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
2019-04-15
Dubbo

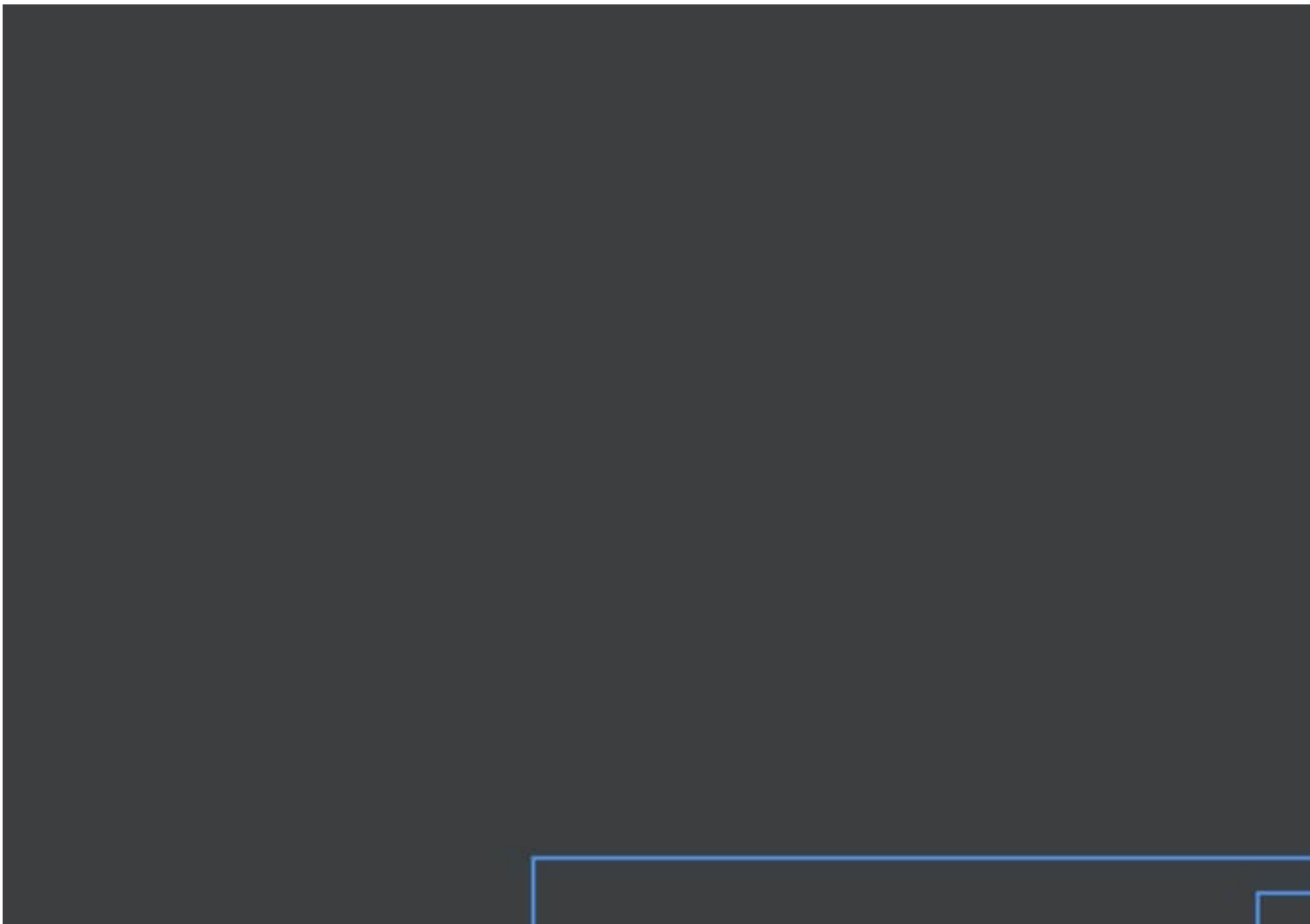
精尽 Dubbo 源码解析 —— 集群容错（四）之 LoadBalance 实现

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（三）之 Directory 实现》 一文，分享 dubbo-cluster 模块， loadbalance 包，各种 LoadBalance 实现类。

