# 概述

dubbo架构图，来源于<http://dubbo.apache.org/zh-cn/docs/dev/design.html>



需要理清层级关系，最底层是序列化。

# 知识点

## Spring Schema

参照Demo学习

## Dubbo SPI

参考官网样例

<http://dubbo.apache.org/zh-cn/blog/introduction-to-dubbo-spi.html>

博客

<https://cxis.me/2017/02/18/Dubbo%E4%B8%ADSPI%E6%89%A9%E5%B1%95%E6%9C%BA%E5%88%B6%E8%AF%A6%E8%A7%A3/>

### 概念

|  |  |
| --- | --- |
| 概念 | 描述 |
| 扩展点 | 被注解SPI标注的接口 |
| 扩展 | 扩展点的具体实现 |
| 扩展实例 | 某个扩展的具体实例（已经在内存被创建） |
| 扩展自适应实例 | 一种特殊的扩展实例，其扩展的具体实现是通过自动生成的。 |
| 扩展别名 | 一个扩展的别名，位于配置文件中 |
| ExtensionLoader | dubbo扩展机制的“调度中心”，类似于JDK SPI的ServiceLoader，负责扩展的加载和生命周期 |
| @SPI | 标注注解，用于指定扩展点，没有标注该注解的接口被ExtensionLoader加载会报异常 |
| @Adaptive |  |

### 原理

原理参照Demo逐步调试代码。

## Proxy

### 生产者代理（provider stub）

### 消费者代理（consumer stub）

重点代码在ReferenceConfig的Init方法，调用createProxy方法。

生成proxy0类是调用Proxy类中的getProxy方法

# 服务发布过程

1. spring 加载ServiceBean时，会调用对应的export方法将服务暴露。
2. 以zookeeper为注册中心时，在/dubbo节点下创建一个以接口的全限定名的znode（永久节点），然后会在这个节点下创建四个子节点providers,consumers,routers，configurators（4个节点都是永久节点），当有新provider加入时，会在providers节点下创建临时节点，节点包括provider的所有信息，通了当有consumer加入会在consumers节点下创建临时节点；
3. 注册服务实在ZookeeperRegistry#doRegister方法，该方法调用create时是递归创建znode

# 服务订阅过程

1. Spring加载<dubbo:reference>标签，Spring加载<dubbo:reference>标签时会创建一个代理对象。代理对象的创建过程在ReferenceBean的afterPropertiesSet方法。
2. consumer启动时，ReferenceBean实现了Spring的FactoryBean接口，会调用对应的getObject方法，这个方法会去zookeeper上创建一个consumer节点（此处不会创建provider，router，configurator），详细代码在ZookeeperRegistry#doRegister，同时会订阅注册中心Zookeeper该接口下的providers，routers，configurators这三个节点的信息（订阅之前先要判断这三个节点是否存在，不存在就会创建），具体订阅代码在RegistryProtocol#doRefer----->ZookeeperRegistry#doSubscribe，这段代码执行结束前会调用notify通知该consumer去拉取providers等信息。详细如第三点所说；
3. 启动之后的consumer中的zookeeper会监听（订阅）接口节点下的子节点，如果发生变动会通知consumers目录下的所有consumer，对应dubbo中的源码在com.alibaba.dubbo.registry.integration.RegistryDirectory#notify（可以通过增加provider调试consumer的notify方法）

# 服务提供者

## 线程相关

1. NettyServerWorker线程监听到请求数据，将数据提交给DubboServerHandler线程池

## 总结

dubbo服务提供者是通过netty监听网络请求，当服务调用请求到达时，netty会将请求向上传递，指导调用dubbo封装的nettyserver。

1. netty监听网络请求
2. netty会调用dubbo实现的decode
3. netty将请求传给dubbo，调用com.alibaba.dubbo.remoting.transport.netty4. NettyServerHandler#channelRead方法。
4. 经过层层上传，最终会将请求放到dubbo业务调度线程池。com.alibaba.dubbo.remoting.transport.dispatcher.all. AllChannelHandler#received方法，线程池的任务是ChannelEventRunnable。
5. ChannelEventRunnable的run方法会根据任务状态执行不同操作。在此处有个代码性能优化，因为run方法执行最多的状态是RECEIVED（也就是有90%（瞎说的比例）的run方法调用会执行这个状态），所以把这个状态放到if代码，其他状态通过switch来处理，这样可以提供性能。
6. 调用DecodeHandler的再次判断是否需要解码操作

## 服务提供者Proxy创建时机

Dubbo的代理是通过spi来扩展的，扩展点事ProxyFactory，默认是采用javassit来实现动态代理，对应的扩展实现是JavassistProxyFactory，获取方法是getInvoker，因此打个断点便可知道服务提供者proxy创建时机。

为啥代理方式没有cglib？

因为cglib是通过类继承来实现动态代理，通过dubbo实现的远程调用中一般都是通过接口来实现服务的，因此没有cglib。

### javassit代理实现

这种代理实现中，服务提供者proxy涉及三类对象

* 第一个是通过服务接口的实现对象realObject，通过spring配置文件配置或者自动扫描生成；
* Wrapper类对象wrapper，该类的invokeMethod方法会获取到realObject方法，根据服务方法名字调用对应的方法。wrapper对象存放在Wrapper的static map中缓存起来。
* AbstractProxyInvoker子类对象，invoker对象包含一个wrapper对象wrapper。

因此当发起服务调用时，dubbo首先根据接口名获取对应的invoker对象，调用invoke方法，invoke方法会触发wrapper调用invokemethod方法，然后调用realobject中的服务方法。

### jdk代理实现

通过在标签中加入<dubbo:service proxy="jdk" />来修改代理方式。

这种方式的代理主要通过AbstractProxyInvoker子类对象来完成，该子类对象的doInvoke方法通过反射调用realobject的服务方法。

# 服务消费全过程

参考官网<https://dubbo.apache.org/zh-cn/docs/source_code_guide/service-invoking-process.html>

自我总结技术点

1. Spring加载dubbo bean（理解Spring schema）
2. 通过spring获取对应的consumer代理对象（代理对象何时生成，代理如何实现）？

代理类是在ReferenceConfig的init方法中创建的，getBean的时候创建对应的代理对象。

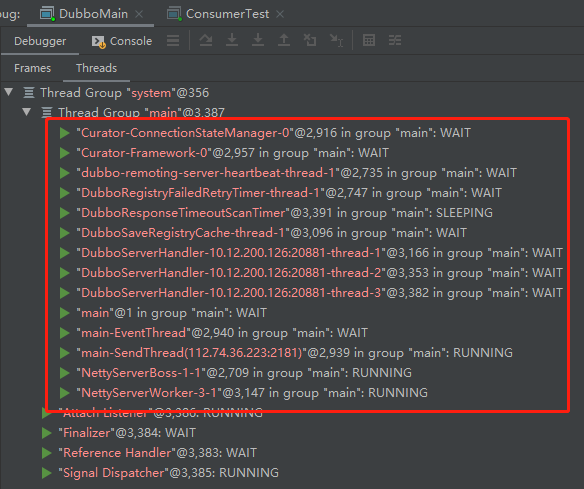
1. 代理对象调用接口方法；
2. 根据路由规则过滤掉不符合要求的invoker；
3. 调用抽象的Invoker，通过LoadBalance筛选provider（前提是多个提供者），此处掌握负载均衡算法
4. 通过具体的invoker（负载均衡选出来的）调用

