# Dubbo

## Dubbo是什么

Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。

其核心部分包含:

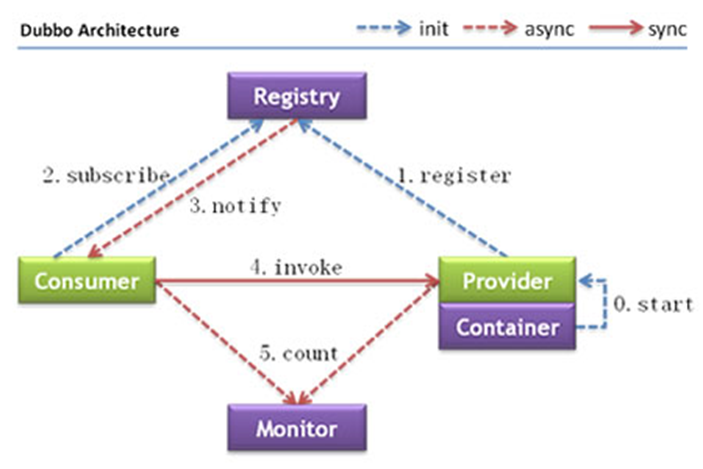
1. 透明化的远程通讯: 提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。

2. 软负载均衡及集群容错: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

3. 服务自动注册与发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## Dubbo的架构

### 2.1节点角色：



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: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次调和调用时间的监控中心。

Container: 服务运行容器。

### 2.2调用关系：

1.服务容器负责启动，加载，运行服务提供者。

2.服务提供者在启动时，向注册中心注册自己提供的服务。

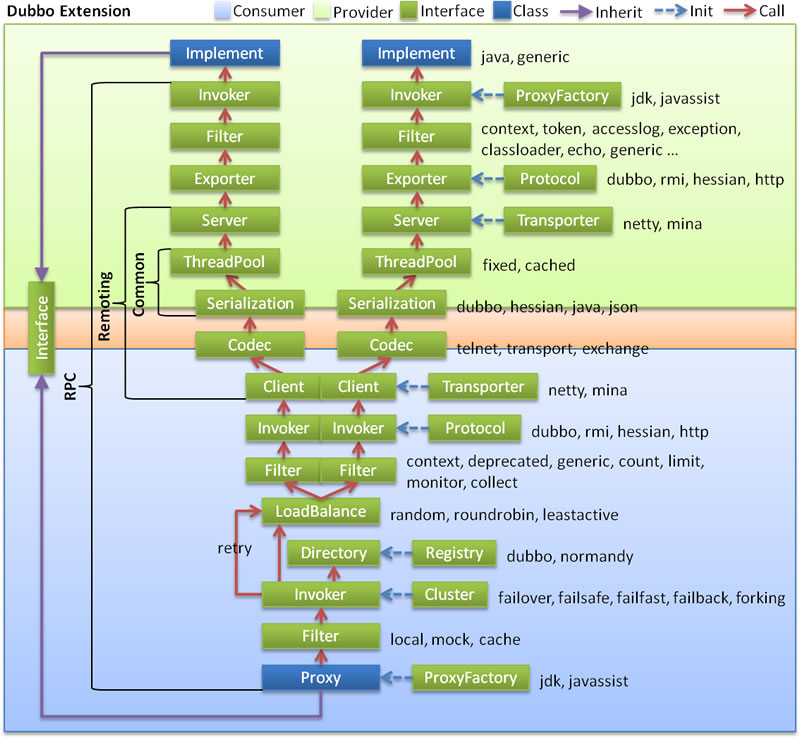
3.服务消费者在启动时，向注册中心订阅自己所需的服务。

4.注册中心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。

5.服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

6.服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

将上面抽象的调用流程图展开，详细如图所示：



### 2.5连通关系：

（1）

注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。

监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示。

服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销。

服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销。

注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外。

注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者。

注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表。

注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

（2）

监控中心宕掉不影响使用，只是丢失部分采样数据。

数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务。

注册中心对等集群，任意一台宕掉后，将自动切换到另一台。

注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯。

服务提供者无状态，任意一台宕掉后，不影响使用。

服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复。

（3）

注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心。

服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者。

### 2.3协议支持

Dubbo支持多种协议，如下所示：

Dubbo协议 Hessian协议

HTTP协议 RMI协议

WebService协议

Thrift协议 Memcached协议 Redis协议

在通信过程中，不同的服务等级一般对应着不同的服务质量，那么选择合适的协议便是一件非常重要的事情。你可以根据你应用的创建来选择。例如，使用RMI协议，一般会受到防火墙的限制，所以对于外部与内部进行通信的场景，就不要使用RMI协议，而是基于HTTP协议或者Hessian协议。

默认使用Dubbo协议

连接个数：单连接

连接方式：长连接

传输协议：TCP

传输方式：NIO异步传输

序列化：Hessian二进制序列化

适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要使用dubbo协议传输大文件或超大字符串

使用场景：常规远程服务方法调用

从上面的适用范围总结，dubbo适合小数据量大并发的服务调用，以及消费者机器远大于生产者机器数的情况，不适合传输大数据量的服务比如文件、视频等，除非请求量很低。

### 2.4负载均衡

dubbo提供4种负载均衡方式:

Random，随机，按权重配置随机概率，调用量越大分布越均匀，默认是这种方式。

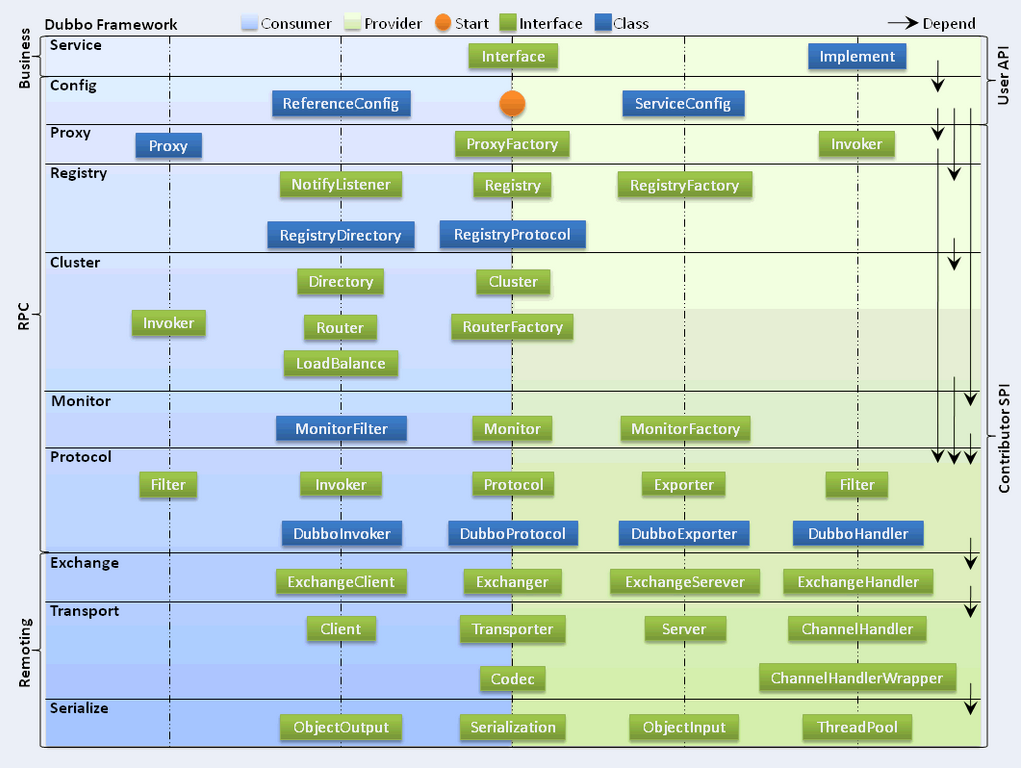
RoundRobin，轮询，按权重设置轮询比例，如果存在比较慢的机器容易在这台机器的请求阻塞较多。

LeastActive，最少活跃调用数，不支持权重，只能根据自动识别的活跃数分配，不能灵活调配。

ConsistentHash，一致性hash，对相同参数的请求路由到一个服务提供者上，如果有类似灰度发布需求可采用。

dubbo的负载均衡机制是在客户端调用时通过内存中的服务方信息及配置的负责均衡策略选择，如果对自己系统没有一个全面认知，建议先采用random方式。

## Dubbo的原理



框架分层架构中，各个层次的设计要点：

**1、服务接口层（Service）：**该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

**2、配置层（Config）**：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。

**3、服务代理层（Proxy）：**服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。

**4、服务注册层（Registry）：**封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。

**5、集群层（Cluster）：**封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。

**6、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。

**7、远程调用层（Protocol）：**封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。

**8、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。

**9、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。

**10、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。

上述各层之间关系的描述，如下所示：

1、在RPC中，Protocol是核心层，也就是只要有Protocol + Invoker + Exporter就可以完成非透明的RPC调用，然后在Invoker的主过程上Filter拦截点。

2、图中的Consumer和Provider是抽象概念，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，不用Client和Server的原因是Dubbo在很多场景下都使用Provider、Consumer、Registry、Monitor划分逻辑拓普节点，保持统一概念。

3、而Cluster是外围概念，所以Cluster的目的是将多个Invoker伪装成一个Invoker，这样其它人只要关注Protocol层Invoker即可，加上Cluster或者去掉Cluster对其它层都不会造成影响，因为只有一个提供者时，是不需要Cluster的。

4、Proxy层封装了所有接口的透明化代理，而在其它层都以Invoker为中心，只有到了暴露给用户使用时，才用Proxy将Invoker转成接口，或将接口实现转成Invoker，也就是去掉Proxy层RPC是可以Run的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。

