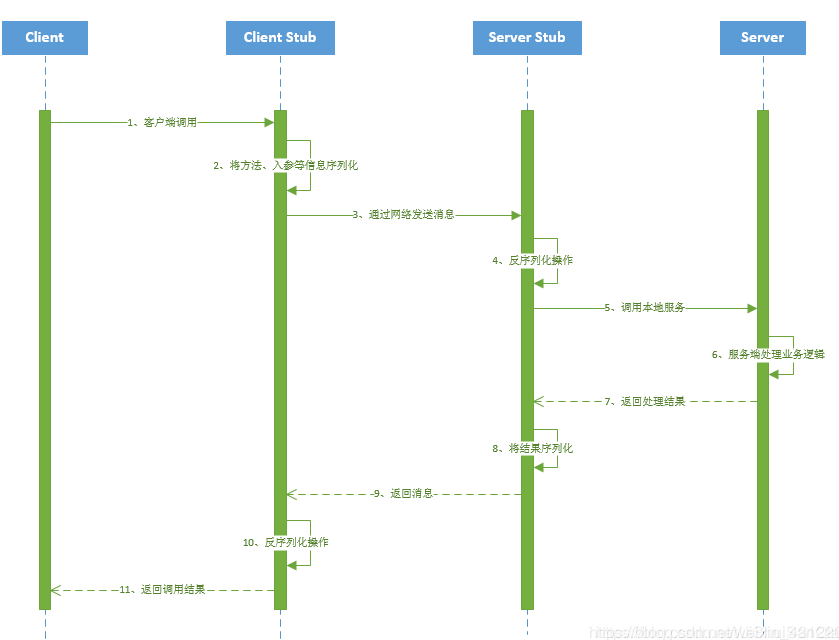
1. Dubbo
2. RPC是什么？

RPC(Remote Procedure Call) 即远程过程调用。它的核心功能有以下几个部分实现的：

1. 客户端：调用远程方法的一端，即服务消费端。
2. 客户端(stub)存根：其实就是代理类。代理类主要做的时间就是将调用方法、类、参数等信息传递到服务端。
3. 网络传输：网络传输就是你要把调用的方法的信息比如说参数啊从客户端传到服务端，然后服务端执行完后服务端再把返回结果通过网络传输给客户端。网络传输的实现方式有很多种，比如基本的socket或者性能以及封装更加优秀的netty(推荐)。
4. 服务端：即服务提供方。
5. 服务端(stub)存根：这个桩不是代理类，是指接收到客户端执行方法的请求后，去指定对应的方法然后返回结果给客户端的类。

具体原理如下图：



第一步，服务消费段以本地调用的方式调用远程服务；

第二步，客户端Stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体(序列化)RpcRequest；

第三步，客户端Stub找到远程服务地址，并将消息发送到服务端；

第四步，服务端Stub接收到消息将消息反序列化为java对象RpcRequest；

第五步，服务端Stub根据RpcRequest中的类、方法、、方法参数等信息调用本地的方法；

第六步，服务端Stub得到方法执行结果并将组装成能够进行网络传输的消息体RpcResponse(序列化)发送给客户端Stub；

第七步，客户端stub接收到消息并将消息反序列化为java对象RpcResponse,这样也就得到了远程调用的结果。

1. Dubbo是什么？

Dubbo是一个分布式、高性能的RPC服务框架，提供服务自动注册、自动发现等高效的服务治理方案，可以和spring无缝集成。

1. Dubbo的核心组件有哪些？

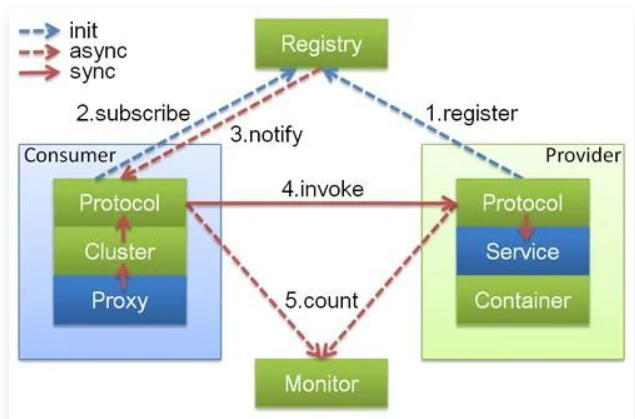
Provider: 服务提供方，向注册中心注册自己的服务。必须

Consumer： 服务消费方，向注册中心订阅自己所需的服务。必须

Registry：服务注册与发现注册中心，会返回服务提供者地址列表给消费者。非必须

Monitor：统计服务的调用次数和调用时间的监控中心，服务消费者和提供者会定时发送统计数据到监控中心。非必需

Container：服务运行容器，负责加载、运行服务提供者。必须



1. Dubbo的服务注册与发现的流程

·服务容器Container负责启动、加载、运行服务提供者；

·服务提供者provider启动时，向注册中心注册服务；

·服务消费者consumer启动时，向注册中心订阅自己所需的服务；

·注册中心registry返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

·服务消费者consumer从注册中心提供的服务提供者地址列表中，基于负载均衡算法，选择一台提供者进行调用，如果调用失败，再根据容错机制选择可用服务进行调用；

·服务消费者consumer和提供者provider在内存中累计调用次数和调用，定时每分钟发送一次统计数据到监测中心monitor。

1. Dubbo的核心功能有哪些？

Remoting:网络通信框架，提供对多种NIO框架抽象封装，包括”同步转异步”和”请求-响应”模式的信息交换方式。

Cluster:服务框架，提供基于接口方法的透明远程过程调用，包括多协议支持，以及软件负载均衡、失败容错、地址路由、动态配置等集群支持。

Registry:服务注册，基于注册中心目录服务，是服务消费者能动态的查找服务提供方，使地址透明，使服务提供者可以方便增加或减少机器。

1. Dubbo的核心配置有哪些？

| **标签** | **用途** | **解释** |
| --- | --- | --- |
| [dubbo:service/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 服务配置 | 用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心 |
| [dubbo:reference/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 引用配置 | 用于创建一个远程服务代理，一个引用可以指向多个注册中心 |
| [dubbo:protocol/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 协议配置 | 用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受 |
| [dubbo:application/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 应用配置 | 用于配置当前应用信息，不管该应用是提供者还是消费者 |
| [dubbo:module/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 模块配置 | 用于配置当前模块信息，可选 |
| [dubbo:registry/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 注册中心配置 | 用于配置连接注册中心相关信息 |
| [dubbo:monitor/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 监控中心配置 | 用于配置连接监控中心相关信息，可选 |
| [dubbo:provider/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 提供方配置 | 当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选 |
| [dubbo:consumer/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 消费方配置 | 当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选 |
| [dubbo:method/](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 方法配置 | 用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息 |
| [dubbo:argument](https://gitee.com/souyunku/DevBooks/blob/master/docs/Dubbo/Dubbo%E6%9C%80%E6%96%B02021%E5%B9%B4%E9%9D%A2%E8%AF%95%E9%A2%98%E5%A4%A7%E6%B1%87%E6%80%BB%EF%BC%8C%E9%99%84%E7%AD%94%E6%A1%88.md) | 参数配置 | 用于指定方法参数配置 |

1. Dubbo支持哪些协议？

dubbo:单一长链接和nio异步通讯，适合大并发小数据量(建议小于100k)的服务调用，以及消费者远大于提供者。传输协议是TCP，异步、hession序列化；默认是服务阻塞的，没有返回值时可以异步调用,尽量不要用该协议传输大文件或者超大字符串；(默认协议)

rmi:采用JDK标准的rmi协议实现，传输参数和返回参数对象需要实现Serializable接口，使用java标准系列化机制，使用阻塞式短连接，消费者和提供者个数差不多，可传文件，传输协议TCP，同步传输，使用常规的远程服务调用与rmi互操作。依赖低版本的Common-Collections包时，java序列化存在安全漏洞；

http:基于http表单提交的远程调用协议，使用spring的HttpInvoke实现。多个短连接，传输协议http，提供者个数多于消费者，需要给应用程序和浏览器js调用；

webservice:基于webservice的远程调用协议，集成cxf实现，提供与原生webservice的互操作。多个短连接，基于http传输，同步传输，使用系统集成和跨语言调用。

Sessian:集成hession服务，基于http通信，采用servlet暴露服务。Dubbo内嵌jeey作为服务器时默认实现，提供与hession服务互操作。多个短连接，同步http传输，hesion序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件;

redis：基于redis实现的RPC协议。

thrif：thrif是facebook捐给apache的一个RPC框架，当前dubbo支持的thrif协议是对thrif原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如service name、magic number等。

