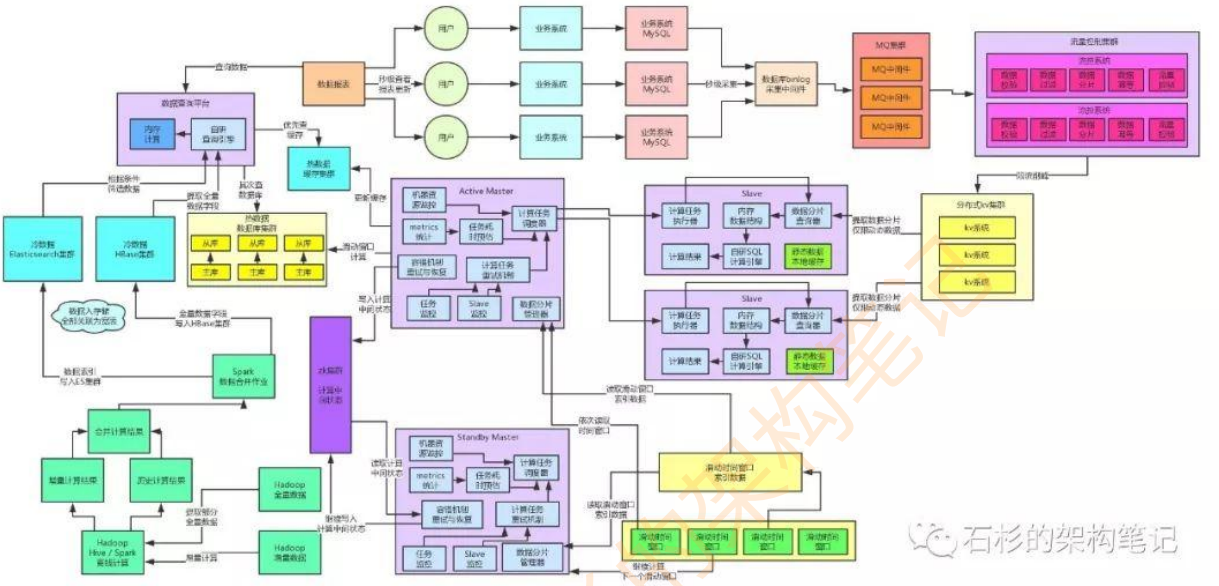
最终，整套查询架构抗住每秒 10 万的并发查询请求，都没问题。

本文作为这个架构演进系列的最后一篇文章，我们来聊聊高可用这个话题。所谓的高可用是啥意思呢?

简单来说，就是如此复杂的架构中，任何一个环节都可能会故障，比如 MQ 集群可能会挂掉、KV 集群可能会挂掉、MySQL 集群可能会挂掉。那你怎么才能保证说，你这套复杂架构中任何一个环节挂掉了，整套系统可以继续运行?

这就是所谓的**全链路 99.99% 高可用架构**，因为我们的平台产品是付费级别的，付费级别，必须要为客户做到最好，可用性是务必要保证的!

我们先来看看目前为止的架构是长啥样子的。



二、MQ 集群高可用方案

异步转同步 + 限流算法 + 限制性丢弃流量

MQ 集群故障其实是有概率的，而且挺正常的，因为之前就有的大型互联网公司，MQ 集群故障之后，导致全平台几个小时都无法交易，严重的会造成几个小时公司就有数千万的损失。我们之前也遇到过 MQ 集群故障的场景，但是并不是这个系统里。

大家想一下，如果这个链路中，万一 MQ 集群故障了，会发生什么?

看看右上角那个地方，数据库 binlog 采集中间件就无法写入数据到 MQ 集群了啊，然后后面的流控集群也无法消费和存储数据到 KV 集群了。这套架构将会彻底失效，无法运行。

这个是我们想要的效果吗？那肯定不是的，如果是这样的效果，这个架构的可用性保障也太差了。

因此在这里，我们针对 MQ 集群的故障，设计的高可用保障方案是：**异步转同步 + 限流算法 + 限制性丢弃流量。**

简单来说，数据库 binlog 采集环节一旦发现了 MQ 集群故障，也就是尝试多次都无法写入数据到 MQ 集群，此时就会触发降级策略。不再写入数据到 MQ 集群，而是转而直接调用流控集群

提供的备用流量接收接口，直接发送数据给流控集群。

但是流控集群也比较尴尬，之前用 MQ 集群就是削峰的啊，高峰期可以稍微积压一点数据在 MQ 集群里，避免流量过大，冲垮后台系统。

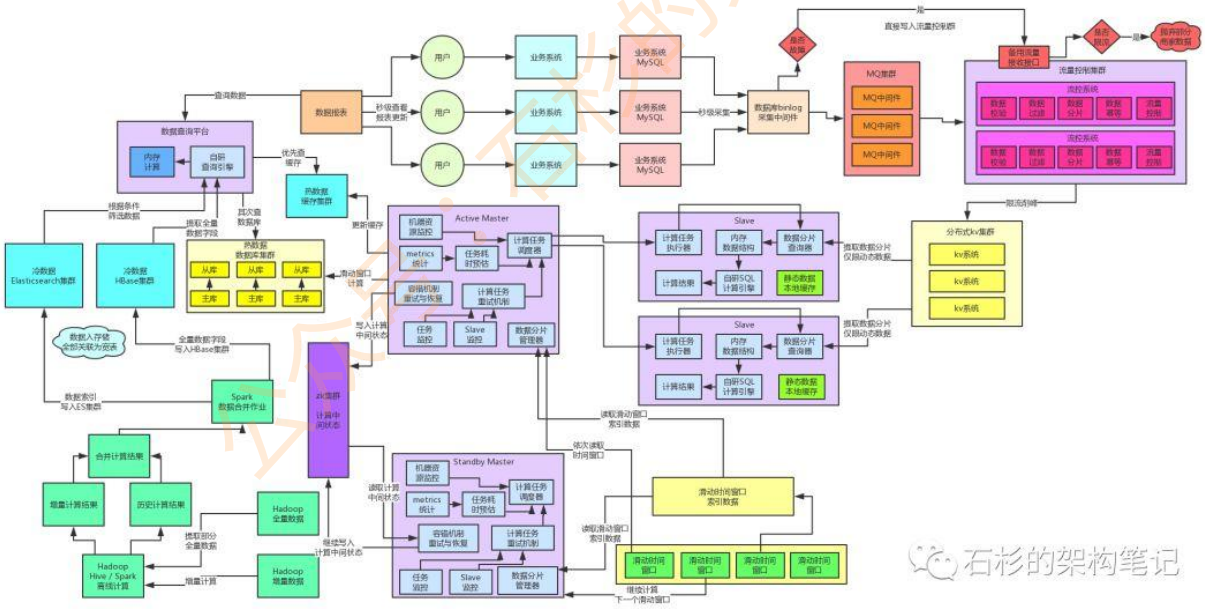
所以流控集群的备用流量接收接口，都是实现了限流算法的，也就是如果发现一旦流量过大超过了阈值，直接采取丢弃的策略，抛弃部分流量。

但是这个抛弃部分流量也是有讲究的，你要怎么抛弃流量？如果你不管三七二十一，胡乱丢弃流量，可能会导致所有的商家看到的数据分析结果都是不准确的。因此当时选择的策略是，仅仅选择少量商家的数据全量抛弃，但是大部分商家的数据全量保存。

也就是说，比如你的平台用户有 20 万吧，可能在这个丢弃流量的策略下，有 2 万商家会发现看不到今天的数据了，但是 18 万商家的数据是不受影响，都是准确的。但是这个总比 20 万商家的数据全部都是不准确的好吧，所以在降级策略制定的时候，都是有权衡的。

这样的话，在 MQ 集群故障的场景下，虽然可能会丢弃部分流量，导致最终数据分析结果有偏差，但是大部分商家的数据都是正常的。

大家看看下面的图，**高可用保障环节全部选用浅红色来表示**，这样很清晰。



三、KV 集群高可用保障方案

临时扩容 Slave 集群 + 内存级分片存储 + 小时级数据粒度

下一个问题，如果 KV 集群挂了怎么办？这个问题我们还真的遇到过，不过也不是在这个系统里，是在另外一个我们负责过的核心系统里，KV 集群确实出过故障，直接从持续好多个小时，导致公司业务都几近于停摆，损失也是几千万级别的。

大家看看那个架构图的右侧部分，如果 KV 集群挂了咋办？那也是灾难性的，因为我们的架构选型里，直接就是基于 kv 集群来进行海量数据存储的，要是 KV 挂了，没任何高可用保障措施的话，会导致流控集群无法把数据写入 KV 集群，此时后续环节就无法继续计算了。

我们当时考虑过要不要引入另外一套存储进行双写，比如引入一套 hbase 集群，但是那样依赖会搞的更加的复杂，打铁还需自身硬，还是要从自身架构来做优化。

因此，当时选择的一套 kv 集群降级的预案是：临时扩容 Slave 集群 + 小时级数据粒度 + 内存级分片存储。

简单来说，就是一旦发现 kv 集群故障，直接报警。我们收到报警之后，就会立马启动临时预案，手动扩容部署 N 倍的 Slave 计算集群。

接着同样会手动打开流控集群的一个降级开关，然后流控集群会直接按照预设的 hash 算法分发数据到各个 Slave 计算节点。

这就是关键点，不要再基于 kv 集群存数据了，本身我们的 Slave 集群就是分布式计算的，那不是刚好可以临时用作分布式存储吗！直接流控集群分发数据到 Slave 集群就行了，Slave 节点将数据留存在内存中即可。

然后 Master 节点在分发数据计算任务的时候，会保证计算任务分发到某个 Slave 节点之后，他只要基于本地内存中的数据计算即可。

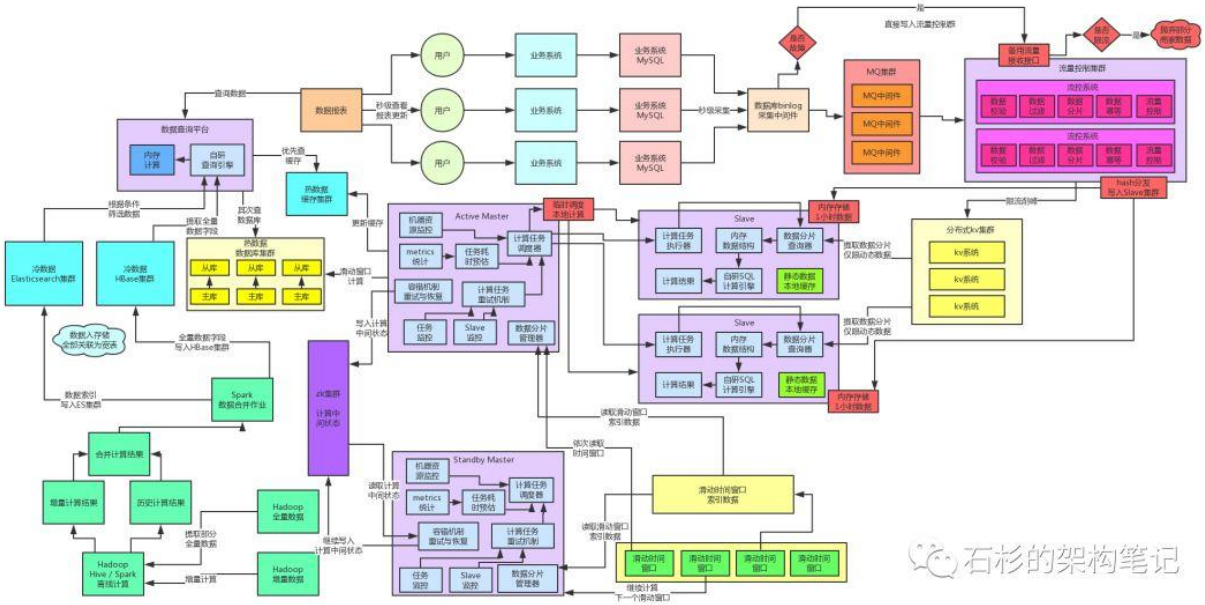
将 Master 节点和 Slave 节点都重构一下，重构成本不会太高，但是这样就实现了本地数据存储 + 本地数据计算的效果了。

但是这里同样有一个问题，要知道当日数据量可是很大的！如果你都放 Slave 集群内存里还得了？

所以说，既然是降级，又要做一个 balance 了。我们选择的是小时级数据粒度的方案，也就是说，仅仅在 Slave 集群中保存最近一个小时的数据，然后计算数据指标的时候，只能产出每个小时的数据指标。

但是如果是针对一天的数据需要计算出来的数据指标，此时降级过后就无法提供了，因为内存中永远只有最近一个小时的数据，这样才能保证 Slave 集群的内存不会被撑爆。

对用户而言，就是只能看当天每个小时的数据指标，但是全天汇总的暂时就无法看到。



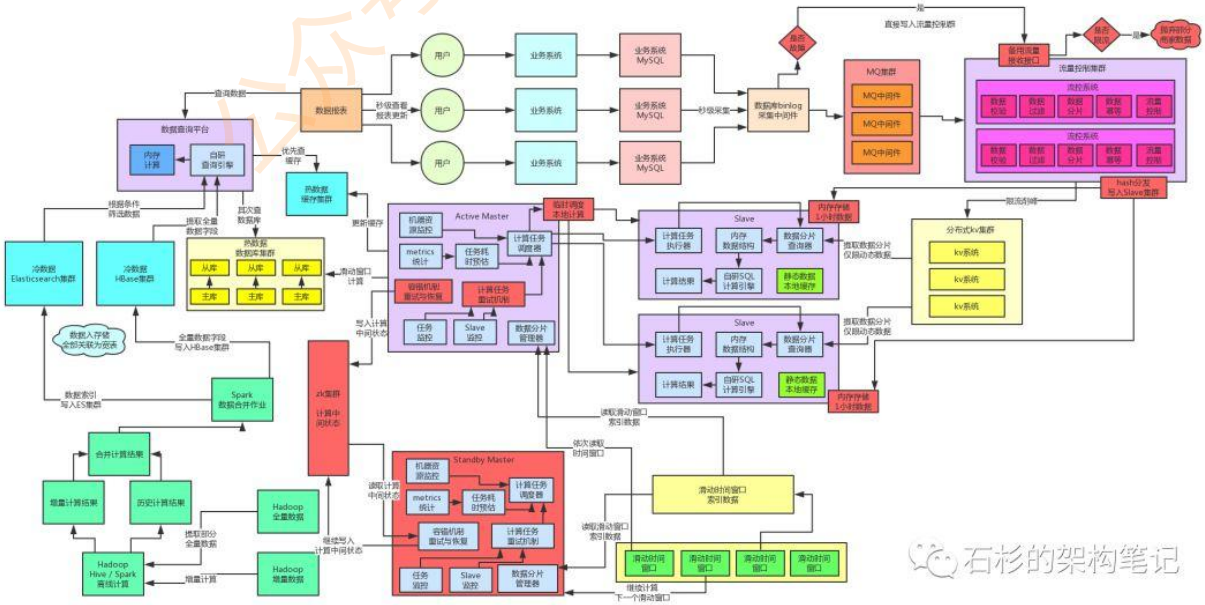
四、实时计算链路高可用保障方案

计算任务重分配 + 主备切换机制

下一块就是实时计算链路的高可用保障方案了，其实这个之前给大家说过了，实时计算链路是一个分布式的架构，所以要么是 Slave 节点宕机，要么是 Master 节点宕机。

其实这个倒没什么，因为 Slave 节点宕机，Master 节点感知到了，会重新分配计算任务给其他的计算节点；如果 Master 节点宕机，就会基于 Active-Standby 的高可用架构，自动主备切换。

咱们直接把架构图里的实时计算链路中的高可用环节标成红色就可以了。



五、热数据高可用保障方案

自研缓存集群查询引擎 + JVM 本地缓存 + 限流机制

接着咱们来看左侧的数据查询那块，热数据也就是提供实时计算链路写入当日数据的计算结果的，用的是 MySQL 集群来承载主体数据，然后前面挂载一个缓存集群。

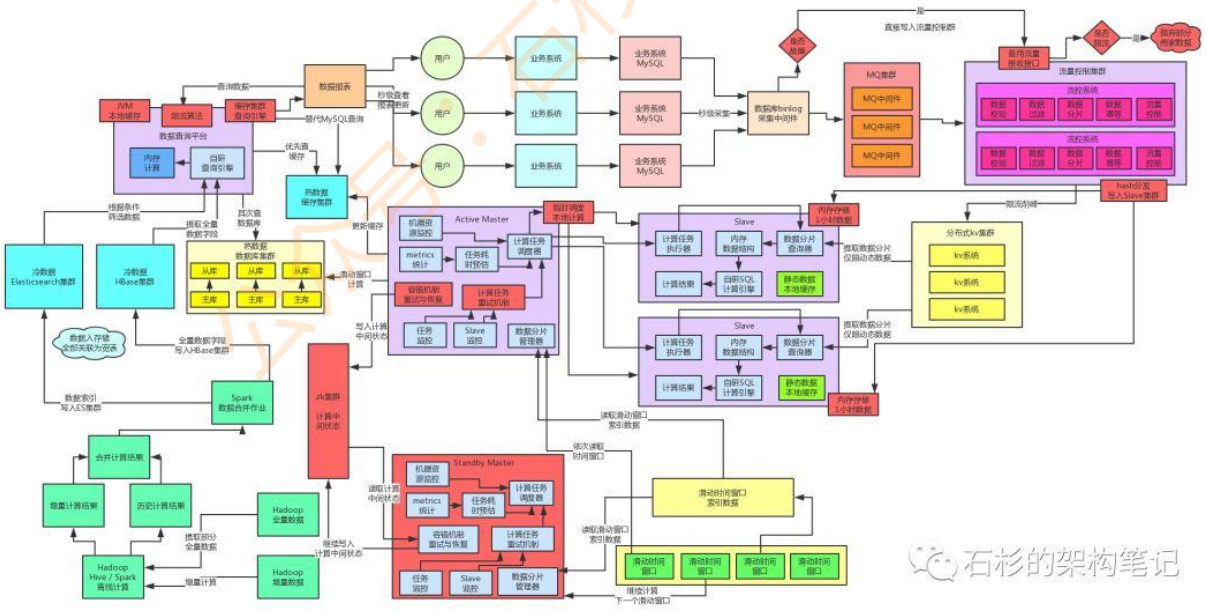
如果出现故障，只有两种情况：一种是 MySQL 集群故障，一种是缓存集群故障。

咱们分开说，如果是 MySQL 集群故障，我们采取的方案是：实时计算结果直接写入缓存集群，然后因为没有 MySQL 支撑，所以没法使用 SQL 来从 MySQL 中组装报表数据。

因此，我们自研了一套基于缓存集群的内存级查询引擎，支持简单的查询语法，可以直接对缓存集群中的数据实现条件过滤、分组聚合、排序等基本查询语义，然后直接对缓存中的数据查询分析过后返回。

但是这样唯一的不好，就是缓存集群承载的数据量远远没有 MySQL 集群大，所以会导致部分用户看不到数据，部分用户可以看到数据。不过这个既然是降级，那肯定是要损失掉部分用户体验的。

如果是缓存集群故障，我们会有一个查询平台里的本地缓存，使用 ehcache 等框架就可以实现，从 mysql 中查出来的数据在查询平台的 jvm 本地缓存里 cache 一下，也可以用作一定的缓存支撑高并发的效果。而且查询平台实现限流机制，如果查询流量超过自身承载范围，就限流，直接对查询返回异常响应。



六、冷数据高可用保障方案

收集查询日志 + 离线日志分析 + 缓存高频查询

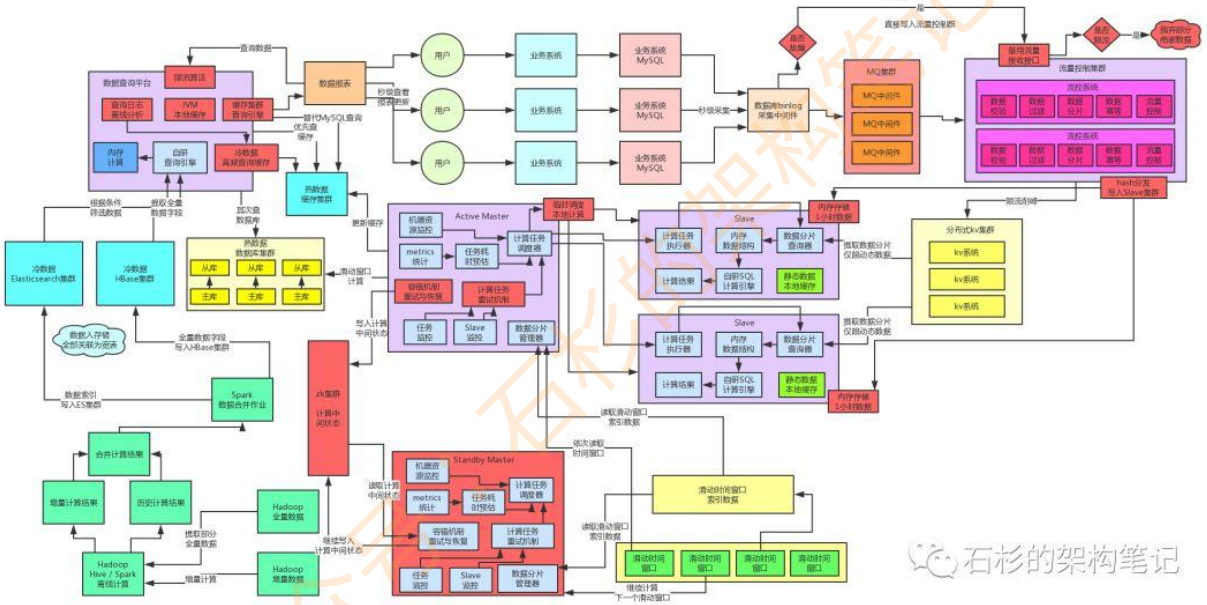
其实大家看上面的图就知道，冷数据架构本身就比比较复杂，涉及到 ES、HBase 等东西，如果你要是想做到一点 ES、HBase 宕机，然后还搞点儿什么降级方案，还是挺难的。

你总不能 ES 不能用了，临时走 Solr？或者 HBase 不能用了，临时走 KV 集群？都不行。那个实现复杂度太高，不合适。

所以当时我们采取的方法就是，对最近一段时间用户发起的离线查询的请求日志进行收集，然后对请求日志在每天凌晨进行分析，分析出来那种每个用户会经常、多次、高频发起的冷数据查询请求，然后对这个特定的查询（比如特殊的一组条件，时间范围，维度组合）对应的结果，进行缓存。

这样就直接把各个用户高频发起的冷数据查询请求的结果每天动态分析，动态放入缓存集群中。比如有的用户每天都会看一下上周一周的数据分析结果，或者上个月一个月的数据分析结果，那么就可以把这些结果提前缓存起来。

一旦 ES、HBase 等集群故障，直接对外冷数据查询，仅仅提供这些提前缓存好的高频查询即可，非高频无缓存的查询结果，就是看不到了。



七、最终总结

上述系统到目前为止，已经演进到非常不错的状态了，因为这套架构已经解决了百亿流量高并发写入，海量数据存储，高性能计算，高并发查询，高可用保障，等一系列的技术挑战。线上生产系统运行非常稳定，足以应对各种生产级的问题。

其实再往后这套系统架构还可以继续演进，因为大型系统的架构演进，可以持续 N 多年，比如我们后面还有分布式系统全链路数据一致性保障、高稳定性工程质量保障，等等一系列的事情，不过文章就不再继续写下去了，因为文章承载内容量太少，很难写清楚所有的东西。

其实有不少同学跟我反馈说，感觉看不懂这个架构演进系列的文章，其实很正常，因为文章承载内容较少，这里有大量的细节性的技术方案和落地的实施，都没法写出来，只能写一下大型系统架构不断演进，解决各种线上技术挑战的一个过程。

我觉得对于一些年轻的同学，主要还是了解一下系统架构演进的过程，对于一些年长已经做架构设计的兄弟，应该可以启发一些思路，欢迎公众号后台给我留言，探讨这些技术问题。

如何在上万并发场景下设计可扩展架构（上）？

> 作者:中华石杉 原文地址

一、写在前面

之前更新过一个 **“亿级流量系统架构”** 系列，主要讲述了一个大规模商家数据平台的如下几个方面：

- 如何支撑百亿级数据的存储与计算
- 如何设计高容错分布式计算系统
- 如何设计承载百亿流量的高性能架构
- 如何设计每秒十万查询的高并发架构
- 如何设计全链路99.99%高可用架构

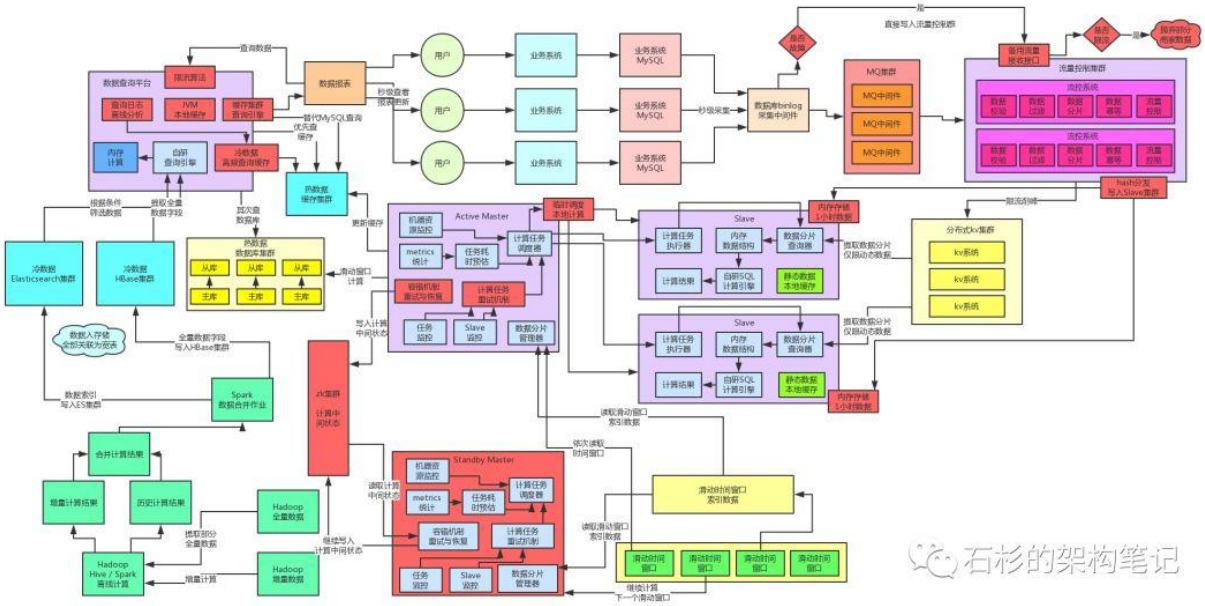
接下来，我们将会继续通过几篇文章，对这套系统的可扩展架构、数据一致性保障等方面进行探讨。

如果没看过本系列文章的同学可以先回过头看看之前写的几篇文章：

二、背景回顾

如果大家看过之前的一系列文章，应该依稀还记得上一篇文章最后，整个系统架构大致演进到了如下图的一个状态。

如果没看过之前的系列文章，上来猛一看下面这个图，绝对一脸懵逼，就看到一片 “花花绿绿”。这个也没办法，复杂的系统架构都是特别的庞杂的。



三、实时计算平台与数据查询平台之间的耦合

好，咱们正式开始！这篇文章咱们来聊聊这套系统里的不同子系统之间通信过程的一个可扩展性的架构处理。

这里面蕴含了线上复杂系统之间交互的真实场景和痛点，相信对大家能够有所启发。

我们就关注一下上面的架构图里左侧的部分，处于中间位置的那个实时计算平台在完成了每一个数据分片的计算过后，都会将计算结果写入到最左侧的数据查询平台中。

出于种种考量，因为计算结果的数据量相比于原始数据的数据量，实际上已经少了一个数量级了。

所以，我们选择的是实时计算平台直接将数据写入到数据查询平台的 MySQL 数据库集群中，然后数据查询平台基于 MySQL 数据库集群来对外提供查询请求。

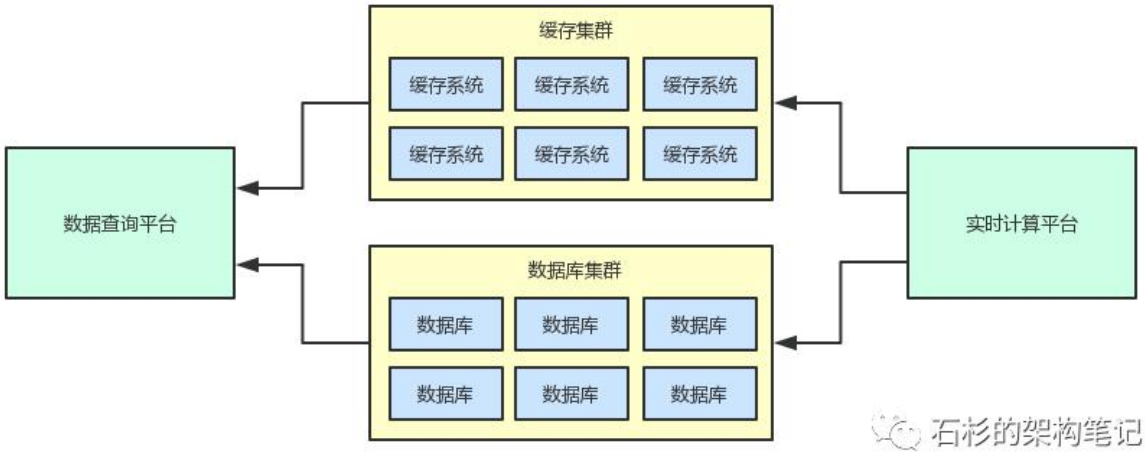
此外，为了保证当天的实时计算结果能够高并发的被用户查询，因此当时采取的是实时计算平台的计算结果同时双写缓存集群和数据库集群。

这样，数据查询平台可以优先走缓存集群，如果找不到缓存才会从数据库集群里回查数据。

所以上述就是实时计算平台与数据查询平台之间在某一个时期的一个典型的系统耦合架构。

两个不同的系统之间，通过同一套数据存储（数据库集群 + 缓存集群）进行了耦合。

大家看看下面的图，再来清晰的感受一下系统之间耦合的感觉。



系统耦合痛点 1：被动承担的高并发写入压力

大家如果仔细看过之前的系列文章，大概就该知道，在早期主要是集中精力对实时计算平台的架构做了大量的演进，以便于让他可以支撑超高并发写入、海量数据的超高性能计算，最后就可以抗住每秒数万甚至数十万的数据涌入的存储和计算。

但是因为早期采用了上图的这种最简单、最高效、最实用的耦合交互方式，实时计算平台直接把每个数据分片计算完的结果写入共享存储中，就导致了一个很大的问题。

实时计算平台能抗住超高并发写入没问题了，而且还能快速的高性能计算也没问题。

但是，他同时会随着数据量的增长，越来越高并发的将计算结果写入到一个数据库集群中。而这个数据库集群在团队划分的时候，实际上是交给数据查询平台团队来负责维护的。

也就是说，对实时计算平台团队来说，他们是不 care 那个数据库集群是什么状态的，而就是不停的把数据写入到那个集群里去。

但是，对于数据查询平台团队来说，他们就会被动的承担实时计算平台越来越高并发压力写入的数据。

这个时候数据查询平台团队的同学很可能处于这样的一种焦躁中：本来自己这块系统也有很多架构上的改进点要做，比如说之前提到的冷数据查询引擎的自研。

但是呢，他们却要不停的被线上数据库服务器的报警搞的焦头烂额，疲于奔命。

因为数据库服务器单机写入压力可能随着业务增长，迅速变成每秒 5000~6000 的写入压力，每天到了高峰期，线上服务器的 CPU、磁盘、IO、网络等压力巨大，报警频繁。

此时数据查询平台团队的架构演进节奏就会被打乱，因为必须被动的去根据实时计算平台的写入压力来进行调整，必须立马停下手中的工作，然后去考虑如何对数据库集群做分库分表的方案，如何对表进行扩容，如何对库进行扩容。

同时结合分库分表的方案，数据查询平台自身的查询机制又要跟着一起改变，大量的改造工作，调研工作，数据迁移工作，上线部署工作，代码改造工作。

实际上，上面说的这种情况，绝对是不合理的。

因为整个这套数据平台是一个大互联网公司里核心业务部门的一个核心系统，他是数十个 Java 工程师与大数据工程师通力合作一起开发，而且里面划分为了多个 team。

比如说数据接入系统是一个团队负责，实时计算平台是一个团队负责，数据查询平台是一个团队负责，离线数据仓库是一个团队负责，等等。

所以只要分工合作了以后，那么就不应该让一个团队被动的去承担另外一个团队猛然增长的写入压力，这样会打破每个团队自己的工作节奏。

导致这个问题的根本原因，就是因为两个系统间，没有做任何解耦的处理。

这就导致数据查询平台团队根本无法对实时计算平台涌入过来的数据做任何有效的控制和管理，这也导致了 “被动承担高并发写入压力” 问题的发生。

这种系统耦合导致的被动高并发写入压力还不只是上面那么简单，实际在上述场景中，线上生产环境还发生过各种奇葩的事情：

某一次线上突然产生大量的热数据，热数据计算结果涌入数据查询平台，因为没做任何管控，几乎一瞬间导致某台数据库服务器写入并发达到 1 万 +，DBA 焦急的担心数据库快宕机了，所有人也都被搞的焦头烂额，心理崩溃。

> ! 系统耦合痛点 2：数据库运维操作导致的线上系统性能剧烈抖动

在这种系统耦合的场景下，反过来实时计算平台团队的同学其实心里也会呐喊：我们心里也苦啊！

因为反过来大家可以思考一下，线上数据库中的表结构改变，那几乎可以说是再正常不过了，尤其是高速迭代发展中的业务。

需求评审会上，要是不小心碰上某个产品经理，今天改需求，明天改需求。工程师估计会怒火冲天的想要砍人。但是没办法，最后还是得为五斗米折腰，该改的需求还是得改。该改的表结构也还是要改，改加的索引也还是要加。

但是大家考虑一个点，如果说对上述这种强耦合的系统架构，单表基本都是在千万级别的数据量，同时还有单台数据库服务器每秒几千的写入压力。

在这种场景下，在线上走一个 MySQL 的 DDL 语句试一试？奉劝大家千万别胡乱尝试，因为数据查询团队里的年轻同学，干过这个事儿。

实际的结果就是，DDL 咔嚓一执行，对线上表结构进行修改，直接导致实时计算平台的写入数据库的性能急剧下降 10 倍以上。。。

然后连带导致实时计算平台的数据分片计算任务大量的延迟。再然后，因为实时计算之后的数据无法尽快反馈到存储中，无法被用户查询到，导致了大量的线上投诉。

并且，DDL 语句执行的还特别的慢，耗时数十分钟才执行完毕，这就导致数十分钟里，整套系统出现了大规模的计算延迟，数据延迟。

一直到数十分钟之后 DDL 语句执行完毕，实时计算平台才通过自身的自动延迟调度恢复机制慢慢恢复了正常的计算。

orz..... 于是从此之后，数据查询平台的攻城狮，必须得小心翼翼的在每天凌晨 2 点~ 3 点之间进行相关的数据库运维操作，避免影响线上系统的性能稳定性。

但是，难道人家年轻工程师没有女朋友？难道年长工程师没有老婆孩子？经常在凌晨 3 点看看窗外的风景，然后打个滴滴回家，估计没任何人愿意。

其实上述问题，说白了，还是因为两套系统直接通过存储耦合在了一起，导致了任何一个系统只要有点异动，直接就会影响另外一个系统。耦合！耦合！还是耦合！系统耦合痛点 N。。。

其实上面只不过是挑了其中两个系统耦合痛点来说明而已，文章篇幅有限，很难把上述长达数月的耦合状态下的各种痛点一一说明，实际线上生产环境的痛点还包括不限于：

- 实时计算平台自身写入机制有 bug 导致的数据丢失，结果让数据查询平台的同学去排查；
- 实时计算平台对缓存集群和数据库集群进行双写的时候，双写一致性的保证机制，居然还需要自己来实现，直接导致自己的代码里混合了大量不属于自己的业务逻辑；
- 数据查询平台有时候做了分库分表运维操作之后，比如扩容库和表，居然还得让实时计算平台的同学配合着一起修改代码配置，一起测试和部署上线
- 数据查询平台和实时计算平台两个 team 的同学在上述大量耦合场景下，经常天天一起加班到凌晨深夜，各自的女朋友都以为他们打算在一起了，但实际情况是一堆大老爷们天天被搞的焦头烂额，苦不堪言，都不愿意多看对方一眼
- 因为系统耦合导致的各种问题，两个 team 都要抽时间精力来解决，影响了自己那套系统的架构演进进度，没法集中人力和时间做真正有价值和意义的事情

四、下集预告

下一篇文章，我们就来聊一聊针对这些痛点，如何灵活的运用 MQ 消息中间件技术来进行复杂系统之间的解耦，同时解耦过后如何来自行对流量数据进行管控，解决各种系统耦合的问题。

敬请期待：

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）？

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）？

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）？

作者:中华石杉 原文地址

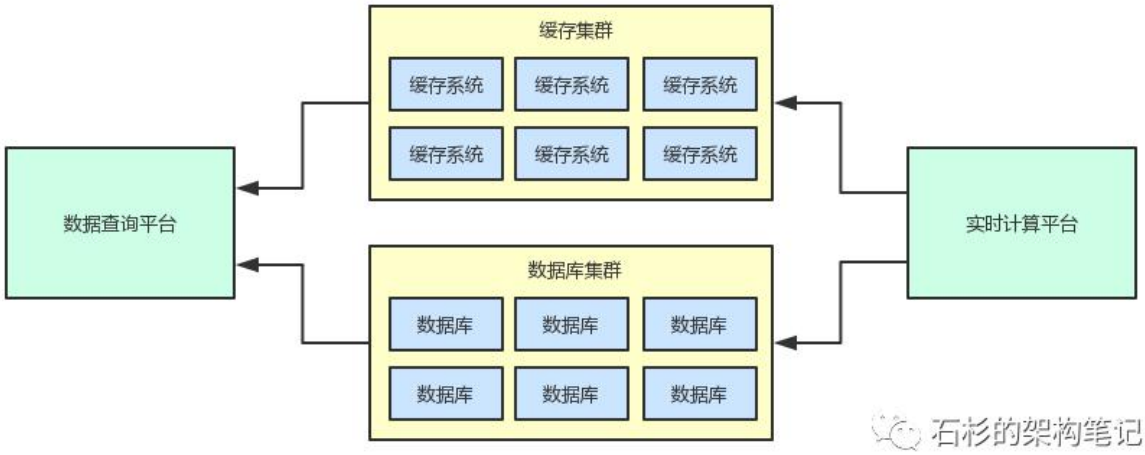
目录

一、前情提示

上一篇文章 亿级流量系统架构之如何在上万并发场景下设计可扩展架构（上），给大家初步讲述了一套大规模复杂系统中，两个核心子系统之间一旦耦合，会发生哪些令人崩溃的场景。如果还没看上篇文章的，建议先看一下。

这篇文章，咱们就给大家来说一说通过 MQ 消息中间件的使用，如何重构系统之间的耦合，让系统具备高度的可扩展性。

首先来回看一下之前画的一张两个系统之间进行耦合的一个大图，从这个图里我们可以看到两个系统完全通过一套共享存储（数据库集群 + 缓存集群）进行了耦合。



二、清晰的划分系统边界

只要有耦合，一旦要解决耦合，那么第一个要干的事儿就是先划分清楚系统之间的边界。

比如上面那两套系统都共享了一套存储集群，那么大家可以先思考一下，两个系统之间的边界应该如何划分？也就是说，中间那套缓存集群和数据库集群，到底应该是属于哪个系统？

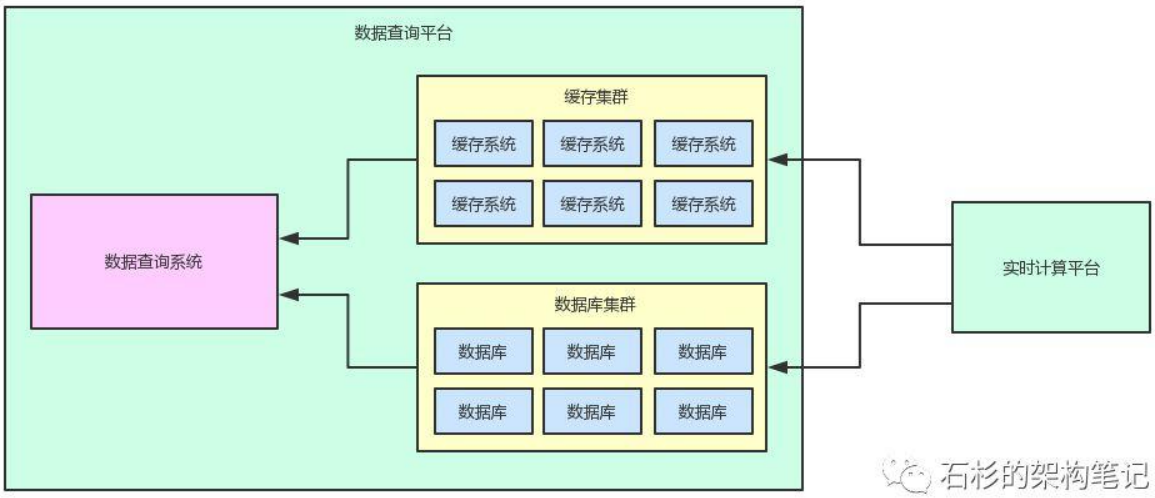
首先我们看一下，缓存集群和数据库集群主要是给谁用的？

很明显就是给数据查询平台用的，说白了，那两套集群都是数据查询平台赖以生存的核心底层数据存储，这里存储的数据也都是属于数据查询平台的核心数据。

对于实时计算平台来说，他只不过是将自己计算后的结果写入到缓存集群和数据库集群罢了。

实时计算平台只要写入过后，后续就不会再管那些数据了，所以这两套集群明显是不属于实时计算平台的。

好，那么系统之间的边界就很清晰的划分清楚了，大家看一下如下的图。首先从系统整体架构的架构而言，两套系统之间的关系应该是下面这样子的。



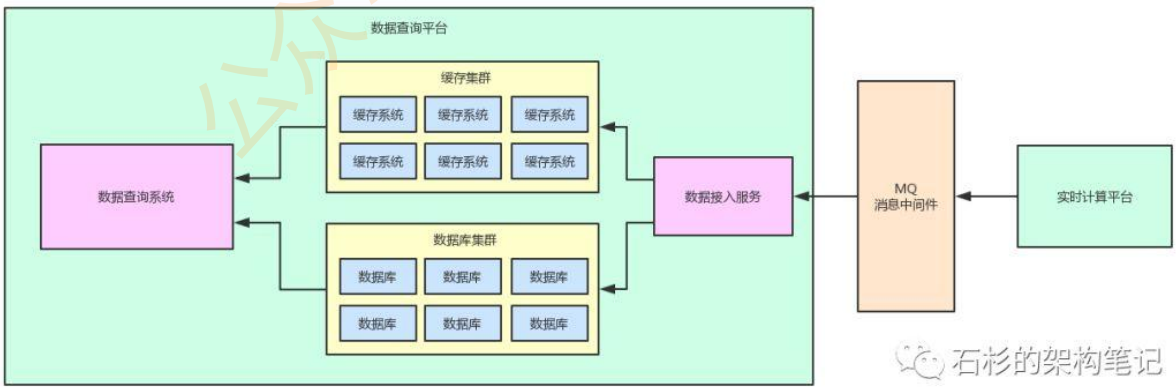
三、引入消息中间件解耦

只要划分清楚了系统之间的边界，接着下一步，就是引入消息中间件来进行解耦了。

如果大家对消息中间件的使用场景还不太熟悉的，可以参考之前的一篇文章：哥们，你们的系统架构中为什么要引入消息中间件？这篇文章里面，对消息中间件的各种使用场景都有说明。

我们只要引入一个消息中间件，然后让实时计算平台将计算好的数据按照预设的格式直接写入到消息中间件即可。

同时，数据查询平台需要增加一个数据接入服务，这个数据接入服务就是负责将消息中间件里的数据消费出来，然后落地写入到本地的缓存集群和数据库集群。



如上图所示，此时两个系统之间已经不再直接基于共享数据存储进行耦合了，中间加入了 MQ 消息中间件。

这个消息中间件仅仅就是用于两个系统之间的数据交互和传输，职责简单，清晰明了。

这样做最大的好处，就是数据查询平台自身可以对涌入自身平台的数据按照自己的需求进行定制化的管控了，不会像之前那样的被动。

实际上在上述架构之下，涌入数据查询平台的所有数据，都需要经过数据接入服务那一关。在数据接入服务那里就可以随意根据自己的情况进行管理。

四、利用消息中间件削峰填谷

还记得上一篇文章我们提到，这两个系统之间第一个大痛点，就是实时计算平台会高并发写入数据查询平台，之前不做任何管控的时候，导致各种意外发生。

举个例子，比如快速增长的写库压力导致数据查询平台必须优先 cover 住分库分表那块的架构，打破自己的架构演进节奏；

比如突然意外出现的热数据因为不做任何写入管控，一下子差点把数据库服务器击垮。

因此一旦用消息中间件在中间挡了一层之后，我们就可以进行削峰填谷了。

那什么叫做削峰填谷呢？其实很简单，我们先来看看，如果不做任何管控，实时计算平台写入数据库集群的写并发曲线图，大概如下面所示。

在高峰期，写入会有一个陡然上升的尖峰。

就好比说，平时每秒写入并发就 500，但是高峰期写入并发请求有 5000，那么大家就会看到上面的那张图，在高峰期突然冒出来一个尖峰，一下子涌入并发 5000 请求，此时数据查询平台的数据库集群可能就会受不了。

但是，如果我们在数据接入服务里做一个限流控制呢？

也就是说，在数据接入服务里，根据当前数据查询平台的数据库集群能承载的并发上限，比如说就是最多承载每秒 3000。

好！那么数据接入服务自己就控制好，每秒最多就往自己本地的数据库集群里写入最多每秒 3000 的请求压力。

此时就会出现削峰填谷的效果，大家看下面的图。

因为在高峰期瞬时写入压力最大有 5000/s，但是数据接入服务做了流量控制，最多就往本地数据库集群写入 3000/s，那么每秒就会有 2000 条数据在消息中间件里做一个积压。

但是积压一会儿不要紧，最起码保证说在高峰期，这个向上的尖峰被削平了，这就是所谓的削峰。

然后在高峰期过了之后，本来每秒可能就 100/s 的写入压力，但是此时数据接入服务会持续不断的从消息中间件里取出来数据然后持续以最大 3000/s 的写入压力往本地数据库集群里写入。

那么在低峰期，大家看到还会持续一段时间是 3000/s 的写入速度往本地数据库里写。

原来的图里在低峰期是谷底，现在谷底被填平了，这就是所谓的填谷。

通过这套削峰填谷的机制，就可以保证数据查询平台完全能够以自己接受的了的速率，均匀的把 MQ 里的数据拿出来写入自己本地数据库集群中。

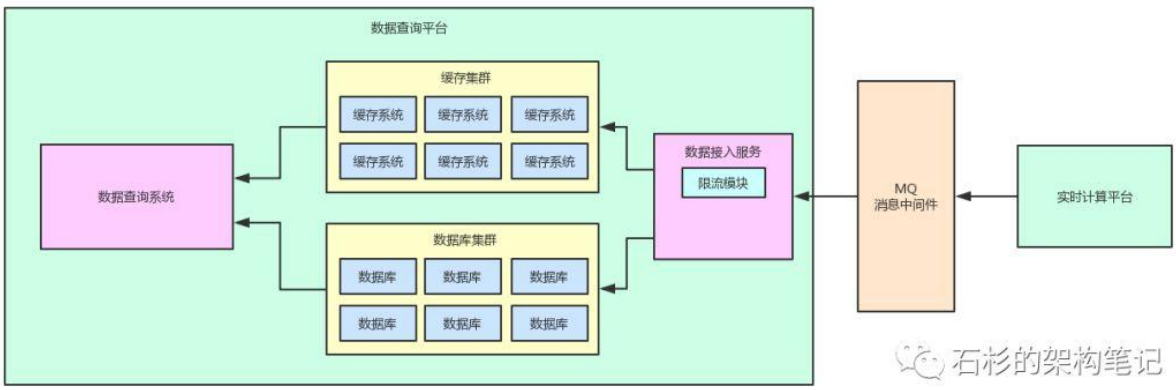
这样子无论实时计算平台多高的并发请求压力过来，哪怕是那种异常的热数据，瞬间上万并发请求过来也无所谓了。

因为 MQ 中间件可以抗住瞬间高并发写入，但是数据查询平台永远都是稳定匀速的写入自己本地数据库。

这样的话，数据查询平台就不需要去过多的 care 实时计算平台带给自己的压力了，可以按照自己的节奏规划好整体架构的演进策略，按照自己的脚本去迭代架构。

说了那么多，老规矩！给大家来一张图，此时的架构图如下所示。

大伙儿可以直观的感受一下，在数据接入服务中多了一个限流的模块。



五、手动流量开关配合数据库运维操作

现在基于消息中间件将两个系统隔离开来之后，另外一个大的好处就是：数据查询平台做任何数据运维的操作，比如说 DDL、分库分表扩容、数据迁移，等等诸如此类的操作，已经跟实时计算平台彻底无关了。

实时计算平台主要就是简单的往消息中间件写入，其他的就不用管了。

然后如果数据查询平台要做一些数据库运维的操作，此时就可以通过在数据接入服务中加入一个手动流量开关，临时将流量开关关闭一会儿。

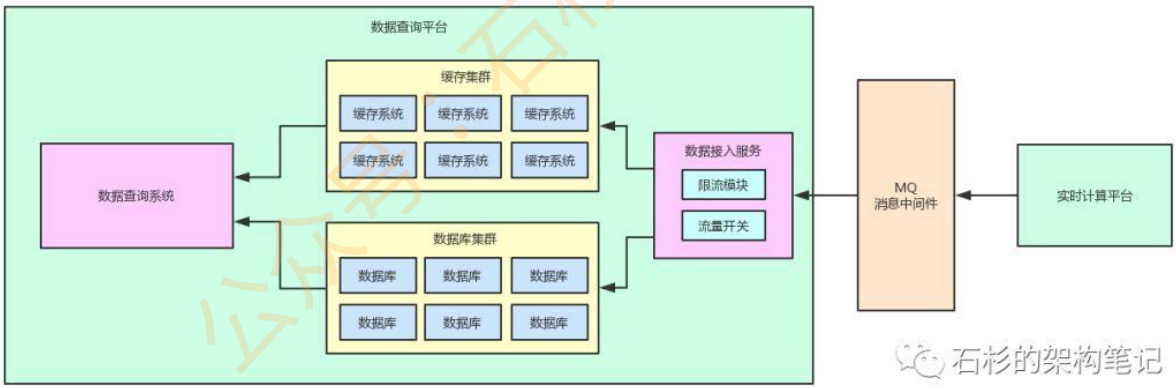
比如选择一个下午大家都在工作或者午睡的时候，相对低峰的时期，半小时内关闭流量开关。

然后此时数据接入服务就不会继续往本地数据库写入数据了，此时写入操作就会停止，然后就在半小时内迅速完成数据库运维操作。

等相关操作完成之后，再次打开流量开关，继续从 MQ 里消费数据再快速写入到本地数据库内即可。

这样，就可以完全避免了同时写入数据，还同时进行数据库运维操作的窘境。否则在早期耦合的状态下，每次进行数据库运维操作，还得实时计算平台团队的同学配合一起进行各种复杂操作，才能避免线上出现故障，现在完全不需要人家的参与了，自己团队就可以搞定。

整个过程，我们还是用一张图，给大家呈现一下：



六、支持多系统同时订阅数据

引入消息中间件之后，还有另外一个好处，就是其他的一些系统也可以按照自己的需要去 MQ 里订阅实时计算平台计算好的数据。

举个例子，在这套平台里，还有数据质量监控系统，需要获取计算数据进行数据结果准确性和质量的监控。

另外，还有数据链路监控系统，同样需要将 MQ 里的数据作为数据计算链路中的一个核心点数据采集过来，进行数据全链路的监控和自动追踪。

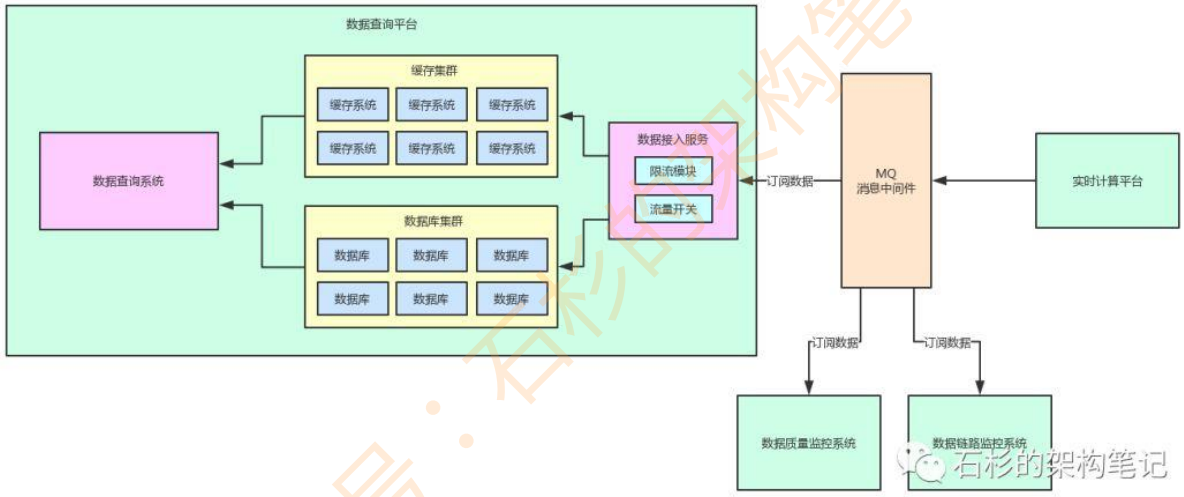
如果没有引入 MQ 消息中间件概念的话，那么是不是就会导致实时计算平台除了将数据写入一份到数据库集群，还需要通过接口发送给数据质量监控系统？还需要发送给数据链路监控系统？这样简直是坑爹到不行，N 个系统全部耦合在一起。

之前的文章 **《哥们，你们的系统架构中为什么要引入消息中间件？》** 就阐述了这种多系统订阅同一份数据，但是通过接口调用耦合在一起的窘境。

这样每次要是有一点变动，各个系统的负责人都在一起开会商讨，修改代码，修改接口，考虑各种调用细节，等等。

但是现在有了消息中间件，完全可以通过 MQ 支持的 “Pub/Sub” 消息订阅模型，不同的系统都可以来订阅同一份数据，大家自己按需消费，按需处理，各个系统之间完全解耦。

整个系统的可扩展性瞬间提升了很多，因为各个系统各自迭代和演进架构，都不需要强依赖其他的系统了。



七、系统解耦后的感受

云开雾散！各个团队的同学终于不用天天扯皮，今天说你的系统影响了我，明天是我的系统影响了你。

同时也压根儿不用去关注其他的系统，只要有一个总架构师把控好整体架构，各个 team 都按照这个分工协作来做即可。

消息中间件的引入，消除了系统的耦合性，大幅度提升了系统的可扩展性，各个 team 都可以快速的独立的迭代扩展自己的架构和系统。

PS：最重要的，不同 team 的同学，再也不用为了一些鸡毛蒜皮的事儿加班到深更半夜，导致他们的女朋友觉得他们要在一起了。。。

八、下集预告

下一篇文章，是关于可扩展架构的最后一篇。我们把整体架构梳理完毕了之后，就可以来看一看具体到 MQ 消息中间件的层面，他是怎么通过 “Pub/Sub” 的订阅模型，让一份数据发布出去，然后让多个不同的系统来订阅同一份数据的。

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）？

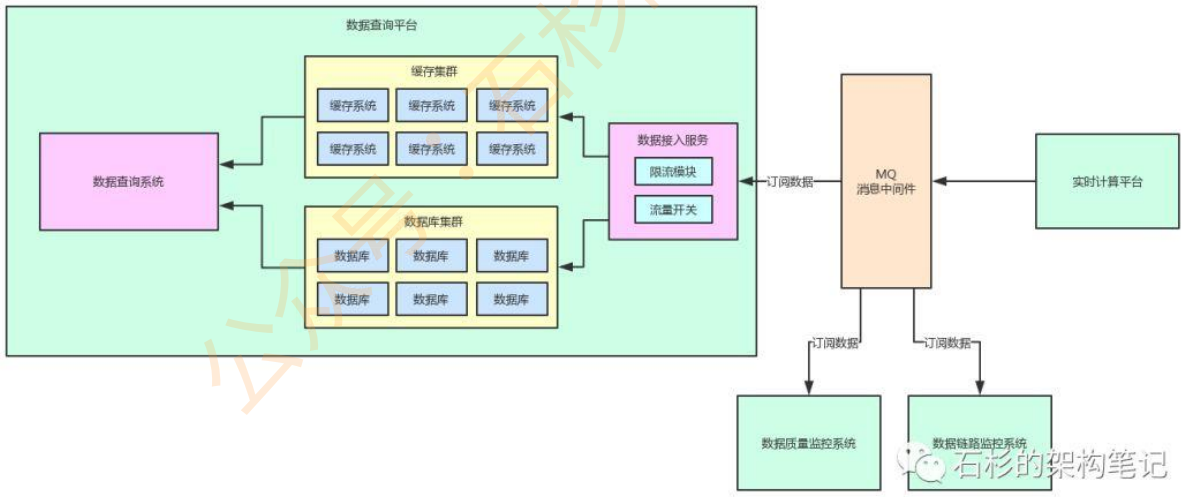
> **作者:中华石杉** 原文地址

一、前情提示

上一篇文章《亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）？》分析了一下如何利用消息中间件对系统进行解耦处理。

同时，我们也提到了使用消息中间件还有利于一份数据被多个系统同时订阅，供多个系统来使用于不同的目的。

目前的一个架构如下图所示。



在这个图里，我们可以清晰的看到，实时计算平台发布的一份数据到消息中间件里，接着，会进行如下步骤：

- 数据查询平台，会订阅这份数据，并落入自己本地的数据库集群和缓存集群里，接着对外提供数据查询的服务
- 数据质量监控系统，会对计算结果按照一定的业务规则进行监控，如果发现有数据计算错误，则会立马进行报警

- 数据链路追踪系统，会采集计算结果作为一个链路节点，同时对一条数据的整个完整计算链路都进行采集并组装出来一系列的数据计算链路落地存储，最后如果某个数据计算错误了，就可以立马通过计算链路进行回溯排查问题

因此上述场景中，使用消息中间件一来可以解耦，二来还可以实现消息 “Pub/Sub” 模型，实现消息的发布与订阅。

这篇文章，咱们就来看看，假如说基于 RabbitMQ 作为消息中间件，如何实现一份数据被多个系统同时订阅的 “Pub/Sub” 模型。

二、基于消息中间件的队列消费模型

上面那个图，其实就是采用的 RabbitMQ 最基本的队列消费模型的支持。

也就是说，你可以理解为 RabbitMQ 内部有一个队列，生产者不断的发送数据到队列里，消息按照先后顺序进入队列中排队。

接着，假设队列里有 4 条数据，然后我们有 2 个消费者一起消费这个队列的数据。

此时每个消费者会均匀的被分配到 2 条数据，也就是说 4 条数据会均匀的分配给各个消费者，每个消费者只不过是处理一部分数据罢了，这个就是典型的队列消费模型。

如果有同学对这块基于 RabbitMQ 如何实现有点不太清楚的话，可以参考之前的一些文章：

- **《哥们，消息中间件在你们项目里是如何落地的？》**
- **《扎心！线上服务宕机时，如何保证数据 100% 不丢失？》**
- **《消息中间件集群崩溃，如何保证百万生产数据不丢失？》**

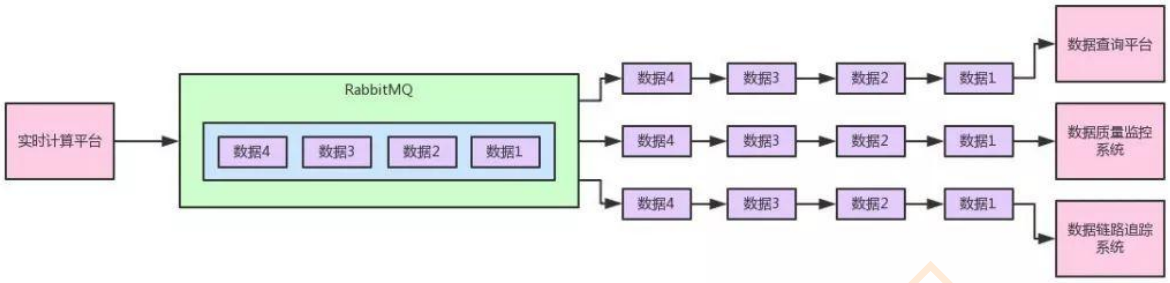
之前这几篇文章，基本给出了上述那个最基本的队列消费模型的 RabbitMQ 代码实现，以及如何保证消费者宕机时数据不丢失，如何让 RabbitMQ 集群对 queue 和 message 都进行持久化。基本上整体代码实现都比较完整，大家可以去参考一下。

三、基于消息中间件的 “Pub/Sub” 模型

但是消息中间件还可以实现一种 “Pub/Sub”模型，也就是 “发布 / 订阅” 模型，Pub 就是 Publish，Sub 就是 Subscribe。

这种模型是可以支持多个系统同时消费一份数据的。也就是说，你发布出去的每条数据，都会广播给每个系统。

给大家来一张图，一起来感受一下。



如上图所示。也就是说，我们想要实现的上图的效果，实时计算平台发布一系列的数据到消息中间件里。

然后数据查询平台、数据质量监控系统、数据链路追踪系统，都会订阅数据，都会消费到同一份完整的数据，每个系统都可以根据自己的需要使用数据。

这，就是所谓的 “Pub/Sub” 模型，一个系统发布一份数据出去，多个系统订阅和消费到一模一样的一份数据。

那如果要实现上述的效果，基于 RabbitMQ 应该怎么来处理呢?

四、RabbitMQ 中的 exchange 到底是个什么东西?

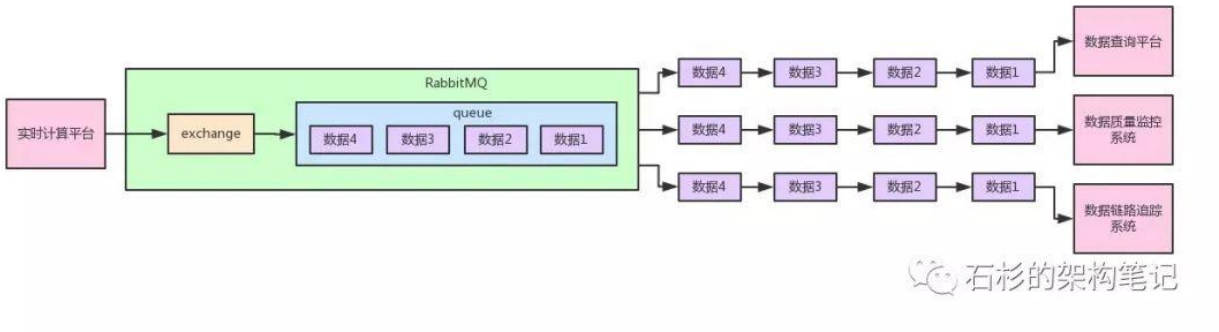
实际上来说，在 RabbitMQ 里面是不允许生产者直接投递消息到某个 queue（队列）里的，而是只能让生产者投递消息给 RabbitMQ 内部的一个特殊组件，叫做 “exchange”。

关于这个 exchange，大概你可以把这个组件理解为一种消息路由的组件。

也就是说，实时计算平台发送出去的 message 到 RabbitMQ 中都是由一个 exchange 来接收的。

然后这个 exchange 会根据一定的规则决定要将这个 message 路由转发到哪个 queue 里去，这个实际上就是 RabbitMQ 中的一个核心的消息模型。

大家看下面的图，一起来理解一下。



五、默认的 exchange

在之前的文章里，我们投递消息到 RabbitMQ 的时候，也没有用什么 exchange，但是为什么就还是把消息投递到了 queue 里去呢?

那是因为我们用了默认的 exchange，他会直接把消息路由到你指定的那个 queue 里去，所以如果简单用队列消费模型，不就省去了 exchange 的概念了吗。

```
channel.basicPublish(
    "",
    "warehouse_schedule_delivery",
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes());
```


上面这段就是之前我们给大家展示的，让消息持久化的一种投递消息的方式。

大家注意里面的第一个参数，是一个空的字符串，这个空字符串的意思，就是说投递消息到默认的 exchange 里去，然后他就会路由消息到我们指定的 queue 里去。

六、将消息投递到 fanout exchange

在 RabbitMQ 里，exchange 这种组件有很多种类型，比如说：direct、topic、headers 以及 fanout。这里咱们就来看看最后一种，fanout 这种类型的 exchange 组件。

这种 exchange 组件其实非常的简单，你可以创建一个 fanout 类型的 exchange，然后给这个 exchange 绑定多个 queue。

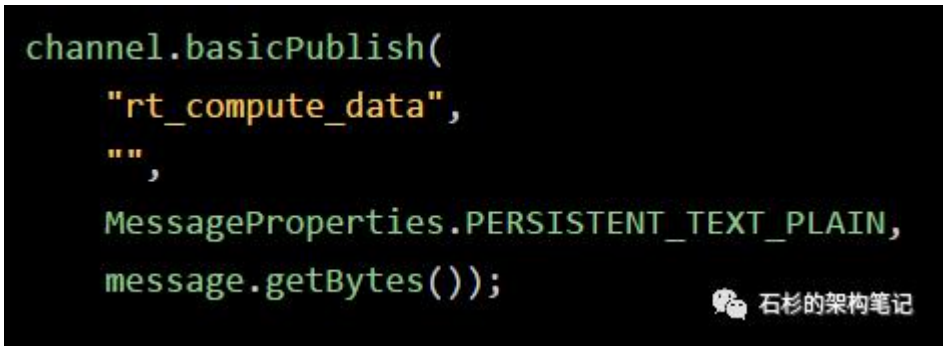
接着只要你投递一条消息到这个 exchange，他就会把消息路由给他绑定的所有 queue。

使用下面的代码就可以创建一个 exchange，比如说在实时计算平台（生产者）的代码里，可以加入下面的一段，创建一个 fanout 类型的 exchange。第一个参数我们叫做“rt_compute_data”，这个就是 exchange 的名字，rt 就是 “RealTime” 的缩写，意思就是实时计算系统的计算结果数据。

第二个参数就是定义了这个 exchange 的类型是 “fanout”。

```
channel.exchangeDeclare("rt_compute_data",  "fanout");
```

接着我们就采用下面的代码来投递消息到我们创建好的 exchange 组件里去：



大家会注意到，此时消息就是投递到指定的 exchange 里去了，但是路由到哪个 queue 里去呢？此时我们暂时还没确定，要让消费者自己来把自己的 queue 绑定到这个 exchange 上去才可以。

七、绑定自己的队列到 exchange 上去消费 我们对消费者的代码也进行修改，之前我们在这里关闭了 autoAck 机制，然后每次都是自己手动 ack。

```
// 在消费者里也声明下exchange
channel.exchangeDeclare("rt_compute_data", "fanout");

// 这里是指创建一个持久化的队列
// 名字的意思，就是这个队列是查询平台作为消费者来消费的
channel.queueDeclare("rt_compute_data_query", true, false, false, null);

// 这里就是把这个消费者自己的队列绑定到了exchange上去
channel.queueBind("rt_compute_data_query", "rt_compute_data", "");

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    try {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] 接收到消息，准备进行处理 '" + message + "'");
    } finally {
        System.out.println(" [x] 完成消息处理");

        // 这就是核心代码，手动在代码里对RabbitMQ进行ack
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
      }
};

channel.basicConsume(QUEUE_NAME, false, deliverCallback, consumerTag -> { });
```


上面的代码里，每个消费者系统，都会有一些不一样，就是每个消费者都需要定义自己的队列，然后绑定到 exchange 上去。比如：

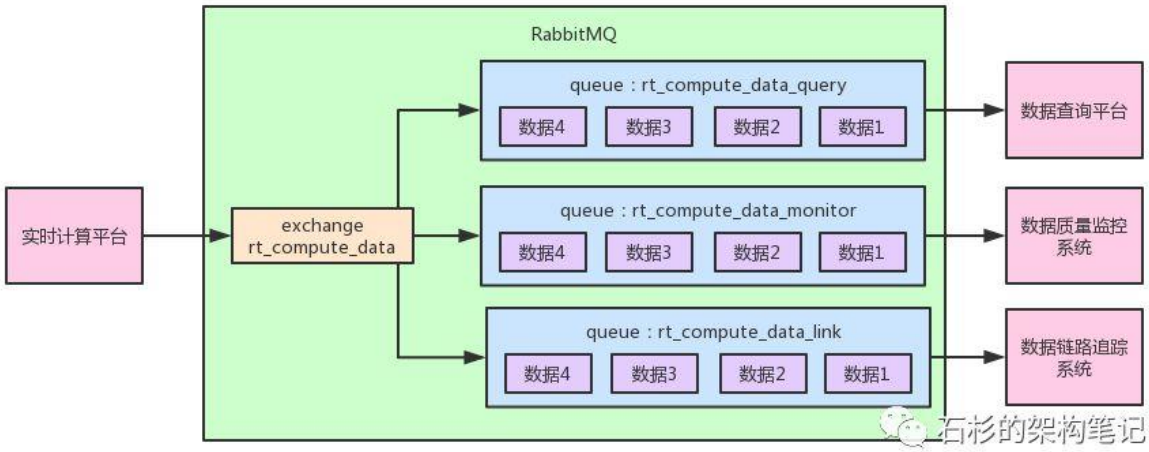
- 数据查询平台的队列是 “rt_compute_data_query”
- 数据质量监控平台的队列是 “rt_compute_data_monitor”
- 数据链路追踪系统的队列是 “rt_compute_data_link”

这样，每个订阅这份数据的系统其实都有一个属于自己的队列，然后队列里被会被 exchange 路由进去实时计算平台生产的所有数据。

而且因为是多个队列的模式，每个系统都可以部署消费者集群来进行数据的消费和处理，非常的方便。

八、整体架构图

最后，给大家来一张大图，我们再跟着图，来捋一捋整个流程。



如上图所示，首先，实时计算平台会投递消息到 “rt_compute_data” 这个 “exchange” 里去，但是他没指定这个 exchange 要路由消息到哪个队列，因为这个他本身是不知道的。

接着数据查询平台、数据质量监控系统、数据链路追踪系统，就可以声明自己的队列，都绑定到 exchange 上去。

因为 queue 和 exchange 的绑定，在这里是要由订阅数据的平台自己指定的。而且因为这个 exchange 是 fanout 类型的，他只要接收到了数据，就会路由数据到所有绑定到他的队列里去，这样每个队列里都有同样的一份数据，供对应的平台来消费。

而且针对每个平台自己的队列，自己还可以部署消费服务集群来消费自己的一个队列，自己的队列里的数据还是会均匀分发给各个消费服务实例来处理，每个消费服务实例会获取到一部分的数据。

大家思考一下，这样是不是就实现了不同的系统订阅一份数据的 “Pub/Sub” 的模型?

当然，其实 RabbitMQ 还支持各种不同类型的 exchange，可以实现各种复杂的功能。

后续我们将会给大家通过实际的线上系统架构案例，来阐述消息中间件技术的各种用法。

亿级流量架构第二弹：你的系统真的无懈可击吗?

> **作者:中华石杉** 原文地址

> ! 在《亿级流量系统架构》系列第一阶段中，我们从零开始，讲述了一个大型数据平台的几个方面的构建，包括：

- 如何承载百亿级数据的存储挑战

- 如何承载设计高容错的分布式架构
- 如何设计高性能架构，使之能承载百亿级流量
- 如何设计高并发架构，能够支撑住每秒数十万的并发查询
- 如何设计全链路 99.99% 的高可用架构

好！架构演进到这个时候，系统是否无懈可击了呢?

当然不是!

自古以来，能够瓦解一个军队战斗力的，不仅有外力冲击，还有内部因素。

同样，对于咱们的亿级流量系统，外部的冲击我们抗住了，现在的考验，来自于系统自身。而首当其冲的，就是系统的可扩展性带来的严重挑战。。。

因此在第二阶段，咱们用了大量的篇幅，分为上中下三篇，详细的讨论了该架构在可扩展性方面的痛点和改进。

跨过了 2018 年，你是否还记得这些痛点以及针对的技术方案呢?

