

Dubbo 的 RPC 调用过程解析

说明:

该文档分析了 Dubbo 框架中 RPC 调用的整个流程, 并基于源代码按照执行时序进行说明。

涉及的关键点包括: Directory、路由、负责均衡、集群容错、过滤器 Filter 以及监控模块等。

可以作为 Dubbo 框架二次开发的参考。

RPC 调用在客户端(即 Consumer 端)触发, 其配置文件 applicationContext.xml 文件中会有如下的定义:

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

这一行定义会为服务接口 xxx.xxx.Service 在本地生成一个远程代理, 在 Dubbo 中这个代理用 com.alibaba.dubbo.common.bytecode.proxy0 的实例来表示, 另外, 由于这个代理存在于本地, 因此就可以像本地 bean 一样调用该服务, 具体的通信过程由代理负责。

这个代理实例中仅仅包含一个 handler 对象(InvokerInvocationHandler 类的实例), handler 中则包含了 RPC 调用中非常核心的一个接口 Invoker<T>的实现, Invoker 接口的定义如下:

```
public interface Invoker<T> extends Node {  
    Class<T> getInterface();  
    //调用过程的具体表示形式  
    Result invoke(Invocation invocation) throws RpcException;  
}
```

Invoker<T>接口的核心方法是 invoke(Invocation invocation), 方法的参数 Invocation 是一个调用过程的抽象, 也是 Dubbo 框架的核心接口, 该接口中包含如何获取调用方法的名称、参数类型列表、参数列表以及绑定的数据, 定义代码如下:

```
public interface Invocation {  
    //调用的方法名称  
    String getMethodName();  
    //调用方法的参数的类型列表  
    Class<?>[] getParameterTypes();  
    //调用方法的参数列表  
    Object[] getArguments();  
    //调用时附加的数据, 用 map 存储  
    Map<String, String> getAttachments();  
    //根据 key 来获取附加的数据  
    String getAttachment(String key);  
    //getAttachment(String key)的拓展, 支持默认值获取  
    String getAttachment(String key, String defaultValue);  
    //获取真实的调用者实现  
    Invoker<?> getInvoker();  
}
```

Invocation 接口和 Invoker<T>接口是配套使用的, 二者相互依存, 从 Invoker<T>的调用方法定义 invoke(Invocation invocation)就可以看出这一点。如果是 RPC 调用时, Invocation 的具体实现就是 RPCInvocation, 该方法会抛出 RpcException, 该异常意味着调用失败,

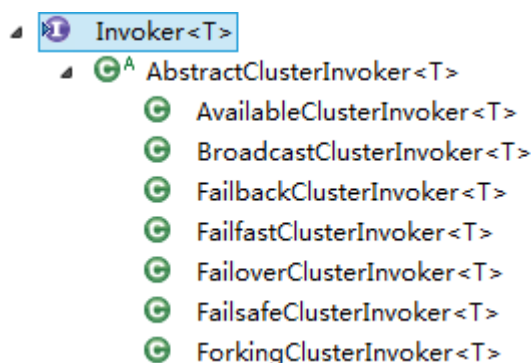
代理中的 handler 实例中包含的 `Invoker<T>` 接口实现者是 `MockClusterInvoker`，其中 `MockClusterInvoker` 仅仅是一个 `Invoker` 的包装，并且也实现了接口 `Invoker<T>`，其只是用于实现 Dubbo 框架中的 mock 功能，我们可以从他的 `invoke` 方法的实现中看出，代码如下：

```
public Result invoke(Invocation invocation) throws RpcException {
    Result result = null;
    /*这一行代码用于获取该服务是否提供 mock 功能，如果提供，则 url 中会包含 mock 关键字*/
    String value = directory.getUrl().getMethodParameter(invocation.getMethodName(), Constants.MOCK_KEY, Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || value.equalsIgnoreCase("false")){
        //没有 mock 的过程，直接调用
        result = this.invoker.invoke(invocation);
    } else if (value.startsWith("force")) {
        //日志记录代码部分省略
        //force:direct mock，这里用于处理强制 mock 的情况，不执行远程调用
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock，这里处理失败后 mock 的情况，即出现调用异常时执行 mock
        try {
            result = this.invoker.invoke(invocation);
        } catch (RpcException e) {
            if (e.isBiz()) {
                throw e;
            } else {
                //省略日志记录部分的代码
                //这里执行 mockInvoke 的逻辑，在本地伪装结果
                result = doMockInvoke(invocation, e);
            }
        }
    }
    return result;
}
```