1. Dubbo超时时间设置？

在服务提供端和服务消费端都可以设置超时时间，如果都设置了，以服务消费端的超时时间为准。如果请求超时，服务端线程不会停止，直至线程跑完，消费端则会收到异常信息。

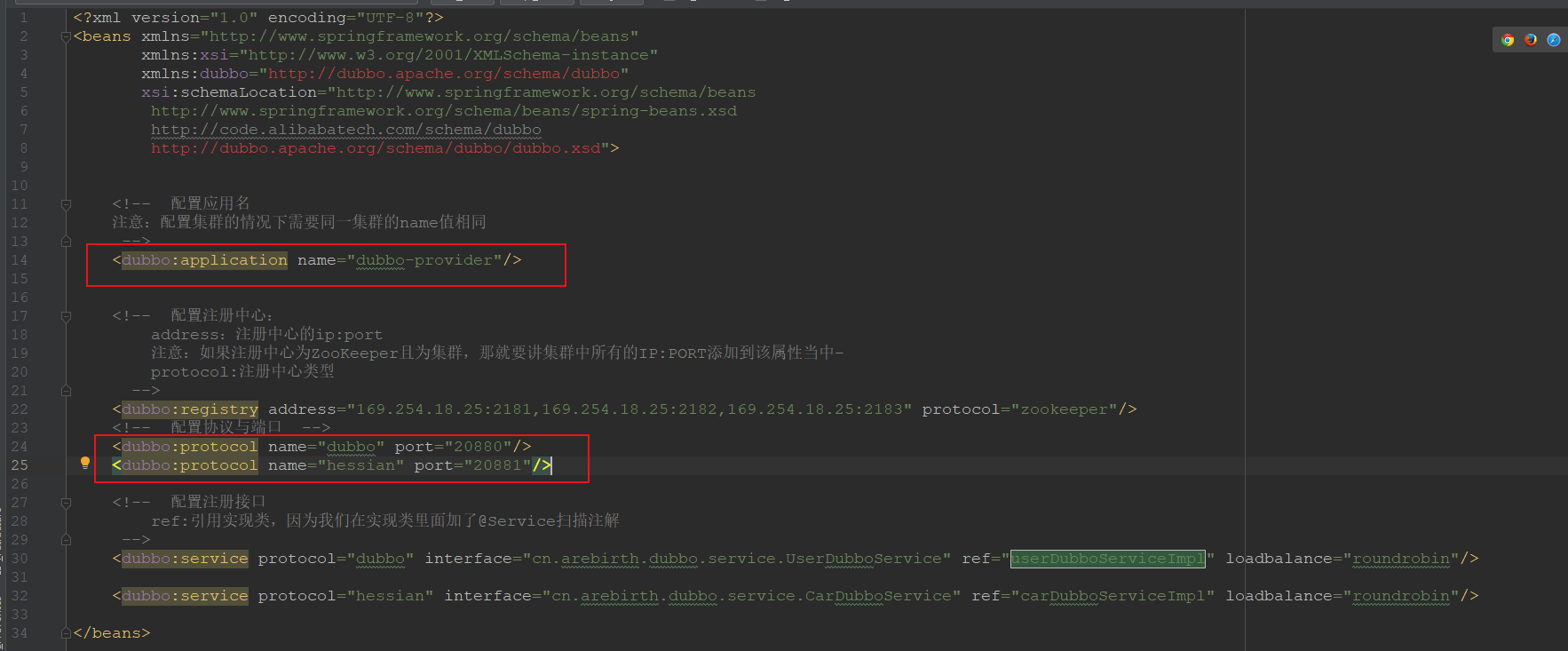
1. Dubbo集群

集群具体的做法是对服务提供者的配置文件进行修改，application name相同，同一协议下端口必须不同。

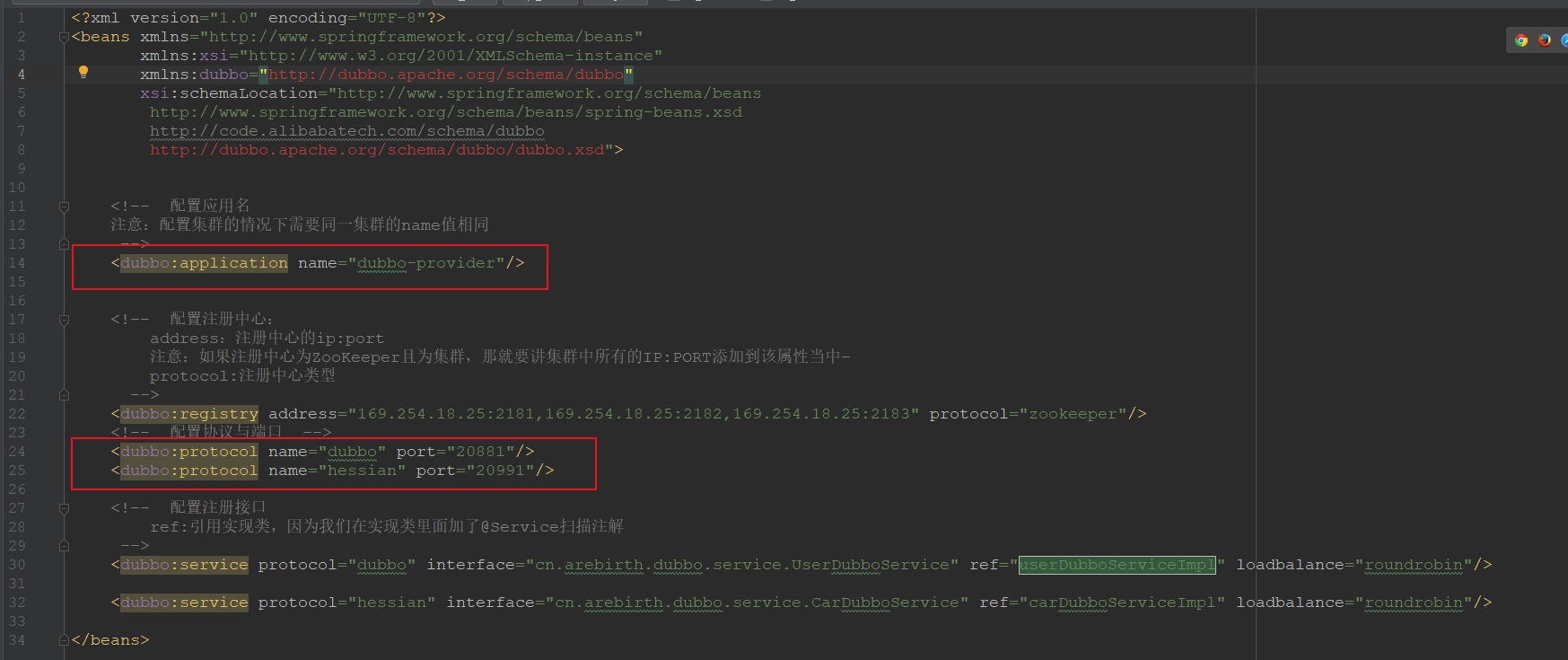


下面来看一下配置的实例：

Provider1:



Provider2:



1. Dubbo集群负载均衡策略

|  |  |
| --- | --- |
| **集群负载均衡方案** | **说明** |
| Random LoadBalance | 随机选取提供者策略，有利于动态调整提供者权重。(默认) |
| RoundRobin LoadBalance | 轮询选取提供者策略，平均分布，但是存在请求累积的问题。比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。 |
| LeastActive LoadBalance | 最少活跃调用策略，使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。解决慢提供者接收更少的请求。 |
| ConstantHash LoadBalance | 一致性Hash策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动。 |

1. Dubbo集群容错方案

|  |  |
| --- | --- |
| **集群容错方案** | **说明** |
| Failover cluster(失败自动切换) | 服务调用失败时，自动重试(默认2次)其他服务器(默认) |
| Failfast Cluster(快速失败) | 服务调用失败时，立即报错，直发起一次调用 |
| Failsage Cluster(失败安全) | 服务调用失败时，直接忽略 |
| Failback Cluster(失败自动回复) | 服务调用失败时，记录失败请求，定时重发 |
| Forking Cluster(并行调用多个服务器) | 服务调用失败时，并行调用多个服务器，只要一个成功即返回 |
| Broadcast Cluster | 广播逐个调用所有提供者，任意一个报错则报错 |

1. Dubbo中zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么？

可以通信的，启动dubbo时，消费者会从zk拉取注册的生产者的地址接口等数据，缓存在本地，每次调用，按照本地存储的地址进行调用。

注册中心对等集群，任意一台注册中心宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通信。服务提供者无状态，任意一台宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复。但如果是新增dubbo服务，注册中心挂掉就会有对新增的服务有影响。

1. Dubbo支持的注册中心

Zookeeper:

优点：支持分布式，很多周边产品。

缺点：受限于zk自身的稳定性。

Multicast:

优点：去中心化，不需要单独安装软件。

缺点：provider和consumer、registry不能跨机房(路由)。

redis:

优点：支持集群，性能高。

缺点：要求服务器时间同步，否则可能出现集群失败的问题。

simple:

