crazyhzm 265

Dubbo源码解析（三十八）集群——LoadBalance

dubbo java 阅读约 45 分钟

集群——LoadBalance

> 目标：介绍dubbo中集群的负载均衡，介绍dubbo-cluster下loadBalance包的源码。

前言

负载均衡，说的通俗点就是要一碗水端平。在这个时代，公平是很重要的，在网络请求的时候同样是这个道理，我们有很多机器，但是请求老是到某个服务器上，而某些服务器又常年空闲，导致了资源的浪费，也增加了服务器因为压力过载而宕机的风险。这个时候就需要负载均衡的出现。它就相当于是一个天秤，通过各种策略，可以让每台服务器获取到适合自己处理能力的负载，这样既能够为高负载的服务器分流，还能避免资源浪费。负载均衡分为软件的负载均衡和硬件负载均衡，我们这里讲到的是软件负载均衡，在dubbo中，需要对消费者的调用请求进行分配，避免少数服务提供者负载过大，其他服务空闲的情况，因为负载过大会导致服务请求超时。这个时候就需要负载均衡起作用了。Dubbo 提供了4种负载均衡实现：

1. RandomLoadBalance：基于权重随机算法
2. LeastActiveLoadBalance：基于最少活跃调用数算法
3. ConsistentHashLoadBalance：基于 hash 一致性
4. RoundRobinLoadBalance：基于加权轮询算法

具体的实现看下面解析。

源码分析

（一）AbstractLoadBalance

该类实现了LoadBalance接口，是负载均衡的抽象类，提供了权重计算的功能。

1.select

```
@Override
public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    // 如果invokers为空则返回空
    if (invokers == null || invokers.isEmpty())
        return null;
    // 如果invokers只有一个服务提供者，则返回一个
    if (invokers.size() == 1)
        return invokers.get(0);
    // 调用doSelect进行选择
    return doSelect(invokers, url, invocation);
}
```

该方法是选择一个invoker，关键的选择还是调用了doSelect方法，不过doSelect是一个抽象方法，由上述四种负载均衡策略来各自实现。

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.DEFAULT_WEIGHT);
    if (weight > 0) {
        // 获得启动时间戳
        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分钟
            int warmup = invoker.getUrl().getParameter(Constants.WARMUP_KEY, Constants.DEFAULT_WARMUP);
            // 如果服务运行时间小于预热时间，则重新计算服务权重，即降权
            if (uptime > 0 && uptime < warmup) {
                weight = calculateWarmupWeight(uptime, warmup, weight);
            }
        }
    }
    return weight;
}
```

该方法是获得权重的方法，计算权重在calculateWarmupWeight方法中实现，该方法考虑到了jvm预热的过程。

3.calculateWarmupWeight

```
static int calculateWarmupWeight(int uptime, int warmup, int weight) {
    // 计算权重 (uptime / warmup) * weight，进度百分比 * 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);
}
```

该方法是计算权重的方法，其中计算公式是(uptime / warmup) *weight，含义就是进度百分比* 权重值。

（二）RandomLoadBalance

该类是基于权重随机算法的负载均衡实现类，我们先来讲讲原理，比如我有有一组服务器 servers = [A, B, C]，他们他们对应的权重为 weights = [6, 3, 1]，权重总和为10，现在把这些权重值平铺在一维坐标值上，分别出现三个区域，A区域为[0,6)，B区域为[6,9)，C区域为[9,10)，然后产生一个[0, 10)的随机数，看该数字落在哪个区间内，就用哪台服务器，这样权重越大的，被击中的概率就越大。

```
public class RandomLoadBalance extends AbstractLoadBalance {

    public static final String NAME = "random";

    /**
     * 随机数产生器
     */
    private final Random random = new Random();

