链滴

Motan_LoadBalance

作者：Lord-X
原文链接：https://ld246.com/article/1588075005816
来源网站：链滴

Motan系列文章

- Motan如何完成与Spring的集成
- Motan的SPI插件扩展机制
- Motan服务注册
- Motan服务调用
- Motan心跳机制
- Motan负载均衡策略
- Motan高可用策略

LoadBalance即负载均衡策略（下面简称LB），Motan为LB提供了多种方案，默认为 ActiveWeight，并支持自定义扩展（通过SPI机制），Motan是在Client端做的负载均衡。目前支持以下几种：

- ActiveWeight(默认)

低并发度优先： referer 的某时刻的 call 数越小优先级越高。

配置方式：

```
<motan:protocol ... loadbalance="activeWeight"/>
```

- Random

随机选择。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，有利于动态调整提供者权重。

配置方式：

```
<motan:protocol ... loadbalance="random"/>
```

- RoundRobin

轮循选择，调用比较均匀。

配置方式：

```
<motan:protocol ... loadbalance="roundrobin"/>
```

- LocalFirst

本地服务优先获取策略，对referers根据ip顺序查找本地服务，多存在多个本地服务，获取Active最的本地服务进行服务。

当不存在本地服务，但是存在远程RPC服务，则根据ActivWeight获取远程RPC服务。

当两者都存在，所有本地服务都应优先于远程服务，本地RPC服务与远程RPC服务内部则根据Active eight进行。

配置方式：

```
<motan:protocol ... loadbalance="localFirst"/>
```

- Consistent

一致性 Hash，相同参数的请求总是发到同一提供者。

配置方式：

```
<motan:protocol ... loadbalance="consistent"/>
```

- ConfigurableWeight

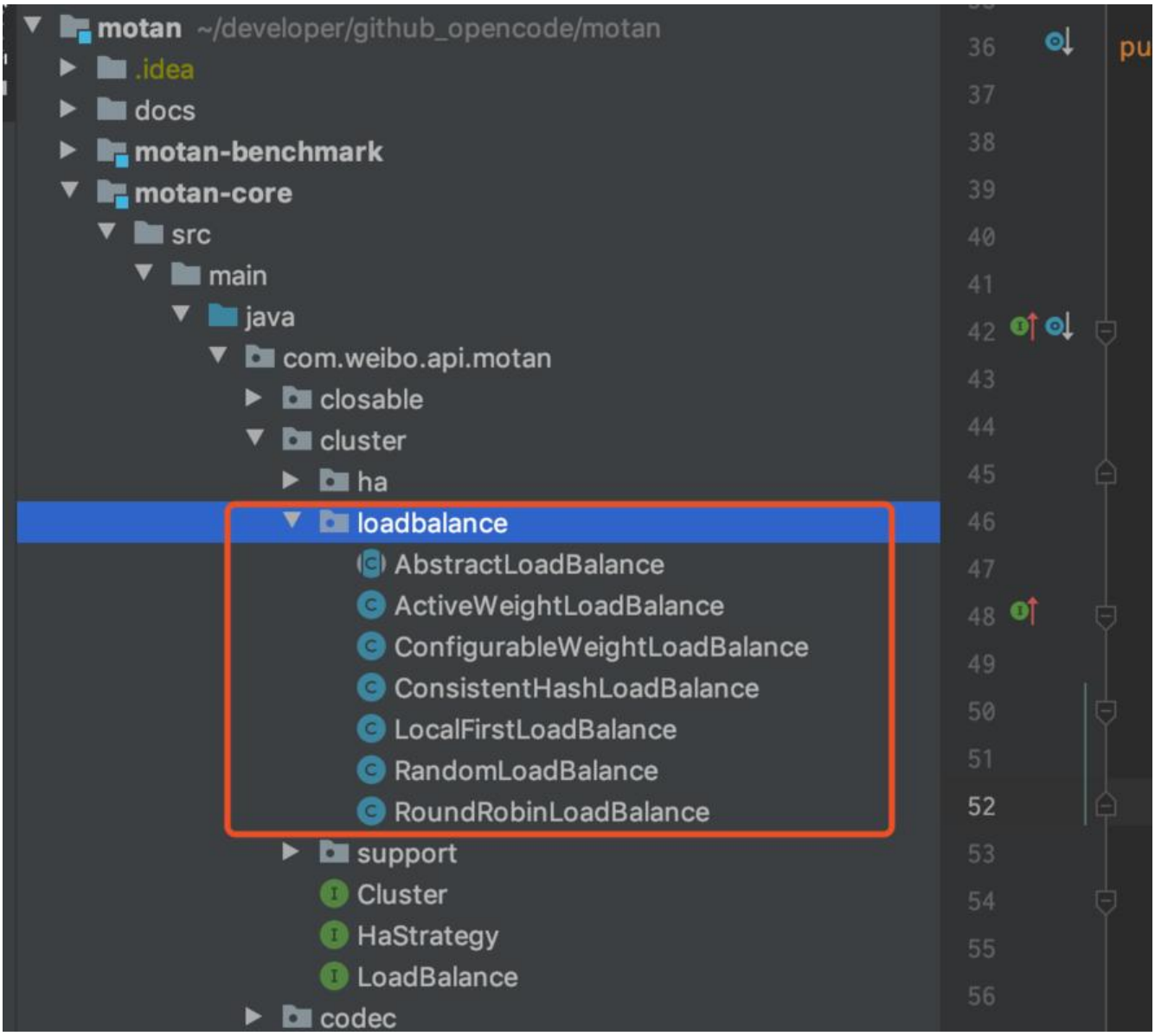
权重可配置的负载均衡策略。

配置方式：

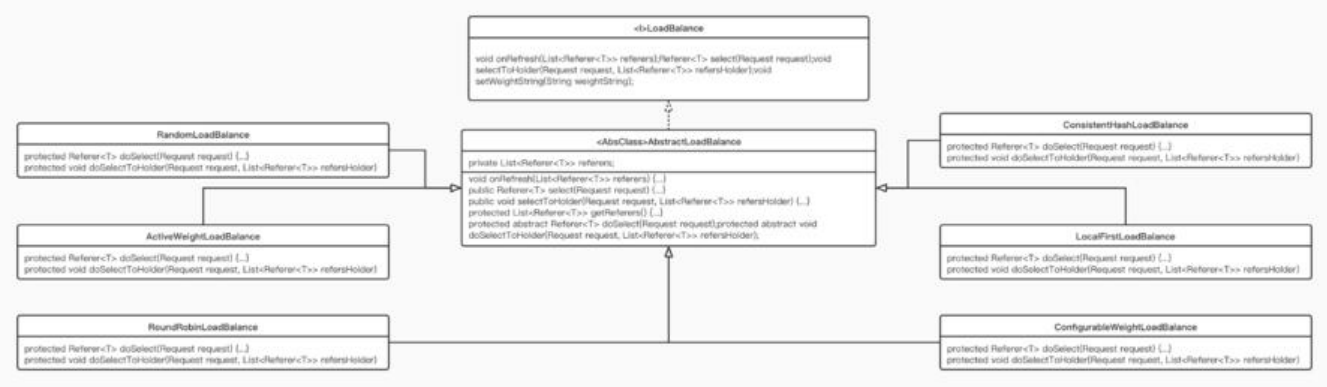
```
<motan:protocol ... loadbalance="configurableWeight"/>
```

0 工程结构及继承关系

下图展示了LB源码在工程中的位置。



下面来看下LB的体系结构。



LoadBalance作为顶层接口，定义了LB的主要功能。最主要的两个方法为 select() 和 selectToHolder() 用于从Cluster中根据规则选择一个Referer调用。

