Dubbo 源码解析

本文为本人阅读 dubbo 源码时所作的知识整理，其中部分资料引用自本人所搜索到的网上资料。

目录

目录

1 源码阅读路径

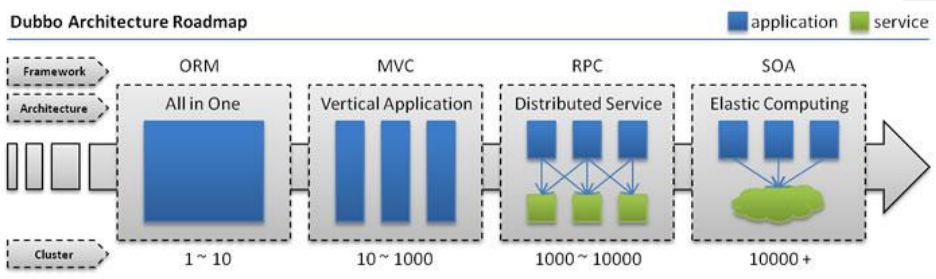
Dubbo 代码量看起来不大，但实际上涉及到的知识点非常多；因此建议对 java 开发处于初学者就 不要看这么复杂的东西了；如果要看，需要按如下知识点去储备知识：

1) Java 语言编程知识（《java 编程思想》）
2) Spring 框架开发及应用
   这里重点推荐夏昕写的两本《SpringGuide》《Hibernate 开发指南》
   《Spring 实战》、牛逼一点的可以看看《Spring 源码解析》）
   如果要实际的开源项目，建议看看 jeecg 的开源代码；
3) Java 网络编程
   《Java_TCPIP_Socket 网络编程.pdf》对 java 的 socket 编程有非常深入的介绍；
   《Java 网络编程》
   另外这一部分一定要弄清楚 NIO，以及比较流行的网络编程框架，比如 mina，netty 等等，自己跑几个例子看看；
4) Java rpc 机制
   从功能来看，dubbo 就是一个 rpc 框架，但是他实际上包含 rpc 相关的所有东西，其底层实现还是这些 java rpc 机制；
   这里可以了解一下 rmi, hessian,thrift,webservice 等等；
5) Java 其他内容
   此外，dubbo 还涉及到其他一些 java 的内容，重点包括：序列化、SPI、代理、Classloader、ScriptEngine 等东西；
6) 设计模式
   这一部分是所有软件架构都会涉及到的内容；

2 背景

这里主要说明为什么需要 dubbo?

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



- **单一应用架构**

  当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的 数据访问框架(ORM) 是关键。

- **垂直应用架构**

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。

此时，用于加速前端页面开发的 Web 框架(MVC) 是关键。

- **分布式服务架构**

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。

此时，用于提高业务复用及整合的 分布式服务框架(RPC) 是关键。

- **流动计算架构**

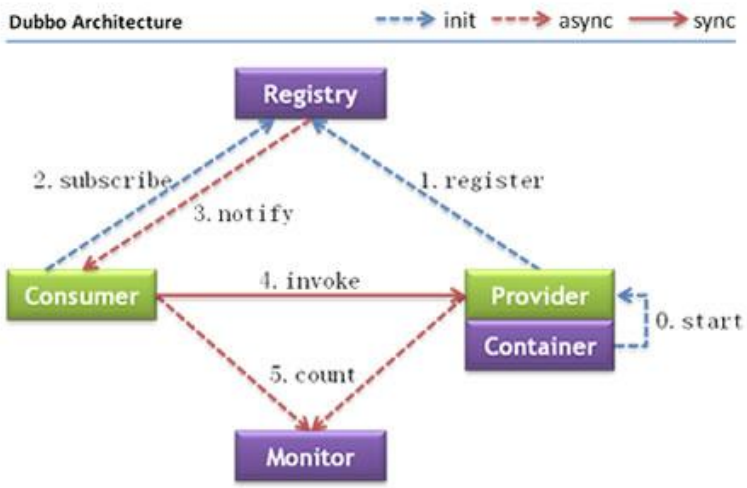
当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。

此时，用于提高机器利用率的 资源调度和治理中心(SOA) 是关键。

实际上这样是一种云架构，实现了服务的动态扩展；在高并发情况下，服务端可以快速部署机器，而对于应用端没有任何其他影响（除了服务并发处理能力提升）；当然，在当前 PC\web\wap\app 多端的情况下，使用 SOA 结构，可以最大程度复用已有代码，避免重复开发。

业界存在一种比较有意思的说法：小米在秒杀的时候，访问量暴涨，此时小米就会去申请很多阿里云服务器（1 小时内免费），快速部署服务，等秒杀完成后，快速归还；在如此高并发下硬件新增成本几乎为 0。

3 Dubbo 架构



节点角色说明：

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。

调用关系说明：

0. 服务容器负责启动，加载，运行服务提供者。

1. 服务提供者在启动时，向注册中心注册自己提供的服务。

2. 服务消费者在启动时，向注册中心订阅自己所需的服务。

3. 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。

4. 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。

5. 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。

(1) 连通性：

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) 健状性：

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；

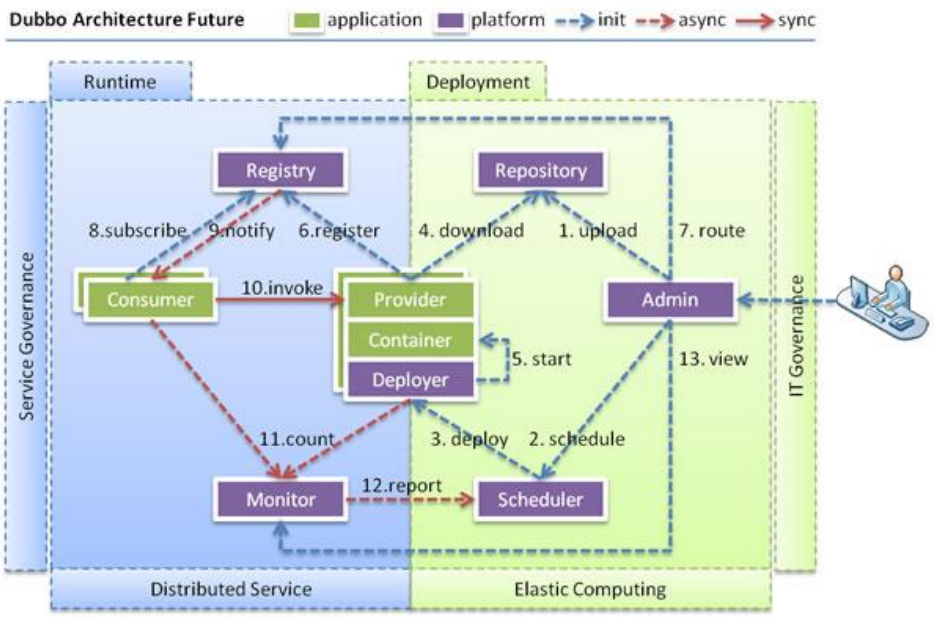
(3) 伸缩性：

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；

(4) 升级性：

当服务集群规模进一步扩大，带动 IT 治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力：



Deployer: 自动部署服务的本地代理。

Repository: 仓库用于存储服务应用发布包。

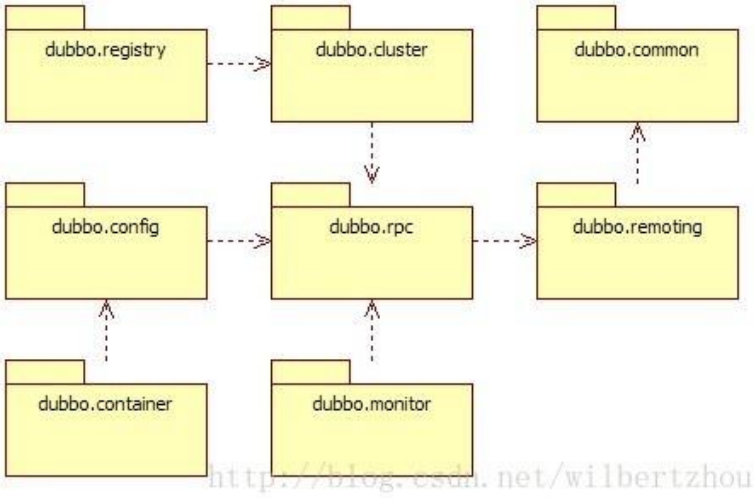
Scheduler: 调度中心基于访问压力自动增减服务提供者。

Admin: 统一管理控制台。

4 HelloWorld 例子

这里就不具体介绍了，下载 dubbo 的源码，然后把 dubbo-demo-provider 和 dubbo-demo-consumer 这两个项目跑一跑；基本上如果是简单使用的话，基本上只需要写业务处理代码，然后按照 dubbo 的要求进行部署即可；

5 源文件概述



Dubbo 源文件主要包含以上这么多包，其中：

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 Standlone 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。

整体上按照分层结构进行分包，与分层的不同点在于：

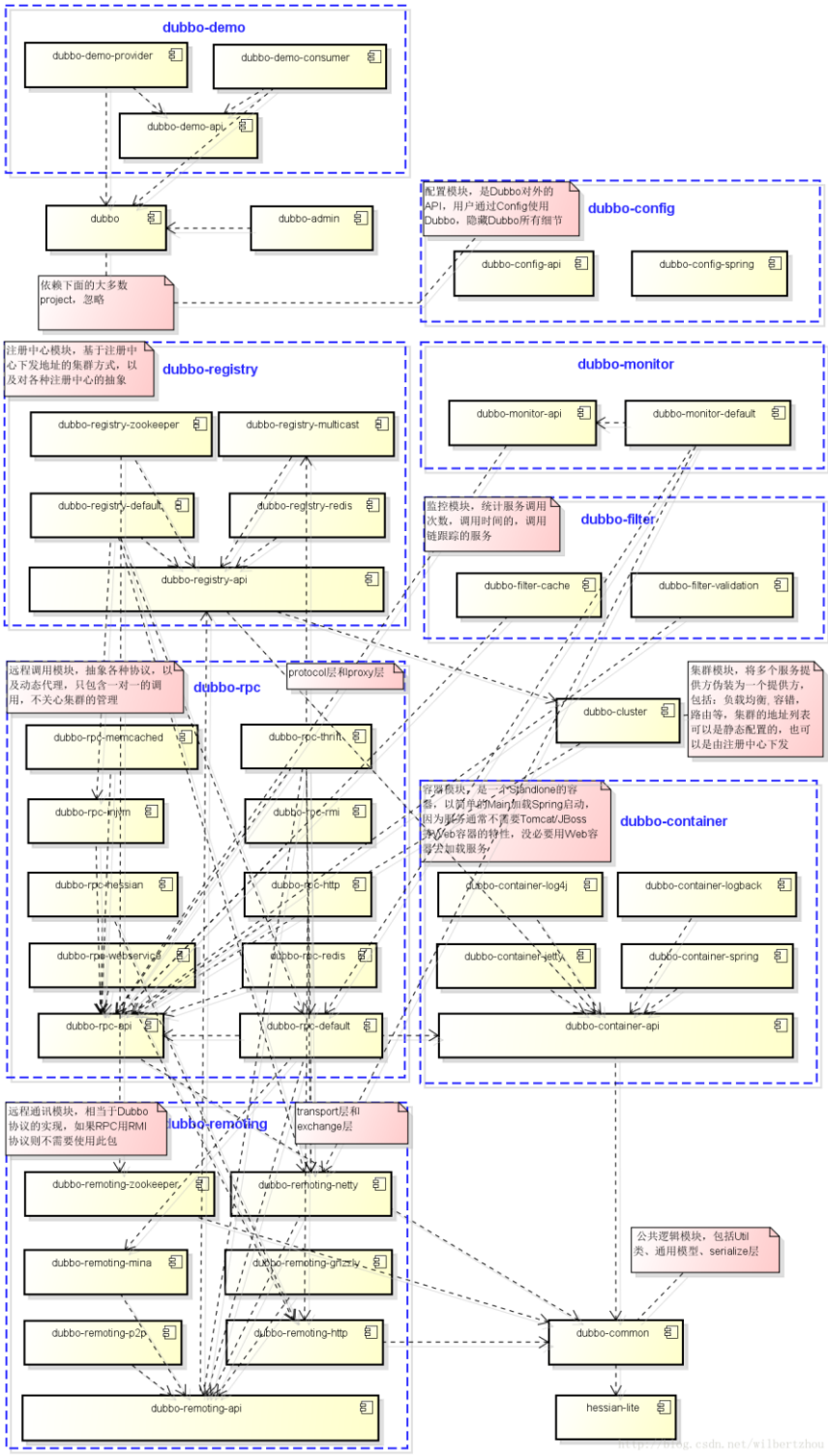
container 为服务容器，用于部署运行服务，没有在层中画出。

protocol 层和 proxy 层都放在 rpc 模块中，这两层是 rpc 的核心，在不需要集群时(只有一个提供者)，可以只使用这两层完成 rpc 调用。

transport 层和 exchange 层都放在 remoting 模块中，为 rpc 调用的通讯基础。

serialize 层放在 common 模块中，以便更大程度复用。

下面是更详细的 Project 关系图，依赖关系线有点乱。整个模块是从上到下传递依赖的。


dubbo-demo
dubbo-demo-provider
dubbo-demo-consumer
dubbo-demo-api
dubbo
dubbo-admin
依赖下面的大多数project，忽略
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-config
dubbo-config-api
dubbo-config-spring
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-registry
dubbo-registry-zookeeper
dubbo-registry-multicast
dubbo-registry-default
dubbo-registry-redis
dubbo-registry-api
dubbo-monitor
dubbo-monitor-api
dubbo-monitor-default
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-filter
dubbo-filter-cache
dubbo-filter-validation
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-rpc
protocol层和proxy层
dubbo-rpc-memcached
dubbo-rpc-thrift
dubbo-rpc-injvm
dubbo-rpc-rmi
dubbo-rpc-hessian
dubbo-rpc-http
dubbo-rpc-webservice
dubbo-rpc-redis
dubbo-rpc-api
dubbo-rpc-default
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
容器模块，是一个Standlone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
dubbo-container
dubbo-container-log4j
dubbo-container-logback
dubbo-container-jetty
dubbo-container-spring
dubbo-container-api
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-remoting
transport层和exchange层
dubbo-remoting-zookeeper
dubbo-remoting-netty
dubbo-remoting-mina
dubbo-remoting-grizzly
dubbo-remoting-p2p
dubbo-remoting-http
dubbo-remoting-api
公共逻辑模块，包括Util类、通用模型、serialize层
dubbo-common
hessian-lite
http://blog.csdn.net/wilbertzhou

6 核心机制分析

这里对 dubbo 中涉及到的核心机制进行分析，包括设计模式、bean 加载、扩展点机制、动态代理及远程调用流程。

6.1 设计模式

1、工厂模式

ServiceConfig 中有个字段，代码是这样的：

查看文本打印

```
private static final Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
```

Dubbo 里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了 jdkspi 的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在 classpath 下增加个文件就可以了，代码零侵入。另外，像上面的 Adaptive 实现，可以做到调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的实现类。

2、装饰器模式

Dubbo 在启动和调用阶段都大量使用了装饰器模式。以 Provider 提供的调用链为例，具体的调用链代码是在 ProtocolFilterWrapper 的 buildInvokerChain 完成的，具体是将注解中含有 group=provider 的 Filter 实现，按照 order 排序，最后的调用顺序是

查看文本打印

```
EchoFilter-》ClassLoaderFilter-》GenericFilter-》ContextFilter-》ExceptionFilter-》TimeoutFilter-》MonitorFilter-》TraceFilter。
```

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter 的作用是判断是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像 ClassLoaderFilter 则只是在主功能上添加了功能，更改当前线程的 ClassLoader，这是典型的装饰器模式。

3、观察者模式

Dubbo 的 provider 启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服务，订阅时，采用了观察者模式，开启一个 listener。注册中心会每 5 秒定时检查是否有服务更新，如果有更新，向该服务的提供者发送一个 notify 消息，provider 接受到 notify 消息后，即运行 NotifyListener 的 notify 方法，执行监听器方法。

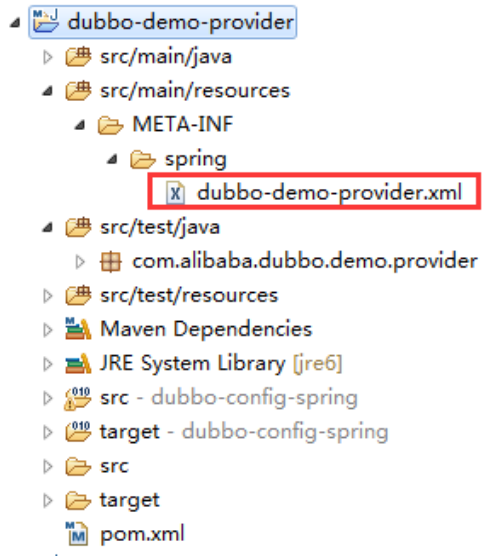
4、动态代理模式

Dubbo 扩展 jdkspi 的类 ExtensionLoader 的 Adaptive 实现是典型的动态代理实现。Dubbo 需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪个实现类，所以采用先生成代理类的方法，能够做到灵活的调用。生成代理类的代码是 ExtensionLoader 的 createAdaptiveExtensionClassCode 方法。代理类的主要逻辑是，获取 URL 参数中指定参数的值作为获取实现类的 key。

6.2 Bean 加载

6.2.1 Spring 可扩展 Schema

在 dubbo-demo-provider 项目中，我们在如下文件中配置服务提供方：



文件内容为：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://code.alibabatech.com/schema/dubbo
http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <bean id="demoService"
class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />

    <dubbo:service protocol="rmi"
interface="com.alibaba.dubbo.demo.DemoService" ref="demoService" />

</beans>
```

本节将分析这一部分的实现过程，即如何通过配置文件生成实现类对象；

基于 Spring 可扩展 Schema 提供自定义配置支持(spring 配置文件中 配置标签支持)
http://www.cnblogs.com/jifeng/archive/2011/09/14/2176599.html
扩展 spring schema 的方法，代码下载 具体例子，有源码
http://www.yihaomen.com/article/java/438.htm

在很多情况下，我们需要为系统提供可配置化支持，简单的做法可以直接基于 Spring 的标准 Bean 来配置，但配置较为复杂或者需要更多丰富控制的时候，会显得非常笨拙。一般的做法会用原生态的方式去解析定义好的 xml 文件，然后转化为配置对象，这种方式当然可以解决所有问题，但实现起来比较繁琐，特别是是在配置非常复杂的时候，解析工作是一个不得不考虑的负担。Spring 提供了可扩展 Schema 的支持，这是一个不错的折中方案，完成一个自定义配置一般需要以下步骤：

- 设计配置属性和 JavaBean
- 编写 XSD 文件
- 编写 NamespaceHandler 和 BeanDefinitionParser 完成解析工作
- 编写 spring.handlers 和 spring.schemas 串联起所有部件
- 在 Bean 文件中应用

下面结合一个小例子来实战以上过程

1）设计配置属性和 JavaBean

首先当然得设计好配置项，并通过 JavaBean 来建模，本例中需要配置 People 实体，配置属性 name 和 age（id 是默认需要的）

```
public class People {

    private String id;

    private String name;

    private Integer age;

}
```

2）编写 XSD 文件

为上一步设计好的配置项编写 XSD 文件，XSD 是 schema 的定义文件，配置的输入和解析输出都是以 XSD 为契约，本例中 XSD 如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema

    xmlns="http://blog.csdn.net/cutesource/schema/people"

    xmlns:xsd="http://www.w3.org/2001/XMLSchema"

    xmlns:beans="http://www.springframework.org/schema/beans"

    targetNamespace="http://blog.csdn.net/cutesource/schema/people"

    elementFormDefault="qualified"

    attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans" />

    <xsd:element name="people">

        <xsd:complexType>

            <xsd:complexContent>

                <xsd:extension base="beans:identifiedType">

                    <xsd:attribute name="name" type="xsd:string" />

                    <xsd:attribute name="age" type="xsd:int" />

                </xsd:extension>

            </xsd:complexContent>

        </xsd:complexType>

    </xsd:element>

</xsd:schema>
```

关于 xsd:schema 的各个属性具体含义就不作过多解释，可以参见 http://www.w3school.com.cn/schema/schema_schema.asp；

<xsd:element name="people">对应着配置项节点的名称，因此在应用中会用 people 作为节点名来引用这个配置

<xsd:attribute name="name" type="xsd:string" /> 和 <xsd:attribute name="age" type="xsd:int" />对应着配置项 people 的两个属性名，因此在应用中可以配置 name 和 age 两个属性，分别是 string 和 int 类型

完成后需把 xsd 存放在 classpath 下，一般都放在 META-INF 目录下（本例就放在这个目录下）

3）编写 NamespaceHandler 和 BeanDefinitionParser 完成解析工作

下面需要完成解析工作，会用到 NamespaceHandler 和 BeanDefinitionParser 这两个概念。具体说来 NamespaceHandler 会根据 schema 和节点名找到某个 BeanDefinitionParser，然后由 BeanDefinitionParser 完成具体的解析工作。因此需要分别完成 NamespaceHandler 和 BeanDefinitionParser 的实现类，Spring 提供了默认实现类 NamespaceHandlerSupport 和 AbstractSingleBeanDefinitionParser，简单的方式就是去继承这两个类。本例就是采取这种方式：

```
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;
public class MyNamespaceHandler extends NamespaceHandlerSupport {
    public void init() {
        registerBeanDefinitionParser("people", new PeopleBeanDefinitionParser());
    }
}
```

其中 registerBeanDefinitionParser("people", new PeopleBeanDefinitionParser());就是用来把节点名和解析类联系起来，在配置中引用 people 配置项时，就会用 PeopleBeanDefinitionParser 来解析配置。PeopleBeanDefinitionParser 就是本例中的解析类：

```
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;
public class PeopleBeanDefinitionParser extends AbstractSingleBeanDefinitionParser {
    protected Class getBeanClass(Element element) {
        return People.class;
    }
    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        String name = element.getAttribute("name");
        String age = element.getAttribute("age");
        String id = element.getAttribute("id");
        if (StringUtils.hasText(id)) {
            bean.addPropertyValue("id", id);
        }
        if (StringUtils.hasText(name)) {
            bean.addPropertyValue("name", name);
        }
        if (StringUtils.hasText(age)) {
            bean.addPropertyValue("age", Integer.valueOf(age));
```

```
        }
    }
}
```

