

东软睿道内部公开

文件编号：D000-

Spring Cloud微服务架构

版本：1.0.0

第6章 Hystrix容错处理

东软睿道教育信息技术有限公司
(版权所有，翻版必究)

Copyright © Neusoft Educational Information Technology Co., Ltd
All Rights Reserved



本章教学目标

- ✓ 了解分布式系统面临的问题、雪崩效应；
- ✓ 了解Hystrix简介；
- ✓ 理解服务容错、服务熔断与服务降级的区别；
- ✓ 掌握服务熔断；
- ✓ 掌握服务降级；
- ✓ 掌握Hystrix Dashboard监控数据；
- ✓ 掌握Turbine监控集群数据；

本章教学内容

节	知识点	掌握程度	难易程度	教学形式	对应在线微课
Hystrix简介	分布式系统面临的问题	了解		线下	
	雪崩效应	了解		线下	
	服务容错	理解		线下	
	Hystrix简介	了解		线下	
使用Hystrix实现容错	Hystrix服务熔断	掌握		线下	
	测试服务熔断	掌握		线下	
	Hystrix服务降级	掌握	难	线下	
	测试服务降级	掌握		线下	
	服务熔断与服务降级的区别	理解		线下	
使用Hystrix Dashboard 监控数据	Hystrix Dashboard监控数据	掌握	难	线下	
	服务监控测试	掌握		线下	
使用Turbine监控集群数据	Turbine监控集群数据	掌握		线下	
	集群监控测试	掌握		线下	

CONTENTS

目录

01

Hystrix简介

02

使用Hystrix实现容错

03

使用Hystrix Dashboard监控数据

04

使用Turbine监控集群数据

分布式系统面临的问题

- ❖ 复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免地失败。
- ❖ 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，移动端将调用订单服务的创建订单接口，此时创建订单接口又会向库存服务请求出货(判断是否有足够库存来出货)。 此时若库存服务因自身处理逻辑等原因造成响应缓慢，会直接导致创建订单服务的线程被挂起，以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。 如果在高并发情况下之下， 因这些挂起的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，最终导致订单服务不可用。

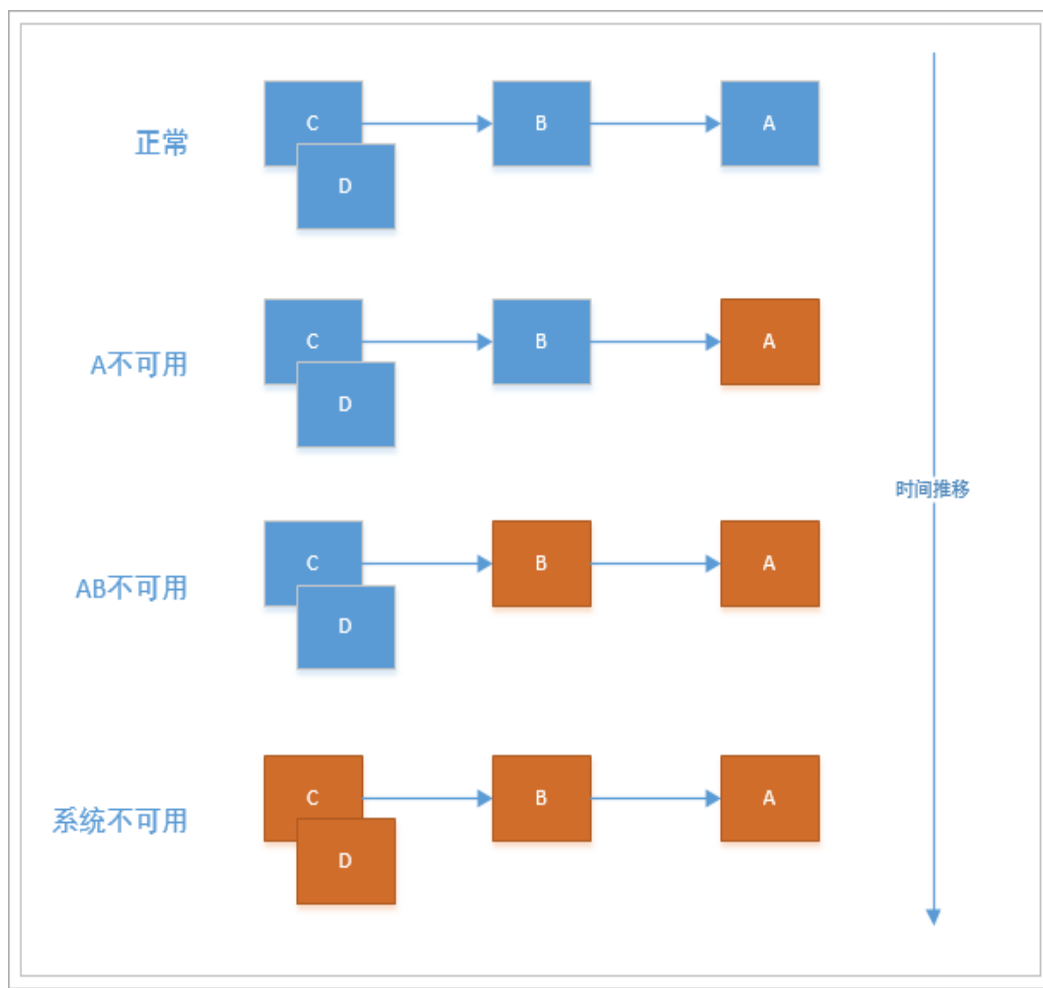
雪崩效应

- ❖ 在微服务架构中，存在这么多的服务单元，若一个单元出现故障，就很容易因依赖关系引发故障蔓延，最终导致整个系统瘫痪，这种现象被称之为**雪崩效应**。
- ❖ 服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用，并将不可用逐渐放大的过程。



雪崩效应

- ❖ 如图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



服务容错

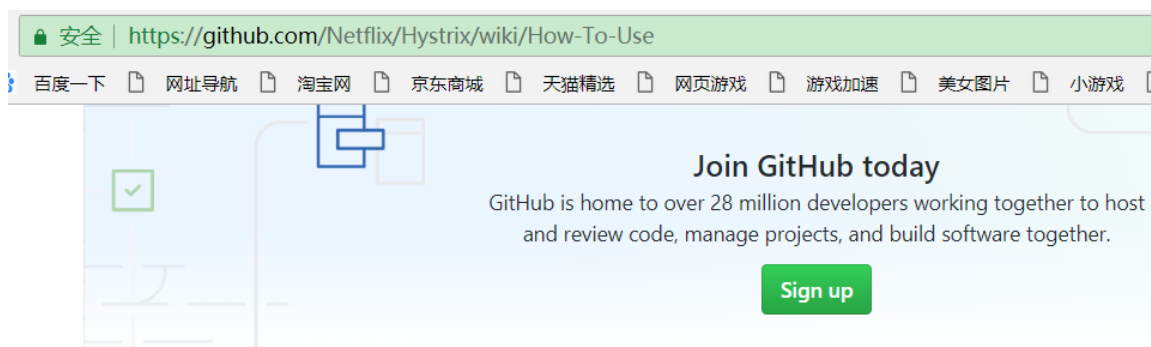
- ❖ 微服务往往服务众多，各服务之间相互调用，若消费者在调用提供者时出现由网络、提供者服务自身问题等导致接口出现故障或延迟；此时消费者的请求不断增加，必定会出现因等待响应而造成任务积压，线程无法释放，进一步导致服务的瘫痪。
- ❖ 微服务架构既然存在这样的隐患，那么针对服务的容错性必然会进行服务降级，依赖隔离，断路器等一线列的服务保护机制。
- ❖ “断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

Hystrix简介

- ❖ Spring Cloud Hystrix中实现了线程隔离、断路器等一系列的服务保护功能。它也是基于Netflix的开源框架 Hystrix实现的，该框架目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。
- ❖ Hystrix具备了服务降级、服务熔断、线程隔离、请求缓存、请求合并以及服务监控等强大功能。

Hystrix简介

- ❖ Hystrix项目在github上托管：
<https://github.com/Netflix/Hystrix/>
 - ▶ 可以参考学习



How To Use

Matt Jacobs edited this page on 4 Jul 2017 · 48 revisions

Contents

1. "Hello World!"
2. Synchronous Execution
3. Asynchronous Execution
4. Reactive Execution
5. Reactive Commands
6. Fallback

