Dubbo集群 之 容错

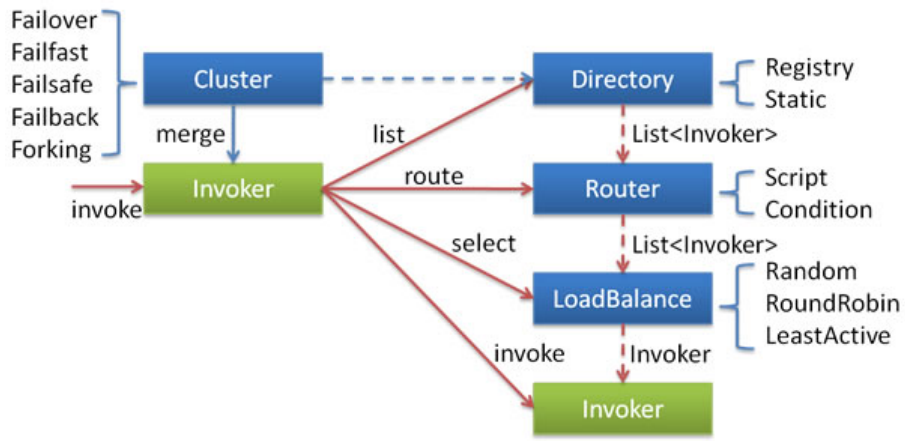
> *在分布式系统中，集群某个某些节点出现问题是大概率事件，因此在设计分布式RPC框架的过程中，必须要把失败作为设计的一等公民来对待。*

由官网中的这一段话，可以看出容错在微服务框架中的作用举足轻重，它是保证服务高可用性的关键举措。容错策略有好几种，但却没有适合所有场景的银弹，因此Dubbo提供能如下几种供用户据需要选用， `Failover` 是默认方案：

1. `Failover` (失败自动切换)
2. `Failsafe` (失败安全)
3. `Failfast` (快速失败)
4. `Failback` (失败自动恢复)
5. `Forking` (并行调用)
6. `Broadcast` (广播调用)

容错架构

Dubbo中的容错是和集群中的其它组件一起发生作用的，其关系如下图所示：



图中各组件的作用及关系：

- `Invoker` 表示的是一个微服务的引用实例，它封装了 `Provider` 地址及 `Service` 接口信息；
- `Directory` 目录服务，根据服务名得到的同一个微服务的多个引用实例集合， 该集合会动态变化的，比如注册中心推送变更；
- `Cluster` 将 `Directory` 中的多个 `Invoker` 伪装成一个 Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个；

- `Router` 负责从多个 `Invoker` 中按路由规则选出子集，比如读写分离，应用隔离等；
- `LoadBalance` 负责从多个 `Invoker` 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选；

> **NOTE**
>
> `ClusterInvoker` 可以拥有自己的配置，继承自 `Directory` 的配置总线URL。
>
> Dubbo中，在应用内部提供配置同步入站和获取功能被视作目录服务，也即 `Directory` ；而向所有微服务、应用提供配置管理、获取、同步出站、入站功能，独立于微服务存在的服务被视作注册中心，也即 `Registry` （由于配置中心往往和注册中心融为一体，对外被同一当做注册中心）。

容错实现

各个不同的策略继承自 `AbstractClusterInvoker` ，以共用一套基础代码，它负责将 `Directory` 、`Router` 、 `LoadBalance` 联结起来，构成一个有机整体，以最终提供一个负责实际RPC调用的 `Invoker` 对象。

下文中均以 `ClusterInvoker` 代替所有 `AbstractClusterInvoker` 实现，也即有集群特性的 `Invoker` 。

服务级别的 `ClusterInvoker` 单例

像其它 `Invoker` 一样， `ClusterInvoker` 是一个行为实体，专门负责某个特定引用微服务的容错处理，其过程都封装在内部。一个被引用的微服务只会使用一种容错策略，在应用实例启动的那时就已经确定了——由配置文件等事先确定。这也就说明应用需要在合适的时机为每一种被引用微服务选用一种策略并创建它的一个 `ClusterInvoker` 实例。 `ClusterInvoker` 支持泛型，泛型参数是微服务的接口类型，也就是说每一种 `ClusterInvoker` 实现虽然本身不是单例的，但支持服务级别的全局单例，只有明白这个特性才能更清楚的理解代码中出现的粘滞是咋回事。

《Dubbo之SPI扩展点加载》曾剖析，Dubbo中的扩展点具类都是单例的，因而可以认为Dubbo的SPI机制是另一种形式的单例的创建模式，但不能直接作用在 `ClusterInvoker` 上，因而专门提炼出了如下的 `Cluster` 扩展点，经由其 `join()` 为每一种引用微服务创建一个 `ClusterInvoker` 实例：

```java
@SPI(FailoverCluster.NAME)
public interface Cluster {

    //Merge the directory invokers to a virtual invoker.
    @Adaptive
    <T> Invoker<T> join(Directory<T> directory) throws RpcException;

}
```

注：代码中的SPI相关声明可以看出，Dubbo默认选用的是 Failover 这种容错模式。

另外针对每一种容错策略的实现，均会有类似如下 ClusterInvoker 的创建类，代码几乎一样：

```java
public class FailoverCluster implements Cluster {

    public final static String NAME = "failover";

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

}
```

使用这种创建型模式的好处，是将具体的实现和用户接口层面解耦，将结构的复杂性隐藏在内部。当然上述关于 服务级别的全局单例 的表述是有问题的，类似容错这种领域通用机制在设计之初就会当做一个扩展点来实现，但以 ClusterInvoker 为中心，换了一个不同的视觉来看待 Cluster ，对Dubbo的SPI机制会有一个更加全面的理解。

以上述源码为例，由于 FailoverCluster 是单例的，而创建 FailoverClusterInvoker 对象的 join 方法基本只在一个微服务被引用的时候才会被调用一回，因而 ClusterInvoker 总体而言于某个特定微服务是全局单例的，也就是说一个微服务会对应的一个 ClusterInvoker 实例。

父类 AbstractClusterInvoker

> *Merge the directory invokers to a virtual invoker.*

这个表述来自接口 Cluster ，Dubbo将 ClusterInvoker 视作一个虚拟的微服务引用实例，原因经过框架层层封装处理后，原本一个微服务同时有多个实例在后台默默地为应用提供服务，但上层开发人员的视线里只会看到一个 invoker 引用实例。

另一层意思很明显，所有这些背后在默默提供服务的实例来自 directory 目录服务这个组件，它负责了感知来自配置中心的变化，确保新增或者掉线的实例最终会体现在 ClusterInvoker 这个虚拟的 invoker 引用实例上。