AbstractLoadBalance实现了LoadBalance接口，并提供了 select() 和 selectToHolder() 方法的通实现，同时又定义了 doSelect() 和 doSelectToHolder() 两个抽象方法，select() 和 selectToHolder() 别调用了这两个方法，这里是一个模板设计模式的实现，doSelect() 和 doSelectToHolder()会交给

原文链接：Motan_LoadBalance

体的实现类实现，定义自己的Referer选取逻辑。

最后，6个具体实现类extends了AbstractLoadBalance，并实现 doSelect() 和 doSelectToHolder() 完成具体的Referer选取。

PS：doSelect() 和 doSelectToHolder() 都是选取Referer的逻辑，只是调用的地方不同。doSelect() 由 FailfastHaStrategy 调用，doSelectToHolder() 由 FailoverHaStrategy 调用。也就是说，这两方法分别对应到两个不同的HA具体实现上。对于默认HA策略的 FailoverHaStrategy 来说，需要实失效转移功能，所以这里可以暂时理解为，doSelectToHolder 用于支持失效转移，而 doSelect 用支持快速失败。

1 LoadBalance策略的选取及设置

Motan会在Cluster初始化阶段设置LB策略。LB策略可通过上述配置来定义，如果不配置，默认使用 A tiveWeight，即低并发度优先策略。

在实现上，LoadBalance采用插件化开发（即SPI），每个LoadBalance的具体实现都是一个SPI，其现类都标注了 @SpiMeta 注解，在setLoadBalance时，根据具体的LB名称即可找到具体的实现。以 ctiveWeight 为例：

```
@SpiMeta(name = "activeWeight")
public class ActiveWeightLoadBalance<T> extends AbstractLoadBalance<T> {

}
```

