

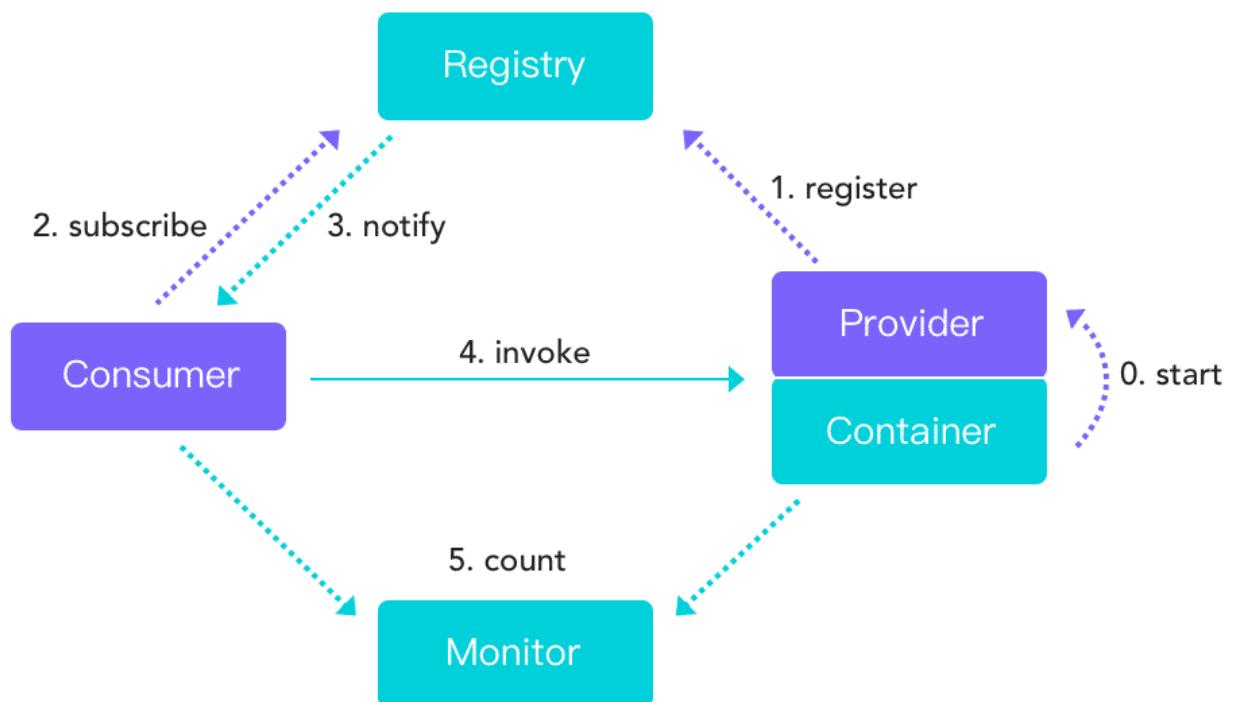
# Diving-in-Dubbo

“ Apache Dubbo™ 是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。

翻开Dubbo的官方文档的首页，首先映入眼帘的是如上一段醒目的文字，还有右边如下关于Dubbo架构的表意图示：

## Dubbo Architecture

..... init    ..... async    → sync



Dubbo已经是 Apache 基金会的顶级项目，甚至一些后起的框架也在汲取其精华、借鉴它的设计。在 Java 开发社区，它有着大量的拥趸，甚至吸引了很大一个研读源码的群体。在开发者眼中，它被做一个微服务框架对待，这早已超过 RPC 框架的范畴。它的成功离不开如下本身丰富的特性支持，更离不开阿里这个千亿规模的企业对开源文化的拥抱和贡献。



### 面向接口代理的高性能RPC调用

提供高性能的基于代理的远程调用能力，服务以接口为粒度，为开发者屏蔽远程调用底层细节。



### 智能负载均衡

内置多种负载均衡策略，智能感知下游节点健康状况，显著减少调用延迟，提高系统吞吐量。



### 服务自动注册与发现

支持多种注册中心服务，服务实例上下线实时感知。



### 高度可扩展能力

遵循微内核+插件的设计原则，所有核心能力如Protocol、Transport、Serialization被设计为扩展点，平等对待内置实现和第三方实现。



### 运行期流量调度

内置条件、脚本等路由策略，通过配置不同的路由规则，轻松实现灰度发布，同机房优先等功能。



### 可视化的服务治理与运维

提供丰富服务治理、运维工具：随时查询服务元数据、服务健康状态及调用统计，实时下发路由策略、调整配置参数。

使用 Java 做企业级开发离不开大量优秀开源框架的支持，然而早年间这些框架大部分是中国本土以外的开发者贡献的，国人鲜有贡献。然而随着阿里在商业上的成功，广袤的中国市场、海量的网络请求，传统开源件也渐渐变得难负重任。阿里去IOE的进程中，投入重金网罗并培养了大量优秀人才，他们博采众长，吸纳了大量国际开源社区关于分布式开发的最佳实践，开发了不少优秀的分布式基础框架或中间件。它们经历了阿里这个生态环境的捶打，在实战中被烤炼，经受了高负荷的考验，阿里秉持开放心态将其中一些开源了，Dubbo就是其中之一。阿里开源的举动为一些开发者带来了福音，同时也带动了国内其他一些互联网公司对开源的拥抱。自此，越来越多的国人开发者涌现在开源社区，早年不贡献只索取的态势也在渐渐改变，而这也促进了国内IT方面的工程水平和能力，也直接促进了这几年中国移动互联网的高速发展。

## Why ?

《深潜Dubbo》于本人而言，是一本大部头的技术书籍。类似剖析Dubbo源码的网文汗牛充栋，为什么我坚持要写？

其实在写此书之前，我有过几年的分布式和微服务的开发经验。然而就像绝大部分开发者一样，重心更多放在业务逻辑开发上，对框架的实现原理却鲜有了解，大部分时候能够用得很灵活，但遇到问题后，却没法快速定位出根源因素。加上国内IT界风行的加班文化，几乎很难挤出时间去深度了解一门技术的底层实现原理，更别说读懂源码，数次的尝试，数次的不了了之。开发框架就像一把双刃剑，给业务开发人员提供了极大便利的同时，其底层运行机制或实现原理却像幽灵一般，拒我们于千里之外。技术人员本身的言语木讷，加上本人对刷题的天生抗拒，没法讲透的机制原理，所有这些让自己陷入一种技术人员的穷人困境。

于喜爱编程的人而言，能够成为技术人是一件非常幸福的事情，一个人，一台电脑，他就能享受着创造的乐趣。我想，女娲所在的世界虽然孤苦清冷，但是能够捏泥为人，她必定是幸福的。早些年，个人认为技术的存在意义是用于解决业务问题的，因而将重心偏于应用开发，讲究代码的可扩展性和适配能力，享受着编码所带来的乐趣，然而却天真的想依靠参与产品开发完成高级技能的实践与内化，忽略了深度的理论积累，职业上因此没法及时获得突破，时常怀疑自己是否“叶公好龙”。

Java 是一门面向对象的编程语言，换言之，它是很讲究编码结构的，比如业界所提倡的面向对象编程和设计的 S.O.L.I.D原则

(<https://learnku.com/articles/4160/solid-notes-on-object-oriented-design-and-programming-oodoop>)

，其实设计模式正是遵守这些原则所总结出来的最佳实践。优良的编码结构是创建一个易于维护和扩展的软件系统的基础，即使系统在不断的迭代中，也不会因此而变得混乱。曾经见过一些业务开发人员，包括来自某些大厂的，所编业务代码相当凌乱，他们甚至不知道什么是重构。后续接力的开发人员功底再扎实、能力再强，在这种凌乱面前适配变化也会显得无所适从，即便拥有好的调试方案，也会如履薄冰，生怕锅从天上来。

一门技术或者一个框架都是为了解决某类通用问题而存在的，为了适配问题变化的多样性，有着各种各样的技能点。然而通常，这些技能点，于大多数只能接触到增删改查操作的业务开发人员而言，只有不到20%会在日常编码中被我们用到，书本上学到的其它技能点很多是没法亲自去实践一遍的，比方说 Spring 框架多处用到的代理机制。

## How ?

一个优秀的框架，它必定是遵守 S.O.L.I.D原则 的，有着优良的代码结构和极强的可扩展能力，同时为了框架的强大也会应用到一些 高级技能 。Dubbo作为一个 RPC 框架， 微内核+插件 的设计原则保证它了灵活的扩展性，还应用了诸多分布式的技术，经历过大厂生产环境的严苛验证，代码简短精悍，诸如此类等，让我有了对它探索的欲望。

框架源码很大程度上更像一部剧本，是由很多零散的细节有机组合“拼凑”而成的，单就某个细节，我们很难理解它的存在，只有把它放大到一个更为宏大的系统层面，纵向需要知道它的前世今生，横向需要察觉它与周遭的联系，只有这样我们才能通晓其存在的形式与意义。而这一纵一横，却急速加重了人脑的认知负荷。这种负荷普通人是难以承受的，但是厉害的编剧，即便撰写过多部剧本，它们依然可以部部精彩，没有重样。这其中的奥秘就在于，尽管细节浩如烟海，但细节的成因或者细节间的组合关系却逃脱不了有限的几个套路。换言之，一个厉害的编剧或者导演，必定是一个套路梳理和总结的高手。

有人说如果想要了解一片海域的生态情况，最好的方式是去研究它的珊瑚礁，它会带领你了解到所有的秘密。然而，假如想要了解该海域状况的是一只鱼，那么势必会困难重重，因为据说鱼儿的记忆只有区区5分钟。其实于我们大多数人而言，阅读源码是一件非常痛苦的事情，我们的处境就像前面所说的鱼儿一样，并不会好太多。

然而，这个过程却又是十分必要的，它是我们熟练掌握某种技术的必经之路，一种技能或者设计手段，只有内化了，才足矣自诩掌握了，就像学会游泳一样，此后，无论间隔多久，跳进水里，你依然会游，只是技巧上可能有些生疏了。因此就像编剧一样，我们阅读源码的目的不是为了知道，而是为了让自己浸润在那个语境中，熟知或者总结出一个个能够成为剧作的套路，让这些渐渐内化的套路为己所用。

有人会说，写得好的书籍那么多，何必大费周章去研读源码了？也许你和我有过一样的感觉，曾经接触过很多介绍技术的书籍，会把实现的机制或者原理描述得很生动易懂，但是当我们放下书本去思考到底是怎么实现的时，却往往没法想出个所以然来，真是拿起时“学富五车”，放下后却“江郎才尽”，难免

会陷入智不如人的自我否定中。深入原因是组成一个系统的所有细节或技能点之间有着纵横交错的关系，这些书籍着重讲了单个背后实现的机制，却往往忽略了它和周边的关系，也忽视了不同读者理解的差异性。

曾经多次阅读源码的尝试，也让我意识到读懂不等于通透，内化最好的方式是尝试去尝试剖析所以然，然后将其写下来，于是这进一步促成我去写《深潜Dubbo》一书。遗憾的是，日常自然语言中同样一句话，不同的人会有不同的表达，理解的意思也不尽相同，不像程序源码，能够精确地表达某个过程或者意图，几乎不存在二义性，因此请原谅我没法给您不含大量源码的《深潜Dubbo》。另外，于后进生而言，学生时代老师眼中所谓“显然”，往往是他们所跨不过去的一道坎，一丁点的懵懵懂懂却常常酿成大面积的知识短路，因此本书中关于机制的剖析中，我除了详细阐述实现过程外，还会尽量在该过程不被扰乱的前提下，将相关源码打散重组，剔除不相干部分，然后再呈现给读者您。

## Then ?

有朋友曾建议我将构成《深潜Dubbo》的序列文章以公众号的形式分享出来，在仔细思考之后，个人觉得不妥，理由是阅读源码是一个思绪逐渐展开需要慢慢咀嚼的过程，并不适合在移动场景中进行。另外，定期不定期的放出一些 标题党 式的文章来，无形中会给大家带来某种焦虑感，IT人所在的行业，本身就节奏快压力大，我应诚意对待自己读者，尽量少地打扰您们。

通读框架源码，尝试过的读者会知道，这个过程会比较辛苦，会因为各种原因没法坚持下去。不过此刻，我却为您感到开心，既然您能读到这里，大概率说明您也遇到过和我同样的困境，或者不希望走上和我同样的弯路，也在积极寻求突破，期待自己能够获得真知，而不仅仅是想停留在“知道”这个层面。《深潜Dubbo》一书的创作过程中，我一直秉持着作品心态，期待能够帮助到更多的读者，带着他们还有您，一起去克服通读源码时遇到的重重困难。

最后，非常期待您的反馈和支持，它将鼓励我继续创作，也会让此书变得更加完善。如果您觉得本书能帮上自己，希望同我一起坚持完成对Dubbo框架源码的探索，那么我鼓励您尽量多地给本身赞赏，可能它本身并不值那个价值，但坚持无价，赞赏给了您还有我坚持下去的勇气和决心。况且您不必像我一样辞掉工作专心研读源码，只需挤出下班之后的一点业余时间随我梳理的思路慢慢品味，相信您会不虚此行。



静以储势·Shuke 的赞赏码

如果此书能为您辉煌的职业生涯贡献一点点力量的话，那将是我人生莫大的荣幸。

# Dubbo中的类管理 和 ClassLoader

---

作为一个RPC开放框架，Dubbo是非常值得学习的，阿里优秀的IT工程师贡献着宝贵的经验，万丈高楼平地起，复杂的系统离不开一砖一瓦的参与组成，总是由简单的元素通过有机排列组合构建而成的，其中一些辅助构建系统的工具或组件不可或缺，学习他们是快速理解Dubbo源码关键，也能够辅助开发者自身提高Java编码的基础能力，而其他更加偏顶层的内容则有利于帮助更加快速获知如何实现一个分布式系统的全貌。

这篇文章则结合ClassLoader分析Dubbo中相关的工具代码。

## Java基础：ClassLoader

Java中需要利用ClassLoader，将由Java源文件编译得到class字节码文件加载到JVM虚拟机中，JVM并不是一次性加载所需要的全部类的，它是按需延迟加载。JDK内置的URLClassLoader可以用于加载位于网络上静态文件服务器提供的jar包和class文件。它不但可以加载远程类库，还可以加载本地路径的类库，取决于构造器中不同的地址形式。*ExtensionClassLoader* 和 *AppClassLoader* 都是 *URLClassLoader* 的子类，它们都是从本地文件系统里加载类库。

```
some.java → (javac) → some.class
```

### Java语言系统自带的3个类加载器

**BootstrapClassLoader** 启动类加载器，C++实现，加载核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等，可指定jvm启动参数 -Xbootclasspath 改变加载目录。

**ExtentionClassLoader** 扩展类加载器，加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件，-Djava.ext.dirs 参数加载指定目录。

**AppClassLoader** 也称**SystemAppClass** 加载当前应用的classpath的所有类，`ClassLoader.getSystemClassLoader()` 可直接获取。

### 加载顺序

类的基本加载顺序：**Appclass Loader ← Extention ClassLoader ← Bootstrap ClassLoader**

除了启动类加载器之外，其他的类加载器都是 `java.lang.ClassLoader` 的子类，因此有对应的 Java 对象。这些类加载器需要先由另一个类加载器，比如说启动类加载器，加载至 Java 虚拟机中，方能执行类加载。

### 双亲委派模型

一个类加载器查找class和resource时，是通过“委托模式”进行的，它首先调用 `loadClass()` 判断这个class是不是已经加载成功，如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap ClassLoader，如果Bootstrap classloader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身调用 `findClass()` 去加载目标类，最后调用 `defineClass()` 将字节码转换成Class对象。这种机制就叫做双亲委派。

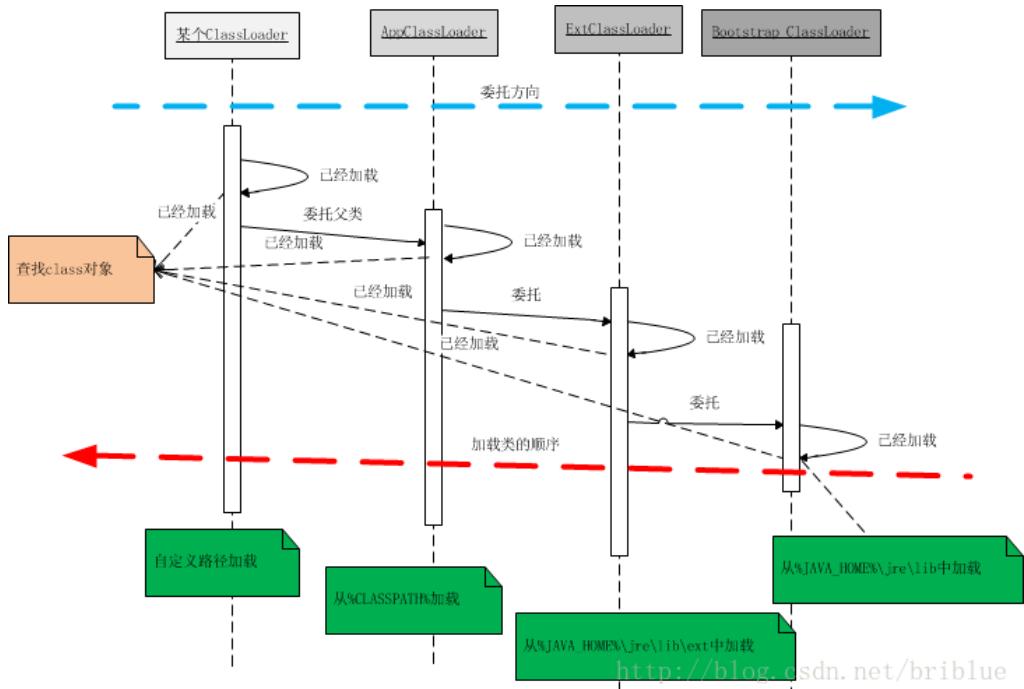


图 1: 双亲委派模型

其部署基本实现伪代码如下:

```

class ClassLoader {
    // 加载入口, 定义了双亲委派规则
    Class loadClass(String name) {
        // 是否已经加载了
        Class t = this.findFromLoaded(name);
        if(t == null) {
            // 交给双亲
            t = this.parent.loadClass(name);
        }
        if(t == null) {
            // 双亲都不行, 只能靠自己了
            t = this.findClass(name);
        }
        return t;
    }

    // 交给子类自己去实现
    Class findClass(String name) {
        throw ClassNotFoundException();
    }

    // 组装Class对象
    Class defineClass(byte[] code, String name) {
        return buildClassFromCode(code, name);
    }
}

class CustomClassLoader extends ClassLoader {

    Class findClass(String name) {
        // 寻找字节码
        byte[] code = findCodeFromSomewhere(name);
        // 组装Class对象
        return this.defineClass(code, name);
    }
}

```

## Class.forName vs ClassLoader.loadClass

`forName()` 方法同样也是使用调用者 Class 对象的 `ClassLoader` 来加载目标类，另外它还有个多参数版本，可以指定 `ClassLoader`。

```
Class<?> forName(String name)
Class<?> forName(String name, boolean initialize, ClassLoader cl)
```

JAVA

和 `ClassLoader.loadClass()` 的区别是它可以获取原生类型的 Class，而 `ClassLoader.loadClass()` 则会报错。

```
Class<?> x = Class.forName("[I");
System.out.println(x);

x = ClassLoader.getSystemClassLoader().loadClass("[I");
System.out.println(x);

-----
class [I

Exception in thread "main" java.lang.ClassNotFoundException: [I
...
```

JAVA

## 高级特性 · Thread.contextClassLoader

1. **ClassLoader** 相当于类的命名空间，犹如沙箱，起到了类隔离的作用。同一 `ClassLoader` 里面的类名是唯一的，不同的 `ClassLoader` 可以持有同名的类。
2. 不同的 **ClassLoader** 加载的全名一样的 **Class** 类，实际上是不同的类，“类加载器 + 全类名”得完全一样。
3. 双亲委派机制中，parent 具有更高的加载优先级，被其加载的类会被所有子 `ClassLoader` 共享。

上述这些特性归结起来就是共享 & 隔离，结合 **Thread.contextClassLoader** 线程上下文类加载器利用其实现版本隔离，在 Java 社区的一些框架中用的比较广。

```
class Thread {
    ...
    private ClassLoader contextClassLoader;

    public ClassLoader getContextClassLoader() {
        return contextClassLoader;
    }

    public void setContextClassLoader(ClassLoader cl) {
        this.contextClassLoader = cl;
    }
    ...
}
```

JAVA

线程的 **contextClassLoader** 默认是从父线程那里继承过来的，`main` 线程启动时默认设置为 `AppClassLoader`，没特意指定的情况下，此后衍生的线程均自动继承它作为自己的类加载器。这意味着可以跨线程共享类，只要线程同享同一个类加载器，也可以通过设置不同的类加载器做隔离处理。如果我们对业务进行划分，不同的业务使用不同的线程池，线程池内部共享同一个 `contextClassLoader`，线程池之间使用不同的 `contextClassLoader`，就可以很好的起到隔离保护的作用，避免类版本冲突。

在JVM中，类型被定义在一个叫SystemDictionary 的数据结构中，该数据结构接受类加载器和全类名作为参数，返回类型实例。

NOTE

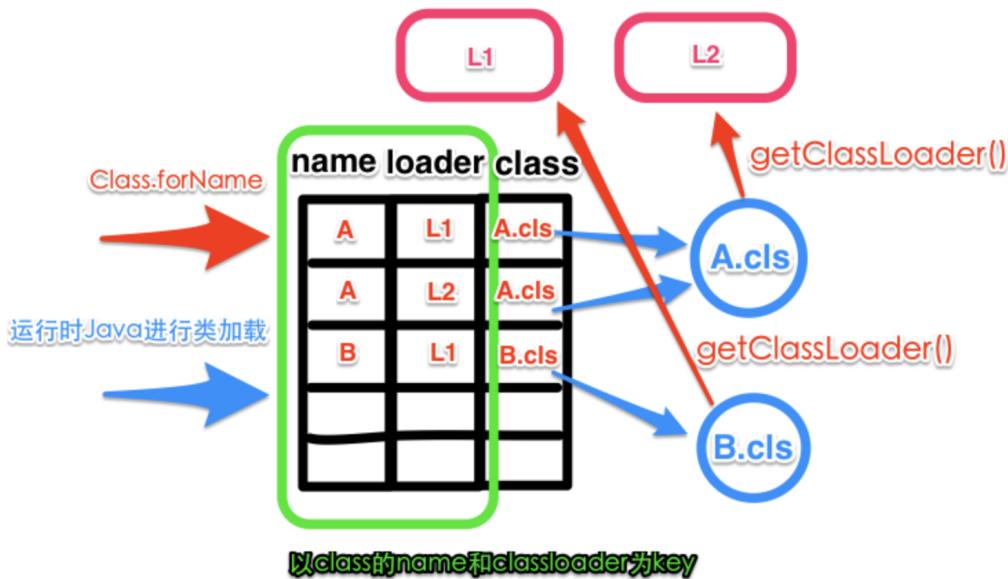


图 1: 类名解析机制

类型加载时，需要传入类加载器loader和需要加载的全类名name，如果在 SystemDictionary 中能够命中一条记录，则返回class 列上对应的类型实例引用，如果无法命中记录，则会调用loader.loadClass(name);进行类型加载。

蚂蚁金服出品：轻量级类隔离框架 **sofa-ark**

## 相关异常

1. **NoClassDefFoundError** JVM或者类加载器实例尝试加载类型的定义，但是该定义却没有找到，影响了执行路径而抛出的异常，编译期找到，而运行期却没有找到，也即当前运行classpath缺少对应的jar包或者\*.class。比如Maven依赖中设置 `<scope>provided</scope>` 找到jar包没有装载到classpath中。
2. **NoSuchMethodError** 代表期望类型确实存在，但是一个不正确的版本被加载了，常见于Maven坐标的变动，使得应用依赖了多个相同内容，不同版本的jar包，以致在运行时选择了非期望的版本。使用`-verbose:class`排查。
3. **LinkageError** 同一个限定名的class类被多个不同的ClassLoader加载后，相互交叉使用导致的类冲突的情况。同一个限定名的class在不同的classLoader 中属于不同的 Class实例，而JVM在加载某一个类时，需要加载所有import进入的Class，这种情况下，如果自定义的classLoader中存在与parentClassLoader 需要加载相同限定名的Class时，就会抛出 `java.Lang.LinkageError`。
4. **ClassCastException** 在传统的双亲委派模型下，这种异常不会发生，但如果使用了一个优先使用自身repository中类型的 **ClassLoader**，并且通过反射赋值 给当前另外一个 **ClassLoader**，则会出现这种异常，如下：

```

CachedClassLoader cl = null;

cl = new CachedClassLoader(
    new URL[] {
        new File("/Users/weipeng2k/.m2/repository/org/apache/mina/mina-core/2.0.7/mina-core-2.0.7.jar")
            .toURI().toURL()
    }, this.getClass().getClassLoader());

try {
    Class<?> klass = cl.loadClass("org.apache.mina.proxy.utils.MD4");
    //等号左边MD4这个类型是使用当前ClassLoader加载的，而右边klass所表示的这个类型则是由自定义的CachedClassLoader所加载的
    MD4 md4 = (MD4) klass.newInstance();
    ...
} catch (Exception ex) {
    throw new RuntimeException(ex);
} finally {
    cl.close();
}

//CachedClassLoader的逻辑如下，注意下述结构破坏了双亲委派机制，先使用自身的逻辑加载类
Class loadClass(String name) {
    try {
        clazz = findClass(name);

        if (clazz != null) {
            return clazz;
        }
    } catch (ClassNotFoundException ex) {

    }
    return super.loadClass(name);
}

```

Alibaba中间件团队出品: **Middleware-Detector**

## ClassLoader.getResource()与getResources()

由类加载器可知，在Java编程中，资源的加载处理的重要性非常明显，某种程度上可以讲\*.class文件本身也是被作为一种资源加载到JVM中的。Java认为 资源是类似图像、音频、文本等的数据，可以通过类代码以独立于代码位置的方式访问。和类的加载逻辑很像，也是优先由parent类加载器获取到目标资源，如果parent为null，则会使用jvm类加载器**BootstrapClassLoader**的路径，若还是没有获取到，则会使用当前 **ClassLoader** 实现的，**findResource()** 或 **findResources()** 进一步查找。其优先顺序如下：

ParentClassLoader → BootstrapClassLoader → CurrentClassLoader

**getResource()** 与 **getResources()** 实现机制基本一样，区别是前者仅返回第一个满足条件的，而后者则返回所有匹配的。为便于理解，以前者为例，先看在 **ClassLoader** 中的实现：

```

public URL getResource(String name) {
    URL url;
    if (parent != null) {
        url = parent.getResource(name);
    } else {
        url = getBootstrapResource(name);
    }
    if (url == null) {
        url = findResource(name);
    }
    return url;
}
/**
 * Find resources from the VM's built-in classloader.
 */
private static URL getBootstrapResource(String name) {
    URLClassPath ucp = getBootstrapClassPath();
    Resource res = ucp.getResource(name);
    return res != null ? res.getURL() : null;
}
/**
 * Finds the resource with the given name. Class loader implementations
 * should override this method to specify where to find resources.
 */
protected URL findResource(String name) {
    return null;
}

```

**NOTE**

`getResource()` 或 `getResources()` 使用的路径相对于当前**ClassLoader**的根目录，他们会匹配压缩到jar的资源文件。在Spring中经常以 `classpath` 或 `classpath*` 作为前缀指定加载资源的目录，区别是前者不能搜索位于jar包内的资源。

## Class.getResource()

在Class类中还有一个类似于 `ClassLoader.getResource()` 的资源查找方法，不同的是，它对传入的名称路径做了一些加工处理，如果入参name是绝对路径以"/"开头，则会将其去掉，否则会在其前面加上类的路径，随后委托加载当前类的**ClassLoader**进一步获取java.net.URL。如下：

```

public class Class<T>{

    public java.net.URL getResource(String name) {
        name = resolveName(name);
        ClassLoader cl = getClassLoader0();
        if (cl==null) {
            // A system class.
            return ClassLoader.getSystemResource(name);
        }
        return cl.getResource(name);
    }

    /**
     * Add a package name prefix if the name is not absolute
     * Remove leading "/" if name is absolute
     */
    private String resolveName(String name) {
        if (name == null) {
            return name;
        }
        if (!name.startsWith("/")) {
            Class<?> c = this;
            while (c.isArray()) {
                c = c.getComponentType();
            }
            String baseName = c.getName();
            int index = baseName.lastIndexOf('.');
            if (index != -1) {
                name = baseName.substring(0, index).replace('.', '/') +
                    "/" + name;
            }
        } else {
            name = name.substring(1);
        }
        return name;
    }
}

```

**IMPORTANT**

获取Class所在根目录的绝对路径

由上推知知，使用 `this.getClass().getResuorce("").getFile()` 可以获得当前类所在的绝对路径，然而打成jar包后由于**security domain**问题，得到的是 null值。解决方法如下：

```
getClass().getProtectionDomain().getCodeSource().getLocation().getPath();
```

## Dubbo中的类管理

### ClassUtils

#### ClassUtils.getClassLoader()

直接调用 `getClass().getClassLoader` 有时无法获得有效的值，`getClassLoader()` 获取一个保证不为空值的**ClassLoader**，源码如下：

```

/**
 * get class loader
 *
 * @param clazz
 * @return class loader
 */
public static ClassLoader getClassLoader(Class<?> clazz) {
    ClassLoader cl = null;
    try {
        cl = Thread.currentThread().getContextClassLoader();
    } catch (Throwable ex) {
        // Cannot access thread context ClassLoader - falling back to system class loader...
    }
    if (cl == null) {
        // No thread context class loader -> use class loader of this class.
        cl = clazz.getClassLoader();
        if (cl == null) {
            // getClassLoader() returning null indicates the bootstrap ClassLoader
            try {
                cl = ClassLoader.getSystemClassLoader();
            } catch (Throwable ex) {
                // Cannot access system ClassLoader - oh well, maybe the caller can live with null...
            }
        }
    }
    return cl;
}

```

由上文我们知道 `Thread.currentThread().getContextClassLoader()` 所代表的作用，使用它是实现线程间类版本隔离的重要手段。

### ClassUtils.forName()

Java自带的 `Class.forName()` 使用上不是很方便，如下一些表达方式是不支持的：

```

Class.forName("byte")
Class.forName("java.lang.String[]")

```

`ClassUtils.forName()` 对其进行了增强处理。首先它使用私有静态变量 `Map<String, Class<?>>` `PRIMITIVE_TYPE_NAME_MAP` 保存了如下关系：

序号	表达式	类型
1	boolean、byte、char、double、float、int、long、short	boolean.class、byte.class、char.class、double.class、float.class、int.class、long.class、short.class
2	[Z、[B、[C、[D、[F、[I、[J、[S	boolean[].class、byte[].class、char[].class、double[].class、float[].class、int[].class、long[].class、short[].class

最后看看如下源码：

```

public static Class<?> forName(String name, ClassLoader classLoader)
    throws ClassNotFoundException, LinkageError {

    //解析基本类型
    Class<?> clazz = resolvePrimitiveClassName(name);
    if (clazz != null) {
        return clazz;
    }

    // "java.lang.String[]" style arrays
    if (name.endsWith(ARRAY_SUFFIX)) {
        String elementClassName = name.substring(0, name.length() - ARRAY_SUFFIX.length());
        Class<?> elementClass = forName(elementClassName, classLoader);
        return Array.newInstance(elementClass, 0).getClass();
    }

    // "[Ljava.lang.String;" style arrays
    int internalArrayMarker = name.indexOf(INTERNAL_ARRAY_PREFIX);
    if (internalArrayMarker != -1 && name.endsWith(";")) {
        String elementClassName = null;
        if (internalArrayMarker == 0) {
            elementClassName = name
                .substring(INTERNAL_ARRAY_PREFIX.length(), name.length() - 1);
        } else if (name.startsWith("["))
            elementClassName = name.substring(1);
        Class<?> elementClass = forName(elementClassName, classLoader);
        return Array.newInstance(elementClass, 0).getClass();
    }

    ClassLoader classLoaderToUse = classLoader;
    if (classLoaderToUse == null) {
        classLoaderToUse = getClassLoader();
    }
    return classLoaderToUse.loadClass(name);
}

```

## Version

在Dubbo中出现好几处如下代码，其目的是根据关键传入类名检查当前classpath下是否存在多个同名类，进而从侧面判别是否加载了多个版本的jar：

```

static {
    // check duplicate jar package
    Version.checkDuplicate(Exchangers.class);
}

```

实现的基本原理是将根据类名获取对应\*.class的其相对于classpath的路径，判别是否有多个这样的文件，如下：

```

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

public static void checkDuplicate(Class<?> cls) {
    checkDuplicate(cls, false);
}

public static void checkDuplicate(String path, boolean failOnError) {
    try {
        // search in caller's classloader
        Set<String> files = getResources(path);
        // duplicated jar is found
        if (files.size() > 1) {
            String error = "Duplicate class " + path + " in " + files.size() + " jar " + files;
            if (failOnError) {
                throw new IllegalStateException(error);
            } else {
                logger.error(error);
            }
        }
    } catch (Throwable e) {
        logger.error(e.getMessage(), e);
    }
}

/**
 * search resources in caller's classloader
 */
private static Set<String> getResources(String path) throws IOException {
    Enumeration<URL> urls = ClassHelper.getCallerClassLoader(Version.class).getResources(path);
    Set<String> files = new HashSet<String>();
    while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        if (url != null) {
            String file = url.getFile();
            if (file != null && file.length() > 0) {
                files.add(file);
            }
        }
    }
    return files;
}

```

---

完结

# Dubbo之SPI动态代码编译

Dubbo的SPI动态扩展点加载中，若当前扩展点中的方法含有 `@Adaptive` 注解，但却不存在相应含有在类上注解了 `@Adaptive` 的具类，Dubbo就会为其动态生成一个扩展点具类，生成源码只是第一步，由源码得到相应的元数据一个`Class<?>`对象才是最关键的一步，也即下述源码：

```
private Class<?> createAdaptiveExtensionClass() {  
    //生成代码  
    String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();  
  
    //获取classloader  
    ClassLoader classLoader = findClassLoader();  
  
    //使用SPI获取用于编译字符串得到类的Compiler  
    Compiler compiler = ExtensionLoader.getExtensionLoader(Compiler.class)  
        .getAdaptiveExtension();  
  
    //完成字符串到Class<?>对象的转换处理  
    return compiler.compile(code, classLoader);  
}
```

此文的主角正是围绕着 `Compiler` 转开的，它的作用是将字符串源码编译成类字节码，再将字节码加载成 `Class<?>` 对象。`Compiler` 是一个扩展点，Dubbo为其提供了两种实现—— `JdkCompiler` 和 `JavassistCompiler`，分别由JDK和Javassist支持，默认使用前者，本文也只介绍前者。

扩展点定义如下：

```
@SPI("javassist")  
public interface Compiler {  
  
    /**  
     * Compile java source code.  
     *  
     * @param code      Java source code  
     * @param classLoader classloader  
     * @return Compiled class  
     */  
    Class<?> compile(String code, ClassLoader classLoader);  
}
```

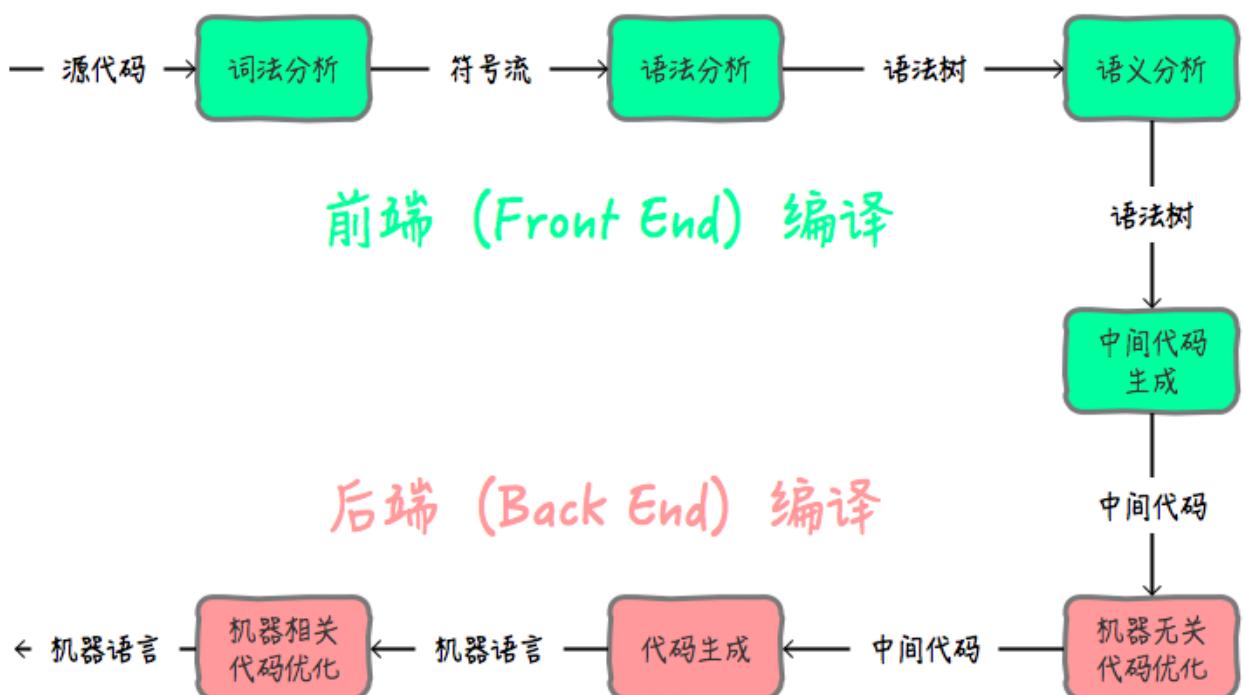
下文将展开讨论的 `JdkCompiler` 除了是 `Compiler` 扩展点实现具类外，实际上扩展自 `AbstractCompiler`，后者存在的主要目的是从入参 `code` 中提取出类的名称，然后再以其作为参数试图使用当前 `ClassLoader` 加载出类对象，如果不存在才会进一步让子类实现的下述抽象方法去编译源码加载出对应类对象。

这里涉及的编译实际上只仅限于Java的前端编译，和后端编译相距甚远。遇到两陌生概念了？莫急，下文会先系统介绍下他们，也就是先搞清楚 **Compiler** 背后到底在干啥。

## Java 编译原理基本介绍

编译，通常是指将源码转换成机器码的过程，但于Java来说，这个过程有点不一样，源码先转换成在JVM能理解的字节码<sup>\*</sup>.class文件，字节码于机器来说是理解不了的，还得进一步由JVM负责将其翻译成机器指令。两个过程都是Java的编译，被归纳成前端编译和后端编译：

- “
- **前端编译**: 主要指与源语言有关但与目标机无关的部分，包括词法分析、语法分析、语义分析与中间代码生成，是将.java文件编译成.class的编译过程。
  - **后端编译**: 主要指与目标机有关的部分，包括代码优化和目标代码生成等，是将.class文件翻译成机器指令的编译过程。



**NOTE**

和优化相关的部分，都放在后端的即时编译器部分，有利于 JRuby、Clojure、Groovy 等基于 JVM 的编程语言产生的 Class 文件同样能享受到编译器优化所带来的好处。

本文所讨论的 Dubbo 动态编译所指的是前端编译，下述对其细节展开更深入的介绍。

## 聚焦前端编译



图 10-4 Javac 的编译过程<sup>④</sup>

前端编译过程大致可以分为上图所示的 3 个过程，分别是：

1. 解析与填充符号表过程：
  - a. 词法分析
  - b. 语法分析
2. 插入式注解处理器的注解处理过程；
3. 分析与字节码生成：
  - a. 标注检查
  - b. 数据流分析
  - c. 解语法糖
  - d. 字节码生成

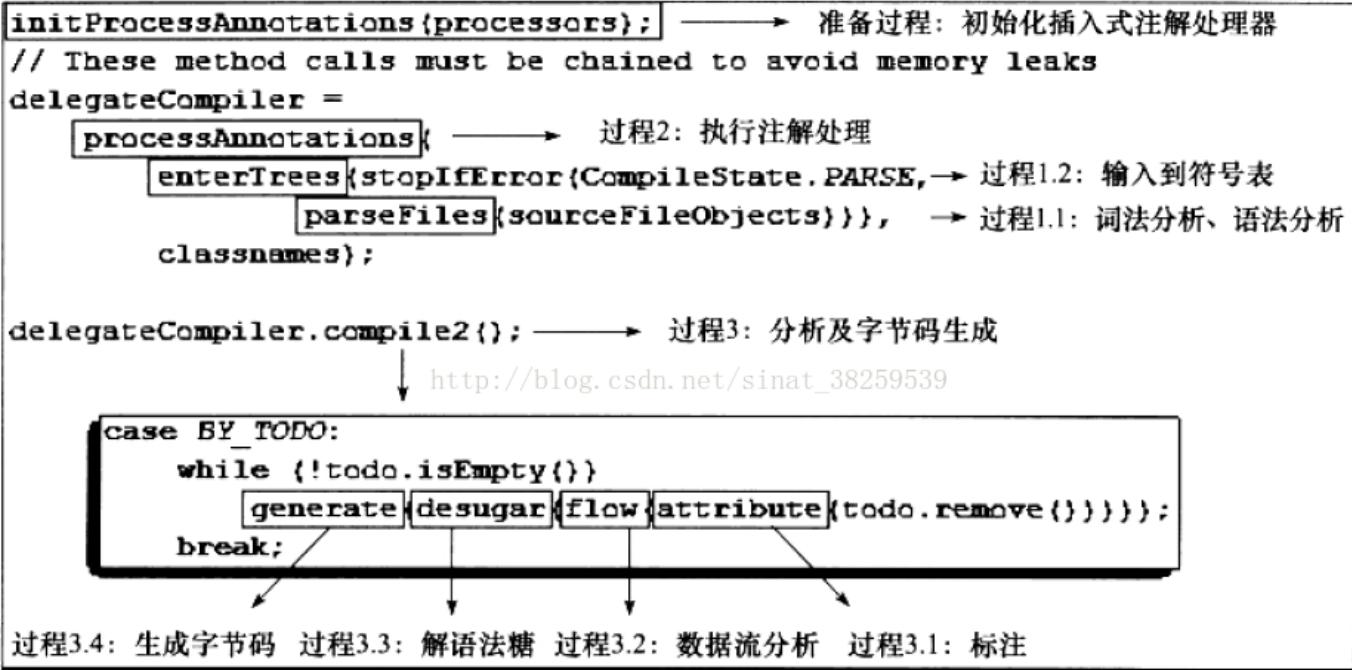


图 10-5 Javac 编译过程的主体代码

## 解析与填充符号表

### 词法、语法分析

经典的程序编译原理中包含了词法分析和语法分析两个步骤。

词法分析是将源代码的字符流转变为标记（Token）集合，关键字、变量名、字面量、运算符都可以成为标记，如“int a=b+2”这句代码包含了 6 个标记，分别是 int、a、=、b、+、2。

语法分析是根据 Token 序列构造抽象语法树的过程，抽象语法树（Abstract Syntax Tree, AST）是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构（Construct），例如包、类型、修饰符、运算符、接口、返回值甚至代码注释等都可以是一个语法结构。

### NOTE

语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的，需要由语义分析来提供保障。

## 填充符号表

符号表（Symbol Table）是由一组符号地址和符号信息构成的表格，可以认为是哈希表中 K-V 值对的形式（实际上它可以是有序符号表、树状符号表、栈结构符号表等）。语义分析阶段用于语义检查（如检查一个名字的使用和原先的说明是否一致）和产生中间代码；目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

## NOTE

符号表填充阶段，若发现用户代码中没有提供任何构造函数，那编译器将会添加一个没有参数的、访问性public、protected或private与当前类一致的默认构造函数。

## 注解处理器

我们见过的大多数注解是在代码运行期发挥作用的，Java中还存在一些能够在编译期间发生作用的注解，可以对抽象语法树中的任意元素做读取、修改、添加操作。若存在编译期注解，编译器将回到解析及填充符号表的过程重新处理，直到所有插入式注解处理器都没有再对语法树进行修改为止，每一次循环称为一个 Round，“javac的编译过程”一图中的回环体现的就是这点。

## 语义分析

语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查，如进行类型审查。下述示例中的只有第一次对 d 进行的赋值运算通过了编译，尽管所有代码行都能构成正确的语法树。

```
//定义了3个变量
int a = 1;
boolean b = false;
char c = 2;
```

JAVA

```
//后续可能出现的赋值运算
int d = a + c;
int d = b + c;
char d = a + c;
```

语义分析实际上又分成了标注检查和数据及控制流分析两个步骤，按流程图，他们和解语法糖、字节码生成一起纳入到一个章节。

## 标注检查

检查内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。

## 数据及控制流分析

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检测出诸如程序局部变量是在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。编译时期的数据及控制流分析与类加载时数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验只有在编译期或运行期才能进行。

另外在java中，局部变量可以声明为final型，但是它也仅仅是在编译器保障了变量的不变性，和没有声明final的编译出来的class文件是一模一样的，也就是该局部变量的final声明没法影响到运行期。

```
// 方法一带有 final 修饰
public void foo(final int arg) {
    final int var = 0;
    // do something
}

// 方法二没有 final 修饰
public void foo(int arg) {
    int var = 0;
    // do something
}
```

根本原因是局部变量与字段实例变量、类变量是有区别的，它在常量池中没有 CONSTANT\_Fieldref\_info 的符号引用，自然就没有访问标志（Access\_Flags）的信息，甚至可能连名称都不会保留下 来取决于编译时的选项，自然在 Class 文件中不可能知道一个局部变量是不是声明为 final 了。

## 解语法糖

语法糖是方便程序员使用而在开发语言中提供语法，java中常用语法糖，如泛型、变长参数、自动装箱 / 拆箱等，不是JVM运行时候支持的，在编译阶段需要将他们还原回简单的基础语法结构，这个过程称为解语法糖。

**NOTE** Java语法糖：断言语句、方法变长参数、数字字面量带下划线分割、自动装箱和拆箱、switch 支持枚举和字符串、try-with-resource、lambda表达式、内部类、枚举类、条件编译。

## 字节码生成

这个阶段负责：1) 前面各个步骤所生成的信息语法树、符号表转化成字节码写到磁盘中；2) 做如下少量代码添加和转换成处理。

1. 字符串的加操作替换为 StringBuffer 或 StringBuilder 取决于目标代码的版本是否大于或等于JDK1.5 的 append() 操作。
2. 添加构造器方法到语法树中，在其中按序纳入加入元素使之运行期按此顺序运行：
  - a. 实例构造器 <init>() 方法：1) 调用父类实例构造器；2) 实例变量初始化；3) 执行 {...} 语句块；
  - b. 类构造器 <cinit>() 方法：1) 类变量初始化；2) 执行 static{...} 语句块；

**NOTE**

<clinit>() 方法中无须调用父类的 <clinit>() 方法，虚拟机会自动保证父类构造器的执行，但在 <clinit>() 方法中经常会生成调用 java.lang.Object 的 <init>() 方法的代码

## Jdk版编译支持

上一章节已经系统介绍过 前端编译，知道了Dubbo所涉动态编译背后究竟在做什么工作，接下来需要熟悉下如何使用API来编译源码。

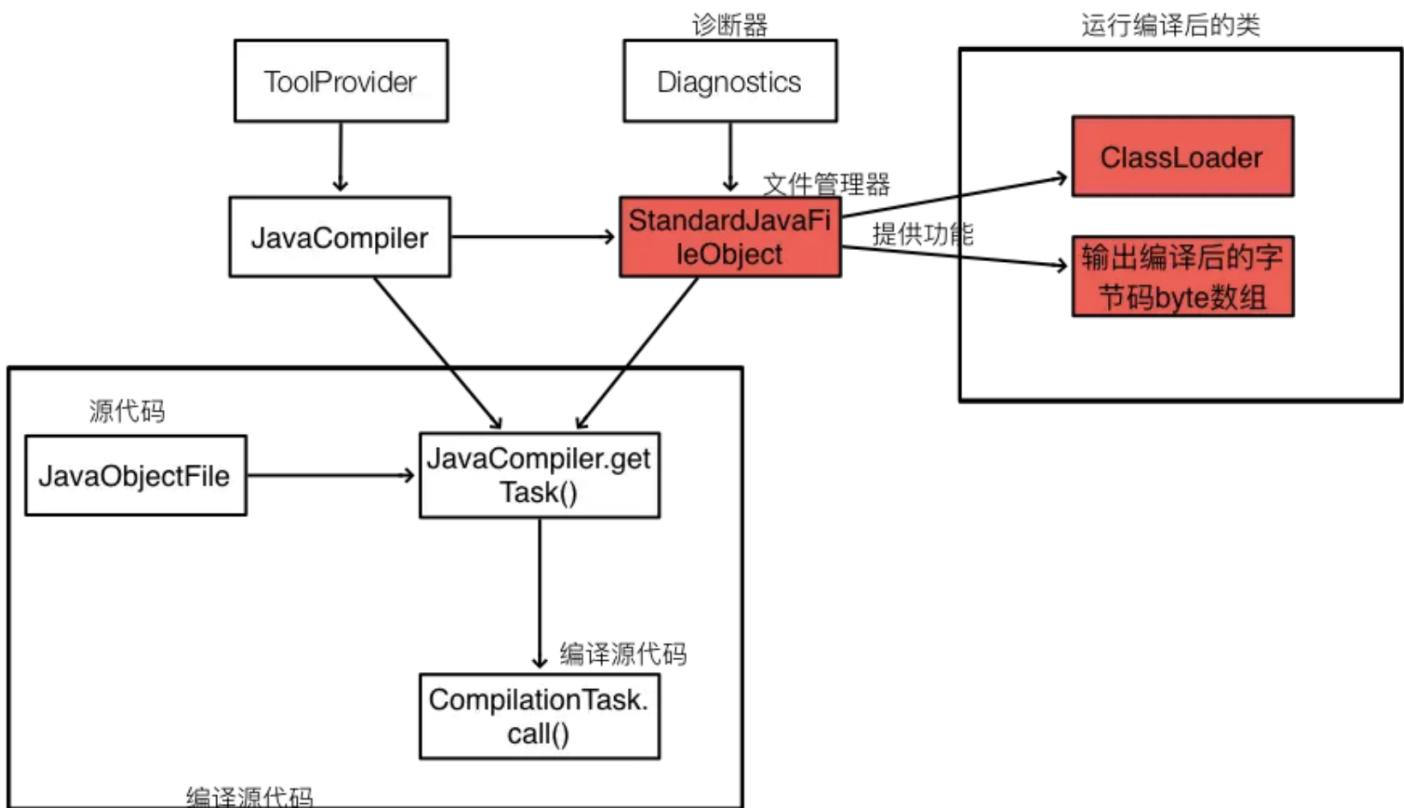
JDK中提供一个javac命令，使用它我们可以将Java源码编码生成字节码文件( eg: javac SomeSample.java )，同时Java也提供了API供开发者调用完成目标源码的编译处理。

```
public class CompileFileToFile{  
  
    public static void main(String[] args) {  
        //获取系统Java编译器  
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();  
        //获取Java文件管理器  
        StandardJavaFileManager fileManager =  
            compiler.getStandardFileManager(null, null, null);  
        //定义要编译的源文件  
        File file = new File("/path/to/file");  
        //通过源文件获取到要编译的Java类源码迭代器，包括所有内部类  
        //其中每个类都是一个 JavaFileObject，也被称为一个汇编单元  
        Iterable<? extends JavaFileObject> compilationUnits =  
            fileManager.getJavaFileObjects(file);  
        //生成编译任务  
        JavaCompiler.CompilationTask task =  
            compiler.getTask(null, fileManager, null, null, null, compilationUnits);  
        //执行编译任务  
        task.call();  
    }  
}
```

JAVA

上述源码只是呈现了最基本的使用，实际上要在程序中完成源码的编译处理，需要有如下几个必要支持元素：

1. 编译器：完成前端编译涵盖的所有细节；
2. 源码文本：待编译的源码，被处理对象；
3. 源代码、字节码文件的管理：由文件系统支持，包括文件创建和管理；
4. 编译过程中的选项：要编译的代码版本、目标，源代码位置，classpath和编码等等；
5. 编译中编译器输出的诊断信息：告知编译成功还是失败，会有什么隐患提出警告信息；



## 熟悉 JavaCompiler

本章节开始部分呈现的源码部分说明，如果Java开发人员想通过调用Java内置API的方式对源码文本进行编译处理，必先通过 `ToolProvider.getSystemJavaCompiler()` 获取到 `JavaCompiler` 编译器，它也是访问入口，其它提供支撑的代码元素都围绕它转开。使用它可以获取 `JavaCompiler.CompilationTask` 实例，然后此实例来执行对应的编译任务，其实这个执行过程是一个并发的过程。

### NOTE

JDK在 `rt.jar` 中定义了 `JavaCompiler` 接口，并在 `tools.jar` 中提供了默认实现，其实现必须符合Java语言规范，同时使用实例生成的类文件也需要符合Java虚拟机规范，比如要作用在JDK6或以上版本来说，就得能够支持注解处理。

## 基本元素

为了尽最大程度地保证 `JavaCompiler` 的灵活性和可扩展性，Java在更高的抽象层次提供了如下组成元素：

- **FileObject**接口：代表普通的文件或者数据源，例如一个 `FileObject` 对象可以是代表一个普通的文件、内存中的缓存、数据库中的数据，它定义了包括读写操作、读取信息，删除文件等一些操作；
- **JavaFileObject**接口：→ `FileObject`，代表一个Java源码文件或者Java的class字节码文件；

- **ForwardingFileObject**类: `ForwardingFileObject<F extends FileObject>` implements `FileObject` 为提高扩展性, 将行为委托给其代理的 `FileObject` 对象。
- **SimpleJavaFileObject**类: → `JavaFileObject` , 供扩展定制, 只提供了最基本的实现, 大多数方法直接抛异常处理;
- **ForwardingJavaFileObject**类: `ForwardingFileObject<F extends FileObject>` implements `FileObject` 为提高扩展性, 将行为委托给其代理的 `JavaFileObject` 对象。
- **Diagnostic<T>**接口: 代表源码文件中的特定位置的一个编译问题, `position` 是一个以0开始相对于文件起始位置的字符偏移量, `line` 和 `column` 的起始偏移位置则以1开始;
- **DiagnosticListener<T>**接口: 用于给调用方通知诊断内容, 通知内容为 `Diagnostic` , 若使用方没有提供, 则使用 `System.err` 将诊断内容打印出来;
- **DiagnosticCollector**类: → `DiagnosticListener` , 用于汇总所有诊断信息;
- **JavaFileManager**接口: 文件管理服务, 用来创建 `JavaFileObject` , 包括从特定位置输出和输入一个 `JavaFileObject` ;
- **ForwardingJavaFileManager**类: `ForwardingJavaFileManager<M extends JavaFileManager>` implements `JavaFileManager` , 为提高扩展性, 将行为委托给其代理的 `JavaFileManager` 对象。
- **StandardJavaFileManager**: 源码编译器离不开文件操作, 可以认为该接口对一些基础操作进行了抽象, 用于解决读取普通文件并从中获得 `FileObject` 对象的问题, 更多是为了方便定制:  
1) 定制 `JavaCompiler` 如何对文件进行读写处理; 2) 定制如何在多个并发编译任务间进行共享;
- **JavaFileManager.Location**和**StandardLocation**: 描述的是 `JavaFileObject` 对象的位置, 由 `JavaFileManager` 使用来决定在哪创建或者搜索文件。

## IMPORTANT

本章节中列举的组成元素中, 有几个名称形如 `ForwardingXXX` 的类, 它们正是Java提供给开发者做定制扩展处理的, 他们将行为委托给了真正实现同一接口的 JDK内置类, 由于是对被委托对象的封装代理, 因此也就意味着在符合当前接口方法签名和 `JavaCompiler` 某些约定条件的前提下, 可以对行为进行自定义处理。

## 关于使用

关于 `JavaCompiler` 的上述所有描述, 看完之后总让人有一种“知道所有的道理, 但还是过不好这一生”的惆怅, 在阐述 `JdkCompiler` 实现之前, 有必要就其使用相关的内容做更多介绍。

准备完所有必要的元素后，从 `JavaCompiler` 源码编译器获得 `JavaCompiler.CompilationTask` 实例，唤起 `call()` 方法执行编译操作，这个编译过程中，从 `JavaFileManager` 的视觉来说是：

编译器会首先通过其 `list()` 方法获取指定位置、`package`下的所有符合要求的 `JavaFileObject` 对象源码，还可以支持递归扫描子 `package` 下的源码、继承的类、实现的接口；编译完则调用其 `getJavaFileForOutput()` 方法获取用于 `class` 字节码输出的 `JavaFileObject` 对象。

然而从 `JavaFileObject` 的视觉来看是：

编译器会首先使用用户提供的 `JavaFileObject` 对象获取源码内容 `getCharContent()`，执行完编译后同样会使用该对象输出字节码 `openOutputStream()`。

下述对源码的进一步追踪也佐证了这一点，`JavaCompiler` 编译完源码后的处理步骤大体如下述源码：

```
//①源码编译完之后，JavaCompiler会调用JavaFileManager获得一个输出型的JavaFileObject对象  
JavaFileObject outFile = fileManager.getJavaFileForOutput(  
    outLocn, name, JavaFileObject.Kind.CLASS, c.sourcefile);  
...  
//②进阶这会开启输出流，将编译好的字节码对象写入  
OutputStream out = outFile.openOutputStream();  
writeClassFile(out, c);
```

相对地，在编译前获取源码的处理步骤大致如下：

```
//①获取所有输入源执行“词法、语法分析”  
//public List<JCCompilationUnit> parseFiles(Iterable<JavaFileObject>)  
parseFiles(sourceFileObjects)  
  
//②对单个输入源做“词法、语法分析”  
//protected JCCompilationUnit parse(JavaFileObject, CharSequence)  
parse(filename, readSource(filename));  
  
//③源码文本只是简单的调用getCharContent()方法取得  
public CharSequence readSource(JavaFileObject filename) {  
    try {  
        inputFiles.add(filename);  
        return filename.getCharContent(false);  
    } catch (IOException e) {  
        log.error("error.reading.file", filename, JavacFileManager.getMessage(e));  
        return null;  
    }  
}
```

综上，定制点主要集中在对 `JavaFileObject` 的 `getCharContent()` 和 `openOutputStream()` 这一对方法中。如下完整签名又说明源码输入文件对象和输出对象是有成对关系的。

```
public JavaFileObject getJavaFileForOutput  
    (Location location, String className, JavaFileObject.Kind kind, FileObject sibling)
```

进一步说，就是实现中可以根据需要决定是否将输入输出统一在同一个 `JavaFileObject` 实现类上，仔细看看该接口的如下方法定义`JavaFileObject`接口扩展自`FileObject`（Dubbo中的实现采取正是这种方式）：

```
public interface FileObject {  
    /**  
     * Gets an InputStream for this file object.  
     */  
    InputStream openInputStream() throws IOException;  
  
    /**  
     * Gets an OutputStream for this file object.  
     */  
    OutputStream openOutputStream() throws IOException;  
  
    /**  
     * Gets a reader for this object. The returned reader will  
     * replace bytes that cannot be decoded with the default  
     * translation character. In addition, the reader may report a  
     * diagnostic unless {@code ignoreEncodingErrors} is true.  
     */  
    Reader openReader(boolean ignoreEncodingErrors) throws IOException;  
  
    /**  
     * Gets the character content of this file object, if available.  
     * Any byte that cannot be decoded will be replaced by the default  
     * translation character. In addition, a diagnostic may be  
     * reported unless {@code ignoreEncodingErrors} is true.  
     */  
    CharSequence getCharContent(boolean ignoreEncodingErrors) throws IOException;  
  
    /**  
     * Gets a Writer for this file object.  
     */  
    Writer openWriter() throws IOException;  
}
```

## JdkCompiler 实现

有了上述章节的背景知识，理解 `JdkCompiler` 相对就容易了，它申明了3个内部静态类，`JavaFileObjectImpl`、`JavaFileManagerImpl`、`ClassLoaderImpl`，为方便讨论下述先分后总，逐个章节展开。

### JavaFileObjectImpl

该类扩展自 SimpleJavaFileObject，目的是为 JavaCompiler 提供输入源source和输出源bytecode，总体源码如下：

```
private static final class JavaFileObjectImpl extends SimpleJavaFileObject {  
  
    private final CharSequence source;  
    private ByteArrayOutputStream bytecode;  
  
    public JavaFileObjectImpl(final String baseName, final CharSequence source) {  
        super(ClassUtils.toURI(baseName + ClassUtils.JAVA_EXTENSION), Kind.SOURCE);  
        this.source = source;  
    }  
  
    JavaFileObjectImpl(final String name, final Kind kind) {  
        super(ClassUtils.toURI(name), kind);  
        source = null;  
    }  
  
    public JavaFileObjectImpl(URI uri, Kind kind) {  
        super(uri, kind);  
        source = null;  
    }  
  
    @Override  
    public CharSequence getCharContent(final boolean ignoreEncodingErrors) throws  
UnsupportedOperationException {  
        if (source == null) {  
            throw new UnsupportedOperationException("source == null");  
        }  
        return source;  
    }  
  
    @Override  
    public InputStream openInputStream() {  
        return new ByteArrayInputStream(getByteCode());  
    }  
  
    @Override  
    public OutputStream openOutputStream() {  
        return bytecode = new ByteArrayOutputStream();  
    }  
  
    //JavaCompiler对源码编译完后，会将二级制字节码输出  
    //到bytecode这个ByteArrayOutputStream输出流中  
    public byte[] getByteCode() {  
        return bytecode.toByteArray();  
    }  
}
```

## ClassLoaderImpl

Java近二十年的发展，虽然也支持解释执行，但总体上是编译执行为主，特别是Dubbo中使用动态编译的场景都集中在生成扩展点具类上，虽然该场景下一个扩展点只会有一个实例，但其对一个字节码执行的频度往往是比较高的，不适合使用解释执行方法。另外编译执行会有前端编译和后端编译的优化，

也能提供执行效率。

只有经过编译的源码，才能被 `ClassLoader` 加载到JVM中运行`class`字节码，这也是 `JdkCompiler` 存在的目的，显然后者编译得到的`class`字节码数据并不是通常存储在磁盘中的 `*.class` 文件或者内含在jar包中，这些字节码是缓存在内存中的，由 `JdkCompiler` 使用一个 `JavaFileObjectImpl` 类型对象输出到 `bytecode` 输出流上，是一般 `ClassLoader` 执行类加载时所搜寻的classpath目录中找不到的。

所谓类加载是指 `ClassLoader` 将`class`字节码转换成存储在方法区中的 `Class<?>` 类对象元数据这么一个过程，`ClassLoaderImpl` 担纲着将 `JdkCompiler` 编译得到的字节码加载成类对象的重任。

`ClassLoader` 的加载模式中，存在一个双亲委托的说法，意思是先尝试使用 `parent` 的 `loadClass(name)` 方法加载指定类，如果没有找到对应 `name` 的字节码内容返回null，本应是一个``Class<?>`类型的对象，当前 `ClassLoader` 便会执行自身所覆写 `findClass(clzName)` 方法，由其载入与 `clzName` 对应的字节码数据，最后使用父类的 `defineClass(clzName,bytes,0,bytes.length)` 载入最终的 `Class<?>` ``类实例。

综上，`ClassLoaderImpl` 定义了一个 `Map<String, JavaFileObject>` 类型的 `classes` 容器，并提供了 `add(String name, JavaFileObject file)` 方法，以便装入编译完源码后得到 `JavaFileObject` 类型对象，有了他们，调用其 `loadClass(name)` 方法便能完成内存字节码加载，最终获得类对象。

JAVA

```

private final class ClassLoaderImpl extends ClassLoader {
    private final Map<String, JavaFileObject> classes = new HashMap<String, JavaFileObject>();
    ClassLoaderImpl(final ClassLoader parentClassLoader) {
        super(parentClassLoader);
    }
    Collection<JavaFileObject> files() {
        return Collections.unmodifiableCollection(classes.values());
    }
    @Override
    protected Class<?> findClass(final String qualifiedClassName)
        throws ClassNotFoundException {
        JavaFileObject file = classes.get(qualifiedClassName);
        if (file != null) {
            byte[] bytes = ((JavaFileObjectImpl) file).getByteCode();
            return defineClass(qualifiedClassName, bytes, 0, bytes.length);
        }
        try {
            return org.apache.dubbo.common.utils.ClassUtils
                .forNameWithCallerClassLoader(qualifiedClassName, getClass());
        } catch (ClassNotFoundException nf) {
            return super.findClass(qualifiedClassName);
        }
    }
    void add(final String qualifiedClassName, final JavaFileObject javaFile) {
        classes.put(qualifiedClassName, javaFile);
    }
    @Override
    protected synchronized Class<?> loadClass(final String name, final boolean resolve)
        throws ClassNotFoundException {
        return super.loadClass(name, resolve);
    }
    @Override
    public InputStream getResourceAsStream(final String name) {
        if (name.endsWith(ClassUtils.CLASS_EXTENSION)) {
            String qualifiedClassName = name.substring(0, name.length() -
ClassUtils.CLASS_EXTENSION.length()).replace('/', '.');
            JavaFileObjectImpl file = (JavaFileObjectImpl)
classes.get(qualifiedClassName);
            if (file != null) {
                return new ByteArrayInputStream(file.getByteCode());
            }
        }
        return super.getResourceAsStream(name);
    }
}

```

上述源码中有几处比较特殊的地方：

1. `findClass()` 方法体中，如果没能从容器中加载到指定 `name` 名字的 `Class<?>` 对象，dubbo会使用加载当前 `JdkCompiler` 的那个 `ClassLoader` 去加载，若后续又抛错了，则忽略异常，并试图调用 `super.findClass()` 获取；
2. `loadClass()` 方法被覆写，只是方法签名中添加了 `synchronized` 修饰符，原因是其所处 `JdkCompiler` 实例环境是被并发使用的，`classes` 容器也成了争用资源；

## JavaFileManagerImpl

有了上述章节的一系列铺垫后，即便不看代码，对 `JavaFileManagerImpl` 要做的事情也已基本能掌握个十之八九。简单来讲就是在并发环境下，为 `JavaCompiler` 提供输入源和输出源均是 `JavaFileObjectImpl` 实例。

输出源的处理在剖析 `ClassLoaderImpl` 时已经阐明——将输出型的 `JavaFileObject` 实例使用 `add()` 方法添加到它的容器中去。

装入的 `fileObjects` 容器中的 `JavaFileObject` 对象所使用的Key键构建方式： "SOURCE\_PATH/" + 包名 + "/" + Java文件名称，如：  
`SOURCE_PATH/org.apache.dubbo.common.compiler.support/HelloServiceImpl8.java`。

```
private static final class JavaFileManagerImpl extends ForwardingJavaFileManager<JavaFileManager> {
    private final ClassLoaderImpl classLoader;

    @Override
    public JavaFileObject getJavaFileForOutput(Location location, String qualifiedName,
                                                Kind kind, FileObject outputFile)
        throws IOException {
        JavaFileObject file = new JavaFileObjectImpl(qualifiedName, kind);
        classLoader.add(qualifiedName, file);
        return file;
    }

    @Override
    public ClassLoader getClassLoader(JavaFileManager.Location location) {
        return classLoader;
    }
    ...
}
```

于输入源的处理，则相对复杂点，和 `ClassLoaderImpl` 一样，它同样声明了一个 `Map<String, JavaFileObject>` 类型的容器，也即属性 `fileObjects`，由外围类将源码提供给它，如下：

```
private static final class JavaFileManagerImpl extends
ForwardingJavaFileManager<JavaFileManager> {

    private final Map<URI, JavaFileObject> fileObjects = new HashMap<URI,
JavaFileObject>();
    @Override
    public FileObject getFileForInput(Location location, String packageName, String
relativeName) throws IOException {
        FileObject o = fileObjects.get(uri(location, packageName, relativeName));
        if (o != null) {
            return o;
        }
        return super.getFileForInput(location, packageName, relativeName);
    }

    public void putFileForInput(StandardLocation location, String packageName, String
relativeName, JavaFileObject file) {
        fileObjects.put(uri(location, packageName, relativeName), file);
    }
    ...
}
```

注：关于 `list()` 和 `inferBinaryName()` 这两个方法的具体作用不是很清楚，本文不做分析。

## 最终实现

上文中已经隐约表示 `JavaFileManagerImpl` 和 `ClassLoaderImpl` 的实现都是支持并发的，关于 `FileManager` 有关的 API 文档中也提到共享能够提高吞吐量，因此作为以单例模式存在的 `JdkCompiler` 会在类的实例化时便将他们作为自身的属性也实例化了。

`ClassLoaderImpl` 作为自定义 `ClassLoader` 的实现，需要有一个称之为 `parent` 的 `ClassLoader` 作为其代理目标，同时 `JavaFileManagerImpl` 也代理了一个默认的目标 `StandardJavaFileManager` 对象，因而在构建他们的实例时，先得将被代理的对象准备好，另外构造函数中也准备了编译器需要的其它元素。

```
public class JdkCompiler extends AbstractCompiler {  
  
    private final JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();  
  
    private final DiagnosticCollector<JavaFileObject> diagnosticCollector = new  
DiagnosticCollector<JavaFileObject>();  
  
    private final ClassLoaderImpl classLoader;  
  
    private final JavaFileManagerImpl javaFileManager;  
  
    private volatile List<String> options;  
  
    public JdkCompiler() {  
        //编译选项  
        options = new ArrayList<String>();  
        options.add("-source");  
        options.add("1.6");  
        options.add("-target");  
        options.add("1.6");  
  
        //被代理的JavaFileManager对象  
        StandardJavaFileManager manager =  
compiler.getStandardFileManager(diagnosticCollector, null, null);  
        final ClassLoader loader = Thread.currentThread().getContextClassLoader();  
        if (loader instanceof URLClassLoader  
            &&  
(!"sun.misc.Launcher$AppClassLoader".equals(loader.getClass().getName()))) {  
            try {  
                URLClassLoader urlClassLoader = (URLClassLoader) loader;  
                List<File> files = new ArrayList<File>();  
                for (URL url : urlClassLoader.getURLs()) {  
                    files.add(new File(url.getFile()));  
                }  
                //将加载当前JdkCompiler类的ClassLoader中的所有classpath设置给manager  
                manager.setLocation(StandardLocation.CLASS_PATH, files);  
            } catch (IOException e) {  
                throw new IllegalStateException(e.getMessage(), e);  
            }  
        }  
  
        //对目标ClassLoader再封装一次，以满足Java的安全要求  
        classLoader = AccessController.doPrivileged(new  
PrivilegedAction<ClassLoaderImpl>() {  
            @Override  
            public ClassLoaderImpl run() {  
                return new ClassLoaderImpl(loader);  
            }  
        });  
        javaFileManager = new JavaFileManagerImpl(manager, classLoader);  
    }  
    ...  
}
```

JAVA

最终在 `doCompile()` 方法实现中，先使用源码构建 `JavaFileObjectImpl` 类型的输入源对象，塞入到 `JavaFileManagerImpl` 设置的容器中，随后从中取得输入源调用 `JavaCompiler` 的 `getTask()` 得到编译任务，紧接着唤起 `call()` 完成源码的编译操作，最后使用同一个 `name` 调用 `ClassLoaderImpl` 将字节码加载成 `Class<?>` 对象。

```
public Class<?> doCompile(String name, String sourceCode) throws Throwable {  
    int i = name.lastIndexOf('.');  
    String packageName = i < 0 ? "" : name.substring(0, i);  
    String className = i < 0 ? name : name.substring(i + 1);  
    JavaFileObjectImpl javaFileObject = new JavaFileObjectImpl(className,  
sourceCode);  
    javaFileManager.putFileForInput(StandardLocation.SOURCE_PATH, packageName,  
        className + ClassUtils.JAVA_EXTENSION, javaFileObject);  
    Boolean result = compiler.getTask(null, javaFileManager, diagnosticCollector,  
options,  
        null, Arrays.asList(javaFileObject)).call();  
    if (result == null || !result) {  
        throw new IllegalStateException("Compilation failed. class: " + name + ",  
diagnostics: " + diagnosticCollector);  
    }  
    return classLoader.loadClass(name);  
}
```

---

完结

# Dubbo之SPI扩展点加载

Dubbo中的一个重要理念是“微内核、可扩展”，其高可扩展性离不开SPI<sub>扩展点发现机制</sub>的支持，它是对Java原生SPI的增强实现，同时规避了后者的一些不足：

1. 按需加载扩展点具类，避免JDK标准SPI的一次性加载带来的初始化耗时以及扩展点具类没被用上的资源浪费；
2. **AOP** 支持，抽调公共逻辑，对同一扩展点具类做装饰处理，依需要在其方法前后增加逻辑代码；
3. **IoC** 支持，一个扩展点可以直接 setter 注入其它扩展点；

## NOTE

为方便阐述，下述将所有 扩展点实现 统称为 扩展点具类，也即扩展点的某种具体实现类。

## 基础

### 基本使用

约定：

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

## NOTE

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

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/org.apache.dubbo.rpc.Protocol，内容为：

xxx=com.alibaba.xxx.XxxProtocol

实现类内容：

```
package com.alibaba.xxx;

import org.apache.dubbo.rpc.Protocol;

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

JAVA

## 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" />
```

### NOTE

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。

## 高级特性

Dubbo中的SPI机制是通过 `ExtensionLoader` 实现的，它的一些高阶特性需要通过注解`@Activate`和`@Adaptive`搭配完成。

### 扩展点自动包装 AOP支持

在《Dubbo与设计模式》一文中，已经对装饰者模式进行过深入的探讨。Dubbo中所谓的扩展点AOP支持，其实现其实是装饰者对被装饰者在行为委派时就其前后增加对应的特性业务代码，将对应扩展点的公共逻辑抽离到装饰者`Wrapper`类。如对某扩展点提供了`Wrapper`装饰者类，那么Dubbo会就此扩展点其它非`Wrapper`类实现做包装处理，返回是该`Wrapper`类的实例。

以上文出现的RPC框架层中的子层协议层扩展点 `Protocol` 为例，Dubbo使用该机制实现了RPC中的拦截链特性，具体请参看《 Dubbo RPC 之 Protocol协议层（二） 》一文。

### 扩展点自动装配 IoC支持

`ExtensionLoader` 在加载扩展点时，会对 `setter` 方法进行判定，如果其入参是另外一个扩展点的话，如果扩展点有多个实现的话，需要采用 扩展点自适应机制 来确定使用哪个。

### 扩展点自适应 @Adaptive

当一个扩展点有多个实现时，依赖它的扩展点需要有某种机制确定到底使用最终使用哪个。Dubbo中被俗称为配置总线的URL，主要用于将配置作为参数在方法调用帧间传递，其背后实际上一个Key-Value键值对形式的Map容器，`ExtensionLoader` 利用 URL + `@Adaptive` 这对组合来确定选用哪个实现。

ExtensionLoader 会综合 URL + @Adaptive + 被依赖扩展点接口类获得 Adaptive 实例，从而使得在方法执行时才决定被选用的其它扩展点具类成为可能，而当前对应 Adaptive 实现是在加载扩展点里动态生成。以 Dubbo 的 `Transporter` 扩展点为例：

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

JAVA

实例中，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。

### 扩展点自激活 @Activate

“ 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 org.apache.dubbo.common.extension.Activate;  
import org.apache.dubbo.rpc.Filter;  
  
@Activate // 无条件自动激活  
public class XxxFilter implements Filter {  
    // ...  
}
```

JAVA

或：

```
import org.apache.dubbo.common.extension.Activate;  
import org.apache.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 {  
    // ...  
}
```

JAVA

或：

```
import org.apache.dubbo.common.extension.Activate;
import org.apache.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 {
    ...
}
```

JAVA

## 实现

在具体实现上，概括来讲，Dubbo将整个过程大体分为如下3步：

1. 读取 classpath 下的SPI配置文件；
2. 根据配置解析得到扩展点具类及其依赖扩展点具类的Class对象集；
3. 根据当前扩展点所持有的Class对象集按需实例化某个具类，包括注入其它扩展点具类实例；

但为了方便理解，下面剖析具体实现的章节将按照由表入里、依赖前置的原则逐层展开。

## 获取扩展点加载器

```
ExtensionLoader.getExtensionLoader(SomeSPI.class).getXXXExtension(... args)
```

JAVA

每当需要获取一个扩展点实例时，总会调用一段类似如上片段的代码，其中 `ExtensionLoader` 是 SPI 实现的核心类，是按需延时加载扩展点具类的加载器，每一个被 `@SPI` 标注的扩展点会对应它的一个实例，Dubbo 使用一个 `ConcurrentMap<Class<?>, ExtensionLoader<?>>` 类型的 Map 容器将这种关系缓存起来，避免重复加载。

`ExtensionLoader` 构造函数是私有的，它的实例进行使用工厂方法获得，在实例化时，会对其中两个最为关键的 `final` 型属性赋值，`type` 是表征扩展点的接口类型，而 `objectFactory` 是用于最终获取扩展点实例的工厂。

```

public class ExtensionLoader<T> {
    ...
    private static final ConcurrentMap<Class<?>, ExtensionLoader<?>>
        EXTENSION_LOADERS = new ConcurrentHashMap<>();
    ...
    private final ExtensionFactory objectFactory;
    private final Class<?> type;
    private ExtensionLoader(Class<?> type) {
        this.type = type;
        objectFactory = (type == ExtensionFactory.class ? null :
            ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension());
    }
    ...
    private static <T> boolean withExtensionAnnotation(Class<T> type) {
        return type.isAnnotationPresent(SPI.class);
    }
    @SuppressWarnings("unchecked")
    public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
        if (type == null) {
            throw new IllegalArgumentException("Extension type == null");
        }
        if (!type.isInterface()) {
            throw new IllegalArgumentException("Extension type (" + type + ") is not an interface!");
        }
        if (!withExtensionAnnotation(type)) {
            throw new IllegalArgumentException("Extension type (" + type +
                ") is not an extension, because it is NOT annotated with @" +
                SPI.class.getSimpleName() + "!");
        }
        ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        if (loader == null) {
            EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
            loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        }
        return loader;
    }
    ...
}

```

## 获取扩展点实例

获得扩展点加载器后，便可使用该加载器将扩展点具类类型的实例化，`getXXXExtension(... args)`已经表示可用的实例化方法依据特性要求存在8种形式，为方便进一步讨论，下述细分为两章节加以剖析，将扩展点方法按特性分成两组，前4组用于实例化那种指定一个Key键只能对应一个实现的扩展点具类，后4组则完全用于实例化带有自激活特性的扩展点具类。

### 实例化具名扩展点具类

除了自激活型扩展点具类，其它类型的扩展点均可以认为是带键Key的： a) SPI配置文件的键值对； b) @SPI 标注中指定的名称； c) @Adaptive 标注在扩展点具类上。

1. `T getAdaptiveExtension()`：若某注解点有一个实现标注了 `@Adaptive`，利用该方法可以直接获取其实例，于一个注解点，Dubbo只允许一个有该标注的实现；
2. `T getDefaultExtension()`：若 `@SPI` 注解带有值，那么Dubbo使用该值可以获取一个默认的扩展点具类；
3. `T getExtension(String name)`：由SPI配置文件中出现的键值对中的键来实例化对应值表示的扩展点具类，实际上 `getDefaultExtension()` 最终也是通过委托它实现；
4. `T getLoadedExtension(String name)`：和 `getExtension(String name)` 的不同之处在于，它只试图去获取已经完成实例化的扩展点具类，如果不存在既有实例，便直接返回 `null` 值；

于具名扩展点实现类来说，需要有个存储其实例的缓存，这个缓存是一个指向 `Holder` 对象的引用，或者是含有它的Map容器，其实例是采用惰性机制进行实例化，其只能实例化一次，一个实现只存在单一的实例。Dubbo中无处不在的并发，`Key` 键所对应的扩展点具类对象自然成了被争用的资源，需要加锁处理。然而在加锁时该对象可能还不存在，因而引入了一个持有它的单元素容器类 `Holder`，加锁前均能获取或者新生成一个对应的 `Holder` 实例，获得锁后再判别对应扩展点具类是否存在实例，不存在则调用 `createAdaptiveExtension()` (c)或者 `createExtension()` (a)+(b)创建对象。

```

public class ExtensionLoader<T> {

    ...
    private volatile Throwable createAdaptiveInstanceError;

    private final Holder<Object> cachedAdaptiveInstance = new Holder<>();

    private final ConcurrentHashMap<String, Holder<Object>> cachedInstances
        = new ConcurrentHashMap<>();

    public T getAdaptiveExtension() {
        Object instance = cachedAdaptiveInstance.get();
        if (instance == null) {
            if (createAdaptiveInstanceError != null) {
                throw new IllegalStateException("Failed to create adaptive instance: "
+
                    createAdaptiveInstanceError.toString(),
                    createAdaptiveInstanceError);
            }
        }

        synchronized (cachedAdaptiveInstance) {
            instance = cachedAdaptiveInstance.get();
            if (instance == null) {
                try {
                    instance = createAdaptiveExtension();
                    cachedAdaptiveInstance.set(instance);
                } catch (Throwable t) {
                    createAdaptiveInstanceError = t;
                    throw new IllegalStateException("Failed to create adaptive
instance: " + t.toString(), t);
                }
            }
        }
    }

    return (T) instance;
}

public T getExtension(String name) {
    if (StringUtils.isEmpty(name)) {
        throw new IllegalArgumentException("Extension name == null");
    }

    //直接使用true表示获取默认扩展点实例
    if ("true".equals(name)) {
        return getDefaultExtension();
    }
    final Holder<Object> holder = getOrCreateHolder(name);
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                instance = createExtension(name);
                holder.set(instance);
            }
        }
    }
    return (T) instance;
}

```

JAVA

```

        }
    }
}

return (T) instance;
}

/**
* Return default extension, return <code>null</code> if it's not configured.
*/
public T getDefaultExtension() {
    getExtensionClasses();
    if (StringUtils.isBlank(cachedDefaultName) || "true".equals(cachedDefaultName))
{
    return null;
}
//获得cachedDefaultName后，反过来调用getExtension()
return getExtension(cachedDefaultName);
}

public T getLoadedExtension(String name) {
    if (StringUtils.isEmpty(name)) {
        throw new IllegalArgumentException("Extension name == null");
    }
    Holder<Object> holder = getOrCreateHolder(name);
    return (T) holder.get();
}

//cachedInstances本身已经是线程安全的，顾无需重复加锁
private Holder<Object> getOrCreateHolder(String name) {
    Holder<Object> holder = cachedInstances.get(name);
    if (holder == null) {
        cachedInstances.putIfAbsent(name, new Holder<>());
        holder = cachedInstances.get(name);
    }
    return holder;
}
...
}
}

```

上述代码中出现了一个声明了 `volatile` 可见性保证的 `Throwable` 类型字段 `createAdaptiveInstanceError`，目的很明显——当多个线程同一时间针对某一特定扩展点调用 `getAdaptiveExtension()` 时，获得锁的线程若遇到异常，可以依靠 `volatile` 第一时间告诉其他参与争用的线程，避免重复执行必然发生错误的代码段。

## 实例化自激活扩展点具类

另外还存在如下其它4种形如 `getActivateExtension(URL url, ... args)` 的方法，用于实例化当前扩展点所有具有自激活特性的实现，上文中已提及于一个扩展点，标注了 `@Activate` 自激活的扩展点具类是可以存在多个，并且它存在3种形式：a) 无条件自激活；b) 设置 `group="provider" | "consumer"` 限定作用方；c) 配置总线中存在Key键`@Activate`注解中的配置值所对应的配置才激活。

### 1. `List<T> getActivateExtension(URL url, String key)`

2. List<T> getActivateExtension(URL url, String key, String group)
3. List<T> getActivateExtension(URL url, String[] values)
4. List<T> getActivateExtension(URL url, String[] values, String group)

如果 @Activate 注解中没有配置 group , 那么当前自激活扩展点具类可以作用于 provider 和 consumer 双方, 否则只能作用在出现在配置中的一方:

```
private boolean isMatchGroup(String group, String[] groups) {  
    if (StringUtils.isEmpty(group)) {  
        return true;  
    }  
    if (groups != null && groups.length > 0) {  
        for (String g : groups) {  
            if (group.equals(g)) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

JAVA

满足Key键的自激活扩展点具类, 当前配置总线中需要存在非对应Key键的非空 false|0|null|N/A 配置值, 其中总线URL中的Key键可能是加了前缀为RPC方法名称"somemethod"+".":

```

public class ConfigUtils {
    ...
    public static boolean isEmpty(String value) {
        return !isEmpty(value);
    }

    public static boolean isEmpty(String value) {
        return StringUtils.isEmpty(value)
            || "false".equalsIgnoreCase(value)
            || "0".equalsIgnoreCase(value)
            || "null".equalsIgnoreCase(value)
            || "N/A".equalsIgnoreCase(value);
    }
    ...
}

private boolean isActive(String[] keys, URL url) {
    if (keys.length == 0) {
        return true;
    }
    for (String key : keys) {
        for (Map.Entry<String, String> entry : url.getParameters().entrySet()) {
            String k = entry.getKey();
            String v = entry.getValue();
            if ((k.equals(key) || k.endsWith("." + key))
                && ConfigUtils.isNotEmpty(v)) {
                return true;
            }
        }
    }
    return false;
}

```

于扩展点中接口前声明的 @Activate，其配置中所表示的扩展点具类集合是Dubbo默认支持的，一般会采用首个，当没有找到对应名字的扩展点具类时会采用第二个，以此类推，开发者也可以通过在配置总线设置 `"- default" | "Name4SpiImpl"` 前缀将其指定的扩展点实现排除，若为`-default`，`@Activate` 中的所有配置指定依赖了哪些扩展点具类的Key键均会被忽略。在调用 `getActivateExtension()` 方法时，除非排除，Dubbo会自动加入所有自适配扩展点具类实例，记为ListO，他们会按照配置的 `order` 值的从小到大排序。假如 `values` 中配有 `"default"`，将自定义加载的其它扩展点具类对象一分为二变成前后ListA和ListB两个集合，那么最终得到的所有扩展点具类实例集合 `ListR=[ListA + ListO + ListB]`。在没有配 `"default"` 时，可以认为ListA是空值，此时有 `ListR=[ListO + ListB]`。另外如果 `values` 参数中包含了应该在ListO中出现的元素对应扩展点具类名称，那么对应实例将会“移入”到ListA或ListB中，这个特性让Dubbo可以在调用时决定所有依赖扩展点具类实例的最终执行顺序。

```
public List<T> getActivateExtension(URL url, String[] values, String group) { JAVA
    List<T> exts = new ArrayList<>();
    List<String> names = values == null ? new ArrayList<>(0) : Arrays.asList(values);
    if (!names.contains(REMOVE_VALUE_PREFIX + DEFAULT_KEY)) {
        getExtensionClasses();
        for (Map.Entry<String, Object> entry : cachedActivates.entrySet()) {
            String name = entry.getKey();
            Object activate = entry.getValue();

            String[] activateGroup, activateValue;

            if (activate instanceof Activate) {
                activateGroup = ((Activate) activate).group();
                activateValue = ((Activate) activate).value();
            } else if (activate instanceof com.alibaba.dubbo.common.extension.Activate)
{
                activateGroup = ((com.alibaba.dubbo.common.extension.Activate)
activate).group();
                activateValue = ((com.alibaba.dubbo.common.extension.Activate)
activate).value();
            } else {
                continue;
            }
            if (isMatchGroup(group, activateGroup)
                //参数中已经出现的扩展点实现实例移入到ListA或ListB中
                && !names.contains(name)
                //参数中加了“-”前缀的自适配实现被排除掉
                && !names.contains(REMOVE_VALUE_PREFIX + name)
                && isActive(activateValue, url)) {
                exts.add(getExtension(name));
            }
        }
        exts.sort(ActivateComparator.COMPARATOR);
    }
    List<T> usrs = new ArrayList<>();
    for (int i = 0; i < names.size(); i++) {
        String name = names.get(i);
        if (!name.startsWith(REMOVE_VALUE_PREFIX)
            && !names.contains(REMOVE_VALUE_PREFIX + name)) {
            //default之前的实例呈现在ListA位置，自适配的为List0
            if (DEFAULT_KEY.equals(name)) {
                if (!usrs.isEmpty()) {
                    exts.addAll(0, usrs);
                    usrs.clear();
                }
            } else {
                usrs.add(getExtension(name));
            }
        }
    }
    //加入最后的ListB部分
    if (!usrs.isEmpty()) {
        exts.addAll(usrs);
    }
    //得到[ListA + List0 + ListB]或[List0 + ListB]
```

```
    return exts;
}
```

其它3个变种方法均是对上述这个方法的调用，如下：

```
public List<T> getActivateExtension(URL url, String key) {
    return getActivateExtension(url, key, null);
}
public List<T> getActivateExtension(URL url, String[] values) {
    return getActivateExtension(url, values, null);
}
public List<T> getActivateExtension(URL url, String key, String group) {
    String value = url.getParameter(key);
    return getActivateExtension(url, StringUtils.isEmpty(value)
        ? null : COMMA_SPLIT_PATTERN.split(value), group);
}
```

JAVA

## 获取扩展点具类元数据

上述关于 ExtensionLoader 的整个源码剖析阐述中，更多关于SPI功能性的，比较浅层次。获取到实例前的最关键一环是得到扩展点的具类信息，也即需要得到产生实例的元数据，上述多次出现的 getExtensionClasses() 正是确保元数据先加载的一个步骤，该方法利用锁和锁的双检形式保证了只会被加载一次，如下源码所示：

```
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get();
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

JAVA

## SPI文件解析

Dubbo的SPI机制中，扩展点实际上是一个接口，其具体实现是由应用配置在 META-INF/dubbo 资源目录中的一个和接口同名的文件中，在编译期间是不知道该具体实现的，只有等到需要用时，才会综合“配置总线URL、@SPI、@Adaptive、@Activate”等信息去决定加载对应的具类，然后由后者获得对应的实例，因此解析RPC配置文件必须前置且很非常关键的一步。

## 读取SPI配置文件集

第一步需要获取到当前方法调用帧中正在使用的 ClassLoader，再结合默认给定的SPI配置 classpath 目录和当前扩展点接口类名获取到所有SPI配置文件。一个Java工程中，通常会依赖其它的jar包，和当前工程一样，他们都各自有自己的 classpath 目录，里面除了包含被编译过的class文件外，一般也会含有一些放置在 ./META-INF 目录中的配置文件。当调用 ClassLoader 的 getResources(fileName) 方法或 ClassLoader.getSystemResources(fileName) 时，这些依赖jar包中的配置文件也会一同被获取到，因而需要迭代获得多个SPI配置文件，逐个执行文件解析。

```
private void loadDirectory(Map<String, Class<?>> extensionClasses, String dir, String type) {  
    String fileName = dir + type;  
    try {  
        Enumeration<java.net.URL> urls;  
        ClassLoader classLoader = findClassLoader();  
        if (classLoader != null) {  
            urls = classLoader.getResources(fileName);  
        } else {  
            urls = ClassLoader.getSystemResources(fileName);  
        }  
        if (urls != null) {  
            while (urls.hasMoreElements()) {  
                java.net.URL resourceURL = urls.nextElement();  
                loadResource(extensionClasses, classLoader, resourceURL);  
            }  
        }  
    } catch (Throwable t) {  
        logger.error("Exception occurred when loading extension class (interface: " +  
                    type + ", description file: " + fileName + ".)", t);  
    }  
}
```

上述代码中的 findClassLoader()，实际上是调用 ClassUtils.getClassLoader(ExtensionLoader.class) 获得当前 ClassLoader 实例，它总是按照如是顺序其尝试获取该实例：1) Thread.currentThread().getContextClassLoader(); 2) ExtensionLoader.class.getClassLoader(); 3) ClassLoader.getSystemClassLoader()。在尝试了3种方式还是获取不到一个 ClassLoader 实例时，Dubbo便会直接使用 ClassLoader.getSystemResources(fileName)。

“ META-INF/services/、META-INF/dubbo/、META-INF/dubbo/internal/ 三个值，都是dubbo寻找扩展实现类的配置文件存放路径，也就是我在上述（一）注解@SPI中讲到的以接口全限定名命名的配置文件存放的路径。区别在于 META-INF/services/ 是dubbo为了兼容jdk的SPI扩展机制思想而设存在的， META-INF/dubbo/internal/ 是dubbo 内部提供的扩展的配置文件路径，而 META-INF/dubbo/ 是为了给用户自定义的扩展实现配置文件存放。

也就是说Dubbo中的SPI配置文件所在classpath的位置是有要求，不能随便放置，具体实现表达如下：

```
private static final String SERVICES_DIRECTORY = "META-INF/services/";  
private static final String DUBBO_DIRECTORY = "META-INF/dubbo/";  
private static final String DUBBO_INTERNAL_DIRECTORY = DUBBO_DIRECTORY + "internal/";  
  
/**  
 * synchronized in getExtensionClasses  
 */  
private Map<String, Class<?>> loadExtensionClasses() {  
  
    //默认扩展点加载，下文将加以阐述  
    cacheDefaultExtensionName();  
  
    Map<String, Class<?>> extensionClasses = new HashMap<>();  
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName());  
    loadDirectory(extensionClasses, DUBBO_DIRECTORY, type.getName());  
    loadDirectory(extensionClasses, SERVICES_DIRECTORY, type.getName());  
  
    return extensionClasses;  
}
```

## 逐行解析SPI配置文件

接下来便是由SPI配置文件逐行解析出配置信息，如下源码，总体步骤如下：

1. 由参数 resourceURL 获取到文件字节流；
2. 使用装饰器 InputStreamReader 基于字节流得到字符流；
3. 套上另外一个装饰器 BufferedReader 获取到带有缓冲功能的字符流；
4. 逐行读取字符流，将当前行的注释信息忽略，随后取得等号 "=" 两边的名称和扩展点类名；
5. 针对当前行执行类的元数据加载操作；

```

private void loadResource(Map<String, Class<?>> extensionClasses, ClassLoader
classLoader, java.net.URL resourceURL) {
    try {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(resourceURL.openStream(), StandardCharsets.UTF_8))) {
            String line;
            while ((line = reader.readLine()) != null) {
                final int ci = line.indexOf('#');
                if (ci >= 0) {
                    line = line.substring(0, ci);
                }
                line = line.trim();
                if (line.length() > 0) {
                    try {
                        String name = null;
                        int i = line.indexOf('=');
                        if (i > 0) {
                            name = line.substring(0, i).trim();
                            line = line.substring(i + 1).trim();
                        }
                        if (line.length() > 0) {
                            loadClass(extensionClasses, resourceURL,
Class.forName(line, true, classLoader), name);
                        }
                    } catch (Throwable t) {
                        IllegalStateException e = new IllegalStateException("Failed to
load extension class (interface: " + type + ", class line: " + line + ") in " +
resourceURL + ", cause: " + t.getMessage(), t);
                        exceptions.put(line, e);
                    }
                }
            }
        }
    } catch (Throwable t) {
        logger.error("Exception occurred when loading extension class (interface: " +
type + ", class file: " + resourceURL + ") in " + resourceURL, t);
    }
}

```

注：上述代码中使用了Java7中的 `try...{}catch...{}`，会自动完成I/O中资源的回收处理。

## IMPORTANT

Dubbo利用 `Class.forName()` 方法根据扩展点具类全名获取到对应的Class对象具类元数据，它会自动完成类的定位、加载、链接。由于指定 `initialize=true`，同一具类若没有初始化过，Java会执行Class对象的初始化处理。

```

public static Class<?> forName(String name, boolean initialize, ClassLoader loader)

```

## 扩展点具类元数据加载

`loadClass()` 是为扩展点准备好其所有实现具类的元数据，也即表示具类的 `Class<?>` 对象，有了它们才能进一步获取到具类的实例。

类加载，也即 `loadClass()` 所表示的这个过程，由于特性要求涉及到不少技术点，整个过程比较复杂，按惯例，还是化繁为简，先零后整，逐个击破。

### IMPORTANT

`ExtensionLoader` 会为每一个扩展点准备它的一个实例，`loadClass()` 这一环已经是在加载扩展点具类信息了，但所有的具类信息都是汇总在同一个 `ExtensionLoader` 实例下加以管理的。

扩展点的实例惰性加载的第一步是获取具类元数据，第二步才是根据这些准备好的元数据按照当前配置总线要求完成某个具类的实例化操作。上述已经提到，第一步对于当前 `type` 扩展点只会执行一次，第二步则是每一个具类会执行一次实例的初始化操作，后续需要他们的时候均能直接从缓存提取到。

具类的加载过程采用的是分治思想，按照当前具类的类注解`@Adaptive`、`@Activate`、`@Extension`、构造函数是否入参为当前扩展点接口类拆解成4种具类的加载过程：1) 自适配具类；2) 装饰型具类；3) 自激活型具类；4) 一般具类。

在SPI配置文件中，可以根据需要就对应扩展点具类以 "," 作为分隔符赋予多个名字。JDK中标准的 SPI 配置是没有 Key 键的，为了兼容，`ExtensionLoader` 使用 `findAnnotationName()` 方法创建 Key，规则是若具类配置了 `@Extension` 注解，取其名称，否则取具类本身的名称或者去除扩展点接口名后缀后得到的小写字符串。

整个具类元数据的总体加载过程如下：

```
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL  
resourceURL, Class<?> clazz, String name) throws NoSuchMethodException {  
    //具类必须是扩展点接口的实现类  
    if (!type.isAssignableFrom(clazz)) {  
        throw new IllegalStateException("Error occurred when loading extension class  
interface: " +  
            type + ", class line: " + clazz.getName() + " , class "  
            + clazz.getName() + " is not subtype of interface.");  
    }  
  
    if (clazz.isAnnotationPresent(Adaptive.class)) {  
        //自适配具类元数据加载  
        cacheAdaptiveClass(clazz);  
    } else if (isWrapperClass(clazz)) {  
        //装饰型具类元数据加载  
        cacheWrapperClass(clazz);  
    } else {  
        clazz.getConstructor(); //确保有一个无参构造函数，没有则报错  
  
        if (StringUtils.isEmpty(name)) {  
            name = findAnnotationName(clazz);  
            if (name.length() == 0) {  
                throw new IllegalStateException("No such extension name for the class "  
+ clazz.getName() + " in the config " + resourceURL);  
            }  
        }  
  
        String[] names = NAME_SEPARATOR.split(name);  
        if (ArrayUtils.isNotEmpty(names)) {  
            //自激活型具类元数据加载  
            cacheActivateClass(clazz, names[0]);  
  
            for (String n : names) {  
                //一般具类元数据加载  
                cacheName(clazz, n);  
                saveInExtensionClass(extensionClasses, clazz, n);  
            }  
        }  
    }  
}  
  
private String findAnnotationName(Class<?> clazz) {  
    org.apache.dubbo.common.Extension extension =  
    clazz.getAnnotation(org.apache.dubbo.common.Extension.class);  
    if (extension != null) {  
        return extension.value();  
    }  
  
    String name = clazz.getSimpleName();  
    if (name.endsWith(type.getSimpleName())) {  
        name = name.substring(0, name.length() - type.getSimpleName().length());  
    }  
    return name.toLowerCase();  
}
```

JAVA

每一种细分具类都有对应的缓存装载其类型元数据，也相应需要配合一些验证逻辑，下述分子章节进一步阐述：

## 自适配具类

保证只会有一个标注了 @Adaptive 注解的具类，注意它使用一个声明了 volatile 可见性的属性进行存取。

```
private volatile Class<?> cachedAdaptiveClass = null; JAVA

/**
 * cache Adaptive class which is annotated with <code>Adaptive</code>
 */
private void cacheAdaptiveClass(Class<?> clazz) {
    if (cachedAdaptiveClass == null) {
        cachedAdaptiveClass = clazz;
    } else if (!cachedAdaptiveClass.equals(clazz)) {
        throw new IllegalStateException("More than 1 adaptive class found: "
            + cachedAdaptiveClass.getName()
            + ", " + clazz.getName());
    }
}
```

## 装饰型具类

可以层层装饰，因此使用 Set 作为集合容器。代码中以当前扩展点的接口类信息作为入参调用 clazz.getConstructor(type) 便轻松判断当前具类是否为装饰类。

```
private Set<Class<?>> cachedWrapperClasses;
```

JAVA

```
/**  
 * cache wrapper class  
 * <p>  
 * like: ProtocolFilterWrapper, ProtocolListenerWrapper  
 */  
private void cacheWrapperClass(Class<?> clazz) {  
    if (cachedWrapperClasses == null) {  
        cachedWrapperClasses = new ConcurrentHashSet<>();  
    }  
    cachedWrapperClasses.add(clazz);  
}  
  
/**  
 * test if clazz is a wrapper class  
 * <p>  
 * which has Constructor with given class type as its only argument  
 */  
private boolean isWrapperClass(Class<?> clazz) {  
    try {  
        clazz.getConstructor(type);  
        return true;  
    } catch (NoSuchMethodException e) {  
        return false;  
    }  
}
```

## 自激活型具类

可以有多个，比如 Filter，和一般的具类缓存类的元数据 Class<?> 对象不同，它只缓存“名字”和“@Activate对象”的键值信息。存取容器Map的值应该 Activate 类型，但因该注解有两个，为兼容，委曲求全，被声明成了 Object 类型。

```

private final Map<String, Object> cachedActivates = new ConcurrentHashMap<>();

/**
 * cache Activate class which is annotated with <code>Activate</code>
 * <p>
 * for compatibility, also cache class with old alibaba Activate annotation
 */
private void cacheActivateClass(Class<?> clazz, String name) {
    Activate activate = clazz.getAnnotation(Activate.class);
    if (activate != null) {
        cachedActivates.put(name, activate);
    } else {
        // support com.alibaba.dubbo.common.extension.Activate
        com.alibaba.dubbo.common.extension.Activate oldActivate =
        clazz.getAnnotation(com.alibaba.dubbo.common.extension.Activate.class);
        if (oldActivate != null) {
            cachedActivates.put(name, oldActivate);
        }
    }
}

```

JAVA

## 一般具类

可以有多个，缓存了“名称”和“类的元数据 Class<?> 对象”的键值信息。如下 ExtensionLoader 还反向缓存了他们间的关系，由于一般具类可以有多个“名称”，因此会存在多个“名称”指向同一个 Class<?> 对象”的情况。

```

private final Holder<Map<String, Class<?>>> cachedClasses = new Holder<>();

private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHashMap<>();

/**
 * cache name
 */
private void cacheName(Class<?> clazz, String name) {
    if (!cachedNames.containsKey(clazz)) {
        cachedNames.put(clazz, name);
    }
}

/**
 * put clazz in extensionClasses
 */
private void saveInExtensionClass(Map<String, Class<?>> extensionClasses, Class<?>
clazz, String name) {
    Class<?> c = extensionClasses.get(name);
    if (c == null) {
        extensionClasses.put(name, clazz);
    } else if (c != clazz) {
        throw new IllegalStateException("Duplicate extension " + type.getName() + " " +
name + " " + name + " on " + c.getName() + " and " + clazz.getName());
    }
}

```

JAVA

## 实例化扩展点具类

有了各扩展点具类的元数据信息 `Class<?>` 后，创建其实例就比较简单了。对于所有类型的具类来说，获得其实例实际上需要三步：

1. 检查是否存在一个对应的实例，有直接返回，没有则继续第二步；
2. 使用 `Class<?>` 生成一个对象；
3. 注入处理：业务特性需求，有些具类需要依赖其它的扩展点接口，从而添加了相应的 `setter`，需要给第二步生成的对象注入被依赖扩展点具类实例；

## 扩展点具类实例化处理

注：自适配具类存在两种形式，一种是开发人员使用 `@Adaptive` 注解的扩展点实现，另一类是由 Dubbo 根据扩展点中其方法中的 `@Adaptive` 注解来生成的扩展点实现。整个实例化过程有点不一样，另起章节讨论。

开发者为 Dubbo 提供了扩展点具类后，需要加入到相应 SPI 扩展点配置文件中，否则就成了孤魂野鬼不会发生作用，要用它时也找不到。键值对的硬性要求这一点，确保了尽管有多种类型的具类，但他们的实例获取方式是一致的，使用的基本都是 `createExtension()`。

### NOTE

Dubbo 要求同一个扩展点的不同具类需要配置不同的名称，否则会强行抛错。

Dubbo 的 SPI 规定一个具类只能存在一个实例，也就是说扩展点使用的是单例模式，因此声明了一个全局 `ConcurrentMap<Class<?>, Object>` 类型的容器缓存类到对象间的关系，确保唯一性。

仔细阅读过上文的，不难看出扩展点的具类的构造函数只能存在两种形式：1) 无参；2) 类型为当前扩展点接口的单一入参。前者简单调用 `clazz.newInstance()` 便能获取到实例 `instance`。

后一种形式，也就是上文提及的用于支持类 AOP 特性的 `Wrapper` 装饰者类，它的目的是用于装饰前一种无参构造函数扩展点具类，不会为其单独创建实例，借助缓存 `cachedWrapperClasses`，`ExtensionLoader` 挨个遍历其元素，对 `instance` 加以层层包装并完成注入处理。

实例创建过程中有点特殊的是，无论对应具类的实例是从缓存中获取的，还是新创建的，都会执行自动装配和自动包装这两个步骤，具体原因不明。

```

private static final ConcurrentHashMap<Class<?>, Object> EXTENSION_INSTANCES
= new ConcurrentHashMap<>();

private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        //自动装配
        injectExtension(instance);

        //自动包装
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (CollectionUtils.isNotEmpty(wrapperClasses)) {
            for (Class<?> wrapperClass : wrapperClasses) {
                //1.取得装饰者扩展点具类
                //2.使用type获取构造函数
                //3.将instance传入调用构造函数创建示例，完成包装处理
                instance = injectExtension((T)
                    wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException("Extension instance (name: " + name + ", class:
" +
            type + ") couldn't be instantiated: " + t.getMessage(), t);
    }
}

```

JAVA

上文中有两处关于异常的点比较特殊，`throw findException(name)` 和 `exceptions.put(line, e)`，不难看出 `ExtensionLoader` 将加载具类元数据时候出现的错误延迟到创建实例时才抛出，系统运行期间这两个时段可能是紧接着发生的，也可能前后相距比较大的时差，上文可以找到线索。

## IMPORTANT

扩展点具类的实例创建过程相当简短精悍，自动装配、包装处理的风轻云淡，在研读相关实现源码时，有时会因为忽略掉这些细节而迷失方向，有些断片般的迷离。从SPI的自动包装的实现过程可以看出，一个扩展点，如果有多个包装类，那么在实例化的时候，它的任意其它非包装类扩展点具类，均会被这些类包装一遍，最后返回一个经过了层层包裹的对象，并且每一层都已经完成依赖注入处理，返回实例的最终行为是它们的总和。

### 依赖注入处理

上述用到的注入方法 `injectExtension()`，是完成自动装配的，简单讲就是遍历目标对象的所有 `setter` 方法，若方法入参不为基本类型，便试图调用 `objectFactory.getExtension()` 获取一个扩展点实例，取到了则给设入处理。

```
JAVA

private T injectExtension(T instance) {
    if (objectFactory == null) {
        return instance;
    }

    try {
        for (Method method : instance.getClass().getMethods()) {
            if (!isSetter(method)) {
                continue;
            }
            if (method.getAnnotation(DisableInject.class) != null) {
                continue;
            }

            //获取入参类型
            Class<?> pt = method.getParameterTypes()[0];
            if (ReflectUtils.isPrimitives(pt)) {
                continue;
            }

            try {
                String property = getSetterProperty(method);
                Object object = objectFactory.getExtension(pt, property);
                if (object != null) {
                    method.invoke(instance, object);
                }
            } catch (Exception e) {
                logger.error("Failed to inject via method " + method.getName()
                            + " of interface " + type.getName() + ":" + e.getMessage(),
e);
            }
        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}

private String getSetterProperty(Method method) {
    return method.getName().length() > 3 ?
        method.getName().substring(3, 4).toLowerCase()
        + method.getName().substring(4) : "";
}

//含有一个入参的public型set方法
private boolean isSetter(Method method) {
    return method.getName().startsWith("set")
        && method.getParameterTypes().length == 1
        && Modifier.isPublic(method.getModifiers());
}
```

上述代码中说明，可以使用 `@DisableInject` 注解显示告知当前具类忽略掉扩展点注入处理。另外可以认为 `objectFactory.getExtension()` 实际上就是调用了某扩展点所对应的 `ExtensionLoader` 实例由入参类型获得的 `getExtension(name)` 方法。

## 自适配具类的实例化

自适配具类中，由于其 `Class<?>` 对象是单独使用 `cachedAdaptiveClass` 缓存的，因而其实例化相对比较直接。但是当前扩展点若没有具类注解 `@Adaptive`，`ExtensionLoader` 会使用拼接字符串的方式动态生成当前扩展点接口实现的全部代码，随后完成其编译操作获得所生成具类的元数据——一个 `Class<?>` 对象。

JAVA

```
private T createAdaptiveExtension() {
    try {
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can't create adaptive extension " + type + ", cause: " + e.getMessage(), e);
    }
}

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

    //没有提供@Adaptive标注的具类时，动态创建代理具类
    return cachedAdaptiveClass = createAdaptiveExtensionClass();
}

private Class<?> createAdaptiveExtensionClass() {
    //生成代码
    String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();

    //获取classloader
    ClassLoader classLoader = findClassLoader();

    //使用SPI获取用于编译字符串得到类的Compiler
    org.apache.dubbo.common.compiler.Compiler compiler =
        ExtensionLoader.getExtensionLoader(org.apache.dubbo.common.compiler.Compiler.class)
            .getAdaptiveExtension();

    //完成字符串到Class<?>对象的转换处理
    return compiler.compile(code, classLoader);
}
```

其中 `AdaptiveClassCodeGenerator` 以 `@SPI` 注解配置的值作为默认值生成一个后缀为 "\$Adaptive" 的代理具类，该代理具类会将扩展点接口中标注了 `@Adaptive` 的方法委托给通过 `ExtensionLoader` 获取到的当前扩展点具类实例的同名方法。

假设我们定义了如下一个扩展点接口：

```
package org.apache.dubbo.common.extension.ext2  
@SPI  
public interface EgSpi {  
    @Adaptive({"key_first","key_second","key_third"})  
    String echo(UrlHolder holder, String s);  
  
    String bang(URL url, int i);  
  
    @Adaptive  
    void conn(URL url, int timeout);  
}
```

JAVA

经 AdaptiveClassCodeGenerator 处理后会生成下述代码，为了便于阅读，对代码进行了美化处理。

```

package org.apache.dubbo.common.extension.ext2;
import org.apache.dubbo.common.extension.ExtensionLoader;
public class EgSpi$Adaptive implements EgSpi {
    public String echo(UrlHolder holder, String str) {
        if (holder == null) throw new IllegalArgumentException(
            "UrlHolder argument == null");
        if (holder.getUrl() == null) throw new IllegalArgumentException(
            "UrlHolder argument getUrl() == null");

        URL url = holder.getUrl();
        String extName = url.getParameter("key_first",
            url.getParameter("key_second", url.getParameter("key_third")));

        if (extName == null)
            throw new IllegalStateException(
                "Failed to get extension (EgSpi) name from url (" +
                + url.toString() + ") use keys([eg.spi])");
        EgSpi extension = (EgSpi) ExtensionLoader.getExtensionLoader
            (EgSpi.class).getExtension(extName);
        return extension.echo(holder, str);
    }

    public String bang(URL url, int i) {
        throw new UnsupportedOperationException(
            "The method public abstract String EgSpi.bang(URL,int)" +
            " of interface EgSpi is not adaptive method!");
    }

    public void conn(URL urlArg, int timeout) {
        if (urlArg == null) throw new IllegalArgumentException("url == null");
        URL url = urlArg;
        String extName = url.getParameter("eg.spi", "test");
        if (extName == null)
            throw new IllegalStateException(
                "Failed to get extension (EgSpi) name from url (" +
                + url.toString() + ") use keys([eg.spi])");
        EgSpi extension = (EgSpi) ExtensionLoader.getExtensionLoader
            (EgSpi.class).getExtension(extName);
        extension.conn(urlArg, timeout);
    }
}

```

从上述生成代码可以看出：

1. 扩展点中没有声明 @Adaptive 注解的方法，对应生成代理具类的方法只会简单抛错处理；
2. 针对已声明 @Adaptive 注解的方法，正常生成代理逻辑：
  - a. Dubbo会找到首个类型为**URL**的入参传入配置总线获取目标具类的实例；
  - b. 如果没有类型为**URL**的入参，会试图迭代所有入参，挨个试探是否能含有 `getUrl()` 方法，若有则通过它获取到**URL**配置总线；

- c. Dubbo的SPI机制中，每一个扩展点具类都会有一个唯一对应的Key键，扩展点调用方需要使用该Key键通过 `ExtensionLoader` 动态获取到扩展点实例，调用方可以借助配置总线URL传入Key键，在URL中需要有合适的配置项，`@Adaptive` 注解的值正是为此提供支持的，值可以配置多个，按顺序取用，直到首次找到可用的扩展点实例为止；
  - d. 如果 `@Adaptive` 没有配置值，则当前扩展点所在配置总线中配置项字符串型的Key键被设定为：取扩展点接口名，按词分割，取小写，最后以“.”拼接；
  - e. 配置总线中若没有相应配置项，则使用由 `@SPI` 注解得到名词 `cachedDefaultName`，如果该值为 `null`，会因 `extName == null` 抛错；
3. 不管是直接还是间接获取的 URL 配置总线，均不能为 `null`；
  4. 生成类的包名和扩展点所处的包名一致；

## 自适配具类的源码生成

源码生成总体实现上并没有多复杂，基本原理是利用当前扩展点接口本身信息和 `@Adaptive` 注解信息生成扩展点代理具类，就其各个方法，从入参寻找到配置总线URL，以 `@Adaptive` 的值作为配置项从中取到目标扩展点具类的名称，`ExtensionLoader` 使用该名称动态地获得扩展点实例，从而最终得以将生成具类的当前委托给目标具类。

### 类声明

声明代理具类的实现比较简单，简单的对应关系如下：

1. package 语句：和扩展点同包，`type.getPackage().getName()`
2. 引入 `ExtensionLoader` 依赖：`ExtensionLoader.class.getName()`
3. 类声明：`String.format("public class %s$Adaptive implements %s", type.getSimpleName(), type.getCanonicalName())`

### 方法签名生成

众所周知，总体上Java的一个方法包含了方法名、出参、入参、方法体，以及抛出异常声明。异常也是一种出参，其中方法名取的就是当前扩展点接口的名称——`method.getName()`。凡是参数，具有自己的类型，Java中可以统一用 `Class<?>` 表示，类型可能含有泛型等复杂组成，因而生成代理具类源码时需要有完整的文本表示，也即需要使用 `.getCanonicalName()` 取得，另外一个好处是由于它使用的是全名，就避免了复杂的 `import` 导入处理。

Java中的出参是单一的，直接使用 `method.getReturnType().getCanonicalName()`，而入参和异常出参稍微复杂点，如下：

```

//generate method arguments
private String generateMethodArguments(Method method) {
    Class<?>[] pts = method.getParameterTypes();
    return IntStream.range(0, pts.length)
        .mapToObj(i -> String.format(CODE_METHOD_ARGUMENT,
            pts[i].getCanonicalName(), i))
        .collect(Collectors.joining(", "));
}

//generate method throws
private String generateMethodThrows(Method method) {
    Class<?>[] ets = method.getExceptionTypes();
    if (ets.length > 0) {
        String list =
Arrays.stream(ets).map(Class::getCanonicalName).collect(Collectors.joining(", "));
        return String.format(CODE_METHOD_THROWS, list);
    } else {
        return "";
    }
}

private String generateReturnAndInvocation(Method method) {
    String returnStatement = method.getReturnType().equals(void.class) ? "" : "return ";
    String args = IntStream.range(0, method.getParameters().length)
        .mapToObj(i -> String.format(CODE_EXTENSION_METHOD_INVOKE_ARGUMENT, i))
        .collect(Collectors.joining(", "));

    return returnStatement + String.format("extension.%s(%s);\n", method.getName(),
args);
}

```

## 方法体生成

方法体生成的代码涉及细节稍微比较多，下面挑拣几个重要的点阐述下：

先说说解决获取URL的问题，如下源码所示，大概思路是遍历所有入参，对每个入参的所有方法逐个检查，如果找到返回类型为URL且非 static 的 public 型无参 getter 方法，则直接：

```

private String generateUrlAssignmentIndirectly(Method method) {
    Class<?>[] pts = method.getParameterTypes();

    // find URL getter method
    for (int i = 0; i < pts.length; ++i) {
        for (Method m : pts[i].getMethods()) {
            String name = m.getName();
            if ((name.startsWith("get") || name.length() > 3)
                && Modifier.isPublic(m.getModifiers())
                && !Modifier.isStatic(m.getModifiers())
                && m.getParameterTypes().length == 0
                && m.getReturnType() == URL.class) {
                return generateGetUrlNullCheck(i, pts[i], name);
            }
        }
    }

    // getter method not found, throw
    throw new IllegalStateException("Failed to create adaptive class for interface " +
        type.getName()
            + ": not found url parameter or url attribute in parameters of
method " + method.getName());
}

/***
 * 1, test if argi is null
 * 2, test if argi.getXX() returns null
 * 3, assign url with argi.getXX()
 */
private String generateGetUrlNullCheck(int index, Class<?> type, String method) {
    // Null point check
    StringBuilder code = new StringBuilder();
    code.append(String.format("if (arg%d == null) throw new
IllegalArgumentException(\"%s argument == null\");\n",
        index, type.getName()));
    code.append(String.format("if (arg%d.%s() == null) throw new
IllegalArgumentException(\"%s argument %s() == null\");\n",
        index, method, type.getName(), method));

    code.append(String.format("%s url = arg%d.%s();\n", URL.class.getName(), index,
method));
    return code.toString();
}

```

## NOTE

java中的访问修饰符总共有12个，分别是 `public`、`private`、`protected`、`static`、`final`、`synchronized`、`volatile`、`transient`、`native`、`interface`、`abstract`、`strictfp`，他们可以汇总在一个2字节的int类型变量加以表达，`0000 0000 0000 0000`，从低到高，分别各占一位，元素含有该修饰符就标记为1，否则为0，比如某个方法加了 `synchronized` 修饰符，那第6位便是1。Method对象中有一个 `modifiers` 变量，2字节的int类型，它告知外界这个方法的可见性、是否为static等。这样便可以使用高效的位操作判别方法的特性，比如检测方法是否是 static 的：

```
//JAVA
//0x00000008 = 0000 0000 0000 1000

//The {@code int} value representing the {@code static}
modifier.
public static final int STATIC = 0x00000008;

public static boolean isStatic(int mod) {
    return (mod & STATIC) != 0;
}
```

上文已经提到对于没有提供值的 `@Adaptive` 方法注解，Dubbo会默认生成一个配置项，如下所示：

```
private String[] getMethodAdaptiveValue(Adaptive adaptiveAnnotation) {
    String[] value = adaptiveAnnotation.value();
    // value is not set, use the value generated from class name as the key
    if (value.length == 0) {
        String splitName = StringUtils.camelToSplitName(type.getSimpleName(), ".");
        value = new String[]{splitName};
    }
    return value;
}
```

最后在 `@Adaptive` 注解中，如果出现了 "protocol"，如果其出现在前面，会优先以它作为扩展点的名称获取扩展点实例，否则只有没有在URL中找到对应配置值是才使用它。如下述代码片段所示：

```
//①场景: @Adaptive({"protocol", "key2"})
String extName = url.getProtocol() == null ? (url.getParameter("key2")) :
url.getProtocol();

//②场景: @Adaptive({"key1", "protocol"})
String extName = url.getParameter("key1", url.getProtocol());

//③场景: @Adaptive({"key1", "protocol", "key2"})
String extName = url.getParameter("key1", url.getProtocol() == null ?
(url.getParameter("key2")) : url.getProtocol());
```

JAVA

---

关于Dubbo SPI，最重要的一环这里没有涉及，就代码生成之后的动态编译处理。搞Java业务开发，这些比较深入的技能点，一般是没法接触到的，下文我将带大家去探访。

完结

# 定时轮算法及其实现

HashedWheelTimer定时轮算法被广泛使用，netty、dubbo甚至是操作系统Linux中都有其身影，用于管理及维护大量Timer调度算法。

一个HashedWheelTimer是环形结构，类似一个时钟，分为很多槽，一个槽代表一个时间间隔，每个槽使用双向链表存储定时任务，指针周期性的跳动，跳动到一个槽位，就执行该槽位的定时任务。

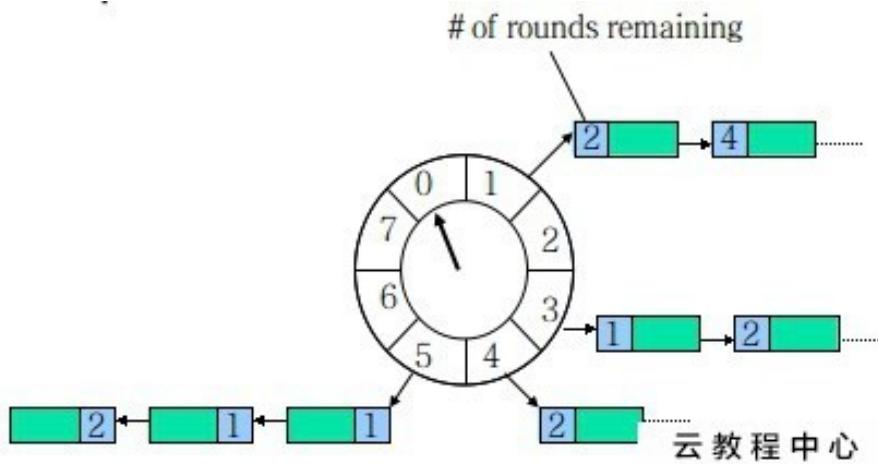


图 1: 时间轮算法

## 实现

定时轮的具体实现中，按照职责不同，可分为 时钟引擎、时钟槽、定时任务 3个主要角色，为透彻理解其实现，行文有穿插，这一部分抹掉了具体实现语言的特性。

### 定时任务——HashedWheelTimeout

在具体实现中定时任务*HashedWheelTimeout*扮演着双重角色，既是双向链表的节点，同时也是实际调度任务*TimerTask*的容器，其由引擎在滴答运行起始时刻使用**&取hash**装入对应的时钟槽。

#### 关键属性

`HashedWheelTimeout next, prev` : 当前定时任务在链表中的前驱和后继引用

`TimerTask task` : 实际被调度的任务

`long deadline` : 该时间是相对于引擎的`startTime`的，由公式 `currentTime + delay - startTime` 得到，时间单位一般为纳秒

`delay`: 任务在提交时给出的相对当前的滞后执行时间

`currentTime`: 当前内核时间

`int state`: 定时任务当前所处状态, `INIT`<sup>0</sup>—初始态, `CANCELLED`<sup>1</sup>—已被取消, `EXPIRED`<sup>2</sup>—已过期

### NOTE

状态的标记是在发起 `expire` 或 `cancel` 操作的起始瞬间完成的

`HashedWheelTimeout` 本身支持的操作并不多, 如下:

1. `remove`: 调用所属时钟槽的 `remove` 将自身从中移除, 若尚未被装入槽中, 需要额外对所属定时轮的 `pendingTimeouts` 执行 -1 处理
2. `expire`: 任务到期, 驱动 `TimerTask` 运行
3. `cancel`: 任务被提交方取消, 取消的任务被装入定时轮的 `cancelledTimeouts` 队列中, 待引擎进入下一个滴答时刻调用其 `remove` 移除自身

## 时钟槽——`HashedWheelBucket`

时钟槽实际上就是一个用于缓存和管理定时任务的双向链表容器。每一个节点也即一个定时任务。它持有链表的首尾两个节点, 由于每个节点均持有前驱和后继的引用, 因此利用链表的这个特性可以完成如下操作:

1. `addTimeout`: 新增尾节点, 添加任务
2. `pollTimeout`: 移除首节点, 取出头部任务
3. `remove`: 根据引用移除指定节点, 被移除任务已被处理或被取消, 对所属定时轮的 `pendingTimeouts` 相应 -1 处理
4. `clearTimeouts`: 循环调用 `pollTimeout` 获取到所有未超时或者未被取消的任务
5. `expireTimeouts`: 从首节点开始循环遍历槽中所有节点, 调用 `remove` 取出到期任务运行 `expire` 或直接移除 `remove` 被取消的任务, 对其它正常任务的剩余轮数执行 -1 操作

## 时钟引擎——`HashedWheelTimer`

时钟引擎有节律地周期性运作, 总是根据当前时钟滴答选定对应的时钟槽, 从链表头部开始迭代, 对每一个任务计算出其是否属于当前时钟周期, 属于则取出运行, 否则便将对剩下时钟周期数执行减一操作。

另外，引擎维持着两个缓存定时任务的阻塞队列，其中一个用于接受外界断断续续地投递进来的，另外一个则用于缓存那些主动取消的，引擎需要在滴答开始期间先行将他们装入对应的时钟槽或从中移除他们。

### 关键属性

`Queue<HashedWheelTimeout> timeouts、cancelledTimeouts`：队列，用于缓存外界主动提交或取消的任务

`int workerState`：定时轮当前所处状态：

**NOTE** 状态值

`init0`——初始态

`started1`——已开始运行

`shutdown2`——已结束运行

`startTime`：当前定时轮正式开始调度任务的时间，此后所有提交的定时任务第一个任务提交的开始，引擎就开始正式执行了，均以该时间点作为起点

`ticks`：滴答，由时钟引擎维护，是步长为1的单调递增计数，也即 `ticks+=1`

`ticksDuration`：滴答时长，每轮询一个特定时钟槽代表代表走完一个滴答时长

`pendingTimeouts` 当前定时轮实时任务剩余数

`n`：时钟轮槽数为n，不一定和期望达到的槽数一致，取大于且最靠近的2的幂次方值，其计算公式为  $n=2^x$

`mask`：掩码，`mask = n - 1`，执行 `ticks & mask` 便能定位到对应位置的时钟槽，效果上相当于 `ticks % (mask + 1)`，由n这个2的幂次方保证

## 引擎内核——Worker

时钟引擎实际上分为对外接口和调度运行两部分，可以想象内核就是一个引擎的心脏起搏器驱动着定时轮的运行，完成任务的调度，实现上对应一个工作线程，为方便理解，先单独阐述他们所依赖的内核状态。

### 内核状态

对于任何引擎来说，状态机是其关键组成，因此状态值的控制对其而言是至关重要，因而将这部分作为单独的部分阐述。内核状态有定时轮维护管理，对外提供的接口都要借助它实现。初始时便为init状态，当引擎被设计成不可复活时，便不存在 `init/started/shutdown → init` 这样的迁移过程。

## NOTE `start()`

### *init → started*

于引擎的整个生命周期而言，这个状态的迁移过程只允许发生一次，实现中会结合`startTime`做防御性保护，直到整个过程完成为止

### *started → started*

表示引擎已经被启用，一般直接忽略，否则也会等效于什么也不做

### *shutdown → started*

定时轮一般被设计为不能复活的，这种情况下该过程是不允许发生的，属于外界调用方的越界行为

## NOTE `stop()`

### *init → shutdown*

尚未开始就进入终结状态，一般发生在对定时轮已经完成初始化，但尚未给其提交任务或调用过`start()`操作。但还有另外一种特殊的情形，在多个线程对同一定时轮进行操作时发生争用，一个线程在另外一个线程刚开始进入`start()`操作时，调用了`stop()`

### *started → shutdown*

正常的引擎关闭操作，一旦进入该过程，会持续到引擎完全终止，对应到实现上就是结束Work线程

### *shutdown → shutdown*

该迁移过程没有实际语义，一般直接跳过

## 外部接口

这部分内容实际上已经在内核状态一节已经有过具体阐述

`start`：用于定时轮开启引擎，但外界不一定需要调用此方法启用定时轮，因为外界每次调用`newTimeout()`提交任务时，定时轮都会主动调用该接口，以确保引擎已经处于运行状态。

`stop`：完成定时轮引擎的关闭过程，返回未被处理的定时任务

`Timeout newTimeout(TimerTask task, long delay, TimeUnit unit)`：用于向引擎提交任务，在任务被正式加入`timeouts`队列之前：1) 定时轮会首先调用`start()`确保引擎已经启动；然后为加入的`Timeout`计算出`deadline`值。

## 调度运行

有了以上的分析，对定时轮的任务调度也就不难理解了，简单而言就是周期性的执行滴答操作，对应如下几个操作：

1. 等待进入滴答周期
2. 时钟转动，滴答周期开始：
  - a. 将外界主动取消的装载在`cancelledTimeouts`队列的任务逐个移除
  - b. 将外界提交的装载在`timeouts`队列的任务逐个载入对应的时钟槽里
3. 根据当前`tick`定位对应时钟槽，执行其中的定时任务
4. 检测引擎内核状态是否已经被终止，若未被终止，则循环执行上述操作，否则往下继续执行
5. 将下述方式获取到未被处理的任务加入`unprocessedTimeouts`队列：
  - a. 遍历时钟槽调用`clearTimeouts()`
  - b. 对`timeouts`队列中未被加入槽中循环调用`poll()`
6. 移除最后一个滴答周期后加入到`cancelledTimeouts`队列任务

### IMPORTANT

相邻两个滴答周期的开始时间理论上来说是等距的，但是结束时间则会随该周期所需处理任务的数目及时长有所变化。因而引擎剩下的休眠时间需要使用如下公式获得：

$$\text{tickDuration} * (\text{tick} + 1) - (\text{currentTime} - \text{startTime})$$

## 定时轮在dubbo中的应用

实际上，定时轮算法并不直接用于等周期性的执行某些提交任务，向其提交的任务只会到期执行一次，但具体应用中，会利用每次任务的执行，调用`newTimeout()`提交`Timer`所引用的当前任务，使其在若干单位时间后重新继续执行。这样做的好处是，如果诸如IO等耗时任务，甚至是某些原因导致的当下执行任务卡住比较长时间，后面不会有同样的任务不断提交进来，而导致任务堆积至无法处理。可见这里说的额周期性任务不是严格固定每x单位时间执行一次的任务。

Dubbo中对定时轮的应用主要体现在如下几个方面：

1. 失败重试

- a. 注册 **Register**
- b. 取消注册 **Unregister**
- c. 订阅 **Subscribe**
- d. 取消订阅 **Unsubscribe**

## 2. 周期任务

- a. 心跳 **Heartbeat**
- b. 重连 **Reconnect**
- c. 下线 **CloseChannel**

定时轮用单一的线程去管理触发Task的运行，Task执行期间，不能直接抛异常，否则会导致整个定时轮引擎的奔溃而使得提交的后续任务无法执行。Task的模式如下：

JAVA

## IMPORTANT

