crazyhzm 265

Dubbo源码解析（三十五）集群——cluster

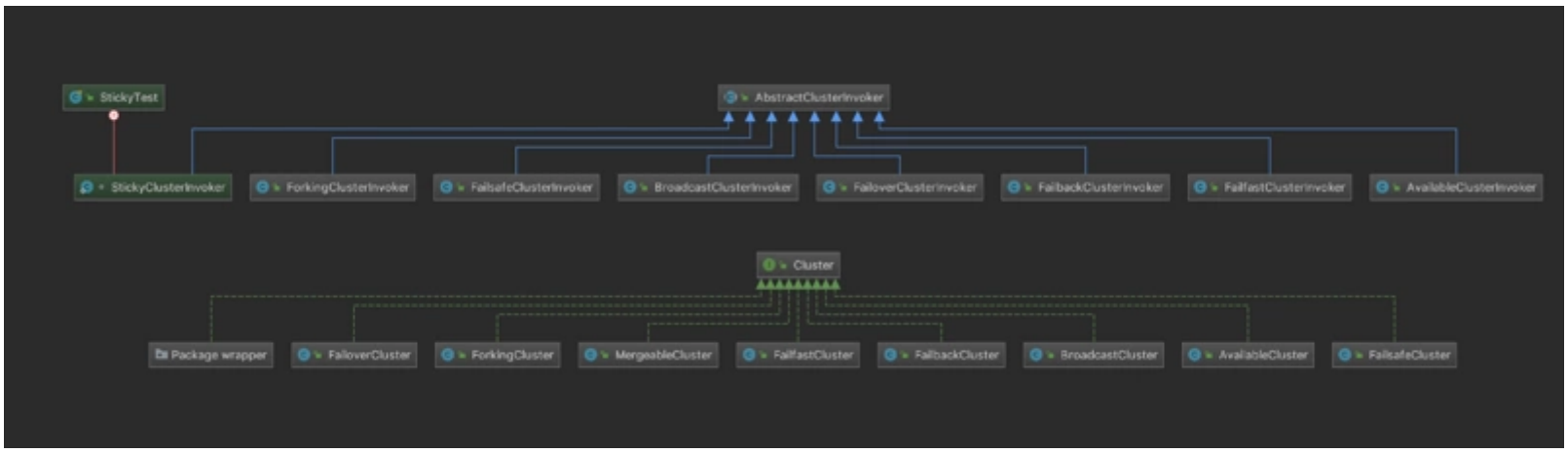
dubbo java 阅读约 54 分钟

集群——cluster

> 目标：介绍dubbo中集群容错的几种模式，介绍dubbo-cluster下support包的源码。

前言

集群容错还是很好理解的，就是当你调用失败的时候所作出的措施。先来看看有哪些模式：



图有点小，见谅，不过可以眯着眼睛看稍微能看出来一点，每一个Cluster实现类都对应着一个invoker，因为这个模式启用的时间点就是在调用的时候，而我在之前的文章里面讲过，invoker贯穿来整个服务的调用。不过这里除了调用失败的一些模式外，还有几个特别的模式，他们应该说成是失败的措施，而已调用的方式。

1. Failsafe Cluster：失败安全，出现异常时，直接忽略。失败安全就是当调用过程中出现异常时，FailsafeClusterInvoker 仅会打印异常，而不会抛出异常。适用于写入审计日志等操作
2. Failover Cluster：失败自动切换，当调用出现失败的时候，会自动切换集群中其他服务器，来获得invoker重试，通常用于读操作，但重试会带来更长延迟。一般都会设置重试次数。
3. Failfast Cluster：只会进行一次调用，失败后立即抛出异常。适用于幂等操作，比如新增记录。
4. Failback Cluster：失败自动恢复，在调用失败后，返回一个空结果给服务提供者。并通过定时任务对失败的调用记录并且重传，适合执行消息通知等操作。
5. Forking Cluster：会在线程池中运行多个线程，来调用多个服务器，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。一般会设置最大并行数。
6. Available Cluster：调用第一个可用的服务器，仅仅应用于多注册中心。
7. Broadcast Cluster：广播调用所有提供者，逐个调用，在循环调用结束后，只要任意一台报错就报错。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
8. Mergeable Cluster：该部分在分组聚合讲述。
9. MockClusterWrapper：该部分在本地伪装讲述。

源码分析

（一）AbstractClusterInvoker

该类实现了Invoker接口，是集群Invoker的抽象类。

1.属性

```
private static final Logger logger = LoggerFactory
        .getLogger(AbstractClusterInvoker.class);
/**
 * 目录，包含多个invoker
 */
protected final Directory<T> directory;

/**
 * 是否需要核对可用
 */
protected final boolean availablecheck;

/**
 * 是否销毁
 */
private AtomicBoolean destroyed = new AtomicBoolean(false);

/**
 * 粘滞连接的Invoker
 */
private volatile Invoker<T> stickyInvoker = null;
```

