|  |
| --- |
| KCloudy |
| **SpringBoot In Linux** |
| [文档副标题] |

|  |
| --- |
| 田 长军  [日期] |

目录

[**1.** **部署前准备** 0](#_Toc21768647)

[**1.1** **安装Java** 0](#_Toc21768648)

[**1.2** **安装Maven** 0](#_Toc21768649)

[**1.2.1** **下载Maven安装包** 0](#_Toc21768650)

[**1.2.2** **解压Maven安装包** 0](#_Toc21768651)

[**1.2.3** **配置Maven路径** 0](#_Toc21768652)

[**1.2.4** **验证Maven是否配置成功** 0](#_Toc21768653)

[**1.2.5** **使用mvn -v时，出现错误：JAVA\_HOME should point to a JDK not a JRE** 0](#_Toc21768654)

[**1.2.6** **配置国内阿里云Maven镜像** 0](#_Toc21768655)

[**1.3** **安装Git** 0](#_Toc21768656)

[**1.4** **安装Docker** 0](#_Toc21768657)

[**1.4.1** **使用yum源安装** 0](#_Toc21768658)

[**1.4.2** **安装Docker Compose** 0](#_Toc21768659)

[**1.4.3** **安装Docker Machine** 0](#_Toc21768660)

[**1.4.4** **使用国内Docker镜像源** 0](#_Toc21768661)

[**1.4.5** **启动及测试** 0](#_Toc21768662)

[**2.** **Spring Consul** 0](#_Toc21768663)

[**2.1** **安装Consul** 0](#_Toc21768664)

[**2.1.1** **安装三个Consul服务** 0](#_Toc21768665)

[**2.1.2** **使用Docker安装Consul** 0](#_Toc21768666)

[**2.1.3** **其他** 0](#_Toc21768667)

[**2.2** **其他** 0](#_Toc21768668)

[**3.** **其他** 0](#_Toc21768669)

**SpringBoot In Linux**

1. **部署前准备**
   1. **安装Java**

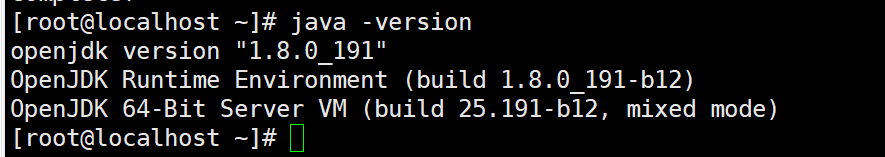
sudo yum -y install java-1.8.0-openjdk #jre

sudo yum -y install java-1.8.0-openjdk-devel #jdk

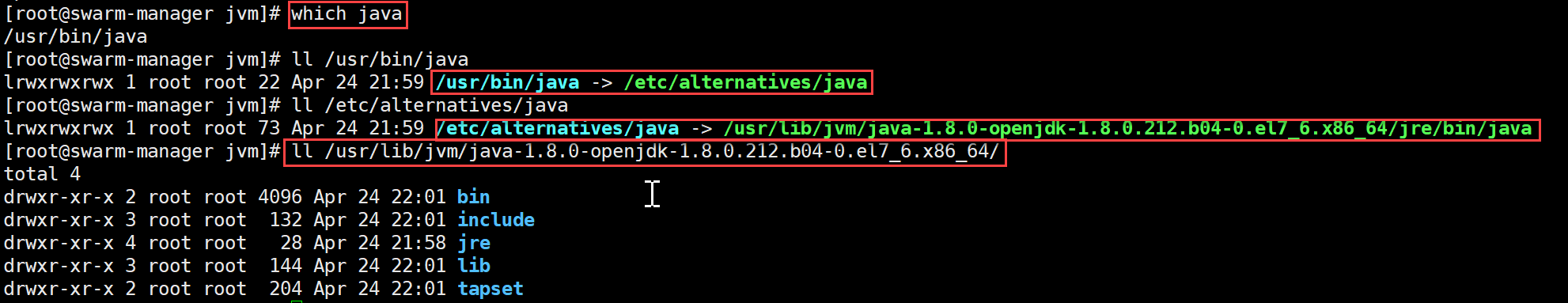
但对于centos有个问题，不可以安装成gcj(例如：gij(GNU libgcj) version 4.4.6 20110731(Red Hat 4.4.6-3)),导致[Jekins不工作](https://issues.jenkins-ci.org/browse/JENKINS-743),如果查看java版本类似上例，需要卸载，安装其他版本

java -version #查看Java版本

yum remove java #卸载异常版本



which java #查看java执行路径，备注：java实际的路径为-/usr/lib/jvm



设置java路径：vi /etc/profile 在文件最后添加下列配置

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.212.b04-0.el7\_6.x86\_64

export JRE\_HOME=$JAVA\_HOME/jre

export CLASSPATH=$JAVA\_HOME/lib:$JRE\_HOME/lib:$CLASSPATH

export PATH=$JAVA\_HOME/bin:$JRE\_HOME/bin:$PATH

使配置生效： source /etc/profile

* 1. **安装Maven**
     1. **下载Maven安装包**

wget <https://mirrors.tuna.tsinghua.edu.cn/apache/maven/maven-3/3.6.2/binaries/apache-maven-3.6.2-bin.tar.gz>

* + 1. **解压Maven安装包**

tar -zxvf apache-maven-3.6.2-bin.tar.gz

mv apache-maven-3.6.2 /usr/local/maven3

* + 1. **配置Maven路径**

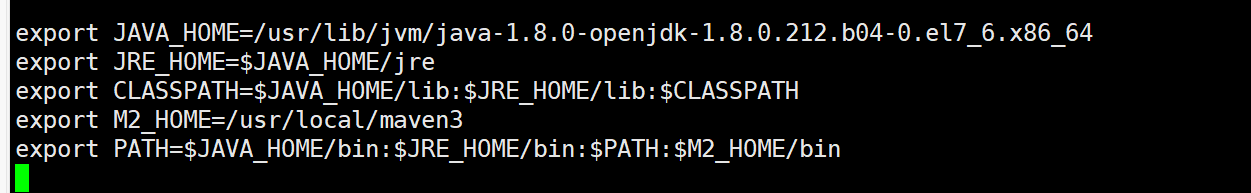
vi /etc/profile #最后添加以下内容

export M2\_HOME=/usr/local/maven3

export PATH=$PATH:$JAVA\_HOME/bin:$M2\_HOME/bin #保存退出后运行下面的命令使配置生效，或者重启服务器生效

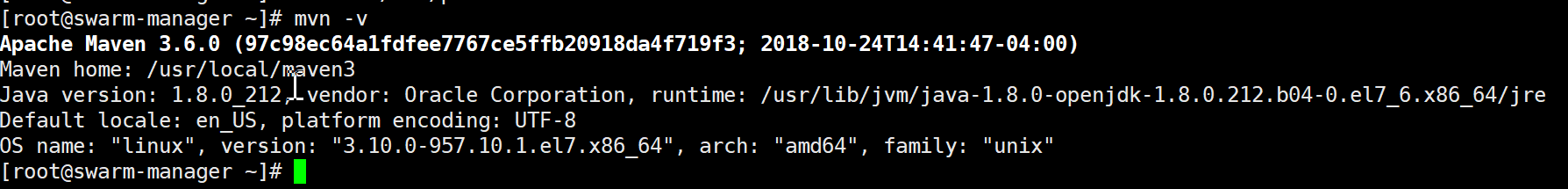
source /etc/profile

配置结果如下图：

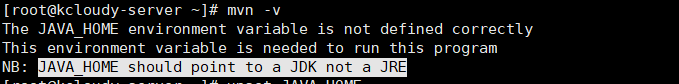


* + 1. **验证Maven是否配置成功**

mvn -v



* + 1. **使用mvn -v时，出现错误：JAVA\_HOME should point to a JDK not a JRE**



unset JAVA\_HOME #重置下JAVA\_HOME

* + 1. **配置国内阿里云Maven镜像**

vi /usr/local/maven3/conf/settings.xml

配置mirrors的子节点，添加如下mirror

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>central</mirrorOf>

<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>



* 1. **安装Git**

sudo yum -y install git

git --version

* 1. **安装Docker**
     1. **使用yum源安装**

sudo wget -P /etc/yum.repos.d/ <http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo>