5、而Remoting实现是Dubbo协议的实现，如果你选择RMI协议，整个Remoting都不会用上，Remoting内部再划为Transport传输层和Exchange信息交换层，Transport层只负责单向消息传输，是对Mina、Netty、Grizzly的抽象，它也可以扩展UDP传输，而Exchange层是在传输层之上封装了Request-Response语义。

6、Registry和Monitor实际上不算一层，而是一个独立的节点，只是为了全局概览，用层的方式画在一起。

### 3.1Dubbo源码模块图

Dubbo以包结构来组织各个模块，各个模块及其关系，如图所示：

可以通过Dubbo的代码（使用Maven管理）组织，与上面的模块进行比较。简单说明各个包的情况：

dubbo-common 公共逻辑模块，包括Util类和通用模型。

dubbo-remoting 远程通讯模块，相当于Dubbo协议的实现，如果RPC用RMI协议则不需要使用此包。

dubbo-rpc 远程调用模块，抽象各种协议，以及动态代理，只包含一对一的调用，不关心集群的管理。

dubbo-cluster 集群模块，将多个服务提供方伪装为一个提供方，包括：负载均衡、容错、路由等，集群的地址列表可以是静态配置的，也可以是由注册中心下发。

dubbo-registry 注册中心模块，基于注册中心下发地址的集群方式，以及对各种注册中心的抽象。

dubbo-monitor 监控模块，统计服务调用次数，调用时间的，调用链跟踪的服务。

dubbo-config 配置模块，是Dubbo对外的API，用户通过Config使用Dubbo，隐藏Dubbo所有细节。

dubbo-container 容器模块，是一个Standalone的容器，以简单的Main加载Spring启动，因为服务通常不需要Tomcat/JBoss等Web容器的特性，没必要用Web容器去加载服务。

### 3.2Dubbo初始化过程

和Spring中启动过程类似，spring启动时，将bean装载进容器中的时候，首先要解析bean。所以dubbo也是先读配置文件解析服务。

**解析服务：**

1）、基于dubbo.jar内的Meta-inf/spring.handlers配置，spring在遇到dubbo名称空间时，会回调DubboNamespaceHandler类。

2）、所有的dubbo标签，都统一用DubboBeanDefinitionParser进行解析，基于一对一属性映射，将XML标签解析为Bean对象。

源码截图：

在ServiceConfig.export 或者ReferenceConfig.get 初始化时，将Bean对象转会为url格式，将所以Bean属性转成url的参数。

然后将URL传给Protocol扩展点，基于扩展点的Adaptive机制，根据URL的协议头，进行不同协议的服务暴露和引用。

**暴露服务：**

a、 只暴露服务端口

在没有使用注册中心的情况，这种情况一般适用在开发环境下，服务的调用这和提供在同一个IP上，只需要打开服务的端口即可。

即，当配置 or

ServiceConfig解析出的URL的格式为：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptiver机制，通过URL的“dubbo：//”协议头识别，直接调用DubboProtocol的export（）方法，打开服务端口。

b、向注册中心暴露服务：

和上一种的区别：需要将服务的IP和端口一同暴露给注册中心。

ServiceConfig解析出的url格式为：

registry://registry-host/com.alibaba.dubbo.registry.RegistryService?export=URL.encode(“dubbo://service-host/com.xxx.TxxService?version=1.0.0”)

基于扩展点的Adaptive机制，通过URL的“registry：//”协议头识别，调用RegistryProtocol的export方法，将export参数中的提供者URL先注册到注册中心，再重新传给Protocol扩展点进行暴露：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

**引用服务：**

a、直接引用服务：

在没有注册中心的，直连提供者情况下，

ReferenceConfig解析出的URL格式为：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过url的“dubbo：//”协议头识别，直接调用DubboProtocol的refer方法，返回提供者引用。

b、从注册中心发现引用服务：

此时，ReferenceConfig解析出的URL的格式为： registry://registry-host/com.alibaba.dubbo.registry.RegistryService?refer=URL.encode(“consumer://consumer-host/com.foo.FooService?version=1.0.0”)

基于扩展点的Apaptive机制，通过URL的“registry：//”协议头识别，就会调用RegistryProtocol的refer方法，基于refer参数总的条件，查询提供者URL，如：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

### 3.3Dubbo调用过程

1.client一个线程调用远程接口，生成一个唯一的ID（比如一段随机字符串，UUID等），Dubbo是使用AtomicLong从0开始累计数字的

2.将打包的方法调用信息（如调用的接口名称，方法名称，参数值列表等），和处理结果的回调对象callback，全部封装在一起，组成一个对象object

3.向专门存放调用信息的全局ConcurrentHashMap里面put(ID, object)

4.将ID和打包的方法调用信息封装成一对象connRequest，使用IoSession.write(connRequest)异步发送出去

5.当前线程再使用callback的get()方法试图获取远程返回的结果，在get()内部，则使用synchronized获取回调对象callback的锁， 再先检测是否已经获取到结果，如果没有，然后调用callback的wait()方法，释放callback上的锁，让当前线程处于等待状态。

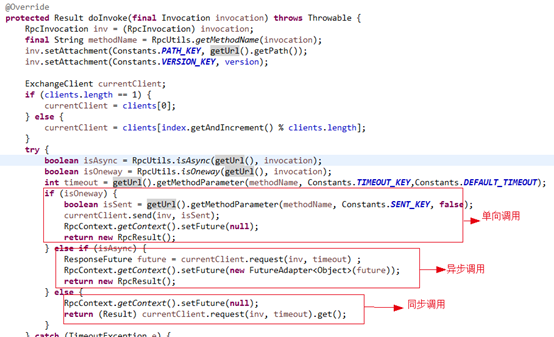
6.服务端接收到请求并处理后，将结果（此结果中包含了前面的ID，即回传）发送给客户端，客户端socket连接上专门监听消息的线程收到消息，分析结果，取到ID，再从前面的ConcurrentHashMap里面get(ID)，从而找到callback，将方法调用结果设置到callback对象里。

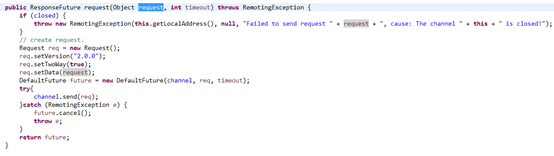
7.监听线程接着使用synchronized获取回调对象callback的锁（因为前面调用过wait()，那个线程已释放callback的锁了），再notifyAll()，唤醒前面处于等待状态的线程继续执行（callback的get()方法继续执行就能拿到调用结果了），至此，整个过程结束。

## 4、Dubbo调用过程

消费端调用远程服务接口时，使用上和调用普通的java接口是没有任何区别，但是服务消费者和提供者是跨JVM和主机的，客户端如何封装请求让服务端理解请求并且解析服务端返回的接口调用结果，服务端如何解析客户端的请求并且向客户端返回调用结果，这些框架是如何实现的，下面就来看下这部分的代码。   
消费端调用提供端服务的过程要执行下面几个步骤：   
1. 消费端触发请求   
2. 消费端请求编码   
3. 提供端请求解码   
4. 提供端处理请求   
5. 提供端响应结果编码   
6. 消费端响应结果解码

**消费端触发请求**

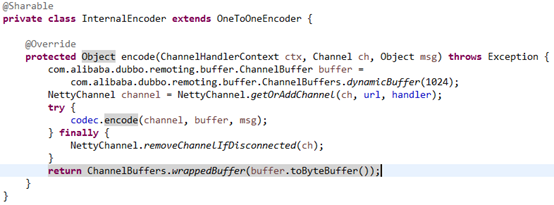
在消费者初始化的时候，会生成一个消费者代理注册到容器中，该代理回调中持有一个MockClusterInvoker实例，消费调用服务接口时它的invoke会被调用，此时会构建一个RpcInvocation对象，把服务接口的method对象和参数放到RpcInvocation对象中，作为MockClusterInvoker.invoke方法的参数，在这个invoke方法中，判断请求是否需要mock，是否配置了mock属性，是强制mock还是失败后mock，关于mock这里先不详细展开，这里只看下核心流程。   
MockClusterInvoker.invoke会调用FailfastClusterInvoker.invoke，大系统中为了服务高可用同一个服务一般会有多个应用服务器提供，要先挑选一个提供者提供服务。在服务接口消费者初始化时，接口方法和提供者Invoker对应关系保存在RegistryDirectory的methodInvokerMap中，通过调用的方法名称（或方法名称+第一个参数）改方法对应的提供者invoker列表，如注册中心设置了路由规则，对这些invoker根据路由规则进行过滤。   
com.alibaba.dubbo.registry.integration.RegistryDirectory.doList(Invocation)   
  
com.alibaba.dubbo.rpc.cluster.directory.AbstractDirectory.list(Invocation)   
   
读取到所有符合条件的服务提供者invoker之后，由LoadBalance组件执行负载均衡，从中挑选一个invoker进行调用，框架内置支持的负载均衡算法包括random（随机）、roundrobin（R-R循环）、leastactive（最不活跃）、consistenthash（一致性hash），应用可配置，默认random。   
methodInvokerMap保存的是持有DubboInvoker（dubbo协议）实例的InvokerDelegete对象，是Invoker-Filter链的头部，先激活Filter连然后最终调到DubboInvoker.invoke(RpcInvocation)，此时远程调用分三种类型：   
1. 单向调用，无需获取关注调用结果的，无需等待接口返回结果，注意调用结果不要单纯跟返回值混淆了，异常也是调用结果。   
2. 异步调用，需要关注返回结果，但是不会同步等待接口调用结束，会异步的获取返回返回结果，这种情况给调用者返回一个Future，但是不同步等待Future.get返回调用结果   
3. 同步调用，需要同步等待服务调用结束获取调用结果，给调用者返回一个Future并且Future.get等待结果，此时接口调用线程会挂起等待响应。   
  
