链滴

ribbon 的几种负载均衡

作者：wenyl

原文链接：https://ld246.com/article/1638409501245

来源网站：链滴

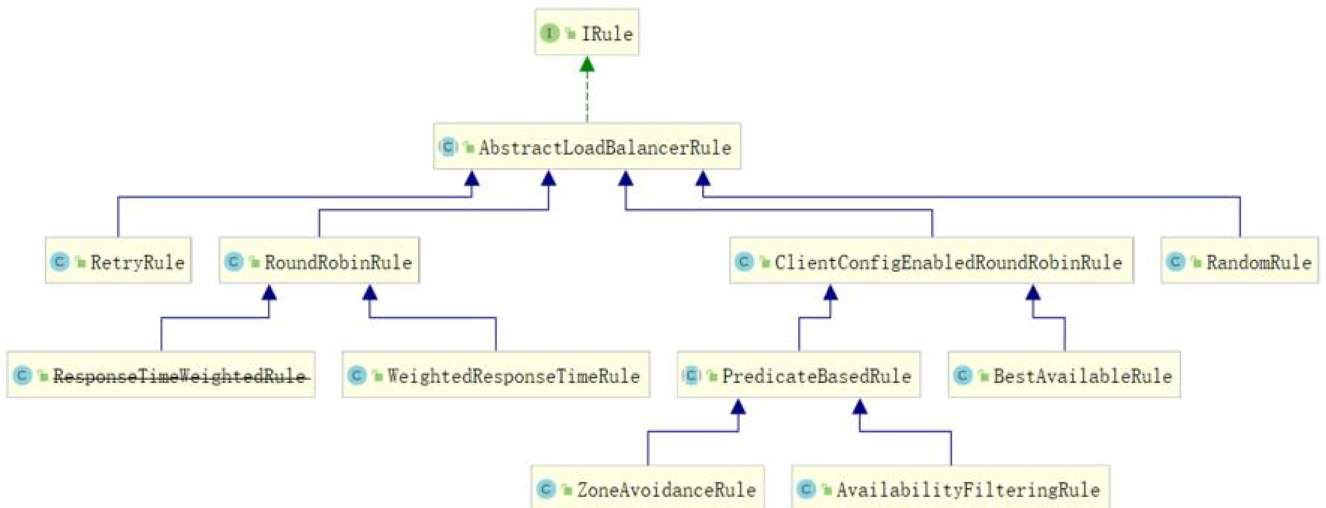
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

1、简介

spring cloud的负载均衡由ribbon组件实现，ribbon是NetFlix发布的客户端负载均衡器。

2、ribbon负载均衡规则

ribbon负载均衡策略的结构类图



IRule接口中共有三个方法

```java
public interface IRule{
    /*
     * choose one alive server from lb.allServers or
     * lb.upServers according to key
     *
     * @return choosen Server object. NULL is returned if none
     *  server is available
     */

    public Server choose(Object key);

    public void setLoadBalancer(ILoadBalancer lb);

    public ILoadBalancer getLoadBalancer();
}
```

抽象类AbstractLoadBalancerRule实现了IRule接口，定义了一个ILoadBalancer类型变量，用于ILoadBalancer接口定义了软负载均衡的操作方法。

```java
public abstract class AbstractLoadBalancerRule implements IRule, IClientConfigAware {

    private ILoadBalancer lb;

    @Override
    public void setLoadBalancer(ILoadBalancer lb){
        this.lb = lb;
    }

    @Override
    public ILoadBalancer getLoadBalancer(){
        return lb;
    }
}
```

3、负载均衡策略

3.1、轮询策略

轮询策略在RoundRobinRule类中实现，它将可用服务存储在一个List中，然后定义了一个原子操作，每次调用就+1，下次调用就把这个值作为List的下标，以此决定要调用哪个服务。