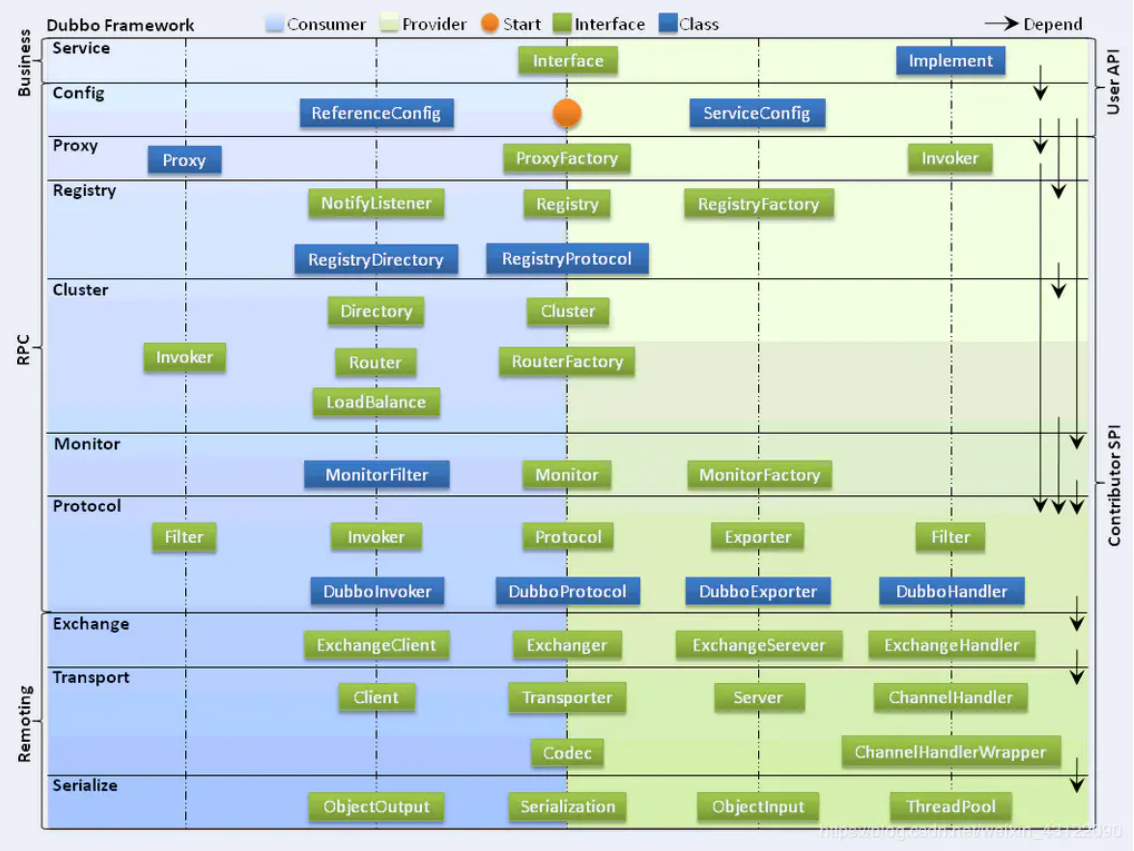
优点：标准RPC服务，没有兼容问题。

缺点：不支持集群。

1. Dubbo的配置文件是如何加载到spring中的？

Spring容器在启动的时候，会读取dubbo.properties文件，如果找不到或者没有找到dubbo相关配置参数，就去其他properties配置文件去寻找dubbo配置,同时，会加载dubbo的xml配置文件，在xml文件中会有spring默认的一些schema和dubbo自定义的schema，每个schema对应一个自己的NamespaceHandler,在NamespaceHandler里会通过BeanDefinitionParser解析配置信息并转化为需要加载的Bean对象。

1. Dubbo中用到的一些设计模式？
2. Dubbo架构设计



·接口服务层（Service）：该层与业务逻辑相关，根据 provider 和 consumer 的业务设计对应的接口和实现

·配置层（Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心

·服务代理层（Proxy）：服务接口透明代理，生成服务的客户端 Stub 和 服务端的 Skeleton，以ServiceProxy 为中心，扩展接口为 ProxyFactory

·服务注册层（Registry）：封装服务地址的注册和发现，以服务 URL 为中心，扩展接口为RegistryFactory、Registry、RegistryService

·路由层（Cluster）：封装多个提供者的路由和负载均衡，并桥接注册中心，以Invoker 为中心，扩展接口为 Cluster、Directory、Router 和 LoadBlancce

·监控层（Monitor）：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为MonitorFactory、Monitor 和 MonitorService

·远程调用层（Protocal）：封装 RPC 调用，以 Invocation 和 Result 为中心，扩展接口为Protocal、Invoker 和 Exporter

·信息交换层（Exchange）：封装请求响应模式，同步转异步。以 Request 和Response 为中心，扩展接口为 Exchanger、ExchangeChannel、ExchangeClient 和 ExchangeServer

·网络传输 层（Transport）：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为Channel、Transporter、Client、Server 和 Codec

·数据序列化层（Serialize）：可复用的一些工具，扩展接口为 Serialization、ObjectInput、ObjectOutput 和 ThreadPool

1. Dubbo在安全机制方面是如何解决的？

Dubbo通过token令牌防止用户绕过注册中心直连，然后在注册中心上管理授权，Dubbo还提供服务黑白名单，来控制服务所允许的调用方。

1. Dubbo和spring cloud的区别

最大的区别：

Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

而 SpringCloud 是基于 Http 协议+Rest 接口调用远程过程的通信，相对来说，Http 请求会有更大的报文，占的带宽也会更多。但是REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖。

1. Dubbo直连

在开发环境中，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连。直连将以接口为的那位，忽略注册中心的提供者列表。有三种方式可以进行直连：

首先修改,修改服务提供者的配置文件provider.xml，禁用注册中心,如下：

<dubbo:registry address="127.0.0.1:2181" register="false" />

或者

<dubbo:registry address="127.0.0.1:2181?register=false" />

方式一：在消费者配置文件consumer.xml中指定url，如下：



方式二：在JVM启动参数中加入-D参数映射服务地址，如图：



方式三:映射配置文件

在本机端脑用户下新建文件dubbo-resolve.properties,然后再文件中加入需要直连的服务，如下图：



1. Dubbo通信协议dubbo协议为什么消费者比提供者个数多?为什么不适合传大包?

因为dubbo协议采用单一长连接，假设网络为千兆网卡(1024Mbit/s=128MByte/s),根据测试经验数据每条连接最多只能压满7MByte/s(假设带宽为54Mbps时,带宽不同压满数也不一样)，理论上一个服务提供者需要20个消费者才能压满网卡。所以消费者比提供者个数多。

1Byte=8bit，1024Mbit=128MByte，54Mbps=7Mbyte/s

Mbps=Mbit/s 即兆比特每秒(1000000bit/s),是一种传输速率单位，传输速率是设备的数据交换能力，也叫”带宽”,单位是Mbps,目前主流的集线器带宽主要有10Mbps、54Mbps/100Mbps自适应型、100Mbps、150Mbps四种。

那为什么不适合传大数据包呢？

假如，每次请求的数据包大小为500KByte,假设网络为千兆(1024Mbit/s=128MByte/s)，带宽为54Mbps(7MByte/s),那么单个服务提供者的TPS(每秒处理事务数)最大为：128\*1024KByte/500KByte=262。单个消费者调用单个服务者的TPS(每秒处理事务数)最大为：7\*1024KByte/500KByte = 14。所以数据包越大支持的并发数越小。

请求的连接数受网卡限制，不受单个连接传输速率(带宽)限制。单个连接受传输速率(带宽)限制。

1. Dubbo服务降级

<https://blog.csdn.net/zhaokuner/article/details/106748622>

<https://www.cnblogs.com/raoshaoquan/articles/6636067.html>

1. Dubbo如何优雅停机？

Dubbo是通过JDK的ShutdownHook来完成优雅停机的，所以如果使用kill -9 PID等强制关闭指令，是不会执行优雅停机的，只有通过kill PID才会执行。

1. Dubbo SPI与java SPI？

SPI,全称Service Provider Interface,是被第三方实现或者扩展的API，它可以用来启动框架扩展和替换组件。

<https://blog.csdn.net/cnm10050/article/details/105026393>

<http://www.cainiaoxueyuan.com/bc/4635.html>

1. TPS、QPS、系统吞吐量的区别和理解
2. TPS:

Transaction Per Second,即每秒处理的事务数。一个事务是指客户端向服务器发送请求后服务器做出响应的过程，从发送请求开始到收到响应结果结束，以此来计算使用的时间和完成的事务数量,可以是一个接口、多个接口、一个业务流程等等。

如果每秒能够完成n次一个事务，TPS就是n。

1. QPS:

Queries Per Second,即每秒查询率，是一台服务器每秒能够响应的查询次数(数据库中的每秒执行查询sql的次数)，显然，这个不够全面，不能描述增、删、改。

1. TPS与QPS区别：

如果是对一个查询接口压测，且这个接口内部不会再去请求其他接口，那么TPS=QPS，否则TPS≠QPS，比如访问一个页面会请求三次，一次请求会产生一个“T”,产生三个“Q”。

1. 吞吐量：

或者叫做最大吞吐能力，每秒的响应请求数量,跟TPS一个意思。一个系统的吞吐量与请求对cpu的消耗、还不接口、IO等等紧密关联。单个请求对CPU消耗越高，外部系统接口、IO影响速度越慢，系统吞吐能力越低，反之越高。