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        // 获得服务长度
        int length = invokers.size(); // Number of invokers
        // 总的权重
        int totalWeight = 0; // The sum of weights
        // 是否有相同的权重
        boolean sameWeight = true; // Every invoker has the same weight?
        // 遍历每个服务，计算相应权重
        for (int i = 0; i < length; i++) {
            int weight = getWeight(invokers.get(i), invocation);
            // 计算总的权重值
            totalWeight += weight; // Sum
            // 如果前一个服务的权重值不等于后一个则sameWeight为false
            if (sameWeight && i > 0
                    && weight != getWeight(invokers.get(i - 1), invocation)) {
                sameWeight = false;
```

该算法比较好理解，当然 RandomLoadBalance 也存在一定的缺点，当调用次数比较少时，Random 产生的随机数可能会比较集中，此时多数请求会落到同一台服务器上，不过影响不大。

（三）LeastActiveLoadBalance

该负载均衡策略基于最少活跃调用数算法，某个服务活跃调用数越小，表明该服务提供者效率越高，也就表明单位时间内能够处理的请求更多。此时应该选择该类服务器。实现很简单，就是每一个服务都有一个活跃数active来记录该服务的活跃值，每收到一个请求，该active就会加1，，没完成一个请求，active就会减1。在服务运行一段时间后，性能好的服务提供者处理请求的速度更快，因此活跃数下降的也越快，此时这样的服务提供者能够优先获取到新的服务请求。除了最小活跃数，还引入了权重值，也就是当活跃数一样的时候，选择利用权重法来进行选择，如果权重也一样，那么随机选择一个。

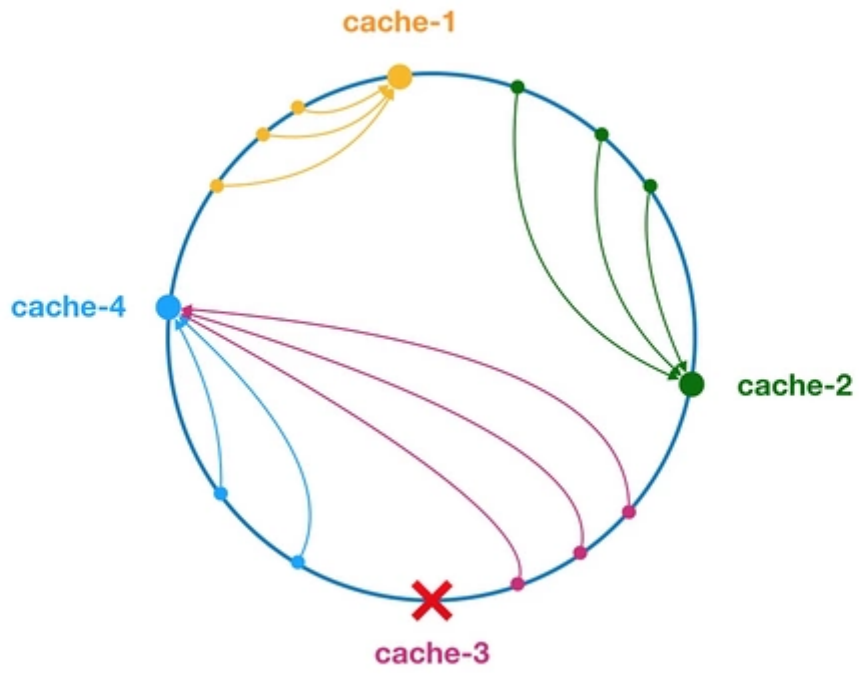
```
                    && afterWarmup != firstWeight) {
                sameWeight = false;
            }
        }
    }
    // assert(leastCount > 0)
    // 当只有一个 Invoker 具有最小活跃数，此时直接返回该 Invoker 即可
    if (leastCount == 1) {
        // If we got exactly one invoker having the least active value, return this invoker directly.
        return invokers.get(leastIndexs[0]);
    }
    // 有多个 Invoker 具有相同的最小活跃数，但它们之间的权重不同
    if (!sameWeight && totalWeight > 0) {
        // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly
based on totalWeight.
        // 随机生成一个数字
        int offsetWeight = random.nextInt(totalWeight) + 1;
        // Return a invoker based on the random value.
        // 相关算法可以参考RandomLoadBalance
        for (int i = 0; i < leastCount; i++) {
            int leastIndex = leastIndexs[i];
            offsetWeight -= getWeight(invokers.get(leastIndex), invocation);
            if (offsetWeight <= 0)
                return invokers.get(leastIndex);
        }
    }
    // If all invokers have the same weight value or totalWeight=0, return evenly.
```

前半部分在进行最小活跃数的策略，后半部分在进行权重的随机策略，可以参见RandomLoadBalance。

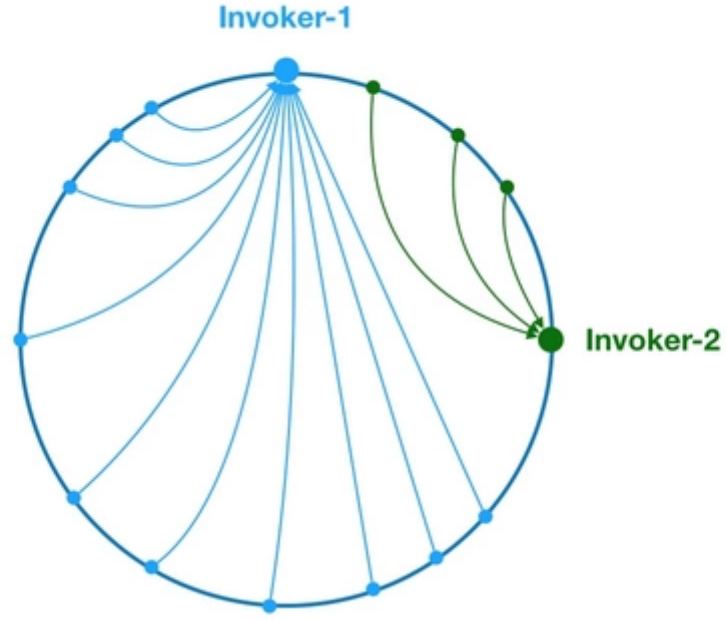
（四）ConsistentHashLoadBalance

该类是负载均衡基于 hash 一致性的逻辑实现。一致性哈希算法由麻省理工学院的 Karger 及其合作者于1997年提供出的，一开始被大量运用于缓存系统的负载均衡。它的工作原理是这样的：首先根据 ip 或其他的信息为缓存节点生成一个 hash，在dubbo中使用参数进行计算hash。并将这个 hash 投射到 [0, 232 - 1] 的圆环上，当有查询或写入请求时，则生成一个 hash 值。然后查找第一个大于或等于该 hash 值的缓存节点，并到这个节点中查询或写入缓存项。如果当前节点挂了，则在下一次查询或写入缓存时，为缓存项查找另一个大于其 hash 值的缓存节点即可。大致效果如下图所示（引用一下官网的图）

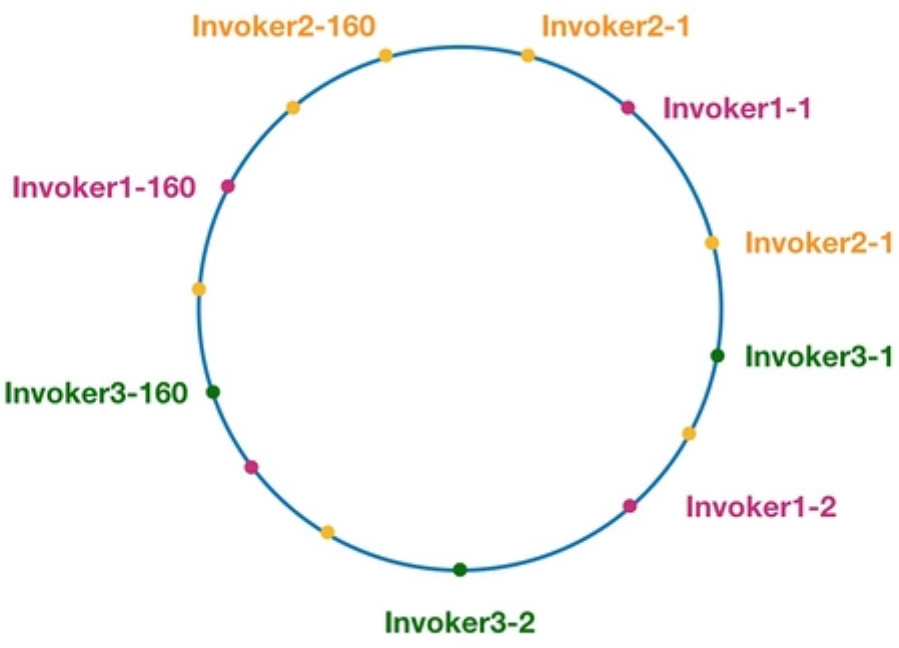
每个缓存节点在圆环上占据一个位置。如果缓存项的 key 的 hash 值小于缓存节点 hash 值，则到该缓存节点中存储或读取缓存项，这里有两个概念不要弄混，缓存节点就好比dubbo中的服务提供者，会有很多的服务提供者，而缓存项就好比是服务引用的消费者。比如下面绿色点对应的缓存项也就是服务消费者将会被存储到 cache-2 节点中。由于 cache-3 挂了，原本应该存到该节点中的缓存项也就是服务消费者最终会存储到 cache-4 节点中，也就是调用cache-4 这个服务提供者。



但是在hash一致性算法并不能够保证hash算法的平衡性，就拿上面的例子来看，cache-3挂掉了，那该节点下的所有缓存项都要存储到 cache-4 节点中，这就导致hash值低的一直往高的存储，会面临一个不平衡的现象，见下图：



可以看到最后会变成类似不平衡的现象，那我们应该怎么避免这样的事情，做到平衡性，那就需要引入虚拟节点，虚拟节点是实际节点在 hash 空间的复制品，“虚拟节点”在 hash 空间中以hash值排列。比如下图：



可以看到各个节点都被均匀分布在圆环上，而某一个服务提供者居然有多个节点存在，分别跟其他节点交错排列，这样做的目的就是避免数据倾斜问题，也就是由于节点不够分散，导致大量请求落到了同一个节点上，而其他节点只会接收到了少量请求的情况。类似第二张图的情况。

看完原理，接下来我们来看看代码

1.doSelect

