

开发者指南

- 参与
 - 流程
 - 任务
- 版本管理
- 源码构建
- 框架设计
 - 整体设计
 - 模块分包
 - 依赖关系
 - 调用链
 - 暴露服务时序
 - 引用服务时序
 - 领域模型
 - 基本原则
- 扩展点加载
 - 扩展点配置
 - 扩展点自动包装
 - 扩展点自动装配
 - 扩展点自适应
 - 扩展点自动激活
- 实现细节
 - 初始化过程细节
 - 远程调用细节
 - 远程通讯细节
- SPI参考手册
 - 协议扩展
 - 调用拦截扩展
 - 引用监听扩展
 - 暴露监听扩展
 - 集群扩展
 - 路由扩展
 - 负载均衡扩展
 - 合并结果扩展
 - 注册中心扩展
 - 监控中心扩展
 - 扩展点加载扩展
 - 动态代理扩展
 - 编译器扩展
 - 消息派发扩展
 - 线程池扩展
 - 序列化扩展
 - 网络传输扩展
 - 信息交换扩展
 - 组网扩展
 - Telnet命令扩展
 - 状态检查扩展
 - 容器扩展
 - 页面扩展
 - 缓存扩展
 - 验证扩展
 - 日志适配扩展
- 技术兼容性测试
 - Protocol TCK
 - Registry TCK
- 公共契约
 - URL
 - 日志
- 坏味道
 - URL转换
 - 调用参数
 - 扩展点的加载
 - Callback功能
 - Lazy连接
 - 共享连接
 - sticky 策略
 - 服务提供者选择逻辑
- 编码约定
- 检查列表
- 设计原则

参与

流程

(#)

- 1. 如果是扩展功能，直接新增工程，黑盒依赖Dubbo进行扩展。
- 2. 如果是改BUG，或修改框架本身，可以从Dubbo的GitHub上Fork工程。
- 3. 修改后通过Push Request反馈修改。

任务

(#)

功能	分类	优先级	状态	认领者	计划完成时间	进度
《用户指南》翻译	文档	高	未认领	待定	待定	0%
《开发指南》翻译	文档	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
扩展点兼容性测试	测试	高	已认领	罗立树	待定	0%
性能基准测试	测试	高	未认领	待定	待定	0%
功能单元测试	测试	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
JTA/XA分布式事务	拦截扩展	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
Thrift	协议扩展	高	开发完成	阎刚	2012-04-27	90%
ICE	协议扩展	高	未认领	待定	待定	0%
ACE	协议扩展	低	未认领	待定	待定	0%
JSON-RPC	协议扩展	低	未认领	待定	待定	0%
XML-RPC	协议扩展	低	未认领	待定	待定	0%
JSR181&CXF(WebService)	协议扩展	高	开发完成	白文志	2012-04-27	90%
JSR311&JSR339(RestfulWebService)	协议扩展	高	未认领	待定	待定	0%
JMS&ActiveMQ	协议扩展	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
Protobuf	序列化扩展	高	调研	朱启恒	2012-02-30	20%
Avro	序列化扩展	低	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
XSocket	传输扩展	低	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
CGLib	动态代理扩展	低	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
JNDI	注册中心扩展	高	未认领	待定	待定	0%
LDAP	注册中心扩展	低	未认领	待定	待定	0%
JSR140&SLP	注册中心扩展	高	未认领	待定	待定	0%
UDDI	注册中心扩展	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
JMX	监控中心扩展	高	未认领	待定	待定	0%
SNMP	监控中心扩展	高	未认领	待定	待定	0%
Cacti	监控中心扩展	高	未认领	待定	待定	0%
Nagios	监控中心扩展	高	未认领	待定	待定	0%
Logstash	监控中心扩展	高	未认领	待定	待定	0%
JRobin	监控中心扩展	高	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
Maven	服务安装包仓库	低	未认领	待定	待定	0%
Subversion	服务安装包仓库	低	未认领	待定	待定	0%
JCR/JSR283	服务安装包仓库	低	未认领	待定	待定	0%
功能	分类	优先级	状态	认领者	计划完成时间	进度
SimpleDeployer	本地部署代理	低	未认领	待定	待定	0%
SimpleScheduler	资源调度器	低	未认领	待定	待定	0%

版本管理

(+) (#)

新功能的开发 和 稳定性的提高 对产品都很重要。

但是添加新功能对影响稳定性，Dubbo使用如下的版本开发模式来保障两者。

2个版本并行开发

- **BugFix**版本，低版本，比如2.4.x。是**GA**版本，线上使用的版本，只会BugFix，升级第三位版本号。
这个版本可放在SVN的Fix分支上。
- 新功能版本，高版本，比如2.5.x。加新功能的版本，会给对新功能有需求的应用试用。
这个版本可放在SVN的Trunk上。

2.5.x的新功能基本稳定后，进入2.5.x试用阶段。找足够多的应用试用2.5.x版本。

在2.5.x够稳定后：

1. 2.5.x成为GA版本，只BugFix，推广使用此版本。
如何可行，可以推进应用在期望的时间点内升级到GA版本。
2. 2.4.x不再开发，应用碰到Bug让直接升级。（这个称为“夕阳条款”）
3. 从2.5.x拉成分支2.6.0，作为新功能开发版本。

优势

- 保持GA版本是稳定的！因为：
 - 只会作BugFix
 - 成为GA版本前有试用阶段
- 新功能可以高版本中快速响应，并让应用能试用新功能。
- 不会版本过多，导致开发和维护成本剧增

用户要配合的职责

由于开发只会BugFix GA版本，所以用户需要积极跟进升级到GA版本，以Fix发现的问题。

定期升级版本用户带来了不安。这是一个伪命题，说明如下：

- GA经过一个试用阶段保持稳定。
- GA版本有Bug会火速Fix
- 相对出问题才升级到GA版本（可以跨了多个版本）定期升级平摊风险（类似小步快跑）。
经历过周期长的大项目的同学会有这样的经历，三方库版本不时间不升级，结果出了问题不得不升级到新版本（跨了多个版本）风险巨大。

源码构建

(+) (#)

Browser:

To browse the source tree directly:

<https://github.com/alibaba/dubbo>

Git:

Use this command to check out the latest project source code:

```
git clone https://github.com/alibaba/dubbo dubbo
```

Powered by: [Git](#)

Branches

We use the trunk for the next main release; then we use a branch for any bug fixes on the previous major release. You can look at all branches here:

<https://github.com/alibaba/dubbo/tags>

Building

Dubbo uses **Maven** as its build tool. If you don't fancy using Maven you can use your IDE directly or **Download** a distribution or JAR.

Required:

- Java 1.5 or better
- Download and install Maven 2.2.1 or better.
- Get the latest Source

```
svn checkout http://code.alibabatech.com/svn/dubbo/trunk dubbo
```

Maven options

To build dubbo maven has to be configured to use more memory

```
set MAVEN_OPTS=-Xmx1024m -XX:MaxPermSize=512m
```

A normal build

```
mvn install
```

Doing a Quick Build

Available as of Dubbo 2.0

The following skips building the manual, the distro and does not execute the unit tests.

```
mvn install -Dmaven.test.skip
```

Using an IDE

If you prefer to use an IDE then you can auto-generate the IDE's project files using maven plugins. e.g.

```
mvn eclipse:eclipse
```

or

```
mvn idea:idea
```

Importing into Eclipse

If you have not already done so, you will need to make Eclipse aware of the Maven repository so that it can build everything. In the preferences, go to Java->Build Path->Classpath and define a new Classpath Variable named M2_REPO that points to your local Maven repository (i.e., ~/m2/repository on Unix and c:\Documents and Settings\<user>\m2repository on Windows).

You can also get Maven to do this for you:

```
mvn eclipse:configure-workspace -Declipse.workspace=/path/to/the/workspace/
```

Building source jars

If you want to build jar files with the source code, that for instance Eclipse can import so you can debug the Dubbo code as well. Then you can run this command from the dubbo root folder:

```
mvn clean source:jar install -Dmaven.test.skip
```

框架设计

(+)(#)

整体设计



如果你觉得图过于复杂，请查看：[>>框架图绘制步骤动画](#)

图例说明：

- 图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。
- 图中从下至上分为十层，各层均为单向依赖，右边的黑色箭头代表层之间的依赖关系，每一层都可以剥离上层被复用，其中，**Service**和**Config**层为API，其它各层均为SPI。
- 图中绿色小块的为扩展接口，蓝色小块为实现类，图中只显示用于关联各层的实现类。
- 图中蓝色虚线为初始化过程，即启动时组装链，红色实线为方法调用过程，即运行时调用链，紫色三角箭头为继承，可以把子类看作父类的同一个节点，线上的文字为调用的方法。

各层说明：

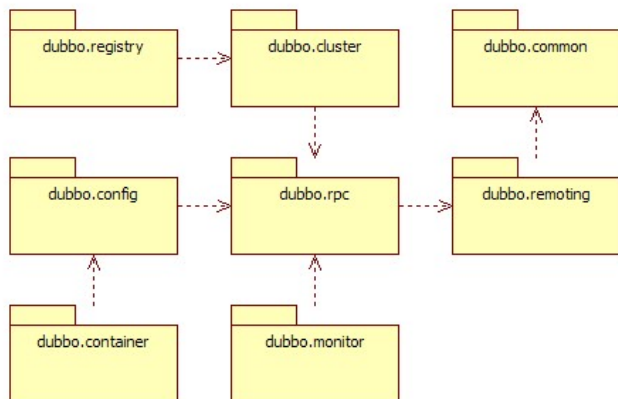
- **config**，配置层，对外配置接口，以**ServiceConfig**、**ReferenceConfig**为中心，可以直接new配置类，也可以通过spring解析配置生成配置类
- **proxy**，服务代理层，服务接口透明代理，生成服务的客户端**Stub**和服务端**Skeleton**，以**ServiceProxy**为中心，扩展接口为**ProxyFactory**
- **registry**，注册中心层，封装服务地址的注册与发现，以服务URL为中心，扩展接口为**RegistryFactory**、**Registry**、**RegistryService**
- **cluster**，路由层，封装多个提供者的路由及负载均衡，并桥接注册中心，以**Invoker**为中心，扩展接口为**Cluster**、**Directory**、**Router**、**LoadBalance**
- **monitor**，监控层，RPC调用次数和调用时间监控，以**Statistics**为中心，扩展接口为**MonitorFactory**、**Monitor**、**MonitorService**
- **protocol**，远程调用层，封装RPC调用，以**Invocation**、**Result**为中心，扩展接口为**Protocol**、**Invoker**、**Exporter**
- **exchange**，信息交换层，封装请求响应模式，同步转异步，以**Request**、**Response**为中心，扩展接口为**Exchanger**、**ExchangeChannel**、**ExchangeClient**、**ExchangeServer**

- transport, 网络传输层, 抽象mina和netty为统一接口, 以Message为中心, 扩展接口为Channel, Transporter, Client, Server, Codec
- serialize, 数据序列化层, 可复用的一些工具, 扩展接口为Serialization, ObjectInput, ObjectOutput, ThreadPool

关系说明:

- 在RPC中, Protocol是核心层, 也就是只要有Protocol + Invoker + Exporter就可以完成非透明的RPC调用, 然后在Invoker的主过程上Filter拦截点。
- 图中的Consumer和Provider是抽象概念, 只是想看看图者更直观的了解哪些类属于客户端与服务端, 不用Client和Server的原因是Dubbo在很多场景下都使用Provider, Consumer, Registry, Monitor划分逻辑拓普节点, 保持统一概念。
- 而Cluster是外围概念, 所以Cluster的目的是将多个Invoker伪装成一个Invoker, 这样其它人只要关注Protocol层Invoker即可, 加上Cluster或者去掉Cluster对其它层都不会造成影响, 因为只有一个提供者时, 是不需要Cluster的。
- Proxy层封装了所有接口的透明化代理, 而在其它层都以Invoker为中心, 只有到了暴露给用户使用时, 才用Proxy将Invoker转成接口, 或将接口实现转成Invoker, 也就是去掉Proxy层RPC是可以Run的, 只是不那么透明, 不那么看起来像调本地服务一样调远程服务。
- 而Remoting实现是Dubbo协议的实现, 如果你选择RMI协议, 整个Remoting都不会用上, Remoting内部再划为Transport传输层和Exchange信息交换层, Transport层只负责单向消息传输, 是对Mina, Netty, Grizzly的抽象, 它也可以扩展UDP传输, 而Exchange层是在传输层之上封装了Request-Response语义。
- Registry和Monitor实际上不算一层, 而是一个独立的节点, 只是为了全局概览, 用层的方式画在一起。

模块分包



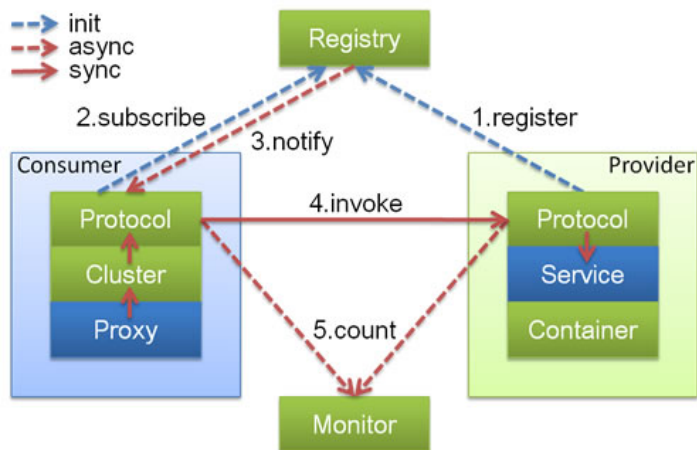
模块说明:

- dubbo-common 公共逻辑模块, 包括Util类和通用模型。
- dubbo-remoting 远程通讯模块, 相当于Dubbo协议的实现, 如果RPC用RMI协议则不需要使用此包。
- dubbo-rpc 远程调用模块, 抽象各种协议, 以及动态代理, 只包含一对一的调用, 不关心集群的管理。
- dubbo-cluster 集群模块, 将多个服务提供方伪装为一个提供方, 包括: 负载均衡, 容错, 路由等, 集群的地址列表可以是静态配置的, 也可以是由注册中心下发。
- dubbo-registry 注册中心模块, 基于注册中心下发地址的集群方式, 以及对各种注册中心的抽象。
- dubbo-monitor 监控模块, 统计服务调用次数, 调用时间的, 调用链跟踪的服务。
- dubbo-config 配置模块, 是Dubbo对外的API, 用户通过Config使用Dubbo, 隐藏Dubbo所有细节。
- dubbo-container 容器模块, 是一个Standalone的容器, 以简单的Main加载Spring启动, 因为服务通常不需要Tomcat/JBoss等Web容器的特性, 没必要用Web容器去加载服务。

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。

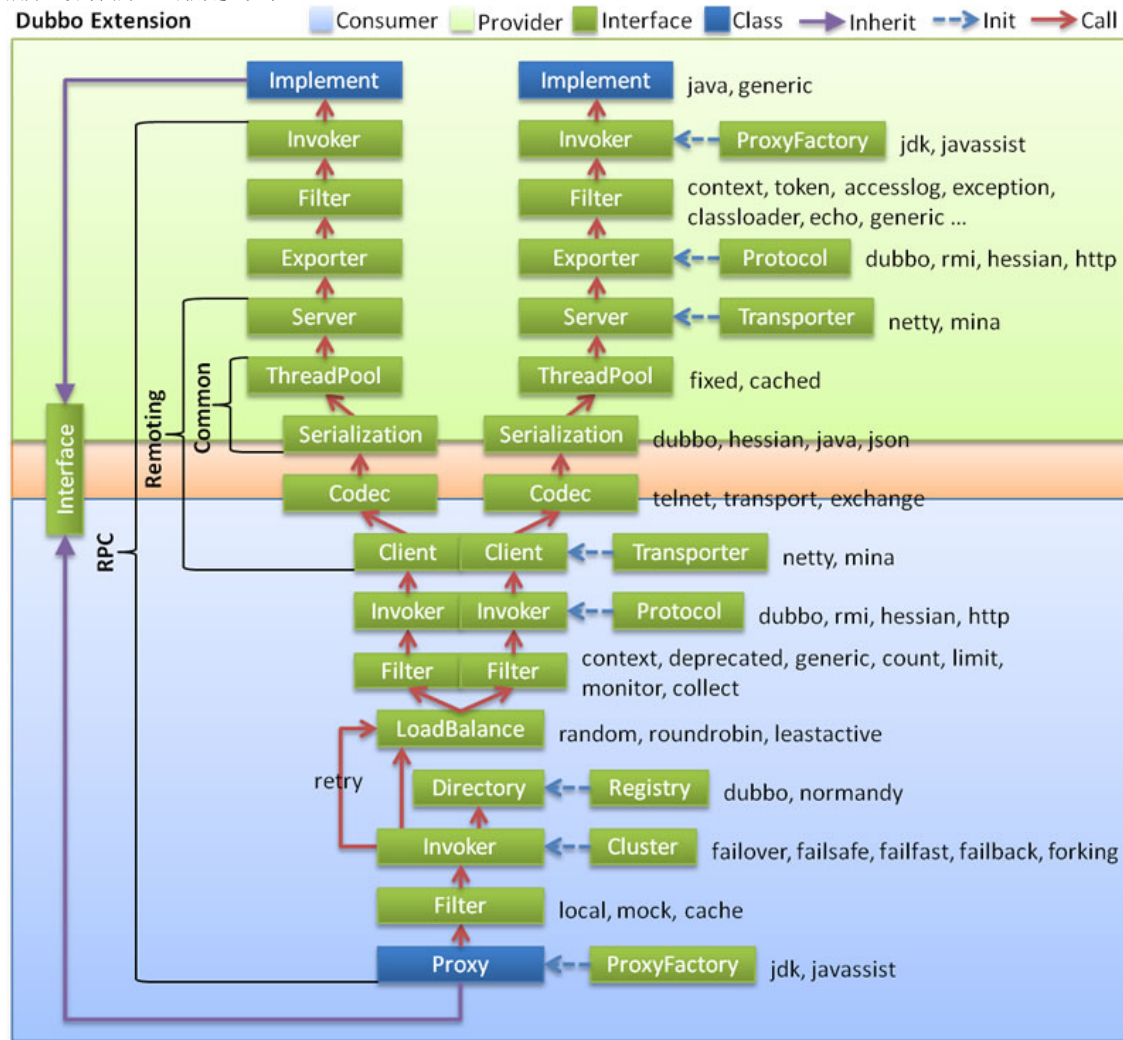
依赖关系



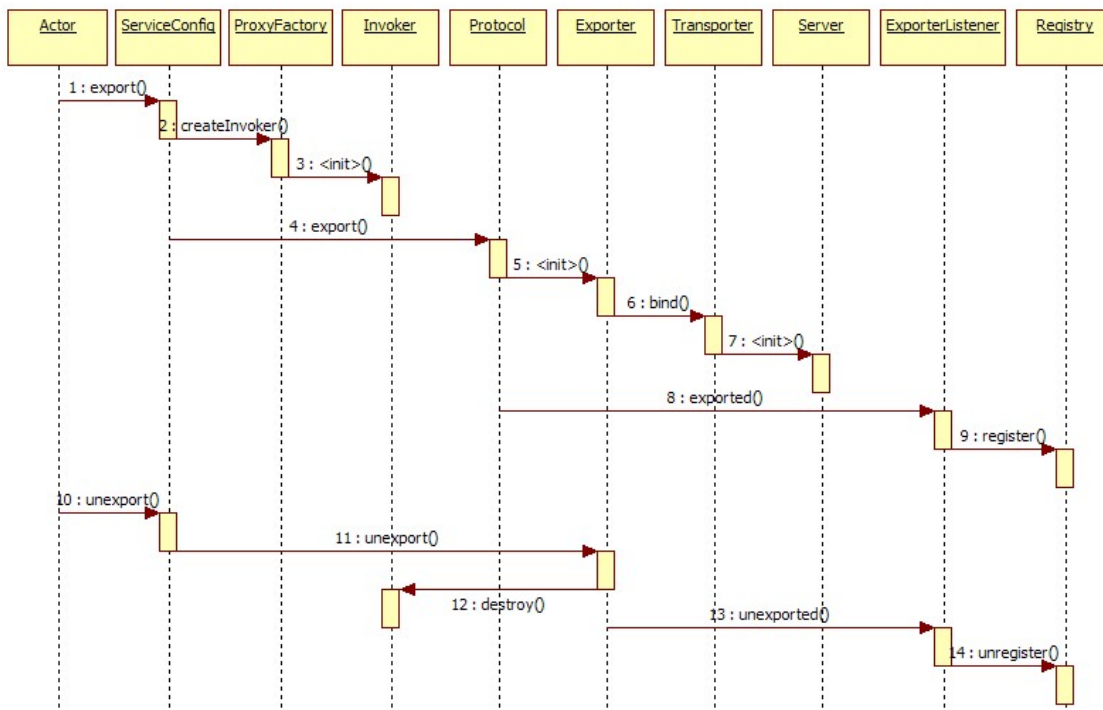
图例说明:

- 图中小方块Protocol, Cluster, Proxy, Service, Container, Registry, Monitor代表层或模块, 蓝色的表示与业务有交互, 绿色的表示只对Dubbo内部交互。
- 图中背景方块Consumer, Provider, Registry, Monitor代表部署逻辑拓普节点。
- 图中蓝色虚线为初始化时调用, 红色虚线为运行时异步调用, 红色实线为运行时同步调用。
- 图中只包含RPC的层, 不包含Remoting的层, Remoting整体都隐含在Protocol中。

展开总设计图的红色调用链，如下：

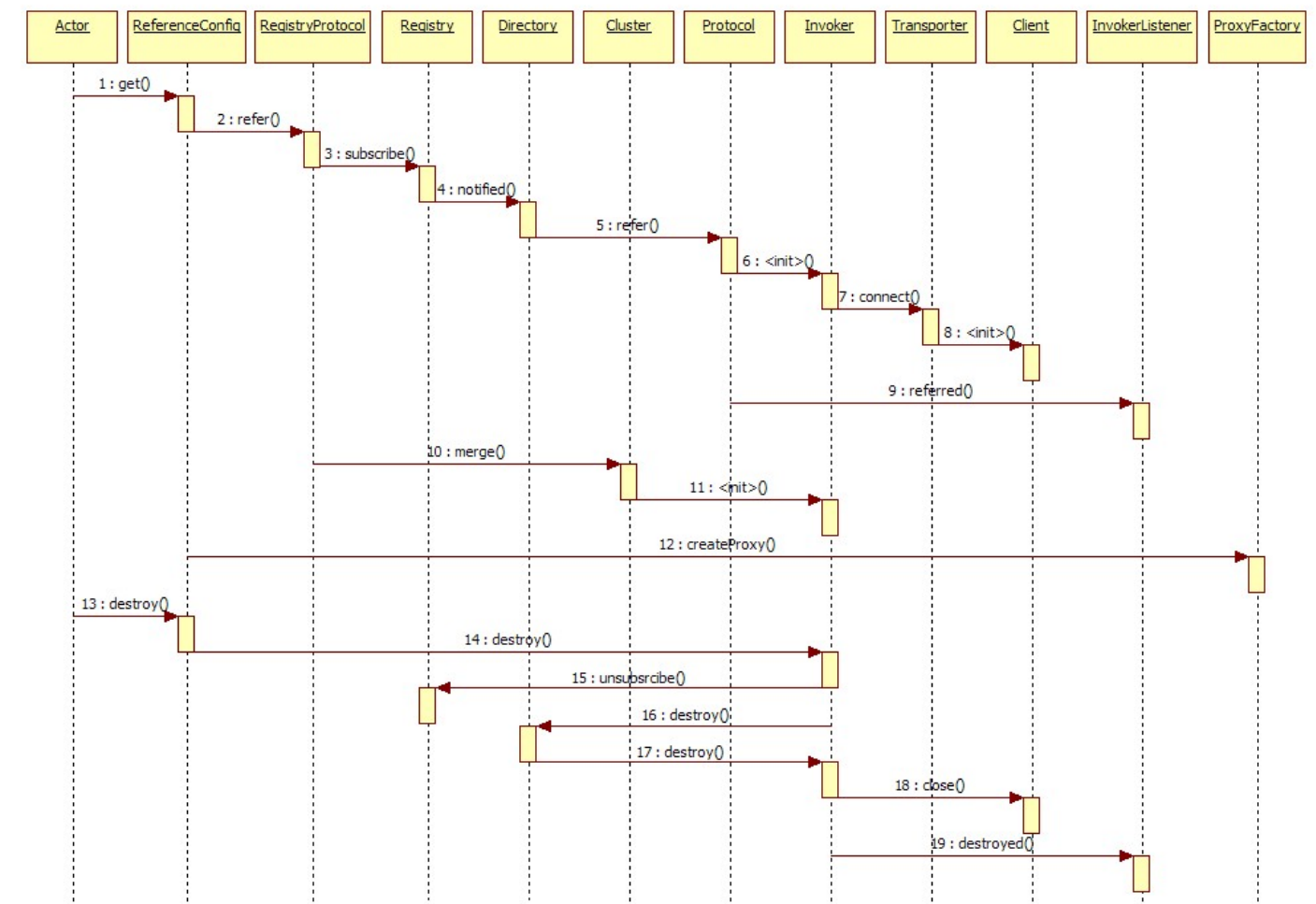


展开总设计图左边服务提供方暴露服务的蓝色初始化链，时序图如下：



引用服务时序

展开总设计图右边服务消费方引用服务的蓝色初始化链，时序图如下：



领域模型

在Dubbo的核心领域模型中：

- Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。
- Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
- Invocation是会话域，它持有调用过程中的变量，比如方法名，参数等。

基本原则

- 采用Microkernel + Plugin模式，Microkernel只负责组将Plugin，Dubbo自身的功能也是通过扩展点实现的，也就是Dubbo的所有功能点都可被用户自定义扩展所替换。
- 采用URL作为配置信息的统一格式，所有扩展点都通过传递URL携带配置信息。

更多设计原则参见：《框架设计原则》

扩展点加载

(+) (#)

扩展点配置

来源：

Dubbo的扩展点加载从JDK标准的SPI(Service Provider Interface)扩展点发现机制加强而来。

Dubbo改进了JDK标准的SPI的以下问题：

- JDK标准的SPI会一次性实例化扩展点所有实现，如果有扩展实现初始化很耗时，但如果没用上也加载，会很浪费资源。
- 如果扩展点加载失败，连扩展点的名称都拿不到了。比如：JDK标准的ScriptEngine，通过getName();获取脚本类型的名称，但如果RubyScriptEngine因为所依赖的jruby.jar不存在，导致RubyScriptEngine类加载失败，这个失败原因被吃掉了，和ruby对应不起来，当用户执行ruby脚本时，会报不支持ruby，而不是真正失败的原因。
- 增加了对扩展点IoC和AOP的支持，一个扩展点可以直接setter注入其它扩展点。

约定：

在扩展类的jar包内，放置扩展点配置文件：META-INF/dubbo/接口全限定名，内容为：配置名=扩展实现类全限定名，多个实现类用换行符分隔。

(注意：这里的配置文件是放在你自己的jar包内，不是dubbo本身的jar包内，Dubbo会全ClassPath扫描所有jar包内同名的这个文件，然后进行合并)

扩展Dubbo的协议示例：

在协议的实现jar包内放置文本文件：META-INF/dubbo/com.alibaba.dubbo.rpc.Protocol，内容为：

```
xxx=com.alibaba.xxx.XxxProtocol
```

实现类内容:

```
package com.alibaba.xxx;

import com.alibaba.dubbo.rpc.Protocol;

public class XxxProtocol implements Protocol {

    // ...

}
```

