Atitit 负载均衡 ribbon robbin的使用

目录

[1. Pc+俩快网卡方案 1](#_Toc16905)

[2. 场景与性能 1](#_Toc16499)

[2.1. Location指令并发，提升一个数据量 1](#_Toc1105)

[2.2. Nignx 处理并发一般在1-2w， 1](#_Toc15444)

[2.3. 自制硬件负载均衡器也是不错选择 多网卡 2](#_Toc1700)

[3. Code LoadBalanceTest 2](#_Toc6431)

[3.1. 客户端负载均衡Springcloud Ribbon简介 3](#_Toc29728)

[3.2. 带key的选择实现负载均衡 5](#_Toc14262)

# Pc+俩快网卡方案

硬件负载均衡解决方案是直接在服务器和外部网络间安装负载均衡设备，这种设备通常称之为负载均衡器，由于专门的设备完成专门的任务，独立于操作系统，整体性能得到大量提高，加上多样化的负载均衡策略，智能化的流量管理，可达到最佳的负载均衡需求。

负载均衡器有多种多样的形式，除了作为独立意义上的负载均衡器外，有些负载均衡器集成在交换设备中，置于服务器与Internet链接之间，有些则以两块网络适配器将这一功能集成到PC中，一块连接到Internet上，一块连接到后端服务器群的内部网络上。一般而言，硬件负载均衡在功能、性能上优于软件方式，不过成本昂贵。

# 场景与性能

## Location指令并发，提升一个数据量

## Nignx 处理并发一般在1-2w，

1. 可以承担高的负载压力且稳定，一般能支撑超过1万次的并发；

****使用场景：****一般应用与PV 几十万甚至百万的互联网应用，一般的软件负载均衡器例如Nignx 处理并发一般在1-2w，不足以支撑如此大的网络请求，所以在其之前通常会防止类似F5这样的硬件负载均衡器帮助控制网络请求。如果一般的互联网企业网络请求数在1w左右的可以考虑使用Nignx（软件负载均衡器）就足以满足当前的业务了

.f5则. HTTP请求数：7M 最大L4并发连接数：24M

## 自制硬件负载均衡器也是不错选择 多网卡

千m，万兆网卡。。

# Code LoadBalanceTest

**package** zuulx;

**import** com.netflix.loadbalancer.BaseLoadBalancer;

**import** com.netflix.loadbalancer.RandomRule;

**import** com.netflix.loadbalancer.Server;

**public** **class** LoadBalanceTest {

**public** **static** **void** main(String[] args) {

BaseLoadBalancer blb=**new** BaseLoadBalancer();

blb.addServer(**new** Server("host0", 80));

blb.addServer(**new** Server("host1", 81));

blb.addServer(**new** Server("host2", 82));

blb.setRule(**new** RandomRule());

System.***out***.println(blb.chooseServer() +" "); ;

}

}

## 客户端负载均衡Springcloud Ribbon简介

Ribbon时Netflix发布的负载均衡器，它有助于控制HTTP和TCP客户端的行为。为Ribbon配置服务提供者地址列表后，Ribbon就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon默认为我们提供了很多的负载均衡算法，例如轮询、随机等。我们也可以自定负载均衡算法。

有了Ribbon之后，我们的工程架构：

使用负载均衡带来的好处很明显：

1. 当集群里的1台或者多台服务器down的时候，剩余的没有down的服务器可以保证服务的继续使用
2. 使用了更多的机器保证了机器的良性使用，不会由于某一高峰时刻导致系统cpu急剧上升

负载均衡有好几种实现策略，常见的有：

1. 随机 (Random)
2. 轮询 (RoundRobin)
3. 一致性哈希 (ConsistentHash)
4. 哈希 (Hash)
5. 加权（Weighted

ribbon是一个为客户端提供负载均衡功能的服务，它内部提供了一个叫做ILoadBalance的接口代表负载均衡器的操作，比如有添加服务器操作、选择服务器操作、获取所有的服务器列表、获取可用的服务器列表等等。ILoadBalance的继承关系如下：

完整过程是：

LoadBalancerClient（RibbonLoadBalancerClient是实现类）在初始化的时候（execute方法），会通过ILoadBalance（BaseLoadBalancer是实现类）向Eureka注册中心获取服务注册列表，并且每10s一次向EurekaClient发送“ping”，来判断服务的可用性，如果服务的可用性发生了改变或者服务数量和之前的不一致，则从注册中心更新或者重新拉取。LoadBalancerClient有了这些服务注册列表，就可以根据具体的IRule来进行负载均衡。

其中RandomRule表示随机策略、RoundRobinRule表示轮询策略、WeightedResponseTimeRule表示加权策略、BestAvailableRule表示请求数最少策略等等。