如果忘了，没关系，跟着本文，温故知新。笔者希望各位在重拾记忆的同时，能有新的收获，并且能把文中的某些技术方案在自己公司中实际落地实践。

同样，对于可扩展性方案的复习，也是为后面系统在其他方面的改进打下基础，这样大伙儿读后面的文章时，不至于因为中间知识的断层而一脸懵逼。。。

对亿级流量架构可扩展性的讨论，咱们分成了上中下三篇。其中上篇，开门见山，发现问题：实时计算平台与数据查询平台之间耦合严重，并造成了诸多痛点：

- 数据查询团队被动承担了本不该他们承担的高并发写入压力
- 数据库运维操作导致的线上系统性能剧烈抖动
- 实时计算平台团队因为自身写入机制的 bug 导致数据丢失，结果让数据查询团队来进行排查，典型的甩锅!
- 实时计算平台团队，竟然需要自己来实现双写一致性的保障机制，直接导致代码里混合了大量不属于自己团队业务逻辑的代码
- 数据查询平台做了分库分表的操作，需要实时计算平台 team 一起修改配置，一起测试部署上线

总之，这些痛点，导致的结果是两个团队的同学天天腻在一起，而且是被迫的。。。

对于上面这些系统痛点的成因，你还有印象吗？如果忘了，猛戳下面链接，先赶紧去复习一波吧，知道了病症，才好对症下药！

> **猛戳下方链接：**

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（上）？

好，通过了上篇文章，我们已经知道了系统耦合造成的各种痛点，真的很痛！

那么现在，就该针对这些痛点，对症下药。看看下面的内容，你还能记起吗？

- 类似于中医的“望闻问切”，解决问题的第一步，就是找到病因。而到咱们这里，解决耦合的第一步，则是清晰的划分出系统边界。
- 划分出边界之后，第二件事，当然就是解耦。如何解耦：利用消息中间件
- 好！现在我们引入了消息中间件解耦，你是否还记得上篇文章中的一个痛点：实时计算平台高并发写入时，数据查询平台要无辜承受高并发的写入压力
- 那我们引入了中间件之后，通过消息中间件进行削峰填谷，就能解决这个问题了，关于什么是削峰填谷，以及如何实行，还记得吗？
- 解耦过后，我们通过手动流量开关来配合数据库运维，直接自己团队的同学在某个低锋时段关闭流量开关，迅速完成数据库运维操作。这不又解决了一大痛点吗！
- 好处还不止这些，比如，我们引入中间件解耦之后，其他系统不也可以按需去 MQ 里，订阅实时计算平台计算好的数据吗！再不用看其他平台的脸色了

总体来讲，解耦之后，各个团队各司其职，不用天天被迫腻在一起。而没有了人为的各种干预，系统也运转的更加流畅高效。

关于这些针对性的解决方案，笔者建议大家再仔细看看，这都是真实线上生产总结出的经验，也许里面的某些方案能够帮到你！

> **猛戳下方链接：**

亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）？

讲完了实际的落地方案，我们来到了亿级流量架构可扩展性的下篇。

在可扩展性中篇的讨论中，我们提到了解耦的好处之一，是可以实现消息的“Pub/Sub”模型，即不同平台都可以根据自身需要去订阅同一份数据。

那么下篇，我们讨论的主题就是基于消息中间件的“Pub/Sub”模型，并以 RabbitMQ 为例，详细阐述了其在代码层面的落地实践。

什么是 exchange？默认的 exchange 是啥？如何绑定自己的队列到 exchange 上去消费？这些还记得吗？如果忘了，猛戳下面的链接，赶紧的回顾一下！

> **猛戳下方链接：**

[亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）？]

亿级流量系统架构之如何保证百亿流量下的数据一致性（上）

> **作者:中华石杉** [原文地址]

目录

一、前情提示

这篇文章，咱们继续来聊聊之前的亿级流量架构的演进，之前对这个系列的文章已经更新到了可扩展架构的设计，如果有不太清楚的同学，建议一定先回看一下之前的文章：

-[亿级流量系统架构之如何支撑百亿级数据的存储与计算]

-[亿级流量系统架构之如何设计高容错分布式计算系统]

-[亿级流量系统架构之如何设计承载百亿流量的高性能架构]

-[亿级流量系统架构之如何设计每秒十万查询的高并发架构]

-[亿级流量系统架构之如何设计全链路 99.99% 高可用架构]

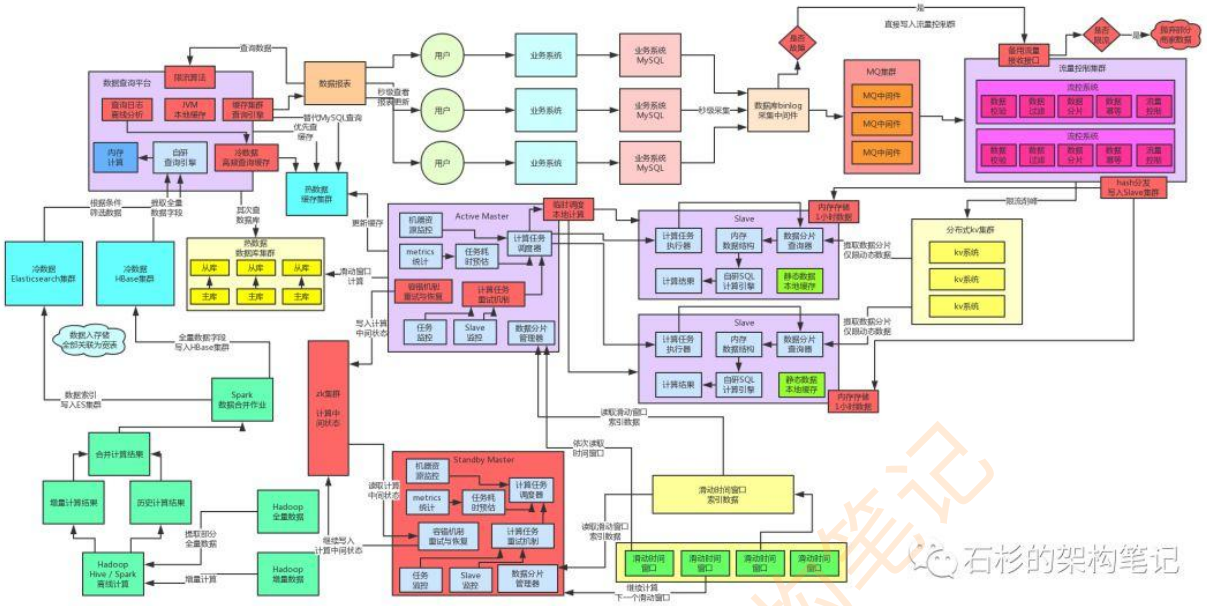
-[亿级流量系统架构之如何在上万并发场景下设计可扩展架构（上）]

-[亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）]

-[亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）]

老规矩！我们首先看一下这个复杂的系统架构演进到当前阶段，整体的架构图是什么样子的。

笔者再次友情提醒，如果各位小伙伴对下面这个复杂的架构图还有什么不理解的地方，一定要先回看之前的文章，因为系列文必须对上下文有清晰的理解和认识。



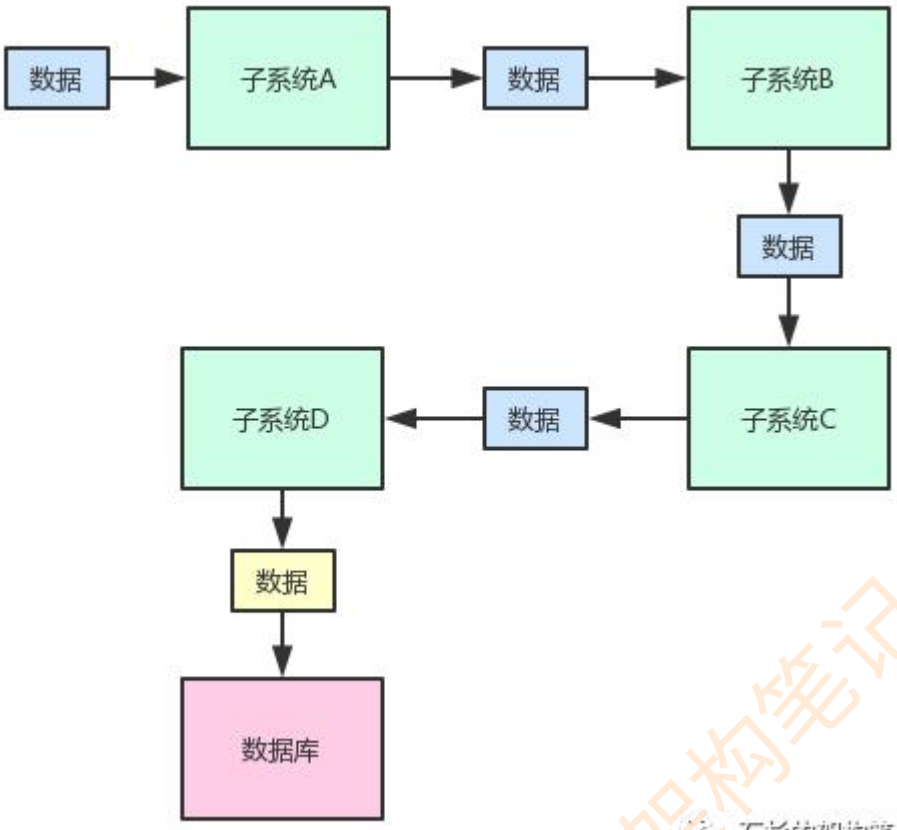
接着文本我们来聊聊一个核心系统每天承载百亿流量的背景下，应该如何来保证复杂系统中的数据一致性？

二、什么是数据一致性？

简单来说，在一个复杂的系统中一定会对一些数据做出非常复杂的处理，而且可能是多个不同的子系统，甚至是多个服务。

对一个数据按照一定的顺序依次做出复杂的业务逻辑的执行，最终可能就会生产出一份宝贵的系统核心数据，落地到存储里去，比如说在数据库里存储。

给大家来一张手绘彩图，感受下这个现场的氛围：



从上图中我们就可以看到，多个系统如何对一个数据依次进行处理，最终拿到一份核心数据，并落地到存储里去。

那么在这个过程中，就可能会产生所谓的数据不一致的问题。

什么意思呢？给大家举一个最简单的例子，我们本来期望数据的变化过程是：数据 1 -> 数据 2 -> 数据 3 -> 数据 4。

那么最后落地到数据库里的应该是数据 4，对不对？

结果呢？不知道为啥，经过上面那个复杂的分布式系统中的各个子系统，或者是各个服务的协作处理，最后居然搞出来一个数据 87。

搞了半天，搞了一个跟数据 4 风马牛不相及的一个东西，最后落地到了数据库里。

然后啊，这套系统的最终用户，可能通过前台的界面看到了一个莫名其妙的数据 87。

这就尴尬了，用户明显会觉得这个数据有错误，就会反馈给公司的客服，此时就会上报 bug 到工程师团队，大家就开始吭哧吭哧的找问题。

上面说的这个场景，其实就是一种数据不一致的问题，也是我们接下来几篇文章要讨论的一个问题。

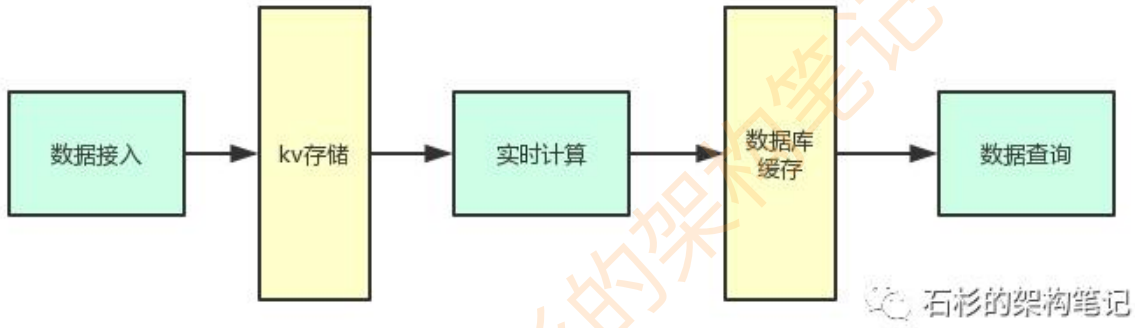
实际上，在任何一个大规模分布式系统里，都会存在类似的问题。无论是电商，O2O，还是本文举例的数据平台系统，都一样。

三、一个数据计算链路的梳理

那么既然已经明确了问题，接下来就来看看在数据平台这个系统里，到底是什么问题可能会导致一个最终落地存储的数据的异常呢?

要明白这个问题，咱们先回过头看看，在之前提过的数据平台这个项目里，一个最终落地的数据的计算链路是什么样的?

大家看看下面的图：



如上图所示，其实从最简单的一个角度来说，这个数据计算的链路大概也就是上面的那个样子。

- 首先，通过 MySQL binlog 采集中间件获取到数据，转发给数据接入层。
- 然后，数据接入层会把原始数据落地到 kv 存储里去
- 接着，是实时计算平台会从 kv 存储里提取数据进行计算
- 最后，会将计算结果写入到数据库 + 缓存的集群里。数据查询平台会从数据库 + 缓存的集群里提取数据，提供用户来进行查询

看起来很简单，对吧?

但是哪怕是这个系统里，数据计算链路，也绝对不是这么简单的。

如果大家看过之前的系列文章的话，就应该知道，这个系统为了支撑高并发、高可用、高性能等场景，引入了大量的复杂机制。

所以实际上一条原始数据进入到系统，一直到最后落地到存储里，计算链路还会包含下面的东西：

- 接入层的限流处理
- 实时计算层的失败重试
- 实时计算层的本地内存存储的降级机制
- 数据分片的聚合与计算，单条数据在这里可能会进入一个数据分片里
- 数据查询层的多级缓存机制

上面只不过是随便列举了几条。然而哪怕只是上述几条，都可以把一个数据的计算链路变得复杂很多倍了。

四、数据计算链路的 bug

既然大家已经明白了，在一个复杂系统里，一份核心数据可能是经过一个极为复杂的计算链路的处理，中间百转千回，任何可能的情况都会发生。

那么就可以理解在大型分布式系统中，数据不一致的问题是如何产生的了。

其实原因非常的简单，说白了，就是数据计算链路的 bug。

也就是说，在数据的计算过程中，某个子系统出现了 bug，并没有按照我们预期的行为去处理，导致最终产出去的数据变得错误了。

那么，为什么会在数据计算链路中出现这种 bug 呢?

原因很简单，如果大家曾经参与过上百人协作的大型分布式系统，或者是主导过上百人协作开发的大型分布式系统的架构设计，应该对核心数据的异常和错误非常熟悉，并且会感到头疼不已。

大规模分布式系统中，动辄上百人协作开发。很可能某个子系统或者是某个服务的负责人，对数据的处理逻辑理解偏差了，代码里写了一个隐藏的 bug。

而这个 bug，轻易不会触发，并且在 QA 测试环境还没测出来，结果带着一颗定时炸弹，系统上线。

最后在线上某种特殊的场景下，触发了这个 bug，导致最终的数据出现问题。

五、电商库存数据的不一致问题

接触过电商的同学，可能此时脑子里就可以快速的想到一个类似的经典场景：电商中的库存。

在大规模的电商系统中，库存数据绝对是核心中的核心。但是实际上，在一个分布式系统中，很多系统可能都会采用一定的逻辑来更新库存。

这就可能导致跟上述说的场景类似的问题，就是多个系统都更新库存，但就是某个系统对库存的更新出现了 bug。

这可能是因为那个系统的负责人没理解到底应该如何更新库存，也或者是他更新的时候采用的逻辑，没有考虑到一些特殊情况。

这样导致的结果就是，系统里的库存和仓库中实际的库存，死活对不上。但就是不知道到底哪个环节出了问题，导致库存数据出错。

这个，其实就是一个典型的数据不一致的问题。

六、大型系统的数据不一致排查有多困难

当面对一个大型分布式系统时，如果你之前压根儿没考虑过数据不一致的问题，那么我敢打赌，当你负责的系统在线上被客服反馈有某个核心数据不一致的时候，你绝对会一脸蒙圈。

因为一个核心数据的处理，少则涉及几个系统的协作处理，多则涉及十个以上的系统的协作处理。

如果你没有留存任何日志、或者仅仅就是有部分日志，然后基本就只能所有人干瞪眼，大家大眼对小眼，都盯着自己的代码看。

大家根据一个数据最后的错误结果，比如数据 87。10 多个人对着自己的代码，反复的思考，冥思苦想。

然后每个人都在大脑中疯狂的模拟自己代码的运行，但是就是想不明白，为什么本来应该是数据 4 的，结果出来了一个数据 87?

所以现实问题就是这样，这种数据不一致的问题，大概有以下几个痛点：

- 自己基本无法主动提前感知到数据问题，要被动等待用户发现，反馈给客服，这很可能导致你的产品被大量投诉，老板很生气，后果很严重。
- 即使客服告诉了你数据错了，但是你们没法还原现场，没有留存证据，基本就是一群工程师对着代码想象，猜测。
- 即使你解决了一次数据不一致的问题，但是以后也许还有下一次，这样搞下去，会导致团队里好几个能干的小伙儿时间都搭在这种破事儿上。

七、下篇预告

所以针对本文描述的大型分布式系统数据不一致的问题，下篇文章我们将给出：在百亿流量的场景下，一套复杂系统我们是如何构建整套核心数据保证方案的。

敬请期待：

-亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）

-亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）

亿级流量系统架构之如何保证百亿流量下的数据一致性（中）？

> 作者:中华石杉 原文地址

目录

上篇文章 亿级流量系统架构之如何在上万并发场景下设计可扩展架构（上），初步给大家分析了一下，一个复杂的分布式系统中，数据不一致的问题是怎么产生的。

简单来说，就是一个分布式系统中的多个子系统（或者服务）协作处理一份数据，但是最后这个数据的最终结果却没有符合期望。

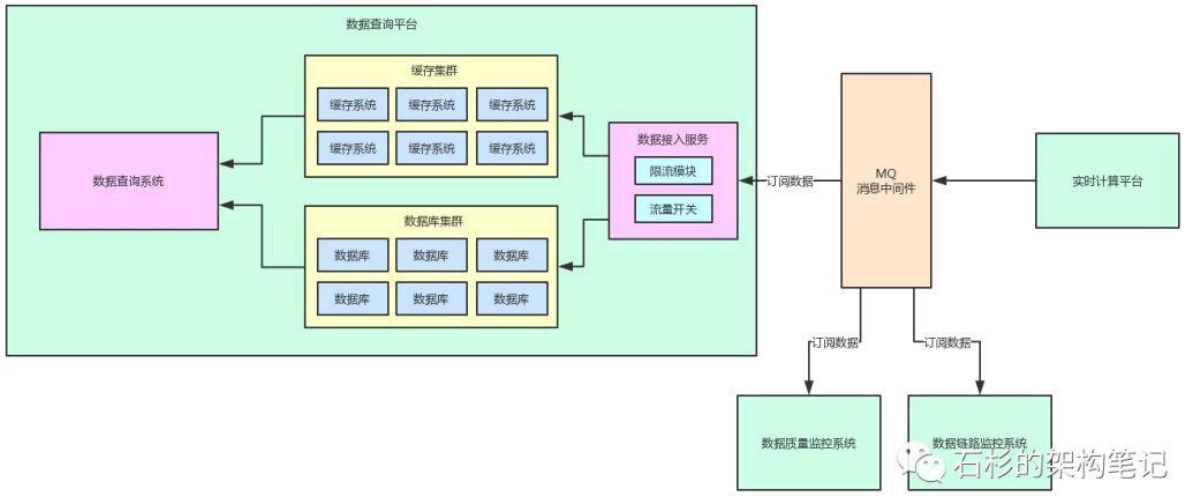
这是一种非常典型的数据不一致的问题。当然在分布式系统中，数据不一致问题还有其他的一些情况。

比如说多个系统都要维护一份数据的多个副本，结果某个系统中的数据副本跟其他的副本不一致，这也是数据不一致。

但是这几篇文章，说的主要是我们上篇文章分析的那种数据不一致的问题到底应该如何解决。

1 多系统订阅数据回顾

我们先来看一张图，是之前讲系统架构解耦的时候用的一张图。



好！通过上面这张图，我们来回顾一下之前做了系统解耦之后的一个架构图。

其实，实时计算平台会把数据计算的结果投递到一个消息中间件里。

然后，数据查询平台、数据质量监控系统、数据链路追踪系统，各个系统都需要那个数据计算结果，都会去订阅里面的数据。

这个就是当前的一个架构，所以这个系列文章分析到这里，大家也可以反过来理解了之前为什么要做系统架构的解耦了。

因为一份核心数据，是很多系统都可能会需要的。通过引入 MQ 对架构解耦了之后，各个系统就可以按需订阅数据了。

2 核心数据的监控系统

如果要解决核心数据的不一致问题，首先就是要做核心数据的监控。

有些同学会以为这个监控就是用 falcon 之类的系统，做业务 metrics 监控就可以了，但是其实并不是这样。

这种核心数据的监控，远远不是做一个 metrics 监控可以解决的。

在我们的实践中，必须要自己开发一个核心数据的监控系统，在里面按照自己的需求，针对复杂的数据校验逻辑开发大量的监控代码。

我们用那个数据平台项目来举例，自己写的数据质量监控系统，需要把核心的一些数据指标从 MQ 里消费出来，这些数据指标都是实时计算平台计算好的。

那么此时，就需要自定义一套监控逻辑了，这种监控逻辑，不同的系统都是完全不一样的。

比如在这种数据类的系统里，很可能对数据指标 A 的监控逻辑是如下这样的：

> **数据指标 A = 数据指标 B + 数据指标 C - 数据指标 D * 24。**

每个核心指标都是有自己的一个监控公式的，这个监控公式，就是负责开发实时计算平台的同学，他们写的数据计算逻辑，是知道数据指标之间的逻辑关系的。

所以此时就有了一个非常简单的思路：

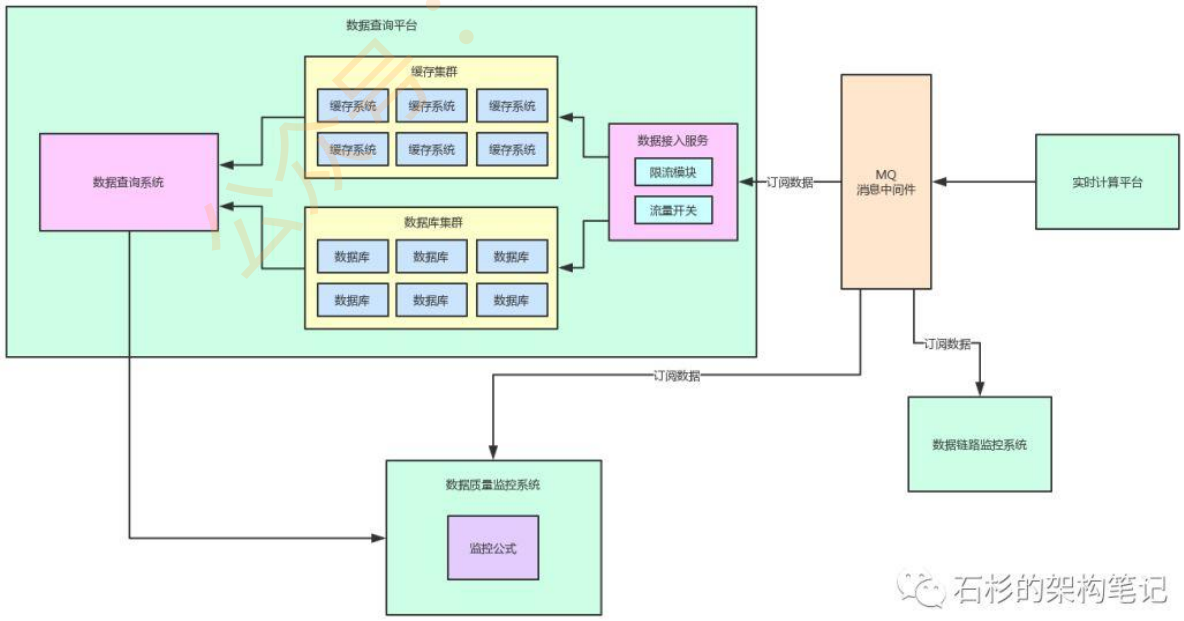
- 首先，这个数据监控系统从 MQ 里消费到每一个最新计算出来的核心数据指标
- 然后根据预先定义好的监控公式，从数据查询平台里调用接口获取出来公式需要的其他数据指标
- 接着，按照公式进行监控计算。

如果监控计算过后发现几个数据指标之间的关系居然不符合预先定义好的那个规则，那么此时就可以立马发送报警了（短信、邮件、IM 通知）。

工程师接到这报警之后，就可以立马开始排查，为什么这个数据居然会不符合预先定义好的一套业务规则呢。

这样就可以解决数据问题的第一个痛点：不需要等待用户发现后反馈给客服了，自己系统第一时间就发现了数据的异常。

同样，给大家上一张图，直观的感受一下。



3 电商库存数据如何监控

如果用电商里的库存数据来举例也是一样的，假设你想要监控电商系统中的核心数据：库存数据。

首先第一步，在微服务架构中，你必须要收口。

也就是说，在彻底的服务化中，你要保证所有的子系统 / 服务如果有任何库存更新的操作，全部走接口调用请求库存服务。只能是库存服务来负责库存数据在数据库层面的更新操作，这样就完成了收口。

收口了之后做库存数据的监控就好办了，完全可以采用 MySQL binlog 采集的技术，直接用 Mysql binlog 同步中间件来监控数据库中库存数据涉及到的表和字段。

只要库存服务对应的数据库中的表涉及到增删改操作，都会被 Mysql binlog 同步中间件采集后，发送到数据监控系统中去。

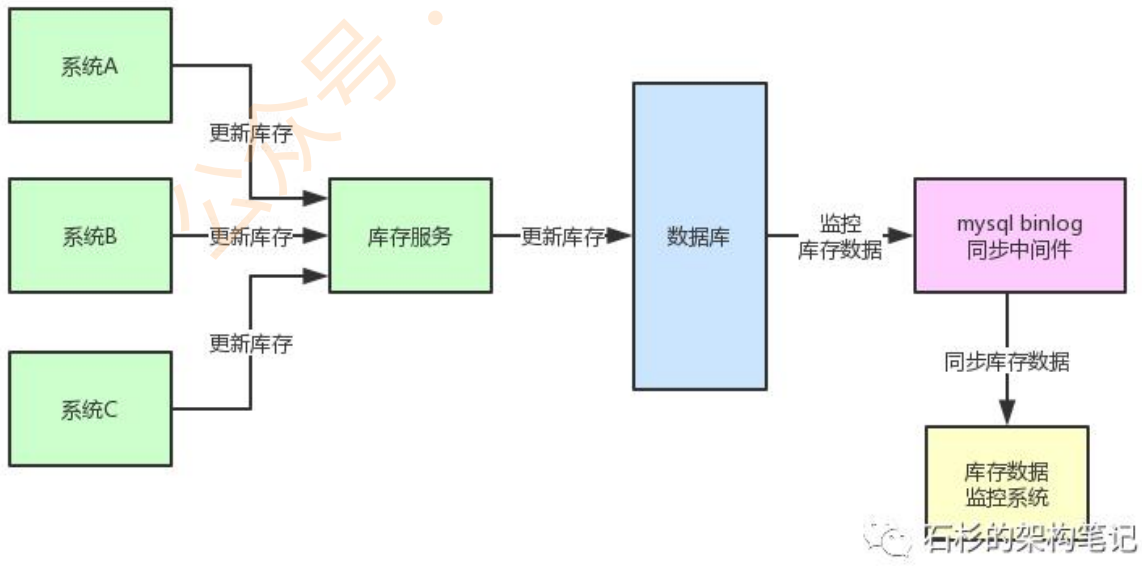
此时，数据监控系统就可以采用预先定义好的库存数据监控逻辑，来查验这个库存数据是否准确。

这个监控逻辑可以是很多种的，比如可以后台走异步线程请求到实际的 C/S 架构的仓储系统中，查一下实际的库存数量。

或者是根据一定的库存逻辑来校验一下，举个例子：

> **虚拟库存 + 预售库存 + 冻结库存 + 可销售库存 = 总可用库存数**

当然，这就是举个例子，实际如何监控，大家根据自己的业务来做就好了。



4 数据计算链路追踪

此时我们已经解决了第一个问题，主动监控系统中的少数核心数据，在第一时间可以自己先收到报警发现核心是护具有异常。

但是此时我们还需要解决第二个问题，那就是当你发现核心数据出错之后，如何快速的排查问题到底出在哪里？

比如，你发现数据平台的某个核心指标出错，或者是电商系统的某个商品库存数据出错，此时你要排查数据到底为什么错了，应该怎么办呢？

很简单，此时我们必须要做数据计算链路的追踪。

也就是说，你必须要知道这个数据从最开始到底是经历了哪些环节和步骤，每个环节到底如何更新了数据，更新后的数据又是什么，还有要记录下来每次数据变更后的监控检查点。

比如说：

- 步骤 A -> 步骤 B -> 步骤 C -> 2018-01-01 10： 00： 00

第一次数据更新后，数据监控检查点，数据校验情况是准确，库存数据值为 1365；

- 步骤 A -> 步骤 B -> 步骤 D -> 步骤 C -> 2018-01-01 11： 05： 00

第二次数据更新后，数据监控检查点，数据校验情况是错误，库存数据值为 1214

类似上面的那种数据计算链路的追踪，是必须要做的。

因为你必须要知道一个核心数据，他每次更新一次值经历了哪些中间步骤，哪些服务更新过他，那一次数据变更对应的数据监控结果如何。

此时，如果你发现一个库存数据出错了，立马可以人肉搜出来这个数据过往的历史计算链路。

你可以看到这条数据从一开始出现，然后每一次变更的计算链路和监控结果。

比如上面那个举例，你可能发现第二次库存数据更新后结果是 1214，这个值是错误的。

然后你一看，发现其实第一次更新的结果是正确的，但是第二次更新的计算链路中多了一个步骤 D 出来，那么可能这个步骤 D 是服务 D 做了一个更新。

此时，你就可以找服务 D 的服务人问问，结果可能就会发现，原来服务 D 没有按照大家约定好的规则来更新库存，结果就导致库存数据出错。

这个，就是排查核心数据问题的一个通用思路。

5 百亿流量下的数据链路追踪

如果要做数据计算链路，其实要解决的技术问题只有一个，那就是在百亿流量的高并发下，任何一个核心数据每天的计算链路可能都是上亿的，此时你应该如何存储呢？

其实给大家比较推荐的，是用 elasticsearch 技术来做这种数据链路的存储。

因为 es 一方面是分布式的，支持海量数据的存储。

而且他可以做高性能的分布式检索，后续在排查数据问题的时候，是需要对海量数据做高性能的多条件检索的。

所以，我们完全可以独立出来一个数据链路追踪系统，并设置如下操作：

- 数据计算过程中涉及到的各个服务，都需要对核心数据的处理发送一条计算链路日志到数据链路追踪系统。
- 然后，数据链路追踪系统就可以把计算链路日志落地到存储里去，按照一定的规则建立好对应的索引字段。
- 举个例子，索引字段：核心数据名称，核心数据 id，本次请求 id，计算节点序号，本次监控结果，子系统名称，服务名称，计算数据内容，等等。

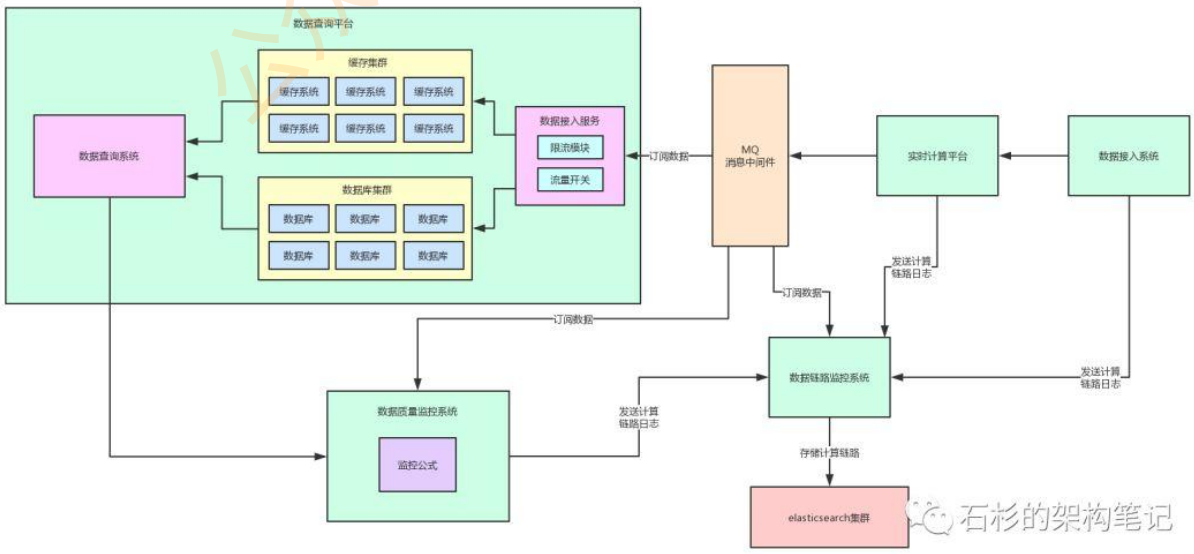
此时一旦发现某个数据出错，就可以立即根据这条数据的 id，从 es 里提取出来历史上所有的计算链路。

而且还可以给数据链路追踪系统开发一套用户友好的前端界面，比如在界面上可以按照请求 id 展示出来每次请求对应的一系列技术步骤组成的链路。

此时会有什么样的体验呢？我们立马可以清晰的看到是哪一次计算链路导致了数据的出错，以及过程中每一个子系统 / 服务对数据做了什么样的修改。

然后，我们就可以追本溯源，直接定位到出错的逻辑，进行分析和修改。

说了那么多，还是给大家来一张图，一起来感受一下这个过程。



6 自动化数据链路分析

到这里为止，大家如果能在自己公司的大规模分布式系统中，落地上述那套数据监控 + 链路追踪的机制，就已经可以非常好的保证核心数据的准确性了。

通过这套机制，核心数据出错时，第一时间可以收到报警，而且可以立马拉出数据计算链路，快速的分析数据为何出错。

但是，如果要更进一步的节省排查数据出错问题的人力，那么可以在数据链路追踪系统里面加入一套自动化数据链路分析的机制。

大家可以反向思考一下，假如说现在你发现数据出错，而且手头有数据计算链路，你会怎么检查?

不用说，当然是大家坐在一起唾沫横飞的分析了，人脑分析。

比如说，步骤 A 按理说执行完了应该数据是 X，步骤 B 按理说执行完了应该数据是 Y，步骤 C 按理说执行完了应该数据是 Z。

结果，诶！步骤 C 执行完了怎么数据是 ZZZ 呢？？看来问题就出在步骤 C 了！

然后去步骤 C 看看，发现原来是服务 C 更新的，此时服务 C 的负责人开始吭哧吭哧的排查自己的代码，看看到底为什么接收到一个数据 Y 之后，自己的代码会处理成数据 ZZZ，而不是数据 Z 呢?

最后，找到了代码问题，此时就 ok 了，在本地再次复现数据错误，然后修复 bug 后上线即可。

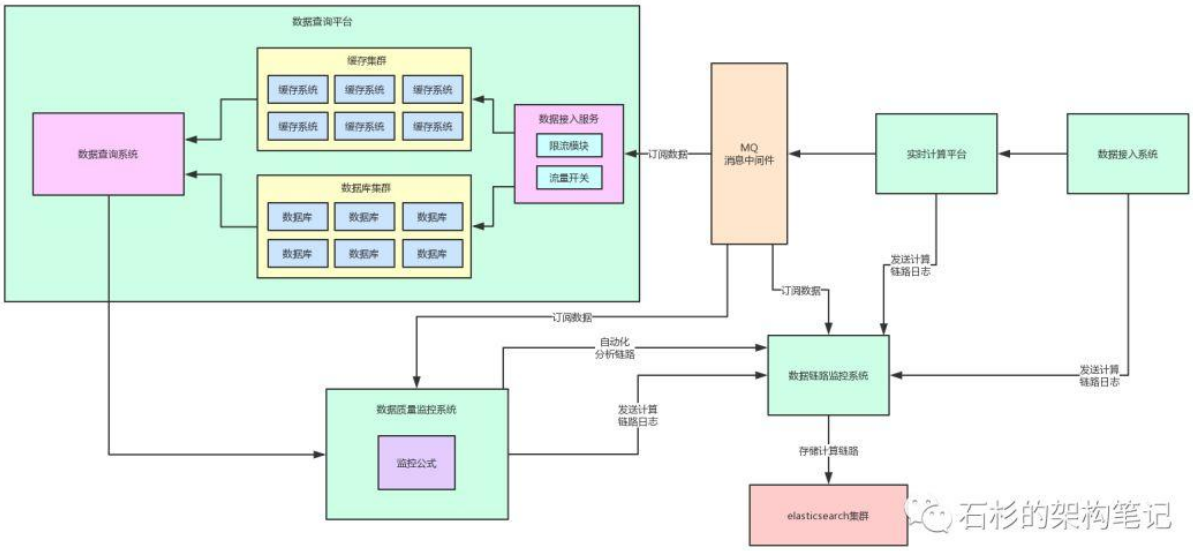
所以，这个过程的前半部分，是完全可以自动化的。也就是你写一套自动分析数据链路的代码，就模拟你人脑分析链路的逻辑即可，自动一步步分析每个步骤的计算结果。这样就可以把数据监控系统和链路追踪系统打通了。

一旦数据监控系统发现数据出错，立马可以调用链路追踪系统的接口，进行自动化的链路分析，看看本次数据出错，到底是链路中的哪个服务 bug 导致的数据问题。

接着，将所有的信息汇总起来，发送一个报警通知给相关人等。

相关人员看到报警之后，一目了然，所有人立马知道本次数据出错，是链路中的哪个步骤，哪个服务导致的。

最后，那个服务的负责人就可以立马根据报警信息，排查自己的系统中的代码了。



7 下篇预告

到这篇文章为止，我们基本上梳理清楚了大规模的负责分布式系统中，如何保证核心数据的一致性。

那么下篇文章，我们再就技术实现中涉及到的一些 MQ 技术的细节，基于 RabbitMQ 来进行更进一步的分析。

敬请期待：

-亿级流量系统架构之如何在上万并发场景下设计可扩展架构（下）

亿级流量系统架构之如何保证百亿流量下的数据一致性（下）？

> 作者:中华石杉 原文地址

目录

- 六、更加强大而且灵活的按需订阅

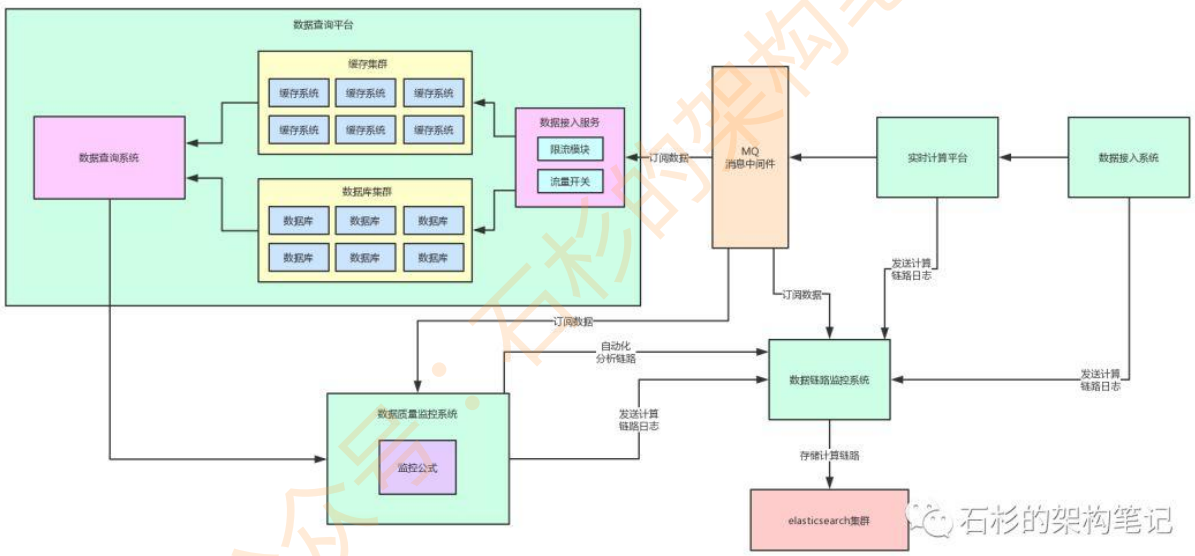
一、前情提示

上一篇文章-[亿级流量系统架构之如何在上万并发场景下设计可扩展架构（中）]，我们已经给出了一整套的数据一致性的保障方案。

我们从如下三个角度，给出了方案如何实现。并且通过数据平台和电商系统进行了举例分析。

- 核心数据的监控
- 数据链路追踪
- 自动化数据链路分析

目前为止，我们的架构图大概如下所示：



并且咱们之前对于这种架构下，如何基于 MQ 进行解耦的实现也做了详细的说明。

有不清楚的同学，可以具体看一下之前的三篇文章：

- [如何在上万并发场景下设计可扩展架构（上）]
- [如何在上万并发场景下设计可扩展架构（中）]
- [如何在上万并发场景下设计可扩展架构（下）]

那么这篇文章，我们就基于这个架构，在数据一致性方面做进一步的说明。同样，我们以 RabbitMQ 这个消息中间件来举例。

二、 选择性的订阅部分核心数据

首先一个基于 MQ 实现的细节点就在于，比如对数据监控系统而言，他可能仅仅只是要从 MQ 里订阅部分数据来消费罢了。

这个是啥意思呢？因为比如实时计算平台他是会将自己计算出来的所有的数据指标都投递到 MQ 里去的。

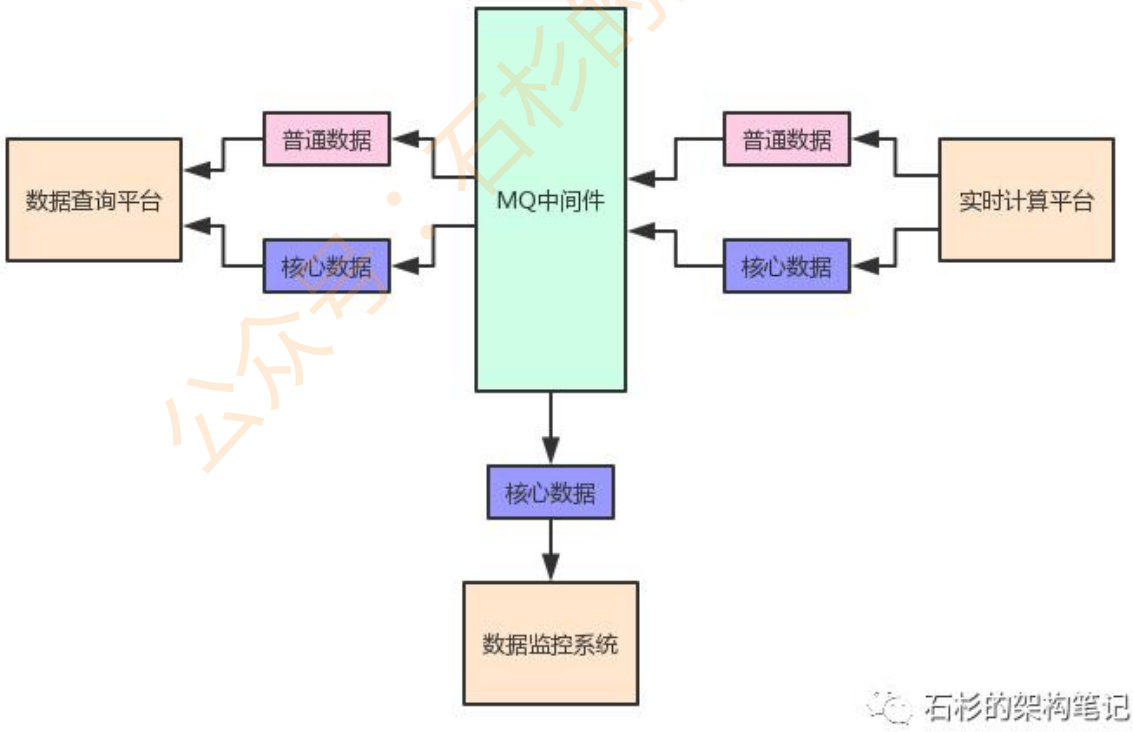
但是这些数据指标可能是多达几十个甚至是几百个的，这里面不可能所有数据指标都是核心数据吧？

基本上按照我们过往经验而言，对于这种数据类的系统核心数据指标，大概就占到 10% 左右的比例而已。

然后对于数据查询平台而言，他可能是需要把所有的数据指标都消费出来，然后落地到自己的存储里去的。

但是对于数据监控系统而言，他只需要过滤出 10% 的核心数据指标即可，所以他需要的是有选择性的订阅数据。

咱们看看下面的图，立马就明白是什么意思了。



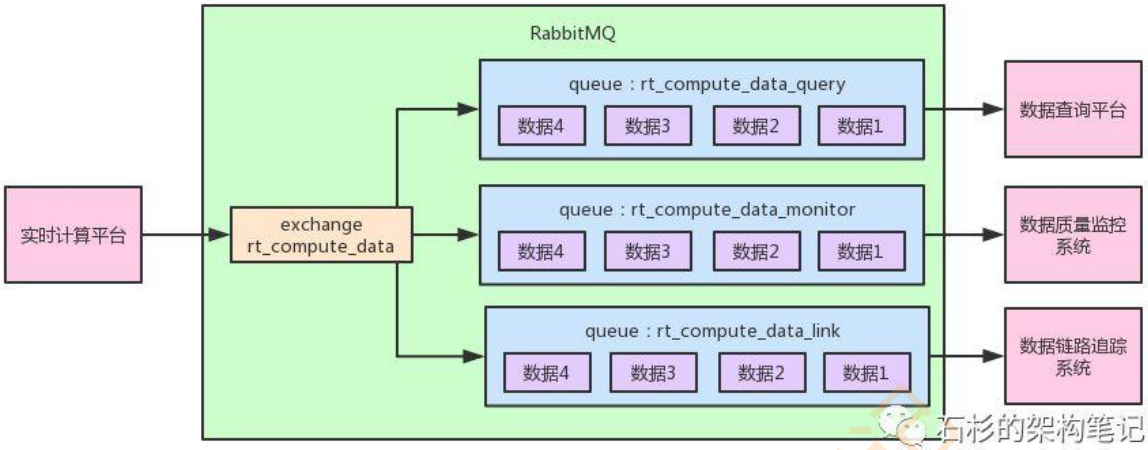
三、RabbitMQ 的 queue 与 exchange 的绑定回顾

不知道大家是否还记得之前讲解基于 RabbitMQ 实现多系统订阅同一份数据的场景。

我们采用的是每个系统使用自己的一个 queue，但是都绑定到一个 fanout exchange 上去，然后生产者直接投递数据到 fanout exchange。

fanout exchange 会分发一份数据，绑定到自己的所有 queue 上去，然后各个系统都会从自己的 queue 里拿到相同的一份数据。

大家再看看下面的图回顾一下。



在这里有一个关键的代码如下所示：

```
// 在消费者里也声明下exchange
channel.exchangeDeclare("rt_compute_data","fanout");

// 这里是创建一个持久化的队列
// 名字的意思，就是这个队列是查询平台作为消费者来消费的
channel.queueDeclare("rt_compute_data_query", true, false, false, null);

// 这里就是把这个消费者自己的队列绑定到了exchange上去
channel.queueBind("rt_compute_data_query", "rt_compute_data", "");
```


也就是说，把自己创建的 queue 绑定到 exchange 上去，这个绑定关系在 RabbitMQ 里有一个专业的术语叫做：binding。

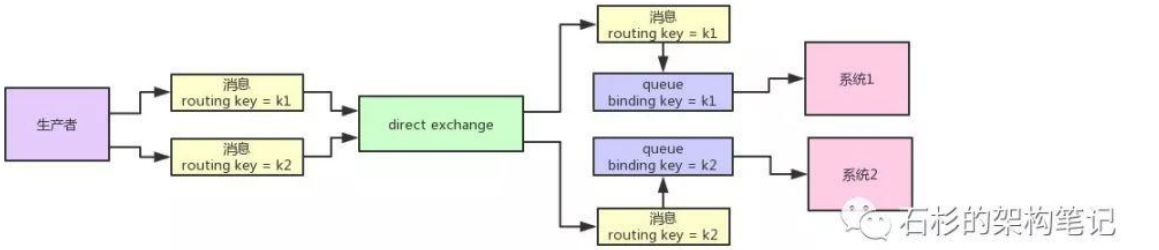
四、direct exchange 实现消息路由

如果仅仅使用之前的 fanout exchange，那么是无法实现不同的系统按需订阅数据的，如果要实现允许不同的系统按需订阅数据，那么需要使用 direct exchange。

direct exchange 允许你在投递消息的时候，给每个消息打上一个 routing key。同时 direct exchange 还允许 binding 到自己的 queue 指定一个 binding key。

这样，direct exchange 就会根据消息的 routing key 将这个消息路由到相同 binding key 对应的 queue 里去，这样就可以实现不同的系统按需订阅数据了。

说了这么多，是不是感觉有点晕，老规矩，咱们来一张图，直观的感受一下怎么回事儿：



而且一个 queue 是可以使用多个 binding key 的，比如说使用 “k1” 和“k2”两个 binding key 的话，那么 routing key 为 “k1” 和“k2”的消息都会路由到那个 queue 里去。

同时不同的 queue 也可以指定相同的 ruoting key，这个时候就跟 fanout exchange 其实是一样的了，一个消息会同时路由到多个 queue 里去。

五、按需订阅数的代码实现

首先在生产者那块，比如说实时计算平台吧，他就应该是要定义一个 direct exchange 了。

如下代码所示，所有的数据都是投递到这个 exchange 里去，比如我们这里使用的 exchange 名字就是 “rt_data”，意思就是实时数据计算结果，类型是 “direct”：

```php
channel.exchangeDeclare(
                "rt_data",
                "direct");
```

而且，在投递消息的时候，要给一个消息打上标签，也就是他的 routing key，表明这个消息是普通数据还是核心数据，这样才能实现路由，如下代码所示：

```
channel.basicPublish("rt_data",
                     "common_data",
                     MessageProperties.PERSISTENT_TEXT_PLAIN,
                     message.getBytes());
```


上面第一个参数是指定要投递到哪个 exchange 里去，第二个参数就是 routing key，这里的 “common_data”代表了是普通数据，也可以用 “core_data” 代表核心数据，实时计算平台根据自己的情况指定普通或者核心数据。

然后消费者在进行 queue 和 exchange 的 binding 的时候，需要指定 binding key，代码如下所示：

```
channel.exchangeDeclare("rt_data", "direct");

channel.queueDeclare("rt_data_monitor", true, false, false, null);

channel.queueBind("rt_data_monitor", "rt_data", "core_data");
```


上面第一行就是在消费者那里，比如数据监控系统那里，也是定义一下 direct exchange。

然后第二行就是定义一个 “rt_data_monitor“这个 queue。

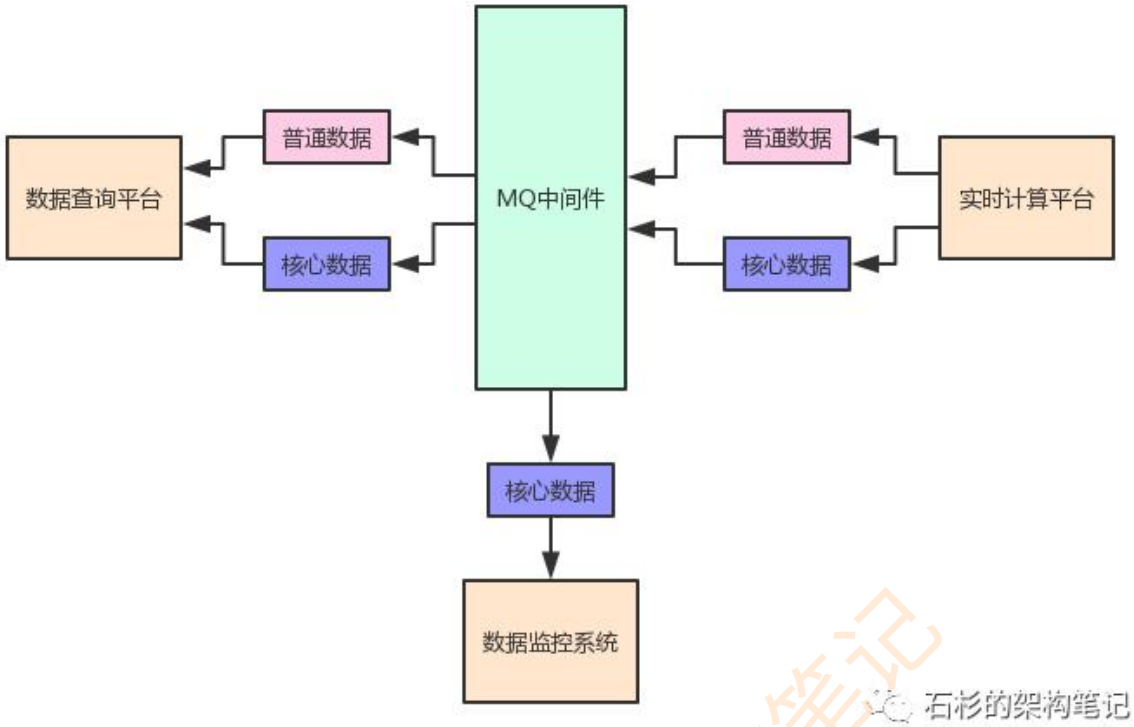
第三行就是对 queue 和 exchange 进行绑定，指定了 binding key 是 “core_data”。

如果是数据查询系统，他是普通数据和核心数据都要的，那么就可以在 binding key 里指定多个值，用逗号隔开，如下所示：

```php
channel.queueBind(
                "rt_data_query",
                "rt_data",
                "common_data, core_data");
```

到这里，大家就明白如何对数据打上不同的标签（也就是 routing key），然后让不同的系统按需订阅自己需要的数据了（也就是指定 binding key），这种方式用到了 direct exchange 这种类型，非常的灵活。

最后，再看看之前画的那幅图，大家再来感受一下即可：



六、更加强大而且灵活的按需订阅

RabbitMQ 还支持更加强大而且灵活的按需数据订阅，也就是使用 topic exchange，其实跟 direct exchange 是类似的，只不过功能更加的强大罢了。

比如说你定义一个 topic exchange，然后 routing key 就需要指定为用点号隔开的多个单词，如下所示：

```
channel.exchangeDeclare("rt_data", "topic");

// 这里第二个参数routing key就要这样子来设置了，意思就是商品类的普通数据
channel.basicPublish("rt_data",
                     "product.common.data",
                     MessageProperties.PERSISTENT_TEXT_PLAIN,
                     message.getBytes());
```


然后，你在设置 binding key 的时候，他是支持通配符的。* 匹配一个单词，# 匹配 0 个或者多个单词，比如说你的 binding key 可以这么来设置：

```
channel.exchangeDeclare("rt_data", "direct");

channel.queueDeclare("rt_data_monitor", true, false, false, null);

channel.queueBind("rt_data_monitor", "rt_data", "product.*.*");
```


这个 product.. ，就会跟 “product.common.data” 匹配上，意思就是，可能某个系统就是对商品类的数据指标感兴趣，不管是普通数据还是核心数据。

所以到这里，大家就应该很容易明白了，通过 RabbitMQ 的 direct、topic 两种 exchange，我们可以轻松实现各种强大的数据按需订阅的功能。

通过本文，我们就将最近讲的数据一致性保障方案里的一些 MQ 中间件落地的细节给大家说明白了。

兄弟，用大白话告诉你小白都能看懂的Hadoop架构原理

> **作者:中华石杉** 原文地址

目录

一、前情概要

这篇文章给大家聊聊 Hadoop 在部署了大规模的集群场景下，大量客户端并发写数据的时候，文件契约监控算法的性能优化。

看懂这篇文章需要一些 Hadoop 的基础知识背景，还不太了解的兄弟，可以先看看之前的文章：**兄弟，用大白话告诉你小白都能看懂的Hadoop架构原理**。

二、背景引入

先给大家引入一个小的背景，假如多个客户端同时要并发的写 Hadoop HDFS 上的一个文件，大家觉得这个事儿能成吗?

明显不可以接受啊，兄弟们，HDFS 上的文件是不允许并发写的，比如并发的追加一些数据什么的。

所以说，HDFS 里有一个机制，叫做文件契约机制。

也就是说，同一时间只能有一个客户端获取 NameNode 上面一个文件的契约，然后才可以写入数据。此时如果其他客户端尝试获取文件契约的时候，就获取不到，只能干等着。

通过这个机制，就可以保证同一时间只有一个客户端在写一个文件。

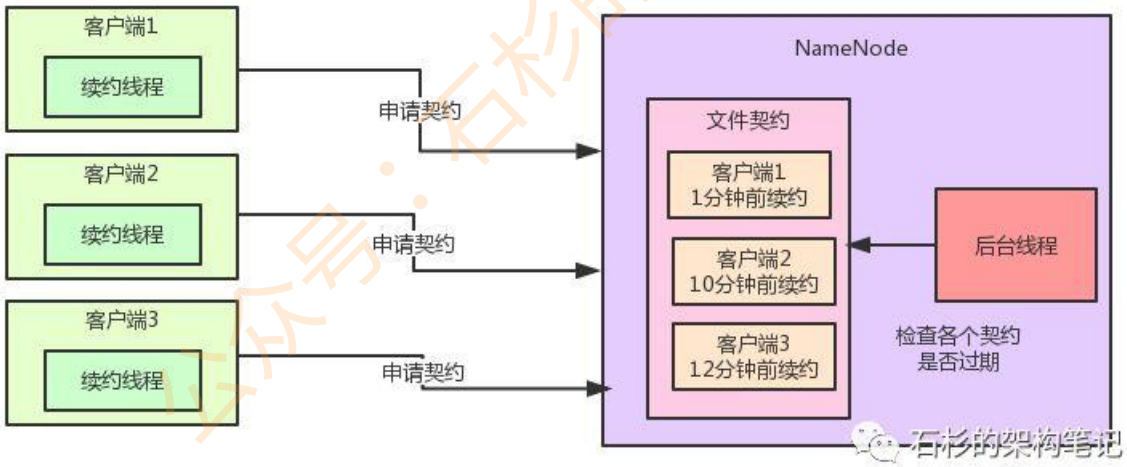
在获取到了文件契约之后，在写文件的过程期间，那个客户端需要开启一个线程，不停的发送请求给 NameNode 进行文件续约，告诉 NameNode：

NameNode 大哥，我还在写文件啊，你给我一直保留那个契约好吗？

而 NameNode 内部有一个专门的后台线程，负责监控各个契约的续约时间。

如果某个契约很长时间没续约了，此时就自动过期掉这个契约，让别的客户端来写。

说了这么多，老规矩，给大家来一张图，直观的感受一下整个过程。



三、问题凸现

好，那么现在问题来了，假如我们有一个大规模部署的 Hadoop 集群，同时存在的客户端可能多达成千上万个。

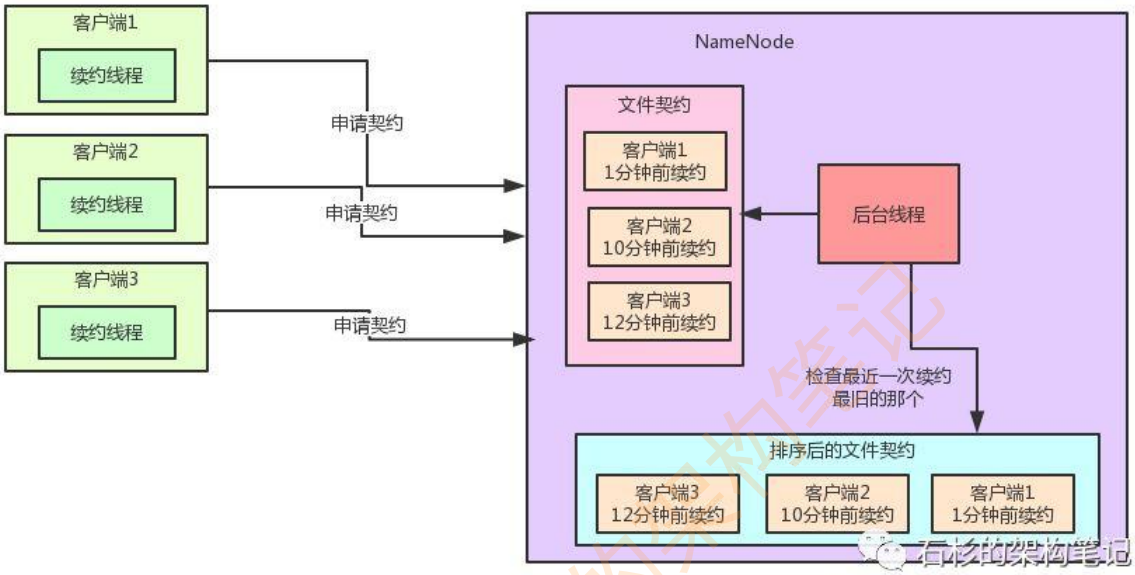
此时 NameNode 内部维护的那个文件契约列表会非常非常的大，而监控契约的后台线程又需要频繁的每隔一段时间就检查一下所有的契约是否过期。

比如，每隔几秒钟就遍历大量的契约，那么势必造成性能不佳，所以说这种契约监控机制明显是不适合大规模部署的 hadoop 集群的。

四、Hadoop 的优化方案

那么 Hadoop 是如何对文件契约监控算法进行优化的呢？咱们来一步一步的看一下他的实现逻辑。

首先，我们一起来看看下面这张手绘图：



其实奥秘十分的简单，每次一个客户端发送续约请求之后，就设置这个契约的最近一次续约时间。

然后，基于一个 TreeSet 数据结构来根据最近一次续约时间对契约进行排序，每次都把续约时间最老的契约排在最前头，这个排序后的契约数据结构十分的重要。

TreeSet 是一种可排序的数据结构，他底层基于 TreeMap 来实现。

TreeMap 底层则基于红黑树来实现，可以保证元素没有重复，同时还能按照我们自己定义的排序规则在你每次插入一个元素的时候来进行自定义的排序。

所以这里我们的排序规则：就是按照契约的最近一次续约时间来排序。

其实这个优化就是如此的简单，就是维护这么一个排序数据结构而已。

我们现在来看一下 Hadoop 中的契约监控的源码实现：

每次检查契约是否过期的时候，你不要遍历成千上万的契约，那样遍历效率当然会很低下。

我们完全可以就从 TreeSet 中获取续约时间最老的那个契约，假如说连最近一次续约时间最老的那个契约都还没过期，那么就不用继续检查了啊！这说明续约时间更近的那些契约绝对不会过期！

举个例子：续约时间最老的那个契约，最近一次续约的时间是 10 分钟以前，但是我们判断契约过期的限制是超过 15 分钟不续约就过期那个契约。

这个时候，连 10 分钟以前续约的契约都没有过期，那么那些 8 分钟以前，5 分钟以前续约的契约，肯定也不会过期啊！

这个机制的优化对性能的提升是相当有帮助的，因为正常来说，过期的契约肯定还是占少数，所以压根儿不用每次都遍历所有的契约来检查是否过期。

我们只需要检查续约时间最旧的那几个契约就可以了，如果一个契约过期了，那么就删掉那个契约，然后再检查第二旧的契约好了。以此类推。

通过这个 TreeSet 排序 + 优先检查最旧契约的机制，有效的将大规模集群下的契约监控机制的性能提升至少 10 倍以上，这种思想是非常值得我们学习和借鉴的。

给大家稍微引申一下，在 Spring Cloud 微服务架构中，Eureka 作为注册中心其实也有续约检查的机制，跟 Hadoop 是类似的。

如果想了解 Eureka 注册中心相关技术的朋友，建议看一下：《拜托，面试请不要再问我 Spring Cloud 底层原理！》。

但是在 Eureka 中就没有实现类似的续约优化机制，而是暴力的每一轮都遍历所有的服务实例的续约时间。

如果你面对的是一个大规模部署的微服务系统呢，情况就不妙了！

部署了几十万台机器的大规模系统，有几十万个服务实例的续约信息驻留在 Eureka 的内存中，难道每隔几秒钟都要遍历几十万个服务实例的续约信息吗?

最后给大家提一句，优秀的开源项目，蕴含着很多优秀的设计思想。多看各种优秀开源项目的源码，是短时间内快速、大幅度提升一个人的技术功底和技术水平的方式，大家不妨尝试一下。

大规模集群下 Hadoop NameNode 如何承载每秒上千次的高并发访问

> **作者:中华石杉** 原文地址

目录

一、写在前面

上篇文章我们已经初步给大家解释了 Hadoop HDFS 的整体架构原理，相信大家都有了一定的认识和了解。

如果没看过上篇文章的同学可以看一下：《兄弟，用大白话告诉你小白都能听懂的 Hadoop 架构原理》这篇文章。

本文我们来看看，如果大量客户端对 NameNode 发起高并发（比如每秒上千次）访问来修改元数据，此时 NameNode 该如何抗住?

二、问题源起

我们先来分析一下，高并发请求 NameNode 会遇到什么样的问题。

大家现在都知道了，每次请求 NameNode 修改一条元数据（比如说申请上传一个文件，那么就需要在内存目录树中加入一个文件），都要写一条 edits log，包括两个步骤：

- 写入本地磁盘。

- 通过网络传输给 JournalNodes 集群。

但是如果对 Java 有一定了解的同学都该知道多线程并发安全问题吧?

NameNode 在写 edits log 时的第一条原则:

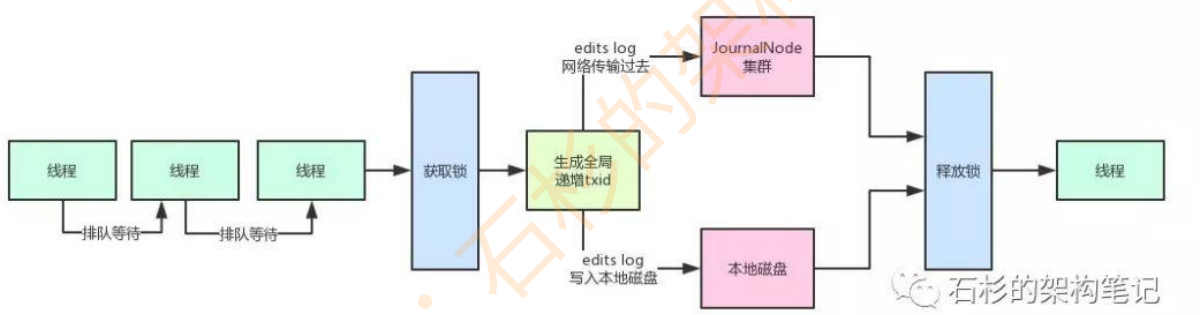
必须保证每条 edits log 都有一个全局顺序递增的 transactionId（简称为 txid），这样才可以标识出来一条一条的 edits log 的先后顺序。

那么如果要保证每条 edits log 的 txid 都是递增的，就必须得加锁。

每个线程修改了元数据，要写一条 edits log 的时候，都必须按顺序排队获取锁后，才能生成一个递增的 txid，代表这次要写的 edits log 的序号。

好的，那么问题来了，**大家看看下面的图。**

如果每次都是在一个加锁的代码块里，生成 txid，然后写磁盘文件 edits log，网络请求写入 journalnodes 一条 edits log，会咋样?



不用说，这个绝对完蛋了!

NameNode 本身用多线程接收多个客户端发送过来的并发的请求，结果多个线程居然修改完内存中的元数据之后，排着队写 edits log!

而且你要知道，写本地磁盘 + 网络传输给 journalnodes，都是很耗时的啊!性能两大杀手：磁盘写 + 网络写!

如果 HDFS 的架构真要是这么设计的话，基本上 NameNode 能承载的每秒的并发数量就很少了，可能就每秒处理几十个并发请求处理撑死了!

三、HDFS 优雅的解决方案

所以说，针对这个问题，人家 HDFS 是做了不少的优化的!

首先大家想一下，既然咱们不希望每个线程写 edits log 的时候，串行化排队生成 txid + 写磁盘 + 写 JournalNode，那么是不是可以搞一个内存缓冲?

也就是说，多个线程可以快速的获取锁，生成 txid，然后快速的将 edits log 写入内存缓冲。

接着就快速的释放锁，让下一个线程继续获取锁后，生成 id + 写 edits log 进入内存缓冲。

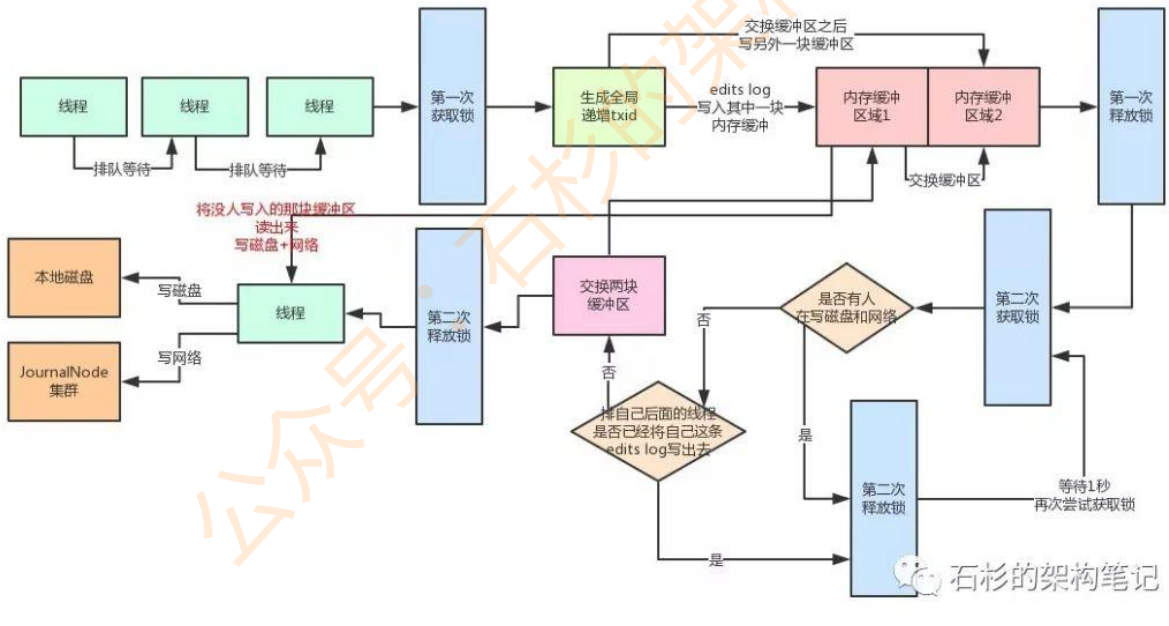
然后接下来有一个线程可以将内存中的 edits log 刷入磁盘，但是在这个过程中，还是继续允许其他线程将 edits log 写入内存缓冲中。

但是这里又有一个问题了，如果针对同一块内存缓冲，同时有人写入，还同时有人读取后写磁盘，那也有问题，因为不能并发读写一块共享内存数据！

所以 HDFS 在这里采取了** double-buffer 双缓冲机制**来处理！将一块内存缓冲分成两个部分：

- 其中一个部分可以写入
- 另外一个部分用于读取后写入磁盘和 JournalNodes。

大家可能感觉文字叙述不太直观，老规矩，**咱们来一张图**，按顺序给大家阐述一下。



（1）分段加锁机制 + 内存双缓冲机制

首先各个线程依次第一次获取锁，生成顺序递增的 txid，然后将 edits log 写入内存双缓冲的区域 1，接着就立马第一次释放锁了。

趁着这个空隙，后面的线程就可以再次立马第一次获取锁，然后立即写自己的 edits log 到内存缓冲。

写内存那么快，可能才耗时几十微妙，接着就立马第一次释放锁了。所以这个并发优化绝对是有效果的，大家有没有感受到?

接着各个线程竞争第二次获取锁，有线程获取到锁之后，就看看，有没有谁在写磁盘和网络?

如果没有，好，那么这个线程是个幸运儿！直接交换双缓冲的区域 1 和区域 2，接着第二次释放锁。这个过程相当快速，内存里判断几个条件，耗时不了几微秒。

好，到这一步为止，内存缓冲已经被交换了，后面的线程可以立马快速的依次获取锁，然后将 edits log 写入内存缓冲的区域 2，区域 1 中的数据被锁定了，不能写。

怎么样，是不是又感受到了一点点多线程并发的优化?

#####（2）多线程并发吞吐量的百倍优化 接着，之前那个幸运儿线程，将内存缓冲的区域 1 中的数据读取出来（此时没人写区域 1 了，都在写区域 2），将里面的 edtis log 都写入磁盘文件，以及通过网络写入 JournalNodes 集群。

这个过程可是很耗时的！但是没关系啊，人家做过优化了，在写磁盘和网络的过程中，是不持有锁的！

因此后面的线程可以噼里啪啦的快速的第一次获取锁后，立马写入内存缓冲的区域 2，然后释放锁。

这个时候大量的线程都可以快速的写入内存，没有阻塞和卡顿！

怎么样？并发优化的感觉感受到了没有！

#####（3）缓冲数据批量刷磁盘 + 网络的优化 那么在幸运儿线程吭哧吭哧把数据写磁盘和网络的过程中，排在后面的大量线程，快速的第一次获取锁，写内存缓冲区域 2，释放锁，之后，这些线程第二次获取到锁后会干嘛?

他们会发现有人在写磁盘啊，兄弟们！所以会立即休眠 1 秒，释放锁。

此时大量的线程并发过来的话，都会在这里快速的第二次获取锁，然后发现有人在写磁盘和网络，快速的释放锁，休眠。

怎么样，这个过程没有人长时间的阻塞其他人吧！因为都会快速的释放锁，所以后面的线程还是可以迅速的第一次获取锁后写内存缓冲！

again！并发优化的感觉感受到了没有?