# 线程角度

为了更加全面的了解Provider和Consumer启动后运行情况，再次从线程角度分析具体的运行情况。

## provider线程

调试dubbo\_demo中的provider，可以看到provider中启动的线程情况，如下，其中main group中的线程是provider的业务线程：



各个线程的功能简述如下：

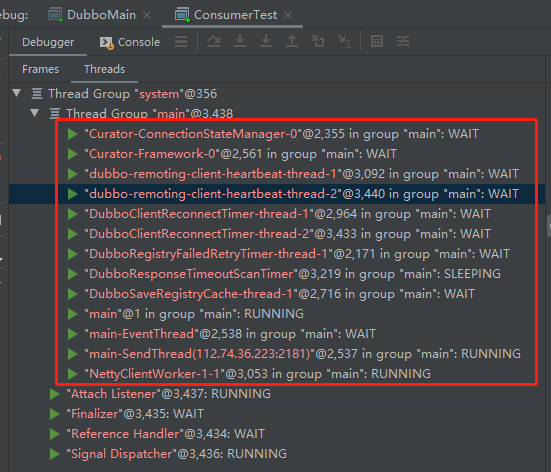
* zookeeper相关，其中main-SendThread(ip:port)线程是zkclien提供的，主要负责zkclient 向zk服务器发送heartbeat，如果有其他event需要发送给zk也是通过这个线程去发送；main-EventThread是zkclient接收来自zk服务器的事件线程；Curator打头的线程也是zkclient相关的一些线程，具体业务暂时不清楚。
* netty相关，NettyServerBoss（此处是Dubbo命名的线程，IO线程池），负责监听consumer发送的请求并封存成event；NettyServerWorker（此处是Dubbo命名的线程，IO线程池）负责处理NettyServerBoss提交的事件，主要负责IO ；有关详细可以参考netty的reactor线程模型；
* dubbo相关。

重点关注一下dubbo相关的线程

1. dubbo-remoting-server-heartbeat线程，负责provider和consumer之间的心跳。provider端会每隔60秒发送一次heartbeat，如果3次（180秒）没有得到回复，则provider会主动端口和consumer之间的channel；同理，consumer端也会每隔60秒发送一次heartbeat，3次没回复，consumer端主动断开channel；详细代码HeaderExchangeServer#startHeartbeatTimer；
2. DubboRegistryFailedRetryTimer线程，这个是连接注册中心失败时重试定时器，为了验证这个线程可以将zookeeper的ip配置错误进行验证。详细代码FailbackRegistry#FailbackRegistry；
3. DubboSaveRegistryCache，这个线程是将注册中心的节点信息保存都本地文件；
4. DubboServerHandler线程池（业务线程池）。这个是调用业务使用的线程，线程池大小默认为200，可以通过protocol标签配置threads。通过debug可以得知一次consumer发起的请求会触发业务线程池3次调用（如果线程池空， 会创建3个线程来处理，一次连接，一次业务处理，一次断开连接）；

## consumer线程

consumer端的线程运行情况和provider端类似，具体如下所示。



* zk相关线程同provider；
* netty相关，NettyClientWorker，处理IO；
* dubbo相关，DubboClientHandler，处理Dubbo业务，这个线程有保活时间（keepalive=60s，超时会被回收）

