Zookeeper 入门文章

Java后端 5天前

以下文章来源于编程新说 ，作者编程新说李新杰

编程新说

现任架构师，已工作11年，Java技术栈，计算机基础，用心写文章，欢迎关注！

大家好，今天来讲讲zookeeper，其实很早就计划写关于它的文章，但是由于各种原因一直推到了今天。

本文会以类比的方式循序渐进、层层展开。各位坐稳了，让我们开启一段大脑的旅程。

边界的产生与突破

不觉间孩子已经上小学了，前段时间还参加了一次家长会，那就以学校和开会来说吧，这大家都很熟悉。

如果一个班要想开班会，那随时开都行，不需要提前安排与通知，因为一个班级从内部看就是一个整体，在班级内，同学之间以及与老师之间都可以随意交流，没有任何隔阂与阻碍。

一个班级从外部看就是一个独立的个体，因为班级与班级之间是完全独立的，因此一个班级的学生和老师都不会随便跑到其它班级去。这是因为存在着一个边界，即班级边界。

正是这个班级边界把班级隔开了，边界之内的事情，如班会，可以随便开展，因为它和边界之外的一切都无关。但是一旦涉及到边界之外，也就是跨边界，那么问题就产生了。

比如学校要开一个全体班级大会，肯定会提前安排好时间地点，以及各个班级在操场上的排列顺序，还要提前进行相应的通知。

为什么一个班的班会可以随时随地进行，而全体班级大会就要提前安排与通知呢？就是因为它跨了班级边界，是一个跨边界问题。

而且班级与班级之间互相独立，互相不太熟悉，可能沟通起来也不容易，因此需要提前安排好。

那如何通知呢？可以让班级之间互相通知，如一班通知二班，二班通知三班等等。也可以由一个独立于所有班级之外的人，如教务处或学生处的人，来依次通知所有班级。

这两种通知方法在现实中都有使用，所有没有绝对的好与坏之分，视情况而定即可。

读者需要明确这两种方法代表了处理此类问题的两种方式，一种是独立个体之间互相直接交流来解决，一种是需要第三方介入来协调解决。

这里可以得出一个结论，边界的产生是一种自然现象，而且通常边界不会被打碎或消失，但是可以通过其它手段让边界两边的事物进行交流协商，这顶多是算是一种“突破”吧。

计算机相关的边界产生与突破

上一小节的描述非常简单，相信所有人都能明白。接着就来说说和计算机相关的边界。其实有很多，我们就说一两种吧。

操作系统里面有内核空间和用户空间，它们之间是有边界的，但是它们之间依然是可以交流的，因为操作系统的开发者已经做好了交流的方式方法。

每个应用程序通常都是一个进程，由于应用之间通常差别较大，而且还有一些其它方面的考虑，如安全问题，所以进程之间是有边界的，即进程边界。

操作系统是按进程分配资源的，因此一个进程内部的线程共享这些资源。由于进程边界的存在，这些资源不能被别的进程使用。所以进程就像是一个班级。

由于不同进行之间通常不需要交流，就像班级之间通常也不怎么交流一样，所以默认情况下进程之间无法交流，这与操作系统的内核和用户空间是不同的。

但总归有特殊情况吧，如果进程间需要交流怎么办？那只能由开发人员自己想办法，如通过Socket，来实现。这种情况在中间件里很常见，如Nginx就涉及多个进程。

因为中间件的开发者一般都是牛X的人，他们能够搞定。但问题是绝大多数开发人员都是搞业务开发的，他们受能力、时间或金钱限制，往往做不出来生产级别的交流方法。

可是有时候业务人员开发的应用程序的进程之间也是需要交流的，就像要开全体班级大会那样，我们可以类比着来寻求解决方案。

我们可以让进程之间直接互相交流，就像班级之间互相通知那样，这一方面对开发人员要求高且费时费力，另一方面是当进程多了之后，它们之间的直接交流就变成了一张网，会很乱。

为了说明这一点，我们看个简单示例。假如张三、李四、王五是同事，周五下午下班时互相穿错了衣服，遗憾的是晚上回到家后才发现。他们都想在第二天，就是周六，换过来。

张三需要去找李四，李四需要去找王五，王五又需要去找张三，假设他们都住的相距较远，这会是一个颇为复杂的问题。那么如果有20个人都互相穿错了衣服呢，这将会是一个更加复杂的问题。

可以看出，如果个体之间互相直接交流的话，随着个体数目的增多，将会变得无比混乱与复杂。比较好的解决方法可能大家都想到了。

那就是约定一个合适的地方，如公司，张三、李四、王五都过去，互相交换完衣服后各自回家。这种方法随着个体的增多效果会越来越好。

其实这种方法就是全体班级开会时的第二种通知方法，由一个第三方无关人员介入来协调处理，此时这个第三方就是教务处或学生处。

那么对于多个进程之间的互相交流的解决方法也是这样的，由一个第三方无关进程介入来协调处理，此时这个第三方就是ZooKeeper。

这种方法还有一个好处，就是在一定程度上降低了个体的复杂性与要求，以及由此产生的额外问题。

比如有的班级的班主任脾气不好或不好说话，没有其它班级的班主任愿意去通知他，此时由教务处人员去通知，就不会有这个问题。

对于进程来说，降低了对业务开发人员的要求，不需要具备完整的进程间通信相关知识，同时降低了进程本身的复杂度，不需要支持完整的进程间通信，可能只需支持客户端即可。

这种方式的另一个好处是可以被抽象出来做成一个独立的中间件供大家使用，ZooKeeper就是这样的。

所以从本质来说，ZooKeeper就是一个第三方，也称中间人，它搭建了一个平台，让所有其它进程通过它来进行间接的交流。

ZooKeeper的数据模型

计算机其实就是用来处理或存储数据的，运行在它上面的软件大都也是如此。zookeeper作为多进程的协调者，肯定是跑不了了。

存储数据和摆放物品是一样的，不能随意乱扔，这样既占地方，又不好看，也难寻找。所以必须得有一定的层次结构。这就是计算机的专业课数据结构了。

最简单的数据结构就是数组或链表了。它们被称为线性表，是一维的，具有线性关系，即前后顺序，优点是简单，缺点是功能不够强大。

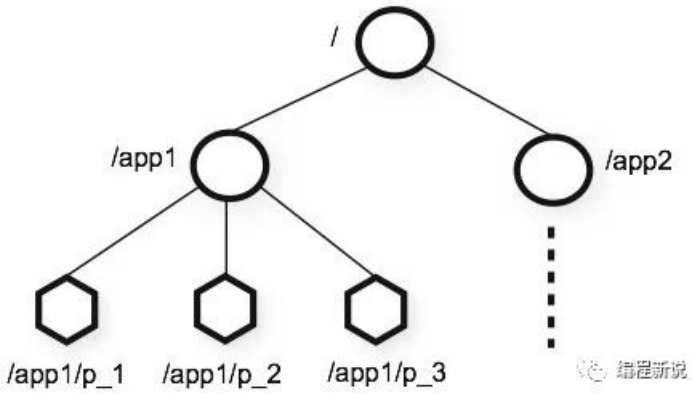
然后就是树了，可以认为它是两维的，左右是兄弟关系，上下是父子关系，因此具有从属关系。它是一个功能与复杂

度兼顾的结构。现实生活中的各类组织架构大都是树形的。

再复杂的就是图了，它是网状结构，可以认为是多维的，由于任何节点都可以连通，因此它表达一种多边关系。虽功能强大但也很复杂。现实中的铁路网和人际关系网大都是网状的。

当然，这是三大类数据结构，每一类中又可以分为很多种。比如树就有很多种变体，虽然都叫树，但有的差别还是很大的。

ZooKeeper选择了树作为自己存储数据的结构，其实它和文件系统也非常相似，如下图：



谈到数据就离不开增、删、改、查，对应树来说，增就是添加新的节点到树中，删就是从树中删除某个节点，改就是修改树中某个节点上存放的数据，查就是找到树中某个节点读取它上面存放的数据。

说白了就是树形表示的是一种结构，真正的数据是在节点上放着呢，叶子节点或非叶子节点都可以。

ZooKeeper应该具备的能力

我们从最常见的场景入手，从宏观上了解下zookeeper是如何使用的，以及它应该具备哪些能力。

场景一：

有两个应用程序进程A和B，A先处理数据，处理完后通知B，B再接着处理。我们应该如何利用zookeeper来完成这个呢？一起来分析一下。

首先，进程A连接上zookeeper，在上面创建一个节点来表示自己的存在，假设节点名称就叫foo吧。

然后在节点上设置一个数据叫doing，表示自己正在处理数据。过了一会处理完后，把节点上的数据更新为done。

这样进程A的工作就算完了。可是这怎么去影响到进程B呢？我们知道zookeeper完成的是进程间的间接交流，即进程之间是不碰面的。因此只能借助于这个树形里的节点。

进程B也要连上zookeeper，然后找到foo节点，看好它上面的数据是否由doing变成了done，如果是自己就开始处理数据，如果否那就继续等着。

问题是进程B不能自己老盯着foo节点啊，这样太累了，伤神，况且它还要做其它事情呢。那这个事情应该由谁来做呢？很显然是zookeeper嘛。

于是进程B就对zookeeper说，你给我盯着foo节点，什么时候变成done了通知我一声，我就开干了。

因此，zookeeper需要具有盯梢能力和通知其它进程的能力。这在zookeeper中对应一个专业术语，叫Watch。

Watch的作用和用法与上面描述的一样。就是进程B找到foo节点，在上面放一个Watch就可以了。

这样zookeeper就知道进程B对foo节点比较关注，于是zookeeper就盯着foo节点，一有风吹草动，马上通知进程B。

备注：关于Watch有非常多的细节问题，这里就不谈了。

需要注意的是，这个Watch是一次性的，即只能使用一次。也就是说，zookeeper通知过进程B之后，Watch就被用掉了，以后就不会再通知了。

如果进程B还需要被通知怎么办？很简单，那就在foo节点上再放一个新的Watch即可。如此这般下去，就可以保证一直被通知了。

我想这个Watch之所以被设计成一次性的，就是zookeeper不想让自己太累。睁着一双大眼，盯的东西太多太久的话，确实很累。

另外，zookeeper在通知进程B的时候，是可以把foo节点存放的数据一并发送过去的。

细心的朋友可能已经发现，zookeeper可以主动向进程B发通知或推数据，说明zookeeper和进程B之间的连接需要被一直保持。

因为进程B的位置比较随意，本来就是业务进程嘛。一旦连接断开，就像断了线的风筝，zookeeper再也无法找到进程B了。

不过zookeeper的位置是固定的，一旦连接断掉后，进程B可以再次向zookeeper发起连接请求，如果断开的时间足够短的话，进程B应该还可以在zookeeper上找回自己曾经拥有的一切。

这就涉及到了会话，因此zookeeper还要有一定的会话延续能力，方便在断开时间不长的时候找回原来的会话。

因此zookeeper应该有，监视节点、通知进程、保持长连接，会话延续等这样的能力。

场景二：

有时为了高可用或高性能，通常会把一个应用程序运行多份。假如运行了四份，那就是四个进程，分别是A、B、C、D。

当一个调用过来时，发现A、B、C、D都可以调，那就根据配置的负载均衡策略选出一个调用即可。

假设D进程所在的机器不幸掉电了，其实就是D挂了，那么此时再来一个调用的话，会发现只有A、B、C可以调，D自动就不存在了。

这其实就是Dubbo功能的一部分，那该如何基于zookeeper实现呢？照例一起分析下吧。

由于zookeeper是基于树形的数据结构，所以还是要拿节点说事。当进程A启动时，需要连接上zookeeper，然后创建一个节点来代表自己。

节点名称和节点上存放的数据可以根据实际情况来定，至少要包括该进程运行的IP和端口信息。进程B、C、D也做同样的事情。

如果让进程A、B、C、D的节点都位于同一个父节点下面，这样当一个调用过来后，只要找到这个父节点，读出它的所有子节点，就得到了所有可调用的进程信息。

如果某一时刻，进程D挂掉了，那么父节点下面进程D对应的那个节点应该会自动被zookeeper删除。这在zookeeper里有个专业术语，叫临时节点（Ephemeral Node）。那么与之对应的自然就是永久节点了。

其实工作过程是这样的，业务进程启动后与zookeeper建立连接，然后在zookeeper里创建临时节点并写入自己的相关信息。接着通过周期性的心跳和zookeeper保持住连接。

一旦业务进程挂掉，zookeeper将接受不到心跳了，那么在超过一定的时间后，zookeeper将会删除与之对应的临时节点，表示这个业务进程不再可用了。

Dubbo的做法是将接口名称和IP端口信息和我们设置的信息整合成一个类似URL的字符串，然后以这个字符串作为名称来创建临时节点。

临时节点不允许有孩子节点，只有永久节点才可以。

本文内容都非常简单，很容易理解，所以即使初次接触zookeeper的朋友，看到这里也算是入门了。

如果想再往深里讲的话，全部都是一些细节问题了。

（完）

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。同时标星（置顶）本公众号可以第一时间接受到博文推送。

推荐阅读

1. 女程序员做了个梦。。。
2. Spring 和 Spring Boot 之间到底有啥区别？
3. IDEA 新特性：提前知道代码怎么走
4. ping 命令还能这么玩？

Zookeeper 面试 23 连问，这些你都会吗？

JeffreyLcm Java后端 2月14日

作者 | JeffreyLcm

segmentfault.com/a/1190000014479433

1.ZooKeeper是什么?

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

2.ZooKeeper提供了什么?

1、文件系统
2、通知机制

3.Zookeeper文件系统

Zookeeper提供一个多层级的节点命名空间（节点称为znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得Zookeeper不能用于存放大量的数据，每个节点的存放数据上限为1M。

4.四种类型的znode

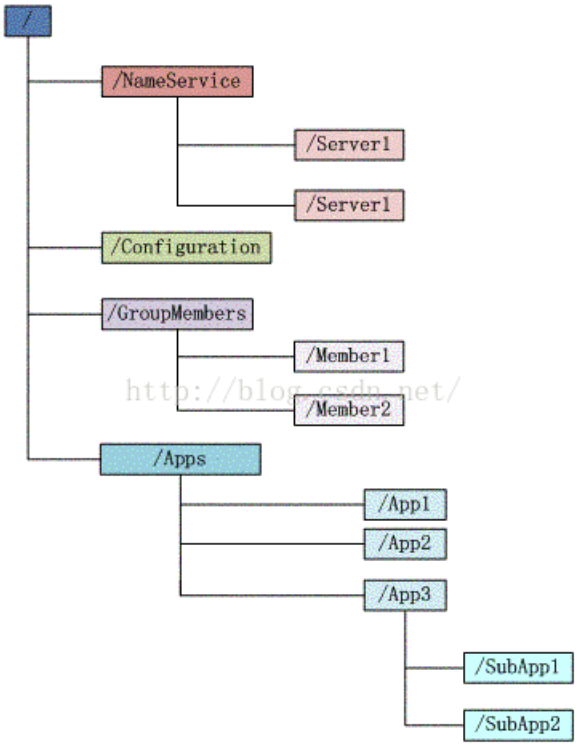
1、PERSISTENT-持久化目录节点
客户端与zookeeper断开连接后，该节点依旧存在

2、PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点
客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点
客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号



5.Zookeeper通知机制

client端会对某个znode建立一个watcher事件，当该znode发生变化时，这些client会收到zk的通知，然后client可以根据znode变化来做出业务上的改变等。

6.Zookeeper做了什么?

