https://www.cnblogs.com/ASPNET2008/p/6491427.html

1. [**dubbo源码分析（一）**](http://www.cnblogs.com/ASPNET2008/p/6491427.html)

**阅读源码的作用**

* 提取设计思路，增强设计能力
* 理解运行机制，便于快速解决问题以及功能扩展

**常见有关dubbo的问题**

* dubbo的负载均衡是在哪个组件中处理的？
* dubbo默认的负载均衡算法是什么？
* 如果注册中心挂掉了客户端是否能够继续调用dubbo？
* 一个请求从调用端到服务端的处理流程是什么？

如果你有仔细的研读dubbo的开发文档，大部分问题都是有答案的，为了进一步了解细节就有必要对源码进行了解。

**源码分析计划**

使用dubbo时间也有快两年时间了，虽然官方的dubbo已经停止维护，但也有其它组织在继续维护：

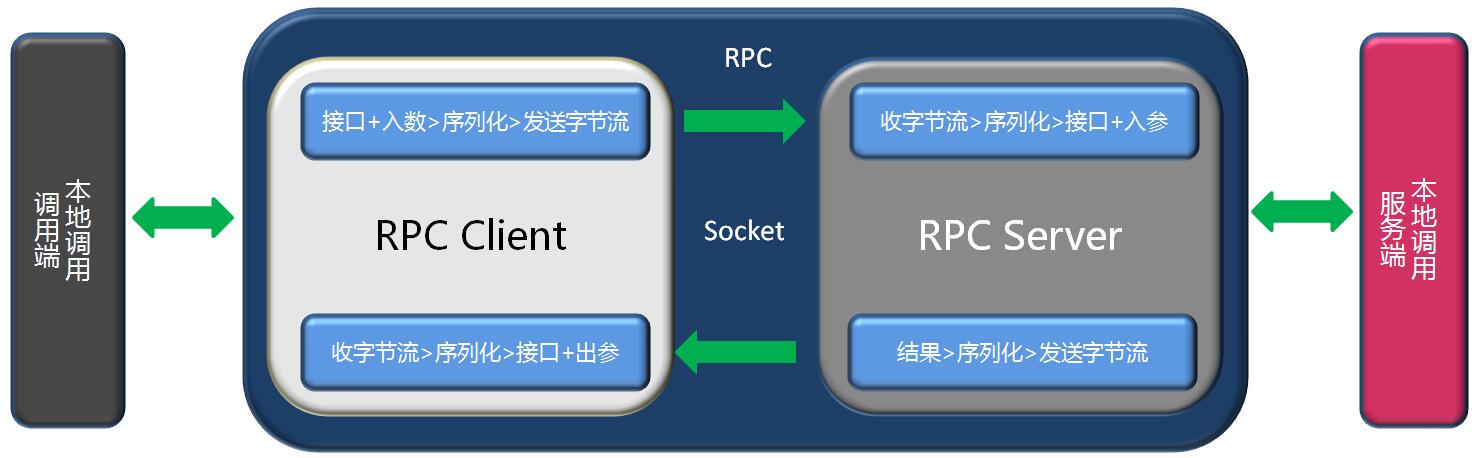
* dubbox，是当当维护的
* 还有一个是基于jdk8版本的

一直想看看dubbo的源码，但没有集中时间来学习。最近利用了一部分时间所以将我的学习经历记录下来，主要从这几步来展开，只想搞清楚大体流程，对于其它一些细节我目前并不太关注，比如序列化，线程池，集群，上下文，异步回调等。

* RPC框架的简易结构
* dubbo客户端的初始化
* dubbo服务端的初始化
* dubbo客户端处理请求流程
* dubbo服务端处理请求流程

**RPC简易结构**

为了更加清楚的看清楚RPC的结构，主要看以下几个核心组件即可，其余的组件都是围绕它们来完成。



* 序列化组件，因为需要远程调用，所以需要将参数以二进制流的形式进行编码操作
* 发送组件，将二进制流发送到服务端
* 接收组件，接收调用端发送的二进制流以及接收服务端返回的二进制流

**dubbo客户端的初始化**

从这开始来实际看下dubbo的工作流程。RPC的亮点在于将远程调用的细节隐藏起来，使得调用远程服务像调用本地服务一样简单，而实现上面的功能就是代理。

1. **代理的作用**

RPC框架隐藏了具体的实现细节，客户端通过调用特定的代理类来访问远程服务，下面是一个dubbo远程接口的本地引用。

<dubbo:reference check="false"

timeout="200000"

interface="com.product.core.service.ProductFacadeService"

id="productFacadeService">

ReferenceConfig这个类的createProxy是用来生成远程服务的本地代理，最终交给RegistryProtocol来处理。两处核心代码：

* refprotocol.refer,与注册中心相关
* proxyFactory.getProxy，获取代理

private T createProxy(Map<String, String> map) {

//...

if (isJvmRefer) {

//...