其中 element.getAttribute 就是用配置中取得属性值，bean.addPropertyValue 就是把属性值放到 bean 中。

4）编写 spring.handlers 和 spring.schemas 串联起所有部件

上面几个步骤走下来会发现开发好的 handler 与 xsd 还没法让应用感知到，就这样放上去是没法把前面做的工作纳入体系中的，spring 提供了 spring.handlers 和 spring.schemas 这两个配置文件来完成这项工作，这两个文件需要我们自己编写并放入 META-INF 文件夹中，这两个文件的地址必须是 META-INF/spring.handlers 和 META-INF/spring.schemas，spring 会默认去载入它们，本例中 spring.handlers 如下所示：

```
http\://blog.csdn.net/cutesource/schema/people=study.schemaExt.MyNamespaceHandler
```

以上表示当使用到名为"http://blog.csdn.net/cutesource/schema/people"的 schema 引用时，会通过 study.schemaExt.MyNamespaceHandler 来完成解析

spring.schemas 如下所示：

```
http\://blog.csdn.net/cutesource/schema/people.xsd=META-INF/people.xsd
```

以上就是载入 xsd 文件

5）在 Bean 文件中应用

到此为止一个简单的自定义配置以完成，可以在具体应用中使用了。使用方法很简单，和配置一个普通的 spring bean 类似，只不过需要基于我们自定义 schema，本例中引用方式如下所示：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cutesource="http://blog.csdn.net/cutesource/schema/people"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.x
sd
http://blog.csdn.net/cutesource/schema/people http://blog.csdn.net/cutesource/schema/people.xsd">
    <cutesource:people id="cutesource" name="袁志俊" age="27"/>
</beans>
```

其中 xmlns:cutesource="http://blog.csdn.net/cutesource/schema/people"是用来指定自定义 schema，xsi:schemaLocation 用来指定 xsd 文件。<cutesource:people id="cutesource" name="zhijun.yuanzj" age="27"/>是一个具体的自定义配置使用实例。

最后就可以在具体程序中使用基本的 bean 载入方式来载入我们的自定义配置对象了，如：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("application.xml");
People p = (People)ctx.getBean("cutesource");
System.out.println(p.getId());
System.out.println(p.getName());
System.out.println(p.getAge());
```

会输出：

cutesource

以上就是一个基于 Spring 可扩展 Schema 提供自定义配置支持实战过程，一些复杂应用和技巧还有待挖掘

6.2.2 Spring 加载 bean 流程

6.2.2.1 解析 xml 中的 bean 定义

dubbo 加载 Bean 和远程调用分析（1）
http://blog.csdn.net/lanmo555/article/details/44017035
DUBBO 配置规则详解
http://my.oschina.net/bieber/blog/378638
Dubbo 介绍 2- 源码分析，通过 schema 启动服务 对解析的深入分析
http://chenjingbo.iteye.com/blog/2008325

根据上一小节对于 spring 扩展 schema 的介绍，大概可以猜到 dubbo 中相关的内容是如何实现的。

我们可以看到 dubbo-container-spring 启动时，是启动了 spring 上下文。此时它会去解析 spring 的 bean 配置文件，具体的解析工作是由 dubbo-config-spring 完成的。我们可以看到 dubbo-demo-provider 项目中引用到了对应的类。

这里有个东西没有找到资料，就是 spring 是如何读到 dubbo-config-spring 下的相应配置文件呢？我个人猜测是 spring 会扫描各个 jar 下的 META-INF 文件夹，看是否有 spring.handlers、spring.schemas 这两个文件。

在《Dubbo 介绍 2- 源码分析，通过 schema 启动服务》中的例子：

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.          xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
5.          xsi:schemaLocation="http://www.springframework.org/schema/beans
6.           http://www.springframework.org/schema/beans/spring-beans.xsd
7.           http://code.alibabatech.com/schema/dubbo
8.           http://code.alibabatech.com/schema/dubbo/dubbo.xsd
9.           ">
10.     <dubbo:application name="hello-world-app" />
11.     <dubbo:registry  protocol="zookeeper" address="10.125.195.174:2181" />
12.     <dubbo:protocol name="dubbo" port="20880" />
13.     <dubbo:service interface="demo.service.DemoService"
14.                    ref="demoService" />        <!-- 和本地 bean 一样实现服务 -->
15.     <bean id="demoService" class="demo.service.DemoServiceImpl" />
16. </beans>
```

可以看到，对应的 handler 是 DubboNamespaceHandler。对应的源码如下

```
public class DubboNamespaceHandler extends NamespaceHandlerSupport {

    static {
        Version.checkDuplicate(DubboNamespaceHandler.class);
    }

    public void init() {
        registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, tru
e));
        registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));
        registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));
        registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));
        registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));
        registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));
        registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));
        registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));
        registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));
        registerBeanDefinitionParser("annotation", new DubboBeanDefinitionParser(AnnotationBean.class, true));
    }
}
```

从这里也可以看到，对应的支持的标签其实不多。所有的 Parser 都封装到了 DubboBeanDefinitionParser 中。对应的 class，就是传入的 beanClass。比如 application 的就是 ApplicationConfig。module 的就是 ModuleConfig。经过 Parser 的转换，provider.xml 大概可以变成如下的样子(具体的解析不多解释了)

```
 <bean id="hello-world-app" class="com.alibaba.dubbo.config.ApplicationConfig"/>
<bean id="registryConfig" class="com.alibaba.dubbo.config.RegistryConfig">
    <property name="address" value="10.125.195.174:2181"/>
    <property name="protocol" value="zookeeper"/>
</bean>
<bean id="dubbo" class="com.alibaba.dubbo.config.ProtocolConfig">
    <property name="port" value="20880"/>
</bean>
<bean id="demo.service.DemoService" class="com.alibaba.dubbo.config.spring.ServiceBean">
    <property name="interface" value="demo.service.DemoService"/>
    <property name="ref" ref="demoService"/>
</bean>
<bean id="demoService" class="demo.service.DemoServiceImpl" />
```

实际上在生成 bean 的过程主要是把这些属性都梳理清楚，生成对应的类；

6.2.2.2 onApplicationEvent

从代码分析 ServiceBean 中 export 过程（暴露服务）是在如下过程触发的：

```
    public void onApplicationEvent(ApplicationEvent event) {
        if
(ContextRefreshedEvent.class.getName().equals(event.getClass().getNam
e())) {
        if (isDelay() && ! isExported() && ! isUnexported()) {
                if (logger.isInfoEnabled()) {
                    logger.info("The service ready on spring started.
service: " + getInterface());
                }
                export();
            }
        }
    }
```

该函数的调用过程从 debug 的结果上看，onApplicationEvent 函数应该是系统调用的；org.springframework.context.support.AbstractApplicationContext 类中如下函数：

```
    public void publishEvent(ApplicationEvent event) {
        Assert.notNull(event, "Event must not be null");
        if (logger.isTraceEnabled()) {
            logger.trace("Publishing event in context [" + getId() + "]:
" + event);
        }
        getApplicationEventMulticaster().multicastEvent(event);
        if (this.parent != null) {
            this.parent.publishEvent(event);
        }
    }
```

实际上以上函数是调用如下函数的 finishRefresh 中调用的；

```
    public void refresh() throws BeansException, IllegalStateException
{
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            prepareRefresh();
            // Tell the subclass to refresh the internal bean factory.
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
            // Prepare the bean factory for use in this context.
            prepareBeanFactory(beanFactory);
            try {
    // Allows post-processing of the bean factory in context subclasses.
                postProcessBeanFactory(beanFactory);
```

```
    // Invoke factory processors registered as beans in the context.
                invokeBeanFactoryPostProcessors(beanFactory);
                // Register bean processors that intercept bean creation.
                registerBeanPostProcessors(beanFactory);
                // Initialize message source for this context.
                initMessageSource();
                // Initialize event multicaster for this context.
                initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
                onRefresh();
                // Check for listener beans and register them.
                registerListeners();
                // Instantiate all remaining (non-lazy-init) singletons.
                finishBeanFactoryInitialization(beanFactory);
                // Last step: publish corresponding event.
                finishRefresh();
            }
            catch (BeansException ex) {
    // Destroy already created singletons to avoid dangling resources.
                beanFactory.destroySingletons();
                // Reset 'active' flag.
                cancelRefresh(ex);
                // Propagate exception to caller.
                throw ex;
            }
        }
    }
```

而实际上是 org.springframework.context.support.ClassPathXmlApplicationContext

```
public ClassPathXmlApplicationContext(String[] configLocations) throws BeansException {
        this(configLocations, true, null);
    }

public ClassPathXmlApplicationContext(String[] configLocations,
boolean refresh, ApplicationContext parent)
            throws BeansException {
        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
```

而以上这些在项目中实在如下地方调用：SpringContainer

```
public void start() {
       String configPath = ConfigUtils.getProperty(SPRING_CONFIG);
```

```
        if (configPath == null || configPath.length() == 0) {
            configPath = DEFAULT_SPRING_CONFIG;
        }
        context = new
ClassPathXmlApplicationContext(configPath.split("[,\\s]+"));
        context.start();
    }
```

6.2.2.3 Main

从以上的分析中，我们看到 Container 负责加载 xml 配置，生成对应的 bean；

同时我们注意到 Main 类的 Main 函数中：

```
if ("true".equals(System.getProperty(SHUTDOWN_HOOK_KEY))) {
                Runtime.getRuntime().addShutdownHook(new Thread() {
                    public void run() {
                        for (Container container : containers) {
                            try {
                                container.stop();
                                logger.info("Dubbo " +
container.getClass().getSimpleName() + " stopped!");
                            } catch (Throwable t) {
                                logger.error(t.getMessage(), t);
                            }
                            synchronized (Main.class) {
                                running = false;
                                Main.class.notify();
                            }
                        }
                    }
                });
                }
    ......
synchronized (Main.class) {
            while (running) {
                try {
                    Main.class.wait();
                } catch (Throwable e) {
                }
            }
            }
```

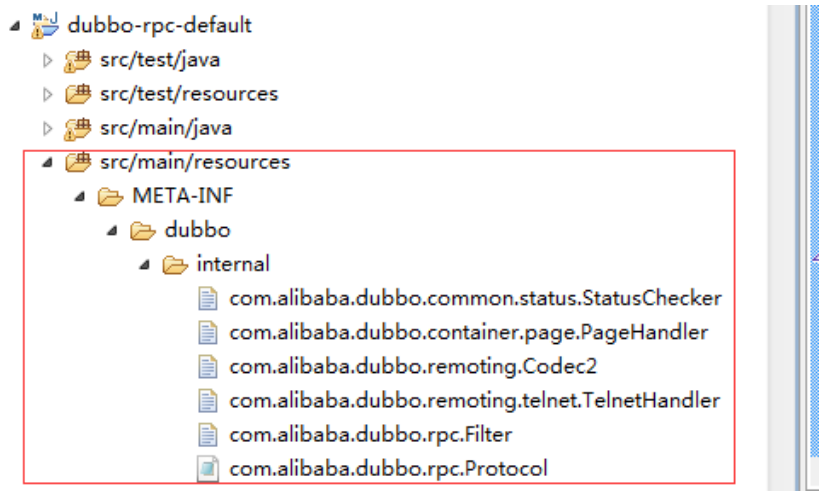
可见程序已启动就一直跑；

6.3 Extension 机制

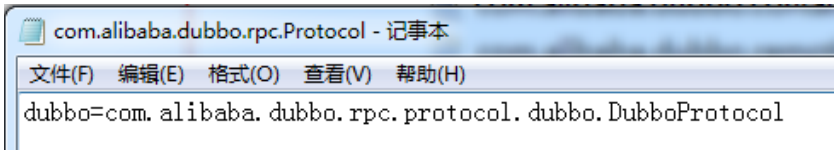
在看 dubbo 源代码时如果没有了解扩展点机制，那么看到代码就是一片凌乱。

6.3.1 Java SPI

首先要搞清楚 java spi 机制，这个可以自己网上搜索，或者看如下文章：
Dubbo 原理解析-Dubbo 内核实现之 SPI 简单介绍
http://blog.csdn.net/quhongwei_zhanqiu/article/details/41577159
搞清楚这一点之后，你就会看到



而查看文件 com.alibaba.dubbo.rpc.Protocol，内容为



这个是不是跟 SPI 机制非常像，只是 java spi 机制是 java 后台帮你实现读取文件并对接具体的实现类，而 dubbo 是自己去读文件；

6.3.2 扩展点

6.3.2.1 扩展点配置

扩展点机制有几个要点：

1） **根据关键字去读取配置文件，获得具体的实现类**；
比如在 dubbo-demo-provider.xml 文件中配置：
<dubbo:service protocol="rmi" interface="com.alibaba.dubbo.demo.DemoService"

ref="demoService" />

则会根据 rmi 去读取具体的协议实现类 RmiProtocol.java；

2） 注解@SPI 和@Adaptive

- **@SPI 注解**：可以认为是定义默认实现类；

  比如 Protocol 接口中，定义默认协议时 dubbo；

  @SPI("dubbo")

  public interface Protocol {}

- **@Adaptive 注解**：该注解打在接口方法上；调 ExtensionLoader.getAdaptiveExtension()获取设配类，会先通过前面的过程生成 java 的源代码，在通过编译器编译成 class 加载。但是 Compiler 的实现策略选择也是通过 ExtensionLoader.getAdaptiveExtension()，如果也通过编译器编译成 class 文件那岂不是要死循环下去了吗？

  此时分析 ExtensionLoader.getAdaptiveExtension()函数，对于有实现类上去打了注解@Adaptive 的 dubbo spi 扩展机制，它获取设配类不在通过前面过程生成设配类 java 源代码，而是在读取扩展文件的时候遇到实现类打了注解@Adaptive 就把这个类作为设配类缓存在 ExtensionLoader 中，调用是直接返回。

3） filter 和 listener

在生成具体的实现类对象时，不是直接读取类文件，而是在读取类文件的基础上，通过 filter 和 listener 去封装类对象；

6.3.2.2 扩展点加载流程

以如下例子为例：

private static final Protocol refprotocol =

ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

在这个语句中，我来分析，首先 Protocol 类带有 SPI 注解，因此我们可以确认默认是使用 dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol 作为默认扩展点；

```
@SPI("dubbo")
public interface Protocol {
    /**
     * 获取缺省端口，当用户没有配置端口时使用。
     *
     * @return 缺省端口
     */
    int getDefaultPort();
    /**
     * 暴露远程服务：<br>
     * 1. 协议在接收请求时，应记录请求来源方地址信息：
RpcContext.getContext().setRemoteAddress();<br>
     * 2. export()必须是幂等的，也就是暴露同一个URL的Invoker两次，和暴露一次没有区别。<br>
     * 3. export()传入的Invoker由框架实现并传入，协议不需要关心。<br>
     *
     * @param <T> 服务的类型
```

```
     * @param invoker 服务的执行体
     * @return exporter 暴露服务的引用，用于取消暴露
     * @throws RpcException 当暴露服务出错时抛出，比如端口已占用
     */
    @Adaptive
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

    /**
     * 引用远程服务：<br>
     * 1．当用户调用refer()所返回的Invoker对象的invoke()方法时，协议需相应执行同URL远端export()传入的Invoker对象的invoke()方法。<br>
     * 2．refer()返回的Invoker由协议实现，协议通常需要在此Invoker中发送远程请求。<br>
     * 3．当url中有设置check=false时，连接失败不能抛出异常，并内部自动恢复。<br>
     *
     * @param <T> 服务的类型
     * @param type 服务的类型
     * @param url 远程服务的URL地址
     * @return invoker 服务的本地代理
     * @throws RpcException 当连接服务提供方失败时抛出
     */
    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;
    /**
     * 释放协议：<br>
     * 1．取消该协议所有已经暴露和引用的服务。<br>
     * 2．释放协议所占用的所有资源，比如连接和端口。<br>
     * 3．协议在释放后，依然能暴露和引用新的服务。<br>
     */
    void destroy();
}
```

这里对应的函数为：

```
    @SuppressWarnings("unchecked")
    public T getAdaptiveExtension() {
        Object instance = cachedAdaptiveInstance.get();
        if (instance == null) {
            if(createAdaptiveInstanceError == null) {
                synchronized (cachedAdaptiveInstance) {
                    instance = cachedAdaptiveInstance.get();
                    if (instance == null) {
                        try {
                            instance = createAdaptiveExtension();
                            cachedAdaptiveInstance.set(instance);
                        } catch (Throwable t) {
```

```
                        createAdaptiveInstanceError = t;
                        throw new IllegalStateException("fail to create
adaptive instance: " + t.toString(), t);
                    }
                }
            }
        }
        else {
            throw new IllegalStateException("fail to create adaptive
instance: " + createAdaptiveInstanceError.toString(),
createAdaptiveInstanceError);
        }
    }
    return (T) instance;
}
```

这里看到 createAdaptiveExtension 函数：

```
@SuppressWarnings("unchecked")
private T createAdaptiveExtension() {
    try {
        return injectExtension((T)
getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can not create adaptive
extenstion " + type + ", cause: " + e.getMessage(), e);
    }
}

private Class<?> getAdaptiveExtensionClass() {
    getExtensionClasses();
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }
    return cachedAdaptiveClass = createAdaptiveExtensionClass();
}
```

6.3.2.2.1 动态生成类

而 cachedAdaptiveInstance 类则是若有 cachedAdaptiveClass 对象，则直接返回，否则通过生成类文件，然后 complier 出来的，其文件内容如下：从该文件可以很容易看出只有标注了@Adaptive 注释的函数会在运行时动态的决定扩展点实现；

```
package com.alibaba.dubbo.rpc;
import com.alibaba.dubbo.common.extension.ExtensionLoader;
public class Protocol$Adpative implements
```

```
com.alibaba.dubbo.rpc.Protocol {
    public void destroy() {
        throw new UnsupportedOperationException("method public abstract
void com.alibaba.dubbo.rpc.Protocol.destroy() of interface
com.alibaba.dubbo.rpc.Protocol is not adaptive method!");
    }

    public int getDefaultPort()
    {
        throw new UnsupportedOperationException("method public abstract
int com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of interface
com.alibaba.dubbo.rpc.Protocol is not adaptive method!");
    }

    public com.alibaba.dubbo.rpc.Exporter
export(com.alibaba.dubbo.rpc.Invoker arg0) throws
com.alibaba.dubbo.rpc.Invoker
    {
        if (arg0 == null) throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument ==
null");
        if (arg0.getUrl() == null) throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
getUrl() == null");com.alibaba.dubbo.common.URL url = arg0.getUrl();
        String extName = ( url.getProtocol() == null ? "dubbo" :
url.getProtocol() );
        if(extName == null) throw new IllegalStateException("Fail to get
extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
url.toString() + ") use keys([protocol])");
        com.alibaba.dubbo.rpc.Protocol extension =
(com.alibaba.dubbo.rpc.Protocol)ExtensionLoader.getExtensionLoader(co
m.alibaba.dubbo.rpc.Protocol.class).getExtension(extName);
        return extension.export(arg0);
    }

    public com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0,
com.alibaba.dubbo.common.URL arg1) throws java.lang.Class
    {
        if (arg1 == null) throw new IllegalArgumentException("url ==
null");
        com.alibaba.dubbo.common.URL url = arg1;
        String extName = ( url.getProtocol() == null ? "dubbo" :
url.getProtocol() );
        if(extName == null) throw new IllegalStateException("Fail to get
```

```
extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
url.toString() + ") use keys([protocol])");
        com.alibaba.dubbo.rpc.Protocol extension =
(com.alibaba.dubbo.rpc.Protocol)ExtensionLoader.getExtensionLoader(co
m.alibaba.dubbo.rpc.Protocol.class).getExtension(extName);
        return extension.refer(arg0, arg1);
    }
}
```

此时我们分析 getExtension 函数：

```
    /**
     * 返回指定名字的扩展。如果指定名字的扩展不存在，则抛异常 {@link
IllegalStateException}.
     *
     * @param name
     * @return
     */
    @SuppressWarnings("unchecked")
    public T getExtension(String name) {
        if (name == null || name.length() == 0)
            throw new IllegalArgumentException("Extension name == null");
        if ("true".equals(name)) {
            return getDefaultExtension();
        }
        Holder<Object> holder = cachedInstances.get(name);
        if (holder == null) {
            cachedInstances.putIfAbsent(name, new Holder<Object>());
            holder = cachedInstances.get(name);
        }
        Object instance = holder.get();
        if (instance == null) {
            synchronized (holder) {
                instance = holder.get();
                if (instance == null) {
                    instance = createExtension(name);
                    holder.set(instance);
                }
            }
        }
        return (T) instance;
    }
```

此时我们分析 createExtension

```
    @SuppressWarnings("unchecked")
    private T createExtension(String name) {
```

```
        Class<?> clazz = getExtensionClasses().get(name);
        if (clazz == null) {
            throw findException(name);
        }
        try {
            T instance = (T) EXTENSION_INSTANCES.get(clazz);
            if (instance == null) {
                EXTENSION_INSTANCES.putIfAbsent(clazz, (T)
clazz.newInstance());
                instance = (T) EXTENSION_INSTANCES.get(clazz);
            }
            injectExtension(instance);
            Set<Class<?>> wrapperClasses = cachedWrapperClasses;
            if (wrapperClasses != null && wrapperClasses.size() > 0) {
                for (Class<?> wrapperClass : wrapperClasses) {
                    instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
                }
            }
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance(name: " +
name + ", class: " +
                    type + ")  could not be instantiated: " + t.getMessage(),
t);
        }
    }
```

而这里 injectExtension 类，则是为生成的 instance 注入变量；

其目标是搜索所有 set 开头，同时只有一个入参的函数，执行该函数，对变量进行注入；

```
    private T injectExtension(T instance) {
        try {
            if (objectFactory != null) {
                for (Method method : instance.getClass().getMethods()) {
                    if (method.getName().startsWith("set")
                            && method.getParameterTypes().length == 1
                            && Modifier.isPublic(method.getModifiers())) {
                        Class<?> pt = method.getParameterTypes()[0];
                        try {
                            String property = method.getName().length() > 3 ?
method.getName().substring(3, 4).toLowerCase() +
method.getName().substring(4) : "";
                            Object object = objectFactory.getExtension(pt,
property);
                            if (object != null) {
```