- 命名服务
- 配置管理
- 集群管理
- 分布式锁
- 队列管理

7.zk的命名服务（文件系统）

命名服务是指通过指定的名字来获取资源或者服务的地址，利用zk创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

8.zk的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在zk的znode下，当有配置发生改变时，也就是znode发生变化时，可以通过改变zk中某个目录节点的内容，利用watcher通知给各个客户端，从而更改配置。

9.Zookeeper集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：是否有机器退出和加入、选举master。

对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好。

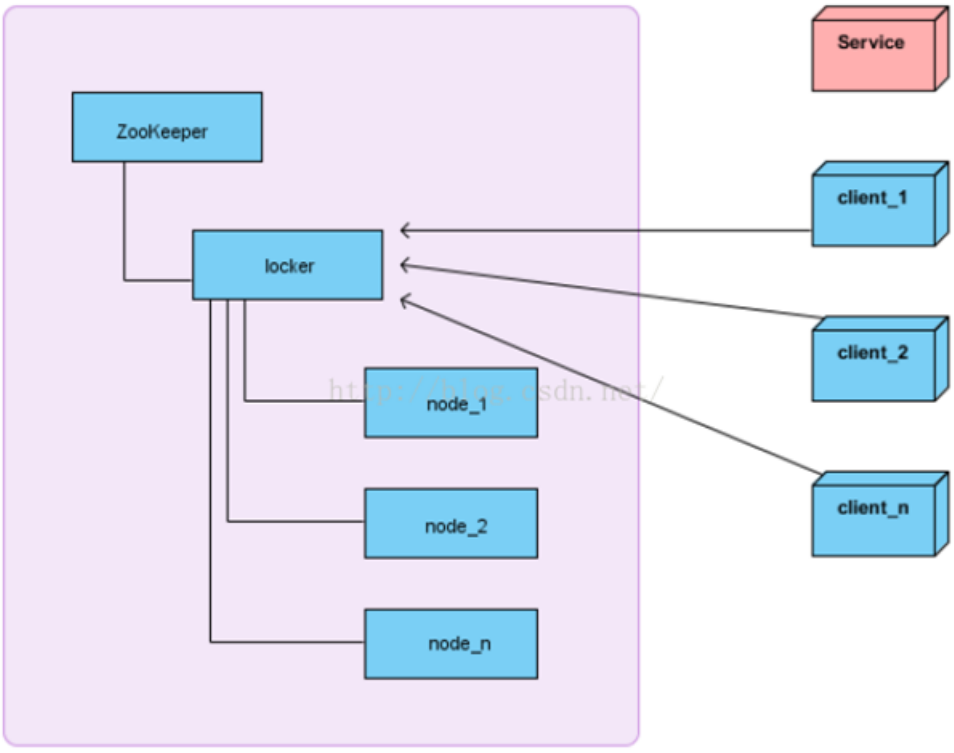
10.Zookeeper分布式锁（文件系统、通知机制）

有了zookeeper的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的distribute_lock 节点就释放出锁。

对于第二类， /distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，编号最小的获得锁，用完删除，依次方便。

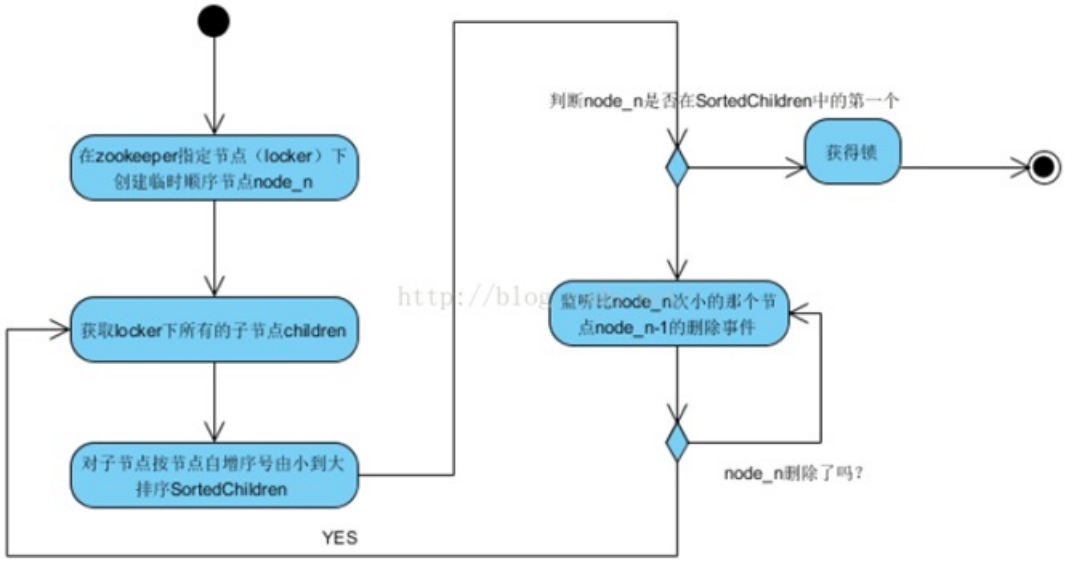
11.获取分布式锁的流程



在获取分布式锁的时候在locker节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用createNode方法在locker下创建临时顺序节点，然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。

客户端获取到所有的子节点path之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时对其注册事件监听器。

之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。



代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于BaseDistributedLock，实现了基于Zookeeper实现分布式锁的细节。

12.Zookeeper队列管理（文件系统、通知机制）

两种类型的队列：

- 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
- 队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建PERSISTENT_SEQUENTIAL节点，创建成功时Watcher通知等待的队列，队列删除序列号最小的节点用以消费。

此场景下Zookeeper的znode用于消息存储，znode存储的数据就是消息队列中的消息内容，SEQUENTIAL序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

13.Zookeeper数据复制

Zookeeper作为一个集群提供一致的数据服务，自然，它要在所有机器间做数据复制。数据复制的好处：

- 容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；
- 提高系统的扩展能力 ： 把负载分布到多个节点上，或者增加节点来提高系统的负载能力；
- 提高性能：让客户端本地访问就近的节点，提高用户访问速度。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

- 写主(WriteMaster) ：对数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；
- 写任意(Write Any)：对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对zookeeper来说，它采用的方式是写任意。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立observer的原因），而响应能力则取决于具体实现方式，是延迟复制保持最终一致性，还是立即复制快速响应。

14.Zookeeper工作原理

Zookeeper 的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。

Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和 leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

15.zookeeper是如何保证事务的顺序一致性的?

zookeeper采用了递增的事务Id来标识，所有的proposal（提议）都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch（时期; 纪元; 世; 新时代）用来标识leader是否发生改变，如果有新的leader产生出来，epoch会自增，低32位用来递增计数。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

16.Zookeeper 下 Server工作状态

每个Server在工作过程中有三种状态：

- LOOKING：当前Server不知道leader是谁，正在搜寻
- LEADING：当前Server即为选举出来的leader
- FOLLOWING：leader已经选举出来，当前Server与之同步

17.zookeeper是如何选取主leader的?

当leader崩溃或者leader失去大多数的follower，这时zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。

1、Zookeeper选主流程(basic paxos)

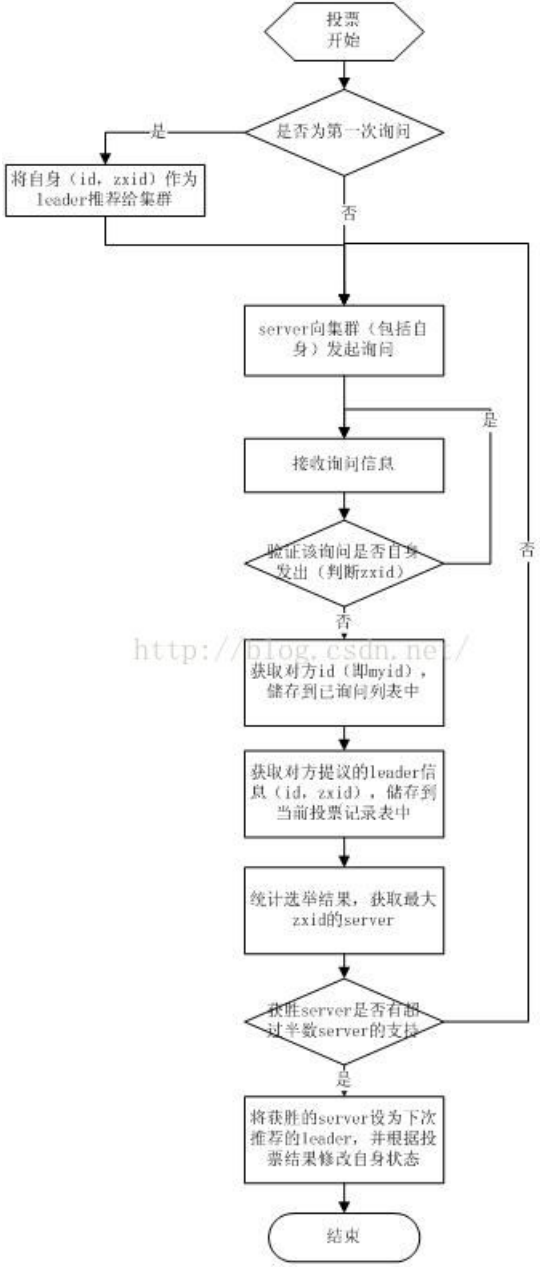
（1）选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

（2）选举线程首先向所有Server发起一次询问(包括自己)；

（3）选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

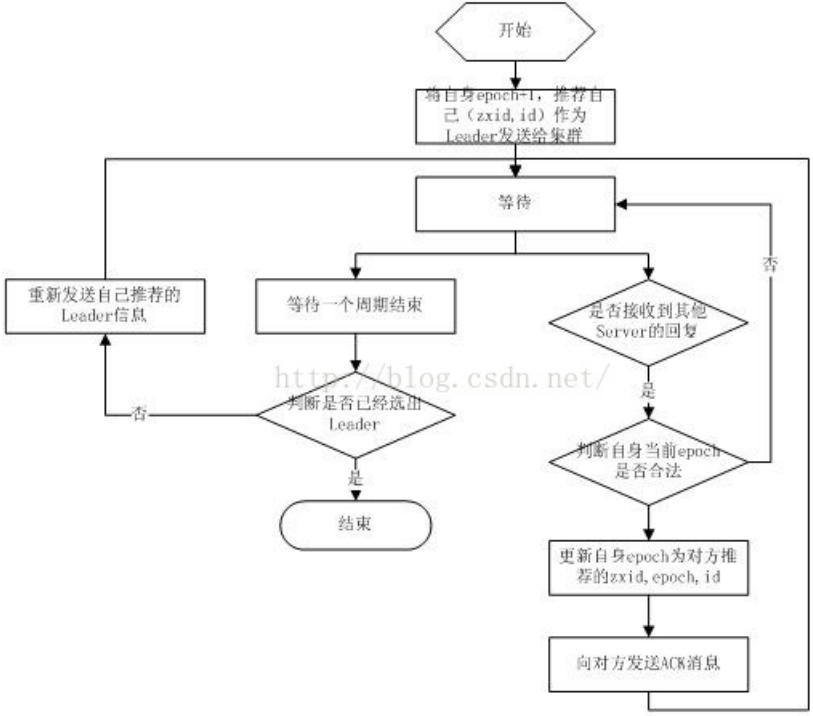
（4）收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

（5）线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数，设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1. 每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。



2、Zookeeper选主流程(basic paxos)

fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和 zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。



18.Zookeeper同步流程

选完Leader以后，zk就进入状态同步过程。

- Leader等待server连接；
- Follower连接leader，将最大的zxid发送给leader；
- Leader根据follower的zxid确定同步点；
- 完成同步后通知follower 已经成为uptodate状态；
- Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

19.分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后zk将这些变化发送给注册了这个节点的watcher的所有客户端。

对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

20.机器中为什么会有leader?

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行leader选举。

21.zk节点宕机如何处理?

Zookeeper本身也是集群，推荐配置不少于3个服务器。Zookeeper自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个Follower宕机，还有2台服务器提供访问，因为Zookeeper上的数据是有多个副本的，数据并不会丢失；

如果是一个Leader宕机，Zookeeper会选举出新的Leader。

ZK集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。
所以

- 3个节点的cluster可以挂掉1个节点(leader可以得到2票>1.5)
- 2个节点的cluster就不能挂掉任何1个节点了(leader可以得到1票<=1)

22.zookeeper负载均衡和nginx负载均衡区别

zk的负载均衡是可以调控，nginx只是能调权重，其他需要可控的都需要自己写插件；但是nginx的吞吐量比zk大很多，应该说按业务选择用哪种方式。

23.zookeeper watch机制

Watch机制官方声明：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。

Zookeeper机制的特点：

1、一次性触发数据发生改变时，一个watcher event会被发送到client，但是client只会收到一次这样的信息。

2、watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。

3、数据监视Zookeeper有数据监视和子数据监视getdata() and exists()设置数据监视，getchildren()设置了子节点监视。

4、注册watcher getData、exists、getChildren

5、触发watcher create、delete、setData

6、setData()会触发znode上设置的data watch（如果set成功的话）。一个成功的create() 操作会触发被创建的znode上的数据watch，以及其父节点上的child watch。而一个成功的delete()操作将会同时触发一个znode的data watch和child watch（因为这样就没有子节点了），同时也会触发其父节点的child watch。

