Dubbo的mock和集群容错源码分析

1、mock的源码分析

1.1、功能描述

在前面的高级用法中我们讲过mock的使用方式以及作用，在这里不再赘述，这里针对mock的集中配置形式来分析一下mock的实现原理。

1.2、mock = "force:return gpwy"

1.2.1、使用场景

这种一般不会出现在代码配置中，一般都是在服务治理的时候进行配置的，如果指定后端接口有问题了，可以在dubbo-admin中配置该接口对应的某方法进行强制降级。

1.2.2、源码分析

有关mock的实现逻辑全部都在MockClusterInvoker中，当我们用代理对象调用的时候，代码会走到代理的advice类，也就是InvokerInvocationHandler中，然后走到MigrationInvoker，然后走到了MockClusterInvoker当中，调到了其invoke方法，代码如下：

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

    //获取url中的mock参数
    String value = getUrl().getMethodParameter(invocation.getMethodName(), MOCK_KEY,
Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || "false".equalsIgnoreCase(value)) {
        //no mock
        //如果没mock则直接走后面调用逻辑
        result = this.invoker.invoke(invocation);
    } else if (value.startsWith("force")) {
        //如果是force开头
        if (logger.isWarnEnabled()) {
            logger.warn("force-mock: " + invocation.getMethodName() + " force-mock enabled
, url : " + getUrl());
        }
        //强制降级，调用mock的实现类逻辑
        //force:direct mock
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock
        try {
            result = this.invoker.invoke(invocation);

            //fix:#4585
```

```
            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 : " + getUrl(), e);
            }
            result = doMockInvoke(invocation, e);
        }
    }
    return result;
}
```

从上述代码我们可以看到，MockClusterInvoker当中就是走了三套逻辑：

1、没有配置mock的情况

2、mock="force:"的情况

3、配置了mock的其他情况

我们都知道如果配置了force:就代表要进行强制降级，就不会走后端的rpc调用了，所以这里是直接调用到了

```
result = doMockInvoke(invocation, null);
```

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

    //选择一个MockInvoker的实例，这里是选不到的
    List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);
    if (CollectionUtils.isEmpty(mockInvokers)) {
        //所以代码会走这里，创建一个MockInvoker对象
        minvoker = (Invoker<T>) new MockInvoker(getUrl(), directory.getInterface());
    } else {
        minvoker = mockInvokers.get(0);
    }
    try {
        //调用mock的实现类方法
        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;
}
```

我们再看看mockInvoker的invoke方法

```
@Override
public Result invoke(Invocation invocation) throws RpcException {
    if (invocation instanceof RpcInvocation) {
        ((RpcInvocation) invocation).setInvoker(this);
    }
    String mock = null;
    if (getUrl().hasMethodParameter(invocation.getMethodName())) {
        mock = getUrl().getParameter(invocation.getMethodName() + "." + MOCK_KEY);
    }
    if (StringUtils.isBlank(mock)) {
        //获取配置的mock的属性值
        mock = getUrl().getParameter(MOCK_KEY);
    }

    if (StringUtils.isBlank(mock)) {
        throw new RpcException(new IllegalAccessException("mock can not be null. url :" +
url));
    }
    //把force:前缀去掉，获取后面的值
    mock = normalizeMock(URL.decode(mock));
    //如果是return开头
    if (mock.startsWith(RETURN_PREFIX)) {
        //获取return后面的值
        mock = mock.substring(RETURN_PREFIX.length()).trim();
        try {
            //获取返回值类型
            Type[] returnTypes = RpcUtils.getReturnTypes(invocation);
            //把return后面的值包装成返回值类型
            Object value = parseMockValue(mock, returnTypes);
            //注解把结果返回没走后端rpc调用
            return AsyncRpcResult.newDefaultAsyncResult(value, invocation);
        } catch (Exception ew) {
            throw new RpcException("mock return invoke error. method :" +
invocation.getMethodName()
                    + ", mock:" + mock + ", url: " + url, ew);
        }
        //如果是throw
    } else if (mock.startsWith(THROW_PREFIX)) {
        mock = mock.substring(THROW_PREFIX.length()).trim();
```

```
        if (StringUtils.isBlank(mock)) {
            throw new RpcException("mocked exception for service degradation.");
        } else { // user customized class
            //获取异常实例
            Throwable t = getThrowable(mock);
            //直接往上抛异常
            throw new RpcException(RpcException.BIZ_EXCEPTION, t);
        }
    } else { //impl mock
        //mock实现类的方式
        try {
            Invoker<T> invoker = getInvoker(mock);
            //调用mock实例
            return invoker.invoke(invocation);
        } catch (Throwable t) {
            throw new RpcException("Failed to create mock implementation class " + mock,
t);
        }
    }
}
```

