Atitit 数据库 负载均衡 方法总结

目录

[1. 对称模型负载均衡 vs 非对称模型 1](#_Toc847)

[1.1. 业务分离法 1](#_Toc23629)

[1.2. App + db分布式分离法 1](#_Toc22175)

[2. 服务器负载均衡有三大基本Feature：负载均衡算法，健康检查和会话保持 1](#_Toc1332)

[3. 负载均衡算法 2](#_Toc22662)

[3.1. 随机 轮训 权重法（适用于对称模型） 2](#_Toc23116)

[3.2. 最快响应时间（Fast Reponse time）： 2](#_Toc22263)

[3.3. 根据指令 查询 读写分离法 2](#_Toc15215)

[3.4. 根据userkey ，时间key计算法 2](#_Toc7502)

[4. 按照网络链路层次分类 2](#_Toc11720)

[5. 常见常见场景 3](#_Toc28229)

[5.1. 全局库key-db模式 3](#_Toc23169)

[5.2. 主从模式读写分离负载均衡 3](#_Toc17000)

[5.3. 分区模式 3](#_Toc9170)

[6. 6. 常见工具 4 4](#_Toc5989)

[7. Ribbon的api比较标准规范 4](#_Toc2981)

[7.1. com.netflix.loadbalancer.ILoadBalancer接口实现的。 4](#_Toc16048)

[7.1.1. com.netflix.loadbalancer.BaseLoadBalancer 5](#_Toc16417)

[7.1.2. DynamicServerListLoadBalancer 5](#_Toc20036)

[7.1.3. ZoneAwareLoadBalancer 6](#_Toc1509)

# 对称模型负载均衡 vs 非对称模型

## 业务分离法

## App + db分布式分离法

# 服务器负载均衡有三大基本Feature：负载均衡算法，健康检查和会话保持

，这三个Feature是保证负载均衡正常工作的基本要素。

# 负载均衡算法

## 随机 轮训 权重法（适用于对称模型）

## 最快响应时间（Fast Reponse time）：

新的连接传递给那些响应最快的服务器。当其中某个服务器发生故障，AX就把其从服务器队列中拿出，不参加下一次的用户请求的分配，直到其恢复正常

## 根据指令 查询 读写分离法

## 根据userkey ，时间key计算法

# 按照网络链路层次分类

#### ****链路层（OSI 第二层）负载均衡****

　　在通信协议的数据链路层修改mac地址，进行负载均衡。 　　数据分发时，不修改ip地址（因为还看不到ip地址），只修改目标mac地址，并且配置所有后端服务器虚拟ip和负载均衡器ip地址一致，达到不修改数据包的源地址和目标地址，进行数据分发的目的。 　　实际处理服务器ip和数据请求目的ip一致，不需要经过负载均衡服务器进行地址转换，可将响应数据包直接返回给用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈。也称为直接路由模式（DR模式）。如下图：

#### ****传输层（OSI 第四层）负载均衡****

　　传输层是 OSI 第四层，包括 TCP 和 UDP。流行的传输层负载均衡器有 HAProxy（这个也用于应用层负载均衡）和 IPVS。 ****主要通过报文中的目标地址和端口****，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

#### ****应用层（OSI 第七层）负载均衡****

应用层是 OSI 第七层。它包括 HTTP、HTTPS 和 WebSockets。一款非常流行又久经考验的应用层负载均衡器就是 Nginx[恩静埃克斯 = Engine X]。 　　所谓七层负载均衡，也称为“内容交换”，也就是主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。注意此时可以看到具体的http请求的完整url，因此可以实现下图所示的分发

七层负载均衡的好处，是使得整个网络更"智能化"，比如上面列举的负载均衡的好处，大部分都基于七层负载均衡。例如访问一个网站的用户流量，可以通过七层的方式，将对图片类的请求转发到特定的图片服务器并可以使用缓存技术；将对文字类的请求可以转发到特定的文字服务器并可以使用压缩技术。当然这只是七层应用的一个小案例，从技术原理上，这种方式可以对客户端的请求和服务器的响应进行任意意义上的修改，极大的提升了应用系统在网络层的灵活性。 　　另外一个常常被提到功能就是安全性。网络中最常见的SYN Flood攻击，即黑客控制众多源客户端，使用虚假IP地址对同一目标发送SYN攻击，通常这种攻击会大量发送SYN报文，耗尽服务器上的相关资源，以达到Denial of Service(DoS)的目的。从技术原理上也可以看出，四层模式下这些SYN攻击都会被转发到后端的服务器上；而七层模式下这些SYN攻击自然在负载均衡设备上就截止，不会影响后台服务器的正常运营。另外负载均衡设备可以在七层层面设定多种策略，过滤特定报文，例如SQL Injection等应用层面的特定攻击手段，从应用层面进一步提高系统整体安全。 　　现在的七层负载均衡，主要还是着重于应用广泛的HTTP协议，所以其应用范围主要是众多的网站或者内部信息平台等基于B/S开发的系统。 四层负载均衡则对应其他TCP应用，例如基于C/S开发的ERP等系统。