2.select

```
protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers,
List<Invoker<T>> selected) throws RpcException {
    // 如果invokers为空，则返回null
    if (invokers == null || invokers.isEmpty())
        return null;
    // 获得方法名
    String methodName = invocation == null ? "" : invocation.getMethodName();

    // 是否启动了粘滞连接
    boolean sticky = invokers.get(0).getUrl().getMethodParameter(methodName, Constants.CLUSTER_STICKY_KEY,
Constants.DEFAULT_CLUSTER_STICKY);
    {
        //ignore overloaded method
        // 如果上一次粘滞连接的调用不在可选的提供者列合内，则直接设置为空
        if (stickyInvoker != null && !invokers.contains(stickyInvoker)) {
            stickyInvoker = null;
        }
        //ignore concurrency problem
        // stickyInvoker不为null,并且没在已选列表中，返回上次的服务提供者stickyInvoker，但之前强制校验可达性。
        // 由于stickyInvoker不能包含在selected列表中，通过代码看，可以得知forking和failover集群策略，用不了sticky属性
        if (sticky && stickyInvoker != null && (selected == null || !selected.contains(stickyInvoker))) {
            if (availablecheck && stickyInvoker.isAvailable()) {
                return stickyInvoker;
            }
        }
    }
```

该方法实现了使用负载均衡策略选择一个调用者。首先，使用loadbalance选择一个调用者。如果此调用者位于先前选择的列表中，或者如果此调用者不可用，则重新选择，否则返回第一个选定的调用者。重新选择，重选的验证规则：选择>可用。这条规则可以保证所选的调用者最少有机会成为之前选择的列表中的一个，也是保证这个调用程序可用。

3.doSelect

```
private Invoker<T> doSelect(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers,
List<Invoker<T>> selected) throws RpcException {
    if (invokers == null || invokers.isEmpty())
        return null;
    // 如果只有一个 ，就直接返回这个
    if (invokers.size() == 1)
        return invokers.get(0);
    // 如果没有指定用哪个负载均衡策略，则默认用随机负载均衡策略
    if (loadbalance == null) {
        loadbalance =
ExtensionLoader.getExtensionLoader(LoadBalance.class).getExtension(Constants.DEFAULT_LOADBALANCE);
    }
    // 调用负载均衡选择
    Invoker<T> invoker = loadbalance.select(invokers, getUrl(), invocation);

    //If the `invoker` is in the  `selected` or invoker is unavailable && availablecheck is true, reselect.
    // 如果选择的提供者，已在selected中或者不可用则重新选择
    if ((selected != null && selected.contains(invoker))
            || (!invoker.isAvailable() && getUrl() != null && availablecheck)) {
        try {
            // 重新选择
            Invoker<T> rinvoker = reselect(loadbalance, invocation, invokers, selected, availablecheck);
            if (rinvoker != null) {
                invoker = rinvoker;
            } else {
                //Check the index of current selected invoker, if it's not the last one, choose the one at
```

该方法是用负载均衡选择一个invoker的主要逻辑。

4.reselect

```
private Invoker<T> reselect(LoadBalance loadbalance, Invocation invocation,
                            List<Invoker<T>> invokers, List<Invoker<T>> selected, boolean availablecheck)
        throws RpcException {

    //Allocating one in advance, this list is certain to be used.
    //预先分配一个重选列表，这个列表是一定会用到的.
    List<Invoker<T>> reselectInvokers = new ArrayList<Invoker<T>>(invokers.size() > 1 ? (invokers.size() - 1) :
invokers.size());

    //First, try picking a invoker not in `selected`.
    //先从非select中选
    //把不包含在selected中的提供者，放入重选列表reselectInvokers，让负载均衡器选择
    if (availablecheck) { // invoker.isAvailable() should be checked
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                if (selected == null || !selected.contains(invoker)) {
                    reselectInvokers.add(invoker);
                }
            }
        }
        // 在重选列表中用负载均衡器选择
        if (!reselectInvokers.isEmpty()) {
            return loadbalance.select(reselectInvokers, getUrl(), invocation);
        }
    } else { // do not check invoker.isAvailable()
        // 不核对服务是否可以，把不包含在selected中的提供者，放入重选列表reselectInvokers，让负载均衡器选择
```

