芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-05-05
Dubbo

精尽 Dubbo 源码解析 —— 集群容错（八）之 Mock 实现

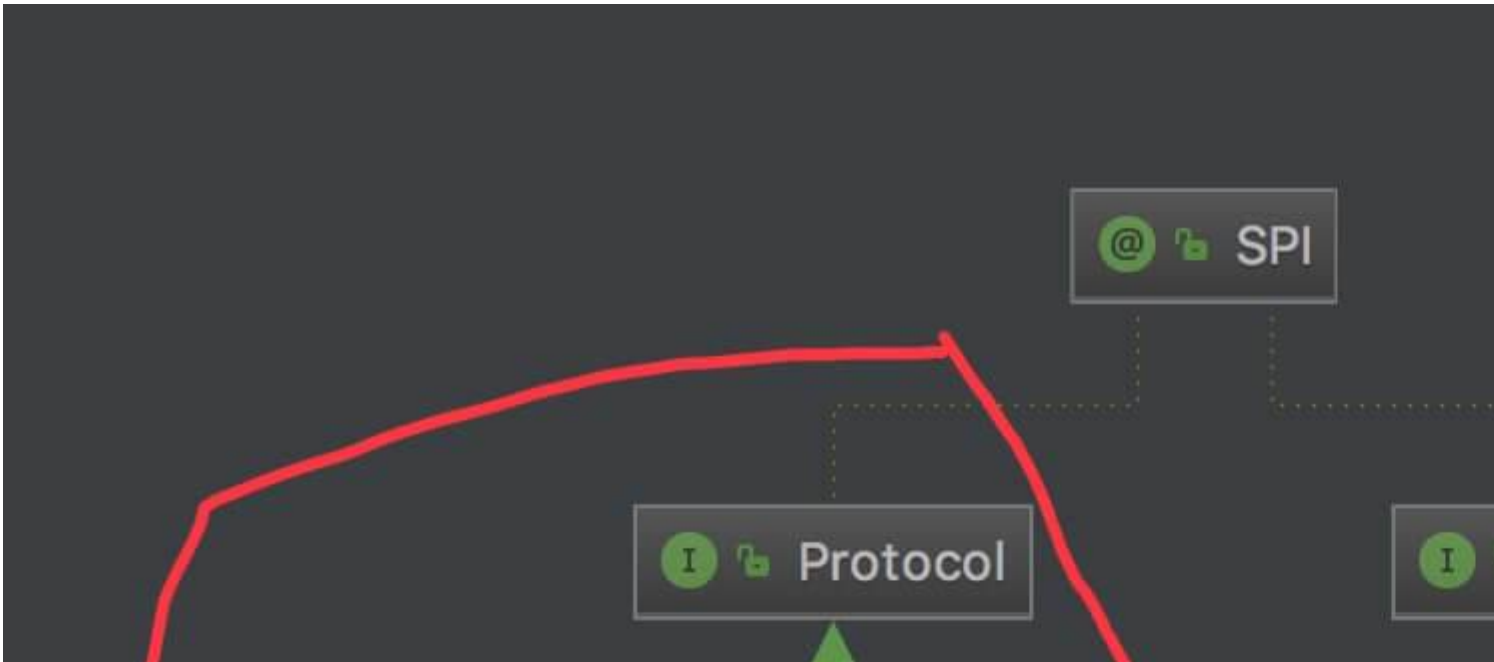
> 本文基于 Dubbo 2.6.1 版本，望知悉。

1. 概述

本文接 《精尽 Dubbo 源码解析 —— 集群容错（七）之 Router 实现》 一文，分享 `dubbo-cluster` 模块， `mock` 包，实现 Dubbo 如下功能：

> 本地伪装：通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。
> 服务降级：可以通过服务降级功能，临时屏蔽某个出错的非关键服务，并定义降级后的返回策略。
>
> 老芳芳：如果不熟悉的胖友，推荐结合着 《Dubbo 用户指南 —— 本地伪装》 和 《Dubbo 用户指南 —— 服务降级》 一起学习。
>
> 注意，《Dubbo 用户指南 —— 本地伪装》 的文档有点问题：Spring XML 是通过 `<dubbo:reference />` 配置，而不是 `<dubbo:service />` 。

本文涉及类如下图：



分成两个部分：

- MockClusterWrapper + MockClusterInvoker + MockClusterSelector
- MockProtocol + MockInvoker

