

\Dubbo注册中心

注册中心概述

在Dubbo微服务体系中，注册中心是其核心组件之一。Dubbo通过注册中心实现了分布式环境中各服务之间的注册与发现，是各个分布式节点之间的纽带。其主要作用如下：

- 动态加入。一个服务提供者通过注册中心可以动态地把自己暴露给其他消费者，无须消费者逐个去更新配置文件。
- 动态发现。一个消费者可以动态地感知新的配置、路由规则和新的服务提供者，无须重启服务使之生效。
- 动态调整。注册中心支持参数的动态调整，新参数自动更新到所有相关服务节点。
- 统一配置。避免了本地配置导致每个服务的配置不一致问题

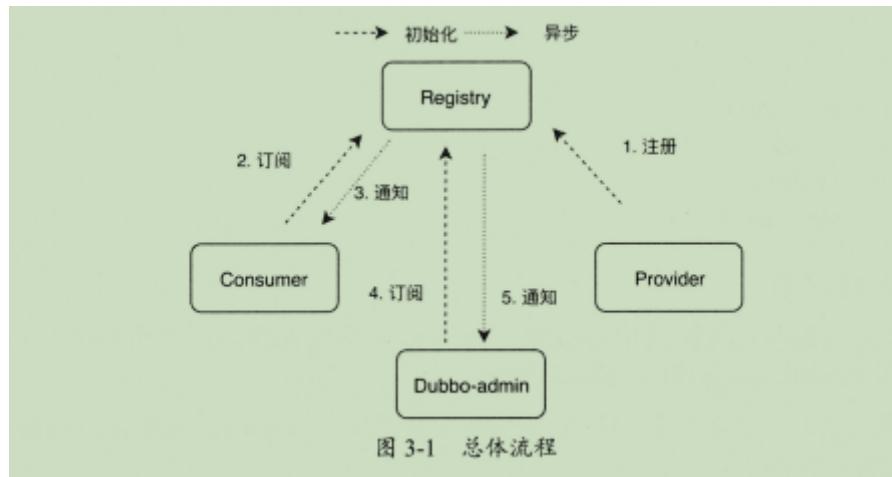
从dubbo-registry的模块中可以看到，Dubbo主要包含四种注册中心的实现，分别是ZooKeeper、Redis、Simple、Multicast。

其中ZooKeeper是官方推荐的注册中心，在生产环境中有过实际使用，具体的实现在Dubbo源码的dubbo-registry-zookeeper模块中。阿里内部并没有使用Redis作为注册中心，Redis注册中心并没有经过长时间运行的可靠性验证，其稳定性依赖于Redis本身。Simple注册中心是一个简单的基于内存的注册中心实现，它本身就是一个标准的RPC服务，不支持集群，也可能出现单点故障。Multicast模式则不需要启动任何注册中心，只要通过广播地址，就可以互相发现。服务提供者启动时，会广播自己的地址。消费者启动时，会广播订阅请求，服务提供者收到订阅请求，会根据配置广播或单播给订阅者。不建议在生产环境使用。

工作流程

注册中心的总体流程比较简单，Dubbo官方也有比较详细的说明，总体流程如图所示。

- 服务提供者启动时，会向注册中心写入自己的元数据信息，同时会订阅配置元数据信息。
- 消费者启动时，也会向注册中心写入自己的元数据信息，并订阅服务提供者、路由和配置元数据信息。
- 服务治理中心(dubbo-admin)启动时，会同时订阅所有消费者、服务提供者、路由和配置元数据信息。
第3章Dubbo注册中心
- 当有服务提供者离开或有新的服务提供者加入时，注册中心服务提供者目录会发生变化，变化信息会动态通知给消费者、服务治理中心。
- 当消费方发起服务调用时，会异步将调用、统计信息等上报给监控中心 dubbo-monitor simple。



ZooKeeper 原理概述

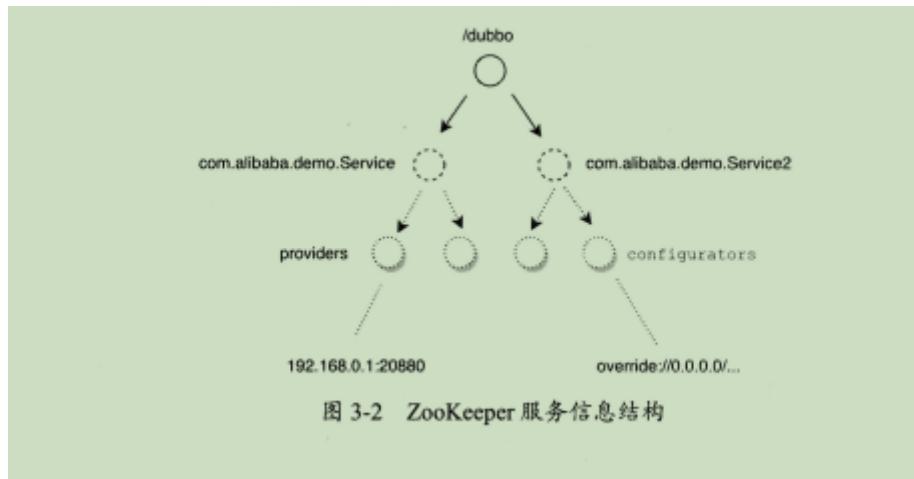
Dubbo使用ZooKeeper作为注册中心时，只会创建持久节点和临时节点两种，对创建的顺序并没有要求。

/dubbo/com.foo.BarService/providers是服务提供者在ZooKeeper注册中心的路径示例，是一种树形结构，该结构分为四层：root（根节点，对应示例中的dubbo）、service（接口名称对应示例中的com.foo.BarService）、四种服务目录（对应示例中的providers, other目录还有consumers、routers、configurators）。在服务分类节点下是具体的Dubbo服务URL。树形结构示例如下：

```
+ /dubbo
+-- service
    +-- providers
    +-- consumers
    +-- routers
    +-- configurators
```

树形结构的关系：

- (1) 树的根节点是注册中心分组，下面有多个服务接口，分组值来自用户配置dubbo:registry>中的group属性，默认是/dubbo。
- (2) 服务接口下包含4类子目录，分别是providers、consumers、routers、configurators这个路径是持久节点。
- (3) 服务提供者目录(/dubbo/service/providers)下面包含的接口有多个服务者URL元数据信息。
- (4) 服务消费者目录(/dubbo/service/consumers)下面包含的接口有多个消费者URL元数据信息。
- (5) 路由配置目录(/dubbo/service/routers)下面包含多个用于消费者路由策略URL元数据信息。
- (6) 动态配置目录(/dubbo/service/configurators)下面包含多个用于服务者动态配置URL元数据信息。



服务元数据中的所有参数都是以键值对形式存储的。以服务元数据为例：

dubbo://192.168.0.1.20880/com.alibaba.demo.Service?category=provider&name=demo-provider&..服务元数据中包含2个键值对，第1个key为category, key关联的值为provider。

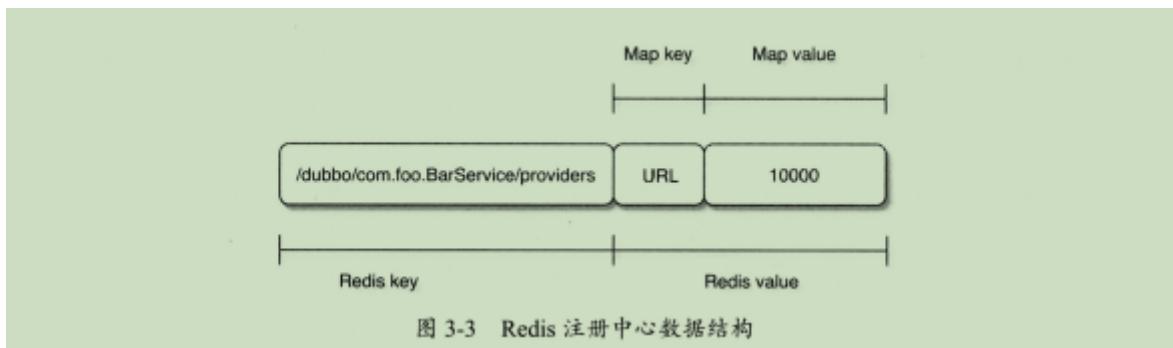
在Dubbo中启用注册中心可以参考如下方式：

```
<beans>
    <!--适用于ZooKeeper 一个集群有多个节点，多个IP和端口用逗号分隔-->
    <dubbo:registry protocol="zookeeper" address="ip:port>ip:port" />

    <!--适用于ZooKeeper多个集群有多个节点，多个IP和端口用竖线分隔-->
    <dubbo:registry protocol="zookeeper" address="ip:port|ip:port" />
</beans>
```

Redis原理概述

Redis注册中心也沿用了 Dubbo抽象的Root、Service> Type> URL四层结构。但是由于Redis属于NoSQL数据库，数据都是以键值对的形式保存的，并不能像ZooKeeper一样直接实现树形目录结构。因此，Redis使用了key/Map结构实现了这个需求，Root、Service、Type组合成Redis的keyo Redis的value是一个Map结构，URL作为Map的key,超时时间作为Map的value。



订阅/发布

ZooKeeper 的实现

发布的实现

ZooKeeper发布代码非常简单，只是调用了ZooKeeper的客户端库在注册中心上创建一个目录，如代码清单所示。

取消发布也很简单，只是把ZooKeeper注册中心上对应的路径删除，如代码清单所示。

```
//代码清单3-2 zkClient创建目录源码  
  
zkClient.create(toUrlPath(url)  
url.getParameter(Constants.DYNAMICKEY, true));  
  
//代码清单3-3 zkClient删除路径源码  
zkClient.delete(toUrlPath(url));
```

订阅的实现

Dubbo中有哪些ZooKeeper客户端实现？

无论服务提供者还是消费者，或者是服务治理中心，任何一个节点连接到ZooKeeper注册中心都需要使用一个客户端，Dubbo在dubbo-remoting-zookeeper模块中实现了ZooKeeper客户端的统一封装，定义了统一的client API，并用两种不同的ZooKeeper开源客户端库实现了这个接口：

- Apache Curator
- zkClient

<!--用户可以在<dubbo: registry>的client属性中设置curator、zkClient来使用不同的客户端实现库，如果不设置则默认使用Curator作为实现-->

订阅通常有pull和push两种方式，一种是客户端定时轮询注册中心拉取配置，另一种是注册中心主动推送数据给客户端。这两种方式各有利弊，目前Dubbo采用的是第一次启动拉取方式，后续接收事件重新拉取数据。

在服务暴露时，服务端会订阅configurators用于监听动态配置，在消费端启动时，消费端会订阅providers、routers和configurators这三个目录，分别对应服务提供者、路由和动态配置变更通知。

ZooKeeper注册中心采用的是“事件通知” + “客户端拉取”的方式，客户端在第一次连接上注册中心时，会获取对应目录下全量的数据。并在订阅的节点上注册一个watcher，客户端与注册中心之间保持TCP长连接，后续每个节点有任何数据变化的时候，注册中心会根据watcher的回调主动通知客户端（事件通知），客户端接到通知后，会把对应节点下的全量数据都拉取过来（客户端拉取），这一点在NotifyListener#notify(List urls)接口上就有约束的注释说明。全量拉取有一个局限，当微服务节点较多时会对网络造成很大的压力。

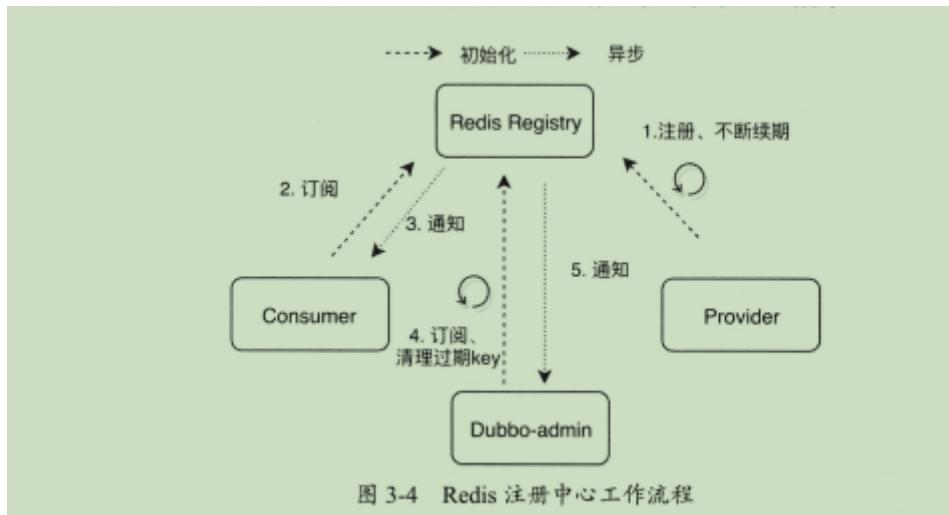
ZooKeeper的每个节点都有一个版本号，当某个节点的数据发生变化（即事务操作）时，该节点对应的版本号就会发生变化，并触发watcher事件，推送数据给订阅方。版本号强调的是变更次数，即使该节点的值没有变化，只要有更新操作，依然会使版本号变化。

Redis 的实现

使用Redis作为注册中心，其订阅发布实现方式与ZooKeeper不同。我们在Redis注册中心的数据结构中已经了解到，Redis订阅发布使用的是过期机制和publish/subscribe通道。服务提供者发布服务，首先会在Redis中创建一个key，然后在通道中发布一条register事件消息。但服务的key写入Redis后，发布者需要周期性地刷新key过期时间，在RedisRegistry构造方法中会启动一个expireExecutor定时调度线程池，不断调用deferExpired()方法去延续key的超时时间。如果服务提供者服务宕机，没有续期，则key会因为超时而被Redis删除，服务也就会被认定为下线。

订阅方首次连接上注册中心，会获取全量数据并缓存在本地内存中。后续的服务列表变化则通过publish/subscribe通道广播，当有服务提供者主动下线的时候，会在通道中广播一条unregister事件消息，订阅方收到后则从注册中心拉取数据，更新本地缓存的服务列表。新服务提供者上线也是通过通道事件触发更新的。

如果使用Redis作为服务注册中心，会依赖于服务治理中心。如果服务治理中心定时调度，则还会触发清理逻辑：获取Redis上所有的key并进行遍历，如果发现key已经超时，则删除Redis上对应的key。清除完后，还会在通道中发起对应key的unregister事件，其他消费者监听到取消注册事件后会删除本地对应服务的数据，从而保证数据的最终一致。



缓存机制

```
/*内存中的缓存notified是ConcurrentHashMap里面又嵌套了一个Map，外层Map的key是消费者的 URL，内层 Map 的 key 是分类，包含 providers> consumers> routes> configurators四种。value则是对应的服务列表，对于没有服务提供者提供服务的URL，它会以特殊的empty://前缀开头。*/
private final Properties properties = new Properties();
private File file; <—磁盘文件服务缓存对象
private final ConcurrentMap<URL, Map<String, List<URL>>> notified = new
ConcurrentHashMap<URL, Map<String, List<URL>>>();
```

缓存的加载

在服务初始化的时候，AbstractRegistry构造函数里会从本地磁盘文件中把持久化的注册数据读到Properties对象里，并加载到内存缓存中。

Properties保存了所有服务提供者的URL,使用URL#serviceKey()作为key,提供者列表、路由规则列表、配置规则列表等作为value。由于value是列表，当存在多个的时候使用空格隔开。还有一个特殊的key.registries,保存所有的注册中心的地址。如果应用在启动过程中，注册中心无法连接或宕机，则Dubbo框架会自动通过本地缓存加载Invokers。

缓存的保存与更新

缓存的保存有同步和异步两种方式。异步会使用线程池异步保存，如果线程在执行过程中出现异常，则会再次调用线程池不断重试，

```
if (syncSaveFile) {
    //同步保存
    doSaveProperties(version);
} else {
    //异步保存，放入线程池。会传入一个AtomicLong 的版本号，保证数据是最新的
    registryCacheExecutor.execute(new SaveProperties(version));
}
```

AbstractRegistry#notify方法中封装了更新内存缓存和更新文件缓存的逻辑。当客户端第一次订阅获取全量数据，或者后续由于订阅得到新数据时，都会调用该方法进行保存。

重试机制

com.alibaba.dubbo.registry.support.FailbackRegistry 继承了AbstractRegistry，并在此基础上增加了失败重试机制作为抽象能力。

ZookeeperRegistry和RedisRegistry继承该抽象方法后，直接使用即可。

FailbackRegistry抽象类中定义了一个ScheduledExecutorService，每经过固定间隔（默认为5秒）调用 FailbackRegistry#retry()方法。另外，该抽象类中还有五个比较重要的集合。

在定时器中调用retry方法的时候，会把这五个集合分别遍历和重试，重试成功则从集合中移除。
FailbackRegistry实现了 subscribe、unsubscribe等通用方法，里面调用了未实现的模板方法，会由子类实现。通用方法会调用这些模板方法，如果捕获到异常，则会把URL添加到对应的重试集合中，以供定时器去重试。

```
//发起注册失败的URL集合
private final ConcurrentHashMap<URL, FailedRegisteredTask> failedRegistered = new
ConcurrentHashMap<URL, FailedRegisteredTask>();
//取消注册失败的URL集合
private final ConcurrentHashMap<URL, FailedUnregisteredTask> failedUnregistered = new
ConcurrentHashMap<URL, FailedUnregisteredTask>();
//发起订阅失败的监听器集合
private final ConcurrentHashMap<Holder, FailedSubscribedTask> failedSubscribed = new
ConcurrentHashMap<Holder, FailedSubscribedTask>();
//取消订阅失败的监听器集合
private final ConcurrentHashMap<Holder, FailedUnsubscribedTask> failedUnsubscribed = new
ConcurrentHashMap<Holder, FailedUnsubscribedTask>();
//通知失败的URL集合
ConcurrentMap failedNotified
```

Dubbo扩展点加载机制

加载机制概述

基于Dubbo SPI加载机制，让整个框架的接口和具体实现完全解耦，从而奠定了整个框架良好可扩展性的基础。

Dubbo几乎所有的功能组件都是基于扩展机制(SPI)实现的。

Java SPI

Java SPI使用了策略模式，一个接口多种实现。我们只声明接口，具体的实现并不在程序中直接确定，而是由程序之外的配置掌控，用于具体实现的装配。具体步骤如下：

- (1) 定义一个接口及对应的方法。
- (2) 编写该接口的一个实现类。
- (3) 在META-INF/services/目录下，创建一个以接口全路径命名的文件，如com.test.spi.PrintService
- (4) 文件内容为具体实现类的全路径名，如果有多个，则用分行符分隔。
- (5) 在代码中通过java.util.ServiceLoader来加载具体的实现类。

如此一来，PrintService的具体实现就可以由文件com.test.spi.Printservice中配置的实现类来确定了。

```

public static void main(String[] args) {
    //调用SPI具体的实现
    ServiceLoader<PrintService> serviceServiceLoader =
    ServiceLoader.load(PrintService.class);
    for (Printservice printservice : serviceServiceLoader) {
        //此处会输出: hello world 获取所有的SPI实现，循环调用
        //通过java.util.ServiceLoader可以获取所有的接口实现，具体调用哪个实现，可以通过用户定制的规则来决定。
        printService.printInfo(); printInfo()方法，会打印出 hello world
    }
}

```

扩展点加载机制的改进

与Java SPI相比，Dubbo SPI做了一定的改进和优化，官方文档中有这么一段：

1. JDK标准的SPI会一次性实例化扩展点所有实现，如果有扩展实现则初始化很耗时，如果没用上也加载，则浪费资源。
2. 如果扩展加载失败，则连扩展的名称都获取不到了。比如JDK标准的ScriptEngine，通过getName()获取脚本类型的名称，如果RubyScriptEngine因为所依赖的jruby.jar不存在，导致RubyScriptEngine类加载失败，这个失败原因被“吃掉”了，和Ruby对应不起来，当用户执行Ruby脚本时，会报不支持Ruby，而不是真正失败的原因。
3. 增加了对扩展IOC和AOP的支持，一个扩展可以直接setter注入其他扩展。在Java SPI的使用示例章节（代码清单4-1）中已经看到，java.util.ServiceLoader会一次把Printservice接口下的所有实现类全部初始化，用户直接调用即可。Dubbo SPI只是加载配置文件中的类，并分成不同的种类缓存在内存中，而不会立即全部初始化，在性能上有更好的表现。具体的实现原理会在后面讲解，此处演示一个使用示例。我们把代码清单4-1中的Printservice改造成Dubbo SPI的形式。

扩展点的配置规范

Dubbo SPI和Java SPI类似，需要在META-INF/dubbo/下放置对应的SPI配置文件，文件名称需要命名为接口的全路径名。配置文件的内容为key=扩展点实现类全路径名，如果有多个实现类则使用换行符分隔。其中，key会作为DubboSPI注解中的传入参数。另外，Dubbo SPI还兼容了Java SPI的配置路径和内容配置方式。在Dubbo启动的时候，会默认扫这三个目录下的配置文件：META-INF/services/、META-INF/dubbo/、META-INF/dubbo/internal/。

表 4-1 Dubbo SPI 配置规范

规 范 名	规 范 说 明
SPI 配置文件路径	META-INF/services/、META-INF/dubbo/、META-INF/dubbo/internal/
SPI 配置文件名称	全路径类名
文件内容格式	key=value 方式，多个用换行符分隔

扩展点的分类与缓存

Dubbo SPI可以分为Class缓存、实例缓存。这两种缓存又能根据扩展类的种类分为普通扩展类、包装扩展类（Wrapper类）、自适应扩展类（Adaptive类）等。

- Class缓存：Dubbo SPI获取扩展类时，会先从缓存中读取。如果缓存中不存在，则加载配置文件，根据配置把Class缓存到内存中，并不会直接全部初始化。
- 实例缓存：基于性能考虑，Dubbo框架中不仅缓存Class，也会缓存Class实例化后的对象。每次获取的时候，会先从缓存中读取，如果缓存中读不到，则重新加载并缓存起来。这也是为什么Dubbo SPI相对Java SPI性能上有优势的原因，因为Dubbo SPI缓存的Class并不会全部实例化，而是按需实例化并缓存，因此性能更好。

被缓存的Class和对象实例可以根据不同的特性分为不同的类别：

- 1 普通扩展类。最基础的，配置在SPI配置文件中的扩展类实现。
- 2 包装扩展类。这种Wrapper类没有具体的实现，只是做了通用逻辑的抽象，并且需要在构造方法中传入一个具体的扩展接口的实现。属于Dubbo的自动包装特性。
- 3 自适应扩展类。一个扩展接口会有多种实现类，具体使用哪个实现类可以不写死在配置或代码中，在运行时，通过传入URL中的某些参数动态来确定。这属于扩展点的自适应特性。
- 4 其他缓存，如扩展类加载器缓存、扩展名缓存等。

扩展点的特性

1.自动包装

自动包装是提到的一种被缓存的扩展类，ExtensionLoader在加载扩展时，如果发现这个扩展类包含其他扩展点作为构造函数的参数，则这个扩展类就会被认为是Wrapper类

2.自动加载

除了在构造函数中传入其他扩展实例，我们还经常使用setter方法设置属性值。如果某个扩展类是另外一个扩展点类的成员属性，并且拥有setter方法，那么框架也会自动注入对应的扩展点实例。ExtensionLoader在执行扩展点初始化的时候，会自动通过setter方法注入对应的实现类。

如果扩展类属性是一个接口，它有多种实现，那么具体注入哪一个呢？这就涉及第三个特性——自适应。

3.自适应

我们使用@Adaptive注解，可以动态地通过URL中的参数来确定要使用哪个具体的实现类。从而解决自动加载中的多实例注入问题。

@Adaptive传入了两个Constants中的参数，它们的值分别是“server”和“transporter”。当外部调用Transporter#bind方法时，会动态从传入的参数“URL”中提取key参数“server”的value值，如果能匹配上某个扩展实现类则直接使用对应的实现类；如果未匹配上，则继续通过第二个key参数“transporter”提取value值。如果都没匹配上，则抛出异常。也就是说，如果@Adaptive中传入了多个参数，则依次进行实现类的匹配，直到最后抛出异常。

这种动态寻找实现类的方式比较灵活，但只能激活一个具体的实现类，如果需要多个实现类同时被激活，如Filter可以同时有多个过滤器；或者根据不同的条件，同时激活多个实现类，如何实现？这就涉及最后一个特性——自动激活

4.自动激活

使用@Activate注解，可以标记对应的扩展点默认被激活启用。该注解还可以通过传入不同的参数，设置扩展点在不同的条件下被自动激活。主要的使用场景是某个扩展点的多个实现类需要同时启用。

扩展点注解

扩展点注解：@SPI

@SPI注解可以使用在类、接口和枚举类上，Dubbo框架中都是使用在接口上。它的主要作用就是标记这个接口是一个Dubbo SPI接口，即是一个扩展点，可以有多个不同的内置或用户定义的实现。运行时需要通过配置找到具体的实现类。

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE})  
public @interface SPI {  
    /**  
     * default extension name
```

```

    *通过这个属性，我们可以传入不同的参数来设置这个接口的默认实现类。
    */
    String value() default "";
}

@SPI("netty")
public interface Transporter{
    ...
}

```

扩展点自适应注解：©Adaptive

@Adaptive注解可以标记在类、接口、枚举类和方法上，但是在整个Dubbo框架中，只有几个地方使用在类级别上，如AdaptiveExtensionFactory和AdaptiveCompiler，其余都标注在方法上。如果标注在接口的方法上，即方法级别注解，则可以通过参数动态获得实现类。方法级别注解在第一次getExtension时，会自动生成和编译一个动态的Adaptive类，从而达到动态实现类的效果。

在扩展点接口的多个实现里，只能有一个实现上可以加@Adaptive注解。如果多个实现类都有该注解，则会抛出异常：More than 1 adaptive class found.

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Adaptive {
    /**
     * Adaptive可以传入多个key值，在初始化Adaptive注解的接口时，会先对传入的URL进行key值匹配，第一个key没匹配上则匹配第二个，以此类推。直到所有的key匹配完毕，如果还没有匹配到，则会使用“驼峰规则”匹配，如果也没匹配到，则会抛出IllegalStateException异常。
     */
    //什么是“驼峰规则”呢？如果包装类（Wrapper）没有用Adaptive指定key值，则Dubbo会自动把接口名称根据驼峰大小写分开，并用符号连接起来，以此来作为默认实现类的名称，如org.apache.dubbo.xxx.YyyInvokerWrapper 中的 YyyInvokerWrapper 会被转换为yyy.invoker.wrapper
    String[] value() default {};
}

```

为什么有些实现类上会标注©Adaptive呢？放在实现类上，主要是为了直接固定对应的实现而不需要动态生成代码实现，就像策略模式直接确定实现类。在代码中的实现方式是：

ExtensionLoader中会缓存两个与©Adaptive有关的对象，一个缓存在cachedAdaptiveClass中，即Adaptive具体实现类的Class类型；另外一个缓存在cachedAdaptiveInstance中，即Class的具体实例化对象。在扩展点初始化时，如果发现实现类有@Adaptive注解，则直接赋值给cachedAdaptiveClass，后续实例化类的时候，就不会再动态生成代码，直接实例化cachedAdaptiveClass，并把实例缓存到cachedAdaptiveInstance中。如果注解在接口方法上，则会根据参数，动态获得扩展点的实现，会生成Adaptive类，再缓存到achedAdaptiveInstance中。

扩展点自动激活注解：@Activate

@Activate可以标记在类、接口、枚举类和方法上。主要使用在有多个扩展点实现、需要根据不同条件被激活的场景中，如Filter需要多个同时激活，因为每个Filter实现的是不同的功能。@Activate可传入的参数很多。

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Deprecated

```

```

public @interface Activate {
    //URL中的分组如果匹配则激活，则可以设置多个
    String[] group() default {};
    //查找URL中如果含有该key值，则会激活
    String[] value() default {};
    //填写扩展点列表，表示哪些扩展点要在本扩展点之前
    @Deprecated
    String[] before() default {};
    //填写扩展点列表，表示哪些扩展点要在本扩展点之后
    @Deprecated
    String[] after() default {};
    //整型，直接的排序信息
    int order() default 0;
}

