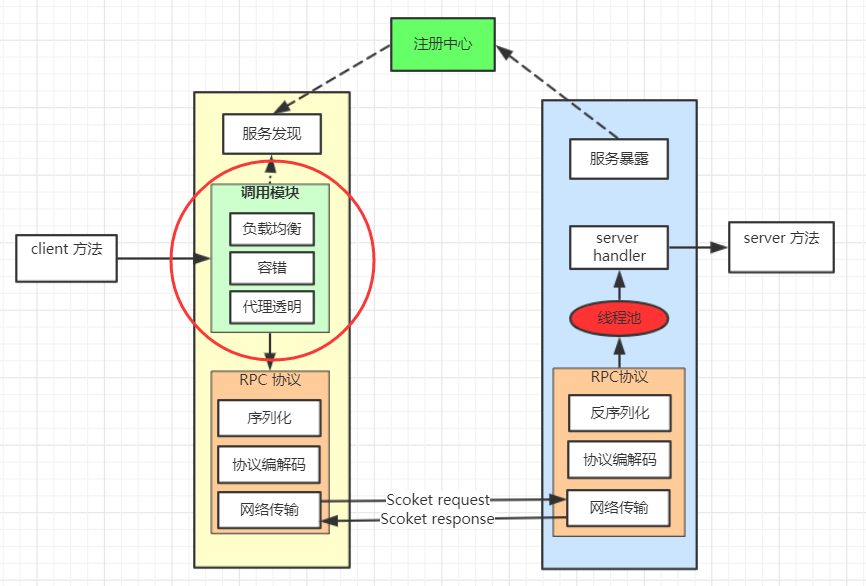
一、Dubbo 调用模块基本组成

**Dubbo调用模块概述：**

dubbo调用模块核心功能是发起一个远程方法的调用并顺利拿到返回结果，其体系组成如下：

1. **透明代理：**通过动态代理技术，屏蔽远程调用细节以提高编程友好性。
2. **负载均衡：**当有多个提供者是，如何选择哪个进行调用的负载算法。
3. **容错机制：**当服务调用失败时采取的策略
4. **调用方式：**支持同步调用、异步调用



透明代理：

透明代理的意思的：dubbo对调用远程服务模块进行了封装，让使用者像调用本地服务一样使用即可。

参见源码：

com.alibaba.dubbo.config.ReferenceConfig#createProxy

com.alibaba.dubbo.common.bytecode.ClassGenerator

com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory

**负载均衡**

Dubbo 目前官方支持以下负载均衡策略：

1. **随机**(random)：按权重设置随机概率。此为默认算法.
2. **轮循**(roundrobin):按公约后的权重设置轮循比率。
3. **最少活跃调用数**(leastactive):相同活跃数的随机，活跃数指调用前后计数差。活跃是指正在处理的请求的个数。越少的就会被调用。比如A和B同时被调用，这时A和B都是1，B被调用完，变成0。又来请求，那么A还是1，b是0，那么就调用B。目的是让更慢的机器收到更少的请求，
4. **一致性Hash**(consistenthash ):相同的参数总是发到同一台机器

