使用 Spring Cloud 构建微服务

王天青 | 2024年7月

TRANSWARP
星 环 科 技

Spring Cloud 是一个基于 Spring Boot 实现的云应用开发工具，它为基于 JVM 的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

Spring Cloud 包含了多个子项目（针对分布式系统中涉及的多个不同开源产品），比如：Spring Cloud Config、Spring Cloud Netflix、Spring Cloud CloudFoundry、Spring Cloud AWS、Spring Cloud Security、Spring Cloud Commons、Spring Cloud Zookeeper、Spring Cloud CLI 等项目。

服务注册与发现

- ✓ 创建“服务注册中心”
- ✓ 创建“服务提供方”
- ✓ 创建“服务消费者”

TRANSWARP
星 环 科 技

创建“服务注册中心”

通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常的简单，只需要在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：

```
@EnableEurekaServer
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}
```

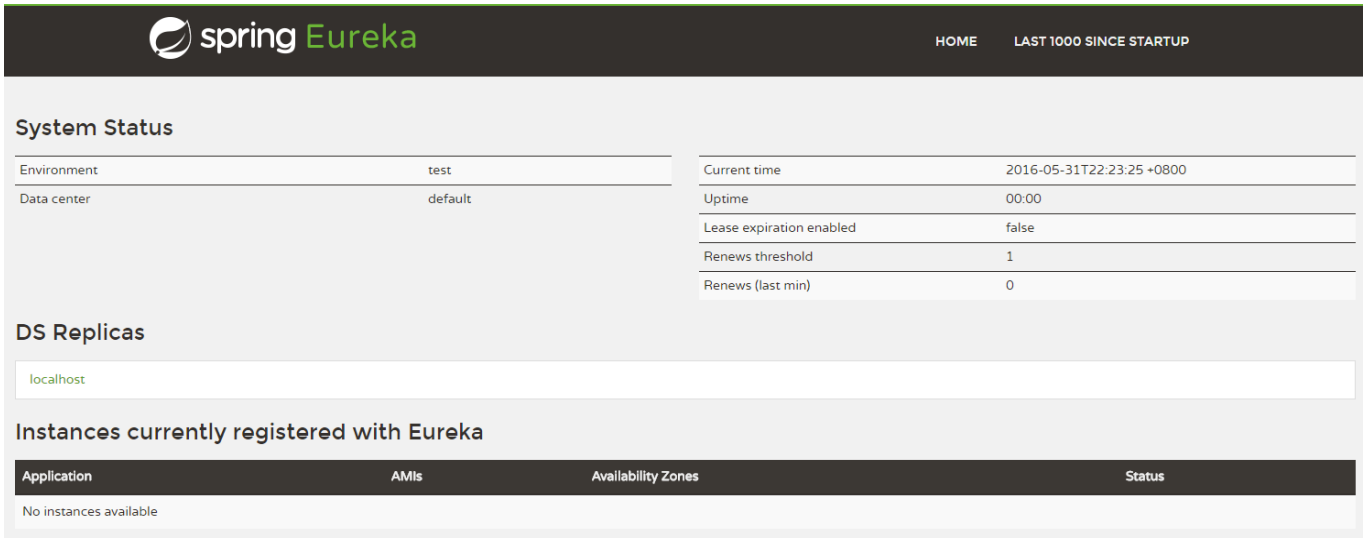
创建“服务注册中心”

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需要在application.properties中问增加如下配置：

```
server.port=1111
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/
```

创建“服务注册中心”

启动工程后，访问：http://localhost:1111/



创建“服务注册中心” - HA

独立模式

只要存在某种监视器或弹性运行时间（例如Cloud Foundry），两个高速缓存（客户机和服务器）和心跳的组合使独立的Eureka服务器对故障具有相当的弹性。在独立模式下，您可能更喜欢关闭客户端行为，因此不会继续尝试并且无法访问其对等体

多节点模式

通过运行多个实例并请求他们相互注册，可以使Eureka更具弹性和可用性。事实上，这是默认的行为，所以你需要做的只是为对方添加一个有效的 serviceUrL

创建“服务注册中心” - HA

```yaml
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```

创建“服务提供方”

假设我们有一个提供计算功能的微服务模块，我们实现一个RESTful API，通过传入两个参数a和b，最后返回a + b的结果。

```
@RestController
public class ComputeController {
    private final Logger logger = Logger.getLogger(getClass());

    @Autowired
    private DiscoveryClient client;

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public Integer add(@RequestParam Integer a, @RequestParam Integer b) {
        ServiceInstance instance = client.getLocalServiceInstance();
        Integer r = a + b;
        logger.info("/add, host:" + instance.getHost() + ", service_id:" + instance.getServiceId() + ", result:" + r);
        return r;
    }

}
```

创建“服务提供方”

最后在主类中通过加上@EnableDiscoveryClient注解，该注解能激活Eureka中的DiscoveryClient实现，才能实现Controller中对服务信息的输出。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ComputeServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(ComputeServerApplication.class, args);
  }
}
```