随机策略很简单，就是从服务器中随机选择一个服务器，RandomRule的实现代码如下：

Atitit 负载均衡的总结

Atitit CateIT重要架构技术负载均衡 iduah1 impt

体系树路径：CS IT>重要架构技术》负载均衡

密级和保密期限：：

Keywords和摘要：none

目录

1. 场景 提升稳定性和性能 1

2. 负载均衡原理 2

3. 常见策略 2

3.1. 轮训 随机 权重 2

4. 分类模式 2

4.1. 负载均衡俩种模式 客户端 vs 服务端 2

4.2. 对称模式 vs 非对称方式 2

5. 目前比较常用的负载均衡技术主要有： 3

5.1. 1. 基于DNS的负载均衡 3

5.2. 2. 反向代理负载均衡 3

5.3. 3. 基于NAT的负载均衡技术 3

5.4. 1、基于特定服务器软件的负载均衡  网络协议 “重定向”功能 Location指令 3

6. 常见工具 4

6.1. F5 dns lvs Nginx 4

6.2. Ngnix apache dubbo 4

6.3. Springcloud Zuul服务端 负载均衡 4

6.4. 客户端负载均衡Springcloud Ribbon简介 4

6.5. Location指令客户端跳转也可以 4

7. 类似概念 4

7.1. 负载均衡与集群的关系 ，.重定向”功能 5

8. 内部细节实现 5

8.1. Atitit 负载均衡的心跳与探活tebi 5

8.2. 特殊的路由探活 5

9. Ref 5

## 带key的选择实现负载均衡

**package** zuulx;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.netflix.client.config.IClientConfig;

**import** com.netflix.loadbalancer.AbstractLoadBalancerRule;

**import** com.netflix.loadbalancer.BaseLoadBalancer;

**import** com.netflix.loadbalancer.DynamicServerListLoadBalancer;

**import** com.netflix.loadbalancer.RandomRule;

**import** com.netflix.loadbalancer.Server;

**import** com.netflix.loadbalancer.ZoneAwareLoadBalancer;

**public** **class** LoadBalanceTest {

**public** **class** KeyServerRule **extends** AbstractLoadBalancerRule{

@Override

**public** Server choose(Object key) {

List<Server> allServerList =**this**.getLoadBalancer().getAllServers();

**for** (Server server : allServerList) {

// System.out.println(server.getZone());

**if** (server.getZone().toString().toLowerCase().equals(key.toString().toLowerCase()))

**return** server;

}

**return** **null**;

}

@Override

**public** **void** initWithNiwsConfig(IClientConfig clientConfig) {

// **TODO** Auto-generated method stub

}

}

**public** **static** **void** main(String[] args) {

// extracted();

// 带key的实现机制

//LoadbalanceImp blb = new LoadBalanceTest().new LoadbalanceImp();

// DynamicServerListLoadBalancer blb=new DynamicServerListLoadBalancer();

BaseLoadBalancer blb=**new** BaseLoadBalancer();

Server svr1 = **new** Server("svr1");

svr1.setZone("key1");

Server svr2 = **new** Server("svr2");

svr2.setZone("key2");

List li = **new** ArrayList<>();

li.add(svr2);

li.add(svr1);

blb.addServer(svr1);

blb.addServer(svr2);

blb.setRule( **new** LoadBalanceTest(). **new** KeyServerRule());

// blb.addServers(li);

// System.out.println(blb.choose("key1"));

System.***out***.println(blb.getAllServers());

System.***out***.println(blb.chooseServer("key1"));

}

**private** **static** **void** extracted() {

BaseLoadBalancer blb = **new** BaseLoadBalancer();

blb.addServer(**new** Server("host0", 80));

blb.addServer(**new** Server("host1", 81));

blb.addServer(**new** Server("host2", 82));

blb.setRule(**new** RandomRule());

System.***out***.println(blb.chooseServer() + " ");

;

}

**package** zuulx;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.netflix.loadbalancer.BaseLoadBalancer;

**import** com.netflix.loadbalancer.DynamicServerListLoadBalancer;

**import** com.netflix.loadbalancer.RandomRule;

**import** com.netflix.loadbalancer.Server;

**import** com.netflix.loadbalancer.ZoneAwareLoadBalancer;

**public** **class** LoadBalanceTest {

**public** **class** LoadbalanceImp **extends** DynamicServerListLoadBalancer {

**public** Server chooseServer(Object key) {

List<Server> allServerList = **this**.allServerList;

**for** (Server server : allServerList) {

// System.out.println(server.getZone());

**if** (server.getZone().toString().toLowerCase().equals(key.toString().toLowerCase()))

**return** server;

}

**return** **null**;

}

}

**public** **static** **void** main(String[] args) {

// extracted();

// 带key的实现机制

LoadbalanceImp blb = **new** LoadBalanceTest().**new** LoadbalanceImp();

// DynamicServerListLoadBalancer blb=new DynamicServerListLoadBalancer();

Server svr1 = **new** Server("svr1");

svr1.setZone("key1");

Server svr2 = **new** Server("svr2");

svr2.setZone("key2");

List li = **new** ArrayList<>();

li.add(svr2);

li.add(svr1);

blb.addServer(svr1);

blb.addServer(svr2);

// blb.addServers(li);

// System.out.println(blb.choose("key1"));

System.***out***.println(blb.getAllServers());

System.***out***.println(blb.chooseServer("key1"));

}

**private** **static** **void** extracted() {

BaseLoadBalancer blb = **new** BaseLoadBalancer();

blb.addServer(**new** Server("host0", 80));

blb.addServer(**new** Server("host1", 81));

blb.addServer(**new** Server("host2", 82));

blb.setRule(**new** RandomRule());

System.***out***.println(blb.chooseServer() + " ");

;

}

}