1.微服务

微服务的特点:

- 单一职责:微服务中每一个服务都对应唯一的业务能力,做到单一职责
- 微:微服务的服务拆分粒度很小
- 面向服务:面向服务是说每个服务都要对外暴露Rest风格服务接口API.并不关心服务的技术实现,做到与平台和语言无关.也不限定用什么技术实现,只要提供Rest的接口即可.
- 自治:自治是说服务间互相独立,互不干扰.

2.服务调用方式

2.1.RPC和HTTP

常见的远程调用方式有以下2种:

- RPC:(Remote Produce Call)远程过程调用,类似的还有RMI(Remote Method Invoke).自定义数据格式,基于原生TCP通信,速度快,效率高.早期的webservice,现在热门的dubbo.都是RPC的典型代表
- Http:Http是一种网络传输协议,基于TCP,规定了数据传输的格式.现在客户端浏览器与服务端通信的方式基本都是采用http协议.也可以用来进行远程服务调用.缺点就是消息封装臃肿.优势是对服务的提供和调用方没有任何技术限定,自由灵活,更符合微服务理念

2.2.HTTP客户端工具

Spring提供了一个**RestTemplate**模板工具类,对基于Http的客户端进行了封装,并且实现了对象与json的序列化和反序列化.非常方便.RestTemplate并没有限定Http的客户端类型,而是进行了抽象.

首先在项目中注入一个RestTemplate对象,可以在启动类位置注册,在服务的消费者端直接`@Autowired`并使用即可.通过RestTemplate的`getForObject()`方法,传递url地址及实体类的字节码,RestTemplate会自动发起请求,接收响应,并且帮我们对响应结果进行反序列化.

目前RestTemplate提供了三种重载方法.

```java
@Override
@Nullable
public <T> T getForObject(String url, Class<T> responseType, Object... uriVariables) throws RestClientException {
    RequestCallback requestCallback = acceptHeaderRequestCallback(responseType);
    HttpMessageConverterExtractor<T> responseExtractor =
            new HttpMessageConverterExtractor<>(responseType, getMessageConverters(), logger);
    return execute(url, HttpMethod.GET, requestCallback, responseExtractor, uriVariables);
}
```

```java
@Override
@Nullable
public <T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables) throws RestClientExcepti
    RequestCallback requestCallback = acceptHeaderRequestCallback(responseType);
    HttpMessageConverterExtractor<T> responseExtractor =
            new HttpMessageConverterExtractor<>(responseType, getMessageConverters(), logger);
    return execute(url, HttpMethod.GET, requestCallback, responseExtractor, uriVariables);
}
```

```java
@Override
@Nullable
public <T> T getForObject(URI url, Class<T> responseType) throws RestClientException {
    RequestCallback requestCallback = acceptHeaderRequestCallback(responseType);
    HttpMessageConverterExtractor<T> responseExtractor =
            new HttpMessageConverterExtractor<>(responseType, getMessageConverters(), logger);
    return execute(url, HttpMethod.GET, requestCallback, responseExtractor);
}
```

3.Eureka

3.1.CAP原则

CAP原则又称CAP定理,指的是在一个分布式系统中,Consistency(一致性),Availability(可用性),Partition tolerance(分区容错性),三者不可兼得.

Eureka遵循的的就是AP原则.

3.2.Eureka对比Zookeeper

作为服务注册中心,Eureka比Zookeeper好在哪里.

注明的CAP理论指出,一个分布式系统不可能同时满足C(一致性),A(可用性)和P(分区容错性).由于分区容错性P在是在分布式系统中必须要保证的,因此我们只能在A和C之间进行权衡.

因此:**Zookeeper保证的是CP,Eureka是AP**

3.2.1.Zookeeper保证CP

在Zookeeper中会出现这样一种情况,当master节点因为网络问题故障与其他节点失去联系时候,剩余节点会重新进行leader选举.问题在于,选举leader的时间太长,30-120s,且选举期间整个zookeeper集群都是不可用的.这就导致在选择具期间注册服务瘫痪.在云部署的环境下,因网络问题使得zookeeper集群失去master节点是较大概率会发生的事.虽然服务能够最终恢复,但是漫长的选举时间导致的注册长期不可用是不能容忍的.

