Table of Contents

这篇文档的目标读者是对 dubbo 源码、设计有兴趣的，或者有意愿加入 dubbo 开发的人群。主要涵盖了 dubbo 的框架设计、扩展机制、编码规范、版本管理、构建等话题。

源码构建

代码签出

通过以下的这个命令签出最新的项目源码 [1]：

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

分支

我们使用 master 作为主干版本的开发，使用分支作为维护版本。可以通过 https://github.com/alibaba/dubbo/tags 来查看所有版本的标签。

构建

Dubbo 使用 maven 作为构建工具。

要求

- Java 1.5 以上的版本
- Maven 2.2.1 或者以上的版本

构建之前需要配置以下的 `MAVEN_OPTS`

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

使用以下命令做一次构建

```
mvn clean install
```

可以通过以下的构建命令来跳过单元测试

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

构建源代码 jar 包

通过以下命令以构建 Dubbo 的源代码 jar 包，方便用来调试 Dubbo 源代码

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

IDE 支持

使用以下命令来生成 IDE 的工程

Intellij Idea

```
mvn idea:idea
```

eclipse

```
mvn eclipse:eclipse
```

在 eclipse 中导入

首先，需要在 eclipse 中配置 maven 仓库。通过 Preferences -> Java -> Build Path -> Classpath 定义 `M2_REPO` 的 classpath 变量指向本地的 maven 仓库。[2]

也可以通过以下的 maven 命令配置：

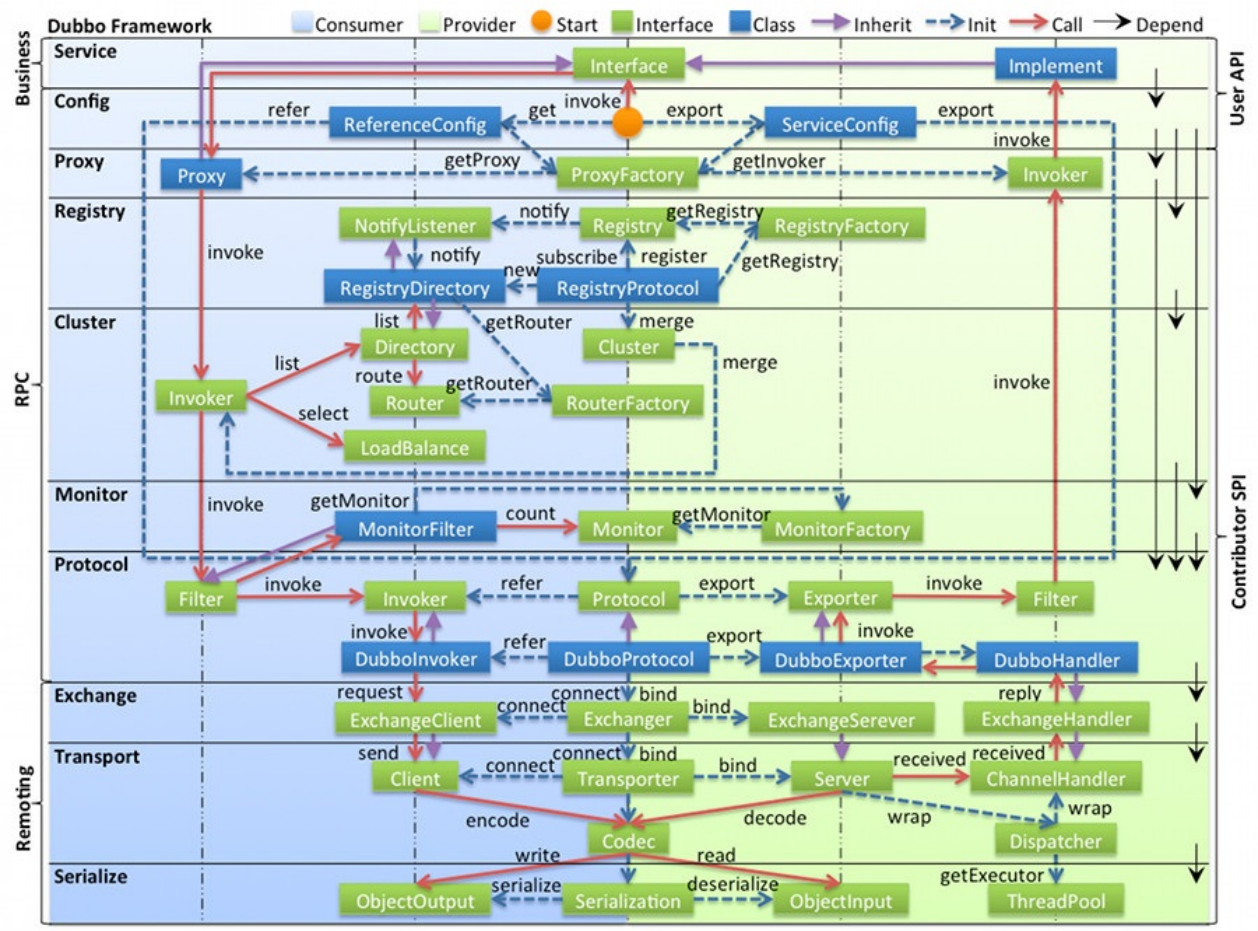
```
mvn eclipse:configure-workspace -Declipse.workspace=/path/to/the
/workspace/
```

> 1. 也可以直接在 https://github.com/alibaba/dubbo 上浏览源代码 ↩

> 2. UNIX 下的路径是 ${HOME}/.m2/repository, Windows 下的路径是 C:\Documents and Settings\.m2\repository ↩

框架设计

整体设计



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，即运行时调时链，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

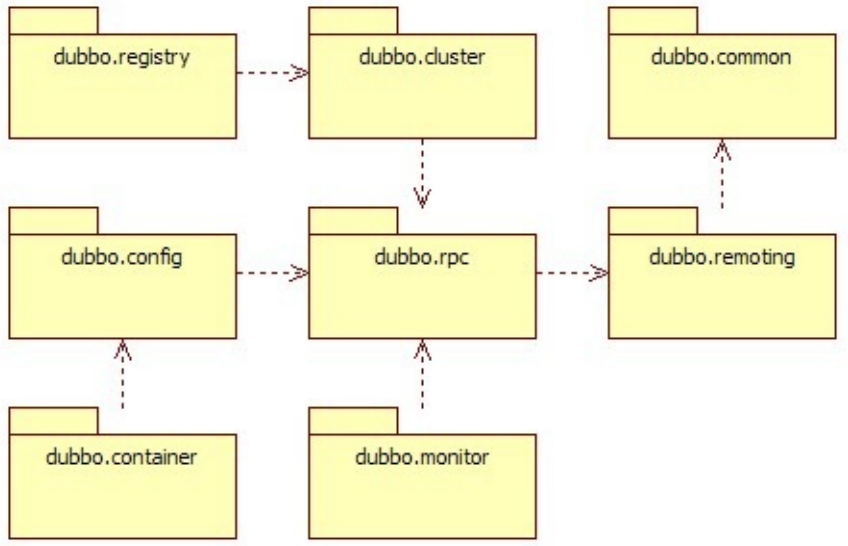
- **config** 配置层：对外配置接口，以 `ServiceConfig`，`ReferenceConfig` 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
- **proxy** 服务代理层：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 以 `ServiceProxy` 为中心，扩展接口为 `ProxyFactory`
- **registry** 注册中心层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 `RegistryFactory`，`Registry`，`RegistryService`
- **cluster** 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：封将 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，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，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：

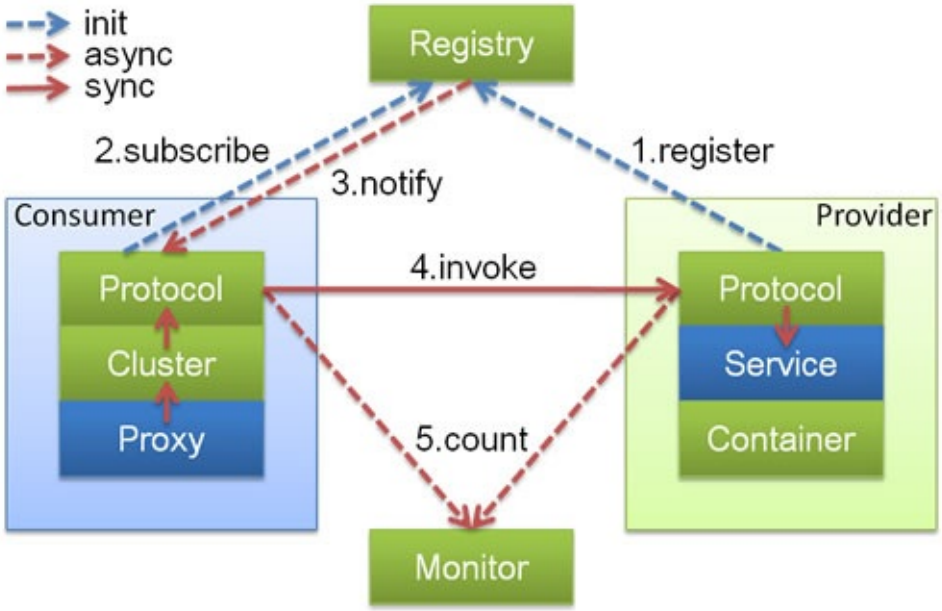
- **dubbo-common** 公共逻辑模块：包括 Util 类和通用模型。
- **dubbo-remoting** 远程通讯模块：相当于 Dubbo 协议的实现，如果 RPC 用 RMI协议则不需要使用此包。
- **dubbo-rpc** 远程调用模块：抽象各种协议，以及动态代理，只包含一对一的调用，不关心集群的管理。
- **dubbo-cluster** 集群模块：将多个服务提供方伪装为一个提供方，包括：负载均衡, 容错，路由等，集群的地址列表可以是静态配置的，也可以是由注册中心下发。

- **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 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，在不需要集群也就是只有一个提供者时，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



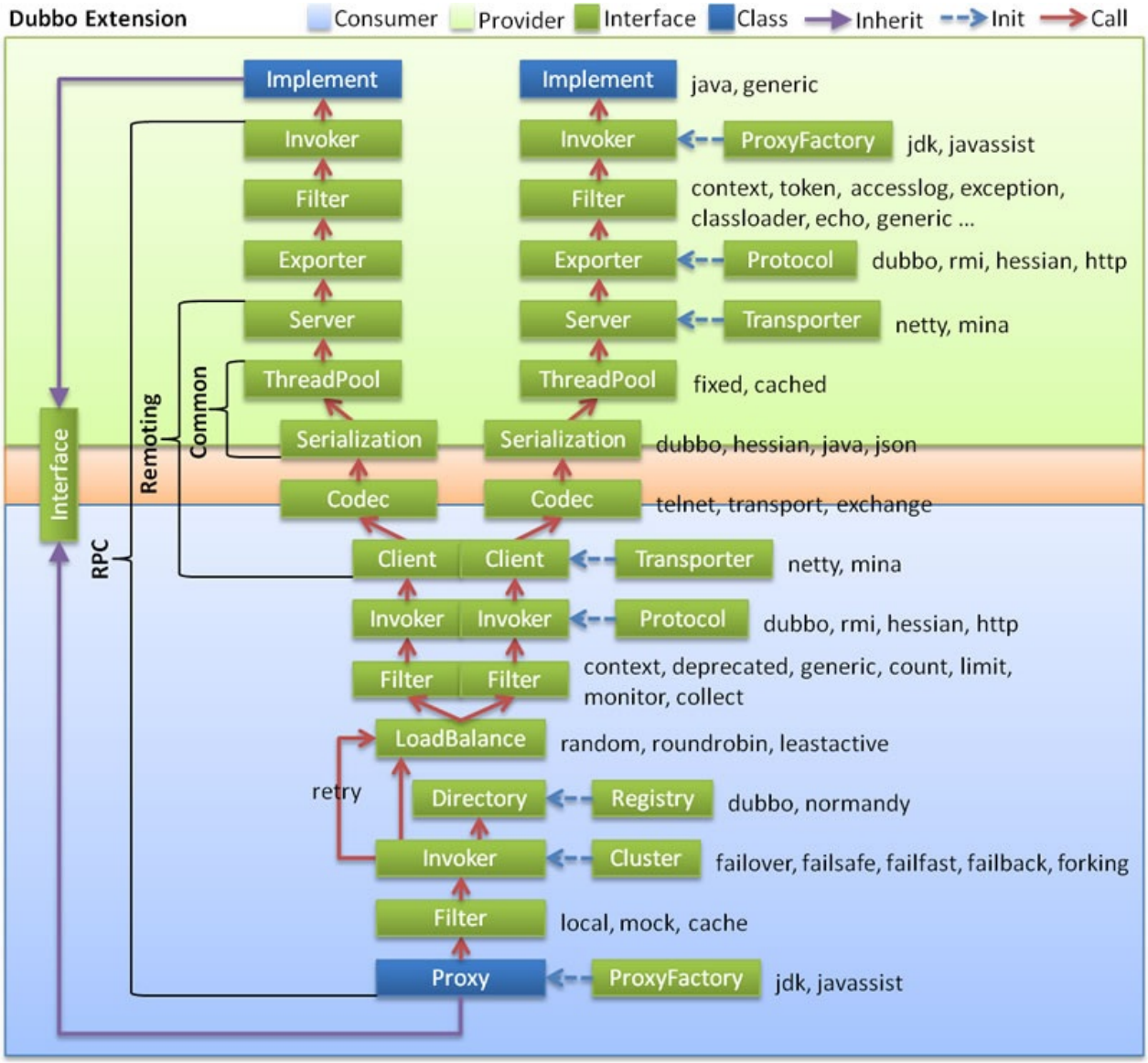
图例说明：

- 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 代表部署逻辑拓扑节点。
- 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。