} else {

//...

if (urls.size() == 1) {

invoker = refprotocol.refer(interfaceClass, urls.get(0));

} else {

//...

}

}

//...

// 创建服务代理

return (T) proxyFactory.getProxy(invoker);

}

本地直连不走上面的逻辑

1. **注册中心**

由于服务注册到ZK，所以调用端要想调用服务端需要取得服务的注册信息然后建立网络连接。

private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {

RegistryDirectory directory = new RegistryDirectory(type, url);

directory.setRegistry(registry);

directory.setProtocol(this.protocol);

URL subscribeUrl = new URL("consumer", NetUtils.getLocalHost(), 0, type.getName(), directory.getUrl().getParameters());

if(!"\*".equals(url.getServiceInterface()) && url.getParameter("register", true)) {

registry.register(subscribeUrl.addParameters(new String[]{"category", "consumers", "check", String.valueOf(false)}));

}

directory.subscribe(subscribeUrl.addParameter("category", "providers,configurators,routers"));

return cluster.join(directory);

}

上面代码两个作用：

* 将调用端以消费者的身份进行信息注册
* 监听注册中心的信息变化以刷新本地对服务注册信息的缓存

由于有服务端信息的本地缓存，所以当注册中心挂掉后调用端依然能够工作，也就是说调用端不是强依赖服务端。

1. **连接服务端**

上面的subscribe方法会监听注册中心的变化，当获取到服务注册信息后会触发ProtocolListenerWrapper的refer方法：

public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {

return (Invoker)("registry".equals(url.getProtocol())?this.protocol.refer(type, url):  
  
new ListenerInvokerWrapper(this.protocol.refer(type, url),   
  
Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(InvokerListener.class).getActivateExtension(url, "invoker.listener"))));

}

上面的this.protocol就是DubboProtocol这个类，此类的其它方法暂时先不关注，只看refer。

public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {

this.optimizeSerialization(url);

DubboInvoker invoker = new DubboInvoker(serviceType, url, this.getClients(url), this.invokers);

this.invokers.add(invoker);

return invoker;

}

getClients负责创建链接以供调用端调用服务端方法时使用。

private ExchangeClient[] getClients(URL url) {

boolean service\_share\_connect = false;

int connections = url.getParameter("connections", 0);

if(connections == 0) {

service\_share\_connect = true;

connections = 1;

}

ExchangeClient[] clients = new ExchangeClient[connections];

for(int i = 0; i < clients.length; ++i) {

if(service\_share\_connect) {

clients[i] = this.getSharedClient(url);

} else {

clients[i] = this.initClient(url);

}

}

return clients;

}

1. **创建代理**

回到ReferenceConfig的createProxy方法，最后会调用proxyFactory.getProxy，此方法最终会调用JavassistProxyFactory的getProxy，一看javassist就知道是利用字节码来实现代理功能。

public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {

return Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandler(invoker));

