

HystrixDashbord

Hystrix（注意 是单纯的Hystrix） 提供了对于微服务调用状态的监控（信息）， 但是，需要结合 spring-boot-actuator 模块一起使用.

在包含了 hystrix的项目中， 引入依赖：

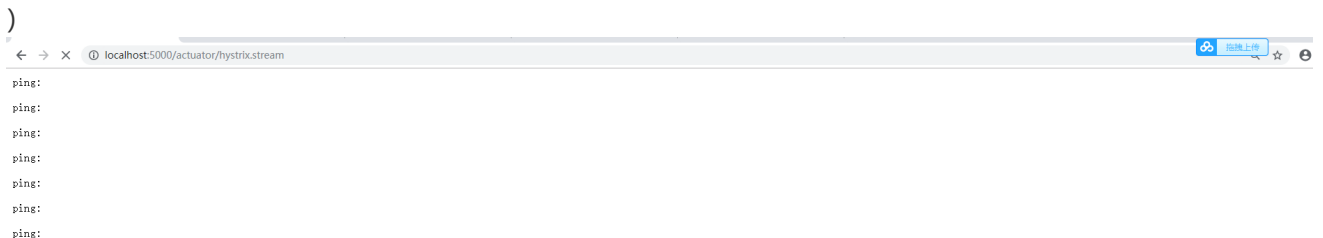
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

这时候访问/actuator/hystrix.stream 便可以看见微服务调用的状态信息

（需要注意的是， 在Spring Finchley 版本以前访问路径是/hystrix.stream， 如果是Finchley 的话 还得在yml里面加入配置：

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

因为spring Boot 2.0.x以后的Actuator 只暴露了info 和health 2个端点,这里我们把所有端点开放。



这里会发现没有任何信息， 因为我刚启动项目， 我们来调用几个接口看看：

```
↩ → × 🌐 localhost:3000/actuator/nxtrix/stream

ping:

data:
{"type":"HystrixCommand","name":"power","group":"UserController","currentTime":1549952462938,"isCircuitBreakerOpen":false,"errorPercentage":100,"errorCount":1,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountFails":0,"rollingCountFailures":0,"rollingCountFallbacks":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":1,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":0,"latencyExecute":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"latencyTotal_mean":0,"latencyTotal":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"propertyValue_circuitBreakerRequestVolumeThreshold":5,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":3000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"power"}

data:
{"type":"HystrixThreadPool","name":"power","currentTime":1549952462938,"currentActiveCount":0,"currentCompletedTaskCount":1,"currentCorePoolSize":1,"currentLargestPoolSize":1,"currentMaximPoolSize":1,"currentPoolSize":1,"currentQueueSize":0,"currentTaskCount":1,"rollingCountThreadsExecuted":0,"rollingMaxActiveThreads":0,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}

ping:

data:
{"type":"HystrixCommand","name":"power","group":"UserController","currentTime":1549952463437,"isCircuitBreakerOpen":false,"errorPercentage":100,"errorCount":1,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountFails":0,"rollingCountFailures":0,"rollingCountFallbacks":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":1,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":0,"latencyExecute":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"latencyTotal_mean":0,"latencyTotal":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"propertyValue_circuitBreakerRequestVolumeThreshold":5,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":3000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"power"}

data:
{"type":"HystrixThreadPool","name":"power","currentTime":1549952463437,"currentActiveCount":0,"currentCompletedTaskCount":1,"currentCorePoolSize":1,"currentLargestPoolSize":1,"currentMaximPoolSize":1,"currentPoolSize":1,"currentQueueSize":0,"currentTaskCount":1,"rollingCountThreadsExecuted":0,"rollingMaxActiveThreads":0,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}

ping:

data:
{"type":"HystrixCommand","name":"power","group":"UserController","currentTime":1549952463938,"isCircuitBreakerOpen":false,"errorPercentage":100,"errorCount":1,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountFails":0,"rollingCountFailures":0,"rollingCountFallbacks":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":1,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":0,"latencyExecute":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"latencyTotal_mean":0,"latencyTotal":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"propertyValue_circuitBreakerRequestVolumeThreshold":5,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":3000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"power"}
```