我们看一下normalizeMock方法

```
public static String normalizeMock(String mock) {
    if (mock == null) {
        return mock;
    }

    mock = mock.trim();

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

    //如果是只有一个return 则加上一个return null
    if (RETURN_KEY.equalsIgnoreCase(mock)) {
        return RETURN_PREFIX + "null";
    }

    if (ConfigUtils.isDefault(mock) || "fail".equalsIgnoreCase(mock) ||
"force".equalsIgnoreCase(mock)) {
        return "default";
    }

    if (mock.startsWith(FAIL_PREFIX)) {
        mock = mock.substring(FAIL_PREFIX.length()).trim();
    }

    //把force:去掉
    if (mock.startsWith(FORCE_PREFIX)) {
        mock = mock.substring(FORCE_PREFIX.length()).trim();
    }
```

```
    if (mock.startsWith(RETURN_PREFIX) || mock.startsWith(THROW_PREFIX)) {
        mock = mock.replace('`', '"');
    }

    return mock;
}
```

我们获取到mock的返回值内容后需要把该返回值包装成方法返回值类型，所以这里必须要有一个返回值类型的包装，我们看一下parseMockValue方法：

```
public static Object parseMockValue(String mock, Type[] returnTypes) throws Exception {
    Object value = null;
    if ("empty".equals(mock)) {
        value = ReflectUtils.getEmptyObject(returnTypes != null && returnTypes.length > 0 ?
(Class<?>) returnTypes[0] : null);
    } else if ("null".equals(mock)) {
        value = null;
    } else if ("true".equals(mock)) {
        value = true;
    } else if ("false".equals(mock)) {
        value = false;
    } else if (mock.length() >= 2 && (mock.startsWith("\"") && mock.endsWith("\"")
            || mock.startsWith("\'") && mock.endsWith("\'"))) {
        value = mock.subSequence(1, mock.length() - 1);
    } else if (returnTypes != null && returnTypes.length > 0 && returnTypes[0] ==
String.class) {
        value = mock;
    } else if (StringUtils.isNumeric(mock, false)) {
        value = JSON.parse(mock);
    } else if (mock.startsWith("{")) {
        value = JSON.parseObject(mock, Map.class);
    } else if (mock.startsWith("[")) {
        value = JSON.parseObject(mock, List.class);
    } else {
        value = mock;
    }
    if (ArrayUtils.isNotEmpty(returnTypes)) {
        value = PojoUtils.realize(value, (Class<?>) returnTypes[0], returnTypes.length > 1
? returnTypes[1] : null);
    }
    return value;
}
```

从上面的逻辑来看，如果是force:return 则会不走rpc直接返回一个结果，然后把这个结果包装成方法的返回值类型。

1.3、mock = "true"

1.3.1、使用场景

如果配置的是mock="true"就是一种约定俗成的方式，那么这种方式就代表会走远程调用，然后远程调用如果出现了RpcException的时候就会掉到降级逻辑，这个降级逻辑的定义必须满足两点：

1、类名必须是接口名+"Mock"

2、类必须定义在接口的同包名下

配置如下：

```
@DubboReference(check = false,mock = "true")
```

1.3.2、源码分析

在MockClusterInvoker中就会走这个逻辑：

```
//fail-mock
try {
    //后端接口调用
    result = this.invoker.invoke(invocation);

    //fix:#4585
    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
: " + getUrl(), e);
    }
    result = doMockInvoke(invocation, e);
}
```

从上面逻辑来看，会先进行invoke调用，如果后端调用有问题则会被catch捕获然后走doMockInvoke逻辑进行降级处理。我们来看看doMockInvoke中的逻辑；在MockInvoker中就会走到实现类mock逻辑；

```
//mock实现类的方式
try {
    //获取Invoker对象
    Invoker<T> invoker = getInvoker(mock);
    //调用mock实例
    return invoker.invoke(invocation);
} catch (Throwable t) {
    throw new RpcException("Failed to create mock implementation class " + mock, t);
}
```