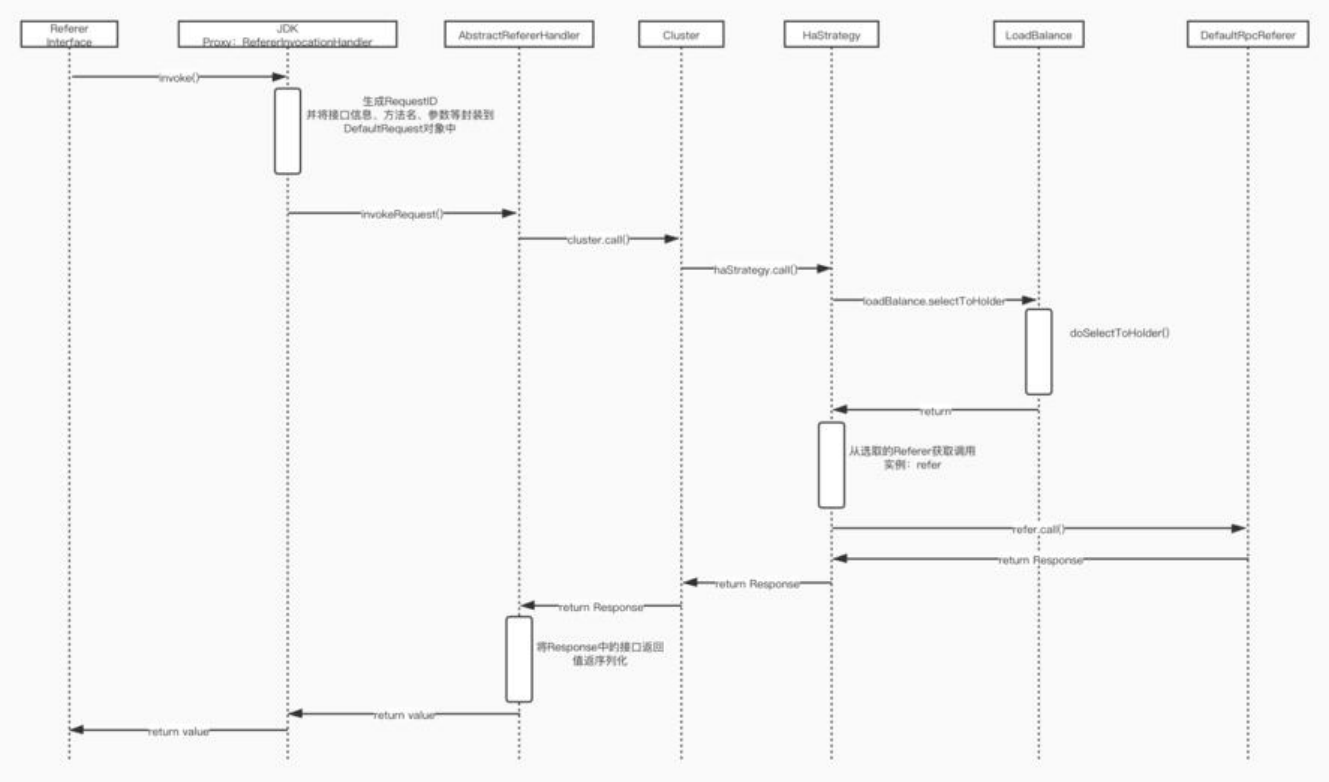
然后在 ClusterSupport 中通过SPI获取LoadBalance的具体实现，并setLoadBalance。

```
private void prepareCluster() {
    String clusterName = url.getParameter(URLParamType.cluster.getName(), URLParamType.c
uster.getValue());
    // 获取LoadBalance策略，优选获取用户配置的，如果没有配置，取URLParamType配置的默认
，即 activeWeight
    String loadbalanceName = url.getParameter(URLParamType.loadbalance.getName(), URLPa
amType.loadbalance.getValue());
    String haStrategyName = url.getParameter(URLParamType.haStrategy.getName(), URLPar
mType.haStrategy.getValue());

    cluster = ExtensionLoader.getExtensionLoader(Cluster.class).getExtension(clusterName);
    // 通过SPI的方式获取具体LoadBalance实现
    LoadBalance<T> loadBalance = ExtensionLoader.getExtensionLoader(LoadBalance.class).ge
Extension(loadbalanceName);
    HaStrategy<T> ha = ExtensionLoader.getExtensionLoader(HaStrategy.class).getExtension(h
StrategyName);
    ha.setUrl(url);
    cluster.setLoadBalance(loadBalance); // setLoadBalance
    cluster.setHaStrategy(ha);
    cluster.setUrl(url);
}
```

2 调用时序

配置好LB以后，再来看一下调用时序。以下时序图说明了一个RPC请求的调用时序，主要关注HaStrat gy到LoadBalance的调用。



至此，本文已说明LB的设置以及LB的调用时机，下面介绍几个具体的LB实现。

3 LoadBalance的实现

下面介绍三种LoadBalance的具体实现。

- ActiveWeightLoadBalance：低并发度优先策略
- RandomLoadBalance：随机策略
- RoundRobinLoadBalance：轮询策略

3.1 ActiveWeightLoadBalance

ActiveWeightLoadBalance是默认的LB策略，其Referer的选取策略是低并发度优先，即某一时刻，取正在处理请求数最少的那个Referer。

先来看ActiveWeightLoadBalance的doSelectToHolder方法，他的作用是在集群中选出最多10个可的节点，并将这些节点按照并发数升序排序。这样一来，第一个就是并发度最低的那个。