这些密密麻麻的，就是我们的微服务监控的信息，但是，这种json格式的字符串，难免会让人不太好阅读，所以，这时候需要我们的主角登场了：

HystrixDashbord

什么是HystrixDashbord/如何使用？

Dashbord 翻译一下的意思是 仪表盘，顾名思义，hystrix监控信息的仪表盘，那这个仪表盘到底是什么样子呢？以及 怎么来使用呢？

我们新建一个项目 加入依赖：

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

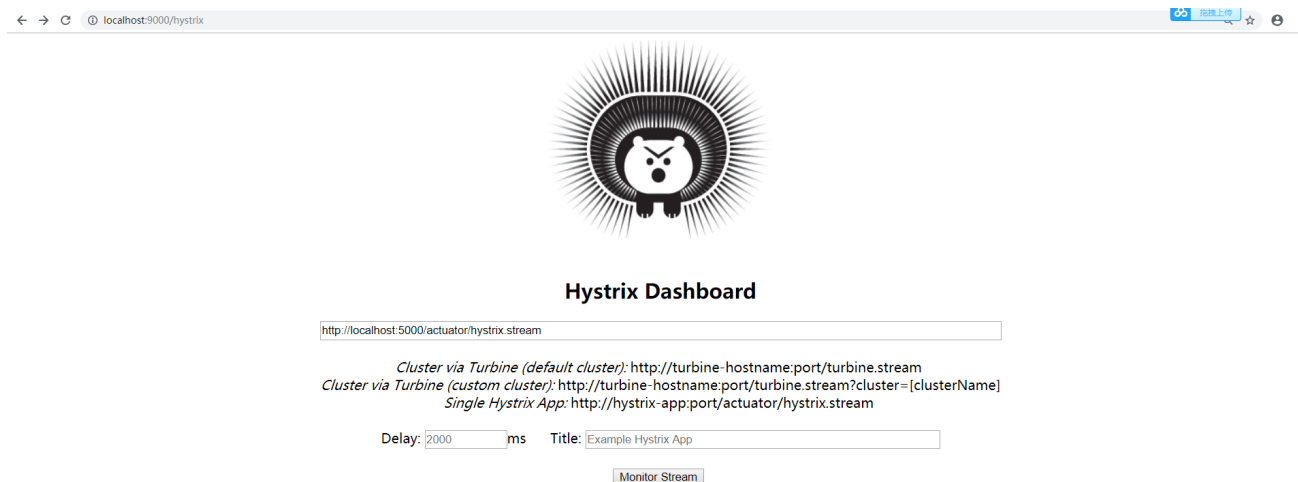
在spring boot启动类上面加入注解EnableHystrixDashboard

```
@SpringBootApplication
@EnableHystrixDashboard
public class AppHystrixDashbord {

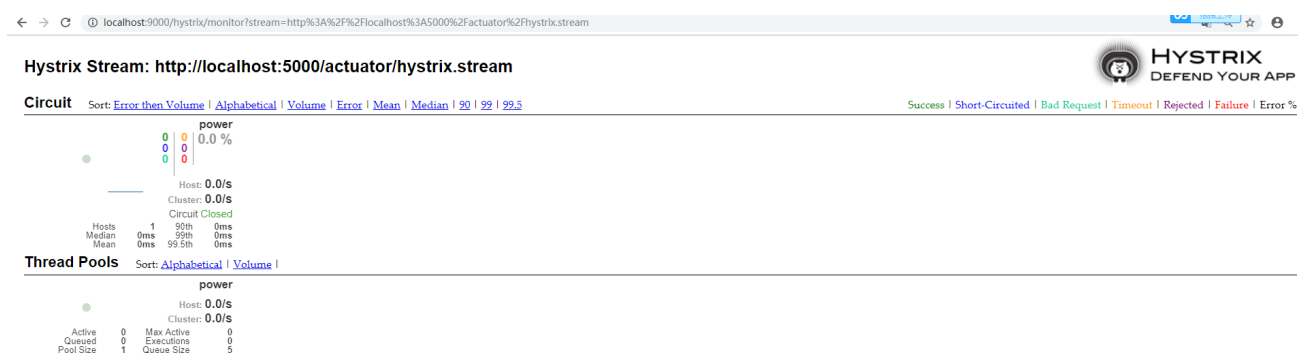
    public static void main(String[] args) {
        SpringApplication.run(AppHystrixDashbord.class);
    }

}
```