重点看一下getInvoker方法：

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

    //接口类型
    Class<T> serviceType = (Class<T>) ReflectUtils.forName(url.getServiceInterface());
    //核心代码 ，获取Mock实现类实例
    T mockObject = (T) getMockObject(mockService, serviceType);
    //获取用于调用Mock实例类的invoker对象
    invoker = PROXY_FACTORY.getInvoker(mockObject, serviceType, url);
    if (MOCK_MAP.size() < 10000) {
        MOCK_MAP.put(mockService, invoker);
    }
    return invoker;
}
```

getMockObject逻辑

```
public static Object getMockObject(String mockService, Class serviceType) {
    //如果配置的是true或者default ，则走默认mock实现类
    boolean isDefault = ConfigUtils.isDefault(mockService);
    if (isDefault) {
        //默认实现类就是  :  接口完整限定名+Mock
        mockService = serviceType.getName() + "Mock";
    }

    Class<?> mockClass;
    try {
        // 如果 mockService不是配置的true，如果是配置的类的完整限定名
        mockClass = ReflectUtils.forName(mockService);
    } catch (Exception e) {
        if (!isDefault) {// does not check Spring bean if it is default config.
            ExtensionFactory extensionFactory =

 ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension();
            Object obj = extensionFactory.getExtension(serviceType, mockService);
            if (obj != null) {
                return obj;
            }
        }
        throw new IllegalStateException("Did not find mock class or instance "
                + mockService
                + ", please check if there's mock class or instance implementing interface
"
                + serviceType.getName(), e);
    }
    if (mockClass == null || !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);
    }
}
```

从上面的代码我们可以看到，获取到实现类的实例的方式就是两种：

1、如果配置的是true，实现类就是：接口名+"Mock"

2、直接配置的类的完整限定名

在看看获取invoker的代码：

```
invoker = PROXY_FACTORY.getInvoker(mockObject, serviceType, url);
```

这个代码最终会走到JavassistProxyFactory类的getInvoker方法中，spi的方式获取实例，这里就不再赘述

```
@Override
public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
    // TODO Wrapper cannot handle this scenario correctly: the classname contains '$'
    final Wrapper wrapper = Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0
? proxy.getClass() : type);
    return new AbstractProxyInvoker<T>(proxy, type, url) {
        @Override
        protected Object doInvoke(T proxy, String methodName,
                                  Class<?>[] parameterTypes,
                                  Object[] arguments) throws Throwable {
            return wrapper.invokeMethod(proxy, methodName, parameterTypes, arguments);
        }
    };
}
```

这里是先生成了一个代理类wrapper，这个代理类中有一个invokeMethod方法，只要你传给他要掉的类的实例，方法名称，参数类型和参数列表就可以完成方法的调用，然后AbstractProxyInvoker持有了wrapper的引用，我们如果需要调用一个类的方法，只有用invoker对象调用invoke方法就可以了。这里为什么能调到MockServiceMock就不再赘述了。

1.4、mock = "xx.LocalMockService"

1.4.1、使用场景

同mock=“true”

配置：

```
@DubboReference(check = false,mock = "cn.enjoy.mock.LocalMockService")
```

1.4.2、源码分析

同mock="true"

1.5、mock = "throw xx"

1.5.1、使用场景

当调用接口出现问题后需要直接抛出异常的情况可以实现它

配置如下：

@DubboReference(check = false,mock = "throw java.lang.RuntimeException")

1.5.2、源码分析

```
else if (mock.startsWith(THROW_PREFIX)) {
            mock = mock.substring(THROW_PREFIX.length()).trim();
            if (StringUtils.isBlank(mock)) {
                throw new RpcException("mocked exception for service degradation.");
            } else { // user customized class
                //获取异常实例
                Throwable t = getThrowable(mock);
                //直接往上抛异常
                throw new RpcException(RpcException.BIZ_EXCEPTION, t);
            }
        }
```

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

    try {
        Throwable t;
        //反射异常类
        Class<?> bizException = ReflectUtils.forName(throwstr);
        Constructor<?> constructor;
        constructor = ReflectUtils.findConstructor(bizException, String.class);
        //异常实例化
        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);
    }
```

```
}
```

从源码分析来看，这里是注解获取到了配置的异常类的字符串，然后反射实例化，然后把这个异常直接往上抛了。

2、集群容错源码

前面我分析了MockClusterInvoker中完成了的Mock逻辑的正规过程，现在我们流往下走，流程走到集群容错的invoker对象中来了，下面我们来分析集群容错的逻辑