生命周期管理

`ClusterInvoker` 是重依赖于 `Directory` 的，他们都实现了 `Node` 接口，一些表征微服务的关键信息也来源于后者，甚至有关生命周期的管理行为和后者密切相关，如下：

JAVA

```java
public abstract class AbstractClusterInvoker<T> implements Invoker<T> {


    protected final Directory<T> directory;

    protected final boolean availablecheck;

    private AtomicBoolean destroyed = new AtomicBoolean(false);

    private volatile Invoker<T> stickyInvoker = null;

    public AbstractClusterInvoker(Directory<T> directory) {
        this(directory, directory.getUrl());
    }

    public AbstractClusterInvoker(Directory<T> directory, URL url) {
        if (directory == null) {
            throw new IllegalArgumentException("service directory == null");
        }

        this.directory = directory;
        //sticky: invoker.isAvailable() should always be checked before using when
availablecheck is true.
        this.availablecheck = url.getParameter(CLUSTER_AVAILABLE_CHECK_KEY,
DEFAULT_CLUSTER_AVAILABLE_CHECK);
    }

    @Override
    public Class<T> getInterface() {
        return directory.getInterface();
    }

//================================
// Node接口实现
//================================

    @Override
    public URL getUrl() {
        return directory.getUrl();
    }

    @Override
    public boolean isAvailable() {
        Invoker<T> invoker = stickyInvoker;
        if (invoker != null) {
            return invoker.isAvailable();
        }
        return directory.isAvailable();
    }

    @Override
    public void destroy() {
        if (destroyed.compareAndSet(false, true)) {
            directory.destroy();
        }
    }
```

```
    protected void checkWhetherDestroyed() {
        if (destroyed.get()) {
            throw new RpcException("Rpc cluster invoker for " + getInterface() + " on
consumer " + NetUtils.getLocalHost()
                    + " use dubbo version " + Version.getVersion()
                    + " is now destroyed! Can not invoke any more.");
        }
    }
    ...
}
```

和其它类型的 `Invoker` 一样， `ClusterInvoker` 也是工作在并发场景中，对象的销毁只能发生一次，因此加入了 `AtomicBoolean` 类型的 `destroyed` 属性。一旦被 destroyed 的实例便不能再处理服务，需要抛错处理，如 `checkWhetherDestroyed()` 方法所示。

另外还有两个比较关键的属性，此处有必要提前介绍下，它们会穿梭于本章节的其他后面源码中：

1. `availablecheck`： 如果没有配置如下参数，发送RPC请求给一个服务实例时将不管是它否处于可用状态；
    - `"cluster.availablecheck"`： true | false；默认值 DEFAULT_CLUSTER_AVAILABLE_CHECK=true；
2. `stickyInvoker`： 基于 `ClusterInvoker` 的单例模式，实现微服务的粘滞特性；

主体流程

`ClusterInvoker` 的主体逻辑都围绕着其实现接口 `Invoker` 定义的 `invoke()` 方法展开，父类将基础的公共逻辑封装起来，再定义一个抽象方法，由子类去覆写实现，这在Dubbo是很常用的一种手法。

具体实现源码中，逻辑很清晰，分为如下几步：

1. 首先检验当前虚拟服务实例是否可用；
2. 然后将当前线程中的附属参数设给RPC方法的入参 `invocation` ，另外如果结合配置总线中传入的参数等判断当前RPC方式是否被异步调用，如果是则分配一个全局唯一的ID编号，具体参考《Dubbo RPC 之 Protocol协议层（一）》中的相关章节；
3. 紧接着使用目录服务 `directory` 导出所有可用的服务实例；
4. 最后根据总线配置获取负载均衡策略默认加权随机，传入给当前类中定义的 `doInvoke()` 方法完成RPC方法的调用。

```java
public abstract class AbstractClusterInvoker<T> implements Invoker<T> {

    protected abstract Result doInvoke(Invocation invocation, List<Invoker<T>>
invokers,
                                       LoadBalance loadbalance) throws RpcException;
    @Override
    public Result invoke(final Invocation invocation) throws RpcException {
        checkWhetherDestroyed();

        // binding attachments into invocation.
        Map<String, String> contextAttachments =
RpcContext.getContext().getAttachments();
        if (contextAttachments != null && contextAttachments.size() != 0) {
            ((RpcInvocation) invocation).addAttachments(contextAttachments);
        }

        List<Invoker<T>> invokers = list(invocation);
        LoadBalance loadbalance = initLoadBalance(invokers, invocation);
        RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
        return doInvoke(invocation, invokers, loadbalance);
    }

    protected List<Invoker<T>> list(Invocation invocation) throws RpcException {
        return directory.list(invocation);
    }

    protected LoadBalance initLoadBalance(List<Invoker<T>> invokers, Invocation
invocation) {
        if (CollectionUtils.isNotEmpty(invokers)) {
            return ExtensionLoader.getExtensionLoader(LoadBalance.class)
                .getExtension(invokers.get(0).getUrl()
                    .getMethodParameter(RpcUtils.getMethodName(invocation),
                        LOADBALANCE_KEY, DEFAULT_LOADBALANCE));
        } else {
            return ExtensionLoader.getExtensionLoader(LoadBalance.class)
                .getExtension(DEFAULT_LOADBALANCE);
        }
    }
    ...
}
```

公共逻辑 `select()`

`AbstractClusterInvoker` 有好几个扩展实现，子类的主要职责是处理容错，错误发生前后如何从多个服务实例中挑选到合适的一个，这业务逻辑所有子类都是一致的。

先看看由父类提供给子类调用的 `select()` 方法，其定义如下：

```java
protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation,
        List<Invoker<T>> invokers, List<Invoker<T>> selected) throws RpcException{
            ...
}
```

从其所有入参看，和抽象方法 `doInvoke()` 相比多出一个 `List<Invoker<T>> selected` 入参，也就是说逻辑转入到 `doInvoke()` 后，由子类在执行其策略的相关的业务时，使用相同的参数调用 `select()` 完成目标 `Invoker` 的选取操作。根据仔细查看源码， `selected` 中盛装的服务实例实际是要备当前的方法调用所被排除的。

`select()` 方法执行的总体逻辑操作如下：

1. 粘滞处理；
2. 使用负载均衡策略完成目标 `Invoker` 的挑选处理；
3. 如果被选到的服务实例不满足要求，调用 `reselect()` 方法做重选处理；

