Springcloud（Eureka+Zuul+Hystrix+Stream）+Swagger使用说明

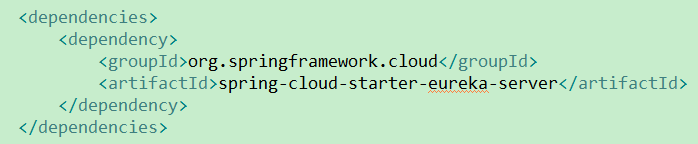
## Eureka（服务发现）使用详解

此处新建两个Springboot工程来实现，一个为Eureka服务注册中心，一个为使用服务注册中心的客户端。

### Eureka服务端工程配置

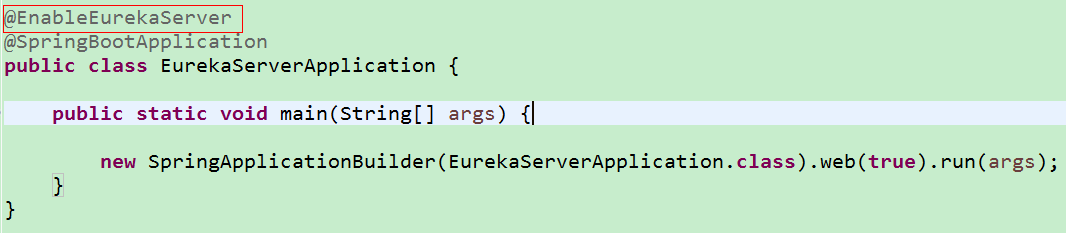
#### pom.xml依赖

在pom.xml加入Eureka依赖，如下图



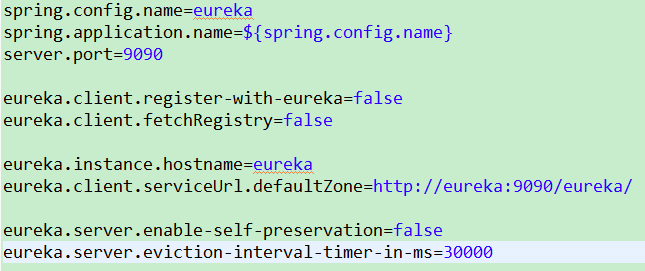
#### Eureka注解

在启动类加入Eureka服务端注解，如下图



#### application.properties配置

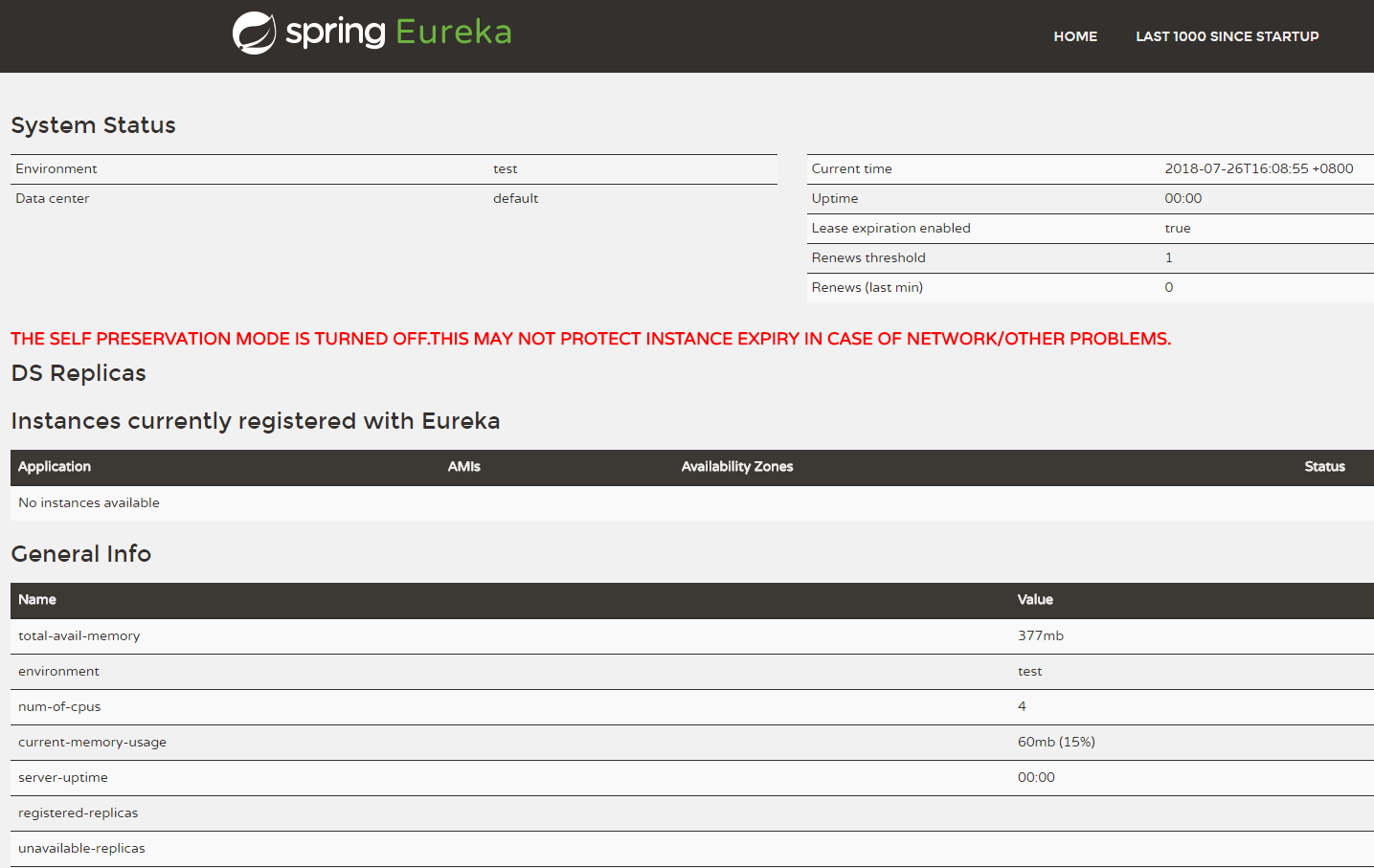
在application.properties配置文件中增加Eureka相关配置信息，如下图



