Dubbo服务降级

服务调用通常是跨进程的，中间复杂的网络环境，加上远端的业务代码又不受客户端开发人员控制，发生异常几乎是必然的，只是概率分布的高低而已。因此一个合理的微服务接入开发，非常有必要对被引用微服务调用过程预置异常处理逻辑，也即客户端根据业务需要所做的服务降级。

Dubbo中的服务降级实现主要是通过 Mock本地伪装 机制实现的。

Mock 本地伪装

> **NOTE**
>
> Mock 本地伪装机制的实现借助了集群组块的中的路由和容错，若不熟悉这二者的话，请挪步《Dubbo集群 之 容错》和《Dubbo集群 之 路由》一探究竟。

Dubbo使用了 Mock 本地伪装机制实现了服务降级，从表象来看 本地伪装 是 本地存根 的一个子集，但实现方式却全然不同，关于后者具体请参考《Dubbo服务代理》一文。 本地伪装 是面向除 MockClusterWrapper 外的所有 Cluster 实现的（MockClusterWrapper 具类是 Cluster 扩展点的装饰实现），大致思路是：

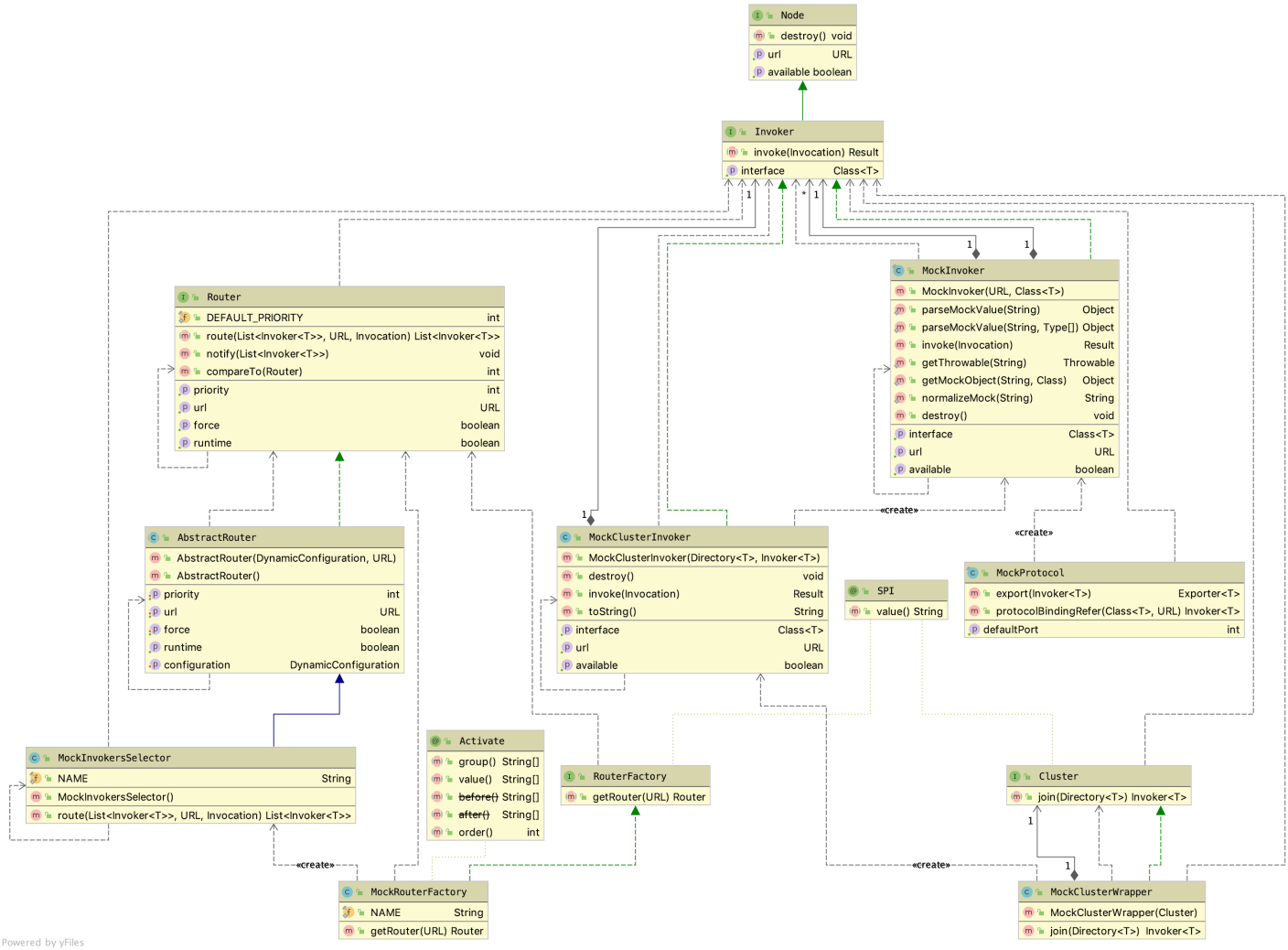
1. 首先在服务发现机制的基础上，由被装饰者 cluster 将实时发现的若干引用实例伪装成一个 Invoker<T> 实例 invoker ；
2. 然后，使用 MockClusterInvoker 装饰 invoker 得到新的被引用微服务实例 wrappedInvoker ；
3. 最后， wrappedInvoker.invoke(invocation) 的执行逻辑中，会根据总线配置 url[[methodName + "."] + "mock"] 确定如何调用 invoker.invoke(invocation) 以及遇异常时所应执行的降级逻辑；