注意: 扩展点使用单一实例加载(请确保扩展实现的线程安全性), **Cache**在**ExtensionLoader**中。

扩展点自动包装

自动**Wrap**扩展点的**Wrapper**类

ExtensionLoader会把加载扩展点时(通过扩展点配置文件中内容), 如果该实现有拷贝构造函数, 则判定为扩展点**Wrapper**类。

Wrapper类同样实现了扩展点接口。

Wrapper类内容:

```
package com.alibaba.xxx;

import com.alibaba.dubbo.rpc.Protocol;

public class XxxProtocolWrapper implements Protocol {
    Protocol impl;

    public XxxProtocolWrapper(Protocol protocol) { impl = protocol; }

    // 接口方法做一个操作后, 再调用extension的方法
    public void refer() {
        //... 一些操作
        impl .refer();
        // ... 一些操作
    }

    // ...

}
```

Wrapper不是扩展点实现, 用于从**ExtensionLoader**返回扩展点时, **Wrap**在扩展点实现外。即从**ExtensionLoader**中返回的实际上是**Wrapper**类的实例, **Wrapper**持有了实际的扩展点实现类。

扩展点的**Wrapper**类可以有多个, 也可以根据需要新增。

通过**Wrapper**类可以把所有扩展点公共逻辑移至**Wrapper**中。新加的**Wrapper**在所有的扩展点上添加了逻辑, 有些类似AOP (**Wrapper**代理了扩展点)。

扩展点自动装配

加载扩展点时, 自动注入依赖的扩展点

加载扩展点时, 扩展点实现类的成员如果为其它扩展点类型, **ExtensionLoader**在会自动注入依赖的扩展点。

ExtensionLoader通过扫描扩展点实现类的所有**set**方法来判定其成员。

即**ExtensionLoader**会执行扩展点的拼装操作。

示例: 有两个为扩展点**CarMaker**(造车者)、**wheelMaker**(造轮者)

接口类如下:

```
public interface CarMaker {
    Car makeCar();
}

public interface WheelMaker {
    Wheel makeWheel();
}
```

CarMaker的一个实现类:

```
public class RaceCarMaker implements CarMaker {
    WheelMaker wheelMaker;

    public setWheelMaker(WheelMaker wheelMaker) {
        this.wheelMaker = wheelMaker;
    }
}
```



```

public Car makeCar() {
    // ...
    Wheel wheel = wheelMaker.makeWheel();
    // ...
    return new RaceCar(wheel, ...);
}
}

```

ExtensionLoader加载CarMaker的扩展点实现RaceCar时，setWheelMaker方法的WheelMaker也是扩展点则会注入WheelMaker的实现。

这里带来另一个问题，ExtensionLoader要注入依赖扩展点时，如何决定要注入依赖扩展点的哪个实现。在这个示例中，即是在多个WheelMaker的实现中要注入哪个。这个问题在下面一点“Adaptive实例”中说明。

扩展点自适应

扩展点的**Adaptive**实例

ExtensionLoader注入的依赖扩展点是一个Adaptive实例，直到扩展点方法执行时才决定调用是一个扩展点实现。

Dubbo使用URL对象（包含了Key-Value）传递配置信息。

扩展点方法调用会有URL参数（或是参数有URL成员）

这样依赖的扩展点也可以从URL拿到配置信息，所有的扩展点自己定好配置的Key后，配置信息从URL上从最外层传入。URL在配置传递上即是一条总线。

示例：有两个为扩展点CarMaker（造车者）、wheelMaker(造轮者)

接口类如下：

```

public interface CarMaker {
    Car makeCar(URL url);
}

public interface WheelMaker {
    Wheel makeWheel(URL url);
}

```

CarMaker的一个实现类：

```

public class RaceCarMaker implements CarMaker {
    WheelMaker wheelMaker;

    public setWheelMaker(WheelMaker wheelMaker) {
        this.wheelMaker = wheelMaker;
    }

    public Car makeCar(URL url) {
        // ...
        Wheel wheel = wheelMaker.makeWheel(url);
        // ...
        return new RaceCar(wheel, ...);
    }
}

```

当上面执行

```

// ...
Wheel wheel = wheelMaker.makeWheel(url);
// ...

```

时，注入的Adaptive实例可以提取约定Key来决定使用哪个WheelMaker实现来调用对应实现的真正的makeWheel方法。

如提取wheel.type key即url.get("wheel.type")来决定WheelMake实现。

Adaptive实例的逻辑是固定，指定提取的URL的Key，即可以代理真正的实现类上，可以动态生成。

在Dubbo的ExtensionLoader的扩展点类开对应的Adaptive实现是在加载扩展点里动态生成。指定提取的URL的Key通过@Adaptive注解在接口方法上提供。

下面是Dubbo的Transporter扩展点的代码：

```

public interface Transporter {
    @Adaptive({"server", "transport"})
    Server bind(URL url, ChannelHandler handler) throws RemotingException;

    @Adaptive({"client", "transport"})
    Client connect(URL url, ChannelHandler handler) throws RemotingException;
}

```

对于bind方法表示，Adaptive实现先查找"server"key，如果该Key没有值则找"transport"key值，来决定代理到哪个实际扩展点。

3. Dubbo配置模块中扩展点的配置

Dubbo配置模块中，扩展点均有对应配置属性或标签，通过配置指定使用哪个扩展实现。

比如: `<dubbo:protocol name="xxx" />`

扩展点自动激活

对于集合类扩展点, 比如: `Filter`, `InvokerListener`, `ExportListener`, `TelnetHandler`, `StatusChecker`等, 可以同时加载多个实现, 此时, 可以用自动激活来简化配置, 如:

```
import com.alibaba.dubbo.common.extension.Activate;
import com.alibaba.dubbo.rpc.Filter;

@Activate // 无条件自动激活
public class XxxFilter implements Filter {
    // ...
}
```

或:

```
import com.alibaba.dubbo.common.extension.Activate;
import com.alibaba.dubbo.rpc.Filter;

@Activate("xxx") // 当配置了xxx参数, 并且参数为有效值时激活, 比如配了cache="lru", 自动激活CacheFilter。
public class XxxFilter implements Filter {
    // ...
}
```

或:

```
import com.alibaba.dubbo.common.extension.Activate;
import com.alibaba.dubbo.rpc.Filter;

@Activate(group = "provider", value = "xxx") // 只对提供方激活, group可选"provider"或"consumer"
public class XxxFilter implements Filter {
    // ...
}
```

实现细节

(+) (#)

初始化过程细节

(+) (#)

解析服务

- 基于dubbo.jar内的META-INF/spring.handlers配置, Spring在遇到dubbo名称空间时, 会回调DubboNamespaceHandler。
- 所有dubbo的标签, 都统一用DubboBeanDefinitionParser进行解析, 基于一对一属性映射, 将XML标签解析为Bean对象。
- 在ServiceConfig.export()或ReferenceConfig.get()初始化时, 将Bean对象转换URL格式, 所有Bean属性转成URL的参数。
- 然后将URL传给Protocol扩展点, 基于扩展点的Adaptive机制, 根据URL的协议头, 进行不同协议的服务暴露或引用。

暴露服务

(1) 只暴露服务端口:

- 在没有注册中心, 直接暴露提供者的情况下, 即:
 - `<dubbo:service registry="N/A" />` or `<dubbo:registry address="N/A" />`
- ServiceConfig解析出的URL的格式为:
 - `dubbo://service-host/com.foo.FooService?version=1.0.0`
- 基于扩展点的Adaptive机制, 通过URL的"dubbo://"协议头识别, 直接调用DubboProtocol的export()方法, 打开服务端口。

(2) 向注册中心暴露服务:

- 在有注册中心, 需要注册提供者地址的情况下, 即:
 - `<dubbo:registry address="zookeeper://10.20.153.10:2181" />`
- ServiceConfig解析出的URL的格式为:
 - `registry://registry-host/com.alibaba.dubbo.registry.RegistryService?export=URL.encode("dubbo://service-host/com.foo.FooService?version=1.0.0")`
- 基于扩展点的Adaptive机制, 通过URL的"registry://"协议头识别, 就会调用RegistryProtocol的export()方法, 将export参数中的提供者URL, 先注册到注册中心, 再重新传给Protocol扩展点进行暴露:
 - `dubbo://service-host/com.foo.FooService?version=1.0.0`
- 基于扩展点的Adaptive机制, 通过提供者URL的"dubbo://"协议头识别, 就会调用DubboProtocol的export()方法, 打开服务端口。

引用服务

(1) 直连引用服务:

- 在没有注册中心, 直连提供者的情况下, 即:
 - `<dubbo:reference url="dubbo://service-host/com.foo.FooService?version=1.0.0" />`
- ReferenceConfig解析出的URL的格式为:
 - `dubbo://service-host/com.foo.FooService?version=1.0.0`

- 基于扩展点的Adaptive机制，通过URL的"dubbo://"协议头识别，直接调用DubboProtocol的refer()方法，返回提供者引用。

(2) 从注册中心发现引用服务：

- 在有注册中心，通过注册中心发现提供者地址的情况下，即：
 - <dubbo:registry address="zookeeper://10.20.153.10:2181" />
- ReferenceConfig解析出的URL的格式为：
 - registry://registry-host/com.alibaba.dubbo.registry.RegistryService?refer=URL.encode("consumer://consumer-host/com.foo.FooService?version=1.0.0")
- 基于扩展点的Adaptive机制，通过URL的"registry://"协议头识别，就会调用RegistryProtocol的refer()方法，基于refer参数中的条件，查询提供者URL，如：
 - dubbo://service-host/com.foo.FooService?version=1.0.0
- 基于扩展点的Adaptive机制，通过提供者URL的"dubbo://"协议头识别，就会调用DubboProtocol的refer()方法，得到提供者引用。
- 然后RegistryProtocol将多个提供者引用，通过Cluster扩展点，伪装成单个提供者引用返回。

拦截服务

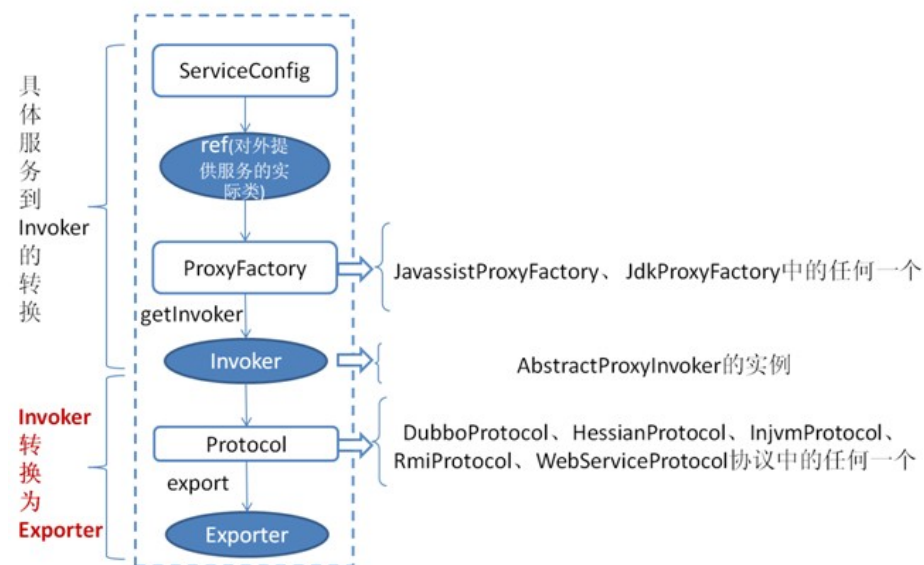
- 基于扩展点的Wrapper机制，所有的Protocol扩展点都会自动套上Wrapper类。
- 基于ProtocolFilterWrapper类，将所有Filter组装成链，在链的最后一节调用真实的引用。
- 基于ProtocolListenerWrapper类，将所有InvokerListener和ExporterListener组装集合，在暴露和引用前后，进行回调。
- 包括监控在内，所有附加功能，全部通过Filter拦截实现。

远程调用细节

(+) (#)

✓ 作者：白文志 (来自开源社区)

服务提供者暴露一个服务的详细过程



上图是服务提供者暴露服务的主过程：

首先ServiceConfig类拿到对外提供服务的实际类ref(如：HelloWorldImpl),然后通过ProxyFactory类的getInvoker方法使用ref生成一个AbstractProxyInvoker实例，到这一步就完成具体服务到Invoker的转化。接下来就是Invoker转换到Exporter的过程。

Dubbo处理服务暴露的关键就在Invoker转换到Exporter的过程(如上图中的红色部分)，下面我们以Dubbo和RMI这两种典型协议的实现来进行说明：

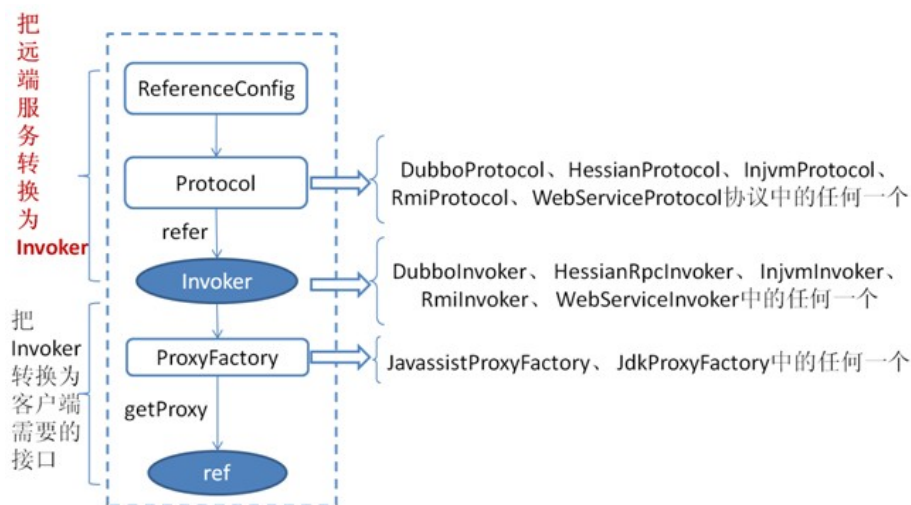
Dubbo的实现

Dubbo协议的Invoker转为Exporter发生在DubboProtocol类的export方法，它主要是打开socket侦听服务，并接收客户端发来的各种请求，通讯细节由Dubbo自己实现。