LoadBalance 子类如下图：



我们可以看到，目前一共有四个子类，意味着内置了四种负载均衡的选择算法。

老艿艿：本文对应 《Dubbo 用户指南 —— 负载均衡》 文档。

2. LoadBalance

com.alibaba.dubbo.rpc.cluster.LoadBalance ， LoadBalance 接口。代码如下：

```
@SPI(RandomLoadBalance.NAME)
public interface LoadBalance {

    /**
     * select one invoker in list.
     *
     * 从 Invoker 集合中，选择一个
     *
     * @param invokers   invokers.
     * @param url        refer url
     * @param invocation invocation.
     * @return selected invoker.
     */
    @Adaptive("loadbalance")
    <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) throws RpcException;

}
```

@SPI(RandomLoadBalance.NAME) 注解，Dubbo SPI 拓展点，默认为 "random" ，即随机。
@Adaptive 注解，基于 Dubbo SPI Adaptive 机制，加载对应的 Cluster 实现，使用 URL.loadbalance 属性。
#selectList<Invoker<T>>, URL, Invocation) 接口方法，从 Invoker 集合中，选择一个。

3. AbstractLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.AbstractLoadBalance ，实现 LoadBalance 接口，LoadBalance 抽象类，提供了权重计算的功能。

3.1 select

#select(List<Invoker<T>>, URL, Invocation) 实现方法，默认只有一个 Invoker 时，直接选择返回。代码如下：

```
    @Override
public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    if (invokers == null || invokers.isEmpty()) {
        return null;
    }
    if (invokers.size() == 1) {
        return invokers.get(0);
    }
    return doSelect(invokers, url, invocation);
}
```

子类实现 #doSelect(List<Invoker<T>>, URL, Invocation) 抽象方法，提供自定义的负载均衡策略。

```
protected abstract <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation);
```

3.2 getWeight

```
protected int getWeight(Invoker<?> invoker, Invocation invocation) {
    // 获得 weight 配置，即服务权重。默认为 100
    int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.WEIGHT_KEY, Constants.DEFA
    if (weight > 0) {
        long timestamp = invoker.getUrl().getParameter(Constants.REMOTE_TIMESTAMP_KEY, 0L);
        if (timestamp > 0L) {
            // 获得启动总时长
            int uptime = (int) (System.currentTimeMillis() - timestamp);
            // 获得预热需要总时长。默认为 10 * 60 * 1000 = 10 分钟
            int warmup = invoker.getUrl().getParameter(Constants.WARMUP_KEY, Constants.DEFAULT_WARMUP);
            // 处于预热中，计算当前的权重
            if (uptime > 0 && uptime < warmup) {
                weight = calculateWarmupWeight(uptime, warmup, weight);
            }
        }
    }
    return weight;
}
```

考虑到 JVM 自身会有预热的过程，所以服务提供者一启动就直接承担 100% 的流量，可能会出现很吃力的情况。因此权重的计算，默认自带了预热的过程。#calculateWarmupWeight(uptime, warmup, weight) 静态方法，代码如下：

```
static int calculateWarmupWeight(int uptime, int warmup, int weight) {
    // 计算权重
    int ww = (int) ((float) uptime / ((float) warmup / (float) weight));
    // 权重范围为 [0, weight] 之间
    return ww < 1 ? 1 : (ww > weight ? weight : ww);
}
```

- 计算权重的代码这么写看起来比较“绕”，我们来修改成 (uptime / warmup) * weight ，是否就好理解多了，相当于进度百分比 * 权重。
- 如下是我飞哥举的一个例子，感觉非常赞。

  根据calculateWarmupWeight()方法实现可知，随着provider的启动时间越来越长，慢慢提升权重直到weight，且权重最小值为1，所以：

  - 如果 provider 运行了 1 分钟，那么 weight 为 10，即只有最终需要承担的 10% 流量；
  - 如果 provider 运行了 2 分钟，那么 weight 为 20，即只有最终需要承担的 20% 流量；
  - 如果 provider 运行了 5 分钟，那么 weight 为 50，即只有最终需要承担的 50% 流量；
    … …
  - 如果 provider 运行了 10 分钟，那么 weight 为 100，即只有最终需