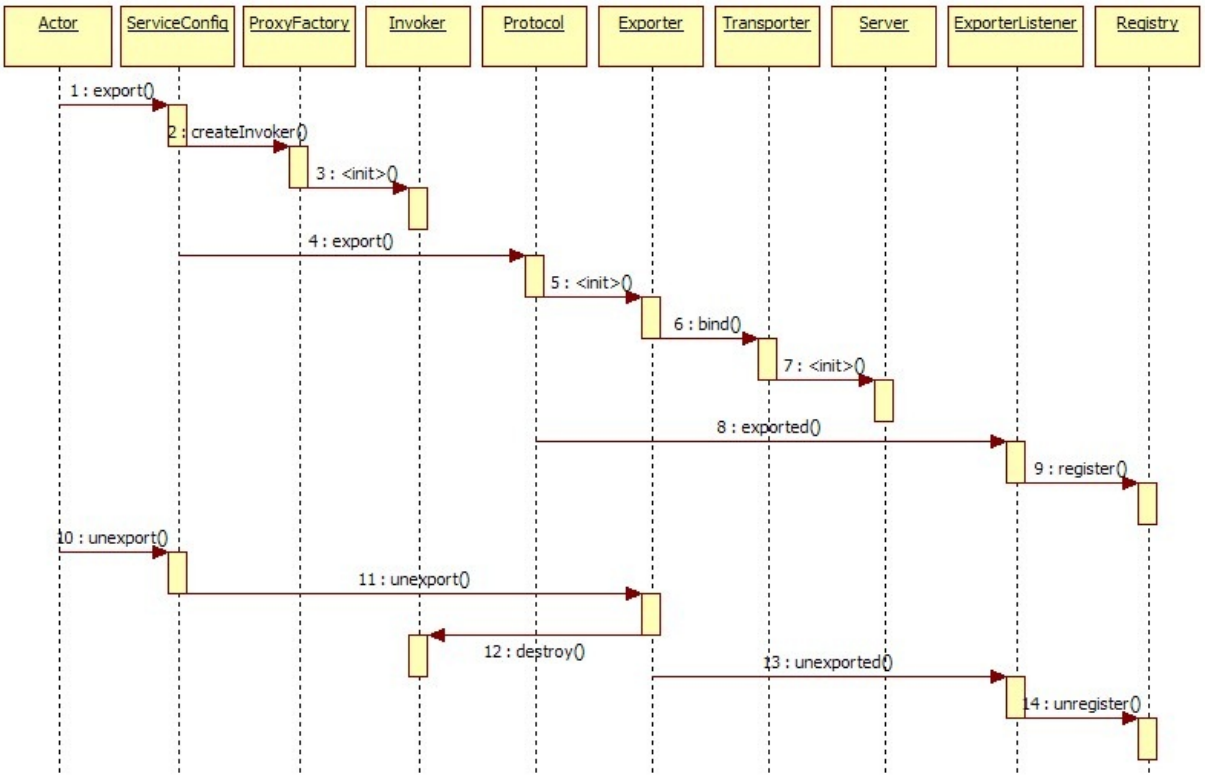
调用链

展开总设计图的红色调用链，如下：



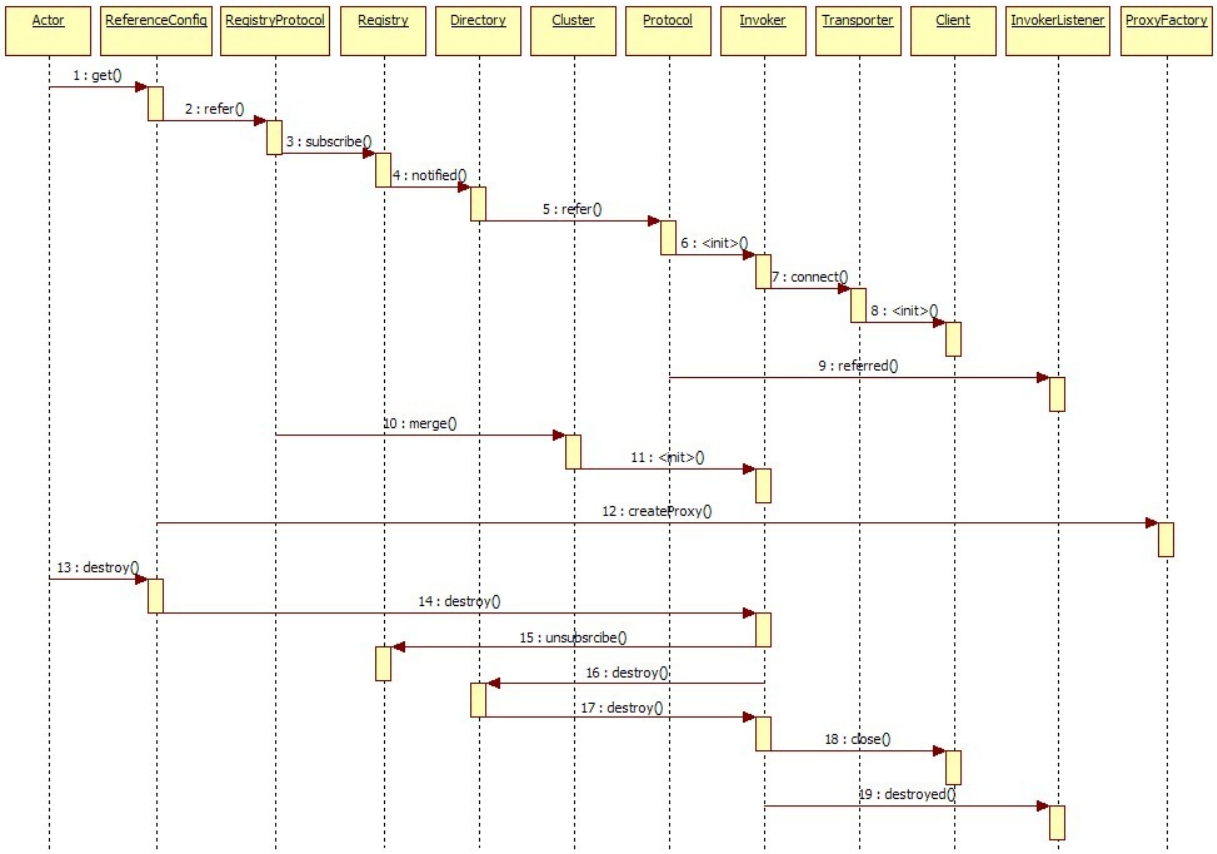
暴露服务时序

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。

基本设计原则

- 采用 Microkernel + Plugin 模式，Microkernel 只负责组装 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[1]，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。

示例：

以扩展 Dubbo 的协议为例，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[2]：

```
package com.alibaba.xxx;

import com.alibaba.dubbo.rpc.Protocol;

public class XxxProtocol implemenets Protocol {
    // ...
}
```

配置模块中的配置

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" />
```

扩展点特性

扩展点自动包装

自动包装扩展点的 Wrapper 类。`ExtensionLoader` 在加载扩展点时，如果加载到的扩展点有拷贝构造函数，则判定为扩展点 Wrapper 类。

Wrapper类内容：

```
package com.alibaba.xxx;

import com.alibaba.dubbo.rpc.Protocol;

public class XxxProtocolWrapper implemenets Protocol {
    Protocol impl;

    public XxxProtocol(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 类同样实现了扩展点接口，但是 Wrapper 不是扩展点的真正实现。它的用途主要是用于从 `ExtensionLoader` 返回扩展点时，包装在真正的扩展点实现外。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，即 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 setter 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 implemenets 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 扩展点自适应 中说明。

扩展点自适应

`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` 、`WheelMaker`

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 implemenets 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 Remoting
Exception;