粘滞处理

在微服务开发中，总有些服务的实现是没法完全做到幂等的，前一个RPC方法调用和后一个有着某种关系，需要落实到同一个服务实例上。因此包括Dubbo在内的很多微服务框架都实现了粘滞特性，实际上负载均衡中的一致性hash策略也是一种粘滞手段，不同的是它完成的客户端到服务端的一对一隐式绑定，而 `ClusterInvoker` 实现的是前后两个RPC方法的粘滞。

上文已经提到过， `ClusterInvoker` 本身的单例特性是其实现粘滞的前提。

```java
protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation,
                           List<Invoker<T>> invokers, List<Invoker<T>> selected)
throws RpcException {

    if (CollectionUtils.isEmpty(invokers)) {
        return null;
    }
    String methodName = invocation == null ? StringUtils.EMPTY :
invocation.getMethodName();

    boolean sticky = invokers.get(0).getUrl()
            .getMethodParameter(methodName, CLUSTER_STICKY_KEY,
DEFAULT_CLUSTER_STICKY);

    //若粘滞对象并不包含在invokers被选范围则抹掉上一次的粘滞记忆
    if (stickyInvoker != null && !invokers.contains(stickyInvoker)) {
        stickyInvoker = null;
    }
    //ignore concurrency problem
    if (sticky && stickyInvoker != null && (selected == null ||
!selected.contains(stickyInvoker))) {
        if (availablecheck && stickyInvoker.isAvailable()) {
            return stickyInvoker;
        }
    }

    Invoker<T> invoker = doSelect(loadbalance, invocation, invokers, selected);

    //在配置需要粘滞的情况下，需要为下一次RPC方法调用记忆stickyInvoker
    if (sticky) {
        stickyInvoker = invoker;
    }
    return invoker;
}
```

如上述源码，一进入方法，就先获取配置项 `invoacation[methodName] + ".sticky"` 的值，如果上一个处理RPC方法 `stickyInvoker` 不在被排除集合中，并且它处于可用状态，则 `stickyInvoker` 就是当前RPC方法选中的服务实例。否则需要进入下一步获得执行RPC方法的 `Invoker` 实例，方法返回之前，将被选得的 `Invoker` 实例赋值给 `stickyInvoker` ，为下一次RPC方法保留记忆。

`doSelect()` 服务实例选取

服务实例 `Invoker` 对象的选取操作是由 `LoadBalance` 在给定的集合中使用特定算法策略完成的。然而如果被选中的实例 R 如果在被排除的集合中，或者处于不可用态，就需要执行 `reselect()` 重选逻辑。倘若重选也没有获得一个合适的返回 `null` 空值， `ClusterInvoker` 就会在集合中选择排在 R 的下一个或集合中首个位置的实例。过程中若出现异常， `doSelect()` 方法均给调用方返回 R 。

JAVA

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

    if (CollectionUtils.isEmpty(invokers)) {//集合为空时返回null
        return null;
    }
    if (invokers.size() == 1) {//仅有一个实例时直接返回首个
        return invokers.get(0);
    }
    Invoker<T> invoker = loadbalance.select(invokers, getUrl(), invocation);

    if ((selected != null && selected.contains(invoker))
            || (!invoker.isAvailable() && getUrl() != null && availablecheck)) {
        try {
            Invoker<T> rInvoker = reselect(loadbalance, invocation, invokers, selected,
availablecheck);
            if (rInvoker != null) {
                invoker = rInvoker;
            } else {
                int index = invokers.indexOf(invoker);
                try {
                    //Avoid collision
                    invoker = invokers.get((index + 1) % invokers.size());
                } catch (Exception e) {
                    //执行到这里表示invokers集合突然发生变化了，则直接返回loadbalance计算得到的实
例
                    logger.warn(e.getMessage() + " may because invokers list dynamic
change, ignore.", e);
                }
            }
        } catch (Throwable t) {
            logger.error("cluster reselect fail reason is :" + t.getMessage() +
                " if can not solve, you can set cluster.availablecheck=false in url",
t);
        }
    }
    return invoker;
}
```

源码中的可用状态监测为啥不在 `loadbalance.select()` 之前就赛选掉已经实例 `unavailable` 的了，初步看起来有点费解。微服务环境中，服务实例因上下线处于不可用状态是高概率事件，更不用说他们所处的包括网络在内的环境各异，导致随时断线或者无法响应。

`reselect()` 重选操作

负载均衡辛苦挑选到的微服务引用实例却不可用，这时 `ClusterInvoker` 只能使用 `reselect()` 执行重选处理。整个重选逻辑相对比较简单：

1. 首先从给定集合中剔除掉如下两种情况的实例：
    a. 不可用的；

b. 在被排除列表中的；

2. 根据剔除后的集合是否为空执行如下操作：

   a. 不为空，调用 `loadbalance.select()` 选一个；

   b. 为空，则对被排除列表中的可用服务实例集合做负载处理，该过程若没找到合适的则直接返回 `null` ；

JAVA

```
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.
    List<Invoker<T>> reselectInvokers = new ArrayList<>(
            invokers.size() > 1 ? (invokers.size() - 1) : invokers.size());

    // First, try picking a invoker not in `selected`.
    for (Invoker<T> invoker : invokers) {
        if (availablecheck && !invoker.isAvailable()) {
            continue;
        }

        if (selected == null || !selected.contains(invoker)) {
            reselectInvokers.add(invoker);
        }
    }

    if (!reselectInvokers.isEmpty()) {
        return loadbalance.select(reselectInvokers, getUrl(), invocation);
    }

    // Just pick an available invoker using loadbalance policy
    if (selected != null) {
        for (Invoker<T> invoker : selected) {
            if ((invoker.isAvailable()) // available first
                    && !reselectInvokers.contains(invoker)) {
                reselectInvokers.add(invoker);
            }
        }
    }
    if (!reselectInvokers.isEmpty()) {
        return loadbalance.select(reselectInvokers, getUrl(), invocation);
    }

    return null;
}
```

`ClusterInvoker` 实现

在 `dubbo-cluster` 这个模块中的 `META-INF/dubbo/internal` 目录下存在一个名为 `org.apache.dubbo.rpc.cluster.Cluster` 的SPI配置文件，其中配置如下，也就是说容错策略的实现远超文首提到的那几种。

