dubbo中提供了多种集群调用策略：

1、FailbackClusterInvoker：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、FailfastClusterInvoker: 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;

3、FailoverClusterInvoker: 失败转移，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;

4、FailsafeClusterInvoker: 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;

5、FokingClusterInvoker: 并行调用，只要一个成功即返回，通常用于实时性要求较高的操作，但需要浪费更多服务资源;

6、MergeableClusterInvoker：合并多个组的返回数据;

开发者可以根据实际情况选择合适的策略，这里我们选择FailoverClusterInvoker（官方推荐）进行讲解，通过它来了解集群调用处理的问题，了解它以后其他的策略也很容易了。

由多个相同服务共同组成的一套服务，通过分布式的部署，达到服务的高可用，这就是集群。与单机的服务不同的是，我们至少需要：1、地址服务（Directory）；2、负载均衡（LoadBalance）。 地址服务用于地址的管理，如缓存地址、服务上下线的处理、对外提供地址列表等，通过地址服务，我们可以知道所有可用服务的地址信息。负载均衡，则是通过一定的算法将压力分摊到各个服务上。 好了，知道这两个概念后，我们开始正式的代码阅读。

当应用需要调用服务时，会通过invoke方法发起调用(AbstractClusterInvoker)：

[java] view plain copy

```
1.     public Result invoke(final Invocation invocation) throws
RpcException {
2.         // 是否被销毁
3.         checkWheatherDestoried();
4.
5.         LoadBalance loadbalance;
```

```
6.          // 通过地址服务获取所有可用的地址信息
7.          List<Invoker<T>> invokers = list(invocation);
8.          
9.          if (invokers != null && invokers.size() > 0) {
10.             // 如果存在地址信息，则根据地址中的配置加载LoadBalance，注意负责均衡策略配置的优先级 privider > consumer
11.             loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).getExtension(invokers.get(0).getUrl()
12.                     .getMethodParameter(invocation.getMethodName(),Constants.LOADBALANCE_KEY, Constants.DEFAULT_LOADBALANCE));
13.         } else {
14.             // 如果暂时没有地址信息，则使用默认的负载均衡策略策略 (random)
15.             loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).getExtension(Constants.DEFAULT_LOADBALANCE);
16.         }
17.         // 如果是异步的话需要加入相应的信息
18.         RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
19.         // 根据地址及负载均衡策略发起调用
20.         return doInvoke(invocation, invokers, loadbalance);
21.     }
22. 
23.     protected  List<Invoker<T>> list(Invocation invocation) throws RpcException {
24.      List<Invoker<T>> invokers = directory.list(invocation);
25.      return invokers;
26.     }
```

doInvoke则是各个子类来实现，以FailoverClusterInvoker为例：

[java] view plain copy

```
1. public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {
2.     List<Invoker<T>> copyinvokers = invokers;
3.     // 检查地址列表是否正确（需要确保有可用的地址）
4.     checkInvokers(copyinvokers, invocation);
5.     // 从retries参数获取重试的次数，如retries=3，则最大可能调用的次数为4
6.     // 需要注意的是默认的重试次数为2（及最多执行3次），对于一些写服务来
```

```
说，如果无法做到幂等，最好是将retries参数设为0，或者使用failfast策略
7.      int len =
getUrl().getMethodParameter(invocation.getMethodName(),
Constants.RETRIES_KEY, Constants.DEFAULT_RETRIES) + 1;
8.      if (len <= 0) {
9.          len = 1;
10.     }
11.     // retry loop.
12.     RpcException le = null; // last exception.
13.     List<Invoker<T>> invoked = new ArrayList<Invoker<T>>
(copyinvokers.size()); // invoked invokers.
14.     Set<String> providers = new HashSet<String>(len);
15.     // 发起指定次数的调用，一旦其中一次成功则返回
16.     for (int i = 0; i < len; i++) {
17.         //重试时，进行重新选择，避免重试时invoker列表已发生变化.
18.         //注意：如果列表发生了变化，那么invoked判断会失效，因为
invoker示例已经改变
19.         if (i > 0) {
20.             checkWheatherDestoried();
21.             copyinvokers = list(invocation);
22.             //重新检查一下
23.             checkInvokers(copyinvokers, invocation);
24.         }
25.         // 根据负载均衡算法得到一个地址
26.         Invoker<T> invoker = select(loadbalance, invocation,
copyinvokers, invoked);
27.         // 记录发起过调用的地址，防止重试时调用了已经调用过的地址
28.         invoked.add(invoker);
29.         //
30.         RpcContext.getContext().setInvokers((List)invoked);
31.         try {
32.             // 通过之前选出的地址进行调用
33.             Result result = invoker.invoke(invocation);
34.             // 调用成功，判断是否重试过，如果重试过，记录下警告信息，记
录失败的地址
35.             if (le != null && logger.isWarnEnabled()) {
36.                 logger.warn("Although retry the method " +
invocation.getMethodName()
37.                         + " in the service " +
getInterface().getName()
38.                         + " was successful by the provider "
+ invoker.getUrl().getAddress()
39.                         + ", but there have been failed
providers " + providers
40.                         + " (" + 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) {
                // 如果是业务异常则直接抛出错误，其他（如超时等错误）则不重试
                if (e.isBiz()) { // biz exception.
                    throw e;
                }
                le = e;
            } catch (Throwable e) {
                le = new RpcException(e.getMessage(), e);
            } finally {
                // 发生过调用的地址记录下来
                providers.add(invoker.getUrl().getAddress());
            }
        }
        throw new RpcException(le != null ? le.getCode() : 0,
"Failed to invoke the method "
                + invocation.getMethodName() + " 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 != null ? le.getMessage() : ""), le != null
&& le.getCause() != null ? le.getCause() : le);
    }
```