2. MockClusterWrapper

com.alibaba.dubbo.rpc.cluster.support.wrapper.MockClusterWrapper ，实现 Cluster 接口，Mock Cluster Wrapper 实现类。代码如下：

```
public class MockClusterWrapper implements Cluster {

    /**
     * 真正的 Cluster 对象
     */
    private Cluster cluster;

    public MockClusterWrapper(Cluster cluster) {
        this.cluster = cluster;
    }

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new MockClusterInvoker<T>(directory, // <2>
                this.cluster.join(directory)); // <1>
    }

}
```

cluster 字段，真正的 Cluster 对象。因为 MockClusterWrapper 是 Dubbo SPI Wrapper 类，所以对应的 Cluster 对象，都会被它所包装。不理解的胖友，可以看下 《精尽 Dubbo 源码分析 —— 拓展机制 SPI》 的 「4.4.2 createExtension」 的【第 24 至 30 行】。

<1> 处：调用 Cluster#join(directory) 方法，创建真正的 Cluster Invoker 对象。

<2> 处：创建 MockClusterInvoker 对象。详细解析，见 「3. MockClusterInvoker」 。

3. MockClusterInvoker

com.alibaba.dubbo.rpc.cluster.support.wrapper.MockClusterInvoker ，实现 Invoker 接口，MockClusterWrapper 对应的 Invoker 实现类。

3.1 构造方法

```
private final Directory<T> directory;
/**
 * 真正的 Invoker 对象
 */
private final Invoker<T> invoker;

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

3.2 invoke

```
 1: @Override
 2: public Result invoke(Invocation invocation) throws RpcException {
 3:     Result result;
 4:     // 获得 "mock" 配置项，有多种配置方式
 5:     String value = directory.getUrl().getMethodParameter(invocation.getMethodName(), Constants.MOCK_KEY, Boolean.
 6:     //【第一种】无 mock
 7:     if (value.length() == 0 || value.equalsIgnoreCase("false")) {
 8:         // no mock
 9:         // 调用原 Invoker ，发起 RPC 调用
10:         result = this.invoker.invoke(invocation);
11:     //【第二种】强制服务降级 http://dubbo.apache.org/zh-cn/docs/user/demos/service-downgrade.html
12:     } else if (value.startsWith("force")) {
13:         if (logger.isWarnEnabled()) {
14:             logger.info("force-mock: " + invocation.getMethodName() + " force-mock enabled , url : " + directory.
15:         }
16:         // force:direct mock
17:         // 直接调用 Mock Invoker ，执行本地 Mock 逻辑
18:         result = doMockInvoke(invocation, null);
19:     // 【第三种】失败服务降级 http://dubbo.apache.org/zh-cn/docs/user/demos/service-downgrade.html
20:     } else {
21:         // fail-mock
22:         try {
23:             // 调用原 Invoker ，发起 RPC 调用
24:             result = this.invoker.invoke(invocation);
25:         } catch (RpcException e) {
26:             // 业务性异常，直接抛出
27:             if (e.isBiz()) {
28:                 throw e;
29:             } else {
30:                 if (logger.isWarnEnabled()) {
31:                     logger.info("fail-mock: " + invocation.getMethodName() + " fail-mock enabled , url : " + dire
32:                 }
33:                 // 失败后，调用 Mock Invoker ，执行本地 Mock 逻辑
34:                 result = doMockInvoke(invocation, e);
35:             }
36:         }
37:     }
38:     return result;
39: }
```

第 5 行：获得 "mock" 配置项。根据不同的配置，分成三种情况。

========== 第一种：无 Mock ==========

第 10 行：只调用真正的 invoker 的 #invoke(invocation) 方法，发起 RPC 调用，即不进行 Mock 逻辑。

========== 第二种：强制服务降级 ==========

第 12 行："mock" 配置项以 "force" 开头，强制服务降级，即 《Dubbo 用户指南 —— 服务降级》 。

第 18 行：直接调用 #doMockInvoke(invocation, null) 方法，调用 Mock Invoker ，执行本地 Mock 逻辑。详细解析，见 「3.3 doMockInvoke」 。

========== 第三种：失败服务降级 ==========

第 24 行：优先，调用真正的 invoker 的 #invoke(invocation) 方法，发起 RPC 调用，即不进行 Mock 逻辑。

第 25 至 36 行：处理 RpcException 异常。也仅仅处理这种类型的异常。

- 第 26 至 28 行：若发生业务性异常，直接抛出异常。

- 第 29 至 35 行：失败后，调用 #doMockInvoke(invocation, null) 方法，调用 Mock Invoker ，执行本地 Mock 逻辑。详细解析，见 「3.3 doMockInvoke」 。

总的来说，MockClusterInvoker 的 #invoke(Invocation) 方法的过程，根据不同的 "mock" 配置 ，“组合”调用真正的和 Mock 的 Invoker 。

3.3 doMockInvoke