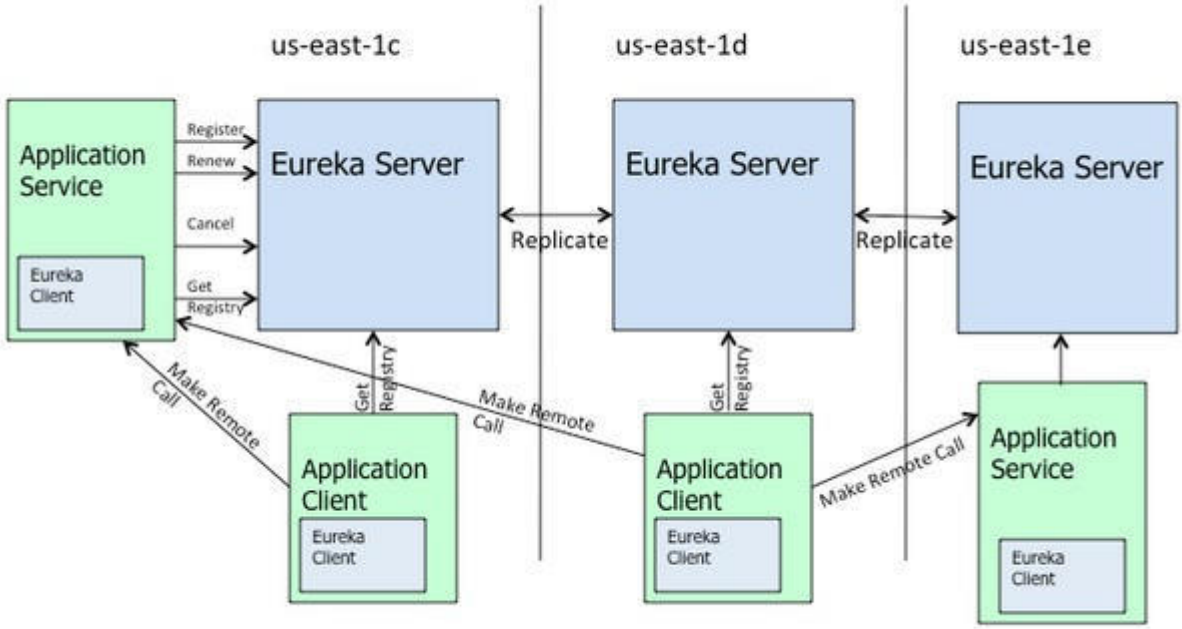
3.2.2.Eureka保证AP

Eureka各个节点都是平等的,几个节点挂掉不会影响正常节点的工作,剩余节点依然可以提供注册和查询服务.而Eureka的客户端在向某个Eureka注册或如果发现连接失败,则会自动切换至其他节点,只要一台Eureka还在,就能保证注册服务可用(保证可用性),只不过查到的信息可能不是最新的(不保证强一致性).除此之外,**Eureka还有一种自我保护机制.如果在15分钟内超过85%的节点都没有正常的心跳,那么Eureka就认为客户端与注册中心出现了网络故障**,此时会出现以下几种情况.

1. Eureka不再从注册列表中移除因为长时间没有收到心跳而应该过期的服务
2. Eureka仍然能够接受新服务的注册和查询请求,但是不会被同步到其他节点上(即保证当前节点依然可用).
3. 当网络稳定时,当前实例新的注册信息会被同步到其它节点中.

   因此,Eureka可以很好的应对因网络故障导致部分节点失去联系的情况,而不会像Zookeeper那样使整个注册服务瘫痪.

3.3.Eureka原理



- 处于不同节点的eureka通过replicate(复制)进行数据同步.
- Application Sservice为服务提供者
- Application Client为服务消费者
- Make Remote Call完成一次服务调用

当服务启动后向Eureka注册,Eureka Server会将注册信息向其他Eureka Server进行同步,当服务消费者要调用服务提供者,则向服务注册中心获取服务提供者地址,然后会将服务提供者地址缓存在本地,下次再调用时,则直接从本地缓存中取,完成一次调用.

当服务注册中新Eureka Server检测到服务者因为宕机,网络原因不可用时,则在服务注册中心将服务置为DOWN状态,并把当前服务提供者向订阅者发布,订阅过的服务消费者更新本地缓存.

服务提供者在启动后,周期性(默认30s)向Eureka Server发送心跳,以证明当前服务是可用状态.Eureka Server在一定时间(默认90s)未收到客户端的心跳,则认为服务宕机,注销该实例.

3.4.Eureka注册中心程序

父pom文件(确定SpringCloud及SpringBoot的版本)

```
    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
        <relativePath/>
    </parent>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <spring-cloud.version>Greenwich.RELEASE</spring-cloud.version>
        <mapper.starter.version>2.1.5</mapper.starter.version>
        <mysql.version>5.1.47</mysql.version>
        <pageHelper.starter.version>1.2.5</pageHelper.starter.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <!-- springCloud -->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!-- 通用Mapper -->
            <dependency>
                <groupId>tk.mybatis</groupId>
                <artifactId>mapper-spring-boot-starter</artifactId>
                <version>${mapper.starter.version}</version>
            </dependency>
            <!-- mysql驱动 -->
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>${mysql.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
```

```
    </build>
```

子POM文件,添加eureka的依赖

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>
    </dependencies>
```

Eureka的yml文件:单机版的eureka的service-url可以直接写http://127.0.0.1:10086/eureka.也可以写http://${eureka.instance.hostname}:${server.port}/eureka.如果想要搭建eureka集群环境,那么必须要修改hosts文件,不然eureka会认为你在一台主机上搭建集群.如果在一台服务器上搭建多个节点,通过localhost进行访问.是不支持集群的.

在配置完hosts文件之后,我们需要刷新DNS,让host文件生效.

```
ipconfig/flushdns
```

```
spring:
  profiles:
    active: singleTon
  application:
    name: eureka-server # application name是体现在注册的instance名
# eureka:
  # server:
    # enable-self-preservation: false # 关闭自我保护模式（缺省为打开）
---
# 单机版
server:
  port: 10086
spring:
  profiles: singleTon

eureka:
  instance:
    hostname: host-eureka-server01
  client:
    register-with-eureka: false #单机版是不能够自己注册自己
    fetch-registry: false #单机版不能够自己拉取自己
    service-url:
      dafaultZone: http://${eureka.instance.hostname}:${server.port}/eureka #这是eureka服务
地址
---
# 集群节点1
spring:
  profiles: replicas01
eureka:
  instance:
    hostname: eureka01 # 当前节点域名名称,体现在DS-replicas中的名字
```

