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
2019-04-05
Dubbo

精尽 Dubbo 源码解析 —— 集群容错（二）之 Cluster 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（一）之抽象 API》 一文，分享 dubbo-cluster 模块， support 包，各种 Cluster 实现类。

Cluster 子类如下图：

我们可以看到，每个 Cluster 实现类，对应一个专属于其的 Invoker 实现类。下面，我们一个一个子类往下看。

老艿艿：本文对应 《Dubbo 用户指南 —— 集群容错》 文档。

2. AvailableCluster

com.alibaba.dubbo.rpc.cluster.support.AvailableCluster ，实现 Cluster 接口，调用首个可用服务器，目前用于多注册中心引用。代码如下：

```
public class AvailableCluster implements Cluster {

    public static final String NAME = "available";

    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new AvailableClusterInvoker<T>(directory);
    }

}
```

对应 Invoker 实现类为 AvailableClusterInvoker 。

2.1 AvailableClusterInvoker

com.alibaba.dubbo.rpc.cluster.support.AvailableClusterInvoker ，实现 AbstractClusterInvoker 抽象类，AvailableCluster Invoker 实现类。代码如下：

```
public class AvailableClusterInvoker<T> extends AbstractClusterInvoker<T> {

    public AvailableClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcExcep
        // 循环候选的 Invoker 集合，调用首个可用的 Invoker 对象。
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) { // 可用
                // 发起 RPC 调用
                return invoker.invoke(invocation);
            }
        }
        throw new RpcException("No provider available in " + invokers);
    }

}
```

3. BroadcastCluster

com.alibaba.dubbo.rpc.cluster.support.BroadcastCluster ，实现 Cluster 接口，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。代码如下

:

```
public class BroadcastCluster implements Cluster {

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new BroadcastClusterInvoker<T>(directory);
    }

}
```

对应 Invoker 实现类为 BroadcastClusterInvoker 。

3.1 BroadcastClusterInvoker

com.alibaba.dubbo.rpc.cluster.support.BroadcastClusterInvoker ，实现 AbstractClusterInvoker 抽象类，BroadcastCluster Invoker 实现类。代码如下：

```
public class BroadcastClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

    public BroadcastClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws Rp
        // 检查 invokers 即可用Invoker集合是否为空，如果为空，那么抛出异常
        checkInvokers(invokers, invocation);
        // 设置已经调用的 Invoker 集合，到 Context 中
        RpcContext.getContext().setInvokers((List) invokers);
        // 保存最后一次调用的异常
        RpcException exception = null;
        // 保存最后一次调用的结果
        Result result = null;
        // 循环候选的 Invoker 集合，调用所有 Invoker 对象。
        for (Invoker<T> invoker : invokers) {
            try {
                // 发起 RPC 调用
                result = invoker.invoke(invocation);
            } catch (RpcException e) {
                exception = e;
                logger.warn(e.getMessage(), e);
            } catch (Throwable e) {
                exception = new RpcException(e.getMessage(), e); // 封装成 RpcException 异常
                logger.warn(e.getMessage(), e);
            }
        }
        // 若存在一个异常，抛出该异常
        if (exception != null) {
            throw exception;
        }
        return result;
    }
```

```
}
```

4. FailbackCluster

com.alibaba.dubbo.rpc.cluster.support.FailbackCluster ，实现 Cluster 接口，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。代码如下：

```
public class FailbackCluster implements Cluster {

    public final static String NAME = "failback";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new FailbackClusterInvoker<T>(directory);
    }

}
```

对应 Invoker 实现类为 FailbackClusterInvoker 。

4.1 FailbackClusterInvoker

com.alibaba.dubbo.rpc.cluster.support.FailbackClusterInvoker ，实现 AbstractClusterInvoker 抽象类，FailbackCluster Invoker 实现类。

4.1.1 构造方法

```
/**
 * 重试频率
 */
private static final long RETRY_FAILED_PERIOD = 5 * 1000;

/**
 * ScheduledExecutorService 对象
 */
private final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(2, new NamedThread
/**
 * 失败任务集合
 */
private final ConcurrentMap<Invocation, AbstractClusterInvoker<?>> failed = new ConcurrentHashMap<Invocation, Abstrac
/**
 * 重试任务 Future
 */
private volatile ScheduledFuture<?> retryFuture;

public FailbackClusterInvoker(Directory<T> directory) {
    super(directory);
}
```