sudo yum install -y docker-ce-18.09.3-3.el7 docker-ce-cli-18.09.3 containerd.io-18.09.3

docker --version

* + 1. **安装Docker Compose**

curl -L https://github.com/docker/compose/releases/download/1.23.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose

docker-compose --version

* + 1. **安装Docker Machine**

sudo wget -P /etc/yum.repos.d/ http://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo

sudo yum install -y VirtualBox-5.1 #安装VirtualBox

curl -L https://github.com/docker/machine/releases/download/v0.16.1/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine &&

chmod +x /tmp/docker-machine &&

sudo cp /tmp/docker-machine /usr/local/bin/docker-machine

docker-machine -version

* + 1. **使用国内Docker镜像源**

vi /etc/docker/daemon.json #修改为以下内容

{

"exec-opts": ["native.cgroupdriver=systemd"],

"registry-mirrors" : [

"http://ovfftd6p.mirror.aliyuncs.com",

"http://registry.docker-cn.com",

"http://docker.mirrors.ustc.edu.cn",

"http://hub-mirror.c.163.com"

],

"insecure-registries" : [

"registry.docker-cn.com",

"docker.mirrors.ustc.edu.cn"

],

"debug" : true,

"experimental" : true

}

* + 1. **启动及测试**
       1. **启动Docker**

service docker start

chkconfig docker on

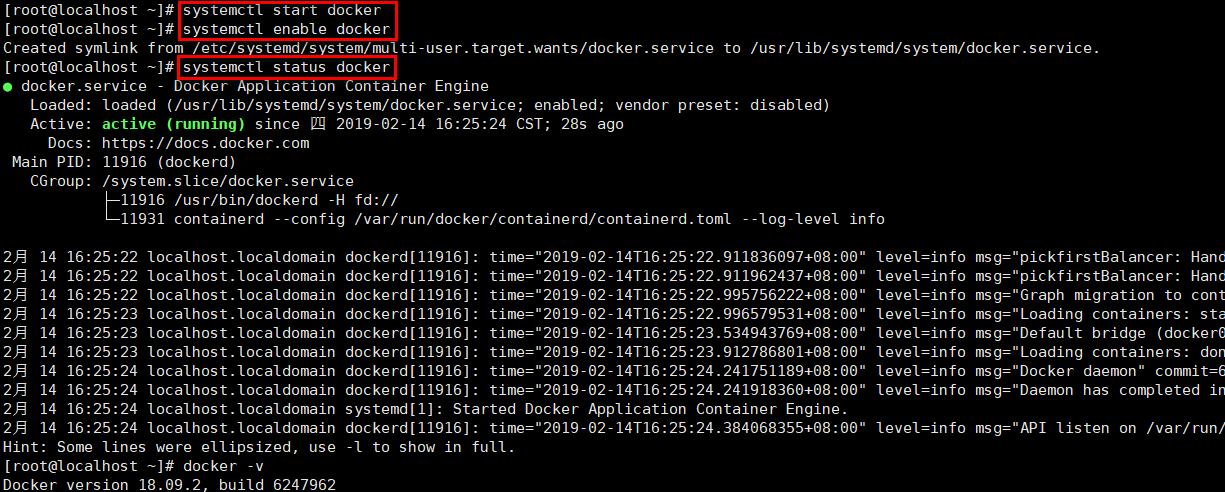
#LCTT 译注：此处采用了旧式的 sysv 语法，如采用CentOS 7中支持的新式 systemd 语法，如下：

systemctl start docker.service #开机自启 Docker CE

systemctl enable docker.service

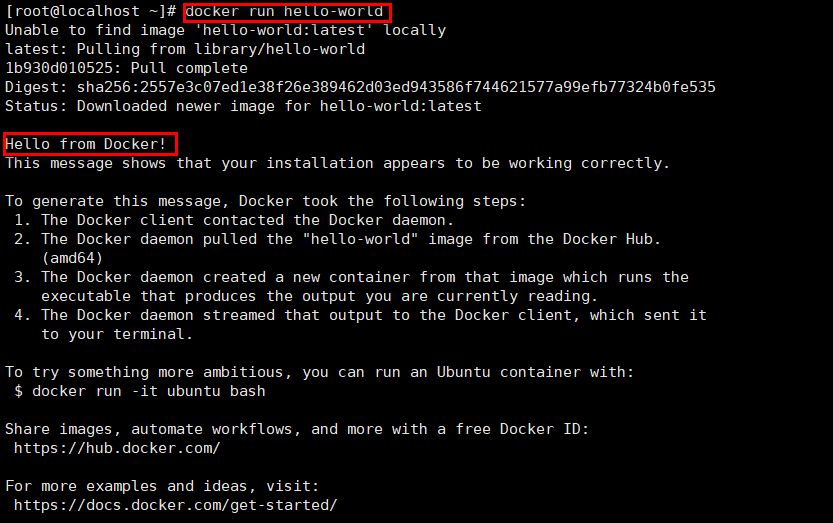
systemctl status docker #查看Docker状态

docker -v #查看docker版本



* + - 1. **测试Docker是否安装正确**

docker run hello-world



1. **注册中心：Consul**
   1. **安装Consul**
      1. **安装三个Consul服务**

192.168.56.12:8300作为Consul node1(leader)

192.168.56.12:8400作为Consul node2

192.168.56.12:8500作为Consul node3

* + 1. **使用Docker安装Consul**
* 搜索镜像

docker search consul

* 下载镜像

docker pull progrium/consul

* 安装镜像

#consul 服务端 node1

docker run -d -h node1 --name consul\_node1 --restart=always \

-v /data/consul:/consul/data \

-p 8300:8300 -p 8301:8301 -p 8301:8301/udp -p 8302:8302 -p 8302:8302/udp -p 8400:8400 -p 8500:8500 \

-e 'CONSUL\_LOCAL\_CONFIG={"skip\_leave\_on\_interrupt": true}' \

-e CONSUL\_BIND\_INTERFACE=eth0 \

progrium/consul -server -bootstrap

#获取 node1 的IP

JOIN\_IP="$(docker inspect -f '{{.NetworkSettings.IPAddress}}' consul\_node1)"

#consul 服务端 node2

docker run --name consul\_node2 -d --restart=always \

-e 'CONSUL\_LOCAL\_CONFIG={"skip\_leave\_on\_interrupt": true}' \

-e CONSUL\_BIND\_INTERFACE=eth0 \

progrium/consul -h node2 –server -join $JOIN\_IP

#consul 服务端 node3

docker run --name consul\_node3 -d --restart=always \

-e 'CONSUL\_LOCAL\_CONFIG={"skip\_leave\_on\_interrupt": true}' \

-e CONSUL\_BIND\_INTERFACE=eth0 \

progrium/consul –h node3 -server -join $JOIN\_IP

agent : 表示启动 Agent 进程。

-server：表示启动 Consul Server 模式。

-client：表示启动 Consul Cilent 模式。

-bootstrap：表示这个节点是 Server-Leader ，每个数据中心只能运行一台服务器。技术角度上讲 Leader 是通过 Raft 算法选举的，但是集群第一次启动时需要一个引导 Leader，在引导群集后，建议不要使用此标志。