可以看到failover策略实现很简单，得到地址信息后，通过负载均衡算法选取一个地址来发送请求， 如果产生了非业务异常则按照配置的次数进行重试。

下面我们来看看地址的选取过程：

[java] view plain copy

```java
    /**
     * 使用loadbalance选择invoker.</br>
     * a)先lb选择，如果在selected列表中 或者 不可用且做检验时，进入下一步(重选),否则直接返回</br>
     * b)重选验证规则：selected > available .保证重选出的结果尽量不在select中，并且是可用的
     *
     * @param availablecheck 如果设置true，在选择的时候先选invoker.available == true
     * @param selected 已选过的invoker.注意：输入保证不重复
     *
     */
    protected Invoker<T> select(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers, List<Invoker<T>> selected) throws RpcException {
        if (invokers == null || invokers.size() == 0)
            return null;
        String methodName = invocation == null ? "" : invocation.getMethodName();

        // 如果sticky为true，则该接口上的所有方法使用相同的provider
        boolean sticky = invokers.get(0).getUrl().getMethodParameter(methodName,Constants.CLUSTER_STICKY_KEY, Constants.DEFAULT_CLUSTER_STICKY) ; {
            //如果之前的provider已经不存在了则将其设置为null
            if ( stickyInvoker != null && !invokers.contains(stickyInvoker) ){
                stickyInvoker = null;
            }
            // 如果sticky为true，且之前有调用过的provider且该provider未失败则继续使用该provider
            if (sticky && stickyInvoker != null && (selected == null || !selected.contains(stickyInvoker))){
                if (availablecheck && stickyInvoker.isAvailable()){
                    return stickyInvoker;
                }
            }
        }
        // 重新选择invoker
        Invoker<T> invoker = doselect(loadbalance,
```

```
invocation, invokers, selected);
30.          if (sticky){
31.             stickyInvoker = invoker;
32.          }
33.
34.          return invoker;
35.      }
36.
37.      private Invoker<T> doselect(LoadBalance loadbalance,
Invocation invocation, List<Invoker<T>> invokers,
List<Invoker<T>> selected) throws RpcException{
38.         if (invokers == null || invokers.size() == 0)
39.              return null;
40.
41.          if (invokers.size() == 1)
42.              return invokers.get(0);
43.          // 如果只有两个invoker,且产生过失败，则退化成轮循
44.          if (invokers.size() == 2 && selected != null &&
selected.size() > 0) {
45.              return selected.get(0) == invokers.get(0) ?
invokers.get(1) : invokers.get(0);
46.          }
47.
48.          // 通过负载均衡算法得到一个invoker
49.          Invoker<T> invoker = loadbalance.select(invokers,
getUrl(), invocation);
50.          //如果 selected中包含（优先判断） 或者 需要检测可用性且当前选
择的invoker不可用 则重新选择.
51.          if( (selected != null && selected.contains(invoker))
||(!invoker.isAvailable() && getUrl()!=null && availablecheck)){
52.              try{
53.                  Invoker<T> rinvoker = reselect(loadbalance,
invocation, invokers, selected, availablecheck);
54.                  if(rinvoker != null){
55.                      invoker = rinvoker;
56.                  }else{
57.                      //如果重新选择没有选出invoker，则选第一次选的下
一个invoker.
58.                      int index = invokers.indexOf(invoker);
59.                      try{
60.                          //最后在避免碰撞
61.                          invoker = index <invokers.size()-1?
invokers.get(index+1) :invoker;
62.                      }catch (Exception e) {
63.                         logger.warn(e.getMessage()+" may
```

```
because invokers list dynamic change, ignore.",e);
                    }
                }
            }catch (Throwable t){
                logger.error("clustor relselect fail reason
is :"+t.getMessage() +" if can not slove ,you can set
cluster.availablecheck=false in url",t);
            }
        }

        return invoker;
    }

    /**
     * 重选，先从非selected的列表中选择，没有在从selected列表中选择.
     * @param loadbalance
     * @param invocation
     * @param invokers
     * @param selected
     * @return
     * @throws RpcException
     */
    private Invoker<T> reselect(LoadBalance
loadbalance,Invocation invocation,
                                List<Invoker<T>> invokers,
List<Invoker<T>> selected ,boolean availablecheck)
            throws RpcException {

        //预先分配一个，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());

        //先从非select中选
        if( availablecheck ){ //选isAvailable 的非select
            for(Invoker<T> invoker : invokers){
                if(invoker.isAvailable()){
                    if(selected ==null ||
!selected.contains(invoker)){
                        reselectInvokers.add(invoker);
                    }
                }
            }
            if(reselectInvokers.size()>0){
                return  loadbalance.select(reselectInvokers,
```

```
getUrl(), invocation);
101.             }
102.         }else{ //选全部非select
103.             for(Invoker<T> invoker : invokers){
104.                 if(selected ==null ||
!selected.contains(invoker)){
105.                     reselectInvokers.add(invoker);
106.                 }
107.             }
108.             if(reselectInvokers.size()>0){
109.                 return  loadbalance.select(reselectInvokers,
getUrl(), invocation);
110.             }
111.         }
112.         //最后从select中选可用的.
113.         {
114.             if(selected != null){
115.                 for(Invoker<T> invoker : selected){
116.                     if((invoker.isAvailable()) //优先选
available
117.                                 &&
!reselectInvokers.contains(invoker)){
118.                         reselectInvokers.add(invoker);
119.                     }
120.                 }
121.             }
122.             if(reselectInvokers.size()>0){
123.                 return  loadbalance.select(reselectInvokers,
getUrl(), invocation);
124.             }
125.         }
126.         return null;
127.     }
```