要承担的 100% 流量；

"weight" 配置项，默认为 100 。

"warmup" 配置项，默认为 10 * 60 * 1000 = 10 分钟。

4. RandomLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance ，实现 AbstractLoadBalance 抽象类，随机，按权重设置随机概率。

在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

```
 1: public class RandomLoadBalance extends AbstractLoadBalance {
 2:
 3:     public static final String NAME = "random";
 4:
 5:     private final Random random = new Random();
 6:
 7:     @Override
 8:     protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
 9:         int length = invokers.size(); // Number of invokers
10:         int totalWeight = 0; // The sum of weights
11:         boolean sameWeight = true; // Every invoker has the same weight?
12:         // 计算总权限
13:         for (int i = 0; i < length; i++) {
14:             int weight = getWeight(invokers.get(i), invocation); // 获得权重
15:             totalWeight += weight; // Sum
16:             if (sameWeight && i > 0 && weight != getWeight(invokers.get(i - 1), invocation)) {
17:                 sameWeight = false;
18:             }
19:         }
20:         // 权重不相等，随机后，判断在哪个 Invoker 的权重区间中
21:         if (totalWeight > 0 && !sameWeight) {
22:             // 随机
23:             // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly base
24:             int offset = random.nextInt(totalWeight);
25:             // Return a invoker based on the random value.
26:             // 区间判断
27:             for (Invoker<T> invoker : invokers) {
28:                 offset -= getWeight(invoker, invocation);
29:                 if (offset < 0) {
30:                     return invoker;
31:                 }
32:             }
33:         }
34:         // 权重相等，平均随机
35:         // If all invokers have the same weight value or totalWeight=0, return evenly.
36:         return invokers.get(random.nextInt(length));
37:     }
38:
39: }
```

第 12 至 19 行：计算总权重，并判断所有 Invoker 是否相同权重。
第 20 至 33 行：权重不相等，随机权重后，判断在哪个 Invoker 的权重区间中。

第 36 行：权重相等，直接随机选择 Invoker 即可。相等于对【第 20 至 33 行】的优化。

算法说明

FROM 飞哥的 《dubbo源码-负载均衡》

假定有3台dubbo provider:

10.0.0.1:20884, weight=2
10.0.0.1:20886, weight=3
10.0.0.1:20888, weight=4

随机算法的实现：
totalWeight=9;

假设offset=1（即random.nextInt(9)=1）
1-2=-1<0？是，所以选中 10.0.0.1:20884, weight=2
假设offset=4（即random.nextInt(9)=4）
4-2=2<0？否，这时候offset=2， 2-3<0？是，所以选中 10.0.0.1:20886, weight=3
假设offset=7（即random.nextInt(9)=7）
7-2=5<0？否，这时候offset=5， 5-3=2<0？否，这时候offset=2， 2-4<0？是，所以选中 10.0.0.1:20888, weight=4

5. RoundRobinLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance ，实现 AbstractLoadBalance 抽象类，轮循，按公约后的权重设置轮循比率。

存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

```
 1: public class RoundRobinLoadBalance extends AbstractLoadBalance {
 2:
 3:     public static final String NAME = "roundrobin";
 4:
 5:     /**
 6:      * 服务方法与计数器的映射
 7:      *
 8:      * KEY：serviceKey + "." + methodName
 9:      */
10:     private final ConcurrentMap<String, AtomicPositiveInteger> sequences = new ConcurrentHashMap<String, AtomicPo
11:
12:     @Override
13:     protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
14:         String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
15:         int length = invokers.size(); // Number of invokers
16:         int maxWeight = 0; // The maximum weight
17:         int minWeight = Integer.MAX_VALUE; // The minimum weight
18:         final LinkedHashMap<Invoker<T>, IntegerWrapper> invokerToWeightMap = new LinkedHashMap<Invoker<T>, Intege
19:         int weightSum = 0;
20:         // 计算最小、最大权重，总的权重和。
21:         for (int i = 0; i < length; i++) {
22:             int weight = getWeight(invokers.get(i), invocation);
23:             maxWeight = Math.max(maxWeight, weight); // Choose the maximum weight
```

```
24:             minWeight = Math.min(minWeight, weight); // Choose the minimum weight
25:             if (weight > 0) {
26:                 invokerToWeightMap.put(invokers.get(i), new IntegerWrapper(weight));
27:                 weightSum += weight;
28:             }
29:         }
30:         // 获得 AtomicPositiveInteger 对象
31:         AtomicPositiveInteger sequence = sequences.get(key);
32:         if (sequence == null) {
33:             sequences.putIfAbsent(key, new AtomicPositiveInteger());
34:             sequence = sequences.get(key);
35:         }
36:         // 获得当前顺序号，并递增 + 1
37:         int currentSequence = sequence.getAndIncrement();
38:         // 权重不相等，顺序根据权重分配
39:         if (maxWeight > 0 && minWeight < maxWeight) {
40:             int mod = currentSequence % weightSum; // 剩余权重
41:             for (int i = 0; i < maxWeight; i++) { // 循环最大权重
42:                 for (Map.Entry<Invoker<T>, IntegerWrapper> each : invokerToWeightMap.entrySet()) { // 循环 Invoke
43:                     final Invoker<T> k = each.getKey();
44:                     final IntegerWrapper v = each.getValue();
45:                     // 剩余权重归 0 ，当前 Invoker 还有剩余权重，返回该 Invoker 对象
46:                     if (mod == 0 && v.getValue() > 0) {
47:                         return k;
48:                     }
49:                     // 若 Invoker 还有权重值，扣除它( value )和剩余权重( mod )。
50:                     if (v.getValue() > 0) {
51:                         v.decrement();
52:                         mod--;
53:                     }
54:                 }
55:             }
56:         }
57:         // 权重相等，平均顺序获得
58:         // Round robin
59:         return invokers.get(currentSequence % length);
60:     }
61:
62: }
```

第 18 行：invokerToWeightMap 变量，Invoker 与其权重的映射。其中，IntegerWrapper 为 RoundRobinLoadBalance 的内部类。代码如下：

```
private static final class IntegerWrapper {

    // 权重值
    private int value;

    // ... 省略 构造 / getting / setting 方法

    // 扣除一
    public void decrement() {
        this.value--;
    }

}
```

第 20 至 29 行：计算最小、最大权重，总的权重和，并初始化 invokerToWeightMap 。

- 其中，最小权重用来判断，所有 Invoker 的权重是否都相等。

第 30 至 35 行：获得对应的 AtomicPositiveInteger 对象，作为顺序计数器。
第 37 行：获得当前顺序号，并递增 + 1 。注意，递增要放后面，不然就不是从头开始了。
第 38 至 56 行：权重不相等，顺序根据权重分配。因为顺序分配的过程需要考虑权重，所以看起来比较“绕”。我们可以理解成：

- 顺序发 mod 次牌
- 每向一个 Invoker 发一次牌，它的剩余 weight 减一。当且仅当向有剩余 weight 的 Invoker 发牌。
- 当没有可发的 mod 牌时，选择该 Invoker 。

第 59 行：权重相等，直接平均顺序分配。相等于对【第 38 至 56 行】的优化。

FROM 飞哥的 《dubbo源码-负载均衡》

假定有3台权重都一样的dubbo provider:

```
10.0.0.1:20884, weight=100
10.0.0.1:20886, weight=100
10.0.0.1:20888, weight=100
```

轮询算法的实现：
其调用方法某个方法(key)的 sequence 从 0 开始：

```
sequence=0时，选择invokers.get(0%3)=10.0.0.1:20884
sequence=1时，选择invokers.get(1%3)=10.0.0.1:20886
sequence=2时，选择invokers.get(2%3)=10.0.0.1:20888
sequence=3时，选择invokers.get(3%3)=10.0.0.1:20884
sequence=4时，选择invokers.get(4%3)=10.0.0.1:20886
sequence=5时，选择invokers.get(5%3)=10.0.0.1:20888
```

如果有3台权重不一样的dubbo provider：

```
10.0.0.1:20884, weight=50
10.0.0.1:20886, weight=100
10.0.0.1:20888, weight=150
```

调试过很多次，这种情况下有问题；留一个TODO;

6. LeastActiveLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance ，实现 AbstractLoadBalance 抽象类，最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