-ui：表示启动 Web UI 管理器，默认开放端口 8500，所以上面使用 Docker 命令把 8500 端口对外开放。

-node：节点的名称，集群中必须是唯一的。

-client：表示 Consul 将绑定客户端接口的地址，0.0.0.0 表示所有地址都可以访问。

-join：表示加入到某一个集群中去。 如：-json=192.168.1.23

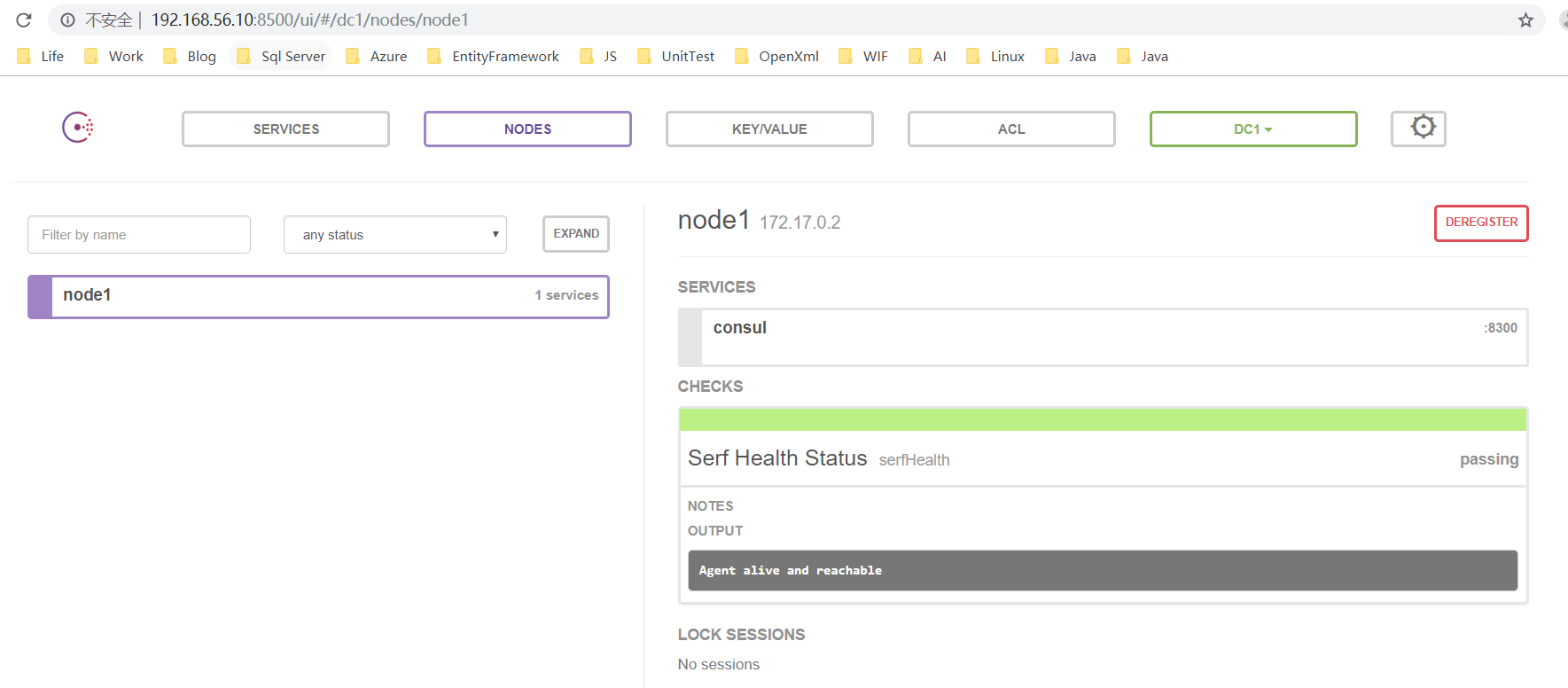
* 看集群成员

docker exec -t consul\_node1 consul members

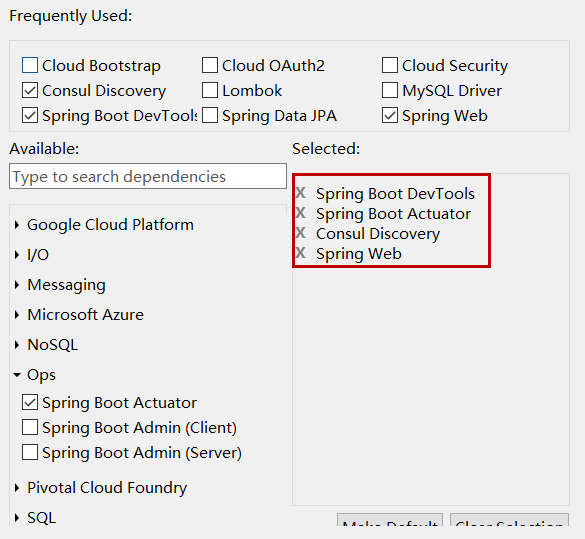
docker exec -t consul\_node1 consul operator raft list-peers

* + 1. **安装完成，使用WebUI查看结果**

访问路径：<http://192.168.56.10:8500/ui/#/dc1/services/consul>



* 1. **创建Consul的服务端**
     1. **创建SpringCloud项目，选择相关组件如下：**



pom.xml如下

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-consul-discovery</artifactId>

</dependency>

* + 1. **配置Consul服务端**

spring:

application:

name: service-provider

cloud:

consul:

discovery:

health-check-path: /actuator/health # 健康健康路径，也可以自己写

health-check-interval: 10s # 检测轮训时间 1m 代码1分钟

#instance-id: consul-client00 实例ID，唯一值

instance-id: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

host: 192.168.56.10

port: 8500

* + 1. **设置启动类**

@EnableDiscoveryClient

@SpringBootApplication

public class SpringBootConsulApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootConsulApplication.class, args);

}

}

* + 1. **创建相关服务的RestController**

@RestController

public class HelloController {

@RequestMapping("/hello")

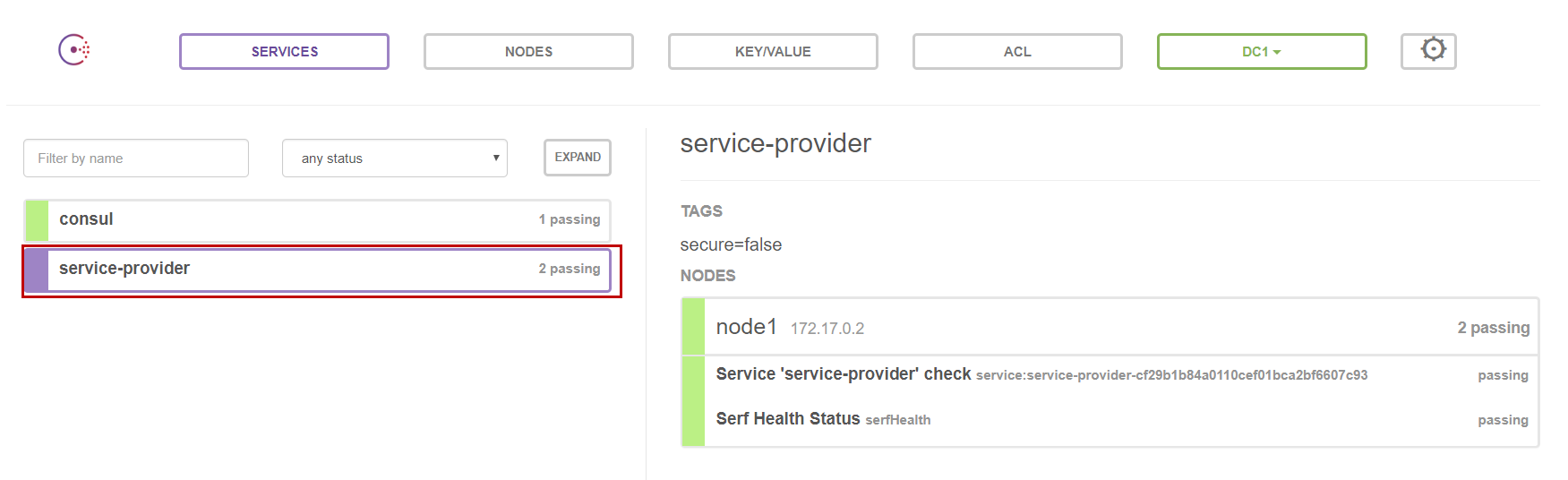
public @ResponseBody User hello() {

return new User(1, "hello consul");