需要注意的是，`MockClusterInvoker` 实现了 `Invoker<T>` 接口，但它没有真正地实现 `invoke` 的逻辑，只是包装了一个 `Invoker` 的真实实现，从 `MockClusterInvoker` 的定义代码部分就可以看出，代码如下：

```
public class MockClusterInvoker<T> implements Invoker<T>{
    private final Directory<T> directory;
    private final Invoker<T> invoker;
    public MockClusterInvoker(Directory<T> directory, Invoker<T> invoker) {
        this.directory = directory;
        this.invoker = invoker;
    }
}
```

其中真实的 `Invoker` 实现包含了集群容错的功能，因此可以得知 Dubbo 的集群容错是在 `Invoker` 中实现的，`Invoker` 接口的实现者包括如下：



这些实现者都是 Dubbo 推荐的几种容错实现，可以在配置文件中进行选择，默认的是 FailoverClusterInvoker 实现，即失败重试。

现在，进入 FailoverClusterInvoker 的实现中，可以发现其核心方法 invoke(Invocation invocation) 是继承自 AbstractClusterInvoker<T> 抽象类。AbstractClusterInvoker<T> 中的 invoke(Invocation invocation) 方法的实现代码如下：

```
public Result invoke(final Invocation invocation) throws RpcException {
    checkWheatherDestoried();
    LoadBalance loadbalance;
    List<Invoker<T>> invokers = list(invocation);
    if (invokers != null && invokers.size() > 0) {
//这里省略了获取 loadbalance 实例的实现代码，具体思路就是用拓展类加载器根据配置加载 loadbalance 实例
    } else {
        loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).
            getExtension(Constants.DEFAULT_LOADBALANCE);
    }
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    return doInvoke(invocation, invokers, loadbalance);
}
```

调用过程中首先通过方法 list(invocation) 获取 Invoker 的列表 List<Invoker<T>>，这个列表可以对应到服务提供者的列表。其中，list(invocation) 方法的实现代码如下：

```
protected List<Invoker<T>> list(Invocation invocation) throws RpcException {
    List<Invoker<T>> invokers = directory.list(invocation);
    return invokers;
}
```

AbstractClusterInvoker 的 list(invocation) 方法中会用到另一个比较关键的接口 Directory，它是 Dubbo 框架中用于封装服务提供者列表的一个数据结构。Directory 的定义代码如下：

```
public interface Directory<T> extends Node {
    Class<T> getInterface();
    List<Invoker<T>> list(Invocation invocation) throws RpcException;
}
```

Directory 接口的核心方法是 list(Invocation invocation)，该方法的参数是一个 Invocation 对象，该 Invocation 对象表示的是一次（远程）调用过程，在上面已经说明过了，其中包含了调用的方法名称、参数类型列表、参数列表以及附加的数据集合，list 方法返回一个调用者的列表 List<Invoker<T>>，其中 directory 接口的实例实例是 RegistryDirectory 类的对象，RegistryDirectory 的 list(invocation) 方法继承自 AbstractDirectory，具体实现如下：

```

public List<Invoker<T>> list(Invocation invocation) throws RpcException {
    if (destroyed)
        throw new RpcException("Directory already destroyed .url: " + getUrl());
    List<Invoker<T>> invokers = doList(invocation);
    List<Router> localRouters = this.routers; // local reference
    if (localRouters != null && localRouters.size() > 0) {
        for (Router router: localRouters){
            try {
                if (router.getUrl() == null || router.getUrl().
                    getParameter(Constants.RUNTIME_KEY, true)) {
                    invokers = router.route(invokers, getConsumerUrl(), invocation);
                }
            } catch (Throwable t) {
                logger.error("Failed to execute router: " + getUrl() + ",
                    cause: " + t.getMessage(), t);
            }
        }
    }
    return invokers;
}