我们大部分使用场景都是同步调用，所以主要看一下同步调用。如果使用者配置了多个connections按顺序选择一个ExchangeClient和服务器通信，同步调用时调用HeaderExchangeClient.request->HeaderExchangeChannel.request。

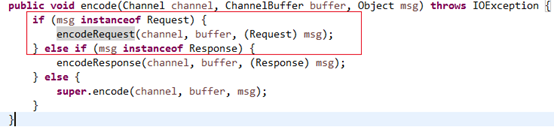
com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeChannel.request(Object, int)   
  
这里的request参数是RpcInvocation对象，包含调用的方法、参数等信息，timeout参数是接口超时时间，把这些信息封装在Request对象中，调用channel.send，这个channel对象就是和服务端打交道的NettyClient实例，NettyClient.send调用NettyChannel.send。

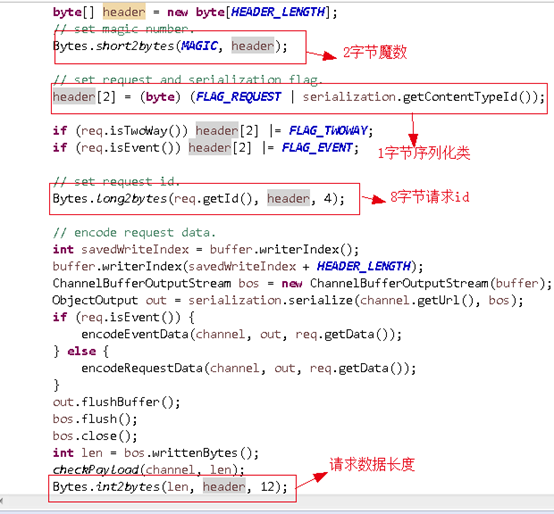
com.alibaba.dubbo.remoting.transport.netty.NettyChannel.send(Object, boolean)   
  
这里的sent参数决定是否等待请求消息发出，sent=true 等待消息发出，消息发送失败将抛出异常，sent=false 不等待消息发出，将消息放入IO队列，即刻返回。默认情况下都是false。NettyChannel中有channel属性，这个channel是Netty框架中的组件，负责客户端和服务端链路上的消息传递，channel.write把请求消息写入，这里的message是上面封装的Request对象。这里的IO模型是非阻塞的，线程不用同步等待所有消息写完，而是直接返回。调用Netty框架的IO事件之后会触发Netty框架的IO事件处理链。

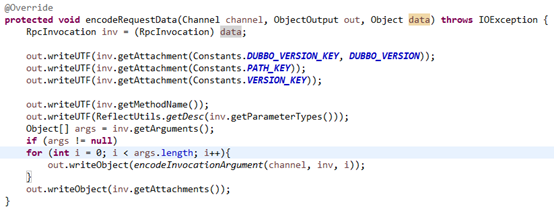
**消费端请求编码**

在消费者初始化创建NettyClient时了解到了，NettyClient添加了三个事件处理器组成处理器链：NettyCodecAdapter.decoder->NettyCodecAdapter.encoder->NettyHandler，其中NettyCodecAdapter.encoder下行事件处理器（实现了ChannelDownstreamHandler接口），NettyCodecAdapter. decoder是上行事件处理器（实现了ChannelUpstreamHandler接口），NettyHandler是上行事件+下行时间处理器（同时实现了ChannelUpstreamHandler和ChannelDownstreamHandler接口）。channel.write在Netty框架中是一个下行事件，所以NettyCodecAdapter.encoder和NettyHandler处理器会被回调，下行事件的事件处理器调用顺序是从后到前，即后添加的处理器先执行。   
NettyHandler没有对请求消息做任何加工，只是触发dubbo框架的一些回调，这些回调里面没有做任何核心的事情，

com.alibaba.dubbo.remoting.transport.netty.NettyHandler.writeRequested(ChannelHandlerContext, MessageEvent)   
  
encoder顾名思义就是编码器，它的主要工作就是把数据按照客户端-服务端的约定协议对请求信息和返回结果进行编码。看下它的encode方法：   
  
下行事件触发之后依次调用handleDownstream->doEncode->encode，在encode中对Request对象进行编码。这个msg参数就是上面被write的Request对象，这里的Codec2组件是DubboCountCodec实现，DubboCountCodec.encode调用DubboCodec.Encode

com.alibaba.dubbo.remoting.exchange.codec.ExchangeCodec.encode(Channel, ChannelBuffer, Object)   
  
根据协议，消息中写入16个字节的消息头：   
1、1-2字节，固定的魔数   
2、第3个字节，第8位存储数据类型是请求数据还是响应数据，其它7位存储序列化类型，约定和服务端的序列化-反序列化协议   
3、5-12个字节，请求id   
4、13-16个字节，请求数据长度

com.alibaba.dubbo.remoting.exchange.codec.ExchangeCodec.encodeRequest(Channel, ChannelBuffer, Request)   
  
从URL中查找序列化扩展点名称，加载序列化组件把请求对象序列化成二进制。消费端和提供端的序列化反序列化协议要配套，所以这个序列化协议一般是在提供端指定的，指定的协议类型会在提供者和消费者初始化的时候写入到URL对象中，框架中默认的序列化协议是hessian2。消息体数据包含dubbo版本号、接口名称、接口版本、方法名称、参数类型列表、参数、附加信息，把它们按顺序依次序列化，数据写入到类型为ChannelBuffer的buffer参数中，然后把ChannelBuffer封装成Netty框架的org.jboss.netty.buffer.ChannelBuffer。如果参数中有回调接口，还需要在消费端启动端口监听提供端的回调，这里不展开。

com.alibaba.dubbo.rpc.protocol.dubbo.DubboCodec.encodeRequestData(Channel, ObjectOutput, Object)   
  
然后把封装好的ChannelBuffer写到链路发送到服务端，这里消费端前半部分的工作就完成，接下来目光要转移到服务端。

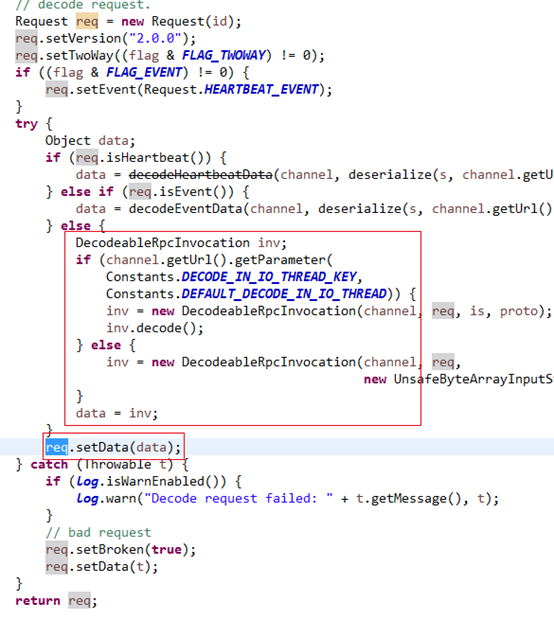
org.jboss.netty.handler.codec.oneone.OneToOneEncoder.doEncode(ChannelHandlerContext, MessageEvent)   

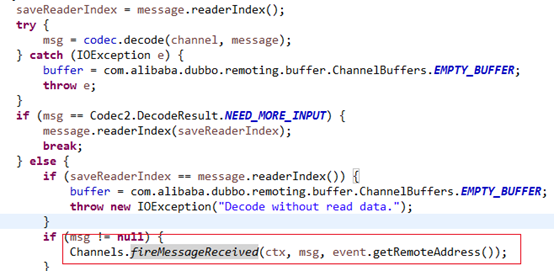

**提供端请求解码**

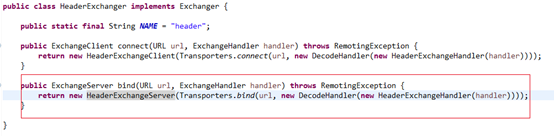
在看提供端初始化代码的时候看到，框架在创建NettyServer时，也会创建netty框架的IO事件处理器链：NettyCodecAdapter.decoder->NettyCodecAdapter.encoder->NettyHandler

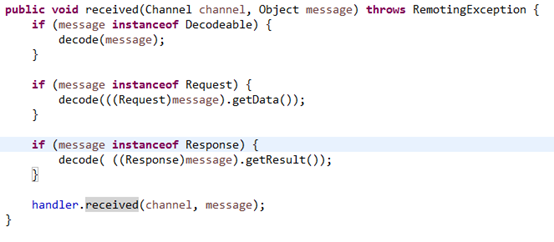
com.alibaba.dubbo.remoting.transport.netty.NettyServer.doOpen()   
  
客户端发送数据到服务端时会触发服务端的上行IO事件并且启动处理器回调，NettyCodecAdapter.decoder和NettyHandler是上行事件处理器，上行事件处理器调用顺序是从前到后执行，即先添加的处理器先执行，所以先触发NettyCodecAdapter.decoder再触发NettyHandler。   
由NettyCodecAdapter.decoder对请求进行解码，把消息翻译成提供端可理解的，上行事件调用decoder的handleUpstream->messageReceived

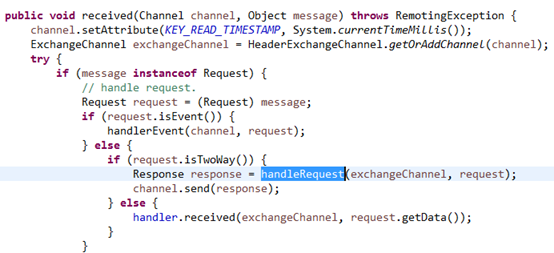
com.alibaba.dubbo.remoting.transport.netty.NettyCodecAdapter.InternalDecoder.messageReceived(ChannelHandlerContext, MessageEvent)   
  
com.alibaba.dubbo.remoting.exchange.codec.ExchangeCodec.decode(Channel, ChannelBuffer, int, byte[])   
  