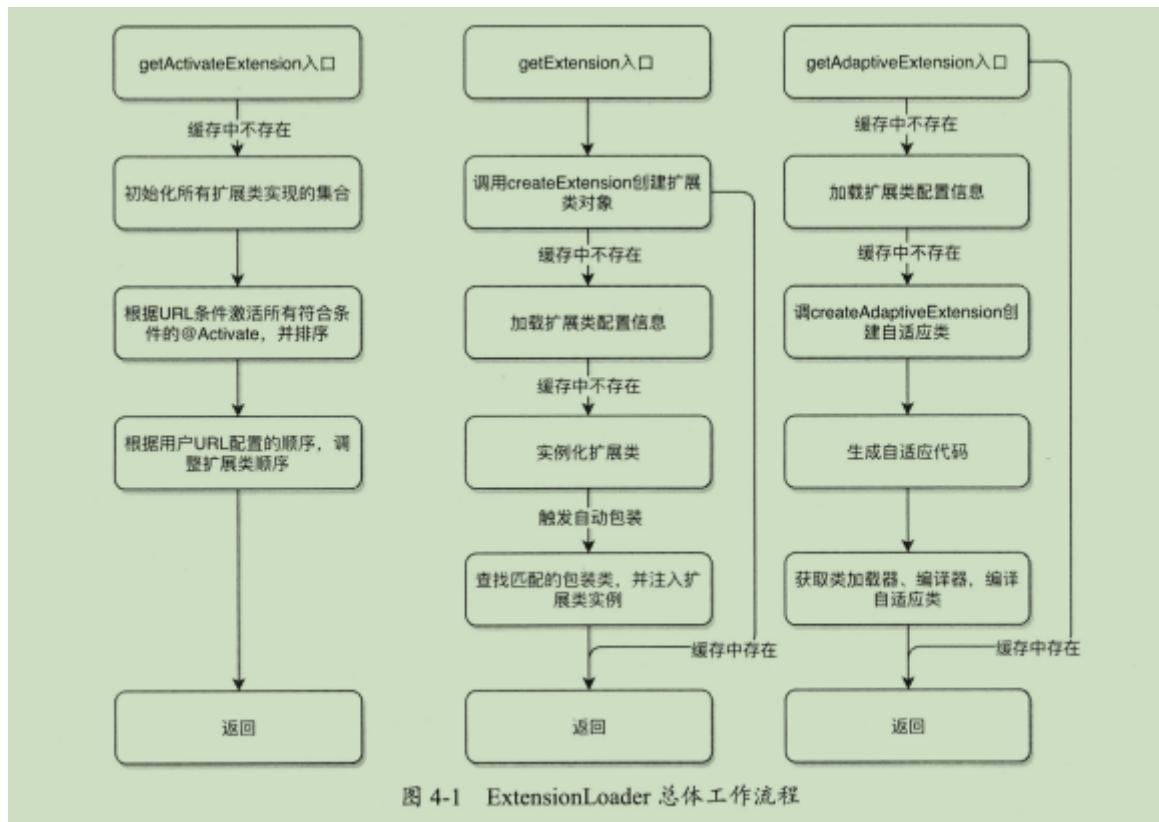
```

ExtensionLoader 的工作原理

ExtensionLoader是整个扩展机制的主要逻辑类，在这个类里面实现了配置的加载、扩展类缓存、自适应对象生成等所有工作.

工作流程

ExtensionLoader 的逻辑入口可以分为getExtension、getAdaptiveExtension、getActivateExtension 三个，分别是获取普通扩展类、获取自适应扩展类、获取自动激活的扩展类。总体逻辑都是从调用这三个方法开始的，每个方法可能会有不同的重载的方法，根据不同的传入参数进行调整，



getExtension (String name)是整个扩展加载器中最核心的方法，实现了一个完整的普通扩展类加载过程。加载过程中的每一步，都会先检查缓存中是否已经存在所需的数据，如果存在则直接从缓存中读取，没有则重新加载。这个方法每次只会根据名称返回一个扩展点实现类。

初始化的过程可以分为4步：

(1)框架读取SPI对应路径下的配置文件，并根据配置加载所有扩展类并缓存(不初始化)。

(2)根据传入的名称初始化对应的扩展类。

(3) 尝试查找符合条件的包装类：包含扩展点的setter方法，例如setProtocol(Protocol protocol)方法会自动注入protocol扩展点实现；包含与扩展点类型相同的构造函数，为其注入扩展类实例，例如本次初始化了一个Class A, 初始化完成后，会寻找构造参数中需要Class A的包装类(Wrapper),然后注入Class A实例，并初始化这个包装类。

(4) 返回对应的扩展类实例。

getAdaptiveExtension也相对独立，只有加载配置信息部分与**getExtension**共用了同一个方法。和获取普通扩展类一样，框架会先检查缓存中是否有已经初始化好的Adaptive实例，没有则调用**createAdaptiveExtension**重新初始化。初始化过程分为4步：

(1) 和**getExtension**一样先加载配置文件。

(2) 生成自适应类的代码字符串。

(3) 获取类加载器和编译器，并用编译器编译刚才生成的代码字符串。Dubbo 一共有三种类型的编译器实现。

(4) 返回对应的自适应类实例。

getExtension 的实现原理

在调用**createExtension**开始创建的过程中，也会先检查缓存中是否有配置信息，如果不存在扩展类，则会从 META-INF/services/、 META-INF/dubbo/、 META-INF/dubbo/internal/这几个路径中读取所有的配置文件，通过I/O读取字符流，然后通过解析字符串，得到配置文件中对应的扩展点实现类的全称(如 com.alibaba.dubbo.common.extensionloader.activate.impl.GroupActivateExtImpl).

getAdaptiveExtension 的实现原理

在**getAdaptiveExtension()**方法中，会为扩展点接口自动生成实现类字符串，实现类主要包含以下逻辑：为接口中每个有@Adaptive注解的方法生成默认实现(没有注解的方法则生成空实现)，每个默认实现都会从URL中提取Adaptive参数值，并以此为依据动态加载扩展点。然后，框架会使用不同的编译器，把实现类字符串编译为自适应类并返回。

生成代码的逻辑主要分为7步，具体步骤如下：

(1) 生成package、import、类名称等头部信息。此处只会引入一个类ExtensionLoader。为了不写其他类的import方法，其他方法调用时全部使用全路径。类名称会变为“接口名称\$Adaptive”的格式。例如：Transporter 接口会生成 Transporter\$Adaptive。

(2) 遍历接口所有方法，获取方法的返回类型、参数类型、异常类型等。为第(3)步判断是否为空值做准备。

(3)生成参数为空校验代码，如参数是否为空的校验。如果有远程调用，还会添加Invocation参数为空的校验。

(4) 生成默认实现类名称。如果@Adaptive注解中没有设定默认值，则根据类名称生成，如 YyyInvokerWrapper会被转换为yyy.invoker.wrappero生成的规则是不断找大写字母，并把它们用连接起来。得到默认实现类名称后，还需要知道这个实现是哪个扩展点的。

(5) 生成获取扩展点名称的代码。根据@Adaptive注解中配置的key值生成不同的获取代码，例如：如果是@Adaptive("protocol"),则会生成 ur1.getProtocol()

(6) 生成获取具体扩展实现类代码。最终还是通过**getExtension(extName)**方法获取自适应扩展类的真正实现。如果根据URL中配置的key没有找到对应的实现类，则会使用第(4)步中生成的默认实现类名称去找。

(7) 生成调用结果代码。

如果一个接口上既有@SPI("impl")注解，方法上又有@Adaptive("impl2")注解，那么会以哪个key作为默认实现呢？最终动态生成的实现方法会是url.getParameter("impl2", "impl"),即优先通过@Adaptive注解传入的key去查找扩展实现类；如果没找到，则通过@SPI注解中的key去查找；如果@SPI注解中没有默认值，则把类名转化为key,再去查找。

getActivateExtension 的实现原理

@Activate的实现原理，先从它的入口方法说起。

getActivateExtension(URL url, String key, String group)方法可以获取所有自动激活扩展点。参数分别是URL.URL中指定的key(多个则用逗号隔开)和URL中指定的组信息(group)。

其实现逻辑非常简单，当调用该方法时，主线流程分为4步：

- (1) 检查缓存，如果缓存中没有，则初始化所有扩展类实现的集合。
- (2) 遍历整个@Activate注解集合，根据传入URL匹配条件(匹配group、name等)，得到所有符合激活条件的扩展类实现。然后根据据@Activate中配置的before、after、order等参数进行排序
- (3) 遍历所有用户自定义扩展类名称，根据用户URL配置的顺序，调整扩展点激活顺序(遵循用户在URL中配置的顺序，例如 URL 为 test ://localhost/test?ext=order|default,则扩展点ext的激活顺序会遵循先order再default,其中default代表所有有@Activate注解的扩展点)。
- (4) 返回所有自动激活类集合。获取Activate扩展类实现，也是通过getExtension得到的。因此，可以认为getExtension是其他两种Extension的基石。

此处有一点需要注意，如果URL的参数中传入了-default,则所有的默认@Activate都不会被激活，只有URL参数中指定的扩展点会被激活。如果传入了

符号开头的扩展点名，则该扩展点也不会被自动激活。例如：-xxxx,表示名字为xxxx的扩展点不会被激活。

ExtensionFactory 的实现原理

我们可以知道ExtensionLoader类是整个SPI的核心。但是，ExtensionLoader类本身又是如何被创建的呢？

实现这个特性的ExtensionLoader类，本身又是通过工厂方法ExtensionFactory创建的，并且这个工厂接口上也有SPI注解，还有多个实现。

```
@SPI
public interface ExtensionFactory {
    /**
     * Get extension.
     *
     * @param type object type.
     * @param name object name.
     * @return object instance.
     */
    <T> T getExtension(Class<T> type, String name);
}
```

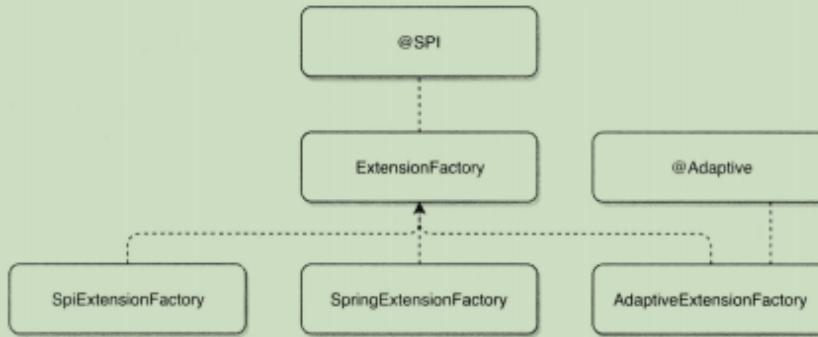


图 4-2 工厂类之间的关系

可以看到，除了 `AdaptiveExtensionFactory`,还有 `SpiExtensionFactory` 和 `SpringExtensionFactory`两个工厂。也就是说，我们除了可以从Dubbo SPI管理的容器中获取扩展点实例，还可以从Spring容器中获取。

那么Dubbo和Spring容器之间是如何打通的呢？ 我们先来看`SpringExtensionFactory`的实现，该工厂提供了保存Spring上下文的静态方法，可以把Spring上下文保存到Set集合中。当调用`getExtension`获取扩展类时，会遍历Set集合中所有的Spring上下文，先根据名字依次从每个Spring容器中进行匹配，如果根据名字没匹配到，则根据类型去匹配，如果还没匹配到则返回null

那么Spring的上下文又是在什么时候被保存起来的呢？在`ReferenceBean`和`ServiceBean`中会调用静态方法保存Spring上下文，即一个服务被发布或被引用的时候，对应的Spring 上下文会被保存下来。

我们再看一下`SpiExtensionFactory`,主要就是获取扩展点接口对应的Adaptive实现类。例如：某个扩展点实现类 `ClassA` 上有`@Adaptive` 注解，则调用 `SpiExtensionFactory#getExtension`会直接返回`ClassA`实例。

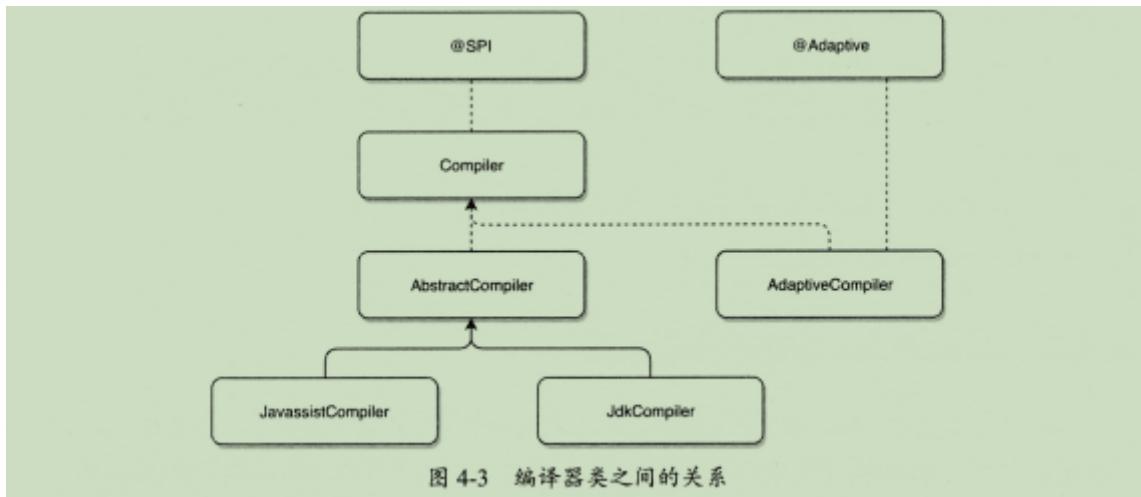
被`AdaptiveExtensionFactory`缓存的工厂会通过`TreeSet`进行排序，SPI排在前面，Spring排在后面。当调用`getExtension`方法时，会遍历所有的工厂，先从SPI容器中获取扩展类；如果没找到，则再从Spring容器中查找。我们可以理解为，`AdaptiveExtensionFactory`持有了所有的具体工厂实现，它的`getExtension`方法中只是遍历了它持有的所有工厂，最终还是调用SPI或Spring工厂实现的`getExtension`方法。

扩展点动态编译的实现

Dubbo SPI的自适应特性让整个框架非常灵活，而动态编译又是自适应特性的基础，因为动态生成的自适应类只是字符串，需要通过编译才能得到真正的Class。虽然我们可以使用反射来动态代理一个类，但是在性能上和直接编译好的Class会有一定的差距。Dubbo SPI通过代码的动态生成，并配合动态编译器，灵活地在原始类基础上创建新的自适应类。

总体结构

Dubbo中有三种代码编译器，分别是JDK编译器、Javassist编译器和AdaptiveCompiler编译器。这几种编译器都实现了 `Compiler`接口。



```

//Compiler接口上含有一个SPI注解，注解的默认值是@SPI("javassist")，很明显，Javassist编译器将作为默认编译器。如果用户想改变默认编译器，则可以通过<dubbo:application compiler="jdk" />标签进行配置。
@SPI("javassist")
public interface Compiler {
    Class<?> compile(String code, ClassLoader classLoader);
}

```

AdaptiveCompiler上面有@Adaptive注解，说明AdaptiveCompiler会固定为默认实现，这个Compiler的主要作用和AdaptiveExtensionFactory相似，就是为了管理其他Compile。

AdaptiveCompiler#setDefaultCompiler 方法会在 ApplicationConfig 中被调用，也就是 Dubbo 在启动时，会解析配置中的<dubbo:application compiler="jdk" />标签，获取设置的值，初始化对应的编译器。如果没有标签设置，则使用@SPI('javassist')中的设置，即JavassistCompiler。

Abstractcompiler的主要抽象逻辑如下：

- (1) 通过正则匹配出包路径、类名，再根据包路径、类名拼接出全路径类名。
- (2) 尝试通过Class.forName加载该类并返回，防止重复编译。如果类加载器中没有这个类，则进入第3步。
- (3) 调用doCompile方法进行编译。这个抽象方法由子类实现。

Javassist动态代码编译

代码清单 4-21 Javassist 使用示例

```
ClassPool classPool = ClassPool.getDefault(); ① 初始化 Javassist 的类池
CtClass ctClass = classPool.makeClass("Hello World"); ② 创建一个 Hello World 类
CtMethod ctMethod = CtNewMethod.make(" ③ 添加一个 test 方法, 会打印 Hello World,
    public static void test(){           直接传入方法的字符串
        System.out.println(\"Hello World\");
    }", ctClass);
ctClass.addMethod(ctMethod);
Class aClass = ctClass.toClass(); ④ 生成类

Object object = aClass.newInstance(); ⑤ 通过反射调用这个类实例
Method m = aClass.getDeclaredMethod("test", null);
m.invoke(object, null);

// ⑥ 控制台会打印出: Hello World
```

看完Javassist使用示例，其实Dubbo中DavassistCompiler的实现原理也很清晰了。由于我们之前已经生成了代码字符串，因此在JavassitCompiler中，就是不断通过正则表达式匹配不同部位的代码，然后调用Javassit库中的API生成不同部位的代码，最后得到一个完整的Class对象。具体步骤如下：

- (1) 初始化Javassit,设置默认参数，如设置当前的classpath。
- (2) 通过正则匹配出所有import的包，并使用Javassit添加import。
- (3) 通过正则匹配出所有extends的包，创建Class对象，并使用Javassit添加extends。
- (4) 通过正则匹配出所有implements包，并使用Javassit添加implements。
- (5) 通过正则匹配出类里面所有内容，即得到 {} 中的内容，再通过正则匹配出所有方法，并使用Javassit添加类方法。
- (6) 生成Class对象。

JavassitCompiler继承了抽象类Abstractcompiler,需要实现父类定义的一个抽象方法doCompileo以上步骤就是整个doCompile方法在JavassitCompiler中的实现。

JDK动态代码编译

JdkCompiler是Dubbo编译器的另一种实现，使用了JDK自带的编译器，原生JDK编译器包位于 javax.tools下。主要使用了三个东西：JavaFileObject 接口、ForwardingJavaFileManager 接口、JavaCompiler.CompilationTask 方法。整个动态编译过程可以简单地总结为：首先初始化一个 JavaFileObject 对象，并把代码字符串作为参数传入构造方法，然后调用 JavaCompiler.CompilationTask 方法编译出具体的类。JavaFileManager 负责管理类文件的输入/输出位置。以下是每个接口/方法的简要介绍：

- (1) JavaFileObject接口。字符串代码会被包装成一个文件对象，并提供获取二进制流的接口。Dubbo框架中的JavaFileObjectImpl类可以看作该接口一种扩展实现，构造方法中需要传入生成好的字符串代码，此文件对象的输入和输出都是ByteArray流。由于SimpleJavaFileObject、JavaFileObject之间的关系属于JDK中的知识。
- (2) JavaFileManager接口。主要管理文件的读取和输出位置。JDK中没有可以直接使用的实现类，唯一的实现类ForwardingJavaFileManager构造器又是protect类型。因此Dubbo中定制化实现了一个 JavaFileManagerImpl类，并通过一个自定义类加载器ClassLoaderImpl完成资源的加载。
- (3) JavaCompiler.CompilationTask 把 JavaFileObject 对象编译成具体的类。

Dubbo启停原理解析

配置解析

基于schema设计解析

Spring框架对Java产生了深远的影响，Dubbo框架也直接集成了Spring的能力，利用了Spring配置文件扩展出自定义的解析方式。Dubbo配置约束文件在dubbo-config/dubbo-config/spring/src/main/resources/dubbo.xsd中，在IntelliJ IDEA中能够自动查找这个文件，当用户使用属性时进行自动提示。

dubbo.xsd文件用来约束使用XML配置时的标签和对应的属性，比如Dubbo中的< dubbo:service>和< dubbo:reference>标签等。Spring在解析到自定义的namespace标签时（比如< dubbo:service>标签），会查找对应的spring.schemas和spring.handlers文件，最终触发Dubbo的DubboNamespaceHandler类来进行初始化和解析。我们先看以下两个文件的内容：

其中，spring.schemas文件指明约束文件的具体路径，spring.handlers文件指明DubboNamespaceHandler类来解析标签

```
// spring,schemas 文件  
http://dubbo.apache.org/schema/dubbo/dubbo.xsd=META-INF/dubbo.xsd  
http://code.alibabatech.com/schema/dubbo/dubbo.xsd=META-INF/compat/dubbo.xsd  
// spring.handlers  
http://dubbo.apache.org/schema/dubbo=org.apache.dubbo.config.spring.schema.Dubb  
oNamespaceHandler  
http://code.alibabatech.com/schema/dubbo=org.apache.dubbo.config.spring.schema.  
DubboNamespaceHandler
```

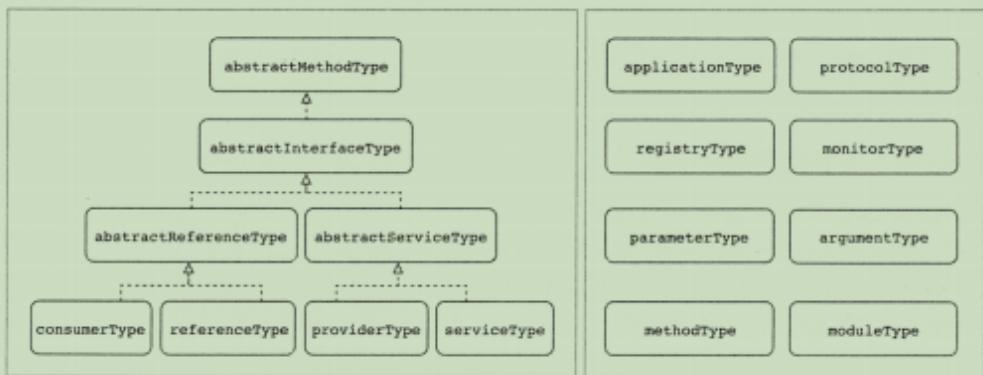


图 5-1 Dubbo schema 层级的详细设计

Dubbo 设计的粒度很多都是针对方法级别设计的，比如方法级别的 timeout、retries 和 mock 特性。这里包含的模块详细用法可以参考文档：<http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html>。在图 5-1 中，左边代表 schema 有继承关系的类型，右边是独立的类型。schema 模块说明如表 5-1 所示。

表 5-1 schema 模块说明

类型 定义	功 能 概 述
applicationType	配置应用级别的信息，比如应用的名称、应用负责人和应用的版本等
protocolType	配置服务提供者暴露的协议，Dubbo 允许同时配置多个协议，但只能有一个协议默认暴露
registryType	配置注册中心的地址和协议，Dubbo 也允许多个注册中心同时使用
providerType	配置服务提供方的全局配置，比如服务方设置了 timeout，消费方会自动透传超时
consumerType	配置消费方全局的配置，比如 connections 属性代表客户端会创建 TCP 的连接数，客户端全局配置会覆盖 providerType 透传的属性
serviceType	配置服务提供方接口范围信息，比如服务暴露的接口和具体实现类等
referenceType	配置消费方接口范围信息，比如引用的接口名称和是否泛化调用标志等
moduleType	配置应用所属模块相关信息
monitorType	配置应用监控上报相关地址
methodType	配置方法级别参数，主要应用于<dubbo:service>和<dubbo:reference>子标签
argumentType	配置应用方法参数等辅助信息，比如高级特性中异步参数回调索引的配置等
parameterType	选项参数配置，可以作为<dubbo:protocol>、<dubbo:service>、<dubbo:reference>、<dubbo:provider>和<dubbo:consumer>子标签，方便添加自定义参数，会透传到框架的 URL 中

接下来我们看一个dubbo.xsd的真实的配置，以protocolType模块为例。

我们可以简单理解其为协议定义约束字段，只有在这里定义的属性才会在Dubbo的XML配置文件中智能提示，当我们基于Dubbo做二次开发时，应该在schema中添加合适的字段，同时应该在dubbo-config-api对应的Config类中添加属性和get & set方法，这样用户在配置属性框架时会自动注入这个值。只有属性定义是不够的，为了让Spring正确解析标签，我们要定义element标签，与代码清单5-1中的protocolType进行绑定，这里以protocolType示例展示，如代码清单5-2所示。

代码清单5·1协议类型属性定义

```

<xsd:complexType name="protocolType">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The unique identifier for a bean. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>

```

```

        </xsd:attribute>
    <xsd:anyAttribute namespace="##other" processContents="lax">
</xsd:complexType>

代码清单5-2 定义标签配置
<xsd:element name="protocol" type="protocolType">
    <xsd:annotation>
        <xsd:documentation><![CDATA[ Service provider config ]]>
    </xsd:documentation>
    <xsd:appinfo>
        <tool:annotation>
            <tool:exports
type="org.apache.dubbo.config.ProtocolConfig"/>
        </tool:annotation>
    </xsd:appinfo>
    </xsd:annotation>
</xsd:element>

```

如果新增特性，比如增加epoll特性，则只需要在 providerType、 consumerType、 ProviderConfig 和 ConsumerConfig 中增加 epoll属性和方法即可。如果使用已经存在schema类型(比如说 protocolType),则只需要添加新属性即可，也不需要定义新的element标签。如果接口是级别通用的，一般我们只需要在interfaceType中增加属性即可，继承自interfaceType的类型会拥有该字段。同理，在Dubbo对应类AbstractInterfaceConfig中增加属性和方法即可。

基于XML配置原理解析

主要解析逻辑入口是在DubboNamespaceHandler类中完成的

```

@Override
public void init() {
    registerBeanDefinitionParser("application", new
DubboBeanDefinitionParser(ApplicationConfig.class, true));
    registerBeanDefinitionParser("module", new
DubboBeanDefinitionParser(ModuleConfig.class, true));
    registerBeanDefinitionParser("registry", new
DubboBeanDefinitionParser(RegistryConfig.class, true));
    registerBeanDefinitionParser("config-center", new
DubboBeanDefinitionParser(ConfigCenterBean.class, true));
    registerBeanDefinitionParser("metadata-report", new
DubboBeanDefinitionParser(MetadataReportConfig.class, true));
    registerBeanDefinitionParser("monitor", new
DubboBeanDefinitionParser(MonitorConfig.class, true));
    registerBeanDefinitionParser("metrics", new
DubboBeanDefinitionParser(MetricsConfig.class, true));
    registerBeanDefinitionParser("ssl", new
DubboBeanDefinitionParser(SslConfig.class, true));
    registerBeanDefinitionParser("provider", new
DubboBeanDefinitionParser(ProviderConfig.class, true));
    registerBeanDefinitionParser("consumer", new
DubboBeanDefinitionParser(ConsumerConfig.class, true));
    registerBeanDefinitionParser("protocol", new
DubboBeanDefinitionParser(ProtocolConfig.class, true));
    registerBeanDefinitionParser("service", new
DubboBeanDefinitionParser(ServiceBean.class, true));
    registerBeanDefinitionParser("reference", new
DubboBeanDefinitionParser(ReferenceBean.class, false));
}

```

```

        registerBeanDefinitionParser("annotation", new
AnnotationBeanDefinitionParser());
    }
}

```

DubboNamespaceHandler主要把不同的标签关联至解析实现类中。registerBeanDefinitionParser方法约定了在Dubbo框架中遇到标签application、module和registry等都会委托给DubboBeanDefinitionParser处理。

本质上都是把属性注入Spring框架的BeanDefinition。如果属性是引用对象，则Dubbo默认会创建RuntimeBeanReference类型注入，运行时由Spring注入引用对象。通过对属性解析的理解，其实Dubbo只做了属性提取的事情，运行时属性注入和转换都是Spring处理的。Dubbo框架生成的BeanDefinition最终还是会委托Spring创建对应的Java对象，dubbo.xsd中定义的类型都会有与之对应的POJO。

基于注解配置原理解析

重启开源后，Dubbo的注解已经完全重写了，因为原来注解是基于AnnotationBean实现的，主要存在以下几个问题：

- 注解支持不充分，需要XML配置<dubbo:annotation>；
- @ServiceBean不支持Spring AOP；
- @Reference不支持字段继承性

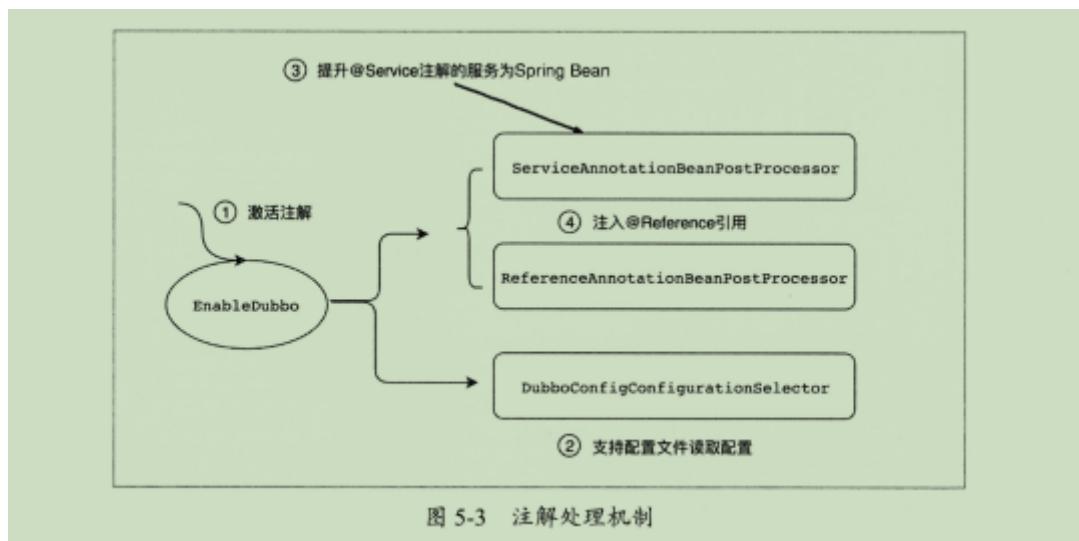


图 5-3 注解处理机制

注解处理逻辑主要包含3部分内容，第一部分是如果用户使用了配置文件，则框架按需生成对应Bean，第二部分是要将所有使用Dubbo的注解@Service的class提升为Bean，第三部分要为使用@Reference注解的字段或方法注入代理对象。

```

@EnableDubboConfig
@DubboComponentScan
public @interface EnableDubbo {
}
@Import(DubboConfigConfigurationSelector.class)//EnableDubboConfig 注解
public @interface EnableDubboConfig {
}
@Import(DubboComponentScanRegistrar.class)//DubboComponentScan 注解
public @interface DubboComponentScan (
)

```

当Spring容器启动的时候，如果注解上面使用`Import`,则会触发其注解方法`selectImports`,比如`EnableDubboConfig`注解中指定的`DubboConfigConfigurationSelector.class`,会自动触发`DubboConfigConfigurationSelector#selectImports`方法。如果业务方配置了Spring的`@PropertySource`或`XML`等价的配置(比如配置了框架`dubbo.registry.address`和`dubbo.application`等属性)，则Dubbo框架会在`DubboConfigConfigurationSelectort#selectImports`中自动生成相应的配置承载对象，比如`Applicationconfig`等。细心的读者可能发现`DubboConfigConfiguration`里面标注了`@EnableDubboConfigBindings`,`@EnableDubboConfigBindings`同样指定了`@Import(DubboConfigBindingsRegistrar.class)`。因为`@EnableDubboConfigBindings`允许指定多个`@EnableDubboConfigBinding`注解,Dubbo会根据用户配置属性自动填充这些承载的对象.

如何对服务提供者通过注`@Service`进行暴露的，注解扫描也委托给Spring,本质上使用asm库进行字节码扫描注解元数据，感兴趣的读者可以参考Spring源代码`SimpleMetadataReader`。当用户使用注解`@DubboComponentScan`时，会激活`DubboComponentScan#Registrar`,同时生成`ServiceAnnotationBeanPostProcessor`和`ReferenceAnnotationBeanPostProcessor`两种处理器，通过名称很容易知道分别是处理服务注解和消费注解。我们首先分析服务注解逻辑，因为`ServiceAnnotationBeanPostProcessor`处理器实现了`BeanDefinitionRegistryPostProcessor`接口，Spring容器中所有Bean注册之后回调`postProcessBeanDefinitionRegistry`方法开始扫描`@Service`注解并注入容器。

服务暴露的实现原理

配置承载初始化

不管在服务暴露还是服务消费场景下，Dubbo框架都会根据优先级对配置信息做聚合处理,目前默认覆盖策略主要遵循以下几点规则:

- (1) -D 传递给JVM参数优先级最高，比如`-Ddubbo.protocol.port=20880`
- (2) 代码或XML配置优先级次高，比如Spring中XML文件指定`dubbo:protocolport='20880'`
- (3) 配置文件优先级最低，比如`dubbo.properties`文件指定`dubbo.protocol.port=20880`

一般推荐使用`dubbo.properties`作为默认值，只有XML没有配置时，`dubbo.properties`配置项才会生效，通常用于共享公共配置，比如应用名等。

Dubbo的配置也会受到provider的影响，这个属于运行期属性值影响，同样遵循以下几点规则：

- (1) 如果只有provider端指定配置，则会自动透传到客户端(比如`timeout`)
- (2) 如果客户端也配置了相应属性，则服务端配置会被覆盖(比如`timeout`)

运行时属性随着框架特性可以动态添加，因此覆盖策略中包含的属性没办法全部列出来，一般不允许透传的属性都会在`ClusterUtils#mergeUrl`中进行特殊处理。

远程服务的暴露机制

整体RPC的暴露原理

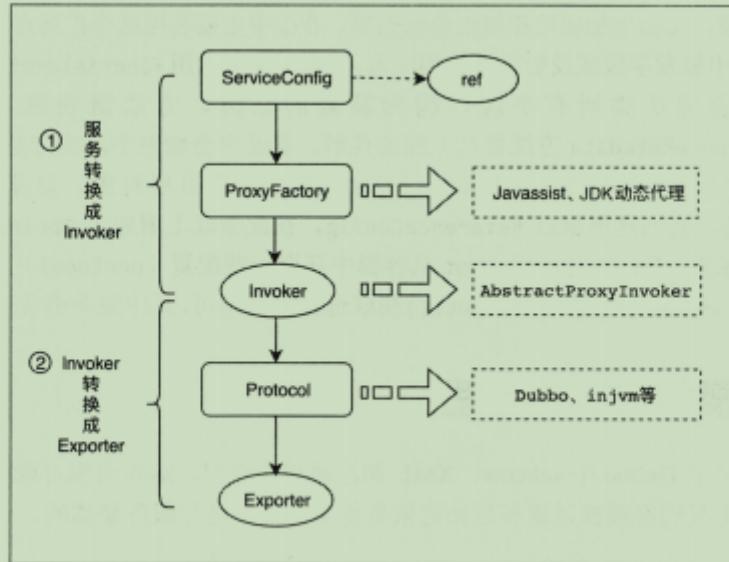


图 5-4 服务暴露整体机制

在整体上看，Dubbo框架做服务暴露分为两大部分，第一步将持有的服务实例通过代理转换成Invoker，第二步会把Invoker通过具体的协议（比如Dubbo 转换成Exporter,框架做了这层抽象也大大方便了功能扩展。这里的Invoker可以简单理解成一个真实的服务对象实例，是Dubbo框架实体域，所有模型都会向它靠拢，可向它发起invoke调用。它可能是一个本地的实现，也可能是一个远程的实现，还可能是一个集群实现。

框架真正进行服务暴露的入口点在ServiceConfig#doExport中，无论XML还是注解，都会转换成ServiceBean,它继承自ServiceConfig.主要处理思路就是遍历服务的所有方法，如果没有值则尝试从-D选项中读取，如果还没有则自动从配置文件dubbo.properties中读取。

Dubbo支持多注册中心同时写，如果配置了服务同时注册多个注册中心，则会在ServiceConfig#doExportUrls中依次暴露.

代码清单 5-11 多协议多注册中心暴露