启动项目后访问/hystrix能看见一个类似tomcat的首页：



在中间这个输入框中，填入需要监控的微服务的监控地址 也就是/actuator/hystrix.stream点击按钮，就会跳转到仪表盘页面：



当然，如果你微服务没有发生过调用，那么这个页面就会一直显示加载中，我这里是调用后的效果。

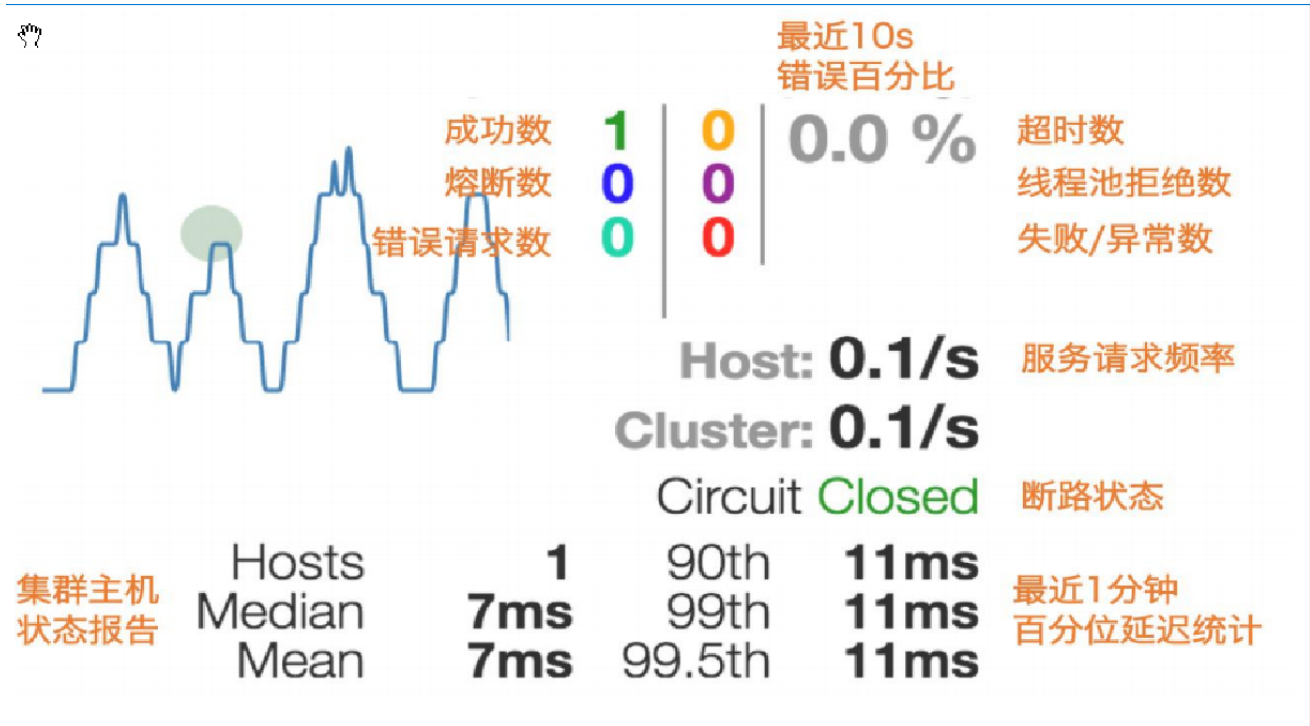
Hystrix仪表盘解释：

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，它的健康度从绿色

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，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。