```
try {
    if (sthCheck()) {
        logger.warn("Sth happended");
        doBuz();
    }
} catch (Throwable t) {
    logger.warn("Exception when do sth ", t);
}
```

## 周期任务

在dubbo中每一个连接被表征为一个Channel通道，dubbo节点间建立连接相互通信，单个节点需要维护和多个连入节点的连接：①通过持续发送心跳检测以保持连接 ②对超过了一定时间段处于空闲状态的连接进行下线处理；③对已经掉线非下线处理的Channel进行重连处理。

基本的步骤如下：

1. 滴答运行时Task通过回调获得当前节点的所有连入Channel
2. 对没有被关闭的节点执行实际的任务操作，比如心跳
3. 通过`volatile`的可见性保证属性检测当前任务是否被取消，是返回，否继续
4. 若定时轮是否还在运行，则使用其提供的`newTimeout()`提交一个新的Task

以下结合源码进行分析：

```
public abstract class AbstractTimerTask implements TimerTask {  
    /**  
     * 该属性比较关键，真正执行Task操作的是定时轮所持有的线程，而唤起``cancel()``操作的是提交任务的其它线程  
     */  
    protected volatile boolean cancel = false;  
  
    public void cancel() {  
        this.cancel = true;  
    }  
  
    ....//省略部分代码  
  
    private void reput(Timeout timeout, Long tick) {  
        if (timeout == null || tick == null) {  
            throw new IllegalArgumentException();  
        }  
  
        if (cancel) {  
            return;  
        }  
  
        Timer timer = timeout.timer();  
        if (timer.isStop() || timeout.isCancelled()) {  
            return;  
        }  
  
        timer.newTimeout(timeout.task(), tick, TimeUnit.MILLISECONDS);  
    }  
  
    @Override  
    public void run(Timeout timeout) throws Exception {  
        Collection<Channel> c = channelProvider.getChannels();  
        for (Channel channel : c) {  
            if (channel.isClosed()) {  
                continue;  
            }  
            doTask(channel);  
        }  
        reput(timeout, tick);  
    }  
  
    protected abstract void doTask(Channel channel);  
  
    interface ChannelProvider {  
        Collection<Channel> getChannels();  
    }  
}
```

从以上代码可以看出，利用定时轮实现间隔时间任务的模式比较固定，如下：

JAVA

```
//保证立马被定时轮获知任务已被取消
protected volatile boolean cancel = false;

public void run(Timeout timeout) throws Exception {
    //执行任务业务逻辑
    doTask()
    //重新
    reput(timeout, tick);
}

private void reput(Timeout timeout, Long tick) {
    NOTE
    if (cancel) {
        return;
    }

    //确认定时轮还处于运行状态
    Timer timer = timeout.timer();
    if (timer.isStop() || timeout.isCancelled()) {
        return;
    }

    //向定时轮重新提交一个新的__Task__，指定tick单位时间后执行
    timer.newTimeout(timeout.task(), tick,
    TimeUnit.MILLISECONDS);
}
```

周期性任务中对每一个Channel所做的事情比较简单，实际上是在满足条件的情况下调用channel的指定操作

### 1. 心跳—— channel.send(req)

```
Request req = new Request();
req.setVersion(Version.getProtocolVersion());
//双边通讯
req.setTwoWay(true);
//事件类型：心跳
req.setEvent(Request.HEARTBEAT_EVENT);
```

JAVA

1. 重连—— ((Client) channel).reconnect()

2. 下线—— channel.close()

## 失败重试

网络情况的的复杂多变性，使得一件原本在单机上很轻易的事情，分布式应用中，为确保某类型的操作能发生可能需要重试多次。除了catch到异常后进行重试和对重试次数有规定外，和上述的周期任务实现几乎一样。模式如下：

```

/**
 * times of retry.
 * retry task is execute in single thread so that the times is not need volatile.
 */
//重试次数并没有被申明为volatile，原因是该变量只会被定时轮引擎中的工作线程所使用到，投递任务的那个线程
并没有直接接触
private int times = 1;
...
protected void reput(Timeout timeout, long tick) {
    if (timeout == null) {
        throw new IllegalArgumentException();
    }

    Timer timer = timeout.timer();
    if (timer.isStop() || timeout.isCancelled() || isCancel()) {
        return;
    }
    times++;
    timer.newTimeout(timeout.task(), tick, TimeUnit.MILLISECONDS);
}

@Override
public void run(Timeout timeout) throws Exception {
    if (timeout.isCancelled() || timeout.timer().isStop() || isCancel()) {
        // other thread cancel this timeout or stop the timer.
        return;
    }
    if (times > retryTimes) {
        // reach the most times of retry.
        logger.warn("Final failed to execute task " + taskId + ", url: "
            + url + ", retry " + retryTimes + " times.");
        return;
    }
    if (logger.isInfoEnabled()) {
        logger.info(taskId + " : " + url);
    }
    try {
        doRetry(url, registry, timeout);
    } catch (Throwable t) {

        // Ignore all the exceptions and wait for the next retry
        logger.warn("Failed to execute task " + taskId + ", url: "
            + url + ", waiting for again, cause:" + t.getMessage(), t);

        // reput this task when catch exception.
        reput(timeout, retryPeriod);
    }
}

protected abstract void doRetry(URL url, FallbackRegistry registry, Timeout timeout);

```

# Dubbo注册中心

注册中心在微服务架构中的作用举足轻重，有了它服务提供者Provider和注册者Consumer就能感知彼此。从下述Dubbo架构图示中可知：①Provider 从容器启动后的初始化阶段便会向注册中心完成注册操作；②Consumer启动初始化阶段完成对所需Provider的订阅操作；③另外在Provider发生变化，需要通知对应注册了监听此变化的Consumer。

## Dubbo Architecture

..... init    ..... async    —— sync

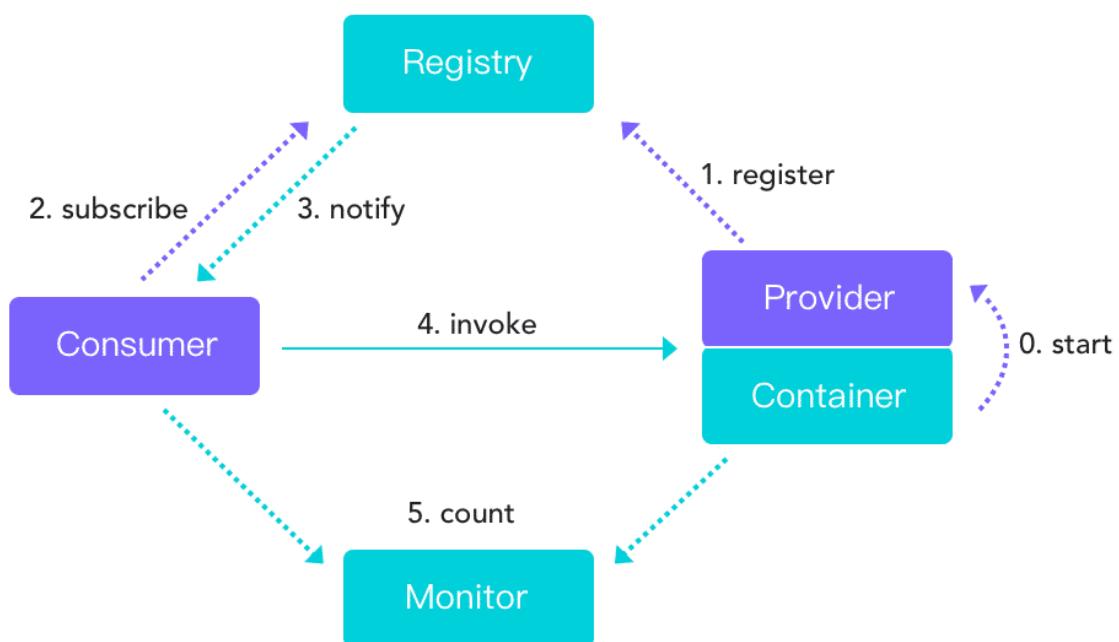


图 1: Dubbo 架构

Registry只是Consumer和Provider能感知彼此状态变化的一种便捷途径而已，彼此的实际通讯交互过程是直接进行的，于Registry是透明无感的。Provider状态发生了变化，会由Registry主动推送订阅了该Provider的Consumer们，这保证了Consumer感知Provider状态变化的及时性，和具体业务需求逻辑交互解耦，也提升了系统的稳定性。



Dubbo中存在很多概念，有些理解起来就觉得特别费劲。如本文的**Registry**，翻译过来的意思是**注册中心**，但它是应用本地的，真正的注册中心是其他独立部署的进程，或进程组成的集群，比如**Zookeeper**。本地的**Registry**通过和**Zookeeper**等进行实时的信息同步，维持这些内容的一致性，从而实现了**注册中心**这个特性。另外就**Registry**而言，**Consumer**和**Provider**只是个用户视觉的概念，他们一律被看做是一份**URL**数据。

## Registry定义

注册中心实现目前在业界是一门比较成熟的技术，有多种方案，为了搞清楚Dubbo的注册中心到底是如何运作的，本文将根据**AbstractRegistry**源码进行逐步解析。在进一步阐述之前先看看其实现接口的定义

```
    /**
     * 注册数据，比如：提供者地址，消费者地址，路由规则，覆盖规则，等数据。
     *
     * 注册需处理契约：<br>
     * 1. 当URL设置了check=false时，注册失败后不报错，在后台定时重试，否则抛出异常。<br>
     * 2. 当URL设置了dynamic=false参数，则需持久存储，否则，当注册者出现断电等情况异常退出时，需自动
     *    删除。<br>
     * 3. 当URL设置了category=routers时，表示分类存储，缺省类别为providers，可按分类部分通知数据。
     <br>
     * 4. 当注册中心重启，网络抖动，不能丢失数据，包括断线自动删除数据。<br>
     * 5. 允许URI相同但参数不同的URL并存，不能覆盖。<br>
     *
     * @param url 注册信息，不允许为空，如：dubbo://10.20.153.10/com.alibaba.foo.BarService?
     version=1.0.0&application=kylin
     */
    void register(URL url);

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

    /**
     * 订阅符合条件的已注册数据，当有注册数据变更时自动推送。
     *
     * 订阅需处理契约：<br>
     * 1. 当URL设置了check=false时，订阅失败后不报错，在后台定时重试。<br>
     * 2. 当URL设置了category=routers，只通知指定分类的数据，多个分类用逗号分隔，并允许星号通配，表
     *    示订阅所有分类数据。<br>
     * 3. 允许以interface,group,version,classifier作为条件查询，如：
     interface=com.alibaba.foo.BarService&version=1.0.0<br>
     * 4. 并且查询条件允许星号通配，订阅所有接口的所有分组的所有版本，或：
     interface=*&group=*&version=*&classifier=*<br>
     * 5. 当注册中心重启，网络抖动，需自动恢复订阅请求。<br>
     * 6. 允许URI相同但参数不同的URL并存，不能覆盖。<br>
     * 7. 必须阻塞订阅过程，等第一次通知完后再返回。<br>
     *
     * @param url 订阅条件，不允许为空，如：
     consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&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。
     *
```

```

    * 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);

/**
 * 查询符合条件的已注册数据, 与订阅的推模式相对应, 这里为拉模式, 只返回一次结果。
 *
 * @see com.alibaba.dubbo.registry.NotifyListener#notify(List)
 * @param url 查询条件, 不允许为空, 如:
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&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);
}

}

```

## 具体实现

在Dubbo中注册中心的实现方式有多种, 包括: ①Zookeeper; ②Etcd; ③Consul; ④Redis; ⑤Multicast。如上文所示, 他们提供的最基础的功能就是 "注册、订阅、通知" 这三项, 有着很强的共性。最后在微服务这种分布式架构系统中, 容错处理不可或缺, 注册中心使用本地缓存文件作为容错机制。以下将从这4方面分开阐述。



Dubbo中, URL的有着很强的通用性, 它可以完全用于表征某类型的节点, 比如Consumer、Provider

本文中Consumer和Provider代表并不完全就是微服务中所指的服务消费端和服务提供端, 是相对于注册、订阅和通知这三个操作而言的, 任意节点Node都可以根据自身需要在Registry注册成Provider或者订阅它所感兴趣的由其它Provider触发的事件。

### 扫盲——URL、Unmodifiable View、Node

1. URL, 统一资源定位符, 顾名思义是一个在系统框架中唯一界定资源的标识符可以简单理解为字符串, 在Dubbo中URL是一个复杂的存在, 作为公共契约的承载体或称: 统一配置模型、配置总线, 具体参考: [URL统一模型 @ Dubbo](#)  
(<https://dubbo.apache.org/zh-cn/blog/introduction-to-dubbo-url.html>)
2. 多线程环境下, 即便线程安全的容器, 简单的通过获取其引用, 后续对其迭代, 迭代过程中, 其所含元素包括个数及内容可能随时会改变, 为了获得所在线程的当下视图, 需要使用到Java集合框架提供的 `unmodifiableXXX` 辅助方法取得当下非可变视图, 如下:

```
public Set<URL> getRegistered() {  
    return Collections.unmodifiableSet(registered);  
}  
  
public Map<URL, Set<NotifyListener>> getSubscribed() {  
    return Collections.unmodifiableMap(subscribed);  
}  
  
public Map<URL, Map<String, List<URL>>> getNotified() {  
    return Collections.unmodifiableMap(notified);  
}
```

JAVA

3.在Dubbo中， Registry、 Consumer、 Provider等能够独立部署的节点， 均被表示为Node节点，各个具体实现节点需要向往提供获取自身URL、 可用状态 检查的方法， 以及销毁操作， 如下：

```
public interface Node {  
  
    URL getUrl();  
  
    boolean isAvailable();  
  
    void destroy();  
}
```

JAVA

## 扫盲——Consumer 和 Provider 的匹配

Registry 实现中， 使用 UrlUtils.isMatch(consumerUrl, providerUrl) 来检查 Consumer 和 Provider 的匹配问题， 其代码实现虽然很短， 但没那么容易理解， 和订阅、 通知密切相关， 有必要在这边剖析一下其细节。

首先是关于 Service Key 的取值， 下述两个URL对应的值分别为 com.dubbo.interfaceName<sup>①</sup> 和 org.dubbo.interfaceName<sup>②</sup>， 也就是说在 path 和参数 interface 二者中， 优先取后者。于 Consumer来说， 可以使用 interface 作为被引用微服务的标识， 而 path 留作它用。

```
//①  
dubbo://admin:hello1234@10.20.130.230:20880/org.dubbo.interfaceName?  
interface=com.dubbo.interfaceName&group=group1&version=1.0.0  
//②  
dubbo://admin:hello1234@10.20.130.230:20880/org.dubbo.interfaceName?  
group=group1&version=1.0.0
```

JAVA

另外方法中还用到了 isMatchCategory(category, categories) ， 根据其定义的规则， 下述情况下 category 是和 categories 匹配的：

1. categories 值为空， category 值为 "providers" ；
2. categories 值为 "\*" ；

3. categories 不包含 "\_" + category (含"\_"的情况下);

4. categories 包含 或 等于 category ;

总体而言，按优先顺序，二者需要在 Service Key 、 category 、 enabled 、 group 、 version 、 classifier 这几项均匹配。

```

public static boolean isMatch(URL consumerUrl, URL providerUrl) {
    String consumerInterface = consumerUrl.getServiceInterface();
    String providerInterface = providerUrl.getServiceInterface();

    //等价于 consumerI != * && providerI != * && consumerI != providerI
    //也就是consumerI和providerI都没有设置为通配符时，二者又不相等的情况，肯定不匹配
    if (!ANY_VALUE.equals(consumerInterface)
        || ANY_VALUE.equals(providerInterface)
        || StringUtils.isEqual(consumerInterface, providerInterface))) {
        return false;
    }

    //若配置的consumer配置的category范围不包含provider所配置，也不匹配
    if (!isMatchCategory(providerUrl.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY),
        consumerUrl.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY))) {
        return false;
    }

    //若provider配置了enabled=false，而consumer没有配置enabled=*, 则也不匹配
    if (!providerUrl.getParameter(ENABLED_KEY, true)
        && !ANY_VALUE.equals(consumerUrl.getParameter(ENABLED_KEY))) {
        return false;
    }

    String consumerGroup = consumerUrl.getParameter(GROUP_KEY);
    String consumerVersion = consumerUrl.getParameter(VERSION_KEY);
    String consumerClassifier = consumerUrl.getParameter(CLASSIFIER_KEY, ANY_VALUE);

    String providerGroup = providerUrl.getParameter(GROUP_KEY);
    String providerVersion = providerUrl.getParameter(VERSION_KEY);
    String providerClassifier = providerUrl.getParameter(CLASSIFIER_KEY, ANY_VALUE);

    //最后这里要求group、version、还有consumerClassifier均能匹配
    //在consumer一端，三者均能使用通配符，通配符表示匹配任何值
    return (ANY_VALUE.equals(consumerGroup)
        || StringUtils.isEqual(consumerGroup, providerGroup)
        || StringUtils.contains(consumerGroup, providerGroup))
        && (ANY_VALUE.equals(consumerVersion)
        || StringUtils.isEqual(consumerVersion, providerVersion))
        && (consumerClassifier == null
        || ANY_VALUE.equals(consumerClassifier)
        || StringUtils.isEqual(consumerClassifier, providerClassifier));
}

public static boolean isMatchCategory(String category, String categories) {
    if (categories == null || categories.length() == 0) {
        return DEFAULT_CATEGORY.equals(category);
    } else if (categories.contains(ANY_VALUE)) {
        return true;
    } else if (categories.contains(REMOVE_VALUE_PREFIX)) {
        return !categories.contains(REMOVE_VALUE_PREFIX + category);
    } else {
        return categories.contains(category);
    }
}

```

## 注册

Provider在启动后的初始化阶段，会主动向注册中心提交注册信息，同样，需要下线处理时，也会主动发出注销申请。 Provider注册相关的逻辑其实很简单，如下：

```
public abstract class AbstractRegistry implements Registry {
```

JAVA

```
    private final Set<URL> registered = new ConcurrentHashSet<>();  
    ...  
  
    public void register(URL url) {  
        if (url == null) {  
            throw new IllegalArgumentException("register url == null");  
        }  
        registered.add(url);  
    }  
  
    public void unregister(URL url) {  
        if (url == null) {  
            throw new IllegalArgumentException("unregister url == null");  
        }  
        registered.remove(url);  
    }  
  
    public void destroy() {
```

Set<URL> destroyRegistered = new HashSet<>(getRegistered());  
 if (!destroyRegistered.isEmpty()) {

for (URL url : destroyRegistered) {  
 //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。

```
                if (url.getParameter(Constants.DYNAMIC_KEY, true)) {  
                    try {  
                        unregister(url);  
                    } catch (Throwable t) {  
                        logger.warn("Failed to unregister url " + url + " to registry "  
                                + getUrl() + " on destroy, cause: "  
                                + t.getMessage(), t);  
                    }  
                }  
            }  
        }  
        ...  
    }
```

//恢复方法，在注册中心断开，重连成功的时候

```
    protected void recover() throws Exception {  
        //把内存缓存中的registered取出来遍历进行注册  
        Set<URL> recoverRegistered = getRegistered();  
        if (!recoverRegistered.isEmpty()) {  
            for (URL url : recoverRegistered) {  
                register(url);  
            }  
        }  
        ...  
    }  
    ...  
}
```

从上述源码可知，Dubbo使用了并发包下的 `ConcurrentHashSet` 作为所有Provider的注册信息容器，确保了线程安全和Provider注册资源URL的全局唯一性。

### recover() 源码疑惑解析

上述关于recover的代码片段，单看起来会觉得很蹊跷，在试图恢复时，仅仅简单地从内存缓存中获取了所有已注册provider的URL视图，然后逐个调用 `register()` 重新注册，然而 `register()` 也仅仅是将URL加入到 `registered` 集合中。

总体而言，Java是一门纯OOP的编程语言，其继承和多态特性，决定了一个对象的某个方法的具体行为最终取决于该方法的覆写轨迹。也就是说，该方法实际的执行操作由运行时对象决定的，如果其对应类覆写了父类方法时，调用了 `super.()`，则父类的行为得到保留，否则会被无情地擦除。

最终的注册中心实现类并不是直接继承于 `AbstractRegistry` 的，Dubbo要求相应注册中心能够提供基本的重试机制以保证注册中心的可用性。`FallbackRegistry` 作为其直接继承类，覆写了 `register()` 方法，包含了一系列重试相关逻辑，乃至调用最终在 Zookeeper、Etcd集群完成实际注册的 `doRegister(URL url)` 方法。

## 订阅

Consumer在启动后的初始化阶段，会主动向注册中心提交订阅请求，同样，需要下线处理时，也会主动发出注销申请。每一个Consumer都有唯一的URL，它可以同时订阅多个感兴趣的事件，具体参考下述源码：

```

public abstract class AbstractRegistry implements Registry {

    private final ConcurrentHashMap<URL, Set<NotifyListener>> subscribed = new
    ConcurrentHashMap<>();

    public void subscribe(URL url, NotifyListener listener) {
        if (url == null) {
            throw new IllegalArgumentException("subscribe url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("subscribe listener == null");
        }
        Set<NotifyListener> listeners = subscribed.computeIfAbsent(url, n -> new
        ConcurrentHashSet<>());
        listeners.add(listener);
    }

    public void unsubscribe(URL url, NotifyListener listener) {
        if (url == null) {
            throw new IllegalArgumentException("unsubscribe url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("unsubscribe listener == null");
        }
        Set<NotifyListener> listeners = subscribed.get(url);
        if (listeners != null) {
            listeners.remove(listener);
        }
    }

    //恢复方法，在注册中心断开，重连成功的时候
    protected void recover() throws Exception {
        ...
        //把内存缓存中的subscribed取出来遍历进行订阅
        Map<URL, Set<NotifyListener>> recoverSubscribed = new HashMap<>
        (getSubscribed());
        if (!recoverSubscribed.isEmpty()) {
            for (Map.Entry<URL, Set<NotifyListener>> entry :
            recoverSubscribed.entrySet()) {
                URL url = entry.getKey();
                for (NotifyListener listener : entry.getValue()) {
                    subscribe(url, listener);
                }
            }
        }
    }

    public void destroy() {
        ...
        //把内存缓存中的subscribed取出来遍历进行取消订阅
        Map<URL, Set<NotifyListener>> destroySubscribed = new HashMap<>
        (getSubscribed());
        if (!destroySubscribed.isEmpty()) {
            for (Map.Entry<URL, Set<NotifyListener>> entry :
            destroySubscribed.entrySet()) {

```

JAVA

```

    URL url = entry.getKey();
    for (NotifyListener listener : entry.getValue()) {
        try {
            unsubscribe(url, listener);
        } catch (Throwable t) {
            logger.warn("Failed to unsubscribe url " + url + " to registry
"
                + getUrl() + " on destroy, cause: " + t.getMessage(), t);
        }
    }
}
...
}

```

通过上述源码发现Consumer在订阅感兴趣的事件时，传入的参数只包含URL和NotifyListener两个参数，这和直觉有些冲突，订阅的事件应该来自提供服务某些候选的Provider，那Dubbo怎么确定有哪些候选项呢？这得回到上文中提到 RegistryService 接口定义及URL实现语义上，实际上接口定义讲得很清楚，URL的Path部分标识订阅服务的Consumer，而Query参数部分则是用于匹配候选Provider的。

## 通知

Consumer和Provider向Registry提供了订阅和注册数据后，Registry会在Provider的状态发生了变化时，根据Consumer的订阅情况，触发相对应事件，将 Consumer所感兴趣的Provider数据 notify() 给Consumer。同时Consumer也可以主动 lookup() 获取查询所匹配的Provider数据。想进一步搞清楚 3者间关系，先看看其数据定义：

```

private final ConcurrentHashMap<URL, Set<NotifyListener>> subscribed
    = new ConcurrentHashMap<>();

private final Set<URL> registered
    = new HashSet<>();

//key 【URL】：表征Consumer的URL
//value 【Map<String, List<URL>>】：
// 键为分类标识，值为该分类下所有对应Provider的URL
private final ConcurrentHashMap<URL, Map<String, List<URL>>> notified
    = new ConcurrentHashMap<>();

```

JAVA

Provider向Registry注册，提供自身的表征信息URL，Consumer则向Registry订阅其所关注的事件 NotifyListener，在发生相应事件时，Registry会将所有Provider表征信息URL按Category分组 notify() 给Consumer。然而这并不是全貌，subscribed 集合的Key中还潜藏着另外一层含义，其URL 携带的参数，用于明确告知Registry，符合哪些特征的Provider表征信息才是它所真正所关注的。也就是说有了这两组数据，就大致知道Provider状态有变动时 该通知哪些Consumer了。

`notified` 用于按Category分组装填那些最近已经通知过Consumer的所有Provider表征信息，这里记录的信息是便于实现 `lookup()` 的。

```

public interface NotifyListener {
    void notify(List<URL> urls);
}

public abstract class AbstractRegistry implements Registry {

    /**
     * Notify changes from the Provider side.
     *
     * @param url      consumer side url
     * @param listener listener
     * @param urls     provider latest urls
     */
    protected void notify(URL url, NotifyListener listener, List<URL> urls) {
        if (url == null) {
            throw new IllegalArgumentException("notify url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("notify listener == null");
        }
        if ((CollectionUtils.isEmpty(urls))
            //意即Consumer不是匹配任意Interface
            && !Constants.ANY_VALUE.equals(url.getServiceInterface())) {
            logger.warn("Ignore empty notify urls for subscribe url " + url);
            return;
        }

        //按CATEGORY进行分组，将表征Provider的urls中Consumer感兴趣的那些装进result中
        Map<String, List<URL>> result = new HashMap<>();
        for (URL u : urls) {
            //检测Consumer对当前Provider是否感兴趣
            if (UrlUtils.isMatch(url, u)) {
                String category = u.getParameter(Constants.CATEGORY_KEY,
                    Constants.DEFAULT_CATEGORY);
                List<URL> categoryList = result.computeIfAbsent(category, k -> new
                    ArrayList<>());
                categoryList.add(u);
            }
        }
        if (result.size() == 0) {
            return;
        }

        //使用表征Consumer的url获得对应感兴趣的
        Map<String, List<URL>> categoryNotified = notified.computeIfAbsent(url, u ->
            new ConcurrentHashMap<>());
        for (Map.Entry<String, List<URL>> entry : result.entrySet()) {
            String category = entry.getKey();
            List<URL> categoryList = entry.getValue();

            //将上述按CATEGORY分组的所有Provider置入notified中
            categoryNotified.put(category, categoryList);

            //针对当前CATEGORY分组下的所有Provider回调listener
            listener.notify(categoryList);
            // We will update our cache file after each notification.
        }
    }
}

```

```

        // When our Registry has a subscribe failure due to network jitter, we can
        // return at least the existing cache URL.
        saveProperties(url);
    }
}

//当参数中所有providers信息有变动时，通知所有订阅他们的consumer们这一变动
//这些providers属于同一个微服务部署的多个不同实例
protected void notify(List<URL> urls) {
    if (CollectionUtils.isEmpty(urls)) {
        return;
    }

    for (Map.Entry<URL, Set<NotifyListener>> entry : getSubscribed().entrySet()) {
        URL url = entry.getKey();
        //这里说明传入的一组URL是类似的，要么都能匹配到Consumer，否则全不
        if (!UrlUtils.isMatch(url, urls.get(0))) {
            continue;
        }

        Set<NotifyListener> listeners = entry.getValue();
        if (listeners != null) {
            for (NotifyListener listener : listeners) {
                try {
                    notify(url, listener, filterEmpty(url, urls));
                } catch (Throwable t) {
                    logger.error("Failed to notify registry event, urls: " +
                        urls + ", cause: " + t.getMessage(), t);
                }
            }
        }
    }
}

@Override
public List<URL> lookup(URL url) {
    List<URL> result = new ArrayList<>();

    //将所有按Category分组好的Provider的URL信息装入到一维的result中
    Map<String, List<URL>> notifiedUrls = getNotified().get(url);
    if (notifiedUrls != null && notifiedUrls.size() > 0) {
        for (List<URL> urls : notifiedUrls.values()) {
            for (URL u : urls) {
                if (!Constants.EMPTY_PROTOCOL.equals(u.getProtocol())) {
                    result.add(u);
                }
            }
        }
    } else {
        //使用原子引用类型保存Provider的URL信息，封装在List中，最初其内容为空
        final AtomicReference<List<URL>> reference = new AtomicReference<>();
        //生成NotifyListener，Dubbo会在Providers状态发生变化时notify给根据URL特征能匹配的
        Consumer<List<URL>> listener = reference::set;
        //完成订阅操作，确保listener能被回调到
        subscribe(url, listener); // Subscribe logic guarantees the first notify to
    }
}

```

```

return
    //获取reference中的内容，并将获取到值设到result返回值中，感觉这里大多数情况下是无用的,
    //除非多线程环境下，刚好要执行这语句的时候，CPU资源已经让渡给其它线程notify操作了
    List<URL> urls = reference.get();
    if (CollectionUtils.isNotEmpty(urls)) {
        for (URL u : urls) {
            if (!Constants.EMPTY_PROTOCOL.equals(u.getProtocol())) {
                result.add(u);
            }
        }
    }
    return result;
}
...
}

```

## 容错设计

We will update our cache file after each notification. When our Registry has a subscribe failure due to network jitter, we can return at least the existing cache URL.

在 `notify()` 实现中，有上述注释，大意是因为网络抖动导致订阅失败时，为保证服务的可靠性，作为最次的方案，订阅者可向注册中心调用 `public List<URL> getCacheUrls(URL url)` 获取所有匹配到的最近注册在Registry的Provider的URL。

在分布式架构中，作为担纲PRC通讯框架的Dubbo，它解决的是微服务间协作的难题，大多数情况下，仅仅作为Provider和Consumer的一套基础依赖和应用一起打包部署，Dubbo自身并没有单独部署，本文所述的Registry也仅仅是其中一个依赖模块，由其完成到Zookeeper、Etcd、Consul等Server的注册订阅操作。

大致的设计方案如下： Registry在每次 `notify()` 通知时，均将当前被`notify`的Consumer能匹配到所有Provider的URL组成的List写入到Properties中，它的Key值为 Consumer的URL中获取到的 `ServiceKey`，根据需求同步或异步地将这些内容在文件锁的辅助下互斥地保存到对应的 `/.dubbo/dubbo-registry-[当前应用名]-[当前 Registry所在的IP地址].cache` 文件中。为了确保在多线程环境下文件的保存不发生冲突，Dubbo使用了基于版本号的乐观锁，只有获取到了最新的版本号，才能执行。

相关代码如下：

```
public abstract class AbstractRegistry implements Registry {  
  
    private static final int MAX_RETRY_TIMES_SAVE_PROPERTIES = 3;  
  
    private final AtomicInteger savePropertiesRetryTimes = new AtomicInteger();  
  
    // Local disk cache, where the special key value.registries records the list of  
    // registry centers, and the others are the list of notified service providers  
    private final Properties properties = new Properties();  
  
    private final AtomicLong lastCacheChanged = new AtomicLong();  
  
    //将文件中缓存的信息恢复到缓存properties中  
    private void loadProperties() {  
        if (file != null && file.exists()) {  
            InputStream in = null;  
            try {  
                in = new FileInputStream(file);  
                properties.load(in);  
                if (logger.isInfoEnabled()) {  
                    logger.info("Load registry cache file " + file + ", data: " +  
properties);  
                }  
            } catch (Throwable e) {  
                logger.warn("Failed to load registry cache file " + file, e);  
            } finally {  
                if (in != null) {  
                    try {  
                        in.close();  
                    } catch (IOException e) {  
                        logger.warn(e.getMessage(), e);  
                    }  
                }  
            }  
        }  
    }  
  
    private void saveProperties(URL url) {  
        if (file == null) {  
            return;  
        }  
  
        try {  
            StringBuilder buf = new StringBuilder();  
            Map<String, List<URL>> categoryNotified = notified.get(url);  
            if (categoryNotified != null) {  
                for (List<URL> us : categoryNotified.values()) {  
                    for (URL u : us) {  
                        if (buf.length() > 0) {  
                            buf.append(URL_SEPARATOR);  
                        }  
                        buf.append(u.toFullString());  
                    }  
                }  
            }  
        }  
    }  
}
```

JAVA

```
properties.setProperty(url.getServiceKey(), buf.toString());
long version = lastCacheChanged.incrementAndGet();
if (syncSaveFile) {
    doSaveProperties(version);
} else {
    registryCacheExecutor.execute(new SaveProperties(version));
}
} catch (Throwable t) {
    logger.warn(t.getMessage(), t);
}
}

public void doSaveProperties(long version) {
    //只有获得最新版本号才能执行保存操作
    if (version < lastCacheChanged.get()) {
        return;
    }
    if (file == null) {
        return;
    }
    // Save
    try {
        //获取*.lock文件，不存在则新建
        File lockfile = new File(file.getAbsolutePath() + ".lock");
        if (!lockfile.exists()) {
            lockfile.createNewFile();
        }
        //由*.lock文件获取到其持有的锁
        try (RandomAccessFile raf = new RandomAccessFile(lockfile, "rw");
             FileChannel channel = raf.getChannel()) {
            FileLock lock = channel.tryLock();
            if (lock == null) {
                throw new IOException("Can not lock the registry cache file " +
file.getAbsolutePath() + ", ignore and retry later, maybe multi java process use the
file, please config: dubbo.registry.file=xxx.properties");
            }
            // Save
            try {
                if (!file.exists()) {
                    file.createNewFile();
                }

                //使用store()操作保存Properties到文件中
                try (FileOutputStream outputFile = new FileOutputStream(file)) {
                    properties.store(outputFile, "Dubbo Registry Cache");
                }
            } finally {
                //释放文件锁
                lock.release();
            }
        }
    } catch (Throwable e) {
        //如果因锁获取失败等原因导致的异常，对当前操作进行重试处理
        savePropertiesRetryTimes.incrementAndGet();
        if (savePropertiesRetryTimes.get() >= MAX_RETRY_TIMES_SAVE_PROPERTIES) {
            logger.warn("Failed to save registry cache file after retrying " +
MAX_RETRY_TIMES_SAVE_PROPERTIES + " times, cause: " + e.getMessage(), e);
        }
    }
}
```

```

        savePropertiesRetryTimes.set(0);
        return;
    }
    if (version < lastCacheChanged.get()) {
        savePropertiesRetryTimes.set(0);
        return;
    } else {
        registryCacheExecutor.execute(new
SaveProperties(lastCacheChanged.incrementAndGet()));
    }
    logger.warn("Failed to save registry cache file, will retry, cause: " +
e.getMessage(), e);
}
...
}

```

假如已经有一个线程正在执行 `doSaveProperties()` 操作，已经执行到“// Save”这个位置，另外一个线程也试图发起该操作，由于它会获得最新的 `version`，因此它也能继续往下执行，这时就很有可能发生共享资源的争用，接下来使用的文件锁刚好保证了这种互斥性。

Dubbo为了防止ookeeper等的注册中心出现网络抖动情况而导致Consumer订阅操作无法顺利进行需要以文件的方式缓存最新Consumer匹配到的Provider的URL信息，在每次 `notify()` 都会调用 `saveProperties()` 将最新数据保存起来。总所周知，IO是比较费时的，这势必降低效率，因而另外提供一个异步保存这些数据到\*.properties文件的操作。

```

public abstract class AbstractRegistry implements Registry {
    // File cache timing writing
    private final ExecutorService registryCacheExecutor =
Executors.newFixedThreadPool(1, new NamedThreadFactory("DubboSaveRegistryCache",
true));
    // Is it synchronized to save the file
    private final boolean syncSaveFile;

    private class SaveProperties implements Runnable {
        private long version;

        private SaveProperties(long version) {
            this.version = version;
        }

        @Override
        public void run() {
            doSaveProperties(version);
        }
    }
    ...
}

```

JAVA

## 失败重试处理

作为分布式服务的注册模块，其稳定性和容错性的要求会比较苛刻，由于真正负责注册数据处理的是部署在另一台主机的Zookeeper、etcd等网络节点，跨越网络IO的操作的失败概率很高，因此对应动作相应也会有着比较高的概率会失败，作为服务可靠性保障，重试机制的重要性不言而喻。

从上述代码中我们知道，注册中心有 `notify`、`subscribe`、`unsubscribe`、`register`、 `unregister` 5个主要操作，重试也就是针对这几个动作。Dubbo并没有把这部分实现在基类 `AbstractRegistry` 中，做了扩展实现—— `FailbackRegistry`，无论是接入Zookeeper还是诸如 etcd 等其他作为注册中心的分布式协作间，Dubbo都要求能提供失败重试机制。

`FailbackRegistry` 中定义了如下几个待子类实现向其他分布式中间件实现 `subscribe`、`unsubscribe`、`register`、 `unregister` 这4个操作的模板抽象方法：

```
public abstract void doRegister(URL url);
public abstract void doUnregister(URL url);
public abstract void doSubscribe(URL url, NotifyListener listener);
public abstract void doUnsubscribe(URL url, NotifyListener listener);
```

JAVA

参考[定时轮算法 · HashedWheelTimer](#)，在充分理解了定时轮算法后，重试实现的原理其实比较容易理解。

## failedRegistered\failedUnregistered

实现比较简单，仅以 `failedRegistered` 为例：

```

public abstract class FallbackRegistry extends AbstractRegistry {
    ...
    private final ConcurrentHashMap<URL, FailedRegisteredTask> failedRegistered = new
    ConcurrentHashMap<URL, FailedRegisteredTask>();

    public void removeFailedRegisteredTask(URL url) {
        failedRegistered.remove(url);
    }

    private void addFailedRegistered(URL url) {
        FailedRegisteredTask oldOne = failedRegistered.get(url);
        if (oldOne != null) {
            return;
        }
        FailedRegisteredTask newTask = new FailedRegisteredTask(url, this);
        oldOne = failedRegistered.putIfAbsent(url, newTask);
        if (oldOne == null) {
            // never has a retry task. then start a new task for retry.
            retryTimer.newTimeout(newTask, retryPeriod, TimeUnit.MILLISECONDS);
        }
    }

    //在Provider发起或者重试register操作时均会调用
    private void removeFailedRegistered(URL url) {
        FailedRegisteredTask f = failedRegistered.remove(url);
        if (f != null) {
            f.cancel();
        }
    }

    ConcurrentHashMap<URL, FailedRegisteredTask> getFailedRegistered() {
        return failedRegistered;
    }

    @Override
    public void register(URL url) {
        super.register(url);
        removeFailedRegistered(url);
        removeFailedUnregistered(url);
        try {
            // Sending a registration request to the server side
            doRegister(url);
        } catch (Exception e) {
            Throwable t = e;

            // If the startup detection is opened, the Exception is thrown directly.
            boolean check = getUrl().getParameter(Constants.CHECK_KEY, true)
                && url.getParameter(Constants.CHECK_KEY, true)
                && !CONSUMER_PROTOCOL.equals(url.getProtocol());
            boolean skipFallback = t instanceof SkipFallbackWrapperException;
            if (check || skipFallback) {
                if (skipFallback) {
                    t = t.getCause();
                }
                throw new IllegalStateException("Failed to register " + url + " to
registry " + getUrl().getAddress() + ", cause: " + t.getMessage(), t);
            }
        }
    }
}

```

JAVA

```

    } else {
        logger.error("Failed to register " + url + ", waiting for retry, cause:
" + t.getMessage(), t);
    }

    // Record a failed registration request to a failed list, retry regularly
    addFailedRegistered(url);
}
}

public abstract void doRegister(URL url);
...
}

```

从以上代码不难发现，当Provider向Registry发起 `register` 时，如果该操作失败，若未开启启动检测特性，则会给定时轮添加一个重试任务—— `FailedRegisteredTask`，后者在若干时间获得某个滴答运行时机时会重新执行 `doRegister`，以完成到Zookeeper等的注册操作。

`FailedRegisteredTask` 中重试逻辑如下，回调 `registry()`，再将自身这个任务从 `failedRegistered` 这个任务容器中移除：

```
registry.doRegister(url);
registry.removeFailedRegisteredTask(url);
```

JAVA

## failedSubscribed\failedUnsubscribed

这三者的重试逻辑和上述基本相同，在进一步了解前，先看看下述源码。从上文分析得知，*Consumer*订阅事件，是由*Registry*根据其URL参数判别有哪些注册了的Provider 匹配该 *Consumer*的，因而其自身（由URL表征）和其订阅的NotifyListener是强绑定关系，再根据重试需要和相应重试Task形成形如 `<>URL,NotifyListener>, *Task>` 的绑定关系。

```
public abstract class FallbackRegistry extends AbstractRegistry {
```

JAVA

```
    ...

    private final ConcurrentHashMap<Holder, FailedSubscribedTask> failedSubscribed =
        new ConcurrentHashMap<Holder, FailedSubscribedTask>();

    private final ConcurrentHashMap<Holder, FailedUnsubscribedTask> failedUnsubscribed =
        new ConcurrentHashMap<Holder, FailedUnsubscribedTask>();

    private final ConcurrentHashMap<Holder, FailedNotifiedTask> failedNotified =
        new ConcurrentHashMap<Holder, FailedNotifiedTask>();

    static class Holder {

        private final URL url;

        private final NotifyListener notifyListener;

        Holder(URL url, NotifyListener notifyListener) {
            if (url == null || notifyListener == null) {
                throw new IllegalArgumentException();
            }
            this.url = url;
            this.notifyListener = notifyListener;
        }

        @Override
        public int hashCode() {
            return url.hashCode() + notifyListener.hashCode();
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof Holder) {
                Holder h = (Holder) obj;
                return this.url.equals(h.url) &&
this.notifyListener.equals(h.notifyListener);
            } else {
                return false;
            }
        }
    }
}
```

...  
}

Dubbo认为在执行 `subscribe` 操作时，如果发生异常，那么说明负责承担注册中心角色的 Zookeeper等中间件出现了网络抖动，这时会调用 `getCacheUrls(url)` 获取最近缓存的所有匹配当前Consumer关注的所有Providers，有值则会调用 `notify()` 通知该Consumer，否则才发起重试逻辑。如下所示：

```

public abstract class FallbackRegistry extends AbstractRegistry {
    ...
    public void subscribe(URL url, NotifyListener listener) {
        super.subscribe(url, listener);
        removeFailedSubscribed(url, listener);
        try {
            // Sending a subscription request to the server side
            doSubscribe(url, listener);
        } catch (Exception e) {
            Throwable t = e;

            List<URL> urls = getCacheUrls(url);
            if (CollectionUtils.isNotEmpty(urls)) {
                notify(url, listener, urls);
                logger.error("Failed to subscribe " + url + ", Using cached list: " +
                urls + " from cache file: " +
                + getUrl().getParameter(FILE_KEY, System.getProperty("user.home") +
                "/dubbo-registry-" + url.getHost() + ".cache") + ", cause: " + t.getMessage(), t);
            } else {
                // If the startup detection is opened, the Exception is thrown
                directly.
                boolean check = getUrl().getParameter(Constants.CHECK_KEY, true)
                    && url.getParameter(Constants.CHECK_KEY, true);
                boolean skipFallback = t instanceof SkipFallbackWrapperException;
                if (check || skipFallback) {
                    if (skipFallback) {
                        t = t.getCause();
                    }
                    throw new IllegalStateException("Failed to subscribe " + url + ",
                    cause: " + t.getMessage(), t);
                } else {
                    logger.error("Failed to subscribe " + url + ", waiting for retry,
                    cause: " + t.getMessage(), t);
                }
            }
        }

        // Record a failed registration request to a failed list, retry regularly
        addFailedSubscribed(url, listener);
    }
}
...
}

```

## failedNotified

从上文分析可知 `notify` 实际完成的操作是，就对应Consumer所关注的Provider回调 `NotifyListener`事件，而被关注对象列表时动态变化的，因而也 导致对应的 `notify` 操作实现比较特殊，以下是其所有相关源码。

JAVA

```
public abstract class FallbackRegistry extends AbstractRegistry {
```

JAVA

```
...  
    private void removeFailedSubscribed(URL url, NotifyListener listener) {  
        Holder h = new Holder(url, listener);  
        FailedSubscribedTask f = failedSubscribed.remove(h);  
        if (f != null) {  
            f.cancel();  
        }  
        removeFailedUnsubscribed(url, listener);  
        removeFailedNotified(url, listener);  
    }
```

```
    private void removeFailedNotified(URL url, NotifyListener listener) {  
        Holder h = new Holder(url, listener);  
        FailedNotifiedTask f = failedNotified.remove(h);  
        if (f != null) {  
            f.cancel();  
        }  
    }
```

```
    private void addFailedNotified(URL url, NotifyListener listener, List<URL> urls) {  
        Holder h = new Holder(url, listener);  
        FailedNotifiedTask newTask = new FailedNotifiedTask(url, listener);  
        FailedNotifiedTask f = failedNotified.putIfAbsent(h, newTask);  
        if (f == null) {  
            // never has a retry task. then start a new task for retry.  
            newTask.addUrlToRetry(urls);  
            retryTimer.newTimeout(newTask, retryPeriod, TimeUnit.MILLISECONDS);  
        } else {  
            // just add urls which needs retry.  
            newTask.addUrlToRetry(urls);  
        }  
    }
```

```
@Override
```

```
protected void notify(URL url, NotifyListener listener, List<URL> urls) {  
    if (url == null) {  
        throw new IllegalArgumentException("notify url == null");  
    }  
    if (listener == null) {  
        throw new IllegalArgumentException("notify listener == null");  
    }  
    try {  
        doNotify(url, listener, urls);  
    } catch (Exception t) {  
        // Record a failed registration request to a failed list, retry regularly  
        addFailedNotified(url, listener, urls);  
        logger.error("Failed to notify for subscribe " + url + ", waiting for  
retry, cause: " + t.getMessage(), t);  
    }  
}
```

...

```

public final class FailedNotifiedTask extends AbstractRetryTask {

    private static final String NAME = "retry subscribe";

    private final NotifyListener listener;

    private final List<URL> urls = new CopyOnWriteArrayList<>();

    public FailedNotifiedTask(URL url, NotifyListener listener) {
        super(url, null, NAME);
        if (listener == null) {
            throw new IllegalArgumentException();
        }
        this.listener = listener;
    }

    public void addUrlToRetry(List<URL> urls) {
        if (CollectionUtils.isEmpty(urls)) {
            return;
        }
        this.urls.addAll(urls);
    }

    public void removeRetryUrl(List<URL> urls) {
        this.urls.removeAll(urls);
    }

    @Override
    protected void doRetry(URL url, FallbackRegistry registry, Timeout timeout) {
        if (CollectionUtils.isNotEmpty(urls)) {
            listener.notify(urls);
            urls.clear();
        }
        //在下一个周期重试当前Task
        reput(timeout, retryPeriod);
    }
}

```

由其对应的Task定义可以看出，如果该Task没有因为 `subscribe/unsubscribe` 操作而调用 `removeFailedSubscribed` 被移除，那么该Task会一直周期性的运行下去，由 `doRetry()` 的逻辑——只有匹配Provider的urls容器内容不为空的时候，才会回调 `listener.notify()` 执行实际的通知操作，并随后清理该容器，也就是说某个固定 `<URL,NotifyListener>` 绑定所对应的 `FailedNotifiedTask` 被设计成周期重试任务，并且一旦添加 就会长期缓存在内存中，后续的 `notify` 重试，仅仅是将对应的匹配 `Provider` 的 `urls` 加入到其容器中以等待下一个滴答运行时刻。

## recover

最后有个比较特殊的地方是， `recover()` 方法也被覆写了，改为调用 `addFailedRegistered(url)` 和 `addFailedSubscribed(url, listener)`，由定时轮驱动异步完成相应的恢复工作。

完結

# Zookeeper与Dubbo

ZK是“一种为分布式应用所设计的高可用、高性能且一致的开源协调服务”，呈树形目录结构，因支持变更推送，且工业强度较高，是Dubbo微服务框架中首推的一种注册中心实现方案。

除了作为注册中心，ZK还被当做配置中心、元数据中心，而他们的实现都需要依赖于一个称作 `dubbo-remoting-zookeeper` 的ZK客户端模块。

## ZK 注册中心

### 基本特性

ZK注册中心架构示意图如下：

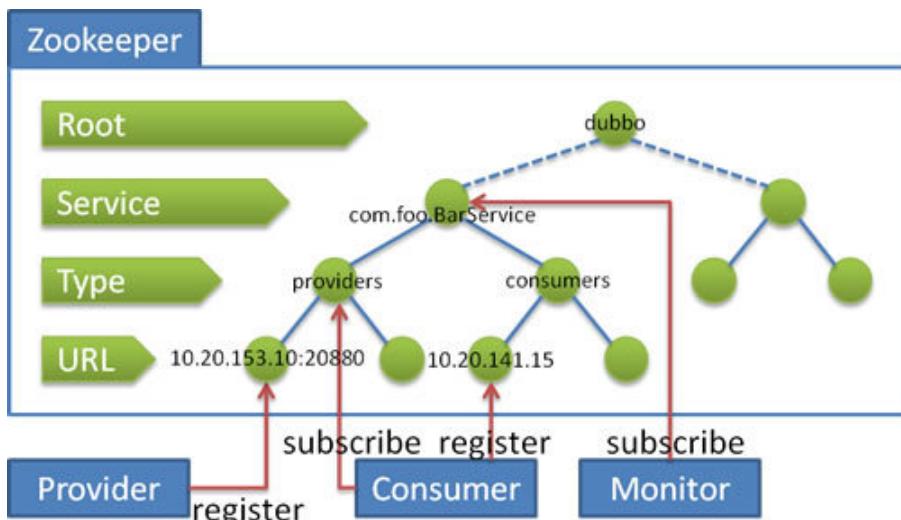


图 1: Dubbo 注册中心Zookeeper示意图

将上述示意图反过来，可以发现由ZK负责管理的Dubbo节点形如一棵树或者一片森林。无根树、多个group注册到同一ZK集群上。树的深度为4，其含义从顶向下分别如下：

1. **Root** 根目录，通过 `<dubbo:registry group="dubbo" />` 的`group`来设置ZK的根节点，缺省值是`dubbo`。
2. **Service** 服务接口全名，如 `com.foo.BarService`。
3. **Type** 参与构成Dubbo服务的有4大件，他们扮演着不同的行为，可以被视为机器节点 Node，都有着自身的URL，分别是`Provider`服务提供者、`Consumer`服务消费者、`Route`路由规则、`Configuration`覆写规则。每一个 Type 节点下对应一个列表，维护相应类型的参与URL 节点。
4. **URL** 对应Type的URL。

**NOTE**

叶节点对应着具体的能用URL表达的Dubbo的Node节点，树的非叶节点则对应着某种分类，从根节点到叶节点以"/"拼接形成的Path唯一确定了树中的某个节点。

**Path**示例及ZG对应源码中的方法：

1. `/(dubbo | regUrl["group"]) toRootPath()`
2. `[toRootPath()]/ toRootDir()`
3. `toRootPath() [/(url["interface"] | url.path)] toServicePath(url)`
4. `toServicePath(url) / (providers | consumers | routes | configurations) toCategoryPath(url)、  
toCategoriesPath(url)`
5. `toCategoryPath(url) / URL.encode(url.toFullString()) toUrlPath(url)`

**NOTE**

URL数据没有特别指明参数 `url["category"]` 时，会直接使用默认类别 `providers`，参数值是由","号分割的字符串。

另外上述示意图还表述了如下对应组件启动时的流程：

1. **Provider:** 向 `/dubbo/com.foo.BarService/providers` 目录下写入自己的 URL 地址
2. **Consumer:** 订阅 `/dubbo/com.foo.BarService/providers` 目录下的提供者 URL 地址。并向 `/dubbo/com.foo.BarService/consumers` 目录下写入自己的 URL 地址
3. **Monitor:** 订阅 `/dubbo/com.foo.BarService` 目录下的所有提供者和消费者 URL 地址。

支持以下功能：

1. 当提供者出现断电等异常停机时，注册中心能自动删除提供者信息
2. 当注册中心重启时，能自动恢复注册数据，以及订阅请求
3. 当会话过期时，能自动恢复注册数据，以及订阅请求
4. 当设置 `<dubbo:registry check="false" />` 时，记录失败注册和订阅请求，后台定时重试
5. 可通过 `<dubbo:registry username="admin" password="1234" />` 设置 ZK 登录信息
6. 可通过 `<dubbo:registry group="dubbo" />` 设置 ZK 的根节点，不设置将使用无根树
7. 支持 \* 号通配符 `<dubbo:reference group="" version="" />`，可订阅服务的所有分组和所有版本的提供者

## 具体实现

Dubbo中ZK注册中心方案由*ZookeeperRegistry*实现，代码比较简单，其继承关系为：

```
ZookeeperRegistry ← FallbackRegistry() ← AbstractRegistry(Registry) ← (Node, RegistryService)
```

具体如下图所示：

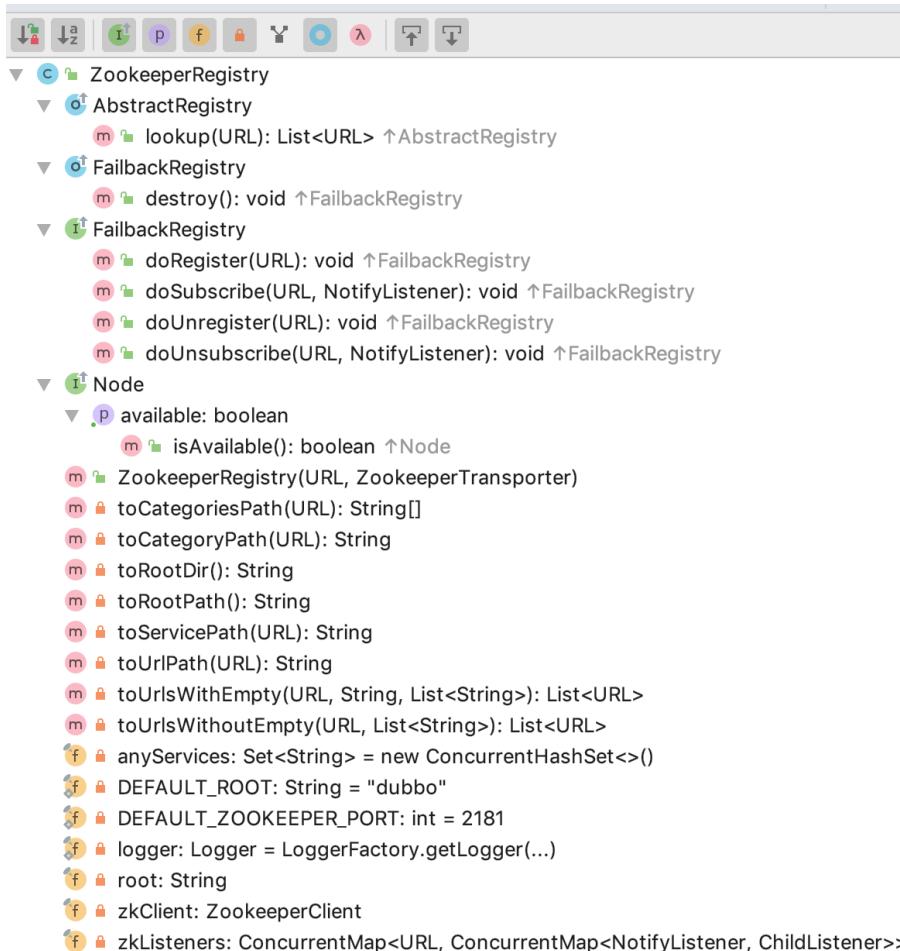


图 2: *ZookeeperRegistry*代码结构图

## 生命周期行为

*ZookeeperRegistry*初始化时会和ZK建立连接，获得*ZookeeperClient*这个线程安全的ZK客户端调用工具，并增加对应的状态监听器，确保ZK重启发生重连行为时及时 `recover()`——恢复注册数据和订阅请求； `isConnected()` 用于检测当前ZG这个Node是否可用； `close()` 用于关ZK连接。

```

public ZookeeperRegistry(URL url, ZookeeperTransporter zookeeperTransporter){
    super(url);
    ...
    zkClient.addStateListener(state -> {
        if (state == StateListener.RECONNECTED) {
            try {
                //没有开启check模式时，该行为会借助定时轮引擎反复重试
                recover();
            } catch (Exception e) {
                logger.error(e.getMessage(), e);
            }
        }
    });
}
@Override
public boolean isAvailable() {
    return zkClient.isConnected();
}

@Override
public void destroy() {
    super.destroy();
    try {
        zkClient.close();
    } catch (Exception e) {
        logger.warn("Failed to close zookeeper client " + getUrl() + ", cause: " +
e.getMessage(), e);
    }
}

```

## 查找匹配的Node

ZG并没有沿用其超类 AbstractRegistry 的lookup行为，直接覆写，由给定URL解析得到其对应的一到多个**Type**节点，汇总他们的子节点**叶节点**，也即获得一个 `List<URL>`，再根据这个list逐个与url进行匹配，最终获得所有匹配的URL节点。

JAVA

```

@Override
public List<URL> lookup(URL url) {
    if (url == null) {
        throw new IllegalArgumentException("lookup url == null");
    }
    try {
        List<String> providers = new ArrayList<>();
        for (String path : toCategoriesPath(url)) {
            List<String> children = zkClient.getChildren(path);
            if (children != null) {
                providers.addAll(children);
            }
        }
        return toUrlsWithoutEmpty(url, providers);
    } catch (Throwable e) {
        throw new RpcException("Failed to lookup " + url + " from zookeeper " +
getUrl() + ", cause: " + e.getMessage(), e);
    }
}

private List<URL> toUrlsWithoutEmpty(URL consumer, List<String> providers) {
    List<URL> urls = new ArrayList<>();
    if (CollectionUtils.isNotEmpty(providers)) {
        for (String provider : providers) {
            provider = URL.decode(provider);
            if (provider.contains(Constants.PROTOCOL_SEPARATOR)) {
                URL url = URL.valueOf(provider);
                if (UrlUtils.isMatch(consumer, url)) {
                    urls.add(url);
                }
            }
        }
    }
    return urls;
}

```

## 注册与订阅

从《【二】Dubbo注册中心》可知，ZK注册中心需要实现如下几个抽象模板方法，下述将分别就他们加以阐述。

JAVA

```

void doRegister(URL url);

void doUnregister(URL url);

void doSubscribe(URL url, NotifyListener listener);

void doUnsubscribe(URL url, NotifyListener listener);

```

doRegister(URL url)、doUnregister(URL url)

注册和取消注册均是在ZK中调用ZookeeperClient对应的方法找到合适的节点位置，在其之下创建或删除一个叶节点，实现比较简单。在创建节点时使用了 DYNAMIC\_KEY 获取参数，确定创建节点是动态的还是固态的，对应ZK中的临时节点或持久节点。

## IMPORTANT

zookeeper的持久节点是节点创建后，就一直存在，直到有删除操作来主动清除这个节点，不会因为创建该节点的客户端会话失效而消失。而临时节点的生命周期和客户端会话绑定。也就是说，如果客户端会话失效，那么这个节点就会自动被清除掉。注意，这里提到的是会话失效，而非连接断开。另外，在临时节点下面不能创建子节点。

注册中心创建节点时用的都是临时节点，目的正是利用了会话断开节点被自动移除从而引发对应节点变动事件，触发接入注册中心所有其他 Dubbo 服务节点第一时间感知已有节点已经离线，及时做相应响应处理。

```
@Override  
public void doRegister(URL url) {  
    try {  
        zkClient.create(toUrlPath(url), url.getParameter(Constants.DYNAMIC_KEY, true));  
    } catch (Throwable e) {  
        throw new RpcException("Failed to register " + url + " to zookeeper " +  
getURL() + ", cause: " + e.getMessage(), e);  
    }  
}  
@Override  
public void doUnregister(URL url) {  
    try {  
        zkClient.delete(toUrlPath(url));  
    } catch (Throwable e) {  
        throw new RpcException("Failed to unregister " + url + " to zookeeper " +  
getURL() + ", cause: " + e.getMessage(), e);  
    }  
}  
  
doSubscribe(URL url, NotifyListener listener)
```

想进一步理解其原理，我们先看看下述关于subscribe的定义：

```
/**
 * 订阅符合条件的已注册数据，当有注册数据变更时自动推送。
 *
 * 订阅需处理契约：<br>
 * 1. 当URL设置了check=false时，订阅失败后不报错，在后台定时重试。<br>
 * 2. 当URL设置了category=routers，只通知指定分类的数据，多个分类用逗号分隔，并允许星号通配，表示订阅所有分类数据。<br>
 * 3. 允许以interface,group,version,classifier作为条件查询，如：
interface=com.alibaba.foo.BarService&version=1.0.0<br>
 * 4. 并且查询条件允许星号通配，订阅所有接口的所有分组的所有版本，或：
interface=*&group=*&version=*&classifier=*<br>
 * 5. 当注册中心重启，网络抖动，需自动恢复订阅请求。<br>
 * 6. 允许URI相同但参数不同的URL并存，不能覆盖。<br>
 * 7. 必须阻塞订阅过程，等第一次通知完后再返回。<br>
 *
 * @param url 订阅条件，不允许为空，如：consumer://10.20.153.10/com.alibaba.foo.BarService?
version=1.0.0&application=kylin
 * @param listener 变更事件监听器，不允许为空
 */

```

也就是说订阅的目的是为了让被关注的节点在有变化时能及时通知，能感知到被关注者的状态变化，是服务可靠性的一个重要保障。

下述CacheListener章节已经深入的阐明了这种设计模式，一对一绑定的监听接口由下往上，下一层的驱动着上一层的执行，这样会使得上下层间的关系是松耦合的，底层不用管上层的具体实现，增强了可扩展性。ChildListener 是 ZK 客户端实现，而 NotifyListener 则是注册中心 ZK 版的实现，前者是为后者提供服务的。换言之是 ZookeeperRegistry 将框架上层调用方对其添加 NotifyListener 监听器转换成了它对 ZK 客户端添加 ChildListener 监听器，因而两个监听器是一对一的强绑定关系，一个 ChildListener 对象只从属于另一个 NotifyListener 对象。另外客户端可以就同一个URL数据所表示的注册中心节点添加多个 ChildListener 监听器，因此ZG声明了如下一个 2 级键值对的容器。

//Map中，Key的作用是索引，也即根据Key能找到对应的Value，二者建立关联的目的是在已知Key的情况下，能够找到相应的Value，以便其它业务操作

**ConcurrentMap<URL, ConcurrentMap<NotifyListener, ChildListener>> zkListeners**

```
public interface NotifyListener {
    void notify(List<URL> urls);
}

//ZK中的节点呈树形结构，除叶节点外，都拥有一到多个子节点，当子节点的数量或者数据发生变化时
//会触发该回调事件，内容分别是当前节点和子节点所在的"Path部分"
public interface ChildListener {
    void childChanged(String path, List<String> children);
}
```

相关代码比较复杂，涉及多级嵌套Map，类似这种容器在Dubbo等中间件中经常出现，其内容填充基本套路如下：

JAVA

```
{  
    Map<Key2, Value> innerMp = mp.get(Key1_1);  
  
    if (null == innerMp) {  
        mp.putIfAbsent(Key1_1, new HashMap<>());  
        innerMp = mp.get(Key1_1);  
    }  
  
    Value v1 = innerMp.get(Key1_2);  
  
    if (null == v1) {  
        innerMp.putIfAbsent(Key1_2, buildValueForV1());  
        v1 = innerMp.get(Key1_2);  
    }  
}
```

## NOTE

上述假定Map<Key1,Map<Key2,Value>> mp，具体处理时已知入参 Key1\_1、Key2\_1、Value值用buildValueForV1()得到

上文中提到，有两种类型的订阅，Consumer订阅的是某一个或者多个Type节点下的URL子节点，而Monitor订阅的是特定Service下的所有URL节点，在实现上后者是通过前者达成的。代码比较长，依然拆分成更小片段加以深入剖析。

### 监听特定 service

对应业务代码如下，实际为方便理解可以先上将源码中的 [TAG\_x:start,TAG\_x:end] 部分可以挪动到 for 循环之前，也即 TAG\_pos 所标识的位置，其基本步骤为：

1. 先使用嵌套 map 处理所述方式将 NotifyListener 类型的入参 listener 映射成 ChildListener 实例 zkListener，zkListener 的目的是在 ZK 目录节点 parentPath 的子节点发生变化时，调用 notify(...) 将变化后的当前所剩子节点全貌 currentChilds 作为参数通过回调 listener 告诉调用方；
2. 使用 toCategoriesPath(url) 取得当前客户端所订阅 ZK 目录节点下的所有子节点，也即被订阅节点下的 Type 型节点的全路径；
3. 挨个遍历各 Type 节点路径 path：

- a. 使用 ZK 客户端 zkClient 在 ZK 注册中心创建一个非临时节点，随即将 zkListener 监听器添加到该节点上，此后所有该 Type 节点下的子节点发生变化，当前客户端都会通过 listener 回调感知到；
  - b. 将上一步给 path 所在节点添加监听器时返回的所有叶节点数据经 toUrlsWithEmpty(...) 过滤处理后转换获得的 URL 数据列表收集到 urls 容器中；
4. 调用 notify(...) 方法，将订阅的所有叶节点数据 urls 通过回调 listener 监听器通知调用方，这样做的目的是客户端在提交订阅请求后就能立马可以异步获得所有感兴趣的页节点数据，比如客户端在引用服务后的一瞬间，便开始拥有完备的分布式网络服务信息，从而对目标微服务发起调用；

```
public void doSubscribe(final URL url, final NotifyListener listener) {  
    ...  
    List<URL> urls = new ArrayList<>();  
    //TAG_pos  
    //获取URL匹配的所有Type (providers|consumers|routes|configurations中一种)  
    for (String path : toCategoriesPath(url)) {  
        //TAG_x:start  
        ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);  
        if (listeners == null) {  
            zkListeners.putIfAbsent(url, new ConcurrentHashMap<>());  
            listeners = zkListeners.get(url);  
        }  
        ChildListener zkListener = listeners.get(listener);  
        if (zkListener == null) {  
            //Type节点下若有新的节点加入，则调用notify回调NotifyListener  
            listeners.putIfAbsent(listener,  
                (parentPath, currentChilds) ->  
                    ZookeeperRegistry.this.notify(url, listener,  
                        toUrlsWithEmpty(url, parentPath, currentChilds))  
            );  
            zkListener = listeners.get(listener);  
        }  
        //TAG_x:end  
        zkClient.create(path, false);  
        List<String> children = zkClient.addChildListener(path, zkListener);  
        if (children != null) {  
            urls.addAll(toUrlsWithEmpty(url, path, children));  
        }  
    }  
  
    notify(url, listener, urls);  
    ...  
}
```

源码中多处调用的 toUrlsWithEmpty(...) 如下，其目的是当 ZK 客户端回调框架上层的 NotifyListener 监听器时，对注册中心返回的 Type 类节点下的所有子节点做进一步过滤处理，这个复杂的过滤业务逻辑是没法由注册中心本身完成的。我们都清楚尽管一个微服务的所有实例只会注册到和 toServicePath(url) 对应的节点上，但是客户端订阅可以指明更加细致的范围，可以指定 category、enabled、group、version、classifier 这几个中的一到多个参数做范围筛选。

另外Dubbo的 ZK 客户端将一个微服务的实例注册到注册中心时，表示这个实例的节点是一个URL数据的字符串完整表示的叶节点，因此在检查匹配情况时所调用的 `isMatch(consumer, url)` 需要先将字符串转换为URL实例。

然而，很有可能经过当前客户端指定参数过滤后，压根没有匹配到任何叶节点，这时 `ZookeeperRegistry` 应该明确告知其调用方——返回只有一个设 `url.protocol = "empty"` 的 URL实例的列表，[《Dubbo集群之目录服务》](#) 一文的服务引用实例刷新流程中就有关于利用这一机制的说明，即针对目标微服务的服务发现功能被停用。

```
JAVA
private List<URL> toUrlsWithEmpty(URL consumer, String path, List<String> providers) {
    List<URL> urls = toUrlsWithoutEmpty(consumer, providers);
    if (urls == null || urls.isEmpty()) {
        int i = path.lastIndexOf(PATH_SEPARATOR);
        String category = i < 0 ? path : path.substring(i + 1);
        // 使用入参consumer衍生一个新的URL实例, url.protocol = "empty", url["category"] =
        {category}
        URL empty = URLBuilder.from(consumer)
            .setProtocol(EMPTY_PROTOCOL)
            .addParameter(CATEGORY_KEY, category)
            .build();
        urls.add(empty);
    }
    return urls;
}

private List<URL> toUrlsWithoutEmpty(URL consumer, List<String> providers) {
    List<URL> urls = new ArrayList<>();
    if (CollectionUtils.isNotEmpty(providers)) {
        for (String provider : providers) {
            provider = URL.decode(provider);
            if (provider.contains(PROTOCOL_SEPARATOR)) {
                URL url = URL.valueOf(provider);
                if (UrlUtils.isMatch(consumer, url)) {
                    urls.add(url);
                }
            }
        }
    }
    return urls;
}
```

## 监听所有 service

有了上一小章节的源码剖析，再来理解本章节中的相关源码就容易很多了。

如果当前客户端指定需要监听任意的 service，也即 `url["interface"] = "*"`，实际上具体实现上是针对 `toRootPath() root` 节点下的所有 service 都添加上同一个 `NotifyListener` 监听器 `listener`，这是通过 `subscribe(...)` 递归调用完成的，也即。

如下具体源码，针对 `root` 节点添加的 `ChildListener` 监听器中，其源码的 `for` 循环的意思是如果一个子节点在 `anyServices` 容器中不曾出现过，比如新上线了一个微服务，则就该 `service` 启用订阅操作。

同样，为 `root` 节点增加监听器的过程中会返回其所有的子节点，此时为这些子节点增加监听器的时机最恰当，前面基于事件回调的 `subscribe(...)` 只有在有新的 `service` 节点加入或者被删除一个 `service` 节点时才会被执行，再者因为它是异步的，这样势必造成延迟。

```

//用于确保不针对同一个节点经由subscribe迭代doSubscribe操作
private final Set<String> anyServices = new ConcurrentHashSet<>();

@Override
public void doSubscribe(final URL url, final NotifyListener listener) {
    ...
    String root = toRootPath();
    ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
    if (listeners == null) {
        zkListeners.putIfAbsent(url, new ConcurrentHashMap<>());
        listeners = zkListeners.get(url);
    }
    ChildListener zkListener = listeners.get(listener);
    if (zkListener == null) {
        //监听Root之下的Service节点们
        listeners.putIfAbsent(listener, (parentPath, currentChilds) -> {
            for (String child : currentChilds) {
                child = URL.decode(child);
                //防止重复执行doSubscribe操作
                if (!anyServices.contains(child)) {
                    anyServices.add(child);
                    //设置ServiceInterface参数, 递归doSubscribe操作, 转入下一个主分支
                    subscribe(url.setPath(child).addParameters(Constants.INTERFACE_KEY,
child,
                        Constants.CHECK_KEY, String.valueOf(false)), listener);
                }
            }
        });
        zkListener = listeners.get(listener);
    }
    //创建Root根节点, 据具体实现, 在对应节点存在时则直接略过
    zkClient.create(root, false);
    //将新增加的ChildListener加入到ZK中, 以监听节点变化, 增加监听器时会返回现存的子节点们(若在树中存在对应Path的节点)
    List<String> services = zkClient.addChildListener(root, zkListener);
    if (CollectionUtils.isNotEmpty(services)) {
        for (String service : services) {
            service = URL.decode(service);
            anyServices.add(service);
            subscribe(url.setPath(service).addParameters(Constants.INTERFACE_KEY,
service,
                Constants.CHECK_KEY, String.valueOf(false)), listener);
        }
    }
    ...
}

```

监听所有 `service` 节点，会由客户端传入的 url，为每一个 `service` 节点衍生一份对应的URL数据，由设 `url.path = {service}`、`url["interface"] = {service}`、`url["check"] = false` 得到，如下：

## NOTE

JAVA

```
url.setPath(service).addParameters(  
    ConstantsINTERFACE_KEY, service,  
    ConstantsCHECK_KEY, String.valueOf(false)  
) ;
```

## 总体流程

`doSubscribe(...)` 方法的总体轮廓如下，不难发现 ① 处的代码块最终会递归调用 ② 处的代码块。

```
public void doSubscribe(final URL url, final NotifyListener listener) {  
    try {  
        if (ANY_VALUE.equals(url.getServiceInterface())) {  
            ...//①: 监听所有``service``  
        } else {  
            ...//②: 监听特定``service``  
        }  
    } catch (Throwable e) {  
        throw new RpcException("Failed to subscribe " + url  
            + " to zookeeper " + getUrl() + ", cause: " + e.getMessage(), e);  
    }  
}
```

汇总所有信息，总体而言就是：

1. 当Consumer调用 `subscribe()` 执行订阅操作时，Dubbo会从URL参数判别出调用端所关注 Service和Type类型，由其定位到Type这一级别的节点，然后增加ZK子节点监听器—— `ChildListener`，确保相关的URL节点在加入或者撤出ZK注册中心时，能够及时回调 `NotifyListener` 告知订阅者Consumer节点状态变化。
2. Root节点上子节点的变化意味着有相应的Service加入或撤出，于Monitor这种需要监听全部节点变化的订阅者，会直接在Root节点上创建一个 `ChildListener` 监听器，于每一个新加入的 Service节点，Dubbo会为其产生URL `doSubscribe(...)` 的参数的一个副本，指定其为 `INTERFACE_KEY` 的值，通过 `subscribe(...)` 调用 递归调用 `doSubscribe(...)`，切换到和 Consumer调用 `subscribe(...)` 一样的订阅操作。值得注意是，默认情况下只会订阅 `Type = Provider` 类型的 URL节点，除非在url中设置了 `url["category"]` 这个参数。

## IMPORTANT

源码中多次调用了 `zkClient.create(somePath, false)`，这个 `create` 操作只是为了确保 ZK 注册中心中存在对应 path，也即如果其它接入到 ZK 的微服务实例不曾创建对应 path 时，当前客户端便新创建该 path，否则相当于啥也没干。

另外随后调用的 `addChildListener(...)` 在执行完监听器添加的过程后会返回对应节点上的所有子节点，相当于主动执行了一次数据的拉取处理，而变化的子节点部分则通过结合类如 `Set`、`Map` 这样的能够用于排重的缓存集合在回调事件中获知。

### `doUnsubscribe(URL url, NotifyListener listener)`

实际上就是将 `doSubscribe(...)` 方法添加的监听器给移除掉，总体而言比较简单，如下述源码所示：

```
//该方法比较简单，实际上就是根据unsubscribe操作，能够找到对应的ZK节点，移除其对应的ChildListener监听器
@Override
public void doUnsubscribe(URL url, NotifyListener listener) {
    ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
    if (listeners != null) {
        ChildListener zkListener = listeners.get(listener);
        if (zkListener != null) {
            if (Constants.ANY_VALUE.equals(url.getServiceInterface())) {
                String root = toRootPath();
                zkClient.removeChildListener(root, zkListener);
            } else {
                for (String path : toCategoriesPath(url)) {
                    zkClient.removeChildListener(path, zkListener);
                }
            }
        }
    }
}
```

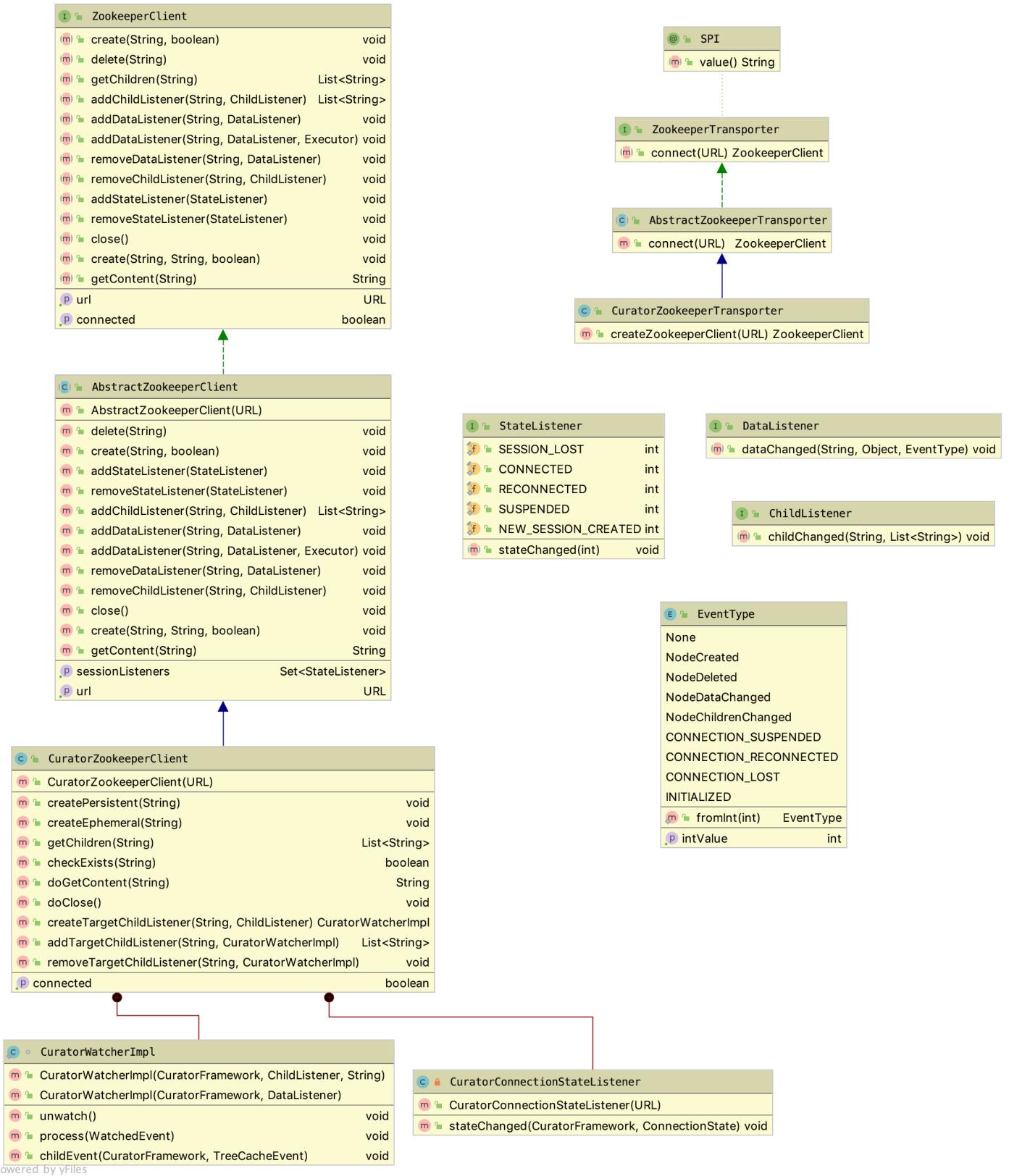
## NOTE

`toCategoryPath(url)` 会获取到形如 `/dubbo/com.foo.BarService/providers` 的 Path，调用 `ZookeeperClient.create()` 后，ZK会在树中创建对应的分支，如：`dubbo ← com.foo.BarService ← providers`。

## ZK 客户端

文初已提及，Zookeeper作为分布式应用中的协调服务服务，可以用作实现Dubbo的注册中心、配置中心、元数据中心。然而就如文章开头部分Dubbo注册中心Zookeeper示意图所展示的那样，ZK只是负责了数据在树形结构中的存取，以其几点发生变化的通知处理而已，依赖于其他的功能实现都是在客户端实现的，也就是说行为和数据是分离的，就如同单相思一般，对方在远端亭亭玉立，屌丝在此岸尽情意淫。

因此Dubbo要利用ZK实现特定功能的话，需要抽离一个接口层，用于桥接远端的ZK服务和本地上层应用的调用，也即兼顾同远端ZK的通讯和为上面应用层提供访问入口，根据Dubbo的基于领域驱动的设计特点，前者被称为 `Transporter`，后者被叫做 `Client`。对应源码中的实现，有两个最关键的接口 `ZookeeperTransporter` 和 `ZookeeperClient`。



如上类图所示，右上角部分的 `ZookeeperTransporter` 只定义了一个方法，目的是和ZK服务端建立连接，在此基础上再创建 `ZookeeperClient` 实例供上层访问，而两个接口最终都是依赖于 `apache-curator` 这个库ZK客户端的社区实现来实现的。

## ZookeeperTransporter

该接口被声明为了一个扩展点，意即开发者可以做自定义实现，其定义如下，Dubbo会为其生成一个代理类，最终的调用会被委托与 `url.getParameter("client", url.getParameter("transporter", "curator"))` 所指定名称的对应扩展具类的实例上，如果配置项能取到值，“client”和“transporter”中则优先使用前者，二者没有的情况下使用默认值“curator”。

```
@SPI("curator")
public interface ZookeeperTransporter {

    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})
    ZookeeperClient connect(URL url);

}
```

JAVA

## AbstractZookeeperTransporter & CuratorZookeeperTransporter

如下所示，ZK客户端模块中的 `Transporter` 的实现，只是简单的指定其 `createZookeeperClient(url)` 方法返回一个 `CuratorZookeeperClient` 类型的 `ZookeeperClient` 对象，也就是说Dubbo实际上是将 `AbstractZookeeperTransporter` 当做了抽象工厂，而 `CuratorZookeeperTransporter` 是其工厂实现。

```
public class CuratorZookeeperTransporter extends AbstractZookeeperTransporter {
    @Override
    public ZookeeperClient createZookeeperClient(URL url) {
        return new CuratorZookeeperClient(url);
    }
}
```

JAVA

上述已经论述过ZK客户端模块中为啥有 `Transporter` 的存在，然而实现上是与之相悖的，只是简单当做了创建工厂，作为缓存用于存取 `ZookeeperClient` 对象，因为后者同时实现访问远端和为上层提供客户端访问入口。

在配置注册中心时，Dubbo允许为其设置冷备，也即当一个ZK服务实例变得不可用时，切换到另外一个ZK服务实例，利用冗余增强系统的可用性，如 `"zookeeper://127.0.0.1:22379/org.apache.dubbo.registry.RegistryService?backup=127.0.0.1:2379,127.0.0.1:32379"`，这例子中同一份 `ZookeeperClient` 对象会享有3套地址，分别是 `127.0.0.1:22379`、`127.0.0.1:2379`、`127.0.0.1:32379`。同一时刻这几套地址指向同一个可用的ZK服务实例，如果该实例不可用会切换到另外一个实例。

```

private final Map<String, ZookeeperClient> zookeeperClientMap = new ConcurrentHashMap<>();
();

List<String> getURLBackupAddress(URL url) {
    List<String> addressList = new ArrayList<String>();
    addressList.add(url.getAddress());

    addressList.addAll(url.getParameter(RemotingConstants.BACKUP_KEY,
    Collections.EMPTY_LIST));
    return addressList;
}

void writeToClientMap(List<String> addressList, ZookeeperClient zookeeperClient) {
    for (String address : addressList) {
        zookeeperClientMap.put(address, zookeeperClient);
    }
}

```

`fetchAndUpdateZookeeperClientCache(addressList)` 方法总是返回第一个可用的实例（在当前客户端由获取到的实例状态），并将 `addressList` 列表中的所有注册中心的IP地址均映射到该实例上，也就是说在可用为前提的条件下始终按位置优先取实例。而如果没有取用到一个可用的实例，则返回 `null`。

一般而言表征注册中心的URL数据中，其 `host` 代表是主机，在列表的首位，而 `backup` 参数中都顺序放在列表的后面。

```

ZookeeperClient fetchAndUpdateZookeeperClientCache(List<String> addressList) { JAVA
    ZookeeperClient zookeeperClient = null;
    for (String address : addressList) {
        if ((zookeeperClient = zookeeperClientMap.get(address)) != null &&
zookeeperClient.isConnected()) {
            break;
        }
    }
    if (zookeeperClient != null && zookeeperClient.isConnected()) {
        writeToClientMap(addressList, zookeeperClient);
    }
    return zookeeperClient;
}

```

在给定注册中心的URL数据，使用 `connect(url)` 方法获取 `ZookeeperClient` 实例时，会首先尝试先从本机缓存中获取，如果缓存中没有可用实例，则会在双检锁机制保证线程安全的情况下再调用 `createZookeeperClient(url)` 建立连接，随后把该连接对应的实例映射给所有主备注册中心的IP地址。

```

public abstract class AbstractZookeeperTransporter implements ZookeeperTransporter { JAVA

    @Override
    public ZookeeperClient connect(URL url) {
        ZookeeperClient zookeeperClient;
        List<String> addressList = getURLBackupAddress(url);
        if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null
&& zookeeperClient.isConnected()) {
            logger.info("find valid zookeeper client from the cache for address: " +
url);
            return zookeeperClient;
        }
        synchronized (zookeeperClientMap) {
            if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null && zookeeperClient.isConnected()) {
                logger.info("find valid zookeeper client from the cache for address: " +
+ url);
                return zookeeperClient;
            }

            zookeeperClient = createZookeeperClient(url);
            logger.info("No valid zookeeper client found from cache, therefore create a
new client for url. " + url);
            writeToClientMap(addressList, zookeeperClient);
        }
        return zookeeperClient;
    }

    protected abstract ZookeeperClient createZookeeperClient(URL url);
}

```

## ZookeeperClient

本章节的类图已经显示，`ZookeeperClient` 定义了大量的方法，均用于和ZK打交道。ZK中的节点以层式树形组织，其节点的路径标识方式和 Unix 文件系统类似，使用斜杆加以分割，客户端可以将其当做一个目录系统看待，可以：① 增删节点；② 针对节点存取数据；③ 获取子节点；④ 增删数据、增删子节点的事件监听器的增删处理；⑤ 增删用于同步ZK实例的状态的监听器；⑥ ZK实例基本状态管理。

```

public interface ZookeeperClient {
    //①
    void create(String path, boolean ephemeral);

    void delete(String path);

    //②
    void create(String path, String content, boolean ephemeral);

    String getContent(String path);

    //③
    List<String> getChildren(String path);

    //④
    List<String> addChildListener(String path, ChildListener listener);

    /**
     * @param path: directory. All of child of path will be listened.
     * @param listener
     */
    void addDataListener(String path, DataListener listener);

    /**
     * @param path: directory. All of child of path will be listened.
     * @param listener
     * @param executor another thread
     */
    void addDataListener(String path, DataListener listener, Executor executor);

    void removeDataListener(String path, DataListener listener);

    void removeChildListener(String path, ChildListener listener);

    //⑤
    void addStateListener(StateListener listener);

    void removeStateListener(StateListener listener);

    //⑥
    boolean isConnected();

    void close();

    URL getUrl();
}

```

## AbstractZookeeperClient & CuratorZookeeperClient

Dubbo在实现 ZK 客户端的时候，刻意设计成可扩展的。针对 ZookeeperClient 的实现也是这样，使用模板设计模式把一些不需要和远端 ZK 服务实例交互的逻辑沉淀到基类 AbstractZookeeperClient 中。

## CuratorZookeeperClient 实例化

JAVA

`ZookeeperClient` 实例的创建总是发生在当前客户端检查到没有可用实例时，根据 `CuratorZookeeperTransporter` 的源码实现，实例创建既是建立连接的过程，而这是其它一切和远端ZK发生通讯行为的前提，因此接下来我们先首先看看基于 `Curator` 客户端工具的 `CuratorZookeeperClient` 实例化过程。

如下述源码所示，`CuratorZookeeperClient` 实际上是将行为委托给了 `CuratorFramework` 这个由 `Curator` 提供的API接口：

1. 建立连接之前需要为这个API准备如下环境参数：
  - 连接超时，单位毫秒，`url["timeout"]`，默认 5s；
  - 会话超时，单位毫秒，`url["zk.session.expire"]`，默认 1m；
  - 重试策略，1次，间隔时间 1s；
  - 连接字符串，`url[host] + ":" + url[port] + [", " + url[backup]]`；
  - 认证口令，`[url.username] + ":" + [url.password]`；
2. 准备好参数后，调用 `CuratorFrameworkFactory.Builder` 的 `build()` 方法获得 `CuratorFramework` 实例 `client`；
3. 为 `client` 增加服务实例的状态监听器；
4. 为 `client` 调用其 `start()` 方法，正式启用该对象；
5. 为 `client` 调用其 `blockUntilConnected(timeout, TimeUnit.MILLISECONDS)` 方法完成最后的连接建立操作；
6. 连接建立失败或者超时都会抛出异常处理。

```
public class CuratorZookeeperClient extends AbstractZookeeperClient<CuratorZookeeperClient.CuratorWatcherImpl, CuratorZookeeperClient.CuratorWatcherImpl> {
```

```
    private static final String ZK_SESSION_EXPIRE_KEY = "zk.session.expire";  
  
    private final CuratorFramework client;  
  
    public CuratorZookeeperClient(URL url) {  
        super(url);  
        try {  
            int timeout = url.getParameter(TIMEOUT_KEY, DEFAULT_CONNECTION_TIMEOUT_MS);  
            int sessionExpireMs = url.getParameter(ZK_SESSION_EXPIRE_KEY,  
                DEFAULT_SESSION_TIMEOUT_MS);  
            CuratorFrameworkFactory.Builder builder = CuratorFrameworkFactory.builder()  
                .connectString(url.getBackupAddress())  
                .retryPolicy(new RetryNTimes(1, 1000))  
                .connectionTimeoutMs(timeout)  
                .sessionTimeoutMs(sessionExpireMs);  
  
            String authority = url.getAuthority();  
            if (authority != null && authority.length() > 0) {  
                builder = builder.authorization("digest", authority.getBytes());  
            }  
            client = builder.build();  
            client.getConnectionStateListenable().addListener(  
                new CuratorConnectionStateListener(url));  
            client.start();  
            boolean connected = client.blockUntilConnected(timeout,  
                TimeUnit.MILLISECONDS);  
            if (!connected) {  
                throw new IllegalStateException("zookeeper not connected");  
            }  
        } catch (Exception e) {  
            throw new IllegalStateException(e.getMessage(), e);  
        }  
    }  
    ...  
}
```

源码中关于状态监听的实现将在下述相关章节中剖析。

## 增删节点

ZK中的节点增删需要有完整的从根节点到叶节点的 path 路径，为避免不必要的网络I/O，`AbstractZookeeperClient` 针对需要持久化的节点，会在其创建过程中使用 `ConcurrentHashSet` 容器记录起来。基于该 Set 容器的节点创建持久节点的过程如下：

1. 点创建前先检查容器中是否存在，存在直接返回；
2. 不存在的话会先调用 `checkExists(path)` 方法检验远端ZK服务实例中是否含有该节点：
  - 含有则加入容器，并直接返回；

JAVA

- 不含有，转入下一步；

### 3. 使用 path 创建节点，随后将其加入到容器中；

这个过程中的第 3 步骤之前实际还隐藏了另外一个步骤，就是创建过程中会以递归的方式先创建最后一个 "/" 之前的 path 部分，实现代码相当简洁。

```
public abstract class AbstractZookeeperClient<TargetDataListener, TargetChildListener>JAVA
    implements ZookeeperClient {

    private final Set<String> persistentExistNodePath = new ConcurrentHashSet<>();

    @Override
    public void delete(String path){
        //never mind if ephemeral
        persistentExistNodePath.remove(path);
        deletePath(path);
    }

    @Override
    public void create(String path, boolean ephemeral) {
        if (!ephemeral) {
            if(persistentExistNodePath.contains(path)){
                return;
            }
            if (checkExists(path)) {
                persistentExistNodePath.add(path);
                return;
            }
        }
        int i = path.lastIndexOf('/');
        if (i > 0) {
            create(path.substring(0, i), false);
        }
        if (ephemeral) {
            createEphemeral(path);
        } else {
            createPersistent(path);
            persistentExistNodePath.add(path);
        }
    }

    protected abstract void deletePath(String path);

    protected abstract void createPersistent(String path);

    protected abstract void createEphemeral(String path);

    protected abstract boolean checkExists(String path);

    ...
}
```

流程中刻意抹去了临时节点相关的处理，和持久型节点创建过程不同的时，临时节点的创建只关注整个 path，因为它不需要记录在 `persistentExistNodePath` 容器中。

流程用到的几个和 ZK 服务打交道，用于操作数据节点的方法，都被声明为抽象的，留在子类实现，如下，可见 Curator 使用的流逝编程风格中，大体是先指定 action 行为，再定义该行为发生在哪条 path 路径上，代码十分干净简洁。

```

public class CuratorZookeeperClient extends AbstractZookeeperClient
    <CuratorZookeeperClient.CuratorWatcherImpl,
CuratorZookeeperClient.CuratorWatcherImpl> {

    @Override
    protected void deletePath(String path) {
        try {
            client.delete().forPath(path);
        } catch (NoNodeException e) {
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    public void createPersistent(String path) {
        try {
            client.create().forPath(path);
        } catch (NodeExistsException e) {
            logger.warn("ZNode " + path + " already exists.", e);
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    public void createEphemeral(String path) {
        try {
            client.create().withMode(CreateMode.EPHEMERAL).forPath(path);
        } catch (NodeExistsException e) {
            logger.warn("ZNode " + path + " already exists, since we will" +
                " only try to recreate a node on a session expiration" +
                ", this duplication might be caused by a delete delay" +
                " from the zk server, which means the old expired session" +
                " may still holds this ZNode and the server just hasn't" +
                " got time to do the deletion. In this case, " +
                "we can just try to delete and create again.", e);
            deletePath(path);
            createEphemeral(path);
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    public boolean checkExists(String path) {
        try {
            if (client.checkExists().forPath(path) != null) {
                return true;
            }
        } catch (Exception e) {
        }
        return false;
    }

    ...
}

```

JAVA

`createEphemeral(path)` 逻辑中，若 catch 到异常，会先将对应 path 先删除，再调用一次该方法，其原因在日志逻辑中有体现。

## 节点数据存取

从附带数据的节点做数据的存取操作，其基本流程和上述增删节点没有太大差别，很容易理解，所有代码如下。数据存取前都会先判断对应 path 的节点是否存在，存入时，如果 path 存在则先删除叶节点内容，再创建该节点，附入数据。

```
public abstract class AbstractZookeeperClient<TargetDataListener, TargetChildListener>  
    implements ZookeeperClient {  
    @Override  
    public void create(String path, String content, boolean ephemeral) {  
        if (checkExists(path)) {  
            delete(path);  
        }  
        int i = path.lastIndexOf('/');  
        if (i > 0) {  
            create(path.substring(0, i), false);  
        }  
        if (ephemeral) {  
            createEphemeral(path, content);  
        } else {  
            createPersistent(path, content);  
        }  
    }  
  
    @Override  
    public String getContent(String path) {  
        if (!checkExists(path)) {  
            return null;  
        }  
        return doGetContent(path);  
    }  
  
    protected abstract void createPersistent(String path, String data);  
  
    protected abstract void createEphemeral(String path, String data);  
  
    protected abstract String doGetContent(String path);  
    ...  
}
```

`create(path, content, ephemeral)` 方法在 catch 到 `NodeExistsException` 异常时，表示 path 对应节点存在，则直接设入数据，这表明了父类中的 `createPersistent(path, data)` 方法不必执行首个 `if` 语句块，但这从最大程度兼容各种 ZK 客户端的实现这一角度讲，这又是很合理的。

```

public class CuratorZookeeperClient extends AbstractZookeeperClient
    <CuratorZookeeperClient.CuratorWatcherImpl,
CuratorZookeeperClient.CuratorWatcherImpl> {

    static final Charset CHARSET = Charset.forName("UTF-8");

    @Override
    protected void createPersistent(String path, String data) {
        byte[] dataBytes = data.getBytes(CHARSET);
        try {
            client.create().forPath(path, dataBytes);
        } catch (NodeExistsException e) {
            try {
                client.setData().forPath(path, dataBytes);
            } catch (Exception e1) {
                throw new IllegalStateException(e.getMessage(), e1);
            }
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    protected void createEphemeral(String path, String data) {
        byte[] dataBytes = data.getBytes(CHARSET);
        try {
            client.create().withMode(CreateMode.EPHEMERAL).forPath(path, dataBytes);
        } catch (NodeExistsException e) {
            logger.warn("ZNode " + path + " already exists, since we will" +
                    " only try to recreate a node on a session expiration" +
                    ", this duplication might be caused by a delete delay" +
                    " from the zk server, which means the old expired session" +
                    " may still holds this ZNode and the server just hasn't" +
                    " got time to do the deletion. In this case, " +
                    "we can just try to delete and create again.", e);
            deletePath(path);
            createEphemeral(path, data);
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    public String doGetContent(String path) {
        try {
            byte[] dataBytes = client.getData().forPath(path);
            return (dataBytes == null || dataBytes.length == 0) ? null : new
String(dataBytes, CHARSET);
        } catch (NoNodeException e) {
            // ignore NoNode Exception.
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
        return null;
    }
}

```

JAVA

```
...  
}
```

## 子节点获取

`getChildren(path)` 是除 `isConnected()` 方法以外的唯一在子类中直接实现接口方法，实现也很简单。

```
public List<String> getChildren(String path) {  
    try {  
        return client.getChildren().forPath(path);  
    } catch (NoNodeException e) {  
        return null;  
    } catch (Exception e) {  
        throw new IllegalStateException(e.getMessage(), e);  
    }  
}
```

JAVA

## 节点变动监听

超类 `AbstractZookeeperClient` 声明了两个泛型参数 `<TargetDataListener, TargetChildListener>`，分别是用于监听节点数据和子节点变化情况的监听器。ZK客户端类图的右边部分中显示，声明了分别用于监听ZK客户端状态 `StateListener`、节点数据 `DataListener`、子节点 `ChildListener` 变化的 3 个监听器，为啥这里还要使用泛型了？

Dubbo是做RPC框架向外提供服务的，不仅需要考虑可扩展性，还得隔离掉它所依赖的第三方组件模块，避免应用代码、Dubbo、第三方组件 3 者耦合在一起。尽管Dubbo中实现 ZK 客户端只使用了 `Curator`，但并不意味着它不会直接使用 ZK 本身或者其他第三方提供的客户端工具，这就需要最大程度的考虑通用性；另外，无论使用什么工具在Dubbo中实现 ZK 客户端，ZK 服务实例的变化都是推送到本地，而非主动拉取，也就是说基于响应的监听模式是必定要采用的，而监听器都是第三方提供的，直接向框架上层暴露这种设计师很糟糕的，不仅失去灵活性，还使得扩展实现变得困难，代码变得复杂而难以管理。

这样，另一个问题又来了，监听器逻辑是一种被动调用行为，并不是调用方自己触发的，那处于上层 ZK 客户端定义的 3 个监听器是谁触发执行的呢？

不难看出，第三方 ZK 客户端的监听器和本地的监听器是一一对应的，前者是由 `ZookeeperClient` 实现使用其提供的接口注入的，而 `ZookeeperClient` 对后者能执行更多的控制，因此可以在前者的回调逻辑中调用后者。进一步说，`TargetDataListener` 和 `TargetChildListener` 是面向 `ZookeeperClient` 实现的，其实现由Dubbo的 ZK 客户端提供，实现逻辑被固化下来了，而 `StateListener`、`DataListener`、`ChildListener` 则是面向 `ZookeeperClient` 的使用方的。

啰嗦了这么多后，就不难理解 `AbstractZookeeperClient` 声明了如下述的两个两级 Map 容器了（`DataListener` 和 `ChildListener` 监听具体节点的变化而非整个ZK客户端）：

```
private final ConcurrentMap<String, ConcurrentMap<ChildListener, TargetChildListener>>  
    childListeners = new ConcurrentHashMap<>();  
  
private final ConcurrentMap<String, ConcurrentMap<DataListener, TargetDataListener>>  
    listeners = new ConcurrentHashMap<>();
```

JAVA

如下述父类源码实现，每次增删监听器都得深入到 Map 容器第二级，调用方增加一个 ChildListener 或 DataListener 监听器，父类会调用其扩张实现类的方法创建 TargetChildListener 或 TargetDataListener 监听器。代码中将创建和增加当做两个不同的行为，原因是如果容器中若存在对应的监听器则可重用，无需再创建，只需在节点上加入该监听器就可。

```
public abstract class AbstractZookeeperClient<TargetDataListener, TargetChildListener>JAVA
    implements ZookeeperClient {

    @Override
    public List<String> addChildListener(String path, final ChildListener listener) {
        ConcurrentMap<ChildListener, TargetChildListener> listeners =
        childListeners.get(path);
        if (listeners == null) {
            childListeners.putIfAbsent(path, new ConcurrentHashMap<ChildListener,
TargetChildListener>());
            listeners = childListeners.get(path);
        }
        TargetChildListener targetListener = listeners.get(listener);
        if (targetListener == null) {
            listeners.putIfAbsent(listener, createTargetChildListener(path, listener));
            targetListener = listeners.get(listener);
        }
        return addTargetChildListener(path, targetListener);
    }

    @Override
    public void addDataListener(String path, DataListener listener) {
        this.addDataListener(path, listener, null);
    }

    @Override
    public void addDataListener(String path, DataListener listener, Executor executor)
{
        ConcurrentMap<DataListener, TargetDataListener> dataListenerMap =
        listeners.get(path);
        if (dataListenerMap == null) {
            listeners.putIfAbsent(path, new ConcurrentHashMap<DataListener,
TargetDataListener>());
            dataListenerMap = listeners.get(path);
        }
        TargetDataListener targetListener = dataListenerMap.get(listener);
        if (targetListener == null) {
            dataListenerMap.putIfAbsent(listener, createTargetDataListener(path,
listener));
            targetListener = dataListenerMap.get(listener);
        }
        addTargetDataListener(path, targetListener, executor);
    }

    @Override
    public void removeDataListener(String path, DataListener listener ) {
        ConcurrentMap<DataListener, TargetDataListener> dataListenerMap =
        listeners.get(path);
        if (dataListenerMap != null) {
            TargetDataListener targetListener = dataListenerMap.remove(listener);
            if(targetListener != null){
                removeTargetDataListener(path, targetListener);
            }
        }
    }
}
```

```

@Override
public void removeChildListener(String path, ChildListener listener) {
    ConcurrentMap<ChildListener, TargetChildListener> listeners =
childListeners.get(path);
    if (listeners != null) {
        TargetChildListener targetListener = listeners.remove(listener);
        if (targetListener != null) {
            removeTargetChildListener(path, targetListener);
        }
    }
}

protected abstract TargetChildListener createTargetChildListener(String path,
ChildListener listener);

protected abstract List<String> addTargetChildListener(String path,
TargetChildListener listener);

protected abstract TargetDataListener createTargetDataListener(String path,
DataListener listener);

protected abstract void addTargetDataListener(String path, TargetDataListener
listener);

protected abstract void addTargetDataListener(String path, TargetDataListener
listener, Executor executor);

protected abstract void removeTargetDataListener(String path, TargetDataListener
listener);

protected abstract void removeTargetChildListener(String path, TargetChildListener
listener);
...
}

```

源码的最后还有多达 7 个带子类实现的抽象方法，着实令人晕懵。到子类看其实现，有关于子节点的监听器的增删处理相对简单很多。

实现中关于 Curator 中的 TreeCache 不能直白的理解，可以翻看 [Zookeeper 客户端 Curator 使用详解](#) (<https://www.throwable.club/2018/12/16/zookeeper-curator-usage>)一文先了解。

**“ Zookeeper 原生支持通过注册 Watcher 来进行事件监听，但是开发者需要反复注册 (Watcher 只能单次注册单次使用)。 Cache 是 Curator 中对事件监听的包装，可以看作是对事件监听的本地缓存视图，能够自动为开发者处理反复注册监听。 ”**

*Path Cache* 用来监控一个 ZNode 的子节点。当一个子节点增加、更新、删除时， *Path Cache* 会改变它的状态，会包含最新的子节点，子节点的数据和状态。

*Node Cache* 与 *Path Cache* 类似， *Node Cache* 只是监听某一个特定的节点。

*Tree Cache* 可以监控整个子树上的所有节点，类似于 *PathCache* 和 *NodeCache* 的组合。

## IMPORTANT

### 连接掉线处理

当连接掉线时，*TreeCache* 将继续保持它在失去连接之前的状态，在连接恢复后，*TreeCache* 将为所有在断开连接期间发生的添加、删除和更新发出正常的子事件。

### 初始化时的同步处理 *INITIALIZED*

在启动时，缓存与服务器同步其内部状态，在发现新节点时发布一系列 *NODE\_ADDED* 事件。一旦缓存被完全同步，*INITIALIZED* 事件就会被发布。此事件之后发布的所有事件都表示实际的服务器端变化。

在重新连接时，缓存将重新将其内部状态与服务器同步，并在其内部状态完全刷新后再次触发 *INITIALIZED* 事件。

注意：由于初始化时的填充过程本就是异步的，所以可以在发布 *INITIALIZED* 事件之前观察服务器端更改（例如 *NODE\_UPDATED*）。

从下述源码可以看出，Dubbo 中的 ZK 客户端实现仅允许一个节点存在一个对应的 *TreeCache* 对象 path 作为映射的 key，但是实现上存在一个 bug，如果对应的 key 存在对应的 *TreeCache* 对象时，它便会变成游离状态，但是除非被垃圾回收，否则加入到其中的 *treeCacheListener* 也同时在起作用。

另外可以看出在为 *TreeCache* 增加数据监听器时，可以另行指定执行监听器的 Executor。

```
public class CuratorZookeeperClient extends AbstractZookeeperClient
    <CuratorZookeeperClient.CuratorWatcherImpl,
CuratorZookeeperClient.CuratorWatcherImpl> {

    private Map<String, TreeCache> treeCacheMap = new ConcurrentHashMap<>();

    @Override
    public CuratorZookeeperClient.CuratorWatcherImpl createTargetChildListener(String
path, ChildListener listener) {
        return new CuratorZookeeperClient.CuratorWatcherImpl(client, listener, path);
    }

    @Override
    public List<String> addTargetChildListener(String path, CuratorWatcherImpl
listener) {
        try {
            return client.getChildren().usingWatcher(listener).forPath(path);
        } catch (NoNodeException e) {
            return null;
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
    }

    @Override
    public void removeTargetChildListener(String path, CuratorWatcherImpl listener) {
        listener.unwatch();
    }

    @Override
    protected CuratorZookeeperClient.CuratorWatcherImpl createTargetDataListener(String
path, DataListener listener) {
        return new CuratorWatcherImpl(client, listener);
    }

    @Override
    protected void addTargetDataListener(String path,
CuratorZookeeperClient.CuratorWatcherImpl treeCacheListener) {
        this.addTargetDataListener(path, treeCacheListener, null);
    }

    @Override
    protected void addTargetDataListener(String path,
CuratorZookeeperClient.CuratorWatcherImpl treeCacheListener, Executor executor) {
        try {
            TreeCache treeCache = TreeCache.newBuilder(client,
path).setCacheData(false).build();
            treeCacheMap.putIfAbsent(path, treeCache);

            if (executor == null) {
                treeCache.getListenable().addListener(treeCacheListener);
            } else {
                treeCache.getListenable().addListener(treeCacheListener, executor);
            }

            treeCache.start();
        } catch (Exception e) {
            throw new IllegalStateException("Failed to start tree cache for path " + path, e);
        }
    }
}
```

JAVA

```

        } catch (Exception e) {
            throw new IllegalStateException("Add treeCache listener for path:" + path,
e);
        }
    }

@Override
protected void removeTargetDataListener(String path,
CuratorZookeeperClient.CuratorWatcherImpl treeCacheListener) {
    TreeCache treeCache = treeCacheMap.get(path);
    if (treeCache != null) {
        treeCache.getListenable().removeListener(treeCacheListener);
    }
    treeCacheListener.dataListener = null;
}

...
}

```

泛型参数监听器 `TargetChildListener` 和 `TargetDataListener` 在子类被实现在同一个类上，但从效果上来讲，这策略并不理想，导致不必要的内存开销，“汝之蜜糖，彼之砒霜”没那么严重。由Dubbo为框架上层实现的 `ChildListener` 或 `DataListener` 在 Curator 的 ZK 客户端监听器实现作为属性出现，内置事件回调和调用 `listener.unwatch()` 或 `treeCacheListener.dataListener = null` 的并不在同一个线程中，因此有并发的问题存在，因而声明了 `volatile` 可见性修饰符。

`TreeCacheListener` 主要响应的节点本身的状态变化，包括其上数据的变动，另外当前连接的状态变化也包括在响应中，Dubbo收集这些变化并回调框架上层实现 `dataChanged()` 方法。

```

public interface DataListener {

    void dataChanged(String path, Object value, EventType eventType);
}

public enum EventType {
    None(-1),
    NodeCreated(1),
    NodeDeleted(2),
    NodeDataChanged(3),
    NodeChildrenChanged(4),
    CONNECTION_SUSPENDED(11),
    CONNECTION_RECONNECTED(12),
    CONNECTION_LOST(12),
    INITIALIZED(10);

    private final int intValue;
    EventType(int intValue) {
        this.intValue = intValue;
    }

    public int getIntValue() {
        return intValue;
    }
}

static class CuratorWatcherImpl implements TreeCacheListener {

    private volatile DataListener dataListener;

    public CuratorWatcherImpl(CuratorFramework client, DataListener dataListener) {
        this.dataListener = dataListener;
    }

    @Override
    public void childEvent(CuratorFramework client, TreeCacheEvent event) throws Exception {
        if (dataListener != null) {
            if (logger.isDebugEnabled()) {
                logger.debug("listen the zookeeper changed. The changed data:" + event.getData());
            }
            TreeCacheEvent.Type type = event.getType();
            EventType eventType = null;
            String content = null;
            String path = null;
            switch (type) {
                case NODE_ADDED:
                    eventType = EventType.NodeCreated;
                    path = event.getData().getPath();
                    content = event.getData().getData() == null ? "" : new String(event.getData().getData(), CHARSET);
                    break;
                case NODE_UPDATED:
                    eventType = EventType.NodeDataChanged;
                    path = event.getData().getPath();
                    content = event.getData().getData() == null ? "" : new

```

```

String(event.getData().getData(), CHARSET);
        break;
    case NODE_REMOVED:
        path = event.getData().getPath();
        eventType = EventType.NodeDeleted;
        break;
    case INITIALIZED:
        eventType = EventType.INITIALIZED;
        break;
    case CONNECTION_LOST:
        eventType = EventType.CONNECTION_LOST;
        break;
    case CONNECTION_RECONNECTED:
        eventType = EventType.CONNECTION_RECONNECTED;
        break;
    case CONNECTION_SUSPENDED:
        eventType = EventType.CONNECTION_SUSPENDED;
        break;

    }
    dataListener.dataChanged(path, content, eventType);
}
}
}

```

由上述源码可见，`INITIALIZED`、`CONNECTION\_LOST`、`CONNECTION\_RECONNECTED`、`CONNECTION\_SUSPENDED` 对应的是 ZK 服务的状态变化，而 `NodeCreated`、`NODE\_UPDATED`、`NODE\_REMOVED` 对应的是节点包括数据在内的自身变化。

子节点变化事件类型有 `None`<sup>-1</sup>，`NodeCreated`<sup>1</sup>，`NodeDeleted`<sup>2</sup>，`NodeDataChanged`<sup>3</sup>，`NodeChildrenChanged`<sup>4</sup> 5 种，ZK 客户端连入或者从 ZK 服务断开时的响应事件类型为 `None`，其余 4 种，均执行 `client.getChildren().usingWatcher(this).forPath(path)`，相当于在当前 `path` 节点重新赋值了监听器，并且取得该节点下的所有子节点。

```
public interface ChildListener {
```

```
    void childChanged(String path, List<String> children);
```

```
}
```

```
static class CuratorWatcherImpl implements CuratorWatcher {
```

```
    private CuratorFramework client;
    private volatile ChildListener childListener;
    private String path;

    public CuratorWatcherImpl(CuratorFramework client, ChildListener listener, String
path) {
        this.client = client;
        this.childListener = listener;
        this.path = path;
    }
```

```
    public void unwatch() {
        this.childListener = null;
    }
```

```
    @Override
    public void process(WatchedEvent event) throws Exception {
        if (event.getType() == Watcher.Event.EventType.None) {
            return;
        }

        if (childListener != null) {
            childListener.childChanged(path,
client.getChildren().usingWatcher(this).forPath(path));
        }
    }
}
```

## ZK 服务状态监听

Curator 客户端实现了到ZK服务实例的连接状态管理，有如下 5 种，Dubbo 的 CuratorZookeeperClient 也对应设置了几个整形的状态值，当ZK状态发生变化回调设置给 Curator 的 ConnectionStateListener 监听器时，后者会附上该状态值触发回调由Dubbo定义的 StateListener 接口。

- CONNECTED - CONNECTED = 1 : 和ZK处于连接状态；
- SUSPENDED - SUSPENDED = 3 : 连接断开了，和 LOST 的区别是，后者是正式切断的连接；
- RECONNECTED - RECONNECTED = 2 or NEW\_SESSION\_CREATED = 4 : 曾经转入到 suspended、lost、或 read-only 状态的连接已重新建立；
- LOST - SESSION\_LOST = 0 : Curator一旦认为ZK会话超时便设状态为 LOST : 1) ZK告知会话超时；2) Curator关闭了其内部管理的ZK连接实例；3) 设置的会话超时因网络分裂而过期；

JAVA

- READ\_ONLY - 略：当因网络设备故障时导致了网络一分为二网络分裂，调用方可以使用 `CuratorFrameworkFactory.Builder#canBeReadOnly()` 标识成只读状态；

在 `dubbo-remoting-zookeeper` 中定义包括 `StateListener` 在内的监听器由 Dubbo 负责在超类 `AbstractZookeeperClient` 统一做存取维护，其回调是由响应 ZK 事件的定义在 `Curator` 中的监听器完成的。

```
JAVA
```

```

public interface StateListener {
    int SESSION_LOST = 0;
    int CONNECTED = 1;
    int RECONNECTED = 2;
    int SUSPENDED = 3;
    int NEW_SESSION_CREATED = 4;
    void stateChanged(int connected);
}

public abstract class AbstractZookeeperClient<TargetDataListener, TargetChildListener>
    implements ZookeeperClient {

    private final Set<StateListener> stateListeners = new
    CopyOnWriteArraySet<StateListener>();

    @Override
    public void addStateListener(StateListener listener) {
        stateListeners.add(listener);
    }

    @Override
    public void removeStateListener(StateListener listener) {
        stateListeners.remove(listener);
    }

    public Set<StateListener> getSessionListeners() {
        return stateListeners;
    }

    protected void stateChanged(int state) {
        for (StateListener sessionListener : getSessionListeners()) {
            sessionListener.stateChanged(state);
        }
    }

    ...
}

```

方法 `stateChanged(state)` 轮询回调所有的加入到集合 `stateListeners` 中的 ZK 客户端状态监听器，从而驱动着相关的业务逻辑。

`CuratorZookeeperClient` 初始化时为 `Curator` 指定了状态监听器——`CuratorConnectionStateListener`，每一次状态的变更都会由它负责间接调用上述 `stateChanged(state)` 方法。

```
public class CuratorZookeeperClient extends AbstractZookeeperClient
    <CuratorZookeeperClient.CuratorWatcherImpl,
CuratorZookeeperClient.CuratorWatcherImpl> {
```

```
private class CuratorConnectionStateListener implements ConnectionStateListener {
    private final long UNKNOWN_SESSION_ID = -1L;

    private long lastSessionId;
    private URL url;

    public CuratorConnectionStateListener(URL url) {
        this.url = url;
    }

    @Override
    public void stateChanged(CuratorFramework client, ConnectionState state) {
        int timeout = url.getParameter(TIMEOUT_KEY, DEFAULT_CONNECTION_TIMEOUT_MS);
        int sessionExpireMs = url.getParameter(ZK_SESSION_EXPIRE_KEY,
DEFAULT_SESSION_TIMEOUT_MS);

        long sessionId = UNKNOWN_SESSION_ID;
        try {
            sessionId = client.getZookeeperClient().getZooKeeper().getSessionId();
        } catch (Exception e) {
            logger.warn("Curator client state changed, but failed to get the
related zk session instance.");
        }

        if (state == ConnectionState.LOST) {
            logger.warn("Curator zookeeper session " +
Long.toHexString(lastSessionId) + " expired.");
            CuratorZookeeperClient.this.stateChanged(StateListener.SESSION_LOST);
        } else if (state == ConnectionState.SUSPENDED) {
            logger.warn("Curator zookeeper connection of session " +
Long.toHexString(sessionId) + " timed out. " +
"connection timeout value is " + timeout +
", session expire timeout value is " + sessionExpireMs);
            CuratorZookeeperClient.this.stateChanged(StateListener.SUSPENDED);
        } else if (state == ConnectionState.CONNECTED) {
            lastSessionId = sessionId;

            logger.info("Curator zookeeper client instance initiated successfully,
session id is "
                + Long.toHexString(sessionId));

            CuratorZookeeperClient.this.stateChanged(StateListener.CONNECTED);
        } else if (state == ConnectionState.RECONNECTED) {
            if (lastSessionId == sessionId && sessionId != UNKNOWN_SESSION_ID) {
                logger.warn("Curator zookeeper connection recovered from connection
lose, " +
"reuse the old session " + Long.toHexString(sessionId));
            }
        }
    }

    CuratorZookeeperClient.this.stateChanged(StateListener.RECONNECTED);
} else {
    logger.warn("New session created after old session lost, " +
"old session " + Long.toHexString(lastSessionId))
```

JAVA

```
        + ", new session " + Long.toHexString(sessionId));
lastSessionId = sessionId;

CuratorZookeeperClient.this.stateChanged(StateListener.NEW_SESSION_CREATED);
    }
}

}
...
}
```

除了打印日志实现看起来有点啰嗦，整个代码实现很简洁。每次状态变更回调时，`CuratorConnectionStateListener` 均会获取到当前 ZK 服务中当前会话的 `sessionId` 编号，`ConnectionState.CONNECTED` 状态回调时直接将其赋值给 `lastSessionId`，而 `ConnectionState.RECONNECTED` 状态回调中则用其同最近记录的 `lastSessionId` 对比，如果相等，则说明了连接断开后从同一会话中恢复了，否则表示一个新的会话的创建。

## ZK 动态配置中心

《Dubbo 配置管理》一文中已经详述了动态配置相关的内容，本章节只着重剖析 ZK 版的实现。由 `DynamicConfiguration` 的定义可知，需要提供从 ZK 服务获取配置项或治理规则的功能实现。另外，接口中还定义了针对节点增删 `ConfigurationListener` 监听器的两个方法，然而这仅仅是定义了一种行为规范而已，其回调处理还得由 ZK 动态配置中心负责。

### CacheListener

从前文关于利用 Curator 实现 ZK 客户端的剖析中可知，ZK 服务中的数据或节点变更事件是由对端推送到本地的，紧接着 Curator 接入请求便驱动回调 ZK 客户端实现的监听器（由 Curator 内部定义）A，A 的回调由进一步引起和 A 一一绑定的由 ZK 客户端定义提供的监听器（如 `DataListener`）B 的调用。延用此方法，动态配置中心提供的 B 实现，可以在回调逻辑中调用 `ConfigurationListener` 监听器 C，也就是这个过程形成了一条类似 A → B → C 的监听器驱动链条。接下来先看看中间 B 的实现——`CacheListener`。

不同于 A 和 B 的一一绑定关系，B 和 C 是一对多的，映射关系由 path 体现，因此 `CacheListener` 声明了 `Map<String, Set<ConfigurationListener>>` 类型的容器来关联 C —— `ConfigurationListener`。

```

public class CacheListener implements DataListener {
    private static final int MIN_PATH_DEPTH = 5;

    private Map<String, Set<ConfigurationListener>> keyListeners = new
ConcurrentHashMap<>();

    public void addListener(String key, ConfigurationListener configurationListener) {
        Set<ConfigurationListener> listeners = this.keyListeners.computeIfAbsent(key, k
-> new CopyOnWriteArraySet<>());
        listeners.add(configurationListener);
    }

    public void removeListener(String key, ConfigurationListener configurationListener)
{
        Set<ConfigurationListener> listeners = this.keyListeners.get(key);
        if (listeners != null) {
            listeners.remove(configurationListener);
        }
    }
    ...
}

```

如下述源码中关于 DataListener 事件回调实现中， CacheListener 只关注指定深度 (5, 配置路径 /{namespace}/config/{group}/{key} 斜杆切分后, 分为5段) 下的的如下变化事件：

1. ConfigChangeType.ADDED | ConfigChangeType.MODIFIED , 含数据
2. ConfigChangeType.DELETED , 允许附带数据为空

收到这类事件通知时, 便组装一个 ConfigChangeEvent 对象回调所有对应当前 path 的 ConfigurationListener 监听器。关于 initializedLatch , 等会会提及。

```
public class CacheListener implements DataListener {
    private static final int MIN_PATH_DEPTH = 5;

    private CountDownLatch initializedLatch;
    private String rootPath;

    public CacheListener(String rootPath, CountDownLatch initializedLatch) {
        this.rootPath = rootPath;
        this.initializedLatch = initializedLatch;
    }

    @Override
    public void dataChanged(String path, Object value, EventType eventType) {
        if (eventType == null) {
            return;
        }

        if (eventType == EventType.INITIALIZED) {
            initializedLatch.countDown();
            return;
        }

        if (path == null || (value == null && eventType != EventType.NodeDeleted)) {
            return;
        }

        if (path.split("/").length >= MIN_PATH_DEPTH) {
            ConfigChangeType changeType;
            switch (eventType) {
                case NodeCreated:
                    changeType = ConfigChangeType.ADDED;
                    break;
                case NodeDeleted:
                    changeType = ConfigChangeType.DELETED;
                    break;
                case NodeDataChanged:
                    changeType = ConfigChangeType.MODIFIED;
                    break;
                default:
                    return;
            }
        }