```

private void doExportUrls() {
    List<URL> registryURLs = loadRegistries(true); ① 获取当前服务对  
应的注册中心实例
    for (ProtocolConfig protocolConfig : protocols) {
        doExportUrlsFor1Protocol(protocolConfig, registryURLs); ② 如果服务指定暴露多个协议 (Dubbo、REST)，则依次暴露服务
    }
}

```

Dubbo也支持相同服务暴露多个协议，比如同时暴露Dubbo和REST协议，框架内部会依次对使用的协议都做一次服务暴露，每个协议注册元数据都会写入多个注册中心。在①中会自动获取用户配置的注册中心，如果没有显示指定服务注册中心，则默认会用全局配置的注册中心。在②中处理多协议服务暴露的场景，真实服务暴露逻辑是在doExportUrlsFor1Protocol方法中实现的。

在doExportUrlsFor1Protocol中进行暴露的代码逻辑。

- ①主要通过反射获取配置对象并放到map中用于后续构造URL参数(比如应用名等)。
- ②主要区分全局配置，默认在属性前面增加default.前缀，当框架获取URL中的参数时，如果不存在则会自动尝试获取default.前缀对应的值。
- ③主要处理本地内存JVM协议暴露。
- ④主要追加监控上报地址，框架会在拦截器中执行数据上报，这部分是可选的。

⑤会通过动态代理的方式创建Invoker对象，在服务端生成的是AbstractProxyInvoker实例，所有真实的方法调用都会委托给代理，然后代理转发给服务ref调用。目前框架实现两种代理：JavassistProxyFactory和JdkProxyFactory。

JavassistProxyFactory模式原理：创建Wrapper子类，在子类中实现invokeMethod方法，方法体内会为每个ref方法都做方法名和方法参数匹配校验，如果匹配则直接调用即可，相比JdkProxyFactory省去了反射调用的开销。JdkProxyFactory模式是我们常见的用法，通过反射获取真实对象的方法，然后调用即可。

⑥主要先触发服务暴露(端口打开等)，然后进行服务元数据注册。

⑦主要处理没有使用注册中心的场景，直接进行服务暴露，不需要元数据注册，因为这里暴露的URL信息是以具体RPC协议开头的，并不是以注册中心协议开头的。

为了更容易地理解服务暴露与注册中心的关系，以下列表项分别展示有注册中心和无注册中心的URL：

- registry://host:port/com.alibaba.dubbo.registry.RegistryService?
protocol=zookeeper&export=dubbo://ip:port/xxx?...
- dubbo://ip:host/xxx.Service?timeout=1000&...

protocol实例会自动根据服务暴露URL自动做适配，有注册中心场景会取出具体协议，比如ZooKeeper，首先会创建注册中心实例，然后取出export对应的具体服务URL，最后用服务URL对应的协议(默认为Dubbo)进行服务暴露，当服务暴露成功后把服务数据注册到ZooKeeper。如果没有注册中心，则在⑦中会自动判断URL对应的协议(Dubbo)并直接暴露服务，从而没有经过注册中心。

在将服务实例ref转换成Invoker之后，如果有注册中心时，则会通过RegistryProtocol#export进行更细粒度的控制，比如先进行服务暴露再注册服务元数据。注册中心在做服务暴露时依次做了以下几件事情。

- (1) 委托具体协议(Dubbo)进行服务暴露，创建NettyServer监听端口和保存服务实例。
- (2) 创建注册中心对象，与注册中心创建TCP连接。
- (3) 注册服务元数据到注册中心。
- (4) 订阅configurators节点，监听服务动态属性变更事件。
- (5) 服务销毁收尾工作，比如关闭端口、反注册服务信息等。

代码清单 5-13 注册中心控制服务暴露

```

public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker); ←
    ① 打开端口，把服务实例存储到 map
    URL registryUrl = getRegistryUrl(originInvoker);
    final Registry registry = getRegistry(originInvoker); ← ② 创建注册中心实例
    final URL registeredProviderUrl = getRegisteredProviderUrl(originInvoker);

    boolean register = registeredProviderUrl.getParameter("register", true);
    ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl,
    registeredProviderUrl);

    if (register) {
        register(registryUrl, registeredProviderUrl); ← ③ 服务暴露之后，注册服务元数据
    }
}

```

102 | 深入理解 Apache Dubbo 与实战

```

    ProviderConsumerRegTable.getProviderWrapper(originInvoker).setReg(true);
}

final URL overrideSubscribeUrl = getSubscribedOverrideUrl(registeredProviderUrl);
final OverrideListener overrideSubscribeListener = new
OverrideListener(overrideSubscribeUrl, originInvoker);
overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);
registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener); ←
//Ensure that a new exporter instance is returned every time export
return new Exporter<T>() {
    public Invoker<T> getInvoker() { ④ 监听服务接口下 configurators
        return exporter.getInvoker();  节点，用于处理动态配置
    }
};

public void unexport() {
    try {
        exporter.unexport(); ← ⑤ Invoker 销毁时注销端口和 map 中服务实例等资源
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
    try {
        registry.unregister(registeredProviderUrl); ← ⑥ 移除已注册的元数据
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
    try {
        overrideListeners.remove(overrideSubscribeUrl); ←
        registry.unsubscribe(overrideSubscribeUrl, ⑦ 去掉订阅配置监
        overrideSubscribeListener); 听器
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
}
};

}
}

```

在①中进行服务暴露前，框架会做拦截器初始化，Dubbo在加载 protocol 扩展点时会自动注入 Protocol Listenerwrapper 和 ProtocolFilterWrappero真实暴露时会按照图5-5所示的流程执行。

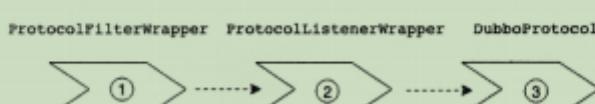


图 5-5 拦截器初始化

在ProtocolListenerWrapper实现中，在对服务提供者进行暴露时回调对应的监听器方法。

ProtocolFilterWrapper会调用下一级 ListenerExporterWrapper#export方法，在该方法内部会触发buildInvokerChain进行拦截器构造，

①：在触发Dubbo协议暴露前先对服务Invoker做了一层拦截器构建，在加载所有拦截器时会过滤只对provider生效的数据。

②：首先获取真实服务ref对应的Invoker并挂载到整个拦截器链尾部，然后逐层包裹其他拦截器，这样保证了真实服务调用是最后触发的。

③：逐层转发拦截器服务调用，是否调用下一个拦截器由具体拦截器实现。

在构造调用拦截器之后会调用Dubbo协议进行服务暴露

①根据服务分组、版本、接口和端口构造key

②把exporter存储到单例DubboProtocol中

①和②：中主要根据服务分组、版本、服务接口和暴露端口作为key用于关联具体服务Invoker。

③：对服务暴露做校验判断，因为同一个协议暴露有很多接口，只有初次暴露的接口才需要打开端口监听，

④触发HeaderExchanger中的绑定方法，最后会调用底层NettyServer进行处理。在初始化Server过程中会初始化很多Handler用于支持一些特性，比如心跳、业务线程池处理编解码的Handler和面向方法调用的Handler

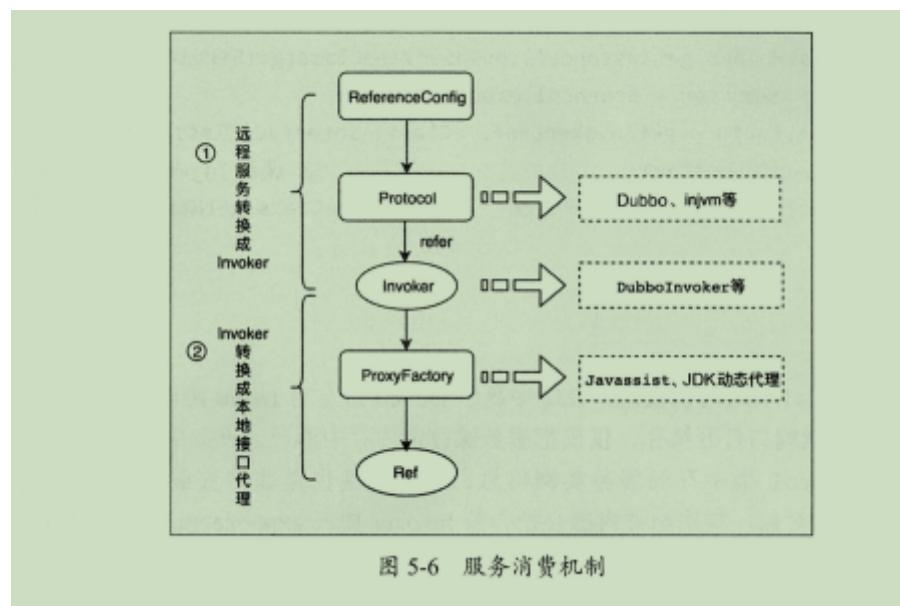
本地服务的暴露机制

①Dubbo指定用injvm协议暴露服务，这个协议比较特殊，不会做端口打开操作，仅仅把服务保存在内存中而已。

②会提取URL中的协议，在InjvmProtocol类中存储服务实例信息，它的实现也是非常直截了当的，直接返回InjvmExporter实例对象，构造函数内部会把当前Invoker加入exporterMap

服务消费的实现原理

单注册中心消费原理



在整体上看，Dubbo框架做服务消费也分为两大部分，第一步通过持有远程服务实例生成Invoker，这个Invoker在客户端是核心的远程代理对象。第二步会把Invoker通过动态代理转换成实现用户接口的动态代理引用。这里的Invoker承载了网络连接、服务调用和重试等功能，在客户端，它可能是一个远程的实现，也可能是一个集群实现。

框架真正进行服务引用的入口点在ReferenceBean#getObject，不管是XML还是注解，都会转换成ReferenceBean，它继承自ReferenceConfig，在服务消费前也会按照5.2.1节的覆盖策略生效，主要处理思路就是遍历服务的所有方法，如果没有值则会尝试从-D选项中读取，如果还没有则自动从配置文件dubbo.properties中读取。

Dubbo支持多注册中心同时消费，如果配置了服务同时注册多个注册中心，则会在ReferenceConfig#createProxy中合并成一个Invoker。

在createProxy实现中完成了远程代理对象的创建及代理对象的转换等工作

①会优先判断是否在同一个JVM中包含要消费的服务，默认场景下，Dubbo会通过②找出内存中injvm协议的服务，其实injvm协议是比较容易理解的，前面提到服务实例都放到内存map中，消费也是直接获取实例调用而已。

③：主要在注册中心中追加消费者元数据信息，应用启动时订阅注册中心、服务提供者参数等合并时会用到这部分信息。

④：处理只有一个注册中心的场景，这种场景在客户端中是最常见的，客户端启动拉取服务元数据，订阅provider、路由和配置变更。

⑤和⑥：分别处理多注册中心的场景

当经过注册中心消费时，主要通过RegistryProtocol#refer触发数据拉取、订阅和服务Invoker转换等操作，其中最核心的数据结构是RegistryDirectory

主要完成了注册中心实例的创建，元数据注册到注册中心及订阅的功能。

①会根据用户指定的注册中心进行协议替换，具体注册中心协议会在启动时用registry存储对应值。

②会创建注册中心实例，这里的URL其实是注册中心地址，真实消费方的元数据信息是放在refer属性中存储的。

③主要提取消费方refer中保存的元数据信息，如果包含多个分组值则会把调用结果值做合并处理。

④触发真正的服务订阅和Invoker转换。

⑤RegistryDirectory实现了NotifyListener接口，服务变更会触发这个类回调notify方法，用于重新引用服务。

⑥负责把消费方元数据信息注册到注册中心，比如消费方应用名、IP和端口号等。

⑦处理provider、路由和动态配置订阅。

⑧除了通过Cluster将多个服务合并，同时默认也会启用FailoverCluster策略进行服务调用重试。

当在⑦中第一次发起订阅时会进行一次数据拉取操作，同时触发RegistryDirectory#notify方法，这里的通知数据是某一个类别的全量数据，比如providers和routers类别数据。当通知providers数据时，在RegistryDirectory#toInvokers方法内完成Invoker转换。

Dubbo框架允许在消费方配置只消费指定协议的服务，

具体协议过滤在①中进行处理，支持消费多个协议，允许消费多个协议时，在配置Protocol值时用逗号分隔即可。

②消费信息是客户端处理的，需要合并服务端相关信息，比如远程IP和端口等信息，通过注册中心获取这些信息，解耦了消费方强绑定配置。

- ③消除重复推送的服务列表，防止重复引用。
- ④使用具体的协议发起远程连接等操作。在真实远程连接建立后也会发起拦截器构建操作，处理逻辑在ProtocolFilterWrapper#refer中触发链式构造

具体Invoker创建是在DubboProtocol#refer中实现的，Dubbo协议在返回DubboInvoker对象之前会先初始化客户端连接对象。Dubbo支持客户端是否立即和远程服务建立TCP连接是由参数是否配置了lazy属性决定的，默认会全部连接。DubboProtocol#refer内部会调用DubboProtocol#initClient负责建立客户端连接和初始化Handler。

- ①：支持lazy延迟连接，在真实发生RPC调用时创建。
- ②：立即发起远程TCP连接，具体使用底层传输也是根据配置transporter决定的，默认是Netty传输。在②中会触发HeaderExchanger#connect调用，用于支持心跳和在业务线程中编解码Handler，最终会调用Transporters#connect生成Netty客户端处理。

多注册中心消费原理

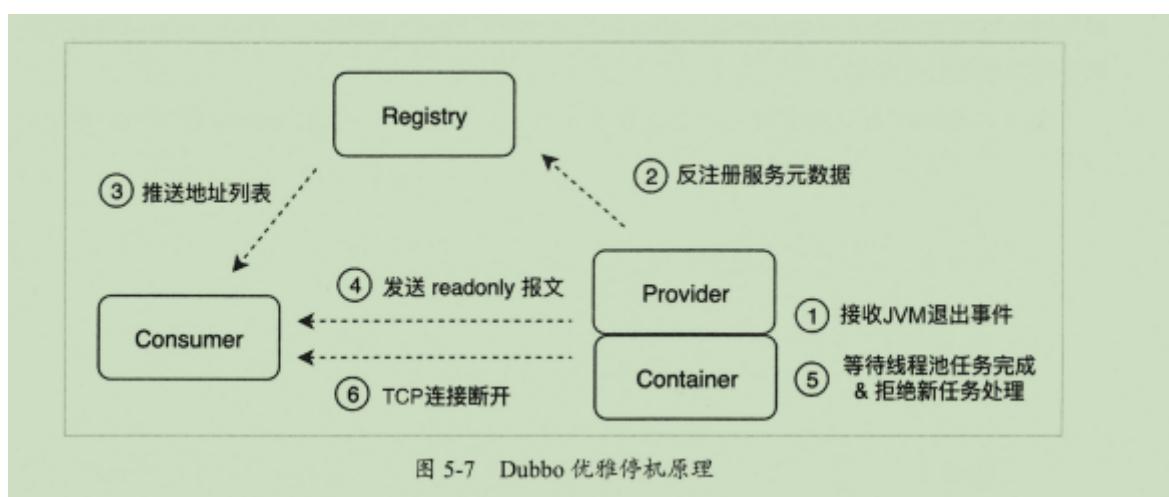
多注册中心消费原理比较简单，每个单独注册中心抽象成一个单独的Invoker，多个注册中心实例最终通过StaticDirectory保存所有的Invoker，最终通过Cluster合并成一个Invoker。我们可以回到5.3.1节代码清单5-18代码逻辑中，⑤是逐个获取注册中心的服务，并添加到invokers列表，这里多个注册中心逐一获取服务。⑥主要将集群服务合并成一个Invoker，这里也不难理解，第一层包含的服务Invoker是注册中心实例，对应注册中心实例的Invoker对象内部持有真实的提供者对象列表。这里还有一个特殊点，在多注册中心场景下，默认使用的集群策略是available。

直连服务消费原理

Dubbo可以绕过注册中心直接向指定服务(直接指定目标IP和端口)发起RPC调用，使用直连模式可以方便在某些场景下使用，比如压测指定机器等。Dubbo框架也支持同时指定直连多台机器进行服务调用。

- ①允许用分号指定多个直连机器地址，多个直连机器调用会使用负载均衡，更多场景是单个直连，但是不建议在生产环境中使用直连模式，因为上游服务发布会影响服务调用方。
- ②允许配置注册中心地址，这样可以通过注册中心发现服务消费。
- ③指定服务调用协议、IP和端口，注意这里的URL没有添加refer和注册中心协议，默认是Dubbo会直接触发DubboProtocol进行远程消费，不会经过RegistryProtocol去做服务发现。

优雅停机原理解析



Dubbo中实现的优雅停机机制主要包含6个步骤：

- (1) 收到kill 9进程退出信号，Spring容器会触发容器销毁事件
 (2) provider端会取消注册服务元数据信息。

3 consumer端会收到最新地址列表（不包含准备停机的地址）。

4 Dubbo协议会发送readonly事件报文通知consumer服务不可用。

5 服务端等待已经执行的任务结束并拒绝新任务执行。

6.断开连接

为什么还要再发送readonly报文呢？

这里主要考虑到注册中心推送服务有网络延迟，以及客户端计算服务列表可能占用一些时间。Dubbo协议发送readonly时间报文时，consumer端会设置响应的provider为不可用状态，下次负载均衡就不会调用下线的机器。

Dubbo远程调用

Dubbo调用介绍

简单的RPC调用，则需要把服务调用信息传递到服务端，每次服务调用的一些公用的信息包括服务调用接口、方法名、方法参数类型和方法参数值等，在传递方法参数值时需要先序列化对象并经过网络传输到服务端，在服务端需要按照客户端序列化顺序再做一次反序列化来读取信息，然后拼装成请求对象进行服务反射调用，最终将调用结果再传给客户端。

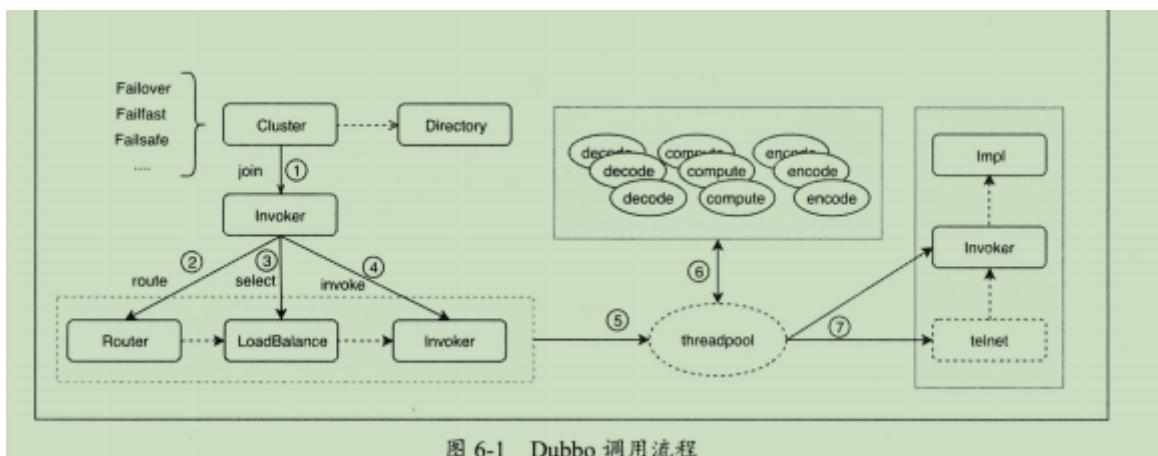


图 6-1 Dubbo 调用流程

首先在客户端启动时会从注册中心拉取和订阅对应的服务列表，Cluster会把拉取的服务列表聚合成一个Invoker,每次RPC调用前会通过Directory#list获取providers地址（已经生成好的Invoker列表），获取这些服务列表给后续路由和负载均衡使用。对应图6.1,在①中主要是将多个服务提供者做聚合。在框架内部另外一个实现Directory接口是RegistryDirectory类，它和接口名是一对一的关系（每一个接口都有一个RegistryDirectory实例），主要负责拉取和订阅服务提供者、动态配置和路由项。

先会触发路由操作，然后将路由结果得到的服务列表作为负载均衡参数，经过负载均衡后会选出一台机器进行RPC调用，这3个步骤依次对应于②、③和④。客户端经过路由和负载均衡后，会将请求交给底层I/O线程池（比如Netty 处理，I/O线程池主要处理读写、序列化和反序列化等逻辑，因此这里一定不能阻塞操作，Dubbo也提供参数控制 decode.in.io 参数，在处理反序列化对象时会在业务线程池中处理。在⑤中包含两种类似的线程池，一种是I/O线程池Netty ,另一种是Dubbo业务线程池（承载业务方法调用）。

目前Dubbo将服务调用和Telnet调用做了端口复用，在编解码层面也做了适配。在Telnet调用时，会新建一个TCP连接，传递接口、方法和JSON格式的参数进行服务调用，在编解码层面简单读取流中的字符串（因为不是Dubbo标准头报文），最终交给Telnet对应的Handler去解析方法调用。如果是非Telnet调用，则服务提供方会根据传递过来的接口、分组和版本信息查找Invoker对应的实例进行反射调用。在⑦中进行了端口复用，如果是Telnet调用，则先找到对应的Invoker进行方法调用。Telnet和正常RPC调用不一样的地方是序列化和反序列化使用的不是Hessian方式，而是直接使用fastjson进行处理。

Dubbo协议详解

偏移字节		0								1								2								3															
偏移比特位		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31								
0	0	魔法数高位								魔法数低位								请求 响应	需要 往返	事件	序列化ID				状态																
4	32	RPC 请求ID																																							
8	64	消息体数据长度																																							
12	96	dubbo version , service name , service version , method name , parameter types , arguments , attachments																																							
16	128	...																																							

图 6-2 Dubbo 协议

发现一次RPC调用包括协议头和协议体两部分。

16字节长的报文头部主要携带了魔法数(0xdabb),以及当前请求报文是否是Request、Response 心跳和事件的信息，请求时也会携带当前报文体内序列化协议编号。除此之外，报文头部还携带了请求状态，以及请求唯一标识和报文体长度。

表 6-1 Dubbo 协议字段解析

偏移比特位	字段描述	作用
0~7	魔数高位	存储的是魔法数高位（0xda00）
8~15	魔数低位	存储的是魔法数高位（0xbb）
16	数据包类型	是否为双向的 RPC 调用（比如方法调用有返回值），0 为 Response，1 为 Request
17	调用方式	仅在第 16 位被设为 1 的情况下有效 0 为单向调用，1 为双向调用 比如在优雅停机时服务端发送 readonly 不需要双向调用，这里标志位就不会设定

续表

偏移比特位	字段描述	作用
18	事件标识	0 为当前数据包是请求或响应包 1 为当前数据包是心跳包，比如框架为了保活 TCP 连接，每次客户端和服务端互相发送心跳包时这个标志位被设定 设置了心跳报文不会透传到业务方法调用，仅用于框架内部保活机制
19~23	序列化器编号	2 为 Hessian2Serialization 3 为 JavaSerialization 4 为 CompactedJavaSerialization 6 为 FastJsonSerialization 7 为 NativeJavaSerialization 8 为 KryoSerialization 9 为 FstSerialization
24~31	状态	20 为 OK 30 为 CLIENT_TIMEOUT 31 为 SERVER_TIMEOUT 40 为 BAD_REQUEST 50 为 BAD_RESPONSE
32~95	请求编号	这 8 个字节存储 RPC 请求的唯一 id，用来将请求和响应做关联
96~127	消息体长度	占用的 4 个字节存储消息体长度。在一次 RPC 请求过程中，消息体中依次会存储 7 部分内容

在协议报文头部的 status 中，完整状态响应码和作用如表 6-2 所示。

表 6-2 完整状态响应码和作用

状态值	状态符号	作用
20	OK	正确返回
30	CLIENT_TIMEOUT	客户端超时
31	SERVER_TIMEOUT	服务端超时
40	BAD_REQUEST	请求报文格式错误
50	BAD_RESPONSE	响应报文格式错误
60	SERVICE_NOT_FOUND	未找到匹配的服务
70	SERVICE_ERROR	服务调用错误

第 6 章 Dubbo 远程调用 |

续表

状态值	状态符号	作用
80	SERVER_ERROR	服务端内部错误
90	CLIENT_ERROR	客户端错误
100	SERVER_THREADPOOL_EXHAUSTED_ERROR	服务端线程池满拒绝执行

主要根据以下标记判断返回值，如表 6-3 所示。

表 6-3 Dubbo 响应标记

状态值	状态符号	作用
5	RESPONSE_NULL_VALUE_WITH_ATTACHMENTS	响应空值包含隐藏参数
4	RESPONSE_VALUE_WITH_ATTACHMENTS	响应结果包含隐藏参数
3	RESPONSE_WITH_EXCEPTION_WITH_ATTACHMENTS	异常返回包含隐藏参数
2	RESPONSE_NULL_VALUE	响应空值
1	RESPONSE_VALUE	响应结果
0	RESPONSE_WITH_EXCEPTION	异常返回

我们知道在网络通信中（基于TCP 需要解决网络粘包/解包的问题，一些常用解决办法比如用回车、换行、固定长度和特殊分隔符等进行处理，通过对前面协议的理解，我们很容易发现Dubbo其实就是用特殊符号exdabb魔法数来分割处理粘包问题的。

在实际使用场景中，客户端会使用多线程并发调用服务，Dubbo 是如何做到正确响应调用线程的呢？关键点在于协议头全局请求 id 标识，我们先来看一下原理图，如图 6-3 所示。

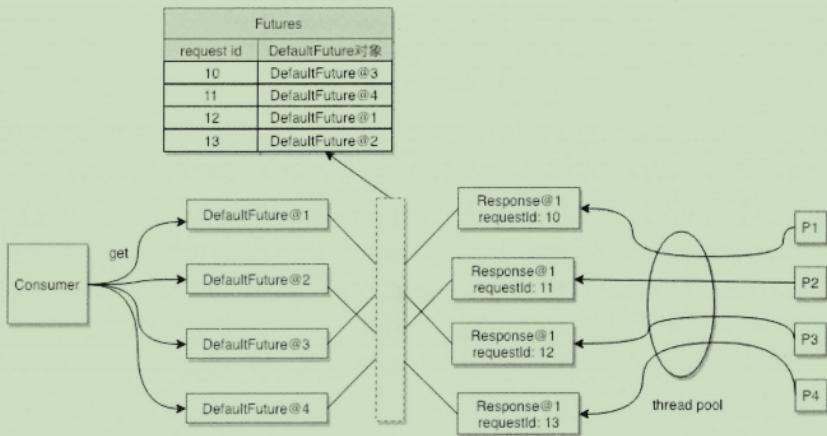


图 6-3 Dubbo 请求响应

当客户端多个线程并发请求时，框架内部会调用 DefaultFuture 对象的 get 方法进行等待。在请求发起时，框架内部会创建 Request 对象，这个时候会被分配一个唯一 id，DefaultFuture 可以从 Request 对象中获取 id，并将关联关系存储到静态 HashMap 中，就是图 6-3 中的 Futures 集合。当客户端收到响应时，会根据 Response 对象中的 id，从 Futures 集合中查找对应的 DefaultFuture 对象，最终会唤醒对应的线程并通知结果。客户端也会启动一个定时扫描线程去探测超时没有返回的请求。

编解码器原理

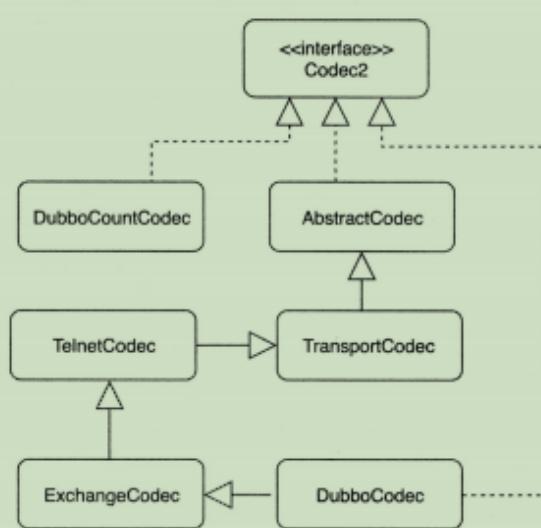


图 6-4 Dubbo 编解码关系

AbstractCodec 主要提供基础能力，比如校验报文长度和查找具体编解码器等。

TransportCodec 主要抽象编解码实现，自动帮我们去调用序列化、反序列化实现和自动 cleanup 流。

我们通过Dubbo编解码继承结构可以清晰看到，DubboCodec继承自ExchangeCodec,它又再次继承了TelnetCodec实现。我们前面说过Telnet实现复用了Dubbo协议端口，其实就是在这一层编解码做了通用处理。因为流中可能包含多个RPC请求，Dubbo框架尝试一次性读取更多完整报文编解码生成对象，也就是图中的DubboCountCodec,它的实现思想比较简单，依次调用DubboCodec去解码，如果能解码成完整报文，则加入消息列表，然后触发下一个Handler方法调用。

Dubbo协议编码器

Dubbo中的编码器主要将Java对象编码成字节流返回给客户端，主要做两部分事情，构造报文头部，然后对消息体进行序列化处理。所有编解码层实现都应该继承自ExchangeCodec,Dubbo协议编码器也不例外。当Dubbo协议编码请求对象时，会调用ExchangeCodec#encode方法。

主要职责是将Dubbo请求对象编码成字节流(包括协议报文头部)。

- ①主要提取URL中配置的序列化协议或默认协议。
- ②会创建16字节的报文头部。
- ③首先会将魔法数写入头部并占用2个字节。
- ④主要设置请求标识和消息体中使用的序列化协议。
- ⑤会复用同一个字节，标记这个请求需要服务端返回。
- ⑥主要承载请求的唯一标识，这个标识用于匹配响应的数据。
- ⑦会在buffer中预留16字节存储头部，
- ⑧序列化请求部分，比如方法名等信息。
- ⑨会检查编码后的报文是否超过大小限制(默认是8MB)。
- ⑩将消息体长度写入头部偏移量(第12个字节)，长度占用4个字节。
- ⑪将buffer定位到报文头部开始，
- ⑫将构造好的头部写入buffer。
- ⑬再将buffer写入索引执行消息体结尾的下一个位置。

在⑧中会调用encodeRequestData方法对RpcInvocation调用进行编码，这部分主要就是对接口、方法、方法参数类型、方法参数等进行编码，在DubboCodec#encodeRequestData中重写了这个方法实现。

主要职责是将Dubbo方法调用参数和值编码成字节流。在编码消息体的时候，

- ①主要先写入框架的版本，这里主要用于支持服务端版本隔离和服务端隐式参数透传给客户端的特性。
- ②向服务端写入调用的接口。
- ③指定接口的版本，默认版本为0.0.0,Dubbo允许同一个接口有多个实现，可以指定版本或分组来区分。
- ④指定远程调用的接口方法。
- ⑤将方法参数类型以Java类型方式传递给服务端。
- ⑥循环对参数值进行序列化。
- ⑦写入隐式参数HashMap,这里可能包含timeout和group等动态参数。

我们继续分析**编码响应对象**，理解了编码请求对象后，比较好理解响应，响应实现在 ExchangeCodec#encodeResponse中，

的主要职责是将Dubbo响应对象编码成字节流(包括协议报文头部)。在编码响应中，

- ①获取用户指定或默认的序列化协议，
- ②构造报文头部(16字节)。
- ③同样将魔法数填入报文头部前2个字节。
- ④会将服务端配置的序列化协议写入头部。
- ⑤报文头部中status会保存服务端调用状态码。
- ⑥会将请求唯一id设置回响应头中。
- ⑦空出16字节头部用于存储响应体报文。
- ⑧会对服务端调用结果进行编码。
- ⑨主要对响应报文大小做检查，默认校验是否超过8MB大小。
- ⑩将消息体长度写入头部偏移量(第12个字节)，长度占用4个字节。
- ⑪将buffer定位到报文头部开始，
- ⑫将构造好的头部写入buffer。
- ⑬再将buffer写入索引执行消息体结尾的下一个位置。
- ⑭主要处理编码报错复位buffer,否则导致缓冲区中数据错乱。

15会将异常响应返回到客户端，防止客户端只有等到超时才能感知服务调用返回。

⑯和⑰中主要对报错进行了细分，处理服务端报文超过限制和具体报错原因。为了防止报错对象无法在客户端反序列化，在服务端会将异常信息转成字符串处理

编码响应消息提的部分，在⑧中处理响应，具体实现在DubboCodec#encodeResponseData中

主要职责是将Dubbo方法调用状态和返回值编码成字节流。编码响应体也是比较简单的，

- ①判断客户端的版本是否支持隐式参数从服务端传递到客户端。
- ②和③处理正常服务调用，并且返回值为null的场景，用一个字节标记。
- ④处理方法正常调用并且有返回值，先写一个字节标记并序列化结果。
- ⑤处理方法调用发生异常，写一个字节标记并序列化异常对象。
- ⑥处理客户端支持隐式参数回传，记录服务端Dubbo版本，并返回服务端隐式参数。

Dubbo协议解码器

解码工作分为2部分，第1部分解码报文的头部(16字节)，第2部分解码报文体内容，以及如何把报文体转换成RpcInvocation。当服务端读取流进行解码时，会触发

ExchangeCodec#decode方法，Dubbo协议解码继承了这个类实现，但是在解析消息体时，Dubbo协议重写了decodeBody方法。

解码头部的部分

代码清单 6-5 解码报文头