}

Proxy，这个类的实现比较复杂就不详细分析了，得到了proxy将它放在容器中，当调用端调用服务端代码时就有了实例跟本地的实例没什么区别。

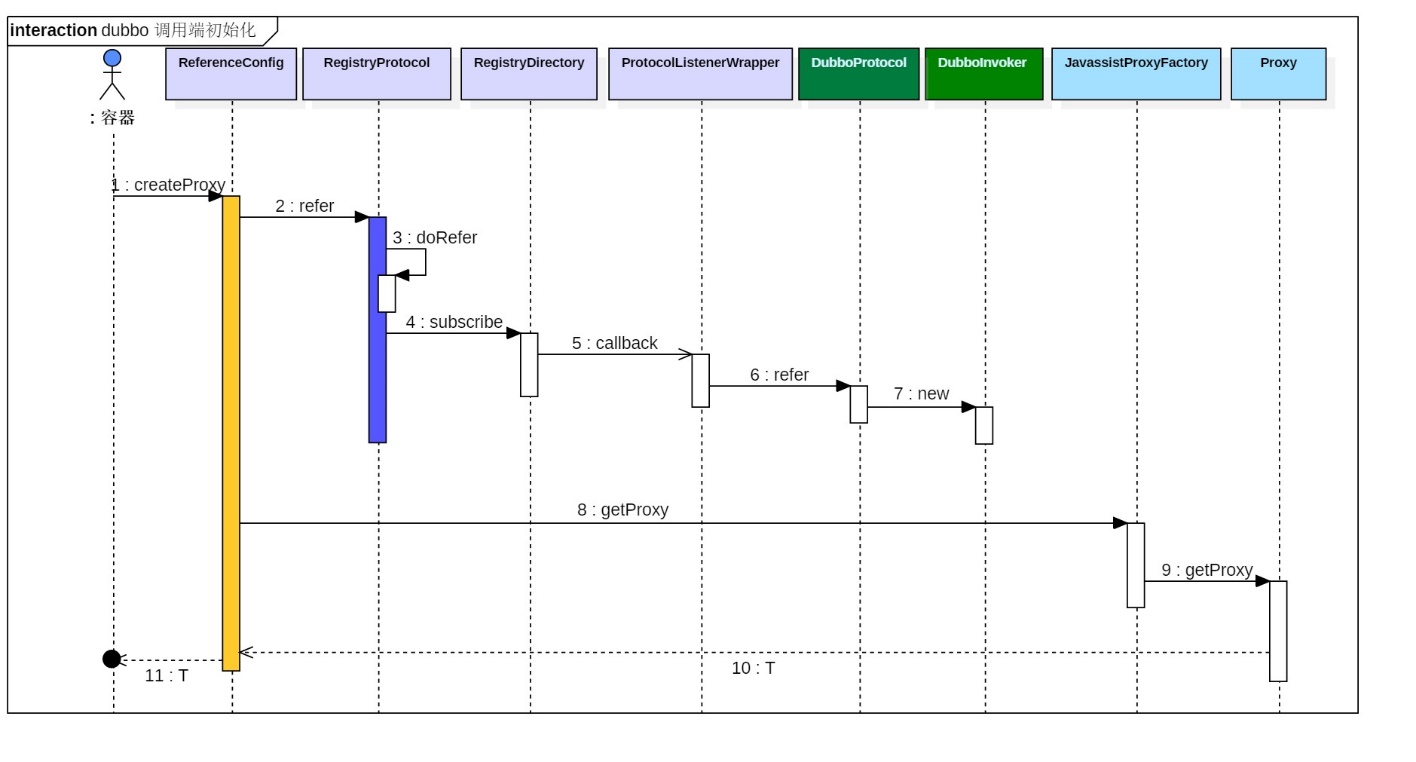
public static Proxy getProxy(Class... ics) {

return getProxy(ClassHelper.getCallerClassLoader(Proxy.class), ics);