        //对path做一些简单的处理，转换为以"."作为分隔符的字符串
        String key = pathToKey(path);

        ConfigChangeEvent configChangeEvent = new ConfigChangeEvent(key, (String)
value, changeType);
        Set<ConfigurationListener> listeners = keyListeners.get(path);
        if (CollectionUtils.isNotEmpty(listeners)) {
            listeners.forEach(listener -> listener.process(configChangeEvent));
        }
    }
}
```

JAVA

## ZookeeperDynamicConfiguration

最后回到 DynamicConfiguration 的实现类 ZookeeperDynamicConfiguration 剖析上来，它是本章节的重点。后者将增删 ConfigurationListener 监听器的实现委托给了 CacheListener，如下：

```
public class ZookeeperDynamicConfiguration implements DynamicConfiguration {  
    ...  
    @Override  
    public void addListener(String key, String group, ConfigurationListener listener) {  
        cacheListener.addListener(getPathKey(group, key), listener);  
    }  
  
    @Override  
    public void removeListener(String key, String group, ConfigurationListener  
        listener) {  
        cacheListener.removeListener(getPathKey(group, key), listener);  
    }  
  
    private String getPathKey(String group, String key) {  
        return rootPath + PATH_SEPARATOR + group + PATH_SEPARATOR + key;  
    }  
}
```

实现中用 getPathKey(group, key) 方法构造的 path key 为 ZK 服务中完整目录路径，形如 /{namespace}/config/{group}/{key}，其中 namespace 的值默认为 "dubbo"，rootPath 取值为 "/" + (url["namespace"] | "dubbo") + "/config"。组成中 {key} 的构成一般会遵守约定俗成的规范，指定配置所属的微服务或应用、配置的功能范畴，或者简单信息，示例如 {service}.configurators, {service}.tagrouters, {group}.dubbo.properties。

显然 path key 是标识叶节点的一维字符串呈现，当然可以用于从 ZK 服务中获取配置数据。如下，获取治理规则 getRule(...) 和一般配置项 getProperties(...) 的实现并没有什么区别。

```
public class ZookeeperDynamicConfiguration implements DynamicConfiguration {
    @Override
    public Object getInternalProperty(String key) {
        return zkClient.getContent(key);
    }

    @Override
    public String getRule(String key, String group, long timeout) throws
IllegalStateException {
        return (String) getInternalProperty(getPathKey(group, key));
    }

    @Override
    public String getProperties(String key, String group, long timeout) throws
IllegalStateException {
        if (StringUtils.isEmpty(group)) {
            group = DEFAULT_GROUP;
        }
        return (String) getInternalProperty(getPathKey(group, key));
    }
    ...
}
```

JAVA

到这里，构建一个读取 ZK 配置中心数据的客户端 ZookeeperDynamicConfiguration 实例，内部实现分为两个步骤：

1. 首先需要使用 Curator 建立一个接入 ZK 服务的连接；
2. 然后针对该连接加入 DataListener 数据监听器—— CacheListener；

其构造函数如下，其中使用了一个并发安全的 CountDownLatch 类型的计数器 initializedLatch，并将其作为构造函数的参数交给了 CacheListener CListener，此后便利用 ZookeeperTransporter 这个创建工作建立到 ZK 服务的连接，获得一个客户端的实例，最后为该实例调用其 addDataListener(...) 方法加入 CListener 监听器。

```

public class ZookeeperDynamicConfiguration implements DynamicConfiguration {
    private static final Logger logger =
LoggerFactory.getLogger(ZookeeperDynamicConfiguration.class);

    private Executor executor;
    private String rootPath;
    private final ZookeeperClient zkClient;
    private CountDownLatch initializedLatch;

    private CacheListener cacheListener;
    private URL url;

    ZookeeperDynamicConfiguration(URL url, ZookeeperTransporter zookeeperTransporter) {
        this.url = url;
        rootPath = PATH_SEPARATOR + url.getParameter(CONFIG_NAMESPACE_KEY,
DEFAULT_GROUP) + "/config";

        initializedLatch = new CountDownLatch(1);
        this.cacheListener = new CacheListener(rootPath, initializedLatch);
        this.executor = Executors.newFixedThreadPool(1, new
NamedThreadFactory(this.getClass().getSimpleName(), true));

        zkClient = zookeeperTransporter.connect(url);

        //监听"/{namespace}|dubbo/config"下的所有事件
        zkClient.addDataListener(rootPath, cacheListener, executor);
        try {
            // Wait for connection
            long timeout = url.getParameter("init.timeout", 5000);
            boolean isCountDown = this.initializedLatch.await(timeout,
TimeUnit.MILLISECONDS);
            if (!isCountDown) {
                throw new IllegalStateException("Failed to receive INITIALIZED event
from zookeeper, pls. check if url "
                    + url + " is correct");
            }
        } catch (InterruptedException e) {
            logger.warn("Failed to build local cache for config center (zookeeper)." +
url);
        }
    }
    ...
}

```

JAVA

对照上述源码，再回到 CacheListener 实现源码中，看看其属性 initializedLatch 的作用。ZK 服务和本地客户端是两个经过网络链路连接的主机进程，万水千山后成功建立连接，并不意味着 Curator 客户端也已就绪——需要同步所关注目录的所有情况（参考 Curator 的 INITIALIZED 事件），因此为保证此后的配置读取不发生错误，需要将当前线程阻塞。

那 ZK 客户端怎么知道 Zookeeper 动态配置中心什么时候就绪了？实际上上述源码分析中已经有讲过，ZK 服务会在诸如增删节点、节点数据变化、客户端接入或断开时给 Curator 客户端 CC 推送对应的通知，而 CC 在回调 ZK 客户端 ZC 提供的 TreeCacheListener 监听器时，事件类型被 ZC 由

`TreeCacheEvent.Type#INITIALIZED` 装换为 `EventType#INITIALIZED`，ZC 随后用它回调框架上层提供的 `EventListener` 监听器实现 `CacheListener`。

啰嗦了这么多，总之一句话，就绪通知在 `CacheListener` 这个监听器回调中收到，收到后正处于阻塞状态的 ZK 客户端便可返回，为应用层提供配置存取服务。

另外源码中还设置了超时，超时配置为 `(url["init.timeout"] | 5000)` 毫秒，连接建立后若超时，便抛异常处理。

## IMPORTANT

`DataListener` 监听器监听的是 ZK 服务内目录结构中的一棵子树上的所有发生事件，如动态配置聚焦于整个 `"/" + (url["namespace"] | "dubbo") + "/config"` 节点一棵子树，而由 `DataListener` 回调驱动的 `ConfigurationListener` 监听器则针对的是子树中某个叶节点的数据变化事件。

---

完结

# Dubbo远程通讯 · 网络传输层

真实世界一件很简单的事情，映射到计算机中，由于涉及太多细节，最后实现阶段往往会觉得非常复杂，因此善于抽象出合适的模型以近似表达真实世界的事物 是一种非常厉害的能力。作为RPC框架的Dubbo，就有很多这样的闪光点，比如前面提到的URL配置总线，本文将要涉及的远程通讯实现算一个。

## 概念点

在面向对象编程中，我们经常发现一种情况，功能差异特别大的两种实体，却在某方面有着很大的共性，而这共性方面刚好又恰恰是我们需要关注的，为了便于程序处理，我们会将这种共性抽象出来，但这也导致理解上的不便，Dubbo中的Endpoint<sup>端</sup>正是这样一种存在。

作为分布式系统构建的集成件，Dubbo的远程通讯采取的是传统CS架构，在Wiki上有如下一段描述：

**“Client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.[1] Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests. Examples of computer applications that use the client-server model are Email, network printing, and the World Wide Web.**

大概意思是"CS模式是一种分布式应用架构，提供资源或服务的为Server，而发出请求的是Client，他们一起协作参与完成任务或：工作负载。通常Server和Client能够通过彼此独立的硬件进行计算机网络通讯。Server中运行着一或多个程序，其资源由多个接入Client共享，而Client间却彼此独立，只能请求Server端的内容或者 调起其功能。因此由**Client**端启动与等待接入请求的**Server**端的通讯会话。电子邮件，网络打印和万维网都属于这种架构模式"。

## Endpoint——端

**Endpoint**, 是一个抽象概念, 可以认为它是一个能够发生网络通讯的节点, 端之间可以进行双向网络通讯, 基于此衍生出 **通道 (Channel)**、**客户端 (Client)**、**服务端 (Server)** 三个概念, 当然也可能如上述所说, 端是由这3者倒推抽象得到的。

端具有如下特征:

1. 它的身份由**URL**配置总线中的*URL*所表示, 这个URL只是用于在Dubbo唯一标识这个端, 并携带端所特有参数, 供设置、读取;
2. 能够发生通讯的前提是具有真实的物理网络地址, 由 **InetSocketAddress** 表示, 表示为 **ip:port** 或 **host:port** ;
3. 端能够主动发起消息传递操作, 也可以将自己从Dubbo管辖的集群下线;
4. 具有感知通讯事件的能力

如下述接口所示:

```
public interface Endpoint {  
    URL getUrl();  
  
    ChannelHandler getChannelHandler();  
  
    InetSocketAddress getLocalAddress();  
  
    void send(Object message) throws RemotingException;  
  
    //=====//sent, 过去分词, 用于控制同步还是异步发送消息的, true值表示消息已经成功发出, 实际对于同步IO来说, 无论如何当前方法都是需要阻塞到IO已经完成的  
    //其值不会有任何影响, 而对于支持异步IO的中间件, 比如Netty, true值, 则会直接等到消息已经成功发出, 才会返回, 否则为异步执行, 方法立即返回,  
    //这时IO还没执行完甚至是还没有启动执行  
    //=====  
    void send(Object message, boolean sent) throws RemotingException;  
  
    void close();  
  
    //Graceful close the channel.  
    void close(int timeout);  
  
    void startClose();  
  
    //检查端是否已关闭  
    boolean isClosed();  
}
```

## Channel——通道

**Channel**是**Client**和**Server**的信号传输通道，消息发送端会往通道输入消息，而接收端会从通道读消息。并且接收端发现通道没有消息，就去做其他事情了，不会造成阻塞。所以channel可以读也可以写，并且可以异步读写。可以这么认为，Channel就是一个消息管道，Client是消费方，一有需要就试着从通道取内容，而Server是提供方，按需要将消息断断续续或：源源不断装入通道，当然这过程也可以反过来。不同的是Client只能绑定一个Channel，而Server则能绑定多个，因此对应到Client和Server是一对多的关系。理论上二者同Channel间的关系都是聚合。

Dubbo中的**Channel**具有如下特征：

1. 可以向Channel写入或者从中读取上下文本地属性
2. 从Channel的一侧能够获取到另一侧的物理网络地址
3. 能够检测当前Channel的双端是否还处于连接状态

接口定义源码如下：

```
public interface Channel extends Endpoint {  
  
    InetSocketAddress getRemoteAddress();  
  
    boolean isConnected();  
  
    //=====  
    //attribute并不会经通道被发往对端，只是随Channel一起被绑定的属性值  
    //通讯过程虽然很短暂，但瞬间跨越大量方法栈帧（函数调用）  
    //能够就其绑定Channel本地值是非常必要的，便于跨帧获得上下文值，有点类似于线程本地变量容器ThreadLocal  
    //=====  
    boolean hasAttribute(String key);  
  
    Object getAttribute(String key);  
  
    void setAttribute(String key, Object value);  
  
    void removeAttribute(String key);  
}
```

## Client\Server——客户端\服务端

**Client**和**Server**分别是CS模式中的客户端和服务端，属于传输层，更多的体现的是语义上的差别，并不区分请求和应答职责，二者拥有的都是发送能力。但客户端拥有体现其特有职责的重连能力，连接肯定都是由客户端发起，它一般是在连接超时时由心跳任务发起。客户端没有显式的连接以及断连语义，在客户端被初始化出来时就默认开启并建立与服务端连接，且通过定时任务维护通道的连接状态。因此客户端和服务端除了重连以外都只有close和send两个影响网络输出的动作。

Dubbo中Client的实现使用继承而非聚合处理和Channel的关系，每次连入一个Client，Server端均会产生一个与之绑定的Channel，这些Channel信息会维护在一个聚合容器中，可以根据Client的网络物理地址获取到。

另外Client和Server均实现了 `Resetable` 和 `IdleSensible` 接口，前者用于重试本地上下文参数，而后者则是在检测到空闲连接时，能够做出相应处理， Server端关闭连接，而Client端则发送心跳到服务端。

接口定义源码如下：

```
public interface Client extends Endpoint, Channel, Resetable, IdleSensible {  
    void reconnect() throws RemotingException;  
}  
  
public interface Server extends Endpoint, Resetable, IdleSensible {  
    boolean isBound();  
    Collection<Channel> getChannels();  
    Channel getChannel(InetSocketAddress remoteAddress);  
}  
  
public interface Resetable {  
    void reset(URL url);  
}  
  
/**  
 * Indicate whether the implementation (for both server and client) has the ability to  
 * sense and handle idle connection.  
 * If the server has the ability to handle idle connection, it should close the  
 * connection when it happens, and if  
 * the client has the ability to handle idle connection, it should send the heartbeat  
 * to the server.  
 */  
public interface IdleSensible {  
    /**  
     * Whether the implementation can sense and handle the idle connection. By default  
     * it's false, the implementation  
     * relies on dedicated timer to take care of idle connection.  
     *  
     * @return whether has the ability to handle idle connection  
     */  
    default boolean canHandleIdle() {  
        return false;  
    }  
}
```

## ChannelHandler

Dubbo网络通讯中，Channel是Client和Server之间的信号传输器，过程中端间的存在着连接、发送消息、接收消息、断连、异常捕获这些行为，对应存在着**connected**、**sent**、**received**、**disconnected**、**caught**这5个网络事件。利用事件点进行功能扩展和增强正是很多编程框架和中间件的必备武功，我们知道Dubbo的端之间发生的基于socket的网络通讯，但是dubbo本身并不负责通讯IO的处理，这种行为由被委托Netty等第三方网络通讯组件负责，Dubbo将这些基本能力抽象，形成对外的统一扩展接口插件化，再由**ChannelHandler**在合适的事件点按照场景进行扩展增强处理。白话一点说，其具体实现是对第三方网络通讯组件进行适配，后者在I/O就绪后回调其提供的5个网络事件处理函数。

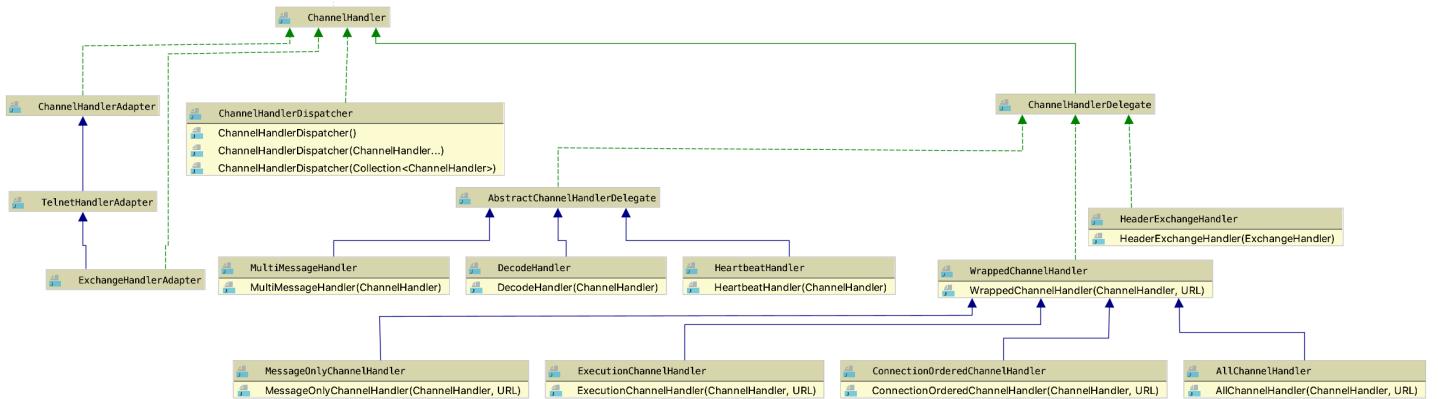
网络行为触发产生对应的事件，而事件也可以反过来触发新的网络行为。

**NOTE** eg: **HeartbeatHandler**在收到接收到请求后**received**事件，首先会确认是否为心跳请求，若是，则会通过接受消息的那个Channel发回一个心跳相应，进而触发了对端的**received**事件。

**ChannelHandler**采用装饰者模式方式实现，其接口定义如下：

```
@SPI  
public interface ChannelHandler {  
  
    void connected(Channel channel) throws RemotingException;  
  
    void disconnected(Channel channel) throws RemotingException;  
  
    void sent(Channel channel, Object message) throws RemotingException;  
  
    void received(Channel channel, Object message) throws RemotingException;  
  
    void caught(Channel channel, Throwable exception) throws RemotingException;  
}
```

理解**ChannelHandler**的实现是理解整个Dubbo远程通讯的一个关键点，由于其设计使用了装饰者模式，理解起来没有那么直观，具体请参考《Dubbo与设计模式》一文中的装饰者模式这一小章节便于后续理顺其实现逻辑，**ChannelHandler**的大体类UML图如下。

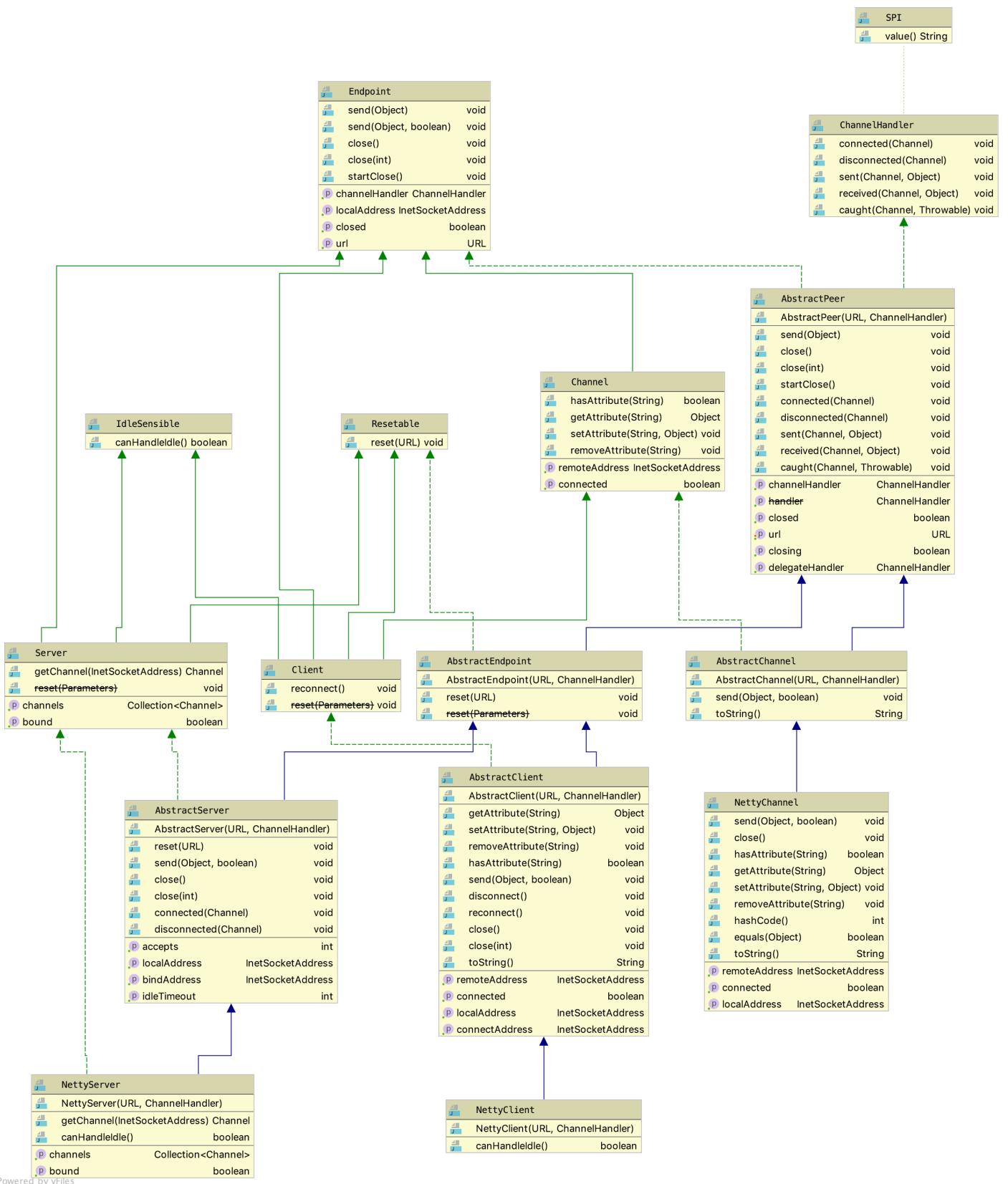


图：装饰模式-Decrator

上图中WrapChannelHandler是实现Dubbo线程派发的关键部分，具体请参考《Dubbo之线程管理》一文。

## 基础能力实现

如下UML生成图所示，无论是服务端、客户端，还是桥接二者的通道实现，都继承实现自 AbstractPeer，后者分别实现了EndPoint和ChannelHandler两个接口，这表明在Dubbo中的服务端、客户端和通道具有Endpoint的特性，同时还能感知并响应网络通讯事件。而其下的AbstractPoint 则定义了服务端和客户端重设参数的能力，AbstractChannel作为通道基类则显得过分简单。



Powered by yfiles

图: Dubbo的端实现

## NOTE

图中所表示的同时会出现在 `NettyChannel`、`NettyServer`、`NettyClient` 中的容易混淆的两个方法：`Endpoint` 定义的 `send()` 和 `ChannelHandler` 定义的 `sent()`。前者用于通过 Channel 通道主动向对方发送消息，而后者则是在已发送消息后告知处理结果用的，属于 I/O 响应事件回调。

## AbstractPeer

`AbstractPeer` 提炼抽象了通讯对端的公共能力，它具有响应通讯事件的能力实现 `ChannelHandler` 接口，但这种能力是委托给所引用的 `ChannelHandler` 达成的。`AbstractPeer` 中声明了两个 `volatile` 类型的表示端是否处于关闭状态的变量，该状态下，是禁止再向对端发送消息或者接受来自对端的消息的，也无法向对端发起连接请求。其实现关闭的方式也很简单，只需对应改变状态值。注意：这里所说的关闭操作实际上关闭的是所绑定对应的 `Channel`。

public abstract class AbstractPeer implements Endpoint, ChannelHandler {

//=====

//端在本机JVM中会被多个线程共用，因此需要使用volatile变量让所有线程在第一时间知道其是否处于可用状态  
//=====

// closing closed means the process is being closed and close is finished  
**private volatile boolean** closing;

**private volatile boolean** closed;

@Override  
**public boolean** isClosed() {  
 **return** closed;  
}

**public boolean** isClosing() {  
 **return** closing && !closed;  
}

@Override  
**public void** close() {  
 closed = **true**;  
}

@Override  
**public void** close(**int** timeout) {  
 close();  
}

//通讯端关闭需持续一段时间，等最终完成关闭会调用close()方法

@Override  
**public void** startClose() {  
 **if** (isClosed()) {  
 **return**;  
 }  
 closing = **true**;  
}

//=====

//发送通讯行为的事件在已关闭状态是禁用的

//=====

@Override  
**public void** connected(Channel ch) **throws** RemotingException {  
 **if** (closed) {  
 **return**;  
 }  
 handler.connected(ch);  
}

@Override  
**public void** sent(Channel ch, Object msg) **throws** RemotingException {  
 **if** (closed) {  
 **return**;  
 }  
 handler.sent(ch, msg);

JAVA

```
}

@Override
public void received(Channel ch, Object msg) throws RemotingException {
    if (closed) {
        return;
    }
    handler.received(ch, msg);
}

//=====
//感知断链和异常事件
//=====

@Override
public void disconnected(Channel ch) throws RemotingException {
    handler.disconnected(ch);
}

@Override
public void caught(Channel ch, Throwable ex) throws RemotingException {
    handler.caught(ch, ex);
}
}
```

## AbstractEnpoint

Dubbo中，参数的传递的信使始终是Url，负责通讯的端需要有合适的Codec2编解码器对对传输的数据进行编码解码。

```
public abstract class AbstractEndpoint extends AbstractPeer implements Resetable { JAVA

//=====
//该类中定义了如下3个参数，职责也主要是通过信使判断是否需要对他们进行重设
//分别对应的参数Key为：codec、timeout、connect.timeout
//其默认值分别对应telnet、1s、3s
//=====

    private Codec2 codec;

    private int timeout;

    private int connectTimeout;

    //构建初期Dubbo会根据传入的url设值，没有明确指定的情况下使用默认参数
    public AbstractEndpoint(URL url, ChannelHandler handler) {
        super(url, handler);
        this.codec = getChannelCodec(url);
        this.timeout = url.getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
        this.connectTimeout = url.getPositiveParameter(Constants.CONNECT_TIMEOUT_KEY,
    Constants.DEFAULT_CONNECT_TIMEOUT);
    }

    //使用Dubbo自身的SPI机制根据参数获取当前JVM中对应的Codec2实现
    protected static Codec2 getChannelCodec(URL url) {
        String codecName = url.getParameter(Constants.CODEC_KEY, "telnet");
        if (ExtensionLoader.getExtensionLoader(Codec2.class).hasExtension(codecName)) {
            return
        ExtensionLoader.getExtensionLoader(Codec2.class).getExtension(codecName);
        } else {
            return new CodecAdapter(ExtensionLoader.getExtensionLoader(Codec.class)
                .getExtension(codecName));
        }
    }

    @Override
    public void reset(URL url) {
        if (isClosed()) {
            throw new IllegalStateException("Failed to reset parameters "
                + url + ", cause: Channel closed. channel: " + getLocalAddress());
        }
        if (url.hasParameter(TIMEOUT_KEY)) {
            int t = url.getParameter(TIMEOUT_KEY, 0);
            if (t > 0) {
                this.timeout = t;
            }
        }
        if (url.hasParameter(Constants.CONNECT_TIMEOUT_KEY)) {
            int t = url.getParameter(Constants.CONNECT_TIMEOUT_KEY, 0);
            if (t > 0) {
                this.connectTimeout = t;
            }
        }
        if (url.hasParameter(Constants.CODEC_KEY)) {
            this.codec = getChannelCodec(url);
        }
    }
}
```

```
}
```

## AbstractChannel

前文已经提到，Channel是客户端和服务端通讯的信号通道，有着多对一的绑定关系。

**AbstractChannel** 作为抽象基类单独提炼出来，也仅仅是完成最基础的一部分特性，如下所示，但另外一方面而言，这个看起来可有可无的存在实际是又是必要的，类似Netty等的I/O框架都有直接定义名为Channel的接口，这和Dubbo 定义的Channel接口虽然不冲突，但在同一个类中出现，难免产生混淆。

```
public abstract class AbstractChannel extends AbstractPeer implements Channel {  
    public AbstractChannel(URL url, ChannelHandler handler) {  
        super(url, handler);  
    }  
  
    //子类实现该具体方法时一般要调用super.send()操作，确保正确继承父类定义的行为  
    @Override  
    public void send(Object message, boolean sent) throws RemotingException {  
        if (isClosed()) {  
            throw new RemotingException(this, "Failed to send message "  
                + (message == null ? "" : message.getClass().getName() + ":" +  
message  
                + ", cause: Channel closed. channel: " + getLocalAddress() + " -> "  
+ getRemoteAddress());  
        }  
    }  
    //该toString()方法仅仅用于告知通讯的双方IP地址  
    @Override  
    public String toString() {  
        return getLocalAddress() + " -> " + getRemoteAddress();  
    }  
}
```

## NettyChannel

顾名思义，NettyChannel的通道特性是委托给Netty实现的，调用其定义的Channel接口，也就是说二者存在的一一对应关系。微服务架构中，一个Client客户端往往需要连接多个其它第三方的Server服务端，也即同一个JVM中存在着多份这样的关系，因此NettyChannel中定义了如下一个线程安全的Map容器简单缓存实现， 注意它是全局的静态私有变量：

```
private static final ConcurrentHashMap<Channel, NettyChannel>  
CHANNEL_MAP = new ConcurrentHashMap<Channel, NettyChannel>();
```

为了更好的管理这种映射关系，NettyChannel的构造函数被设计成了私有的，需要调用对应的getOrAddChannel() 静态方法获得实例，同时静态方法只能在本Package中使用，这说明直接接触Netty的部分也被局限在一个小的范围，模块化边界更加清晰。

```

private NettyChannel(Channel channel, URL url, ChannelHandler handler) {
    super(url, handler);
    if (channel == null) {
        throw new IllegalArgumentException("netty channel == null;");
    }
    this.channel = channel;
}

//=====
//只有active状态的channel才会被装入到缓存，同时该状态下，是不允许脱离缓存，避免处于游离状态
//=====

static NettyChannel getOrAddChannel(Channel ch, URL url, ChannelHandler handler) {
    if (ch == null) {
        return null;
    }
    //根据Netty之Channel查找现存映射关系，若存在直接返回
    NettyChannel ret = CHANNEL_MAP.get(ch);
    if (ret == null) {

        NettyChannel nettyChannel = new NettyChannel(ch, url, handler);

        if (ch.isActive()) {//只有Channel处于激活有效状态，才执行下述代码

            //getOrAddChannel是一全局静态方法，存在并发问题，虽然上述发现并不存在对应关系
            //但此后依然可能加入了其映射关系，因此需要使用现场安全的putIfAbsent
            //如果存在直接返回原有NettyChannel值，否则返回新加入的值nettyChannel

            ret = CHANNEL_MAP.putIfAbsent(ch, nettyChannel);
        }

        //此前并不存在对应映射关系，直接返回新创建的值nettyChannel
        if (ret == null) {
            ret = nettyChannel;
        }
    }
    return ret;
}
static void removeChannelIfDisconnected(Channel ch) {
    if (ch != null && !ch.isActive()) {
        CHANNEL_MAP.remove(ch);
    }
}

```

## IMPORTANT

上述容易被人忽视的地方是，CHANNEL\_MAP实际上在任意时刻只会为同一个Client缓存一份 <Channel, NettyChannel> 的键值关系，每一次调用 getOrAddChannel(Channel, URL, ChannelHandler) 都会传入当前Client所持有的最新 channel变量 volatile类型。

JAVA

上文中提到Channel需要实现自己的本地属性存取函数，目的是为了跨函数栈帧获取到Dubbo通道本地的上下文值，具体实现如下：

```
private final Map<String, Object> attributes =
    new ConcurrentHashMap<String, Object>();

@Override
public boolean hasAttribute(String key) {
    return attributes.containsKey(key);
}

@Override
public Object getAttribute(String key) {
    return attributes.get(key);
}

@Override
public void setAttribute(String key, Object value) {
    // The null value is unallowed in the ConcurrentHashMap.
    if (value == null) {
        attributes.remove(key);
    } else {
        attributes.put(key, value);
    }
}

@Override
public void removeAttribute(String key) {
    attributes.remove(key);
}
```

通道使用完需要执行一些打扫战争的清理工作，依次执行如下4个动作：

1. 调用父类定义的 `close()` 方法，改变对应的volatile类型的状态值；
2. 如果channelNetty定义的那个已处于InActive状态，则从ConcurrentMap缓存中移除；
3. 清理所有Channel本地缓存的属性值；
4. 调用Netty的Channel的Close方法，将其通道功能关闭

```
public void close() {

    super.close();

    removeChannelIfDisconnected(channel);

    attributes.clear();

    channel.close();
}
```

最后便是通道的消息发送功能的实现，Netty支持异步I/O，因此可以通过参数告知是同步发送消息还是异步

```
public void send(Object message, boolean sent) throws RemotingException {  
    // whether the channel is closed  
    super.send(message, sent);  
  
    boolean success = true;  
    int timeout = 0;  
    try {  
        //通道消息写入最后必须执行flush操作，否则对端会一直处于IO等待状态  
        ChannelFuture future = channel.writeAndFlush(message);  
        if (sent) {  
            // wait timeout ms  
            timeout = getUrl().getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);  
            //如果在规定时间还未完成，便返回  
            success = future.await(timeout);  
        }  
        //future的特性是不仅缓存结果值，还会缓存异常（如果存在的话）  
        Throwable cause = future.cause();  
        if (cause != null) {  
            throw cause;  
        }  
    } catch (Throwable e) {  
        throw new RemotingException(this, "Failed to send message " + message + " to "  
+ getRemoteAddress() + ", cause: " + e.getMessage(), e);  
    }  
    if (!success) {  
        throw new RemotingException(this, "Failed to send message " + message + " to "  
+ getRemoteAddress()  
        + "in timeout(" + timeout + "ms) limit");  
    }  
}
```

## AbstractClient → NettyClient

AbstractClient采用模板模式定义实现了I/O通讯的中公共的行为，如下具体行为则由具体实现类针对特定I/O框架做进一步实现。

```
protected abstract void doOpen() throws Throwable;  
  
protected abstract void doClose() throws Throwable;  
  
protected abstract void doConnect() throws Throwable;  
  
protected abstract void doDisconnect() throws Throwable;  
  
protected abstract Channel getChannel();
```

上述的 `getChannel()` 方法是其它所有操作的基础，这和Dubbo中Consumer能和多个Server发生通讯有关。一个Consumer可以和多个Server保持通讯往来，同时一个Server绝大部分情况是会和多个Consumer发生联系的，Dubbo会为一个Consumer创建了多份Client，每一份Client仅仅关联唯一指定的Server，Client和Server使用通道建立连接和发生通讯，其间具有多对一的关系。另外Client和Channel是一对一的组合关系，因此Client的大部分生命周期行为都被委托给 Channel实现。

也就是说从Client的视觉来说，它和Server的关系是一对一的，但是Channel的生命周期于Client来说是短暂的，通过仔细阅读代码发现，Dubbo实际上最多只保持一个Channel处于开启状态，新的连接进来，老的就被close掉，并从NettyChannel定义的私有、全局静态且线程安全的变量 `CHANNEL_MAP` 中剔除，此外Dubbo利用了并发中的一些诸如`volatile`、`ReentrantLock`、`ConcurrentMap`的技巧来保证线程安全。

**NOTE**

由Netty创建的Channel每次连接会创建一份新的，Dubbo自己维护的NettyChannel与其生命周期基本是一样的，这还得回到Netty的hash值计算方式来，由源码可以看出，其Hash值的计算的唯一根据是Netty所创建的那份Channel。

```

/**
 * netty client bootstrap
 */
private static final NioEventLoopGroup nioEventLoopGroup =
    new NioEventLoopGroup(Constants.DEFAULT_IO_THREADS,
        new DefaultThreadFactory("NettyClientWorker", true));

private Bootstrap bootstrap;

//=====
//channel被申明成了volatile, 每一次调用doConnect()发生新的连接都会替换该值
//=====
/** 
 * current channel. Each successful invocation of {@link NettyClient#doConnect()} will
 * replace this with new channel and close old channel.
 * <b>volatile, please copy reference to use.</b>
 */
private volatile Channel channel;

@Override
protected org.apache.dubbo.remoting.Channel getChannel() {
    Channel c = channel;
    if (c == null || !c.isActive()) {
        return null;
    }

    //调用下述方法, 确保任何时刻都能获得client当前最新创建的Netty之Channel
    return NettyChannel.getOrAddChannel(c, getUrl(), this);
}

//=====
//client完成连接这个动作实际上是完成Client到Server通道的建立
//=====

@Override
protected void doConnect() throws Throwable {
    long start = System.currentTimeMillis();
    ChannelFuture future = bootstrap.connect(getConnectAddress());
    try {
        //等待连接完成
        boolean ret = future.awaitUninterruptibly(getConnectTimeout(), MILLISECONDS);

        if (ret && future.isSuccess()) {
            //在指定时间内成功获取到连接通道

            Channel newChannel = future.channel();
            try {
                //新的通道建立, 老的那份就会被移除

                // Close old channel
                // copy reference
                Channel oldChannel = NettyClient.this.channel;
                if (oldChannel != null) {

```

```

        try {
            if (logger.isInfoEnabled()) {
                logger.info("Close old netty channel " + oldChannel + " on
create new netty channel " + newChannel);
            }
            oldChannel.close();
        } finally {
            //将此前的channel自从CHANNEL_MAP中移除
            NettyChannel.removeChannelIfDisconnected(oldChannel);
        }
    }
} finally {

    if (NettyClient.this.isClosed()) {
        //Channel虽然成功获得连接，但等待连接这段时间内客户端已经关闭，这时新建立的通道
也需要关闭

```

```

        try {
            if (logger.isInfoEnabled()) {
                logger.info("Close new netty channel " + newChannel + ",

because the client closed.");
            }
            newChannel.close();
        } finally {
            NettyClient.this.channel = null;
            NettyChannel.removeChannelIfDisconnected(newChannel);
        }
    } else {
        //NettyClient中的这份channel变量是volatile类型的，更新对其它线程可见
        NettyClient.this.channel = newChannel;
    }
}
} else if (future.cause() != null) {
    throw new RemotingException(this, "client(url: " + getUrl() + ") failed to
connect to server "
        + getRemoteAddress() + ", error message is:" +
future.cause().getMessage(), future.cause());
} else {
    throw new RemotingException(this, "client(url: " + getUrl() + ") failed to
connect to server "
        + getRemoteAddress() + " client-side timeout "
        + getConnectTimeout() + "ms (elapsed: " +
(System.currentTimeMillis() - start) + "ms) from netty client "
        + NetUtils.getLocalHost() + " using dubbo version " +
Version.getVersion());
}
} finally {
    // just add new valid channel to NettyChannel's cache
    if (!isConnected()) {
        //future.cancel(true);
    }
}
}
}

@Override

```

```
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((channel == null) ? 0 : channel.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    NettyChannel other = (NettyChannel) obj;
    if (channel == null) {
        if (other.channel != null) {
            return false;
        }
    } else if (!channel.equals(other.channel)) {
        return false;
    }
    return true;
}
```

## 重入锁下的连接管理

一般而言，一个Dubbo应用程序，除了作为Consumer消费其它Server提供的服务外，也会作为Server向外提供服务。假设存在一种这样的场景，某个Dubbo应用 在凌晨期间会启动定时任务从其它Server同步数据，而白天则对外直接提供服务，作为Consumer的角色仅限于凌晨这段时间。这时其对应的Client被实例化后 会一直持续到任务同步完成后，然后结束其持续了数十分钟乃至几个小时的生命周期，尽管如此，它的生命周期依然会比每次发生数据传输时才建立的Channel通道 的长得多。在这相对比较漫长的生涯过程中，由于宿主应用是运行在多线程环境中的，这个只连接指定Server的Client会被作为一种共享资源产生竞态条件，因而 加锁操作是必须的，避免连接、断连、重连这几个操作被重复执行。

```

//=====
//这里锁被声明为可重入的
//=====
private final Lock connectLock = new ReentrantLock();

protected void connect() throws RemotingException {
    //执行业务前先获得锁
    connectLock.lock();

    try {
        //如果当前线程已经完成了到channel的连接绑定，则返回
        if (isConnected()) {
            return;
        }

        //委托具体实现类的实例完成连接操作
        doConnect();

        //上述操作之后，还没有获得连接，则报告异常信息
        if (!isConnected()) {
            throw new RemotingException(this, "Failed connect to server " +
getRemoteAddress() + " from " + getClass().getSimpleName() + " "
+ NetUtils.getLocalHost() + " using dubbo version " +
Version.getVersion()
+ ", cause: Connect wait timeout: " + getConnectTimeout() + "ms.");
        } else {
            if (logger.isInfoEnabled()) {
                logger.info("Succeed connect to server " + getRemoteAddress() + " from "
+ getClass().getSimpleName() + " "
+ NetUtils.getLocalHost() + " using dubbo version " +
Version.getVersion()
+ ", channel is " + this.getChannel());
            }
        }
    }

} catch (RemotingException e) {
    throw e;

} catch (Throwable e) {
    throw new RemotingException(this, "Failed connect to server " +
getRemoteAddress() + " from " + getClass().getSimpleName() + " "
+ NetUtils.getLocalHost() + " using dubbo version " +
Version.getVersion()
+ ", cause: " + e.getMessage(), e);

} finally {
    //finally模块中释放锁
    connectLock.unlock();
}
}

//=====
//断连时，先获得连接通道，关闭之，后续执行doDisconnect，委托具体实现类完成其他断连相关扫尾工作

```

```

//=====
public void disconnect() {
    connectLock.lock();
    try {
        try {
            Channel channel = getChannel();
            if (channel != null) {
                channel.close();
            }
        } catch (Throwable e) {
            logger.warn(e.getMessage(), e);
        }
        try {
            doDisconnect();
        } catch (Throwable e) {
            logger.warn(e.getMessage(), e);
        }
    } finally {
        connectLock.unlock();
    }
}

//=====
//重连只有在当前连接丢失的情况下才能再次进行，每次重连之前需要先执行disconnect操作，将此前的现场信息清除掉
//=====
@Override
public void reconnect() throws RemotingException {
    if (!isConnected()) {
        connectLock.lock();
        try {
            if (!isConnected()) {
                disconnect();
                connect();
            }
        } finally {
            connectLock.unlock();
        }
    }
}

```

## NOTE

可重入锁也即如果一个线程已经获得锁，再次尝试获取锁时会即刻被放行，锁计数值+1，嵌套调用的方法以与获取锁相反的顺序逐个释放锁，锁计数值依次-1，当计数值为0时，当前线程完成锁的释放，使得其它线程有机会获取到锁。重入锁避免了同一线程再次获取锁时会出现死锁或者获锁造成的等待时间消耗。

通常锁会和申明为**volatile**的变量结合使用，由其可见性保证当前线程能第一时间获知其值的变化。

## 获取channel发送消息

分布式的参与主机需要在网络的作用下才能发送通讯，这些主机所处环境甚至是异构的，掉线是一件很普遍的事，因而使用通道发送消息之前要检测当前连接是否已经断开，Dubbo允许配置Client在断连后进行重连，相关参数为“send.reconnect”。

```
private final boolean needReconnect;

public AbstractClient(URL url, ChannelHandler handler) throws RemotingException {
    super(url, handler);
    //send.reconnect参数告知在发送消息时是否需要做重连处理
    needReconnect = url.getParameter(Constants.SEND_RECONNECT_KEY, false);
    ...
}

@Override
public void send(Object message, boolean sent) throws RemotingException {
    if (needReconnect && !isConnected()) {
        connect();
    }

    //确保处于连接状态下再发送请求
    Channel channel = getChannel();
    //TODO Can the value returned by getChannel() be null? need improvement.
    if (channel == null || !channel.isConnected()) {
        throw new RemotingException(this, "message can not send, because channel is
closed . url:" + getUrl());
    }
    channel.send(message, sent);
}
```

## 资源竞争中的重连操作

在《定时轮算法及其实现》一文中的应用案例中有提到，Dubbo会专门分配一个线程利用定时轮周期性地完成重连操作，完成业务操作的关键代码正是这里提到的 `reconnect()`，如下述源码所示，如果检测到绑定Channel已经断连，或最近一次Channel的读取时间是否“> idleTimeout”，则执行重连操作。

```

/**
 * ReconnectTimerTask
 */
public class ReconnectTimerTask extends AbstractTimerTask {

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

    private final int idleTimeout;

    public ReconnectTimerTask(ChannelProvider channelProvider, Long
heartbeatTimeoutTick, int idleTimeout) {
        super(channelProvider, heartbeatTimeoutTick);
        this.idleTimeout = idleTimeout;
    }

    @Override
    protected void doTask(Channel channel) {
        try {
            Long lastRead = lastRead(channel);
            Long now = now();

            // Rely on reconnect timer to reconnect when AbstractClient.doConnect fails
            to init the connection
            if (!channel.isConnected()) {
                try {
                    logger.info("Initial connection to " + channel);
                    //Client继承实现了Channel接口，具体行为实现委托给了所引用的Channel
                    ((Client) channel).reconnect();
                } catch (Exception e) {
                    logger.error("Fail to connect to " + channel, e);
                }
                // check pong at client
            } else if (lastRead != null && now - lastRead > idleTimeout) {
                logger.warn("Reconnect to channel " + channel + ", because heartbeat
read idle time out: "
                        + idleTimeout + "ms");
                try {
                    ((Client) channel).reconnect();
                } catch (Exception e) {
                    logger.error(channel + "reconnect failed during idle time.", e);
                }
            }
        } catch (Throwable t) {
            logger.warn("Exception when reconnect to remote channel " +
channel.getRemoteAddress(), t);
        }
    }

    public class HeaderExchangeClient implements ExchangeClient → (Client, ExchangeChannel)
    {
        //执行周期任务的时间轮引擎
        private static final HashedWheelTimer IDLE_CHECK_TIMER = new HashedWheelTimer(
            new NamedThreadFactory("dubbo-client-idleCheck", true), 1,
        TimeUnit.SECONDS, TICKS_PER_WHEEL);
        ...
    }
}

```

```
private void startReconnectTask(URL url) {  
    if (shouldReconnect(url)) {  
        AbstractTimerTask.ChannelProvider cp = () ->  
    Collections.singletonList(HeaderExchangeClient.this);  
        int idleTimeout = getIdleTimeout(url);  
        long heartbeatTimeoutTick = calculateLeastDuration(idleTimeout);  
        this.reconnectTimerTask = new ReconnectTimerTask(cp, heartbeatTimeoutTick,  
idleTimeout);  
        //提交周期任务  
        IDLE_CHECK_TIMER.newTimeout(reconnectTimerTask, heartbeatTimeoutTick,  
TimeUnit.MILLISECONDS);  
    }  
}  
...  
}
```

## 委托绑定Channel实现的行为

AbstractClient实现了Client接口，而后者又融合了Endpoint、Channel、Resetable、IdleSensible这4个接口，如下述源码所示的特性实际上是委托给当前绑定的激活态Channel达成的，也就是包括Client本地属性存取在内的操作实际上是由绑定Channel完成的。

```
public Object getAttribute(String key) {
    Channel channel = getChannel();
    if (channel == null) {
        return null;
    }
    return channel.getAttribute(key);
}

public void setAttribute(String key, Object value) {
    Channel channel = getChannel();
    if (channel == null) {
        return;
    }
    channel.setAttribute(key, value);
}

public void removeAttribute(String key) {
    Channel channel = getChannel();
    if (channel == null) {
        return;
    }
    channel.removeAttribute(key);
}

public boolean hasAttribute(String key) {
    Channel channel = getChannel();
    if (channel == null) {
        return false;
    }
    return channel.hasAttribute(key);
}

@Override
public InetSocketAddress getRemoteAddress() {
    Channel channel = getChannel();
    if (channel == null) {
        return getUrl().toInetSocketAddress();
    }
    return channel.getRemoteAddress();
}

@Override
public InetSocketAddress getLocalAddress() {
    Channel channel = getChannel();
    if (channel == null) {
        return InetSocketAddress.createUnresolved(NetUtils.getLocalHost(), 0);
    }
    return channel.getLocalAddress();
}

@Override
public boolean isConnected() {
    Channel channel = getChannel();
    if (channel == null) {
        return false;
    }
    return channel.isConnected();
}
```

```
    }
    return channel.isConnected();
}
```

## Client的诞生和消亡

所涉及内容和Dubbo线程派发模型有着莫大的关系，线程池随Client产生而生产，随Client关闭而销毁，有关细节请参考《Dubbo之线程管理》，下述简要看看其实现。

### close

Close的基本步骤如下，如果调用 `close(int timeout)` 则会先调用线程池优雅终止方法 `ExecutorUtil.gracefulShutdown()`：

1. 调用父类 `AbstractPeer.close()` 标记 `closed` 为true;
2. 调用 `ExecutorUtil.shutdownNow(executor, 100)` 确保使用到的线程池被释放，相关执行任务被正常终止；
3. 调用 `AbstractClient.disconnect()` 关闭当前用于通信的Channel;
4. 调用 `AbstractClientXXXImpl.doClose()` 执行一些收尾工作；

```
@Override
public void close() {

    try {
        super.close();
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }

    try {
        if (executor != null) {
            ExecutorUtil.shutdownNow(executor, 100);
        }
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }

    try {
        disconnect();
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }

    try {
        doClose();
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }
}

@Override
public void close(int timeout) {
    ExecutorUtil.gracefulShutdown(executor, timeout);
    close();
}
```

## open

客户端Client到Server的连接在其实例化时就发生了，其基本为：1）先调用具体实现类的 `doOpen()` 方法准备必要的资源；2）调用 `connect()` 建立连接。步骤中的任意一步有错，便调用 `close()` 方法关闭当前Client。

```

public AbstractClient(URL url, ChannelHandler handler) throws RemotingException {
    ...
    try {
        doOpen();
    } catch (Throwable t) {
        close();
        throw new RemotingException(...);
    }
    try {
        // connect.
        connect();
        ... log
    } catch (RemotingException t) {
        if (url.getParameter(Constants.CHECK_KEY, true)) {
            close();
            throw t;
        } else {
            ... log
        }
    } catch (Throwable t) {
        close();
        throw new RemotingException(...);
    }
}

...
}

```

以Netty为例，`doOpen`实现就是负责构建Netty的启动器，设置好对应的环境参数，传入对应的ChannelHandler，由Netty在对应I/O事件驱动回调其相应方法。

## Netty之于Dubbo

本文中涉及到的网络通讯中间件，都只以Netty4为分析对象，关于Netty的原理部分的解析放在该章节，有利于在充分理解既有的Dubbo基础组件的基础之上 能够深入掌握Netty和Dubbo是如何协作的。

### 构建Netty启动器

#### NettyClient

于Client，Netty启动器构建是在其`doOpen()`实现中完成的：

1. 为当前JVM构建全局唯一用于Client处理网络I/O事件的NioEventLoopGroup线程池；
2. 传入所需环境参数，构建Bootstrap；
3. 通过调用`bootstrap.handler(new ChannelInitializer() {})`给对应的通道**pipeline**设置各种handler；
4. 在**pipeline**加入最重要的自定义 `NettyClientHandler <- ChannelDuplexHandler`，由其将网络I/O事件桥接到Dubbo；

```

public class NettyClient extends AbstractClient {

    //同一JVM中的所有Client共享同一I/O线程池
    private static final NioEventLoopGroup nioEventLoopGroup = new NioEventLoopGroup(
        Constants.DEFAULT_IO_THREADS, new DefaultThreadFactory("NettyClientWorker",
        true));

    private static final String SOCKS_PROXY_HOST = "socksProxyHost";

    private static final String SOCKS_PROXY_PORT = "socksProxyPort";

    private static final String DEFAULT_SOCKS_PROXY_PORT = "1080";

    private Bootstrap bootstrap;

    @Override
    protected void doOpen() throws Throwable {

        //最重要的一个handler, 用于将Netty的网络I/O事件桥接到Dubbo,
        //也即在Netty的I/O事件点触发对应ChannelHandler的事件
        final NettyClientHandler nettyClientHandler = new NettyClientHandler(getUrl(),
        this);

        //创建netty的客户端启动器, 设置环境参数
        bootstrap = new Bootstrap();
        bootstrap.group(nioEventLoopGroup)
            .option(ChannelOption.SO_KEEPALIVE, true)//保活
            .option(ChannelOption.TCP_NODELAY, true)//无延迟
            //使用默认的内存分配方式
            .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            //.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, getTimeout())
            .channel(NioSocketChannel.class);

        //设置超时时间, 最小3s, 其值来自AbstractEndpoint这个抽象类设置的connectTimeout值
        if (getConnectTimeout() < 3000) {
            bootstrap.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 3000);
        } else {
            bootstrap.option(ChannelOption.CONNECT_TIMEOUT_MILLIS,
            getConnectTimeout());
        }

        //ChannelInitializer用于对通道执行初始化操作, 用完会被从上下文中移除
        bootstrap.handler(new ChannelInitializer() {

            @Override
            protected void initChannel(Channel ch) throws Exception {
                //获取URL中传入的心跳周期时间, 键: heartbeat, 默认值: 60 * 1000(一分钟)
                int heartbeatInterval = UrlUtils.getHeartbeat(getUrl());

                //由Dubbo实现的编解码适配器
                NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(),
                NettyClient.this);

                ch.pipeline()//.addLast("logging",new
                LoggingHandler(LogLevel.INFO))//for debug
            }
        });
    }
}

```

```

//增加用于编解码的handler: 二进制数据 ↔ Java Object对象
.addLast("decoder", adapter.getDecoder())
.addLast("encoder", adapter.getEncoder())

//增设空闲处理handler, 当有段时间没有执行I/O读写事件时会执行心跳处理
.addLast("client-idle-handler", new
IdleStateHandler(heartbeatInterval, 0, 0, MILLISECONDS))

.addLast("handler", nettyClientHandler);

//使用代理访问网络
//Socks5让有权限的用户可以穿过过防火墙的限制, 使得高权限用户可以访问外部资源
String socksProxyHost = ConfigUtils.getProperty(SOCKS_PROXY_HOST);
if(socksProxyHost != null) {
    int socksProxyPort = Integer.parseInt(ConfigUtils.getProperty(
        SOCKS_PROXY_PORT, DEFAULT_SOCKS_PROXY_PORT));
    Socks5ProxyHandler socks5ProxyHandler = new Socks5ProxyHandler(
        new InetSocketAddress(socksProxyHost, socksProxyPort));
    ch.pipeline().addFirst(socks5ProxyHandler);
}
}

});

...
}

```

## NettyServer

在《Dubbo线程管理》对Netty到Dubbo的线程模型已经有关比较详述的内容，为更加准确的理解Netty Server的启动器初始化过程，特呈现下图，由其可知：

一个Netty Server包含着如下几个关键组件：1) Boss Group；2) Work Group；3) Pipeline；4) NioEventGroup；5) Selector；6) ChannelHandler

其过程如下：

1. 被称为Boss Group的线程池使用单个线程NioEventGroup，其中的Selector负责select来自Client的连接I/O请求，成功建立连接后会为Client构建一个通讯通道Channel
2. 随后Channel会被注册到某个负责select 读写I/O请求的Selector上，该Selector位于被称为Work Group的线程池的某个NioEventGroup线程上；
3. 如果发现有读写I/O就绪事件，Netty内核会将事件传递到绑定当前Channel的Pipeline上，其中的ChannelHandler会根据自身特性挨个处理事件回调；

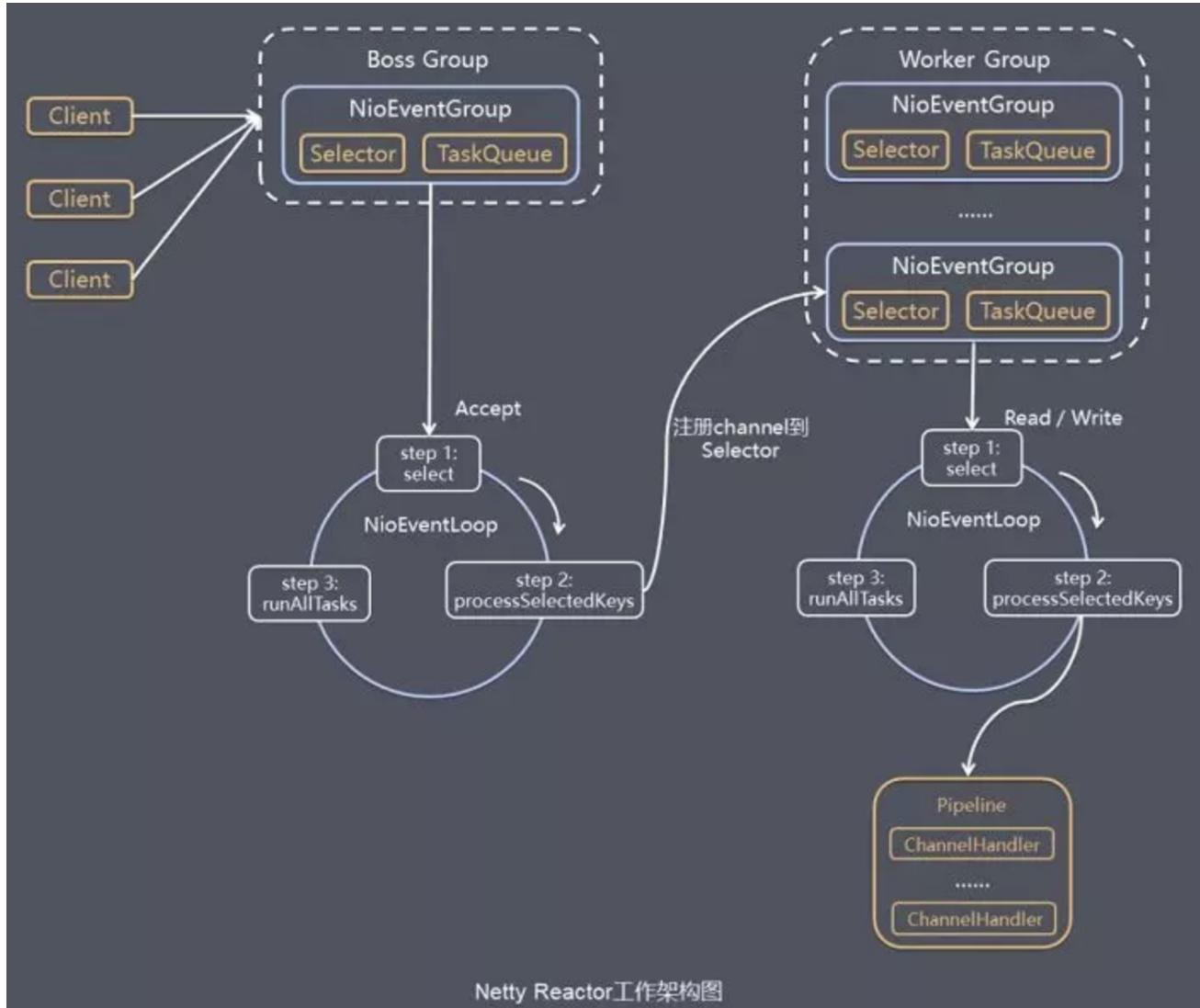


图: Netty 服务端架构图

其实过程和Netty Client基本类似，尤其是配置Handler处理其信息的编解码过程。只不过服务端会有多个Client接入，

```
public class NettyServer extends AbstractServer implements Server {
    /**
     * the cache for alive worker channel.
     * <ip:port, dubbo channel>
     */
    private Map<String, Channel> channels;
    /**
     * netty server bootstrap.
     */
    private ServerBootstrap bootstrap;
    /**
     * the boss channel that receive connections and dispatch these to worker channel.
     */
    private io.netty.channel.Channel channel;

    private EventLoopGroup bossGroup;
    private EventLoopGroup workerGroup;
    /**
     * Init and start netty server
     *
     * @throws Throwable
     */
    @Override
    protected void doOpen() throws Throwable {
        bootstrap = new ServerBootstrap();

        bossGroup = new NioEventLoopGroup(1, new
DefaultThreadFactory("NettyServerBoss", true));
        workerGroup = new
NioEventLoopGroup(getUrl().getPositiveParameter(IO_THREADS_KEY,
Constants.DEFAULT_IO_THREADS),
                new DefaultThreadFactory("NettyServerWorker", true));

        final NettyServerHandler nettyServerHandler = new NettyServerHandler(getUrl(),
this);
        channels = nettyServerHandler.getChannels();

        bootstrap.group(bossGroup, workerGroup)
            .channel(NioServerSocketChannel.class)
            .childOption(ChannelOption.TCP_NODELAY, Boolean.TRUE)
            .childOption(ChannelOption.SO_REUSEADDR, Boolean.TRUE)
            .childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            .childHandler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws Exception {
                    // FIXME: should we use getTimeout()?
                    int idleTimeout = UrlUtils.getIdleTimeout(getUrl());
                    NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(),
getUrl(), NettyServer.this);
                    ch.pipeline()//.addLast("logging",new
LoggingHandler(LogLevel.INFO))//for debug
                        .addLast("decoder", adapter.getDecoder())
                        .addLast("encoder", adapter.getEncoder())
                        .addLast("server-idle-handler", new IdleStateHandler(0,
0, idleTimeout, MILLISECONDS))
                        .addLast("handler", nettyServerHandler);
                }
            });
    }
}
```

JAVA

```

        }

    });

    //获取专门负责建立连接的通道
    // bind
    ChannelFuture channelFuture = bootstrap.bind(getBindAddress());
    //等待future直到变为done状态，也即等到负责I/O连接的这个通道Channel已经生成
    channelFuture.syncUninterruptibly();
    channel = channelFuture.channel();

}

}

```

上述源码所表示的Netty Server Bootstrap构建过程可以用如下更为直观的时序图表示：

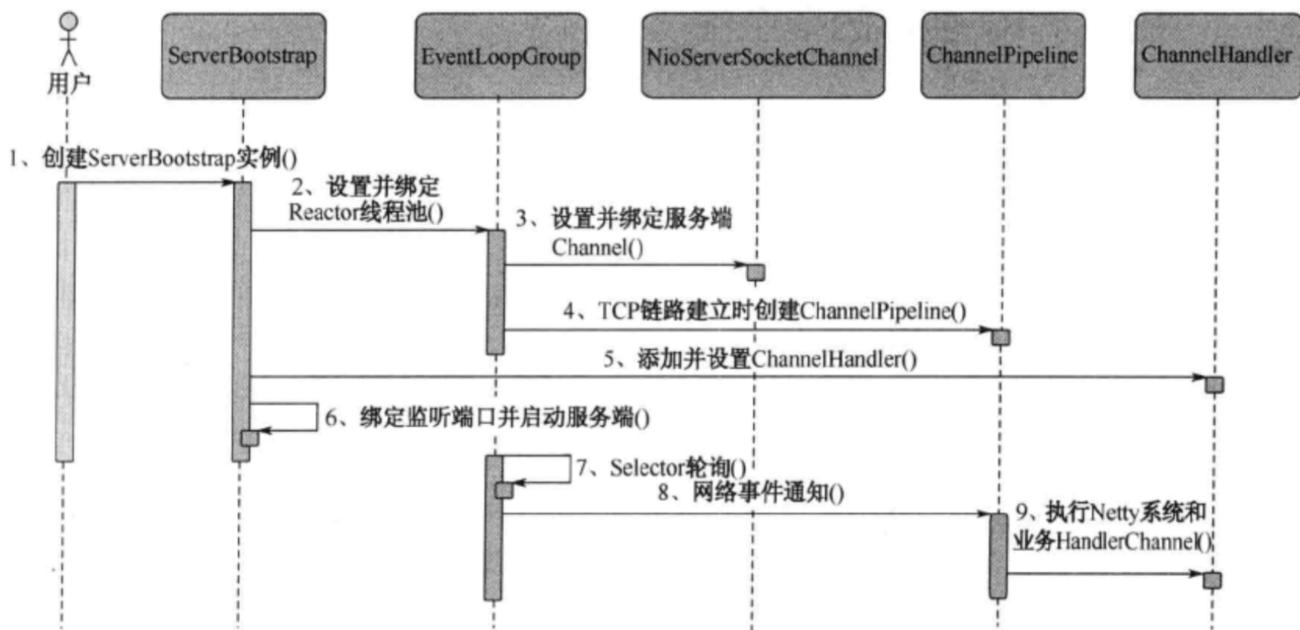


图 13-1 Netty 服务端创建时序图

图：*Netty Bootstrap构建时序图*

## 回调网络I/O事件

文中反复提及，Dubbo中有关网络行为是委托给类似Netty等的第三方中间件完成的，其**ChannelHandler**实现是供他们回调的，也是Dubbo网络能力增强的切入点。

对netty比较熟悉的攻城狮看到上面这个**ChannelHandler**会瞬间似曾相识，在Netty中确有一个名字一模一样的接口，用于处理 I/O 事件或拦截 I/O 操作。

一个使用了Netty作为通讯的App，最靠近底层负责网络通讯的Netty，将App向对端write写出的操作回调看做Outbound出站事件，反之从对端read读入的操作的回调则被看做是Inbound入站事件。Netty根据这个将网络I/O事件分为出站和入站两种，分别由**ChannelHandler**的扩展接口**ChannelInboundHandler**和**ChannelOutboundHandler**的实现处理。Netty的内核I/O线程专门负责处理具体的I/O，I/O事件就绪则回调业务层相应接口实现，也即其I/O具体处理过程于App业务层来说是无感的，它只能感知到的是I/O回调事件，由回调实现业务所需，这正是响应式编程的精髓所在。

调用Channel或ChannelHandlerContext的I/O请求驱动着Netty触发着本地Outbound事件，另一方面，经过网络信号传输，又间接地驱动着对端App的Netty 触发着其本地Inbound事件。netty在回调事件事件时会将其作为 **ChannelHandler** 的当前上下文信息持有者参数传入，由于一条Pipeline链只对应唯一的Channel，且Pipeline链的生命周期只局限于Netty的某个特定work线程中，因此就当前线程而言，不会存在Channel的资源争用问题。

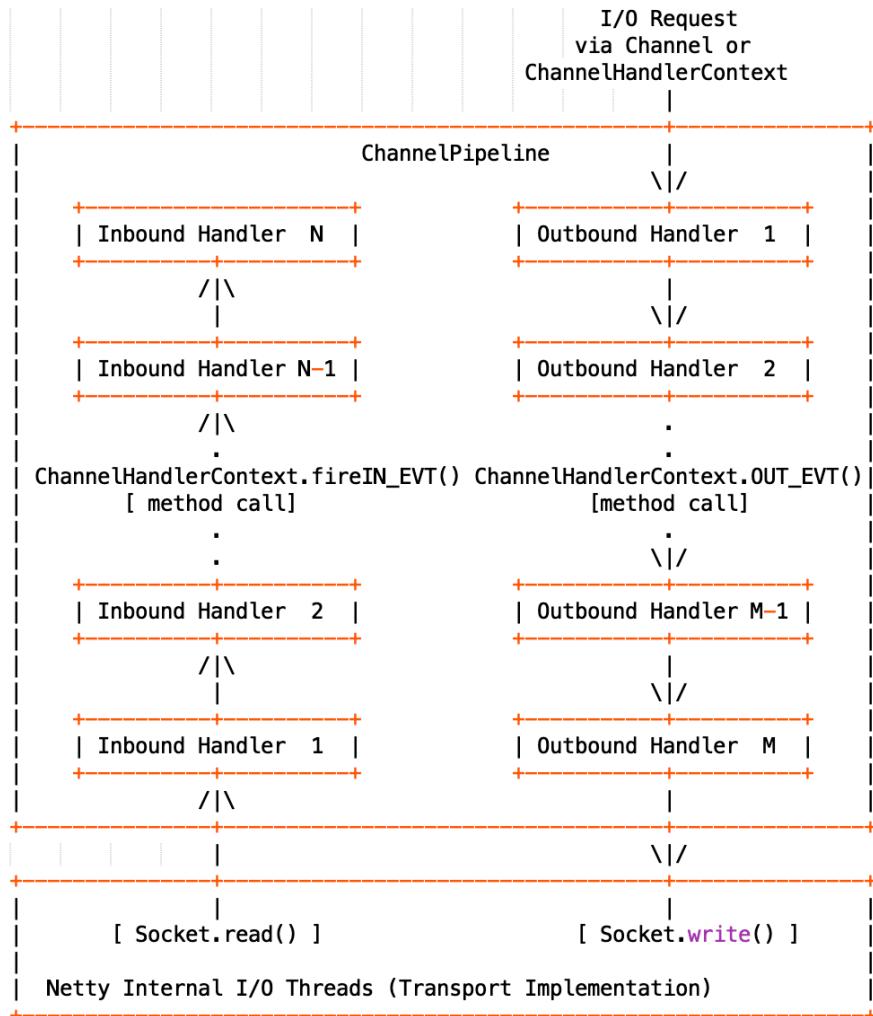


图: Pipeline出入站

每一个**ChannelHandler**实例对象会被绑定到一个叫做 **ChannelHandlerContext** 的对象中，多个这样的对象节点串联一起构成一条叫做**ChannelPipeline**的双 向链表，**Channel**生成时就会对应产生这样一条Pipeline。具体执行时，根据当前的出入站方向，**ChannelHandlerContext** 依次负责从链表的头部或者尾部开始顺序找到下一个最近节点并回调绑定在它身上的**ChannelHandler**，与方向相反的会被自动忽略。

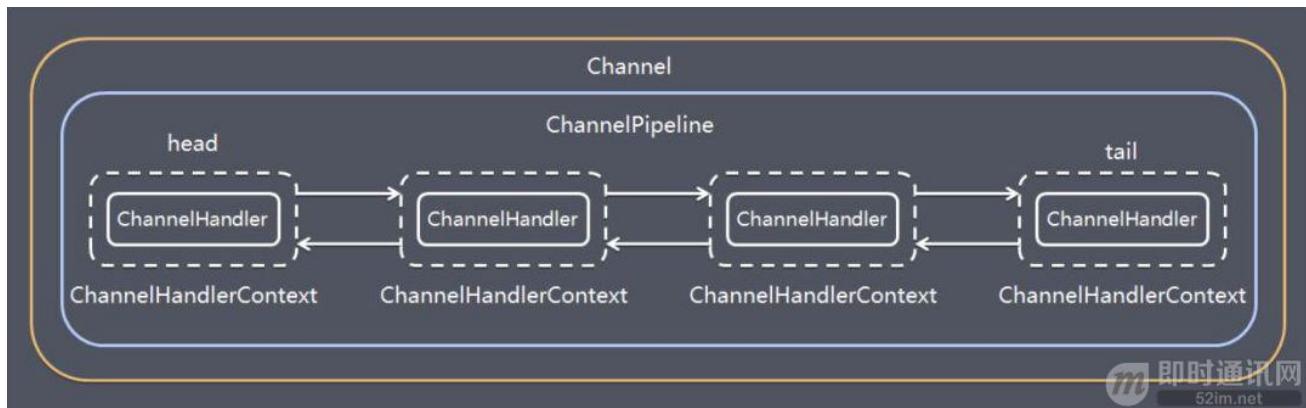


图: Pipeline 链表

为便于开发出入站I/O事件处理器，Netty提供了如下几个适配器类：

1. `ChannelInboundHandlerAdapter` 用于处理入站 I/O 事件。
2. `ChannelOutboundHandlerAdapter` 用于处理出站 I/O 操作。
3. `ChannelDuplexHandler` 用于处理入站和出站事件。

下述章节中的**NettyClientHandler**和**NettyServerHandler**便是扩展**ChannelDuplexHandler**实现的。出站事件是APP主动唤起的，因此netty中 实现主要是**ChannelInboundHandler**的回调业务。

### NettyClientHandler and NettyServerHandler

**ChannelInboundHandler**和**ChannelOutboundHandler**所表示的出入站事件和Dubbo中的**ChannelHandler**有着如下的一一对应关系：

1. **channelActive** → `connected`
2. **channelInactive** → `disconnected`
3. **channelRead** → `received`
4. **exceptionCaught** → `caught`
5. **write** → `sent`

入站读事件 `channelActive`、`channelInactive`、`channelRead`、`exceptionCaught`

如下源码所示的几个回调事件，两个handler的实现几乎一样，不同的是**NettyServerHandler**定义了一个 `<ip:port, dubbo channel>` 键值对Map， 用于缓存当前处于活跃态的 NettyChannel：

1. 首先通过ctx获取到对应的Netty方Channel；
2. 再以它作为参数调用 `getOrAddChannel` 得到对应的 NettyChannel 实例；
3. 随后调用Dubbo定义的 `ChannelHandler` 的对应事件回调方法；

4. 最后在finally快中调用 NettyChannel.removeChannelIfDisconnected(channel) , 确保一旦Channel失活，便从缓存中移除；

```
java  
@Override  
public void channelActive(ChannelHandlerContext ctx) throws Exception {  
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);  
    try {  
        //only for NettyServerHandler~~~~~start  
        if (channel != null) {  
            channels.put(NetUtils.toAddressString((InetSocketAddress)  
ctx.channel().remoteAddress()), channel);  
        }  
        //only for NettyServerHandler~~~~~ end  
  
        handler.connected(channel);  
    } finally {  
        NettyChannel.removeChannelIfDisconnected(ctx.channel());  
    }  
}  
  
@Override  
public void channelInactive(ChannelHandlerContext ctx) throws Exception {  
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);  
    try {  
        //only for NettyServerHandler~~~~~start  
        channels.remove(NetUtils.toAddressString((InetSocketAddress)  
ctx.channel().remoteAddress()));  
        //only for NettyServerHandler~~~~~ end  
  
        handler.disconnected(channel);  
    } finally {  
        NettyChannel.removeChannelIfDisconnected(ctx.channel());  
    }  
}  
  
@Override  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);  
    try {  
        handler.received(channel, msg);  
    } finally {  
        NettyChannel.removeChannelIfDisconnected(ctx.channel());  
    }  
}  
  
@Override  
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  
throws Exception {  
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);  
    try {  
        handler.caught(channel, cause);  
    } finally {  
        NettyChannel.removeChannelIfDisconnected(ctx.channel());  
    }  
}
```

## 出站写事件 write

稍微比较特殊点的操作是涉及重写**ChannelOutboundHandler**的 public void `write(ChannelHandlerContext, Object, ChannelPromise)` 方法。上文已经提到，出站事件是由当前App主动唤起的，Netty内核对于I/O处理的最终情况也是通过回调告知App的。回调中的 `ChannelPromise` 参数 用于进一步回调获得最终操作结果，其结果如下示意图：

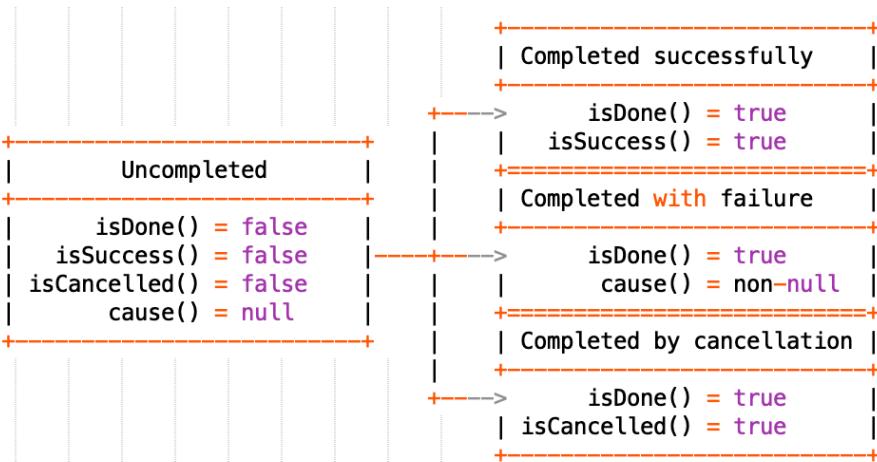


图: Netty Future Result

如下源码，Client端处理成功则回调 `ChannelHandler.sent()`，否则回调 `ChannelHandler.received()`。服务端则相对很简化，Dubbo认为消息发送出去收到write回调，便即可认为成功。

```
public class NettyClientHandler extends ChannelDuplexHandler {
```

```
...
```

```
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
throws Exception {
    super.write(ctx, msg, promise);
    final NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url,
handler);
    final boolean isRequest = msg instanceof Request;

    // We add listeners to make sure our out bound event is correct.
    // If our out bound event has an error (in most cases the encoder fails),
    // we need to have the request return directly instead of blocking the invoke
process.
    promise.addListener(future -> {
        try {
            if (future.isSuccess()) {
                // if our future is success, mark the future to sent.
                handler.sent(channel, msg);
                return;
            }
        }

        Throwable t = future.cause();
        if (t != null && isRequest) {
            Request request = (Request) msg;
            Response response = buildErrorResponse(request, t);
            handler.received(channel, response);
        }
    } finally {
        NettyChannel.removeChannelIfDisconnected(ctx.channel());
    }
});
```

```
}
```

```
...
```

```
public class NettyServerHandler extends ChannelDuplexHandler {
```

```
...
```

```
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
throws Exception {
    super.write(ctx, msg, promise);
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url,
handler);
    try {
        handler.sent(channel, msg);
    } finally {
        NettyChannel.removeChannelIfDisconnected(ctx.channel());
    }
}
```

JAVA

```
...  
}
```

## 空闲事件 userEventTriggered

“占着茅坑不拉屎的行为”在哪都有点人神共愤，计算资源尤为宝贵的服务器更是如此，如果相应连入的客户端通道Channel一段时间没有发生过读写操作，Dubbo 会一剑封喉，直接将Channel关闭掉，这有效的避免了因为客户端对应实例因为宕机等原因依然为期保持Channel产生的附加资源浪费。封喉后如果客户端需和服务通讯，则需再次做连接处理。而客户端这边，只要自身还处于激活状态，就一直想和服务端保持一个连接状态，也就是常说的长连接，Netty中可以利用回调方法 `public void userEventTriggered(ChannelHandlerContext, Object)` 实现，具体如下代码所示：

```

public class NettyServerHandler extends ChannelDuplexHandler {
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception {
        // server will close channel when server don't receive any heartbeat from
client until timeout.
        if (evt instanceof IdleStateEvent) {
            NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url,
handler);
            try {
                logger.info("IdleStateEvent triggered, close channel " + channel);
                channel.close();
            } finally {
                NettyChannel.removeChannelIfDisconnected(ctx.channel());
            }
        }
        super.userEventTriggered(ctx, evt);
    }
}

public class NettyClientHandler extends ChannelDuplexHandler {
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception {
        // send heartbeat when read idle.
        if (evt instanceof IdleStateEvent) {
            try {
                NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url,
handler);
                if (logger.isDebugEnabled()) {
                    logger.debug("IdleStateEvent triggered, send heartbeat to channel "
+ channel);
                }
                //构建心跳包
                Request req = new Request();
                req.setVersion(Version.getProtocolVersion());
                req.setTwoWay(true);
                req.setEvent(Request.HEARTBEAT_EVENT);

                //发送心跳请求
                channel.send(req);
            } finally {
                NettyChannel.removeChannelIfDisconnected(ctx.channel());
            }
        } else {
            super.userEventTriggered(ctx, evt);
        }
    }
}

```

注：上述操作依然会调用 `removeChannelIfDisconnected()` #检查当前对应Netty的Channel是否失活，如果是，便移除之。

## Transporter

上文已经知道，Dubbo使用netty等第三方网络I/O中间件构建传输层，利用端口号建立端到端的通讯连接，屏蔽掉了下层的具体细节。于开发而言，实际上对应的其实是封装了TCP/IP的Socket，服务端需要绑定到一个固定的端口<sub>对应某进程的编号</sub>接受来自客户端的连接，而客户端则需要连接到服务端的某个端口以完成业务请求，如下**Transporter**是Dubbo提供的，能同时用于服务端绑定端口和客户端发起端口连接的可扩展接口。

```
@SPI("netty")
public interface Transporter {

    //Bind a server.
    @Adaptive({Constants.SERVER_KEY, Constants.TRANSPORTER_KEY})
    Server bind(URL url, ChannelHandler handler) throws RemotingException;

    //Connect to a server.
    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})
    Client connect(URL url, ChannelHandler handler) throws RemotingException;
}
```

对应实现也很简单，实际上就是相应构建NettyServer和NettyClient的实例。

```
public class NettyTransporter implements Transporter {

    public static final String NAME = "netty";

    @Override
    public Server bind(URL url, ChannelHandler listener) throws RemotingException {
        return new NettyServer(url, listener);
    }

    @Override
    public Client connect(URL url, ChannelHandler listener) throws RemotingException {
        return new NettyClient(url, listener);
    }
}
```

其实例构建则是经Transporters利用Dubbo SPI加载所配置Transporter实现，如下细节：

```
public class Transporters {

    static {
        // check duplicate jar package
        Version.checkDuplicate(Transporters.class);
        Version.checkDuplicate(RemotingException.class);
    }

    private Transporters() {
    }

    public static Server bind(String url, ChannelHandler... handler) throws
RemotingException {
        return bind(URL.valueOf(url), handler);
    }

    public static Server bind(URL url, ChannelHandler... handlers) throws
RemotingException {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        if (handlers == null || handlers.length == 0) {
            throw new IllegalArgumentException("handlers == null");
        }
        ChannelHandler handler;
        if (handlers.length == 1) {
            handler = handlers[0];
        } else {
            handler = new ChannelHandlerDispatcher(handlers);
        }
        return getTransporter().bind(url, handler);
    }

    public static Client connect(String url, ChannelHandler... handler) throws
RemotingException {
        return connect(URL.valueOf(url), handler);
    }

    public static Client connect(URL url, ChannelHandler... handlers) throws
RemotingException {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        ChannelHandler handler;
        if (handlers == null || handlers.length == 0) {
            //回调事件使用空实现
            handler = new ChannelHandlerAdapter();
        } else if (handlers.length == 1) {
            //单个ChannelHandler无需包装
            handler = handlers[0];
        } else {
            //将多个handler包装，由包装类在事件回调是逐个回调
            handler = new ChannelHandlerDispatcher(handlers);
        }
        return getTransporter().connect(url, handler);
    }
}
```

JAVA

```
//利用SPI机制获取锁配置的Transporter扩展实现
public static Transporter getTransporter() {
    return
ExtensionLoader.getExtensionLoader(Transporter.class).getAdaptiveExtension();
}

}
```

## 通道监听者派发器 ChannelHandlerDispatcher

上文中 `ChannelHandler` 反复出现，非常高频，它是理解整个Dubbo网络传输层的关键，使用装饰者模式实现，利用组合模式将业务实现真实的承载主体作为参数在其装饰者实例化时传入，装饰者们层层加码，外层完成对里层的功能特性的增强，Dubbo利用该机制完成了Server端、Client端、网络I/O中间件通道 Channel适配、线程派发、解码、心跳等各种特性的实现。

文中 `ChannelHandler` 总是和 网络I/O事件 成套出现，原因是前者实际上就是通道Channel的监听器，监听着通道的网络I/O事件。由于 业务实现真实的承载主体是在装饰器类实例化时作为构造器的参数传入的，这就也有了同时提供多份 `ChannelHandler` 被装饰者实现的可能性，上文中的 `Transporters` 也佐证了这一点。

实现原理很简单，可以认为它是装饰器的变种实现，不同的是，它组合进了多个被装饰者，装饰者被调用时，所有被装饰者的同名方法会挨个被调用。当然，被装饰者们需要用一个集合容器维护，这又让其具备了另外一个特性，就是可以在运行期间动态的添加或者移除被装饰者。

因此 `ChannelHandlerDispatcher` 会有类似如下模板代码：

```
public void XXX(Channel channel) {
    for (ChannelHandler listener : channelHandlers) {
        try {
            listener.XXX(channel);
        } catch (Throwable t) {
            logger.error(t.getMessage(), t);
        }
    }
}
```

JAVA

具体实现上，Dubbo使用了 `CopyOnWriteArrayList` 读写分离、支持并发的容器，如下源码：

```

public class ChannelHandlerDispatcher implements ChannelHandler {

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

    private final Collection<ChannelHandler> channelHandlers =
        new CopyOnWriteArrayList<ChannelHandler>();

    public ChannelHandlerDispatcher() {
    }

    public ChannelHandlerDispatcher(ChannelHandler... handlers) {
        this(handlers == null ? null : Arrays.asList(handlers));
    }

    public ChannelHandlerDispatcher(Collection<ChannelHandler> handlers) {
        if (CollectionUtils.isNotEmpty(handlers)) {
            this.channelHandlers.addAll(handlers);
        }
    }

    public Collection<ChannelHandler> getChannelHandlers() {
        return channelHandlers;
    }

    public ChannelHandlerDispatcher addChannelHandler(ChannelHandler handler) {
        this.channelHandlers.add(handler);
        return this;
    }

    public ChannelHandlerDispatcher removeChannelHandler(ChannelHandler handler) {
        this.channelHandlers.remove(handler);
        return this;
    }

    . . . //其它像上述模板实现的I/O回调方法
}

```

JAVA

## NOTE

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器，适用于读多写少的并发场景。

注：派发器 本质上是将某个行为委托给同一个接口的多个实例可能来自多个实现类来完成，可以根据具体的策略将他们统一平等看待，如上文 `ChannelHandlerDispatcher`，也可以有其它的策略，根据业务特性需求选用其中的一到多个。

---

完结

# Dubbo之线程管理

## 模型由来

Java线程和内核线程的1对1的关系使得其产生、销毁、切换都会耗费大量的CPU时间，每一个线程也最少占据1M的内存资源，这些都限制着Java不可能像类似 GoLang这类采用协程机制的编程语言可以任意创建执行任务用的轻量级线程。同时由于受到JVM实现方式的限制，其I/O的实现也还尚未完全逼近操作系统内核的 异步I/O。

### NOTE

“阻塞I/O、非阻塞I/O、I/O复用（**select**和**poll**）、信号驱动I/O（**SIGIO**）、异步I/O”这5种I/O模型，效率依次递增。

因而线程和I/O是Java开发中始终不可逃避的两个中心话题，可以说近些年Java生态的发展，始终没有离开对二者进行的优化设计。Netty正是巧妙的使用了 多路复用机制，使得业务线程不必阻塞等待在 I/O操作上，单位时间可以获得更大的吞吐量。隔壁JS社区NodeJs异步编程的高速发展更是直接刺激着Vertx全异步 编程框架的诞生，后者则直接是由Netty发展起来的。实际上单独避开I/O谈线程或者反过来都是不妥的，I/O模型只是尽量在设法降低业务线程的阻塞等待耗时，让其有机会利用这段 CPU时间去执行其它任务，反之线程模型则是讲究如何更好地利用好优化的I/O模型，因而二者相辅相成，目标都是在尽最大限度的压榨CPU等 硬件资源，提升系统性能。

## Dubbo方案——Dispatcher 派发模型

白话一点说，线程模型需要处理的面对以下几个问题：

1. I/O速度相比CPU，地上一年，天上一天；
2. 线程是一种昂贵的资源，创建和销毁耗费都巨大；
3. 线程池中的线程调度也是个比较耗时的过程；

Dubbo官网中针对这些不同的场景处理有如下的一段描述：



1. 如果事件处理的逻辑能迅速完成，并且不会发起新的 IO 请求，比如只是在内存中记个标识，则直接在 IO 线程上处理更快，因为减少了线程池调度。
2. 但如果事件处理逻辑较慢，或者需要发起新的 IO 请求，比如需要查询数据库，则必须派发到线程池，否则 IO 线程阻塞，将导致不能接收其它请求。

为了解决这些问题，Dubbo提出了如下几种 **Dispatcher** 派发模型

- “
  - 1. **all** 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
  - 2. **direct** 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
  - 3. **message** 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
  - 4. **execution** 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
  - 5. **connection** 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

同时还提了如下4中类型的**ThreadPool**

- “
  - 1. **fixed** 固定大小线程池，启动时建立线程，不关闭，一直持有。（缺省）
  - 2. **cached** 缓存线程池，空闲一分钟自动删除，需要时重建。
  - 3. **limited** 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。
  - 4. **eager** 优先创建Worker线程池。在任务数量大于corePoolSize但是小于maximumPoolSize时，优先创建Worker来处理任务。当任务数量大于maximumPoolSize时，将任务放入阻塞队列中。阻塞队列充满时抛出RejectedExecutionException。（相比于cached:cached在任务数量超过maximumPoolSize时直接抛出异常而不是将任务放入阻塞队列）

## Dispatcher 原理

Dubbo本身的大部分职责是负责RPC通讯，I/O处理占据了大量内容。由上文得知从网络I/O切换到业务处理这个过程中，需要平衡好线程和I/O间的关系——1) 线程间切换是个耗时的动作；2) 任何时候都不应该造成I/O线程阻塞。

上面反复提及的网络I/O实质是网络Socket通讯，Dubbo将该部分委托给诸如Grizzly、Mina、Netty等的网络I/O中间件，由他们处理具体的网络I/O，Dubbo只负责响应对应的5种网络I/O事件，分别是**connected**、**sent**、**received**、**disconnected**、**caught**，其中**sent**、**received**分别对应于RPC的请求和响应，是直接与业务方接入接出相关的。毋庸置疑，为保证吞吐量这些中间件已经采用了优化过的线程模型，也就是使用多路复用I/O，有专门的线程负责网络连接，由另外的work线程池负责处理读写I/O操作——从 SocketChannel 中读取报文或向 SocketChannel 写入报文。以下图所示的Netty服务端为例，bossGroup这个线程池中的NioEventLoop线程是专门负责Accept来自Client发起的请求，连接建立成功后，会为Client生成一个对应的 Channel 信息通道NioSocketChannel，并随后将其注册到workerGroup线程池中某个NioEventLoop线上一般被称为work线程，后者会不断的轮询注册到自身的NioSocketChannel，检测是否有读写事件准备好，若有，调用对应的ChannelHandler进行处理。

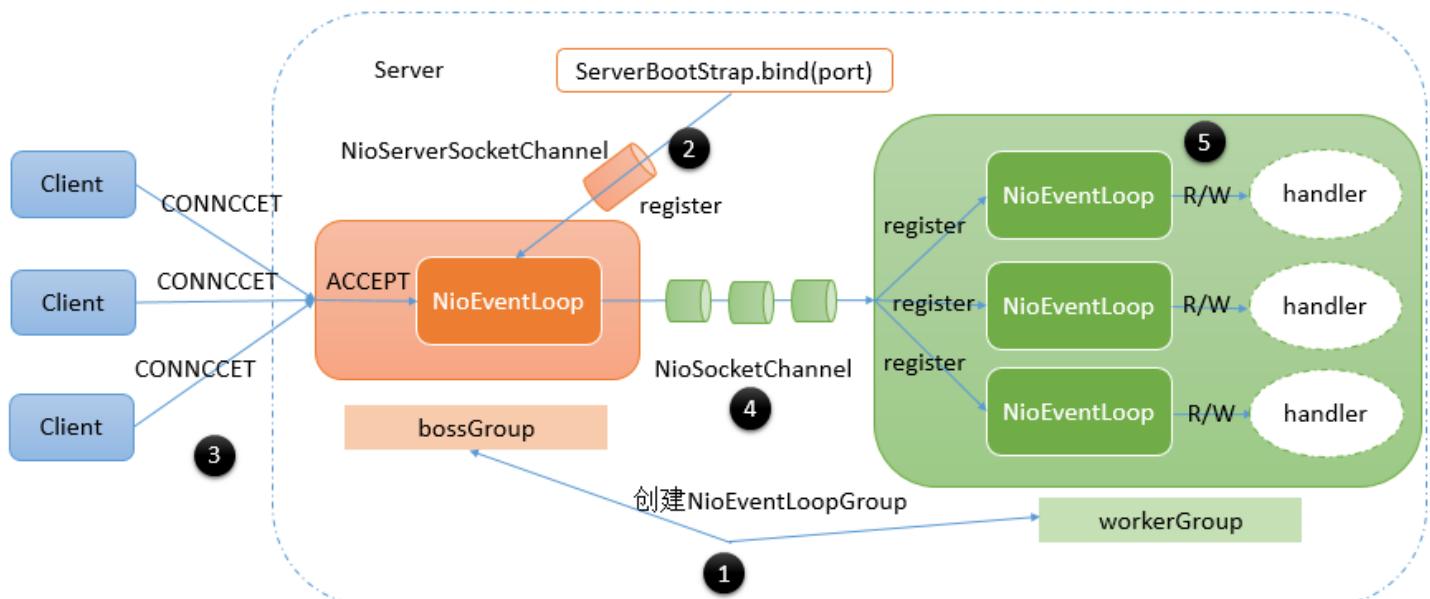


图 1: Netty 线程模型

但是上图所表达的这个模型依然是有欠缺的，work线程负责调度执行的ChannelHandler，是执行业务处理的处所，但也肩负着网络I/O的读写处理，于CPU密集型耗时任务而言，会阻塞work线程，严重影响Netty对Socket的处理速度，导致响应不及时，因而Netty建议另外使用专门的业务线程池异步执行耗时任务。[《京东的 Netty 实践，京麦 TCP 网关长连接容器架构》](#)

(<https://www.infoq.cn/article/jd-netty>)一文中提到使用Netty实现TCP网关，除了上文中提到bossGroup和workerGroup这两个线程池，还多出一个executorGroup线程池。关于他们的用途，文中描述如下：

**“BossGroup**用于接收客户端的TCP连接，**WorkerGroup**用于处理 I/O、执行系统 Task 和定时任务，**ExecutorGroup**用于处理网关业务加解密、限流、路由，及将请求转发给后端的抓取服务等业务操作。

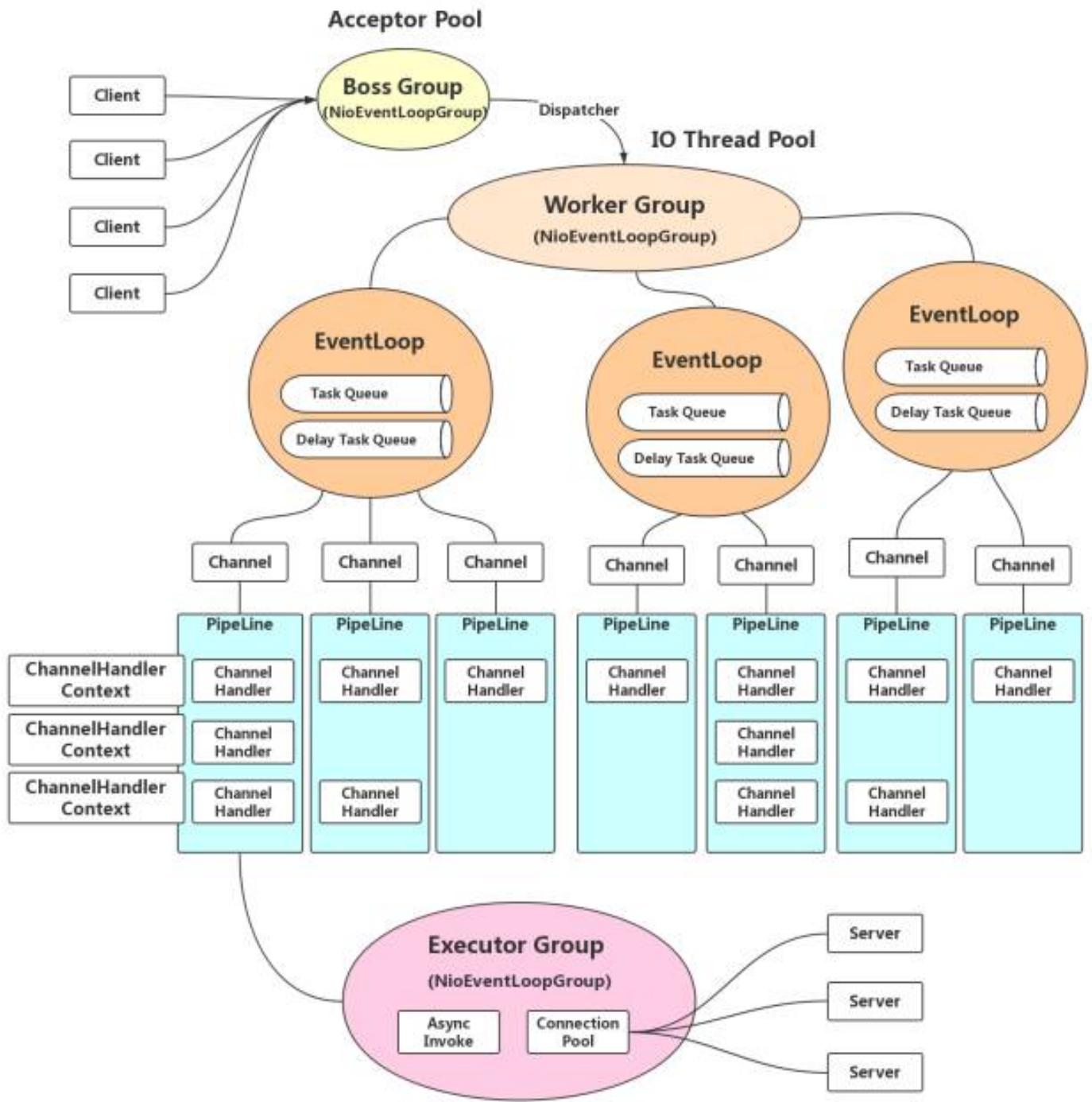


图 2: Netty 线程模型 实例

自然，在同一套体系下优秀的实践总是得以沿袭和被相互借鉴，**Dispatcher**也是参透了这套精髓，如下图所示只有work线程这部分是Dubbo源码直接接触的部分 的Dubbo线程模型

(<https://www.cnblogs.com/java-zhao/p/7822766.html>)，work线程会将流程转给Dubbo管理的Client线程池或者Server线程池，**Dispatcher**决定了业务执行是否有3或者5这个步骤，若没有，对应的业务会直接在work线程中执行。

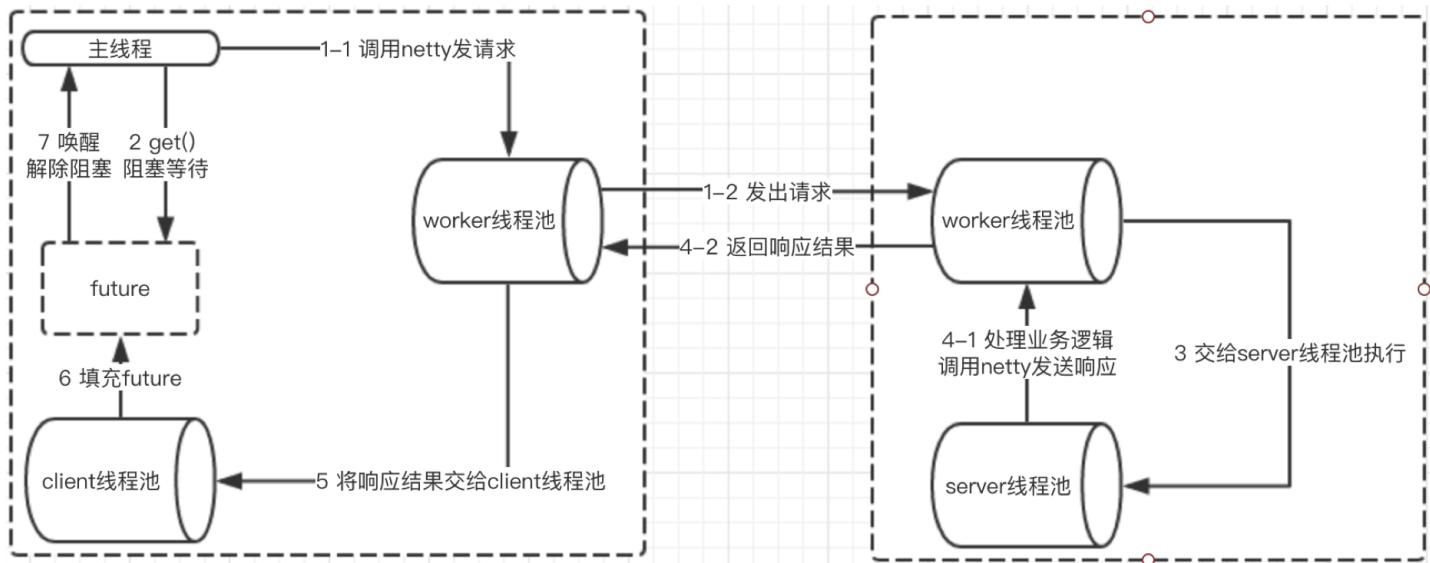


图 3: Dubbo 线程模型

其线程模型的大概流程如下:

- 1. 客户端的主线程发出一个请求后获得future，在执行get时进行阻塞等待；
- 2. 服务端使用worker线程（netty通信模型）接收到请求后，将请求提交到server线程池中进行处理
- 3. server线程处理完成之后，将相应结果返回给客户端的worker线程池（netty通信模型），最后，worker线程将响应结果提交到client线程池进行处理
- 4. client线程将响应结果填充到future中，然后唤醒等待的主线程，主线程获取结果，返回给客户端

由于考虑到线程切换本身也是个耗时操作，**Dispatcher**的几个不同实现方案正是根据不同的应用场景分别提炼的解决方案，可以根据业务需求决定采用哪种**Dispatcher**实现方案来完成对应的网络I/O事件。

## 具体实现

### Dispatcher实现方式

先看其定义，接口使用Dubbo的SPI机制，默认采用的**all**模式，其两个参数，一个是用于传递处理网络I/O事件的ChannelHandler，另一个是用于传递参数的URL，而返回参数又是一个ChannelHandler。从上文分析可知，**Dispatcher**做线程派发的对象正是ChannelHandler处理的网络I/O事件，由接口定义大概可以猜出，其实现是对ChannelHandler做增强处理，使之符合线程派发这个需求目标。

JAVA

```
@SPI(AllDispatcher.NAME)
public interface Dispatcher {

    /**
     * dispatch the message to threadpool.
     *
     * @param handler
     * @param url
     * @return channel handler
     */
    @Adaptive({Constants.DISPATCHER_KEY, "dispatcher", "channel.handler"})
    // The last two parameters are reserved for compatibility with the old
    configuration
    ChannelHandler dispatch(ChannelHandler handler, URL url);

}

//=====
//Dispatcher的子类并没有直接实现线程派发，委托给了对应的ChannelHandlerDelegate实现，
//它仅仅是创建并返回后者的实例。每一种具体实现均完全采用了如下模板
//=====

public class XXXXDispatcher implements Dispatcher {

    public static final String NAME = "message";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        return new XXXXChannelHandler(handler, url);
    }

}

/**
 * Direct dispatcher
 */
public class DirectDispatcher implements Dispatcher {

    public static final String NAME = "direct";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        //直接返回原handler，不进行包装处理
        return handler;
    }

}
```

再细看如下的类URML图，从下半截可以看出除了`DirectDispatcher`外，**Dispatcher**的其它每一个实现均会创建一种类型的 `ChannelHandlerDelegate`，由类图的上半截得知，整个实现采用了装饰器模式，抽象基类 `WrappedChannelHandler` 下的每一种装饰器实现均对应了一种线程派发方式。

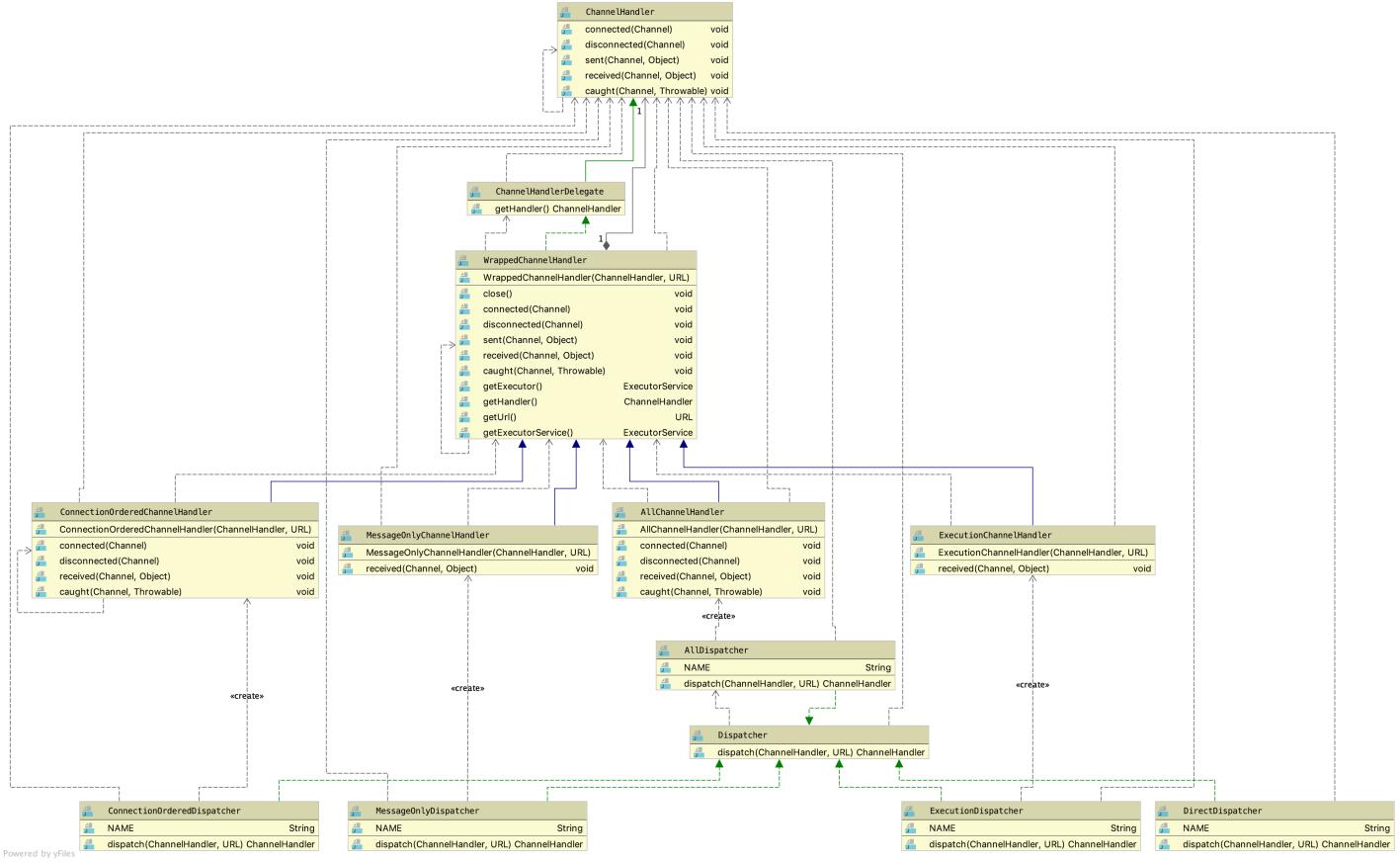


图 4: *Dubbo Dispatcher UML示意图*

所谓线程派发，简言之，是将某个具体行为的方法调用转换为**Runnable**提交到线程池异步执行，由线程池的调度器决定它的执行时机，当前线程的方法执行 栈帧将会立即返回被释放。最简单的示例如下述源码：

```
public class MessageOnlyChannelHandler extends WrappedChannelHandler {
```

JAVA

```
    public MessageOnlyChannelHandler(ChannelHandler handler, URL url) {
        super(handler, url);
    }

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

        //获取父类的任务执行调度器
        ExecutorService executor = getExecutorService();
        try {

            //将具体行为委托给对应的ChannelEventRunnable执行
            executor.execute(new ChannelEventRunnable(channel,
                handler, ChannelState.RECEIVED, message));
        } catch (Throwable t) {
            throw new ExecutionException(message, channel,
                getClass() + " error when process received event .", t);
        }
    }
}
```

## WrappedChannelHandler

WrappedChannelHandler这个抽象基类简化了具体线程派发的实现，将诸如获取任务执行调度器的公共部分都放在基类中，其它的网络I/O事件响应都是简单地调用所包装ChannelHandler的对应方法，由实现类根据需要覆写为异步执行。如果没有通过SPI机制指定ThreadPool，Dubbo会使用全局的SHARED\_EXECUTOR作为异步任务调度执行器，它持有一个可缓冲的线程池，否则会为每一个Client或者Server创建一个由具体ThreadPool持有的指定类型的线程池。

### NOTE

可缓冲的线程池，特点是当线程池的大小超过了处理任务所需要的线程时，就会回收部分最近60秒不执行任务的空闲的线程，线程池大小只受到操作系统（或者说JVM）能够创建的最大线程数的限制

```
public class WrappedChannelHandler implements ChannelHandlerDelegate {
```

JAVA

```
    protected static final Logger logger =
LoggerFactory.getLogger(WrappedChannelHandler.class);

    protected final ChannelHandler handler;

    protected final URL url;

    public WrappedChannelHandler(ChannelHandler handler, URL url) {
        this.handler = handler;
        this.url = url;

        //通过SPI机制获取Dubbo中的ThreadPool实现， SPI机制使用的全局ConcurrentMap缓存映射关系,
        //完全不用担心会同时存在ThreadPool多个实例
        executor = (ExecutorService)
ExtensionLoader.getExtensionLoader(ThreadPool.class)
            .getAdaptiveExtension().getExecutor(url);

        //Server端和Client端使用不同的componentKey缓存当前用到的executor
        //在同一个JVM中，可能会存在Server和Client
        String componentKey = Constants.EXECUTOR_SERVICE_COMPONENT_KEY;
        if (CONSUMER_SIDE.equalsIgnoreCase(url.getParameter(SIDE_KEY))) {
            componentKey = CONSUMER_SIDE;
        }
        //缓存容器，具体用途下文线程池优雅终止会涉及到
        DataStore dataStore =
ExtensionLoader.getExtensionLoader(DataStore.class).getDefaultExtension();
        dataStore.put(componentKey, Integer.toString(url.getPort()), executor);
    }

//=====
//executor是用于异步执行任务的调度器，由具体派发实现决定如何调用。
//=====

    protected static final ExecutorService SHARED_EXECUTOR =
Executors.newCachedThreadPool(new NamedThreadFactory("DubboSharedHandler", true));

    protected final ExecutorService executor;

    public ExecutorService getExecutor() {
        return executor;
    }

    //如果应用没有配置对应的SPI文件，构造方式使用SPI机制获取的ExecutorService可能会为空
    public ExecutorService getExecutorService() {
        ExecutorService cexecutor = executor;
        if (cexecutor == null || cexecutor.isShutdown()) {
            cexecutor = SHARED_EXECUTOR;
        }
        return cexecutor;
    }

    public void close() {
        try {
```

```

        if (executor != null) {
            executor.shutdown();
        }
    } catch (Throwable t) {
        logger.warn("fail to destroy thread pool of server: " + t.getMessage(), t);
    }
}

//=====
//ChannelHandler不一定被包装过
//=====

@Override
public ChannelHandler getHandler() {
    if (handler instanceof ChannelHandlerDelegate) {
        return ((ChannelHandlerDelegate) handler).getHandler();
    } else {
        return handler;
    }
}

public URL getUrl() {
    return url;
}

//=====
//简单调用所包装ChannelHandler的对应方法，由子类根据需要改写为异步执行
//=====

@Override
public void connected(Channel channel) throws RemotingException {
    handler.connected(channel);
}

@Override
public void disconnected(Channel channel) throws RemotingException {
    handler.disconnected(channel);
}

@Override
public void sent(Channel channel, Object message) throws RemotingException {
    handler.sent(channel, message);
}

@Override
public void received(Channel channel, Object message) throws RemotingException {
    handler.received(channel, message);
}

@Override
public void caught(Channel channel, Throwable exception) throws RemotingException {
    handler.caught(channel, exception);
}
}

```

## ChannelEventRunnable

上文中出现的ChannelEventRunnable将ChannelHandler中的所有事件行为均统一到一个Runnable中的，尽管他们的参数有所不同，好处是可以使用统一的风格触发具体事件执行，线程派发具体实现可以使用统一的Runnable调用。

```

public class ChannelEventRunnable implements Runnable {
    private static final Logger logger =
LoggerFactory.getLogger(ChannelEventRunnable.class);

    private final ChannelHandler handler;
    private final Channel channel;
    private final ChannelState state;
    private final Throwable exception;
    private final Object message;

    public ChannelEventRunnable(Channel channel, ChannelHandler handler, ChannelState
state) {
        this(channel, handler, state, null);
    }

    public ChannelEventRunnable(Channel channel, ChannelHandler handler, ChannelState
state, Object message) {
        this(channel, handler, state, message, null);
    }

    public ChannelEventRunnable(Channel channel, ChannelHandler handler, ChannelState
state, Throwable t) {
        this(channel, handler, state, null, t);
    }

    public ChannelEventRunnable(Channel channel, ChannelHandler handler, ChannelState
state, Object message, Throwable exception) {
        this.channel = channel;
        this.handler = handler;
        this.state = state;
        this.message = message;
        this.exception = exception;
    }

    @Override
    public void run() {
        if (state == ChannelState.RECEIVED) {
            try {
                handler.received(channel, message);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error,
channel is " + channel
                        + ", message is " + message, e);
            }
        } else {
            switch (state) {
            case CONNECTED:
                try {
                    handler.connected(channel);
                } catch (Exception e) {
                    logger.warn("ChannelEventRunnable handle " + state + " operation
error, channel is " + channel, e);
                }
                break;
            case DISCONNECTED:
                try {

```

```

        handler.disconnected(channel);
    } catch (Exception e) {
        logger.warn("ChannelEventRunnable handle " + state + " operation
error, channel is " + channel, e);
    }
    break;
case SENT:
    try {
        handler.sent(channel, message);
    } catch (Exception e) {
        logger.warn("ChannelEventRunnable handle " + state + " operation
error, channel is " + channel
                + ", message is " + message, e);
    }
    break;
case CAUGHT:
    try {
        handler.caught(channel, exception);
    } catch (Exception e) {
        logger.warn("ChannelEventRunnable handle " + state + " operation
error, channel is " + channel
                + ", message is: " + message + ", exception is " +
exception, e);
    }
    break;
default:
    logger.warn("unknown state: " + state + ", message is " + message);
}
}

//5种网络I/O事件
public enum ChannelState {
    CONNECTED,
    DISCONNECTED,
    SENT,
    RECEIVED,
    CAUGHT
}
}

```

## Dispatcher的各种线程派发细节

WrappedChannelHandler的每一个子类代表一种线程派发模型，它们决定了是否对网络I/O事件是否派发到业务线程池去异步执行，对应关系如下：

1. **all** AllChannelHandler → connected、disconnected、received、caught
2. **connection** ConnectionOrderedChannelHandler → connected、disconnected、received、caught
3. **message** MessageOnlyChannelHandler → received
4. **execution** ExecutionChannelHandler → received

## 5. **direct** 直接返回传入的 ChannelHandler，也就是在work线程中同步执行所有的的网络I/O回调事件

上述陈列的对应关系看起来让人有些懵，结合上文可知，虽然不同的派发方式对同一种网络I/O事件执行了业务线程派发处理，但是具体实现方式是不同的。

初看起来 `received` 回调事件是指Channel接收到请求，然而这个理解是不准确的，通讯过程中 Client和Server以Channel作为信息发送的通道，本文中的 线程派发过程中，我们可以初略的认为 Netty就是那个Channel，Client通过Channel发送请求给Server，随后Channel会告知Server I/O就绪——回调Server 提供的 `received` 实现，这时Server端接受到的是一个称为Request的请求对象。

如果Channel是双工的，Server通过同一Channel给Client发送了Request请求，若干ms后，Client处理完Server发给它的请求，会通过Channel回送一个响应给Server，Server端I/O就绪后，同样Channel也会告知Server I/O就绪，同样也是回调Server提供的 `received` 实现，只不过这次Server 端接受到的是一个称为Response的响应对象。

以 `ExecutionChannelHandler` 和 `MessageOnlyChannelHandler` 为例。

public class ExecutionChannelHandler extends WrappedChannelHandler {

JAVA

```
    public ExecutionChannelHandler(ChannelHandler handler, URL url) {
        super(handler, url);
    }

    @Override
    public void received(Channel channel, Object message) throws RemotingException {
        ExecutorService executor = getExecutorService();

        //接受到的是来自对端的请求
        if (message instanceof Request) {
            try {
                executor.execute(new ChannelEventRunnable(channel, handler,
                    ChannelState.RECEIVED, message));
            } catch (Throwable t) {

                if (t instanceof RejectedExecutionException) {
                    Request request = (Request) message;

                    if (request.isTwoWay()) {//检测到时双工方式
                        String msg = "Server side(" + url.getIp() + "," + url.getPort()
                            + ") thread pool is exhausted, detail msg:" +
                        t.getMessage();
                        Response response = new Response(request.getId(),
                            request.getVersion());
                        response.setStatus(Response.SERVER_THREADPOOL_EXHAUSTED_ERROR);
                        response.setErrorMessage(msg);

                        //通过channel回传一个Response响应对象
                        channel.send(response);
                        return;
                    }
                }
            }
            throw new ExecutionException(message, channel, getClass() + " error
when process received event.", t);
        }
    } else {
        //非Request请求，直接使用I/O线程走流程
        handler.received(channel, message);
    }
}
```

public class MessageOnlyChannelHandler extends WrappedChannelHandler {

```
    public MessageOnlyChannelHandler(ChannelHandler handler, URL url) {
        super(handler, url);
    }
```

//无论接受到来自对方的是Response响应还是Request请求，均派发到业务线程去执行

```
    @Override
    public void received(Channel channel, Object message) throws RemotingException {
        ExecutorService executor = getExecutorService();
        try {
            executor.execute(new ChannelEventRunnable(channel, handler,
```

```
        ChannelState.RECEIVED, message));
    } catch (Throwable t) {
        throw new ExecutionException(message, channel, getClass() + " error when
process received event .", t);
    }
}
```

**connection** 类型的线程派发模型实现有点特殊，为确保连接和断连能够有序的执行，**ConnectionOrderedChannelHandler** 单独维护了一个任务 执行调度器，其队列是 **LinkedBlockingQueue**。如下述源码：

```

public class ConnectionOrderedChannelHandler extends WrappedChannelHandler {

    protected final ThreadPoolExecutor connectionExecutor;
    private final int queuewarninglimit;

    public ConnectionOrderedChannelHandler(ChannelHandler handler, URL url) {
        super(handler, url);
        String threadName = url.getParameter(THREAD_NAME_KEY, DEFAULT_THREAD_NAME);
        connectionExecutor = new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            //该任务队列是确保连接和断连事件回调依次执行的关键
            new LinkedBlockingQueue<Runnable>
                (url.getPositiveParameter(CONNECT_QUEUE_CAPACITY, Integer.MAX_VALUE)),
            new NamedThreadFactory(threadName, true),
            new AbortPolicyWithReport(threadName, url)
        ); // FIXME There's no place to release connectionExecutor!
        queuewarninglimit = url.getParameter(CONNECT_QUEUE_WARNING_SIZE,
        DEFAULT_CONNECT_QUEUE_WARNING_SIZE);
    }

    @Override
    public void connected(Channel channel) throws RemotingException {
        try {
            checkQueueLength();
            connectionExecutor.execute(new ChannelEventRunnable(channel, handler,
            ChannelState.CONNECTED));
        } catch (Throwable t) {
            throw new ExecutionException("connect event", channel, getClass() + " error
when process connected event .", t);
        }
    }

    @Override
    public void disconnected(Channel channel) throws RemotingException {
        try {
            checkQueueLength();
            connectionExecutor.execute(new ChannelEventRunnable(channel, handler,
            ChannelState.DISCONNECTED));
        } catch (Throwable t) {
            throw new ExecutionException("disconnected event", channel, getClass() + " error
when process disconnected event .", t);
        }
    }

    private void checkQueueLength() {
        if (connectionExecutor.getQueue().size() > queuewarninglimit) {
            logger.warn(new IllegalThreadStateException("connectionordered channel
handler `queue size: " + connectionExecutor.getQueue().size() + " exceed the warning
limit number :" + queuewarninglimit));
        }
    }
    //其它事件直接分派到业务线程池中，实现和“all模式”一致，此处省略其余细节
    ...
}

```

JAVA

# Dispatcher 应用

## 产生——端的派发特性支持

上文已经分析过，**Dispatcher**实际上是产生一个 ChannelHandler 的装饰器，由具体的装饰器实现决定是否将Netty等网络I/O框架在回调对应的事件时是否将流程分派到业务线程执行。如下所示，Client和Server在初始化时会先调用 ChannelHandlers.wrap() 将传入的 ChannelHandler 做装饰处理，根据配置选用目标 Dispatcher。

```

public class ChannelHandlers {
    ...

    public static ChannelHandler wrap(ChannelHandler handler, URL url) {
        return ChannelHandlers.getInstance().wrapInternal(handler, url);
    }

    protected ChannelHandler wrapInternal(ChannelHandler handler, URL url) {
        //最外层装饰，支持同时接受多条请求
        return new MultiMessageHandler(
            //中间层装饰，增加心跳处理
            new HeartbeatHandler(
                //最里层装饰，增加线程派发特性
                ExtensionLoader.getExtensionLoader(Dispatcher.class)
                    .getAdaptiveExtension().dispatch(handler, url)));
    }
}

public class NettyServer extends AbstractServer implements Server {

    public NettyServer(URL url, ChannelHandler handler) throws RemotingException {
        super(url, ChannelHandlers.wrap(handler, ExecutorUtil.setThreadName(url,
SERVER_THREAD_POOL_NAME)));
    }

    ...
}

/**
 * 所有Client实现都会继承该类
 */
public abstract class AbstractClient extends AbstractEndpoint implements Client {
    ...

    protected static ChannelHandler wrapChannelHandler(URL url, ChannelHandler
handler) {
        url = ExecutorUtil.setThreadName(url, CLIENT_THREAD_POOL_NAME);
        url = url.addParameterIfAbsent(THREADPOOL_KEY, DEFAULT_CLIENT_THREADPOOL);
        return ChannelHandlers.wrap(handler, url);
    }
}

public class NettyClient extends AbstractClient {

    public NettyClient(final URL url, final ChannelHandler handler) throws
RemotingException {
        super(url, wrapChannelHandler(url, handler));
    }

    ...
}

```

## 消亡——端的线程池优雅终止

所谓线程池优雅终止就是在执行`close`操作之前让正在运行中的业务线程有机会获得资源完成其剩下的工作，而非一招致命。**Dispatcher**的实现始终没有离开 `ExecutorService` 的有力支持，其线程池优雅终止实现也是基于这个任务执行调度器，在进一步阐述Dubbo线程池优雅终止的实现方案前，先

看看ExecutorService 中有关 shutdown 操作的定义。

```
public interface ExecutorService extends Executor {  
  
    /**  
     * Initiates an orderly shutdown in which previously submitted  
     * tasks are executed, but no new tasks will be accepted.  
     * Invocation has no additional effect if already shut down.  
     *  
     * <p>This method does not wait for previously submitted tasks to  
     * complete execution. Use {@link #awaitTermination awaitTermination}  
     * to do that.  
     *  
     * @throws SecurityException if a security manager exists and  
     *         shutting down this ExecutorService may manipulate  
     *         threads that the caller is not permitted to modify  
     *         because it does not hold {@link  
     *             java.lang.RuntimePermission}{@code ("modifyThread")},  
     *         or the security manager's {@code checkAccess} method  
     *         denies access.  
     */  
    void shutdown();  
  
    /**  
     * Attempts to stop all actively executing tasks, halts the  
     * processing of waiting tasks, and returns a list of the tasks  
     * that were awaiting execution.  
     *  
     * <p>This method does not wait for actively executing tasks to  
     * terminate. Use {@link #awaitTermination awaitTermination} to  
     * do that.  
     *  
     * <p>There are no guarantees beyond best-effort attempts to stop  
     * processing actively executing tasks. For example, typical  
     * implementations will cancel via {@link Thread#interrupt}, so any  
     * task that fails to respond to interrupts may never terminate.  
     *  
     * @return list of tasks that never commenced execution  
     * @throws SecurityException if a security manager exists and  
     *         shutting down this ExecutorService may manipulate  
     *         threads that the caller is not permitted to modify  
     *         because it does not hold {@link  
     *             java.lang.RuntimePermission}{@code ("modifyThread")},  
     *         or the security manager's {@code checkAccess} method  
     *         denies access.  
     */  
    List<Runnable> shutdownNow();  
  
    /**  
     * Returns {@code true} if this executor has been shut down.  
     *  
     * @return {@code true} if this executor has been shut down  
     */  
    boolean isShutdown();  
  
    /**  
     * Returns {@code true} if all tasks have completed following shut down.  
     * Note that {@code isTerminated} is never {@code true} unless  
     */
```

```

 * either {@code shutdown} or {@code shutdownNow} was called first.
 *
 * @return {@code true} if all tasks have completed following shut down
 */
boolean isTerminated();

/**
 * Blocks until all tasks have completed execution after a shutdown
 * request, or the timeout occurs, or the current thread is
 * interrupted, whichever happens first.
 *
 * @param timeout the maximum time to wait
 * @param unit the time unit of the timeout argument
 * @return {@code true} if this executor terminated and
 *         {@code false} if the timeout elapsed before termination
 * @throws InterruptedException if interrupted while waiting
 */
boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException;

...
}

```

**Dispatcher**的线程池优雅终止实现正是借助如下3个主要的方法实现的，下述阐述有助于更加准确的理解他们的意图：

1. `void shutdown()` 顺序关闭先前已经提交的任务，不再接受新任务。调用完方法后，会立即返回，并不会等待所有已经提交的任务完成。

2. `List<Runnable> shutdownNow()` 尝试终止所有正在执行的任务，返回正在等待尚未执行任务列表。调用完方法后，会立即返回，并不会等待所有正在执行的任务终止。

除尽力尝试停止处理正在执行的任务之外，没有任何保证。例如，典型的实现将通过`Thread.interrupt`取消，因此任何无法响应中断的任务都可能永远不会终止。

3. `boolean awaitTermination(long timeout, TimeUnit unit)` 阻塞等待`ExecutorService`，直到如下三个情况中的任何一个出现才返回：

- 1) `shutdown(now)` 后所有的任务执行完成或终止
- 2) 出现超时
- 3) 当前线程被 `interrupted`

如下Java线程的状态转换图所示Java线程进入终止状态的前提是线程进入RUNNABLE状态，而实际上线程也可能处在休眠状态，也就是说，我们要想终止一个线程，首先要把线程的状态从休眠状态转换到RUNNABLE状态。

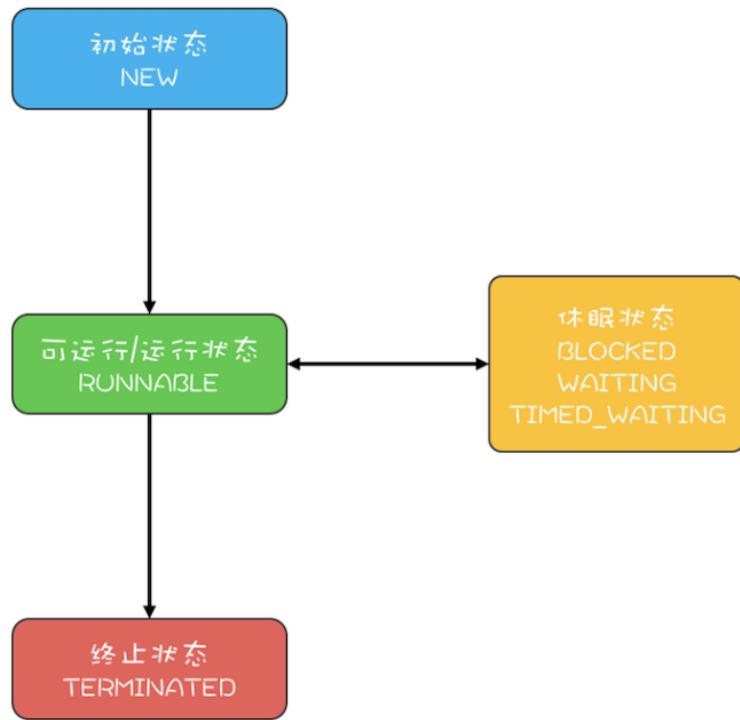


图 5: 线程状态转换图

Java Thread类提供的 `interrupt()` 方法可以将休眠状态的线程转换到**RUNNABLE**状态，这让线程有机会执行完剩下的任务部分，借助这个特性，另外加上一个**volatile**类型的线程终止的标志位可以在java使用一个线程通知另外一个线程实现线程池优雅终止。

Dubbo中由 `ExecutorUtil` 专门负责处理这些close问题。

JAVA

```

public class ExecutorUtil {
    private static final Logger logger = LoggerFactory.getLogger(ExecutorUtil.class);

    //声明最多只有一个线程的ThreadPoolExecutor, 其任务使用阻塞式链表队列, 确保有序执行,
    //用于执行ExecutorService的优雅关闭操作, 由整个JVM中的所有线程共享
    private static final ThreadPoolExecutor SHUTDOWN_EXECUTOR = new
    ThreadPoolExecutor(0, 1,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(100),
        new NamedThreadFactory("Close-ExecutorService-Timer", true));

    /**
     * 检测在调用shutdown(now)后ExecutorService是否已经完成或终止队列中所有的任务
     */
    public static boolean isTerminated(Executor executor) {
        if (executor instanceof ExecutorService) {
            if (((ExecutorService) executor).isTerminated()) {
                return true;
            }
        }
        return false;
    }

    private static void newThreadToCloseExecutor(final ExecutorService es) {
        if (!isTerminated(es)) { //如果ExecutorService未处于终止态
            SHUTDOWN_EXECUTOR.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        //循环1000次, 每隔10毫秒执行一次, 确保ExecutorService正在执行的任务能够
                        终止
                        for (int i = 0; i < 1000; i++) {
                            //尝试终止正在执行的任务
                            es.shutdownNow();
                            if (es.awaitTermination(10, TimeUnit.MILLISECONDS)) {
                                //成功等待到所有任务处理完毕
                                break;
                            }
                        }
                    } catch (InterruptedException ex) {
                        //从中断中恢复, 将唯一的一个线程归还给SHUTDOWN_EXECUTOR
                        //JVM中其它线程能正常使用它
                        Thread.currentThread().interrupt();
                    } catch (Throwable e) {
                        logger.warn(e.getMessage(), e);
                    }
                }
            });
        }
    }

    public static void gracefulShutdown(Executor executor, int timeout) {
        //检查是否符合终止条件, 只有ExecutorService类型的Executor才符合
        if (!(executor instanceof ExecutorService) || isTerminated(executor)) {
    }
}

```

```
        return;
    }

    final ExecutorService es = (ExecutorService) executor;
    try {
        //该方法仅仅是告知不再接受新任务的提交
        es.shutdown();
    } catch (SecurityException ex2) {
        return;
    } catch (NullPointerException ex2) {
        return;
    }
    try {
        //等待已提交的完成执行完成,
        if (!es.awaitTermination(timeout, TimeUnit.MILLISECONDS)) { //终止前, 出现了超时
            //尝试终止正在执行的任务
            es.shutdownNow();
        }
        //如果没有进入方法体, 表示ExecutorService在超时前已经完成关闭操作
    } catch (InterruptedException ex) { //处于休眠、阻塞、等待或超时等待的过程中, 被interrupted
        //尝试中断正在执行的任务
        es.shutdownNow();
        //捕获异常, 再次调用interrupt方法, 将中断状态重新设置为true
        //保留当前线程原有的状态
        Thread.currentThread().interrupt();
    }
    //如果ExecutorService仍未完成任务执行至关闭或者中断所有正在执行的任务,
    //便将尝试执行ExecutorService关闭的操作转给SHUTDOWN_EXECUTOR来执行
    if (!isTerminated(es)) {
        newThreadToCloseExecutor(es);
    }
}

//=====
//非优雅版终止处理, 区别是, 一上来就会尝试停止正在执行的任务, 随后等待所有任务结束, 如果超时或者被中断, 也会启用
//newThreadToCloseExecutor努力尝试等待到所有任务执行完
//=====

public static void shutdownNow(Executor executor, final int timeout) {
    if (!(executor instanceof ExecutorService) || isTerminated(executor)) {
        return;
    }
    final ExecutorService es = (ExecutorService) executor;
    try {
        es.shutdownNow();
    } catch (SecurityException ex2) {
        return;
    } catch (NullPointerException ex2) {
        return;
    }
    try {
```

```

        es.awaitTermination(timeout, TimeUnit.MILLISECONDS);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
    if (!isTerminated(es)) {
        newThreadToCloseExecutor(es);
    }
}

...
}

```

线程池优雅终止的步骤总结如下：

1. 当前线程调用 `shutdown()` 告知ExecutorService不再接受新提交的任务；
2. 随后尝试等待若干ms直到其完成
  - a. 如果超时返回，则调用 `shutdownNow()` 告知ExecutorService尝试终止正在执行的任务；
  - b. 否则在超时前若当前线程捕获到遭遇被 `interrupted` 的异常，则再次调用 `shutdownNow()`，确保ExecutorService收到“尝试终止正在执行的 任务”的通知，随后调用 `Thread.currentThread().interrupt()` 恢复当前线程状态；
3. 至此，如果ExecutorService还处于未终止状态，则给 `SHUTDOWN_EXECUTOR` 这个使用单线程专门顺序执行终止处理的 `ThreadPoolExecutor` 发出一个 任务，由其继续执行终止处理。
4. `shutdown`线程会尝试每隔10毫秒执行组合调用 `shutdownNow()` 和 `awaitTermination()` 一次，如果后者检测到正常终止或者超过1000次(总计10s) 这时就不再尝试
5. 若`shutdown`线程执行任务期间遭遇 `interrupted` 产生的异常，则对其进行恢复，确保当前JVM的其它线程能正常使用 `SHUTDOWN_EXECUTOR` 这个线程池。

## 线程池优雅终止的应用

上文中介绍线程派发时，其实现装饰器抽象基类`WrappedChannelHandler`中有如下一段代码，利用`DataSource`将Client或Server实例化时获得的`ExecutorService`缓存起来，其目的只有一个，就是用于实现端的关闭——资源回收处理有效的任务运行期都集中于网络I/O事件的处理。

```

//=====
//DataStore的结构: <component name or id, <data-name, data-value>>
//ConcurrentMap<String, ConcurrentMap<String, Object>>
//可以理解是双键结构的缓存容器，需要使用两级键获取到目标值
//=====
DataStore dataStore = ExtensionLoader.getExtensionLoader(
    DataStore.class).getDefaultExtension();
dataStore.put(componentKey,
    Integer.toString(url.getPort()), executor);

```

JAVA

在AbstractClient和AbstractServer初始化的最后阶段均存在一段获取该缓存容器中ExecutorService的代码片段，由上文分析可知，这个值是在他们的具体实现类中装入派发模式时置入的，也就是说，如果没有使用线程派发机制，那么事先就没有置入该值，其值为null。此处的实现方式值得商榷，传入的 ChannelHandler 经过了多层装饰，如果使用了线程派发，通过递归回调 ChannelHandlerDelegate.getHandler() 判断是否为 WrappedChannelHandler 也可以获得该值。

```
public AbstractClient(URL url, ChannelHandler handler) throws RemotingException {  
    super(url, handler);  
    ...  
  
    executor = (ExecutorService) ExtensionLoadergetExtension(DataStore.class)  
        .getDefaultValue().get(CONSUMER_SIDE, Integer.toString(url.getPort()));  
    ExtensionLoader.getExtensionLoader(DataStore.class)  
        .getDefaultValue().remove(CONSUMER_SIDE, Integer.toString(url.getPort()));  
}  
public AbstractServer(URL url, ChannelHandler handler) throws RemotingException {  
    super(url, handler);  
    ...  
  