```

@Override
public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {
    int readable = buffer.readableBytes();
    byte[] header = new byte[Math.min(readable, HEADER_LENGTH)];
    buffer.readBytes(header);           ① 最多读取 16 个字节，并分配存储空间
    return decode(channel, buffer, readable, header);
}

@Override
protected Object decode(Channel channel, ChannelBuffer buffer, int readable, byte[] header)
throws IOException {
    if (readable > 0 && header[0] != MAGIC_HIGH && header[1] != 0xdabb) { ② 处理流起始处不是 Dubbo 魔法数
        if (readable > 1 && header[1] != MAGIC_LOW) {
            int length = header.length;          ③ 流中还有数据可以读取
            if (header.length < readable) {       ④ 为 header 重新分配空间，用
                header = Bytes.copyOf(header, readable);  来存储流中所有可读字节
                buffer.readBytes(header, length, readable - length);  ⑤ 将流中剩余字节读取到 header 中
            }
            for (int i = 1; i < header.length - 1; i++) {
                if (header[i] == MAGIC_HIGH && header[i + 1] == MAGIC_LOW) {
                    buffer.readerIndex(buffer.readerIndex() - header.length + i);  ⑥ 将 buffer 读索引指向回
                    header = Bytes.copyOf(header, i);  Dubbo 报文开头处(0xdabb)
                    break;  ⑦ 将流起始处至下一个 Dubbo
                }  报文之间的数据放到 header 中
            }
        }
        return super.decode(channel, buffer, readable, header);  ⑧ 主要用于解析 header 数据，比如用于 Telnet
    }
    if (readable < HEADER_LENGTH) {  ⑨ 如果读取数据长度小于 16 个字节，则
        return DecodeResult.NEED_MORE_INPUT;  期待更多数据
    }

    int len = Bytes.bytesToInt(header, 12);  ⑩ 提取头部存储的报文长度，并校验长度是否超过限制
}

```

```

checkPayload(channel, len);

int tt = len + HEADER_LENGTH;
if (readable < tt) {  ⑪ 校验是否可以读取完整 Dubbo 报文，否则期待更多数据
    return DecodeResult.NEED_MORE_INPUT;
}

// limit input stream.
ChannelBufferInputStream is = new ChannelBufferInputStream(buffer, len);

try {
    return decodeBody(channel, is, header);  ⑫ 解码消息体，is 流是完整 的 RPC 调用报文
} finally {
    if (is.available() > 0) {  ⑬ 如果解码过程有问题，则跳过这次 RPC 调用报文
        try {
            if (logger.isWarnEnabled()) {
                logger.warn("Skip input stream " + is.available());
            }
            StreamUtils.skipUnusedStream(is);
        } catch (IOException e) {
            logger.warn(e.getMessage(), e);
        }
    }
}
}

```

整体实现解码过程中要解决粘包和半包问题。

①最多读取Dubbo报文头部(16字节)，如果流中不足16字节，则会把流中数据读取完毕。在decode方法中会先判断流当前位置是不是Dubbo报文开始处，在流中判断报文分割点是通过②判断的(Oxdabb魔法数)。如果当前流中没有遇到完整Dubbo报文(在③中会判断流可读字节数)，在④中会为剩余可读流分配存储空间，在⑤中会将流中数据全部读取并追加在header数组中。当流被读取完后，会查找流中第一个Dubbo报文开始处的索引，在⑥中会将buffer索引指向流中第一个Dubbo报文开始处(0xdabb)0在⑦

中主要将流中从起始位置(初始buffer的readerindex)到第一个Dubbo报文开始处的数据保存在header中，用于⑧解码header数据，目前常用的场景有Telnet调用等。

在正常场景中解析时，在⑨中首先判断当次读取的字节是否多于16字节，否则等待更多网络数据到来。在⑩中会判断Dubbo报文头部包含的消息体长度，然后校验消息体长度是否超过限制（默认为8MB）。在⑪中会校验这次解码能否处理整个报文。在⑫中处理消息体解码，这个是强协议相关的，因此Dubbo协议重写了这部分实现，我们先看一下在DubboCodec中是如何处理的。

```
代码清单 6-6 解码请求报文

@Override
protected Object decodeBody(Channel channel, InputStream is, byte[] header) throws
IOException {
    byte flag = header[2], proto = (byte) (flag & SERIALIZATION_MASK);
    // get request id.
    long id = Bytes.bytes2long(header, 4);
    if ((flag & FLAG_REQUEST) == 0) {
        // decode response.
        // ...
    } else {
        Request req = new Request(id); ←① 请求标志位被设置，创建 Request 对象
        req.setVersion(Version.getProtocolVersion());
        req.setTwoWay((flag & FLAG_TWOWAY) != 0);
        if ((flag & FLAG_EVENT) != 0) {
            req.setEvent(Request.HEARTBEAT_EVENT);
        }
        try {
            Object data;
            ObjectInput in = CodecSupport.deserialize(channel.getUrl(), is, proto);
            if (req.isHeartbeat()) {
                data = decodeHeartbeatData(channel, in);
            } else if (req.isEvent()) {
                data = decodeEventData(channel, in);
            } else {
                DecodeableRpcInvocation inv;
                if (channel.getUrl().getParameter(
                    Constants.DECODE_IN_IO_THREAD_KEY,
                    Constants.DEFAULT_DECODE_IN_IO_THREAD)) {
                    inv = new DecodeableRpcInvocation(channel, req, is, proto); ←② 在 I/O 线程中直接解码
                    inv.decode();
                } else {

```

```
        inv = new DecodeableRpcInvocation(channel, req,           ←③
                                         new UnsafeByteArrayInputStream(readMessageData(is)), proto);
    }
    data = inv;
}
req.setData(data); ←④ 将 RpcInvocation 作为 Request 的数据域
} catch (Throwable t) {
    if (log.isWarnEnabled()) {
        log.warn("Decode request failed: " + t.getMessage(), t);
    }
    req.setBroken(true); ←⑤ 解码失败，先做标记并存储异常
    req.setData(t);
}
return req;
}
}
```

站在解码器的角度，解码请求一定是通过标志判断类别的，否则不知道是请求还是响应，Dubbo报文16字节头部长度包含了FLAG_REQUEST标志位。

- ①：根据这个标志位创建请求对象，
- ②：在I/O线程中直接解码(比如在Netty的I/O线程中)，然后简单调用decode解码，解码逻辑在后面会详细探讨。
- ③：实际上不做解码，延迟到业务线程池中解码。
- ④：将解码消息体作为RpcInvocation放到请求数据域中。如果解码失败了，则会通过⑤标记，并把异常原因记录下来。

这里没有提到的是心跳和事件的解码，这两种解码非常简单，心跳报文是没有消息体的，事件有消息体，在使用Hessian2协议的情况下默认会传递字符R,当优雅停机时会通过发送readonly事件来通知客户端服务端不可用。

如何把消息体转换成RpcInvocation对象

在解码请求时，是严格按照客户端写数据顺序来处理的。

- ①会读取远端传递的框架版本，
- ②会读取调用接口全称，
- ③会读取调用的服务版本，用来实现分组和版本隔离。
- ④会读取调用方法的名称，
- ⑤读取方法参数类型，通过类型能够解析出实际参数个数。
- ⑥中会对方法参数值依次读取，这里具体解析参数值是和序列化协议相关的。
- ⑦读取隐式参数，比如同机房优先调用会读取其中的tag值。
- ⑧为了支持异步参数回调，因为参数是回调客户端方法，所以需要在服务端创建客户端连接代理

解码响应，解码响应会调用DubboCodec#decodeBody方法

在读取服务端响应报文时，先读取状态标志，然后根据状态标志判断后续的数据内容。在编码响应对象中，响应结果首先会写一个字节标记位。

- ①处理标记位代表返回值为Null的场景。
- ②代表正常返回，首先判断请求方法的返回值类型，返回值类型方便底层反序列化正确读取，将读取的值存在result字段中。
- ④处理服务端返回异常对象的场景，同时会将结果保存在exception字段中。
- ⑤处理返回值为Null,并且支持服务端隐式参数透传给客户端，在客户端会继续读取保存在HashMap中的隐式参数值。当然，还有其他场景，比如RPC调用有返回值，RPC调用抛出异常时需要隐式参数给客户端的场景，可以举一反三，不再重复说明。

Telnet调用原理

Telnet指令解析原理

Telnet指令解析被设置成了扩展点TelnetHandler,每个Telnet指令都会实现这个扩展点。我们首先查看这个扩展点的定义

```
@SPI
public interface TelnetHandler {
    String telnet(Channel channel, String message) throws RemotingException;
}
```

完成Telnet指令转发的核心实现类是TelnetHandlerAdapter,它的实现非常简单，首先将用户输入的指令识别成commandC比如invoke、ls和status),然后将剩余的内容解析成message,message会交给命令实现者去处理。实现代码类在TelnetHandlerAdapter#telnet中

- ①提取Telnet一行消息的首个字符串作为命令，如果命令行有空格，则将后面的内容作为字符串，
- ②提取并存储到message中。
- ③判断并加载是否有对应的扩展点，如果存在对应的Telnet扩展点，则会通过加载具体的扩展点并调用其telnet方法，最后连同返回结果并追加消息结束符(在⑤中处理)返回给调用方。

对常用命令Invoke调用进行探讨，在InvokeTelnetHandler中本地实现Telnet类调用

当本地没有客户端，想测试服务端提供的方法时，可以使用Telnet登录到远程服务器(Telnet IP port),根据invoke指令执行方法调用来获得结果。当用户输入invoke指令时，会被转发到代码清单6-11对应的Handler。

- ①提取方法调用信息(去除参数信息)，
- ②会提取调用括号内的信息作为参数值。
- ③提取方法调用的接口信息，
- ④提取接口调用的方法名称。
- ⑤会将传递的JSON参数值转换成fastjson对象，
- ⑥根据接口名称、方法和参数值查找对应的方法和Invoker对象。在真正方法调用前，需要通过把fastjson对象转换成Java对象，触发方法调用并返回结果值

Telnet实现健康监测

Telnet提供了健康检查的命令，可以在Telnet连接成功后执行status -1查看线程池、内存和注册中心等状态信息。为了完成线程池监控、内存和注册中心监控等诉求，Telnet提供了新的扩展点Statuscheck。

当执行status命令时会触发StatusTelnetHandler#telnet调用，这个方法的实现也比较简单，它会加载所有实现Statuschecker扩展点的类，然后调用所有扩展点的check方法。

```
@SPI  
public interface Statuschecker {  
    status check();  
}
```

表 6-4 健康监测对应的实现和作用

健康检查实现类	作用
DataSourceStatusChecker	数据库状态检查
LoadStatusChecker	系统平均负载检查
MemoryStatusChecker	JVM 内存检查
RegistryStatusChecker	注册中心状态检查
ServerStatusChecker	Dubbo 服务暴露检查
SpringStatusChecker	Spring 状态检查
ThreadPoolStatusChecker	Dubbo 线程池检查

ChannelHandler

Dubbo内部使用的ChannelHandler组件的原理,Dubbo框架内部使用大量Handler组成类似链表，依次处理具体逻辑，比如编解码、心跳时间戳和方法调用Handler等。因为Netty每次创建Handler都会经过ChannelPipeline,大量的事件经过很多Pipeline会有较多的开销，因此Dubbo会将多个Handler聚合为一个Handler。

核心Handler和线程模型

Dubbo中Handler (ChannelHandler)的5种状态

表 6-5 生命周期状态

状态	描述
connected	Channel 已经被创建
disconnected	Channel 已经被断开
sent	消息被发送
received	消息被接收
caught	捕获到异常

表 6-6 Dubbo 常用 Handler

Handler	作用
ExchangeHandlerAdapter	用于查找服务方法并调用
HeaderExchangeHandler	封装处理 Request/Response 和 Telnet 调用能力
DecodeHandler	支持在 Dubbo 线程池中做解码
ChannelHandlerDispatcher	封装多 Handler 广播调用
AllChannelHandler	支持 Dubbo 线程池调用业务方法
HeartbeatHandler	支持心跳处理
MultiMessageHandler	支持流中多消息报文批处理
ConnectionOrderedChannelHandler	单独线程池处理 TCP 的连接和断开
MessageOnlyChannelHandler	仅在线程池处理接收报文，其他事件在 I/O 线程处理
WrappedChannelHandler	基于内存 key-value 存储封装和共享线程池能力，比如记录线程池等
NettyServerHandler	封装 Netty 服务端事件，处理连接、断开、读取、写入和异常等
NettyClientHandler	封装 Netty 客户端事件，处理连接、断开、读取、写入和异常等

图6-5展示了同时具有入站和出站ChannelHandler的布局，如果有一个入站事件被触发，比如连接或数据读取，那么它会从ChannelPipeline头部开始一直传播到Channelpipeline的尾端。出站的I/O事件将从ChannelPipeline最右边开始，然后向左传播。当然，在ChannelPipeline传播事件时，它会测试入站是否实现了 ChannelInboundHandler接口，如果没有实现则会自动跳过，出站时会监测是否实现 ChannelOutboundHandler,如果没有实现，那么也会自动跳过。在Dubbo框架中实现的这两个接口类主要是NettyServerHandler和NettyClientHandler。Dubbo通过装饰者模式层包装Handler,从而不需要将每个Handler都追加到Pipeline中。在NettyServer 和 NettyClient 中最多有 3 个 Handler,分别是编码、解码和 NettyServerHandler或 NettyClientHandler。

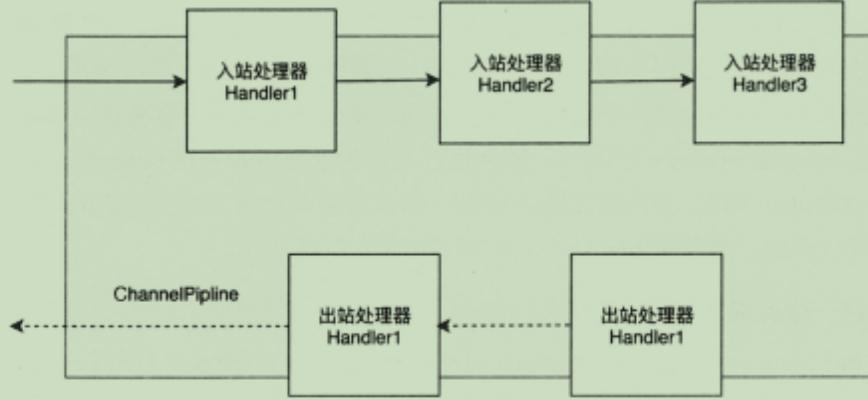


图 6-5 ChannelHandler

代码清单 6-13 服务方 ExchangeHandlerAdapter 内部类实现

```

private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {

    @Override
    public Object reply(ExchangeChannel channel, Object message) throws
    RemotingException {
        if (message instanceof Invocation) {
            Invocation inv = (Invocation) message;
            Invoker<?> invoker = getInvoker(channel, inv); <-- ① 查找 Invocation
            // ...
            RpcContext.getContext().setRemoteAddress(channel.getRemoteAddress());
            return invoker.invoke(inv); <-- ② 调用业务方具体方法
        }
        throw new RemotingException(channel, "Unsupported request: "
            + (message == null ? null : (message.getClass().getName() + ":" +
            message))
            + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider:
            " + channel.getLocalAddress());
    }
}

```

代码清单6-13中给出的Handler实现是触发业务方法调用的关键，在服务暴露时服务端已经按照特定规则(端口、接口名、接口版本和接口分组)把实例Invoker存储到HashMap中，客户端调用过来时必须携带相同信息构造的key,找到对应Exporter然后调用。在①中查找当前已经暴露的服务，后面会继续分析这个方法实现。在②中主要包含实例的Filter和真实业务对象，当触发invoker#invoke方法时，就会执行具体的业务逻辑。在DubboProtocol中，我们继续跟踪getInvoker调用,会发现在服务端唯一标识的服务是由4部分组成的：端口、接口名、接口版本和接口分组。

服务端Invoker查找

- ①主要获取协议暴露的端口，比如Dubbo协议默认的端口为20880。
- ②获取客户端传递过来的接口名称(大部分场景都是接口名)。
- ③主要根据服务端口、接口名、接口分组和接口版本构造唯一的key。
- ④：简单从HashMap中取出对应的Exporter并调用Invoker属性值。分析到这里，读者应该能理解RPC调用在服务端处理的逻辑了。

我们先看一下 Dubbo 中是如何实现线程派发的，如图 6-6 所示。

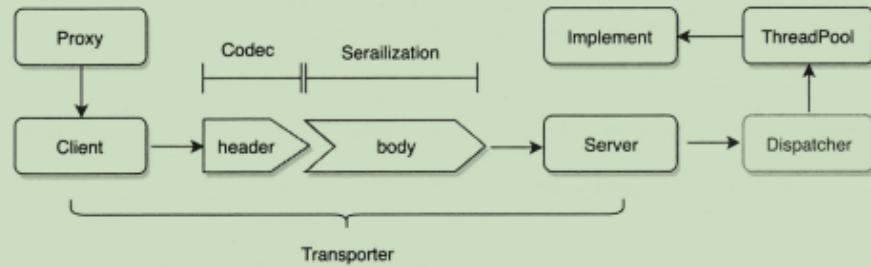


图 6-6 Dubbo 线程模型

在图 6-6 中，Dispatcher 就是线程池派发器。这里需要注意的是，Dispatcher 真实的职责是创建具有线程派发能力的 ChannelHandler，比如 AllChannelHandler、MessageOnlyChannelHandler 和 ExecutionChannelHandler 等，其本身并不具备线程派发能力。

Dispatcher 属于 Dubbo 中的扩展点，这个扩展点用来动态产生 Handler，以满足不同的场景。目前 Dubbo 支持以下 6 种策略调用，如表 6-7 所示。

表 6-7 线程分发策略

分发策略	分发实现	作用
all	AllDispatcher	将所有 I/O 事件交给 Dubbo 线程池处理，Dubbo 默认启用
connection	ConnectionOrderedDispatcher	单独线程池处理连接断开事件，和 Dubbo 线程池分开
direct	DirectDispatcher	所有方法调用和事件处理在 I/O 线程中，不推荐
execution	ExecutionDispatcher	只在线程池处理接收请求，其他事件在 I/O 线程池中
message	MessageOnlyChannelHandler	只在线程池处理请求和响应事件，其他事件在 I/O 线程池中
mockdispatcher	MockDispatcher	默认返回 Null

具体业务方需要根据使用场景启用不同的策略。建议使用默认策略即可，如果在 TCP 连接中需要做安全加密或校验，则可以使用 ConnectionOrderedDispatcher 策略。如果引入新的线程池，则不可避免地导致额外的线程切换，用户可在 Dubbo 配置中指定 `dispatcher` 属性让具体策略生效。

Dubbo 请求响应 Handler

在 Dubbo 框架内部，所有方法调用会被抽象成 Request/Response，每次调用（一次会话）都会创建一个请求 Request，如果是方法调用则会返回一个 Response 对象。HeaderExchangeHandler 用来处理这种场景，它主要负责以下 4 种事情。

- (1) 更新发送和读取请求时间戳。
- (2) 判断请求格式或编解码是否有错，并响应客户端失败的具体原因。
- (3) 处理 Request 请求和 Response 正常响应。
- (4) 支持 Telnet 调用

代码清单 6-15 请求响应 Handler 实现

```
@Override  
public void received(Channel channel, Object message) throws RemotingException {  
    channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis()); ←① 更新事件时间戳  
    ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);  
    try {  
        if (message instanceof Request) {  
            Request request = (Request) message;  
            if (request.isEvent()) {  
                handlerEvent(channel, request); ←② 处理 readonly 事件，在 channel 中打标  
            } else {  
                if (request.isTwoWay()) {  
                    Response response = handleRequest(exchangeChannel, request); ←③ 处理方法调用并返回给客户端  
                    channel.send(response);  
                } else {  
                    handler.received(exchangeChannel, request.getData());  
                }  
            }  
        } else if (message instanceof Response) {  
            handleResponse(channel, (Response) message); ←④ 接收响应  
        } else if (message instanceof String) {  
            if (isClientSide(channel)) { ←⑤ 客户端不支持 Telnet 调用  
                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); ←⑥ 触发 Telnet 调用，并返回  
                if (echo != null && echo.length() > 0) {  
                    channel.send(echo);  
                }  
            }  
        } else {  
            handler.received(exchangeChannel, message);  
        }  
    } finally {  
        HeaderExchangeChannel.removeChannelIfDisconnected(channel);  
    }  
}
```

```
    channel.send(echo);  
}  
}  
}  
} else {  
    handler.received(exchangeChannel, message);  
}  
}  
} finally {  
    HeaderExchangeChannel.removeChannelIfDisconnected(channel);  
}  
}
```

- ①：负责响应读取时间并更新时间戳，在Dubbo心跳处理中会使用当前值并判断是否超过空闲时间。
- ②：主要处理事件类型，目前主要处理readonly事件，用于Dubbo优雅停机。当注册中心反注册元数据时，因为网络原因，客户端不能及时感知注册中心事件，服务端会发送 readonly报文告知下线。
- ④：处理收到的Response响应，告知业务调用方。
- ⑤：校验客户端不支持Telnet调用，因为只有服务提供方暴露服务才有意义。
- ⑥：触发Telnet调用，并将字符串返回给Telnet客户端，关于Telent调用在前面已经分析过了。

接下来我们继续分析如何处理请求和响应（`HeaderExchangeHandler#handleRequest`, `handleResponse`），如代码清单 6-16 所示。

代码清单 6-16 处理请求报文

```
Response handleRequest(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); ←① 处理请求格式不正确(编解码)，并把异常转换成字符串返回
        else msg = data.toString();
        res.setErrorMassage("Fail to decode request due to: " + msg);
        res.setStatus(Response.BAD_REQUEST);

        return res;
    }

    // find handler by message class.
    Object msg = req.getData();
    try {
        Object result = handler.reply(channel, msg); ←② 调用 DubboProtocol#reply,
        res.setStatus(Response.OK);                  触发方法调用
        res.setResult(result);
    } catch (Throwable e) {
        res.setStatus(Response.SERVICE_ERROR); ←③ 方法调用失败
        res.setErrorMassage(StringUtils.toString(e));
    }
    return res;
}

static void handleResponse(Channel channel, Response response) throws
RemotingException {
    if (response != null && !response.isHeartbeat()) {
        DefaultFuture.received(channel, response); ←④ 唤醒阻塞的线程
    }
}
```

在处理请求时，因为在编解码层报错会透传到Handler，所以在①中首先会判断是否是因为请求报文不正确，如果发生错误，则服务端会将具体异常包装成字符串返回，如果直接使用异常对象，则可能造成无法序列化的错误。在②中触发Dubbo协议方法调用，并且把方法调用返回值发送给客户端。如果调用发生未知错误，则会通过③做容错并返回。当发送请求时，会在DefaultFuture中保存请求对象并阻塞请求线程，在④中会唤醒阻塞线程并将Response中的结果通知调用方。

Dubbo 心跳 Handler

Dubbo默认客户端和服务端都会发送心跳报文，用来保持TCP长连接状态。在客户端和服务端，Dubbo内部开启一个线程循环扫描并检测连接是否超时，在服务端如果发现超时则会主动关闭客户端连接，在客户端发现超时则会主动重新创建连接。默认心跳检测时间是60秒，具体应用可以通过heartbeat进行配置。

- ①：遍历所有的Channel,在服务端对应的是所有客户端连接，在客户端对应的是服务端连接。
- ②：主要忽略已经关闭的Socket连接。
- ③：判断当前TCP连接是否空闲，如果空闲就发送心跳报文。目前判断是否是空闲的，根据Channel是否有读或写来决定，比如1分钟内没有读或写就发送心跳报文。
- ④：处理客户端超时重新建立TCP连接，目前的策略是检查是否在3分钟内(用户可以设置)都没有成功接收或发送报文。如果在服务端监测则会通过
- ⑤主动关闭远程客户端连接。

Dubbo集群容错

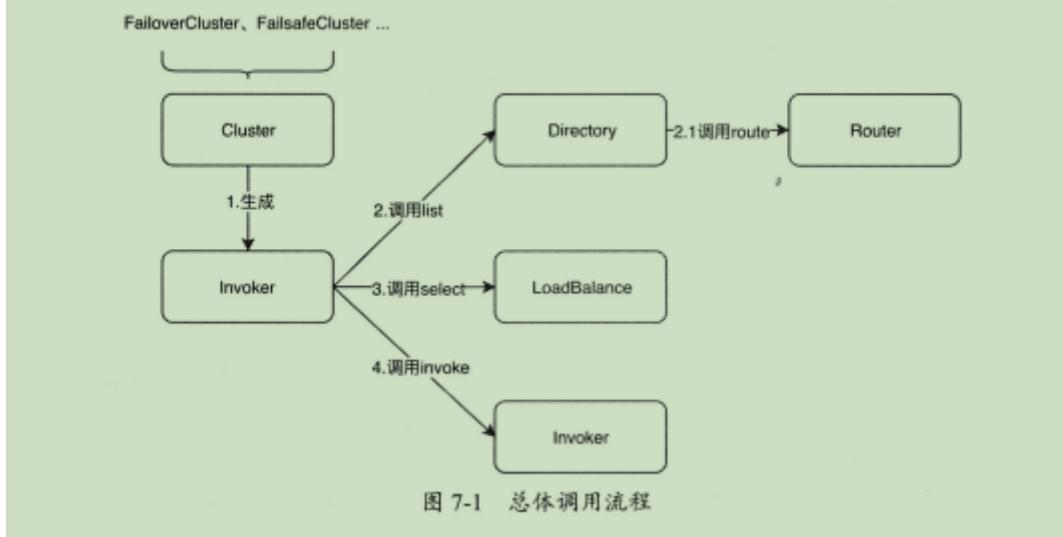
Cluster 层概述

我们可以把Cluster看作一个集群容错层，该层中包含Cluster、Directory、Router、LoadBalance几大核心接口。注意这里要区分Cluster层和Cluster接口，Cluster层是抽象概念，表示的是对外的整个集群容错层；Cluster是容错接口，提供Failover、Failfast等容错策略。

由于Cluster层的实现众多，因此本节介绍的流程是一个基于Abstractclusterinvoker的全量流程，某些实现可能只使用了该流程的一小部分。Cluster的总体工作流程可以分为以下几步：

- 1 生成Invoker对象。不同的Cluster实现会生成不同类型的Clusterinvoker对象并返回。然后调用Clusterinvoker的Invoker方法，正式开始调用流程。
- 2 获得可调用的服务列表。首先会做前置校验，检查远程服务是否已被销毁。然后通过Directory#list方法获取所有可用的服务列表。接着使用Router接口处理该服务列表，根据路由规则过滤一部分服务，最终返回剩余的服务列表。
- 3 做负载均衡。在第2步中得到的服务列表还需要通过不同的负载均衡策略选出一个服务，用作最后的调用。首先框架会根据用户的配置，调用ExtensionLoader获取不同负载均衡策略的扩展点实现（具体负载均衡策略会在后面讲解）。然后做一些后置操作，如果是异步调用则设置调用编号。接着调用子类实现的doInvoke方法（父类专门留了这个抽象方法让子类实现），子类会根据具体的负载均衡策略选出一个可以调用的服务。
- 4 做RPC调用。首先保存每次调用的Invoker到RPC上下文，并做RPC调用。然后处理调用结果，对于调用出现异常、成功、失败等情况，每种容错策略会有不同的处理方式。

总体调用流程如图 7-1 所示。



容错机制的实现

Cluster接口一共有9种不同的实现，每种实现分别对应不同的ClusterInvoker

容错机制概述

表 7-1 容错机制的特性

机制名	机制简介
Failover	当出现失败时，会重试其他服务器。用户可以通过 <code>retries="2"</code> 设置重试次数。这是 Dubbo 的默认容错机制，会对请求做负载均衡。通常使用在读操作或幂等的写操作上，但重试会导致接口的延迟增大，在下游机器负载已经达到极限时，重试容易加重下游服务的负载
Failfast	快速失败，当请求失败后，快速返回异常结果，不做任何重试。该容错机制会对请求做负载均衡，通常使用在非幂等接口的调用上。该机制受网络抖动的影响较大
Failsafe	当出现异常时，直接忽略异常。会对请求做负载均衡。通常使用在“佛系”调用场景，即不关心调用是否成功，并且不想抛异常影响外层调用，如某些不重要的日志同步，即使出现异常也无所谓
Failback	请求失败后，会自动记录在失败队列中，并由一个定时线程池定时重试，适用于一些异步或最终一致性的请求。请求会做负载均衡
Forking	同时调用多个相同的服务，只要其中一个返回，则立即返回结果。用户可以配置 <code>forks="最大并行调用数"</code> 参数来确定最大并行调用的服务数量。通常使用在对接口实时性要求极高的调用上，但也会浪费更多的资源
Broadcast	广播调用所有可用的服务，任意一个节点报错则报错。由于是广播，因此请求不需要做负载均衡。通常用于服务状态更新后的广播

续表

机制名	机制简介
Mock	提供调用失败时，返回伪造的响应结果。或直接强制返回伪造的结果，不会发起远程调用
Available	最简单的方式，请求不会做负载均衡，遍历所有服务列表，找到第一个可用的节点，直接请求并返回结果。如果没有可用的节点，则直接抛出异常
Mergeable	Mergeable 可以自动把多个节点请求得到的结果进行合并

Cluseter的具体实现:用户可以在[dubbo:service](#)、[dubbo:reference](#)、[dubbo:consumer](#)、[dubbo:provider](#)标签上通过cluster属性设置。

对于Failover容错模式，用户可以通过retries属性来设置最大重试次数。可以设置在dubbo: reference标签上，也可以设置在细粒度的方法标签dubbo:method上。

对于Forking容错模式，用户可通过forks="最大并行数"属性来设置最大并行数。假设设置的forks数为n，可用的服务数为v,当n<v时，即可用的服务数大于配置的并行数，则并行请求n个服务；当n>v时，即可用的服务数小于配置的并行数，则请求所有可用的服务V。

对于Mergeable容错模式，用可以在dubbo:reference标签中通过merger="true"开启，合并时可以通过group="*"属性指定需要合并哪些分组的结果。默认会根据方法的返回值自动匹配合并器，如果同一个类型有两个不同的合并器实现，则需要在参数中指定合并器的名字merger="合并器名"）。例如:用户根据某List类型的返回结果实现了多个合并器，则需要手动指定合并器名称，否则框架不知道要用哪个。如果想调用返回结果的指定方法进行合并（如返回了一个Set,想调用Set#addAll方法），则可以通过merger=".addAll"配置来实现。

代码清单7-1官方Mergeable配置示例