而且这时，一定会有很多线程发现，好像之前那个幸运儿线程的 txid 是排在自己之后的，那么肯定就把自己的 edits log 从缓冲里写入磁盘和网络了。

这些线程甚至都不会休眠等待，直接就会返回后去干别的事情了，压根儿不会卡在这里。这里又感受到并发的优化没有?

然后那个幸运儿线程写完磁盘和网络之后，就会唤醒之前休眠的那些线程。

那些线程会依次排队再第二次获取锁后进入判断，咦！发现没有人在写磁盘和网络了！

然后就会再判断，有没有排在自己之后的线程已经将自己的 edtis log 写入磁盘和网络了。

- 如果有的话，就直接返回了。
- 没有的话，那么就成为第二个幸运儿线程，交换两块缓冲区，区域 1 和区域 2 交换一下。
- 然后释放锁，自己开始吭哧吭哧的将区域 2 的数据写入磁盘和网络。

但是这个时候没有关系啊，后面的线程如果要写 edits log 的，还是可以第一次获取锁后立马写内存缓冲再释放锁。以此类推。

四、总结

其实这套机制还是挺复杂的，涉及到了分段加锁以及内存双缓冲两个机制。

通过这套机制，NameNode 保证了多个线程在高并发的修改元数据之后写 edits log 的时候，不会说一个线程一个线程的写磁盘和网络，那样性能实在太差，并发能力太弱了！

所以通过上述那套复杂的机制，尽最大的努力保证，一个线程可以批量的将一个缓冲中的多条 edits log 刷入磁盘和网络。

在这个漫长的吭哧吭哧的过程中，其他的线程可以快速的高并发写入 edits log 到内存缓冲里，不会阻塞其他的线程写 edits log。

所以，正是依靠以上机制，最大限度优化了 NameNode 处理高并发访问修改元数据的能力！

Hadoop如何将TB级大文件的上传性能优化上百倍?

> **作者:中华石杉** 原文地址

目录

一、写在前面

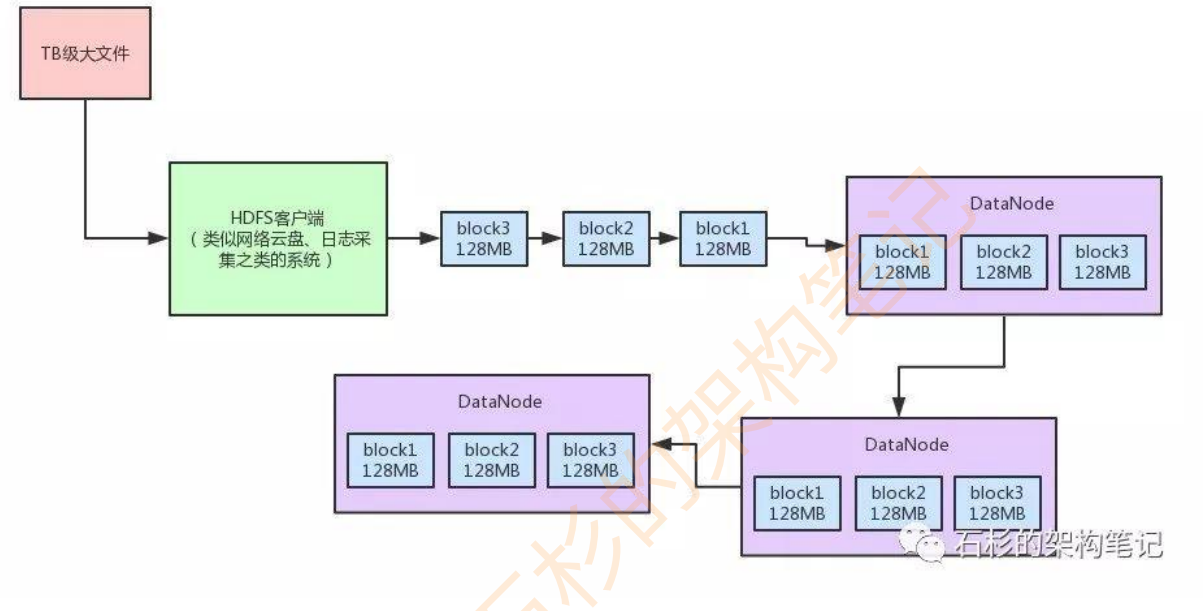
上一篇文章，我们聊了一下 Hadoop 中的 NameNode 里的 edits log 写机制。

主要分析了 edits log 写入磁盘和网络的时候，是如何通过分段加锁以及双缓冲的机制，大幅度提升了多线程并发写 edits log 的吞吐量，从而支持高并发的访问。

如果没看那篇文章的同学，可以回看一下：**大规模集群下Hadoop NameNode如何承载每秒上千次的高并发访问**

这篇文章，我们来看看，Hadoop 的 HDFS 分布式文件系统的文件上传的性能优化。

首先，我们还是通过一张图来回顾一下文件上传的大概的原理。



由上图所示，文件上传的原理，其实说出来也简单。

比如有个 TB 级的大文件，太大了，HDFS 客户端会给拆成很多 block，一个 block 就是 128MB。

这个 HDFS 客户端你可以理解为是云盘系统、日志采集系统之类的东西。

比如有人上传一个 1TB 的大文件到网盘，或者是上传个 1TB 的大日志文件。

然后，HDFS 客户端把一个一个的 block 上传到第一个 DataNode

第一个 DataNode 会把这个 block 复制一份，做一个副本发送给第二个 DataNode。

第二个 DataNode 发送一个 block 副本到第三个 DataNode。

所以你会发现，一个 block 有 3 个副本，分布在三台机器上。任何一台机器宕机，数据是不会丢失的。

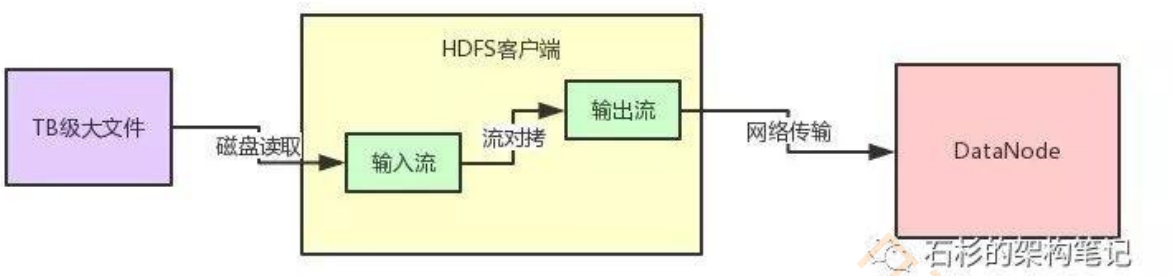
最后，一个 TB 级大文件就被拆散成了 N 多个 MB 级的小文件存放在很多台机器上了，这不就是分布式存储么？

二、原始的文件上传方案

今天要讨论的问题，就是那个 HDFS 客户端上传 TB 级大文件的时候，到底是怎么上传呢？

我们先来考虑一下，如果用一个比较原始的方式来上传，应该怎么做？

大概能想到的是下面这个图里的样子。



很多 java 的初学者，估计都知道这样来上传文件，其实无非就是不停的从本地磁盘文件用输入流读取数据，读到一点，就立马通过网络的输出流写到 DataNode 里去。

上面这种流程图的代码，估计刚毕业的同学都可以立马写出来。因为对文件的输入流最多就是个 FileInputStream。

而对 DataNode 的输出流，最多就是个 Socket 返回的 OutputStream。

然后中间找一个小的内存 byte[] 数组，进行流对拷就行了，从本地文件读一点数据，就给 DataNode 发一点数据。

但是如果你要这么弄，那性能真是极其的低下了，网络通信讲究的是适当频率，每次 batch 批量发送，你得读一大批数据，通过网络通信发一批数据。

不能说读一点点数据，就立马来一次网络通信，就发出去这一点点的数据。

如果按照上面这种原始的方式，绝对会导致网络通信效率极其低下，大文件上传性能很差。

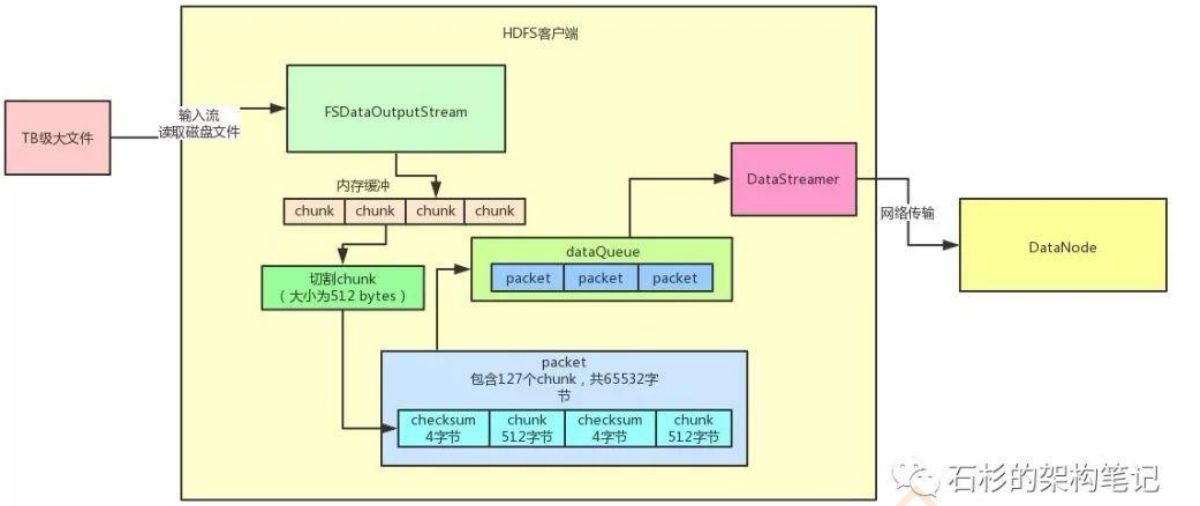
为什么这么说呢？

相当于你可能刚读出来几百个字节的数据，立马就写网络，卡顿个比如几百毫秒。

然后再读下一批几百个字节的数据，再写网络卡顿个几百毫秒，这个性能很差，在工业级的大规模分布式系统中，是无法容忍的。

三、HDFS 对大文件上传的性能优化

好，看完了原始的文件上传，那么我们来看看，Hadoop 中的大文件上传是如何优化性能的呢？一起来看看下面那张图。



首先你需要自己创建一个针对本地 TB 级磁盘文件的输入流。

然后读到数据之后立马写入 HDFS 提供的 FSDataOutputStream 输出流。

这个 FSDataOutputStream 输出流在干啥?

大家觉得他会天真的立马把数据通过网络传输写给 DataNode 吗?

答案当然是否定的了！这么干的话，不就跟之前的那种方式一样了！

1.Chunk 缓冲机制

首先，数据会被写入一个 chunk 缓冲数组，这个 chunk 是一个 512 字节大小的数据片段，你可以这么来理解。

然后这个缓冲数组可以容纳多个 chunk 大小的数据在里面缓冲。

光是这个缓冲，首先就可以让客户端快速的写入数据了，不至于说几百字节就要进行一次网络传输，想一想，是不是这样?

2.Packet 数据包机制

接着，当 chunk 缓冲数组都写满了之后，就会把这个 chunk 缓冲数组进行一下 chunk 切割，切割为一个一个的 chunk，一个 chunk 是一个数据片段。

然后多个 chunk 会直接一次性写入另外一个内存缓冲数据结构，就是 Packet 数据包。

一个 Packet 数据包，设计为可以容纳 127 个 chunk，大小大致为 64mb。所以说大量的 chunk 会不断的写入 Packet 数据包的内存缓冲中。

通过这个 Packet 数据包机制的设计，又可以在内存中容纳大量的数据，进一步避免了频繁的网络传输影响性能。

3.内存队列异步发送机制

当一个 Packet 被塞满了 chunk 之后，就会将这个 Packet 放入一个内存队列来进行排队。

然后有一个 DataStreamer 线程会不断的获取队列中的 Packet 数据包，通过网络传输直接写一个 Packet 数据包给 DataNode。

如果一个 Block 默认是 128mb 的话，那么一个 Block 默认会对应两个 Packet 数据包，每个 Packet 数据包是 64MB。

也就是说，传送两个 Packet 数据包给 DataNode 之后，就会发一个通知说，一个 Block 的数据都传输完毕。

这样 DataNode 就知道自己收到一个 Block 了，里面包含了人家发送过来的两个 Packet 数据包。

四、总结

OK，大家看完了上面的那个图以及 Hadoop 采取的大文件上传机制，是不是感觉设计的很巧妙?

说白了，工业级的大规模分布式系统，都不会采取特别简单的代码和模式，那样性能很低下。

这里都有大量的并发优化、网络 IO 优化、内存优化、磁盘读写优化的架构设计、生产方案在里面。

所以大家观察上面那个图，HDFS 客户端可以快速的将 tb 级大文件的数据读出来，然后快速的交给 HDFS 的输出流写入内存。

基于内存里的 chunk 缓冲机制、packet 数据包机制、内存队列异步发送机制。绝对不会有任何网络传输的卡顿，导致大文件的上传速度变慢。

反而通过上述几种机制，可以上百倍的提升一个 TB 级大文件的上传性能。

看 Hadoop 底层算法如何优雅的将大规模集群性能提升 10 倍以上?

> **作者:中华石杉** [原文地址]

目录

一、前情概要

这篇文章给大家聊聊 Hadoop 在部署了大规模的集群场景下，大量客户端并发写数据的时候，文件契约监控算法的性能优化。

看懂这篇文章需要一些 Hadoop 的基础知识背景，还不太了解的兄弟，可以先看看之前的文章：**兄弟，用大白话告诉你小白都能看懂的Hadoop架构原理**

二、背景引入

先给大家引入一个小的背景，假如多个客户端同时要并发的写 Hadoop HDFS 上的一个文件，大家觉得这个事儿能成吗？

明显不可以接受啊，兄弟们，HDFS 上的文件是不允许并发写的，比如并发的追加一些数据什么的。

所以说，HDFS 里有一个机制，叫做文件契约机制。

也就是说，同一时间只能有一个客户端获取 NameNode 上面一个文件的契约，然后才可以写入数据。此时如果其他客户端尝试获取文件契约的时候，就获取不到，只能干等着。

通过这个机制，就可以保证同一时间只有一个客户端在写一个文件。

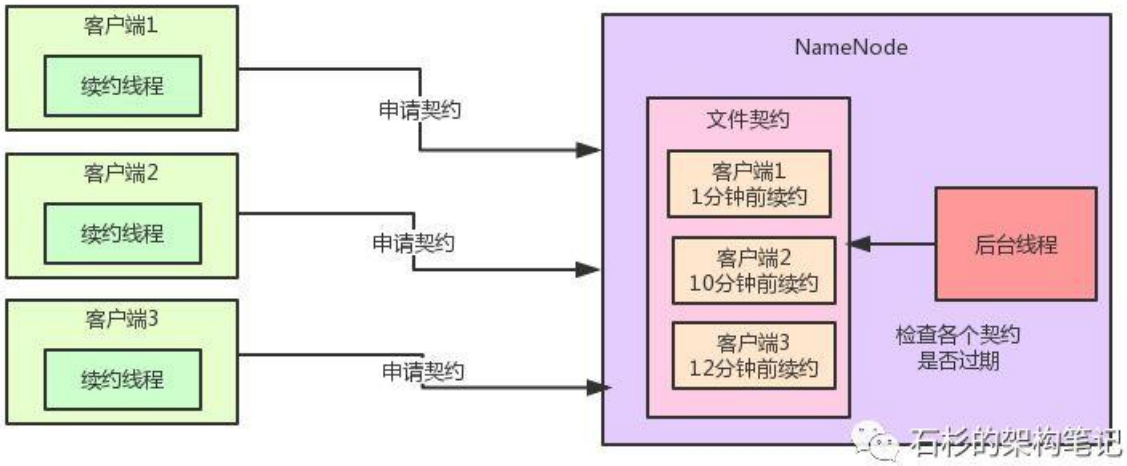
在获取到了文件契约之后，在写文件的过程期间，那个客户端需要开启一个线程，不停的发送请求给 NameNode 进行文件续约，告诉 NameNode：

NameNode 大哥，我还在写文件啊，你给我一直保留那个契约好吗？

而 NameNode 内部有一个专门的后台线程，负责监控各个契约的续约时间。

如果某个契约很长时间没续约了，此时就自动过期掉这个契约，让别的客户端来写。

说了这么多，老规矩，给大家来一张图，直观的感受一下整个过程。



三、问题凸现

好，那么现在问题来了，假如我们有一个大规模部署的 Hadoop 集群，同时存在的客户端可能多达成千上万个。

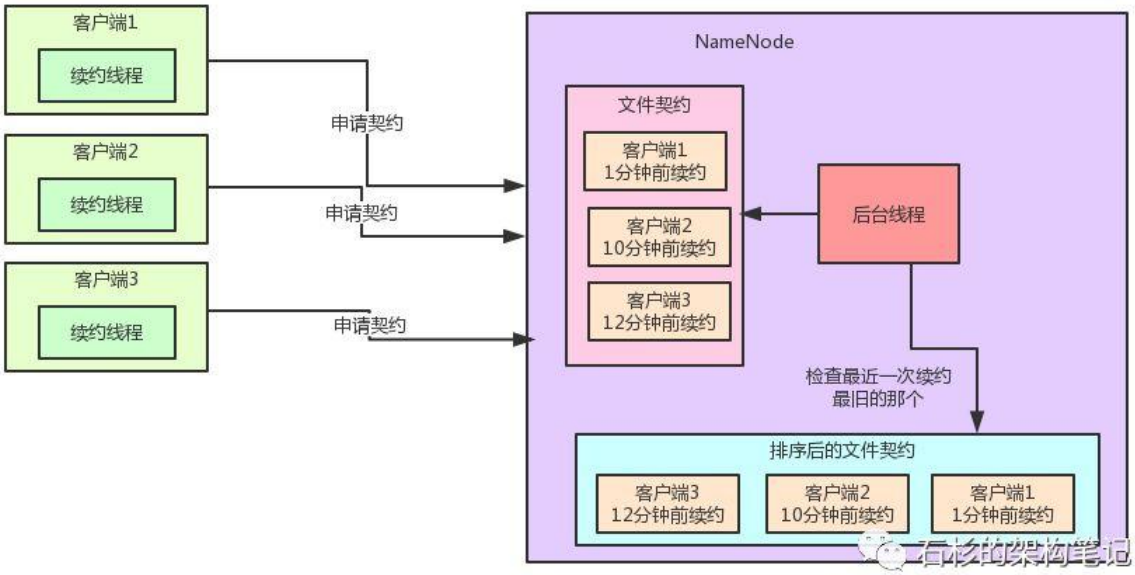
此时 NameNode 内部维护的那个文件契约列表会非常非常的大，而监控契约的后台线程又需要频繁的每隔一段时间就检查一下所有的契约是否过期。

比如，每隔几秒钟就遍历大量的契约，那么势必造成性能不佳，所以说这种契约监控机制明显是不适合大规模部署的 hadoop 集群的。

四、Hadoop 的优化方案

那么 Hadoop 是如何对文件契约监控算法进行优化的呢？咱们来一步一步的看一下他的实现逻辑。

首先，我们一起来看看下面这张手绘图：



其实奥秘十分的简单，每次一个客户端发送续约请求之后，就设置这个契约的最近一次续约时间。

然后，基于一个 TreeSet 数据结构来根据最近一次续约时间对契约进行排序，每次都把续约时间最老的契约排在最前头，这个排序后的契约数据结构十分的重要。

TreeSet 是一种可排序的数据结构，他底层基于 TreeMap 来实现。

TreeMap 底层则基于红黑树来实现，可以保证元素没有重复，同时还能按照我们自己定义的排序规则在你每次插入一个元素的时候来进行自定义的排序。

所以这里我们的排序规则：就是按照契约的最近一次续约时间来排序。

其实这个优化就是如此的简单，就是维护这么一个排序数据结构而已。

我们现在来看一下 Hadoop 中的契约监控的源码实现：

```
Lease leaseToCheck = null;
try {
    leaseToCheck = sortedLeases.first();
} catch(NoSuchElementException e) {}

while(leaseToCheck != null) {
    if (!leaseToCheck.expiredHardLimit()) {
        break;
    }
}
```

每次检查契约是否过期的时候，你不要遍历成千上万的契约，那样遍历效率当然会很低下。

我们完全可以就从 TreeSet 中获取续约时间最老的那个契约，假如说连最近一次续约时间最老的那个契约都还没过期，那么就不用继续检查了啊！这说明续约时间更近的那些契约绝对不会过期！

举个例子：续约时间最老的那个契约，最近一次续约的时间是 10 分钟以前，但是我们判断契约过期的限制是超过 15 分钟不续约就过期那个契约。

这个时候，连 10 分钟以前续约的契约都没有过期，那么那些 8 分钟以前，5 分钟以前续约的契约，肯定也不会过期啊！

这个机制的优化对性能的提升是相当有帮助的，因为正常来说，过期的契约肯定还是占少数，所以压根儿不用每次都遍历所有的契约来检查是否过期。

我们只需要检查续约时间最旧的那几个契约就可以了，如果一个契约过期了，那么就删掉那个契约，然后再检查第二旧的契约好了。以此类推。

通过这个 TreeSet 排序 + 优先检查最旧契约的机制，有效的将大规模集群下的契约监控机制的性能提升至少 10 倍以上，这种思想是非常值得我们学习和借鉴的。

给大家稍微引申一下，在 Spring Cloud 微服务架构中，Eureka 作为注册中心其实也有续约检查的机制，跟 Hadoop 是类似的。

如果想了解 Eureka 注册中心相关技术的朋友，建议看一下：**拜托！面试请不要再问我Spring Cloud底层原理**

但是在 Eureka 中就没有实现类似的续约优化机制，而是暴力的每一轮都遍历所有的服务实例的续约时间。

如果你面对的是一个大规模部署的微服务系统呢，情况就不妙了！

部署了几十万台机器的大规模系统，有几十万个服务实例的续约信息驻留在 Eureka 的内存中，难道每隔几秒钟都要遍历几十万个服务实例的续约信息吗？

最后给大家提一句，优秀的开源项目，蕴含着很多优秀的设计思想。多看各种优秀开源项目的源码，是短时间内快速、大幅度提升一个人的技术功底和技术水平的方式，大家不妨尝试一下。

【面试题】为什么使用消息队列？消息队列有什么优点和缺点？Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么优点和缺点？

- 为什么使用消息队列?
- 消息队列有什么优点和缺点?
- Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别，以及适合哪些场景?

面试官心理分析

其实面试官主要是想看看：

- **第一**，你知不知道你们系统里为什么要用消息队列这个东西?
  不少候选人，说自己项目里用了 Redis、MQ，但是其实他并不知道自己为什么要用这个东西。其实说白了，就是为了用而用，或者是别人设计的架构，他从头到尾都没思考过。
  没有对自己的架构问过为什么的人，一定是平时没有思考的人，面试官对这类候选人印象通常很不好。因为面试官担心你进了团队之后只会木头木脑的干呆活儿，不会自己思考。

- **第二**，你既然用了消息队列这个东西，你知不知道用了有什么好处&坏处?
  你要是没考虑过这个，那你盲目弄个 MQ 进系统里，后面出了问题你是不是就自己溜了给公司留坑？你要是没考虑过引入一个技术可能存在的弊端和风险，面试官把这类候选人招进来了，基本可能就是挖坑型选手。就怕你干 1 年挖一堆坑，自己跳槽了，给公司留下无穷后患。

- **第三**，既然你用了 MQ，可能是某一种 MQ，那么你当时做没做过调研?
  你别傻乎乎的自己拍脑袋看个人喜好就瞎用了一个 MQ，比如 Kafka，甚至都从没调研过业界流行的 MQ 到底有哪几种。每一个 MQ 的优点和缺点是什么。每一个 MQ **没有绝对的好坏**，但是就是看用在哪个场景可以**扬长避短，利用其优势，规避其劣势**。
  如果是一个不考虑技术选型的候选人招进了团队，leader 交给他一个任务，去设计个什么系统，他在里面用一些技术，可能都没考虑过选型，最后选的技术可能并不一定合适，一样是留坑。

面试题剖析

为什么使用消息队列

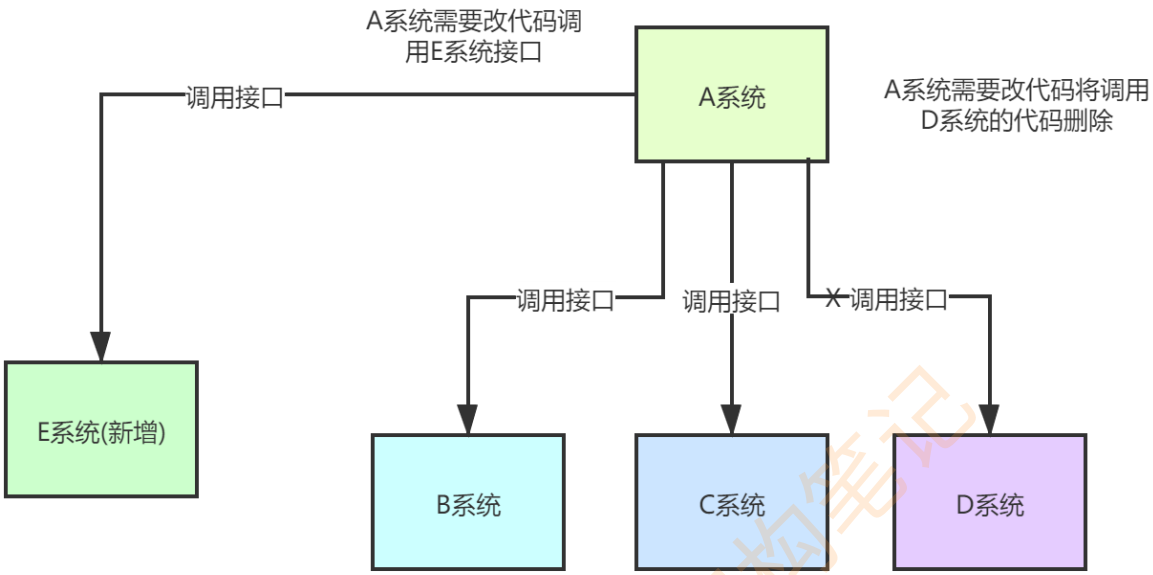
其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么?

面试官问你这个问题，**期望的一个回答**是说，你们公司有个什么**业务场景**，这个业务场景有个什么技术挑战，如果不用 MQ 可能会很麻烦，但是你现在用了 MQ 之后带给了你很多的好处。

先说一下消息队列常见的使用场景吧，其实场景有很多，但是比较核心的有 3 个：**解耦**、**异步**、**削峰**。

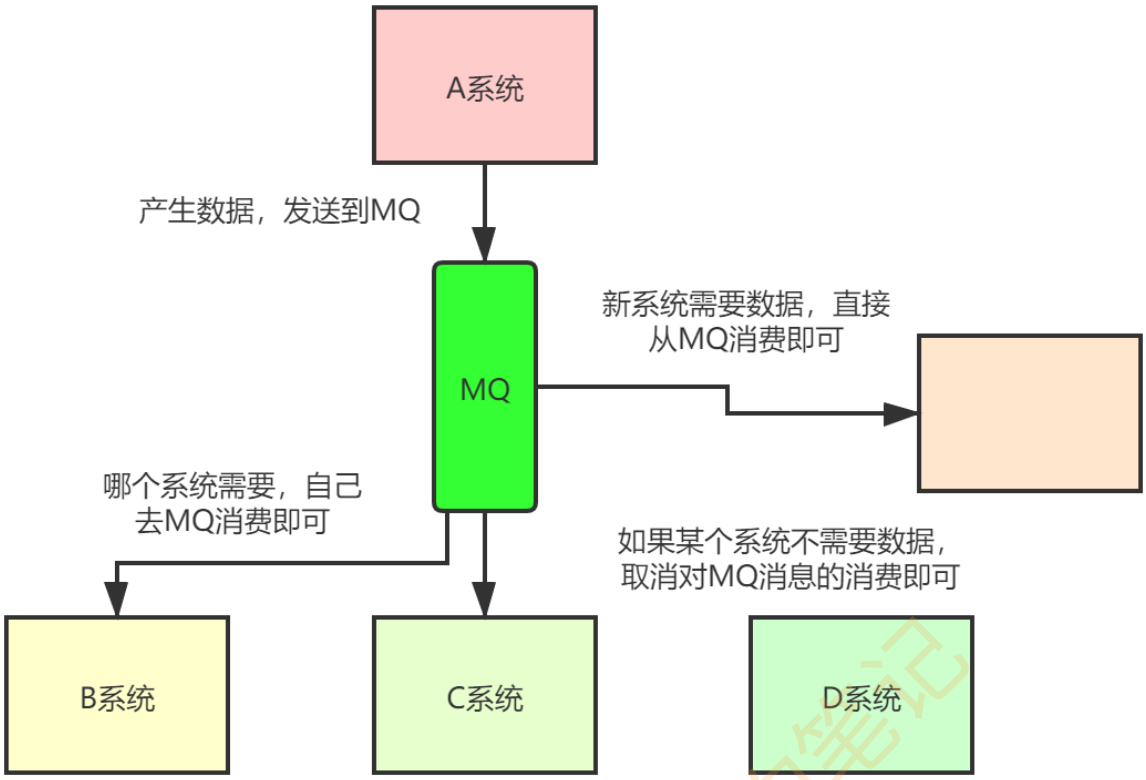
解耦

看这么个场景。A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃……



在这个场景中，A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？头发都白了啊！

如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

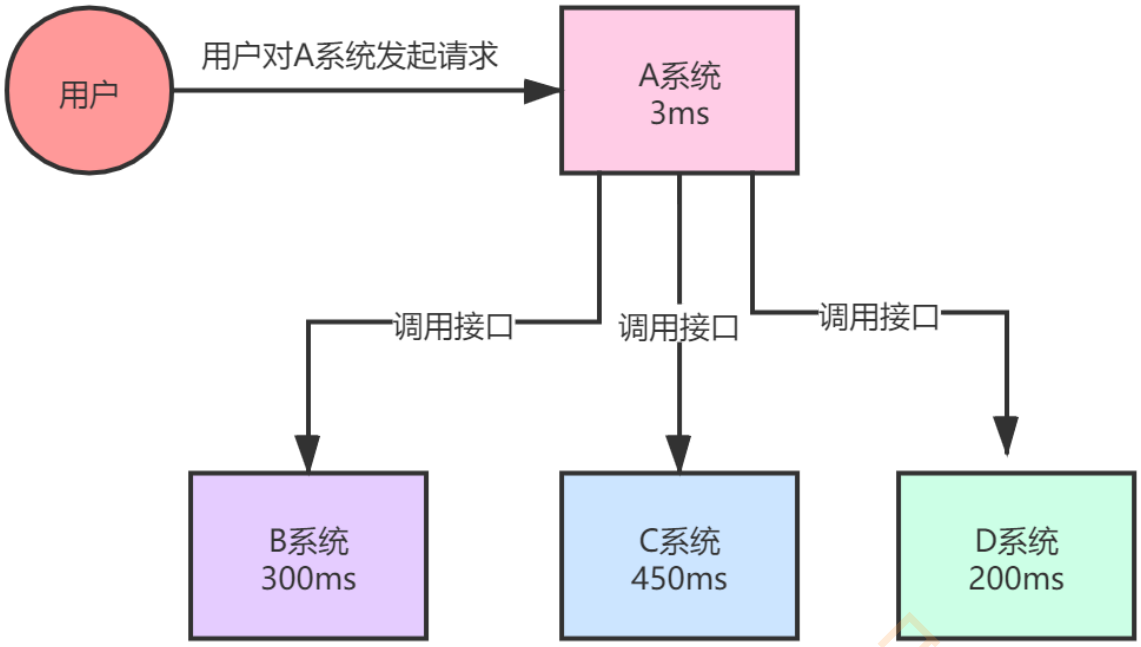


总结：通过一个 MQ，Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。

面试技巧：你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个 MQ 去进行系统的解耦。在简历中体现出来这块东西，用 MQ 作解耦。

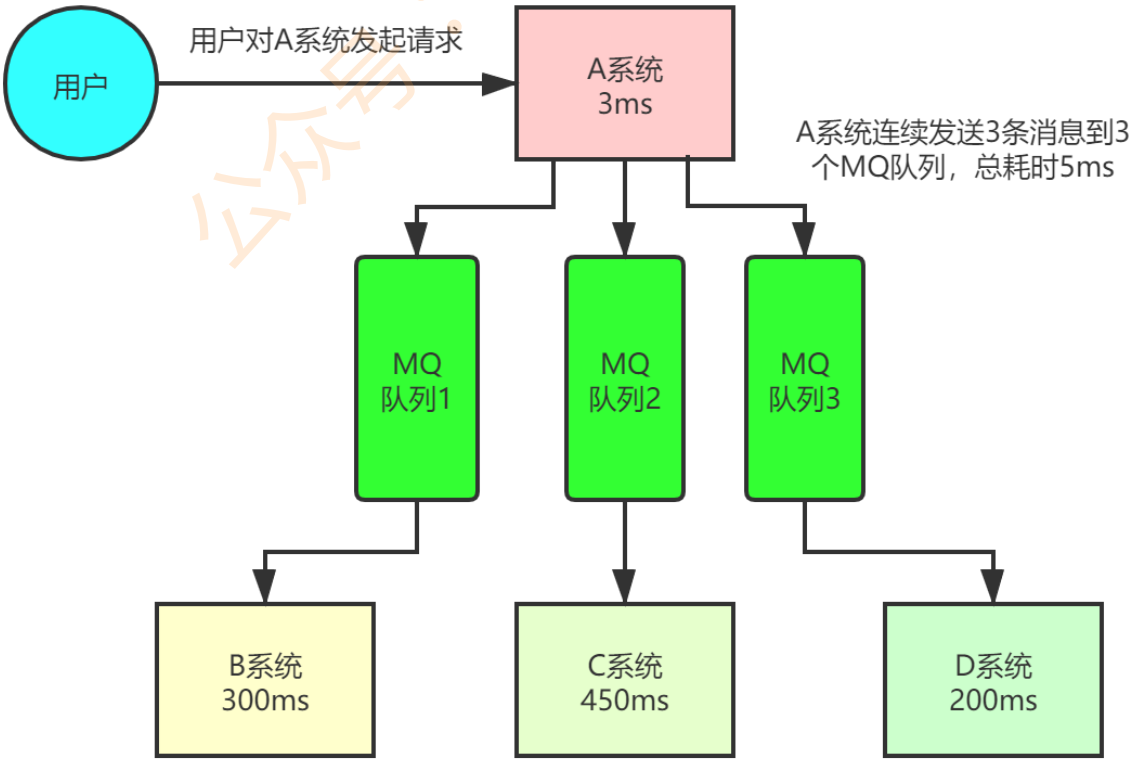
异步

再来看一个场景，A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求，等待个 1s，这几乎是不可接受的。



一般互联网类的企业，对于用户直接的操作，一般要求是每个请求都必须在 200 ms 以内完成，对用户几乎是无感知的。

如果**使用 MQ**，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了，爽！网站做得真好，真快！

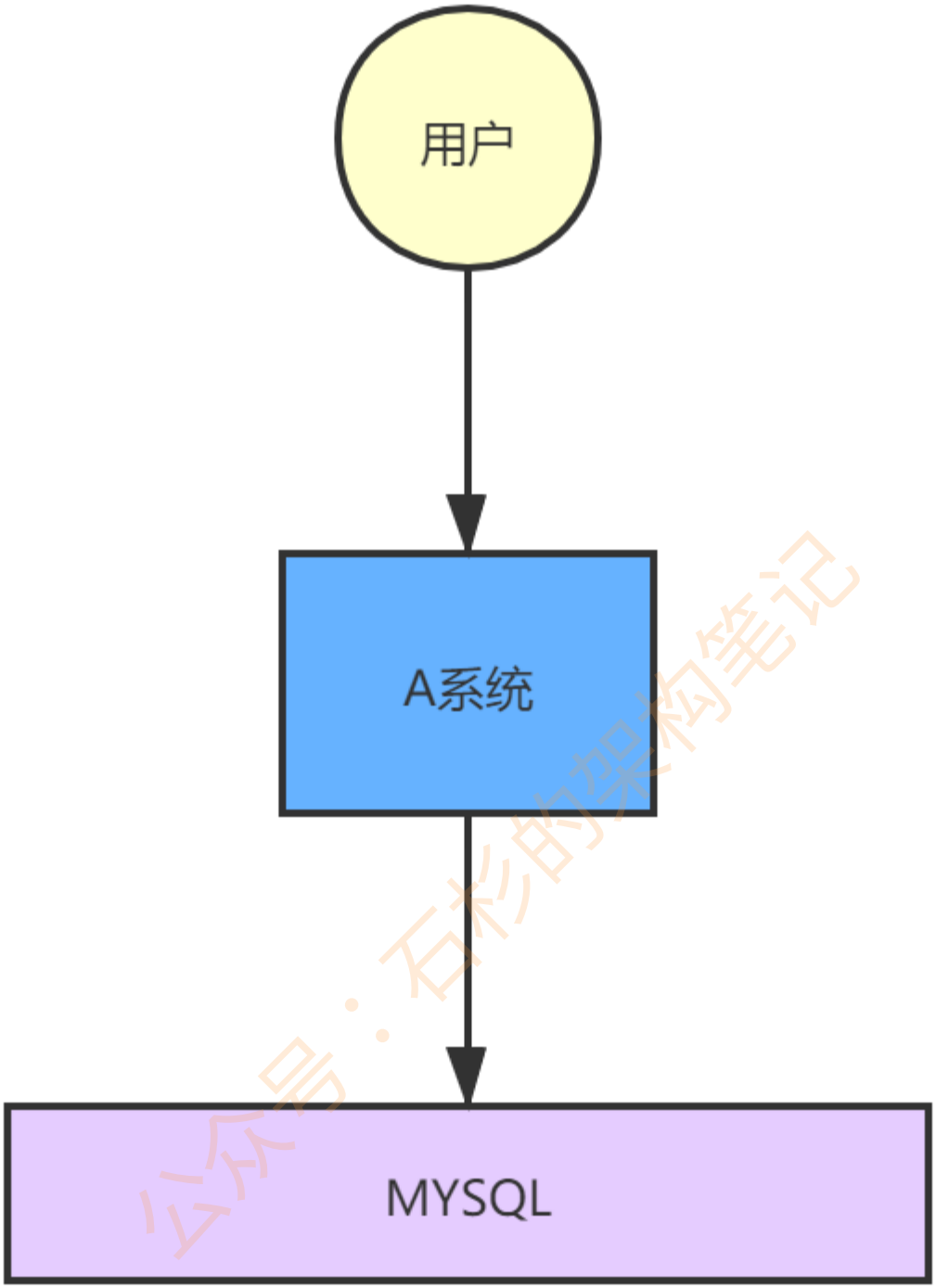


削峰

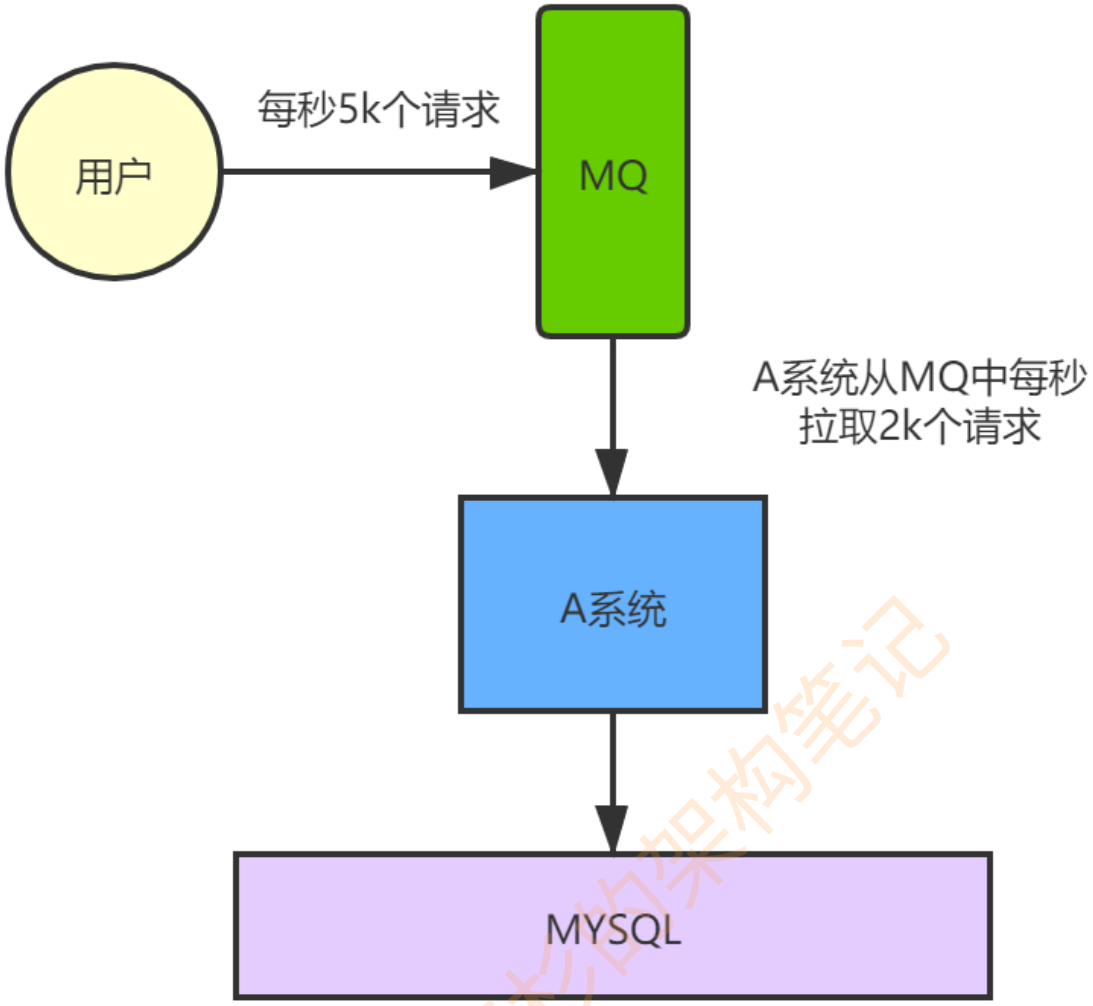
每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00 ~ 13:00，每秒并发请求数量突然会暴增到 5k+ 条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。

一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。



如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。



这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。

消息队列有什么优缺点

优点上面已经说了，就是**在特殊场景下有其对应的好处，解耦、异步、削峰**。

缺点有以下几个：

- 系统可用性降低
  系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？如何保证消息队列的高可用，可以**点击这里查看**。

- 系统复杂度提高
  硬生生加个 MQ 进来，你怎么**保证消息没有重复消费**？怎么**处理消息丢失的情况**？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

- 一致性问题

  A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，做好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的。

Kafka、ActiveMQ、RabbitMQ、RocketMQ 有什么优缺点?

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

		强，性能极好，延时很低		

综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出品，但社区可能有突然黄掉的风险（目前 RocketMQ 已捐给 **Apache**，但 GitHub 上的活跃度其实不算高）对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝对不会黄。

所以**中小型公司**，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择；**大型公司**，基础架构研发实力较强，用 RocketMQ 是很好的选择。

如果是**大数据领域**的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

【面试题】 - 如何保证消息队列的高可用?

面试官心理分析

如果有人问到你 MQ 的知识，**高可用是必问的**。**上一讲**提到，MQ 会导致**系统可用性降低**。所以只要你用了 MQ，接下来问的一些要点肯定就是围绕着 MQ 的那些缺点怎么来解决了。

要是你傻乎乎的就干用了一个 MQ，各种问题从来没考虑过，那你就杯具了，面试官对你的感觉就是，只会简单使用一些技术，没任何思考，马上对你的印象就不太好了。这样的同学招进来要是做个 20k 薪资以内的普通小弟还凑合，要是做薪资 20k+ 的高工，那就惨了，让你设计个系统，里面肯定一堆坑，出了事故公司受损失，团队一起背锅。

面试题剖析

这个问题这么问是很好的，因为不能问你 Kafka 的高可用性怎么保证？ActiveMQ 的高可用性怎么保证？一个面试官要是这么问就显得很没水平，人家可能用的就是 RabbitMQ，没用过 Kafka，你上来问人家 Kafka 干什么？这不是摆明了刁难人么。

所以有水平的面试官，问的是 MQ 的高可用性怎么保证？这样就是你用过哪个 MQ，你就说说你对那个 MQ 的高可用性的理解。

RabbitMQ 的高可用性

RabbitMQ 是比较有代表性的，因为是**基于主从**（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。

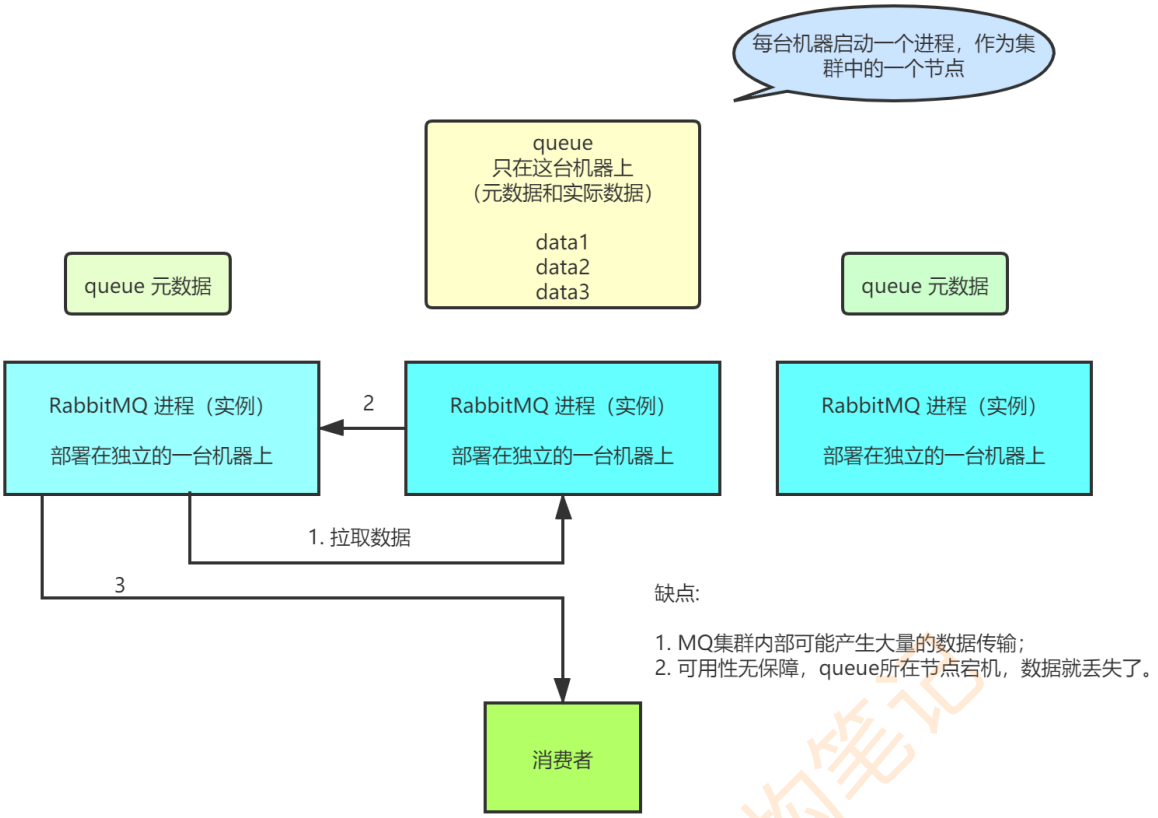
RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的😄，没人生产用单机模式。

普通集群模式（无高可用性）

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你**创建的 queue，只会放在一个 RabbitMQ 实例上**，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。



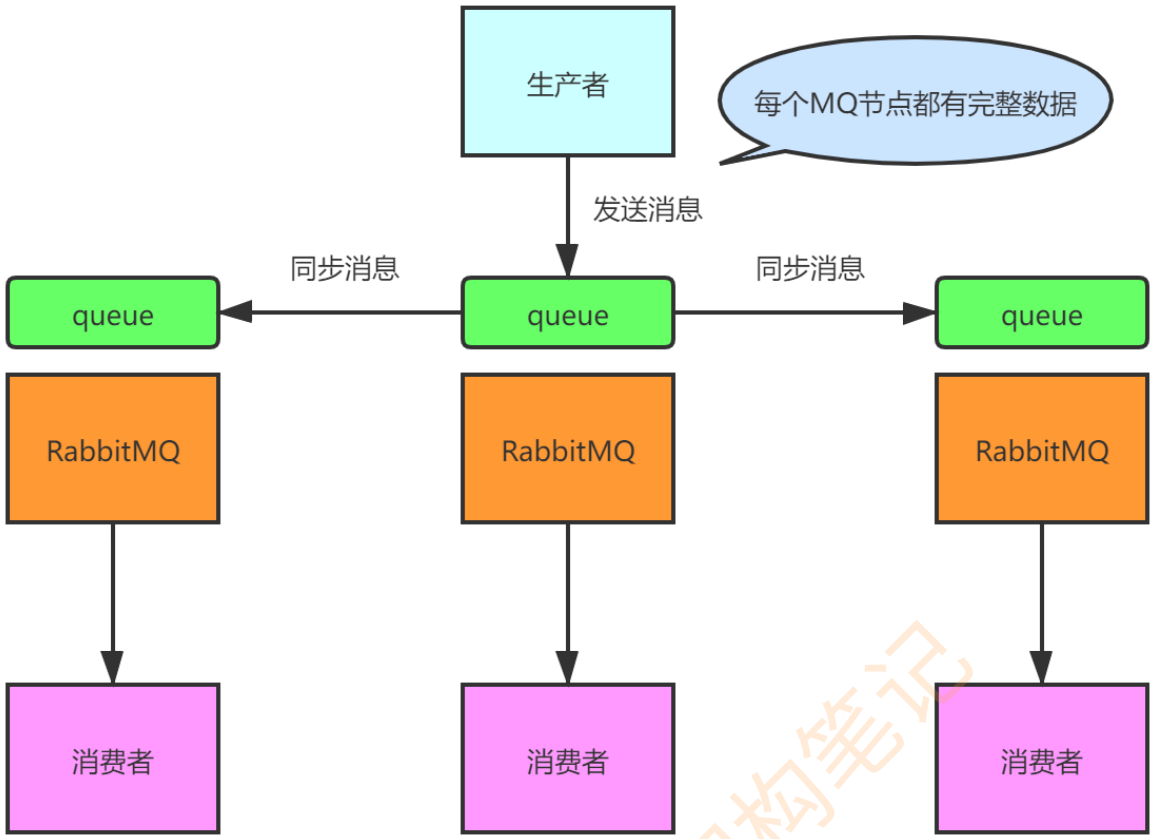
这种方式确实很麻烦，也不怎么好，**没做到所谓的分布式**，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个 queue 所在实例消费数据，前者有**数据拉取的开销**，后者导致**单实例性能瓶颈**。

而且如果那个放 queue 的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你**开启了消息持久化**，让 RabbitMQ 落地存储消息的话，**消息不一定会丢**，得等这个实例恢复了，然后才可以继续从这个 queue 拉取数据。

所以这个事儿就比较尴尬了，这就**没有什么所谓的高可用性**，**这方案主要是提高吞吐量的**，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式（高可用性）

这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会**存在于多个实例上**，就是说，每个 RabbitMQ 节点都有这个 queue 的一个**完整镜像**，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把**消息同步**到多个实例的 queue 上。



那么**如何开启这个镜像集群模式**呢？其实很简单，RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是**镜像集群模式的策略**，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！第二，这么玩儿，不是分布式的，就**没有扩展性可言**了，如果某个 queue 负载很重，你加机器，新增的机器也包含了这个 queue 的所有数据，并**没有办法线性扩展**你的 queue。你想，如果这个 queue 的数据量很大，大到这个机器上的容量无法容纳了，此时该怎么办呢？

Kafka 的高可用性

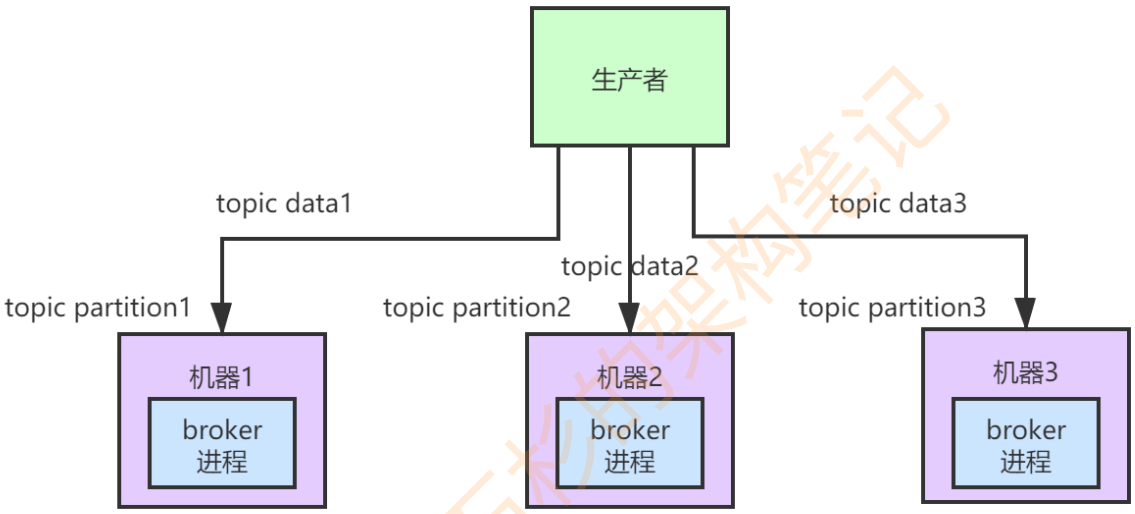
Kafka 一个最基本的架构认识：由多个 broker 组成，每个 broker 是一个节点；你创建一个 topic，这个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据。

这就是**天然的分布式消息队列**，就是说一个 topic 的数据，是**分散放在多个机器上的，每个机器就放一部分数据**。

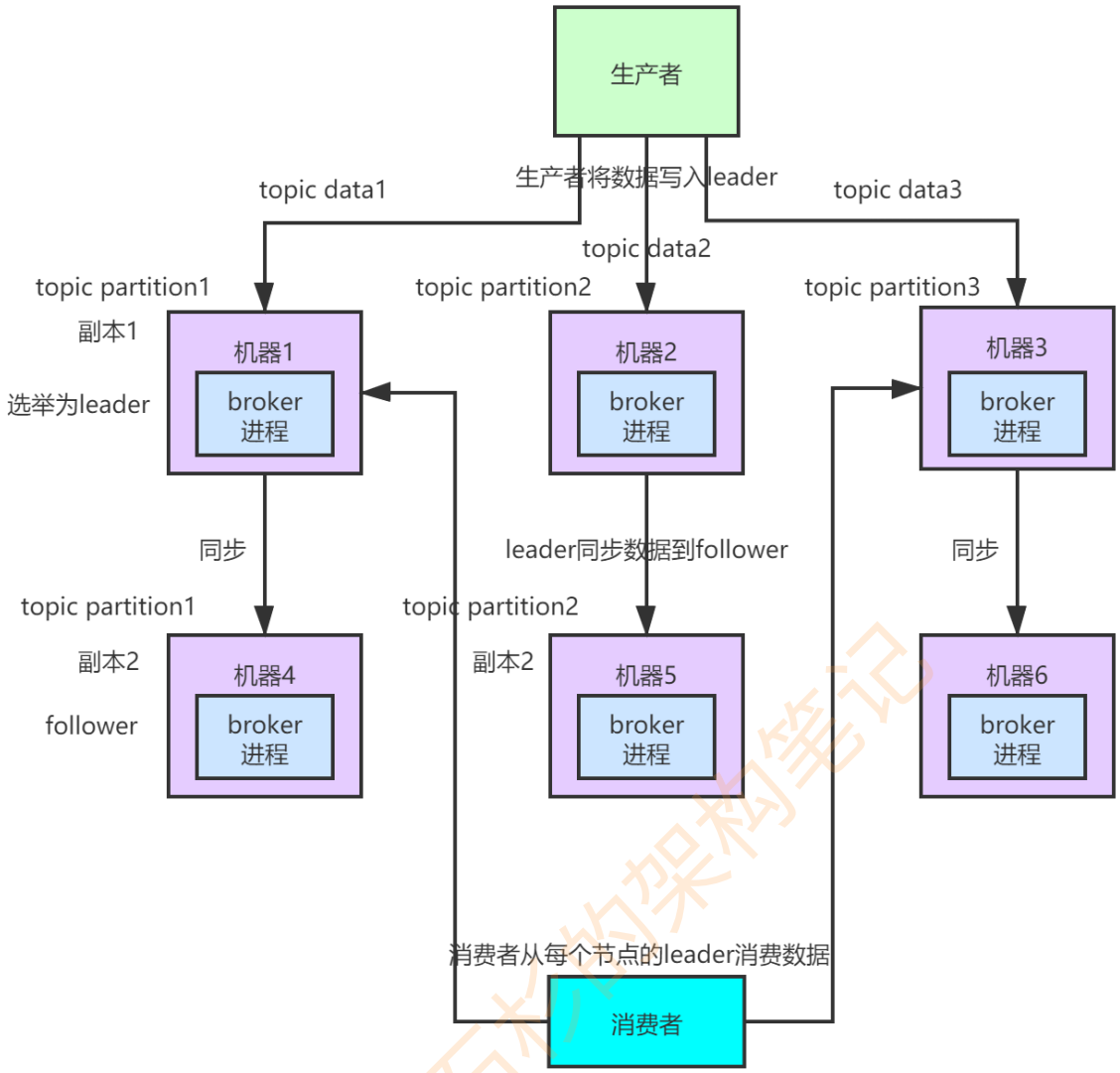
实际上 RabbmitMQ 之类的，并不是分布式消息队列，它就是传统的消息队列，只不过提供了一些集群、HA(High Availability, 高可用性) 的机制而已，因为无论怎么玩儿，RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

Kafka 0.8 以前，是没有 HA 机制的，就是任何一个 broker 宕机了，那个 broker 上的 partition 就废了，没法写也没法读，没有什么高可用性可言。

比如说，我们假设创建了一个 topic，指定其 partition 数量是 3 个，分别在三台机器上。但是，如果第二台机器宕机了，会导致这个 topic 的 1/3 的数据就丢了，因此这个是做不到高可用的。



Kafka 0.8 以后，提供了 HA 机制，就是 replica（复制品） 副本机制。每个 partition 的数据都会同步到其它机器上，形成自己的多个 replica 副本。所有 replica 会选举一个 leader 出来，那么生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上的数据即可。只能读写 leader? 很简单，**要是你可以随意读写每个 follower，那么就要 care 数据一致性的问题**，系统复杂度太高，很容易出问题。Kafka 会均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。



这么搞，就有所谓的**高可用性**了，因为如果某个 broker 宕机了，没事儿，那个 broker上面的 partition 在其他机器上都有副本的。如果这个宕机的 broker 上面有某个 partition 的 leader，那么此时会从 follower 中**重新选举**一个新的 leader 出来，大家继续读写那个新的 leader 即可。这就有所谓的高可用性了。

写数据的时候，生产者就写 leader，然后 leader 将数据落地写本地磁盘，接着其他 follower 自己主动从 leader 来 pull 数据。一旦所有 follower 同步好数据了，就会发送 ack 给 leader，leader 收到所有 follower 的 ack 之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）

消费的时候，只会从 leader 去读，但是只有当一个消息已经被所有 follower 都同步成功返回 ack 的时候，这个消息才会被消费者读到。

看到这里，相信你大致明白了 Kafka 是如何保证高可用机制的了，对吧？不至于一无所知，现场还能给面试官画画图。要是遇上面试官确实是 Kafka 高手，深挖了问，那你只能说不好意思，太深入的你没研究过。

【面试题】- 如何保证消息不被重复消费？或者说，如何保证消息消费的幂等性?

面试官心理分析

其实这是很常见的一个问题，这俩问题基本可以连起来问。既然是消费消息，那肯定要考虑会不会重复消费？能不能避免重复消费？或者重复消费了也别造成系统异常可以吗？这个是 MQ 领域的基本问题，其实本质上还是问你**使用消息队列如何保证幂等性**，这个是你架构里要考虑的一个问题。

面试题剖析

回答这个问题，首先你别听到重复消息这个事儿，就一无所知吧，你**先大概说一说可能会有哪些重复消费的问题**。

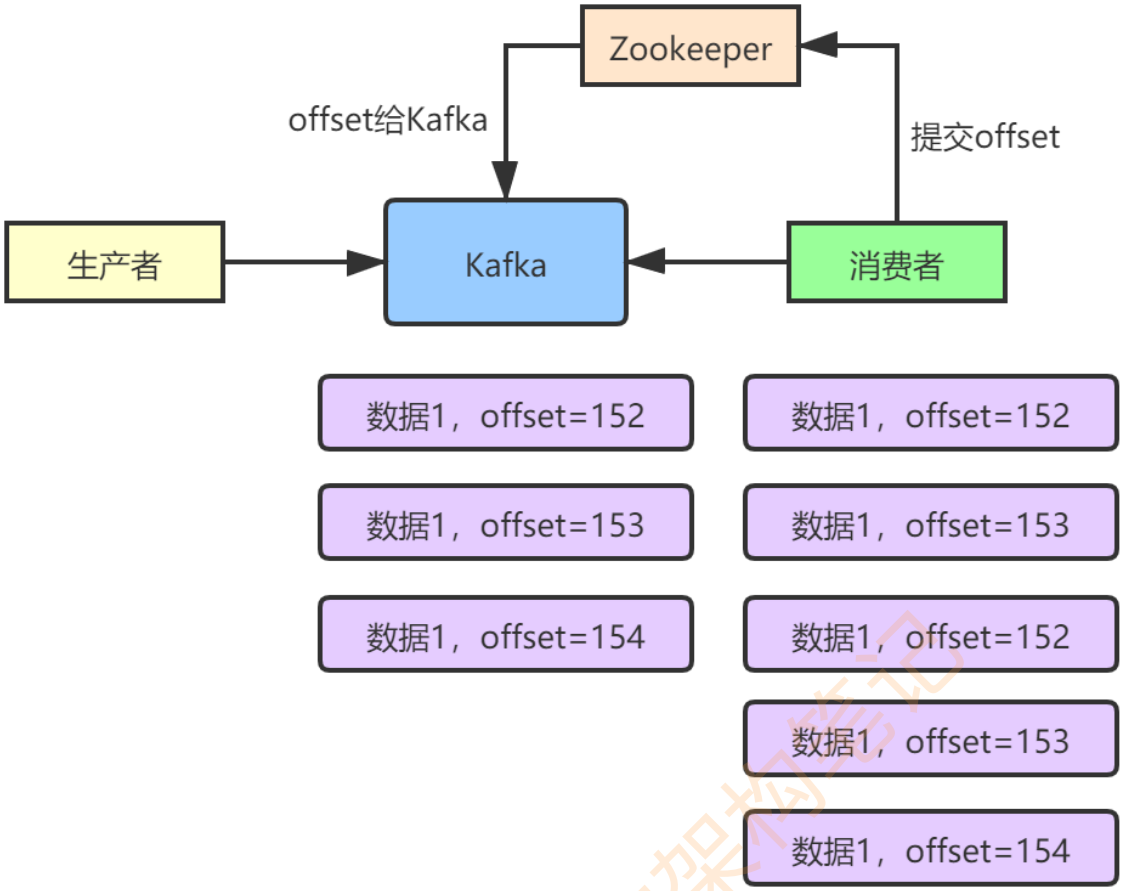
首先，比如 RabbitMQ、RocketMQ、Kafka，都有可能会出现消息重复消费的问题，正常。因为这问题通常不是 MQ 自己保证的，是由我们开发来保证的。挑一个 Kafka 来举个例子，说说怎么重复消费吧。

Kafka 实际上有个 offset 的概念，就是每个消息写进去，都有一个 offset，代表消息的序号，然后 consumer 消费了数据之后，**每隔一段时间**（定时定期），会把自己消费过的消息的 offset 提交一下，表示“我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的 offset 来继续消费吧”。

但是凡事总有意外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接 kill 进程了，再重启。这会导致 consumer 有些消息处理了，但是没来得及提交 offset，尴尬了。重启之后，少数消息会再次消费一次。

举个栗子。

有这么个场景。数据 1/2/3 依次进入 kafka，kafka 会给这三条数据每条分配一个 offset，代表这条数据的序号，我们就假设分配的 offset 依次是 152/153/154。消费者从 kafka 去消费的时候，也是按照这个顺序去消费。假如当消费者消费了 `offset=153` 的这条数据，刚准备去提交 offset 到 zookeeper，此时消费者进程被重启了。那么此时消费过的数据 1/2 的 offset 并没有提交，kafka 也就不知道你已经消费了 `offset=153` 这条数据。那么重启之后，消费者会找 kafka 说，嘿，哥儿们，你给我接着把上次我消费到的那个地方后面的数据继续给我传递过来。由于之前的 offset 没有提交成功，那么数据 1/2 会再次传过来，如果此时消费者没有去重的话，那么就会导致重复消费。



如果消费者干的事儿是拿一条数据就往数据库里写一条，会导致说，你可能就把数据 1/2 在数据库里插入了 2 次，那么数据就错啦。

其实重复消费不可怕，可怕的是你没考虑到重复消费之后，**怎么保证幂等性**。

举个例子吧。假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。

一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性。

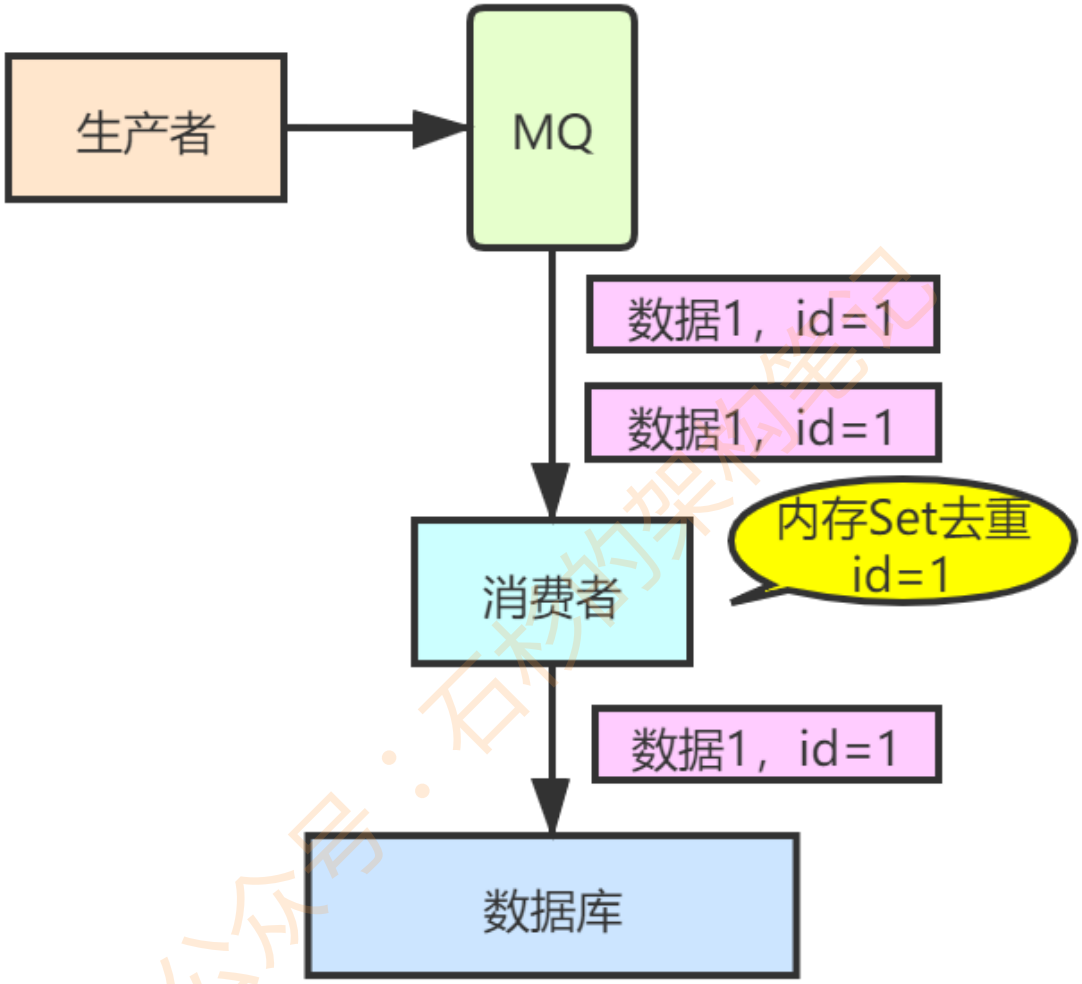
幂等性，通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，**不能出错**。

所以第二个问题来了，怎么保证消息队列消费的幂等性?

其实还是得结合业务来思考，我这里给几个思路：

- 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。
- 比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。

- 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。
- 比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。



当然，如何保证 MQ 的消费是幂等性的，需要结合具体的业务来看。

【面试题】 - 如何保证消息的可靠性传输？或者说，如何处理消息丢失的问题？

面试官心理分析

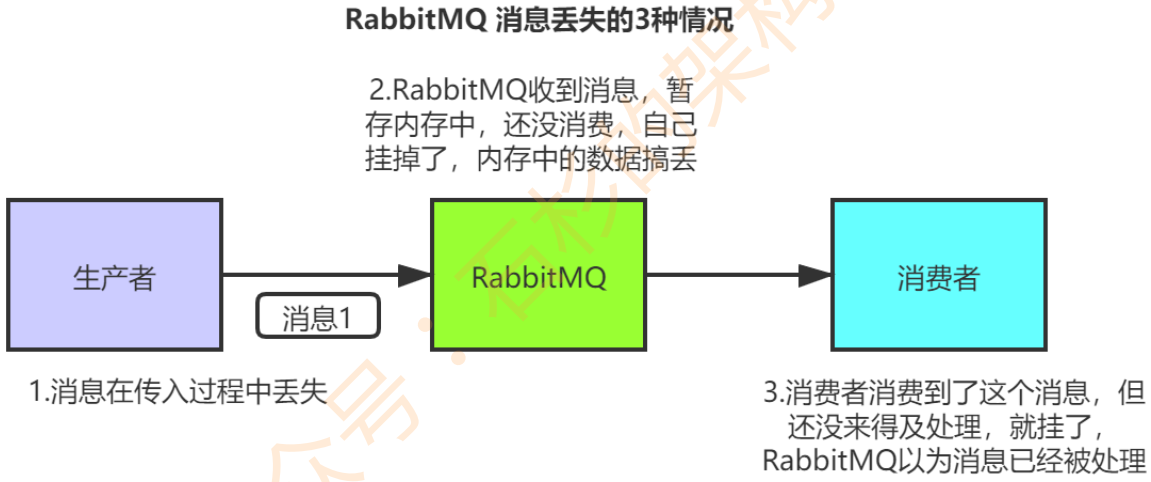
这个是肯定的，用 MQ 有个基本原则，就是**数据不能多一条，也不能少一条**，不能多，就是前面说的**重复消费和幂等性问题**。不能少，就是说这数据别搞丢了。那这个问题你必须得考虑一下。

如果说你这个是用 MQ 来传递非常核心的消息，比如说计费、扣费的一些消息，那必须确保这个 MQ 传递过程中**绝对不会把计费消息给弄丢**。

面试题剖析

数据的丢失问题，可能出现在生产者、MQ、消费者中，咱们从 RabbitMQ 和 Kafka 分别来分析一下吧。

RabbitMQ



生产者弄丢了数据

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者**发送数据之前**开启 RabbitMQ 事务 `channel.txSelect` ，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 `channel.txRollback` ，然后重试发送消息；如果收到了消息，那么可以提交事务 `channel.txCommit` 。

```java
// 开启事务
channel.txSelect
```

```
try {
    // 这里发送消息
} catch (Exception e) {
    channel.txRollback

    // 这里再次重发这条消息
}

// 提交事务
channel.txCommit
```

但是问题是，RabbitMQ 事务机制（同步）一搞，基本上**吞吐量会下来，因为太耗性能**。

所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 `confirm` 模式，在生产者那里设置开启 `confirm` 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 `ack` 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 `nack` 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 `confirm` 机制最大的不同在于，**事务机制是同步的**，你提交一个事务之后会**阻塞**在那儿，但是 `confirm` 机制是**异步**的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你的一个接口通知你这个消息接收到了。

所以一般在生产者这块**避免数据丢失**，都是用 `confirm` 机制的。

RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须**开启 RabbitMQ 的持久化**，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，**恢复之后会自动读取之前存储的数据**，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，**可能导致少量数据丢失**，但是这个概率较小。

设置持久化有**两个步骤**：

- 创建 queue 的时候将其设置为持久化
  这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是它是不会持久化 queue 里的数据的。
- 第二个是发送消息的时候将消息的 `deliveryMode` 设置为 2
  就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

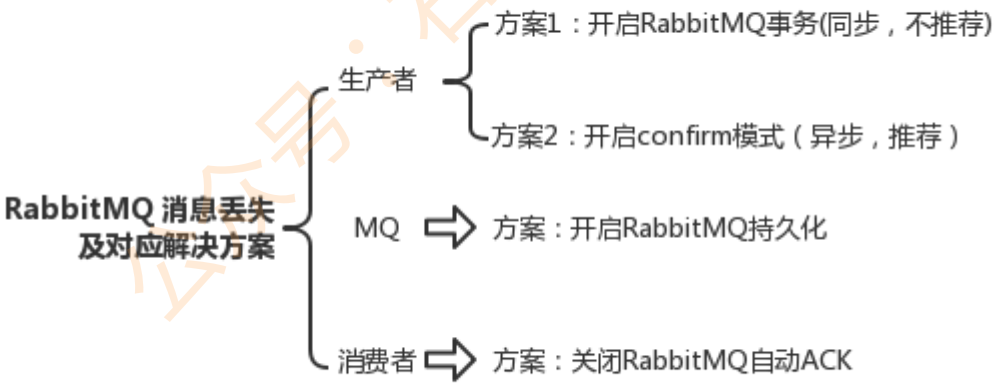
注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失。

所以，持久化可以跟生产者那边的 `confirm` 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 `ack` 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 `ack` ，你也是可以自己重发的。

消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，**刚消费到，还没处理，结果进程挂了**，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 `ack` 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 `ack` ，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack` 了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



Kafka

消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你消费到了这个消息，然后消费者那边**自动提交了 offset**，让 Kafka 以为你已经消费好了这个消息，但其实你才刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

这不是跟 RabbitMQ 差不多吗，大家都知道 Kafka 会自动提交 offset，那么只要**关闭自动提交** offset，在处理完之后自己手动提交 offset，就可以保证数据不会丢。但是此时确实还是**可能会有重复消费**，比如你刚处理完，还没提交 offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的 Kafka 消费者消费到了数据之后是写到一个内存的 queue 里先缓冲一下，结果有的时候，你刚把消息写入内存 queue，然后消费者会自动提交 offset。然后此时我们重启了系统，就会导致内存 queue 里还没来得及处理的数据就丢失了。

Kafka 弄丢了数据

这块比较常见的一个场景，就是 Kafka 某个 broker 宕机，然后重新选举 partition 的 leader。大家想想，要是此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后选举某个 follower 成 leader 之后，不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前 Kafka 的 leader 机器宕机了，将 follower 切换为 leader 之后，就会发现说这个数据就丢了。

所以此时一般是要求起码设置如下 4 个参数：

- 给 topic 设置 `replication.factor` 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。
- 在 Kafka 服务端设置 `min.insync.replicas` 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。
- 在 producer 端设置 `acks=all` ：这个是要求每条数据，必须是**写入所有 replica 之后，才能认为是写成功了**。
- 在 producer 端设置 `retries=MAX` （很大很大很大的一个值，无限次重试的意思）：这个是**要求一旦写入失败，就无限重试**，卡在这里了。

我们生产环境就是按照上述要求配置的，这样配置之后，至少在 Kafka broker 端就可以保证在 leader 所在 broker 发生故障，进行 leader 切换时，数据不会丢失。

生产者会不会弄丢数据?