把数据读取到ChannelBuffer之后扔给Codec2组件进行解码处理，这里有个半包传输处理，因为这里使用的是非阻塞式的IO模型，非阻塞IO的特点是线程的读取数据是事件触发式，是由一个Selector组件轮询准备就绪的IO事件，发现准备就绪的事件之后通知线程读取，这种模式的好处是可以极大的优化线程模型，只需少数几个线程处理所有客户端和服务端连接，而阻塞IO需要线程和连接要一对一，但是非阻塞IO远高于阻塞式IO，不像阻塞式IO读写数据时只有数据读完或者超时才会返回，这样能保证读到的数据肯定是完整，而非阻塞模式方法返回之后可能只读到一部分数据，框架的处理是在解析消息时检查消息的长度确定是否有完整的数据，如果数据不完整返回NEED\_MORE\_INPUT，保存当前解析的位置等待链路的下次IO事件，在下次IO事件到达时从上次保存的位置开始解析。   
读取到完整的数据之后解析数据头，读取魔数、序列化类型、以及请求id，读取第3个字节判断改数据是消费端请求数据还是提供端响应数据（因为消费端和提供端解码器代码是共用的），并且从1-7位从读出序列化类型，并且根据此序列化类型加载序列化组件对消息进行反序列化按顺序读取消费端写入的dubbo版本号、接口名称、接口版本、方法名称、参数类型列表、参数、附加信息，写入DecodeableRpcInvocation对象对应的属性中。

com.alibaba.dubbo.rpc.protocol.dubbo.DubboCodec.decodeBody(Channel, InputStream, byte[])   
  
com.alibaba.dubbo.rpc.protocol.dubbo.DecodeableRpcInvocation.decode(Channel, InputStream)   
  
创建一个Request对象，把DecodeableRpcInvocation对象对象设置到Request对象的data属性中。   
com.alibaba.dubbo.rpc.protocol.dubbo.DubboCodec.decodeBody(Channel, InputStream, byte[])   
  
解码完成之后，激活下一个处理器的messageReceived事件，并且把解码后的对象封装在MessageEvent中。

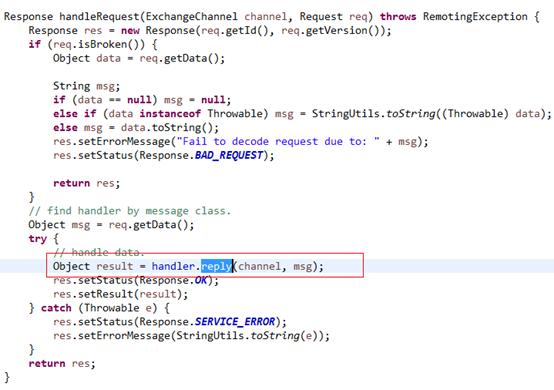
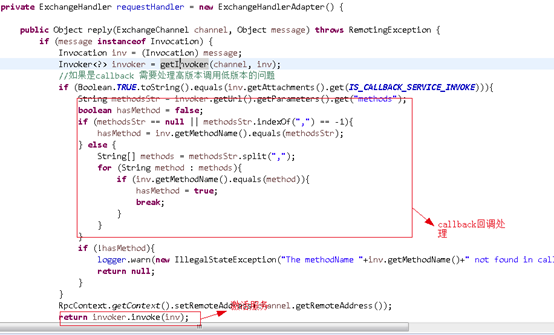
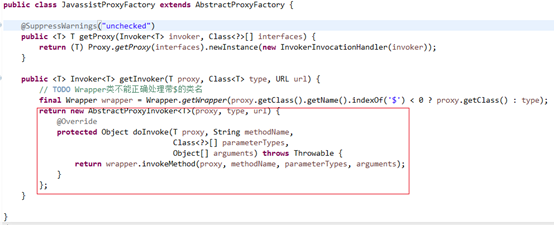
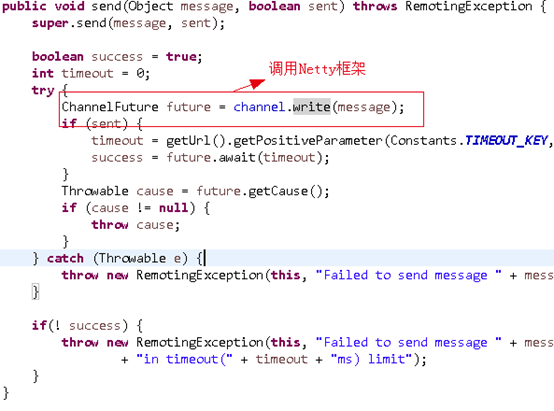
com.alibaba.dubbo.remoting.transport.netty.NettyCodecAdapter.InternalDecoder.messageReceived(ChannelHandlerContext, MessageEvent)   
  
org.jboss.netty.channel.Channels.fireMessageReceived(ChannelHandlerContext, Object, SocketAddress)    
Decoder执行完之后，事件进入到下一个处理器NettyHandler，看下NettyHandler中的代码：   
  
这里直接交给handler处理了，这个handler封装了很多层：DecodeHandler->HeaderExchangeHandler->DubboProtocol.requestHandler，中间封装了好几万层这里只把重要的列出来，源头是从创建NettyServer的时候传过来的。

com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.createServer(URL)    
先会走到DecodeHandler.received：

com.alibaba.dubbo.remoting.transport.DecodeHandler.received(Channel, Object)   
  
这个message是Request类型的，要先decode一下，因为在之前已经解码过了，所以这里不会做任何事情，直接走下一个handler.received，这个handler就是HeaderExchangeHandler：

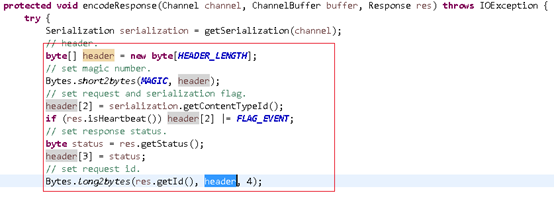
com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.received(Channel, Object)   
  
普通的同步接口twoWay属性是true走handleRequest方法处理请求，处理结束之后调用channel.send把结果返回到客户端。

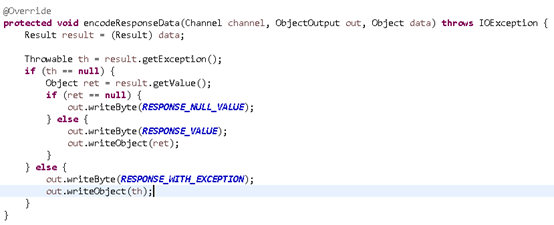
**提供端处理请求**

请求处理再走下一个handler的reply，这个handler就是DubboProtocol.requestHandler，把request对象中的data取出来传到requestHandler中，这个data就是前面的解码后的DecodeableRpcInvocation对象它是Invocation接口的一个实现：   
com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.handleRequest(ExchangeChannel, Request)   
  
com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol   
  
查找提供端请求对应的Invoker，在接口提供者初始化时，每个接口都会创建一个Invoker和Exporter，Exporter持有invoker实例，Exporter对象保存在DubboProtocol的exporterMap中，key是由URL生成的serviceKey，此时通过Invocation中的信息就可还原该serviceKey并且找到对应的Exporter和Invoker，在分析提供者初始化代码时知道它是Invoker-Filter的头节点，激活Filter后调用由ProxyFactory生成的Invoker：   
  
调用invoker.invoke时，通过反射调用最终的服务实现执行相关逻辑。   
服务执行结束之后，创建一个Response对象返回给客户端。在执行服务实现时会出现两种结果：成功和失败，如果成功，把返回值设置到Response的result中，Response的status设置成OK，如果失败，把失败异常设置到Response的errorMessage中，status设置成SERVICE\_ERROR。   
回到HeaderExchangeHandler.received中的代码，在handleRequest之后，调用channel.send把Response发送到客户端，这个channel封装客户端-服务端通信链路，最终会调用Netty框架，把响应写回到客户端。   


**提供端响应结果编码**

提供端要按照和消费端的协议把Response按照特定的协议进行编码，把编码后的数据写回到消费端，从上面的代码可以看到，在NettyServer初始化的时候，定义了三个IO事件处理器，服务端往客户端回写响应时产生下行事件，处理下行事件处理器，NettyCodecAdapter.encoder和NettyHandler是下行事件处理器，先激活NettyHandler，再激活NettyCodecAdapter. encoder，在NettyCodecAdapter. encoder对响应结果进行编码，还是通过Code2组件和请求编码时使用的组件一样，把响应类型和响应结果依次写回到客户端，根据协议会写入16个字节的数据头，包括：   
1、1-2字节魔数   
2、第3个字节，序列化组件类型，约定和客户端的序列化-反序列化协议   
3、第4个字节，响应状态，是OK还是error   
4、5-13个字节，响应id，这里的id和request中的id一样   
5、13-16个字节，响应数据长度

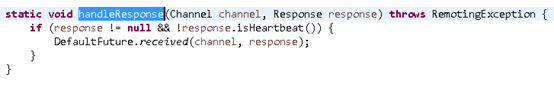
com.alibaba.dubbo.remoting.exchange.codec.ExchangeCodec.encodeResponse(Channel, ChannelBuffer, Response)   
  
返回结果有三种结果：1、没有返回值即返回类型是void；2、有返回值并且执行成功；3、服务调用异常。

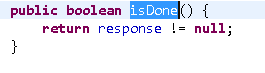
com.alibaba.dubbo.rpc.protocol.dubbo.DubboCodec.encodeResponseData(Channel, ObjectOutput, Object)   
  