```
//搜索所有分组，根据返回结果的类型自动查找合并器。该接口中getMenuItems方法不做合并
```

```
<dubbo:reference interface="com.xxx.MenuService" group="*" merger="true">
    <dubbo:method name="getMenuItems" merger="false" />
</dubbo:reference>
//指定方法合并结果
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger="mymerge" />
</dubbo:reference>
//调用返回结果的指定方法进行合并
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger=".addAll" />
</dubbo:reference>
```

Cluster 接口关系

容错的接口主要分为两大类，第一类是Cluster类，第二类是ClusterInvoker类。Cluster和ClusterInvoker之间的关系也非常简单：Cluster接口下面有多种不同的实现，每种实现中都需要实现接口的join方法，在方法中会“new”一个对应的ClusterInvoker实现。

Cluster是最上层的接口，下面一共有9个实现类。Cluster接口上有SPI注解

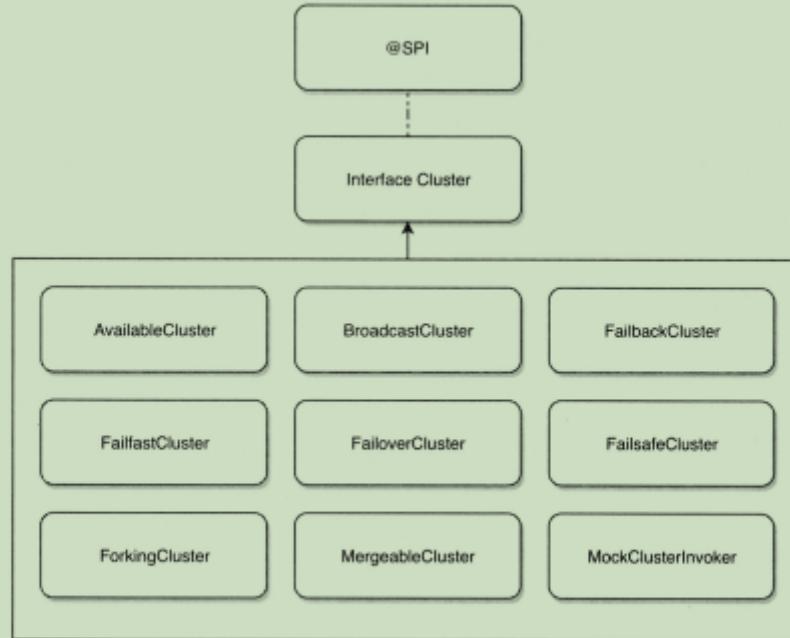


图 7-2 Cluster 接口的类图关系

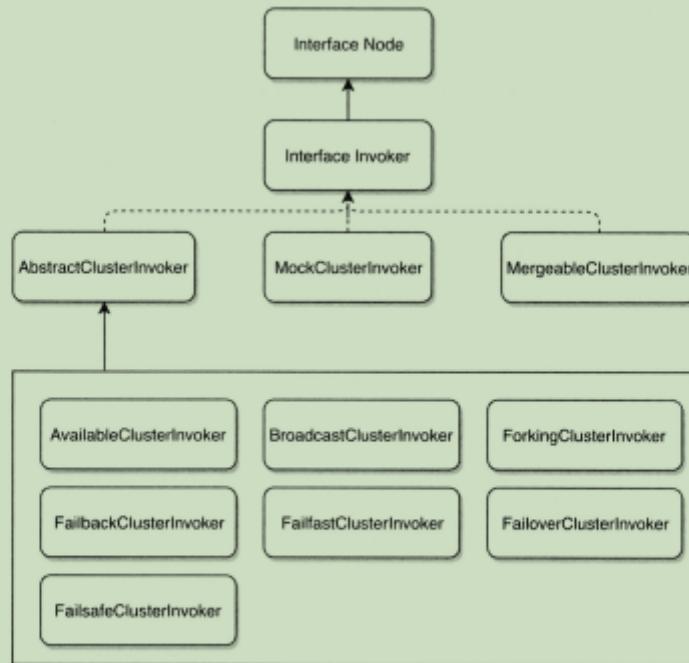


图 7-3 ClusterInvoker 总体类结构

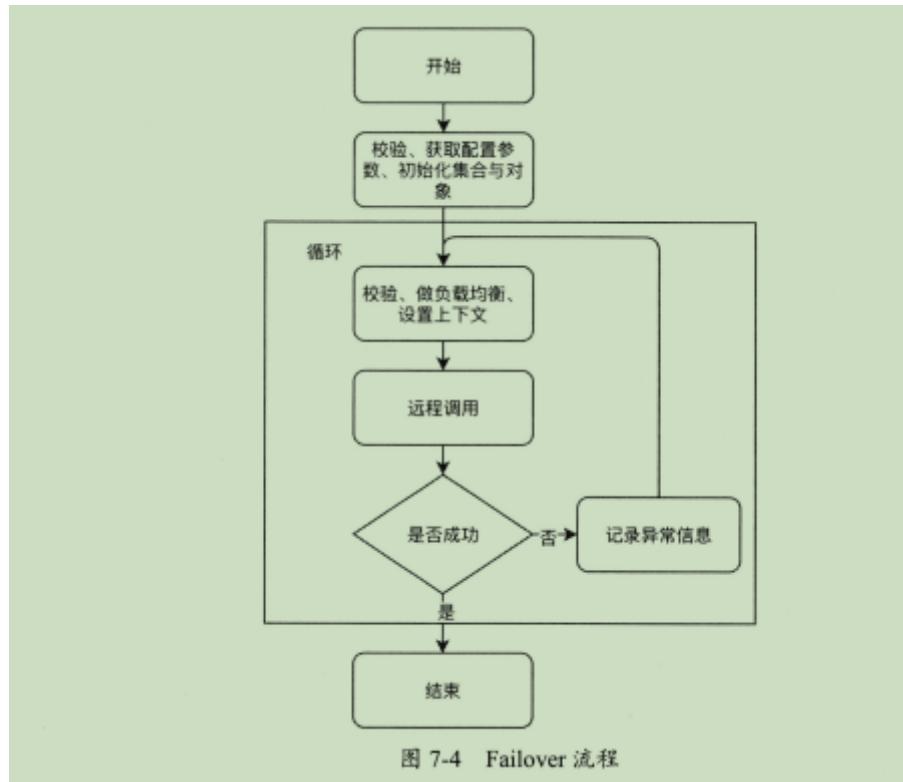
Failover 策略

Cluster 接口上有 SPI 注解 @SPI(FailoverCluster.NAME), 即默认实现是 Failover。该策略的代码逻辑如下:

- (1) 校验。校验从 AbstractClusterInvoker 传入的 Invoker 列表是否为空。
- (2) 获取配置参数。从调用 URL 中获取对应的 retries 重试次数。
- (3) 初始化一些集合和对象。用于保存调用过程中出现的异常、记录调用了哪些节点(这个会在负载均衡中使用，在某些配置下，尽量不要一直调用同一个服务)。
- (4) 使用 for 循环实现重试，for 循环的次数就是重试的次数。成功则返回，否则继续循环。如果 for 循环完，还没有一个成功的返回，则抛出异常，把(3)中记录的信息抛出去。

前3步都是做一些校验、数据准备的工作。第4步开始真正的调用逻辑。以下步骤是 for 循环中的逻辑：

- 校验。如果for循环次数大于1，即有过一次失败，则会再次校验节点是否被销毁、传入的Invoker列表是否为空。
- 负载均衡。调用select方法做负载均衡，得到要调用的节点，并记录这个节点到步骤3的集合里，再把已经调用的节点信息放进RPC上下文中。
- 远程调用。调用invoker#invoke方法做远程调用，成功则返回，异常则记录异常信息，再做下次循环



Failfast 策略

Failfast会在失败后直接抛出异常并返回，实现非常简单，步骤如下：

- (1) 校验。校验从AbstractClusterInvoker传入的Invoker列表是否为空。
- (2) 负载均衡。调用select方法做负载均衡，得到要调用的节点。
- (3) 进行远程调用。在try代码块中调用invoker#invoke方法做远程调用。如果捕获到异常，则直接封装成RpcException抛出。

整个过程非常简短，也不会做任何中间信息的记录。

Failsafe 策略

Failsafe调用时如果出现异常，则会直接忽略。实现也非常简单，步骤如下：

- 1) 校验传入的参数。校验从AbstractClusterInvoker传入的Invoker列表是否为空。
- 2 负载均衡。调用select方法做负载均衡，得到要调用的节点
- (3)远程调用。在try代码块中调用invoker#invoke方法做远程调用，“catch”到任何异常都直接“吞掉”，返回一个空的结果集。

```

//代码清单7·3 Failsafe调用源码
public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
  try {}  

    //校验传入的参数  

    checkinvokers(invokers, invocation);
  }
}

```

```

//负载均衡
Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
//远程调用
return invoker.invoke(invocation);
} catch (Throwable e) {
    //捕获到异常，直接返回一个空的结果集
    return new RpcResult();
}
}

```

Fallback 策略

Fallback如果调用失败，则会定期重试。FallbackClusterInvoker里面定义了一个ConcurrentHashMap，专门用来保存失败的调用。另外定义了一个定时线程池，默认每5秒把所有失败的调用拿出来，重试一次。如果调用重试成功，则会从ConcurrentHashMap中移除。

doinvoke的调用逻辑如下：

- (1) 校验传入的参数。校验从AbstractClusterInvoker传入的Invoker列表是否为空。
- (2) 负载均衡。调用select方法做负载均衡，得到要调用的节点。
- (3) 远程调用。在try代码块中调用invoker#invoke方法做远程调用，“catch”到异常后直接把invocation保存到重试的ConcurrentHashMap中，并返回一个空的结果集。
- (4) 定时线程池会定时把ConcurrentHashMap中的失败请求拿出来重新请求，请求成功则从ConcurrentHashMap中移除。如果请求还是失败，则异常也会被“catch”住，不会影响ConcurrentHashMap中后面的重试

Available 策略

Available是找到第一个可用的服务直接调用，并返回结果。步骤如下：

- (1) 遍历从AbstractClusterInvoker传入的Invoker列表，如果Invoker是可用的，则直接调用并返回。
- (2) 如果遍历整个列表还没找到可用的Invoker，则抛出异常

Broadcast 策略

Broadcast会广播给所有可用的节点，如果任何一个节点报错，则返回异常。步骤如下：

- (1) 前置操作。校验从AbstractClusterInvoker传入的Invoker列表是否为空；在RPC上下文中设置Invoker列表；初始化一些对象，用于保存调用过程中产生的异常和结果信息等。
- (2) 循环遍历所有Invoker，直接做RPC调用。任何一个节点调用出错，并不会中断整个广播过程，会先记录异常，在最后广播完成后再抛出。如果多个节点异常，则只有最后一个节点的异常会被抛出，前面的异常会被覆盖。

Forking 策略

Forking可以同时并行请求多个服务，有任何一个返回，则直接返回。相对于其他调用策略，Forking的实现是最复杂的。其步骤如下：

- (1) 准备工作。校验传入的Invoker列表是否可用；初始化一个Invoker集合，用于保存真正要调用的Invoker列表；从URL中得到最大并行数、超时时间。
- (2) 获取最终要调用的Invoker列表。假设用户设置最大的并行数为n，实际可以调用的最大服务数为V。如果n<0或n<v，则说明可用的服务数小于用户的设置，因此最终要调用的Invoker H能有v个；如果n>v，则会循环调用负载均衡方法，不断得到可调用的Invoker，加入步骤1中的Invoker集合里。

这里有一点需要注意：在Invoker加入集合时，会做去重操作。因此，如果用户设置的负载均衡策略每次返回的都是同一个Invoker，那么集合中最后只会存在一个Invoker，也就是只会调用一个节点。

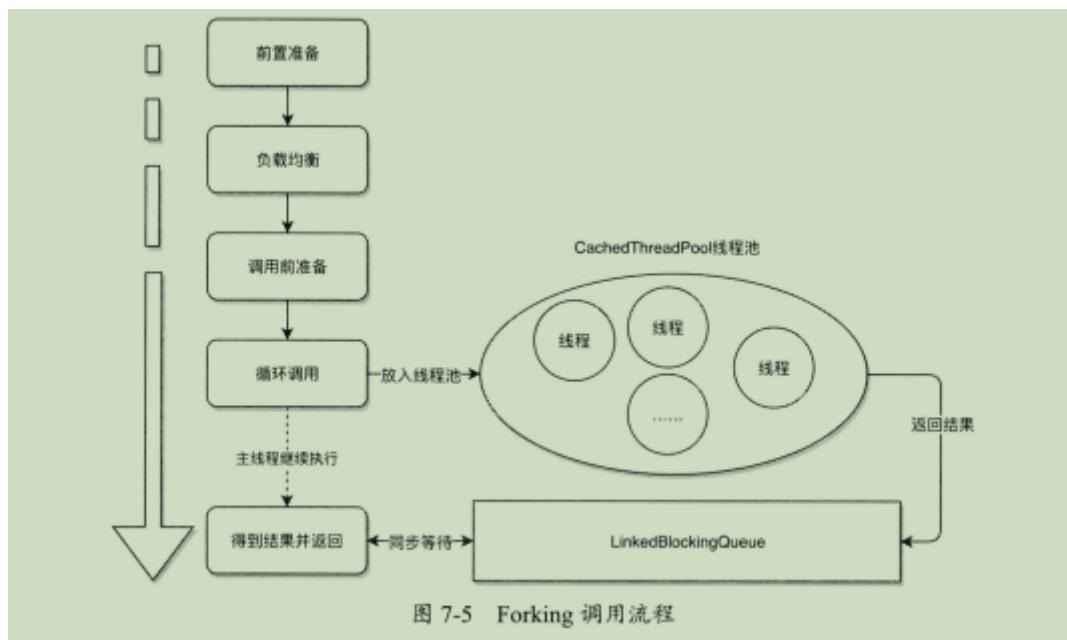
(3) 调用前的准备工作。设置要调用的Invoker列表到RPC上下文；初始化一个异常计数器；初始化一个阻塞队列，用于记录并行调用的结果。

(4) 执行调用。循环使用线程池并行调用，调用成功，则把结果加入阻塞队列；调用失败，则失败计数+1。如果所有线程的调用都失败了，即失败计数 \geq 所有可调用的Invoker时，则把异常信息加入阻塞队列。

这里有一点需要注意：并行调用是如何保证个别调用失败不返回异常信息，只有全部失败才返回异常信息的呢？因为有判断条件，当失败计数 N 所有可调用的Invoker时，才会把异常信息放入阻塞队列，所以只有当最后一个Invoker也调用失败时才会把异常信息保存到阻塞队列，从而达到全部失败才返回异常的效果。

(5) 同步等待结果。由于步骤4中的步骤是在线程池中执行的，因此主线程还会继续往下执行，主线程中会使用阻塞队列的poll(-超时时间)方法，同步等待阻塞队列中的第一个结果，如果是正常结果则返回，如果是异常则抛出。

从上面步骤可以得知，Forking的超时是通过在阻塞队列的poll方法中传入超时时间实现的；线程池中的并发调用会获取第一个正常返回结果。只有所有请求都失败了，Forking才会失败。



Merge

当一个接口有多种实现，消费者又需要同时引用不同的实现时，可以用group来区分不同的实现，如下所示。

```
<dubbo:service group="group1" interface="com.xxx.testService" />
```

```
<dubbo:service group="group2" interface="com.xxx.testservice" />
```

如果我们需要并行调用不同group的服务，并且要把结果集合并起来，贝需要用到Merger特性。Merger实现了多个服务调用后结果合并的逻辑。

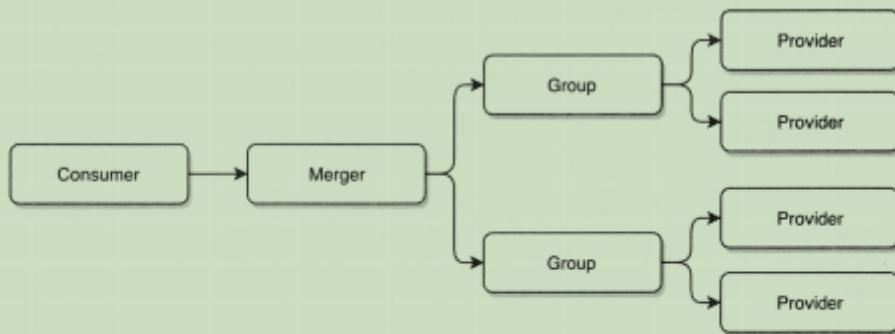


图 7-13 Merger 的工作方式

框架中有一些默认的合并实现。Merger 接口上有@SPI 注解，没有默认值，属于 SPI 扩展点。用户可以基于 Merger 扩展点接口实现自己的自定义类型合并器。本节主要介绍现有抽象逻辑及已有实现。

总体结构

通过 MergerFactory 获得各种具体的 Merger 实现。Merger 的 12 种默认实现的关系如图 7-14 所示。

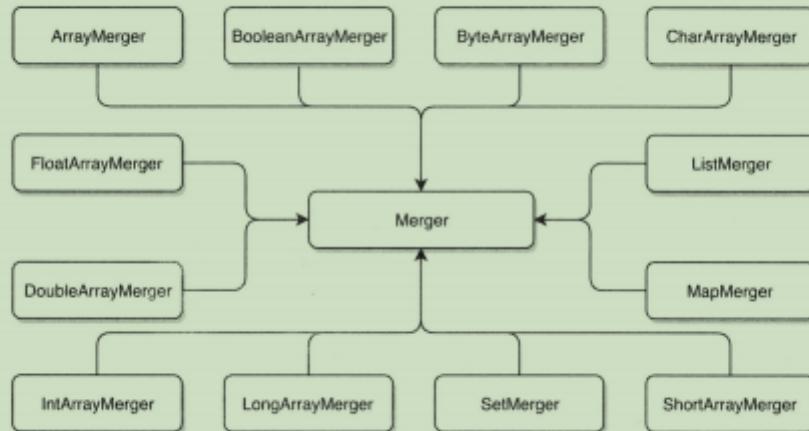


图 7-14 Merger 的 12 种默认实现的关系

内置的合并我们可以分为四类：Array、Set、List、Map，实现都比较简单，我们只列举 MapMerger 的实现。

代码清单 7-19 内置合并器代码示例

```

@Override
public Map<?, ?> merge(Map<?, ?>... items) {
    if (items.length == 0) { ← 如果结果集为空，则直接返回 null
        return null;
    }
    Map<Object, Object> result = new HashMap<Object, Object>(); ←
    for (Map<?, ?> item : items) {           如果结果集不为空则新建一个 Map,
        if (item != null) {                   遍历返回的结果集并放入新的 Map
            result.putAll(item);
        }
    }
    return result;
}
  
```

MergeableClusterInvoker 机制

MergeableClusterInvoker串起了整个合并器逻辑，在讲解MergeableClusterInvoker的机制之前，我们先回顾一下整个调用的过程：MergeableCluster#join方法中直接生成并返回了MergeableClusterInvoker, MergeableClusterInvoker#invoke 方法又通过 MergerFactory 工厂获取不同的Merger接口实现，完成了合并的具体逻辑。

MergeableCluster并没有继承抽象的Cluster实现，而是独立完成了自己的逻辑。因此，它的整个逻辑和之前的Failover等机制不同，其步骤如下：

- (1) 前置准备。通过directory获取所有Invoker列表。
- (2) 合并器检查。判断某个方法是否有合并器，如果没有，则不会并行调用多个group,找到第一个可以调用的Invoker直接调用就返回了。如果有合并器，则进入第3步。
- (3) 获取接口的返回类型。通过反射获得返回类型，后续要根据这个返回值查找不同的合并器。
- (4) 并行调用。把Invoker的调用封装成一个个Callable对象，放到线程池中执行，保存线程池返回的future对象到HashMap中，用于等待后续结果返回。
- (5) 等待fixture对象的返回结果。获取配置的超时参数，遍历(4)中得到的fixture对象，设置Future#get的超时时间，同步等待得到并行调用的结果。异常的结果会被忽略，正常的结果会被保存到list中。如果最终没有返回结果，则直接返回一个空的RpcResult；如果只有一个结果，那么也直接返回，不需要再做合并；如果返回类型是void,则说明没有返回值，也直接返回。
- (6) 合并结果集。如果配置的是merger=".addAll",则直接通过反射调用返回类型中的.addAll方法合并结果集。例如：返回类型是Set,则调用Set.addAll来合并结果

Mock

在Cluster中，还有最后一个MockClusterWrapper,由它实现了 Dubbo的本地伪装。这个功能的使用场景较多，通常会应用在以下场景中：服务降级；部分非关键服务全部不可用，希望主流程继续进行；在下游某些节点调用异常时，可以以Mock的结果返回

Mock常见的使用方式

Mock只有在拦截到RpcException的时候会启用，属于异常容错方式的一种。业务层面其实也可以用try-catch来实现这种功能，如果使用下沉到框架中的Mock机制，则可以让业务的实现更优雅。常见配置如下

配置方式1：可以在配置文件中配置

```
<dubbo:reference interface="com.foo.BarService" mock="true" />
<dubbo:reference interface="com.foo.BarService" mock="com.foo.BarServiceMock" />
<dubbo:reference interface="com.foo.BarService" mock="return null" />
```

//提供Mock实现，如果Mock配置了 true或default，则实现的类名必须是接口名+Mock，如配置方式1否则会直接取Mock参数值作为Mock实现类，如配置方式2

```
package com.foo;
public class BarServiceMock implements BarService {
    public String sayHello(String name) {
        //可以伪造容错数据，此方法只在出现RpcException时被执行
        return "容错数据";
    }
}
```

当接口配置了 Mock,在RPC调用抛出RpcException时就会执行Mock方法。最后一种return null的配置方式通常会在想直接忽略异常的时候使用。

服务的降级是在dubbo-admin中通过override协议更新Invoker的Mock参数实现的。如果Mock参数设置为mock=force: return+null,则表明是强制Mock,强制Mock会让消费者对该服务的调用直接返回null,不再发起远程调用。通常使用在非重要服务已经不可用的时候,可以屏蔽下游对上游系统造成的影响。此外,还能把参数设置为mock=fail: return+null,这样消费者还是会发起远程调用,不过失败后会返回null,但是不抛出异常。最后,如果配置的参数是以throw开头的,即mock= throw,则直接抛出RpcException,不会发起远程调用。

Mock的总体结构

Mock涉及的接口比较多,整个流程贯穿 Cluster 和 Protocol 层,接口之间的逻辑关系如图 7-15 所示。

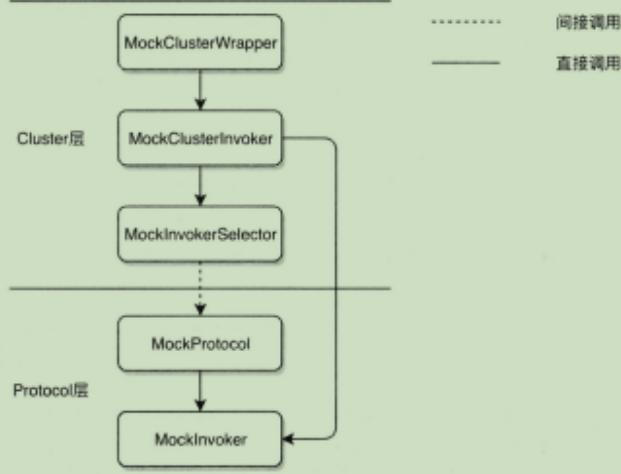


图 7-15 接口之间的逻辑关系

从图7-15我们可以得知,主要流程分为Cluster层和Protocol层。

- MockClusterWrapper是一个包装类, 包装类会被自动注入合适的扩展点实现, 它的逻辑很简单, 只是把被包装扩展类作为初始化参数来创建并返回一个MockClusterInvoker
- MockClusterInvoker和其他的ClusterInvoker一样, 在Invoker方法中完成了主要逻辑。
- MockInvokersSelector 是 Router 接口的一种实现, 用于过滤出 Mock 的 Invoker。
- MockProtocol根据用户传入的URL和类型生成一个MockInvoker0
- MockInvoker实现最终的Invoker逻辑。

MockInvoker与MockClusterInvoker看起来都是Invoker,它们之间有什么区别呢?

首先, 强制Mock、失败后返回Mock结果等逻辑是在MockClusterInvoker里处理的; 其次, MockClusterInvoker在某些逻辑下, 会生成MockInvoker并进行调用; 然后, 在MockInvoker里会处理mock="return null"、mock="throw xxx"或mock=com.xxService这些配置逻辑。最后, MockInvoker还会被MockProtocol在引用远程服务的时候创建。我们可以认为, MockClusterInvoker会处理一些Class级别的Mock逻辑, 例如: 选择调用哪些Mock类。MockInvoker处理的是方法级别的Mock逻辑, 如返回值

Mock的实现原理

MockClusterInvoker 的实现原理

MockClusterWrapper是一个包装类, 它在创建 MockClusterInvoker的时候会把被包装的Invoker传入构造方法, 因此MockClusterInvoker内部天生就含有一个Invoker的引用。

MockClusterInvoker的invoke方法处理了主要逻辑, 步骤如下:

(1) 获取Invoker的Mock参数。前面已经说过，该Invoker是在构造方法中传入的。如果该Invoker根本没有配置Mock，则直接调用Invoker的invoke方法并把结果返回；如果配置了 Mock参数，则进入下一步。

(2) 判断参数是否以force开头，即判断是否强制Mock。如果是强制Mock，则进入doMockInvoke逻辑，这部分逻辑在后面统一讲解。如果不以force开头，则进入失败后Mock的逻辑。

(3) 失败后调用doMockInvoke逻辑返回结果。在try代码块中直接调用Invoker的invoke方法，如果抛出了异常，则在catch代码块中调用doMockInvoke逻辑。

强制Mock和失败后Mock都会调用doMockInvoke逻辑，其步骤如下：

(1) 通过selectMockInvoker获得所有Mock类型的Invoker。selectMockInvoker在对象的attachment属性中偷偷放进一个invocation.need.mock=true的标识。directory在list方法中列出所有Invoker的时候，如果检测到这个标识，则使用MockInvokersSelector来过滤Invoker，而不是使用普通route实现，最后返回Mock类型的Invoker列表。如果一个Mock类型的Invoker都没有返回，则通过directory的URL新创建一个MockInvoker；如果有Mock类型的Invoker，则使用第一个。

(2) 调用MockInvoker的invoke方法。在try-catch中调用invoke方法并返回结果。如果出现了异常，并且是业务异常，则包装成一个RpcResult返回，否则返回RpcException异常。

MockInvokersSelector 的实现原理

在doMockInvoke的第一步中，directory会使用MockInvokersSelector来过滤出Mock类型的Invoker。MockInvokersSelector是Router接口的其中一种实现。它路由时的具体逻辑如下：

(1) 判断是否需要做Mock过滤。如果attachment为空，或者没有invocation.need.mock=true的标识，则认为不需要做Mock过滤，进入步骤2；如果找到这个标识，则进入步骤3。

(2) 获取非Mock类型的Invoker。遍历所有的Invoker，如果它们的protocol中都没有Mock参数，则整个列表直接返回。否则，把protocol中所有没有Mock标识的取出来并返回。

(3) 获取Mock类型的Invoker。遍历所有的Invoker，如果它们的protocol中都没有Mock参数，则直接返回null。否则，把protocol中所有含有Mock标识的取出来并返回。

MockProtocol 与 MockInvoker 的实现原理

MockProtocol也是协议的一种，主要是把注册中心的Mock URL转换为MockInvoker对象。

代码清单 7-22 MockProtocol 源码

```
@Override  
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
    throw new UnsupportedOperationException(); // 不能暴露，否则会抛异常  
}  
  
