crazyhzm 265

# Dubbo源码解析（四十一）集群——Mock

dubbo java 阅读约 32 分钟

# 集群——Mock

> 目标：介绍dubbo中集群的Mock，介绍dubbo-cluster下关于服务降级和本地伪装的源码。

# 前言

本文讲解两块内容，分别是本地伪装和服务降级，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。而服务降级则是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。

# 源码分析

## （一）MockClusterWrapper

```
public class MockClusterWrapper implements Cluster {

    private Cluster cluster;

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

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

}
```

该类是服务降级的装饰器类，对Cluster进行了功能增强，增强了服务降级的功能。

## （二）MockClusterInvoker

该类是服务降级中定义降级后的返回策略的实现。

### 1.属性

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

/**
 * 目录
 */
private final Directory<T> directory;

/**
 * invoker对象
 */
private final Invoker<T> invoker;
```

### 2.invoke

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

    // 获得 "mock" 配置项，有多种配置方式
    String value = directory.getUrl().getMethodParameter(invocation.getMethodName(), Constants.MOCK_KEY,
Boolean.FALSE.toString()).trim();
    // 如果没有mock
    if (value.length() == 0 || value.equalsIgnoreCase("false")) {
        //no mock
        // 直接调用
        result = this.invoker.invoke(invocation);
        // 如果强制服务降级
    } else if (value.startsWith("force")) {
        if (logger.isWarnEnabled()) {
            logger.info("force-mock: " + invocation.getMethodName() + " force-mock enabled , url : " +
directory.getUrl());
        }
        //force:direct mock
        // 直接调用 Mock Invoker ，执行本地 Mock 逻辑
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock
        // 失败服务降级
        try {
            // 否则正常调用
```

该方法是定义降级后的返回策略的实现，根据配置的不同来决定不用降级还是强制服务降级还是失败后再服务降级。

## 3.doMockInvoke

```
@SuppressWarnings({"unchecked", "rawtypes"})
private Result doMockInvoke(Invocation invocation, RpcException e) {
    Result result = null;
    Invoker<T> minvoker;

    // 路由匹配 Mock Invoker 集合
    List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
    // 如果mockInvokers为空，则创建一个MockInvoker
    if (mockInvokers == null || mockInvokers.isEmpty()) {
        // 创建一个MockInvoker
        minvoker = (Invoker<T>) new MockInvoker(directory.getUrl());
    } else {
        // 取出第一个
        minvoker = mockInvokers.get(0);
    }
    try {
        // 调用invoke
        result = minvoker.invoke(invocation);
    } catch (RpcException me) {
        // 如果抛出异常，则返回异常结果
        if (me.isBiz()) {
            result = new RpcResult(me.getCause());
        } else {
            throw new RpcException(me.getCode(), getMockExceptionMessage(e, me), me.getCause());
        }
    } catch (Throwable me) {
```

该方法是执行本地Mock,服务降级。

## 4.selectMockInvoker

```
private List<Invoker<T>> selectMockInvoker(Invocation invocation) {
    List<Invoker<T>> invokers = null;
    //TODO generic invoker?
    if (invocation instanceof RpcInvocation) {
        //Note the implicit contract (although the description is added to the interface declaration, but
extensibility is a problem. The practice placed in the attachement needs to be improved)
        // 注意隐式契约（尽管描述被添加到接口声明中，但是可扩展性是一个问题。附件中的做法需要改进）
        ((RpcInvocation) invocation).setAttachment(Constants.INVOCATION_NEED_MOCK, Boolean.TRUE.toString());
        //directory will return a list of normal invokers if Constants.INVOCATION_NEED_MOCK is present in invocation,
otherwise, a list of mock invokers will return.
        // 如果调用中存在Constants.INVOCATION_NEED_MOCK，则目录将返回正常调用者列表，否则，将返回模拟调用者列表。
        try {
            invokers = directory.list(invocation);
        } catch (RpcException e) {
            if (logger.isInfoEnabled()) {
                logger.info("Exception when try to invoke mock. Get mock invokers error for service:"
                        + directory.getUrl().getServiceInterface() + ", method:" + invocation.getMethodName()
                        + ", will contruct a new mock with 'new MockInvoker()'.", e);
            }
        }
    }
    return invokers;
}
```