# 消息体

Dubbo的请求消息Request和响应消息Response的构造是在ExchangeCode类或者其子类中构造，具体构造是按照dubbo自定义的协议来。

Dubbo协议定义参考

<https://dubbo.apache.org/zh-cn/blog/dubbo-protocol.html>

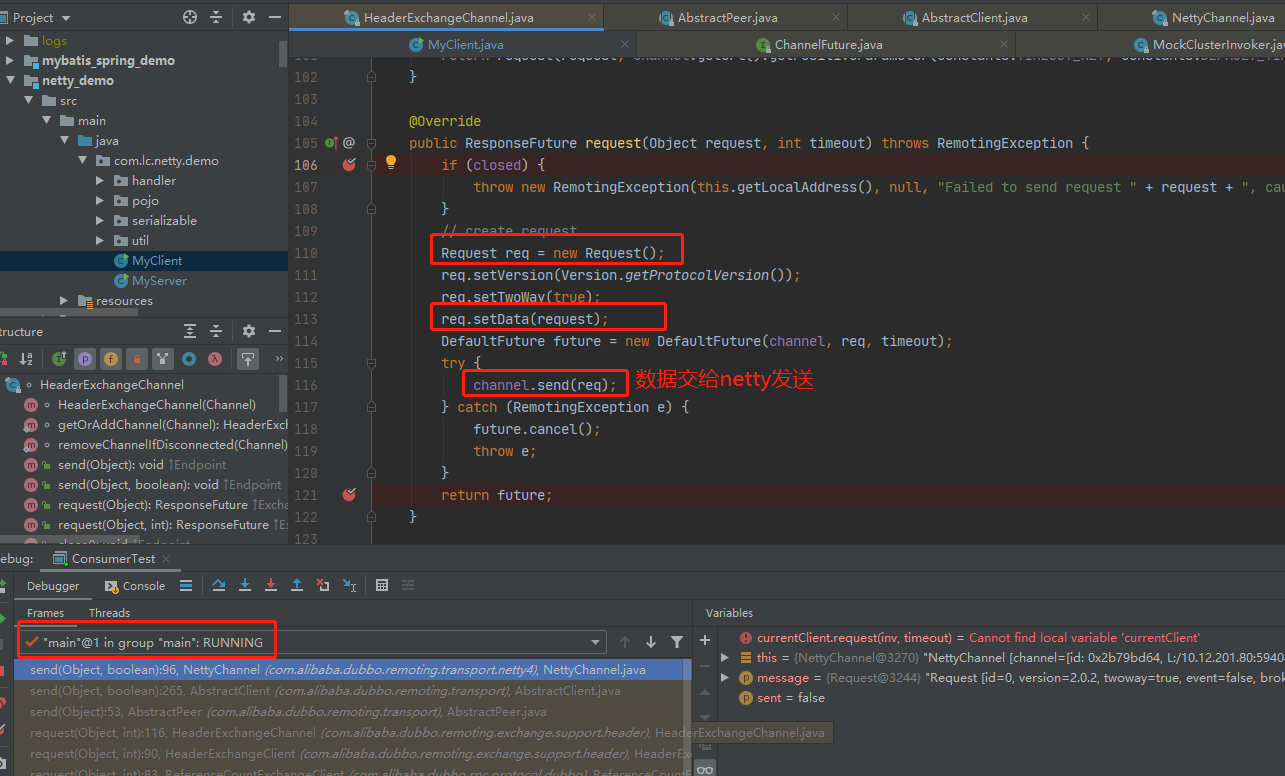
## 消息体构造

Dubbo在构造消息体时按照Big-Endian方式写入数据。其方法入口ExchangeCodec#encode，根据需要构造的消息类型调用不同的方法。

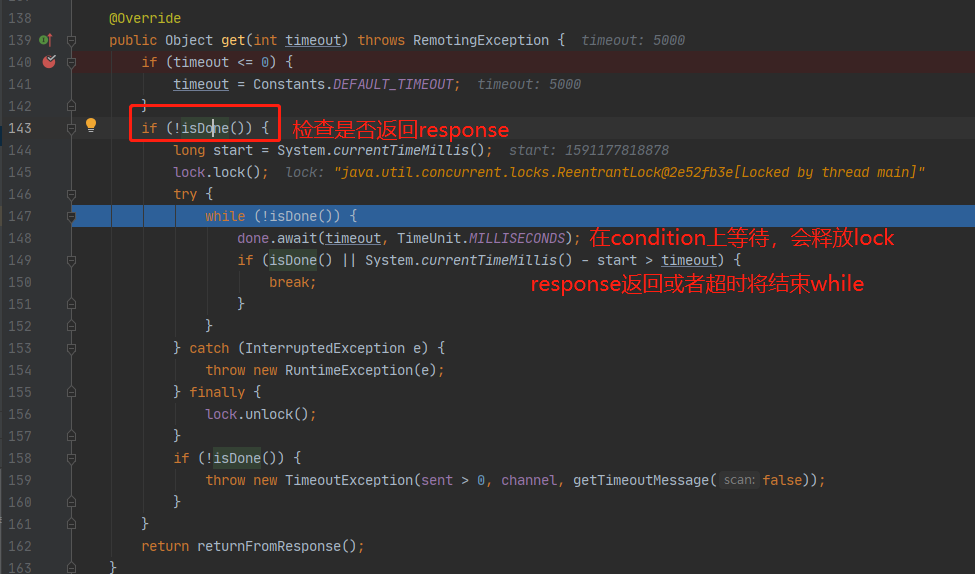
### 请求消息

入口ExchangeCodec#encodeRequest。

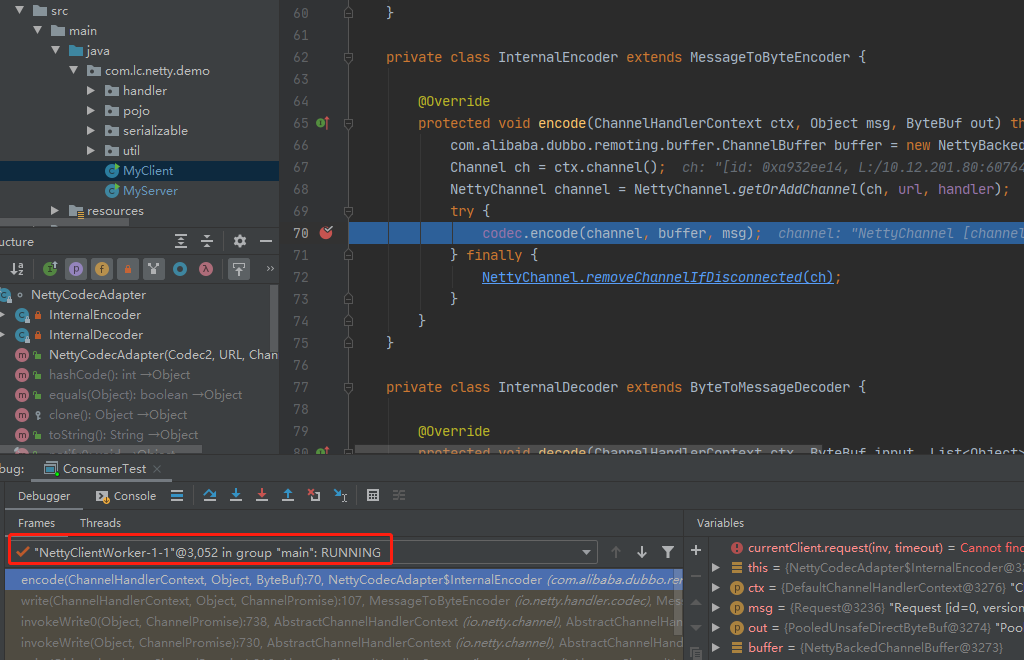
第一步：当消费者发起rpc调用时，dubbo会把本次调用封装为Request（com.alibaba.dubbo.remoting.exchange.Request）对象，创建该对象的代码位于HeaderExchangeChannel#request，该方法创建完Request对象后，会把数据交给netty进行发送，返回一个自定义的Future，最后会调用future的get进行等待。此时完成这些调用的线程是main线程；



第二步：main线程把数据发给netty后，将调用get进行等待。此处自定义实现的Future.get其思想是对共享对象response是否空进行判断，因为涉及多线程操作response，该对象设置为volatile；同时这里需要区分是同步（sync）还是异步（async），如果是异步，把消息发给netty后直接返回，否则需要等到相应结果再返回（或者超时返回异常）。



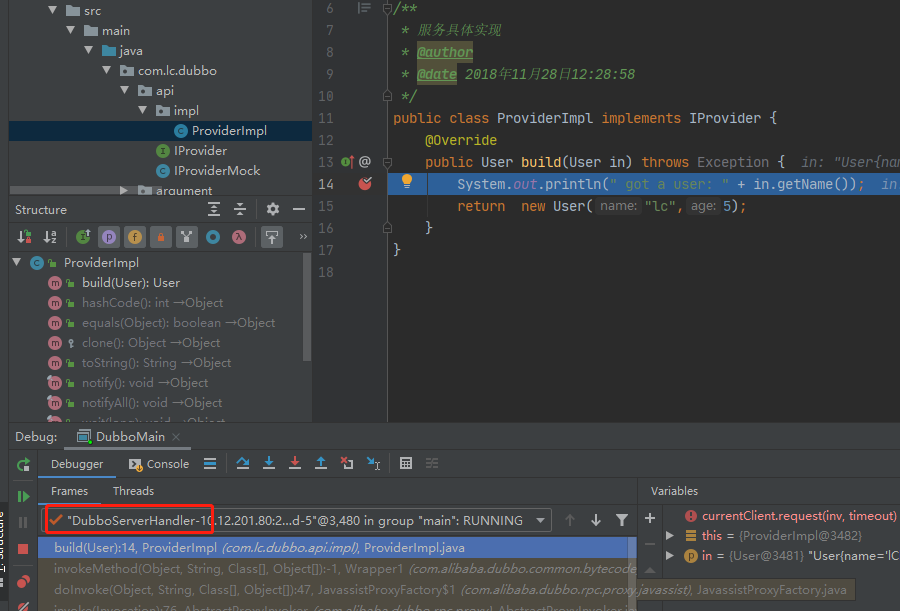
第三步：Netty中的worker线程调用dubbo提供的encode，详见NettyCodecAdapter#InternalEncoder#encode方法，最终会进入ExchangeCodec#encode方法，构造请求消息；