# 常见常见场景

## 全局库key-db模式

更具userid 选择服务器

## 主从模式读写分离负载均衡

根据 读写sql实现选择数据库服务器

一般是驱动已经实现了。也可以使用Ribbon自己实现

## 分区模式

# 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

# Ribbon的api比较标准规范

## com.netflix.loadbalancer.ILoadBalancer接口实现的。

总结一下：  
ILoadBalancer接口实现类做了以下的一些事情：  
1.维护了存储服务实例Server对象的二个列表。一个用于存储所有服务实例的清单，一个用于存储正常服务的实例清单  
2.初始化得到可用的服务列表，启动定时任务去实时的检测服务列表中的服务的可用性，并且间断性的去更新服务列表，结合注册中心。  
3.选择可用的服务进行调用（这个一般交给IRule去实现，不同的轮询策略）

**三个很重要的概念**

* ServerList接口：定义用于获取服务器列表的方法的接口,主要实现DomainExtractingServerList接口，每隔30s种执行getUpdatedListOfServers方法进行服务列表的更新。
* ServerListUpdater接口：主要实现类EurekaNotificationServerListUpdater和PollingServerListUpdater（默认使用的是PollingServerListUpdater，结合Eureka注册中心，定时任务的方式进行服务列表的更新）
* ServerListFilter接口：根据LoadBalancerStats然后根据一些规则去过滤部分服务，比如根据zone（区域感知）去过滤。（主要实现类ZonePreferenceServerListFilter的getFilteredListOfServers会在更新服务列表的时候去执行）。

### com.netflix.loadbalancer.BaseLoadBalancer

com.netflix.loadbalancer.BaseLoadBalancer类是Ribbon负载均衡器的基础实现类，在该类中定义了很多关于负载均衡器相关的基础内容。

定义并维护了两种存储服务实例Server对象的列表。一个用于存储所有服务实例的清单，一个用于存储正常服务的实例清单。

负载均衡的处理原则IRule对象，从BaseLoadBalancer中chooseServer(Object key)的实现源码可以知道，负载均衡器实际将服务实例选择的任务委托给IRule实例中的choose函数来实现。  
默认初始化了RoundRobinRule实现，RoundRobinRule实现了最基本且常用的线性负载均衡规则。

getReachableServers():获取可用的服务实例列表。由于BaseLoadBalancer中单独维护了一个正常服务的实例清单，所以直接返回即可

@Overridepublic List<Server> getReachableServers() {

return Collections.unmodifiableList(upServerList);}

getAllServers():获取所有的服务实例列表。由于BaseLoadBalancer中单独维护了一个正常服务的实例清单，所以直接返回即可。

### DynamicServerListLoadBalancer

com.netflix.loadbalancer.DynamicServerListLoadBalancer类继承com.netflix.loadbalancer.BaseLoadBalancer类，它是对基础负载均衡器的扩展。  
该负载均衡器中，实现了服务实例清单在运行期的动态更新能力；同时，它还具备了对服务实例清单的过滤功能，也就是说，我们可以通过过滤器来选择性的获取一批服务实例清单。

ZoneAffinityServerListFilter：该过滤器基于"区域感知（Zone Affinity）"的方式实现服务实例的过滤，也就说，它会根据提供服务的实例所处于的区域（Zone）与消费者自身所处区域（Zone）进行比较，过滤掉那些不是同处一个区域的实例

### ZoneAwareLoadBalancer

ZoneAwareLoadBalancer负载均衡器是对DynamicServerListLoadBalancer的扩展。在DynamicServerListLoadBalancer中，我们可以看到它并没有重写选择具体服务实例的chooseServer函数，所以它依然会采用在BaseLoadBalancer中实现的算法。使用RoundRobinRule规则，以线性轮询的方式来选择调用的服务实例，该算法实现简单并没有区域（Zone）的概念，所以它会把所有实例视为一个Zone下的节点来看待，这样就会周期性的跨区域（Zone）访问的情况，由于跨区域会产生更高的延迟，这些实例主要以防止区域性故障实现高可用为目的而不能作为常规访问的实例，所以在多区域部署的情况会出现一定的性能问题，而该负载均衡器则可以规避这样的问题。

spring cloud ribbon学习三：负载均衡器ILoadBalancer接口及其实现 - 简书

**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() + " ");

;

}