```TEXT
mock=org.apache.dubbo.rpc.cluster.support.wrapper.MockClusterWrapper
failover=org.apache.dubbo.rpc.cluster.support.FailoverCluster
failfast=org.apache.dubbo.rpc.cluster.support.FailfastCluster
failsafe=org.apache.dubbo.rpc.cluster.support.FailsafeCluster
failback=org.apache.dubbo.rpc.cluster.support.FailbackCluster
forking=org.apache.dubbo.rpc.cluster.support.ForkingCluster
available=org.apache.dubbo.rpc.cluster.support.AvailableCluster
mergeable=org.apache.dubbo.rpc.cluster.support.MergeableCluster
Broadcast=org.apache.dubbo.rpc.cluster.support.BroadcastCluster
registryaware=org.apache.dubbo.rpc.cluster.support.RegistryAwareCluster
```

不过在继续看具体的策略实现前，还得先回到 `AbstractClusterInvoker` 抽象类来看看上文没有涉及到的 `checkInvokers()` 方法，如下：

```JAVA
protected void checkInvokers(List<Invoker<T>> invokers, Invocation invocation) {
    if (CollectionUtils.isEmpty(invokers)) {
        throw new RpcException(RpcException.NO_INVOKER_AVAILABLE_AFTER_FILTER,
            "Failed to invoke the method "
                + invocation.getMethodName() + " in the service " +
getInterface().getName()
                + ". No provider available for the service " +
directory.getUrl().getServiceKey()
                + " from registry " + directory.getUrl().getAddress()
                + " on the consumer " + NetUtils.getLocalHost()
                + " using the dubbo version " + Version.getVersion()
                + ". Please check if the providers have been started and registered.");
    }
}
```

> **NOTE**
>
> 关于用于实现 本地伪装 的装饰类 `MockClusterWrapper` ，请挪步《Dubbo服务降级》一文。

Available

这是一种最简单的实现，每次RPC请求进入，什么负载均衡、粘滞啥的统统不要，挨个迭代候选集中所有的实例，遇到的第一个可用实例便执行最终RPC方法调用并返回：

```java
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 RpcException {
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                return invoker.invoke(invocation);
            }
        }
        throw new RpcException("No provider available in " + invokers);
    }

}
```

RegistryAware

相比于 `Available` 类型的容错策略，该策略实现分为两部分，后面部分和 `Available` 完全相同，前面部分是挑到首个含有配置项 `"registry.default"` 为true的实例做RPC调用。

```java
public class RegistryAwareClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

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

    @Override
    @SuppressWarnings({"unchecked", "rawtypes"})
    public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers,
LoadBalance loadbalance) throws RpcException {
        // First, pick the invoker (XXXClusterInvoker) that comes from the local
registry, distinguish by a 'default' key.
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable() && invoker.getUrl().getParameter(REGISTRY_KEY +
"." + DEFAULT_KEY, false)) {
                return invoker.invoke(invocation);
            }
        }
        // If none of the invokers has a local signal, pick the first one available.
        for (Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                return invoker.invoke(invocation);
            }
        }
        throw new RpcException("No provider available in " + invokers);
    }
}
```

Broadcast(广播调用)

> *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。*

实现也很简单，先检查候选集是否存在可用的微服务引用实例，然后将候选集设入到本地异步上下文中，最后遍历候选集中所有的微服务引用实例，每个实例调用一次RPC方法，每次RPC方法调用时，临时变量 `result` 和 `exception` 分别记录正常和异常结果，方法执行到最后，如果 `exception` 不为空，则返回异常，否则返回 `result` ，如下所示：

JAVA

```
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 {
        checkInvokers(invokers, invocation);
        RpcContext.getContext().setInvokers((List) invokers);
        RpcException exception = null;
        Result result = null;
        for (Invoker<T> invoker : invokers) {
            try {
                result = invoker.invoke(invocation);
            } catch (RpcException e) {
                exception = e;
                logger.warn(e.getMessage(), e);
            } catch (Throwable e) {
                exception = new RpcException(e.getMessage(), e);
                logger.warn(e.getMessage(), e);
            }
        }
        if (exception != null) {
            throw exception;
        }
        return result;
    }

}
```

Failfast(快速失败)

> *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。*

和上述容错策略不同的是，该策略中会调用父类的 `select()` 方法执行诸如粘滞、负载均衡、重选处理，挑选到目标实例后便执行RPC方法调用，如果 catch 到异常，则转换为 `RpcException` 向上层抛出。

JAVA

```java
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 {
        checkInvokers(invokers, invocation);
        Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
        try {
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz
exception.
                throw (RpcException) e;
            }
            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 invocation. Last error is: " +
e.getMessage(),
                    e.getCause() != null ? e.getCause() : e);
        }
    }
}
```

Failsafe (失败安全)

> *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。*

和 `Failfast` 容错策略唯一不同的在于异常处理，catch 到异常时，只是在日志中简单记录异常信息，并返回一个完成态的的 `Result` 对象。

JAVA

```java
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 {
            checkInvokers(invokers, invocation);
            Invoker<T> invoker = select(loadbalance, invocation, invokers, null);
            return invoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Failsafe ignore exception: " + e.getMessage(), e);
            return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); //
ignore
        }
    }
}
```

Failover (失败自动切换)

> *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。*

相比而言，实现稍微复杂点，总体步骤如下：

1. 方法刚进入时，先检查候选列表的可用性，随后从配置中心目录服务的配置总线中以配置项 `url[methodName]+".retries"` 获取重试次数 `len`；
2. 声明临时变量：1） `le` 记录最后一次执行遇到的异常；2） `invoked` 记录已经执行过RPC调用的服务实例；3） `providers` 记录已经被重试过的其它实例所在的机器的Ip地址信息；
3. 进入循环执行如下逻辑处理，循环最多迭代 `len` 次：
    a. 为避免可用候选集发生变化，重新执行相关的前置处理
        i. 检查目录服务是否下线；
        ii. 调用 `list()` 方法列出所有可用的服务引用实例；
        iii. 检查候选列表的可用性；
    b. 使用最新的候选集，以及 `invoked` 等调用 `select()` 方法获取到一个可用的目标微服务引用实例 I；
    c. 将 I 加入到 `invoked`，同时在 finally 块中将 I 的 Ip 地址信息加入到 `providers` 中，并将已经发生变化的 `invoked` 设入到当前本地线程上下文中；

d. 使用 I 执行RPC远程方法调用；

e. 如果 RPC 方法调用成功直接返回结果，否则如果出现业务类异常，则直接抛异常处理，其它类型异常则记录到 `le` ，并进入下一循环，继续流程；

4. 循环执行完，使用 `le` 中的信息抛异常处理；

JAVA