JAVA

```
public class MockClusterWrapper implements Cluster {
    private Cluster cluster;

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

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



MockClusterInvoker<T>

默认情况下，也就是如果消费端开发者没有为被引用微服务配置 Mock 特性，就相当于没有对实例 invoker 进行 MockClusterInvoker 的装饰处理。有一种场景是客户端使用的某个微服务只是预定义好服务接口，尚处于开发过程中，这时如果直接发起RPC调用，可以肯定的是必定会返回空值或者抛出异常，因此Dubbo允许配置 url[[methodName + "."] + "mock"] = "force:" + ("throw" | "return") + XXX ，也即强制使用 Mock 行为，不走远程调用。其它场景则是先 invoker.invoke(invocation) ，遇到非业务 RpcException 异常后再执行对应的 Mock 行为，如果是业务异常，则表示服务端成功接收到了请求，只是业务逻辑处理异常，是需要直接向上抛给上端应用层的。由于异常既可以直接携带在 Result 返回的，也可以 throw 直接抛出，因此最后的 fail-mock 块执行了两次 doMockInvoke(…) 。

JAVA

```
public class MockClusterInvoker<T> implements Invoker<T> {

    private final Directory<T> directory;
    private final Invoker<T> invoker;

    public MockClusterInvoker(Directory<T> directory, Invoker<T> invoker) {
        this.directory = directory;
        this.invoker = invoker;
    }
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        Result result = null;
        String value =
directory.getUrl().getMethodParameter(invocation.getMethodName(),
            MOCK_KEY, Boolean.FALSE.toString()).trim();
        if (value.length() == 0 || "false".equalsIgnoreCase(value)) {
            //no mock
            result = this.invoker.invoke(invocation);
        } else if (value.startsWith("force")) {
            if (logger.isWarnEnabled()) {
                logger.warn("force-mock: " + invocation.getMethodName()
                    + " force-mock enabled , url : " + directory.getUrl());
            }
            result = doMockInvoke(invocation, null);
        } else {
            //fail-mock
            try {
                result = this.invoker.invoke(invocation);
                if(result.getException() != null && result.getException() instanceof
RpcException){
                    RpcException rpcException= (RpcException)result.getException();
                    if(rpcException.isBiz()){ throw  rpcException;}
                    else {
                        result = doMockInvoke(invocation, rpcException);
                    }
                }
            } catch (RpcException e) {
                if (e.isBiz()) { throw e; }
                if (logger.isWarnEnabled()) {
                    logger.warn("fail-mock: " + invocation.getMethodName()
                        + " fail-mock enabled , url : " + directory.getUrl(), e);
                }
                result = doMockInvoke(invocation, e);
            }
        }
        return result;
    }
    ...
}
```

然而就像上述代码所示的，真正处理异常的 Mock 行为是由下述 `doMockInvoke(inv, expt)` 方法负责的，它会首先试图调用 `selectMockInvoker(invocation)` 在当前客户端应用中找到与 `invocation` 特征匹配的特定微服务的所有 `Invoker<T>` 实例，若匹配到，则取其中一个，否则则构

建一 MockInvoker 实例，用于 Mock 处理。和调用方不同的是，关于异常处理的方式反过来了，执行后若是遇到业务异常，则直接返回一个 Result 对象，因此上述源码中的 fail-mock 模块中至少不会因为业务异常陷入 try-catch 循环。

JAVA