}

1. **初始化时序图**

下图中的callback是指监听注册中心后回调产生的调用关系。



**总结**

本文提到了分析框架源码的作用，RPC简单结构，并制定了一个源码分析的计划。最后以调用端在启动时执行的dubbo流程为起点开始了dubbo流程的源码分析。

http://www.cnblogs.com/ASPNET2008/p/7292472.html

# [dubbo源码分析（二）：超时原理以及应用场景](http://www.cnblogs.com/ASPNET2008/p/7292472.html)

本篇主要记录dubbo中关于超时的常见问题，实现原理，解决的问题以及如何在服务降级中体现作用等。

# 超时问题

为了检查对dubbo超时的理解，尝试回答如下几个问题，如果回答不上来或者不确定那么说明此处需要再多研究研究。

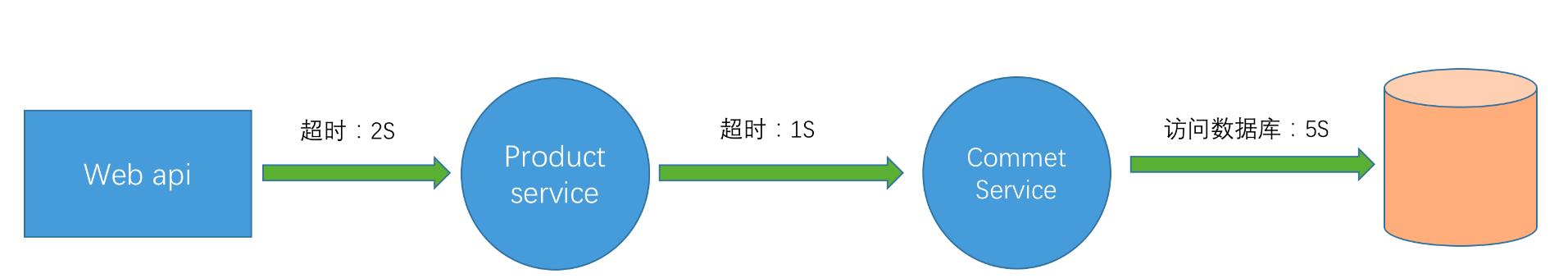
我只是针对个人的理解提问题，并不代表我理解的就是全面深入的，但我的问题如果也回答不了，那至少说明理解的确是不够细的。

* 超时是针对消费端还是服务端？
* 超时在哪设置？
* 超时设置的优先级是什么？
* 超时的实现原理是什么？
* 超时解决的是什么问题？

# 问题解答

## RPC场景

本文所有问题均以下图做为业务场景，一个web api做为前端请求，product service是产品服务，其中调用comment service(评论服务)获取产品相关评论，comment service从持久层中加载数据。



## 超时是针对消费端还是服务端？

* 如果是争对消费端，那么当消费端发起一次请求后，如果在规定时间内未得到服务端的响应则直接返回超时异常，但服务端的代码依然在执行。
* 如果是争取服务端，那么当消费端发起一次请求后，一直等待服务端的响应，服务端在方法执行到指定时间后如果未执行完，此时返回一个超时异常给到消费端。

dubbo的超时是争对客户端的，由于是一种NIO模式，消费端发起请求后得到一个ResponseFuture，然后消费端一直轮询这个ResponseFuture直至超时或者收到服务端的返回结果。虽然超时了，但仅仅是消费端不再等待服务端的反馈并不代表此时服务端也停止了执行。

按上图的业务场景，看看生成的日志：

product service:报超时错误，因为comment service 加载数据需要5S，但接口只等1S 。

Caused by: com.alibaba.dubbo.remoting.TimeoutException: Waiting server-side response timeout. start time: 2017-08-05 18:14:52.751, end time: 2017-08-05 18:14:53.764, client elapsed: 6 ms, server elapsed: 1006 ms, timeout: 1000 ms, request: Request [id=0, version=2.0.0, twoway=true, event=false, broken=false, data=RpcInvocation [methodName=getCommentsByProductId, parameterTypes=[class java.lang.Long], arguments=[1], attachments={traceId=6299543007105572864, spanId=6299543007105572864, input=259, path=com.jim.framework.dubbo.core.service.CommentService, interface=com.jim.framework.dubbo.core.service.CommentService, version=0.0.0}]], channel: /192.168.10.222:53204 -> /192.168.10.222:7777

at com.alibaba.dubbo.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:107) ~[dubbo-2.5.3.jar:2.5.3]

at com.alibaba.dubbo.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:84) ~[dubbo-2.5.3.jar:2.5.3]

at com.alibaba.dubbo.rpc.protocol.dubbo.DubboInvoker.doInvoke(DubboInvoker.java:96) ~[dubbo-2.5.3.jar:2.5.3]

... 42 common frames omitted

comment service : 并没有异常，而是慢慢悠悠的执行自己的逻辑：

2017-08-05 18:14:52.760 INFO 846 --- [2:7777-thread-5] c.j.f.d.p.service.CommentServiceImpl : getComments start:Sat Aug 05 18:14:52 CST 2017

2017-08-05 18:14:57.760 INFO 846 --- [2:7777-thread-5] c.j.f.d.p.service.CommentServiceImpl : getComments end:Sat Aug 05 18:14:57 CST 2017

从日志来看，超时影响的是消费端，与服务端没有直接关系。

## 超时在哪设置？

### 消费端

* 全局控制

<dubbo:consumer timeout="1000"></dubbo:consumer>

* 接口控制
* 方法控制

### 服务端

* 全局控制

<dubbo:provider timeout="1000"></dubbo:provider>

* 接口控制
* 方法控制

可以看到dubbo针对超时做了比较精细化的支持，无论是消费端还是服务端，无论是接口级别还是方法级别都有支持。

## 超时设置的优先级是什么？

上面有提到dubbo支持多种场景下设置超时时间，也说过超时是针对消费端的。那么既然超时是针对消费端，为什么服务端也可以设置超时呢？

这其实是一种策略，其实服务端的超时配置是消费端的缺省配置，即如果服务端设置了超时，任务消费端可以不设置超时时间，简化了配置。

另外针对控制的粒度，dubbo支持了接口级别也支持方法级别，可以根据不同的实际情况精确控制每个方法的超时时间。所以最终的优先顺序为：客户端方法级>客户端接口级>客户端全局>服务端方法级>服务端接口级>服务端全局。