此处将Eureka注册中心端口设置为9090，可根据需求自定义。

#### 启动Eureka服务

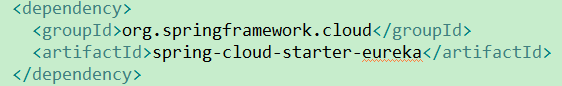
到此Eureka服务端注册中心搭建完成，启动服务，访问路径<http://localhost:9090/>，看到如下界面表示配置成功，服务启动正常。



### Eureka客户端工程配置

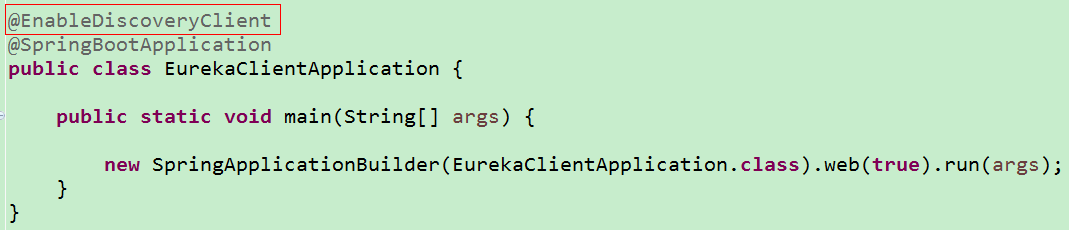
#### pom.xml依赖

在pom.xml加入Eureka依赖，此依赖不同于服务端Eureka依赖，后缀没有server，如下图



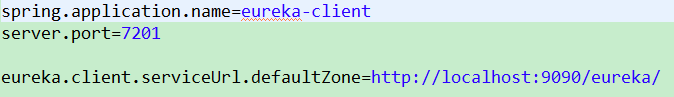
#### Eureka注解

在启动类加入Eureka客户端注解，如下图



#### application.properties配置

在application.properties配置文件中增加Eureka相关配置信息，如下图

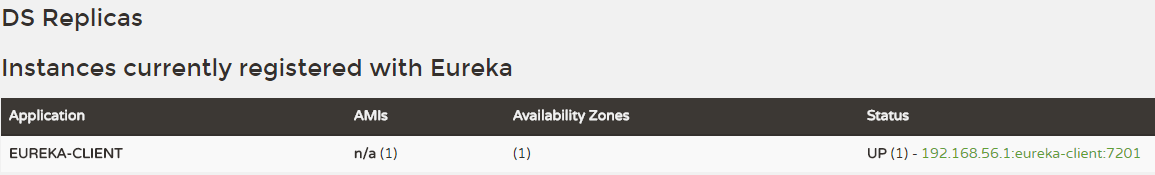


此处将应用客户端工程端口设为7201，可根据需求自定义。

eureka.client.serviceUrl.dafaultZone路径指向Eureka服务端位置。

#### 启动Eureka服务

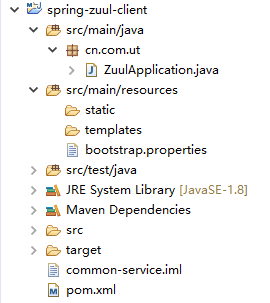
到此Eureka客户端搭建完成，启动服务，访问路径<http://localhost:9090/>，看到如下内容表示客户端服务成功被注册到服务端注册中心。