```
1: private Result doMockInvoke(Invocation invocation, RpcException e) {
2:     Result result;
3:     // 第一步，获得 Mock Invoker 对象
4:     Invoker<T> mInvoker;
5:     // 路由匹配 Mock Invoker 集合
6:     List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
7:     // 如果不存在，创建 MockInvoker 对象
8:     if (mockInvokers == null || mockInvokers.isEmpty()) {
9:         mInvoker = (Invoker<T>) new MockInvoker(directory.getUrl());
10:    // 如果存在，选择第一个
11:    } else {
12:        mInvoker = mockInvokers.get(0);
13:    }
14:    // 第二步，调用，执行本地 Mock 逻辑
15:    try {
16:        result = mInvoker.invoke(invocation);
17:    } catch (RpcException me) {
18:        if (me.isBiz()) {
19:            result = new RpcResult(me.getCause());
20:        } else {
21:            throw new RpcException(me.getCode(), getMockExceptionMessage(e, me), me.getCause());
22:        }
23:    } catch (Throwable me) {
24:        throw new RpcException(getMockExceptionMessage(e, me), me.getCause());
25:    }
26:    return result;
27: }
```

第 3 至 13 行：第一步，获得 MockInvoker 对象。
- 第 6 行：调用 #selectMockInvoker(invocation) 方法，路由匹配 Mock Invoker 集合。详细解析，见 「3.4 selectMockInvoker」 。
- 第 7 至 9 行：若不存在，创建 MockInvoker 对象。
- 第 10 至 13 行：若已不能再，选择第一个 Mock Invoker 对象。

第 14 至 16 行：调用 MockInvoker#invoke(invocation) 方法，执行本地 Mock 逻辑。
第 17 至 25 行：处理异常。

3.4 selectMockInvoker

```
1: private List<Invoker<T>> selectMockInvoker(Invocation invocation) {
2:     List<Invoker<T>> invokers = null;
3:     if (invocation instanceof RpcInvocation) {
4:         // 存在隐含契约(虽然在接口声明中增加描述，但扩展性会存在问题.同时放在 attachment 中的做法需要改进
5:         ((RpcInvocation) invocation).setAttachment(Constants.INVOCATION_NEED_MOCK, Boolean.TRUE.toString());
6:         // directory 根据 invocation 中 attachment 是否有 Constants.INVOCATION_NEED_MOCK，来判断获取的是 normal i
7:         try {
8:             invokers = directory.list(invocation);
```

```
 9:         } catch (RpcException e) {
10:             if (logger.isInfoEnabled()) {
11:                 logger.info("Exception when try to invoke mock. Get mock invokers error for service:" + directory
12:             }
13:         }
14:     }
15:     return invokers;
16: }
```

第 5 行：设置 RpcInvocation 的 "invocation.need.mock" 为 true 。
第 8 行：调用 Directory#list(invocation) 方法，获得所有 Invoker 集合。因为【第 5 行】设置了 "invocation.need.mock" 为 true ，所以实际获得的是 MockInvoker 集合。详细解析，见「3.4.1 MockInvokersSelector」 。

3.4.1 MockInvokersSelector

com.alibaba.dubbo.rpc.cluster.router.MockInvokersSelector ，实现 Router 接口，MockInvoker 路由选择器实现类。

因为 AbstractDirectory 的 #setRouters(List<Router> routers) 方法中，都会添加 MockInvokersSelector ，如下图所示：

```
protected void setRouters(List<Router> routers) {
    // copy list // 复制 routers ，因为下面要修改
    routers = routers == null ? new ArrayList<Rou
    // append url router
    // 拼接 `url` 中，配置的路由规则
    String routerkey = url.getParameter(Constants
    if (routerkey != null && routerkey.length() >
        RouterFactory routerFactory = ExtensionLo
        routers.add(routerFactory.getRouter(url))
    }
    // append mock invoker selector
    routers.add(new MockInvokersSelector());
    // 排序
    Collections.sort(routers);
    // 赋值给属性
    this.routers = routers;
}
```

所以，每次 Directory#list(invocation) 的过程中，都会执行 MockInvokersSelector 的路由逻辑。

#route(List<Invoker<T>> invokers, URL url, Invocation invocation) 方法，根据 "invocation.need.mock" 路由匹配对应类型的 Invoker 集合：

1、若为 true ，MockInvoker 集合。
2、若为 false ，普通 Invoker 集合。

代码如下：