```

可以看出，AbstractDirectory 类的 list(invocation)方法首先会获取当前 Invocation 的调用列表 List<Invoker<T>>，由抽象方法 doList(invocation)实现，具体的实现放在了 AbstractDirectory 的子类中。具体可以参考 com.alibaba.dubbo.registry.integration 包下的 RegistryDirectory 的 doList(invocation) 方法实现，代码就不在这里列出了。

接下来就是对获取到的当前 Invocation 的调用列表 List<Invoker<T>>进行路由处理了，每个 Consumer 会在本地缓存（或者从注册中心获取）路由集合 List<Router> localRouters，然后判断集合中的每一个路由规则是否可以对当前调用进行过滤，如果路由规则符合当前调用，就对调用列表 List<Invoker<T>>进行筛选，去除不符合的调用者。

筛选逻辑在 Router 接口的 route(List<Invoker<T>> invokers, URL url, Invocation invocation)方法中实现，Router 接口的定义代码如下：

```

public interface Router extends Comparable<Router> {
    //获取当前路由的 url
    URL getUrl();
    //路由方法，对传入的 List<Invoker<T>>进行路由筛选，筛选的条件包括 Invocation
    对象和 consumer 的 url
    <T> List<Invoker<T>> route(List<Invoker<T>> invokers, URL url, Invocation
    invocation) throws RpcException;
}

```

Router 接口的实现主要有两个，一个是 ConditionRouter 即条件路由，另一个是 ScriptRouter 即脚本路由。Dubbo 的用户指南中如下描述：

Feature	Maturity	Strength	Problem	Advise	User
条件路由规则	Stable	基于条件表达式的路由规则，功能简单易用	有些复杂多分支条件情况，规则很难描述	可用于生产环境	Alibaba
脚本路由规则	Tested	基于脚本引擎的路由规则，功能强大	没有运行沙箱，脚本能力过于强大，可能成为后门	试用	

下面主要介绍条件路由 ConditionRouter，用户指南中对条件路由的说明如下：

路由规则

(+) (#)



2.2.0 以上版本支持



路由规则扩展点：[路由扩展](#)

向注册中心写入路由规则：(通常由监控中心或治理中心的页面完成)

```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("condition://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" +
URL.encode("http://10.20.160.198/wiki/display/dubbo/host = 10.20.153.10 => host = 10.20.153.11") + "));
```

其中：

- **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，请填入具体 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，缺省生效。
- **force=false**
当路由结果为空时，是否强制执行，如果不强制执行，路由结果为空的路由规则将自动失效，可不填，缺省为 **false**。
- **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，必须设为 true，需要注意设置会影响调用的性能，可不填，缺省为 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，缺省为 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。

条件路由规则

基于条件表达式的路由规则，如：

```
host = 10.20.153.10 => host = 10.20.153.11
```

规则：

- **"=>"**之前的为消费者匹配条件，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。
- **"=>"**之后为提供者地址列表的过滤条件，所有参数和提供者的 URL 进行对比，消费者最终只拿到过滤后的地址列表。

- 如果匹配条件为空，表示对所有消费方应用，如：=> host != 10.20.153.11
- 如果过滤条件为空，表示禁止访问，如：host = 10.20.153.10 =>

表达式：

- 参数支持：
 - 服务调用信息，如：method, argument 等 (暂不支持参数路由)
 - URL 本身的字段，如：protocol, host, port 等
 - 以及 URL 上的所有参数，如：application, organization 等
- 条件支持：
 - 等号"="表示"匹配"，如：host = 10.20.153.10
 - 不等号"!="表示"不匹配"，如：host != 10.20.153.10
- 值支持：
 - 以逗号","分隔多个值，如：host != 10.20.153.10,10.20.153.11
 - 以星号"*"结尾，表示通配，如：host != 10.20.*
 - 以美元符"\$"开头，表示引用消费者参数，如：host = \$host