```
                        method.invoke(instance, object);
                    }
                } catch (Exception e) {
                    logger.error("fail to inject via method " +
method.getName()
                            + " of interface " + type.getName() + ":
" + e.getMessage(), e);
                }
            }
        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}
```

此时我们的目光转到如下一段代码：

```
Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && wrapperClasses.size() > 0) {
            for (Class<?> wrapperClass : wrapperClasses) {
                instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
```

在分析这段代码的作用之前，我们先来分析一下

`Set<Class<?>> cachedWrapperClasses` 是如何被赋值的；

此时我们转到

`private void loadFile(Map<String, Class<?>> extensionClasses, String dir)`

函数：

```
if (clazz.isAnnotationPresent(Adaptive.class)) {
    if(cachedAdaptiveClass == null) {
        cachedAdaptiveClass = clazz;
    } else if (! cachedAdaptiveClass.equals(clazz)) {
        throw new IllegalStateException("More than 1 adaptive class found: "
                + cachedAdaptiveClass.getClass().getName()
                + ", " + clazz.getClass().getName());
    }
} else {
    try {
        clazz.getConstructor(type);
        Set<Class<?>> wrappers = cachedWrapperClasses;
        if (wrappers == null) {
            cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
            wrappers = cachedWrapperClasses;
        }
        wrappers.add(clazz);
    } catch (NoSuchMethodException e) {
```

分析：这里实际上是如果该类带有 Adaptive 注解，则认为是 cachedAdaptiveClass；若该类没有 Adaptive 注解，则判断该类是否带有参数是 type 类型的构造函数，若有，则认为是

wrapper 类；
于是我们分析如下文件
dubbo-rpc-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol
其内容为：

```
filter=com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper
listener=com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper
mock=com.alibaba.dubbo.rpc.support.MockProtocol
```

我们分析这三个类，会发现 mock 类没有参数为 Protocol 的自定义参数，而其他两个均有；
此时我们返回到 createExtension 函数：

```
Set<Class<?>> wrapperClasses = cachedWrapperClasses;
          if (wrapperClasses != null && wrapperClasses.size() > 0) {
              for (Class<?> wrapperClass : wrapperClasses) {
                  instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
              }
          }
```

此时可以发现这里对 instance 加了装饰类；对于 Protocol 来说加了两个装饰类 ProtocolFilterWrapper 和 ProtocolListenerWrapper；
也就/injectExtension 实例化包装类，并注入接口的适配器， 注意这个地方返回的是最后一个包装类。

6.3.2.3 扩展点装饰

6.3.2.3.1 Filter

这个地方非常重要，dubbo 机制里面日志记录、超时等等功能都是在这一部分实现的。

如上一节在介绍扩展点加载时所述，在生成 Protocol 的 invoker 时，实际上使用了装饰模式，第一个是 filter，第二个是 listener；

我们先来看 filter，具体 ProtocolFilterWrapper 类：

```
public class ProtocolFilterWrapper implements Protocol {

    private final Protocol protocol;

    public ProtocolFilterWrapper(Protocol protocol){
        if (protocol == null) {
            throw new IllegalArgumentException("protocol == null");
        }
        this.protocol = protocol;
    }

    public int getDefaultPort() {
        return protocol.getDefaultPort();
    }
```

```
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException
{
        if
(Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol()))
{
            return protocol.export(invoker);
        }
        return protocol.export(buildInvokerChain(invoker,
Constants.SERVICE_FILTER_KEY, Constants.PROVIDER));
    }

    public <T> Invoker<T> refer(Class<T> type, URL url) throws
RpcException {
        if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
            return protocol.refer(type, url);
        }
        return buildInvokerChain(protocol.refer(type, url),
Constants.REFERENCE_FILTER_KEY, Constants.CONSUMER);
    }

    public void destroy() {
        protocol.destroy();
    }

    private static <T> Invoker<T> buildInvokerChain(final Invoker<T>
invoker, String key, String group) {
        Invoker<T> last = invoker;
        List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension
(invoker.getUrl(), key, group);
        if (filters.size() > 0) {
            for (int i = filters.size() - 1; i >= 0; i --) {
                final Filter filter = filters.get(i);
                final Invoker<T> next = last;
                last = new Invoker<T>() {
                    public Class<T> getInterface() {
                        return invoker.getInterface();
                    }
                    public URL getUrl() {
                        return invoker.getUrl();
                    }
                    public boolean isAvailable() {
                        return invoker.isAvailable();
```

```
                }
                public Result invoke(Invocation invocation) throws
RpcException {
                    return filter.invoke(next, invocation);
                }

                public void destroy() {
                    invoker.destroy();
                }

                @Override
                public String toString() {
                    return invoker.toString();
                }
            };
        }
    }
    return last;
}
}
```

这个类有 3 个特点，第一它有一个参数为 Protocol protocol 的构造函数；第二，它实现了 Protocol 接口；第三，它使用职责链模式，对 export 和 refer 函数进行了封装；

对于函数封装，有点类似于 tomcat 的 filter 机制；我们来看 buildInvokerChain 函数：它读取所有的 filter 类，利用这些类封装 invoker；

```
List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension
(invoker.getUrl(), key, group);
```

我们看如下文件：

/dubbo-rpc-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.Filter

```
echo=com.alibaba.dubbo.rpc.filter.EchoFilter
generic=com.alibaba.dubbo.rpc.filter.GenericFilter
genericimpl=com.alibaba.dubbo.rpc.filter.GenericImplFilter
token=com.alibaba.dubbo.rpc.filter.TokenFilter
accesslog=com.alibaba.dubbo.rpc.filter.AccessLogFilter
activelimit=com.alibaba.dubbo.rpc.filter.ActiveLimitFilter
classloader=com.alibaba.dubbo.rpc.filter.ClassLoaderFilter
context=com.alibaba.dubbo.rpc.filter.ContextFilter
consumercontext=com.alibaba.dubbo.rpc.filter.ConsumerContextFilter
exception=com.alibaba.dubbo.rpc.filter.ExceptionFilter
executelimit=com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter
deprecated=com.alibaba.dubbo.rpc.filter.DeprecatedFilter
compatible=com.alibaba.dubbo.rpc.filter.CompatibleFilter
timeout=com.alibaba.dubbo.rpc.filter.TimeoutFilter
```

这其中涉及到很多功能，包括权限验证、异常、超时等等，当然可以预计计算调用时间等等应该也是在这其中的某个类实现的；

这里我们可以看到 export 和 refer 过程都会被 filter 过滤，那么如果记录接口调用时间时，服务器端部分只是记录接口在服务器端的执行时间，而客户端部分会记录接口在服务器端的执行时间+网络传输时间。

6.3.2.3.2 Listener

这里我们看到对应的类为：ProtocolListenerWrapper

```
public class ProtocolListenerWrapper implements Protocol {

    private final Protocol protocol;

    public ProtocolListenerWrapper(Protocol protocol){
        if (protocol == null) {
            throw new IllegalArgumentException("protocol == null");
        }
        this.protocol = protocol;
    }

    public int getDefaultPort() {
        return protocol.getDefaultPort();
    }

    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException
{
        if
(Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol()))
{
            return protocol.export(invoker);
        }
        return new ListenerExporterWrapper<T>(protocol.export(invoker),

Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(Expor
terListener.class)
                    .getActivateExtension(invoker.getUrl(),
Constants.EXPORTER_LISTENER_KEY)));
    }

    public <T> Invoker<T> refer(Class<T> type, URL url) throws
RpcException {
        if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
            return protocol.refer(type, url);
```

```
        }
        return new ListenerInvokerWrapper<T>(protocol.refer(type, url),
                Collections.unmodifiableList(

ExtensionLoader.getExtensionLoader(InvokerListener.class)
                        .getActivateExtension(url,
Constants.INVOKER_LISTENER_KEY)));
    }

    public void destroy() {
        protocol.destroy();
    }

}
```

在这里我们可以看到 export 和 refer 分别对应了不同的 Wrapper；
经过 debug，发现 ExporterListener 并没有实现类，同时通过 debug 也会发现 ListenerExporterWrapper 在执行过程中确实 listeners 变量是空的；
而对于 ListenerInvokerWrapper，我们发现在如下文件中：
/dubbo-rpc-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.InvokerListener
deprecated=com.alibaba.dubbo.rpc.listener.DeprecatedInvokerListener

而对于 DeprecatedInvokerListener 实际上只是实现了

```
@Activate(Constants.DEPRECATED_KEY)
public class DeprecatedInvokerListener extends InvokerListenerAdapter {

    private static final Logger LOGGER =
LoggerFactory.getLogger(DeprecatedInvokerListener.class);

    public void referred(Invoker<?> invoker) throws RpcException {
        if (invoker.getUrl().getParameter(Constants.DEPRECATED_KEY,
false)) {
            LOGGER.error("The service " + invoker.getInterface().getName()
+ " is DEPRECATED! Declare from " + invoker.getUrl());
        }
    }

}
```

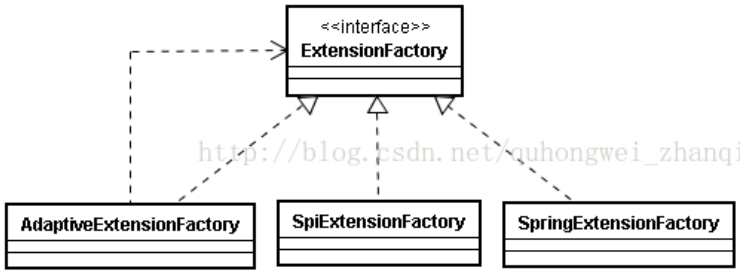
6.3.2.4 ExtensionFactory

ExtensionLoader.java 中注意到 injectExtension 函数是为了设置所生成的对象的 field；其

方法为对于有 set 函数的 field 进行设置。此时用到了 ExtensionFactory；说白了 ExtensionFactory 就是根据类型和名字来获取对象。

根据调用该类中带 set 开头（同时只有一个参数）的函数时，

下面我们来看看 ObjectFactory 是如何根据类型和名字来获取对象的，ObjectFactory 也是基于 dubbo 的 spi 扩展机制的：




```
/**
 * SpiExtensionFactory
 *
 * @author william.liangf
 */
public class SpiExtensionFactory implements ExtensionFactory {

    public <T> T getExtension(Class<T> type, String name) {
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
            ExtensionLoader<T> loader =
ExtensionLoader.getExtensionLoader(type);
            if (loader.getSupportedExtensions().size() > 0) {
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }
}

/**
 * SpringExtensionFactory
 *
 * @author william.liangf
 */
public class SpringExtensionFactory implements ExtensionFactory {

    private static final Set<ApplicationContext> contexts = new
ConcurrentHashSet<ApplicationContext>();

    public static void addApplicationContext(ApplicationContext context)
{
```

```
        contexts.add(context);
    }

    public static void removeApplicationContext(ApplicationContext
context) {
        contexts.remove(context);
    }

    @SuppressWarnings("unchecked")
    public <T> T getExtension(Class<T> type, String name) {
        for (ApplicationContext context : contexts) {
            if (context.containsBean(name)) {
                Object bean = context.getBean(name);
                if (type.isInstance(bean)) {
                    return (T) bean;
                }
            }
        }
        return null;
    }

}
```

它跟 Compiler 接口一样设配类注解@Adaptive 是打在类 AdaptiveExtensionFactory 上的不是通过 javassist 编译生成的。

AdaptiveExtensionFactory 持有所有 ExtensionFactory 对象的集合，dubbo 内部默认实现的对象工厂是 SpiExtensionFactory 和 SpringExtensionFactory，他们经过 TreeMap 排好序的查找顺序是优先先从 SpiExtensionFactory 获取，如果返回空在从 SpringExtensionFactory 获取。

这里我们可以看如下两个文件：

/dubbo-common/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.common.extension.ExtensionFactory

```
adaptive=com.alibaba.dubbo.common.extension.factory.AdaptiveExtension
Factory
spi=com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory
```

/dubbo-config-spring/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.common.extension.ExtensionFactory

```
spring=com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory
```

这里我们来分一下 ObjectFactory 的初始化过程，会对 SPI 及 Adaptive 注解机制有更深入的了解：

```
    private ExtensionLoader(Class<?> type) {
        this.type = type;
```

```
        objectFactory = (type == ExtensionFactory.class ? null :
ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiv
eExtension());
    }
```

这里我们看到 *getExtensionLoader 会是一个死循环，因为会不断调用；但是实际上在* getAdaptiveExtension 中会直接返回被 Adaptive 注解的类，因此避免了死循环；

在 getExtensionClasses 函数中，在读取文件加载类的过程过程中，会判断该类是否带有 Adaptive 注解，如果是，则直接赋值：

loadFile 函数中：

```
    if (clazz.isAnnotationPresent(Adaptive.class)) {
        if(cachedAdaptiveClass == null) {
            cachedAdaptiveClass = clazz;
        } else if (! cachedAdaptiveClass.equals(clazz)) {
            throw new IllegalStateException("More than 1 adaptive class found: "
                    + cachedAdaptiveClass.getClass().getName()
                    + ", " + clazz.getClass().getName());
        }
    } else {
```

这里同时可以确认，对于一个 spi 接口，有且只有一个类带有 adaptive 注解，否则会出错；

这里我们继续分析 AdaptiveExtensionFactory 类：

```
@Adaptive
public class AdaptiveExtensionFactory implements ExtensionFactory {

    private final List<ExtensionFactory> factories;

    public AdaptiveExtensionFactory() {
        ExtensionLoader<ExtensionFactory> loader =
ExtensionLoader.getExtensionLoader(ExtensionFactory.class);
        List<ExtensionFactory> list = new ArrayList<ExtensionFactory>();
        for (String name : loader.getSupportedExtensions()) {
            list.add(loader.getExtension(name));
        }
        factories = Collections.unmodifiableList(list);
    }

    public <T> T getExtension(Class<T> type, String name) {
        for (ExtensionFactory factory : factories) {
            T extension = factory.getExtension(type, name);
            if (extension != null) {
                return extension;
            }
        }
        return null;
    }
}
```

从这里我们看出，ObjectFactory. getExtension(Class<T> type, String name)是先从 SpiExtensionFactory 获得扩展点，再从 SpringExtensionFactory 获得扩展点；

6.4 代理

6.4.1 Invoker 调用

代理这里需要搞清楚 java 的几种生成代理的方式：普通代理、JDK、Javassist 库动态代理、Javassist 库动态字节码代理；

生成代理的目的是你调用 invoker 的相关函数后，就会等于是调用 DubboInvoker 中的相关函数；也就是将本地调用转为网络调用并获得结果；

// 创建服务代理

return (T) proxyFactory.getProxy(invoker);

private static final ProxyFactory proxyFactory = ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();

ProxyFactory.java

```
package com.alibaba.dubbo.rpc;
@SPI("javassist")
public interface ProxyFactory {

    /**
     * create proxy.
     *
     * @param invoker
     * @return proxy
     */
    @Adaptive({Constants.PROXY_KEY})  proxy
    <T> T getProxy(Invoker<T> invoker) throws RpcException;

    /**
     * create invoker.
     *
     * @param <T>
     * @param proxy
     * @param type
     * @param url
     * @return invoker
     */
    @Adaptive({Constants.PROXY_KEY})
```

```
    <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) throws
RpcException;

}
```

```
stub=com.alibaba.dubbo.rpc.proxy.wrapper.StubProxyFactoryWrapper
jdk=com.alibaba.dubbo.rpc.proxy.jdk.JdkProxyFactory
javassist=com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory
```

```
    @SuppressWarnings("unchecked")
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
        return (T) Proxy.getProxy(interfaces).newInstance(new
InvokerInvocationHandler(invoker));
    }
```

而我们看到 InvokerInvocationHandler 类时，就知道具体的代理是如何工作的；

```
public class InvokerInvocationHandler implements InvocationHandler {

    private final Invoker<?> invoker;

    public InvokerInvocationHandler(Invoker<?> handler){
        this.invoker = handler;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        String methodName = method.getName();
        Class<?>[] parameterTypes = method.getParameterTypes();
        if (method.getDeclaringClass() == Object.class) {
            return method.invoke(invoker, args);
        }
        if ("toString".equals(methodName) && parameterTypes.length == 0)
{
            return invoker.toString();
        }
        if ("hashCode".equals(methodName) && parameterTypes.length == 0)
{
            return invoker.hashCode();
        }
        if ("equals".equals(methodName) && parameterTypes.length == 1) {
            return invoker.equals(args[0]);
        }
        return invoker.invoke(new RpcInvocation(method,
args)).recreate();
```

```
    }

}
```

此时我们在 DubboInvoker.java 类中可以看到 invoke 函数中最终会调用 doInvoke 函数，此时转为网络调用；

这里我们通过 RpcInvocation 对象，可以确认客户端传递给 invoker 的信息：

```
    public RpcInvocation(Invocation invocation, Invoker<?> invoker) {
        this(invocation.getMethodName(),
invocation.getParameterTypes(),
                invocation.getArguments(), new HashMap<String,
String>(invocation.getAttachments()),
                invocation.getInvoker());
        if (invoker != null) {
            URL url = invoker.getUrl();
            setAttachment(Constants.PATH_KEY, url.getPath());
            if (url.hasParameter(Constants.INTERFACE_KEY)) {
                setAttachment(Constants.INTERFACE_KEY,
url.getParameter(Constants.INTERFACE_KEY));
            }
            if (url.hasParameter(Constants.GROUP_KEY)) {
                setAttachment(Constants.GROUP_KEY,
url.getParameter(Constants.GROUP_KEY));
            }
            if (url.hasParameter(Constants.VERSION_KEY)) {
                setAttachment(Constants.VERSION_KEY,
url.getParameter(Constants.VERSION_KEY, "0.0.0"));
            }
            if (url.hasParameter(Constants.TIMEOUT_KEY)) {
                setAttachment(Constants.TIMEOUT_KEY,
url.getParameter(Constants.TIMEOUT_KEY));
            }
            if (url.hasParameter(Constants.TOKEN_KEY)) {
                setAttachment(Constants.TOKEN_KEY,
url.getParameter(Constants.TOKEN_KEY));
            }
            if (url.hasParameter(Constants.APPLICATION_KEY)) {
                setAttachment(Constants.APPLICATION_KEY,
url.getParameter(Constants.APPLICATION_KEY));
            }
        }
}
```

6.4.2 JDK 代理

这里实际上只是根据 JDK 代理机制来生成相应的代理

```
public class JdkProxyFactory extends AbstractProxyFactory {

    @SuppressWarnings("unchecked")
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
        return (T)
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
interfaces, new InvokerInvocationHandler(invoker));
    }

    public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
        return new AbstractProxyInvoker<T>(proxy, type, url) {
            @Override
            protected Object doInvoke(T proxy, String methodName,
                                      Class<?>[] parameterTypes,
                                      Object[] arguments) throws Throwable {
                Method method = proxy.getClass().getMethod(methodName,
parameterTypes);
                return method.invoke(proxy, arguments);
            }
        };
    }

}
```

6.4.3 Javaassist 代理（动态）

```
proxyFactory.getProxy(invoker)  --》
JavassistProxyFactory-》
(T) Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandl
er(invoker));
```

这里使用 Javassist 创建了两个 CLASS。

```
public class Proxy1 extends Proxy
  implements ClassGenerator.DC
{
  public Object newInstance(InvocationHandler paramInvocationHandler)
  {
    return new proxy1(paramInvocationHandler);
  }
```

```
    }

public class proxy1
  implements ClassGenerator.DC, EchoService, DemoService
{
  public static Method[] methods;
  private InvocationHandler handler;

  public String sayHello(String paramString)
  {
    Object[] arrayOfObject = new Object[1];
    arrayOfObject[0] = paramString;
    Object localObject =
this.jdField_handler_of_type_JavaLangReflectInvocationHandler.invoke(
this, jdField_methods_of_type_ArrayOfJavaLangReflectMethod[0],
arrayOfObject);
    return ((String)localObject);
  }

  public Object $echo(Object paramObject)
  {
    Object[] arrayOfObject = new Object[1];
    arrayOfObject[0] = paramObject;
    Object localObject =
this.jdField_handler_of_type_JavaLangReflectInvocationHandler.invoke(
this, jdField_methods_of_type_ArrayOfJavaLangReflectMethod[1],
arrayOfObject);
    return ((Object)localObject);
  }

  public proxy1(InvocationHandler paramInvocationHandler)
  {
    this.jdField_handler_of_type_JavaLangReflectInvocationHandler =
paramInvocationHandler;
  }
    }
```

返回 proxy1，xxx 接口的子类。

这里有个小问题不理解：

在 Javassist 创建类的时候，实际上在调用时:

```
    public ClassGenerator addMethod(String name, int mod, Class<?> rt,
Class<?>[] pts, Class<?>[] ets, String body)
    {
        StringBuilder sb = new StringBuilder();
```

```
        sb.append(modifier(mod)).append('
').append(ReflectUtils.getName(rt)).append(' ').append(name);
        sb.append('(');
        for(int i=0;i<pts.length;i++)
        {
            if( i > 0 )
                sb.append(',');
            sb.append(ReflectUtils.getName(pts[i]));
            sb.append(" arg").append(i);
        }
        sb.append(')');
        if( ets != null && ets.length > 0 )
        {
            sb.append(" throws ");
            for(int i=0;i<ets.length;i++)
            {
                if( i > 0 )
                    sb.append(',');
                sb.append(ReflectUtils.getName(ets[i]));
            }
        }
        sb.append('{').append(body).append('}');
        return addMethod(sb.toString());
    }
```

此时生成的 sb 字段内容如下：

```
public java.lang.String sayHello(java.lang.String arg0)
{
    Object[] args = new Object[1];
    args[0] = ($w)$1;
    Object ret = handler.invoke(this, methods[0], args);
    return (java.lang.String)ret;
}
```

没搞明白这里的($w)$1 机制是如何转化为参数对象的；