CONTENTS

目录

01

Hystrix简介

02

使用Hystrix实现容错

03

使用Hystrix Dashboard监控数据

04

使用Turbine监控集群数据

Hystrix服务熔断

- ❖ 举个例子解释，生活中每家每户都在用电，小明家的电线因为故障导致了小明家停电了。而小李、小张家的电是正常使用的。电力公司没有因为小明家有故障线路而停掉其他人家的电，同时小明家没有使用有故障的电路的电。这时即为熔断。
- ❖ 熔断机制是应对雪崩效应的一种微服务链路保护机制。
- ❖ 当某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回"错误"的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。
- ❖ 熔断的目的是当A服务模块中的某块程序出现故障后为了不影响其他客户端的请求而做出的及时回应。
- ❖ 在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。

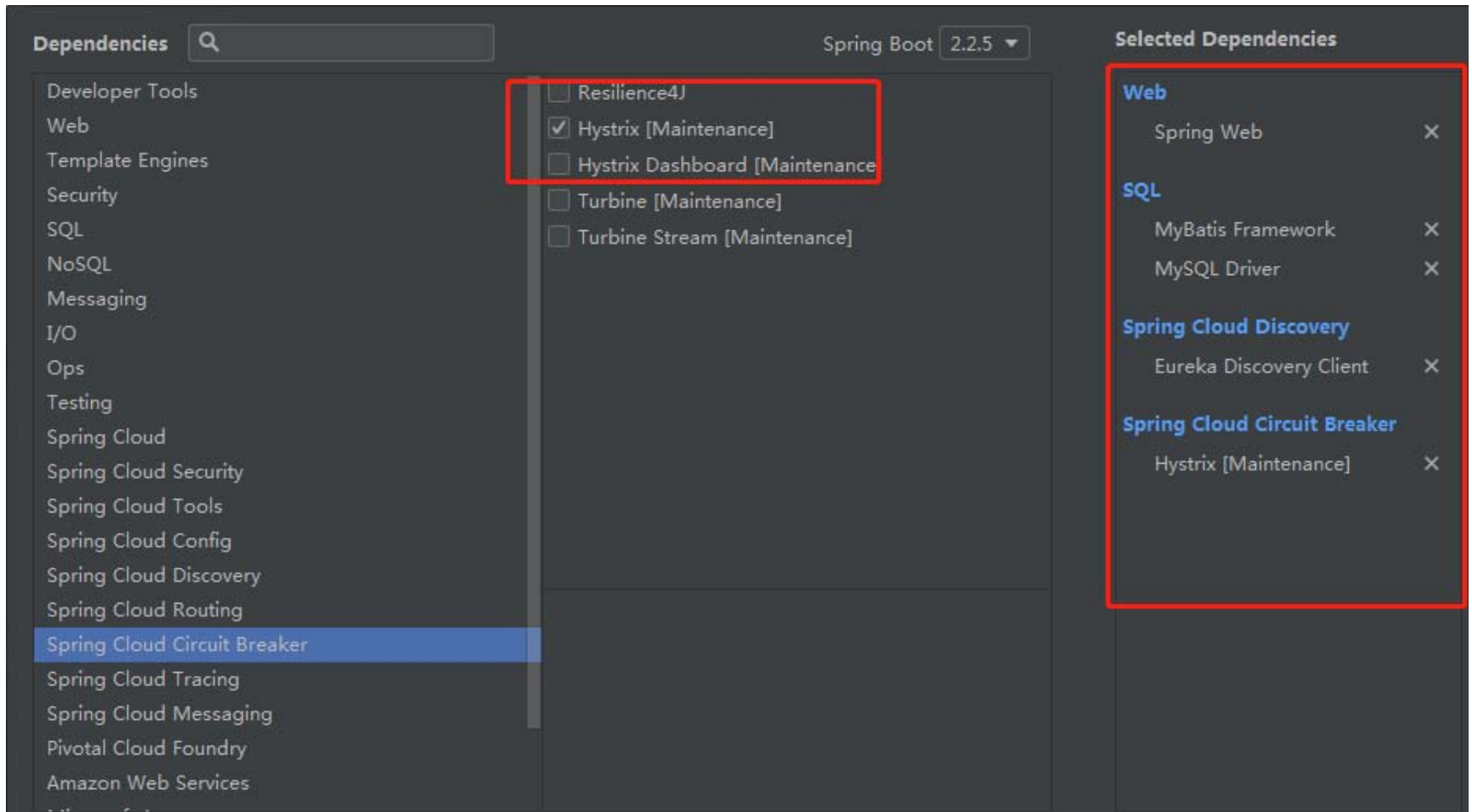
Hystrix服务熔断

❖ 实现步骤

- ▶ 新建Breaker工程
- ▶ 修改application.yml
- ▶ 修改Controller
- ▶ 修改主启动类，添加@EnableCircuitBreaker

Hystrix服务熔断

❖ 新建工程



Hystrix服务熔断

❖ 修改全局配置文件（application.yml）

```
spring:
  application:
    name: provider
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8&serverTimezone=GMT%2B8
    username: root
    password: 123456
  server:
    port: 8001
  eureka:
    client:
      service-url:
        defaultZone:
http://eureka.com:7001/eureka/, http://eureka2.com:7002/eureka/, http://eureka3.com:7003/eureka/
    instance:
      prefer-ip-address: true
      instance-id: user-provider
  info:
    app.name: provider
    company.name: neusoft
    author.name: ykp
  logging:
    level:
      root: INFO
```

Hystrix服务熔断

❖ 修改Controller

- ▶ 添加@HystrixCommand, 说明方法报异常后, 该如何处理
- ▶ 追加报异常后的处理方法

```
@HystrixCommand(fallbackMethod = "fallback")
@RequestMapping("/user/findUserById/{id}")
public User findUserById(@PathVariable("id") int id) throws
Exception {
    User user = userService.findUserById(id);
    if(user == null) throw new Exception();
    else return user;
}

public User fallback(@PathVariable("id") int id) {
    User user = new User();
    user.setId(id);
    user.setLoginName("The user id " + id + " is not found!");
    user.setDbSource("no this data in Database");
    return user;
}
```


Hystrix服务熔断

❖ 主启动类

- ▶ 修改类名
- ▶ 添加@EnableCircuitBreaker，对hystrixR熔断机制的支持

@SpringBootApplication

@EnableEurekaClient

@EnableDiscoveryClient

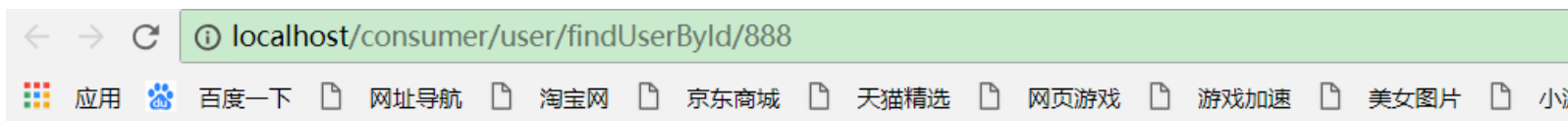
@EnableCircuitBreaker

@MapperScan("edu.neu.springboot.breaker.repository")

测试服务熔断

❖ 测试

- ▶ 先启动3个eureka集群
- ▶ 再启动breaker
- ▶ 浏览器<http://localhost/consumer/user/findUserById/888>



```
{"id":888,"loginName":"该ID: 888没有没有对应的用户","username":null,"password":null,"dbSource":"no this database in MySQL"}
```

测试服务熔断

❖ 结论

- ▶ 当某个服务单元发生故障之后，通过断路器的故障监控，向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常。
- ▶ 断路器的基本作用就是@HystrixCommand注解的方法失败后，系统将自动切换到fallbackMethod方法执行。

Hystrix服务降级

- ❖ 举个例子解释，我们去银行排队办理业务，大部分的银行分为普通窗口、特殊窗口（VIP窗口，老年窗口）。某一天银行大厅排普通窗口的人巨多。这时特殊窗口贴出告示说某时刻之后再开放。那么这时特殊窗口的工作人员就可以空出来去帮其他窗口办理业务，提高办事效率，已达到解决普通窗口排队的人过的目的。这时即为降级。
- ❖ 当整体资源快不够用了，将某些服务先关掉，当资源够用的时候，再把关闭的服务开启回来。
- ❖ 降级的目的是为了解决整体项目的压力，而牺牲掉某一服务模块而采取的措施。
- ❖ 服务的降级处理是在客户端实现的，与服务器端没有关系。