吞吐量/TPS = 系统同时处理的请求并发数/平均响应时间。

1. 性能相关

<https://www.cnblogs.com/uncleyong/p/11059556.html>

1. Zookeeper
2. Zookeeper是什么？

zk是一个开放源码的分布式协调服务。分布式应用程序可以基于zk实现服务发布/订阅、命名服务、分布式锁、集群管理(Master选举、负载均衡、分布式协调/通知、分布式队列)等功能。**zk提供了文件系统和消息通知**。

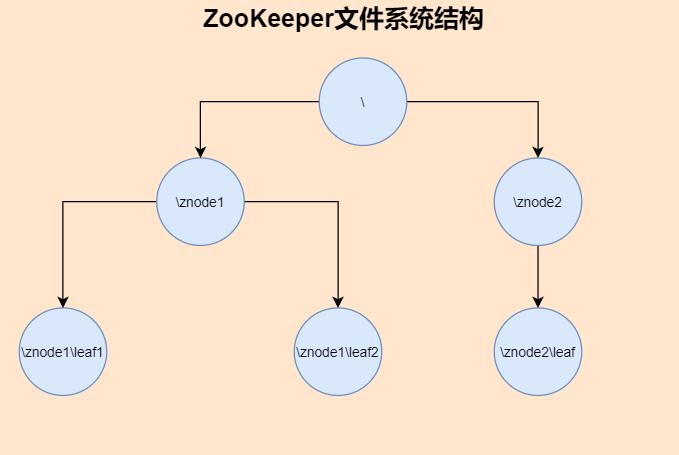
客户端的读请求(服务消费者)可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是有所连接的zk机器来处理。**对于写请求会同时发送给其他zk机器并且达成一直，请求才会返回成功。因此，随着zk集群中机器的增多，读请求的吞吐会提高但是写请求的吞吐会下降**。

1. Zookeeper文件系统

zk提供一个多层级的节点(znode)命名空间，这个层级就是一套树形存储结构，类似于linux文件系统路径，也是从根节点开始，可以用来保存数据。与文件系统不同的是，这些节点都可以设置关联的数据，而linux文件系统中只有文件节点可以存放数据而目录节点不行。zk为了保证高吞吐量和低延时，在内存中维护这个树形的目录结构，从而导致zk不能存放大量的数据，每个节点存放的数据上线为1M。

节点类型：

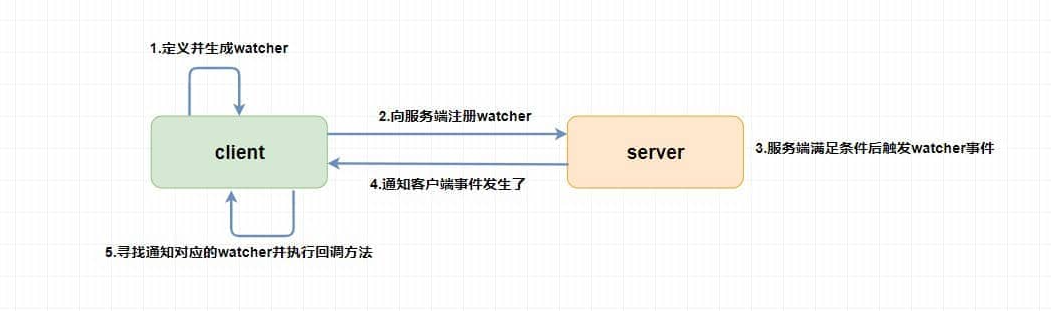
|  |  |
| --- | --- |
| **节点类型** | **节点描述** |
| PERSISTENT(持久化节点) | 当客户端与zk断开连接之后，该节点依旧存在；  只有删除节点操作时，该节点才会消失；  可创建子节点。 |
| PERSISTENT\_SEQUENTIAL(持久化编号节点) | 当客户端与zk断开连接之后，该节点依旧存在，只是zk给该节点名称进行了顺序编号；  只有删除节点操作时，该节点才会消失；  可创建子节点。 |
| EPHEMERAL(临时节点) | 当客户端与zk断开连接之后，该节点被删除；  该节点不能创建子节点。 |
| EPHEMAERAL\_SEQUENTIAL(临时编号节点) | 当客户端与zk断开连接之后，该节点被删除，只是zk给该节点进行了顺序编号；  该节点不能创建子节点。 |



1. Zookeeper消息(数据变更)通知以及watcher机制

zk采用了观察者设计模式，客户端会向某个znode注册一个watcher事件来监听该znode是否发生变化，一旦znode发生改变(比如当前znode数据的新增、修改或删除或者子znode的新增、修改或删除)，就会触发watcher事件，zk服务端向指定客户端发送一个事件通知，客户端接收到事件通知之后,做出业务上的改变。

工作机制如图：



watch机制工作流程如下：

<https://www.runoob.com/w3cnote/zookeeper-watcher.html>

<https://blog.csdn.net/SCUTJAY/article/details/106835774>

<https://www.cnblogs.com/shamo89/p/9787176.html>

<https://www.cnblogs.com/wuzhenzhao/p/9994450.html>

<https://blog.csdn.net/fenglongmiao/article/details/79305010>

1. **客户端生成Watcher对象，向服务端注册Watcher的请求**：

首先，客户端定义并生成一个**Watcher对象**，然后通过zk客户端的**getData**、**getChildren**和**exsit**三个接口向zk服务端注册watcher对象。注册之后，客户端会将当前客户端请求进行**标记**，将其设置为watcher监听，同时会封装一个Watcher的注册信息**watchRegistration**对象，用于暂时**保存数据节点路径和watcher对应关系**。

在zk中，**Packet**可以看作一个最小通信单元，用于进行客户端与服务端之间的网络传输，WatchRegistration对象会被封装到Packet中，然后放入发送队列中等待客户端发送。随后客户端就会启动**sendThread线程**向服务端发送这个请求，该线程的**readResponse()方法**同时等待请求的返回，然后将watcher注册到**Watchmanager**中进行管理。

1. **服务端满足条件后处理watcher**

服务端接收到客户端的注册请求后，会执行以下几个过程：

1. **封装WatchedEvent**

将通知状态、事件类型以及节点路径封装成一个WatchedEvent。WatchedEvent是整个watcher通知机制的最小单元。

1. **查询Watcher**

根据节点路径查找对应的watcher。如果没找到watcher，说明没有任何客户端在该数据节点上注册过watcher，直接退出。如果找到了watcher，从**WatcherManager的watchTable和watch2Paths两个存储结构中删除watcher**。

注意：WatchManager是zk服务端watcher的管理者，其内部管理的watchTable和watch2paths两个存储结构，其中watchTable是从**数据节点路径的粒度**来托管Watcher，watch2Paths是从watcher的粒度来控制事件触发需要触发的数据节点。

1. **调用process方法来触发watcher**

当满足事件触发条件后，服务端给客户端发送一个通知，将WatchedEvent包装成WatcherEvent,以便进行网络传输序列化。

1. **客户端回调Watcher**

SendThread线程的readResponse()方法接收服务端发送过来的数据，该方法中会对replyHdr中的xid进行判断，等于-1，表明是一个通知类型的响应，会作以下处理：

1. **将字节流反序列化成WatcherEvnet对象**；
2. **处理chrooPath**：如果客户端设置了chrootPath属性，那么需要对服务端传过来的完整的节点路径进行chrootPath处理，生成客户端的一个相对节点路径。
3. **将WatcherEvent对象转换成WatchedEvent对象**。
4. **回调Watcher**:SendThread通过EventThread.queueEvent方法将WatchedEvent对象交给EventThread线程队列进行串行同步处理。

**代码如下**：

void readResponse() throws IOException {

ByteBufferInputStream bbis = new ByteBufferInputStream(incomingBuffer);

BinaryInputArchive bbia = BinaryInputArchive.getArchive(bbis);

ReplyHeader replyHdr = new ReplyHeader();

replyHdr.deserialize(bbia, "header");

if (replyHdr.getXid() == -2) {

// -2 is the xid for pings

if (LOG.isDebugEnabled()) {

LOG.debug("Got ping response for sessionid: 0x"

+ Long.toHexString(sessionId) + " after "

+ ((System.nanoTime() - lastPingSentNs) / 1000000) + "ms");

}

return;

}

if (replyHdr.getXid() == -4) {

// -4 is the xid for AuthPacket

if(replyHdr.getErr() == KeeperException.Code.AUTHFAILED.intValue()) {

zooKeeper.state = States.AUTH\_FAILED;

eventThread.queueEvent( new WatchedEvent(Watcher.Event.EventType.None, Watcher.Event.KeeperState.AuthFailed, null) );

}

if (LOG.isDebugEnabled()) {

LOG.debug("Got auth sessionid:0x"+ Long.toHexString(sessionId));

}

return;

}

if (replyHdr.getXid() == -1) {

// -1 means notification

if (LOG.isDebugEnabled()) {

LOG.debug("Got notification sessionid:0x" + Long.toHexString(sessionId));

}

WatcherEvent event = new WatcherEvent();

event.deserialize(bbia, "response");

// convert from a server path to a client path

if (chrootPath != null) {

String serverPath = event.getPath();

if(serverPath.compareTo(chrootPath)==0)

event.setPath("/");

else

event.setPath(serverPath.substring(chrootPath.length()));

}

WatchedEvent we = new WatchedEvent(event);

if (LOG.isDebugEnabled()) {

LOG.debug("Got " + we + " for sessionid 0x" + Long.toHexString(sessionId));

}

eventThread.queueEvent( we );

return;

}

if (pendingQueue.size() == 0) {

throw new IOException("Nothing in the queue, but got "+ replyHdr.getXid());

}

Packet packet;

synchronized (pendingQueue) {

packet = pendingQueue.remove();

}

/\*

\* Since requests are processed in order, we better get a response

\* to the first request!