## Zuul（服务网关）使用详解

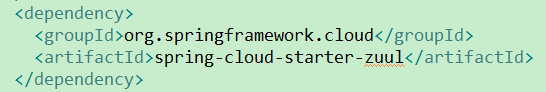
### 新建Zuul服务网关工程

新建一个Zuul服务网关工程，工程名根据自身需求定义，此处示例如图



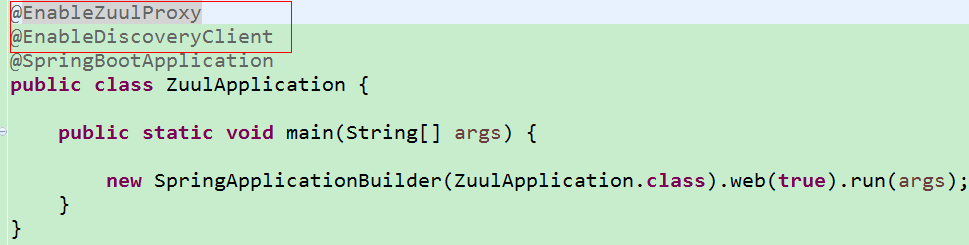
### pom.xml依赖

在pom.xml加入Zuul依赖，如下图



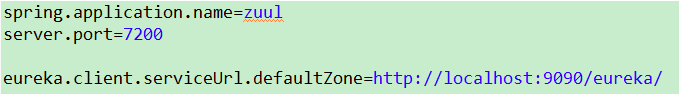
### Zuul注解

在启动类加入Zuul注解，同时加入Eureka客户端注解，将该服务也注册到Eureka服务端注册中心，如下图



### application.properties配置

在application.properties配置文件中增加Eureka相关配置信息，如下图

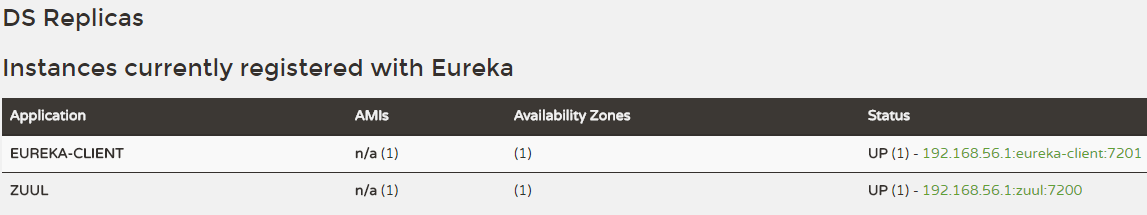


此处将Zuul服务网关工程端口设为7200，可根据需求自定义。

eureka.client.serviceUrl.dafaultZone路径指向Eureka服务端位置。

### 启动Zuul服务

到此Zuul服务网关搭建完成，启动服务，访问路径<http://localhost:9090/>，看到如下内容表示Zuul服务成功被注册到服务端注册中心。



### 访问接口

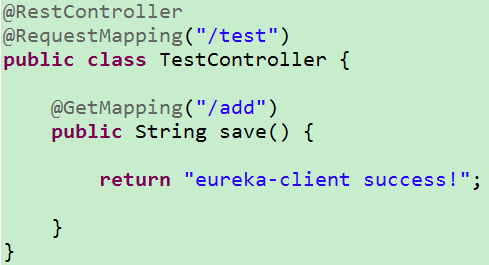
Eureka服务端已经发现上面我们搭建的两个服务，一个是Eureka客户端（eureka-client）和Zuul服务网关（zuul）。

此时Zuul会创建路由规则，转发到eureka-client服务的请求规则为：/ eureka-client/\*\*。

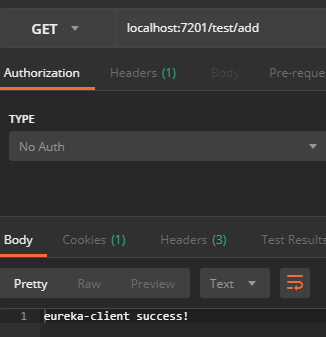
具体指原先访问eureka-client服务接口路径为：http:localhost:7201/\*\*。