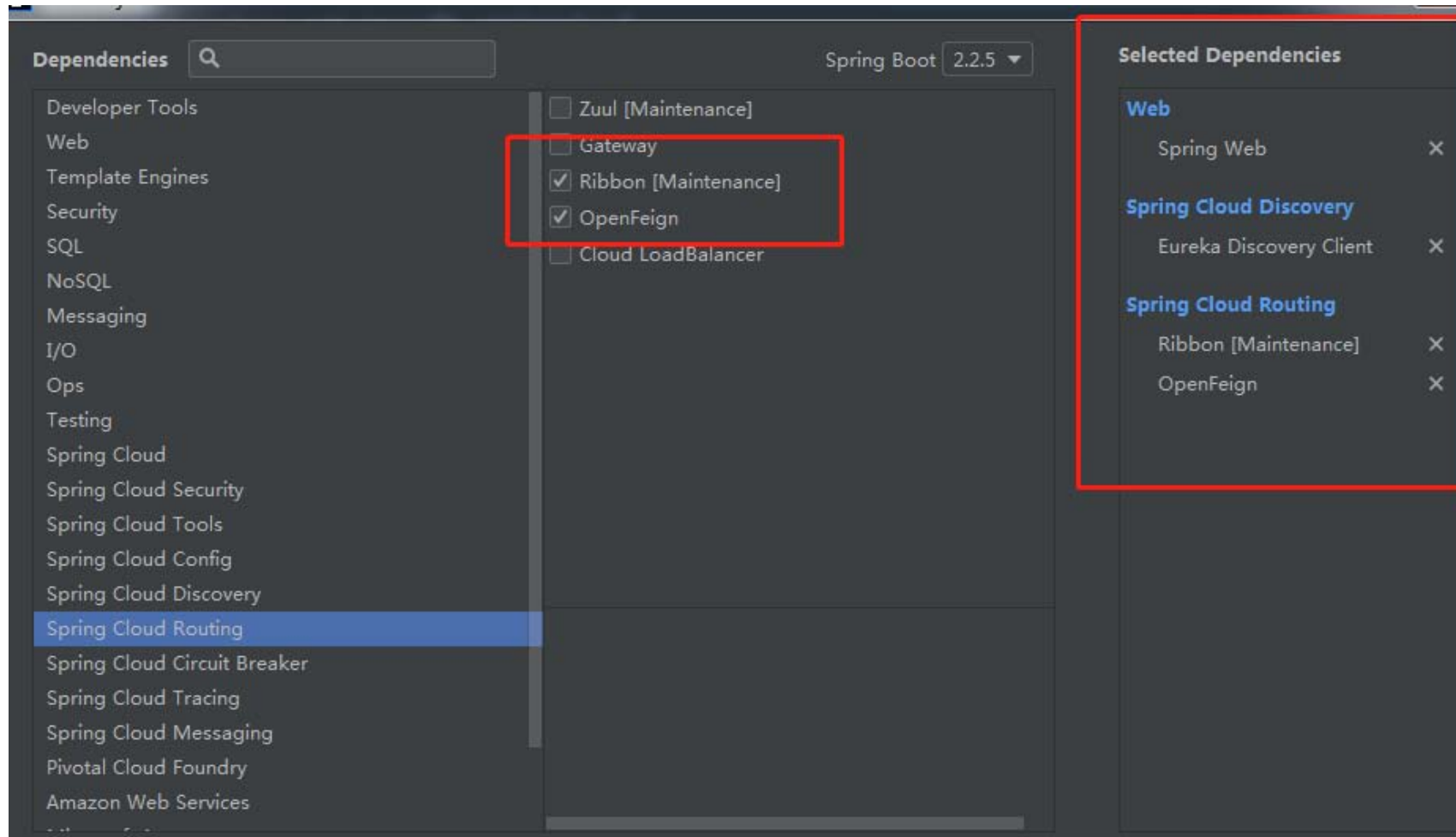
Hystrix服务降级

❖ 实现步骤

- ▶ 新建Fallback工程（参考Feign工程）
- ▶ 新建一个实现了FallbackFactory接口的类
- ▶ 修改UserService接口
- ▶ 修改全局配置文件

Hystrix服务降级

❖ 新建Fallback工程（参考Feign工程）



Hystrix服务降级

- ❖ 新建一个实现了FallbackFactory接口的类
 - ▶ 接口的实现方法为对应方法发生异常时的处理

@Component

```
public class UserServiceFallbackFactory implements FallbackFactory {
```

```
    @Override
```

```
    public Object create(Throwable throwable) {
```

```
        return new UserService() {
```

```
            @Override
```

```
            public List<User> findAll() {
```

```
                return null;
```

```
            }
```

```
            @Override
```

```
            public User findUserById(int id) {
```

```
                User user = new User();
```

```
                user.setId(id);
```

```
                user.setLoginName("The user id " + id + " is not found!");
```

```
                user.setDbSource("no this data in Database");
```

```
                return user;
```

```
            }
```

```
        };
```

```
    }
```

```
}
```

Hystrix服务降级

- ❖ 修改UserService接口，在注解@FeignClient中添加fallbackFactory属性值
 - ❖ 表明出现异常时，由fallbackFactory属性指定的类来处理

```
@FeignClient(value = "provider", fallbackFactory =  
edu.neu.springboot.fallback.config.UserServiceFallbackFactory.class)  
@Service  
public interface UserService  
{  
    @RequestMapping(value = "/user/findAll", method = RequestMethod.GET)  
    List<User> findAll();  
  
    @RequestMapping(value = "/user/findUserById/{id}", method =  
RequestMethod.GET)  
    User findUserById(@PathVariable("id") int id);  
}
```


Hystrix服务降级

❖ 修改全局配置文件

server:

port: 80

spring:

application:

name: fallback

eureka:

client:

service-url:

defaultZone: http://eureka.com:7001/eureka/

register-with-eureka: false

feign:

hystrix:

enabled: true

测试服务降级

❖ 测试

- ▶ 先启动3个eureka集群
- ▶ 再启动一个provider
- ▶ 最后启动fallback
- ▶ 正常访问测试 `http://localhost/consumer/user/findUserById/1`



```
{"id":1,"loginName":"user111","username":"张三","password":"123456","dbSource":"mybatis"}
```

测试服务降级

❖ 测试

- ▶ 故意关闭微服务provider
- ▶ 客户端自己调用提示错误信息
 - ★ 浏览器<http://localhost/consumer/user/findUserById/1>



{ "id":1, "loginName": "该ID: 1没有对应的用户", "username": null, "password": null, "dbSource": "no this data in MySQL" }

- ▶ 此时服务端provider已经down了，但是我们做了服务降级处理，让客户端在服务端不可用时也会获得提示信息而不会挂起耗死服务器

服务熔断与服务降级的区别

- ❖ 两者其实从某些角度看是有一定的类似性的：
 - ▶ 目的很一致，都是从可用性可靠性着想，为防止系统的整体缓慢甚至崩溃，采用的技术手段；
 - ▶ 最终表现类似，对于两者来说，最终让用户体验到的是某些功能暂时不可达或不可用；
 - ▶ 粒度一般都是服务级别，当然，业界也有不少更细粒度的做法，比如做到数据持久层（允许查询，不允许增删改）；
 - ▶ 自治性要求很高，熔断模式一般都是服务基于策略的自动触发，降级虽说可人工干预，但在微服务架构下，完全靠人显然不可能，开关预置、配置中心都是必要手段；

服务熔断与服务降级的区别

❖ 而两者的区别也是明显的：

- ▶ 触发原因不太一样，服务熔断一般是某个服务（下游服务）故障引起，而服务降级一般是从整体负荷考虑；
- ▶ 管理目标的层次不太一样，熔断其实是一个框架级的处理，每个微服务都需要（无层级之分），而降级一般需要对业务有层级之分（比如降级一般是从最外围服务开始）

CONTENTS

目录

01

Hystrix简介

02

使用Hystrix实现容错

03

使用Hystrix Dashboard监控数据

04

使用Turbine监控集群数据

Hystrix Dashboard监控数据

- ❖ Hystrix Dashboard是Hystrix的一个组件，Hystrix Dashboard提供一个断路器的监控面板，可以使我们更好的监控服务和集群的状态，仅仅使用Hystrix Dashboard只能监控到单个断路器的状态，实际开发中还需要结合Turbine使用。
- ❖ Spring Cloud Hystrix Dashboard的底层原理是间隔一定时间去“Ping”目标服务，返回的结果是最新的监控数据，最后将数据显示出来。