相比来说，LeastActiveLoadBalance 是 RandomLoadBalance 的加强版，基于最少活跃调用数。

```
1: public class LeastActiveLoadBalance extends AbstractLoadBalance {
2:
3:     public static final String NAME = "leastactive";
4:
5:     private final Random random = new Random();
6:
```

```
 7:     @Override
 8:     protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
 9:         int length = invokers.size(); // 总个数
10:         int leastActive = -1; // 最小的活跃数
11:         int leastCount = 0; // 相同最小活跃数的个数
12:         int[] leastIndexes = new int[length]; // 相同最小活跃数的下标
13:         int totalWeight = 0; // 总权重
14:         int firstWeight = 0; // 第一个权重，用于于计算是否相同
15:         boolean sameWeight = true; // 是否所有权重相同
16:         // 计算获得相同最小活跃数的数组和个数
17:         for (int i = 0; i < length; i++) {
18:             Invoker<T> invoker = invokers.get(i);
19:             int active = RpcStatus.getStatus(invoker.getUrl(), invocation.getMethodName()).getActive(); // 活跃数
20:             int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.WEIGHT_KEY, Co
21:             if (leastActive == -1 || active < leastActive) { // 发现更小的活跃数，重新开始
22:                 leastActive = active; // 记录最小活跃数
23:                 leastCount = 1; // 重新统计相同最小活跃数的个数
24:                 leastIndexes[0] = i; // 重新记录最小活跃数下标
25:                 totalWeight = weight; // 重新累计总权重
26:                 firstWeight = weight; // 记录第一个权重
27:                 sameWeight = true; // 还原权重相同标识
28:             } else if (active == leastActive) { // 累计相同最小的活跃数
29:                 leastIndexes[leastCount++] = i; // 累计相同最小活跃数下标
30:                 totalWeight += weight; // 累计总权重
31:                 // 判断所有权重是否一样
32:                 if (sameWeight && weight != firstWeight) {
33:                     sameWeight = false;
34:                 }
35:             }
36:         }
37:         // assert(leastCount > 0)
38:         if (leastCount == 1) {
39:             // 如果只有一个最小则直接返回
40:             return invokers.get(leastIndexes[0]);
41:         }
42:         if (!sameWeight && totalWeight > 0) {
43:             // 如果权重不相同且权重大于0则按总权重数随机
44:             int offsetWeight = random.nextInt(totalWeight);
45:             // 并确定随机值落在哪个片断上
46:             for (int i = 0; i < leastCount; i++) {
47:                 int leastIndex = leastIndexes[i];
48:                 offsetWeight -= getWeight(invokers.get(leastIndex), invocation);
49:                 if (offsetWeight <= 0) {
50:                     return invokers.get(leastIndex);
51:                 }
52:             }
53:         }
54:         // 如果权重相同或权重为0则均等随机
55:         return invokers.get(leastIndexes[random.nextInt(leastCount)]);
56:     }
57:
58: }
```

第 16 至 36 行：计算获得相同最小活跃数的数组(leastIndexes)和个数(leastCount)。注意，leastIndexes 是重用的，所以需要 leastCount 作为下标。

- 每个 Invoker 的活跃数计算，通过 RpcStatus ，在 《精尽 Dubbo 源码分析 —— 过滤器（四）之 ActiveLimitFilter && ExecuteLimitFilter》 已经有详细解析。

第 38 行：如果只有一个最小则直接返回。

========== 如下部分，和 RandomLoadBalance 类似 ==========

第 42 至 53 行：权重不相等，随机权重后，判断在哪个 Invoker 的权重区间中。
第 55 行：权重相等，直接随机选择 Invoker 即可。相等于对【第 42 至 53 行】的优化。

算法说明

FROM 飞哥的 《dubbo源码-负载均衡》

最小活跃数算法实现：
假定有3台dubbo provider:

10.0.0.1:20884, weight=2，active=2
10.0.0.1:20886, weight=3，active=4
10.0.0.1:20888, weight=4，active=3

active=2最小，且只有一个2，所以选择10.0.0.1:20884

假定有3台dubbo provider:

10.0.0.1:20884, weight=2，active=2
10.0.0.1:20886, weight=3，active=2
10.0.0.1:20888, weight=4，active=3
active=2最小，且有2个，所以从[10.0.0.1:20884,10.0.0.1:20886]中选择；