RMI的实现

RMI协议的Invoker转为Exporter发生在RmiProtocol类的export方法，它通过Spring或Dubbo或JDK来实现RMI服务，通讯细节这一块由JDK底层来实现，这就省了不少工作量。

服务消费者消费一个服务的详细过程



上图是服务消费的主过程：

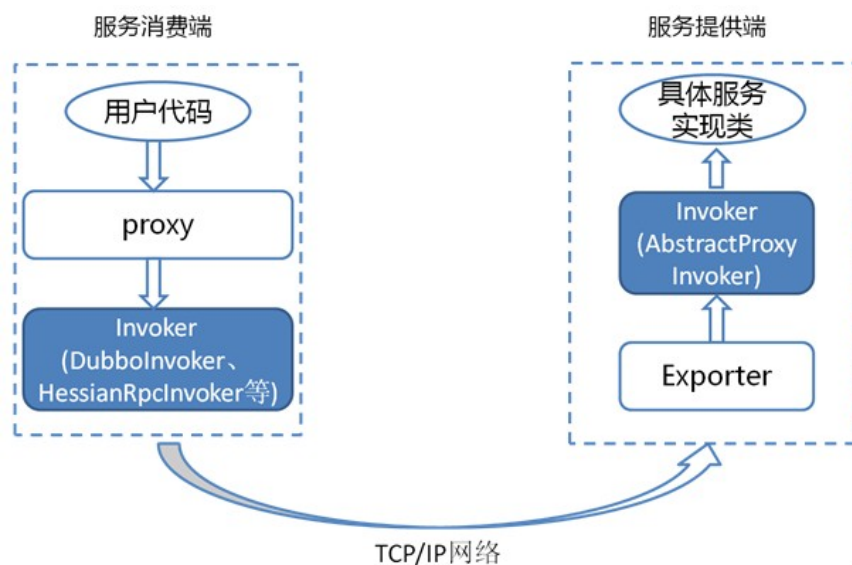
首先ReferenceConfig类的init方法调用Protocol的refer方法生成Invoker实例(如上图中的红色部分)，这是服务消费的关键。接下来把Invoker转换为客户端需要的接口(如：HelloWorld)。

关于每种协议如RMI/Dubbo/Web service等它们在调用refer方法生成Invoker实例的细节和上一章节所描述的类似。

满眼都是Invoker

由于Invoker是Dubbo领域模型中非常重要的一个概念，很多设计思路都是向它靠拢。这就使得Invoker渗透在整个实现代码里，对于刚开始接触Dubbo的人，确实容易给搞混了。

下面我们用一个精简的图来说明最重要的两种Invoker：服务提供Invoker和服务消费Invoker：



为了更好的解释上面这张图，我们结合服务消费和提供者的代码示例来进行说明：

服务消费者代码

```
public class DemoClientAction {
    private DemoService demoService;

    public void setDemoService(DemoService demoService) {
        this.demoService = demoService;
    }

    public void start() {
        String hello = demoService.sayHello("world" + i);
    }
}
```

上面代码中的'DemoService'就是上图中服务消费端的proxy，用户代码通过这个proxy调用其对应的Invoker(DubboInvoker、HessianRpcInvoker、InjvmInvoker、RmiInvoker、WebServiceInvoker中的任何一个)，而该Invoker实现了真正的远程服务调用。

服务提供者代码

```
public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) throws RemoteException {
        return "Hello " + name;
    }
}
```

```
}

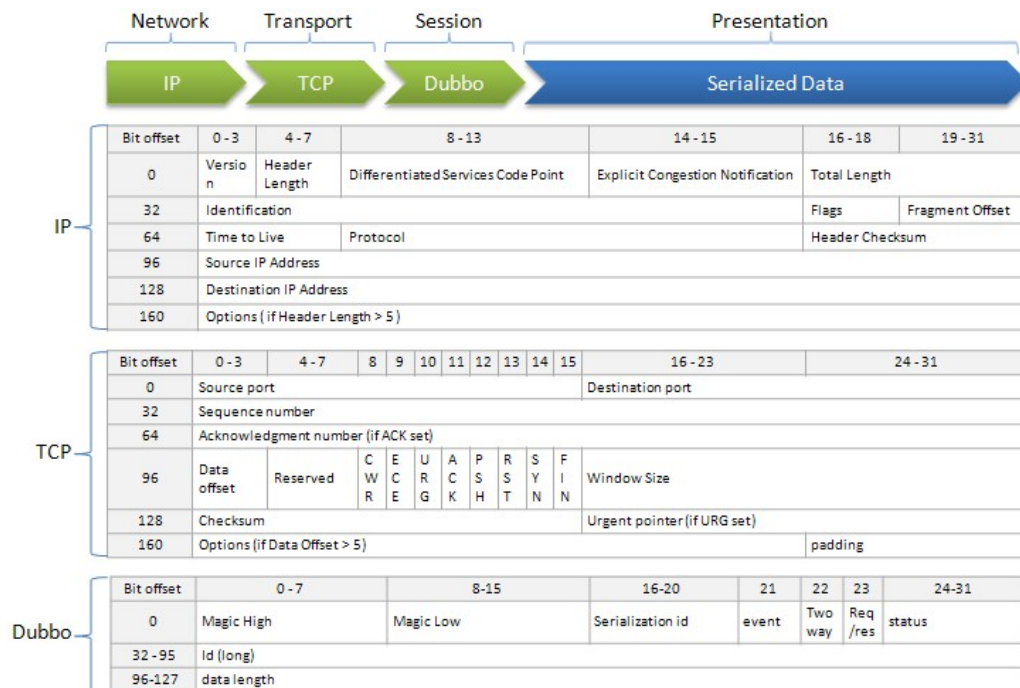
```

上面这个类会被封装成为一个AbstractProxyInvoker实例，并新生成一个Exporter实例。这样当网络通讯层收到一个请求后，会找到对应的Exporter实例，并调用它所对应的AbstractProxyInvoker实例，从而真正调用了服务提供者的代码。Dubbo里还有一些其他的Invoker类，但上面两种是最重要的。

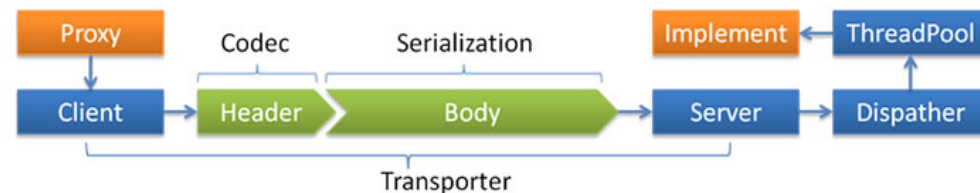
远程通讯细节

(+)(#)

协议头约定



线程派发模型



- Dispatcher
 - all, direct, message, execution, connection
- ThreadPool
 - fixed, cached

SPI参考手册

(+)(#)

⚠ SPI使用范围

扩展接口仅用于系统集成，或Contributor扩展功能插件。

协议扩展

(+)(#)

(1) 扩展说明：

RPC协议扩展，封装远程调用细节。

契约：

- 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。
- 其中，refer()返回的Invoker由协议实现，协议通常需在此Invoker中发送远程请求，export()传入的Invoker由框架实现并传入，协议不需要关心。

注意：

- 协议不关心业务接口的透明代理，以Invoker为中心，由外层将Invoker转换为业务接口。

- 协议不一定要是TCP网络通讯，比如通过共享文件，IPC进程间通讯等。

(2) 扩展接口：

```
com.alibaba.dubbo.rpc.Protocol
com.alibaba.dubbo.rpc.Exporter
com.alibaba.dubbo.rpc.Invoker
```

```
public interface Protocol {

    /**
     * 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
     */
    <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，需内部自动恢复。<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
     */
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

}
```

(3) 扩展配置：

```
<dubbo:protocol id="xxx1" name="xxx" /> <!-- 声明协议，如果没有配置id，将以name为id -->
<dubbo:service protocol="xxx1" /> <!-- 引用协议，如果没有配置protocol属性，将在ApplicationContext中自动扫描protocol配置 -->
<dubbo:provider protocol="xxx1" /> <!-- 引用协议缺省值，当<dubbo:service>没有配置prototol属性时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.rpc.injvm.InjvmProtocol
com.alibaba.dubbo.rpc.dubbo.DubboProtocol
com.alibaba.dubbo.rpc.rmi.RmiProtocol
com.alibaba.dubbo.rpc.http.HttpProtocol
com.alibaba.dubbo.rpc.http.hessian.HessianProtocol
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
        |-XxxProtocol.java (实现Protocol接口)
        |-XxxExporter.java (实现Exporter接口)
        |-XxxInvoker.java (实现Invoker接口)
      |-resources
        |-META-INF
          |-dubbo
            |-com.alibaba.dubbo.rpc.Protocol (纯文本文件，内容为: xxx=com.xxx.XxxProtocol)
```

XxxProtocol.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.Protocol;

public class XxxProtocol implements Protocol {
```

```

    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        return new XxxExporter(invoker);
    }
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        return new XxxInvoker(type, url);
    }
}

```

XxxExporter.java

```

package com.xxx;

import com.alibaba.dubbo.rpc.support.AbstractExporter;

public class XxxExporter<T> extends AbstractExporter<T> {
    public XxxExporter(Invoker<T> invoker) throws RemotingException{
        super(invoker);
        // ...
    }
    public void unexport() {
        super.unexport();
        // ...
    }
}

```

XxxInvoker.java

```

package com.xxx;

import com.alibaba.dubbo.rpc.support.AbstractInvoker;

public class XxxInvoker<T> extends AbstractInvoker<T> {
    public XxxInvoker(Class<T> type, URL url) throws RemotingException{
        super(type, url);
    }
    protected abstract Object doInvoke(Invocation invocation) throws Throwable {
        // ...
    }
}

```

META-INF/dubbo/com.alibaba.dubbo.rpc.Protocol

```
xxx=com.xxx.XxxProtocol
```

调用拦截扩展

(+) (#)

(1) 扩展说明

服务提供方和服务消费方调用过程拦截，Dubbo本身的大多功能均基于此扩展点实现，每次远程方法执行，该拦截都会被执行，请注意对性能的影响。

约定：

- 用户自定义filter默认在内置filter之后。
- 特殊值default，表示缺省扩展点插入的位置。
 - 比如：filter="xxx,default,yyy"，表示xxx在缺省filter之前，yyy在缺省filter之后。
- 特殊符号-，表示剔除。
 - 比如：filter="-foo1"，剔除添加缺省扩展点foo1。
 - 比如：filter="-default"，剔除添加所有缺省扩展点。
- provider和service同时配置的filter时，累加所有filter，而不是覆盖。
 - 比如：<dubbo:provider filter="xxx,yyy" />和<dubbo:service filter="aaa,bbb" />，则xxx,yyy,aaa,bbb均会生效。
 - 如果要覆盖，需配置：<dubbo:service filter="-xxx,-yyy,aaa,bbb" />

(2) 扩展接口：

```
com.alibaba.dubbo.rpc.Filter
```

(3) 扩展配置：

```

<dubbo:reference filter="xxx,yyy" /> <!-- 消费方调用过程拦截 -->
<dubbo:consumer filter="xxx,yyy"/> <!-- 消费方调用过程缺省拦截器，将拦截所有reference -->
<dubbo:service filter="xxx,yyy" /> <!-- 提供方调用过程拦截 -->
<dubbo:provider filter="xxx,yyy"/> <!-- 提供方调用过程缺省拦截器，将拦截所有service -->

```

(4) 已知扩展：


```
com.alibaba.dubbo.rpc.filter.EchoFilter
com.alibaba.dubbo.rpc.filter.GenericFilter
com.alibaba.dubbo.rpc.filter.GenericImplFilter
com.alibaba.dubbo.rpc.filter.TokenFilter
com.alibaba.dubbo.rpc.filter.AccessLogFilter
com.alibaba.dubbo.rpc.filter.CountFilter
com.alibaba.dubbo.rpc.filter.ActiveLimitFilter
com.alibaba.dubbo.rpc.filter.ClassLoaderFilter
com.alibaba.dubbo.rpc.filter.ContextFilter
com.alibaba.dubbo.rpc.filter.ConsumerContextFilter
com.alibaba.dubbo.rpc.filter.ExceptionFilter
com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter
com.alibaba.dubbo.rpc.filter.DeprecatedFilter
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|-java
|   |-com
|       |-xxx
|           |-XxxFilter.java (实现Filter接口)
|-resources
|   |-META-INF
|       |-dubbo
|           |-com.alibaba.dubbo.rpc.Filter (纯文本文件, 内容为: xxx=com.xxx.XxxFilter)
```

XxxFilter.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.Filter;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.Invocation;
import com.alibaba.dubbo.rpc.Result;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxFilter implements Filter {
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        // before filter ...
        Result result = invoker.invoke(invocation);
        // after filter ...
        return result;
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.rpc.Filter

```
xxx=com.xxx.XxxFilter
```

引用监听扩展

(+) (#)

(1) 扩展说明:

当有服务引用时, 触发该事件。

(2) 扩展接口:

```
com.alibaba.dubbo.rpc.InvokerListener
```

(3) 扩展配置:

```
<dubbo:reference listener="xxx,yyy" /> <!-- 引用服务监听 -->
<dubbo:consumer listener="xxx,yyy" /> <!-- 引用服务缺省监听器 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.rpc.listener.DeprecatedInvokerListener
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|-java
|-com
|-xxx
|-XxxInvokerListener.java (实现InvokerListener接口)
|-resources
|-META-INF
|-dubbo
|-com.alibaba.dubbo.rpc.InvokerListener (纯文本文件, 内容为: xxx=com.xxx.XxxInvokerListener)
```

XxxInvokerListener.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.InvokerListener;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxInvokerListener implements InvokerListener {
    public void referred(Invoker<?> invoker) throws RpcException {
        // ...
    }
    public void destroyed(Invoker<?> invoker) throws RpcException {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.rpc.InvokerListener