现在为：http:localhost:7200/eureka-client/\*\*。

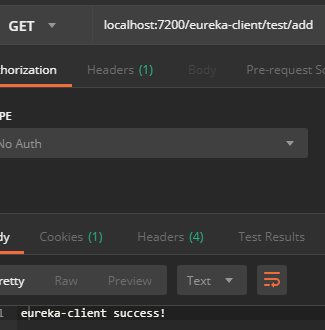
示例代码，在eureka-client工程加入接口如下图



没加入Zuul服务网关前，访问路径为：http:localhost:7201/test/add，访问结果如下图



加入Zuul服务网关后，访问路径为：http:localhost:7200/eureka-client/test/add，结果如下图

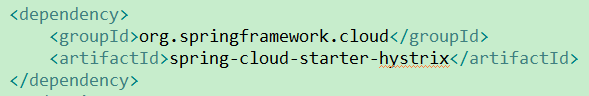


加入多个应用服务，原理相同，访问路径前缀指向Zuul服务网关地址，后面跟上各应用服务名称和接口名称，即会被路由到对应的应用服务处理相关请求。

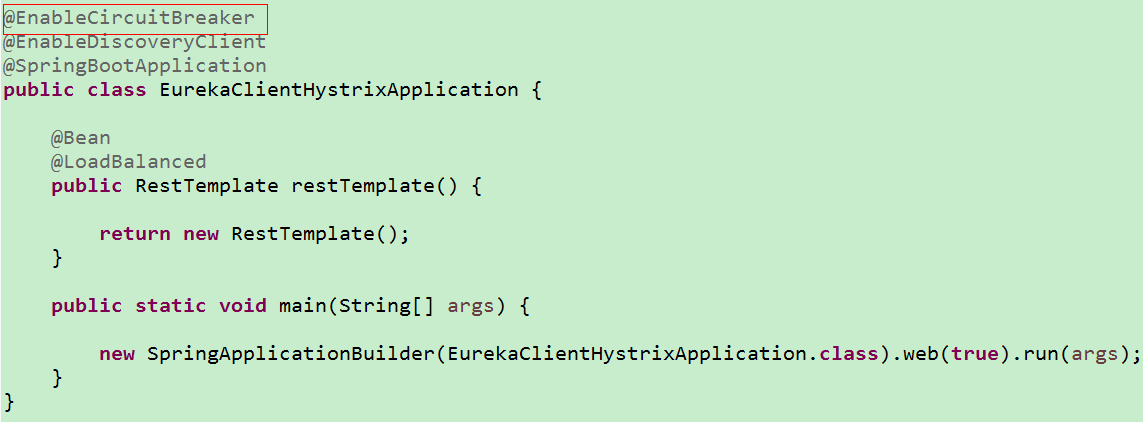
## Hystrix（断路器）使用详解

### Hystrix服务降级

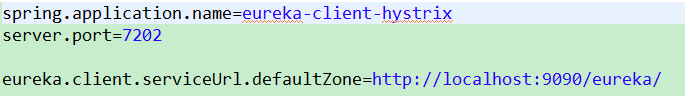
新建一个hystrix应用工程，命名为eureka-client-hystrix。在pom.xml中加入hystrix依赖



在启动类加入hystrix注解



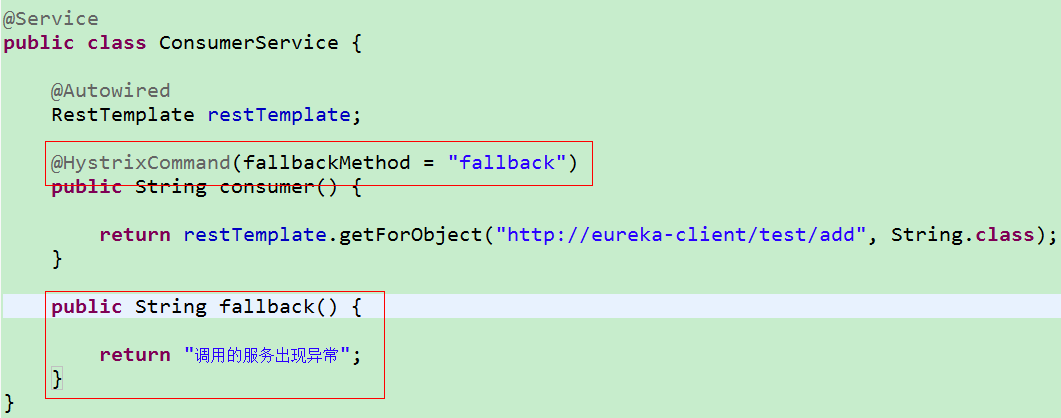
在application.properties配置文件中相关配置信息



此处将应用客户端工程端口设为7202，可根据需求自定义。

eureka.client.serviceUrl.dafaultZone路径指向Eureka服务端位置。

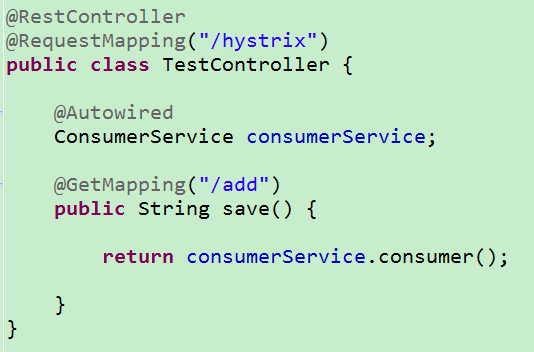
创建一个业务处理类ConsumerService，类中业务方法调用上面创建的eureka-client接口



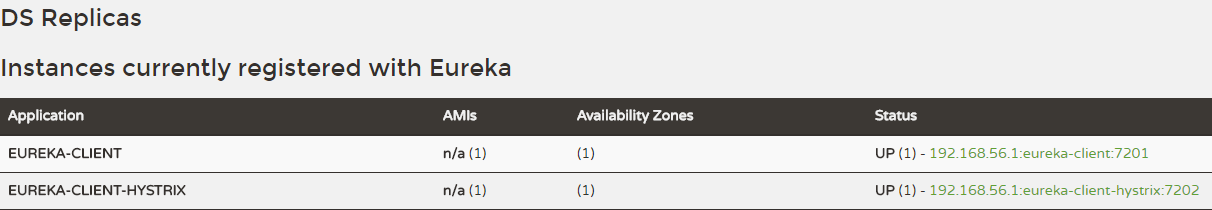
@HystrixCommand注解用来指定服务降级方法，降级方法fallback()。

即当consumer()方法体内，调用eureka-client服务出现异常时，如网络中断，eureka-client服务宕机等问题时，会自动触发调用降级逻辑，执行方法fallback()方法。

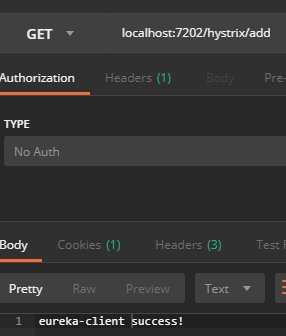
创建一个controller，加入测试接口，用来调用ConsumerService