曲线：用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

整图解释：



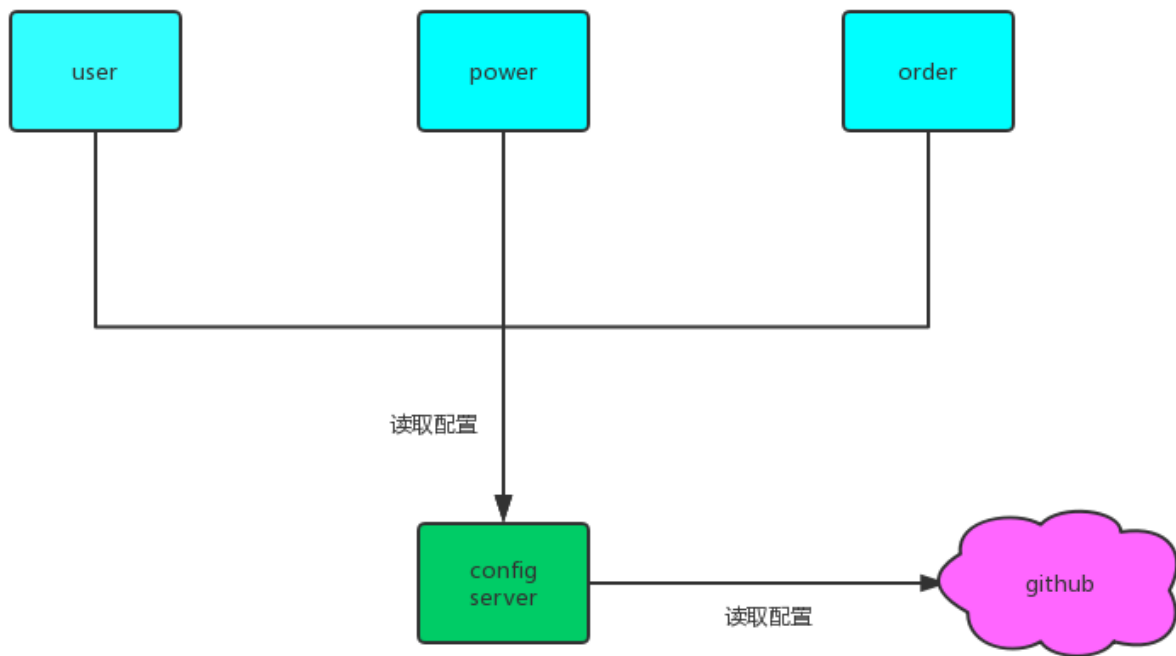
Spring-Cloud-Config

config是什么?

我们既然要做项目，那么就少不了配置，传统的项目还好，但是我们微服务项目，每个微服务就要做独立的配置，这样难免有点复杂，所以，config项目出来了，它就是为了解决这个问题：把你所有的微服务配置通过某个平台：

比如 github，gitlib 或者其他 git 仓库 进行集中化管理（当然，也可以放在本地）。

可能这样讲有点抽象，我们来看一张图：



大概是这样一个关系

如何使用config?

刚刚讲完理论，那么我们来实践一下，怎么配置这个config呢？我们刚刚说过 由一个config server 来管理所有的配置文件，那么我们现在新建一个config server 项目 然后引入依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

spring-cloud 的依赖我们就不提了

然后启动类上面加入注解EnableConfigServer:

```
@SpringBootApplication
@EnableConfigServer
public class AppConfig {

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

yml配置:

```




server:
  port: 8080
spring:
  application:
    name: test

cloud:
  config:
    server:
      git:
        uri: https://github.com/513667225/my-spring-cloud-config.git #配置文件在github
上的地址
        # search-paths: foo,bar* #Configserver会在 Git仓库根目录、 foo子目录, 以及所有以
bar开始的子目录中查找配置文件。
        # clone-on-start: true #启动时就clone仓库到本地, 默认是在配置被首次请求时, config
server才会clone git仓库
        #native:
        #search-locations: classpath:/config #若配置中心在本地, 本地的地址

```

配置好以后, 我们先试试通过config server来读取配置

这里我在github上有一些配置文件:

 test-config.yml	测试提交	21 days ago
 test-config1.yml	测试提交	21 days ago
 test-config2.yml	测试提交	21 days ago

Help people interested in this repository understand your project by adding a README.
 Add a README

我们来看看test-config的内容:

1 contributor

22 lines (15 sloc) | 210 Bytes

Raw Blame History