}

}

* + 1. **启动项目，在Consul中查看已注册的服务**



* 1. **创建Consul的消费端**
     1. **创建SpringCloud项目（项目结构同消费端）**
     2. **配置Consul消费端**

server.port=8504

spring.application.name=service-consumer

spring.cloud.consul.host=192.168.56.10

spring.cloud.consul.port=8500

#设置不需要注册到 consul 中

spring.cloud.consul.discovery.register=false

* + 1. **测试调用服务端Rest服务**

@RestController

public class ServiceController {

@Autowired

private LoadBalancerClient loadBalancer;

@Autowired

private DiscoveryClient discoveryClient;

/\*\*

\* 获取所有服务

\*/

@RequestMapping("/services")

public Object services() {

return discoveryClient.getInstances("service-provider");

}

/\*\*

\* 从所有服务中选择一个服务（轮询）

\*/

@RequestMapping("/discover")

public Object discover() {

return loadBalancer.choose("service-provider ").getUri().toString();

}

}

Controller 中有俩个方法，一个是获取所有服务名为service-producer的服务信息并返回到页面，一个是随机从服务名为service-producer的服务中获取一个并返回到页面。

* + 1. **启动项目，访问Services服务：**<http://127.0.0.1:8504/services>，返回结果如下：

[{"instanceId":"service-provider-cf29b1b84a0110cef01bca2bf6607c93","serviceId":"service-provider","host":"192.168.2.115","port":8503,"secure":false,"metadata":{"secure":"false"},"uri":"http://192.168.2.115:8503","scheme":null}]

* + 1. **使用RestTemplate访问服务端接口**

/\*\*

\* 调用远程接口

\*

\* @return

\*/

@RequestMapping("/call")

public User call() {

ServiceInstance serviceInstance = loadBalancer.choose("service-provider");

System.out.println("服务地址：" + serviceInstance.getUri());

System.out.println("服务名称：" + serviceInstance.getServiceId());

User callServiceResult = new RestTemplate().getForObject(serviceInstance.getUri().toString() + "/hello",

User.class);

System.out.println(callServiceResult);

return callServiceResult;

}

* + 1. **使用Feign访问服务端接口**
       1. **设置启动类**

@EnableFeignClients

@EnableDiscoveryClient

@SpringBootApplication

public class SpringBootConsulApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootConsulApplication.class, args);

}

}

* + - 1. **Feign服务类**

@FeignClient(name= "service-provider")

public interface HelloRemoteService {

@RequestMapping(value = "/hello")

public User hello(@RequestParam(value = "id") int id);

}

* + - 1. **测试调用类**

@Autowired

private HelloRemoteService userService;

/\*\*

\* 使用Feign，调用远程接口

\*

\* @return

\*/

@RequestMapping("/FeignCall")

public User FeignCall(@RequestParam(value = "id") int id) {

User callServiceResult = userService.hello(id);

System.out.println(callServiceResult);

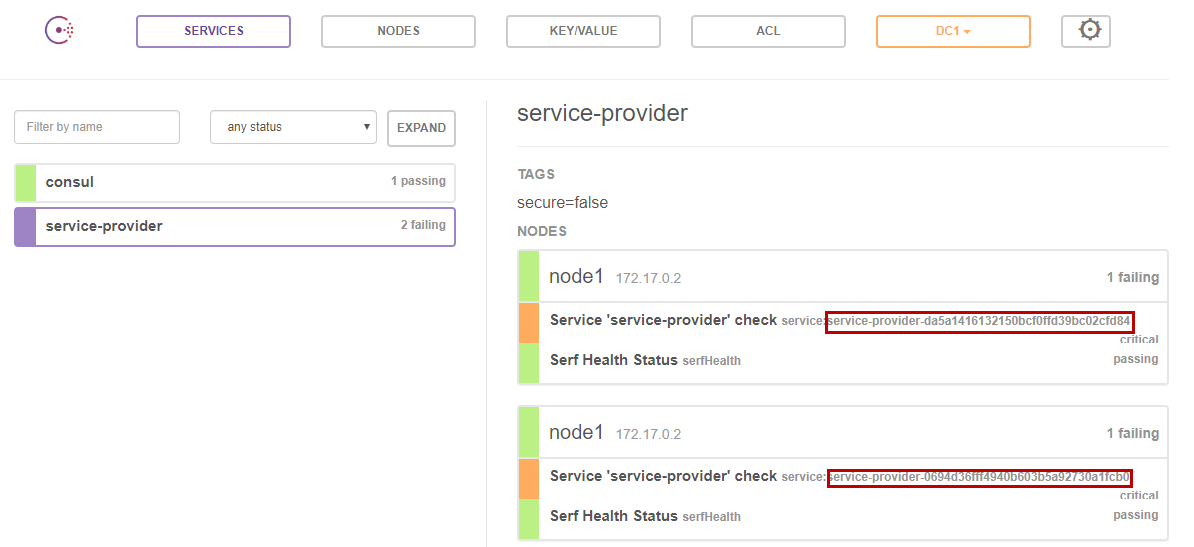
return callServiceResult;

}

* 1. **Consul移除无效的服务**

[http://192.168.56.10:8500/v1/agent/service/deregister/<serviceID](http://192.168.56.10:8500/v1/agent/service/deregister/%3cserviceID)>

serviceID如下截图：



删除无效的节点

<http://192.168.56.10:8500/v1/agent/force-leave/4b36b27317a0>

* 1. **GitHub代码地址**

<https://github.com/KentTain/SpringCloudDemo.git>

1. **声明式调用：Feign**
   1. **Feign调用WebApi**
      1. **配置pom.xml**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

* + 1. **客户端--启动设置：@EnableFeignClients**

@EnableFeignClients

@SpringBootApplication

public class SpringBootConsulApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootConsulApplication.class, args);

}

}

* + 1. **Feign服务类**

@FeignClient(name= "service-provider")

public interface HelloRemoteService {

@RequestMapping(value = "/hello")

public User hello(@RequestParam(value = "id") int id);

}

* + 1. **测试调用类**

@Autowired

private HelloRemoteService userService;

/\*\*

\* 使用Feign，调用远程接口

\* @return

\*/

@RequestMapping("/FeignCall")

public User FeignCall(@RequestParam(value = "id") int id) {

User callServiceResult = userService.hello(id);

System.out.println(callServiceResult);

return callServiceResult;

}

* 1. **Feign添加OAuth2 AuthorizatinHeader**
     1. **实现feign.RequestInterceptor接口**

import feign.RequestInterceptor;

import feign.RequestTemplate;

import lombok.extern.slf4j.Slf4j;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContext;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.oauth2.provider.authentication.OAuth2AuthenticationDetails;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FeignOauth2RequestInterceptor implements RequestInterceptor {

private final String AUTHORIZATION\_HEADER = "Authorization";

private final String BEARER\_TOKEN\_TYPE = "Bearer";

@Override

public void apply(RequestTemplate requestTemplate) {

SecurityContext securityContext = SecurityContextHolder.getContext();

Authentication authentication = securityContext.getAuthentication();

if (authentication != null && authentication.getDetails() instanceof OAuth2AuthenticationDetails) {