示例：

1. 排除预发布机：

```
=> host != 172.22.3.91;
```

2. 白名单：(注意：一个服务只能有一条白名单规则，否则两条规则交叉，就都被筛选掉了)

```
host != 10.20.153.10, 10.20.153.11 =>
```

3. 黑名单：

```
host = 10.20.153.10, 10.20.153.11 =>
```

4. 服务寄宿在应用上，只暴露一部分的机器，防止整个集群挂掉：

```
=> host = 172.22.3.1*, 172.22.3.2*
```

5. 为重要应用提供额外的机器：

```
application != kylin => host != 172.22.3.95, 172.22.3.96;
```

6. 读写分离：

```
method = find*, list*, get*, is* => host = 172.22.3.94, 172.22.3.95, 172.22.3.96;
```

```
method != find*, list*, get*, is* => host = 172.22.3.97, 172.22.3.98;
```

7. 前后台分离：

```
application = bops => host = 172.22.3.91, 172.22.3.92, 172.2.3.93;
```

```
application != bops => host = 172.22.3.94, 172.22.3.95, 172.22.3.96;
```

8. 隔离不同机房网段：

```
host != 172.22.3.* => host != 172.22.3.*;
```

9. 提供者与消费者部署在同集群内，本机只访问本机的服务：

```
=> host = $host;
```

ConditionRouter 的 route(List<Invoker<T>> invokers, URL url, Invocation invocation)方法实现代码如下：

```
public <T> List<Invoker<T>> route(List<Invoker<T>> invokers, URL url, Invocation invocation) throws RpcException {
    if (invokers == null || invokers.size() == 0)
        return invokers;
    try {
        if (! matchWhen(url))
            return invokers;
        List<Invoker<T>> result = new ArrayList<Invoker<T>>();
        if (thenCondition == null) {
            //这里省略记录日志的代码
            return result;
        }
    }
}
```

```

    }
    for (Invoker<T> invoker : invokers) {
        if (matchThen(invoker.getUrl(), url))
            result.add(invoker);
    }
    if (result.size() > 0) {
        return result;
    } else if (force) {
        //这里省略记录日志的代码
        return result;
    }
} catch (Throwable t) {
    //这里省略记录日志的代码
}return invokers;
}

```

在代码实现中可以得知，消费者匹配规则（"=>"之前的条件）用 `whencondition` 表示，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；提供者地址列表过滤规则（"=>"之后的条件）用 `thencondition` 表示，所有参数和提供者的 URL 进行对比，消费者最终只拿到过滤后的地址列表。如果匹配条件为空，表示对所有消费方应用，如果过滤条件为空，表示禁止访问。

ConditionRouter 判断 `whencondition` 是否匹配当前调用时仅仅根据 `consumer` 的 `url` 来判断，虽然调用实例 `Invocation` 也作为参数传入，但是并没有参与 `whencondition` 的匹配过程，因此在 `whencondition` 中不能包含方法的参数信息，如果路由时需要根据调用方法的参数进行动态判断，则需要依据 `invocation` 中的 `parameterTypes` 和 `arguments` 来判断，因此需要改写的地方就是 `matchWhen` 方法。

如果需要考虑支持方法参数的动态路由，则需要结合 `parameterTypes` 和 `arguments` 来匹配，因为 `Invocation` 中保存了方法调用的参数类型列表、参数列表的信息，在执行路由规则时可以很容易拿到这两个数据。但是另一方面，基于条件表达式的规则在描述基本类型或者 `String` 类型时可以很容易实现，但是复杂类型尤其是自定义的对象类型的匹配则非常地复杂，可以考虑基于 `ONGL` 表达式引擎来实现。