\*/

try {

if (packet.header.getXid() != replyHdr.getXid()) {

packet.replyHeader.setErr(KeeperException.Code.CONNECTIONLOSS.intValue());

throw new IOException("Xid out of order. Got " + replyHdr.getXid() + " expected " + packet.header.getXid());

}

packet.replyHeader.setXid(replyHdr.getXid());

packet.replyHeader.setErr(replyHdr.getErr());

packet.replyHeader.setZxid(replyHdr.getZxid());

if (replyHdr.getZxid() > 0) {

lastZxid = replyHdr.getZxid();

}

if (packet.response != null && replyHdr.getErr() == 0) {

packet.response.deserialize(bbia, "response");

}

if (LOG.isDebugEnabled()) {

LOG.debug("Reading reply sessionid:0x" + Long.toHexString(sessionId) + ", packet:: " + packet);

}

} finally {

finishPacket(packet);

}

}

1. Zookeeper特性？

①顺序一致性

从同一客户端发起的事务请求，最终将会严格按照顺序被应用到zk中。

有序性是zk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zk最新的zxid。

②原子性

所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。

③单一系统映像

无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。

④可靠性

一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

1. Zookeeper部署方式

**zk部署需要先安装jdk,配置环境变量JAVA\_HOME,因为zk服务器使用java编写创建，它运行在JVM，所以需要使用JDK1.6或更高版本。**

①单机部署

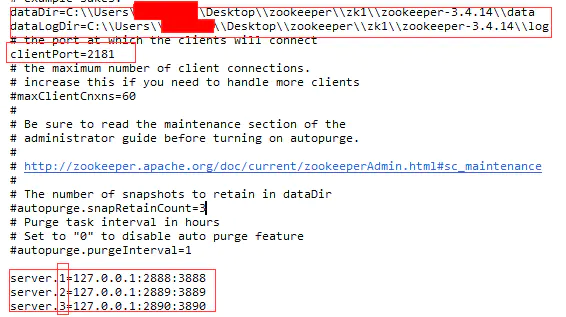


②伪集群部署

**第一步**，**将单机部署模式下配置好的zk文件夹复制三分**，分别命名为zk1、zk2、zk3，代表集群中的三个节点；



**第二步**，**分别修改zk1、zk2、zk3下的/conf/zoo.cfg中的配置**，修改如下(zk2、zk3下的配置与zk1类似，**不同之处在于：dataDir和dataLogDir的路径、clientPort端口号**)：



server.num=ip/domain：port1：port2，参数含义如下：

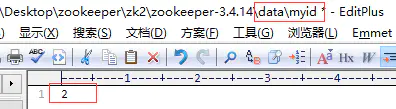
num：表示第几号服务器；

ip/domain ：服务器域名或者ip地址。

port1：当前服务器和Leader服务器交换信息的端口；

port2：当前Leader服务器宕机后，重新选举Leader时服务器之间交换信息的端口；

**第三步**，在zk1、zk2、zk3的data文件夹下**创建myid文件，文件中添加当前服务器num**；



**第四步**，分别启动每个zk服务器，启动的过程中会报错，这是因为当前服务器节点发现其他节点不在线，等集群节点全部启动会就会显示正常，在启动完成之后，会有一个节点当选为Leader节点。如下图：

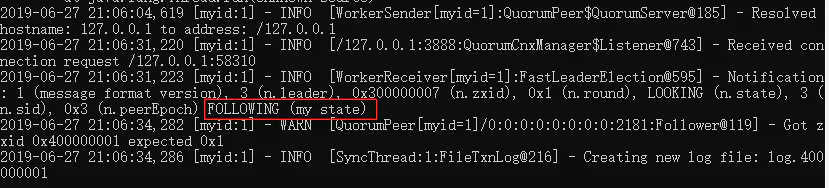


图-zk1服务器节点

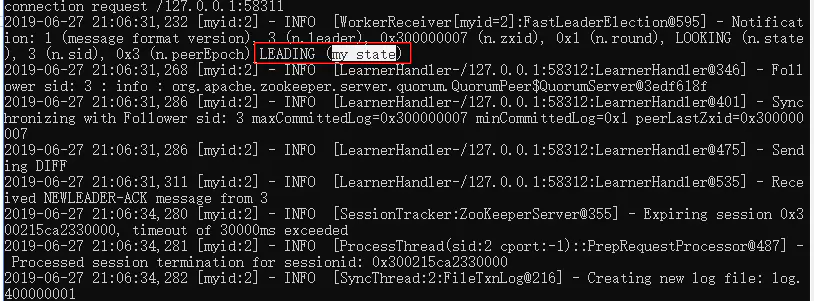


图-zk2服务器节点

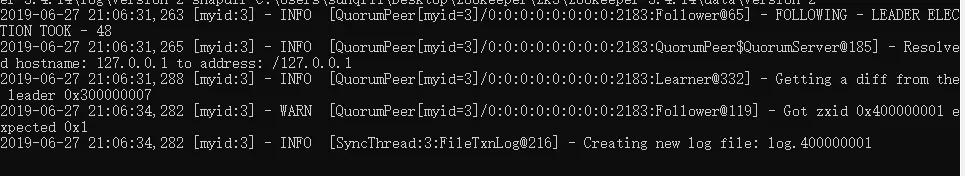


图-zk3服务器节点

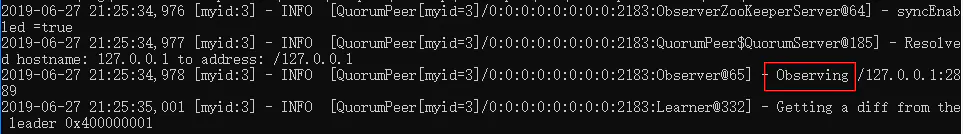
如果在伪集群中**设置observer角色**，则配置需要略微调整。假设zk3节点变成observer角色，则zk1、zk2、zk3修改如下：

server.1=127.0.0.1:2888:3888

server.2=127.0.0.1:2889:3889

server.3=127.0.0.1:2890:3890:observer

另外，zk3的zoo.cfg需要添加一个配置：peerType=observer，以用来开启observer；启动日志如下图：



③集群

安装方式与伪集群部署操作一样。

1. Zookeeper集群

①集群中的角色

**Leader**:领导者角色。集群通过选举过程从所有服务器中选举其中一台作为Leader，它为客户端提供读和写服务，他是整个集群工作机制的核心。它会发起并维护与各Follower以及Observer间的心跳。主要承担的任务为：

·**事务请求(写请求)的唯一调度者,也可以处理处理写请求**，以保证集群**事务处理的顺序性**；

·集群内部各节点的调度者；

·接受所有Follower的提案请求并统一协调发起投票,负责与所有Follower进行内部数据交换；

**Follower**:追随者角色。主要承担的任务为：

·参与Leader选举投票；

·处理客户端非事务请求(读请求)；

·转发事务请求(创建、更新、删除节点等请求)给Leader服务器；

·参与事务请求的同步提案投票，同时与Leader进行数据交换(同步)；

**Observer**:观察者，观察zk集群的最新状态变化，并将这些状态变更同步过来。Observer服务器在工作原理上和Follower基本一致。对于非事务请求，可以直接独立处理，而对于事务请求，则会转发给Leader服务器进行处理。和Follower唯一区别是，**Observer不参与任何投票**，包括事情请求提交投票、Leader选举投票，仅仅是观察集群状态变化并把变化同步过来。

②集群中为什么要有Observer角色？

因为zk集群中，对于事务请求，Leader会发送所有Follower节点让他们投票，Follower节点越多，花在事务投票同步过程和Leader选举过程的时间就越长，而Observery与Follower一样对事务请求可以直接处理。所以提高了集群的读性能而不影响写性能。如果急群众Observer太多，Follower太少的话会降低系统的可用性。

③如何判断集群是否可用？

Zk集群中有一半以上(大于一半)的机器正常工作的话，集群对外提供的服务就是可用的。如果一个Leader、100个Follower,没有Observer,那么就算有50台机器宕机，集群照样可用，假设现在采用1个Leader、10个Follower、90个Observer,只要集群里面有大于5台除了Observer的机器宕机，集群就会处于不可用状态，至少处理不了事务请求，因为事务需要投票，且需要一半以上的机器投票才能提供服务，而Observer不能投票。只要Leader+Follower中大于5台宕机，就处理不了事务请求了。

④如何集群为啥最好是奇数台？

因为假如集群中有三台机器，那么最多只允许1台机器宕机；而如果集群中有4台机器，也最多只允许1台机器宕机，所以4台机器和三台机器的效果一样，反而还浪费了一台机器，没必要，所以建议集群中机器数量为奇数。