选择invoker的过程如下：

1、如果配置sticky为true，则查看之前是否有已经调用过的invoker，如果有且可用则直接使用

2、如果只有一个地址，则直接使用该地址

3、如果有两个地址，且已经调用过一个地址，则使用另一个地址

4、第一次使用负载均衡算法得到一个invoker。满足两个条件则使用该invoker，如果不满足则继续第5步的重新选择

条件1 该地址在该请求中未使用（注一次请求含第一次调用及后

面的重试）

条件2 设置了检测可用性且可用 或 没设置检测可用性

5、如果设置了检测可用性则获取所有可用且本次请求未使用过的invoker，如果未设置则获取所有本次请求未使用过的invoker，

如果得到的invoker不为空，则使用负载均衡从这批invoker中选择一个

6、如果还是没有选出invoker则从已经使用过的invoker中找可用的invoker，从这些可用的invoker中利用负载均衡算法得到一个invoker

7、如果以上步骤均未选出invoker，则选择第4步得到的invoker的下一个invoker，如果第4步得到的invoker已经是最后一个则直接选此invoker

可以看到虽然有负载均衡策略，但仍然有很多分支状况需要处理。

选出invoker后就可以进行调用了。后面的过程由单独的文章继续讲解。我们回过头来看看负载均衡算法的实现， 首先看看负载均衡的接口：

[java] view plain copy

```java
1. @SPI(RandomLoadBalance.NAME)
2. public interface LoadBalance {
3.
4.     /**
5.      * select one invoker in list.
6.      *
7.      * @param invokers invokers.
8.      * @param url refer url
9.      * @param invocation invocation.
10.      * @return selected invoker.
11.      */
12.     @Adaptive("loadbalance")
13.     <T> Invoker<T> select(List<Invoker<T>> invokers, URL url,
    Invocation invocation) throws RpcException;
14.
15. }
```

负载均衡提供的select方法共有三个参数，invokers：可用的服务列表，url:包含consumer的信息，invocation:当前调用的信息。 默认的负载均衡算法为RandomLoadBalance， 这里我们就先讲讲这个随机调度算法。