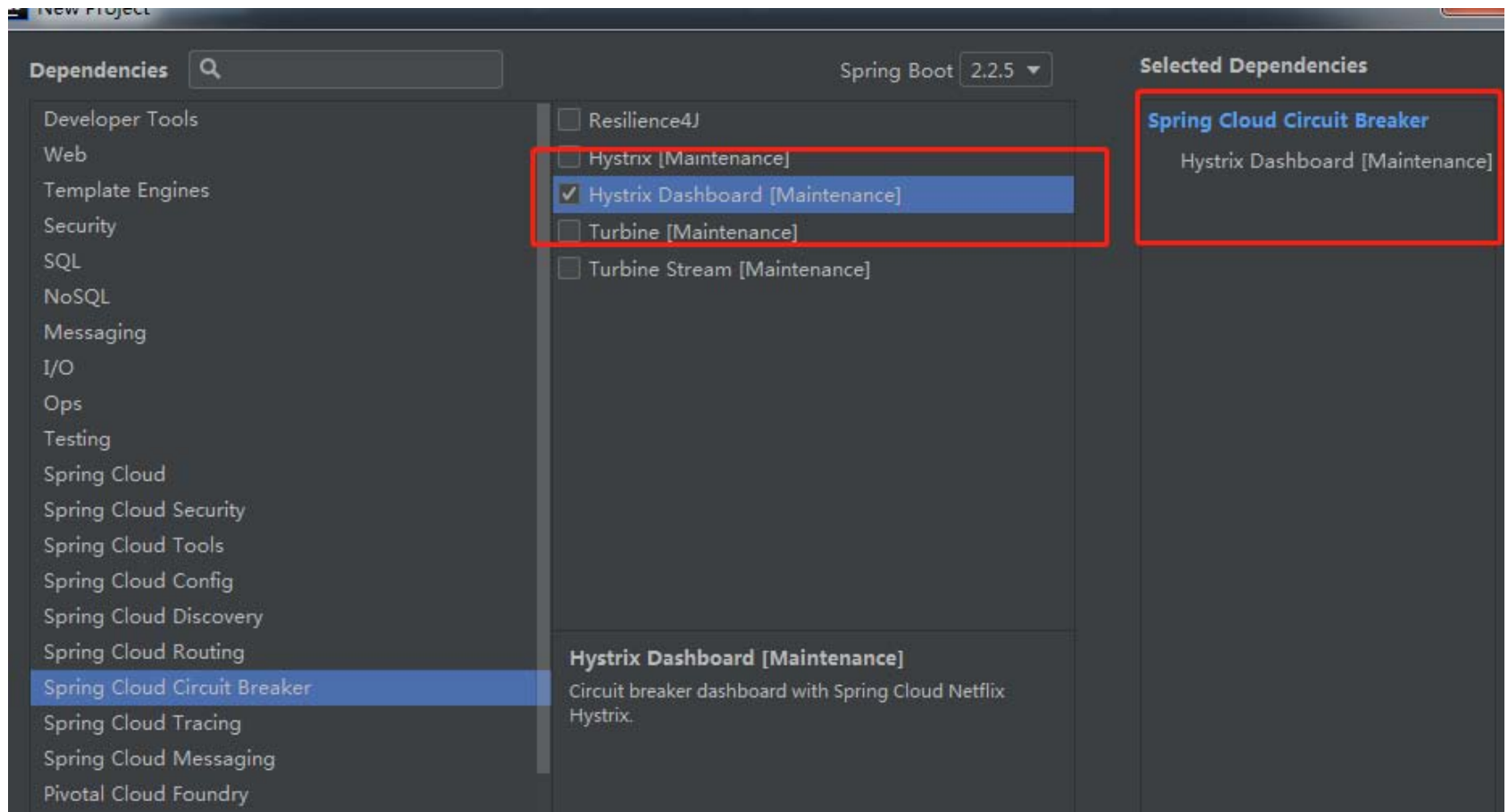
Hystrix Dashboard监控数据

❖ 实现步骤

- ▶ 新建dashboard工程
- ▶ 全局配置文件
- ▶ 修改主启动类
- ▶ 参照eureka工程的provider模块新建provider1、2、3，3个模块，所有Provider微服务都需要监控依赖配置
- ▶ 测试

Hystrix Dashboard监控数据

❖ 新建dashboard工程



Hystrix Dashboard监控数据

❖ 修改全局配置文件

spring:

application:

name: dashboard

server:

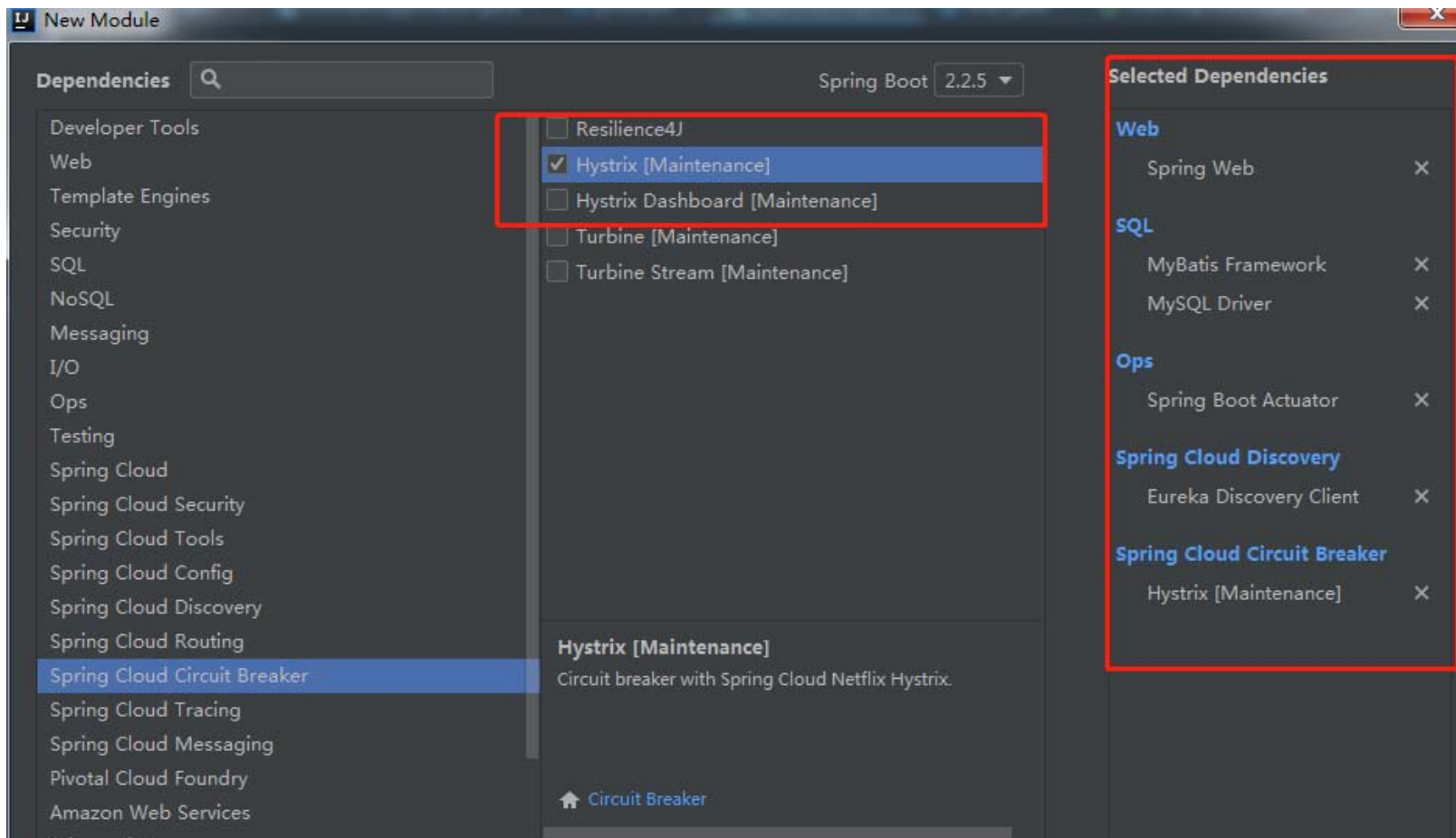
port: 9001

Hystrix Dashboard监控数据

- ❖ 修改主启动类
 - ▶ 添加新注解@EnableHystrixDashboard
 - @SpringBootApplication
 - @EnableHystrixDashboard