OAuth2AuthenticationDetails details = (OAuth2AuthenticationDetails) authentication.getDetails();

requestTemplate.header(AUTHORIZATION\_HEADER, String.format("%s %s", BEARER\_TOKEN\_TYPE, details.getTokenValue()));

}

}

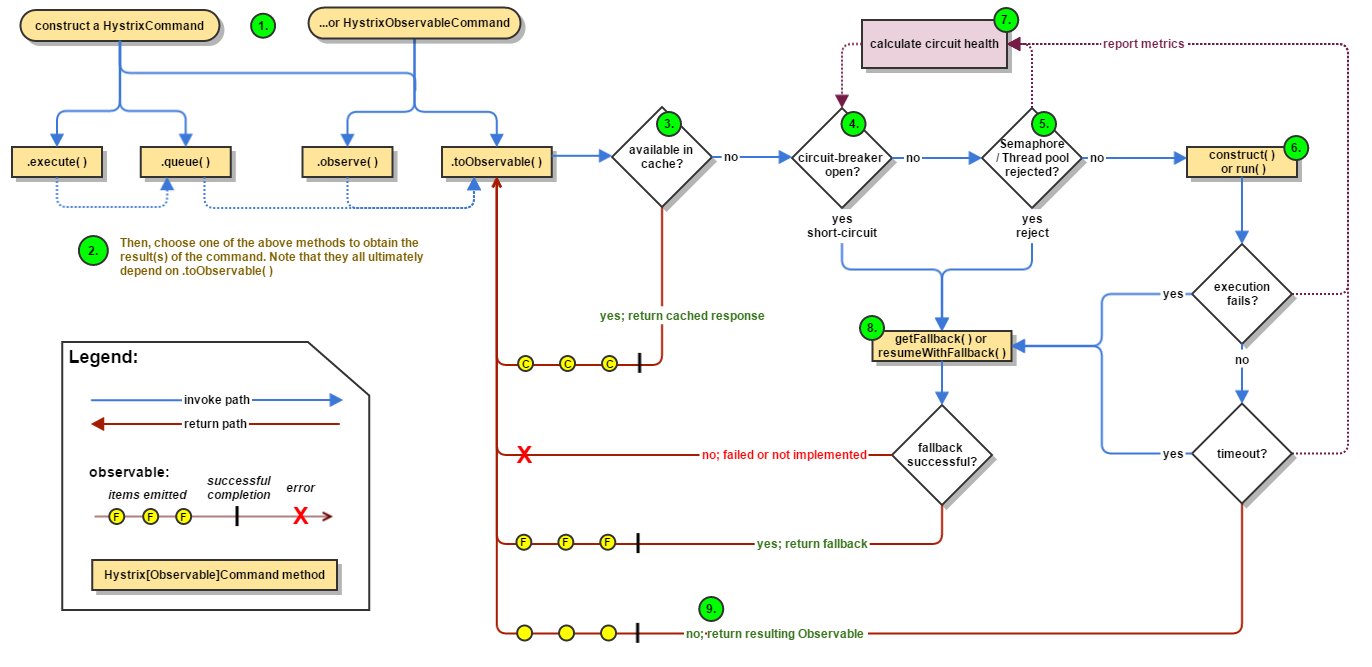
}

* + 1. **调用接口时，Feign自动添加Token**

1. **熔断器：Hystricx**
   1. **Hystrix介绍**

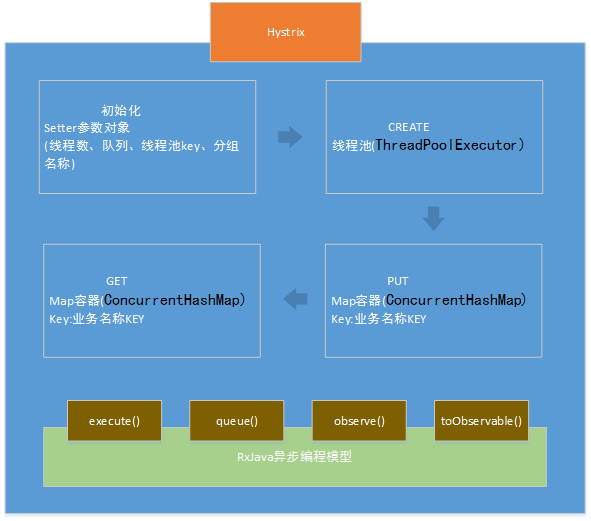
Hystrix的设计原则是什么？

* 资源隔离（线程池隔离和信号量隔离）机制：限制调用分布式服务的资源使用，某一个调用的服务出现问题不会影响其它服务调用。
* 限流机制：限流机制主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。
* 熔断机制：当失败率达到阀值自动触发降级（如因网络故障、超时造成的失败率真高），熔断器触发的快速失败会进行快速恢复。
* 降级机制：超时降级、资源不足时（线程或信号量）降级 、运行异常降级等，降级后可以配合降级接口返回托底数据。
* 缓存支持：提供了请求缓存、请求合并实现
* 通过近实时的统计/监控/报警功能，来提高故障发现的速度
* 通过近实时的属性和配置热修改功能，来提高故障处理和恢复的速度
  1. **Hystrix整体工作流程**



整个流程可以大致归纳为如下几个步骤：

* 创建HystrixCommand或者HystrixObservableCommand对象
* 执行 Command
* 检查请求结果是否被缓存
* 检查是否开启了短路器
* 检查 线程池/队列/semaphore 是否已经满
* 执行 HystrixObservableCommand.construct() or HystrixCommand.run()
* 计算短路健康状况
* 调用fallback降级机制
* 返回依赖请求的真正结果
  1. **Hystrix特性**
     1. **资源隔离**
* 说明：在Hystrix中, 主要通过线程池来实现资源隔离. 通常在使用的时候我们会根据调用的远程服务划分出多个线程池. 例如调用产品服务的Command放入A线程池, 调用账户服务的Command放入B线程池. 这样做的主要优点是运行环境被隔离开了. 这样就算调用服务的代码存在bug或者由于其他原因导致自己所在线程池被耗尽时, 不会对系统的其他服务造成影响. 但是带来的代价就是维护多个线程池会对系统带来额外的性能开销. 如果是对性能有严格要求而且确信自己调用服务的客户端代码不会出问题的话, 可以使用Hystrix的信号模式(Semaphores)来隔离资源



* Command执行方式

execute()：以同步堵塞方式执行 run()。调用 execute() 后，hystrix先创建一个新线程运行run()，接着调用程序要在 execute() 调用处一直堵塞着，直到 run() 运行完成。

queue()：以异步非堵塞方式执行 run() 。调用 queue() 就直接返回一个 Future 对象，同时hystrix创建一个新线程运行 run()，调用程序通过 Future.get() 拿到 run() 的返回结果，而Future.get() 是堵塞执行的。

observe()：立即执行，即事件subscribe()完成注册前执行 run()/construct() 。

第一步是事件注册前，先调用 observe() 自动触发执行 run()/construct()（如果继承的是HystrixCommand，hystrix将创建新线程非堵塞执行run()；如果继承的是HystrixObservableCommand，将以调用程序线程堵塞执行construct())，

第二步是从 observe() 返回后调用程序调用 subscribe() 完成事件注册，如果 run()/construct() 执行成功则触发 onNext() 和 onCompleted() ，如果执行异常则触发 onError() 。

toObservable()：延时执行，即事件subscribe()完成事件注册后执行 run()/construct() 。

第一步是事件注册前，调用 toObservable() 就直接返回一个 Observable<String> 对象，

第二步调用 subscribe() 完成事件注册后自动触发执行 run()/construct()（如果继承的是HystrixCommand，hystrix将创建新线程非堵塞执行 run() ，调用程序不必等待 run() ；如果继承的是HystrixObservableCommand，将以调用程序线程堵塞执行 construct()，调用程序等待construct()执行完才能继续往下走），如果 run()/construct() 执行成功则触发 onNext() 和 onCompleted() ，如果执行异常则触发 onError() 。