该方法是是重新选择的逻辑实现。

5.invoke

```
@Override
public Result invoke(final Invocation invocation) throws RpcException {
    // 核对是否已经销毁
    checkWhetherDestroyed();
    LoadBalance loadbalance = null;

    // binding attachments into invocation.
    // 获得上下文的附加值
    Map<String, String> contextAttachments = RpcContext.getContext().getAttachments();
    // 把附加值放入到会话域中
    if (contextAttachments != null && contextAttachments.size() != 0) {
        ((RpcInvocation) invocation).addAttachments(contextAttachments);
    }

    // 生成服务提供者集合
    List<Invoker<T>> invokers = list(invocation);
    if (invokers != null && !invokers.isEmpty()) {
        // 获得负载均衡器
        loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).getExtension(invokers.get(0).getUrl()
                .getMethodParameter(RpcUtils.getMethodName(invocation), Constants.LOADBALANCE_KEY,
Constants.DEFAULT_LOADBALANCE));
    }
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    return doInvoke(invocation, invokers, loadbalance);
}
```

该方法是invoker接口必备的方法，调用链的逻辑，不过主要的逻辑在doInvoke方法中，该方法是该类的抽象方法，让子类只关注doInvoke方法。

6.list

```
protected List<Invoker<T>> list(Invocation invocation) throws RpcException {
    // 把会话域中的invoker加入集合
    List<Invoker<T>> invokers = directory.list(invocation);
    return invokers;
}
```

该方法是调用了directory的list方法，从会话域中获得所有的Invoker集合。关于directory我会在后续文章讲解。

（二）AvailableCluster

```
public class AvailableCluster implements Cluster {

    public static final String NAME = "available";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {

        // 创建一个AbstractClusterInvoker
        return new AbstractClusterInvoker<T>(directory) {
            @Override
            public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws
RpcException {
                // 遍历所有的involer，只要有一个可用就直接调用。
                for (Invoker<T> invoker : invokers) {
                    if (invoker.isAvailable()) {
                        return invoker.invoke(invocation);
                    }
                }
                throw new RpcException("No provider available in " + invokers);
            }
        };

    }

}
```

Available Cluster我在上面已经讲过了，只要找到一个可用的，则直接调用。

（三）BroadcastCluster

```
public class BroadcastCluster implements Cluster {

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        // 创建一个BroadcastClusterInvoker
        return new BroadcastClusterInvoker<T>(directory);
    }

}
```

关键实现在于BroadcastClusterInvoker。

（四）BroadcastClusterInvoker

```
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 RpcException {
        // 检测invokers是否为空
        checkInvokers(invokers, invocation);
        // 把invokers放到上下文
        RpcContext.getContext().setInvokers((List) invokers);
        RpcException exception = null;
        Result result = null;
        // 遍历invokers，逐个调用，在循环调用结束后，只要任意一台报错就报错
        for (Invoker<T> invoker : invokers) {
            try {
                result = invoker.invoke(invocation);
            } catch (RpcException e) {
                exception = e;
                logger.warn(e.getMessage(), e);
            } catch (Throwable e) {
```

（五）ForkingCluster

```
public class ForkingCluster implements Cluster {

    public final static String NAME = "forking";

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

}
```

（六）ForkingClusterInvoker

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

    /**
     * 线程池
     * Use {@link NamedInternalThreadFactory} to produce {@link
com.alibaba.dubbo.common.threadlocal.InternalThread}
     * which with the use of {@link com.alibaba.dubbo.common.threadlocal.InternalThreadLocal} in {@link
RpcContext}.
     */
    private final ExecutorService executor = Executors.newCachedThreadPool(
            new NamedInternalThreadFactory("forking-cluster-timer", true));

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

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance)
throws RpcException {
        try {
            // 检测invokers是否为空
            checkInvokers(invokers, invocation);
            final List<Invoker<T>> selected;
            // 获取 forks 配置
            final int forks = getUrl().getParameter(Constants.FORKS_KEY, Constants.DEFAULT_FORKS);
```

(七) FailbackCluster

```
public class FailbackCluster implements Cluster {

    public final static String NAME = "failback";

    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        // 创建一个FailbackClusterInvoker
        return new FailbackClusterInvoker<T>(directory);
    }

}
```

(八) FailbackClusterInvoker

1.属性

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

// 重试间隔
private static final long RETRY_FAILED_PERIOD = 5 * 1000;

/**
 * 定时器
 * Use {@link NamedInternalThreadFactory} to produce {@link com.alibaba.dubbo.common.threadlocal.InternalThread}
 * which with the use of {@link com.alibaba.dubbo.common.threadlocal.InternalThreadLocal} in {@link RpcContext}.
 */
private final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(2,
        new NamedInternalThreadFactory("failback-cluster-timer", true));

/**
 * 失败集合
 */
private final ConcurrentMap<Invocation, AbstractClusterInvoker<?>> failed = new ConcurrentHashMap<Invocation,
AbstractClusterInvoker<?>>();
/**
 * future
 */
private volatile ScheduledFuture<?> retryFuture;
```