所有字段，都是和重试相关，胖友看下注释。

4.1.2 doInvoke

```
@Override
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcExcept
    try {
        // 检查 invokers 即可用Invoker集合是否为空，如果为空，那么抛出异常
        checkInvokers(invokers, invocation);
        // 根据负载均衡机制从 invokers 中选择一个Invoker
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        // RPC 调用得到 Result
        return invoker.invoke(invocation);
    } catch (Throwable e) {
        logger.error("Failback to invoke method " + invocation.getMethodName() + ", wait for retry in background. Ign
        // 添加到失败任务
        addFailed(invocation, this);
        return new RpcResult(); // ignore
    }
}
```

若 RPC 调用失败，则调用 `#addFailed(invocation, this)` 方法，添加到 `failed` 中，后台定时重试。

4.1.3 addFailed

```
private void addFailed(Invocation invocation, AbstractClusterInvoker<?> router) {
    // 若定时任务未初始化，进行创建
    if (retryFuture == null) {
        synchronized (this) {
            if (retryFuture == null) {
                retryFuture = scheduledExecutorService.scheduleWithFixedDelay(new Runnable() {

                    public void run() {
                        // collect retry statistics
                        try {
                            retryFailed();
                        } catch (Throwable t) { // Defensive fault tolerance
                            logger.error("Unexpected error occur at collect statistic", t);
                        }
                    }
                }, RETRY_FAILED_PERIOD, RETRY_FAILED_PERIOD, TimeUnit.MILLISECONDS);
            }
        }
    }
    // 添加到失败任务
    failed.put(invocation, router);
}
```

创建的定时任务，会调用 `#retryFailed()` 方法，重试任务，发起 RCP 调用。

4.1.4 retryFailed

```
void retryFailed() {
    if (failed.size() == 0) {
        return;
```

```
        }
        // 循环重试任务，逐个调用
        for (Map.Entry<Invocation, AbstractClusterInvoker<?>> entry : new HashMap<Invocation, AbstractClusterInvoker<?>>(
            Invocation invocation = entry.getKey();
            Invoker<?> invoker = entry.getValue();
            try {
                // RPC 调用得到 Result
                invoker.invoke(invocation);
                // 移除失败任务
                failed.remove(invocation);
            } catch (Throwable e) {
                logger.error("Failed retry to invoke method " + invocation.getMethodName() + ", waiting again.", e);
            }
        }
    }
```

循环重试任务，逐个发起 RPC 调用。若调用成功，移除该失败任务出 `failed` 集合。

在极端情况下，存在一个 BUG ，复现步骤如下：

1. 假设目前有两个服务提供者 A、B 。
2. 首先调用 A 服务，假设超时，添加到 `failed` 中。
3. 重试调用 B 服务（A 服务亦可），假设再次超时，添加到 `failed` 中。
4. 因为 `#doInvoker(...)` 方法，调用失败，不会抛出异常（当然也不能），导致 `#retryFailed(...)` 方法，误以为调用成功，错误的移除该失败任务出 `failed` 集合。

那么能不能在 `#retryFailed(...)` 方法中，先移除该失败任务出 `failed` 集合呢，再发起 PRC 调用呢？答案是不可以，因为在调用 `#doInvoke(...)` 方法之前，可能发生异常，导致失败任务的丢失。

那么该怎么办？有两种方式：

1. 上述方案的基础上，在 `#retryFailed(...)` 方法的移除处理中，增加调用 `#addFailed(...)` 方法。
2. 枚举一个 `FAILED_RESULT` 对象，让 `#doInvoke(...)` 方法发生异常时，返回该对象。这样 `#retryFailed(...)` 方法，在移除出 `failed` 集合时，增加下是否执行成功的判断。

笔者倾向第二种，逻辑更加线性和易懂。

5. FailfastCluster

`com.alibaba.dubbo.rpc.cluster.support.FailfastCluster` ，实现 Cluster 接口，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。代码如下：

```
public class FailfastCluster implements Cluster {

    public final static String NAME = "failfast";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new FailfastClusterInvoker<T>(directory);
    }

}
```