@Override  
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {  
    return new MockInvoker<T>(url); // 直接把引用的 Mock URL 转换为一个  
} // MockInvoker 对象
```

例如，我们在注册中心/dubbo/com.test.xxxService/providers这个服务提供者的目录下，写入一个Mock的URL：mock://192.168.0.123/com.test.xxxService

在MockInvoker的invoke方法中，主要处理逻辑如下：

(1) 获取Mock参数值。通过URL获取Mock配置的参数，如果为空则抛出异常。优先会获取方法级的Mock参数，例如：以methodName.mock为key去获取参数值；如果取不到，则尝试以mock为key获取对应的参数值。

(2) 处理参数值是return的配置。如果只配置了一个return,即mock=return,则返回一个空的RpcResult;如果return后面还跟了别的参数,则首先解析返回类型,然后结合Mock参数和返回类型,返回Mock值。现支持以下类型的参数: Mock参数值等于empty,根据返回类型返回new xxx()空对象;如果参数值是null> true> false,则直接返回这些值;如果是其他字符串,则返回字符串;如果是数字、List、Map类型,则返回对应的JSON串;如果都没匹配上,则直接返回Mock的参数值。

(3) 处理参数值是throw的配置。如果throw后面没有字符串,则包装成一个RpcException异常,直接抛出;如果throw后面有自定义的异常类,则使用自定义的异常类,并包装成一个RpcException异常抛出。

(4) 处理Mock实现类。先从缓存中取,如果有则直接返回。如果缓存中没有,则先获取接口的类型,如果Mock的参数配置的是true或default,则尝试通过“接口名+Mock”查找Mock实现类,例如: TestService会查找Mock实现TestServiceMock0如果是其他配置方式,则通过Mock的参数值进行查找,例如: 配置了 mock=com.xxx.testservice ,则会查找com.xxx.testservice

Directory 的实现

整个容错过程中首先会使用Directory#list来获取所有的Invoker列表。Directory也有多种实现子类,既可以提供静态的Invoker列表,也可以提供动态的Invoker列表。静态列表是用户自己设置的Invoker列表;动态列表根据注册中心的数据动态变化,动态更新Invoker列表的数据,整个过程对上层透明。

总体实现



Directory是顶层的接口。AbstractDirectory封装了通用的实现逻辑。抽象类包含RegistryDirectory和StaticDirectory两个子类。下面分别介绍每个

类的职责和工作:

(1) AbstractDirectory封装了通用逻辑,主要实现了四个方法:检测Invoker是否可用,销毁所有Invoker, list方法,还留了一个抽象的doList方法给子类自行实现。list方法是最主要的方法,用于返回所有可用的list,逻辑分为两步:

- 调用抽象方法doList获取所有Invoker列表,不同子类有不同的实现;
- 遍历所有的router,进行Invoker的过滤,最后返回过滤好的Invoker列表。doList抽象方法则是返回所有的Invoker列表,由于是抽象方法,子类继承后必须要有自己的实现。

(2) RegistryDirectory属于Directory的动态列表实现,会自动从注册中心更新Invoker列表、配置信息、路由列表。

(3) StaticDirectory属于Directory的静态列表实现,即将传入的Invoker列表封装成静态的Directory对象,里面的列表不会改变。因为Cluster「#join(Directory <T> directory)方法需要传入Directory对象,因此该实现主要使用在一些上层已经知道自己要调用哪些Invoker,只需要包装一个Directory对象返回即可的场景。在ReferenceConfig#createProxy和RegistryDirectory#toMergeMethodInvokerMap中使用了Cluster#join方法。StaticDirectory的逻辑非常简单,在构造方法中需要传入Invoker列表,doList方法则直接返回初始化时传入的列表。

RegistryDirectory 的实现

RegistryDirectory中有两条比较重要的逻辑线，第一条，框架与注册中心的订阅，并动态更新本地Invoker列表、路由列表、配置信息的逻辑；第二条，子类实现父类的doList方法。

订阅与动态更新

订阅和动态更新逻辑。这个逻辑主要涉及subscribe、notify、refreshinvoker三个方法

notify就是监听到配置中心对应的URL的变化，然后更新本地的配置参数。监听的URL分为三类：配置configurators、路由规则router、Invoker列表。工作流程如下：

(1) 新建三个List,分别用于保存更新的Invoker URL、路由配置URL、配置URL。遍历监听返回的所有URL,分类后放入三个List中。

(2) 解析并更新配置参数。

- 对于router类参数，首先遍历所有router类型的URL,然后通过Router工厂把每个URL包装成路由规则，最后更新本地的路由信息。这个过程会忽略以empty开头的URL
- 对于Configurator类的参数，管理员可以在dubbo-admin动态配置功能上修改生产者的参数，这些参数会保存在配置中心的configurators类目下。notify监听到URL配置参数的变化，会解析并更新本地的Configurator配置。
- 对于Invoker类型的参数，如果是empty协议的URL,则会禁用该服务，并销毁本地缓存的Invoker；如果监听到的Invoker类型URL都是空的，则说明没有更新，直接使用本地的老缓存；如果监听到的Invoker类型URL不为空，则把新的URL和本地老的URL合并，创建新的Invoker,找出差异的老Invoker并销毁

监听更新配置的整个过程如图 7-7 所示。

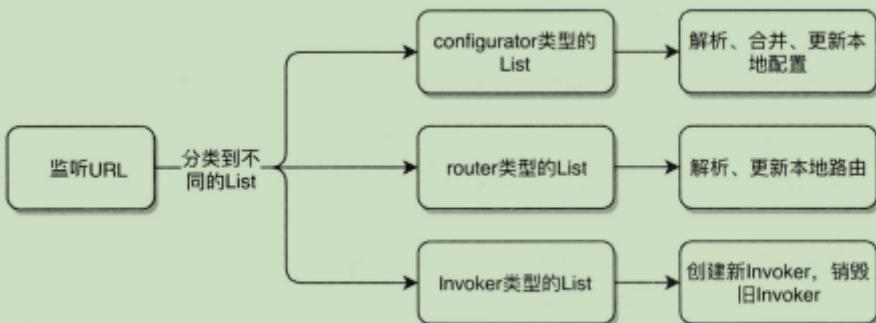
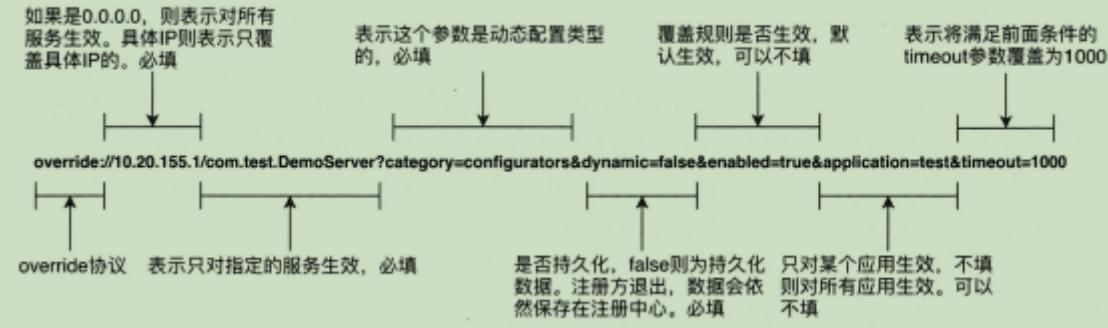


图 7-7 监听更新配置的整个过程

dubbo-admin 上更新路由规则或参数是通过“`override://`”协议实现的，dubbo-admin 的使用方法可以查看官方文档。`override` 协议的 URL 会覆盖更新本地 URL 中对应的参数。如果是“`empty://`”协议的 URL，则会清空本地的配置，这里会调用 Configurator 接口来实现该功能。`override` 参数示例如图 7-8 所示，如果有其他的参数需要覆盖，则直接加在 URL 上即可。



doList的实现

notify中更新的Invoker列表最终会转化为一个字典Map<String, List<Invoker>>methodInvokerMap。key是对应的方法名称，value是整个Invoker列表。doList的最终目标就是在字典里匹配出可以调用的Invoker列表，并返回给上层。其主要步骤如下：

- (1) 检查服务是否被禁用。如果配置中心禁用了某个服务，则该服务无法被调用。如果服务被禁用则会抛出异常。
- (2) 根据方法名和首参数匹配Invoker。这是一个比较奇特的特性。根据方法名和首参数查找对应的Invoker列表，暂时没看到相关的应用场景。

代码清单 7-5 首参数匹配 Invoker 使用示例

```
void test(String arg) ← 假设有 test 方法
test("123"); ← 调用该方法
test.123 ← 用 test.123 在 methodInvokerMap 中匹配对应的 Invoker 列表
```

如果在这一步没有匹配到Invoker列表，则进入第3步。

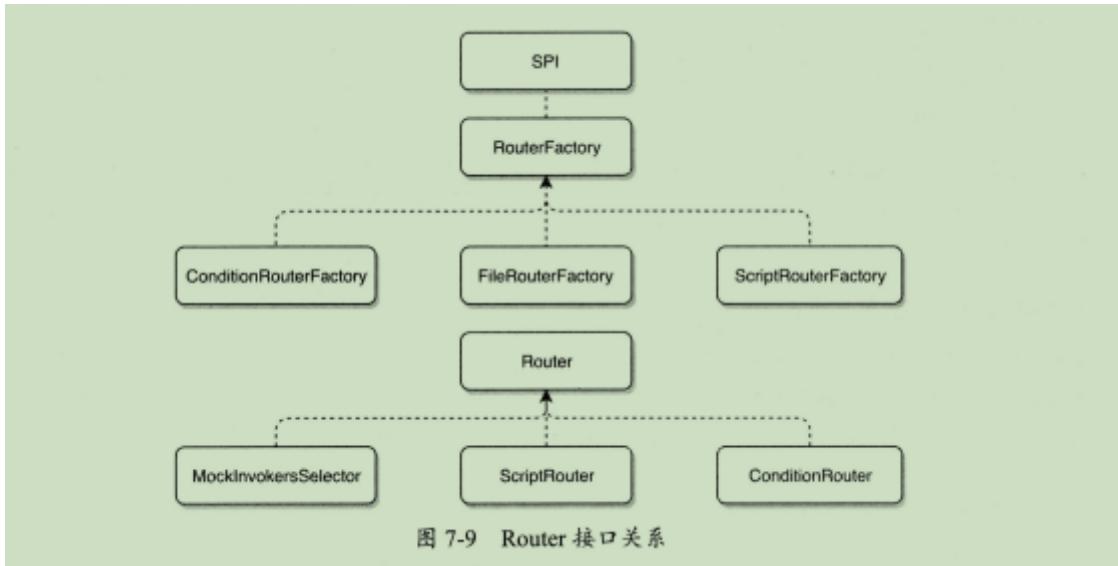
- (3) 根据方法名匹配Invoker以方法名为key去methodInvokerMap中匹配Invoker列表，如果还是没有匹配到，则进入第4步。
- (4) 根据“*”匹配Invoker用星号去匹配Invoker列表，如果还没有匹配到，则进入最后一步兜底操作。
- (5) 遍历methodInvokerMap,找到第一个Invoker列表返回。如果还没有，则返回一个空列表。

路由的实现

路由接口会根据用户配置的不同路由策略对Invoker列表进行过滤，只返回符合规则的Invoker。例如：如果用户配置了接口 A 的所有调用，都使用 IP 为 192.168.1.22 的节点，则路由会过滤其他的 Invoker，只返回 IP 为 192.168.1.22 的 Invoker。

路由的总体结构

路由分为条件路由、文件路由、脚本路由，对应dubbo-admin中三种不同的规则配置方式。条件路由是用户使用Dubbo定义的语法规则去写路由规则;文件路由则需要用户提交一个文件,里面写着对应的路由规则，框架基于文件读取对应的规则；脚本路由则是使用JDK自身的脚本引擎解析路由规则脚本，所有JDK脚本引擎支持的脚本都能解析，默认是JavaScript



代码清单7-7路由的SPI配置

```
file=org.apache.dubbo.rpc.cluster.router.file.FileRouterFactory  
script=org.apache.dubbo.rpc.cluster.router.script.ScriptRouterFactory  
condition=org.apache.dubbo.rpc.cluster.router.condition.ConditionRouterFactory
```

条件路由的参数规则

条件路由使用的是 condition:// 协议，URL 形式是“condition:// 0.0.0.0/com.fbo.BarService?category=routers&dynamic=false&rule=”+ URL.encode(“host = 10.20.153.10 => host = 10.20.153.11”)。我们可以看到，最后的路由规则会用 URL.encode 进行编码。

表 7-2 路由规则

参数名称	含义
condition://	表示路由规则的类型，支持条件路由规则和脚本路由规则，可扩展，必填
0.0.0.0	表示对所有 IP 地址生效，如果只想对某个 IP 的生效，则填入具体 IP，必填
com.foo.BarService	表示只对指定服务生效，必填
category=routers	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，默认生效
force=false	当路由结果为空时，是否强制执行，如果不强制执行，则路由结果为空的路由规则将自动失效，可不填，默认为 false

续表

参数名称	含义
runtime=false	是否在每次调用时执行路由规则，否则只在提供者地址列表变更时预先执行并缓存结果，调用时直接从缓存中获取路由结果。如果用了参数路由，则必须设为 true，需要注意设置会影响调用的性能，可不填，默认为 false
priority=1	路由规则的优先级，用于排序，优先级越大越靠前执行，可不填，默认为 0
rule=URL.encode("host = 10.20.153.10 => host = 10.20.153.11")	表示路由规则的内容，必填

代码清单7-8路由规则配置示例

```
method = find* => host = 192.168.1.22
```

- 这条配置说明所有调用find开头的方法都会被路由到IP为192.168.1.22的服务节点上。
- => 之前的部分为消费者匹配条件，将所有参数和消费者的URL进行对比，当消费者满足匹配条件时，对该消费者执行后面的过滤规则。
- => 之后的部分为提供者地址列表的过滤条件，将所有参数和提供者的URL进行对比，消费者最终只获取过滤后的地址列表。
 - 如果匹配条件为空，则表示应用于所有消费方，如=> host != 192.168.1.22。
 - 如果过滤条件为空，则表示禁止访问，如host = 192.168.1.22 =>。整个规则的表达式支持 protocol 等占位符方式，也支持=、!=等条件。值可以支持多个，用逗号分隔，如host = 192.168.1.22,192.168.1.23；如果以""号结尾，则说明是通配符，如host = 192.168.1.表示匹配192.168.1网段下所有的IP。

条件路由的实现

条件路由的具体实现类是ConditionRouter, Dubbo会根据自定义的规则语法来实现路由规则。

ConditionRouter构造方法的逻辑

ConditionRouterFactory在初始化ConditionRouter的时候，其构造方法中含有规则解析的逻辑。步骤如下：

(1) 根据URL的键rule获取对应的规则字符串。以=>为界，把规则分成两段，前面部分为whenRule,即消费者匹配条件；后面部分为thenRule,即提供者地址列表的过滤条件。我们以代码清单7-8的规则为例，其会被解析为whenRule: method = find*和thenRule: host =192.168.1.22

(2) 分别解析两个路由规则。调用parseRule方法，通过正则表达式不断循环匹配whenRule和thenRule字符串。解析的时候，会根据key-value之间的分隔符对key-value做分类(如果A=B,则分隔符为=)，支持的分隔符形式有：A=B、A&B、A!=B、A,B这4种形式。最终参数都会被封装成一个个MatchPair对象，放入Map中保存。Map的key是参数值，value是MatchPair对象。若以代码清单7-8的规则为例，则会生成以method为key的when Map,以host为key的then Mapo value 则分别是包装了find 和 192.168.1.22 的MatchPair 对象。MatchPair对象是用来做什么的呢？这个对象一共有两个作用。第一个作用是通配符的匹配和占位符的赋值。MatchPair对象是内部类，里面只有一个isMatch方法，用于判断值是否能匹配得上规则。规则里的\$、等通配符都会在MatchPair对象中进行匹配。其中\$支持protocol、username、password、host、port、path这几个动态参数的占位符。例如：规则中写了\$protocol,则会自动从URL中获取protocol的值，并赋值进去。第二个作用是缓存规则。MatchPair对象中有两个Set集合，一个用于保存匹配的规则，如=find；另一个则用于保存不匹配的规则，如!=find.这两个集合在后续路由规则匹配的时候会使用到

route方法的实现原理

ConditionRouter继承了 Router接口，需要实现接口的route方法。该方法的主要功能是过滤出符合路由规则的Invoker列表，即做具体的条件匹配判断，其步骤如下：

(1) 校验。如果规则没有启用，则直接返回；如果传入的Invoker列表为空，则直接返回空；如果没有任何的whenRule匹配，即没有规则匹配，则直接返回传入的Invoker列表；如果whenRule有匹配的，但是thenRule为空，即没有匹配上规则的Invoker,则返回空。

(2) 遍历Invoker列表，通过thenRule找出所有符合规则的Invoker加入集合。例如：匹配规则中的method名称和当前URL中的method是不是相等。

(3) 返回结果。如果结果集不为空，则直接返回；如果结果集为空，但是规则配置了force=true,即强制过滤，那么就会返回空结果集；非强制则不过滤，即返回所有Invoker列表。

文件路由的实现

文件路由是把规则写在文件中，文件中写的是自定义的脚本规则，可以是JavaScript,Groovy等，URL中对应的key值填写的是文件的路径。文件路由主要做的就是把文件中的路由脚本读出来，然后调用路由的工厂去匹配对应的脚本路由做解析。

代码清单 7-9 文件路由实现逻辑

```
//  
String protocol = url.getParameter(Constants.ROUTER_KEY, ScriptRouterFactory.NAME);  
String type = null;           ← 把类型为 file 的 protocol 替换为 script 类型  
String path = url.getPath();  ← 解析文件的后缀名，后续用于匹配的  
if (path != null) {          ← 到底是什么脚本，如 JS、Groovy 等  
    int i = path.lastIndexOf('.');  
    if (i > 0) {  
        type = path.substring(i + 1);  
    }  
}  
                                ← 读取文件  
String rule = IOUtils.read(new FileReader(new File(url.getAbsolutePath())));  
boolean runtime = url.getParameter(Constants.RUNTIME_KEY, false);  
URL script = url.setProtocol(protocol)           ← 生成路由工厂可以识别的  
.addParameter(Constants.TYPE_KEY, type)          URL，并把参数添加进去  
.addParameter(Constants.RUNTIME_KEY, runtime)  
.addParameterAndEncoded(Constants.RULE_KEY, rule);  
return routerFactory.getRouter(script);           ← 再次调用路由的工厂，由于前面配置了 protocol  
                                                为 script 类型，这里会使用脚本路由进行解析
```

脚本路由的实现

脚本路由使用JDK自带的脚本解析器解析脚本并运行，默认使用JavaScript解析器，其逻辑分为构造方法和route方法两大部分。构造方法主要负责一些初始化的工作，route方法则是具体的过滤逻辑执行的地方。

代码清单 7-10 脚本示例

```
function route(invokers) {  
    var result = new java.util.ArrayList(invokers.size());           ← 创建了一个 List  
    for (i = 0; i < invokers.size(); i++) {  
        if ("10.20.153.10".equals(invokers.get(i).getUrl().getHost())) {  
            result.add(invokers.get(i));           ← 遍历传入的所有 Invoker，过滤所有 IP 不是  
        }                                       10.20.153.10 的 Invoker  
    }  
    return result;  
} (invokers);           ← 表示立即执行方法
```

我们在写JavaScript脚本的时候需要注意，一个服务只能有一条规则，如果有多条规则，并且规则之间没有交集，则会把所有的Invoker都过滤。另外，脚本路由中也没看到沙箱约束，因此会有注入的风险。

下面我们来看一下脚本路由的构造方法逻辑：

(1) 初始化参数。获取规则的脚本类型、路由优先级。如果没有设置脚本类型，则默认设置为JavaScript类型，如果没有解析到任何规则，则抛出异常。

(2) 初始化脚本执行引擎。根据脚本的类型，通过Java的ScriptEngineManager创建不同的脚本执行器，并缓存起来。route方法的核心逻辑就是调用脚本引擎，获取执行结果并返回。

负载均衡的实现

包装后的负载均衡

在很多容错策略中都会使用负载均衡方法，并且所有的容错策略中的负载均衡都使用了抽象父类 Abstractclusterinvoker中定义的Invoker select方法，而并不是直接使用LoadBalance方法。因为抽象父类在LoadBalance的基础上又封装了一些新的特性：

(1) 粘滞连接。Dubbo中有一种特性叫粘滞连接，以下内容摘自官方文档：

```
<!--粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者“挂了”，再连接另一台。粘滞连接将自动开启延迟连接，以减少长连接数。-->
<dubbo:protocol name="Hdubbo" sticky="true" />
```

(2) 可用检测。Dubbo调用的URL中，如果含有cluster.availablecheck=false，则不会检测远程服务是否可用，直接调用。如果不设置，则默认会开启检查，对所有的服务都做是否可用的检查，如果不可用，则再次做负载均衡。

(3) 避免重复调用。对于已经调用过的远程服务，避免重复选择，每次都使用同一个节点。这种特性主要是为了避免并发场景下，某个节点瞬间被大量请求。

整个逻辑过程大致可以分为4步：

(1) 检查URL中是否有配置粘滞连接，如果有则使用粘滞连接的Invoker。如果没有配置粘滞连接，或者重复调用检测不通过、可用检测不通过，则进入第2步。

(2) 通过ExtensionLoader获取负载均衡的具体实现，并通过负载均衡做节点的选择。对选择出来的节点做重复调用、可用性检测，通过则直接返回，否则进入第3步。

(3) 进行节点的重新选择。如果需要做可用性检测，则会遍历Directory中得到的所有节点，过滤不可用和已经调用过的节点，在剩余的节点中重新做负载均衡；如果不需要做可用性检测，那么也会遍历Directory中得到的所有节点，但只过滤已经调用过的，在剩余的节点中重新做负载均衡。这里存在一种情况，就是在过滤不可用或已经调用过的节点时，节点全部被过滤，没有剩下任何节点，此时进入第4步。

(4) 遍历所有已经调用过的节点，选出所有可用的节点，再通过负载均衡选出一个节点并返回。如果还找不到可调用的节点，则返回null。

从上述逻辑中，我们可以得知，框架会优先处理粘滞连接。否则会根据可用性检测或重复调用检测过滤一些节点，并在剩余的节点中做负载均衡。如果可用性检测或重复调用检测把节点都过滤了，则兜底的策略是：在已经调用过的节点中通过负载均衡选择出一个可用的节点。

负载均衡的总体结构

表 7-3 负载均衡算法

负载均衡算法名称	效果说明
Random LoadBalance	随机，按权重设置随机概率。在一个节点上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者的权重
RoundRobin LoadBalance	轮询，按公约后的权重设置轮询比例。存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没“挂”，当请求调到第二台时就卡在那里，久而久之，所有请求都卡在调到第二台上
LeastActive LoadBalance	最少活跃调用数，如果活跃数相同则随机调用，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大
ConsistentHash LoadBalance	一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者“挂”时，原本发往该提供者的请求，基于虚拟节点，会平摊到其他提供者，不会引起剧烈变动。默认只对第一个参数“Hash”，如果要修改，则配置<dubbo:parameter key="hash.arguments" value="0,1" />。默认使用 160 份虚拟节点，如果要修改，则配置<dubbo:parameter key="hash.nodes" value="320" />

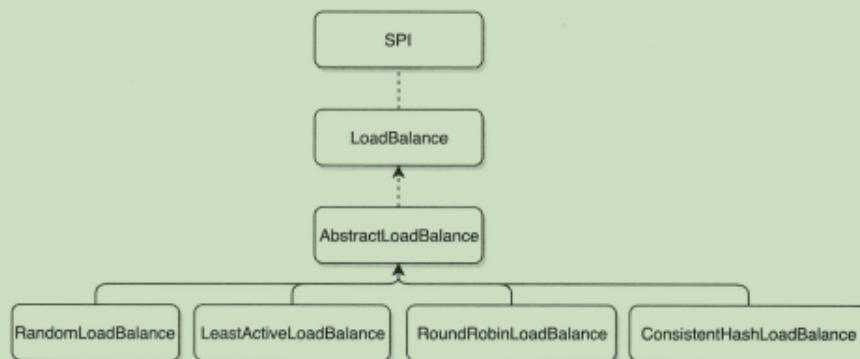


图 7-10 负载均衡的接口关系

抽象父类 `AbstractLoadBalance` 有两个权重相关的方法：`calculateWarmupWeight` 和 `getWeight`。`getWeight` 方法就是获取当前 `Invoker` 的权重，`calculateWarmupWeight` 是计算具体的权重。`getWeight` 方法中会调用 `calculateWarmupWeight`，如代码清单 7-13 所示。

代码清单 7-13

```

int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(),
    Constants.WEIGHT_KEY, Constants.DEFAULT_WEIGHT); ← 通过 URL 获取当前 Invoker
if (weight > 0) {
    long timestamp = invoker.getUrl().getParameter(Constants.REMOTE_TIMESTAMP_KEY, 0L); ←
    if (timestamp > 0L) {
        int uptime = (int) (System.currentTimeMillis() - timestamp); ←
            求差值，得到已经预热了多久
        int warmup = invoker.getUrl().getParameter(Constants.WARMUP_KEY,
            Constants.DEFAULT_WARMUP); ← 获取设置的总预热时间
        if (uptime > 0 && uptime < warmup) {
            weight = calculateWarmupWeight(uptime, warmup, weight); ←
                计算出最后的权重
        }
    }
}
return weight;

```

calculateWarmupWeight的计算逻辑比较简单，由于框架考虑了服务刚启动的时候需要有一个预热的过程，如果一启动就给予100%的流量，则可能会让服务崩溃，因此实现了calculateWarmupWeight方法用于计算预热时候的权重，计算逻辑是：(启动至今时间/给予的预热总时间)X权重。例如：假设我们设置A服务的权重是5,让它预热10分钟，则第一分钟的时候，它的权重变为 $(1/10) \times 5 = 0.5$, $0.5/5 = 0.1$,也就是只承担10%的流量；10分钟后，权重就变为 $(10/10) \times 5 = 5$,也就是权重变为设置的100%，承担了所有的流量。

Random负载均衡

Random负载均衡是按照权重设置随机概率做负载均衡的。这种负载均衡算法并不能精确地平均请求，但是随着请求数量的增加，最终结果是大致平均的。它的负载计算步骤如下：

- (1) 计算总权重并判断每个Invoker的权重是否一样。遍历整个Invoker列表，求和总权重。在遍历过程中，会对比每个Invoker的权重，判断所有Invoker的权重是否相同。
- (2) 如果权重相同，则说明每个Invoker的概率都一样，因此直接用nextInt随机选一个Invoker返回即可。
- (3) 如果权重不同，则首先得到偏移值，然后根据偏移值找到对应的Invoker

代码清单 7-14 随机负载均衡源码

```
int offset = ThreadLocalRandom.current().nextInt(totalWeight);
for (int i = 0; i < length; i++) { ←
    offset -= weights[i];
    if (offset < 0) { ←
        return invokers.get(i);
    }
}
```

遍历所有的Invoker，累减，得到被选中的Invoker

根据总权重计算出一个随机的偏移量，此处使用了ThreadLocalRandom性能会更好

看源码可能还没理解原理，下面做一个场景假设：假设有4个Invoker，它们的权重分别是1、2、3、4，则总权重是 $1+2+3+4=10$ 。说明每个Invoker分别有 $1/10$ 、 $2/10$ 、 $3/10$ 、 $4/10$ 的概率会被选中。然后nextInt(10)会返回0~10之间的一个整数，假设为5。如果进行累减，则减到3后会小于0，此时会落入3的区间，即选择3号Invoker，如图7-11所示。

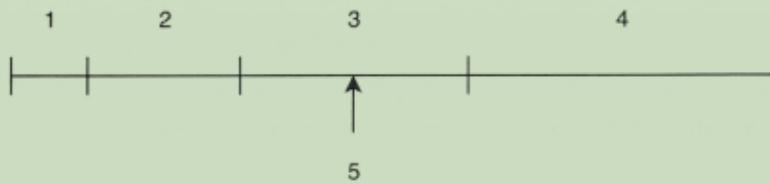


图 7-11 随机选择的区间

RoundRobin 负载均衡

权重轮询负载均衡会根据设置的权重来判断轮询的比例。普通轮询负载均衡的好处是每个节点获得的请求会很均匀，如果某些节点的负载能力明显较弱，则这个节点会堆积比较多的请求。因此普通的轮询还不能满足需求，还需要能根据节点权重进行干预。**权重轮询又分为普通权重轮询和平滑权重轮询**。普通权重轮询会造成某个节点会突然被频繁选中，这样很容易突然让一个节点流量暴增。Nginx中有一种叫平滑轮询的算法(smooth weighted round-robin balancing)，这种算法在轮询时会穿插选择其他节点，让整个服务器选择的过程比较均匀，不会“逮住”一个节点一直调用。Dubbo框架中最新的RoundRobin代码已经改为平滑权重轮询算法。

我们先来看一下Dubbo中RoundRobin负载均衡的工作步骤，如下：

(1) 初始化权重缓存Map。以每个Invoker的URL为key,对象WeightedRoundRobin为value生成一个ConcurrentMap,并把这个 Map 保存到全局的 methodWeightMap 中: ConcurrentHashMap <String, ConcurrentHashMap> methodWeightMap。methodWeightMap的key是每个接口+方法名。这一步只会生成这个缓存Map,但里面是空的, 第2步才会生成每个Invoker对应的键值。

代码清单 7-15 WeightedRoundRobin 对象源码

```
private int weight; ← Invoker 设定的权重
private AtomicLong current = new AtomicLong(0); ←
private long lastUpdate; ←
最后一次更新的时间, 用于后续缓存超时的判断
```

考虑到并发场景下某个 Invoker 会被同时选中, 表示该节点被所有线程选中的权重总和
例如: 某节点权重是 100, 被 4 个线程同时选中, 则变为 400

(2) 遍历所有Invoker。首先, 在遍历的过程中把每个Invoker的数据填充到第1步生成的权重缓存Map中。其次, 获取每个Invoker的预热权重, 新版的框架RoundRobin也支持预热, 通过和Random负载均衡中相同的方式获得预热阶段的权重。如果预热权重和Invoker设置的权重不相等, 则说明还在预热阶段, 此时会以预热权重为准。然后, 进行平滑轮询。每个Invoker会把权重加到自己的current属性上, 并更新当前Invoker的lastUpdate。同时累加每个Invoker的权重到totalweighto最终, 遍历完后, 选出所有Invoker中current最大的作为最终要调用的节点。

(3) 清除已经没有使用的缓存节点。由于所有的Invoker的权重都会被封装成一个weightedRoundRobin对象, 因此如果可调用的Invoker列表数量和缓存weightedRoundRobin对象的Map大小不相等, 则说明缓存Map中有无用数据 (有些Invoker已经不在了, 但Map中还有缓存) .

4 返回Invoker。注意, 返回之前会把当前Invoker的current减去总权重。这是平滑权重轮询中重要的一步。

算法逻辑:

1 每次请求做负载均衡时, 会遍历所有可调用的节点 (Invoker列表) 。对于每个Invoker,让它的current = current + weight。属性含义见weightedRoundRobin对象。同时累加每个Invoker 的 weight 到totalWeight,即 totalweight = totalweight + weight

2 遍历完所有Invoker后, current值最大的节点就是本次要选择的节点。最后, 把该节点的 current 值减去 totalWeight,即 current = current - totalweight

假设有3个Invoker：A、B、C，它们的权重分别为1、6、9，初始current都是0，则平滑权重轮询过程如表7-4所示。

表7-4 平滑权重轮询过程

请求次数	被选中前Invoker的current值	被选中后Invoker的current值	被选中的节点
1	{1,6,9}	{1,6,-7}	C
2	{2,12,2}	{2,-4,2}	B
3	{3,2,11}	{3,2,-5}	C
4	{4,8,4}	{4,-8,4}	B
5	{5,-2,13}	{5,-2,-3}	C
6	{6,4,6}	{-10,4,6}	A
7	{-9,10,15}	{-9,10,-1}	C
8	{-8,16,8}	{-8,0,8}	B
9	{-7,6,17}	{-7,6,1}	C
10	{-6,12,10}	{-6,-4,10}	B
11	{-5,2,19}	{-5,2,3}	C

续表

请求次数	被选中前Invoker的current值	被选中后Invoker的current值	被选中的节点
12	{-4,8,12}	{-4,8,-4}	C
13	{-3,14,5}	{-3,-2,5}	B
14	{-2,4,14}	{-2,4,-2}	C
15	{-1,10,7}	{-1,-6,7}	B
16	{0,0,16}	{0,0,0}	C

从这16次的负载均衡来看，我们可以清楚地得知，A刚好被调用了1次，B刚好被调用了6次，C刚好被调用了9次。符合权重轮询的策略，因为它们的权重比是1:6:9。此外，C并没有被频繁地一直调用，其中会穿插B和A的调用。至于平滑权重轮询的数学原理，就不在本书讨论的范围内了，感兴趣的读者可以去进一步了解。

LeastActive 负载均衡

LeastActive负载均衡称为最少活跃调用数负载均衡，即框架会记下每个Invoker的活跃数，每次只从活跃数最少的Invoker里选一个节点。这个负载均衡算法需要配合ActiveLimitFilter过滤器来计算每个接口方法的活跃数。最少活跃负载均衡可以看作Random负载均衡的“加强版”，因为最后根据权重做负载均衡的时候，使用的算法和Random的一样。

代码清单 7-16 LeastActive 负载均衡源码

```
...
for (int i = 0; i < length; i++) { ← 初始化各种计数器，如最小活跃数计数器、总权重计数器等
    ... ← 获得 Invoker 的活跃数、预热权重
    if (leastActive == -1 || active < leastActive) { ← 第一次，或者发现有更小的活跃数
        ...
        ← 不管是第一次还是有更小的活跃数，之前的计数都要重新开始
        ← 这里置空之前的计数。因为只计数最小的活跃数
    }
} else if (active == leastActive) {
}
}
...
← 当前 Invoker 的活跃数与计数相同
← 说明有 N 个 Invoker 都是最小计数，全部保存到集合中
← 后续就在它们里面根据权重选一个节点
...
← 如果只有一个 Invoker 则直接返回
...
← 如果权重不一样，则使用和 Random 负载均衡一样的权重算法找到一个 Invoker 并返回
...
← 如果权重相同，则直接随机选一个返回
```

从代码清单7-16中我们可以得知其逻辑：遍历所有Invoker,不断寻找最小的活跃数(leastActive),如果有多个Invoker的活跃数都等于leastActive,则把它们保存到同一个集合中，最后在这个Invoker集合中再通过随机的方式选出一个Invoker。

那最少活跃的计数又是如何知道的呢？

在ActiveLimitFilter中，只要进来一个请求，该方法的调用的计数就会原子性+1。整个Invoker调用过程会在try-catch-finally中，无论调用结束或出现异常，finally中都会把计数原子-1。该原子计数就是最少活跃数。

一致性Hash负载均衡

一致性Hash负载均衡可以让参数相同的请求每次都路由到相同的机器上。这种负载均衡的方式可以让请求相对平均，相比直接使用Hash而言，当某些节点下线时，请求会平摊到其他服务提供者，不会引起剧烈变动。

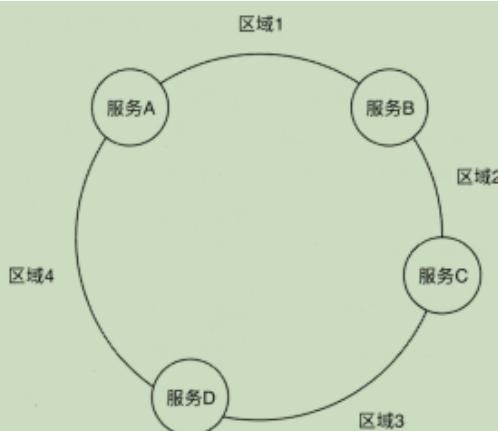


图 7-12 普通一致性 Hash

普通一致性Hash会把每个服务节点散列到环形上，然后把请求的客户端散列到环上，顺时针往前找到的第一个节点就是要调用的节点。假设客户端落在区域2,则顺时针找到的服务C就是要调用的节点。当服务C宕机下线，则落在区域2部分的客户端会自动迁移到服务D上。这样就避免了全部重新散列的问题。

普通的一致性Hash也有一定的局限性，它的散列不一定均匀，容易造成某些节点压力大。因此Dubbo框架使用了优化过的Ketama一致性Hash。这种算法会为每个真实节点再创建多个虚拟节点，让节点在环形上的分布更加均匀，后续的调用也会随之更加均匀。

代码清单 7-17 一致性 Hash 负载均衡源码