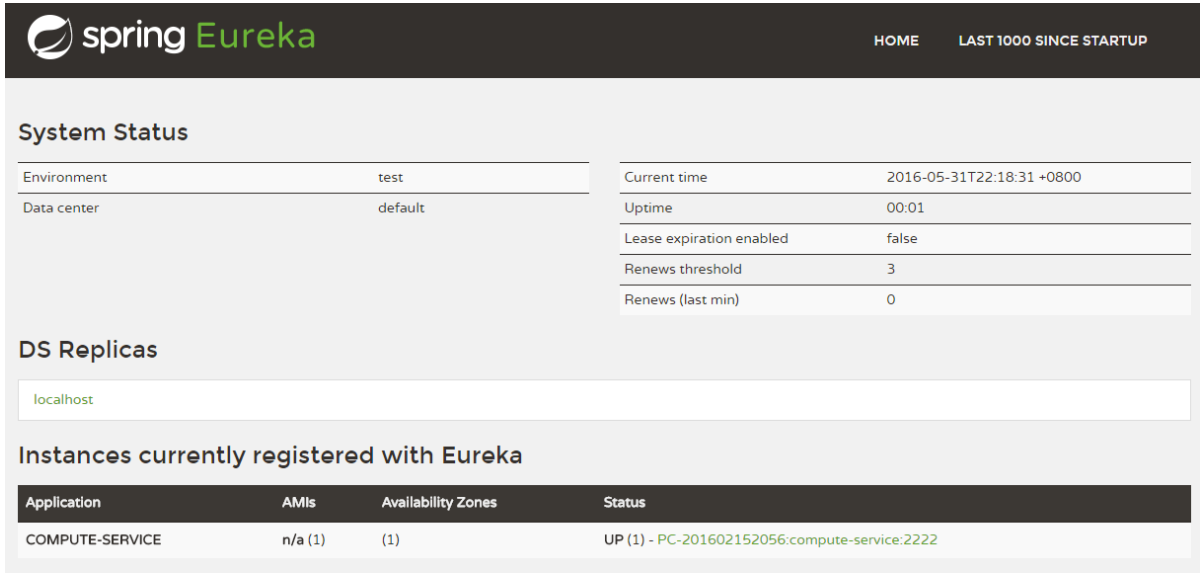
创建“服务提供方”

我们在完成了服务内容的实现之后，再继续对application.properties做一些配置工作

```
spring.application.name=compute-service
server.port=2222
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

创建“服务提供方”

可以看到，我们定义的服务被注册了。



Ribbon

Ribbon是一个基于HTTP和TCP客户端的负载均衡器。Feign 中也使用 Ribbon，后续会介绍 Feign 的使用。

Ribbon可以在通过客户端中配置的 ribbonServerList 服务端列表去轮询访问以达到均衡负载的作用。

当 Ribbon 与 Eureka 联合使用时，ribbonServerList 会被 DiscoveryEnabledNIWSServerList 重写，扩展成从 Eureka 注册中心中获取服务端列表。同时它也会用 NIWSDiscoveryPing 来取代 IPing，它将职责委托给 Eureka 来确定服务端是否已经启动。

- 启动eureka-server中的服务注册中心：eureka-server
- 启动compute-service中的服务提供方：compute-service
- 修改compute-service中的server-port为2223，再启动一个服务提供方：compute-service

创建“服务消费者”

通过 @EnableDiscoveryClient 注解来添加发现服务能力。创建 RestTemplate 实例，并通过 @LoadBalanced 注解开启均衡负载能力。

```
@SpringBootApplication
@EnableDiscoveryClient
public class RibbonApplication {
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
    public static void main(String[] args) {
        SpringApplication.run(RibbonApplication.class, args);
    }
}
```

创建“服务消费者”

创建 ConsumerController 来消费 COMPUTE-SERVICE 的 add 服务。通过直接 RestTemplate 来调用服务，计算10 + 20的值。

```
@RestController
public class ConsumerController {
    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public String add() {
        return restTemplate.getForEntity("http://COMPUTE-SERVICE/add?a=10&b=20", String.class).getBody();
    }
}
```

创建“服务消费者”

Feign是一个声明式的 Web Service 客户端，它使得编写 Web Serivce 客户端变得更加简单。我们只需要使用 Feign 来创建一个接口并用注解来配置它既可完成。它具备可插拔的注解支持，包括 Feign 注解和 JAX-RS 注解。Feign 也支持可插拔的编码器和解码器。Spring Cloud 为 Feign 增加了对Spring MVC 注解的支持，还整合了 Ribbon 和 Eureka 来提供均衡负载的HTTP客户端实现。

下面，通过一个例子来展现 Feign 如何方便的声明对上述 computer-service 服务的定义和调用。

创建“服务消费者”

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

创建“服务消费者”

定义 compute-service 服务的接口，具体如下：

```
@FeignClient("compute-service")
public interface ComputeClient {
    @RequestMapping(method = RequestMethod.GET, value = "/add")
    Integer add(@RequestParam(value = "a") Integer a, @RequestParam(value = "b") Integer b);
}
```

创建“服务消费者”

在 web 层中调用上面定义的 ComputeClient，具体如下：

```
@RestController
public class ConsumerController {
    @Autowired
    ComputeClient computeClient;
    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public Integer add() {
        return computeClient.add(10, 20);
    }
}
```

2
chapter

断路器

- ✓ 创建“断路器”
- ✓ 断路器监控

TRANSWARP
星 环 科 技

熔断器

在微服务架构中，我们将系统拆分成了一个个的服务单元，各单元间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会出现因等待出现故障的依赖方响应而形成任务积压，最终导致自身服务的瘫痪。

举个例子，在一个电商网站中，我们可能会将系统拆分成，用户、订单、库存、积分、评论等一系列的服务单元。用户创建一个订单的时候，在调用订单服务创建订单的时候，会向库存服务来请求出货（判断是否有足够库存来出货）。此时若库存服务因网络原因无法被访问到，导致创建订单服务的线程进入等待库存申请服务的响应，在漫长的等待之后用户会因为请求库存失败而得到创建订单失败的结果。如果在高并发情况之下，因这些等待线程在等待库存服务的响应而未能释放，使得后续到来的创建订单请求被阻塞，最终导致订单服务也不可用。

Ribbon 中引入 Hystrix

- 依次启动eureka-server、compute-service、eureka-ribbon工程
- 访问http://localhost:1111/可以看到注册中心的状态
- 访问http://localhost:3333/add，调用eureka-ribbon的服务，该服务会去调用compute-service 的服务，计算出10+20的值，页面显示30
- 关闭 compute-service 服务，访问 http://localhost:3333/add，我们获得报错信息

在 eureka-ribbon 的主类 RibbonApplication 中使用 @EnableCircuitBreaker 注解开启断路器功能：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class EurekaRibbonClientApplication {

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

    public static void main(String[] args) {
        SpringApplication.run(EurekaRibbonClientApplication.class, args);
    }
}
```