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

对于 bind() 方法，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。

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 {
    // ...
}
```

1. 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 ↩

2. 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），缓存在 `ExtensionLoader` 中 ↩

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 协议扩展点，基于扩展点的 扩展点自适应机制，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[1]，`ServiceConfig` 解析出的 URL 的格式为： `dubbo://service-host/com.foo.FooService?version=1.0.0` 。

基于扩展点自适应机制，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[2]，`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")` ，

基于扩展点自适应机制，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` ，然后基于扩展点自适应机制，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[3]，`ReferenceConfig` 解析出的 URL 的格式为： `dubbo://service-host/com.foo.FooService?version=1.0.0` 。

基于扩展点自适应机制，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[4]，`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")` 。

基于扩展点自适应机制，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` 。

基于扩展点自适应机制，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

基于扩展点自适应机制，所有的 `Protocol` 扩展点都会自动套上 `Wrapper` 类。

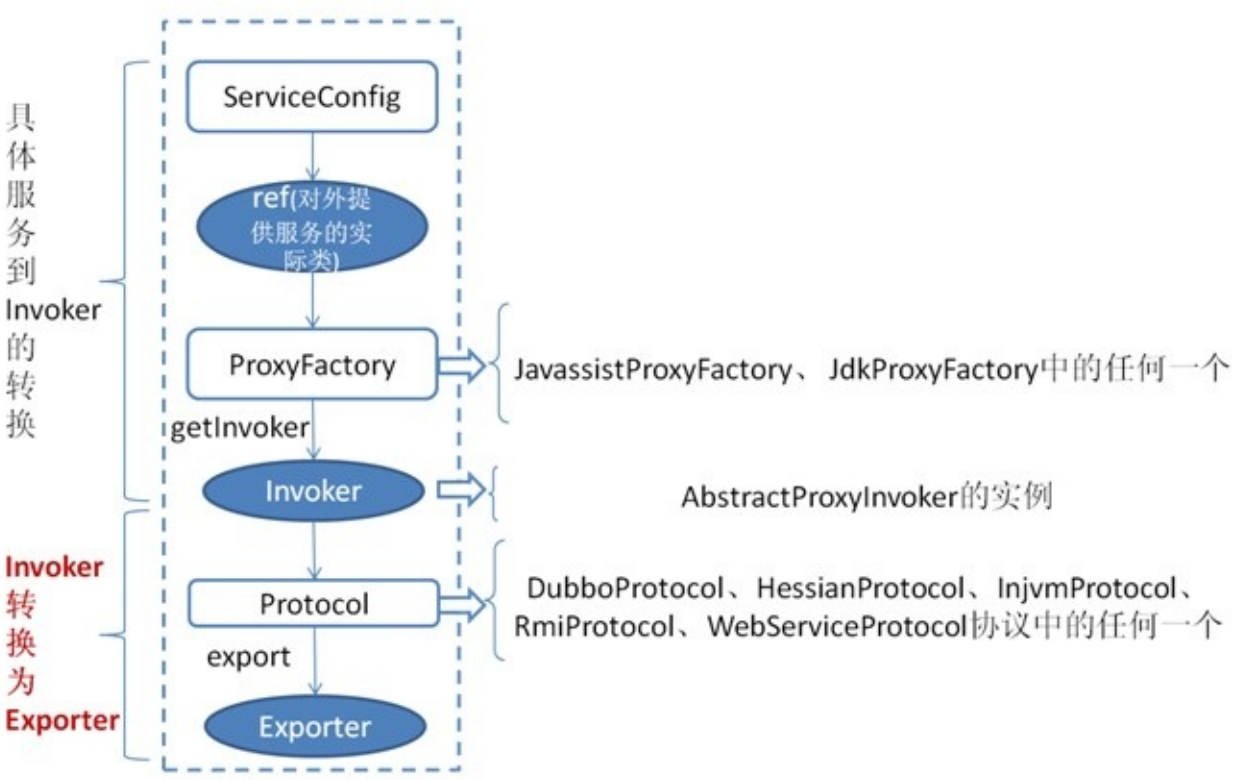
基于 `ProtocolFilterWrapper` 类，将所有 `Filter` 组装成链，在链的最后一节调用真实的引用。

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

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，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：

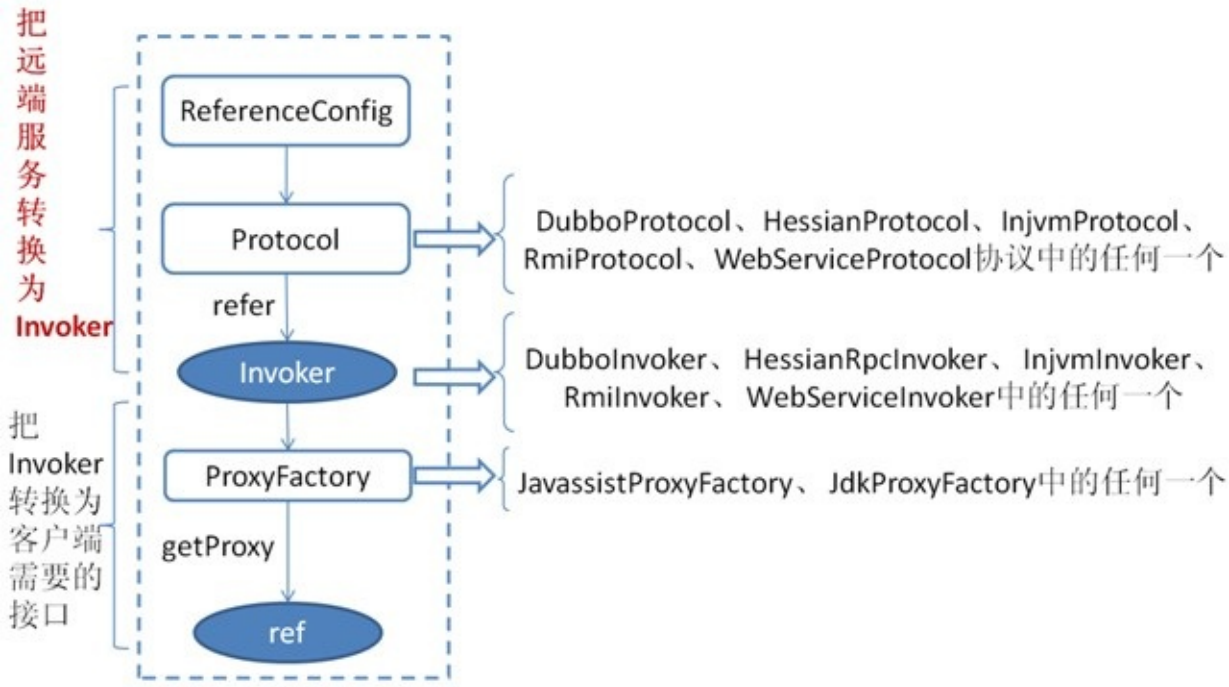
Dubbo 的实现

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



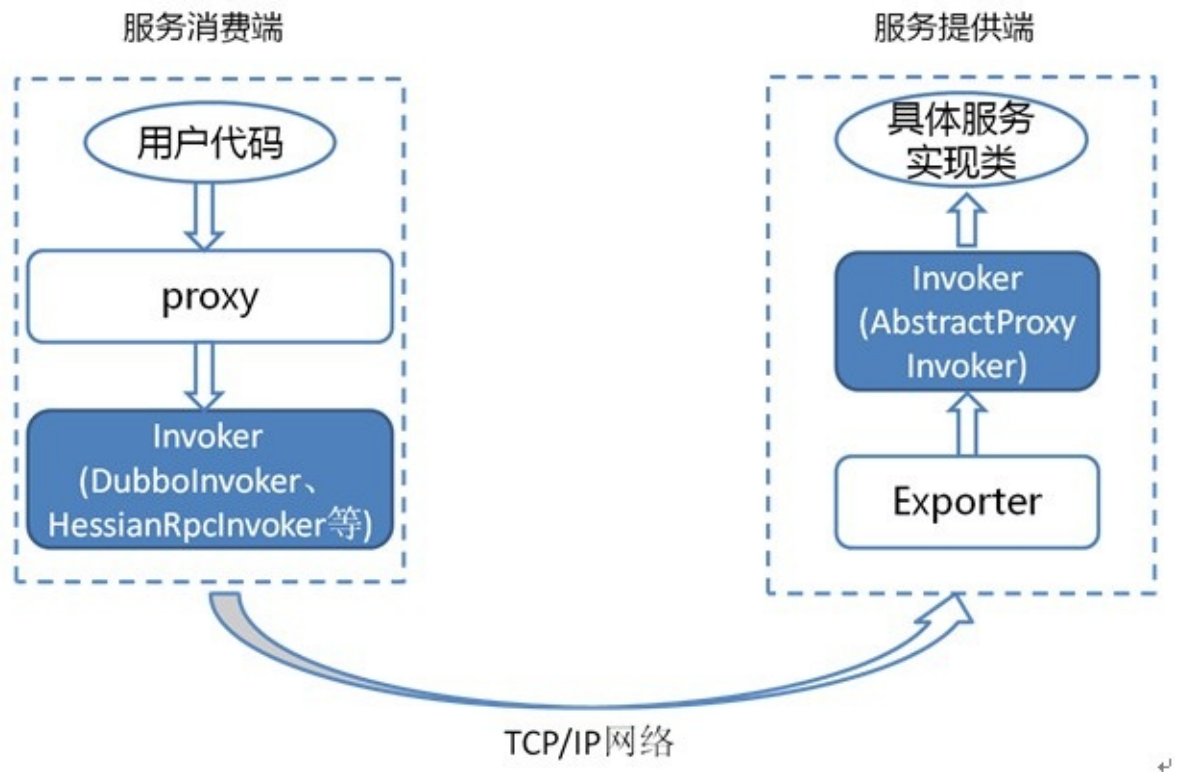
上图是服务消费的主过程：

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` [5]，而该 `Invoker` 实现了真正的远程服务调用。

服务提供者代码：

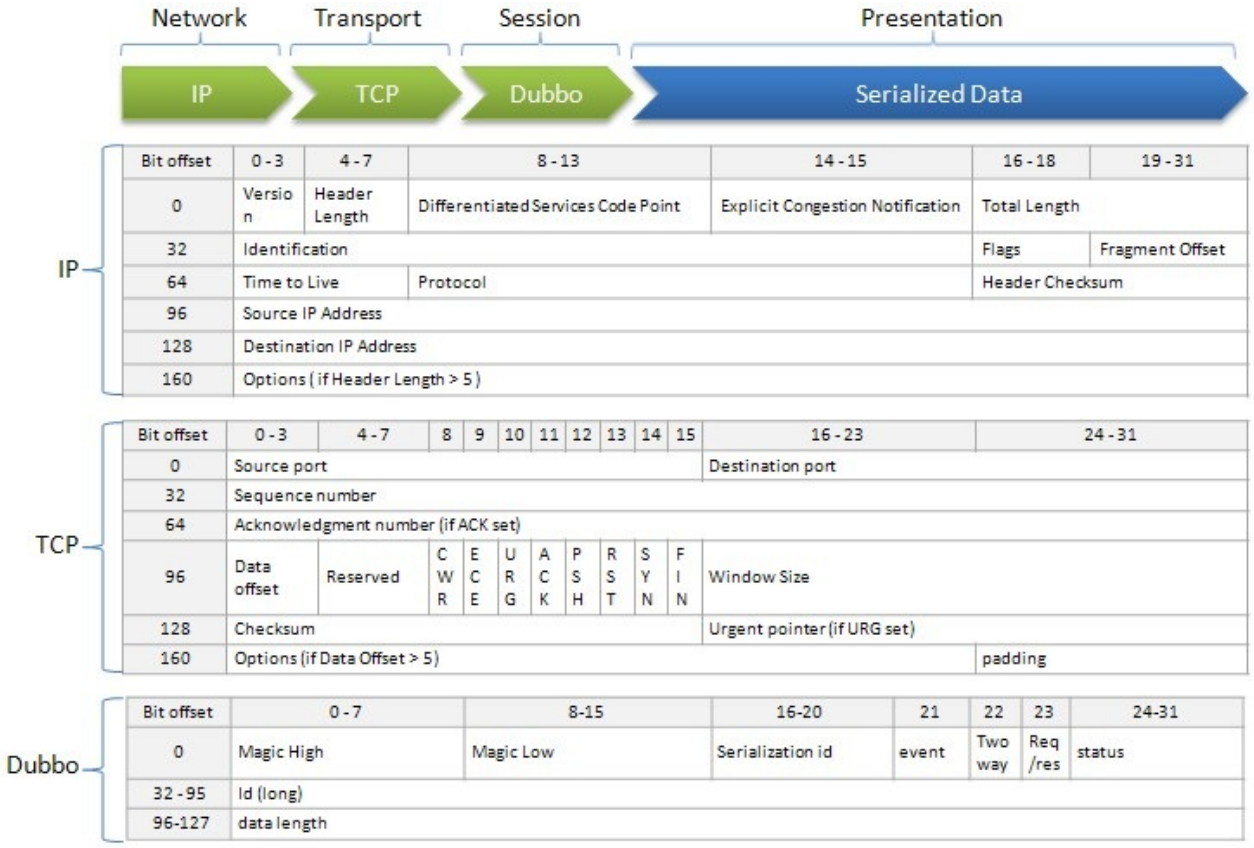
```
public class DemoServiceImpl implements DemoService {

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

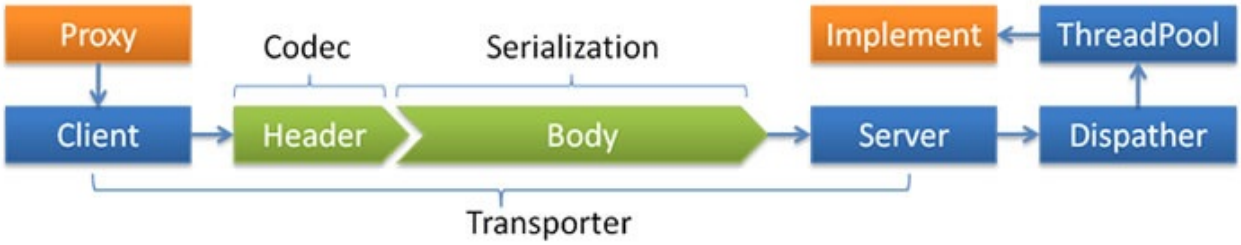
上面这个类会被封装成为一个 `AbstractProxyInvoker` 实例，并新生成一个 `Exporter` 实例。这样当网络通讯层收到一个请求后，会找到对应的 `Exporter` 实例，并调用它所对应的 `AbstractProxyInvoker` 实例，从而真正调用了服务提供者的代码。Dubbo 里还有一些其他的 `Invoker` 类，但上面两种是最重要的。

远程通讯细节

协议头约定



线程派发模型



- Dispather: `all` , `direct` , `message` , `execution` , `connection`
- ThreadPool: `fixed` , `cached`

1. 即： `<dubbo:service regisrty="N/A" />` 或者 `<dubbo:registry address="N/A" />` ↩

2. 即: `<dubbo:registry address="zookeeper://10.20.153.10:2181" />` ↩

3. 即： `<dubbo:reference url="dubbo://service-host/com.foo.FooService?version=1.0.0" />` ↩

4. 即： `<dubbo:registry address="zookeeper://10.20.153.10:2181" />` ↩

5

> 5. `DubboInvoker`、`HessianRpcInvoker`、`InjvmInvoker`、`RmiInvoker`、`WebServiceInvoker`中的任何一个 ↩

SPI 扩展实现

SPI 扩展接口仅用于系统集成，或 Contributor 扩展功能插件。

协议扩展

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，协议通常需要在此 `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。

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.getCon
text().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 RpcExcepti
on;

    /**
     * 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 RpcExcep
tion;

}
```

扩展配置

```
<!-- 声明协议，如果没有配置id，将以name为id -->
<dubbo:protocol id="xxx1" name="xxx" />
<!-- 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:service 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 -->
<dubbo:provider protocol="xxx1" />
```

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`

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 Rpc
Exception {
        return new XxxExporter(invoker);
    }
    public <T> Invoker<T> refer(Class<T> type, URL url) throws R
pcException {
        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 RemotingExcept
ion{
        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 RemotingExc
eption{
        super(type, url);
    }
    protected abstract Object doInvoke(Invocation invocation) th
rows Throwable {
        // ...
    }
}
```

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

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

调用拦截扩展

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" />`

扩展接口

`com.alibaba.dubbo.rpc.Filter`

扩展配置

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

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`

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 invocati
on) 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

扩展说明

当有服务引用时，触发该事件。

扩展接口

`com.alibaba.dubbo.rpc.InvokerListener`

扩展配置

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

已知扩展

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

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 RpcExceptio
n {
        // ...
    }
}
```

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

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

暴露监听扩展

扩展说明

当有服务暴露时，触发该事件。

扩展接口

`com.alibaba.dubbo.rpc.ExporterListener`

扩展配置

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

已知扩展

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

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 RpcExcepti
on {
        // ...
    }
    public void unexported(Exporter<?> exporter) throws RpcExcep
tion {
        // ...
    }
}
```

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

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

集群扩展

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。

扩展接口

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

扩展配置

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

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`

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

扩展说明

从多个服务提者方中选择一个进行调用。

扩展接口

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

已知扩展

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

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

扩展说明

从多个服务提者方中选择一个进行调用

扩展接口

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

扩展配置

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

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`

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

扩展说明

合并返回结果，用于分组聚合。

扩展接口

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

扩展配置

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

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`

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

扩展说明

负责服务的注册与发现。

扩展接口

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

扩展配置

```
<!-- 定义注册中心 -->
<dubbo:registry id="xxx1" address="xxx://ip:port" />
<!-- 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:service 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 -->
<dubbo:provider registry="xxx1" />
```

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 RegistryS
ervice
    /**
     * 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pro
viders，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。<b
r>
     * 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);

}
```

已知扩展

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

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：

```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

扩展说明

负责服务调用次和调用时间的监控。

扩展接口

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

扩展配置

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

已知扩展

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

扩展示例

Maven 项目结构：

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxMonitorFactoryjava (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：

```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

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。

扩展接口

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

扩展配置

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

已知扩展

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

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.ExtensionFa
ctory (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

扩展说明

将 `Invoker` 接口转换成业务接口。

扩展接口

`com.alibaba.dubbo.rpc.ProxyFactory`

扩展配置

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

已知扩展

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

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 RpcExceptio
n {
        // ...
    }
    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

扩展说明

Java 代码编译器，用于动态生成字节码，加速调用。

扩展接口

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

扩展配置

自动加载

已知扩展

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

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：