备注：execute()和queue()是HystrixCommand中的方法，observe()和toObservable()是HystrixObservableCommand 中的方法。**其中HystrixCommand是用来获取一条数据的；HystrixObservableCommand是用来获取多条数据的**。从底层实现来讲，HystrixCommand其实也是利用Observable实现的（如果我们看Hystrix的源码的话，可以发现里面大量使用了RxJava），虽然HystrixCommand只返回单个的结果，但HystrixCommand的queue方法实际上是调用了toObservable().toBlocking().toFuture()，而execute方法实际上是调用了queue().get()。

* 获取单个产品Command

public class GetProductInfoCommand extends HystrixCommand<ProductInfo>{

private Long productId;

public GetProductInfoCommand(Long productId) {

super(HystrixCommandGroupKey.Factory.asKey("GetProductInfoCommandGroup"));

this.productId=productId;

}

@Override

protected ProductInfo run() throws Exception {

String url = "http://127.0.0.1:8082/getProductInfo?productId="+productId;

String response = HttpClientUtils.sendGetRequest(url);

return JSONObject.parseObject(response,ProductInfo.class);

}

}

//使用

HystrixCommand<ProductInfo> command = new GetProductInfoCommand(productId);

ProductInfo productInfo=command.execute();

* 获取产品列表Command

// 获取产品列表Command

public class GetProductInfosCommand extends HystrixObservableCommand<ProductInfo> {

private String[] productIds;

public GetProductInfosCommand(String[] productIds) {

super(HystrixCommandGroupKey.Factory.asKey("GetProductInfoGroup"));

this.productIds = productIds;

}

@Override

protected Observable<ProductInfo> construct() {

return Observable.create(new Observable.OnSubscribe<ProductInfo>() {

public void call(Subscriber<? super ProductInfo> observer) {

try {

for(String productId : productIds) {

String url = "http://127.0.0.1:8082/getProductInfo?productId=" + productId;

String response = HttpClientUtils.sendGetRequest(url);

ProductInfo productInfo = JSONObject.parseObject(response, ProductInfo.class);

observer.onNext(productInfo);

}

observer.onCompleted();

} catch (Exception e) {

observer.onError(e);

}

}

}).subscribeOn(Schedulers.io());

}

}

//使用

HystrixObservableCommand<ProductInfo> getProductInfosCommand =

new GetProductInfosCommand(productIds.split(","));

Observable<ProductInfo> observable = getProductInfosCommand.observe();

//observable = getProductInfosCommand.toObservable(); // 还没有执行

observable.subscribe(new Observer<ProductInfo>() { // 等到调用subscribe然后才会执行

public void onCompleted() {

System.out.println("获取完了所有的商品数据");

}

public void onError(Throwable e) {

e.printStackTrace();

}

public void onNext(ProductInfo productInfo) {

System.out.println(productInfo);

}

});

* + 1. **限流（通过配置）**

限流在日常生活中很常见，比如节假日你去一个旅游景点，为了不把景点撑爆，管理部门通常会在外面设置拦截，限制景点的进入人数（等有人出来之后，再放新的人进去）。对应到计算机中，比如要搞活动、秒杀等，通常都会限流。在Hystrix中：

* 如果是线程隔离，可以通过线程数+队列大小限制。参数如下：

hystrix.threadpool.default.coreSize

hystrix.threadpool.default.maxQueueSize

hystrix.threadpool.default.queueSizeRejectionThreshold

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds

* 如果是信号量隔离，可以设置最大并发请求数。参数如下：

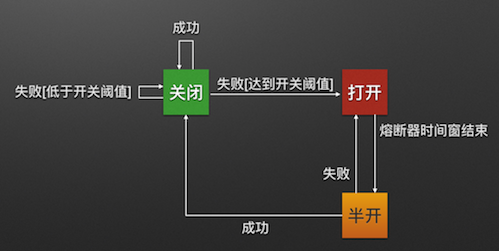
hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests

* + 1. **熔断（CircuitBreaker）**

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

熔断器开关相互转换的逻辑如下图：

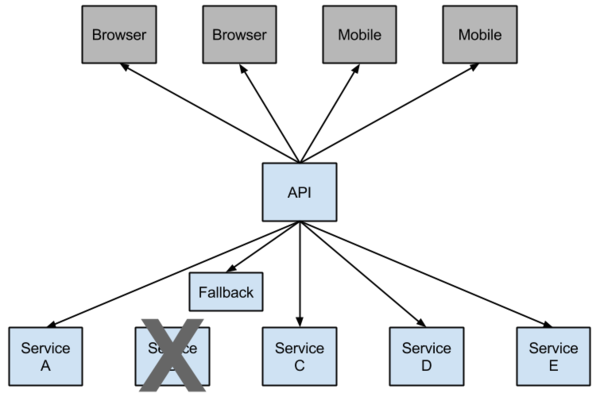


熔断器就是保护服务高可用的最后一道防线。

当Hystrix Command请求后端服务时，在一定时间内（metrics.rollingStats.timeInMilliseconds，默认10s），请求次数超过了最低要求（circuitBreaker.requestVolumeThreshold，默认20次），并且其失败数量超过一定比例(circuitBreaker.errorThresholdPercentage，默认50%)，断路器会切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态一段时间后(circuitBreaker.sleepWindowInMilliseconds，默认5秒), 自动切换到半开路状态(HALF-OPEN). 这时会判断下一次请求的返回情况, 如果请求成功, 断路器切回闭路状态(CLOSED), 否则重新切换到开路状态(OPEN). Hystrix的断路器就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力.

* + 1. **降级（Fallback）**

Fallback相当于是降级操作。所谓降级，就是指在Hystrix执行非核心链路功能失败的情况下，该如何处理，比如返回默认值或者从缓存中取值



触发降级的情况

1、hystrix调用各种接口，或者访问外部依赖（如mysql、redis等等）时，执行方法中抛出了异常。

2、对每个外部依赖，无论是服务接口，中间件，资源隔离，对外部依赖只能用一定量的资源去访问，线程池/信号量，如果资源池已满，则后续的请求将会被 reject，即进行限流。

3、访问外部依赖的时候，访问时间过长，可能就会导致超时，报一个TimeoutException异常，即Timeout机制。

上述三种情况，都是常见的异常情况，对外部依赖的东西访问的时候出现了异常，发送异常事件到断路器中去进行统计。

4、如果断路器发现异常事件的占比达到了一定的比例，直接开启断路器。

上述四种情况，都会去调用fallback降级机制。

**如果要实现回退或者降级处理，代码上需要实现HystrixCommand.getFallback()方法或者是HystrixObservableCommand. HystrixObservableCommand()。**

* + 1. **Hystrix请求缓存（request cache）**

Hystrix支持将一个请求结果缓存起来，在同一个请求上下文中，具有相同key的请求将直接从缓存中取出结果，很适合查询类的接口，可以使用缓存进行优化，减少请求开销，从而跳过真实服务的访问请求。

Hystrix请求结果缓存的作用：

1、在同一个请求上下文中，可以减少使用相同参数请求原始服务的开销。

3、请求缓存在 run() 和 construct() 执行之前生效，所以可以有效减少不必要的线程开销。

要使用Hystrix cache功能：

1、需要构建 RequestContext ，可以在拦截器中使用 HystrixRequestContext.initializeContext() 和 HystrixRequestContext.shutdown() 来初始化 RequestContext 和 关闭RequestContext资源。

2、需要重写 HystrixCommand 或 HystrixObservableCommand 中的 getCacheKey() 方法，指定缓存的 key，开启缓存配置。