```
1: @Override
2: public <T> List<Invoker<T>> route(final List<Invoker<T>> invokers, URL url, final Invocation invocation) throws R
3:     // 获得普通 Invoker 集合
4:     if (invocation.getAttachments() == null) {
5:         return getNormalInvokers(invokers);
6:     } else {
7:         // 获得 "invocation.need.mock" 配置项
8:         String value = invocation.getAttachments().get(Constants.INVOCATION_NEED_MOCK);
9:         // 获得普通 Invoker 集合
10:        if (value == null) {
11:            return getNormalInvokers(invokers);
12:        // 获得 MockInvoker 集合
13:        } else if (Boolean.TRUE.toString().equalsIgnoreCase(value)) {
14:            return getMockedInvokers(invokers);
15:        }
16:    }
17:    // 其它，不匹配，直接返回 `invokers` 集合
18:    return invokers;
19: }
```

分成三种情况，我们一个一个来看。在看具体情况之前，我们先来看 #hasMockProviders(invokers) 方法，判断是否有 MockInvoker 。代码如下：

```
private <T> boolean hasMockProviders(final List<Invoker<T>> invokers) {
    boolean hasMockProvider = false;
    for (Invoker<T> invoker : invokers) {
        if (invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) { // 协议为 "mock"
            hasMockProvider = true;
            break;
        }
    }
    return hasMockProvider;
}
```

- 通过 protocol = "mock" 来判断，是否为 MockInvoker 。所以只要不为 MockInvoker ，就是普通 Invoker 。关于 "mock" 协议，我们稍后在 「4. MockProtocol」 中，详细解析。

【第一种】第 3 至 5 行 || 第 9 至 11 行：若未设置 "invocation.need.mock" 配置项，调用 #getNormalInvokers(invokers) 方法，获得普通 Invoker 集合。代码如下：

```
1: private <T> List<Invoker<T>> getNormalInvokers(final List<Invoker<T>> invokers) {
2:     // 不包含 MockInvoker 的情况下，直接返回 `invokers` 集合
3:     if (!hasMockProviders(invokers)) {
4:         return invokers;
5:     } else {
```

```
 6:         // 若包含 MockInvoker 的情况下，过滤掉 MockInvoker ，创建普通 Invoker 集合
 7:         List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(invokers.size());
 8:         for (Invoker<T> invoker : invokers) {
 9:             if (!invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) {
10:                 sInvokers.add(invoker);
11:             }
12:         }
13:         return sInvokers;
14:     }
15: }
```

- 第 2 至 4 行：调用 #hasMockProviders(invokers) 方法，判断不包含 MockInvoker 的情况，直接返回 invokers 集合。
- 第 6 至 13 行：若包含 MockInvoker 的情况，过滤掉 MockInvoker ，创建普通 Invoker 集合。

【第二种】第 12 至 15 行：若设置 "invocation.need.mock" 配置项为 true ，调用 #getMockedInvokers(invokers) 方法，获得 MockInvoker 集合。代码如下：

```
private <T> List<Invoker<T>> getMockedInvokers(final List<Invoker<T>> invokers) {
    // 不包含 MockInvoker 的情况下，直接返回 null
    if (!hasMockProviders(invokers)) {
        return null;
    }
    // 过滤掉普通 kInvoker ，创建 MockInvoker 集合
    List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(1); // 一般情况就一个，所以设置了默认数组大小为 1 。
    for (Invoker<T> invoker : invokers) {
        if (invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) {
            sInvokers.add(invoker);
        }
    }
    return sInvokers;
}
```

- 和 #getNormalInvokers(invokers) 方法，相反。比较易懂，胖友看代码注释。

【第三种】第 18 行：其它，不匹配，直接返回 invokers 集合。理论上，应该调用和【第一种】一样，调用 #getNormalInvokers(invokers) 方法。不过没关系， 从目前代码来看，这块是走不到的。

4. MockProtocol

com.alibaba.dubbo.rpc.support.MockProtocol ，实现 AbstractProtocol 抽象类，用于在服务消费者，通过类型为 "mock" 的 URL ，引用创建 MockInvoker 对象。代码如下：