下面是定义，在RoundRobinRule构造函数中，会初始化值为0

```java
private AtomicInteger nextServerCyclicCounter;
```

```java
public Server choose(ILoadBalancer lb, Object key) {
    if (lb == null) {
        log.warn("no load balancer");
        return null;
    }

    Server server = null;
    int count = 0;
    while (server == null && count++ < 10) {
        List<Server> reachableServers = lb.getReachableServers();
        List<Server> allServers = lb.getAllServers();
        int upCount = reachableServers.size();
        int serverCount = allServers.size();

        if ((upCount == 0) || (serverCount == 0)) {
            log.warn("No up servers available from load balancer: " + lb);
            return null;
        }

        int nextServerIndex = incrementAndGetModulo(serverCount);
        server = allServers.get(nextServerIndex);

        if (server == null) {
            /* Transient. */
            Thread.yield();
            continue;
        }

        if (server.isAlive() && (server.isReadyToServe())) {
            return (server);
        }
```

3.2、随机策略

随机策略在RandomRule实现，RandomRule定义规则，从现有服务中，随机选择一个服务，具体做就是根据可选服务数量，选出一个随机数作为下标，获取服务

```java
Random rand;
```

```java
while (server == null) {
    if (Thread.interrupted()) {
        return null;
    }
    List<Server> upList = lb.getReachableServers();
    List<Server> allList = lb.getAllServers();

    int serverCount = allList.size();
    if (serverCount == 0) {
        /*
         * No servers. End regardless of pass, because subsequent passes
         * only get more restrictive.
         */
        return null;
    }

    int index = rand.nextInt(serverCount);
    server = upList.get(index);
```

3.3、根据响应时间分配权重的策略

这个策略由WeightedResponseTimeRule实现，使用这个策略的话，相应时间越长的服务，越不容被选中，响应时间越短的被选中的概率越大。

WeightedResponseTimeTule定义了一个定时任务30S执行一次

```java
try {
    logger.info("Weight adjusting job started");
    AbstractLoadBalancer nlb = (AbstractLoadBalancer) lb;
    LoadBalancerStats stats = nlb.getLoadBalancerStats();
    if (stats == null) {
        // no statistics, nothing to do
        return;
    }
    double totalResponseTime = 0;
    // find maximal 95% response time
    for (Server server : nlb.getAllServers()) {
        // this will automatically load the stats if not in cache
        ServerStats ss = stats.getSingleServerStat(server);
        totalResponseTime += ss.getResponseTimeAvg();
    }
    // weight for each server is (sum of responseTime of all servers - responseTime)
    // so that the longer the response time, the less the weight and the less likely to be chosen
    Double weightSoFar = 0.0;

    // create new list and hot swap the reference
    List<Double> finalWeights = new ArrayList<~>();
    for (Server server : nlb.getAllServers()) {
        ServerStats ss = stats.getSingleServerStat(server);
        double weight = totalResponseTime - ss.getResponseTimeAvg();
        weightSoFar += weight;
        finalWeights.add(weightSoFar);
    }
    setWeights(finalWeights);
```

每次执行会先计算所有服务的相应时间的和

```java
for (Server server : nlb.getAllServers()) {
    // this will automatically load the stats if not in cache
    ServerStats ss = stats.getSingleServerStat(server);
    totalResponseTime += ss.getResponseTimeAvg();
}
```

然后用时间和减去每个服务的相应平均时间作为权重值再加上上一个服务的权重值，这样一来，相应间越长的服务，分配的权重就会越低，由此就可以得到各个服务的权重

```java
List<Double> finalWeights = new ArrayList<~>();
for (Server server : nlb.getAllServers()) {
    ServerStats ss = stats.getSingleServerStat(server);
    double weight = totalResponseTime - ss.getResponseTimeAvg();
    weightSoFar += weight;
    finalWeights.add(weightSoFar);
}
setWeights(finalWeights);
```

最后得到的各个服务的值就是一个依次递增的数列，数列的项与前一项差距越小说明相应所花时间越，这么说可能太抽象，我们假设五个服务，把他们的list的值在x轴上做排列如下

原文链接：ribbon 的几种负载均衡

第三个节点因为相应的平均时间较长，所以权重小，距离上一个节点距离就近一些，此时，我们取出后一个节点（他的值最大），然后生成一个在0-这个节点值之间的随机数，那么，命中第二个和第三节点之间的概率就会小一些，换而言之，命中其他响应时间较短的节点的概率自然就提高了