```
String methodName = RpcUtils.getMethodName(invocation); ← 获得方法名
String key = invokers.get(0).getUrl().getServiceKey() + "." + methodName; ←
                                                               以接口名+方法名拼接出 key
int identityHashCode = System.identityHashCode(invokers); ←
                                                               把所有可以调用的 Invoker 列表进行 “Hash”
ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key); ←
                                                               现在 Invoker 列表的 Hash 码和之前的不一样，说明
                                                               Invoker 列表已经发生了变化，则重新创建 Selector
if (selector == null || selector.identityHashCode != identityHashCode) {
    selectors.put(key, new ConsistentHashSelector<T>(invokers, methodName,
identityHashCode));
    selector = (ConsistentHashSelector<T>) selectors.get(key);
}
return selector.select(invocation); ← 通过 selector 选出一个 Invoker
```

整个逻辑的核心在 `ConsistentHashSelector` 中，因此我们继续来看 `ConsistentHashSelector` 是如何初始化的。`ConsistentHashSelector` 初始化的时候会对节点进行散列，散列的环形是使用一个 `TreeMap` 实现的，所有的真实、虚拟节点都会放入 `TreeMap`。把节点的 IP+递增数字做“MD5”，以此作为节点标识，再对标识做“Hash”得到 `TreeMap` 的 key，最后把可以调用的节点作为 `TreeMap` 的 value，如代码清单 7-18 所示。

代码清单 7-18 一致性 Hash 散列源码

```
for (Invoker<T> invoker : invokers) { ← 遍历所有的节点
    String address = invoker.getUrl().getAddress(); ← 得到每个节点的 IP
    for (int i = 0; i < replicaNumber / 4; i++) { ←
        byte[] digest = md5(address + i); ←
                                                               replicaNumber 是生成的虚拟节
                                                               点数，默认为 160 个
        for (int h = 0; h < 4; h++) { ←
            long m = hash(digest, h); ←
            virtualInvokers.put(m, invoker); ←
                                                               以 IP+递增数字做 MD5，以此作为节点标识
                                                               对标识做 “Hash” 得到 TreeMap 的 key，以
                                                               Invoker 为 value
        }
    }
}
```

`TreeMap`实现一致性Hash：在客户端调用时候，只要对请求的参数也做“MD5”即可。虽然此时得到的MD5值不一定能对应到`TreeMap`中的一个key,因为每次的请求参数不同。但是由于`TreeMap`是有序的树形结构，所以我们可以调用`TreeMap`的`ceilingEntry`方法，用于返回一个至少大于或等于当前给定key的Entry,从而达到顺时针往前找的效果。如果找不到，则使用`firstEntry`返回第一个节点.

Dubbo过滤器

Dubbo过滤器概述

过滤器的使用

我们知道Dubbo中已经有很多内置的过滤器，并且大多数都是默认启用的，如`ContextFilter`对于自行扩展的过滤器，要如何启用呢？

一种方式是使用`@Activate`注解默认启用；另一种方式是在配置文件中配置，

代码清单10.1过滤器配置

```
<!--消费方调用过程拦截-->
<dubbo:reference filter="xxxJyyy" />
<!--消费方调用过程默认拦截器，将拦截所有reference -->
<dubbo:consumer filter="xxx,yyy"/>
<!--服务提供方调用过程拦截-->
<dubbo:service filter="xxxJyyy" />
<!--服务提供方调用过程默认拦截器，将拦截所有service -->
<dubbo:provider filter="xxxJyyy"/>
```

以上就是常见的配置方式，下面我们来了解一下配置上的一些“潜规则”：

1 过滤器顺序。

- 用户自定义的过滤器的顺序默认会在框架内置过滤器之后，我们可以使用filter="xxx,default"这种配置方式让自定义的过滤器顺序靠前。
- 我们在配置filter="xxx,yyy"时，写在前面的xxx会比yyy的顺序要靠前。

2 剔除过滤器。对于一些默认的过滤器或自动激活的过滤器，有些方法不想使用这些过滤器，则可以使用加过滤器名称来过滤，如filter="-xxFilter"会让xxFilter不生效。如果不想使用所有默认启用的过滤器，则可以配置filter="-default"来进行剔除。

3 过滤器的叠加。如果服务提供者、消费者端都配置了过滤器，则两边的过滤器不会互相覆盖，而是互相叠加，都会生效。如果需要覆盖，则可以在消费方使用“-”的方式剔除对应的过滤器。

过滤器的总体结构

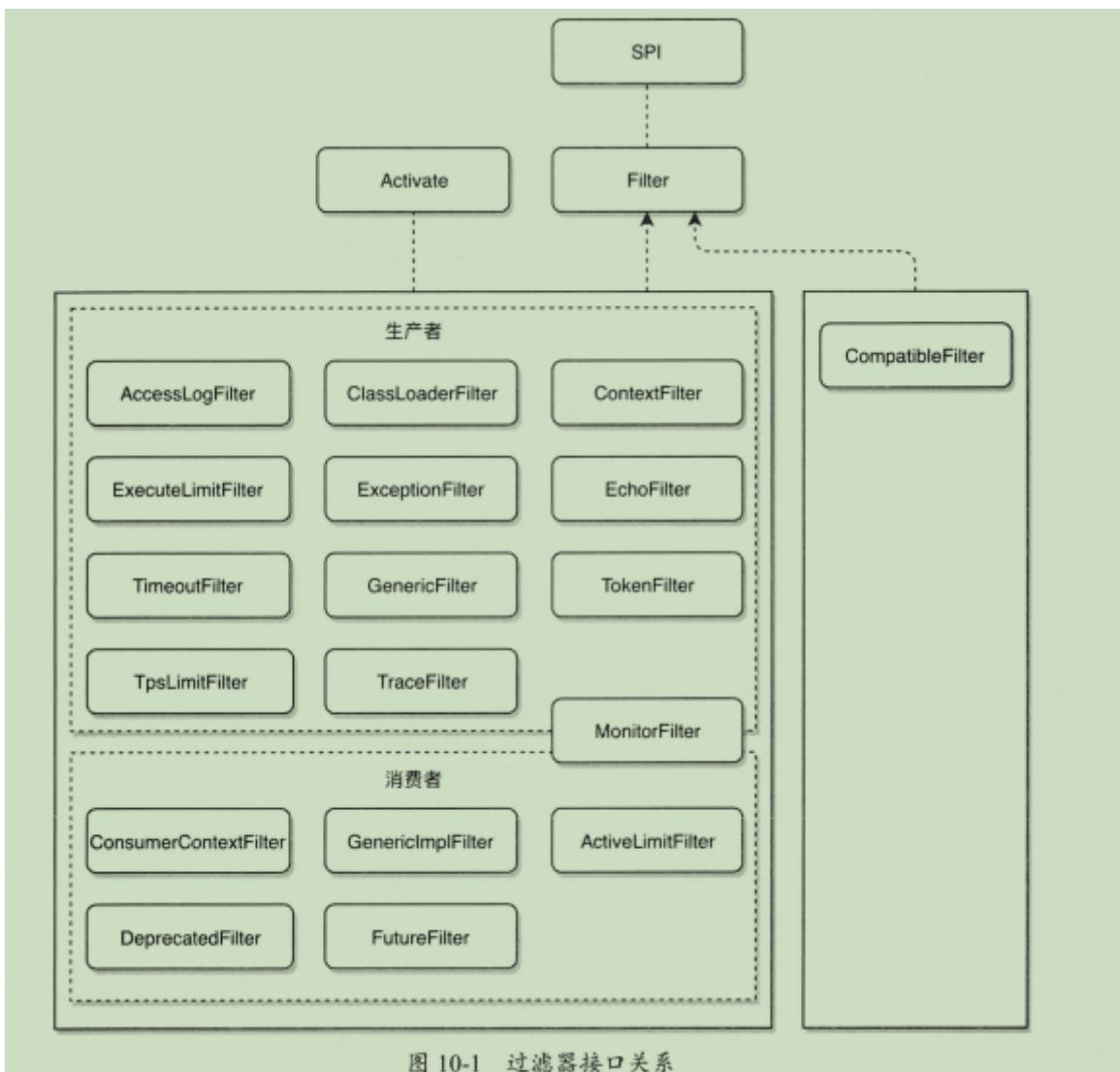


图 10-1 过滤器接口关系

从图10-1可以看到所有的内置过滤器中除了 CompatibleFilter 特别突出，只继承了 Filter 接口，即不会被默认激活，其他的内置过滤器都使用了旧 @Activate 注解，即默认被激活。Filter 接口上有也 PI 注解，说明过滤器是一个扩展点，用户可以基于这个扩展点接口实现自己的过滤器。所有的过滤器会被分为消费者和服务提供者两种类型，消费者类型的过滤器只会在服务引用时被加入Invoker，服务提供者类型的过滤器只会在服务暴露的时候被加入对应的Invoker。MonitorFilter 比较特殊，它会同时在暴露和引用时被加入Invoker。

表 10-1 过滤器作用列表

过滤器名	作用	是否本章讲解	使用方
AccessLogFilter	打印每一次请求的访问日志。如果需要访问的日志只出现在指定的 appender 中，则可以在 log 的配置文件中配置 additivity	是	服务提供者
ActiveLimitFilter	用于限制消费者端对服务端的最大并行调用数	是	消费者
ExecuteLimitFilter	同上，用于限制服务端的最大并行调用数	是	服务提供者
ClassLoaderFilter	用于切换不同线程的类加载器，服务调用完成后会还原回去	是	服务提供者
CompatibleFilter	用于使返回值与调用程序的对象版本兼容，默认不启用。如果启用，则会把 JSON 或 fastjson 类型的返回值转换为 Map 类型；如果返回类型和本地接口中定义的不同，则会做 POJO 的转换	否	-
ConsumerContextFilter	为消费者把一些上下文信息设置到当前线程的 RpcContext 对象中，包括 invocation、local host、remote host 等	否	消费者
ContextFilter	同上，但是为服务提供者服务	否	服务提供者
DeprecatedFilter	如果调用的方法被标记为已弃用，那么 DeprecatedFilter 将记录一个错误消息	是	消费者
EchoFilter	用于回声测试，在之前章节中已经有介绍	否	服务提供者
ExceptionFilter	用于统一的异常处理，防止出现序列化失败	是	服务提供者
GenericFilter	用于服务提供者端，实现泛化调用，实现序列化的检查和处理	否	服务提供者
GenericImplFilter	同上，但用于消费者端	否	消费者
TimeoutFilter	如果某些服务调用超时，则自动记录告警日志	是	服务提供者
TokenFilter	服务提供者下发令牌给消费者，通常用于防止消费者绕过注册中心直接调用服务提供者	是	服务提供者
TpsLimitFilter	用于服务端的限流，注意与 ExecuteLimitFilter 区分	是	服务提供者
FutureFilter	在发起 invoke 或得到返回值、出现异常的时候触发回调事件	是	消费者
TraceFilter	Trace 指令的使用	否	服务提供者
MonitorFilter	监控并统计所有的接口的调用情况，如成功、失败、耗时。后续 DubboMonitor 会定时把该过滤器收集的数据发送到 Dubbo-Monitor 服务上	否	服务提供者 + 消费者

由于表 10-1 中的 GenericFilter、EchoFilter 等过滤器在“第 9 章 Dubbo 高级特性”中

每个过滤器的使用方不一样，有的是服务提供者使用，有的是消费者使用。**Dubbo 是如何保证服务提供者不会使用消费者的过滤器的呢？**

答案就在 @Activate 注解上，该注解可以设置过滤器激活的条件和顺序，如 @Activate (group = Constants.PROVIDER, order = -110000) 表示在服务提供端扩展点实现才有效，并且过滤器的顺序是 -110000。

过滤器链初始化的实现原理

服务的暴露与引用会使用Protocol层，而ProtocolFilterWrapper包装类则实现了过滤器链的组装。在服务的暴露与引用过程中，会使用ProtocolFilterWrapper#buildInvokerChain方法组装整个过滤器链。

代码清单 10-2 服务的暴露与引用过程

```
① 暴露服务的时候会调用 buildInvokerChain
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
        return protocol.export(invoker);
    }
    return protocol.export(buildInvokerChain(invoker, Constants.SERVICE_FILTER_KEY,
    Constants.PROVIDER)); ← 此处会传入 Constants.PROVIDER，标识自己是服
}                                务提供者类型的调用链

public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
        return protocol.refer(type, url); ← ② 引用远程服务的时候也会调
    }                                用 buildInvokerChain
    return buildInvokerChain(protocol.refer(type, url), Constants.REFERENCE_FILTER_KEY,
    Constants.CONSUMER); ← 此处会传入 Constants.CONSUMER，标识自己是消费类型的调用链
}
```

下buildInvokerChain方法是如何构造调用链的，总的来说可以分为两步：

- (1) 获取并遍历所有过滤器。通过ExtensionLoader#getActivateExtension方法获取所有的过滤器并遍历。
- (2) 使用装饰器模式，增强原有Invoker，组装过滤器链。使用装饰器模式，像俄罗斯套娃一样，把过滤器一个又一个地“套”到Invoker上

代码清单 10-3 构造调用链源码

```
Invoker<T> last = invoker; ← 保存引用，后续用于把真正的调用者保存到过滤器链的最后
List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class).
getActivateExtension(invoker.getUrl(), key, group); ←
if (!filters.isEmpty()) {           获取所有的过滤器，包括有@Activate 注解
    filters.forEach(filter -> {      默认启动的和用户在 XML 中自定义配置的
        if (filter instanceof DubboFilter) {
            filter.setInvoker(invoker);
        }
    });
}
for (int i = filters.size() - 1; i >= 0; i--) { ← 对过滤器做倒排遍历，即从尾到头
    final Filter filter = filters.get(i);
    final Invoker<T> next = last; ← 注意这段逻辑，把 last 节点变成 next 节
    last = new Invoker<T>() {           点，并放到 Filter 链的 next 中
        ... ← 省略无关紧要的方法
        @Override
        public Result invoke(Invocation invocation) throws RpcException {
            设置过滤器链的下一个节点，不断循环形成过滤器链
            Result result = filter.invoke(next, invocation); ←
                异步调用和同步调用的处理
            if (result instanceof AsyncRpcResult) { ←
                AsyncRpcResult asyncResult = (AsyncRpcResult) result;
                asyncResult.thenApplyWithContext(r -> filter.onResponse(r, invoker,
invocation));
                return asyncResult;
            } else {
                return filter.onResponse(result, invoker, invocation);
            }
        }
        ....
    };
}
}
return last;
```

源码中为什么要倒排遍历呢？因为是通过从里到外构造匿名类的方式构造Invoker的，所以只有倒排，最外层的Invoker才能是第一个过滤器。我们来看一个例子：

假设有过滤器A、B、C和Invoker，会按照C、B、A倒序遍历，过滤器链构建顺序为：C—Invoker, B—C—Invoker, A—B—C—Invoker。最终调用时的顺序就会变为A是第一个过滤器

服务提供者过滤器的实现原理

AccessLogFilter 的实现原理

AccessLogFilter是一个日志过滤器，如果想记录服务每一次的请求日志，则可以开启这个过滤器。虽然AccessLogFilter有旧Activate注解，默认会被激活，但还是需要手动配置来开启日志的打印。我们有两种方式来配置开启AccessLogFilter

```
<!--将日志输出到应用本身的log中-->
<dubbo:protocol accessLog="true"/>
<!--只是某个服务提供者或消费者打印log-->
<dubbo:provider accessLog="default"/>
<dubbo:service accessLog="true">
<!--将日志输出到指定的文件-->
<dubbo:protocol accessLog="custom-access.log" />
```

AccessLogFilter 的实现

AccessLogFilter的构造方法中会加锁并初始化一个定时线程池ScheduledThreadPool该线程池只有在指定了输出的log文件时才会用到,ScheduledThreadPool中的线程会定时把队列中的日志数据写入文件。在构造方法中主要是初始化线程池,而打印日志的逻辑主要在invoke方法中, 其逻辑如下:

- (1) 获取参数。获取上下文、接口名、版本、分组信息等参数, 用于日志的构建。
- (2) 构建日志字符串。根据步骤 1 中的数据开始组装日志, 最终会得到一个日志字符串, 类似于
[2019-01-15 20:13:58] 192.168.1.17:20881 -> 192.168.1.17:20882 - com.test.demo.DemoService
testFunction java.lang.String [null]

3 日志打印。如果用户配置了使用应用本身的日志组件, 则直接通过封装的LoggerFactory打印日志; 如果用户配置了日志要输出到自定义的文件中, 则会把日志加入一个 ConcurrentMap<String, ConcurrentHashSet>中暂存, key 是自定义的 accesslog 值 (如accesslogicustom-access.log") , value就是对应日志集合。后续等待定时线程不断遍历整个Map, 把日志写入对应的文件.

ExecuteLimitFilter 的实现原理

ExecuteLimitFilter用于限制每个服务中每个方法的最大并发数, 有接口级别和方法级别的配置方式。我们先看看官方文档中是如何配置的:

```
<!--每个方法的并发执行数（或占用线程池线程数）不能超过10个-->
<dubbo:service interface="com.foo.BarService" executes="10" />
<!--限制 com.foo. BarService 的 sayHello方法的并发执行数（或占用线程池线程数）不能超过10
个-->
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10"/>
</dubbo:service>
<!--如果不设置, 则默认不做限制; 如果设置了小于等于0的数值, 那么也会不做任何限制。-->
```

其实现原理是: 在框架中使用一个ConcurrentMap缓存了并发数的计数器。为每个请求URL生成一个IdentityString, 并以此为key; 再以每个IdentityString生成一个RpcStatus对象, 并以此为value。RpcStatus对象用于记录对应的并发数。在过滤器中, 会以try-catch-finally的形式调用过滤器链的下一个节点。因此, 在开始调用之前, 会通过URL获得RpcStatus对象, 把对象中的并发数计数器原子+1, 在finally中再将原子-1。只要在计数器+1的时候, 发现当前计数比设置的最大并发数大时, 就会抛出异常, 提示已经超过最大并发数, 请求就会被终止并直接返回。

ClassLoaderFilter 的实现原理

ClassLoaderFilter主要的工作是: 切换当前工作线程的类加载器到接口的类加载器, 以便和接口的类加载器的上下文一起工作。

代码清单 10-5 ClassLoaderFilter 源码

```
保存当前线程的类加载器
ClassLoader ocl = Thread.currentThread().getContextClassLoader(); <-->
Thread.currentThread().setContextClassLoader(invoker.getInterface().getClassLoader()); <-->
try {
    return invoker.invoke(invocation); <-->
} finally {
    Thread.currentThread().setContextClassLoader(ocl); <-->
}                                把当前线程的上下文类加载器设置为接口的类加载器
                                    继续过滤器链的下一个节点
                                    把当前线程的上下文类加载器还原回去
```

代码清单 10-6 同一个类加载器中的反射调用

```
public class ClassA {           ←① 在 ClassA 中通过反射调用了 ClassB
    public void callClassB() {
        try {
            Class clazz = Class.forName("com.ClassB");
            Method method = clazz.getMethod("print");
            Object o = clazz.newInstance();
            method.invoke(o);
        } catch (Exception e) {
            // ...
        }
    }
}

public class ClassB {           ←② ClassB
    public void print(){
        System.out.println("ClassB");
    }
}

③ 调用 ClassA
ClassA classA = new ClassA();
classA.callClassB();
```

如果 ClassA 和 ClassB 都是同一个类加载器加载的，则它们之间是可以互相访问的，ClassA 的调用会输出 ClassB。但是，如果 ClassA 和 ClassB 是不同的类加载器加载的呢？不同类加载器加载示例如图 10-2 所示。

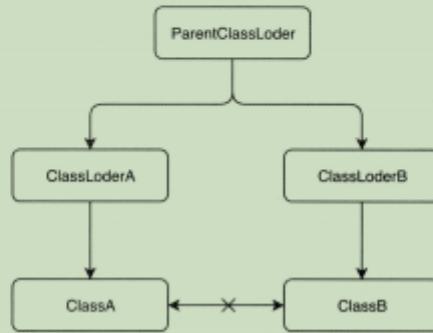


图 10-2 不同类加载器加载示例

如果要实现违反双亲委派模型来查找Class,那么通常会使用上下文类加载器(ContextClassLoader)。当前框架线程的类加载器可能和Invoker接口的类加载器不是同一个，而当前框架线程中又需要获取Invoker的类加载器中的一些 Class,为了避免出现 ClassNotFoundException,此时只需要使用 Thread.currentThread().getContextClassLoader()就可以获取Invoker的类加载器，进而获得这个类加载器中的 Class

ContextFilter 的实现原理

ContextFilter主要记录每个请求的调用上下文。每个调用都有可能产生很多中间临时信息，我们不可能要求在每个接口上都加一个上下文的参数，然后一路往下传。通常做法都是放在ThreadLocal中，作为一个全局参数，当前线程中的任何一个地方都可以直接读/写上下文信息。

ContextFilter就是统一在过滤器中处理请求的上下文信息，它为每个请求维护一个RpcContext对象，该对象中维护两个InternalThreadLocal (它是优化过的ThreadLocal,具体可以搜索Netty的InternalThreadLocal做了什么优化)，分别记录local和server的上下文。每次收到或发起RPC调用的时候，上下文信息都会发生改变。例如：A调用B, B调用C。当A调用B且B还未调用C时，RpcContext中保

存A调用B的上下文；当B开始调用C的时候，RpcContext中保存B调用C的上下文。发起调用时候的上下文是由ConsumerContextFilter实现的，这个是消费者端的过滤器，ContextFilter保存的是收到的请求的上下文。

ContextFilter的主要逻辑如下：

- (1) 清除异步属性。防止异步属性传到过滤器链的下一个环节。
- (2) 设置当前请求的上下文，如Invoker信息、地址信息、端口信息等。如果前面的过滤器已经对上下文设置了一些附件信息(attachments是一个Map,里面可以保存各种key-value数据)，则和Invoker的附件信息合并。
- (3) 调用过滤器链的下一个节点。
- (4) 清除上下文信息。对于异步调用的场景，即使是同一个线程，处理不同的请求也会创建一个新的RpcContext对象。因此调用完成后，需要清理对应的上下文信息。

ExceptionFilter 的实现原理

它的关注点不在于捕获异常，而是为了找到那些返回的自定义异常，但异常类可能不存在于消费者端，从而防止消费者端序列化失败。对于所有没有声明Unchecked的方法抛出的异常，ExceptionFilter会把未引入的异常包装到RuntimeException中，并把异常原因字符串化后返回。

因此，ExceptionFilter的逻辑都在onResponse方法中。ExceptionFilter过滤器会打印出ERROR级别的错误日志，但并不会处理泛化调用，即Invoker的接口是GenericService.

下面我们来看一下onResponse方法中是如何处理异常的：

- (1)判断是否是泛化调用。如果是泛化调用则直接不处理了。
- (2)直接抛出一些异常。如果异常是Java自带异常，并且是必须显式使用try-catch来捕获的异常，则直接抛出。如果异常在Invoker的签名中出现，则直接抛出异常，并打印error logo.如果异常类和接口类在同一个jar包中，则直接抛出异常；如果是JDK中的异常，则直接抛出；如果是Dubbo中定义的异常，则直接抛出。
- (3)处理在步骤(2)中无法处理的异常。把异常转换为字符串，并包装成一个RuntimeException放入RpcResult中返回。

TimeoutFilter 的实现原理

TimeoutFilter主要是日志类型的过滤器，它会记录每个Invoker的调用时间，如果超过了接口设置的timeout值，则会打印一条警告日志，并不会干扰业务的正常运行

代码清单 10-7 TimeoutFilter 源码

```
long start = System.currentTimeMillis(); ← 获取开始时间
Result result = invoker.invoke(invocation); ← 继续调用
long elapsed = System.currentTimeMillis() - start; ← 获取调用持续时间
if (invoker.getUrl() != null ← 如果超时了
    && elapsed > invoker.getUrl().getMethodParameter(invocation.getMethodName(),
        "timeout", Integer.MAX_VALUE)) {
    if (logger.isWarnEnabled()) {
        ...
    }
}
return result;
```

打印一条警告日志

TokenFilter 的实现原理

在Dubbo中，如果某些服务提供者不想让消费者绕过注册中心直连自己，则可以使用令牌验证。总体的工作原理是，服务提供者在发布自己的服务时会生成令牌，与服务一起注册到注册中心。消费者必须通过注册中心才能获取有令牌的服务提供者的URL。TokenFilter是在服务提供者端生效的过滤器，它的工作就是对请求的令牌做校验。我们首先来看一下开启令牌校验的配置方式

```
<!--随机Token令牌，使用UUID生成 全局设置开启令牌验证-->
<dubbo:provider interface="com.foo.BarService" token="true" />
<!--固定Token令牌，相当于密码-->
<dubbo:provider interface="com.foo.BarService" token="123456" />

<!--随机Token令牌，使用UUID生成 服务级别设置-->
<dubbo:service interface="com.foo.BarService" token="true" />
<!--固定Token令牌，相当于密码-->
<dubbo:service interface="com.foo.BarService" token="123456" />

<!--随机Token令牌，使用UUID生成 协议级别设置-->
<dubbo:protocol name="dubbo" token="true" />
<!--固定Token令牌，相当于密码-->
<dubbo:protocol name="dubbo" token="123456" />
```

我们来看一下整个令牌的工作流程：

- (1) 消费者从注册中心获得服务提供者包含令牌的URL
- (2) 消费者RPC调用设置令牌。具体是在RpcInvocation的构造方法中，把服务提供者的令牌设置到附件(attachments)中一起请求服务提供者。
- (3) 服务提供者认证令牌。

这就是TokenFilter所做的工作。

TokenFilter的工作原理很简单，收到请求后，首先检查这个暴露出去的服务是否有令牌信息。如果有，则获取请求中的令牌，如果和接口的令牌匹配，则认证通过，否则认证失败并抛出异常。

TpsLimitFilter 的实现原理

TpsLimitFilter主要用于服务提供者端的限流。我们会发现在org.apache.dubbo.rpc.Filter这个SPI配置文件中，并没有TpsLimitFilter的配置，因此如果需要使用，则用户要自己添加对应的配置。

TpsLimitFilter的限流是基于令牌的，即一个时间段内只分配N个令牌，每个请求过来都会消耗一个令牌，耗完即止，后面再来的请求都会被拒绝。限流对象的维度支持分组、版本和接口级别，默认通过interface + group + version作为唯一标识来判断是否超过最大值。我们先看一下如何配置：

```
<!--每次发放 1000个令牌，令牌刷新的间隔是1秒，如果不配置，则默认是60秒|-->
<dubbo:parameter key="tps" value="1000" />
<dubbo:parameter key="tps.interval" value="1000" />
```

具体的实现逻辑主要关注DefaultTPSLimiter#isAllowable，会用这个方法判断是否触发限流，如果不触发就直接通过了。isAllowable方法的逻辑如下：

- (1) 获取URL中的参数，包含每次发放的令牌数、令牌刷新时间间隔。
- (2) 如果设置了每次发放的令牌数则开始限流校验。DefaultTPSLimiter内部用一个ConcurrentMap缓存每个接口的令牌数，key是interface + group + version，value是一个Statitem对象，它包装了令牌刷新时间间隔、每次发放的令牌数等属性。首先判断上次发放令牌的时间点到现在是否超过时间间隔了，如果超过了就重新发放令牌，之前没用完的不会叠加，而是直接覆盖。然后，通过CAS的方式-1令牌，减掉后令牌数如果小于0则会触发限流。

消费者过滤器的实现原理

ActiveLimitFilter 的实现原理

ActiveLimitFilter是消费者端的过滤器，限制的是客户端的并发数。官方文档中使用的配置如下：

```
<!--限制com.foo.BarService的每个方法在每个客户端的并发执行数（或占用连接的请求数）不能超过10个-->
<dubbo:service interface="com.foo.BarService" actives="10" />
<dubbo:reference interface="com.foo.BarService" actives="10" />
<!--限制 com.foo.BarService 的 sayHello方法在每个客户端的并发执行数（或占用连接的请求数）不能超过10个-->
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10"/>
</dubbo:service>
```

如果dubbo:service和dubbo:reference都配了actives，则dubbo:reference优先。如果设置了actives小于等于0，则不做并发限制。

下面我们看一下ActiveLimitFilter的具体实现逻辑：

- (1) 获取参数。获取方法名、最大并发数等参数，为下面的逻辑做准备。
- (2) 如果达到限流阈值，和服务提供者端的逻辑不一样，并不是直接抛出异常，而是先等待直到超时，因为请求是有timeout属性的。当并发数达到阈值时，会先加锁抢占当前接口的RpcStatus对象，然后通过wait方法进行等待。此时会有两种结果，第一种是某个Invoker在调用结束后，并发把计数器原子-1并触发一个notify，会有一个在wait状态的线程被唤醒并继续执行逻辑。第二种是wait等待超时都没有被唤醒，此时直接抛出异常，如代码清单10 9所示。

代码清单 10-9 ActiveLimitFilter 源码

```
if (max > 0) {
    long timeout = invoker.getUrl().getMethodParameter(invocation.getMethodName(),
    Constants.TIMEOUT_KEY, 0);           ← 获取超时时间
    long start = System.currentTimeMillis();
    long remain = timeout;
    int active = count.getActive();   ← 获取当前并发数
    if (active >= max) {
        synchronized (count) {
            while ((active = count.getActive()) >= max) { ← 加锁，并循环获取当前并发数，如果大于限流阈值则等待
                try {
                    count.wait(remain);
                } catch (InterruptedException e) {
                }
                long elapsed = System.currentTimeMillis() - start;
                remain = timeout - elapsed; 当被 notify 唤醒后，会先判断是否已经超时，然后继续执行 while 循环判断是否已经低于限流阈值
                if (remain <= 0) {           继续执行 while 循环判断是否已经低于限流阈值
                    //
                    ...
                }   ← 超时，抛出异常
            }
        }
    }
}   ← 当前并发数低于限流阈值，则会从上面的 while 循环跳出并来到这里
try {   ←
    long begin = System.currentTimeMillis();
    RpcStatus.beginCount(url, methodName); ← 并发计数器原子+1
    try {
        Result result = invoker.invoke(invocation); ← 执行 Invoker 调用

```

242 | 深入理解 Apache Dubbo 与实战

```
        RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, true); ← 调用结束，并发计数器原子-1
        return result;
    } catch (RuntimeException t) {
        ...
    }
} finally {
    if (max > 0) {           ← 当前请求已经结束，通过 notify 唤醒另外一个线程
        synchronized (count) {
            count.notify();
        }
    }
}
```

(3)如果满足调用阈值，则直接进行调用，成功或失败都会原子-1对应并发计数。最后会唤醒一个等待中的线程。

ConsumerContextFilter 的实现原理

ConsumerContextFilter会和ContextFilter配合使用。因为在微服务环境中，有很多链式调用，如A-B-C-D。收到请求时，当前节点可以被看作一个服务提供者，由ContextFilter设置上下文。当发起请求到下一个服务时，当前服务变为一个消费者，由ConsumerContextFilter设置上下文。其工作逻辑主要如下：

- (1) 设置当前请求上下文，如Invoker信息、地址信息、端口信息等。
- (2) 服务调用。清除Response ±下文，然后继续调用过滤器链的下一个节点。
- (3) 清除上下文信息。每次服务调用完成，都会把附件上下文清除，如隐式传参。

DeprecatedFilter 的实现原理

DeprecatedFilter会检查所有调用，如果方法已经通过dubbo:parameter设置了 deprecated=true，则会打印一段ERROR级别的日志。这个错误日志只会打印一次，判断是否打印过的key维度是：接口名+方法名。打印过的key都会被加入一个Set中保存，后续就不会再打印了，

FutureFilter 的实现原理

FutureFilter主要实现框架在调用前后出现异常时，触发调用用户配置的回调方法，如以下配置：