```java
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 {
        List<Invoker<T>> copyInvokers = invokers;
        checkInvokers(copyInvokers, invocation);
        String methodName = RpcUtils.getMethodName(invocation);
        int len = getUrl().getMethodParameter(methodName, RETRIES_KEY, DEFAULT_RETRIES)
+ 1;
        if (len <= 0) {
            len = 1;
        }
        // retry loop.
        RpcException le = null; // last exception.
        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.
            if (i > 0) {
                checkWhetherDestroyed();
                copyInvokers = list(invocation);
                // check again
                checkInvokers(copyInvokers, invocation);
            }
            Invoker<T> invoker = select(loadbalance, invocation, copyInvokers,
invoked);
            invoked.add(invoker);
            RpcContext.getContext().setInvokers((List) invoked);
            try {
                Result result = invoker.invoke(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);
    }

}
```

Forking (并行调用)

> *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。*

从描述中不难看出，多个RPC请求同时发出，一旦获取首个成功返回的结果便完成了整个RPC调用，否则以为所有的RPC请求均以失败告终。

实现上稍显复杂，我们将对源码打散处理，逐段分析。

配置中心可以为该策略设置 `并行数-forks` 、 `超时-timeout` 参数，默认值分别为 2 和 1000ms。在具体处理时，如果配置值大于候选集可用个数，则将所有服务实例都加入到 `selected` 中，否则挨个调用 `select()` 方法，将挑选到的实例加入到 `selected` ，直到达到并发数目为止。

```java
//①并行环境准备
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<>();
    for (int i = 0; i < forks; i++) {
        Invoker<T> invoker = select(loadbalance, invocation, invokers, selected);
        if (!selected.contains(invoker)) {
            //Avoid add the same invoker several times.
            selected.add(invoker);
        }
    }
}
RpcContext.getContext().setInvokers((List) selected);
```

虽然默认上，每个微服务引用实例在处理RPC请求时，均以异步方式调用，但 `ForkingClusterInvoker` 还是采用了和其它策略迥异的实现方式，使用了在异步编程做任务调度时常用的 `BlockingQueue` ，专门开辟一个线程池给自己处理并行的RPC请求。由异步线程负责处理请求，请求的结果塞入阻塞队列，当前线程从队列取得元素，典型的生产者-消费者模型。

另外还增加了一个 `AtomicInteger` 类型的原子变量 count，如果异步线程中 catch 到异常，便计数加一，一旦该计数值为前述 `selected` 的大小时，便说明发出的所有并行RPC请求失败。

```java
//②生产环节
final AtomicInteger count = new AtomicInteger();
final BlockingQueue<Object> ref = new LinkedBlockingQueue<>();
for (final Invoker<T> invoker : selected) {
    executor.execute(() -> {
        try {
            Result result = invoker.invoke(invocation);
            ref.offer(result);
        } catch (Throwable e) {
            int value = count.incrementAndGet();
            if (value >= selected.size()) {
                ref.offer(e);
            }
        }
    });
}
```

在后面的消费环节中，当前线程利用阻塞队列同步从中同步获取到结果。若结果为 `Result` ，直接返回，否则结果为 `Throwable` 类型，或者请求没被及时处理而超时均会抛出异常。

```java
//③消费环节
try {
    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);
}
```

最后所有的代码汇总如下：

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

    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 {
            //①并行环境准备

            //②生产环节

            //③消费环节
        } finally {
            // clear attachments which is binding to current thread.
            RpcContext.getContext().clearAttachments();
        }
    }
}
```

Failback（失败自动恢复）

> *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。*

从下述代码来看， Failback 容错策略的实现和 `Failsafe`基本无甚差异，出现异常时，返回一个完成态的 Result 给调用方。

```java
public class FailbackClusterInvoker<T> extends AbstractClusterInvoker<T> {

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

    @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 invoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Failback to invoke method " + invocation.getMethodName() + ",
wait for retry in background. Ignored exception: "
                    + e.getMessage() + ", ", e);
            addFailed(loadbalance, invocation, invokers, invoker);
            return AsyncRpcResult.newDefaultAsyncResult(null, null, invocation); //
ignore
        }
    }
    ...
}
```

上述代码中 `addFailed(loadbalance, invocation, invokers, invoker);` 这一行是整个 `Failback` 的容错实现部分，背后牵涉内容比较多，要理解其实现，还需要先回到咱们的第一篇分析Dubbo源码实现的文章——《【一】定时轮算法 · HashedWheelTimer》，序列文章的起始就重点分析了Dubbo的定时机制是如何实现的，原因是Dubbo中有大量的实现需要用到定时器定期执行一些任务。

可以简单的将 **定时轮** 看做是 **定时任务调度器 + 任务** 的总和，调用方需要先实例化一个称为 `Timer` 的定时器，与此同时为其准备一个供任务调度的专用线程池，在需要的时机向其提交一个 `TimerTask` 定时任务。另外调用方自身被销毁时，也应该回收被 `Timer` 占用线程池资源。

RetryTimerTask

重试任务是异步的，并不会影响到当前RPC调用的及时响应，因此在提交重试任务时，需要缓存那一刻的有关执行环境变量，包括RPC方法的入参、使用到的负载均衡策略、候选的微服务引用实例等。任务被调度时执行如下操作：

1. 调用 `select()` 方法从候选集中选一个服务实例重新执行RPC方法调用；
2. 若捕获到异常，只要重试次数没有超过，便在 `Timer` 还存续的基础上再次执行重试任务；

JAVA

```java
private class RetryTimerTask implements TimerTask {
    private final Invocation invocation;
    private final LoadBalance loadbalance;
    private final List<Invoker<T>> invokers;
    private final int retries;
    private final long tick;
    private Invoker<T> lastInvoker;
    private int retryTimes = 0;

    RetryTimerTask(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>>
invokers, Invoker<T> lastInvoker, int retries, long tick) {
        this.loadbalance = loadbalance;
        this.invocation = invocation;
        this.invokers = invokers;
        this.retries = retries;
        this.tick = tick;
        this.lastInvoker=lastInvoker;
    }

    @Override
    public void run(Timeout timeout) {
        try {
            Invoker<T> retryInvoker = select(loadbalance, invocation, invokers,
Collections.singletonList(lastInvoker));
            lastInvoker = retryInvoker;
            retryInvoker.invoke(invocation);
        } catch (Throwable e) {
            logger.error("Failed retry to invoke method " + invocation.getMethodName()
+ ", waiting again.", e);
            if ((++retryTimes) >= retries) {
                logger.error("Failed retry times exceed threshold (" + retries + "), We
have to abandon, invocation->" + invocation);
            } else {
                rePut(timeout);
            }
        }
    }

    private void rePut(Timeout timeout) {
        if (timeout == null) {
            return;
        }

        Timer timer = timeout.timer();
        if (timer.isStop() || timeout.isCancelled()) {
            return;
        }

        timer.newTimeout(timeout.task(), tick, TimeUnit.SECONDS);
    }
}
```