```yaml
  client:
    service-url:
      defaultZone: http://eureka02:10088/eureka, http://eureka03:10089/eureka
server:
  port: 10087
---
# 集群节点2
spring:
  profiles: replicas02
eureka:
  instance:
    hostname: eureka02
  client:
    service-url:
      defaultZone: http://eureka01:10087/eureka, http://eureka03:10089/eureka
server:
  port: 10088
---
# 集群节点3
spring:
  profiles: replicas03
eureka:
  instance:
    hostname: eureka03
  client:
    service-url:
      defaultZone: http://eureka01:10087/eureka, http://eureka02:10088/eureka
server:
  port: 10089
```

多个Eureka Server之间会互相注册为服务,当服务提供者注册到Eureka Server集群中的某个节点时,该节点会把服务的信息同步给集群中的每个节点,从而实现高可用集群.因此,无论客户端访问到Eureka Server集群中的任意一个节点,都可以获取到完整的服务列表信息.

之后再主启动类上添加@EnableEurekaServer,并启动该微服务.

3.5.服务生产者程序

3.5.1:服务注册

```xml
        <!--eureka客户端-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
```

```yaml
server:
  port: 8081
spring:
  application:
    name: item-service
```

```
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/leyou?characterEncoding=utf-8
    username: root
    password: root

eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true
    ip-address: 127.0.0.1

mybatis:
  mapper-locations: mappers/*.xml #指定mybatis的mapper文件所放置位置
  type-aliases-package: com.leyou.item.pojo #实体类所在的包
  configuration:
    map-underscore-to-camel-case: true #开启驼峰命名
logging:
  level:
    com.leyou: debug
```

添加一个spring.application.name属性来指定应用名称,将来会作为服务的id进行使用.

```
@SpringBootApplication
@EnableEurekaClient
@MapperScan("com.leyou.item.mapper")
public class LyItemApplication {
    public static void main(String[] args) {
        SpringApplication.run(LyItemApplication.class,args);
    }
}
```

@EnableEurekaClient也可以使用@EnableDiscoveryClient,不过@EnableDiscoveryClient的使用范围更广,而@EnableEurekaClient只使用于Eureka注册中心.

服务提供者在启动时,会检测配置属性中的:`eureka.client.register-with-erueka=true`参数是否为true,默认事实上就是true,如果值确实为true,则会向EurekaServer发起一个Rest请求,并携带自己的元数据.Eureka Server会把这些信息保存到一个双层Map结构Map结构中.

- 第一层Map的Key就是服务id,一般是配置中的`spring.application.name`属性
- 第二层Map的Key是服务的实例id.一般host+serviceId+port,例如:`localhost:user-service:8081`
- 值则是服务的实例对象,也就是所一个服务,可以同时启动多个不同实例,形成集群.

user-service默认注册时使用的是主机名,如果我们想用ip进行注册,可以在user-service的application.yml添加配置:

```
eureka:
  instance:
    ip-address: 127.0.0.1 # ip地址
    prefer-ip-address: true # 更倾向于使用ip，而不是host名
    instance-id: ${eureka.instance.ip-address}:${server.port} # 自定义实例的id
```

3.5.2.服务续约

在注册服务完成以后,服务提供者维持一个心跳(定时向EurekaServer发起Rest请求),告诉EurekaServer"我还活着".这个我们称为服务的续约(renewal).

```
eureka:
  instance:
        lease-renewal-interval-in-seconds: 30
    lease-expiration-duration-in-seconds: 90
```

- lease-renewal-interval-in-seconds: 服务续约(renew)的间隔,默认为30s
- lease-expiration-duration-in-seconds:服务失效时间,默认值90s

默认情况下每个30s服务会向注册中心发送一次心跳,证明自己还活着.如果超过90s没有发送心跳.EurekaServer就会认为该服务宕机,会从服务列表中移除,这两个值在生产环境不要修改,默认即可.

3.5.3.获取服务列表

当服务消费者启动时,会检测`eureka.client.fetch-registry=true`参数的值,如果为true,则会从Eureka Server服务的列表只读备份,然后缓存在本地.并且每隔30s会重新获取并更新数据,我们可以通过下面的参数来修改.

```
eureka:
  client:
    registry-fetch-interval-seconds: 30
```

3.6.服务消费者程序

服务的消费者同样也需要引入eureka-client的坐标

```
server:
  port: 19001
spring:
  application:
    name: item-consumer #应用名称
eureka:
  client:
    service-url:
      defaultZone: http://localhost:10086/eureka
    # register-with-eureka: false #消费者并不需要注册进eureka,当然也可以注册金eureka
```

在编写消费者的时候,我们可以直接通过RestTemplate去调用user-service服务

这个时候会抛出异常:java.net.UnknownHostException: USER-SERVICE,想要通过服务名称对服务进行调用的话,我们需要对服务的消费端实现负载均衡.才能通过user-service进行对服务的调用.

```
    private final static String PREFIX_URL = "http://USER-SERVICE/user";

@Autowired
private RestTemplate restTemplate;

@GetMapping(path = "/findAll",name = "查找所有")
public String findAll() {
    List userList = restTemplate.getForObject(PREFIX_URL + "/findAll", List.class);
    return userList.toString();
}
```

3.7.服务下线,失效剔除和自我保护

3.7.1.服务下线

当服务进行正常关闭操作时候,它会触发一个服务下线的REST请求给Eureka Server,告诉服务注册中性:"我要下线了".服务中心接受到请求之后,将该服务置为下线状态.

3.7.2.失效剔除

有时我们的服务可能由于种种原因使服务不能正常工作,而服务注册中心并未受到"服务下线"的请求,相对于服务提供者的"服务续约"操作,服务注册中心在启动时会创建一个定时任务,默认每隔一端时间(默认60s)将当前清单中超时(默认为90s)没有续约的服务剔除,这个操作成为失效剔除.

我们通过`eureka.server.eviction-interval-timer-in-ms`参数对其进行修改,单位时毫秒.

3.7.3.自我保护

我们关停一个服务.就会在Eureka面板看到

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

这是触发了Eureka的自我保护机制,当服务未按时进行心跳续约时,Eureka会统计服务实例最新15分钟心跳续约的比例是否低于85%,在生产环境下,因为网络延迟等原因,心跳失败实例的比例很有可能超标.但时此时就把服务剔除列表并不妥当,因为服务可能没有宕机.Eureka在这段时间内不会剔除任何服务实例,知道网络恢复正常,生产环境下很有效,保证了大多数服务依然可用,不过也有可能获取到失败的服务实例,因此服务调用者必须做好服务的失败容错.