```
public class MockClusterInvoker<T> implements Invoker<T> {
    private Result doMockInvoke(Invocation invocation, RpcException e) {
        Result result = null;
        Invoker<T> minvoker;

        List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
        if (CollectionUtils.isEmpty(mockInvokers)) {
            minvoker = (Invoker<T>) new MockInvoker(
                directory.getUrl(), directory.getInterface());
        } else {
            minvoker = mockInvokers.get(0);
        }
        try {
            result = minvoker.invoke(invocation);
        } catch (RpcException me) {
            if (me.isBiz()) {
                result = AsyncRpcResult.newDefaultAsyncResult(me.getCause(),
invocation);
            } else {
                throw new RpcException(me.getCode(),
                    getMockExceptionMessage(e, me), me.getCause());
            }
        } catch (Throwable me) {
            throw new RpcException(getMockExceptionMessage(e, me), me.getCause());
        }
        return result;
    }

    private String getMockExceptionMessage(Throwable t, Throwable mt) {
        String msg = "mock error : " + mt.getMessage();
        if (t != null) {
            msg = msg + ", invoke error is :" + StringUtils.toString(t);
        }
        return msg;
    }
    ...
}
```

方法 selectMockInvoker(invocation) 的目的是利用 directory 服务发现搜集到所有 url.protocol = "mock" 的微服务引用实例，涉及到的 MockProtocol 及Mock版的 Router 实现将在下文相关章节讨论。

```java
public class MockClusterInvoker<T> implements Invoker<T> {
    private List<Invoker<T>> selectMockInvoker(Invocation invocation) {
        List<Invoker<T>> invokers = null;
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation) invocation).setAttachment(INVOCATION_NEED_MOCK,
Boolean.TRUE.toString());
            try {
                invokers = directory.list(invocation);
            } catch (RpcException e) {
                if (logger.isInfoEnabled()) {
                    logger.info("Exception when try to invoke mock. Get mock invokers"
                        + " error for service:" +
directory.getUrl().getServiceInterface()
                        + ", method:" + invocation.getMethodName()
                        + ", will contruct a new mock with 'new MockInvoker()'.", e);
                }
            }
        }
        return invokers;
    }
    ...
}
```

MockInvoker<T>

显然上述 MockClusterInvoker<T> 只负责了 Mock实现 中的总体流程逻辑，具体细节却没有涉及，而这恰恰是 MockInvoker<T> 的职责。

进一步分析实现前，我们先看看有关 Mock本地伪装 的具体应用。首先总线配置可以是方法级别的，也即上述出现的 url[[methodName + "."] + "mock"] 表示；其次，总线配置支持 ("force" | "fail") + ":" 前缀，本地的 Mock行为 于前者是强制执行的，远程的RPC请求不会执行，于后者则只有远程的RPC请求发生异常时才执行，相当于默认的没有配前缀的情况；最后，指定的 Mock行为 有如下几种方式：

- 简洁配置：("return" | "throw" | ("true" | "default" | "fail" | "force"))，前者会相当于 "return null"，中者则抛出一个默认的 RPCException 异常，而后者会归化为 "default"，需要客户端提供一个在服务接口全名后附有 Mock 后缀的接口实现；
- 返回指定字面值："return" + ("empty" | "null" | "true" | "false" | JSON格式字符串 | 字符串字面量)；
    - 其中 "empty" 代表空，基本类型的默认值，或者集合类的空值，而 JSON格式字符串 会被反序列化得到相应接口方法返回类型的对象。
- 指定目标：("throw" + XXXException | "return" + XXXInterfaceImpl)，这两种情况均需存在对应的匹配全类名的类，或为异常，或为接口实现；