7、当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

8、Watch是轻量级的，其实就是本地JVM的Callback，服务器端只是存了是否有设置了Watcher的布尔类型

推荐阅读

1. 如何轻松阅读 GitHub 上的项目源码？

2. IntelliJ IDEA新增禅模式和LightEdit模式

3. 安利一款 IDEA 中强大的代码生成利器

4. 如何获取靠谱的新型冠状病毒疫情

Zookeeper：分布式架构详解、分布式技术详解、分布式事务

高级互联网架构 Java后端 2019-10-26

作者 | Java高级互联网架构

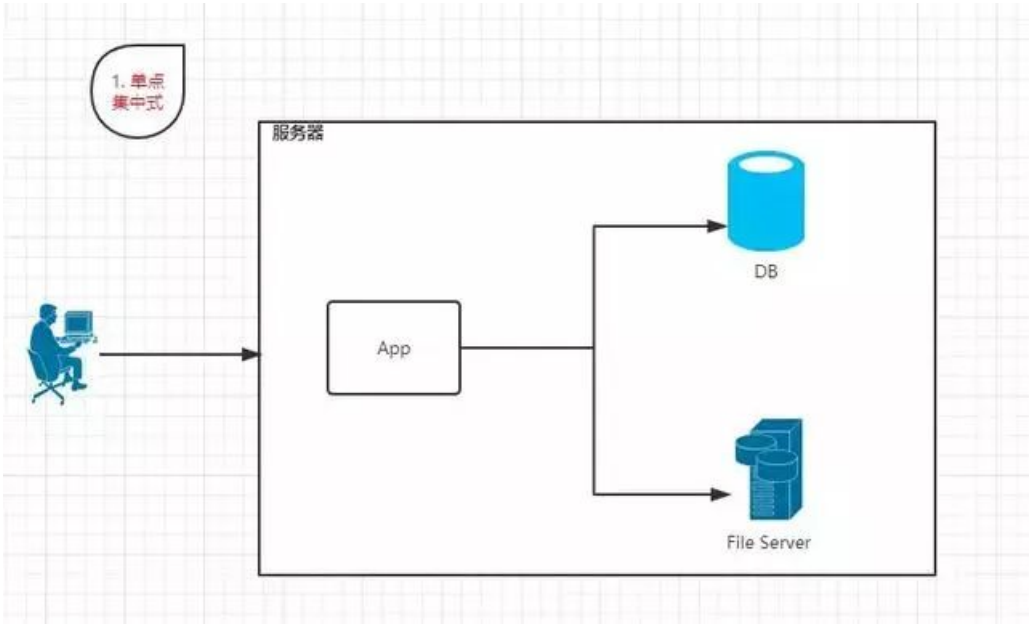
链接 | toutiao.com/a6742369092881089028/

一、分布式架构详解

1、分布式发展历程

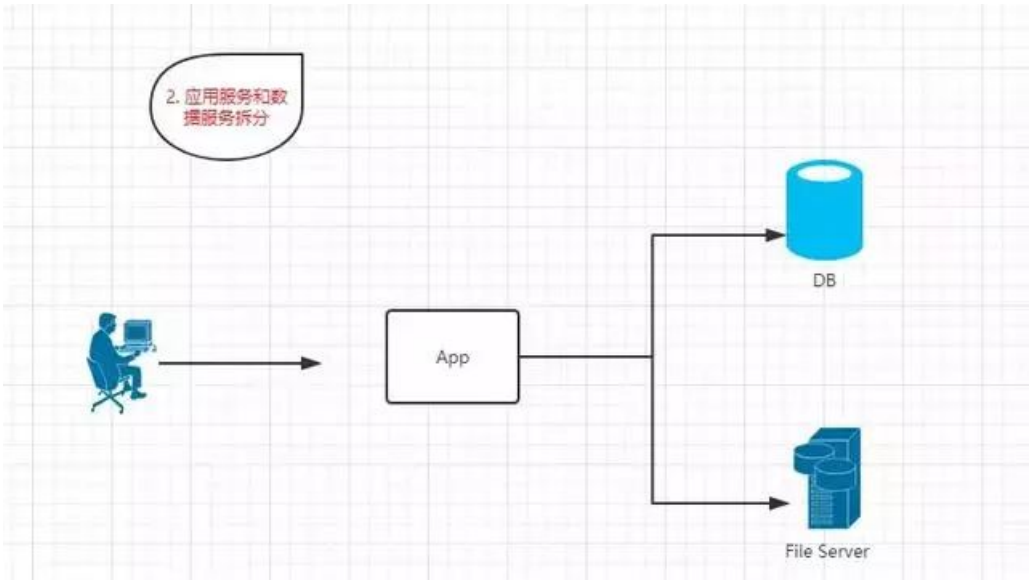
1.1 单点集中式

特点：App、DB、FileServer都部署在一台机器上。并且访问请求量较少



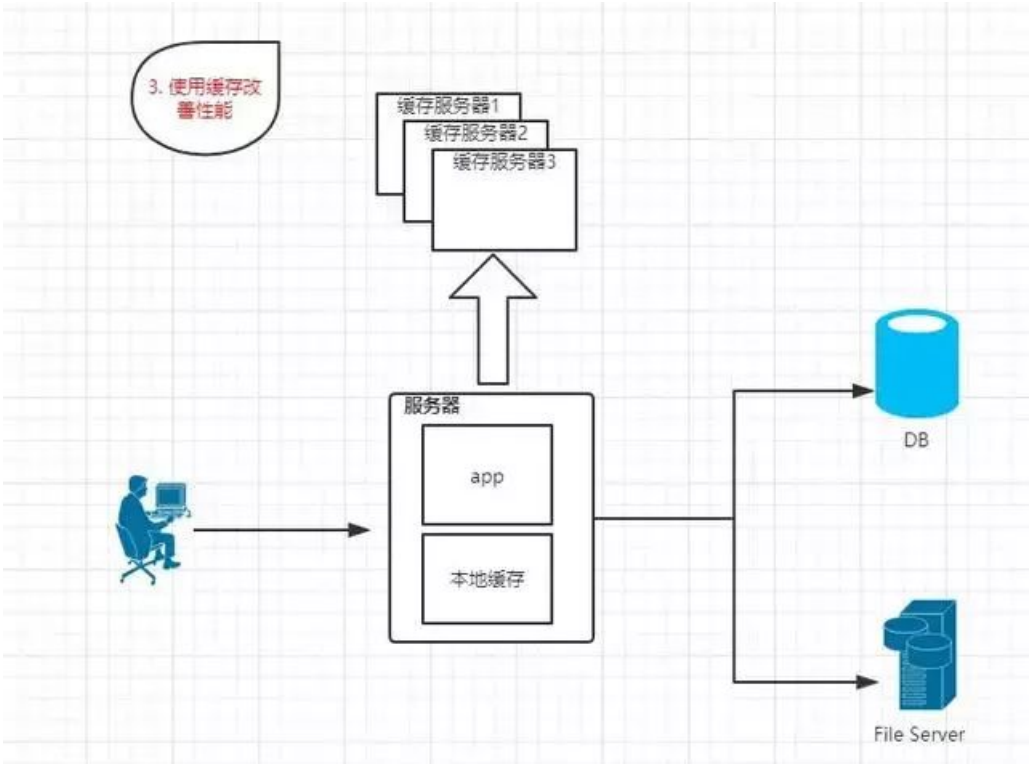
1.2 应用服务和数据服务拆分

特点：App、DB、FileServer分别部署在独立服务器上。并且访问请求量较少



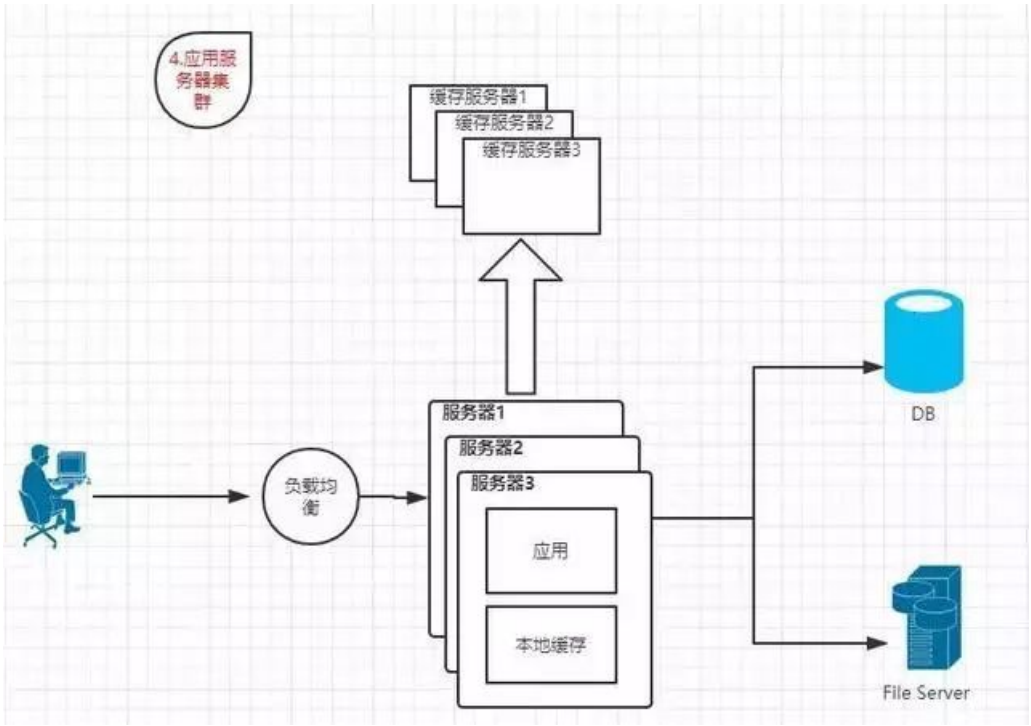
1.3 使用缓存改善性能

特点：数据库中频繁访问的数据存储在缓存服务器中，减少数据库的访问次数，降低数据库的压力



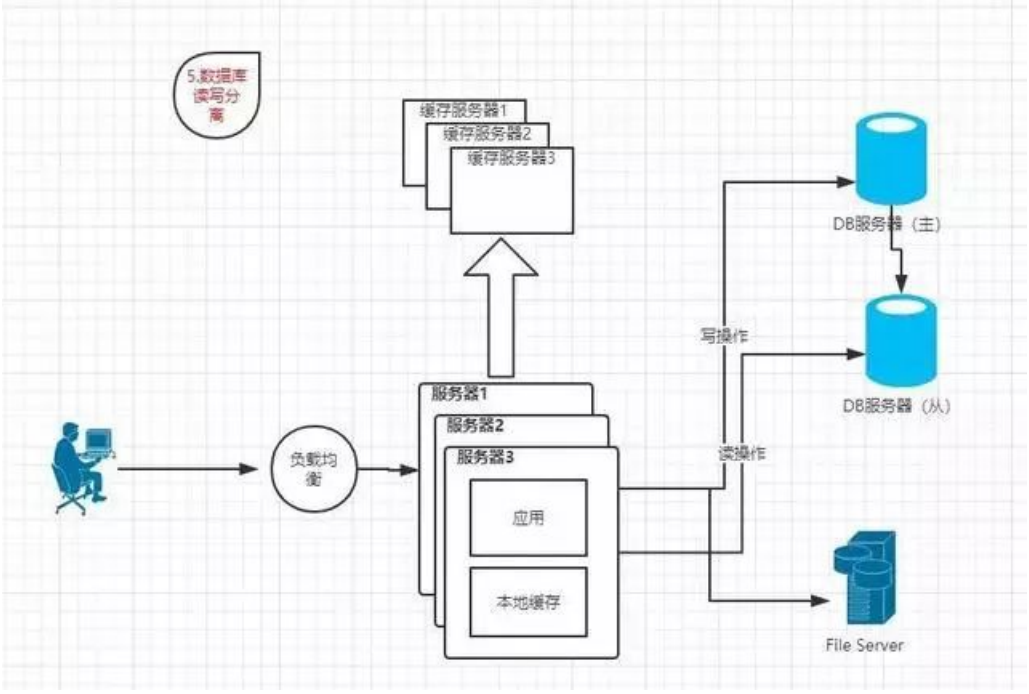
1.4 应用服务器集群

特点：多台应用服务器通过负载均衡同时对外提供服务，解决单台服务器处理能力上限的问题



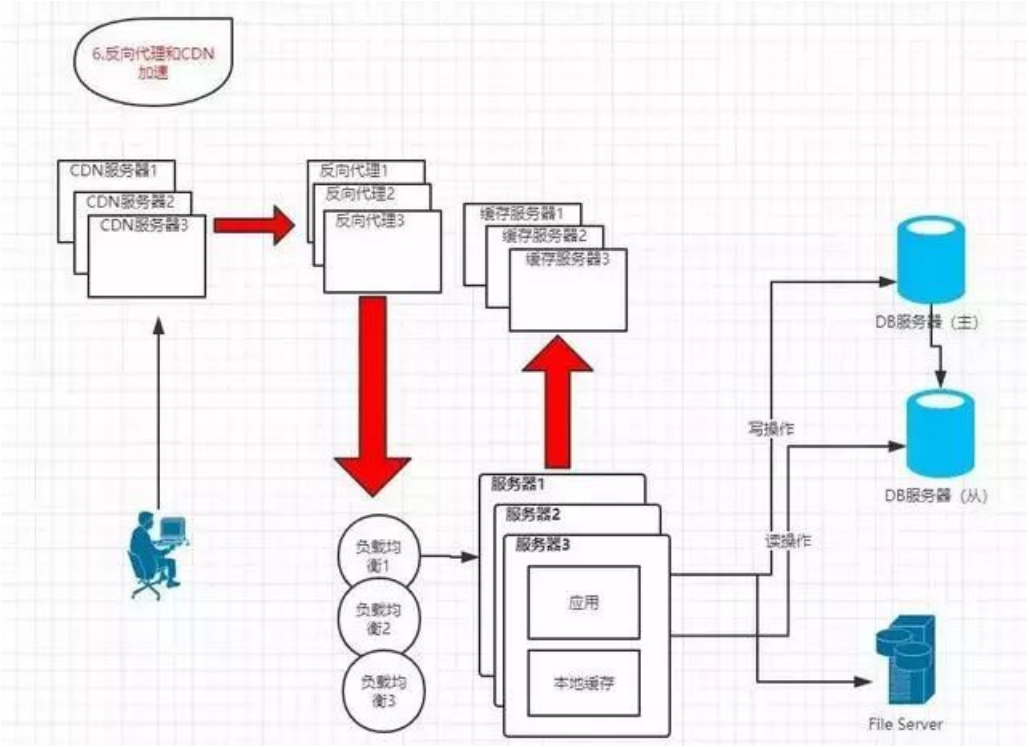
1.5 数据库读写分离

特点：数据库进行读写分离（主从）设计，解决数据库的处理压力



1.6 反向代理和CDN加速

特点：采用反向代理和CDN加快系统的访问速度

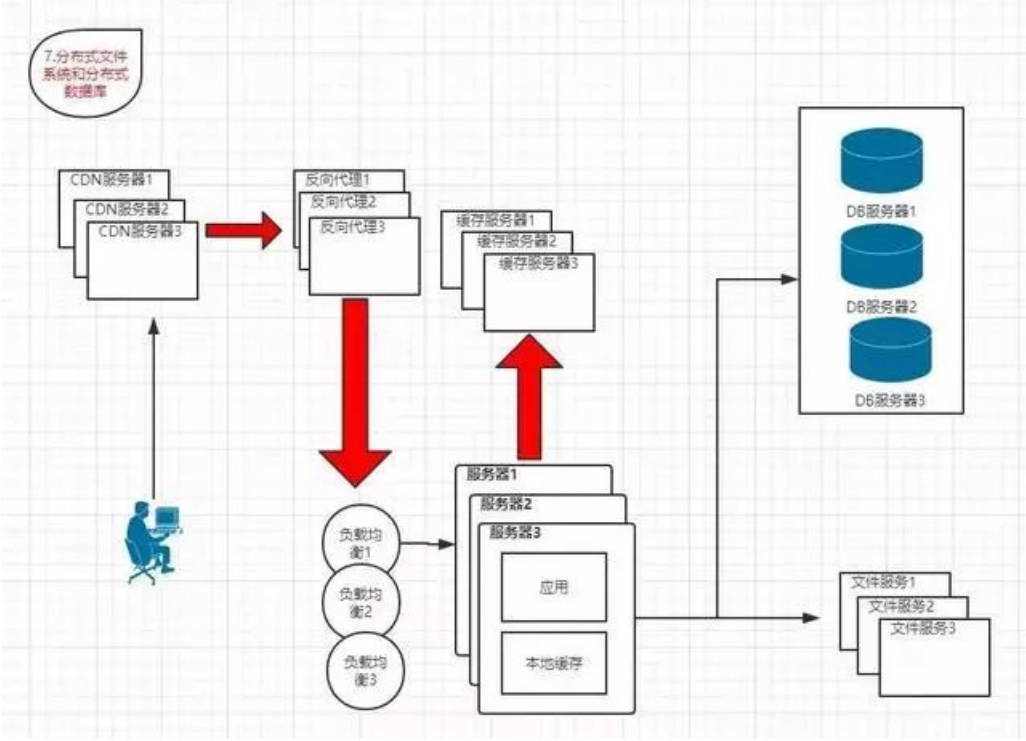


1.7 分布式文件系统和分布式数据库

特点：数据库采用分布式数据库，文件系统采用分布式文件系统

随着业务的发展，最终数据库读写分离也将无法满足需求，需要采用分布式数据库和分布式文件系统来支撑

分布式数据库是数据库拆分后的最后方法，只有在单表规模非常庞大的时候才使用，更常用的数据库拆分手段是业务分库，将不同业务的数据库部署在不同的机器上



二、 分布式技术详解

1. 并发性

2. 分布性

大任务拆分成多个任务部署到多台机器上对外提供服务

3. 缺乏全局时钟

时间要统一

4. 对等性

一个服务部署在多台机器上是一样的，无任何差别

5. 故障肯定会发生

硬盘坏了 CPU烧了....

三、分布式事务

1. ACID

原子性（Atomicity）：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性（Consistency）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。

比如A有500元，B有300元，A向B转账100，无论怎么样，A和B的总和总是800元

隔离性（Isolation）：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

持久性（Durability）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

2. 2P/3P

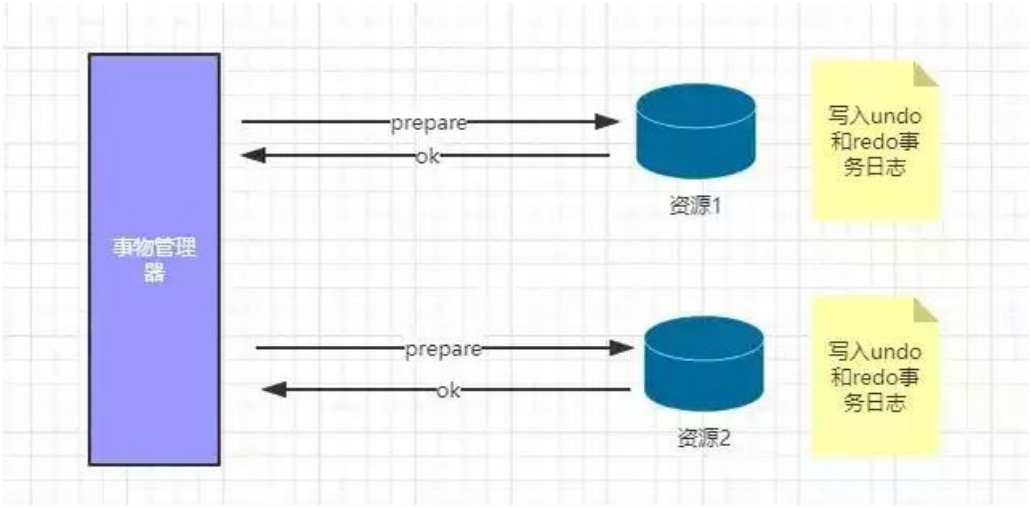
2P= Two Phase commit 二段提交（RDBMS（关系型数据库管理系统）经常就是这种机制，保证强一致性）