再来看具体选择节点的代码，代码中，他选取了服务权重列表的最后一个服务权重为基准，做了一系判断

```
// last one in the list is the sum of all weights
double maxTotalWeight = currentWeights.size() == 0 ? 0 : currentWeights.get(currentWeights.size() - 1);
// No server has been hit yet and total weight is not initialized
// fallback to use round robin
if (maxTotalWeight < 0.001d || serverCount != currentWeights.size()) {
    server =  super.choose(getLoadBalancer(), key);
    if(server == null) {
        return server;
    }
}
```

此处，先取出最后一个服务的权重，因为这个服务权重是所有服务权重之和，如果最大的权重都小于0 001d，说明所有服务相应时间都很短或者可用服务和权重列表数量不一致（有服务在期间挂了或其），就默认使用轮询策略选择一个服务返回

```
else {
    // generate a random weight between 0 (inclusive) to maxTotalWeight (exclusive)
    double randomWeight = random.nextDouble() * maxTotalWeight;
    // pick the server index based on the randomIndex
    int n = 0;
    for (Double d : currentWeights) {
        if (d >= randomWeight) {
            serverIndex = n;
            break;
        } else {
            n++;
        }
    }

    server = allList.get(serverIndex);
}
```

这里就是按照之前说的，取一个在权重和的范围内的随机数，选择一个服务即可

3.4、最低并发策略

最低并发策略在BestAvailableRule中实现，这个算法就是给定一个随机的服务列表（为了避免大量务请求时选择同一个并发数最小的服务造成服务导致崩溃，通过ServerListSubsetFilter来限制获取一随机列表），然后选择出一个并发数最小的服务来请求。

```java
@Override
public Server choose(Object key) {
    if (loadBalancerStats == null) {
        return super.choose(key);
    }
    List<Server> serverList = getLoadBalancer().getAllServers();
    int minimalConcurrentConnections = Integer.MAX_VALUE;
    long currentTime = System.currentTimeMillis();
    Server chosen = null;
    for (Server server: serverList) {
        ServerStats serverStats = loadBalancerStats.getSingleServerStat(server);
        if (!serverStats.isCircuitBreakerTripped(currentTime)) {
            int concurrentConnections = serverStats.getActiveRequestsCount(currentTime);
            if (concurrentConnections < minimalConcurrentConnections) {
                minimalConcurrentConnections = concurrentConnections;
                chosen = server;
            }
        }
    }
    if (chosen == null) {
        return super.choose(key);
    } else {
        return chosen;
    }
}
```

3.5、可用性策略

可用性策略在AvailabilityFilteringRule中实现，他按照轮询规则来依次获取服务，如果当前获取到的务符合规则，则返回服务，如果连续十次没有获取到符合条件的服务，就调用父类的轮询方法来直接取服务。

```java
@Override
public Server choose(Object key) {
    int count = 0;
    Server server = roundRobinRule.choose(key);
    while (count++ <= 10) {
        if (predicate.apply(new PredicateKey(server))) {
            return server;
        }
        server = roundRobinRule.choose(key);
    }
    return super.choose(key);
}
```

选取规则默认使用AvailabilityPredicate，选取代码如下

```java
@Override
public boolean apply(@Nullable PredicateKey input) {
    LoadBalancerStats stats = getLBStats();
    if (stats == null) {
        return true;
    }
    return !shouldSkipServer(stats.getSingleServerStat(input.getServer()));
}


private boolean shouldSkipServer(ServerStats stats) {
    if ((CIRCUIT_BREAKER_FILTERING.get() && stats.isCircuitBreakerTripped())
            || stats.getActiveRequestsCount() >= activeConnectionsLimit.get()) {
        return true;
    }
    return false;
}
```