```xml
<!--①-->
<dubbo:reference interface="com.foo.BarService" mock="com.foo.BarServiceMock" />

<!--②-->
<dubbo:reference id="demoService" check="false" interface="com.foo.BarService">
    <dubbo:parameter key="sayHello.mock" value="return fake"/>
</dubbo:reference>
```

可见 Mock行为 实际上分为 3 种：1） Mock Value，直接返回值；2） Mock Object，将调用委托给实现了同一接口的类；3） Mock Throwable，抛错指定类型的异常。总体逻辑如下

```java
final public class MockInvoker<T> implements Invoker<T> {
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        String mock = getUrl().getParameter(invocation.getMethodName() + "." +
MOCK_KEY);
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation) invocation).setInvoker(this);
        }
        if (StringUtils.isBlank(mock)) {
            mock = getUrl().getParameter(MOCK_KEY);
        }

        if (StringUtils.isBlank(mock)) {
            throw new RpcException(new IllegalAccessException("mock can not be null.
url :" + url));
        }
        mock = normalizeMock(URL.decode(mock));
        if (mock.startsWith(RETURN_PREFIX)) {
            //Mock Value 返回
        } else if (mock.startsWith(THROW_PREFIX)) {
            //Mock Throwable抛错
        } else {
            //Mock Object 回调
        }
    }
    ...
}
```

Mock Value 返回

相对而言，返回具体值的这种场景，涉及到类型操作和构建目标类型对象，细节挺多，感兴趣的可以按图索骥看看具体实现。基本步骤是先利用 invocation 获取到接口方法的出参类型，然后结合该类型和 Mock 字面值 调用 parseMockValue(…) 转换为目标类型的值，最后调用 AsyncRpcResult.newDefaultAsyncResult(value, invocation) 填充并构建一个 AsyncRpcResult 类型对象。

JAVA

```java
final public class MockInvoker<T> implements Invoker<T> {
    ...
    public static Object parseMockValue(String mock) throws Exception {
        return parseMockValue(mock, null);
    }

    public static Object parseMockValue(String mock, Type[] returnTypes) throws
Exception {
        Object value = null;
        ...
        return value;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        ...
        //Mock Value 返回
        mock = mock.substring(RETURN_PREFIX.length()).trim();
        try {
            Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
            Object value = parseMockValue(mock, returnTypes);
            return AsyncRpcResult.newDefaultAsyncResult(value, invocation);
        } catch (Exception ew) {
            throw new RpcException("mock return invoke error. method :" +
invocation.getMethodName()
                    + ", mock:" + mock + ", url: " + url, ew);
        }
        ...
    }
}
```

Mock Object 回调

显然，Mock配置为 `url[[("force" | "fail") + ":"] + ("default" | "return " + XXXInterfaceImpl)]` 时，需要使用反射实例化被引用微服务接口 `serviceType` 的一个本地实现，如下述源码所示，先使用全类名获得对应的类元数据 `mockClass` 后，还需要判断它是否实现自 `serviceType` 。

```java
final public class MockInvoker<T> implements Invoker<T> {
    public static Object getMockObject(String mockService, Class serviceType) {
        if (ConfigUtils.isDefault(mockService)) {
            mockService = serviceType.getName() + "Mock";
        }

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

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

Mock Object 是目标微服务接口的本地伪装实现，和本地存根实现不同的是，后者是接口的一个装饰者实现，而本地伪装认为发往对端的RPC调用已经失败，或者被 force 禁用了。尽管在有了伪装实现的全类名后利用反射获得了目标类元信息和目标类的实例 mockObj ，可以从入参 invocation 中获得针对 mockObj 的当前被调用方法名以及入参信息，但还是没法像硬编码那样可以直接发方法调用，也就是针对目标方法的调用依然只能依靠反射机制。这里比较聪明的处理方式是直接利用既有代理层的 ProxyFactory 生成一个 AbstractProxyInvoker 实例（一般只用于服务端），由其利用反射机制将当前 invoke(invocation) 调用转换为对 mockObj 指定方法的调用。