```
<!--用户编写的回调方法，里面有onreturn、 onthrow、 oninvoke几个方法-->
<bean id="callback" class="com.test.callback"/>
<!--在 testMethod 的调用前后出现异常时， 分别调用回调类的方法-->
<dubbo:reference id="testService" interface="com.testService">
    <dubbo:method name="testMethod" onreturn="callback.onreturn"
onthrow="callback.onthrow" oninvoke ="callback.oninvoke"/>
</dubbo:reference>
```

由于整个逻辑是在过滤器链中执行的，FutureFilter在执行下一个节点的invoke方法前调用oninvoke回调方法就能实现调用前的回调。方法在服务引用初始化的时候就会把配置文件中的回调方法保存到ConsumerMethodManager中，后续使用的时候直接取出来就可以调用。不过需要注意的是，oninvoke回调只会对异步调用有效。

当调用有返回结果的时候，会执行FutureFilter#onResponse的逻辑。对于同步调用的方法，则直接判断返回的result是否有异常，有异常则同步调用onthrow回调方法，没有异常则同步调用 onreturn 回调方法。对于异步调用，会通过CompletableFuture的thenApply方法来执行onthrow 或 onretum 的回调。

Dubbo高级特性

目前Dubbo框架在支持RPC通信的基础上，提供了大量的高级特性，比如服务端Telnet调用、 Telnet调用统计、服务版本和分组隔离、隐式参数、异步调用、泛化调用、上下文信息和结果缓存等特性。

表 9-1 Dubbo 支持的高级特性

特 性	作 用
服务分组和版本	支持同一个服务有多个分组和多个版本实现，用于服务强隔离、服务多个版本实现
参数回调	当消费方调用服务提供方时，支持服务提供方能够异步回调到当前消费方，用于 stub 做热数据缓存等
隐式参数	支持客户端隐式传递参数到服务端
异步调用	并行发起多个请求，但只使用一个线程，用于业务请求非阻塞场景
泛化调用	泛化调用主要用于消费端没有 API 接口的情况。不需要引入接口 jar 包，而是直接通过 GenericService 接口来发起服务调用。框架会自动把 POJO 对象转为 Map，只要参数名能对应上即可。适合网关和跨框架集成等场景
上下文信息	上下文中存放的是当前调用过程中所需的环境信息
Telnet 操作	支持服务端调用、状态检查和跟踪服务调用统计等
Mock 调用	用于方法调用失败时，构造 Mock 测试数据并返回
结果缓存	结果缓存，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量

服务分组和版本

Dubbo中提供的服务分组和版本是强隔离的，如果服务指定了服务分组和版本，则消费方调用也必须传递相同的分组名称和版本名称。

假设我们有订单查询接口 com.alibaba.pay.order.QueryService,这个接口包含不同的版本实现，比如版本分别为1.0.0-stable和2.0.0,在服务端对应的实现名称分别为com.alibaba.pay.order.StableQueryService 和 com.alibaba.pay.order.PerfomanceQueryService 我们可以在服务暴露时指定配置，如代码清单9-1所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://dubbo.apache.org/schema/dubbo
        http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:service interface="com.alibaba.pay.order.QueryService"
        class="com.alibaba.pay.order.StableQueryService" version="1.0.0-stable"/>
    <dubbo:service interface="com.alibaba.pay.order.QueryService"
        class="com.alibaba.pay.order.PerfomanceQueryService" version="2.0.0"/>
    <!--省略其他Dubbo配置-->
</beans>
```

在代码清单9.1中发现，服务暴露直接配置version属性即可，如果要为服务指定分组，则继续添加group属性即可。因为这个特性是强隔离的，消费方必须在配置文件中指定消费的版本。如果消费方式为泛化调用或注解引用，那么也需要指定对应的相同名称的版本号，如代码清单9-2所示。

```
代码清单9.2 消费方指定版本
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://dubbo.apache.org/schema/dubbo
        http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:reference interface="com.alibaba.pay.order.QueryService" version="1.0.0-stable"/>
    <dubbo:reference interface="com.alibaba.pay.order.QueryService"
        version="2.0.0"/>
    <!--省略其他Dubbo配置-->
</beans>
```

在消费方〈dubbo:reference〉标签中指定要消费的版本号时，在服务拉取时会在客户端做一次过滤。如果要消费指定的分组，那么还需要指定group属性。当服务提供方进行服务暴露时，服务端会根据serviceGroup/serviceName:serviceversion:port组合成key,然后服务实现作为value保存在DubboProtocol类的exporterMap字段中。这个字段是一个HashMap对象，当服务消费调用时，根据消费方传递的服务分组、服务接口、版本号和服务暴露时的协议端口号重新构造这个key,然后从内存Map中查找对应实例进行调用。

当客户端指定了分组和版本时，在DubboInvoker构造函数中会将URL中包含的接口、分组、Token和timeout加入attachment，同时将接口上的版本号存储在version字段。当发起RPC请求时，通过DubboCodec把这些信息发送到服务器端，服务器端收到这些关键信息后重新组装成key，然后查找业务实现并调用。

当Dubbo客户端启动时，实际上会把调用接口所有的协议节点都拉取下来，然后根据本地URL配置的接口、category、分组和版本做过滤，具体过滤是在注册中心层面实现的。以ZooKeeper注册中心为例，当注册中心推送列表时，会调用ZookeeperRegistry#toUrlsWithoutEmpty方法，这个方法会把所有服务列表进行一次过滤

代码清单 9-3 过滤服务分组和版本

```
private List<URL> toUrlsWithoutEmpty(URL consumer, List<String> providers) {  
    List<URL> urls = new ArrayList<URL>();  
    if (providers != null && !providers.isEmpty()) {  
        for (String provider : providers) {  
            provider = URL.decode(provider); ① 遍历所有的服务列表  
并解码特殊字符  
            if (provider.contains("://")) {  
                URL url = URL.valueOf(provider);  
                if (UrlUtils.isMatch(consumer, url)) { ② 根据接口、category、  
版本和分组过滤  
                    urls.add(url);  
                }  
            }  
        }  
    }  
    return urls;  
}
```

Dubbo中接收服务列表是在RegistryDirectory中完成的，它收到的列表是全量的列表。

RegistryDirectory主要将URL转换成可以调用的Invokers，在获取列表前会经过①把服务列表解码，用于解码被转译的字符。消费指定分组和版本关键逻辑在②中，它会将特定接口的全量列表和消费方URL进行匹配，匹配规则是校验接口名、类别、版本和分组是否一致。消费方默认的类别是providers

参数回调

Dubbo支持异步参数回调，当消费方调用服务端方法时，允许服务端在某个时间点回调回客户端的方法。在服务端回调到客户端时，服务端不会重新开启TCP连接，会复用已经建立的从客户端到服务端的TCP连接

代码清单 9-4 异步回调服务端实现

```
public interface CallbackService { ←① 服务提供方暴露的接口
    void addListener(String key, CallbackListener listener);
}

public interface CallbackListener { ←② 消费方被回调的方法
    void changed(String msg);
}

public class CallbackServiceImpl implements CallbackService { ←③ 服务提供方接口实现
    private final Map<String, CallbackListener> listeners = new
    ConcurrentHashMap<String, CallbackListener>();

    public void addListener(String key, CallbackListener listener) { ←④ 服务提供方接口实现
        listeners.put(key, listener);
    }

    public CallbackServiceImpl() {
        Thread t = new Thread(new Runnable() {
            public void run() {
                while(true) {
                    try {
                        for(Map.Entry<String, CallbackListener> entry : listeners.entrySet()){
                            try {
                                entry.getValue().changed(getChanged(entry.getKey())); ←⑤ 服务端定时每5秒回调客户端一次
                            } catch (Throwable t) {
                                listeners.remove(entry.getKey());
                            }
                        }
                        Thread.sleep(5000);
                    } catch (Throwable ignored) {
                    }
                }
            });
        });
        t.start();
    }

    private String getChanged(String key) {
        return "Changed: " + new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date());
    }
}
```

要实现异步参数回调，我们首先定义一个服务提供者接口，这里举例为CallbackService,注意其方法的第2个参数是接口，被回调的参数顺序不重要。第2个参数代表我们想回调的客户端CallbackListener接口，具体什么时候回调和调用哪个方法是由服务提供方决定的。对应到代码清单9-4中，①是我们定义的服务提供方服务接口，定义了addListener方法，用于给客户端调用。②定义了客户端回调接口，这个接口实现在客户端完成。③对应普通Dubbo服务实现。当客户端调用addListener方法时，会将客户端回调实例加入listeners,用于服务端定时回调客户端。服务提供者在初始化时会开启一个线程，它轮询检查是否有回调加入，如果有则每隔5秒回调客户端。在⑤中每隔5秒处理多个回调方法。

当服务提供方完成后，我们需要编写消费方代码，用于调用服务提供者addListener方法，把客户端加入回调列表，如代码清单9-5所示。

代码清单 9-5 消费异步回调服务

```
ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("classpath:consumer.xml");  
context.start();  
  
CallbackService callbackService = (CallbackService)  
context.getBean("callbackService");
```

216 | 深入理解 Apache Dubbo 与实战

```
callbackService.addListener("foo.bar", new CallbackListener(){      ←  
    public void changed(String msg) {          ① 调用普通服务提供者,  
        System.out.println("callback1:" + msg);      同时指定回调实现  
    }  
});
```

客户端调用也是非常简单的，主要是调用服务提供者服务并把自己加入回调列表，同时指定 key 和对应的回调方法。①会获取 Spring 的消费配置<dubbo:reference ...>实例，调用 CallbackService#addListener，然后创建接口匿名类实现。

在服务暴露和消费代码写完后，接下来我们需要做适当配置，告诉 Dubbo 框架哪个参数是异步回调，如代码清单 9-6 所示。

```
<!--服务提供方配置-->  
<dubbo:service interface="CallbackService" class="CallbackServiceImpl"  
connections="1" callbacks="1000">  
  <!--指定addListener方法的第2个参数是回调方法-->  
  <dubbo:method name="addListener">  
    <dubbo:argument index="1" callback="true" />  
  </dubbo:method>  
</dubbo:service>  
<!--服务消费方配置-->  
<dubbo:reference id="callbackService" interface="CallbackService" />
```

实现异步回调的原理比较容易理解，客户端在启动时，会拉取服务CallbackService元数据，因为服务端配置了异步回调信息，这些信息会透传给客户端。客户端在编码请求时，会发现第2个方法参数为回调对象。此时，客户端会暴露一个Dubbo协议的服务，服务暴露的端口号是本地TCP连接自动生成的端口。在客户端暴露服务时，会将客户端回调参数对象内存id存储在attachment中，对应的key为sys_callback_arg-回调参数索引。这个key在调用普通服务addListener时会传递给服务端，服务端回调客户端时，会把这个key对应的值再次放到attachment中传给客户端。从服务端回调到客户端的attachment会用keycallback.service.instdid保存回调参数实例id,用于查找客户端暴露的服务

客户端调用服务端方法时，并不会把第2个异步参数实例序列化并传给服务端。当服务端解码时，会先检查参数是不是异步回调参数。如果发现是异步参数回调，那么在服务端解码参数值时，会自动创建到消费方的代理。服务端创建回调代理实例Invoker类型是ChannelWrappedInvoker，比较特殊的是，构造函数的service值是客户端暴露对象id，当回调发生时，会把keycallback, service, instid保存的对象id传给客户端，这样就能正确地找到客户端暴露的服务了。

隐式参数

Dubbo 服务提供者或消费者启动时，配置元数据会生成 URL，一般是不可变的。在很多实际的使用场景中，在服务运行期需要动态改变属性值，在做动态路由和灰度发布场景中需要这个特性。Dubbo 框架支持消费方在 `RpcContext#setAttachment` 方法中设置隐式参数，在服务端 `RpcContext#getAttachment` 方法中获取隐式传递。

当客户端发起调用前，设置隐藏参数，框架会在拦截器中把当前线程隐藏参数传递到 `RpcInvocation` 的 `attachment` 中，服务端在拦截器中提取隐藏参数并设置到当前线程 `RpcContext` 中。隐式传参的详细原理如图 9-1 所示。

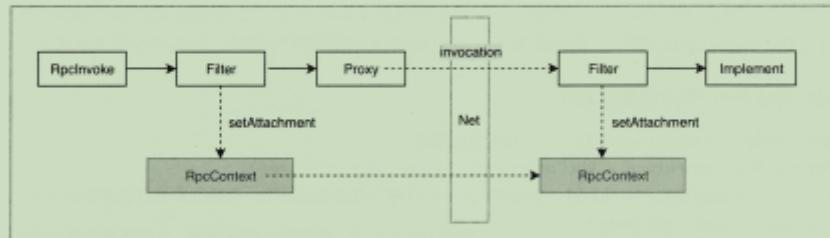


图 9-1 隐式传参的详细原理

通过 API 的方式设置参数和提取参数，如代码清单 9-7 所示。

代码清单 9-7 隐式传参使用

```
RpcContext.getContext().setAttachment("index", "1"); ← 在客户端设置隐式传参  
xxxService.xxx(); <-- 后面的远程调用都会隐式将这些参数发送到服务器端  
public class XxxServiceImpl implements XxxService { <-- 在服务端获取隐式参数  
    public void xxx() {  
        String index = RpcContext.getContext().getAttachment("index"); <-- 获取客户端隐式传入的参数  
    }  
}
```

```
public void xxx() {  
    String index = RpcContext.getContext().getAttachment("index"); <--  
}
```

在消费方调用服务方传递隐式参数时，会在 `AbstractInvoker#invoke` 方法调用中合并 `RpcContext#getAttachments()` 参数。用户的隐式参数会被合并到 `RpcInvocation` 中的 `attachment` 字段，这个字段发送给服务端。在服务提供方收到请求时，在 `ContextFilter#invoke` 中提取 `RpcInvocation` 中的 `attachment` 信息，并设置到当前线程上下文中。因为后端业务方法调用和拦截器在同一个线程中执行，所以直接使用 `RpcContext.getContext().getAttachment` 获取值即可。在图 9-1 中会发现客户端在拦截器中（`ConsumerContextFilter`）执行 `setAttachments` 方法，这个主要支持服务端透传隐式参数给客户端。

异步调用

本节主要聚焦 Dubbo 在客户端支持异步调用方面的内容，在编写本书时，Dubbo 还未支持服务端异步调用，2.7.0+ 版本才在服务端支持异步调用。在客户端实现异步调用非常简单，在消费接口时配置异步标识，在调用时从上下文中获取 Future 对象，在期望结果返回时再调用阻塞方法 Future.get() 即可。我们给出了在客户端实现异步调用的实例，如代码清单 9-8 所示。

代码清单 9-8 客户端异步调用

```
fooService.findFoo(fooId); // 触发异步调用
Future<Foo> fooFuture = RpcContext.getContext().getFuture();
// 在发起其他 RPC 调用时，先获取 Future 引用，当结果返回后，会被通知和设置到此 Future
Foo foo = fooFuture.get(); // 如果 foo 已返回，则直接获取返回值，否则当前线程会被阻塞并等待
// ... 客户端非阻塞处理其他任务
```

通过代码清单 9-8，我们知道在客户端发起异步调用时，应该在保存当前调用的 Future 后，再发起其他远程调用，否则前一次异步调用的结果可能丢失（异步 Future 对象会被上下文覆盖）。因为框架要明确知道用户意图，所以需要再明确开启使用异步特性，在 <dubbo:reference ...> 标签中指定 async 标记，如代码清单 9-9 所示。

代码清单 9-9 消费方配置异步标识

```
<!-- 省略其他消费方配置 -->
<dubbo:reference id="fooService" interface="com.alibaba.foo.FooService"
    async="true"/>
```

Dubbo 异步调用流程如图 9-2 所示。

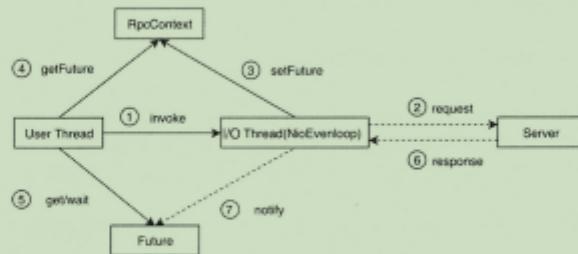


图 9-2 Dubbo 异步调用流程

站在 Dubbo 客户端角度来说，直接发起 RPC 调用端属于用户线程。用户线程①发起任意远程方法调用，最终会通过 I/O 线程发送网络报文。在真实发送报文前会在用户线程中设置当前异步请求 Future（③）。因此在用户线程发起下一个远程方法调用前，需要先保存异步 Future 对象（④）。Dubbo 框架会把异步请求对象保存在 DefaultFuture 类中，当服务端响应或超时时，被挂起的用户线程将被唤醒（⑤）。用户线程设置异步 Future 对象的逻辑在 DubboInvoker#doInvoke 方法中完成，感兴趣的读者可以参阅 DubboInvoker 中对应的源码实现。

泛化调用

Dubbo 泛化调用特性可以在不依赖服务接口 API 包的场景中发起远程调用。这种特性特别适合框架集成和网关类应用开发。Dubbo 在客户端发起泛化调用并不要求服务端是泛化暴露。假设我们调用服务端 com.xxx.XxxService#sayHello 方法，可以实现如代码清单方法 9-10 所示的方法。

代码清单 9-10 泛化调用示例

```
ReferenceConfig<GenericService> ref = new ReferenceConfig<>();
ApplicationConfig appConfig = new ApplicationConfig("demo-consumer");

RegistryConfig registryConfig = new RegistryConfig();
registryConfig.setProtocol("zookeeper");
registryConfig.setAddress("localhost:2181");

ref.setProtocol("dubbo");
```

220 | 深入理解 Apache Dubbo 与实战

```
ref.setApplication(appConfig);
ref.setRegistry(registryConfig);
ref.setInterface("com.xxx.XxxService");

ref.setGeneric(true); <— ① 标识泛化调用

GenericService genericService = ref.get(); <— ② 创建远程代理
Object result = genericService.$invoke("sayHello", new String[]
{"java.lang.String"}, new Object[] {"world"}); <— ③ 发起远程调用
```

目前泛化调用必需的参数主要包括应用名称、注册中心（或者是直连调用地址）、真实接口名称和泛化标识。在发起远程服务调用时，`GenericService` 方法参数类型分别为真实方法名、真实方法参数类型签名和真实参数值。这里有一个注意事项，每次动态创建的 `GenericService` 实例比较重，需要建立 TCP 连接，处理注册中心订阅和服务列表等计算，因此需要缓存 `ReferenceConfig` 对象进行复用。但是往往很多业务开发时，忘记设置 `ReferenceConfig` 对象的 `Check` 方法为 `false`，导致在没有服务提供者时，触发框架抛出 `No provider available` 的异常，从而导致缓存命中失败。

其实泛化的实现原理相对比较好理解，服务端在处理服务调用时，在 `GenericFilter` 拦截器中先把 `RpcInvocation` 中传递过来的参数类型和参数值提取出来，然后根据传递过来的接口名、方法名和参数类型查找服务端被调用的方法。获取真实方法后，主要提取真实方法参数类型（可能包含泛化类型），然后将参数值做 Java 类型转换。最后用解析后的参数值构造新的 `DynamicInvocation` 对象发起调用。

上下文信息

Dubbo 上下文信息的获取和存储同样是基于 JDK 的 ThreadLocal 实现的。上下文中存放的是当前调用过程中所需的环境信息。RpcContext 是一个 ThreadLocal 的临时状态记录器，当收到或发送 RPC 时，当前线程关联的 RpcContext 状态都会变化。比如：A 调用 B，B 再调用 C，则在 B 机器上，在 B 调用 C 之前，RpcContext 记录的是 A 调用 B 的信息，在 B 调用 C 之后，RpcContext 记录的是 B 调用 C 的信息。

假设在服务端有 DemoServiceImpl 实现，在代码清单 9-11 中展示了上下文使用实例：

代码清单 9-11 服务端上下文的获取和使用

```
public class DemoServiceImpl implements DemoService {  
  
    public void hello() {
```

```
        boolean isProviderSide = RpcContext.getContext().isProviderSide();      ←  
        // 本端是否为提供端，这里会返回 true  
        String clientIP = RpcContext.getContext().getRemoteHost(); ←  
            获取远程客户端 IP 地址  
        String application = RpcContext.getContext().getUrl().getParameter  
("application"); ← 获取当前服务配置信息，所有配置信息都将转换为 URL 的参数  
            | 注意：每发起 RPC 调用，上下文状态会变化  
            | 这里假设调用 yyyService 服务 done 方法  
        yyyService.done(); ←  
        boolean isProviderSide = RpcContext.getContext().isProviderSide(); ←  
    }  此时本端变成消费端，这里会返回 false  
}
```

在客户端和服务端分别有一个拦截设置当前上下文信息，对应的分别为 ConsumerContextFilter 和 ContextFilter。在客户端拦截器实现中，因为 Invoker 包含远程服务信息，因此直接设置远程 IP 等信息。在服务端拦截器中主要设置本地地址，这个时候无法获取远程调用地址。设置远程地址主要在 DubboProtocol#ExchangeHandlerAdapter.reply 方法中完成，可以直接通过 channel.getRemoteAddress 方法获取。

结果缓存

Dubbo 框架提供了对服务调用结果进行缓存的特性，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量。因为每次调用都会使用 `JSON.toJSONString` 方法将请求参数转换成字符串，然后拼装唯一的 key，用于缓存唯一键。如果不能接受缓存造成的开销，则谨慎使用这个特性。

如果要使用缓存，则可以在消费方添加如下配置：

```
<dubbo:reference cache="lru" .../>
```

`lru` 缓存策略是框架默认使用的，因此我们会对它进行简单的说明。它的原理比较简单，缓存对应实现类是 `LRUCache`。缓存实现类 `LRUCache` 继承了 JDK 的 `LinkedHashMap` 类，`LinkedHashMap` 是基于链表的实现，它提供了钩子方法 `removeEldestEntry`，它的返回值用于判断每次向集合中添加元素时是否应该删除最少访问的元素。`LRUCache` 重写了这个方法，当缓存值达到 1000 时，这个方法会返回 `true`，链表会把头部节点移除。链表每次添加数据时都会在队列尾部添加，因此队列头部就是最少访问的数据（`LinkedHashMap` 在更新数据时，会把更新数据更新到列表尾部）。

Telnet 操作

目前Dubbo支持通过Telnet登录进行简单的运维，比如查看特定机器暴露了哪些服务、显示服务端口连接列表、跟踪服务调用情况、调用本地服务和服务健康状况等。

当服务发布时，如果注册中心没有对应的服务，那么我们可以初步使用ls命令检查Dubbo服务是否正确暴露了。ls主要提供了查询已经暴露的服务列表、查询服务详细信息和查询指定服务接口信息等功能。ls命令的用法如下：

```
ls options [service]
```

命令说明：

- service代表要查询的服务接口名称，可以是短名称或全名称；
- options代表支持的命令参数；
- -l显示服务详细信息列表或服务方法的详细信息。

表 9-2 ls 命令示例

命 令 示 例	作 用
ls	显示服务列表
ls -l	显示服务详细列表
ls HelloService	显示 HelloService 服务的方法列表
ls -l HelloService	显示 HelloService 服务的方法详细信息列表（包括参数类型和返回值）

ls命令的实现主要基于ListTelnetHandler, Dubbo框架的Telnet调用只对Dubbo协议提供支持。它的原理非常简单，当服务端收到ls命令和参数时，会加载ListTelnetHandler并执行，然后触发 `DubboProtocol.getDubboProtocol().getExporters()`方法获取所有已经暴露的服务，获取暴露的接口名和暴露服务别名(path属性)进行匹配，将匹配的结果进行输出。如果是查看服务暴露的方法，则框架会获取暴露接口名，然后反射获取所有方法并输出

ps命令用于查看提供服务本地端口的连接情况，ps的命令用法如下：

```
ps options [port]
```

命令说明：

- port代表要查询的服务暴露的端口；
- options代表支持的命令参数；
- -l显示服务暴露的所有端口或服务端端口建立连接的信息。

`ps` 命令示例如表 9-3 所示。

表 9-3 `ps` 命令示例

命令示例	作用
<code>ps</code>	显示服务暴露的端口列表
<code>ps -l</code>	显示服务地址列表
<code>ps 20880</code>	显示端口上的连接信息
<code>ps -l 20880</code>	显示端口上的连接详细信息（客户端 IP 和 port，服务端 IP 和 port）

`ps` 命令实现类对应 `PortTelnetHandler` 类，当 Dubbo 服务暴露时，会把关联端口的服务端实例加入 `DubboProtocol` 类的 `serverMap` 字段。当执行 `ps` 命令时，`PortTelnetHandler` 类会通过 `DubboProtocol.getDubboProtocol().getServers()` 提取暴露的 server 实例。它持有了端口号和所有客户端连接信息等。当无法确认命令对应的后端实现时，可以查找和扩展点名称相同的文件，它包含扩展点所有的实现定义，比如 `com.alibaba.dubbo.remoting.telnet.TelnetHandler`。

`trace` 用于统计服务方法的调用信息，比如跟踪服务调用方法返回值、连接信息和耗时等。
`trace` 命令示例如表 9-4 所示。

表 9-4 `trace` 命令示例

命令示例	作用
<code>trace HelloService</code>	跟踪 1 次 HelloService 服务任意方法的调用情况
<code>trace HelloService 10</code>	最多跟踪 10 次 HelloService 服务任意方法的调用情况
<code>trace HelloService echo</code>	跟踪 1 次 HelloService 服务 echo 方法的调用情况
<code>trace HelloService echo 10</code>	最多跟踪 10 次 HelloService 服务 echo 方法的调用情况

`trace service [method] [count]`

命令说明：

- `service` 代表要查询的服务接口名称，可以是短名称或全名称；
- `method` 代表要跟踪的方法；
- `count` 代表跟踪的最大次数。

如果在使用 `trace` 命令跟踪方法调用时指定了最大次数，则不需要重复执行 `trace` 命令，当服务接口方法调用超过了最大次数后，不会把调用结果信息推送给 Telnet 客户端。

`trace` 命令对应的实现类是 `TraceTelnetHandler`，它本身不会执行任何方法调用，首先根据传递的接口和方法查找对应的 `Invoker`，然后把当前的 Telnet 连接（`Channel`）、接口、方法和最大执行次数信息记录在 `TraceFilter` 中，当接口方法被调用时，`TraceFilter` 会取出对应的 Telnet 连接（`Channel`），并把调用结果信息发送给 Telnet 客户端。

`count` 命令也用于统计服务信息，但它主要统计方法调用成功数、失败数、正在并发执行数、平均耗时和最大耗时。如果在服务方暴露服务时配置了 `executes` 属性，那么使用 `count` 命令可以统计并发调用信息。

`count` 命令示例如表 9-5 所示。

表 9-5 `count` 命令示例

命令示例	作用
<code>count HelloService</code>	统计 1 次 HelloService 服务任意方法的调用情况
<code>count HelloService 10</code>	最多统计 10 次 HelloService 服务任意方法的调用情况
<code>count HelloService echo</code>	统计 1 次 HelloService 服务 echo 方法的调用情况
<code>count HelloService echo 10</code>	最多统计 10 次 HelloService 服务 echo 方法的调用情况

`count service [method] [count]`

命令说明：

- `service` 代表要查询的服务接口名称，可以是短名称或全名称；
- `method` 代表要跟踪的方法；
- `count` 代表跟踪的最大次数。

count命令对应的实现类是CountTelnetHandler,每次执行count命令时在服务端会启动一个线程去循环统计当前调用次数。比如统计10次，在线程中每间隔1秒执行一次统计，直到达到统计次数时退出线程。框架会使用RpcStatus类记录并发调用信息，CountTelnetHandler负责提取这些统计信息并输出给Telnet客户端。

Mock 调用

Dubbo提供服务容错的能力，通常用于服务降级，比如验权服务，当服务提供方“挂掉”后，客户端不抛出异常，而是通过Mock数据返回授权失败。

目前Dubbo提供以下几种方式来使用Mock能力：

- (1) <dubbo:reference mock="true" .../>
- (2) <dubbo:reference mock="com.foo.BarServiceMock" . . . />
- (3) <dubbo:reference mock="return null" .../>
- (4) <dubbo:reference mock="throw com.alibaba.XXXException" . . . />
- (5) <dubbo:reference mock="force:return fake" . . . />
- (6) <dubbo:reference mock="force:throw com.foo.MockException" . . . />

当Dubbo服务提供者调用抛出RpcException时，框架会降级到本地Mock伪装。以接口com.foo.BarService为例，第1种和第2种的使用方式是等价的，当直接指定mock=true时，客户端启动时会查找并加载com.foo.BarServiceMock类。查找规则根据接口名加Mock后缀组合成新的实现类，当然也可以使用自己的Mock实现类指定给Mock属性。当在Mock中指定return null时，允许调用失败返回空值。当在Mock中指定throw或throw com.alibaba.XXXException时，框架会抛出即com.Exception和用户自定义异常 com.alibaba.XXXException。2.6.5版本以前（包括当前版本），因为实现有缺陷，在使用方式4、5和6中需要更新后的版本支持。目前默认场景都是在没有服务提供者或调用失败时，触发Mock调用，如果不想发起RPC调用直接使用Mock数据，则需要在配置中指定force:语法（同样需要版本高于2.6.5）。

这些Mock关键逻辑是在哪里处理的呢？处理Mock伪装对应的实现类是MockClusterInvoker，因为MockClusterWrapper是对扩展点Cluster的包装，当框架在加载Cluster扩展点时会自动使用MockClusterWrapper类对Cluster实例进行包装（默认是FailoverCluster或MockClusterInvoker对应的实现如代码清单9-12所示）。

代码清单 9-12 MockClusterInvoker 对应的实现

```
public Result invoke(Invocation invocation) throws RpcException {
    Result result = null;

    String value = directory.getUrl().getMethodParameter(invocation.getMethodName(),
        Constants.MOCK_KEY, Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || value.equalsIgnoreCase("false")) {
        result = this.invoker.invoke(invocation);           ① 如果没有指定 Mock，则不需要本地伪装
    } else if (value.startsWith("force")) {
        if (logger.isWarnEnabled()) {
            logger.info("force-mock: " + invocation.getMethodName() + " force-mock
enabled , url : " + directory.getUrl());
        }
        result = doMockInvoke(invocation, null);           ② Mock 指定了 force，不发起 RPC
    } else {                                              调用，直接本地伪装
        try {
            result = this.invoker.invoke(invocation);           ③ 配置了 Mock，先发起 RPC
        } catch (RpcException e) {                            调用
            if (e.isBiz()) {
                throw e;
            } else {
                if (logger.isWarnEnabled()) {
                    logger.warn("fail-mock: " + invocation.getMethodName() + " fail-mock
enabled , url : " + directory.getUrl(), e);
                }
                result = doMockInvoke(invocation, e);           ④ 调用报错，降级 Mock 伪装
            }
        }
    }
    return result;
}
```

代码清单 9-12 中主要完成服务降级伪装。在①中如果没有配置 Mock，则直接发起 RPC 调用。2.6.5 版本虽然支持 force 特性，但因为有 bug，②中的这段代码实际上并不会执行。在 2.6.5 版本以后，如果用户为 Mock 指定了 force，则直接在本地伪装而不发起 RPC 调用。在③中先处理正常 RPC 调用，如果调用出错则会降级到 Mock 调用。在④中具体 Mock 数据是由开发者自己编码完成的。Dubbo 框架对常用的返回值做了支持，比如接口返回布尔值，可以直接在 Mock 中指定 return true。