```
public String sayHello(String paramString)
  {
    Object[] arrayOfObject = new Object[1];
    arrayOfObject[0] = paramString;
    Object localObject =
this.jdField_handler_of_type_JavaLangReflectInvocationHandler.invoke(
this, jdField_methods_of_type_ArrayOfJavaLangReflectMethod[0],
arrayOfObject);
    return ((String)localObject);
  }
```

6.5 远程调用流程

在我看来默认协议的 rpc 过程是比较复杂的，其中涉及到了各个方面，其余各协议实际上有对这个过程进行简化；因此看懂了默认协议的 rpc 过程，其他协议就非常容易懂了。

Java 远程通讯可选技术及原理 本文对 java rpc 机制有非常详细的介绍

http://wenku.baidu.com/view/c5851a4502768e9951e738bb.html

6.5.1 通信过程

我们可以通过如下 7 点分析 RPC 通信过程：

1、是基于什么协议实现的？

2、怎么发起请求？

3、怎么将请求转化为符合协议的格式的？

4、使用什么传输协议传输？

5、响应端基于什么机制来接收请求？

6、怎么将流还原为传输格式的？

7、处理完毕后怎么回应？

此时我们用默认协议来分析：

1) 是基于什么协议实现的？

Dubbo 协议（通信使用 netty 协议）

2) 怎么发起请求？

请求内容封装在 RpcInvocation 对象，利用 netty 客户端发请求

3) 怎么将请求转化为符合协议的格式的？

对象序列化（其实还包含编码及解码过程）

4) 使用什么传输协议传输？

Netty（本质是 NIO 及 socket）

5) 响应端基于什么机制来接收请求？

Netty 的响应机制

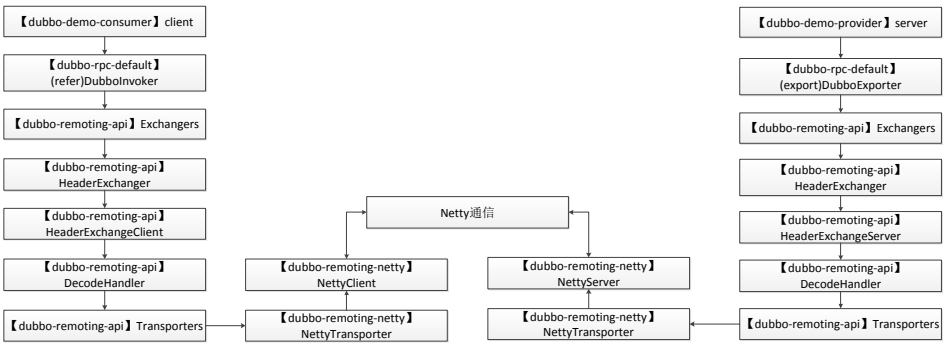
6) 怎么将流还原为传输格式的？

反序列化

7) 处理完毕后怎么回应？

回应内容封装为 RpcResult 对象中，序列化后通过 netty 传给客户端

这个流程可以通过下图表示：



要了解这一过程，最好的方式就是在各处设置断点，然后在 consumer 端和 provider 端 debug 几次。

6.5.2 序列化

这一部分需要先阅读如下 4 篇文章了解 java 序列化的基本概念及原理；

Java 序列化算法透析 有对序列化原理进行详细描述，包括默认序列化方式下生成的文件格式

http://longdick.iteye.com/blog/458557

Java 序列化的高级认识 基础知识

http://www.ibm.com/developerworks/cn/java/j-lo-serial/

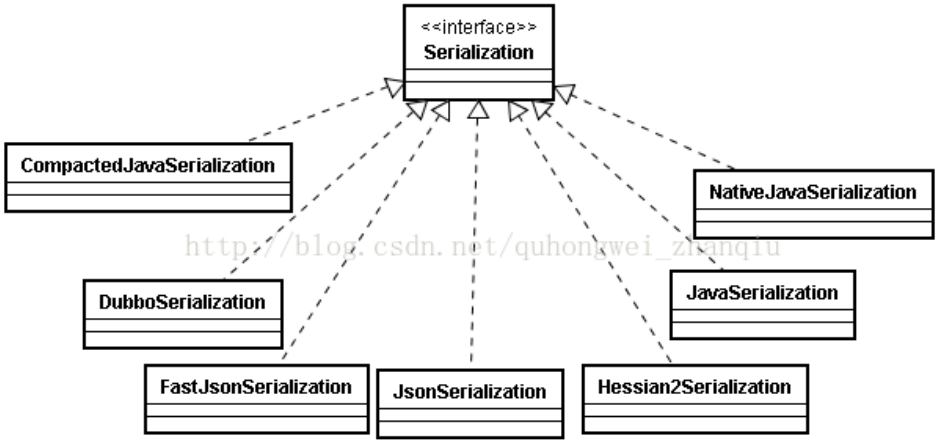
详解 java 序列化（一） 两篇文章，对序列化和反序列化过程中涉及到父类与子类的问题进行描述

http://blog.csdn.net/moreevan/article/details/6697777

序列化的几种方式 对生成文件大小进行分析

http://my-corner.iteye.com/blog/1776512

序列化：dubbo 提供了一系列的序列化反序列化对象工具。



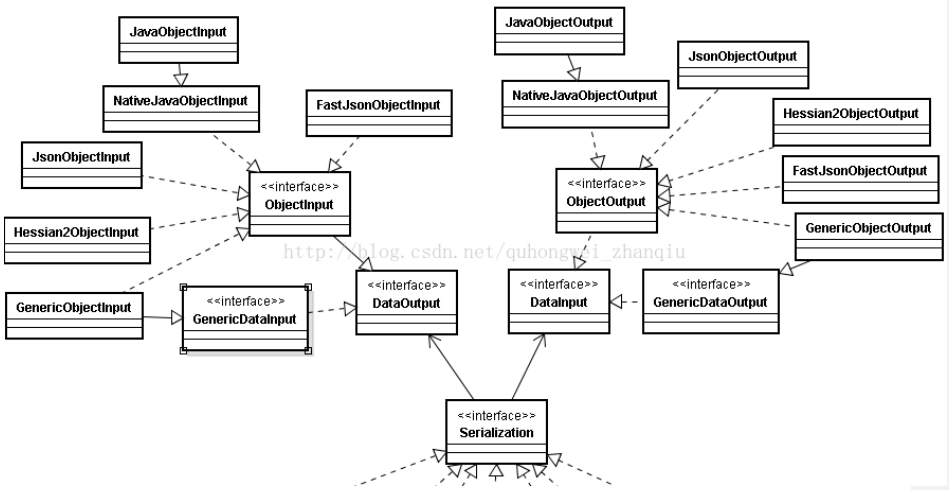
Serialization 接口定义

```
@SPI("hessian2")
public interface Serialization {
    /**
     * get content type id
     *
     * @return content type id
     */
    byte getContentTypeId();
    /**
     * get content type
     *
```

```
     * @return content type
     */
    String getContentType();
    /**
     * create serializer
     * @param url
     * @param output
     * @return serializer
     * @throws IOException
     */
    @Adaptive
    ObjectOutput serialize(URL url, OutputStream output) throws
IOException;
    /**
     * create deserializer
     * @param url
     * @param input
     * @return deserializer
     * @throws IOException
     */
    @Adaptive
    ObjectInput deserialize(URL url, InputStream input) throws
IOException;
}
```

SPI 注解指定了序列化的默认实现为 hessian2；

Serialization 依赖于 jdk 的 OutputStream，InputStream，因为各具体的序列化工具依赖于 OutputStream，InputStream。又为了屏蔽各个序列化接口对 dubbo 侵入 dubbo 定义统一的 DataOutput DataInput 接口来适配各种序列化工具的输入输出；



我们拿默认的序列化 Hessian2Serialization 来举例来说明

```
public class Hessian2Serialization implements Serialization {
```

```
    public static final byte ID = 2;
    public byte getContentTypeId() {
        return ID;
    }
    public String getContentType() {
        return "x-application/hessian2";
    }
    public ObjectOutput serialize(URL url, OutputStream out) throws
IOException {
        return new Hessian2ObjectOutput(out);
    }
    public ObjectInput deserialize(URL url, InputStream is) throws
IOException {
        return new Hessian2ObjectInput(is);
    }
    }
```

Hessian2Serialization 构建基于 Hessian 的 Dubbo 接口 Output,Input 实现， Dubbo 是基于 Output, Input 接口操作不需要关心具体的序列化反序列化实现方式。

```
public class Hessian2ObjectOutput implements ObjectOutput
{
    private final Hessian2Output mH2o;
    public Hessian2ObjectOutput(OutputStream os)
    {
        mH2o = new Hessian2Output(os);

    mH2o.setSerializerFactory(Hessian2SerializerFactory.SERIALIZER_FA
CTORY);
    }
    public void writeBool(boolean v) throws IOException
    {
        mH2o.writeBoolean(v);
    }

     ……中间这部分省略，节省篇幅

    public void writeBytes(byte[] b, int off, int len) throws IOException
    {
        mH2o.writeBytes(b, off, len);
    }
    public void writeUTF(String v) throws IOException
    {
        mH2o.writeString(v);
    }
    public void writeObject(Object obj) throws IOException
```

```
    {
        mH2o.writeObject(obj);
    }
    public void flushBuffer() throws IOException
    {
        mH2o.flushBuffer();
    }
    }
```

Hessian2ObjectInput 读取 Hessian 序列化的数据使用方式同上面类似就不在堆代码了请自己翻看源代码；

实际上：1）序列化：读取对象字段，按照一定格式写入文件（当然也可以使其他媒介）；2）反序列化：利用反射机制生成类对象，从媒介中读取对象信息，将这些字段信息赋给对象。

6.5.3 Encode 和 Decode

在 dubbo 协议的传输过程中，client 并非只是单纯将 RpcInvocation 对象序列化后传递给 server，同理 server 也不是将 RpcResult 对象反序列化后传递给 client；其中涉及到编解码的过程；

这里我们看到 NettyClient.java 代码中（NettyServer.java 也有类似）：

```
        bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
            public ChannelPipeline getPipeline() {
                NettyCodecAdapter adapter = new
NettyCodecAdapter(getCodec(), getUrl(), NettyClient.this);
                ChannelPipeline pipeline = Channels.pipeline();
                pipeline.addLast("decoder", adapter.getDecoder());
                pipeline.addLast("encoder", adapter.getEncoder());
                pipeline.addLast("handler", nettyHandler);
                return pipeline;
            }
            });
```

这里实际上是 netty 机制中会对 channle 中的内容进行编解码；

而我们看 NettyCodecAdapter 中，实际上处理编解码的是 Codec2，这是一个接口，其具体的类在 dubbo-rpc-default 工程中的 com.alibaba.dubbo.remoting.Codec2 文件中定义：

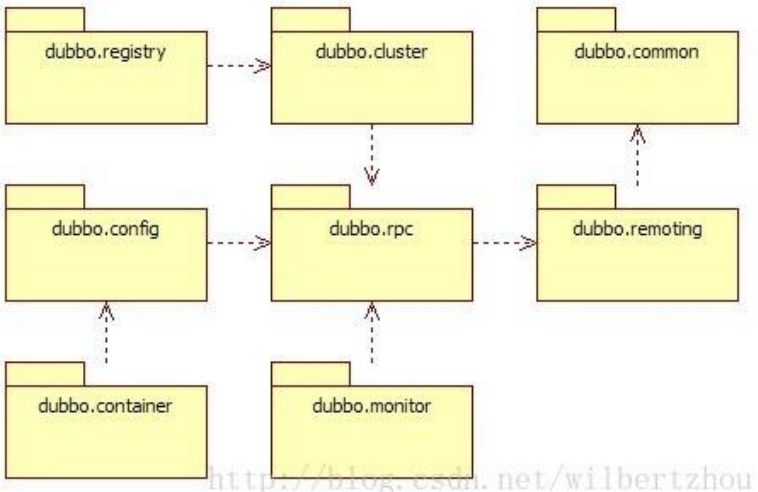
```
dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboCountCodec
```

最后真正做编解码任务的是 DubboCodec，除了对 RpcInvocation 和 RpcResult 序列化或者反序列化外，还会加入一些其他信息；

7 过程分析

7.1 Refer & export

7.1.1 调用顺序




```
<dubbo:reference id="xxxService" interface="xxx.xxx.Service"/>
```

```
public Object getObject() throws Exception {
        return get();
    }

public synchronized T get() {
        if (destroyed){
            throw new IllegalStateException("Already destroyed!");
        }
              if (ref == null) {
                        init();
              }
              return ref;
    }

private void init() {
                  if (initialized) {
                      return;
                  }
```

```
            initialized = true;
        省略....
    ref = createProxy(map);//在这里使用了动态代理生成对象
}
```

ref = createProxy(map);//在这里使用了动态代理生成了代理对象(这里也可以成为远程代理，因为在这个代理中进行了远程调用),ref 即 getBean 返回的对象，这样在 action 调用 demoService 是就会使用代理处理器 com.alibaba.dubbo.rpc.proxy.InvokerInvocationHandler

```
@SuppressWarnings({ "unchecked", "rawtypes", "deprecation" })
    private T createProxy(Map<String, String> map) {
        URL tmpUrl = new URL("temp", "localhost", 0, map);
        final boolean isJvmRefer;
        if (isInjvm() == null) {
            if (url != null && url.length() > 0) { //指定URL的情况下，不做本地引用
                isJvmRefer = false;
            } else if (InjvmProtocol.getInjvmProtocol().isInjvmRefer(tmpUrl)) {
                //默认情况下如果本地有服务暴露，则引用本地服务.
                isJvmRefer = true;
            } else {
                isJvmRefer = false;
            }
        } else {
            isJvmRefer = isInjvm().booleanValue();
        }

        if (isJvmRefer) {
            URL url = new URL(Constants.LOCAL_PROTOCOL, NetUtils.LOCALHOST, 0, interfaceClass.getName()).addParameters(map);
            invoker = refprotocol.refer(interfaceClass, url);
            if (logger.isInfoEnabled()) {
                logger.info("Using injvm service " + interfaceClass.getName());
            }
        } else {
            if (url != null && url.length() > 0) { // 用户指定URL，指定的URL可能是对点对直连地址，也可能是注册中心URL
                String[] us = Constants.SEMICOLON_SPLIT_PATTERN.split(url);
                if (us != null && us.length > 0) {
                    for (String u : us) {
                        URL url = URL.valueOf(u);
```

```
                    if (url.getPath() == null || url.getPath().length()
== 0) {
                        url = url.setPath(interfaceName);
                    }
                    if
(Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {

urls.add(url.addParameterAndEncoded(Constants.REFER_KEY,
StringUtils.toQueryString(map)));
                    } else {
                        urls.add(ClusterUtils.mergeUrl(url, map));
                    }
                }
            }
        } else { // 通过注册中心配置拼装URL
            List<URL> us = loadRegistries(false);
            if (us != null && us.size() > 0) {
                for (URL u : us) {
                    URL monitorUrl = loadMonitor(u);
                    if (monitorUrl != null) {
                        map.put(Constants.MONITOR_KEY,
URL.encode(monitorUrl.toFullString()));
                    }

urls.add(u.addParameterAndEncoded(Constants.REFER_KEY,
StringUtils.toQueryString(map)));
                }
            }
            if (urls == null || urls.size() == 0) {
                throw new IllegalStateException("No such any registry
to reference " + interfaceName  + " on the consumer " +
NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion()
+ ", please config <dubbo:registry address=\"...\" /> to your spring
config.");
            }
        }

        if (urls.size() == 1) {
            invoker = refprotocol.refer(interfaceClass, urls.get(0));
        } else {
            List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();
            URL registryURL = null;
            for (URL url : urls) {
                invokers.add(refprotocol.refer(interfaceClass, url));
```

```
                if
(Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                    registryURL = url; // 用了最后一个registry url
                }
            }
            if (registryURL != null) { // 有 注册中心协议的URL
                // 对有注册中心的Cluster 只用 AvailableCluster
                URL u = registryURL.addParameter(Constants.CLUSTER_KEY,
AvailableCluster.NAME);
                invoker = cluster.join(new StaticDirectory(u,
invokers));
            } else { // 不是 注册中心的URL
                invoker = cluster.join(new StaticDirectory(invokers));
            }
        }
    }

    Boolean c = check;
    if (c == null && consumer != null) {
        c = consumer.isCheck();
    }
    if (c == null) {
        c = true; // default true
    }
    if (c && ! invoker.isAvailable()) {
        throw new IllegalStateException("Failed to check the status of
the service " + interfaceName + ". No provider available for the service
" + (group == null ? "" : group + "/") + interfaceName + (version == null ?
"" : ":" + version) + " from the url " + invoker.getUrl() + " to the consumer
" + NetUtils.getLocalHost() + " use dubbo version " +
Version.getVersion());
    }
    if (logger.isInfoEnabled()) {
        logger.info("Refer dubbo service " + interfaceClass.getName()
+ " from url " + invoker.getUrl());
    }
    // 创建服务代理
    return (T) proxyFactory.getProxy(invoker);
}
```

7.1.2 生成 Invoker

这里调用 invoker = refprotocol.refer(interfaceClass, urls.get(0));来生成 invoker；
其代码如下：

```
    public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws
RpcException {
        // create rpc invoker.
        DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url,
getClients(url), invokers);
        invokers.add(invoker);
        return invoker;
    }
```

1、目前客户端 invoke 后，调用 DubboInvoker 的 doInvoke 函数；
2、doInvoke 函数中，通过 client 的 send 函数发送数据或者执行 currentClient.request；

```
    @Override
    protected Result doInvoke( final Invocation invocation) throws Throwable {
        RpcInvocation inv = (RpcInvocation) invocation;
        final String methodName = RpcUtils.getMethodName(invocation);
        inv.setAttachment(Constants. PATH_KEY, getUrl().getPath());
        inv.setAttachment(Constants. VERSION_KEY, version );

        ExchangeClient currentClient;
        if (clients .length == 1) {
            currentClient = clients[0];
        } else {
            currentClient = clients[index.getAndIncrement() % clients. lengt
h];
        }
        try {
            boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
            boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
            int timeout = getUrl().getMethodParameter(methodName,
Constants.TIMEOUT_KEY ,Constants.DEFAULT_TIMEOUT);
```

```
            if (isOneway) {
                boolean isSent = getUrl().getMethodParameter(methodName,
Constants.SENT_KEY, false );
                currentClient.send(inv, isSent);
                RpcContext. getContext().setFuture(null);
                return new RpcResult();
            } else if (isAsync) {
                ResponseFuture future = currentClient.request(inv, timeout) ;
                RpcContext. getContext().setFuture(new FutureAdapter<Object
>(future));
                return new RpcResult();
            } else {
                RpcContext. getContext().setFuture(null);
                return (Result) currentClient.request(inv, timeout).get();
            }
        } catch (TimeoutException e) {
            throw new RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke
remote method timeout. method: " + invocation.getMethodName() + ", provider: " +
getUrl() + ", cause: " + e.getMessage(), e);
        } catch (RemotingException e) {
            throw new RpcException(RpcException.NETWORK_EXCEPTION, "Failed to
invoke remote method: " + invocation.getMethodName() + ", provider: " + getUrl()
+ ", cause: " + e.getMessage(), e);
        }
    }
```

3、NettyClient 的 send 函数

```
    public void send(Object message, boolean sent) throws RemotingException {
```

```
        if (send_reconnect && !isConnected()){
            connect();
        }
        Channel channel = getChannel();
        //TODO getChannel 构造失败了，不现实连接，等待 ull 阿姆斯特丹
        if (channel == null || ! channel.isConnected()) {
          throw new RemotingException(this, "message can not send, because
channel is closed . url:" + getUrl());
        }
        channel.send(message, sent);
    }
```

此时实际上是利用 netty 的网络通信机制进行通信；

而 netty 的机制保证数据接收处理：

```
 @Override
    protected void doOpen() throws Throwable {
        NettyHelper. setNettyLoggerFactory();
        bootstrap = new ClientBootstrap( channelFactory);
        // config
        // @see org.jboss.netty.channel.socket.SocketChannelConfig
        bootstrap.setOption("keepAlive" , true);
        bootstrap.setOption("tcpNoDelay" , true);
        bootstrap.setOption("connectTimeoutMillis" , getTimeout());
        final NettyHandler nettyHandler = new NettyHandler(getUrl(), this);
        bootstrap.setPipelineFactory(new ChannelPipelineFactory () {
            public ChannelPipeline getPipeline() {
                NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(),
getUrl(), NettyClient.this );
```

```
                ChannelPipeline pipeline = Channels.pipeline();
                pipeline.addLast( "decoder", adapter.getDecoder());
                pipeline.addLast( "encoder", adapter.getEncoder());
                pipeline.addLast( "handler", nettyHandler);
                return pipeline;
            }
        });
    }
```

此时若收到数据，则会调用 NettyHandler 的 messageReceived 函数

```
    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) t
hrows Exception {
        NettyChannel channel =
NettyChannel. getOrAddChannel(ctx.getChannel(), url, handler );
        try {
            handler.received(channel, e.getMessage());
        } finally {
            NettyChannel. removeChannelIfDisconnected(ctx.getChannel());
        }
    }
```

此时，会发现，最后是调用 DubboProtocol 中 requestHandler 的 received 函数
在 Handler 的调用链中实际上是类似职责链模式（类似 tomcat 的 filter），外层的 Handler 调用内层的 Handler，并在调用前后加上一些逻辑；
最后，在 received 函数中：

```
        @Override
        public void received(Channel channel, Object
message) throws RemotingException {
            if (message instanceof Invocation) {
```

```
                reply((ExchangeChannel) channel, message);
            } else {
                super.received(channel, message);
            }
        }
```

这里可以发现同时会发现 super .received(channel, message)是空函数，因此若是客户端处理到这里，就不做任何处理了；
所以可以看到，如果是服务器端，调用的是 reply 函数：

```
        public Object reply(ExchangeChannel channel, Object
message) throws RemotingException {
            if (message instanceof Invocation) {
                Invocation inv = (Invocation) message;
                Invoker<?> invoker = getInvoker(channel, inv);
                //如果是 callback 需要处理高版本调用低版本的问题
                if (Boolean.TRUE.toString().equals(inv.getAttachments().get(
IS_CALLBACK_SERVICE_INVOKE))){
                    String methodsStr =
invoker.getUrl().getParameters().get("methods" );
                    boolean hasMethod = false;
                    if (methodsStr == null || methodsStr.indexOf(",") == -1){
                        hasMethod = inv.getMethodName().equals(methodsStr);
                    } else {
                        String[] methods = methodsStr.split( ",");
                        for (String method : methods){
                            if (inv.getMethodName().equals(method)){
                                hasMethod = true;
                                break;
                            }
```