另外，开发者指南中提供了对路由的拓展，如下：

路由扩展

(+) (#)

(1) 扩展说明：

从多个服务提供者方中选择一个进行调用。

(2) 扩展接口：

```

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

```

(3) 扩展配置：

```

<dubbo:protocol router="xxx" />

```

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

(4) 已知扩展：

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

(5) 扩展示例：

Maven 项目结构

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

XxxRouterFactory.java

```
package com.xxx;  
  
import com.alibaba.dubbo.rpc.cluster.RouterFactory;  
import com.alibaba.dubbo.rpc.Invoker;  
import com.alibaba.dubbo.rpc.Invocation;  
import com.alibaba.dubbo.rpc.RpcException;  
  
public class XxxRouterFactory implements RouterFactory {  
    public <T> List<Invoker<T>> select(List<Invoker<T>> invokers, Invocation invocation)  
throws RpcException {  
        // ...  
    }  
}
```

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

?

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

AbstractClusterInvoker 的 invoke(final Invocation invocation)方法在完成了对 Invocation 的调用者的路由之后，会获取负载均衡实例 LoadBalance loadbalance，然后根据当前 Invocation 实例、调用者列表和复杂均衡实例执行 AbstractClusterInvoker.doInvoke()抽象方法，该方法有其子类实现。

FailoverClusterInvoker 的 doInvoke 方法实现代码如下：

```
public Result doInvoke(Invocation, final List<Invoker<T>>, LoadBalance) throws RpcException{
    List<Invoker<T>> copyinvokers = invokers;
    checkInvokers(copyinvokers, invocation);
    //获取 url 中 retries 关键字的值
    int len = getUrl().getMethodParameter(invocation.getMethodName(),
        Constants.RETRIES_KEY, Constants.DEFAULT_RETRIES) + 1;
    if (len <= 0)
        len = 1;
    // retry loop.
    RpcException le = null; // last exception.
    // invoked invokers.
    List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyinvokers.size());
    Set<String> providers = new HashSet<String>(len);
    for (int i = 0; i < len; i++) {
        //重试时，进行重新选择，避免重试时 invoker 列表已发生变化.
        //注意：如果列表发生变化，那么 invoked 判断会失效，因为 invoker 示例已经改变
        if (i > 0) {
            checkWheatherDestoried();
            copyinvokers = list(invocation);
            //重新检查一下
            checkInvokers(copyinvokers, invocation);
        }
        Invoker<T> invoker = select(loadbalance, invocation, copyinvokers, invoked);
        invoked.add(invoker);
        RpcContext.getContext().setInvokers((List)invoked);
        try {
            Result result = invoker.invoke(invocation);
            if (le != null && logger.isWarnEnabled()) {
                //省略记录日志的代码
            }
            return result;
        } catch (RpcException e) {
            if (e.isBiz())// biz exception.
                throw e;
            le = e;
        } catch (Throwable e) {
            le = new RpcException(e.getMessage(), e);
        } finally {
            providers.add(invoker.getUrl().getAddress());
        }
    }
    throw new RpcException("...");
}
```

关于 FailoverClusterInvoker 的容错实现在代码中很容易看出，即根据调用者列表进行一次调用，直到调用成功或者达到 retries 次数的上限，这样即实现了基于失败重试的容错机制。

关于负载均衡部分，其逻辑体现在 AbstractClusterInvoker 的方法 Invoker<T> select(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers, List<Invoker<T>> selected) throws RpcException 中，该方法会调用 LoadBalance 接口的实例的 select(List<Invoker<T>> invokers, URL url, Invocation invocation) throws RpcException 方法，其中 LoadBalance 接口的定义代码如下：