```
xxx=com.xxx.XxxInvokerListener
```

暴露监听扩展

(+) (#)

(1) 扩展说明:

当有服务暴露时, 触发该事件。

(2) 扩展接口:

```
com.alibaba.dubbo.rpc.ExporterListener
```

(3) 扩展配置:

```
<dubbo:service listener="xxx,yyy" /> <!-- 暴露服务监听 -->
<dubbo:provider listener="xxx,yyy" /> <!-- 暴露服务缺省监听器 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.registry.directory.RegistryExporterListener
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|-java
|-com
|-xxx
|-XxxExporterListener.java (实现ExporterListener接口)
```

```
| -resources
|   |-META-INF
|     |-dubbo
|       |-com.alibaba.dubbo.rpc.ExporterListener (纯文本文件, 内容为: xxx=com.xxx.XxxExporterListener)
```

XxxExporterListener.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.ExporterListener;
import com.alibaba.dubbo.rpc.Exporter;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxExporterListener implements ExporterListener {
    public void exported(Exporter<?> exporter) throws RpcException {
        // ...
    }
    public void unexported(Exporter<?> exporter) throws RpcException {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.rpc.ExporterListener

```
xxx=com.xxx.XxxExporterListener
```

集群扩展

(+) (#)

(1) 扩展说明:

当有多个服务提供方时, 将多个服务提供方组织成一个集群, 并伪装成一个提供方。

(2) 扩展接口:

```
com.alibaba.dubbo.rpc.cluster.Cluster
```

(3) 扩展配置:

```
<dubbo:protocol cluster="xxx" />
<dubbo:provider cluster="xxx" /> <!-- 缺省值配置, 如果<dubbo:protocol>没有配置cluster时, 使用此配置 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.rpc.cluster.support.FailoverCluster
com.alibaba.dubbo.rpc.cluster.support.FailfastCluster
com.alibaba.dubbo.rpc.cluster.support.FailsafeCluster
com.alibaba.dubbo.rpc.cluster.support.FailbackCluster
com.alibaba.dubbo.rpc.cluster.support.ForkingCluster
com.alibaba.dubbo.rpc.cluster.support.AvailableCluster
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|  |-java
|    |-com
|      |-xxx
|        |-XxxCluster.java (实现Cluster接口)
|-resources
|  |-META-INF
|    |-dubbo
|      |-com.alibaba.dubbo.rpc.cluster.Cluster (纯文本文件, 内容为: xxx=com.xxx.XxxCluster)
```

XxxCluster.java

```
package com.xxx;
```

```
import com.alibaba.dubbo.rpc.cluster.Cluster;
import com.alibaba.dubbo.rpc.cluster.support.AbstractClusterInvoker;
import com.alibaba.dubbo.rpc.cluster.Directory;
import com.alibaba.dubbo.rpc.cluster.LoadBalance;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.Invocation;
import com.alibaba.dubbo.rpc.Result;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxCluster implements Cluster {
    public <T> Invoker<T> merge(Directory<T> directory) throws RpcException {
        return new AbstractClusterInvoker<T>(directory) {
            public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {
                // ...
            }
        };
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.rpc.cluster.Cluster

```
xxx=com.xxx.XxxCluster
```

路由扩展

(+) (#)

(1) 扩展说明：

从多个服务提供者中选择一个进行调用。

(2) 扩展接口：

```
com.alibaba.dubbo.rpc.cluster.RouterFactory
com.alibaba.dubbo.rpc.cluster.Router
```

(3) 扩展配置：

```
<dubbo:protocol router="xxx" />
<dubbo:provider router="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置loadbalance时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.rpc.cluster.router.ScriptRouterFactory
com.alibaba.dubbo.rpc.cluster.router.FileRouterFactory
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
|  |-java
|     |-com
|         |-xxx
|             |-XxxRouterFactory.java (实现LoadBalance接口)
|-resources
|  |-META-INF
|     |-dubbo
|         |-com.alibaba.dubbo.rpc.cluster.RouterFactory (纯文本文件，内容为：xxx=com.xxx.XxxRouterFactory)
```

XxxRouterFactory.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.cluster.RouterFactory;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.Invocation;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxRouterFactory implements RouterFactory {
```

```

    public <T> List<Invoker<T>> select(List<Invoker<T>> invokers, Invocation invocation) throws RpcException {
        // ...
    }
}

```

META-INF/dubbo/com.alibaba.dubbo.rpc.cluster.RouterFactory

```
xxx=com.xxx.XxxRouterFactory
```

负载均衡扩展

(+) (#)

(1) 扩展说明：

从多个服务提供者方中选择一个进行调用。

(2) 扩展接口：

```
com.alibaba.dubbo.rpc.cluster.LoadBalance
```

(3) 扩展配置：

```

<dubbo:protocol loadbalance="xxx" />
<dubbo:provider loadbalance="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置loadbalance时，使用此配置 -->

```

(4) 已知扩展：

```

com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance
com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance
com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance

```

(5) 扩展示例：

Maven项目结构

```

src
|-main
  |-java
    |-com
      |-xxx
        |-XxxLoadBalance.java (实现LoadBalance接口)
  |-resources
    |-META-INF
      |-dubbo
        |-com.alibaba.dubbo.rpc.cluster.LoadBalance (纯文本文件，内容为：xxx=com.xxx.XxxLoadBalance)

```

XxxLoadBalance.java

```

package com.xxx;

import com.alibaba.dubbo.rpc.cluster.LoadBalance;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.Invocation;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxLoadBalance implements LoadBalance {
    public <T> Invoker<T> select(List<Invoker<T>> invokers, Invocation invocation) throws RpcException {
        // ...
    }
}

```

META-INF/dubbo/com.alibaba.dubbo.rpc.cluster.LoadBalance

```
xxx=com.xxx.XxxLoadBalance
```

合并结果扩展

(+) (#)

(1) 扩展说明：

合并返回结果，用于分组聚合。

(2) 扩展接口：

```
com.alibaba.dubbo.rpc.cluster.Merger
```

(3) 扩展配置：

```
<dubbo:method merger="xxx" />
```

(4) 已知扩展：

```
com.alibaba.dubbo.rpc.cluster.merger.ArrayMerger
com.alibaba.dubbo.rpc.cluster.merger.ListMerger
com.alibaba.dubbo.rpc.cluster.merger.SetMerger
com.alibaba.dubbo.rpc.cluster.merger.MapMerger
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
        |-XxxMerger.java (实现Merger接口)
  |-resources
    |-META-INF
      |-dubbo
        |-com.alibaba.dubbo.rpc.cluster.Merger (纯文本文件，内容为: xxx=com.xxx.XxxMerger)
```

XxxMerger.java

```
package com.xxx;

import com.alibaba.dubbo.rpc.cluster.Merger;

public class XxxMerger<T> implements Merger<T> {
    public T merge(T... results) {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.rpc.cluster.Merger

```
xxx=com.xxx.XxxMerger
```

注册中心扩展

(+) (#)

(1) 扩展说明：

负责服务的注册与发现。

(2) 扩展接口：

```
com.alibaba.dubbo.registry.RegistryFactory
com.alibaba.dubbo.registry.Registry
```

(3) 扩展配置：

```
<dubbo:registry id="xxx1" address="xxx://ip:port" /> <!-- 定义注册中心 -->
<dubbo:service registry="xxx1" /> <!-- 引用注册中心, 如果没有配置registry属性, 将在ApplicationContext中自动扫描registry配置 -->
<dubbo:provider registry="xxx1" /> <!-- 引用注册中心缺省值, 当<dubbo:service>没有配置registry属性时, 使用此配置 -->
```

(4) 扩展契约:

RegistryFactory.java

```
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
     */
    Registry getRegistry(URL url);

}
```

RegistryService.java

```
public interface RegistryService { // Registry extends RegistryService

    /**
     * 注册服务.
     *
     * 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=overrides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);

    /**
     * 取消注册服务.
     *
     * 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);

    /**
     * 订阅服务.
     *
     * 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=overrides, 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&application=kylin
     * @param listener 变更事件监听器, 不允许为空
     */
    void subscribe(URL url, NotifyListener listener);

    /**
     * 取消订阅服务.
     *
     * 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&application=kylin
     * @param listener 变更事件监听器, 不允许为空
     */
    void unsubscribe(URL url, NotifyListener listener);

    /**
     * 查询注册列表, 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&application=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);  
}
```

NotifyListener.java

```
public interface NotifyListener {  
  
    /**  
     * 当收到服务变更通知时触发。  
     *  
     * 通知需处理契约: <br>  
     * 1. 总是以服务接口和数据类型为维度全量通知，即不会通知一个服务的同类型的部分数据，用户不需要对比上一次通知结果。<br>  
     * 2. 订阅时的第一次通知，必须是一个服务的所有类型数据的全量通知。<br>  
     * 3. 中途变更时，允许不同类型的数据分开通知，比如: providers, consumers, routes, overrides, 允许只通知其中一种类型，但该类型的数据必须是全量的，不是增量的。<br>  
     * 4. 如果一种类型的数据为空，需通知一个empty协议并带category参数的标识性URL数据。<br>  
     * 5. 通知者(即注册中心实现)需保证通知的顺序，比如: 单线程推送，队列串行化，带版本对比。<br>  
     *  
     * @param urls 已注册信息列表，总不为空，含义同{@link com.alibaba.dubbo.registry.RegistryService#lookup(URL)}的返回值。  
     */  
    void notify(List<URL> urls);  
}
```

(5) 已知扩展:

```
com.alibaba.dubbo.registry.support.dubbo.DubboRegistryFactory
```

(6) 扩展示例:

Maven项目结构

```
src  
|-main  
  |-java  
    |-com  
      |-xxx  
        |-XxxRegistryFactoryjava (实现RegistryFactory接口)  
        |-XxxRegistry.java (实现Registry接口)  
  |-resources  
    |-META-INF  
      |-dubbo  
        |-com.alibaba.dubbo.registry.RegistryFactory (纯文本文件，内容为: xxx=com.xxx.XxxRegistryFactory)
```

XxxRegistryFactory.java

```
package com.xxx;  
  
import com.alibaba.dubbo.registry.RegistryFactory;  
import com.alibaba.dubbo.registry.Registry;  
import com.alibaba.dubbo.common.URL;  
  
public class XxxRegistryFactory implements RegistryFactory {  
    public Registry getRegistry(URL url) {  
        return new XxxRegistry(url);  
    }  
}
```

XxxRegistry.java

```
package com.xxx;  
  
import com.alibaba.dubbo.registry.Registry;  
import com.alibaba.dubbo.registry.NotifyListener;  
import com.alibaba.dubbo.common.URL;  
  
public class XxxRegistry implements Registry {  
    public void register(URL url) {  
        // ...  
    }  
    public void unregister(URL url) {  
        // ...  
    }  
    public void subscribe(URL url, NotifyListener listener) {  
        // ...  
    }  
}
```

```
    public void unsubscribe(URL url, NotifyListener listener) {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.registry.RegistryFactory

```
xxx=com.xxx.XxxRegistryFactory
```

监控中心扩展

(+) (#)

(1) 扩展说明：

负责服务调用次和调用时间的监控。

(2) 扩展接口：

```
com.alibaba.dubbo.monitor.MonitorFactory
com.alibaba.dubbo.monitor.Monitor
```

(3) 扩展配置：

```
<dubbo:monitor address="xxx://ip:port" /> <!-- 定义监控中心 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.monitor.support.dubbo.DubboMonitorFactory
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
        |-XxxMonitorFactory.java (实现MonitorFactory接口)
        |-XxxMonitor.java (实现Monitor接口)
      |-resources
        |-META-INF
          |-dubbo
            |-com.alibaba.dubbo.monitor.MonitorFactory (纯文本文件，内容为：xxx=com.xxx.XxxMonitorFactory)
```

XxxMonitorFactory.java

```
package com.xxx;

import com.alibaba.dubbo.monitor.MonitorFactory;
import com.alibaba.dubbo.monitor.Monitor;
import com.alibaba.dubbo.common.URL;

public class XxxMonitorFactory implements MonitorFactory {
    public Monitor getMonitor(URL url) {
        return new XxxMonitor(url);
    }
}
```

XxxMonitor.java

```
package com.xxx;

import com.alibaba.dubbo.monitor.Monitor;

public class XxxMonitor implements Monitor {
    public void count(URL statistics) {
        // ...
    }
}
```

```
}  
}
```

META-INF/dubbo/com.alibaba.dubbo.monitor.MonitorFactory

```
xxx=com.xxx.XxxMonitorFactory
```

扩展点加载扩展

(+) (#)

(1) 扩展说明：

扩展点本身的加载容器，可从不同容器加载扩展点。

(2) 扩展接口：

```
com.alibaba.dubbo.common.extension.ExtensionFactory
```

(3) 扩展配置：

```
<dubbo:application compiler="jdk" />
```

(4) 已知扩展：

```
com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory  
com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory
```

(5) 扩展示例：

Maven项目结构

```
src  
|-main  
  |-java  
    |-com  
      |-xxx  
        |-XxxExtensionFactory.java (实现ExtensionFactory接口)  
  |-resources  
    |-META-INF  
      |-dubbo  
        |-com.alibaba.dubbo.common.extension.ExtensionFactory (纯文本文件，内容为：xxx=com.xxx.XxxExtensionFactory)
```

XxxExtensionFactory.java

```
package com.xxx;  
  
import com.alibaba.dubbo.common.extension.ExtensionFactory;  
  
public class XxxExtensionFactory implements ExtensionFactory {  
    public Object getExtension(Class<?> type, String name) {  
        // ...  
    }  
}
```

META-INF/dubbo/com.alibaba.dubbo.common.extension.ExtensionFactory

```
xxx=com.xxx.XxxExtensionFactory
```

动态代理扩展

(+) (#)

(1) 扩展说明：

将Invoker接口转换成业务接口。

(2) 扩展接口:

```
com.alibaba.dubbo.rpc.ProxyFactory
```

(3) 扩展配置:

```
<dubbo:protocol proxy="xxx" />
<dubbo:provider proxy="xxx" /> <!-- 缺省值配置，当<dubbo:protocol>没有配置proxy属性时，使用此配置 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.rpc.proxy.JdkProxyFactory
com.alibaba.dubbo.rpc.proxy.JavassistProxyFactory
```

(5) 扩展示例:

Maven项目结构

src
|-main
 |-java
 |-com
 |-xxx
 |-XxxProxyFactory.java (实现ProxyFactory接口)
 |-resources
 |-META-INF
 |-dubbo
 |-com.alibaba.dubbo.rpc.ProxyFactory (纯文本文件，内容为: xxx=com.xxx.XxxProxyFactory)

XxxProxyFactory.java

package com.xxx;

import com.alibaba.dubbo.rpc.ProxyFactory;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.RpcException;

public class XxxProxyFactory implements ProxyFactory {
 public <T> T getProxy(Invoker<T> invoker) throws RpcException {
 // ...
 }
 public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) throws RpcException {
 // ...
 }
}

META-INF/dubbo/com.alibaba.dubbo.rpc.ProxyFactory

xxx=com.xxx.XxxProxyFactory

编译器扩展

(+)(#)

(1) 扩展说明:

Java代码编译器，用于动态生成字节码，加速调用。

(2) 扩展接口:

```
com.alibaba.dubbo.common.compiler.Compiler
```

(3) 扩展配置:

自动加载

(4) 已知扩展:

```
com.alibaba.dubbo.common.compiler.support.JdkCompiler
com.alibaba.dubbo.common.compiler.support.JavassistCompiler
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
        |-XxxCompiler.java (实现Compiler接口)
  |-resources
    |-META-INF
      |-dubbo
        |-com.alibaba.dubbo.common.compiler.Compiler (纯文本文件, 内容为: xxx=com.xxx.XxxCompiler)
```

XxxCompiler.java

```
package com.xxx;

import com.alibaba.dubbo.common.compiler.Compiler;

public class XxxCompiler implements Compiler {
    public Object getExtension(Class<?> type, String name) {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.common.compiler.Compiler

```
xxx=com.xxx.XxxCompiler
```

消息派发扩展

(+)(#)

(1) 扩展说明:

通道信息派发器, 用于指定线程池模型。

(2) 扩展接口:

```
com.alibaba.dubbo.remoting.Dispatcher
```

(3) 扩展配置:

```
<dubbo:protocol dispatcher="xxx" />
<dubbo:provider dispatcher="xxx" /> <!-- 缺省值设置, 当<dubbo:protocol>没有配置dispatcher属性时, 使用此配置 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.remoting.transport.dispatcher.all.AllDispatcher
com.alibaba.dubbo.remoting.transport.dispatcher.direct.DirectDispatcher
com.alibaba.dubbo.remoting.transport.dispatcher.message.MessageOnlyDispatcher
com.alibaba.dubbo.remoting.transport.dispatcher.execution.ExecutionDispatcher
com.alibaba.dubbo.remoting.transport.dispatcher.connection.ConnectionOrderedDispatcher
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
```

```
|-XxxDispatcher.java (实现Dispatcher接口)
|-resources
|  |-META-INF
|  |  |-dubbo
|  |  |  |-com.alibaba.dubbo.remoting.Dispatcher (纯文本文件, 内容为: xxx=com.xxx.XxxDispatcher)
```

XxxDispatcher.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.Dispatcher;

public class XxxDispatcher implements Dispatcher {
    public Group lookup(URL url) {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.remoting.Dispatcher

```
xxx=com.xxx.XxxDispatcher
```

线程池扩展

(+) (#)

(1) 扩展说明:

服务提供方线程实现策略, 当服务器收到一个请求时, 需要在线程池中创建一个线程去执行服务提供方业务逻辑。

(2) 扩展接口:

```
com.alibaba.dubbo.common.threadpool.ThreadPool
```

(3) 扩展配置:

```
<dubbo:protocol threadpool="xxx" />
<dubbo:provider threadpool="xxx" /> <!-- 缺省值设置, 当<dubbo:protocol>没有配置threadpool时, 使用此配置 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.common.threadpool.FixedThreadPool
com.alibaba.dubbo.common.threadpool.CachedThreadPool
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|  |-java
|  |  |-com
|  |  |  |-xxx
|  |  |  |  |-XxxThreadPool.java (实现ThreadPool接口)
|  |-resources
|  |  |-META-INF
|  |  |  |-dubbo
|  |  |  |  |-com.alibaba.dubbo.common.threadpool.ThreadPool (纯文本文件, 内容为: xxx=com.xxx.XxxThreadPool)
```

XxxThreadPool.java

```
package com.xxx;

import com.alibaba.dubbo.common.threadpool.ThreadPool;
import java.util.concurrent.Executor;

public class XxxThreadPool implements ThreadPool {
    public Executor getExecutor() {
        // ...
    }
}
```

```
}
```

META-INF/dubbo/com.alibaba.dubbo.common.threadpool.ThreadPool

```
xxx=com.xxx.XxxThreadPool
```

序列化扩展

(+)(#)

(1) 扩展说明：

将对象转成字节流，用于网络传输，以及将字节流转为对象，用于在收到字节流数据后还原成对象。

(2) 扩展接口：

```
com.alibaba.dubbo.common.serialize.Serialization  
com.alibaba.dubbo.common.serialize.ObjectInput  
com.alibaba.dubbo.common.serialize.ObjectOutput
```

(3) 扩展配置：

```
<dubbo:protocol serialization="xxx" /> <!-- 协议的序列化方式 -->  
<dubbo:provider serialization="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置serialization时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.common.serialize.dubbo.DubboSerialization  
com.alibaba.dubbo.common.serialize.hessian.Hessian2Serialization  
com.alibaba.dubbo.common.serialize.java.JavaSerialization  
com.alibaba.dubbo.common.serialize.java.CompactedJavaSerialization
```

(5) 扩展示例：

Maven项目结构

```
src  
|-main  
  |-java  
    |-com  
      |-xxx  
        |-XxxSerialization.java (实现Serialization接口)  
        |-XxxObjectInput.java (实现ObjectInput接口)  
        |-XxxObjectOutput.java (实现ObjectOutput接口)  
      |-resources  
        |-META-INF  
          |-dubbo  
            |-com.alibaba.dubbo.common.serialize.Serialization (纯文本文件，内容为：xxx=com.xxx.XxxSerialization)
```

XxxSerialization.java

```
package com.xxx;  
  
import com.alibaba.dubbo.common.serialize.Serialization;  
import com.alibaba.dubbo.common.serialize.ObjectInput;  
import com.alibaba.dubbo.common.serialize.ObjectOutput;  
  
public class XxxSerialization implements Serialization {  
    public ObjectOutput serialize(Parameters parameters, OutputStream output) throws IOException {  
        return new XxxObjectOutput(output);  
    }  
    public ObjectInput deserialize(Parameters parameters, InputStream input) throws IOException {  
        return new XxxObjectInput(input);  
    }  
}
```

META-INF/dubbo/com.alibaba.dubbo.common.serialize.Serialization

```
xxx=com.xxx.XxxSerialization
```


网络传输扩展

(+)(#)

(1) 扩展说明：

远程通讯的服务器及客户端传输实现。

(2) 扩展接口：

```
com.alibaba.dubbo.remoting.Transporter
com.alibaba.dubbo.remoting.Server
com.alibaba.dubbo.remoting.Client
```

(3) 扩展配置：

```
<dubbo:protocol transporter="xxx" /> <!-- 服务器和客户端使用相同的传输实现 -->
<dubbo:protocol server="xxx" client="xxx" /> <!-- 服务器和客户端使用不同的传输实现 -->
<dubbo:provider transporter="xxx" server="xxx" client="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置transporter/server/client属性时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.remoting.transport.transporter.netty.NettyTransporter
com.alibaba.dubbo.remoting.transport.transporter.mina.MinaTransporter
com.alibaba.dubbo.remoting.transport.transporter.grizzly.GrizzlyTransporter
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
|  |-java
|     |-com
|        |-xxx
|           |-XxxTransporter.java (实现Transporter接口)
|           |-XxxServer.java (实现Server接口)
|           |-XxxClient.java (实现Client接口)
|  |-resources
|     |-META-INF
|        |-dubbo
|           |-com.alibaba.dubbo.remoting.Transporter (纯文本文件，内容为：xxx=com.xxx.XxxTransporter)
```

XxxTransporter.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.Transporter;

public class XxxTransporter implements Transporter {
    public Server bind(URL url, ChannelHandler handler) throws RemotingException {
        return new XxxServer(url, handler);
    }
    public Client connect(URL url, ChannelHandler handler) throws RemotingException {
        return new XxxClient(url, handler);
    }
}
```

XxxServer.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.transport.transporter.AbstractServer;

public class XxxServer extends AbstractServer {
    public XxxServer(URL url, ChannelHandler handler) throws RemotingException{
        super(url, handler);
    }
    protected void doOpen() throws Throwable {
        // ...
    }
    protected void doClose() throws Throwable {
```

```

    // ...
}
public Collection<Channel> getChannels() {
    // ...
}
public Channel getChannel(InetSocketAddress remoteAddress) {
    // ...
}
}
}

```

XxxClient.java

```

package com.xxx;

import com.alibaba.dubbo.remoting.transport.transporter.AbstractClient;

public class XxxClient extends AbstractClient {
    public XxxServer(URL url, ChannelHandler handler) throws RemotingException{
        super(url, handler);
    }
    protected void doOpen() throws Throwable {
        // ...
    }
    protected void doClose() throws Throwable {
        // ...
    }
    protected void doConnect() throws Throwable {
        // ...
    }
    public Channel getChannel() {
        // ...
    }
}

```

META-INF/dubbo/com.alibaba.dubbo.remoting.Transporter

```
xxx=com.xxx.XxxTransporter
```

信息交换扩展

(+) (#)

(1) 扩展说明：

基于传输层之上，实现Request-Response信息交换语义。

(2) 扩展接口：

```

com.alibaba.dubbo.remoting.exchange.Exchanger
com.alibaba.dubbo.remoting.exchange.ExchangeServer
com.alibaba.dubbo.remoting.exchange.ExchangeClient

```

(3) 扩展配置：

```

<dubbo:protocol exchanger="xxx" />
<dubbo:provider exchanger="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置exchanger属性时，使用此配置 -->

```

(4) 已知扩展：

```
com.alibaba.dubbo.remoting.exchange.exchanger.HeaderExchanger
```

(5) 扩展示例：

Maven项目结构

```

src
|-main
  |-java
    |-com
      |-xxx
        |-XxxExchanger.java (实现Exchanger接口)
        |-XxxExchangeServer.java (实现ExchangeServer接口)
        |-XxxExchangeClient.java (实现ExchangeClient接口)
  |-resources

```

```
| -META-INF
|   |-dubbo
|     |-com.alibaba.dubbo.remoting.exchange.Exchanger (纯文本文件，内容为: xxx=com.xxx.XxxExchanger)
```

XxxExchanger.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.exchange.Exchanger;

public class XxxExchanger implements Exchanger {
    public ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException {
        return new XxxExchangeServer(url, handler);
    }
    public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {
        return new XxxExchangeClient(url, handler);
    }
}
```

XxxExchangeServer.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.exchange.ExchangeServer;

public class XxxExchangeServer implements ExchangeServer {
    // ...
}
```

XxxExchangeClient.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.exchange.ExchangeClient;

public class XxxExchangeClient implements ExchangeClient {
    // ...
}
```

META-INF/dubbo/com.alibaba.dubbo.remoting.exchange.Exchanger

```
xxx=com.xxx.XxxExchanger
```

组网扩展

(+) (#)

(1) 扩展说明：

对等网络节点组网器。

(2) 扩展接口：

```
com.alibaba.dubbo.remoting.p2p.Networker
```

(3) 扩展配置：

```
<dubbo:protocol networker="xxx" />
<dubbo:provider networker="xxx" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置networker属性时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.remoting.p2p.support.MulticastNetworker
com.alibaba.dubbo.remoting.p2p.support.FileNetworker
```

(5) 扩展示例：

Maven项目结构

```
src
|-main
|-java
|-com
|-xxx
|-XxxNetworker.java (实现Networker接口)
|-resources
|-META-INF
|-dubbo
|-com.alibaba.dubbo.remoting.p2p.Networker (纯文本文件, 内容为: xxx=com.xxx.XxxNetworker)
```

XxxNetworker.java

```
package com.xxx;

import com.alibaba.dubbo.remoting.p2p.Networker;

public class XxxNetworker implements Networker {
    public Group lookup(URL url) {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.remoting.p2p.Networker

```
xxx=com.xxx.XxxNetworker
```

Telnet命令扩展

(+) (#)

(1) 扩展说明:

所有服务器均支持telnet访问, 用于人工干预。

(2) 扩展接口:

```
com.alibaba.dubbo.remoting.telnet.TelnetHandler
```

(3) 扩展配置:

```
<dubbo:protocol telnet="xxx,yyy" />
<dubbo:provider telnet="xxx,yyy" /> <!-- 缺省值设置, 当<dubbo:protocol>没有配置telnet属性时, 使用此配置 -->
```

(4) 已知扩展:

```
com.alibaba.dubbo.remoting.telnet.support.ClearTelnetHandler
com.alibaba.dubbo.remoting.telnet.support.ExitTelnetHandler
com.alibaba.dubbo.remoting.telnet.support.HelpTelnetHandler
com.alibaba.dubbo.remoting.telnet.support.StatusTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.ListTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.ChangeTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.CurrentTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.InvokeTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.TraceTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.CountTelnetHandler
com.alibaba.dubbo.rpc.dubbo.telnet.PortTelnetHandler
```

(5) 扩展示例:

Maven项目结构

```
src
|-main
|-java
|-com
|-xxx
|-XxxTelnetHandler.java (实现TelnetHandler接口)
|-resources
|-META-INF
|-dubbo
|-com.alibaba.dubbo.remoting.telnet.TelnetHandler (纯文本文件, 内容为: xxx=com.xxx.XxxTelnetHandler)
```

XxxTelnetHandler.java
<pre>package com.xxx; import com.alibaba.dubbo.remoting.telnet.TelnetHandler; @Help(parameter="...", summary="...", detail="...") public class XxxTelnetHandler implements TelnetHandler { public String telnet(Channel channel, String message) throws RemotingException { // ... } }</pre>
META-INF/dubbo/com.alibaba.dubbo.remoting.telnet.TelnetHandler
<pre>xxx=com.xxx.XxxTelnetHandler</pre>
用法
<pre>telnet 127.0.0.1 20880 dubbo> xxx args</pre>

状态检查扩展

(+) (#)

(1) 扩展说明：

检查服务依赖各种资源的状态，此状态检查可同时用于telnet的status命令和hosting的status页面。

(2) 扩展接口：

<pre>com.alibaba.dubbo.common.status.StatusChecker</pre>
--

(3) 扩展配置：

<pre><dubbo:protocol status="xxx,yyy" /> <dubbo:provider status="xxx,yyy" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置status属性时，使用此配置 --></pre>
--

(4) 已知扩展：

<pre>com.alibaba.dubbo.common.status.support.MemoryStatusChecker com.alibaba.dubbo.common.status.support.LoadStatusChecker com.alibaba.dubbo.rpc.dubbo.status.ServerStatusChecker com.alibaba.dubbo.rpc.dubbo.status.ThreadPoolStatusChecker com.alibaba.dubbo.registry.directory.RegistryStatusChecker com.alibaba.dubbo.rpc.config.spring.status.SpringStatusChecker com.alibaba.dubbo.rpc.config.spring.status.DataSourceStatusChecker</pre>

(5) 扩展示例：

Maven项目结构
<pre>src -main -java -com -xxx -XxxStatusChecker.java（实现StatusChecker接口） -resources -META-INF -dubbo -com.alibaba.dubbo.common.status.StatusChecker（纯文本文件，内容为：xxx=com.xxx.XxxStatusChecker）</pre>
XxxStatusChecker.java
<pre>package com.xxx;</pre>

```
import com.alibaba.dubbo.common.status.StatusChecker;

public class XxxStatusChecker implements StatusChecker {
    public Status check() {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.common.status.StatusChecker

xxx=com.xxx.XxxStatusChecker

容器扩展

(+) (#)

(1) 扩展说明：

服务容器扩展，用于自定义加载内容。

(2) 扩展接口：

com.alibaba.dubbo.container.Container

(3) 扩展配置：

java com.alibaba.dubbo.container.Main spring jetty log4j

(4) 已知扩展：

com.alibaba.dubbo.container.spring.SpringContainer
com.alibaba.dubbo.container.spring.JettyContainer
com.alibaba.dubbo.container.spring.Log4jContainer

(5) 扩展示例：

Maven项目结构

```
src
|-main
  |-java
    |-com
      |-xxx
        |-XxxContainer.java (实现Container接口)
  |-resources
    |-META-INF
      |-dubbo
        |-com.alibaba.dubbo.container.Container (纯文本文件，内容为：xxx=com.xxx.XxxContainer)
```

XxxContainer.java

```
package com.xxx;

com.alibaba.dubbo.container.Container;

public class XxxContainer implements Container {
    public Status start() {
        // ...
    }
    public Status stop() {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.container.Container

xxx=com.xxx.XxxContainer

页面扩展

(+) (#)

(1) 扩展说明：

对等网络节点组网器。

(2) 扩展接口：

com.alibaba.dubbo.container.page.PageHandler

(3) 扩展配置：

<dubbo:protocol page="xxx,yyy" />
<dubbo:provider page="xxx,yyy" /> <!-- 缺省值设置，当<dubbo:protocol>没有配置page属性时，使用此配置 -->

(4) 已知扩展：

com.alibaba.dubbo.container.page.pages.HomePageHandler
com.alibaba.dubbo.container.page.pages.StatusPageHandler
com.alibaba.dubbo.container.page.pages.LogPageHandler
com.alibaba.dubbo.container.page.pages.SystemPageHandler

(5) 扩展示例：

Maven项目结构

src
|-main
 |-java
 |-com
 |-xxx
 |-XxxPageHandler.java（实现PageHandler接口）
 |-resources
 |-META-INF
 |-dubbo
 |-com.alibaba.dubbo.container.page.PageHandler（纯文本文件，内容为：xxx=com.xxx.XxxPageHandler）

XxxPageHandler.java

package com.xxx;

import com.alibaba.dubbo.container.page.PageHandler;

public class XxxPageHandler implements PageHandler {
 public Group lookup(URL url) {
 // ...
 }
}

META-INF/dubbo/com.alibaba.dubbo.container.page.PageHandler

xxx=com.xxx.XxxPageHandler

缓存扩展

(+) (#)

(1) 扩展说明：

用请求参数作为key，缓存返回结果。

(2) 扩展接口：

com.alibaba.dubbo.cache.CacheFactory

(3) 扩展配置：

```
<dubbo:service cache="lru" />
<dubbo:service><dubbo:method cache="lru" /></dubbo:service> <!-- 方法级缓存 -->
<dubbo:provider cache="xxx,yyy" /> <!-- 缺省值设置，当<dubbo:service>没有配置cache属性时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.cache.support.lru.LruCacheFactory
com.alibaba.dubbo.cache.support.threadlocal.ThreadLocalCacheFactory
com.alibaba.dubbo.cache.support.jcache.JCacheFactory
```

(5) 扩展示例：

Maven项目结构

src
|-main
|-java
|-com
|-xxx
|-XxxCacheFactory.java (实现StatusChecker接口)
|-resources
|-META-INF
|-dubbo
|-com.alibaba.dubbo.cache.CacheFactory (纯文本文件，内容为：xxx=com.xxx.XxxCacheFactory)

XxxCacheFactory.java

package com.xxx;

import com.alibaba.dubbo.cache.CacheFactory;

public class XxxCacheFactory implements CacheFactory {
 public Cache getCache(URL url, String name) {
 return new XxxCache(url, name);
 }
}

XxxCacheFactory.java

package com.xxx;

import com.alibaba.dubbo.cache.Cache;

public class XxxCache implements Cache {
 public Cache(URL url, String name) {
 // ...
 }
 public void put(Object key, Object value) {
 // ...
 }
 public Object get(Object key) {
 // ...
 }
}

META-INF/dubbo/com.alibaba.dubbo.cache.CacheFactory

xxx=com.xxx.XxxCacheFactory

验证扩展

(+) (#)

(1) 扩展说明：

参数验证扩展点。

(2) 扩展接口：

```
com.alibaba.dubbo.validation.Validation
```

(3) 扩展配置：

```
<dubbo:service validation="xxx,yyy" />
<dubbo:provider validation="xxx,yyy" /> <!-- 缺省值设置，当<dubbo:service>没有配置validation属性时，使用此配置 -->
```

(4) 已知扩展：

```
com.alibaba.dubbo.validation.support.jvalidation.JValidation
```

(5) 扩展示例：

Maven项目结构

src
|-main
|-java
|-com
|-xxx
|-XxxValidation.java (实现Validation接口)
|-resources
|-META-INF
|-dubbo
|-com.alibaba.dubbo.validation.Validation (纯文本文件，内容为：xxx=com.xxx.XxxValidation)

XxxValidation.java

package com.xxx;

import com.alibaba.dubbo.validation.Validation;

public class XxxValidation implements Validation {
 public Object getValidator(URL url) {
 // ...
 }
}

XxxValidator.java

package com.xxx;

import com.alibaba.dubbo.validation.Validator;

public class XxxValidator implements Validator {
 public XxxValidator(URL url) {
 // ...
 }
 public void validate(Invocation invocation) throws Exception {
 // ...
 }
}

META-INF/dubbo/com.alibaba.dubbo.validation.Validation

xxx=com.xxx.XxxValidation

日志适配扩展

(+) (#)

(1) 扩展说明：

日志输出适配扩展点。

(2) 扩展接口：

```
com.alibaba.dubbo.common.logger.LoggerAdapter
```

(3) 扩展配置：

```
<dubbo:application logger="xxx" />
```

```
-Ddubbo:application.logger=xxx
```

(4) 已知扩展:

```
slf4j=com.alibaba.dubbo.common.logger.slf4j.Slf4jLoggerAdapter  
jcl=com.alibaba.dubbo.common.logger.jcl.JclLoggerAdapter  
log4j=com.alibaba.dubbo.common.logger.log4j.Log4jLoggerAdapter  
jdk=com.alibaba.dubbo.common.logger.jdk.JdkLoggerAdapter
```

(5) 扩展示例:

Maven项目结构

```
src  
|-main  
  |-java  
    |-com  
      |-xxx  
        |-XxxLoggerAdapter.java (实现LoggerAdapter接口)  
  |-resources  
    |-META-INF  
      |-dubbo  
        |-com.alibaba.dubbo.common.logger.LoggerAdapter (纯文本文件, 内容为: xxx=com.xxx.XxxLoggerAdapter)
```

XxxLoggerAdapter.java

```
package com.xxx;  
  
import com.alibaba.dubbo.common.logger.LoggerAdapter;  
  
public class XxxLoggerAdapter implements LoggerAdapter {  
    public Logger getLogger(URL url) {  
        // ...  
    }  
}
```

XxxLogger.java

```
package com.xxx;  
  
import com.alibaba.dubbo.common.logger.Logger;  
  
public class XxxLogger implements Logger {  
    public XxxLogger(URL url) {  
        // ...  
    }  
    public void info(String msg) {  
        // ...  
    }  
    // ...  
}
```

META-INF/dubbo/com.alibaba.dubbo.common.logger.LoggerAdapter

```
xxx=com.xxx.XxxLoggerAdapter
```

技术兼容性测试

(+)(#)

TCK定义: http://en.wikipedia.org/wiki/Technology_Compatibility_Kit

Dubbo的协议, 通讯, 序列化, 注册中心, 负载均衡等扩展点, 都有多种可选策略, 以应对不同应用场景, 而我们的测试用例很分散, 当用户自己需要加一种新的实现时, 总是不确定能否满足扩展点的完整契约。

所以, 我们需要对核心扩展点写TCK (Technology Compatibility Kit), 用户增加一种扩展实现, 只需通过TCK, 即可确保与框架的其它部分兼容运行, 可以有效提高整体健壮性, 也方便第三方扩展者接入, 加速开源社区的成熟。

开源社区的行知同学已着手研究这一块, 他的初步想法是借鉴JBoss的CDI-TCK, 做一个Dubbo的TCK基础框架, 在此之上实现Dubbo的扩展点TCK用例。

参见:

<http://docs.jboss.org/cdi/tck/reference/1.0.1-Final/html/introduction.html>

如果大家有兴趣, 也可以一起研究, 和行知一块讨论。

Protocol TCK

(+) (#)

Registry TCK

(+) (#)

公共契约

(+) (#)

✔ 这里记录的是Dubbo公共契约，希望所有扩展点遵守。

URL

- 所有扩展点参数都包含URL参数，URL作为上下文信息贯穿整个扩展点设计体系。
- URL采用标准格式：protocol://username:password@host:port/path?key=value&key=value

日志

- 如果不可恢复或需要报警，打印ERROR日志。
- 如果可恢复异常，或瞬时的状态不一致，打印WARN日志。
- 正常运行时的中间状态提示，打印INFO日志。

坏味道

(+) (#)

⚠ 这里记录的是Dubbo设计或实现不优雅的地方。

URL转换

1. 点对点暴露和引用服务

1.1. 直接暴露服务：

EXPORT(dubbo://provider-address/com.xxx.XxxService?version=1.0.0)

1.2. 点对点直连服务：

REFER(dubbo://provider-address/com.xxx.XxxService?version=1.0.0)

2. 通过注册中心暴露服务

2.1. 向注册中心暴露服务：

EXPORT(registry://registry-address/com.alibaba.dubbo.registry.RegistryService?registry=dubbo&export=ENCODE(dubbo://provider-address/com.xxx.XxxService?version=1.0.0))

2.2. 获取注册中心：url.setProtocol(url.getParameter("registry", "dubbo"))

GETREGISTRY(dubbo://registry-address/com.alibaba.dubbo.registry.RegistryService)

2.3. 注册服务地址：url.getParameterAndDecoded("export")

REGISTER(dubbo://provider-address/com.xxx.XxxService?version=1.0.0)

3. 通过注册中心引用服务

3.1. 从注册中心订阅服务：

REFER(registry://registry-address/com.alibaba.dubbo.registry.RegistryService?registry=dubbo&refer=ENCODE(version=1.0.0))

3.2. 获取注册中心：url.setProtocol(url.getParameter("registry", "dubbo"))

GETREGISTRY(dubbo://registry-address/com.alibaba.dubbo.registry.RegistryService)

3.3. 订阅服务地址：url.addParameters(url.getParameterAndDecoded("refer"))

SUBSCRIBE(dubbo://registry-address/com.xxx.XxxService?version=1.0.0)

3.4. 通知服务地址：url.addParameters(url.getParameterAndDecoded("refer"))

NOTIFY(dubbo://provider-address/com.xxx.XxxService?version=1.0.0)

4. 注册中心推送路由规则

4.1. 注册中心路由规则推送：

NOTIFY(route://registry-address/com.xxx.XxxService?router=script&type=js&rule=ENCODE(function{...}))

4.2. 获取路由器：url.setProtocol(url.getParameter("router", "script"))

GETROUTE(script://registry-address/com.xxx.XxxService?type=js&rule=ENCODE(function{...}))

5. 从文件加载路由规则

5.1. 从文件加载路由规则：

GETROUTE(file://path/file.js?router=script)

5.2. 获取路由器: `url.setProtocol(url.getParameter("router", "script")).addParameter("type", SUFFIX(file)).addParameter("rule", READ(file))`
`GETROUTE(script://path/file.js?type=js&rule=ENCODE(function{...}))`