PS：为啥要选出多个可用节点，而不是选出一个就行？

因为LB的调用上游是HA，默认情况下，HA策略是失效转移，即如果一个节点不可用了，要再次对其的节点尝试调用。所以这里要选出多个，用于满足HA的失效转移策略。

```
protected void doSelectToHolder(Request request, List<Referer<T>> refersHolder) {
    // 获取集群的所有节点
    List<Referer<T>> referers = getReferers();

    int refererSize = referers.size();
    // 随机一个起始索引出来（不从0开始索引），目的是，当集群规模较大时，保证所有节点都有被取的机会
```

```
    int startIndex = ThreadLocalRandom.current().nextInt(refererSize);
    int currentCursor = 0;
    int currentAvailableCursor = 0;
    // MAX_REFERER_COUNT的值是10，意味着最多选出10个有效节点作为备选调用
    while (currentAvailableCursor < MAX_REFERER_COUNT && currentCursor < refererSize) {
        Referer<T> temp = referers.get((startIndex + currentCursor) % refererSize);
        currentCursor++;

        if (!temp.isAvailable()) {
            continue;
        }

        currentAvailableCursor++;

        refersHolder.add(temp);
    }
    // 按并发数升序排序
    Collections.sort(refersHolder, new LowActivePriorityComparator<T>());
}
```

这里最大规模集群做了一个优化，例如集群有100个节点，并且都是可用状态，那么，startIndex是10以内的一个随机值，例如95，由于最多选取10个，所以最终选择的节点是：[
5, 96, 97, 98, 99, 0, 1, 2, 3, 4]。

如果startIndex每次都从0开始，后面的节点就没有机会入选了。

最后将选取结果按并发度升序排序，得到最终结果。

并发度如何计算?

由上述代码可知，排序依据是 LowActivePriorityComparator 这个比较器。来看下这个：

```
static class LowActivePriorityComparator<T> implements Comparator<Referer<T>> {
    @Override
    public int compare(Referer<T> referer1, Referer<T> referer2) {
        return referer1.activeRefererCount() - referer2.activeRefererCount();
    }
}
```

这里涉及到了Referer的 activeRefererCount，这个字段的定义在 AbstractReferer 中，参考如下源：

```
protected AtomicInteger activeRefererCount = new AtomicInteger(0);

public Response call(Request request) {
    if (!isAvailable()) {
        throw new MotanFrameworkException(this.getClass().getSimpleName() + " call Error: no
e is not available, url=" + url.getUri()
                + " " + MotanFrameworkUtil.toString(request));
    }
    // activeRefererCount + 1
    incrActiveCount(request);
    Response response = null;
    try {
        response = doCall(request);
```

```
        return response;
    } finally {
        // activeRefererCount - 1
        decrActiveCount(request, response);
    }
}
```

可以得出结论，这个值，其实是在某一Referer的RPC调用前+1，调用结束后-1，这样就得到了某一Reerer的并发数。

3.2 RandomLoadBalance

RandomLoadBalance，即从集群中随机选取一个，如果选取的是不可用的节点，就继续随机，直到到可用的为止。

```
protected void doSelectToHolder(Request request, List<Referer<T>> refersHolder) {
    List<Referer<T>> referers = getReferers();

    int idx = (int) (ThreadLocalRandom.current().nextDouble() * referers.size());
    for (int i = 0; i < referers.size(); i++) {
        Referer<T> referer = referers.get((i + idx) % referers.size());
        if (referer.isAvailable()) {
            refersHolder.add(referer);
        }
    }
}
```

3.3 RoundRobinLoadBalance

即轮询策略。在大规模集群下，仍然是最多选取10个可用的节点。

```
private AtomicInteger idx = new AtomicInteger(0);

protected void doSelectToHolder(Request request, List<Referer<T>> refersHolder) {
    List<Referer<T>> referers = getReferers();

    int index = getNextNonNegative();
    for (int i = 0, count = 0; i < referers.size() && count < MAX_REFERER_COUNT; i++) {
        Referer<T> referer = referers.get((i + index) % referers.size());
        if (referer.isAvailable()) {
            refersHolder.add(referer);
            count++;
        }
    }
}

// get non-negative int
private int getNextNonNegative() {
    return MathUtil.getNonNegative(idx.incrementAndGet());
}
```

idx 是一个共享实例变量，每次 doSelectToHolder 递增1，然后与referer数取模，以达到轮询效果

同样的，为了满足HA的失败转移策略，这里会选取多个可用referer，最多取10个。

参考

- Motan_Github