接下来的算法与随机算法类似：

假设offset=1（即random.nextInt(5)=1）
1-2=-1<0？是，所以选中 10.0.0.1:20884, weight=2
假设offset=4（即random.nextInt(5)=4）
4-2=2<0？否，这时候offset=2， 2-3<0？是，所以选中 10.0.0.1:20886, weight=3

7. ConsistentHashLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.ConsistentHashLoadBalance ，实现 AbstractLoadBalance 抽象类，一致性 Hash，相同参数的请求总是发到同一提供者。

当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

```
1: public class ConsistentHashLoadBalance extends AbstractLoadBalance {
2:
3:     /**
4:      * 服务方法与一致性哈希选择器的映射
5:      *
6:      * KEY: serviceKey + "." + methodName
7:      */
8:     private final ConcurrentMap<String, ConsistentHashSelector<?>> selectors = new ConcurrentHashMap<String, Cons
9:
10:    @SuppressWarnings("unchecked")
11:    @Override
12:    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
13:        String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
14:        // 基于 invokers 集合，根据对象内存地址来计算定义哈希值
15:        int identityHashCode = System.identityHashCode(invokers);
```

```
16:         // 获得 ConsistentHashSelector 对象。若为空，或者定义哈希值变更（说明 invokers 集合发生变化），进行创建新
17:         ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key);
18:         if (selector == null || selector.identityHashCode != identityHashCode) {
19:             selectors.put(key, new ConsistentHashSelector<T>(invokers, invocation.getMethodName(), identityHashCo
20:             selector = (ConsistentHashSelector<T>) selectors.get(key);
21:         }
22:         return selector.select(invocation);
23:     }
24: }
```

第 15 行：调用 System#identityHashCode(Object) 方法，基于 invokers 集合，根据对象内存地址来计算定义哈希值。
第 16 至 21 行：获得 ConsistentHashSelector 对象。若为空，或者定义哈希值变更（说明 invokers 集合发生变化），进行创建新的 ConsistentHashSelector 对象。
第 22 行：调用 ConsistentHashSelector#select(invocation) 方法，选择一个 Invoker 对象。

7.1 ConsistentHashSelector

ConsistentHashSelector ，是 ConsistentHashLoadBalance 的内部类，一致性哈希选择器，基于 Ketama 算法。

老艿艿：下文参考 《Ketama一致性Hash算法(含Java代码)》 文章。从该文章中，我们可以看到，Spy Memcached Client 也采用这种算法。

7.1.1 构造方法

```
/**
 * 虚拟节点与 Invoker 的映射关系
 */
private final TreeMap<Long, Invoker<T>> virtualInvokers;
/**
 * 每个Invoker 对应的虚拟节点数
 */
private final int replicaNumber;
/**
 * 定义哈希值
 */
private final int identityHashCode;
/**
 * 取值参数位置数组
 */
private final int[] argumentIndex;
```

```
 1: ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, int identityHashCode) {
 2:     this.virtualInvokers = new TreeMap<Long, Invoker<T>>();
 3:     // 设置 identityHashCode
 4:     this.identityHashCode = identityHashCode;
 5:     URL url = invokers.get(0).getUrl();
 6:     // 初始化 replicaNumber
 7:     this.replicaNumber = url.getMethodParameter(methodName, "hash.nodes", 160);
 8:     // 初始化 argumentIndex
 9:     String[] index = Constants.COMMA_SPLIT_PATTERN.split(url.getMethodParameter(methodName, "hash.arguments", "0
10:     argumentIndex = new int[index.length];
11:     for (int i = 0; i < index.length; i++) {
12:         argumentIndex[i] = Integer.parseInt(index[i]);
```

```
13:     }
14:     // 初始化 virtualInvokers
15:     for (Invoker<T> invoker : invokers) {
16:         String address = invoker.getUrl().getAddress();
17:         // 每四个虚拟结点为一组，为什么这样？下面会说到
18:         for (int i = 0; i < replicaNumber / 4; i++) {
19:             // 这组虚拟结点得到惟一名称
20:             byte[] digest = md5(address + i);
21:             // Md5是一个16字节长度的数组，将16字节的数组每四个字节一组，分别对应一个虚拟结点，这就是为什么上面把
22:             for (int h = 0; h < 4; h++) {
23:                 // 对于每四个字节，组成一个long值数值，做为这个虚拟节点的在环中的惟一key
24:                 long m = hash(digest, h);
25:                 virtualInvokers.put(m, invoker);
26:             }
27:         }
28:     }
29: }
```