    //fixme replace this with better method  
    DataStore dataStore = ExtensionLoader.  
        getExtensionLoader(DataStore.class).getDefaultValue();  
    executor = (ExecutorService) dataStore.get(  
        Constants.EXECUTOR_SERVICE_COMPONENT_KEY, Integer.toString(url.getPort()));  
}
```

剩下的应用比较简单，但是有一处不得不提的是，如下述代码，在Client执行 close() 的时候有检查 executor 是否为null，而执行 close(int timeout) 却没有这种检查，可能对不使用线程派发模式的场景来说，谈不上线程池的优雅终止，也就不会调用到该方法，有待细究代码实现。

```
public abstract class AbstractClient extends AbstractEndpoint implements Client {  
    @Override  
    public void close() {  
  
        ...  
        try {  
            if (executor != null) {  
                ExecutorUtil.shutdownNow(executor, 100);  
            }  
        } catch (Throwable e) {  
            logger.warn(e.getMessage(), e);  
        }  
        ...  
    }  
  
    @Override  
    public void close(int timeout) {  
        ExecutorUtil.gracefulShutdown(executor, timeout);  
        close();  
    }  
}
```

JAVA

# Dubbo远程通讯 · 信息交换层

---

“ 请求-响应 是一种面向网络的通讯模式，由一台主机响应另一台主机的特定数据请求，通常，在发送完整的消息之前有一系列这样的交换而这通常是基于在 两个应用程序通过一个通道进行的双向对话。

由上述表述可以看出，平常所熟悉的 请求-响应 模式表面上是一问一答的，实际一个应答背后可能对应着多个交互往来，另外其通讯是基于双工通道的。

Dubbo将端到端的网络通讯相关的行为封装网络传输层**Transporter**，具体的网络I/O则委托给了类似Netty等的网络通讯中间件，而更关注业务语义的 信息交换行为则抽象成 **Exchanger** 置于会话层中，官方文档中有如下表述，说明其关注重点是一问一答。为了单位时间能够获得更大的吞吐量 避免同步阻塞产生不必要的资源浪费，Dubbo在这一抽象层中完成了同步转异步的操作。

“ **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*

## Transporter层基础

### 实现方式介绍

相对Dubbo的网络传输层，这一层所涉及实现细节及复杂度均小很多，如下UML图所示，上半截的 **Endpoint**、**Channel**、**Client**、**Server**、**ChannelHandler** 等均已 在传输层定义和实现，而下半截的 **ExchangeChannel**、**ExchangeClient**、**ExchangeServer**、**ExchangeChannelHandler** 这几个关键组件则是在当前交换层 定义的，实现方式几乎是沿袭传输层且一一对应的，上一层是对下一层的封装和增强，当前层只聚焦于其业务实现所需，这是分层模式思想的精髓。

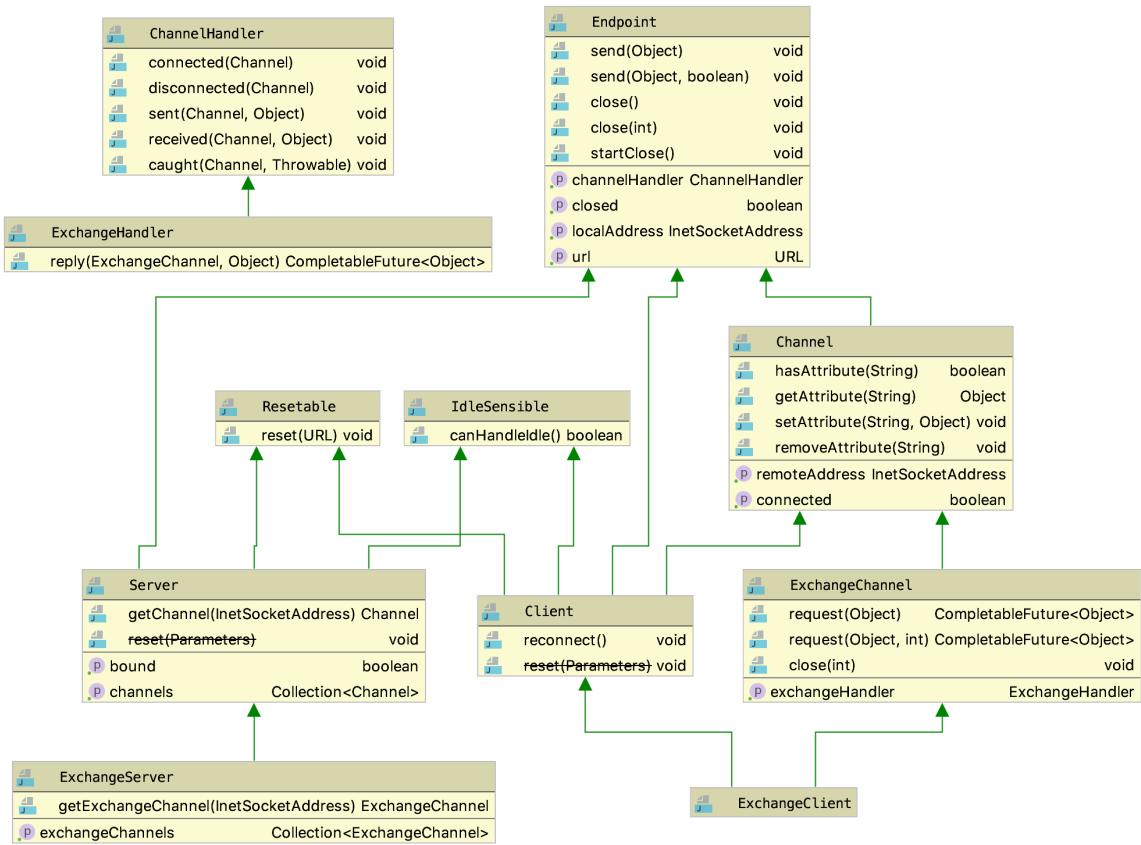


图: Dubbo Exchange UML

依然, Client和Channel的一对一绑定关系, Client的行为直接委托给当前连接的Channel实现, 因此如下两个同步转异步的操作被定义在 ExchangeChannel 中。

```

public interface ExchangeChannel extends Channel {
    // send request.
    CompletableFuture<Object> request(Object request)
        throws RemotingException;

    // send request.
    CompletableFuture<Object> request(Object request, int timeout)
        throws RemotingException;
}

```

JAVA

信息交换层和传输层的关系, 最直观的表示如下图, 入站的请求或响应数据流总是经由交换层转入传输层, 出站的响应或请求则反之。

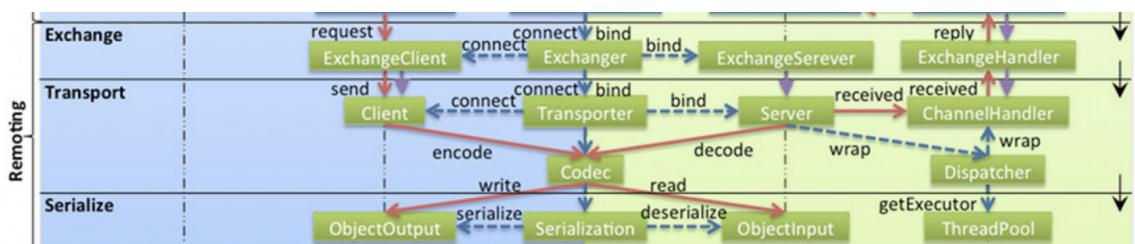


图: Dubbo Framework Remoting

对应Dubbo中的具体实现方式是类的继承关系, 如 ExchangeChannel 表达的是交换层的行为, 其父类 Channel 表达的是更为底层的传输层行为, 因此该层大多接口定义的方法行为实际上已由其扩展的接口定义, 通过一些类型的强制转换手段可以达到同样的效果, 当然不能就此认为这些定义多余, 从语义上来说 这些定义确保了该层的行为完

整性，使之更加符合分层架构的风格，获取该层定义相关对象也更加便利。具体如下述代码：

JAVA

```
final class HeaderExchangeChannel implements ExchangeChannel {
    ...
    @Override
    public ChannelHandler getChannelHandler() {
        return channel.getChannelHandler();
    }

    @Override
    public ExchangeHandler getExchangeHandler() {
        return (ExchangeHandler) channel.getChannelHandler();
    }
    ...
}

public class HeaderExchangeServer implements ExchangeServer {
    ...
    @Override
    public Collection<ExchangeChannel> getExchangeChannels() {
        Collection<ExchangeChannel> exchangeChannels =
            new ArrayList<ExchangeChannel>();

        Collection<Channel> channels = server.getChannels();
        if (CollectionUtils.isNotEmpty(channels)) {
            for (Channel channel : channels) {
                exchangeChannels.add(
                    HeaderExchangeChannel.getOrAddChannel(channel));
            }
        }
        return exchangeChannels;
    }

    //ExchangeChannel → Channel
    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Collection<Channel> getChannels() {
        return (Collection) getExchangeChannels();
    }

    //ExchangeChannel → Channel
    @Override
    public Channel getChannel(InetSocketAddress remoteAddress) {
        return getExchangeChannel(remoteAddress);
    }

    @Override
    public ExchangeChannel getExchangeChannel(InetSocketAddress remoteAddress) {
        Channel channel = server.getChannel(remoteAddress);
        return HeaderExchangeChannel.getOrAddChannel(channel);
    }
    ...
}
```

## Request / Response 核心模型定义

Table 1. Request / Response

属性	Request	Response	作用

属性	Request	Response	作用
final long <b>mId</b>	✓	✓	会话ID，Response的ID来自对应的Request，计算方式： AtomicLong.getAndIncrement()
String <b>mVersion</b>	✓	✓	版本号，实际上是当前所使用的Dubbo环境的Dubbo RPC protocol version，由 <code>Version.getProtocolVersion()</code> 取得
boolean <b>mEvent</b>	✓	✓	<p>心跳、只读事件标识</p> <pre>public static final String HEARTBEAT_EVENT = null, READONLY_EVENT = "R";</pre> <pre>public void setEvent(String event) {     this.mEvent = true;     this.mData = event; }  public boolean isHeartbeat() {     return mEvent &amp;&amp; HEARTBEAT_EVENT == mData; }  public void setHeartbeat(boolean isHeartbeat) {     if (isHeartbeat) {         setEvent(HEARTBEAT_EVENT);     } }</pre>
boolean <b>mTwoWay</b>	✓	✗	双向标识，若为true，在一段收到请求后，需要给对方回应一个应答，让对方确认我方已经收到请求，根据当前处理状态发回对应的响应，存在正常、异常、心跳3种响应
boolean <b>mBroken</b>	✓	✗	请求发送到对方后，由Netty等还原成Request对象，如果对方在解码环节过程中遇到异常，便会设置该标识，由其他handler根据该标识做进一步处理。

属性	Request	Response	作用
Object <b>mData</b>	✓	✗	封装请求内容的数据容器，可以是任何Java类型的对象  <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><ol style="list-style-type: none"><li>1. String HEARTBEAT_EVENT = null mData的值为null值时被认为心跳事件</li><li>2. String READONLY_EVENT = "R" mData的值为“R”则说明服务器端整处于正在关闭中的状态，不再执行有关写操作等。</li></ol></div>
Object <b>mResult</b>	✗	✓	封装响应内容的数据容器，可以是任何Java类型的对象
byte <b>mStatus</b>	✗	✓	响应状态，总共有如下10种状态

属性	Request	Response	作用
			<p style="text-align: right;">JAVA</p> <pre> public static final byte OK = 20, //ok.  CLIENT_TIMEOUT = 30, //client side timeout.  SERVER_TIMEOUT = 31, //server side timeout.  CHANNEL_INACTIVE = 35, //channel inactive, directly return the unfinished requests.  BAD_REQUEST = 40, //request format error.  BAD_RESPONSE = 50, //response format error.  SERVICE_NOT_FOUND = 60, //service not found.  SERVICE_ERROR = 70, //service error.  SERVER_ERROR = 80, //internal server error.  CLIENT_ERROR = 90, //internal server error. </pre>
String <b>mErrorMsg</b>	✗	✓	可读错误响应消息，返回给请求方

## 周期任务

因Dubbo需要时刻检查当前自身的相关组件的状态，多处出现周期定时任务的身影，本序列文章的开始就介绍了Dubbo的定时轮算法。在Exchanger这个框架层中有如下三种类型的定时任务：

1. **Client 心跳**: Client总是定期的给对方发送心跳事件，维持长联状态；

通道上最近读操作或写操作距当前时间已经超过一个心跳周期

2. **Client 重连**: Client定期检测当前所持Channel通道是否掉线，掉线则重连；

通道已经处于掉线状态，或者最近读操作已经超过最大允许闲置时间

3. **Server** 下线处理：为避免接入Client方由于宕机或断电后还持有其连接Channel通道产生资源浪费，Server会定期轮询连入Channel的状态，若有一段 时间未发生读写事件，则将其剔除处理——`close()`。

通道上最近读操作或写操作距当前时间已经超过最大闲置允许时间

本文不在赘述定时轮算法相关原理或机制，简要说明具体应用依据。在[Dubbo远程通讯 · 网络传输层]这一文中已经提到，Dubbo实现的通道Channel持有一个缓存通道本地环境变量的键值对Map，当前层在I/O回调事件发生时，会向其中写入 `HeaderExchangeHandler` 中的读写时间戳 `KEY_READ_TIMESTAMP` 和 `KEY_WRITE_TIMESTAMP`，周期定时任务只要根据实现需求执行对应检查处理便可。它们和I/O回调事件的关系如下：

- **KEY\_READ\_TIMESTAMP**: `connected`、`received`
- **KEY\_WRITE\_TIMESTAMP**: `connected`、`sent`

`disconnected` 回调则会清除这些时间戳标识。

注：另外每次心跳触发的I/O回调事件也会执行同样的读写时间戳的写入操作，当然通道断连后，无需处理 `disconnected` 回调。

最近读写时间戳的实现是通过装饰器类 `HeartbeatHandler` 实现的，机制是通过它在绑定于Client或Server上的Channel的回调网络I/O事件，根据事件类型对应写入读或者写时间戳。对于 `received` 回调，如果判别到对方发送的是心跳请求，在对方 `isTwoWay()` 的特性基础上会发送一个心跳响应回去，心跳响应则只会简单打印日志。

```

public class HeartbeatHandler extends AbstractChannelHandlerDelegate {

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

    public static final String KEY_READ_TIMESTAMP = "READ_TIMESTAMP";

    public static final String KEY_WRITE_TIMESTAMP = "WRITE_TIMESTAMP";

    public HeartbeatHandler(ChannelHandler handler) {
        super(handler);
    }

    @Override
    public void connected(Channel channel) throws RemotingException {
        setReadTimestamp(channel);
        setWriteTimestamp(channel);
        handler.connected(channel);
    }

    @Override
    public void disconnected(Channel channel) throws RemotingException {
        clearReadTimestamp(channel);
        clearWriteTimestamp(channel);
        handler.disconnected(channel);
    }

    @Override
    public void sent(Channel channel, Object message) throws RemotingException {
        setWriteTimestamp(channel);
        handler.sent(channel, message);
    }

    @Override
    public void received(Channel channel, Object message) throws RemotingException {
        setReadTimestamp(channel);
        if (isHeartbeatRequest(message)) {
            Request req = (Request) message;
            if (req.isTwoWay()) {
                Response res = new Response(req.getId(), req.getVersion());
                res.setEvent(Response.HEARTBEAT_EVENT);
                channel.send(res);
                if (logger.isInfoEnabled()) {
                    int heartbeat = channel.getUrl().getParameter(Constants.HEARTBEAT_KEY, 0);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Received heartbeat from remote channel " +
channel.getRemoteAddress() +
                            ", cause: The channel has no data-transmission exceeds a heartbeat
period" +
                            + (heartbeat > 0 ? ":" + heartbeat + "ms" : ""));
                    }
                }
            }
            return;
        }
        if (isHeartbeatResponse(message)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Receive heartbeat response in thread " +
Thread.currentThread().getName());
            }
            return;
        }
        handler.received(channel, message);
    }
}

```

} ...

## NOTE

KEY\_READ\_TIMESTAMP 和 KEY\_WRITE\_TIMESTAMP 这两个Channel本地属性除了用在 HeartbeatHandler，也在 HeaderExchangeHandler 被用到，为啥会被两个 ChannelHandler 网络I/O事件回调器使用？因为是后者会被所有Dubbo支持的网络I/O中间的支持，但前者则取决于他们是否有能力支持心跳机制，也即 IdleSensible，不支持的情况下只会在Client端启用 ReconnectTimerTask 周期定时重连任务。

## 定时轮应用细节

URL是所有Dubbo的关键组件的配置总线，有关其配置都是需要单独维护和使用的，因此这里的心跳周期时长和闲置时长都是从URL中获取到的。Dubbo中相关时机 计算方式如下：

1. Client端每1/3心跳周期时段检测是否需要发送心跳包；
2. 每1/3心跳周期时段服务端或客户端检测是否需要关闭连接或执行重连处理。

闲置超时时长，默认是  $3 \times$  心跳周期，最少为  $2 \times$  心跳周期，客户端的和服务端的检测均是按同一个周期执行的，假定服务端结束一个周期的时间，客户端的恰好刚刚已经开始，如果客户端按小于两个周期时间执行超时计算，则会错过客户端的断连后的重试操作。

```

//=====
//DEFAULT_HEARTBEAT: 1m, 默认的心跳周期时长。
//=====

public class UrlUtils {
    public static int getIdleTimeout(URL url) {
        int heartBeat = getHeartbeat(url);
        // idleTimeout should be at least more than twice heartBeat because possible retries of
client.
        int idleTimeout = url.getParameter(Constants.HEARTBEAT_TIMEOUT_KEY, heartBeat * 3);
        if (idleTimeout < heartBeat * 2) {
            throw new IllegalStateException("idleTimeout < heartbeatInterval * 2");
        }
        return idleTimeout;
    }

    public static int getHeartbeat(URL url) {
        return url.getParameter(Constants.HEARTBEAT_KEY, Constants.DEFAULT_HEARTBEAT);
    }
}

//=====
//HEARTBEAT_CHECK_TICK: 值为3。
//LEAST_HEARTBEAT_DURATION: 1000ms, 默认的最少间隔周期时长。但最终取决于客户端或服务端配置
//=====

/**
 * Each interval cannot be less than 1000ms.
 */
private long calculateLeastDuration(int time) {
    if (time / HEARTBEAT_CHECK_TICK <= 0) {
        return LEAST_HEARTBEAT_DURATION;
    } else {
        return time / HEARTBEAT_CHECK_TICK;
    }
}

public class HeaderExchangeClient implements ExchangeClient {
...
    private void startHeartBeatTask(URL url) {
        if (!client.canHandleIdle()) {
            AbstractTimerTask.ChannelProvider cp = () ->
                Collections.singletonList(HeaderExchangeClient.this);

            //根据配置总线中解析心跳周期时长
            int heartbeat = getHeartbeat(url);
            long heartbeatTick = calculateLeastDuration(heartbeat);
            this.heartBeatTimerTask = new HeartbeatTimerTask(cp, heartbeatTick, heartbeat);
            IDLE_CHECK_TIMER.newTimeout(heartBeatTimerTask, heartbeatTick, TimeUnit.MILLISECONDS);
        }
    }

    private void startReconnectTask(URL url) {
        if (shouldReconnect(url)) {
            AbstractTimerTask.ChannelProvider cp = () ->
                Collections.singletonList(HeaderExchangeClient.this);

            //根据配置总线中解析计算闲置超时时长
            int idleTimeout = getIdleTimeout(url);
            long heartbeatTimeoutTick = calculateLeastDuration(idleTimeout);
            this.reconnectTimerTask = new ReconnectTimerTask(cp, heartbeatTimeoutTick, idleTimeout);
            IDLE_CHECK_TIMER.newTimeout(reconnectTimerTask, heartbeatTimeoutTick,
TimeUnit.MILLISECONDS);
        }
    }
}

```

```

...
}

public class HeaderExchangeServer implements ExchangeServer {
...
    private void startIdleCheckTask(URL url) {
        if (!server.canHandleIdle()) {
            AbstractTimerTask.ChannelProvider cp = () ->
                unmodifiableCollection(HeaderExchangeServer.this.getChannels());

            int idleTimeout = getIdleTimeout(url);
            long idleTimeoutTick = calculateLeastDuration(idleTimeout);
            CloseTimerTask closeTimerTask = new CloseTimerTask(cp, idleTimeoutTick, idleTimeout);
            this.closeTimerTask = closeTimerTask;

            // init task and start timer.
            IDLE_CHECK_TIMER.newTimeout(closeTimerTask, idleTimeoutTick, TimeUnit.MILLISECONDS);
        }
    }
...
}

```

## 具体实现细节

### ExchangeChannel → HeaderExchangeChannel

**ExchangeChannel** 的实现方式是包装另外一个现成的Channel，除下述方法外，基本所有的行为都是直接委托给这个内嵌的Channel，包括通道本地变量 Attribute 的存取、自身和对端的 InetSocketAddress 的获取、连接的状态获取、 getUrl() 等。当两个**HeaderExchangeChannel** 实例中内嵌 Channel 等同时，这两对象便彼此 equals()。

#### 初涉同步转异步

网络传输层中已提及，消息的传送是通过绑定到端和端的 Channel 通道完成的，在此基础上**ExchangeChannel** 担负的同步转异步处理的职责，大致原理是：

1. 先实例化一个持有该 Channel 通道、当前 Request 请求对象、超时时间的 DefaultFuture→CompletableFuture；
2. 由内嵌 Channel 先完成正常的发送操作；
3. 随后还没等到结果返回就给调用方返回 DefaultFuture<Object> 对象

DefaultFuture 类中持有一个全局<Long, DefaultFuture> 键值对 Map

4. 该层中注册 ExchangeChannelHandler 在被回调 received(Channel, Object) 时，会从入站的 Response 响应解析得到会话 ID，由其从 Map 中获取到 DefaultFuture 实例对象
5. 根据 Response 的状态选择调用 complete(T) 或 completeExceptionally(Throwable) 最终完成同步转异步的处理。

上述简要说明了同步转异步的过程，具体细节不只这么复杂，先有个概念，便于渐渐理解 Exchanger 这一框架层的实现。

有关请求发送的实现代码如下，构建 Request 请求对象，将其封装在 DefaultFuture 中返回：

JAVA

```
@Override
public CompletableFuture<Object> request(Object request) throws RemotingException {
    return request(request, channel.getUrl().
        getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT));
}

@Override
public CompletableFuture<Object> request(Object request, int timeout)
    throws RemotingException {

    if (closed) {
        throw new RemotingException(this.getLocalAddress(), null,
            "Failed to send request " + request +
            ", cause: The channel " + this + " is closed!");
    }
    // create request.
    Request req = new Request();
    req.setVersion(Version.getProtocolVersion());

    //告知需要有响应返回
    req.setTwoWay(true);

    //将实际的请求内容包装在Request的mData字段中
    req.setData(request);

    //返回结果前，将Request封装在DefaultFuture中
    DefaultFuture future = DefaultFuture.newFuture(channel, req, timeout);
    try {
        //调用内嵌传输层channel完成请求数据的发送操作，下述调用没有阻塞
        channel.send(req);
    } catch (RemotingException e) {
        //如果遇到异常，则调用CompletableFuture的cancel
        future.cancel();
        throw e;
    }
    return future;
}
```

## close状态管理

HeaderExchangeChannel中有一个比较特殊的 volatile boolean closed 类型的变量，该状态量和内嵌传输层的Channel通道实例中持有的互不相干，它的存在仅服务于当前信息交换层，更多是为了给当前通道尚未收到结果的DefaultFuture来个优雅处理，确保它们正常完成或者超时结束后再关闭传输层的 Channel通道。这个实现依赖于DefaultFuture类中持有一个全局<Long, Channel>键值对Map，只要该Map中含有当前持有的传输层 Channel，便需优雅close。

```

private volatile boolean closed = false;

@Override
public boolean isClosed() {
    return closed;
}

// graceful close
@Override
public void close(int timeout) {
    if (closed) {
        return;
    }
    closed = true;
    if (timeout > 0) {
        long start = System.currentTimeMillis();
        while (DefaultFuture.hasFuture(channel)
                && System.currentTimeMillis() - start < timeout) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                logger.warn(e.getMessage(), e);
            }
        }
    }
    close();
}

```

## 创建方式

在网络传输层中介绍过因Channel属于一种专属I/O管控资源，需要对NettyChannel进行适当保护，防止被误用，限定其只能在当前包中使用，因而类的作用域被声明为 `default` 的，构造函数也被声明为 `private`，必须通过能够记录 `<io.netty.channel.Channel, NettyChannel>` 键值对关系的 `getOrAddChannel` 创建。本质是由于 NettyChannel的I/O通讯行为实现本非其本身，是委托给作为键的Channel完成的，之间存在一对一的强绑定关系。

同样，`HeaderExchangeChannel` 也存在一个类似的同名方法，不同的是它和Dubbo自身声明的 `org.apache.dubbo.remoting.Channel` 是一种依附关系，高层对基层的依赖，因此在实现上，其绑定关系是直接使用后者的本地变量存取容器加以表达的。在当前层中的每一种I/O回调事件中，会先调用该方法确保正处于活态的连入**Channel**绑入了 `HeaderExchangeChannel` 实例，它是确保信息交换层的职责得以体现的关键一环。当然 I/O回调的最后无论如何会调用 `removeChannelIfDisconnected`，将已经失活的Channel的绑定关系移除。

```

private final Channel channel;

HeaderExchangeChannel(Channel channel) {
    if (channel == null) {
        throw new IllegalArgumentException("channel == null");
    }
    this.channel = channel;
}

//根据Channel创建需要绑定在其上对应的HeaderExchangeChannel
static HeaderExchangeChannel getOrAddChannel(Channel ch) {
    if (ch == null) {
        return null;
    }
    HeaderExchangeChannel ret = (HeaderExchangeChannel) ch.getAttribute(CHANNEL_KEY);
    if (ret == null) {
        ret = new HeaderExchangeChannel(ch);

        //如果Channel不存在于连接激活态中，创建的HeaderExchangeChannel实例便会处于游离态中
        if (ch.isConnected()) {
            ch.setAttribute(CHANNEL_KEY, ret);
        }
    }
    return ret;
}

//根据本地变量容器中的Channel移除对应的HeaderExchangeChannel
static void removeChannelIfDisconnected(Channel ch) {
    if (ch != null && !ch.isConnected()) {
        ch.removeAttribute(CHANNEL_KEY);
    }
}

```

## HeaderExchangeClient → ExchangeClient

如传输层的NettyClient实现，HeaderExchangeClient也实现了 HeaderExchangeChannel 接口，后者扩展自 Channel这个基础接口，也就是说他的行为 同样最终是委托给 HeaderExchangeChannel 完成的。使用实现接口代替组合引用，可以直接利用委托模式给外界提供便利的访问接口，也更加准确的描述了 Client和Channel严格的一一对应关系，后者于前者而言是一种强依赖存在。

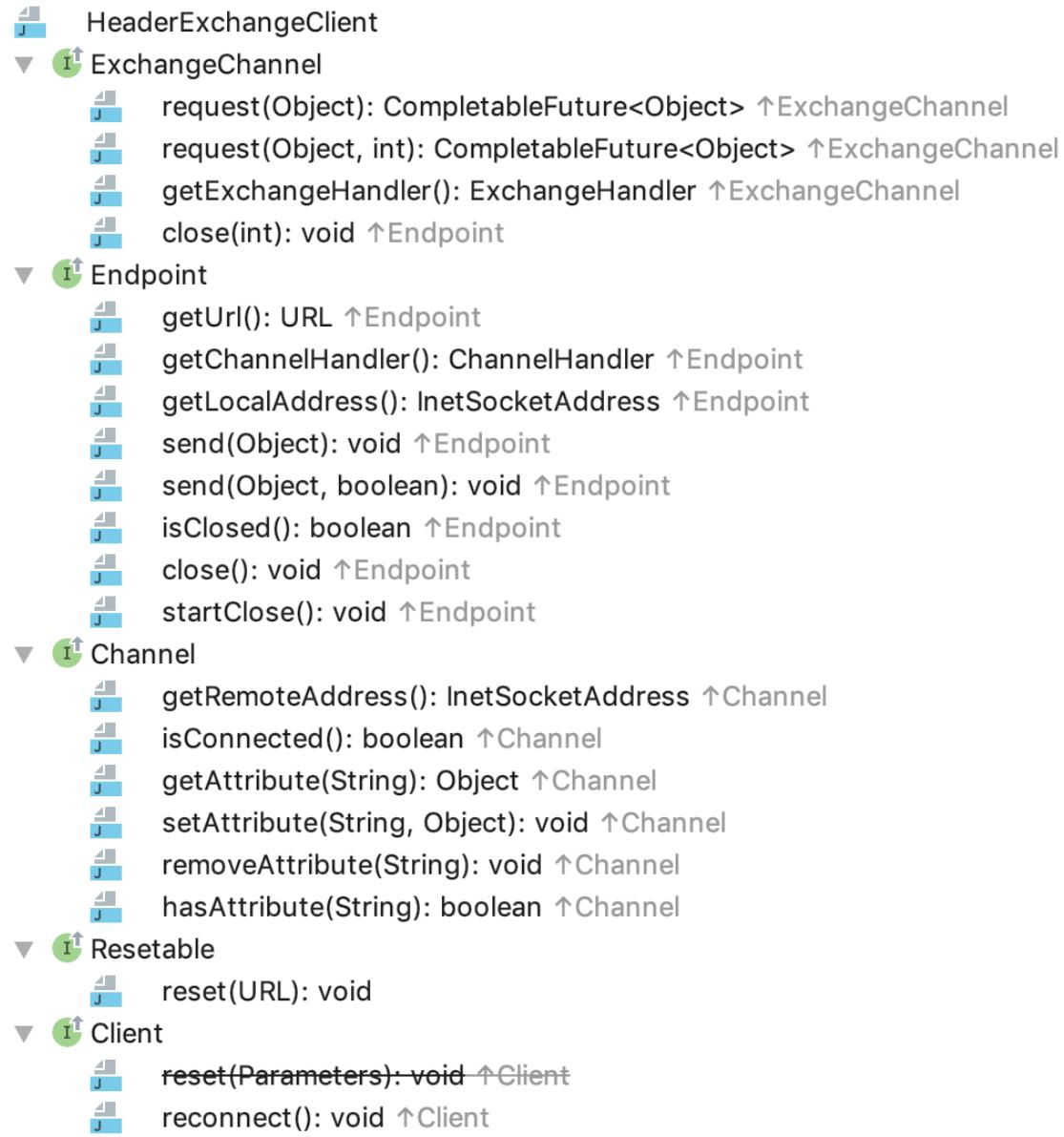


图: Dubbo Exchange UML

在其基础能力已被 `HeaderExchangeChannel` 实现的基础上，如上文所述Client需要负责：1) 维持其绑定 Channel通道和对端Server的长连状态；2) 在因自身故障掉线后采取重连处理。因而它的侧重点是定时轮资源的分配、定时任务的创建和取消。定时轮实际上是一个维持了单一线程的周期任务调度器，在Java 中线程是一种宝贵的资源，并非每个Client都能独享一份，因此同一个JVM内所有Client共享一个全局的定时轮实例。

注：同一Jvm中的所有Server实例也是共享一个定时轮实例，不和Client共用的原因是有些应用使用了Dubbo的客户端访问其他服务资源，但并不对外提供服务，反过来提供服务的Dubbo应用也不一定会拥有需要访问Dubbo服务的Client实例。

```
public class HeaderExchangeClient implements ExchangeClient {

    private final Client client;
    private final ExchangeChannel channel;

    //定时轮所有Client共享一份
    private static final HashedWheelTimer IDLE_CHECK_TIMER = new HashedWheelTimer(
        new NamedThreadFactory("dubbo-client-idleCheck", true), 1, TimeUnit.SECONDS,
    TICKS_PER_WHEEL);

    //保持长连的心跳检测任务和重连任务是每个Client实例都独有的，互不干扰
    private HeartbeatTimerTask heartBeatTimerTask;
    private ReconnectTimerTask reconnectTimerTask;

    public HeaderExchangeClient(Client client, boolean startTimer) {
        Assert.notNull(client, "Client can't be null");
        this.client = client;

        //client实现了Channel接口
        this.channel = new HeaderExchangeChannel(client);

        if (startTimer) {
            //标识启用定时轮的情况下，每个Client实例初始化时启用周期任务执行重连和心跳处理
            URL url = client.getUrl();
            startReconnectTask(url);
            startHeartBeatTask(url);
        }
    }

    @Override
    public CompletableFuture<Object> request(Object request) throws RemotingException {
        //委托给ExchangeChannel完成请求
        return channel.request(request);
    }

    ...

    @Override
    public void close() {
        doClose();
        channel.close();
    }

    @Override
    public void close(int timeout) {
        // Mark the client into the closure process
        startClose();
        doClose();
        channel.close(timeout);
    }

    private void doClose() {
        if (heartBeatTimerTask != null) {
            heartBeatTimerTask.cancel();
        }

        if (reconnectTimerTask != null) {
            reconnectTimerTask.cancel();
        }
    }

    @Override
    public void startClose() {
        channel.startClose();
    }
}
```

```
    ...  
}
```

## NOTE

在HeaderExchangeClient关闭执行资源回收之前，一定得先取消当前正在异步执行的定时任务，其周期性不间断的运行会导致资源的消耗。更糟糕的是通道 Channel已经关闭了，所有上述提到的周期任务都会执行网络I/O处理，必定会抛错，资源消耗相比更加严重。

## HeaderExchangeServer → ExchangeServer

同 HeaderExchangeClient 一样，HeaderExchangeServer 并没有直接继承自某个Server的实现比如 NettyServer，而是使用组合的方式内置一个 Server 引用，这样做好处是并没有绑定传输层的某一种具体实现，可以根据需要灵活的组合传输层的具体Server实现，不至于产生类爆炸的现象，保持了框架的精炼简洁。因而，扩展自 Server 接口部分的行为大多都委托给了所引用的 Server 实例。

上文中已经提及 Exchanger 框架层的 Server 为了避免资源消耗，需要周期性的检测是否有连入 Client 因为宕机等原因已经掉线，若掉线则将其清除出去，具体应用和原理已经介绍，这里不再赘述。

和 HeaderExchangeClient 不同的是，Server 端是被多个 Client 接入的，因而其状态维护就变得比较关键：

1. 在关闭之前先得检测是否还有处于活动态的；
2. 连入 Channel 通道，处于正在关闭的过程中时，需要及时知会所有接入 Client，自己不再接受写操作；
3. 一旦启用关闭操作，作为服务提供方，后续便不能再往外向所有连入 Client 发送除 ReadOnly 事件的任何信息；

具体如下源码：

```

public class HeaderExchangeServer implements ExchangeServer {

    使用原子变量，确保并发时，第一时间获知当前的关闭状态
    private AtomicBoolean closed = new AtomicBoolean(false);

    ...

    @Override
    public boolean isClosed() {
        return server.isClosed();
    }

    //只要还有一个活动态的连入Channel通道，便认为Server还处于运行态
    private boolean isRunning() {
        Collection<Channel> channels = getChannels();
        for (Channel channel : channels) {

            /**
             * If there are any client connections,
             * our server should be running.
             */

            if (channel.isConnected()) {
                return true;
            }
        }
        return false;
    }

    @Override
    public void close() {
        doClose();
        server.close();
    }

    @Override
    public void close(final int timeout) {
        startClose();
        if (timeout > 0) {
            final long max = (long) timeout;
            final long start = System.currentTimeMillis();

            //只读消息的发送与否取决于Server端的总线配置
            if (getUrl().getParameter(Constants.CHANNEL_SEND_READONLYEVENT_KEY, true)) {
                sendChannelReadOnlyEvent();
            }

            //超时内，只要还有Client没有断连，则继续等待
            while (HeaderExchangeServer.this.isRunning()
                    && System.currentTimeMillis() - start < max) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    logger.warn(e.getMessage(), e);
                }
            }
        }

        //取消当前闲置检测周期任务
        doClose();
        server.close(timeout);
    }
}

```

```

//大部分行为都委托给内置的server实例完成
@Override
public void startClose() {
    server.startClose();
}

private void sendChannelReadOnlyEvent() {

    //构建只读消息请求，不再接受写请求，无需响应
    Request request = new Request();
    request.setEvent(Request.READONLY_EVENT);
    request.setTwoWay(false);
    request.setVersion(Version.getProtocolVersion());

    //获取所有连入的处于已连接状态的通道Channel，挨个发送只读消息
    Collection<Channel> channels = getChannels();
    for (Channel channel : channels) {
        try {
            if (channel.isConnected()) {
                channel.send(request, getUrl().getParameter(
                    Constants.CHANNEL_READONLYEVENT_SENT_KEY, true));
            }
        } catch (RemotingException e) {
            logger.warn("send cannot write message error.", e);
        }
    }
}

//由成功执行原子变量closed改写的线程取消任务
private void doClose() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }
    cancelCloseTask();
}

private void cancelCloseTask() {
    if (closeTimerTask != null) {
        closeTimerTask.cancel();
    }
}

@Override
public void reset(URL url) {
    server.reset(url);
    try {
        //getUrl()获取的是当前Server预制的配置值
        int currHeartbeat = getHeartbeat(getUrl());
        int currIdleTimeout = getIdleTimeout(getUrl());

        //配置总线url中获取的是动态置入的配置
        int heartbeat = getHeartbeat(url);
        int idleTimeout = getIdleTimeout(url);

        //如果重设入参的配置值和当前Server不一致，则先取消当前的闲置检测任务,
        //使用url入参重新刷
        if (currHeartbeat != heartbeat || currIdleTimeout != idleTimeout) {
            cancelCloseTask();
            startIdleCheckTask(url);
        }
    } catch (Throwable t) {
        logger.error(t.getMessage(), t);
    }
}

```

```
//Server被关闭后，便不能通过连入Channel通道往所有Client发送消息
@Override
public void send(Object message, boolean sent) throws RemotingException {
    if (closed.get()) {
        throw new RemotingException(this.getLocalAddress(), null, "Failed to send message " +
message
                + ", cause: The server " + getLocalAddress() + " is closed!");
    }
    server.send(message, sent);
}

...

```

## 同步转异步实现

Dubbo中的同步转异步是在当前信息交换层中处理的，实现采用了 ChannelHandler 的网络I/O事件回调机制。5个事件中，任务最繁重的当属 `sent` 和 `received`，也是同步转异步的主战场。上面已经大概介绍了其实现过程，本章节将深入其实现细节。

### 了解 DefaultFuture

`DefaultFuture` 的底层实现是 `CompletableFuture<Object>`，可初步认为后者是一个基于fork-join的异步多线程任务编排框架，有关其运作机制 及实现原理，将会在另外一篇文章独立展开，本章节只介绍其在 `DefaultFuture` 实现中的应用。

### 场景分析

端到端的网络通讯行为几乎绝大部分场景都发生在两台主机间，可能相距千里，即便在同一台主机的两个进程间，分属于不同的JVM实例，如果不是收到彼端主动 推送过来的信息，此端没有任何途径感知彼端发生了什么。而网络通讯是个复杂的过程，异常是几乎不可避免的，往往发生了，也就意味着收不到彼端传送过来 的消息了，归纳起来就是需要针对如下几种场景做异常处理：

1. 消息已准备好，经此端传输层发往对端的过程中，遇到故障抛出异常；
2. 消息经本地传输层发送出去后，网络传输过程中遇到异常；
3. 发送消息后，成功抵达对端，彼端响应处理出现异常；
4. 彼端成功处理响应，回传响应遇到异常；

于第一种异常，由于属于同一JVM乃至同一线程内，普通的 `try-catch` 足够应对，第三种异常，Dubbo框架会负责捕获并回传相应的异常响应，也能妥善 应对。其它两种异常属于链路层异常，由于已没法感知到彼端，又不可能无限制的等下去，因此需要配合超时检测机制，一旦超时便在此端做超时响应处理。可见，超时异常响应是由本机自发构造，这也确保了 `CompletableFuture<Object>` 不会长时间未被响应回调，客户端能得到及时反馈。

### 实现缘由

也就是说 `DefaultFuture` 的业务职能核心是处理响应，包括正常响应、异常响应以及本机检测产生的超时响应。实现采用了 `DefaultFuture → CompletableFuture<Object>`，至于为什么，下面我们逐步撕开其面纱：

1. Dubbo将重I/O操作的远程通讯委托给了诸如Netty等第三方中间件，效率、吞吐量、硬件利用率等的综合考量，往往它们会采用基于I/O多路复用机制甚至异步方式实现，这也就意味着I/O就绪后会基于通知回调方式告知调用方，不会造成其阻塞；
2. 基于通知回调方式的编程实际就是响应式编程，因任务编排比较困难或容易造成回调地狱，绝大部分习惯了直线思维的开发人员是没法适应这种编程方式的，为迎合着绝大部分人，Dubbo使用了同步转异步方案；
3. 在《Dubbo远程通讯·网络传输层》一文中已经提到过，当前Client使用在连的Netty Channel通道发出消息后便即刻返回，后续的响应包括异常响应通过I/O回调才能获知；
4. 当然如果配置总线传入了sent参数，便阻塞直到获得对方的响应，同样是效率的考量，Dubbo中这个参数不会被鼓励。当前信息交换框架层，相对而言所涉代码粒度跨度相当大，一个能适应该场景的非阻塞实现方案不可或缺，最好是能够完美配合Netty的回调方式，还要不影响到传输层work线程的调度方式，这些要求也就限制了几乎只能使用CompletableFuture：
  - a. 它不要求一定要有执行计算的主体部分，简单实例化后便可交给线程池，且不阻塞调用方；
  - b. 可通过调用complete()或completeExceptionally()将直接给result赋值调度完成该Future；
  - c. 正常或异常响应结果可通过whenComplete()回调感知到；
5. 随之而来的问题是，对于一个通讯往来，调用方调用Channel.send()后便即刻返回，同时它需要持有CompletableFuture实例获取处理结果，而该实例结果完成处理是有ChannelHandler I/O回调负责，但后者始终是针对Channel做回调处理的，一句话“ChannelHandler怎么根据Channel获取到CompletableFuture实例”？
6. 对应关系通过<Channel, CompletableFuture>表示？但是，通过在连的Channel通道发生的通讯往来并非只一个回合Request→Response，每一个回合都需要维护一个的生命周期仅限于该回合的CompletableFuture实例，所幸一对Request→Response组合中共同持有一个全局单调唯一递增的编号，因此上述对应关系需通过该ID编号间接关联；
7. 回到Channel和CompletableFuture的对应关系上，很显然，是一对多的，因此依后者来管理这种关系是最合适的，但为什么不使用组合和使用继承扩展方案，并且只覆写了cancel()这么一个方法？
8. 使用继承不必实例化多个相关实例CompletableFuture和及上述关系管理类，与其生命周期紧密相随的内容更容易被管理，用完便扔，不用手动清理，由垃圾收集器根据需要负责清理。至于为啥要覆写cancel()，API文档中的如下一段话，也许能解释为什么Dubbo的protocol协议层印证了这一点，大概意思是如果其被调用了，那么依赖它的未完成CompletableFuture实例会被异常完成处理，如下：

**“**If not already completed, completes this CompletableFuture with a CancellationException. Dependent CompletableFutures that have not already completed will also complete exceptionally with a CompletionException caused by this CancellationException.

## 源码分析

由上文分析得知，在同步转异步的过程中，DefaultFuture的业务职能核心是处理响应，需要负责管理协同和Channel的关系，还要启用定时轮算法检测超时任务。

## 关系维护

从上述分析中得出其生命周期只限于一个 Request→Response 往来这个过程，DefaultFuture 是在 Request 实例诞生时得以构建的，随其消亡。更宏观 方面是，需要使用静态内部变量全局地维护实例 Channel 到 CompletableFuture 的关系，以便 ChannelHandler 执行 I/O 事件回调时通过前者 获取到后者，当然如上述，这是间接的。尽管 Request 的生命周期可能很短，但是于整个 JVM 内，二者关系依然时刻可能都是一对多的，请求可以并发地发 送出去。

```
public class DefaultFuture extends CompletableFuture<Object> {  
  
    //dubbo中的请求发送是允许并发的  
    private static final Map<Long, Channel> CHANNELS = new ConcurrentHashMap<>();  
  
    private static final Map<Long, DefaultFuture> FUTURES = new ConcurrentHashMap<>();  
  
    // invoke id.  
    // 从request中获取，作为属性出现，便于使用  
    private final Long id;  
  
    private final Channel channel;  
  
    //Request和当前DefaultFuture是一对一的关系  
    private final Request request;  
  
    private DefaultFuture(Channel channel, Request request, int timeout) {  
        this.channel = channel;  
        this.request = request;  
        this.id = request.getId();  
  
        ...  
  
        // put into waiting map.  
        FUTURES.put(id, this);  
        CHANNELS.put(id, channel);  
    }  
  
    public static DefaultFuture getFuture(long id) {  
        return FUTURES.get(id);  
    }  
  
    //因Future是同Channel一起加入的，因此可以通过Channel是否在关系对中得出是否存在对应future的结论  
    public static boolean hasFuture(Channel channel) {  
        return CHANNELS.containsValue(channel);  
    }  
}
```

## 消息已发确认

传输层在确认信息成功发送后，会立马回调注册 ChannelHandler 的 sent(Channel, Object) 方法通知这一消息。DefaultFuture 中持有一个内存可见的 volatile 型的 sent 变量，在确认发送后改写该值，以便超时检测任务能及时知晓，有了这一依据超时检测任务便能及时判别异常发生在此端还是彼端了（成功发送出去说明通讯链路是正常的，回传瞬间发生链路异常的概率很小，即便发生也同一认为是对端异常）。TimeoutCheckTask 由负责 检测超时的线程池分配线程调度，而获取 DefaultFuture 实例改写该状态值处于 Netty 视觉的工作线程中，因此该标识属于共享资源，需要 volatile 标识。

```
private volatile long sent;

public static void sent(Channel channel, Request request) {
    DefaultFuture future = FUTURES.get(request.getId());
    if (future != null) {
        future.doSent();
    }
}

private boolean isSent() {
    return sent > 0;
}

private void doSent() {
    sent = System.currentTimeMillis();
}
```

## 超时检测

依然超时检测离不开时钟轮的支持，同样也是使用全局专用的名为 TIME\_OUT\_TIMER 单一 Timer 实例，每一个 DefaultFuture 实例都拥有一个超时检测任务，任务随 DefaultFuture 一起实例化。需要注意的是这里使用的 TimeoutCheckTask 实例承载的是一次性任务，因此运行完后便不可能再被使用。

### NOTE

调用 Timeout 实例的 cancel() 方法会将自身取消，被装入定时轮的 cancelledTimeouts 队列中，待时钟引擎进入下一个滴答时刻将其移除，也就是 任务被取消

```

public class DefaultFuture extends CompletableFuture<Object> {

    //超时检测时间轮，每一个ticket周期为30ms
    public static final Timer TIME_OUT_TIMER = new HashedWheelTimer(
        new NamedThreadFactory("dubbo-future-timeout", true),
        30,
        TimeUnit.MILLISECONDS);

    private final long start = System.currentTimeMillis();

    private final int timeout;

    // 每一个DefaultFuture拥有一个超时检测任务
    private Timeout timeoutCheckTask;

    private DefaultFuture(Channel channel, Request request, int timeout) {
        ...

        //默认设置的超时为1s，超时时间可以直接传入，或者从Channel通道关联配置总线中获取
        this.timeout = timeout > 0 ? timeout : channel.getUrl().getPositiveParameter(TIMEOUT_KEY,
DEFAULT_TIMEOUT);

        ...
    }

    private int getTimeout() {
        return timeout;
    }

    /**
     * check time out of the future
     */
    private static void timeoutCheck(DefaultFuture future) {
        TimeoutCheckTask task = new TimeoutCheckTask(future.getId());
        future.timeoutCheckTask = TIME_OUT_TIMER.newTimeout(task, future.getTimeout(),
TimeUnit.MILLISECONDS);
    }

    /**
     * init a DefaultFuture
     * 1.init a DefaultFuture
     * 2.timeout check
     *
     * @param channel channel
     * @param request the request
     * @param timeout timeout
     * @return a new DefaultFuture
     */
    public static DefaultFuture newFuture(Channel channel, Request request, int timeout) {
        final DefaultFuture future = new DefaultFuture(channel, request, timeout);
        // timeout check
        timeoutCheck(future);
        return future;
    }

    private String getTimeoutMessage(boolean scan) {
        long nowTimestamp = System.currentTimeMillis();
        return (sent > 0 ? "Waiting server-side response timeout" : "Sending request timeout in
client-side")
            + (scan ? " by scan timer" : "") + ". start time: "
            + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date(start))) + ", end
time: "
            + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date())) + ","
            + (sent > 0 ? " client elapsed: " + (sent - start)

```

```

        + " ms, server elapsed: " + (nowTimestamp - sent)
        : " elapsed: " + (nowTimestamp - start)) + " ms, timeout: "
        + timeout + " ms, request: " + request + ", channel: " + channel.getLocalAddress()
        + " -> " + channel.getRemoteAddress();
    }

private static class TimeoutCheckTask implements TimerTask {

    //记录Request的编号ID，间接关联DefaultFuture实例
    // 实际上也可以直接关联
    private final Long requestID;

    TimeoutCheckTask(Long requestID) {
        this.requestID = requestID;
    }

    @Override
    public void run(Timeout timeout) {
        DefaultFuture future = DefaultFuture.getFuture(requestID);
        //当前任务是一次性的，因此如果没有找到对应的future或者future已经完成，
        // 立马返回后会被定时轮移除掉
        if (future == null || future.isDone()) {
            return;
        }

        // 构造异常请求
        // create exception response.
        Response timeoutResponse = new Response(future.getId());
        // set timeout status.
        timeoutResponse.setStatus(future.isSent() ? Response.SERVER_TIMEOUT :
Response.CLIENT_TIMEOUT);
        timeoutResponse.setErrorMessage(future.getTimeoutMessage(true));

        //最后一个为true的timeout参数，告知当前是超时检测任务，由于任务已经执行完，无需再执行任务的取消操作
        // handle response.
        DefaultFuture.received(future.getChannel(), timeoutResponse, true);
    }
}
...
}
}

```

## 响应处理

此端成功接收到对方的响应说明，一个通讯往来已经完成，相应DefaultFuture也该寿终正寝了，表示 Channel 到 CompletableFuture 的关系也需要 被解除。最后阶段需要构建Response对象，通过父类的 complete() 通知上层调用方正常处理结果，或者调用completeExceptionally通知异常完成。

覆盖的 cancel() 方法中构造了一个 Response 对象，表明当前请求也许允许此端异常而被放弃。另外它没有调用 received 方法，而是在调用完 doReceived 随即将关系解除，由于对应 DefaultFuture 实例已经不在 FUTURES 中，持有的定时任务启动运行后没有机会再次构造Response 重复通知实例持有方。

```

public static void received(Channel channel, Response response) {
    received(channel, response, false);
}

//timeout参数识别是否为超时检测任务中发起的对本方法的调用
public static void received(Channel channel, Response response, boolean timeout) {
    try {
        DefaultFuture future = FUTURES.remove(response.getId());
        if (future != null) {
            Timeout t = future.timeoutCheckTask;

            //非来自超时检测任务的调用，由于当前任务已经完成，需要将其从定时轮中取消
            if (!timeout) {
                // decrease Time
                t.cancel();
            }
            future.doReceived(response);
        } else {
            logger.warn("The timeout response finally returned at "
                    + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date()))
                    + ", response " + response
                    + (channel == null ? "" : ", channel: " + channel.getLocalAddress()
                    + " -> " + channel.getRemoteAddress()));
        }
    } finally {
        //CHANNELS中，有可能多个ID关联着一个Channel，移除单个，并不会影响其他的
        CHANNELS.remove(response.getId());
    }
}

//通知持有当前``CompletableFuture<Object>``实例的调用方最终处理结果
private void doReceived(Response res) {
    if (res == null) {
        throw new IllegalStateException("response cannot be null");
    }
    if (res.getStatus() == Response.OK) {
        this.complete(res.getResult());
    } else if (res.getStatus() == Response.CLIENT_TIMEOUT
            || res.getStatus() == Response.SERVER_TIMEOUT) {

        this.completeExceptionally(new TimeoutException(res.getStatus()
                == Response.SERVER_TIMEOUT, channel, res.getErrorMessage()));
    } else {
        this.completeExceptionally(new RemotingException(channel, res.getErrorMessage()));
    }
}

//为了避免依赖当前对象的CompletableFuture实例因Cancel而被异常完成，覆写cancel方法
// 同时覆写也在语义上符合Dubbo的需要
@Override
public boolean cancel(boolean mayInterruptIfRunning) {
    Response errorResult = new Response(id);
    errorResult.setStatus(Response.CLIENT_ERROR);
    errorResult.setErrorMessage("request future has been canceled.");
    this.doReceived(errorResult);
    FUTURES.remove(id);
    CHANNELS.remove(id);
    return true;
}

public void cancel() {
    this.cancel(true);
}

```

**NOTE**

上述代码中，分别针对因本地异常主动放弃、本地超时或者彼端超时、任务正常响应构建了 `Response`，但在通知 `CompletableFuture` 实例持有方时，只有正常响应的 `Response` 被原样返回，其它两个场景均被解析成了对应的异常，尽管多了一层转换，但好处很明显，统一看待，编码更灵活方便，保持了架构的干净整洁。

## HeaderExchangeHandler

**IMPORTANT**

当前网络传输层的 `HeaderExchangeHandler` 并不是该层中 `ExchangeHandler` 的实现，其接口实现关系为 `HeaderExchangeHandler → ChannelHandlerDelegate → ChannelHandler`，也就是说它是一个普通的网络I/O事件监听器实现，而后者是一个多了两个扩展特性的接口：1) `String telnet(Channel, String)`；2) `CompletableFuture<Object> reply(ExchangeChannel, Object)`。前者使用组合的方式将后者作为属性元素纳入，在收到需要响应的入站请求时，会根据请求类型将流程转到后者两个方法中的一个。同时后者的具体实现大部分集中在更高一级的Protocol协议层，`Request` 对象在当前层被屏蔽，取其Object类型的 `mData` 作为 `reply` 的参数。

## 读写时间戳记录

上文已经提到，为了无论在那种网络I/O中间件下，都能够支持客户端周期性地“检测是否掉线，若掉线则重连”，因此 `HeaderExchangeHandler` 在每次I/O回调时 都会执行记录最近读、写时间戳。结合上文“周期任务”，具体实现代码如下：

```

public class HeaderExchangeHandler implements ChannelHandlerDelegate{

    public void connected(Channel channel) throws RemotingException {
        channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis());
        channel.setAttribute(KEY_WRITE_TIMESTAMP, System.currentTimeMillis());
        ...
    }

    public void disconnected(Channel channel) throws RemotingException {
        channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis());
        channel.setAttribute(KEY_WRITE_TIMESTAMP, System.currentTimeMillis());
        ...
    }

    public void received(Channel channel, Object message) throws RemotingException {
        channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis());
        ...
    }

    public void sent(Channel channel, Object message) throws RemotingException {
        channel.setAttribute(KEY_WRITE_TIMESTAMP, System.currentTimeMillis());
        ...
    }
}

```

## 上层Channel实例创建

另外，从上文中已经知道当前信息交换层和下一层所有几个关键组件均是一一对应关系，基本上该层的组件基于下层的实例使用组合方式创建。特殊点的是，支持异步外发消息的ExchangeChannel，因其I/O行为是委托给第三方中间件完成的，当前应用在回调中被动完成业务逻辑，因此同 NettyChannel 一样，其实例创建是由底层驱动更高一级的框架层完成的，“(io.netty).Channel → o.a.d.r.transport.NettyChannel → o.a.d.r.e.s.header.HeaderExchangeChannel”，显然这个实例化操作只能在5个网络I/O回调方法中完成，由于通道Channel在一个“Request → Response往来”中，底层基本不会发生变化，因此会在当前框架层的 ChannelHandler 的回调方法中创建 o.a.d.r.Channel 实例，又因需要在每个I/O回调方法执行这一操作，不得不结合Map来确保只创建一个与底层Channel实例对应的该实例。结合上文介绍过的 getOrAddChannel 和 removeChannelIfDisconnected，统一视觉看之，实现细节中便有了如下的模板代码：

```

@Override
public void XXX(Channel channel, ...) throws RemotingException {
    ExchangeChannel exchangeChannel = HeaderExchangeChannel
        .getOrAddChannel(channel);

    try {
        handler.XXX(exchangeChannel, ...);
    } finally {
        HeaderExchangeChannel.removeChannelIfDisconnected(channel);
    }
}

```

## 入站数据处理

入站的数据主要包括彼端发往此端的请求**Request**、响应**Response**、用于服务治理的纯String类型的 telnet 命令，前者又包含心跳请求、单向 请求、双向请求3种类型，如下：

```

@Override
public void received(Channel channel, Object message) throws RemotingException {
    channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis());
    final ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
    try {
        if (message instanceof Request) {
            // handle request.
            Request request = (Request) message;
            if (request.isEvent()) {
                handlerEvent(channel, request);
            } else {
                if (request.isTwoWay()) {
                    handleRequest(exchangeChannel, request);
                } else {
                    //单向Request无需Exchanger层继续处理，交由Transport信息传输层
                    // 做进一步处理
                    handler.received(exchangeChannel, request.getData());
                }
            }
        } else if (message instanceof Response) {
            handleResponse(channel, (Response) message);
        } else if (message instanceof String) {
            ... //入站telnet命令处理
        } else {
            //非Dubbo实现的入站数据类型
            handler.received(exchangeChannel, message);
        }
    } finally {
        HeaderExchangeChannel.removeChannelIfDisconnected(channel);
    }
}

```

## 入站响应处理

上文已提及此端构建完请求使用Channel通道发送出去后就立即返回，调用方持有 DefaultFuture 实例，由 Dubbo 在网络 I/O 事件回调方法中通知其响应已经抵达，也即调用 DefaultFuture.received 完成整个异步操作 (complete CompletableFuture)。

```

static void handleResponse(Channel channel, Response response) throws RemotingException {
    if (response != null && !response.isHeartbeat()) {
        DefaultFuture.received(channel, response);
    }
}

```

## 入站事件处理

入站的事件分为心跳、只读。这里只需要对后者加以处理，表示服务端正在关闭，告知客户端不再接受写操作，此时其 Request 请求对象中的 mData 属性值为“R”。Client 的内置 Channel 通道中的 CHANNEL\_ATTRIBUTE\_READONLY\_KEY 本地属性值表示对应通往服务端的通道当前是否可正常使用，Dubbo 会依据所有本实例注册的 Client 中的这些值判别当前实例是否处于活跃态。

```

void handlerEvent(Channel channel, Request req) throws RemotingException {
    if (req.getData() != null && req.getData().equals(Request.READONLY_EVENT)) {
        channel.setAttribute(Constants.CHANNEL_ATTRIBUTE_READONLY_KEY, Boolean.TRUE);
    }
}

public class DubboInvoker<T> extends AbstractInvoker<T>{

    public boolean isAvailable() {
        if (!super.isAvailable()) {
            return false;
        }

        //当所有client均不能执行写操作时，便认为DubboInvoker实例不可用
        for (ExchangeClient client : clients) {
            if (client.isConnected() &&
!client.hasAttribute(Constants.CHANNEL_ATTRIBUTE_READONLY_KEY)) {
                //cannot write == not Available ?
                return true;
            }
        }
        return false;
    }
}

```

## 入站telnet命令处理

Dubbo支持使用telnet命令对服务端进行治理，但Client客户端是不支持的，因此入站数据为String类型时，需要判别当前是否为客户端。telnet命令支持也是在Exchanger信息交换层完成的，`public interface ExchangeHandler extends ChannelHandler, TelnetHandler`。

```

private static boolean isClientSide(Channel channel) {
    InetSocketAddress address = channel.getRemoteAddress();
    URL url = channel getUrl();
    return url.getPort() == address.getPort() &&
        NetUtils.filterLocalHost(url.getIp())
            .equals(NetUtils.filterLocalHost(address.getAddress().getHostAddress()));
}

@Override
public void received(Channel channel, Object message) throws RemotingException {
    ...
    if (isClientSide(channel)) {
        Exception e = new Exception("Dubbo client can not supported string message: "
            + message + " in channel: " + channel + ", url: " + channel.getUrl());
        logger.error(e.getMessage(), e);
    } else {
        String echo = handler.telnet(channel, (String) message);
        if (echo != null && echo.length() > 0) {
            //处理完telnet命令请求后，发回处理结果
            channel.send(echo);
        }
    }
    ...
}

```

## 初涉响应的同步转异步处理

上文中已经介绍过Dubbo中 `DefaultFuture` 是实现“同步 → 异步”的转换的关键支撑，主要是在网络I/O事件回调中根据当下响应状况构建响应 `Response`，而 `HeaderExchangeChannel` 则用于构建异步请求 `Request`，同时获得一个 `DefaultFuture` 实例，便于调用方使用回调 获取到结果。然而对于如何将彼端发到此端的请求做异步处理，避免因上级业务层繁重的任务处理导致阻塞却未有涉及，显然入站请求的处理的切入点还是 `ChannelHandler.received(Channel, Object)`，这正是 `HeaderExchangeHandler` 的主要职责之一，当然仅限于收到的正常且需要发回响应的 `Request`。具体实现如下：

```
void handleRequest(final ExchangeChannel channel, Request req) throws RemotingException {  
    ...  
    //获取彼端的请求体  
    Object msg = req.getData();  
    try {  
  
        //调用传入的ExchangeHandler实现的reply方法做响应处理  
        CompletionStage<Object> future = handler.reply(channel, msg);  
  
        //上级应用层完成结果处理后，根据响应情况，构建正常或异常响应  
        future.whenComplete((appResult, t) -> {  
            try {  
                if (t == null) {  
                    res.setStatus(Response.OK);  
                    res.setResult(appResult);  
                } else {  
                    res.setStatus(Response.SERVICE_ERROR);  
                    res.setErrorString(StringUtils.toString(t));  
                }  
  
                //在回调中使用底层通道Channel实现(NettyChannel)发回响应  
                channel.send(res);  
            } catch (RemotingException e) {  
                logger.warn("Send result to consumer failed, channel is " + channel + ", msg is " +  
e);  
            } finally {  
                // HeaderExchangeChannel.removeChannelIfDisconnected(channel);  
            }  
        });  
    } catch (Throwable e) {  
  
        //若上级应用层处理遇到异常，直接构建失败响应发送给彼端  
        res.setStatus(Response.SERVICE_ERROR);  
        res.setErrorString(StringUtils.toString(e));  
        channel.send(res);  
    }  
}
```

## 解码失败的入站请求及异常响应

网络交换过程中，始终存在一个二进制码流到Java实例数据转换的过程，也就是需要有编解码操作的存在，交换层负责将传输层的入站请求解码封装成 `Request` 对象。如果解码失败，并不会直接采用抛错形式告知上级的调用方，而依然是封装成 `Request` 对象，设置 `mBroken` 为true，因此在 `received` 后仅仅需要通过Channel通道发回异常响应便可。

同样，还有一种是由I/O中间件底层监视发现并驱动异常回调caught的情况，单向请求或者心跳事件都会被忽略。

```

void handleRequest(final ExchangeChannel channel, Request req)
    throws RemotingException {

    Response res = new Response(req.getId(), req.getVersion());
    if (req.isBroken()) {
        Object data = req.getData();

        String msg;
        if (data == null) {
            msg = null;
        } else if (data instanceof Throwable) {
            msg = StringUtils.toString((Throwable) data);
        } else {
            msg = data.toString();
        }
        res.setErrorMessage("Fail to decode request due to: " + msg);
        res.setStatus(Response.BAD_REQUEST);

        channel.send(res);
        return;
    }
    ...
}

@Override
public void caught(Channel channel, Throwable exception) throws RemotingException {
    if (exception instanceof ExecutionException) {
        ExecutionException e = (ExecutionException) exception;
        Object msg = e.getRequest();
        if (msg instanceof Request) {
            Request req = (Request) msg;
            if (req.isTwoWay() && !req.isHeartbeat()) {
                Response res = new Response(req.getId(), req.getVersion());
                res.setStatus(Response.SERVER_ERROR);
                res.setErrorMessage(StringUtils.toString(e));
                channel.send(res);
                return;
            }
        }
    }
    ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
    try {
        handler.caught(exchangeChannel, exception);
    } finally {
        HeaderExchangeChannel.removeChannelIfDisconnected(channel);
    }
}

```

## 出站处理

同样，出站的数据也包括了请求和响应，在同步转异步的实现中，`DefaultFuture` 需要判别请求是否已经发送出去，以便超时检测出现超时时能告知调用方超时是发生在此端还是彼端，这个操作时在 `sent` 网络I/O事件回调中完成的。

```

@Override
public void sent(Channel channel, Object message) throws RemotingException {
    //截获传入handler引发的异常
    Throwable exception = null;
    try {
        channel.setAttribute(KEY_WRITE_TIMESTAMP, System.currentTimeMillis());
        ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
        try {
            handler.sent(exchangeChannel, message);
        } finally {
            HeaderExchangeChannel.removeChannelIfDisconnected(channel);
        }
    } catch (Throwable t) {
        exception = t;
    }
    if (message instanceof Request) {
        Request request = (Request) message;
        DefaultFuture.sent(channel, request);
    }

    //将异常统一转换为RemotingException继续往上抛
    if (exception != null) {
        if (exception instanceof RuntimeException) {
            throw (RuntimeException) exception;
        } else if (exception instanceof RemotingException) {
            throw (RemotingException) exception;
        } else {
            throw new RemotingException(channel.getLocalAddress(),
                channel.getRemoteAddress(),
                exception.getMessage(), exception);
        }
    }
}
}

```

## ExchangeHandler

在上一章节中已经介绍过，当前网络传输层的核心职责是完成同步转异步的处理，异步发送到对方的请求表现在 ExchangeChannel 扩展通道接口上，是一种主动行为，而异步发送响应给对方则表现在 ExchangeHandler 这个接口上， ChannelHandler 的实现 HeaderExchangeHandler 在收到彼端 请求后，根据是否需要响应解析出 Object类型的 mData 作为参数后将具体的应答业务操作委托给 ExchangeHandler。

ExchangeHandler 接口定义如下：

```

public interface ExchangeHandler extends ChannelHandler, TelnetHandler {

    /**
     * reply.
     *
     * @param channel
     * @param request
     * @return response
     * @throws RemotingException
     */
    CompletableFuture<Object> reply(ExchangeChannel channel, Object request) throws
    RemotingException;

}

@SPI
public interface TelnetHandler {

    /**
     * telnet.
     *
     * @param channel
     * @param message
     */
    String telnet(Channel channel, String message) throws RemotingException;
}

```

## Replier & ReplierDispatcher

`ExchangeHandler` 实际上是为信息交换层提供了一个供上传直接调用的 `reply(ExchangeChannel, Object)` 方法，它屏蔽了本层的实现细节，连 `Request/Response` 的存在也不会让其意识到。从 `ExchangeHandler` 来看，可以针对任意类型的 `request` 做应答处理，也就是说于使用同一个 `o.a.d.r.Channel` 通道进行信息交换的 `Client ↔ Server` 对而言，以面向对象编程的视觉来看，实际上隐含了 `reply(ExchangeChannel, Object)` 方法可以应对各种不同数据类型的 `request` 入参的，也即可泛型化。因而Dubbo定义了如下的接口：

```

public interface Replier<T> {

    /**
     * reply.
     *
     * @param channel
     * @param request
     * @return response
     * @throws RemotingException
     */
    Object reply(ExchangeChannel channel, T request)
        throws RemotingException;
}

```

从上述 `reply()` 方法可以看出，方法签名已经发生了变化，出参不再是 `CompletableFuture<Object>`，也即当前层的异步被转换成了上层的同步。

为了应对这种泛型化，`派发器` 模型又再一次被搬上舞台，利用它来解决根据入参类型提供不同版本的 `Replier` 接口实现问题，框架层的 抽象层次更高，类结构更清晰，也化解了上层需要大量使用 `IF-ELSE` 的粗笨编码形式。

泛型化必然涉及到类型的处理，即需要提供 Class类型 到 Replier 接口实现的映射关系 Map<Class<?>, Replier<?>>，由于 Client ↔ Server 的信息交换是并发的，因而Dubbo使用了支持并发的 ConcurrentHashMap 作为其键值对容器，键值对关系不是强制需要的，可以提供默认通用版本的实现，像最初，都当做Object对待。

```
public class ReplierDispatcher implements Replier<Object> {  
  
    private final Replier<?> defaultReplier;  
  
    private final Map<Class<?>, Replier<?>> repliers = new ConcurrentHashMap<Class<?>, Replier<?>>();  
  
    public ReplierDispatcher() {  
        this(null, null);  
    }  
  
    public ReplierDispatcher(Replier<?> defaultReplier) {  
        this(defaultReplier, null);  
    }  
  
    public ReplierDispatcher(Replier<?> defaultReplier, Map<Class<?>, Replier<?>> repliers) {  
        this.defaultReplier = defaultReplier;  
        if (repliers != null && repliers.size() > 0) {  
            this.repliers.putAll(repliers);  
        }  
    }  
  
    public <T> ReplierDispatcher addReplier(Class<T> type, Replier<T> replier) {  
        repliers.put(type, replier);  
        return this;  
    }  
  
    public <T> ReplierDispatcher removeReplier(Class<T> type) {  
        repliers.remove(type);  
        return this;  
    }  
  
    private Replier<?> getReplier(Class<?> type) {  
        for (Map.Entry<Class<?>, Replier<?>> entry : repliers.entrySet()) {  
            if (entry.getKey().isAssignableFrom(type)) {  
                return entry.getValue();  
            }  
        }  
        if (defaultReplier != null) {  
            return defaultReplier;  
        }  
        throw new IllegalStateException("Replier not found, Unsupported message object: " + type);  
    }  
  
    @Override  
    @SuppressWarnings({"unchecked", "rawtypes"})  
    public Object reply(ExchangeChannel channel, Object request) throws RemotingException {  
        return ((Replier) getReplier(request.getClass())).reply(channel, request);  
    }  
}
```

**NOTE**

**ReplierDispatcher** 派发器只需要对一个Request做一次Response应答处理，是严格的一一对应关系，因此它不像本文涉及到的其它几个派发器，它是单选的。

## 通道监听者派发器 ExchangeHandlerDispatcher

同传输层一样，信息交换层同样存在一个派发器 **ExchangeHandlerDispatcher**，它是 **ExchangeHandler** 的装饰器实现，当然也是"ChannelHandler" 和"TelnetHandler"的装饰器实现，原因是 **interface ExchangeHandler extends ChannelHandler, TelnetHandler**，因此它将5个基础网络I/O事件回调委托给了更下层的 **ChannelHandlerDispatcher**，因而具体实现上就会对应存在如下的模板方法：

```
@Override  
public void XXX(Channel channel, ...) {  
    handlerDispatcher.XXX(channel, ...);  
}
```

JAVA

更进一步而言，该派发器 **ExchangeHandlerDispatcher** 实际上是组合了 **ChannelHandlerDispatcher** 派发器，不止如此，它还组合了派发器 **ReplierDispatcher** 和 **TelnetHandler** **TelnetHandler** 只有一种形式的存在，无需增加一个派发器实现。

```

public class ExchangeHandlerDispatcher implements ExchangeHandler {

    private final ReplierDispatcher replierDispatcher;

    private final ChannelHandlerDispatcher handlerDispatcher;

    private final TelnetHandler telnetHandler;

//=====
// 几个构造函数中，维telnetHandler一直无需变化
//=====

    public ExchangeHandlerDispatcher() {
        replierDispatcher = new ReplierDispatcher();
        handlerDispatcher = new ChannelHandlerDispatcher();
        telnetHandler = new TelnetHandlerAdapter();
    }

    public ExchangeHandlerDispatcher(Replier<?> replier) {
        replierDispatcher = new ReplierDispatcher(replier);
        handlerDispatcher = new ChannelHandlerDispatcher();
        telnetHandler = new TelnetHandlerAdapter();
    }

    public ExchangeHandlerDispatcher(ChannelHandler... handlers) {
        replierDispatcher = new ReplierDispatcher();
        handlerDispatcher = new ChannelHandlerDispatcher(handlers);
        telnetHandler = new TelnetHandlerAdapter();
    }

    public ExchangeHandlerDispatcher(Replier<?> replier, ChannelHandler... handlers) {
        replierDispatcher = new ReplierDispatcher(replier);
        handlerDispatcher = new ChannelHandlerDispatcher(handlers);
        telnetHandler = new TelnetHandlerAdapter();
    }

//=====
// 直接提供方法调用handlerDispatcher增删ChannelHandler实现
//=====

    public ExchangeHandlerDispatcher addChannelHandler(ChannelHandler handler) {
        handlerDispatcher.addChannelHandler(handler);
        return this;
    }

    public ExchangeHandlerDispatcher removeChannelHandler(ChannelHandler handler) {
        handlerDispatcher.removeChannelHandler(handler);
        return this;
    }

//=====
// 直接提供方法调用replierDispatcher增删Replier接口实现
//=====

    public <T> ExchangeHandlerDispatcher addReplier(Class<T> type, Replier<T> replier) {
        replierDispatcher.addReplier(type, replier);
        return this;
    }

    public <T> ExchangeHandlerDispatcher removeReplier(Class<T> type) {
        replierDispatcher.removeReplier(type);
        return this;
    }

    ...// 其它5个类似模板方法的I/O事件回调
}

```

```
@Override
@SuppressWarnings({"unchecked", "rawtypes"})
public CompletableFuture<Object> reply(ExchangeChannel channel, Object request) throws
RemotingException {
    return CompletableFuture.completedFuture(((Replier) replierDispatcher).reply(channel,
request));
}

@Override
public String telnet(Channel channel, String message) throws RemotingException {
    return telnetHandler.telnet(channel, message);
}

}
```

**NOTE** ExchangeHandlerDispatcher 作为装饰器就是一个接口实现，它不是必须的，如果没有提供，自然就不需要运行包括 ReplierDispatcher 在内的成套逻辑。

---

完结

# Dubbo RPC 之 Protocol协议层（一）

此前几乎所有的关于Dubbo实现的剖析文章均有提到，Dubbo是一个RPC通讯框架，直白一点说就是开发人员在进行分布式开发时，可以像调用本地方法一样对远端机器提供的服务发起调用。这个过程中所有关于网络编程的具体细节都被框架给封装了，对上面的应用层屏蔽了细节，如果开发人员不细究底层实现原理，基本会对其无感。Dubbo的RPC框架层担任的正是这个封装过程，将顶端应用层的本地调用转换为对底端远端层Remoting的远程调用，如下图所示，整个RPC框架层被细分成了7层，占据了整个Dubbo实现的很大篇幅，本文主要聚焦于阐述Protocol这一层的具体实现。

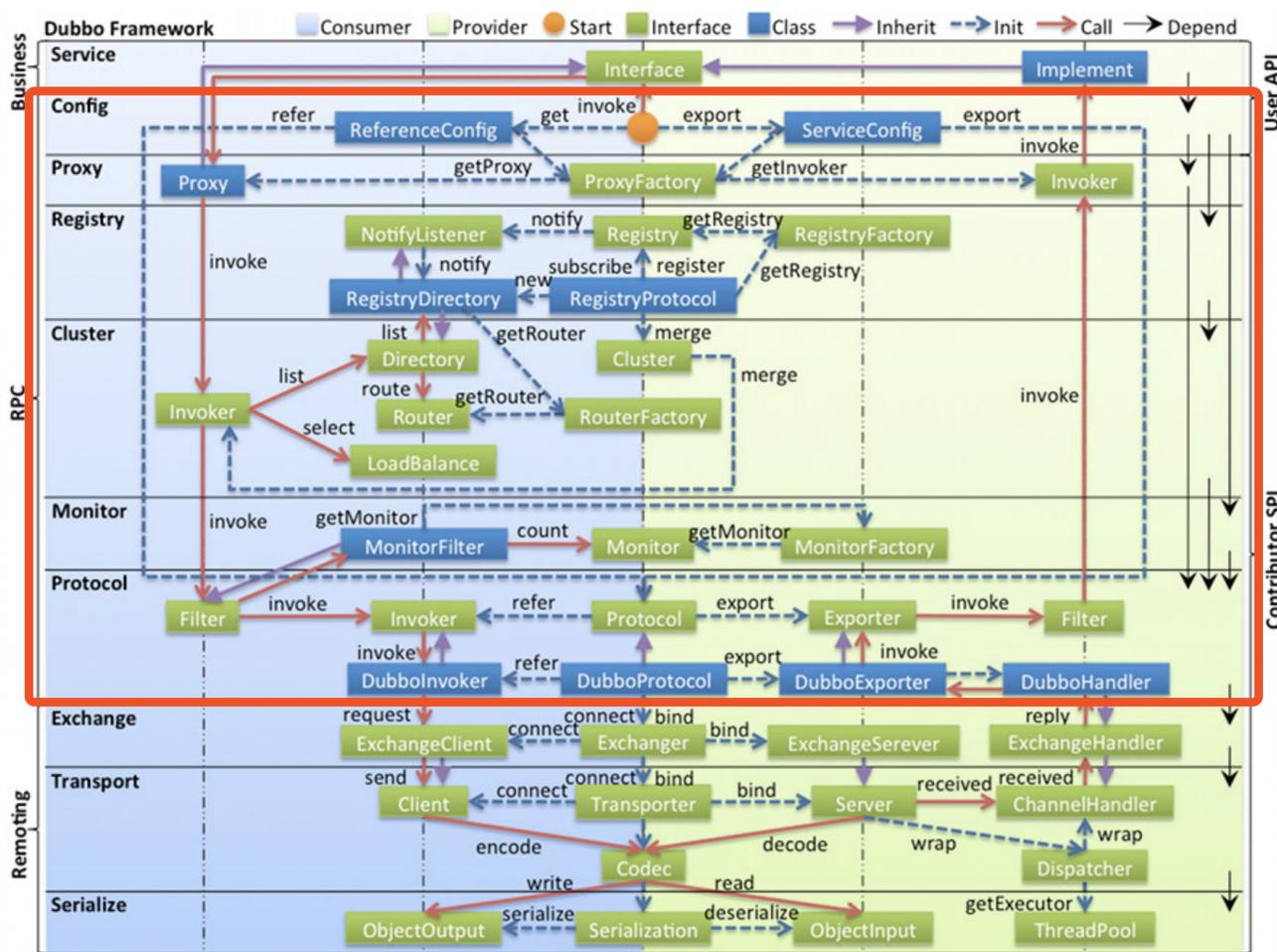


图: Dubbo Framework Protocol

## 协议层由来

两个实体若要能相互通信，就要约定某种共同的交流语言，就像两位不会讲彼此语言的小伙子要完成对话，他们可以约定用中文、英文或者其他语言交流，不会所用语言的一方或者两方，要么找个翻译，要么使用Google手机实时翻译。从这个场景中可以看出，在约定了交流语种的前提下，实际参与会话的实体不一定得懂该语种，可以通过第三方代理完成。在软件领域，这个用于完成对话的工具举例场景中的自然语言被认为是一种契约。

分布式应用开发中，所涉服务可能由多个团队完成，甚至用的都不是同一种编程语言，为了对付这种情况，Dubbo的RPC体系中专门提炼了一个可以扩展的 **Protocol** 协议层。众所周知，比如Http是被普遍支持的通讯协议，只要提供支持，跨语种的通讯就能被轻松解决。也就是说Dubbo中的远端请求实际上不限于网络请求，也可以是跨两个JVM的IPC进程间请求，Dubbo的典型特征是“微内核、可扩展”，**Protocol** 更是将这种风格体现得淋漓尽致，其内置的支持实现有：

- o.a.d.r.p.injvm. **InJvmProtocol**：用于同一个JVM中的客户端和服务端通讯
- o.a.d.r.p.dubbo. **DubboProtocol**：默认实现
- o.a.d.r.p.rmi. **RmiProtocol**：跨JVM的原生远程过程调用
- o.a.d.r.p.http. **HttpProtocol**：HTTP协议支持
- o.a.d.r.p.http.hessian. **HessianProtocol**

官网中有关于 **Protocol** 的介绍相当简洁，如下：

“**RPC** 协议扩展，封装远程调用细节。

简短一句话说明了其主要职责，从代码的角度看是这么一个过程——“客户端程序拥有一套服务端提供的服务接口 **interface**，而接口的具体实现在服务端，RPC将本地的接口调用转换为对服务端的出站请求，而协议层将远端请求有关的具体细节封装起来，当请求抵达服务端后，协议层将对应的入站请求转为对接口的调用，也就是说此时在服务端完成了接口实现的本地调用”。细究之，协议层完成的客户端出站请求封装和服务端入站请求封装是两个互为逆向的过程。

## 关键组件

从上文的表述知道，**Protocol** 实际要做的是一个转译操作，完成应用层中的原生的Java源码方法调用到网络层的远端通讯间的相互转换过程由通讯双方选定协议支持。我们都清楚框架始终是某类通用问题的解决方案，这个RPC肯定是面向所有Java能够表达的方法签名的。但尽管方法的签名表示有着万千变化，但从更大的粒度而言，实际上概括起来又只有出参、入参、方法名这3个基本元素包括出参入参类型，熟悉Java反射的人不难明白这点，简言之就是任何一个方法签名都可以转换成一个**Method**对象，更进一步讲这个过程不需要关心方法或者说业务接口的具体细节，剩下的就是如何解决在方法和远端请求入站和出站间进行转换的问题，它也是 **Protocol** 所关注的重点。

综上，Dubbo抽离出如下几个关键组件接口：

1. **Invocation & Result**: 分别用于封装原生Java代码中方法接口的中入参和出参，包括参数类型，二者组合起来便能表达一次方法调用。
2. **Invoker**: Dubbo中每一个 **Invoker** 实例代表一个服务，而后者对应一个 **interface** 接口，接口中的所有方法均可以用如下语句表达：

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

3. **Protocol**: 支持扩展，框架层中从底往上首次将服务的这个概念引入，类似 **Transporter**、**Exchanger** 等，处于Dubbo架构的中分线位置，是框架分层的一个概念，根据此前剖析文章不难看出，其实现是横跨客户端和服务端的。
- a. 客户端需要根据接口类和其配置总线参数获得一个 **Invoker** 实例，供上层将原生的Java接口调用转换为远端请求，因此 **Protocol** 有责任为其提供一个封装了网络层用于向彼端发送网络请求的 **ExchangeClient** 实例。
  - b. 服务端则需要完成服务的暴露处理，当来自客户端的 **Invoker** 的请求抵达时<sub>网络请求</sub>，Dubbo会将来自彼端的请求转换成 **Invocation** 对象，以便 **Protocol** 根据其中信息获得一个由Dubbo框架使用动态代理模式实现的 **Invoker** 实例，随后便可以使用该实例间接完成对业务接口实现的调用，获得响应结果后，经过网络层处理发回给客户端。

## 关键接口定义

在清楚了协议层的具体职责与设计初衷后，以下按照以往惯例，我们的分析仍然聚焦在Dubbo的默认支持实现上，跟上文顺序相反，由简单到复杂，逐步深入，本章节先大概介绍一下各个结合定义，下一章节则着重剖析具体实现。

### 服务接口入参封装体 **Invocation**

**Invocation** 用于封装Java服务接口中的入参，包括了入参类型和方法名称。《Dubbo远程通讯·网络传输层》一文中曾经提到一个特性，就是用于实际通讯的通道 **Channel** 可以像 **ThreadLocal** 一样拥有自己的上下文本地变量，当前框架层利用它可以存储一些用于框架功能增强或流程控制的参数，它的作用范围是绑定在 **Channel** 实例上，容易确保线程安全，保持了框架实现的优雅干净。同样 **Protocol** 协议层的表示每一次接口方法调用的 **Invocation** 也可以存取其本地变量，被视作附属参数 **attachment**。

```

public interface Invocation {
    //=====
    // 表征接Java接口中的入参，包括了入参类型和方法名称
    //=====
    String getMethodName();

    Class<?>[] getParameterTypes();

    Object[] getArguments();

    //=====
    // 用于存取接口调用的本地附属参数
    //=====
    Map<String, String> getAttachments();

    void setAttachment(String key, String value);

    void setAttachmentIfAbsent(String key, String value);

    String getAttachment(String key);

    String getAttachment(String key, String defaultValue);

    //=====
    // 获取实现当前Invocation调度的服务接口调度器Invoker
    //=====
    Invoker<?> getInvoker();

}

}

```

## 服务接口出参封装体 Result

**Result** 用于封装服务接口方法调用的结果，也就是出参，在Java中一个方法调用只会返回一个结果对象，**Object** 可以代表所有类型的对象，如果业务逻辑处理异常，会抛出 **XXXException**，其实它也可以被认为是另外一种形式的出参。虽然两种出参都可以统一为 **Object** 对象，但一般上框架会分别对待，有利于框架实现。

另外从《Dubbo远程通讯·信息交换层》已经知悉，Dubbo为了充分压榨硬件性能、确保很高的吞吐率，在信息交换层已经做了同步转异步的处理，因此对应到当前的协议层实现来说，RPC方法调用的返回结果也是需要异步获取的。这种异步实现是通过 **CompletableFuture<T> → CompletionStage<T>** 达成的，理论上其和 **Result** 是一种组合关系，由于二者实例之间的一对一的绑定关系，外加资源回收处理的便利性考量，Dubbo使用了接口扩展来绑定这种关系，这样一来 **Result** 成了一个行为类。

同样，在具体实现细节中，请求操作和等待响应操作实际上是两个相互独立的阶段，二者在时间发生有着严格的先后顺序，同请求阶段一样，它需要存取本地参数，但不应共享，因此接口中也定义了数个存取**Result**本地附属参数 **attachment** 的方法。

另外为了请求方的操作的便利性，比如说使用默认提供值同步获取结果，在调用方的上下文中以响应式获取结果。

```
public interface Result extends CompletionStage<Result>, Future<Result>, Serializable {  
  
    //=====  
    // 出参有两种类型，正常结果Object，抛出的异常Exception，处理是否正常需要使用 hasException 提前判断  
    //=====  
    Object getValue();  
  
    void setValue(Object value);  
  
    Throwable getException();  
  
    void setException(Throwable t);  
  
    boolean hasException();  
  
    /**  
     * Recreate.  
     * <p>  
     * <code>  
     * if (hasException()) {  
     * throw getException();  
     * } else {  
     * return getValue();  
     * }  
     * </code>  
     *  
     * @return result.  
     * @throws if has exception throw it.  
     */  
    Object recreate() throws Throwable;  
  
    //=====  
    // 用于存取接口调用的本地附属参数  
    //=====  
    Map<String, String> getAttachments();  
  
    void addAttachments(Map<String, String> map);  
  
    void setAttachments(Map<String, String> map);  
  
    String getAttachment(String key);  
  
    String getAttachment(String key, String defaultValue);  
  
    void setAttachment(String key, String value);  
  
    //=====  
    // 要求即时返回结果，若对方还未完成不会等到对方完成，使用提供的值作为Result结果  
    //=====  
    /**  
     * Returns the specified {@code valueIfAbsent} when not complete, or  
     * returns the result value or throws an exception when complete.  
     *  
     * @see CompletableFuture#getNow(Object)  
     */  
    Result getNow(Result valueIfAbsent);
```

```

//=====
// 使用响应式编程在回调中获取对端的处理结果，调用方在调用点持有自己的上下文，便于业务处理
// NOTE: 如名称所示，该方法在回调时，Dubbo会确保它拥有和此前原生方法调用时的上下文信息
//=====

    /**
     * Add a callback which can be triggered when the RPC call finishes.
     * <p>
     * Just as the method name implies, this method will guarantee the callback
     * being triggered under the same context as when the call was started,
     * see implementation in {@link Result#whenCompleteWithContext(BiConsumer)}
     *
     * @param fn
     * @return
     */
    Result whenCompleteWithContext(BiConsumer<Result, Throwable> fn);

    default CompletableFuture<Result> completionFuture() {
        return toCompletableFuture();
    }
}

```

## 服务接口调度器 Invoker

服务接口调度器，其作用上文已经介绍过，其实例和服务提供者接口定义是一对一的，通过接口类对象绑定，因此其定义中声明了一个 `getInterface()` 方法。另外一个 `Invoker` 对象代表了一个微服务，作为微服务它有着自己的生命周期和配置参数<sub>微服务的元数据</sub>，因此接口扩展自 `Node`，使用配置总线`URL`处理配置的存取问题。

当然 `Invoker` 的具体实现上很灵活，就像前面的剖析文章中提到的 `ChannelHandler`，以它为起点实现了微服务的许多其它特性。

```

public interface Invoker<T> extends Node {

    /**
     * get service interface.
     *
     * @return service interface.
     */
    Class<T> getInterface();

    /**
     * invoke.
     *
     * @param invocation
     * @return result
     * @throws RpcException
     */
    Result invoke(Invocation invocation) throws RpcException;

}

```

## 可扩展协议接口 Protocol

之所以把 **Protocol** 这个最为关键的接口放在最后才介绍，单就其官方给定的下面接口文档，在不熟悉实现细节的和设计原理时，理解起来相当费劲。在有了上面的那些铺垫后，再回来理解文档中要表达的意图就比较容易。

```
public interface Protocol {  
    /**  
     * 暴露远程服务: <br>  
     * 1. 协议在接收请求时，应记录请求来源方地址信息：  
     RpcContext.getContext().setRemoteAddress();<br>  
     * 2. export()必须是幂等的，也就是暴露同一个URL的Invoker两次，和暴露一次没有区别。<br>  
     * 3. export()传入的Invoker由框架实现并传入，协议不需要关心。<br>  
     *  
     * @param <T> 服务的类型  
     * @param invoker 服务的执行体  
     * @return exporter 暴露服务的引用，用于取消暴露  
     * @throws RpcException 当暴露服务出错时抛出，比如端口已占用  
     */  
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;  
  
    /**  
     * 引用远程服务: <br>  
     * 1. 当用户调用refer()所返回的Invoker对象的invoke()方法时，协议需相应在远端执行export()的入  
     参Invoker对象的invoke()方法，对应关系是两端的Invoker对象同URL。<br>  
     * 2. refer()返回的Invoker由协议实现，协议通常需要在此Invoker中发送远程请求。<br>  
     * 3. 当url中有设置check=false时，连接失败不能抛出异常，需内部自动恢复。<br>  
     *  
     * @param <T> 服务的类型  
     * @param type 服务的类型  
     * @param url 远程服务的URL地址  
     * @return invoker 服务的本地代理  
     * @throws RpcException 当连接服务提供方失败时抛出  
     */  
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;  
}
```

## 协议支持实现

上文曾说Dubbo在协议层提供了多种通讯方式实现，本文暂只就默认支持的Dubbo版和Injvm版的实现加以剖析。为便于各种版本协议的实现，Dubbo封装了一些基础的逻辑，本章节先就这些细节展开。

## RpcInvocation

**RpcInvocation** 是 **Invocation** 的实现。总体上实现比较简单，只需要根据接口要求能够表达一次方法的几个基本元素就足够，因此 **RpcInvocation** 对应定义了如下几个属性：

1. String **methodName**：实例所代表方法名称。

2. `Class<?>[] parameterTypes` : 入参类型, 数组, 和`arguments`严格一一对应。
3. `Object[] arguments` : 具体入参数数据。
4. `Map<String, String> attachments` : 附属参数, 由于`Invocation`只对应一次方法调用, 并没有存在资源争用的情况, 普通`Map`就足够。
5. `transient Invoker<?> invoker` : `Invoker`调度器引用, 后者属行为类, 因而被声明为`transient`。

上文已经说过, `Invocation` 和 Java 原生程序中的方法调用是一对一的关系, 如下构造方法便印证了这一点, 从 `Method` 对象中获取到方法的名称和参数类型。另外由于它代表是一次具体的PRC方法调用而不是一个普通的本地方法调用, 因此还需要加入一个无法从 `Method` 对象中获取的入参 `Object[] arguments`。

```

private transient Class<?> returnType;

private transient InvokeMode invokeMode;

public RpcInvocation(String methodName, Class<?>[] parameterTypes,
    Object[] arguments, Map<String, String> attachments, Invoker<?> invoker) {
    this.methodName = methodName;
    this.parameterTypes = parameterTypes == null ? new Class<?>[0] : parameterTypes;
    this.arguments = arguments == null ? new Object[0] : arguments;
    this.attachments = attachments == null ? new HashMap<String, String>() :
attachments;
    this.invoker = invoker;
}

public RpcInvocation(Invocation invocation, Invoker<?> invoker) {
    this(invocation.getMethodName(), invocation.getParameterTypes(),
        invocation.getArguments(), new HashMap<String, String>
(invocation.getAttachments()),
        invocation.getInvoker());

//=====
// 将微服务配置元数据信息设到Invocation的本地参数容器中
//=====

if (invoker != null) {
    URL url = invoker.getUrl();
    setAttachment(PATH_KEY, url.getPath());
    if (url.hasParameter(INTERFACE_KEY)) {
        setAttachment(INTERFACE_KEY, url.getParameter(INTERFACE_KEY));
    }
    if (url.hasParameter(GROUP_KEY)) {
        setAttachment(GROUP_KEY, url.getParameter(GROUP_KEY));
    }
    if (url.hasParameter(VERSION_KEY)) {
        setAttachment(VERSION_KEY, url.getParameter(VERSION_KEY, "0.0.0"));
    }
    if (url.hasParameter(TIMEOUT_KEY)) {
        setAttachment(TIMEOUT_KEY, url.getParameter(TIMEOUT_KEY));
    }
    if (url.hasParameter(TOKEN_KEY)) {
        setAttachment(TOKEN_KEY, url.getParameter(TOKEN_KEY));
    }
    if (url.hasParameter(APPLICATION_KEY)) {
        setAttachment(APPLICATION_KEY, url.getParameter(APPLICATION_KEY));
    }
}
}

public RpcInvocation(Method method, Object[] arguments, Map<String, String> attachment)
{
    this(method.getName(), method.getParameterTypes(), arguments, attachment, null);
    this.returnType = method.getReturnType();
}

```

上述源码片段中展示了其中一个构造方法中，Dubbo有将在Invoker实例设入的微服务配置元数据作为附属参数设置到Invocation中去，采用冗余手段，以空间换时间，可以快速便捷的拿到相关上下文参数，基于优先原则访问这些值，只有发现对应键值在附属参数不存在时，才绕道Invoker实例的配置总线获取。

另外它还呈现了两个声明为 `transient` 的变量，前者表达式 `Method` 对象出参的类型，后者这表示当前执行上下文中 `Invocation` 以何种方式 `sync`、`async`、`future` 调度的，大部分时候是前者决定了后者的值。

## 由 `Invocation` 获取出参

根据 `Invocation` 这个接口的特性，它是用于表征状态类的，可被持久化，而对应调用方法的出参不是其关注重点。只有 `RpcInvocation` 加入了持有 `Class<?>` `returnType` 属性，为了尽可能获取到 `Invocation` 对象的出参类型，因此Dubbo在 `RpcUtils` 对应定义了如下一个方法，深入其细节会发现Invoker实例所表征的服务会将对应接口的名称以 `interface` 为键值存入到配置总线URL中，Dubbo根据该类名获取到服务的类对象，再由 `Invocation` 实例中的方法名称和入参类型获取到 `Method` 实例，以根据它进一步获取到出参类型，出参为 `void` 或者不满足源码过滤条件的都视作出参类型为 `null`。

```

public class RpcUtils {

    ...
    public static Class<?> getReturnType(Invocation invocation) {
        try {
            if (invocation != null && invocation.getInvoker() != null
                && invocation.getInvoker().getUrl() != null
                && !invocation.getMethodName().startsWith("$")) {

                String service = invocation.getInvoker()
                    .getUrl().getServiceInterface();
                if (StringUtils.isNotEmpty(service)) {
                    Class<?> invokerInterface = invocation
                        .getInvoker().getInterface();
                    Class<?> cls = invokerInterface != null ?
                        ReflectUtils.forName(invokerInterface.getClassLoader(),
service)
                            : ReflectUtils.forName(service);

                    Method method = cls.getMethod(invocation.getMethodName(),
                        invocation.getParameterTypes());

                    if (method.getReturnType() == void.class) {
                        return null;
                    }
                    return method.getReturnType();
                }
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
            return null;
        }
        ...
    }
}

```

## 由 Invocation 获取方法调度模式

Dubbo可以根据出参类型和总线、附属参数等知晓当前被调用RPC方法的是被何种模式调度的，总共3种调度模式：如果接口方法本身的出参是CompletableFuture类型的则为 FUTURE 模式，如果配置参数设了 `async` 标识则为 ASYNC 异步模式，否则便是同步 SYNC 同步模式。

在进一步了解具体实现细节前，需要了解下的是Dubbo中有两个以"\$"字母打头的特殊方法，分别名为 "\$invoke" 和 "\$invokeAsync"，目前只需要了解其接口定义和应用场景，具体实现将在下文中深入阐述：

```

/**
 * Generic service interface
 *
 * @export
 */
public interface GenericService {

    //Method name, e.g. findPerson. If there are overridden methods,
    //parameter info is required, e.g. findPerson(java.lang.String)
    Object $invoke(String method, String[] parameterTypes, Object[] args)
        throws GenericException;

    default CompletableFuture<Object> $invokeAsync(String method,
        String[] parameterTypes, Object[] args) throws GenericException {

        Object object = $invoke(method, parameterTypes, args);
        if (object instanceof CompletableFuture) {
            return (CompletableFuture<Object>) object;
        }

        return CompletableFuture.completedFuture(object);
    }
}

```

“泛接口调用方式主要用于客户端没有API接口及模型类元的情况，参数及返回值中的所有POJO均用Map表示，通常用于框架集成，比如：实现一个通用的服务测试框架，可通过*GenericService*调用所有服务实现。

也就是说此场景下客户端并不需要维护和同步微服务接口签名（包括方法、入参、出参以及出入参类型相关的定义），只需要提供“方法名、入参、出参、出入参类型”这几个元素即可，此时有 `methodName ∈ ["$invoke", "$invokeAsync"]`。而以字符串直接提供的方法名会被作为 `arguments` 中的第一个元素。Dubbo中，微服务的配置也全部在配置总线URL中体现，是可以配置到方法这一级别的，可以设置 `url["$invoke.async"] = true` 告知Dubbo需要异步调度该泛接口。

```
public class RpcUtils {  
    ...  
  
    public static boolean isReturnTypeFuture(Invocation inv) {  
        Class<?> clazz;  
        if (inv instanceof RpcInvocation) {  
            clazz = ((RpcInvocation) inv).getReturnType();  
        } else {  
            clazz = getReturnType(inv);  
        }  
        //出参类型为CompletableFuture则为FUTURE模式  
        return (clazz != null && CompletableFuture.  
            class.isAssignableFrom(clazz)) || isGenericAsync(inv);  
    }  
  
    public static boolean isGenericAsync(Invocation inv) {  
        return $INVOKE_ASYNC.equals(inv.getMethodName());  
    }  
  
    public static InvokeMode getInvokeMode(URL url, Invocation inv) {  
        if (isReturnTypeFuture(inv)) {  
            return InvokeMode.FUTURE;  
        } else if (isAsync(url, inv)) {  
            return InvokeMode.ASYNC;  
        } else {  
            return InvokeMode.SYNC;  
        }  
    }  
  
    //先从附属参数获取异步设置参数，如果值为false，则配置总线中进一步获取方法参数  
    public static boolean isAsync(URL url, Invocation inv) {  
        boolean isAsync;  
        if (Boolean.TRUE.toString().equals(inv.getAttachment(ASYNC_KEY))) {  
            isAsync = true;  
        } else {  
            isAsync = url.getMethodParameter(getMethodName(inv), ASYNC_KEY, false);  
        }  
        return isAsync;  
    }  
  
    //范接口的方法名称为``$invoke``或``$invokeAsync``，指定方法名称置于入参中第一个位置  
    public static String getMethodName(Invocation invocation) {  
        if ($INVOKE.equals(invocation.getMethodName())  
            && invocation getArguments() != null  
            && invocation getArguments().length > 0  
            && invocation getArguments()[0] instanceof String) {  
            return (String) invocation getArguments()[0];  
        }  
        return invocation.getMethodName();  
    }  
    ...  
}
```

```
public enum InvokeMode {  
    SYNC, ASYNC, FUTURE  
}
```

---

## Result 的实现 AsyncRpcResult

注：下文中反复出现的 `RpcContext` 相当关键，使用Java中`ThreadLocal`的翻版实现`InternalThreadLocal`，同一个变量，使用它的多个线程各自拥有一份拷贝，也即所谓的线程本地变量，综合了性能等方面的考虑因素。这一大章节先不细究其实现，只需知道其存在价值和用法即可。

### 理论分析

`AsyncRpcResult` 是 `Result` 的接口实现，上文已经交代过，底层采用异步调用方式处理远端请求，也即请求发送出去就返回了，资源已经让渡出去，待服务端完成业务处理再经网络回传响应结果，最后由Dubbo执行反序列化处理，将结果扔进一个`Result`对象返回给应用层。稍加思考便会产生如是疑问，服务端的响应回来之后，如何获得对应的表示原生请求的 `Invocation` 对象以及它被调度时的上下文环境信息？为获得结果，我们得继续往下深入。

先看看如下关于 `AsyncRpcResult` 的文档信息：

**“** This class represents an unfinished RPC call, it will hold some context information for this call, for example `RpcContext` and `Invocation`, so that when the call finishes and the result returns, it can guarantee all the contexts being recovered as the same as when the call was made before any callback is invoked.

As `Result` implements `CompletionStage`, `AsyncRpcResult` allows you to easily build a async filter chain whose status will be driven entirely by the state of the underlying RPC call.

`AsyncRpcResult` does not contain any concrete value (except the underlying value bring by `CompletableFuture`), consider it as a status transfer node. `getValue()` and `getException()` are all inherited from `Result` interface, implementing them are mainly for compatibility consideration. Because many legacy Filter implementation are most possibly to call `getValue` directly.

上文中前段的大意是“`AsyncRpcResult` 对象代表了一个未完成的RPC调用，它持有该调用中包括 `RpcContext` 和 `Invocation` 在内的上下文信息，因而当完成调用结果返回时，能够保证完好如初地就地恢复请求发出后还未被执行任何回调时的所有上下文信息”。这初步解开了我们上述发现的疑团，当然更加具体还得等接下来深入剖析其实现细节。

## NOTE

恢复的时机是“请求发出后还未被执行任何回调时”，  
`RpcContext` 这个表征执行上下文的对象，可能在回调之前其内容已经发生了改变，原因是这期间同一个线程可能被用于执行其他的RPC调用，因而需要保存方法调度时的 `RpcContext` 引用，便于在回调前刹那恢复现场信息。

中段则表明“由于 `AsyncRpcResult` 实现了 `CompletionStage` 接口，因而可以非常容易地构建一条异步过滤器链，其状态将完全由底层RPC调用的状态驱动”。

最后一段给出的信息则尤为关键，基本意思是 `AsyncRpcResult` 本不应该持有除 `CompletableFuture` 携带外的任何具体值，应该把它当做一种状态传输节点。历史原因，为了兼容性实现了 `getValue()` 和 `getException()` 方法，依然还有许多遗留的过滤器使用这两个方法。

## 实现细节

从上述一个章节介绍知悉，`AsyncRpcResult` 最重要的工作便是现场维护，其实现需要的几个如下组成元素：

1. `Invocation`：代表原生方法的一次调用，携带了入参、入参类型、以及方法名，不过上文已经说明，它是历史遗留，它的存在更多是兼容性考虑；
2. `RpcContext stored`：原生方法被调度时的上下文信息，保留现场；
3. `RpcContext tmp`：原生方法被回调时的，所运行线程持有的其原生方法正在执行时的上下文信息；

## 现场信息保护和恢复

在具体介绍其实现之前，需要先回过头去看看此前 `Result` 中定义的一个关键回调方法 `whenCompleteWithContext`，该方法也正是异步过滤器链实现的关键，下文将有所体现。现场信息保护的原理是 `AsyncRpcResult` 被实例化时，会缓存当时的RPC调用时的 `RpcContext` 上下文A原样保留现场信息，结果返回执行回调时，会再一次获取当前线程和 `Invocation` 调度时的线程可能不同中持有的 `RpcContext` 上下文B，接着将B缓存到一个 `tmpContext` 的临时变量中，随后将A的内容恢复到当前线程中，也就是替换掉其现有的内容B，此后才回调类型为 `BiConsumer<Result, Throwable>` 的入参函数，最后使用 `tmpContext` 恢复当前线程回调时的上下文信息，也即还原到B。

具体实现源码如下，奇怪的是源码中会有两套配对的 `RpcContext`，一个被称之为 `context`，另一个为 `serverContext`，为啥会这样，得等后面对实现有更深入的了解。

```
public class AsyncRpcResult extends AbstractResult {  
  
    ...  
    private RpcContext storedContext;  
    private RpcContext storedServerContext;  
    private Invocation invocation;  
  
    public AsyncRpcResult(Invocation invocation) {  
        this.invocation = invocation;  
        this.storedContext = RpcContext.getContext();  
        this.storedServerContext = RpcContext.getServerContext();  
    }  
  
    public AsyncRpcResult(AsyncRpcResult asyncRpcResult) {  
        this.invocation = asyncRpcResult.getInvocation();  
        this.storedContext = asyncRpcResult.getStoredContext();  
        this.storedServerContext = asyncRpcResult.getStoredServerContext();  
    }  
  
    @Override  
    public Result whenCompleteWithContext(BiConsumer<Result, Throwable> fn) {  
        CompletableFuture<Result> future = this.whenComplete((v, t) -> {  
            beforeContext.accept(v, t);  
            fn.accept(v, t);  
            afterContext.accept(v, t);  
        });  
        //=====  
        //下半段代码：订阅当前对象的完成事件，第一时间获取其结果，薪火相传  
        //=====  
        AsyncRpcResult nextStage = new AsyncRpcResult(this);  
        nextStage.subscribeTo(future);  
        return nextStage;  
    }  
  
    public void subscribeTo(CompletableFuture<?> future) {  
        future.whenComplete((obj, t) -> {  
            if (t != null) {  
                this.completeExceptionally(t);  
            } else {  
                this.complete((Result) obj);  
            }  
        });  
    }  
  
    /**  
     * tmp context to use when the thread switch to Dubbo thread.  
     */  
    private RpcContext tmpContext;  
    private RpcContext tmpServerContext;  
  
    private BiConsumer<Result, Throwable> beforeContext = (appResponse, t) -> {  
        tmpContext = RpcContext.getContext();  
        tmpServerContext = RpcContext.getServerContext();  
        RpcContext.restoreContext(storedContext);  
        RpcContext.restoreServerContext(storedServerContext);  
    };
```

JAVA

```
private BiConsumer<Result, Throwable> afterContext = (appResponse, t) -> {
    RpcContext.restoreContext(tmpContext);
    RpcContext.restoreServerContext(tmpServerContext);
};

...
}
```

## subscribeTo & whenCompleteWithContext

上述源码中展示了一个比较独特的方法——`subscribeTo(CompletableFuture<?>)`，这段代码理解起来还是比较费劲，接下来的章节慢慢分析之。

为理解方便，我们假设一个现实生活的中的场景，读者给报社下了一份订单——`subscribeTo`，要求订阅时尚杂志，后者在新一期杂志出来后时，总会及时给读者快速邮递最新的杂志。这里读者是订阅方，而报社是被订阅方，下面分析将被直接表示为 读者 和 报社。

首先暂且将其之前的`whenCompleteWithContext` 先放一边。大概意思是入参`future` 所代表的这个`CompletableFuture` 类型对象动作完成时，会回调其`whenComplete` 方法，回调代码块所做的事情就是将其结果设给当前的`AsyncRpcResult` 对象，通知其完成。实质也就是`AsyncRpcResult` 这个`future` 对象的结果值是由另外一个`CompletableFuture` 在其完成时才填充的，也即只有后者的完成了其操作，通过回调将其获得的值传递给前者。前提是入参`future` 的返回值也是要求是`Result` 类型，这便有构成了一种形如“ $A \leftarrow B$ ”链式操作， $B$ 订阅了 $A$ ，只有 $B$ 完成其自身的设值操作，才会在其回调将值传递给 $A$ ，知会 $A$ 整个链式操作完成，表面上 $B$ 是依赖 $A$ 的，但 $A$ 只有在 $B$ 完成时，才能完成其闭环操作。

综上，也就是“读者向报社发起`subscribeTo` 订单申请，报社完成最新一期的杂志印制工作后，及时邮递给读者，也即在其完成回调事件中完成成果交付，这时候读者便拥有了最新杂志。”

现在回到`whenCompleteWithContext`，聚焦于后半段代码，理解它的关键是要搞清楚对象间关系，其调用`subscribeTo` 方法的不是当前`this` 对象，而是一个新创建的`nextStage` 实例，该实例会被`whenCompleteWithContext` 的调用方引用，当前`this` 对象完成操作后，会回调其`whenComplete`，将其完成值填充给`nextStage` 实例，后者也是一个`CompletableFuture` 类型对象，因此可以在其`whenComplete` 回调方法得知最终响应结果。

依然我们假设“读者：当前对象，报社：实际`Result` 完成方，借阅方：返回的`nextStage`”，借阅方向读者发起`subscribeTo`，要求其在读完杂志后第一时间给他借阅，读者也发起`subscribeTo` 订阅操作，向报社订购杂志，杂志的传递关系很明显：“借阅方  $\leftarrow$  读者  $\leftarrow$  报社”。

到这里就很明显了，在链条上每调用一次`whenCompleteWithContext` 实际上就是产生一个借书的借阅方，而每调用一次`subscribeTo` 操作，则是增加一个杂志生产方——报社，越往后的越趋向杂志的生产源头。

## NOTE

CompletableFuture 的 whenComplete 方法可以回调多次，按顺序依次回调。

## AppResponse

上文中已提及 AsyncRpcResult 是一个行为类，理论上不应该存储具体的状态值，由于历史原因需维持兼容性，Dubbo将接口 Result 定义的其它接口全部委托给了 AppResponse，而后者是一个状态类，理论上只需存取状态值即可，同样是因为兼容原因，继承实现了同样实现了 Result 接口。可以这么认为“`public class AsyncRpcResult implements CompletionStage<AppResponse>`”，由于只需要存储状态值，它的实现很简单，同样 AsyncRpcResult 中委托它实现状态值存取的方法实现也会比较简单，唯一值得一看的是下述的 `recreate()` 方法源码，其它具体请查看Dubbo源码。

```
@Override  
public Object recreate() throws Throwable {  
    if (exception != null) {  
        // fix issue#619  
        try {  
            // get Throwable class  
            Class clazz = exception.getClass();  
            while (!clazz.getName().equals(Throwable.class.getName())) {  
                clazz = clazz.getSuperclass();  
            }  
            // get stackTrace value  
            Field stackTraceField = clazz.getDeclaredField("stackTrace");  
            stackTraceField.setAccessible(true);  
            Object stackTrace = stackTraceField.get(exception);  
            if (stackTrace == null) {  
                exception.setStackTrace(new StackTraceElement[0]);  
            }  
        } catch (Exception e) {  
            // ignore  
        }  
        throw exception;  
    }  
    return result;  
}
```

有过Java开发经验的都知道，Java抛出的异常信息是带有当前方法所在线程的调用帧信息的，也即 `StackTrace`，它的作用是辅助排查问题，随着异常在一直往上抛的过程中，其涉及帧信息也逐个在增加，最终呈现的异常信息会很长。另外如果服务提供端报了大量的 NPE 异常，JVM 为了性能会做优化，会重新编译，不再打印异常堆栈，只会抛出预定义的NPE异常`java.lang.NullPointerException: null`。这样就导致真正的异常信息被隐藏了，因此上述源码对这种情况做进一步处理了，首先找到当前异常对象的`Throwable`下一级的顶层类，然后获取 `StackTrace` 帧信息，若值为`null`，则设值为 `new StackTraceElement[0]`，获得效果是“只要`StackTrace!=null`，就不会用错误的异常栈来赋值给 `StackTrace`，这样修改后，consumer会跟provider保持一致，即抛出没有异常栈的异常，这样就不会误导用户了”，具体问题请查看[Dubbo调用端hessian反序列化抛NPE问题研究](#) (<https://www.yuque.com/fa902k/id5z6r/sr041v>)。

## AbstractInvoker<T>

有了上述章节的铺垫后，`Invoker` 的实现就比较好理解了，下面我们一步步来加以分析。

### 基础实现

上述已经讲过：1) 一个 `Invoker` 实例实际上对应了Java中的一个服务接口，需要知晓当前实例所对应的接口类名；2) 每个微服务都有自身的配置参数，配置使用配置总线URL进行存取，效率上的考虑，Dubbo采用了时空互换，将微服务配置元数据设到了附属参数中；3) 微服务被认为是一个 `Node` 节点，有配置参数，能获取可用状态，可销毁处理。

这些实现都放在基类 `AbstractInvoker` 中：

```
public abstract class AbstractInvoker<T> implements Invoker<T> {  
    ...  
  
    //=====  
    // 时空交换, 将微服务的配置总线中配置元数据置入附属参数中  
    //=====  
    private final Map<String, String> attachment;  
  
    private static Map<String, String> convertAttachment(URL url, String[] keys) {  
        if (ArrayUtils.isEmpty(keys)) {  
            return null;  
        }  
        Map<String, String> attachment = new HashMap<String, String>();  
        for (String key : keys) {  
            String value = url.getParameter(key);  
            if (value != null && value.length() > 0) {  
                attachment.put(key, value);  
            }  
        }  
        return attachment;  
    }  
  
    //=====  
    // 对应微服务实例的类名信息, 唯一标识一个微服务  
    //=====  
    private final Class<T> type;  
  
    @Override  
    public Class<T> getInterface() {  
        return type;  
    }  
  
    //=====  
    // 微服务配置总线  
    //=====  
    private final URL url;  
  
    @Override  
    public URL getUrl() {  
        return url;  
    }  
  
    //=====  
    // 微服务可用状态, 使用volatile, 确保在并发多线程的可见性  
    //=====  
    private volatile boolean available = true;  
  
    @Override  
    public boolean isAvailable() {  
        return available;  
    }  
  
    protected void setAvailable(boolean available) {  
        this.available = available;  
    }  
}
```

JAVA

```

//=====
// 微服务被销毁了才认为不可用，使用原子变量确保多线程环境操作的安全性
//=====

private AtomicBoolean destroyed = new AtomicBoolean(false);

@Override
public void destroy() {
    if (!destroyed.compareAndSet(false, true)) {
        return;
    }
    setAvailable(false);
}

public boolean isDestroyed() {
    return destroyed.get();
}

...
}

```

## Result invoke(Invocation) 实现

Invoker 的每一种具体实现实际对应了一种协议，本文中讨论的是默认的Dubbo协议支持。Dubbo 将其中的一些模板实现代码放在父类 `AbstractInvoker<T>` 中，而和具体协议相关的实现委托给子类进一步实现，因此申明了如下的抽象方法：

```
protected abstract Result doInvoke(Invocation invocation) throws Throwable;
```

JAVA

以下着重分析以下模板代码中实现的逻辑，过程也比较简单，步骤如下：

- ① 给代表原生方法调用的 `Invocation` 实例inv关联其所处运行环境——当前微服务实例；
- ② 按约定给inv的附属数据容器中置入微服务配置元数据；
- ③ 将当前线程运行环境设置的上下文参数加入到inv的附属参数容器中；
- ④ 设置原生方法最终被调度时所采用的模式：Future or Async or Sync；
- ⑤ 给异步模式执行下的inv设置递增的唯一ID编号；
- ⑥ 执行具体协议实现子类提供的 `doInvoke()` 实现，如果调用期间遇到异常，则返回一个异常完成的 `AsyncRpcResult` 对象；

```
public abstract class AbstractInvoker<T> implements Invoker<T> {
```

JAVA

```
    ...

    public Result invoke(Invocation inv) throws RpcException {
        //(-) if invoker is destroyed due to address refresh from registry, let's allow
        the current invoke to proceed
        if (destroyed.get()) {
            logger.warn("Invoker for service " + this + " on consumer " +
NetUtils.getLocalHost() + " is destroyed,
                    + ", dubbo version is " + Version.getVersion() + ", this invoker
should not be used any longer");
        }

        //①
        RpcInvocation invocation = (RpcInvocation) inv;
        invocation.setInvoker(this);

        //②
        if (CollectionUtils.isNotEmptyMap(attachment)) {
            invocation.addAttachmentsIfAbsent(attachment);
        }

        //③
        Map<String, String> contextAttachments =
        RpcContext.getContext().getAttachments();
        if (CollectionUtils.isNotEmptyMap(contextAttachments)) {
            //(-) invocation.addAttachmentsIfAbsent(context){@link RpcInvocatio ...
            invocation.addAttachments(contextAttachments);
        }

        //④
        invocation.setInvokeMode(RpcUtils.getInvokeMode(url, invocation));
        //⑤
        RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);

        //⑥
        try {
            return doInvoke(invocation);
        } catch (InvocationTargetException e) { // biz exception
            Throwable te = e.getTargetException();
            if (te == null) {
                return AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
            } else {
                if (te instanceof RpcException) {
                    ((RpcException) te).setCode(RpcException.BIZ_EXCEPTION);
                }
                return AsyncRpcResult.newDefaultAsyncResult(null, te, invocation);
            }
        } catch (RpcException e) {
            if (e.isBiz()) {
                return AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
            } else {
                throw e;
            }
        } catch (Throwable e) {
            return AsyncRpcResult.newDefaultAsyncResult(null, e, invocation);
        }
    }
```

```
    }  
}  
  
...  
}
```

可以看出涉及过程虽然比较多，但理解起来不算费劲，但有些点需要单独拎出来说道说道的，其一是两处注解说明，其二是使用到的外部工具方法。

## 特别注释

代码中出现的两处注解后的代码行细究起来有点违反直觉让人费解，在注解的基础上才能和其运行上下文环境联系起来，大致意思分别是：

- (一) 由于注册中心刷新了地址，导致微服务实例被销毁，此时对应的 `Invocation` 还是允许在这个微服务环境Invoker实例下完成调度的。
- (二) 当前 `invoke()` 方法在Dubbo的重试机制下会被再次调用，此时会有过滤器Filter调用 `RpcContext.setAttachment(String, String)` 设置上下文参数。如果调用 `addAttachmentsIfAbsent` 存留的还是旧的上下文参数，可能已经失效，如外发的 `traceId` 和 `spanId`。

## 所涉外部工具方法

另外出现的两个之前没有涉及的工具方法“`RpcUtils.attachInvocationIdIfAsync()`”和“`AsyncResult.newDefaultAsyncResult()`”。

Dubbo中的配置总线使用相当广泛，几乎所有的关键组件都会用到它，一个表征方法调用的 `Invocation` 实例也利用到这一特性来决定是否给其设置被调度的唯一ID编号，默认情况下，只有 `Async` 异步模式才会赋予这个编号，此外就是通过设置“`url[methodName + ".invocationid.autoattach"]`”参数来自动赋值编号。

```

public class RpcUtils {

    ...