代码中判断，如果断路器已经被打开，或者服务的连接数超过最大连接数限制就过滤这个服务，进行一个服务的判断。

3.6、区域权重策略

这个策略在ZoneAvoidanceRule中实现。

```java
public ZoneAvoidanceRule() {
    super();
    ZoneAvoidancePredicate zonePredicate = new ZoneAvoidancePredicate( rule: this);
    AvailabilityPredicate availabilityPredicate = new AvailabilityPredicate( rule: this);
    compositePredicate = createCompositePredicate(zonePredicate, availabilityPredicate);
}

private CompositePredicate createCompositePredicate(ZoneAvoidancePredicate p1, AvailabilityPredicate p2) {
    return CompositePredicate.withPredicates(p1, p2)
                         .addFallbackPredicate(p2)
                         .addFallbackPredicate(AbstractServerPredicate.alwaysTrue())
                         .build();

}
```

```java
/**
 * Get the filtered servers from primary predicate, and if the number of the filtered servers
 * are not enough, trying the fallback predicates
 */
@Override
public List<Server> getEligibleServers(List<Server> servers, Object loadBalancerKey) {
    List<Server> result = super.getEligibleServers(servers, loadBalancerKey);
    Iterator<AbstractServerPredicate> i = fallbacks.iterator();
    while (!(result.size() >= minimalFilteredServers && result.size() > (int) (servers.size() * minimalFilteredPercentage))
            && i.hasNext()) {
        AbstractServerPredicate predicate = i.next();
        result = predicate.getEligibleServers(servers, loadBalancerKey);
    }
    return result;
}
```

这个策略中有两个条件，将ZoneAvoidancePredicate作为主要规则，AvailabilityPredicate作为次规则，然后用CompositePredicate将这两个规则封装作为复核规则，主要规则满足则直接返回服务否则调用次要规则获取服务，然后在获取到的服务中轮询调用。

ZoneAvoidanceRule.getAvailableZones用来获取可用服务，流程如下

```
int instanceCount = zoneSnapshot.getInstanceCount();
if (instanceCount == 0) {
    availableZones.remove(zone);            剔除服务数量为零的区域
    limitedZoneAvailability = true;
} else {
    double loadPerServer = zoneSnapshot.getLoadPerServer();
    if (((double) zoneSnapshot.getCircuitTrippedCount())       剔除熔断率
            / instanceCount >= triggeringBlackoutPercentage    >triggeringBlackoutPercenta
            || loadPerServer < 0) {                            ge(默认99%)或负载小于0的区
        availableZones.remove(zone);
        limitedZoneAvailability = true;
    } else {
        if (Math.abs(loadPerServer - maxLoadPerServer) < 0.000001d) {
            // they are the same considering double calculation
            // round error
            worstZones.add(zone);           将负载超过最大负载的区域添加到worstZones中
        } else if (loadPerServer > maxLoadPerServer) {
            maxLoadPerServer = loadPerServer;
            worstZones.clear();
            worstZones.add(zone);
        }
    }
}
```

这里的worstZones并不需要全部从可用服务中剔除，但是要随机剔除一个

```
if (maxLoadPerServer < triggeringLoad && !limitedZoneAvailability) {
    // zone override is not needed here
    return availableZones;
}
String zoneToAvoid = randomChooseZone(snapshot, worstZones);
if (zoneToAvoid != null) {
    availableZones.remove(zoneToAvoid);
}
return availableZones;
```

3.7、重试策略

重试策略在RetryRule中实现，他的具体做法是在选定的负载均衡策略上，如果调用失败则不断调用定的负载策略获取服务，直到超过最大时间限制或成功，其默认负载策略是轮询。

```
IRule subRule = new RoundRobinRule();
long maxRetryMillis = 500;
```

原文链接：ribbon 的几种负载均衡

```java
public Server choose(ILoadBalancer lb, Object key) {
    long requestTime = System.currentTimeMillis();
    long deadline = requestTime + maxRetryMillis;

    Server answer = null;

    answer = subRule.choose(key);

    if (((answer == null) || (!answer.isAlive()))
            && (System.currentTimeMillis() < deadline)) {

        InterruptTask task = new InterruptTask( millis: deadline
                - System.currentTimeMillis());

        while (!Thread.interrupted()) {
            answer = subRule.choose(key);

            if (((answer == null) || (!answer.isAlive()))
                    && (System.currentTimeMillis() < deadline)) {
                /* pause and retry hoping it's transient */
                Thread.yield();
            } else {
                break;
            }
        }

        task.cancel();
    }

    if ((answer == null) || (!answer.isAlive())) {
        return null;
    } else {
        return answer;
    }
}
```

最大时间和判断

原文链接：ribbon 的几种负载均衡