示例如下，当各服务正常启动时，如下图

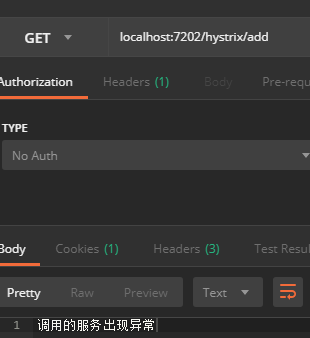


访问hystrix接口，正常结果如下图



此时正确返回了eureka-client返回的结果。

接下来关掉eureka-client服务，再次访问hystrix接口，结果如下图



当调用eureka-client服务异常时，成功进入降级逻辑处理方法。

### Hystrix依赖隔离

在上面实现hystrix服务降级中，使用了@HystrixCommand注解来将某个函数包装成了Hystrix命令，这里除了定义服务降级之外，Hystrix框架就会自动的为这个函数实现调用的隔离。所以，依赖隔离、服务降级在使用时候都是一体化实现的，这样利用Hystrix来实现服务容错保护在编程模型上就非常方便的，并且考虑更为全面。

通过对依赖服务的线程池隔离实现，可以带来如下优势：

* 应用自身得到完全的保护，不会受不可控的依赖服务影响。即便给依赖服务分配的线程池被填满，也不会影响应用自身的额其余部分。
* 可以有效的降低接入新服务的风险。如果新服务接入后运行不稳定或存在问题，完全不会影响到应用其他的请求。
* 当依赖的服务从失效恢复正常后，它的线程池会被清理并且能够马上恢复健康的服务，相比之下容器级别的清理恢复速度要慢得多。
* 当依赖的服务出现配置错误的时候，线程池会快速的反应出此问题（通过失败次数、延迟、超时、拒绝等指标的增加情况）。同时，我们可以在不影响应用功能的情况下通过实时的动态属性刷新（后续会通过Spring Cloud Config与Spring Cloud Bus的联合使用来介绍）来处理它。
* 当依赖的服务因实现机制调整等原因造成其性能出现很大变化的时候，此时线程池的监控指标信息会反映出这样的变化。同时，我们也可以通过实时动态刷新自身应用对依赖服务的阈值进行调整以适应依赖方的改变。
* 除了上面通过线程池隔离服务发挥的优点之外，每个专有线程池都提供了内置的并发实现，可以利用它为同步的依赖服务构建异步的访问。

Hystrix中除了使用线程池之外，还可以使用信号量来控制单个依赖服务的并发度，信号量的开销要远比线程池的开销小得多，但是它不能设置超时和实现异步访问。所以，只有在依赖服务是足够可靠的情况下才使用信号量。在HystrixCommand和HystrixObservableCommand中2处支持信号量的使用：

* 命令执行：如果隔离策略参数execution.isolation.strategy设置为SEMAPHORE，Hystrix会使用信号量替代线程池来控制依赖服务的并发控制。
* 降级逻辑：当Hystrix尝试降级逻辑时候，它会在调用线程中使用信号量。

信号量的默认值为10，我们也可以通过动态刷新配置的方式来控制并发线程的数量。对于信号量大小的估算方法与线程池并发度的估算类似。仅访问内存数据的请求一般耗时在1ms以内，性能可以达到5000rps，这样级别的请求我们可以将信号量设置为1或者2，我们可以按此标准并根据实际请求耗时来设置信号量。

### Hystrix断路器

断路器，当某个服务单元发生故障（类似用电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），直接切断原来的主逻辑调用。对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，当休眠时间窗到期，断路器将进入半熔断状态，释放一次请求到原来的主逻辑上，如果此次请求正常返回，那么断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

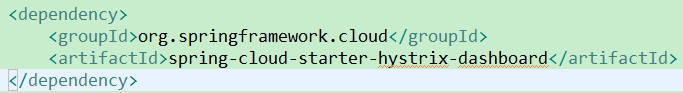
断路器起作用的三个重要参数：快照时间窗、请求总数下限、错误百分比下限。

* 快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。
* 请求总数下限：在快照时间窗内，必须满足请求总数下限才有资格根据熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用此时不足20次，即时所有的请求都超时或其他原因失败，断路器都不会打开。
* 错误百分比下限：当请求总数在快照时间窗内超过了下限，比如发生了30次调用，如果在这30次调用中，有16次发生了超时异常，也就是超过50%的错误百分比，在默认设定50%下限情况下，这时候就会将断路器打开。

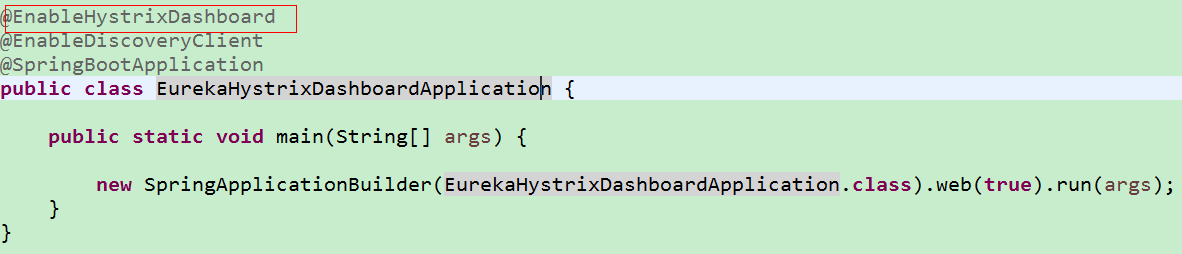
通过上面的一系列机制，hystrix的断路器实现了对依赖资源故障的端口、对降级策略的自动切换以及对主逻辑的自动恢复机制。这使得我们的微服务在依赖外部服务或资源的时候得到了非常好的保护，同时对于一些具备降级逻辑的业务需求可以实现自动化的切换与恢复，相比于设置开关由监控和运维来进行切换的传统实现方式显得更为智能和高效。