```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

扩展说明

通道信息派发器，用于指定线程池模型。

扩展接口

`com.alibaba.dubbo.remoting.Dispatcher`

扩展配置

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

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`

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

扩展说明

服务提供方线程程实现策略，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。

扩展接口

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

扩展配置

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

<dubbo:provider threadpool="xxx" />
```

已知扩展

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

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

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。

扩展接口

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

扩展配置

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

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`

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.Serializati
on (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, OutputS
tream output) throws IOException {
        return new XxxObjectOutput(output);
    }
    public ObjectInput deserialize(Parameters parameters, InputS
tream 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

扩展说明

远程通讯的服务器及客户端传输实现。

扩展接口

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

扩展配置

```
<!-- 服务器和客户端使用相同的传输实现 -->
<dubbo:protocol transporter="xxx" />
<!-- 服务器和客户端使用不同的传输实现 -->
<dubbo:protocol 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 -->
<dubbo:provider transporter="xxx" server="xxx" client="xxx" />
```

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`

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.Abstract
Server;

public class XxxServer extends AbstractServer {
    public XxxServer(URL url, ChannelHandler handler) throws Rem
otingException{
        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.Abstract
Client;

public class XxxClient extends AbstractClient {
    public XxxServer(URL url, ChannelHandler handler) throws Rem
otingException{
        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

扩展说明

基于传输层之上，实现 Request-Response 信息交换语义。

扩展接口

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

扩展配置

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

已知扩展

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

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 handl
er) throws RemotingException {
        return new XxxExchangeClient(url, handler);
    }
}
```

XxxExchangeServer.java：

```
package com.xxx;

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

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

XxxExchangeClient.java：

```
package com.xxx;

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

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

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

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

组网扩展

扩展说明

对等网络节点组网器。

扩展接口

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

扩展配置

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

已知扩展

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

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

扩展说明

所有服务器均支持 telnet 访问，用于人工干预。

扩展接口

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

扩展配置

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

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`

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.TelnetHandle
r (纯文本文件，内容为：xxx=com.xxx.XxxTelnetHandler)
```

XxxTelnetHandler.java：

```
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 {
        // ...
    }
}
```

META-INF/dubbo/com.alibaba.dubbo.remoting.telnet.TelnetHandler：

```
xxx=com.xxx.XxxTelnetHandler
```

用法

```
telnet 127.0.0.1 20880
dubbo> xxx args
```

状态检查扩展

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。

扩展接口

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

扩展配置

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

已知扩展

- `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`

扩展示例

Maven 项目结构：

```
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)
```

XxxStatusChecker.java：

```
package com.xxx;

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

扩展说明

服务容器扩展，用于自定义加载内容。

扩展接口

`com.alibaba.dubbo.container.Container`

扩展配置

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

已知扩展

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

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

扩展说明

对等网络节点组网器。

扩展接口

`com.alibaba.dubbo.container.page.PageHandler`

扩展配置

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

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`

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

扩展说明

用请求参数作为 key，缓存返回结果。

扩展接口

`com.alibaba.dubbo.cache.CacheFactory`

扩展配置

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

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`

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：

```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

扩展说明

参数验证扩展点。

扩展接口

`com.alibaba.dubbo.validation.Validation`

扩展配置

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

已知扩展

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

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

扩展说明

日志输出适配扩展点。

扩展接口

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

扩展配置

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

或者：

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

已知扩展

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

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
```

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。

编码约定

代码风格

Dubbo 的源代码和 JavaDoc 遵循以下的规范：

- Code Conventions for the Java Programming Language
- How to Write Doc Comments for the Javadoc Tool

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[1]。
- 配置属性全部用小写，多个单词用"-"号分隔[2]。
- URL参数全部用小写，多个单词用"."号分隔[3]。
- 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` 只负载加载扩展点，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。

[1]. Java 约定 ↩

2

> [2]. Spring 约定 ↩

> [3]. Dubbo 约定 ↩

设计原则

本章节的设计原则摘录自梁飞在 javaeye 上发表的系列文章。

魔鬼在细节

> http://javatar.iteye.com/blog/1056664

最近一直担心 Dubbo 分布式服务框架后续如果维护人员增多或变更，会出现质量的下降， 我在想，有没有什么是需要大家共同遵守的，根据平时写代码时的一习惯，总结了一下在写代码过程中，尤其是框架代码，要时刻牢记的细节。可能下面要讲的这些，大家都会觉得很简单，很基础，但要做到时刻牢记。在每一行代码中都考虑这些因素，是需要很大耐心的， 大家经常说，魔鬼在细节中，确实如此。

防止空指针和下标越界

这是我最不喜欢看到的异常，尤其在核心框架中，我更愿看到信息详细的参数不合法异常。这也是一个健状的程序开发人员，在写每一行代码都应在潜意识中防止的异常。基本上要能确保一次写完的代码，在不测试的情况，都不会出现这两个异常才算合格。

保证线程安全性和可见性

对于框架的开发人员，对线程安全性和可见性的深入理解是最基本的要求。需要开发人员，在写每一行代码时都应在潜意识中确保其正确性。因为这种代码，在小并发下做功能测试时，会显得很正常。但在高并发下就会出现莫明其妙的问题，而且场景很难重现，极难排查。

尽早失败和前置断言

尽早失败也应该成为潜意识，在有传入参数和状态变化时，均在入口处全部断言。一个不合法的值和状态，在第一时间就应报错，而不是等到要用时才报错。因为等到要用时，可能前面已经修改其它相关状态，而在程序中很少有人去处理回滚逻辑。这样报错后，其实内部状态可能已经混乱，极易在一个隐蔽分支上引发程序不可恢复。

分离可靠操作和不可靠操作

这里的可靠是狭义的指是否会抛出异常或引起状态不一致，比如，写入一个线程安全的 Map，可以认为是可靠的，而写入数据库等，可以认为是不可靠的。开发人员必须在写每一行代码时，都注意它的可靠性与否，在代码中尽量划分开，并对失败做异常处理，并为容错，自我保护，自动恢复或切换等补偿逻辑提供清晰的切入点，保证后续增加的代码不至于放错位置，而导致原先的容错处理陷入混乱。

异常防御，但不忽略异常

这里讲的异常防御，指的是对非必须途径上的代码进行最大限度的容忍，包括程序上的 BUG，比如：获取程序的版本号，会通过扫描 Manifest 和 jar 包名称抓取版本号，这个逻辑是辅助性的，但代码却不少，初步测试也没啥问题，但应该在整个 getVersion() 中加上一个全函数的 try-catch 打印错误日志，并返回基本版本，因为 getVersion() 可能存在未知特定场景异常，或被其他的开发人员误修改逻辑(但一般人员不会去掉 try-catch)，而如果它抛出异常会导致主流程异常，这是我们不希望看到的。但这里要控制个度，不要随意 try-catch，更不要无声无息的吃掉异常。

缩小可变域和尽量 **final**

如果一个类可以成为不变类(Immutable Class)，就优先将它设计成不变类。不变类有天然的并发共享优势，减少同步或复制，而且可以有效帮忙分析线程安全的范围。就算是可变类，对于从构造函数传入的引用，在类中持有时，最好将字段 final，以免被中途误修改引用。不要以为这个字段是私有的，这个类的代码都是我自己写的，不会出现对这个字段的重新赋值。要考虑的一个因素是，这个代码可能被其他人修改，他不知道你的这个弱约定，final 就是一个不变契约。

降低修改时的误解性，不埋雷

前面不停的提到代码被其他人修改，这也开发人员要随时紧记的。这个其他人包括未来的自己，你要总想着这个代码可能会有人去改它。我应该给修改的人一点什么提示，让他知道我现在的设计意图，而不要在程序里面加潜规则，或埋一些容易忽视的雷，比如：你用 null 表示不可用，size 等于 0 表示黑名单，这就是一个雷，下一个修改者，包括你自己，都不会记得有这样的约定，可能后面为了改某个其它 BUG，不小心改到了这里，直接引爆故障。对于这个例子，一个原则就是永远不要区分 null 引用和 empty 值。

提高代码的可测性

这里的可测性主要指 Mock 的容易程度，和测试的隔离性。至于测试的自动性，可重复性，非偶然性，无序性，完备性(全覆盖)，轻量性(可快速执行)，一般开发人员，加上 JUnit 等工具的辅助基本都能做到，也能理解它的好处，只是工作量问题。这里要特别强调的是测试用例的单一性(只测目标类本身)和隔离性(不传染失败)。现在的测试代码，过于强调完备性，大量重复交叉测试，看起来没啥坏处，但测试代码越多，维护代价越高。经常出现的问题是，修改一行代码或加一个判断条件，引起 100 多个测试用例不通过。时间一紧，谁有这个闲功夫去改这么多形态各异的测试用例？久而久之，这个测试代码就已经不能真实反应代码现在的状况，很多时候会被迫绕过。最好的情况是，修改一行代码，有且只有一行测试代码不通过。如果修改了代码而测试用例还能通过，那也不行，表示测试没有覆盖到。另外，可 Mock 性是隔离的基础，把间接依赖的逻辑屏蔽掉。可 Mock 性的一个最大的杀手就是静态方法，尽量少用。

一些设计上的基本常识

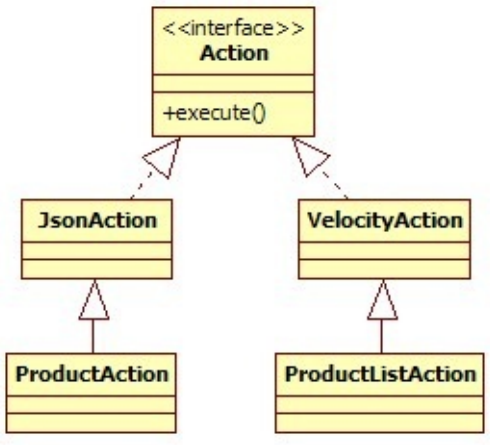
> http://javatar.iteye.com/blog/706098

最近给团队新人讲了一些设计上的常识，可能会对其它的新人也有些帮助，把暂时想到的几条，先记在这里。

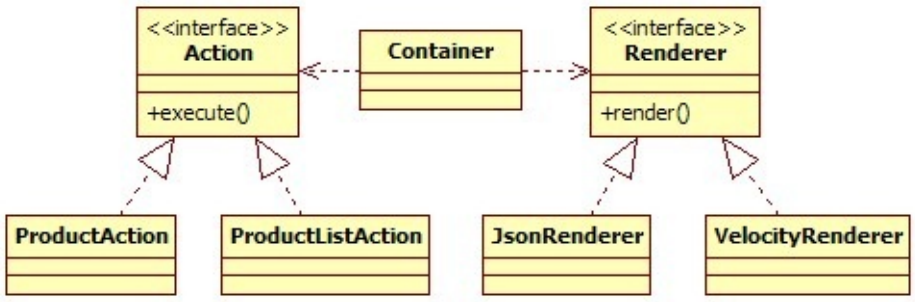
API 与 SPI 分离

框架或组件通常有两类客户，一个是使用者，一个是扩展者。API (Application Programming Interface) 是给使用者用的，而 SPI (Service Provide Interface) 是给扩展者用的。在设计时，尽量把它们隔离开，而不要混在一起。也就是说，使用者是看不到扩展者写的实现的。

比如：一个 Web 框架，它有一个 API 接口叫 Action，里面有个 execute() 方法，是给使用者用来写业务逻辑的。然后，Web 框架有一个 SPI 接口给扩展者控制输出方式，比如用 velocity 模板输出还是用 json 输出等。如果这个 Web 框架使用一个都继承 Action 的 VelocityAction 和一个 JsonAction 做为扩展方式，要用 velocity 模板输出的就继承 VelocityAction，要用 json 输出的就继承 JsonAction，这就是 API 和 SPI 没有分离的反面例子，SPI 接口混在了 API 接口中。



合理的方式是，有一个单独的 Renderer 接口，有 VelocityRenderer 和 JsonRenderer 实现，Web 框架将 Action 的输出转交给 Renderer 接口做渲染输出。

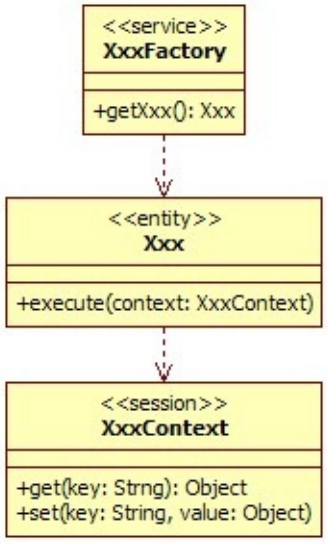


服务域/实体域/会话域分离

任何框架或组件，总会有核心领域模型，比如：Spring 的 Bean，Struts 的 Action，Dubbo 的 Service，Napoli 的 Queue 等等。这个核心领域模型及其组成部分称为实体域，它代表着我们要操作的目标本身。实体域通常是线程安全的，不管是通过不变类，同步状态，或复制的方式。