    //获取表征方法调用的Invocation的编号
    public static Long getInvocationId(Invocation inv) {
        String id = inv.getAttachment(ID_KEY);
        return id == null ? null : new Long(id);
    }

    /**
     * Idempotent operation: invocation id will be added in async operation by default
     *
     * @param url
     * @param inv
     */
    public static void attachInvocationIdIfAsync(URL url, Invocation inv) {
        if (isAttachInvocationId(url, inv) && getInvocationId(inv) == null
            && inv instanceof RpcInvocation) {

            ((RpcInvocation) inv).setAttachment(ID_KEY,
                String.valueOf(INVOKE_ID.getAndIncrement()));
        }
    }

    //如果和Invocation
    private static boolean isAttachInvocationId(URL url, Invocation invocation) {
        String value = url.getMethodParameter(invocation.getMethodName(),
            AUTO_ATTACH_INVOCATIONID_KEY);
        if (value == null) {
            // add invocationid in async operation by default
            return isAsync(url, invocation);
        } else if (Boolean.TRUE.toString().equalsIgnoreCase(value)) {
            return true;
        } else {
            return false;
        }
    }

    ...
}

```

另外在 `AsyncRpcResult` 中有一套 `newDefaultAsyncResult()` 方法是没有提及的，其作用是在结果已知或遇到异常时直接返回一个已经完成的 `AsyncRpcResult` 实例，有了它才可以确保上文提到的 `subscribeTo()` 和 `whenCompleteWithContext()` 等一套复杂的方法调度机制依然可以work的比较好。

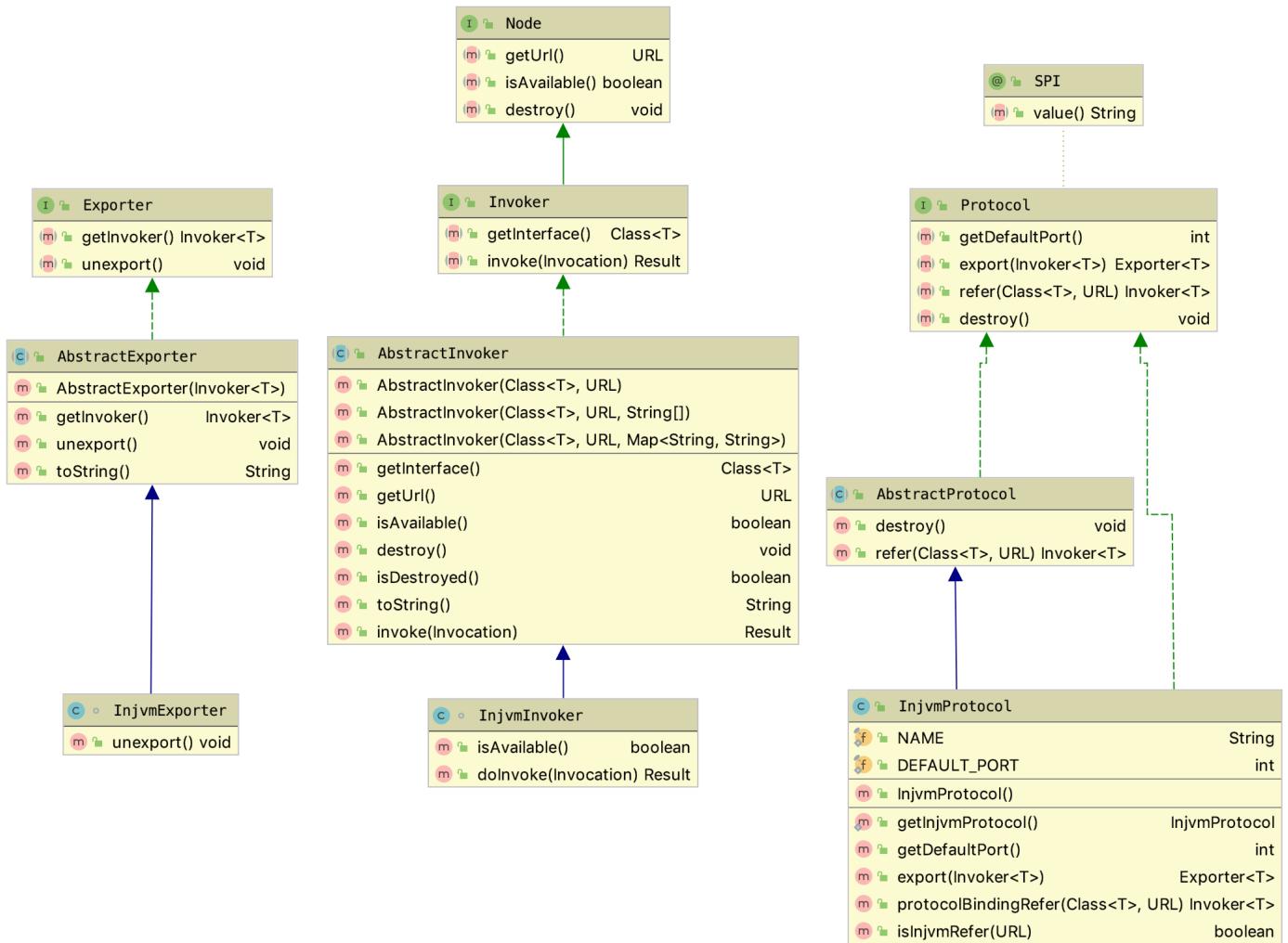
```
public class AsyncRpcResult extends AbstractResult {  
    ...  
  
    public static AsyncRpcResult newDefaultAsyncResult(Object value,  
        Throwable t, Invocation invocation) {  
        AsyncRpcResult asyncRpcResult = new AsyncRpcResult(invocation);  
        AppResponse appResponse = new AppResponse();  
        if (t != null) {  
            appResponse.setException(t);  
        } else {  
            appResponse.setValue(value);  
        }  
        asyncRpcResult.complete(appResponse);  
        return asyncRpcResult;  
    }  
  
    ...  
}
```

## Injvm 协议实现

顾名思义，Injvm解决的是部署在同一个JVM中的微服务客户端和服务端的RPC调用问题，显然这就等同于Java本地的原生方法调用，但却使得调用这个过程被拉长了，多了不少中间环节。然而，这却是必要的：框架是一类问题的通用解决方案，往往某类型的业务可能存在着多个流程有差异的实现版本，却都遵循着一套共用的基础逻辑，即便差异的部分也是实现的同一套接口，这既有利于框架加入新的扩展实现，也有利于保持整体流程上的一致性，更有利于将该类型业务作为一个分支子流程编织到更加宏大的系统流程中。

### 流程分析

相对而言，Injvm版协议支持并不需要底层的跨进程通讯流程参与，因此其实现相对是最简单的。如下述类图所示：



Powered by yFiles

图中涉及了 Exporter、Invoker、Protocol 这3个关键接口，它们有各自的抽象基类实现，并且 Injvm 针对各抽象基类提供了具体实现。简单来看，是 InjvmProtocol 作为创建工厂为调用方实例化了 InjvmExporter 和 InjvmInvoker。但这里有必要大致了解下其实例化流程，理清二者间的关系。

- **InjvmInvoker<T>** :

1. 调用方通过 `refer(serviceType, url)` 传入服务接口类型和配置总线参数URL；
2. InjvmProtocol 利用其抽象方法 `protocolBindingRefer(serviceType, url)` 实现构建并返回一个 InjvmInvoker 实例，记做 injvmInvoker；
3. `refer(serviceType, url)` 使用 `AsyncToSyncInvoker` 包装 injvmInvoker，按需在被 `invoke()` 时将异步调用转为同步，给调用方返回包装后的 Invoker 实例 `asyncInjvmInvoker`；

- **InjvmExporter<T>** :

1. 首先，调用方经 `ProxyFactory#getInvoker(implement, type, url)` 获得一个 `AbstractProxyInvoker` 实例，记为 proxyInvoker，它利用反射机制将 `Invoker<T>#invoke(invocation)` 调用转为对 implement 的本地调用；
2. 随后，经由 `Protocol$Adaptive` 将 `export(invoker)` 方法调用转给 InjvmProtocol；

3. 最后，`InjvmProtocol` 直接构建并返回一个 `invoker` 属性赋值为 `proxyInvoker` 的 `InjvmExporter<T>` 实例 `injvmExporter`；

`Injvm` 协议所发起的RPC调用实际上就是本地间接对服务接口实现的调用，那这个衔接的过程是怎么体现的呢？从上述两个实例化流程可以推断，`asyncInjvmInvoker` 和 `injvmExporter` 必然存在着某种联系，使得方法调用行为能够从前者转移到属于后者的 `proxyInvoker` 上去。

## IMPORTANT

根据SPI机制，Dubbo会为 `Protocol` 扩展点生成一个名为 `Protocol$Adaptive` 的适配型代理类，当 `export(invoker)` 或者 `refer(type, url)` 方法被调用时，它会根据入参搜寻到URL配置总线，由其 `url.protocol` 结合SPI配置找到目标 `Protocol` 扩展点具类，然后将当前方法调用委托给该具类的同名方法。

## 源码剖析

显然，经过上述章节的分析，我们基本已经清楚了实现机制——`InjvmInvoker<T>` 的方法调用被转移到了 `InjvmExporter<T>`，由其 `proxyInvoker` 负责响应。如下源码 `exporter.getInvoker().invoke(invocation)` 是最关键的一句：

```

class InjvmInvoker<T> extends AbstractInvoker<T> {

    private final String key;

    private final Map<String, Exporter<?>> exporterMap;

    InjvmInvoker(Class<T> type, URL url, String key, Map<String, Exporter<?>>
exporterMap) {
        super(type, url);
        this.key = key;
        this.exporterMap = exporterMap;
    }

    @Override
    public boolean isAvailable() {
        InjvmExporter<?> exporter = (InjvmExporter<?>) exporterMap.get(key);
        if (exporter == null) {
            return false;
        } else {
            return super.isAvailable();
        }
    }

    @Override
    public Result doInvoke(Invocation invocation) throws Throwable {
        Exporter<?> exporter = InjvmProtocol.getExporter(exporterMap, getUrl());
        if (exporter == null) {
            throw new RpcException("Service [" + key + "] not found.");
        }
        RpcContext.getContext().setRemoteAddress(LOCALHOST_VALUE, 0);
        return exporter.getInvoker().invoke(invocation);
    }
}

```

实际上，Injvm 使用了同一个 `Map<String, Exporter<?>>` 类型的 Map 容器记录所有本地导出的 `Exporter<?>` 对象，Key 键的取值为 `serviceKey = [{group}]/[{interfaceName}][:{version}]`，而这个容器实际上就是 `AbstractProtocol#exporterMap`，执行 `InjvmExporter<T>#unexport()` 操作时，会将对应 Key 键的 `Exporter<?>` 对象移除。如下述源码：

```
public class InjvmProtocol extends AbstractProtocol implements Protocol {  
    ...  
    @Override  
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
        return new InjvmExporter<T>(invoker, invoker.getUrl().getServiceKey(),  
        exporterMap);  
    }  
  
    @Override  
    public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws  
    RpcException {  
        return new InjvmInvoker<T>(serviceType, url, url.getServiceKey(), exporterMap);  
    }  
}  
  
class InjvmExporter<T> extends AbstractExporter<T> {  
  
    private final String key;  
  
    private final Map<String, Exporter<?>> exporterMap;  
  
    InjvmExporter(Invoker<T> invoker, String key, Map<String, Exporter<?>> exporterMap)  
    {  
        super(invoker);  
        this.key = key;  
        this.exporterMap = exporterMap;  
        exporterMap.put(key, this);  
    }  
  
    @Override  
    public void unexport() {  
        super.unexport();  
        exporterMap.remove(key);  
    }  
}
```

我们曾经有说过，每一个扩展点具类都是单例的，为了编码的便利，Dubbo中有些具类会提供对应的单例获取方法，如下：

```

public class InjvmProtocol extends AbstractProtocol implements Protocol {
    ...
    private static InjvmProtocol INSTANCE;

    public InjvmProtocol() {
        INSTANCE = this;
    }

    public static InjvmProtocol getInjvmProtocol() {
        if (INSTANCE == null) {
            ExtensionLoader.getExtensionLoader(Protocol.class).
                getExtension(InjvmProtocol.NAME); // load
        }
        return INSTANCE;
    }
}

```

一个 refer 引用发生时，若 url.protocol != "injvm"，只要配置了 (url["scope"] = "local" | url["injvm"] = true)，那肯定会判定为JVM内引用，否则 (url["scope"] = "remote" | url["generic"] = true)，则必定不是JVM内引用，除此之外只有当前 url 存在对应 Exporter<T> 才会认为是JVM内引用。

```

public class InjvmProtocol extends AbstractProtocol implements Protocol {
    ...
    public boolean isInjvmRefer(URL url) {
        String scope = url.getParameter(SCOPE_KEY);
        if (SCOPE_LOCAL.equals(scope) || (url.getParameter(LOCAL_PROTOCOL, false))) {
            return true;
        } else if (SCOPE_REMOTE.equals(scope)) {
            return false;
        } else if (url.getParameter(GENERIC_KEY, false)) {
            return false;
        } else if (getExporter(exporterMap, url) != null) {
            return true;
        } else {
            return false;
        }
    }
}

```

最后，refer(serviceType, url) 微服务引用方法中，第二个 url 参数是可以使用通配符的，用于匹配一个目标微服务的 任意分组 或者 任意版本 实现，当然前提是微服务接口类名是匹配的，匹配的工具方法如下：

```
public class UrlUtils {  
    public static boolean isServiceKeyMatch(URL pattern, URL value) {  
        return pattern.getParameter(INTERFACE_KEY).equals(  
            value.getParameter(INTERFACE_KEY))  
            && isItemMatch(pattern.getParameter(GROUP_KEY),  
            value.getParameter(GROUP_KEY))  
            && isItemMatch(pattern.getParameter(VERSION_KEY),  
            value.getParameter(VERSION_KEY));  
    }  
    static boolean isItemMatch(String pattern, String value) {  
        if (pattern == null) {  
            return value == null;  
        } else {  
            return "*".equals(pattern) || pattern.equals(value);  
        }  
    }  
}
```

JAVA

然而，即便 serviceKey 已经匹配了，倘若通过 Exporter<?>.getInvoker() 查找到的 Invoker<?> 服务实例在其总线中配置了如是 url["generic"] = ("true" | "nativejava" | "bean" | "protobuf-json" | "raw.return") 这个参数，则说明对应微服务引用实例使用了泛化调用。

```
public class InjvmProtocol extends AbstractProtocol implements Protocol {  
    ...  
    static Exporter<?> getExporter(Map<String, Exporter<?>> map, URL key) {  
        Exporter<?> result = null;  
  
        if (!key.getServiceKey().contains("*")) {  
            result = map.get(key.getServiceKey());  
        } else {  
            if (CollectionUtils.isNotEmptyMap(map)) {  
                for (Exporter<?> exporter : map.values()) {  
                    if (UrlUtils.isServiceKeyMatch(key,  
                        exporter.getInvoker().getUrl())) {  
                        result = exporter;  
                        break;  
                    }  
                }  
            }  
        }  
  
        if (result == null) {  
            return null;  
        } else if (ProtocolUtils.isGeneric(  
            result.getInvoker().getUrl().getParameter(GENERIC_KEY))) {  
            return null;  
        } else {  
            return result;  
        }  
    }  
}
```

JAVA

# Dubbo 协议实现 之 DubboInvoker<T>

## IMPORTANT

这里的 DubboInvoker，其实例所代表的微服务是客户端一方的概念，表示引用服务端暴露的服务。

DubboInvoker 是协议层中Dubbo协议下的 Invoker 实现，当然主要是完成原生方法调用转换后的 RPC方法调用，它在连接池中以轮询方式选用一个 ExchangeClient 实例将请求发送出去，随后返回一个 AsyncRpcResult 实例，在双工模式需要返回响应的情况下，需要等到服务端处理完请求返回响应才完成这个 CompleteableFuture<AsyncRpcResult>，否则返回的是一个值为null，已经完成处理的值。

## 状态管理

先来看看其状态管理，在《Dubbo远程通讯·信息交换层》一文中，我们已经对 `isAvailable()` 做过初步的分析，其意义是只要还有一个 ExchangeClient 实例处于 非只读 状态，当前实例所代表的微服务就还可用，前提是微服务还没有执行关闭操作。

```
public class DubboInvoker<T> extends AbstractInvoker<T> {  
  
    private final ExchangeClient[] clients;  
  
    private final Set<Invoker<?>> invokers;  
    ...  
  
    @Override  
    public boolean isAvailable() {  
        if (!super.isAvailable()) {  
            return false;  
        }  
        for (ExchangeClient client : clients) {  
            if (client.isConnected() && !client.hasAttribute(  
                Constants.CHANNEL_ATTRIBUTE_READONLY_KEY)) {  
                //cannot write == not Available ?  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

一个微服务实例在Java应用中，绝大部分情况下会被多个线程并发使用，由于涉及到对共享资源 `clients` 和 `invokers` 的处理，因而 `destroy` 操作需要加锁处理。父类 `AbstractInvoker#destroy()` 方法定义要求该销毁操作只能执行一次有效操作，子类 `DubboInvoker#destroy()` 方法利用双检锁来确保这一点：

假如当前实例的该方法被A、B两个线程并发地调用了，在`!super.isDestroyed()`情况下，只会有一个线程A能在`TAG_lock`位置上顺利获得锁，线程B等到该锁被释放后也获得了锁，但是由于已经执行过了`destroy`销毁操作，`!super.isDestroyed()`这个条件已经不成立了，因此无论哪个线程，一旦进入临界区，均需执行`super.isDestroyed()`条件检测。

```
public class DubboInvoker<T> extends AbstractInvoker<T> {
    private final ReentrantLock destroyLock = new ReentrantLock();

    @Override
    public void destroy() {

        if (super.isDestroyed()) {
            return;
        } else {
            //double check to avoid dup close
            destroyLock.lock(); //TAG_lock
            try {
                if (super.isDestroyed()) {
                    return;
                }
                super.destroy();
                if (invokers != null) {
                    invokers.remove(this);
                }
                //=====
                //挨个关闭所使用到客户端连接池
                //=====
                for (ExchangeClient client : clients) {
                    try {
                        client.close(ConfigurationUtils.getServerShutdownTimeout());
                    } catch (Throwable t) {
                        logger.warn(t.getMessage(), t);
                    }
                }
            } finally {
                destroyLock.unlock();
            }
        }
    }

    ...
}
```

## doInvoke 执行流程

该章节开头部分已经初步介绍过其执行的基本流程，具体实现上也很简单，使用求魔%选定`ExchangeClient`对象后，组合请求发送和响应处理两阶段操作，给调用返回一个`Result`类型的`CompletableFuture<Result>`实例。

代码中用到的AtomicPositiveInteger是一个线程安全的可用于单调递增地获得一个整数值，用于轮询获取 ExchangeClient 连接实例。

上文中已经花费大量篇幅阐述 AsyncRpcResult 的 subscribeTo 方法执行的原理，这里的应用简单讲就是新生成一个 AsyncRpcResult 对象，用它订阅 ExchangeClient 的 request() 方法返回的 CompletableFuture<Object>，后者会在其完成回调事件中将结果塞入这个新产生的对象。

```
public class DubboInvoker<T> extends AbstractInvoker<T> {  
    ...  
  
    private final AtomicPositiveInteger index = new AtomicPositiveInteger();  
  
    @Override  
    protected Result doInvoke(final Invocation invocation) throws Throwable {  
        RpcInvocation inv = (RpcInvocation) invocation;  
        final String methodName = RpcUtils.getMethodName(invocation);  
        inv.setAttachment(PATH_KEY, getUrl().getPath());  
        inv.setAttachment(VERSION_KEY, version);  
  
        ExchangeClient currentClient;  
        if (clients.length == 1) {  
            currentClient = clients[0];  
        } else {  
            currentClient = clients[index.getAndIncrement() % clients.length];  
        }  
        try {  
            boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);  
            int timeout = getUrl().getMethodPositiveParameter(methodName,  
                TIMEOUT_KEY, DEFAULT_TIMEOUT);  
            if (isOneway) {  
                boolean isSent = getUrl().getMethodParameter(methodName,  
                    Constants.SENT_KEY, false);  
                currentClient.send(inv, isSent);  
                return AsyncRpcResult.newDefaultAsyncResult(invocation);  
            } else {  
  
                //=====  
                // 构建AsyncRpcResult实例，订阅由request()返回的CompletableFuture<Object>  
                //=====  
                AsyncRpcResult asyncRpcResult = new AsyncRpcResult(inv);  
                CompletableFuture<Object> responseFuture =  
                    currentClient.request(inv, timeout);  
                asyncRpcResult.subscribeTo(responseFuture);  
  
                // save for 2.6.x compatibility, for example,  
                // TraceFilter in Zipkin uses com.alibaba.xxx.FutureAdapter  
                FutureContext.getContext().setCompatibleFuture(responseFuture);  
                return asyncRpcResult;  
            }  
        } catch (TimeoutException e) {  
            throw new RpcException(RpcException.TIMEOUT_EXCEPTION,  
                "Invoke remote method timeout. method: "  
                + invocation.getMethodName() + ", provider: " + getUrl()  
                + ", cause: " + e.getMessage(), e);  
        } catch (RemotingException e) {  
            throw new RpcException(RpcException.NETWORK_EXCEPTION,  
                "Failed to invoke remote method: "  
                + invocation.getMethodName() + ", provider: "  
                + getUrl() + ", cause: " + e.getMessage(), e);  
        }  
    }  
    ...  
}
```

JAVA

有必要提一下的是，DubboInvoker的构造函数中使用 new String[]{INTERFACE\_KEY, GROUP\_KEY, TOKEN\_KEY, TIMEOUT\_KEY} 表示基于Dubbo协议的微服务实例所关注的配置总线中的元数据，分别是“1) 唯一标识微服务的接口；2) 所属分组；3) 运行环境所需的token；4) 超时时间”，像 Invocation 实现一样，将他们从配置总线中设置为自身的附属参数，方便取用。

最后，需要看看上述用于判定当前请求是否为单工模式的 RpcUtils#isOneway(url, inv) 方法的实现，如下，默认是双工模式，否则需要显示设定 url[methodName + ".return"] = false。

```
public class RpcUtils {  
    public static boolean isOneway(URL url, Invocation inv) {  
        boolean isOneway;  
        if (Boolean.FALSE.toString().equals(inv.getAttachment(RETURN_KEY))) {  
            isOneway = true;  
        } else {  
            isOneway = !url.getMethodParameter(getMethodName(inv), RETURN_KEY, true);  
        }  
        return isOneway;  
    }  
    ...  
}
```

JAVA

---

篇幅原因，有关DubboProtocol的实现分析被抽调到下一篇文章。

# Dubbo RPC 之 Protocol 协议层 (二)

本文将聚焦于Dubbo协议下的 Protocol 实现 DubboProtocol，以及利用装饰模式实现的 ProtocolFilterWrapper 和 ListenerExporterWrapper。

## NOTE

本文所涉及剖析源码不少依赖于Dubbo的SPI机制，在翻阅之前先请仔细阅读《Dubbo之SPI扩展点加载》一文。

## DubboProtocol → AbstractProtocol

在本文开篇为了搞清楚Protocol协议层的作用，耗费不少笔墨，上来就剖析其实现，难免会感觉吃力，因此之后便着重挨个分析相关的`Result`、`Invocation`、`Invoker`，扫清障碍后，下述我们结合代码为本文画上一个圆满的句号。

### AbstractProtocol 基类

同样，该类是为所有协议实现提供最基础的实现，主要着墨点在：

1. 整个协议实例的资源销毁处理。
2. 对所有的 Invoker 实现提供一个 AsyncToSyncInvoker 封装类，其目的是如果其实例所表示的微服务接口没有在签名或者配置总线声明自己使用异步机制调度时，便将其转换为同步调用。

### 异步转同步操作 AsyncToSyncInvoker

AsyncToSyncInvoker 的实现原理很简单，是一个装饰类，实际完成工作的是被装饰的另一个 Invoker 实例，因此对应接口的几乎所有行为会直接委托给后者，只根据其自身特点将实现 invoke() 方法加以特别处理。

过程便是先调用被封装 invoker 对象的同名 invoke() 方法，获得一个超类为 CompleteableFuture<Result> 的 Result 类的对象R，如果从入参 Invocation 中检验到不需要使用异步调用模式，则调用后者的 get() 方法，它会等待直到响应回来，除非超时。除此之外没什么差别，依然返回的是 invoker.invoke(invocation) 返回的结果R。

```

public class AsyncToSyncInvoker<T> implements Invoker<T> {
    private Invoker<T> invoker;

    public Result invoke(Invocation invocation) throws RpcException {
        Result asyncResult = invoker.invoke(invocation);

        try {
            if (InvokeMode.SYNC == ((RpcInvocation) invocation).getInvokeMode()) {
                asyncResult.get(Integer.MAX_VALUE, TimeUnit.MILLISECONDS);
            }
        } catch (InterruptedException | ExecutionException | Throwable e) {
            ...
        }
        return asyncResult;
    }

    ...
}

```

JAVA

在应用程序作为客户端使用 refer() 方法引用微服务的时候， AbstractProtocol 基类会利用该 AsyncToSyncInvoker 对其引用的任何微服务实例进行一层包装适配。因此另外提炼了一个需子类实现的抽象方法，如下：

```

/**
 * 客户端使用接口的类型和配置总线URL获得一个彼端微服务的引用实例
 */
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    return new AsyncToSyncInvoker<>(protocolBindingRefer(type, url));
}

protected abstract <T> Invoker<T> protocolBindingRefer
    (Class<T> type, URL url) throws RpcException;

```

JAVA

## 资源销毁处理

一个 Invoker 对象可以被认为是一个微服务实例，分成两大类，服务端微服务实例和客户端微服务引用实例，仔细阅读过上文应该不难明白他们的差异性，前者是真正提供服务的，其Invoker类无需具体协议实现，而后者是客户端这边的一个概念，使得应用可以像引用本地服务一样去对前者发起请求。

为了对微服务实例方便进行统一管理， AbstractProtocol 申明了如下两个变量， exporterMap 用于缓存所有服务端和客户端导出的所有微服务实例，而 invokers 则仅仅用于缓存所有做RPC调用的微服务引用实例。细究起来，其实 exporterMap 和 invokers 的作用也基本限定在做销毁处理而已，尽管前者将实例封装在一个 Exporter 类型的对象中。

```
protected final Map<String, Exporter<?>> exporterMap =
    new ConcurrentHashMap<String, Exporter<?>>();
```

```
protected final Set<Invoker<?>> invokers =
    new ConcurrentHashSet<Invoker<?>>();
```

JAVA

资源的销毁过程比较简单，逐个遍历容器中的元素，先从容器中移除其有元素应用，再调用元素自身的`destroy()`或`unexport()`方法，实际上方法`unexport()`背后执行的依然是一个`Invoker`实例的`destroy()`。

```
@Override
public void destroy() {
    for (Invoker<?> invoker : invokers) {
        if (invoker != null) {
            invokers.remove(invoker);
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Destroy reference: " + invoker.getUrl());
                }
                invoker.destroy();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
    for (String key : new ArrayList<String>(exporterMap.keySet())) {
        Exporter<?> exporter = exporterMap.remove(key);
        if (exporter != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Unexport service: " + exporter.getInvoker().getUrl());
                }
                exporter.unexport();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
}
```

## Dubbo协议实现 DubboProtocol

`DubboProtocol`是基于Dubbo协议对`Protocol`的实现，前面这半句话说感觉说得颇有问题，协议实际上如文章开头所言，只有在通讯双方发生会话时才有意义，其中包含了信息的编解码处理、流程控制等一系列复杂的约定，而`Protocol`的实现类仅仅是相关实现细节的最后总组装而已。

整个`DubboProtocol`的实现相当复杂，我们下面将按照客户端和服务端分成两个大的章节，逐步深入。

### 服务引用实现

如下述源码所示，服务引用的过程，实际上是产生微服务Invoker实例的过程，初步看起来流程相当精简。

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

如前文已经提及Protocol需要负责给客户端引用微服务实例提供用于远端通讯的ExchangeClient，也就是 `getClients(url)` 所表示的那一截代码，然而沿着它扩散开来，有如从公园外墙推开一扇门，大有洞天，难以尽收眼底。眼花缭乱之际也是眩晕之时，要一探究竟，咱还是先把绕道其看看和其密切相关的两个 ExchangeClient 接口实现类—— `LazyConnectExchangeClient` & `ReferenceCountExchangeClient`。

## ReferenceCountExchangeClient

微服务开发过程中，一个应用引用多个微服务实例是很常见的事情，一个服务端微服务作为一个应用占有了一个JVM虚拟机，自然就拥有了其所在主机的唯一端口号，而连接它的客户端为了效率上的考量会利用连接池供多个线程并发地访问它，Dubbo的实现中，从单一客户端连接同一个微服务的微服务引用实例是可以存在多份的。这种情况下，一个 ExchangeClient 对象就可以被这多份实例共享，这时就不能随随便便被 `close` 掉，只有不再有微服务引用实例使用它时才可 `close`。基于这种需求，Dubbo专门为 ExchangeClient 提供了一个使用引用计数的封装类 `ReferenceCountExchangeClient`。

具体实现上很简单，和一般的装饰器类一样，实现 ExchangeClient 接口，其无关当前特定业务的接口方法全部委托给其实现同一接口的引用属性 `client` 完成，在特定方法进行业务逻辑改写处理。它声明了一个表示引用计数的基于CAS实现的原子变量 `AtomicInteger referenceCount`，被实例化时，执行+1操作，后面一旦被另外其它一个同微服务的服务引用 Invoker 对象所使用，便再次执行+1操作，而 `close` 操作时会首先对其执行-1操作，然后检查该属性是否为0，为0可以调用被封装的 `client` 对象的 `close()` 方法安全关闭，否则直接忽略掉。

在当前Protocol协议层Dubbo对 ExchangeClient 的正常 `close` 操作做了更进一步处理，会使用 `LazyConnectExchangeClient` 封装将已经关闭的对象，如果当前 `ReferenceCountExchangeClient` 实例被再一次调用，该实例会被神奇般的复活。当然为了不导致其多层嵌套引用一个 `ReferenceCountExchangeClient` 类型的 ExchangeClient 实例对象，`LazyConnectExchangeClient` 不再直接封装一个 ExchangeClient 实例，而是基于后者获取其 `ExchangeHandler` 引用实现 ExchangeClient 接口。

```
final class ReferenceCountExchangeClient implements ExchangeClient {  
  
    private final URL url;  
    private final AtomicInteger referenceCount = new AtomicInteger(0);  
  
    private ExchangeClient client;  
  
    public ReferenceCountExchangeClient(ExchangeClient client) {  
        this.client = client;  
        referenceCount.incrementAndGet();  
        this.url = client.getUrl();  
    }  
    ...  
    /**  
     * close() is not idempotent any longer  
     */  
    @Override  
    public void close() {  
        close(0);  
    }  
  
    @Override  
    public void close(int timeout) {  
        if (referenceCount.decrementAndGet() <= 0) {  
            if (timeout == 0) {  
                client.close();  
  
            } else {  
                client.close(timeout);  
            }  
  
            replaceWithLazyClient();  
        }  
    }  
}  
/**  
 * when closing the client, the client needs to be set to  
 LazyConnectExchangeClient, and if a new call is made,  
 * the client will "resurrect".  
 *  
 * @return  
 */  
private void replaceWithLazyClient() {  
    // this is a defensive operation to avoid client is closed by accident, the  
 initial state of the client is false  
    URL lazyUrl = URLBuilder.from(url)  
        .addParameter(LAZY_CONNECT_INITIAL_STATE_KEY, Boolean.FALSE)  
        .addParameter(RECONNECT_KEY, Boolean.FALSE)  
        .addParameter(SEND_RECONNECT_KEY, Boolean.TRUE.toString())  
        .addParameter("warning", Boolean.TRUE.toString())  
        .addParameter(LazyConnectExchangeClient.REQUEST_WITH_WARNING_KEY,  
 true)  
        .addParameter("_client_memo",  
 "referencecounthandler.replacewithlazyclient")  
        .build();  
  
    /**
```

JAVA

```
* the order of judgment in the if statement cannot be changed.  
*/  
if (!(client instanceof LazyConnectExchangeClient) || client.isClosed()) {  
    client = new LazyConnectExchangeClient(lazyUrl,  
client.getExchangeHandler());  
}  
}  
/**  
 * The reference count of current ExchangeClient, connection will be closed if all  
invokers destroyed.  
*/  
public void incrementAndGetCount() {  
    referenceCount.incrementAndGet();  
}  
...  
}
```

## LazyConnectExchangeClient

本质上就如同 `ReferenceCountExchangeClient` 一样，`LazyConnectExchangeClient` 也是用于对某个目标 `ExchangeClient` 实例进行封装，但实现上功能和目的完全不同，后者的主要目的是在已知“配置总线URL”和“网络事件监听器ExchangeHandler”这两个输入的情况下延迟创建 `ExchangeClient` 实例，将这一时刻推迟到业务出站请求之时。

在《【六】Dubbo远程通讯之信息交换层》一文中曾提到，由于Dubbo的分层模型，网络I/O事件的回调是自下往上、逐层执行的，上层是对下一层的封装和增强，因此如果某一层一种组件对象的创建是依赖比其更低一级的，那么只能在其网络I/O事件的回调中反向完成其创建操作，还得使用一些排重手段，保证只会实例化一次，另外还需搭配一些前验代码。

类似对象创建方式在惰性实例化时也很常见，比如一个类提供了n个方法，其中有几个方法会涉及到实例化。`LazyConnectExchangeClient` 中的 `client` 属性采取的便是这种方式。

如下述代码所示，分为初始化实现和调用两部分：第一部分使用`volatile`可见性修饰符、`ReentrantLock`可重入锁、锁的双检这几种机制确保在多线程并发情况下依然只会安全的实例化 `ExchangeClient` 对象一次；第二部分则是初始化调用，在每一个出站事件回调方法中均调用第一部分提供的 `initClient()` 方法，确保只有一个实例。

```
final class LazyConnectExchangeClient implements ExchangeClient {  
  
    ...  
    private final URL url;  
    private final ExchangeHandler requestHandler;  
  
    private volatile ExchangeClient client;  
    private final Lock connectLock = new ReentrantLock();  
  
    private void initClient() throws RemotingException {  
        if (client != null) {  
            return;  
        }  
        if (logger.isInfoEnabled()) {  
            logger.info("Lazy connect to " + url);  
        }  
        connectLock.lock();  
        try {  
            if (client != null) {  
                return;  
            }  
            this.client = Exchangers.connect(url, requestHandler);  
        } finally {  
            connectLock.unlock();  
        }  
    }  
//=====  
//调用initClient初始化ExchangeClient实例  
//=====  
    @Override  
    public void send(Object message) throws RemotingException {  
        initClient();  
        client.send(message);  
    }  
  
    @Override  
    public void send(Object message, boolean sent) throws RemotingException {  
        initClient();  
        client.send(message, sent);  
    }  
  
    @Override  
    public CompletableFuture<Object> request(Object request, int timeout)  
        throws RemotingException {  
        warning();  
        initClient();  
        return client.request(request, timeout);  
    }  
  
    @Override  
    public CompletableFuture<Object> request(Object request)  
        throws RemotingException {  
        warning();  
        initClient();  
        return client.request(request);  
    }
```

JAVA

```
    }
    ...
}
```

于一些不涉及数据出站处理的方法，`LazyConnectExchangeClient` 专门为其实现提供了如下的前验检查代码，它们分别是“`removeAttribute`、`setAttribute`、`reconnect`、`reset`、`getChannelHandler`”，检查通过则直接使用被封装的 `ExchangeClient` 实例完成功能，这类方法的特点是调用方需要感知到行为的发生。

```
private void checkClient() {
    if (client == null) {
        throw new IllegalStateException(
            "LazyConnectExchangeClient state error. the client has not be init
.url:" + url);
    }
}
```

JAVA

对于 `close` 类操作则处理相对很简单，不满足条件时，直接进行忽略处理，如下所示：

```
@Override
public void startClose() {
    if (client != null) {
        client.startClose();
    }
}
```

JAVA

有两个在构造方法中出现的配置总线参数，这里有必要提及下，分别是 `url["send.reconnect"]` 和 `url["lazyclient_request_with_warning"]`。如下述源码所示，前者在当前对象创建时加入到配置总线中，用于确保该内嵌对象在向彼端发送请求之时，所使用的通道Channel是连接着的（在使用 `Client` 发送数据时，若连接断开，则自动连接）；而后者则是用于确认是否需要提示警告信息，需要的话，则每 5000 次的方法调用会提醒一次，对于一个频繁使用的微服务，其所使用 `ExchangeClient` 不应采用惰性模式。。

```
final class LazyConnectExchangeClient implements ExchangeClient {
```

JAVA

```
...  
/**  
 * when this warning rises from invocation, program probably have bug.  
 */  
protected static final String REQUEST_WITH_WARNING_KEY =  
"lazyclient_request_with_warning";  
protected final boolean requestWithWarning;  
private final int warning_period = 5000;  
private AtomicLong warningcount = new AtomicLong(0);  
  
public LazyConnectExchangeClient(URL url, ExchangeHandler requestHandler) {  
    // lazy connect, need set send.reconnect = true, to avoid channel bad status.  
    this.url = url.addParameter(SEND_RECONNECT_KEY, Boolean.TRUE.toString());  
    this.requestHandler = requestHandler;  
  
    //DEFAULT_LAZY_CONNECT_INITIAL_STATE的默认值为true  
    this.initialState = url.getParameter(LAZY_CONNECT_INITIAL_STATE_KEY,  
DEFAULT_LAZY_CONNECT_INITIAL_STATE);  
    this.requestWithWarning = url.getParameter(REQUEST_WITH_WARNING_KEY, false);  
}  
  
/**  
 * If {@link #REQUEST_WITH_WARNING_KEY} is configured, then warn once every 5000  
invocations.  
 */  
private void warning() {  
    if (requestWithWarning) {  
        if (warningcount.get() % warning_period == 0) {  
            logger.warn(new IllegalStateException("safe guard client , should not  
be called ,must have a bug."));  
        }  
        warningcount.incrementAndGet();  
    }  
}
```

## ExchangeClient 候选集准备

经过上述的两个小章节的铺垫后，这时再回过头来，便可以比较轻松地理解Dubbo中是如何准备 ExchangeClient 的候选集的。在关于 ReferenceCountExchangeClient 的实现探究过程中，我们清楚，对于分别占用一个JVM的一对“客户端 ↔ 服务端”来说，他们之间存在通讯连接通道 Channel 和 ExchangeClient —— 绑定可以存在多份，同时客户端也可以具备多份服务引用 Invoker 实例，实现上“Invoker 服务引用实例”和“ExchangeClient 客户端通讯处理实例”的关系可以是一对一或一对多的独占模式，也可以使多对一或者多对多的共享模式。

默认情况下，也即没有设置 url["connections"] 参数，采用的是共享模式，这时可以通过设置 url["shareconnections"] 或者系统参数 env["shareconnections"]，防止默认只有一个共享的通讯连接通道而引发的瓶颈问题。此外，显示配置的情况下使用的是独占模式。

```

private ExchangeClient[] getClients(URL url) {
    // whether to share connection

    boolean useShareConnect = false;

    int connections = url.getParameter(CONNECTIONS_KEY, 0);
    List<ReferenceCountExchangeClient> shareClients = null;
    // if not configured, connection is shared, otherwise, one connection for one
    service
    if (connections == 0) {
        useShareConnect = true;

        /**
         * The xml configuration should have a higher priority than properties.
         */
        String shareConnectionsStr = url.getParameter(
            SHARE_CONNECTIONS_KEY, (String) null);
        connections = Integer.parseInt(StringUtils.isBlank(shareConnectionsStr) ?
            ConfigUtils.getProperty(SHARE_CONNECTIONS_KEY, DEFAULT_SHARE_CONNECTIONS) :
            shareConnectionsStr);
        shareClients = getSharedClient(url, connections);
    }

    ExchangeClient[] clients = new ExchangeClient[connections];
    for (int i = 0; i < clients.length; i++) {
        if (useShareConnect) {
            clients[i] = shareClients.get(i);

        } else {
            clients[i] = initClient(url);
        }
    }

    return clients;
}

```

## 独占模式下的单个 ExchangeClient 的初始化操作

独占模式下的 ExchangeClient 的初始化相对来说比较简单：

1. 首先，由于性能问题， DubboProtocol 协议实现在网络传输层不会选用低效的BIO模式，确保能找到 (url["server"] | url["client"] | "netty") 所指定的 Transporter 扩展点实现；
2. 然后再配置总线中增设解码 url["codec"] = "dubbo" 和心跳参数 url["heartbeat"] = "60000"，确保：1）在信息交换层采用了Dubbo协议的编解码；2）使用心跳机制维持客户端到服务端的长连接，默认心跳时长为一分钟；
3. 随后就是使用上述增设了参数的配置总线url参数实例化 ExchangeClient 对象或者 LazyConnectExchangeClient 对象，后者需总线中已指定 url["lazy"] = "true"；

```

/**
 * Create new connection
 *
 * @param url
 */
private ExchangeClient initClient(URL url) {

    // client type setting.
    String str = url.getParameter(CLIENT_KEY,
        url.getParameter(SERVER_KEY, DEFAULT_Remoting_CLIENT));

    url = url.addParameter(CODEC_KEY, DubboCodec.NAME);
    // enable heartbeat by default
    url = url.addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT));

    // BIO is not allowed since it has severe performance issue.
    if (str != null && str.length() > 0 &&
        !ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported client type: " + str + ", " +
            " supported client type is " + StringUtils.join(
                ExtensionLoader.getExtensionLoader(Transporter.class).
                    getSupportedExtensions(), " "));
    }

    ExchangeClient client;
    try {
        // connection should be lazy
        if (url.getParameter(LAZY_CONNECT_KEY, false)) {
            client = new LazyConnectExchangeClient(url, requestHandler);

        } else {
            client = Exchangers.connect(url, requestHandler);
        }
    } catch (RemotingException e) {
        throw new RpcException("Fail to create remoting client for service(" + url +
    "): " + e.getMessage(), e);
    }

    return client;
}

```

我们知道，在微服务开发中，一个服务可以定义多个接口，也即对应着Java中的 `interface` 和其中定义的若干方法，一个服务端服务通常占用了一个JVM虚拟机，处于其中的服务接口可能被访问的频度差异非常巨大，也有可能分布是比较均匀的。

## 共享模式下的单个 ExchangeClient 的初始化操作

实际上共享模式只是独占模式的一种特例，因此其 `ExchangeClient` 的实例化直接调用了 `initClient()`，这也意味着 `ReferenceCountExchangeClient` 可以用于包装 `LazyConnectExchangeClient`，如下述源码所示：

```
/**  
 * Bulk build client  
 *  
 * @param url  
 * @param connectNum  
 * @return  
 */  
private List<ReferenceCountExchangeClient>  
    buildReferenceCountExchangeClientList(URL url, int connectNum) {  
    List<ReferenceCountExchangeClient> clients = new ArrayList<>();  
  
    for (int i = 0; i < connectNum; i++) {  
        clients.add(buildReferenceCountExchangeClient(url));  
    }  
  
    return clients;  
}  
  
/**  
 * Build a single client  
 *  
 * @param url  
 * @return  
 */  
private ReferenceCountExchangeClient buildReferenceCountExchangeClient(URL url) {  
    ExchangeClient exchangeClient = initClient(url);  
  
    return new ReferenceCountExchangeClient(exchangeClient);  
}
```

共享模式下的初始化之所以复杂，原因是对于同一个微服务的客户端服务引用 `Invoker`，每次其新创建实例的时候，均需要执行对 `ReferenceCountExchangeClient` 实例的计数器的 +1 操作，它可能是一个已经创建了并缓存在 `Map<String, List<ReferenceCountExchangeClient>>` 缓存 Map 中键为微服务的 address 地址。由于该 Map 是共享的，并发模式下需要确保其安全，业务上需要确保能够稳定地提供相应数量的共享 `ReferenceCountExchangeClient` 对象，因此还需要在线程安全的前提下实现对已经失效或 `close` 掉的实例做替换处理。

```
/**
 * <host:port,Exchanger>
 */
private final Map<String, List<ReferenceCountExchangeClient>> referenceClientMap = new
ConcurrentHashMap<>();
private final ConcurrentHashMap<String, Object> locks = new ConcurrentHashMap<>();

/***
 * Get shared connection
 *
 * @param url
 * @param connectNum connectNum must be greater than or equal to 1
 */
private List<ReferenceCountExchangeClient> getSharedClient(URL url, int connectNum) {
    String key = url.getAddress();
    List<ReferenceCountExchangeClient> clients = referenceClientMap.get(key);

    if (checkClientCanUse(clients)) {
        batchClientRefIncr(clients);
        return clients;
    }

    locks.putIfAbsent(key, new Object());
    synchronized (locks.get(key)) {
        clients = referenceClientMap.get(key);
        // dubbo check
        if (checkClientCanUse(clients)) {
            batchClientRefIncr(clients);
            return clients;
        }
    }

    // connectNum must be greater than or equal to 1
    connectNum = Math.max(connectNum, 1);

    // If the clients is empty, then the first initialization is
    if (CollectionUtils.isEmpty(clients)) {
        clients = buildReferenceCountExchangeClientList(url, connectNum);
        referenceClientMap.put(key, clients);
    } else {
        for (int i = 0; i < clients.size(); i++) {
            ReferenceCountExchangeClient referenceCountExchangeClient
                = clients.get(i);
            // If there is a client in the list that is no longer available,
            //create a new one to replace him.
            if (referenceCountExchangeClient == null ||
                referenceCountExchangeClient.isClosed()) {
                clients.set(i, buildReferenceCountExchangeClient(url));
                continue;
            }
        }
        referenceCountExchangeClient.incrementAndGetCount();
    }
}
/***
```

```
* I understand that the purpose of the remove operation
* here is to avoid the expired url key
* always occupying this memory space.
*/
locks.remove(key);

return clients;
}
}
```

上述代码中也使用了很常见的锁的双检机制，当传入给 `checkClientCanUse` 的 `ReferenceCountExchangeClient` 对象列表中的只要有一个对象处于无效或者 `close` 状态，便随后进入主体逻辑中，确保满足数目要求的基础上，列表中所有的对象均可用，也即 `ExchangeClient` 所代表的客户端和服务端保持长连状态。否则只会简单的对属于目标服务的下的 `ReferenceCountExchangeClient` 进行 +1 操作。

```

/**
 * Check if the client list is all available
 *
 * @param referenceCountExchangeClients
 * @return true-available, false-unavailable
 */
private boolean checkClientCanUse(List<ReferenceCountExchangeClient>
referenceCountExchangeClients) {
    if (CollectionUtils.isEmpty(referenceCountExchangeClients)) {
        return false;
    }

    for (ReferenceCountExchangeClient referenceCountExchangeClient :
referenceCountExchangeClients) {
        // As long as one client is not available, you need to replace the unavailable
client with the available one.
        if (referenceCountExchangeClient == null ||
referenceCountExchangeClient.isClosed()) {
            return false;
        }
    }

    return true;
}

/**
 * Increase the reference Count if we create new invoker shares same connection, the
connection will be closed without any reference.
 *
 * @param referenceCountExchangeClients
 */
private void batchClientRefIncr(List<ReferenceCountExchangeClient>
referenceCountExchangeClients) {
    if (CollectionUtils.isEmpty(referenceCountExchangeClients)) {
        return;
    }

    for (ReferenceCountExchangeClient referenceCountExchangeClient :
referenceCountExchangeClients) {
        if (referenceCountExchangeClient != null) {
            referenceCountExchangeClient.incrementAndGetCount();
        }
    }
}

```

## 服务导出实现

上述有关 DubboProtocol 源码的分析中，已经刻意地忽略掉了 requestHandler 创建的问题，在前述有关Protocol协议层的讨论中也没有提及客户端服务导出的相关逻辑。实际上服务导出并不限于服务端，它同时也存在于客户端。

在前面的Dubbo实现源码剖析中，我们已经知道，不管是客户端还是服务端，都可以直接使用其通道 Channel 向彼端主动发送消息数据，但是对于来自彼端的请求则于应用层来说是被动的，只能在网络 I/O 事件就绪后，由框架回调应用层的逻辑代码。因此 Dubbo 在协议层需要在回调方法 `received()` 中，将收到的代表原生请求 `message` 类型为 `Invocation` 的请求转给对应的微服务实例或微服务引用实例处理，处理完再返回结果。

因此：

- 在微服务的原生 Java 方法发起调用之前，服务端需要导出提供服务的 `Invoker`，而客户端则需要导出引用服务的 `Invoker`，便于发起 RPC 调用；
- 无论是客户端还是服务端，均需要根据某种规则获取到 `Invoker` 对象 微服务或其客户端引用的实例；
- 对于服务端还需要创建该 `Invoker` 的 `ExchangeServer` 服务实例，服务连入客户端；
- 实现 `ExchangeHandler` 被装饰者业务逻辑，响应 `Invocation` 类型入站请求详见下述相关章节；

## Invoker 实例 服务实例&引用实例 导出

在服务导出实现源码中，`Exporter` 接口及其实现类 `DubboExporter<T>` → `AbstractExporter<T>`，存在的目的更多的是保持框架分层业务语义上的完整性，用于封装一个 `Invoker` 实例，便于后续进行销毁处理。其实例化也即导出，调用 `unexport` 时便驱动执行 `Invoker` 实例的 `destroy()` 方法，为了确保只会销毁操作不会重复执行，声明了一个辅助变量——`volatile boolean unexported`。

所有被导出 `Invoker` 实例先被装入一个 `DubboExporter` 实例，随后整体载入到 `Map<String, Exporter<?>> exporterMap` 缓存中，其中的键值的表示形式为 “`[group/]serviceName[:version]:port`”[XXX]: XXX 可选，其中包含的 4 个元素分别对应配置总线 URL 中的值：1) `url["group"]`；2) `url.path`；3) `url["version"]`；4) `url.port`。

服务导出是由 Dubbo 框架调用方法 `public <T> Exporter<T> export(Invoker<T> invoker)` `throws RpcException` 完成的，其传入的 `Invoker` 要么是框架使用动态代理方式实现的，要么就是协议层中的由第三方提供的类似 `DubboInvoker` 实现。

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

JAVA

```
    // export service.  
    // 构建完Key之后，将invoker缓存起来  
    String key = serviceKey(url);  
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);  
    exporterMap.put(key, exporter);
```

...

```
//配置总线告知是server端才需要创建服务实例，分下参见下文  
openServer(url);  
optimizeSerialization(url);
```

```
return exporter;
```

```
}
```

## ExchangeServer 创建

默认而言，如果服务配置总线中没有设置 `url["isserver"]`，Dubbo会默认为当前的 `export()` 操作准备提供服务的 `ExchangeServer` 实例。上述提到，引用同一个服务端微服务实例的所有 `ReferenceCountExchangeClient` 对象会被装入到一个列表中，最后再以 `<host:port,Exchanger>` 的形式缓存起来，也即类型为 `ConcurrentHashMap<String, List<ReferenceCountExchangeClient>>` 的 `referenceClientMap` 变量。

同样，所有服务端提供服务的 `ExchangeServer` 也会以类似的形式缓存在 `ConcurrentHashMap<String, ExchangeServer>` 类型的 `serverMap` 中，在创建服务实例时，若发现已经存在对应的实例，则会使用配置总线对其参数做重设处理。

```
private final Map<String, ExchangeServer> serverMap = new ConcurrentHashMap<>();  
  
private void openServer(URL url) {  
    // find server.  
    String key = url.getAddress();  
    //client can export a service which's only for server to invoke  
    boolean isServer = url.getParameter(IS_SERVER_KEY, true);  
    if (isServer) {  
        ExchangeServer server = serverMap.get(key);  
  
        //use double check mode to create service instance  
        if (server == null) {  
            synchronized (this) {  
                server = serverMap.get(key);  
                if (server == null) {  
                    serverMap.put(key, createServer(url));  
                }  
            }  
        } else {  
            // server supports reset, use together with override  
            server.reset(url);  
        }  
    }  
}
```

JAVA

下述创建 ExchangeServer 实例的创建过程中， Exchangers.bind(url, requestHandler) 为最核心的一句，需要指定 url["channel.readonly.sent"] = (|true)、url["heartbeat"] = (|60000)、url["codec"] = "dubbo"，同时需要确保当前应用中存在由 url["server"] 和 url["client"] 所配置的 Transporter 扩展点。

```

private ExchangeServer createServer(URL url) {
    //配置总线相关参数设置
    url = URLBuilder.from(url)
        // send readonly event when server closes, it's enabled by default
        .addParameterIfAbsent(CHANNEL_READONLYEVENT_SENT_KEY,
        Boolean.TRUE.toString())
        // enable heartbeat by default
        .addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT))
        .addParameter(CODEC_KEY, DubboCodec.NAME)
        .build();
    String str = url.getParameter(SERVER_KEY, DEFAULT_Remoting_SERVER);

    //校验是否配置服务类型已经存在相应的实现，由SPI指定
    if (str != null && str.length() > 0 && !ExtensionLoader.
        getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported server type: " + str + ", url: " + url);
    }

    ExchangeServer server;
    try {
        server = Exchangers.bind(url, requestHandler);
    } catch (RemotingException e) {
        throw new RpcException("Fail to start server(url: "
            + url + ") " + e.getMessage(), e);
    }

    str = url.getParameter(CLIENT_KEY);
    if (str != null && str.length() > 0) {
        Set<String> supportedTypes = ExtensionLoader.getExtensionLoader(
            Transporter.class).getSupportedExtensions();
        if (!supportedTypes.contains(str)) {
            throw new RpcException("Unsupported client type: " + str);
        }
    }

    return server;
}

```

## 响应RPC调用

前面已经阐述过入站的网络数据包括Request请求和Response响应，当他们的网络I/O事件就绪时，便会触发绑定在通道上的 HeaderExchangeHandler 事件监听器 A 的 received() 方法，处于更加底层的信息交换层会将其中的 Request 请求（需要返回响应）转发给 ExchangeHandler 对象 B 定义的 reply() 方法，由其构建 CompletableFuture<Object> 类型的响应结果。①

注：

- 1) **ExchangeHandler**: public CompletableFuture<Object> reply(ExchangeChannel channel, Object message) throws RemotingException ;
- 2) **ChannelHandler**: public void received(Channel channel, Object message) throws RemotingException

## IMPORTANT

B的类型是一个扩充版的 `ChannelHandler`，而A的类型 `HeaderExchangeHandler` 则是前者装饰者实现，也即A封装了B，A的I/O回调最终都会委托给B。

在Dubbo协议层中，对类型为 `Invocation` 非 `[Request、Response、String]` 外的入站请求做了同样的处理，也就是 `ExchangeHandler#reply()` 方法会进一步调用 `Invoker` 实例的 `invoke()` 方法，以完成对应Java原生方法的调用，或者由Java原生方法转换后的跨机网络请求。<sup>②</sup>

需要注意的是，上述讨论的两种被调用的场景①和②，`reply()` 均属于同一个 `ExchangeHandler` 对象，因此要求第一种场景中，其 `Request` 对象封装的 `mData` 也是 `Invocation` 类型的，否则会抛出异常。

大体实现如下述源码所示，会首先从 `exporterMap` 缓存中取得相对应的 `Invoker` 实例，使用它回调表征原生Java方法的 `Invocation` 对象：

```
JAVA  
  
private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {  
    ...  
    @Override  
    public CompletableFuture<Object> reply(ExchangeChannel channel, Object message)  
    throws RemotingException {  
  
        //当前ExchangeHandlerAdapter只响应消息为Invocation类型的请求  
        if (!(message instanceof Invocation)) {  
            throw new RemotingException(channel, "Unsupported request: "  
                + (message == null ? null : (message.getClass().getName() + ":" +  
message))  
                + ", channel: consumer: " + channel.getRemoteAddress()  
                + " -> provider: " + channel.getLocalAddress());  
        }  
  
        Invocation inv = (Invocation) message;  
        Invoker<?> invoker = getInvoker(channel, inv);  
        ...  
  
        RpcContext.getContext().setRemoteAddress(channel.getRemoteAddress());  
  
        //间接完成对应Java原生方法的调用或者由Java原生方法转换后的跨机网络请求  
        Result result = invoker.invoke(inv);  
  
        //先调用completionFuture()将CompletionStage<Result>  
        // 转换成CompletableFuture<Result>  
        //再调用thenApply(Function.identity())装换成CompletableFuture<Object>  
        //利用了泛型出参自带类型转换特性，也即：  
        // <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)  
        return result.completionFuture().thenApply(Function.identity());  
    }  
  
    @Override  
    public void received(Channel channel, Object message) throws RemotingException {  
        if (message instanceof Invocation) {  
            reply((ExchangeChannel) channel, message);  
  
        } else {  
            super.received(channel, message);  
        }  
    }  
    ...  
  
    //根据当前通道内含信息及Invocation对象中的本地参数容器构建serviceKey,  
    //由其从exporterMap键值对中最终获取到Invoker对象  
    Invoker<?> getInvoker(Channel channel, Invocation inv) throws RemotingException {  
        int port = channel.getLocalAddress().getPort();  
        String path = inv.getAttachments().get(PATH_KEY);  
  
        ...  
        String serviceKey = serviceKey(port, path,  
            inv.getAttachments().get(VERSION_KEY), inv.getAttachments().get(GROUP_KEY));  
  
        DubboExporter<?> exporter = (DubboExporter<?>) exporterMap.get(serviceKey);  
    }
```

```

if (exporter == null) {
    throw new RemotingException(channel, "Not found exported service: "
        + serviceKey + " in " + exporterMap.keySet() + ", may be version or group
mismatch "
        + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider: "
        + channel.getLocalAddress() + ", message:" + inv);
}

return exporter.getInvoker();
}

```

另外，Dubbo允许为微服务实例或者引用实例配置 url["ondisconnect"] 和 url["onconnect"]，监听链入或者断链时的监听，如下述示例配置的 ondisconnect：

XML

```

<beans>
    <bean id="demoService" class="org.apache.dubbo.samples.impl.DemoServiceImpl"/>
    <dubbo:service async="true" interface="org.apache.dubbo.samples.api.DemoService"
        version="1.2.3" group="dubbo-simple" ref="demoService"
        ondisconnect="disCallback"
        executes="4500" retries="7" owner="vict" timeout="5300"/>
</beans>

```

实际也就是对5种典型的网络I/O事件的 connected 和 disconnected 做适配处理，为其创建相应的 RpcInvocation 类的 Invocation 实例，以该实例和被回调方法接受的通道 channel 入参为参数，调用当前被装饰 ExchangeHandler 对象的 received() 方法。当然，只有在配置了链入或者断链的监听方法，对应事件回调中才会运行实际的 received(channel, invocation) 业务代码。

```
private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {
```

JAVA

```
    ...  
  
    @Override  
    public void connected(Channel channel) throws RemotingException {  
        invoke(channel, ON_CONNECT_KEY);  
    }  
  
    @Override  
    public void disconnected(Channel channel) throws RemotingException {  
        if (logger.isDebugEnabled()) {  
            logger.debug("disconnected from " + channel.getRemoteAddress()  
                + ",url:" + channel.getUrl());  
        }  
        invoke(channel, ON_DISCONNECT_KEY);  
    }  
  
    private void invoke(Channel channel, String methodKey) {  
        Invocation invocation = createInvocation(channel, channel.getUrl(), methodKey);  
        if (invocation != null) {  
            try {  
                received(channel, invocation);  
            } catch (Throwable t) {  
                logger.warn("Failed to invoke event method " +  
                    invocation.getMethodName() + "(), cause: " + t.getMessage(), t);  
            }  
        }  
    }  
  
    private Invocation createInvocation(Channel channel, URL url, String methodKey) {  
        String method = url.getParameter(methodKey);  
        if (method == null || method.length() == 0) {  
            return null;  
        }  
  
        RpcInvocation invocation = new RpcInvocation(method, new Class<?>[0], new  
Object[0]);  
        invocation.setAttachment(PATH_KEY, url.getPath());  
        invocation.setAttachment(GROUP_KEY, url.getParameter(GROUP_KEY));  
        invocation.setAttachment(INTERFACE_KEY, url.getParameter(INTERFACE_KEY));  
        invocation.setAttachment(VERSION_KEY, url.getParameter(VERSION_KEY));  
        if (url.getParameter(STUB_EVENT_KEY, false)) {  
            invocation.setAttachment(STUB_EVENT_KEY, Boolean.TRUE.toString());  
        }  
  
        return invocation;  
    }  
};
```

上述 `createInvocation(...)` 方法的实现表明，微服务实例或者引用实例的链入或者断链监听是通过本地存根机制实现的，这里暂时忽略处理，后续在分析PRC中的代理实现时会对类似AOP的存根机制做深入剖析，具体参考《Dubbo服务代理》一文。

其它

有关DubboProtocol的源码实现基本已剖析完，但前文提及的 `requestHandler` 类似于幽灵般的存在，服务实例和服务引用实例都有使用到，它的实现始终是从 `exporterMap` 缓存中获取的，但 `refer()` 产生的 `Invoker` 服务引用实例机会就是漂浮着的存在，并没有被存入 `exporterMap`，仅从当前框架层几乎没法一览全貌，尚待后续。

## ProtocolFilterWrapper >> Protocol

在微服务开发涉及RPC请求的场景中，常常有些和特定业务无关的需求，比如某个接口访问量的数据采集、记录访问日志、设置访问令牌等。类似的场景，在一般类似Spring的开发框架会采用拦截器来实现，同样Dubbo中也提供了类似的机制，拦截服务提供方和服务消费方的RPC调用，Dubbo中类似TPS限额的不少内置特性也是基于这一机制实现。

Dubbo内部给的实现方案是，采用装饰者模式，对 `Protocol` 实现做一层装饰，在其导出微服务实例，或者引出微服务引用实例时，加入一个拦截链，供框架或者应用层纳入更多的特性，大致源码如下：

```
public class ProtocolFilterWrapper implements Protocol {  
    private final Protocol protocol;  
  
    public ProtocolFilterWrapper(Protocol protocol) {  
        if (protocol == null) {  
            throw new IllegalArgumentException("protocol == null");  
        }  
        this.protocol = protocol;  
    }  
  
    @Override  
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
        if (REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {  
            return protocol.export(invoker);  
        }  
        return protocol.export(buildInvokerChain(invoker, SERVICE_FILTER_KEY,  
CommonConstants.PROVIDER));  
    }  
  
    @Override  
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {  
        if (REGISTRY_PROTOCOL.equals(url.getProtocol())) {  
            return protocol.refer(type, url);  
        }  
        return buildInvokerChain(protocol.refer(type, url), REFERENCE_FILTER_KEY,  
CommonConstants.CONSUMER);  
    }  
    ...  
}
```

## Filter 接口定义

Dubbo的内部RPC调用过程是异步的，出站请求和相对应的入站响应是两个界限明显的分段过程，前者不会因为后者还未执行完没有获得结果而阻塞，因此表征原生方法调用的 `Result invoke(Invocation invocation)` 出参 `Result` 会被设计成扩展 `CompletionStage<Result>` 的接口，RPC处理结果是基于响应式回调机制设置给 `Result` 的。同样，其拦截器也应该相应被设计成两阶段式的，如下述接口定义，其中用于响应阶段的 `Listener` 是可选的，如果实现了，需要接口实现类扩展自 `ListenableFilter` 抽象类。

```
@SPI JAVA
public interface Filter {
    /**
     * Does not need to override/implement this method.
     */
    Result invoke(Invoker<?> invoker, Invocation invocation)
        throws RpcException;

    interface Listener {
        void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation);
        void onError(Throwable t, Invoker<?> invoker, Invocation invocation);
    }
}

public abstract class ListenableFilter implements Filter {
    protected Listener listener = null;

    public Listener listener() {
        return listener;
    }
}
```

## 拦截器执行原理

Dubbo中的拦截器的调度实现设计得非常巧妙，尽管其执行也是按顺序挨个执行的，但并没有直接呆板地使用列表遍历的形式，而是采用类似单向链表的形式，上一个拦截器运行完，会接着驱动下一个拦截器来接棒执行。

具体实现上，Dubbo会为每一个 `Filter` 创建并实例化一个 `Invoker` 的匿名内部类，在其 `invoke()` 方法体中执行当前 `Filter` 对象的 `Result invoke(Invoker<?> invoker, Invocation invocation)` 方法，`Filter` 对象所对应的实现类要确保在该方法体内以其两个入参执行类似如下的一个代码片段：

...//前验处理或前置特性业务逻辑实现

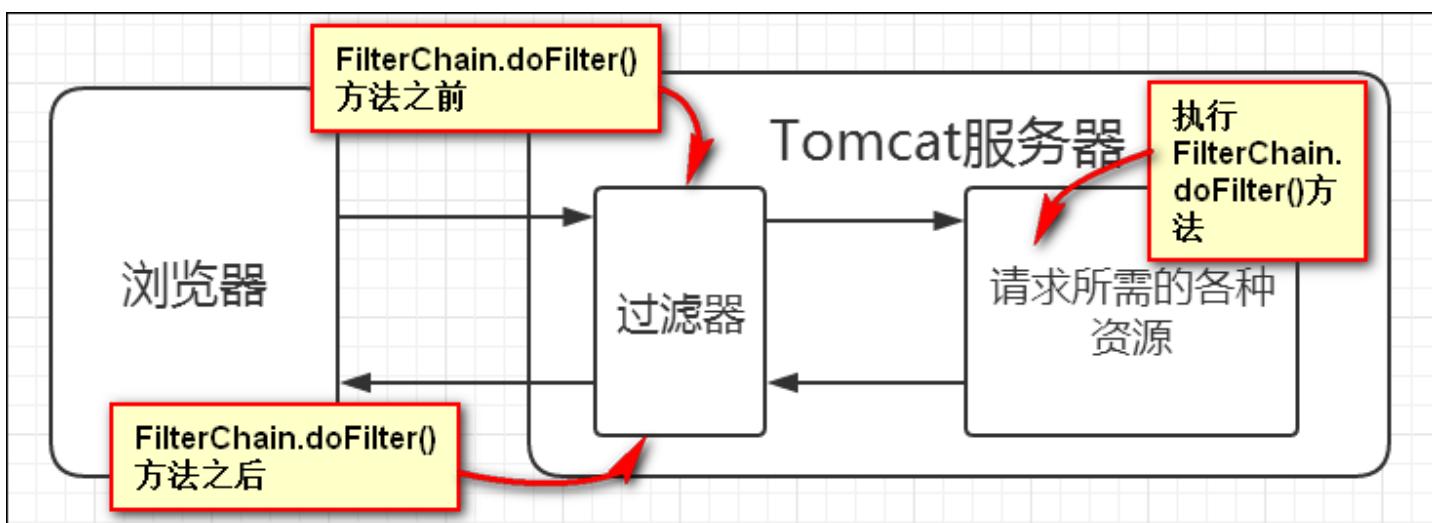
```
Result result = invoker.invoke(invocation);
...//后验处理或后置特性业务逻辑实现
return result;
```

不难看出，在当前 `Invoker` 对象上执行其 `invoke(Invocation)` 方法，其执行结果取决于其在 `Filter` 对象上调用 `invoke(Invoker<?>, Invocation)` 方法传入的首个 `Invoker` 类型入参。

这里可以认为这个入参是被当前对象所属的匿名内部类给装饰了，如果它也是类似被装饰的 `Invoker` 类型对象，那么最后代码执行轨迹就会递归地一直往下调用直到碰到异常或者首次能返回 `Result` 值的为止，随后便由下往上逐层返回这个结果。

可以看出每个 `Invoker` 对象可以根据当前特性需要决定是先执行自身业务逻辑还是先调用它所装饰的另一个 `Invoker` 对象的 `invoke(Invocation)` 方法，也就是说个体上而言，装饰者和被装饰者的逻辑执行顺序是不确定的，但总体而言，经过层层装饰之后形成的递归关系，理解起来感觉比较混乱。然而这种类似AOP的环绕场景，从RPC调用的视觉来看瞬间廓然开朗，就是每一个 `Filter` 装饰者间接等价可以根据自身需要决定逻辑代码的执行时刻：1) 在发出出站请求之前；2) 在收到入站响应之后；3) 上述两个时刻。

熟悉Servlet过滤器实现原理的童鞋此时定会心领神会，笑意舒展，直观如下图所示：



概括下：在顺序上越排在前面的 `Filter`，其前置逻辑越先执行，而后置逻辑则越后执行。

## 拦截器调度源码

弄懂机制后，读源码就比较轻松了，总体实现代码如下：

```

private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, JAVA
String group) {
    Invoker<T> last = invoker;
    List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class)
        .getActivateExtension(invoker.getUrl(), key, group);

    if (!filters.isEmpty()) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new Invoker<T>() {
                @Override
                public Result invoke(Invocation invocation) throws RpcException {
                    Result asyncResult;
                    try {
                        asyncResult = filter.invoke(next, invocation);
                    } catch (Exception e) {
                        // onError callback
                        if (filter instanceof ListenableFilter) {
                            Filter.Listener listener = ((ListenableFilter)
                                filter).listener();
                            if (listener != null) {
                                listener.onError(e, invoker, invocation);
                            }
                        }
                        throw e;
                    }
                    return asyncResult;
                }
                ...
            };
        }
    }
    return new CallbackRegistrationInvoker<>(last, filters);
}

```

上述这段代码中，有几个需要注意的地方：

1. 入参 `invoker` 对象是作为首个被装饰者出现的，也即它是实际处理RPC调用的微服务实例或者微服务引用实例，使用 `getActivateExtension` 获取到的所有 `filters` 已经按照优先级排好序，越高的越靠近底层的RPC调用。
2. `Invocation` 类型入参基本不会发生变化，是伴随整个拦截链的，隐含的意思是，`Fitler` 实现可以根据需要在其本地参数容器存入相应的键值对，让其在链中传播，供其他 `Fitler` 实现联动逻辑。这种特性也适用于可携带本地参数容器的 `Result`。
3. `catch` 代码块表示，拦截链中包括当前节点在内的其它前驱中某个 `Filter` 两个阶段都可能会发生处理出现了异常，并且显示地 `Throw` 出来了，若这些 `Filter` 属监听型，则回调其 `onError()`，通知异常发生。另外异常也可以由 `Result``` 携带返回，这后面一种类型的异常处理，整个拦截链依然可以无感知地继续 `work`。

另外上述有关 `Invoker` 内部类实现逻辑中省略了如下的代码段，结合上述代码，不难理解最初被装饰的那个 `invoker` 对象始终是业务逻辑运行的主战场，其它环绕它执行的 `Invoker` 是独立于业务逻辑之外的增强和补充，因而链上的所有 `Invoker` 节点都使用 `invoker` 获取相关状态。

```
new Invoker<T>() {  
  
    @Override  
    public Class<T> getInterface() {  
        return invoker.getInterface();  
    }  
  
    @Override  
    public URL getUrl() {  
        return invoker.getUrl();  
    }  
  
    @Override  
    public boolean isAvailable() {  
        return invoker.isAvailable();  
    }  
    @Override  
    public void destroy() {  
        invoker.destroy();  
    }  
  
    @Override  
    public String toString() {  
        return invoker.toString();  
    }  
    ...  
}
```

## Filter 共享反馈结果

拦截器实现中，并非所有内部 `Invoker` 装饰者等价于 Filter 实现会回调 `onError()`，结果正常时也不会被回调 `onResponse()`，如果异常结果携带在出参 `Result` 中，这些回调就根本不会发生。而特性上要求所有加入到链中的 Filter，只要有要求均能在有结果包括 Exception 获得通知。因此在上一章节中代码片段中出现了 `return new CallbackRegistrationInvoker<>(last, filters)`，它表示 `last` 这个 `Invoker` 对象最后又被装饰了一次，目的是让链所有的 `ListenableFilter` 能通过回调感知到处理结果，如下述源码所示：

```
static class CallbackRegistrationInvoker<T> implements Invoker<T> {
```

JAVA

```
    private final Invoker<T> filterInvoker;
    private final List<Filter> filters;

    public CallbackRegistrationInvoker(Invoker<T> filterInvoker, List<Filter> filters)
    {
        this.filterInvoker = filterInvoker;
        this.filters = filters;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        Result asyncResult = filterInvoker.invoke(invocation);

        asyncResult = asyncResult.whenCompleteWithContext((r, t) -> {
            for (int i = filters.size() - 1; i >= 0; i--) {
                Filter filter = filters.get(i);
                // onResponse callback
                if (filter instanceof ListenableFilter) {
                    Filter.Listener listener = ((ListenableFilter) filter).listener();
                    if (listener != null) {
                        if (t == null) {
                            listener.onResponse(r, filterInvoker, invocation);
                        } else {
                            listener.onError(t, filterInvoker, invocation);
                        }
                    }
                } else {
                    filter.onResponse(r, filterInvoker, invocation);
                }
            }
        });
        return asyncResult;
    }
    ....类似上一章节最后呈现的那段代码
}
```

细究两处异常处理实现，如果前者先发生，后者是没有机会执行的，可以认为前者是短路型异常处理。原因是 `CallbackRegistrationInvoker` 是最后一个被执行的 `Invoker` 装饰者对象。不论是哪种方案，`onError` 的回调时机都处于拦截链回退的途中。

---

完结

# Dubbo RPC 之 Protocol协议层（三）

---

在《Dubbo RPC 之 Protocol协议层（二）》中已经深入地阐述了拦截链的运行机制了，本文将着重讲讲其应用。在一些技术类的文章中，特别是公众号，总会把微服务中的一些技术点吹的神乎其神，读者总是感觉一愣一愣的，由于机制的讲解和具体实现往往存在着很大的鸿沟，难免会让人疑团重重，产生强烈的焦虑感。本序列文章一方面全盘剖析一个微服务 框架底层到底是如何实现，另外一方面也是旨在帮助由单体式架构迁移到微服务这一领域的开发者全方位了解微服务、了解分布式开发，因此每次涉及的重要技术细节，行文风格总有打破砂锅问到底的味道。

本文将由易及难，逐个阐述基于拦截链中过滤器实现支持的功能。

## 服务实例之 ClassLoader 保持

《Dubbo类管理与 ClassLoader》一文中曾提及：仰赖于不同的 ClassLoader 对象加载签名完全一样的类时，JVM 并不会认为他们是同一个类，因而可以通过给当前 Thread 设置 contextClassLoader 对所加载的类做版本隔离管理。

Dubbo 在拦截链中实现了 ClassLoaderFilter 过滤器，其实现相对而言比较简单，首先调用 Thread.currentThread().getContextClassLoader() 将其缓存起来，记为 A，然后获取入参 invoker 微服务实例服务接口所使用的类加载器，记为 B，将 B 设置给当前线程，待 invoker.invoke(invocation) 执行完后，将当前线程的类加载器恢复为 A。

注：在拦截链中，晚于当前节点的所有 Filter 对象和入参 invoker 所表示的微服务实例都将使用 B 这个加载器进行类的加载处理。

```

JAVA
/**
 * Set the current execution thread class loader to service interface's class loader.
 */
@Activate(group = CommonConstants.PROVIDER, order = -30000)
public class ClassLoaderFilter implements Filter {

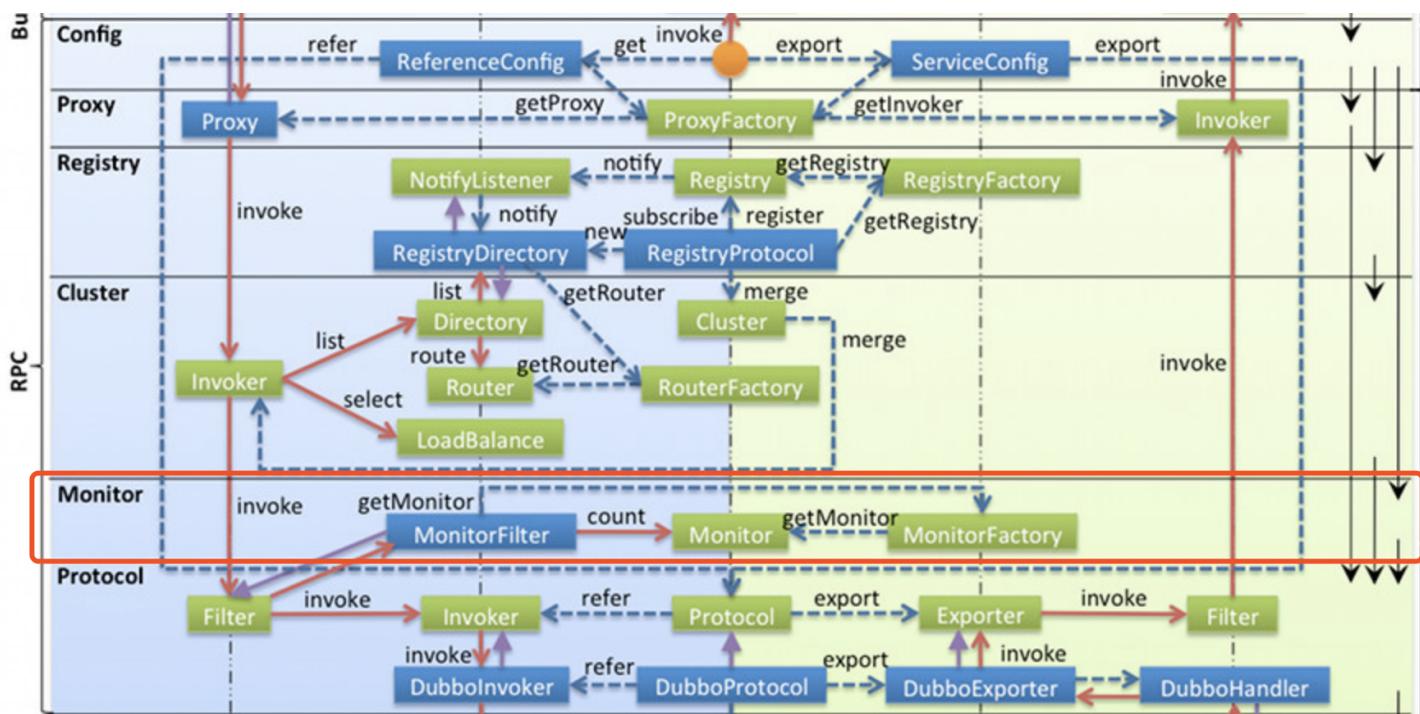
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException
    {
        ClassLoader ocl = Thread.currentThread().getContextClassLoader();

        Thread.currentThread().setContextClassLoader(invoker.getInterface().getClassLoader());
        try {
            return invoker.invoke(invocation);
        } finally {
            Thread.currentThread().setContextClassLoader(ocl);
        }
    }
}

```

## 服务监控

Dubbo的框架分层中，有一层称为监控层的，薄薄的一片，很简洁，实际实现代码也没有多少内容，放在本文阐述主要是因为它也是使用 `Filter` 拦截链机制实现的。而它能作为一个抽象层被单独划分在框架图示中，也足矣说明其重要性，微服务框架中，服务治理占据着比较大的比分，必须依赖于监控得出的各项统计数据，包括服务的调用次数、调用时间等，该数据需要定时周期地发送到监控中心服务器。



在进一步阐述其实现前，先普及下预先认知：

`Filter` 是一个扩展点，其具类 `MonitorFilter` 是单例模式的，围绕着跨网络请求的实际RPC方法调用有事先执行的 前置逻辑 先于 `invoker.invoke(invocation)` 和事后执行的 后置逻辑 `Filter.Listener` 接口回调。

Dubbo中的服务监控的基本实现方案是以RPC方法为单位由

`Statistics[application,service,method,group,version,client,server]`共同决定，先基于拦截链机制使用 `MonitorFilter` 采集每次RPC方法调用的数据，将其装载在服务总线中发送给 `Monitor`，后者会将一个周期内的所有接受到的数据汇总起来发送给监听服务器。

注：本章节中的下述中的所有讨论都是基于假设：服务实例是需要统计监控数据的，其配置总线中已经含有 "monitor" 配置项。

## MonitorFilter

拦截链中的 `MonitorFilter` 负责的是单次数据的采集，针对每次服务调用，其基本实现方式是：

1. 在 `MonitorFilter` 的前置逻辑中执行：

- a. 给表征当前RPC方法调用的 `Invocation` 实例本地参数容器中置入  
`<"monitor_filter_start_time", System.currentTimeMillis()>`；
- b. 使用 `invoker.getInterface().getName() + "." + invocation.getMethodName()` 从记录微服务方法当前并发次数的 `ConcurrentMap<String, AtomicInteger>` 容器中获得对应计数器 `my_counter`，执行 +1 操作；

2. 在基于通知回调方式执行后置逻辑中执行：

- a. 无论是在 `onResponse()` 还是 `onError()` 回调中，都获取 `my_counter` 执行 -1 操作；
- b. 采集包括 `my_counter` 在内的所有如下数据(分3组)：

i. 配置总线URL基础组成Dubbo中的复合数据使用配置总线携带传递，需满足关键构成：

- `protocol: "count"`
- `host: NetUtils.getLocalHost()`
- `port: 0|invoker.url.getPort()`
- `path: invoker.getInterface().getName() + "/" + RpcUtils.getMethodName(invocation)`

ii. `Statistics` 键用于唯一确定一个RPC方法的组合数据：

- `<"application", invoker.url["application"]>`
- **service:** `<"interface", invoker.getInterface().getName()>`
- `<"method", RpcUtils.getMethodName(invocation)>`

- <"group",invoker.url["group"]>
- <"version",invoker.url["version"]>
- **client | server**: <"consumer",RpcContext.getContext().getRemoteHost()> | <"provider",invoker.url.getAddress()>

iii. 统计项 监控层需要的各项统计数据：

- <"concurrent",my\_counter>
- <"success"|"failure",1>
- <"elapsed",System.currentTimeMillis() - invocation.url["monitor\_filter\_start\_time"]>
- <"input",invocation.url["input"]>
- <"output",invocation.url["output"]>

c. 使用采取数据构建配置总线URL实例，形如： count://host/interface?

```
application=foo&method=foo&provider=10.20.153.11:20880&success=12&failure=2&elapsed=135423423
```

3. 获取 `Monitor` 实例，将得到的URL实例传递给它；

注：input和output指的是入站接收到的或出站发送出去的数据比特量

## 源码实现注意点

## Monitor

`Monitor` 负责数据的收集处理，并将单位间隔时间的每个RPC方法的所有调用汇总并发送给监控服务器。

## 数据汇总

汇总数据的计算方式是每次 `Monitor` 收到数据后，针对接收到的由配置总线URL携带的数据进行如下操作：

1. 取出用于构建 `Statistics` 实例A的组成数据 `Statistics` 键；
2. 使用实例A获得对应RPC方法的既有监控数据项（统计数据装在 `ConcurrentMap<Statistics,AtomicReference<long[]>>` 类型的 `statisticsMap` 容器中）；
3. 进行如下数据汇总处理记为 `SummaryData`：
  - <"success", + url["success"]>
  - <"failure", + url["failure"]>
  - <"input", + url["input"]>

- <"output", + url["output"]>
- <"elapsed", + url["elapsed"]>
- <"concurrent", (+ url["elapsed"])/2>
- <"max.input", max(,url["input"])>
- <"max.output", max(,url["output"])>
- <"max.elapsed", max(,url["elapsed"])>
- <"max.concurrent", max(,url["concurrent"])>

## 实现源码剖析

下述从更加宏观的粒度按特性分组梳理下整个 `DubboMonitor` 的实现结构。

### 监控数据的周期推送

Monitor周期性的发送监控数据给监控服务器，其实现原理是使用 `ScheduledExecutorService` 大概每1分钟，遍历 `statisticsMap` 容器中的汇总的所有RPC方法的统计数据，逐个做如下处理：

注：遍历元素：`Map.Entry<Statistics, AtomicReference<long[]>> entry`，值是一个含有10个元素的数组，和 `SummaryData` 一一对应

1. 获取统计数据 `SummaryData`，按照既有键值对模式构建URL实例 `urlData`；
2. 使用监控微服务 `MonitorService` 的引用实例，使用 `urlData` 唤起对RPC方法 `collect()` 的调用；
3. 对 `entry` 中的 `AtomicReference<long[]>` 类型值做清零处理；

JAVA

```

public class DubboMonitor implements Monitor {
    private final ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(3,
            new NamedThreadFactory("DubboMonitorSendTimer", true));

    private final ScheduledFuture<?> sendFuture;

    public DubboMonitor(Invoker<MonitorService> monitorInvoker, MonitorService
monitorService) {
        this.monitorInvoker = monitorInvoker;
        this.monitorService = monitorService;
        this.monitorInterval = monitorInvoker.getUrl().getPositiveParameter("interval",
60000);
        // collect timer for collecting statistics data
        sendFuture = scheduledExecutorService.scheduleWithFixedDelay(() -> {
            try {
                // collect data
                send();
            } catch (Throwable t) {
                logger.error("Unexpected error occur at send statistic, cause: " +
t.getMessage(), t);
            }
        }, monitorInterval, monitorInterval, TimeUnit.MILLISECONDS);
    }

    @Override
    public void destroy() {
        try {
            ExecutorUtil.cancelScheduledFuture(sendFuture);
        } catch (Throwable t) {
            logger.error("Unexpected error occur at cancel sender timer, cause: " +
t.getMessage(), t);
        }
        monitorInvoker.destroy();
    }
    ...
}

```

### 针对 entry 值的设置和清零处理

Dubbo微服务中，RPC方法调用是一种高并发场景，因而其监控数据的统计也需要做对应支持，否则汇总出来的数据会存在gap，失之毫厘，谬以千里，为保证准确性Dubbo实现中使用了 `AtomicReference`，利用其CAS机制做设值和清零处理。如下述代码含部分伪代码所示：

```

private final ConcurrentHashMap<Statistics, AtomicReference<long[]>> statisticsMap = new
ConcurrentHashMap(); JAVA

public void send() {
    ...
    for (Map.Entry<Statistics, AtomicReference<long[]>> entry :
statisticsMap.entrySet()) {
        ...//使用collect()方法汇总的数据构建URL配置总线实例,
        //将汇总数据通过monitorService服务引用实例发送给监控服务器
        monitorService.collect(url);
        // get statistics data
        Statistics statistics = entry.getKey();
        AtomicReference<long[]> reference = entry.getValue();
        //发送汇总数据后的清零处理
        // reset
        long[] current;
        long[] update = new long[LENGTH];
        do {
            current = reference.get();
            if (current == null) {
                update[0,5] = 0;
            } else {
                update[0,5] = current[0,5] - url[***];
            }
        } while (!reference.compareAndSet(current, update));
        ...
    }
    ...
}

public void collect(URL url) {
    ...
    //汇总设值处理
    // use CompareAndSet to sum
    long[] current;
    long[] update = new long[LENGTH];
    do {
        current = reference.get();
        if (current == null) {
            update = success;
        } else {
            update = (current + url[***]) | max(current,url[***]);
        }
    } while (!reference.compareAndSet(current, update));
    ...
}

```

上述源码中的清零处理中的 else 块理解起来比较费劲，背后的原因是 send() 和 collect(URL url) 在并发环境下执行，尽管可以认为 send() 是串行执行的，但是 collect(URL url) 并发量很大，和它存在资源争用的问题。比如在 send() 方法调用完向监控服务发送数据的RPC方法后，get 到 entry 值，将清零后的值放在 update 临时变量中，随后打算设回到 entry，但是 compareAndSet(current, update) 返回 false，也就是此前瞬间有个过程 collect(URL url) 改变了它的值，如果此时再次 CAS 简单的清零处理，那么后面这个汇总步骤就相当于丢失了。

## Monitor 和 MonitorService

文中已经提到 Monitor 监控统计的数据最终会调用RPC接口发送到监控服务器，这个跨机器的接口使用 MonitorService 表示，而实际上 Monitor 也实现自该接口，也就是说 Monitor 的实现 DubboMonitor 可以认为是前者的代理，如下述源码所示：

```
public interface Monitor extends Node, MonitorService {  
}  
  
public interface MonitorService {  
    .../其它组成SummaryData需要使用的常量键  
  
    void collect(URL statistics);  
  
    List<URL> lookup(URL query);  
}  
  
public class DubboMonitor implements Monitor{  
  
    private final Invoker<MonitorService> monitorInvoker;  
  
    private final MonitorService monitorService;  
  
    ...  
  
    @Override  
    public List<URL> lookup(URL query) {  
        return monitorService.lookup(query);  
    }  
  
    @Override  
    public URL getUrl() {  
        return monitorInvoker.getUrl();  
    }  
  
    @Override  
    public boolean isAvailable() {  
        return monitorInvoker.isAvailable();  
    }  
  
    @Override  
    public void destroy() {  
        ...  
        monitorInvoker.destroy();  
    }  
}
```

## 并发量控制

瞬间大量涌入的RPC请求势必造成服务Provider短时间处于高负荷承载状态，原本正常范围能顺利处理的请求迟迟没法响应，更糟糕的是可能会因为机器反应不过来而导致宕机风险，因此对于某些CPU密集型或者I/O密集型任务，需要做相应的并发配置。

Dubbo可以针对微服务的服务实例和引用实例分别做并发数配置，其配置可以是方法级别的，如下：

```
<!-- 服务端配置方式 -->
<dubbo:service interface="com.foo.BarService" executes="10">
    <dubbo:method name="sayHello" executes="5" />
    <dubbo:method name="echo"/>
</dubbo:service>

<!-- 客户端配置方式 -->
<dubbo:reference interface="com.foo.BarService" actives="10">
    <dubbo:method name="sayHello" actives="5" />
    <dubbo:method name="echo"/>
</dubbo:service>
```

上述配置，在服务导出或者服务引入的时候会被转换成配置总线URL中对应的项，方法级别的会加上前缀，形如 `methodName+ "." + configItem`，只有加上了如上配置，就会自动激活用于并发控制的 Filter 扩展点具类—— `ExecuteLimitFilter` 或 `ActiveLimitFilter`。因服务端和客户端要采集的数据项是一致的，思路基本也一致，因而大部分公共逻辑都被抽离到 `RpcStatus` 这个类加以处理。

```
@Activate(group = CommonConstants.PROVIDER, value = EXECUTES_KEY)
public class ExecuteLimitFilter extends ListenableFilter {...}

@Activate(group = CONSUMER, value = ACTIVES_KEY)
public class ActiveLimitFilter extends ListenableFilter {...}
```

## RpcStatus

正如其名字所示，`RpcStatus` 是用于记录一个微服务RPC方法当前状态的行为类，自微服务的服务实例或引用实例启动开始就开始记录，在运行的任何时候使用它都能获取到一个目标RPC方法的当前状态。若已配置，`ExecuteLimitFilter` 或 `ActiveLimitFilter` 在微服务导出或者引入时随拦截链生成而激活，后续的每一次RPC方法被调用或调用时都会改变与之相绑定的 `RpcStatus` 的状态值。

`RpcStatus` 采集的当前状态值包括如下几项，所有这些变量都原子的RPC方法在并发环境下被调用，实例化开始时就经过初始化的，并且声明为final的：

- 并发数： `AtomicInteger active`
- 调用总数： `AtomicLong total`
- 失败总数： `AtomicInteger failed`
- 总计耗时： `AtomicLong totalElapsed`

- 失败总计耗时: AtomicLong failedElapsed
- 最大耗时: AtomicLong maxElapsed
- 最大失败耗时: AtomicLong failedMaxElapsed
- 最大成功耗时: AtomicLong succeededMaxElapsed

下述数据是可以依赖上述变量获取的，这些推导类方法属于 `RpcStatus` 实例本身：

- 平均TPS: `getAverageTp()`:  $\text{totalElapsed} < 1000 ? \text{total} : \text{total} \div (\text{totalElapsed} \div 1000)$
- 成功总数: `getSucceeded()`: `total - failed`
- 成功总计耗时: `getSucceededElapsed()`: `totalElapsed - failedElapsed`
- 平均成功耗时: `getSucceededAverageElapsed()`: `getSucceededElapsed() / getSucceeded()`
- 平均失败耗时: `getFailedAverageElapsed()`: `failedElapsed / failed`
- 平均耗时: `getAverageElapsed()`: `totalElapsed / total`

上述提过 `RpcStatus` 统计的状态可以是方法级别的，也可以是服务级别的，并且每一个配置了并发数参数的服务或者方法都需要隐式绑定的一个 `RpcStatus` 实例，因此声明如下两个静态变量，用于映射关系并缓存这些实例的容器：

```

private static final ConcurrentHashMap<String, RpcStatus> SERVICE_STATISTICS
    = new ConcurrentHashMap<>();

private static final ConcurrentHashMap<String, ConcurrentHashMap<String, RpcStatus>>
METHOD_STATISTICS
    = new ConcurrentHashMap<>();

//声明为私有
private RpcStatus() {}

public static RpcStatus getStatus(URL url) {
    String uri = url.toIdentityString();
    RpcStatus status = SERVICE_STATISTICS.get(uri);
    if (status == null) {
        SERVICE_STATISTICS.putIfAbsent(uri, new RpcStatus());
        status = SERVICE_STATISTICS.get(uri);
    }
    return status;
}

public static void removeStatus(URL url) {
    String uri = url.toIdentityString();
    SERVICE_STATISTICS.remove(uri);
}

public static RpcStatus getStatus(URL url, String methodName) {
    String uri = url.toIdentityString();
    ConcurrentHashMap<String, RpcStatus> map = METHOD_STATISTICS.get(uri);
    if (map == null) {
        METHOD_STATISTICS.putIfAbsent(uri, new ConcurrentHashMap<String, RpcStatus>());
        map = METHOD_STATISTICS.get(uri);
    }
    RpcStatus status = map.get(methodName);
    if (status == null) {
        map.putIfAbsent(methodName, new RpcStatus());
        status = map.get(methodName);
    }
    return status;
}

public static void removeStatus(URL url, String methodName) {
    String uri = url.toIdentityString();
    ConcurrentHashMap<String, RpcStatus> map = METHOD_STATISTICS.get(uri);
    if (map != null) {
        map.remove(methodName);
    }
}

```

其中用于用于隐式绑定服务和服务的方法的二级Key键分别是：

- 一级Key键服务级，由url.toIdentityString()取得： [protocol + "://" + [username[":" + password] + "@"][host[":" + port]]["/" + path]
- 二级Key键方法级： (url.toIdentityString() + ) methodName

无论是用于服务端还是客户端的状态统计，基本思路都是一致的，在进入Filter的前置逻辑中先执行检验当前RPC方法调用是否操作最大并发数的限制，如果超过就会被拒绝。RpcStatus类提供了相应的静态方法，超过 active 值返回false，拒绝逻辑由调用它的Filter负责，如下：

```
public static void beginCount(URL url, String methodName) {  
    beginCount(url, methodName, Integer.MAX_VALUE);  
}  
  
public static boolean beginCount(URL url, String methodName, int max) {  
    max = (max <= 0) ? Integer.MAX_VALUE : max;  
    RpcStatus appStatus = getStatus(url);  
    RpcStatus methodStatus = getStatus(url, methodName);  
    if (methodStatus.active.get() == Integer.MAX_VALUE) {  
        return false;  
    }  
    if (methodStatus.active.incrementAndGet() > max) { // TagX  
        methodStatus.active.decrementAndGet();  
        return false;  
    } else {  
        appStatus.active.incrementAndGet();  
        return true;  
    }  
}
```

上述 TagX 标识的前的逻辑判断及其后的代码块中，先执行 incrementAndGet()，在发现超过限制后，随后立马又执行 decrementAndGet()。为啥不直接取得值同 max 对比？文中我们一直强调，RPC方法是在并发环境下执行的，因为这个原因 RpcStatus 把所有的用于统计状态的属性都定义成了原子变量，假设同一瞬间，有两个方法都进入了TagX前的判断逻辑，发现都满足要求，也就是并没有超过并发量，这时就都会执行 else{...} 逻辑块，也即都执行了 incrementAndGet()，这样就会出现最终并发量会比实际配置要大的情况。

绝大部分的统计只能在Filter的响应逻辑中通过回调异步执行，因而本章节使用到的两个 Filter 扩展自 ListenableFilter。有关执行统计的逻辑并不复杂，如下所示：

```
public static void endCount(URL url, String methodName, long elapsed, boolean succeeded) {
    endCount(getStatus(url), elapsed, succeeded);
    endCount(getStatus(url, methodName), elapsed, succeeded);
}

private static void endCount(RpcStatus status, long elapsed, boolean succeeded) {
    status.active.decrementAndGet();
    status.total.incrementAndGet();
    status.totalElapsed.addAndGet(elapsed);
    if (status.maxElapsed.get() < elapsed) {
        status.maxElapsed.set(elapsed);
    }
    if (succeeded) {
        if (status.succeededMaxElapsed.get() < elapsed) {
            status.succeededMaxElapsed.set(elapsed);
        }
    } else {
        status.failed.incrementAndGet();
        status.failedElapsed.addAndGet(elapsed);
        if (status.failedMaxElapsed.get() < elapsed) {
            status.failedMaxElapsed.set(elapsed);
        }
    }
}
```

JAVA

## ExecuteLimitFilter

详解完 `RpcStatus` 后，理解 `ExecuteLimitFilter` 的实现就很简单了。首先看看基于响应的回调逻辑，父类 `ListenableFilter` 要求子类提供 `Filter` 中定义的 `Listener` 回调接口实现，`ExecuteLimitListener` 即为实现。`Filter` 前置逻辑中记录了RPC方法被调用的时间，其值以 "execugtelimit\_filter\_start\_time" 为标识记录在RPC方法参数 `invocation` 对象的参数容器中。

```
@Activate(group = CommonConstants.PROVIDER, value = EXECUTES_KEY)
public class ExecuteLimitFilter extends ListenableFilter {
```

```
    private static final String EXECUTELIMIT_FILTER_START_TIME =
"execugelimit_filter_start_time";

    public ExecuteLimitFilter() {
        super.listener = new ExecuteLimitListener();
    }
    ...
    static class ExecuteLimitListener implements Listener {
        @Override
        public void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation) {
            RpcStatus.endCount(invoker.getUrl(), invocation.getMethodName(),
                getElapsed(invocation), true);
        }

        @Override
        public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
            if (t instanceof RpcException) {
                RpcException rpcException = (RpcException)t;
                if (rpcException.isLimitExceed()) {
                    return;
                }
            }
            RpcStatus.endCount(invoker.getUrl(), invocation.getMethodName(),
                getElapsed(invocation), false);
        }
    }

    private long getElapsed(Invocation invocation) {
        String beginTime =
invocation.getAttachment(EXECUTELIMIT_FILTER_START_TIME);
        return StringUtils.isNotEmpty(beginTime) ?
            System.currentTimeMillis() - Long.parseLong(beginTime) : 0;
    }
}
```

JAVA

ExecuteLimitListener 异常响应回调逻辑处理中，对因超过并发量抛出的异常直接忽略处理。服务端对接入请求检测超过并发量时就会直接抛异常，这类型的异常 RpcStatus 不做统计。

```

public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    URL url = invoker.getUrl();
    String methodName = invocation.getMethodName();
    int max = url.getMethodParameter(methodName, EXECUTES_KEY, 0);
    if (!RpcStatus.beginCount(url, methodName, max)) {
        throw new RpcException(RpcException.LIMIT_EXCEEDED_EXCEPTION,
            "Failed to invoke method " + invocation.getMethodName() + " in provider"
        +
            url + ", cause: The service using threads greater than
<dubbo:service executes=\"" + max +
            "\" /> limited.");
    }
    invocation.setAttachment(EXECUTELIMIT_FILTER_START_TIME,
    String.valueOf(System.currentTimeMillis()));
    try {
        return invoker.invoke(invocation);
    } catch (Throwable t) {
        if (t instanceof RuntimeException) {
            throw (RuntimeException) t;
        } else {
            throw new RpcException("unexpected exception when ExecuteLimitFilter", t);
        }
    }
}

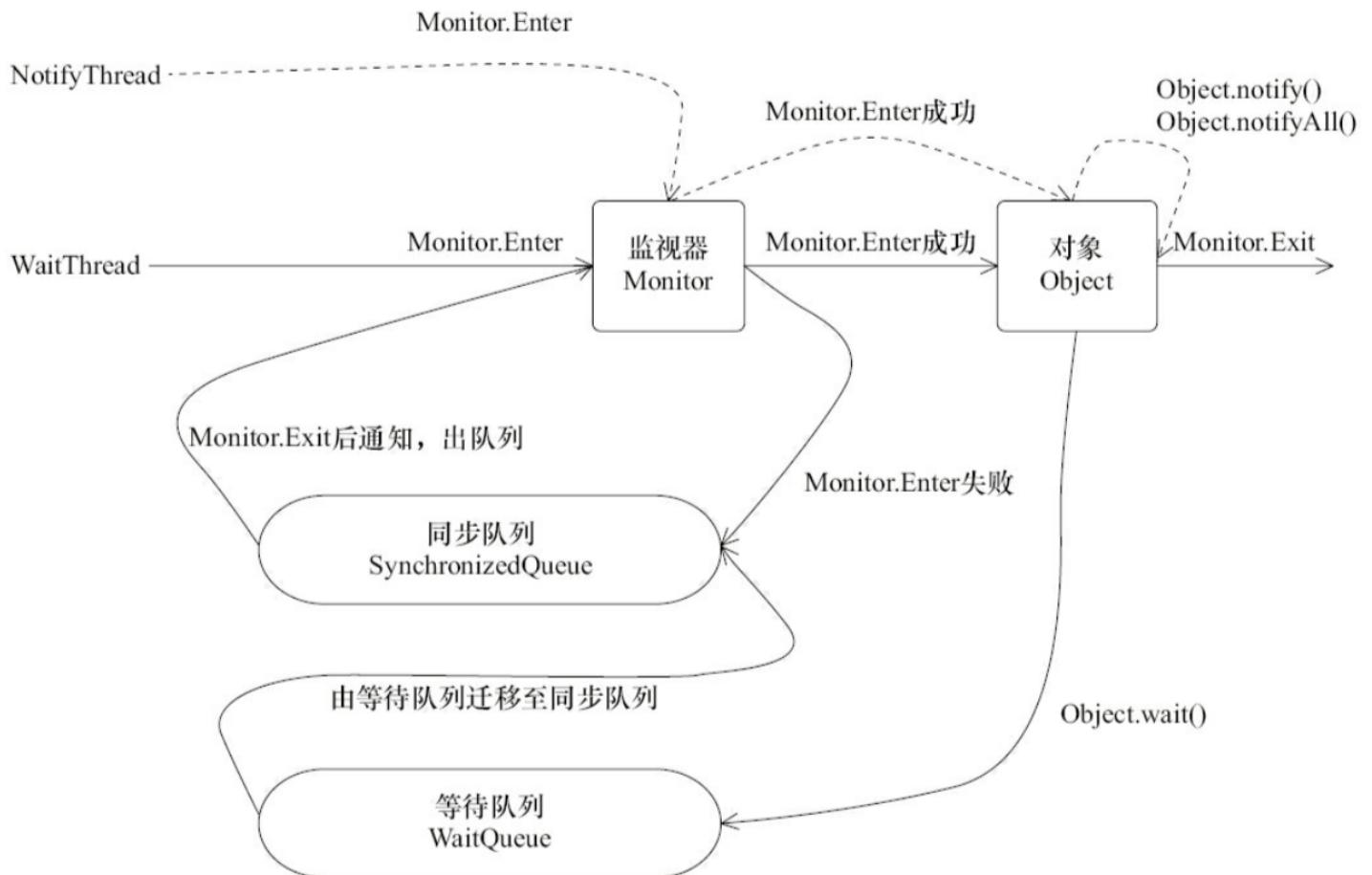
```

在剖析拦截链相关的章节中已经提到，`Filter` 构成一条拦截链，先挨个执行前置逻辑，到最后才执行真正的RPC方法调用，执行完后反向顺序挨个执行这些 `Filter` 的后置逻辑，链上的任意点抛出异常都会结束这个流程。一个 `Filter` 如果提供了 `Filter.Listener` 实现，那么链上发生的任何异常都会通过回调其 `onError()` 方法告知。

## ActiveLimitFilter

相比服务端，客户端的 `ActiveLimitFilter` 实现比较复杂点，当前并发数已超额的情况下，不能直接对客户端程序提交的RPC请求做拒绝处理，应该尽可能地在超时之前让其有机会执行——等并发数下降第一时间获得计算资源。超额说明隐式绑定于当前RPC方法的 `RpcStatus` 对象在多线程环境下是一个争用资源，因而可以结合锁机制让当前RPC方法获得执行机会。

在进一步剖析的 `ActiveLimitFilter` 实现前，我们先看看如下图所示的Java锁中的等待-通知机制。



- “
1. 每个互斥锁，也即图中的由对象持有的Monitor监视器，都持有两个队列，分别是同步队列和等待队列；
  2. 同一时刻，只允许一个线程进入 synchronized 保护的临界区，若已有线程进入临界区，其他线程只能进入同步队列等待；
  3. 在临界区的线程WaitThread，发现某些条件不满足，调用 wait() 方法，释放持有的互斥锁，进入等待队列；
  4. 同步队列中的其他线程NotifyThread 获得锁并进入临界区，当线程 WaitThread要求的条件满足时，它通过 notify()/notifyAll() 发出通知；
  5. 处于等待队列中的线程WaitThread 转移到同步队列，如果竞争获得锁则从 wait() 返回重新回到临界区执行剩下的逻辑代码；

## NOTE

1. `sleep` 不会释放互斥锁;
2. 执行 `notify()`/`notifyAll()` 并不会释放互斥锁，在 `synchronized` 代码块结束后才真正的释放互斥锁;
3. `notify()`/`notifyAll()` 执行后会将线程从等待队列移入到同步队列，前者挑选一个，后者则取得所有，被移动的线程状态由 `WAITING` 变成 `BLOCKED`；

回到正题，当 `ActiveLimitFilter` 发现当前RPC方法在所属服务引用实例上已经超过了配置并发数，便开始执行如下逻辑：

1. 基于配置项 `invoker.url[methodName + "." + "timeout"]` 获取超时时间 `timeout`
2. 将剩下超时时间设为 `remain = timeout`，并记录当前时间 `start`
3. 尝试使用 `synchronized` 获取到对应的 `rpcStatus` 锁，没有获取到则在此位置阻塞，否则在其临界区执行如下循环逻辑：
  - a. 若检测到当前并没有超过并发数，则释放锁退出当前临界区；
  - b. 调用 `rpcStatus.wait(remain)` 释放锁进入条件队列；
  - c. 超时或者因通知重新获得锁后，计算剩下的超时时间 `remain = timeout - System.currentTimeMillis() + start;`；
  - d. 如果 `remain <= 0` 则抛错处理，否则进入下一次循环；

```

@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    ...
    if (!RpcStatus.beginCount(url, methodName, max)) {
        long timeout = invoker.getUrl().getMethodParameter(invocation.getMethodName(),
TIMEOUT_KEY, 0);
        long start = System.currentTimeMillis();
        long remain = timeout;
        synchronized (rpcStatus) {
            while (!RpcStatus.beginCount(url, methodName, max)) {
                try {
                    rpcStatus.wait(remain);
                } catch (InterruptedException e) {
                    // ignore
                }
                long elapsed = System.currentTimeMillis() - start;
                remain = timeout - elapsed;
                if (remain <= 0) {
                    throw new RpcException(RpcException.LIMIT_EXCEEDED_EXCEPTION,
                        "Waiting concurrent invoke timeout in client-side for
service: "
                        + ... + ". max concurrent invoke limit: " + max);
                }
            }
        }
    }
    ...
}

```

上述代码中，使用 `synchronized + wait` 组合在某个线程X中等待条件的出现，根据语法规则，需要由 `synchronized + notify|notifyAll` 组合在另外一个线程Y条件满足时通知X，那究竟谁是Y？我们已经清楚，RPC方法执行完后会通过异步回调 `ActiveLimitListener` 的 `onResponse` 和 `onError` 方法告知执行结果，而他们是由 `CompletableFuture` 新产生的线程负责执行的，这个执行回调的线程也即我们要找的Y，具体可以参看《Dubbo RPC 之 Protocol协议层（一）》中的讲述 `subscribeTo` & `whenCompleteWithContext` 这一章节。

如下，定义了 `notifyFinish()`，该方法直接获取 `rpcStatus` 锁执行 `notifyAll()` 通知。从并发数控制这一场景来讲，当一个RPC请求XReq发现超额时，肯定是当前针对同一个RPC方法的正在运行的请求已经满额，XReq随后便会执行加锁处理，并且紧接着大概率调用了 `wait()` 让自己进入等待队列，因而先获得锁的线程在处理完请求后，不管三七二十一先获取锁，随后便 `notifyAll()` 通知所有处于WAITING状态的线程转移到BLOCKING等锁状态。能够获取锁要么说明在同步队列竞争获锁成功，要么说明本身并不超额没有竞争就获得锁，也就是说这里可以采用双检机制结合前文提到 `TagX` 方式进行效率优化，虽然当前逻辑处于异步的回调中，但它却决定了后续其它RPC请求是否能更快得到处理。

```
static class ActiveLimitListener implements Listener {
    @Override
    public void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation) {
        ...
        int max = invoker.getUrl().getMethodParameter(methodName, ACTIVES_KEY, 0);
        notifyFinish(RpcStatus.getStatus(url, methodName), max);
    }

    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
        ...
        int max = invoker.getUrl().getMethodParameter(methodName, ACTIVES_KEY, 0);
        notifyFinish(RpcStatus.getStatus(url, methodName), max);
    }

    private void notifyFinish(final RpcStatus rpcStatus, int max) {
        if (max > 0) {
            synchronized (rpcStatus) {
                rpcStatus.notifyAll();
            }
        }
    }
}
```

JAVA

# 【十一】Dubbo集群之负载均衡

“ In computing, load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy.