对应 Invoker 实现类为 FailfastClusterInvoker 。

5.1 FailfastInvoker

com.alibaba.dubbo.rpc.cluster.support.FailbackClusterInvoker ，实现 AbstractClusterInvoker 抽象类，Failfast Invoker 实现类。代码如下：

```
public class FailfastClusterInvoker<T> extends AbstractClusterInvoker<T> {

    public FailfastClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcExcep
        // 检查 invokers 即可用Invoker集合是否为空，如果为空，那么抛出异常
        checkInvokers(invokers, invocation);
        // 根据负载均衡机制从 invokers 中选择一个Invoker
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        try {
            // RPC 调用得到 Result
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            // 若是业务性质的异常，直接抛出
            if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz exception.
                throw (RpcException) e;
            }
            // 封装 RpcException 异常，并抛出
            throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0,
                    "Failfast invoke providers " + invoker.getUrl() + " " + loadbalance.getClass().getSimpleName() +
        }
    }

}
```

和 FailbackClusterInvoker 差异点，在于对异常的处理。

6. FailsafeCluster

com.alibaba.dubbo.rpc.cluster.support.FailsafeCluster ，实现 Cluster 接口，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。

代码如下：

```
public class FailfastCluster implements Cluster {

    public final static String NAME = "failfast";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new FailfastClusterInvoker<T>(directory);
    }

}
```

对应 Invoker 实现类为 FailsafeClusterInvoker 。

6.1 FailsafeClusterInvoker

com.alibaba.dubbo.rpc.cluster.support.FailsafeClusterInvoker ，实现 AbstractClusterInvoker 抽象类，Failsafe Invoker 实现类。代码如下：

```
public class FailsafeClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

    public FailsafeClusterInvoker(Directory<T> directory) {
        super(directory);
    }

    @Override
    public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcExcep
        try {
            // 检查 invokers 即可用Invoker集合是否为空，如果为空，那么抛出异常
            checkInvokers(invokers, invocation);
            // 根据负载均衡机制从 invokers 中选择一个Invoker
            Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
            // RPC 调用得到 Result
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            // 打印异常日志
            logger.error("Failsafe ignore exception: " + e.getMessage(), e);
            // 忽略异常
            return new RpcResult(); // ignore
        }
    }

}
```

和 FailfastInvoker 差异点，在于对异常的处理。

7. ForkingCluster

com.alibaba.dubbo.rpc.cluster.support.ForkingCluster ，实现 Cluster 接口，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。

代码如下：

```
public class ForkingCluster implements Cluster {

    public final static String NAME = "forking";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new ForkingClusterInvoker<T>(directory);
    }

}
```