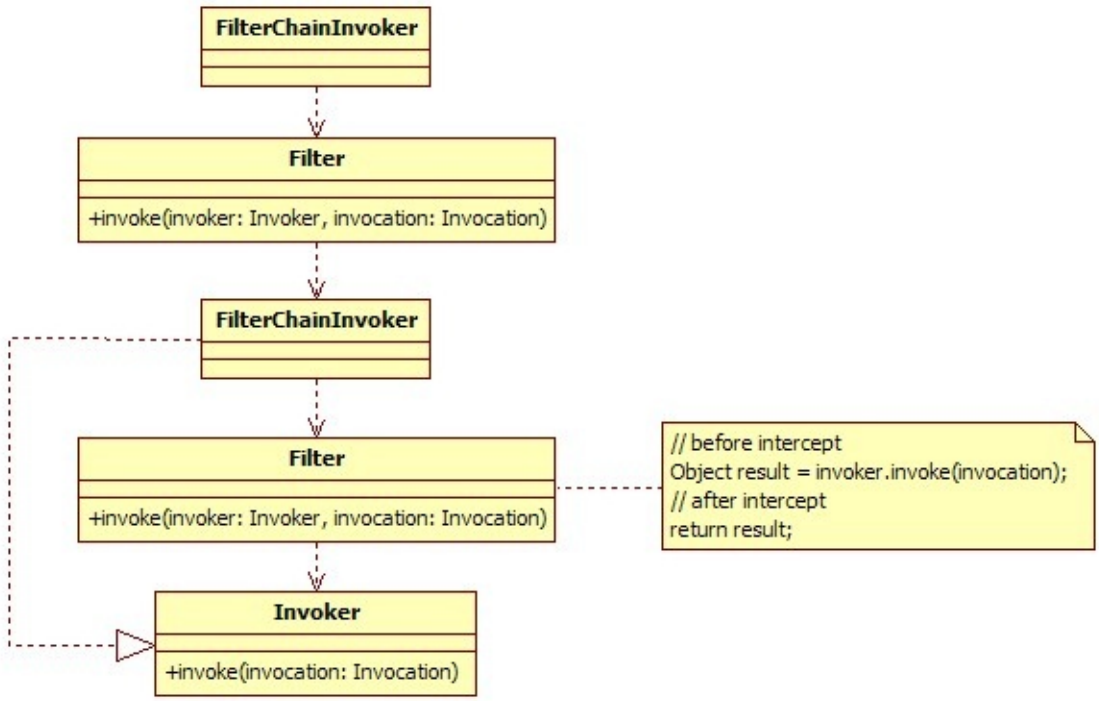
服务域也就是行为域，它是组件的功能集，同时也负责实体域和会话域的生命周期管理， 比如 Spring 的 ApplicationContext，Dubbo 的 ServiceManager 等。服务域的对象通常会比较重，而且是线程安全的，并以单一实例服务于所有调用。

什么是会话？就是一次交互过程。会话中重要的概念是上下文，什么是上下文？比如我们说：“老地方见”，这里的“老地方”就是上下文信息。为什么说“老地方”对方会知道，因为我们前面定义了“老地方”的具体内容。所以说，上下文通常持有交互过程中的状态变量等。会话对象通常较轻，每次请求都重新创建实例，请求结束后销毁。简而言之：把元信息交由实体域持有，把一次请求中的临时状态由会话域持有，由服务域贯穿整个过程。



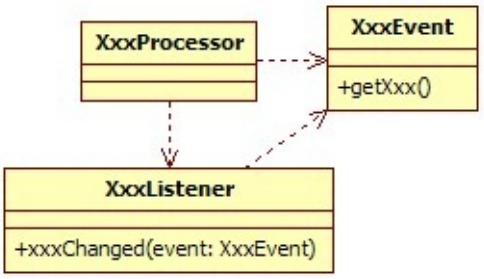
在重要的过程上设置拦截接口

如果你要写个远程调用框架，那远程调用的过程应该有一个统一的拦截接口。如果你要写一个 ORM 框架，那至少 SQL 的执行过程，Mapping 过程要有拦截接口；如果你要写一个 Web 框架，那请求的执行过程应该要有拦截接口，等等。没有哪个公用的框架可以 Cover 住所有需求，允许外置行为，是框架的基本扩展方式。这样，如果有人想在远程调用前，验证下令牌，验证下黑白名单，统计下日志；如果有人想在 SQL 执行前加下分页包装，做下数据权限控制，统计下 SQL 执行时间；如果有人想在请求执行前检查下角色，包装下输入输出流，统计下请求量，等等，就可以自行完成，而不用侵入框架内部。拦截接口，通常是把过程本身用一个对象封装起来，传给拦截器链，比如：远程调用主过程为 invoke()，那拦截器接口通常为 invoke(Invocation)，Invocation 对象封装了本来要执行过程的上下文，并且 Invocation 里有一个 invoke() 方法，由拦截器决定什么时候执行，同时，Invocation 也代表拦截器行为本身，这样上一拦截器的 Invocation 其实是包装的下一拦截器的过程，直到最后一个拦截器的 Invocation 是包装的最终的 invoke() 过程；同理，SQL 主过程为 execute()，那拦截器接口通常为 execute(Execution)，原理一样。当然，实现方式可以任意，上面只是举例。



重要的状态的变更发送事件并留出监听接口

这里先要讲一个事件和上面拦截器的区别，拦截器是干预过程的，它是过程的一部分，是基于过程行为的，而事件是基于状态数据的，任何行为改变的相同状态，对事件应该是一致的。事件通常是事后通知，是一个 Callback 接口，方法名通常是过去式的，比如 onChanged()。比如远程调用框架，当网络断开或连上应该发出一个事件，当出现错误也可以考虑发出一个事件，这样外围应用就有可能观察到框架内部的变化，做相应适应。



扩展接口职责尽可能单一，具有可组合性

比如，远程调用框架它的协议是可以替换的。如果只提供一个总的扩展接口，当然可以做到切换协议，但协议支持是可以细分为底层通讯，序列化，动态代理方式等等。如果将接口拆细，正交分解，会更便于扩展者复用已有逻辑，而只是替换某部分实现策略。当然这个分解的粒度需要把握好。

微核插件式，平等对待第三方

大凡发展的比较好的框架，都遵守微核的理念。Eclipse 的微核是 OSGi， Spring 的微核是 BeanFactory，Maven 的微核是 Plexus。通常核心是不应该带有功能性的，而是一个生命周期和集成容器，这样各功能可以通过相同的方式交互及扩展，并且任何功能都可以被替换。如果做不到微核，至少要平等对待第三方，即原作者能实现的功能，扩展者应该可以通过扩展的方式全部做到。原作者要把自己也当作扩展者，这样才能保证框架的可持续性及由内向外的稳定性。

不要控制外部对象的生命周期

比如上面说的 Action 使用接口和 Renderer 扩展接口。框架如果让使用者或扩展者把 Action 或 Renderer 实现类的类名或类元信息报上来，然后在内部通过反射 newInstance() 创建一个实例，这样框架就控制了 Action 或 Renderer 实现类的生

命周期，Action 或 Renderer 的生老病死，框架都自己做了，外部扩展或集成都无能为力。好的办法是让使用者或扩展者把 Action 或 Renderer 实现类的实例报上来，框架只是使用这些实例，这些对象是怎么创建的，怎么销毁的，都和框架无关，框架最多提供工具类辅助管理，而不是绝对控制。

可配置一定可编程，并保持友好的 CoC 约定

因为使用环境的不确定因素很多，框架总会有一些配置，一般都会到 classpath 直扫某个指定名称的配置，或者启动时允许指定配置路径。做为一个通用框架，应该做到凡是能配置文件做的一定要能通过编程方式进行，否则当使用者需要将你的框架与另一个框架集成时就会带来很多不必要的麻烦。

另外，尽可能做一个标准约定，如果用户按某种约定做事时，就不需要该配置项。比如：配置模板位置，你可以约定，如果放在 templates 目录下就不用配了，如果你想换个目录，就配置下。

区分命令与查询，明确前置条件与后置条件

这个是契约式设计的一部分，尽量遵守有返回值的方法是查询方法，void 返回的方法是命令。查询方法通常是幂等性的，无副作用的，也就是不改变任何状态，调 n 次结果都是一样的，比如 get 某个属性值，或查询一条数据库记录。命令是指有副作用的，也就是会修改状态，比如 set 某个值，或 update 某条数据库记录。如果你的方法即做了修改状态的操作，又做了查询返回，如果可能，将其拆成写读分离的两个方法，比如：User deleteUser(id)，删除用户并返回被删除的用户，考虑改为 getUser() 和 void 的 deleteUser()。另外，每个方法都尽量前置断言传入参数的合法性，后置断言返回结果的合法性，并文档化。

增量式扩展，而不要扩充原始核心概念

参见：谈谈扩充式扩展与增量式扩展

谈谈扩充式扩展与增量式扩展

> http://javatar.iteye.com/blog/690845

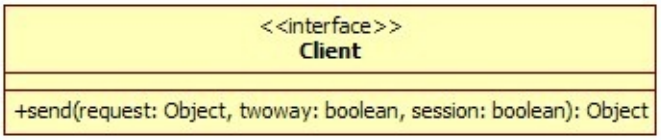
我们平台的产品越来越多，产品的功能也越来越多。平台的产品为了适应各 BU 和部门以及产品线的需求，势必会将很多不相干的功能凑在一起，客户可以选择性的使用。为了兼容更多的需求，每个产品，每个框架，都在不停的扩展，而我们经常会选择一些扩展的扩展方式，也就是将新旧功能扩展成一个通用实现。我想讨论是，有些情况下也可以考虑增量式的扩展方式，也就是保留原功能的简单性，新功能独立实现。我最近一直做分布式服务框架的开发，就拿我们项目中的问题开涮吧。

比如：远程调用框架，肯定少不了序列化功能，功能很简单，就是把流转成对象，对象转成流。但因有些地方可能会使用 osgi，这样序列化时，IO 所在的 ClassLoader 可能和业务方的 ClassLoader 是隔离的。需要将流转换成 byte[] 数组，然后传给业务方的 ClassLoader 进行序列化。为了适应 osgi 需求，把原来非 osgi 与 osgi 的场景扩展了一下，这样，不管是不是 osgi 环境，都先将流转成 byte[] 数组，拷贝一次。然而，大部分场景都用不上 osgi，却为 osgi 付出了代价。而如果采用增量式扩展方式，非 osgi 的代码原封不动，再加一个 osgi 的实现，要用 osgi 的时候，直接依赖 osgi 实现即可。

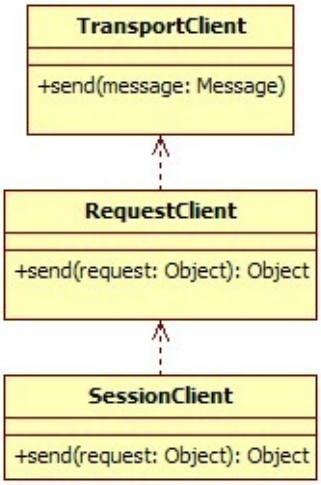
再比如：最开始，远程服务都是基于接口方法，进行透明化调用的。这样，扩展接口就是， invoke(Method method, Object[] args)，后来，有了无接口调用的需求，就是没有接口方法也能调用，并将 POJO 对象都转换成 Map 表示。因为 Method 对象是不能直接 new 出来的，我们不自觉选了一个扩展式扩展，把扩展接口改成了 invoke(String methodName, String[] parameterTypes, String returnTypes, Object[] args)，导致不管是不是无接口调用，都得把 parameterTypes 从 Class[] 转成 String[]。如果选用增量式扩展，应该是保持原有接口不变，增加一个 GeneralService 接口，里面有一个通用的 invoke() 方法，和其它正常业务上的接口一样的调用方式，扩展接口也不用变，只是 GeneralServiceImpl 的 invoke() 实现会将收到的调用转给目标接口，这样就能将新功能增量到旧功能上，并保持原来结构的简单性。

再再比如：无状态消息发送，很简单，序列化一个对象发过去就行。后来有了同步消息发送需求，需要一个 Request/Response 进行配对，采用扩展式扩展，自然想到，无状态消息其实是一个没有 Response 的 Request，所以在 Request 里加一个

boolean 状态，表示要不要返回 Response。如果再来一个会话消息发送需求，那就再加一个 Session 交互，然后发现，原来同步消息发送是会话消息的一种特殊情况，所有场景都传 Session，不需要 Session 的地方无视即可。



如果采用增量式扩展，无状态消息发送原封不动，同步消息发送，在无状态消息基础上加一个 Request/Response 处理，会话消息发送，再加一个 SessionRequest/SessionResponse 处理。



配置设计

> http://javatar.iteye.com/blog/949527

Dubbo 现在的设计是完全无侵入，也就是使用者只依赖于配置契约。经过多个版本的发展，为了满足各种需求场景，配置越来越多。为了保持兼容，配置只增不减，里面潜伏着各种风格，约定，规则。新版本也将配置做了一次调整，去掉了 dubbo.properties，改为全 spring 配置。将想到的一些记在这，备忘。

配置分类

首先，配置的用途是有多种的，大致可以分为：

1. 环境配置，比如：连接数，超时等配置。
2. 描述配置，比如：服务接口描述，服务版本等。
3. 扩展配置，比如：协议扩展，策略扩展等。

配置格式

通常环境配置，用 properties 配置会比较方便，因为都是一些离散的简单值，用 key-value 配置可以减少配置的学习成本。

而描述配置，通常信息比较多，甚至有层次关系，用 xml 配置会比较方便，因为树结构的配置表现力更强。如果非常复杂，也可以考自定义 DSL 做为配置。有时候这类配置也可以用 Annotation 代替， 因为这些配置和业务逻辑相关，放在代码里也是合理的。