熔断器

改造原来的服务消费方式，新增 ComputeService 类，在使用 ribbon 消费服务的函数上增加 @HystrixCommand 注解来指定回调方法。

```
@Service
public class ComputeService {
    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "addServiceFallback")
    public String addService() {
        return restTemplate.getForEntity("http://COMPUTE-SERVICE/add?a=10&b=20",
String.class).getBody();
    }

    public String addServiceFallback() {
        return "error";
    }
}
```

提供 rest 接口的 Controller 改为调用 ComputeService 的 addService

```
@RestController
public class ConsumerController {
    @Autowired
    private ComputeService computeService;


    @RequestMapping(value = "/add", method = RequestMethod.GET)
    public String add() {
        return computeService.addService();
    }
}
```

验证断路器的回调

1. 依次启动eureka-server、compute-service、eureka-ribbon工程
2. 访问http://localhost:1111/可以看到注册中心的状态
3. 访问http://localhost:3333/add，页面显示：30
4. 关闭compute-service服务后再访问http://localhost:3333/add，页面显示：error

Feign 使用 Hystrix

使用 @FeignClient 注解中的 fallback 属性指定回调类

```
@FeignClient(value = "compute-service", fallback = ComputeClientHystrix.class)
public interface ComputeClient {
    @RequestMapping(method = RequestMethod.GET, value = "/add")
    Integer add(@RequestParam(value = "a") Integer a, @RequestParam(value = "b") Integer b);
}
```

创建回调类 ComputeClientHystrix，实现 @FeignClient 的接口，此时实现的方法就是对应 @FeignClient 接口中映射的 fallback 函数。

```
@Component
public class ComputeClientHystrix implements ComputeClient {
    @Override
    public Integer add(@RequestParam(value = "a") Integer a, @RequestParam(value = "b") Integer
b) {
        return -9999;
    }
}
```

熔断器 – 监控

在 Spring Cloud 中构建一个 Hystrix Dashboard 非常简单，只需要下面四步：

1. 创建一个标准的 Spring Boot 工程，命名为：hystrix-dashboard。
2. 增加依赖内容如下：
    1. spring-cloud-starter-hystrix
    2. spring-cloud-starter-hystrix-dashboard
    3. spring-boot-starter-actuator
3. 为应用主类加上 @EnableHystrixDashboard，启用 Hystrix Dashboard 功能。
4. 根据实际情况修改 application.properties 配置文件，比如：选择一个未被占用的端口等

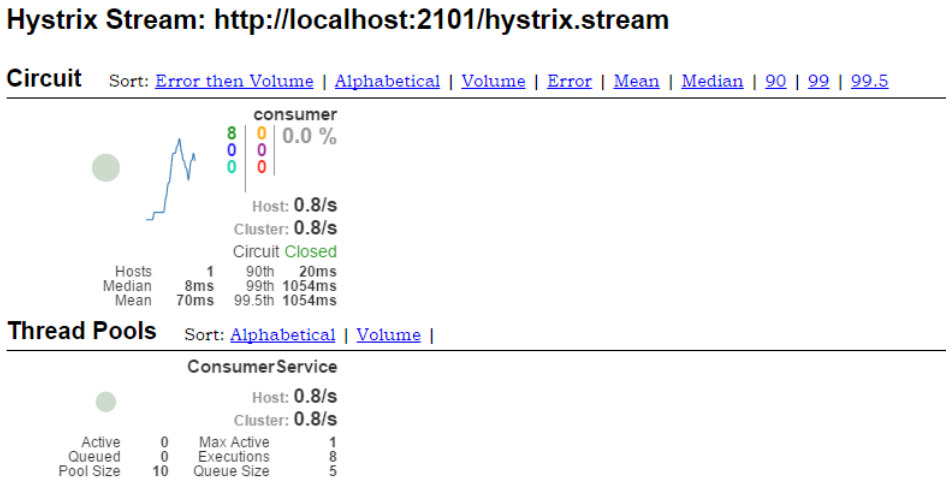
熔断器 – 监控 – 客户端配置

在服务实例 pom.xml 中的 dependencies 节点中新增 spring-boot-starter-actuator 监控模块以开启监控相关的端点，并确保已经引入断路器的依赖 spring-cloud-starter-hystrix：