如果按照上述的思路设置了 `acks=all` ，一定不会丢，要求是，你的 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

【面试题】 - 如何保证消息的顺序性?

面试官心理分析

其实这个也是用 MQ 的时候必问的话题，第一看看你了不了解顺序这个事儿？第二看看你有没有办法保证消息是有顺序的？这是生产系统中常见的问题。

面试题剖析

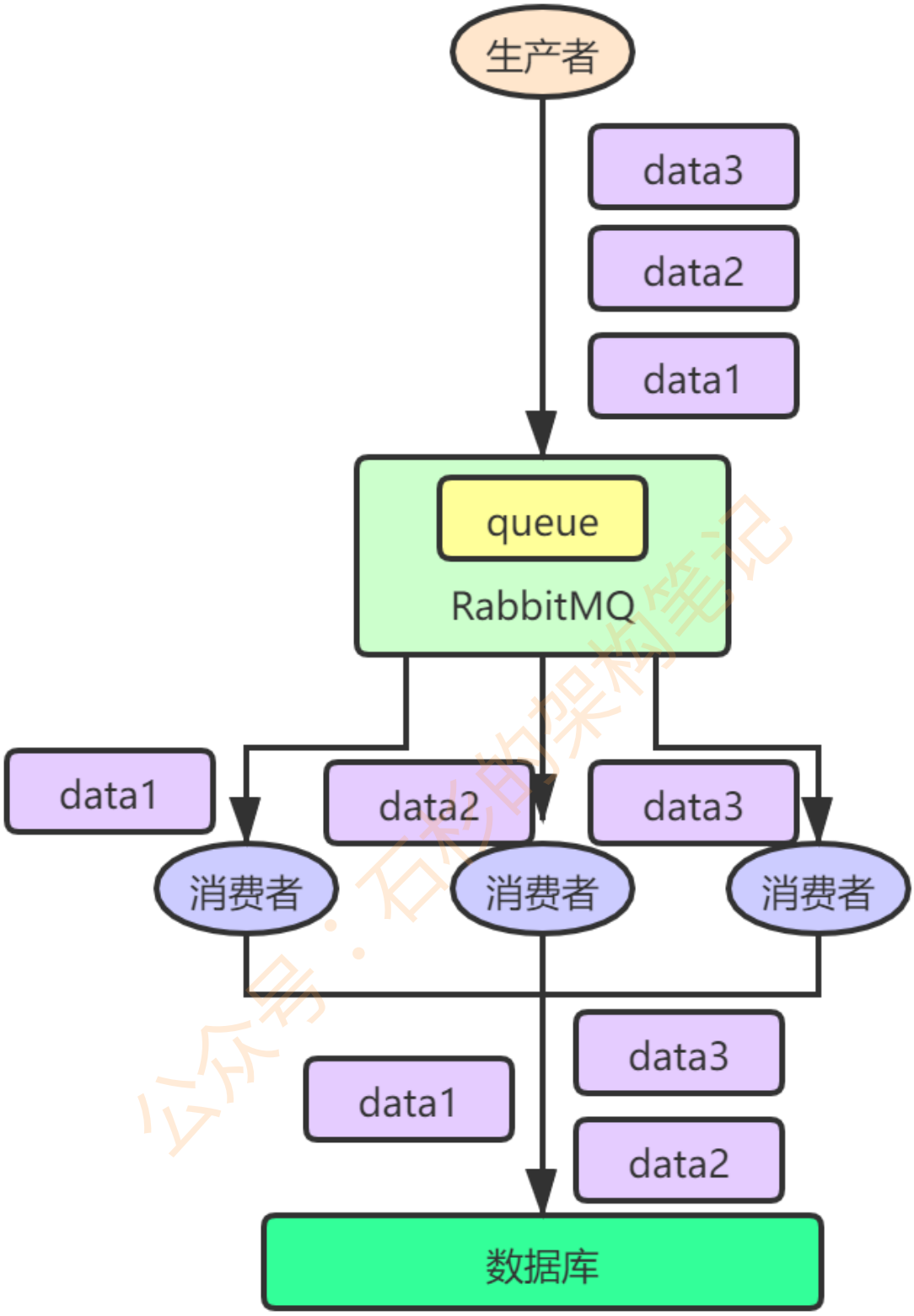
我举个例子，我们以前做过一个 mysql `binlog` 同步的系统，压力还是非常大的，日同步数据要达到上亿，就是说数据从一个 mysql 库原封不动地同步到另一个 mysql 库里面去（mysql -> mysql）。常见的一点在于说比如大数据 team，就需要同步一个 mysql 库过来，对公司的业务系统的数据做各种复杂的操作。

你在 mysql 里增删改一条数据，对应出来了增删改 3 条 `binlog` 日志，接着这三条 `binlog` 发送到 MQ 里面，再消费出来依次执行，起码得保证人家是按照顺序来的吧？不然本来是：增加、修改、删除；你楞是换了顺序给执行成删除、修改、增加，不全错了么。

本来这个数据同步过来，应该最后这个数据被删除了；结果你搞错了这个顺序，最后这个数据保留下来了，数据同步就出错了。

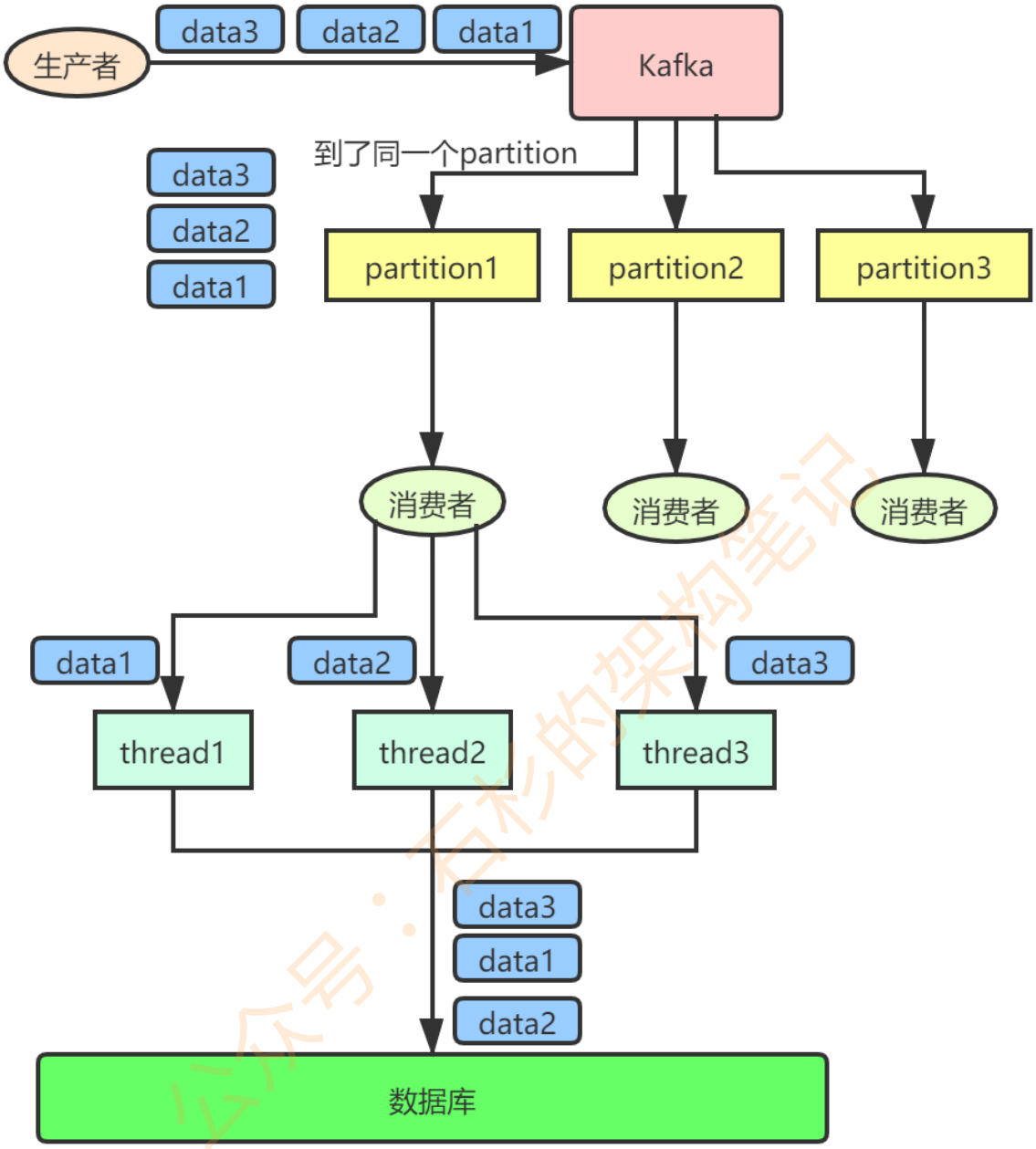
先看看顺序会错乱的俩场景：

- **RabbitMQ**：一个 queue，多个 consumer。比如，生产者向 RabbitMQ 里发送了三条数据，顺序依次是 data1/data2/data3，压入的是 RabbitMQ 的一个内存队列。有三个消费者分别从 MQ 中消费这三条数据中的一条，结果消费者2先执行完操作，把 data2 存入数据库，然后是 data1/data3。这不明显乱了。



- **Kafka**：比如说我们建了一个 topic，有三个 partition。生产者在写的时候，其实可以指定一个 key，比如说我们指定了某个订单 id 作为 key，那么这个订单相关的数据，一定会被分发到同一个 partition 中去，而且这个 partition 中的数据一定是有顺序的。
  消费者从 partition 中取出来数据的时候，也一定是有顺序的。到这里，顺序还是 ok 的，没有错乱。接着，我们在消费者里可能会搞**多个线程来并发处理消息**。因为如果消费者是单

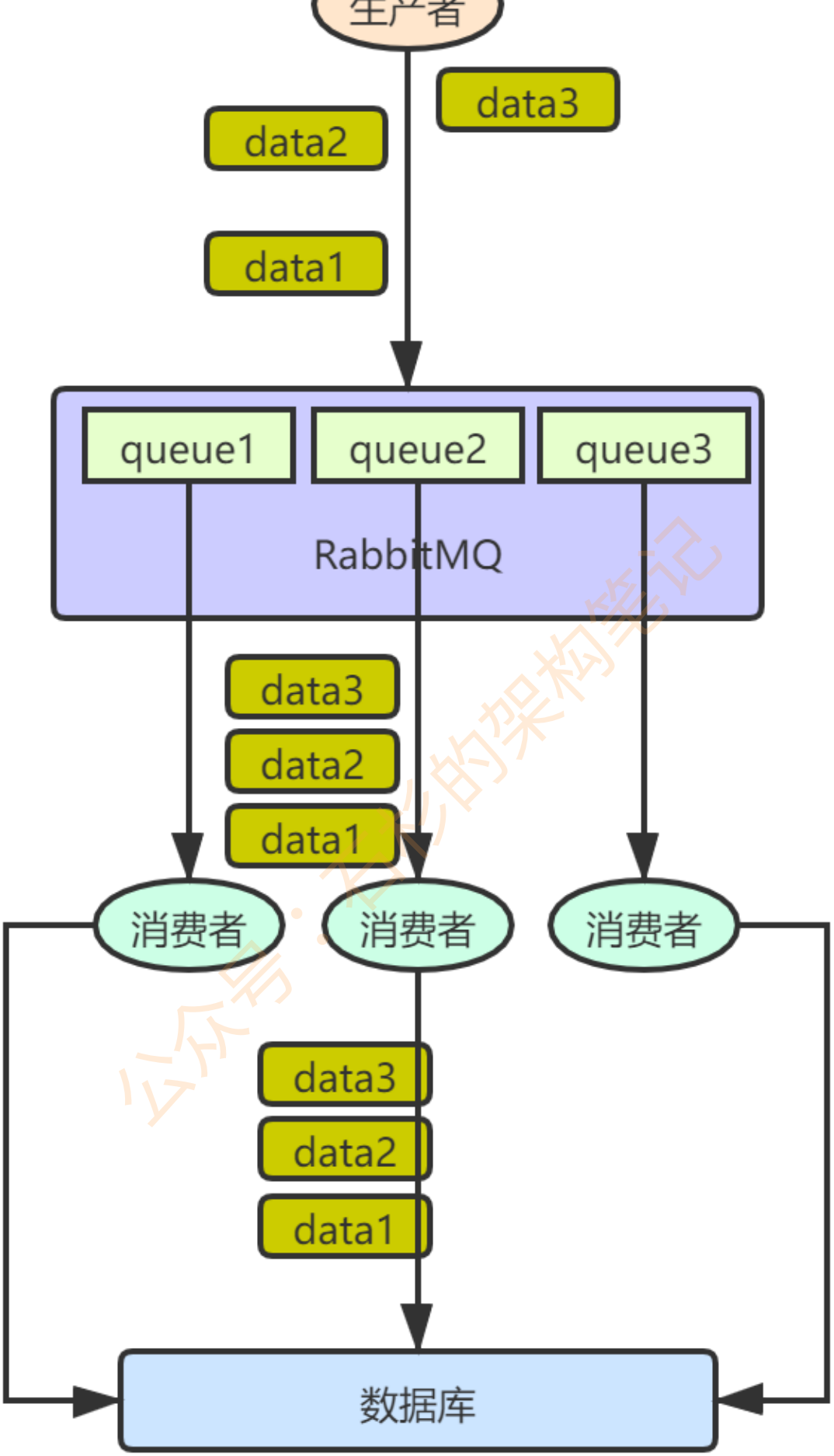
线程消费处理，而处理比较耗时的话，比如处理一条消息耗时几十 ms，那么 1 秒钟只能处理几十条消息，这吞吐量太低了。而多个线程并发跑的话，顺序可能就乱掉了。



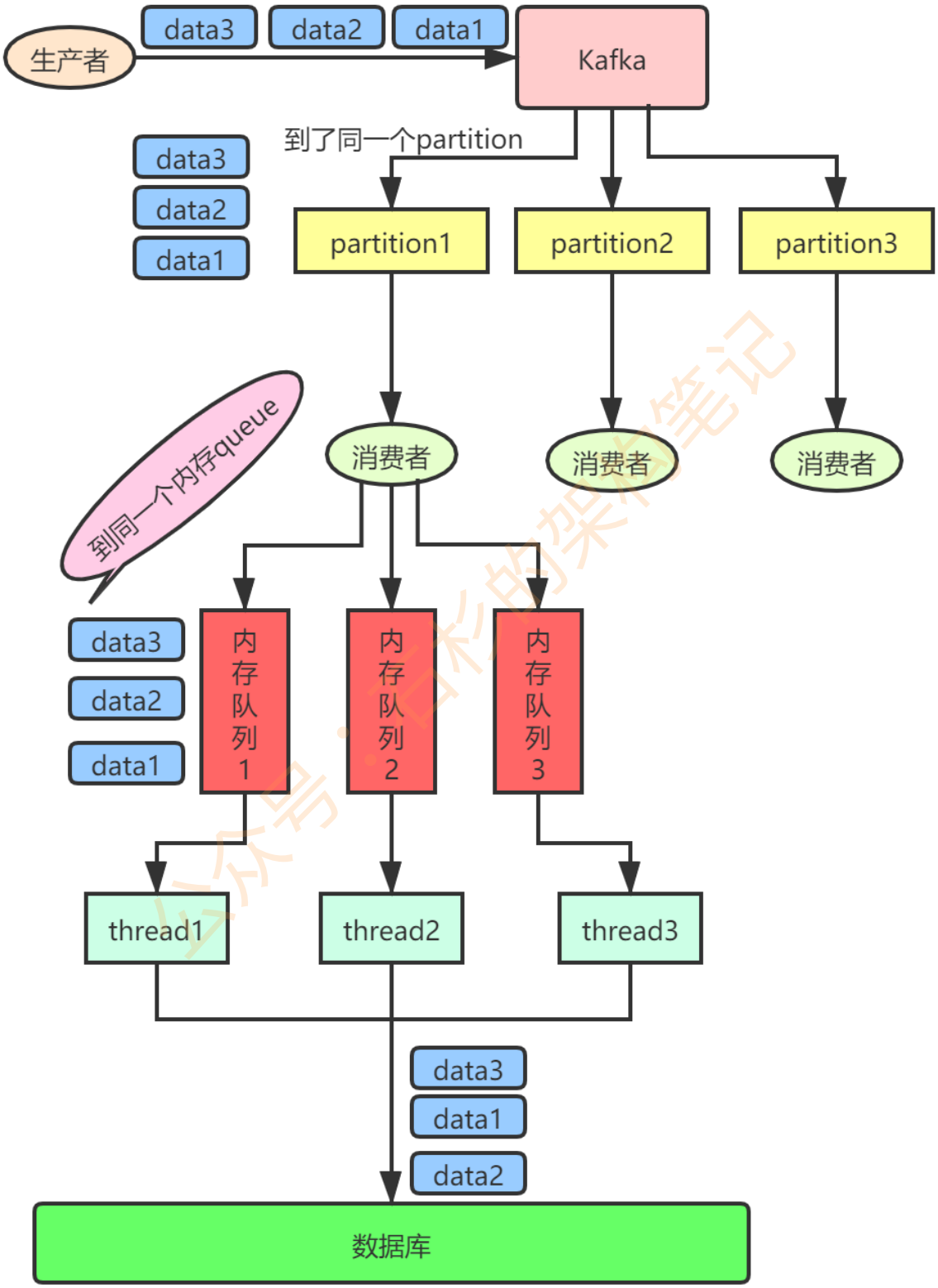
解决方案

RabbitMQ

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。


生产者
data3
data2
data1
queue1
queue2
queue3
RabbitMQ
data3
data2
data1
消费者
消费者
消费者
data3
data2
data1
数据库

- 一个 topic，一个 partition，一个 consumer，内部单线程消费，单线程吞吐量太低，一般不会用这个。
- 写 N 个内存 queue，具有相同 key 的数据都到同一个内存 queue；然后对于 N 个线程，每个线程分别消费一个内存 queue 即可，这样就能保证顺序性。



【面试题】 - 如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

面试官心理分析

你看这问法，其实本质针对的场景，都是说，可能你的消费端出了问题，不消费了；或者消费的速度极其慢。接着就坑爹了，可能你的消息队列集群的磁盘都快写满了，都没人消费，这个时候怎么办？或者是这整个就积压了几个小时，你这个时候怎么办？或者是你积压的时间太长了，导致比如 RabbitMQ 设置了消息过期时间后就没了怎么办？

所以就这事儿，其实线上挺常见的，一般不出，一出就是大 case。一般常见于，举个例子，消费端每次消费之后要写 mysql，结果 mysql 挂了，消费端 hang 那儿了，不动了；或者是消费端出了个什么岔子，导致消费速度极其慢。

面试题剖析

关于这个事儿，我们一个一个来梳理吧，先假设一个场景，我们现在消费端出故障了，然后大量消息在 mq 里积压，现在出事故了，慌了。

大量消息在 mq 里积压了几个小时了还没解决

几千万条数据在 MQ 里积压了七八个小时，从下午 4 点多，积压到了晚上 11 点多。这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复 consumer 的问题，让它恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是 1000 条，一秒 3 个消费者是 3000 条，一分钟就是 18 万条。所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概 1 小时的时间才能恢复过来。

一般这个时候，只能临时紧急扩容了，具体操作步骤和思路如下：

- 先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。
- 新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。
- 然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，**消费之后不做耗时的处理**，直接均匀轮询写入临时建立好的 10 倍数量的 queue。
- 接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。

- 等快速消费完积压数据之后，**得恢复原先部署的架构，重新**用原先的 consumer 机器来消费消息。

mq 中的消息过期失效了

假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是**大量的数据会直接搞丢**。

这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是**批量重导**，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。

假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq 都快写满了

如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，**消费一个丢弃一个，都不要了**，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

【面试题】- 如果让你写一个消息队列，该如何进行架构设计？说一下你的思路。

面试官心理分析

其实聊到这个问题，一般面试官要考察两块：

- 你有没有对某一个消息队列做过较为深入的原理的了解，或者从整体了解把握住一个消息队列的架构原理。
- 看看你的设计能力，给你一个常见的系统，就是消息队列系统，看看你能不能从全局把握一下整体架构设计，给出一些关键点出来。

说实话，问类似问题的时候，大部分人基本都会蒙，因为平时从来没有思考过类似的问题，大多数人就是平时埋头用，从来不去思考背后的一些东西。类似的问题，比如，如果让你来设计一个 Spring 框架你会怎么做？如果让你来设计一个 Dubbo 框架你会怎么做？如果让你来设计一个 MyBatis 框架你会怎么做？

面试题剖析

其实回答这类问题，说白了，不求你看过那技术的源码，起码你要大概知道那个技术的基本原理、核心组成部分、基本架构构成，然后参照一些开源的技术把一个系统设计出来的思路说一下就好。

比如说这个消息队列系统，我们从以下几个角度来考虑一下：

- 首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

- 其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

- 其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

- 能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

mq 肯定是很复杂的，面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。

lucene 和 es 的前世今生

lucene 是最先进、功能最强大的搜索库。如果直接基于 lucene 开发，非常复杂，即便写一些简单的功能，也要写大量的 Java 代码，需要深入理解原理。

elasticsearch 基于 lucene，隐藏了 lucene 的复杂性，提供了简单易用的 restful api / Java api 接口（另外还有其他语言的 api 接口）。

- 分布式的文档存储引擎

- 分布式的搜索引擎和分析引擎
- 分布式，支持 PB 级数据

es 的核心概念

Near Realtime

近实时，有两层意思：

- 从写入数据到数据可以被搜索到有一个小延迟（大概是 1s）
- 基于 es 执行搜索和分析可以达到秒级

Cluster 集群

集群包含多个节点，每个节点属于哪个集群都是通过一个配置来决定的，对于中小型应用来说，刚开始一个集群就一个节点很正常。

Node 节点

Node 是集群中的一个节点，节点也有一个名称，默认是随机分配的。默认节点会去加入一个名称为 `elasticsearch` 的集群。如果直接启动一堆节点，那么它们会自动组成一个 elasticsearch 集群，当然一个节点也可以组成 elasticsearch 集群。

Document & field

文档是 es 中最小的数据单元，一个 document 可以是一条客户数据、一条商品分类数据、一条订单数据，通常用 json 数据结构来表示。每个 index 下的 type，都可以存储多条 document。一个 document 里面有多个 field，每个 field 就是一个数据字段。

```json
{
    "product_id": "1",
    "product_name": "iPhone X",
    "product_desc": "苹果手机",
    "category_id": "2",
    "category_name": "电子产品"
}
```

Index

索引包含了一堆有相似结构的文档数据，比如商品索引。一个索引包含很多 document，一个索引就代表了一类相似或者相同的 ducument。

Type

类型，每个索引里可以有一个或者多个 type，type 是 index 的一个逻辑分类，比如商品 index 下有多个 type：日化商品 type、电器商品 type、生鲜商品 type。每个 type 下的 document 的 field 可能不太一样。

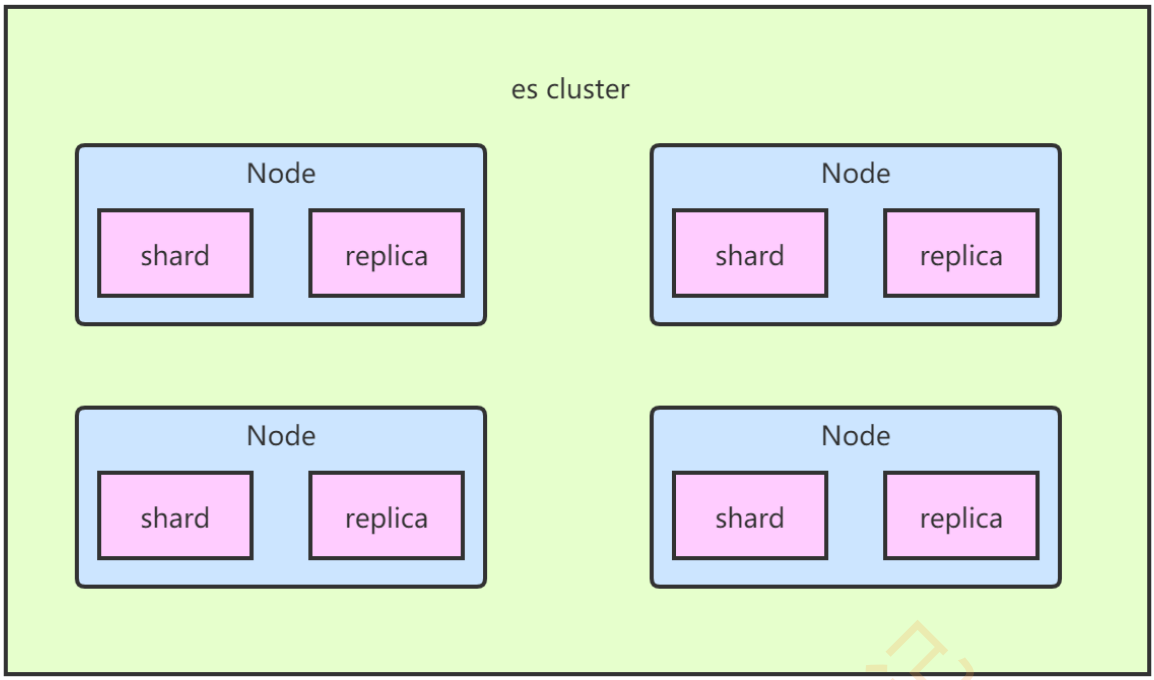
shard

单台机器无法存储大量数据，es 可以将一个索引中的数据切分为多个 shard，分布在多台服务器上存储。有了 shard 就可以横向扩展，存储更多数据，让搜索和分析等操作分布到多台服务器上去执行，提升吞吐量和性能。每个 shard 都是一个 lucene index。

replica

任何一个服务器随时可能故障或宕机，此时 shard 可能就会丢失，因此可以为每个 shard 创建多个 replica 副本。replica 可以在 shard 故障时提供备用服务，保证数据不丢失，多个 replica 还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认 5 个），replica shard（随时修改数量，默认 1 个），默认每个索引 10 个 shard，5 个 primary shard，5个 replica shard，最小的高可用配置，是 2 台服务器。

这么说吧，shard 分为 primary shard 和 replica shard。而 primary shard 一般简称为 shard，而 replica shard 一般简称为 replica。



es 核心概念 vs. db 核心概念

es	db
index	数据库
type	数据表
docuemnt	一行数据

以上是一个简单的类比。

【面试题】 - es 的分布式架构原理能说一下么（es 是如何实现分布式的啊）？

面试官心理分析

在搜索这块，lucene 是最流行的搜索库。几年前业内一般都问，你了解 lucene 吗？你知道倒排索引的原理吗？现在早已经 out 了，因为现在很多项目都是直接用基于 lucene 的分布式搜索引擎—— ElasticSearch，简称为 es。

而现在分布式搜索基本已经成为大部分互联网行业的 Java 系统的标配，其中尤为流行的就是 es，前几年 es 没火的时候，大家一般用 solr。但是这两年基本大部分企业和项目都开始转向 es

了。

所以互联网面试，肯定会跟你聊聊分布式搜索引擎，也就一定会聊聊 es，如果你确实不知道，那你真的就 out 了。

如果面试官问你第一个问题，确实一般都会问你 es 的分布式架构设计能介绍一下么？就看看你对分布式搜索引擎架构的一个基本理解。

面试题剖析

ElasticSearch 设计的理念就是分布式搜索引擎，底层其实还是基于 lucene 的。核心思想就是在多台机器上启动多个 es 进程实例，组成了一个 es 集群。

es 中存储数据的**基本单位是索引**，比如说你现在要在 es 中存储一些订单数据，你就应该在 es 中创建一个索引 `order_idx` ，所有的订单数据就都写到这个索引里面去，一个索引差不多就是相当于是 mysql 里的一张表。

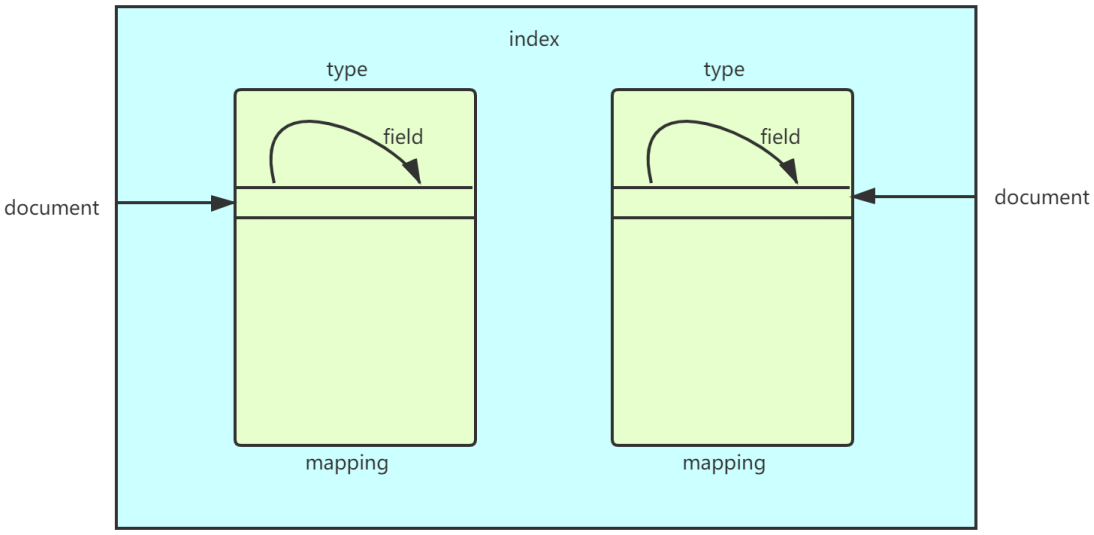
```
index -> type -> mapping -> document -> field。
```

这样吧，为了做个更直白的介绍，我在这里做个类比。但是切记，不要划等号，类比只是为了便于理解。

index 相当于 mysql 里的一张表。而 type 没法跟 mysql 里去对比，一个 index 里可以有多个 type，每个 type 的字段都是差不多的，但是有一些略微的差别。假设有一个 index，是订单 index，里面专门是放订单数据的。就好比说你在 mysql 中建表，有些订单是实物商品的订单，比如一件衣服、一双鞋子；有些订单是虚拟商品的订单，比如游戏点卡，话费充值。就两种订单大部分字段是一样的，但是少部分字段可能有略微的一些差别。

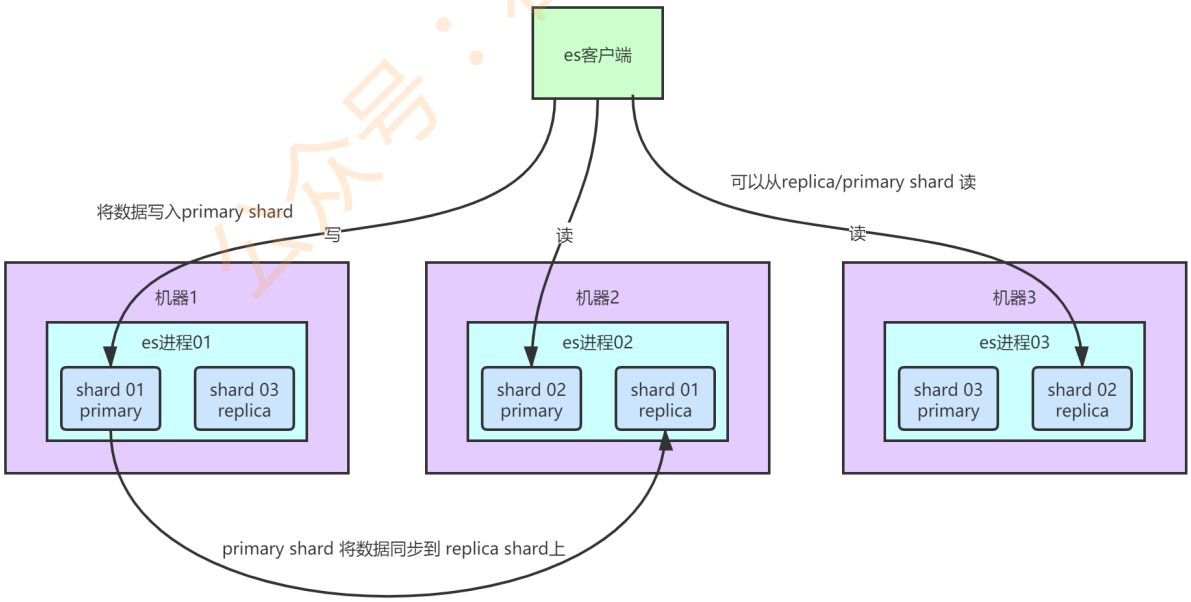
所以就会在订单 index 里，建两个 type，一个是实物商品订单 type，一个是虚拟商品订单 type，这两个 type 大部分字段是一样的，少部分字段是不一样的。

很多情况下，一个 index 里可能就一个 type，但是确实如果说是一个 index 里有多个 type 的情况（**注意**， `mapping types` 这个概念在 ElasticSearch 7.X 已被完全移除，详细说明可以参考**官方文档**），你可以认为 index 是一个类别的表，具体的每个 type 代表了 mysql 中的一个表。每个 type 有一个 mapping，如果你认为一个 type 是具体的一个表，index 就代表多个 type 同属于的一个类型，而 mapping 就是这个 type 的**表结构定义**，你在 mysql 中创建一个表，肯定是要定义表结构的，里面有哪些字段，每个字段是什么类型。实际上你往 index 里的一个 type 里面写的一条数据，叫做一条 document，一条 document 就代表了 mysql 中某个表里的一行，每个 document 有多个 field，每个 field 就代表了这个 document 中的一个字段的值。



你搞一个索引，这个索引可以拆分成多个 `shard` ，每个 shard 存储部分数据。拆分多个 shard 是有好处的，一是**支持横向扩展**，比如你数据量是 3T，3 个 shard，每个 shard 就 1T 的数据，若现在数据量增加到 4T，怎么扩展，很简单，重新建一个有 4 个 shard 的索引，将数据导进去；二是**提高性能**，数据分布在多个 shard，即多台服务器上，所有的操作，都会在多台机器上并行分布式执行，提高了吞吐量和性能。

接着就是这个 shard 的数据实际是有多个备份，就是说每个 shard 都有一个 `primary shard` ，负责写入数据，但是还有几个 `replica shard` 。 `primary shard` 写入数据之后，会将数据同步到其他几个 `replica shard` 上去。



通过这个 replica 的方案，每个 shard 的数据都有多个备份，如果某个机器宕机了，没关系啊，还有别的数据副本在别的机器上呢。高可用了吧。

es 集群多个节点，会自动选举一个节点为 master 节点，这个 master 节点其实就是干一些管理的工作的，比如维护索引元数据、负责切换 primary shard 和 replica shard 身份等。要是 master 节点宕机了，那么会重新选举一个节点为 master 节点。

如果是非 master节点宕机了，那么会由 master 节点，让那个宕机节点上的 primary shard 的身份转移到其他机器上的 replica shard。接着你要是修复了那个宕机机器，重启了之后，master 节点会控制将缺失的 replica shard 分配过去，同步后续修改的数据之类的，让集群恢复正常。

说得更简单一点，就是说如果某个非 master 节点宕机了。那么此节点上的 primary shard 不就没了。那好，master 会让 primary shard 对应的 replica shard（在其他机器上）切换为 primary shard。如果宕机的机器修复了，修复后的节点也不再是 primary shard，而是 replica shard。

其实上述就是 ElasticSearch 作为分布式搜索引擎最基本的一个架构设计。

【面试题】 - es 写入数据的工作原理是什么啊？es 查询数据的工作原理是什么啊？底层的 lucene 介绍一下呗？倒排索引了解吗？

面试官心理分析

问这个，其实面试官就是要看看你了解不了解 es 的一些基本原理，因为用 es 无非就是写入数据，搜索数据。你要是不明白你发起一个写入和搜索请求的时候，es 在干什么，那你真的是.....

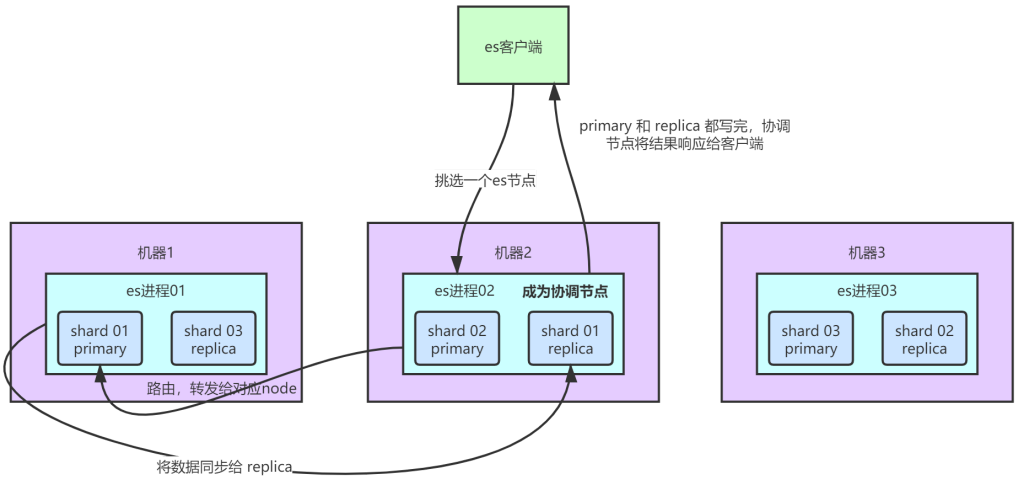
对 es 基本就是个黑盒，你还能干啥？你唯一能干的就是用 es 的 api 读写数据了。要是出点什么问题，你啥都不知道，那还能指望你什么呢？

面试题剖析

es 写数据过程

- 客户端选择一个 node 发送请求过去，这个 node 就是 `coordinating node` （协调节点）。
- `coordinating node` 对 document 进行**路由**，将请求转发给对应的 node（有 primary shard）。
- 实际的 node 上的 `primary shard` 处理请求，然后将数据同步到 `replica node` 。

- coordinating node 如果发现 primary node 和所有 replica node 都搞定之后，就返回响应结果给客户端。



es 读数据过程

可以通过 doc id 来查询，会根据 doc id 进行 hash，判断出来当时把 doc id 分配到了哪个 shard 上面去，从那个 shard 去查询。

- 客户端发送请求到**任意**一个 node，成为 coordinate node 。
- coordinate node 对 doc id 进行哈希路由，将请求转发到对应的 node，此时会使用 round-robin **随机轮询算法**，在 primary shard 以及其所有 replica 中随机选择一个，让读请求负载均衡。
- 接收请求的 node 返回 document 给 coordinate node 。
- coordinate node 返回 document 给客户端。

es 搜索数据过程

es 最强大的是做全文检索，就是比如你有三条数据：

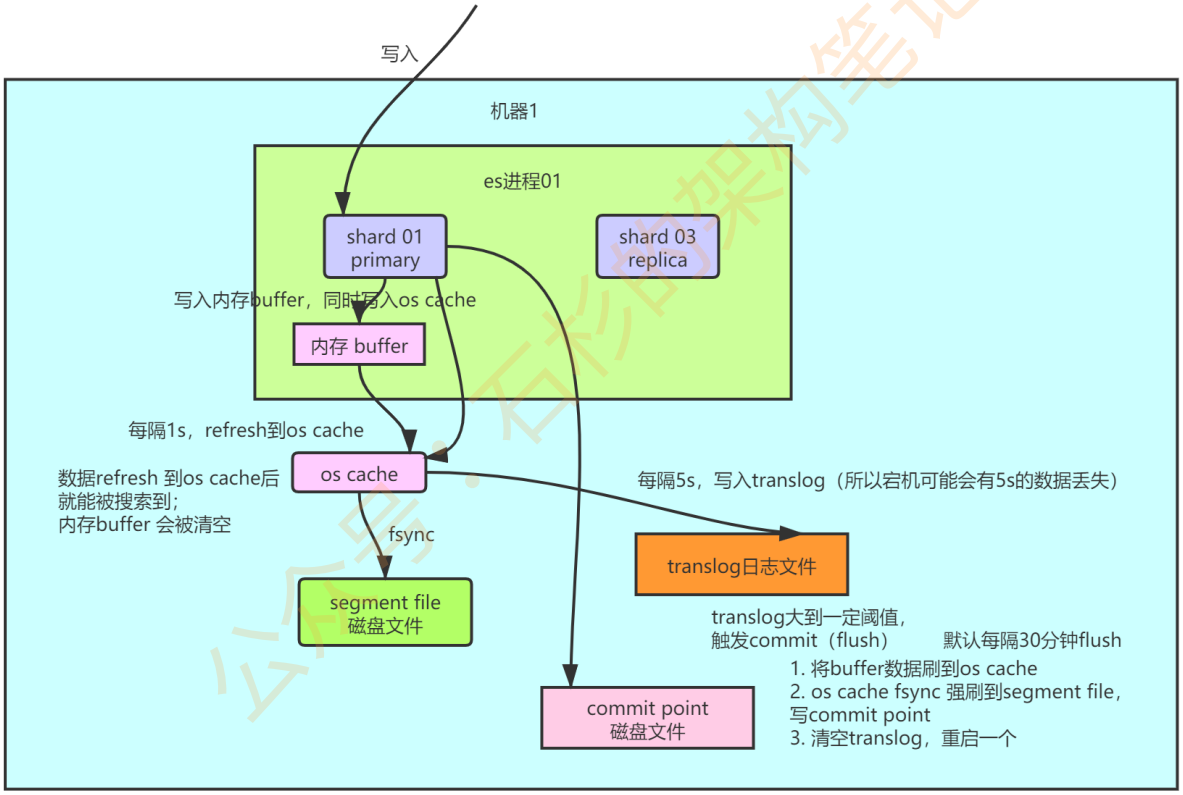
```
java真好玩儿啊
java好难学啊
j2ee特别牛
```

你根据 java 关键词来搜索，将包含 java 的 document 给搜索出来。es 就会给你返回：java真好玩儿啊，java好难学啊。

- 客户端发送请求到一个 `coordinate node` 。
- 协调节点将搜索请求转发到**所有**的 shard 对应的 `primary shard` 或 `replica shard` ，都可以。
- query phase：每个 shard 将自己的搜索结果（其实就是一些 `doc id` ）返回给协调节点，由协调节点进行数据的合并、排序、分页等操作，产出最终结果。
- fetch phase：接着由协调节点根据 `doc id` 去各个节点上**拉取实际**的 `document` 数据，最终返回给客户端。

> **写请求是写入 primary shard，然后同步给所有的 replica shard；读请求可以从 primary shard 或 replica shard 读取，采用的是随机轮询算法。**

写数据底层原理



先写入内存 buffer，在 buffer 里的时候数据是搜索不到的；同时将数据写入 translog 日志文件。

如果 buffer 快满了，或者到一定时间，就会将内存 buffer 数据 `refresh` 到一个新的 `segment file` 中，但是此时数据不是直接进入 `segment file` 磁盘文件，而是先进入 `os cache` 。这个过程就是 `refresh` 。

每隔 1 秒钟，es 将 buffer 中的数据写入一个**新的** `segment file` ，每秒钟会产生一个**新的磁盘文件** `segment file` ，这个 `segment file` 中就存储最近 1 秒内 buffer 中写入的数据。

但是如果 buffer 里面此时没有数据，那当然不会执行 refresh 操作，如果 buffer 里面有数据，默认 1 秒钟执行一次 refresh 操作，刷入一个新的 segment file 中。

操作系统里面，磁盘文件其实都有一个东西，叫做 `os cache`，即操作系统缓存，就是说数据写入磁盘文件之前，会先进入 `os cache`，先进入操作系统级别的一个内存缓存中去。只要 `buffer` 中的数据被 refresh 操作刷入 `os cache` 中，这个数据就可以被搜索到了。

为什么叫 es 是**准实时**的？ `NRT`，全称 `near real-time`。默认是每隔 1 秒 refresh 一次的，所以 es 是准实时的，因为写入的数据 1 秒之后才能被看到。可以通过 es 的 `restful api` 或者 `java api`，**手动**执行一次 refresh 操作，就是手动将 buffer 中的数据刷入 `os cache` 中，让数据立马就可以被搜索到。只要数据被输入 `os cache` 中，buffer 就会被清空了，因为不需要保留 buffer 了，数据在 translog 里面已经持久化到磁盘去一份了。

重复上面的步骤，新的数据不断进入 buffer 和 translog，不断将 `buffer` 数据写入一个又一个新的 `segment file` 中去，每次 `refresh` 完 buffer 清空，translog 保留。随着这个过程推进，translog 会变得越来越大。当 translog 达到一定长度的时候，就会触发 `commit` 操作。

commit 操作发生第一步，就是将 buffer 中现有数据 `refresh` 到 `os cache` 中去，清空 buffer。然后，将一个 `commit point` 写入磁盘文件，里面标识着这个 `commit point` 对应的所有 `segment file`，同时强行将 `os cache` 中目前所有的数据都 `fsync` 到磁盘文件中去。最后**清空** 现有 translog 日志文件，重启一个 translog，此时 commit 操作完成。

这个 commit 操作叫做 `flush`。默认 30 分钟自动执行一次 `flush`，但如果 translog 过大，也会触发 `flush`。flush 操作就对应着 commit 的全过程，我们可以通过 es api，手动执行 flush 操作，手动将 os cache 中的数据 fsync 强刷到磁盘上去。

translog 日志文件的作用是什么？你执行 commit 操作之前，数据要么是停留在 buffer 中，要么是停留在 os cache 中，无论是 buffer 还是 os cache 都是内存，一旦这台机器死了，内存中的数据就全丢了。所以需要将数据对应的操作写入一个专门的日志文件 `translog` 中，一旦此时机器宕机，再次重启的时候，es 会自动读取 translog 日志文件中的数据，恢复到内存 buffer 和 os cache 中去。

translog 其实也是先写入 os cache 的，默认每隔 5 秒刷一次到磁盘中去，所以默认情况下，可能有 5 秒的数据会仅仅停留在 buffer 或者 translog 文件的 os cache 中，如果此时机器挂了，会**丢失** 5 秒钟的数据。但是这样性能比较好，最多丢 5 秒的数据。也可以将 translog 设置成每次写操作必须是直接 `fsync` 到磁盘，但是性能会差很多。

实际上你在这里，如果面试官没有问你 es 丢数据的问题，你可以在这里给面试官炫一把，你说，其实 es 第一是准实时的，数据写入 1 秒后可以搜索到；可能会丢失数据的。有 5 秒的数据，停留在 buffer、translog os cache、segment file os cache 中，而不在磁盘上，此时如果宕机，会导致 5 秒的**数据丢失**。

总结一下，数据先写入内存 buffer，然后每隔 1s，将数据 refresh 到 os cache，到了 os cache 数据就能被搜索到（所以我们才说 es 从写入到能被搜索到，中间有 1s 的延迟）。每隔 5s，将数

据写入 translog 文件（这样如果机器宕机，内存数据全没，最多会有 5s 的数据丢失），translog 大到一定程度，或者默认每隔 30mins，会触发 commit 操作，将缓冲区的数据都 flush 到 segment file 磁盘文件中。

> **数据写入 segment file 之后，同时就建立好了倒排索引。**

删除/更新数据底层原理

如果是删除操作，commit 的时候会生成一个 `.del` 文件，里面将某个 doc 标识为 `deleted` 状态，那么搜索的时候根据 `.del` 文件就知道这个 doc 是否被删除了。

如果是更新操作，就是将原来的 doc 标识为 `deleted` 状态，然后新写入一条数据。

buffer 每 refresh 一次，就会产生一个 `segment file`，所以默认情况下是 1 秒钟一个 `segment file`，这样下来 `segment file` 会越来越多，此时会定期执行 merge。每次 merge 的时候，会将多个 `segment file` 合并成一个，同时这里会将标识为 `deleted` 的 doc 给**物理删除掉**，然后将新的 `segment file` 写入磁盘，这里会写一个 `commit point`，标识所有新的 `segment file`，然后打开 `segment file` 供搜索使用，同时删除旧的 `segment file`。

底层 lucene

简单来说，lucene 就是一个 jar 包，里面包含了封装好的各种建立倒排索引的算法代码。我们用 Java 开发的时候，引入 lucene jar，然后基于 lucene 的 api 去开发就可以了。

通过 lucene，我们可以将已有的数据建立索引，lucene 会在本地磁盘上面，给我们组织索引的数据结构。

倒排索引

在搜索引擎中，每个文档都有一个对应的文档 ID，文档内容被表示为一系列关键词的集合。例如，文档 1 经过分词，提取了 20 个关键词，每个关键词都会记录它在文档中出现的次数和出现位置。

那么，倒排索引就是**关键词到文档** ID 的映射，每个关键词都对应着一系列的文件，这些文件中都出现了关键词。

举个栗子。

有以下文档：

DocId	Doc
1	谷歌地图之父跳槽 Facebook
2	谷歌地图之父加盟 Facebook
3	谷歌地图创始人拉斯离开谷歌加盟 Facebook
4	谷歌地图之父跳槽 Facebook 与 Wave 项目取消有关
5	谷歌地图之父拉斯加盟社交网站 Facebook

对文档进行分词之后，得到以下**倒排索引**。

WordId	Word	DocIds
1	谷歌	1,2,3,4,5
2	地图	1,2,3,4,5
3	之父	1,2,4,5
4	跳槽	1,4
5	Facebook	1,2,3,4,5
6	加盟	2,3,5
7	创始人	3
8	拉斯	3,5
9	离开	3
10	与	4
..	..	..

另外，实用的倒排索引还可以记录更多的信息，比如文档频率信息，表示在文档集合中有多少个文档包含某个单词。

那么，有了倒排索引，搜索引擎可以很方便地响应用户的查询。比如用户输入查询 `Facebook` ，搜索系统查找倒排索引，从中读出包含这个单词的文档，这些文档就是提供给用户的搜索结果。

要注意倒排索引的两个重要细节：

- 倒排索引中的所有词项对应一个或多个文档；
- 倒排索引中的词项**根据字典顺序升序排列**

> **上面只是一个简单的栗子，并没有严格按照字典顺序升序排列。**

【面试题】 - es 在数据量很大的情况下（数十亿级别）如何提高查询效率啊?

面试官心理分析

这个问题是肯定要问的，说白了，就是看你有没有实际干过 es，因为啥？其实 es 性能并没有你想象中那么好的。很多时候数据量大了，特别是有几亿条数据的时候，可能你会懵逼的发现，跑个搜索怎么一下 `5~10s` ，坑爹了。第一次搜索的时候，是 `5~10s` ，后面反而就快了，可能就几百毫秒。

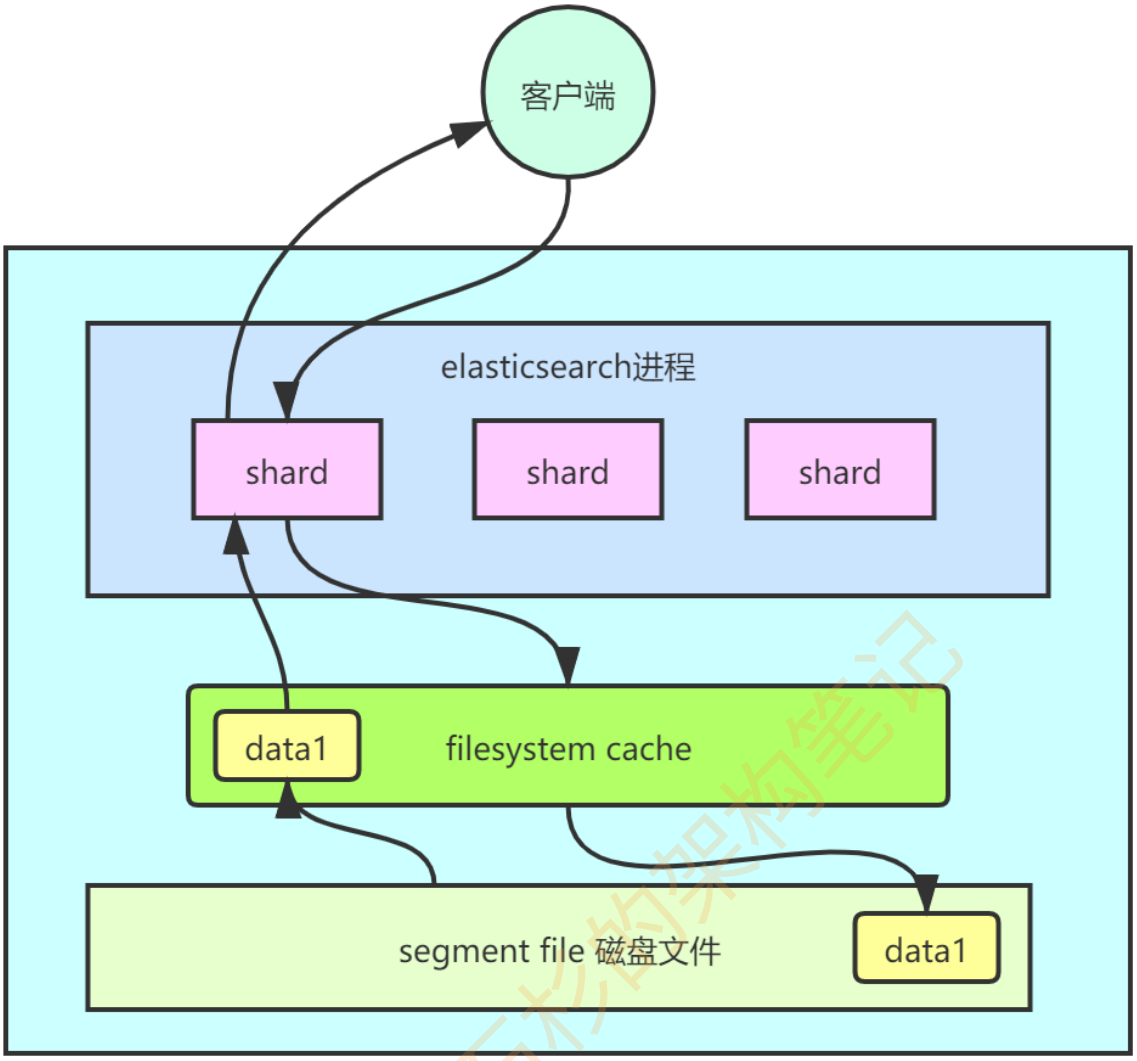
你就很懵，每个用户第一次访问都会比较慢，比较卡么？所以你要是没玩儿过 es，或者就是自己玩玩儿 demo，被问到这个问题容易懵逼，显示出你对 es 确实玩儿的不怎么样?

面试题剖析

说实话，es 性能优化是没有什么银弹的，啥意思呢？就是**不要期待着随手调一个参数，就可以万能的应对所有的性能慢的场景**。也许有的场景是你换个参数，或者调整一下语法，就可以搞定，但是绝对不是所有场景都可以这样。

性能优化的杀手锏——filesystem cache

你往 es 里写的数据，实际上都写到磁盘文件里去了，**查询的时候**，操作系统会将磁盘文件里的数据自动缓存到 `filesystem cache` 里面去。



es 的搜索引擎严重依赖于底层的 `filesystem cache` ，你如果给 `filesystem cache` 更多的内存，尽量让内存可以容纳所有的 `idx segment file` 索引数据文件，那么你搜索的时候就基本都是走内存的，性能会非常高。

性能差距究竟可以有多大？我们之前很多的测试和压测，如果走磁盘一般肯定上秒，搜索性能绝对是秒级别的，1秒、5秒、10秒。但如果是走 `filesystem cache` ，是走纯内存的，那么一般来说性能比走磁盘要高一个数量级，基本上就是毫秒级的，从几毫秒到几百毫秒不等。

这里有个真实的案例。某个公司 es 节点有 3 台机器，每台机器看起来内存很多，64G，总内存就是 `64 * 3 = 192G` 。每台机器给 es jvm heap 是 `32G` ，那么剩下来留给 `filesystem cache` 的就是每台机器才 `32G` ，总共集群里给 `filesystem cache` 的就是 `32 * 3 = 96G` 内存。而此时，整个磁盘上索引数据文件，在 3 台机器上一共占用了 `1T` 的磁盘容量，es 数据量是 `1T` ，那么每台机器的数据量是 `300G` 。这样性能好吗？ `filesystem cache` 的内存才 100G，十分之一的数据可以放内存，其他的都在磁盘，然后你执行搜索操作，大部分操作都是走磁盘，性能肯定差。

归根结底，你要让 es 性能要好，最佳的情况下，就是你的机器的内存，至少可以容纳你的总数据量的一半。

根据我们自己的生产环境实践经验，最佳的情况下，是仅仅在 es 中就存少量的数据，就是你要**用来搜索的那些索引**，如果内存留给 `filesystem cache` 的是 100G，那么你就将索引数据控制在 `100G` 以内，这样的话，你的数据几乎全部走内存来搜索，性能非常之高，一般可以在 1 秒以内。

比如说你现在有一行数据。 `id,name,age ....` 30 个字段。但是你现在搜索，只需要根据 `id,name,age` 三个字段来搜索。如果你傻乎乎往 es 里写入一行数据所有的字段，就会导致说 `90%` 的数据是不用来搜索的，结果硬是占据了 es 机器上的 `filesystem cache` 的空间，单条数据的数据量越大，就会导致 `filesystem cahce` 能缓存的数据就越少。其实，仅仅写入 es 中要用来检索的**少数几个字段**就可以了，比如说就写入 es `id,name,age` 三个字段，然后你可以把其他的字段数据存在 mysql/hbase 里，我们一般是建议用 `es + hbase` 这么一个架构。

hbase 的特点是**适用于海量数据的在线存储**，就是对 hbase 可以写入海量数据，但是不要做复杂的搜索，做很简单的一些根据 id 或者范围进行查询的这么一个操作就可以了。从 es 中根据 name 和 age 去搜索，拿到的结果可能就 20 个 `doc id` ，然后根据 `doc id` 到 hbase 里去查询每个 `doc id` 对应的**完整的数据**，给查出来，再返回给前端。

写入 es 的数据最好小于等于，或者是略微大于 es 的 filesystem cache 的内存容量。然后你从 es 检索可能就花费 20ms，然后再根据 es 返回的 id 去 hbase 里查询，查 20 条数据，可能也就耗费个 30ms，可能你原来那么玩儿，1T 数据都放 es，会每次查询都是 5~10s，现在可能性能就会很高，每次查询就是 50ms。

数据预热

假如说，哪怕是你就按照上述的方案去做了，es 集群中每个机器写入的数据量还是超过了 `filesystem cache` 一倍，比如说你写入一台机器 60G 数据，结果 `filesystem cache` 就 30G，还是有 30G 数据留在了磁盘上。

其实可以做**数据预热**。

举个例子，拿微博来说，你可以把一些大V，平时看的人很多的数据，你自己提前后台搞个系统，每隔一会儿，自己的后台系统去搜索一下热数据，刷到 `filesystem cache` 里去，后面用户实际上来看这个热数据的时候，他们就是直接从内存里搜索了，很快。

或者是电商，你可以将平时查看最多的一些商品，比如说 iphone 8，热数据提前后台搞个程序，每隔 1 分钟自己主动访问一次，刷到 `filesystem cache` 里去。

对于那些你觉得比较热的、经常会有人访问的数据，最好**做一个专门的缓存预热子系统**，就是对热数据每隔一段时间，就提前访问一下，让数据进入 `filesystem cache` 里面去。这样下次别人访问的时候，性能一定会好很多。

冷热分离

es 可以做类似于 mysql 的水平拆分，就是说将大量的访问很少、频率很低的数据，单独写一个索引，然后将访问很频繁的热数据单独写一个索引。最好是将**冷数据写入一个索引中，然后热数据写入另外一个索引中**，这样可以确保热数据在被预热之后，尽量都让他们留在 `filesystem os cache` 里，**别让冷数据给冲刷掉**。

你看，假设你有 6 台机器，2 个索引，一个放冷数据，一个放热数据，每个索引 3 个 shard。3 台机器放热数据 index，另外 3 台机器放冷数据 index。然后这样的话，你大量的时间是在访问热数据 index，热数据可能就占总数据量的 10%，此时数据量很少，几乎全都保留在 `filesystem cache` 里面了，就可以确保热数据的访问性能是很高的。但是对于冷数据而言，是在别的 index 里的，跟热数据 index 不在相同的机器上，大家互相之间都没什么联系了。如果有人访问冷数据，可能大量数据是在磁盘上的，此时性能差点，就 10% 的人去访问冷数据，90% 的人在访问热数据，也无所谓了。

document 模型设计

对于 MySQL，我们经常有一些复杂的关联查询。在 es 里该怎么玩儿，es 里面的复杂的关联查询尽量别用，一旦用了性能一般都不太好。

最好是先在 Java 系统里就完成关联，将关联好的数据直接写入 es 中。搜索的时候，就不需要利用 es 的搜索语法来完成 join 之类的关联搜索了。

document 模型设计是非常重要的，很多操作，不要在搜索的时候才想去执行各种复杂的乱七八糟的操作。es 能支持的操作就那么多，不要考虑用 es 做一些它不好操作的事情。如果真的有那种操作，尽量在 document 模型设计的时候，写入的时候就完成。另外对于一些太复杂的操作，比如 join/nested/parent-child 搜索都要尽量避免，性能都很差的。

分页性能优化

es 的分页是较坑的，为啥呢？举个例子吧，假如你每页是 10 条数据，你现在要查询第 100 页，实际上是会把每个 shard 上存储的前 1000 条数据都查到一个协调节点上，如果你有个 5 个 shard，那么就有 5000 条数据，接着协调节点对这 5000 条数据进行一些合并、处理，再获取到最终第 100 页的 10 条数据。

分布式的，你要查第 100 页的 10 条数据，不可能说从 5 个 shard，每个 shard 就查 2 条数据，最后到协调节点合并成 10 条数据吧？你**必须**得从每个 shard 都查 1000 条数据过来，然后根据你的需求进行排序、筛选等等操作，最后再次分页，拿到里面第 100 页的数据。你翻页的时候，翻的越深，每个 shard 返回的数据就越多，而且协调节点处理的时间越长，非常坑爹。所以用 es 做分页的时候，你会发现越翻到后面，就越是慢。

我们之前也是遇到过这个问题，用 es 作分页，前几页就几十毫秒，翻到 10 页或者几十页的时候，基本上就要 5~10 秒才能查出来一页数据了。

有什么解决方案吗?

不允许深度分页（默认深度分页性能很差）

跟产品经理说，你系统不允许翻那么深的页，默认翻的越深，性能就越差。

类似于 app 里的推荐商品不断下拉出来一页一页的

类似于微博中，下拉刷微博，刷出来一页一页的，你可以用 `scroll api` ，关于如何使用，自行上网搜索。

scroll 会一次性给你生成**所有数据的一个快照**，然后每次滑动向后翻页就是通过**游标** `scroll_id` 移动，获取下一页下一页这样子，性能会比上面说的那种分页性能要高很多很多，基本上都是毫秒级的。

但是，唯一的一点就是，这个适合于那种类似微博下拉翻页的，**不能随意跳到任何一页的场景**。也就是说，你不能先进入第 10 页，然后去第 120 页，然后又回到第 58 页，不能随意乱跳页。所以现在很多产品，都是不允许你随意翻页的，app，也有一些网站，做的就是你只能往下拉，一页一页的翻。

初始化时必须指定 `scroll` 参数，告诉 es 要保存此次搜索的上下文多长时间。你需要确保用户不会持续不断翻页翻几个小时，否则可能因为超时而失败。

除了用 `scroll api` ，你也可以用 `search_after` 来做， `search_after` 的思想是使用前一页的结果来帮助检索下一页的数据，显然，这种方式也不允许你随意翻页，你只能一页页往后翻。初始化时，需要使用一个唯一值的字段作为 sort 字段。

【面试题】 - es 生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片?

面试官心理分析

这个问题，包括后面的 redis 什么的，谈到 es、redis、mysql 分库分表等等技术，面试必问！就是你生产环境咋部署的？说白了，这个问题没啥技术含量，就是看你有没有在真正的生产环境里干过这事儿！

有些同学可能是没在生产环境中干过的，没实际去拿线上机器部署过 es 集群，也没实际玩儿过，也没往 es 集群里面导入过几千万甚至是几亿的数据量，可能你就不太清楚这里面的一些生产项目中的细节。

如果你是自己就玩儿过 demo，没碰过真实的 es 集群，那你可能此时会懵。别懵，你一定要云淡风轻的回答出来这个问题，表示你确实干过这事儿。

面试题剖析

其实这个问题没啥，如果你确实干过 es，那你肯定了解你们生产 es 集群的实际情况，部署了几台机器？有多少个索引？每个索引有多大数据量？每个索引给了多少个分片？你肯定知道！

但是如果你确实没干过，也别虚，我给你说一个基本的版本，你到时候就简单说一下就好了。

- es 生产集群我们部署了 5 台机器，每台机器是 6 核 64G 的，集群总内存是 320G。
- 我们 es 集群的日增量数据大概是 2000 万条，每天日增量数据大概是 500MB，每月增量数据大概是 6 亿，15G。目前系统已经运行了几个月，现在 es 集群里数据总量大概是 100G 左右。
- 目前线上有 5 个索引（这个结合你们自己业务来，看看自己有哪些数据可以放 es 的），每个索引的数据量大概是 20G，所以这个数据量之内，我们每个索引分配的是 8 个 shard，比默认的 5 个 shard 多了 3 个 shard。

大概就这么说一下就行了。

【面试题】 - 项目中缓存是如何使用的？为什么要用缓存？缓存使用不当会造成什么后果？

面试官心理分析

这个问题，互联网公司必问，要是一个人连缓存都不太清楚，那确实比较尴尬。

只要问到缓存，上来第一个问题，肯定是先问问你项目哪里用了缓存？为啥要用？不用行不行？如果用了以后可能会有什么不良的后果？

这就是看看你对缓存这个东西背后有没有思考，如果你就是傻乎乎的瞎用，没法给面试官一个合理的解答，那面试官对你印象肯定不太好，觉得你平时思考太少，就知道干活儿。

面试题剖析

项目中缓存是如何使用的?

这个，需要结合自己项目的业务来。

为什么要用缓存?

用缓存，主要有两个用途：**高性能**、**高并发**。

高性能

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作 mysql，半天查出来一个结果，耗时 600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办?

缓存啊，折腾 600ms 查出来的结果，扔缓存里，一个 key 对应一个 value，下次再有人查，别走 mysql 折腾 600ms 了，直接从缓存里，通过一个 key 查出来一个 value，2ms 搞定。性能提升 300 倍。

就是说对于一些需要复杂操作耗时查出来的结果，且确定后面不怎么变化，但是有很多读请求，那么直接将查询出来的结果放在缓存中，后面直接读缓存就好。

高并发

mysql 这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql 单机支撑到 `2000QPS` 也开始容易报警了。

所以要是你有个系统，高峰期一秒钟过来的请求有 1万，那一个 mysql 单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放 mysql。缓存功能简单，说白了就是 `key-value` 式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发 so easy。单机承载并发量是 mysql 单机的几十倍。

> **缓存是走内存的，内存天然就支撑高并发。**

用了缓存之后会有什么不良后果?

常见的缓存问题有以下几个：

- 缓存与数据库双写不一致
- 缓存雪崩、缓存穿透
- 缓存并发竞争

后面再详细说明。

【面试题】 - redis 和 memcached 有什么区别? redis 的线程模型是什么？为什么 redis 单线程却能支撑高并发?

面试官心理分析

这个是问 redis 的时候，最基本的问题吧，redis 最基本的一个内部原理和特点，就是 redis 实际上是个**单线程工作模型**，你要是这个都不知道，那后面玩儿 redis 的时候，出了问题岂不是什么都不知道?

还有可能面试官会问问你 redis 和 memcached 的区别，但是 memcached 是早些年各大互联网公司常用的缓存方案，但是现在近几年基本都是 redis，没什么公司用 memcached 了。

面试题剖析

redis 和 memcached 有啥区别?

redis 支持复杂的数据结构

redis 相比 memcached 来说，拥有**更多的数据结构**，能支持更丰富的数据操作。如果需要缓存能够支持更复杂的结构和操作， redis 会是不错的选择。

redis 原生支持集群模式

在 redis3.x 版本中，便能支持 cluster 模式，而 memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据。

性能对比

由于 redis 只使用**单核**，而 memcached 可以使用**多核**，所以平均每一个核上 redis 在存储小数据时比 memcached 性能更高。而在 100k 以上的数据中，memcached 性能要高于 redis。虽然 redis 最近也在存储大数据的性能上进行优化，但是比起 memcached，还是稍有逊色。

redis 的线程模型

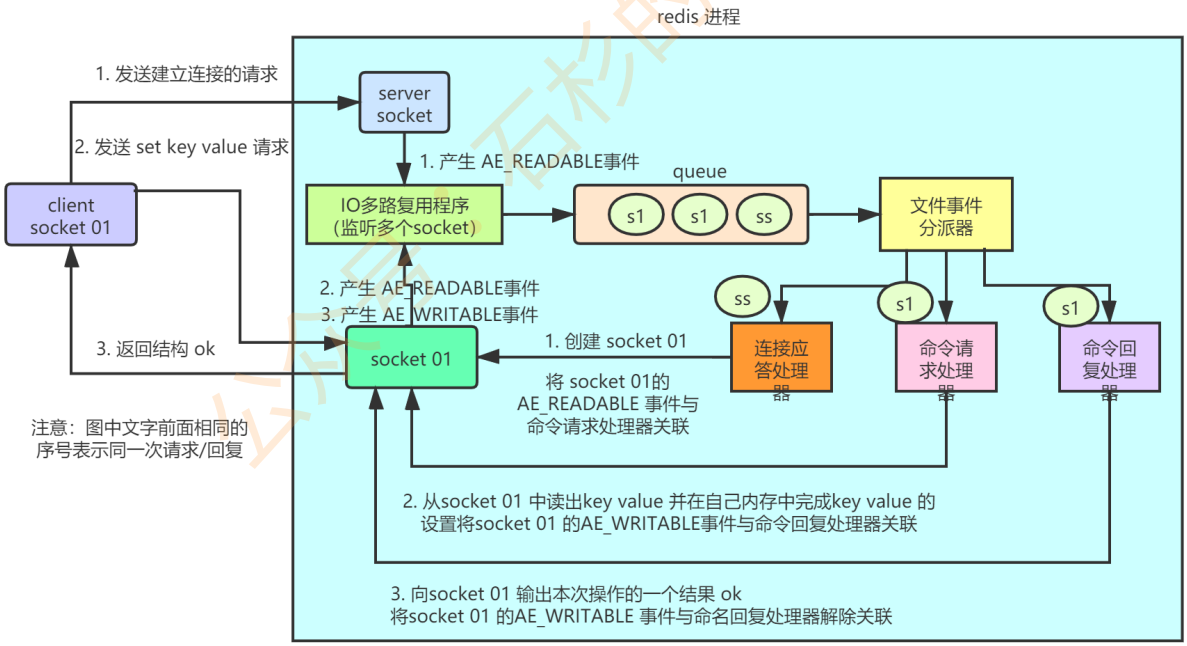
redis 内部使用文件事件处理器 `file event handler` ，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，将产生事件的 socket 压入内存队列中，事件分派器根据 socket 上的事件类型来选择对应的事件处理器进行处理。

文件事件处理器的结构包含 4 个部分：

- 多个 socket
- IO 多路复用程序
- 文件事件分派器
- 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）

多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将产生事件的 socket 放入队列中排队，事件分派器每次从队列中取出一个 socket，根据 socket 的事件类型交给对应的事件处理器进行处理。

来看客户端与 redis 的一次通信过程：



要明白，通信是通过 socket 来完成的，不懂的同学可以先去看一看 socket 网络编程。

首先，redis 服务端进程初始化的时候，会将 server socket 的 `AE_READABLE` 事件与连接应答处理器关联。

客户端 socket01 向 redis 进程的 server socket 请求建立连接，此时 server socket 会产生一个 `AE_READABLE` 事件，IO 多路复用程序监听到 server socket 产生的事件后，将该 socket 压入队列中。文件事件分派器从队列中获取 socket，交给**连接应答处理器**。连接应答处理器会创建一

个能与客户端通信的 socket01，并将该 socket01 的 `AE_READABLE` 事件与命令请求处理器关联。

假设此时客户端发送了一个 `set key value` 请求，此时 redis 中的 socket01 会产生 `AE_READABLE` 事件，IO 多路复用程序将 socket01 压入队列，此时事件分派器从队列中获取到 socket01 产生的 `AE_READABLE` 事件，由于前面 socket01 的 `AE_READABLE` 事件已经与命令请求处理器关联，因此事件分派器将事件交给命令请求处理器来处理。命令请求处理器读取 socket01 的 `key value` 并在自己内存中完成 `key value` 的设置。操作完成后，它会将 socket01 的 `AE_WRITABLE` 事件与命令回复处理器关联。

如果此时客户端准备好接收返回结果了，那么 redis 中的 socket01 会产生一个 `AE_WRITABLE` 事件，同样压入队列中，事件分派器找到相关联的命令回复处理器，由命令回复处理器对 socket01 输入本次操作的一个结果，比如 `ok` ，之后解除 socket01 的 `AE_WRITABLE` 事件与命令回复处理器的关联。

这样便完成了一次通信。关于 Redis 的一次通信过程，推荐读者阅读《**Redis 设计与实现——黄健宏**》进行系统学习。

为啥 redis 单线程模型也能效率这么高?