解码后的数据会写入到通信链路中。

**消费端响应结果解码**

服务端给客户端回写数据之后，客户端会收到IO事件，一个上行事件。NettyClient中有两个上行事件处理器NettyCodecAdapter.decoder和NettyHandler，按照顺序decoder先执行对服务端传过来的数据进行解码，解析出序列化协议、响应状态、响应id（即请求id）。把响应body数据读到DecodeableRpcResult对象中，进行解析同时加载处理原始Request数据，这个Request对象在请求时会被缓存到DefaultFuture中，加载Request的目的是因为Request中Invocation中携带了服务接口的返回值类型信息，需要根据这个类型把响应解析创建对应类型的对象。

com.alibaba.dubbo.rpc.protocol.dubbo.DecodeableRpcResult.decode(Channel, InputStream)   
  
创建Response对象并且把解析出结果或异常设置到Response中。   
decoder把响应解析成Response对象中，NettyHandler接着往下处理，同样触发它的messageReceive事件，在提供端解码的时候看到了，它的handler封装关系是：DecodeHandler->HeaderExchangeHandler->DubboProtocol.requestHandler，主要处理在HeaderExchangeHandler中：

com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.handleResponse(Channel, Response)   
  
com.alibaba.dubbo.remoting.exchange.support.DefaultFuture.doReceived(Response)   
   
这里主要做的事情是唤醒调用者线程，并且把Response设置到DefaultFuture中，在消费者触发请求的代码中可以看到，消费端调用接口的时候请求写到提供端之后，会调用DefaultFuture.get阻塞等待响应结果：

com.alibaba.dubbo.remoting.exchange.support.DefaultFuture.get(int)   
  
com.alibaba.dubbo.remoting.exchange.support.DefaultFuture.isDone()   
   
在done这个Condition上进行条件等待，DefaultFuture.doReceive时，设置response唤醒done，此时调用线程被唤醒并且检查是否已经有了response（避免假唤醒），唤醒之后返回response中的result，调用端即拿到了接口的调用结果（返回值或异常），整个远程服务接口的调用流程就完成了。

**超时处理**

前面说了在进行接口调用时会出现两种情况：接口调用成功、接口调用异常，其实还有一种情况就是接口调用超时。在消费端等待接口返回时，有个timeout参数，这个时间是使用者设置的，可在消费端设置也可以在提供端设置，done.await等待时，会出现两种情况跳出while循环，一是线程被唤醒并且已经有了response，二是等待时间已经超过timeout，此时也会跳出while，当跳出while循环并且Future中没有response时，就说明接口已超时抛出TimeoutException，框架把TimeoutException封装成RpcException抛给应用层。

# Zookeeper

## Zookeeper是什么

ZooKeeper是一个开源的分布式协调服务，分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

## Zookeeper概念

### 2.1 集群角色

在 ZooKeeper 中，有三种角色：

**Leader**

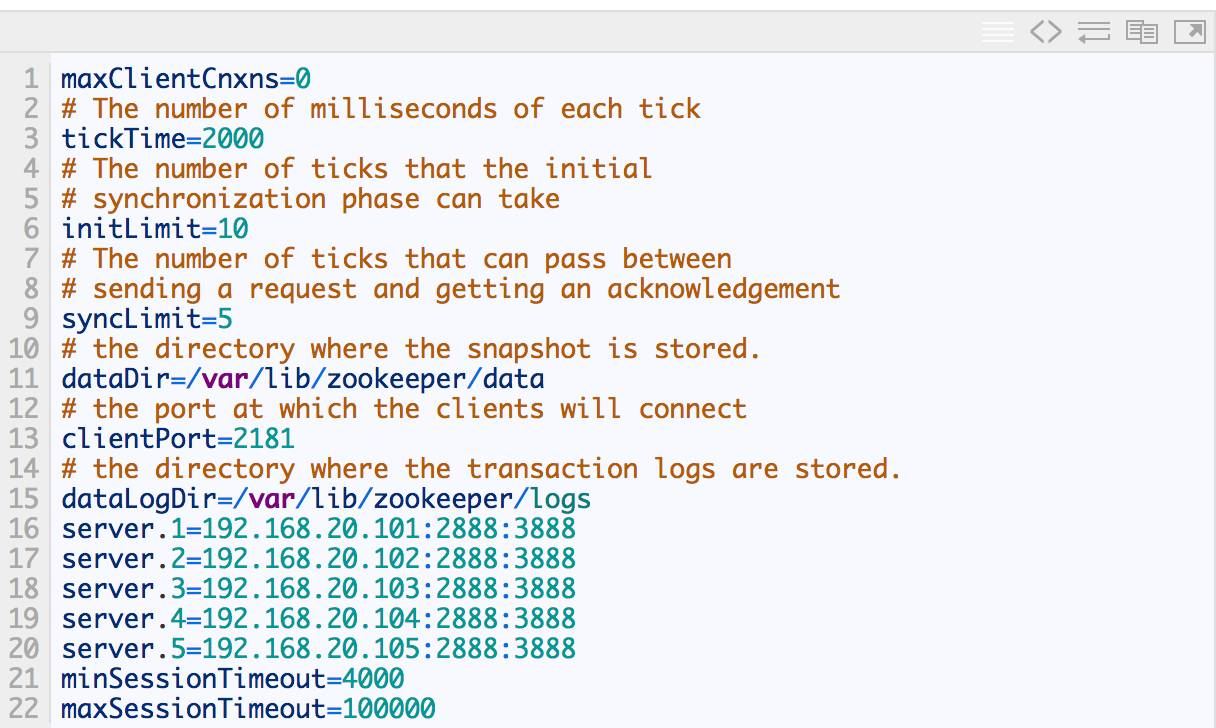
**Follower**

**Observer**

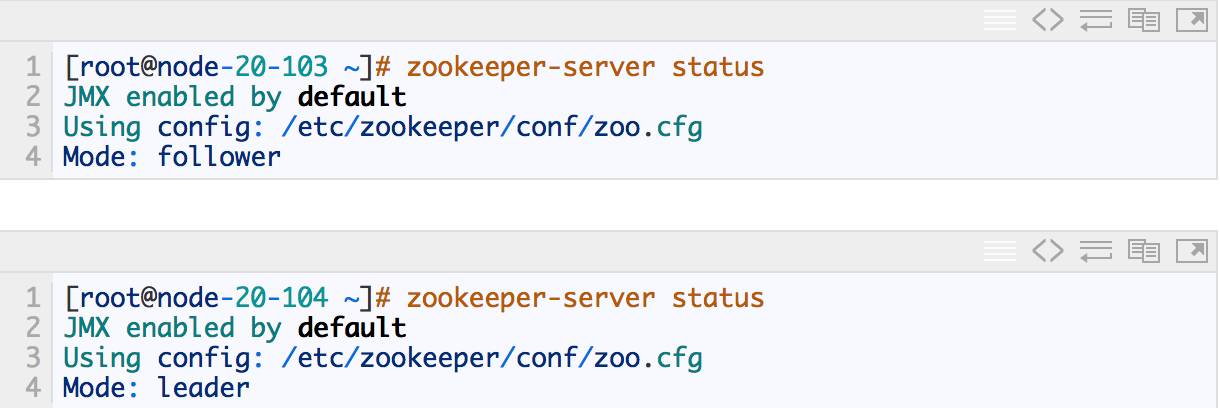
一个 ZooKeeper 集群同一时刻只会有一个 Leader，其他都是 Follower 或 Observer。

ZooKeeper 配置很简单，每个节点的配置文件(zoo.cfg)都是一样的，只有 myid 文件不一样。myid 的值必须是 zoo.cfg中server.{数值}的{数值}部分。

zoo.cfg 文件内容示例：



在装有 ZooKeeper 的机器的终端执行 zookeeper-server status 可以看当前节点的 ZooKeeper 是什么角色（Leader or Follower）。



如上，node-20-104 是 Leader，node-20-103 是 follower。

ZooKeeper 默认只有 Leader 和 Follower 两种角色，没有 Observer 角色。为了使用 Observer 模式，在任何想变成Observer的节点的配置文件中加入:peerType=observer 并在所有 server 的配置文件中，配置成 observer 模式的 server 的那行配置追加 :observer，例如：

server.1:localhost:2888:3888:observer

ZooKeeper 集群的所有机器通过一个 Leader 选举过程来选定一台被称为『Leader』的机器，Leader服务器为客户端提供读和写服务，负责进行投票的发起和决议，更新系统状态。

Follower主要有四个功能：

• 1. 向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；PING消息是指Learner的心跳信息；REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；ACK消息是 Follower的对提议的回复，超过半数的Follower通过，则commit该提议；REVALIDATE消息是用来延长SESSION有效时间。

• 2 .接收Leader消息并进行处理；

• 3 .接收Client的请求，如果为写请求，发送给Leader进行投票；

• 4 .返回Client结果。

Follower 和 Observer 都能提供读服务，不能提供写服务。两者唯一的区别在于，Observer 机器不参与 Leader 选举过程，也不参与写操作的『过半写成功』策略，因此 Observer 可以在不影响写性能的情况下提升集群的读性能。



### 2.2 会话（Session）

Session 是指客户端会话，在讲解客户端会话之前，我们先来了解下客户端连接。在 ZooKeeper 中，一个客户端连接是指客户端和 ZooKeeper 服务器之间的TCP长连接。

ZooKeeper 对外的服务端口默认是2181，客户端启动时，首先会与服务器建立一个TCP连接，从第一次连接建立开始，客户端会话的生命周期也开始了，通过这个连接，客户端能够通过心跳检测和服务器保持有效的会话，也能够向 ZooKeeper 服务器发送请求并接受响应，同时还能通过该连接接收来自服务器的 Watch 事件通知。

Session 的 SessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 SessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