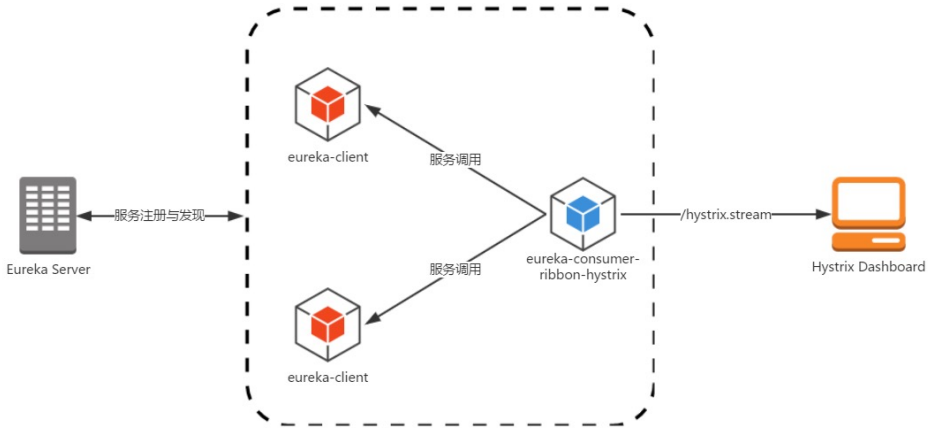
```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

熔断器 – 监控 – 客户端配置

到这里已经完成了所有的配置，我们可以在Hystrix Dashboard的首页输入http://localhost:2101/hystrix.stream，已启动对“eureka-consumer-ribbon-hystrix”的监控，点击“Monitor Stream”按钮，此时我们可以看到如下页面



熔断器



1. eureka-server：服务注册中心
2. eureka-client：服务提供者
3. eureka-consumer-ribbon-hystrix：使用 ribbon 和 hystrix 实现的服务消费者
4. hystrix-dashboard：用于展示eureka-consumer-ribbon-hystrix 服务的 Hystrix 数据

熔断器 – HTTP收集

添加依赖

```
compile('org.springframework.cloud:spring-cloud-starter-eureka')
compile('org.springframework.cloud:spring-cloud-starter-turbine')
testCompile('org.springframework.boot:spring-boot-starter-test')
```

启动类

```
@Configuration
@EnableAutoConfiguration
@EnableTurbine
@EnableDiscoveryClient
public class TurbineApplication {

    public static void main(String[] args) {
        SpringApplication.run(TurbineApplication.class, args);
    }
}
```

熔断器 – HTTP收集

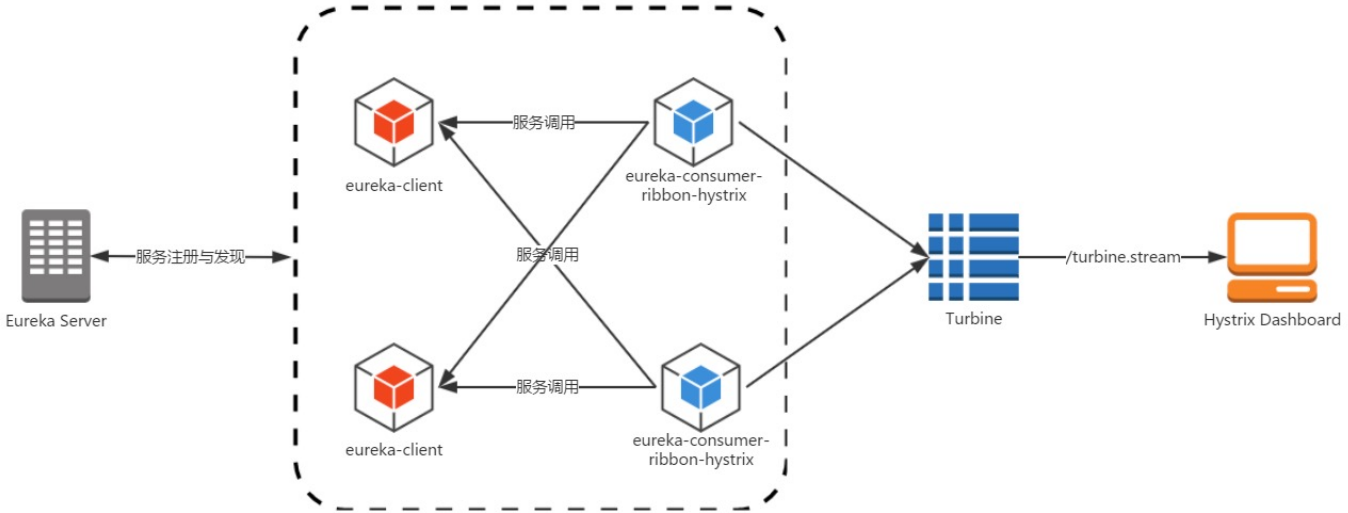
修改配置文件