```
eureka:
  server:
    enable-self-preservation: false # 关闭自我保护模式（缺省为打开）
```

4.Ribbon

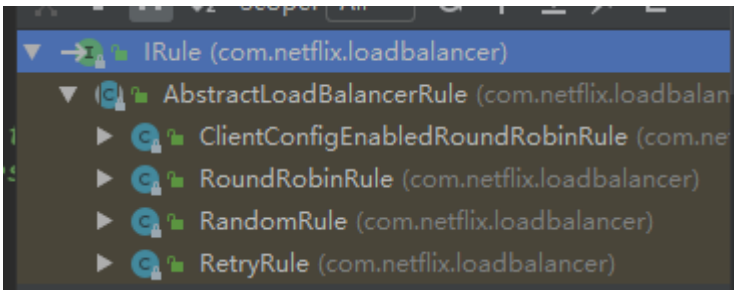
在Eureka中已经帮我们集成了负载均衡组件Ribbon,所以我们无需引入新的依赖,直接修改代码即可.

```
    @Bean
@LoadBalanced
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

这个时候,我们就可以直接通过微服务的service-id去调用服务了.对于集群环境默认使用的就是轮询策略.

@LoadBalance这个注解源码中我们可以看到该注解会把RestTemplate配置使用一个负载均衡的客户端.

```
package org.springframework.cloud.client.loadbalancer;

import ...

/**
 * Annotation to mark a RestTemplate bean to be configured to use a LoadBalancerClient.
 * @author Spencer Gibb
 */
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Qualifier
public @interface LoadBalanced {
}
```



按`Ctrl+H`我们可以知道IRule该接口下的所有的实现类,这些类都是相应的负债均衡的执行规则,默认是采用的RoundRobinRule,即轮询.如果我们想要使用别的负载均衡规则,我们可以直接往Spring容器中去添加该组件即可.

```
@Bean
@LoadBalanced
public RestTemplate restTemplate(){
    return new RestTemplate();
}

@Bean
public IRule randomRule(){
    return new RandomRule();
}
```

Ribbon摸了拟采用的是懒加载,即第一次访问时才会去创建负载均衡客户端,如果需要采用饥饿加载,即项目启动即创建,可以这样配置.

```
ribbon:
  eager-load:
    enabled: true #项目启动立即加载远程调用对象
    clients: user-service #指定服务名称
```

5.Hystrix

服务器支持的线程和并发数有限,请求一直阻塞,会导致服务器资源耗尽,从而导致其它服务都不可用,形成雪崩效应.

5.1.服务降级

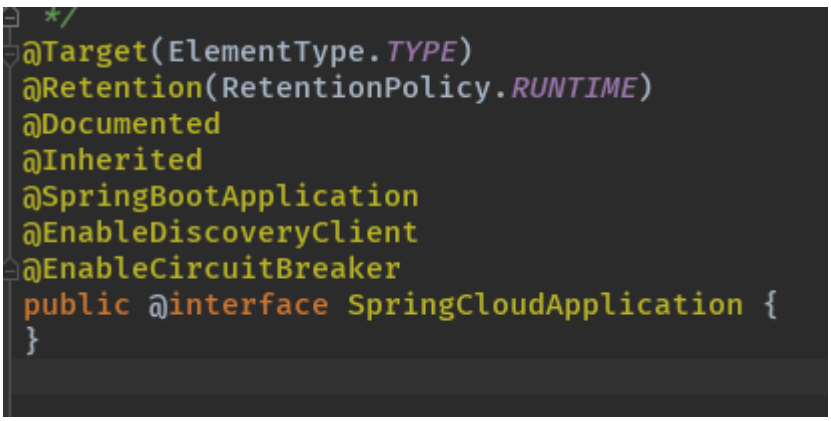
Hystrix为每个依赖服务调用分配一个小的线程池,如果线程池已满调用将被立即拒绝,默认不采用排队,加速失败判定时间.用户的请求将不再直接访问服务,而是通过线程池中的空闲线程来访问服务,如果**线程池已满**或者**请求超时**,则会进行**降级处理.**

服务降级:优先保证核心服务,而非核心服务不可用或弱可用.

1.首先是添加依赖:

```
            <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    </dependency>
```

2.在启动类上添加注解:@EnableCircuitBreaker或者使用@EnableHystrix,或者在启动类上添加@SpringCloudApplication的注解,@SpringCloudApplication注解中组合了@EnableCiruitBreaker注解.



3.最后是编写降级逻辑,我们通过HystixCommond来完成.

```
        @HystrixCommand(fallbackMethod = "fallbackMethod01")
    @GetMapping(path = "/findAll",name = "查找所有")
    public String findAll() {
        List userList = restTemplate.getForObject(PREFIX_URL + "/findAll", List.class);
        return userList.toString();
    }

    public String fallbackMethod01(){
        return "对不起,网络繁忙";
    }
```

要注意,因为降级逻辑方法必须与正常逻辑方法保证:**相同的参数和返回值声明**.是失败逻辑中返回User对象没有太大意义.一般会返回友好提示.