Hystrix Dashboard监控数据

❖ 参照eureka工程的provider模块新建provider1、2、3，3个模块



Hystrix Dashboard监控数据

❖ Provider模块添加监控依赖

▶ 修改主启动类

★ 创建ServletRegistrationBean

```
@SpringBootApplication
@EnableEurekaClient
@MapperScan("edu.neu.springboot.provider.repository")
@EnableDiscoveryClient
@EnableCircuitBreaker//对hystrix熔断机制的支持
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
    // 此配置是为了服务监控而配置，与服务容错本身无关，
    // ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
    // 只要在自己的项目里配置上下面的servlet就可以了
    @Bean
    public ServletRegistrationBean getServlet() {
        HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
        ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);
        registrationBean.setLoadOnStartup(1);
        registrationBean.addUrlMappings("/hystrix.stream");
        registrationBean.setName("HystrixMetricsStreamServlet");
        return registrationBean;
    }
}
```

Hystrix Dashboard监控数据

❖ Provider模块添加监控依赖

- ▶ 参考breaker工程的Controller修改provider模块的Controller

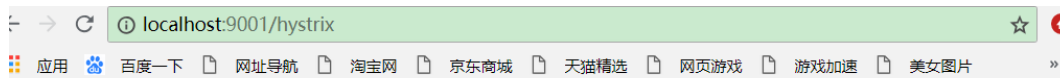
```
@HystrixCommand(fallbackMethod = "fallback")
@RequestMapping("/user/findUserById/{id}")
public User findUserById(@PathVariable("id") int id) throws Exception {
    User user = userService.findUserById(id);
    if(user == null) throw new Exception();
    else return user;
}

public User fallback(@PathVariable("id") int id) {
    User user = new User();
    user.setId(id);
    user.setLoginName("The user id " + id + " is not found!");
    user.setDbSource("no this data in Database");
    return user;
}
```

Hystrix Dashboard监控数据

❖ 测试

- ▶ 启动dashboard
- ▶ 浏览器输入http://localhost:9001/hystrix
 - ★ 进入监控面板主页面
 - ★ 此时没有具体的监控信息，需要输入要监控的消费者地址及监控信息的轮询时间和标题



Hystrix Dashboard

Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream

Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]

Single Hystrix App: http://hystrix-app:port/hystrix.stream

Delay: ms Title:

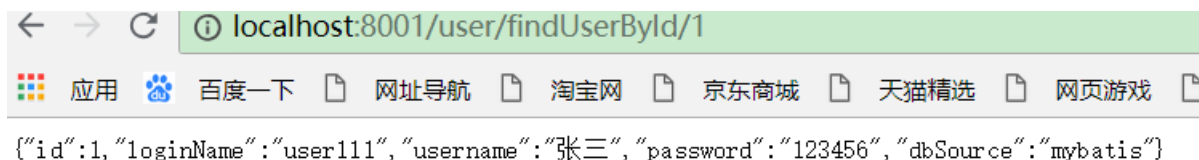
Hystrix Dashboard监控数据

- ❖ Hystrix Dashboard共支持三种不同的监控方式：
 - ▶ 单体Hystrix 消费者：
 - ★ 通过`http://hystrix-app:port/hystrix.stream`开启，实现对具体某个服务实例的监控。
 - ▶ 默认集群监控：
 - ★ 通过`http://turbine-hostname:port/turbine.stream`开启，实现对默认集群的监控。
 - ▶ 自定义集群监控：
 - ★ 通过`http://turbine-hostname:port/turbine.stream?cluster=[clusterName]`开启，实现对`clusterName`集群的监控。
- ❖ 此处讲述对单体Hystrix 消费者的监控，后面整合Turbine集群后再说明后集群的监控方式。

服务监控测试

❖ 步骤

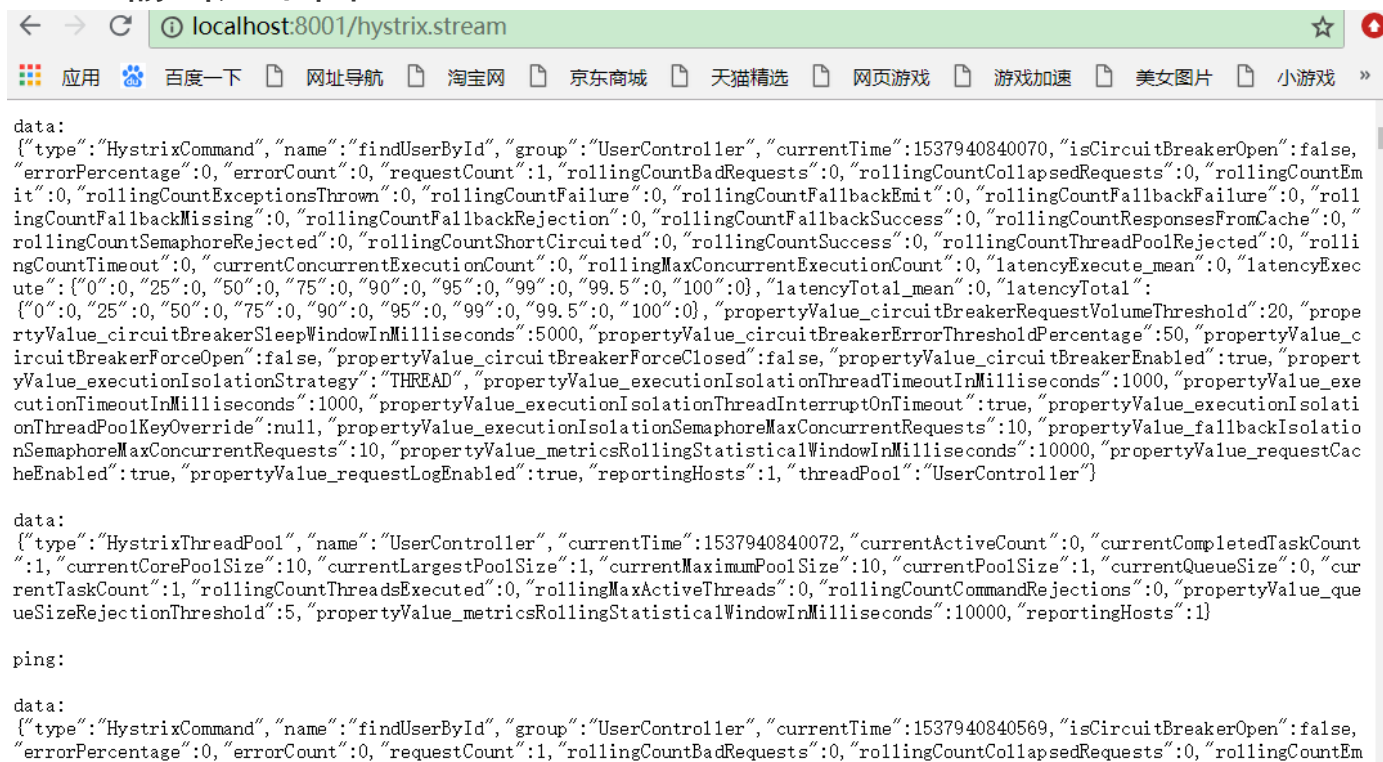
- ▶ 启动3个eureka集群
- ▶ 启动provider1模块
- ▶ 浏览器
 - ★ <http://localhost:8001/user/findUserById/1>
- ▶ 观察监控窗口