```
public final class MockProtocol extends AbstractProtocol {

    @Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        throw new UnsupportedOperationException();
    }

    @Override
```

```
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        return new MockInvoker<T>(url);
    }

    @Override
    public int getDefaultPort() {
        return 0;
    }

}
```

#export(Invoker) 实现方法，不允许调用，直接抛出 UnsupportedOperationException 异常。

#refer(Class<T> type, Url) 实现方法，引用创建 MockInvoker 对象。一般情况下，我们可以通过 dubbo-admin 运维平台或者直接向 Zookeeper 写入静态 URL ，例如：

```
// 实际写入的 URL
/dubbo/com.alibaba.dubbo.demo.DemoService/providers/mock%3A%2F%2F10.20.153.11%2Fcom.alibaba.dubbo.demo.DemoServ

// decode URL
/dubbo/com.alibaba.dubbo.demo.DemoService/providers/mock://10.20.153.11/com.alibaba.dubbo.demo.DemoService?dyna
```

- 为什么要是静态 URL 呢？因为非静态 URL ，可能被注册中心删除。

当然，我们在 「3.3 doMockInvoke」 中也看到，即使不手动添加 "mock" URL ，在【第 9 行】代码中也会自动创建 MockInvoker 对象。

5. MockInvoker

com.alibaba.dubbo.rpc.support.MockInvoker ，实现 Invoker 接口，Mock Invoker 实现类。

5.1 构造方法

```
/**
 * ProxyFactory$Adaptive 对象
 */
private final static ProxyFactory proxyFactory = ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveEx

/**
 * mock 与 Invoker 对象的映射缓存
 *
 * @see #getInvoker(String)
 */
private final static Map<String, Invoker<?>> mocks = new ConcurrentHashMap<String, Invoker<?>>();
/**
 * mock 与 Throwable 对象的映射缓存
 *
 * @see #getThrowable(String)
 */
private final static Map<String, Throwable> throwables = new ConcurrentHashMap<String, Throwable>();

/**
```

```
     * URL 对象
     */
    private final URL url;

    public MockInvoker(URL url) {
        this.url = url;
    }
```

5.2 invoke

#invoke(Invocation) 实现方法，执行 Mock 逻辑。代码如下：

```
 1: @Override
 2: public Result invoke(Invocation invocation) throws RpcException {
 3:     if (invocation instanceof RpcInvocation) {
 4:         ((RpcInvocation) invocation).setInvoker(this);
 5:     }
 6:     // 获得 `"mock"` 配置项，方法级 > 类级
 7:     String mock = getUrl().getParameter(invocation.getMethodName() + "." + Constants.MOCK_KEY);
 8:     if (StringUtils.isBlank(mock)) {
 9:         mock = getUrl().getParameter(Constants.MOCK_KEY);
10:     }
11:     if (StringUtils.isBlank(mock)) { // 不允许为空
12:         throw new RpcException(new IllegalAccessException("mock can not be null. url :" + url));
13:     }
14:     // 标准化 `"mock"` 配置项
15:     mock = normalizedMock(URL.decode(mock));
16:     // 等于 "return " ，返回值为空的 RpcResult 对象
17:     if (Constants.RETURN_PREFIX.trim().equalsIgnoreCase(mock.trim())) {
18:         RpcResult result = new RpcResult();
19:         result.setValue(null);
20:         return result;
21:     // 以 "return " 开头，返回对应值的 RpcResult 对象
22:     } else if (mock.startsWith(Constants.RETURN_PREFIX)) {
23:         mock = mock.substring(Constants.RETURN_PREFIX.length()).trim();
24:         mock = mock.replace('`', '"');
25:         try {
26:             // 解析返回类型
27:             Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
28:             // 解析返回值
29:             Object value = parseMockValue(mock, returnTypes);
30:             // 创建对应值的 RpcResult 对象，并返回
31:             return new RpcResult(value);
32:         } catch (Exception ew) {
33:             throw new RpcException("mock return invoke error. method :" + invocation.getMethodName() + ", mock:"
34:         }
35:     // 以 "throw" 开头，抛出 RpcException 异常
36:     } else if (mock.startsWith(Constants.THROW_PREFIX)) {
37:         mock = mock.substring(Constants.THROW_PREFIX.length()).trim();
38:         mock = mock.replace('`', '"');
39:         if (StringUtils.isBlank(mock)) { // 禁止为空
40:             throw new RpcException(" mocked exception for Service degradation. ");
41:         } else { // user customized class
42:             // 创建自定义异常
43:             Throwable t = getThrowable(mock);
44:             // 抛出业务类型的 RpcException 异常
45:             throw new RpcException(RpcException.BIZ_EXCEPTION, t);
```