源码中，没执行一次任务，总会将执行当前RPC方法的服务实例记录在 `lastInvoker` 零时变量中，后面的任务执行就将其排除在候选集之外，避免将同一个任务反复提交给一个出现故障的实例。

Timer 管理

如下源码所示， Timer 的采用了实例延迟初始化的方式，结合双检锁机制，确保处于并发环境下的 Failback 只会实例化一次 Timer 。 addFailed() 方法在 failTimer 已经赋值后，便实例化 RetryTimerTask 实例向其提交定时任务。

JAVA

```
private static final long RETRY_FAILED_PERIOD = 5;

private volatile Timer failTimer;

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());
    }
}

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

    int retriesConfig = getUrl().getParameter(RETRIES_KEY, DEFAULT_FAILBACK_TIMES);
    if (retriesConfig <= 0) {
        retriesConfig = DEFAULT_FAILBACK_TIMES;
    }
    int failbackTasksConfig = getUrl().getParameter(FAIL_BACK_TASKS_KEY,
DEFAULT_FAILBACK_TASKS);
    if (failbackTasksConfig <= 0) {
        failbackTasksConfig = DEFAULT_FAILBACK_TASKS;
    }
    retries = retriesConfig;
    failbackTasks = failbackTasksConfig;
}
```

源码的最后还将类的构造函数也呈现了，在 Directory 中可以为 Failback 配置重试次数 retries 和当前最大允许挂起的任务数 failbacktasks 参数。

最后在 Invoker 实例被销毁时，确保 failTimer 的资源被回收处理。

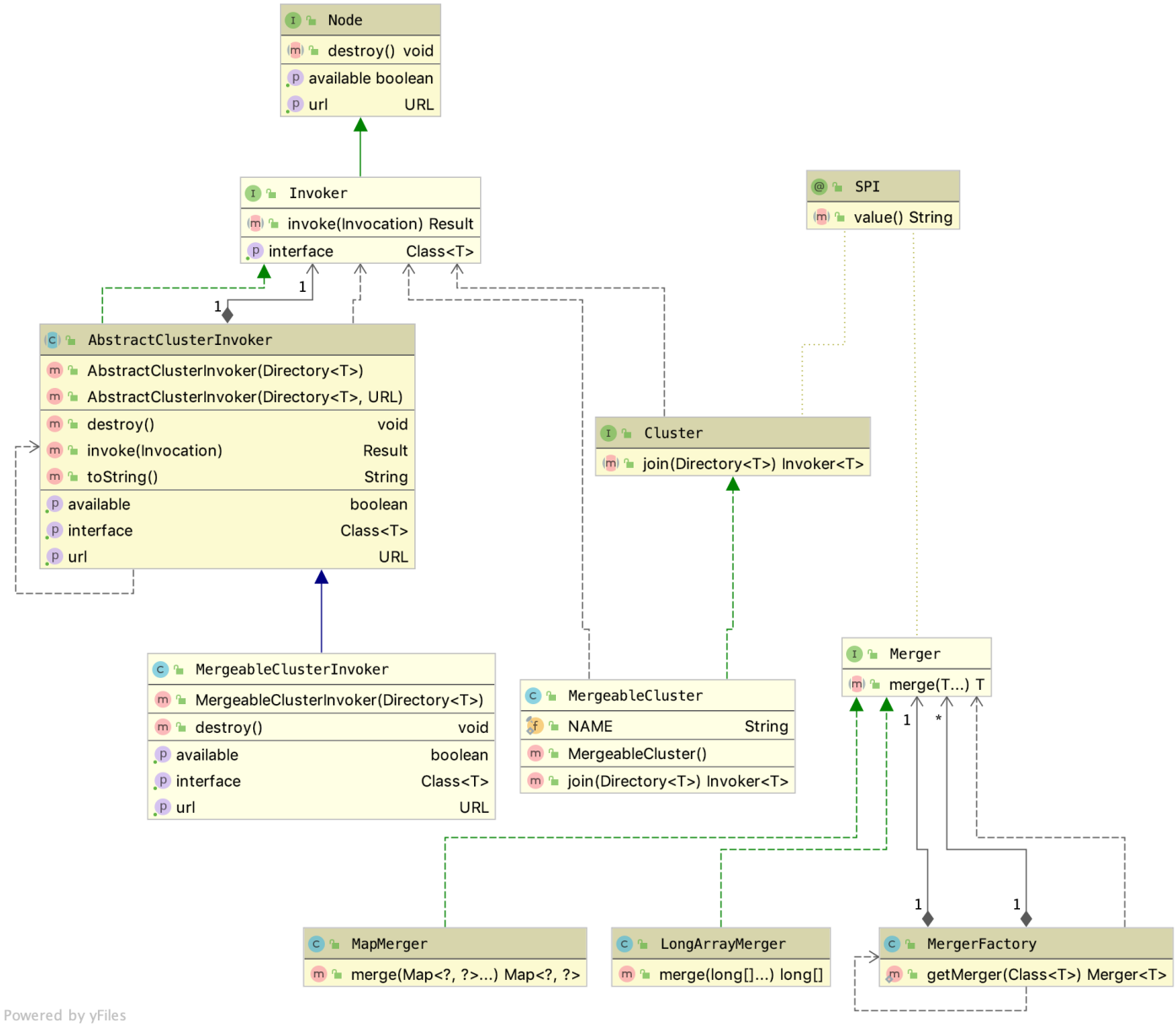
```java
@Override
public void destroy() {
    super.destroy();
    if (failTimer != null) {
        failTimer.stop();
    }
}
```

Mergeable (结果聚合)

> *按组合并返回结果，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。*

上述官网中的这一段表述中说明一个服务接口可以存在多种实现，也即对应着多个不同的微服务，用分组来区分他们，而调用客户端可以利用集群的 `Mergeable` 特性，汇总每种 group 的返回结果，实现聚合。

严格来说 `MergeableClusterInvoker` 这个 `ClusterInvoker` 实现并不属于一种容错策略，只是同样由扩展 `AbstractClusterInvoker` 抽象类实现的一种集群特性。下图是整个实现的类图：



Merger 聚合器扩展点

从类图中可以看出 Mergeable 的实现中需要用到另外一个名为 Merger 的扩展点，用于将服务接口的出参做聚合处理，也即将两个集合做归并处理。Dubbo提供的默认实现有 ArrayMerger、BooleanArrayMerger、ByteArrayMerger、CharArrayMerger、DoubleArrayMerger、FloatArrayMerger、IntArrayMerger、ListMerger、LongArrayMerger、MapMerger、SetMerger、ShortArrayMerger 。它们的实现大同小异，本文只看看 ArrayMerger 实现。