### 2.3 数据节点（ZNode）

在谈到分布式的时候，一般『节点』指的是组成集群的每一台机器。而ZooKeeper 中的数据节点是指数据模型中的数据单元，称为 ZNode。ZooKeeper 将所有数据存储在内存中，数据模型是一棵树（ZNode Tree），由斜杠（/）进行分割的路径，就是一个ZNode，如 /hbase/master，其中 hbase 和 master 都是 ZNode。每个 ZNode 上都会保存自己的数据内容，同时会保存一系列属性信息。

注：

这里的 ZNode 可以理解成既是Unix里的文件，又是Unix里的目录。因为每个 ZNode 不仅本身可以写数据（相当于Unix里的文件），还可以有下一级文件或目录（相当于Unix里的目录）。

在 ZooKeeper 中，ZNode 可以分为持久节点和临时节点两类。

**持久节点**

所谓持久节点是指一旦这个 ZNode 被创建了，除非主动进行 ZNode 的移除操作，否则这个 ZNode 将一直保存在 ZooKeeper 上。

**临时节点**

临时节点的生命周期跟客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。

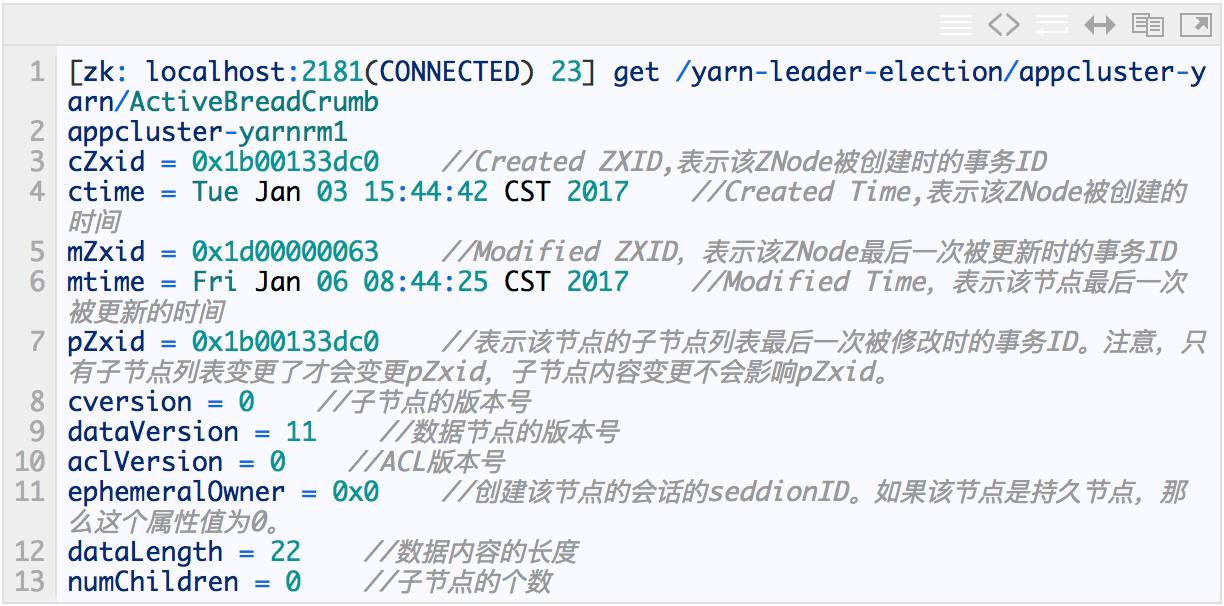
另外，ZooKeeper 还允许用户为每个节点添加一个特殊的属性：SEQUENTIAL。一旦节点被标记上这个属性，那么在这个节点被创建的时候，ZooKeeper 就会自动在其节点后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

### 2.4 版本

ZooKeeper 的每个 ZNode 上都会存储数据，对应于每个 ZNode，ZooKeeper 都会为其维护一个叫作 Stat 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本，分别是 version（当前ZNode的版本）、cversion（当前ZNode子节点的版本）和 aversion（当前 ZNode 的 ACL 版本）。

### 2.5 状态信息

每个 ZNode 除了存储数据内容之外，还存储了 ZNode 本身的一些状态信息。用 get 命令可以同时获得某个 ZNode 的内容和状态信息。如下：



在 ZooKeeper 中，version 属性是用来实现乐观锁机制中的『写入校验』的（保证分布式数据原子性操作）。

### 2.6 事务操作

在ZooKeeper中，能改变ZooKeeper服务器状态的操作称为事务操作。一般包括数据节点创建与删除、数据内容更新和客户端会话创建与失效等操作。对应每一个事务请求，ZooKeeper 都会为其分配一个全局唯一的事务ID，用 ZXID 表示，通常是一个64位的数字。每一个 ZXID 对应一次更新操作，从这些 ZXID 中可以间接地识别出 ZooKeeper 处理这些事务操作请求的全局顺序。

### 2.7 Watcher

Watcher（事件监听器），是 ZooKeeper 中一个很重要的特性。ZooKeeper允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper 服务端会将事件通知到感兴趣的客户端上去。该机制是 ZooKeeper 实现分布式协调服务的重要特性。

### 2.8 ACL

ZooKeeper 采用 ACL（Access Control Lists）策略来进行权限控制。ZooKeeper 定义了如下5种权限。

CREATE: 创建子节点的权限。

READ: 获取节点数据和子节点列表的权限。

WRITE：更新节点数据的权限。

DELETE: 删除子节点的权限。

ADMIN: 设置节点ACL的权限。

注意：CREATE 和 DELETE 都是针对子节点的权限控制。

## ZooKeeper的ZAB协议

### 3.1 ZAB协议介绍

ZooKeeper并没有完全采用Paxos算法，而是使用了一种称为Zookeeper Atomoc Broadcast(ZAB，Zookeeper原子消息广播协议)的协议作为其数据一致性的核心算法。

几个名词定义：

（1） Serverid：在配置server时，给定的服务器的标示id。

（2）ZAB协议的事务编号ZXID是一个64位的数字，低32位是一个简单的单调递增计数器，针对客户端的每一个事务请求都会对该计数器加1；高32位是Leader周期epoch的编号，每次选举产生一个新的Leader服务器，就会从这个Leader服务器上取出本地日志中最大事务Proposal的ZXID，从该ZXID中解析出对应的epoch编号对其加1作为新的epoch编号，并将低32位置0作为新的ZXID。

（3）Epoch：选举的轮数，即逻辑时钟。随着选举的轮数++

（4）Server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

OBSERVING：leader是否有改变，然后同步leader的状态

ZAB协议主要包括消息广播和崩溃恢复两个过程。进一步细分可以分为三个阶段：发现(Discovery)、同步(Synchronization)和广播(Broadcast)。组成ZAB协议的每个分布式进程会循环地执行这三个阶段，我们将这样一个循环称为一个主进程周期。

### 3.2 ZAB协议崩溃恢复

ZAB协议会让ZK集群进入崩溃恢复模式的情况如下：

（1）当服务框架在启动过程中

（2）当Leader服务器出现网络中断，崩溃退出与重启等异常情况。

（3）当集群中已经不存在过半的服务器与Leader服务器保持正常通信。

当leader挂掉后，集群无法进行工作，所以需要一个高效且可靠的leader选举算法。Zookeeper的实现是FastLeaderElection算法。

#### 3.2.1服务器启动时期的Leader选举

**步骤一**：每个Server投出一票。他们两都选自己为Leader，投票的内容为（SID，ZXID）。SID即Server的id，安装zookeeper时配置文件中所配置的myid；ZXID，事务id，为节点的更新程度，ZXID越大，代表Server对Znode的操作越新。由于服务器初始化，每个Sever上的Znode为0，所以Server1投的票为（1,0），Server2为（2,0）。投票信息还包括：逻辑时钟的值,这个值从0开始递增,每次选举对应一个值；本机在当前选举过程中的状态。两Server将各自投票发给集群中其他机器。

**步骤二**：每个Server接收来自其他Server的投票。集群中的每个Server先判断投票有效性，如检查是不是本轮的投票，是不是来自Looking状态的服务器投的票。

**步骤三**：处理投票。每个服务器都需要将别人的投票和自己的投票进行比较，比较规则：

- 首先对比ZXID。ZXID大的服务器优先作为Leader

- 若ZXID相同，比如初始化的时候，每个Server的ZXID都为0，就会比较myid，myid大的选出来做Leader。

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0，再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

**步骤四**：统计投票。每轮投票之后，每台服务器都会统计投票信息，如果判定有某个Server有过半的票数投它，那么该Server将会作为Leader。当接收来自过半Follower的CEPOCH消息后，准Leader会从所有接收到的CEPOCH消息中选取出最大的epoch值，然后加1作为新的epoch值，生成NEWEPOCH消息给这些过半的Follower。

**步骤五**：改变服务器状态。当确定了Leader之后，每个Server更新自己的状态，Leader将状态更新为Leading，Follower将状态更新为Following。当Follower接收到来自准Leader的NEWEPOCH消息后，检测如果当前CEPOCH值小于新的epoch值，就会将CEPOCH的epoch重新赋值。Leader接收来自过半Follower的确认消息之后，Leader就会从这过半服务器中选出一个Follower作为初始化事务集合。

**阶段二：同步**

步骤一：

#### 3.2.2服务器运行时期的Leader选举

在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，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。

(1) 变更状态。Leader挂后，余下的非Observer服务器都会讲自己的服务器状态变更为LOOKING，然后开始进入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，此时假定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) 接收来自各个服务器的投票。与启动时过程相同。

(4) 处理投票。与启动时过程相同，此时，Server1将会成为Leader。

(5) 统计投票。与启动时过程相同。

(6) 改变服务器的状态。与启动时过程相同。

上边选出的只是准leader，要想变成leader还需完成数据同步。

**数据同步：**

同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。只有当集群过半机器都同步完成，准 leader 才会成为真正的 leader。follower 只会接收 zxid 比自己的 lastZxid 大的提议。