## 超时的实现原理是什么？

之前有简单提到过, dubbo默认采用了netty做为网络组件，它属于一种NIO的模式。消费端发起远程请求后，线程不会阻塞等待服务端的返回，而是马上得到一个ResponseFuture，消费端通过不断的轮询机制判断结果是否有返回。因为是通过轮询，轮询有个需要特别注要的就是避免死循环，所以为了解决这个问题就引入了超时机制，只在一定时间范围内做轮询，如果超时时间就返回超时异常。

源码

### ResponseFuture接口定义

public interface ResponseFuture {

/\*\*

\* get result.

\*

\* @return result.

\*/

Object get() throws RemotingException;

/\*\*

\* get result with the specified timeout.

\*

\* @param timeoutInMillis timeout.

\* @return result.

\*/

Object get(int timeoutInMillis) throws RemotingException;

/\*\*

\* set callback.

\*

\* @param callback

\*/

void setCallback(ResponseCallback callback);

/\*\*

\* check is done.

\*

\* @return done or not.

\*/

boolean isDone();

}

### ReponseFuture的实现类：DefaultFuture

只看它的get方法，可以清楚看到轮询的机制。

public Object get(int timeout) throws RemotingException {

if (timeout <= 0) {

timeout = Constants.DEFAULT\_TIMEOUT;

}

if (! isDone()) {

long start = System.currentTimeMillis();

lock.lock();

try {

while (! isDone()) {

done.await(timeout, TimeUnit.MILLISECONDS);

if (isDone() || System.currentTimeMillis() - start > timeout) {

break;

}

}

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

lock.unlock();

}

if (! isDone()) {

throw new TimeoutException(sent > 0, channel, getTimeoutMessage(false));

}

}

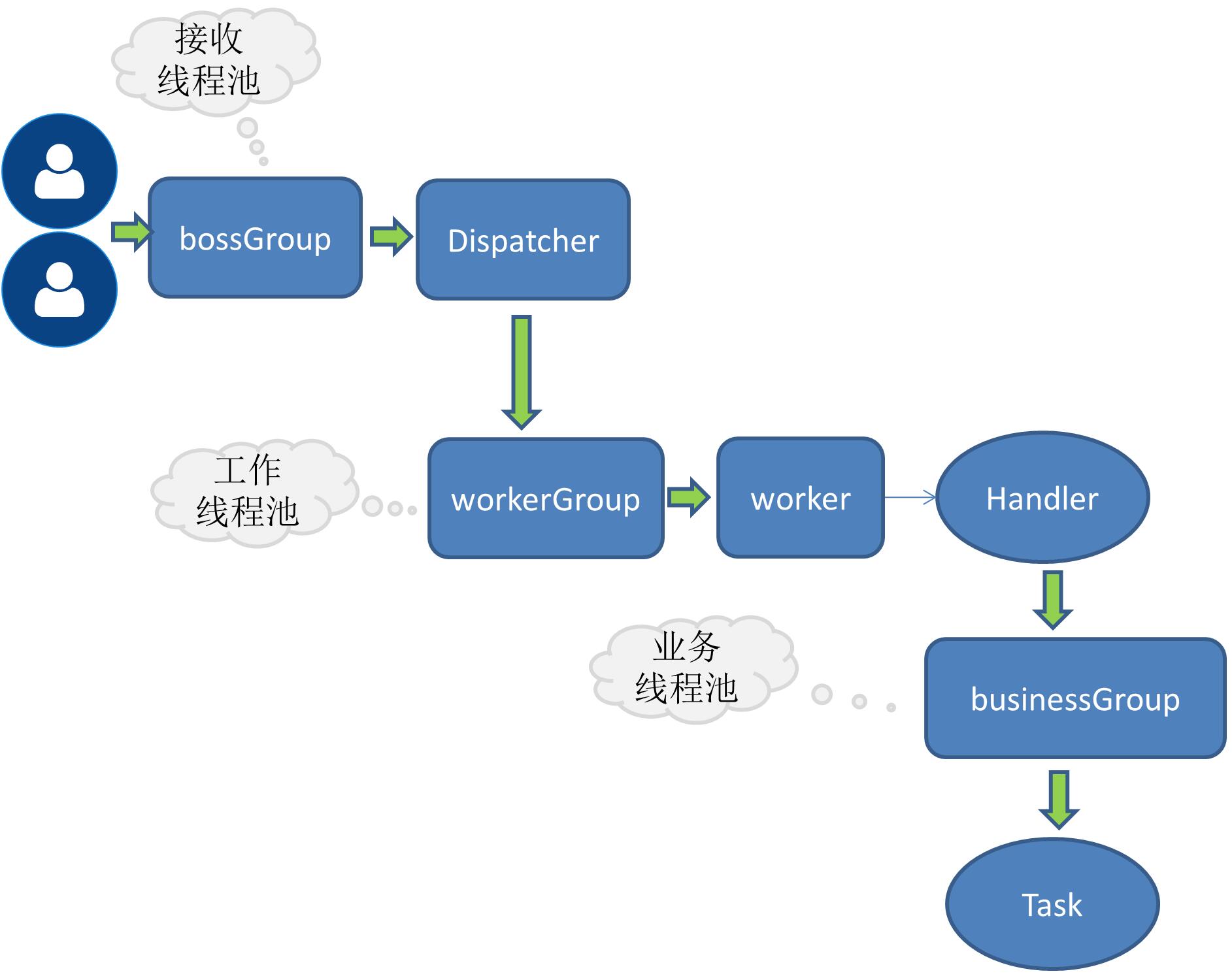
return returnFromResponse();