JAVA

```java
public class ArrayMerger implements Merger<Object[]> {

    public static final ArrayMerger INSTANCE = new ArrayMerger();

    @Override
    public Object[] merge(Object[]... items) {
        if (ArrayUtils.isEmpty(items)) {
            return new Object[0];
        }

        //先统计入参数组中，元素值为null的个数
        int i = 0;
        while (i < items.length && items[i] == null) {
            i++;
        }

        //如果入参数组中的所有元素均为null，则直接返回空的数组
        if (i == items.length) {
            return new Object[0];
        }

        //流程到这里说明入参数组中存在非null的元素

        //使用第一个非null的元素获取其数组元素的类型
        Class<?> type = items[i].getClass().getComponentType();

        //检测入参数组中的所有数组元素是否类型一致，并汇总这些非null数组中的所有元素个数
        int totalLen = 0;
        for (; i < items.length; i++) {
            if (items[i] == null) {
                continue;
            }
            Class<?> itemType = items[i].getClass().getComponentType();
            if (itemType != type) {
                throw new IllegalArgumentException("Arguments' types are different");
            }
            totalLen += items[i].length;
        }

        //汇总元素个数为0，则返回空的数组
        if (totalLen == 0) {
            return new Object[0];
        }

        //使用Array的反射功能申请一个数组
        Object result = Array.newInstance(type, totalLen);

        int index = 0;
        for (Object[] array : items) {
            if (array != null) {
                for (int j = 0; j < array.length; j++) {
                    //使用Array的反射功能设置数组指定索引位置的元素
                    Array.set(result, index++, array[j]);
                }
            }
        }
```

```
            return (Object[]) result;
        }
    }
```

既然是扩展点，用户当然可以自己提供一个扩展点具类，对于数据聚合合并结果这种需求在开发中也是常见的需求，实际生产中的数据不再是简单的基础类型，有的甚至有着比较复杂的合并规则，这也意味着必须留有给开发者扩展的切入点。

聚合器创建工厂 MergerFactory

在 MergeableClusterInvoker 实现中，需要动态的根据服务接口的出参类型获取到对应的聚合器，由于和Dubbo自身实现的SPI机制密切相关，建议先看看本序列文章中的《Dubbo之SPI扩展点加载》一文。

总体实现思路是，将所有 Merger 扩展点具类的单例以其所实现泛型参数 Class<?> 装入到声明了 static 的 Map<Class<?>, Merger<?>> 容器中，在需要的时候直接使用服务接口的出参类型作为 Key 键从该容器中获得对应的 Merger 实例。 另外总体而言Dubbo是很注重节约资源的，不会应用一初始化就做SPI的加载完成容器的填充，毕竟单就 Merger 而言，应用中可能压根没有用到结果聚合这一特性，预先加载就意味着内存资源的浪费，因而源码实现中采用了延迟加载方案。

在具体实现中，容器 Map 的具体类型为 ConcurrentMap<Class<?>, Merger<?>> ，原因是 Merger 的创建工厂 MergerFactory 是在 ClusterInvoker 所处并发环境环境下被使用。

JAVA

```
public class MergerFactory {

    private static final ConcurrentMap<Class<?>, Merger<?>> MERGER_CACHE =
            new ConcurrentHashMap<Class<?>, Merger<?>>();

    public static <T> Merger<T> getMerger(Class<T> returnType) {
        if (returnType == null) {
            throw new IllegalArgumentException("returnType is null");
        }

        Merger result;
        if (returnType.isArray()) {
            //若返回类型为数组，则获取其元素类型
            Class type = returnType.getComponentType();
            result = MERGER_CACHE.get(type);

            //result为null可能意味着SPI未加载
            if (result == null) {
                //执行SPI加载并获取扩展点具类实例，也即一个聚合器实现
                loadMergers();
                result = MERGER_CACHE.get(type);
            }
            if (result == null && !type.isPrimitive()) {
                result = ArrayMerger.INSTANCE;
            }
        } else {
            result = MERGER_CACHE.get(returnType);
            if (result == null) {
                loadMergers();
                result = MERGER_CACHE.get(returnType);
            }
        }
        return result;
    }

    static void loadMergers() {
        //首选获取SPI配置文件配置的所有支持扩展点实现，名称到具类的映射关系
        Set<String> names = ExtensionLoader.getExtensionLoader(Merger.class)
                .getSupportedExtensions();
        for (String name : names) {
            //逐个根据扩展点名称加载具类
            Merger m =
ExtensionLoader.getExtensionLoader(Merger.class).getExtension(name);

            //使用ReflectUtils.getGenericClass(m.getClass())获取到Merger实例所实现的泛型类型
            MERGER_CACHE.putIfAbsent(ReflectUtils.getGenericClass(m.getClass()), m);
        }
    }
}
```

MergeableClusterInvoker

相比容错类型的 `ClusterInvoker` ，该实现显得更加复杂，所有的代码都集中在一个方法中，但是总体步骤是比较清晰的，如下：

1. 确保有服务实例候选集可用；
2. 检查是否配置了 `url[invocation[methodName]] + ".merger"`：
   a. 若无这项配置，则选用候选集中的 首个可用优先 或 首个 服务实例发起RPC调用，随即返回；
   b. 若有，则继续下一步；
3. 遍历候选集，挨个异步发起PRC调用，并将结果装入 `Map<String, Result>` 类型的 `results` 容器；
4. 遍历 `results` ，逐个同步获取 `Result` 结果，如果异常则记录日志，否则装入 `List<Result>` 类型的 `resultList` 容器中；
5. 若 `resultList` 为空 或 被调用微服务接口方法的出参为void，则返回一个完成态的 `AsyncRpcResult` 对象，若只有一个元素直接返回，否则继续下一步；
6. 遍历 `resultList` 做结果聚合处理；

实现细节比较丰富，按照以往惯例，我们下面一段段打散加以分析。

第一段代码集中在 1、2 这两个步骤，不过在抛错处理时的 `e.isNoInvokerAvailableAfterFilter()` 这句代码值得注意，被调用方法中使用的异常标识 `NO_INVOKER_AVAILABLE_AFTER_FILTER` 值为6只在 `ClusterInvoker` 中用到，这说明被引用的微服务实例很有可能是其它虚拟 `Invoker` ，也即 `ClusterInvoker` 实例。