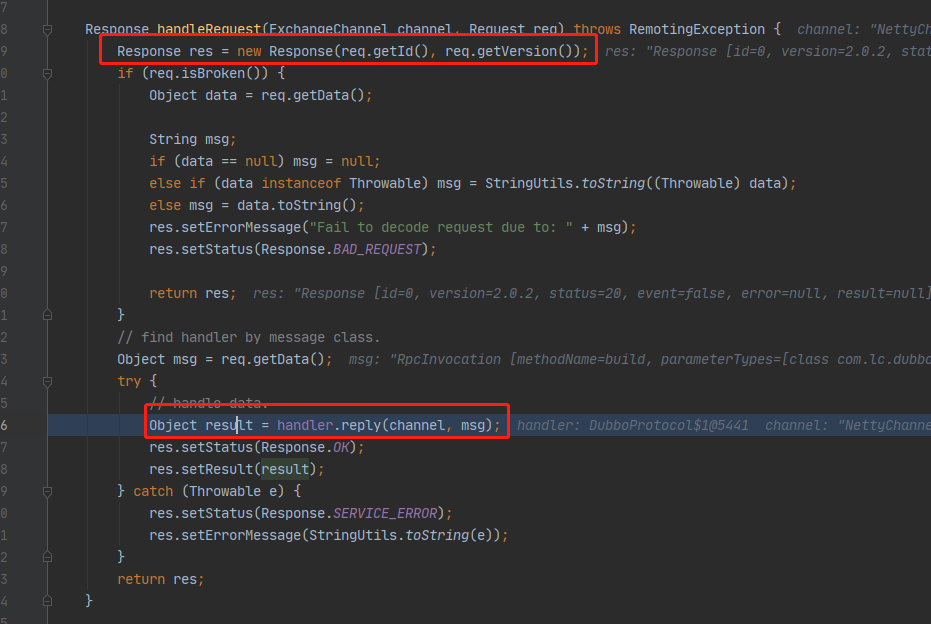
### 响应消息

Dubbo线程调用完provider提供的方法实现后，需要封装Response对象，其创建代码位于，主要步骤如下：

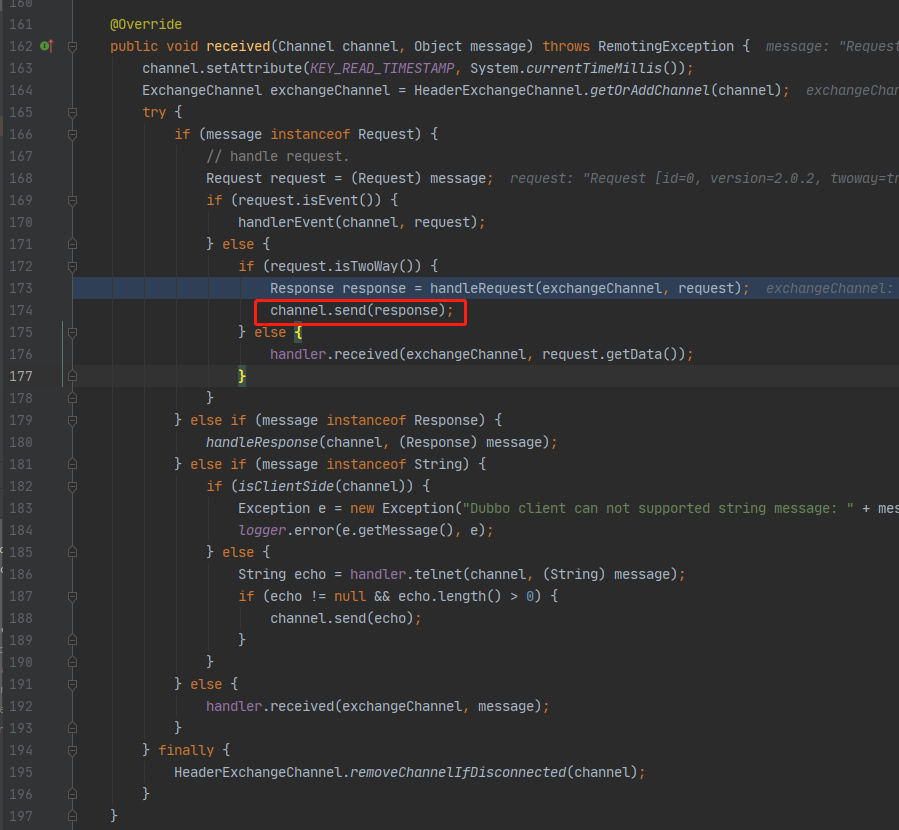
第一步，dubbo线程调用provider提供的实现



第二步将provider返回对象封装为Response，代码入口HeaderExchangeHandler#handleRequest方法



第三步，将response对象发送给netty，代码入口HeaderExchangeHandler#received方法

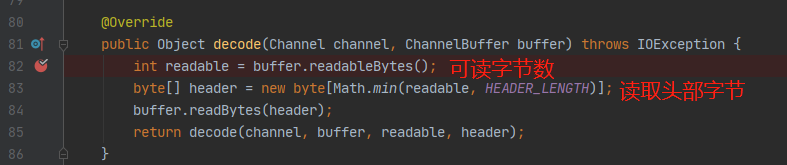


第四步（与request构造的第三步一样），Netty中的worker线程调用dubbo提供的encode，详见NettyCodecAdapter#InternalEncoder#encode方法，最终会进入ExchangeCodec#encode方法，构造相应消息。