```
spring.application.name=turbine
server.port=8989
management.port=8990
eureka.client.serviceUrl.defaultZone=http://localhost:1001/eureka/
turbine.app-config=eureka-consumer-ribbon-hystrix
turbine.cluster-name-expression="default"
turbine.combine-host-port=true
```

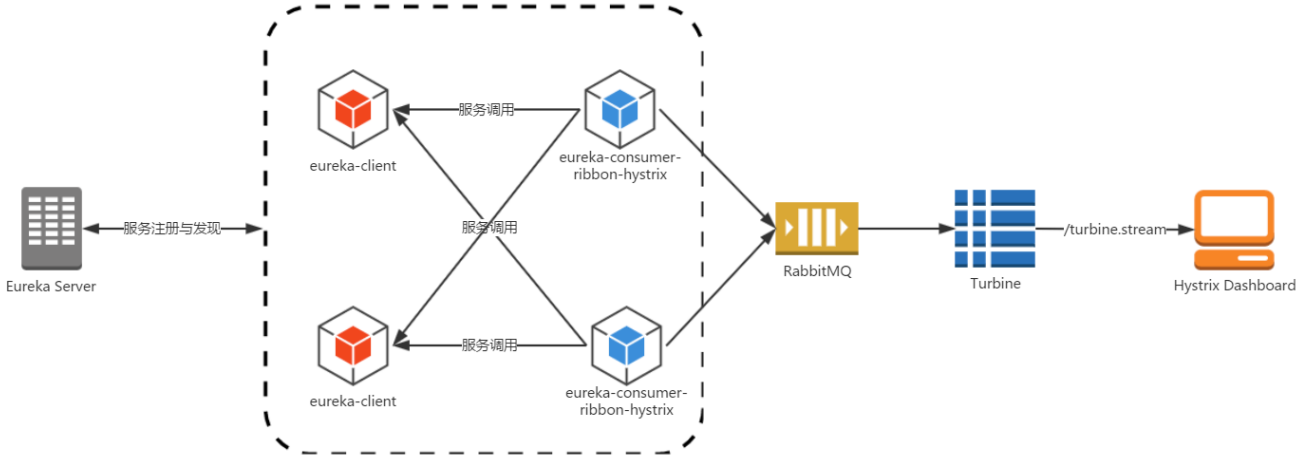
参数说明

- turbine.app-config参数指定了需要收集监控信息的服务名；
- turbine.cluster-name-expression 参数指定了集群名称为default，当我们服务数量非常多的时候，可以启动多个Turbine服务来构建不同的聚合集群，而该参数可以用来区分这些不同的聚合集群，同时该参数值可以在Hystrix仪表盘中用来定位不同的聚合集群，只需要在Hystrix Stream的URL中通过cluster参数来指定；
- turbine.combine-host-port参数设置为true，可以让同一主机上的服务通过主机名与端口号的组合来进行区分，默认情况下会以host来区分不同的服务，这会使得在本地调试的时候，本机上的不同服务聚合成一个服务来统计。

熔断器 - Turbine


服务调用
eureka-client
eureka-consumer-ribbon-hystrix
服务注册与发现
服务调用
Eureka Server
Turbine
/turbine.stream
Hystrix Dashboard
服务调用
eureka-client
eureka-consumer-ribbon-hystrix

熔断器 – Turbine - AMQP



3
chapter

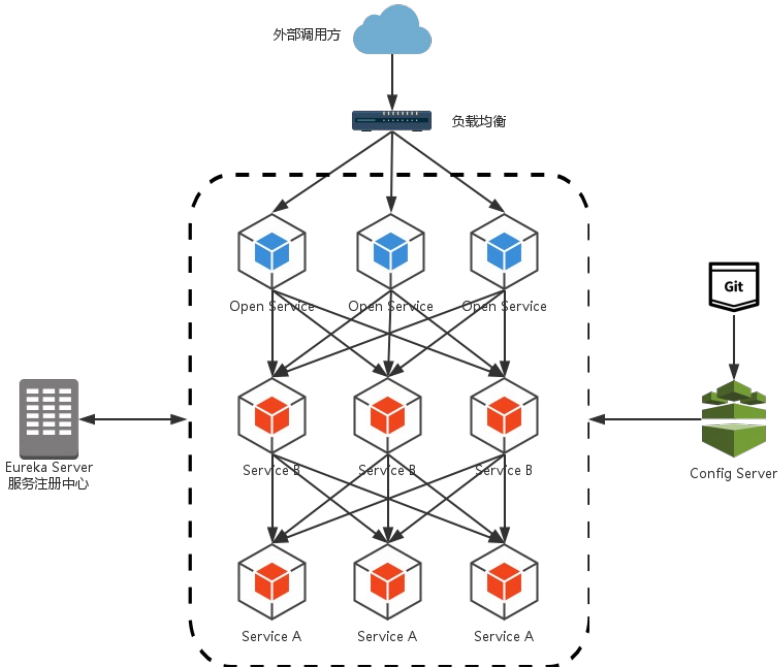
服务网关

- ✓ 服务路由配置
- ✓ 过滤器

TRANSWARP
星 环 科 技

服务网关

通过之前几篇 Spring Cloud 中几个核心组件的介绍，我们已经可以构建一个简略的（不够完善）微服务架构了。比如下图所示：



服务网关

在该架构中，我们的服务集群包含：内部服务Service A和Service B，他们都会注册与订阅服务至Eureka Server，而Open Service是一个对外的服务，通过均衡负载公开至服务调用方。本文我们把焦点聚集在对外服务这块，这样的实现是否合理，或者是否有更好的实现方式呢?

先来说说这样架构需要做的一些事儿以及存在的不足：

- 首先，破坏了服务无状态特点。为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。
- 其次，无法直接复用既有接口。当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

服务网关

添加服务依赖

```
dependencies {
   compile('org.springframework.cloud:spring-cloud-starter-eureka')
   compile('org.springframework.cloud:spring-cloud-starter-zuul')
   testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

创建应用主类，并使用@EnableZuulProxy注解开启Zuul的功能。

```
@EnableZuulProxy
@SpringBootApplication
public class ApiGatewayZuulApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayZuulApplication.class, args);
    }
}
```

创建配置文件 application.yaml，并加入服务名、端口号、eureka 注册中心的地址：

```yaml
spring:
  application:
    name: api-gateway
server:
  port: 1101
eureka:
  client:
    serviceUrl:
      defaultZone: http://127.0.0.1:1111/eureka/
```