设置方式支持如下四种方式设置，优先级由低至高

<!-- 服务端级别-->

<dubbo:service interface="..." loadbalance="roundrobin" />

<!-- 客户端级别-->

<dubbo:reference interface="..." loadbalance="roundrobin" />

<!-- 服务端方法级别-->

<dubbo:service interface="...">

    <dubbo:method name="..." loadbalance="roundrobin"/>

</dubbo:service>

<!-- 客户端方法级别-->

<dubbo:reference interface="...">

    <dubbo:method name="..." loadbalance="roundrobin"/>

</dubbo:reference>

#TODO 一至性hash 演示

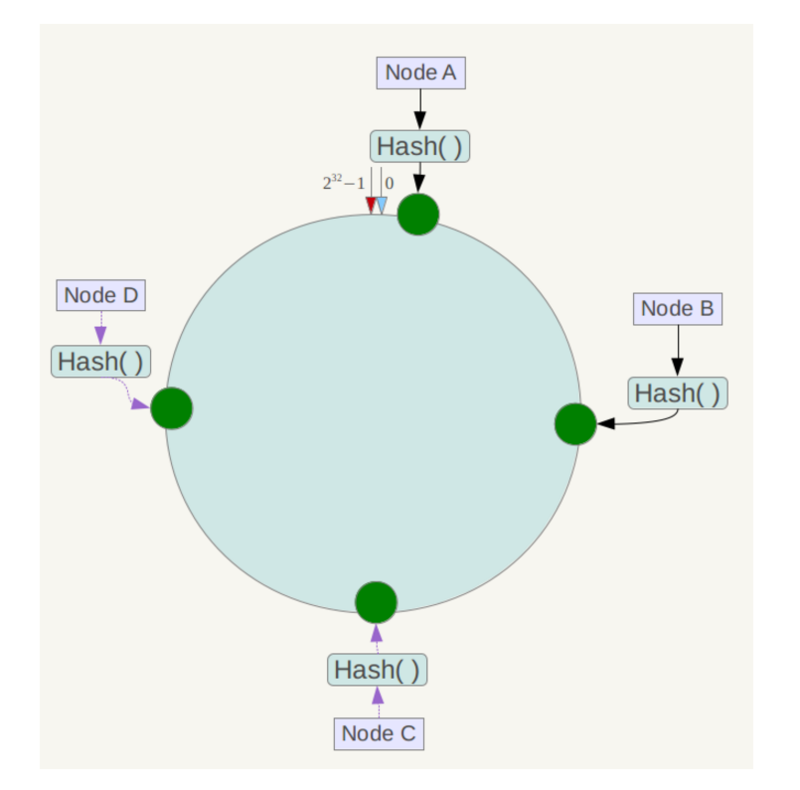
* 配置loadbalance
* 配置需要hash 的参数与虚拟节点数
* 发起远程调用

一致性hash和简单hash的区别：

简单hash时，在动态扩展或服务宕机时，取模的时候，会使得整个系统都失效。因为正常时hash的结果和故障时hash的结果时不一致的。

一致性hash，如果动态扩展和服务宕机时，故障只会出现在一小部分。

一至性hash 算法详解：



**容错**

Dubbo 官方目前支持以下容错策略：

失败：网络，调用超时，业务异常（自己写的代码抛出异常）不算。

1. **失败自动切换：**调用失败后基于retries=“2” 属性重试其它服务器。总共调用三次（2+开始的一次）
2. **快速失败：**快速失败，只发起一次调用，失败立即报错。
3. **勿略失败：**失败后勿略，不抛出异常给客户端。
4. **失败重试：**失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作
5. **并行调用:**只要一个成功即返回，并行调用指定数量机器，可通过 forks="2" 来设置最大并行数。
6. **广播调用：**广播调用所有提供者，逐个调用，任意一台报错则报错。场景：如果是对所有的服务提供者进行统一的操作，比如清除每个程序的缓存标记。

设置方式支持如下两种方式设置，优先级由低至高

<!--

Failover 失败自动切换 retries="1" 切换次数

Failfast 快速失败

Failsafe 勿略失败

Failback 失败重试，5秒后仅重试一次

Forking 并行调用  forks="2" 最大并行数

Broadcast 广播调用

-->

<dubbo:service interface="..." cluster="broadcast" />

<dubbo:reference interface="..." cluster="broadcast"/ >

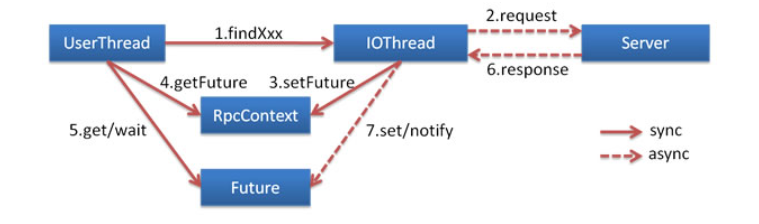
注：容错机制 在基于 API设置时无效 如   referenceConfig.setCluster("failback"); 经测试不启作用

**异步调用**

异步调用是指发起远程调用之后获取结果的方式。

1. 同步等待结果返回（默认）
2. 异步等待结果返回
3. 不需要返回结果

Dubbo 中关于异步等待结果返回的实现流程如下图：



异步调用配置:

<dubbo:reference id="asyncDemoService" interface="com.tuling.teach.service.async.AsyncDemoService">

                 <!-- 异步调async：true 异步调用 false 同步调用-->

    <dubbo:method name="sayHello1" async="false"/>

    <dubbo:method name="sayHello2" async="false"/>

     <dubbo:method name="notReturn" return="false"/>

</dubbo:reference>

注：在进行异步调用时 容错机制不能为 cluster="forking" 或 cluster="broadcast"