当完成Leader选举后，进行故障恢复的第二步就是数据同步： Leader服务器会为每一个Follower服务器准备一个队列，并将那些没有被各个Follower服务器同步的事务以Proposal的形式逐条发给各个Follower服务器，并在每一个Proposal后都紧跟一个commit消息，表示该事务已经被提交，当follower服务器将所有尚未同步的事务proposal都从leader服务器同步过来并成功应用到本地后，leader服务器就会将该follower加入到真正可用的follower列表中。（新选举周期，epoch已经更新了）

Fast Leader选举算法中提到的同步数据时使用的逻辑时钟,它的初始值是0,每次选举过程都会递增的,在leader正式上任之后做的第一件事情,就是根据当前保存的数据id值,设置最新的逻辑时钟值。

随后,leader构建NEWLEADER封包,该封包的数据是当前最大数据的id,广播给所有的follower,也就是告知follower leader保存的数据id是多少,大家看看是不是需要同步。然后,leader根据follower数量给每个follower创建一个线程LearnerHandler,专门负责接收它们的同步数据请求。leader主线程开始阻塞在这里,等待其他follower的回应(也就是LearnerHandler线程的处理结果),同样的,只有在超过半数的follower已经同步数据完毕,这个过程才能结束,leader才能正式成为leader。

leader所做的工作：

所以其实leader与follower同步数据的大部分操作都在LearnerHandler线程中处理的,接着看这一块。

leader接收到的来自某个follower封包一定是FOLLOWERINFO，该封包告知了该服务器保存的数据id.之后根据这个数据id与本机保存的数据进行比较:

1) 如果数据完全一致,则发送DIFF封包告知follower当前数据就是最新的了。

2) 判断这一阶段之内有没有已经被提交的提议值,如果有,那么:

a) 如果有部分数据没有同步,那么会发送DIFF封包将有差异的数据同步过去，同时将follower没有的数据逐个发送COMMIT封包给follower要求记录下来。

b) 如果follower数据id更大,那么会发送TRUNC封包告知截除多余数据.（一台leader数据没同步就宕掉了，选举之后恢复了，数据比现在leader更新）

3) 如果这一阶段内没有提交的提议值,直接发送SNAP封包将快照同步发送给follower。

4）消息完毕之后,发送UPTODATE封包告知follower当前数据就是最新的了,再次发送NEWLEADER封包宣称自己是leader,等待follower的响应。

follower做的工作：

（1）会尝试与leader建立连接,这里有一个机制,如果一定时间内没有连接上,就报错退出,重新回到选举状态。

（2）其次在发送FOLLOWERINFO封包,该封包中带上自己的最大数据id,也就是会告知leader本机保存的最大数据id。

（3）根据前面对LeaderHandler的分析,leader会根据不同的情况发送DIFF,UPTODATE,TRUNC,SNAP,依次进行处理就是了,此时follower跟leader的数据也就同步上了。

（4）由于leader端发送的最后一个封包是UPTODATE,因此在接收到这个封包之后follower结束同步数据过程,发送ACK封包回复leader。

以上过程中,任何情况出现的错误,服务器将自动将选举状态切换到LOOKING状态,重新开始进行选举。

### 3.3 ZAB消息广播

客户端提交事务请求时Leader节点为每一个请求生成一个事务Proposal，将其发送给集群中所有的Follower节点，收到过半Follower的反馈后开始对事务进行提交，ZAB协议使用了原子广播协议；在ZAB协议中只需要得到过半的Follower节点反馈Ack就可以对事务进行提交，这也导致了Leader节点崩溃后可能会出现数据不一致的情况，ZAB使用了崩溃恢复来处理数字不一致问题；消息广播使用了TCP协议进行通讯所以保证了接受和发送事务的顺序性。广播消息时Leader节点为每个事务Proposal分配一个全局递增的ZXID（事务ID），每个事务Proposal都按照ZXID顺序来处理；

Leader节点为每一个Follower节点分配一个队列按事务ZXID顺序放入到队列中，且根据队列的规则FIFO来进行事务的发送。Follower节点收到事务Proposal后会将该事务以事务日志方式写入到本地磁盘中，成功后反馈Ack消息给Leader节点，Leader在接收到过半Follower节点的Ack反馈后就会进行事务的提交，以此同时向所有的Follower节点广播Commit消息，Follower节点收到Commit后开始对事务进行提交；

## ZooKeeper典型应用场景

ZooKeeper 是一个高可用的分布式数据管理与协调框架。基于对ZAB算法的实现，该框架能够很好地保证分布式环境中数据的一致性。也是基于这样的特性，使得 ZooKeeper 成为了解决分布式一致性问题的利器。

### 3.1 数据发布与订阅（配置中心）

数据发布与订阅，即所谓的配置中心，顾名思义就是发布者将数据发布到 ZooKeeper 节点上，供订阅者进行数据订阅，进而达到动态获取数据的目的，实现配置信息的集中式管理和动态更新。

在我们平常的应用系统开发中，经常会碰到这样的需求：系统中需要使用一些通用的配置信息，例如机器列表信息、数据库配置信息等。这些全局配置信息通常具备以下3个特性。

数据量通常比较小。

数据内容在运行时动态变化。

集群中各机器共享，配置一致。

对于这样的全局配置信息就可以发布到 ZooKeeper上，让客户端（集群的机器）去订阅该消息。

发布/订阅系统一般有两种设计模式，分别是推（Push）和拉（Pull）模式。

**推**：服务端主动将数据更新发送给所有订阅的客户端。

**拉**：客户端主动发起请求来获取最新数据，通常客户端都采用定时轮询拉取的方式。

ZooKeeper 采用的是推拉相结合的方式。如下：

客户端想服务端注册自己需要关注的节点，一旦该节点的数据发生变更，那么服务端就会向相应的客户端发送Watcher事件通知，客户端接收到这个消息通知后，需要主动到服务端获取最新的数据（推拉结合）。

### 3.2 命名服务(Naming Service)

命名服务也是分布式系统中比较常见的一类场景。在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务，远程对象等等——这些我们都可以统称他们为名字（Name）。

其中较为常见的就是一些分布式服务框架（如RPC、RMI）中的服务地址列表。通过在ZooKeepr里创建顺序节点，能够很容易创建一个全局唯一的路径，这个路径就可以作为一个名字。

ZooKeeper 的命名服务即生成全局唯一的ID。

### 3.3 分布式协调/通知

ZooKeeper 中特有 Watcher 注册与异步通知机制，能够很好的实现分布式环境下不同机器，甚至不同系统之间的通知与协调，从而实现对数据变更的实时处理。使用方法通常是不同的客户端都对ZK上同一个 ZNode 进行注册，监听 ZNode 的变化（包括ZNode本身内容及子节点的），如果 ZNode 发生了变化，那么所有订阅的客户端都能够接收到相应的Watcher通知，并做出相应的处理。

ZK的分布式协调/通知，是一种通用的分布式系统机器间的通信方式。

#### 3.3.1 心跳检测

机器间的心跳检测机制是指在分布式环境中，不同机器（或进程）之间需要检测到彼此是否在正常运行，例如A机器需要知道B机器是否正常运行。在传统的开发中，我们通常是通过主机直接是否可以相互PING通来判断，更复杂一点的话，则会通过在机器之间建立长连接，通过TCP连接固有的心跳检测机制来实现上层机器的心跳检测，这些都是非常常见的心跳检测方法。

下面来看看如何使用ZK来实现分布式机器（进程）间的心跳检测。

基于ZK的临时节点的特性，可以让不同的进程都在ZK的一个指定节点下创建临时子节点，不同的进程直接可以根据这个临时子节点来判断对应的进程是否存活。通过这种方式，检测和被检测系统直接并不需要直接相关联，而是通过ZK上的某个节点进行关联，大大减少了系统耦合。

#### 3.3.2 工作进度汇报

在一个常见的任务分发系统中，通常任务被分发到不同的机器上执行后，需要实时地将自己的任务执行进度汇报给分发系统。这个时候就可以通过ZK来实现。在ZK上选择一个节点，每个任务客户端都在这个节点下面创建临时子节点，这样便可以实现两个功能：

通过判断临时节点是否存在来确定任务机器是否存活。

各个任务机器会实时地将自己的任务执行进度写到这个临时节点上去，以便中心系统能够实时地获取到任务的执行进度。

### 3.4 Master选举

Master 选举可以说是 ZooKeeper 最典型的应用场景了。比如 HDFS 中 Active NameNode 的选举、YARN 中 Active ResourceManager 的选举和 HBase 中 Active HMaster 的选举等。

针对 Master 选举的需求，通常情况下，我们可以选择常见的关系型数据库中的主键特性来实现：希望成为 Master 的机器都向数据库中插入一条相同主键ID的记录，数据库会帮我们进行主键冲突检查，也就是说，只有一台机器能插入成功——那么，我们就认为向数据库中成功插入数据的客户端机器成为Master。

依靠关系型数据库的主键特性确实能够很好地保证在集群中选举出唯一的一个Master。

但是，如果当前选举出的 Master 挂了，那么该如何处理？谁来告诉我 Master 挂了呢？显然，关系型数据库无法通知我们这个事件。但是，ZooKeeper 可以做到！

利用 ZooKeepr 的强一致性，能够很好地保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即 ZooKeeper 将会保证客户端无法创建一个已经存在的 ZNode。

也就是说，如果同时有多个客户端请求创建同一个临时节点，那么最终一定只有一个客户端请求能够创建成功。利用这个特性，就能很容易地在分布式环境中进行 Master 选举了。

成功创建该节点的客户端所在的机器就成为了 Master。同时，其他没有成功创建该节点的客户端，都会在该节点上注册一个子节点变更的 Watcher，用于监控当前 Master 机器是否存活，一旦发现当前的Master挂了，那么其他客户端将会重新进行 Master 选举。