3P= Three Phase commit 三段提交

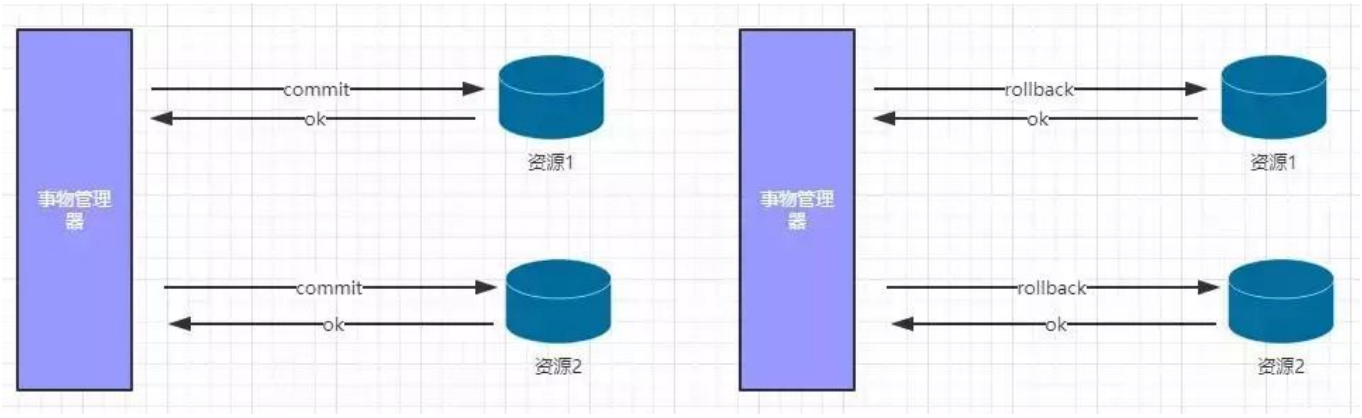
说明：2P/3P是为了保证事务的ACID（原子性、一致性、隔离性、持久性）

2.1 2P的两个阶段

阶段1：提交事务请求（投票阶段）询问是否可以提交事务



阶段2：执行事务提交（commit、rollback） 真正的提交事务

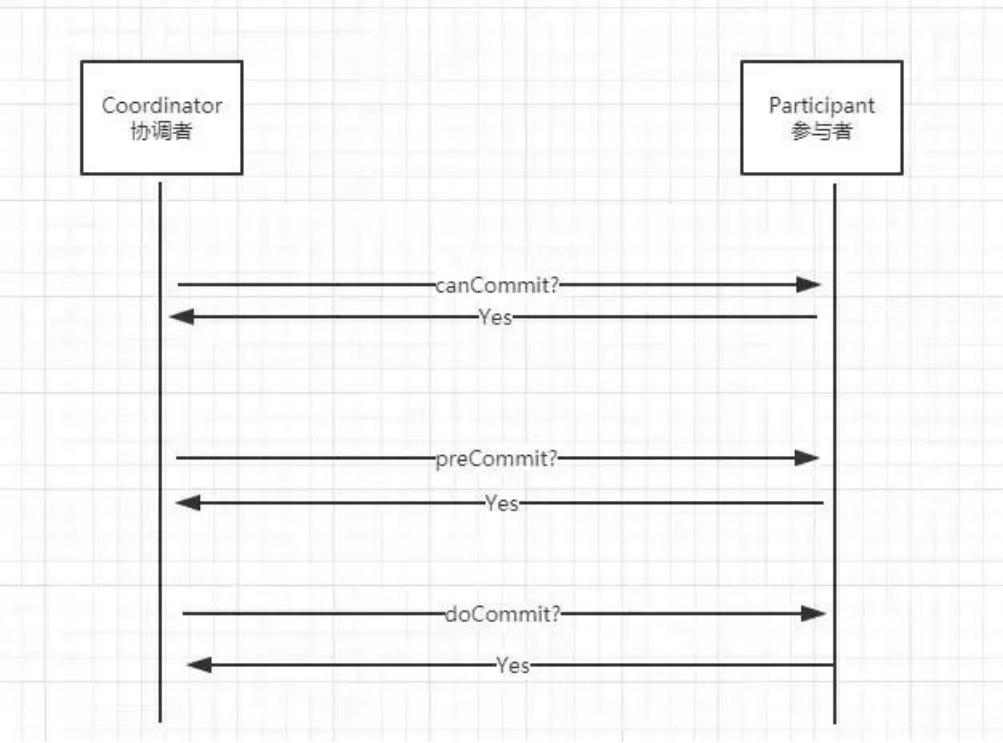


2.2 3P的三个阶段

阶段1：是否提交-询问是否可以做事务提交

阶段2：预先提交-预先提交事务

阶段3：执行事务提交（commit、rollback）真正的提交事务



说明：3P把2P的阶段一拆分成了前面两个阶段

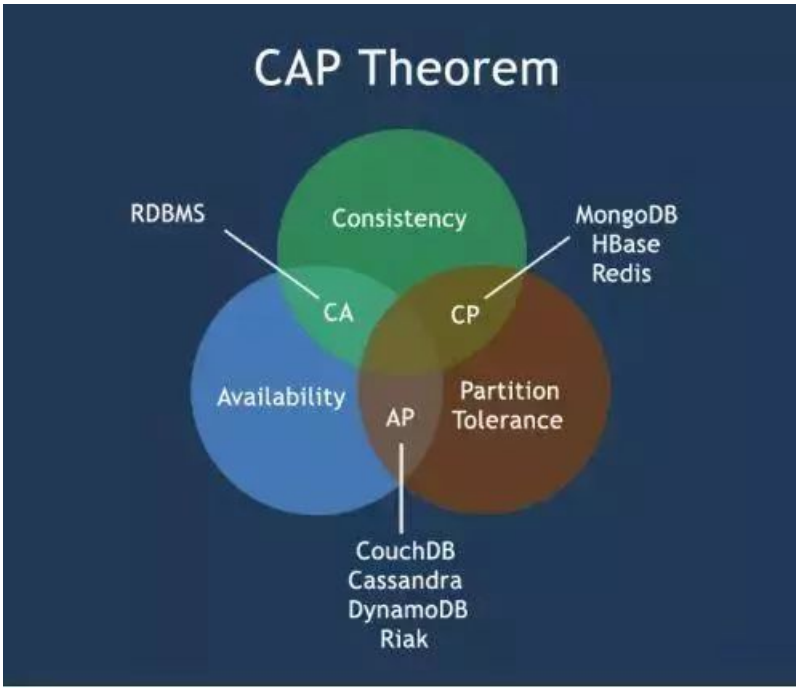
3. CAP理论

一致性（Consistency）：分布式数据库的数据保持一致

可用性（Availability）：任何一个节点挂了，其他节点可以继续对外提供服务

分区容错性（网络分区）Partition tolerance：一个数据库所在的机器坏了，如硬盘坏了，数据丢失了，可以新增一台机器，然后从其他正常的机器把备份的数据同步过来

CAP理论的特点：CAP只能满足其中2条



CA(放弃P)：将所有的数据放在一个节点。满足一致性、可用性。

AP(放弃C)：放弃强一致性，用最终一致性来保证。

CP(放弃A)：一旦系统遇见故障，受到影响的服务器需要等待一段时间，在恢复期间无法对外提供服务。

举例说明CAP理论：

有3台机器分别有3个数据库分别有两张表,数据都是一样的

Machine1-db1-tbl_person、tbl_order

Machine2-db2-tbl_person、tbl_order

Machine3-db3-tbl_person、tbl_order

1）当向machine1的db1的表tbl_person、tbl_order插入数数据时，同时要把插入的数据同步到machine2、machine3，这就是一致性

2）当其中的一台机器宕机了，可以继续对外提供服务，把宕机的机器重新启动起来可以继续服务，这就是可用性

3）当machine1的机器坏了，数据全部丢失了，不会有任何问题，因为machine2和machine3上还有数据，重新加一台机器machine4，把machine2和machine3其中一台机器的备份数据同步过来就可以了，这就是分区容错性

4. BASE理论

基本可用（bascially available）、软状态（soft state）、最终一致性（Eventually consistent）

基本可用：在分布式系统出现故障，允许损失部分可用性（服务降级、页面降级）

软状态：允许分布式系统出现中间状态。而且中间状态不影响系统的可用性。

1、这里的中间状态是指不同的data replication之间的数据更新可以出现延时的最终一致性

2、如CAP理论里面的示例，当向machine1的db1的表tbl_person、tbl_order插入数数据时，同时要把插入的数据同步到machine2、machine3，当machine3的网络有问题时，同步失败，但是过一会网络恢复了就同步成功了，这个同步失败的状态就称为软状态，因为最终还是同步成功了。

最终一致性：data replications经过一段时间达到一致性。

5. Paxos算法

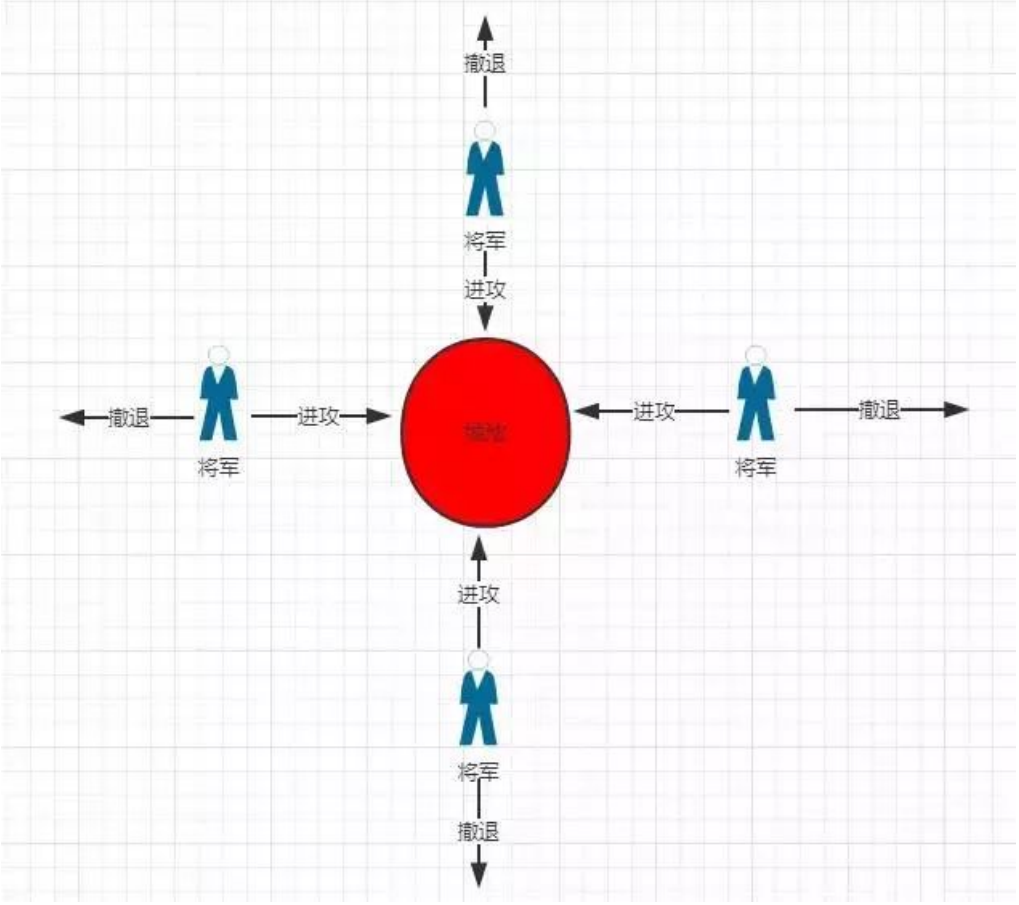
5.1 介绍Paxos算法之前我们先来看一个小故事

拜占庭将军问题

拜占庭帝国就是5~15世纪的东罗马帝国，拜占庭即现在土耳其的伊斯坦布尔。我们可以想象，拜占庭军队有许多分支，驻扎在敌人城外，每一分支由各自的将军指挥。假设有11位将军，将军们只能靠通讯员进行通讯。在观察敌人以后，忠诚的将军们必须制订一个统一的行动计划——进攻或者撤退。然而，这些将军里有叛徒，他们不希望忠诚的将军们能达成一致，因而影响统一行动计划的制订与传播。

问题是：将军们必须有一个协议，使所有忠诚的将军们能够达成一致，而且少数几个叛徒不能使忠诚的将军们作出错误的计划——使有些将军进攻而另一些将军撤退。

假设有9位忠诚的将军，5位判断进攻，4位判断撤退，还有2个间谍恶意判断撤退，虽然结果是错误的撤退，但这种情况完全是允许的。因为这11位将军依然保持着状态一致性。



总结：

1）11位将军进攻城池

2）同时进攻（议案、决议）、同时撤退（议案、决议）

3）不管撤退还是进攻，必须半数的将军统一意见才可以执行

4）将军里面有叛徒，会干扰决议生成

5.2 下面就来介绍一下Paxos算法

Google Chubby的作者Mike Burrows说过这个世界上只有一种一致性算法，那就是Paxos，其它的算法都是残次品。

Paxos：多数派决议（最终解决一致性问题）

Paxos算法有三种角色：Proposer，Acceptor，Learner

Proposer：提交者（议案提交者）

提交议案(判断是否过半)，提交批准议案(判断是否过半)

Acceptor：接收者（议案接收者）

接受议案或者驳回议案，给proposer回应(promise)

Learner：学习者（打酱油的）

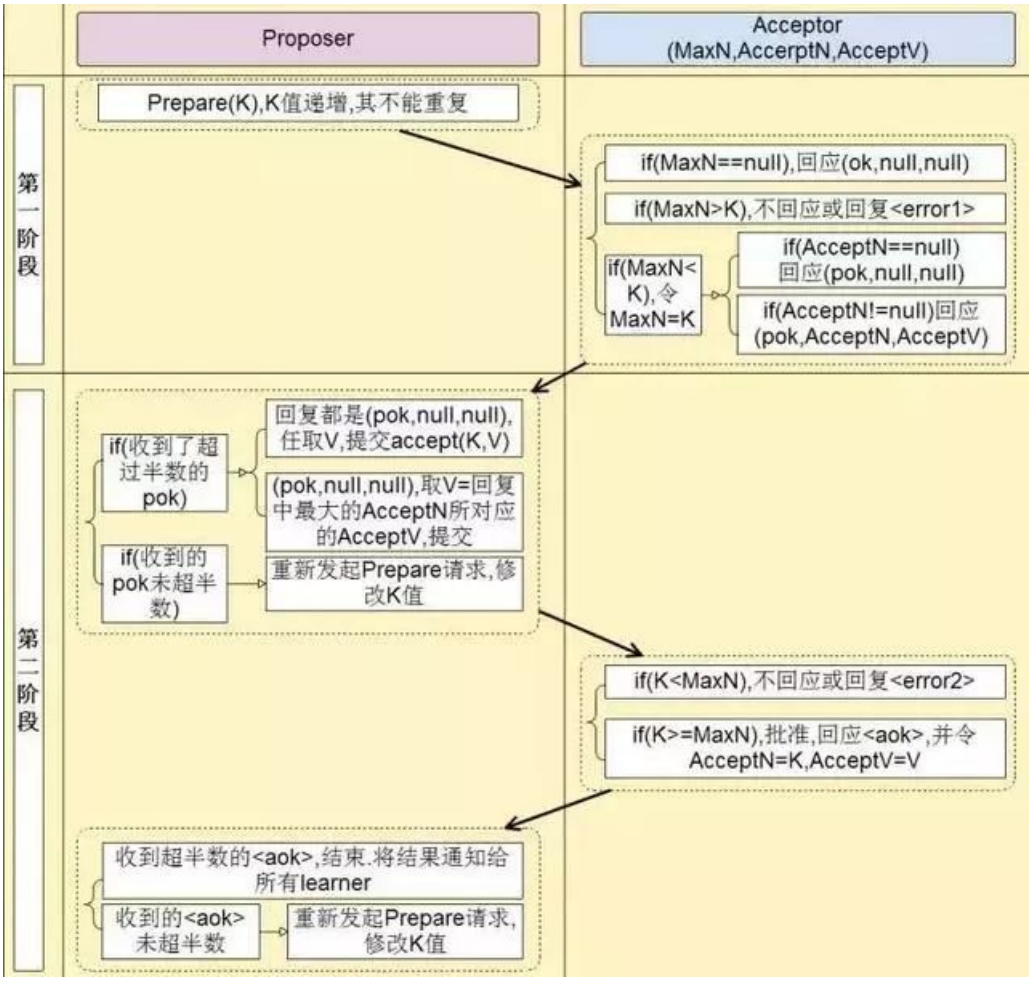
如果议案产生，学习议案。

设定1：如果Acceptor没有接受议案，那么他必须接受第一个议案

设定2：每个议案必须有一个编号，并且编号只能增长，不能重复。越往后越大。

设定3：接受编号大的议案，如果小于之前接受议案编号，那么不接受

设定4：议案有2种(提交的议案，批准的议案)