另外，针对 mockObj 的方法调用并不是一次性的，也不仅是只针对它的其中一个方法做调用，为了避免不必要的CPU开销，特意申明了一个全局的 Map<String, Invoker<?>> 类型的 MOCK_MAP 缓存容器，Key键是 mockObj 对应类的全类名，而 mockObj 缓存在Value值（一个 AbstractProxyInvoker 实现的对象）中。

> **NOTE**
>
> 正常流程中客户端发起的 invoke(invocation) 调用中，最内核的 Invoker 执行体是 DubboInvoker ，而Mock伪装中的则是 AbstractProxyInvoker 的匿名实现。

JAVA

```
final public class MockInvoker<T> implements Invoker<T> {
    ...
    private final static ProxyFactory PROXY_FACTORY = ExtensionLoader.
        getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();

    private final static Map<String, Invoker<?>> MOCK_MAP =
        new ConcurrentHashMap<String, Invoker<?>>();

    private Invoker<T> getInvoker(String mockService) {
        Invoker<T> invoker = (Invoker<T>) MOCK_MAP.get(mockService);
        if (invoker != null) {
            return invoker;
        }

        Class<T> serviceType = (Class<T>)
ReflectUtils.forName(url.getServiceInterface());
        T mockObject = (T) getMockObject(mockService, serviceType);
        invoker = PROXY_FACTORY.getInvoker(mockObject, serviceType, url);
        if (MOCK_MAP.size() < 10000) {
            MOCK_MAP.put(mockService, invoker);
        }
        return invoker;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        ...
        //Mock Object 回调
        try {
            Invoker<T> invoker = getInvoker(mock);
            return invoker.invoke(invocation);
        } catch (Throwable t) {
            throw new RpcException("Failed to create mock implementation class " +
mock, t);
        }
        ...
    }
}
```

Mock Throwable 抛错

同样，Mock配置为 `url[[("force" | "fail") + ":"] + "throw " + [XXXException]]` 时，也需要实例化一个异常类。一般而言，一个应用会根据需要对异常进行分类，只会声明少数几个异常类，因而声明了对应的Map容器，规避反复实例化所带来的开销。对应配置的异常类型必须带有一个字符串类型的构造函数，所有的异常实例的报错信息均一样：

JAVA

```
final public class MockInvoker<T> implements Invoker<T> {
    ...
    private final static Map<String, Throwable> THROWABLE_MAP =
            new ConcurrentHashMap<String, Throwable>();

    public static Throwable getThrowable(String throwstr) {
        Throwable throwable = THROWABLE_MAP.get(throwstr);
        if (throwable != null) {
            return throwable;
        }

        try {
            Throwable t;
            Class<?> bizException = ReflectUtils.forName(throwstr);
            Constructor<?> constructor;
            constructor = ReflectUtils.findConstructor(bizException, String.class);
            t = (Throwable) constructor.newInstance(new Object[]{
                "mocked exception for service degradation."});
            if (THROWABLE_MAP.size() < 1000) {
                THROWABLE_MAP.put(throwstr, t);
            }
            return t;
        } catch (Exception e) {
            throw new RpcException("mock throw error :" + throwstr + " argument
error.", e);
        }
    }
}
```

异常处理时的调用方逻辑如下，无论是否指定 XXXException 时，都会抛出 RpcException 异常，只是当指明异常类名时，对应的异常被标记为业务异常了， getThrowable(mock) 所得沦为内嵌异常详情了。

JAVA

```
final public class MockInvoker<T> implements Invoker<T> {
    ...
    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        ...
        //Mock Throwable抛错
        mock = mock.substring(THROW_PREFIX.length()).trim();
        if (StringUtils.isBlank(mock)) {
            throw new RpcException("mocked exception for service degradation.");
        } else { // user customized class
            Throwable t = getThrowable(mock);
            throw new RpcException(RpcException.BIZ_EXCEPTION, t);
        }
        ...
    }
}
```

MockInvokersSelector 路由实现

为了保证微服务的高可用性，实现诸如不同机房网段隔离、黑白名单、读写分离、前后台分离等特性，通常会为一个微服务在不同的机房或者网段上部署多个实例，客户端则根据业务需要为被引用微服务配置所需的路由规则，利用它们过滤出被引用微服务的目标实例集合。正如《Dubbo集群 之 路由》一文所言，通常用于过滤的路由器有多个，最终得到的集合是这些路由器作用的总和。