另外扩展配置，可能不尽相同。如果只是策略接口实现类替换，可以考虑 properties 等结构。如果有复杂的生命周期管理，可能需要 XML 等配置。有时候扩展会通过注册接口的方式提供。

配置加载

对于环境配置，在 java 世界里，比较常规的做法，是在 classpath 下约定一个以项目为名称的 properties 配置，比如：log4j.properties，velocity.properties等。产品在初始化时，自动从 classpath 下加载该配置。我们平台的很多项目也使用类似策略，如：dubbo.properties，comsat.xml 等。这样有它的优势，就是基于约定，简化了用户对配置加载过程的干预。但同样有它的缺点，当 classpath 存在同样的配置时，可能误加载，以及在 ClassLoader 隔离时，可能找不到配置，并且，当用户希望将配置放到统一的目录时，不太方便。

Dubbo 新版本去掉了 dubbo.properties，因为该约定经常造成配置冲突。

而对于描述配置，因为要参与业务逻辑，通常会嵌到应用的生命周期管理中。现在使用 spring 的项目越来越多，直接使用 spring 配置的比较普遍，而且 spring 允许自定义 schema，配置简化后很方便。当然，也有它的缺点，就是强依赖 spring，可以提编程接口做了配套方案。

在 Dubbo 即存在描述配置，也有环境配置。一部分用 spring 的 schame 配置加载，一部分从 classpath 扫描 properties 配置加载。用户感觉非常不便，所以在新版本中进行了合并，统一放到 spring 的 schame 配置加载，也增加了配置的灵活性。

扩展配置，通常对配置的聚合要求比较高。因为产品需要发现第三方实现，将其加入产品内部。在 java 世界里，通常是约定在每个 jar 包下放一个指定文件加载，比如：eclipse 的 plugin.xml，struts2 的 struts-plugin.xml 等，这类配置可以考虑 java 标准的服务发现机制，即在 jar 包的 META-INF/services 下放置接口类全名文件，内容为每行一个实现类类名，就像 jdk 中的加密算法扩展，脚本引擎扩展，新的 JDBC 驱动等，都是采用这种方式。参见：ServiceProvider 规范。

Dubbo 旧版本通过约定在每个 jar 包下，放置名为 dubbo-context.xml 的 spring 配置进行扩展与集成，新版本改成用 jdk 自带的 META-INF/services 方式，去掉过多的 spring 依赖。

可编程配置

配置的可编程性是非常必要的，不管你以何种方式加载配置文件，都应该提供一个编程的配置方式，允许用户不使用配置文件，直接用代码完成配置过程。因为一个产品，尤其是组件类产品，通常需要和其它产品协作使用，当用户集成你的产品时，可能需要适配配置方式。

Dubbo 新版本提供了与 xml 配置一对一的配置类，如：ServiceConfig 对应 `<dubbo:service />`，并且属性也一对一，这样有利于文件配置与编程配置的一致性理解，减少学习成本。

配置缺省值

配置的缺省值，通常是设置一个常规环境的合理值，这样可以减少用户的配置量。通常建议以线上环境为参考值，开发环境可以通过修改配置适应。缺省值的设置，最好在最外层的配置加载就做处理。程序底层如果发现配置不正确，就应该直接报错，容错在最外层做。如果在程序底层使用时，发现配置值不合理，就填一个缺省值，很容易掩盖表面问题，而引发更深层次的问题。并且配置的中间传递层，很可能并不知道底层使用了一个缺省值，一些中间的检测条件就可能失效。Dubbo 就出现过这样的问题，中间层用“地址”做为缓存 Key， 而底层，给“地址”加了一个缺省端口号，导致不加端口号的“地址”和加了缺省端口的“地址”并没有使用相同的缓存。

配置一致性

配置总会隐含一些风格或潜规则，应尽可能保持其一致性。比如：很多功能都有开关，然后有一个配置值：

1. 是否使用注册中心，注册中心地址。
2. 是否允许重试，重试次数。

你可以约定：

1. 每个都是先配置一个 boolean 类型的开关，再配置一个值。
2. 用一个无效值代表关闭，N/A地址，0重试次数等。

不管选哪种方式，所有配置项，都应保持同一风格，Dubbo 选的是第二种。相似的还有，超时时间，重试时间，定时器间隔时间。如果一个单位是秒，另一个单位是毫秒(C3P0的配置项就是这样)，配置人员会疯掉。

配置覆盖

提供配置时，要同时考虑开发人员，测试人员，配管人员，系统管理员。测试人员是不能修改代码的，而测试的环境很可能较为复杂，需要为测试人员留一些“后门”，可以在外围修改配置项。就像 spring 的 PropertyPlaceholderConfigurer 配

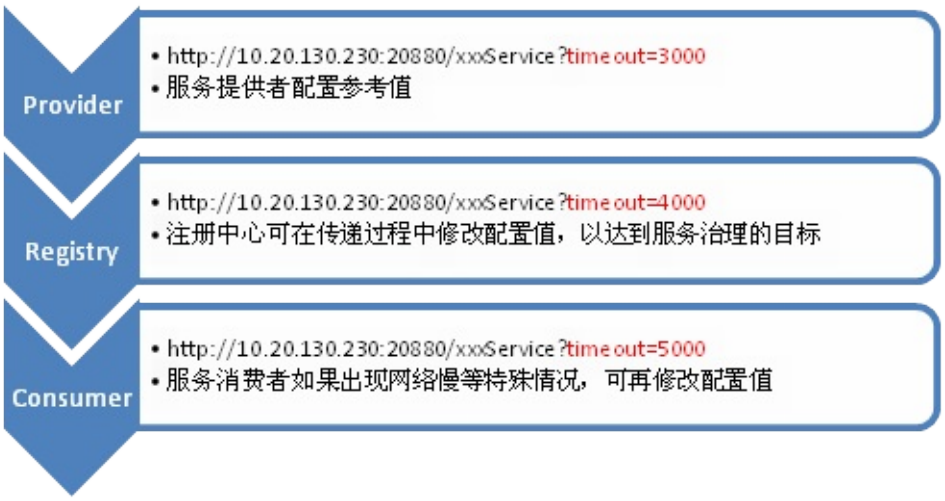
置，支持 `SYSTEM_PROPERTIES_MODE_OVERRIDE` ，可以通过 JVM 的 -D 参数，或者像 hosts 一样约定一个覆盖配置文件，在程序外部，修改部分配置，便于测试。

Dubbo 支持通过 JVM 参数 `-Dcom.xxx.XxxService=dubbo://10.1.1.1:1234` 直接使远程服务调用绕过注册中心，进行点对点测试。还有一种情况，开发人员增加配置时，都会按线上的部署情况做配置，如： `<dubbo:registry address="${dubbo.registry.address}" />` 因为线上只有一个注册中心，这样的配置是没有问题的，而测试环境可能有两个注册中心，测试人员不可能去修改配置，改为： `<dubbo:registry address="${dubbo.registry.address1}" />` ， `<dubbo:registry address="${dubbo.registry.address2}" />` ，所以这个地方，Dubbo 支持在 ${dubbo.registry.address} 的值中，通过竖号分隔多个注册中心地址，用于表示多注册中心地址。

配置继承

配置也存在“重复代码”，也存在“泛化与精化”的问题。比如：Dubbo 的超时时间设置，每个服务，每个方法，都应该可以设置超时时间。但很多服务不关心超时，如果要求每个方法都配置，是不现实的。所以 Dubbo 采用了方法超时继承服务超时，服务超时再继承缺省超时，没配置时，一层层向上查找。

另外，Dubbo 旧版本所有的超时时间，重试次数，负载均衡策略等都只能在服务消费方配置。但实际使用过程中发现，服务提供方比消费方更清楚，但这些配置项是在消费方执行时才用到的。新版本，就加入了在服务提供方也能配这些参数，通过注册中心传递到消费方， 做为参考值，如果消费方没有配置，就以提供方的配置为准，相当于消费方继承了提供方的建议配置值。而注册中心在传递配置时，也可以在中途修改配置，这样就达到了治理的目的，继承关系相当于：服务消费者 --> 注册中心 --> 服务提供者



配置向后兼容

向前兼容很好办，你只要保证配置只增不减，就基本上能保证向前兼容。但向后兼容，也是要注意的，要为后续加入新的配置项做好准备。如果配置出现一个特殊配置，就应该为这个“特殊”情况约定一个兼容规则，因为这个特殊情况，很有可能在以后还会发生。比如：有一个配置文件是保存“服务=地址”映射关系的，其中有一行特殊，保存的是“注册中心=地址”。现在程序加载时，约定“注册中心”这个Key是特殊的，做特别处理，其它的都是“服务”。然而，新版本发现，要加一项“监控中心=地址”，这时，旧版本的程序会把“监控中心”做为“服务”处理，因为旧代码是不能改的，兼容性就很会很麻烦。如果先前约定“特殊标识+XXX”为特殊处理，后续就会方便很多。

向后兼容性，可以多向HTML5学习，参见：HTML5设计原理

设计实现的健壮性

> http://oldratlee.com/380/tech/java/robustness-of-implement.html

Dubbo 作为远程服务暴露、调用和治理的解决方案，是应用运转的经络，其本身实现健壮性的重要程度是不言而喻的。

这里列出一些 Dubbo 用到的原则和方法。

日志

日志是发现问题、查看问题一个最常用的手段。日志质量往往被忽视，没有日志使用上的明确约定。重视 Log 的使用，提高 Log 的信息浓度。日志过多、过于混乱，会导致有用的信息被淹没。

要有效利用这个工具要注意：

严格约定WARN、ERROR级别记录的内容

- WARN 表示可以恢复的问题，无需人工介入。
- ERROR 表示需要人工介入问题。

有了这样的约定，监管系统发现日志文件的中出现 ERROR 字串就报警，又尽量减少了发生。过多的报警会让人疲倦，使人对报警失去警惕性，使 ERROR 日志失去意义。再辅以人工定期查看 WARN 级别信息，以评估系统的“亚健康”程度。

日志中，尽量多的收集关键信息

哪些是关键信息呢？

- 出问题时的现场信息，即排查问题要用到的信息。如服务调用失败时，要给出使用 Dubbo 的版本、服务提供者的 IP、使用的是哪个注册中心；调用的是哪个服务、哪个方法等等。这些信息如果不给出，那么事后人工收集的，问题过后现场可能已经不能复原，加大排查问题的难度。
- 如果可能，给出问题的原因和解决方法。这让维护和问题解决变得简单，而不是寻求精通者（往往是实现者）的帮助。

同一个或是一类问题不要重复记录多次

同一个或是一类异常日志连续出现几十遍的情况，还是常常能看到的。人眼很容易漏掉淹没在其中不一样的重要日志信息。要尽量避免这种情况。在可以预见会出现的情况，有必要加一些逻辑来避免。

如为一个问题准备一个标志，出问题后打日志后设置标志，避免重复打日志。问题恢复后清除标志。

虽然有点麻烦，但是这样做保证日志信息浓度，让监控更有效。

界限设置

资源是有限的，CPU、内存、IO 等等。不要因为外部的请求、数据不受限的而崩溃。

线程池(ExectorService)的大小和饱和策略

Server 端用于处理请求的 ExectorService 设置上限。ExecutorService 的任务等待队列使用有限队列，避免资源耗尽。当任务等待队列饱和时，选择一个合适的饱和策略。这样保证平滑劣化。

在 Dubbo 中，饱和策略是丢弃数据，等待结果也只是请求的超时。

达到饱和时，说明已经达到服务提供方的负荷上限，要在饱和策略的操作中日志记录这个问题，以发出监控警报。记得注意不要重复多次记录哦。（注意，缺省的饱和策略不会有这些附加的操作。）根据警报的频率，已经决定扩容调整等等，避免系统问题被忽略。

集合容量

如果确保进入集合的元素是可控的且是足够少，则可以放心使用。这是大部分的情况。如果不能保证，则使用有有界的集合。当到达界限时，选择一个合适的丢弃策略。

容错-重试-恢复

高可用组件要容忍其依赖组件的失败。

Dubbo 的服务注册中心

目前服务注册中心使用了数据库来保存服务提供者和消费者的信息。注册中心集群不同注册中心也通过数据库来之间同步数据，以感知其它注册中心上提供者。注册中心会内存中保证一份提供者和消费者数据，数据库不可用时，注册中心独立对外正常运转，只是拿不到其它注册中心的数据。当数据库恢复时，重试逻辑会内存中修改的数据写回数据库，并拿到数据库中新数据。

服务的消费者

服务消息者从注册中心拿到提供者列表后，会保存提供者列表到内存和磁盘文件中。这样注册中心宕后消费者可以正常运转，甚至可以在注册中心宕机过程中重启消费者。消费者启动时，发现注册中心不可用，会读取保存在磁盘文件中提供者列表。重试逻辑保证注册中心恢复后，更新信息。

重试延迟策略

上一点的子问题。Dubbo 中碰到有两个相关的场景。

数据库上的活锁

注册中心会定时更新数据库一条记录的时间戳，这样集群中其它的注册中心感知它是存活。过期注册中心和它的相关数据 会被清除。数据库正常时，这个机制运行良好。但是数据库负荷高时，其上的每个操作都会很慢。这就出现：