⑤zk集群的服务器状态？

**LOOKING**：寻找Leader中。

**LEADING**: Leader状态，对应的节点为Leader。

**FOLLOWING**: Follower状态，对应的节点为Follower。

**OBSERVER**: Observer状态，对应的节点为Follower。

⑥zk集群中的脑裂现象？

什么是脑**裂现象**？

对于一个集群，通常多台机器会部署在不同机房，来提高这个集群的可用性。保证可用性的同时，会发生一种机房间网络线路故障，导致机房间网络不通，而集群被割裂成几个小集群。这时候子集群各自选主导致“脑裂”的情况。举例说明：比如现在有一个由 6 台服务器所组成的一个集群，部署在了 2 个机房，每个机房 3 台。正常情况下只有 1 个 leader，但是当两个机房中间网络断开的时候，每个机房的 3 台服务器都会认为另一个机房的 3 台服务器下线，而选出自己的 leader 并对外提供服务。若没有过半机制，当网络恢复的时候会发现有 2 个 leader。仿佛是 1 个大脑（leader）分散成了 2 个大脑，这就发生了脑裂现象。脑裂期间 2 个大脑都可能对外提供了服务，这将会带来数据一致性等问题。

或者由于假死导致脑裂。

什么是假死？

由于心跳超时（网络原因导致的）认为master死了，但其实master还存活着。由于假死会发起新的master选举，选举出一个新的master，但旧的master网络又通了，导致出现了两个master，有的客户端连接到旧得master，有的客户端连接到新得master。

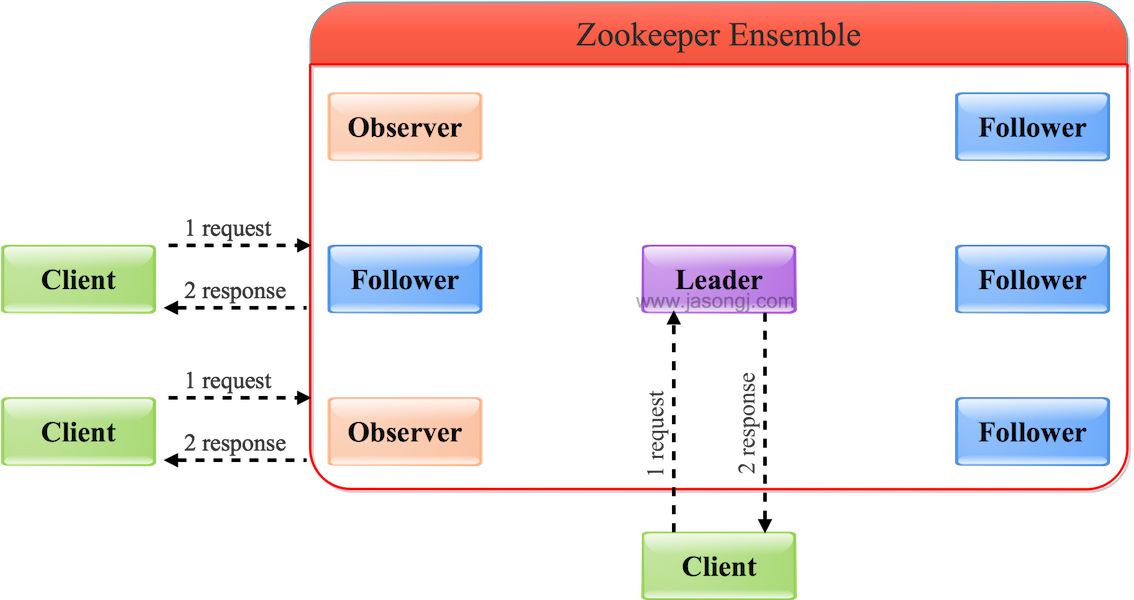
Zk是怎么防止脑裂现象的？

虽然机房之间不能进行网络通信，但是机房之间得zkserver还是可以相互通信的，ZooKeeper 的**过半机制**导致不可能产生 2 个 leader，因为少于等于一半是不可能产生 leader 的，这就很大程度上避免发生脑裂。

另外，zk还可以通过以下几种方式解决脑裂问题：

1. **Quorums (法定人数) 方式**: 比如3个节点的集群，Quorums = 2, 也就是说集群可以容忍1个节点失效，这时候还能选举出1个lead，集群还可用。比如4个节点的集群，它的Quorums = 3，Quorums要超过3，相当于集群的容忍度还是1，如果2个节点失效，那么整个集群还是无效的。**这是zookeeper防止"脑裂"默认采用的方法**。
2. **Redundant communications (冗余通信/添加心跳线)**：集群中采用多种通信方式，防止一种通信方式失效导致集群中的节点无法通信。
3. **Fencing (共享资源/启用磁盘锁) 方式**：比如能看到共享资源就表示在集群中，能够获得共享资源的锁的就是Leader，看不到共享资源的，就不在集群中。
4. **设置仲裁机制**:脑裂导致的后果最主要的原因就是从节点不知道该连接哪一台namenode，此时如果有一方来决定谁留下，谁放弃就最好了。因此出现了仲裁机制，比如提供一个参考的IP地址，当出现脑裂现象时，双方接收不到对方的心跳机制，但是能同时ping参考IP，如果有一方ping不通，那么表示该节点网络已经出现问题，则该节点需要自行退出争抢资源的行列，或者更好的方法是直接强制重启，这样能更好的释放曾经占有的共享资源，将服务的提供功能让给功能更全面的namenode节点。

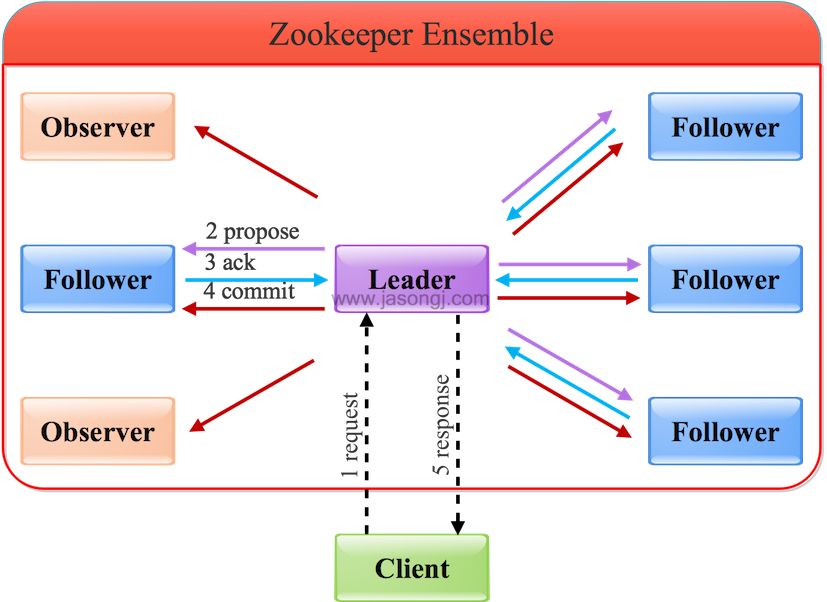
⑦zk集群的读操作流程



任何一种角色都可以直接处理度请求，从本地内存中读取数据并返回给客户端即可。

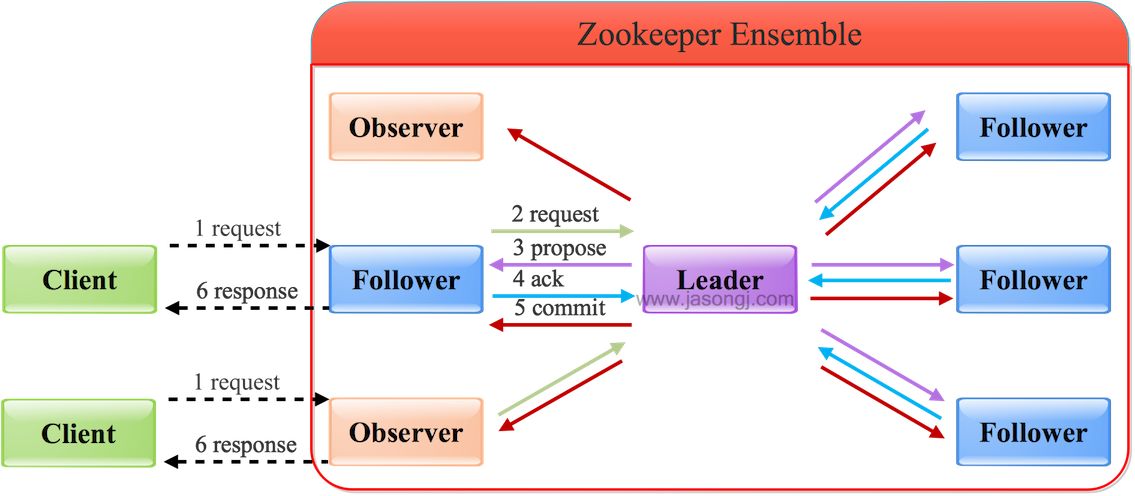
⑧zk集群的写操作流程？

情况一：直接向leader节点写



1. 客户端向Leader发起写请求;
2. Leader将写请求以Proposal的形式发给所有Follower并等待ACK，因为Observer无权投票，所以无需向Observer发送写请求等待ACK;
3. Follower收到Leader的Proposal后返回ACK;
4. Leader得到过半数的ACK(Leader对自己默认有一个ACK)后向所有的Follower和Observer发送Commit同步数据(虽然Observer无权投票，但仍须同步Leader的数据，从而在处理度请求时可以返回尽可能新的数据);
5. Leader将处理结果返回给客户端。