- 纯内存操作。
- 核心是基于非阻塞的 IO 多路复用机制。
- C 语言实现，一般来说，C 语言实现的程序“距离”操作系统更近，执行速度相对会更快。
- 单线程反而避免了多线程的频繁上下文切换问题，预防了多线程可能产生的竞争问题。

【面试题】 - redis 都有哪些数据类型？分别在哪些场景下使用比较合适?

面试官心理分析

除非是面试官感觉看你简历，是工作 3 年以内的比较初级的同学，可能对技术没有很深入的研究，面试官才会问这类问题。否则，在宝贵的面试时间里，面试官实在不想多问。

其实问这个问题，主要有两个原因：

- 看看你到底有没有全面的了解 redis 有哪些功能，一般怎么来用，啥场景用什么，就怕你别就会最简单的 KV 操作；
- 看看你在实际项目里都怎么玩儿过 redis。

要是你回答的不好，没说出几种数据类型，也没说什么场景，你完了，面试官对你印象肯定不好，觉得你平时就是做个简单的 set 和 get。

面试题剖析

redis 主要有以下几种数据类型：

- string
- hash
- list
- set
- sorted set

string

这是最简单的类型，就是普通的 set 和 get，做简单的 KV 缓存。

```bash
set college szu
```

hash

这个是类似 map 的一种结构，这个一般就是可以将结构化的数据，比如一个对象（前提是**这个对象没嵌套其他的对象**）给缓存在 redis 里，然后每次读写缓存的时候，可以就操作 hash 里的**某个字段**。

```bash
hset person name bingo
hset person age 20
hset person id 1
hget person name
```

```json
person = {
    "name": "bingo",
    "age": 20,
    "id": 1
}
```

list

list 是有序列表，这个可以玩儿出很多花样。

比如可以通过 list 存储一些列表型的数据结构，类似粉丝列表、文章的评论列表之类的东西。

比如可以通过 lrange 命令，读取某个闭区间内的元素，可以基于 list 实现分页查询，这个是很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西，性能高，就一页一页走。

```bash
# 0开始位置，-1结束位置，结束位置为-1时，表示列表的最后一个位置，即查看所有。
lrange mylist 0 -1
```

比如可以搞个简单的消息队列，从 list 头怼进去，从 list 尾巴那里弄出来。

```bash
lpush mylist 1
lpush mylist 2
lpush mylist 3 4 5

# 1
rpop mylist
```


set

set 是无序集合，自动去重。

直接基于 set 将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于 jvm 内存里的 HashSet 进行去重，但是如果你的某个系统部署在多台机器上呢？得基于 redis 进行全局的 set 去重。

可以基于 set 玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁？对吧。

把两个大 V 的粉丝都放在两个 set 中，对两个 set 做交集。

```bash
#-------操作一个set-------
# 添加元素
```

```
sadd mySet 1

# 查看全部元素
smembers mySet

# 判断是否包含某个值
sismember mySet 3

# 删除某个/些元素
srem mySet 1
srem mySet 2 4

# 查看元素个数
scard mySet

# 随机删除一个元素
spop mySet

#-------操作多个set-------
# 将一个set的元素移动到另外一个set
smove yourSet mySet 2

# 求两set的交集
sinter yourSet mySet

# 求两set的并集
sunion yourSet mySet

# 求在yourSet中而不在mySet中的元素
sdiff yourSet mySet
```


sorted set

sorted set 是排序的 set，去重但可以排序，写进去的时候给一个分数，自动根据分数排序。

bash

```
zadd board 85 zhangsan
zadd board 72 lisi
zadd board 96 wangwu
zadd board 63 zhaoliu

# 获取排名前三的用户（默认是升序，所以需要 rev 改为降序）
```

```
zrevrange board 0 3

# 获取某用户的排名
zrank board zhaoliu
```

【面试题】 - redis 的过期策略都有哪些？内存淘汰机制都有哪些？手写一下 LRU 代码实现？

面试官心理分析

如果你连这个问题都不知道，上来就懵了，回答不出来，那线上你写代码的时候，想当然的认为写进 redis 的数据就一定会存在，后面导致系统各种 bug，谁来负责？

常见的有两个问题：

- 往 redis 写入的数据怎么没了？

可能有同学会遇到，在生产环境的 redis 经常会丢掉一些数据，写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明 redis 你就没用对啊。redis 是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个 G 的内存，但是可以有几个 T 的硬盘空间。redis 主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如 redis 就只能用 10G，你要是往里面写了 20G 的数据，会咋办？当然会干掉 10G 的数据，然后就保留 10G 的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

- 数据明明过期了，怎么还占用着内存？

这是由 redis 的过期策略来决定。

面试题剖析

redis 过期策略

redis 过期策略是：**定期删除+惰性删除**。

所谓**定期删除**，指的是 redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。

假设 redis 里放了 10w 个 key，都设置了过期时间，你每隔几百毫秒，就检查 10w 个 key，那 redis 基本上就死了，cpu 负载会很高的，消耗在你的检查过期 key 上了。注意，这里可不是每隔 100ms 就遍历所有的设置过期时间的 key，那样就是一场性能上的**灾难**。实际上 redis 是每隔 100ms **随机抽取**一些 key 来检查和删除的。

但是问题是，定期删除可能会导致很多过期 key 到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，在你获取某个 key 的时候，redis 会检查一下 ，这个 key 如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。

> **获取 key 的时候，如果此时 key 已经过期，就删除，不会返回任何东西。**

但是实际上这还是有问题的，如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期 key 堆积在内存里，导致 redis 内存块耗尽了，咋整?

答案是：**走内存淘汰机制**。

内存淘汰机制

redis 内存淘汰机制有以下几个：

- noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了。
- **allkeys-lru**：当内存不足以容纳新写入数据时，在**键空间**中，移除最近最少使用的 key（这个是**最常用**的）。
- allkeys-random：当内存不足以容纳新写入数据时，在**键空间**中，随机移除某个 key，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的 key 给干掉啊。
- volatile-lru：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，移除最近最少使用的 key（这个一般不太合适）。
- volatile-random：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，**随机移除**某个 key。
- volatile-ttl：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，有**更早过期时间**的 key 优先移除。

手写一个 LRU 算法

你可以现场手写最原始的 LRU 算法，那个代码量太大了，似乎不太现实。

不求自己纯手工从底层开始打造出自己的 LRU，但是起码要知道如何利用已有的 JDK 数据结构实现一个 Java 版的 LRU。

java

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }
}
```


【面试题】如何保证 redis 的高并发和高可用？redis 的主从复制原理能介绍一下么？redis 的哨兵原理能介绍一下么？

面试官心理分析

其实问这个问题，主要是考考你，redis 单机能承载多高并发？如果单机扛不住如何扩容扛更多的并发？redis 会不会挂？既然 redis 会挂那怎么保证 redis 是高可用的？

其实针对的都是项目中你肯定要考虑的一些问题，如果你没考虑过，那确实你对生产系统中的问题思考太少。

面试题剖析

如果你用 redis 缓存技术的话，肯定要考虑如何用 redis 来加多台机器，保证 redis 是高并发的，还有就是如何让 redis 保证自己不是挂掉以后就直接死掉了，即 redis 高可用。

由于此节内容较多，因此，会分为两个小节进行讲解。

- redis 主从架构
- redis 基于哨兵实现高可用

redis 实现**高并发**主要依靠**主从架构**，一主多从，一般来说，很多项目其实就足够了，单主用来写入数据，单机几万 QPS，多从用来查询数据，多个从实例可以提供每秒 10w 的 QPS。

如果想要在实现高并发的同时，容纳大量的数据，那么就需要 redis 集群，使用 redis 集群之后，可以提供每秒几十万的读写并发。

redis 高可用，如果是做主从架构部署，那么加上哨兵就可以了，就可以实现，任何一个实例宕机，可以进行主备切换。

【面试题】 - redis 的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

面试官心理分析

redis 如果仅仅只是将数据缓存在内存里面，如果 redis 宕机了再重启，内存里的数据就全部都弄丢了啊。你必须得用 redis 的持久化机制，将数据写入内存的同时，异步的慢慢的将数据写入磁盘文件里，进行持久化。

如果 redis 宕机重启，自动从磁盘上加载之前持久化的一些数据就可以了，也许会丢失少许数据，但是至少不会将所有数据都弄丢。

这个其实一样，针对的都是 redis 的生产环境可能遇到的一些问题，就是 redis 要是挂了再重启，内存里的数据不就全丢了？能不能重启的时候把数据给恢复了？

面试题剖析

持久化主要是做灾难恢复、数据恢复，也可以归类到高可用的一个环节中去，比如你 redis 整个挂了，然后 redis 就不可用了，你要做的事情就是让 redis 变得可用，尽快变得可用。

重启 redis，尽快让它对外提供服务，如果没做数据备份，这时候 redis 启动了，也不可用啊，数据都没了。

很可能说，大量的请求过来，缓存全部无法命中，在 redis 里根本找不到数据，这个时候就死定了，出现**缓存雪崩**问题。所有请求没有在 redis 命中，就会去 mysql 数据库这种数据源头中去找，一下子 mysql 承接高并发，然后就挂了...

如果你把 redis 持久化做好，备份和恢复方案做到企业级的程度，那么即使你的 redis 故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务。

redis 持久化的两种方式

- RDB：RDB 持久化机制，是对 redis 中的数据执行**周期性**的持久化。
- AOF：AOF 机制对每条写入命令作为日志，以 `append-only` 的模式写入一个日志文件中，在 redis 重启的时候，可以通过**回放** AOF 日志中的写入指令来重新构建整个数据集。

通过 RDB 或 AOF，都可以将 redis 内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云等云服务。

如果 redis 挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动 redis，redis 就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务。

如果同时使用 RDB 和 AOF 两种持久化机制，那么在 redis 重启的时候，会使用 **AOF** 来重新构建数据，因为 AOF 中的**数据更加完整**。

RDB 优缺点

- RDB 会生成多个数据文件，每个数据文件都代表了某一个时刻中 redis 的数据，这种多个数据文件的方式，**非常适合做冷备**，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说 Amazon 的 S3 云服务上去，在国内可以是阿里云的 ODPS 分布式存储上，以预定好的备份策略来定期备份 redis 中的数据。

- RDB 对 redis 对外提供的读写服务，影响非常小，可以让 redis **保持高性能**，因为 redis 主进程只需要 fork 一个子进程，让子进程执行磁盘 IO 操作来进行 RDB 持久化即可。

- 相对于 AOF 持久化机制来说，直接基于 RDB 数据文件来重启和恢复 redis 进程，更加快速。

- 如果想要在 redis 故障时，尽可能少的丢失数据，那么 RDB 没有 AOF 好。一般来说，RDB 数据快照文件，都是每隔 5 分钟，或者更长时间生成一次，这个时候就得接受一旦 redis 进程宕机，那么会丢失最近 5 分钟的数据。

- RDB 每次在 fork 子进程来执行 RDB 快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒。

AOF 优缺点

- AOF 可以更好的保护数据不丢失，一般 AOF 会每隔 1 秒，通过一个后台线程执行一次 `fsync` 操作，最多丢失 1 秒钟的数据。
- AOF 日志文件以 `append-only` 模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复。
- AOF 日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在 `rewrite` log 的时候，会对其中的指令进行压缩，创建出一份需要恢复数据的最小日志出来。在创建新日志文件的时候，老的日志文件还是照常写入。当新的 merge 后的日志文件 ready 的时候，再交换新老日志文件即可。
- AOF 日志文件的命令通过非常可读的方式进行记录，这个特性非常**适合做灾难性的误删除的紧急恢复**。比如某人不小心用 `flushall` 命令清空了所有数据，只要这个时候后台 `rewrite` 还没有发生，那么就可以立即拷贝 AOF 文件，将最后一条 `flushall` 命令给删了，然后再将该 `AOF` 文件放回去，就可以通过恢复机制，自动恢复所有数据。
- 对于同一份数据来说，AOF 日志文件通常比 RDB 数据快照文件更大。
- AOF 开启后，支持的写 QPS 会比 RDB 支持的写 QPS 低，因为 AOF 一般会配置成每秒 `fsync` 一次日志文件，当然，每秒一次 `fsync` ，性能也还是很高的。（如果实时写入，那么 QPS 会大降，redis 性能会大大降低）
- 以前 AOF 发生过 bug，就是通过 AOF 记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似 AOF 这种较为复杂的基于命令日志 / merge / 回放的方式，比基于 RDB 每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有 bug。不过 AOF 就是为了避免 rewrite 过程导致的 bug，因此每次 rewrite 并不是基于旧的指令日志进行 merge 的，而是**基于当时内存中的数据进行指令的重新构建**，这样健壮性会好很多。

RDB 和 AOF 到底该如何选择

- 不要仅仅使用 RDB，因为那样会导致你丢失很多数据；
- 也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug；
- redis 支持同时开启开启两种持久化方式，我们可以综合使用 AOF 和 RDB 两种持久化机制，用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。

【面试题】 - redis 集群模式的工作原理能说一下么？在集群模式下，redis 的 key 是如何寻址的？分布式寻址都有哪些算法？了解一致性 hash 算法吗？

面试官心理分析

在前几年，redis 如果要搞几个节点，每个节点存储一部分的数据，得**借助一些中间件**来实现，比如说有 `codis` ，或者 `twemproxy` ，都有。有一些 redis 中间件，你读写 redis 中间件，redis 中间件负责将你的数据分布式存储在多台机器上的 redis 实例中。

这两年，redis 不断在发展，redis 也不断有新的版本，现在的 redis 集群模式，可以做到在多台机器上，部署多个 redis 实例，每个实例存储一部分的数据，同时每个 redis 主实例可以挂 redis 从实例，自动确保说，如果 redis 主实例挂了，会自动切换到 redis 从实例上来。

现在 redis 的新版本，大家都是用 redis cluster 的，也就是 redis 原生支持的 redis 集群模式，那么面试官肯定会就 redis cluster 对你来个几连炮。要是你没用过 redis cluster，正常，以前很多人用 codis 之类的客户端来支持集群，但是起码你得研究一下 redis cluster 吧。

如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个 G，单机就足够了，可以使用 replication，一个 master 多个 slaves，要几个 slave 跟你要求的读吞吐量有关，然后自己搭建一个 sentinel 集群去保证 redis 主从架构的高可用性。

redis cluster，主要是针对**海量数据+高并发+高可用**的场景。redis cluster 支撑 N 个 redis master node，每个 master node 都可以挂载多个 slave node。这样整个 redis 就可以横向扩容了。如果你要支撑更大数据量的缓存，那就横向扩容更多的 master 节点，每个 master 节点就能存放更多的数据了。

面试题剖析

redis cluster 介绍

- 自动将数据进行分片，每个 master 上放一部分数据
- 提供内置的高可用支持，部分 master 不可用时，还是可以继续工作的

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

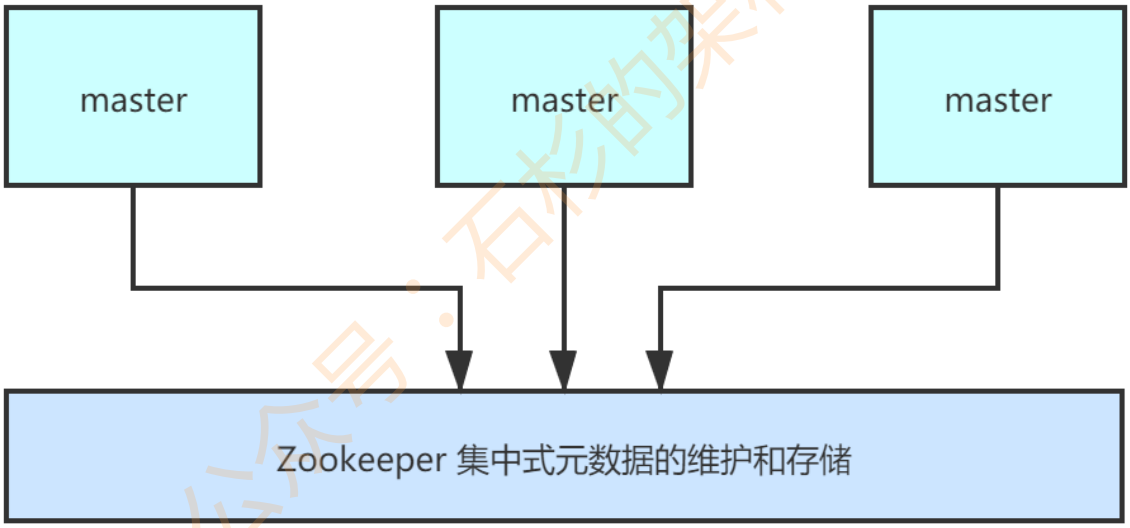
16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议， `gossip` 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

节点间的内部通信机制

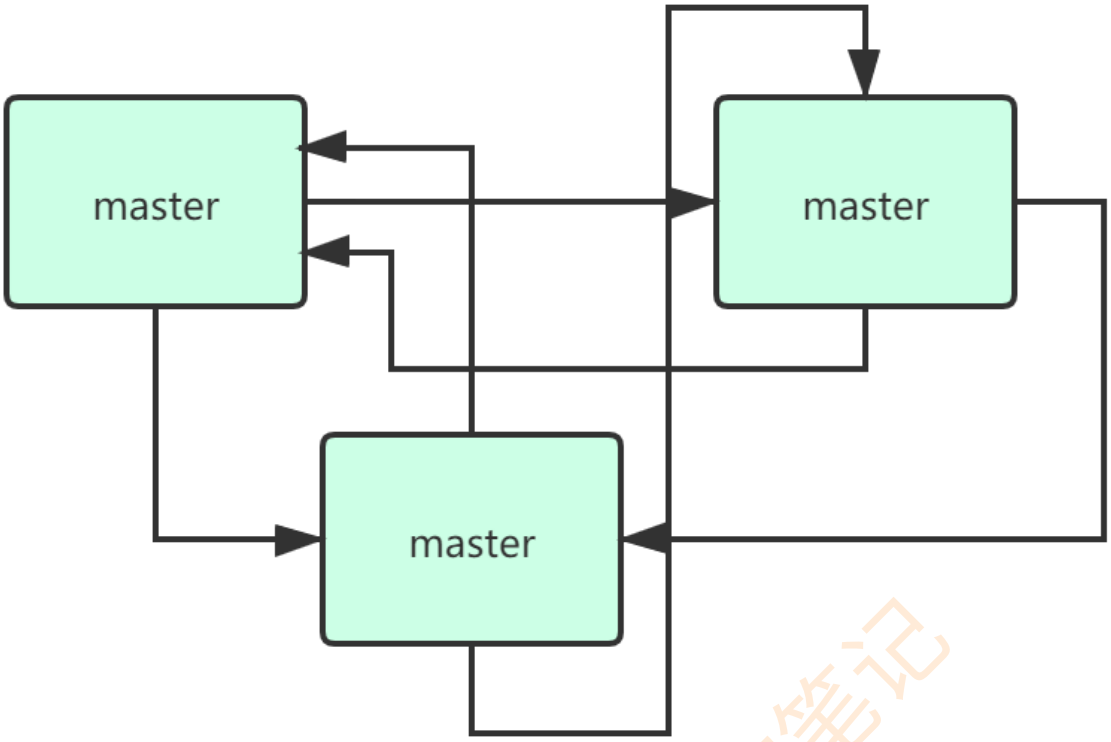
基本通信原理

集群元数据的维护有两种方式：集中式、Gossip 协议。redis cluster 节点间采用 gossip 协议进行通信。

集中式是将集群元数据（节点信息、故障等等）几种存储在某个节点上。集中式元数据集中存储的一个典型代表，就是大数据领域的 `storm` 。它是分布式的大数据实时计算引擎，是集中式的元数据存储的结构，底层基于 zookeeper（分布式协调的中间件）对所有元数据进行存储维护。



redis 维护集群元数据采用另一个方式， `gossip` 协议，所有节点都持有一份元数据，不同的节点如果出现了元数据的变更，就不断将元数据发送给其它的节点，让其它节点也进行元数据的变更。



集中式的**好处**在于，元数据的读取和更新，时效性非常好，一旦元数据出现了变更，就立即更新到集中式的存储中，其它节点读取的时候就可以感知到；**不好**在于，所有的元数据的更新压力全部集中在一个地方，可能会导致元数据的存储有压力。

gossip 好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续打到所有节点上去更新，降低了压力；不好在于，元数据的更新有延时，可能导致集群中的一些操作会有一些滞后。

- 10000 端口：每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如 7001，那么用于节点间通信的就是 17001 端口。每个节点每隔一段时间都会往另外几个节点发送 `ping` 消息，同时其它几个节点接收到 `ping` 之后返回 `pong` 。
- 交换的信息：信息包括故障信息，节点的增加和删除，hash slot 信息等等。

gossip 协议

gossip 协议包含多种消息，包含 `ping` ，`pong` ，`meet` ，`fail` 等等。

- meet：某个节点发送 meet 给新加入的节点，让新节点加入集群中，然后新节点就会开始与其它节点进行通信。

```bash
redis-trib.rb add-node
```

其实内部就是发送了一个 gossip meet 消息给新加入的节点，通知那个节点去加入我们的集群。

- ping：每个节点都会频繁给其它节点发送 ping，其中包含自己的状态还有自己维护的集群元数据，互相通过 ping 交换元数据。
- pong：返回 ping 和 meeet，包含自己的状态和其它信息，也用于信息广播和更新。
- fail：某个节点判断另一个节点 fail 之后，就发送 fail 给其它节点，通知其它节点说，某个节点宕机啦。

ping 消息深入

ping 时要携带一些元数据，如果很频繁，可能会加重网络负担。

每个节点每秒会执行 10 次 ping，每次会选择 5 个最久没有通信的其它节点。当然如果发现某个节点通信延时达到了 `cluster_node_timeout / 2` ，那么立即发送 ping，避免数据交换延时过长，落后的时间太长了。比如说，两个节点之间都 10 分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题。所以 `cluster_node_timeout` 可以调节，如果调得比较大，那么会降低 ping 的频率。

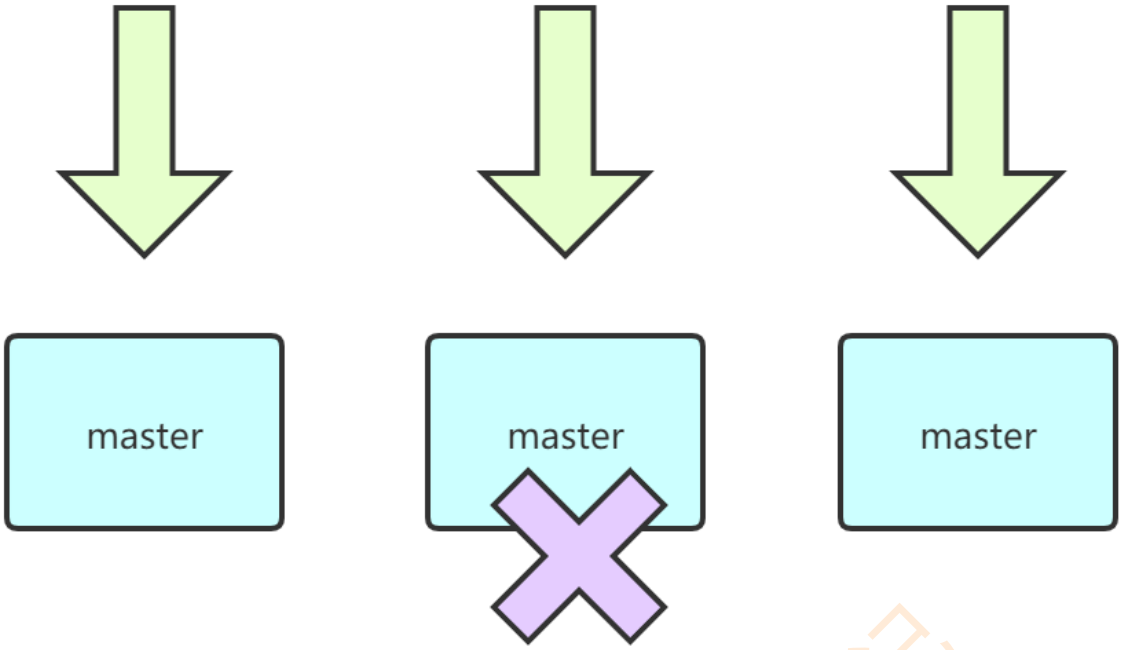
每次 ping，会带上自己节点的信息，还有就是带上 1/10 其它节点的信息，发送出去，进行交换。至少包含 `3` 个其它节点的信息，最多包含 `总节点数减 2` 个其它节点的信息。

分布式寻址算法

- hash 算法（大量缓存重建）
- 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
- redis cluster 的 hash slot 算法

hash 算法

来了一个 key，首先计算 hash 值，然后对节点数取模。然后打在不同的 master 节点上。一旦某一个 master 节点宕机，所有请求过来，都会基于最新的剩余 master 节点数去取模，尝试去取数据。这会导致**大部分的请求过来，全部无法拿到有效的缓存**，导致大量的流量涌入数据库。



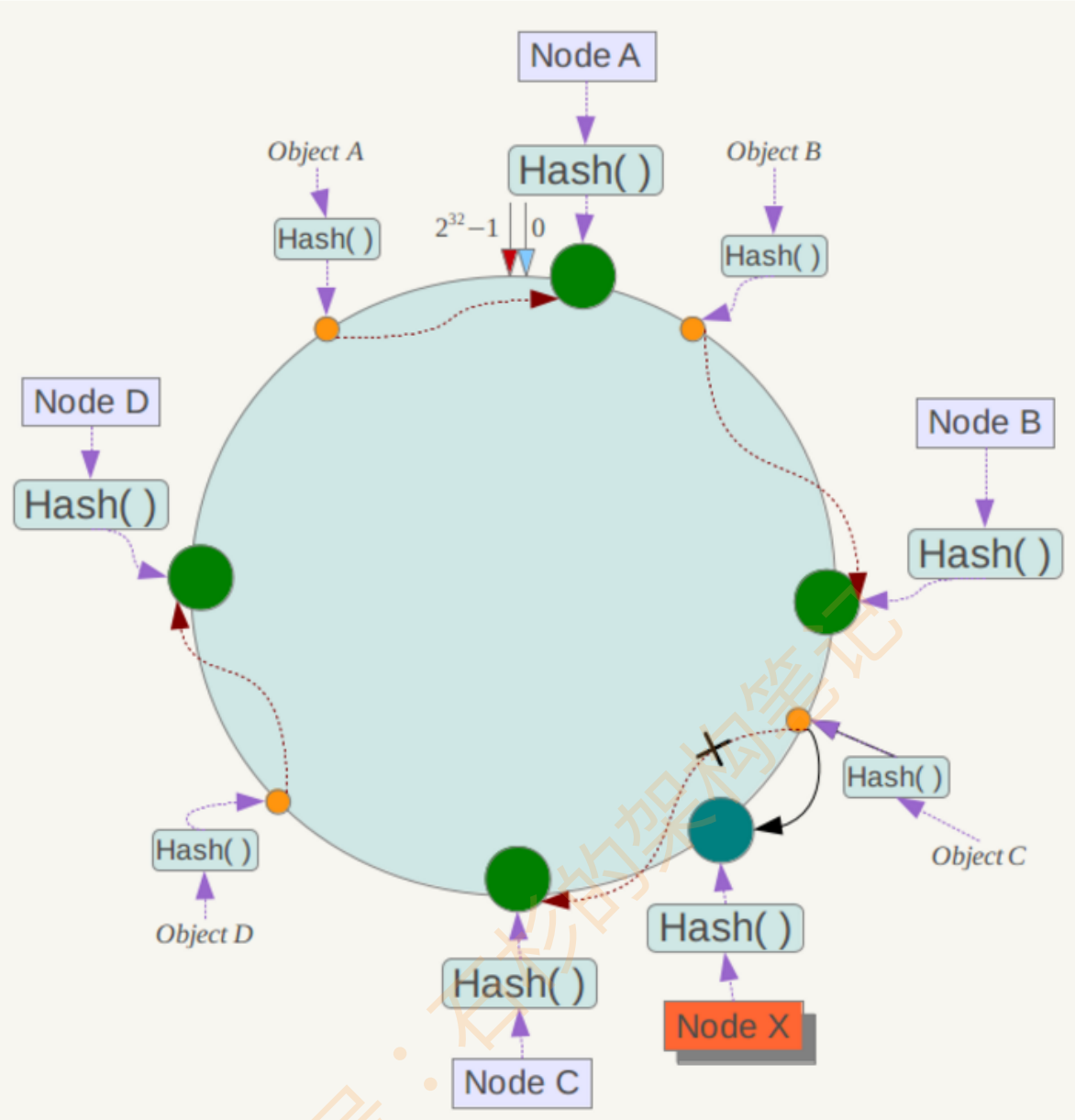
一致性 hash 算法

一致性 hash 算法将整个 hash 值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织，下一步将各个 master 节点（使用服务器的 ip 或主机名）进行 hash。这样就能确定每个节点在其哈希环上的位置。

来了一个 key，首先计算 hash 值，并确定此数据在环上的位置，从此位置沿环**顺时针“行走”**，遇到的第一个 master 节点就是 key 所在位置。

在一致性哈希算法中，如果一个节点挂了，受影响的数据仅仅是此节点到环空间前一个节点（沿着逆时针方向行走遇到的第一个节点）之间的数据，其它不受影响。增加一个节点也同理。

燃鹅，一致性哈希算法在节点太少时，容易因为节点分布不均匀而造成**缓存热点**的问题。为了解决这种热点问题，一致性 hash 算法引入了虚拟节点机制，即对每一个节点计算多个 hash，每个计算结果位置都放置一个虚拟节点。这样就实现了数据的均匀分布，负载均衡。

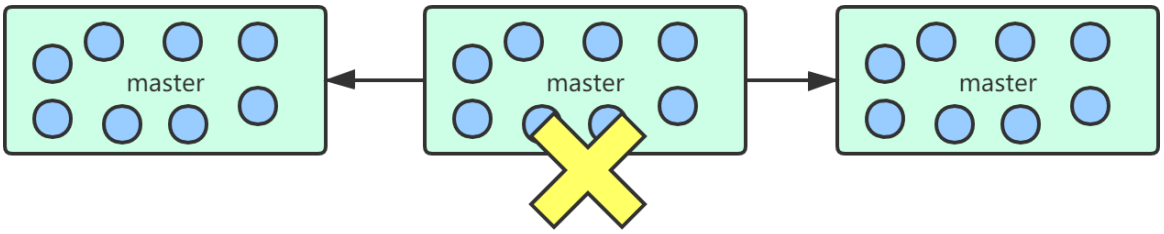


redis cluster 的 hash slot 算法

redis cluster 有固定的 `16384` 个 hash slot，对每个 `key` 计算 `CRC16` 值，然后对 `16384` 取模，可以获取 key 对应的 hash slot。

redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有 5000 多个 hash slot。hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master 的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动 hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 `hash tag` 来实现。

任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。



redis cluster 的高可用与主备切换原理

redis cluster 的高可用的原理，几乎跟哨兵是类似的。

判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是 `pfail` ，**主观宕机**。如果多个节点都认为另外一个节点宕机了，那么就是 `fail` ，**客观宕机**，跟哨兵的原理几乎一样，sdown，odown。

在 `cluster-node-timeout` 内，某个节点一直没有返回 `pong` ，那么就被认为 `pfail` 。

如果一个节点认为某个节点 `pfail` 了，那么会在 `gossip ping` 消息中， `ping` 给其他节点，如果**超过半数**的节点都认为 `pfail` 了，那么就会变成 `fail` 。

从节点过滤

对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。

检查每个 slave node 与 master node 断开连接的时间，如果超过了 `cluster-node-timeout * cluster-slave-validity-factor` ，那么就**没有资格**切换成 `master` 。

从节点选举

每个从节点，都根据自己对 master 复制数据的 offset，来设置一个选举时间，offset 越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举。

所有的 master node 开始 slave 选举投票，给要进行选举的 slave 进行投票，如果大部分 master node `(N/2 + 1)` 都投票给了某个从节点，那么选举通过，那个从节点可以切换成 master。

从节点执行主备切换，从节点切换为主节点。

与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，redis cluster 功能强大，直接集成了 replication 和 sentinel 的功能。

【面试题】 了解什么是 redis 的雪崩、穿透和击穿？redis 崩溃之后会怎么样？系统该如何应对这种情况？如何处理 redis 的穿透？

面试官心理分析

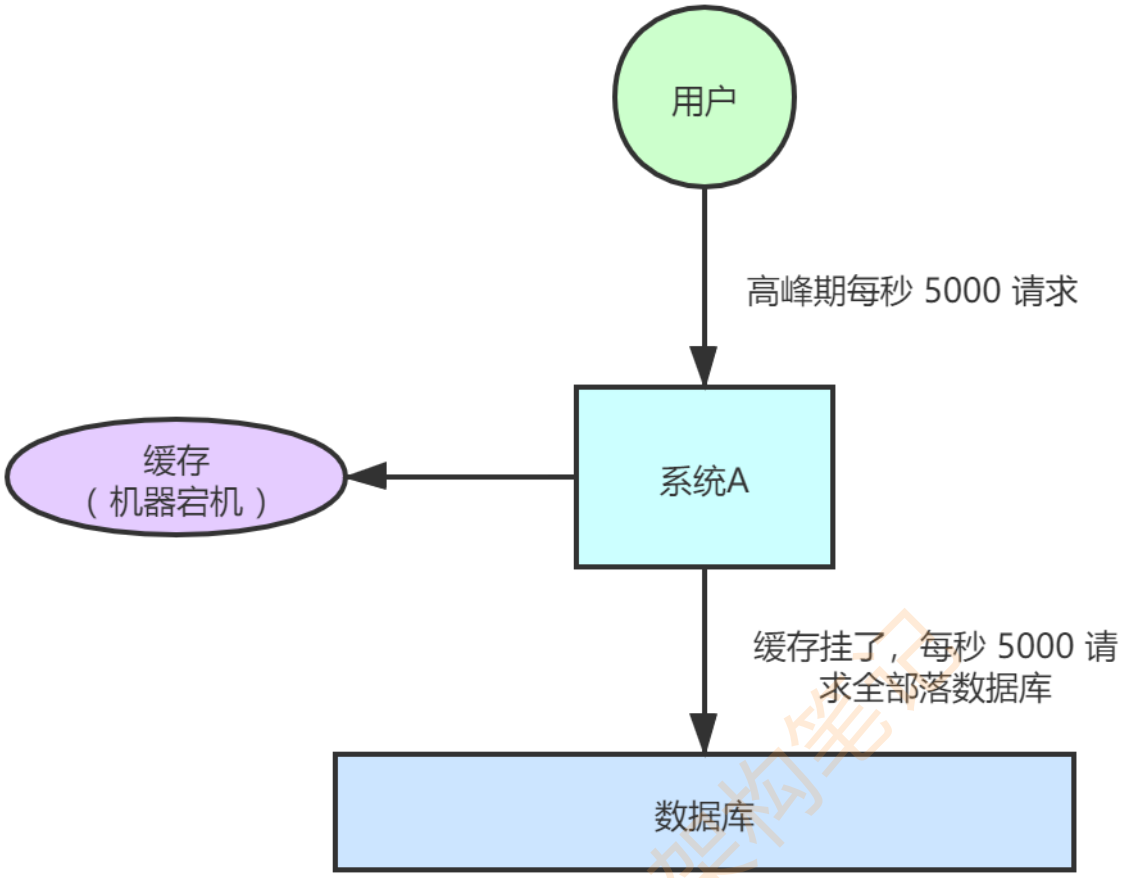
其实这是问到缓存必问的，因为缓存雪崩和穿透，是缓存最大的两个问题，要么不出现，一旦出现就是致命性的问题，所以面试官一定会问你。

面试题剖析

缓存雪崩

对于系统 A，假设每天高峰期每秒 5000 个请求，本来缓存在高峰期可以扛住每秒 4000 个请求，但是缓存机器意外发生了全盘宕机。缓存挂了，此时 1 秒 5000 个请求全部落数据库，数据库必然扛不住，它会报一下警，然后就挂了。此时，如果没有采用什么特别的方案来处理这个故障，DBA 很着急，重启数据库，但是数据库立马又被新的流量给打死了。

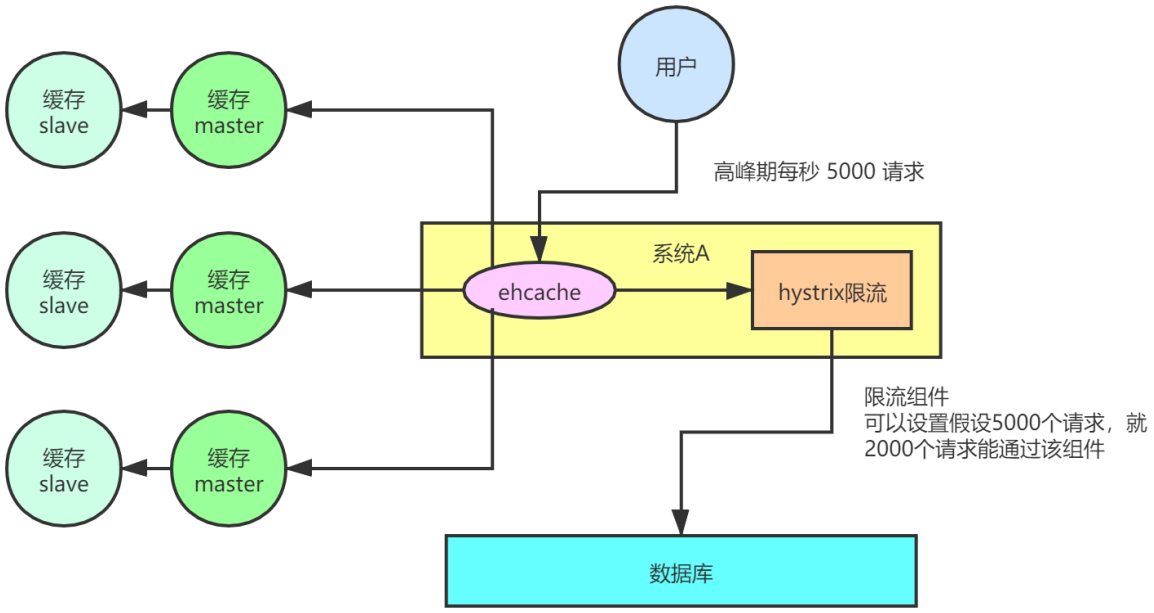
这就是缓存雪崩。



大约在 3 年前，国内比较知名的一个互联网公司，曾因为缓存事故，导致雪崩，后台系统全部崩溃，事故从当天下午持续到晚上凌晨 3~4 点，公司损失了几千万。

缓存雪崩的事前事中事后的解决方案如下。

- 事前：redis 高可用，主从+哨兵，redis cluster，避免全盘崩溃。
- 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
- 事后：redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。



用户发送一个请求，系统 A 收到请求后，先查本地 ehcache 缓存，如果没查到再查 redis。如果 ehcache 和 redis 都没有，再查数据库，将数据库中的结果，写入 ehcache 和 redis 中。

限流组件，可以设置每秒的请求，有多少能通过组件，剩余的未通过的请求，怎么办？**走降级**！可以返回一些默认的值，或者友情提示，或者空白的值。

好处：

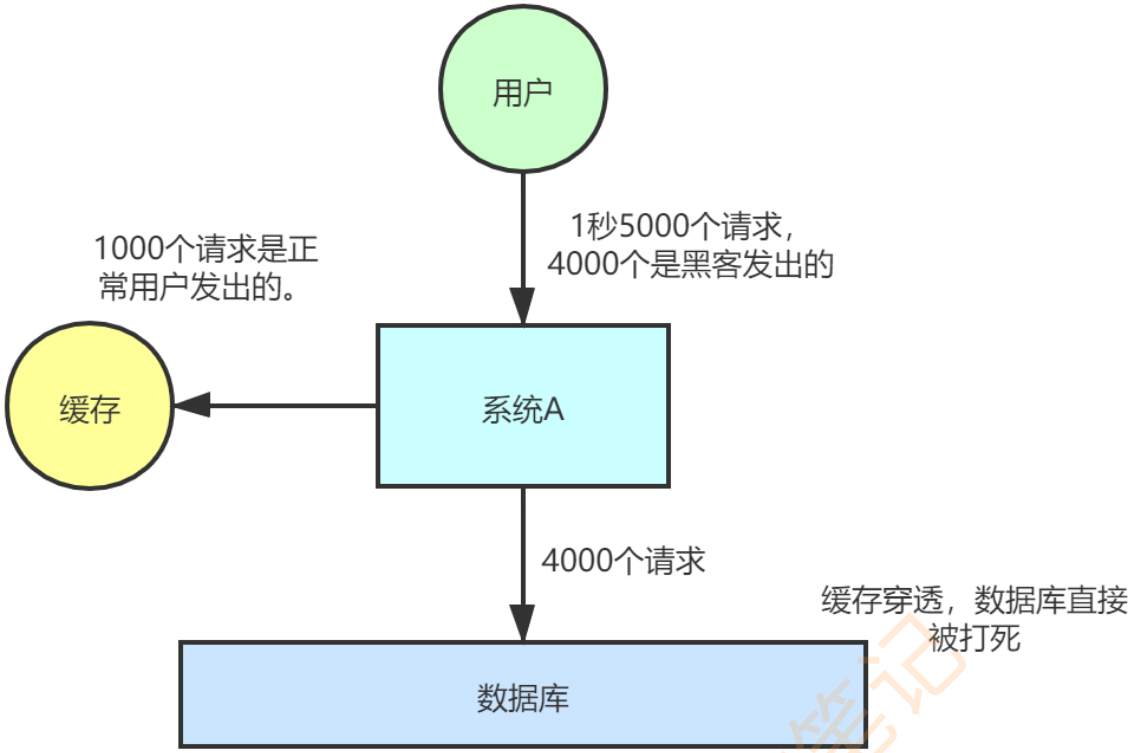
- 数据库绝对不会死，限流组件确保了每秒只有多少个请求能通过。
- 只要数据库不死，就是说，对用户来说，2/5 的请求都是可以被处理的。
- 只要有 2/5 的请求可以被处理，就意味着你的系统没死，对用户来说，可能就是点击几次刷不出来页面，但是多点几次，就可以刷出来一次。

缓存穿透

对于系统A，假设一秒 5000 个请求，结果其中 4000 个请求是黑客发出的恶意攻击。

黑客发出的那 4000 个攻击，缓存中查不到，每次你去数据库里查，也查不到。

举个栗子。数据库 id 是从 1 开始的，结果黑客发过来的请求 id 全部都是负数。这样的话，缓存中不会有，请求每次都“**视缓存于无物**”，直接查询数据库。这种恶意攻击场景的缓存穿透就会直接把数据库给打死。



解决方式很简单，每次系统 A 从数据库中只要没查到，就写一个空值到缓存里去，比如 `set -999 UNKNOWN` 。然后设置一个过期时间，这样的话，下次有相同的 key 来访问的时候，在缓存失效之前，都可以直接从缓存中取数据。

缓存击穿

缓存击穿，就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。

解决方式也很简单，可以将热点数据设置为永远不过期；或者基于 redis or zookeeper 实现互斥锁，等待第一个请求构建完缓存之后，再释放锁，进而其它请求才能通过该 key 访问数据。

【面试题】 - 如何保证缓存与数据库的双写一致性?

面试官心理分析

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题?

面试题剖析

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统**不是严格要求** “缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：**读请求和写请求串行化**，串到一个**内存队列**里去。

串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

Cache Aside Pattern

最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。

- 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
- 更新的时候，**先更新数据库，然后再删除缓存**。

为什么是删除缓存，而不是更新缓存?

原因很简单，很多时候，在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

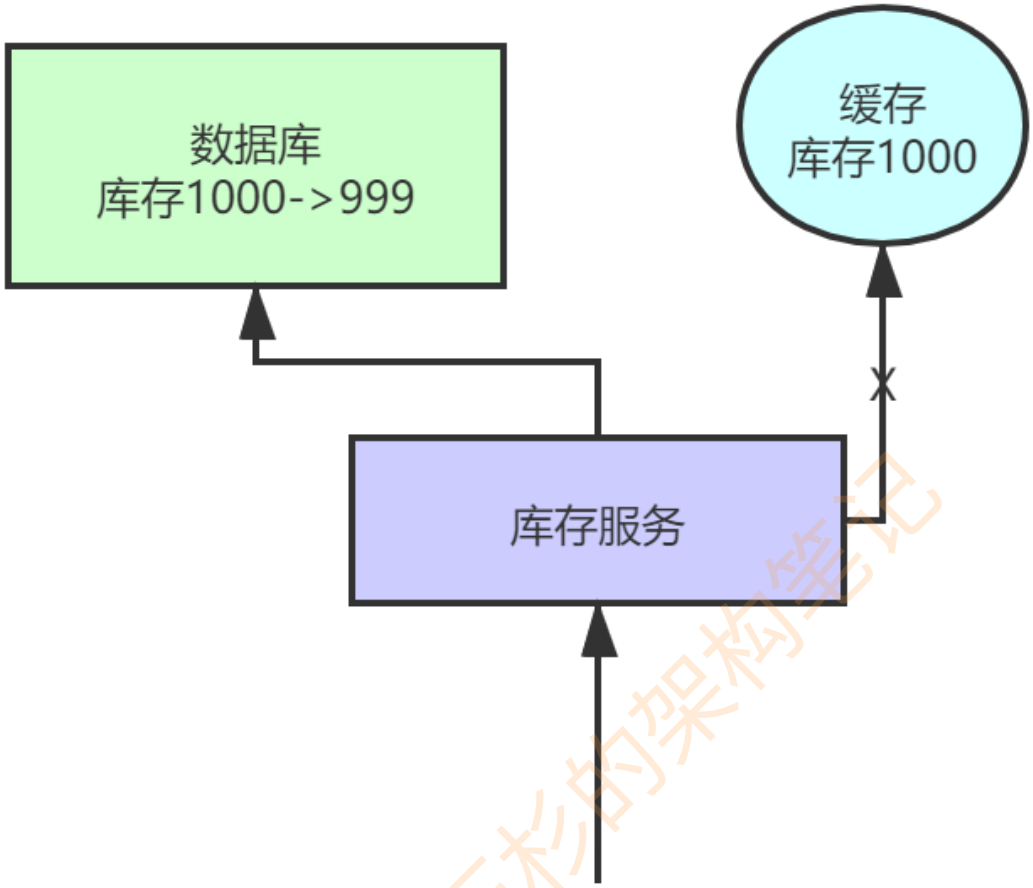
另外更新缓存的代价有时候是很高的。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于**比较复杂的缓存数据计算的场景**，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，**这个缓存到底会不会被频繁访问到?**

举个栗子，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有**大量的冷数据**。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。**用到缓存才去算缓存。**

其实删除缓存，而不是更新缓存，就是一个 lazy 计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像 mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的 list，没有必要说每次查询部门，都里面的 1000 个员工的数据也同时查出来啊。80% 的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询 1000 个员工。

最初级的缓存不一致问题及解决方案

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。



解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据**，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样了...

为什么上亿流量高并发场景下，缓存会出现这个问题?

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题。其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就 1 万次，那么很少的情况下，会出现刚才描述的那种不一致的场景。但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就**可能会出现上述的数据库+缓存不一致的情况**。

解决方案如下：

更新数据的时候，根据**数据的唯一标识**，将操作路由之后，发送到一个 jvm 内部队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个 jvm 内部队列中。

一个队列对应一个工作线程，每个工作线程**串行**拿到对应的操作，然后一条一条的执行。这样的话，一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，没有读到缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这里有一个**优化点**，一个队列中，其实**多个更新缓存请求串在一起是没意义的**，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中。

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。

高并发的场景下，该解决方案要注意的问题：

- 读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回。

该解决方案，最大的风险点在于说，**可能数据更新很频繁**，导致队列中积压了大量更新操作在里面，然后**读请求会发生大量的超时**，最后导致大量的请求直接走数据库。务必通过一些模拟真实的测试，看看更新数据的频率是怎样的。

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要**部署多个服务**，每个服务分摊一些数据的更新操作。如果一个内存队列里居然会挤压 100 个商品的库存修改操作，每隔库存修改操作要耗费 10ms 去完成，那么最后一个商品的读请求，可能等待 10 * 100 = 1000ms = 1s 后，才能得到数据，这个时候就导致**读请求的长时阻塞**。

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会 hang 多少时间，如果读请求在 200ms 返回，如果你计算过后，哪怕是最繁忙的时候，积压 10 个更新操作，最多等待 200ms，那还可以的。

如果一个内存队列中可能积压的更新操作特别多，那么你就要**加机器**，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来**实际粗略测算一下**。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

- 读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时 hang 在服务上，看服务能不能扛的住，需要多少机器才能扛住最大的极限情况的峰值。

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大。

- 多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须**保证**说，执行数据更新操作，以及执行缓存更新操作的请求，都通过 Nginx 服务器**路由到相同的服务实例上**。

比如说，对同一个商品的读写请求，全部路由到同一台机器上。可以自己去做服务间的按照某个请求参数的 hash 路由，也可以用 Nginx 的 hash 路由功能等等。

- 热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能会造成某台机器的压力过大。就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以其实要根据业务系统去看，如果更新频率不是太高的话，这个问题的影响并不是特别大，但是的确可能某些机器的负载会高一些。

【面试题】 - redis 的并发竞争问题是什么？如何解决这个问题？了解 redis 事务的 CAS 方案吗？

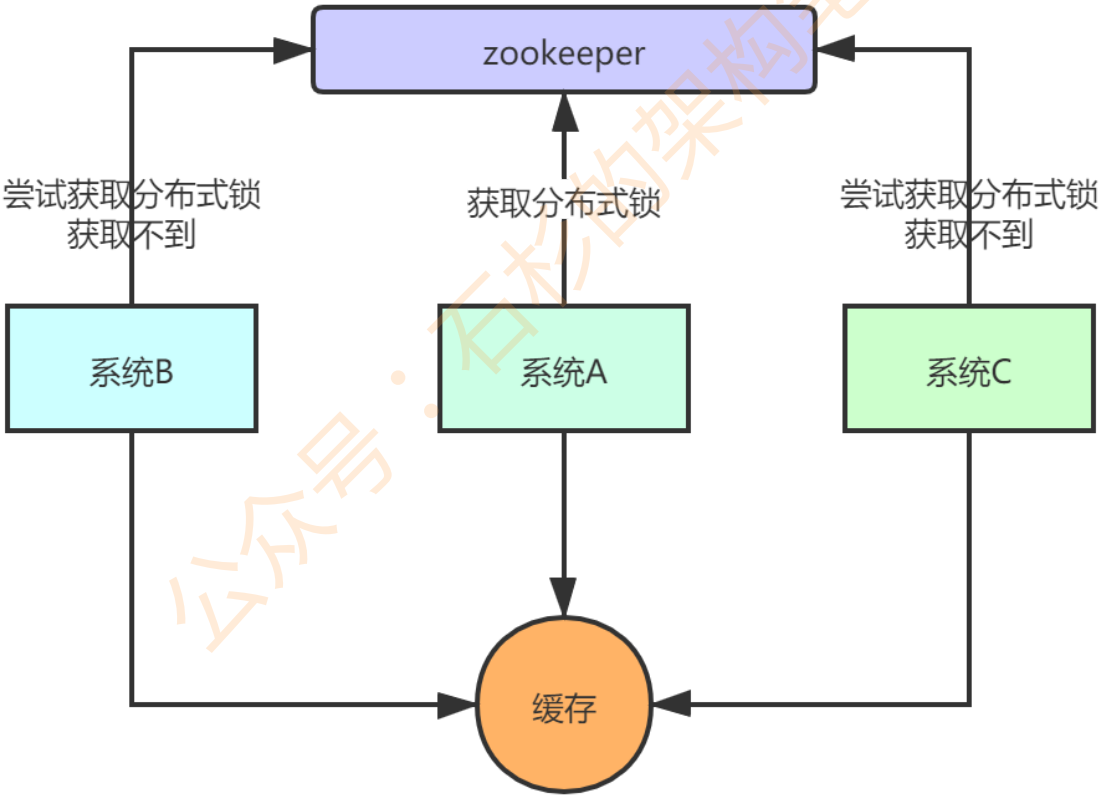
面试官心理分析

这个也是线上非常常见的一个问题，就是**多客户端同时并发写**一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

而且 redis 自己就有天然解决这个问题的 CAS 类的乐观锁方案。

面试题剖析

某个时刻，多个系统实例都去更新某个 key。可以基于 zookeeper 实现分布式锁。每个系统通过 zookeeper 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 key，别人都不允许读和写。



你要写入缓存的数据，都是从 mysql 里查出来的，都得写入 mysql 中，写入 mysql 中的时候必须保存一个时间戳，从 mysql 查出来的时候，时间戳也查出来。

每次要**写之前，先判断**一下当前这个 value 的时间戳是否比缓存里的 value 的时间戳要新。如果是的话，那么可以写，否则，就不能用旧的数据覆盖新的数据。

【面试题】 - 生产环境中的 redis 是怎么部署的?

面试官心理分析

看看你了解不了解你们公司的 redis 生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的 redis 是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上 redis 给几个 G 的内存？设置了哪些参数？压测后你们 redis 集群承载多少 QPS？

兄弟，这些你必须是门儿清的，否则你确实是没好好思考过。

面试题剖析

redis cluster，10 台机器，5 台机器部署了 redis 主实例，另外 5 台机器部署了 redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求/s。

机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 redis 进程的是10g内存，一般线上生产环境，redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

5 台机器对外提供读写，一共有 50g 内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis 从实例会自动变成主实例继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。

【面试题】 - 为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

面试官心理分析

其实这块肯定是扯到**高并发**了，因为分库分表一定是为了**支撑高并发、数据量大**两个问题的。而且现在说实话，尤其是互联网类的公司面试，基本上都会来这么一下，分库分表如此普遍的技术问题，不问实在是不行，而如果你不知道那也实在是说不过去！

面试题剖析

为什么要分库分表？（设计高并发系统的时候，数据库层面该如何设计？）

说白了，分库分表是两回事儿，大家可别搞混了，可能是光分库不分表，也可能是光分表不分库，都有可能。

我先给大家抛出来一个场景。

假如我们现在是一个小创业公司（或者是一个 BAT 公司刚兴起的一个新部门），现在注册用户就 20 万，每天活跃用户就 1 万，每天单表数据量就 1000，然后高峰期每秒钟并发请求最多就 10。天，就这种系统，随便找一个有几年工作经验的，然后带几个刚培训出来的，随便干干都可以。

结果没想到我们运气居然这么好，碰上个 CEO 带着我们走上了康庄大道，业务发展迅猛，过了几个月，注册用户数达到了 2000 万！每天活跃用户数 100 万！每天单表数据量 10 万条！高峰期每秒最大请求达到 1000！同时公司还顺带着融资了两轮，进账了几个亿人民币啊！公司估值达到了惊人的几亿美金！这是小独角兽的节奏！

好吧，没事，现在大家感觉压力已经有点大了，为啥呢？因为每天多 10 万条数据，一个月就多 300 万条数据，现在咱们单表已经几百万数据了，马上就破千万了。但是勉强还能撑着。高峰期请求现在是 1000，咱们线上部署了几台机器，负载均衡搞了一下，数据库撑 1000QPS 也还凑合。但是大家现在开始感觉有点担心了，接下来咋整呢……

再接下来几个月，我的天，CEO 太牛逼了，公司用户数已经达到 1 亿，公司继续融资几十亿人民币啊！公司估值达到了惊人的几十亿美金，成为了国内今年最牛逼的明星创业公司！天，我们太幸运了。

但是我们同时也是不幸的，因为此时每天活跃用户数上千万，每天单表新增数据多达 50 万，目前一个表总数据量都已经达到了两三千万了！扛不住啊！数据库磁盘容量不断消耗掉！高峰期并发达到惊人的 `5000~8000` ！别开玩笑了，哥。我跟你保证，你的系统支撑不到现在，已经挂掉了！

好吧，所以你看到这里差不多就理解分库分表是怎么回事儿了，实际上这是跟着你的公司业务发展走的，你公司业务发展越好，用户就越多，数据量越大，请求量越大，那你单个数据库一定扛不住。

分表

比如你单表都几千万数据了，你确定你能扛住么？绝对不行，**单表数据量太大**，会极大影响你的 sql **执行的性能**，到了后面你的 sql 可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。

分表是啥意思？就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户 id 来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。这样可以控制每个表的数据量在可控的范围内，比如每个表就固定在 200 万以内。

分库

分库是啥意思？就是你一个库一般我们经验而言，最多支撑到并发 2000，一定要扩容了，而且一个健康的单库并发值你最好保持在每秒 1000 左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。

这就是所谓的**分库分表**，为啥要分库分表？你明白了吧。

#	分库分表前	分库分表后
并发支撑情况	MySQL 单机部署，扛不住高并发	MySQL从单机到多机，能承受的并发增加了多倍
磁盘使用情况	MySQL 单机磁盘容量几乎撑满	拆分为多个库，数据库服务器磁盘使用率大大降低
SQL 执行性能	单表数据量太大，SQL 越跑越慢	单表数据量减少，SQL 执行效率明显提升

用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？

这个其实就是看看你了解哪些分库分表的中间件，各个中间件的优缺点是啥？然后你用过哪些分库分表的中间件。

比较常见的包括：

- Cobar
- TDDL
- Atlas
- Sharding-jdbc
- Mycat

Cobar

阿里 b2b 团队开发和开源的，属于 proxy 层方案，就是介于应用服务器和数据库服务器之间。应用程序通过 JDBC 驱动访问 Cobar 集群，Cobar 根据 SQL 和分库规则对 SQL 做分解，然后分发到 MySQL 集群不同的数据库实例上执行。早些年还可以用，但是最近几年都没更新了，基本没啥人用，差不多算是被抛弃的状态吧。而且不支持读写分离、存储过程、跨库 join 和分页等操作。

TDDL

淘宝团队开发的，属于 client 层方案。支持基本的 crud 语法和读写分离，但不支持 join、多表查询等语法。目前使用的也不多，因为还依赖淘宝的 diamond 配置管理系统。

Atlas

360 开源的，属于 proxy 层方案，以前是有一些公司在用的，但是确实有一个很大的问题就是社区最新的维护都在 5 年前了。所以，现在用的公司基本也很少了。

Sharding-jdbc

当当开源的，属于 client 层方案，目前已经更名为 `ShardingSphere` （后文所提到的 `Sharding-jdbc` ，等同于 `ShardingSphere` ）。确实之前用的还比较多一些，因为 SQL 语法支持也比较多，没有太多限制，而且截至 2019.4，已经推出到了 `4.0.0-RC1` 版本，支持分库分表、读写分离、分布式 id 生成、柔性事务（最大努力送达型事务、TCC 事务）。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从 2017 年一直到现在，是有不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，个人认为算是一个现在也**可以选择的方案**。

Mycat

基于 Cobar 改造的，属于 proxy 层方案，支持的功能非常完善，而且目前应该是非常火的而且不断流行的数据库中间件，社区很活跃，也有一些公司开始在用了。但是确实相比于 Sharding jdbc 来说，年轻一些，经历的锤炼少一些。

总结

综上，现在其实建议考量的，就是 Sharding-jdbc 和 Mycat，这两个都可以去考虑使用。

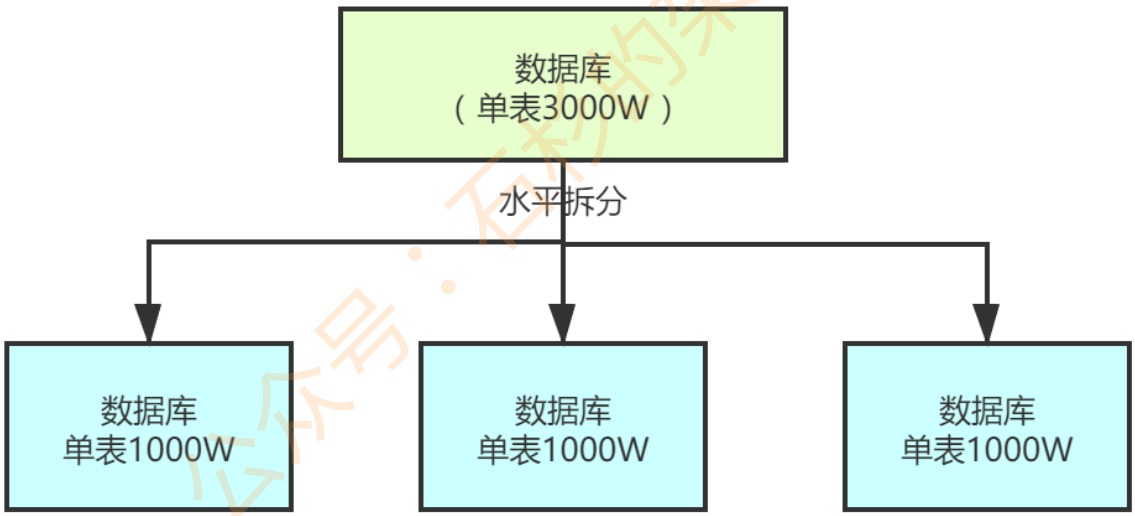
Sharding-jdbc 这种 client 层方案的**优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高**，但是如果遇到升级啥的需要各个系统都重新升级版本再发布，各个系统都需要**耦合** Sharding-jdbc 的依赖；

Mycat 这种 proxy 层方案的**缺点在于需要部署**，自己运维一套中间件，运维成本高，但是**好处在于对于各个项目是透明的**，如果遇到升级之类的都是自己中间件那里搞就行了。

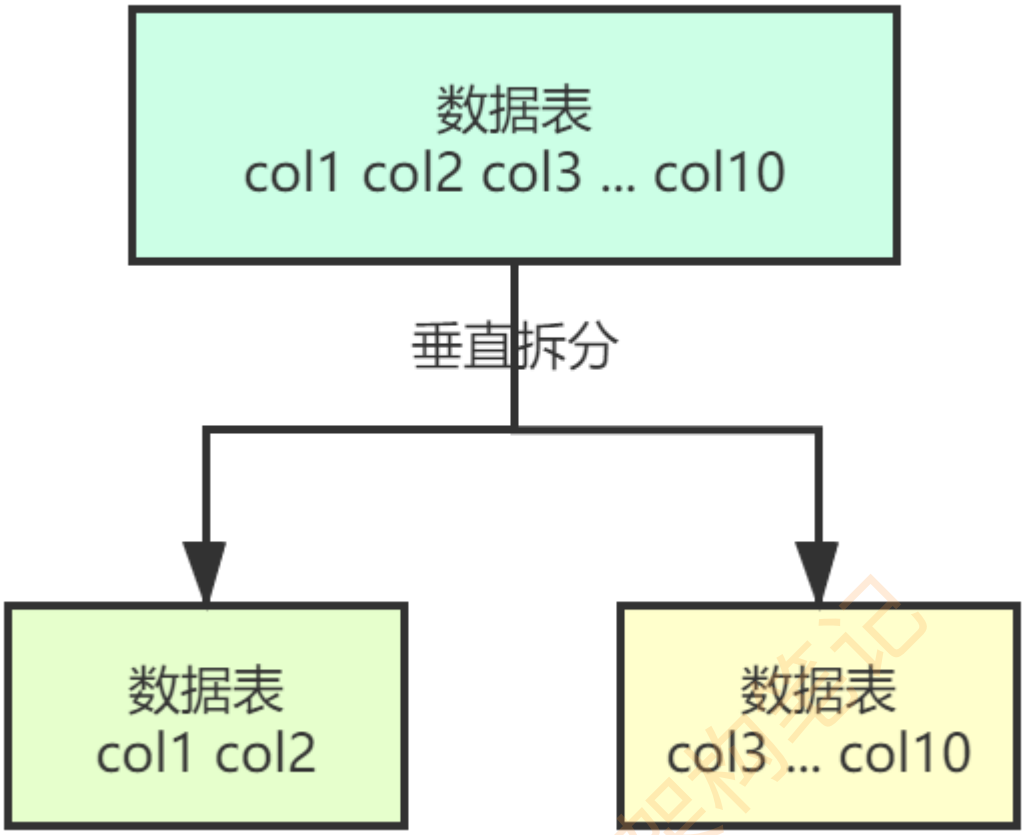
通常来说，这两个方案其实都可以选用，但是我个人建议中小型公司选用 Sharding-jdbc，client 层方案轻便，而且维护成本低，不需要额外增派人手，而且中小型公司系统复杂度会低一些，项目也没那么多；但是中大型公司最好还是选用 Mycat 这类 proxy 层方案，因为可能大公司系统和项目非常多，团队很大，人员充足，那么最好是专门弄个人来研究和维护 Mycat，然后大量项目直接透明使用即可。

你们具体是如何对数据库如何进行垂直拆分或水平拆分的?

水平拆分的意思，就是把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的，所有库表的数据加起来就是全部数据。水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来扛更高的并发，还有就是用多个库的存储容量来进行扩容。



垂直拆分的意思，就是**把一个有很多字段的表给拆分成多个表，或者是多个库上去**。每个库表的结构都不一样，每个库表都包含部分字段。一般来说，会**将较少的访问频率很高的字段放到一个表里去**，然后**将较多的访问频率很低的字段放到另外一个表里去**。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。



这个其实挺常见的，不一定我说，大家很多同学可能自己都做过，把一个大表拆开，订单表、订单支付表、订单商品表。

还有**表层面的拆分**，就是分表，将一个表变成 N 个表，就是**让每个表的数据量控制在一定范围内**，保证 SQL 的性能。否则单表数据量越大，SQL 性能就越差。一般是 200 万行左右，不要太多，但是也得看具体你怎么操作，也可能是 500 万，或者是 100 万。你的SQL越复杂，就最好让单表行数越少。

好了，无论分库还是分表，上面说的那些数据库中间件都是可以支持的。就是基本上那些中间件可以做到你分库分表之后，**中间件可以根据你指定的某个字段值**，比如说 userid，**自动路由到对应的库上去，然后再自动路由到对应的表里去**。

你就得考虑一下，你的项目里该如何分库分表？一般来说，垂直拆分，你可以在表层面来做，对一些字段特别多的表做一下拆分；水平拆分，你可以说是并发承载不了，或者是数据量太大，容量承载不了，你给拆了，按什么字段来拆，你自己想好；分表，你考虑一下，你如果哪怕是拆到每个库里去，并发和容量都 ok 了，但是每个库的表还是太大了，那么你就分表，将这个表分开，保证每个表的数据量并不是很大。

而且这儿还有两种**分库分表的方式**：

- 一种是按照 range 来分，就是每个库一段连续的数据，这个一般是按比如**时间范围**来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了。

- 或者是按照某个字段 hash 一下均匀分散，这个较为常用。

range 来分，好处在于说，扩容的时候很简单，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用 range，要看场景。

hash 分发，好处在于说，可以平均分配每个库的数据量和请求压力；坏处在于说扩容起来比较麻烦，会有一个数据迁移的过程，之前的数据需要重新计算 hash 值重新分配到不同的库或表。

【面试题】 - 现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上?

面试官心理分析

你看看，你现在已经明白为啥要分库分表了，你也知道常用的分库分表中间件了，你也设计好你们如何分库分表的方案了（水平拆分、垂直拆分、分表），那问题来了，你接下来该怎么把你那个单库单表的系统给迁移到分库分表上去?

所以这都是一环扣一环的，就是看你有没有全流程经历过这个过程。

面试题剖析

这个其实从 low 到高大上有好几种方案，我们都玩儿过，我都给你说一下。

停机迁移方案

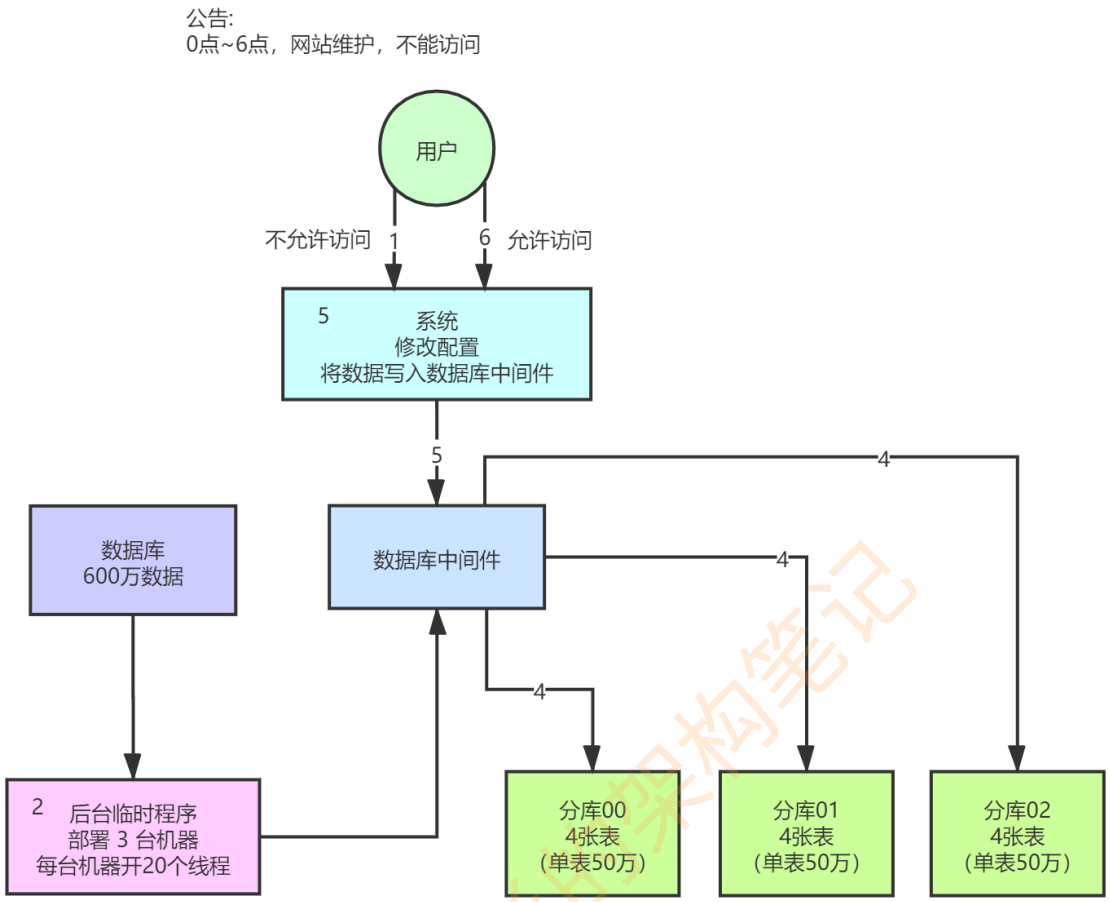
我先给你说一个最 low 的方案，就是很简单，大家伙儿凌晨 12 点开始运维，网站或者 app 挂个公告，说 0 点到早上 6 点进行运维，无法访问。

接着到 0 点停机，系统停掉，没有流量写入了，此时老的单库单表数据库静止了。然后你之前得写好一个**导数的一次性工具**，此时直接跑起来，然后将单库单表的数据哗哗哗读出来，写到分库分表里面去。

导数完了之后，就 ok 了，修改系统的数据库连接配置啥的，包括可能代码和 SQL 也许有修改，那你就用最新的代码，然后直接启动连到新的分库分表上去。

验证一下，ok了，完美，大家伸个懒腰，看看看凌晨 4 点钟的北京夜景，打个滴滴回家吧。

但是这个方案比较 low，谁都能干，我们来看看高大上一点的方案。



双写迁移方案

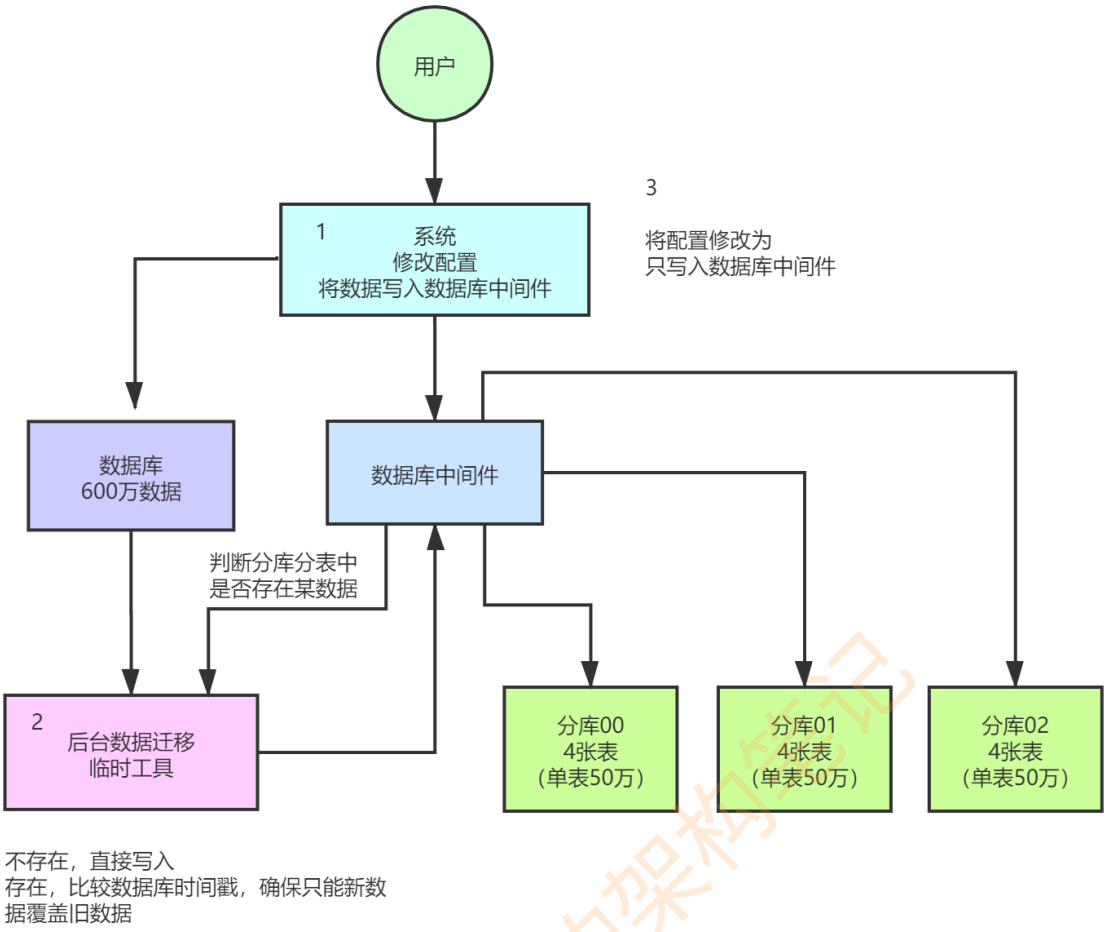
这个是我们常用的一种迁移方案，比较靠谱一些，不用停机，不用看北京凌晨 4 点的风景。

简单来说，就是在线上系统里面，之前所有写库的地方，增删改操作，**除了对老库增删改，都加上对新库的增删改**，这就是所谓的**双写**，同时写俩库，老库和新库。

然后**系统部署**之后，新库数据差太远，用之前说的导数工具，跑起来读老库数据写新库，写的时候要根据 gmt_modified 这类字段判断这条数据最后修改的时间，除非是读出来的数据在新库里没有，或者是比新库的数据新才会写。简单来说，就是不允许用老数据覆盖新数据。

导完一轮之后，有可能数据还是存在不一致，那么就程序自动做一轮校验，比对新老库每个表的每条数据，接着如果有不一样的，就针对那些不一样的，从老库读数据再次写。反复循环，直到两个库每个表的数据都完全一致为止。

接着当数据完全一致了，就 ok 了，基于仅仅使用分库分表的最新代码，重新部署一次，不就仅仅基于分库分表在操作了么，还没有几个小时的停机时间，很稳。所以现在基本玩儿数据迁移之类的，都是这么干的。



【面试题】 - 如何设计可以动态扩容缩容的分库分表方案?

面试官心理分析

对于分库分表来说，主要是面对以下问题：

- 选择一个数据库中间件，调研、学习、测试；
- 设计你的分库分表的一个方案，你要分成多少个库，每个库分成多少个表，比如 3 个库，每个库 4 个表；
- 基于选择好的数据库中间件，以及在测试环境建立好的分库分表的环境，然后测试一下能否正常进行分库分表的读写；
- 完成单库单表到分库分表的**迁移**，双写方案；
- 线上系统开始基于分库分表对外提供服务；
- 扩容了，扩容成 6 个库，每个库需要 12 个表，你怎么来增加更多库和表呢?

这个是你必须面对的一个事儿，就是你已经弄好分库分表方案了，然后一堆库和表都建好了，基于分库分表中间件的代码开发啥的都好了，测试都 ok 了，数据能均匀分布到各个库和各个

表里去，而且接着你还通过双写的方案咔嚓一下上了系统，已经直接基于分库分表方案在搞了。

那么现在问题来了，你现在这些库和表又支撑不住了，要继续扩容咋办？这个可能就是说你的每个库的容量又快满了，或者是你的表数据量又太大了，也可能是你每个库的写并发太高了，你得继续扩容。

这都是玩儿分库分表线上必须经历的事儿。

面试题剖析

停机扩容（不推荐）

这个方案就跟停机迁移一样，步骤几乎一致，唯一的一点就是那个导数的工具，是把现有库表的数据抽出来慢慢倒入到新的库和表里去。但是最好别这么玩儿，有点不太靠谱，因为既然**分库分表**就说明数据量实在是太大了，可能多达几亿条，甚至几十亿，你这么玩儿，可能会出问题。

从单库单表迁移到分库分表的时候，数据量并不是很大，单表最大也就两三千万。那么你写个工具，多弄几台机器并行跑，1小时数据就导完了。这没有问题。

如果 3 个库 + 12 个表，跑了一段时间了，数据量都 1~2 亿了。光是导 2 亿数据，都要导个几个小时，6 点，刚刚导完数据，还要搞后续的修改配置，重启系统，测试验证，10 点才可以搞完。所以不能这么搞。

优化后的方案

一开始上来就是 32 个库，每个库 32 个表，那么总共是 1024 张表。

我可以告诉各位同学，这个分法，第一，基本上国内的互联网肯定都是够用了，第二，无论是并发支撑还是数据量支撑都没问题。

每个库正常承载的写入并发量是 1000，那么 32 个库就可以承载 32 * 1000 = 32000 的写并发，如果每个库承载 1500 的写并发，32 * 1500 = 48000 的写并发，接近 5 万每秒的写入并发，前面再加一个MQ，削峰，每秒写入 MQ 8 万条数据，每秒消费 5 万条数据。

有些除非是国内排名非常靠前的这些公司，他们的最核心的系统的数据库，可能会出现几百台数据库的这么一个规模，128 个库，256 个库，512 个库。

1024 张表，假设每个表放 500 万数据，在 MySQL 里可以放 50 亿条数据。

每秒 5 万的写并发，总共 50 亿条数据，对于国内大部分的互联网公司来说，其实一般来说都够了。

谈分库分表的扩容，**第一次分库分表，就一次性给他分个够**，32 个库，1024 张表，可能对大部分的中小型互联网公司来说，已经可以支撑好几年了。

一个实践是利用 `32 * 32` 来分库分表，即分为 32 个库，每个库里一个表分为 32 张表。一共就是 1024 张表。根据某个 id 先根据 32 取模路由到库，再根据 32 取模路由到库里的表。

orderId	id % 32 (库)	id / 32 % 32 (表)
259	3	8
1189	5	5
352	0	11
4593	17	15

刚开始的时候，这个库可能就是逻辑库，建在一个数据库上的，就是一个 mysql 服务器可能建了 n 个库，比如 32 个库。后面如果要拆分，就是不断在库和 mysql 服务器之间做迁移就可以了。然后系统配合改一下配置即可。

比如说最多可以扩展到 32 个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到 1024 个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是 1024 个表。

这么搞，是不用自己写代码做数据迁移的，都交给 dba 来搞好了，但是 dba 确实是需要做一些库表迁移的工作，但是总比你自己写代码，然后抽数据导数据来的效率高得多吧。

哪怕是要减少库的数量，也很简单，其实说白了就是按倍数缩容就可以了，然后修改一下路由规则。

这里对步骤做一个总结：

1. 设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是 32 库 * 32 表，对于大部分公司来说，可能几年都够了。
2. 路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表
3. 扩容的时候，申请增加更多的数据库服务器，装好 mysql，呈倍数扩容，4 台服务器，扩到 8 台服务器，再到 16 台服务器。
4. 由 dba 负责将原先数据库服务器的库，迁移到新的数据库服务器上去，库迁移是有一些便捷的工具的。
5. 我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址。
6. 重新发布系统，上线，原先的路由规则变都不用变，直接可以基于 n 倍的数据库服务器的资源，继续进行线上系统的提供服务。

【面试题】 - 分库分表之后，id 主键如何处理?

面试官心理分析

其实这是分库分表之后你必然要面对的一个问题，就是 id 咋生成？因为要是分成多个表之后，每个表都是从 1 开始累加，那肯定不对啊，需要一个**全局唯一**的 id 来支持。所以这都是你实际生产环境中必须考虑的问题。

面试题剖析

基于数据库的实现方案

数据库自增 id

这个就是说你的系统里每次得到一个 id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个 id。拿到这个 id 之后再往对应的分库分表里去写入。

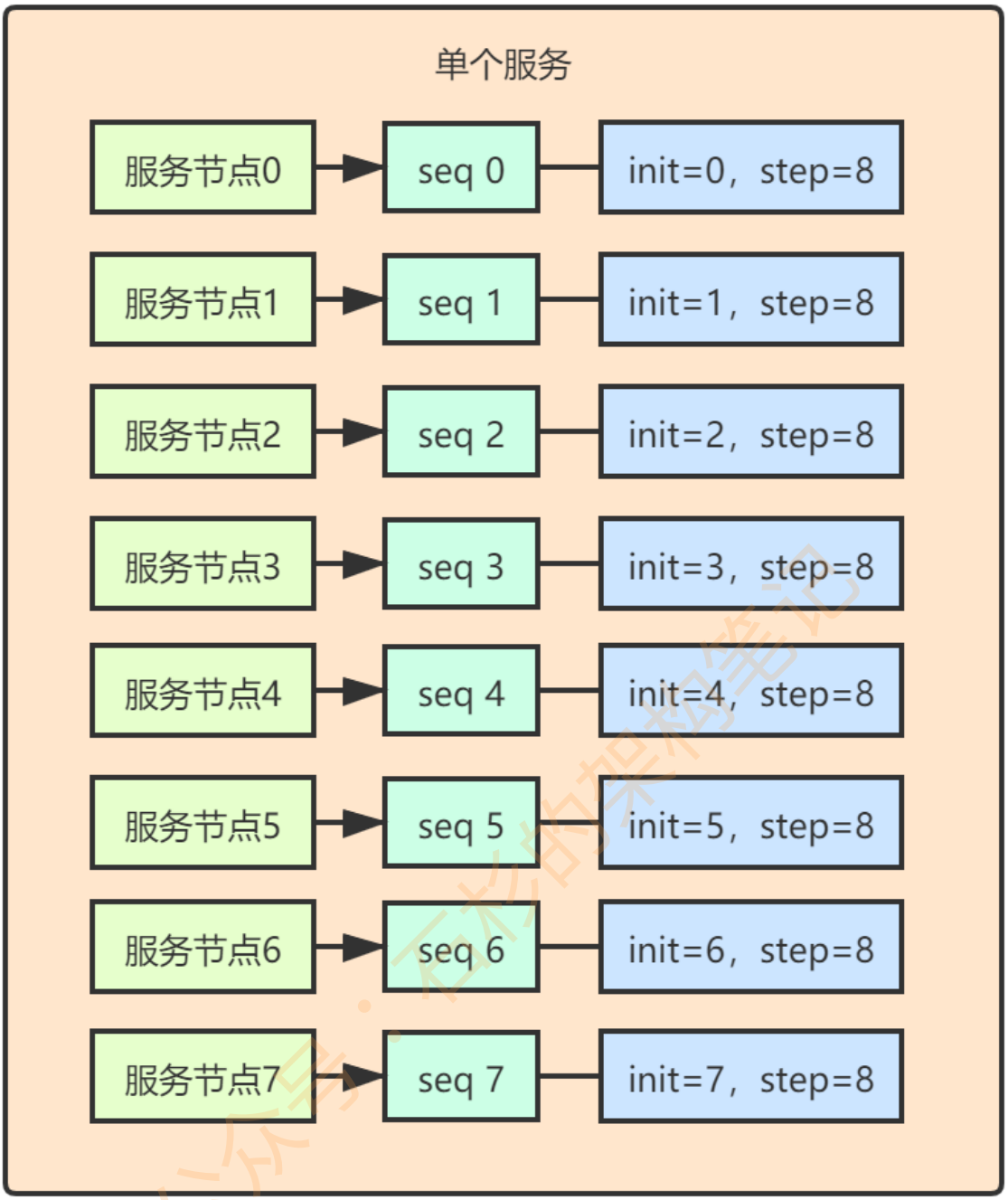
这个方案的好处就是方便简单，谁都会用；**缺点就是单库生成**自增 id，要是高并发的话，就会有瓶颈的；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前 id 最大值，然后自己递增几个 id，一次性返回一批 id，然后再把当前最大 id 值修改成递增几个 id 之后的一个值；但是**无论如何都是基于单个数据库**。

适合的场景：你分库分表就俩原因，要不就是单库并发太高，要不就是单库数据量太大；除非是你**并发不高，但是数据量太大**导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。

设置数据库 sequence 或者表自增字段步长

可以通过设置数据库 sequence 或者表的自增字段步长来进行水平伸缩。

比如说，现在有 8 个服务节点，每个服务节点使用一个 sequence 功能来产生 ID，每个 sequence 的起始 ID 不同，并且依次递增，步长都是 8。



适合的场景：在用户防止产生的 ID 重复时，这种方案实现起来比较简单，也能达到性能目标。但是服务节点固定，步长也固定，将来如果还要增加服务节点，就不好搞了。

UUID

好处就是本地生成，不要基于数据库来了；不好之处就是，UUID 太长了、占用空间大，**作为主键性能太差**了；更重要的是，UUID 不具有有序性，会导致 B+ 树索引在写的时候有过多的随机写操作（连续的 ID 可以产生部分顺序写），还有，由于在写的时候不能产生有顺序的 append 操作，而需要进行 insert 操作，将会读取整个 B+ 树节点到内存，在插入这条记录后会将整个节点写回磁盘，这种操作在记录占用空间比较大的情况下，性能下降明显。

适合的场景：如果你是要随机生成个什么文件名、编号之类的，你可以用 UUID，但是作为主键是不能用 UUID 的。