1）Prepare阶段（议案提交）

a）Proposer希望议案V。首先发出Prepare请求至大多数Acceptor。Prepare请求内容为序列号K

b）Acceptor收到Prepare请求为编号K后，检查自己手里是否有处理过Prepare请求。

c）如果Acceptor没有接受过任何Prepare请求，那么用OK来回复Proposer，代表Acceptor必须接受收到的第一个议案（设定1）

d）否则，如果Acceptor之前接受过任何Prepare请求（如：MaxN），那么比较议案编号，如果K<MaxN，则用reject或者error回复Proposer

e）如果K>=MaxN，那么检查之前是否有批准的议案，如果没有则用OK来回复Proposer，并记录K

f）如果K>=MaxN，那么检查之前是否有批准的议案，如果有则回复批准的议案编号和议案内容（如：<AcceptN, AcceptV>，AcceptN为批准的议案编号，AcceptV为批准的议案内容）

2）Accept阶段（批准阶段）

a）Proposer收到过半Acceptor发来的回复，回复都是OK，且没有附带任何批准过的议案编号和议案内容。那么Proposer继续提交批准请求，不过此时会连议案编号K和议案内容V一起提交（<K, V>这种数据形式）

b）Proposer收到过半Acceptor发来的回复，回复都是OK，且附带批准过的议案编号和议案内容（<pok，议案编号，议案内容>）。那么Proposer找到所有回复中超过半数的那个（假设为<pok，AcceptNx，AcceptVx>）作为提交批准请求（请求为

<K，AcceptVx>）发送给Acceptor。

c）Proposer没有收到过半Acceptor发来的回复，则修改议案编号K为K+1，并将编号重新发送给Acceptors（重复Prepare阶段的过程）

d）Acceptor收到Proposer发来的Accept请求，如果编号K<MaxN则不回应或者reject。

e）Acceptor收到Proposer发来的Accept请求，如果编号K>=MaxN则批准该议案，并设置手里批准的议案为<K，接受议案的编号，接受议案的内容>，回复Proposer。

f）经过一段时间Proposer对比手里收到的Accept回复，如果超过半数，则结束流程（代表议案被批准），同时通知Leaner可以学习议案。

g) 经过一段时间Proposer对比手里收到的Accept回复，如果未超过半数，则修改议案编号重新进入Prepare阶段。

tips: 欢迎关注微信公众号：Java后端，获取更多推送。

5.3 Paxos示例

示例1：先后提议的场景

角色：

proposer：参谋1，参谋2

acceptor：将军1，将军2，将军3（决策者）

1）参谋1发起提议，派通信兵带信给3个将军，内容为（编号1）；

2）3个将军收到参谋1的提议，由于之前还没有保存任何编号，因此把（编号1）保存下来，避免遗忘；同时让通信兵带信回去，内容为（ok）；

3）参谋1收到至少2个将军的回复，再次派通信兵带信给3个将军，内容为（编号1，进攻时间1）；

4）3个将军收到参谋1的时间，把（编号1，进攻时间1）保存下来，避免遗忘；同时让通信兵带信回去，内容为（Accepted）；

5）参谋1收到至少2个将军的（Accepted）内容，确认进攻时间已经被大家接收；

6）参谋2发起提议，派通信兵带信给3个将军，内容为（编号2）；

7）3个将军收到参谋2的提议，由于（编号2）比（编号1）大，因此把（编号2）保存下来，避免遗忘；又由于之前已经接受参谋1的提议，因此让通信兵带信回去，内容为（编号1，进攻时间1）；

8）参谋2收到至少2个将军的回复，由于回复中带来了已接受的参谋1的提议内容，参谋2因此不再提出新的进攻时间，接受参谋1提出的时间；

示例2：交叉场景

角色：

proposer：参谋1，参谋2

acceptor：将军1，将军2，将军3（决策者）

1）参谋1发起提议，派通信兵带信给3个将军，内容为（编号1）；

2）3个将军的情况如下

a）将军1和将军2收到参谋1的提议，将军1和将军2把（编号1）记录下来，如果有其他参谋提出更小的编号，将被拒绝；同时让通信兵带信回去，内容为（ok）；

b）负责通知将军3的通信兵被抓，因此将军3没收到参谋1的提议；

3）参谋2在同一时间也发起了提议，派通信兵带信给3个将军，内容为（编号2）；

4）3个将军的情况如下

a）将军2和将军3收到参谋2的提议，将军2和将军3把（编号2）记录下来，如果有其他参谋提出更小的编号，将被拒绝；同时让通信兵带信回去，内容为（ok）；

b）负责通知将军1的通信兵被抓，因此将军1没收到参谋2的提议；

5）参谋1收到至少2个将军的回复，再次派通信兵带信给有答复的2个将军，内容为（编号1，进攻时间1）；

6）2个将军的情况如下

a）将军1收到了（编号1，进攻时间1），和自己保存的编号相同，因此把（编号1，进攻时间1）保存下来；同时让通信兵带信回去，内容为（Accepted）；

b）将军2收到了（编号1，进攻时间1），由于（编号1）小于已经保存的（编号2），因此让通信兵带信回去，内容为（Rejected，编号2）；

7）参谋2收到至少2个将军的回复，再次派通信兵带信给有答复的2个将军，内容为（编号2，进攻时间2）；

8）将军2和将军3收到了（编号2，进攻时间2），和自己保存的编号相同，因此把（编号2，进攻时间2）保存下来，同时让通信兵带信回去，内容为（Accepted）；

9）参谋2收到至少2个将军的（Accepted）内容，确认进攻时间已经被多数派接受；

10）参谋1只收到了1个将军的（Accepted）内容，同时收到一个（Rejected，编号2）；参谋1重新发起提议，派通信兵带信给3个将军，内容为（编号3）；

11）3个将军的情况如下

a）将军1收到参谋1的提议，由于（编号3）大于之前保存的（编号1），因此把（编号3）保存下来；由于将军1已经接受参谋1前一次的提议，因此让通信兵带信回去，内容为（编号1，进攻时间1）；

b）将军2收到参谋1的提议，由于（编号3）大于之前保存的（编号2），因此把（编号3）保存下来；由于将军2已经接受参谋2的提议，因此让通信兵带信回去，内容为（编号2，进攻时间2）；

c）负责通知将军3的通信兵被抓，因此将军3没收到参谋1的提议；

12）参谋1收到了至少2个将军的回复，比较两个回复的编号大小，选择大编号对应的进攻时间作为最新的提议；参谋1再次派通信兵带信给有答复的2个将军，内容为（编号3，进攻时间2）；

13）将军1和将军2收到了（编号3，进攻时间2），和自己保存的编号相同，因此保存（编号3，进攻时间2），同时让通信兵带信回去，内容为（Accepted）；

14）参谋1收到了至少2个将军的（accepted）内容，确认进攻时间已经被多数派接受。

四. Zookeeper ZAB协议

Zookeeper Automic Broadcast(ZAB)，即Zookeeper原子性广播，是Paxos经典实现

术语：

quorum：集群过半数的集合

1. ZAB(zookeeper)中节点分四种状态

looking：选举Leader的状态（崩溃恢复状态下）

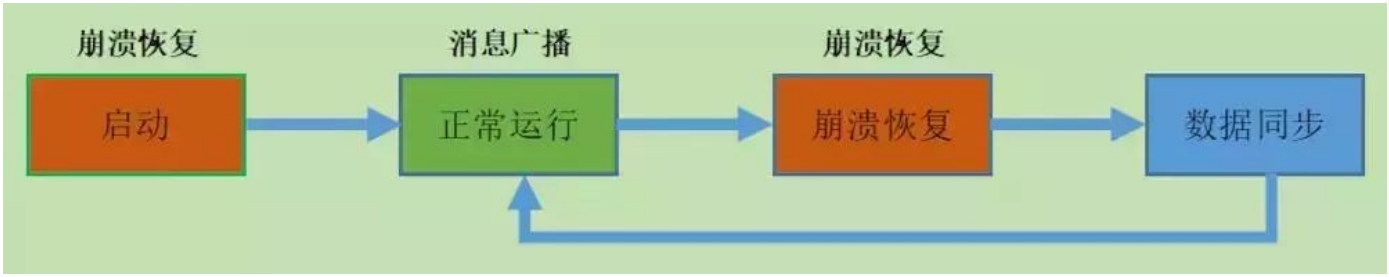
following：跟随者（follower）的状态，服从Leader命令

leading：当前节点是Leader，负责协调工作。

observing：observer(观察者)，不参与选举，只读节点。

2. ZAB中的两个模式（ZK是如何进行选举的）

崩溃恢复、消息广播



1）崩溃恢复

leader挂了，需要选举新的leader

a.每个server都有一张选票，如（3,9），选票投自己。

b.每个server投完自己后，再分别投给其他还可用的服务器。如把Server3的（3,9）分别投给Server4和Server5，一次类推

c.比较投票，比较逻辑：优先比较Zxid，Zxid相同时才比较myid。比较Zxid时，大的做leader；比较myid时，小的做leader

d.改变服务器状态（崩溃恢复->数据同步，或者崩溃恢复->消息广播）

相关概念补充说明：

epoch周期值

acceptedEpoch（比喻：年号）：follower已经接受leader更改年号的（newepoch）提议。

currentEpoch（比喻：当前的年号）：当前的年号

lastZxid：history中最近接收到的提议zxid(最大的值)

history：当前节点接受到事务提议的log

Zxid数据结构说明：

cZxid = 0x10000001b

64位的数据结构

高32位：10000

Leader的周期编号+myid的组合

低32位：001b

事务的自增序列（单调递增的序列）只要客户端有请求，就+1

当产生新Leader的时候，就从这个Leader服务器上取出本地log中最大事务Zxid，从里面读出epoch+1，作为一个新epoch，并将低32位置0（保证id绝对自增）

2）消息广播（类似2P提交）

a.Leader接受请求后，将这个请求赋予全局的唯一64位自增Id（zxid）。

b.将zxid作为议案发给所有follower。

c.所有的follower接受到议案后，想将议案写入硬盘后，马上回复Leader一个ACK（OK）。

d.当Leader接受到合法数量（过半）Acks，Leader给所有follower发送commit命令。

e.follower执行commit命令。

注意：到了这个阶段，ZK集群才正式对外提供服务，并且Leader可以进行消息广播，如果有新节点加入，还需要进行同步。

3）数据同步

a.取出Leader最大lastZxid（从本地log日志来）

b.找到对应zxid的数据，进行同步（数据同步过程保证所有follower一致）

c.只有满足quorum同步完成，准Leader才能成为真正的Leader

推荐阅读

1. Java 开发中如何正确的踩坑
2. 当我遵循了这 16 条规范写代码
3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践

学Java，请关注公众号：Java后端

喜欢文章，点个在看

阅读原文

「附源码」Dubbo+Zookeeper 的 RPC 远程调用框架

码农云帆哥 Java后端 2019-10-11

作者 | 码农云帆哥

链接 | blog.csdn.net/sinat_27933301

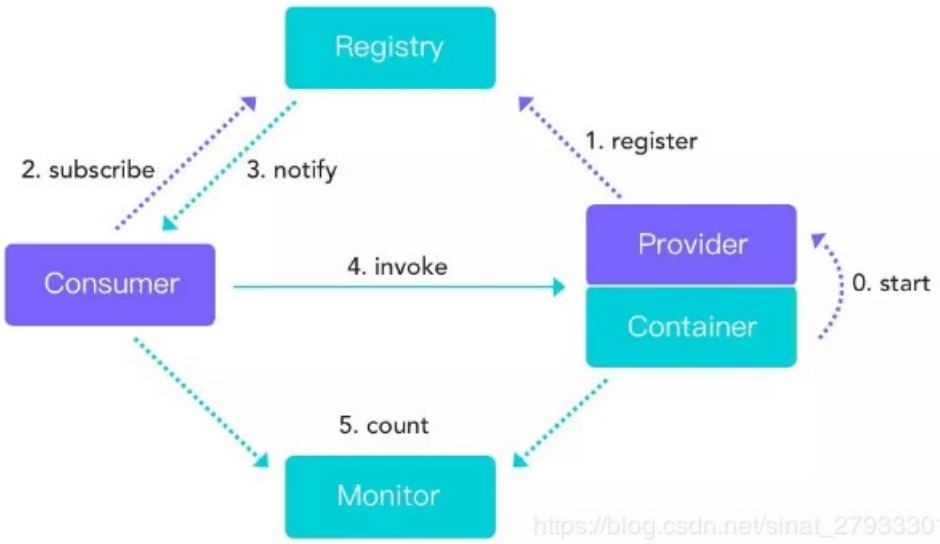
上一篇:从零搭建创业公司后台技术栈

这是一个基于Dubbo+Zookeeper 的 RPC 远程调用框架 demo，希望读者可以通过这篇文章大概能看懂这一个简单的框架搭建。

Demo 源码获取方式：关注微信公众号「Java后端」，回复「DZ」获取

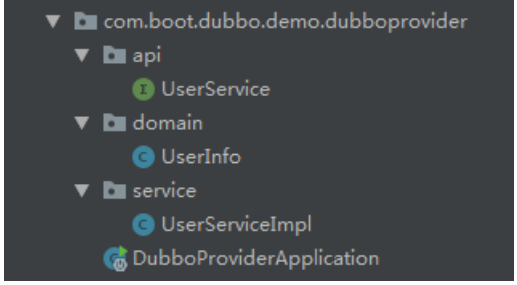
Dubbo是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和Spring框架无缝集成。

Dubbo是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。微信搜索 web_resource 关注获取更多推送



- provider：暴露服务的服务提供方
- consumer：调用远程服务的服务消费方
- registry：服务注册于发现的注册中心
- monitor：统计服务调用次数和调用时间的监控中心
- container：服务运行容器

一、dubbo-provider（服务提供方）



1、Java Bean

为什么要实现Serializable接口？当我们需要把对象的状态信息通过网络进行传输，或者需要将对象的状态信息持久化，以便将来使用时都需要把对象进行序列化。

使用了Lombok，它能通过注解的方式，在编译时自动为属性生成构造器、getter/setter、equals、hashcode、toString方法。

```
@Data
public class UserInfo implements Serializable {
    private String account;
    private String password;
}
```

2、UserService

服务提供方　暴露的服务，将注册到zookeeper上。

```
public interface UserService {
    // 定义用户登录的api
    UserInfo login(UserInfo user)
;
}
```

3、UserServiceImpl

服务提供方暴露的服务对应的实现类。

```
@Component
@Service(interfaceClass = UserService.class)
public class UserServiceImpl implements UserService {
    public UserInfo login(UserInfo user)
{
        UserInfo reUser = new UserInfo()
;
        reUser.setAccount("登录的账号为:"+user.getAccount());
        reUser.setPassword("登录的密码为:"+user.getPassword());

        return reUser;
    }
}
```

4、DubboProviderApplication

```
@SpringBootApplication
@EnableDubboConfiguration // 启用dubbo自动配置
public class DubboProviderApplication {

    public static void main(String[] args)
{
        SpringApplication.run(DubboProviderApplication.class, args);
    }
}
```

5、pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
>
    <modelVersion>4.0.0</modelVersion
>

    <groupId>com.boot.dubbo.demo</groupId
>
    <artifactId>dubbo-provider</artifactId
>
    <version>0.0.1-SNAPSHOT</version
>
    <packaging>jar</packaging
>

    <name>dubbo-provider</name
>
    <description>Demo project for Spring Boot</description
>

    <parent
>
        <groupId>org.springframework.boot</groupId
>
        <artifactId>spring-boot-starter-parent</artifactId
>
        <version>2.0.6.RELEASE</version
>
        <relativePath/> <!-- lookup parent from repository -->
    </parent
>

    <properties
>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
```

```
        <java.version>1.8</java.version
    >
    </properties
    >

    <dependencies
    >
        <dependency
        >
            <groupId>org.springframework.boot</groupId
            >
            <artifactId>spring-boot-starter-web</artifactId
            >
        </dependency
        >

        <!--Dubbo依赖-->
        <dependency
        >
            <groupId>com.alibaba.spring.boot</groupId
            >
            <artifactId>dubbo-spring-boot-starter</artifactId
            >
            <version>2.0.0</version
            >
        </dependency
        >

        <!--自动生成getter，setter，equals，hashCode和toString等等-->
        <dependency
        >
            <groupId>org.projectlombok</groupId
            >
            <artifactId>lombok</artifactId
            >
            <version>1.16.20</version
            >
            <scope>provided</scope
            >
        </dependency
        >