identityHashCode 字段，定义哈希值。

replicaNumber 字段，每个 Invoker 对应的虚拟节点数，默认为 160 。

- 可通过 <dubbo:parameter key="hash.nodes" value="320" /> 自定义，对应【第 7 行】代码。

argumentIndex 字段，选择 Invoker 时，计算 Hash 值的参数位置数组，默认为第一个参数。

- 可通过 <dubbo:parameter key="hash.arguments" value="0,1" /> 自定义， 对应【第 8 至 13 行】代码。

virtualInvokers 字段，虚拟节点与 Invoker 的映射关系。对应【第 14 至 28 行】进行初始化。

- 第 15 行：循环每个 Invoker 对象。
- 第 18 行：循环 replicaNumber / 4 次，每四个虚拟节点为一组，为什么这样呢？详细见【第 20 行】。
- 第 20 行：拼接 address + i 作为虚拟节点名的唯一名称。调用 #md5(value) 方法，计算 MD5 。代码如下：

```
private byte[] md5(String value) {
    MessageDigest md5;
    try {
        md5 = MessageDigest.getInstance("MD5");
    } catch (NoSuchAlgorithmException e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
    md5.reset();
    byte[] bytes;
    try {
        bytes = value.getBytes("UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
    md5.update(bytes);
    return md5.digest();
}
```

  - MD5 是一个 16 字节长度的数组，将 16 字节的数组每四个字节一组，分别对应一个虚拟结点，这就是为什么上面把虚拟结点四个划分一组的原因

- 第 22 行：顺序循环每四个字节。
- 第 24 行：调用 #hash(byte[] digest, int number) 方法，对于每四个字节，组成一个 Long 值数值，做为这个虚拟节点的在环中的惟一 KEY 。代码如下：

```
private long hash(byte[] digest, int number) {
    return (((long) (digest[3 + number * 4] & 0xFF) << 24)
            | ((long) (digest[2 + number * 4] & 0xFF) << 16)
            | ((long) (digest[1 + number * 4] & 0xFF) << 8)
            | (digest[number * 4] & 0xFF))
            & 0xFFFFFFFFL;
}
```

  - x
- 第 25 行：添加 Invoker 到 virtualInvokers 中。

7.1.2 select

```
public Invoker<T> select(Invocation invocation) {
    // 基于方法参数，获得 KEY
    String key = toKey(invocation.getArguments());
    // 计算 MD5 值
    byte[] digest = md5(key);
    // 计算 KEY 值
    return selectForKey(hash(digest, 0));
}
```

调用 #toKey(Object[] args) 方法，基于方法参数，获得 KEY 。代码如下：

```
private String toKey(Object[] args) {
    StringBuilder buf = new StringBuilder();
    for (int i : argumentIndex) {
        if (i >= 0 && i < args.length) {
            buf.append(args[i]);
        }
    }
    return buf.toString();
}
```

调用 #md5(key) 方法，计算 MD5 值。

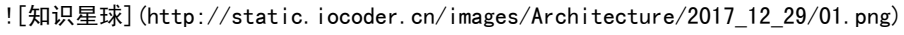
调用 #hash(digest, hash) 方法，计算 KEY 值。

调用 #selectForKey(hash) 方法，选一个 Invoker 对象。代码如下：

```
private Invoker<T> selectForKey(long hash) {
    // 得到大于当前 key 的那个子 Map ，然后从中取出第一个 key ，就是大于且离它最近的那个 key
    Map.Entry<Long, Invoker<T>> entry = virtualInvokers.tailMap(hash, true).firstEntry();
    // 不存在，则取 virtualInvokers 第一个
 if (entry == null) {
        entry = virtualInvokers.firstEntry();
    }
```

```
                // 存在，则返回
                return entry.getValue();
            }
```

666. 彩蛋



小文一篇，美滋滋！

文章目录