### Hystrix监控面板

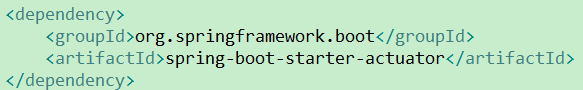
新建一个hystrix-dashboard应用工程。在pom.xml中加入hystrix-dashboard依赖



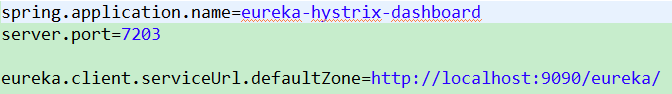
在启动类加入hystrix-dashboard注解



在被监控的hystrix工程pom.xml中，加入spring-boot-starter-actuator依赖



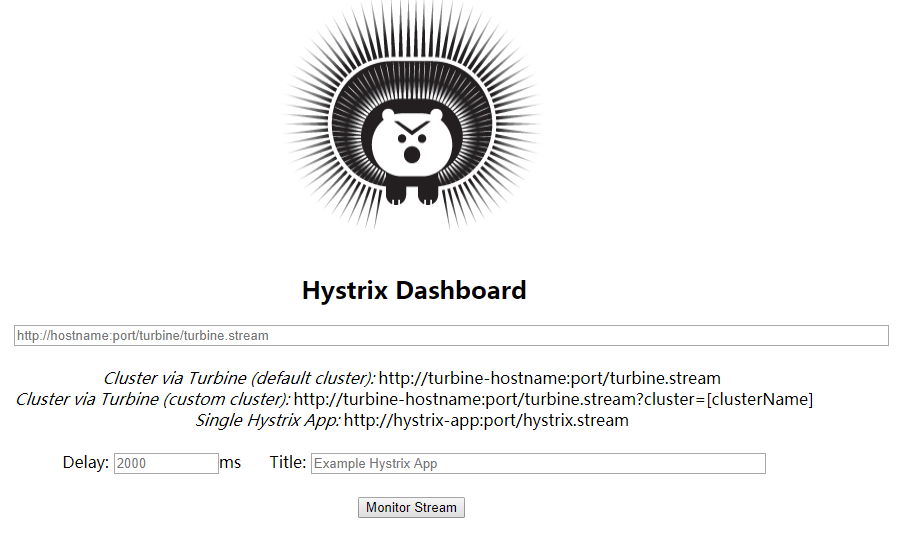
在application.properties配置文件中相关配置信息



此处将应用客户端工程端口设为7203，可根据需求自定义。

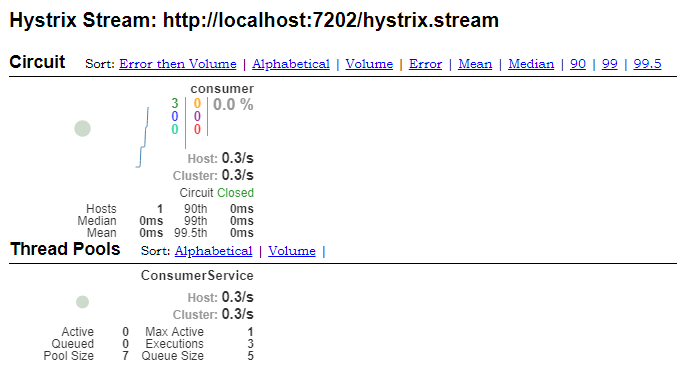
eureka.client.serviceUrl.dafaultZone路径指向Eureka服务端位置。