本章节的 `MockInvokersSelector` 其实是一个路由器实现，目的是为客户端筛选出本地伪装的被引用微服务实现。相对应地，还有一个专门用于创建其实例的工厂类 `MockRouterFactory` ，源码中的 `@Activate` 注解说明 `MockRouterFactory` 是自动激活的，换言之，它所创建 `MockInvokersSelector` 路由器是Dubbo路由链 `RouterChain` 内置实例，所有需要用到路由功能的地方，只要 `inv["invocation.need.mock"] = true` ，它均会发生作用。

JAVA

```java
@Activate
public class MockRouterFactory implements RouterFactory {
    public static final String NAME = "mock";

    @Override
    public Router getRouter(URL url) {
        return new MockInvokersSelector();
    }
}
```

上文中提到的 `selectMockInvoker(inv)` 方法被间接调用过好几次， `doMockInvoke(inv, expt)` 方法的开头会视图用它获得满足 `url.protocol = "mock"` 的所有被引用微服务实例集合，当获得结果为空时才会执行本地 `Mock 伪装` 行为。

`MockInvokersSelector` 最核心的一段代码如下所示，也就是说类似Mock本地伪装示例中的 `url[[methodName + "."] + "mock"]` 配置引起了 `MockClusterInvoker#selectMockInvoker(inv)` 设置 `inv["invocation.need.mock"] = true` 参数，而后者又间接引起 `MockInvokersSelector` 去筛选 `url.protocol = "mock"` 的引用实例。反之，如果没有前面 `refer` 时的配置，也即不用 `Mock 本地伪装` 时，所有配置了 `url.protocol = "mock"` 的引用实例会被剔除掉。

JAVA

```java
public class MockInvokersSelector extends AbstractRouter {

    public static final String NAME = "MOCK_ROUTER";
    private static final int MOCK_INVOKERS_DEFAULT_PRIORITY = Integer.MIN_VALUE;

    public MockInvokersSelector() {
        this.priority = MOCK_INVOKERS_DEFAULT_PRIORITY;
    }

    @Override
    public <T> List<Invoker<T>> route(final List<Invoker<T>> invokers,
                                      URL url, final Invocation invocation) throws
RpcException {
        if (CollectionUtils.isEmpty(invokers)) {
            return invokers;
        }

        if (invocation.getAttachments() == null) {
            return getNormalInvokers(invokers);
        } else {
            String value = invocation.getAttachments().get(INVOCATION_NEED_MOCK);
            if (value == null) {
                return getNormalInvokers(invokers);
            } else if (Boolean.TRUE.toString().equalsIgnoreCase(value)) {
                return getMockedInvokers(invokers);
            }
        }
        return invokers;
    }
    ...
}
```

上述源码表示优先级的 `priority` 被设置成 `Integer.MIN_VALUE`，这说明了 `MockInvokersSelector` 这个路由器实现在每次路由链执行时是优先被采用的。

最后留下的上述被用到的其它代码则非常简单，一个被引用微服务的实例列表，应用 `hasMockProviders(invokers)` 能够判别出是否含有满足 `url.protocol = "mock"` 的伪装实例，， `getMockedInvokers(invokers)` 获取所有伪装实例，而 `getNormalInvokers(invokers)` 则取得剔除伪装实例后的所有实例。

JAVA

```java
public class MockInvokersSelector extends AbstractRouter {
    private <T> List<Invoker<T>> getMockedInvokers(final List<Invoker<T>> invokers) {
        if (!hasMockProviders(invokers)) {
            return null;
        }
        List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(1);
        for (Invoker<T> invoker : invokers) {
            if (invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL)) {
                sInvokers.add(invoker);
            }
        }
        return sInvokers;
    }

    private <T> List<Invoker<T>> getNormalInvokers(final List<Invoker<T>> invokers) {
        if (!hasMockProviders(invokers)) {
            return invokers;
        } else {
            List<Invoker<T>> sInvokers = new ArrayList<Invoker<T>>(invokers.size());
            for (Invoker<T> invoker : invokers) {
                if (!invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL)) {
                    sInvokers.add(invoker);
                }
            }
            return sInvokers;
        }
    }

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

TODO::其它服务降级方式

完结