服务监控测试

❖ 浏览器直接访问<http://localhost:8001/hystrix.stream>

▶ 输出如下图:



```
data:
{"type":"HystrixCommand","name":"findUserById","group":"UserController","currentTime":1537940840070,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmitted":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":0,"latencyExecute_mean":0,"latencyExecute_min":0,"latencyExecute_max":0,"latencyTotal_mean":0,"latencyTotal_min":0,"latencyTotal_max":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"UserController"}

data:
{"type":"HystrixThreadPool","name":"UserController","currentTime":1537940840072,"currentActiveCount":0,"currentCompletedTaskCount":1,"currentCorePoolSize":10,"currentLargestPoolSize":1,"currentMaximumPoolSize":10,"currentPoolSize":1,"currentQueueSize":0,"currentTaskCount":1,"rollingCountThreadsExecuted":0,"rollingMaxActiveThreads":0,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}

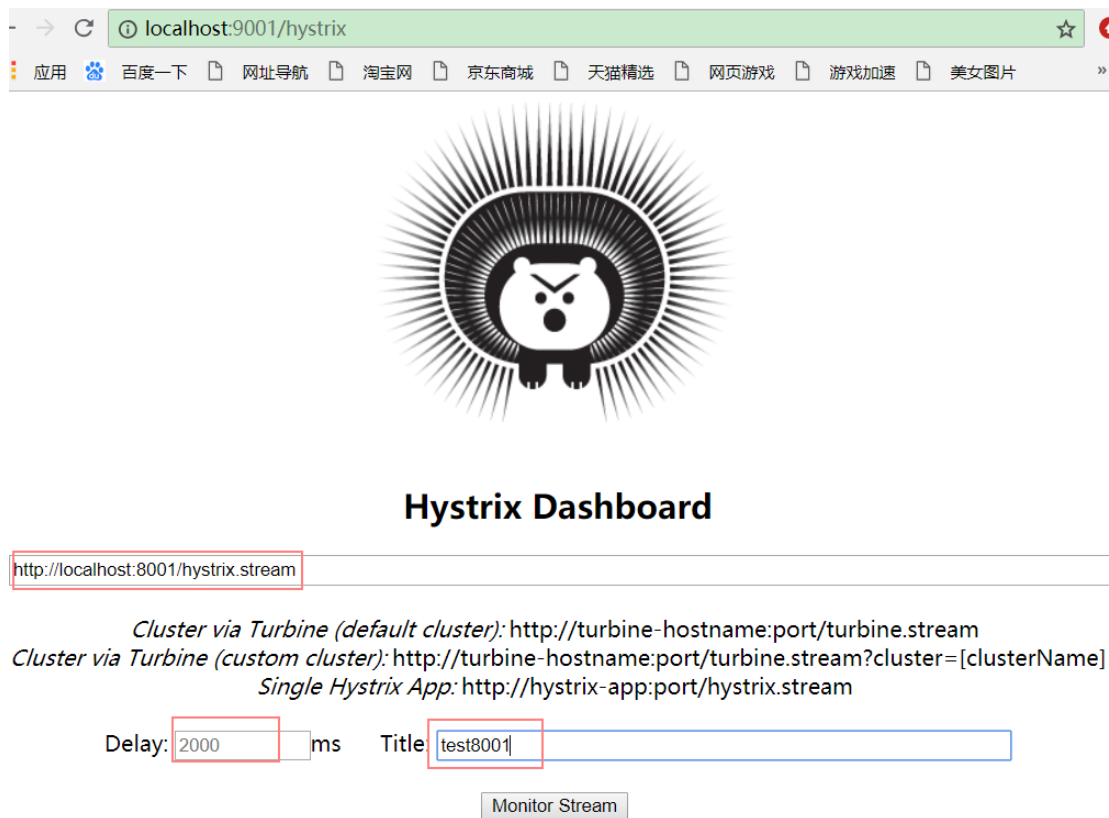
ping:
```

```
data:
{"type":"HystrixCommand","name":"findUserById","group":"UserController","currentTime":1537940840569,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmitted":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":0,"latencyExecute_mean":0,"latencyExecute_min":0,"latencyExecute_max":0,"latencyTotal_mean":0,"latencyTotal_min":0,"latencyTotal_max":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"UserController"}
```

❖ Spring Cloud Hystrix Dashboard是间隔一定时间去“Ping”目标服务，返回的结果是最新的监控数据

服务监控测试

- ❖ 上图的监控数据以图形方式显示：
 - ▶ 需要输入要监控的消费者地址及监控信息的轮询时间和标题
 - ▶ 点击 Monitor Stream后，进入监控窗口



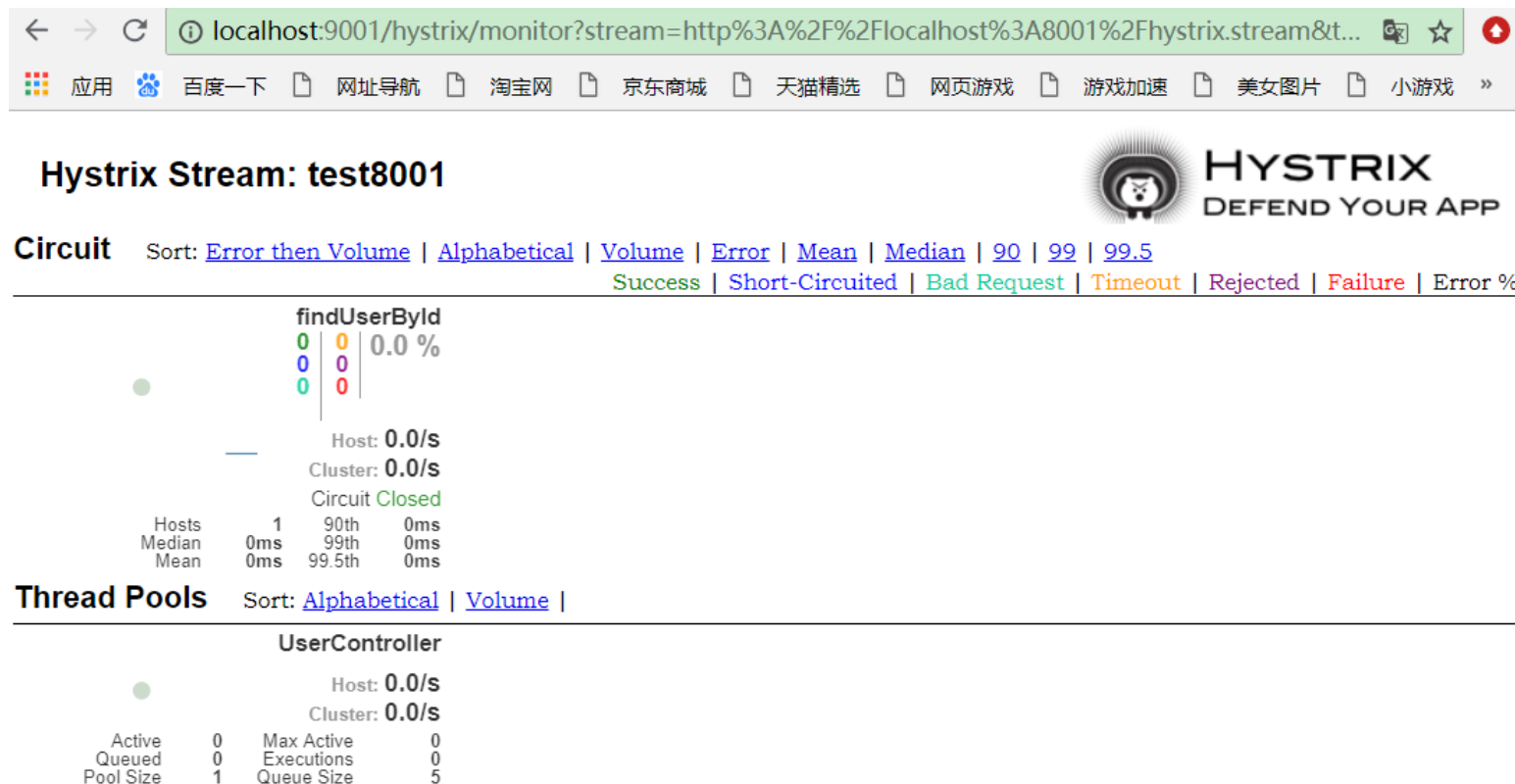
服务监控测试

❖ 输入项说明

- ▶ 要监控的消费者地址 (<http://localhost:8001/hystrix.stream>)
- ▶ 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, 可以通过配置该属性来降低客户端的网络和CPU消耗。
- ▶ Title: 该参数对应了头部标题Hystrix Stream之后的内容, 默认会使用具体监控实例的URL, 可以通过配置该信息来展示更合适的标题。

服务监控测试

- ▶ 随便调用一个被hystrix管理的远程调用接口后，页面会刷新出类似如下的页面。

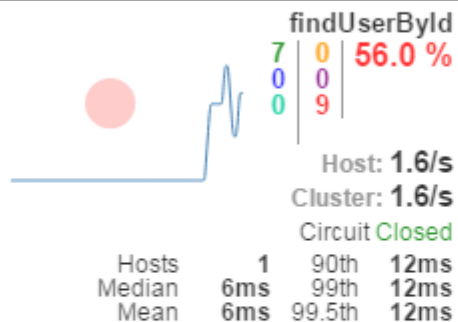


服务监控测试

- 多次调用provider (`http://localhost:8001/user/findUserById/1`)
访问存在以及不存在数据

Hystrix Stream: provider8001

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)