启动应用，访问<http://localhost:7203/hystrix>，服务正常将会看到如下界面

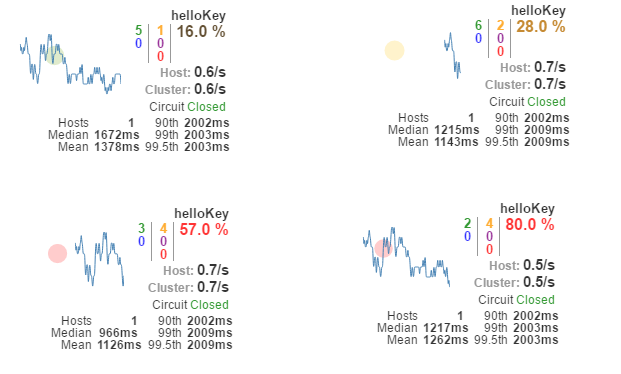
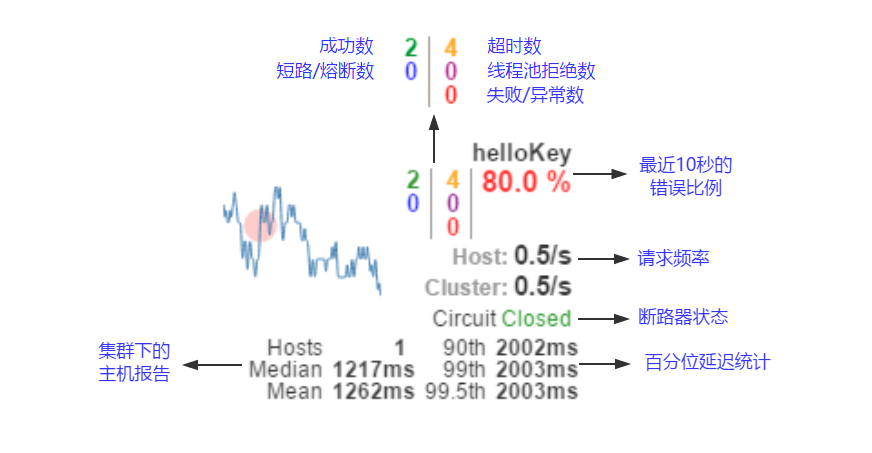


* Delay：该参数用来控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，我们可以通过配置该属性来降低客户端的网络和CPU消耗。
* Title：该参数对应了上图头部标题Hystrix Stream之后的内容，默认会使用具体监控实例的URL，我们可以通过配置该信息来展示更合适的标题。

在界面地址栏输入：<http://localhost:7202/hystrix.stream>，前缀为被监控的hystrix服务路径，然后点击“Monitor Stream”按钮，此时我们可以看到如下页面



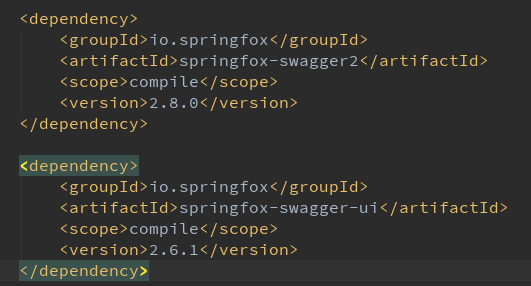
监控信息的左上部分找到两个重要的图形信息：一个实心圆和一条曲线。

* + 实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，如下图所示，它的健康度从绿色、黄色、橙色、红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，我们就可以在大量的实例中快速的发现故障实例和高压力实例。  
    [](http://blog.didispace.com/content/images/posts/spring-cloud-starter-dalston-5-1-3.png)
  + 曲线：用来记录2分钟内流量的相对变化，我们可以通过它来观察到流量的上升和下降趋势。
* 其他一些数量指标如下图所示：  
  [](http://blog.didispace.com/content/images/posts/spring-cloud-starter-dalston-5-1-4.png)

## Swagger使用详解

### pom.xml依赖

在pom.xml加入Swagger依赖，如下图



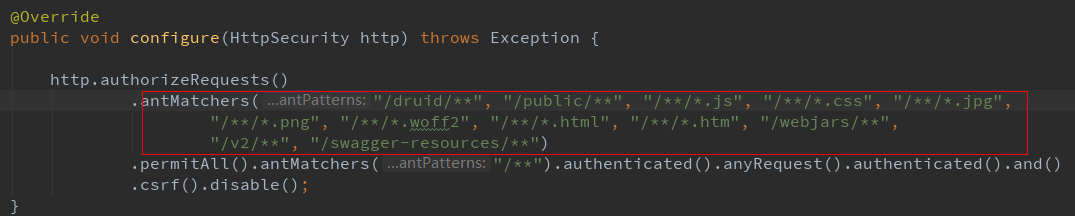
### 创建Swagger配置类

在启动类同级目录增加Swagger配置类APPBaseSwagger.java，内容如下附件：



### 调整权限验证对Swagger放行

修改ResourceServerConfig.java，如下图：



### 添加文档内容

完成上述配置后，启动服务，访问地址：<http://127.0.0.1:6201/swagger-ui.html>，已经可以生成文档内容。以下Swagger注解可以丰富文档内容。

[@Api](https://my.oschina.net/u/2396174)：用在类上，说明该类的作用。

@ApiOperation：注解来给API增加方法说明

@ApiImplicitParams : 用在方法上包含一组参数说明

@ApiImplicitParam：用来注解来给方法入参增加说明

@ApiResponses：用于表示一组响应

@ApiResponse：用在@ApiResponses中，一般用于表达一个错误的响应信息

    l   **code**：数字，例如400

    l   **message**：信息，例如"请求参数没填好"