这样就实现了 Master 的动态选举。

### 3.5 分布式锁

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

分布式锁又分为排他锁和共享锁两种。

#### 3.5.1 排他锁

排他锁（Exclusive Locks，简称X锁），又称为写锁或独占锁。

如果事务T1对数据对象O1加上了排他锁，那么在整个加锁期间，只允许事务T1对O1进行读取和更新操作，其他任何事务都不能在对这个数据对象进行任何类型的操作（不能再对该对象加锁），直到T1释放了排他锁。

可以看出，排他锁的核心是如何保证当前只有一个事务获得锁，并且锁被释放后，所有正在等待获取锁的事务都能够被通知到。

如何利用 ZooKeeper 实现排他锁？

**定义锁**

ZooKeeper 上的一个 ZNode 可以表示一个锁。例如 /exclusive\_lock/lock节点就可以被定义为一个锁。

**获得锁**

如上所说，把ZooKeeper上的一个ZNode看作是一个锁，获得锁就通过创建 ZNode 的方式来实现。所有客户端都去 /exclusive\_lock节点下创建临时子节点 /exclusive\_lock/lock。ZooKeeper 会保证在所有客户端中，最终只有一个客户端能够创建成功，那么就可以认为该客户端获得了锁。同时，所有没有获取到锁的客户端就需要到/exclusive\_lock节点上注册一个子节点变更的Watcher监听，以便实时监听到lock节点的变更情况。

**释放锁**

因为 /exclusive\_lock/lock 是一个临时节点，因此在以下两种情况下，都有可能释放锁。

当前获得锁的客户端机器发生宕机或重启，那么该临时节点就会被删除，释放锁。

正常执行完业务逻辑后，客户端就会主动将自己创建的临时节点删除，释放锁。

无论在什么情况下移除了lock节点，ZooKeeper 都会通知所有在 /exclusive\_lock 节点上注册了节点变更 Watcher 监听的客户端。这些客户端在接收到通知后，再次重新发起分布式锁获取，即重复『获取锁』过程。

#### 3.5.2 共享锁

共享锁（Shared Locks，简称S锁），又称为读锁。如果事务T1对数据对象O1加上了共享锁，那么T1只能对O1进行读操作，其他事务也能同时对O1加共享锁（不能是排他锁），直到O1上的所有共享锁都释放后O1才能被加排他锁。

总结：可以多个事务同时获得一个对象的共享锁（同时读），有共享锁就不能再加排他锁（因为排他锁是写锁）

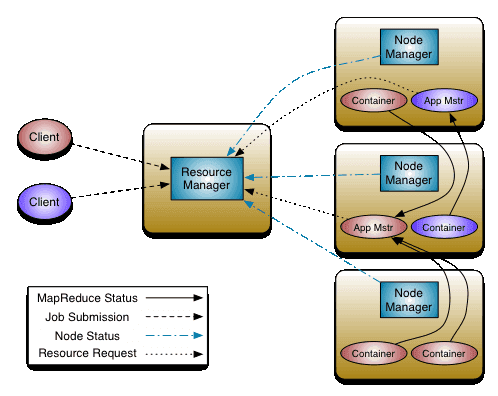
## ZooKeeper在大型分布式系统中的应用

前面已经介绍了 ZooKeeper 的典型应用场景。本节将以常见的大数据产品 Hadoop 和 HBase 为例来介绍 ZooKeeper 在其中的应用，帮助大家更好地理解 ZooKeeper 的分布式应用场景。

### 4.1 ZooKeeper在Hadoop中的应用

在 Hadoop 中，ZooKeeper 主要用于实现 HA(Hive Availability），包括 HDFS的 NamaNode 和 YARN 的 ResourceManager 的 HA。同时，在 YARN 中， ZooKeepr 还用来存储应用的运行状态。

HDFS 的 NamaNode 和 YARN 的 ResourceManager 利用 ZooKeepr 实现 HA 的原理是一样的，所以本节以YARN为例来介绍。



从上图可以看出，YARN主要由ResourceManager（RM）、NodeManager（NM）、ApplicationMaster（AM）和Container四部分组成。其中最核心的就是ResourceManager。

ResourceManager 负责集群中所有资源的统一管理和分配，同时接收来自各个节点（NodeManager）的资源汇报信息，并把这些信息按照一定的策略分配给各个应用程序（Application Manager），其内部维护了各个应用程序的ApplicationMaster信息、NodeManager信息以及资源使用信息等。

为了实现HA，必须有多个ResourceManager并存（一般就两个），并且只有一个ResourceManager处于Active状态，其他的则处于Standby状态，当Active节点无法正常工作（如机器宕机或重启）时，处于Standby的就会通过竞争选举产生新的Active节点。

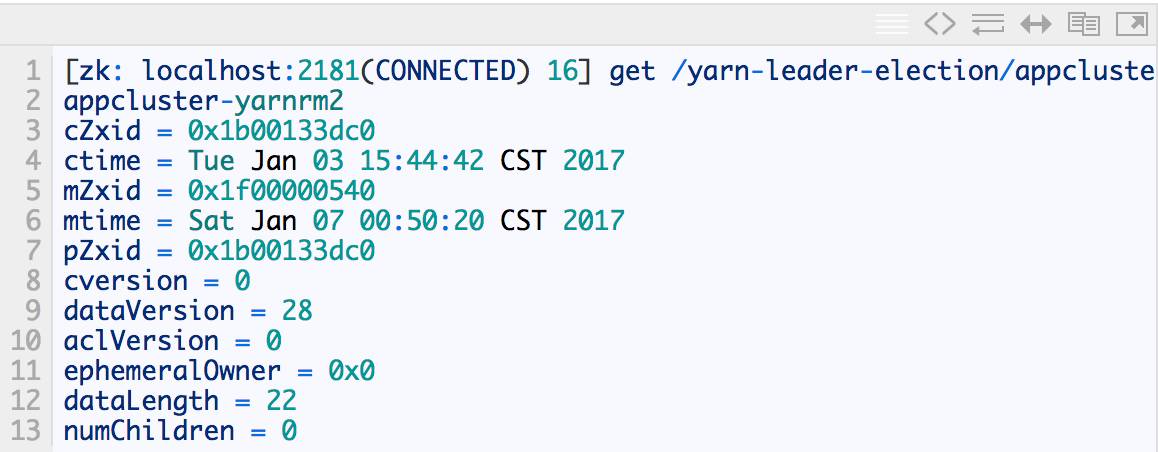
### 4.2 主备切换

下面我们就来看看YARN是如何实现多个ResourceManager之间的主备切换的。

1. 创建锁节点

在 ZooKeeper 上会有一个/yarn-leader-election/appcluster-yarn的锁节点，所有的 ResourceManager 在启动的时候，都会去竞争写一个Lock子节点：/yarn-leader-election/appcluster-yarn/ActiveBreadCrumb，该节点是临时节点。

ZooKeepr 能够为我们保证最终只有一个ResourceManager能够创建成功。创建成功的那个 ResourceManager 就切换为 Active 状态，没有成功的那些 ResourceManager 则切换为 Standby 状态。



可以看到此时集群中 ResourceManager2 为 Active。

1、注册 Watcher 监听：

所有 Standby 状态的 ResourceManager 都会向 /yarn-leader-election/appcluster-yarn/ActiveBreadCrumb 节点注册一个节点变更的Watcher监听，利用临时节点的特性，能够快速感知到Active状态的ResourceManager的运行情况。

2、主备切换：

当Active状态的ResourceManager出现诸如宕机或重启的异常情况时，其在ZooKeeper上连接的客户端会话就会失效，因此/yarn-leader-election/appcluster-yarn/ActiveBreadCrumb节点就会被删除。此时其余各个Standby状态的ResourceManager就都会接收到来自ZooKeeper服务端的Watcher事件通知，然后会重复进行步骤1的操作。

以上就是利用 ZooKeeper 来实现 ResourceManager 的主备切换的过程，实现了 ResourceManager 的HA。

HDFS 中 NameNode 的 HA 的实现原理跟 YARN 中 ResourceManager 的 HA 的实现原理相同。其锁节点为/hadoop-ha/mycluster/ActiveBreadCrumb。

### 4.3 ResourceManager状态存储

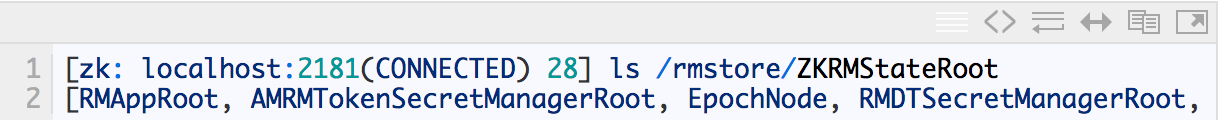
在 ResourceManager 中，RMStateStore 能够存储一些 RM 的内部状态信息，包括 Application 以及它们的 Attempts 信息、Delegation Token 及 Version Information 等。需要注意的是，RMStateStore 中的绝大多数状态信息都是不需要持久化存储的，因为很容易从上下文信息中将其重构出来，如资源的使用情况。在存储的设计方案中，提供了三种可能的实现，分别如下。

基于内存实现，一般是用于日常开发测试。

基于文件系统的实现，如HDFS。

基于 ZooKeeper 实现。

由于这些状态信息的数据量都不是很大，因此 Hadoop 官方建议基于 ZooKeeper 来实现状态信息的存储。在 ZooKeepr 上，ResourceManager 的状态信息都被存储在 /rmstore 这个根节点下面。



RMAppRoot 节点下存储的是与各个 Application 相关的信息，RMDTSecretManagerRoot 存储的是与安全相关的 Token 等信息。每个 Active 状态的 ResourceManager 在初始化阶段都会从 ZooKeeper 上读取到这些状态信息，并根据这些状态信息继续进行相应的处理。