服务网关

到这里，一个基于Spring Cloud Zuul服务网关就已经构建完毕。启动该应用，一个默认的服务网关就构建完毕了。由于Spring Cloud Zuul在整合了Eureka之后，具备默认的服务路由功能，即：当我们这里构建的api-gateway应用启动并注册到eureka之后，服务网关会发现上面我们启动的两个服务eureka-client和eureka-consumer，这时候Zuul就会创建两个路由规则。每个路由规则都包含两部分，一部分是外部请求的匹配规则，另一部分是路由的服务ID。针对当前示例的情况，Zuul会创建下面的路由规则：

- 转发到eureka-client服务的请求规则为：/eureka-client/**
- 转发到eureka-consumer服务的请求规则为：/eureka-consumer/**

服务网关 – 传统路由

所谓的传统路由配置方式就是在不依赖于服务发现机制的情况下，通过在配置文件中具体指定每个路由表达式与服务实例的映射关系来实现API网关对外部请求的路由。

- 单实例配置：通过一组zuul.routes.<route>.path与zuul.routes.<route>.url参数对的方式配置，比如：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.url=http://localhost:8080/
```

- 多实例配置：通过一组zuul.routes.<route>.path与zuul.routes.<route>.serviceId参数对的方式配置，比如

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
ribbon.eureka.enabled=false
user-service.ribbon.listOfServers=http://localhost:8080/,http://localhost:8081/
```

服务网关 –服务路由配置

服务路由我们在上一篇中也已经有过基础的介绍和体验，Spring Cloud Zuul通过与Spring Cloud Eureka的整合，实现了对服务实例的自动化维护，所以在使用服务路由配置的时候，我们不需要向传统路由配置方式那样为serviceId去指定具体的服务实例地址，只需要通过一组zuul.routes.<route>.path与zuul.routes.<route>.serviceId参数对的方式配置即可。

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

服务网关 –服务路由配置

在 Spring Cloud Netflix中，Zuul 巧妙的整合了 Eureka 来实现面向服务的路由。实际上，我们可以直接将 API 网关也看做是 Eureka 服务治理下的一个普通微服务应用。它除了会将自己注册到 Eureka 服务注册中心上之外，也会从注册中心获取所有服务以及它们的实例清单。所以，在Eureka 的帮助下，API 网关服务本身就已经维护了系统中所有 serviceId 与实例地址的映射关系。当有外部请求到达API网关的时候，根据请求的 URL 路径找到最佳匹配的 path 规则，API 网关就可以知道要将该请求路由到哪个具体的 serviceId 上去。由于在 API 网关中已经知道 serviceId 对应服务实例的地址清单，那么只需要通过 Ribbon 的负载均衡策略，直接在这些清单中选择一个具体的实例进行转发就能完成路由工作了。

服务网关 -过滤器

通过上面所述的两篇我们，我们已经能够实现请求的路由功能，所以我们的微服务应用提供的接口就可以通过统一的API网关入口被客户端访问到了。但是，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都需要有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能，所有请求都会被毫无保留地转发到具体的应用并返回结果，为了实现对客户端请求的安全校验和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法并不可取，它会增加日后的系统维护难度，因为同一个系统中的各种校验逻辑很多情况下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，冗余代码的出现是我们不希望看到的。所以，比较好的做法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。在完成了剥离之后，有不少开发者会直接在微服务应用中通过调用鉴权服务来实现校验，但是这样的做法仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于业余的逻辑拆分出原有的微服务应用，冗余的拦截器或过滤器依然会存在。

服务网关 -过滤器的实现

```
public class AccessFilter extends ZuulFilter  {
  private static Logger log = LoggerFactory.getLogger(AccessFilter.class);
  @Override
  public String filterType() {
    return "pre";
  }
  @Override
  public int filterOrder() {
    return 0;
  }
  @Override
  public boolean shouldFilter() {
    return true;
  }
}
```

服务网关 -过滤器的实现

```
@Override
  public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();
    log.info("send {} request to {}", request.getMethod(),
request.getRequestURL().toString());
    Object accessToken = request.getParameter("accessToken");
    if(accessToken == null) {
      log.warn("access token is empty");
      ctx.setSendZuulResponse(false);
      ctx.setResponseStatusCode(401);
      return null;
    }
    log.info("access token ok");
    return null;
  }
```

- filterType：过滤器的类型，它决定过滤器在请求的哪个生命周期中执行。这里定义为pre，代表会在请求被路由之前执行。
- filterOrder：过滤器的执行顺序。当请求在一个阶段中存在多个过滤器时，需要根据该方法返回的值来依次执行。
- shouldFilter：判断该过滤器是否需要被执行。这里我们直接返回了true，因此该过滤器对所有请求都会生效。实际运用中我们可以利用该函数来指定过滤器的有效范围。
- run：过滤器的具体逻辑。这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