* 配置HystrixRequestContextServletFilter

@WebFilter(filterName = "hystrixRequestContextServletFilter",urlPatterns = "/\*",asyncSupported = true)

public class HystrixRequestContextServletFilter implements Filter {

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

HystrixRequestContext context = HystrixRequestContext.initializeContext();

try {

chain.doFilter(request, response);

} finally {

context.shutdown();

}

}

@Override

public void init(FilterConfig filterConfig) throws ServletException {

}

@Override

public void destroy() {

}

}

* 开启缓存功能：继承HystrixCommand或HystrixObservableCommand，覆盖getCacheKey()方法，指定缓存的key，开启缓存配置。

private static final HystrixCommandKey COMMAND\_KEY= HystrixCommandKey.Factory.asKey("GetProductInfoCommand");

@Override

protected String getCacheKey() {

return "product\_info\_"+productId;

}

public static void flushCache(Long productId){

HystrixRequestCache.getInstance(COMMAND\_KEY, HystrixConcurrencyStrategyDefault.getInstance()).clear("product\_info\_"+productId);

}

* + 1. **Hystrix请求合并（request collapser）**
    2. **资料**

<https://me.csdn.net/sun_qiangwei>

* 1. **Feign使用Hystrix**
     1. **服务端--添加超时及异常API**

@RestController

public class HelloController {

private static Logger LOGGER = LoggerFactory.getLogger(HelloController.class);

@RequestMapping("/hello")

public @ResponseBody User hello(@RequestParam(value = "id") int id) {

LOGGER.info("invoking hello endpoint");

return new User(id, "hello consul from provider 1");

}

@RequestMapping("/timeout")

public String timeout() throws InterruptedException {

LOGGER.info("invoking timeout endpoint");

Thread.sleep(10000L);

return "timeout from provider 1";

}

@RequestMapping("/exception")

public String exception() {

LOGGER.info("invoking exception endpoint");

if (System.currentTimeMillis() % 2 == 0) {

throw new RuntimeException("random exception from provider 1");

}

return "exception from provider 1";

}

}

* + 1. **客户端--启动设置**

@EnableFeignClients

@EnableHystrixDashboard

@EnableCircuitBreaker

@SpringBootApplication

public class SpringBootConsulApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootConsulApplication.class, args);

}

}

* + 1. **客户端--Yaml配置**

server:

port: 8504

spring:

application:

name: service-consumer

cloud:

consul:

discovery:

register: false

host: 192.168.56.10

port: 8500

feign:

httpclient:

connection-timeout: 5000

connection-timer-repeat: 5000

enabled: true

max-connections: 200

max-connections-per-route: 50

hystrix:

enabled: true

ribbon:

# 暂不开启熔断机制

hystrix:

enabled: false

# 配置ribbon默认的超时时间

ConnectTimeout: 5000

ReadTimeout: 5000

# 是否开启重试

OkToRetryOnAllOperations: true

# 每个实例重试次数

MaxAutoRetries: 2

# 重试的时候实例切换次数

MaxAutoRetriesNextServer: 3

## hystrix相关配置

## hystrix默认会读取classpath下的config.properties文件，application会覆盖config.properties中的属性

hystrix:

threadpool:

# 指定服务的配置

user-service:

coreSize: 20

maxQueueSize: 200

queueSizeRejectionThreshold: 3

# userThreadPool是UserTimeOutCommand中配置的threadPoolKey

userThreadPool:

coreSize: 20

maxQueueSize: 20

queueSizeRejectionThreshold: 3

# 这是默认的配置

default:

coreSize: 10

maxQueueSize: 200

queueSizeRejectionThreshold: 2

command:

# 指定feign客户端中具体的方法

HelloRemoteService#timeout():

execution:

isolation:

thread:

timeoutInMilliseconds: 5000

userCommandKey:

execution:

isolation:

thread:

timeoutInMilliseconds: 1000

# 这是默认的配置

default:

execution:

timeout:

enabled: true

isolation:

strategy: THREAD

thread:

timeoutInMilliseconds: 15000

interruptOnTimeout: true

interruptOnFutureCancel: false

semaphore:

maxConcurrentRequests: 2

fallback:

enabled: true

isolation:

semaphore:

maxConcurrentRequests: 10

circuitBreaker:

enabled: true

forceOpen: false

forceClosed: false

requestVolumeThreshold: 4

errorThresholdPercentage: 50

sleepWindowInMilliseconds: 10000

metrics:

rollingStats:

timeInMilliseconds: 5000

numBuckets: 10

rollingPercentile:

enabled: true

timeInMilliseconds: 60000

numBuckets: 6

bucketSize: 100

healthSnapshot:

intervalInMilliseconds: 500

* + 1. **客户--消费接口设置**

@Component

@FeignClient(name= "service-provider", fallback = HelloRemoteFallbackService.class)

public interface HelloRemoteService {

@RequestMapping(value = "/hello")

public User hello(@RequestParam(value = "id") int id);

@RequestMapping(value = "/timeout", method = RequestMethod.GET)

public String timeout();

@RequestMapping(value = "/exception", method = RequestMethod.GET)

public String exception();

}

* + 1. **客户端--Fallback设置**

@Component

public class HelloRemoteFallbackService implements HelloRemoteService {

@Override

public User hello(int id)

{

System.out.println("调用服务失败--hello");

return null;

}

@Override

public String timeout() {

System.out.println("调用服务失败--timeout");

return "timeout 降级";

}

@Override

public String exception() {

System.out.println("调用服务失败--exception");

return "exception 降级";