2.doInvoke

```
@Override
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws
RpcException {
    try {
        // 检测invokers是否为空
        checkInvokers(invokers, invocation);
        // 选择出invoker
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        // 调用
        return invoker.invoke(invocation);
    } catch (Throwable e) {
        logger.error("Failback to invoke method " + invocation.getMethodName() + ", wait for retry in background.
Ignored exception: "
                + e.getMessage() + ", ", e);
        // 如果失败，则加入到失败队列，等待重试
        addFailed(invocation, this);
        return new RpcResult(); // ignore
    }
}
```

该方法是选择invoker调用的逻辑，在抛出异常的时候，做了失败重试的机制，主要实现在addFailed。

3.addFailed

```
private void addFailed(Invocation invocation, AbstractClusterInvoker<?> router) {
    if (retryFuture == null) {
        // 锁住
        synchronized (this) {
            if (retryFuture == null) {
                // 创建定时任务，每隔5秒执行一次
                retryFuture = scheduledExecutorService.scheduleWithFixedDelay(new Runnable() {

                    @Override
                    public void run() {
                        // collect retry statistics
                        try {
                            // 对失败的调用进行重试
                            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 invocation 和 invoker 到 failed 中
    failed.put(invocation, router);
}
```

该方法做的事创建了定时器，然后把失败的调用放入到集合中。

4.retryFailed

```
void retryFailed() {
    // 如果失败队列为0，返回
    if (failed.size() == 0) {
        return;
    }
    // 遍历失败队列
    for (Map.Entry<Invocation, AbstractClusterInvoker<?>> entry : new HashMap<Invocation, AbstractClusterInvoker<?>>(
            failed).entrySet()) {
        // 获得会话域
        Invocation invocation = entry.getKey();
        // 获得invoker
        Invoker<?> invoker = entry.getValue();
        try {
            // 重新调用
            invoker.invoke(invocation);
            // 从失败队列中移除
            failed.remove(invocation);
        } catch (Throwable e) {
            logger.error("Failed retry to invoke method " + invocation.getMethodName() + ", waiting again.", e);
        }
    }
}
```

这个方法是调用失败的invoker重新调用的机制。

（九）FailfastCluster

```
public class FailfastCluster implements Cluster {

    public final static String NAME = "failfast";

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

}
```

（十）FailfastClusterInvoker

```
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
RpcException {
        // 检测invokers是否为空
        checkInvokers(invokers, invocation);
        // 选择一个invoker
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        try {
            // 调用
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz exception.
                // 抛出异常
                throw (RpcException) e;
            }
            // 抛出异常
            throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0, "Failfast invoke
providers " + invoker.getUrl() + " " + loadbalance.getClass().getSimpleName() + " select from all providers " +
invokers + " for service " + getInterface().getName() + " method " + invocation.getMethodName() + " on consumer "
+ NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion() + ", but no luck to perform the
```

逻辑比较简单，调用抛出异常就直接抛出。

(十一) FailoverCluster

```
public class FailoverCluster implements Cluster {

    public final static String NAME = "failover";

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

}
```

(十二) FailoverClusterInvoker

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

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

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

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance loadbalance)
throws RpcException {
        // 复制一个invoker集合
        List<Invoker<T>> copyinvokers = invokers;
        // 检测是否为空
        checkInvokers(copyinvokers, invocation);
        // 获取重试次数
        int len = getUrl().getMethodParameter(invocation.getMethodName(), Constants.RETRIES_KEY,
Constants.DEFAULT_RETRIES) + 1;
        if (len <= 0) {
            len = 1;
        }
        // retry loop.
        // 记录最后一个异常
        RpcException le = null; // last exception.
        List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyinvokers.size()); // invoked invokers.
```

该类实现了失败重试的容错策略，当调用失败的时候，记录下异常，然后循环调用下一个选择出来的invoker，直到重试次数用完，抛出最后一次的异常。

(十三) FailsafeCluster

```
public class FailsafeCluster implements Cluster {

    public final static String NAME = "failsafe";

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

}
```

(十四) FailsafeClusterInvoker

```
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
RpcException {
        try {
            // 检测invokers是否为空
            checkInvokers(invokers, invocation);
            // 选择一个invoker
            Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
            // 调用
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            // 如果失败打印异常，返回一个空结果
            logger.error("Failsafe ignore exception: " + e.getMessage(), e);
            return new RpcResult(); // ignore
        }
    }
}
```

逻辑比较简单，就是不抛出异常，只是打印异常。

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cluster实现的部分，讲了几种调用方式和容错策略。接下来我将开始对集群模块关于配置规则部分进行讲解。

crazyhzm

265