在实现了自定义过滤器之后，它并不会直接生效，我们还需要为其创建具体的Bean才能启动该过滤器，比如，在应用主类中增加如下内容

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {
  public static void main(String[] args) {
    new SpringApplicationBuilder(Application.class).web(true).run(args);
  }
  @Bean
  public AccessFilter accessFilter() {
    return new AccessFilter();
  }
}
```

- http://localhost:1101/api-a/hello：返回401错误。
- http://localhost:1101/api-a/hello&accessToken=token：正确路由到 hello-service 的 /hello 接口，并返回"Hello World”。

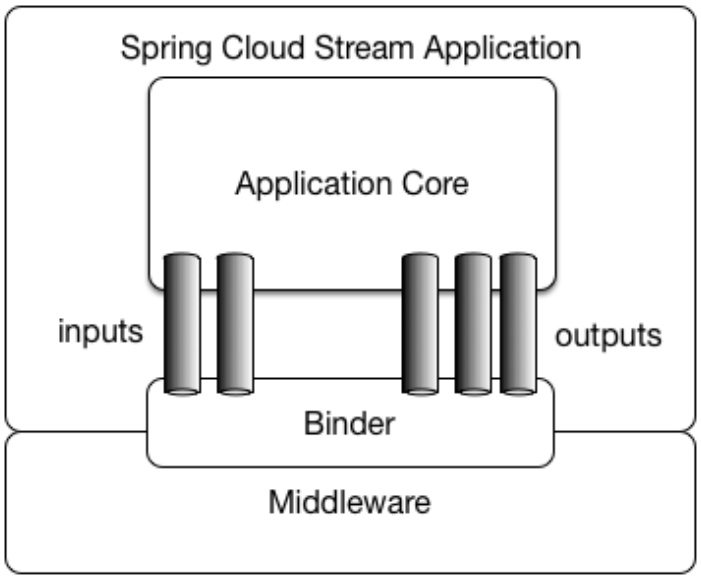
Spring Cloud Stream

TRANSWARP
星 环 科 技

Spring Cloud Stream

Spring Cloud Stream，用精简的语言概括，他本质上其实就是让开发人员使用消息中间件变得简单。基于 Spring Integration 并利用 Spring Boot 提供了自动配置，提供了极为方便的消息中间件使用体验。

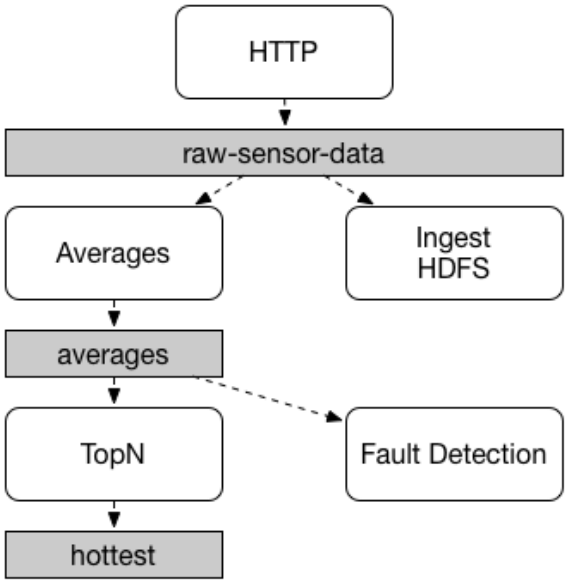
Spring Cloud Stream -应用模型



一个 Spring Cloud Stream 应用程序由一个中间件中立的核心组成。该应用程序通过 Spring Cloud Stream 注入到其中的输入和输出通道与外界进行通信。渠道通过中间件特定的 Binder 实现连接到外部的 broker。

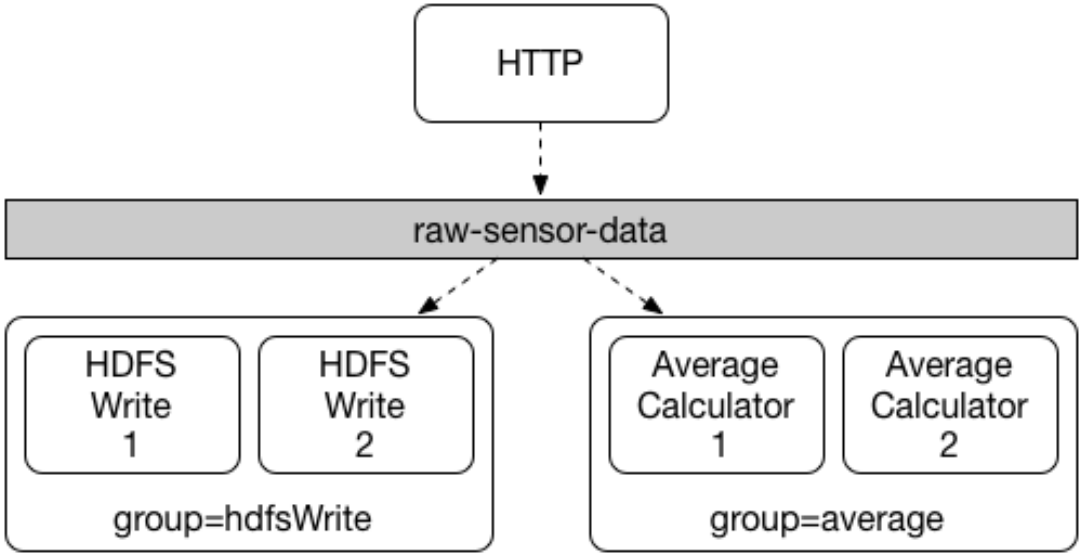
Spring Cloud Stream -订阅支持

应用之间的通信遵循发布订阅模式，其中通过共享主题广播数据。这可以在下图中看到，它显示了一组交互式的Spring Cloud Stream应用程序的典型部署。



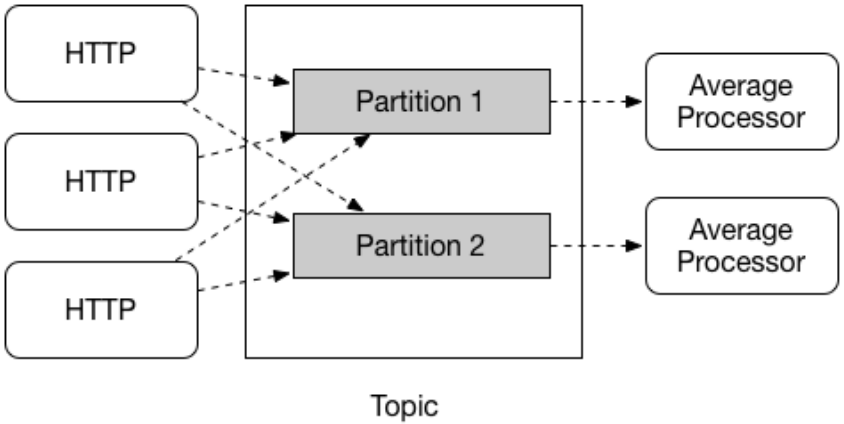
发布订阅通信模型降低了生产者和消费者的复杂性，并允许将新应用程序添加到拓扑中，而不会中断现有流。

Spring Cloud Stream -消费群体



订阅给定目标的所有组都会收到已发布数据的副本，但每个组中只有一个成员从该目的地接收给定的消息。默认情况下，当未指定组时，Spring Cloud Stream将应用程序分配给与所有其他消费者组发布 - 订阅关系的匿名独立单个成员消费者组。

Spring Cloud Stream -分区支持



Spring Cloud Stream 提供对给定应用程序的多个实例之间的分区数据的支持。在分区场景中，物理通信介质（例如，代理主题）被视为被构造成多个分区。一个或多个生产者应用程序实例将数据发送到多个消费者应用程序实例，并确保由共同特征标识的数据由相同的消费者实例处理。

Spring Cloud Stream -触发绑定@EnableBinding