该方法是路由匹配 Mock Invoker 集合。

## （三）MockInvokersSelector

该类是路由选择器实现类。

### 1.route

```
@Override
public <T> List<Invoker<T>> route(final List<Invoker<T>> invokers,
                                  URL url, final Invocation invocation) throws RpcException {
    // 如果附加值为空，则直接
    if (invocation.getAttachments() == null) {
        // 获得普通的invoker集合
        return getNormalInvokers(invokers);
    } else {
        // 获得是否需要降级的值
        String value = invocation.getAttachments().get(Constants.INVOCATION_NEED_MOCK);
        // 如果为空，则获得普通的Invoker集合
        if (value == null)
            return getNormalInvokers(invokers);
        else if (Boolean.TRUE.toString().equalsIgnoreCase(value)) {
            // 获得MockedInvoker集合
            return getMockedInvokers(invokers);
        }
    }
    return invokers;
}
```

该方法是根据配置来决定选择普通的invoker集合还是mockInvoker集合。

### 2.getMockedInvokers

```
private <T> List<Invoker<T>> getMockedInvokers(final List<Invoker<T>> invokers) {
    // 如果没有MockedInvoker，则返回null
    if (!hasMockProviders(invokers)) {
        return null;
    }
    // 找到MockedInvoker，往sInvokers中加入，并且返回
    List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(1);
    for (Invoker<T> invoker : invokers) {
        if (invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) {
            sInvokers.add(invoker);
        }
    }
    return sInvokers;
}
```

该方法是获得MockedInvoker集合。

### 3.getNormalInvokers

```
private <T> List<Invoker<T>> getNormalInvokers(final List<Invoker<T>> invokers) {
    // 如果没有MockedInvoker，则返回普通的Invoker 集合
    if (!hasMockProviders(invokers)) {
        return invokers;
    } else {
        // 否则 去除MockedInvoker，把普通的Invoker 集合返回
        List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(invokers.size());
        for (Invoker<T> invoker : invokers) {
            // 把不是MockedInvoker的invoker加入sInvokers，返回
            if (!invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) {
                sInvokers.add(invoker);
            }
        }
        return sInvokers;
    }
}
```

该方法是获得普通的Invoker集合，不包含mock的。

### 4.hasMockProviders

```
private <T> boolean hasMockProviders(final List<Invoker<T>> invokers) {
    boolean hasMockProvider = false;
    for (Invoker<T> invoker : invokers) {
        // 如果有一个是MockInvoker，则返回true
        if (invoker.getUrl().getProtocol().equals(Constants.MOCK_PROTOCOL)) {
            hasMockProvider = true;
            break;
        }
    }
    return hasMockProvider;
}
```

该方法是判断是否有MockInvoker。

以上三个类是对服务降级功能的实现，下面两个类是对本地伪装的实现。

## （四）MockProtocol

该类实现了AbstractProtocol接口，是服务

```
final public class MockProtocol extends AbstractProtocol {

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

    @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 {
        // 创建MockInvoker
        return new MockInvoker<T>(url);
    }
}
```

## ( 五 ) MockInvoker

本地伪装的invoker实现类。

### 1.属性

```
/**
 * 代理工厂
 */
private final static ProxyFactory proxyFactory =
ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();
/**
 * mock 与 Invoker 对象的映射缓存
 */
private final static Map<String, Invoker<?>> mocks = new ConcurrentHashMap<String, Invoker<?>>();
/**
 * 异常集合
 */
private final static Map<String, Throwable> throwables = new ConcurrentHashMap<String, Throwable>();

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

### 2.parseMockValue

```
public static Object parseMockValue(String mock, Type[] returnTypes) throws Exception {
    Object value = null;
    // 如果mock为empty，则
    if ("empty".equals(mock)) {
        // 获得空的对象
        value = ReflectUtils.getEmptyObject(returnTypes != null && returnTypes.length > 0 ? (Class<?>)
returnTypes[0] : null);
    } else if ("null".equals(mock)) {
        // 如果为null，则返回null
        value = null;
    } else if ("true".equals(mock)) {
        // 如果为true，则返回true
        value = true;
    } else if ("false".equals(mock)) {
        // 如果为false，则返回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)) {
        // 是数字
        value = JSON.parse(mock);
```

该方法是解析mock值

## 3.invoke