JAVA

```
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
        LoadBalance loadbalance) throws RpcException {
    checkInvokers(invokers, invocation);
    String merger = getUrl().getMethodParameter(invocation.getMethodName(),
MERGER_KEY);
    if (ConfigUtils.isEmpty(merger)) { // If a method doesn't have a merger, only
invoke one Group
        for (final Invoker<T> invoker : invokers) {
            if (invoker.isAvailable()) {
                try {
                    return invoker.invoke(invocation);
                } catch (RpcException e) {
                    if (e.isNoInvokerAvailableAfterFilter()) {
                        log.debug("No available provider for service" +
directory.getUrl().getServiceKey() + " on group " +
invoker.getUrl().getParameter(GROUP_KEY) + ", will continue to try another group.");
                    } else {
                        throw e;
                    }
                }
            }
        }
        return invokers.iterator().next().invoke(invocation);
    }
    ...
}
```

第二段代码对应步骤 3、4、5。在遍历所有服务实例候选集将 `Result` 结果装入到 `results` 时，会将RPC调用入参Invocation另行复制一份，拷入服务实例的配置后，并通过标记 `async` 告知Dubbo异步执行接下来的RPC调用。但是仔细分析代码发现会存在一个问题，我们都知道一个微服务存在的多个实例对应的是同一个ServiceKey `{group}/{interfaceName}:{version}`，代码中的 put 操作将会导致同一个微服务的所有在候选集中的实例都异步执行了，但只有最后一个实例的结果会被 catch 到。

JAVA

```java
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
        LoadBalance loadbalance) throws RpcException {
    ...
    Map<String, Result> results = new HashMap<>();
    for (final Invoker<T> invoker : invokers) {
        RpcInvocation subInvocation = new RpcInvocation(invocation, invoker);
        subInvocation.setAttachment(ASYNC_KEY, "true");
        results.put(invoker.getUrl().getServiceKey(), invoker.invoke(subInvocation));
    }

    List<Result> resultList = new ArrayList<Result>(results.size());

    for (Map.Entry<String, Result> entry : results.entrySet()) {
        Result asyncResult = entry.getValue();
        try {
            Result r = asyncResult.get();
            if (r.hasException()) {
                log.error("Invoke " + getGroupDescFromServiceKey(entry.getKey()) +
                                " failed: " + r.getException().getMessage(),
                        r.getException());
            } else {
                resultList.add(r);
            }
        } catch (Exception e) {
            throw new RpcException("Failed to invoke service " + entry.getKey() + ": "
+ e.getMessage(), e);
        }
    }

    if (resultList.isEmpty()) {
        return AsyncRpcResult.newDefaultAsyncResult(invocation);
    } else if (resultList.size() == 1) {
        return resultList.iterator().next();
    }

    Class<?> returnType;
    try {
        returnType = getInterface().getMethod(
                invocation.getMethodName(),
invocation.getParameterTypes()).getReturnType();
    } catch (NoSuchMethodException e) {
        returnType = null;
    }

    if (returnType == void.class) {
        return AsyncRpcResult.newDefaultAsyncResult(invocation);
    }
    ...
}
```

剩下的最后一段代码所涉细节比较多，相较也难以理解，根据得到 merge 配置值分为两段。

有时一个服务接口的方法出参，其类型并非一个数组，而是一个其它聚合，比如 `List` ，这类聚合有个特点，就是可以将其它同类型对象中的所有聚集合元素加入自身或新建的这个集合来，比如 `List` 的 `addAll` 方法，这时就可以做如下配置 `"."+methodName` ，Dubbo会采用递归的方式将上述 `resultList` 列表中的所有元素平摊汇总到一块：

XML

```xml
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger=".addAll" />
</dubbo:reference>
```

如下源码所示，这类型的方法的一个典型特征是，它只有一个入参，入参的类型和方法所属接口或类的类型一致；出参则分为两种情况，一种是 void 型的，另一种是和入参类型一致的，前一种表示合入参数中的集合，而后一种则是新建一个集合，将自身和参数中的集合一起合入。

JAVA

```
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
        LoadBalance loadbalance) throws RpcException {
    ...
    Object result = null;
    if (merger.startsWith(".")) {
        merger = merger.substring(1);
        Method method;
        try {
            //获取到方法名称
            method = returnType.getMethod(merger, returnType);
        } catch (NoSuchMethodException e) {
            throw new RpcException("Can not merge result because missing method [ " +
merger + " ] in class [ " +
                    returnType.getClass().getName() + " ]");
        }
        //不要求配置方法是public的，private的方法使用反射设置可访问
        if (!Modifier.isPublic(method.getModifiers())) {
            method.setAccessible(true);
        }

        //取出第一个元素，预备后续元素在第一个基础上进行合并处理
        result = resultList.remove(0).getValue();
        try {
            if (method.getReturnType() != void.class
                    && method.getReturnType().isAssignableFrom(result.getClass())) {
                //返回类型不为void的情况，result指针不断变化，每次得到一个新的集合对象，
                //当前的所有元素汇总了每一次迭代的
                for (Result r : resultList) {
                    result = method.invoke(result, r.getValue());
                }
            } else {
                //将resultList中的所有集合中元素都汇总到最初的那个result上
                for (Result r : resultList) {
                    method.invoke(result, r.getValue());
                }
            }
        } catch (Exception e) {
            throw new RpcException("Can not merge result: " + e.getMessage(), e);
        }
    } else {...}
    ...
}
```

在使用结果聚合特性时，经常会有如下配置：

XML

```
<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb" merger="true" />
```

其中 merge 的值可以配置成 true 或者 default，表示使用 `MergerFactory` 这个聚合器创建工厂根据服务接口方法的返回参数的类型获取到对应的聚合器。明确指定名称则使用SPI机制根据配置名获取对应的聚合器已缓存。

JAVA

```java
protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers,
        LoadBalance loadbalance) throws RpcException {
    ...
    Object result = null;
    if (merger.startsWith(".")) {...}
    else {
        Merger resultMerger;
        if (ConfigUtils.isDefault(merger)) {
            resultMerger = MergerFactory.getMerger(returnType);
        } else {
            resultMerger =
ExtensionLoader.getExtensionLoader(Merger.class).getExtension(merger);
        }
        if (resultMerger != null) {
            List<Object> rets = new ArrayList<Object>(resultList.size());
            for (Result r : resultList) {
                rets.add(r.getValue());
            }
            result = resultMerger.merge(
                    rets.toArray((Object[]) Array.newInstance(returnType, 0)));
        } else {
            throw new RpcException("There is no merger to merge result.");
        }
    }
    return AsyncRpcResult.newDefaultAsyncResult(result, invocation);
}
```

完结