```java
UUID.randomUUID().toString().replace("-", "") -> sfsdf23423rr234sfdaf
```

获取系统当前时间

这个就是获取当前时间即可，但是问题是，**并发很高的时候**，比如一秒并发几千，**会有重复的情况**，这个是肯定不合适的。基本就不用考虑了。

适合的场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个 id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号。

snowflake 算法

snowflake 算法是 twitter 开源的分布式 id 生成算法，采用 Scala 语言实现，是把一个 64 位的 long 型的 id，1 个 bit 是不用的，用其中的 41 bit 作为毫秒数，用 10 bit 作为工作机器 id，12 bit 作为序列号。

- 1 bit：不用，为啥呢？因为二进制里第一个 bit 为如果是 1，那么都是负数，但是我们生成的 id 都是正数，所以第一个 bit 统一都是 0。
- 41 bit：表示的是时间戳，单位是毫秒。41 bit 可以表示的数字多达 `2^41 - 1`，也就是可以标识 `2^41 - 1` 个毫秒值，换算成年就是表示69年的时间。
- 10 bit：记录工作机器 id，代表的是这个服务最多可以部署在 2^10台机器上哪，也就是1024台机器。但是 10 bit 里 5 个 bit 代表机房 id，5 个 bit 代表机器 id。意思就是最多代表 `2^5` 个机房（32个机房），每个机房里可以代表 `2^5` 个机器（32台机器）。
- 12 bit：这个是用来记录同一个毫秒内产生的不同 id，12 bit 可以代表的最大正整数是 `2^12 - 1 = 4096`，也就是说可以用这个 12 bit 代表的数字来区分**同一个毫秒内**的 4096 个不同的 id。

```
0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000
```

```java
public class IdWorker {

    private long workerId;
```

```
private long datacenterId;
private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence) {
    // sanity check for workerId
    // 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0
    if (workerId > maxWorkerId || workerId < 0) {
        throw new IllegalArgumentException(
                String.format("worker Id can't be greater than %d or l
    }
    if (datacenterId > maxDatacenterId || datacenterId < 0) {
        throw new IllegalArgumentException(
                String.format("datacenter Id can't be greater than %d
    }
    System.out.printf(
            "worker starting. timestamp left shift %d, datacenter id b
            timestampLeftShift, datacenterIdBits, workerIdBits, sequen

    this.workerId = workerId;
    this.datacenterId = datacenterId;
    this.sequence = sequence;
}

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;
private long datacenterIdBits = 5L;

// 这个是二进制运算，就是 5 bit最多只能有31个数字，也就是说机器id最多只能是32以内
private long maxWorkerId = -1L ^ (-1L << workerIdBits);

// 这个是一个意思，就是 5 bit最多只能有31个数字，机房id最多只能是32以内
private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);
private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;
private long datacenterIdShift = sequenceBits + workerIdBits;
private long timestampLeftShift = sequenceBits + workerIdBits + datace
private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId() {
    return workerId;
```

```
}

public long getDatacenterId() {
    return datacenterId;
}

public long getTimestamp() {
    return System.currentTimeMillis();
}

public synchronized long nextId() {
    // 这儿就是获取当前时间戳，单位是毫秒
    long timestamp = timeGen();

    if (timestamp < lastTimestamp) {
        System.err.printf("clock is moving backwards.  Rejecting reque
        throw new RuntimeException(String.format(
                "Clock moved backwards.  Refusing to generate id for %
    }

    if (lastTimestamp == timestamp) {
        // 这个意思是说一个毫秒内最多只能有4096个数字
        // 无论你传递多少进来，这个位运算保证始终就是在4096这个范围内，避免你自己
        sequence = (sequence + 1) & sequenceMask;
        if (sequence == 0) {
            timestamp = tilNextMillis(lastTimestamp);
        }
    } else {
        sequence = 0;
    }

    // 这儿记录一下最近一次生成id的时间戳，单位是毫秒
    lastTimestamp = timestamp;

    // 这儿就是将时间戳左移，放到 41 bit那儿；
    // 将机房 id左移放到 5 bit那儿；
    // 将机器id左移放到5 bit那儿；将序号放最后12 bit；
    // 最后拼接起来成一个 64 bit的二进制数字，转换成 10 进制就是个 long 型
    return ((timestamp - twepoch) << timestampLeftShift) | (datacenter
            | (workerId << workerIdShift) | sequence;
}

private long tilNextMillis(long lastTimestamp) {
    long timestamp = timeGen();
```

```
            while (timestamp <= lastTimestamp) {
                timestamp = timeGen();
            }
            return timestamp;
        }

        private long timeGen() {
            return System.currentTimeMillis();
        }

        // ---------------测试---------------
        public static void main(String[] args) {
            IdWorker worker = new IdWorker(1, 1, 1);
            for (int i = 0; i < 30; i++) {
                System.out.println(worker.nextId());
            }
        }

    }
```

怎么说呢，大概这个意思吧，就是说 41 bit 是当前毫秒单位的一个时间戳，就这意思；然后 5 bit 是你传递进来的一个**机房** id（但是最大只能是 32 以内），另外 5 bit 是你传递进来的**机器** id（但是最大只能是 32 以内），剩下的那个 12 bit序列号，就是如果跟你上次生成 id 的时间还在一个毫秒内，那么会把顺序给你累加，最多在 4096 个序号以内。

所以你自己利用这个工具类，自己搞一个服务，然后对每个机房的每个机器都初始化这么一个东西，刚开始这个机房的这个机器的序号就是 0。然后每次接收到一个请求，说这个机房的这个机器要生成一个 id，你就找到对应的 Worker 生成。

利用这个 snowflake 算法，你可以开发自己公司的服务，甚至对于机房 id 和机器 id，反正给你预留了 5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

这个 snowflake 算法相对来说还是比较靠谱的，所以你要真是搞分布式 id 生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。

【面试题】 - 你们有没有做 MySQL 读写分离？如何实现 MySQL 的读写分离？MySQL 主从复制原理的是啥？如何解决 MySQL 主从同步的延时问题？

面试官心理分析

高并发这个阶段，肯定是需要做读写分离的，啥意思？因为实际上大部分的互联网公司，一些网站，或者是 app，其实都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

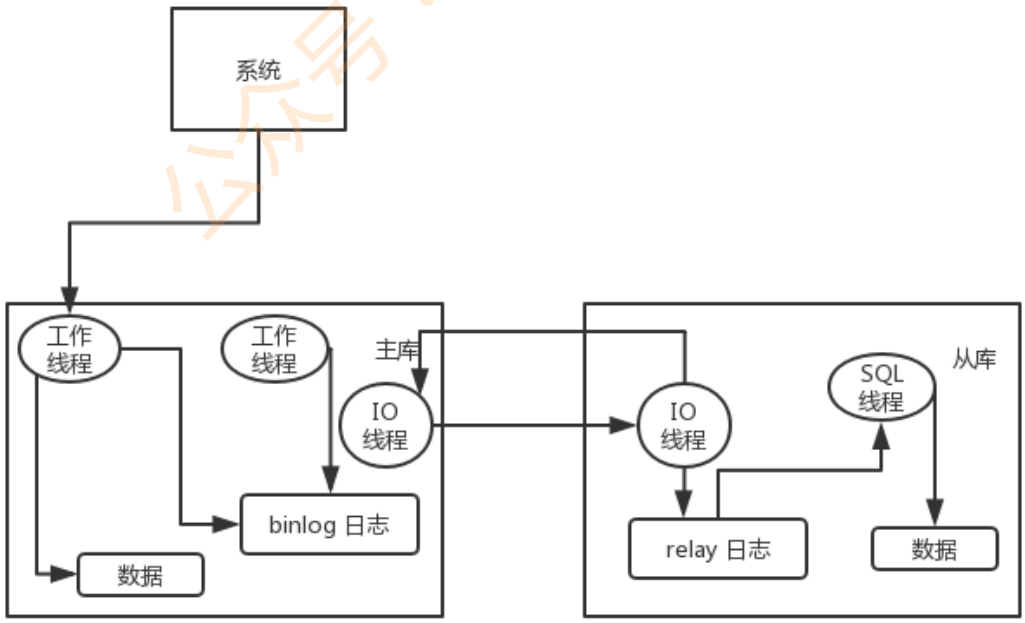
面试题剖析

如何实现 MySQL 的读写分离？

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

MySQL 主从复制原理的是啥？

主库将变更写入 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个 relay 中继日志中。接着从库中有一个 SQL 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。



这里有一个非常重要的一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行的操作，在从库上会串行执行。所以这就是一个非常重要的点了，由于从库从主库拷贝日志以

及串行执行 SQL 的特点，在高并发场景下，从库的数据一定会比主库慢一些，是**有延时**的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。

而且这里还有另外一个问题，就是如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

所以 MySQL 实际上在这一块有两个机制，一个是**半同步复制**，用来解决主库数据丢失问题；一个是**并行复制**，用来解决主从同步延时问题。

这个所谓**半同步复制**，也叫 `semi-sync` 复制，指的就是主库写入 binlog 日志之后，就会将**强制**此时立即将数据同步到从库，从库将日志写入自己本地的 relay log 之后，接着会返回一个 ack 给主库，主库接收到**至少一个从库**的 ack 之后才会认为写操作完成了。

所谓**并行复制**，指的是从库开启多个线程，并行读取 relay log 中不同库的日志，然后**并行重放不同库的日志**，这是库级别的并行。

MySQL 主从同步延时问题（精华）

以前线上确实处理过因为主从同步延时问题而导致的线上的 bug，属于小型的生产事故。

是这个么场景。有个同学是这样写代码逻辑的。先插入一条数据，再把它查出来，然后更新这条数据。在生产环境高峰期，写并发达到了 2000/s，这个时候，主从复制延时大概是在小几十毫秒。线上会发现，每天总有那么一些数据，我们期望更新一些重要的数据状态，但在高峰期时候却没更新。用户跟客服反馈，而客服就会反馈给我们。

我们通过 MySQL 命令：

```sql
show status
```

查看 `Seconds_Behind_Master` ，可以看到从库复制主库的数据落后了几 ms。

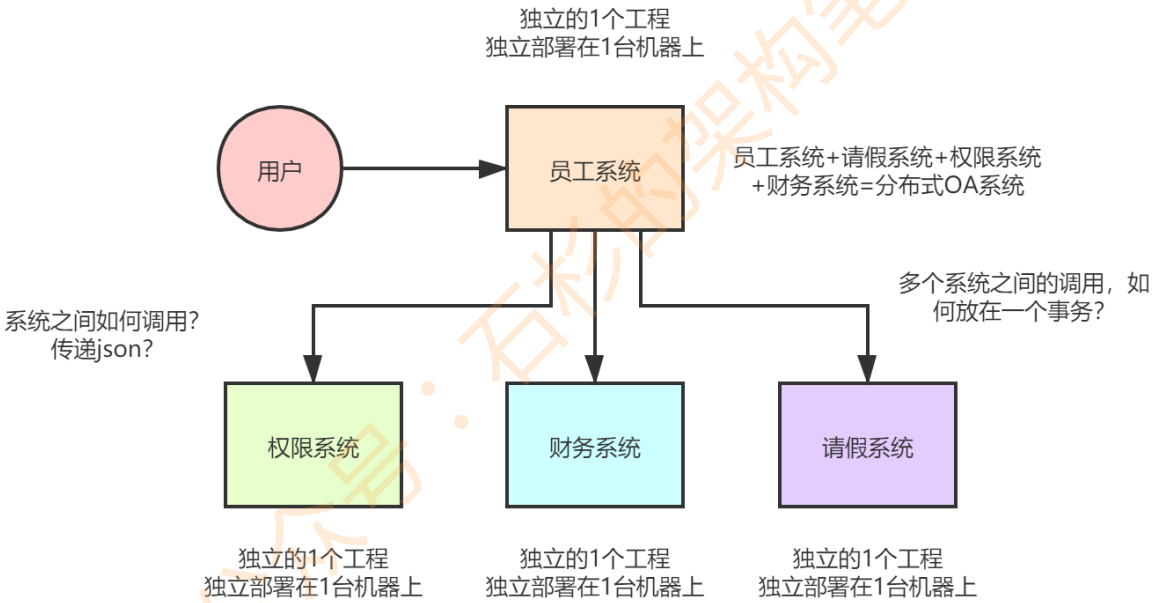
一般来说，如果主从延迟较为严重，有以下解决方案：

- 分库，将一个主库拆分为多个主库，每个主库的写并发就减少了几倍，此时主从延迟可以忽略不计。
- 打开 MySQL 支持的并行复制，多个库并行复制。如果说某个库的写入并发就是特别高，单库写并发达到了 2000/s，并行复制还是没意义。
- 重写代码，写代码的同学，要慎重，插入数据时立马查询可能查不到。
- 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询**设置直连主库**。**不推荐**这种方法，你要是这么搞，读写分离的意义就丧失了。

【面试题】- 分布式系统面试连环炮

有一些同学，之前呢主要是做传统行业，或者外包项目，一直是在那种小的公司，技术一直都搞的比较简单。他们有共同的一个问题，就是都没怎么搞过分布式系统，现在互联网公司，一般都是做分布式的系统，大家都不是做底层的分布式系统、分布式存储系统 hadoop hdfs、分布式计算系统 hadoop mapreduce / spark、分布式流式计算系统 storm。

分布式业务系统，就是把原来用 Java 开发的一个大块系统，给拆分成**多个子系统**，多个子系统之间互相调用，形成一个大系统的整体。假设原来你做了一个 OA 系统，里面包含了权限模块、员工模块、请假模块、财务模块，一个工程，里面包含了一堆模块，模块与模块之间会互相去调用，1 台机器部署。现在如果你把这个系统给拆开，权限系统、员工系统、请假系统、财务系统 4 个系统，4 个工程，分别在 4 台机器上部署。一个请求过来，完成这个请求，这个员工系统，调用权限系统，调用请假系统，调用财务系统，4 个系统分别完成了一部分的事情，最后 4 个系统都干完了以后，才认为是这个请求已经完成了。



> **近几年开始兴起和流行 Spring Cloud，刚流行，还没开始普及，目前普及的是 dubbo，因此这里也主要讲 dubbo。**

面试官可能会问你以下问题。

为什么要进行系统拆分?

- 为什么要进行系统拆分？如何进行系统拆分？拆分后不用dubbo可以吗？dubbo和thrift有什么区别呢?

分布式服务框架

- 说一下的 dubbo 的工作原理？注册中心挂了可以继续通信吗？
- dubbo 支持哪些序列化协议？说一下 hessian 的数据结构？PB 知道吗？为什么 PB 的效率是最高的？
- dubbo 负载均衡策略和高可用策略都有哪些？动态代理策略呢？
- dubbo 的 spi 思想是什么？
- 如何基于 dubbo 进行服务治理、服务降级、失败重试以及超时重试？
- 分布式服务接口的幂等性如何设计（比如不能重复扣款）？
- 分布式服务接口请求的顺序性如何保证？
- 如何自己设计一个类似 dubbo 的 rpc 框架？

分布式锁

- 使用 redis 如何设计分布式锁？使用 zk 来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？

分布式事务

- 分布式事务了解吗？你们如何解决分布式事务问题的？TCC 如果出现网络连不通怎么办？XA 的一致性如何保证？

分布式会话

- 集群部署时的分布式 session 如何实现？

【面试题】- 如何设计一个高并发系统？

面试官心理分析

说实话，如果面试官问你这个题目，那么你必须要使出全身吃奶劲了。为啥？因为你没看到现在很多公司招聘的 JD 里都是说啥，有高并发就经验者优先。

如果你确实有真才实学，在互联网公司里干过高并发系统，那你确实拿 offer 基本如探囊取物，没啥问题。面试官也绝对不会这样来问你，否则他就是蠢。

假设你在某知名电商公司干过高并发系统，用户上亿，一天流量几十亿，高峰期并发量上万，甚至是十万。那么人家一定会仔细盘问你的系统架构，你们系统啥架构？怎么部署的？部署了

多少台机器？缓存咋用的？MQ 咋用的？数据库咋用的？就是深挖你到底是如何扛住高并发的。

因为真正干过高并发的人一定知道，脱离了业务的系统架构都是在纸上谈兵，真正在复杂业务场景而且还高并发的时候，那系统架构一定不是那么简单的，用个 redis，用 mq 就能搞定？当然不是，真实的系统架构搭配上业务之后，会比这种简单的所谓“高并发架构”要复杂很多倍。

如果有面试官问你个问题说，如何设计一个高并发系统？那么不好意思，**一定是因为你实际上没干过高并发系统**。面试官看你简历就没啥出彩的，感觉就不咋地，所以就会问问你，如何设计一个高并发系统？其实说白了本质就是看看你有没有自己研究过，有没有一定的知识积累。

最好的当然是招聘个真正干过高并发的哥儿们咯，但是这种哥儿们人数稀缺，不好招。所以可能次一点的就是招一个自己研究过的哥儿们，总比招一个啥也不会的哥儿们好吧！

所以这个时候你必须得做一把个人秀了，秀出你所有关于高并发的知识！

面试题剖析

其实所谓的高并发，如果你要理解这个问题呢，其实就得从高并发的根源出发，为啥会有高并发？为啥高并发就很牛逼？

我说的浅显一点，很简单，就是因为刚开始系统都是连接数据库的，但是要知道数据库支撑到每秒并发两三千的时候，基本就快完了。所以才有说，很多公司，刚开始干的时候，技术比较 low，结果业务发展太快，有的时候系统扛不住压力就挂了。

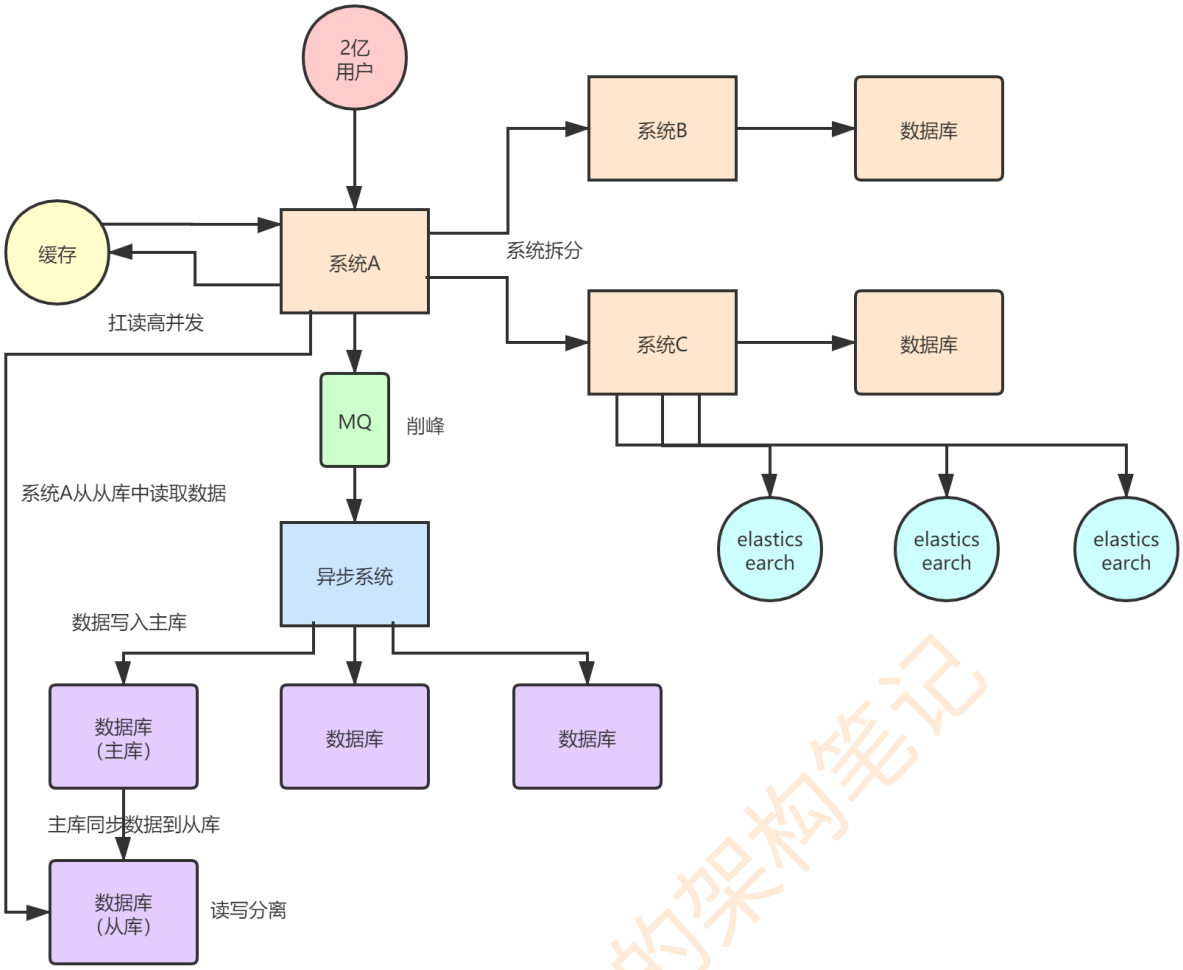
当然会挂了，凭什么不挂？你数据库如果瞬间承载每秒 5000/8000，甚至上万的并发，一定会宕机，因为比如 mysql 就压根儿扛不住这么高的并发量。

所以为啥高并发牛逼？就是因为现在用互联网的人越来越多，很多 app、网站、系统承载的都是高并发请求，可能高峰期每秒并发量几千，很正常的。如果是什么双十一之类的，每秒并发几万几十万都有可能。

那么如此之高的并发量，加上原本就如此之复杂的业务，咋玩儿？真正厉害的，一定是在复杂业务系统里玩儿过高并发架构的人，但是你没有，那么我给你说一下你该怎么回答这个问题：

可以分为以下 6 点：

- 系统拆分
- 缓存
- MQ
- 分库分表
- 读写分离
- ElasticSearch



系统拆分

将一个系统拆分为多个子系统，用 dubbo 来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以扛高并发么。

缓存

缓存，必须得用缓存。大部分的高并发场景，都是**读多写少**，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟人家 redis 轻轻松松单机几万的并发。所以你可以考虑考虑你的项目里，那些承载主要请求的**读场景，怎么用缓存来抗高并发**。

MQ

MQ，必须得用 MQ。可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改，疯了。那高并发绝对搞挂你的系统，你要是用 redis 来承载写那肯定不行，人家是缓存，数据随时就被 LRU 了，数据格式还无比简单，没有事务支持。所以该用 mysql 还得用 mysql 啊。那你咋办？用 MQ 吧，大量的写请求灌入 MQ 里，排队慢慢玩儿，**后边系统消费后慢慢写**，控制在 mysql 承载范围之内。所以你得考虑考虑你的项目里，那些承载

复杂写业务逻辑的场景里，如何用 MQ 来异步写，提升并发性。MQ 单机抗几万并发也是 ok 的，这个之前还特意说过。

分库分表

分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表**拆分为多个表**，每个表的数据量保持少一点，提高 sql 跑的性能。

读写分离

读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，**主库写**入，**从库读**取，搞一个读写分离。**读流量太多**的时候，还可以**加更多的从库**。

ElasticSearch

Elasticsearch，简称 es。es 是分布式的，可以随便扩容，分布式天然就可以支撑高并发，因为动不动就可以扩容加机器来扛更高的并发。那么一些比较简单的查询、统计类的操作，可以考虑用 es 来承载，还有一些全文搜索类的操作，也可以考虑用 es 来承载。

上面的 6 点，基本就是高并发系统肯定要干的一些事儿，大家可以仔细结合之前讲过的知识考虑一下，到时候你可以系统的把这块阐述一下，然后每个部分要注意哪些问题，之前都讲过了，你都可以阐述阐述，表明你对这块是有点积累的。

说句实话，毕竟你真正厉害的一点，不是在于弄明白一些技术，或者大概知道一个高并发系统应该长什么样？其实实际上在真正的复杂的业务系统里，做高并发要远远比上面提到的点要复杂几十倍到上百倍。你需要考虑：哪些需要分库分表，哪些不需要分库分表，单库单表跟分库分表如何 join，哪些数据要放到缓存里去，放哪些数据才可以扛住高并发的请求，你需要完成对一个复杂业务系统的分析之后，然后逐步逐步的加入高并发的系统架构的改造，这个过程是无比复杂的，一旦做过一次，并且做好了，你在这个市场上就会非常的吃香。

其实大部分公司，真正看重的，不是说你掌握高并发相关的一些基本的架构知识，架构中的一些技术，RocketMQ、Kafka、Redis、Elasticsearch，高并发这一块，你了解了，也只能是次一等的人才。对一个有几十万行代码的复杂的分布式系统，一步一步架构、设计以及实践过高并发架构的人，这个经验是难能可贵的。

【面试题】 - 说一下的 dubbo 的工作原理？注册中心挂了可以继续通信吗？说说一次 rpc 请求的流程？

面试官心理分析

MQ、ES、Redis、Dubbo，上来先问你一些**思考性的问题**、**原理**，比如 kafka 高可用架构原理、es 分布式架构原理、redis 线程模型原理、Dubbo 工作原理；之后就是生产环境里可能会碰到的一些问题，因为每种技术引入之后生产环境都可能会碰到一些问题；再来点综合的，就是系统设计，比如让你设计一个 MQ、设计一个搜索引擎、设计一个缓存、设计一个 rpc 框架等等。

那既然开始聊分布式系统了，自然重点先聊聊 dubbo 了，毕竟 dubbo 是目前事实上大部分公司的分布式系统的 rpc 框架标准，基于 dubbo 也可以构建一整套的微服务架构。但是需要自己大量开发。

当然去年开始 spring cloud 非常火，现在大量的公司开始转向 spring cloud 了，spring cloud 人家毕竟是微服务架构的全家桶式的这么一个东西。但是因为很多公司还在用 dubbo，所以 dubbo 肯定会是目前面试的重点，何况人家 dubbo 现在重启开源社区维护了，捐献给了 apache，未来应该也还是有一定市场和地位的。

既然聊 dubbo，那肯定是先从 dubbo 原理开始聊了，你先说说 dubbo 支撑 rpc 分布式调用的架构啥的，然后说说一次 rpc 请求 dubbo 是怎么给你完成的，对吧。

面试题剖析

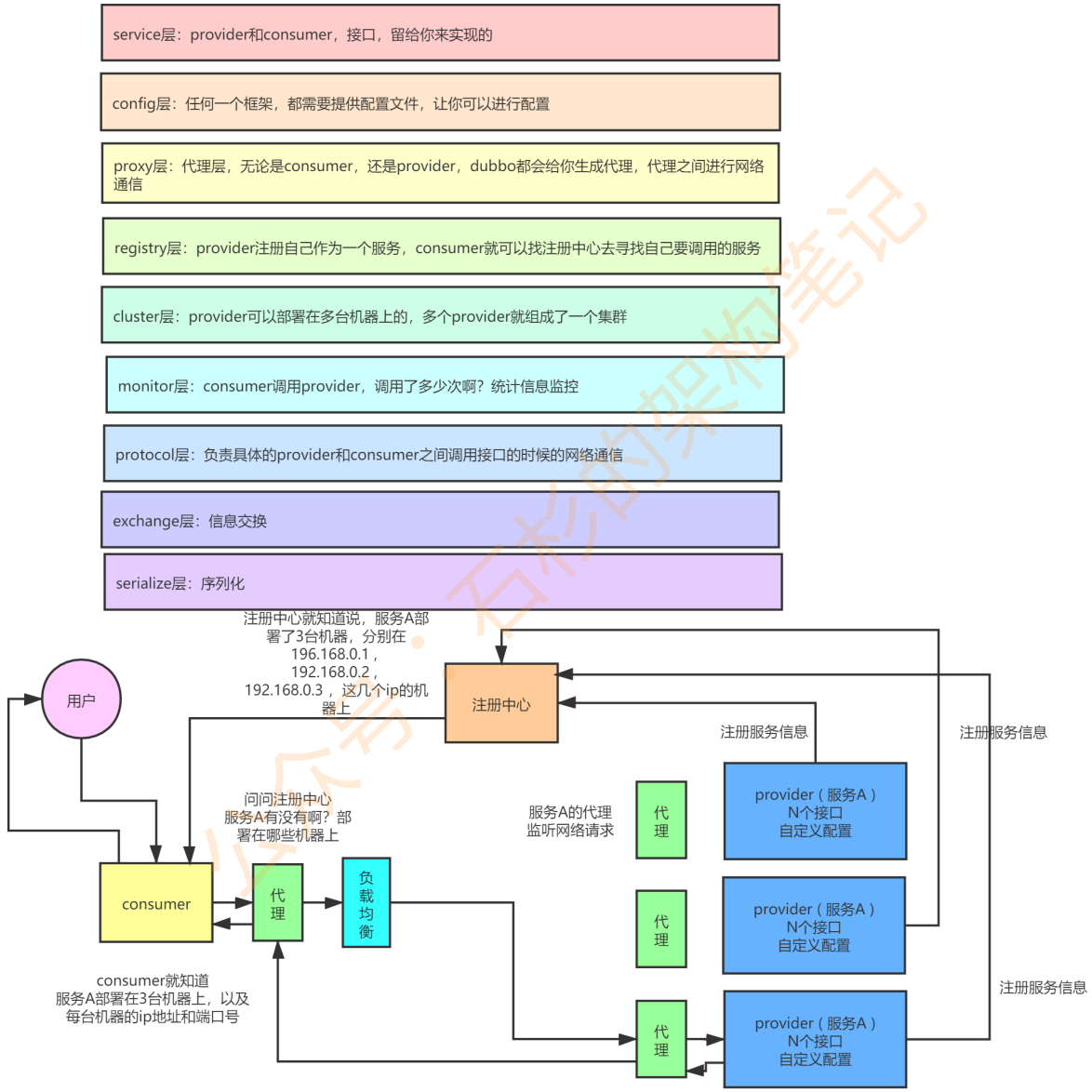
dubbo 工作原理

- 第一层：service 层，接口层，给服务提供者和消费者来实现的
- 第二层：config 层，配置层，主要是对 dubbo 进行各种配置的
- 第三层：proxy 层，服务代理层，无论是 consumer 还是 provider，dubbo 都会给你生成代理，代理之间进行网络通信
- 第四层：registry 层，服务注册层，负责服务的注册与发现
- 第五层：cluster 层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务
- 第六层：monitor 层，监控层，对 rpc 接口的调用次数和调用时间进行监控
- 第七层：protocal 层，远程调用层，封装 rpc 调用
- 第八层：exchange 层，信息交换层，封装请求响应模式，同步转异步
- 第九层：transport 层，网络传输层，抽象 mina 和 netty 为统一接口

- 第十层：serialize 层，数据序列化层

工作流程

- 第一步：provider 向注册中心去注册
- 第二步：consumer 从注册中心订阅服务，注册中心会通知 consumer 注册好的服务
- 第三步：consumer 调用 provider
- 第四步：consumer 和 provider 都异步通知监控中心



注册中心挂了可以继续通信吗？

可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息**拉取到本地缓存**，所以注册中心挂了可以继续通信。

【面试题】 - dubbo 支持哪些通信协议？支持哪些序列化协议？说一下 Hessian 的数据结构？PB 知道吗？为什么 PB 的效率是最高的？

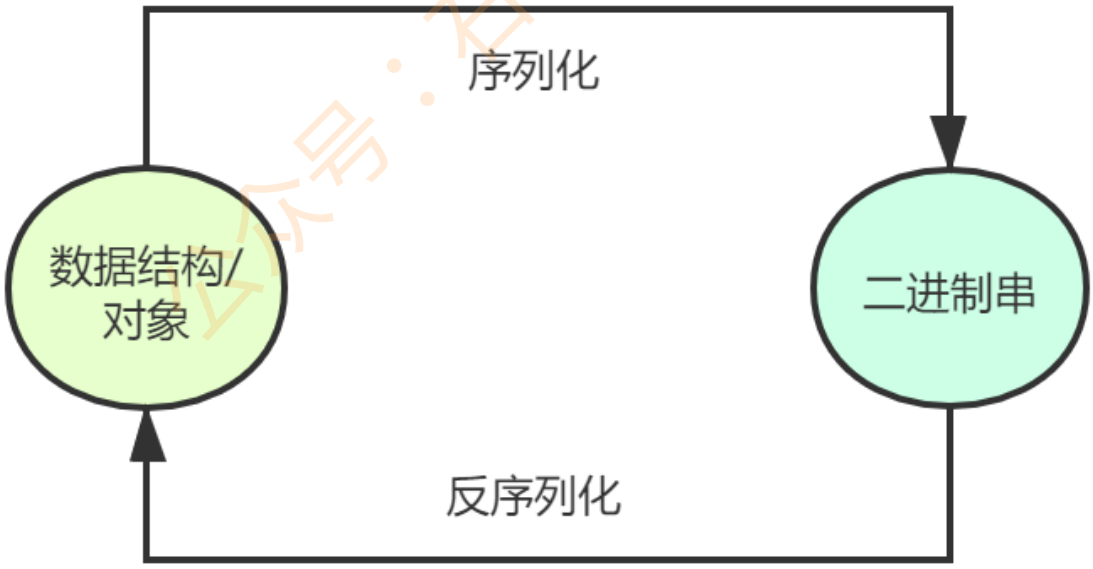
面试官心理分析

上一个问题，说说 dubbo 的基本工作原理，那是你必须知道的，至少要知道 dubbo 分成哪些层，然后平时怎么发起 rpc 请求的，注册、发现、调用，这些是基本的。

接着就可以针对底层进行深入的问问了，比如第一步就可以先问问序列化协议这块，就是平时 RPC 的时候怎么走的？

面试题剖析

序列化，就是把数据结构或者是一些对象，转换为二进制串的过程，而**反序列化**是将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。



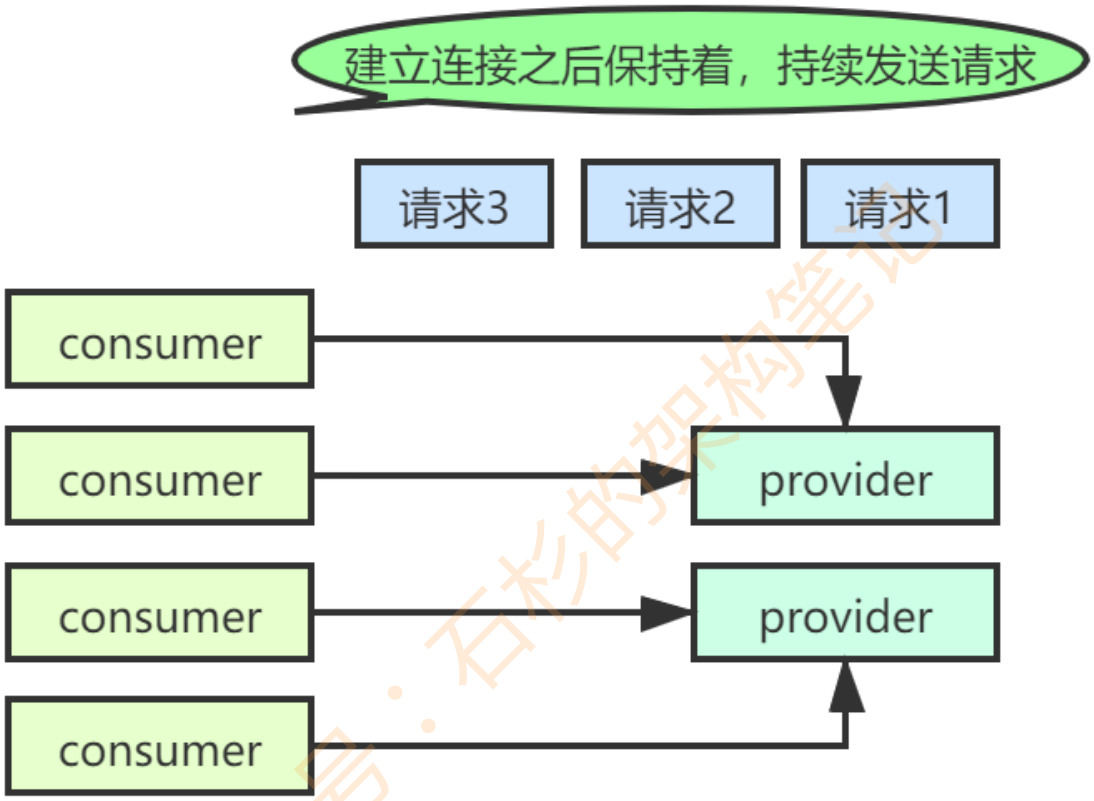
dubbo 支持不同的通信协议

- dubbo 协议

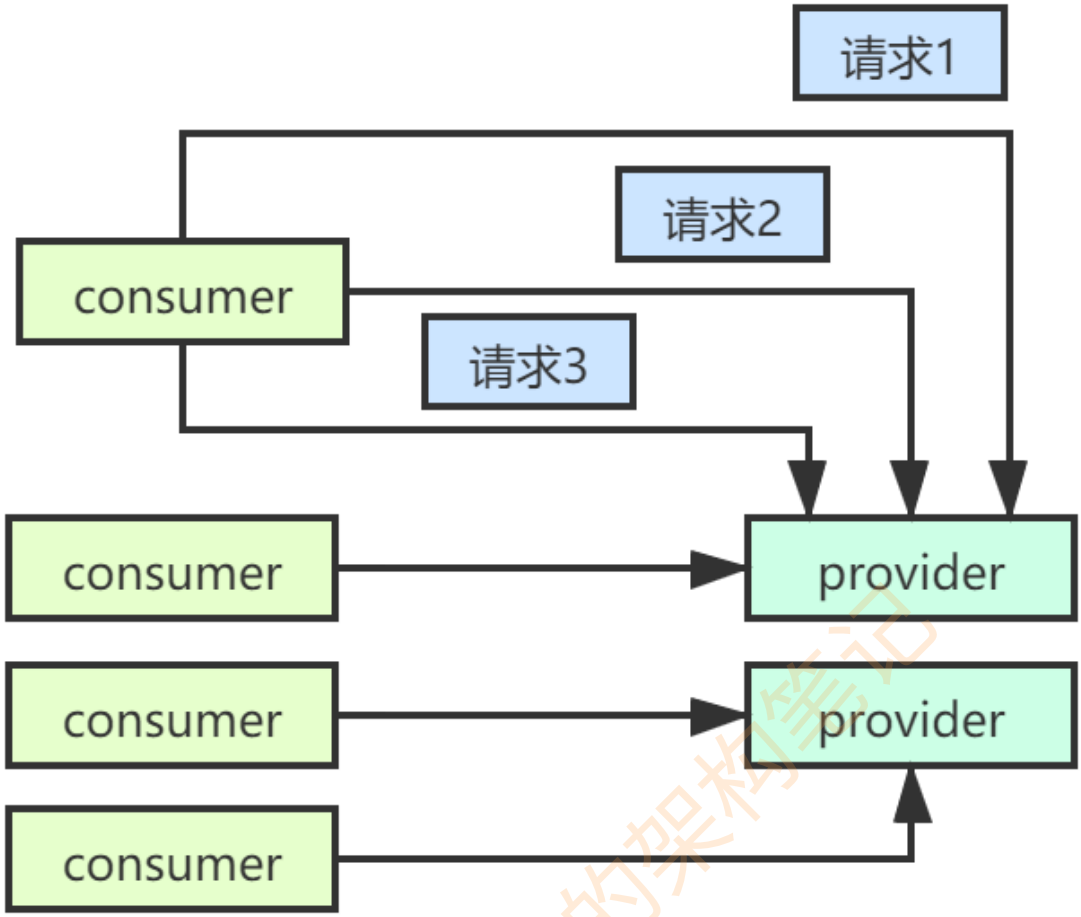
默认就是走 dubbo 协议，单一长连接，进行的是 NIO 异步通信，基于 hessian 作为序列化协议。使用的场景是：传输数据量小（每次请求在 100kb 以内），但是并发量很高。

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就 100 个连接。然后后面直接基于长连接 NIO 异步通信，可以支撑高并发请求。

长连接，通俗点说，就是建立连接过后可以持续发送请求，无须再建立连接。



而短连接，每次要发送请求之前，需要先重新建立一次连接。



- rmi 协议

走 Java 二进制序列化，多个短连接，适合消费者和提供者数量差不多的情况，适用于文件的传输，一般较少用。

- hessian 协议

走 hessian 序列化协议，多个短连接，适用于提供者数量比消费者数量还多的情况，适用于文件的传输，一般较少用。

- http 协议

走 json 序列化。

- webservice

走 SOAP 文本序列化。

dubbo 支持的序列化协议

dubbo 支持 hession、Java 二进制序列化、json、SOAP 文本序列化多种序列化协议。但是 hessian 是其默认的序列化协议。

说一下 Hessian 的数据结构

Hessian 的对象序列化机制有 8 种原始类型：

- 原始二进制数据
- boolean
- 64-bit date（64 位毫秒值的日期）
- 64-bit double
- 32-bit int
- 64-bit long
- null
- UTF-8 编码的 string

另外还包括 3 种递归类型：

- list for lists and arrays
- map for maps and dictionaries
- object for objects

还有一种特殊的类型：

- ref：用来表示对共享对象的引用。

为什么 PB 的效率是最高的?

可能有一些同学比较习惯于 `JSON` or `XML` 数据存储格式，对于 `Protocol Buffer` 还比较陌生。 `Protocol Buffer` 其实是 Google 出品的一种轻量并且高效的结构化数据存储格式，性能比 `JSON` 、 `XML` 要高很多。

其实 PB 之所以性能如此好，主要得益于两个：**第一**，它使用 proto 编译器，自动进行序列化和反序列化，速度非常快，应该比 `XML` 和 `JSON` 快上了 `20~100` 倍；**第二**，它的数据压缩效果好，就是说它序列化后的数据量体积小。因为体积小，传输起来带宽和速度上会有优化。

【面试题】 - dubbo 负载均衡策略和集群容错策略都有哪些？动态代理策略呢?

面试官心理分析

继续深问吧，这些都是用 dubbo 必须知道的一些东西，你得知道基本原理，知道序列化是什么协议，还得知道具体用 dubbo 的时候，如何负载均衡，如何高可用，如何动态代理。

说白了，就是看你对 dubbo 熟悉不熟悉：

- dubbo 工作原理：服务注册、注册中心、消费者、代理通信、负载均衡；
- 网络通信、序列化：dubbo 协议、长连接、NIO、hessian 序列化协议；
- 负载均衡策略、集群容错策略、动态代理策略：dubbo 跑起来的时候一些功能是如何运转的？怎么做负载均衡？怎么做集群容错？怎么生成动态代理？
- dubbo SPI 机制：你了解不了解 dubbo 的 SPI 机制？如何基于 SPI 机制对 dubbo 进行扩展？

面试题剖析

dubbo 负载均衡策略

random loadbalance

默认情况下，dubbo 是 random load balance，即**随机**调用实现负载均衡，可以对 provider 不同实例**设置不同的权重**，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。

roundrobin loadbalance

这个的话默认就是均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

举个栗子。

跟运维同学申请机器，有的时候，我们运气好，正好公司资源比较充足，刚刚有一批热气腾腾、刚刚做好的虚拟机新鲜出炉，配置都比较高：8 核 + 16G 机器，申请到 2 台。过了一段时间，我们感觉 2 台机器有点不太够，我就去找运维同学说，“哥儿们，你能不能再给我一台机器”，但是这时只剩下一台 4 核 + 8G 的机器。我要还是得要。

这个时候，可以给两台 8 核 16G 的机器设置权重 4，给剩余 1 台 4 核 8G 的机器设置权重 2。

leastactive loadbalance

这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给**不活跃的性能差的机器更少的请求**。

consistanthash loadbalance

一致性 Hash 算法，相同参数的请求一定分发到一个 provider 上去，provider 挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。**如果你需要的不是随机负载均衡**，是要一类请求都到一个节点，那就走这个一致性 Hash 策略。

dubbo 集群容错策略

failover cluster 模式

失败自动切换，自动重试其他机器，**默认**就是这个，常见于读操作。（失败重试其它机器）

可以通过以下几种方式配置重试次数：

```xml
<dubbo:service retries="2" />
```

或者

```xml
<dubbo:reference retries="2" />
```

或者

```xml
<dubbo:reference>
    <dubbo:method name="findFoo" retries="2" />
</dubbo:reference>
```

failfast cluster 模式

一次调用失败就立即失败，常见于非幂等性的写操作，比如新增一条记录（调用失败就立即失败）

failsafe cluster 模式

出现异常时忽略掉，常用于不重要的接口调用，比如记录日志。

配置示例如下：

```xml
<dubbo:service cluster="failsafe" />
```

或者

```xml
<dubbo:reference cluster="failsafe" />
```

failback cluster 模式

失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种。

forking cluster 模式

并行调用多个 provider，只要一个成功就立即返回。常用于实时性要求比较高的读操作，但是会浪费更多的服务资源，可通过 `forks="2"` 来设置最大并行数。

broadcacst cluster

逐个调用所有的 provider。任何一个 provider 出错则报错（从 `2.1.0` 版本开始支持）。通常用于通知所有提供者更新缓存或日志等本地资源信息。

dubbo动态代理策略

默认使用 javassist 动态字节码生成，创建代理类。但是可以通过 spi 扩展机制配置自己的动态代理策略。

面试题

dubbo 的 spi 思想是什么?

面试官心理分析

继续深入问呗，前面一些基础性的东西问完了，确定你应该都 ok，了解 dubbo 的一些基本东西，那么问个稍微难一点点的问题，就是 spi，先问问你 spi 是啥？然后问问你 dubbo 的 spi 是怎么实现的？

其实就是看看你对 dubbo 的掌握如何。

面试题剖析

spi 是啥？

spi，简单来说，就是 `service provider interface`，说白了是什么意思呢，比如你有个接口，现在这个接口有 3 个实现类，那么在系统运行的时候对这个接口到底选择哪个实现类呢？这就需要 spi 了，需要**根据指定的配置**或者是**默认的配置**，去**找到对应的实现类**加载进来，然后用这个实现类的实例对象。

举个栗子。

你有一个接口 A。A1/A2/A3 分别是接口A的不同实现。你通过配置 `接口 A = 实现 A2`，那么在系统实际运行的时候，会加载你的配置，用实现 A2 实例化一个对象来提供服务。

spi 机制一般用在哪儿？**插件扩展的场景**，比如说你开发了一个给别人使用的开源框架，如果你想让别人自己写个插件，插到你的开源框架里面，从而扩展某个功能，这个时候 spi 思想就用上了。

Java spi 思想的体现

spi 经典的思想体现，大家平时都在用，比如说 jdbc。

Java 定义了一套 jdbc 的接口，但是 Java 并没有提供 jdbc 的实现类。

但是实际上项目跑的时候，要使用 jdbc 接口的哪些实现类呢？一般来说，我们要**根据自己使用的数据库**，比如 mysql，你就将 `mysql-jdbc-connector.jar` 引入进来；oracle，你就将 `oracle-jdbc-connector.jar` 引入进来。

在系统跑的时候，碰到你使用 jdbc 的接口，他会在底层使用你引入的那个 jar 中提供的实现类。

dubbo 的 spi 思想

dubbo 也用了 spi 思想，不过没有用 jdk 的 spi 机制，是自己实现的一套 spi 机制。

```java
Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).get
```

Protocol 接口，在系统运行的时候，，dubbo 会判断一下应该选用这个 Protocol 接口的哪个实现类来实例化对象来使用。

它会去找一个你配置的 Protocol，将你配置的 Protocol 实现类，加载到 jvm 中来，然后实例化对象，就用你的那个 Protocol 实现类就可以了。

上面那行代码就是 dubbo 里大量使用的，就是对很多组件，都是保留一个接口和多个实现，然后在系统运行的时候动态根据配置去找到对应的实现类。如果你没配置，那就走默认的实现好了，没问题。

```java
@SPI("dubbo")
public interface Protocol {

    int getDefaultPort();

    @Adaptive
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

    void destroy();

}
```


在 dubbo 自己的 jar 里，在 `/META_INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol` 文件中：

```xml
dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol
http=com.alibaba.dubbo.rpc.protocol.http.HttpProtocol
hessian=com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol
```

所以说，这就看到了 dubbo 的 spi 机制默认是怎么玩儿的了，其实就是 Protocol 接口，`@SPI("dubbo")` 说的是，通过 SPI 机制来提供实现类，实现类是通过 dubbo 作为默认 key 去配置文件里找到的，配置文件名称与接口全限定名一样的，通过 dubbo 作为 key 可以找到默认的实现类就是 `com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol` 。

如果想要动态替换掉默认的实现类，需要使用 `@Adaptive` 接口，Protocol 接口中，有两个方法加了 `@Adaptive` 注解，就是说那俩接口会被代理实现。

啥意思呢?

比如这个 Protocol 接口搞了俩 `@Adaptive` 注解标注了方法，在运行的时候会针对 Protocol 生成代理类，这个代理类的那俩方法里面会有代理代码，代理代码会在运行的时候动态根据 url 中的 protocol 来获取那个 key，默认是 dubbo，你也可以自己指定，你如果指定了别的 key，那么就会获取别的实现类的实例了。

如何自己扩展 dubbo 中的组件

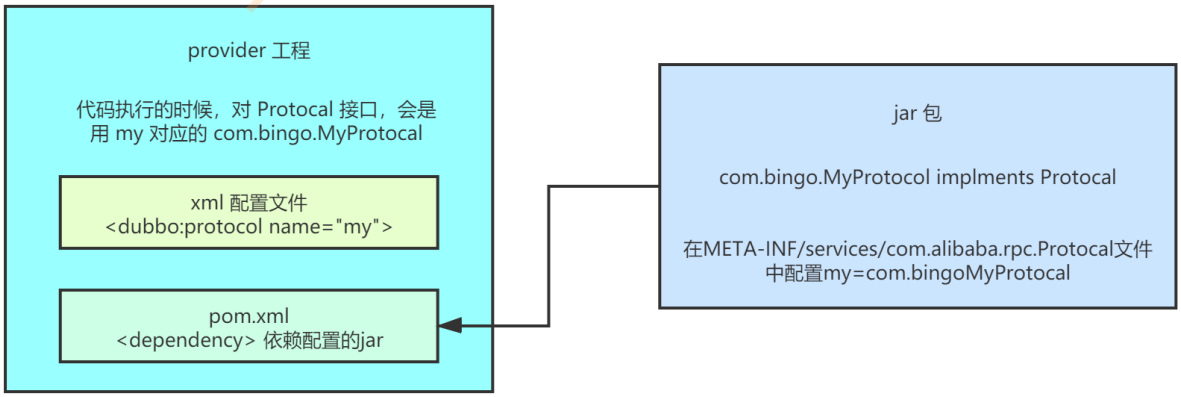
下面来说说怎么来自己扩展 dubbo 中的组件。

自己写个工程，要是那种可以打成 jar 包的，里面的 `src/main/resources` 目录下，搞一个 `META-INF/services` ，里面放个文件叫： `com.alibaba.dubbo.rpc.Protocol` ，文件里搞一个 `my=com.bingo.MyProtocol` 。自己把 jar 弄到 nexus 私服里去。

然后自己搞一个 `dubbo provider` 工程，在这个工程里面依赖你自己搞的那个 jar，然后在 spring 配置文件里给个配置：

```xml
<dubbo:protocol name="my" port="20000" />
```

provider 启动的时候，就会加载到我们 jar 包里的 `my=com.bingo.MyProtocol` 这行配置里，接着会根据你的配置使用你定义好的 MyProtocol 了，这个就是简单说明一下，你通过上述方式，可以替换掉大量的 dubbo 内部的组件，就是扔个你自己的 jar 包，然后配置一下即可。



dubbo 里面提供了大量的类似上面的扩展点，就是说，你如果要扩展一个东西，只要自己写个 jar，让你的 consumer 或者是 provider 工程，依赖你的那个 jar，在你的 jar 里指定目录下配置好接口名称对应的文件，里面通过 `key=实现类` 。

然后对于对应的组件，类似 `<dubbo:protocol>` 用你的那个 key 对应的实现类来实现某个接口，你可以自己去扩展 dubbo 的各种功能，提供你自己的实现。

【面试题】 - 如何基于 dubbo 进行服务治理、服务降级、失败重试以及超时重试?

面试官心理分析

服务治理，这个问题如果问你，其实就是看看你有没有**服务治理**的思想，因为这个是做过复杂微服务的人肯定会遇到的一个问题。

服务降级，这个是涉及到复杂分布式系统中必备的一个话题，因为分布式系统互相来回调用，任何一个系统故障了，你不降级，直接就全盘崩溃？那就太坑爹了吧。

失败重试，分布式系统中网络请求如此频繁，要是因为网络问题不小心失败了一次，是不是要重试?

超时重试，跟上面一样，如果不小心网络慢一点，超时了，如何重试?

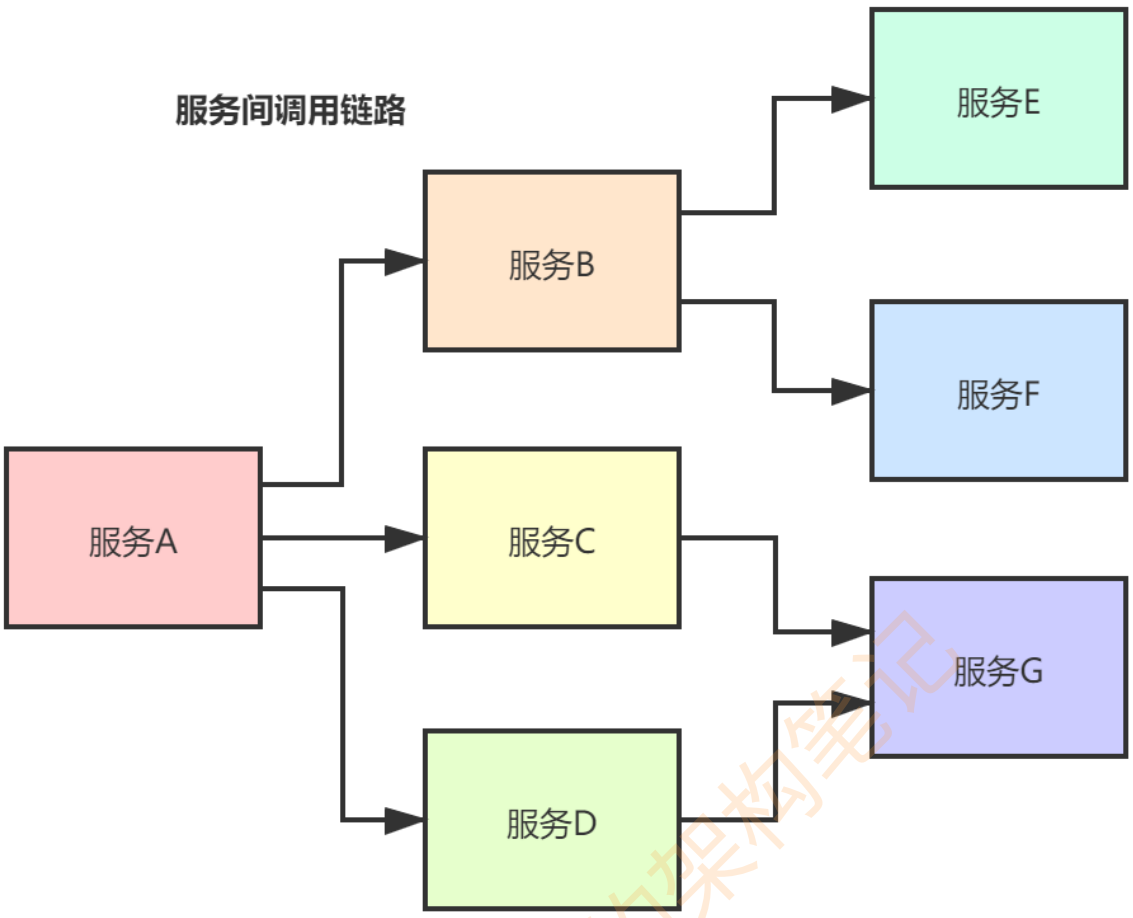
面试题剖析

服务治理

1. 调用链路自动生成

一个大型的分布式系统，或者说是用现在流行的微服务架构来说吧，**分布式系统由大量的服务组成**。那么这些服务之间互相是如何调用的？调用链路是啥？说实话，几乎到后面没人搞的清楚了，因为服务实在太多了，可能几百个甚至几千个服务。

那就需要基于 dubbo 做的分布式系统中，对各个服务之间的调用自动记录下来，然后自动将**各个服务之间的依赖关系和调用链路生成出来**，做成一张图，显示出来，大家才可以看到对吧。



2. 服务访问压力以及时长统计

需要自动统计**各个接口和服务之间的调用次数以及访问延时**，而且要分成两个级别。

- 一个级别是接口粒度，就是每个服务的每个接口每天被调用多少次，TP50/TP90/TP99，三个档次的请求延时分别是多少；
- 第二个级别是从源头入口开始，一个完整的请求链路经过几十个服务之后，完成一次请求，每天全链路走多少次，全链路请求延时的 TP50/TP90/TP99，分别是多少。

这些东西都搞定了之后，后面才可以来看当前系统的压力主要在哪里，如何来扩容和优化啊。

3. 其它

- 服务分层（避免循环依赖）
- 调用链路失败监控和报警
- 服务鉴权
- 每个服务的可用性的监控（接口调用成功率？几个 9？99.99%，99.9%，99%）

服务降级

比如说服务 A 调用服务 B，结果服务 B 挂掉了，服务 A 重试几次调用服务 B，还是不行，那么直接降级，走一个备用的逻辑，给用户返回响应。

举个栗子，我们有接口 `HelloService` 。 `HelloServiceImpl` 有该接口的具体实现。

java

```
public interface HelloService {
   void sayHello();
}

public class HelloServiceImpl implements HelloService {
    public void sayHello() {
        System.out.println("hello world......");
    }
}
```

xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dubbo="htt
    xsi:schemaLocation="http://www.springframework.org/schema/beans

    <dubbo:application name="dubbo-provider" />
    <dubbo:registry address="zookeeper://127.0.0.1:2181" />
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:service interface="com.zhss.service.HelloService" ref="helloSer
    <bean id="helloServiceImpl" class="com.zhss.service.HelloServiceImpl"

</beans>

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans

    <dubbo:application name="dubbo-consumer"  />

    <dubbo:registry address="zookeeper://127.0.0.1:2181" />

    <dubbo:reference id="fooService" interface="com.test.service.FooServic
    </dubbo:reference>
```

```
</beans>
```

我们调用接口失败的时候，可以通过 `mock` 统一返回 null。

mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是 “接口名称+ `Mock` ” 后缀。然后在 Mock 类里实现自己的降级逻辑。

java

```
public class HelloServiceMock implements HelloService {
    public void sayHello() {
        // 降级逻辑
    }
}
```

失败重试和超时重试

所谓失败重试，就是 consumer 调用 provider 要是失败了，比如抛异常了，此时应该是可以重试的，或者调用超时了也可以重试。配置如下：

xml

```
<dubbo:reference id="xxxx" interface="xx" check="true" async="false" retri
```

举个栗子。

某个服务的接口，要耗费 5s，你这边不能干等着，你这边配置了 timeout 之后，我等待 2s，还没返回，我直接就撤了，不能干等你。

可以结合你们公司具体的场景来说说你是怎么设置这些参数的：

- `timeout` ：一般设置为 `200ms` ，我们认为不能超过 `200ms` 还没返回。
- `retries` ：设置 retries，一般是在读请求的时候，比如你要查询个数据，你可以设置个 retries，如果第一次没读到，报错，重试指定的次数，尝试再次读取。

【面试题】 - 分布式服务接口的幂等性如何设计（比如不能重复扣款）？

面试官心理分析

从这个问题开始，面试官就已经进入了**实际的生产问题**的面试了。

一个分布式系统中的某个接口，该如何保证幂等性？这个事儿其实是你做分布式系统的时候必须要考虑的一个生产环境的技术问题。啥意思呢？

你看，假如你有个服务提供一些接口供外部调用，这个服务部署在了 5 台机器上，接着有个接口就是**付款接口**。然后人家用户在前端上操作的时候，不知道为啥，总之就是一个订单**不小心发起了两次支付请求**，然后这俩请求分散在了这个服务部署的不同的机器上，好了，结果一个订单扣款扣两次。

或者是订单系统调用支付系统进行支付，结果不小心因为**网络超时**了，然后订单系统走了前面我们看到的那个重试机制，咔嚓给你重试了一把，好，支付系统收到一个支付请求两次，而且因为负载均衡算法落在了不同的机器上，尴尬了。。。

所以你肯定得知道这事儿，否则你做出来的分布式系统恐怕容易埋坑。

面试题剖析

这个不是技术问题，这个没有通用的一个方法，这个应该**结合业务**来保证幂等性。

所谓**幂等性**，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确的，比如不能多扣款、不能多插入一条数据、不能将统计值多加了 1。这就是幂等性。

其实保证幂等性主要是三点：

- 对于每个请求必须有一个唯一的标识，举个栗子：订单支付请求，肯定得包含订单 id，一个订单 id 最多支付一次，对吧。
- 每次处理完请求之后，必须有一个记录标识这个请求处理过了。常见的方案是在 mysql 中记录个状态啥的，比如支付之前记录一条这个订单的支付流水。
- 每次接收请求需要进行判断，判断之前是否处理过。比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，orderId 已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用再扣款了。

实际运作过程中，你要结合自己的业务来，比如说利用 redis，用 orderId 作为唯一键。只有成功插入这个支付流水，才可以执行实际的支付扣款。

要求是支付一个订单，必须插入一条支付流水，order_id 建一个唯一键 `unique key` 。你在支付一个订单之前，先插入一条支付流水，order_id 就已经进去了。你就可以写一个标识到 redis 里面去， `set order_id payed` ，下一次重复请求过来了，先查 redis 的 order_id 对应的 value，如果是 `payed` 就说明已经支付过了，你就别重复支付了。

【面试题】 - 分布式服务接口请求的顺序性如何保证?