```

1  spring:
2    profiles:
3      active: test
4
5
6  ---
7
8  server:
9    port: 8201
10
11 spring:
12   profiles: dev
13   application:
14     name: test-cloud-dev-2.0
15
16
17 ---
18
19 spring:
20   profiles: test
21   application:
22     name: test-cloud-test-2.0

```

那么如何通过config server来访问呢?

启动项目后，我们可以通过名字来读取里面的配置信息：

```
< → ↻ ⓘ localhost:8080/test-config.yml

spring:
  profiles:
    active: test
```

那我们要获取dev环境/或者test环境下的配置呢？通过-隔开即可。

我们现在来访问 test-config-dev：

```
< → ↻ ⓘ localhost:8080/test-config-dev.yml

server:
  port: 8201
spring:
  application:
    name: test-cloud-dev-2.0
  profiles:
    active: test
```

同理 如果要访问test环境下的配置，改为test即可

其实，config访问配置文件，是需要一个具体的访问规则的，那么这个访问规则到底是什么呢？我们可以在官网找到：

```
/ {application} / {profile} [ / {label} ]
/ {application} - {profile} . yml
/ {label} / {application} - {profile} . yml
/ {application} - {profile} . properties
/ {label} / {application} - {profile} . properties
```

application就是配置文件的名字， profile就是对应的环境 label就是不同的分支 由这个规则可见，我们使用的是第二种规则，剩下的规则，同学们可以自己试试，对于yaml 和properties类型config可以完美转换，也就是说你存的是yaml 但是可以读取为properties类型的反过来也是如此：

客户端从config上获取配置

刚刚给大家简单演示了一下config 以及怎么读取配置，不过实际开发中，更多的不是我们人为去获取，而是由微服务从config上加载配置，那么，怎么来加载呢？

首先，我们需要在我们的微服务加入一个依赖声明他是config的客户端：

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

需要注意的是，这个依赖不包括spring -boot依赖，也就是说，假设你这个项目要当作spring boot来启动的话，还得依赖spring boot

启动类不需要做改动，标准的spring boot启动类即可

需要注意的是yaml文件

以前我们对于spring boot的配置 是在application.yml里面配置的,现在从config上读取配置的话，还得需要一个bootstrap.yml配置文件

解释一下这个bootstrap.yml:

spring cloud有一个“引导上下文”的概念，这是主应用程序的父上下文。引导上下文负责从配置服务器加载配置属性，以及解密外部配置文件中的属性。和主应用程序加载application.(yml或properties)中的属性不同，引导上下文加载(bootstrap.)中的属性。配置在 bootstrap.*中的属性有更高的优先级，因此默认情况下它们不能被本地配置

那么我们application.yml配置文件里面 只需要做一些简单的配置就可以了：

```
spring:
  application:
    name: test-config
```

重点在于bootstrap.yml:

```
spring:
  cloud:
    config:
      name: test-config #这是我们要读取的配置文件名 对应获取规则的{application}
      profile: dev      #这个是要获取的环境 对应的便是{profile}
      label: master     #这个就是获取的节点 对应的是{label}
      uri: http://localhost:8080/ #这就是我们config server的一个地址
```


那么 他就会获取到我们刚刚看到的那个配置:

```
server:
  port: 8201

spring:
  profiles: dev
  application:
    name: test-cloud-dev-2.0
```

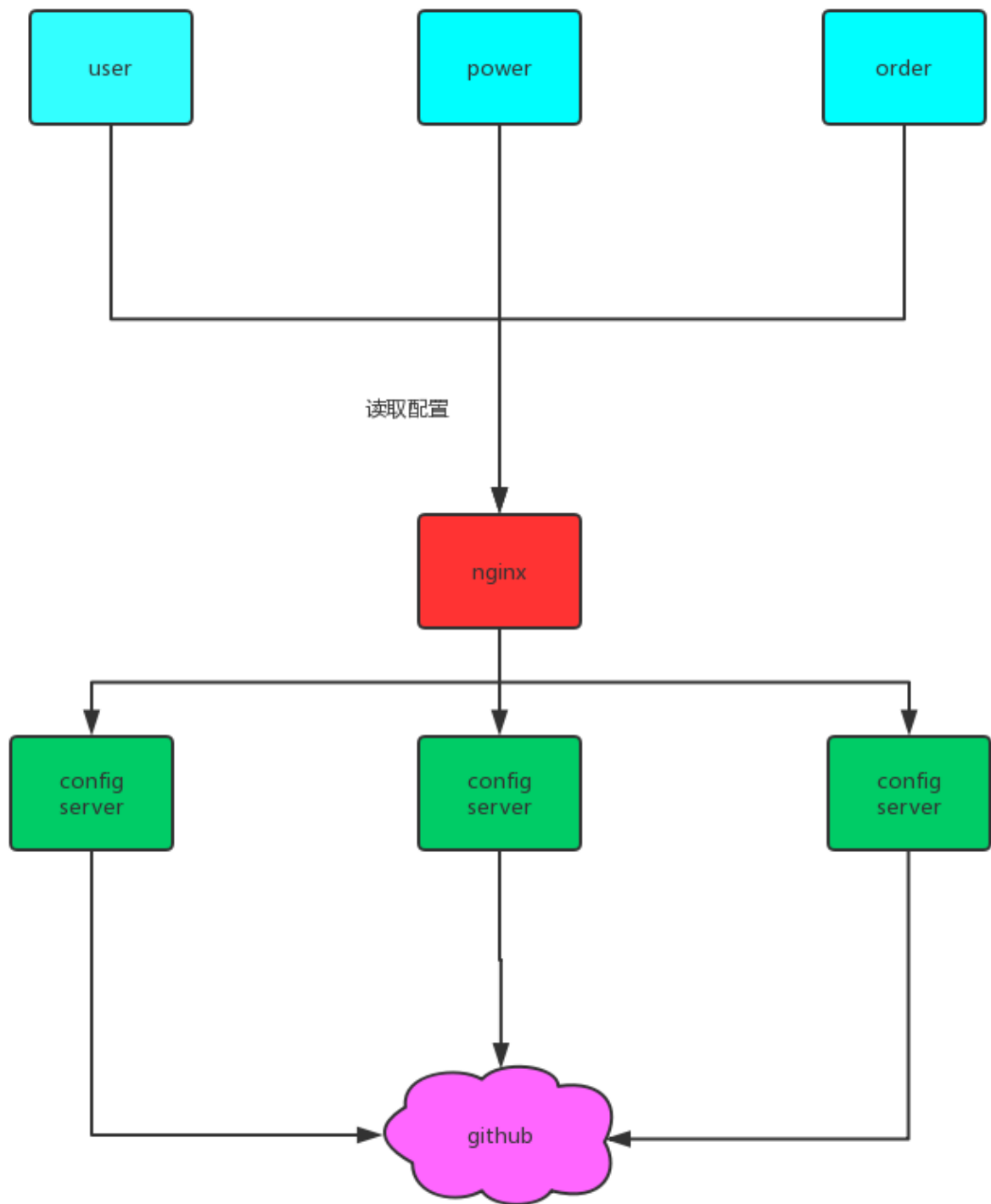
我们来测试一下 看看他会不会使用这个8201端口启动

```
2019-02-12 20:36:03.312 INFO 18300 --- [main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8201
2019-02-12 20:36:03.315 INFO 18300 --- [main] com.AppConfigClient : Started AppConfigClient in 12.812 seconds (JVM running for 13.3s)
2019-02-12 20:36:05.288 ERROR 18300 --- [nfoReplicator-0] c.n.d.s.t.d.RedirectingEurekaHttpClient : Request execution error
```

这里 我们查看启动信息，能发现他现在使用的是我们从config server上读取到的配置。

spring cloud config 高可用

config 高可用可以通过很多种方式，比如说搭建一个nginx:



或者config server注册到eureka上，client端也注册到eureka上，则已经实现高可用

如何注册就不提了，需要注意的点就是当config server都注册完之后 client的配置文件进行以下改动：

```
spring:
  cloud:
    config:
      name: test-config
      profile: dev
      label: master
      discovery:
        enabled: true
        service-id: test-config
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:3000/eureka/
```