```
                    }
                }

                if (!hasMethod){

                    logger.warn( new IllegalStateException("The
methodName "+inv.getMethodName()+" not found in callback service

interface ,invoke will be ignored. please update the api interface. url is:" +

invoker.getUrl()) + " ,invocation is :"+inv );

                    return null ;

                }
            }
            RpcContext.getContext().setRemoteAddress(channel.getRemoteA
ddress());

            return invoker.invoke(inv);

        }

        throw new RemotingException(channel, "Unsupported request: " +

message == null ? null : (message.getClass().getName() + ": " + message) + ",

channel: consumer: " + channel.getRemoteAddress() + " --> provider: " +

channel.getLocalAddress());
    }
```

此时 getInvoker 实际上是从 exporter 中去获取 Invoker；
这里通过 if (message instanceof Invocation)可知，如果不是调用，则此时不做其他处理，也就是客户端调用在这里是不做任何处理的。
因此我们返回到客户端的调用中，会发现 HeaderExchangeClient 的如下函数处理：

```
    public ResponseFuture request(Object request) throws RemotingException {

        return channel .request(request);

    }
```

此时，转到 HeaderExchangeChannel：

```
    public ResponseFuture request(Object

request, int timeout) throws RemotingException {

        if (closed ) {
```

```
            throw new RemotingException(this.getLocalAddress(), null, "Faile
d to send request " + request + ", cause: The channel " + this + " is closed!" );
        }
        // create request.
        Request req = new Request();
        req.setVersion( "2.0.0");
        req.setTwoWay( true);
        req.setData(request);
        DefaultFuture future = new DefaultFuture(channel , req, timeout);
        try{
            channel.send(req);
        } catch (RemotingException e) {
            future.cancel();
            throw e;
        }
        return future;
    }
```

实际上这里返回的是 DefaultFuture；

而从 doInvoke 函数中：

return (Result) currentClient.request(inv, timeout).get();

此时返回前端的是一个 Result 的子对象；

而我们返回到客户端的调用 InvokerInvocationHandler：

```
    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
        String methodName = method.getName();
        Class<?>[] parameterTypes = method.getParameterTypes();
        if (method.getDeclaringClass() == Object.class) {
            return method.invoke(invoker , args);
```

```
        }
        if ("toString" .equals(methodName) && parameterTypes.length == 0) {
            return invoker .toString();
        }
        if ("hashCode" .equals(methodName) && parameterTypes.length == 0) {
            return invoker .hashCode();
        }
        if ("equals" .equals(methodName) && parameterTypes.length == 1) {
            return invoker .equals(args[0]);
        }
        return invoker .invoke(new RpcInvocation(method, args)).recreate();
    }
```

此时会发现最后结果是从 Result 的 recreate()函数返回而来，实际上这是一个；
此时我们可以猜测服务器端返回给客户端的是一个 Response 对象，同时其中的 mResult（private Object mResult）是一个实现了 Result 接口的对象；
我搜索了一下代码，目前只有 RpcResult 对象实现了 Result 接口，因此可以肯定的是服务器端返回给客户端的是一个 RpcResult 对象；

Invoker 屏蔽了通信相关的细节；
此时需要注意的是默认使用的网络实现是 netty；

7.1.3 export

ServiceBean<T>类在设置属性后，调用 export 函数

```
    @SuppressWarnings({ "unchecked", "deprecation" })
    public void afterPropertiesSet() throws Exception {
        ……
        if (! isDelay()) {
            export();
        }
    }
```

此后调用的是 ServiceConfig 中的相关函数：

```
    @SuppressWarnings({ "unchecked", "rawtypes" })
```

```
    private void doExportUrls() {
        List<URL> registryURLs = loadRegistries(true);
        for (ProtocolConfig protocolConfig : protocols) {
            doExportUrlsFor1Protocol(protocolConfig, registryURLs);
        }
    }
```

最终实现的是如下逻辑：

```
     Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class)
interfaceClass, url);

     Exporter<?> exporter = protocol.export(invoker);
      exporters.add(exporter);
```

此时找到 DubboProtocol 类

```
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException
{
        URL url = invoker.getUrl();

        // export service.
        String key = serviceKey(url);
        DubboExporter<T> exporter = new DubboExporter<T>(invoker, key,
exporterMap);
        exporterMap.put(key, exporter);

        //export an stub service for dispaching event
        Boolean isStubSupportEvent =
url.getParameter(Constants.STUB_EVENT_KEY,Constants.DEFAULT_STUB_EVEN
T);
        Boolean isCallbackservice =
url.getParameter(Constants.IS_CALLBACK_SERVICE, false);
        if (isStubSupportEvent && !isCallbackservice){
            String stubServiceMethods =
url.getParameter(Constants.STUB_EVENT_METHODS_KEY);
            if (stubServiceMethods == null || stubServiceMethods.length()
== 0 ){
                if (logger.isWarnEnabled()){
                    logger.warn(new IllegalStateException("consumer ["
+url.getParameter(Constants.INTERFACE_KEY) +
                            "], has set stubproxy support event ,but no stub
methods founded."));
                }
            } else {
                stubServiceMethodsMap.put(url.getServiceKey(),
stubServiceMethods);
```

```
            }
        }
        openServer(url);
        return exporter;
    }
```

接下来，在 openServer 和 createServer 中，最终调用的是如下内容：

```
            server = Exchangers.bind(url, requestHandler);
```

此时我们转到 HeaderExchanger：

```
    public ExchangeServer bind(URL url, ExchangeHandler handler) throws
RemotingException {
        return new HeaderExchangeServer(Transporters.bind(url, new
DecodeHandler(new HeaderExchangeHandler(handler))));
    }
```

剩下的逻辑就是生成 NettyServer，其相应过程与 client 类似；

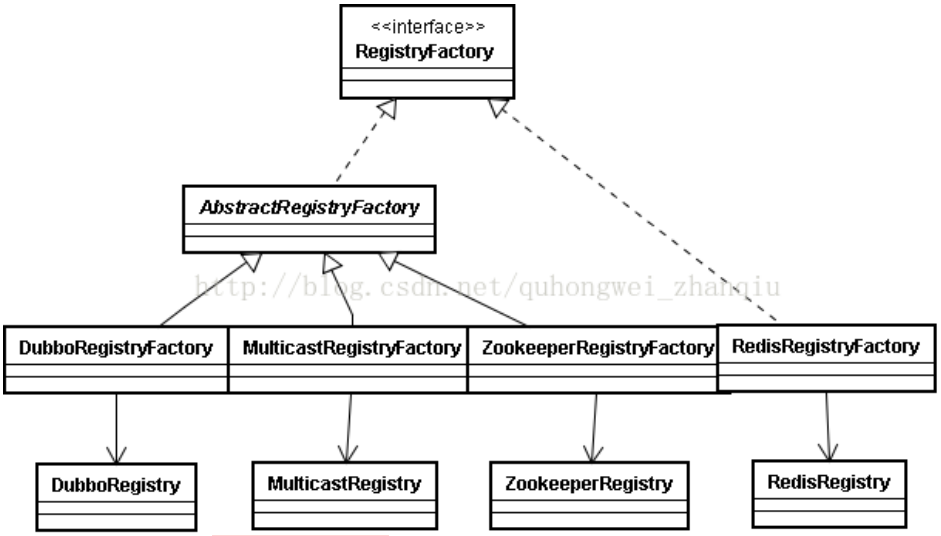
这里注意到 NettyServer 中有一个属性 channels，这里维护了所有客户端的连接；

7.2 Registry

7.2.1 RegistryFactory 和 Registry

```
@SPI("dubbo")
public interface RegistryFactory {
    /**
     * 连接注册中心.
     *
     * 连接注册中心需处理契约：<br>
     * 1. 当设置check=false时表示不检查连接，否则在连接不上时抛出异常。<br>
     * 2. 支持URL上的username:password权限认证。<br>
     * 3. 支持backup=10.20.153.10备选注册中心集群地址。<br>
     * 4. 支持file=registry.cache本地磁盘文件缓存。<br>
     * 5. 支持timeout=1000请求超时设置。<br>
     * 6. 支持session=60000会话超时或过期设置。<br>
     *
     * @param url 注册中心地址，不允许为空
     * @return 注册中心引用，总不返回空
     */
    @Adaptive({"protocol"})
    Registry getRegistry(URL url);
}
```

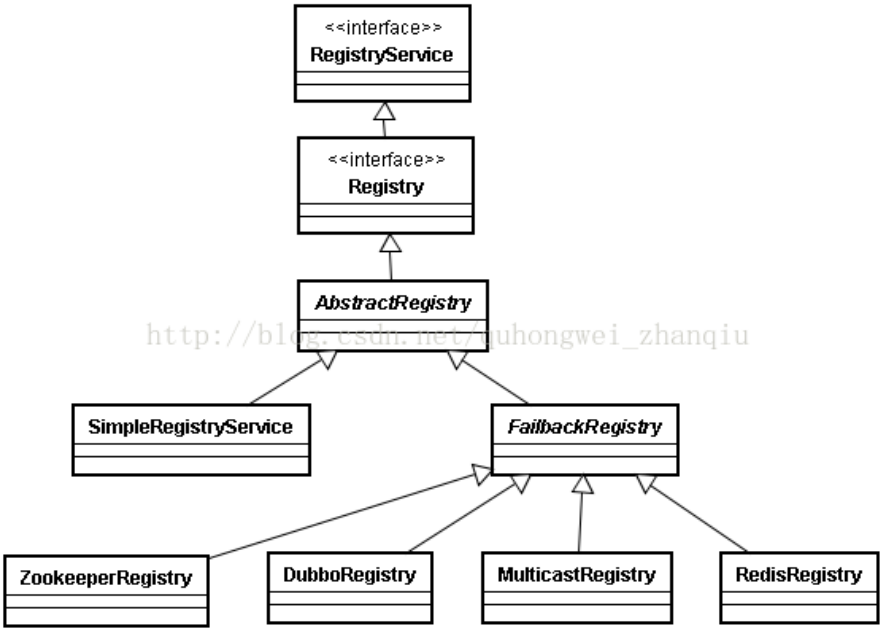
RegistryFactory 默认使用 dubbo 协议；但是实际上阿里并没有开源 dubbo 版本的注册中心，因此这个版本实际上是不可用的。



这里需要跑一下 RedisRegistry、MulticastRegistry、ZookeeperRegistry 这两个的具体过程，特别是 RedisRegistry 需要搭一个环境跑一下；

批注 [x1]: 这里需要实际搭建环境跑一下

注册中心服务类图：



服务接口定义：

```
public interface RegistryService {

    /**
     * 注册数据，比如：提供者地址，消费者地址，路由规则，覆盖规则，等数据。
     *
     * 注册需处理契约：<br>
     * 1. 当URL设置了check=false时，注册失败后不报错，在后台定时重试，否则抛出异
```

```
常。<br>
     * 2．当URL设置了dynamic=false参数，则需持久存储，否则，当注册者出现断电等情况异常退出时，需自动删除。<br>
     * 3．当URL设置了category=routers时，表示分类存储，缺省类别为providers，可按分类部分通知数据。<br>
     * 4．当注册中心重启，网络抖动，不能丢失数据，包括断线自动删除数据。<br>
     * 5．允许URI相同但参数不同的URL并存，不能覆盖。<br>
     *
     * @param url 注册信息，不允许为空，如：dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void register(URL url);

    /**
     * 取消注册.
     *
     * 取消注册需处理契约：<br>
     * 1．如果是dynamic=false的持久存储数据，找不到注册数据，则抛IllegalStateException，否则忽略。<br>
     * 2．按全URL匹配取消注册。<br>
     *
     * @param url 注册信息，不允许为空，如：dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void unregister(URL url);

    /**
     * 订阅符合条件的已注册数据，当有注册数据变更时自动推送.
     *
     * 订阅需处理契约：<br>
     * 1．当URL设置了check=false时，订阅失败后不报错，在后台定时重试。<br>
     * 2．当URL设置了category=routers，只通知指定分类的数据，多个分类用逗号分隔，并允许星号通配，表示订阅所有分类数据。<br>
     * 3．允许以interface,group,version,classifier作为条件查询，如：interface=com.alibaba.foo.BarService&version=1.0.0<br>
     * 4．并且查询条件允许星号通配，订阅所有接口的所有分组的所有版本，或：interface=*&group=*&version=*&classifier=*<br>
     * 5．当注册中心重启，网络抖动，需自动恢复订阅请求。<br>
     * 6．允许URI相同但参数不同的URL并存，不能覆盖。<br>
     * 7．必须阻塞订阅过程，等第一次通知完后再返回。<br>
     *
     * @param url 订阅条件，不允许为空，如：
```

```
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&appl
ication=kylin
     * @param listener 变更事件监听器，不允许为空
     */
    void subscribe(URL url, NotifyListener listener);

    /**
     * 取消订阅.
     *
     * 取消订阅需处理契约：<br>
     * 1. 如果没有订阅，直接忽略。<br>
     * 2. 按全URL匹配取消订阅。<br>
     *
     * @param url 订阅条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&appl
ication=kylin
     * @param listener 变更事件监听器，不允许为空
     */
    void unsubscribe(URL url, NotifyListener listener);

    /**
     * 查询符合条件的已注册数据，与订阅的推模式相对应，这里为拉模式，只返回一次结果。
     *
     * @see com.alibaba.dubbo.registry.NotifyListener#notify(List)
     * @param url 查询条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&appl
ication=kylin
     * @return 已注册信息列表，可能为空，含义同{@link
com.alibaba.dubbo.registry.NotifyListener#notify(List<URL>)}的参数。
     */
    List<URL> lookup(URL url);

}
```

7.2.2 DubboRegistryFactory 创建注册中心过程

1. 根据传入 registryUrl 重新构建

   移除 EXPORT_KEY REFER_KEY

   添加订阅回调参数

   dubbo://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-prov
ider&callbacks=10000&connect.timeout=10000&dubbo=2.5.4-SNAPSHOT&interface=com.alibab
a.dubbo.registry.RegistryService&lazy=true&methods=register,subscribe,unregister,unsubscribe,l
ookup&owner=william&pid=8492&reconnect=false&sticky=true&subscribe.1.callback=true&tim

eout=10000&timestamp=1415783872554&unsubscribe.1.callback=false

2. 根据 url 注册服务接口构建注册目录对象 RegistryDircectory,实现了 NotiyfLisener,这里 NotiyfLisener 实现主要是根据 urls 去 refer 引用远程服务 RegistryService 得到对应的 Invoker，当 urls 变化时重新 refer；目录服务可以列出所有可以执行的 Invoker

3. 利用 cluster 的 join 方法，将 Dirctory 的多个 Invoker 对象伪装成一个 Invoker 对象， 这里默认集群策略得到 FailoverClusterInvoker

4. FailoverClusterInvoker 利用 ProxyFactory 获取到 RegistryService 服务的代理对象

5. 由 RegistryService 服务的代理对象和 FailoverClusterInvoker 构建 dubbo 协议的注册中心注册器 DubboRegistry

6. RegistryDircectory 设置注册器 DubboRegistry， 设置 dubbo 的协议

7. 调用 RegistryDircectory 的 notify(urls)方法

主要是根据 registryUrls， 引用各个注册中心的 RegistryService 服务实现,将引用的服务按 key=menthodName/value=invoker 缓存起来, 目录服务 Directory.list(Invocation)会列出所调用方法的所有 Invoker , 一个 Invoker 代表对一个注册中心的调用实体。

8. 订阅注册中心服务, 服务的提供者调注册中心的服务 RegistryService 属于消费方， 所以订阅服务的 url 的协议是 consumer

consumer://192.168.0.102/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&callbacks=10000&connect.timeout=10000&dubbo=2.5.4-SNAPSHOT&interface=com.alibaba.dubbo.registry.RegistryService&lazy=true&methods=register,subscribe,unregister,unsubscribe,lookup&owner=william&pid=6960&reconnect=false&sticky=true&subscribe.1.callback=true&timeout=10000&timestamp=1415800789364&unsubscribe.1.callback=false

订阅的目的在于在注册中心的数据发送变化的时候反向推送给订阅方

directory.subscribe(url)最终调用注册中心的 RegsryService 远程服务， 它是一个普通的 dubbo 远程调用。要说跟绝大多数 dubbo 远程调用的区别：url 的参数 subscribe.1.callback=true 它的意思是 RegistryService 的 subscribe 方法的第二个参数 NotifyListener 暴露为回调服务；url 的参数 unsubscribe.1.callback=false 的意思是 RegistryService 的 unsubscribe 方法的第二个参数 NotifyListener 暴露的回调服务销毁。

这里 dubbo 协议的注册中心调注册中心的服务采用的默认集群调用策略是 FailOver,选择一台注册中心，只有当失败的时候才重试其他服务器，注册中心实现也比较简单不具备集群功能， 如果想要初步的集群功能可以选用 BroadcastCluster 它至少向每个注册中心遍历调用注册一遍；

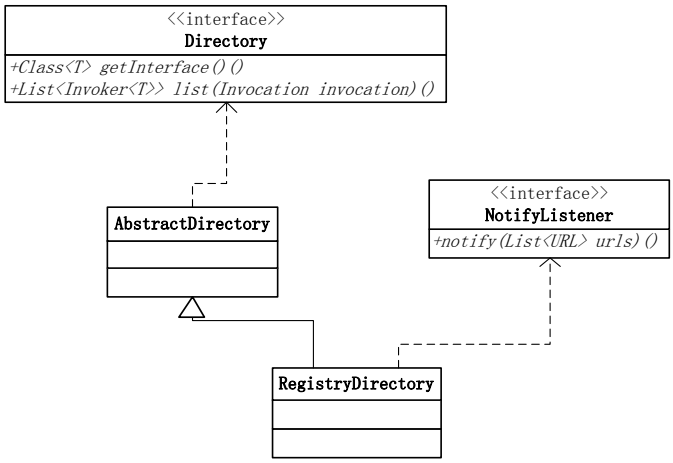
以上过程，主要涉及如下函数：

```
public Registry createRegistry(URL url) {
    url = getRegistryURL(url);
    List<URL> urls = new ArrayList<URL>();
    urls.add(url.removeParameter(Constants.BACKUP_KEY));
    String backup = url.getParameter(Constants.BACKUP_KEY);
    if (backup != null && backup.length() > 0) {
        String[] addresses =
Constants.COMMA_SPLIT_PATTERN.split(backup);
        for (String address : addresses) {
            urls.add(url.setAddress(address));
        }
    }
```

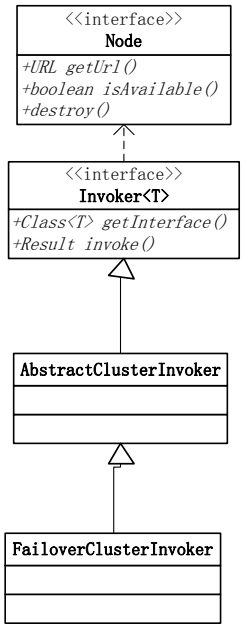
```
        RegistryDirectory<RegistryService> directory = new
RegistryDirectory<RegistryService>(RegistryService.class,
url.addParameter(Constants.INTERFACE_KEY,
RegistryService.class.getName()).addParameterAndEncoded(Constants.REF
ER_KEY, url.toParameterString()));
        Invoker<RegistryService> registryInvoker =
cluster.join(directory);
        RegistryService registryService =
proxyFactory.getProxy(registryInvoker);
        DubboRegistry registry = new DubboRegistry(registryInvoker,
registryService);
        directory.setRegistry(registry);
        directory.setProtocol(protocol);
        directory.notify(urls);
        directory.subscribe(new URL(Constants.CONSUMER_PROTOCOL,
NetUtils.getLocalHost(), 0, RegistryService.class.getName(),
url.getParameters()));
        return registry;
        }
```

这里涉及的内容比较多，包括 RegistryDirectory、cluster 等；

先来关注 RegistryDirectory：



针对 Cluster



默认使用 FailoverCluster：

```
/**
 * 失败转移，当出现失败，重试其它服务器，通常用于读操作，但重试会带来更长延迟。
 *
 * <a href="http://en.wikipedia.org/wiki/Failover">Failover</a>
 *
 * @author william.liangf
 */
public class FailoverCluster implements Cluster {
    public final static String NAME = "failover";
    public <T> Invoker<T> join(Directory<T> directory) throws
RpcException {
        return new FailoverClusterInvoker<T>(directory);
    }
}
```

对于 FailoverClusterInvoker，其最重要的是 doInvoke：

```
   @SuppressWarnings({ "unchecked", "rawtypes" })
    public Result doInvoke(Invocation invocation, final List<Invoker<T>>
invokers, LoadBalance loadbalance) throws RpcException {
    List<Invoker<T>> copyinvokers = invokers;
    checkInvokers(copyinvokers, invocation);
        int len = getUrl().getMethodParameter(invocation.getMethodName(),
Constants.RETRIES_KEY, Constants.DEFAULT_RETRIES) + 1;
        if (len <= 0) {
            len = 1;
        }
```

```
        // retry loop.
        RpcException le = null; // last exception.
        List<Invoker<T>> invoked = new
ArrayList<Invoker<T>>(copyinvokers.size()); // invoked invokers.
        Set<String> providers = new HashSet<String>(len);
        for (int i = 0; i < len; i++) {
        //重试时，进行重新选择，避免重试时invoker列表已发生变化.
        //注意：如果列表发生了变化，那么invoked判断会失效，因为invoker示例已经改
变
        if (i > 0) {
            checkWheatherDestoried();
            copyinvokers = list(invocation);
            //重新检查一下
            checkInvokers(copyinvokers, invocation);
        }
            Invoker<T> invoker = select(loadbalance, invocation,
copyinvokers, invoked);
            invoked.add(invoker);
            RpcContext.getContext().setInvokers((List)invoked);
            try {
                Result result = invoker.invoke(invocation);
                if (le != null && logger.isWarnEnabled()) {
                    logger.warn("Although retry the method " +
invocation.getMethodName()
                            + " in the service " + getInterface().getName()
                            + " was successful by the provider " +
invoker.getUrl().getAddress()
                            + ", but there have been failed providers " +
providers
                            + " (" + providers.size() + "/" +
copyinvokers.size()
                            + ") from the registry " +
directory.getUrl().getAddress()
                            + " on the consumer " + NetUtils.getLocalHost()
                            + " using the dubbo version " +
Version.getVersion() + ". Last error is: "
                            + le.getMessage(), le);
                }
                return result;
            } catch (RpcException e) {
                if (e.isBiz()) { // biz exception.
                    throw e;
                }
                le = e;
```