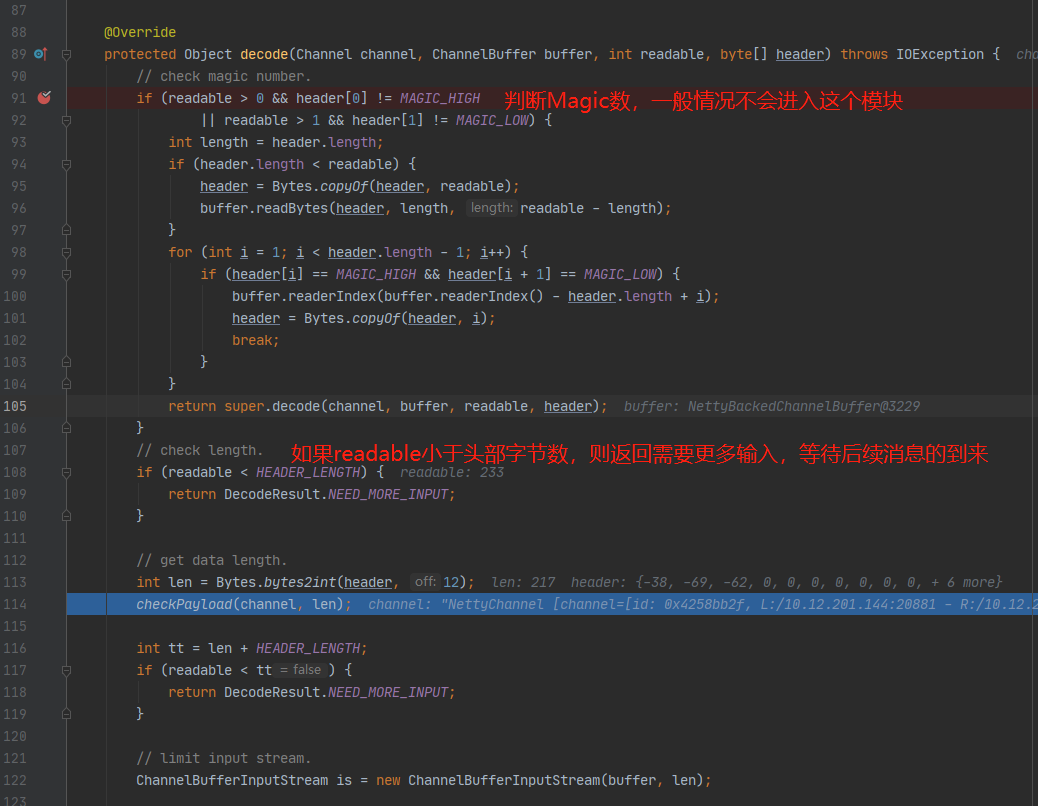
## 粘包解决

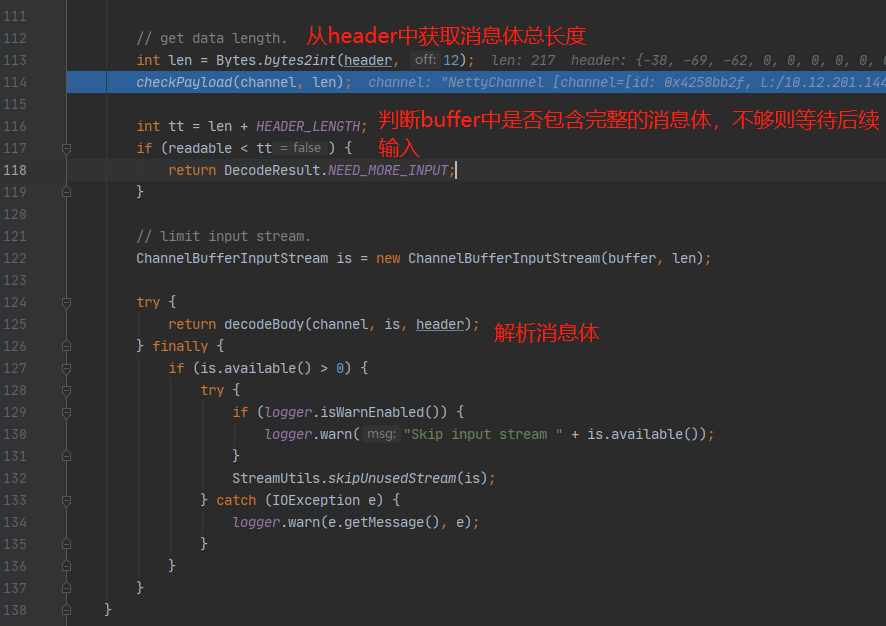
Dubbo中TCP粘包是通过采用其自定义的Codec来解决，粘包解决代码在ExchangeCodec#decode方法。

第一步从buffer中读取数据，这里需要注意的是如果buffer中数据不够16（Dubbo协议头部长度HEADER\_LENGTH）字节，则读取readable字节数；