}

## 超时解决的是什么问题？

设置超时主要是解决什么问题？如果没有超时机制会怎么样？

回答上面的问题，首先要了解dubbo这类rpc产品的线程模型。下图是我之前个人RPC学习产品的示例图，与dubbo的线程模型大致是相同的，有兴趣的可参考我的笔记：[简单RPC框架-业务线程池](http://www.cnblogs.com/ASPNET2008/p/7106820.html)



我们从dubbo的源码看下这下线程模型是怎么用的：

### netty boss

主要是负责socket连接之类的工作。

### netty wokers

将一个请求分给后端的某个handle去处理，比如心跳handle ,执行业务请求的 handle等。

Netty Server中可以看到上述两个线程池是如何初始化的：

首选是open方法，可以看到一个boss一个worker线程池。

protected void doOpen() throws Throwable {

NettyHelper.setNettyLoggerFactory();

ExecutorService boss = Executors.newCachedThreadPool(new NamedThreadFactory("NettyServerBoss", true));

ExecutorService worker = Executors.newCachedThreadPool(new NamedThreadFactory("NettyServerWorker", true));

ChannelFactory channelFactory = new NioServerSocketChannelFactory(boss, worker, getUrl().getPositiveParameter(Constants.IO\_THREADS\_KEY, Constants.DEFAULT\_IO\_THREADS));

bootstrap = new ServerBootstrap(channelFactory);

// ......

}

再看ChannelFactory的构造函数：

public NioServerSocketChannelFactory(Executor bossExecutor, Executor workerExecutor, int workerCount) {

this(bossExecutor, 1, workerExecutor, workerCount);

}

可以看出，boss线程池的大小为1，worker线程池的大小也是可以配置的，默认大小是当前系统的核心数+1，也称为IO线程。

### busines（业务线程池）

为什么会有业务线程池，这里不多解释，可以参考我上面的文章。

缺省是采用固定大小的线程池，dubbo提供了三种不同类型的线程池供用户选择。我们看看这个类：AllChannelHandler，它是其中一种handle，处理所有请求，它的一个作用就是调用业务线程池去执行业务代码，其中有获取线程池的方法：

private ExecutorService getExecutorService() {

ExecutorService cexecutor = executor;

if (cexecutor == null || cexecutor.isShutdown()) {

cexecutor = SHARED\_EXECUTOR;

}

return cexecutor;

}

上面代码中的变量executor来自于AllChannelHandler的父类WrappedChannelHandler，看下它的构造函数：

public WrappedChannelHandler(ChannelHandler handler, URL url) {

//......

executor = (ExecutorService) ExtensionLoader.getExtensionLoader(ThreadPool.class).getAdaptiveExtension().getExecutor(url);

//......