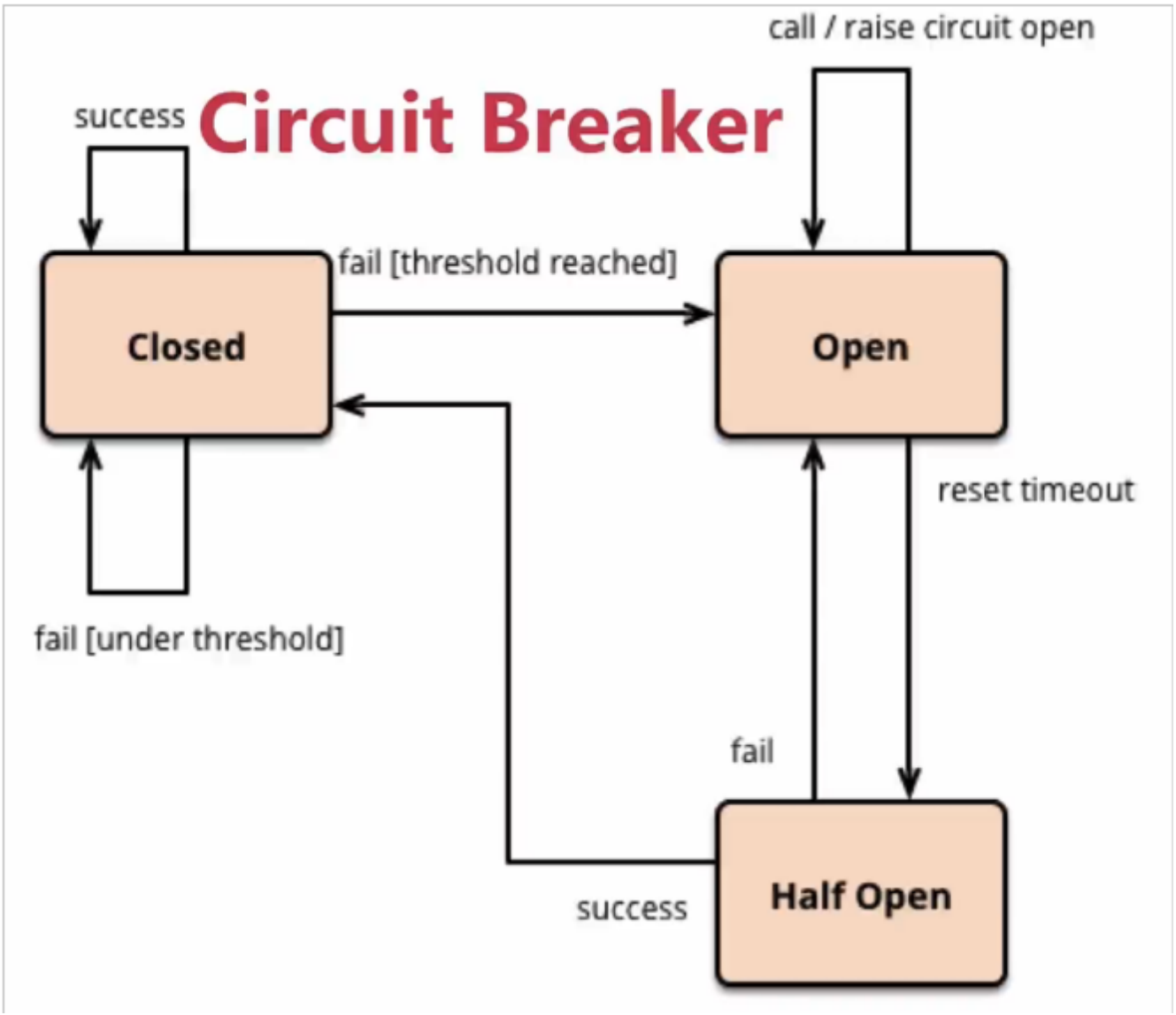
把fallback写在了某个业务方法上,如果这样的方法很多,那就要写很多,所以我们可以把Fallback配置加载类上,实现默认fallback.

```
@RestController
@RequestMapping("/user/consumer")
@DefaultProperties(defaultFallback = "defaultFallBack")
public class UserServiceConsumer {
    private final static String PREFIX_URL = "http://USER-SERVICE/user";

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private UserClient userClient;

    @HystrixCommand
    @GetMapping(path = "/findAll", name = "查找所有")
    public String findAll() {
        List userList = restTemplate.getForObject(PREFIX_URL + "/findAll", List.class);
        return userList.toString();
    }

    public String defaultFallBack() {
        return "默认提示:网络繁忙";
    }
}
```

请求在超过1s后都会返回错误信息,这是因为Hystix的默认超时时长为1,我们可以通过配置修改这个值.

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 1500
```

5.2.服务熔断



状态机有3个状态:

- Closed:关闭状态(熔断器关闭),所有请求都能正常访问
- Open:打开状态(熔断器打开),所有请求都会被降级.Hystrix会对请求计数,当一定时间内失败请求百分比达到阀值,则触发熔断.默认失败比例的阀值是50%,请求次数最少不低于20次.
- Half Open:半开状态,open状态不是永久的,打开后会进入休眠时间(默认5s).随后熔断器会自动进入半开状态,此时会释放1次请求通过,若这个请求时健康的,则会关闭断路器,否则继续保持打开,再进行5秒休眠计时.