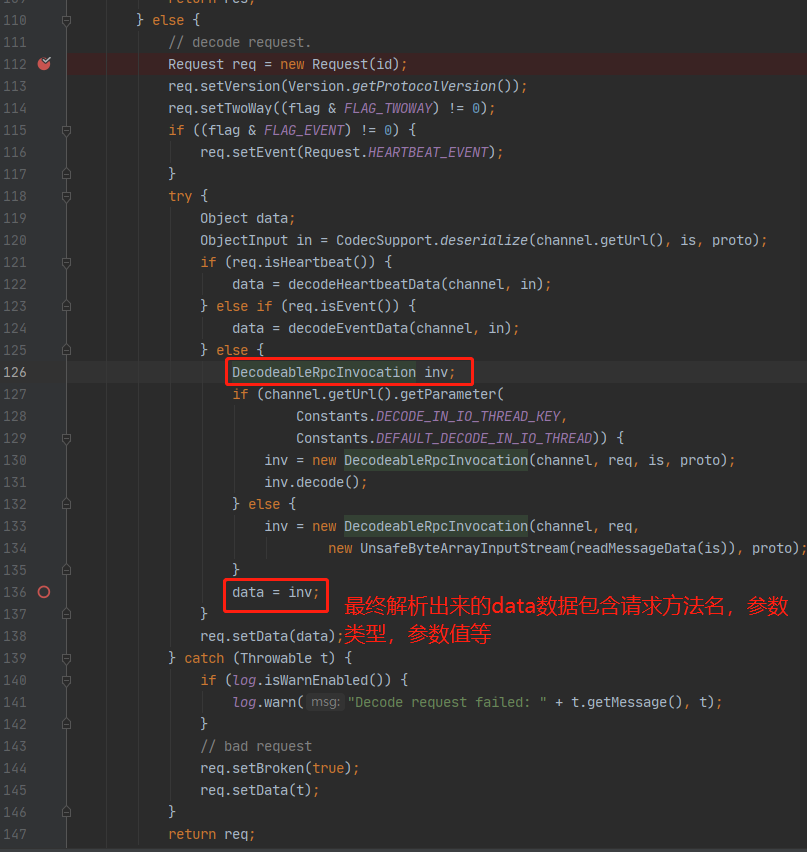


第二步判断是否读读取到完整的消息体（Request或者Response消息体）





第三步解析消息体，调用DubboCodec#decodeBody（本次以解析Request为例）



第四步，如果本次buffer中包含多个消息体，需要循环继续处理



# 调用方式

## OneWay

所谓的OneWay，是指consumer发起调用后不需要provider的响应；这种方式很少使用。

## 同步/异步

同步调用是指consumer发起接口调用请求并将request对象提交给netty后不会立即返回，而是将线程挂起（wait操作）一段时间后再返回；

异步调用是指线程将request对象提交给netty后立即返回，同时线程会通过ThreadLocal思想保存返回的Future，以便后续获取请求结果。

是否等待发送是指调用线程将request对象提交给netty后，是否等待一段时间再返回，这段等待时间主要是等待netty发送。

异步和同步参考DubboInvoker#doInvoke方法，是否等待发送参考NettyChannel#send方法。

## 同步、异步、是否等待

### 同步等待发送



### 同步不等待发送



### 异步等待发送



### 异步不等待发送



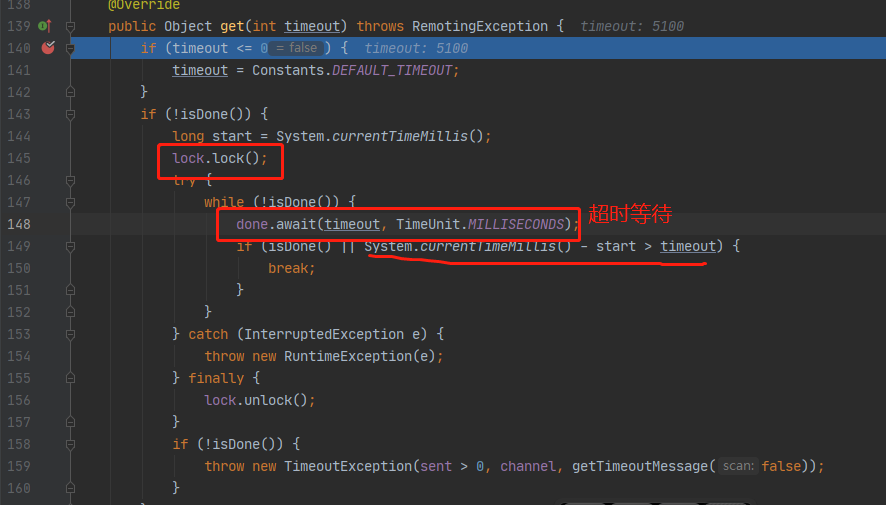
## 实现

### 同步实现

当Consumer发起接口调用时，最终会执行到DubboInvoker#doInvoke方法，如果接口调用方式配置为同步（默认是同步），则会在发起请求后返回一个DefaultFuture，同时调用该Future的get方法等待结果响应。



阅读DefaultFuture的get方法可以得知，同步的实现是通过调用Condition的await方法来完成的，同时会指定其timeout（通过<dubbo:reference>标签设置timeout，如果未设置则获取<dubbo:consumer>中的timeout）。



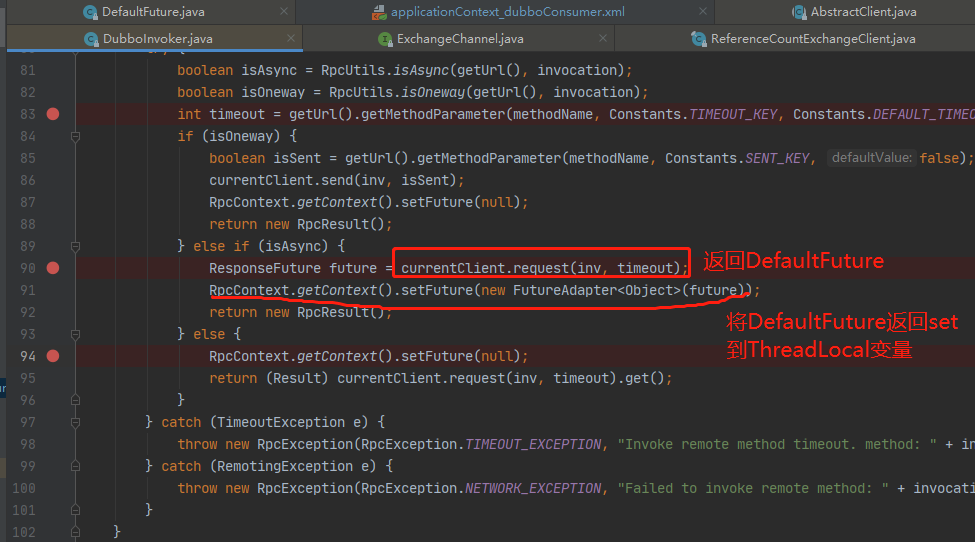
代码中的isDone就是用来判断response是否为null，不为null说明有结果返回，则可以结束超时等待并且向上反馈结果。

通过阅读源码可以知道consumer发起调用时的同步机制实现的关键技术是Condition的await，同时采用volatile来增加response的可见性。

### 异步实现

异步实现和同步实现有很多相似之处，唯一的区别是DefaultFuture的get方法是由谁来调用。在同步实现中，DefaultFuture的get方法由Dubbo框架调用，但是在异步实现中，DubboInvoker#doInvoke方法会将获取到的DefaultFutureTask存放在一个“全局变量”中，业务代码可以通过这个全局变量来获取。

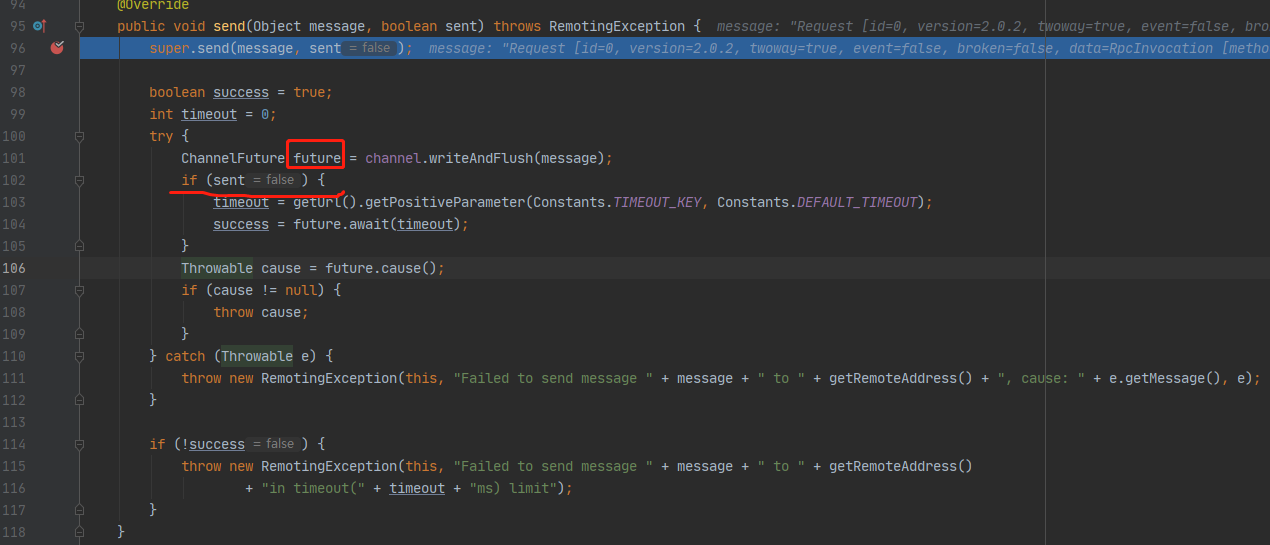
当业务代码获取的调用返回时，此时获取到的返回结果是无效的，需要手动触发获取DefaultFuture，然后调用get方法。在调用get方法之前，业务代码还可以处理其他业务逻辑，处理之后再调用get，因此达到异步调用效果。