2.1、功能描述

集群调用失败时，Dubbo 提供的容错方案

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。

我们来分析一下各种集群容错策略的不同

2.2、FailoverClusterInvoker

2.2.1、使用场景

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。该配置为缺省配置

2.2.2、源码分析

当调用invoke方法进行调用流转时，如果走到了集群容错的逻辑，那么先会走到他们的父类中，父类里面定义了如何获取服务列表和获取负载均衡算法的逻辑，从父类的方法中就可以获取到这两种东西，然后有一个doInvoke方法是一个抽象的方法，该方法是一个钩子方法，采用了模板设计模式，这样的话就会勾到子类的逻辑当中去，是哪个子类的实例就会勾到哪个实例的中去，不同的实例是通过配置参数来决定的：

配置如下：

cluster = "failover"

```
@DubboReference(check = false/*,url = "dubbo://localhost:20880"*/,retries = 3,timeout =
6000,cluster = "failover",loadbalance = "random")
```

```
@Override
public Result invoke(final Invocation invocation) throws RpcException {
    //判断是否销毁
    checkWhetherDestroyed();

    // binding attachments into invocation.
    //        Map<String, Object> contextAttachments =
RpcContext.getClientAttachment().getObjectAttachments();
    //        if (contextAttachments != null && contextAttachments.size() != 0) {
    //            ((RpcInvocation)
invocation).addObjectAttachmentsIfAbsent(contextAttachments);
    //        }

    //获取服务列表
```

```
    List<Invoker<T>> invokers = list(invocation);
    //spi 获取负载均衡类实例
    LoadBalance loadbalance = initLoadBalance(invokers, invocation);
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    return doInvoke(invocation, invokers, loadbalance);
}
```

以上是父类的流程逻辑，获取到了服务列表，spi的方式获取到了负载均衡算法。

```
public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    List<Invoker<T>> copyInvokers = invokers;
    //invokers校验
    checkInvokers(copyInvokers, invocation);
    String methodName = RpcUtils.getMethodName(invocation);
    //计算调用次数
    int len = calculateInvokeTimes(methodName);
    // retry loop.
    RpcException le = null; // last exception.
    //记录已经调用过了的服务列表
    List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyInvokers.size()); // invoked
invokers.
    Set<String> providers = new HashSet<String>(len);
    for (int i = 0; i < len; i++) {
        //Reselect before retry to avoid a change of candidate `invokers`.
        //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
        //如果掉完一次后，服务列表更新了，再次获取服务列表
        if (i > 0) {
            checkWhetherDestroyed();
            copyInvokers = list(invocation);
            // check again
            checkInvokers(copyInvokers, invocation);
        }
        //根据负载均衡算法，选择一个服务调用
        Invoker<T> invoker = select(loadbalance, invocation, copyInvokers, invoked);
        //记录已经调用过的invoker
        invoked.add(invoker);
        RpcContext.getServiceContext().setInvokers((List) invoked);
        try {
            //具体的服务调用逻辑
            Result result = invokeWithContext(invoker, invocation);
            if (le != null && logger.isWarnEnabled()) {
                logger.warn("Although retry the method " + methodName
                        + " in the service " + getInterface().getName()
                        + " was successful by the provider " +
invoker.getUrl().getAddress()
                        + ", but there have been failed providers " + providers
                        + " (" + providers.size() + "/" + copyInvokers.size()
                        + ") from the registry " + directory.getUrl().getAddress()
                        + " on the consumer " + NetUtils.getLocalHost()
                        + " using the dubbo version " + Version.getVersion() + ". Last
error is: "
                        + le.getMessage(), le);
```

```
            }
            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(le.getCode(), "Failed to invoke the method "
            + methodName + " in the service " + getInterface().getName()
            + ". Tried " + len + " times of the providers " + providers
            + " (" + providers.size() + "/" + copyInvokers.size()
            + ") from the registry " + directory.getUrl().getAddress()
            + " on the consumer " + NetUtils.getLocalHost() + " using the dubbo version "
            + Version.getVersion() + ". Last error is: "
            + le.getMessage(), le.getCause() != null ? le.getCause() : le);
}
```

以上就是FailoverClusterInvoker的doinvoke的核心逻辑，可以看到它是通过一个for循环的方式来重试调用的，重试的次数是通过retries属性来配置的。如果重试N次后还是调用失败，那么注解就throw了一个RpcException了。