```
@Override
public Result invoke(Invocation invocation) throws RpcException {
    // 获得 `"mock"` 配置项，方法级 > 类级
    String mock = getUrl().getParameter(invocation.getMethodName() + "." + Constants.MOCK_KEY);
    if (invocation instanceof RpcInvocation) {
        ((RpcInvocation) invocation).setInvoker(this);
    }
    // 如果mock为空
    if (StringUtils.isBlank(mock)) {
        // 获得mock值
        mock = getUrl().getParameter(Constants.MOCK_KEY);
    }

    // 如果还是为空。则抛出异常
    if (StringUtils.isBlank(mock)) {
        throw new RpcException(new IllegalAccessException("mock can not be null. url :" + url));
    }
    // 标准化 `"mock"` 配置项
    mock = normalizeMock(URL.decode(mock));
    // 等于 "return " ，返回值为空的 RpcResult 对象
    if (mock.startsWith(Constants.RETURN_PREFIX)) {
        // 分割
        mock = mock.substring(Constants.RETURN_PREFIX.length()).trim();
        try {
            // 获得返回类型
            Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
```

该方法是本地伪装的核心实现，mock分三种，分别是return、throw、自定义的mock类。

## 4.normalizedMock

```
public static String normalizeMock(String mock) {
    // 若为空，直接返回
    if (mock == null) {
        return mock;
    }

    mock = mock.trim();

    if (mock.length() == 0) {
        return mock;
    }

    if (Constants.RETURN_KEY.equalsIgnoreCase(mock)) {
        return Constants.RETURN_PREFIX + "null";
    }

    // 若果为 "true" "default" "fail" "force" 四种字符串，返回default
    if (ConfigUtils.isDefault(mock) || "fail".equalsIgnoreCase(mock) || "force".equalsIgnoreCase(mock)) {
        return "default";
    }

    // fail:throw/return foo => throw/return
    if (mock.startsWith(Constants.FAIL_PREFIX)) {
        mock = mock.substring(Constants.FAIL_PREFIX.length()).trim();
    }
```

该方法是规范化mock值。

## 5.getThrowable

```
public static Throwable getThrowable(String throwstr) {
    // 从异常集合中取出异常
    Throwable throwable = throwables.get(throwstr);
    // 如果不为空，则抛出异常
    if (throwable != null) {
        return throwable;
    }

    try {
        Throwable t;
        // 获得异常类
        Class<?> bizException = ReflectUtils.forName(throwstr);
        Constructor<?> constructor;
        // 获得构造方法
        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);
        }
        return t;
    } catch (Exception e) {
        throw new RpcException("mock throw error :" + throwstr + " argument error.", e);
    }
}
```

该方法是获得异常。

## 6.getInvoker

```
private Invoker<T> getInvoker(String mockService) {
    // 从缓存中，获得 Invoker 对象，如果有，直接缓存。
    Invoker<T> invoker = (Invoker<T>) mocks.get(mockService);
    if (invoker != null) {
        return invoker;
    }

    // 获得服务类型
    Class<T> serviceType = (Class<T>) ReflectUtils.forName(url.getServiceInterface());
    // 获得MockObject
    T mockObject = (T) getMockObject(mockService, serviceType);
    // 创建invoker
    invoker = proxyFactory.getInvoker(mockObject, serviceType, url);
    if (mocks.size() < 10000) {
        // 加入集合
        mocks.put(mockService, invoker);
    }
    return invoker;
}
```

该方法是获得invoker。

### 7.getMockObject

```
public static Object getMockObject(String mockService, Class serviceType) {
    if (ConfigUtils.isDefault(mockService)) {
        mockService = serviceType.getName() + "Mock";
    }

    // 获得类型
    Class<?> mockClass = ReflectUtils.forName(mockService);
    if (!serviceType.isAssignableFrom(mockClass)) {
        throw new IllegalStateException("The mock class " + mockClass.getName() +
                " not implement interface " + serviceType.getName());
    }

    try {
        // 初始化
        return mockClass.newInstance();
    } catch (InstantiationException e) {
        throw new IllegalStateException("No default constructor from mock class " + mockClass.getName(), e);
    } catch (IllegalAccessException e) {
        throw new IllegalStateException(e);
    }
}
```

该方法是获得mock对象。

## 后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了集群中关于mock实现的部分，到这里为止，集群部分就全部讲完了，这是2.6.x版本的集群，那在2.7中对于路由和配置规则都有相应的大改动，我会在之后2.7版本的讲解中讲到。接下来我将开始对序列化模块进行讲解。