这种方式有个问题是，如果连续发起两次异步调用请求，setFuture方法只会把最后一次返回的DefaultFuture记录下来，其他都被丢弃了。

### 是否等待实现

是否等待是在dubbo将数据提交给netty时，是否等待netty返回（或者限时等待）；其实现是通过配置<dubbo:reference>标签中的sent参数，核心代码如下NettyChannel#send，等待是通过调用netty返回的future对象的await（不是get）方法



# 集群容错

集群容错包含四个部分，分别是服务目录Directory、服务路由Route，集群Cluster和负载均衡LoadBalance。

## 服务目录Directory

参考<https://dubbo.apache.org/zh-cn/docs/source_code_guide/directory.html>

服务目录Directory主要作用是更新invoker链表，如果配置中心的服务提供者信息发送变动（增删改），RegistryDirectory会接收到通知（实现NotifyListener接口），根据节点类型更新不同的信息，比如provider节点更新invoker列表，router根据router列表。更新后的invoker会缓存到本地的map

简单来说，服务目录其实就是注册中心在本地的一个代理，将注册中心的节点信息转换为对应的invoker，router。

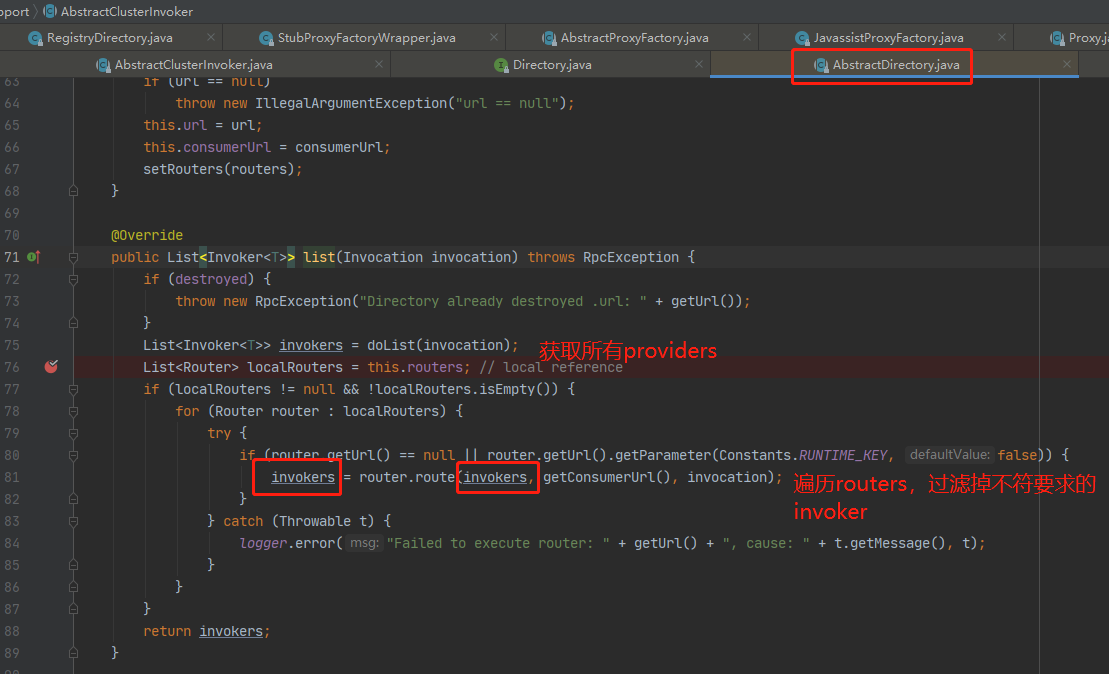
Directory提供的list方法（详见AbstractDirectory#list）则是为调用者返回当前的invoker列表，同时还会调用Router的路由规则（前提是配置了路由规则）。

## 路由Route

参考<http://dubbo.apache.org/zh-cn/blog/dubbo-loadbalance.html>

dubbo中consumer在做负载均衡前需要进行一次路由route，所谓路由，简单理解就是将请求转移到指定的机房。详细代码是AbstractClusterInvoker#invoke方法，这个方法是所有consumer调用服务前都需要执行，其中会调用AbstractDirectory#List方法，这个方法就是将从注册中心Zookeeper拉取过来的所有provider（在consumer端就是invoker），按照路由规则过滤掉不符合条件的invoker。

此处的routers就是从注册中心该接口节点下的routers节点存放的路由信息。



### 路由扩展

参考Dubbo\_Demo例子中有关路由的代码

## 集群Cluster

Dubbo提供多种Cluster，可以查看spi配置文件com.alibaba.dubbo.rpc.cluster.Cluster中的别名。

### Mock

这种Cluster是为Consumer设置的，主要是用于服务降级。如果远程调用Provider失败，并且Consumer设置了mock属性，则会降级调用Consumer本地的Mock服务。

Mock枚举值：

* false：不使用mock服务，如果远程调用失败则反馈上游系统失败；
* true：如果远程调用失败，则调用本地mock服务；
* default，通true含义；
* force：强制使用mock服务，不调用远程服务。

注意事项：

* Mock服务的包名必须和Provider的接口所在包名一样；
* Mock服务的类名必须是接口名+Mock

举例，如果接口包名和类名是：com.lc.api.provider.Provider，则mock服务的全限定名是com.lc.api.provider.ProvideMock.

获取Mock服务实例对象是在MockInvoker#getMockObject方法，其代码如下：



通过代码可以看出，Mock服务必须包含一个缺省构造方法（default constructor）。

### Failover

故障转移（failover，字面意思就是失败则跳过）。Dubbo默认的集群模式。该模式的核心代码是FailoverClusterInvoker#doInvoke方法。

结合代码可以知道doInvoke的逻辑如下：

1. 获取重试次数retry，默认重试2次，则总的是len=retry+1；
2. for i=0->len
   1. 负载均衡选择最合适的invoker（从注册中心订阅所有的provider）
   2. 调用invoker的invoke方法
      1. 成功，返回结果
      2. 不成功，判断是否超过次数，否则继续重试

上面的逻辑比较简单，容易理解，需要掌握的两个点

* 注册中心的Provider是如何订阅到的？
* 被负载均衡选中的invoker是如何生成的

## 负载均衡

Dubbo提供四种负载均衡算法。分别是：

* 加权随机负载均衡
* 加权最小活跃数负载均衡
* 一致性hash负载均衡
* 加权轮训负载均衡

### RandomLoadBalance

算法核心思想是累加各个接口的权重，然后随机生成一个值，判断该值所处区间。算法伪代码：

for i=0:N

sum+=weight[i];

random = random(sum);//生成0到sum区间的随机数

for i=0:N

if(random-weight[i]<=0)

break;

return node[i];