}

获取线程池来自于SPI技术,从代码中可以看出线程池的缺省配置就是上面提到的固定大小线程池。

@SPI("fixed")

public interface ThreadPool {

/\*\*

\* 线程池

\*

\* @param url 线程参数

\* @return 线程池

\*/

@Adaptive({Constants.THREADPOOL\_KEY})

Executor getExecutor(URL url);

}

最后看下是如何将请求丢给线程池去执行的，在AllChannelHandler中有这样的方法：

public void received(Channel channel, Object message) throws RemotingException {

ExecutorService cexecutor = getExecutorService();

try {

cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.RECEIVED, message));

} catch (Throwable t) {

throw new ExecutionException(message, channel, getClass() + " error when process received event .", t);

}

}

典型问题：拒绝服务

如果上面提到的dubbo线程池模型理解了，那么也就容易理解一个问题，当前端大量请求并发出现时，很有可以将业务线程池中的线程消费完，因为默认缺省的线程池是固定大小（我现在版本缺省线程池大小为200），此时会出现服务无法按预期响应的结果，当然由于是固定大小的线程池，当核心线程滿了后也有队列可排,但默认是不排队的，需要排队需要单独配置，我们可以从线程池的具体实现中看：

public class FixedThreadPool implements ThreadPool {

public Executor getExecutor(URL url) {

String name = url.getParameter(Constants.THREAD\_NAME\_KEY, Constants.DEFAULT\_THREAD\_NAME);

int threads = url.getParameter(Constants.THREADS\_KEY, Constants.DEFAULT\_THREADS);

int queues = url.getParameter(Constants.QUEUES\_KEY, Constants.DEFAULT\_QUEUES);

return new ThreadPoolExecutor(threads, threads, 0, TimeUnit.MILLISECONDS,

queues == 0 ? new SynchronousQueue<Runnable>() :

(queues < 0 ? new LinkedBlockingQueue<Runnable>()

: new LinkedBlockingQueue<Runnable>(queues)),

new NamedThreadFactory(name, true), new AbortPolicyWithReport(name, url));

}

}

上面代码的结论是：

* 默认线程池大小为200（不同的dubbo版本可能此值不同）
* 默认线程池不排队，如果需要排队，需要指定队列的大小

当业务线程用完后，服务端会报如下的错误：

Caused by: java.util.concurrent.RejectedExecutionException: Thread pool is EXHAUSTED! Thread Name: DubboServerHandler-192.168.10.222:9999, Pool Size: 1 (active: 1, core: 1, max: 1, largest: 1), Task: 8 (completed: 7), Executor status:(isShutdown:false, isTerminated:false, isTerminating:false), in dubbo://192.168.10.222:9999!

at com.alibaba.dubbo.common.threadpool.support.AbortPolicyWithReport.rejectedExecution(AbortPolicyWithReport.java:53) ~[dubbo-2.5.3.jar:2.5.3]

at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823) [na:1.8.0\_121]

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369) [na:1.8.0\_121]

at com.alibaba.dubbo.remoting.transport.dispatcher.all.AllChannelHandler.caught(AllChannelHandler.java:65) ~[dubbo-2.5.3.jar:2.5.3]

... 17 common frames omitted

通过上面的分析，对调用的服务设置超时时间，是为了避免因为某种原因导致线程被长时间占用，最终出现线程池用完返回拒绝服务的异常。

# 超时与服务降级

按我们文章之前的场景，web api 请求产品明细时调用product service，为了查询产品评论product service调用comment service。如果此时由于comment service异常，响应时间增大到10S（远大于上游服务设置的超时时间），会发生超时异常，进而导致整个获取产品明细的接口异常，这也就是平常说的强依赖。这类强依赖是超时不能解决的，解决方案一般是两种：

* 调用comment service时做异常捕获，返回空值或者返回具体的错误码，消费端根据不同的错误码做不同的处理。
* 调用coment service做服务降级，比如发生异常时返回一个mock的数据,dubbo默认支持mock。

只有通过做异常捕获或者服务降级才能确保某些不重要的依赖出问题时不影响主服务的稳定性。而超时就可以与服务降级结合起来，当消费端发生超时时自动触发服务降级， 这样即使我们的评论服务一直慢，但不影响获取产品明细的主体功能，只不过会牺牲部分体验，用户看到的评论不是真实的，但评论相对是个边缘功能，相比看不到产品信息要轻的多，某种程度上是可以舍弃的。