7.1 ForkingClusterInvoker

老艿艿：BlockQueue 的使用，非常精髓！

com.alibaba.dubbo.rpc.cluster.support.ForkingClusterInvoker ，实现 AbstractClusterInvoker 抽象类，ForkingCluster Invoker 实现类。代码如下：

```
1: public class ForkingClusterInvoker<T> extends AbstractClusterInvoker<T> {
2:
3:     /**
4:      * ExecutorService 对象，并且为 CachedThreadPool 。
5:      */
6:     private final ExecutorService executor = Executors.newCachedThreadPool(new NamedThreadFactory("forking-cluste
7:
8:     public ForkingClusterInvoker(Directory<T> directory) {
9:         super(directory);
10:    }
11:
12:    @Override
13:    @SuppressWarnings({"unchecked", "rawtypes"})
14:    public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throw
15:        // 检查 invokers 即可用Invoker集合是否为空，如果为空，那么抛出异常
16:        checkInvokers(invokers, invocation);
17:        // 保存选择的 Invoker 集合
18:        final List<Invoker<T>> selected;
19:        // 得到最大并行数，默认为 Constants.DEFAULT_FORKS = 2
20:        final int forks = getUrl().getParameter(Constants.FORKS_KEY, Constants.DEFAULT_FORKS);
21:        // 获得调用超时时间，默认为 DEFAULT_TIMEOUT = 1000 毫秒
22:        final int timeout = getUrl().getParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT);
23:        // 若最大并行书小于等于 0，或者大于 invokers 的数量，直接使用 invokers
24:        if (forks <= 0 || forks >= invokers.size()) {
25:            selected = invokers;
26:        } else {
27:            // 循环，根据负载均衡机制从 invokers，中选择一个个Invoker ，从而组成 Invoker 集合。
28:            // 注意，因为增加了排重逻辑，所以不能保证获得的 Invoker 集合的大小，小于最大并行数
29:            selected = new ArrayList<Invoker<T>>();
30:            for (int i = 0; i < forks; i++) {
31:                // 在invoker列表(排除selected)后,如果没有选够,则存在重复循环问题.见select实现.
32:                Invoker<T> invoker = select(loadbalance, invocation, invokers, selected);
33:                if (!selected.contains(invoker)) { //Avoid add the same invoker several times. //防止重复添加invo
34:                    selected.add(invoker);
35:                }
36:            }
37:        }
38:        // 设置已经调用的 Invoker 集合，到 Context 中
39:        RpcContext.getContext().setInvokers((List) selected);
40:        // 异常计数器
41:        final AtomicInteger count = new AtomicInteger();
42:        // 创建阻塞队列
43:        final BlockingQueue<Object> ref = new LinkedBlockingQueue<Object>();
44:        // 循环 selected 集合，提交线程池，发起 RPC 调用
45:        for (final Invoker<T> invoker : selected) {
46:            executor.execute(new Runnable() {
47:                public void run() {
48:                    try {
49:                        // RPC 调用，获得 Result 结果
50:                        Result result = invoker.invoke(invocation);
51:                        // 添加 Result 到 `ref` 阻塞队列
```

```
52:                         ref.offer(result);
53:                     } catch (Throwable e) {
54:                         // 异常计数器 + 1
55:                         int value = count.incrementAndGet();
56:                         // 若 RPC 调用结果都是异常，则添加异常到 `ref` 阻塞队列
57:                         if (value >= selected.size()) {
58:                             ref.offer(e);
59:                         }
60:                     }
61:                 }
62:             });
63:         }
64:         try {
65:             // 从 `ref` 队列中，阻塞等待结果
66:             Object ret = ref.poll(timeout, TimeUnit.MILLISECONDS);
67:             // 若是异常结果，抛出 RpcException 异常
68:             if (ret instanceof Throwable) {
69:                 Throwable e = (Throwable) ret;
70:                 throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0, "Failed to f
71:             }
72:             // 若是正常结果，直接返回
73:             return (Result) ret;
74:         } catch (InterruptedException e) {
75:             throw new RpcException("Failed to forking invoke provider " + selected + ", but no luck to perform th
76:         }
77:     }
78:
79: }
```

第 15 至 39 行：胖友自己看代码注释，比较易懂。
第 41 行：`count` 变量，异常计数器。
第 43 行：`ref` 变量，阻塞队列。通过它，实现线程池异步执行任务的结果通知，非常亮眼。
第 44 至 63 行：循环 `selected` 集合，提交线程池，发起 RPC 调用。
- 第 49 至 52 行：调用 `Invoker#invoke(invocation)` 方法，RPC 调用，成功获得 Result 结果，并将 Result 添加到 `ref` 阻塞队列中。
- 第 53 至 59 行：若调用失败，异常计数器 `count` 加一。当所有的 RPC 调用都完成，并且都是异常时，则添加最后一个异常到 `ref` 阻塞队列。 细节处理很到位。

第 66 行：从 `ref` 队列中，阻塞等待，直到获得到结果或者超时。至此，ForkingClusterInvoker 实现了并行调用，且只要一个成功即返回。当然，还有一个隐性的，所有都失败才返回。
第 67 至 76 行：处理等待的“结果”。

8. FailoverCluster

FailoverCluster ，在 《精尽 Dubbo 源码解析 —— 集群容错（一）之抽象 API》 一文中，我们已经详细解析。

9. MergeableCluster

MergeableCluster ，对应 《Dubbo 用户指南 —— 分组聚合》 文档，我们后续单独写文章分享。

10. MockClusterWrapper

MockClusterWrapper ，对应 《Dubbo 用户指南 ——本地伪装 》 文档，我们后续单独写文章分享。

666. 彩蛋

比想象中的简单一些，花的比较多的时间在 FailbackClusterInvoker ，考虑是否存在 BUG 。

比较亮眼的是 ForkingClusterInvoker ，萌萌的 BlockQueue 。

文章目录

2014 - 2023 芋道源码 |
总访客数 次 && 总访问量 次
回到首页