**异步获取结果演示：**

* 编写异步调用代码
* 编写同步调用代码
* 分别演示同步调用与异步调用耗时

*异步调用结果获取Demo*

demoService.sayHello1("han");

Future<Object> future1 = RpcContext.getContext().getFuture();

demoService.sayHello2("han2");

Future<Object> future2 = RpcContext.getContext().getFuture();

Object r1 = null, r2 = null;

// wait 直到拿到结果 获超时

r1 = future1.get();

// wait 直到拿到结果 获超时

r2 = future2.get();

注意：如果是异步调用，调用完，要立即getFuture()，而不是继续调用。否则第二次调用返回的future会覆盖第一次调用的future。

因为这些future都是保存在RPCContext中，而RPCContext保存在线程的ThreadLocal里面的，而RPCContext中只有一个future，会被不断地覆盖。

|  |
| --- |
| RpcContext.getContext().getFuture();    public static RpcContext getContext() {  return LOCAL.get();  }    private static final ThreadLocal<RpcContext> LOCAL = new ThreadLocal<RpcContext>() {  @Override  protected RpcContext initialValue() {  return new RpcContext();  }  }; |

如果是异步，又是return false。

在DubboInvoker的doInvoke方法中有：

|  |
| --- |
| if (isOneway) {//是否是return false  boolean isSent = getUrl().getMethodParameter(methodName, Constants.SENT\_KEY, false);  currentClient.send(inv, isSent);  RpcContext.getContext().setFuture(null);  return new RpcResult();  } else if (isAsync) {//是否是异步  ResponseFuture future = currentClient.request(inv, timeout);  RpcContext.getContext().setFuture(new FutureAdapter<Object>(future));  return new RpcResult();  } else {//同步  RpcContext.getContext().setFuture(null);  return (Result) currentClient.request(inv, timeout).get();  } |

**过滤器**

**类似于 WEB 中的Filter ，Dubbo本身提供了Filter 功能用于拦截远程方法的调用。其支持自定义过滤器与官方的过滤器使用：**

#TODO 演示添加日志访问过滤:

<dubbo:provider  filter="accesslog" accesslog="logs/dubbo.log"/>

以上配置 就是 为 服务提供者 添加 日志记录过滤器， 所有访问日志将会集中打印至 accesslog 当中

二 、Dubbo 调用非典型使用场景----略

**泛化提供&引用**

**泛化提供**

是指不通过接口的方式直接将服务暴露出去。通常用于Mock框架或服务降级框架实现。

#TODO 示例演示

public static void doExportGenericService() {

    ApplicationConfig applicationConfig = new ApplicationConfig();

    applicationConfig.setName("demo-provider");

    // 注册中心

    RegistryConfig registryConfig = new RegistryConfig();

    registryConfig.setProtocol("zookeeper");

    registryConfig.setAddress("192.168.0.147:2181");

    ProtocolConfig protocol=new ProtocolConfig();

    protocol.setPort(-1);

    protocol.setName("dubbo");

    GenericService demoService = new MyGenericService();

    ServiceConfig<GenericService> service = new ServiceConfig<GenericService>();

    // 弱类型接口名

    service.setInterface("com.tuling.teach.service.DemoService");

    // 指向一个通用服务实现

    service.setRef(demoService);

    service.setApplication(applicationConfig);

    service.setRegistry(registryConfig);

    service.setProtocol(protocol);

    // 暴露及注册服务

    service.export();

}

**泛化引用**

是指不通过常规接口的方式去引用服务，通常用于测试框架。

ApplicationConfig applicationConfig = new ApplicationConfig();

applicationConfig.setName("demo-provider");

// 注册中心

RegistryConfig registryConfig = new RegistryConfig();

registryConfig.setProtocol("zookeeper");

registryConfig.setAddress("192.168.0.147:2181");

// 引用远程服务

ReferenceConfig<GenericService> reference = new ReferenceConfig<GenericService>();

// 弱类型接口名

reference.setInterface("com.tuling.teach.service.DemoService");

// 声明为泛化接口

reference.setGeneric(true);

reference.setApplication(applicationConfig);

reference.setRegistry(registryConfig);