```
46:         }
47:     // 自定义 Mock 类，执行自定义逻辑
48:     } else {
49:         try {
50:             // 创建 Invoker 对象
51:             Invoker<T> invoker = getInvoker(mock);
52:             // 执行 Invoker 对象的调用逻辑
53:             return invoker.invoke(invocation);
54:         } catch (Throwable t) {
55:             throw new RpcException("Failed to create mock implemention class " + mock, t);
56:         }
57:     }
58: }
```

第 6 至 13 行：获得 "mock" 配置项，优先从方法级的参数，再从类级的参数。
第 15 行：调用 #normalizedMock(mock) 方法，标准化 "mock" 配置项。详细解析，见 「5.3 normalizedMock」 。
下面根据 mock ，分成三种情况。
【第 1.1 种】第 16 至 20 行：等于 "return " ，返回值为空 的 RpcResult 对象。
【第 1.2 种】第 21 至 34 行：以 "return" 开头，返回对应值 的 RpcResult 对象。
- 第29 行：调用 #parseMockValue(mock, returnTypes) 方法，解析返回值。详细解析，见 「5.4 parseMockValue」 。

【第 2 种】第 35 至 46 行：以 "throw" 开头，抛出对应的 RpcException 异常。
- 第 43 行：调用 #getThrowable(mock) 对象，创建自定义异常。详细解析，见 「5.5 getThrowable」 。

【第 3 种】第 47 至 57 行：自定义 Mock 类，执行自定义逻辑。
- 第 51 行：调用 #getInvoker(mock) 方法，获得 Invoker 对象。详细解析，见 「5.6 getInvoker」 。
- 第 53 行：调用 Invoker#invoke(invocation) 方法，执行自定义 Mock 逻辑。

5.3 normalizedMock

```
 1: private String normalizedMock(String mock) {
 2:     // 若为空，直接返回
 3:     if (mock == null || mock.trim().length() == 0) {
 4:         return mock;
 5:     // 若果为 "true" "default" "fail" "force" 四种字符串，修改为对应接口 + "Mock" 类
 6:     } else if (ConfigUtils.isDefault(mock) || "fail".equalsIgnoreCase(mock.trim()) || "force".equalsIgnoreCase(mo
 7:         mock = url.getServiceInterface() + "Mock";
 8:     }
 9:     // 若以 "fail:" 开头，去掉该开头
10:     if (mock.startsWith(Constants.FAIL_PREFIX)) {
11:         mock = mock.substring(Constants.FAIL_PREFIX.length()).trim();
12:     // 若以 "force:" 开头，去掉该开头
13:     } else if (mock.startsWith(Constants.FORCE_PREFIX)) {
14:         mock = mock.substring(Constants.FORCE_PREFIX.length()).trim();
15:     }
16:     return mock;
17: }
```

【第一种】第 2 至 4 行：若为空，直接返回。
【第二种】第 5 至 8 行：若为 true default fail force ，修改为对应接口 + "Mock" 。
【第三种】第 9 至 15 行：若以 "fail:" 或 "force:" 开头 ，去掉开头。该开头，仅用于 MockClusterInvoker 表示强制 Mock 还是失败 Mock 。

5.4 parseMockValue

```
public static Object parseMockValue(String mock) {
    return parseMockValue(mock, null);
}