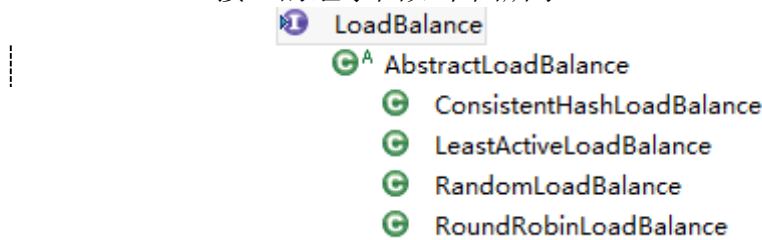
```
public interface LoadBalance {
    @Adaptive("loadbalance")
```

```

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

```

LoadBalance 接口的继承图如下图所示：



默认情况下的负载均衡实现是 RandomLoadBalance，它会按照特定的权重来随机从提供者列表中随机选择一个进行调用。

Dubbo 的开发者指南中提供了负载均衡的拓展，如下：

负载均衡扩展

(+) (#)

(1) 扩展说明：

从多个服务提供者方中选择一个进行调用。

(2) 扩展接口：

?

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

(3) 扩展配置：

?

```

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

```

(4) 已知扩展：

?

```

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

```

(5) 扩展示例：

Maven 项目结构

?

```

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

```

-XxxLoadBalance.java (实现 LoadBalance 接口) -resources -META-INF -dubbo -com.alibaba.dubbo.rpc.cluster.LoadBalance (纯文本文件， 内容为： xxx=com.xxx.XxxLoadBalance)
XxxLoadBalance.java
? <pre> package com.xxx; import com.alibaba.dubbo.rpc.cluster.LoadBalance; import com.alibaba.dubbo.rpc.Invoker; import com.alibaba.dubbo.rpc.Invocation; import com.alibaba.dubbo.rpc.RpcException; public class XxxLoadBalance implements LoadBalance { public <T> Invoker<T> select(List<Invoker<T>> invokers, Invocation invocation) throws RpcException { // ... } } </pre>
META-INF/dubbo/com.alibaba.dubbo.rpc.cluster.LoadBalance
? <pre> xxx=com.xxx.XxxLoadBalance; </pre>

另外，远程调用在实际执行时会添加一个过滤器链，在过滤器链的末尾来真正执行远程调用，过滤器链满足 U 型管规则，可以参考 Tomcat 容器的过滤器规则来理解。

过滤器链的控制逻辑在 ProtocolFilterWrapper 类的 buildInvokerChain() 方法中实现，并由 Invoker 的实现者触发，buildInvokerChain() 方法的代码如下：

```

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

```

```

        public Result invoke(Invocation invocation) throws RpcException {
            return filter.invoke(next, invocation);
        }
        public void destroy() {
            invoker.destroy();
        }
        @Override
        public String toString() {
            return invoker.toString();
        }
    };
}
}
return last;
}

```

过滤器 Filter 的接口定义如下：

```

public interface Filter {
    Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException;
}

```

Filter 的实现类包括 ConsumerContextFilter、MonitorFilter、AccessLogFilter、EchoFilter、MockFilter、CacheFilter 和 FutureFilter 等，每个过滤器完成不同的过滤功能。在默认情况下过滤器链的调用顺序为：ConsumerContextFilter -> MonitorFilter -> FutureFilter。

ConsumerContextFilter 过滤器主要处理消费者调用时携带的上下文信息，并附加到 RpcContext 中，ConsumerContextFilter 的过滤代码如下：

```

public class ConsumerContextFilter implements Filter {
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        RpcContext.getContext()
            .setInvoker(invoker)
            .setInvocation(invocation)
            .setLocalAddress(NetUtils.getLocalHost(), 0)
            .setRemoteAddress(invoker.getUrl().getHost(),
                invoker.getUrl().getPort());
        if (invocation instanceof RpcInvocation)
            ((RpcInvocation)invocation).setInvoker(invoker);
        try {
            return invoker.invoke(invocation);
        } finally {
            RpcContext.getContext().clearAttachments();
        }
    }
}