A 注册中心认为 B 过期，删除 B 的数据。 B 发现自己的数据没有了，重新写入自己的数据的反复操作。这些反复的操作又加重了数据库的负荷，恶化问题。

可以使用下面逻辑：

当 B 发现自己数据被删除时（写入失败），选择等待这段时间再重试。重试时间可以选择指数级增长，如第一次等 1 分钟，第二次 10 分钟、第三次 100 分钟。

这样操作减少后，保证数据库可以冷却（Cool Down）下来。

Client 重连注册中心

当一个注册中心停机时，其它的 Client 会同时接收事件，而去重连另一个注册中心。Client 数量相对比较多，会对注册中心造成冲击。避免方法可以是 Client 重连时随机延时 3 分钟，把重连分散开。

防痴呆设计

> http://javatar.iteye.com/blog/804187

最近有点痴呆，因为解决了太多的痴呆问题。服务框架实施面超来超广，已有 50 多个项目在使用，每天都要去帮应用查问题，来来回回，发现大部分都是配置错误，或者重复的文件或类，或者网络不通等，所以准备在新版本中加入防痴呆设计。估且这么叫吧，可能很简单，但对排错速度还是有点帮助，希望能抛砖引玉，也希望大家多给力，想出更多的防范措施共享出来。

检查重复的jar包

最痴呆的问题，就是有多个版本的相同jar包，会出现新版本的 A 类，调用了旧版本的 B 类，而且和JVM加载顺序有关，问题带有偶然性，误导性，遇到这种莫名其妙的问题，最头疼，所以，第一条，先把它防住，在每个 jar 包中挑一个一定会加载的类，加上重复类检查，给个示例：

```
static {
    Duplicate.checkDuplicate(Xxx.class);
}
```

检查重复工具类：

```
public final class Duplicate {

    private Duplicate() {}

    public static void checkDuplicate(Class cls) {
        checkDuplicate(cls.getName().replace('.', '/') + ".class"
);
    }

    public static void checkDuplicate(String path) {
        try {
            // 在ClassPath搜文件
            Enumeration urls = Thread.currentThread().getContext
ClassLoader().getResources(path);
            Set files = new HashSet();
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                if (url != null) {
                    String file = url.getFile();
                    if (file != null &amp;&amp; file.length() &g
t; 0) {
                        files.add(file);
                    }
                }
            }
            // 如果有多个，就表示重复
            if (files.size() &gt; 1) {
                logger.error("Duplicate class " + path + " in "
+ files.size() + " jar " + files);
            }
        } catch (Throwable e) { // 防御性容错
            logger.error(e.getMessage(), e);
        }
    }

}
```

检查重复的配置文件

配置文件加载错，也是经常碰到的问题。用户通常会和你说：“我配置的很正确啊，不信我发给你看下，但就是报错”。然后查一圈下来，原来他发过来的配置根本没加载，平台很多产品都会在 classpath 下放一个约定的配置，如果项目中有多个，通常会取JVM加载的第一个，为了不被这么低级的问题折腾，和上面的重复jar包一样，在配置加载的地方，加上：

```
Duplicate.checkDuplicate("xxx.properties");
```

检查所有可选配置

必填配置估计大家都会检查，因为没有的话，根本没法运行。但对一些可选参数，也应该做一些检查，比如：服务框架允许通过注册中心关联服务消费者和服务提供者，也允许直接配置服务提供者地址点对点直连，这时候，注册中心地址是可选的，但如果没有配点对点直连配置，注册中心地址就一定要配，这时候也要做相应检查。

异常信息给出解决方案

在给应用排错时，最怕的就是那种只有简单的一句错误描述，啥信息都没有的异常信息。比如上次碰到一个 Failed to get session 异常，就这几个单词，啥都没有，哪个 session 出错? 什么原因 Failed? 看了都快疯掉，因是线上环境不好调试，而且有些场景不是每次都能重现。异常最基本要带有上下文信息，包括操作者，操作目标，原因等，最好的异常信息，应给出解决方案，比如上面可以给出："从 10.20.16.3 到 10.20.130.20:20880 之间的网络不通，请在 10.20.16.3 使用 telnet 10.20.130.20 20880 测试一下网络，如果是跨机房调用，可能是防火墙阻挡，请联系 SA 开通访问权限" 等等，上面甚至可以根据 IP 段判断是不是跨机房。另外一个例子，是 spring-web 的 context 加载，如果在 getBean 时 spring 没有被启动，spring 会报一个错，错误信息写着：请在 web.xml 中加入: `<listener>...` `<init-param>...` ，多好的同学，看到错误的人复制一下就完事了，我们该学学。可以把常见的错误故意犯一遍，看看错误信息能否自我搞定问题， 或者把平时支持应用时遇到的问题及解决办法都写到异常信息里。

日志信息包含环境信息

每次应用一出错，应用的开发或测试就会把出错信息发过来，询问原因，这时候我都会问一大堆套话，用的哪个版本呀？是生产环境还是开发测试环境？哪个注册中心呀？哪个项目中的？哪台机器呀？哪个服务？累啊，最主要的是，有些开发或测试人员根本分不清，没办法，只好提供上门服务，浪费的时间可不是浮云，所以，日志中最好把需要的环境信息一并打进去，最好给日志输出做个包装，统一处理掉，免得忘了。包装Logger接口如：

```
public void error(String msg, Throwable e) {
    delegate.error(msg + " on server " + InetAddress.getLocalHos
t() + " using version " + Version.getVersion(), e);
}
```

获取版本号工具类：

```
public final class Version {

    private Version() {}

    private static final Logger logger = LoggerFactory.getLogger
(Version.class);

    private static final Pattern VERSION_PATTERN = Pattern.compi
le("([0-9][0-9\\.\\-]*)\\.jar");

    private static final String VERSION = getVersion(Version.cla
ss, "2.0.0");

    public static String getVersion(){
        return VERSION;
    }

    public static String getVersion(Class cls, String defaultVer
sion) {
        try {
            // 首先查找MANIFEST.MF规范中的版本号
            String version = cls.getPackage().getImplementationV
```

```
ersion();
            if (version == null || version.length() == 0) {
                version = cls.getPackage().getSpecificationVersi
on();
            }
            if (version == null || version.length() == 0) {
                // 如果MANIFEST.MF规范中没有版本号，基于jar包名获取版
本号
                String file = cls.getProtectionDomain().getCodeS
ource().getLocation().getFile();
                if (file != null &amp;&amp; file.length() &gt; 0
 &amp;&amp; file.endsWith(".jar")) {
                    Matcher matcher = VERSION_PATTERN.matcher(fi
le);
                    while (matcher.find() &amp;&amp; matcher.gro
upCount() &gt; 0) {
                        version = matcher.group(1);
                    }
                }
            }
            // 返回版本号，如果为空返回缺省版本号
            return version == null || version.length() == 0 ? de
faultVersion : version;
        } catch (Throwable e) { // 防御性容错
            // 忽略异常，返回缺省版本号
            logger.error(e.getMessage(), e);
            return defaultVersion;
        }
    }

}
```

kill 之前先 dump

每次线上环境一出问题，大家就慌了，通常最直接的办法回滚重启，以减少故障时间，这样现场就被破坏了，要想事后查问题就麻烦了，有些问题必须在线上的大压力下才会发生，线下测试环境很难重现，不太可能让开发或 Appops 在重启前，先

手工将出错现场所有数据备份一下，所以最好在 kill 脚本之前调用 dump，进行自动备份，这样就不会有人为疏忽。dump脚本示例：

```
JAVA_HOME=/usr/java
OUTPUT_HOME=~/output
DEPLOY_HOME=`dirname $0`
HOST_NAME=`hostname`

DUMP_PIDS=`ps  --no-heading -C java -f --width 1000 | grep "$DEP
LOY_HOME" |awk '{print $2}'`
if [ -z "$DUMP_PIDS" ]; then
    echo "The server $HOST_NAME is not started!"
    exit 1;
fi

DUMP_ROOT=$OUTPUT_HOME/dump
if [ ! -d $DUMP_ROOT ]; then
    mkdir $DUMP_ROOT
fi

DUMP_DATE=`date +%Y%m%d%H%M%S`
DUMP_DIR=$DUMP_ROOT/dump-$DUMP_DATE
if [ ! -d $DUMP_DIR ]; then
    mkdir $DUMP_DIR
fi

echo -e "Dumping the server $HOST_NAME ...\c"
for PID in $DUMP_PIDS ; do
    $JAVA_HOME/bin/jstack $PID > $DUMP_DIR/jstack-$PID.dump 2>&1

    echo -e ".\c"
    $JAVA_HOME/bin/jinfo $PID > $DUMP_DIR/jinfo-$PID.dump 2>&1
    echo -e ".\c"
    $JAVA_HOME/bin/jstat -gcutil $PID > $DUMP_DIR/jstat-gcutil-$
PID.dump 2>&1
    echo -e ".\c"
    $JAVA_HOME/bin/jstat -gccapacity $PID > $DUMP_DIR/jstat-gcca
pacity-$PID.dump 2>&1
    echo -e ".\c"
    $JAVA_HOME/bin/jmap $PID > $DUMP_DIR/jmap-$PID.dump 2>&1
```

```
    echo -e ".\c"
    $JAVA_HOME/bin/jmap -heap $PID > $DUMP_DIR/jmap-heap-$PID.du
mp 2>&1
    echo -e ".\c"
    $JAVA_HOME/bin/jmap -histo $PID > $DUMP_DIR/jmap-histo-$PID.
dump 2>&1
    echo -e ".\c"
    if [ -r /usr/sbin/lsof ]; then
    /usr/sbin/lsof -p $PID > $DUMP_DIR/lsof-$PID.dump
    echo -e ".\c"
    fi
done
if [ -r /usr/bin/sar ]; then
/usr/bin/sar > $DUMP_DIR/sar.dump
echo -e ".\c"
fi
if [ -r /usr/bin/uptime ]; then
/usr/bin/uptime > $DUMP_DIR/uptime.dump
echo -e ".\c"
fi
if [ -r /usr/bin/free ]; then
/usr/bin/free -t > $DUMP_DIR/free.dump
echo -e ".\c"
fi
if [ -r /usr/bin/vmstat ]; then
/usr/bin/vmstat > $DUMP_DIR/vmstat.dump
echo -e ".\c"
fi
if [ -r /usr/bin/mpstat ]; then
/usr/bin/mpstat > $DUMP_DIR/mpstat.dump
echo -e ".\c"
fi
if [ -r /usr/bin/iostat ]; then
/usr/bin/iostat > $DUMP_DIR/iostat.dump
echo -e ".\c"
fi
if [ -r /bin/netstat ]; then
/bin/netstat > $DUMP_DIR/netstat.dump
echo -e ".\c"
fi
```

```
echo "OK!"
```

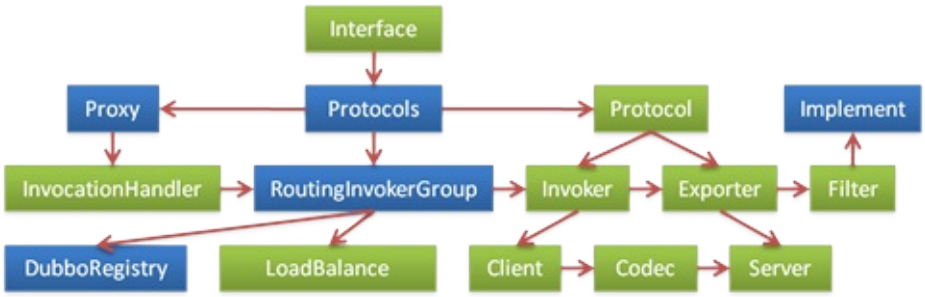
扩展点重构

> http://javatar.iteye.com/blog/1041832

随着服务化的推广，网站对Dubbo服务框架的需求逐渐增多，Dubbo 的现有开发人员能实现的需求有限，很多需求都被 delay，而网站的同学也希望参与进来，加上领域的推动，所以平台计划将部分项目对公司内部开放，让大家一起来实现，Dubbo 为试点项目之一。

既然要开放，那 Dubbo 就要留一些扩展点，让参与者尽量黑盒扩展，而不是白盒的修改代码，否则分支，质量，合并，冲突都会很难管理。

先看一下 Dubbo 现有的设计：



这里面虽然有部分扩展接口，但并不能很好的协作，而且扩展点的加载和配置都没有统一处理，所以下面对它进行重构。

第一步，微核心，插件式，平等对待第三方

即然要扩展，扩展点的加载方式，首先要统一，微核心+插件式，是比较能达到 OCP 原则的思路。

由一个插件生命周期管理容器，构成微核心，核心不包括任何功能，这样可以确保所有功能都能被替换，并且，框架作者能做到的功能，扩展者也一定要能做到，以保证平等对待第三方，所以，框架自身的功能也要用插件的方式实现，不能有任何硬编码。

通常微核心都会采用 Factory、IoC、OSGi 等方式管理插件生命周期。考虑 Dubbo 的适用面，不想强依赖 Spring 等 IoC 容器。自己造一个小的 IoC 容器，也觉得有点过度设计，所以打算采用最简单的 Factory 方式管理插件。