public static Object parseMockValue(String mock, Type[] returnTypes) {
    // 解析值（不考虑返回类型）
    Object value;
    if ("empty".equals(mock)) { // 未赋值的对象，即 new XXX() 对象
        value = ReflectUtils.getEmptyObject(returnTypes != null && returnTypes.length > 0 ? (Class<?>) returnTypes[0]
    } else if ("null".equals(mock)) { // null
        value = null;
    } else if ("true".equals(mock)) { // true
        value = true;
    } else if ("false".equals(mock)) { // false
        value = false;
    } else if (mock.length() >= 2 && (mock.startsWith("\"") && mock.endsWith("\"")
            || mock.startsWith("'") && mock.endsWith("'"))) { // 使用 '' 或 "" 的字符串，截取掉头尾
        value = mock.subSequence(1, mock.length() - 1);
    } else if (returnTypes != null && returnTypes.length > 0 && returnTypes[0] == String.class) { // 字符串
        value = mock;
    } else if (StringUtils.isNumeric(mock)) { // 数字
        value = JSON.parse(mock);
    } else if (mock.startsWith("{")) { // Map
        value = JSON.parseObject(mock, Map.class);
    } else if (mock.startsWith("[")) { // List
        value = JSON.parseObject(mock, List.class);
    } else {
        value = mock;
    }
    // 转换成对应的返回类型
    if (returnTypes != null && returnTypes.length > 0) {
        value = PojoUtils.realize(value, (Class<?>) returnTypes[0], returnTypes.length > 1 ? returnTypes[1] /* 泛型 *
    }
    return value;
}
```

解析值，并转换成对应的返回类型。

5.5 getThrowable

```
private Throwable getThrowable(String throwStr) {
    // 从缓存中，获得 Throwable 对象
    Throwable throwable = throwables.get(throwStr);
    if (throwable != null) {
        return throwable;
    }
    // 不存在，创建 Throwable 对象
    Throwable t;
    try {
        // 获得异常类
        Class<?> bizException = ReflectUtils.forName(throwStr);
        // 获得构造方法
        Constructor<?> constructor = ReflectUtils.findConstructor(bizException, String.class);
        // 创建 Throwable 对象
```

```
            t = (Throwable) constructor.newInstance(new Object[]{" mocked exception for Service degradation. "});
            // 添加到缓存中
            if (throwables.size() < 1000) {
                throwables.put(throwStr, t);
            }
        } catch (Exception e) {
            throw new RpcException("mock throw error :" + throwStr + " argument error.", e);
        }
        return t;
    }
```

代码比较易懂，胖友看代码注释。

5.6 getInvoker

```
    private Invoker<T> getInvoker(String mockService) {
        // 从缓存中，获得 Invoker 对象
        Invoker<T> invoker = (Invoker<T>) mocks.get(mockService);
        if (invoker != null) {
            return invoker;
        }
        // 不存在，创建 Invoker 对象
        // 1. 获得接口类
        Class<T> serviceType = (Class<T>) ReflectUtils.forName(url.getServiceInterface());
        // 2. 若为 `true` `default` ，修改修改为对应接口 + "Mock" 类。这种情况出现在原始 `mock = fail:true` 或 `mock = fo
        if (ConfigUtils.isDefault(mockService)) {
            mockService = serviceType.getName() + "Mock";
        }
        // 3. 获得 Mock 类
        Class<?> mockClass = ReflectUtils.forName(mockService);
        // 4. 校验 Mock 类，实现了接口类
        if (!serviceType.isAssignableFrom(mockClass)) {
            throw new IllegalArgumentException("The mock implemention class " + mockClass.getName() + " not implement int
        }
        try {
            // 5. 创建 Mock 对象
            T mockObject = (T) mockClass.newInstance();
            // 6. 创建 Mock 对应，对应的 Invoker 对象
            invoker = proxyFactory.getInvoker(mockObject, serviceType, url);
            // 7. 添加到缓存
            if (mocks.size() < 10000) {
                mocks.put(mockService, invoker);
            }
            return invoker;
        } catch (InstantiationException e) {
            throw new IllegalStateException("No such empty constructor \"public " + mockClass.getSimpleName() + "()\" in
        } catch (IllegalAccessException e) {
            throw new IllegalStateException(e);
        }
    }
```

代码比较易懂，胖友看代码注释。

6. AbstractInterfaceConfig

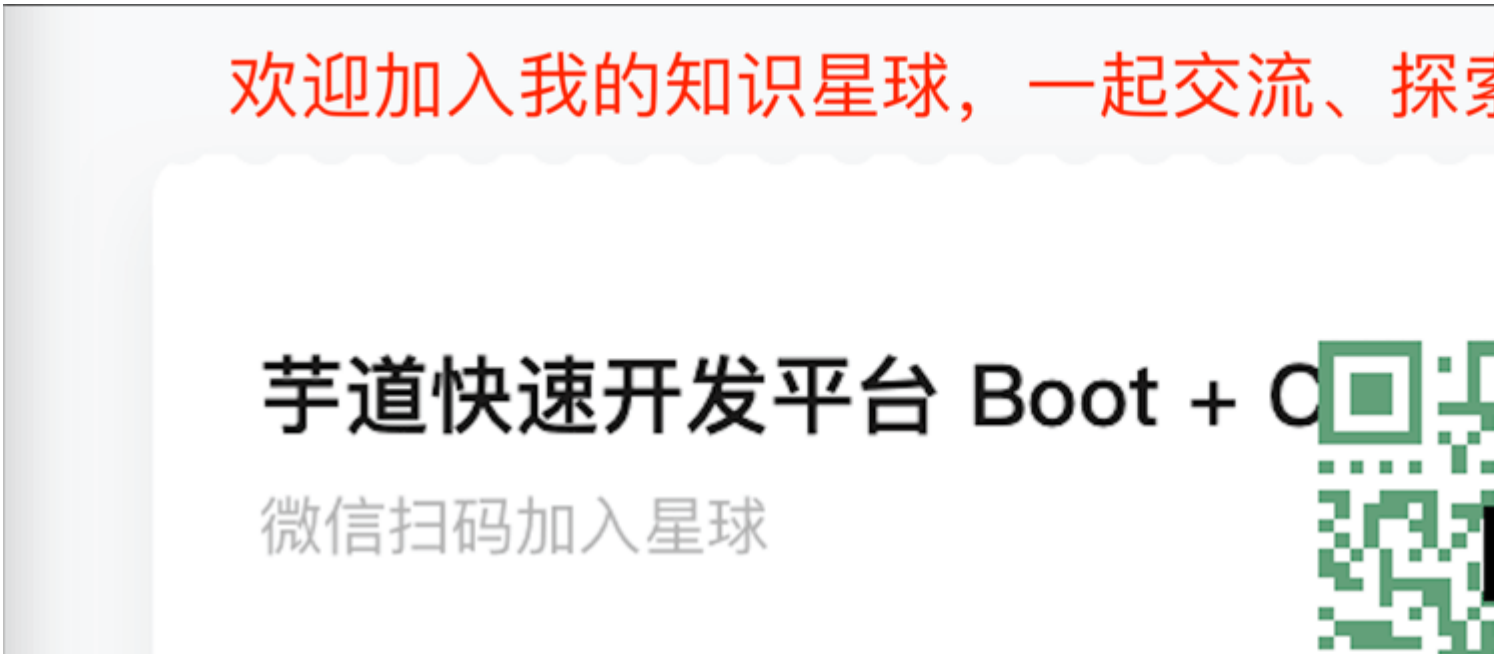
#checkStubAndMock(Class<?> interfaceClass) 方法，校验 Stub 和 Mock 相关的配置。代码如下：