// 用com.alibaba.dubbo.rpc.service.GenericService可以替代所有接口引用

GenericService genericService = reference.get();

Object result = genericService.$invoke("sayHello", new String[]{"java.lang.String"}, new Object[]{"world"});

**隐示传参**

是指通过非常方法参数传递参数，类似于http 调用当中添加cookie值。通常用于分布式追踪框架的实现。使用方式如下 ：

//客户端隐示设置值

RpcContext.getContext().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐

//服务端隐示获取值

String index = RpcContext.getContext().getAttachment("index");

**令牌验证**

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者

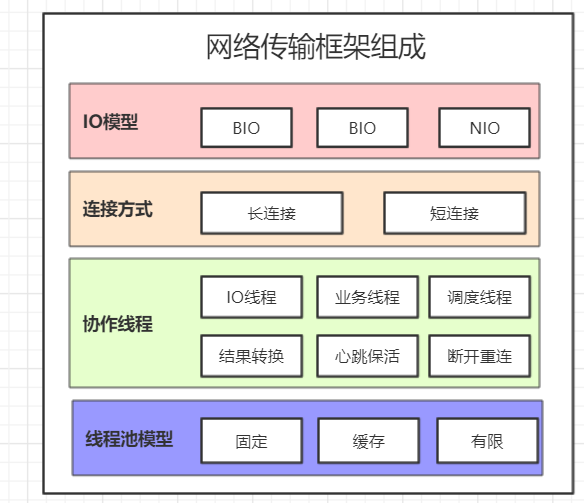


使用：

<!--随机token令牌，使用UUID生成--><dubbo:provider interface="com.foo.BarService" token="true" />

三、调用通信内部实现源码分析

**网络传输的实现组成**



1. **IO模型：**
   1. BIO 同步阻塞
   2. NIO 同步非阻塞
   3. AIO 异步非阻塞
2. **连接模型：**
   1. 长连接
   2. 短连接
3. **线程分类：**
   1. IO线程
   2. 服务端业务线程
   3. 客户端调度线程
   4. 客户端结果exchange线程。
   5. 保活心跳线程
   6. 重连线程
4. **线程池模型：**
   1. 固定数量线程池
   2. 缓存线程池
   3. 有限线程池

**Dubbo 长连接实现与配置**

**初始连接：**

引用服务增加提供者==>获取连接===》是否获取共享连接==>创建连接客户端==》开启心跳检测状态检查定时任务===》开启连接状态检测

源码见：com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol#getClients

**心跳发送：**

在创建一个连接客户端同时也会创建一个心跳客户端，客户端默认基于60秒发送一次心跳来保持连接的存活，可通过 heartbeat  设置。

源码见：*com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeClient#startHeatbeatTimer*

**断线重连：**

每创建一个客户端连接都会启动一个定时任务每两秒中检测一次当前连接状态，如果断线则自动重连。

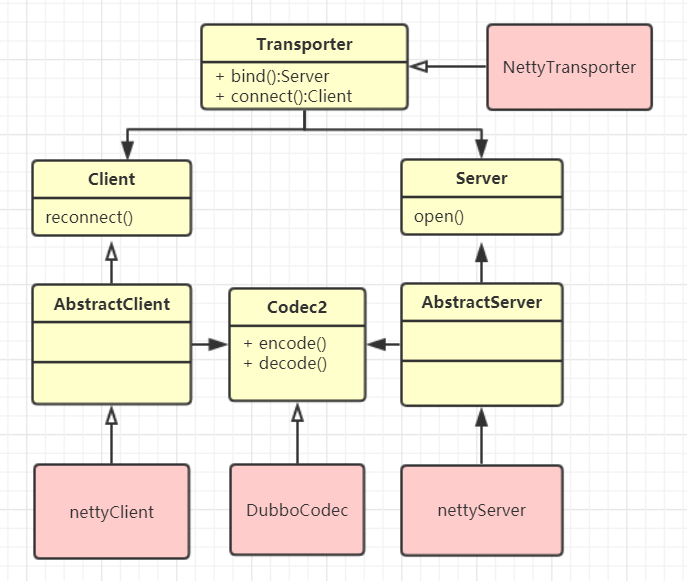
源码见：com.alibaba.dubbo.remoting.transport.AbstractClient#initConnectStatusCheckCommand

**连接销毁:**

基于注册中心通知，服务端断开后销毁

源码见：com.alibaba.dubbo.remoting.transport.AbstractClient#close()