```
circuitBreaker:
  requestVolumeThreshold: 10
  sleepWindowInMilliseconds: 10000
  errorThresholdPercentage: 50
```

- requestVolumeThreshold：触发熔断的最小请求次数，默认20
- errorThresholdPercentage：触发熔断的失败请求最小占比，默认50%
- sleepWindowInMilliseconds：休眠时长，默认是5000毫秒

6.Feign

1.导入依赖

```
            <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
```

2.编写Feign接口:

- 首先这是一个接口,Feign会通过动态代理,帮我们生成实现类
- @FeignClient,声明这是一个Feign客户端,同时通过value属性指定服务名称
- 接口中的定义方法,完全采用SrpingMVC的注解,Feign会根据注解帮我们生成URL,并访问获取结果改造原来的调用逻辑

```
@FeignClient(value = "USER-SERVICE",fallback = UserClientFallback.class)
@RequestMapping("/user")
public interface UserClient {
        //这是服务的提供者暴露的RestAPI
    @GetMapping(path = "/find/{id}")
    User findById(@PathVariable("id") String id);
}
```

3.使用Feign接口

```
@Autowired
private UserClient userClient;

@RequestMapping(path = "/query/{id}")
public String findByIdOnfeign(@PathVariable("id") String id){
    return userClient.findById(id).toString();
}
```

4.开启Feign功能

```
@SpringCloudApplication
@EnableFeignClients
public class UserServiceConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceConsumerApplication.class,args);
    }
}
```

6.1.Ribbon的支持

Feign中本身已经集成了Ribbon依赖和自动配置:我们不需要再注册RestTemplate对象

Fegin内置的ribbon默认设置了请求超时时长,默认是1000ms,我们可以通过手动配置来修改这个超时时长.

```
ribbon:
  ReadTimeout: 2000 # 读取超时时长
  ConnectTimeout: 1000 # 建立链接的超时时长
```

因为Ribbon内部有重试机制,一旦超时,会自动重新发起请求,如果不希望重试,可以添加配置

```
ribbon:
  ConnectTimeout: 500 # 连接超时时长
  ReadTimeout: 2000 # 数据通信超时时长
  MaxAutoRetries: 0 # 当前服务器的重试次数
  MaxAutoRetriesNextServer: 1 # 重试多少次服务
  OkToRetryOnAllOperations: false # 是否对所有的请求方式都重试
```

6.2.Hystix的支持

Feign默认对Hystrix也有集成.只不过我们需要下面的参数来开启

```
feign:
  hystrix:
    enabled: true
```

1.首先,我们需要定义一个类,实现刚才编写的Client接口,作为fallback的处理类

```
@Component
@RequestMapping("fallback/")
//这个可以防止容器中有与父类重复的requestMapping
public class UserClientFallback implements UserClient {
    @Override
    public User findById(String id) {
        User user = new User();
        user.setName("用户查询异常");
        return user;
    }
}
```

2.在client接口中指定刚才编写的实现类

```
@FeignClient(value = "USER-SERVICE",fallback = UserClientFallback.class)
@RequestMapping("/user")
public interface UserClient {
    @GetMapping(path = "/find/{id}")
    User findById(@PathVariable("id") String id);
}
```

6.3.请求压缩

SpringCloud Feign支持对请求和响应进行GZIP压缩,以减少通信过程中的性能损耗

```
feign:
  compression:
    request:
      enabled: true # 开启请求压缩
    response:
      enabled: true # 开启响应压缩
```

同时,我们也可以对请求的数据类型,以及触发压缩的大小下限进行设置

```
feign:
  compression:
    request:
      enabled: true # 开启请求压缩
      mime-types: text/html,application/xml,application/json # 设置压缩的数据类型
      min-request-size: 2048 # 设置触发压缩的大小下限
```

6.4.日志级别

之前我们都是通过`logging.level.xx=debug`来设置日志级别,然而这个对Feign客户端而言不会产生效果,因为`@FeignClient`注解修改的客户端再被代理时候,都会创建一个新的Feign.Logger实例.我们需要额外指定这个日志的级别才可以.

1.设置yml的配置文件的日志级别都为debug

```
logging:
  level:
    org.fechin: debug
```

2.编写配置类,定义日志级别

```
@Configuration
public class FeignConfig {
    @Bean
    Logger.Level feignLoggerLevel(){
        return Logger.Level.BASIC;
    }
}
```

这里指定的Level级别是BASIC,Feign支持4种级别

- NONE:不记录任何日志信息,这是默认值
- BASIC:仅仅记录请求方法,URL以及响应状态码和执行时间
- HEADERS:在BASIC的基础上,额外记录了请求和响应的头信息
- FULL:记录所有请求和响应的明细,包括头信息,请求体,元数据.

3.在FeignClient中指定配置类

```
@FeignClient(value = "USER-SERVICE",fallback = UserClientFallback.class,
        configuration = FeignConfig.class)
@RequestMapping("/user")
public interface UserClient {