面试官心理分析

其实分布式系统接口的调用顺序，也是个问题，一般来说是不用保证顺序的。但是**有时候**可能确实是需要**严格的顺序**保证。给大家举个例子，你服务 A 调用服务 B，先插入再删除。好，结果俩请求过去了，落在不同机器上，可能插入请求因为某些原因执行慢了一些，导致删除请求先执行了，此时因为没数据所以啥效果也没有；结果这个时候插入请求过来了，好，数据插入进去了，那就尴尬了。

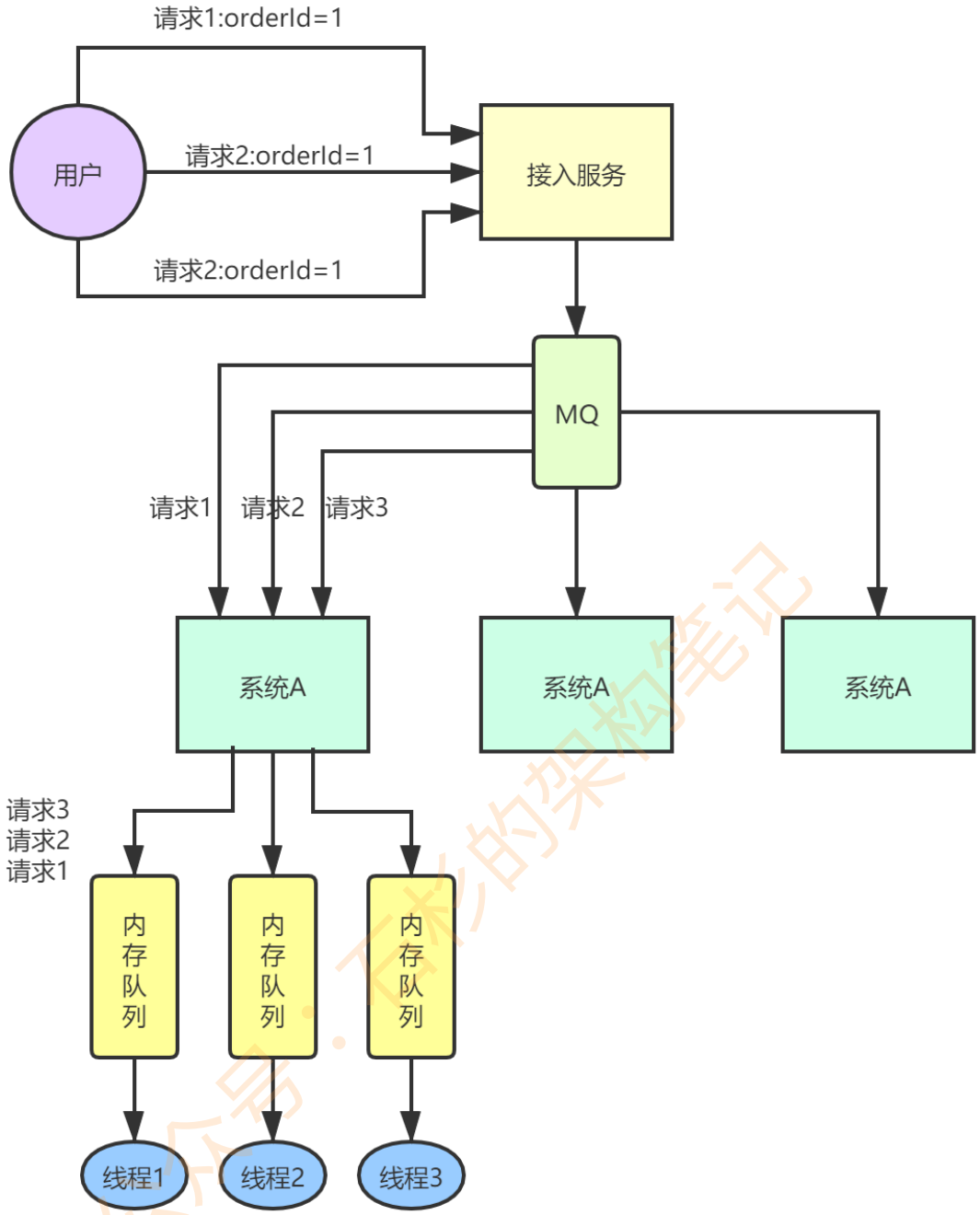
本来应该是 “先插入 -> 再删除”，这条数据应该没了，结果现在 “先删除 -> 再插入”，数据还存在，最后你死都想不明白是怎么回事。

所以这都是分布式系统一些很常见的问题。

面试题剖析

首先，一般来说，个人建议是，你们从业务逻辑上设计的这个系统最好是不需要这种顺序性的保证，因为一旦引入顺序性保障，比如使用**分布式锁**，会**导致系统复杂度上升**，而且会带来**效率低下**，热点数据压力过大等问题。

下面我给个我们用过的方案吧，简单来说，首先你得用 dubbo 的一致性 hash 负载均衡策略，将比如某一个订单 id 对应的请求都给分发到某个机器上去，接着就是在那个机器上，因为可能还是多线程并发执行的，你可能得立即将某个订单 id 对应的请求扔一个**内存队列**里去，强制排队，这样来确保他们的顺序性。



但是这样引发的后续问题就很多，比如说要是某个订单对应的请求特别多，造成某台机器成**热点**怎么办？解决这些问题又要开启后续一连串的复杂技术方案.....曾经这类问题弄的我们头疼不已，所以，还是建议什么呢？

最好是比如说刚才那种，一个订单的插入和删除操作，能不能合并成一个操作，就是一个删除，或者是其它什么，避免这种问题的产生。

【面试题】 - 如何自己设计一个类似 Dubbo 的 RPC 框架？

面试官心理分析

说实话，就这问题，其实就跟问你如何自己设计一个 MQ 一样的道理，就考两个：

- 你有没有对某个 rpc 框架原理有非常深入的理解。
- 你能不能从整体上来思考一下，如何设计一个 rpc 框架，考考你的系统设计能力。

面试题剖析

其实问到你这问题，你起码不能认怂，因为是知识的扫盲，那我不可能给你深入讲解什么 kafka 源码剖析，dubbo 源码剖析，何况我就算讲了，你要真的消化理解和吸收，起码个把月以后了。

所以我给大家一个建议，遇到这类问题，起码从你了解的类似框架的原理入手，自己说说参照 dubbo 的原理，你来设计一下，举个例子，dubbo 不是有那么多分层么？而且每个分层是干啥的，你大概是不是知道？那就按照这个思路大致说一下吧，起码你不能懵逼，要比那些上来就懵，啥也说不出来的人要好一些。

举个栗子，我给大家说个最简单的回答思路：

- 上来你的服务就得去注册中心注册吧，你是不是得有个注册中心，保留各个服务的信息，可以用 zookeeper 来做，对吧。
- 然后你的消费者需要去注册中心拿对应的服务信息吧，对吧，而且每个服务可能会存在于多台机器上。
- 接着你就该发起一次请求了，咋发起？当然是基于动态代理了，你面向接口获取到一个动态代理，这个动态代理就是接口在本地的一个代理，然后这个代理会找到服务对应的机器地址。
- 然后找哪个机器发送请求？那肯定得有个负载均衡算法了，比如最简单的可以随机轮询是不是。
- 接着找到一台机器，就可以跟它发送请求了，第一个问题咋发送？你可以说用 netty 了，nio 方式；第二个问题发送啥格式数据？你可以说用 hessian 序列化协议了，或者是别的，对吧。然后请求过去了。
- 服务器那边一样的，需要针对你自己的服务生成一个动态代理，监听某个网络端口了，然后代理你本地的服务代码。接收到请求的时候，就调用对应的服务代码，对吧。

这就是一个最最基本的 rpc 框架的思路，先不说你有多牛逼的技术功底，哪怕这个最简单的思路你先给出来行不行？

【面试题】 - 为什么要进行系统拆分？如何进行系统拆分？拆分后不用 dubbo 可以吗？

面试官心理分析

从这个问题开始就进行分布式系统环节了，现在出去面试分布式都成标配了，没有哪个公司不问问你分布式的事儿。你要是不会分布式的东西，简直这简历没法看，没人会让你去面试。

其实为啥会这样呢？这就是因为整个大行业技术发展的原因。

早些年，印象中在 2010 年初的时候，整个 IT 行业，很少有人谈分布式，更不用说微服务，虽然很多 BAT 等大型公司，因为系统的复杂性，很早就是分布式架构，大量的服务，只不过微服务大多基于自己搞的一套框架来实现而已。

但是确实，那个年代，大家很重视 ssh2，很多中小型公司几乎大部分都是玩儿 struts2、spring、hibernate，稍晚一些，才进入了 spring mvc、spring、mybatis 的组合。那个时候整个行业的技术水平就是那样，当年 oracle 很火，oracle 管理员很吃香，oracle 性能优化啥的都是 IT 男的大杀招啊。连大数据都没人提，当年 OCP、OCM 等认证培训机构，火的不行。

但是确实随着时代的发展，慢慢的，很多公司开始接受分布式系统架构了，这里面尤为对行业有至关重要影响的，是阿里的 dubbo，**某种程度上而言，阿里在这里推动了行业技术的前进。**

正是因为有阿里的 dubbo，很多中小型公司才可以基于 dubbo，来把系统拆分成很多的服务，每个人负责一个服务，大家的代码都没有冲突，服务可以自治，自己选用什么技术都可以，每次发布如果就改动一个服务那就上线一个服务好了，不用所有人一起联调，每次发布都是几十万行代码，甚至几百万行代码了。

直到今日，很高兴看到分布式系统都成行业面试标配了，任何一个普通的程序员都该掌握这个东西，其实这是行业的进步，也是所有 IT 码农的技术进步。所以既然分布式都成标配了，那么面试官当然会问了，因为很多公司现在都是分布式、微服务的架构，那面试官当然得考察考察你了。

面试题剖析

为什么要将系统进行拆分？

网上查查，答案极度零散和复杂，很琐碎，原因一大坨。但是我这里给大家直观的感受：

要是**不拆分**，一个大系统几十万行代码，20 个人维护一份代码，简直是悲剧啊。代码经常改着改着就冲突了，各种代码冲突和合并要处理，非常耗费时间；经常我改动了我的代码，你调用了我的，导致你的代码也得重新测试，麻烦的要死；然后每次发布都是几十万行代码的系统一起发布，大家得一起提心吊胆准备上线，几十万行代码的上线，可能每次上线都要做很多的检查，很多异常问题的处理，简直是又麻烦又痛苦；而且如果我现在打算把技术升级到最新的 spring 版本，还不行，因为这可能导致你的代码报错，我不敢随意乱改技术。

假设一个系统是 20 万行代码，其中 A 在里面改了 1000 行代码，但是此时发布的时候是这个 20 万行代码的大系统一块儿发布。就意味着 20 万上代码在线上就可能出现各种变化，20 个人，每个人都要紧张地等在电脑面前，上线之后，检查日志，看自己负责的那一块儿有没有什么问题。

A 就检查了自己负责的 1 万行代码对应的功能，确保 ok 就闪人了；结果不巧的是，A 上线的时候不小心修改了线上机器的某个配置，导致另外 B 和 C 负责的 2 万行代码对应的一些功能，出错了。

几十个人负责维护一个几十万行代码的单块应用，每次上线，准备几个礼拜，上线 -> 部署 -> 检查自己负责的功能。

拆分了以后，整个世界清爽了，几十万行代码的系统，拆分成 20 个服务，平均每个服务就 1~2 万行代码，每个服务部署到单独的机器上。20 个工程，20 个 git 代码仓库，20 个开发人员，每个人维护自己的那个服务就可以了，是自己独立的代码，跟别人没关系。再也没有代码冲突了，爽。每次就测试我自己的代码就可以了，爽。每次就发布我自己的一个小服务就可以了，爽。技术上想怎么升级就怎么升级，保持接口不变就可以了，真爽。

所以简单来说，一句话总结，如果是那种代码量多达几十万行的中大型项目，团队里有几十个人，那么如果不拆分系统，**开发效率极其低下**，问题很多。但是拆分系统之后，每个人就负责自己的一小部分就好了，可以随便玩儿随便弄。分布式系统拆分之后，可以大幅度提升复杂系统大型团队的开发效率。

但是同时，也要**提醒**的一点是，系统拆分成分布式系统之后，大量的分布式系统面临的问题也是接踵而来，所以后面的问题都是在**围绕分布式系统带来的复杂技术挑战**在说。

如何进行系统拆分？

这个问题说大可以很大，可以扯到领域驱动模型设计上去，说小了也很小，我不太想给大家太过于学术的说法，因为你也不可能背这个答案，过去了直接说吧。还是说的简单一点，大家自己到时候知道怎么回答就行了。

系统拆分为分布式系统，拆成多个服务，拆成微服务的架构，是需要拆很多轮的。并不是说上来一个架构师一次就给拆好了，而以后都不用拆。

第一轮；团队继续扩大，拆好的某个服务，刚开始是 1 个人维护 1 万行代码，后来业务系统越来越复杂，这个服务是 10 万行代码，5 个人；第二轮，1个服务 -> 5个服务，每个服务 2 万行代码，每人负责一个服务。

如果是多人维护一个服务，最理想的情况下，几十个人，1 个人负责 1 个或 2~3 个服务；某个服务工作量变大了，代码量越来越多，某个同学，负责一个服务，代码量变成了 10 万行了，他自己不堪重负，他现在一个人拆开，5 个服务，1 个人顶着，负责 5 个人，接着招人，2 个人，

给那个同学带着，3 个人负责 5 个服务，其中 2 个人每个人负责 2 个服务，1 个人负责 1 个服务。

个人建议，一个服务的代码不要太多，1 万行左右，两三万撑死了吧。

大部分的系统，是要进行**多轮拆分**的，第一次拆分，可能就是将以前的多个模块该拆分开来了，比如说将电商系统拆分成订单系统、商品系统、采购系统、仓储系统、用户系统，等等吧。

但是后面可能每个系统又变得越来越复杂了，比如说采购系统里面又分成了供应商管理系统、采购单管理系统，订单系统又拆分成了购物车系统、价格系统、订单管理系统。

扯深了实在很深，所以这里先给大家举个例子，你自己感受一下，**核心意思就是根据情况，先拆分一轮，后面如果系统更复杂了，可以继续分拆**。你根据自己负责系统的例子，来考虑一下就好了。

拆分后不用 dubbo 可以吗?

当然可以了，大不了最次，就是各个系统之间，直接基于 spring mvc，就纯 http 接口互相通信呗，还能咋样。但是这个肯定是有问题的，因为 http 接口通信维护起来成本很高，你要考虑**超时重试**、**负载均衡**等等各种乱七八糟的问题，比如说你的订单系统调用商品系统，商品系统部署了 5 台机器，你怎么把请求均匀地甩给那 5 台机器？这不就是负载均衡？你要是都自己搞那是可以的，但是确实很痛苦。

所以 dubbo 说白了，是一种 rpc 框架，就是说本地就是进行接口调用，但是 dubbo 会代理这个调用请求，跟远程机器网络通信，给你处理掉负载均衡、服务实例上下线自动感知、超时重试等等乱七八糟的问题。那你就不用自己做了，用 dubbo 就可以了。

【面试题】- zookeeper 都有哪些使用场景?

面试官心理分析

现在聊的 topic 是分布式系统，面试官跟你聊完了 dubbo 相关的一些问题之后，已经确认你对分布式服务框架/RPC框架基本都有一些认知了。那么他可能开始要跟你聊分布式相关的其它问题了。

分布式锁这个东西，很常用的，你做 Java 系统开发，分布式系统，可能会有一些场景会用到。最常用的分布式锁就是基于 zookeeper 来实现的。

其实说实话，问这个问题，一般就是看看你是否了解 zookeeper，因为 zookeeper 是分布式系统中很常见的一个基础系统。而且问的话常问的就是说 zookeeper 的使用场景是什么？看你知道

不知道一些基本的使用场景。但是其实 zookeeper 挖深了自然是可以问的很深很深的。

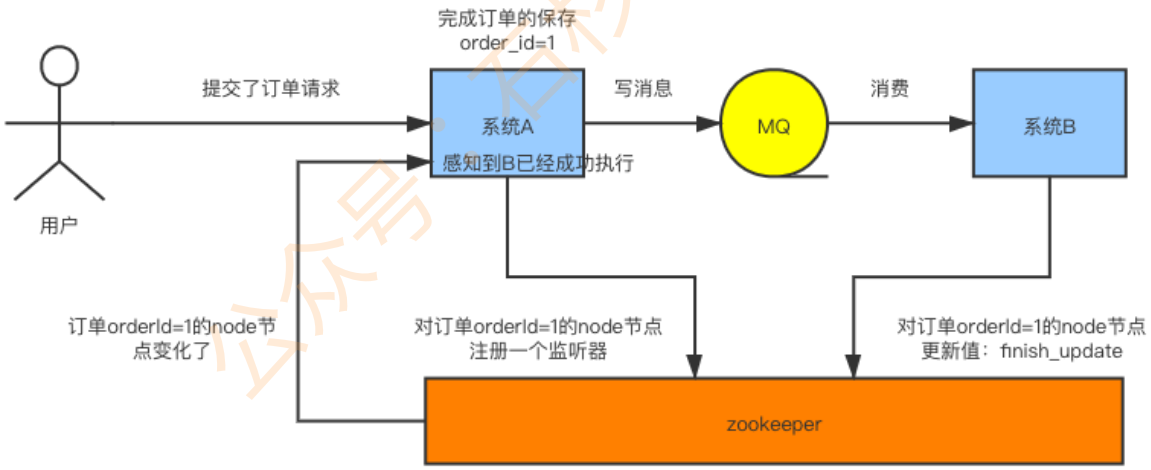
面试题剖析

大致来说，zookeeper 的使用场景如下，我就举几个简单的，大家能说几个就好了：

- 分布式协调
- 分布式锁
- 元数据/配置信息管理
- HA高可用性

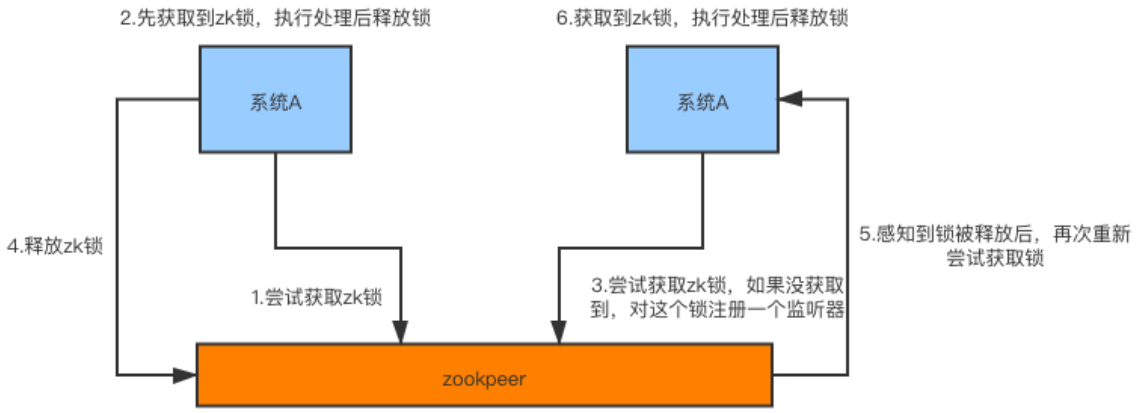
分布式协调

这个其实是 zookeeper 很经典的一个用法，简单来说，就好比，你 A 系统发送个请求到 mq，然后 B 系统消息消费之后处理了。那 A 系统如何知道 B 系统的处理结果？用 zookeeper 就可以实现分布式系统之间的协调工作。A 系统发送请求之后可以在 zookeeper 上**对某个节点的值注册个监听器**，一旦 B 系统处理完了就修改 zookeeper 那个节点的值，A 系统立马就可以收到通知，完美解决。



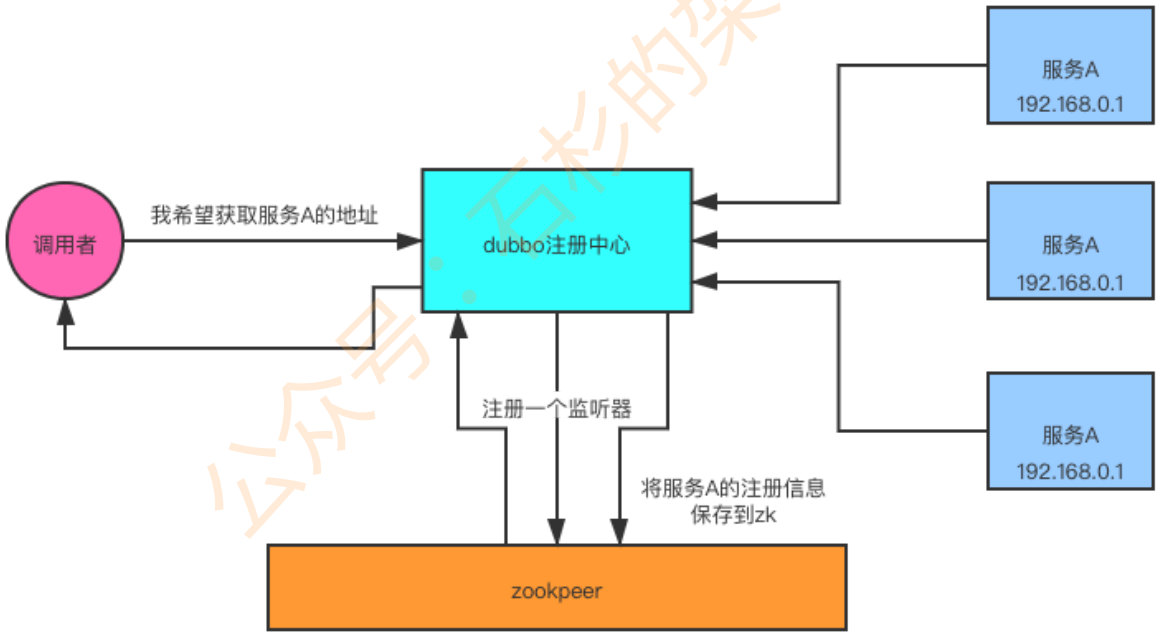
分布式锁

举个栗子。对某一个数据连续发出两个修改操作，两台机器同时收到了请求，但是只能一台机器先执行完另外一个机器再执行。那么此时就可以使用 zookeeper 分布式锁，一个机器接收到了请求之后先获取 zookeeper 上的一把分布式锁，就是可以去创建一个 znode，接着执行操作；然后另外一个机器也**尝试去创建**那个 znode，结果发现自己创建不了，因为被别人创建了，那只能等着，等第一个机器执行完了自己再执行。



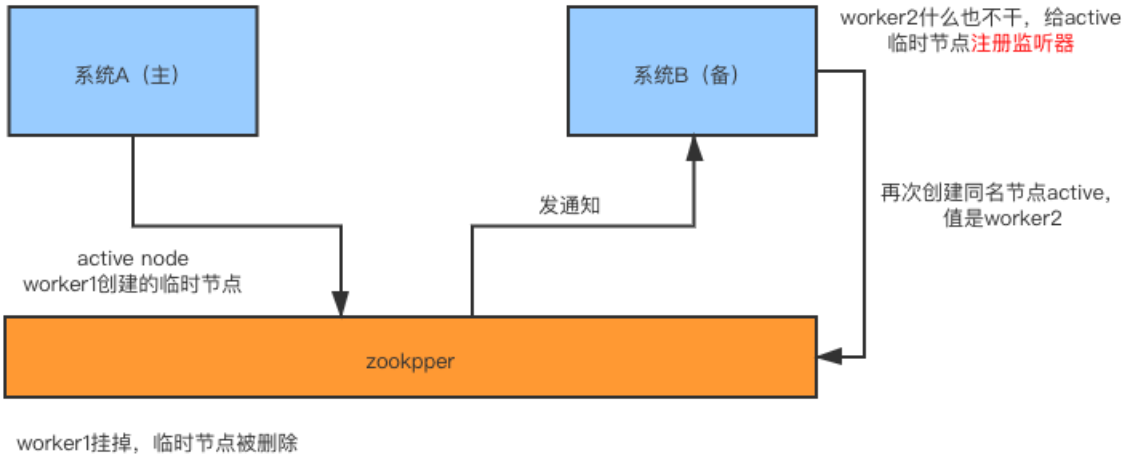
元数据/配置信息管理

zookeeper 可以用作很多系统的配置信息的管理，比如 kafka、storm 等等很多分布式系统都会选用 zookeeper 来做一些元数据、配置信息的管理，包括 dubbo 注册中心不也支持 zookeeper 么？



HA高可用性

这个应该是很常见的，比如 hadoop、hdfs、yarn 等很多大数据系统，都选择基于 zookeeper 来开发 HA 高可用机制，就是一个**重要进程一般会做主备**两个，主进程挂了立马通过 zookeeper 感知到切换到备用进程。



【面试题】 - 一般实现分布式锁都有哪些方式？使用 redis 如何设计分布式锁？使用 zk 来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？

面试官心理分析

其实一般问问题，都是这么问的，先问问你 zk，然后其实是要过渡到 zk 相关的一些问题里去，比如分布式锁。因为在分布式系统开发中，分布式锁的使用场景还是很常见的。

面试题剖析

redis 分布式锁

官方叫做 `RedLock` 算法，是 redis 官方支持的分布式锁算法。

这个分布式锁有 3 个重要的考量点：

- 互斥（只能有一个客户端获取锁）
- 不能死锁
- 容错（只要大部分 redis 节点创建了这把锁就可以）

redis 最普通的分布式锁

第一个最普通的实现方式，就是在 redis 里使用 `setnx` 命令创建一个 key，这样就算加锁。

```
SET resource_name my_random_value NX PX 30000
```

执行这个命令就 ok。

- `NX` ：表示只有 `key` 不存在的时候才会设置成功。（如果此时 redis 中存在这个 key，那么设置失败，返回 `nil` ）
- `PX 30000` ：意思是 30s 后锁自动释放。别人创建的时候如果发现已经有了就不能加锁了。

释放锁就是删除 key ，但是一般可以用 `lua` 脚本删除，判断 value 一样才删除：

```lua
-- 删除锁的时候，找到 key 对应的 value，跟自己传过去的 value 做比较，如果是一样的才删
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

为啥要用 `random_value` 随机值呢？因为如果某个客户端获取到了锁，但是阻塞了很长时间才执行完，比如说超过了 30s，此时可能已经自动释放锁了，此时可能别的客户端已经获取到了这个锁，要是你这个时候直接删除 key 的话会有问题，所以得用随机值加上面的 `lua` 脚本来释放锁。

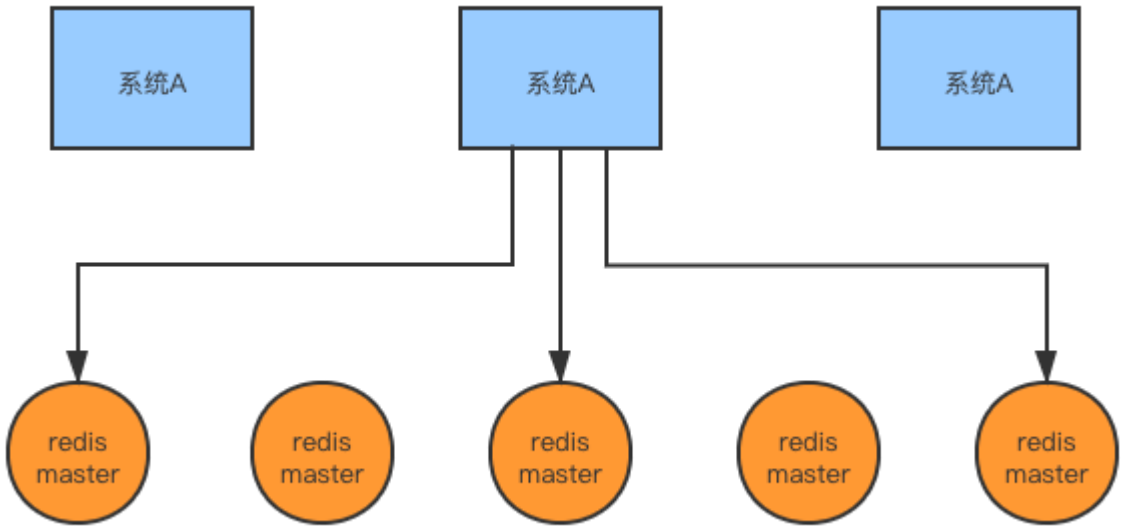
但是这样是肯定不行的。因为如果是普通的 redis 单实例，那就是单点故障。或者是 redis 普通主从，那 redis 主从异步复制，如果主节点挂了（key 就没有了），key 还没同步到从节点，此时从节点切换为主节点，别人就可以 set key，从而拿到锁。

RedLock 算法

这个场景是假设有一个 redis cluster，有 5 个 redis master 实例。然后执行如下步骤获取一把锁：

1. 获取当前时间戳，单位是毫秒；
2. 跟上面类似，轮流尝试在每个 master 节点上创建锁，过期时间较短，一般就几十毫秒；
3. 尝试在**大多数节点**上建立一个锁，比如 5 个节点就要求是 3 个节点 `n / 2 + 1` ；
4. 客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了；
5. 要是锁建立失败了，那么就依次之前建立过的锁删除；

6. 只要别人建立了一把分布式锁，你就得**不断轮询去尝试获取锁**。



Redis 官方给出了以上两种基于 Redis 实现分布式锁的方法，详细说明可以查看：https://redis.io/topics/distlock 。

zk 分布式锁

zk 分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时 znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能**注册个监听器**监听这个锁。释放锁就是删除这个 znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新加锁。

java

```
/**
 * ZooKeeperSession
 *
 * @author bingo
 * @since 2018/11/29
 *
 */
public class ZooKeeperSession {

    private static CountDownLatch connectedSemaphore = new CountDownLatch(

    private ZooKeeper zookeeper;
    private CountDownLatch latch;

    public ZooKeeperSession() {
        try {
```

```
            this.zookeeper = new ZooKeeper("192.168.31.187:2181,192.168.31
            try {
                connectedSemaphore.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("ZooKeeper session established......");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 获取分布式锁
     *
     * @param productId
     */
    public Boolean acquireDistributedLock(Long productId) {
        String path = "/product-lock-" + productId;

        try {
            zookeeper.create(path, "".getBytes(), Ids.OPEN_ACL_UNSAFE, Cre
            return true;
        } catch (Exception e) {
            while (true) {
                try {
                    // 相当于是给node注册一个监听器，去看看这个监听器是否存在
                    Stat stat = zk.exists(path, true);

                    if (stat != null) {
                        this.latch = new CountDownLatch(1);
                        this.latch.await(waitTime, TimeUnit.MILLISECONDS);
                        this.latch = null;
                    }
                    zookeeper.create(path, "".getBytes(), Ids.OPEN_ACL_UNS
                    return true;
                } catch (Exception ee) {
                    continue;
                }
            }

        }
        return true;
```

```
}

/**
 * 释放掉一个分布式锁
 *
 * @param productId
 */
public void releaseDistributedLock(Long productId) {
    String path = "/product-lock-" + productId;
    try {
        zookeeper.delete(path, -1);
        System.out.println("release the lock for product[id=" + produc
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 建立zk session的watcher
 *
 * @author bingo
 * @since 2018/11/29
 *
 */
private class ZooKeeperWatcher implements Watcher {

    public void process(WatchedEvent event) {
        System.out.println("Receive watched event: " + event.getState(

        if (KeeperState.SyncConnected == event.getState()) {
            connectedSemaphore.countDown();
        }

        if (this.latch != null) {
            this.latch.countDown();
        }
    }

}

/**
 * 封装单例的静态内部类
 *
 * @author bingo
```

```
     * @since 2018/11/29
     *
     */
    private static class Singleton {

        private static ZooKeeperSession instance;

        static {
            instance = new ZooKeeperSession();
        }

        public static ZooKeeperSession getInstance() {
            return instance;
        }

    }

    /**
     * 获取单例
     *
     * @return
     */
    public static ZooKeeperSession getInstance() {
        return Singleton.getInstance();
    }

    /**
     * 初始化单例的便捷方法
     */
    public static void init() {
        getInstance();
    }

}
```

也可以采用另一种方式，创建临时顺序节点：

如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁；后面的每个人都会去监听**排在自己前面**的那个人创建的 node 上，一旦某个人释放了锁，排在自己后面的人就会被 zookeeper 给通知，一旦被通知了之后，就 ok 了，自己就获取到了锁，就可以执行代码了。

```java
public class ZooKeeperDistributedLock implements Watcher {

    private ZooKeeper zk;
    private String locksRoot = "/locks";
    private String productId;
    private String waitNode;
    private String lockNode;
    private CountDownLatch latch;
    private CountDownLatch connectedLatch = new CountDownLatch(1);
    private int sessionTimeout = 30000;

    public ZooKeeperDistributedLock(String productId) {
        this.productId = productId;
        try {
            String address = "192.168.31.187:2181,192.168.31.19:2181,192.1
            zk = new ZooKeeper(address, sessionTimeout, this);
            connectedLatch.await();
        } catch (IOException e) {
            throw new LockException(e);
        } catch (KeeperException e) {
            throw new LockException(e);
        } catch (InterruptedException e) {
            throw new LockException(e);
        }
    }

    public void process(WatchedEvent event) {
        if (event.getState() == KeeperState.SyncConnected) {
            connectedLatch.countDown();
            return;
        }

        if (this.latch != null) {
            this.latch.countDown();
        }
    }

    public void acquireDistributedLock() {
        try {
            if (this.tryLock()) {
                return;
            } else {
```

```
            waitForLock(waitNode, sessionTimeout);
        }
    } catch (KeeperException e) {
        throw new LockException(e);
    } catch (InterruptedException e) {
        throw new LockException(e);
    }
}

public boolean tryLock() {
    try {
         // 传入进去的locksRoot + “/” + productId
        // 假设productId代表了一个商品id，比如说1
        // locksRoot = locks
        // /locks/10000000000，/locks/10000000001，/locks/10000000002
        lockNode = zk.create(locksRoot + "/" + productId, new byte[0],

        // 看看刚创建的节点是不是最小的节点
         // locks：10000000000，10000000001，10000000002
        List<String> locks = zk.getChildren(locksRoot, false);
        Collections.sort(locks);

        if(lockNode.equals(locksRoot+"/"+ locks.get(0))){
            //如果是最小的节点,则表示取得锁
            return true;
        }

        //如果不是最小的节点，找到比自己小1的节点
  int previousLockIndex = -1;
        for(int i = 0; i < locks.size(); i++) {
    if(lockNode.equals(locksRoot + “/” + locks.get(i))) {
                previousLockIndex = i - 1;
        break;
    }
   }

   this.waitNode = locks.get(previousLockIndex);
    } catch (KeeperException e) {
        throw new LockException(e);
    } catch (InterruptedException e) {
        throw new LockException(e);
    }
    return false;
}
```

```
    private boolean waitForLock(String waitNode, long waitTime) throws Int
        Stat stat = zk.exists(locksRoot + "/" + waitNode, true);
        if (stat != null) {
            this.latch = new CountDownLatch(1);
            this.latch.await(waitTime, TimeUnit.MILLISECONDS);
            this.latch = null;
        }
        return true;
    }

    public void unlock() {
        try {
            // 删除/locks/10000000000节点
            // 删除/locks/10000000001节点
            System.out.println("unlock " + lockNode);
            zk.delete(lockNode, -1);
            lockNode = null;
            zk.close();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (KeeperException e) {
            e.printStackTrace();
        }
    }

    public class LockException extends RuntimeException {
        private static final long serialVersionUID = 1L;

        public LockException(String e) {
            super(e);
        }

        public LockException(Exception e) {
            super(e);
        }
    }
}
```

redis 分布式锁和 zk 分布式锁的对比

- redis 分布式锁，其实**需要自己不断去尝试获取锁**，比较消耗性能。

- zk 分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。

另外一点就是，如果是 redis 获取锁的那个客户端 出现 bug 挂了，那么只能等待超时时间之后才能释放锁；而 zk 的话，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动释放锁。

redis 分布式锁大家没发现好麻烦吗？遍历上锁，计算时间等等……zk 的分布式锁语义清晰实现简单。

所以先不分析太多的东西，就说这两点，我个人实践认为 zk 的分布式锁比 redis 的分布式锁牢靠、而且模型简单易用。

【面试题】 - 分布式事务了解吗？你们是如何解决分布式事务问题的？

面试官心理分析

只要聊到你做了分布式系统，必问分布式事务，你对分布式事务一无所知的话，确实会很坑，你起码得知道有哪些方案，一般怎么来做，每个方案的优缺点是什么。

现在面试，分布式系统成了标配，而分布式系统带来的**分布式事务**也成了标配了。因为你做系统肯定要用事务吧，如果是分布式系统，肯定要用分布式事务吧。先不说你搞过没有，起码你得明白有哪几种方案，每种方案可能有啥坑？比如 TCC 方案的网络问题、XA 方案的一致性问题。

面试题剖析

分布式事务的实现主要有以下 5 种方案：

- XA 方案
- TCC 方案
- 本地消息表
- 可靠消息最终一致性方案
- 最大努力通知方案

两阶段提交方案/XA方案

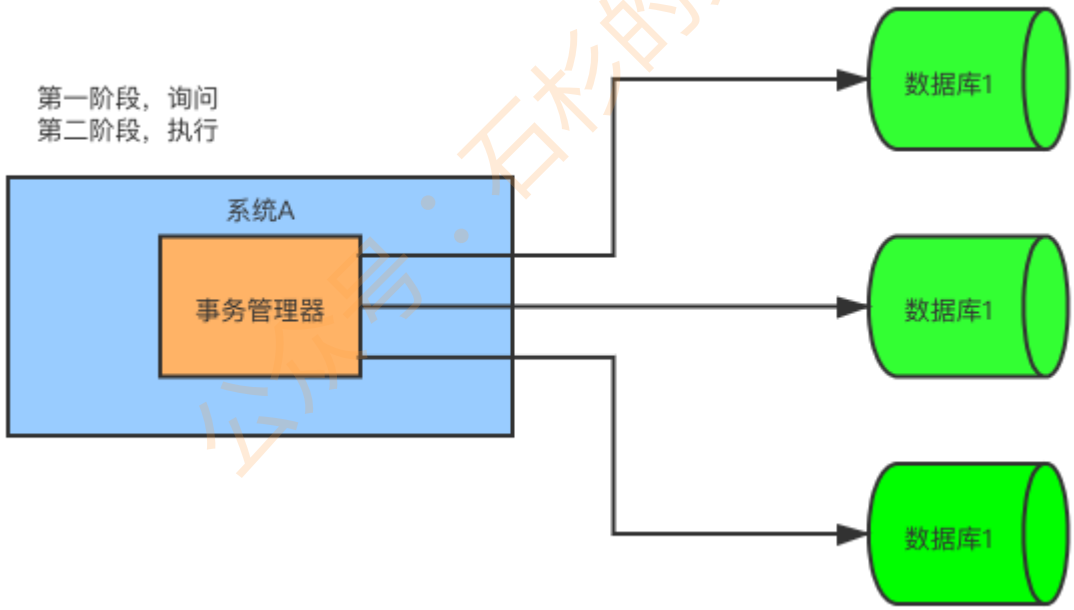
所谓的 XA 方案，即：两阶段提交，有一个**事务管理器**的概念，负责协调多个数据库（资源管理器）的事务，事务管理器先问问各个数据库你准备好了吗？如果每个数据库都回复 ok，那么就正式提交事务，在各个数据库上执行操作；如果任何其中一个数据库回答不 ok，那么就回滚事务。

这种分布式事务方案，比较适合单块应用里，跨多个库的分布式事务，而且因为严重依赖于数据库层面来搞定复杂的事务，效率很低，绝对不适合高并发的场景。如果要玩儿，那么基于 `Spring + JTA` 就可以搞定，自己随便搜个 demo 看看就知道了。

这个方案，我们很少用，一般来说**某个系统内部如果出现跨多个库**的这么一个操作，是**不合规**的。我可以给大家介绍一下， 现在微服务，一个大的系统分成几十个甚至几百个服务。一般来说，我们的规定和规范，是要求**每个服务只能操作自己对应的一个数据库**。

如果你要操作别的服务对应的库，不允许直连别的服务的库，违反微服务架构的规范，你随便交叉胡乱访问，几百个服务的话，全体乱套，这样的一套服务是没法管理的，没法治理的，可能会出现数据被别人改错，自己的库被别人写挂等情况。

如果你要操作别人的服务的库，你必须是通过**调用别的服务的接口**来实现，绝对不允许交叉访问别人的数据库。



TCC 方案

TCC 的全称是： `Try` 、 `Confirm` 、 `Cancel` 。

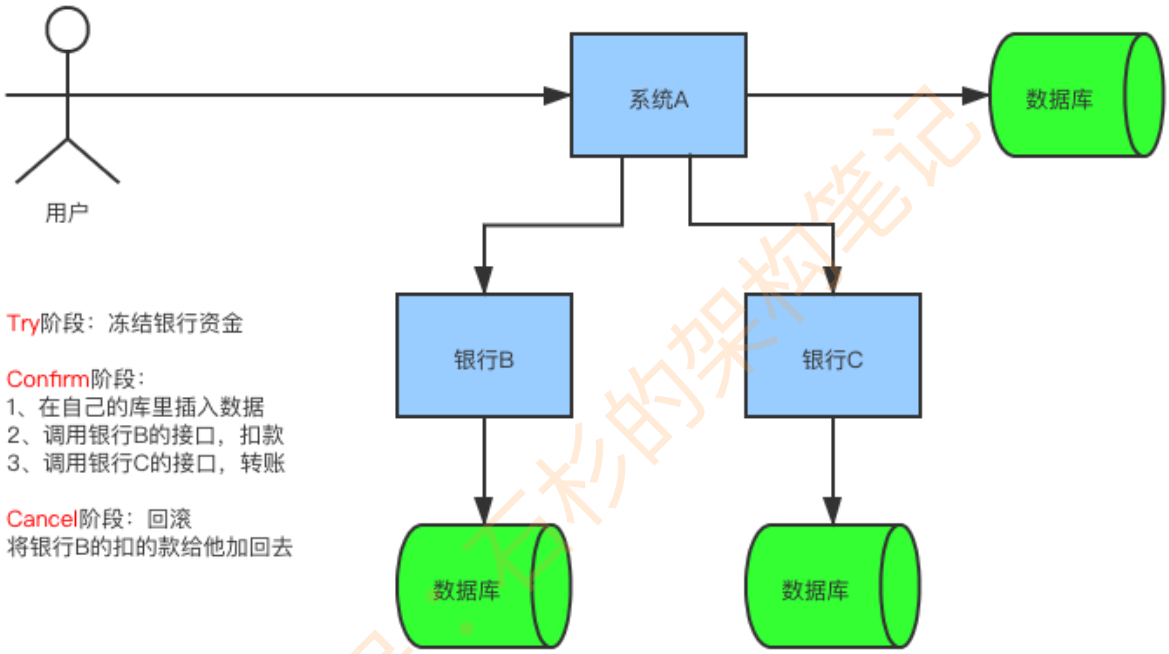
- Try 阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行**锁定或者预留**。
- Confirm 阶段：这个阶段说的是在各个服务中**执行实际的操作**。
- Cancel 阶段：如果任何一个服务的业务方法执行出错，那么这里就需要**进行补偿**，就是执行已经执行成功的业务逻辑的回滚操作。（把那些执行成功的回滚）

这种方案说实话几乎很少人使用，我们用的也比较少，但是也有使用的场景。因为这个**事务回滚**实际上是**严重依赖于你自己写代码来回滚和补偿**了，会造成补偿代码巨大，非常之恶心。

比如说我们，一般来说跟**钱**相关的，跟钱打交道的，**支付**、**交易**相关的场景，我们会用 TCC，严格保证分布式事务要么全部成功，要么全部自动回滚，严格保证资金的正确性，保证在资金上不会出现问题。

而且最好是你的各个业务执行的时间都比较短。

但是说实话，一般尽量别这么搞，自己手写回滚逻辑，或者是补偿逻辑，实在太恶心了，那个业务代码是很难维护的。



本地消息表

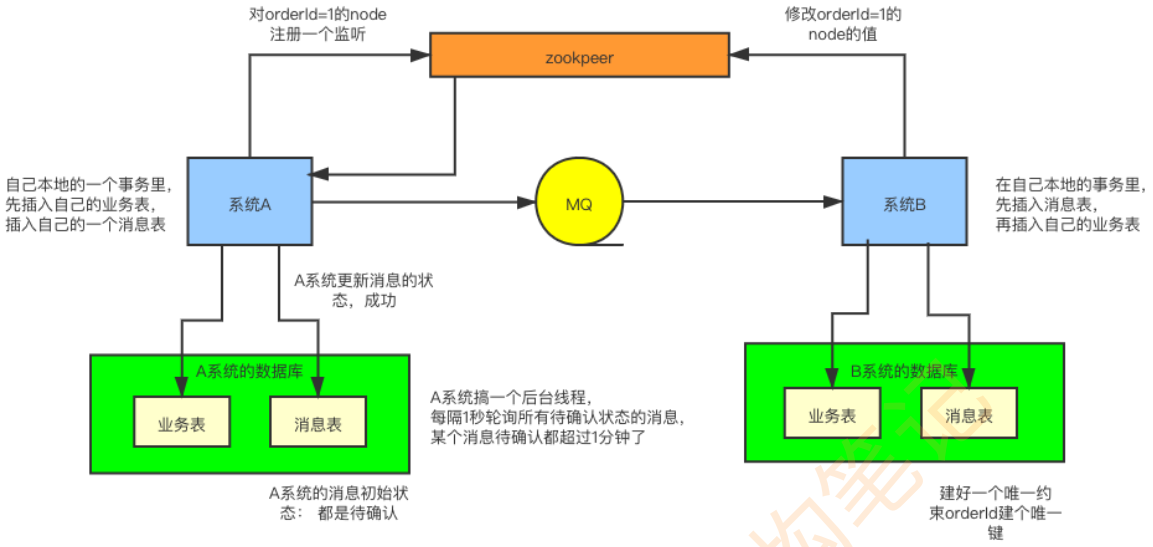
本地消息表其实是国外的 ebay 搞出来的这么一套思想。

这个大概意思是这样的：

1. A 系统在自己本地一个事务里操作同时，插入一条数据到消息表；
2. 接着 A 系统将这个消息发送到 MQ 中去；
3. B 系统接收到消息之后，在一个事务里，往自己本地消息表里插入一条数据，同时执行其他的业务操作，如果这个消息已经被处理过了，那么此时这个事务会回滚，这样**保证不会重复处理消息**；
4. B 系统执行成功之后，就会更新自己本地消息表的状态以及 A 系统消息表的状态；
5. 如果 B 系统处理失败了，那么就不会更新消息表状态，那么此时 A 系统会定时扫描自己的消息表，如果有未处理的消息，会再次发送到 MQ 中去，让 B 再次处理；

6. 这个方案保证了最终一致性，哪怕 B 事务失败了，但是 A 会不断重发消息，直到 B 那边成功为止。

这个方案说实话最大的问题就在于**严重依赖于数据库的消息表来管理事务**啥的，如果是高并发场景咋办呢？咋扩展呢？所以一般确实很少用。

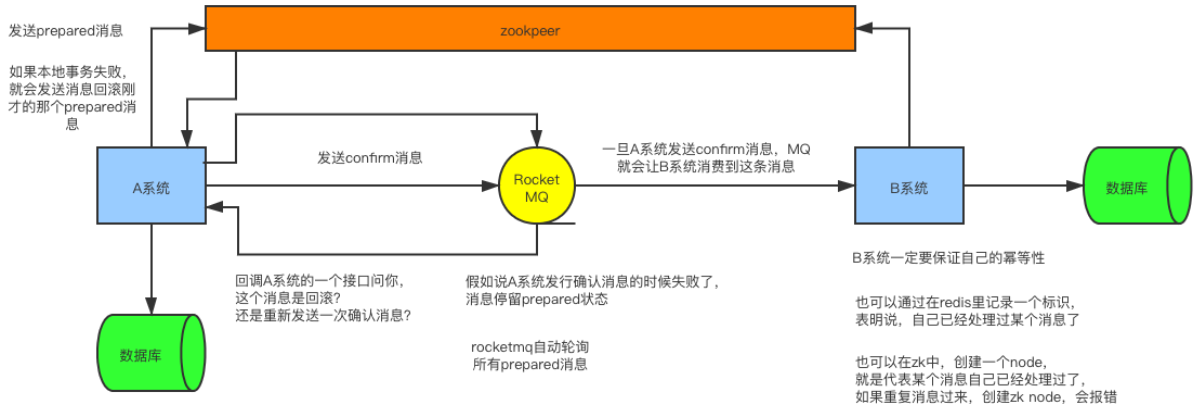


可靠消息最终一致性方案

这个的意思，就是干脆不要用本地的消息表了，直接基于 MQ 来实现事务。比如阿里的 RocketMQ 就支持消息事务。

大概的意思就是：

1. A 系统先发送一个 prepared 消息到 mq，如果这个 prepared 消息发送失败那么就直接取消操作别执行了；
2. 如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉 mq 发送确认消息，如果失败就告诉 mq 回滚消息；
3. 如果发送了确认消息，那么此时 B 系统会接收到确认消息，然后执行本地的事务；
4. mq 会自动**定时轮询**所有 prepared 消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认的消息，是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，而确认消息却发送失败了。
5. 这个方案里，要是系统 B 的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如 B 系统本地回滚后，想办法通知系统 A 也回滚；或者是发送报警由人工来手工回滚和补偿。
6. 这个还是比较合适的，目前国内互联网公司大都是这么玩儿的，要不你举用 RocketMQ 支持的，要不你就自己基于类似 ActiveMQ？RabbitMQ？自己封装一套类似的逻辑出来，总之思路就是这样子的。



最大努力通知方案

这个方案的大致意思就是：

1. 系统 A 本地事务执行完之后，发送个消息到 MQ；
2. 这里会有个专门消费 MQ 的**最大努力通知服务**，这个服务会消费 MQ 然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统 B 的接口；
3. 要是系统 B 执行成功就 ok 了；要是系统 B 执行失败了，那么最大努力通知服务就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃。

你们公司是如何处理分布式事务的?

如果你真的被问到，可以这么说，我们某某特别严格的场景，用的是 TCC 来保证强一致性；然后其他的一些场景基于阿里的 RocketMQ 来实现分布式事务。

你找一个严格资金要求绝对不能错的场景，你可以说你是用的 TCC 方案；如果是一般的分布式事务场景，订单插入之后要调用库存服务更新库存，库存数据没有资金那么的敏感，可以用可靠消息最终一致性方案。

友情提示一下，RocketMQ 3.2.6 之前的版本，是可以按照上面的思路来的，但是之后接口做了一些改变，我这里不再赘述了。

当然如果你愿意，你可以参考可靠消息最终一致性方案来自己实现一套分布式事务，比如基于 RocketMQ 来玩儿。

【面试题】 - 集群部署时的分布式 session 如何实现?

面试官心理分析

面试官问了你一堆 dubbo 是怎么玩儿的，你会玩儿 dubbo 就可以把单块系统弄成分布式系统，然后分布式之后接踵而来的就是一堆问题，最大的问题就是**分布式事务**、**接口幂等性**、**分布式锁**，还有最后一个就是**分布式 session**。

当然了，分布式系统中的问题何止这么一点，非常之多，复杂度很高，这里只是说一下常见的几个问题，也是面试的时候常问的几个。

面试题剖析

session 是啥？浏览器有个 cookie，在一段时间内这个 cookie 都存在，然后每次发请求过来都带上一个特殊的 `jsessionid cookie` ，就根据这个东西，在服务端可以维护一个对应的 session 域，里面可以放点数据。

一般的话只要你没关掉浏览器，cookie 还在，那么对应的那个 session 就在，但是如果 cookie 没了，session 也就没了。常见于什么购物车之类的东西，还有登录状态保存之类的。

这个不多说了，懂 Java 的都该知道这个。

单块系统的时候这么玩儿 session 没问题，但是你要是分布式系统呢，那么多的服务，session 状态在哪儿维护啊？

其实方法很多，但是常见常用的是以下几种：

完全不用 session

使用 JWT Token 储存用户身份，然后再从数据库或者 cache 中获取其他的信息。这样无论请求分配到哪个服务器都无所谓。

tomcat + redis

这个其实还挺方便的，就是使用 session 的代码，跟以前一样，还是基于 tomcat 原生的 session 支持即可，然后就是用一个叫做 `Tomcat RedisSessionManager` 的东西，让所有我们部署的 tomcat 都将 session 数据存储到 redis 即可。

在 tomcat 的配置文件中配置：

```xml
<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHand
```

```
<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionMa
         host="{redis.host}"
         port="{redis.port}"
         database="{redis.dbnum}"
         maxInactiveInterval="60"/>
```

然后指定 redis 的 host 和 port 就 ok 了。

xml

```
<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHand
<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionMa
     sentinelMaster="mymaster"
     sentinels="<sentinel1-ip>:26379,<sentinel2-ip>:26379,<sentinel3-ip>:2
     maxInactiveInterval="60"/>
```

还可以用上面这种方式基于 redis 哨兵支持的 redis 高可用集群来保存 session 数据，都是 ok 的。

spring session + redis

上面所说的第二种方式会与 tomcat 容器重耦合，如果我要将 web 容器迁移成 jetty，难道还要重新把 jetty 都配置一遍?

因为上面那种 tomcat + redis 的方式好用，但是会**严重依赖于web容器**，不好将代码移植到其他 web 容器上去，尤其是你要是换了技术栈咋整？比如换成了 spring cloud 或者是 spring boot 之类的呢?

所以现在比较好的还是基于 Java 一站式解决方案，也就是 spring。人家 spring 基本上承包了大部分我们需要使用的框架，spirng cloud 做微服务，spring boot 做脚手架，所以用 sping session 是一个很好的选择。

在 pom.xml 中配置：

xml

```
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-data-redis</artifactId>
  <version>1.2.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
```

```
    <artifactId>jedis</artifactId>
    <version>2.8.1</version>
</dependency>
```

在 spring 配置文件中配置：

```xml
<bean id="redisHttpSessionConfiguration"
      class="org.springframework.session.data.redis.config.annotation.web.h
    <property name="maxInactiveIntervalInSeconds" value="600"/>
</bean>

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxTotal" value="100" />
    <property name="maxIdle" value="10" />
</bean>

<bean id="jedisConnectionFactory"
      class="org.springframework.data.redis.connection.jedis.JedisConnecti
    <property name="hostName" value="${redis_hostname}"/>
    <property name="port" value="${redis_port}"/>
    <property name="password" value="${redis_pwd}" />
    <property name="timeout" value="3000"/>
    <property name="usePool" value="true"/>
    <property name="poolConfig" ref="jedisPoolConfig"/>
</bean>
```

在 web.xml 中配置：

```xml
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</fi
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

示例代码：

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping("/putIntoSession")
    public String putIntoSession(HttpServletRequest request, String userna
        request.getSession().setAttribute("name",  "leo");
        return "ok";
    }

    @RequestMapping("/getFromSession")
    public String getFromSession(HttpServletRequest request, Model model){
        String name = request.getSession().getAttribute("name");
        return name;
    }
}
```

上面的代码就是 ok 的，给 sping session 配置基于 redis 来存储 session 数据，然后配置了一个 spring session 的过滤器，这样的话，session 相关操作都会交给 spring session 来管了。接着在代码中，就用原生的 session 操作，就是直接基于 spring sesion 从 redis 中获取数据了。

实现分布式的会话有很多种方式，我说的只不过是比较常见的几种方式，tomcat + redis 早期比较常用，但是会重耦合到 tomcat 中；近些年，通过 spring session 来实现。

【面试题】 - Hystrix 介绍

用 Hystrix 构建高可用服务架构参考 **Hystrix Home**。

Hystrix 是什么?

在分布式系统中，每个服务都可能会调用很多其他服务，被调用的那些服务就是**依赖服务**，有的时候某些依赖服务出现故障也是很正常的。

Hystrix 可以让我们在分布式系统中对服务间的调用进行控制，加入一些**调用延迟**或者**依赖故障**的**容错机制**。

Hystrix 通过将依赖服务进行**资源隔离**，进而阻止某个依赖服务出现故障时在整个系统所有的依赖服务调用中进行蔓延；同时Hystrix 还提供故障时的 fallback 降级机制。

总而言之，Hystrix 通过这些方法帮助我们提升分布式系统的可用性和稳定性。

Hystrix 的历史

Hystrix 是高可用性保障的一个框架。Netflix（可以认为是国外的优酷或者爱奇艺之类的视频网站）的 API 团队从 2011 年开始做一些提升系统可用性和稳定性的工作，Hystrix 就是从那时候开始发展出来的。

在 2012 年的时候，Hystrix 就变得比较成熟和稳定了，Netflix 中，除了 API 团队以外，很多其他的团队都开始使用 Hystrix。

时至今日，Netflix 中每天都有数十亿次的服务间调用，通过 Hystrix 框架在进行，而 Hystrix 也帮助 Netflix 网站提升了整体的可用性和稳定性。

2018 年 11 月，Hystrix 在其 Github 主页宣布，不再开放新功能，推荐开发者使用其他仍然活跃的开源项目。维护模式的转变绝不意味着 Hystrix 不再有价值。相反，Hystrix 激发了很多伟大的想法和项目，我们高可用的这一块知识还是会针对 Hystrix 进行讲解。

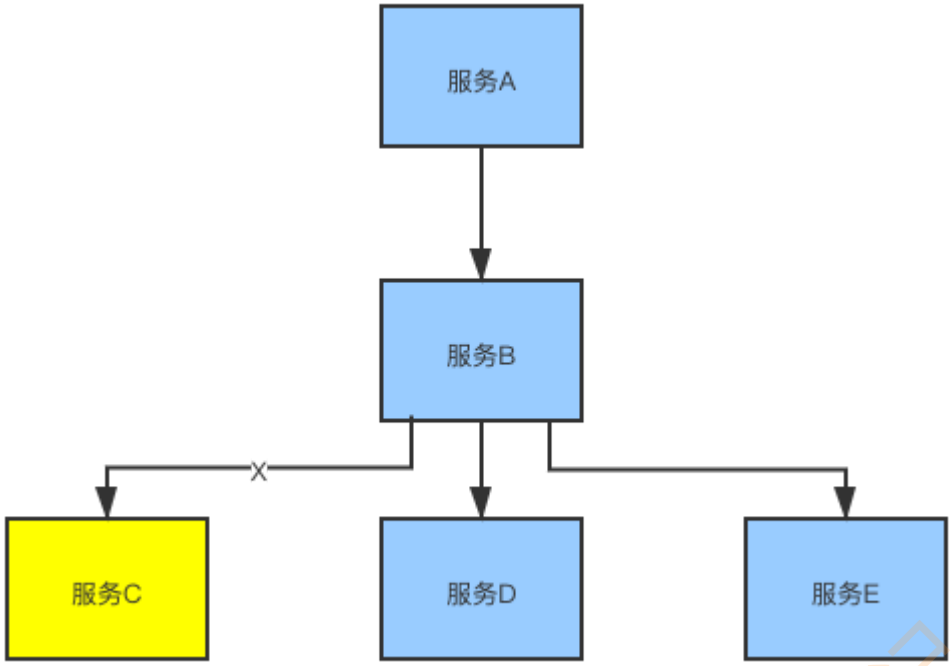
Hystrix 的设计原则

- 对依赖服务调用时出现的调用延迟和调用失败进行**控制和容错保护**。
- 在复杂的分布式系统中，阻止某一个依赖服务的故障在整个系统中蔓延。比如某一个服务故障了，导致其它服务也跟着故障。
- 提供 `fail-fast` （快速失败）和快速恢复的支持。
- 提供 fallback 优雅降级的支持。
- 支持近实时的监控、报警以及运维操作。

举个栗子。

有这样一个分布式系统，服务 A 依赖于服务 B，服务 B 依赖于服务 C/D/E。在这样一个成熟的系统内，比如说最多可能只有 100 个线程资源。正常情况下，40 个线程并发调用服务 C，各 30 个线程并发调用 D/E。

调用服务 C，只需要 20ms，现在因为服务 C 故障了，比如延迟，或者挂了，此时线程会 hang 住 2s 左右。40 个线程全部被卡住，由于请求不断涌入，其它的线程也用来调用服务 C，同样也会被卡住。这样导致服务 B 的线程资源被耗尽，无法接收新的请求，甚至可能因为大量线程不断的运转，导致自己宕机。服务 A 也挂。



Hystrix 可以对其进行资源隔离，比如限制服务 B 只有 40 个线程调用服务 C。当此 40 个线程被 hang 住时，其它 60 个线程依然能正常调用工作。从而确保整个系统不会被拖垮。

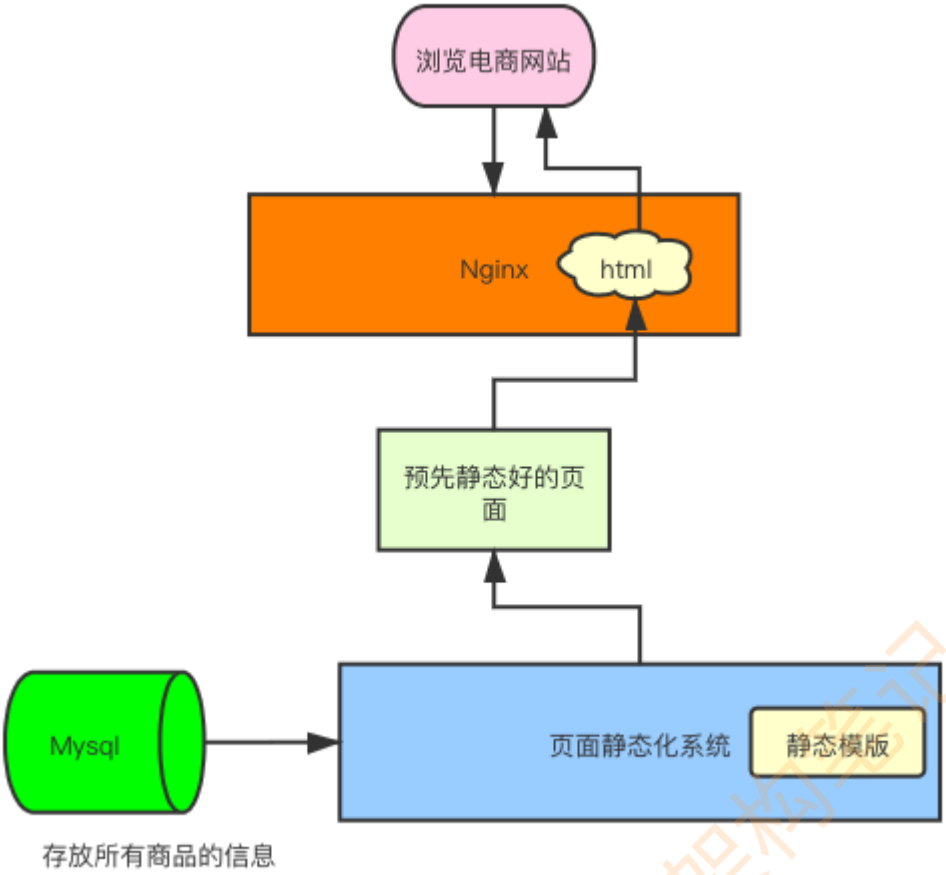
Hystrix 更加细节的设计原则

- 阻止任何一个依赖服务耗尽所有的资源，比如 tomcat 中的所有线程资源。
- 避免请求排队和积压，采用限流和 `fail fast` 来控制故障。
- 提供 fallback 降级机制来应对故障。
- 使用资源隔离技术，比如 `bulkhead`（舱壁隔离技术）、`swimlane`（泳道技术）、`circuit breaker`（断路技术）来限制任何一个依赖服务的故障的影响。
- 通过近实时的统计/监控/报警功能，来提高故障发现的速度。
- 通过近实时的属性和配置**热修改**功能，来提高故障处理和恢复的速度。
- 保护依赖服务调用的所有故障情况，而不仅仅只是网络故障情况。

电商网站的商品详情页系统架构

小型电商网站的商品详情页系统架构

小型电商网站的页面展示采用页面全量静态化的思想。数据库中存放了所有的商品信息，页面静态化系统，将数据填充进静态模板中，形成静态化页面，推入 Nginx 服务器。用户浏览网站页面时，取用一个已经静态化好的 html 页面，直接返回回去，不涉及任何的业务逻辑处理。



下面是页面模板的简单 Demo 。

```html
<html>
    <body>
        商品名称：#{productName}<br>
        商品价格：#{productPrice}<br>
        商品描述：#{productDesc}
    </body>
</html>
```

这样做，**好处**在于，用户每次浏览一个页面，不需要进行任何的跟数据库的交互逻辑，也不需要执行任何的代码，直接返回一个 html 页面就可以了，速度和性能非常高。

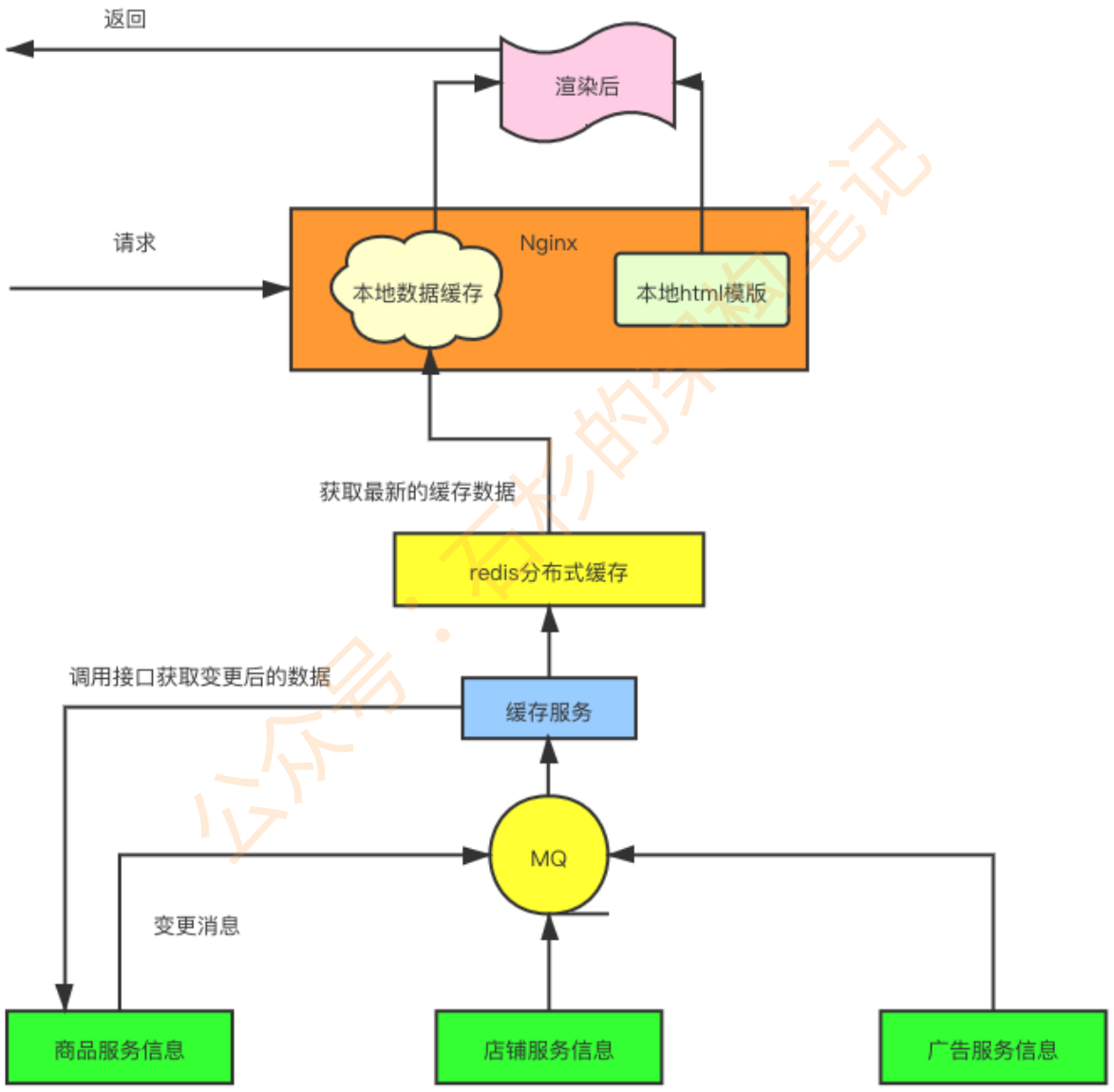
对于小网站，页面很少，很实用，非常简单，Java 中可以使用 velocity、freemarker、thymeleaf 等等，然后做个 cms 页面内容管理系统，模板变更的时候，点击按钮或者系统自动化重新进行全量渲染。

坏处在于，仅仅适用于一些小型的网站，比如页面的规模在几十到几万不等。对于一些大型的电商网站，亿级数量的页面，你说你每次页面模板修改了，都需要将这么多页面全量静态化，靠谱吗？每次渲染花个好几天时间，那你整个网站就废掉了。

大型电商网站的商品详情页系统架构

大型电商网站商品详情页的系统设计中，当商品数据发生变更时，会将变更消息压入 MQ 消息队列中。**缓存服务**从消息队列中消费这条消息时，感知到有数据发生变更，便通过调用数据服务接口，获取变更后的数据，然后将整合好的数据推送至 redis 中。Nginx 本地缓存的数据是有一定的时间期限的，比如说 10 分钟，当数据过期之后，它就会从 redis 获取到最新的缓存数据，并且缓存到自己本地。

用户浏览网页时，动态将 Nginx 本地数据渲染到本地 html 模板并返回给用户。



虽然没有直接返回 html 页面那么快，但是因为数据在本地缓存，所以也很快，其实耗费的也就是动态渲染一个 html 页面的性能。如果 html 模板发生了变更，不需要将所有的页面重新静态化，也不需要发送请求，没有网络请求的开销，直接将数据渲染进最新的 html 页面模板后响应即可。

在这种架构下，我们需要**保证系统的高可用性**。

如果系统访问量很高，Nginx 本地缓存过期失效了，redis 中的缓存也被 LRU 算法给清理掉了，那么会有较高的访问量，从缓存服务调用商品服务。但如果此时商品服务的接口发生故障，调用出现了延时，缓存服务全部的线程都被这个调用商品服务接口给耗尽了，每个线程去调用商品服务接口的时候，都会卡住很长时间，后面大量的请求过来都会卡在那儿，此时缓存服务没有足够的线程去调用其它一些服务的接口，从而导致整个大量的商品详情页无法正常显示。

这其实就是一个商品接口服务故障导致缓存服务资源耗尽的现象。

基于 Hystrix 线程池技术实现资源隔离

上一讲提到，如果从 Nginx 开始，缓存都失效了，Nginx 会直接通过缓存服务调用商品服务获取最新商品数据（我们基于电商项目做个讨论），有可能出现调用延时而把缓存服务资源耗尽的情况。这里，我们就来说说，怎么通过 Hystrix 线程池技术实现资源隔离。

资源隔离，就是说，你如果要把对某一个依赖服务的所有调用请求，全部隔离在同一份资源池内，不会去用其它资源了，这就叫资源隔离。哪怕对这个依赖服务，比如说商品服务，现在同时发起的调用量已经到了 1000，但是线程池内就 10 个线程，最多就只会用这 10 个线程去执行，不会说，对商品服务的请求，因为接口调用延时，将 tomcat 内部所有的线程资源全部耗尽。

Hystrix 进行资源隔离，其实是提供了一个抽象，叫做 command。这也是 Hystrix 最最基本的资源隔离技术。

利用 HystrixCommand 获取单条数据

我们通过将调用商品服务的操作封装在 HystrixCommand 中，限定一个 key，比如下面的 `GetProductInfoCommandGroup` ，在这里我们可以简单认为这是一个线程池，每次调用商品服务，就只会用该线程池中的资源，不会再去用其它线程资源了。