```
            } catch (Throwable e) {
                le = new RpcException(e.getMessage(), e);
            } finally {
                providers.add(invoker.getUrl().getAddress());
            }
        }
        throw new RpcException(le != null ? le.getCode() : 0, "Failed to
invoke the method "
                + invocation.getMethodName() + " in the service " +
getInterface().getName()
                + ". Tried " + len + " times of the providers " + providers
                + " (" + providers.size() + "/" + copyinvokers.size()
                + ") from the registry " + directory.getUrl().getAddress()
                + " on the consumer " + NetUtils.getLocalHost() + " using
the dubbo version "
                + Version.getVersion() + ". Last error is: "
                + (le != null ? le.getMessage() : ""), le != null &&
le.getCause() != null ? le.getCause() : le);
    }
```

这里我们注意到在 RegistryDirectory 中有一个 List<Invoker<T>> doList(Invocation invocation)函数，该函数主要是根据 url 获得所有的 invoker 列表；

我们注意到 RegistryDirectory 的 subscribe 函数内容如下：

```
    public void subscribe(URL url) {
        setConsumerUrl(url);
        registry.subscribe(url, this);
    }
```

此时我们看到 RegistryDirectory 实际上是继承了 NotifyListener 接口;因此注意不的目的是注册消费者（此时应理解为注册服务的消费者），若注册服务 url 发生改变，通知消费者。

7.2.3 注册中心启动

基于 dubbo 协议开源只是给出了默认一个注册中心实现 SimpleRegistryService，它只是一个简单实现，不支持集群，就是利用 Map<String/*ip:port*/, Map<String/*service*/, URL> 来存储服务地址， 具体不在啰嗦了，请读者翻看源代码，可作为自定义注册中的参考。

SimpleRegistryService 本身也是作为一个 dubbo 服务暴露。

```
<dubbo:protocolport="9090" />

<dubbo:service interface="com.alibaba.dubbo.registry.RegistryService"ref="registryService" registry="N/A"
ondisconnect="disconnect"callbacks="1000">
```

```
    <dubbo:methodname="subscribe"><dubbo:argument index="1" callback="true"/></dubbo:method>

    <dubbo:methodname="unsubscribe"><dubbo:argument index="1" callback="false"/></dubbo:method>

</dubbo:service>

<bean id="registryService"class="com.alibaba.dubbo.registry.simple.SimpleRegistryService" />
```

上面是暴露注册中心的 dubbo 服务配置，

定义了注册中心服务的端口号

发布 RegistryService 服务， registry 属性是”N/A”代表不能获取注册中心，注册中心服务的发布也是一个普通的 dubbo 服务的发布，如果没有配置这个属性它也会寻找注册中心，去通过注册中心发布，因为自己本身就是注册中心，直接对外发布服务，外部通过 ip：port 直接使用。

服务发布定义了回调接口， 这里定义了 subscribe 的第二个入参类暴露的回调服务供注册中心回调，用来当注册的服务状态变更时反向推送到客户端。

Dubbo 协议的注册中心的暴露以及调用过程跟普通的 dubbo 服务的其实是一样的，可能跟绝大多数服务的不同的是在 SimpleRegistryService 在被接收订阅请求 subscribe 的时候，同时会 refer 引用调用方暴露的 NotifyListener 服务，当有注册数据变更时自动推送；

实际上这里可以认为是 consumer 也在背后 export 了一个 notify 服务；而注册中心 refer 了一个 notify 服务，这样当注册中心发生变化时，调用 notify 服务去通知 consumer；同理，provider 也默认 export 了一个 notify 服务，注册中心数据变化时，调用该 notify 服务通州 provider，实际上 provider 做的事情就是重新 export 服务；

7.2.4 生产者发布服务

7.2.4.1 Export 发布服务流程

Dubbo 协议向注册中心发布服务：当服务提供方，向 dubbo 协议的注册中心发布服务的时候，是如何获取，创建注册中心的，如何注册以及订阅服务的，下面我们来分析其流程。

看如下配置发布服务：

```
<dubbo:registry protocol=”dubbo” address="127.0.0.1:9090" />
<beanid="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl"/>
<dubbo:serviceinterface="com.alibaba.dubbo.demo.DemoService" ref="demoService"/>
```

1. 指定了哪种的注册中心，是基于 dubbo 协议的，指定了注册中心的地址以及端口号
2. 发布 DemoService 服务，服务的实现为 DemoServiceImpl；

每个<dubbo:service/>在 spring 内部都会生成一个 ServiceBean 实例，ServiceBean 的实例化过程中调用 export 方法来暴露服务；

1. 通过 loadRegistries 获取注册中心 registryUrls

registry://127.0.0.1:9090/com.alibaba.dubbo.registry.RegistryService?application=demo-pro

vider&dubbo=2.5.4-SNAPSHOT&owner=william&pid=7084&registry=dubbo&timestamp=1415711791506

用统一数据模型 URL 表示：

protocol=registry 表示一个注册中心 url

注册中心地址 127.0.0.1:9090

调用注册中心的服务 RegistryService

注册中心协议是 registry=dubbo

。。。。。。

2. 构建发布服务的 URL

dubbo://192.168.0.102:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&owner=william&pid=7084&side=provider&timestamp=1415712331601

发布协议 protocol =dubbo

服务提供者的地址为 192.168.0.102:20880

发布的服务为 com.alibaba.dubbo.demo.DemoService

。。。。。。

3. 遍历 registryUrls 向注册中心注册服务

给每个 registryUrl 添加属性 key 为 export，value 为上面的发布服务 url 得到如下 registryUrl

registry://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.5.4-SNAPSHOT&export=dubbo%3A%2F%2F192.168.0.102%3A20880%2Fcom.alibaba.dubbo.demo.DemoService%3Fanyhost%3Dtrue%26application%3Ddemo-provider%26dubbo%3D2.5.4-SNAPSHOT%26generic%3Dfalse%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26loadbalance%3Droundrobin%26methods%3DsayHello%26owner%3Dwilliam%26pid%3D7084%26side%3Dprovider%26timestamp%3D1415712331601&owner=william&pid=7084&registry=dubbo&timestamp=1415711791506

4. 由发布的服务实例，服务接口以及 registryUrl 为参数，通过代理工厂 proxyFactory 获取 Invoker 对象，Invoker 对象是 dubbo 的核心模型，其他对象都向它靠拢或者转换成它。

5. 通过 Protocol 对象暴露服务 protocol.export(invoker)

通过 DubboProtocol 暴露服务的监听(不是此节内容)

通过 RegistryProtocol 将服务地址发布到注册中心，并订阅此服务；

以上逻辑，在 ServiceConfig 类中的 doExportUrlsFor1Protocol 实现：

```
//配置为none不暴露
        if (! Constants.SCOPE_NONE.toString().equalsIgnoreCase(scope)) {

            //配置不是remote的情况下做本地暴露 (配置为remote，则表示只暴露远程服务)
            if
```

```
(!Constants.SCOPE_REMOTE.toString().equalsIgnoreCase(scope)) {
                exportLocal(url);
            }
            //如果配置不是local则暴露为远程服务.(配置为local，则表示只暴露远程服
务)
            if (!
Constants.SCOPE_LOCAL.toString().equalsIgnoreCase(scope) ){
                if (logger.isInfoEnabled()) {
                    logger.info("Export dubbo service " +
interfaceClass.getName() + " to url " + url);
                }
                if (registryURLs != null && registryURLs.size() > 0
                        && url.getParameter("register", true)) {
                    for (URL registryURL : registryURLs) {
                        url = url.addParameterIfAbsent("dynamic",
registryURL.getParameter("dynamic"));
                        URL monitorUrl = loadMonitor(registryURL);
                        if (monitorUrl != null) {
                            url =
url.addParameterAndEncoded(Constants.MONITOR_KEY,
monitorUrl.toFullString());
                        }
                        if (logger.isInfoEnabled()) {
                            logger.info("Register dubbo service " +
interfaceClass.getName() + " url " + url + " to registry " + registryURL);
                        }
                        Invoker<?> invoker = proxyFactory.getInvoker(ref,
(Class) interfaceClass,
registryURL.addParameterAndEncoded(Constants.EXPORT_KEY,
url.toFullString()));
                        Exporter<?> exporter = protocol.export(invoker);
                        exporters.add(exporter);
                    }
                } else {
                    Invoker<?> invoker = proxyFactory.getInvoker(ref,
(Class) interfaceClass, url);

                    Exporter<?> exporter = protocol.export(invoker);
                    exporters.add(exporter);
                }
            }
        }
            this.urls.add(url);
```

这里 exportLocal 应该只是暴露在 127.0.0.1;

7.2.4.2 RegistryProtocol.export(Invoker)暴露服务

文件/dubbo-registry-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol 中配置如下：

registry=com.alibaba.dubbo.registry.integration.RegistryProtocol

因此上一步中 url 为 registry://开头，因此使用的是 RegistryProtocol 类。

```
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
        //export invoker
        final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);
        //registry provider
        final Registry registry = getRegistry(originInvoker);
        final URL registedProviderUrl = getRegistedProviderUrl(originInvoker);
        registry.register(registedProviderUrl);
        // 订阅override数据
        // FIXME 提供者订阅时，会影响同一JVM即暴露服务，又引用同一服务的的场景，因为subscribed以服务名为缓存的key，导致订阅信息覆盖。
        final URL overrideSubscribeUrl = getSubscribedOverrideUrl(registedProviderUrl);
        final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl);
        overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);
        registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);
        //保证每次export都返回一个新的exporter实例
        return new Exporter<T>() {
            public Invoker<T> getInvoker() {
                return exporter.getInvoker();
            }
            public void unexport() {
                try {
                    exporter.unexport();
                } catch (Throwable t) {
                    logger.warn(t.getMessage(), t);
                }
                try {
                    registry.unregister(registedProviderUrl);
                } catch (Throwable t) {
```

```
                logger.warn(t.getMessage(), t);
                }
                try {
                overrideListeners.remove(overrideSubscribeUrl);
                registry.unsubscribe(overrideSubscribeUrl,
overrideSubscribeListener);
                } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
                }
            }
        };
        }
```

以上函数中，doLocalExport(originInvoker)就是调用正常的protocol的export过程，进行暴露；

这里我们针对 demo 的例子，在 debug 过程中记录如下信息：

registedProviderUrl：

dubbo://192.168.1.103:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.0.0&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&pid=6324&side=provider&timestamp=1428237661384

overrideSubscribeUrl

provider://192.168.1.103:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&category=configurators&check=false&dubbo=2.0.0&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&pid=6324&side=provider&timestamp=1428237661384

注册时应该是将 registedProviderUrl 传递到注册中心，注册中心记录相应信息；这里我们可以理解为，消费者访问注册中心时，根据消费者需要获得的服务去读取服务提供者(url)；

而订阅时，则是根据 overrideSubscribeUrl 地址和 overrideSubscribeListener 监听。overrideSubscribeListener 监听的作用是当提供者的 url 改变时，重新 export；

7.2.5 消费者引用服务

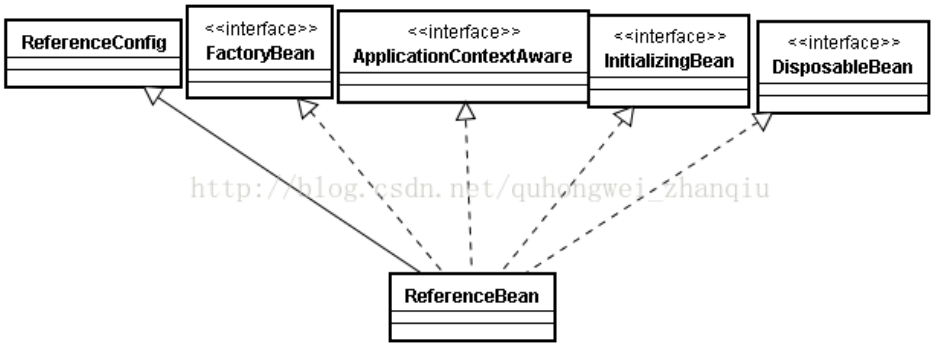
7.2.5.1 Refer 取得 invoker 的过程

```
<dubbo:registry protocol="dubbo" address="127.0.0.1:9098" />

<dubbo:reference id="demoService"interface="com.alibaba.d ubbo.demo.DemoService"/>
```

1. 指定了哪种的注册中心，是基于 dubbo 协议的，指定了注册中心的地址以及端口号；

2. 引用远程 DemoService 服务；

每个<dubbo:reference/>标签 spring 加载的时候都会生成一个 Referenc eBean。



如上图 ReferenceBean 实现了 spring 的 FactoryBean 接口， 实现了此接口的 Bean 通过 spring 的 BeanFactory.getBean(“beanName”)获取的对象不是配置的 bean 本身而是通过 FactoryBean.getObject()方法返回的对象，此接口在 spring 内部被广泛使用，用来获取代理对象等等。这里 getObjec t 方法用来生成对远程服务调用的代理

1. loadRegistries()获取配置的注册中心的 registryUrls
2. 遍历 registryUrls 集合，给 registryUrl 加上 refer key 就是要引用的远程服务
[registry://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.0&pid=2484&refer=application%3Ddemo-consumer%26dubbo%3D2.0.0%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26methods%3DsayHello%26pid%3D2484%26side%3Dconsumer%26timestamp%3D1415879965901&registry=dubbo&timestamp=1415879990670]
3. 遍历 registryUrls 集合，使用 Protocol.refer(interface,regist ryUrl)的到可执行对象 invoker
4. 如果注册中心有多个的话， 通过集群策略 Cluser.join()将多个 invoker 伪装成一个可执行 invoker， 这里默认使用 available 策略
5. 利用代理工厂生成代理对象 proxyFactory.getProxy(invoker)

这里实际上跟 export 过程类似，通过 RegistryProtocol.refer 获得 invoker；

7.2.5.2 RegistryProtocol. Refer 过程

1. 根据传入的 registryUrl 是用来选择 RegistryProcol 它的协议属性是 registry, 下面要选择使用哪种注册中心所以要根据 REGISTRY_KEY 属性重新设置 registrUrl
dubbo://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.0&pid=4524&refer=application%3Ddemo-consumer%26dubbo%3D2.0.0%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26methods%3DsayHello%26pid%3D4524%26side%3Dconsumer%26timestamp%3D1415881461048&timestamp=1415881461113

2. 根据 registrUrl 利用 RegistryFactory 获取注册器（过程跟暴露服务那边一样）， 这里是

dubbo 协议得到的是注册器是 DubboRegistry

引用并订阅注册中心服务，
3. 构建引用服务的 subscribeUrl
consumer://10.5.24.221/com.alibaba.dubbo.demo.DemoService?application=demo-consumer&category=consumers&check=false&dubbo=2.0.0&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=8536&side=consumer&timestamp=1415945205031
并通过注册器向注册中心注册消费方， 主要这里的 category 是 consume rs

4. 构建目录服务 RegistryDirectory,
构建订阅消费者订阅 url，这里主要 category=providers 去注册中心寻找注册的服务提供者
consumer://10.33.37.4/com.alibaba.dubbo.demo.DemoService?application=demo-consumer&category=providers,configurators,routers&dubbo=2.0.0&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9692&side=consumer&timestamp=1415967547508
向注册中心订阅消费方，注册中心根据消费者传入的 url 找到匹配的服务提供者 url (注意：这里服务提供者没有设置 category，注册中心对于没有设置的默认取 providers 值)
dubbo://10.33.37.4:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&owner=william&pid=9828&side=provider&timestamp=1415968955329

然后注册中心回调服务消费者暴露的回调接口来对服务提供者的服务进行引用 refer 生成对应的可执行对象 invoker。服务提供者与服务的消费建立连接,

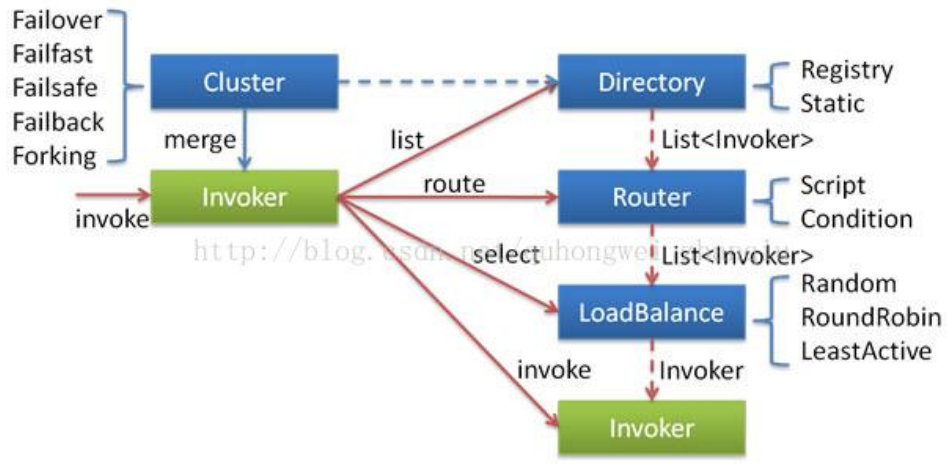
5. 通过 Cluster 合并 directory 中的 invokers， 返回可执行对象 invoker

6. ProxyFactory.getProxy(invoker) 创建代理对象返回给业务方使用

这里 dubbo 协议的注册中心调注册中心的服务采用的默认集群调用策略是 FailOver,选择一台注册中心，只有当失败的时候才重试其他服务器，注册中心实现也比较简单不具备集群功能， 如果想要初步的集群功能可以选用 BroadcastCluster 它至少向每个注册中心遍历调用注册一遍。

7.3 集群&容错

Dubbo 作为一个分布式的服务治理框架，提供了集群部署，路由，软负载均衡及容错机制；
下图描述了 dubbo 调用过程中的对于集群，负载等的调用关系。



7.3.1 Cluster

将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，包含集群的容错机制

Cluster 接口定义

```
@SPI(FailoverCluster.NAME)
public interface Cluster {
   @Adaptive
    <T> Invoker<T>join(Directory<T> directory) throws RpcException;
}
```

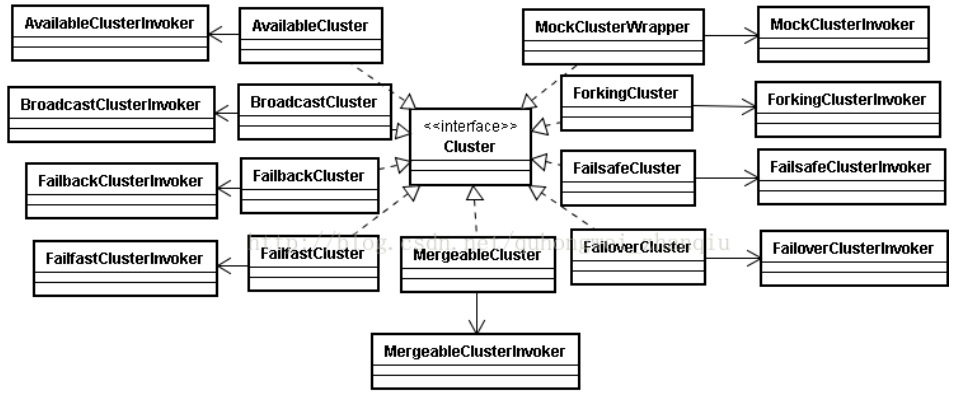
Cluster 可以看做是工厂类， 将目录 directory 下的 invoker 合并成一个统一的 Invoker，根据不同集群策略的 Cluster 创建不同的 Invoker；

我们来看下默认的失败转移，当出现失败重试其他服务的策略，这个 Cluster 实现很简单就是创建 FailoverCluseterInvoker 对象；

```
public class FailoverCluster implements Cluster {
   public final static String NAME ="failover";
   public<T> Invoker<T> join(Directory<T> directory)throws RpcException{
```

```
        return new FailoverClusterInvoker<T>(directory);
    }
}
```

下图展示了 dubbo 提供的所有集群方案



1）AvailableCluster: 获取可用的调用。遍历所有 Invokers 判断 Invoker.isAvalible,只要一个有为 true 直接调用返回，不管成不成功；

2）BroadcastCluster: 广播调用。遍历所有 Invokers, 逐个调用每个调用 catch 住异常不影响其他 invoker 调用；

3）FailbackCluster: 失败自动恢复， 对于 invoker 调用失败， 后台记录失败请求，任务定时重发, 通常用于通知；

4）FailfastCluster: 快速失败，只发起一次调用，失败立即保错，通常用于非幂等性操作；

5）FailoverCluster: 失败转移，当出现失败，重试其它服务器，通常用于读操作，但重试会带来更长延迟；

（1） 目录服务 directory.list(invocation) 列出方法的所有可调用服务
获取重试次数，默认重试两次

（2） 根据 LoadBalance 负载策略选择一个 Invoker

（3） 执行 invoker.invoke(invocation)调用

（4） 调用成功返回

调用失败小于重试次数，重新执行从 3）步骤开始执行
调用次数大于等于重试次数抛出调用失败异常

6）FailsafeCluster: 失败安全，出现异常时，直接忽略，通常用于写入审计日志等操作。

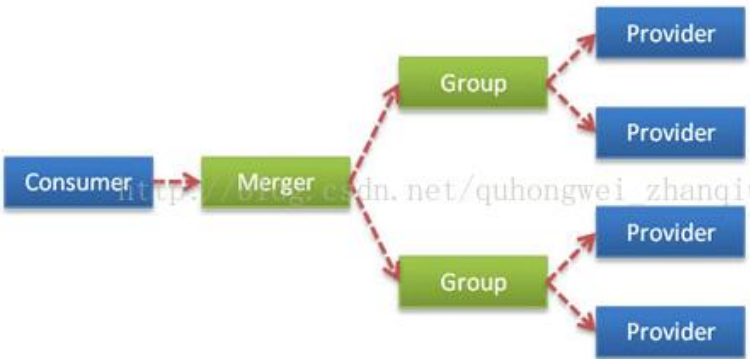
7）ForkingCluster: 并行调用，只要一个成功即返回，通常用于实时性要求较高的操作，但需要浪费更多服务资源。

8） MergeableCluster: 分组聚合， 按组合并返回结果，比如菜单服务，接口一样，但有多种实现，用 group 区分，现在消费方需从每种 group 中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

这个还蛮有意思，我们分析下是如何实现的

（1） 根据 MERGE_KEY 从 url 获取参数值

（2） 为空不需要 merge， 正常调用
（3） 按 group 分组调用，将返回接口保存到集合中
（4） 获取 MERGE_KEY 如果是默认的话，获取默认 merge 策略，主要根据返回类型判断
（5） 如果不是，获取自定义的 merge 策略
（6） Merge 策略合并调用结果返回



9）MockClusterWrapper: 具备调用 mock 功能是其他 Cluster 包装
获取 url 的 MOCK_KEY 属性
（1） 不存在直接调用其他 cluster
（2） 存在值 startsWith("force") 强制 mock 调用
（3） 存在值不是 startsWith("force") 先正常调用， 出现异常在 mock 调用

集群模式的配置
<dubbo:service cluster="failsafe" /> 服务提供方
<dubbo:reference cluster="failsafe" /> 服务消费方

7.3.2 目录服务 Directory

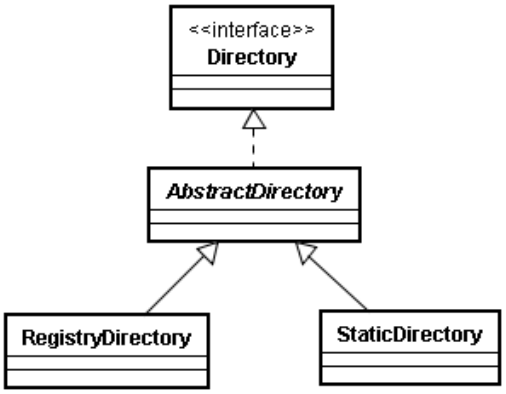
集群目录服务 Directory， 代表多个 Invoker, 可以看成 List<Invoker>,它的值可能是动态变化的比如注册中心推送变更。集群选择调用服务时通过目录服务找到所有服务；

Directory 的接口定义：