    @GetMapping(path = "/find/{id}")
    User findById(@PathVariable("id") String id);
}
```

7.Zuul

Zuul是Netflix的微服务网关,Zuul的核心就是一系列的过滤器.

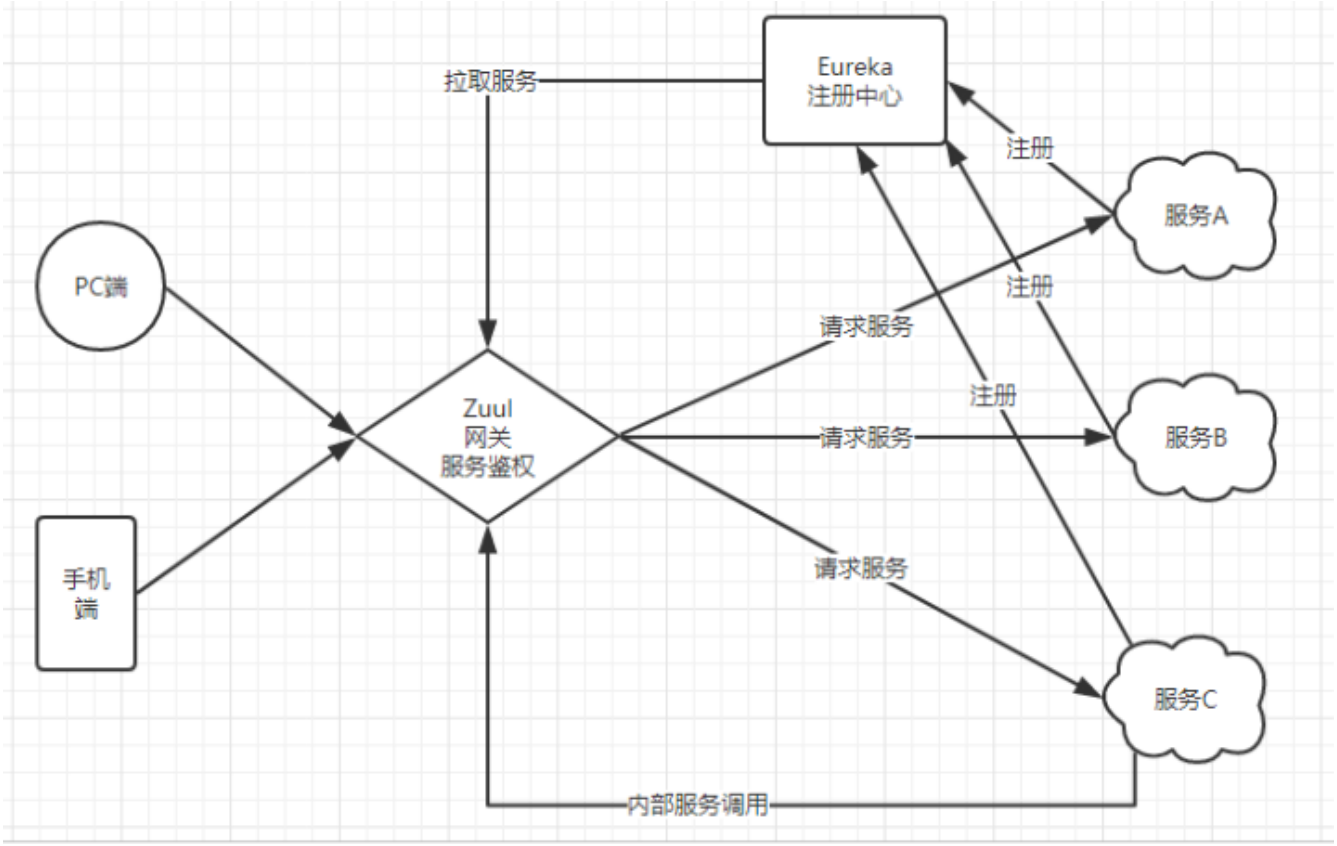
Zuul 是 Netflix 开源的微服务网关，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能。

- 身份认证与安全：识别每个资源的验证要求，并拒绝那些与要求不符的请求。
- 审查与监控：在边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
- 动态路由：动态地将请求路由到不同的后端集群。
- 压力测试：逐渐增加指向集群的流量，以了解性能。
- 负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求。
- 静态响应处理：在边缘位置直接建立部分响应，从而避免其转发到内部集群。
- 多区域弹性：跨越 AWS Region 进行请求路由，旨在实现 ELB（Elastic Load Balancing）使用的多样化，以及让系统的边缘更贴近系统的使用者。

Spring Cloud 对 Zuul 进行了整合与增强。目前，Zuul 使用的默认 HTTP 客户端是 Apache HTTP Client，也可以使用 RestClient 或者`okhttp3.OkHttpClient`。如果想要使用 RestClient，可以设置`ribbon.restclient.enabled=true`;想要使用`okhttp3.OkHttpClient`,可以设置 `ribbon.okhttp.enabled=true`。

7.1.动态路由

不管是来自于客户端的请求,还是服务内部的调用,一切对方服务的请求都会经过Zuul这个网关,然后再由网关来实现鉴权,动态路由等等操作.Zuul就是我们服务的同一入口.



1.添加Zuul依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

2.编写启动类

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class ZuulApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulApplication.class,args);
    }
}
```

3.编写yml配置

```
server:
  port: 10010
spring:
  application:
    name: api-gateway

eureka:
  client:
```

```
    service-url:
      defaultZone: http://eureka01:10086/eureka
zuul:
  prefix: /api
  routes: #路由映射:请求url与目标微服务
    user-service:
      path: /u/**
      serviceId: user-service
    user-consumer:
      path: /c/**
      serviceId: user-consumer
```

4.上述的关于user-service的配置可以简化为一条

```
zuul:
  routes:
    user-service: /user-service/** # 这里是映射路径
```

5.默认的路由规则:默认情况下:一切服务的映射路径就是服务名本身.

例如服务名为：`user-service`，则默认的映射路径就是：`/user-service/**`

如果想要禁用某个路由规则,可以这样:

```
zuul:
  ignored-services:
    - user-service
    - user-consumer
```

6.我们可以通过zuul.prefix=/api来指定路由前缀,这样在发起请求时,路径就要以/api开头.