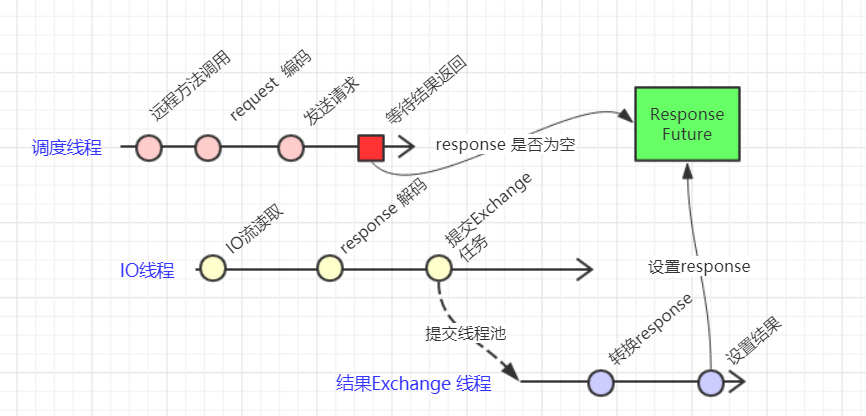
**dubbo传输uml类图:**



**Dubbo 传输协作线程**

1. **客户端调度线程**：用于发起远程方法调用的线程。
2. **客户端结果Exchange线程：**当远程方法返回response后由该线程填充至指定ResponseFuture，并叫醒等待的调度线程。
3. **客户端IO线程：**由传输框架实现，用于request 消息流发送、response 消息流读取与解码等操作。
4. **服务端IO线程**：由传输框架实现，用于request消息流读取与解码 与Response发送。
5. **业务执行线程：**服务端具体执行业务方法的线程

**客户端线程协作流程：**



1. **调度线程**
   1. 调用远程方法
   2. 对request 进行协议编码
   3. 发送request 消息至IO线程
   4. 等待结果的获取
2. **IO线程**
   1. 读取response流
   2. response 解码
   3. 提交Exchange 任务
3. **Exchange线程**
   1. 填写response值 至 ResponseFuture
   2. 唤醒调度线程，通知其获取结果

调用调试：

客户端的执行线程:

1、业务线程

1) DubboInvoker#doInvoke(隐示传公共参数、获取客户端、异步、单向、同步（等待返回结果）)

2)AbstractPeer#send// netty Client客户端发送消息 写入管道

3)DubboCodec#encodeRequestData // Request 协议编码

2、IO线程

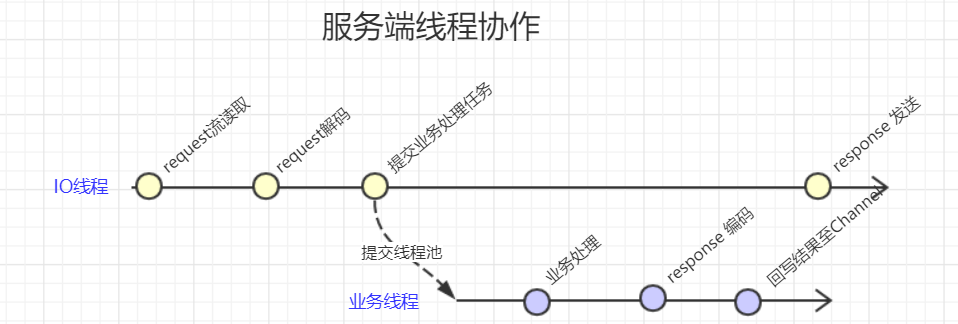
DubboCodec#decodeBody //Response解码

AllChannelHandler#received //// 派发消息处理线程

3、调度线程

DefaultFuture#doReceived // 设置返回结果

**服务端线程协作：**



1. **IO线程：**
   1. request 流读取
   2. request 解码
   3. 提交业务处理任务
2. **业务线程：**
   1. 业务方法执行
   2. response 编码
   3. 回写结果至channel

**线程池**

1. **fixed：**固定线程池,此线程池启动时即创建固定大小的线程数，不做任何伸缩，
2. **cached：**缓存线程池,此线程池可伸缩，线程空闲一分钟后回收，新请求重新创建线程
3. **Limited：**有限线程池,此线程池一直增长，直到上限，增长后不收缩。