调用参数

- path 服务路径
- group 服务分组
- version 服务版本
- dubbo 使用的dubbo版本
- token 验证令牌
- timeout 调用超时

扩展点的加载

1. 自适应扩展点

ExtensionLoader加载扩展点时，会检查扩展点的属性（通过set方法判断），如该属性是扩展点类型，则会注入扩展点对象。因为注入时不能确定使用哪个扩展点（在使用时确定），所以注入的是一个自适应扩展（一个代理）。自适应扩展点调用时，选取一个真正的扩展点，并代理到其上完成调用。Dubbo是根据调用方法参数（上面有调用哪个扩展点的信息）来选取一个真正的扩展点。

在Dubbo给定所有的扩展点上调用都有URL参数（整个扩展点网的上下文信息）。自适应扩展即是从URL确定要调用哪个扩展点实现。URL哪个Key的Value用来确定使用哪个扩展点，这个信息通过的@Adaptive注解在方法上说明。

```
@Extension
public interface Car {
    @Adaptive({"http://10.20.160.198/wiki/display/dubbo/car.type", "http://10.20.160.198/wiki/display/dubbo/transport.type"})
    public run(URL url, Type1 arg1, Type2 arg2);
}
```

由于自适应扩展点的上面的约定，ExtensionLoader会为扩展点自动生成自适应扩展点类(通过字节码)，并将其实例注入。

ExtensionLoader生成的自适应扩展点类如下：

```
package <扩展点接口所在包>;

public class <扩展点接口名>$Adaptive implements <扩展点接口> {
    public <有@Adaptive注解的接口方法>(<方法参数>) {
        if(是否有URL类型方法参数?) 使用该URL参数
        else if(是否有方法类型上有URL属性) 使用该URL属性
        # <else 在加载扩展点生成自适应扩展点类时抛异常，即加载扩展点失败! >

        if(获取的URL == null) {
            throw new IllegalArgumentException("url == null");
        }

        根据@Adaptive注解上声明的Key的顺序，从URL获致Value，作为实际扩展点名。
        如URL没有Value，则使用缺省扩展点实现。如没有扩展点， throw new IllegalStateException("Fail to get extension");

        在扩展点实现调用该方法，并返回结果。
    }

    public <有@Adaptive注解的接口方法>(<方法参数>) {
        throw new UnsupportedOperationException("is not adaptive method!");
    }
}
```

@Adaptive注解使用如下：

如果URL这些Key都没有Value，使用 用 缺省的扩展（在接口的Default中设定的值）。
比如，String[] {"key1", "key2"}，表示

先在URL上找key1的Value作为要Adapt成的Extension名；

key1没有Value，则使用key2的Value作为要Adapt成的Extension名。

key2没有Value，使用缺省的扩展。

如果没有设定缺省扩展，则方法调用会抛出IllegalStateException。

如果不设置则缺省使用Extension接口类名的点分隔小写字串。

即对于Extension接口com.alibaba.dubbo.xxx.YyyInvokerWrapper的缺省值为String[] {"yyy.invoker.wrapper"}

Callback功能

1. 参数回调

1.1主要原理:在一个Consumer->provider的长连接上，自动在Consumer端暴露一个服务（实现方法参数上声明的接口A）， provider端便可反向调用到consumer端的接口实例。

1.2实现细节：

- 为了在传输时能够对回调接口实例进行转换，自动暴露与自动引用目前在DubboCodec中实现.此处需要考虑将此逻辑与codec逻辑分离。
- 在根据Invocation信息获取exporter时，需要判断是否是回调，如果是回调，会从attachments中取得回调服务实例的id，在获取exporter，此处用于consumer端可以对同一个callback接口做不同的实现。

2. 事件通知

2.1主要原理: Consumer在invoke方法时，判断如果有配置onreturn/onerror...则将onreturn对应的参数值(实例方法)加入到异步调用的回调列表中。

2.2实现细节: 参数的传递采用URL，但URL中没有支持string-object，所以将实例方法存储在staticMap中，此处实现需要进行改

造, <http://code.alibabatech.com/jira/browse/DUBBO-168>

Lazy连接

DubboProtocol特有功能, 默认关闭

当客户端与服务端创建代理时, 暂不建立tcp长连接, 当有数据请求时在做连接初始化

此项功能自动关闭连接重试功能, 开启发送重试功能(即发送数据时如果连接已断开, 尝试重新建立连接)。

共享连接

DubboProtocol特有功能, 默认开启

JVM A暴露了多个服务, JVM B引用了A中的多个服务, 共享连接是说A与B多个服务调用是通过同一个TCP长连接进行数据传输, 已达到减少服务端连接数的目的。

实现细节: 对于同一个地址由于使用了共享连接, 那invoker的destroy就需要特别注意, 一方面要满足对同一个地址refer的invoker全部destroy后, 连接需要关闭, 另一方面还需要注意如何避免部分invoker destroy时不能关闭连接。在实现中采用了引用计数的方案, 但为了防范, 在连接关闭时, 重新建立了一个Lazy connection(称为幽灵连接), 用于当出现异常场景时, 避免影响业务逻辑的正常调用。

sticky 策略

有多个服务提供者的情况下, 配置了sticky后, 在提供者可用的情况下, 调用会继续发送到上一次的服务提供者。sticky策略默认开启了连接的lazy选项, 用于避免开启无用的连接。

服务提供者选择逻辑

1. 存在多个服务提供者的情况下, 首先根据Loadbalance进行选择, 如果选择的provider处于可用状态, 则进行后续调用
2. 如果第一步选择的服务提供者不可用, 则从剩余服务提供者列表中继续选择, 如果可用, 进行后续调用
3. 如果所有的服务提供者都不可用, 重新遍历整个列表(优先从没有选过的列表中选择), 判断是否有可用的服务提供者(选择过程中, 不可用的服务提供者可能会恢复到可用状态), 如果有, 则进行后续调用
4. 如果第三步没有选择出可用的服务提供者, 会选第一步选出的invoker中的下一个(如果不是最后一个), 避免碰撞。

编码约定

(+)(#)

Source code and documentation contributed to the Dubbo repositories should observe the:

- [Code Conventions for the Java Programming Language](#)
- [How to Write Doc Comments for the Javadoc Tool](#)

as core references regarding the formatting of code and documentation.

- 异常和日志:
 - 尽可能携带完整的上下文信息, 比如出错原因, 出错的机器地址, 调用对方的地址, 连的注册中心地址, 使用Dubbo的版本等。
 - 尽量将直接原因写在最前面, 所有上下文信息, 在原因后用键值对显示。
 - 抛出异常的地方不用打印日志, 由最终处理异常者决定打印日志的级别, 吃掉异常必需打印日志。
 - 打印ERROR日志表示需要报警, 打印WARN日志表示可以自动恢复, 打印INFO表示正常信息或完全不影响运行。
 - 建议应用方在监控中心配置ERROR日志实时报警, WARN日志每周汇总发送通知。
 - RpcException是Dubbo对外的唯一异常类型, 所有内部异常, 如果要抛出给用户, 必须转为RpcException。
 - RpcException不能有子类型, 所有类型信息用ErrorCode标识, 以便保持兼容。
- 配置和URL:
 - 配置对象属性首字母小写, 多个单词用驼峰命名(Java约定)。
 - 配置属性全部用小写, 多个单词用"-"号分隔(Spring约定)。
 - URL参数全部用小写, 多个单词用"."号分隔(Dubbo约定)。
 - 尽可能用URL传参, 不要自定义Map或其它上下文格式, 配置信息也转成URL格式使用。
 - 尽量减少URL嵌套, 保持URL的简洁性。
- 单元和集成测试:
 - 单元测试统一用JUnit和EasyMock, 集成测试用TestNG, 数据库测试用DBUnit。
 - 保持单元测试用例的运行速度, 不要将性能和大的集成用例放在单元测试中。
 - 保持单元测试的每个用例都用try...finally或tearDown释放资源。
 - 减少while循环等待结果的测试用例, 对定时器和网络的测试, 用以将定时器中的逻辑抽为方法测试。
 - 对于容错行为的测试, 比如failsafe的测试, 统一用LogUtil断言日志输出。
- 扩展点基类与AOP:
 - AOP类都命名为XxxWrapper, 基类都命名为AbstractXxx。
 - 扩展点之间的组合将关系由AOP完成, ExtensionLoader只负责加载扩展点, 包括AOP扩展。
 - 尽量采用IoC注入扩展点之间的依赖, 不要直接依赖ExtensionLoader的工厂方法。
 - 尽量采用AOP实现扩展点的通用行为, 而不要用基类, 比如负载均衡之前的isAvailable检查, 它是独立于负载均衡之外的, 不需要检查的是URL参数关闭。
 - 对多种相似类型的抽象, 用基类实现, 比如RMI, Hessian等第三方协议都已生成了接口代理, 只需将将接口代理转成Invoker即可完成桥接, 它们可以用公共基类实现此逻辑。
 - 基类也是SPI的一部分, 每个扩展点都应该有方便使用的基类支持。
- 模块与分包:
 - 基于复用度分包, 总是一起使用的放在同一包下, 将接口和基类分成独立模块, 大的实现也使用独立模块。
 - 所有接口都放在模块的根包下, 基类放在support子包下, 不同实现用放在以扩展点名字命名的子包下。
 - 尽量保持子包依赖父包, 而不要反向。

检查列表

(+)(#)

1. 发布前checklist:
 - 1.1 jira ticket 过一遍
 - 1.2 svn change list
 - 1.3 ticket关联code
 - 1.4 test code
 - 1.5 find bugs

2. 修复时checklist:


2.1 修复代码前先建ticket2.2 修复代码前先写测试用例2.3 需要伙伴检查2.4 test code(正常流程/异常流程)2.5 讲一遍逻辑2.6 契约文档化2.7 以上内容都写到ticket的评论上2.8 代码注释写清楚，用中文无妨2.9 每个版本要有owner，确保scope和check
3. Partner Check


3.1 Partner以用户的方式运行一下功能3.2 Partner发现问题、添加测试（集成测试）复现总是：Owner完成实现。（保证两方的时间投入PatternerCheck 的给予时间保证）3.3 Owner向Partner讲述一遍实现。


设计原则


- (+) (#)
- 魔鬼在细节中
 - 一些设计上的基本常识
 - 谈谈扩充式扩展与增量式扩展
 - 配置设计
 - 设计实现的健壮性
 - 防痴呆设计
 - 扩展点重构


Labels: [dubbo](#) [developer](#) [guide](#) [zh](#)


- ▼ 13 Child Pages
-  Bad Smell-zh


 Check List-zh


 Code Conventions-zh


 Common Contract-zh


 Design Principles-zh


 Extension Loader-zh


 Framework Design-zh


 Getting Involved-zh

 Implementation Detail-zh

 Source Building-zh

 SPI Reference-zh

 Technology Compatibility Kit-zh

 Version Management-zh

11 Comments ▼

- 

Anonymous

十月 18, 2012

请问这玩意怎么从0开始起起一个HelloWorld
- 

Anonymous

二月 14, 2013

Son of a gun, this is so hleupfl!
- 

Anonymous

二月 16, 2013

I can already tell that's gonna be super heupfl.
- 

Anonymous

二月 19, 2013

g3hS39 zbynxenqweay
- 

Anonymous

一月 22, 2013

怎么没有注册功能了？
- 

Anonymous

四月 22, 2013

想问下管理平台保存的路由信息保存在哪里？我重启了zookeeper和管理平台后，还是出现，而且无法失效和删除，需要删除缓存文件，想问下保存在哪？如何删除，谢谢
- Anonymous



六月 22, 2013

What the hell is this ? So complicated.



Anonymous

七月 29, 2013

大家注意哈，自己下载源代码编译的时候，最好是maven版本是2.x的，我使用maven3.0.3出现：
Could not find metadata org.apache.maven.plugins/maven-metadata.xml in opensesame.snapshots.plugin (<http://code.alibabatech.com/mvn/snapshots>)
[DEBUG] Could not find metadata org.apache.maven.plugins/maven-metadata.xml in opensesame.releases.plugin (<http://code.alibabatech.com/mvn/releases>)
[DEBUG] Could not find metadata org.codehaus.mojo/maven-metadata.xml in opensesame.snapshots.plugin (<http://code.alibabatech.com/mvn/snapshots>)
异常：
org.apache.maven.plugin.prefix.NoPluginFoundForPrefixException: No plugin found for prefix 'X' in the current project and in the plugin groups [org.apache.maven.plugins, org.codehaus.mojo] available from the repositories [local (/Users/kyle/.m2/repository), central.repo.plugin (<http://repo1.maven.org/maven2>), opensesame.snapshots.plugin (<http://code.alibabatech.com/mvn/snapshots>), opensesame.releases.plugin (<http://code.alibabatech.com/mvn/releases>), central (<http://repo1.maven.org/maven2>)]
这个问题，换成maven2.2.1编译，就OK了！



Anonymous

八月 01, 2013

好些jar包找不到啊~比如:hessian-lite 这个貌似是ali内部精简的一个包，在maven仓库找不到



Anonymous

八月 07, 2013

弱弱的问一下：ServiceConfig里面的
private void checkProtocol() {
if ((protocols == null || protocols.size() == 0)
&& provider != null)
Unknown macro: { setProtocols(provider.getProtocols()); }
// 兼容旧版本
if (protocols == null || protocols.size() == 0)
Unknown macro: { setProtocol(new ProtocolConfig()); }
for (ProtocolConfig protocolConfig : protocols) {
if (StringUtils.isEmpty(protocolConfig.getName()))
Unknown macro: { protocolConfig.setName("dubbo"); }
appendProperties(protocolConfig);
}
}中的那个"dubbo"会出错吗？这样设置以后好像最后不能使用dubbo.properties里面的全局设置了，是这样的？



Anonymous

九月 03, 2013

问下，Dubbo已经不更新了吗，我看到好多任务都没人认领？

Add Comment