上述来自于维基百科中关于“负载均衡”的介绍，大致意思是：

“ 在计算机应用中，负载均衡可改善跨多个计算资源（例如计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器）的工作负载分配。它旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单个资源的过载。负载均衡利用冗余方式使用多个组件而不是单个组件提高可靠性和可用性。

## 负载均衡和微服务

在微服务开发领域，可以认为：1) 不同微服务的实例发生通讯时，其RPC调用过程中会产生的一系列复杂的、业务无关的处理逻辑，既是分布式处理；2) 为应对单一服务实例负载过重，或者异常导致的服务不可用，同一个微服务存在多个实例，在这多个实例中挑选一个以响应当前请求，这里涉及的一堆复杂处理逻辑，便对应可以认为是集群处理。本文所讨论的负载均衡应对的正是集群处理中的解决同一微服务多实例负载分配的问题，解决服务实例存在的单点故障问题。

负载均衡，令人容易本能地认为将任务均匀分配。实际上任务的分配的考虑因素可能是负载、性能吞吐量、响应时间、业务特性等，例如同一个服务可能在北京和深圳都有部署实例，其它依赖他们的微服务在负载均衡的作用下会就近路由到其中一个实例上。

技术发展日新月异，负载均衡方案也随之发生着变化，从传统服务端模式到现代微服务模式，其实现方式也经历了多次变革，在进一步了解Dubbo微服务的负载均衡前，我们先大体回顾下各种方案，主要是理解他们的大致实现原理。

## IMPORTANT

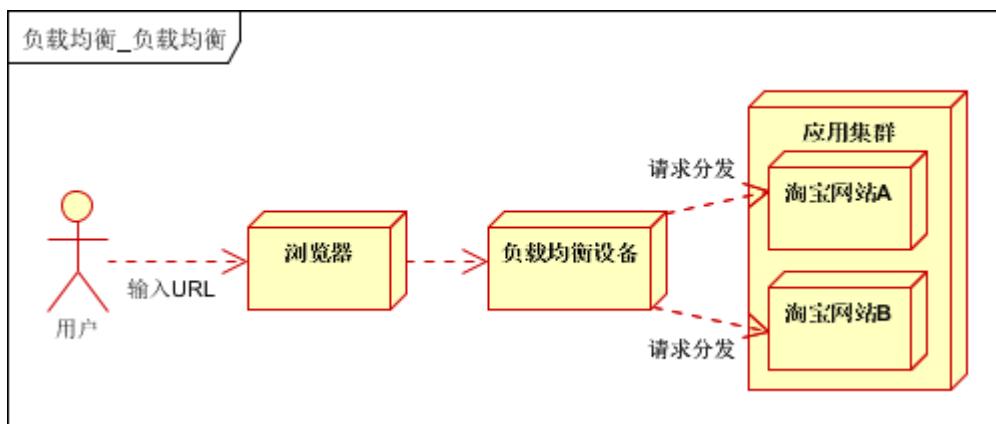
尽管多样，但由于始终和OSI网络通讯模型紧密相关，因而可以对应被划分在不同的层上。

下文提到的 NAT、DR、TUN 和 FULLNAT 这4种模式由于需要使用到端口信息，对应着OSI网络模型中的传输层，因而习惯上被统称为4层负载均衡，显然微服务中采用的 客户端 和 服务端 模式相应地被划分为7层负载均衡。

## 传统服务端模式

注：本章节主要参考大型网站架构系列：负载均衡详解 (<https://cloud.tencent.com/developer/article/1031624>)、吴佳明（普空）：LVS在大规模网络环境中的应用 ([http://blog.sina.com.cn/s/blog\\_620c47630102v2iz.html](http://blog.sina.com.cn/s/blog_620c47630102v2iz.html)) 和 LVS的原理-工作模式 (<https://yizhi.ren/2019/05/03/lvs>)。

负载均衡最常使用的场景是从多个服务器提供单个Internet服务。例如流行的网站、大型Internet中继聊天网络、高带宽文件传输站点、网络新闻服务器、域名系统服务器和数据库。传统负载均衡的经典模式如下图所示：



其中的负载均衡设备根据负载均衡策略将接入的用户请求分发到集群的某一台机器上，同一应用部署到多个机器上便组成了集群。

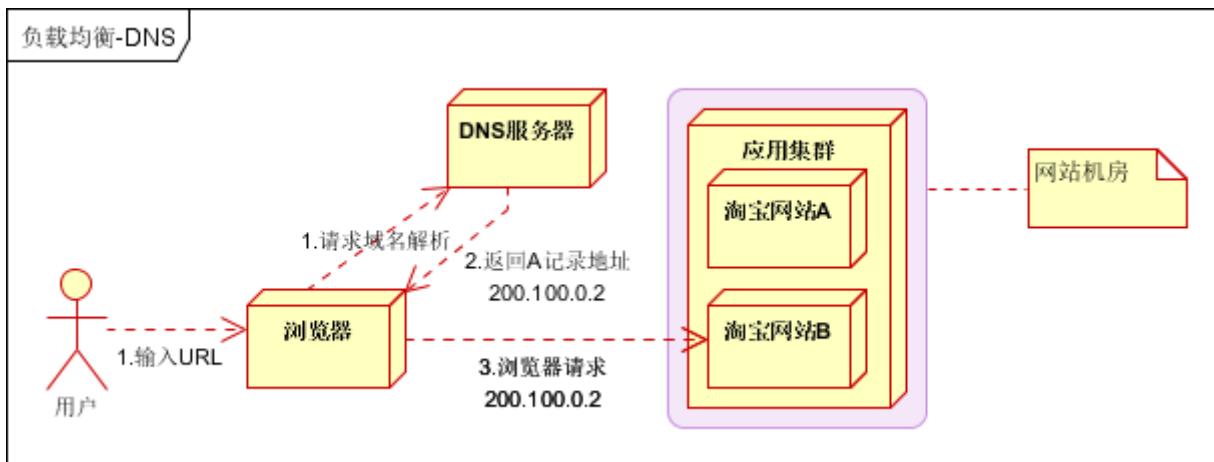


1. DS : *director server*, 即负载均衡器，根据一定的负载均衡算法将流量分发到后端的真实服务器上；
2. RS : *real server* 真实的提供服务的server, 可被DS划分到一个或多个负载均衡组；
3. VIP : VS的IP, client请求服务的DIP (*destination IP address*) , 定义在DS上, client或其网关需要有其路由；

- CIP & DIP & RIP：分别代表客户端client的IP、DS的IP、RS的IP；
- VIP 用于对外提供服务，采用的是公网IP，而 DIP 是负载均衡器的IP，虽和 VIP 属于同一主机，但一般是当前局域网IP。

## DNS负载均衡

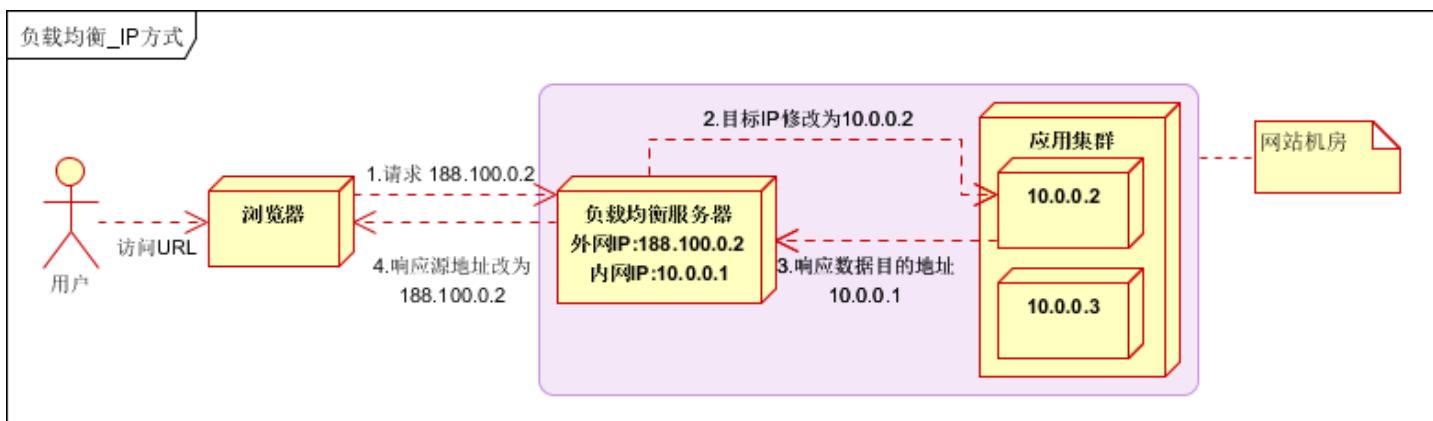
在DNS域名服务器中将集群中所有机器IP配置为指向同一域名，由DNS根据用户请求中内含特征将请求就近转发给其中一个实例。



该模式由于掌控权控制在域名服务商手里，扩展性受到限制。

## (NAT模式)IP负载均衡

用户请求接入后，DS 在系统内核中  $\langle \text{Src:CIP}, \text{Dst:VIP} \rangle$  根据负载均衡算法选用到的 RS 修改为  $\langle \text{Src:CIP}, \text{Dst:RIP} \rangle^{\text{DNAT}}$ ，封装成新的IP请求报文后发送给 RS，RS 处理完请求将响应返回给 DS，DS 将收到的IP响应报文  $\langle \text{Src:RIP}, \text{Dst:CIP} \rangle$  修改为  $\langle \text{Src:VIP}, \text{Dst:CIP} \rangle^{\text{SNAT}}$ ，最后返回给 client。



- 1) RS 的网关必须设置为 DS ;
- 2) 需要开启 IP-Forward 支持 DS 修改源IP并转发数据包；

## NOTE

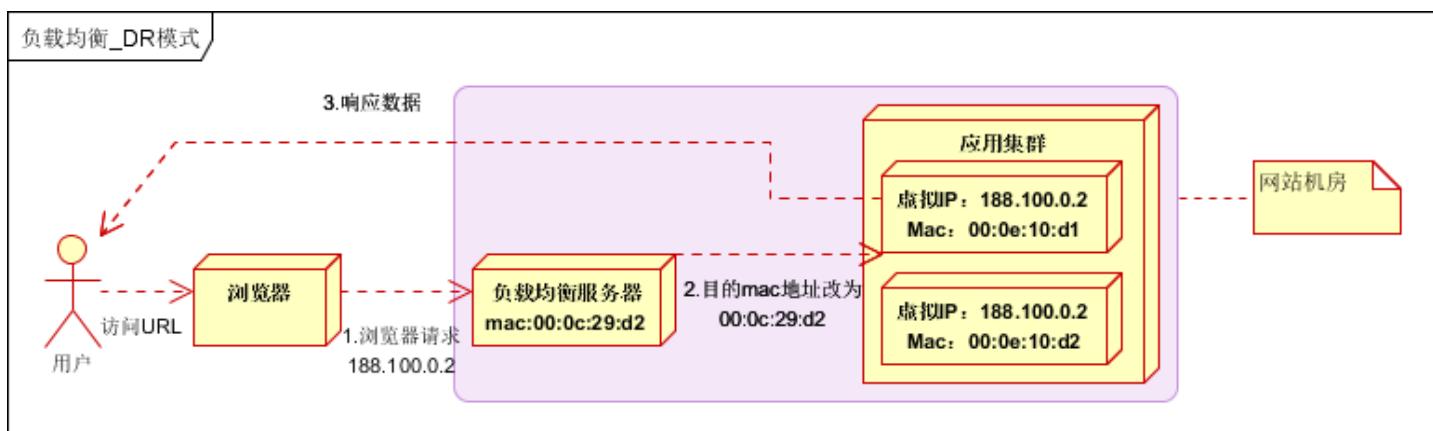
**IP-Forward** : 当主机拥有多于一块的网卡时，其中一块收到数据包，根据数据包的目的ip地址将数据包发往本机另一块网卡，该网卡根据路由表继续发送数据包。

这一模式还存在另外一种形式： DS 在接入请求后，将IP请求报文修改为 <Src:VIP,Dst:RIP>，也即对数据包的目标IP地址进行修改的同时，将数据包的源地址设置为自身，也即源地址转换。

缺点是集群的最大吞吐量受制于负载均衡服务器的带宽。

## (DR模式)链路层负载均衡

RS 集群和 DS 不作为网关出现处于同一物理网络中的，Client发送一个 <Src:CIP,Dst:VIP> IP请求报文，当 DS 接受请求后挑中某个 RS 作为目标机，并将其当前报文中的MAC地址设成 R\_MAC ，随后发送一个ARP广播出去，私有 loopback 环回接口绑定了VIP的所有 RS 接受到请求后，只有MAC地址为 R\_MAC 的 RS 服务器会响应请求。



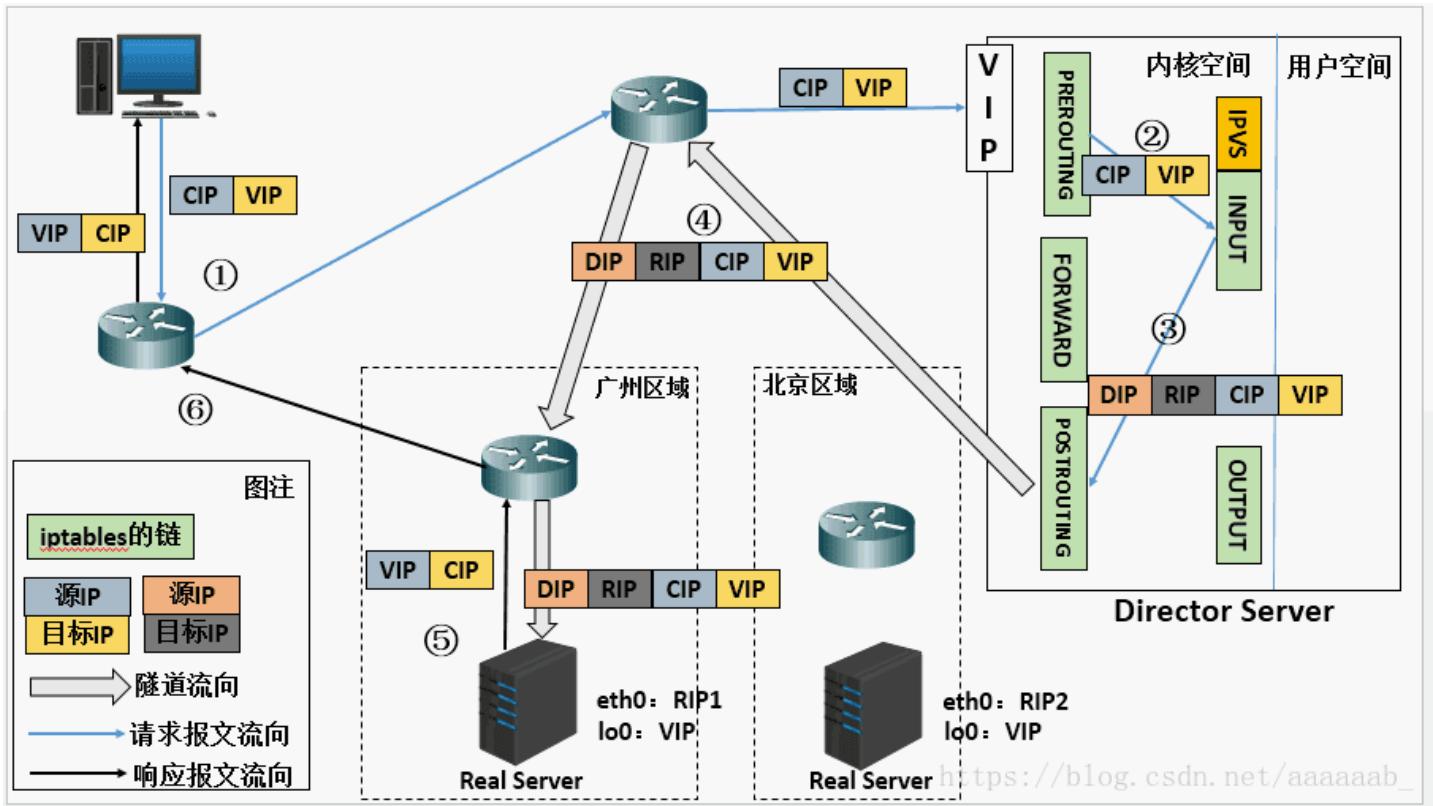
## NOTE

1. RS 集群中所有机器将 VIP 绑定在 loopback 环回接口上的原因有三：1) DS 在转发请求时并没有修改目标IP地址，而报文接受方需要同时匹配目标IP和目标MAC地址，否则丢包处理，因此 RS 主机需要绑定 VIP；2) 在同一个网段中是不能存在IP冲突的，即便是虚拟IP，但 loopback 环回接口绑定的IP地址是本机私有的，不在网络上共享，因而不会产生冲突；3) 当前物理网络中，只存在一个IP为 VIP 的主机，因此 client 使用 VIP 构建报文后能准确匹配到目标 DS 主机。
2. 在局域网中的所有机器都是通过广播的形式进行通讯的，其中一台机器发送的消息会被所有其他机器接受到，后者会将报文中的目标地址包括IP信息和MAC地址和自身的比对，如果一致就接受并做响应处理，否则略过。
3. 网络通讯发生的前提是知晓彼此MAC地址，当局域网中的客户端机器不知道目标机器的MAC地址时，可以在通讯正式发生之前将其设为12个F的广播地址，目标机器收到广播后会返回自身MAC地址信息，这一过程是受ARP协议支持的。DR负载均衡模式中，由于 DS 已经明确指明了目标机的MAC地址，目标IP地址VIP也和本机绑定在环回接口的私有IP一致，因而不会有ARP协议参与响应。

该模式中，DS 无需处理响应和 Ip-Forward 数据转发，因而它不会像IP负载均衡模式中那样成为瓶颈。

## (TUN模式)IP tunnel负载均衡

接收到报文后，DS 依然执行负载算法挑选出 RS；在报文转发给 RS 前会先对其进行IP封包形成IPIP包，报文内容为 [ $<\text{Src:DIP}, \text{Dst:RIP}>, <\text{Src:CIP}, \text{Dst:VIP}>, \dots$ ]，随后报文被发给 RS；RS 收到报文后，剥离掉IP隧道包头，发现还原得到IP报文 [ $<\text{Src:CIP}, \text{Dst:VIP}>, \dots$ ] 中的目标IP也即VIP和绑定在自身环回接口上的一致，因而接受报文并处理响应，最后使用该环回接口经实体网卡将数据发送给 client 由解IP包获得源IP地址CIP确定的客户机。



## NOTE

IP隧道：报文发送方在IP头的外部再包裹一个IP头，接收方先解析出第一个IP头，然后再按照正常流程处理剩下的的IP数据包。

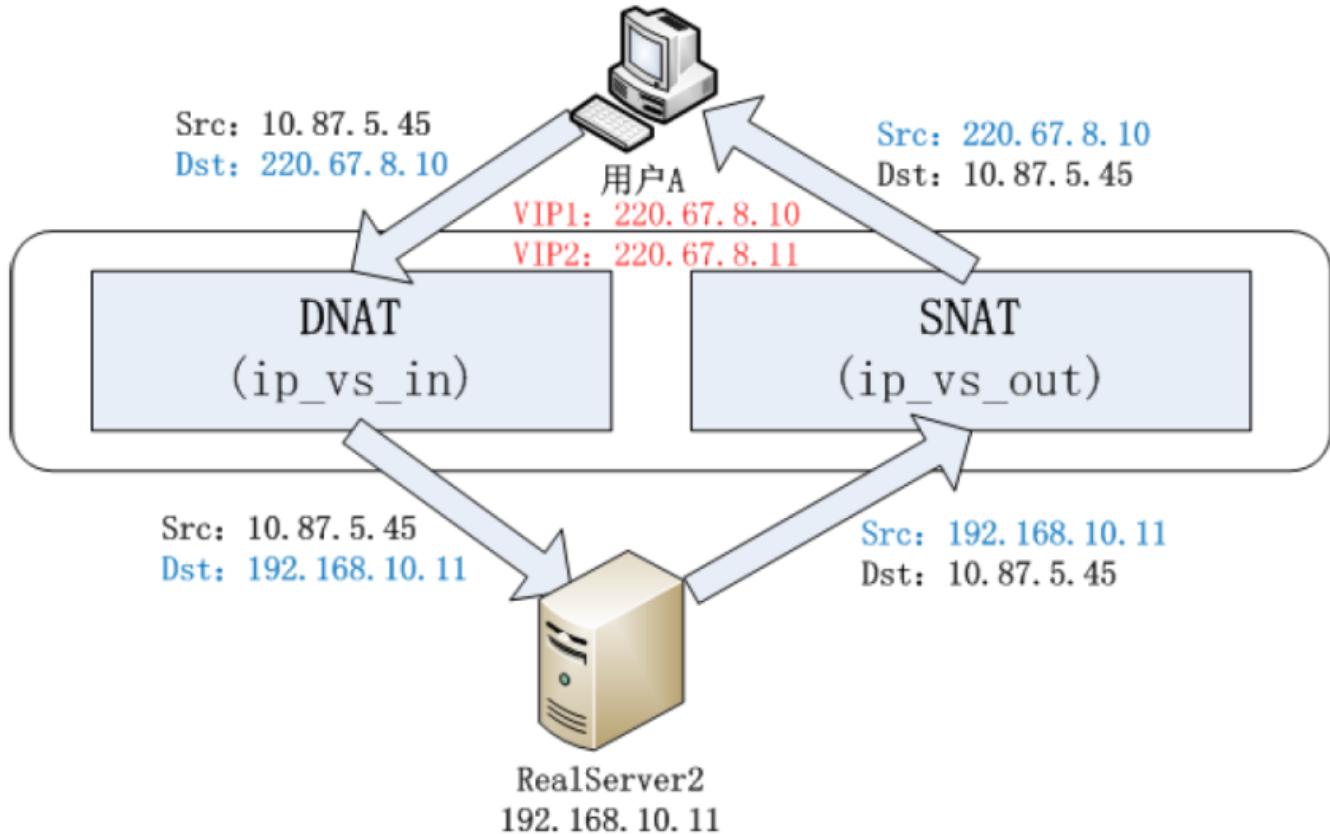
该模式中，每个数据包都要新增IP报头，如果收到的数据包已经达到以太网帧最大长度1.5K，会因为IP报头没法添入而引发异常。

## (FULLNAT模式)IP负载均衡

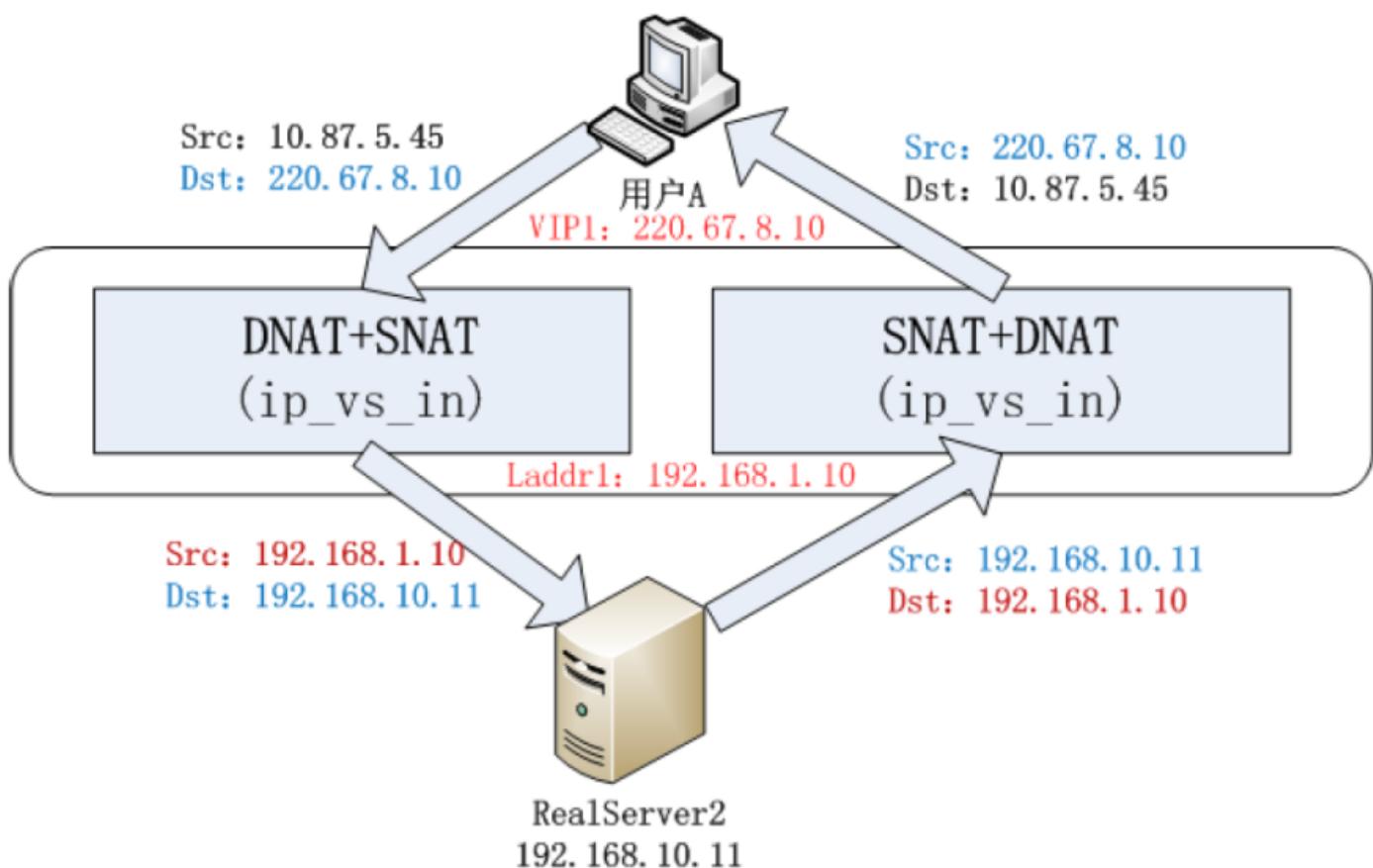
NAT、DR、TUN 这3种模式要么配置复杂，要么限定只能使用于同一局域网，因而淘宝在 NAT 模式的基础上提出了 FULLNAT 模式。

用户请求接入后，DS 在系统内核中  $\langle \text{Src:CIP}, \text{Dst:VIP} \rangle$  根据负载均衡算法选用到的 RS 修改为  $\langle \text{Src:DIP}, \text{Dst:RIP} \rangle$ ，封装成新的IP请求报文后发送给 RS，RS 处理完请求将响应返回给 DS，DS 将收到的IP响应报文  $\langle \text{Src:RIP}, \text{Dst:DIP} \rangle$  修改为  $\langle \text{Src:VIP}, \text{Dst:CIP} \rangle$ ，最后返回给 client。

不同于 NAT 模式的是，NAT 模式中 DS 对入站报文和出站报文分别做 DNAT 和 SNAT 处理，如下图：



而FULLNAT模式对入出站报文均做 $\text{FULLNAT}_{\text{DNAT+SNAT}}$ 处理，在公网VIP和局域网DIP间转换，如下图：



FULLNAT模式 的问题是：RS无法获得CIP，淘宝提出TOA概念，主要原理是“将 client address 放到了TCP Option里面带给后端RS，RS收到后保存在socket的结构体里并通过toa内核模块hook了getname函数，这样当用户调用getname获取远端地址时，返回的是保存在socket的TCPOption的IP。”

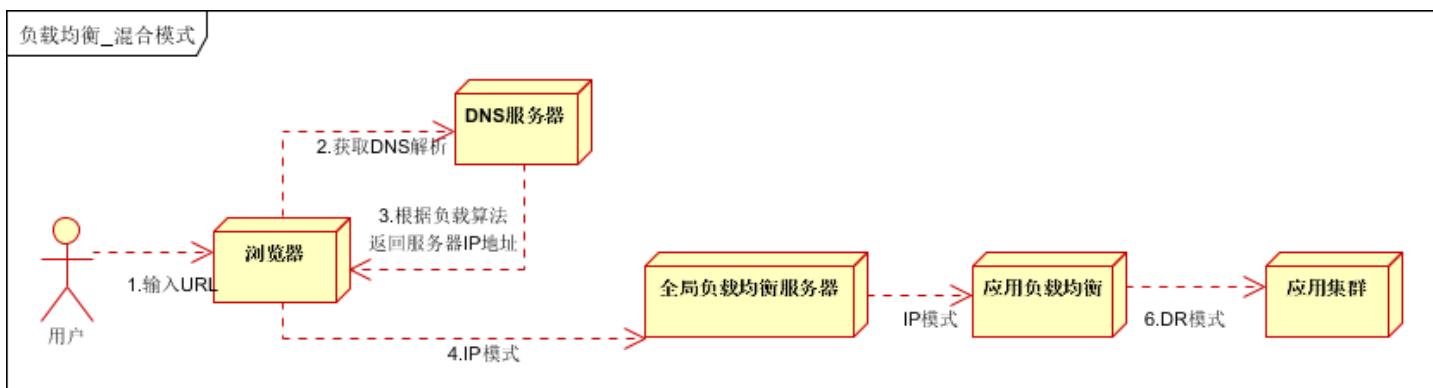
FULLNAT模式 主要的思想是把网关和其下机器的通信，改为了普通的网络通信，从而解决了跨局域网通讯的问题，大大提高了运维部署的便利性。

RS 的增减可能会影响到客户端连接不能固定落到某同一个 RS 主机上，要求DS使用一致性算法来调度客户端的连接。

## 混合型负载均衡

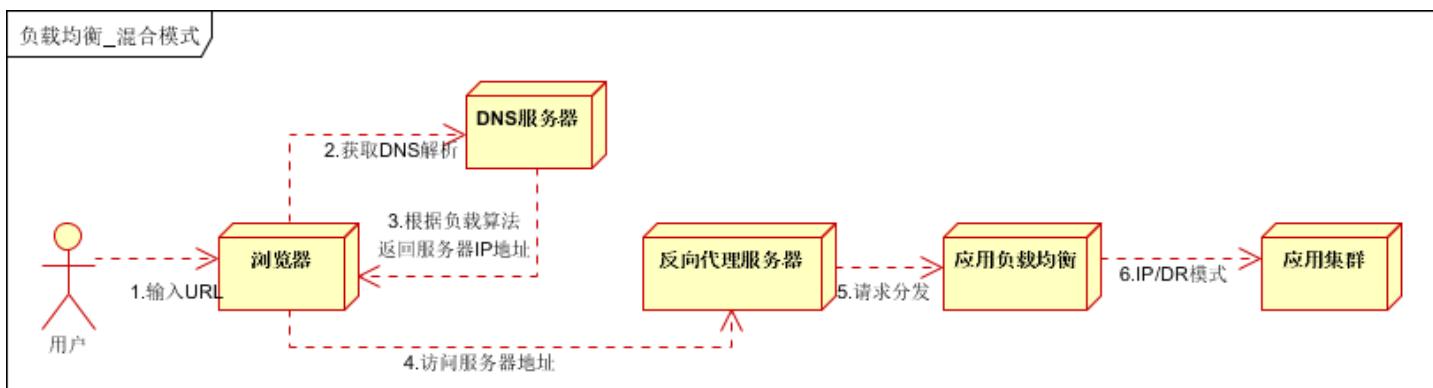
### 多级全局式

由于多个服务器群内硬件设备、各自的规模、提供的服务等的差异，可以考虑给每个服务器群采用最合适的负载均衡方式，然后再将这多个服务器集群组成一个规模更大的服务器集群，在前面再增加一个全局负载均衡服务器，以一个整体向外界提供服务，从而达到最佳的性能。



### 动静分离式

流媒体特别发达的今天，类似新闻、图片社交类的网站，其富文本交互会涉及到大量的静态资源，它们可能占比不算特别大，但访问很频繁，变化却不那么频繁。于提供这类服务的服务器集群而言，可以采用动静分离模式，在负载均衡服务器前再增设一台反向代理服务器，将静态资源交由它处理，该模式下的反向代理服务器可以起到缓存和动态请求分发的作用，当静态资源已有缓存时，可直接返回。



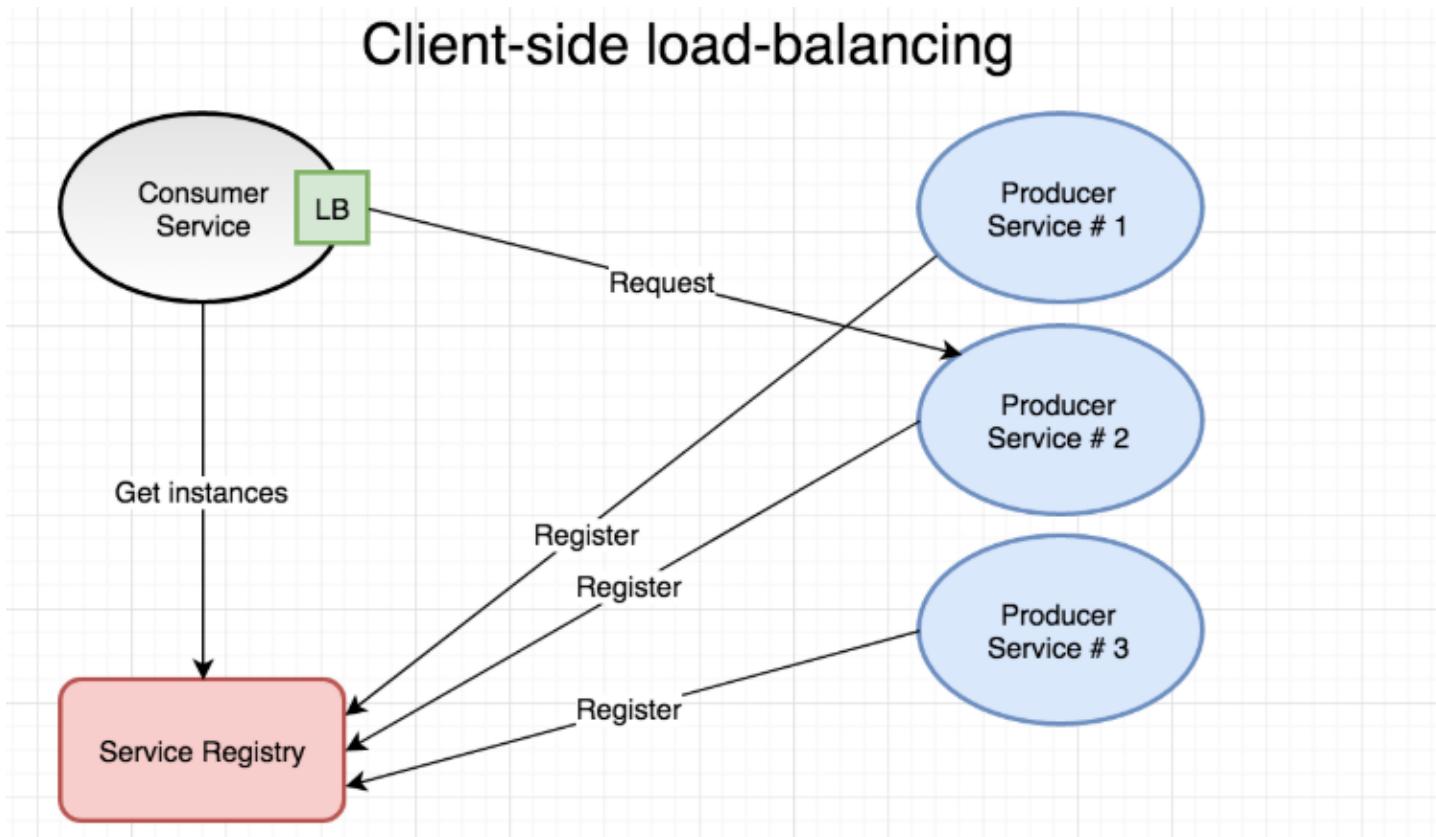
# 现代微服务模式

微服务开发中，传统服务模式使用的负载均衡模式基本不再适用，即便使用同一原理，也得对应重新适配。

## 客户端模式

微服务架构模式中，离不开服务注册中心的支持，也正是因为有了它的加持，大多数微服务框架会采用基于客户端模式的负载均衡实现。基本原理是：

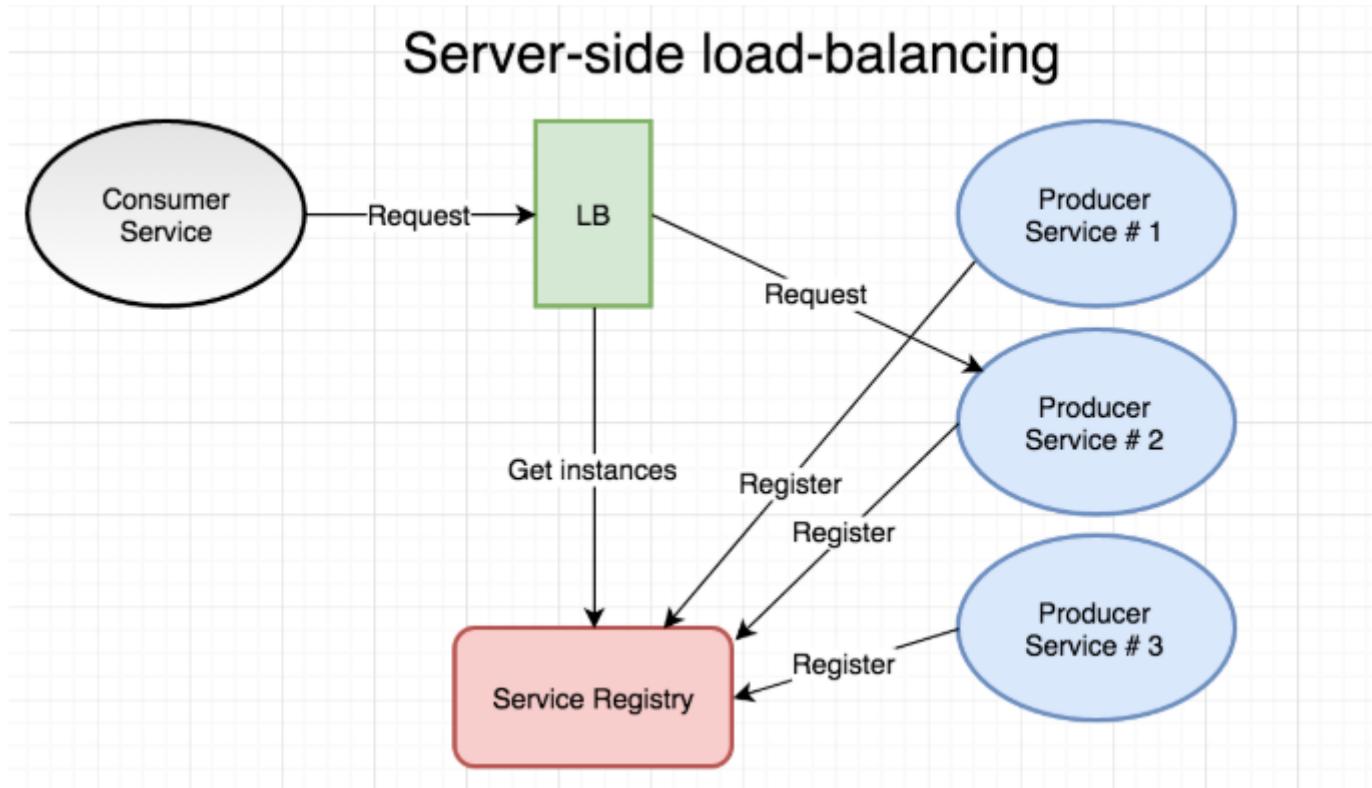
1. 服务提供者Provider的实例启动时会将其服务名称、地址等基本信息上报到注册中心Registry，形成该Provider的一个服务实例列表；
2. 在服务消费者Consumer提供服务逻辑名通过Registry对Provider进行引用时，Registry会返回一个含有1到多个其引用实例的列表RefList；
3. Consumer将RefList缓存在本地，当发起RPC调用时，会采用负载均衡算法从RefList中筛选一个Provider的服务实例已经获取到用于通讯的地址信息(IP, 主机名, 端口号)作为最终被调用对象；
4. Provider和Consumer同Registry始终保持着长连接，当Provider的某个服务实例宕机或下线时，该实例对应会被从Registry移除，同时Consumer也第一时间感知到这种变化，对应更新本地针对Provider的RefList。



这种模式的优点是没有负载均衡服务器存在所产生的中心瓶颈，缺点是具有较高的内部复杂性，也会加重网络流量负载。

## 服务端模式

实际上上述所有提到的所有传统服务端模式的负载均衡方案都可以概括为服务端模式，而本章所述的服务端模式是在微服务开发下使用的负载均衡方案，实现方式非常不同。如下图所示，同上述客户端模式对比，服务消费者Consumer和服务提供者Provider中间横亘着一个依赖于注册中心Registry的负载均衡器LB。



此种模式下，在Consumer提供Provider的服务逻辑名发起RPC调用时，会先经LB先从Registry查询获得Provider的服务引用实例列表RefList，再由LB使用负载均衡算法从RefList中筛选出一个引用实例，最后才能最终完成RPC调用。由于LB所处的位置让它同时具备了反向代理的作用，因而可以内置包括服务发现机制、SSL认证等功能，而这正是承载着实现微服务边车模式的容器的职责所在，也就是说负载均衡器通常是内置在部署微服务的容器中的。

## 负载均衡算法

服务集群中已知存在若干可用 RS，负载均衡器接入请求后，会负责选中一台 RS 主机处理请求，选中的这个过程涉及的正是负责均衡调度算法。具体采用何种算法则需要考量整个集群的特性，尽最大可能地提升整个集群的处理能力。

常见的负载均衡算法有 轮询、随机、最少链接、源地址散列、加权，该章节介绍部分来自[大型网站架构系列：负载均衡详解（2）](#) (<https://www.cnblogs.com/itfly8/p/5043452.html>)。

### 轮询

“ 将所有请求，依次分发到每台服务器上，适合服务器硬件同相同的场景。

优点：服务器请求数目相同；

缺点：服务器压力不一样，不适合服务器配置不同的情况；

## 随机

“ 请求随机分配到各个服务器。

优点：使用简单；

缺点：不适合机器配置不同的场景；

## 最少链接

“ 将请求分配到连接数最少的服务器（目前处理请求最少的服务器）。

优点：根据服务器当前的请求处理情况，动态分配；

缺点：算法实现相对复杂，需要监控服务器请求连接数；

## 源地址散列

“ 根据IP地址进行Hash计算，得到IP地址。

优点：将来自同一IP地址的请求，同一会话期内，转发到相同的服务器；实现会话粘滞。

缺点：目标服务器宕机后，会话会丢失；

## 加权

“ 在轮询，随机，最少链接，Hash'等算法的基础上，通过加权的方式，进行负载服务器分配。

优点：根据权重，调节转发服务器的请求数目；

缺点：使用相对复杂；

## Dubbo负载均衡实现

Dubbo中的负载均衡采取7层客户端模式，实现的算法有 轮询、随机、最少链接、源地址散列 这4种。

## 总体方案

Dubbo微服务中的负载目的是从同一微服务的多个实例中挑选一个微服务引用实例Invoker执行当前RPC方法。Invoker是RPC方法调用的执行体，仅包含了服务级别的配置数据，要实现负载均衡中的粘滞还得有更多方法级别的数据——Invocation类型的入参。Dubbo的负载均衡是一个扩展点，定义及生成代码如下：

```
@SPI(RandomLoadBalance.NAME) JAVA
public interface LoadBalance {

    @Adaptive("loadbalance")
    <T> Invoker<T> select(List<Invoker<T>> invokers, URL referUrl, Invocation
    invocation) throws RpcException;

}

//=====
//对应生成代码
//=====

public class LoadBalance$Adaptive implements LoadBalance {
    public Invoker select(List<Invoker> invokers, URL referUrl, Invocation invocation) throws
    RpcException {
        if (referUrl == null) throw new IllegalArgumentException("url == null");
        URL url = referUrl;
        if (invocation == null) throw new IllegalArgumentException("invocation ==
null");
        String methodName = invocation.getMethodName();
        String extName = url.getParameter(methodName, "loadbalance", "random");
        if (extName == null)
            throw new IllegalStateException("Failed to get extension (LoadBalance) name
from url (" + url.toString() + ") use keys([loadbalance])");
        LoadBalance extension = (LoadBalance) ExtensionLoader.getExtensionLoader(
            LoadBalance.class).getExtension(extName);
        return extension.select(invokers, referUrl, invocation);
    }
}
```

上述生成代码中，说明引用服务时传入的配置总参 URL referUrl 中如果含有配置项 methodName + ". " + "loadbalance"，则采用名称与其值所对应的负载均衡算法扩展点具类，否则则使用随机算法由 "random" 指定。

Dubbo也特地对于无需执行负载均衡算法的场景做了统一处理，定义了如下抽象类，子类相应实现 doSelect() 抽象方法即可：

```
public abstract class AbstractLoadBalance implements LoadBalance {
```

```
    ...  
    @Override  
    public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) {  
        if (CollectionUtils.isEmpty(invokers)) {  
            return null;  
        }  
        if (invokers.size() == 1) {  
            return invokers.get(0);  
        }  
        return doSelect(invokers, url, invocation);  
    }  
  
    protected abstract <T> Invoker<T> doSelect(  
        List<Invoker<T>> invokers, URL url, Invocation invocation);  
}
```

## 算法实现

JVM是按需延迟执行类加载的，虚拟机成功启动后，需要经过一段预热时间，性能才能达到最优。于Dubbo微服务而言也是同一个道理，同一微服务的不同实例，其导出时机并不相同，已经过了预热阶段的实例可以承担更多的流量，而还在预热阶段的实例正处于性能爬升时期，应慢慢释放其流量分摊能力，因此抽象父类 `AbstractLoadBalance` 中还专门为此定义了一个用于根据预热和启动时间计算出某个实例应分摊流量的权重。

### NOTE

已过预热阶段的所有实例的默认权重为100，可以根据实例所处环境I/O、CPU等能力因素单独配置其权重。

预热阶段流量分摊：

- 权重计算公式： 配置权重 × (实例启动耗时 ÷ 实例预热配时)
- 权重取值范围： [1 , 配置权重]

所涉计算因子：

- 配置权重： `invoker.url["methodName + "." + "weight"] | 100`
- 实例导出时间： `exportTime = invoker.url["timestamp"]`
- 实例启动耗时： `System.currentTimeMillis() - exportTime`
- 实例预热配时： `invoker.url["warmup"] | 10 * 60 * 1000`

流量权重计算实现源码如下：

```

public abstract class AbstractLoadBalance implements LoadBalance {
    ...
    static int calculateWarmupWeight(int uptime, int warmup, int weight) {
        int ww = (int) (uptime / ((float) warmup / weight));
        return ww < 1 ? 1 : (Math.min(ww, weight));
    }
    int getWeight(Invoker<?> invoker, Invocation invocation) {
        int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(),
WEIGHT_KEY, DEFAULT_WEIGHT);
        if (weight > 0) {
            long timestamp = invoker.getUrl().getParameter(TIMESTAMP_KEY, 0L);
            if (timestamp > 0L) {
                long uptime = System.currentTimeMillis() - timestamp;
                if (uptime < 0) {
                    return 1;
                }
                int warmup = invoker.getUrl().getParameter(WARMUP_KEY, DEFAULT_WARMUP);
                if (uptime > 0 && uptime < warmup) {
                    weight = calculateWarmupWeight((int)uptime, warmup, weight);
                }
            }
        }
        return Math.max(weight, 0);
    }
}

```

## 随机 加权随机

熟悉概率学的话就比较清楚，在指定区间产生随机数，样本足够大的话，总体而言随机数分布会比较均匀。当所有实例的权重都一样，可以认为由其组成的整数区间中所有元素装入到列表，列表索引依顺序构成的整数序列，每一个元素参与均分获得一个单位的子区间，因而直接用元素总数取随机数获得索引位置是成立的，也即可以使用 `ThreadLocalRandom.current().nextInt(number of invokers)` 可以获得当前随机数的所对应的实例。

权重不等的情况下，可以想象成总区间被放大了，对应产生衍生列表，比如

`arr1[A:2,B:3,C:1]≈arr1'[A,A',B,B',B',C]`，权重取总，在以总值取随机数，不难理解B被取到的概率更高。也就是说可以先使用 `w1 + w2 + ... + wn` 获得权重总和得到 `total`，再依其取随机数——`rd = nextInt(total)`，最后便可以依据 `rd` 来获取到当前随机取到的实例。

接下来如何判断 `rd` 对应的是列表中的哪个索引值呢？`rd` 是在被放大的区间中生成的，以它为索引值找到它对应位置的元素若干相同元素由同一个衍生，由该元素或为其衍生元素，比如 `B'` 在原列表中找到的对应位置即为想要取得的索引值。当然，这只是便于理解的一种想象，继续深入观察下衍生列表 `arr'`，假设 `rd=4`，该位置的元素为第二个 `B'`，而它前面有两个 `A A'` 由 `A` 衍生，还有两个 `B`，这时会发现都列表的元素挨个迭代，取权重累加，直到累加值大于 `rd` 为止，此时所在迭代元素即为想要随机取到的实例。转为表达式也即 `w1 + w2 + ... + wx > rd`，它等价于 `rd - w1 - w2 - ... - wx < 0`，其中 `wx` 是首个表达式的元素。这时理解下述实现源码就不难了，如下：

```
public class RandomLoadBalance extends AbstractLoadBalance {
```

JAVA

```
    public static final String NAME = "random";

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        int length = invokers.size();
        boolean sameWeight = true;
        int[] weights = new int[length];
        int firstWeight = getWeight(invokers.get(0), invocation);
        weights[0] = firstWeight;
        int totalWeight = firstWeight;
        for (int i = 1; i < length; i++) {
            int weight = getWeight(invokers.get(i), invocation);
            weights[i] = weight;
            totalWeight += weight;

            //所有后续的权重和首个权重对比，首个不等就能确认整个列表权重不等
            if (sameWeight && weight != firstWeight) {
                sameWeight = false;
            }
        }

        if (totalWeight > 0 && !sameWeight) {
            //在扩展区间取随机数
            int offset = ThreadLocalRandom.current().nextInt(totalWeight);
            for (int i = 0; i < length; i++) {
                //挨个迭代元素，直到offset - weights[i] < 0
                offset -= weights[i];
                if (offset < 0) {
                    return invokers.get(i);
                }
            }
        }
        // If all invokers have the same weight value or totalWeight=0, return evenly.
        return invokers.get(ThreadLocalRandom.current().nextInt(length));
    }

}
```

**随机数：**随机数的产生取决于种子和算法，这意味着相同算法的情况下，种子的强度决定了随机数的随机性：

- Random的种子是System.currentTimeMillis()，所以它的随机数在理论上和实际中都是线性可预测的；
- SecureRandom比较适合安全性要求高的场景，它会例如如键盘输入时间、CPU时钟、内存使用状态、硬盘空闲空间、IO延时进程数量、线程数量等信息来得到一个近似随机的种子，强度高以致实际中几乎没法预测。

## NOTE

Random是基于CAS实现线程安全的，因此不适合在高并发的多线程容易产生资源争用的环境下使用，而ThreadLocalRandom是线程本地，内部实现也没有使用CAS，因而效率高。

生成新的随机数需要首先根据老的种子生成新的种子，然后使用新的种子来计算得到新的随机数，因此在并发情况下，Random的种子容易成为被争用资源而导致大量线程自旋重试而效率低下。

RPC方法调用时一个高并发场景，因此Dubbo将Random改为了ThreadLocalRandom。

## 轮询 平滑加权轮询

轮询算法中，Dubbo选用了平滑加权版，一能根据节点的性能选择分摊多少流量，二能确保避免普通加权轮询算法导致的持续将流量分配给权重更大的节点，让各节点按自己的能力较为均匀地参与服务。

## 算法解析

该算法最初来源于nginx，分如下两步：

1. 为每个节点加上它的权重值；
2. 选择最大的节点 selected 减去总的权重值；
3. 返回选中节点 selected；

算法本身看起来很简单，但是没法比较直观地加以理解，大致意思是“每次选用节点，所有节点都有机会累加自己的权重，然而当前权重最大的那个会以权重总值快速衰减下去，这样使得初始权重小者有机会超过初始权重大者进而得以被选中”。

以{a:5, b:1, c:2}三个节点举例，其选择过程如下表：

轮数	选择前的当前权重	选择节点	选择后的当前权重
1	{5,1,2}	a	{-3,1,2}
2	{2,2,4}	c	{2,2,-4}
3	{7,3,-2}	a	{-1,3,-2}
4	{4,4,0}	a	{-4,4,0}
5	{1,5,2}	b	{1,-3,2}
6	{6,-2,4}	a	{-2,-2,4}
7	{3,-1,6}	c	{3,-1,-2}
8	{8,0,0}	a	{0,0,0}

上表表示，a、b、c选择的次数符合5:1:2，而且权重大的不会被连续被选。8轮选择后，当前值又回到{0, 0, 0}，以上序列一直循环，始终是平滑的，有关算法的证明请参考[nginx平滑的基于权重轮询算法分析](https://tenfy.cn/2018/11/12/smooth-weighted-round-robin/) (<https://tenfy.cn/2018/11/12/smooth-weighted-round-robin/>)。

## 代码实现

不同于 加权随机， 加权轮询 算法需要记住RPC方法被一个服务实例调用的历史记录，轮流地让各个实例为同一个RPC方法提供服务，权重大者获得更多次调用机会。LoadBalance 是一个扩展点，和其它扩展点一样，其实现具类都是单例的。因而就当前应用而言，它是针对所有引用服务的，每一个都会有一到多个实例，加上RPC方法被并发调用的特性等都确定了需要两级的并发类型的Map容器缓存引用服务的实例，如下：

```
ConcurrentMap<String, ConcurrentMap<String, WeightedRoundRobin>>
methodWeightMap = new ConcurrentHashMap<>();
```

JAVA

显然上述涉及了两级Key，而能唯一确认二级Key的部分是host[":" + port]，如下：

1. 一级Key: group + "/" + interfaceName + ":" + version + "." + invocation[methodName]
2. 二级Key: [protocol + "://" + [username[":" + password] + "@"]][host[":" + port]]["/" + path]

二级Map容器中装入的节点代表了一个服务引用实例，而定义的methodWeightMap缓存中的二级map装入的值类型为WeightedRoundRobin，定义如下源码：

```
protected static class WeightedRoundRobin {  
    private int weight;  
    private AtomicLong current = new AtomicLong(0);  
    private long lastUpdate;  
    public int getWeight() {  
        return weight;  
    }  
    public void setWeight(int weight) {  
        this.weight = weight;  
        current.set(0);  
    }  
    public long increaseCurrent() {  
        return current.addAndGet(weight);  
    }  
    public void sel(int total) {  
        current.addAndGet(-1 * total);  
    }  
    public long getLastUpdate() {  
        return lastUpdate;  
    }  
    public void setLastUpdate(long lastUpdate) {  
        this.lastUpdate = lastUpdate;  
    }  
}
```

上述源码表示，WeightedRoundRobin除了记录对应微服务引用实例配置权重，还定义了处理平滑算法的自身权值行为，并且使用了原子变量保证其多线程环境下的计算安全。

微服务始终处于分布式环境下，服务实例随时都有可能上下线，于Dubbo负载均衡调用而言，就是对同一个服务的调用，入参invokers列表中的元素比先前增加或减少了，新增的可以及时感知到，而减少的则需要借助超时机制在加锁环境下移除掉过期不在线的服务引用实例，这也是WeightedRoundRobin定义lastUpdate属性及其setter和getter的原因，如下代码所示：

JAVA

```

public class RoundRobinLoadBalance extends AbstractLoadBalance {
    private static final int RECYCLE_PERIOD = 60000;

    private AtomicBoolean updateLock = new AtomicBoolean();

    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        //获取二级Map容器, Key: `group + "/" + interfaceName + ":" + version + "." +
        invocation[methodName]`
        String key = invokers.get(0).getUrl().getServiceKey() + "." +
        invocation.getMethodName();
        ConcurrentHashMap<String, WeightedRoundRobin> map = methodWeightMap.get(key);
        if (map == null) {
            methodWeightMap.putIfAbsent(key, new ConcurrentHashMap<String,
WeightedRoundRobin>());
            map = methodWeightMap.get(key);
        }
        ...//算法实现
        //失效引用实例处理, 在最后返回结果时执行
        if (!updateLock.get() && invokers.size() != map.size()) {
            if (updateLock.compareAndSet(false, true)) {
                try {
                    // copy -> modify -> update reference
                    ConcurrentHashMap<String, WeightedRoundRobin> newMap = new
ConcurrentHashMap<>(map);
                    newMap.entrySet().removeIf(item -> now -
item.getValue().getLastUpdate() > RECYCLE_PERIOD);
                    methodWeightMap.put(key, newMap);
                } finally {
                    updateLock.set(false);
                }
            }
        }
        ...
    }
}

```

仔细研读上述代码，有3点值得一提的：

1. 执行 `methodWeightMap` 缓存更新时的其中一个条件是入参 `invokers` 列表元素个数和服务引用实例属于执行当前RPC方法调用的微服务二级缓存的要一致，因为新增的从 `invokers` 入参可以及时发现，因而那种列表中同时出现新增或减少的也会导致 `invokers.size() != map.size()` 条件成立。
2. `RoundRobinLoadBalance` 使用了类型的原子变量 `AtomicBoolean` 作为锁，目的是为了减少锁的自旋带来的性能损耗，`compareAndSet()` 尝试一次执行不成功，便直接略过，说明其它线程已经获得了该锁正在执行目标逻辑，于当前微服务的实例缓存而言，当前时刻该逻辑只需执行一次，无论哪个线程获得机会执行都可以。然而 `updateLock` 这个锁于 `RoundRobinLoadBalance` 单例而言它是全局的，每个服务都对应一个装其引用实例的 `ConcurrentMap<String,`

`WeightedRoundRobin`> 容器，`updateLock` 作用在所有这些二级容器上，如果另有一个高频 RPC 调用存在，其所属服务实例频繁上下线，那么当前逻辑可能长时间没法及时获得机会执行，不过这种场景不常见，概率较低，即时发生也只是多浪费点缓存空间而已。

3. 尽管 `ConcurrentMap<String, WeightedRoundRobin>` 是线程安全的，但代码中还是采用了 `copy → modify → update reference` 这种更新模式，这样做的好处将加锁块的逻辑限定在当前块，可以做到容器正在更新的过程对外不可见。

最后剩下的是算法实现部分，算法有关原理请参考上一子章节，代码如下：

JAVA

```

protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    ...//获取二级Map容器
    int totalWeight = 0;
    long maxCurrent = Long.MIN_VALUE;
    long now = System.currentTimeMillis();
    Invoker<T> selectedInvoker = null;
    WeightedRoundRobin selectedWRR = null;
    for (Invoker<T> invoker : invokers) {
        //[[protocol + ":" + [username[:] + password] + "@"][host[:] + port]]["/" +
path]
        String identifyString = invoker.getUrl().toIdentityString();
        WeightedRoundRobin weightedRoundRobin = map.get(identifyString);
        int weight = getWeight(invoker, invocation);

        if (weightedRoundRobin == null) {
            weightedRoundRobin = new WeightedRoundRobin();
            weightedRoundRobin.setWeight(weight);
            map.putIfAbsent(identifyString, weightedRoundRobin);
        }
        //在预热阶段权重可能发生变化，需要更新
        if (weight != weightedRoundRobin.getWeight()) {
            //weight changed
            weightedRoundRobin.setWeight(weight);
        }
        long cur = weightedRoundRobin.increaseCurrent();
        weightedRoundRobin.setLastUpdate(now);
        if (cur > maxCurrent) {
            maxCurrent = cur;
            selectedInvoker = invoker;
            selectedWRR = weightedRoundRobin;
        }
        totalWeight += weight;
    }

    ....//失效引用实例处理

    if (selectedInvoker != null) {
        selectedWRR.sel(totalWeight);
        return selectedInvoker;
    }
    // should not happen here
    return invokers.get(0);
}

```

## 最少链接 最少活跃调用数

一个微服务的若干实例都在提供服务时，于接入客户端而言，一个实例，发往其中的并发请求越少，意味着被快速得到处理的可能性就越高。Dubbo微服务在注册中心的支持下，客户端持有连入微服务实例的更多相关信息，几乎是实时同步的，因而又进一步促成了使用客户端负载均衡模式的Dubbo将最小活跃数策略作为默认实现之一。

显然，**最小活跃数** 或 **最少活跃调用数** 是服务实例的一种实时状态，需要实时跟踪记录，为支持这一特性，Dubbo在协议层中已经基于拦截链机制提供了Filter实现，具体参考《Dubbo RPC 之 Protocol 协议层（三）》一文中**并发量控制**这一章节。需要的状态值记录在 `ConcurrentMap<String, ConcurrentMap<String, RpcStatus>>` 类型的 `METHOD_STATISTICS` 容器中，`RpcStatus` 是用于汇总一个RPC方法在一个特定服务实例的历史处理情况的。

源码实现总体上来说比较简单，大致逻辑是：1) 每次接入一个RPC请求后，依次遍历给定的对应微服务的所有服务实例，在 `RpcStatus.getStatus(referUrl, methodName).getActive()` 的支持下，对比计算得出所有满足 **最少活跃调用数** 要求的实例；2) 如果满足要求的个数为1，直接返回这个实例；否则，采用上述提到的 **加权随机** 挑选一个。

```
public class LeastActiveLoadBalance extends AbstractLoadBalance {
```

JAVA

```
    public static final String NAME = "leastactive";

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        int length = invokers.size();
        int leastActive = -1;
        int leastCount = 0;
        int[] leastIndexes = new int[length];
        int[] weights = new int[length];
        int totalWeight = 0;
        int firstWeight = 0;
        boolean sameWeight = true;

        for (int i = 0; i < length; i++) {
            Invoker<T> invoker = invokers.get(i);
            int active = RpcStatus.getStatus(invoker.getUrl(),
                invocation.getMethodName()).getActive();
            //计算加权值，在预热阶段会变化
            int afterWarmup = getWeight(invoker, invocation);
            weights[i] = afterWarmup;
            if (leastActive == -1 || active < leastActive) { //①
                leastActive = active;
                leastCount = 1;
                leastIndexes[0] = i;
                totalWeight = afterWarmup;
                firstWeight = afterWarmup;
                sameWeight = true;
            } else if (active == leastActive) { //②
                leastIndexes[leastCount++] = i;
                totalWeight += afterWarmup;
                if (sameWeight && i > 0
                    && afterWarmup != firstWeight) {
                    sameWeight = false;
                }
            }
        }
        if (leastCount == 1) {
            return invokers.get(leastIndexes[0]);
        }
        if (!sameWeight && totalWeight > 0) {
            int offsetWeight = ThreadLocalRandom.current().nextInt(totalWeight);
            for (int i = 0; i < leastCount; i++) {
                int leastIndex = leastIndexes[i];
                offsetWeight -= weights[leastIndex];
                if (offsetWeight < 0) {
                    return invokers.get(leastIndex);
                }
            }
        }
        return
    }
    invokers.get(leastIndexes[ThreadLocalRandom.current().nextInt(leastCount)]);
}
```

```
    }  
}
```

上述源码如下几处有亮点的地方，它们都利用临时空间记录了中间过程值，增加了效率：

1. 在循环迭代的过程中，始终关注最终状态值代码块②，一旦被破坏，则重新开始累计代码块①；
2. `totalWeight`、`firstWeight`、`sameWeight` 的处理；

## 源地址散列 一致性hash算法

分布式盛行前，人们习惯使用大型机运行他们的应用，对外提供服务。此时的服务器几乎承载着所有跟应用相关的后台进程，顶多搭配一些负载均衡策略将几套应用打包对外提供用户界面。大型机造价不菲，一般规模的公司几乎没法承担，就连大型互联公司也为之头疼，阿里在2009年就已经开启了去IOE的征程，如今已经成绩斐然，被业界纷纷效仿。

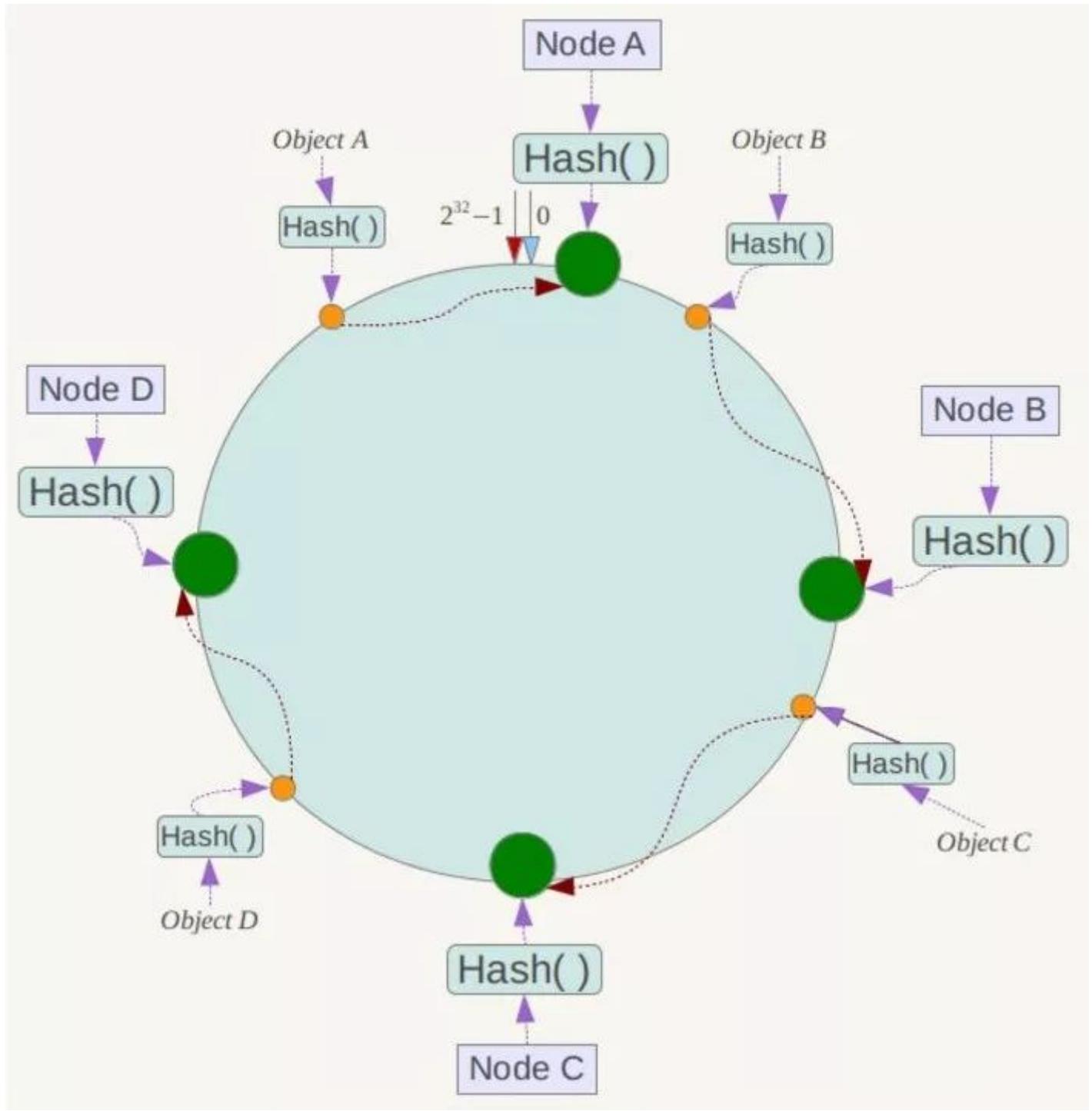
分布式已盛行的时代，开发者们习惯于将多台普通机器组合一起，采用一些冗余、分工、容错等技术手段将应用打散成更小的单元，然后整体向外提供服务。例如一台普通运行着MySQL的机器能承载的数据规模是千万级别，但应用面向的是十亿级别的，这时就会利用分工机制，让若干个这样部署在不同机器上的MySQL实例共同协作，以满足需求（实际生产环境中，往往是超过80%的数据集中在低于20%的数据表中，一般同时采用分表分库方案解决）；使用普通的设备虽然大幅降低了资金的耗费，但也往往意味着更高概率的单机故障，因此前面这个案例中人们还往往为每个MySQL实例提供它专属的备机，也即做主从热备处理。

MySQL分表分库案例中，涉及到一个特别典型的问题，就是如何根据ID无需挨个分表查找、快速定位其所属分表，学过算法的都清楚，hash显然是最高效的一种查询方案，其算法复杂度为  $O(1)$ 。业界通常使用 一致性hash算法 确保同一ID标识的记录只会定位到同一个库的同一张分表中去，因它同其他hash算法相比有着无可比拟的优势。实际上该算法最初应用在分布式缓存中，它有着能避免一般hash算法在某个节点不可用时导致的雪崩效应的能力。缓存有个特点就是一个节点奔溃，其他节点可以替它完成工作没有命中则到数据库中查询，将查到的数据填入缓存，大多数微服务的服务实例也有着这个特点，因此Dubbo将其作为它负载均衡扩展点的一种实现。

## 算法介绍

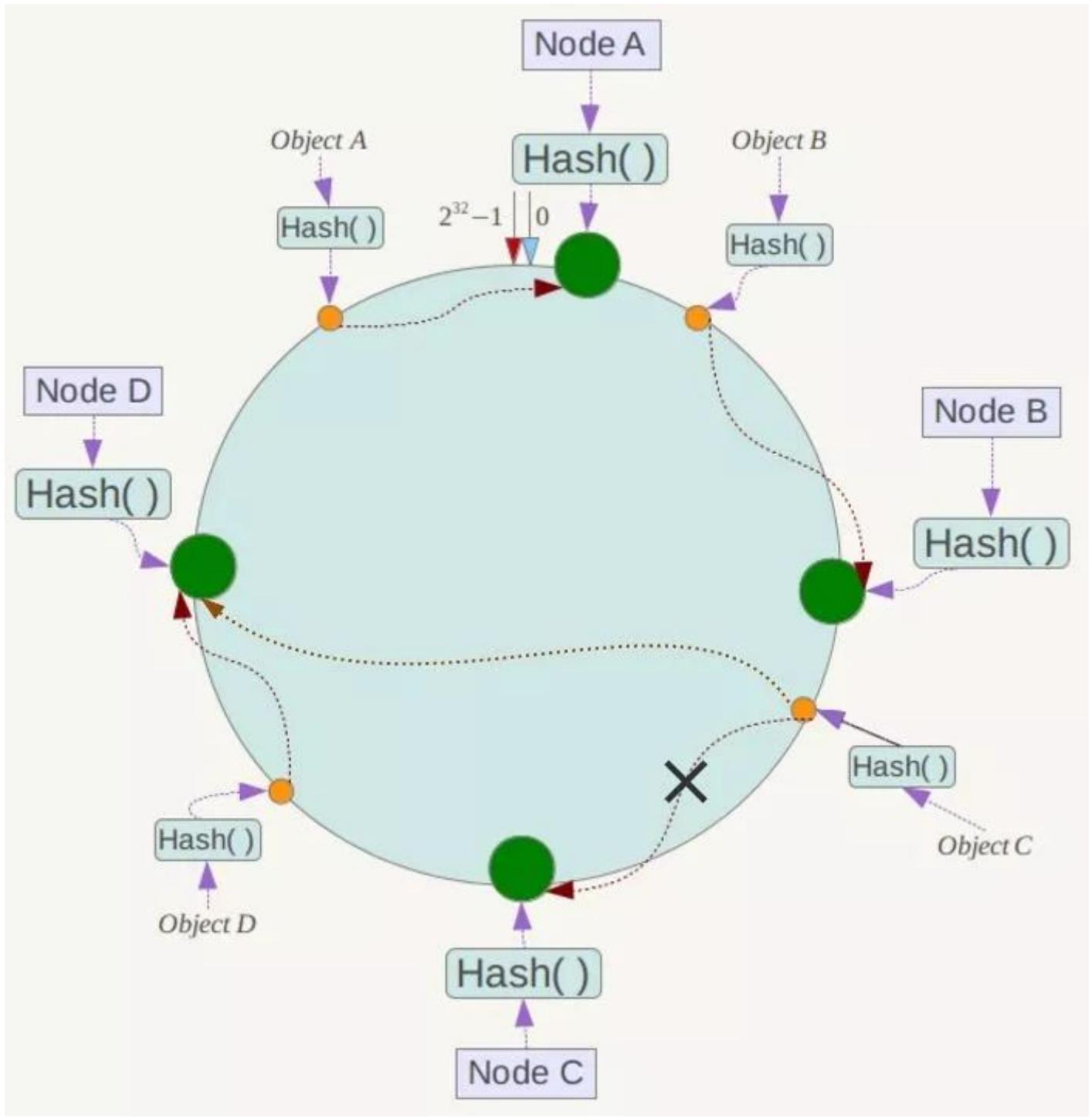
该章节参考一致性哈希算法原理 (<https://www.cnblogs.com/lpfuture/p/5796398.html>) 和什么是一致性Hash算法？ (<https://zhuanlan.zhihu.com/p/34985026>)。

一致性hash算法 将整个哈希值空间组织成一个虚拟圆环，其上顺时钟依序编号分布着  $2^{32}$  个点<sub>0</sub>和  $2^{32}$  位置的两个点重合，我们将其称为 Hash环。对所有参与服务的节点根据其唯一性标识服务器的IP或主机名进行 hash确定它们所在 Hash环 的位置，称它们为 服务节点 。接入客户端查询请求后，提取key按相同函数Hash得到 Hash环 上的点，由该点出发顺时针找到的第一个 服务节点 即为要找的目标。如下图，假设有 A、B、C、D 4个服务节点，接入了4个请求，Key分别为 Object A、Object B、Object C、Object D，hash处理示意图如下：

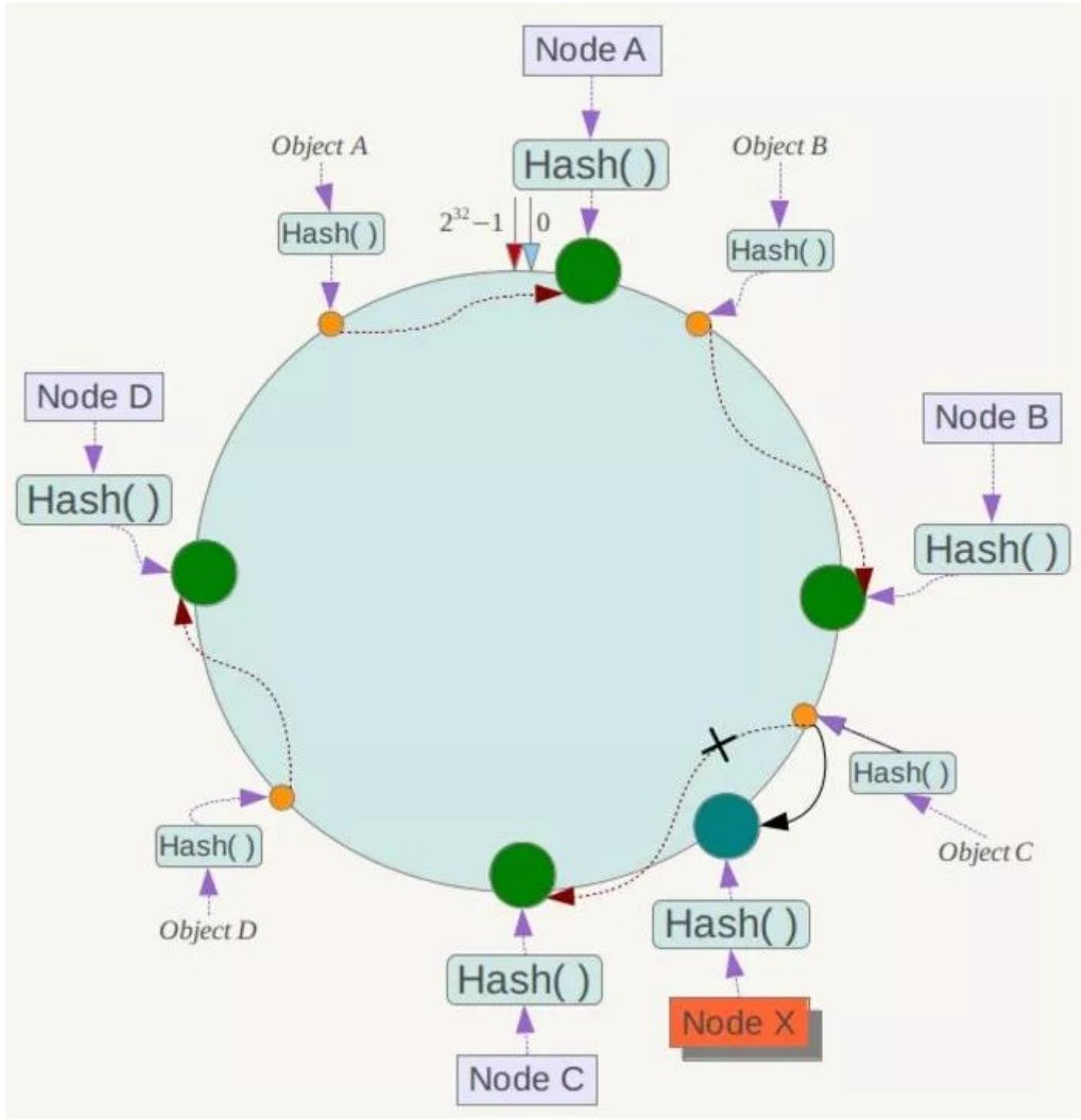


该算法有着良好的容错性和可扩展性。

Hash环中如果有一个服务节点不幸宕机，受影响的仅仅是该节点到前一节点所组成的哈希值空间，这个空间的接受的请求会被定位到后一节点上，如下，假设C掉线：

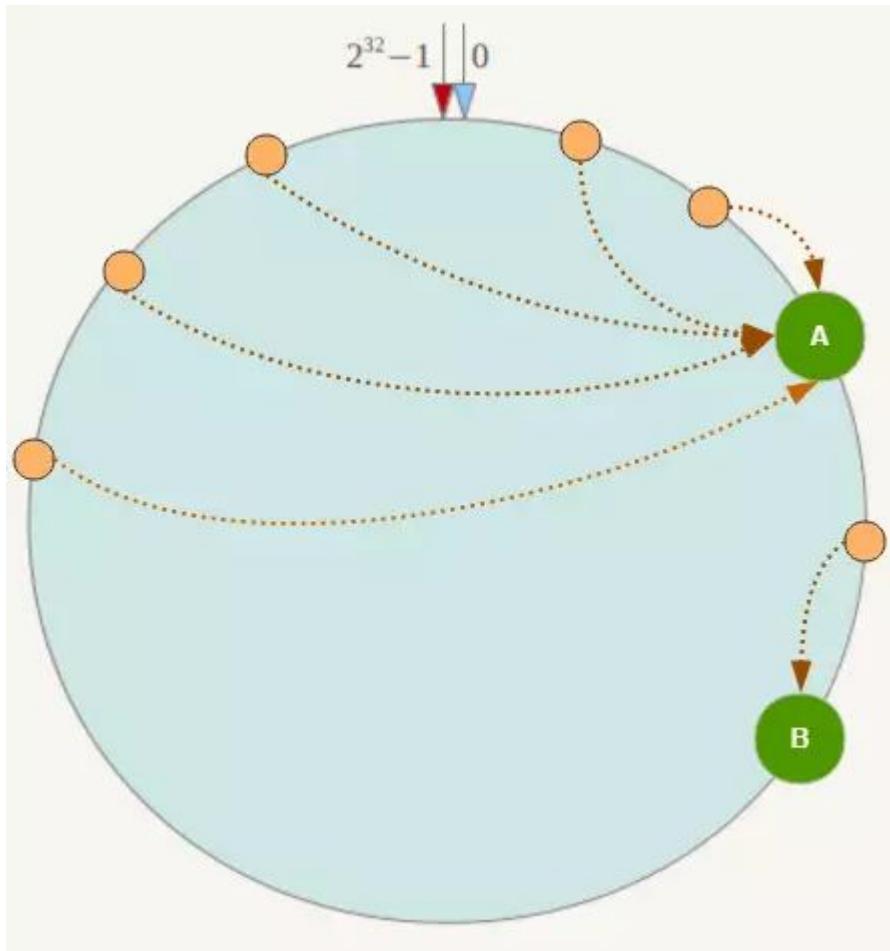


相应的在 Hash 环 动态地填入一个 服务节点，受影响的只是该新增节点到其前一个 服务节点 的所组成的哈希值空间，如下，假设X 服务节点 上线：

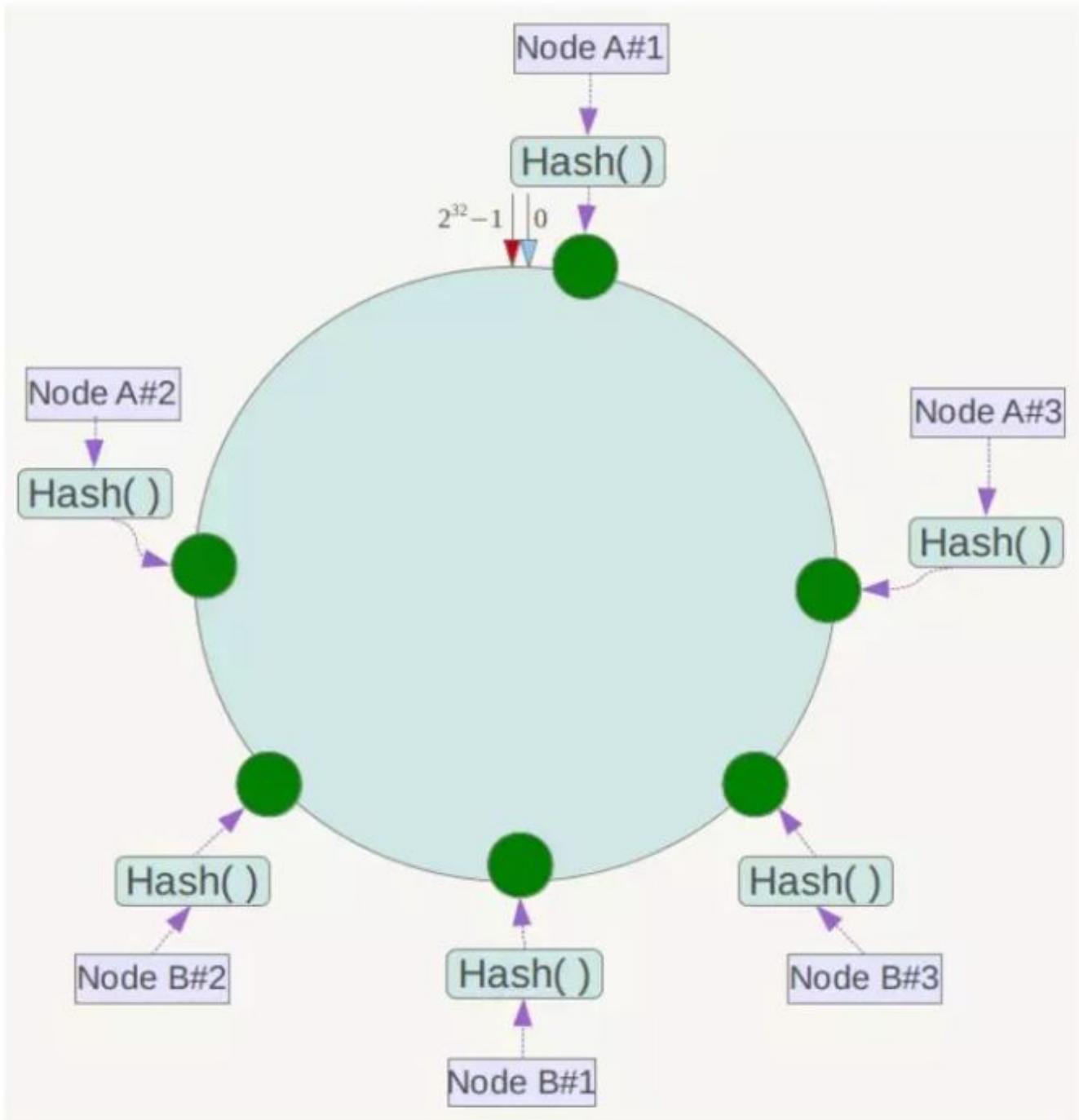


## 数据倾斜

一致性Hash算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题，例如只有两个节点，其环分布如下：



为解决数据倾斜问题，引入了虚拟节点机制，通过给每个服务节点编号，如“Node A#1”、“Node A#2”、“Node A#3”，计算多个hash值空间，如下所示：



在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

## 源码实现

扩展点类 `ConsistentHashLoadBalance` 的实现分为两部分，1) 一致性Hash算法选择器 `ConsistentHashSelector` 实现；2) 获取 `ConsistentHashSelector` 实例，用其挑选执行当前 RPC 方法的服务实例。

同其他算法实现不一样的是，`ConsistentHashSelector` 将算法实现部分以内部静态类的方式包装起来了，好处是在服务实例列表没有发生变化的情况下，可以反复利用，使用空间换时间，提升了微服务的执行效率，因而声明了针对算法实现的缓存容器 `ConcurrentMap<String,`

ConsistentHashSelector<?>> selectors。

Dubbo中的负载均衡目的是从一个微服务的多个服务实例中挑选一个用于执行当前RPC方法，一个微服务可以存在多个方法，也就是说服务实例是针对方法级别的，而非服务级别的，因而使用 group + "/" + interfaceName + ":" + version + "." + invocation.getMethodName() 作为 selectors 的Key键。

```
public class ConsistentHashLoadBalance extends AbstractLoadBalance {  
  
    private final ConcurrentHashMap<String, ConsistentHashSelector<?>> selectors = new  
    ConcurrentHashMap<>();  
  
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation  
invocation) {  
        String methodName = RpcUtils.getMethodName(invocation);  
        String key = invokers.get(0).getUrl().getServiceKey() + "." + methodName;  
        int identityHashCode = System.identityHashCode(invokers);  
        ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>)  
selectors.get(key);  
        if (selector == null || selector.identityHashCode != identityHashCode) {  
            selector = new ConsistentHashSelector<T>(invokers, methodName,  
identityHashCode);  
            selector = (ConsistentHashSelector<T>) selectors.get(key);  
        }  
        return selector.select(invocation);  
    }  
    ...  
}
```

因 identityHashCode() 不管对象是否重写了 hashCode() 方法，都会返回对象的 hash code，上述代码中使用 System.identityHashCode(invokers) 作为服务实例列表是否发生变化的检测依据，只要发生变化，就会重新生成服务实例选择器 ConsistentHashSelector。

下述将展开算法的具体实现部分的剖析，按照惯例，我们将源码打散，和上述算法对应起来，一个点一个点地逐个击破。

一致性hash算法的实现是以hash为基础的，上述提到的 Hash 环内的值的计算实际分为两步，先使用 MD5 计算得到 16 个字节的 byte[]，被切分为 4 段由低位到高位，再利用该数组取其中一段得到 4 个字节计算得到一个 32 位的二进制数字，刚好是一个 long 类型数字。实现源码如下：

```

private byte[] md5(String value) {
    MessageDigest md5;
    try {
        md5 = MessageDigest.getInstance("MD5");
    } catch (NoSuchAlgorithmException e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
    md5.reset();
    byte[] bytes = value.getBytes(StandardCharsets.UTF_8);
    md5.update(bytes);
    return md5.digest();
}

```

//获取number指定的一段4个字节（每字节8位），首先取得的字节形成结果整数的高位

```

private long hash(byte[] digest, int number) {
    return (((long) (digest[3 + number * 4] & 0xFF) << 24)
        | ((long) (digest[2 + number * 4] & 0xFF) << 16)
        | ((long) (digest[1 + number * 4] & 0xFF) << 8)
        | (digest[number * 4] & 0xFF))
        & 0xFFFFFFFFL;
}

```

注：实际上上述方案潜藏着bug，散列方案并不完美，两个不同的数据内容，散列得到的整数可能刚好一样。

一个RPC调用请求接入后，需要使用同一个hash算法对提取到的Key执行hash求值，以确定其到底被那个服务节点所调用，RPC方法表征的是一个原生的Java方法调用，使用`invocation.getArguments()`能提取到Key。

如果使用一致性hash算法作为负载均衡器，Dubbo要求得另外在微服务引用实例的配置总线中加入配置项`invocation[methodName]+".+"+hash.arguments"`，值为以","分割的参数索引位置，顺序随意。

```

public static final String HASH_ARGUMENTS = "hash.arguments";

private final int[] argumentIndex;

ConsistentHashSelector(List<Invoker<T>> invokers,
    String methodName, int identityHashCode) {
    ...
    String[] index = COMMA_SPLIT_PATTERN.split(
        url.getMethodParameter(methodName, HASH_ARGUMENTS, "0"));

    argumentIndex = new int[index.length];
    for (int i = 0; i < index.length; i++) {
        argumentIndex[i] = Integer.parseInt(index[i]);
    }
    ...
}

public Invoker<T> select(Invocation invocation) {
    String key = toKey(invocation.getArguments());
    byte[] digest = md5(key);
    return selectForKey(hash(digest, 0));
}

private String toKey(Object[] args) {
    StringBuilder buf = new StringBuilder();
    for (int i : argumentIndex) {
        if (i >= 0 && i < args.length) {
            buf.append(args[i]);
        }
    }
    return buf.toString();
}

```

## NOTE

选用一致性hash算法的一个重要原因是实现粘滞，也就是说让满足某些特征的数据尽量被分派在同一个实例上运行，或者说针对同一RPC方法的前后两次不同的调用，如果某些参数一样，那么它就应该落在同一个实例上。Dubbo的实现给予了开发充分的自由去决策配置所使用哪些参数作为特征依据。

最后还剩下一个问题，就是如何将服务实例散列到 Hash环上，包括虚拟节点的处理。如果没有特殊配置的话，Dubbo会为每个服务实例产生160个虚拟几点，也可就配置项 `invoker.url[invocation[methodName]+".+"+"hash.nodes"]` 设置值，在映射 Hash环 目标位置时，使用 `invoker.url[host] ":" + invoker.url[port]] + index` 取hash的方式，和上述RPC方法取hash不同的是，这次使用 md5 计算得到的16个字节全部被派上了用场，因而多了 `replicaNumber / 4` 和最里头的一层循环处理。

包括虚拟节点在内的所有服务节点在 Hash 环上表示为一个 0 到  $2^{32}$  的整数，需要对应关联服务实例，也即 `<Long, Invoker<T>>`，另外最好能兼顾实现根据请求数据提取的 Key 找到对应服务节点，Java 中满足这些要求的理想的容器是 TreeMap，如下：

```
public static final String HASH_NODES = "hash.nodes";  
  
private static final class ConsistentHashSelector<T> {  
  
    private final TreeMap<Long, Invoker<T>> virtualInvokers;  
  
    private final int replicaNumber;  
  
    private final int identityHashCode;  
  
    ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, int identityHashCode) {  
        this.virtualInvokers = new TreeMap<Long, Invoker<T>>();  
        this.identityHashCode = identityHashCode;  
        URL url = invokers.get(0).getUrl();  
        this.replicaNumber = url.getMethodParameter(methodName, HASH_NODES, 160);  
        ...//特征参数索引处理  
        for (Invoker<T> invoker : invokers) {  
            String address = invoker.getUrl().getAddress();  
            for (int i = 0; i < replicaNumber / 4; i++) {  
                byte[] digest = md5(address + i);  
                for (int h = 0; h < 4; h++) {  
                    long m = hash(digest, h);  
                    virtualInvokers.put(m, invoker);  
                }  
            }  
        }  
    }  
    private Invoker<T> selectForKey(long hash) {  
        Map.Entry<Long, Invoker<T>> entry = virtualInvokers.ceilingEntry(hash);  
        if (entry == null) {  
            entry = virtualInvokers.firstEntry();  
        }  
        return entry.getValue();  
    }  
    ...  
}
```

TreeMap 使用红黑树实现，插入到其中的元素是按 Key 键排序的，支持 Key 键做升序访问。它实现了 NavigableMap → SortedMap 接口，具有了针对给定搜索目标返回最接近匹配项的导航方法。方法 `lowerEntry()`、`floorEntry()`、`ceilingEntry()` 和 `higherEntry()` 分别返回与小于、小于等于、大于等于、大于给定键的键关联的 Map.Entry 对象，如果不存在这样的键，则返回 null。

另外 `selectForKey()` 方法中在 `ceilingEntry()` 方法没有找到满足要求的节点结果为 null 时，调用了 `virtualInvokers.firstEntry()` 返回第一个节点，这正是 Hash 环闭环形成的位置 0 和  $2^{32}$  重合的点。

完結

## 【十二】Dubbo集群之路由

---

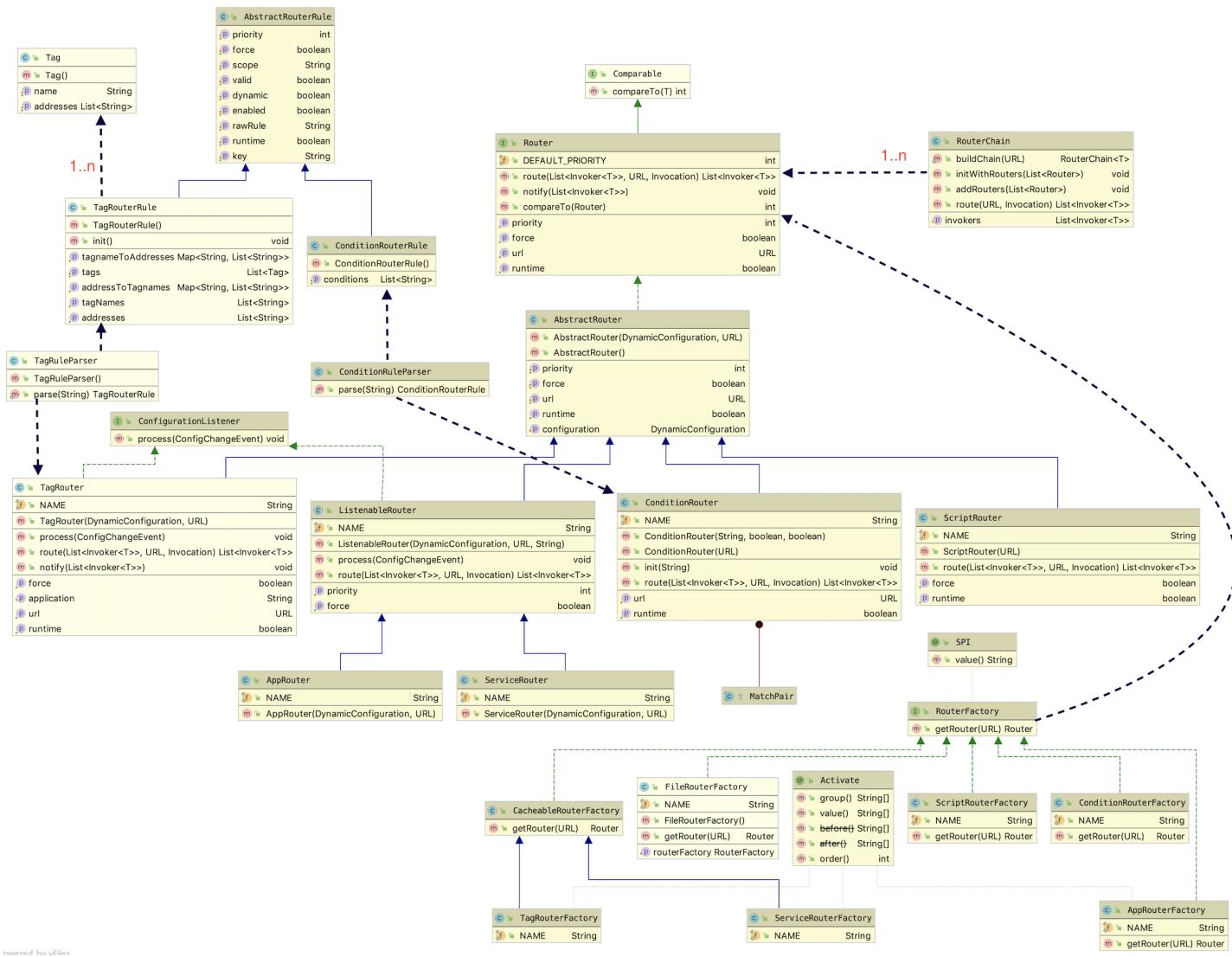
上一篇文章中，我们详细剖析了Dubbo中的负载均衡实现。其实微服务框架中，从服务实例集群中挑选一个实例提交RPC请求是一个目标逐步缩小的过程，比如在负载均衡得到最终的实例前，会先经过路由从一个更大的集合中（包含了当前应用中特定被引用微服务的所有可用实例）过滤得到子集。

来看看Dubbo官方关于其路由 (<http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html>) 的介绍：

“ 路由规则在发起一次RPC调用前起到过滤目标服务器地址的作用，过滤后的地址列表，将作为消费端最终发起RPC调用的备选地址。

- **条件路由**: 支持以服务或*Consumer*应用为粒度配置路由规则。
- **标签路由**: 以*Provider*应用为粒度配置路由规则。

关于如何实现由配置文件(包括 \*.yml )解析得到的路由规则，以及由路由规则如果从所有给定集合中匹配出目标子集，都不是本文的重点，一个技术点如何实现是本序列文章坚持的初衷，首先看看Router实现的大体方式，下图是整个Router实现的类框架图：



从上图中可以看出，实现类大体分为3部分：

1. 路由规则解析：实际上分为解析器和由其解析得到的规则；
2. 路由器：主要包括条件路由和标签路由；
3. 路由器创建工厂：每种工厂实现会创建一种类型的路由器，同时也是一个扩展点，有些具类是自激活的，比如 AppRouterFactory、ServiceRouterFactory、TagRouterFactory 以及未在图中体现的 MockRouterFactory；

## 路由器创建工厂

从上文的类框架图中看，这是最简单的一部分，借助了工厂方法模式创建相应的路由器实现。这一设计模式有个缺陷是一个产品的实现会对应多出一个抽象工厂的实现，尽管Dubbo中到处都有设计模式的身影，但于导致类爆炸的模式的引入是相当谨慎的。稍微深入 RouterFactory 的多个实现的话，发现其7个实现中有几个实现相对于其他的有些不同之处，工厂方法存在的目的正是利用工厂实现类隐藏一个具体产品实现类的创建细节。

## Router实例获取

## NOTE

**RouterFactory** 本身是一个扩展点，也就是说它的每一个实现类都是单例的。

整个路由模块中 **RouterFactory** 的作用比较重要，作为扩展点，其实例决定了如何创建一个 **Router** 实例，如下是其定义：

```
/*
 * Note Router has a different behaviour since 2.7.0, for each type of Router, there
will only has one Router instance for each service.
 */
@SPI
public interface RouterFactory {
    /**
     * Since 2.7.0, most of the time, we will not use @Adaptive feature, so it's kept
only for compatibility.
     *
     * @param url url
     * @return router instance
     */
    @Adaptive("protocol")
    Router getRouter(URL url);
}
```

上述源码中保留了部分注释，1) 自 v2.7.0 后，每个服务只有一个Router实例，详见下文解释；2) 在 v2.7.0 后也不再使用自适配模式获取 **RouterFactory** 的实例。

路由器实现最重要的是由配置数据得到路由规则，用特定的规则过滤得到一个微服务引用实例的目标子集，其中路由配置数据的格式多样，可以是\*.yml、.js、.xml、\*.json甚至是来源于配置中心的二进制数据，具体取决于实现，而这些数据最终都是通过配置总线URL传入的。

很显然，**RouterFactory** 存在的价值便是创建 **Router** 的实例，也就是说使用 **Router** 的接口是 **RouterFactory**，我们继续深入究竟如何获得一个实例。

**RouterFactory** 的创建方式有两种方式，如下所示，**RouterChain** 获取的所有自激活的实例，而 **RegistryDirectory** 则使用了Dubbo中SPI扩展中的自适配模式获取实例，生成的扩展点具类 **RouterFactory** 的代理实现会试图根据 **url.protocol** 从SPI配置文件中获取到所匹配的目标扩展点具类，将相应行为委托给该目标类。

- *RouterChain*

```
public List<RouterFactory> extensionFactories =
ExtensionLoader.getExtensionLoader(RouterFactory.class).getActivateExtension(url,
(String[]) null);
```

- *RegistryDirectory*

```
public RouterFactory ROUTER_FACTORY =  
ExtensionLoader.getExtensionLoader(RouterFactory.class).getAdaptiveExtension();
```

JAVA

## RouterChain

基本上可以说 Router 的使用止于 RouterChain，后者利用它向外提供如下方法，为一个RPC方法调用返回所有能匹配到微服务引用实例，其中配置总线 url 含有匹配目标微服务的特征信息，invocation 则表征了一个PRC方法的包括类型在内的所有入参数据。

```
public List<Invoker<T>> route(URL url, Invocation invocation) {  
    List<Invoker<T>> finalInvokers = invokers;  
    for (Router router : routers) {  
        finalInvokers = router.route(finalInvokers, url, invocation);  
    }  
    return finalInvokers;  
}
```

JAVA

### NOTE

RouterChain#route(...) 方法的执行逻辑是，针对当前被引用微服务的可用实例全集，先缓存在临时容器 finalInvokers 中，然后遍历所有路由器，逐个路由过滤 finalInvokers，每一次迭代均会将所获子集赋值给 finalInvokers，最后得到目标集合。

那 RouterChain 中的所有微服务引用实例又是谁提供的呢？初步接触微服务时，培训教材或者框架指导都会告知开发者，服务提供者在启动的时候会将自身注册到注册中心，对，这些实例来源于注册中心，只不过是过程没那么直接。既然不是自身产生，肯定还得提供由外界设值的方法，如下：

```
// full list of addresses from registry, classified by method name.  
private List<Invoker<T>> invokers = Collections.emptyList();  
  
/**  
 * Notify router chain of the initial addresses from registry at the first time.  
 * Notify whenever addresses in registry change.  
 */  
public void setInvokers(List<Invoker<T>> invokers) {  
    this.invokers = (invokers == null ? Collections.emptyList() : invokers);  
    routers.forEach(router -> router.notify(this.invokers));  
}
```

JAVA

v2.7.0 之前因为每一个表征微服务引用实例的URL产生一个 Router 实例，而之后的版本是随 RouterFactory 的具类都是单例的，在一个应用中 Router 实现也是单例的。

新版的 Router 实例在当前JVM中首次调用 buildChain() 就产生了，后续再次调用不会再产生新的实例，虽然查看源码，过程没那么显然。仔细研读 RouterChain 会发现它本身并不是单例的，是服务级别的，一种类型的被引用微服务会对应产生一个它的实例（由标注了 @Activate 注解的 RouterFactory 具类生成），它们被认为是内置 Router 实例。

老版的 Router 实例是由服务目录 Directory 提供的，每一次可用服务列表发生变动时需要全部更新。

综上，为兼容， RouterChain 中提供两个盛装 Router 实例的 List 容器，并且给前一个添加了 volatile 修饰，每次更新时会加上所有内置 Router 实例。

```
public class RouterChain<T> {
    private volatile List<Router> routers = Collections.emptyList();

    private List<Router> builtinRouters = Collections.emptyList();

    public static <T> RouterChain<T> buildChain(URL url) {
        return new RouterChain<>(url);
    }

    private RouterChain(URL url) {
        List<RouterFactory> extensionFactories =
ExtensionLoader.getExtensionLoader(RouterFactory.class)
            .getActivateExtension(url, (String[]) null);

        List<Router> routers = extensionFactories.stream()
            .map(factory -> factory.getRouter(url))
            .collect(Collectors.toList());

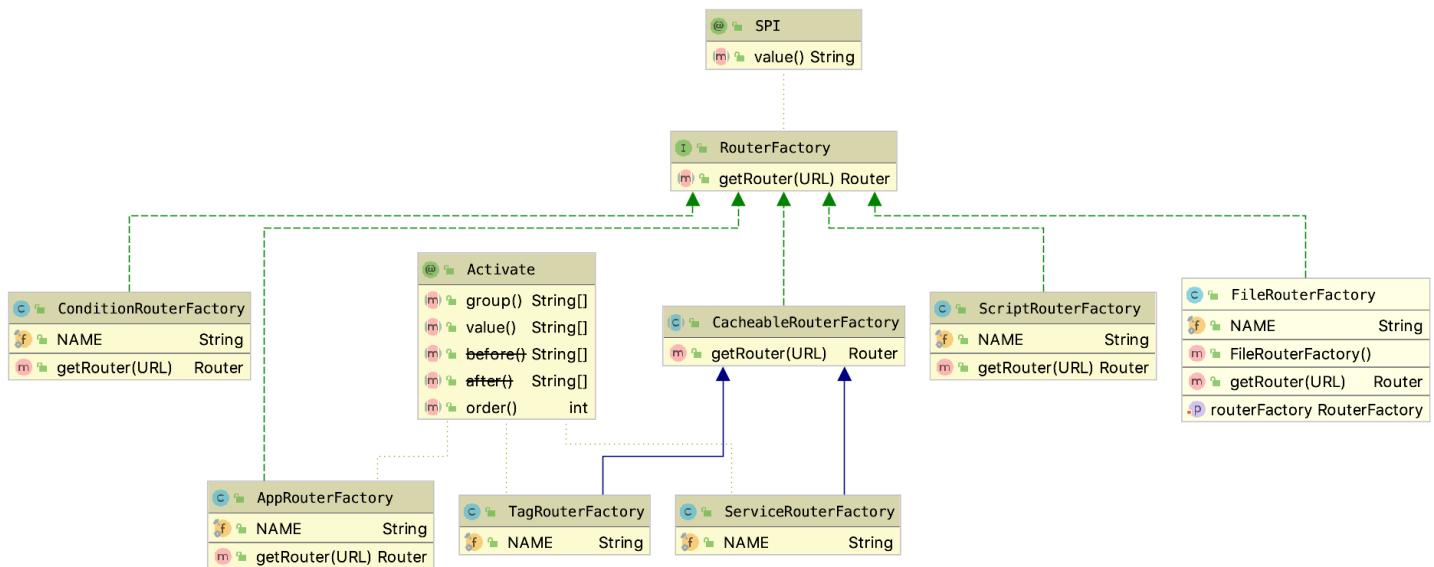
        initWithRouters(routers);
    }

    public void initWithRouters(List<Router> builtinRouters) {
        this.builtinRouters = builtinRouters;
        this.routers = new ArrayList<>(builtinRouters);
        this.sort();
    }

    public void addRouters(List<Router> routers) {
        List<Router> newRouters = new ArrayList<>();
        newRouters.addAll(builtinRouters);
        newRouters.addAll(routers);
        CollectionUtils.sort(newRouters);
        this.routers = newRouters;
    }

    private void sort() {
        Collections.sort(routers);
    }
    ...
}
```

## RouterFactory 工厂实现



上图中体现了有两组几乎一模一样的工厂类实现，分别是：1) ScriptRouterFactory 和 ConditionRouterFactory；2) ServiceRouterFactory 和 TagRouterFactory。相比前面一组，后面一组支持运行时配置更新且是无条件激活的。具体如下源码：

```

public class ScriptRouterFactory implements RouterFactory {

    public static final String NAME = "script";

    @Override
    public Router getRouter(URL url) {
        return new ScriptRouter(url);
    }

}

public class ConditionRouterFactory implements RouterFactory {

    public static final String NAME = "condition";

    @Override
    public Router getRouter(URL url) {
        return new ConditionRouter(url);
    }

}

@Activate(order = 300)
public class ServiceRouterFactory extends CacheableRouterFactory {

    public static final String NAME = "service";

    @Override
    protected Router createRouter(URL url) {
        return new ServiceRouter(DynamicConfiguration.getDynamicConfiguration(), url);
    }

}

@Activate(order = 100)
public class TagRouterFactory extends CacheableRouterFactory {

    public static final String NAME = "tag";

    @Override
    protected Router createRouter(URL url) {
        return new TagRouter(DynamicConfiguration.getDynamicConfiguration(), url);
    }

}

```

JAVA

着重点还是放在后面这一组扩展继承了 CacheableRouterFactory 的，Dubbo要求如果在v2.7.0以上做自定义路由器实现，需要扩展继承它，否则直接实现 RouterFactory 。同前一组实现不同的是，其基类中加入了缓存，每一个能由 '{group}/{interfaceName}:{version}' 唯一标识的被引用微服务实例在首次获取到Router实例后，便会将其缓存以便同一被引用微服务的其它实例重用，更深一层的目的是节约规则解析时间，提升效率。

```

public abstract class CacheableRouterFactory implements RouterFactory {
    private ConcurrentMap<String, Router> routerMap = new ConcurrentHashMap<>();

    @Override
    public Router getRouter(URL url) {
        routerMap.computeIfAbsent(url.getServiceKey(), k -> createRouter(url));
        return routerMap.get(url.getServiceKey());
    }

    protected abstract Router createRouter(URL url);
}

```

JAVA

应用级别的路由器工程类实现稍微特别点，考虑到如下两个原因，`AppRouterFactory` 使用了`volatile`关键词确保只创建一个`AppRouter`实例：

1. 工厂实现类的实例化也是在多线程环境下进行；
2. `AppRouterFactory` 使用了类似Zookeeper和Etcd等的支持键值存取的中间件作为配置存取中心，一个应用只能存在一个用于同该中心交互的实例，否则会无辜浪费计算资源；

源码实现本身很简单，如下：

```

@Activate(order = 200)
public class AppRouterFactory implements RouterFactory {
    public static final String NAME = "app";

    private volatile Router router;

    @Override
    public Router getRouter(URL url) {
        if (router != null) {
            return router;
        }
        synchronized (this) {
            if (router == null) {
                router = createRouter(url);
            }
        }
        return router;
    }

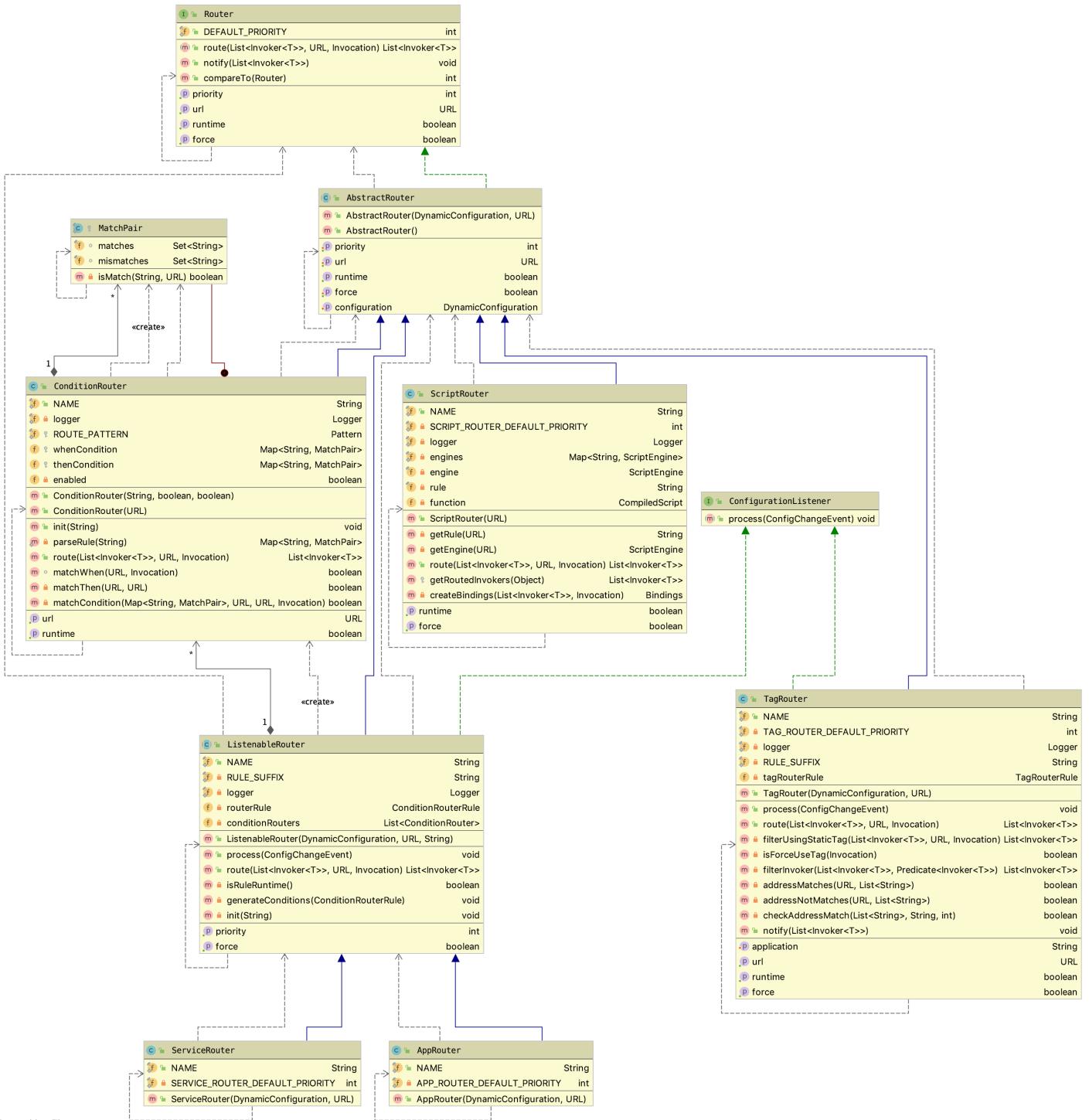
    private Router createRouter(URL url) {
        return new AppRouter(DynamicConfiguration.getDynamicConfiguration(), url);
    }
}

```

JAVA

## 路由器实现

搞清楚了 Router 本身是如何产生的，以及和外界关系后，终于轮到本文的最重要的部分了，路由器实现。这里涉及大量匹配细节，并不是我们需要关心的，本章节将从更加宏观的角度加以剖析，避免落入尘海，关注重点将会更倾向于同整个集群的关系。



## Router 接口定义

从上文中得知，Router 的作用就是为特定被引用微服务的所有实例（由配置中心针对被引用微服务同步得到）根据当前路由配置筛选出一个目标子集。接口定义如下：

```
public interface Router extends Comparable<Router> {  
    int DEFAULT_PRIORITY = Integer.MAX_VALUE;  
  
    URL getUrl();  
  
    //referUrl, 表征被引用微服务的配置总线数据  
    <T> List<Invoker<T>> route(List<Invoker<T>> invokers, URL referUrl, Invocation  
    invocation) throws RpcException;  
  
    default <T> void notify(List<Invoker<T>> invokers) {  
    }  
  
    boolean isRuntime();  
  
    boolean isForce();  
  
    int getPriority();  
  
    @Override  
    default int compareTo(Router o) {  
        if (o == null) {  
            throw new IllegalArgumentException();  
        }  
        return Integer.compare(this.getPriority(), o.getPriority());  
    }  
}
```

JAVA

上述需要特别提及的有如下几处地方：

1. `boolean isForce()`：用于确定在没有匹配到目标引用实例时，当前 `route(...)` 执行结果是否生效，默认配置为false，返回入参传入的服务引用实例集合，否则会返回一个空的结果集；
2. `int getPriority()`：为同一个服务提供路由功能的所有Router实例具有优先级，`RouterChain` 实现中，无论是初始化内置的 `Routers`，还是为兼容 v2.7.0 以前版本的 `addRouters(List<Router> routers)`，均使用依赖优先级的排序；
3. `<T> void notify(List<Invoker<T>> invokers)`：`RouterChain` 中出现过 `routers.forEach(router → router.notify(this.invokers))` 这一句源码，目的是如果微服务引用实例列表有更新，得通知所有相关 `Router` 做出相应处理。

## 配置相关

上文关于 `RouterFactory` 源码实现剖析中，`AppRouter`、`ServiceRouter` 和 `TagRouter` 的实例创建都使用到了下述代码片段：

```
DynamicConfiguration.getDynamicConfiguration()
```

JAVA

仔细翻看它们的基类 `AbstractRouter`，发现除了需要提供 `Router` 必要的 `url URL`、`force boolean` 和 `priority int` 外，还有一个必须在构造函数中就提供值的 `configuration DynamicConfiguration` 类型属性。顾名思义，这和动态配置有关，`DynamicConfiguration` 定义在 `dubbo-configcenter` 包中，也就是说 `AppRouter`、`ServiceRouter` 和 `TagRouter` 都和依赖于配置中心，微服务开发中，像路由规则这种跨实例共享的配置数据被鼓励使用配置中心做存取操作。而处于配置中心的配置项数据发生变化，相关联节点必须要及时感知到，这是保证服务可靠性的前提，因而它们都直接或间接地实现了也定义在 `dubbo-configcenter` 包中的 `ConfigurationListener` 接口。基本实现方式是若配置变更事件是删除，则直接删除此前解析得到的规则，否则重新解析覆盖原有规则。接口定义如下：

```
//Config listener, will get notified when the config it listens on changes. JAVA
public interface ConfigurationListener {

    //Listener call back method. Listener gets notified by this method once there's any
    change happens on the config the listener listens on.
    void process(ConfigChangeEvent event);
}
```

其中 `ConfigChangeEvent` 是一个含有3个属性类，包括配置项 `key` 和配置值 `value`，以及事件类型 `ConfigChangeType {ADDED、MODIFIED、DELETED}`。

## IMPORTANT

Dubbo 中一个应用的路由规则是以应用或服务级别整体存入到配置中心的，取也是整块的，也就是整取整存式的。

另外一方面，路由器的数据源，也即微服务引用实例这些数据来源于注册中心，`Router` 实现本身也要及时感知实例列表的变化。因此我们看到上述 `Router` 接口中定义了 `notify()` 方法，`TagRouter` 实现了该方法。

## NOTE

注册中心和配置中心只是一种逻辑概念，有时候他们可以共享同一个服务，比如使用一个Zookeeper服务集群，只是根据不能的功能使用不同节点数据。有关配置中心和注册中心的实现并非本文主题，将在相关实现剖析文章中体现。

## ListenableRouter AppRouter & ServiceRouter

`AppRouter` 和 `ServiceRouter` 只有很少的一点代码，路由的实现在基类 `ListenableRouter` 中，如下：

```

public class AppRouter extends ListenableRouter {
    public static final String NAME = "APP_ROUTER";
    /**
     * AppRouter should after ServiceRouter
     */
    private static final int APP_ROUTER_DEFAULT_PRIORITY = 150;

    public AppRouter(DynamicConfiguration configuration, URL url) {
        super(configuration, url, url.getParameter(CommonConstants.APPLICATION_KEY));
        this.priority = APP_ROUTER_DEFAULT_PRIORITY;
    }
}

public class ServiceRouter extends ListenableRouter {
    public static final String NAME = "SERVICE_ROUTER";
    /**
     * ServiceRouter should before AppRouter
     */
    private static final int SERVICE_ROUTER_DEFAULT_PRIORITY = 140;

    public ServiceRouter(DynamicConfiguration configuration, URL url) {
        super(configuration, url, DynamicConfiguration.getRuleKey(url));
        this.priority = SERVICE_ROUTER_DEFAULT_PRIORITY;
    }
}

```

从上文已经得知 ListenableRouter 除了需要实现路由微服务引用实例的子集这一主体功能外，还需要及时响应来自配置中心的配置修改事件，确保所使用子集的实时有效。然而前者是委托给 ConditionRouter 实现的，也就是说条件路由支持的粒度可以是应用级别的也可以是服务级别的。

Java面向对象编程中一提及监听器，熟悉设计模式的同学，总会第一时间在脑海中浮现 观察者模式。其实现的基础是被观察主题 Subject 提供了回调接口 Callback，实现了 Callback 的观察者 Observer 需要将自身注册加入到 Subject 的 observers 集合中，有新的事件发生时，Subject 会从 observers 将元素挨个取出，执行其 Callback 回调。

因此 ListenableRouter 在初始化的第一时间调用 addListener() 方法便完成自身的注册处理，具体如下源码：

```
public abstract class ListenableRouter extends AbstractRouter implements ConfigurationListener {
    private static final String RULE_SUFFIX = ".condition-router";

    public ListenableRouter(DynamicConfiguration configuration, URL url, String ruleKey) {
        super(configuration, url);
        this.force = false;
        this.init(ruleKey);
    }
    ...

    private synchronized void init(String ruleKey) {
        if (StringUtils.isEmpty(ruleKey)) {
            return;
        }
        String routerKey = ruleKey + RULE_SUFFIX;
        configuration.addListener(routerKey, this);
        String rule = configuration.getRule(routerKey,
DynamicConfiguration.DEFAULT_GROUP);
        if (StringUtils.isNotEmpty(rule)) {
            this.process(new ConfigChangeEvent(routerKey, rule));
        }
    }
}
```

JAVA

上述 `init()` 方法中，`Router` 向注册中心完成自身的注册后，立马又使用 `routerKey` 从中获取到所有的路由配置数据，然后回调了自身实现的 `ConfigurationListener` 接口，目的是确保及时完成 `conditionRouters` 设值处理，保证主体逻辑的可用。

接口实现逻辑处理如下，如上文所言，配置中心如果将相应的路由规则配置数据删除了，本地相应需要将所有解析得到的路由规则及所有微服务引用实例的列表都清空，直接后果后续RPC请求进入后，找不到可用的引用实例，这种极端情况一般不多见，配置全覆盖式导入的实现方式可能采取的先删后增策略。

```

public abstract class ListenableRouter extends
    AbstractRouter implements ConfigurationListener {
    private List<ConditionRouter> conditionRouters = Collections.emptyList();
    private ConditionRouterRule routerRule;
    @Override
    public synchronized void process(ConfigChangeEvent event) {
        if (logger.isInfoEnabled()) {
            logger.info("Notification of condition rule, change type is: "
                    + event.getChangeType() + ", raw rule is:\n " + event.getValue());
        }

        if (event.getChangeType().equals(ConfigChangeType.DELETED)) {
            routerRule = null;
            conditionRouters = Collections.emptyList();
        } else {
            try {
                routerRule = ConditionRuleParser.parse(event.getValue());
                generateConditions(routerRule);
            } catch (Exception e) {
                logger.error("Failed to parse the raw condition rule and it will"
                        + " not take effect, please check " +
                        "if the condition rule matches with the template, the raw rule
is:\n "
                        + event.getValue(), e);
            }
        }
    }

    private void generateConditions(ConditionRouterRule rule) {
        if (rule != null && rule.isValid()) {
            this.conditionRouters = rule.getConditions()
                .stream()
                .map(condition -> new ConditionRouter(
                    condition, rule.isForce(), rule.isEnabled()))
                .collect(Collectors.toList());
        }
    }
    ...
}

```

`init()` 方法和所实现接口 `process()` 方法均加了 `synchronized` 对象锁修饰符。上文中我们已经阐述过 `AppRouter` 和 `ServiceRouter` 等效于是单例的以非常间接的方式实现，而路由功能是在并发场景下使用的，因而 `process()` 加了当前对象级别的锁不难理解。至于 `init()` 为啥要加锁还得深入编译器的优化，一个对象的初始化实际上分为如下两步，也就是JVM有可能完成第一步操作后，对象于外界已经可见了。

1. `memory = allocate(); //1.分配对象的内存空间`
2. `ctorInstance(memory); //2.初始化对象`

假设 `init()` 方法并未加锁，刚好在其执行完 `addListener()`，CPU时间已经让渡出去了，恰好配置中心负责通知回调的线程抢到了CPU资源，由于回调的 `process()` 方法均加了对象锁，锁只要没有释放，当前 `init()` 就会被阻塞不能继续往下执行，等锁被释放后，`init()` 方法返回后在获得锁又重新执行一回 `process()`。进一步假设前面那个回调是 `ConfigChangeType.DELETED`，这时回过头来发现 `init()` 方法执行的从配置中心拉取配置数据解析得到路由规则这个任务等于是白做了。而 `init()` 加锁后就等于把这本该接连发生的操作给串行化了，不会有这样的并发问题出现。

## IMPORTANT

`AppRouter` 和 `ServiceRouter` 实现上稍微有点不同的是在配置中心的Key的取值：

1. `ServiceRouter` : `{interface}[:" + version] [":" + group] + ".condition-router"`
2. `AppRouter` : `{application} + ".condition-router"`

---

未完待续

# Dubbo集群之容错

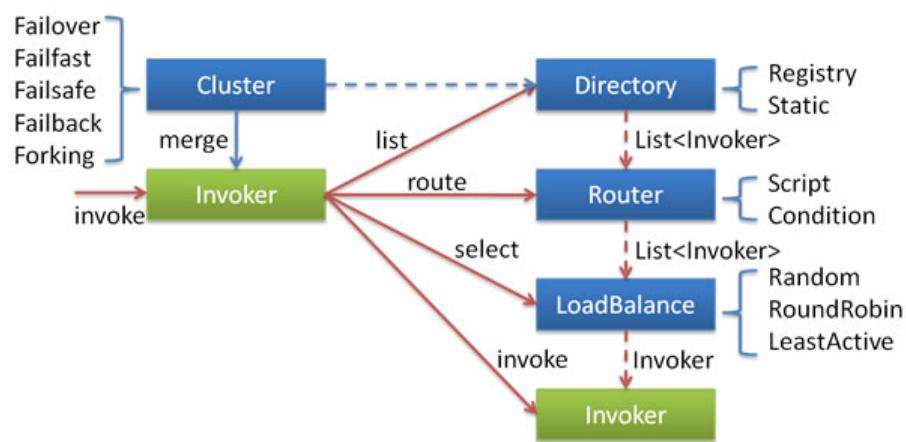
“在分布式系统中，集群某个某些节点出现问题时是大概率事件，因此在设计分布式RPC框架的过程中，必须要把失败作为设计的一等公民来对待。

由官网中的这一段话，可以看出容错在微服务框架中的作用举足轻重，它是保证服务高可用性的关键举措。容错策略有好几种，但却没有适合所有场景的银弹，因此Dubbo提供能如下几种供用户根据需要选用，`Failover` 是默认方案：

1. `Failover` (失败自动切换)
2. `Failsafe` (失败安全)
3. `Failfast` (快速失败)
4. `Fallback` (失败自动恢复)
5. `Forking` (并行调用)
6. `Broadcast` (广播调用)

## 容错架构

Dubbo中的容错是和集群中的其它组件一起发生作用的，其关系如下图所示：



图中各组件的作用及关系：

- **Invoker** 表示的是一个微服务的引用实例，它封装了 **Provider** 地址及 **Service** 接口信息；
- **Directory** 目录服务，根据服务名得到的同一个微服务的多个引用实例集合，该集合会动态变化的，比如注册中心推送变更；
- **Cluster** 将 **Directory** 中的多个 **Invoker** 伪装成一个 **Invoker**，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个；

- Router 负责从多个 Invoker 中按路由规则选出子集，比如读写分离，应用隔离等；
- LoadBalance 负责从多个 Invoker 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选；

`ClusterInvoker` 可以拥有自己的配置，继承自 `Directory` 的配置总线URL。

**NOTE**

Dubbo中，在应用内部提供配置同步入站和获取功能被视作目录服务，也即 `Directory`；而向所有微服务、应用提供配置管理、获取、同步出站、入站功能，独立于微服务存在的服务被视作注册中心，也即 `Registry`（由于配置中心往往和注册中心融为一体，对外被同一当做注册中心）。

## 容错实现

各个不同的策略继承自 `AbstractClusterInvoker`，以共用一套基础代码，它负责将 `Directory`、`Router`、`LoadBalance` 联结起来，构成一个有机整体，以最终提供一个负责实际RPC调用的 `Invoker` 对象。

下文中均以 `ClusterInvoker` 代替所有 `AbstractClusterInvoker` 实现，也即有集群特性的 `Invoker`。

## 服务级别的 `ClusterInvoker` 单例

像其它 `Invoker` 一样，`ClusterInvoker` 是一个行为实体，专门负责某个特定引用微服务的容错处理，其过程都封装在内部。一个被引用的微服务只会使用一种容错策略，在应用实例启动的那时就已经确定了——由配置文件等事先确定。这也说明应用需要在合适的时机为每一种被引用微服务选用一种策略并创建它的一个 `ClusterInvoker` 实例。`ClusterInvoker` 支持泛型，泛型参数是微服务的接口类型，也就是说每一种 `ClusterInvoker` 实现虽然本身不是单例的，但支持服务级别的全局单例，只有明白这个特性才能更清楚的理解代码中出现的粘滞是咋回事。

《Dubbo之SPI扩展点加载》曾剖析，Dubbo中的扩展点具类都是单例的，因而可以认为Dubbo的SPI机制是另一种形式的单例的创建模式，但不能直接作用在 `ClusterInvoker` 上，因而专门提炼出了如下的 `Cluster` 扩展点，经由其 `join()` 为每一种引用微服务创建一个 `ClusterInvoker` 实例：

```
@SPI(FailoverCluster.NAME)
public interface Cluster {

    //Merge the directory invokers to a virtual invoker.
    @Adaptive
    <T> Invoker<T> join(Directory<T> directory) throws RpcException;

}
```

JAVA

注：代码中的SPI相关声明可以看出，Dubbo默认选用的是 Failover 这种容错模式。

另外针对每一种容错策略的实现，均会有类似如下 ClusterInvoker 的创建类，代码几乎一样：

```
public class FailoverCluster implements Cluster {

    public final static String NAME = "failover";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new FailoverClusterInvoker<T>(directory);
    }

}
```

JAVA

使用这种创建型模式的好处，是将具体的实现和用户接口层面解耦，将结构的复杂性隐藏在内部。当然上述关于服务级别的全局单例的表述是有问题的，类似容错这种领域通用机制在设计之初就会当做一个扩展点来实现，但以 ClusterInvoker 为中心，换了一个不同的视觉来看待 Cluster，对Dubbo的 SPI 机制会有一个更加全面的理解。

以上述源码为例，由于 FailoverCluster 是单例的，而创建 FailoverClusterInvoker 对象的 join 方法基本只在一个微服务被引用的时候才会被调用一回，因而 ClusterInvoker 总体而言于某个特定微服务是全局单例的，也就是说一个微服务会对应的一个 ClusterInvoker 实例。

## 父类 AbstractClusterInvoker

“ Merge the directory invokers to a virtual invoker.