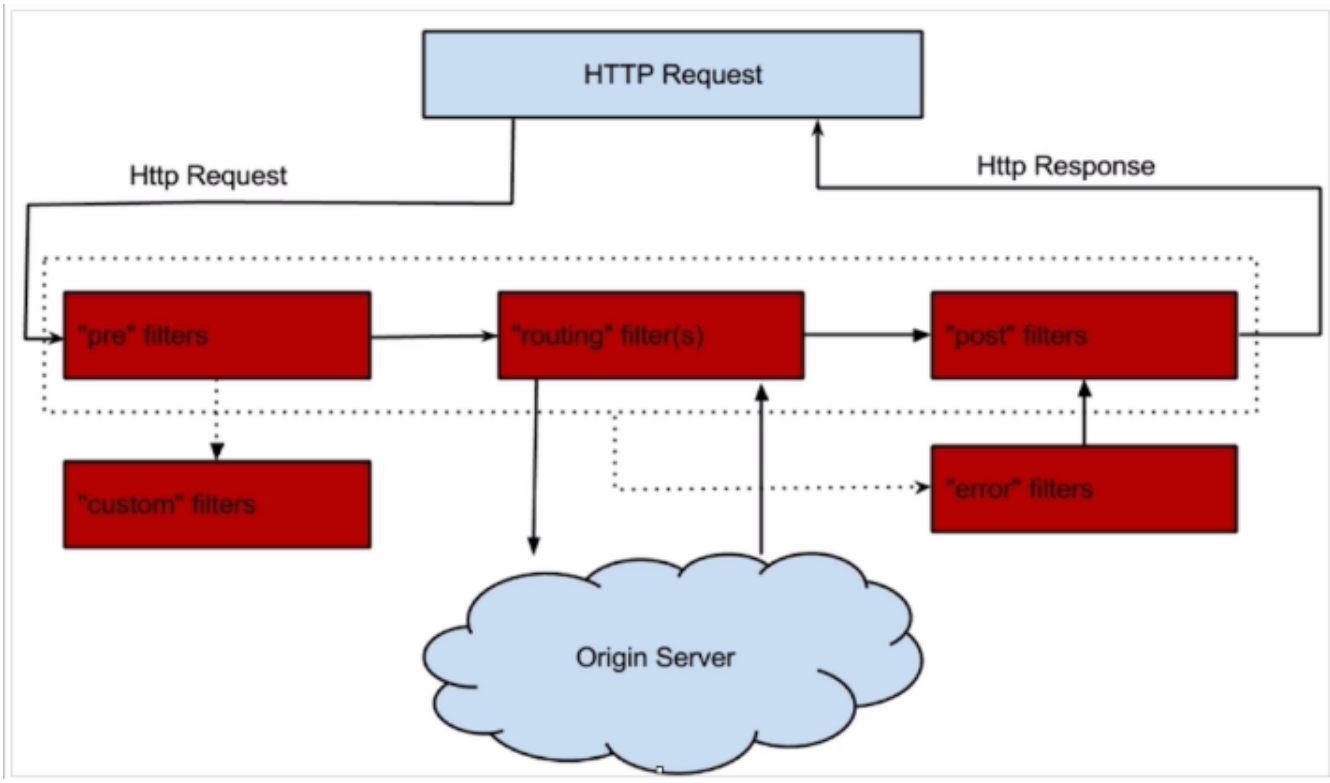
7.2.过滤器

ZuulFilter是过滤器的顶级父类.

```
public abstract ZuulFilter implements IZuulFilter{

    abstract public String filterType();

    abstract public int filterOrder();

    boolean shouldFilter();// 来自IZuulFilter

    Object run() throws ZuulException;// IZuulFilter
}
```

- `shouldFilter`：返回一个`Boolean`值，判断该过滤器是否需要执行。返回true执行，返回false不执行。
- `run`：过滤器的具体业务逻辑。
- `filterType`：返回字符串，代表过滤器的类型。包含以下4种：
    - `pre`：请求在被路由之前执行

    - `route`：在路由请求时调用
    - `post`：在routing和errror过滤器之后调用
    - `error`：处理请求时发生错误调用
- `filterOrder`：通过返回的int值来定义过滤器的执行顺序，数字越小优先级越高。



- 正常流程：
    - 请求到达首先会经过pre类型过滤器，而后到达routing类型，进行路由，请求就到达真正的服务提供者，执行请求，返回结果后，会到达post过滤器。而后返回响应。
- 异常流程：
    - 整个过程中，pre或者routing过滤器出现异常，都会直接进入error过滤器，再error处理完毕后，会将请求交给POST过滤器，最后返回给用户。
    - 如果是error过滤器自己出现异常，最终也会进入POST过滤器，而后返回。
    - 如果是POST过滤器出现异常，会跳转到error过滤器，但是与pre和routing不同的时，请求不会再到达POST过滤器了。

定义过滤器类:

```
@Slf4j
@Component
public class AuthFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }

    @Override
    public int filterOrder() {
        return FilterConstants.PRE_DECORATION_FILTER_ORDER;
    }
```

```
    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() throws ZuulException {
        log.info("经过过滤器");
        // 获取请求上下文
        RequestContext ctx = RequestContext.getCurrentContext();
        // 获取request对象
        HttpServletRequest request = ctx.getRequest();
        // 获取请求参数
        String token = request.getParameter("access-token");
        // 判断是否存在
        if(StringUtils.isEmpty(token)){
            // 如果验证失败,将请求拦截,返回错误提示信息
            ctx.setSendZuulResponse(false);
            // 设置返回状态码
            ctx.setResponseStatusCode(HttpStatus.UNAUTHORIZED.value());
        }
        return null;
    }
}
```

7.3.负载均衡和熔断

Zuul中默认就已经集成了Ribbon负载均衡和Hystix熔断机制。但是所有的超时策略都是走的默认值，比如熔断超时时间只有1S，很容易就触发了。因此建议我们手动进行配置：

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 6000
ribbon:
  ConnectTimeout: 1000
  ReadTimeout: 2000
  MaxAutoRetries: 0
  MaxAutoRetriesNextServer: 1
```