}

}

* + 1. **客户端--相关配置**

# 线程池大小

hystrix.threadpool.default.coreSize=1

# 缓冲区大小， 如果为-1，则不缓冲，直接进行降级 fallback

hystrix.threadpool.default.maxQueueSize=200

# 缓冲区大小超限的阈值，超限就直接降级

hystrix.threadpool.default.queueSizeRejectionThreshold=2

# 执行策略

# 资源隔离模式，默认thread。 还有一种叫信号量

hystrix.command.default.execution.isolation.strategy=THREAD

# 是否打开超时

hystrix.command.default.execution.timeout.enabled=true

# 超时时间，默认1000毫秒

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=15000

# 超时时中断线程

hystrix.command.default.execution.isolation.thread.interruptOnTimeout=true

# 取消时候中断线程

hystrix.command.default.execution.isolation.thread.interruptOnFutureCancel=false

# 信号量模式下，最大并发量

hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests=2

# 降级策略

# 是否开启服务降级

hystrix.command.default.fallback.enabled=true

# fallback执行并发量

hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests=100

# 熔断策略

# 启用/禁用熔断机制

hystrix.command.default.circuitBreaker.enabled=true

# 强制开启熔断

hystrix.command.default.circuitBreaker.forceOpen=false

# 强制关闭熔断

hystrix.command.default.circuitBreaker.forceClosed=false

# 前提条件，一定时间内发起一定数量的请求。 也就是5秒钟内(这个5秒对应下面的滚动窗口长度)至少请求4次，熔断器才发挥起作用。 默认20

hystrix.command.default.circuitBreaker.requestVolumeThreshold=4

# 错误百分比。达到或超过这个百分比，熔断器打开。 比如：5秒内有4个请求，2个请求超时或者失败，就会自动开启熔断

hystrix.command.default.circuitBreaker.errorThresholdPercentage=50

# 10秒后，进入半打开状态（熔断开启，间隔一段时间后，会让一部分的命令去请求服务提供者，如果结果依旧是失败，则又会进入熔断状态，如果成功，就关闭熔断）。 默认5秒

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=10000

# 度量策略

# 5秒为一次统计周期，术语描述：滚动窗口的长度为5秒

hystrix.command.default.metrics.rollingStats.timeInMilliseconds=5000

# 统计周期内 度量桶的数量，必须被timeInMilliseconds整除。作用：

hystrix.command.default.metrics.rollingStats.numBuckets=10

# 是否收集执行时间，并计算各个时间段的百分比

hystrix.command.default.metrics.rollingPercentile.enabled=true

# 设置执行时间统计周期为多久，用来计算百分比

hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds=60000

# 执行时间统计周期内，度量桶的数量

hystrix.command.default.metrics.rollingPercentile.numBuckets=6

# 执行时间统计周期内，每个度量桶最多统计多少条记录。设置为50，有100次请求，则只会统计最近的10次

hystrix.command.default.metrics.rollingPercentile.bucketSize=100

# 数据取样时间间隔

hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds=500

# 设置是否缓存请求，request-scope内缓存

hystrix.command.default.requestCache.enabled=false

# 设置HystrixCommand执行和事件是否打印到HystrixRequestLog中

hystrix.command.default.requestLog.enabled=false

# 限流策略

#如果没有定义HystrixThreadPoolKey，HystrixThreadPoolKey会默认定义为HystrixCommandGroupKey的值

hystrix.threadpool.userGroup.coreSize=1

hystrix.threadpool.userGroup.maxQueueSize=-1

hystrix.threadpool.userGroup.queueSizeRejectionThreshold=800

hystrix.threadpool.userThreadPool.coreSize=1

hystrix.threadpool.userThreadPool.maxQueueSize=-1

hystrix.threadpool.userThreadPool.queueSizeRejectionThreshold=800

hystrix.command.userCommandKey.execution.isolation.thread.timeoutInMilliseconds=5000

* + 1. **资料**

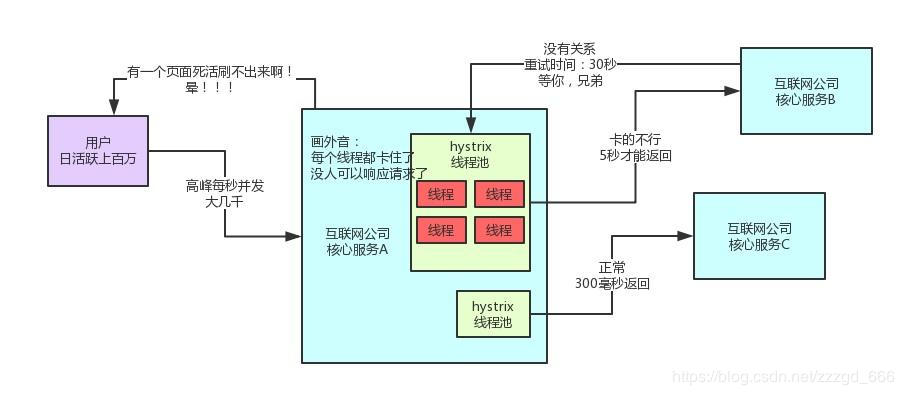
Hystrix属性配置详情：<https://github.com/Netflix/Hystrix/wiki/Configuration>

SpringCloud的限流、熔断和降级：<https://blog.csdn.net/chenxyz707/article/details/80913725>

Feign、Hystrix使用时的超时时间(timeout)设置问题：<https://blog.csdn.net/mxmxz/article/details/84633098>

* 1. **设置TimeOut注意事项**
* 如果hystrix.command.default.execution.timeout.enabled为true,则会有两个执行方法超时的配置,一个就是ribbon的ReadTimeout,一个就是熔断器hystrix的timeoutInMilliseconds, 此时谁的值小谁生效
* 如果hystrix.command.default.execution.timeout.enabled为false,则熔断器不进行超时熔断,而是根据ribbon的ReadTimeout抛出的异常而熔断,也就是取决于ribbon
* ribbon的ConnectTimeout,配置的是请求服务的超时时间,除非服务找不到,或者网络原因,这个时间才会生效
* ribbon还有MaxAutoRetries对当前实例的重试次数,MaxAutoRetriesNextServer对切换实例的重试次数, 如果ribbon的ReadTimeout超时,或者ConnectTimeout连接超时,会进行重试操作
* 由于ribbon的重试机制,通常熔断的超时时间需要配置的比ReadTimeout长,ReadTimeout比ConnectTimeout长,否则还未重试,就熔断了
* 为了确保重试机制的正常运作,理论上（以实际情况为准）建议hystrix的超时时间为:(1 + MaxAutoRetries + MaxAutoRetriesNextServer) \* ReadTimeout
  1. **Hystrix微服务优化实例**

了解了Hystrix的特性和超时效果,再看看下面这个图,服务A调用服务B和服务C,服务C没有太复杂的逻辑处理,300毫秒内就处理返回了,服务B逻辑复杂,Sql语句就长达上百行,经常要卡个5,6秒返回,在大量请求调用到服务B的时候,服务A调用服务B的hystrix线程池已经不堪重负,全部卡住



这里的话,首先考虑的就是服务B的优化,优化SQL,加索引,加缓存, 优化流程,同步改异步,总之缩短响应时间

一个接口，理论的最佳响应速度应该在200ms以内，或者慢点的接口就几百毫秒。

1. 如何设置Hystrix线程池大小，Hystrix线程池大小默认为10

hystrix:

threadpool:

default:

coreSize: 10

每秒请求数 = 1/响应时长(单位s) \* 线程数 = 线程数 / 响应时长(单位s)

即：线程数 = 每秒请求数 \* 响应时长(单位s) + (缓冲线程数)

比如一台服务, 平均每秒大概收到20个请求,每个请求平均响应时长估计在500ms,

线程数 = 20 \* 500 / 1000 = 10

为了应对峰值高并发,加上缓冲线程,比如这里为了好计算设为5,就是 10 + 5 = 15个线程

1. 如何设置超时时间

还拿上面的例子,比如已经配置了总线程是15个,每秒大概20个请求,那么极限情况,每个线程都饱和工作,也就是每个线程一秒内处理的请求为 20 / 15 = ≈ 1.3个 , 那每个请求的最大能接受的时间就是 1000 / 1.3 ≈ 769ms ,往下取小值700ms.

实际情况中,超时时间一般设为比99.5%平均时间略高即可,然后再根据这个时间推算线程池大小

* 1. **其他**

1. **路由网关：Zuul**
   1. **创建Zuul的服务端**
      1. **创建Gateway项目，配置pom.xml：**

pom.xml如下

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-consul-discovery</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

* + 1. **配置Zuul服务端**

server:

max-http-header-size: 10000000

port: 8501

spring:

application:

name: service-gateway

cloud:

consul:

discovery:

preferIpAddress: true # the health check get ' no such host 'error

health-check-path: /actuator/health # 健康健康路径，也可以自己写

health-check-interval: 10s # 检测轮训时间 1m 代码1分钟

hostname: ${spring.application.name}

service-name: ${spring.application.name}

#instance-id: consul-client00 实例ID，唯一值

instance-id: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

host: 192.168.56.10

port: 8500

zuul:

host:

connect-timeout-millis: 10000

socket-timeout-millis: 60000

routes:

helloService:

path: /proxy/\*\*

serviceId: service-provider

sensitiveHeaders: "\*"

ratelimit:

key-prefix: ratelimit-api

# 启动限流服务

enabled: true

behind-proxy: true

default-policy:

# 请求数量

limit: 100

# 请求总时间

quota: 20

# 统计窗口刷新时间

refresh-interval: 60

# 限流类型

type: url

ignored-services: '\*'

add-proxy-headers: true

retryable: true

sensitive-headers:

* + 1. **设置启动类：@EnableZuulProxy**

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

public class SpringBootConsulApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootConsulApplication.class, args);

}

}

* + 1. **创建相关服务的RestController**

@RestController

public class HelloController {

@RequestMapping("/hello")

public @ResponseBody User hello() {

return new User(1, "hello consul");

}

}

1. **其他**