```
protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    // 获得方法名
    String methodName = RpcUtils.getMethodName(invocation);
    // 获得key
    String key = invokers.get(0).getUrl().getServiceKey() + "." + methodName;
    // 获取 invokers 原始的 hashcode
    int identityHashCode = System.identityHashCode(invokers);
    // 从一致性 hash 选择器集合中获得一致性 hash 选择器
    ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key);
    // 如果等于空或者选择器的hash值不等于原始的值，则新建一个一致性 hash 选择器，并且加入到集合
    if (selector == null || selector.identityHashCode != identityHashCode) {
        selectors.put(key, new ConsistentHashSelector<T>(invokers, methodName, identityHashCode));
        selector = (ConsistentHashSelector<T>) selectors.get(key);
    }
    // 选择器选择一个invoker
    return selector.select(invocation);
}
```

该方法也做了一些invokers 列表是不是变动过，以及创建 ConsistentHashSelector等工作，然后调用selector.select来进行选择。

2.ConsistentHashSelector

```
private static final class ConsistentHashSelector<T> {

    /**
     * 存储 Invoker 虚拟节点
     */
    private final TreeMap<Long, Invoker<T>> virtualInvokers;

    /**
     * 每个Invoker 对应的虚拟节点数
     */
    private final int replicaNumber;

    /**
     * 原始哈希值
     */
    private final int identityHashCode;

    /**
     * 取值参数位置数组
     */
    private final int[] argumentIndex;

    ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, int identityHashCode) {
        this.virtualInvokers = new TreeMap<Long, Invoker<T>>();
        this.identityHashCode = identityHashCode;
        URL url = invokers.get(0).getUrl();
```

该类是内部类，是一致性 hash 选择器，首先看它的属性，利用TreeMap来存储 Invoker 虚拟节点，因为需要提供高效的查询操作。再看看它的构造方法，执行了一系列的初始化逻辑，比如从配置中获取虚拟节点数以及参与 hash 计算的参数下标，默认情况下只使用第一个参数进行 hash，并且ConsistentHashLoadBalance 的负载均衡逻辑只受参数值影响，具有相同参数值的请求将会被分配给同一个服务提供者。还有一个select方法，比较简单，先进行md5运算。然后hash，最后选择出对应的invoker。

(五)RoundRobinLoadBalance

该类是负载均衡基于加权轮询算法的实现。那么什么是加权轮询，轮询很好理解，比如我第一个请求分配给A服务器，第二个请求分配给B服务器，第三个请求分配给C服务器，第四个请求又分配给A服务器，这就是轮询，但是这只适合每台服务器性能相近的情况，这种是一种非常理想的情况，那更多的是每台服务器的性能都会有所差异，这个时候性能差的服务器被分到等额的请求，就会需要承受压力大宕机的情况，这个时候我们需要对轮询加权，我举个例子，服务器 A、B、C 权重比为 6:3:1，那么在10次请求中，服务器 A 将收到其中的6次请求，服务器 B 会收到其中的3次请求，服务器 C 则收到其中的1次请求，也就是说每台服务器能够收到的请求归结于它的权重。

1.属性