2.3、FailfastClusterInvoker

2.3.1、使用场景

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

2.3.2、源码分析

```
@Override
public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    checkInvokers(invokers, invocation);
    //负载均衡选择一个invoker
    Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
    try {
        //掉后端接口
        return invokeWithContext(invoker, invocation);
    } catch (Throwable e) {
        if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz exception.
            throw (RpcException) e;
        }
        //如果调用出现异常，直接把异常包装成RpcException往上抛了
        throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() :
0,
                "Failfast invoke providers " + invoker.getUrl() + " " +
loadbalance.getClass().getSimpleName()
                        + " for service " + getInterface().getName()
```

```
                        + " method " + invocation.getMethodName() + " on consumer " +
NetUtils.getLocalHost()
                        + " use dubbo version " + Version.getVersion()
                        + ", but no luck to perform the invocation. Last error is: " +
e.getMessage(),
                e.getCause() != null ? e.getCause() : e);
    }
}
```

可以看到这种集群容错就是如果出现异常不会重试直接往上抛异常了。

2.4、FailsafeClusterInvoker

2.4.1、使用场景

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

2.4.2、源码分析

```
@Override
public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    try {
        checkInvokers(invokers, invocation);
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        return invokeWithContext(invoker, invocation);
    } catch (Throwable e) {
        //可以看到这里，如果调用有异常了，这里是直接打印日志然后返回结果，相当于忽略了异常
        logger.error("Failsafe ignore exception: " + e.getMessage(), e);
        return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); // ignore
    }
}
```

可以看到这里，如果调用有异常了，这里是直接打印日志然后返回结果，相当于忽略了异常.

2.5、FailbackClusterInvoker

2.5.1、使用场景

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

2.5.2、源码分析

```
@Override
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    Invoker<T> invoker = null;
    try {
        checkInvokers(invokers, invocation);
        invoker = select(loadbalance, invocation, invokers, null);
        return invokeWithContext(invoker, invocation);
    } catch (Throwable e) {
```

```
        logger.error("Failback to invoke method " + invocation.getMethodName() + ", wait
for retry in background. Ignored exception: "
                + e.getMessage() + ", ", e);
        //可以看这里把失败调用记录了下来，用于定时器重发
        addFailed(loadbalance, invocation, invokers, invoker);
        return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); // ignore
    }
}
```

```
private void addFailed(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>>
invokers, Invoker<T> lastInvoker) {
    if (failTimer == null) {
        synchronized (this) {
            if (failTimer == null) {
                failTimer = new HashedWheelTimer(
                        new NamedThreadFactory("failback-cluster-timer", true),
                        1,
                        TimeUnit.SECONDS, 32, failbackTasks);
            }
        }
    }
    RetryTimerTask retryTimerTask = new RetryTimerTask(loadbalance, invocation, invokers,
lastInvoker, retries, RETRY_FAILED_PERIOD);
    try {
        failTimer.newTimeout(retryTimerTask, RETRY_FAILED_PERIOD, TimeUnit.SECONDS);
    } catch (Throwable e) {
        logger.error("Failback background works error,invocation->" + invocation + ",
exception: " + e.getMessage());
    }
}
```

这里是采用时间轮的方式来定时重发的，时间轮在其他课程中讲过，这里就不再赘述了。

2.6、ForkingClusterInvoker

2.6.1、使用场景

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

2.6.2、源码分析

```
public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    try {
        checkInvokers(invokers, invocation);
        final List<Invoker<T>> selected;
        final int forks = getUrl().getParameter(FORKS_KEY, DEFAULT_FORKS);
        final int timeout = getUrl().getParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
        if (forks <= 0 || forks >= invokers.size()) {
            selected = invokers;
        } else {
```

```
            selected = new ArrayList<>(forks);
            while (selected.size() < forks) {
                Invoker<T> invoker = select(loadbalance, invocation, invokers, selected);
                if (!selected.contains(invoker)) {
                    //Avoid add the same invoker several times.
                    selected.add(invoker);
                }
            }
        }
        RpcContext.getServiceContext().setInvokers((List) selected);
        final AtomicInteger count = new AtomicInteger();
        final BlockingQueue<Object> ref = new LinkedBlockingQueue<>();
        //selected中是根据forks的配置从服务列表里面根据负载均衡算法选择出来的invoker对象
        for (final Invoker<T> invoker : selected) {
            //异步调用每一个invoker对象
            executor.execute(() -> {
                try {
                    Result result = invokeWithContext(invoker, invocation);
                    //把调用的返回结果存入队列
                    ref.offer(result);
                } catch (Throwable e) {
                    int value = count.incrementAndGet();
                    if (value >= selected.size()) {
                        ref.offer(e);
                    }
                }
            });
        }
        try {
            //等待获取返回结果，如果等待超时直接返回null,这里会拿到第一个先返回的结果直接返回
            Object ret = ref.poll(timeout, TimeUnit.MILLISECONDS);
            if (ret instanceof Throwable) {
                Throwable e = (Throwable) ret;
                throw new RpcException(e instanceof RpcException ? ((RpcException)
e).getCode() : 0, "Failed to forking invoke provider " + selected + ", but no luck to
perform the invocation. Last error is: " + e.getMessage(), e.getCause() != null ?
e.getCause() : e);
            }
            return (Result) ret;
        } catch (InterruptedException e) {
            throw new RpcException("Failed to forking invoke provider " + selected + ", but
no luck to perform the invocation. Last error is: " + e.getMessage(), e);
        }
    } finally {
        // clear attachments which is binding to current thread.
        RpcContext.getClientAttachment().clearAttachments();
    }
}
```