最终决定采用的是 JDK 标准的 SPI 扩展机制，参见：`java.util.ServiceLoader`，也就是扩展者在 jar 包的 `META-INF/services/` 目录下放置与接口同名的文本文件，内容为接口实现类名，多个实现类名用换行符分隔。比如，需要扩展 Dubbo 的协议，只需在 xxx.jar 中放置文件：`META-INF/services/com.alibaba.dubbo.rpc.Protocol`，内容为 `com.alibaba.xxx.XxxProtocol`。Dubbo 通过 ServiceLoader 扫描到所有 Protocol 实现。

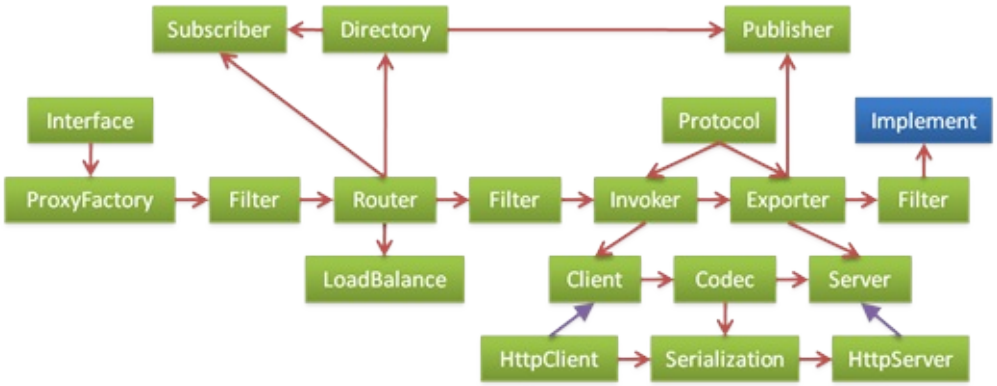
并约定所有插件，都必须标注：`@Extension("name")`，作为加载后的标识性名称，用于配置选择。

第二步，每个扩展点只封装一个变化因子，最大化复用

每个扩展点的实现者，往往都只是关心一件事，现在的扩展点，并没有完全分离。比如：Failover, Route, LoadBalance, Directory 没有完全分开，全由 RoutingInvokerGroup 写死了。

再比如，协议扩展，扩展者可能只是想替换序列化方式，或者只替换传输方式，并且 Remoting 和 Http 也能复用序列化等实现。这样，需为传输方式，客户端实现，服务器端实现，协议头解析，数据序列化，都留出不同扩展点。

拆分后，设计如下：



第三步，全管道式设计，框架自身逻辑，均使用截面拦截实现

现在很多的逻辑，都是放在基类中实现，然后通过模板方法回调子类的实现，包括：local, mock, generic, echo, token, accesslog, monitor, count, limit 等等，可以全部拆分使用 Filter 实现，每个功能都是调用链上的一环。 比如：(基类模板方法)

```
public abstract AbstractInvoker implements Invoker {

    public Result invoke(Invocation inv) throws RpcException {
        // 伪代码
        active ++;
        if (active > max)
            wait();

        doInvoke(inv);

        active --;
        notify();
    }

    protected abstract Result doInvoke(Invocation inv) throws Rp
cException

}
```

改成：(链式过滤器)

```
public abstract LimitFilter implements Filter {

    public Result invoke(Invoker chain, Invocation inv) throws R
pcException {
         // 伪代码
        active ++;
        if (active > max)
            wait();

        chain.invoke(inv);

        active --;
        notify();
    }

}
```

第四步，最少概念，一致性概念模型

保持尽可能少的概念，有助于理解，对于开放的系统尤其重要。另外，各接口都使用一致的概念模型，能相互指引，并减少模型转换，

比如，Invoker 的方法签名为:

```
Result invoke(Invocation invocation) throws RpcException;
```

而 Exporter 的方法签名为:

```
Object invoke(Method method, Object[] args) throws Throwable;
```

但它们的作用是一样的，只是一个在客户端，一个在服务器端，却采用了不一样的模型类。

再比如，URL 以字符串传递，不停的解析和拼装，没有一个 URL 模型类， 而 URL 的参数，却时而 Map, 时而 Parameters 类包装，

```
export(String url)
createExporter(String host, int port, Parameters params);
```

使用一致模型：

```
export(URL url)
createExporter(URL url);
```

再比如，现有的：Invoker, Exporter, InvocationHandler, FilterChain 其实都是 invoke 行为的不同阶段，完全可以抽象掉，统一为 Invoker，减少概念。

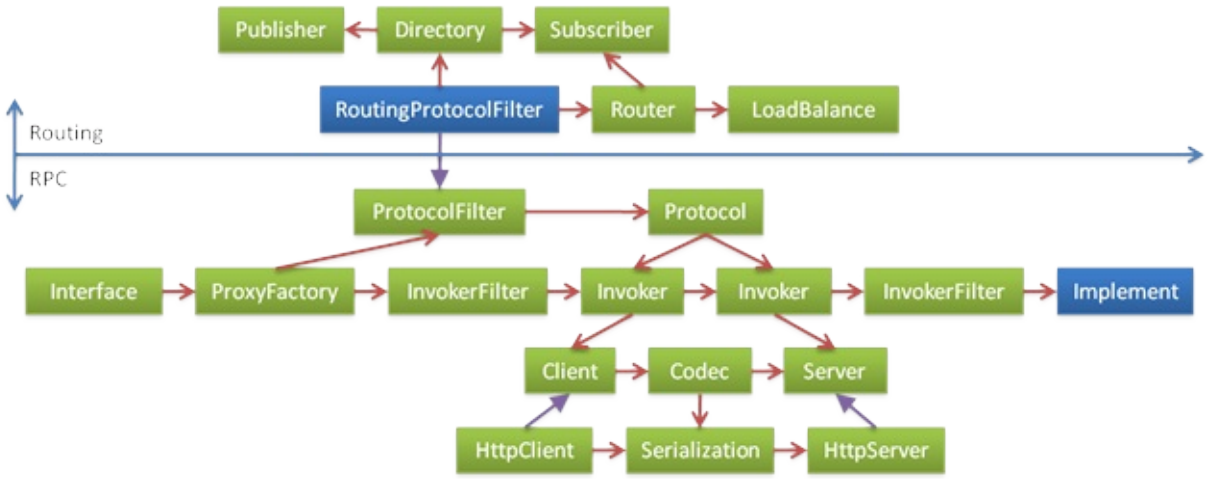
第五步，分层，组合式扩展，而不是泛化式扩展

原因参见：谈谈扩充式扩展与增量式扩展。

泛化式扩展指：将扩展点逐渐抽象，取所有功能并集，新加功能总是套入并扩充旧功能的概念。

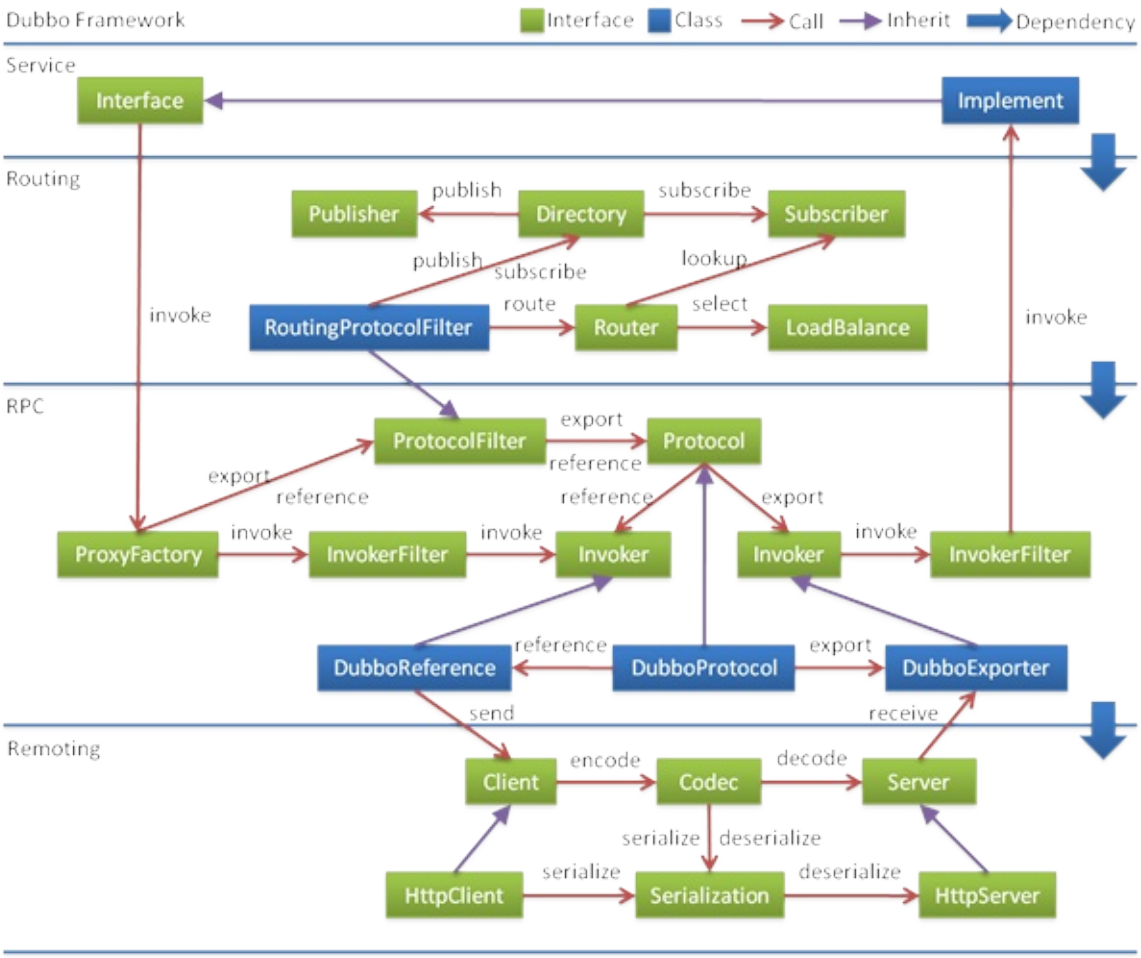
组合式扩展指：将扩展点正交分解，取所有功能交集，新加功能总是基于旧功能之上实现。

上面的设计，不自觉的就将 Dubbo 现有功能都当成了核心功能。上面的概念包含了 Dubbo 现有 RPC 的所有功能，包括：Proxy, Router, Failover, LoadBalance, Subscriber, Publisher, Invoker, Exporter, Filter 等， 但这些都是核心吗？踢掉哪些，RPC 一样可以 Run？而哪些又是不能踢掉的？基于这样考虑，可以将 RPC 分解成两个层次，只是 Protocol 和 Invoker 才是 RPC 的核心。其它，包括 Router, Failover, Loadbalance, Subscriber, Publisher 都不核心，而是 Routing。所以，将 Routing 作为 Rpc 核心的一个扩展，设计如下：



第六步，整理，梳理关系

整理后，设计如下：



版本管理

新功能的开发 和 稳定性的提高 对产品都很重要。但是添加新功能会影响稳定性，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。
- 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。

`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：

- `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。
- `2.4.x` 不再开发，应用碰到 Bug 让直接升级。（这个称为“夕阳条款”）
- 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。这是一个假命题，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，三方库版本长时间不升级，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。

贡献

流程

- 如果是扩展功能，直接新增工程，黑盒依赖 Dubbo 进行扩展。
- 如果是改 BUG，或修改框架本身，可以从 Dubb 的 GitHub 上 Fork 工程。
- 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%
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%
JTA/XA分布式事务	拦截扩展	高	未认领	待定	待定	0%
Thrift	协议扩展	高	开发完成	闫刚	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%
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%
XSocket	传输扩展	低	未认领	待定	待定	0%

CGLib	动态代理扩展	低	未认领	待定	待定	0%
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%
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%
	监控					

Cacti	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%
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%

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%

检查列表

发布前 checklist

- jira ticket 过一遍
- svn change list
- ticket 关联 code
- test code
- find bugs

修复时 checklist

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

Partner Check

- Partner 以用户的方式运行一下功能
- 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）
- Owner 向 Partner 讲述一遍实现。

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

直接暴露服务：

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

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

向注册中心暴露服务：

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

获取注册中心：

```
url.setProtocol(url.getParameter("registry", "dubbo"))
GETREGISTRY(dubbo://registry-address/com.alibaba.dubbo.registry.RegistrySerevice)
```

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

从注册中心订阅服务：

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

获取注册中心：

```
url.setProtocol(url.getParameter("registry", "dubbo"))
GETREGISTRY(dubbo://registry-address/com.alibaba.dubbo.registry.
RegistrySerevice)
```

订阅服务地址：

```
url.addParameters(url.getParameterAndDecoded("refer"))
SUBSCRIBE(dubbo://registry-address/com.xxx.XxxService?version=1.
0.0)
```

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

注册中心路由规则推送：

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

获取路由器：

```
url.setProtocol(url.getParameter("router", "script"))
GETROUTE(script://registry-address/com.xxx.XxxService?type=js&ru
le=ENCODE(function{...}))
```

5. 从文件加载路由规则

从文件加载路由规则：

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

获取路由器：

```
url.setProtocol(url.getParameter("router", "script")).addParamet
er("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>$Adpative 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 `new String[] {"yyy.invoker.wrapper"}`

Callback 功能

1. 参数回调

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。

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

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。

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。

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。

技术兼容性测试

Dubbo 的协议，通讯，序列化，注册中心，负载均策等扩展点，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状性，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

> TODO

Registry TCK

> TODO