[java] view plain copy

```
1. public class RandomLoadBalance extends AbstractLoadBalance {
2. 
3.     public static final String NAME = "random";
4. 
5.     private final Random random = new Random();
6. 
7.     protected <T> Invoker<T> doSelect(List<Invoker<T>>
invokers, URL url, Invocation invocation) {
8.         int length = invokers.size(); // 总个数
9.         int totalWeight = 0; // 总权重
10.         boolean sameWeight = true; // 权重是否都一样
11.         for (int i = 0; i < length; i++) {
12.             int weight = getWeight(invokers.get(i),
invocation);
13.             totalWeight += weight; // 累计总权重
14.             if (sameWeight && i > 0
15.                     && weight != getWeight(invokers.get(i -
1), invocation)) {
16.                 sameWeight = false; // 计算所有权重是否一样
17.             }
18.         }
19.         if (totalWeight > 0 && ! sameWeight) {
20.             // 如果权重不相同且权重大于0则按总权重数随机
21.             int offset = random.nextInt(totalWeight);
22.             // 并确定随机值落在哪个片断上
23.             for (int i = 0; i < length; i++) {
24.                 offset -= getWeight(invokers.get(i),
invocation);
25.                 if (offset < 0) {
26.                     return invokers.get(i);
27.                 }
28.             }
29.         }
30.         // 如果权重相同或权重为0则均等随机
31.         return invokers.get(random.nextInt(length));
32.     }
33. 
34. }
```

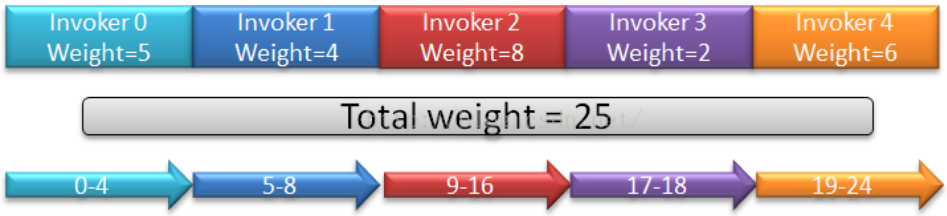
随机调度算法分两种情况：

1、当所有服务提供者权重相同或者无权重时，根据列表size得到一个值，再随机出一个[0, size)的数值，根据这个数值取对应位置的服务提供

者；

2、计算所有服务提供者权重之和，例如以下5个Invoker，总权重为25，则随机出[0, 24]的一个值，根据各个Invoker的区间来取Invoker，

如随机值为10，则选择Invoker2；



需要注意的是取权重的方法getWeight不是直接取配置中的权重，其算法如下：

[java] view plain copy

```
1. protected int getWeight(Invoker<?> invoker, Invocation
invocation) {
2.         // 先获取provider配置的权重（默认100）
3.         int weight =
invoker.getUrl().getMethodParameter(invocation.getMethodName(),
Constants.WEIGHT_KEY, Constants.DEFAULT_WEIGHT);
4.         if (weight > 0) {
5.                 // 获取provider的启动时间
6.             long timestamp =
invoker.getUrl().getParameter(Constants.TIMESTAMP_KEY, 0L);
7.             if (timestamp > 0L) {
8.                         // 计算出启动时长
9.                 int uptime = (int) (System.currentTimeMillis()
- timestamp);
10.                         // 获取预热时间（默认600000，即10分钟），注意warmup不是provider的基本参数，需要通过dubbo:paramater配置
11.                 int warmup =
invoker.getUrl().getParameter(Constants.WARMUP_KEY,
Constants.DEFAULT_WARMUP);
12.                         // 如果启动时长小于预热时间，则需要降权。 权重计算方式为启动时长占预热时间的百分比乘以权重，
13.                         // 如启动时长为20000ms，预热时间为
```

```
60000ms，权重为120，则最终权重为 120 * （1/3） = 40，
                            // 注意calculateWarmupWeight使用float进行计算，因此结果并不精确。
                            if (uptime > 0 && uptime < warmup) {
                        weight = calculateWarmupWeight(uptime, warmup, weight);
                    }
                }
            }
            return weight;
        }

        static int calculateWarmupWeight(int uptime, int warmup, int weight) {
            int ww = (int) ( (float) uptime / ( (float) warmup / (float) weight ) );
            return ww < 1 ? 1 : (ww > weight ? weight : ww);
        }
```

到这里，负载均衡的随机调度算法就分析完了，实现还是比较简单的。dubbo还实现了其他几个算法：

RoundRobinLoadBalance：轮询调度算法（2.5.3版本有bug，2.5.4-snapshot正常，有兴趣的请看后面这个版本的代码） 例如invoker0-权重3，invoker1-权重1，invoker2-权重2，则选取的invoker顺序依次是：第一轮：0,1(本轮invoker1消耗完）,2,0,2(本轮invoker2消耗完）,0(本轮invoker0消耗完）， 第二轮重复第一轮的顺序。

LeastActiveLoadBalance：最少活跃调用数调度算法，通过活跃数统计，找出活跃数最少的provider，如果只有一个最小的则直接选这个，如果活跃数最少的provider有多个，则用与RandomLoadBalance相同的策略来从这几个provider中选取一个。

ConsistentHashLoadBalance：一致性hash调度算法，根据参数计算得到一个provider，后续相同的参数使用同样的provider。