情况二：向Follower/Observer节点写



Follower和Observer都可以接收写请求，但是不能直接处理，而是需要将写请求转发给Leader处理，除了多了这一步之外，其他流程与直接写Leader没有任何区别

1. Zookeeper的CP原则
2. Zookeeper的节点类型或者节点的存储方式

该树形结构中每个节点被称为znode有两种节点，持久化节点和永久节点。

1. ZAB协议

①ZAB协议介绍

ZAB是Zookeeper Atomic Broadcast的缩写，即原子广播协议，该协议是**给zk专门设计的一种保证数据一致性的核心算法**。基于该协议，zk实现了一种主从模式的系统架构来保持急群众各个副本之间的数据一致性。

根据zab协议，所有的写请求都必须通过Leader完成，Leader写入本地日志后再复制到所有的Follower节点。

一旦Leader节点无法工作，ZAB协议能够自动从Follower节点中重新选出一个合适的替代者，即新的Leader，该过程即为选举过程，是ZAB协议中最为重要的复杂的过程。

②ZAB的协议模式

ZAB协议包括了两种基本的模式，分别是**崩溃恢复和消息广播**。

·**崩溃恢复**：当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的 Leader 服务器。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出恢复模式。其中，所谓的状态同步是指数据同步，用来保证集群中存在过半的机器能够和 Leader 服务器的数据状态保持一致。

·**消息广播**：当集群中已经有过半的 Follower 服务器完成了和 Leader 服务器的状态同步，那么整个服务框架就可以进入消息广播模式了。 当一台同样遵守 ZAB 协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播，那么新加入的服务器就会自觉地进入数据恢复模式：找到 Leader 所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。

③ZAB的Leader选举过程

**情况一，服务器启动时期的Leader选举**

若进行Leader选举，则至少需要2台机器，两台的高可用性会差一些，如果Leader宕机，就剩下一台，自己没办法选举，所以一般最低选择使用3台机器。

在集群初始化阶段，当有一台服务器Server1启动时，其单独无法进行和完成Leader选举，当第二台服务器Server2启动时，此时两台机器可以相互通信，每台机器都试图找到Leader，于是进入选举过程，选举过程驶如下：

1. **每个Server发出一个投票。**由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和zxid，使用(myid,zxid)来表示，此时Server1的投票为(1, 0),Server2的投票为(2,0)，然后各自将这个投票发给集群中其他机器。
2. **接受来自各个服务器的投票**。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否时本轮投票、是否来自LOOKONG状态的服务器。
3. **处理投票**。针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下：

**优先检查zxid，zxid比较大的服务器优先作为Leader**，根据数据最新原则，保证了数据的完整性。

**如果zxid相同，那就比较myid，myid较大的服务器作为Leader**。

对于Server1而言，他的投票是(1,0),接受Server2的投票为(2,0),首先会比较两者的zxid,再比较myid，此时Server2的myid最大，于是Server1更新自己的投票为(2,0);

对于Server2而言，其不需要更新自己的投票，还是(2,0)，因为Server2与Server1的zxid相等，但是Server2的myid大于Server1的myid。比较完之后，再次向集群中所有的机器发出上一次投票信息集合，即(2,0)。

1. **统计投票**。每次投票后，服务器都会统计投票信息，判断是否已经有过半的(能投票的，即非Observer的)机器接受到相同的投票信息。对于Server1、Server2而言，都统计出了集群中已经有两台机器接受了(2,0)的投票信息，此时变认为已经选出了Leader.
2. **改变服务器状态**。一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower,那么就变更为FOLLOWING,如果是Leader就变更为LEADING。

**情况二，服务器运行时期的Leader选举**

在zk运行期间，Leade与非Leader服务器各司其职，即便当有**非Leader服务器宕机或者新加入，此时也不会影响Leader**，但是一旦Leader挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举。其过程和启动时期的Leader选举过程基本一致。

假设正在运行的Server1、Server2、Server3三台机器中，Server2是Leader，此时Leader挂掉了，便开始Leader选举。选举过程如下：

1. **变更状态**。Leader挂了之后，余下非Obsever服务器都会将自己的服务器状态变成LOOKING,然后开始进入接下来的Leader选举过程。
2. **每个Server会发出一个投票**。在运行期间，每个服务器上的zxid可能不同，假设Server1的zxid为123，Server3的zxid为122，在第一轮投票中，Server1和Server3都会投自己，产生投票(1,123)、(3,122)，然后各自将投票发送给集群中所有机器。
3. **接受来自各个服务器的投票**。与启动过程相同。
4. **处理投票**。与启动过程相同。此时，Server1将会成为Leader。
5. **统计投票**。与启动过程相同。
6. **改变服务器的状态**。与启动时过程相同。

④ZAB的Leader选举算法

在3.4.0后的zk版本只保留了**基于TCP的FastLeaderElection**，其他三种已经被弃用。

当集群中已存在Leader时，在往集群中添加新的服务器，该机器会试图去选举Leader，此时会被告知当前服务器的Leader信息，对于该机器而言，仅仅需要和Leader机器建立起连接，并进行状态同步即可。

参考文档：

<https://dbaplus.cn/news-141-1875-1.html>

1. Redis
2. Redis？
3. redis客户端

①jedis

采用**同步阻塞式IO**，**采用线程池时是线程安全的**。简单、灵活，api全面，缺点是某些redis高级功能需要自己封装。

②redisTemplete

是基于某个具体实现的再封装，比如springboot1.x时，具体实现时jedis；而到了springboot2.x时，具体实现变成了lettuce.封装的好处就是基于springboot自动装配原理，使得整合redis比较简单，隐藏了具体实现，调用更方便，但是**jedis的执行效率要比redisTemplete高10-30倍。**

③redisson

作为redis的**分布式客户端**，**基于netty采用异步非阻塞IO，是线程安全的**。提供了很多redis的分布式操作和高级功能，例如集群、哨兵、管道等，缺点是api抽象，学习成本高。

④lettuce

1. 分布式锁
2. 基于数据库的分布式锁
3. 基于redis的setnx实现分布式锁

①原理

通过setnx命令将名为key的锁设置到redis中，如果该key存在就插入失败，返回0，表示抢锁失败，否则插入成功，返回1，表示抢锁成功。抢锁成功的线程执行完任务，需要释放锁，以便其他线程可以进入。

②基于redis实现分布式锁的过程会遇到以下问题

问题1：**锁超时**。

当得到锁的线程再执行任务的过程中挂掉，来不及释放锁，这样会形成死锁，及这个key的锁一直存在于redis中，其他线程就会一直set不成功。所以，需要**设置超时时间**。

问题2：**必须保证set和设置超时时间两个操作的原子性**。

如果在设置超时时间时设置失败，同样会导致死锁。所以需要保证set和设置超时时间的原子性。可以**使用高版本的redis客户端工具，调用它的多参数set方法**。

问题3：**执行完任务需要删除锁**。

加入设置了超时时间为60s,但是我执行线程只换了3秒，如果执行完线程之后不主动删除锁，那么会导致其他线程白白等待57s,这样严重影响了效率。

问题4：**在并发情况下，可能会删除其他人的锁**。

比如，在多并发情况下，由于线程A的业务处理时间时间很长，锁自动过期了，但是线程A还是在处理。线程B此时获取了锁，在线程B执行的过程中，线程A业务处理完，然后执行删除所操作，线程C此时又能获取到锁了，导致达不到锁的效果。可以把值指定为uuid，**在删除锁时，判断锁的值是否等于当前线程生成的uuid**。同样也要保证判断和删除操作的原子性，我们此时需要**使用LUA脚本来保证判断和删除操作的原子性**。