```
public interfaceDirectory<T> extends Node {

  //服务类型

Class<T>getInterface();

//列出所有服务的可执行对象

List<Invoker<T>>list(Invocation invocation) throws RpcException;
```

}

Directory 有两个具体实现



StaticDirectory: 静态目录服务， 它的所有 Invoker 通过构造函数传入， 服务消费方引用服务的时候， 服务对多注册中心的引用，将 Invokers 集合直接传入 StaticDirectory 构造器，再由 Cluster 伪装成一个 Invoker

ReferenceConfig.java | Directory.java | AbstractDirectory.java | RegistryDirectory.java | StaticDirectory.java

```
396     for (URL url : urls) {
397         invokers.add(refprotocol.refer(interfaceClass, url));
398         if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
399             registryURL = url; // 用了最后一个registry url
400         }
401     }
402     if (registryURL != null) { // 有注册中心协议的URL
403         // 对有注册中心的Cluster 只用 AvailableCluster
404         URL u = registryURL.addParameter(Constants.CLUSTER_KEY, AvailableCluster.NAME);
405         invoker = cluster.join(new StaticDirectory(u, invokers));
406     } else { // 不是注册中心的URL
407         invoker = cluster.join(new StaticDirectory(invokers));
```


StaticDirectory 的 list 方法直接返回所有 invoker 集合；

RegistryDirectory: 注册目录服务， 它的 Invoker 集合是从注册中心获取的， 它实现了 NotifyListener 接口实现了回调接口 notify(List<Url>)。

比如消费方要调用某远程服务，会向注册中心订阅这个服务的所有服务提供方，订阅时和服务提供方数据有变动时回调消费方的 NotifyListener 服务的 notify 方法 NotifyListener.notify(List<Url>) 回调接口传入所有服务的提供方的 url 地址然后将 urls 转化为 invokers，也就是 refer 应用远程服务；

RegistryDirectory.java | Directory.java | AbstractDirectory.java

```
381             }
382             keys.add(key);
383             // 缓存key为没有合并消费端参数的URL，不管消费端如何合并参数，如果服务端URL发生变化，则重新refer
384             Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference
385             Invoker<T> invoker = localUrlInvokerMap == null ? null : localUrlInvokerMap.get(key);
386             if (invoker == null) { // 缓存中没有，重新refer
387                 try {
388                     boolean enabled = true;
389                     if (url.hasParameter(Constants.DISABLED_KEY)) {
390                         enabled = ! url.getParameter(Constants.DISABLED_KEY, false);
391                     } else {
392                         enabled = url.getParameter(Constants.ENABLED_KEY, true);
393                     }
394                     if (enabled) {
395                         invoker = new InvokerDelegete<T>(protocol.refer(serviceType, url), url, providerUrl);
396                     }
397                 } catch (Throwable t) {
```

到此时引用某个远程服务的 RegistryDirectory 中有对这个远程服务调用的所有 invokers。

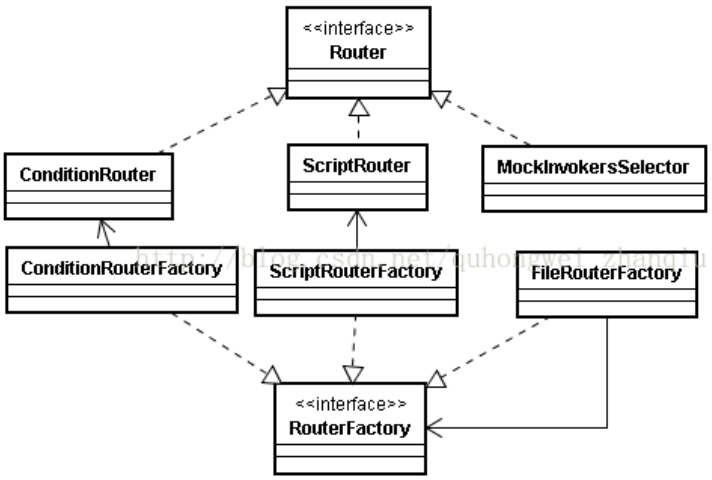
RegistryDirectory.list(invocation)就是根据服务调用方法获取所有的远程服务引用的invoker 执行对象。

7.3.3 router 路由服务

Router 服务路由， 根据路由规则从多个 Invoker 中选出一个子集 AbstractDirectory 是所有目录服务实现的上层抽象， 它在 list 列举出所有 invokers 后，会在通过 Router 服务进行路由过滤。

Router 接口定义：

```
public interface Router extendsComparable<Router> {

   URL getUrl();

<T> List<Invoker<T>> route(List<Invoker<T>>invokers, URL url, Invocation invocation)throws RpcException;

}
```



这里如果要了解具体的路由规则，需要查看阿里的原始文档
http://www.dubbo.io/User+Guide-zh.htm
在该文档的“路由规则”部分有详细介绍；
我们来分析下，路由规则在哪里用到：
在 RegistryDirectory 类中：

```
    private List<Invoker<T>> route(List<Invoker<T>> invokers, String
method) {
        Invocation invocation = new RpcInvocation(method, new Class<?>[0],
new Object[0]);
        List<Router> routers = getRouters();
        if (routers != null) {
```

```
            for (Router router : routers) {
                if (router.getUrl() != null && !
router.getUrl().getParameter(Constants.RUNTIME_KEY, true)) {
                    invokers = router.route(invokers, getConsumerUrl(),
invocation);
                }
            }
        }
        return invokers;
    }
```

而 route 函数又是在如下位置被用到：

toMethodInvokers(Map<String, Invoker<T>> invokersMap)

最终调用是在 refreshInvoker 函数，而 refreshInvoker 又是在 notify 函数中调用的。

7.3.4 负载均衡

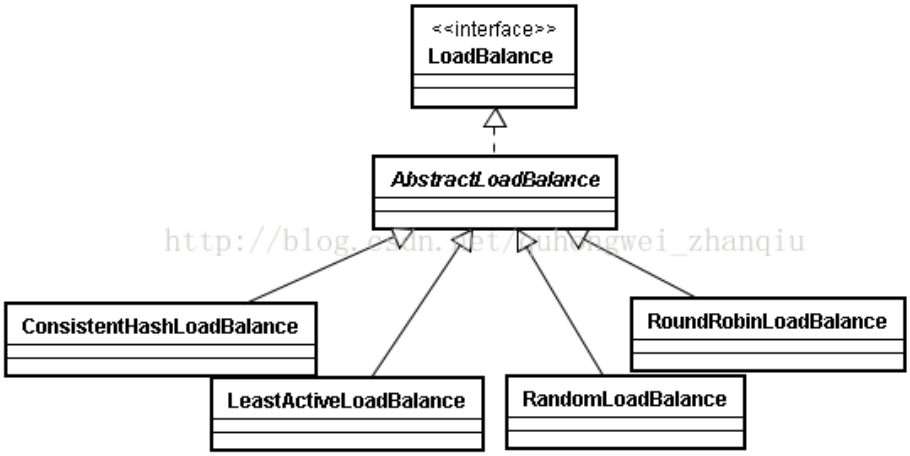
LoadBalance 负载均衡，负责从多个 Invokers 中选出具体的一个 Invoker 用于本次调用，调用过程中包含了负载均衡的算法，调用失败后需要重新选择：

```
@SPI(RandomLoadBalance.NAME)
public interface LoadBalance {
    /**
     * select one invoker in list.
     *
     * @param invokers invokers.
     * @param url refer url
     * @param invocation invocation.
     * @return selected invoker.
     */
    @Adaptive("loadbalance")
    <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation
invocation) throws RpcException;
    }
```

类注解@SPI 说明可以基于 Dubbo 的扩展机制进行自定义的负责均衡算法实现，默认是随机算法；

方法注解@Adaptive 说明能够生成设配方法；

Select 方法设配类通过 url 的参数选择具体的算法， 在从 invokers 集合中根据具体的算法选择一个 invoker；



7.3.4.1 RandomLoadBalance

1. RandomLoadBalance: 随机访问策略，按权重设置随机概率，是默认策略
1）获取所有 invokers 的个数
2）遍历所有 Invokers，获取计算每个 invokers 的权重，并把权重累计加起来
每相邻的两个 invoker 比较他们的权重是否一样，有一个不一样说明权重不均等
3）总权重大于零且权重不均等的情况下
按总权重获取随机数 offset = random.netx(totalWeight);
遍历 invokers 确定随机数 offset 落在哪个片段(invoker 上)

```
int offset = random.nextInt(totalWeight);
// 并确定随机值落在哪个片断上
for (int i = 0; i < length; i++) {
    offset -= getWeight(invokers.get(i), invocation);
    if (offset < 0) {
        return invokers.get(i);
    }
}
```

4）权重相同或者总权重为 0， 根据 invokers 个数均等选择
invokers.get(random.nextInt(length))

7.3.4.2 RoundRobinLoadBalance

RoundRobinLoadBalance:轮询，按公约后的权重设置轮询比率
1）获取轮询 key 服务名+方法名
获取可供调用的 invokers 个数 length

设置最大权重的默认值 maxWeight=0
设置最小权重的默认值 minWeight=Integer.MAX_VALUE
2）遍历所有 Inokers，比较出得出 maxWeight 和 minWeight
3）如果权重是不一样的
根据 key 获取自增序列
自增序列加一与最大权重取模默认得到 currentWeigth
遍历所有 invokers 筛选出大于 currentWeight 的 invokers
设置可供调用的 invokers 的个数 length
4）自增序列加一并与 length 取模，从 invokers 获取 invoker

7.3.4.3 LeastActiveLoadBalance

LeastActiveLoadBalance: 最少活跃调用数， 相同的活跃的随机选择，

活跃数是指调用前后的计数差， 使慢的提供者收到更少的请求，因为越慢的提供者前后的计数差越大。

活跃计数的功能消费者是在 ActiveLimitFilter 中设置的

```
long begin = System.currentTimeMillis();
RpcStatus.beginCount(url, methodName);
try {
    Result result = invoker.invoke(invocation);
    RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, true);
    return result;
} catch (RuntimeException t) {
    RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, false);
    throw t;
}
```


4. 最少活跃的选择过程如下：

1）获取可调用 invoker 的总个数
初始化最小活跃数，相同最小活跃的个数
相同最小活跃数的下标数组
等等

2）遍历所有 invokers， 获取每个 invoker 的获取数 active 和权重
找出最小权重的 invoker
如果有相同最小权重的 inovkers， 将下标记录到数组 leastIndexs[]数组中
累计所有的权重到 totalWeight 变量

3）如果 invokers 的权重不相等且 totalWeight 大于 0
按总权重随机 offsetWeight = random.nextInt(totalWeight)
计算随机值在哪个片段上并返回 invoker

```
int offsetWeight = random.nextInt(totalWeight);
// 并确定随机值落在哪个片断上
for (int i = 0; i < leastCount; i++) {
    int leastIndex = leastIndexs[i];
    offsetWeight -= getWeight(invokers.get(leastIndex), invocation);
    if (offsetWeight <= 0)
        return invokers.get(leastIndex);
}
```


4）如果 invokers 的权重相等或者 totalWeight 等于 0，均等随机

这里用到了 RpcStatus，由 RpcStatus 的类描述可知，该项内容是在 filter 中进行记录；

```
/**
 * URL statistics. (API, Cached, ThreadSafe)
 *
 * @see com.alibaba.dubbo.rpc.filter.ActiveLimitFilter
 * @see com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter
 * @see
com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance
 * @author william.liangf
 */
   public class RpcStatus {}
```

7.3.4.4 ConsistentHashLoadBalance

ConsistentHashLoadBalance:一致性 hash，相同参数的请求总是发到同一个提供者，当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。对于一致性哈希算法介绍网上很多，这个给出一篇 http://blog.csdn.net/sparkliang/article/details/5279393 供参考，读者请自行阅读 ConsistentashLoadBalance 中对一致性哈希算法的实现，还是比较通俗易懂的这里不再啰嗦。

7.3.5 配置规则

配置规则实际上是在生成 invoker 的过程中对 url 进行改写；

```
/**
 * Configurator. (SPI, Prototype, ThreadSafe)
 *
 * @author william.liangf
 */
public interface Configurator extends Comparable<Configurator> {

    /**
     * get the configurator url.
     *
     * @return configurator url.
     */
    URL getUrl();
    /**
     * Configure the provider url.
     * O
```

```
     * @param url - old rovider url.
     * @return new provider url.
     */
    URL configure(URL url);
}
```

目前包含两种规则：AbsentConfigurator 和 OverrideConfigurator；前者是如果缺少项，则新增；而后者是直接覆盖；

具体是在 RegistryDirectory 中的 mergeUrl 函数中用到；

另外，根据 dubbo 文档描述如下：

向注册中心写入动态配置覆盖规则：（通常由监控中心或治理中心的页面完成）

```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptive
Extension();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:218
1"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?
category=configurators&dynamic=false&application=foo&timeout=1000"));
```

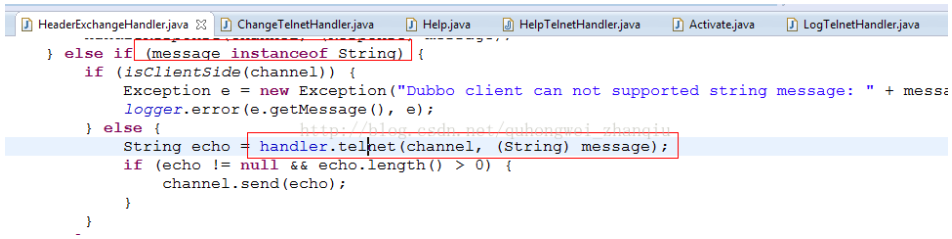
7.4 telnet

telnet 的介绍可以参看《java 网络编程 3》中有一段介绍 telnet；

我们可以理解为，telnet 命令是通过 socket 协议与服务器端通信；

Dubbo 提供了 telnet 命令去查看服务功能：

这里主要介绍一下 dubbo 实现 telnet 命令的整体实现：



当服务器端接收到的消息类型是 string 的时候回调用到 TelnetHandler 的 telent 方法中；

TelnetHanlderAdpter 类会从接收的字符串解析出命令，根据 dubbo 的 spi 扩展机制获取对应的 TelnetHandler 实现；

这里我们查看 DubboProtocol 类中，

private ExchangeHandler requestHandler = new ExchangeHandlerAdapter()

的实现，会发现：

```
public abstract class ExchangeHandlerAdapter extends
TelnetHandlerAdapter implements ExchangeHandler {
```

```
    public Object reply(ExchangeChannel channel, Object msg) throws
RemotingException {
        return null;
    }
    }
```

而 TelnetHandlerAdapter 类的内容为：

```
public class TelnetHandlerAdapter extends ChannelHandlerAdapter
implements TelnetHandler {

    private final ExtensionLoader<TelnetHandler> extensionLoader =
ExtensionLoader.getExtensionLoader(TelnetHandler.class);

    public String telnet(Channel channel, String message) throws
RemotingException {
        String prompt =
channel.getUrl().getParameterAndDecoded(Constants.PROMPT_KEY,
Constants.DEFAULT_PROMPT);
        boolean noprompt = message.contains("--no-prompt");
        message = message.replace("--no-prompt", "");
        StringBuilder buf = new StringBuilder();
        message = message.trim();
        String command;
        if (message.length() > 0) {
            int i = message.indexOf(' ');
            if (i > 0) {
                command = message.substring(0, i).trim();
                message = message.substring(i + 1).trim();
            } else {
                command = message;
                message = "";
            }
        } else {
            command = "";
        }
        if (command.length() > 0) {
            if (extensionLoader.hasExtension(command)) {
                try {
                    String result =
extensionLoader.getExtension(command).telnet(channel, message);
                    if (result == null) {
                        return null;
                    }
                    buf.append(result);
```

```
                } catch (Throwable t) {
                    buf.append(t.getMessage());
                }
            } else {
                buf.append("Unsupported command: ");
                buf.append(command);
            }
        }
        if (buf.length() > 0) {
            buf.append("\r\n");
        }
        if (prompt != null && prompt.length() > 0 && ! noprompt) {
            buf.append(prompt);
        }
        return buf.toString();
    }
}
```

这里我们可以发现

```
String result = extensionLoader.getExtension(command).telnet(channel,
message);
```

这个我们可以理解为，你在 telnet 输入的每个命令，都由一个类对象来处理；

在 com.alibaba.dubbo.remoting.telnet.TelnetHandler 多个文件中有如下配置

clear=com.alibaba.dubbo.remoting.telnet.support.command.ClearTelnetHandler

exit=com.alibaba.dubbo.remoting.telnet.support.command.ExitTelnetHandler

help=com.alibaba.dubbo.remoting.telnet.support.command.HelpTelnetHandler

。。。。。。

对于 telnent 功能的实现方式跟其他的功能类似，由于每个 TelnetHandler 实现太细了，这里对有兴趣的读者自己翻看源码；

7.5 监控

Dubbo 发布代码中，自带了一个简易的监控中心实现。对于一般的小业务这个监控中心应该能够满足需求，对于那些大业务量的大公司一般都会有自己的监控中心，更加丰富的功能如常用的报警短信通知等等。这章讲解分析使得读者能够了解一般的监控中心实现，也使得有自己接入监控中心需求的大概知道如何集成自己的监控中心实现。下面我们就以 dubbo 自带的监控中心开始讲解。

7.5.1 监控中心

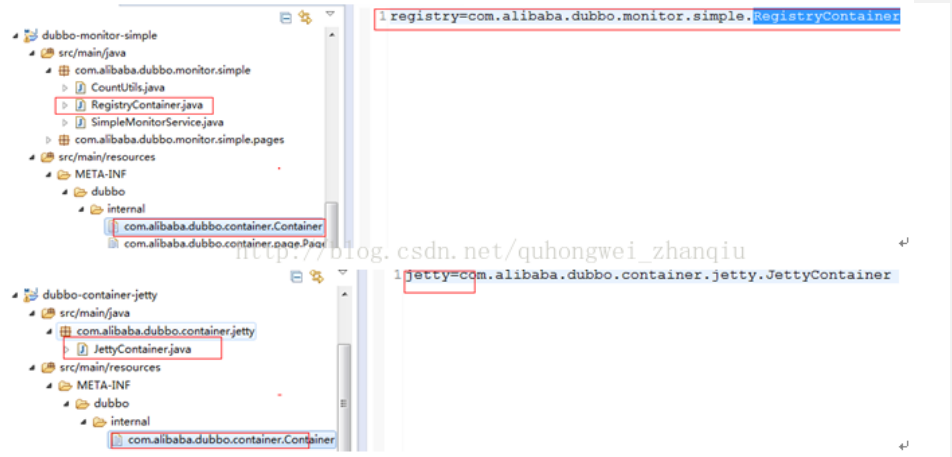
调用流程：1）监控中心暴露服务到注册中心；

2、服务提供端，根据设置，调用 DubboMonitor，monitorfilter，

1. 监控中心启动，我们先看下 dubbo 的属性文件

```
dubbo.container=log4j,spring,registry,jetty
dubbo.application.name=simple-monitor
dubbo.application.owner=
dubbo.registry.address=multicast://224.5.6.7:1234
#dubbo.registry.address=zookeeper://127.0.0.1:2181
#dubbo.registry.address=redis://127.0.0.1:6379
#dubbo.registry.address=dubbo://127.0.0.1:9090
dubbo.protocol.port=7070
dubbo.jetty.port=8080
dubbo.jetty.directory=${user.home}/monitor
dubbo.charts.directory=${dubbo.jetty.directory}/charts
dubbo.statistics.directory=${user.home}/monitor/statistics
#dubbo.log4j.file=logs/dubbo-demo-consumer.log
    #dubbo.log4j.level=WARN
```

相比于 provider、consumer，监控中心的启动注册中心多了 registry, jetty 容器启动：



它们都是基于 dubbo 的 spi 扩展机制的。

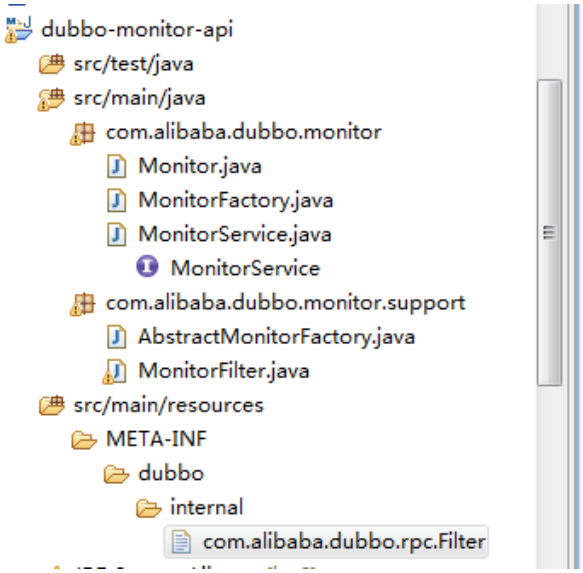
SpringContainer 容器启动就是加载 classpath*:META-INF/spring/ *.xml spring 的配置文件