```
/**
 * 回收间隔
 */
private static int RECYCLE_PERIOD = 60000;
```

2.WeightedRoundRobin

```
protected static class WeightedRoundRobin {
    /**
     * 权重
     */
    private int weight;
    /**
     * 当前已经有多少请求落在该服务提供者身上，也可以看成是一个动态的权重
     */
    private AtomicLong current = new AtomicLong(0);
    /**
     * 最后一次更新时间
     */
    private long lastUpdate;
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
        current.set(0);
    }
    public long increaseCurrent() {
        return current.addAndGet(weight);
    }
    public void sel(int total) {
        current.addAndGet(-1 * total);
    }
```

该内部类是一个加权轮询器，它记录了某一个服务提供者的一些数据，比如权重、比如当前已经有多少请求落在该服务提供者上等。

3.doSelect

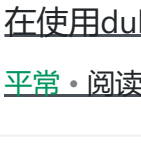
```
@Override
protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
    // key = 全限定类名 + "." + 方法名，比如 com.xxx.DemoService.sayHello
    String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
    ConcurrentMap<String, WeightedRoundRobin> map = methodWeightMap.get(key);
    if (map == null) {
        methodWeightMap.putIfAbsent(key, new ConcurrentHashMap<String, WeightedRoundRobin>());
        map = methodWeightMap.get(key);
    }
    // 权重总和
    int totalWeight = 0;
    // 最小权重
    long maxCurrent = Long.MIN_VALUE;
    // 获得现在的时间戳
    long now = System.currentTimeMillis();
    // 创建已经选择的invoker
    Invoker<T> selectedInvoker = null;
    // 创建加权轮询器
    WeightedRoundRobin selectedWRR = null;

    // 下面这个循环主要做了这样几件事情：
    //   1. 遍历 Invoker 列表，检测当前 Invoker 是否有
    //      相应的 WeightedRoundRobin，没有则创建
    //   2. 检测 Invoker 权重是否发生了变化，若变化了，
    //      则更新 WeightedRoundRobin 的 weight 字段
    //   3. 让 current 字段加上自身权重，等价于 current += weight
```

该方法是选择的核心，其实关键是一些数据记录，在每次请求都会记录落在该服务上的请求数，然后在根据权重来分配，并且会有回收时间来处理一些长时间未被更新的节点。

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负载均衡实现的部分，每个算法都是现在很普遍的负载均衡算法，希望大家细细品味。接下来我将开始对集群模块关于分组聚合部分进行讲解。



crazyhzm

265

关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

dubbo源码解析（二）Dubbo扩展机制SPI

前一篇文章《dubbo源码解析（一）Hello,Dubbo》是对dubbo整个项目大体的介绍，而从这篇文章开始，我将会从源码来解读dub...

CrazyHzm • 阅读 355

dubbo注册服务IP解析异常及IP解析源码分析

在使用dubbo注册服务时会遇到IP解析错误导致无法正常访问.比如:本机设置的IP为172.16.11.111,但实际解析出来的是180.20.174.1...

平常 • 阅读 37

Dubbo 源码分析 - 自适应拓展原理

我在上一篇文章中分析了Dubbo的SPI机制，DubboSPI是Dubbo框架的核心。Dubbo中的很多拓展都是通过SPI机制进行加载的，比...

coolblog • 阅读 329

dubbo源码解析——概要篇

这次源码解析借鉴《肥朝》前辈的dubbo源码解析，进行源码学习。总结起来就是先总体,后局部.也就是先把需要注意的概念先抛...

• 阅读 631

dubbo源码解析（一）Hello,Dubbo

你好，dubbo，初次见面，我想和你交个朋友。先给出一套官方的说法：ApacheDubbo是一款高性能、轻量级基于Java的RPC开源...

CrazyHzm • 阅读 633

Retrofit源码分析三 源码分析

我们先来看一下Retrofit的常见使用方法：上面是Retrofit的最基本使用方法，当然现在使用最多的还是RxJava2+Retrofit搭配使用，...

BlackFlagBin • 阅读 9

dubbo源码解析（十五）远程通信——Mina