### LeastActiveLoadBalance

Dubbo会为每个服务提供者统计当前服务调用数，当一个请求到达，该服务提供者的活跃数+1，请求结束后活跃数-1，LeastActiveLoadBalance就是依照活跃判断该使用哪个服务提供者。加权是为了解决如果有多个相同的最小活跃数，则在这部分提供者中执行加权RandomLoadBalance。

筛选方法就是逐个invoker进行遍历，找出最小的invoker集合，如果有多个invoker可供选择（也就是最小活跃数都一样），则根据weight来筛选。

### RoundRobinLoadBalance

没有加权的轮询是按照所有候选者逐个选中并使用，如果候选者存在差异，可以为性能高的候选者赋更高的权重，性能较差的候选者赋较小的权重。Dubbo中的加权RoundRobinLoadBalance是在轮询的基础上增加权重来筛选候选者，算法的具体执行过程可以参照如下例子：

假设三个候选者A,B,C的权重分别为3,5,2，权重构成一个序列[3,5,2]，同时总权重为10：

Round 1：选择A，同时A的剩余权重减1，此时序列变为了[2,5,2];

Round 2：选择B，同时B的剩余权重减1，此时序列变为了[2,4,2];

Round 3：选择C，同时C的剩余权重减1，此时序列变为了[2,4,1];

Round 4：选择A，同时A的剩余权重减1，此时序列变为了[1,4,1];

Round 5：选择B，同时B的剩余权重减1，此时序列变为了[1,3,1];

Round 6：选择C，同时C的剩余权重减1，此时序列变为了[1,3,0];

Round 7：选择A，同时A的剩余权重减1，此时序列变为了[0,3,0];

Round 8：选择B，同时B的剩余权重减1，此时序列变为了[0,2,0];

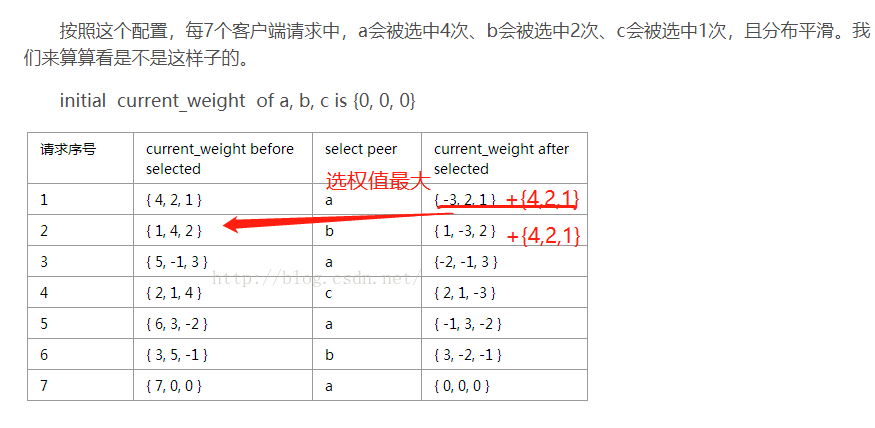
Round 9：选择B，同时B的剩余权重减1，此时序列变为了[0,1,0];

Round 10：选择B，同时B的剩余权重减1，此时序列变为了[0,0,0]

经过10轮，A被选中了3次，B被选中5次，C被选中2次。如果按照一般的轮询，10轮会选择A4次，B3次，C3次。

可以将三个服务提供者构成一个圈，每选中一次则移动一次，并且权值减1，如果减为0则移出圈。

最理想的是采用平滑加权轮询，其算法参考其中一个实例：



### ConsistentHashLoadBalance

Dubbo中一致性哈希的负载均衡实现思路如下：

1. 任意一个服务提供者invoker，默认在hash环上配置160个虚拟节点；
2. 某一个invoker，其160个虚拟节点是按照“ip+port+自增编号”计算hash
3. 某个invoker，将其hash，invoker键值对存放在TreeMap中；
4. 某个消费者在doSelect时，首先计算消费者的hash，然后从TreeMap中筛选出key大于消费者hash的所有invoker（通过TreeMap#tailMap(key)获取）；
5. 从tailMap返回的集合中选出第一个invoker来服务。

如上是dubbo中一致性哈希的基本思路，那如果invoker数量发送变化之后该如何处理呢。

Dubbo中会将invokers这个集合整体计算出一个一致性hash，每次都会利用invokers计算一致性hash，如果和之前的不一致（invokers列表发生变化），则会重新按照上面的思路构造hash一致性环。

# 序列化和反序列化

Dubbo中处理序列化的地方时DubboCodec和ExchangeCodec两个类中。序列化的具体实现在package com.alibaba.dubbo.common.serialize中。

序列化是将请求/响应的header和body序列成可持久化存储或者网络传输的数据；

反序列就是和序列化相反的过程。

序列化的具体实现类采用的spi（类似于工厂模式，通过传入的命令选择具体的实现），默认是hessian2，可以在配置文件中修改序列化方式

<dubbo:protocol name="dubbo" port="20881" serialization="fastjson"/>，比如这个配置就采用的fastjson进行序列化。

现有的序列化方式中，hessian2是性能最好和最可靠的序列化方式，dubbo官方也指出后续会增加更高效更可靠的序列方式fst，kryo等。

参考文档：

<https://dubbo.apache.org/zh-cn/docs/user/serialization.html>

# 模块

## 注册

参考：<https://juejin.im/post/5d5cc544e51d4561df7805a9>

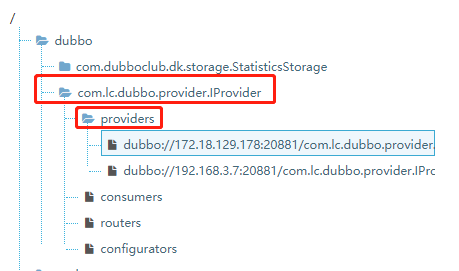
关键字：

AbstractRegistry—内存缓存和文件缓存，文件名格式dubbo-registry-${application.name}-${register\_address}.cache

FailbackRegistry—失败重试，重试机制就是一个定时线程池不断遍历配置的注册中心地址进行注册

### Zookeeper

在学习以Zookeeper作为注册中心之前，首先以一个例子开头，以此对zookeeper有个直观感受。同一个服务提供者，部署在两台服务器上并以同一个zookeeper作为注册中心时，zookeeper的树形结构如下所示：



可以看出dubbo提供中在注册服务时，提供的内网ip，不会提供公网ip。

# 参考

官网https://dubbo.incubator.apache.org/

API官网<https://dubbo.incubator.apache.org/books/dubbo-user-book-en/preface/>

Dubbo介绍<http://www.baeldung.com/dubbo>

Zookeeper简介<https://www.w3cschool.cn/zookeeper/zookeeper_installation.html>

博客<https://juejin.im/post/5bb18c8e5188255c6140de76>

<https://segmentfault.com/a/1190000016741532>（推荐）