```

MonitorFilter 主要对调用过程进行监控，全部代码就不在此列出了，主要说明一下拦截的过程，代码如下：

```

public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    if (invoker.getUrl().hasParameter(Constants.MONITOR_KEY)) {
        // 提供方必须在 invoke()之前获取 context 信息
        RpcContext context = RpcContext.getContext();
        long start = System.currentTimeMillis(); // 记录起始时间戳
        getConcurrent(invoker, invocation).incrementAndGet(); // 并发计数
        try {
            Result result = invoker.invoke(invocation); // 让调用链往下执行
            collect(invoker, invocation, result, context, start, false);
        }
    }
}

```

```

        return result;
    } catch (RpcException e) {
        collect(invoker, invocation, null, context, start, true);
        throw e;
    } finally {
        getConcurrent(invoker, invocation).decrementAndGet(); // 并发计数
    }
} else {
    return invoker.invoke(invocation);
}
}

```

其核心代码是 `collect()` 方法，该方法会将监控的数据通过 `Monitor monitor` 进行收集，收集后存放在本地内存，每隔固定的时间（默认是 60 秒）上传到监控节点上。

`Monitor` 的默认实现是 `com.alibaba.dubbo.monitor.dubbo` 包下的 `DubboMonitor` 类，`DubboMonitor` 中会定义一个 `ConcurrentHashMap` 的对象用于保存监控信息，`MonitorFilter` 的 `collect` 方法在执行时会将监控数据保存到该 `ConcurrentHashMap` 中；另外，`DubboMonitor` 默认会开启一个基于线程池的定时任务执行器 `ScheduledExecutorService`，并且在构造函数中会启动一个周期性执行的任务将 `ConcurrentHashMap` 中的数据发送到监控节点上，发送逻辑在 `send()` 方法中实现，发送的周期是 60000 毫秒（即一分钟）。

`DubboMonitor` 的 `collect` 方法、发送监控数据到远程监控节点的 `send()` 方法以及初始化构造函数代码如下：

```

public void collect(URL url) {
    // 读写统计变量
    int success = url.getParameter(MonitorService.SUCCESS, 0);
    int failure = url.getParameter(MonitorService.FAILURE, 0);
    int input = url.getParameter(MonitorService.INPUT, 0);
    int output = url.getParameter(MonitorService.OUTPUT, 0);
    int elapsed = url.getParameter(MonitorService.ELAPSED, 0);
    int concurrent = url.getParameter(MonitorService.CONCURRENT, 0);
    // 初始化原子引用
    Statistics statistics = new Statistics(url);
    AtomicReference<long[]> reference = statisticsMap.get(statistics);
    if (reference == null) {
        statisticsMap.putIfAbsent(statistics, new AtomicReference<long[]>());
        reference = statisticsMap.get(statistics);
    }
    // CompareAndSet 并发加入统计数据
    long[] current;
    long[] update = new long[LENGTH];
    do {
        current = reference.get();
        if (current == null) {
            update[0] = success;
            update[1] = failure;
            update[2] = input;
            update[3] = output;
            update[4] = elapsed;
            update[5] = concurrent;
            update[6] = input;
            update[7] = output;
            update[8] = elapsed;