    l   **response**：抛出异常的类

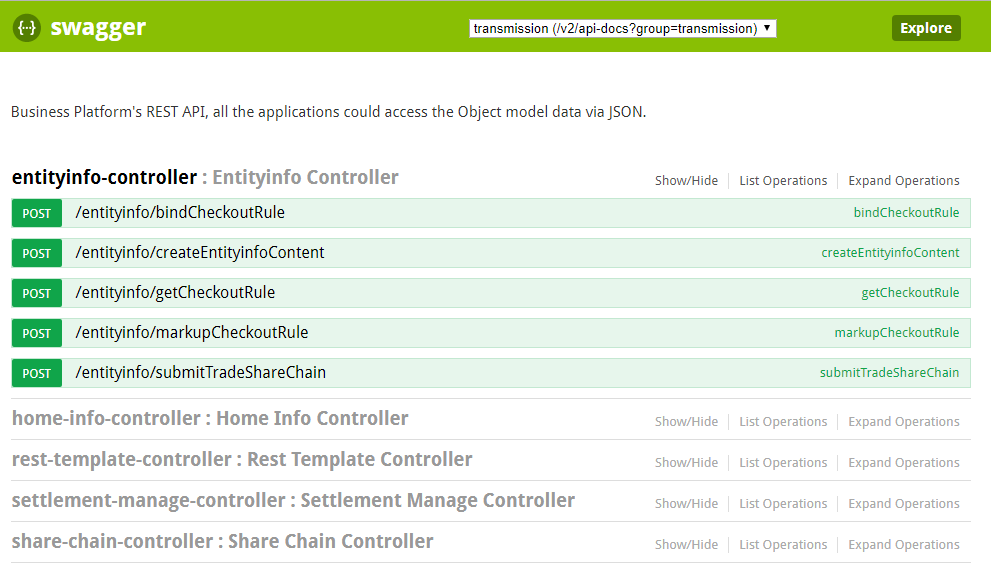
@ApiModel：描述一个Model的信息（一般用在请求参数无法使用@ApiImplicitParam注解进行描述的时候）

    l   **@ApiModelProperty**：描述一个model的属性

### Swagger UI面板说明

启动服务，访问地址：<http://127.0.0.1:6201/swagger-ui.html>。

IP和端口根据自身情况填写，后面不变。打开效果如下图：

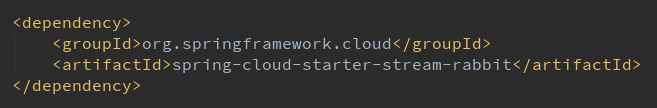


显示所有controller名称，点击controller名称，会展开该controller下所包含的接口，点击接口名称，会展开详细的接口请求参数和返回值等信息。

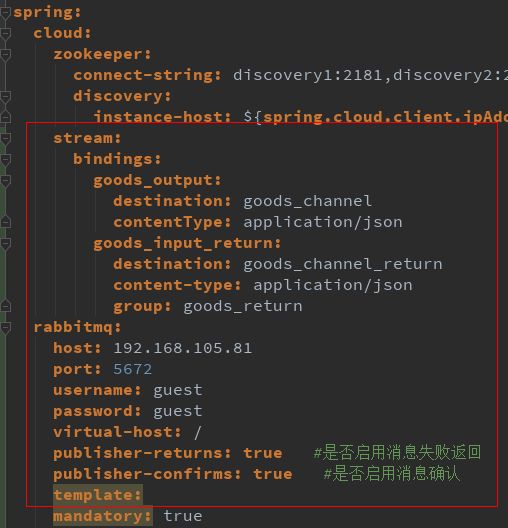
## Stream使用详解

### pom.xml依赖

在pom.xml加入Swagger依赖，如下图



### bootstrap.yml配置



Rabbitmq配置连接信息，用户名密码。

Bindings用于绑定消息发送和接受的通道于交换机。

分组：同一交换机下的消息接受者，如果没有配置分组，则都会收到消息；

配置相同分组，则消息之后被其中一个接收消费，不会重复消费；

配置不同分组，则消息会广播到每个分组，每个分组只有一个能接收消费。

### 生产者

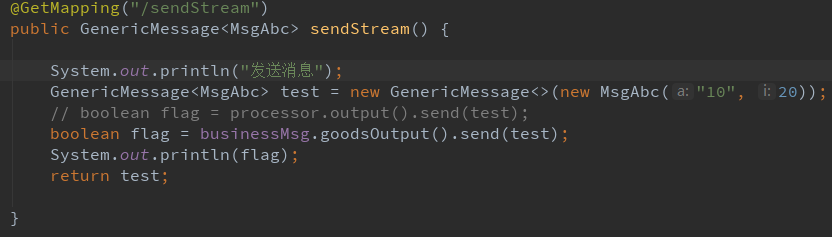
绑定消息通道



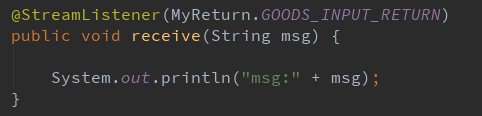
注入通道实例



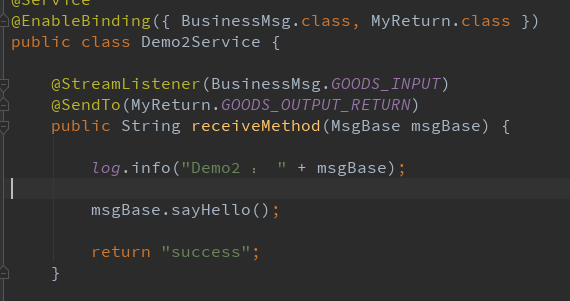
发送消息



接收消费者应答消息



### 消费者



@EnableBingding用于绑定消息通道

@StreamListener用于监听消息

@SendTo用于返回消息应答信息

### 自定义通道

系统自带有Sink，Source，Processor三个通道，也可以自定义，下面为上面配置文件绑定的自定义通道：