        <!--Zookeeper客户端-->
        <dependency
        >
            <groupId>com.101tec</groupId
            >
            <artifactId>zkclient</artifactId
            >
            <version>0.10</version
            >
        </dependency
        >
```

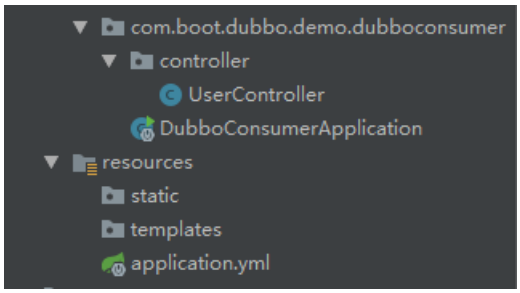
```
        <!--Zookeeper依赖，排除log4j避免依赖冲突-->
        <dependency
>
            <groupId>org.apache.zookeeper</groupId
>
            <artifactId>zookeeper</artifactId
>
            <version>3.4.10</version
>
            <exclusions
>
                <exclusion
>
                    <groupId>org.slf4j</groupId
>
                    <artifactId>slf4j-log4j12</artifactId
>
                </exclusion
>
                <exclusion
>
                    <groupId>log4j</groupId
>
                    <artifactId>log4j</artifactId
>
                </exclusion
>
            </exclusions
>
        </dependency
>

        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-test</artifactId
>
            <scope>test</scope
>
        </dependency
>
    </dependencies
>

    <build
>
        <plugins
>
            <plugin
>
                <groupId>org.springframework.boot</groupId
>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId
>
            </plugin
>
        </plugins
>
    </build
>
</project>
```

6、application.yml

```
spring:
  dubbo
:
    application
:
      name: dubbo-provider
    protocol
:
      name: dubb
o
      port: 2088
0
    registry
:
      address: zookeeper://127.0.0.1:2181
```

二、dubbo-consumer（服务消费方）



1、UserController

```
@RestController
public class UserController {

    @Reference // 引用dubbo服务器提供服务器接口
    private UserService userService;

    @GetMapping("/login"
)
    public UserInfo login(UserInfo userInfo) {
        return userService.login(userInfo);
    }
}
```

2、DubboConsumerApplication

```
@SpringBootApplication
@EnableDubboConfiguration // 启用dubbo自动配置
public class DubboConsumerApplication {

    public static void main(String[] args)
{
        SpringApplication.run(DubboConsumerApplication.class, args);
    }
}
```

3、pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
>
    <modelVersion>4.0.0</modelVersion
>

    <groupId>com.boot.dubbo.demo</groupId
>
    <artifactId>dubbo-consumer</artifactId
>
    <version>0.0.1-SNAPSHOT</version
>
    <packaging>jar</packaging
>

    <name>dubbo-consumer</name
>
    <description>Demo project for Spring Boot</description
>

    <parent
>
        <groupId>org.springframework.boot</groupId
>
        <artifactId>spring-boot-starter-parent</artifactId
>
        <version>2.0.6.RELEASE</version
>
        <relativePath/> <!-- lookup parent from repository -->
    </parent
>

    <properties
>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
```

```
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
>
        <java.version>1.8</java.version
>
    </properties
>

    <dependencies
>
        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-web</artifactId
>
        </dependency
>

        <!--依赖dubbo-provider服务提供者-->
        <dependency
>
            <groupId>com.boot.dubbo.demo</groupId
>
            <artifactId>dubbo-provider</artifactId
>
            <version>0.0.1-SNAPSHOT</version
>
        </dependency
>

        <!--Dubbo依赖-->
        <dependency
>
            <groupId>com.alibaba.spring.boot</groupId
>
            <artifactId>dubbo-spring-boot-starter</artifactId
>
            <version>2.0.0</version
>
        </dependency
>

        <!--Zookeeper客户端-->
        <dependency
>
            <groupId>com.101tec</groupId
>
            <artifactId>zkclient</artifactId
>
            <version>0.10</version
>
        </dependency
>

        <!--Zookeeper依赖，排除log4j避免依赖冲突-->
```

```
        <!--zookeeper依赖，排除log4j避免依赖冲突-->
        <dependency
>
            <groupId>org.apache.zookeeper</groupId
>
            <artifactId>zookeeper</artifactId
>
            <version>3.4.10</version
>
            <exclusions
>
                <exclusion
>
                    <groupId>org.slf4j</groupId
>
                    <artifactId>slf4j-log4j12</artifactId
>
                </exclusion
>
                <exclusion
>
                    <groupId>log4j</groupId
>
                    <artifactId>log4j</artifactId
>
                </exclusion
>
            </exclusions
>
        </dependency
>

        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-test</artifactId
>
            <scope>test</scope
>
        </dependency
>
    </dependencies
>

    <build
>
        <plugins
>
            <plugin
>
                <groupId>org.springframework.boot</groupId
>
                <artifactId>spring-boot-maven-plugin</artifactId
```

```
>
            </plugin
>
        </plugins
>
    </build
>
</project>
```

4、application.yml

```
spring:
  dubbo
:
    application
:
      name: dubbo-consumer
    protocol
:
      name: dubb
o
      port: 2088
0
    registry
:
      address: zookeeper://127.0.0.1:2181
server:
  port: 8081
```

三、Zookeeper安装配置

Zookeeper是什么:

https://blog.csdn.net/sinat_27933301/article/details/80101970

Zookeeper的安装配置:

https://blog.csdn.net/sinat_27933301/article/details/84001530

Zookeeper集群环境搭建:

https://blog.csdn.net/sinat_27933301/article/details/84351404

四、项目启动

五、查看Zookeeper服务注册情况

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[dubbo, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /dubbo
[com.boot.dubbo.demo.api.UserService]
[zk: localhost:2181(CONNECTED) 2]
```

六、通过浏览器发送请求（dubbo-provider的服务注册到Zookeeper上，dubbo-consumer消费者可以调用注册的服务）。

到此，rpc远程调用服务通了！感兴趣的话可以去学习下。

Demo 源码获取方式：关注微信公众号「Java后端」，回复「DZ」获取

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓扫描二维码进群↓

推荐阅读

1. 从零搭建创业公司后台技术栈
2. 如何阅读 Java 源码？
3. 某小公司RESTful、前后端分离的实践
4. 该如何弥补 GitHub 功能缺陷？
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

基于 Zookeeper 的分布式锁实现

田小波 Java后端 2019-12-11

作者 | 田小波

链接 | http://www.tianxiaobo.com

1. 背景

最近在学习 Zookeeper，在刚开始接触 Zookeeper 的时候，完全不知道 Zookeeper 有什么用。且很多资料都是将 Zookeeper 描述成一个“类 Unix/Linux 文件系统”的中间件，导致我很难将类 Unix/Linux 文件系统的 Zookeeper 和分布式应用联系在一起。后来在粗读了《ZooKeeper 分布式过程协同技术详解》和《从Paxos到Zookeeper 分布式一致性原理与实践》两本书，并动手写了一些 CURD demo 后，初步对 Zookeeper 有了一定的了解。不过比较肤浅，为了进一步加深对 Zookeeper 的认识，我利用空闲时间编写了本篇文章对应的 demo – 基于 Zookeeper 的分布式锁实现。通过编写这个分布式锁 demo，使我对 Zookeeper 的 watcher 机制、Zookeeper 的用途等有了更进一步的认识。不过我所编写的分布式锁还是比较简陋的，实现的也不够优美，仅仅是个练习，仅供参考使用。好了，题外话就说到这里，接下来我们就来聊聊基于 Zookeeper 的分布式锁实现。

2. 独占锁和读写锁的实现

在本章，我将分别说明独占锁和读写锁详细的实现过程，并配以相应的流程图帮助大家了解实现的过程。这里先说说独占锁的实现。

2.1 独占锁的实现

独占锁又称排它锁，从字面意思上很容易理解他们的用途。即如果某个操作 O_1 对访问资源 R_1 的过程加锁，在操作 O_1 结束对资源 R_1 访问前，其他操作不允许访问资源 R_1。以上算是对独占锁的简单定义了，那么这段定义在 Zookeeper 的“类 Unix/Linux 文件系统”的结构中是怎样实现的呢？在锁答案前，我们先看张图：

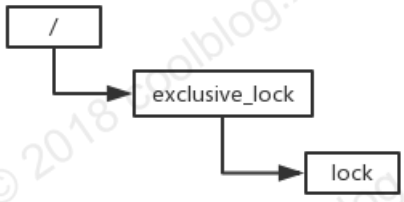


图1 独占锁的 Zookeeper 节点结构

如上图，对于独占锁，我们可以将资源 R_1 看做是 lock 节点，操作 O_1 访问资源 R_1 看做创建 lock 节点，释放资源 R_1 看做删除 lock 节点。这样我们就将独占锁的定义对应于具体的 Zookeeper 节点结构，通过创建 lock 节点获取锁，删除节点释放锁。详细的过程如下：

1. 多个客户端竞争创建 lock 临时节点
2. 其中某个客户端成功创建 lock 节点，其他客户端对 lock 节点设置 watcher
3. 持有锁的客户端删除 lock 节点或该客户端崩溃，由 Zookeeper 删除 lock 节点
4. 其他客户端获得 lock 节点被删除的通知

5. 重复上述4个步骤，直至无客户端在等待获取锁了

上面即独占锁具体的实现步骤，理解起来并不复杂，这里不再赘述。

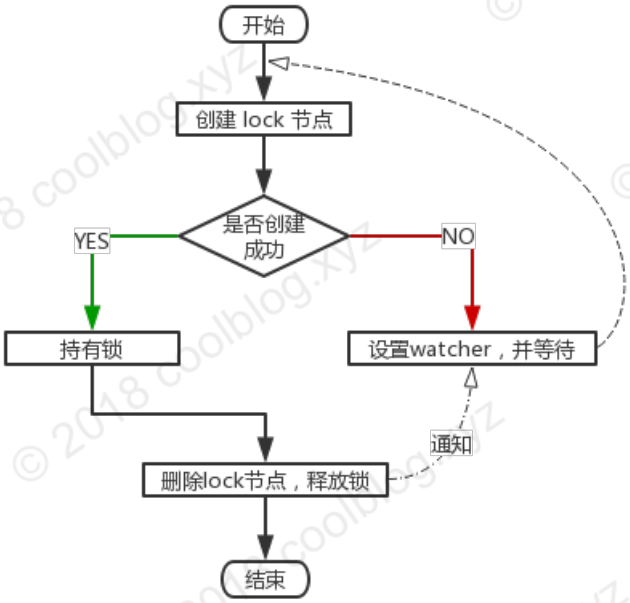


图2 获取独占锁流程图

2.2 读写锁的实现

说完独占锁的实现，这节来说说读写锁的实现。读写锁包含一个读锁和写锁，操作 O_1 对资源 R_1 加读锁，且获得了锁，其他操作可同时对资源 R_1 设置读锁，进行共享读操作。如果操作 O_1 对资源 R_1 加写锁，且获得了锁，其他操作再对资源 R_1 设置不同类型的锁都会被阻塞。总结来说，读锁具有共享性，而写锁具有排他性。那么在 Zookeeper 中，我们可以用怎样的节点结构实现上面的操作呢？

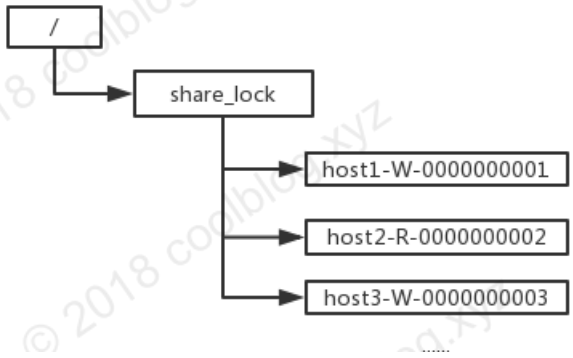


图3 读写锁的 Zookeeper 节点结构

在 Zookeeper 中，由于读写锁和独占锁的节点结构不同，读写锁的客户端不用再去竞争创建 lock 节点。所以在一开始，所有的客户端都会创建自己的锁节点。如果不出意外，所有的锁节点都能被创建成功，此时锁节点结构如图3所示。之后，客户端从 Zookeeper 端获取 /share_lock 下所有的子节点，并判断自己能否获取锁。如果客户端创建的是读锁节点，获取锁的条件（满足其中一个即可）如下：

1. 自己创建的节点序号排在所有其他子节点前面
2. 自己创建的节点前面无写锁节点

如果客户端创建的是写锁节点，由于写锁具有排他性。所以获取锁的条件要简单一些，只需确定自己创建的锁节点是否排在其他

子节点前面即可。

不同于独占锁，读写锁的实现稍微复杂一下。读写锁有两种实现方式，各有异同，接下来就来说说这两种实现方式。

读写锁的第一种实现

第一种实现是对 /share_lock 节点设置 watcher，当 /share_lock 下的子节点被删除时，未获取锁的客户端收到 /share_lock 子节点变动的通知。在收到通知后，客户端重新判断自己创建的子节点是否可以获取锁，如果失败，再次等待通知。详细流程如下：

1. 所有客户端创建自己的锁节点
2. 从 Zookeeper 端获取 /share_lock 下所有的子节点，并对 /share_lock 节点设置 watcher
3. 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。
   否则继续等待
4. 持有锁的客户端删除自己的锁节点，其他客户端收到 /share_lock 子节点变动的通知
5. 重复步骤2、3、4，直至无客户端在等待获取锁了

上述步骤对于的流程图如下：

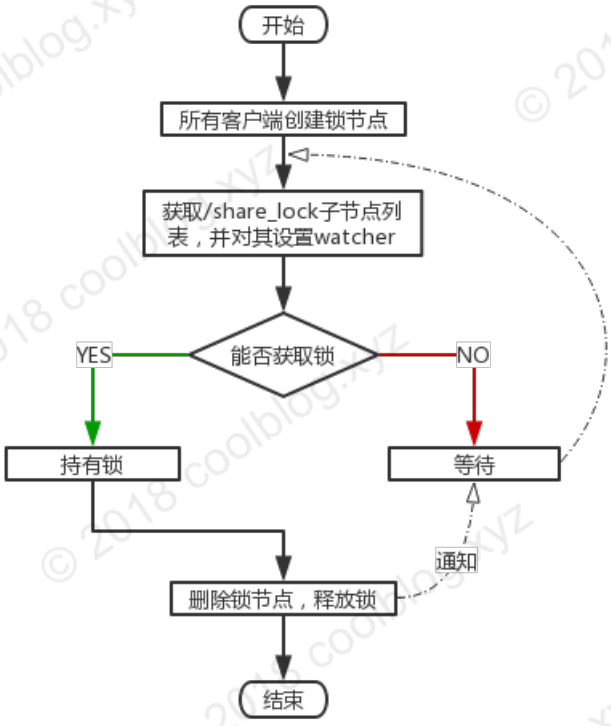


图4 获取读写锁实现1流程图

上面获取读写锁流程并不复杂，但却存在性能问题。以图3所示锁节点结构为例，第一个锁节点 host1-W-0000000001 被移除后，Zookeeper 会将 /share_lock 子节点变动的通知分发给所有的客户端。但实际上，该子节点变动通知除了能影响 host2-R-0000000002 节点对应的客户端外，分发给其他客户端则是在做无用功，因为其他客户端即使获取了通知也无法获取锁。所以这里需要做一些优化，优化措施是让客户端只在自己关心的节点被删除时，再去获取锁。

读写锁的第二种实现

在了解读写锁第一种实现的弊端后，我们针对这一实现进行优化。这里客户端不再对 /share_lock 节点进行监视，而只对自己关心的节点进行监视。还是以图3的锁节点结构进行举例说明，host2-R-0000000002 对应的客户端 C_2 只需监视 host1-W-

0000000001 节点是否被删除即可。而 host3-W-0000000003 对应的客户端 C_3 只需监视 host2-R-0000000002 节点是否被删除即可，只有 host2-R-0000000002 节点被删除，客户端 C_3 才能获取锁。而 host1-W-0000000001 节点被删除时，产生的通知对于客户端 C_3 来说是无用的，即使客户端 C_3 响应了通知也没法获取锁。这里总结一下，不同客户端关心的锁节点是不同的。如果客户端创建的是读锁节点，那么客户端只需找出比读锁节点序号小的最后一个的写锁节点，并设置 watcher 即可。而如果是写锁节点，则更简单，客户端仅需对该节点的上一个节点设置 watcher 即可。详细的流程如下：