```java
public class GetProductInfoCommand extends HystrixCommand<ProductInfo> {

    private Long productId;

    public GetProductInfoCommand(Long productId) {
        super(HystrixCommandGroupKey.Factory.asKey("GetProductInfoCommandG
        this.productId = productId;
    }

    @Override
    protected ProductInfo run() {
```

```
        String url = "http://localhost:8081/getProductInfo?productId=" + p
        // 调用商品服务接口
        String response = HttpClientUtils.sendGetRequest(url);
        return JSONObject.parseObject(response, ProductInfo.class);
    }
}
```

我们在缓存服务接口中，根据 productId 创建 command 并执行，获取到商品数据。

java

```
@RequestMapping("/getProductInfo")
@ResponseBody
public String getProductInfo(Long productId) {
    HystrixCommand<ProductInfo> getProductInfoCommand = new GetProductInfo

    // 通过command执行，获取最新商品数据
    ProductInfo productInfo = getProductInfoCommand.execute();
    System.out.println(productInfo);
    return "success";
}
```

上面执行的是 execute() 方法，其实是同步的。也可以对 command 调用 queue() 方法，它仅仅是将 command 放入线程池的一个等待队列，就立即返回，拿到一个 Future 对象，后面可以继续做其它一些事情，然后过一段时间对 Future 调用 get() 方法获取数据。这是异步的。

利用 HystrixObservableCommand 批量获取数据

只要是获取商品数据，全部都绑定到同一个线程池里面去，我们通过 HystrixObservableCommand 的一个线程去执行，而在这个线程里面，批量把多个 productId 的 productInfo 拉回来。

java

```
public class GetProductInfosCommand extends HystrixObservableCommand<Produ

    private String[] productIds;

    public GetProductInfosCommand(String[] productIds) {
        // 还是绑定在同一个线程池
        super(HystrixCommandGroupKey.Factory.asKey("GetProductInfoGroup"))
        this.productIds = productIds;
    }
```

```
        @Override
        protected Observable<ProductInfo> construct() {
            return Observable.unsafeCreate((Observable.OnSubscribe<ProductInfo

                for (String productId : productIds) {
                    // 批量获取商品数据
                    String url = "http://localhost:8081/getProductInfo?product
                    String response = HttpClientUtils.sendGetRequest(url);
                    ProductInfo productInfo = JSONObject.parseObject(response,
                    subscriber.onNext(productInfo);
                }
                subscriber.onCompleted();

            }).subscribeOn(Schedulers.io());
        }
    }
```

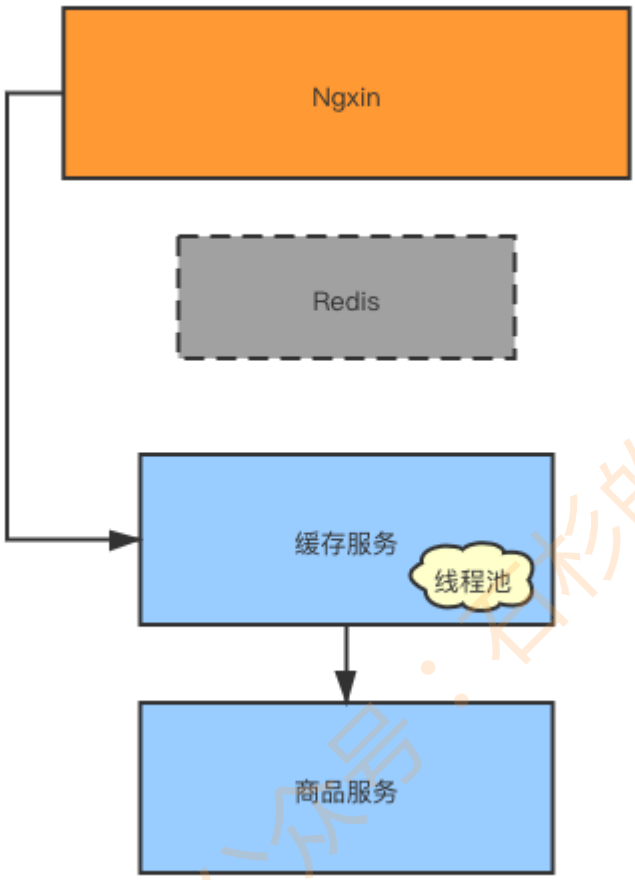
在缓存服务接口中，根据传来的 id 列表，比如是以 , 分隔的 id 串，通过上面的 HystrixObservableCommand，执行 Hystrix 的一些 API 方法，获取到所有商品数据。

java

```
    public String getProductInfos(String productIds) {
        String[] productIdArray = productIds.split(",");
        HystrixObservableCommand<ProductInfo> getProductInfosCommand = new Get
        Observable<ProductInfo> observable = getProductInfosCommand.observe();

        observable.subscribe(new Observer<ProductInfo>() {
            @Override
            public void onCompleted() {
                System.out.println("获取完了所有的商品数据");
            }

            @Override
            public void onError(Throwable e) {
                e.printStackTrace();
            }

            /**
             * 获取完一条数据，就回调一次这个方法
             * @param productInfo
             */
            @Override
```

```
            public void onNext(ProductInfo productInfo) {
                System.out.println(productInfo);
            }
        });
        return "success";
    }
```

我们回过头来，看看 Hystrix 线程池技术是如何实现资源隔离的。



从 Nginx 开始，缓存都失效了，那么 Nginx 通过缓存服务去调用商品服务。缓存服务默认的线程大小是 10 个，最多就只有 10 个线程去调用商品服务的接口。即使商品服务接口故障了，最多就只有 10 个线程会 hang 死在调用商品服务接口的路上，缓存服务的 tomcat 内其它的线程还是可以用来调用其它的服务，干其它的事情。

基于 Hystrix 信号量机制实现资源隔离

Hystrix 里面核心的一项功能，其实就是所谓的**资源隔离**，要解决的最最核心的问题，就是将多个依赖服务的调用分别隔离到各自的资源池内。避免说对某一个依赖服务的调用，因为依赖服务的接口调用的延迟或者失败，导致服务所有的线程资源全部耗费在这个服务的接口调用上。

一旦说某个服务的线程资源全部耗尽的话，就可能导致服务崩溃，甚至说这种故障会不断蔓延。

Hystrix 实现资源隔离，主要有两种技术：

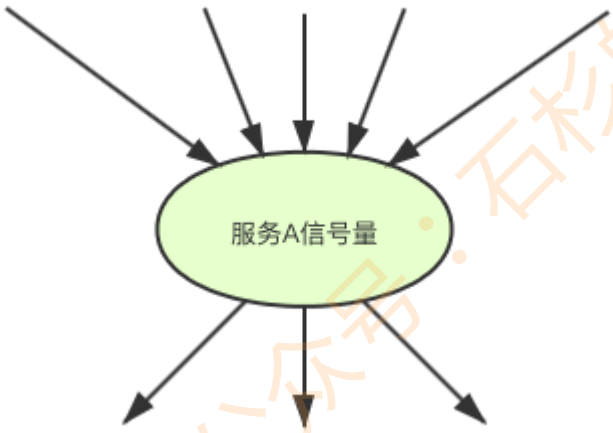
- 线程池
- 信号量

默认情况下，Hystrix 使用线程池模式。

前面已经说过线程池技术了，这一小节就来说说信号量机制实现资源隔离，以及这两种技术的区别与具体应用场景。

信号量机制

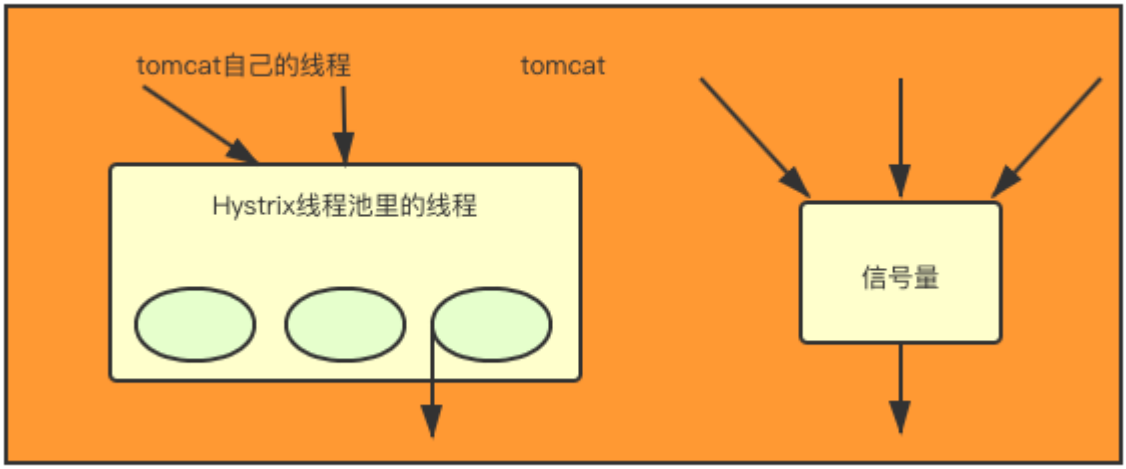
信号量的资源隔离只是起到一个开关的作用，比如，服务 A 的信号量大小为 10，那么就是说它同时只允许有 10 个 tomcat 线程来访问服务 A，其它的请求都会被拒绝，从而达到资源隔离和限流保护的作用。



线程池与信号量区别

线程池隔离技术，并不是说去控制类似 tomcat 这种 web 容器的线程。更加严格的意义上来说，Hystrix 的线程池隔离技术，控制的是 tomcat 线程的执行。Hystrix 线程池满后，会确保说，tomcat 的线程不会因为依赖服务的接口调用延迟或故障而被 hang 住，tomcat 其它的线程不会卡死，可以快速返回，然后支撑其它的事情。

线程池隔离技术，是用 Hystrix 自己的线程去执行调用；而信号量隔离技术，是直接让 tomcat 线程去调用依赖服务。信号量隔离，只是一道关卡，信号量有多少，就允许多少个 tomcat 线程通过它，然后去执行。



适用场景：

- **线程池技术**，适合绝大多数场景，比如说我们对依赖服务的网络请求的调用和访问、需要对调用的 timeout 进行控制（捕捉 timeout 超时异常）。
- **信号量技术**，适合说你的访问不是对外部依赖的访问，而是对内部的一些比较复杂的业务逻辑的访问，并且系统内部的代码，其实不涉及任何的网络请求，那么只要做信号量的普通限流就可以了，因为不需要去捕获 timeout 类似的问题。

信号量简单 Demo

业务背景里，比较适合信号量的是什么场景呢?

比如说，我们一般来说，缓存服务，可能会将一些量特别少、访问又特别频繁的数据，放在自己的纯内存中。

举个栗子。一般我们在获取到商品数据之后，都要去获取商品是属于哪个地理位置、省、市、卖家等，可能在自己的纯内存中，比如就一个 Map 去获取。对于这种直接访问本地内存的逻辑，比较适合用信号量做一下简单的隔离。

优点在于，不用自己管理线程池啦，不用 care timeout 超时啦，也不需要进行线程的上下文切换啦。信号量做隔离的话，性能相对来说会高一些。

假如这是本地缓存，我们可以通过 cityId，拿到 cityName。

```java
public class LocationCache {
    private static Map<Long, String> cityMap = new HashMap<>();

    static {
        cityMap.put(1L, "北京");
    }
```

```
    /**
     * 通过cityId 获取 cityName
     *
     * @param cityId 城市id
     * @return 城市名
     */
    public static String getCityName(Long cityId) {
        return cityMap.get(cityId);
    }
}
```

写一个 GetCityNameCommand，策略设置为**信号量**。run() 方法中获取本地缓存。我们目的就是对获取本地缓存的代码进行资源隔离。

java

```
public class GetCityNameCommand extends HystrixCommand<String> {

    private Long cityId;

    public GetCityNameCommand(Long cityId) {
        // 设置信号量隔离策略
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Ge
                .andCommandPropertiesDefaults(HystrixCommandProperties.Set
                        .withExecutionIsolationStrategy(HystrixCommandProp

        this.cityId = cityId;
    }

    @Override
    protected String run() {
        // 需要进行信号量隔离的代码
        return LocationCache.getCityName(cityId);
    }
}
```

在接口层，通过创建 GetCityNameCommand，传入 cityId，执行 execute() 方法，那么获取本地 cityName 缓存的代码将会进行信号量的资源隔离。

java

```
@RequestMapping("/getProductInfo")
@ResponseBody
```

```
public String getProductInfo(Long productId) {
    HystrixCommand<ProductInfo> getProductInfoCommand = new GetProductInfo

    // 通过command执行，获取最新商品数据
    ProductInfo productInfo = getProductInfoCommand.execute();

    Long cityId = productInfo.getCityId();

    GetCityNameCommand getCityNameCommand = new GetCityNameCommand(cityId)
    // 获取本地内存(cityName)的代码会被信号量进行资源隔离
    String cityName = getCityNameCommand.execute();

    productInfo.setCityName(cityName);

    System.out.println(productInfo);
    return "success";
}
```

Hystrix 隔离策略细粒度控制

Hystrix 实现资源隔离，有两种策略：

- 线程池隔离
- 信号量隔离

对资源隔离这一块东西，其实可以做一定细粒度的一些控制。

execution.isolation.strategy

指定了 HystrixCommand.run() 的资源隔离策略： `THREAD` or `SEMAPHORE` ，一种基于线程池，一种基于信号量。

java

```
// to use thread isolation
HystrixCommandProperties.Setter().withExecutionIsolationStrategy(Execution

// to use semaphore isolation
HystrixCommandProperties.Setter().withExecutionIsolationStrategy(Execution
```

线程池机制，每个 command 运行在一个线程中，限流是通过线程池的大小来控制的；信号量机制，command 是运行在调用线程中，通过信号量的容量来进行限流。

如何在线程池和信号量之间做选择?

默认的策略就是线程池。

线程池其实最大的好处就是对于网络访问请求，如果有超时的话，可以避免调用线程阻塞住。

而使用信号量的场景，通常是针对超大并发量的场景下，每个服务实例每秒都几百的 QPS ，那么此时你用线程池的话，线程一般不会太多，可能撑不住那么高的并发，如果要撑住，可能要耗费大量的线程资源，那么就是用信号量，来进行限流保护。一般用信号量常见于那种基于纯内存的一些业务逻辑服务，而不涉及到任何网络访问请求。

command key & command group

我们使用线程池隔离，要怎么对**依赖服务**、**依赖服务接口**、**线程池**三者做划分呢?

每一个 command，都可以设置一个自己的名称 command key，同时可以设置一个自己的组 command group。

```java
private static final Setter cachedSetter = Setter.withGroupKey(HystrixComm
                                        .andCommandKey(HystrixComm

public CommandHelloWorld(String name) {
    super(cachedSetter);
    this.name = name;
}
```

command group 是一个非常重要的概念，默认情况下，就是通过 command group 来定义一个线程池的，而且还会通过 command group 来聚合一些监控和报警信息。同一个 command group 中的请求，都会进入同一个线程池中。

command thread pool

ThreadPoolKey 代表了一个 HystrixThreadPool，用来进行统一监控、统计、缓存。默认的 ThreadPoolKey 就是 command group 的名称。每个 command 都会跟它的 ThreadPoolKey 对应的 ThreadPool 绑定在一起。

如果不想直接用 command group，也可以手动设置 ThreadPool 的名称。

```
private static final Setter cachedSetter = Setter.withGroupKey(HystrixComm
                                                .andCommandKey(HystrixComm
                                                .andThreadPoolKey(HystrixT

public CommandHelloWorld(String name) {
    super(cachedSetter);
    this.name = name;
}
```

command key & command group & command thread pool

command key，代表了一类 command，一般来说，代表了底层的依赖服务的一个接口。

command group，代表了某一个底层的依赖服务，这是很合理的，一个依赖服务可能会暴露出来多个接口，每个接口就是一个 command key。command group 在逻辑上去组织起来一堆 command key 的调用、统计信息、成功次数、timeout 超时次数、失败次数等，可以看到某一个服务整体的一些访问情况。一般来说，**推荐**根据一个服务区划分出一个线程池，command key 默认都是属于同一个线程池的。

比如说你以一个服务为粒度，估算出来这个服务每秒的所有接口加起来的整体 `QPS` 在 100 左右，你调用这个服务，当前这个服务部署了 10 个服务实例，每个服务实例上，其实用这个 command group 对应这个服务，给一个线程池，量大概在 10 个左右就可以了，你对整个服务的整体的访问 QPS 就大概在每秒 100 左右。

但是，如果说 command group 对应了一个服务，而这个服务暴露出来的几个接口，访问量很不一样，差异非常之大。你可能就希望在这个服务 command group 内部，包含的对应多个接口的 command key，做一些细粒度的资源隔离。就是说，对同一个服务的不同接口，使用不同的线程池。

```
command key -> command group

command key -> 自己的 thread pool key
```

逻辑上来说，多个 command key 属于一个command group，在做统计的时候，会放在一起统计。每个 command key 有自己的线程池，每个接口有自己的线程池，去做资源隔离和限流。

说白点，就是说如果你的 command key 要用自己的线程池，可以定义自己的 thread pool key，就 ok 了。

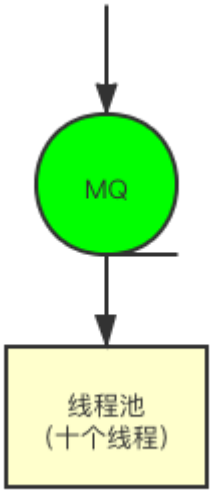
coreSize

设置线程池的大小，默认是 10。一般来说，用这个默认的 10 个线程大小就够了。

```java
HystrixThreadPoolProperties.Setter().withCoreSize(int value);
```

queueSizeRejectionThreshold

如果说线程池中的 10 个线程都在工作中，没有空闲的线程来做其它的事情，此时再有请求过来，会先进入队列积压。如果说队列积压满了，再有请求过来，就直接 reject，拒绝请求，执行 fallback 降级的逻辑，快速返回。



控制 queue 满了之后 reject 的 threshold，因为 maxQueueSize 不允许热修改，因此提供这个参数可以热修改，控制队列的最大大小。

```java
HystrixThreadPoolProperties.Setter().withQueueSizeRejectionThreshold(int v
```

execution.isolation.semaphore.maxConcurrentRequests

设置使用 SEMAPHORE 隔离策略的时候允许访问的最大并发量，超过这个最大并发量，请求直接被 reject。

这个并发量的设置，跟线程池大小的设置，应该是类似的，但是基于信号量的话，性能会好很多，而且 Hystrix 框架本身的开销会小很多。

默认值是 10，尽量设置的小一些，因为一旦设置的太大，而且有延时发生，可能瞬间导致 tomcat 本身的线程资源被占满。

```java
HystrixCommandProperties.Setter().withExecutionIsolationSemaphoreMaxConcur
```

深入 Hystrix 执行时内部原理

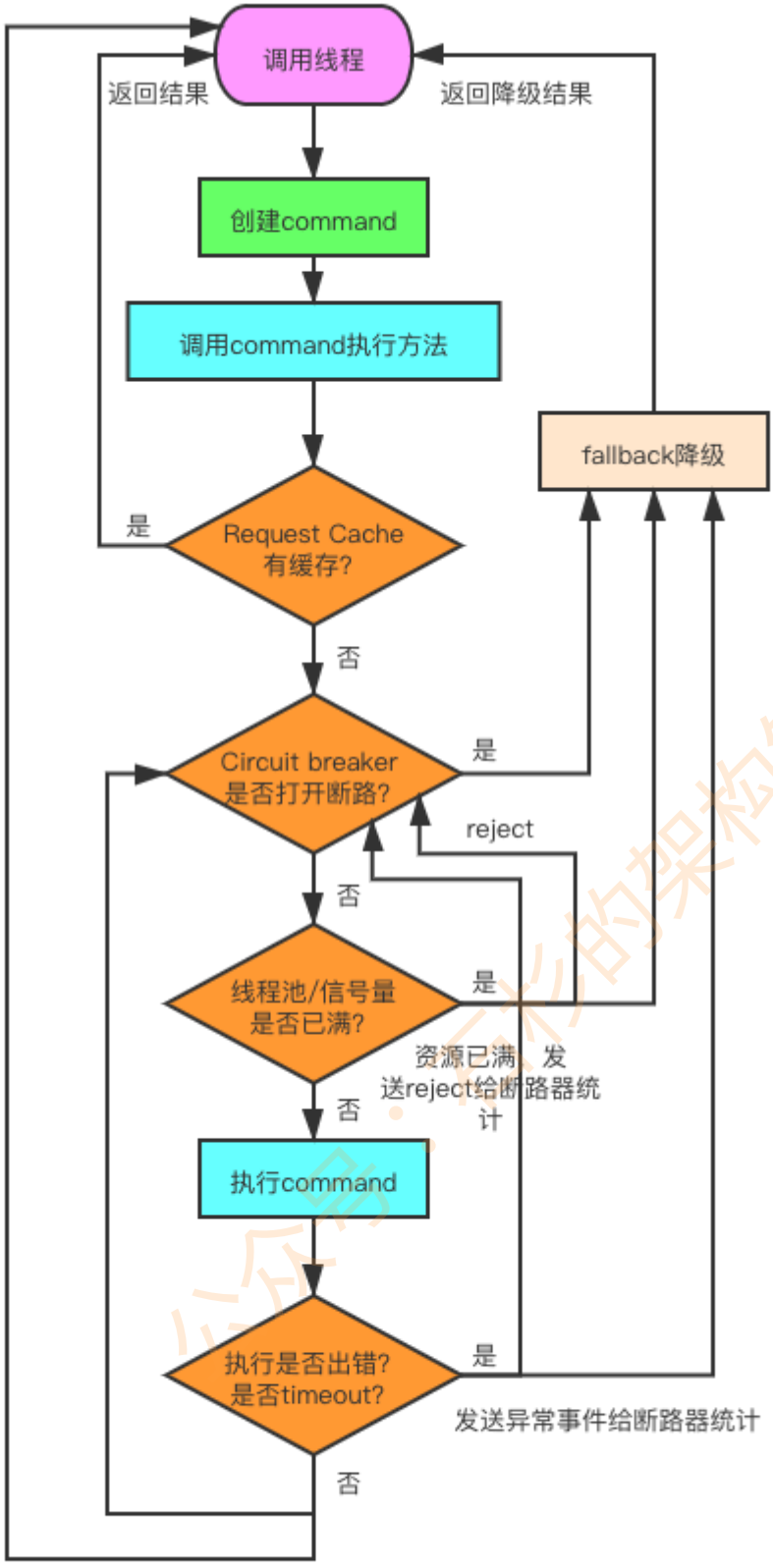
前面我们了解了 Hystrix 最基本的支持高可用的技术：**资源隔离** + **限流**。

- 创建 command；
- 执行这个 command；
- 配置这个 command 对应的 group 和线程池。

这里，我们要讲一下，你开始执行这个 command，调用了这个 command 的 execute() 方法之后，Hystrix 底层的执行流程和步骤以及原理是什么。

在讲解这个流程的过程中，我会带出来 Hystrix 其他的一些核心以及重要的功能。

这里是整个 8 大步骤的流程图，我会对每个步骤进行细致的讲解。学习的过程中，对照着这个流程图，相信思路会比较清晰。



步骤一：创建 command

一个 HystrixCommand 或 HystrixObservableCommand 对象，代表了对某个依赖服务发起的一次请求或者调用。创建的时候，可以在构造函数中传入任何需要的参数。

- HystrixCommand 主要用于仅仅会返回一个结果的调用。
- HystrixObservableCommand 主要用于可能会返回多条结果的调用。

```java
// 创建 HystrixCommand
HystrixCommand hystrixCommand = new HystrixCommand(arg1, arg2);

// 创建 HystrixObservableCommand
HystrixObservableCommand hystrixObservableCommand = new HystrixObservableC
```

步骤二：调用 command 执行方法

执行 command，就可以发起一次对依赖服务的调用。

要执行 command，可以在 4 个方法中选择其中的一个：execute()、queue()、observe()、toObservable()。

其中 execute() 和 queue() 方法仅仅对 HystrixCommand 适用。

- execute()：调用后直接 block 住，属于同步调用，直到依赖服务返回单条结果，或者抛出异常。
- queue()：返回一个 Future，属于异步调用，后面可以通过 Future 获取单条结果。
- observe()：订阅一个 Observable 对象，Observable 代表的是依赖服务返回的结果，获取到一个那个代表结果的 Observable 对象的拷贝对象。
- toObservable()：返回一个 Observable 对象，如果我们订阅这个对象，就会执行 command 并且获取返回结果。

```java
K             value    = hystrixCommand.execute();
Future<K>     fValue   = hystrixCommand.queue();
Observable<K> oValue   = hystrixObservableCommand.observe();
Observable<K> toOValue = hystrixObservableCommand.toObservable();
```

execute() 实际上会调用 queue().get() 方法，可以看一下 Hystrix 源码。

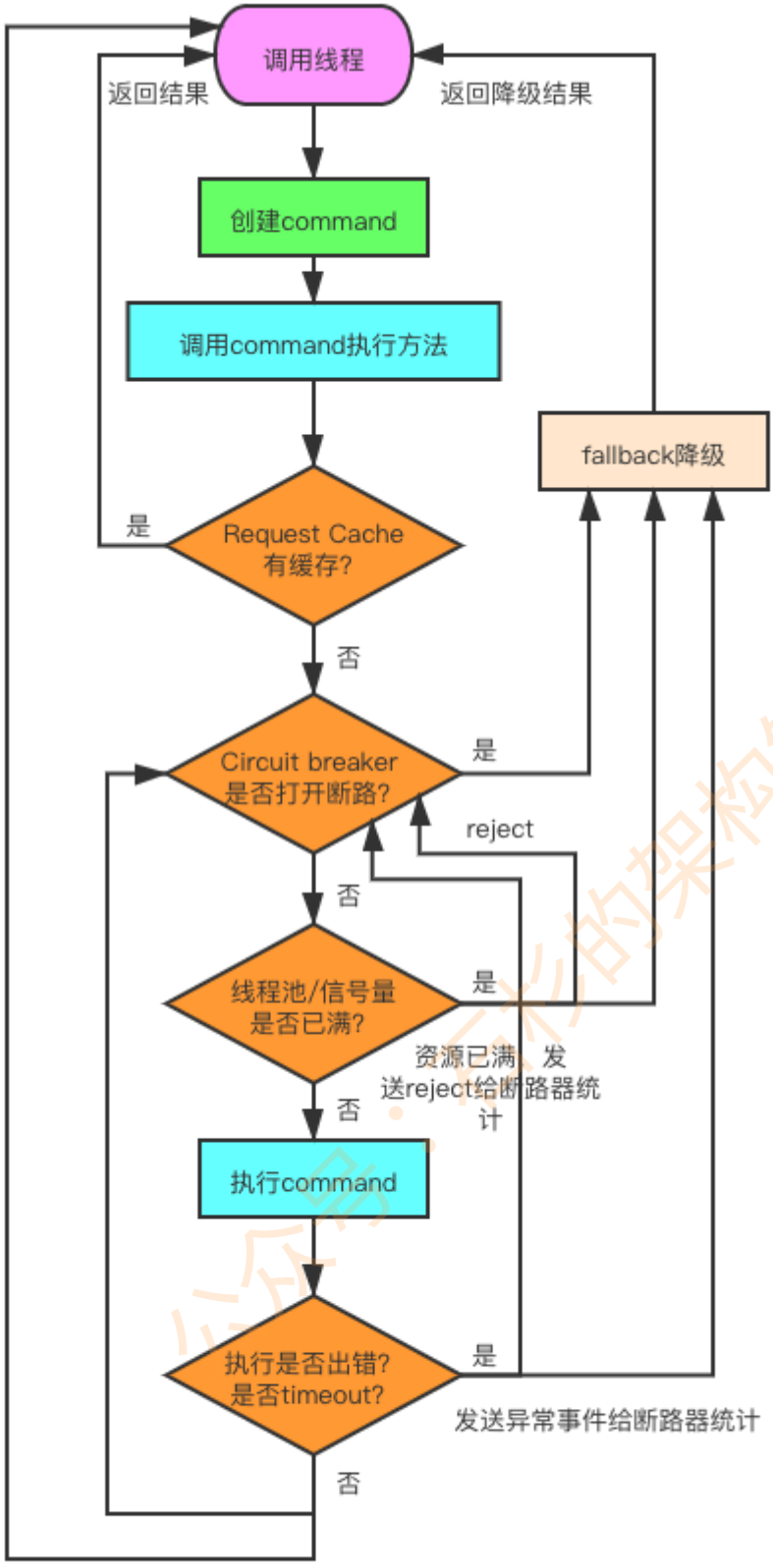
```java
public R execute() {
    try {
        return queue().get();
    } catch (Exception e) {
        throw Exceptions.sneakyThrow(decomposeException(e));
```

```
        }
    }
```

而在 queue() 方法中，会调用 toObservable().toBlocking().toFuture()。

```java
    final Future<R> delegate = toObservable().toBlocking().toFuture();
```

也就是说，先通过 toObservable() 获得 Future 对象，然后调用 Future 的 get() 方法。那么，其实无论是哪种方式执行 command，最终都是依赖于 toObservable() 去执行的。



步骤三：检查是否开启缓存

从这一步开始，就进入到 Hystrix 底层运行原理啦，看一下 Hystrix 一些更高级的功能和特性。

如果这个 command 开启了请求缓存 Request Cache，而且这个调用的结果在缓存中存在，那么直接从缓存中返回结果。否则，继续往后的步骤。

步骤四：检查是否开启了断路器

检查这个 command 对应的依赖服务是否开启了断路器。如果断路器被打开了，那么 Hystrix 就不会执行这个 command，而是直接去执行 fallback 降级机制，返回降级结果。

步骤五：检查线程池/队列/信号量是否已满

如果这个 command 线程池和队列已满，或者 semaphore 信号量已满，那么也不会执行 command，而是直接去调用 fallback 降级机制，同时发送 reject 信息给断路器统计。

步骤六：执行 command

调用 HystrixObservableCommand 对象的 construct() 方法，或者 HystrixCommand 的 run() 方法来实际执行这个 command。

- HystrixCommand.run() 返回单条结果，或者抛出异常。

```java
// 通过command执行，获取最新一条商品数据
ProductInfo productInfo = getProductInfoCommand.execute();
```

- HystrixObservableCommand.construct() 返回一个 Observable 对象，可以获取多条结果。

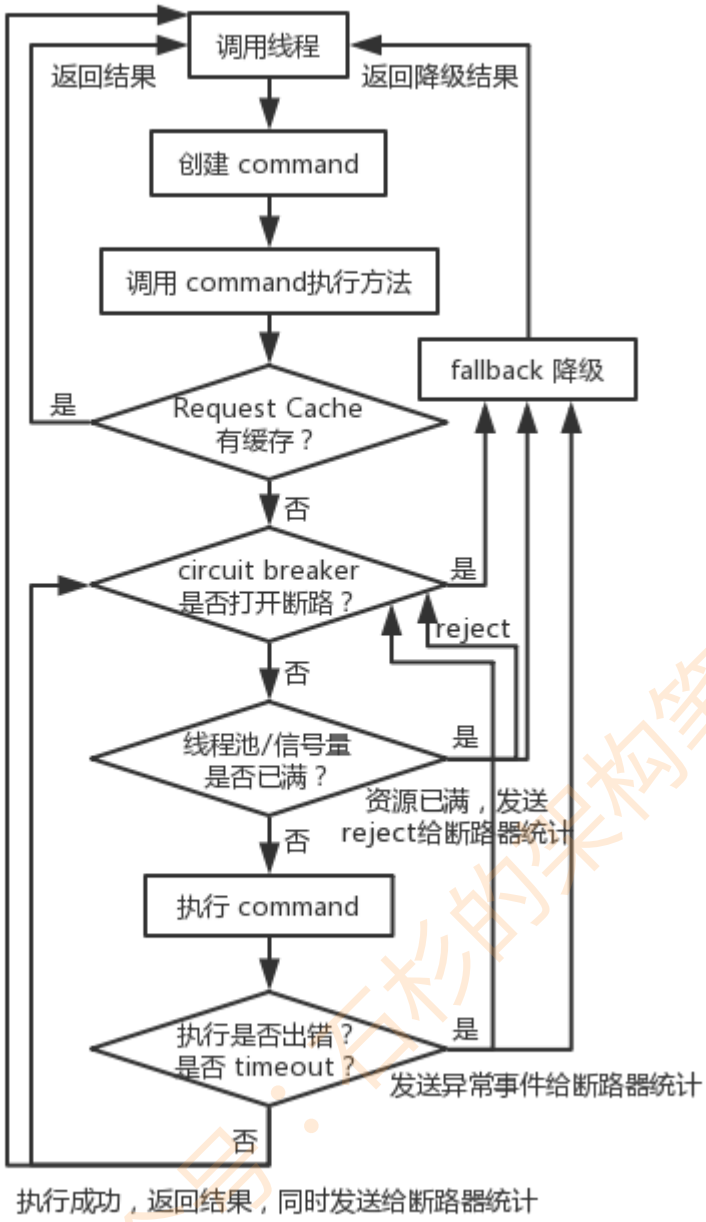
```java
Observable<ProductInfo> observable = getProductInfosCommand.observe();

// 订阅获取多条结果
observable.subscribe(new Observer<ProductInfo>() {
    @Override
    public void onCompleted() {
        System.out.println("获取完了所有的商品数据");
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }
```

```
        /**
         * 获取完一条数据，就回调一次这个方法
         *
         * @param productInfo 商品信息
         */
        @Override
        public void onNext(ProductInfo productInfo) {
            System.out.println(productInfo);
        }
    });
```

如果是采用线程池方式，并且 HystrixCommand.run() 或者 HystrixObservableCommand.construct() 的执行时间超过了 timeout 时长的话，那么 command 所在的线程会抛出一个 TimeoutException，这时会执行 fallback 降级机制，不会去管 run() 或 construct() 返回的值了。另一种情况，如果 command 执行出错抛出了其它异常，那么也会走 fallback 降级。这两种情况下，Hystrix 都会发送异常事件给断路器统计。

注意，我们是不可能终止掉一个调用严重延迟的依赖服务的线程的，只能说给你抛出来一个 TimeoutException。

如果没有 timeout，也正常执行的话，那么调用线程就会拿到一些调用依赖服务获取到的结果，然后 Hystrix 也会做一些 logging 记录和 metric 度量统计。



步骤七：断路健康检查

Hystrix 会把每一个依赖服务的调用成功、失败、Reject、Timeout 等事件发送给 circuit breaker 断路器。断路器就会对这些事件的次数进行统计，根据异常事件发生的比例来决定是否要进行断路（熔断）。如果打开了断路器，那么在接下来一段时间内，会直接断路，返回降级结果。

如果在之后，断路器尝试执行 command，调用没有出错，返回了正常结果，那么 Hystrix 就会把断路器关闭。

步骤八：调用 fallback 降级机制

在以下几种情况中，Hystrix 会调用 fallback 降级机制。

- 断路器处于打开状态；
- 线程池/队列/semaphore满了；
- command 执行超时；
- run() 或者 construct() 抛出异常。

一般在降级机制中，都建议给出一些默认的返回值，比如静态的一些代码逻辑，或者从内存中的缓存中提取一些数据，在这里尽量不要再进行网络请求了。

在降级中，如果一定要进行网络调用的话，也应该将那个调用放在一个 HystrixCommand 中进行隔离。

- HystrixCommand 中，实现 getFallback() 方法，可以提供降级机制。
- HystrixObservableCommand 中，实现 resumeWithFallback() 方法，返回一个 Observable 对象，可以提供降级结果。

如果没有实现 fallback，或者 fallback 抛出了异常，Hystrix 会返回一个 Observable，但是不会返回任何数据。

不同的 command 执行方式，其 fallback 为空或者异常时的返回结果不同。

- 对于 execute()，直接抛出异常。
- 对于 queue()，返回一个 Future，调用 get() 时抛出异常。
- 对于 observe()，返回一个 Observable 对象，但是调用 subscribe() 方法订阅它时，立即抛出调用者的 onError() 方法。
- 对于 toObservable()，返回一个 Observable 对象，但是调用 subscribe() 方法订阅它时，立即抛出调用者的 onError() 方法。

不同的执行方式

- execute()，获取一个 Future.get()，然后拿到单个结果。
- queue()，返回一个 Future。
- observe()，立即订阅 Observable，然后启动 8 大执行步骤，返回一个拷贝的 Observable，订阅时立即回调给你结果。
- toObservable()，返回一个原始的 Observable，必须手动订阅才会去执行 8 大步骤。

基于 request cache 请求缓存技术优化批量商品数据查询接口

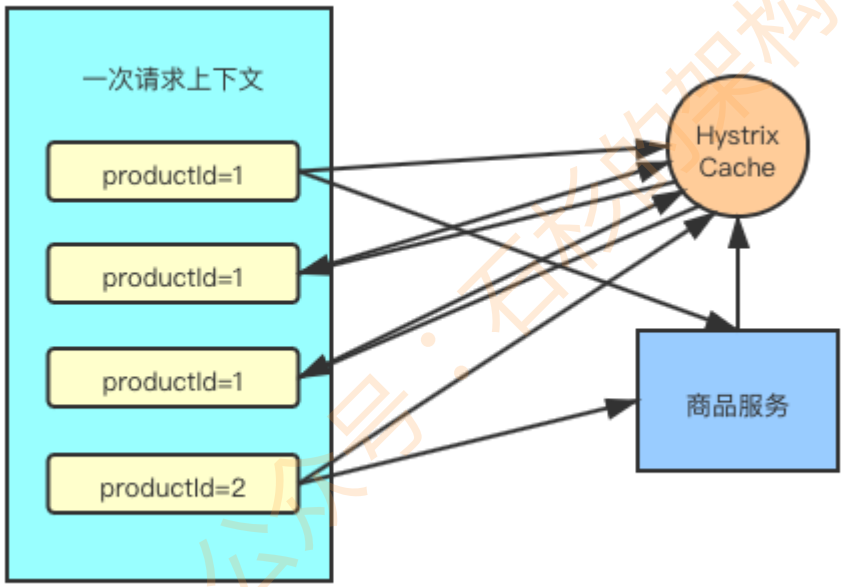
Hystrix command 执行时 8 大步骤第三步，就是检查 Request cache 是否有缓存。

首先，有一个概念，叫做 Request Context 请求上下文，一般来说，在一个 web 应用中，如果我们用到了 Hystrix，我们会在一个 filter 里面，对每一个请求都施加一个请求上下文。就是说，每一次请求，就是一次请求上下文。然后在这次请求上下文中，我们会去执行 N 多代码，调用 N 多依赖服务，有的依赖服务可能还会调用好几次。

在一次请求上下文中，如果有多个 command，参数都是一样的，调用的接口也是一样的，而结果可以认为也是一样的。那么这个时候，我们可以让第一个 command 执行返回的结果缓存在内存中，然后这个请求上下文后续的其它对这个依赖的调用全部从内存中取出缓存结果就可以了。

这样的话，好处在于不用在一次请求上下文中反复多次执行一样的 command，**避免重复执行网络请求，提升整个请求的性能**。

举个栗子。比如说我们在一次请求上下文中，请求获取 productId 为 1 的数据，第一次缓存中没有，那么会从商品服务中获取数据，返回最新数据结果，同时将数据缓存在内存中。后续同一次请求上下文中，如果还有获取 productId 为 1 的数据的请求，直接从缓存中取就好了。



HystrixCommand 和 HystrixObservableCommand 都可以指定一个缓存 key，然后 Hystrix 会自动进行缓存，接着在同一个 request context 内，再次访问的话，就会直接取用缓存。

下面，我们结合一个具体的**业务场景**，来看一下如何使用 request cache 请求缓存技术。当然，以下代码只作为一个基本的 Demo 而已。

现在，假设我们要做一个**批量查询商品数据**的接口，在这个里面，我们是用 HystrixCommand 一次性批量查询多个商品 id 的数据。但是这里有个问题，如果说 Nginx 在本地缓存失效了，重新获取一批缓存，传递过来的 productIds 都没有进行去重，比如 `productIds=1,1,1,2,2` ，那么可能说，商品 id 出现了重复，如果按照我们之前的业务逻辑，可能就会重复对 productId=1 的商品查询三次，productId=2 的商品查询两次。

我们对批量查询商品数据的接口，可以用 request cache 做一个优化，就是说一次请求，就是一次 request context，对相同的商品查询只执行一次，其余重复的都走 request cache。

实现 Hystrix 请求上下文过滤器并注册

定义 HystrixRequestContextFilter 类，实现 Filter 接口。

java

```
/**
 * Hystrix 请求上下文过滤器
 */
public class HystrixRequestContextFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse se
        HystrixRequestContext context = HystrixRequestContext.initializeCo
        try {
            filterChain.doFilter(servletRequest, servletResponse);
        } catch (IOException | ServletException e) {
            e.printStackTrace();
        } finally {
            context.shutdown();
        }
    }

    @Override
    public void destroy() {

    }
}
```

然后将该 filter 对象注册到 SpringBoot Application 中。

java

```
@SpringBootApplication
public class EshopApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(EshopApplication.class, args);
    }

    @Bean
    public FilterRegistrationBean filterRegistrationBean() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistra
        filterRegistrationBean.addUrlPatterns("/*");
        return filterRegistrationBean;
    }
}
```

command 重写 getCacheKey() 方法

在 GetProductInfoCommand 中，重写 getCacheKey() 方法，这样的话，每一次请求的结果，都会放在 Hystrix 请求上下文中。下一次同一个 productId 的数据请求，直接取缓存，无须再调用 run() 方法。

java

```
public class GetProductInfoCommand extends HystrixCommand<ProductInfo> {

    private Long productId;

    private static final HystrixCommandKey KEY = HystrixCommandKey.Factory

    public GetProductInfoCommand(Long productId) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Pr
                .andCommandKey(KEY));
        this.productId = productId;
    }

    @Override
    protected ProductInfo run() {
        String url = "http://localhost:8081/getProductInfo?productId=" + p
        String response = HttpClientUtils.sendGetRequest(url);
        System.out.println("调用接口查询商品数据，productId=" + productId);
        return JSONObject.parseObject(response, ProductInfo.class);
    }

    /**
     * 每次请求的结果，都会放在Hystrix绑定的请求上下文上
```

```
     *
     * @return cacheKey 缓存key
     */
    @Override
    public String getCacheKey() {
        return "product_info_" + productId;
    }

    /**
     * 将某个商品id的缓存清空
     *
     * @param productId 商品id
     */
    public static void flushCache(Long productId) {
        HystrixRequestCache.getInstance(KEY,
                HystrixConcurrencyStrategyDefault.getInstance()).clear("pr
    }
}
```

这里写了一个 flushCache() 方法，用于我们开发手动删除缓存。

controller 调用 command 查询商品信息

在一次 web 请求上下文中，传入商品 id 列表，查询多条商品数据信息。对于每个 productId，都创建一个 command。

如果 id 列表没有去重，那么重复的 id，第二次查询的时候就会直接走缓存。

java

```
@Controller
public class CacheController {

    /**
     * 一次性批量查询多条商品数据的请求
     *
     * @param productIds 以,分隔的商品id列表
     * @return 响应状态
     */
    @RequestMapping("/getProductInfos")
    @ResponseBody
    public String getProductInfos(String productIds) {
        for (String productId : productIds.split(",")) {
```

```
                // 对每个productId，都创建一个command
                GetProductInfoCommand getProductInfoCommand = new GetProductIn
                ProductInfo productInfo = getProductInfoCommand.execute();
                System.out.println("是否是从缓存中取的结果：" + getProductInfoComm
            }

            return "success";
        }
    }
```

发起请求

调用接口，查询多个商品的信息。

```
http://localhost:8080/getProductInfos?productIds=1,1,1,2,2,5
```

在控制台，我们可以看到以下结果。

```
调用接口查询商品数据，productId=1
是否是从缓存中取的结果：false
是否是从缓存中取的结果：true
是否是从缓存中取的结果：true
调用接口查询商品数据，productId=2
是否是从缓存中取的结果：false
是否是从缓存中取的结果：true
调用接口查询商品数据，productId=5
是否是从缓存中取的结果：false
```

第一次查询 productId=1 的数据，会调用接口进行查询，不是从缓存中取结果。而随后再出现查询 productId=1 的请求，就直接取缓存了，这样的话，效率明显高很多。

删除缓存

我们写一个 UpdateProductInfoCommand，在更新商品信息之后，手动调用之前写的 flushCache()，手动将缓存删除。

```java
public class UpdateProductInfoCommand extends HystrixCommand<Boolean> {

    private Long productId;

    public UpdateProductInfoCommand(Long productId) {
        super(HystrixCommandGroupKey.Factory.asKey("UpdateProductInfoGroup
        this.productId = productId;
    }

    @Override
    protected Boolean run() throws Exception {
        // 这里执行一次商品信息的更新
        // ...

        // 然后清空缓存
        GetProductInfoCommand.flushCache(productId);
        return true;
    }
}
```

这样，以后查询该商品的请求，第一次就会走接口调用去查询最新的商品信息。

基于本地缓存的 fallback 降级机制

Hystrix 出现以下四种情况，都会去调用 fallback 降级机制：

- 断路器处于打开的状态。
- 资源池已满（线程池+队列 / 信号量）。
- Hystrix 调用各种接口，或者访问外部依赖，比如 MySQL、Redis、Zookeeper、Kafka 等等，出现了任何异常的情况。
- 访问外部依赖的时候，访问时间过长，报了 TimeoutException 异常。

两种最经典的降级机制

- 纯内存数据
  在降级逻辑中，你可以在内存中维护一个 ehcache，作为一个纯内存的基于 LRU 自动清理的缓存，让数据放在缓存内。如果说外部依赖有异常，fallback 这里直接尝试从 ehcache 中获取数据。

- 默认值
  fallback 降级逻辑中，也可以直接返回一个默认值。

在 `HystrixCommand` ，降级逻辑的书写，是通过实现 getFallback() 接口；而在 `HystrixObservableCommand` 中，则是实现 resumeWithFallback() 方法。

现在，我们用一个简单的栗子，来演示 fallback 降级是怎么做的。

比如，有这么个**场景**。我们现在有个包含 brandId 的商品数据，假设正常的逻辑是这样：拿到一个商品数据，根据 brandId 去调用品牌服务的接口，获取品牌的最新名称 brandName。

假如说，品牌服务接口挂掉了，那么我们可以尝试从本地内存中，获取一份稍过期的数据，先凑合着用。

步骤一：本地缓存获取数据

本地获取品牌名称的代码大致如下。

```java
/**
 * 品牌名称本地缓存
 *
 */

public class BrandCache {

    private static Map<Long, String> brandMap = new HashMap<>();

    static {
        brandMap.put(1L, "Nike");
    }

    /**
     * brandId 获取 brandName
     *
     * @param brandId 品牌id
     * @return 品牌名
     */
    public static String getBrandName(Long brandId) {
        return brandMap.get(brandId);
    }
```

步骤二：实现 GetBrandNameCommand

在 GetBrandNameCommand 中，run() 方法的正常逻辑是去调用品牌服务的接口获取到品牌名称，如果调用失败，报错了，那么就会去调用 fallback 降级机制。

这里，我们直接**模拟接口调用报错**，给它抛出个异常。

而在 getFallback() 方法中，就是我们的**降级逻辑**，我们直接从本地的缓存中，**获取到品牌名称**的数据。

java

```
/**
 * 获取品牌名称的command
 *
 */

public class GetBrandNameCommand extends HystrixCommand<String> {

    private Long brandId;

    public GetBrandNameCommand(Long brandId) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Br
                .andCommandKey(HystrixCommandKey.Factory.asKey("GetBrandNa
                .andCommandPropertiesDefaults(HystrixCommandProperties.Set
                        // 设置降级机制最大并发请求数
                        .withFallbackIsolationSemaphoreMaxConcurrentReques
        this.brandId = brandId;
    }

    @Override
    protected String run() throws Exception {
        // 这里正常的逻辑应该是去调用一个品牌服务的接口获取名称
        // 如果调用失败，报错了，那么就会去调用fallback降级机制

        // 这里我们直接模拟调用报错，抛出异常
        throw new Exception();
    }

    @Override
    protected String getFallback() {
        return BrandCache.getBrandName(brandId);
    }
}
```

`FallbackIsolationSemaphoreMaxConcurrentRequests` 用于设置 fallback 最大允许的并发请求量，默认值是 10，是通过 semaphore 信号量的机制去限流的。如果超出了这个最大值，那么直接 reject。

步骤三：CacheController 调用接口

在 CacheController 中，我们通过 productInfo 获取 brandId，然后创建 GetBrandNameCommand 并执行，去尝试获取 brandName。这里执行会报错，因为我们在 run() 方法中直接抛出异常，Hystrix 就会去调用 getFallback() 方法走降级逻辑。

java

```
@Controller
public class CacheController {

    @RequestMapping("/getProductInfo")
    @ResponseBody
    public String getProductInfo(Long productId) {
        HystrixCommand<ProductInfo> getProductInfoCommand = new GetProduct

        ProductInfo productInfo = getProductInfoCommand.execute();
        Long brandId = productInfo.getBrandId();

        HystrixCommand<String> getBrandNameCommand = new GetBrandNameComma

        // 执行会抛异常报错，然后走降级
        String brandName = getBrandNameCommand.execute();
        productInfo.setBrandName(brandName);

        System.out.println(productInfo);
        return "success";
    }
}
```

关于降级逻辑的演示，基本上就结束了。

深入 Hystrix 断路器执行原理

RequestVolumeThreshold

java

```
HystrixCommandProperties.Setter()
    .withCircuitBreakerRequestVolumeThreshold(int)
```

表示在滑动窗口中，至少有多少个请求，才可能触发断路。

Hystrix 经过断路器的流量超过了一定的阈值，才有可能触发断路。比如说，要求在 10s 内经过断路器的流量必须达到 20 个，而实际经过断路器的流量才 10 个，那么根本不会去判断要不要断路。

ErrorThresholdPercentage

java

```
HystrixCommandProperties.Setter()
    .withCircuitBreakerErrorThresholdPercentage(int)
```

表示异常比例达到多少，才会触发断路，默认值是 50(%)。

如果断路器统计到的异常调用的占比超过了一定的阈值，比如说在 10s 内，经过断路器的流量达到了 30 个，同时其中异常访问的数量也达到了一定的比例，比如 60% 的请求都是异常（报错 / 超时 / reject），就会开启断路。

SleepWindowInMilliseconds

java

```
HystrixCommandProperties.Setter()
    .withCircuitBreakerSleepWindowInMilliseconds(int)
```

断路开启，也就是由 close 转换到 open 状态（close -> open）。那么之后在 `SleepWindowInMilliseconds` 时间内，所有经过该断路器的请求全部都会被断路，不调用后端服务，直接走 fallback 降级机制。

而在该参数时间过后，断路器会变为 `half-open` 半开闭状态，尝试让一条请求经过断路器，看能不能正常调用。如果调用成功了，那么就自动恢复，断路器转为 close 状态。

Enabled

```java
HystrixCommandProperties.Setter()
    .withCircuitBreakerEnabled(boolean)
```

控制是否允许断路器工作，包括跟踪依赖服务调用的健康状况，以及对异常情况过多时是否允许触发断路。默认值是 `true` 。

ForceOpen

```java
HystrixCommandProperties.Setter()
    .withCircuitBreakerForceOpen(boolean)
```

如果设置为 true 的话，直接强迫打开断路器，相当于是手动断路了，手动降级，默认值是 `false` 。

ForceClosed

```java
HystrixCommandProperties.Setter()
    .withCircuitBreakerForceClosed(boolean)
```

如果设置为 true，直接强迫关闭断路器，相当于手动停止断路了，手动升级，默认值是 `false` 。

实例 Demo

HystrixCommand 配置参数

在 GetProductInfoCommand 中配置 Setter 断路器相关参数。

- 滑动窗口中，最少 20 个请求，才可能触发断路。
- 异常比例达到 40% 时，才触发断路。
- 断路后 3000ms 内，所有请求都被 reject，直接走 fallback 降级，不会调用 run() 方法。3000ms 过后，变为 half-open 状态。

run() 方法中，我们判断一下 productId 是否为 -1，是的话，直接抛出异常。这么写，我们之后测试的时候就可以传入 productId=-1，**模拟服务执行异常**了。

在降级逻辑中，我们直接给它返回降级商品就好了。

```java
public class GetProductInfoCommand extends HystrixCommand<ProductInfo> {

    private Long productId;

    private static final HystrixCommandKey KEY = HystrixCommandKey.Factory

    public GetProductInfoCommand(Long productId) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Pr
                .andCommandKey(KEY)
                .andCommandPropertiesDefaults(HystrixCommandProperties.Set
                        // 是否允许断路器工作
                        .withCircuitBreakerEnabled(true)
                        // 滑动窗口中，最少有多少个请求，才可能触发断路
                        .withCircuitBreakerRequestVolumeThreshold(20)
                        // 异常比例达到多少，才触发断路，默认50%
                        .withCircuitBreakerErrorThresholdPercentage(40)
                        // 断路后多少时间内直接reject请求，之后进入half-open状态，
                        .withCircuitBreakerSleepWindowInMilliseconds(3000)
        this.productId = productId;
    }

    @Override
    protected ProductInfo run() throws Exception {
        System.out.println("调用接口查询商品数据，productId=" + productId);

        if (productId == -1L) {
            throw new Exception();
        }

        String url = "http://localhost:8081/getProductInfo?productId=" + p
        String response = HttpClientUtils.sendGetRequest(url);
        return JSONObject.parseObject(response, ProductInfo.class);
    }

    @Override
    protected ProductInfo getFallback() {
        ProductInfo productInfo = new ProductInfo();
        productInfo.setName("降级商品");
```

```
        return productInfo;
    }
}
```

断路测试类

我们在测试类中，前 30 次请求，传入 productId=-1，然后休眠 3s，之后 70 次请求，传入 productId=1。

java

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class CircuitBreakerTest {

    @Test
    public void testCircuitBreaker() {
        String baseURL = "http://localhost:8080/getProductInfo?productId="

        for (int i = 0; i < 30; ++i) {
            // 传入-1，会抛出异常，然后走降级逻辑
            HttpClientUtils.sendGetRequest(baseURL + "-1");
        }

        TimeUtils.sleep(3);
        System.out.println("After sleeping...");

        for (int i = 31; i < 100; ++i) {
            // 传入1，走服务正常调用
            HttpClientUtils.sendGetRequest(baseURL + "1");
        }
    }
}
```

测试结果

测试结果，我们可以明显看出系统断路与恢复的整个过程。

c

```
调用接口查询商品数据，productId=-1
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specifica
```

```
// ...
// 这里重复打印了 20 次上面的结果


ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
// ...
// 这里重复打印了 8 次上面的结果


// 休眠 3s 后
调用接口查询商品数据，productId=1
ProductInfo(id=1, name=iphone7手机, price=5599.0, pictureList=a.jpg,b.jpg,
// ...
// 这里重复打印了 69 次上面的结果
```

前 30 次请求，我们传入的 productId 为 -1，所以服务执行过程中会抛出异常。我们设置了最少 20 次请求通过断路器并且异常比例超出 40% 就触发断路。因此执行了 21 次接口调用，每次都抛异常并且走降级，21 次过后，断路器就被打开了。

之后的 9 次请求，都不会执行 run() 方法，也就不会打印以下信息。

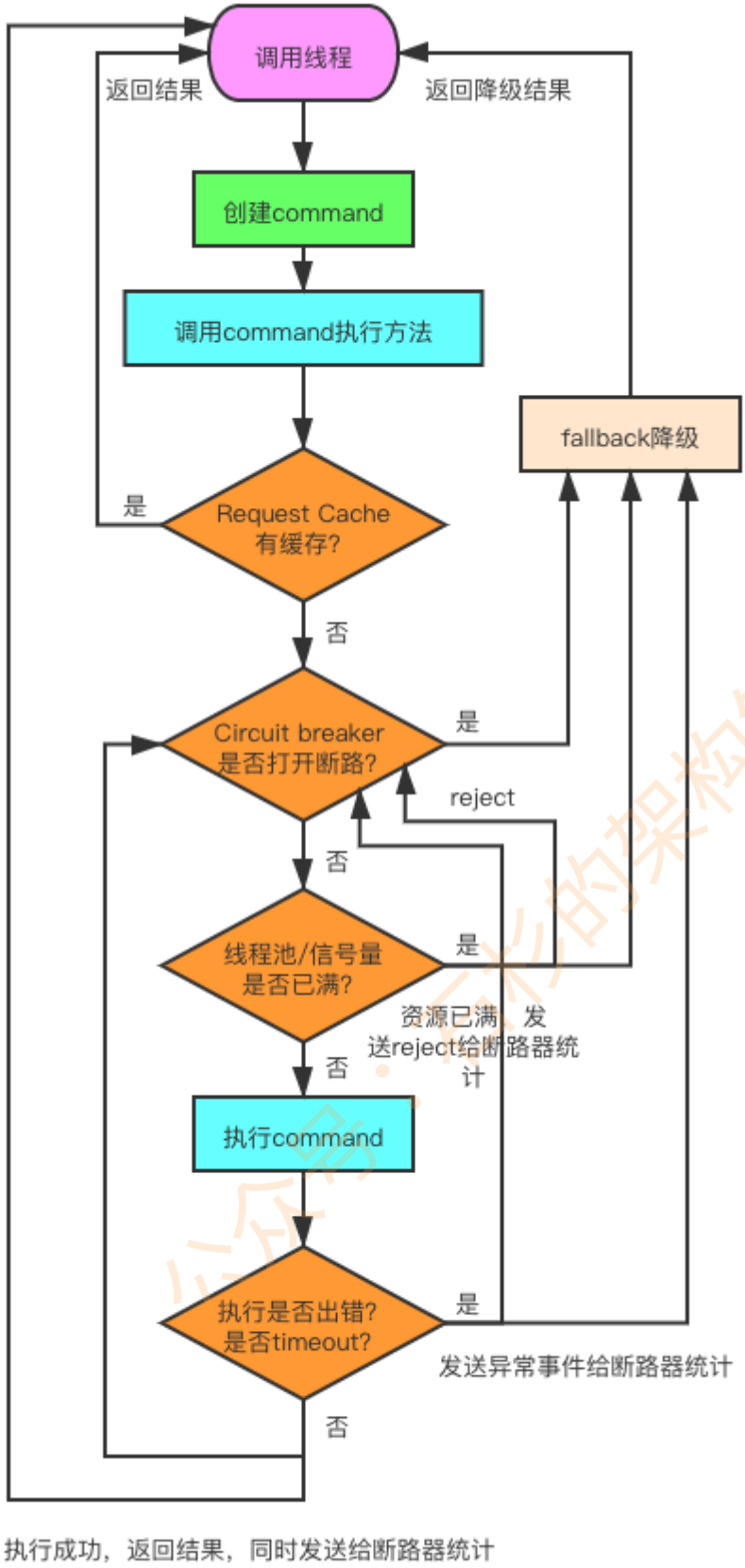
```
调用接口查询商品数据，productId=-1
```

而是直接走降级逻辑，调用 getFallback() 执行。

休眠了 3s 后，我们在之后的 70 次请求中，都传入 productId 为 1。由于我们前面设置了 3000ms 过后断路器变为 `half-open` 状态。因此 Hystrix 会尝试执行请求，发现成功了，那么断路器关闭，之后的所有请求也都能正常调用了。

深入 Hystrix 线程池隔离与接口限流

前面讲了 Hystrix 的 request cache 请求缓存、fallback 优雅降级、circuit breaker 断路器快速熔断，这一讲，我们来详细说说 Hystrix 的线程池隔离与接口限流。



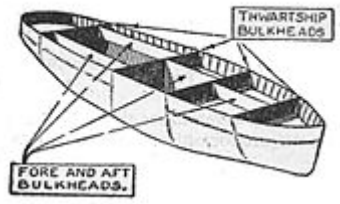
Hystrix 通过判断线程池或者信号量是否已满，超出容量的请求，直接 Reject 走降级，从而达到限流的作用。

限流是限制对后端的服务的访问量，比如说你对 MySQL、Redis、Zookeeper 以及其它各种后端中间件的资源的访问的限制，其实是为了避免过大的流量直接打死后端的服务。

线程池隔离技术的设计

Hystrix 采用了 Bulkhead Partition 舱壁隔离技术，来将外部依赖进行资源隔离，进而避免任何外部依赖的故障导致本服务崩溃。

舱壁隔离，是说将船体内部空间区隔划分成若干个隔舱，一旦某几个隔舱发生破损进水，水流不会在其间相互流动，如此一来船舶在受损时，依然能具有足够的浮力和稳定性，进而减低立即沉船的危险。



Hystrix 对每个外部依赖用一个单独的线程池，这样的话，如果对那个外部依赖调用延迟很严重，最多就是耗尽那个依赖自己的线程池而已，不会影响其他的依赖调用。

Hystrix 应用线程池机制的场景

- 每个服务都会调用几十个后端依赖服务，那些后端依赖服务通常是由很多不同的团队开发的。
- 每个后端依赖服务都会提供它自己的 client 调用库，比如说用 thrift 的话，就会提供对应的 thrift 依赖。
- client 调用库随时会变更。
- client 调用库随时可能会增加新的网络请求的逻辑。
- client 调用库可能会包含诸如自动重试、数据解析、内存中缓存等逻辑。
- client 调用库一般都对调用者来说是个黑盒，包括实现细节、网络访问、默认配置等等。
- 在真实的生产环境中，经常会出现调用者，突然间惊讶的发现，client 调用库发生了某些变化。
- 即使 client 调用库没有改变，依赖服务本身可能有会发生逻辑上的变化。
- 有些依赖的 client 调用库可能还会拉取其他的依赖库，而且可能那些依赖库配置的不正确。
- 大多数网络请求都是同步调用的。
- 调用失败和延迟，也有可能会发生在 client 调用库本身的代码中，不一定就是发生在网络请求中。

简单来说，就是你必须默认 client 调用库很不靠谱，而且随时可能发生各种变化，所以就要用强制隔离的方式来确保任何服务的故障不会影响当前服务。

线程池机制的优点

- 任何一个依赖服务都可以被隔离在自己的线程池内，即使自己的线程池资源填满了，也不会影响任何其他的服务调用。
- 服务可以随时引入一个新的依赖服务，因为即使这个新的依赖服务有问题，也不会影响其他任何服务的调用。
- 当一个故障的依赖服务重新变好的时候，可以通过清理掉线程池，瞬间恢复该服务的调用，而如果是 tomcat 线程池被占满，再恢复就很麻烦。
- 如果一个 client 调用库配置有问题，线程池的健康状况随时会报告，比如成功/失败/拒绝/超时的次数统计，然后可以近实时热修改依赖服务的调用配置，而不用停机。
- 基于线程池的异步本质，可以在同步的调用之上，构建一层异步调用层。

简单来说，最大的好处，就是资源隔离，确保说任何一个依赖服务故障，不会拖垮当前的这个服务。

线程池机制的缺点

- 线程池机制最大的缺点就是增加了 CPU 的开销。
  除了 tomcat 本身的调用线程之外，还有 Hystrix 自己管理的线程池。

- 每个 command 的执行都依托一个独立的线程，会进行排队，调度，还有上下文切换。

- Hystrix 官方自己做了一个多线程异步带来的额外开销统计，通过对比多线程异步调用+同步调用得出，Netflix API 每天通过 Hystrix 执行 10 亿次调用，每个服务实例有 40 个以上的线程池，每个线程池有 10 个左右的线程。）最后发现说，用 Hystrix 的额外开销，就是给请求带来了 3ms 左右的延时，最多延时在 10ms 以内，相比于可用性和稳定性的提升，这是可以接受的。

我们可以用 Hystrix semaphore 技术来实现对某个依赖服务的并发访问量的限制，而不是通过线程池/队列的大小来限制流量。

semaphore 技术可以用来限流和削峰，但是不能用来对调研延迟的服务进行 timeout 和隔离。

`execution.isolation.strategy` 设置为 `SEMAPHORE` ，那么 Hystrix 就会用 semaphore 机制来替代线程池机制，来对依赖服务的访问进行限流。如果通过 semaphore 调用的时候，底层的网络调用延迟很严重，那么是无法 timeout 的，只能一直 block 住。一旦请求数量超过了 semaphore 限定的数量之后，就会立即开启限流。

接口限流 Demo

假设一个线程池大小为 8，等待队列的大小为 10。timeout 时长我们设置长一些，20s。

在 command 内部，写死代码，做一个 sleep，比如 sleep 3s。

- withCoreSize：设置线程池大小。

- withMaxQueueSize：设置等待队列大小。
- withQueueSizeRejectionThreshold：这个与 withMaxQueueSize 配合使用，等待队列的大小，取得是这两个参数的较小值。

如果只设置了线程池大小，另外两个 queue 相关参数没有设置的话，等待队列是处于关闭的状态。

```java
public class GetProductInfoCommand extends HystrixCommand<ProductInfo> {

    private Long productId;

    private static final HystrixCommandKey KEY = HystrixCommandKey.Factory

    public GetProductInfoCommand(Long productId) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Pr
                .andCommandKey(KEY)
                // 线程池相关配置信息
                .andThreadPoolPropertiesDefaults(HystrixThreadPoolProperti
                        // 设置线程池大小为8
                        .withCoreSize(8)
                        // 设置等待队列大小为10
                        .withMaxQueueSize(10)
                        .withQueueSizeRejectionThreshold(12))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Set
                        .withCircuitBreakerEnabled(true)
                        .withCircuitBreakerRequestVolumeThreshold(20)
                        .withCircuitBreakerErrorThresholdPercentage(40)
                        .withCircuitBreakerSleepWindowInMilliseconds(3000)
                        // 设置超时时间
                        .withExecutionTimeoutInMilliseconds(20000)
                        // 设置fallback最大请求并发数
                        .withFallbackIsolationSemaphoreMaxConcurrentReques
        this.productId = productId;
    }

    @Override
    protected ProductInfo run() throws Exception {
        System.out.println("调用接口查询商品数据，productId=" + productId);

        if (productId == -1L) {
            throw new Exception();
        }
```

```
        // 请求过来，会在这里hang住3秒钟
        if (productId == -2L) {
            TimeUtils.sleep(3);
        }

        String url = "http://localhost:8081/getProductInfo?productId=" + p
        String response = HttpClientUtils.sendGetRequest(url);
        System.out.println(response);
        return JSONObject.parseObject(response, ProductInfo.class);
    }

    @Override
    protected ProductInfo getFallback() {
        ProductInfo productInfo = new ProductInfo();
        productInfo.setName("降级商品");
        return productInfo;
    }
}
```

我们模拟 25 个请求。前 8 个请求，调用接口时会直接被 hang 住 3s，那么后面的 10 个请求会先进入等待队列中等待前面的请求执行完毕。最后的 7 个请求过来，会直接被 reject，调用 fallback 降级逻辑。

java

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class RejectTest {

    @Test
    public void testReject() {
        for (int i = 0; i < 25; ++i) {
            new Thread(() -> HttpClientUtils.sendGetRequest("http://localh
        }
        // 防止主线程提前结束执行
        TimeUtils.sleep(50);
    }
}
```

从执行结果中，我们可以明显看出一共打印出了 7 个降级商品。这也就是请求数超过线程池+队列的数量而直接被 reject 的结果。

```
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
调用接口查询商品数据，productId=-2
ProductInfo(id=null, name=降级商品, price=null, pictureList=null, specificat
{"id": -2, "name": "iphone7手机", "price": 5599, "pictureList":"a.jpg,b.jpg
// 后面都是一些正常的商品信息，就不贴出来了
//...
```

基于 timeout 机制为服务接口调用超时提供安全保护

一般来说，在调用依赖服务的接口的时候，比较常见的一个问题就是**超时**。超时是在一个复杂的分布式系统中，导致系统不稳定，或者系统抖动。出现大量超时，线程资源会被 hang 死，从而导致吞吐量大幅度下降，甚至服务崩溃。

你去调用各种各样的依赖服务，特别是在大公司，你甚至都不认识开发一个服务的人，你都不知道那个人的技术水平怎么样，对那个人根本不了解。

Peter Steiner 说过，"On the Internet, nobody knows you're a dog"，也就是说在互联网的另外一头，你都不知道甚至坐着一条狗。

像特别复杂的分布式系统，特别是在大公司里，多个团队、大型协作，你可能都不知道服务是谁的，很可能说开发服务的那个哥儿们甚至是一个实习生。依赖服务的接口性能可能很不稳定，有时候 2ms，有时候 200ms，甚至 2s，都有可能。

如果你不对各种依赖服务接口的调用做超时控制，来给你的服务提供安全保护措施，那么很可能你的服务就被各种垃圾的依赖服务的性能给拖死了。大量的接口调用很慢，大量的线程被卡

死。如果你做了资源的隔离，那么也就是线程池的线程被卡死，但其实我们可以做超时控制，没必要让它们全卡死。

TimeoutMilliseconds

在 Hystrix 中，我们可以手动设置 timeout 时长，如果一个 command 运行时间超过了设定的时长，那么就被认为是 timeout，然后 Hystrix command 标识为 timeout，同时执行 fallback 降级逻辑。

`TimeoutMilliseconds` 默认值是 1000，也就是 1000ms。

```java
HystrixCommandProperties.Setter()
    ..withExecutionTimeoutInMilliseconds(int)
```

TimeoutEnabled

这个参数用于控制是否要打开 timeout 机制，默认值是 true。

```java
HystrixCommandProperties.Setter()
    .withExecutionTimeoutEnabled(boolean)
```

实例 Demo

我们在 command 中，将超时时间设置为 500ms，然后在 run() 方法中，设置休眠时间 1s，这样一个请求过来，直接休眠 1s，结果就会因为超时而执行降级逻辑。

```java
public class GetProductInfoCommand extends HystrixCommand<ProductInfo> {

    private Long productId;

    private static final HystrixCommandKey KEY = HystrixCommandKey.Factory

    public GetProductInfoCommand(Long productId) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Pr
                .andCommandKey(KEY)
                .andThreadPoolPropertiesDefaults(HystrixThreadPoolProperti
```

```
                        .withCoreSize(8)
                        .withMaxQueueSize(10)
                        .withQueueSizeRejectionThreshold(8))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Set
                        .withCircuitBreakerEnabled(true)
                        .withCircuitBreakerRequestVolumeThreshold(20)
                        .withCircuitBreakerErrorThresholdPercentage(40)
                        .withCircuitBreakerSleepWindowInMilliseconds(3000)
                        // 设置是否打开超时，默认是true
                        .withExecutionTimeoutEnabled(true)
                        // 设置超时时间，默认1000(ms)
                        .withExecutionTimeoutInMilliseconds(500)
                        .withFallbackIsolationSemaphoreMaxConcurrentReques
        this.productId = productId;
    }

    @Override
    protected ProductInfo run() throws Exception {
        System.out.println("调用接口查询商品数据，productId=" + productId);

        // 休眠1s
        TimeUtils.sleep(1);

        String url = "http://localhost:8081/getProductInfo?productId=" + p
        String response = HttpClientUtils.sendGetRequest(url);
        System.out.println(response);
        return JSONObject.parseObject(response, ProductInfo.class);
    }

    @Override
    protected ProductInfo getFallback() {
        ProductInfo productInfo = new ProductInfo();
        productInfo.setName("降级商品");
        return productInfo;
    }
}
```

在测试类中，我们直接发起请求。

java

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class TimeoutTest {
```

本文档内容出自于 石杉的架构笔记 欢迎扫描下方二维码关注,回复 PDF，即可获取本项目离线 PDF 文档（656页精华内容）,学习更加方便!

石杉的架构笔记-