Thread Pools Sort: [Alphabetical](#) | [Volume](#) |



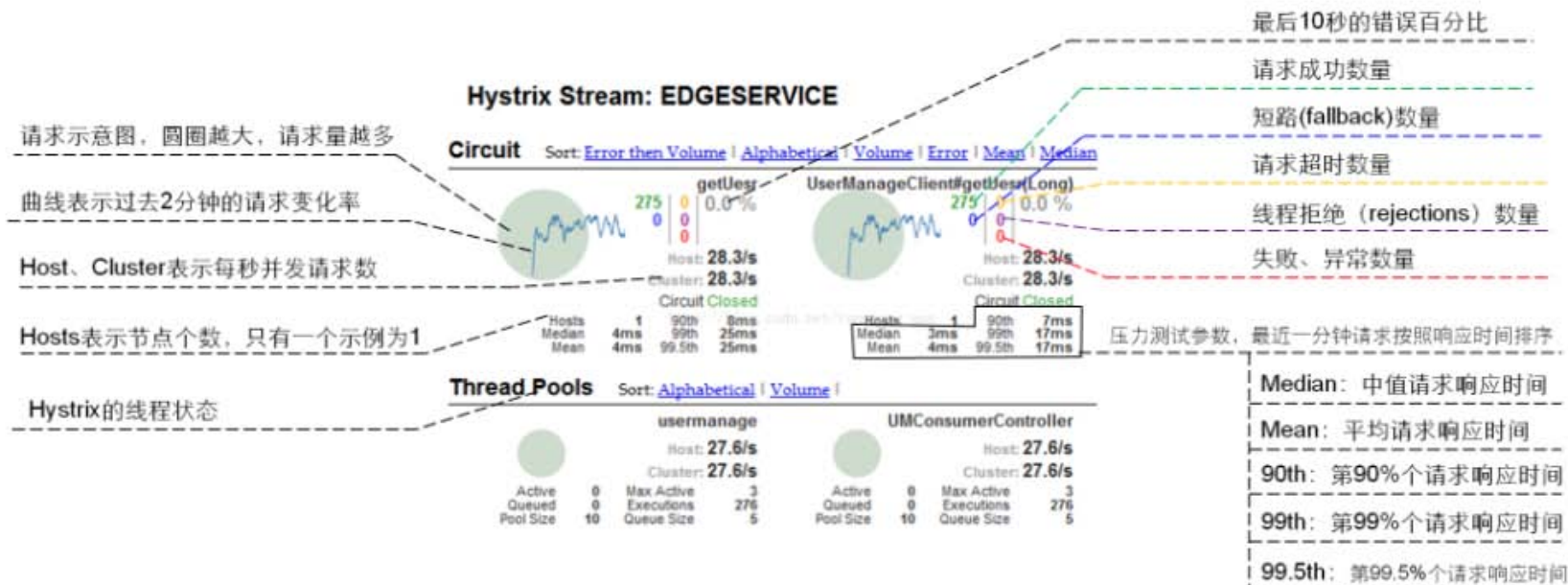
服务监控测试

❖ 观察监控窗口

- ▶ 7个颜色：
- ▶ 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<黄色<橙色<红色递减。
 - ★ 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。

服务监控测试

❖ hystrix dashboard界面监控参数

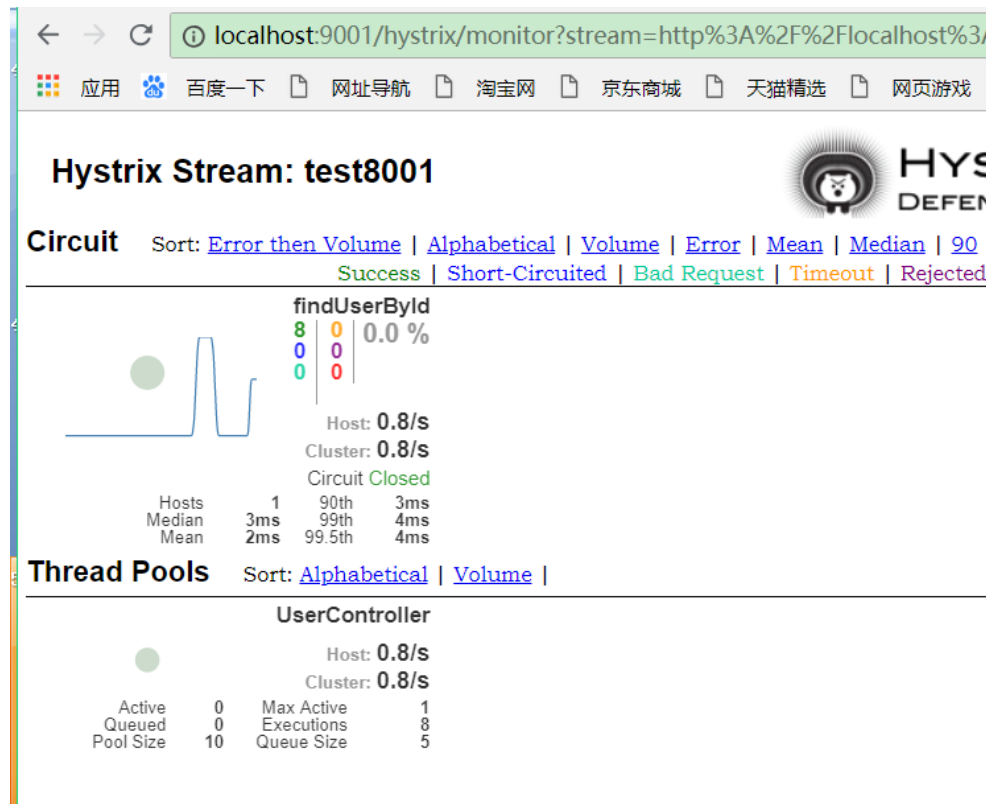
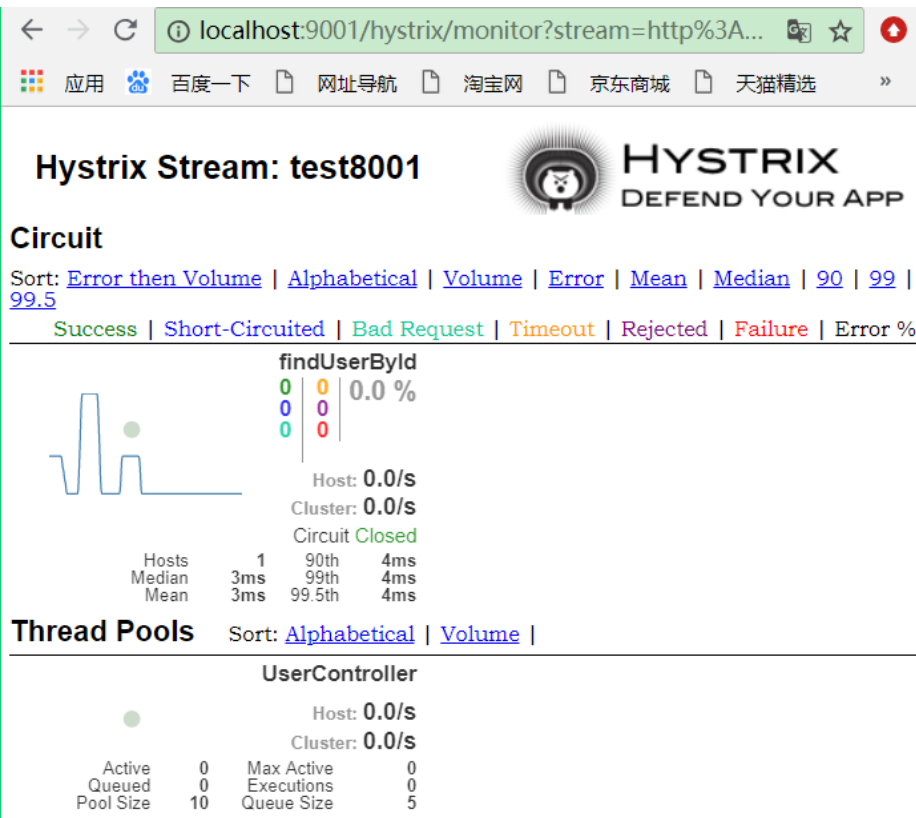


服务监控测试

▶ 浏览器

★ 多次刷新 <http://localhost:8001/user/findUserById/1>

▶ 观察监控窗口



CONTENTS

目录

01

Hystrix简介

02

使用Hystrix实现容错

03

使用Hystrix Dashboard监控数据

04

使用Turbine监控集群数据

Turbine监控集群数据

- ❖ 在复杂的分布式系统中，相同服务的结点经常需要部署上百甚至上千个，很多时候，运维人员希望能够把相同服务的节点状态以一个整体集群的形式展现出来，这样可以更好的把握整个系统的状态。
- ❖ 为此，Netflix提供了一个开源项目（Turbine）来提供把多个 `hystrix.stream` 的内容聚合为一个数据源供Dashboard展示。
- ❖ Turbine项目在github上托管：<https://github.com/Netflix/Turbine>
- ❖ 使用Spring Cloud Turbine实现集群的监控。