```

```

        update[9] = concurrent;
    } else {
        update[0] = current[0] + success;
        update[1] = current[1] + failure;
        update[2] = current[2] + input;
        update[3] = current[3] + output;
        update[4] = current[4] + elapsed;
        update[5] = (current[5] + concurrent) / 2;
        update[6] = current[6] > input ? current[6] : input;
        update[7] = current[7] > output ? current[7] : output;
        update[8] = current[8] > elapsed ? current[8] : elapsed;
        update[9] = current[9] > concurrent ? current[9] : concurrent;
    }
    } while (! reference.compareAndSet(current, update));
}

public void send() {
    if (logger.isInfoEnabled()) {
        logger.info("Send statistics to monitor " + getUrl());
    }
    String timestamp = String.valueOf(System.currentTimeMillis());
    for (Map.Entry<Statistics, AtomicReference<long[]>> entry : statisticsMap.entrySet()) {
        // 获取已统计数据
        Statistics statistics = entry.getKey();
        AtomicReference<long[]> reference = entry.getValue();
        long[] numbers = reference.get();
        long success = numbers[0];
        long failure = numbers[1];
        long input = numbers[2];
        long output = numbers[3];
        long elapsed = numbers[4];
        long concurrent = numbers[5];
        long maxInput = numbers[6];
        long maxOutput = numbers[7];
        long maxElapsed = numbers[8];
        long maxConcurrent = numbers[9];

        // 发送汇总信息
        URL url = statistics.getUrl().addParameters(
            MonitorService.TIMESTAMP, timestamp,
            MonitorService.SUCCESS, String.valueOf(success),
            MonitorService.FAILURE, String.valueOf(failure),
            MonitorService.INPUT, String.valueOf(input),
            MonitorService.OUTPUT, String.valueOf(output),
            MonitorService.ELAPSED, String.valueOf(elapsed),
            MonitorService.CONCURRENT, String.valueOf(concurrent),
            MonitorService.MAX_INPUT, String.valueOf(maxInput),
            MonitorService.MAX_OUTPUT, String.valueOf(maxOutput),
            MonitorService.MAX_ELAPSED, String.valueOf(maxElapsed),
            MonitorService.MAX_CONCURRENT, String.valueOf(maxConcurrent)
        );
        monitorService.collect(url);
        // 减掉已统计数据
        long[] current;
        long[] update = new long[LENGTH];
        do {
            current = reference.get();

```

```

        if (current == null) {
            update[0] = 0;
            ...
            update[5] = 0;
        } else {
            update[0] = current[0] - success;
            ...
            update[5] = current[5] - concurrent;
        }
    } while (! reference.compareAndSet(current, update));
}

public DubboMonitor(Invoker<MonitorService>, MonitorService) {
    this.monitorInvoker = monitorInvoker;
    this.monitorService = monitorService;
    //默认的监控间隔为 60 秒
    this.monitorInterval = monitorInvoker.getUrl().getPositiveParameter("interval", 60000);
    // 启动统计信息收集定时器
    sendFuture = scheduledExecutorService.scheduleWithFixedDelay(new Runnable() {
        public void run() {
            // 收集统计信息
            try {
                send();
            } catch (Throwable t) { // 防御性容错
                //记录发送失败的日志
            }
        }
    }, monitorInterval, monitorInterval, TimeUnit.MILLISECONDS);
}

```

如果要拓展监控部分的功能时，最佳拓展点是 `MonitorFactory`，从 `MonitorFilter` 的 `collect` 方法中可以看出 `DubboMonitor` 实例是由 `MonitorFactory` 构造的，如果用户自定义了 `MonitorFactory`，则可以改变 `DubboMonitor` 的行为，比如改变监控数据的存储形式（因为默认的 `DubboMonitor` 实现中把监控数据保存在内存中，这样当调用过程比较频繁时，监控数据就会非常多，这样势必会造成巨大的内存消耗，并且 `DubboMonitor` 默认开启 3 个线程实例的线程池来执行发送监控数据的任务，这里的线程数量是值得商榷的，并且上传间隔也可以默认设置小一点）。

Dubbo 的开发者指南中介绍了监控部分的拓展，如下：

监控中心扩展

(+) (#)

(1) 扩展说明：

负责服务调用次和调用时间的监控。

(2) 扩展接口：

```

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

```

(3) 扩展配置：

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

(4) 已知扩展:

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

(5) 扩展示例:

Maven 项目结构

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

XxxMonitorFactory.java

```
package com.xxx;
import com.alibaba.dubbo.monitor.MonitorFactory;
import com.alibaba.dubbo.monitor.Monitor;
import com.alibaba.dubbo.common.URL;

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

XxxMonitor.java

```
package com.xxx;
import com.alibaba.dubbo.monitor.Monitor;
public class XxxMonitor implements Monitor {
    public void count(URL statistics) {
        // ...
    }
}
```

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

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