这个表述来自接口 Cluster，Dubbo将 ClusterInvoker 视作一个虚拟的微服务引用实例，原因经过框架层层封装处理后，原本一个微服务同时有多个实例在后台默默地为应用提供服务，但上层开发人员的视线里只会看到一个 invoker 引用实例。

另一层意思很明显，所有这些背后在默默提供服务的实例来自 directory 目录服务这个组件，它负责了感知来自配置中心的变化，确保新增或者掉线的实例最终会体现在 ClusterInvoker 这个虚拟的 invoker 引用实例上。

## 生命周期管理

`ClusterInvoker` 是重依赖于 `Directory` 的，他们都实现了 `Node` 接口，一些表征微服务的关键信息也来源于后者，甚至有关生命周期的管理行为和后者密切相关，如下：

public abstract class AbstractClusterInvoker<T> implements Invoker<T> {

JAVA

```
protected final Directory<T> directory;
protected final boolean availablecheck;
private AtomicBoolean destroyed = new AtomicBoolean(false);
private volatile Invoker<T> stickyInvoker = null;
public AbstractClusterInvoker(Directory<T> directory) {
    this(directory, directory.getUrl());
}
public AbstractClusterInvoker(Directory<T> directory, URL url) {
    if (directory == null) {
        throw new IllegalArgumentException("service directory == null");
    }
    this.directory = directory;
    //sticky: invoker.isAvailable() should always be checked before using when
availablecheck is true.
    this.availablecheck = url.getParameter(CLUSTER_AVAILABLE_CHECK_KEY,
DEFAULT_CLUSTER_AVAILABLE_CHECK);
}
@Override
public Class<T> getInterface() {
    return directory.getInterface();
}

//=====
// Node接口实现
//=====

@Override
public URL getUrl() {
    return directory.getUrl();
}

@Override
public boolean isAvailable() {
    Invoker<T> invoker = stickyInvoker;
    if (invoker != null) {
        return invoker.isAvailable();
    }
    return directory.isAvailable();
}

@Override
public void destroy() {
    if (destroyed.compareAndSet(false, true)) {
        directory.destroy();
    }
}
```

```

protected void checkWhetherDestroyed() {
    if (destroyed.get()) {
        throw new RpcException("Rpc cluster invoker for " + getInterface() + " on
consumer " + NetUtils.getLocalHost()
            + " use dubbo version " + Version.getVersion()
            + " is now destroyed! Can not invoke any more.");
    }
}
...
}

```

和其它类型的 `Invoker` 一样，`ClusterInvoker` 也是工作在并发场景中，对象的销毁只能发生一次，因此加入了 `AtomicBoolean` 类型的 `destroyed` 属性。一旦被 `destroyed` 的实例便不能再处理服务，需要抛错处理，如 `checkWhetherDestroyed()` 方法所示。

另外还有两个比较关键的属性，此处有必要提前介绍下，它们会穿梭于本章节的其他后面源码中：

1. `availablecheck`：如果没有配置如下参数，发送RPC请求给一个服务实例时将不管是它否处于可用状态：
  - "cluster.availablecheck" : true | false; 默认值  
DEFAULT\_CLUSTER\_AVAILABLE\_CHECK=true;
2. `stickyInvoker`：基于 `ClusterInvoker` 的单例模式，实现微服务的粘滞特性；

## 主体流程

`ClusterInvoker` 的主体逻辑都围绕着其实现接口 `Invoker` 定义的 `invoke()` 方法展开，父类将基础的公共逻辑封装起来，再定义一个抽象方法，由子类去覆写实现，这在Dubbo是很常用的一种手法。

具体实现源码中，逻辑很清晰，分为如下几步：

1. 首先检验当前虚拟服务实例是否可用；
2. 然后将当前线程中的附属参数设给RPC方法的入参 `invocation`，另外如果结合配置总线中传入的参数等判断当前RPC方式是否被异步调用，如果是则分配一个全局唯一的ID编号，具体参考《Dubbo RPC 之 Protocol协议层（一）》中的相关章节；
3. 紧接着使用目录服务 `directory` 导出所有可用的服务实例；
4. 最后根据总线配置获取负载均衡策略默认加权随机，传入给当前类中定义的 `doInvoke()` 方法完成RPC方法的调用。

```

public abstract class AbstractClusterInvoker<T> implements Invoker<T> {

    protected abstract Result doInvoke(Invocation invocation, List<Invoker<T>>
invokers,
                                         LoadBalance loadbalance) throws RpcException;

    @Override
    public Result invoke(final Invocation invocation) throws RpcException {
        checkWhetherDestroyed();

        // binding attachments into invocation.
        Map<String, String> contextAttachments =
RpcContext.getContext().getAttachments();
        if (contextAttachments != null && contextAttachments.size() != 0) {
            ((RpcInvocation) invocation).addAttachments(contextAttachments);
        }

        List<Invoker<T>> invokers = list(invocation);
        LoadBalance loadbalance = initLoadBalance(invokers, invocation);
        RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
        return doInvoke(invocation, invokers, loadbalance);
    }

    protected List<Invoker<T>> list(Invocation invocation) throws RpcException {
        return directory.list(invocation);
    }

    protected LoadBalance initLoadBalance(List<Invoker<T>> invokers, Invocation
invocation) {
        if (CollectionUtils.isNotEmpty(invokers)) {
            return ExtensionLoader.getExtensionLoader(LoadBalance.class)
                .getExtension(invokers.get(0).getUrl()
                    .getMethodParameter(RpcUtils.getMethodName(invocation),
LOADBALANCE_KEY, DEFAULT_LOADBALANCE));
        } else {
            return ExtensionLoader.getExtensionLoader(LoadBalance.class)
                .getExtension(DEFAULT_LOADBALANCE);
        }
    }
    ...
}

```

## 公共逻辑 select()

`AbstractClusterInvoker` 有好几个扩展实现，子类的主要职责是处理容错，错误发生前后如何从多个服务实例中挑选到合适的一个，这业务逻辑所有子类都是一致的。

先看看由父类提供给子类调用的 `select()` 方法，其定义如下：

```

protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation,
List<Invoker<T>> invokers, List<Invoker<T>> selected) throws RpcException{
    ...
}

```

从其所有入参看，和抽象方法 `doInvoke()` 相比多出一个 `List<Invoker<T>> selected` 入参，也就是说逻辑转入到 `doInvoke()` 后，由子类在执行其策略的相关的业务时，使用相同的参数调用 `select()` 完成目标 `Invoker` 的选取操作。根据仔细查看源码，`selected` 中盛装的服务实例实际是要备当前的方法调用所被排除的。

`select()` 方法执行的总体逻辑操作如下：

1. 粘滞处理；
2. 使用负载均衡策略完成目标 `Invoker` 的挑选处理；
3. 如果被选到的服务实例不满足要求，调用 `reselect()` 方法做重选处理；

### 粘滞处理

在微服务开发中，总有些服务的实现是没法完全做到幂等的，前一个RPC方法调用和后一个有着某种关系，需要落实到同一个服务实例上。因此包括Dubbo在内的很多微服务框架都实现了粘滞特性，实际上负载均衡中的一致性hash策略也是一种粘滞手段，不同的是它完成的客户端到服务端的一对一隐式绑定，而 `ClusterInvoker` 实现的是前后两个RPC方法的粘滞。

上文已经提到过，`ClusterInvoker` 本身的单例特性是其实现粘滞的前提。

```

protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation,
                           List<Invoker<T>> invokers, List<Invoker<T>> selected)
throws RpcException {

    if (CollectionUtils.isEmpty(invokers)) {
        return null;
    }
    String methodName = invocation == null ? StringUtils.EMPTY :
invocation.getMethodName();

    boolean sticky = invokers.get(0).getUrl()
        .getMethodParameter(methodName, CLUSTER_STICKY_KEY,
DEFAULT_CLUSTER_STICKY);

    //若粘滞对象并不包含在invokers被选范围则抹掉上一次的粘滞记忆
    if (stickyInvoker != null && !invokers.contains(stickyInvoker)) {
        stickyInvoker = null;
    }
    //ignore concurrency problem
    if (sticky && stickyInvoker != null && (selected == null ||
selected.contains(stickyInvoker))) {
        if (availablecheck && stickyInvoker.isAvailable()) {
            return stickyInvoker;
        }
    }
}

Invoker<T> invoker = doSelect(loadbalance, invocation, invokers, selected);

//在配置需要粘滞的情况下，需要为下一次RPC方法调用记忆stickyInvoker
if (sticky) {
    stickyInvoker = invoker;
}
return invoker;
}

```

如上述源码，一进入方法，就先获取配置项 `invoacation[methodName] + ".sticky"` 的值，如果上一个处理RPC方法 `stickyInvoker` 不在被排除集合中，并且它处于可用状态，则 `stickyInvoker` 就是当前RPC方法选中的服务实例。否则需要进入下一步获得执行RPC方法的 `Invoker` 实例，方法返回之前，将被选得的 `Invoker` 实例赋值给 `stickyInvoker`，为下一次RPC方法保留记忆。

## doSelect() 服务实例选取

服务实例 `Invoker` 对象的选取操作是由 `LoadBalance` 在给定的集合中使用特定算法策略完成的。然而如果被选中的实例 `R` 如果在被排除的集合中，或者处于不可用态，就需要执行 `reselect()` 重选逻辑。倘若重选也没有获得一个合适的返回 `null` 空值，`ClusterInvoker` 就会在集合中选择排在 `R` 的下一个或集合中首个位置的实例。过程中若出现异常，`doSelect()` 方法均给调用方返回 `R`。

JAVA

```

private Invoker<T> doSelect(LoadBalance loadbalance, Invocation invocation,
                           List<Invoker<T>> invokers, List<Invoker<T>> selected)
throws RpcException {

    if (CollectionUtils.isEmpty(invokers)) {//集合为空时返回null
        return null;
    }
    if (invokers.size() == 1) {//仅有一个实例时直接返回首个
        return invokers.get(0);
    }
    Invoker<T> invoker = loadbalance.select(invokers, getUrl(), invocation);

    if ((selected != null && selected.contains(invoker))
        || (!invoker.isAvailable() && getUrl() != null && availablecheck)) {
        try {
            Invoker<T> rInvoker = reselect(loadbalance, invocation, invokers, selected,
availablecheck);
            if (rInvoker != null) {
                invoker = rInvoker;
            } else {
                int index = invokers.indexOf(invoker);
                try {
                    //Avoid collision
                    invoker = invokers.get((index + 1) % invokers.size());
                } catch (Exception e) {
                    //执行到这里表示invokers集合突然发生变化了，则直接返回loadbalance计算得到的实
例
                    logger.warn(e.getMessage() + " may because invokers list dynamic
change, ignore.", e);
                }
            }
        } catch (Throwable t) {
            logger.error("cluster reselect fail reason is :" + t.getMessage() +
                        " if can not solve, you can set cluster.availablecheck=false in url",
t);
        }
    }
    return invoker;
}

```

源码中的可用状态监测为啥不在 `loadbalance.select()` 之前就赛选掉已经实例 `unavailable` 的了，初步看起来有点费解。微服务环境中，服务实例因上下线处于不可用状态是高概率事件，更不用说他们所处的包括网络在内的环境各异，导致随时断线或者无法响应。

## `reselect()` 重选操作

负载均衡辛苦挑选到的微服务引用实例却不可用，这时 `ClusterInvoker` 只能使用 `reselect()` 执行重选处理。整个重选逻辑相对比较简单：

1. 首先从给定集合中剔除掉如下两种情况的实例：

- a. 不可用的；

b. 在被排除列表中的；

2. 根据剔除后的集合是否为空执行如下操作：

a. 不为空，调用 `loadbalance.select()` 选一个；

b. 为空，则对被排除列表中的可用服务实例集合做负载处理，该过程若没找到合适的则直接返回 `null`；

```
private Invoker<T> reselect(LoadBalance loadbalance, Invocation invocation,
    List<Invoker<T>> invokers, List<Invoker<T>> selected, boolean availablecheck)
throws RpcException {

    // Allocating one in advance, this list is certain to be used.
    List<Invoker<T>> reselectInvokers = new ArrayList<>(
        invokers.size() > 1 ? (invokers.size() - 1) : invokers.size());

    // First, try picking a invoker not in `selected`.
    for (Invoker<T> invoker : invokers) {
        if (availablecheck && !invoker.isAvailable()) {
            continue;
        }

        if (selected == null || !selected.contains(invoker)) {
            reselectInvokers.add(invoker);
        }
    }

    if (!reselectInvokers.isEmpty()) {
        return loadbalance.select(reselectInvokers, getUrl(), invocation);
    }

    // Just pick an available invoker using loadbalance policy
    if (selected != null) {
        for (Invoker<T> invoker : selected) {
            if ((invoker.isAvailable() // available first
                && !reselectInvokers.contains(invoker))) {
                reselectInvokers.add(invoker);
            }
        }
    }

    if (!reselectInvokers.isEmpty()) {
        return loadbalance.select(reselectInvokers, getUrl(), invocation);
    }

    return null;
}
```

## ClusterInvoker 实现

在 `dubbo-cluster` 这个模块中的 `META-INF/dubbo/internal` 目录下存在一个名为 `org.apache.dubbo.rpc.cluster.Cluster` 的 SPI 配置文件，其中配置如下，也就是说容错策略的实现远超文首提到的那几种。

mock=org.apache.dubbo.rpc.cluster.support.wrapper.MockClusterWrapper  
failover=org.apache.dubbo.rpc.cluster.support.FailoverCluster  
failfast=org.apache.dubbo.rpc.cluster.support.FailfastCluster  
failsafe=org.apache.dubbo.rpc.cluster.support.FailsafeCluster  
fallback=org.apache.dubbo.rpc.cluster.support.FallbackCluster  
forking=org.apache.dubbo.rpc.cluster.support.ForkingCluster  
available=org.apache.dubbo.rpc.cluster.support.AvailableCluster  
mergeable=org.apache.dubbo.rpc.cluster.support.MergeableCluster  
Broadcast=org.apache.dubbo.rpc.cluster.support.BroadcastCluster  
registryaware=org.apache.dubbo.rpc.cluster.support.RegistryAwareCluster

TEXT

不过在继续看具体的策略实现前，还得先回到 `AbstractClusterInvoker` 抽象类来看看上文没有涉及到的 `checkInvokers()` 方法，如下：

```
protected void checkInvokers(List<Invoker<T>> invokers, Invocation invocation) {  
    if (CollectionUtils.isEmpty(invokers)) {  
        throw new RpcException(RpcException.NO_INVOKER_AVAILABLE_AFTER_FILTER,  
            "Failed to invoke the method "  
            + invocation.getMethodName() + " in the service " +  
getInterface().getName()  
            + ". No provider available for the service " +  
directory.getUrl().getServiceKey()  
            + " from registry " + directory.getUrl().getAddress()  
            + " on the consumer " + NetUtils.getLocalHost()  
            + " using the dubbo version " + Version.getVersion()  
            + ". Please check if the providers have been started and registered.");  
    }  
}
```

## NOTE

关于用于实现本地伪装的装饰类 `MockClusterWrapper`，请挪步《Dubbo服务降级》一文。

## Available

这是一种最简单的实现，每次RPC请求进入，什么负载均衡、粘滞啥的统统不要，挨个迭代候选集中所有的实例，遇到的第一个可用实例便执行最终RPC方法调用并返回：

```

public class AvailableClusterInvoker<T> extends AbstractClusterInvoker<T> {

    public AvailableClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                return invoker.invoke(invocation);
            }
        }
        throw new RpcException("No provider available in " + invokers);
    }

}

```

## RegistryAware

相比于 Available 类型的容错策略，该策略实现分为两部分，后面部分和 Available 完全相同，前面部分是挑到首个含有配置项 "registry.default" 为true的实例做RPC调用。

```

public class RegistryAwareClusterInvoker<T> extends AbstractClusterInvoker<T> { JAVA

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

    public RegistryAwareClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        // First, pick the invoker (XXXClusterInvoker) that comes from the local
registry, distinguish by a 'default' key.
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable() && invoker.getUrl().getParameter(REGISTRY_KEY +
"." + DEFAULT_KEY, false)) {
                return invoker.invoke(invocation);
            }
        }
        // If none of the invokers has a local signal, pick the first one available.
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                return invoker.invoke(invocation);
            }
        }
        throw new RpcException("No provider available in " + invokers);
    }

}

```

## Broadcast (广播调用)

“ 广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

实现也很简单，先检查候选集是否存在可用的微服务引用实例，然后将候选集设入到本地异步上下文中，最后遍历候选集中所有的微服务引用实例，每个实例调用一次RPC方法，每次RPC方法调用时，临时变量 `result` 和 `exception` 分别记录正常和异常结果，方法执行到最后，如果 `exception` 不为空，则返回异常，否则返回 `result`，如下所示：

```
public class BroadcastClusterInvoker<T> extends AbstractClusterInvoker<T> {  
  
    private static final Logger logger =  
        LoggerFactory.getLogger(BroadcastClusterInvoker.class);  
  
    public BroadcastClusterInvoker(Directory<T> directory) {  
        super(directory);  
    }  
  
    @Override  
    @SuppressWarnings({"unchecked", "rawtypes"})  
    public Result doInvoke(final Invocation invocation,  
        List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  
        checkInvokers(invokers, invocation);  
        RpcContext.getContext().setInvokers((List) invokers);  
        RpcException exception = null;  
        Result result = null;  
        for (Invoker<T> invoker : invokers) {  
            try {  
                result = invoker.invoke(invocation);  
            } catch (RpcException e) {  
                exception = e;  
                logger.warn(e.getMessage(), e);  
            } catch (Throwable e) {  
                exception = new RpcException(e.getMessage(), e);  
                logger.warn(e.getMessage(), e);  
            }  
        }  
        if (exception != null) {  
            throw exception;  
        }  
        return result;  
    }  
}
```

## Failfast (快速失败)

“ 快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

和上述容错策略不同的是，该策略中会调用父类的 `select()` 方法执行诸如粘滞、负载均衡、重选处理，挑选到目标实例后便执行RPC方法调用，如果 `catch` 到异常，则转换为 `RpcException` 向上层抛出。

```
public class FailfastClusterInvoker<T> extends AbstractClusterInvoker<T> {  
  
    public FailfastClusterInvoker(Directory<T> directory) {  
        super(directory);  
    }  
  
    @Override  
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,  
LoadBalance loadbalance) throws RpcException {  
        checkInvokers(invokers, invocation);  
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);  
        try {  
            return invoker.invoke(invocation);  
        } catch (Throwable e) {  
            if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz  
exception.  
                throw (RpcException) e;  
            }  
            throw new RpcException(e instanceof RpcException ?  
                ((RpcException) e).getCode() : 0,  
                "Failfast invoke providers " + invoker.getUrl()  
                + " " + loadbalance.getClass().getSimpleName()  
                + " select from all providers " + invokers  
                + " for service " + getInterface().getName()  
                + " method " + invocation.getMethodName()  
                + " on consumer " + NetUtils.getLocalHost()  
                + " use dubbo version " + Version.getVersion()  
                + ", but no luck to perform the invocation. Last error is: " +  
e.getMessage(),  
                e.getCause() != null ? e.getCause() : e);  
        }  
    }  
}
```

## Failsafe (失败安全)

“ 失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

和 `Failfast` 容错策略唯一不同的在于异常处理，`catch` 到异常时，只是在日志中简单记录异常信息，并返回一个完成态的的 `Result` 对象。

```

public class FailsafeClusterInvoker<T> extends AbstractClusterInvoker<T> {
    private static final Logger logger =
LoggerFactory.getLogger(FailsafeClusterInvoker.class);

    public FailsafeClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        try {
            checkInvokers(invokers, invocation);
            Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Failsafe ignore exception: " + e.getMessage(), e);
            return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); // ignore
        }
    }
}

```

JAVA

## Failover (失败自动切换)

“失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。

相比而言，实现稍微复杂点，总体步骤如下：

1. 方法刚进入时，先检查候选列表的可用性，随后从配置中心目录服务的配置总线中以配置项 `url[methodName] + ".retries"` 获取重试次数 `len`；
2. 声明临时变量：1) `le` 记录最后一次执行遇到的异常；2) `invoked` 记录已经执行过RPC调用的服务实例；3) `providers` 记录已经被重试过的其它实例所在的机器的Ip地址信息；
3. 进入循环执行如下逻辑处理，循环最多迭代 `len` 次：
  - a. 为避免可用候选集发生变化，重新执行相关的前置处理
    - i. 检查目录服务是否下线；
    - ii. 调用 `list()` 方法列出所有可用的服务引用实例；
    - iii. 检查候选列表的可用性；
  - b. 使用最新的候选集，以及 `invoked` 等调用 `select()` 方法获取到一个可用的目标微服务引用实例 I；
  - c. 将 I 加入到 `invoked`，同时在 `finally` 块中将 I 的 Ip 地址信息加入到 `providers` 中，并将已经发生变化的 `invoked` 设入到当前本地线程上下文中；

- d. 使用 I 执行RPC远程方法调用；
  - e. 如果 RPC 方法调用成功直接返回结果，否则如果出现业务类异常，则直接抛异常处理，其它类型异常则记录到 le，并进入下一循环，继续流程；
4. 循环执行完，使用 le 中的信息抛异常处理；

```

public class FailoverClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

    public FailoverClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        List<Invoker<T>> copyInvokers = invokers;
        checkInvokers(copyInvokers, invocation);
        String methodName = RpcUtils.getMethodName(invocation);
        int len = getUrl().getMethodParameter(methodName, RETRIES_KEY, DEFAULT_RETRIES)
+ 1;
        if (len <= 0) {
            len = 1;
        }
        // retry loop.
        RpcException le = null; // last exception.
        List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyInvokers.size()); // invoked invokers.
        Set<String> providers = new HashSet<String>(len);
        for (int i = 0; i < len; i++) {
            //Reselect before retry to avoid a change of candidate `invokers`.
            //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
            if (i > 0) {
                checkWhetherDestroyed();
                copyInvokers = list(invocation);
                // check again
                checkInvokers(copyInvokers, invocation);
            }
            Invoker<T> invoker = select(loadbalance, invocation, copyInvokers,
invoked);
            invoked.add(invoker);
            RpcContext.getContext().setInvokers((List) invoked);
            try {
                Result result = invoker.invoke(invocation);
                if (le != null && logger.isWarnEnabled()) {
                    logger.warn("Although retry the method " + methodName
                        + " in the service " + getInterface().getName()
                        + " was successful by the provider " +
invoker.getUrl().getAddress()
                        + ", but there have been failed providers " + providers
                        + " (" + providers.size() + "/" + copyInvokers.size()
                        + ") from the registry " + directory.getUrl().getAddress()
                        + " on the consumer " + NetUtils.getLocalHost()
                        + " using the dubbo version " + Version.getVersion() + ".
Last error is: "
                        + le.getMessage(), le);
                }
                return result;
            } catch (RpcException e) {

```

JAVA

```

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

```

## Forking (并行调用)

“并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

从描述中不难看出，多个RPC请求同时发出，一旦获取首个成功返回的结果便完成了整个RPC调用，否则以为所有的RPC请求均以失败告终。

实现上稍显复杂，我们将对源码打散处理，逐段分析。

配置中心可以为该策略设置 并行数-forks、超时-timeout 参数，默认值分别为 2 和 1000ms。在具体处理时，如果配置值大于候选集可用个数，则将所有服务实例都加入到 selected 中，否则挨个调用 select() 方法，将挑选到的实例加入到 selected，直到达到并发数目为止。

```

//①并行环境准备
checkInvokers(invokers, invocation);
final List<Invoker<T>> selected;
final int forks = getUrl().getParameter(FORKS_KEY, DEFAULT_FORKS);
final int timeout = getUrl().getParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
if (forks <= 0 || forks >= invokers.size()) {
    selected = invokers;
} else {
    selected = new ArrayList<>();
    for (int i = 0; i < forks; i++) {
        Invoker<T> invoker = select(loadbalance, invocation, invokers, selected);
        if (!selected.contains(invoker)) {
            //Avoid add the same invoker several times.
            selected.add(invoker);
        }
    }
}
RpcContext.getContext().setInvokers((List) selected);

```

JAVA

虽然默认上，每个微服务引用实例在处理RPC请求时，均以异步方式调用，但 ForkingClusterInvoker 还是采用了和其它策略迥异的实现方式，使用了在异步编程做任务调度时常用的 BlockingQueue，专门开辟一个线程池给自己处理并行的RPC请求。由异步线程负责处理请求，请求的结果塞入阻塞队列，当前线程从队列取得元素，典型的生产者-消费者模型。

另外还增加了一个 AtomicInteger 类型的原子变量 count，如果异步线程中 catch 到异常，便计数加一，一旦该计数值为前述 selected 的大小时，便说明发出的所有并行RPC请求失败。

```

//②生产环节
final AtomicInteger count = new AtomicInteger();
final BlockingQueue<Object> ref = new LinkedBlockingQueue<>();
for (final Invoker<T> invoker : selected) {
    executor.execute(() -> {
        try {
            Result result = invoker.invoke(invocation);
            ref.offer(result);
        } catch (Throwable e) {
            int value = count.incrementAndGet();
            if (value >= selected.size()) {
                ref.offer(e);
            }
        }
    });
}

```

JAVA

在后面的消费环节中，当前线程利用阻塞队列同步从中同步获取到结果。若结果为 Result，直接返回，否则结果为 Throwable 类型，或者请求没被及时处理而超时均会抛出异常。

```
//③消费环节
try {
    Object ret = ref.poll(timeout, TimeUnit.MILLISECONDS);
    if (ret instanceof Throwable) {
        Throwable e = (Throwable) ret;
        throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0, "Failed to forking invoke provider " + selected + ", but no luck to perform the invocation. Last error is: " + e.getMessage(), e.getCause() != null ? e.getCause() : e);
    }
    return (Result) ret;
} catch (InterruptedException e) {
    throw new RpcException("Failed to forking invoke provider " + selected + ", but no luck to perform the invocation. Last error is: " + e.getMessage(), e);
}
```

最后所有的代码汇总如下：

```
public class ForkingClusterInvoker<T> extends AbstractClusterInvoker<T> {
    private final ExecutorService executor = Executors.newCachedThreadPool(
        new NamedInternalThreadFactory("forking-cluster-timer", true));

    public ForkingClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers,
        LoadBalance loadbalance) throws RpcException {
        try {
            //①并行环境准备

            //②生产环节

            //③消费环节
        } finally {
            // clear attachments which is binding to current thread.
            RpcContext.getContext().clearAttachments();
        }
    }
}
```

## Fallback (失败自动恢复)

“ 失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

从下述代码来看，`Fallback` 容错策略的实现和 `Failsafe` 基本无甚差异，出现异常时，返回一个完成态的 `Result` 给调用方。

```

public class FallbackClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

    @Override
    protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        Invoker<T> invoker = null;
        try {
            checkInvokers(invokers, invocation);
            invoker = select(loadbalance, invocation, invokers, null);
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Fallback to invoke method " + invocation.getMethodName() + ", "
wait for retry in background. Ignored exception: "
                + e.getMessage() + ", ", e);
            addFailed(loadbalance, invocation, invokers, invoker);
            return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); // ignore
        }
    }
    ...
}

```

上述代码中 `addFailed(loadbalance, invocation, invokers, invoker);` 这一行是整个 Fallback 的容错实现部分，背后牵涉内容比较多，要理解其实现，还需要先回到咱们的第一篇分析 Dubbo 源码实现的文章——《【一】定时轮算法 · HashedWheelTimer》，序列文章的起始就重点分析了 Dubbo 的定时机制是如何实现的，原因是 Dubbo 中有大量的实现需要用到定时器定期执行一些任务。

可以简单的将 定时轮 看做是 定时任务调度器 + 任务 的总和，调用方需要先实例化一个称为 `Timer` 的定时器，与此同时为其准备一个供任务调度的专用线程池，在需要的时机向其提交一个 `TimerTask` 定时任务。另外调用方自身被销毁时，也应该回收被 `Timer` 占用线程池资源。

## RetryTimerTask

重试任务是异步的，并不会影响到当前RPC调用的及时响应，因此在提交重试任务时，需要缓存那一刻的有关执行环境变量，包括RPC方法的入参、使用到的负载均衡策略、候选的微服务引用实例等。任务被调度时执行如下操作：

1. 调用 `select()` 方法从候选集中选一个服务实例重新执行RPC方法调用；
2. 若捕获到异常，只要重试次数没有超过，便在 `Timer` 还存续的基础上再次执行重试任务；

```

private class RetryTimerTask implements TimerTask {
    private final Invocation invocation;
    private final LoadBalance loadbalance;
    private final List<Invoker<T>> invokers;
    private final int retries;
    private final long tick;
    private Invoker<T> lastInvoker;
    private int retryTimes = 0;

    RetryTimerTask(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>>
invokers, Invoker<T> lastInvoker, int retries, long tick) {
        this.loadbalance = loadbalance;
        this.invocation = invocation;
        this.invokers = invokers;
        this.retries = retries;
        this.tick = tick;
        this.lastInvoker=lastInvoker;
    }

    @Override
    public void run(Timeout timeout) {
        try {
            Invoker<T> retryInvoker = select(loadbalance, invocation, invokers,
Collections.singletonList(lastInvoker));
            lastInvoker = retryInvoker;
            retryInvoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Failed retry to invoke method " + invocation.getMethodName()
+ ", waiting again.", e);
            if (++retryTimes) >= retries) {
                logger.error("Failed retry times exceed threshold (" + retries + "), We
have to abandon, invocation->" + invocation);
            } else {
                rePut(timeout);
            }
        }
    }

    private void rePut(Timeout timeout) {
        if (timeout == null) {
            return;
        }

        Timer timer = timeout.timer();
        if (timer.isStop() || timeout.isCancelled()) {
            return;
        }

        timer.newTimeout(timeout.task(), tick, TimeUnit.SECONDS);
    }
}

```

源码中，没执行一次任务，总会将执行当前RPC方法的服务实例记录在 `lastInvoker` 零时变量中，后面的任务执行就将其排除在候选集之外，避免将同一个任务反复提交给一个出现故障的实例。

## Timer 管理

如下源码所示， Timer 的采用了实例延迟初始化的方式，结合双检锁机制，确保处于并发环境下的 Failback 只会实例化一次 Timer 。 addFailed() 方法在 failTimer 已经赋值后，便实例化 RetryTimerTask 实例向其提交定时任务。

```
private static final long RETRY_FAILED_PERIOD = 5;                                JAVA

private volatile Timer failTimer;

private void addFailed(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>>
invokers, Invoker<T> lastInvoker) {
    if (failTimer == null) {
        synchronized (this) {
            if (failTimer == null) {
                failTimer = new HashedWheelTimer(
                    new NamedThreadFactory("failback-cluster-timer", true),
                    1,
                    TimeUnit.SECONDS, 32, fallbackTasks);
            }
        }
    }
    RetryTimerTask retryTimerTask = new RetryTimerTask(loadbalance, invocation,
    invokers, lastInvoker, retries, RETRY_FAILED_PERIOD);
    try {
        failTimer.newTimeout(retryTimerTask, RETRY_FAILED_PERIOD, TimeUnit.SECONDS);
    } catch (Throwable e) {
        logger.error("Failback background works error, invocation->" + invocation + ", "
exception: " + e.getMessage());
    }
}

public FallbackClusterInvoker(Directory<T> directory) {
    super(directory);

    int retriesConfig = getUrl().getParameter(RETries_KEY, DEFAULT_FAILBACK_TIMES);
    if (retriesConfig <= 0) {
        retriesConfig = DEFAULT_FAILBACK_TIMES;
    }
    int fallbackTasksConfig = getUrl().getParameter(FAIL_BACK_TASKS_KEY,
    DEFAULT_FAILBACK_TASKS);
    if (fallbackTasksConfig <= 0) {
        fallbackTasksConfig = DEFAULT_FAILBACK_TASKS;
    }
    retries = retriesConfig;
    fallbackTasks = fallbackTasksConfig;
}
```

源码的最后还将类的构造函数也呈现了，在 Directory 中可以为 Failback 配置重试次数 retries 和当前最大允许挂起的任务数 fallbacktasks 参数。

最后在 Invoker 实例被销毁时，确保 failTimer 的资源被回收处理。

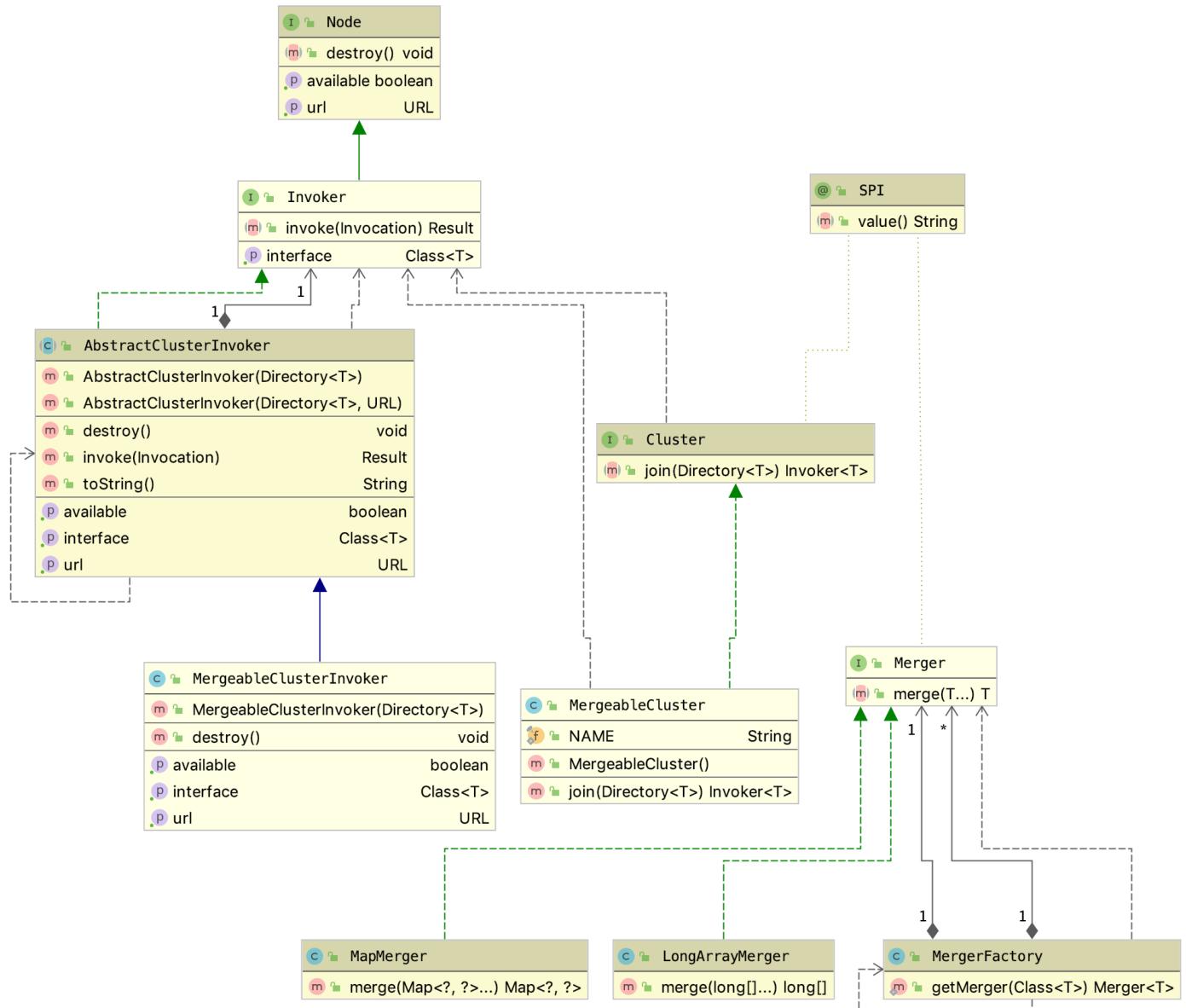
```
@Override  
public void destroy() {  
    super.destroy();  
    if (failTimer != null) {  
        failTimer.stop();  
    }  
}
```

## Mergeable (结果聚合)

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

上述官网中的这一段表述中说明一个服务接口可以存在多种实现，也即对应着多个不同的微服务，用分组来区分他们，而调用客户端可以利用集群的 Mergeable 特性，汇总每种 group 的返回结果，实现聚合。

严格来说 MergeableClusterInvoker 这个 ClusterInvoker 实现并不属于一种容错策略，只是同样由扩展 AbstractClusterInvoker 抽象类实现的一种集群特性。下图是整个实现的类图：



Powered by yFiles

## Merger 聚合器扩展点

从类图中可以看出 **Mergeable** 的实现中需要用到另外一个名为 **Merger** 的扩展点，用于将服务接口的出参做聚合处理，也即将两个集合做归并处理。Dubbo提供的默认实现有 **ArrayListMerger**、**BooleanArrayListMerger**、**ByteArrayMerger**、**CharArrayMerger**、**DoubleArrayListMerger**、**FloatArrayListMerger**、**IntArrayMerger**、**ListMerger**、**LongArrayListMerger**、**MapMerger**、**SetMerger**、**ShortArrayListMerger**。它们的实现大同小异，本文只看看 **ArrayListMerger** 实现。

public class ArrayMerger implements Merger<Object[]> { JAVA

```
    public static final ArrayMerger INSTANCE = new ArrayMerger();

    @Override
    public Object[] merge(Object[]... items) {
        if (ArrayUtils.isEmpty(items)) {
            return new Object[0];
        }

        //先统计入参数组中，元素值为null的个数
        int i = 0;
        while (i < items.length && items[i] == null) {
            i++;
        }

        //如果入参数组中的所有元素均为null，则直接返回空的数组
        if (i == items.length) {
            return new Object[0];
        }

        //流程到这里说明入参数组中存在非null的元素

        //使用第一个非null的元素获取其数组元素的类型
        Class<?> type = items[i].getClass().getComponentType();

        //检测入参数组中的所有数组元素是否类型一致，并汇总这些非null数组中的所有元素个数
        int totalLen = 0;
        for (; i < items.length; i++) {
            if (items[i] == null) {
                continue;
            }
            Class<?> itemType = items[i].getClass().getComponentType();
            if (itemType != type) {
                throw new IllegalArgumentException("Arguments' types are different");
            }
            totalLen += items[i].length;
        }

        //汇总元素个数为0，则返回空的数组
        if (totalLen == 0) {
            return new Object[0];
        }

        //使用Array的反射功能申请一个数组
        Object result = Array.newInstance(type, totalLen);

        int index = 0;
        for (Object[] array : items) {
            if (array != null) {
                for (int j = 0; j < array.length; j++) {
                    //使用Array的反射功能设置数组指定索引位置的元素
                    Array.set(result, index++, array[j]);
                }
            }
        }
    }
}
```

```
        return (Object[]) result;
    }
}
```

既然是扩展点，用户当然可以自己提供一个扩展点具类，对于数据聚合合并结果这种需求在开发中也是常见的需求，实际生产中的数据不再是简单的基础类型，有的甚至有着比较复杂的合并规则，这也意味着必须留有给开发者扩展的切入点。

## 聚合器创建工厂 MergerFactory

在 `MergeableClusterInvoker` 实现中，需要动态的根据服务接口的出参类型获取到对应的聚合器，由于和Dubbo自身实现的SPI机制密切相关，建议先看看本序列文章中的《Dubbo之SPI扩展点加载》一文。

总体实现思路是，将所有 `Merger` 扩展点具类的单例以其所实现泛型参数 `Class<?>` 装入到声明了 `static` 的 `Map<Class<?>, Merger<?>>` 容器中，在需要的时候直接使用服务接口的出参类型作为 Key 键从该容器中获得对应的 `Merger` 实例。另外总体而言Dubbo是很注重节约资源的，不会应用一初始化就做SPI的加载完成容器的填充，毕竟单就 `Merger` 而言，应用中可能压根没有用到结果聚合这一特性，预先加载就意味着内存资源的浪费，因而源码实现中采用了延迟加载方案。

在具体实现中，容器 `Map` 的具体类型为 `ConcurrentMap<Class<?>, Merger<?>>`，原因是 `Merger` 的创建工厂 `MergerFactory` 是在 `ClusterInvoker` 所处并发环境环境下被使用。

```

public class MergerFactory {

    private static final ConcurrentHashMap<Class<?>, Merger<?>> MERGER_CACHE =
        new ConcurrentHashMap<Class<?>, Merger<?>>();

    public static <T> Merger<T> getMerger(Class<T> returnType) {
        if (returnType == null) {
            throw new IllegalArgumentException("returnType is null");
        }

        Merger result;
        if (returnType.isArray()) {
            //若返回类型为数组，则获取其元素类型
            Class type = returnType.getComponentType();
            result = MERGER_CACHE.get(type);

            //result为null可能意味着SPI未加载
            if (result == null) {
                //执行SPI加载并获取扩展点具类实例，也即一个聚合器实现
                loadMergers();
                result = MERGER_CACHE.get(type);
            }
            if (result == null && !type.isPrimitive()) {
                result = ArrayMerger.INSTANCE;
            }
        } else {
            result = MERGER_CACHE.get(returnType);
            if (result == null) {
                loadMergers();
                result = MERGER_CACHE.get(returnType);
            }
        }
        return result;
    }

    static void loadMergers() {
        //首选获取SPI配置文件配置的所有支持扩展点实现，名称到具类的映射关系
        Set<String> names = ExtensionLoader.getExtensionLoader(Merger.class)
            .getSupportedExtensions();
        for (String name : names) {
            //逐个根据扩展点名称加载具类
            Merger m =
                ExtensionLoader.getExtensionLoader(Merger.class).getExtension(name);

            //使用ReflectUtils.getGenericClass(m.getClass())获取到Merger实例所实现的泛型类型
            MERGER_CACHE.putIfAbsent(ReflectUtils.getGenericClass(m.getClass()), m);
        }
    }
}

```

## MergeableClusterInvoker

相比容错类型的 ClusterInvoker，该实现显得更加复杂，所有的代码都集中在一个方法中，但是总体步骤是比较清晰的，如下：

1. 确保有服务实例候选集可用；
2. 检查是否配置了 `url[invocation.getMethodName()] + ".merger"`：
  - a. 若无这项配置，则选用候选集中的 首个可用优先 或 首个 服务实例发起RPC调用，随即返回；
  - b. 若有，则继续下一步；
3. 遍历候选集，挨个异步发起PRC调用，并将结果装入 `Map<String, Result>` 类型的 `results` 容器；
4. 遍历 `results`，逐个同步获取 `Result` 结果，如果异常则记录日志，否则装入 `List<Result>` 类型的 `resultList` 容器中；
5. 若 `resultList` 为空 或 被调用微服务接口方法的出参为 `void`，则返回一个完成态的 `AsyncResult` 对象，若只有一个元素直接返回，否则继续下一步；
6. 遍历 `resultList` 做结果聚合处理；

实现细节比较丰富，按照以往惯例，我们下面一段段打散加以分析。

第一段代码集中在 1、2 这两个步骤，不过在抛错处理时的

`e.isNoInvokerAvailableAfterFilter()` 这句代码值得注意，被调用方法中使用的异常标识 `NO_INVOKER_AVAILABLE_AFTER_FILTER` 值为6只在 `ClusterInvoker` 中用到，这说明被引用的微服务实例很有可能是其它虚拟 `Invoker`，也即 `ClusterInvoker` 实例。

```
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
    LoadBalance loadbalance) throws RpcException {
    checkInvokers(invokers, invocation);
    String merger = getUrl().getMethodParameter(invocation.getMethodName(),
        MERGER_KEY);
    if (ConfigUtils.isEmpty(merger)) { // If a method doesn't have a merger, only
    invoke one Group
        for (final Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                try {
                    return invoker.invoke(invocation);
                } catch (RpcException e) {
                    if (e.isNoInvokerAvailableAfterFilter()) {
                        log.debug("No available provider for service" +
directory.getUrl().getServiceKey() + " on group " +
invoker.getUrl().getParameter(GROUP_KEY) + ", will continue to try another group.");
                    } else {
                        throw e;
                    }
                }
            }
        }
    }
    return invokers.iterator().next().invoke(invocation);
}
...
}
```

JAVA

第二段代码对应步骤 3、4、5。在遍历所有服务实例候选集将 `Result` 结果装入到 `results` 时，会将 RPC 调用入参 `Invocation` 另行复制一份，拷入服务实例的配置后，并通过标记 `async` 告知 Dubbo 异步执行接下来的 RPC 调用。但是仔细分析代码发现会存在一个问题，我们都知道一个微服务存在的多个实例对应的是同一个 `ServiceKey {group}/{interfaceName}:{version}`，代码中的 `put` 操作将会导致同一个微服务的所有在候选集中的实例都异步执行了，但只有最后一个实例的结果会被 `catch` 到。

JAVA

```

protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
    LoadBalance loadbalance) throws RpcException {
    ...
    Map<String, Result> results = new HashMap<>();
    for (final Invoker<T> invoker : invokers) {
        RpcInvocation subInvocation = new RpcInvocation(invocation, invoker);
        subInvocation.setAttachment(ASYNC_KEY, "true");
        results.put(invoker.getUrl().getServiceKey(), invoker.invoke(subInvocation));
    }

    List<Result> resultList = new ArrayList<Result>(results.size());

    for (Map.Entry<String, Result> entry : results.entrySet()) {
        Result asyncResult = entry.getValue();
        try {
            Result r = asyncResult.get();
            if (r.hasException()) {
                log.error("Invoke " + getGroupDescFromServiceKey(entry.getKey()) +
                    " failed: " + r.getException().getMessage(),
                    r.getException());
            } else {
                resultList.add(r);
            }
        } catch (Exception e) {
            throw new RpcException("Failed to invoke service " + entry.getKey() + ":" +
+ e.getMessage(), e);
        }
    }

    if (resultList.isEmpty()) {
        return AsyncRpcResult.newDefaultAsyncResult(invocation);
    } else if (resultList.size() == 1) {
        return resultList.iterator().next();
    }

    Class<?> returnType;
    try {
        returnType = getInterface().getMethod(
            invocation.getMethodName(),
        invocation.getParameterTypes().getReturnType());
    } catch (NoSuchMethodException e) {
        returnType = null;
    }

    if (returnType == void.class) {
        return AsyncRpcResult.newDefaultAsyncResult(invocation);
    }
    ...
}

```

剩下的最后一段代码所涉细节比较多，相较也难以理解，根据得到 merge 配置值分为两段。

有时一个服务接口的方法出参，其类型并非一个数组，而是一个其它聚合，比如 `List`，这类聚合有个特点，就是可以将其它同类型对象中的所有聚集体元素加入自身或新建的这个集合来，比如 `List` 的 `addAll` 方法，这时就可以做如下配置 `".+methodName"`，Dubbo会采用递归的方式将上述 `resultList` 列表中的所有元素平摊汇总到一块：

```
<dubbo:reference interface="com.xxx.MenuService" group="*">>
    <dubbo:method name="getMenuItems" merger=".addAll" />
</dubbo:reference>
```

XML

如下源码所示，这类型的方法的一个典型特征是，它只有一个入参，入参的类型和方法所属接口或类的类型一致；出参则分为两种情况，一种是 `void` 型的，另一种是和入参类型一致的，前一种表示合入参数中的集合，而后一种则是新建一个集合，将自身和参数中的集合一起合入。

JAVA

```

protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
    LoadBalance loadbalance) throws RpcException {
    ...
    Object result = null;
    if (merger.startsWith(".")) {
        merger = merger.substring(1);
        Method method;
        try {
            //获取到方法名称
            method = returnType.getMethod(merger, returnType);
        } catch (NoSuchMethodException e) {
            throw new RpcException("Can not merge result because missing method [ " +
                merger + " ] in class [ " +
                    returnType.getClass().getName() + " ]");
        }
        //不要求配置方法是public的, private的方法使用反射设置可访问
        if (!Modifier.isPublic(method.getModifiers())) {
            method.setAccessible(true);
        }
    }

    //取出第一个元素, 预备后续元素在第一个基础上进行合并处理
    result = resultList.remove(0).getValue();
    try {
        if (method.getReturnType() != void.class
            && method.getReturnType().isAssignableFrom(result.getClass())) {
            //返回类型不为void的情况, result指针不断变化, 每次得到一个新的集合对象,
            //当前的所有元素汇总了每一次迭代的
            for (Result r : resultList) {
                result = method.invoke(result, r.getValue());
            }
        } else {
            //将resultList中的所有集合中元素都汇总到最初的那个result上
            for (Result r : resultList) {
                method.invoke(result, r.getValue());
            }
        }
    } catch (Exception e) {
        throw new RpcException("Can not merge result: " + e.getMessage(), e);
    }
} else {...}
...
}

```

在使用结果聚合特性时, 经常会有如下配置:

XML

```
<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb" merger="true" />
```

其中 merge 的值可以配置成 true 或者 default, 表示使用 MergerFactory 这个聚合器创建工厂根据服务接口方法的返回参数的类型获取到对应的聚合器。明确指定名称则使用 SPI 机制根据配置名获取对应的聚合器已缓存。

```
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
    LoadBalance loadbalance) throws RpcException {
    ...
    Object result = null;
    if (merger.startsWith(".")) {...}
    else {
        Merger resultMerger;
        if (ConfigUtils.isDefault(merger)) {
            resultMerger = MergerFactory.getMerger(returnType);
        } else {
            resultMerger =
                ExtensionLoader.getExtensionLoader(Merger.class).getExtension(merger);
        }
        if (resultMerger != null) {
            List<Object> rets = new ArrayList<Object>(resultList.size());
            for (Result r : resultList) {
                rets.add(r.getValue());
            }
            result = resultMerger.merge(
                rets.toArray((Object[]) Array.newInstance(returnType, 0)));
        } else {
            throw new RpcException("There is no merger to merge result.");
        }
    }
    return AsyncRpcResult.newDefaultAsyncResult(result, invocation);
}
```

JAVA

---

完结

# Dubbo服务降级

服务调用通常是跨进程的，中间复杂的网络环境，加上远端的业务代码又不受客户端开发人员控制，发生异常几乎是必然的，只是概率分布的高低而已。因此一个合理的微服务接入开发，非常有必要对被引用微服务调用过程预置异常处理逻辑，也即客户端根据业务需要所做的服务降级。

Dubbo中的服务降级实现主要是通过 Mock本地伪装 机制实现的。

## Mock 本地伪装

**NOTE** **Mock** 本地伪装机制的实现借助了集群组块的中的路由和容错，若不熟悉这二者的话，请挪步《Dubbo集群之容错》和《Dubbo集群之路由》一探究竟。

Dubbo使用了 **Mock** 本地伪装机制实现了服务降级，从表象来看 **本地伪装** 是 **本地存根** 的一个子集，但实现方式却全然不同，关于后者具体请参考《Dubbo服务代理》一文。**本地伪装** 是面向除 **MockClusterWrapper** 外的所有 **Cluster** 实现的（**MockClusterWrapper** 具类是 **Cluster** 扩展点的装饰实现），大致思路是：

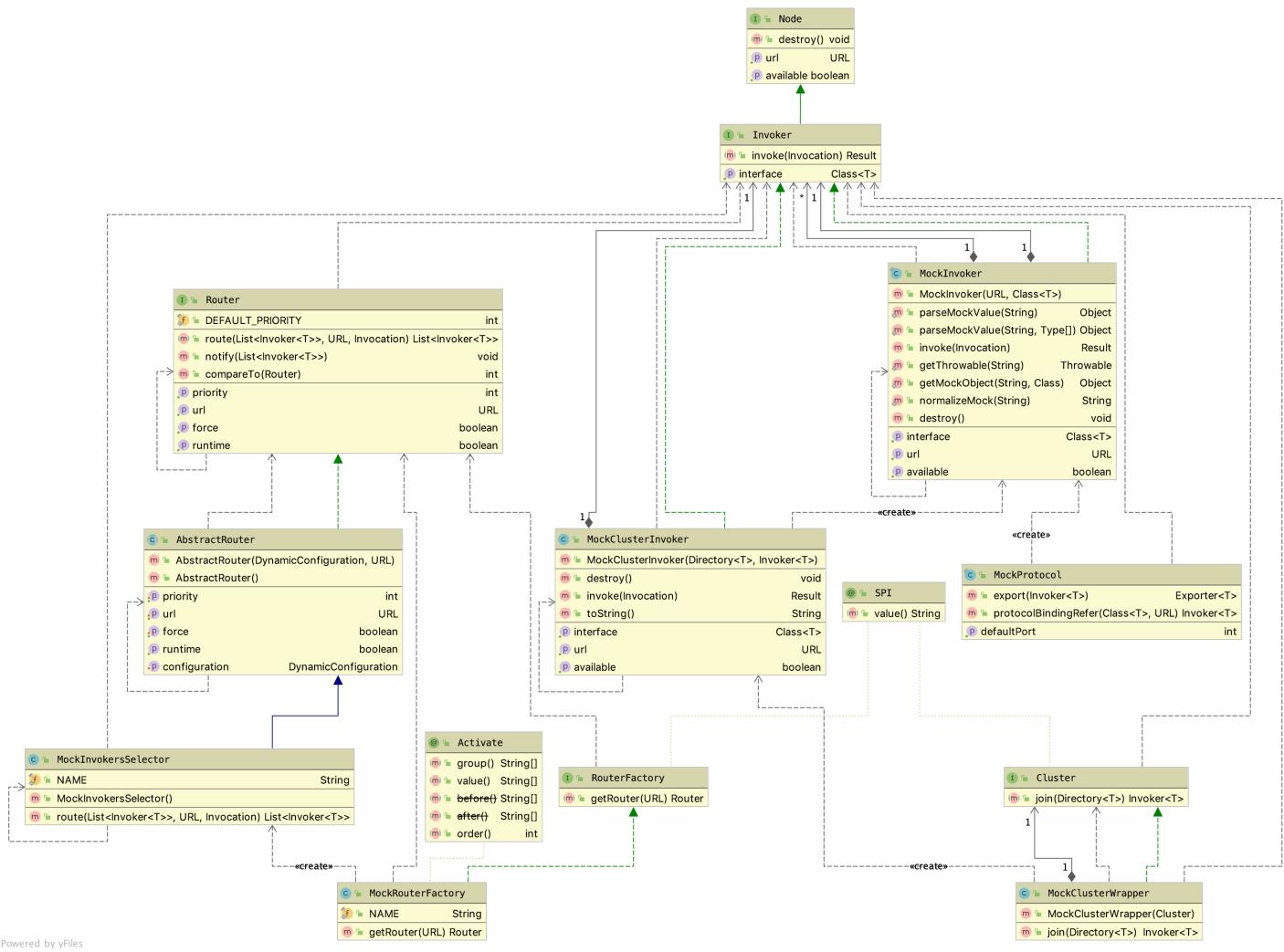
1. 首先在服务发现机制的基础上，由被装饰者 **cluster** 将实时发现的若干引用实例伪装成一个 **Invoker<T>** 实例 **invoker**；
2. 然后，使用 **MockClusterInvoker** 装饰 **invoker** 得到新的被引用微服务实例 **wrappedInvoker**；
3. 最后，**wrappedInvoker.invoke(invocation)** 的执行逻辑中，会根据总线配置 **url[[methodName + "."] + "mock"]** 确定如何调用 **invoker.invoke(invocation)** 以及遇异常时所应执行的降级逻辑；

```
public class MockClusterWrapper implements Cluster {
    private Cluster cluster;

    public MockClusterWrapper(Cluster cluster) {
        this.cluster = cluster;
    }

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new MockClusterInvoker<T>(directory, this.cluster.join(directory));
    }
}
```

JAVA



## MockClusterInvoker<T>

默认情况下，也就是如果消费端开发者没有为被引用微服务配置 Mock 特性，就相当于没有对实例 `invoker` 进行 `MockClusterInvoker` 的装饰处理。有一种场景是客户端使用的某个微服务只是预定定义好服务接口，尚处于开发过程中，这时如果直接发起RPC调用，可以肯定的是必定会返回空值或者抛出异常，因此Dubbo允许配置 `url[[methodName + "."] + "mock"] = "force:" + ("throw" | "return") + XXX`，也即强制使用 Mock 行为，不走远程调用。其它场景则是先 `invoker.invoke(invocation)`，遇到非业务 `RpcException` 异常后再执行对应的 Mock 行为，如果是业务异常，则表示服务端成功接收到了请求，只是业务逻辑处理异常，是需要直接向上抛给上端应用层的。由于异常既可以携带在 `Result` 返回的，也可以 `throw` 直接抛出，因此最后的 `fail-mock` 块执行了两次 `doMockInvoke(...)`。

JAVA

```

public class MockClusterInvoker<T> implements Invoker<T> {

    private final Directory<T> directory;
    private final Invoker<T> invoker;

    public MockClusterInvoker(Directory<T> directory, Invoker<T> invoker) {
        this.directory = directory;
        this.invoker = invoker;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        Result result = null;
        String value =
directory.getUrl().getMethodParameter(invocation.getMethodName(),
            MOCK_KEY, Boolean.FALSE.toString()).trim();
        if (value.length() == 0 || "false".equalsIgnoreCase(value)) {
            //no mock
            result = this.invoker.invoke(invocation);
        } else if (value.startsWith("force")) {
            if (logger.isWarnEnabled()) {
                logger.warn("force-mock: " + invocation.getMethodName()
                    + " force-mock enabled , url : " + directory.getUrl());
            }
            result = doMockInvoke(invocation, null);
        } else {
            //fail-mock
            try {
                result = this.invoker.invoke(invocation);
                if(result.getException() != null && result.getException() instanceof
RpcException){
                    RpcException rpcException= (RpcException)result.getException();
                    if(rpcException.isBiz()){ throw  rpcException;}
                    else {
                        result = doMockInvoke(invocation, rpcException);
                    }
                }
            } catch (RpcException e) {
                if (e.isBiz()) { throw e; }
                if (logger.isWarnEnabled()) {
                    logger.warn("fail-mock: " + invocation.getMethodName()
                        + " fail-mock enabled , url : " + directory.getUrl(), e);
                }
                result = doMockInvoke(invocation, e);
            }
        }
    }
    return result;
}
...
}

```

然而就像上述代码所示的，真正处理异常的 Mock 行为是由下述 `doMockInvoke(inv, expt)` 方法负责的，它会首先试图调用 `selectMockInvoker(invocation)` 在当前客户端应用中找到与 `invocation` 特征匹配的特定微服务的所有 `Invoker<T>` 实例，若匹配到，则取其中一个，否则则构

建一 `MockInvoker` 实例，用于 `Mock` 处理。和调用方不同的是，关于异常处理的方式反过来了，执行后若是遇到业务异常，则直接返回一个 `Result` 对象，因此上述源码中的 `fail-mock` 模块中至少不会因为业务异常陷入 `try-catch` 循环。

```
public class MockClusterInvoker<T> implements Invoker<T> {
    private Result doMockInvoke(Invocation invocation, RpcException e) {
        Result result = null;
        Invoker<T> minvoker;

        List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
        if (CollectionUtils.isEmpty(mockInvokers)) {
            minvoker = (Invoker<T>) new MockInvoker(
                directory.getUrl(), directory.getInterface());
        } else {
            minvoker = mockInvokers.get(0);
        }
        try {
            result = minvoker.invoke(invocation);
        } catch (RpcException me) {
            if (me.isBiz()) {
                result = AsyncRpcResult.newDefaultAsyncResult(me.getCause(),
                    invocation);
            } else {
                throw new RpcException(me.getCode(),
                    getMockExceptionMessage(e, me), me.getCause());
            }
        } catch (Throwable me) {
            throw new RpcException(getMockExceptionMessage(e, me), me.getCause());
        }
        return result;
    }

    private String getMockExceptionMessage(Throwable t, Throwable mt) {
        String msg = "mock error : " + mt.getMessage();
        if (t != null) {
            msg = msg + ", invoke error is :" + StringUtils.toString(t);
        }
        return msg;
    }
}
```

方法 `selectMockInvoker(invocation)` 的目的是利用 `directory` 服务发现搜集到所有 `url.protocol = "mock"` 的微服务引用实例，涉及到的 `MockProtocol` 及 `Mock` 版的 `Router` 实现将在下文相关章节讨论。

```

public class MockClusterInvoker<T> implements Invoker<T> {
    private List<Invoker<T>> selectMockInvoker(Invocation invocation) {
        List<Invoker<T>> invokers = null;
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation) invocation).setAttachment(INVOCATION_NEED_MOCK,
Boolean.TRUE.toString());
            try {
                invokers = directory.list(invocation);
            } catch (RpcException e) {
                if (logger.isInfoEnabled()) {
                    logger.info("Exception when try to invoke mock. Get mock invokers"
+ " error for service:" +
directory.getUrl().getServiceInterface()
+ ", method:" + invocation.getMethodName()
+ ", will construct a new mock with 'new MockInvoker()'.", e);
                }
            }
        }
        return invokers;
    }
    ...
}

```

## MockInvoker<T>

显然上述 `MockClusterInvoker<T>` 只负责了 `Mock` 实现中的总体流程逻辑，具体细节却没有涉及，而这恰恰是 `MockInvoker<T>` 的职责。

进一步分析实现前，我们先看看有关 `Mock` 本地伪装的具体应用。首先总线配置可以是方法级别的，也即上述出现的 `url[[methodName + "."] + "mock"]` 表示；其次，总线配置支持 `("force" | "fail") + ":"` 前缀，本地的 `Mock` 行为于前者是强制执行的，远程的RPC请求不会执行，于后者则只有远程的RPC请求发生异常时才执行，相当于默认的没有配前缀的情况；最后，指定的 `Mock` 行为有如下几种方式：

- 简洁配置：`("return" | "throw" | ("true" | "default" | "fail" | "force"))`，前者会相当于 `"return null"`，中者则抛出一个默认的 `RPCException` 异常，而后者会归化为 `"default"`，需要客户端提供一个在服务接口全名后附有 `Mock` 后缀的接口实现；
- 返回指定字面值：`"return" + ("empty" | "null" | "true" | "false" | JSON格式字符串 | 字符串字面量)`；
  - 其中 `"empty"` 代表空，基本类型的默认值，或者集合类的空值，而 `JSON格式字符串` 会被反序列化得到相应接口方法返回类型的对象。
- 指定目标：`("throw" + XXXException | "return" + XXXInterfaceImpl)`，这两种情况均需存在对应的匹配全类名的类，或为异常，或为接口实现；

```

<!--①-->
<dubbo:reference interface="com.foo.BarService" mock="com.foo.BarServiceMock" />

<!--②-->
<dubbo:reference id="demoService" check="false" interface="com.foo.BarService">
    <dubbo:parameter key="sayHello.mock" value="return fake"/>
</dubbo:reference>

```

XML

可见 Mock 行为实际上分为 3 种：1) Mock Value，直接返回值；2) Mock Object，将调用委托给实现了同一接口的类；3) Mock Throwable，抛错指定类型的异常。总体逻辑如下

```

final public class MockInvoker<T> implements Invoker<T> {
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        String mock = getUrl().getParameter(invocation.getMethodName() + "." + MOCK_KEY);
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation) invocation).setInvoker(this);
        }
        if (StringUtils.isBlank(mock)) {
            mock = getUrl().getParameter(MOCK_KEY);
        }

        if (StringUtils.isBlank(mock)) {
            throw new RpcException(new IllegalAccessException("mock can not be null.
url :" + url));
        }
        mock = normalizeMock(URL.decode(mock));
        if (mock.startsWith(RETURN_PREFIX)) {
            //Mock Value 返回
        } else if (mock.startsWith(THROW_PREFIX)) {
            //Mock Throwable抛错
        } else {
            //Mock Object 回调
        }
    }
    ...
}

```

JAVA

## Mock Value 返回

相对而言，返回具体值的这种场景，涉及到类型操作和构建目标类型对象，细节挺多，感兴趣的可以按图索骥看看具体实现。基本步骤是先利用 `invocation` 获取到接口方法的出参类型，然后结合该类型和 Mock 字面值 调用 `parseMockValue(...)` 转换为目标类型的值，最后调用 `AsyncRpcResult.newDefaultAsyncResult(value, invocation)` 填充并构建一个 `AsyncRpcResult` 类型对象。

```

final public class MockInvoker<T> implements Invoker<T> {
    ...
    public static Object parseMockValue(String mock) throws Exception {
        return parseMockValue(mock, null);
    }

    public static Object parseMockValue(String mock, Type[] returnTypes) throws
Exception {
        Object value = null;
        ...
        return value;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        ...
        //Mock Value 返回
        mock = mock.substring(RETURN_PREFIX.length()).trim();
        try {
            Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
            Object value = parseMockValue(mock, returnTypes);
            return AsyncRpcResult.newDefaultAsyncResult(value, invocation);
        } catch (Exception ew) {
            throw new RpcException("mock return invoke error. method :" +
invocation.getMethodName()
                + ", mock:" + mock + ", url: " + url, ew);
        }
        ...
    }
}

```

JAVA

## Mock Object 回调

显然，Mock配置为url[["force" | "fail"] + ":" ] + ("default" | "return " + XXXInterfaceImpl)时，需要使用反射实例化被引用微服务接口 serviceType 的一个本地实现，如下述源码所示，先使用全类名获得对应的类元数据 mockClass 后，还需要判断它是否实现自 serviceType。

```

final public class MockInvoker<T> implements Invoker<T> {
    public static Object getMockObject(String mockService, Class serviceType) {
        if (ConfigUtils.isDefault(mockService)) {
            mockService = serviceType.getName() + "Mock";
        }

        Class<?> mockClass = ReflectUtils.forName(mockService);
        if (!serviceType.isAssignableFrom(mockClass)) {
            throw new IllegalStateException("The mock class " + mockClass.getName() +
                " not implement interface " + serviceType.getName());
        }

        try {
            return mockClass.newInstance();
        } catch (InstantiationException e) {
            throw new IllegalStateException("No default constructor from mock class " +
                + mockClass.getName(), e);
        } catch (IllegalAccessException e) {
            throw new IllegalStateException(e);
        }
    }
}

...
}

```

Mock Object 是目标微服务接口的本地伪装实现，和本地存根实现不同的是，后者是接口的一个装饰者实现，而本地伪装认为发往对端的RPC调用已经失败，或者被 force 禁用了。尽管在有了伪装实现的全类名后利用反射获得了目标类元信息和目标类的实例 mockObj，可以从入参 invocation 中获得针对 mockObj 的当前被调用方法名以及入参信息，但还是没法像硬编码那样可以直接发方法调用，也就是针对目标方法的调用依然只能依靠反射机制。这里比较聪明的处理方式是直接利用既有代理层的 ProxyFactory 生成一个 AbstractProxyInvoker 实例（一般只用于服务端），由其利用反射机制将当前 invoke(invocation) 调用转换为对 mockObj 指定方法的调用。

另外，针对 mockObj 的方法调用并不是一次性的，也不仅是只针对它的其中一个方法做调用，为了避免不必要的CPU开销，特意申明了一个全局的 Map<String, Invoker<?>> 类型的 MOCK\_MAP 缓存容器，Key键是 mockObj 对应类的全类名，而 mockObj 缓存在Value值（一个 AbstractProxyInvoker 实现的对象）中。

**NOTE** 正常流程中客户端发起的 invoke(invocation) 调用中，最内核的 Invoker 执行体是 DubboInvoker，而 Mock 伪装中的则是 AbstractProxyInvoker 的匿名实现。

```
final public class MockInvoker<T> implements Invoker<T> {
```

JAVA

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

```
    private final static Map<String, Invoker<?>> MOCK_MAP =
        new ConcurrentHashMap<String, Invoker<?>>();
```

```
    private Invoker<T> getInvoker(String mockService) {
        Invoker<T> invoker = (Invoker<T>) MOCK_MAP.get(mockService);
        if (invoker != null) {
            return invoker;
        }
```

```
        Class<T> serviceType = (Class<T>)
ReflectUtils.forName(url.getServiceInterface());
        T mockObject = (T) getMockObject(mockService, serviceType);
        invoker = PROXY_FACTORY.getInvoker(mockObject, serviceType, url);
        if (MOCK_MAP.size() < 10000) {
            MOCK_MAP.put(mockService, invoker);
        }
        return invoker;
    }
```

```
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
```

```
        ...
        //Mock Object 回调
        try {
            Invoker<T> invoker = getInvoker(mock);
            return invoker.invoke(invocation);
        } catch (Throwable t) {
            throw new RpcException("Failed to create mock implementation class " +
mock, t);
        }
        ...
    }
}
```

## Mock Throwable 抛错

同样，Mock配置为url[["force" | "fail"] + ":" + "throw" + [XXXException]]时，也需要实例化一个异常类。一般而言，一个应用会根据需要对异常进行分类，只会声明少数几个异常类，因而声明了对应的Map容器，规避反复实例化所带来的开销。对应配置的异常类型必须带有一个字符串类型的构造函数，所有的异常实例的报错信息均一样：

```

final public class MockInvoker<T> implements Invoker<T> {
    ...
    private final static Map<String, Throwable> THROWABLE_MAP =
        new ConcurrentHashMap<String, Throwable>();

    public static Throwable getThrowable(String throwstr) {
        Throwable throwable = THROWABLE_MAP.get(throwstr);
        if (throwable != null) {
            return throwable;
        }

        try {
            Throwable t;
            Class<?> bizException = ReflectUtils.forName(throwstr);
            Constructor<?> constructor;
            constructor = ReflectUtils.findConstructor(bizException, String.class);
            t = (Throwable) constructor.newInstance(new Object[]{
                "mocked exception for service degradation."});
            if (THROWABLE_MAP.size() < 1000) {
                THROWABLE_MAP.put(throwstr, t);
            }
            return t;
        } catch (Exception e) {
            throw new RpcException("mock throw error :" + throwstr + " argument
error.", e);
        }
    }
}

```

异常处理时的调用方逻辑如下，无论是否指定 XXXException 时，都会抛出 RpcException 异常，只是当指明异常类名时，对应的异常被标记为业务异常了，getThrowable(mock) 所得沦为内嵌异常详情了。

```

final public class MockInvoker<T> implements Invoker<T> {
    ...
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        ...
        //Mock Throwble抛错
        mock = mock.substring(THROW_PREFIX.length()).trim();
        if (StringUtils.isBlank(mock)) {
            throw new RpcException("mocked exception for service degradation.");
        } else { // user customized class
            Throwable t = getThrowable(mock);
            throw new RpcException(RpcException.BIZ_EXCEPTION, t);
        }
        ...
    }
}

```

## MockInvokersSelector 路由实现

为了保证微服务的高可用性，实现诸如不同机房网段隔离、黑白名单、读写分离、前后台分离等特性，通常会为一个微服务在不同的机房或者网段上部署多个实例，客户端则根据业务需要为被引用微服务配置所需的路由规则，利用它们过滤出被引用微服务的目标实例集合。正如《Dubbo集群之路由》一文所言，通常用于过滤的路由器有多个，最终得到的集合是这些路由器作用的总和。

本章节的 `MockInvokersSelector` 其实是一个路由器实现，目的是为客户端筛选出本地伪装的被引用微服务实现。相对应地，还有一个专门用于创建其实例的工厂类 `MockRouterFactory`，源码中的 `@Activate` 注解说明 `MockRouterFactory` 是自动激活的，换言之，它所创建 `MockInvokersSelector` 路由器是 Dubbo 路由链 `RouterChain` 内置实例，所有需要用到路由功能的地方，只要 `inv["invocation.need.mock"] = true`，它均会发生作用。

```
@Activate  
public class MockRouterFactory implements RouterFactory {  
    public static final String NAME = "mock";  
  
    @Override  
    public Router getRouter(URL url) {  
        return new MockInvokersSelector();  
    }  
}
```

JAVA

上文中提到的 `selectMockInvoker(inv)` 方法被间接调用过好几次，`doMockInvoke(inv, expt)` 方法的开头会视图用它获得满足 `url.protocol = "mock"` 的所有被引用微服务实例集合，当获得结果为空时才会执行本地 Mock 伪装行为。

`MockInvokersSelector` 最核心的一段代码如下所示，也就是说类似 Mock 本地伪装示例中的 `url[[methodName + "."] + "mock"]` 配置引起了 `MockClusterInvoker#selectMockInvoker(inv)` 设置 `inv["invocation.need.mock"] = true` 参数，而后者又间接引起 `MockInvokersSelector` 去筛选 `url.protocol = "mock"` 的引用实例。反之，如果没有前面 `refer` 时的配置，也即不用 Mock 本地伪装时，所有配置了 `url.protocol = "mock"` 的引用实例会被剔除掉。

```
public class MockInvokersSelector extends AbstractRouter {  
  
    public static final String NAME = "MOCK_ROUTER";  
    private static final int MOCK_INVOKERS_DEFAULT_PRIORITY = Integer.MIN_VALUE;  
  
    public MockInvokersSelector() {  
        this.priority = MOCK_INVOKERS_DEFAULT_PRIORITY;  
    }  
  
    @Override  
    public <T> List<Invoker<T>> route(final List<Invoker<T>> invokers,  
                                         URL url, final Invocation invocation) throws  
RpcException {  
    if (CollectionUtils.isEmpty(invokers)) {  
        return invokers;  
    }  
  
    if (invocation.getAttachments() == null) {  
        return getNormalInvokers(invokers);  
    } else {  
        String value = invocation.getAttachments().get(INVOCATION_NEED_MOCK);  
        if (value == null) {  
            return getNormalInvokers(invokers);  
        } else if (Boolean.TRUE.toString().equalsIgnoreCase(value)) {  
            return getMockedInvokers(invokers);  
        }  
    }  
    return invokers;  
}  
...  
}
```

上述源码表示优先级的 priority 被设置成 Integer.MIN\_VALUE，这说明了 MockInvokersSelector 这个路由器实现在每次路由链执行时是优先被采用的。

最后留下的上述被用到的其它代码则非常简单，一个被引用微服务的实例列表，应用 hasMockProviders(invokers) 能够判别出是否含有满足 url.protocol = "mock" 的伪装实例，， getMockedInvokers(invokers) 获取所有伪装实例，而 getNormalInvokers(invokers) 则取得剔除伪装实例后的所有实例。

```
public class MockInvokersSelector extends AbstractRouter {  
    private <T> List<Invoker<T>> getMockedInvokers(final List<Invoker<T>> invokers) {  
        if (!hasMockProviders(invokers)) {  
            return null;  
        }  
        List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(1);  
        for (Invoker<T> invoker : invokers) {  
            if (invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL)) {  
                sInvokers.add(invoker);  
            }  
        }  
        return sInvokers;  
    }  
  
    private <T> List<Invoker<T>> getNormalInvokers(final List<Invoker<T>> invokers) {  
        if (!hasMockProviders(invokers)) {  
            return invokers;  
        } else {  
            List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(invokers.size());  
            for (Invoker<T> invoker : invokers) {  
                if (!invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL)) {  
                    sInvokers.add(invoker);  
                }  
            }  
            return sInvokers;  
        }  
    }  
  
    private <T> boolean hasMockProviders(final List<Invoker<T>> invokers) {  
        boolean hasMockProvider = false;  
        for (Invoker<T> invoker : invokers) {  
            if (invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL)) {  
                hasMockProvider = true;  
                break;  
            }  
        }  
        return hasMockProvider;  
    }  
    ...  
}
```

## TODO::其它服务降级方式

---

完结

# Dubbo 配置管理

应用开发中，配置管理的重要性不言而喻。微服务架构模式中，由于组成应用的微服务粒度很小，一个微服务会以集群的方式部署多个实例，彼此之间存在着复杂的协同或调用关系，服务实例甚至以不同的方式部署在不同机房的不同机器上，所有这些都说明在各个微服务进程中单独做配置管理是非常不科学的，因此业界提出使用配置中心做统一维护管理。

有过微服务开发经验的人都知道，不可能所有的配置都放到配置中心的，一般会根据实际需要将那些和环境配套的、需要在线做服务治理的配置项提到配置中心，也就是说配置分为本地和外部两种，就一个服务实例而言本地配置和配置项默认值组成了外部配置的超集，Dubbo将外部配置看做是动态配置。

Dubbo中的配置中心实现是一个扩展点，已经提供了包括 Redis、Sofa、Nacos、Etcd3、Consul、Zookeeper、Default、Multiple、Multicast 多个实现，本文将结合 Zookeeper 等少数几个剖析Dubbo 中的配置中心实现。

## NOTE

Dubbo中的注册中心和配置中心一般是实现在一起的，比如用的是同一 Zookeeper 集群，只是使用的数据节点不一样而已。

## 配置覆写

Dubbo微服务治理中，运维人员可以经配置中心向微服务发出配置覆写请求，如下述示例代码：

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(  
    RegistryFactory.class).getAdaptiveExtension();  
  
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));  
  
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?" +  
    "category=configurators&dynamic=false&application=foo&timeout=1000"));  
  
JAVA
```

示例中的意思是，先获取到一个 RegistryFactory 创建工厂，由后者获得指定 Zookeeper 配置中心 ZK 的 Registry 本地客户端，调用 register() 方法写入覆写规则（将所匹配微服务的 timeout 设为1000），由 ZK实例负责改写配置，并将信息同步给接入到 ZK实例 的相应微服务。关于覆写规则，具体如下述Dubbo官方给的描述：

“ override:// 表示数据采用覆盖方式，支持 override 和 absent，可扩展，必填。

0.0.0.0 表示对所有 IP 地址生效，如果只想覆盖某个 IP 的数据，请填入具体 IP，必填。

com.foo.BarService 表示只对指定服务生效，必填。

`category=configurators` 表示该数据为动态配置类型，必填。

`dynamic=false` 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，必填。

`enabled=true` 覆盖规则是否生效，可不填，缺省生效。

`application=foo` 表示只对指定应用生效，可不填，表示对所有应用生效。

`timeout=1000` 表示将满足以上条件的 `timeout` 参数的值覆盖为 1000。如果想覆盖其它参数，直接加在 `override` 的 URL 参数上。

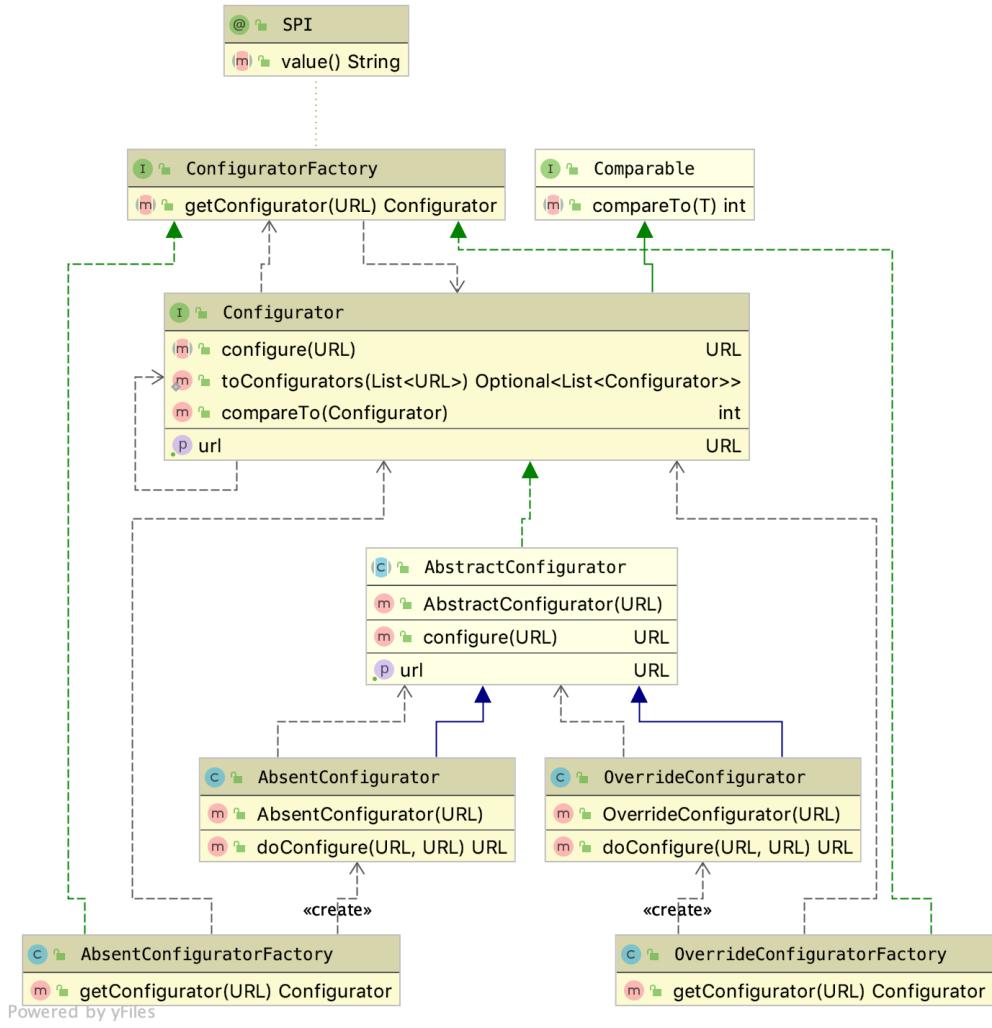
假如我们使用 `override://10.20.153.10/com.foo.BarService?`

`category=configurators&dynamic=false` 作为匹配的目标微服务的条件，加入对应的参数就能实现如下对应的操作目的：

- `disabled=true`：禁用提供者（通常用于临时踢除某台提供者机器，相似的，禁止消费者访问请使用路由规则）
- `weight=200`：调整权重（通常用于容量评估，缺省权重为 100）
- `loadbalance=leastactive`：调整负载均衡策略（缺省负载均衡策略为 random）
- `application=foo&mock=force:return+null`：服务降级（通常用于临时屏蔽某个出错的非关键服务）

## 覆写规则处理

如下图所示的类图，覆写规则的处理实现被安排在 `dubbo-cluster` 模块中。



从《Dubbo注册中心》这篇文章中我们得知，无论是 Consumer 还是 Provider，他们只是一个概念上的存在，帮助我们识别一份 URL 数据所代表的含义。覆盖规则实质是Dubbo根据所给定 `override://` 或 `absent://` 类型的 URL 数据对表征 Provider 的 URL 数据的改写处理，Dubbo中将实现这个改写的行为类命名为 Configurator。

在设计上，Dubbo认为 Configurator 应该是一个扩展点，然而由于每份 Provider 都需要它的实现类的一个实例，SPI的扩展点的单例模式与这个需求是相悖的，因此利用SPI机制声明一个单例的用于创建 Configurator 的工厂类—— ConfiguratorFactory，每一种工厂实现尽管本身是单例的，但却可以随时根据需要创建 Configurator 实例。上文已经知道Dubbo中提供了两种覆盖表征 Provider 的 URL 数据的方式， `override` 和 `absent`，这样上述类图中左半边的部分所表示的含义也是很清楚了。

## 源码实现

如下述 ConfiguratorFactory 扩展点的定义，使用Dubbo的SPI机制生成的代理类调用 `getConfigurator(url)` 方法时，它会根据url入参中的 `protocol` 部分配置的名称获取到某个 ConfiguratorFactory 实现类的实例，将当前调用委托给同名方法。

```

@SPI
public interface ConfiguratorFactory {

    /**
     * get the configurator instance.
     *
     * @param url - configurator url.
     * @return configurator instance.
     */
    @Adaptive("protocol")
    Configurator getConfigurator(URL url);
}

public class AbsentConfiguratorFactory implements ConfiguratorFactory {

    @Override
    public Configurator getConfigurator(URL url) {
        return new AbsentConfigurator(url);
    }
}

public class OverrideConfiguratorFactory implements ConfiguratorFactory {

    @Override
    public Configurator getConfigurator(URL url) {
        return new OverrideConfigurator(url);
    }
}

```

上述用到的 `override://0.0.0.0/com.foo.BarService...` 示例中，使用 SPI 获取的适配类调用 `getConfigurator(url)` 方法，内部调用实际上后面被委托给了 `OverrideConfiguratorFactory` 的实例单例，由其携带入参 `url` 创建一个 `Configurator` 对象。到这里，接下来再理解 `Configurator` 接口中定义的 `toConfigurators()` 方法就轻松多了。

先看看 `Configurator` 的接口方法定义，如下，它的每一个实例都会持有一份覆写规则，客户端可以调用 `configure(url)` 刷新得到新的 URL 数据。

```

public interface Configurator extends Comparable<Configurator> {

    //用于定义覆盖Provider的规则
    URL getUrl();

    //将入参所指Provider使用当前Configurator所配规则覆盖，返回新的URL数据
    URL configure(URL url);

    ...//default方法
}

```

另外 `Configurator` 接口中还定义了两个方法的完整接口定义如下。其中 `toConfigurators()` 将入参所有的覆写规则进行汇总处理，其中有任何一个将 `protocol` 设置为 `empty` 都会导致所有清空所有的覆写规则，列表中若某个 URL 元素的参数在移除掉掉 `anyhost` 这一项后没有其他参数，则会清空此前的覆写规则，最后将汇总得到的 `Configurator` 实例按照 `host` 优先或者 `priority` 排序。显然，`toConfigurators(urls)` 方法存在的目的是，将所有历史覆写规则汇总，用于配置中心给 `Provider` 重新计算 URL 配置数据。

```

/**
 *
 Convert override urls to map for use when re-refer. Send all rules every time, the urls will
be reassembled and calculated
 *
 * URL contract:
 * <ol>
 * <li>override://0.0.0.0/... ( or override://ip:port...?anyhost=true)&param=value1... means
global rules
 * (all of the providers take effect)</li>
 * <li>override://ip:port...?anyhost=false Special rules (only for a certain provider)</li>
 * <li>override:// rule is not supported... ,needs to be calculated by registry itself</li>
 * <li>override://0.0.0.0/ without parameters means clearing the override</li>
 * </ol>
 *
 * @param urls URL list to convert
 * @return converted configurator list
 */
static Optional<List<Configurator>> toConfigurators(List<URL> urls) {
    if (CollectionUtils.isEmpty(urls)) {
        return Optional.empty();
    }

    ConfiguratorFactory configuratorFactory =
        ExtensionLoader.getExtensionLoader(ConfiguratorFactory.class)
            .getAdaptiveExtension();

    List<Configurator> configurators = new ArrayList<>(urls.size());
    for (URL url : urls) {
        if (EMPTY_PROTOCOL.equals(url.getProtocol())) {
            configurators.clear();
            break;
        }
        Map<String, String> override = new HashMap<>(url.getParameters());
        //The anyhost parameter of override may be added automatically, it can't change the
judgement of changing url
        override.remove(ANYHOST_KEY);
        if (override.size() == 0) {
            configurators.clear();
            continue;
        }
        configurators.add(configuratorFactory.getConfigurator(url));
    }
    Collections.sort(configurators);
    return Optional.of(configurators);
}

/**
 * Sort by host, then by priority
 * 1. the url with a specific host ip should have higher priority than 0.0.0.0
 * 2. if two url has the same host, compare by priority value;
 */
@Override
default int compareTo(Configurator o) {
    if (o == null) {
        return -1;
    }

    int ipCompare = getUrl().getHost().compareTo(o.getUrl().getHost());
    // host is the same, sort by priority
}

```

```
if (ipCompare == 0) {
    int i = getUrl().getParameter(PRIORITY_KEY, 0);
    int j = o.getUrl().getParameter(PRIORITY_KEY, 0);
    return Integer.compare(i, j);
} else {
    return ipCompare;
}
}
```

最后我们来看看 `AbstractConfigurator` 类的实现，基本步骤如下：

1. 首先排除掉不会出现覆写的情况：

- 覆写规则 `configuratorUrl`：
  - a. 设置了 `enabled=false`；
  - b. 没有设置 `host`
- 入参中 `url`：
  - a. 值为 `null`；
  - b. `getHost()` 的值为 `null`；

2. 根据覆写规则使用的版本是否包含于 `configVersion` 这个配置执行相应逻辑：

- $\geq 2.7.0$ ：
  - a. 若 `url` 和 `configuratorUrl` 配置的 `side` 参数均为 `consumer`，且 `configuratorUrl` 配置了 `port=0`：
    - 返回 `configureIfMatch(NetUtils.getLocalHost(), url)`；
  - b. 若 `url` 和 `configuratorUrl` 配置的 `side` 参数均为 `provider`，且二者配置了一样的 `port` 值：
    - 返回 `configureIfMatch(url.getHost(), url)`；
- $< 2.7.0$ ：
  - a. 返回 `configureDeprecated(url)`；

3. 返回原 `url` 数据；

```

public URL configure(URL url) {
    // If override url is not enabled or is invalid, just return.
    if (!configuratorUrl.getParameter(ENABLED_KEY, true) ||
        configuratorUrl.getHost() == null || url == null || url.getHost() == null) {
        return url;
    }
    /**
     * This if branch is created since 2.7.0.
     */
    String apiVersion = configuratorUrl.getParameter(CONFIG_VERSION_KEY);
    if (StringUtils.isNotEmpty(apiVersion)) {
        String currentSide = url.getParameter(SIDE_KEY);
        String configuratorSide = configuratorUrl.getParameter(SIDE_KEY);

        if (currentSide.equals(configuratorSide) && CONSUMER.equals(configuratorSide)
            && 0 == configuratorUrl.getPort()) {
            url = configureIfMatch(NetUtils.getLocalHost(), url);

        } else if (currentSide.equals(configuratorSide) &&
                   PROVIDER.equals(configuratorSide) && url.getPort() ==
configuratorUrl.getPort()) {
            url = configureIfMatch(url.getHost(), url);
        }
    }
    /**
     * This else branch is deprecated and is left only to keep compatibility with versions
before 2.7.0
     */
    else {
        url = configureDeprecated(url);
    }
    return url;
}

```

下述继续看看 `configureIfMatch(host, url)` 做了什么操作? 从下述源码可以看出:

只有满足下述条件才会被执行覆写处理 and:

1. `host == "0.0.0.0"` 或者 `host.equals(configuratorUrl.getHost())`;
2. `configuratorUrl` 中的 `providerAddresses` 配置 or:
  - 没有该配置;
  - 值为 "`0.0.0.0`" 通配符;
  - 值包含了 `url` 的 `address` 值, 也即 `host[":" + port]`;
3. `configuratorUrl` 启用排除符的配置项, 并没有包含在 `url` 中, 比如前者中含有 `~key1=kylin`, 而后者并不包含 `key1=kylin`;

```

private URL configureIfMatch(String host, URL url) {
    if (ANYHOST_VALUE.equals(configuratorUrl.getHost()) ||
host.equals(configuratorUrl.getHost())) {
        // TODO, to support wildcards
        String providers = configuratorUrl.getParameter(OVERRIDE_PROVIDERS_KEY);
        if (StringUtils.isEmpty(providers) || providers.contains(url.getAddress()) ||
providers.contains(ANYHOST_VALUE)) {
            String configApplication = configuratorUrl.getParameter(APPLICATION_KEY,
                configuratorUrl.getUsername());
            String currentApplication = url.getParameter(APPLICATION_KEY, url.getUsername());
            if (configApplication == null || ANY_VALUE.equals(configApplication)
                || configApplication.equals(currentApplication)) {
                Set<String> conditionKeys = new HashSet<String>();
                conditionKeys.add(CATEGORY_KEY);
                conditionKeys.add(Constants.CHECK_KEY);
                conditionKeys.add(DYNAMIC_KEY);
                conditionKeys.add(ENABLED_KEY);
                conditionKeys.add(GROUP_KEY);
                conditionKeys.add(VERSION_KEY);
                conditionKeys.add(APPLICATION_KEY);
                conditionKeys.add(SIDE_KEY);
                conditionKeys.add(CONFIG_VERSION_KEY);
                conditionKeys.add(COMPATIBLE_CONFIG_KEY);
                for (Map.Entry<String, String> entry :
configuratorUrl.getParameters().entrySet()) {
                    String key = entry.getKey();
                    String value = entry.getValue();
                    if (key.startsWith("~") || APPLICATION_KEY.equals(key) ||
SIDE_KEY.equals(key)) {
                        conditionKeys.add(key);
                        if (value != null && !ANY_VALUE.equals(value)
                            && !value.equals(url.getParameter(key.startsWith("~") ?
key.substring(1) : key))) {
                            return url;
                        }
                    }
                }
                return doConfigure(url, configuratorUrl.removeParameters(conditionKeys));
            }
        }
    }
    return url;
}

```

从上述源码中可以看出，覆写中移除了一些常见配置项，包括 category、check、dynamic、enabled、group、version、application、side、configVersion、compatible\_config，另外加了排除符“~”前缀的配置项也被移除了。

相对而言，在 < 2.7.0 版本的实现看似简单些，如下：

```

private URL configureDeprecated(URL url) {
    if (configuratorUrl.getPort() != 0) {
        if (url.getPort() == configuratorUrl.getPort()) {
            //当覆盖规则里头配置了port, 且url和该值相等时
            return configureIfMatch(url.getHost(), url);
        }
    } else {
        if (url.getParameter(SIDE_KEY, PROVIDER).equals(CONSUMER)) {
            //若url中指定了side=consumer
            return configureIfMatch(NetUtils.getLocalHost(), url);
        } else if (url.getParameter(SIDE_KEY, CONSUMER).equals(PROVIDER)) {
            //若url中指定了side=provider
            return configureIfMatch(ANYHOST_VALUE, url);
        }
    }
    return url;
}

```

## IMPORTANT

当看到 `NetUtils.getLocalHost()` 这一截代码时，读者可能会有些费解卡壳，因为我们的源码是在一个微观的粒度去剖析的，放回到宏观，就比较容易理解了。

Dubbo微服务中，配置中心和 Provider|Consumer 是完全解耦的，Provider 和 Consumer 也是通过它解耦了。

当运维人员向配置中心 Reg 发送一个治理请求后，Reg 仅仅是发生了一个节点（树形结构中的一条数据）的数据变化而已，而这种变化是它会通知给使用长连接接入的 Provider|Consumer，后者使用本地 Registry 客户端回调相应的方法，执行规则覆写处理。

因此看到针对 `side=consumer` 这种配置覆写场景中，将入参 url 值的 host 改写为自身的本地IP地址。这个IP地址也是它用于注册到配置中心的那个，

`ANYHOST_VALUE.equals(configuratorUrl.getHost()) || host.equals(configuratorUrl.getHost())` 这一条件又保证了只有匹配覆写规则的那个Consumer才能执行该操作。

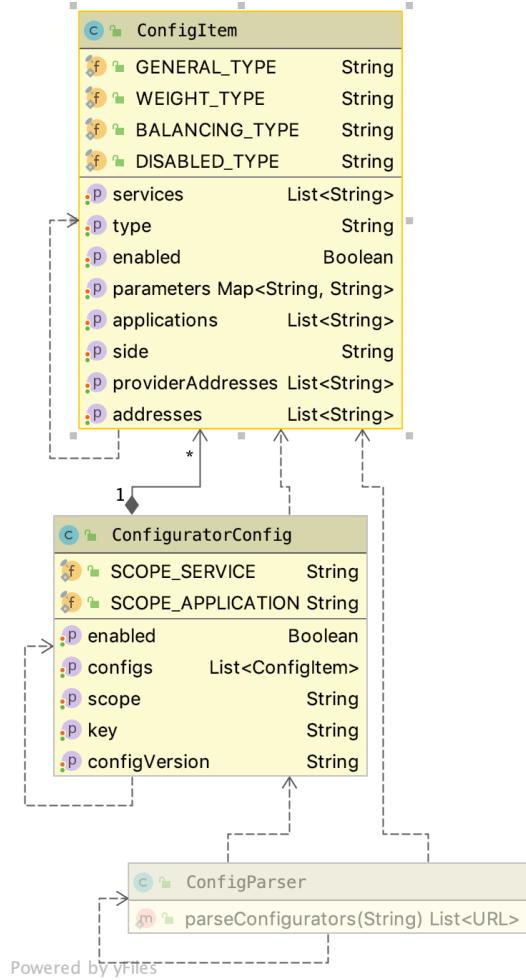
最后轮到 Configurator 的两个实现上场了，他们只有一行代码的区别，Override 模式直接使用覆写规则中的参数覆写，而 Absent 模式仅仅在原 url 数据没有覆写规则中的参数时才加入。

```
public class AbsentConfigurator extends AbstractConfigurator {  
  
    public AbsentConfigurator(URL url) {  
        super(url);  
    }  
  
    @Override  
    public URL doConfigure(URL currentUrl, URL configUrl) {  
        return currentUrl.addParametersIfAbsent(configUrl.getParameters());  
    }  
  
}  
  
public class OverrideConfigurator extends AbstractConfigurator {  
  
    public OverrideConfigurator(URL url) {  
        super(url);  
    }  
  
    @Override  
    public URL doConfigure(URL currentUrl, URL configUrl) {  
        return currentUrl.addParameters(configUrl.getParameters());  
    }  
  
}
```

## 外部配置到URL数据的转换

我们都清楚，Dubbo中的配置数据绝大部分是使用配置总线URL做维护、传递和内存本地存取处理的，而处于注册中心的数据是以节点的形式存储存在树形结构中，因而其下发到本地或本地从中拉取的数据，需要执行特定的转换处理才能最终转换为URL形式的数据。

如下述类图所示，Dubbo在 `dubbo-cluster` 模块中定义了实现配置转换的 `ConfigParser` 类，将从线上获取的配置数据转换为一序列的URL数据。



幸运的是线上的数据获取下来组成的文本是符合YAML规范的，因此可以经过它的转换变成本地的结构化数据，再由它转换为URL数据。下面看看其具体实现，源码比较长，依然打散视之。

首先，将获取的原生数据转换为本地结构化数据，如下，指定自定义的数据类型，告知Yaml如何转换，由其加载成对应类型的数据：

```

private static <T> T parseObject(String rawConfig) {
    Constructor constructor = new Constructor(ConfiguratorConfig.class);
    TypeDescription itemDescription = new TypeDescription(ConfiguratorConfig.class);
    itemDescription.addPropertyParameters("items", ConfigItem.class);
    constructor.addTypeDescription(itemDescription);

    Yaml yaml = new Yaml(constructor);
    return yaml.load(rawConfig);
}

```

JAVA

然后，根据获取到 ConfiguratorConfig 类型的数据中的 scope 值，判定配置是作用在应用级别还是微服务级别，根据级别模式对其所有 ConfigItem 类型数据项转换成URL类型的数据，汇集成列表返回：

```

public static List<URL> parseConfigurators(String rawConfig) {
    List<URL> urls = new ArrayList<>();
    ConfiguratorConfig configuratorConfig = parseObject(rawConfig);

    String scope = configuratorConfig.getScope();
    List<ConfigItem> items = configuratorConfig.getConfigs();

    if (ConfiguratorConfig.SCOPE_APPLICATION.equals(scope)) {
        items.forEach(item -> urls.addAll(appItemToUrls(item, configuratorConfig)));
    } else {
        // service scope by default.
        items.forEach(item -> urls.addAll(serviceItemToUrls(item, configuratorConfig)));
    }
    return urls;
}

```

JAVA

## 到URL数据的转换

### 共同支持函数

从上述定义的数据结构 `ConfiguratorConfig` 和 `ConfigItem` 来看，虽然数据分为两个不同级别的处理，但是数据的组成形式是一样的，具体处理时会有诸多共同之处。

配置项中的 `address` 若被设置为空值，则转换为匹配任意 Provider 或者 Consumer 的通配IP地址 `0.0.0.0`。

```

private static List<String> parseAddresses(ConfigItem item) {
    List<String> addresses = item.getAddresses();
    if (addresses == null) {
        addresses = new ArrayList<>();
    }
    if (addresses.size() == 0) {
        addresses.add(ANYHOST_VALUE);
    }
    return addresses;
}

```

JAVA

设置URL数据的的 `enabled` 参数，如果 `ConfigItem` 类型的数据中的 `type` 值为 `null` 或者等于 `general`，则从上一级取值，否则在当前数据项中取值。

```

private static void parseEnabled(ConfigItem item, ConfiguratorConfig config, StringBuilder urlBuilder) {
    urlBuilder.append("&enabled=");
    if (item.getType() == null || ConfigItem.GENERAL_TYPE.equals(item.getType())) {
        urlBuilder.append(config.getEnabled());
    } else {
        urlBuilder.append(item.getEnabled());
    }
}

```

JAVA

将`ConfigItem`类型的数据转换成URL中形如 `category=dynamicconfigurators[&side={side}] [&key1=val1&key2=val2&...&keyn=valn1,addr2,...,addrn]` 的组成。

```
private static String toParameterString(ConfigItem item) {
    StringBuilder sb = new StringBuilder();
    sb.append("category=");
    sb.append(DYNAMIC_CONFIGURATORS_CATEGORY);
    if (item.getSide() != null) {
        sb.append("&side=");
        sb.append(item.getSide());
    }
    Map<String, String> parameters = item.getParameters();
    if (CollectionUtils.isEmptyMap(parameters)) {
        throw new IllegalStateException("Invalid configurator rule, please specify at least
one parameter " +
            "you want to change in the rule.");
    }

    parameters.forEach((k, v) -> {
        sb.append("&");
        sb.append(k);
        sb.append("=");
        sb.append(v);
    });

    if (CollectionUtils.isNotEmpty(item.getProviderAddresses())) {
        sb.append("&");
        sb.append(OVERRIDE_PROVIDERS_KEY);
        sb.append("=");
        sb.append(CollectionUtils.join(item.getProviderAddresses(), ","));
    }
}

return sb.toString();
}
```

将形如 [{group}]/[{interfaceName}][:{version}] 的 serviceKey 转换成URL中形如 {interfaceName}?[group={group}&][version={version}&] 的组成。

```

private static String appendService(String serviceKey) {
    StringBuilder sb = new StringBuilder();
    if (StringUtils.isEmpty(serviceKey)) {
        throw new IllegalStateException("service field in configuration is null.");
    }

    String interfaceName = serviceKey;
    int i = interfaceName.indexOf("/");
    if (i > 0) {
        sb.append("group=");
        sb.append(interfaceName, 0, i);
        sb.append("&");

        interfaceName = interfaceName.substring(i + 1);
    }
    int j = interfaceName.indexOf(":");
    if (j > 0) {
        sb.append("version=");
        sb.append(interfaceName.substring(j + 1));
        sb.append("&");
        interfaceName = interfaceName.substring(0, j);
    }
    sb.insert(0, interfaceName + "?");

    return sb.toString();
}

```

## 应用级别

先获取所有的匹配目标 Consumer 或者 Provider 的 address 数据，遍历，挨个取 serviceKey 集合 services，若为空，在返回列表 urls 中增加一项URL数据 override://{{addr}}/\*，否则内部再遍历 services，给 urls 中加入URL数据，形如：

```

override://{{addr}}/{{interfaceName}}? [group={{group}}&][version={{version}}&]
category=dynamicconfigurators[&side={{side}}] [&key1=val1&key2=val2&...&keyn=valn]
[&providerAddresses=addr1,addr2,...,addrn] &application={{app}}&enabled={{enabled}}
&category=appdynamicconfigurators&configVersion={{configVersion}}

```

```

private static List<URL> appItemToUrls(ConfigItem item, ConfiguratorConfig config) {
    List<URL> urls = new ArrayList<>();
    List<String> addresses = parseAddresses(item);
    for (String addr : addresses) {
        StringBuilder urlBuilder = new StringBuilder();
        urlBuilder.append("override://").append(addr).append("/");
        List<String> services = item.getServices();
        if (services == null) {
            services = new ArrayList<>();
        }
        if (services.size() == 0) {
            services.add("*");
        }
        for (String s : services) {
            urlBuilder.append	appendService(s));
            urlBuilder.append(toParameterString(item));

            urlBuilder.append("&application=").append(config.getKey());
            parseEnabled(item, config, urlBuilder);

            urlBuilder.append("&category=").append(APP_DYNAMIC_CONFIGURATORS_CATEGORY);
            urlBuilder.append("&configVersion=").append(config.getConfigVersion());

            urls.add(URL.valueOf(urlBuilder.toString()));
        }
    }
    return urls;
}

```

## 微服务级别

先获取所有的匹配目标Consumer或者Provider的 address 数据，遍历，给urls中加入URL数据（如果 ConfigItem 数据的 applications 不为空，则内部多一层循环），形如：

```

override://{{addr}}/{{interfaceName}}? [group={{group}}&][version={{version}}&]
category=dynamicconfigurators[&side={{side}}] [&key1=val1&key2=val2&...&keyn=valn]
[&providerAddresses=addr1,addr2,...,addrn] &enabled={{enabled}}
&category=dynamicconfigurators&configVersion={{configVersion}} [&application={{app}}]

```

```
private static List<URL> serviceItemToUrls(ConfigItem item, ConfiguratorConfig config) { JAVA
    List<URL> urls = new ArrayList<>();
    List<String> addresses = parseAddresses(item);

    addresses.forEach(addr -> {
        StringBuilder urlBuilder = new StringBuilder();
        urlBuilder.append("override://").append(addr).append("/");

        urlBuilder.append	appendService(config.getKey()));
        urlBuilder.append(toParameterString(item));

        parseEnabled(item, config, urlBuilder);

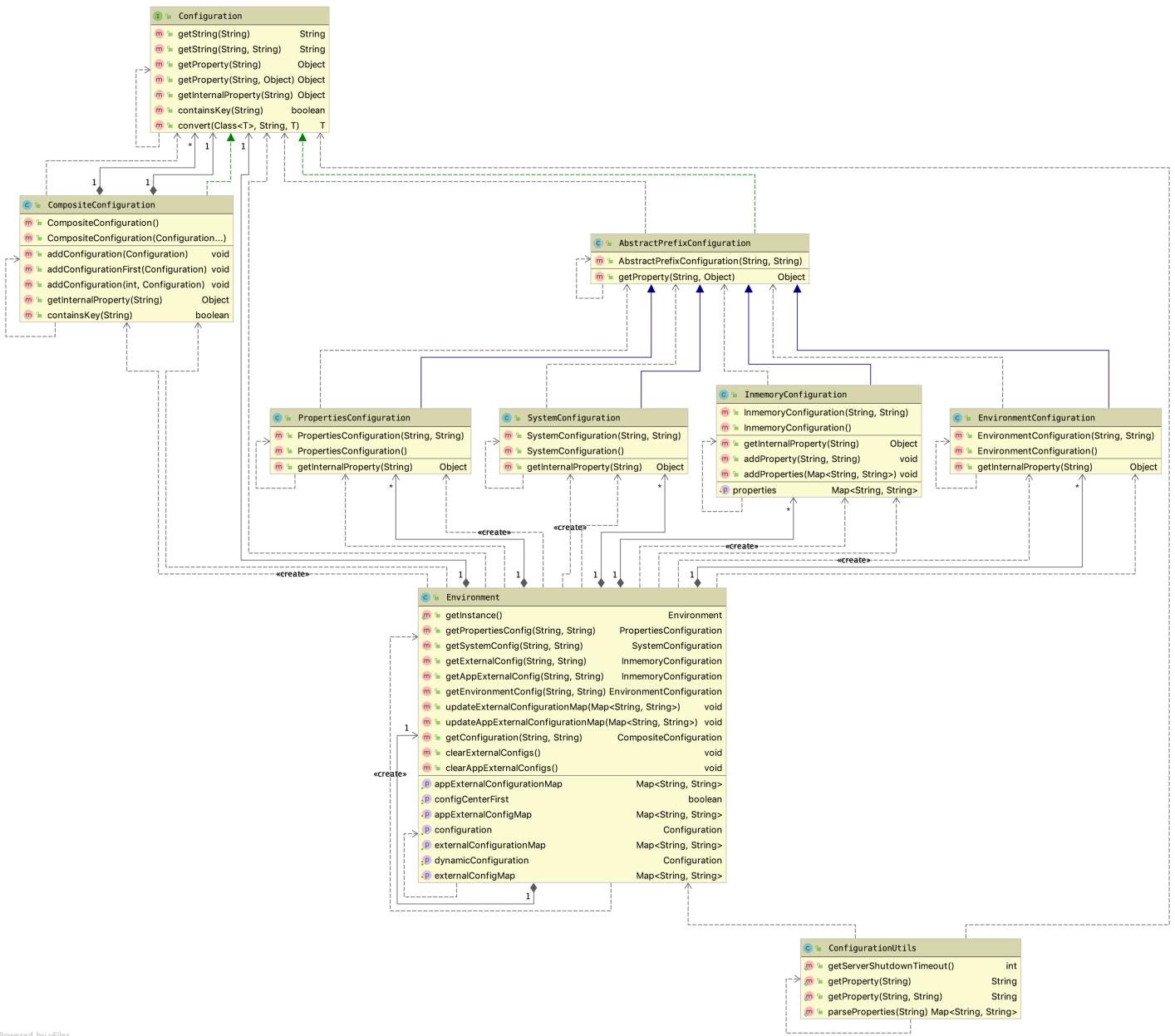
        urlBuilder.append("&category=").append(DYNAMIC_CONFIGURATORS_CATEGORY);
        urlBuilder.append("&configVersion=").append(config.getConfigVersion());

        List<String> apps = item.getApplications();
        if (apps != null && apps.size() > 0) {
            apps.forEach(app ->
                urls.add(URL.valueOf(urlBuilder.append("&application=").append(app).toString())));
        } else {
            urls.add(URL.valueOf(urlBuilder.toString()));
        }
    });

    return urls;
}
```

## 本地配置

宏观上讲，Dubbo中的本地配置实际上包含了多个部分，比如构成工程骨架的元数据、业务逻辑用到的配置项、当前JVM运行的环境参数、微服务治理中的应用本地默认设置等。



Powered by yfiles

本章节涉及的范围只是它的一个子集，如上述类图所示，包括：

1. 系统配置 —— **SystemConfiguration**：当前JVM的系统配置，这里的系统特指JVM虚拟机，特定于应用本身；
2. 环境配置 —— **EnvironmentConfiguration**：当前JVM所在的操作系统的环境配置内容，不受应用支配；
3. 属性配置 —— **PropertiesConfiguration**：用于业务逻辑控制的配置参数，一般配置在 `*.properties` 文件中；
4. 内存配置 —— **InmemoryConfiguration**：微服务应用运行时产生的配置项数据，应用关闭后就消失不见了；

类图中可以看出，所有这些配置读取实现最终会通过 **Environment** 类统一体检接口界面。

## 源码剖析

### Configuration 接口

`Configuration` 接口是Dubbo中所有配置项读取用户接口，包括动态配置中使用到的用户接口。`DynamicConfiguration` 也扩展自它，实际需要实现的只有 `getInternalProperty(key)` 这么一个方法，如下所示：

```
public interface Configuration {  
    //返回的配置数据类型为Object  
    Object getInternalProperty(String key);  
    ...//其它default方法  
}
```

JAVA

另外接口中实现了很多依赖于 `getInternalProperty(key)` 工具方法，便利 Dubbo 的应用开发。

`getProperty(...)` 方法取到的是一个 `Object` 对象，调用方还需要做相应的数据类型转换处理。

```
default Object getProperty(String key) {  
    return getProperty(key, null);  
}  
  
default Object getProperty(String key, Object defaultValue) {  
    Object value = getInternalProperty(key);  
    return value != null ? value : defaultValue;  
}  
  
default boolean containsKey(String key) {  
    return getProperty(key) != null;  
}
```

JAVA

`getString(...)` 方法取到的是一个 `String` 字符串，内部实现已经使用 `convert()` 方法完成了类型的转换处理，调用方无需做数据类型转换处理。由于一般而言，配置数据的都是 `String` 类型，`convert()` 方法中先使用 `(String) getProperty(key)` 获取到一个字符串值，再完成到其它基本类型的转换，否则可以直接使用它完成到其它目标类型的处理，其中 `we only process String properties for now` 注解也提示了这点。

```

default String getString(String key) {
    return convert(String.class, key, null);
}

default String getString(String key, String defaultValue) {
    return convert(String.class, key, defaultValue);
}

default <T> T convert(Class<T> cls, String key, T defaultValue) {
    // we only process String properties for now
    String value = (String) getProperty(key);

    if (value == null) {
        return defaultValue;
    }

    Object obj = value;
    if (cls.isInstance(value)) {
        return cls.cast(value);
    }

    if (Boolean.class.equals(cls) || Boolean.TYPE.equals(cls)) {
        obj = Boolean.valueOf(value);
    } else if (Number.class.isAssignableFrom(cls) || cls.isPrimitive()) {
        if (Integer.class.equals(cls) || Integer.TYPE.equals(cls)) {
            obj = Integer.valueOf(value);
        } else if (Long.class.equals(cls) || Long.TYPE.equals(cls)) {
            obj = Long.valueOf(value);
        } else if (Byte.class.equals(cls) || Byte.TYPE.equals(cls)) {
            obj = Byte.valueOf(value);
        } else if (Short.class.equals(cls) || Short.TYPE.equals(cls)) {
            obj = Short.valueOf(value);
        } else if (Float.class.equals(cls) || Float.TYPE.equals(cls)) {
            obj = Float.valueOf(value);
        } else if (Double.class.equals(cls) || Double.TYPE.equals(cls)) {
            obj = Double.valueOf(value);
        }
    } else if (cls.isEnum()) {
        obj = Enum.valueOf(cls.asSubclass(Enum.class), value);
    }

    return cls.cast(obj);
}

```

## AbstractPrefixConfiguration 超类

Dubbo允许所有的本地配置实现都加上一个前缀，增加可识别性，其好处很明显，配置项数据比较多时，便于开发运维人员快速在脑海分门别类的对应起来，减轻大脑负。

前缀由两部分组成，分别是 `id` 和 `prefix`，加入程序中取配置的项是 `key`，则内部对应的配置项实际是 `[{prefix} + "."][{id} + "."]{key}`。

```

public abstract class AbstractPrefixConfiguration implements Configuration {
    protected String id;
    protected String prefix;

    public AbstractPrefixConfiguration(String prefix, String id) {
        if (StringUtils.isNotEmpty(prefix) && !prefix.endsWith(".")) {
            this.prefix = prefix + ".";
        } else {
            this.prefix = prefix;
        }
        this.id = id;
    }

    @Override
    public Object getProperty(String key, Object defaultValue) {
        Object value = null;
        if (StringUtils.isNotEmpty(prefix)) {
            if (StringUtils.isNotEmpty(id)) {
                value = getInternalProperty(prefix + id + "." + key);
            }
            if (value == null) {
                value = getInternalProperty(prefix + key);
            }
        } else {
            value = getInternalProperty(key);
        }
        return value != null ? value : defaultValue;
    }
}

```

## SystemConfiguration

“ System.getProperty(key) 返回的是JVM进程的变量值

```

public class SystemConfiguration extends AbstractPrefixConfiguration {

    public SystemConfiguration(String prefix, String id) {
        super(prefix, id);
    }

    public SystemConfiguration() {
        this(null, null);
    }

    @Override
    public Object getInternalProperty(String key) {
        return System.getProperty(key);
    }

}

```

## EnvironmentConfiguration

“ System.getenv(key) 返回的是系统环境变量的值。

该实现中，如果使用入参 key 取到的值为空，则尝试将 key 中的 ":" 替换为 "", 转为形如 DUBBO [A-Z]{1,}\_[A-Z]{1,}...\_[A-Z]{1,} 的键，再重新获取配置数据。

```
public class EnvironmentConfiguration extends AbstractPrefixConfiguration {  
  
    public EnvironmentConfiguration(String prefix, String id) {  
        super(prefix, id);  
    }  
  
    public EnvironmentConfiguration() {  
        this(null, null);  
    }  
  
    @Override  
    public Object getInternalProperty(String key) {  
        String value = System.getenv(key);  
        if (StringUtils.isEmpty(value)) {  
            value = System.getenv(StringUtils.toOSStyleKey(key));  
        }  
        return value;  
    }  
  
}  
  
public final class StringUtils {  
    ...  
    public static String toOSStyleKey(String key) {  
  
        //DOT_REGEX = "\\\\.";  
        //UNDERLINE_SEPARATOR = "_";  
  
        key = key.toUpperCase().replaceAll(DOT_REGEX, UNDERLINE_SEPARATOR);  
        if (!key.startsWith("DUBBO_")) {  
            key = "DUBBO_" + key;  
        }  
        return key;  
    }  
    ...  
}
```

JAVA

## PropertiesConfiguration

该类的实现使用了 ConfigUtils.getProperty(key) 方法，由其负责加载工程中的 \*.properties 文件。

```
public class PropertiesConfiguration extends AbstractPrefixConfiguration {  
    private static final Logger logger =  
LoggerFactory.getLogger(PropertiesConfiguration.class);  
  
    public PropertiesConfiguration(String prefix, String id) {  
        super(prefix, id);  
    }  
  
    public PropertiesConfiguration() {  
        this(null, null);  
    }  
  
    @Override  
    public Object getInternalProperty(String key) {  
        return ConfigUtils.getProperty(key);  
    }  
}
```

## InmemoryConfiguration

Dubbo声明了一个 LinkedHashMap<String, String> 做内存配置的容器，用于存取键值对。

```

public class InmemoryConfiguration extends AbstractPrefixConfiguration {

    // stores the configuration key-value pairs
    private Map<String, String> store = new LinkedHashMap<>();

    public InmemoryConfiguration(String prefix, String id) {
        super(prefix, id);
    }

    public InmemoryConfiguration() {
        this(null, null);
    }

    @Override
    public Object getInternalProperty(String key) {
        return store.get(key);
    }

    /**
     * Add one property into the store, the previous value will be replaced if the key exists
     */
    public void addProperty(String key, String value) {
        store.put(key, value);
    }

    /**
     * Add a set of properties into the store
     */
    public void addProperties(Map<String, String> properties) {
        if (properties != null) {
            this.store.putAll(properties);
        }
    }

    /**
     * set store
     */
    public void setProperties(Map<String, String> properties) {
        if (properties != null) {
            this.store = properties;
        }
    }
}

```

## Environment

正如上文所言上述所有 Configuration 的实现，最终都是通过 Enviroment 类对外提供统一的用户界面。从上文的剖析已知他们都扩展自超类 AbstractPrefixConfiguration，支持 [{prefix} + "."][{id} + "."]{key} 前缀，便于分组管理配置信息，一个分组对应它的一个实例。这些实例需要使用 Map 做缓存管理，其Key的计算方式如下，key 的值为 ([{prefix}][{id}]["."] | "dubbo")：

```
private static String toKey(String prefix, String id) {  
    StringBuilder sb = new StringBuilder();  
    if (StringUtils.isNotEmpty(prefix)) {  
        sb.append(prefix);  
    }  
    if (StringUtils.isNotEmpty(id)) {  
        sb.append(id);  
    }  
  
    if (sb.length() > 0 && sb.charAt(sb.length() - 1) != '.') {  
        sb.append(".");  
    }  
    if (sb.length() > 0) {  
        return sb.toString();  
    }  
    return CommonConstants.DUBBO;  
}
```

所有 `AbstractPrefixConfiguration` 类型的配置实例都管理在 `ConcurrentHashMap<String, *>` 中，传入 `prefix` 和 `id` 便能方便地获取到缓存中的实例或者新建并加入到其中的实例，因而也可以将 `Enviroment` 看做是 `AbstractPrefixConfiguration` 类的创建工厂，其实现如下：

```

//①PropertiesConfiguration
private Map<String, PropertiesConfiguration> propertiesConfigs = new ConcurrentHashMap<>();
public PropertiesConfiguration getPropertiesConfig(String prefix, String id) {
    return propertiesConfigs.computeIfAbsent(toKey(prefix, id), k -> new
PropertiesConfiguration(prefix, id));
}

//②SystemConfiguration
private Map<String, SystemConfiguration> systemConfigs = new ConcurrentHashMap<>();
public SystemConfiguration getSystemConfig(String prefix, String id) {
    return systemConfigs.computeIfAbsent(toKey(prefix, id), k -> new
SystemConfiguration(prefix, id));
}

//③EnvironmentConfiguration
private Map<String, EnvironmentConfiguration> environmentConfigs = new ConcurrentHashMap<>();
public EnvironmentConfiguration getEnvironmentConfig(String prefix, String id) {
    return environmentConfigs.computeIfAbsent(toKey(prefix, id), k -> new
EnvironmentConfiguration(prefix, id));
}

//④InmemoryConfiguration → externalConfigurationMap
private Map<String, InmemoryConfiguration> externalConfigs = new ConcurrentHashMap<>();
private Map<String, String> externalConfigurationMap = new HashMap<>();
public InmemoryConfiguration getExternalConfig(String prefix, String id) {
    return externalConfigs.computeIfAbsent(toKey(prefix, id), k -> {
        InmemoryConfiguration configuration = new InmemoryConfiguration(prefix, id);
        configuration.setProperties(externalConfigurationMap);
        return configuration;
    });
}
public void setExternalConfigMap(Map<String, String> externalConfiguration) {
    this.externalConfigurationMap = externalConfiguration;
}
public Map<String, String> getExternalConfigurationMap() {
    return externalConfigurationMap;
}
public void updateExternalConfigurationMap(Map<String, String> externalMap) {
    this.externalConfigurationMap.putAll(externalMap);
}

//⑤InmemoryConfiguration → appExternalConfigurationMap
private Map<String, InmemoryConfiguration> appExternalConfigs = new ConcurrentHashMap<>();
private Map<String, String> appExternalConfigurationMap = new HashMap<>();
public InmemoryConfiguration getAppExternalConfig(String prefix, String id) {
    return appExternalConfigs.computeIfAbsent(toKey(prefix, id), k -> {
        InmemoryConfiguration configuration = new InmemoryConfiguration(prefix, id);
        configuration.setProperties(appExternalConfigurationMap);
        return configuration;
    });
}
public void setAppExternalConfigMap(Map<String, String> appExternalConfiguration) {
    this.appExternalConfigurationMap = appExternalConfiguration;
}
public Map<String, String> getAppExternalConfigurationMap() {
    return appExternalConfigurationMap;
}
public void updateAppExternalConfigurationMap(Map<String, String> externalMap) {

```

```
        this.appExternalConfigurationMap.putAll(externalMap);
    }
```

上面的源码最后两次使用的 `InmemoryConfiguration` 中，其配置容器并没有使用类中定义的 `LinkedHashMap`，而是采用了由外部创建的 `HashMap` 实例。

下一个章节中会提到的 `DynamicConfiguration` 动态配置接口类，一个微服务或应用中只会存在一个实例，`Environment` 也可以作为它的持有器，如下：

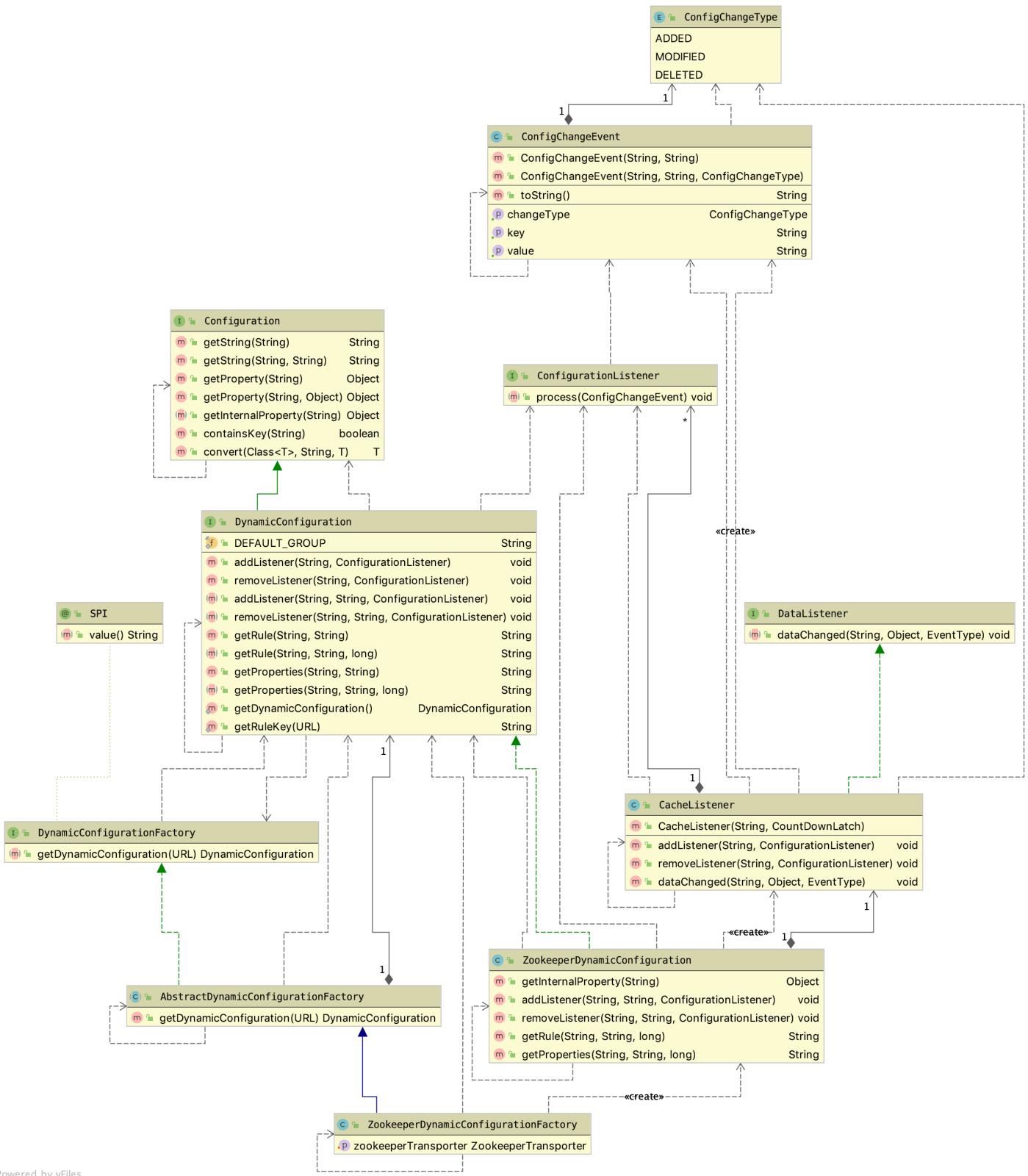
```
private Configuration dynamicConfiguration;                                JAVA

public Optional<Configuration> getDynamicConfiguration() {
    return Optional.ofNullable(dynamicConfiguration);
}

public void setDynamicConfiguration(Configuration dynamicConfiguration) {
    this.dynamicConfiguration = dynamicConfiguration;
}
```

## 动态配置

如图所示，Dubbo将动态配置设计成一个扩展的点，参考上文中关于SPI机制下的工程模式可知，实际意图的扩展点是 `DynamicConfiguration`，具体实现机制这里不再赘述。



Powered by yFiles

和 `DynamicConfiguration` 密切相关的另一个接口是 `ConfigurationListener`。动态配置中，由配置中心负责维护配置数据，当数据发生变更时，它会下发相关通知，由实现方负责响应这种变化，类似同步本地配置内容，其接口定义如下：

```

public interface ConfigurationListener {

    void process(ConfigChangeEvent event);

}

```

JAVA

类图中清晰可见，配置变化的类型有3种，新增、更新和删除。

## DynamicConfiguration

如下述源码所示，根据框架内部使用的场景它定义如下几个方法，定义的接口方法包括3方面：

1. 从配置中心获取治理规则或者配置项数据，支持超时和分组；
2. 针对特定配置或者治理规则增加或者删除监听器，也支持分组；

```
public interface DynamicConfiguration extends Configuration {  
    void addListener(String key, String group, ConfigurationListener listener);  
    void removeListener(String key, String group, ConfigurationListener listener);  
    String getRule(String key, String group, long timeout) throws IllegalStateException;  
    String getProperties(String key, String group, long timeout) throws IllegalStateException;  
}
```

JAVA

另外接口中还增加了一些 default 方法：1) 增删监听器时将分组设置为默认的 dubbo；2) 将超时设置为-1，获取配置或治理规则时一直等待；

```
String DEFAULT_GROUP = "dubbo";  
  
default void addListener(String key, ConfigurationListener listener) {  
    addListener(key, DEFAULT_GROUP, listener);  
}  
  
default void removeListener(String key, ConfigurationListener listener) {  
    removeListener(key, DEFAULT_GROUP, listener);  
}  
  
default String getRule(String key, String group) {  
    return getRule(key, group, -1L);  
}  
  
default String getProperties(String key, String group) throws IllegalStateException {  
    return getProperties(key, group, -1L);  
}
```

JAVA

另外 DynamicConfiguration 接口中还提供了一个 getDynamicConfiguration() 静态方法，它会首先尝试从 Environment 获取缓存在其中的 DynamicConfiguration 实例，若为null，则使用SPI机制获得 DynamicConfigurationFactory 工厂，并获取目标实例。

```
static DynamicConfiguration getDynamicConfiguration() {
    Optional<Configuration> optional = Environment.getInstance().getDynamicConfiguration();
    return (DynamicConfiguration) optional.orElseGet(() ->
        getExtensionLoader(DynamicConfigurationFactory.class)
            .getDefaultValue()
            .getDynamicConfiguration(null));
}

/**
 * The format is '{interfaceName}:[version]:[group]'
 *
 * @return
 */
static String getRuleKey(URL url) {
    return url.getColonSeparatedKey();
}
```

JAVA

## Zookeeper 配置中心实现

该章节的实现剖析请移步《【三】Zookeeper与Dubbo》，Zookeeper相关实现全部汇集在该文。

---

完结

# 【十六】Dubbo集群之目录服务

在接触到一个概念之前，我们需要搞清楚它到底有啥含义？是干啥的？只有整明白后，才能更快地基于它展开学习和研究。Dubbo这个RPC框架，其设计，用今天软件开发中流行的话术讲，是基于DDD领域驱动设计的，自然比较遵从业界约定俗成的认知，涉及不少相关领域方面的概念，比如框架层中每一分层的名称包括功能特性，在集群这个模块我们接触到的负载均衡、路由，以及本文需要展开详述的目录服务也是分布式软件系统中一个领域概念。

目录服务，有时也称作名字服务，下述是维基百科中关于它的描述，简单理解就是提供名称到资源值、服务、Action等的映射服务，比如被广为认知的DNS：

“ 目录 服务 务  
(<https://zh.wikipedia.org/wiki/%E7%9B%AE%E5%BD%95%E6%9C%8D%E5%8A%A1>)  
(英语：*Directory service*) 是一个储存、组织和提供信息访问服务的软件系统，在软件工程中，一个目录是指一组名字和值的映射。它允许根据一个给出的名字来查找对应的值，与词典相似。...名字服务(英语：*Name service*)是一个简单的目录服务，名字服务将一个网络资源的名字与它的网络地址进行映射。...目录服务是一种共享的基础信息服务，可用来定位、管理和组织通用项目和网络资源。

可见，有了目录服务，一个用户不必记住某个网络资源的物理地址，只需要提供这个网络资源的名字就可以找到它。

## 接口定义

前面两篇关于Dubbo集群实现剖析的文章中，已经说过：1) 负载均衡是客户端在一个微服务的所有可用服务引用实例中基于某个策略挑选一个作为处理RPC请求的对象；2) 路由是基于配置中心的路由覆盖规则，为当前RPC方法的匹配到目标微服务的所有满足条件的引用实例集合。

此前在关于RPC远程方法调用实现剖析中，有讲过，一个RPC方法请求体中，方法级别的信息入参及入参类型在 `Invocation` 中体现，而其引用微服务级别的信息元数据却是体现在URL配置总线中的。在注册中心的支持下，客户端并不需要指定其引用微服务的物理地址，简单配置后者的服务名称便可。也就是说需要通过名称获取所有的目标微服务的引用实例 `Invoker` 对象，而这正是Dubbo集群中 `Directory` 存在的价值。

接口定义源码如下：

```
public interface Directory<T> extends Node {  
    //get service type.  
    Class<T> getInterface();  
  
    //list invokers.  
    List<Invoker<T>> list(Invocation invocation) throws RpcException;  
}
```

JAVA

## NOTE

在一个微观的粒度，我们总是很难看清一件事物的本质，就像盲人摸象，把象腿当做柱子，如果想更快速获知对 `Directory` 的总体印象，请参考分组折叠处理的相关描述。

## AbstractDirectory

抽象基类 `AbstractDirectory` 是 `Directory<T>` 的模板实现，它定义了一些基本的行为规范。首先尽管 `Directory` 只服务于被引用的特定微服务，但其执行环境依然是高并发充满竞争的，一些公共的并发状态管理是必要的。

其实 `Directory` 在Dubbo中被当做一个组件模块对待，于注册中心而言，它是一个应用层次的客户端，承担着一些向框架上层提供服务的职责，因而有着自己的生命体征，这在分层设计体系中也是一个比较优秀的设计思想。

如下述源码，作为注册中心的应用客户端，如果它被销毁了，则被引用微服务的任一将要发生RPC调用行为的 `Invoker` 对象需要及时感知到这种状态的变化，`destroyed` 被声明为 `volatile`。

```
private volatile boolean destroyed = false; JAVA

public boolean isDestroyed() {
    return destroyed;
}

@Override
public void destroy() {
    destroyed = true;
}

@Override
public List<Invoker<T>> list(Invocation invocation) throws RpcException {
    if (destroyed) {
        throw new RpcException("Directory already destroyed .url: " + getUrl());
    }

    return doList(invocation);
}

protected abstract List<Invoker<T>> doList(Invocation invocation) throws RpcException;
```

再以 consumerUrl 为例，它于注册中心来说，是其应用客户端的关键标识，客户端用其观察注册中心的一个数据节点，可能因为某些变动，比如切换一个注册中心，框架上层会执行一些重新订阅操作，这时所有引用微服务的相关实例也需要第一时间感知到这一变化，避免在一个错误的节点上执行相关行为，因而 consumerUrl 也被声明成 volatile。

```

private final URL url;

private volatile URL consumerUrl;

public AbstractDirectory(URL url) {
    this(url, null);
}

public AbstractDirectory(URL url, RouterChain<T> routerChain) {
    this(url, url, routerChain);
}

public AbstractDirectory(URL url, URL consumerUrl, RouterChain<T> routerChain) {
    if (url == null) {
        throw new IllegalArgumentException("url == null");
    }

    if (url.getProtocol().equals(REGISTRY_PROTOCOL)) {
        Map<String, String> queryMap =
        StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
        this.url = url.addParameters(queryMap).removeParameter(MONITOR_KEY);
    } else {
        this.url = url;
    }

    this.consumerUrl = consumerUrl;
    setRouterChain(routerChain);
}

public URL getConsumerUrl() {
    return consumerUrl;
}

public void setConsumerUrl(URL consumerUrl) {
    this.consumerUrl = consumerUrl;
}

```

另外源码中的 `url` 表达的是啥呢？要知道在微服务框架中，引用微服务这个行为是要通过注册中心的。以Dubbo的尿性，涉及到数据传递和临时存取的都会借由一个URL数据对象，如 `registry://registry-host/org.apache.dubbo.registry.RegistryService?refer=URL.encode("consumer://consumer-host/com.foo.FooService?version=1.0.0")`，表示被引用微服务的URL数据被编码到了 `refer` 这个参数中，这导致相关特征参数的存取困难，因而构造函数中执行了解码处理。

其次 `Directory` 只是根据注册中心为当前客户列出所有目标被引用微服务的所有可用候选集，需要进一步应用客户端本身定义的一些筛选过滤规则，也就是路由处理，避免PRC调用时的不必要开销。

```
protected RouterChain<T> routerChain;

public RouterChain<T> getRouterChain() {
    return routerChain;
}

public void setRouterChain(RouterChain<T> routerChain) {
    this.routerChain = routerChain;
}

protected void addRouters(List<Router> routers) {
    routers = routers == null ? Collections.emptyList() : routers;
    routerChain.addRouters(routers);
}
```

AbstractDirectory 类的声明有如下一段注释也对该特性加以说明了：

“Invoker list returned from this Directory’s list method have been filtered by Routers”

## Directory 的实现

本文件将只着重该接口实现 RegistryDirectory 的分析，其业务逻辑尽管不是太复杂，但结构不是那么明晰，关联内容太多，试图读懂源码剖为困难。但是它的主体接口实现逻辑却出奇的简单，如下源码，禁用时直接抛错，带分组特性时不执行路由筛选处理，尽量尝试应用路由链，若出错则直接返回候选集。

```

@Override
public List<Invoker<T>> doList(Invocation invocation) {
    if (forbidden) {
        // 1. No service provider 2. Service providers are disabled
        throw new RpcException(RpcException.FORBIDDEN_EXCEPTION,
            "No provider available from registry " +
            getUrl().getAddress() + " for service " +
            + getConsumerUrl().getServiceKey() + " on consumer " +
            NetUtils.getLocalHost() + " use dubbo version " +
            + Version.getVersion() +
            ", please check status of providers(disabled" +
            + ", not registered or in blacklist).");
    }

    if (multiGroup) {
        return this.invokers == null ? Collections.emptyList() : this.invokers;
    }

    List<Invoker<T>> invokers = null;
    try {
        invokers = routerChain.route(getConsumerUrl(), invocation);
    } catch (Throwable t) {
        logger.error("Failed to execute router: " + getUrl() + ", cause: " +
        t.getMessage(), t);
    }

    return invokers == null ? Collections.emptyList() : invokers;
}

```

实际该类作为注册中心的客户端，几乎大部分业务代码都在同步注册中心数据，决定是否刷新 List<Invoker<T>> 类型的候选集，刷新过程涉及到多个部件的联动，比较复杂，下述章节将逐步展示对其的实现剖析。

## 同步覆写规则

产生一个服务实例需要基于综合的各种信息，一般它们被携带在配置总线一个URL实例中。然而总线中的信息来源多样，有程序运行时所产生的，也有源于操作系统环境变量、本地文件配置和内存临时配置，还有注册中心的动态配置（包括元数据、覆写规则、注册信息等）。能及时汇总各个维度的信息是保证服务的高可用性的前提，尤其是最后一种跨机配置数据，穿越复杂的网络链路，还需要保障在各个微服务间的视图一致性，因而相比而言存取更加困难，不得不依托于 Zookeeper 这样的第三方分布式协调中间件做配置的同步处理。

本文所讲的配置同步，主要是指覆写规则的同步，它分为两部分，一部分来源于动态配置中心，另一部分则来源于注册中心，尽管有时动态配置中心和注册中心实际上是同一个服务。

### 从动态配置中心同步 ACL

这种方式的同步主要是基于 AbstractConfiguratorListener 类实现。

关于如何实现配置同步不是本文重点，详情请移步《Dubbo 配置管理》，如果深入该文你就会发现动态配置的同步实际上有两种，一种是拉取模式，还一种是基于事件的推送模式，文章还会引导你移步至《Zookeeper 与 Dubbo》，它会告诉你推送模式是如何由第三方客户端驱动框架上层注入的相关监听器同步动态配置的。

如下源码，`initWith(key)` 方法中使用的是拉模式，首先使用用户接口 `Environment` 或者使用 SPI 加载获取动态配置的客户端——`DynamicConfiguration` 实例，然后就 `key` 所对应的 `path` 加入当前 `ConfigurationListener` 监听器（由 `this` 所在类实现），接着使用客户端取得原生的覆写规则，最后由该规则生成一个可用于覆写本地 URL 数据的 `Configurator` 列表。