等待获取返回结果，如果等待超时直接返回null,这里会拿到第一个先返回的结果直接返回

2.7、BroadcastClusterInvoker

2.7.1、使用场景

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

2.7.2、源码分析

```
public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    checkInvokers(invokers, invocation);
    RpcContext.getServiceContext().setInvokers((List) invokers);
    RpcException exception = null;
    Result result = null;
    URL url = getUrl();
    // The value range of broadcast.fail.threshold must be 0~100.
    // 100 means that an exception will be thrown last, and 0 means that as long as an
exception occurs, it will be thrown.
    // see https://github.com/apache/dubbo/pull/7174
    int broadcastFailPercent = url.getParameter(BROADCAST_FAIL_PERCENT_KEY,
MAX_BROADCAST_FAIL_PERCENT);

    if (broadcastFailPercent < MIN_BROADCAST_FAIL_PERCENT || broadcastFailPercent >
MAX_BROADCAST_FAIL_PERCENT) {
        logger.info(String.format("The value corresponding to the broadcast.fail.percent
parameter must be between 0 and 100. " +
                "The current setting is %s, which is reset to 100.",
broadcastFailPercent));
        broadcastFailPercent = MAX_BROADCAST_FAIL_PERCENT;
    }

    int failThresholdIndex = invokers.size() * broadcastFailPercent /
MAX_BROADCAST_FAIL_PERCENT;
    int failIndex = 0;
    //从这for循环可以看出，就是循环所有的invokers对象，然后一一调用
    for (Invoker<T> invoker : invokers) {
        try {
            result = invokeWithContext(invoker, invocation);
            if (null != result && result.hasException()) {
                Throwable resultException = result.getException();
                if (null != resultException) {
                    exception = getRpcException(result.getException());
                    logger.warn(exception.getMessage(), exception);
                    if (failIndex == failThresholdIndex) {
                        break;
                    } else {
                        failIndex++;
                    }
                }
            }
        } catch (Throwable e) {
            exception = getRpcException(e);
            logger.warn(exception.getMessage(), exception);
            if (failIndex == failThresholdIndex) {
                break;
```

```
            } else {
                failIndex++;
            }
        }
    }

    if (exception != null) {
        if (failIndex == failThresholdIndex) {
            logger.debug(
                    String.format("The number of BroadcastCluster call failures has reached
the threshold %s", failThresholdIndex));
        } else {
            logger.debug(String.format("The number of BroadcastCluster call failures has
not reached the threshold %s, fail size is %s",
                    failIndex));
        }
        throw exception;
    }

    return result;
}
```

从这for循环可以看出，就是循环所有的invokers对象，然后一一调用

2.8、BroadcastClusterInvoker

2.8.1、使用场景

调用目前可用的实例（只调用一个），如果当前没有可用的实例，则抛出异常。通常用于不需要负载均衡的场景。

2.8.2、源码分析

```
@Override
public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    //循环所有的负载列表，isAvailable()是远程调用服务是否存活，如果存活就掉用它，这里没有负载均衡的逻辑
    for (Invoker<T> invoker : invokers) {
        if (invoker.isAvailable()) {
            return invokeWithContext(invoker, invocation);
        }
    }
    throw new RpcException("No provider available in " + invokers);
}
```

循环所有的负载列表，isAvailable()是远程调用服务是否存活，如果存活就掉用它，这里没有负载均衡的逻辑