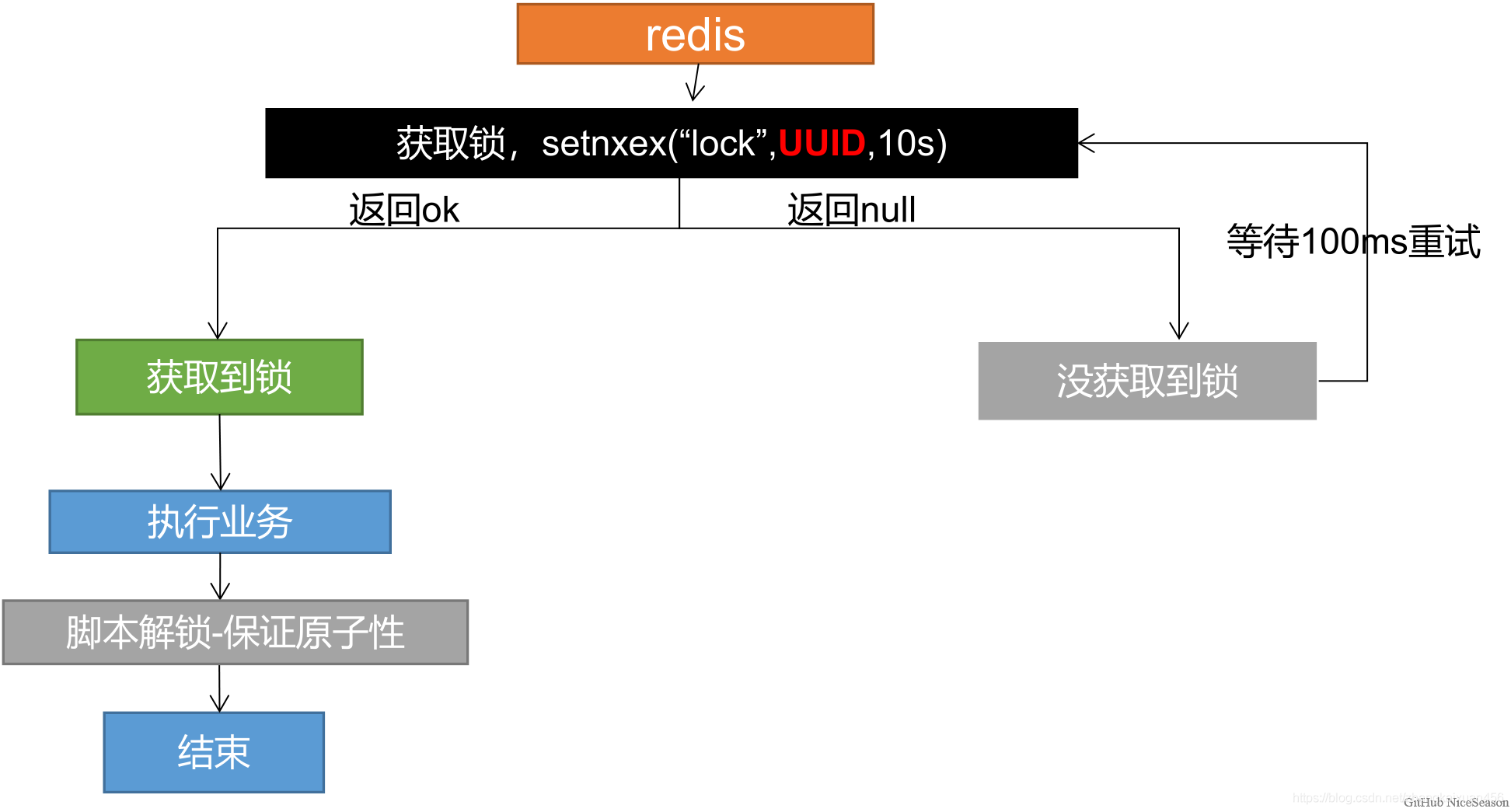
问题5：**续约问题**。

在多并发情况下，由于线程A的业务处理时间时间很长，锁自动过期了，但是线程A还是在处理，而线程B此时获取了锁，造成了同时有两个线程持有锁。这个问题归结为“续约问题”，即A线程每执行完时应该过期时间续约，执行完成才能释放锁。可以**让获得锁的线程开启一个守护线程，用来给快要过期的锁续约，或者通过Redisson的”看门狗”来实现**。问题4也属于续约问题。

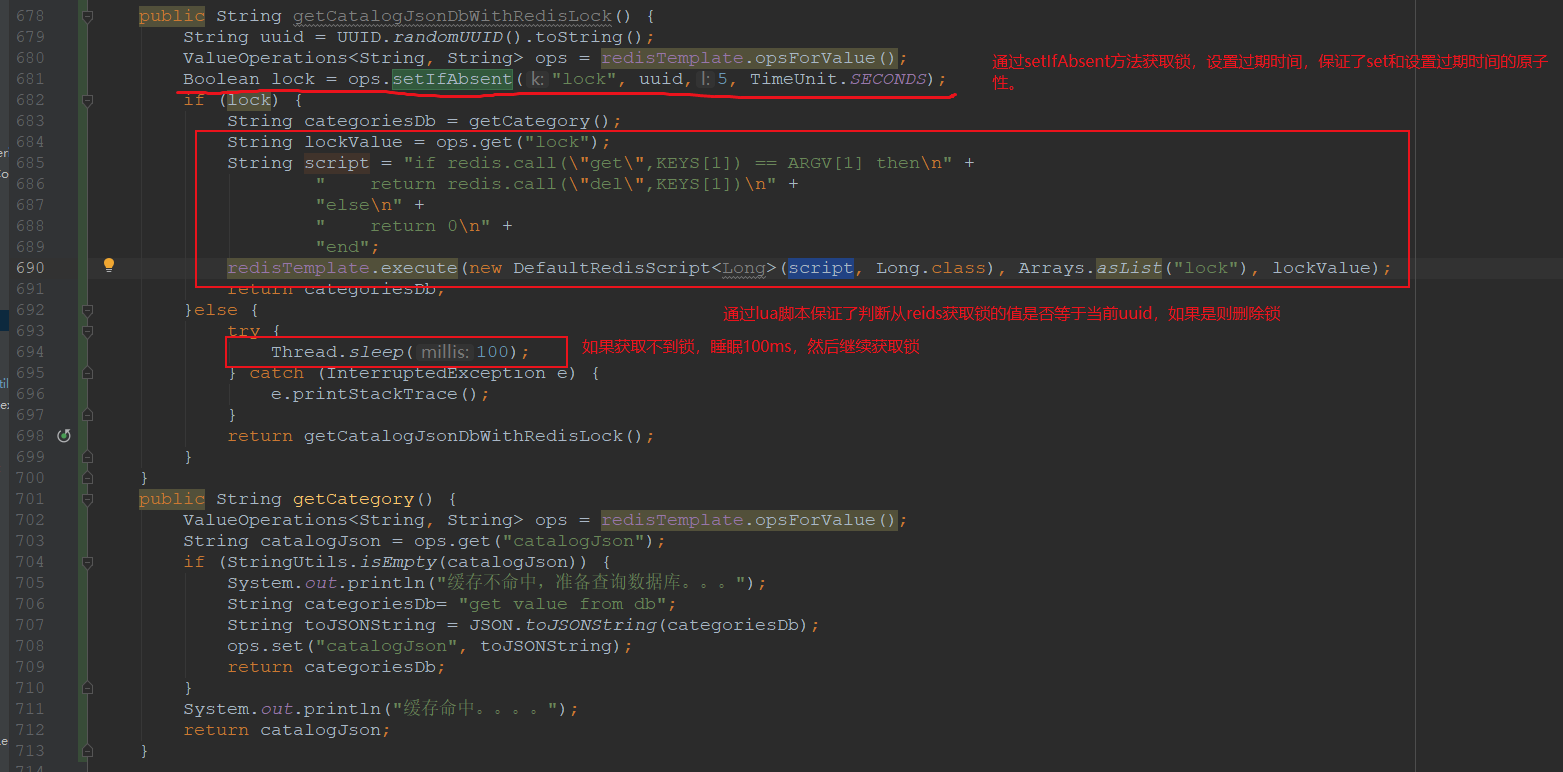
问题6：**集群同步延时问题**。

当项目需要实现rdis高可用时，就得部署多台redis，因为单机的redis，一旦redis挂掉，整个分布式锁就不没用了。在集群场景下，如果线程A在master拿到了锁，在没有把数据同步到slave时，master挂掉了。B线程再拿锁会从slave拿锁，此时线程B能获取到锁，又出现了多个线程同时获取到锁的情况。

最终逻辑如下图：



代码参考如下：



上述代码没有解决续约问题和集群的同步延时问题。

Redis

<https://www.jb51.net/article/125931.htm>

<https://blog.csdn.net/SharingOfficer/article/details/122117251>

https://blog.csdn.net/qq\_29860591/article/details/113243441?utm\_medium=distribute.pc\_aggpage\_search\_result.none-task-blog-2~aggregatepage~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-3-113243441.pc\_agg\_new\_rank&utm\_term=redis%E4%B8%BB%E4%BB%8E%E5%90%8C%E6%AD%A5%E5%BB%B6%E8%BF%9F%E5%AF%BC%E8%87%B4%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E9%97%AE%E9%A2%98&spm=1000.2123.3001.4430

<https://blog.csdn.net/zjcjava/article/details/103025997>

https://blog.csdn.net/qq\_41653753/article/details/94979706?spm=1001.2101.3001.6661.1&utm\_medium=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.pc\_relevant\_default&depth\_1-utm\_source=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.pc\_relevant\_default&utm\_relevant\_index=1

https://blog.csdn.net/Tangwp1214/article/details/102790497?spm=1001.2101.3001.6650.5&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-5.pc\_relevant\_aa&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-5.pc\_relevant\_aa&utm\_relevant\_index=10

1. 基于zk实现分布式锁

<https://www.php.cn/faq/466231.html>

<https://www.cnblogs.com/chenkeyu/p/8047811.html>https://zhuanlan.zhihu.com/p/337052900