```
@Import(...)
@Configuration
@EnableIntegration
public @interface EnableBinding {
    ...
    Class<?>[] value() default {};
}
```

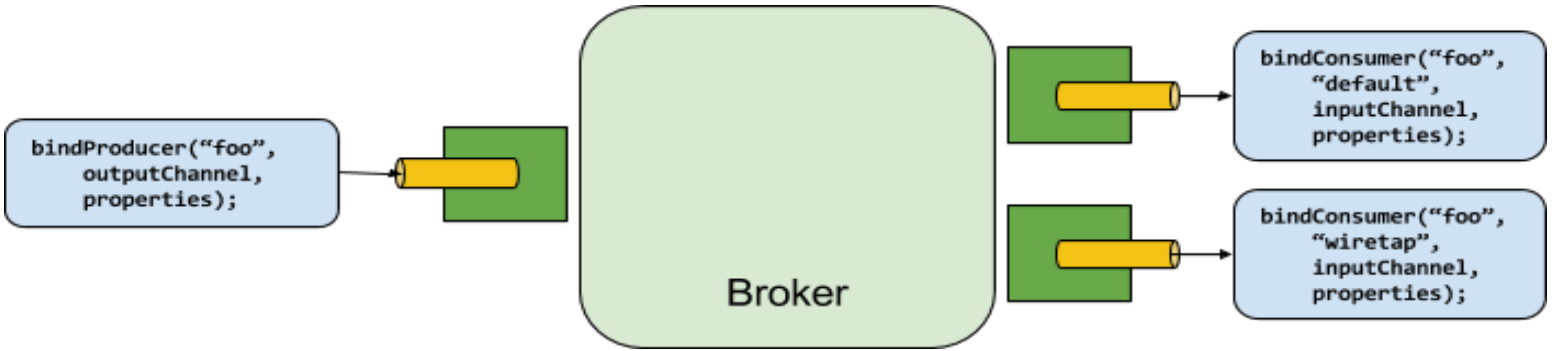
Spring Cloud Stream - @Input和@Output

Spring Cloud Stream 应用程序可以在接口中定义任意数量的输入和输出通道为 @Input 和 @Output 方法：

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();

    @Input("inboundOrders")
    SubscribableChannel orders();
}
```

Spring Cloud Stream -生产和消费消息

Spring Cloud Stream 提供了一个 Binder 抽象，用于连接到外部中间件的物理目标。本节提供有关 Binder SPI，其主要组件和实现特定详细信息背后的主要概念的信息。



```
public interface Binder<T, C extends ConsumerProperties, P extends ProducerProperties> {
    Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

    Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);
}
```

Spring Cloud Stream -入门

要开始创建 Spring Cloud Stream 应用程序，请访问 Spring Initializr 并创建一个名为“GreetingSource”的新 Maven 项目。在下拉菜单中选择 Spring Boot {supported-spring-boot-version}。在“*搜索依赖关系*”文本框中键入 Stream Rabbit 或 Stream Kafka，具体取决于您要使用的 binder。接下来，在与 GreetingSourceApplication 类相同的包中创建一个新类 GreetingSource。给它以下代码：

```
@EnableBinding(Source.class)
public class GreetingSource {

    @InboundChannelAdapter(Source.OUTPUT)
    public String greet() {
        return "hello world " + System.currentTimeMillis();
    }
}
```

要测试驱动此设置，请运行Kafka消息代理。一个简单的方法是使用Docker镜像：

```
# On OS X $ docker run -p 2181:2181 -p 9092:9092 --env
ADVERTISED_HOST=`docker-machine ip \`docker-machine active\`` --
env ADVERTISED_PORT=9092 spotify/kafka
```

```
# On Linux $ docker run -p 2181:2181 -p 9092:9092 --env
ADVERTISED_HOST=localhost --env ADVERTISED_PORT=9092
spotify/kafka
```

消费者应用程序以类似的方式进行编码。返回Initializr并创建另一个名为LoggingSink的项目。然后在与类LoggingSinkApplication相同的包中创建一个新类LoggingSink，并使用以下代码：

```
@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
    public void log(String message) {
        System.out.println(message);
    }
}
```

Q&A

TRANSWARP
星 环 科 技