```
protected final void initWith(String key) {  
    DynamicConfiguration dynamicConfiguration =  
        DynamicConfiguration.getDynamicConfiguration();  
  
    dynamicConfiguration.addListener(key, this);  
  
    String rawConfig = dynamicConfiguration.  
        getRule(key, DynamicConfiguration.DEFAULT_GROUP);  
  
    if (!StringUtils.isEmpty(rawConfig)) {  
        genConfiguratorsFromRawRule(rawConfig);  
    }  
}
```

显然 `initWith(key)` 方法是用在 `AbstractConfiguratorListener` 的实现类的构造函数中的，实际含义是生成动态配置客户端后，立马将覆写规则全量拉取到本地，这是一个必须操作，因推模式需要等到响应的 `path` 发生了变化。

推模式是基于事件回调机制的，如下：在收到 `ConfigChangeType.DELETED` 事件时，需要把对应的覆写规则处理器列表清理掉，其它事件则重新生成该列表；随后调用有子类覆写的 `notifyOverrides()` 方法告知相关方，覆写规则已经发生变动，请同步变更相应的 URL 数据。后面这个同步通知一般也都是基于回调相应提供的监听器实现的。

```
public abstract class AbstractConfiguratorListener implements ConfigurationListener {  
    JAVA  
  
    protected List<Configurator> configurators = Collections.emptyList();  
  
    @Override  
    public void process(ConfigChangeEvent event) {  
        if (logger.isInfoEnabled()) {  
            logger.info("Notification of overriding rule, change type is: "  
                + event.getChangeType() + ", raw config content is:\n " +  
event.getValue());  
        }  
  
        if (event.getChangeType().equals(ConfigChangeType.DELETED)) {  
            configurators.clear();  
        } else {  
            if (!genConfiguratorsFromRawRule(event.getValue())) {  
                return;  
            }  
        }  
  
        notifyOverrides();  
    }  
  
    protected abstract void notifyOverrides();  
  
  
    public List<Configurator> getConfigurators() {  
        return configurators;  
    }  
  
    public void setConfigurators(List<Configurator> configurators) {  
        this.configurators = configurators;  
    }  
  
    ...  
}
```

上述可以看到无论是推模式还是拉模式，都会调用如下实现的  
`genConfiguratorsFromRawRule(rawConfig)`方法，若发生异常，则返回 false，对应上述它被调用的逻辑就是直接返回。相关细节请移步《Dubbo 配置管理》，这里不再赘述。

```
private boolean genConfiguratorsFromRawRule(String rawConfig) {  
    boolean parseSuccess = true;  
    try {  
        configurators = Configurator.toConfigurators(ConfigParser  
            .parseConfigurators(rawConfig)).orElse(configurators);  
    } catch (Exception e) {  
        logger.error("Failed to parse raw dynamic config and" +  
            "it will not take effect, the raw config is: " + rawConfig, e);  
        parseSuccess = false;  
    }  
    return parseSuccess;  
}
```

JAVA

## 配置中心缺席

然而，在没有单独设置配置中心时，对应的 ConfigurationListener 是不会发生作用的。通过仔细查看扩展点 DynamicConfigurationFactory 的实现就会发现有不对劲的地方，如下：

```
@SPI("nop")  
public interface DynamicConfigurationFactory {  
  
    DynamicConfiguration getDynamicConfiguration(URL url);  
}
```

JAVA

它标注了 @SPI("nop")，也即生成的代理类会将行为委托给如下实现类，从 NopDynamicConfiguration 实现来看，啥事也没干。

```
public class NopDynamicConfigurationFactory extends AbstractDynamicConfigurationFactory JAVA
{
    @Override
    protected DynamicConfiguration createDynamicConfiguration(URL url) {
        return new NopDynamicConfiguration(url);
    }
}

public class NopDynamicConfiguration implements DynamicConfiguration {

    public NopDynamicConfiguration(URL url) {}

    @Override
    public Object getInternalProperty(String key) {}

    @Override
    public void addListener(String key, String group, ConfigurationListener listener) {}

    @Override
    public void removeListener(String key, String group, ConfigurationListener listener)
    {}

    @Override
    public String getRule(String key, String group, long timeout) throws
IllegalStateException {
        return null;
    }

    @Override
    public String getProperties(String key, String group, long timeout) throws
IllegalStateException {
        return null;
    }
}
```

## ACL 在客户端的应用

上文有关目录服务的介绍中，很明显它是为客户端提供服务的，是客户端微服务发现机制的实现。覆写规则根据作用范围的不同，分为应用级别和微服务级别，因此在 `RegistryDirectory` 实现中，有两个 `AbstractConfiguratorListener` 覆写规则监听器实现，分别是 `ConsumerConfigurationListener` 和 `ReferenceConfigurationListener`，二者的代码很简短，都被申明为了静态的私有内部类，需要结合上下文理解。

### ConsumerConfigurationListener

从 `Directory` 接口的声明中可知，每一个被引用微服务对应会拥有它的一个实例。而 `ConsumerConfigurationListener` 实例在整个应用中也就声明了一个实例全局的实例，独此一份，因所有实例需要关注应用基本的覆写规则的变化，它们自身的 `subscribe(url)` 方法被调用时，自己就会被作为订阅者增添一个监听器。它们监听的节点都是 “`/({namespace} | dubbo)/config/dubbo/{app}.configurators`”。

当覆盖规则有变化时，本地接受到通知后，便刷新应用中的当前所有微服务的所有引用实例。

```
private static class ConsumerConfigurationListener extends AbstractConfiguratorListener {  
    List<RegistryDirectory> listeners = new ArrayList<>();  
  
    ConsumerConfigurationListener() {  
        this.initWith(ApplicationModel.getApplication() + CONFIGURATORS_SUFFIX);  
    }  
  
    void addNotifyListener(RegistryDirectory listener) {  
        this.listeners.add(listener);  
    }  
  
    @Override  
    protected void notifyOverrides() {  
        listeners.forEach(listener -> listener.refreshInvoker(Collections.emptyList()));  
    }  
}  
  
private static final ConsumerConfigurationListener CONSUMER_CONFIGURATION_LISTENER  
= new ConsumerConfigurationListener();  
  
public void subscribe(URL url) {  
    ...  
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);  
    ...  
}
```

## ReferenceConfigurationListener

同上述不同的是，一个被引用的微服务调用 `subscribe(url)` 时，会为该微服务分配一个 `ReferenceConfigurationListener` 对象，而它监听的节点仅限于自身相关，对应动态配置项为 “`/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators`”。

当覆盖规则有变化时，本地接受到通知后，便刷新应用中的当前 `RegistryDirectory` 实例所对应微服务。

JAVA

```

private ReferenceConfigurationListener serviceConfigurationListener;

private static class ReferenceConfigurationListener extends AbstractConfiguratorListener
{
    private RegistryDirectory directory;
    private URL url;

    ReferenceConfigurationListener(RegistryDirectory directory, URL url) {
        this.directory = directory;
        this.url = url;
        this.initWith(DynamicConfiguration.getRuleKey(url) + CONFIGURATORS_SUFFIX);
    }

    @Override
    protected void notifyOverrides() {
        // to notify configurator/router changes
        directory.refreshInvoker(Collections.emptyList());
    }
}

public void subscribe(URL url) {
    ...
    serviceConfigurationListener = new ReferenceConfigurationListener(this, url);
    ...
}

```

## 从注册中心同步

因注册中心和动态配置中心可以源于同一服务，而同步他们的数据使用都是订阅观察模式，因此在没有另外提供单独的动态配置中心时，可以以注册中心客户端的身份去同步覆写规则。如果二者同时提供了，就会汇总来自他们的覆写规则，一起发生作用。和其它客户端订阅操作一样，只需配置URL数据，调用如下 `subscribe(url)` 方法即可。

JAVA

```

public void subscribe(URL url) {
    setConsumerUrl(url);
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);
    serviceConfigurationListener = new ReferenceConfigurationListener(this, url);
    registry.subscribe(url, this);
}

```

显然上述代码的目的很明显，就是针对某个特定的被引用微服务做订阅处理（`url` 指定了微服务提供者），等待注册中心相关的通知，在通知中执行某些同步处理。源码中的最后一行调用了 `org.apache.dubbo.registry.RegistryService#subscribe(URL url, NotifyListener listener)`，显然 `this` 所在的类实现了 `NotifyListener` 接口。明白了这点，咱可以验证其实现方法继续往下探究。

```

@Override
public synchronized void notify(List<URL> urls) {
    Map<String, List<URL>> categoryUrls = urls.stream()
        .filter(Objects::nonNull)
        .filter(this::isValidCategory)
        .filter(this::isNotCompatibleFor26x)
        .collect(Collectors.groupingBy(url -> {
            if (UrlUtils.isConfigurator(url)) {
                return CONFIGURATORS_CATEGORY;
            } else if (UrlUtils.isRoute(url)) {
                return ROUTERS_CATEGORY;
            } else if (UrlUtils.isProvider(url)) {
                return PROVIDERS_CATEGORY;
            }
            return "";
        }));
}

List<URL> configuratorURLs = categoryUrls.getOrDefault(
    CONFIGURATORS_CATEGORY, Collections.emptyList());

this.configurators = Configurator.toConfigurators(
    configuratorURLs).orElse(this.configurators);

List<URL> routerURLs = categoryUrls.getOrDefault(
    ROUTERS_CATEGORY, Collections.emptyList());

toRouters(routerURLs).ifPresent(this::addRouters);

// providers
List<URL> providerURLs = categoryUrls.getOrDefault(
    PROVIDERS_CATEGORY, Collections.emptyList());

refreshOverrideAndInvoker(providerURLs);
}

```

首先上述代码被修饰了方法级别的 `synchronized`，表示针对某个被引用微服务的当前 `Directory` 对象，它本身作为互斥锁，锁住整个 `notify(urls)` 方法，避免前面一个通知还没响应完，后一个就开始执行了。

然后方法体，熟悉 Java 8 的同学会倍感亲切，先将入参做过滤处理，然后分组，按 URL 数据类型分为覆盖规则、路由规则和微服务实例集，最后按分组做相应的业务逻辑处理：

1. 将覆盖规则转换为 `Configurator` 覆盖规则处理器；
2. 将路由规则数据转换为 `Router` 路由器；
3. 将得到的当前对应引用微服务的所有可用实例，使用 `Configurator` 做覆盖刷新处理；

在继续往下探讨前，我们先看看上述源码中出现的两个有关过滤的函数。`isValidCategory(url)` 的大致意思是需要满足条件 `"route" == url.protocol || (url["category"] | "providers") ∈ ["routers", "providers", "configurators", "dynamicconfigurators", "appdynamicconfigurators"]`，而 `方法则要求 `url["compatible_config"]` 的值为空。`

```
private boolean isValidCategory(URL url) {
    String category = url.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY);
    if ((ROUTERS_CATEGORY.equals(category) || ROUTE_PROTOCOL.equals(url.getProtocol())))
    ||
        PROVIDERS_CATEGORY.equals(category) ||
        CONFIGURATORS_CATEGORY.equals(category) ||
DYNAMIC_CONFIGURATORS_CATEGORY.equals(category) ||
        APP_DYNAMIC_CONFIGURATORS_CATEGORY.equals(category)) {
    return true;
}
logger.warn("Unsupported category " + category + " in notified url: " + url + " from
registry " +
    getUrl().getAddress() + " to consumer " + NetUtils.getLocalHost());
return false;
}

private boolean isNotCompatibleFor26x(URL url) {
    return StringUtils.isEmpty(url.getParameter(COMPATIBLE_CONFIG_KEY));
}
```

另外被调用的 `toRouters(urls)` 方法也值得一提，如下源码在根据URL数据获取 Router 实例时，如果含有 `url["router"]` 参数，则会设 `url.protocol = url["router"]`，原因是 `getRouter(url)` 方法含有 `@Adaptive("protocol")` 声明，SPI机制会在动态生成的代理类中，先使用 `url.protocol` 的值作为 key 去加载它所映射目标 Router 类的实例，然后将 `getRouter(url)` 方法委托给它执行。

```
private static final RouterFactory ROUTER_FACTORY =
    ExtensionLoader.getExtensionLoader(RouterFactory.class).getAdaptiveExtension();

private Optional<List<Router>> toRouters(List<URL> urls) {
    if (urls == null || urls.isEmpty()) {
        return Optional.empty();
    }

    List<Router> routers = new ArrayList<>();
    for (URL url : urls) {
        if (EMPTY_PROTOCOL.equals(url.getProtocol())) {
            continue;
        }
        String routerType = url.getParameter(ROUTER_KEY);
        if (routerType != null && routerType.length() > 0) {
            url = url.setProtocol(routerType);
        }
        try {
            Router router = ROUTER_FACTORY.getRouter(url);
            if (!routers.contains(router)) {
                routers.add(router);
            }
        } catch (Throwable t) {
            logger.error("convert router url to router error, url: " + url, t);
        }
    }
    return Optional.of(routers);
}
```

由 `notify(urls)` 方法体中的最后的 `refreshOverrideAndInvoker(urls)` 调用语句，我们会跟踪进入如下 `overrideDirectoryUrl()` 方法，它是我们此刻关注的重点，

```

private void overrideDirectoryUrl() {
    this.overrideDirectoryUrl = directoryUrl;
    List<Configurator> localConfigurators = this.configurators;
    doOverrideUrl(localConfigurators);

    List<Configurator> localAppDynamicConfigurators =
        CONSUMER_CONFIGURATION_LISTENER.getConfigurators();
    doOverrideUrl(localAppDynamicConfigurators);

    if (serviceConfigurationListener != null) {
        List<Configurator> localDynamicConfigurators =
            serviceConfigurationListener.getConfigurators();
        doOverrideUrl(localDynamicConfigurators);
    }
}

private void doOverrideUrl(List<Configurator> configurators) {
    if (CollectionUtils.isNotEmpty(configurators)) {
        for (Configurator configurator : configurators) {
            this.overrideDirectoryUrl = configurator.configure(overrideDirectoryUrl);
        }
    }
}

```

上述源码中汇总了 3 种覆盖规则：1）来自注册中心的；2）来自配置中心的针对当前应用的；3）来自配置中心的针对当前被引用微服务的。将所有这些覆盖规则处理器按顺序在 `overrideDirectoryUrl` 这条 URL 类型的数据均应用一遍，可见来自注册中心的覆盖规则优先级更高。

`overrideDirectoryUrl` 最初的样子，也就是 `directoryUrl` 是怎么样的？下面我带你一步步去搜索有关它的一些端倪，先看看如下构造函数，入参 `serviceType` 对应着当前被引用微服务的接口类型，第二个入参 `url` 表示的是当前客户端如何通过注册中心引用目标微服务，被引用微服务的相关元数据被包含在 `refer` 参数中，如 `registry://registry-host/org.apache.dubbo.registry.RegistryService?`

```
refer=URL.encode("consumer://consumer-host/com.foo.FooService?version=1.0.0")。
```

也就说这里的 `serviceKey` 表示的是注册中心的客户端服务，如这里的 `org.apache.dubbo.registry.RegistryService`。

```

public RegistryDirectory(Class<T> serviceType, URL url) {
    super(url);
    if (serviceType == null) {
        throw new IllegalArgumentException("service type is null.");
    }
    if (url.getServiceKey() == null || url.getServiceKey().length() == 0) {
        throw new IllegalArgumentException("registry serviceKey is null.");
    }
    this.serviceType = serviceType;
    this.serviceKey = url.getServiceKey();

    //①
    this.queryMap = StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
    this.overrideDirectoryUrl = this.directoryUrl = turnRegistryUrlToConsumerUrl(url);

    String group = directoryUrl.getParameter(GROUP_KEY, "");
    this.multiGroup = group != null && (ANY_VALUE.equals(group) || group.contains(","));
}

```

代码①处是最初生成 `directoryUrl` 的地方，做的事情不多，简单的将 `refer` 参数解码出来转入 `queryMap` 容器，保留入参 `url` 中的除参数的其它部分，随后在其后附上 `queryMap` 中的所有参数，使用如下方法生成了目标 `directoryUrl`，从方法名称可以看出，它的目的是将 `registry url` 转换成 `consumer url`。

```

private URL turnRegistryUrlToConsumerUrl(URL url) {
    // save any parameter in registry that will be useful to the new url.
    String isDefault = url.getParameter(DEFAULT_KEY);
    if (StringUtils.isNotEmpty(isDefault)) {
        queryMap.put(REGISTRY_KEY + "." + DEFAULT_KEY, isDefault);
    }
    return URLBuilder.from(url)
        .setPath(url.getServiceInterface())
        .clearParameters()
        .addParameters(queryMap)
        .removeParameter(MONITOR_KEY)
        .build();
}

```

便为理解，特地通过调试找来了一个实际的例子，如下：

```
//url (registry url)
zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=demo-
consumer&dubbo=2.0.2&pid=63267&refer=application%3Ddemo-
consumer%26check%3Dtrue%26dubbo%3D2.0.2%26interface%3Dorg.apache.dubbo.samples.basic.api
.DemoService%26lazy%3Dfalse%26methods%3DsayHello%26pid%3D63267%26register.ip%3D192.168.0
.6%26release%3D2.7.3%26side%3Dconsumer%26sticky%3Dfalse%26timestamp%3D1572445953542&rele
ase=2.7.3&timestamp=1572445959013

//directoryUrl (consumer url)
zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=demo-
consumer&check=true&dubbo=2.0.2&interface=org.apache.dubbo.samples.basic.api.DemoService
&lazy=false&methods=sayHello&pid=63267&register.ip=192.168.0.6&release=2.7.3&side=consum
er&sticky=false&timestamp=1572445953542
```

## 微服务引用实例刷新

在花费了大量篇幅剖析覆写规则同步后，终于轮到本文的主角登场了——微服务引用实例的刷新处理。以水力发电打比方，如果说前面这个覆写规则处理是将水库中的水引入的话，那后面这个刷新处理就相当于将发电机获得的电能输入到储能设备，显然中间还有个发电机将水的势能转换为电能的过程，“势能 → 电能”这个转换过程，对应的由URL数据到 `Invoker` 对象的转换处理，这是一个复杂的过程，详情请移步至《Dubbo微服务注册》。尽管“电能到储能设备的输送”这个过程很简单，但涉及的细节也挺多，下面我们将按粒度由小及大、从分到总来剖析整个过程。

### URL数据合入

实际上这是就单个被引用微服务的URL数据应用覆写规则处理器，源码中多次出现 `override > -D > Consumer > Provider` 这一注释，也就说各种覆写规则的应用是有优先级的，Directory URL 覆写处理中已经由提及过。废话不多说，先看如下 `mergeUrl(providerUrl)` 的实现，为了不致信息损失，下面这段代码我们保留了所有注释。

```

//Merge url parameters. the order is: override > -D >Consumer > Provider
private URL mergeUrl(URL providerUrl) {
    providerUrl = ClusterUtils.mergeUrl(providerUrl, queryMap); // Merge the consumer
    side parameters

    providerUrl = overrideWithConfigurator(providerUrl);

    providerUrl = providerUrl.addParameter(Constants.CHECK_KEY, String.valueOf(false));
    // Do not check whether the connection is successful or not, always create Invoker!

    // The combination of directoryUrl and override is at the end of notify, which can't
    be handled here
    this.overrideDirectoryUrl =
    this.overrideDirectoryUrl.addParametersIfAbsent(providerUrl.getParameters()); // Merge
    the provider side parameters

    if ((providerUrl.getPath() == null || providerUrl.getPath()
        .length() == 0) && DUBBO_PROTOCOL.equals(providerUrl.getProtocol())) { // 
    Compatible version 1.0
        //fix by tony.chenl DUBBO-44
        String path = directoryUrl.getParameter(INTERFACE_KEY);
        if (path != null) {
            int i = path.indexOf('/');
            if (i >= 0) {
                path = path.substring(i + 1);
            }
            i = path.lastIndexOf(':');
            if (i >= 0) {
                path = path.substring(0, i);
            }
            providerUrl = providerUrl.setPath(path);
        }
    }
    return providerUrl;
}

```

`providerUrl` 是从注册中心通过事件回调同步到本机的，它代表一个被引用微服务中的其中一个实例，就单个实例而言它有可能是朝不保夕的，突然每个事件通知，它就不见了，本地的具体应对情况，下文会提及。

来看看源码中的总体步骤，先合入由当前应用引用服务时传入的客户端参数，然后就该 `providerUrl` 按优先级先后应用所有的覆写规则，最后将来自服务端的参数合入到 `overrideDirectoryUrl`。

另外还做了兼容处理，在 dubbo 1.0 这个版本中，允许注册中心同步下来的 `url.path` 为空，这时需要将 `providerUrl` 的 `path` 设置为服务接口的名称，它是从 `url["interface"]` 中取得的。但是它的字符串形式可能比较复杂，形如 `[sth + "/" + interfaceName] ":" + sth2`，截取的是 `{interfaceName}` 这一部分。

该章节剩下的代码是关于合入覆写规则的，可以参考 Directory URL 覆写处理对照理解。

```

JAVA
private URL overrideWithConfigurator(URL providerUrl) {
    // override url with configurator from "override://" URL for dubbo 2.6 and before
    providerUrl = overrideWithConfigurators(this.configurators, providerUrl);

    // override url with configurator from configurator from "app-name.configurators"
    providerUrl =
overrideWithConfigurators(CONSUMER_CONFIGURATION_LISTENER.getConfigurators(),
providerUrl);

    // override url with configurator from configurators from "service-
name.configurators"
    if (serviceConfigurationListener != null) {
        providerUrl =
overrideWithConfigurators(serviceConfigurationListener.getConfigurators(), providerUrl);
    }

    return providerUrl;
}

private URL overrideWithConfigurators(List<Configurator> configurators, URL url) {
    if (CollectionUtils.isNotEmpty(configurators)) {
        for (Configurator configurator : configurators) {
            url = configurator.configure(url);
        }
    }
    return url;
}

```

## 候选集过滤处理

微服务开发中，为了提高可用性，往往一个微服务会部署多个实例。也就是说，注册中心会在数据同步时，为一个可用微服务返回这一到多个可用实例的URL数据，这时就需要系统地对他们进行“URL数据 → Invoker对象”的转换处理。这中间还得更具客户端一些参数做一些过滤处理，该功能统一实现 in `toInvokers(urls)` 方法中，方法体比较长，我们拆开分析。

一般而言一个微服务提供者会实现多种类型的通信协议支持，尽可能满足接入客户端的风格喜好和能力差异，可能会同一个微服务的多个实例各自支持不同协议。接入的客户端若指定了自己所能接受的通讯协议支持集——`url.protocol` 参数（支持多个时以“,” 分割），如果一个被引用微服务实例并不支持该协议，显然这个实例就不应该在候选集中。这部分逻辑体现在循环体中对单个实例如下处理上：

```

//TAG: 根据协议支持过滤候选集
private Map<String, Invoker<T>> toInvokers(List<URL> urls) {
    ...
    // If protocol is configured at the reference side, only the matching protocol is
    selected
    if (queryProtocols != null && queryProtocols.length() > 0) {
        boolean accept = false;
        String[] acceptProtocols = queryProtocols.split(",");
        for (String acceptProtocol : acceptProtocols) {
            if (providerUrl.getProtocol().equals(acceptProtocol)) {
                accept = true;
                break;
            }
        }
        if (!accept) {
            continue;
        }
    }
    if (EMPTY_PROTOCOL.equals(providerUrl.getProtocol())) {
        continue;
    }
    if (!ExtensionLoader.getExtensionLoader(Protocol.class)
        .hasExtension(providerUrl.getProtocol())) {
        logger.error(new IllegalStateException("Unsupported protocol "
            + providerUrl.getProtocol() + " in notified url: " + providerUrl
            + " from registry " + getUrl().getAddress() +
            " to consumer " + NetUtils.getLocalHost() + ", supported protocol: " +
            ExtensionLoader.getExtensionLoader(Protocol.class).getSupportedExtensions()));
        continue;
    }
    ...
}

```

另外如果一个服务实例被标记为 `url["disabled"] = true` 或 `url["enabled"] = false`，那么表示它因为一些特殊原因被禁用了，这个实例也会被排除在候选集之外。如下所示，一个服务实例，若本地缓存不存在时，需要将服务实例的URL数据转化为Invoker对象加入到本地候选集缓存，这时被禁用的实例便被略过了：

```

//TAG: 到Invoker对象转换时略过被禁用实例
private Map<String, Invoker<T>> toInvokers(List<URL> urls) {
    ...
    try {
        boolean enabled = true;
        if (url.hasParameter(DISABLED_KEY)) {
            enabled = !url.getParameter(DISABLED_KEY, false);
        } else {
            enabled = url.getParameter(ENABLED_KEY, true);
        }
        if (enabled) {
            invoker = new InvokerDelegate<>(protocol.refer(serviceType, url), url,
providerUrl);
        }
    } catch (Throwable t) {
        logger.error("Failed to refer invoker for interface:"
                + serviceType + ",url:( " + url + ")" + t.getMessage(), t);
    }
    if (invoker != null) { // Put new invoker in cache
        newUrlInvokerMap.put(key, invoker);
    }
}

...
}

```

最后再总体的看下 `toInvokers(urls)` 方法的实现，如下，步骤已经很明了，遍历从注册中心同步的目标微服务的所有服务实例的 URL 数据，首先将本地客户端所不支持的通讯协议的实例剔除；然后在当前 `providerUrl` 上应用所有最近同步的覆写规则得到最新的 URL 视图 key，并利用 key 结合 Set 集合特性做排重处理，对于重复的实例数据直接忽略处理；最后将根据 key 找不到的（可能已经缓存在本地）且没被禁用标识的实例转换为 `Invoker` 实例加入。

```

private Map<String, Invoker<T>> toInvokers(List<URL> urls) {
    Map<String, Invoker<T>> newUrlInvokerMap = new HashMap<>();
    if (urls == null || urls.isEmpty()) {
        return newUrlInvokerMap;
    }
    Set<String> keys = new HashSet<>();
    String queryProtocols = this.queryMap.get(Protocol_KEY);
    for (URL providerUrl : urls) {
        ...//TAG: 根据协议支持过滤候选集

        URL url = mergeUrl(providerUrl);

        String key = url.toFullString();
        if (keys.contains(key)) {
            continue;
        }
        keys.add(key);

        Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap;
        Invoker<T> invoker = localUrlInvokerMap == null ? null :
        localUrlInvokerMap.get(key);
        if (invoker == null) { // Not in the cache, refer again

            ...//TAG: 到Invoker对象转换时略过被禁用实例

        } else {
            newUrlInvokerMap.put(key, invoker);
        }
    }
    keys.clear();
    return newUrlInvokerMap;
}

```

源码中 `localUrlInvokerMap` 相当于新申请了一个指针，指向 `this.urlInvokerMap` 指针所指向某个 `Map<String, Invoker<T>>` 容器，因为 `this.urlInvokerMap` 后面可能会指向新的容器。

一个服务实例的 URL 数据在前后两次事件通知中应用覆写规则后，其值可能保持一样，也有可能因为客户端的一些原因不一样。前者会被直接加入到 `newUrlInvokerMap` 这个新的容器中，而后者对应的 `Invoker` 实例可能已经缓存在 `localUrlInvokerMap` 这个老的容器中，只是因为键发生变化，找不到了，这时只要对应应用了覆写规则的 URL 数据没有被标识禁用，便直接做重新引用处理，加入到 `newUrlInvokerMap` 中。也就是说 Dubbo 会不管三七二十一，一个服务实例，其 URL 数据在应用覆写规则后，只要发生变化，便会对其重新引用。

## 本地服务实例缓存清理

从上述章节的实现剖析来看，被引用了的微服务实例会被缓存到本地。然而，因为某些原因当前微服务的多个实例中，某些个实例可能会变得不可用，比如运维人员施加了下线处理操作，获得开发人员有目的地禁用某些在线的实例，以便在线排查 bug，或者是当前应用通过动态的写入一些覆写规则过滤掉某些实例。

当注册中心的客户端通过事件回调同步到某个微服务已经不存在相应的实例时，便会经由监听器 `notify(url,listener,url)` 给监听方只含有一条数据 `urls` 列表，其中 URL 数据的 `url.protocol` 为空，这时就需要执行如下所有缓存中微服务引用实例的清理处理，简直是毁灭性地。

```
private void destroyAllInvokers() {  
    Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference  
    if (localUrlInvokerMap != null) {  
        for (Invoker<T> invoker : new ArrayList<>(localUrlInvokerMap.values())) {  
            try {  
                invoker.destroy();  
            } catch (Throwable t) {  
                logger.warn("Failed to destroy service " + serviceKey  
                           + " to provider " + invoker.getUrl(), t);  
            }  
        }  
        localUrlInvokerMap.clear();  
    }  
    invokers = null;  
}
```

源码中，`localUrlInvokerMap` 指针所指向的容器中的实例一一调用了 `destroy()` 做销毁处理，最后将所有实例移除。此间 `urlInvokerMap` 的指向有可能发生变化。

另外，如候选集过滤这一章节的最后所介绍的，因为覆写规则有更新，导致某些实例因为应用它们后得到一个全新的 key `url.toFullString()`，这时 Dubbo 会做重新引用处理，实际上就是产生了一个新的 `Invoker` 实例，而老的实例实际上还缓存于内存中，这时也需要配合一些清理操作，主要是调用 `invoker.destroy()`。如下，其处理就是比较新老两个 `Map<String, Invoker<T>>` 集合，如果新集合中为空直接调用 `destroyAllInvokers()`，否则会先筛选出新的集合中不存在但老的集合中存在的实例，然后再逐个给 `destroy()` 并做回收内存。

```

private void destroyUnusedInvokers(Map<String, Invoker<T>> oldUrlInvokerMap,
    Map<String, Invoker<T>> newUrlInvokerMap) {
    if (newUrlInvokerMap == null || newUrlInvokerMap.size() == 0) {
        destroyAllInvokers();
        return;
    }
    // check deleted invoker
    List<String> deleted = null;
    if (oldUrlInvokerMap != null) {
        Collection<Invoker<T>> newInvokers = newUrlInvokerMap.values();
        for (Map.Entry<String, Invoker<T>> entry : oldUrlInvokerMap.entrySet()) {
            if (!newInvokers.contains(entry.getValue())) {
                if (deleted == null) {
                    deleted = new ArrayList<>();
                }
                deleted.add(entry.getKey());
            }
        }
    }

    if (deleted != null) {
        for (String url : deleted) {
            if (url != null) {
                Invoker<T> invoker = oldUrlInvokerMap.remove(url);
                if (invoker != null) {
                    try {
                        invoker.destroy();
                        if (logger.isDebugEnabled()) {
                            logger.debug("destroy invoker[" + invoker.getUrl() + "]"
success. ");
                        }
                    } catch (Exception e) {
                        logger.warn("destroy invoker[" + invoker.getUrl()
+ "] failed. " + e.getMessage(), e);
                    }
                }
            }
        }
    }
}

```

## 分组处理

Dubbo中如果一个微服务接口有多种实现，可以使用 `url["group"]` 标识分组，在服务提供端和消费端都根据需要做相应配置，具体可以参考[官方服务分组](#)

(<http://dubbo.apache.org/zh-cn/docs/user/demos/service-group.html>)。另外有一类场景是，需要将不同分组的服务做聚合处理，关于这个特性请参考[官方分组聚合](#)

(<http://dubbo.apache.org/zh-cn/docs/user/demos/group-merger.html>)，另外在《Dubbo集群之容错》一文也特地剖析了其实现。

JAVA

对于这类带有分组特性的服务实例，Dubbo的目录服务需要另加特殊处理，如下源码所示，先根据group信息进行分组，在有多个分组的情况下，对每一个分组先做一次“折叠处理”，也即将同一分组中的多个可用候选Invoker对象伪装成一个虚拟的Invoker对象。

```
private static final Cluster CLUSTER =
ExtensionLoader.getExtensionLoader(Cluster.class).getAdaptiveExtension();

private List<Invoker<T>> toMergeInvokerList(List<Invoker<T>> invokers) {
    List<Invoker<T>> mergedInvokers = new ArrayList<>();
    Map<String, List<Invoker<T>>> groupMap = new HashMap<>();
    for (Invoker<T> invoker : invokers) {
        String group = invoker.getUrl().getParameter(GROUP_KEY, "");
        groupMap.computeIfAbsent(group, k -> new ArrayList<>());
        groupMap.get(group).add(invoker);
    }

    if (groupMap.size() == 1) {
        mergedInvokers.addAll(groupMap.values().iterator().next());
    } else if (groupMap.size() > 1) {
        for (List<Invoker<T>> groupList : groupMap.values()) {//TAG-
            StaticDirectory<T> staticDirectory = new StaticDirectory<>(groupList);
            staticDirectory.buildRouterChain();
            mergedInvokers.add(CLUSTER.join(staticDirectory));
        }
    } else {
        mergedInvokers = invokers;
    }
    return mergedInvokers;
}
```

让我们先梳理下这神奇的过程是怎么发生的？这还得从 Directory、Router、Cluster 三者间的关系说起：它们都是服务于为某个特定被引用微服务的，首先 Directory 根据注册中心同步的数据列出它所有可用实例，Router 则在此基础上根据路由配置做一些过滤筛选处理，得到最终可用的候选集，最后使用某类 Cluster 实现（大部分是一种容错机制）将候选集（一个 List<Invoker> 列表）伪装成一个单一的Invoker 对象，具体执行RPC调用时，会经由某种 LoadBalance 负载策略从候选集挑选一个Invoker 实例，将RPC调用委托给该实例执行，如果期间出现异常，则执行重试处理。

其实这一过程正是当前所在类 RegistryDirectory 所参与的，可见上述这个 折叠处理 实际上是在中嵌套了一层相似的操作。接下来的章节我们来看看这个被嵌套过程中的最核心组成 StaticDirectory 的实现。

## StaticDirectory

同 RegistryDirectory 一样，StaticDirectory 也是扩展自 AbstractDirectory 抽象类。它的对象是根据需要随时产生的，用于特定微服务的可用候选集此间不会发生变化，因此实现相对简单很多。核心的代码片段如下，从 doList(invocation) 方法可以看出，如果没有调用 buildRouterChain() 方法，则 list(invocation) 方法直接返回构造函数传入的候选集。

```

public class StaticDirectory<T> extends AbstractDirectory<T> {
    private final List<Invoker<T>> invokers;
    ...

    public void buildRouterChain() {
        RouterChain<T> routerChain = RouterChain.buildChain(getUrl());
        routerChain.setInvokers(invokers);
        this.setRouterChain(routerChain);
    }

    @Override
    protected List<Invoker<T>> doList(Invocation invocation) throws RpcException {
        List<Invoker<T>> finalInvokers = invokers;
        if (routerChain != null) {
            try {
                finalInvokers = routerChain.route(getConsumerUrl(), invocation);
            } catch (Throwable t) {
                logger.error("Failed to execute router: " + getUrl() + ", cause: " +
t.getMessage(), t);
            }
        }
        return finalInvokers == null ? Collections.emptyList() : finalInvokers;
    }
}

```

## 刷新总流程

在梳理完“电能 → 势能”的这个转换过程的中间细节后，终于到这里，我们可以从更加宏观的视觉来看看微服务引用实例的刷新主流程了。主体流程如下：

- 当注册中心同步回来的 `invokerUrls` 数据明确告知关于当前被引用微服务没有可用对象时：
  - a. 将 `forbidden` 标记设置为 `true`，对外禁用当前 `directory` 对象；
  - b. 清空 `invokers` 候选集，并重置 `routerChain` 中的候选集；
  - c. 调用 `destroyAllInvokers()` 方法销毁所有的本地 `invoker` 候选对象；
- 入参 `invokerUrls` 数据为空，且最近一次调用 `refresh(invokerUrls)` 缓存在 `cachedInvokerUrls` 的数据也为空，则直接 `return` 返回；
- 有可用候选集的情况下：
  1. 将 `forbidden` 标记设置为 `false`；
  2. 重新实例化 `cachedInvokerUrls` 容器对象，记录全部 `invokerUrls` 到其中；
  3. 传入 `invokerUrls` 参数，调用 `toInvokers(urls)` 方法剔除不满足协议要求的数据，得到新的候选集；
  4. 若新的候选集没有可用服务实例，便退出返回；
  5. 将新候选集设给 `routerChain`，由其负责在执行具体RPC请求时经由路由链中对候选集的做进一步筛选过滤处理；

6. 根据是否有分组参数，决定是否对候选集做 折叠处理；
7. 调用 `destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap)` 清理缓存在本地无用候选 Invoker 对象。

具体执行流程如下源码所示，

```
private volatile Map<String, Invoker<T>> urlInvokerMap;
private volatile List<Invoker<T>> invokers;~
private volatile Set<URL> cachedInvokerUrls;
private void refreshInvoker(List<URL> invokerUrls) {
    Assert.notNull(invokerUrls, "invokerUrls should not be null");
    if (invokerUrls.size() == 1
        && invokerUrls.get(0) != null
        && EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol())) {
        this.forbidden = true;
        this.invokers = Collections.emptyList();
        routerChain.setInvokers(this.invokers);
        destroyAllInvokers();
    } else {
        this.forbidden = false;
        Map<String, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap;
        if (invokerUrls == Collections.<URL>emptyList()) {
            invokerUrls = new ArrayList<>();
        }
        if (invokerUrls.isEmpty() && this.cachedInvokerUrls != null) {
            invokerUrls.addAll(this.cachedInvokerUrls);
        } else {
            this.cachedInvokerUrls = new HashSet<>();
            this.cachedInvokerUrls.addAll(invokerUrls);
        }
        if (invokerUrls.isEmpty()) {
            return;
        }
        Map<String, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls);
        if (CollectionUtils.isEmptyMap(newUrlInvokerMap)) {
            logger.error(new IllegalStateException(
                "urls to invokers error .invokerUrls.size :" + invokerUrls.size()
                + ", invoker.size :0. urls :" + invokerUrls.toString()));
            return;
        }
        List<Invoker<T>> newInvokers = Collections.unmodifiableList(new ArrayList<>(
            newUrlInvokerMap.values()));
        routerChain.setInvokers(newInvokers);
        this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;
        this.urlInvokerMap = newUrlInvokerMap;
        try {
            destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap);
        } catch (Exception e) {
            logger.warn("destroyUnusedInvokers error. ", e);
        }
    }
}
```

JAVA

源码中用到了几个申明了 `volatile` 修饰符的变量，表示他们是跨线程的共享资源，使用他们的时候有些特点，状态型的变量总是在临界区的入口处改变，而数据型的变量则在出口处改变，中间执行逻辑处理时要妥善对待数据型的变量，因为它随时有可能发生变化，比如调用 `unmodifiableList(list)` 给 `routerChain` 设置的候选集。

---

完结

# 【十七】Dubbo微服务导入导出

微服务的架构特点是，服务节点在网络中呈网状分布，尽管节点间以直连形式发生通讯，但在通讯连接建立之前，两节点间是彼此相互孤立的，只有通过注册中心这个第三方媒介的协助，客户端给定名称标识获得服务方的可用物理地址列表，再在此基础上利用某些策略挑选其中一个作为最终的服务提供方。

在具体实现上，类似Zookeeper这类注册中心只是负责了数据的存取和节点相关变化的通知推送而已，从框架上层来看，本地实现的客户端是通过感知相应事件，以异步的方式完成数据的同步的。然而具体实现上，订阅者在发起订阅操作时会主动从注册中心拉取数据，生命周期的此后部分发生变化的数据或子节点们则由回调事件感知，也就是客户端和注册中心存在着推拉结合、推为主的互动模式。这种以响应式为主的数据同步方式的好处是可以节省业务请求之前的准备时间，进而大大的提高服务的可用性。从前面相关文章中我们已经知道，无论是负载均衡、路由处理，还是服务发现其实都是利用这个机制在本机客户端完成的。

## 基础

《Dubbo集群之目录服务》一文中所谓的目录服务实际上就是业界所谓的服务发现，因其在当前客户端以被引用微服务作为粒度单元，它的多个实例组成了一个集群，即便没有集群，单个实例也能通过 DubboProtocol 等被单独导入，因而 RegistryDirectory 目录服务实现是分属于集群中的。本文要讨论分析的 RegistryProtocol 将侧重在服务导出，由于这个动作本身所在当前服务是一个服务提供者实例，和注册中心进行数据同步的主要目的是让服务消费者能感知自身的存在，但是并不需要和同一个服务的其它提供者实例有啥关联处理，因而服务导出，也就本文所讲的服务注册是发生在一个比集群更大粒度的分布式网络中的，这样也就比较容易理解为啥 RegistryProtocol 会调用 RegistryDirectory 做服务导入处理了。

综上，无论是服务导入，还是服务导出，都需要放到一个比集群更大的粒度——微服务分布式网络中，只有将数据同步到注册中心这个第三方媒介，或者从中同步数据，才能让微服务实例彼此间能发现或者感知对方。假如说 DubboProtocol 让一个微服务实例意识到自身个体的存在，那么 RegistryProtocol 则是在注册中心的基础上让它意识到个体间关系的存在。

官网中如下实例，服务的导入导出的输入数据源依然是用URL配置总线加以表达的，表征微服务实例或微服务引用实例的URL数据被编码置入到 url["export"] 或 url["refer"] 这个参数中，也正因为如此，源码中涉及多处相关URL数据的处理。

```

//① 服务导入
//    1.1) 直接导入
"dubbo://service-host/com.foo.FooService?version=1.0.0"
//    1.2) 经注册中心导入
"registry://registry-host/org.apache.dubbo.registry.RegistryService?
refer=URL.encode(\"consumer://consumer-host/com.foo.FooService?version=1.0.0\")"

//② 服务导出
//    2.1) 直接导出
"dubbo://service-host/com.foo.FooService?version=1.0.0"
//    2.2) 经注册中心导出
"registry://registry-host/org.apache.dubbo.registry.RegistryService?
export=URL.encode(\"dubbo://service-host/com.foo.FooService?version=1.0.0\")"

```

如果想要通过本文深入的理解 `RegistryProtocol`，还是建议想仔细阅读《Dubbo RPC 之 Protocol 协议层》的三篇文章。在此先再次拧出官方文档中如下关于 `Protocol` 的言简意赅的介绍：

**“RPC 协议扩展，封装远程调用细节。**

对应本文的 `RegistryProtocol` 来说，其实就是将本地同注册中心的数据同步这个细节封装起来，一方面满足了接口实现，另一方面也将注册中心——准确来说是注册中心的客户端同框架上层解耦了。

在进一步剖析源码实现前，先扫清一些认知上的障碍，以便接下来更加系统深入的理解整个实现。

## SPI相关

`RegistryProtocol` 是接口 `Protocol` 的一个扩展点具类，每一个具类都是单例的，根据Dubbo的 SPI机制，会为扩展点接口动态生成一个代理类，代理类的接口方法实现中，会根据接口本身的相关注解，结合SPI配置文件的映射关系，根据名称获取到它的某个具类的实例，最后将当前方法委托给该实例的对应方法。结合 `@SPI("dubbo")` 这个注解，默认实例的映射名称为 "dubbo"，而 `RegistryProtocol` 被映射为 `registry`。

如果一个扩展具类中的 `setter` 方法的参数也是一个扩展点，Dubbo的SPI机制会自动完成其单例的装配处理，`RegistryProtocol` 有多个这样的方法，如下：

```

//@SPI(FailoverCluster.NAME)
private Cluster cluster;
public void setCluster(Cluster cluster) {
    this.cluster = cluster;
}

//@SPI("dubbo")
private Protocol protocol;
public void setProtocol(Protocol protocol) {
    this.protocol = protocol;
}

//@SPI("dubbo")
private RegistryFactory registryFactory;
public void setRegistryFactory(RegistryFactory registryFactory) {
    this.registryFactory = registryFactory;
}

//@SPI("javassist")
private ProxyFactory proxyFactory;
public void setProxyFactory(ProxyFactory proxyFactory) {
    this.proxyFactory = proxyFactory;
}

```

另外如果一个实现了扩展点接口的具类，其构造函数的入参类型也是该扩展点时，那么说明它是一个包装类，这时当前扩展点除包装类外的其它具类均会被所有的包装类给做一次装饰处理，具体行为取决于他们的总和。

根据各自的注解和实现类的情况，对应的默认具类如下<sub>包装类体现在第二级上</sub>：

- **cluster**: org.apache.dubbo.rpc.cluster.support.FailoverCluster ;
  - **MockClusterWrapper**
- **protocol**: org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol ;
  - **ProtocolListenerWrapper**、**ProtocolFilterWrapper**、**QosProtocolWrapper**
- **registryFactory**: org.apache.dubbo.registry.dubbo.DubboRegistryFactory ;
- **proxyFactory**: org.apache.dubbo.rpc.proxy.javassist.JavassistProxyFactory ;
  - **StubProxyFactoryWrapper**

## 单例模式的 RegistryProtocol

本文所讨论的 **RegistryProtocol** 是一个扩展点，在《Dubbo之SPI扩展点加载》一文中，已经阐述了，于整个应用而言，使用 `ExtensionLoader.getExtensionLoader(SomeClz.class)` 加载的扩展点具类等价于是单例的。如下代码前面那个 `public` 的构造函数主要是为了赋值全局静态变量 `INSTANCE`，便于后续的引用处理。

```
public class RegistryProtocol implements Protocol {  
    ...  
    private static RegistryProtocol INSTANCE;  
  
    public RegistryProtocol() {  
        INSTANCE = this;  
    }  
  
    public static RegistryProtocol getRegistryProtocol() {  
        if (INSTANCE == null) {  
  
            ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(REGISTRY_PROTOCOL);  
        }  
        return INSTANCE;  
    }  
}
```

JAVA

## NOTE

从 `RegistryProtocol` 单例这个角度来看，下文中的 `providerConfigurationListener` 变量也等价于是单例的。

## 服务导入

在阅读这一章节的内容之前，最好先熟读《Dubbo集群之目录服务》，文中剖析的 `RegistryDirectory` 存在的目的是为指定的被引用服务接口列出其所有可用的服务实例，该列表会根据注册中心的响应节点变化而动态改变，具体实现上主要仰赖于类似基于和注册中心以事件回调方式同步覆写规则，从而刷新本地缓存的 `Invoker` 引用实例。

### 大体步骤

服务导入对外的接口方位为 `public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException`，从定义看相当简洁，假如对应 `RegistryProtocol` 中的实现是给客户端呈上的一道菜，发出 `refer(...)` 指令后，`RegistryDirectory` 按指令办事，将原料和佐料准备好后，根据既定的烹饪程序做好这道菜。相对应的我们可以认为：

1. 对应微服务上线的所有实例在注册中心注册的数据节点，以及由配置中心同步的覆写规则这些则可以认为是原料；
2. 入参 `url` 中参数指定了引用服务时的限定条件，这就相当于是辅料，相当于为适配客户口味而调制的调味剂；
3. 当前客户端基于中心同步事件回调执行的逻辑，类如利用覆写规则执行刷新服务实例的过程，就好比其中一个烹饪环节，而新得到的实例就像烹制好了的整菜的一部分；
4. 烹饪有好几个环节，各个环节的有机组合和应用才能最终做好这道菜，服务导入涉及如下环节：
  - 构建 `RegistryDirectory` 实例，并为其备好：

- 用于数据同步的 Registry 实例;
- 用于单个微服务实例导入的 Protocol 实例;
- 构建用于客户端执行目标微服务实例集过滤或筛选的的路由链 RouterChain 实例;
- 到注册中心的为指定 url 数据的客户端订阅特定微服务指定类型节点的变化;
- 选用合适的容错机制或者其他类型的 Cluster 将服务实例候选集伪装成一个 Invoker<T> 实例;

大体步骤实现源码如下：

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    Map<String, String> parameters = new HashMap<String, String>
(directory.getUrl().getParameters());
    URL subscribeUrl = new URL(CONSUMER_PROTOCOL,
        parameters.remove(REGISTER_IP_KEY), 0, type.getName(), parameters);

    if (!ANY_VALUE.equals(url.getServiceInterface()) && url.getParameter(REGISTER_KEY,
true)) {
        directory.setRegisteredConsumerUrl(getRegisteredConsumerUrl(subscribeUrl,
url));
        registry.register(directory.getRegisteredConsumerUrl());
    }
    directory.buildRouterChain(subscribeUrl);
    directory.subscribe(subscribeUrl.addParameter(CATEGORY_KEY,
        PROVIDERS_CATEGORY + "," + CONFIGURATORS_CATEGORY + "," +
ROUTERS_CATEGORY));
    Invoker invoker = cluster.join(directory);
//    ProviderConsumerRegTable.registerConsumer(invoker, url, subscribeUrl, directory);
    return invoker;
}
```

## IMPORTANT

每一个被引用微服务在当前客户端均会存在一个 `RegistryDirectory` 实例，其中声明了一个用于装载该服务引用实例的容器——`volatile List<Invoker<T>> invokers`。

当发起 `subscribe(subscribeUrl)` 操作后，会间接发起对 `Registry#subscribe(URL url, NotifyListener listener)` 的调用，后面这个订阅处理会确保注册中心有相应节点的路径存在，并随即增加相应的监听器和主动获取被订阅 provider 类型节点的所有子节点（页节点），也即被引用微服务的实例集合，该集合会被转换成对应的 `List<Invoker<T>>` 并赋值给 `invokers` 变量。此后如果注册中心因为有实例的增加或者删减而导致代表实例的页节点有变动时，则会通过监听器知会客户端，这时 `invokers` 变量则会被重新赋值刷新处理。

事件发生前后，若代表服务端实例的 URL 数据没有变化，则其对应的 `Invoker<T>` 实例会被原样保留，只是引用会被挪入到由 `invokers` 指向的新产生的 `List<Invoker<T>>` 类型容器中。

`RegistryDirectory` 在刷新 `Invoker<T>` 实例时会调用 `protocol.refer(serviceType, url)`，这里的 `protocol` 是由 `RegistryProtocol` 负责赋值的，负责在协议层完成单个服务实例的引用(也即导入处理)，默认是 `DubboProtocol`。

## 处理细节

代码看似很简单，但是隐藏的细节却相当丰富，需要一一详述：

- 基于注册中心的服务导入中，当前客户端自身所关心的数据全部承载在 `regUrl["refer"]` 中，在构建获取 `subscribeUrl` 时，需要先解析得到 `rawUrl = URL.decode(regUrl["refer"])`，假定 `rawUrl[("^register.ip")]` 表示 `rawUrl` 移除 "register.ip" 后所剩的所有参数，则最终 `subscribeUrl` 的构建形式如下：

```

"consumer://" + (rawUrl["register.ip"] | {local ip}) + "/" + {type.getName()} + "?" +
{rawUrl[^"register.ip"]}

//eg:
//consumer://192.168.0.7/org.apache.dubbo.samples.basic.api.DemoService?
//application=demo-consumer&check=true&dubbo=2.0.2&
//interface=org.apache.dubbo.samples.basic.api.DemoService&
//lazy=false&methods=testVoid,sayHello&pid=69391&release=2.7.3&
//side=consumer&sticky=false&timestamp=1573374561281

```

- 在调用 `RegistryDirectory#subscribe(...)` 时，会为入参置 `url["category"] = "providers,configurators,routers"`，也就是任何以 `RegistryDirectory` 导入的引用微服务均会：1) 监听目标微服务的实例上下线情况；2) 同步来自注册中心的覆写规则变化，根据需要刷新本地配置；3) 路由规则的同步刷新，改变过滤或筛选规则，实际上也就是改变可用的目标服务实例的候选范围；
- 如果没有指定 `regUrl["interface"] = "*"` 和 `regUrl["register"] = false`，`RegistryProtocol` 会将当前客户端作为节点注册到注册中心，用于获取注册的 `registeredConsumerUrl` 的逻辑代码如下，其值为置 `subscribeUrl["category", "check"] = "consumers", false` 得到，只是在指定 `regUrl["simplified"] = true` 的情况下，其它参数中只保留 `"application"、"version"、"group"、"dubbo"、"release"` 这些。

JAVA

```

public static final String[] DEFAULT_REGISTER_CONSUMER_KEYS = {
    APPLICATION_KEY, VERSION_KEY, GROUP_KEY, DUBBO_VERSION_KEY, RELEASE_KEY
};

public URL getRegisteredConsumerUrl(final URL consumerUrl, URL registryUrl) {
    if (!registryUrl.getParameter(SIMPLIFIED_KEY, false)) {
        return consumerUrl.addParameters(CATEGORY_KEY, CONSUMERS_CATEGORY,
            CHECK_KEY, String.valueOf(false));
    } else {
        return URL.valueOf(consumerUrl, DEFAULT_REGISTER_CONSUMER_KEYS, null)
            .addParameters(CATEGORY_KEY, CONSUMERS_CATEGORY, CHECK_KEY,
            String.valueOf(false));
    }
}

```

## doRefer(...) 之前

然而基于注册中心的服务导入，在 `doRefer(...)` 之前还有几处细节需要处理。首先需要规整 `regUrl`，也即设 `regUrl.protocol = (regUrl["registry"] | "dubbo")`，移除 `regUrl["registry"]`。其次对于使用 `RegistryProtocol` 引用 `RegistryService` 类型的服务时，是无需经过服务发现机制引用的，因为它不像其他服务一样，行为由远端主机提供，其实现本质而言就是一个注册中心的客户端，远端只负责相关节点及数据的存取，行为则是由本地提供，因此可以通过本机代理机制直接获取到 `RegistryService` 实例。最后如果客户端配置了 `url["group"]`，则说明需要做结果聚合处理，此时使用的 `Cluster` 则应该是 `MergeableCluster`，具体参考《Dubbo集群之容错》一文中 `Mergeable`(结果聚合) 这一章节内容。

```

public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    url = URLBuilder.from(url)
        .setProtocol(url.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY))
        .removeParameter(REGISTRY_KEY)
        .build();
    Registry registry = registryFactory.getRegistry(url);
    if (RegistryService.class.equals(type)) {
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    // group="a,b" or group="*"
    Map<String, String> qs =
    StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
    String group = qs.get(GROUP_KEY);
    if (group != null && group.length() > 0) {
        if ((COMMA_SPLIT_PATTERN.split(group)).length > 1 || "*" .equals(group)) {
            return doRefer(getMergeableCluster(), registry, type, url);
        }
    }
    return doRefer(cluster, registry, type, url);
}

private Cluster getMergeableCluster() {
    return ExtensionLoader.getExtensionLoader(Cluster.class).getExtension("mergeable");
}

```

## 服务导出

同样，微服务的导出也是相对整个微服务分布式网络而言，正如上文所述，一个微服务虽然绝大部分时刻是以集群的形式对外提供服务的，但是就单个服务实例而言，它并不需要知道这些信息，只有服务的消费者在发起具体请求时需要知晓，也即集群信息是由客户端在注册中心的协助下各自独立维护的。

然而，就如同《Dubbo 配置管理》一文中的开头部分所言，微服务的配置管理离不开注册中心这种分布式协调框架的支持。

由于相关源码牵涉比较多的细节，没法一览知义，下述由浅及深，逐个击破。

## 相关 URL 数据处理

Dubbo在生成本地微服务实例的初始阶段时，需要先经过配置层的数据读入处理，然后经由框架代理层将对应接口实现转换成对应的一个原始 `Invoker<T>` 对象—— `originInvoker`，通过该对象的 `getUrl()` 方法能获得原始的URL数据—— `regUrl`。然后有两种方式可以表示当前微服务使用何种注册中心导出，分别是(此处假设使用 `zookeeper` 作为注册中心)：1) `regUrl.protocol = "registry"` 并且 `regUrl["registry"] = "zookeeper"`；2) `regUrl.protocol = "zookeeper"`。针对第一种情况，`RegistryProtocol` 在执行服务到处时会使用如下 `getRegistryUrl(originInvoker)` 获得统一表示，也即第二种的标准表示，同时会移除 `regUrl["registry"]` 参数，若没有明确指定该参数，则会设 `regUrl.protocol = "dubbo"`。

```
private URL getRegistryUrl(Invoker<?> originInvoker) {  
    URL registryUrl = originInvoker.getUrl();  
    if (REGISTRY_PROTOCOL.equals(registryUrl.getProtocol())) {  
        String protocol = registryUrl.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY);  
        registryUrl = registryUrl.setProtocol(protocol).removeParameter(REGISTRY_KEY);  
    }  
    return registryUrl;  
}  
  
JAVA
```

regUrl["export"] 编码封装了当前被导出微服务本身的信息，需经过 getProviderUrl(originInvoker) 解码获得其URL数据—— providerUrl。

```
private URL getProviderUrl(final Invoker<?> originInvoker) {  
    String export = originInvoker.getUrl().getParameterAndDecoded(EXPORT_KEY);  
    if (export == null || export.length() == 0) {  
        throw new IllegalArgumentException("The registry export url is null! registry:  
" + originInvoker.getUrl());  
    }  
    return URL.valueOf(export);  
}  
  
JAVA
```

Dubbo的注册中心中存在一类 "configurators" 节点，一个微服务的相关的覆写规则会作为其子节点出现。其完整URL数据表示—— overrideSubscribeUrl，是在 providerUrl 的基础上获得的，也即设 providerUrl["category", "check"] = "configurators", false、providerUrl.protocol = "provider"。当然，它也是服务实例用于订阅配置类节点的。

```
private URL getSubscribedOverrideUrl(URL registeredProviderUrl) {  
    return registeredProviderUrl.setProtocol(PROVIDER_PROTOCOL)  
        .addParameters(CATEGORY_KEY, CONFIGURATORS_CATEGORY, CHECK_KEY,  
        String.valueOf(false));  
}  
  
JAVA
```

URL配置总线在Dubbo中作为载体起到了上下文参数存取和传递的作用，然而环节传递过程中并不是毫无保留的全盘脱出，多出的参数会扰乱下一环的业务处理，也可会造成某些不必要的数据泄露风险，因此无论这种传递是跨方法的还是跨服务进程的，都会经过必要的筛选处理，或增或减，抑或重新组装 URL实例。

一个服务实例的大部分配置数据都有可能装载在代表它的URL数据中—— providerUrl = regUrl["export"]，其中一部分仅限于本实例使用，集群中其它实例或者它的消费者并不需要知晓，或者说不应该暴露给它们。换言之，代表服务实例完成到注册中心注册的URL数据—— registeredProviderUrl，应该是由 providerUrl 裁剪得到的，其获取方式有如下：

1. 检验是否含有 regUrl["simplified"] = true :
2. 无，默认情况，去掉 providerUrl 中的如下参数：

- 带“.”前缀的参数；
- “monitor”、“bind.ip”、“bind.port”、“qos.enable”、“qos.host”、“qos.port”、“qos.accept.foreign.ip”、“validation”、“interfaces”

3. 有，按如下步骤组装URL数据：

- 保留“application”、“codec”、“exchanger”、“serialization”、“cluster”、“connections”、“deprecated”、“group”、“loadbalance”、“mock”、“path”、“timeout”、“token”、“version”、“warmup”、“weight”、“timestamp”、“dubbo”、“release”这些参数；
- Dubbo优先使用`url["interface"]`参数表示服务接口，没有该参数的情况下使用`url.path`，前者存在的情况下，若和后者不一样，也需要保留；
- `url["extra-keys"]`也参数原样保留；
- 另外参数附有方法前缀的也愿意保留，前缀满足`prefix ∈ url["methods"]`（“,”逗号分隔的方法名称）；

```

private URL getRegisteredProviderUrl(final URL providerUrl, final URL registryUrl) { JAVA
    if (!registryUrl.getParameter(SIMPLIFIED_KEY, false)) {
        return providerUrl.removeParameters(getFilteredKeys(providerUrl))
            .removeParameters(MONITOR_KEY, BIND_IP_KEY, BIND_PORT_KEY, QOS_ENABLE,
                QOS_HOST, QOS_PORT, ACCEPT_FOREIGN_IP, VALIDATION_KEY, INTERFACES);
    } else {
        String extraKeys = registryUrl.getParameter(EXTRA_KEYS_KEY, "");
        if (!providerUrl.getPath().equals(providerUrl.getParameter(INTERFACE_KEY))) {
            if (StringUtils.isNotEmpty(extraKeys)) {
                extraKeys += ",";
            }
            extraKeys += INTERFACE_KEY;
        }
        String[] paramsToRegistry = getParamsToRegistry(DEFAULT_REGISTER_PROVIDER_KEYS
            , COMMA_SPLIT_PATTERN.split(extraKeys));
        return URL.valueOf(providerUrl, paramsToRegistry,
            providerUrl.getParameter(METHODS_KEY, (String[]) null));
    }
}

private static String[] getFilteredKeys(URL url) {
    Map<String, String> params = url.getParameters();
    if (CollectionUtils.isNotEmptyMap(params)) {
        return params.keySet().stream()
            .filter(k -> k.startsWith(HIDE_KEY_PREFIX))
            .toArray(String[]::new);
    } else {
        return new String[0];
    }
}

public static final String[] DEFAULT_REGISTER_PROVIDER_KEYS = {
    APPLICATION_KEY, CODEC_KEY, EXCHANGER_KEY, SERIALIZATION_KEY,
    CLUSTER_KEY, CONNECTIONS_KEY, DEPRECATED_KEY,
    GROUP_KEY, LOADBALANCE_KEY, MOCK_KEY, PATH_KEY, TIMEOUT_KEY,
    TOKEN_KEY, VERSION_KEY, WARMUP_KEY,
    WEIGHT_KEY, TIMESTAMP_KEY, DUBBO_VERSION_KEY, RELEASE_KEY
};

public String[] getParamsToRegistry(String[] defaultKeys, String[]
additionalParameterKeys) {
    int additionalLen = additionalParameterKeys.length;
    String[] registryParams = new String[defaultKeys.length + additionalLen];
    System.arraycopy(defaultKeys, 0, registryParams, 0, defaultKeys.length);
    System.arraycopy(additionalParameterKeys, 0,
        registryParams, defaultKeys.length, additionalLen);
    return registryParams;
}

```

## ACL 在服务端的应用

《Dubbo集群之目录服务》一文中已经花费大量篇幅，深刻阐述了利用覆写规则同步刷新微服务引用实例的实现，与之相似，当本地服务实例监听到来自系统维护人员通过配置中心修改相关配置的事件后，也会对实例做相应的刷新处理。在其“同步覆写规则”这一章节中已经介绍过，本地接受到的事件中会含有对应的覆写规则的文本数据，`AbstractConfiguratorListener`会将其转换成对应的`List<Configurator> configurators`覆写规则处理器列表，实现类会在需要时调用`configurators`改写代表实例的URL数据，正如下述`getConfigedInvokerUrl(configurators, url)`所实现的逻辑那样。而方法最终返回的URL数据则是用于产生新的实例，并替换掉旧的那个。

```
private static URL getConfigedInvokerUrl(List<Configurator> configurators, URL url) {  
    if (configurators != null && configurators.size() > 0) {  
        for (Configurator configurator : configurators) {  
            url = configurator.configure(url);  
        }  
    }  
    return url;  
}
```

相似地，由于一个应用中可以存在多个微服务，因而在服务端依然按照应用级和服务级分别同步覆写规则，对应提供`AbstractConfiguratorListener`抽象类的扩展实现——`ProviderConfigurationListener`和`ServiceConfigurationListener`，分别订阅配置中心对应的“`/({namespace} | dubbo)/config/dubbo/{app}.configurators`”节点和“`/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators`”节点，它们都含有如下一个覆写URL数据的方法：

```
private class (ServiceConfigurationListener | ProviderConfigurationListener) extends AbstractConfiguratorListener{  
    ...  
    private <T> URL overrideUrl(URL url) {  
        return RegistryProtocol.getConfigedInvokerUrl(configurators, url);  
    }  
    ...  
}
```

`AbstractConfiguratorListener`的扩展实现类会在构造函数调用其定义的`initWith(key)`方法，一旦被实例化，也意味着该方法被调用，随后便会主动从配置中心的由`key`代表的对应节点拉取到覆写规则的文本数据，并被转换成`Configurator`对象装入`configurators`容器中，而后续如果相关的治理操作改写了规则，那么`ConfigurationListener`监听器实现会被触发，回调逻辑中会对`configurators`重新赋值。

显然，从属于应用的微服务，在应用覆写规则刷新实例时，需要综合应用级别和自身服务级别的覆写规则，如下，两次调用`overrideUrl(url)`这个方法。

```
private final Map<String, ServiceConfigurationListener> serviceConfigurationListeners = new ConcurrentHashMap<>();  
  
private final ProviderConfigurationListener providerConfigurationListener = new ProviderConfigurationListener();  
  
private URL overrideUrlWithConfig(URL providerUrl, OverrideListener listener) {  
    providerUrl = providerConfigurationListener.overrideUrl(providerUrl);  
    ServiceConfigurationListener serviceConfigurationListener =  
        new ServiceConfigurationListener(providerUrl, listener);  
    serviceConfigurationListeners.put(providerUrl.getServiceKey(),  
        serviceConfigurationListener);  
    return serviceConfigurationListener.overrideUrl(providerUrl);  
}
```

从上述源码中不难发现，当前服务端应用的每一个微服务实例均会对应存在一个 `ServiceConfigurationListener` 实例，该实例中绑定了一个 `OverrideListener` 对象，其定义的方法 `doOverrideIfNecessary()` 正是用于实现服务实例刷新的，也被认为是重新导出。该方法会在父类定义的回调方法 `notifyOverrides()` 的实现中被调用，如下源码，也就是说服务治理引发的事件驱动着服务实例的重新导出处理。

```
private class ProviderConfigurationListener extends AbstractConfiguratorListener {  
    ...  
    @Override  
    protected void notifyOverrides() {  
        overrideListeners.values().forEach(listener -> ((OverrideListener)  
            listener).doOverrideIfNecessary());  
    }  
}  
  
private class ServiceConfigurationListener extends AbstractConfiguratorListener {  
    ...  
    @Override  
    protected void notifyOverrides() {  
        notifyListener.doOverrideIfNecessary();  
    }  
}
```

可见应用级别的覆写规则会引起对应应用中的所有微服务的 `doOverrideIfNecessary()` 方法的回调，这里我们可以认为 `overrideListeners.values()` 等价于从 `serviceConfigurationListeners.values()` 集合中执行 `map(v -> v.notifyListener)` 所得，具体情况下文会涉及。

## OverrideListener

上述章节已经说明了 `OverrideListener` 是利用事件回调机制同步覆写规则，从而执行服务实例刷新的。该类实现了 `NotifyListener` 接口，而后者是注册中心客户端所定义的，用于在被关注的节点或节点相关数据变化时，回调指定的业务逻辑。也就是说覆写规则的数据同步方案实际上是有两种实现方

案，一种是拥有单独的配置中心，另外一种直接利用注册中心，如果两种都有的话，则会共同发生作用。没有提供对应的配置中心实现时，相应 ConfigurationListener 接口实现就不会发生作用。

先看看对应 doOverrideIfNecessary() 方法的实现，如下，步骤很清晰：

1. 首先由 URL.decode(regUrl["export"]) 解析得到 originUrl；
2. 然后根据它计算出 key 值，并由该 key 从 bounds 取得与 originInvoker 对应的 ExporterChangeableWrapper 实例 exporter，它的 invoker 变量缓存了 originInvoker 经过规则覆写后的版本；
3. 随后经 exporter.getInvoker().getUrl() 得到最近被覆写过的URL数据 currentUrl；
4. 接着对 originUrl 应用同步于注册中心和配置中心的覆写规则，得到新的URL数据 newUrl；
5. 最后若 currentUrl.equals(newUrl)，则表示当前发生的覆写操作并没有引起URL数据的变化，只有不相等时才会执行对应服务实例 originInvoker 的重新导出处理；

```

private class OverrideListener implements NotifyListener {

    private final URL subscribeUrl;

    private final Invoker originInvoker;

    private List<Configurator> configurators;

    public OverrideListener(URL subscribeUrl, Invoker originalInvoker) {
        this.subscribeUrl = subscribeUrl;
        this.originInvoker = originalInvoker;
    }

    ...

    public synchronized void doOverrideIfNecessary() {
        final Invoker<?> invoker;
        if (originInvoker instanceof InvokerDelegate) {
            invoker = ((InvokerDelegate<?>) originInvoker).getInvoker();
        } else {
            invoker = originInvoker;
        }
        URL originUrl = RegistryProtocol.this.getProviderUrl(invoker);
        String key = getCacheKey(originInvoker);
        ExporterChangeableWrapper<?> exporter = bounds.get(key);
        if (exporter == null) {
            logger.warn(new IllegalStateException("error state, exporter should not be
null"));
            return;
        }
        //The current, may have been merged many times
        URL currentUrl = exporter.getInvoker().getUrl();
        //Merged with this configuration
        URL newUrl = getConfigedInvokerUrl(configurators, originUrl);
        newUrl =
getConfigedInvokerUrl(providerConfigurationListener.getConfigurators(), newUrl);
        newUrl =
getConfigedInvokerUrl(serviceConfigurationListeners.get(originUrl.getServiceKey())
                .getConfigurators(), newUrl);
        if (!currentUrl.equals(newUrl)) {
            RegistryProtocol.this.reExport(originInvoker, newUrl);
            logger.info("exported provider url changed, origin url: " + originUrl +
                    ", old export url: " + currentUrl + ", new export url: " + newUrl);
        }
    }
}

```

JAVA

剩下有关的实现是同于同步注册中心的覆写规则，如下源码，先对回调事件的节点列表执行执行匹配检查，如果没有匹配则直接返回，否则将所有匹配的URL数据——`url.protocol = "override"` 或 `url["category"] = "configurators"`—转换成覆写规则处理器，最后再同样调用 `doOverrideIfNecessary()` 执行服务实例的重新导出处理。

```

private class OverrideListener implements NotifyListener {
    ...
    public synchronized void notify(List<URL> urls) {
        logger.debug("original override urls: " + urls);

        List<URL> matchedUrls = getMatchedUrls(urls,
        subscribeUrl.addParameter(CATEGORY_KEY,
            CONFIGURATORS_CATEGORY));
        logger.debug("subscribe url: " + subscribeUrl + ", override urls: " +
        matchedUrls);

        // No matching results
        if (matchedUrls.isEmpty()) {
            return;
        }

        this.configurators = Configurator.toConfigurators(classifyUrls(matchedUrls,
        UrlUtils::isConfigurator))
            .orElse(configurators);

        doOverrideIfNecessary();
    }
    private List<URL> getMatchedUrls(List<URL> configuratorUrls, URL currentSubscribe)
    {
        List<URL> result = new ArrayList<URL>();
        for (URL url : configuratorUrls) {
            URL overrideUrl = url;
            // Compatible with the old version
            if (url.getParameter(CATEGORY_KEY) == null &&
OVERRIDE_PROTOCOL.equals(url.getProtocol())) {
                overrideUrl = url.addParameter(CATEGORY_KEY, CONFIGURATORS_CATEGORY);
            }

            // Check whether url is to be applied to the current service
            if (UrlUtils.isMatch(currentSubscribe, overrideUrl)) {
                result.add(url);
            }
        }
        return result;
    }
}

public static boolean UrlUtils#isConfigurator(URL url) {
    return OVERRIDE_PROTOCOL.equals(url.getProtocol()) ||
        CONFIGURATORS_CATEGORY.equals(url.getParameter(CATEGORY_KEY,
DEFAULT_CATEGORY));
}

```

需要注意的是，老的版本中，一个表示配置类的节点，其 `url.protocol = "override"`，而新版本则用 `url["category"] = "configurators"` 配置项加以表达。为了适配 `isMatch(consumerUrl, providerUrl)`（没有要求“`url.protocol`也要匹配），特针对老版本配置类的URL数据中临时加上该项。

## ExporterChangeableWrapper

根据 `Protocol` 的定义，服务导出后需要返回一个对应的 `Exporter` 实例，其目的主要是用于入参 `Invoker<T>` 实例相关的销毁处理。对应 `RegistryProtocol` 中的业务逻辑就是为当前服务实例执行如下动作：

1. 从注册中心注销，也即相应 `provider` 类 节点解注册；
2. 删除用于同步 注册中心 覆写规则的监听器，也即解订阅相应 `configurators` 类 节点；
3. 删除用于同步 配置中心 覆写规则的监听器，也即解订阅相应的 “`/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators`” 节点；
4. 最后利用线程池异步调用 `exporter.unexport()` 方法最终完成销毁处理，其中 `exporter` 是用于完成服务实例的本机销毁处理的；

```

private class ExporterChangeableWrapper<T> implements Exporter<T> {
    ...
    @Override
    public void unexport() {
        String key = getCacheKey(this.originInvoker);
        bounds.remove(key);

        Registry registry = RegistryProtocol.INSTANCE.getRegistry(originInvoker);
        try {
            registry.unregister(registerUrl);
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
        try {
            NotifyListener listener = RegistryProtocol.INSTANCE
                .overrideListeners.remove(subscribeUrl);
            registry.unsubscribe(subscribeUrl, listener);
            DynamicConfiguration.getDynamicConfiguration()
                .removeListener(subscribeUrl.getServiceKey() +
CONFIGURATORS_SUFFIX,
serviceConfigurationListeners.get(subscribeUrl.getServiceKey()));
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }

        executor.submit(() -> {
            try {
                int timeout = ConfigurationUtils.getServerShutdownTimeout();
                if (timeout > 0) {
                    logger.info("Waiting " + timeout
                        + "ms for registry to notify all consumers before unexport. " +
                        "Usually, this is called when you use dubbo API");
                    Thread.sleep(timeout);
                }
                exporter.unexport();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        });
    }
}

```

JAVA

解注册或者解订阅是一个网络I/O操作，总共涉及3个这样的操作，耗时相对会比较长，且没法准确预估全部完成的时间，因此使用了配置的大概时间延时执行`exporter`的销毁处理，超时配置为`conf["dubbo.service.shutdown.wait"]`或`conf["dubbo.service.shutdown.wait.seconds"]`。

由上文已知，本机导出的初始服务实例记为`originInvoker`，此后经通知事件同步覆写规则时都是基于它执行刷新进而得到一个新的`<Invoker<T>, Exporter<T>>`对象组合的。因而`originInvoker`被声明成了`final`型，而`exporter`却是可变的，而这也是类名含有`Changeable`字样的奥义所在，如下所示：

```

private class ExporterChangeableWrapper<T> implements Exporter<T> {
    ...
    private final Invoker<T> originInvoker;
    private Exporter<T> exporter;
    public ExporterChangeableWrapper(Exporter<T> exporter, Invoker<T> originInvoker) {
        this.exporter = exporter;
        this.originInvoker = originInvoker;
    }

    public Invoker<T> getOriginInvoker() {
        return originInvoker;
    }

    @Override
    public Invoker<T> getInvoker() {
        return exporter.getInvoker();
    }

    public void setExporter(Exporter<T> exporter) {
        this.exporter = exporter;
    }

    private URL subscribeUrl;
    private URL registerUrl;

    public void setSubscribeUrl(URL subscribeUrl) {
        this.subscribeUrl = subscribeUrl;
    }

    public void setRegisterUrl(URL registerUrl) {
        this.registerUrl = registerUrl;
    }
}

```

JAVA

源码最后呈现的 `subscribeUrl` 和 `registerUrl`，一个用于订阅 `configurators` 类节点，另一个则用于注册一个 `provider` 类节点。由于 `provider` 类节点是一个服务实例的可公示数据的完整 URL 表示，因此经过应用覆写规则后，`registerUrl` 是会发生变化的。

## InvokerDelegate<T>

可以说，`InvokerDelegate<T>` 这个公有静态内部类是整个 `RegistryProtocol` 源码中涉及代码最少，但理解上却最不直观的一个类，为啥需要它，它到底有啥作用？

先看看其父类 `InvokerWrapper<T>`，`Wrapper` 的含义是采用委托方式实现某一接口方法，而被委托对象（实现同一接口）的行为被封装了，`Wrapper` 类可以对其行为进行改写或者隐藏，如下述源码所示，被委托的 `invoker` 变量是有自己的 `getUrl()` 实现的，但是 `InvokerWrapper<T>` 却利用构造函数传入的 `url` 将其隐藏了，调用同一方法将会得到该 `url`。

```
public class InvokerWrapper<T> implements Invoker<T> {  
  
    private final Invoker<T> invoker;  
  
    private final URL url;  
  
    public InvokerWrapper(Invoker<T> invoker, URL url) {  
        this.invoker = invoker;  
        this.url = url;  
    }  
  
    @Override  
    public URL getUrl() {  
        return url;  
    }  
    ...//利用委托机制直接实现所有Invoker<T>接口的其它方法  
}
```

JAVA

再回到 `InvokerDelegate<T>` 本身，首先它新声明的 `invoker` 属性“覆写”了父类所定义的，行为上没有发生变化，但是解决了父类中由于 `invoker` 被申明为私有而无法访问的问题。其它相比而言只增加了一个 `getInvoker()` 方法，原因是内嵌的 `invoker` 可能也是一个 `InvokerDelegate<T>` 类对象，这种情况下只有通过 `instanceof` 类型判断才能递归获取到最初被封装的那个 `Invoker<T>` 类对象。

```
public static class InvokerDelegate<T> extends InvokerWrapper<T> {  
    private final Invoker<T> invoker;  
  
    public InvokerDelegate(Invoker<T> invoker, URL url) {  
        super(invoker, url);  
        this.invoker = invoker;  
    }  
  
    public Invoker<T> getInvoker() {  
        if (invoker instanceof InvokerDelegate) {  
            return ((InvokerDelegate<T>) invoker).getInvoker();  
        } else {  
            return invoker;  
        }  
    }  
}
```

JAVA

上文中关于同步覆写规则处理的剖析中，有出现过类似的一段代码，根据其应用，我们知道其目的是为了获取最初服务实例在本地导出时所输入的 `providerUrl`。

```
private class OverrideListener implements NotifyListener {  
    ...  
    public synchronized void doOverrideIfNecessary() {  
        final Invoker<?> invoker;  
        if (originInvoker instanceof InvokerDelegate) {  
            invoker = ((InvokerDelegate<?>) originInvoker).getInvoker();  
        } else {  
            invoker = originInvoker;  
        }  
        ...  
    }  
}
```

JAVA

## doLocalExport(...) 和 doChangeLocalExport(...)

见名知意，二者对应的是本地的导出处理，分别对应了服务实例的初始导出过程和同步覆写规则时的重新导出过程。显然，这里的本地导出的主要过程是由 `protocol`，比如说 `DubboProtocol` 来完成的。

上述曾提及Dubbo的框架代理层为当前微服务所最初产生 `Invoker<T>` 实例被记为 `originInvoker`，其URL数据表示是一个包含了与注册相关信息的完整 `regUrl`，真正代表本尊的 URL数据 `providerUrl` 需要另行解析，并且此后随着来自于配置中心的覆写规则同步，它会发生变化。然而，业务逻辑是随代码固化下来了的，能改变的是相关配置，比如实例所运行的上下文环境、业务相关参数，也就是说变化的只是代表 `originInvoker` 的URL数据。因此具体实现时，`originInvoker` 会被封入到一个 `InvokerDelegate<T>` 类型对象中。一方面可以确保框架后续流程中能够直接获取到服务实例的 `providerUrl`，避免每次都需要在 `regUrl` 上另加解析，顶层并不需要或者关心该 `regUrl`。另一方面，框架代理层只需执行一次 `originInvoker` 的生成处理。

章节 `ExporterChangeableWrapper` 中已经阐明微服务实例的销毁是一个必须的I/O流程，销毁是以 `originInvoker` 作为参考坐标系的，即便是在并发环境下，来自注册中心或配置中心的覆写规则同步事件可能随时发生，但任意时刻于特定微服务来说当前应用只会存在一个对应的 `Invoker<T>` 实例，初次导出时是 `originInvoker`，此后则是一个封入了它的 `InvokerDelegate<T>` 类型的包装对象 `delegateInvoker`，因而组合了 `originInvoker`、`delegateInvoker`、`delegateInvoker'sExporter` 三者的 `ExporterChangeableWrapper` 类型对象会使用 `ConcurrentMap<String, ExporterChangeableWrapper<?>>` 类型的安全并发容器 `bounds` 做存取处理，键取 `URL.decode(regUrl["export"])[^["dynamic", "enabled"]]`。

JAVA

```

private final ConcurrentMap<String, ExporterChangeableWrapper<?>> bounds = new
ConcurrentHashMap<>();

private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker,
URL providerUrl) {
    String key = getCacheKey(originInvoker);

    return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {
        Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);
        return new ExporterChangeableWrapper<>(
            (Exporter<T>)protocol.export(invokerDelegate), originInvoker);
    });
}

private <T> ExporterChangeableWrapper doChangeLocalExport(final Invoker<T>
originInvoker, URL newInvokerUrl) {
    String key = getCacheKey(originInvoker);
    final ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>)
bounds.get(key);
    if (exporter == null) {
        logger.warn(new IllegalStateException("error state, exporter should not be
null"));
    } else {
        final Invoker<T> invokerDelegate = new InvokerDelegate<T>(originInvoker,
newInvokerUrl);
        exporter.setExporter(protocol.export(invokerDelegate));
    }
    return exporter;
}

//URL.decode(regUrl["export"])[^["dynamic", "enabled"]]
private String getCacheKey(final Invoker<?> originInvoker) {
    URL providerUrl = getProviderUrl(originInvoker);
    String key = providerUrl.removeParameters("dynamic", "enabled").toFullString();
    return key;
}

```

## export(Invoker<T>) 导出主流程

终于，在理清楚所有细节后，可以进入到主流程看看具体的导出过程了。下述是所有相关剩下的源码，整体过程如下：

- 首先，基于从框架代理层生成的 `originInvoker` 对象获得 `regUrl`、`providerUrl`、`overrideSubscribeUrl` 这 3 个 URL 数据；
- 然后，创建并增设用于从配置中心同步覆写规则的两级监听器，并完成 `providerUrl` 的初始化时的改写处理，基于已改写的 `providerUrl` 执行 `originInvoker` 的本地导出处理，得到 `ExporterChangeableWrapper<T>` 类型的 `exporter` 对象；

3. 紧接着去除 providerUrl 中只用于服务实例本地总线参数，生成 registeredProviderUrl，同时获取应用层提供的注册中心实例 registry，使用二者完成完成当前服务实例到注册中心的登记处理；
4. 将第二个步骤产生的 OverrideListener (实现了 NotifyListener 接口) 监听器设置到 registry 的 overrideSubscribeUrl 这个 configurators 类型的页节点上；
5. 最后，完善 exporter 对象的填值处理，创建并返回一个封装了它的 DestroyableExporter<T> 对象；

JAVA

```

private final Map<URL, NotifyListener> overrideListeners = new ConcurrentHashMap<>();

public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    URL registryUrl = getRegistryUrl(originInvoker);
    // url to export locally
    URL providerUrl = getProviderUrl(originInvoker);

    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
    final OverrideListener overrideSubscribeListener =
        new OverrideListener(overrideSubscribeUrl, originInvoker);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

    providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker,
providerUrl);

    final Registry registry = getRegistry(originInvoker);
    final URL registeredProviderUrl = getRegisteredProviderUrl(providerUrl,
registryUrl);
    ProviderInvokerWrapper<T> providerInvokerWrapper =
        ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl,
registeredProviderUrl);
    boolean register = providerUrl.getParameter(register_KEY, true);
    if (register) {
        register(registryUrl, registeredProviderUrl);
        providerInvokerWrapper.setReg(true);
    }

    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);

    exporter.setRegisterUrl(registeredProviderUrl);
    exporter.setSubscribeUrl(overrideSubscribeUrl);
    //Ensure that a new exporter instance is returned every time export
    return new DestroyableExporter<>(exporter);
}

public void register(URL registryUrl, URL registeredProviderUrl) {
    Registry registry = registryFactory.getRegistry(registryUrl);
    registry.register(registeredProviderUrl);
}

public void unregister(URL registryUrl, URL registeredProviderUrl) {
    Registry registry = registryFactory.getRegistry(registryUrl);
    registry.unregister(registeredProviderUrl);
}

```

## reExport(Invoker<T>, URL) 重新导出主流程

如下述源码所示，重新导出的流程实际实际上很简单，首先执行本地的重导入处理，然后只是简单的将当前服务实例已应用过同步事件覆写规则的 registeredProviderUrl 设给最初服务实例在本地导出时就生成了的 ExporterChangeableWrapper 类型对象 exporter。

```
public <T> void reExport(final Invoker<T> originInvoker, URL newInvokerUrl) {  
    // update local exporter  
    ExporterChangeableWrapper exporter = doChangeLocalExport(originInvoker,  
    newInvokerUrl);  
    // update registry  
    URL registryUrl = getRegistryUrl(originInvoker);  
    final URL registeredProviderUrl = getRegisteredProviderUrl(newInvokerUrl,  
    registryUrl);  
  
    ...//TAG:x  
  
    exporter.setRegisterUrl(registeredProviderUrl);  
}
```

JAVA

然而问题来了，之所以重新导出的原因是运维人员在配置中心改写了相关配置项，从而导致当前微服务实例的覆写规则同步事件收到了通知，这又进一步引起了 `originInvoker` 相关的URL数据的变化。我们都清楚一个微服务的实例是作为临时页节点存储在注册中心的，节点是该实例的完整URL数据表示，此时本地版本已经发生了变化，而注册中心还维持着原样，这肯定会导致不一致。

其实上述源码中 `TAG:x` 处故意给删除了如下一段代码，基本意思是在本地会有一个 `ProviderConsumerRegTable` 缓存容器，类似于注册表，就 `originInvoker` 而言，如果注册表中已经记录的 `registeredProviderUrl` 和当前刷新后的不一致，便先使用旧的值从注册中心执行解注册处理，然后用心的值做登记。

```
public <T> void reExport(final Invoker<T> originInvoker, URL newInvokerUrl) {  
    ...  
    //decide if we need to re-publish  
    ProviderInvokerWrapper<T> providerInvokerWrapper =  
        ProviderConsumerRegTable.getProviderWrapper(registeredProviderUrl,  
    originInvoker);  
    ProviderInvokerWrapper<T> newProviderInvokerWrapper =  
        ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl,  
    registeredProviderUrl);  
  
    if (providerInvokerWrapper.isReg() && !registeredProviderUrl.equals(  
        providerInvokerWrapper.getProviderUrl())) {  
        unregister(registryUrl, providerInvokerWrapper.getProviderUrl());  
        register(registryUrl, registeredProviderUrl);  
        newProviderInvokerWrapper.setReg(true);  
    }  
    ...  
}
```

JAVA

`ProviderInvokerWrapper` 和 `ProviderConsumerRegTable` 相关实现后面有机会再聊。

## RegistryProtocol#destroy()

`RegistryProtocol` 的销毁处理显得相当干净利落，先是从 `bounds` 取出所有所有的 `Exporter` 执行其 `unexport()`，然后删除到配置中心的应用级别的覆写规则同步监听器。

```
public void destroy() {  
    List<Exporter<?>> exporters = new ArrayList<Exporter<?>>(bounds.values());  
    for (Exporter<?> exporter : exporters) {  
        exporter.unexport();  
    }  
    bounds.clear();  
  
    DynamicConfiguration.getDynamicConfiguration().removeListener(  
        ApplicationModel.getApplication() + CONFIGURATORS_SUFFIX,  
        providerConfigurationListener);  
}
```

---

完结

# Dubbo服务代理

---

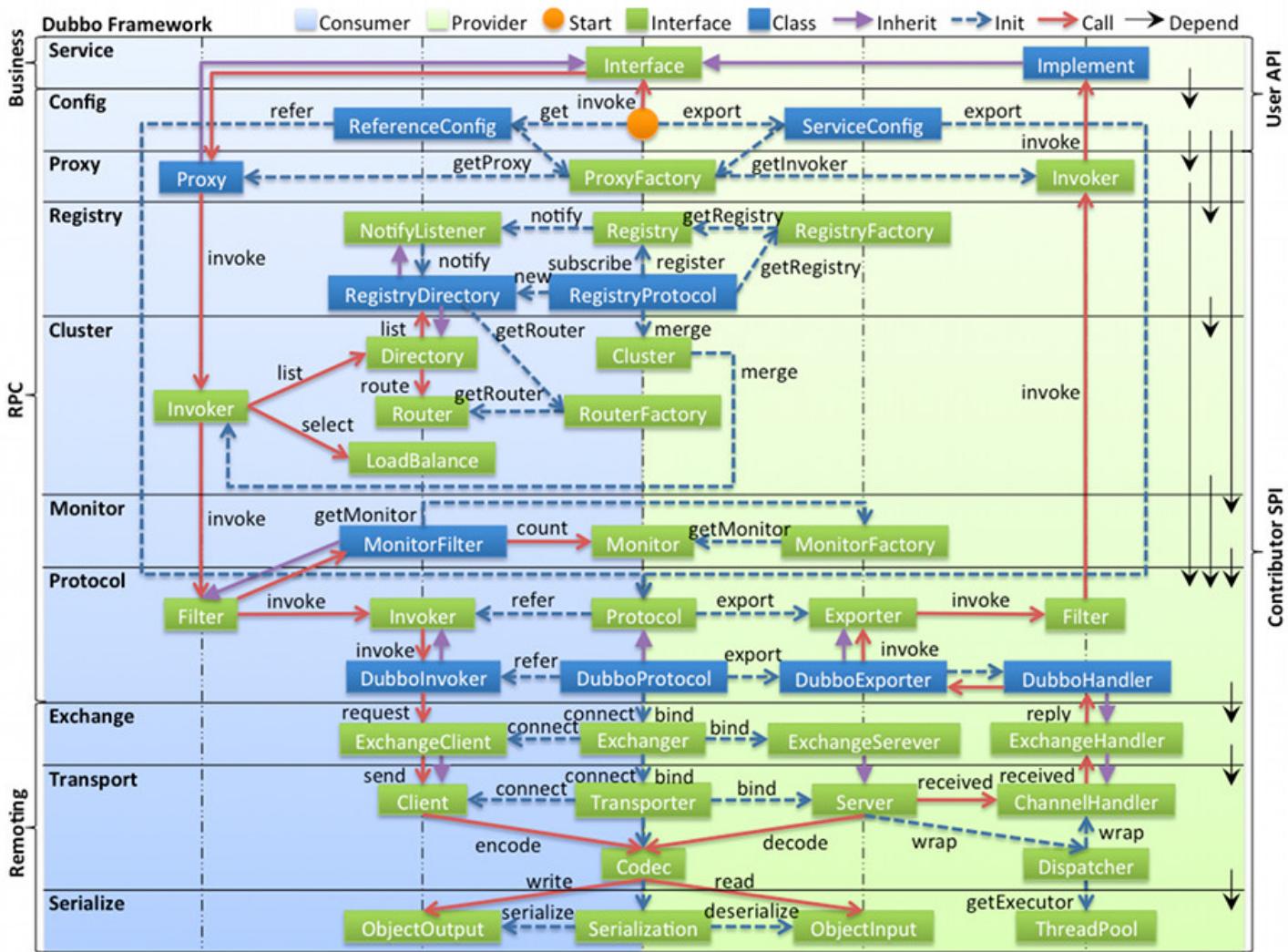
RPC框架存在的主要价值是解耦服务，让开发者能够几乎无感地调用远端服务器提供的接口实现，所有复杂的中间环节都被框架隐藏起来了。一个RPC远程调用过程，涉及的微服务接口，在客户端出站前和服务端入站后的大部分时间，都会被表达成一个 `Invoker<T>` 对象，而这个表达过程正是由框架代理层所担纲（《Dubbo微服务导入导出》一文中也多次表示原始 `Invoker<T>` 对象——`originInvoker` 是由代理层生成的）。

## 基础

了解原理和剖析源码是一枚硬币的两面，互为照应，无论以哪面作为学习研究的起点，都需要以另一面作为线索或者依据，只有这样，才能透彻地理解和掌握一个技术点的内核机制。因此接下来，我会带大家以上帝视觉从更加宏观的角度初步涉猎反面的大致原理和正面的源码类图结构。

## 原理

如下框架示意图可以作为理解Dubbo中关键参与组件的切入点， $\rightarrow$  初始化流程和  $\rightarrow$  调用流程会领着我们一览其大致运行机制。



架构图由中分线分成左边服务消费方C端和右边服务提供方P端，细节这里不做具体阐述，和本文密切相关的的初始化流程和调用流程如下表。

Table 1. **Invoker / Proxy**

流程类型	流程
C端 ->	<b>iProxy :</b> ...> <b>ReferenceConfig</b> ...> <b>ProxyFactory</b> ...> <b>Proxy</b>
	<b>iRemoting :</b> ...> <b>ReferenceConfig</b> ...> <b>DubboProtocol</b> ...> <b>DubboInvoker &amp; ExchangeClient</b>
C端 ->	<b>Proxy</b> — <b>invoke</b> —> <b>DubboInvoker</b> — <b>request</b> —> <b>ExchangeClient</b>

流程 类型	流程
P端 ->	<p><code>iProxy : ...&gt; ServiceConfig ---getInvoker---&gt; ProxyFactory ---&gt; Implement'sProxyInvoker</code></p> <p><code>iRemoting : ...&gt; ServiceConfig ---export---&gt; DubboProtocol</code>  <code>---export---&gt; DubboExporter &amp; DubboHandler</code>  <code>(←ExchangeHandler ∈ ExchangeServer)</code></p>
P端 →	<p><code>DubboHandler —received→ DubboExporter —invoke→</code>  <code>Implement'sProxyInvoker —call→ Implement</code></p>

## NOTE

类的继承或接口实现关系: `Implement'sProxyInvoker | DubboInvoker → Invoker`、  
`Implement | Proxy → Interface`、`ExchangeClient → Client`、`DubboProtocol → Protocol`、`DubboExporter → Exporter`、  
`DubboHandler → ExchangeHandler → ChannelHandler`。

可以看出无论是C端还是P端均有着两条泾渭分明、起始于配置层的初始化流程，其中一条结束于调用代理处，另外一条则结束于底层通讯处，我们假定前者为 `iProxy`，后者为 `iRemoting`，显然他们存在着间接调用关系的，C端对应 `iProxy → iRemoting`，而P端则反过来为 `iRemoting → iProxy`。

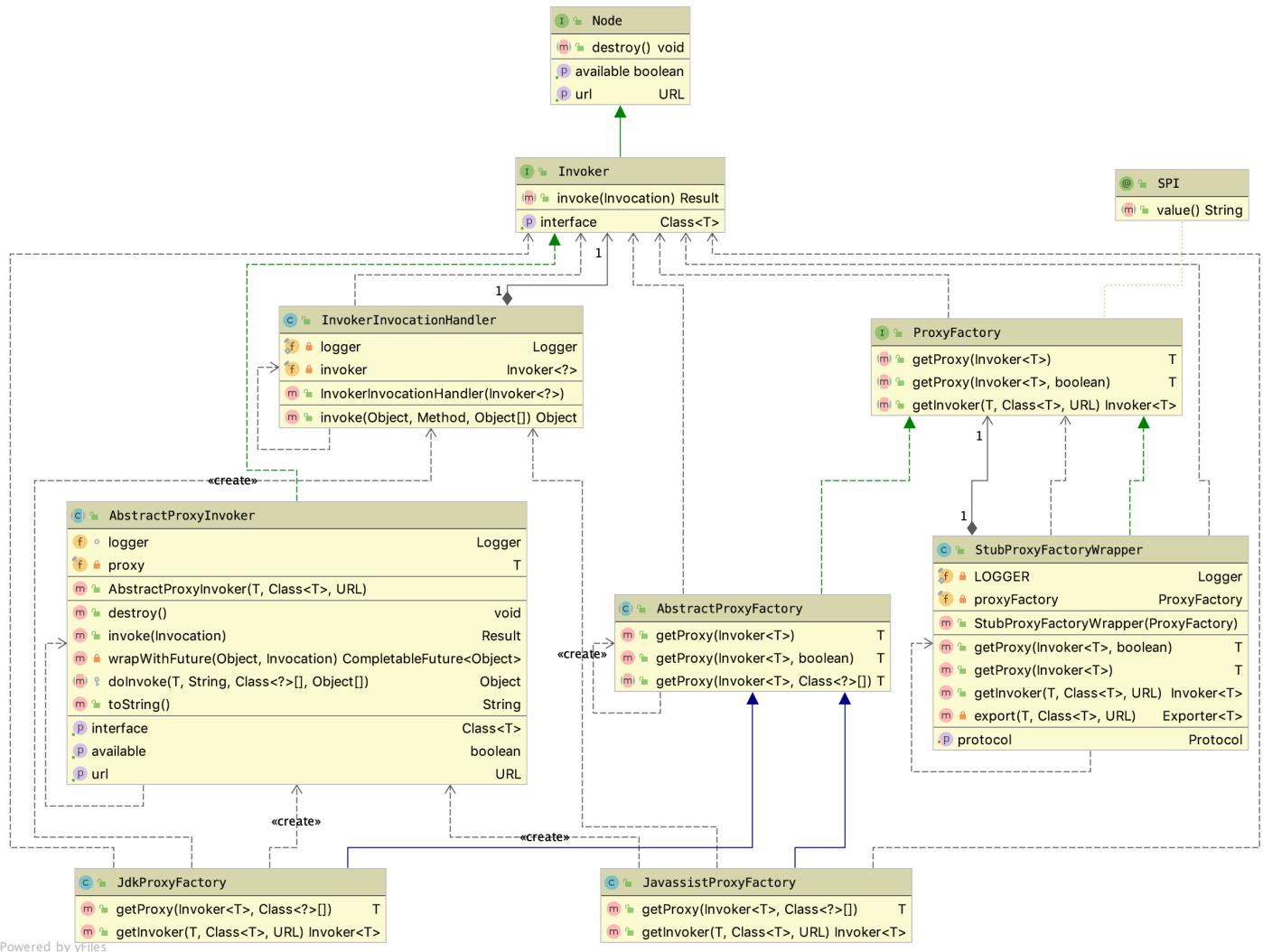
这时关于代理层的运作机制就已经颇为明晰了：

- 在C端的作用是框架在利用配置层进行微服务引用的过程中，会由代理层为服务端定义的接口生成一个 `proxy` 代理实现，由它将本地方法调用中涉及的包括方法名、入参极其类型转换为 `DubboInvoker#invoke(invocation)` 方法的调用入参——一个 `RpcInvocation` 类的对象；
- 在P端，则会将服务方提供的接口实现实例化成一个 `Invoker` 对象，利用Java的反射机制将 `Invoker#invoke(invocation)` 方法中的入参转换为对服务接口相应方法的调用；

## 类图

代理层实现是以扩展点 `ProxyFactory` 为中心展开，如下类图，扩展点实现 `JdkProxyFactory` 和 `JavassitProxyFactory` 都继承自抽象类 `AbstractProxyFactory`，而它们都被 `StubProxyFactoryWrapper` 包装了。 `ProxyFactory` 扩展点具类会利用抽象类

`AbstractProxyInvoker` 以匿名内部类的形式产生 `originInvoker`。右边的 `StubProxyFactoryWrapper` 则利用了Dubbo的SPI机制中的自动包装AOP特性，在动态代理的基础上利用静态代理实现了本地存根特性。



核心接口 `ProxyFactory` 处于框架的中分线位置，既用于客户端（C端），也被用于服务端（P端），对应分别提供了 `getProxy(...)` 和 `getInvoker(...)`，接口定义如下：

```

@SPI("javassist")
public interface ProxyFactory {

    @Adaptive({PROXY_KEY})
    <T> T getProxy(Invoker<T> invoker) throws RpcException;

    @Adaptive({PROXY_KEY})
    <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException;

    @Adaptive({PROXY_KEY})
    <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) throws RpcException;
}
  
```

扩展点声明中，使用注解明确了默认使用 `JavassistProxyFactory` 这个实现，除非另行指定 `url["proxy"] = "jdk"`。

## 具体实现剖析

一个微服务接口，代理层会为其在消费端生成由上面应用层调用的 `proxy` 代理实现，而在服务端则产生 `AbstractProxyInvoker<T>` 的匿名内部类的实例，供RPC框架底层调用其业务实现，暂且将它们分别看做是 C 端代理 和 P 端代理。

### C 端代理实现

Java 实现的框架中，大部分时候都会涉及动态代理，利用运行时代码或字节码生成技术为目标业务类或接口产生一个代理对象，以便在其方法调用的前后织入某些特性功能，或过滤、或增强，或拦截，抑或转置。其中转置的意思是“将符合某些特性的被代理对象的功能屏蔽掉，转给其他业务类完成并返回结果”。当然这只是比较常规的操作，因微服务接口的具体实现不在客户端本地，方法调用的信息会被全部转换成 `Invocation` 类对象，因而 Dubbo 中的代理使用就相当灵活，代理除了实现服务接口外，还会按需实现一些框架定义的接口，甚至是指定自定义的其它接口。

上述类图中已经反映了 Dubbo 为 `ProxyFactory` 提供了 JDK 和 Javassist 两个版本的实现，代理实现中涉及到的公共逻辑都被置于抽象基类 `AbstractProxyFactory` 中，如下：

```
public abstract class AbstractProxyFactory implements ProxyFactory {
```

JAVA

```
    @Override
    public <T> T getProxy(Invoker<T> invoker) throws RpcException {
        return getProxy(invoker, false);
    }

    @Override
    public <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException {
        Class<?>[] interfaces = null;
        String config = invoker.getUrl().getParameter(INTERFACES);
        if (config != null && config.length() > 0) {
            String[] types = COMMA_SPLIT_PATTERN.split(config);
            if (types != null && types.length > 0) {
                interfaces = new Class<?>[types.length + 2];
                interfaces[0] = invoker.getInterface();
                interfaces[1] = EchoService.class;
                for (int i = 0; i < types.length; i++) {
                    interfaces[i + 2] = ReflectUtils.forName(types[i]);
                }
            }
        }
        if (interfaces == null) {
            interfaces = new Class<?>[]{invoker.getInterface(), EchoService.class};
        }

        if (!GenericService.class.isAssignableFrom(invoker.getInterface()) && generic)
        {
            int len = interfaces.length;
            Class<?>[] temp = interfaces;
            interfaces = new Class<?>[len + 1];
            System.arraycopy(temp, 0, interfaces, 0, len);
            interfaces[len] = com.alibaba.dubbo.rpc.service.GenericService.class;
        }

        return getProxy(invoker, interfaces);
    }

    public abstract <T> T getProxy(Invoker<T> invoker, Class<?>[] types);
}
```

从上述源码可以看出，ProxyFactory 会为代理类中添加上以下接口的方法：

- invoker.getInterface()
- org.apache.dubbo.rpc.service.EchoService
- url["interfaces"]，可选
- org.apache.dubbo.rpc.service.GenericService，可选，generic = true 的情况下

GenericService 接口的定义如下，关于 GenericService 接口在《Dubbo RPC 之 Protocol 协议层（一）》中有提及。EchoService 接口是用于实现回声测试，由框架默认实现，定义如下：

```
public interface EchoService {  
    Object $echo(Object message);  
}
```

JAVA

有关于C端代理的实现，JDK和Javassist这两个版本都显得过于简洁，如下：

```
public class JdkProxyFactory extends AbstractProxyFactory {  
  
    @Override  
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {  
        return (T)  
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),  
                           interfaces, new InvokerInvocationHandler(invoker));  
    }  
    ...  
}  
  
public class JavassistProxyFactory extends AbstractProxyFactory {  
  
    @Override  
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {  
        return (T) Proxy.getProxy(interfaces).newInstance(new  
InvokerInvocationHandler(invoker));  
    }  
    ...  
}
```

JAVA

可见 `InvokerInvocationHandler` 是被两个版本的实现公共使用的，如下源码，代理对象 `proxy` 经过它将接口方法调用通过 `invoker.invoke(new RpcInvocation(method, args)).recreate()` 委托给了微服务引用实例 `invoker`，除了 `toString()`、`hashCode()`、`equals(Object)` 这几个方法，其它的都是跨机器进程的RPC远程调用：

```
public class InvokerInvocationHandler implements InvocationHandler {
    private static final Logger logger =
LoggerFactory.getLogger(InvokerInvocationHandler.class);
    private final Invoker<?> invoker;

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

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

        return invoker.invoke(new RpcInvocation(method, args)).recreate();
    }
}
```

JAVA

看起来很简单，是不是？然而，到这里，脑袋中可能总觉得有些地方还是模糊不清的，也许更加直白和可接受的方式是给一个具体的实例，`so, show me demo`。如下源码是给服务方定义的接口 `DemoService` 生成的代理类，其中全局共有静态属性 `methods` 是一个 `Method` 类型的数组，装有 `EchoService`、`DemoService` 这两个接口的方法定义信息，当前代理对象的行为委托是通过 `InvocationHandler` `handler` 完成的。

```

public class proxy0 implements ClassGenerator.DC, EchoService, DemoService {
    // 方法数组
    public static Method[] methods;
    private InvocationHandler handler;

    public proxy0(InvocationHandler invocationHandler) {
        this.handler = invocationHandler;
    }

    public proxy0() {
    }

    public String sayHello(String string) {
        // 将参数存储到 Object 数组中
        Object[] arrobject = new Object[]{string};
        // 调用 InvocationHandler 实现类的 invoke 方法得到调用结果
        Object object = this.handler.invoke(this, methods[0], arrobject);
        // 返回调用结果
        return (String)object;
    }

    /**
     * 回声测试方法 */
    public Object $echo(Object object) {
        Object[] arrobject = new Object[]{object};
        Object object2 = this.handler.invoke(this, methods[1], arrobject);
        return object2;
    }
}

```

## JavassistProxyFactory

### 本地存根实现

一个扩展点实现，若存在一个构造函数，其入参类型和该具类所实现的扩展点一样，则被认作是一个 Wrapper类，用于支持Dubbo的AOP特性。类图章节中已经提到代理层的本地存根 Stub 正是利用 AOP实现的，关于 Stub，官方文档有着准确详述的描述，如下：

“远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub，然后把 Stub 暴露给用户，Stub 可以决定要不要去调 Proxy。”

从描述可知，本地存根的基本思路是，开发者在客户端为微服务接口在本地定义一个 Stub 静态代理，接口方法实现时，会根据业务需要，使用本地实现的代码逻辑包裹 Proxy 的远程RPC调用。假定给定的微服务接口类型为 IServ，由于 Stub 有了如下三个显性要求，因此本地存根可以非常方便的利用反射机制实现：

- 必须实现自 IServ;
- 有一个唯一入参为 IServ 的构造函数;
- 若设 url[("stub" | "local")] = ("true" | "default")，类名必须是 IServ 的全名加上 "Stub" 或 "Local" 后缀，否则类名要明确配置在 url[("stub" | "local")] 中；

下面是两种形式的存根配置，相较于第一种，第二种显然有着更大的灵活性：

```
<!--①-->
<dubbo:service interface="com.foo.BarService" stub="true" />
<!--②-->
<dubbo:service interface="com.foo.BarService" stub="com.foo.impl.MyBarService" />
```

XML

getProxy(invoker) 源码如下所示，方法使用了逻辑后置实现方式，非泛化调用的情况下：

1. 首先由被代理的 proxyFactory 获得 proxy，由存根配置获取到存根实现的类元信息，并确保它实现了微服务的接口；
2. 然后利用反射搜寻存根实现中唯一入参为接口类型的构造函数，将 proxy 作为入参调用该构造函数，随即将创建的实例赋值给 proxy；
3. 最后，按理到此处整个流程应该结束了，但如 Tag\_stub 标记处所示，还有一个存根服务的导出处理，详见下一章节；

JAVA

```

public class StubProxyFactoryWrapper implements ProxyFactory {
    public <T> T getProxy(Invoker<T> invoker) throws RpcException {
        T proxy = proxyFactory.getProxy(invoker);
        if (GenericService.class != invoker.getInterface()) {
            URL url = invoker.getUrl();
            String stub = url.getParameter(STUB_KEY, url.getParameter(LOCAL_KEY));
            if (ConfigUtils.isNotEmpty(stub)) {
                Class<?> serviceType = invoker.getInterface();
                if (ConfigUtils.isDefault(stub)) {
                    if (url.hasParameter(STUB_KEY)) {
                        stub = serviceType.getName() + "Stub";
                    } else {
                        stub = serviceType.getName() + "Local";
                    }
                }
            }
            try {
                Class<?> stubClass = ReflectUtils.forName(stub);
                if (!serviceType.isAssignableFrom(stubClass)) {
                    throw new IllegalStateException("The stub implementation class
"
                                            + stubClass.getName() + " not implement interface "
                                            + serviceType.getName());
                }
                try {
                    Constructor<?> constructor = ReflectUtils.findConstructor(
                        stubClass, serviceType);
                    proxy = (T) constructor.newInstance(new Object[]{proxy});
                    ...//Tag_stub: export stub service
                } catch (NoSuchMethodException e) {
                    throw new IllegalStateException("No such constructor \"public "
                        + stubClass.getSimpleName() + "(" + serviceType.getName()
                        + ")\" in stub implementation class " +
                        stubClass.getName(), e);
                }
            } catch (Throwable t) {
                LOGGER.error("Failed to create stub implementation class " + stub
                    + " in consumer " + NetUtils.getLocalHost() + " use dubbo
version "
                    + Version.getVersion(), cause: " + t.getMessage(), t);
            }
        }
        return proxy;
    }
    ...
}

```

## 本地存根服务导出

并非所有存根包裹流程都会涉及存根服务的导出处理，只有当明确指定了 `url["dubbo.stub.event"] = true` 参数，也即本地存根实现需要提供除服务接口以外的其它 `public` 方法时，才有必要。导出的存根服务和普通服务是存在差异的，Dubbo通过URL配置总线识别

这种差异，识别参数总共 3 个，分别是 `url["dubbo.stub.event"]`、`url["isserver"]`、`url["dubbo.stub.event.methods"]`，对应的值分别为 `true`、`false`、`allStubProxyPublicMethods`。如下述代码所示，`StubProxyFactoryWrapper` 提供了一个 `export(...)` 方法，其中用于本地导出存根服务的 `protocol` 是一个扩展点，会经由 SPI 完成其自动注入，默认为 `DubboProtocol`，可以使用 `url.protocol` 指定其它实现。

```
//export stub service
public <T> T getProxy(Invoker<T> invoker) throws RpcException {
    ...
    URLBuilder urlBuilder = URLBuilder.from(url);
    if (url.getParameter(STUB_EVENT_KEY, DEFAULT_STUB_EVENT)) {
        urlBuilder.addParameter(STUB_EVENT_METHODS_KEY, StringUtils.join(
            Wrapper.getWrapper(proxy.getClass()),
            getDeclaredMethodNames(), ","));
        urlBuilder.addParameter(IS_SERVER_KEY, Boolean.FALSE.toString());
    } try {
        export(proxy, (Class) invoker.getInterface(),
            urlBuilder.build());
    } catch (Exception e) {
        LOGGER.error("export a stub service error.", e);
    }
}
...
}

private Protocol protocol;

public void setProtocol(Protocol protocol) {
    this.protocol = protocol;
}

private <T> Exporter<T> export(T instance, Class<T> type, URL url) {
    return protocol.export(proxyFactory.getInvoker(instance, type, url));
}
```

关于存根服务导出时的涉及相关细节请参考《Dubbo RPC 之 Protocol 协议层》一文中有关章节。

## P端代理实现

P 端的主要职责是将来源于客户端的RPC请求转为对本地服务接口实现的方法调用，同时将 P 端的业务请求响应统一成异步模式，按定义处理结果需经由 `Result` 承载并返回（`Result` 是一个可异步完成的 `CompletionStage<Result>` 类对象）。

总体逻辑实现由 `AbstractProxyInvoker<T>` 完成，为方便阐述，下述对应分为两个子章节：

### RPC请求转本地方法调用

如下述源码，入站的RPC请求体会被表达成一个调用 `Invoker<T>#invoke(invocation)` 方法的 `Invocation` 入参对象，针对特定微服务，指定它被调用的方法，并携带了该方法所需的入参及入参类型。

```
public interface Invoker<T> extends Node {  
    // get service interface.  
    Class<T> getInterface();  
  
    Result invoke(Invocation invocation) throws RpcException;  
}
```

JAVA

然而，一个理想的RPC框架，其跨进程的RPC方法调用，无论是从服务端还是消费端来看，于应用开发者而言都是无感的，也就是说，方法的定义形式和业务逻辑的实现过程均不应受到影响。

通常源码程序中，方法的调用是显示定义的，入参及入参类型必须和定义必须严格一一匹配，否则编译不会通过。然而服务端响应RPC请求是一个自下往上的方法调用过程，调用由RPC框架发起的，没法也不应该在源码程序中指定。多亏了Java中的反射机制，就一个目标对象，入参及入参类型已知的情况下，只需要知道被调用方法的字符串名称，便可正常地发起调用，只是若和定义不匹配时，会在运行时抛出错误，这种不匹配在编译时Java是无力发觉的。

另外一个Java方法，最多有两个出参，对应正常返回和异常返回，可以分别用 Object 和 Throwable 泛指。

综上，AbstractProxyInvoker<T> 中定义了如下抽象方法 doInvoke(...)，以便实现类提供 JDK 和 Javassist 两种反射实现：

```
protected abstract Object doInvoke(T proxy, String methodName,  
        Class<?>[] parameterTypes, Object[] arguments) throws Throwable;
```

JAVA

## 本地调用实现

doInvoke(...) 方法的实现于 jdk 版，相对比较简单，借助Java的反射机制，简短一段代码就搞定，如下：

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

JAVA

然而，javassist 版本的实现如下，看是源码也不多，但由于需要借助 Wrapper 完成，Wrapper 逻辑实现还是很好理解，后续有机会专门介绍。

```
public class JavassistProxyFactory extends AbstractProxyFactory {  
    ...  
    @Override  
    public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {  
        final Wrapper wrapper = Wrapper.getWrapper(  
            proxy.getClass().getName().indexOf('$') < 0 ? proxy.getClass() : type);  
        return new AbstractProxyInvoker<T>(proxy, type, url) {  
            protected Object doInvoke(T proxy, String methodName,  
                Class<?>[] parameterTypes,  
                Object[] arguments) throws Throwable {  
                return wrapper.invokeMethod(proxy, methodName, parameterTypes,  
                    arguments);  
            }  
        };  
    }  
}
```

## 本地方法调用结果的异步化

我们都清楚原生方法调用大多数情况下都是同步的，如果涉及IO阻塞操作，会造成长时间的等待，这样势必大幅降低了吞吐量。于跨进程的RPC调用而言，由于天然就涉及网络IO，情况只会更糟，会有数量级的降低，因而Dubbo框架会对RPC调用过程的相关环节尽可能最大程度地异步化，即便是发生在服务端的最后一个本地方法调用环节。当然这里所述的异步化是仅就主流程而言，

在Java中，绝大部分异步实现均依托于 `CompletableFuture`。如果原生方法定义的出参是它的一个实例，则无需另行处理；否则便用原生调用得到 `value` 值产生一个它的已完成状态实例，这种情况相当于服务端业务实现并没有异步化；最后一种便是后端开发人员使用类似于 `Servlet 3.0` 中的异步接口 `AsyncContext` 启用了 `RpcContext.startAsync()`，下文接下来将开辟新的章节重点介绍它。

```
private CompletableFuture<Object> wrapWithFuture (Object value, Invocation invocation)  
{  
    if (RpcContext.getContext().isAsyncStarted()) {  
        return ((AsyncContextImpl)  
(RpcContext.getContext().getAsyncContext())).getInternalFuture();  
    } else if (value instanceof CompletableFuture) {  
        return (CompletableFuture<Object>) value;  
    }  
    return CompletableFuture.completedFuture(value);  
}
```

从 `Invoker<T>#invoke(invocation)` 方法的定义可以看出，上述 `doInvoke(...)` 方法的 `Object` 出参类型数据最终需要相应转换为 `Result` 类型的对象。可见，RPC请求到本地原生方法调用的转换处理步骤很简洁：

1. 调用 `doInvoke(...)` 方法，由子类完成本地原生方法调用，执行结果缓存在 `value`；
2. 调用 `wrapWithFuture(...)` 方法，将 `Object` 类型的 `value` 封装成一个 `CompletableFuture<Object>` 对象，记为 `future`；
3. 实例化一个 `AsyncRpcResult` 对象，记为 `asyncRpcResult`，作为本地原生方法调用结果的承载体，也是当前方法的返回值；
4. 在 `future.whenComplete(...)` 异步响应回调中，为 `asyncRpcResult` 填充结果，告知处理完成（关于 `AsyncRpcResult` 和 `AppResponse`，详见《Dubbo RPC 之 Protocol 协议层》一文）。

JAVA

```

@Override
public Result invoke(Invocation invocation) throws RpcException {
    try {
        Object value = doInvoke(proxy, invocation.getMethodName(),
            invocation.getParameterTypes(), invocation getArguments());
        CompletableFuture<Object> future = wrapWithFuture(value, invocation);
        AsyncRpcResult asyncRpcResult = new AsyncRpcResult(invocation);
        future.whenComplete((obj, t) -> {
            AppResponse result = new AppResponse();
            if (t != null) {
                if (t instanceof CompletionException) {
                    result.setException(t.getCause());
                } else {
                    result.setException(t);
                }
            } else {
                result.setValue(obj);
            }
            asyncRpcResult.complete(result);
        });
        return asyncRpcResult;
    } catch (InvocationTargetException e) {
        if (RpcContext.getContext().isAsyncStarted() &&
        !RpcContext.getContext().stopAsync()) {
            logger.error("Provider async started, but got an exception from " +
                "the original method, cannot write the exception back to consumer " +
                "because an async result may have returned the new thread.", e);
        }
        return AsyncRpcResult.newDefaultAsyncResult(null, e.getTargetException(),
            invocation);
    } catch (Throwable e) {
        throw new RpcException("Failed to invoke remote proxy method "
            + invocation.getMethodName() + " to " + getUrl() + ", cause: " +
            e.getMessage(), e);
    }
}

```

## P端 AsyncContext 异步方案

于单个服务接口实现来说，如果其业务执行需要阻塞，无论如何是需要有线程来执行的，异步并不会带来资源的节省或者RPC响应性能的提升。但放眼于整个服务进程，由于另行切换到用户线程中执行，可以避免因为长时间占用或阻塞Dubbo线程池中的线程而影响其他服务线程的执行，综合来讲会提升

整体吞吐量。

## P端异步编程

在基于Dubbo的RPC编程中，我们通常使用如下方式实现P端的异步化：

```
public interface AsyncService {
    CompletableFuture<String> sayHello(String name);
}

public class AsyncServiceImpl implements AsyncService {
    @Override
    public CompletableFuture<String> sayHello(String name) {
        RpcContext savedContext = RpcContext.getContext();
        // 建议为supplyAsync提供自定义线程池，避免使用JDK公用线程池
        return CompletableFuture.supplyAsync(() -> {
            System.out.println(savedContext.getAttachment("consumer-key1"));
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "async response from provider.";
        });
    }
}
```

上述这个方案有个缺点，就是规定了接口定义中的出参必须是 `CompletableFuture<?>`，那我们还有其他方法吗？别急，Dubbo已经为我们想得很周全，它提供了一个类似 `Servlet 3.0` 的异步接口 `AsyncContext`，即便方法出参没有声明成 `CompletableFuture<?>`，也能实现P端异步化，如下示例：

```
public interface AsyncService {
    String sayHello(String name);
}

public class AsyncServiceImpl implements AsyncService {
    public String sayHello(String name) {
        final AsyncContext asyncContext = RpcContext.startAsync();
        new Thread(() -> {
            // 如果要使用上下文，则必须要放在第一句执行
            asyncContext.signalContextSwitch();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 写回响应
            asyncContext.write("Hello " + name + ", response from provider.");
        }).start();
        return null;
    }
}
```

上述两个异步编程方案的服务暴露是完全一样的，如下：

```
<bean id="asyncService"
      class="org.apache.dubbo.samples.governance.impl.AsyncServiceImpl"/>

<dubbo:service interface="org.apache.dubbo.samples.governance.api.AsyncService"
      ref="asyncService"/>
```

JAVA

## AsyncContext 实现

从上面的第二异步方案示例中可以看出，接口实现返回值 `null` 并不是真实的业务响应结果，业务逻辑实现在另行开辟的线程中执行的，其处理结果是经由 `AsyncContext#write(value)` 返回给调用方的。结合上文 `wrapWithFuture(...)` 方法可以推断，`AsyncContext` 实质是，将当前RPC请求所需配套的 `CompletableFuture` 实例 `future` 缓存在线程本地上下文中，而其业务方法执行结果正是由 `write` 写入的。

当然，具体实现中，并没有将当前PRC调用所需的 `CompletableFuture` 实例直接捆绑到 `RpcContext` 上，如下源码，它声明了一个被认作是异步上下文的 `asyncContext` 属性，该上下文需要显示调用 `startAsync()` 方法才会开启，因而又得另行配合一个判定是否已开启的 `isAsyncStarted()` 方法。

```
public class RpcContext {
    ...
    private AsyncContext asyncContext;

    public static AsyncContext startAsync() throws IllegalStateException {
        RpcContext currentContext = getContext();
        if (currentContext.asyncContext == null) {
            currentContext.asyncContext = new AsyncContextImpl();
        }
        currentContext.asyncContext.start();
        return currentContext.asyncContext;
    }

    public boolean isAsyncStarted() {
        if (this.asyncContext == null) {
            return false;
        }
        return asyncContext.isAsyncStarted();
    }

    public boolean stopAsync() {
        return asyncContext.stop();
    }

    public AsyncContext getAsyncContext() {
        return asyncContext;
    }
}
```

JAVA

接下来先看看如下 `AsyncContext` 接口，定义了相关管理行为，从几个方法名称可以看出其使用方式，得先调用 `start()` 方法开启异步上下文，业务处理完后，调用 `write(value)` 写入结果，最后调用 `stop()` 做回收处理。

```
public interface AsyncContext {  
    void write(Object value);  
    boolean isAsyncStarted();  
    boolean stop();  
    void start();  
    void signalContextSwitch();  
}
```

JAVA

关于 `signalContextSwitch()`，其作用就是重放在 `startAsync()` 时刻缓存的Dubbo线程本地现场数据，确保当前用户线程能获得其启动前一时刻的相关环境信息。这场景和编程中经常用到闭包有点类似，对应Java中的是匿名内部类，我们经常使用它实现回调接口，比如响应一个鼠标事件，响应处理是晚于匿名内部类的实例生成时刻的，通常响应阶段需要保留生成时刻的变量值，也即意味着相应的变量具备不可变性，比如声明为 `final` 的基本类型变量，或者是通过拷贝方式生成一个 `final` 型的引用类型变量，并确保此后它不会被改变。

`RpcContext#startAsync()` 驱动了 `AsyncContext` 实现 `AsyncContextImpl` 的实例化，而后者在其构造函数中执行保留现场处理，而根据 `AsyncContext` 的使用方式，`startAsync()` 唤起时用户线程还没有启动，还在Dubbo线程中。Dubbo利用了类似Java原生的 `ThreadLocal` 机制做PRC相关操作的线程本地上下文变量的存取处理，如下源码所示，`RpcContext.get[Server]Context() ← RpcContext.restore[Server]Context(storedContext)` 是一对现场数据的保留和重放操作：

```
public class AsyncContextImpl implements AsyncContext {  
  
    private RpcContext storedContext;  
    private RpcContext storedServerContext;  
  
    public AsyncContextImpl() {  
        this.storedContext = RpcContext.getContext();  
        this.storedServerContext = RpcContext.getServerContext();  
    }  
  
    @Override  
    public void signalContextSwitch() {  
        RpcContext.restoreContext(storedContext);  
        RpcContext.restoreServerContext(storedServerContext);  
    }  
    ...  
}
```

JAVA

AsyncContextImpl 整体上显得相当干练，在 start() 启动时，为P端 AbstractProxyInvoker 构建 CompletableFuture<Object> 实例，记为 future，而 write(value) 则将用户线程执行的结果写入 future。

```
public class AsyncContextImpl implements AsyncContext {
    private CompletableFuture<Object> future;

    @Override
    public void start() {
        if (this.started.compareAndSet(false, true)) {
            this.future = new CompletableFuture<>();
        }
    }

    @Override
    public void write(Object value) {
        if (isAsyncStarted() && stop()) {
            if (value instanceof Throwable) {
                Throwable bizExe = (Throwable) value;
                future.completeExceptionally(bizExe);
            } else {
                future.complete(value);
            }
        } else {
            throw new IllegalStateException("The async response has probably been wrote
"
                +"back by another thread, or the asyncContext has been closed.");
        }
    }

    public CompletableFuture<Object> getInternalFuture() {
        return future;
    }

    ...
}
```

另外，为了确保开发人员线程安全地正确使用 AsyncContext，只会执行一个完整的 start() → write(value) 的流程，特意使用了 AtomicBoolean 类型的原子变量 started 和 stopped。由于 start() 和 stop() 即便执行多次，也都只会有一次执行是有效的，因此上述 write(value) 方法实现的首行源码便是先确保已启用 AsyncContext，紧接着执行 stop() 停用 AsyncContext，接下来的源码才在前两个操作都成功的情况下才执行 future 的完成操作。

```
public class AsyncContextImpl implements AsyncContext {  
    private final AtomicBoolean started = new AtomicBoolean(false);  
    private final AtomicBoolean stopped = new AtomicBoolean(false);  
  
    @Override  
    public boolean isAsyncStarted() {  
        return started.get();  
    }  
  
    @Override  
    public boolean stop() {  
        return stopped.compareAndSet(false, true);  
    }  
    ...  
}
```

---

JAVA

完结