```
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderCo
nfigurer">
        <property name="systemPropertiesModeName"
value="SYSTEM_PROPERTIES_MODE_OVERRIDE" />
        <property name="location" value="classpath:dubbo.properties" />
    </bean>

    <bean id="monitorService"
class="com.alibaba.dubbo.monitor.simple.SimpleMonitorService">
        <property name="statisticsDirectory"
value="${dubbo.statistics.directory}" />
```

```
        <property name="chartsDirectory"
value="${dubbo.charts.directory}" />
    </bean>

    <dubbo:application name="${dubbo.application.name}"
owner="${dubbo.application.owner}" />

    <dubbo:registry address="${dubbo.registry.address}" />

    <dubbo:protocol name="dubbo" port="${dubbo.protocol.port}" />

    <dubbo:service
interface="com.alibaba.dubbo.monitor.MonitorService"
ref="monitorService" delay="-1" />

        <dubbo:reference                              id="registryService"
interface="com.alibaba.dubbo.registry.RegistryService" />
```

7.5.2 SimpleMonitorService

7.5.2.1 Monitor 基础类



监控服务的接口定义

```
public interface MonitorService {
```

```
    String APPLICATION = "application";
    String INTERFACE = "interface";
    String METHOD = "method";
    String GROUP = "group";
    String VERSION = "version";
    String CONSUMER = "consumer";
    String PROVIDER = "provider";
    String TIMESTAMP = "timestamp";
    String SUCCESS = "success";
    String FAILURE = "failure";
    String INPUT = Constants.INPUT_KEY;
    String OUTPUT = Constants.OUTPUT_KEY;
    String ELAPSED = "elapsed";
    String CONCURRENT = "concurrent";
    String MAX_INPUT = "max.input";
    String MAX_OUTPUT = "max.output";
    String MAX_ELAPSED = "max.elapsed";
    String MAX_CONCURRENT = "max.concurrent";
    /**
     * 监控数据采集.
     * 1. 支持调用次数统计:
count://host/interface?application=foo&method=foo&provider=10.20.153.
11:20880&success=12&failure=2&elapsed=135423423
     * 1.1 host,application,interface,group,version,method 记录监控来源主
机，应用，接口，方法信息。
     * 1.2 如果是消费者发送的数据，加上provider地址参数，反之，加上来源consumer
地址参数。
     * 1.3 success,faulure,elapsed 记录距上次采集，调用的成功次数，失败次数，
成功调用总耗时，平均时间将用总耗时除以成功次数。
     *
     * @param statistics
     */
    void collect(URL statistics);
    /**
     * 监控数据查询.
     * 1. 支持按天查询:
count://host/interface?application=foo&method=foo&side=provider&view=
chart&date=2012-07-03
     * 1.1 host,application,interface,group,version,method 查询主机，应用，
接口，方法的匹配条件，缺失的条件的表示全部，host用0.0.0.0表示全部。
     * 1.2 side=consumer,provider 查询由调用的哪一端采集的数据，缺省为都查询。
     * 1.3 缺省为view=summary，返回全天汇总信息，支持view=chart表示返回全天趋
势图表图片的URL地址，可以进接嵌入其它系统的页面上展示。
     * 1.4 date=2012-07-03 指定查询数据的日期，缺省为当天。
```

```
     *
     * @param query
     * @return statistics
     */
    List<URL> lookup(URL query);
}
```

注: lookup 方面可能在开源过程中依赖了阿里的什么系统，并没有具体的实现，如果使用着需要此功能则需要根据接口定义自己实现；

MonitorFactory

```
@SPI("dubbo")
public interface MonitorFactory {
    /**
     * Create monitor.
     *
     * @param url
     * @return monitor
     */
    @Adaptive("protocol")
    Monitor getMonitor(URL url);
}
```

7.5.2.2 SimpleMonitorService

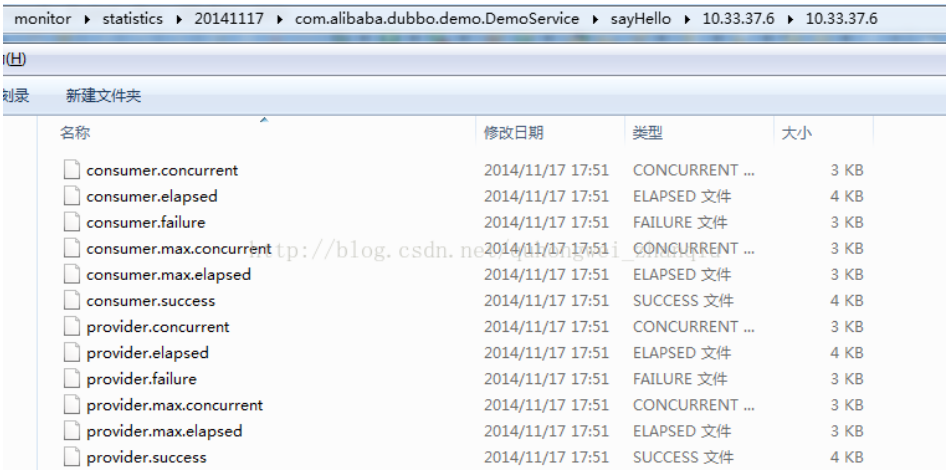
MonitorService 的 dubbo 默认实现 SimpleMonitorService

Collect 方法被远程调用后将数据 url(传过来的 url 包含监控需要的数据)保存到一个阻塞队列中 BlockingQueue<URL>；

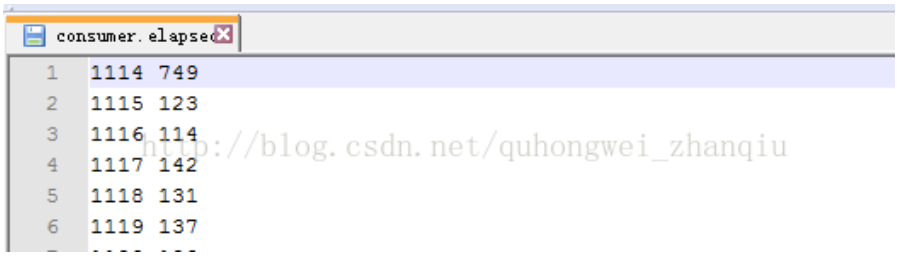
启动定时任务将统计日志记录到本地,

```
String filename =${user.home}/monitor/statistics
                + "/" + day
                + "/" +statistics.getServiceInterface()
                + "/" +statistics.getParameter(METHOD)
                + "/" + consumer
                + "/" + provider
                + "/" + type + "." + key
```

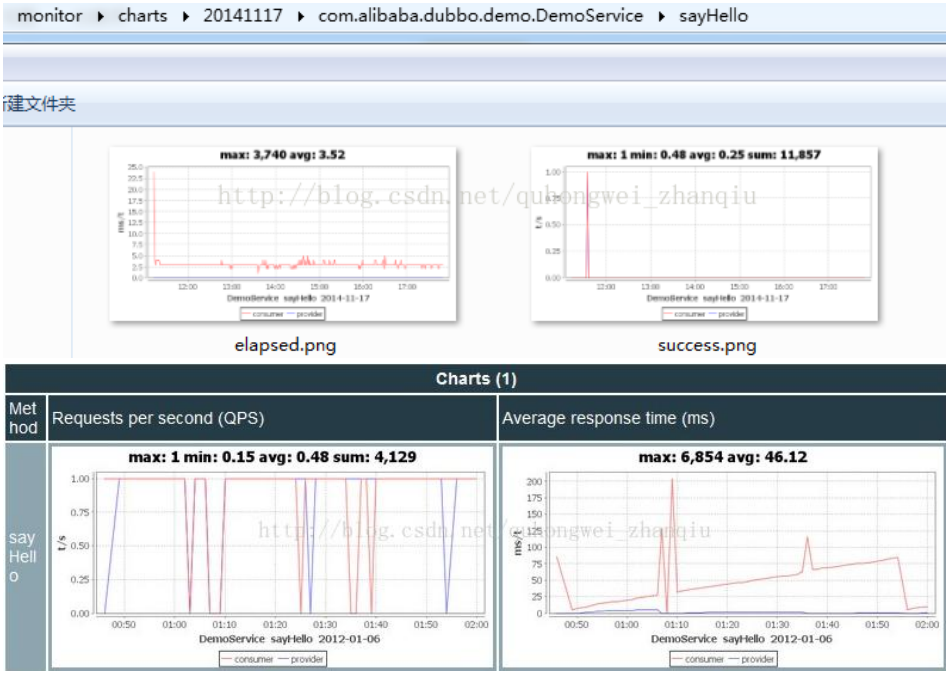
这是文件在本地存储的格式



文件内容如图保存时间方法消费耗时



3. 起定时任务利用 JFreeeChart 绘制图表,保存路径
${user.home}\monitor\charts\date\interfaceName\methodName



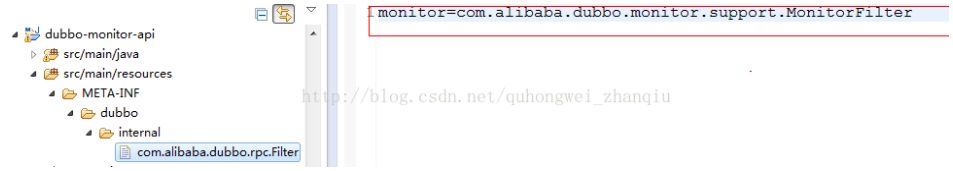
7.5.2.3 产生监控数据

注册中心暴露了 MonitorService 服务，它是被谁调用的呢，监控中心的数据是从哪里来呢，下面我们看下服务提供方与服务的消费方式如何介入监控中心的。

在服务的提供方跟消费方的 dubbo 配置加入如下配置

通过注册中心<dubbo:monitor protocol="registry" />

或者直连 <dubbo:monitor address="127.0.0.1:7070" />



在构建服务的调用链的时候有如上基于监控的扩展，下面我们就来看下这个类

```
@Activate(group = {Constants.PROVIDER, Constants.CONSUMER})
//此过滤器在服务的提供方，服务的消费方应用中被激活，也就是起作用
public class MonitorFilter implements Filter {
    private MonitorFactory monitorFactory;
    public Result invoke(Invoker<?>invoker, Invocation invocation) throws RpcException {
    if(invoker.getUrl().hasParameter(Constants.MONITOR_KEY)) {
        //有注监控中心处理
1.  获取 invoker 的调用上下文
2.  记录起始时间戳
3.  并发计数加一
try {
4.  调用调用链的下一步
5.  采集调用信息
} finally {
6.  并发计数减一
}
    } else {
        //没有配置监控中心，直接往下调用
        return invoker.inovke(invocation);
    }
}
```

上面第 5 点信息采集

1. 计算调用耗时
2. 获取并发数
3. 获取服务名称
4. 获取方法名

5. 判断是服务消费方监控还是服务提供方监控

6. 由工厂类 monitorFactory.getMonitor(监控 url)，获取 DubboMonitor 对象，构建调用监控中心服务的的 Url， url 中包括了监控中心所需的监控信息;

```
monitor.collect(new URL(Constants.COUNT_PROTOCOL,
                        NetUtils.getLocalHost(), localPort,
                        service + "/" + method,
                        MonitorService.APPLICATION, application,
                        MonitorService.INTERFACE, service,
                        MonitorService.METHOD, method,
                        remoteKey, remoteValue,
            error ? MonitorService.FAILURE : MonitorService.SUCCESS, "1",
            MonitorService.ELAPSED, String.valueOf(elapsed),
            MonitorService.CONCURRENT, String.valueOf(concurrent),
                        Constants.INPUT_KEY, input,
                        Constants.OUTPUT_KEY, output));
```

DubboMonitor 是调用监控中心的服务的封装，之所以没有直接调监控中心而是通过 DubboMonitor 调用，是因为监控是附加功能，不应该影响主链路更不应该损害主链路的新能，DubboMonitor 采集到数据后通过任务定时调用监控中心服务将数据提交到监控中心。

这里可以理解为实际上服务提供端和消费端在使用 dubbo 协议监控中心时，是调用 DubboMonitor。这里的初始化逻辑可以理解为：先找到 monitorfilter，然后再初始化该 filter 的过程中，初始化 monitorFactory，而最后是根据 monitorFactory 获得 monitor；

7.5.2.4 RegistryContainer

监控中心 refer 引用了注册中心暴露的 RegistryService 服务，主要是被下面的 RegistryContainer 使用的。

RegistryContainer 主要是到注册中心收集服务，分组，版本信息，并注册回调当注册中心数据发生变化的时候更新到监控中心

下面看下 RegistryContainer 的 start 方法流程：

1. 通过 SpringContainer 获取前面初始化的 RegistryService，得到其实是对注册中心的一个远程代理服务

2. 构建订阅注册中心数据的 URL，看可以看出下面的 url 是订阅服务提供者和服务消费者的所有服务

```
subscribeUrl = newURL(Constants.ADMIN_PROTOCOL, NetUtils.getLocalHost(), 0,"",
        Constants.INTERFACE_KEY,Constants.ANY_VALUE,//所有服务
        Constants.GROUP_KEY,Constants.ANY_VALUE,//所有分组
        Constants.VERSION_KEY, Constants.ANY_VALUE,//所有版本
        Constants.CLASSIFIER_KEY,Constants.ANY_VALUE,//所有分类
Constants.CATEGORY_KEY,Constants.PROVIDERS_CATEGORY + "," + Constants.CONSUMERS_CATEGORY,//服务的提供者和服务的消费者
        Constants.CHECK_KEY,String.valueOf(false));//不检查
```

3. 调注册中心服务 registry.subscirbe(subscribeUrl,listener)订阅所有数据, NotifyListener 在监控中心暴露为回调服务，由注册中心回调

回调接口 NotifyListener 实现的功能主要是按服务提供者和服务的消费者分类，收集服务名称，服务的 url，服务提供方或者消费方的系统相关信息。 同时提供了一系列方法供注册中心调用查询。

在各种 PageHandler 中，你会看到 RegistryContainer 的使用，主要是为了获得服务提供者和服务消费者。

7.5.2.5 JettyContainer

监控中心将采集到的信息通过内置 jetty 来展现给用户，这里为了不依赖与 jsp, velocity，freemarker 等一些编写 web 应用的技术，采用在 servlet 中将 html，css，js 打印出来；

JettyContainer 的 start 方法启动了内置的 jettyweb 容器；

将监控中心访问的本地文件目录设置到 ResourceFilter 中，并设置这个 filter 的访问映射到 jetty 中 ，ResourceFilter 主要是读取本地保存的 JFreeChart 绘制的图片到浏览器中去。

将监控中心的前置控制器 PageServlet，以及这个 servlet 的访问映射配置到 jetty 中。之所以叫 PageServet 为前置控制器，就像其他的 mvc 框架一样用来分发具体的业务类

PageServet 的 init 初始化方法在 web 容器启动的时候加载所有的页面处理器 PageHandler, 用来根据不同的请求生成不同的页面，前面说过这里页面 html 都是通过 java 代码打印出来的。

PageServet 的 init 方法加载所有 PageHandler 时会判断 PageHandler 上是否有@Menu 注解，将有注解的 PageHandler 加入集合，以被 HomePageHandl er 用来生成主页以及各个页面的 uri；

PageServet 的 doGet, doPost 接收浏览器请求，请求以 xx.hml 形式，xx 就是 PageHandler 扩展的 key，找到对应的 PageHandler 绘制对应的页面返回给浏览器。

```
@Menu(name = "Home",desc = "Home page.", order = Integer.MIN_VALUE)
//有注解 name 跟 desc 属性都是在页面中展示给用户看的
public class HomePageHandlerimplements PageHandler {
    public Page handle(URL url) {
        List<List<String>> rows =new ArrayList<List<String>>();
        for (PageHandler handler :PageServlet.getInstance().getMenus()) {
            String uri =ExtensionLoader.getExtensionLoader(PageHandler.class).getExtensionName(handler); //这个 uri 其实就是 PageHandler 扩展配置的 key，页面中用它来请求选择具体的 handler 绘制     //出具体的 page
            Menu menu =handler.getClass().getAnnotation(Menu.class);
            List<String> row = newArrayList<String>();
            row.add("<ahref=\"" + uri + ".html\">" + menu.name() +"</a>");
            row.add(menu.desc());
            rows.add(row);
        }
        return new Page("Home","Menus",   new String[]{"Menu Name", "Menu Desc"}, rows);  //一个 Page 实体就是一个页面，这里包含所有主要 HomePage 的页面内容
    }
```

}

PageHandler 的在 com.alibaba.dubbo.container.page.PageHandler 文件中的扩展配置

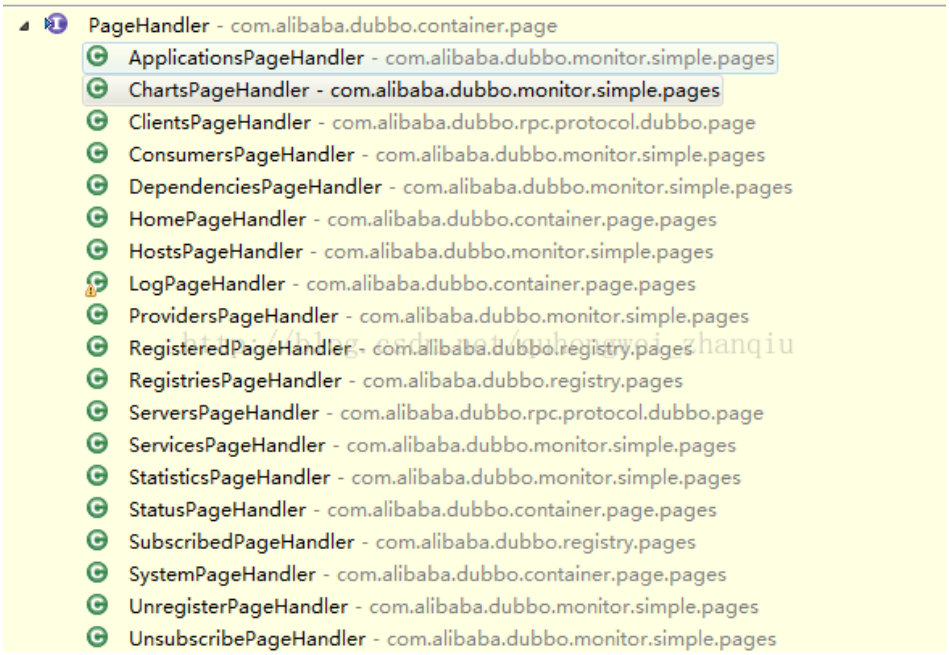
index=com.alibaba.dubbo.container.page.pages.HomePageHandler

providers=com.alibaba.dubbo.monitor.simple.pages.ProvidersPageHandler

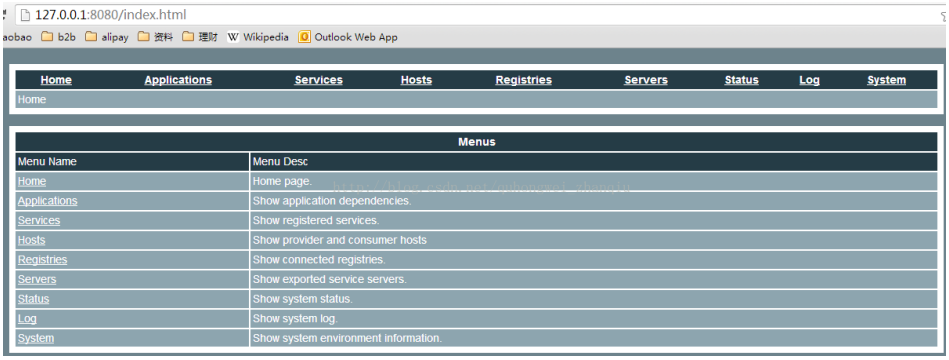
consumers=com.alibaba.dubbo.monitor.simple.pages.ConsumersPageHandler

。。。。

下面截图看下 dubbo 大概提供了哪些扩展



下面截几张图看看监控中心的页面。



127.0.0.1:8080/services.html

obao b2b alipay 资料 理财 Wikipedia Outlook Web App

Home | Applications | Services | Hosts | Registries | Servers | Status | Log | System

Home > Services

Services (2)

Service Name:	Application	Owner	Providers(2)	Consumers(3)	Statistics	Charts
com.alibaba.dubbo.monitor.MonitorService	simple-monitor		Providers(1)	Consumers(2)	Statistics	Charts
com.alibaba.dubbo.demo.DemoService	demo-provider	william	Providers(1)	Consumers(1)	Statistics	Charts

127.0.0.1:8080/statistics.html?service=com.alibaba.dubbo.demo.DemoService&date=20141119&expand=consumer

obao b2b alipay 资料 理财 Wikipedia Outlook Web App

Home | Applications | Services | Hosts | Registries | Servers | Status | Log | System

Home > Services > com.alibaba.dubbo.demo.DemoService > Providers | Consumers | Statistics | Charts > 20141119 > Summary | +Provider | +Consumer

Statistics (2)

Method:	Success	Failure	Avg Elapsed (ms)	Max Elapsed (ms)	Max Concurrent
sayHello	629 --> 629	0 --> 0	4 --> 0	759 --> 42	21 --> 21
|<-- 10.33.37.14	629 --> 629	0 --> 0	4 --> 0	759 --> 42	21 --> 21

127.0.0.1:8080/charts.html?service=com.alibaba.dubbo.demo.DemoService&date=20141119

obao b2b alipay 资料 理财 Wikipedia Outlook Web App

Home | Applications | Services | Hosts | Registries | Servers | Status | Log | System

Home > Services > com.alibaba.dubbo.demo.DemoService > Providers | Consumers | Statistics | Charts > 20141119

Charts (1)

Method	Requests per second (QPS)	Average response time (ms)
sayHello