1. 所有客户端创建自己的锁节点
2. 从 Zookeeper 端获取 /share_lock 下所有的子节点
3. 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。
   否则对自己关心的锁节点设置 watcher
4. 持有锁的客户端删除自己的锁节点，某个客户端收到该节点被删除的通知，并获取锁
5. 重复步骤4，直至无客户端在等待获取锁了

上述步骤对于的流程图如下：

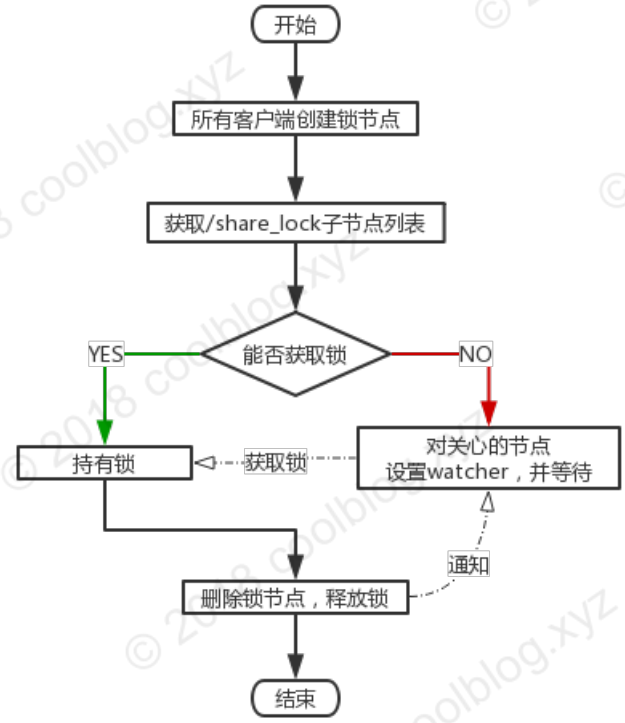


图5 获取读写锁实现2流程图

3. 写在最后

本文较为详细的描述了基于 Zookeeper 分布式锁的实现过程，并根据上面描述的两种锁原理实现了较为简单的分布式锁 demo，代码放在了 github 上（https://github.com/code4wt/distributed_lock），需要的朋友自取。因为这只是一个简单的 demo，代码实现的并不优美，仅供参考。最后，如果你觉得文章还不错的话，欢迎点赞。如果有不妥的地方，也请提出来，我会虚心改之。好了，最后祝大家生活愉快，再见。

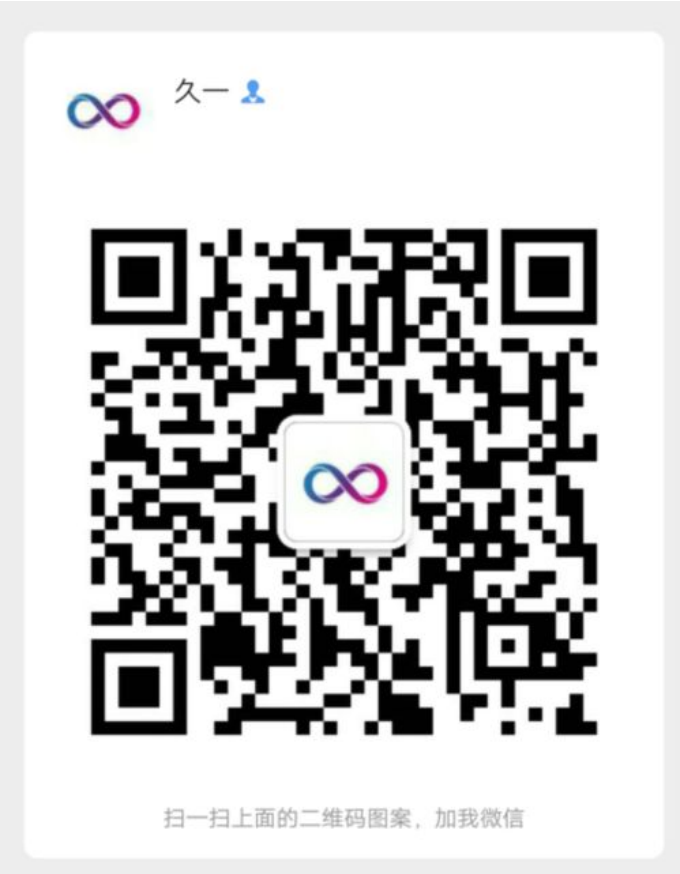
参考

- 《ZooKeeper 分布式过程协同技术详解》
- 《从Paxos到Zookeeper 分布式一致性原理与实践》

【END】

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓扫描二维码进群↓



推荐阅读

1. 送 9 个智能手机，非粉丝请绕道！
2. Nginx 反向代理、负载均衡图文教程 ！
3. 为什么你学不会递归？
4. 一个女生不主动联系你还有机会吗？
5. 团队开发中 Git 最佳实践

阅读原文

微服务中 Zookeeper 的应用及原理

Marvin Mai

Java后端 1月18日

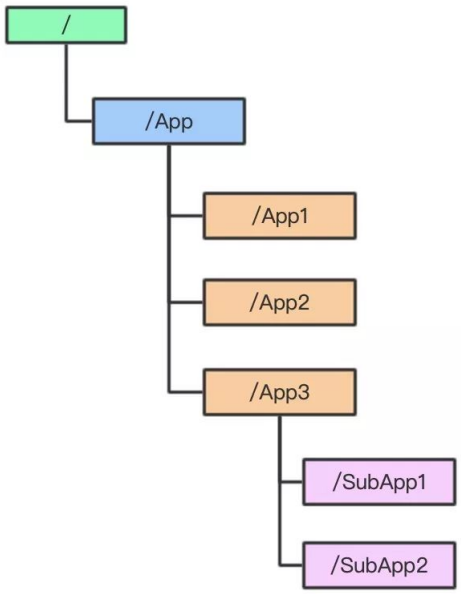
作者 | Marvin Mai

链接 | blog.csdn.net/Mkhaixian2014/article/details/89980476

了解微服务的小伙伴都应该知道Zookeeper，ZooKeeper是一个分布式的,开源的分布式应用程序协调服务。现在比较流行的微服务框架Dubbo、Spring Cloud都可以使用Zookeeper作为服务发现与组册中心。 但是，为什么Zookeeper就能实现服务发现与组册呢?

我们先来了解一下Zookeeper的特性吧，因为它的特性决定了它的使用场景。

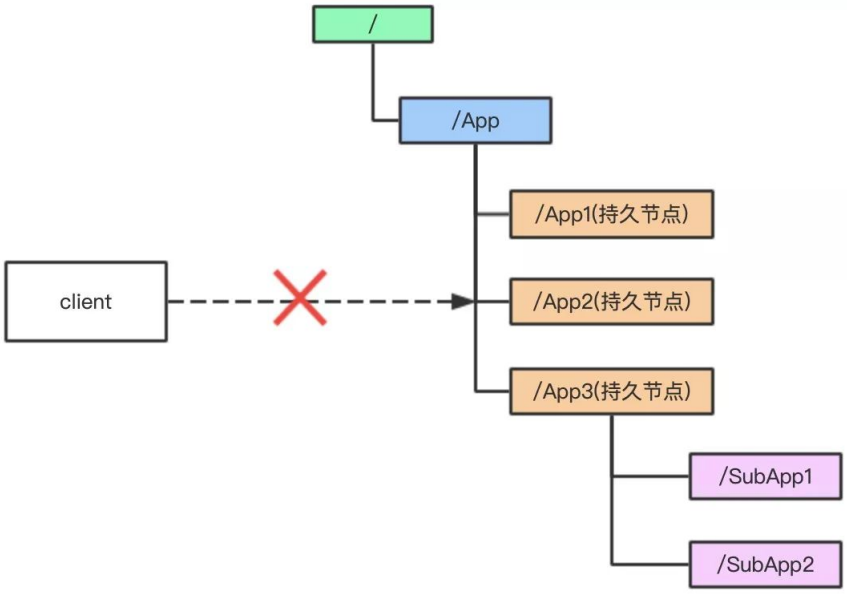
1.树状目录结构



如上图，Zookeeper是一个树状的文件目录结构，有点想应用系统中的文件系统的概念。每个子目录（如App）被称为znode，我们可以对每个znode进行增删改查。

Tips：关注微信公众号：Java后端，获取每日推送。

2.持久节点(Persistent)

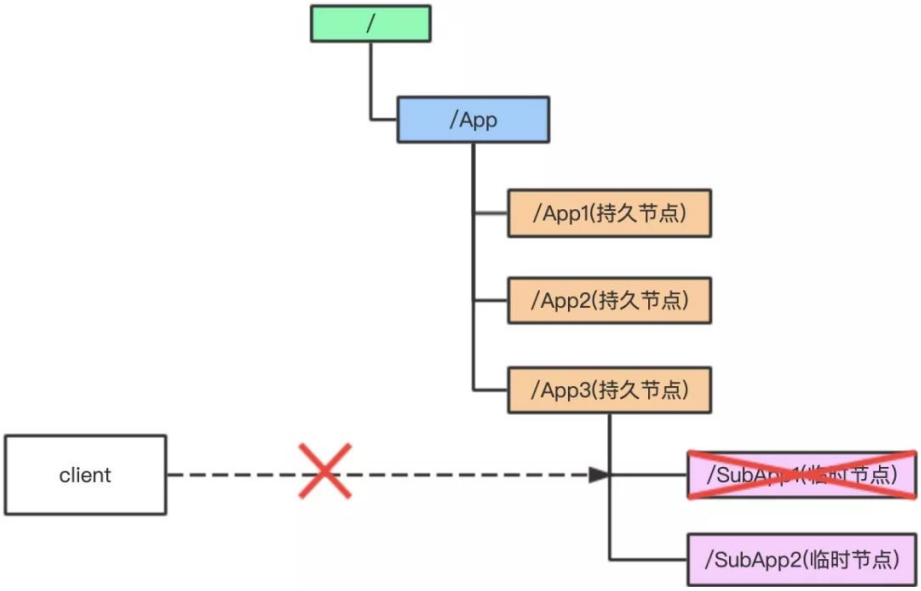


客户端与zookeeper服务端断开连接后，该节点仍然存在。

3.持久有序节点(Persistent_sequential)

在持久节点基础上，由zookeeper给该节点名称进行有序编号，如0000001，0000002。

4.临时节点(Ephemeral)

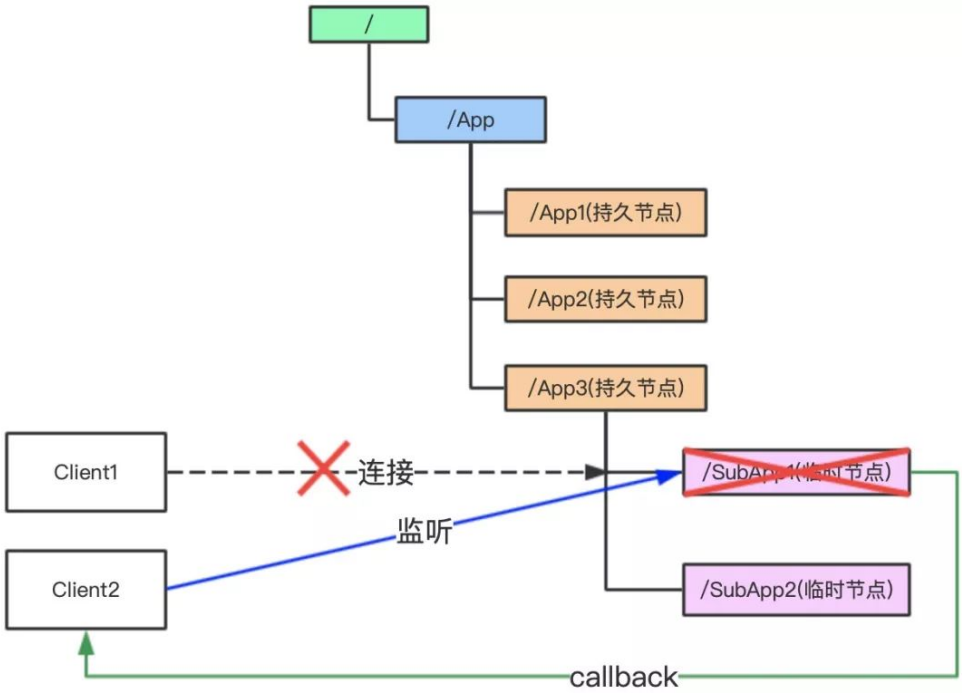


客户端与zookeeper服务端断开连接后，该节点被删除。临时节点下，不存在子节点。

5.临时有序节点(Ephemeral_sequential)

在临时节点基础上，由zookeeper给该节点名称进行有序编号，如0000001，0000002。

6.节点监听(Wacher)

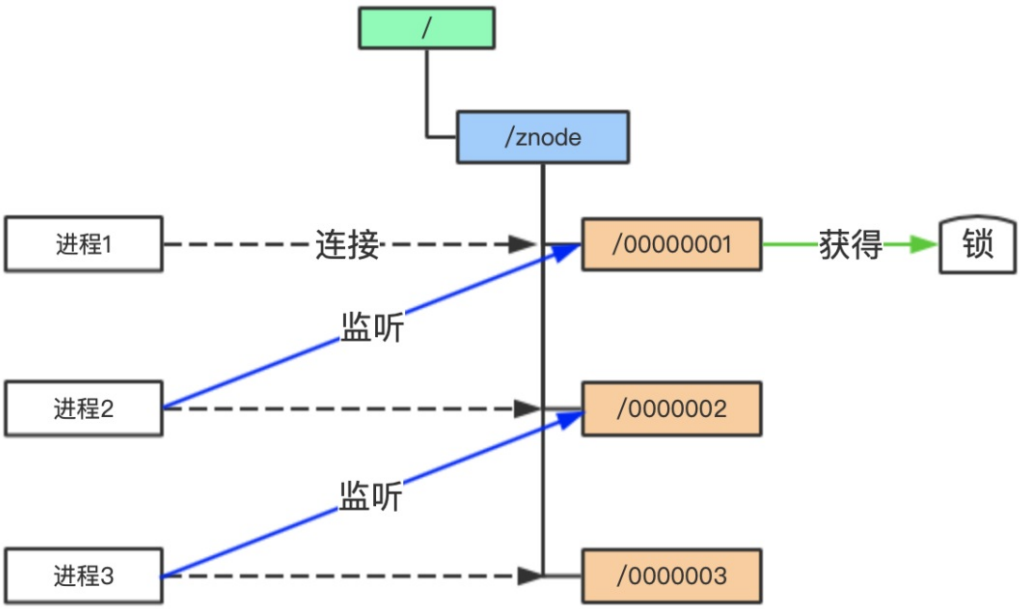


客户端2注册监听它关心的临时节点SubApp1的变化，当临时节点SubApp1发生变化时（如图中被删除的时候），zookeeper会通知客户端2。

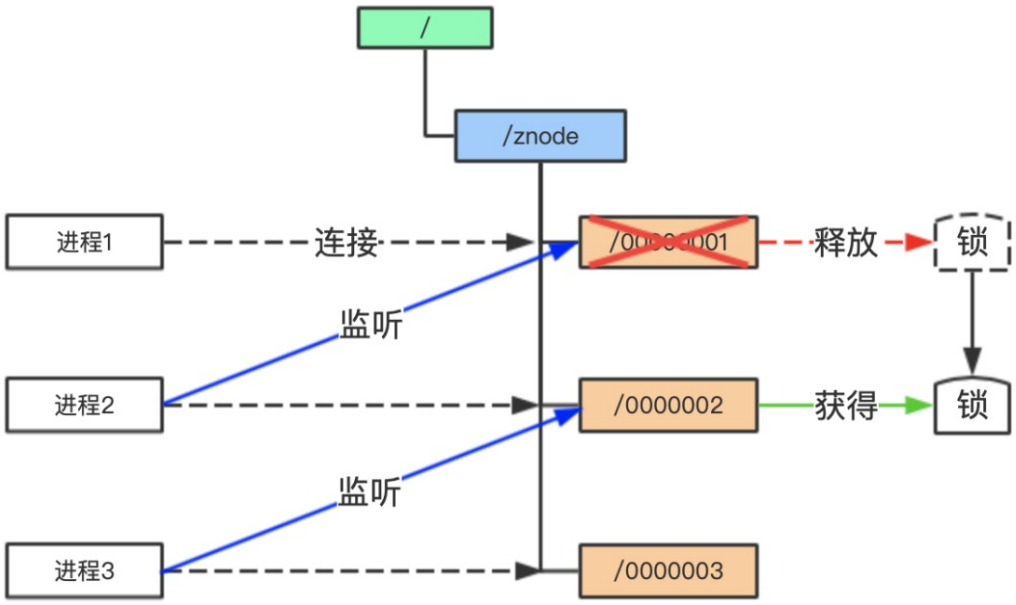
该机制是zookeeper实现分布式协调的重要特性。我们可以通过get，exists，getchildren三种方式对某个节点进行监听。但是该事件只会通知一次。

7.分布式锁

分布式锁主要解决不同进程中的资源同步问题。大家可以联想一下单进程中的多线程共享资源的情况，线程需要访问共享资源，首先要获得锁，操作完共享资源后便释放锁。分布式中，上述的锁就变成了分布式锁了。那这个分布式锁又是如何实现呢？



步骤1： 如图，根据zookeeper有序临时节点的特性，每个进程对应连接一个有序临时节点（进程1对应节点/znode/00000001，进程2对应节点/znode/00000002…如此类推）。每个进程监听对应的上一个节点的变化。编号最小的节点对应的进程获得锁，可以操作资源。

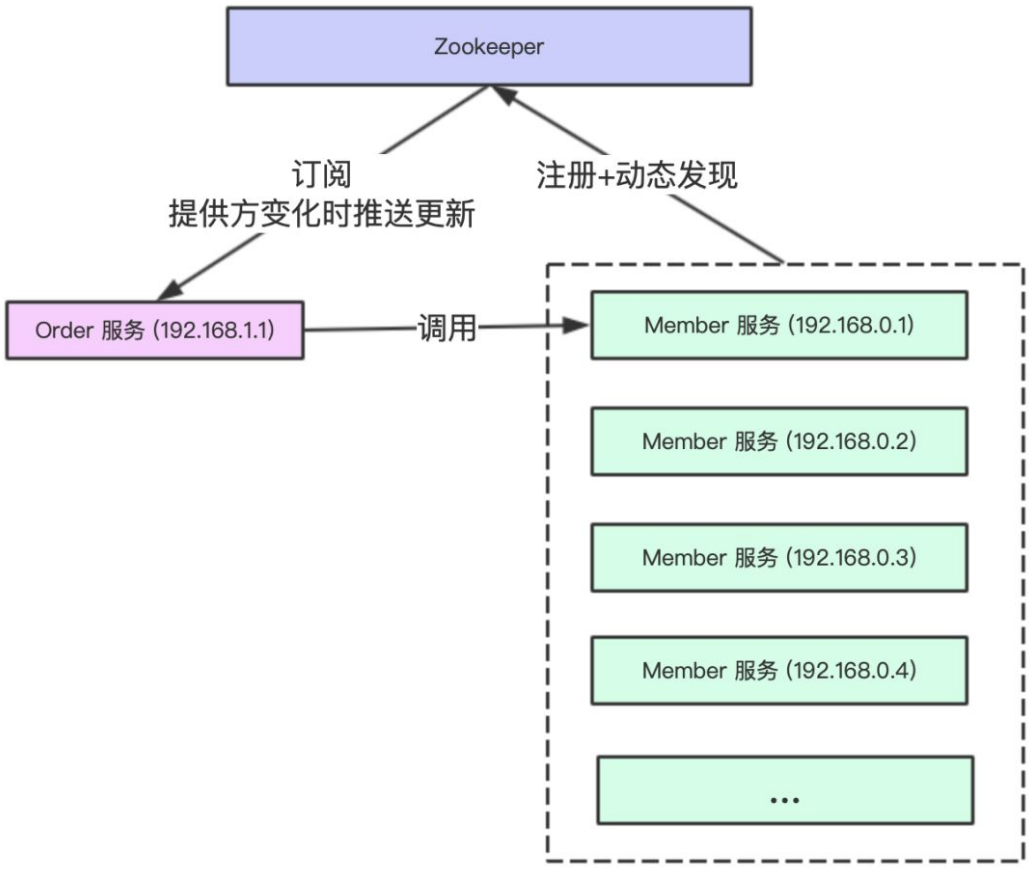


步骤2: 当进程1完成业务后，删除对应的子节点/znode/00000001，释放锁。 此时，编号最小的锁便获得锁（即/znode/00000002对应进程）。

重复以上步骤，保证了多个进程获取的是同一个锁，且只有一个进程能获得锁，就是zookeeper分布式锁的实现原理。

2.服务注册与发现

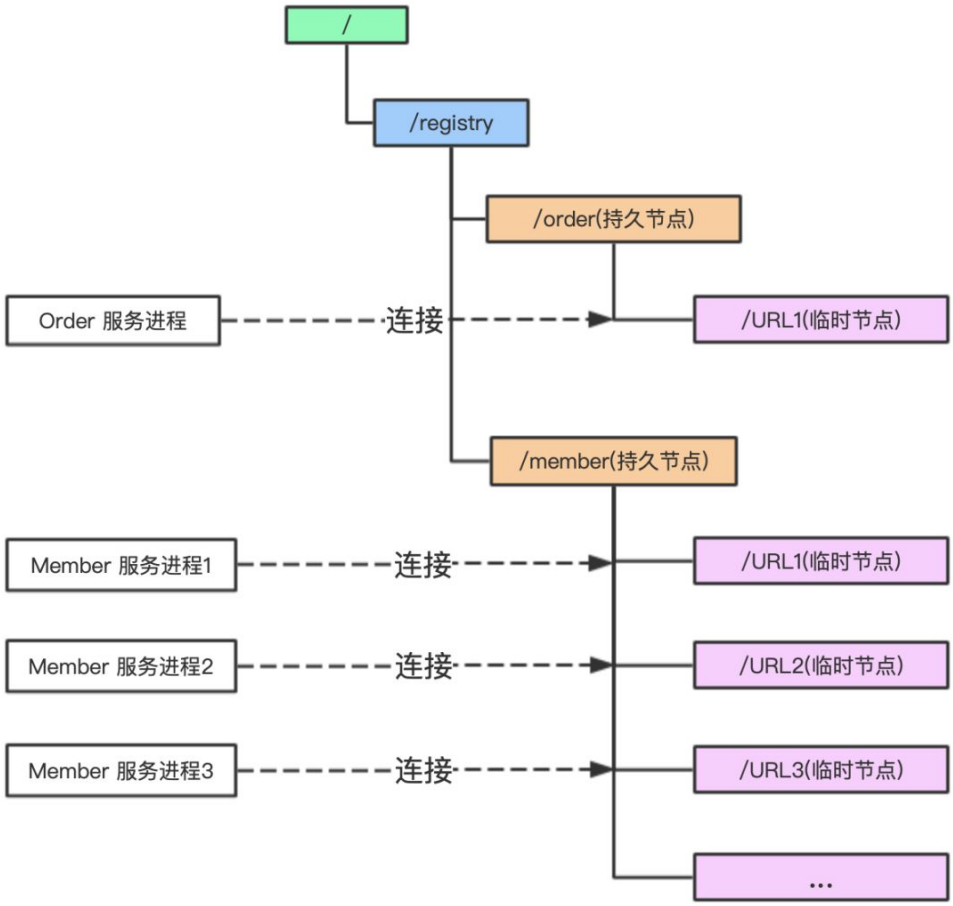
2.1 背景



在微服务中，服务提供方把服务注册到zookeeper中心去如图中的Member服务，但是每个应用可能拆分成多个服务对应不同的Ip地址，zookeeper注册中心可以动态感知到服务节点的变化。

服务消费方（Order 服务）需要调用提供方（Member 服务）提供的服务时，从zookeeper中获取提供方的调用地址列表，然后进行调用。这个过程称为服务的订阅。

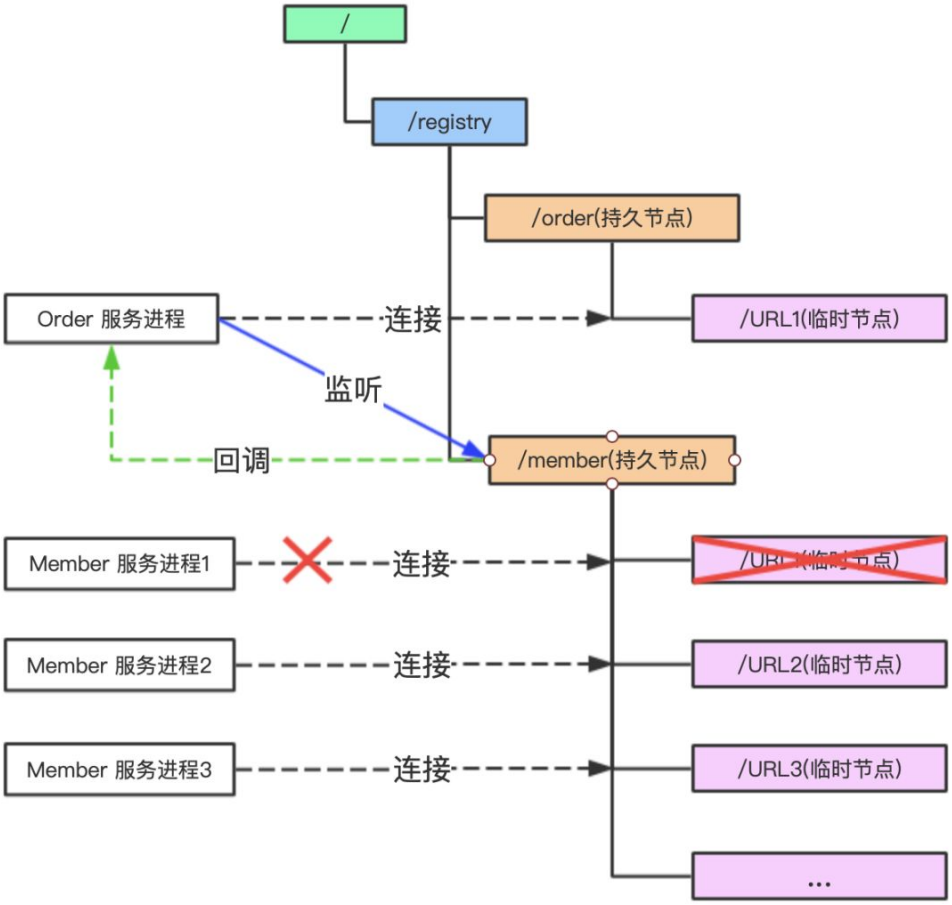
2.2 服务注册原理



rpc框架会在zookeeper的注册目录下，为每个应用创建一个持久节点，如order应用创建order持久节点，member应用创建member持久节点。

然后在对应的持久节点下，为每个微服务创建一个临时节点，记录每个服务的URL等信息。

2.3 服务动态发现原理



由于服务消费方向zookeeper订阅了（监听）服务提供方，一旦服务提供方有变动的时候（增加服务或者减少服务），zookeeper就会把最新的服务提供方列表（member list）推送给服务消费方，这就是服务动态发现的原理。

微信扫描二维码，关注我的公众号

阅读原文

每日一题：Zookeeper 都有哪些使用场景？

Java后端 2019-09-24

原文来自 GitHub 开源社区 Doocs，欢迎 Star 此项目，如果你有独到的见解，同样可以参与贡献此项目。

面试题

简单聊一下 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。

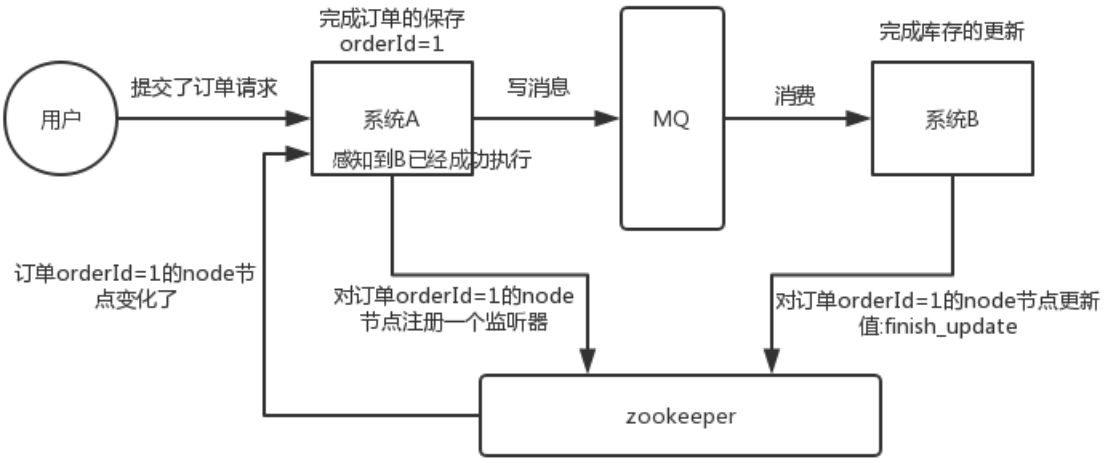
面试题剖析

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

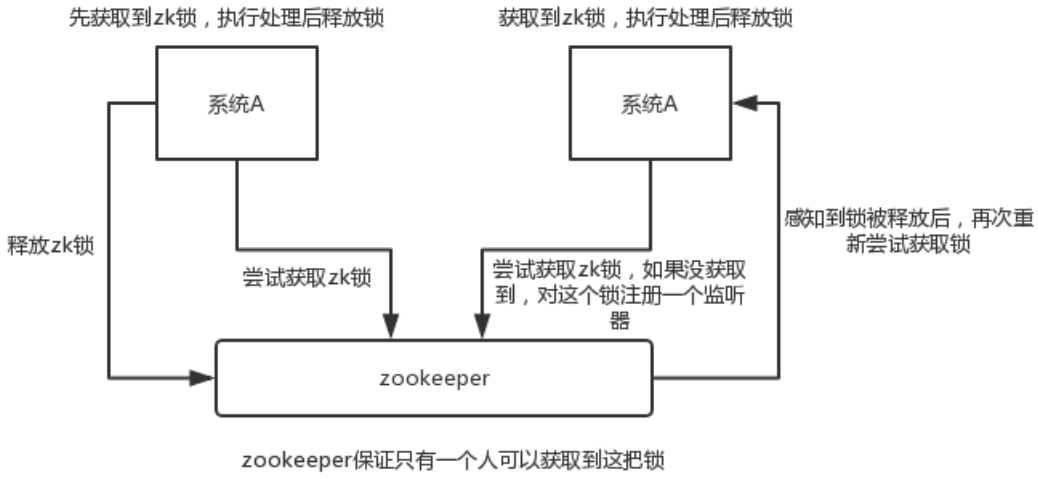
分布式协调

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。

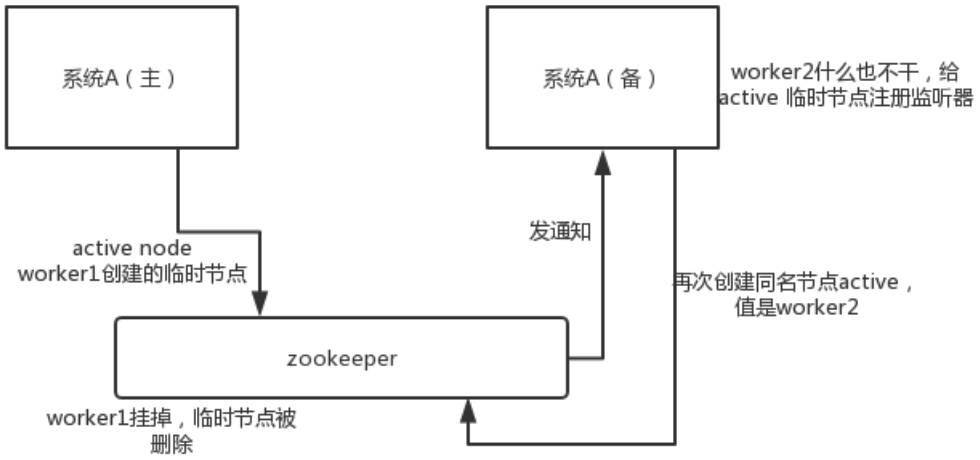


元数据/配置信息管理

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。



-END-

如果看到这里，说明你喜欢这篇文章，帮忙**转发**一下吧，感谢。微信搜索「web_resource」，关注后即可获取每日一题的推送。

推荐阅读

1. 每日一题：消息队列面试常问题目
2. 每日一题：如何保证消息队列的高可用？
3. 每日一题：如何设计一个高并发系统？
4. 每日一题：为什么要进行系统拆分？
5. 每日一题：你有没有做过 MySQL 读写分离？

Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看

阅读原文