```
protected void checkStubAndMock(Class<?> interfaceClass) {
    // 【省略代码】`local` 配置项的校验，和 `stub` 一样。
    // 【省略代码】`stub` 配置项的校验

    // mock 配置校验
    if (ConfigUtils.isNotEmpty(mock)) {
        if (mock.startsWith(Constants.RETURN_PREFIX)) { // 处理 "return " 开头的情况
            String value = mock.substring(Constants.RETURN_PREFIX.length());
            // 校验 Mock 值，配置正确
            try {
                MockInvoker.parseMockValue(value);
            } catch (Exception e) {
                throw new IllegalStateException("Illegal mock json value in <dubbo:service ... mock=\"" + mock + "\"
            }
        } else {
            // 获得 Mock 类
            Class<?> mockClass = ConfigUtils.isDefault(mock) ? ReflectUtils.forName(interfaceClass.getName() + "Mock"
            // 校验是否实现接口类
            if (!interfaceClass.isAssignableFrom(mockClass)) {
                throw new IllegalStateException("The mock implementation class " + mockClass.getName() + " not implem
            }
            // 校验是否有默认构造方法
            try {
                mockClass.getConstructor();
            } catch (NoSuchMethodException e) {
                throw new IllegalStateException("No such empty constructor \"public " + mockClass.getSimpleName() + "
            }
        }
    }
}
```

代码比较易懂，胖友看代码注释。
从 Mock 配置校验的逻辑我们可以看出，不允许配置 "force:" 和 "fail:" 为开头。 所以，不能通过 Java API 或者 XML ，又或者注解来配置 Mock 规则，只能通过配置规则来开启服务降级。具体的配置方式，参见 《Dubbo 用户指南 —— 服务降级》。
    - 同样，也不允许配置 "throws " 开头。

666. 彩蛋



凌晨快 1 点，集群容错，最后一篇，收尾！

文章目录

回到首页