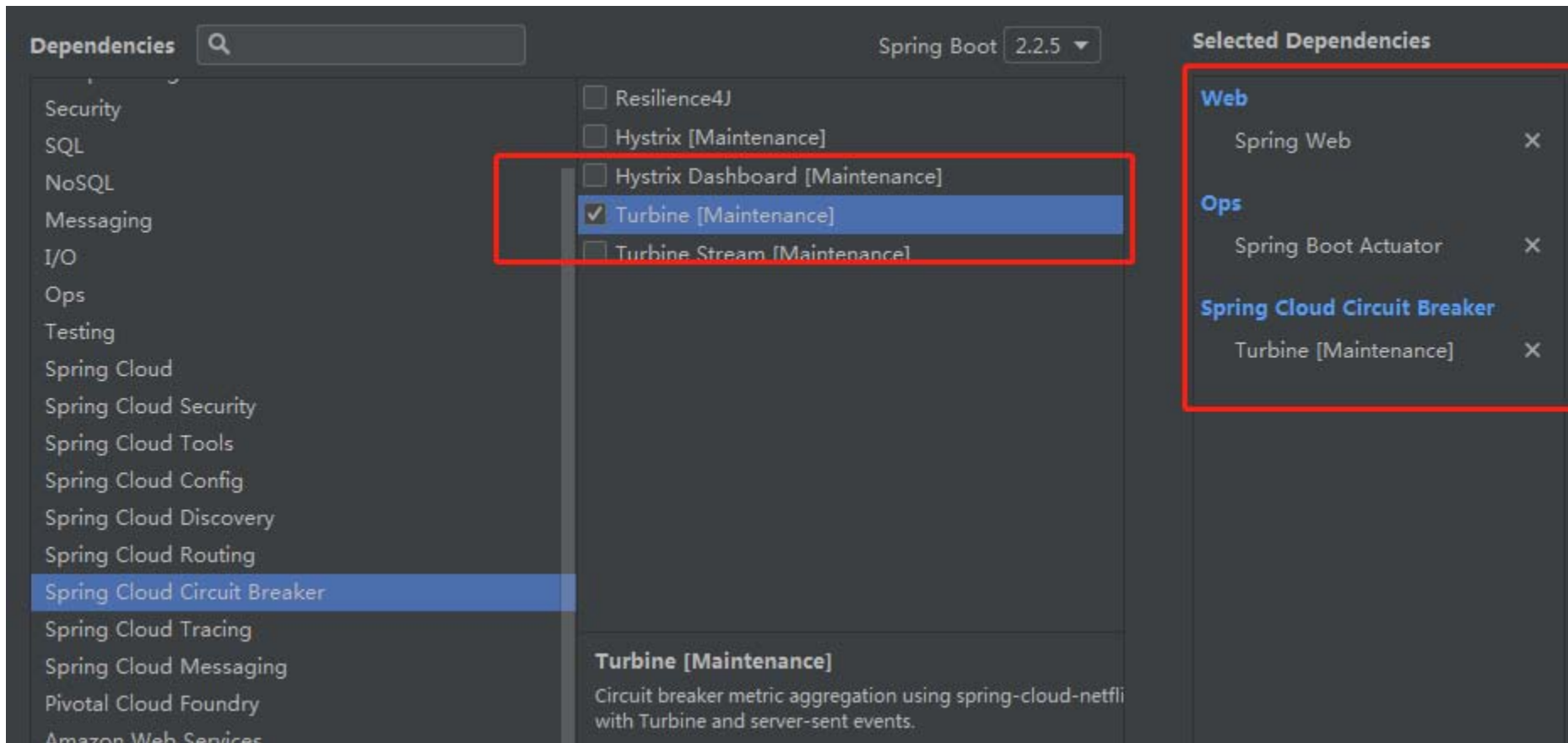
Turbine监控集群数据

❖ 实现步骤

- ▶ 新建turbine工程
- ▶ 修改全局配置文件
- ▶ 修改主启动类，添加注解@EnableTurbine

Turbine监控集群数据

❖ 新建turbine工程



Turbine监控集群数据

❖ 全局配置文件

```
spring:
  application:
    name: turbine
server:
  port: 9002
eureka:
  client:
    service-url:
      defaultZone:
http://eureka.com:7001/eureka/, http://eureka2.com:7002/eureka/, http://eureka3.com:7003/eureka/
turbine:
  #指定收集路径
  instanceUrlSuffix:
    default: /hystrix.stream
  #指定了需要收集监控信息的服务名, 多个以 “,” 进行区分
  app-config: provider
  # 指定集群名称, 若为default则为默认集群, 多个集群则通过此配置区分
  cluster-name-expression: "'default'"
  #此配置默认为false, 则服务是以host进行区分, 若设置为true则以host+port进行区分
  combine-host-port: true
```

Turbine监控集群数据

❖ 主启动类

- ▶ 使用@EnableTurbine注解开启Turbine的监控数据集群配置
@SpringBootApplication
@EnableDiscoveryClient
@EnableTurbine

集群监控测试

❖ 步骤

- ▶ 启动3个eureka集群
- ▶ 启动dashboard工程的provider1、2、3模块
- ▶ 启动turbine
- ▶ 浏览器确认provider及监控数据
 - ★ <http://localhost:8001/user/findUserById/1>
 - ★ <http://localhost:8002/user/findUserById/1>
 - ★ <http://localhost:8003/user/findUserById/1>
 - ★ <http://localhost:8001/hystrix.stream>
 - ★ <http://localhost:8002/hystrix.stream>
 - ★ <http://localhost:8003/hystrix.stream>


```
{"id":1,"loginName":"user111","username":"zhangsan","password":"123456","dbSource":"mybatis"}
```

localhost:8001/hystrix.stream

```
data:
{"type":"HystrixCommand","name":"findUserById","group":"UserController","currentTime":1584004308451,"isCircuitBreaker":0,"rollingCountCollapsedRequests":0,"rollingCountEmitted":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountThreadLocalRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":{"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_circuitBreakerRequestVolumeThresholdErrorThresholdPercentAge":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForcedOnStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimePropertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentValue.metricsRollingStatisticalWindowInMilliseconds":1000,"propertyValue_requestCacheEnabled":true,"propertyValue_re
```

```
data:
{"type":"HystrixThreadPool","name":"UserController","currentTime":1584004308451,"currentActiveCount":0,"currentCompl
1Size":10,"currentPoolSize":10,"currentQueueSize":0,"currentTaskCount":35,"rollingCountThreadsExecuted":0,"rollingMe
shold":5,"propertyValue metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}
```

ping:

```
{
    "type": "HystrixCommand",
    "name": "findUserById",
    "group": "UserController",
    "currentTime": 1584004308952,
    "isCircuitBreakerOpen": true,
    "rollingCountCollapsedRequests": 0,
    "rollingCountEmitted": 0,
    "rollingCountExceptionsThrown": 0,
    "rollingCountFailure": 0,
    "rollingCountFallbackRejection": 0,
    "rollingCountFallbackSuccess": 0,
    "rollingCountResponsesFromCache": 0,
    "rollingCountThreadPoolRejected": 0,
    "rollingCountTimeout": 0,
    "currentConcurrentExecutionCount": 0,
    "rollingMaxConcurrentExecutionCount": 0,
    "latencyTotal_mean": 0,
    "latencyTotal": 0,
    "propertyValue_circuitBreakerForceOpen": false,
    "propertyValue_circuitBreakerRequestVolumeThresholdErrorThresholdPercentage": 50,
    "propertyValue_executionIsolationThreadInheritContext": true,
    "propertyValue_executionIsolationThreadInheritContextStrategy": "THREAD",
    "propertyValue_executionIsolationThreadInheritContextTimeoutInMilliseconds": 1000,
    "propertyValue_executionIsolationSemaphoreMaxConcurrentRequests": 10,
    "propertyValue_executionIsolationThreadKeyOverride": null,
    "propertyValue_executionIsolationSemaphoreMaxConcurrency": 10
}
```

集群监控测试

❖ 查看注册的服务

DS Replicas

eureka2.com

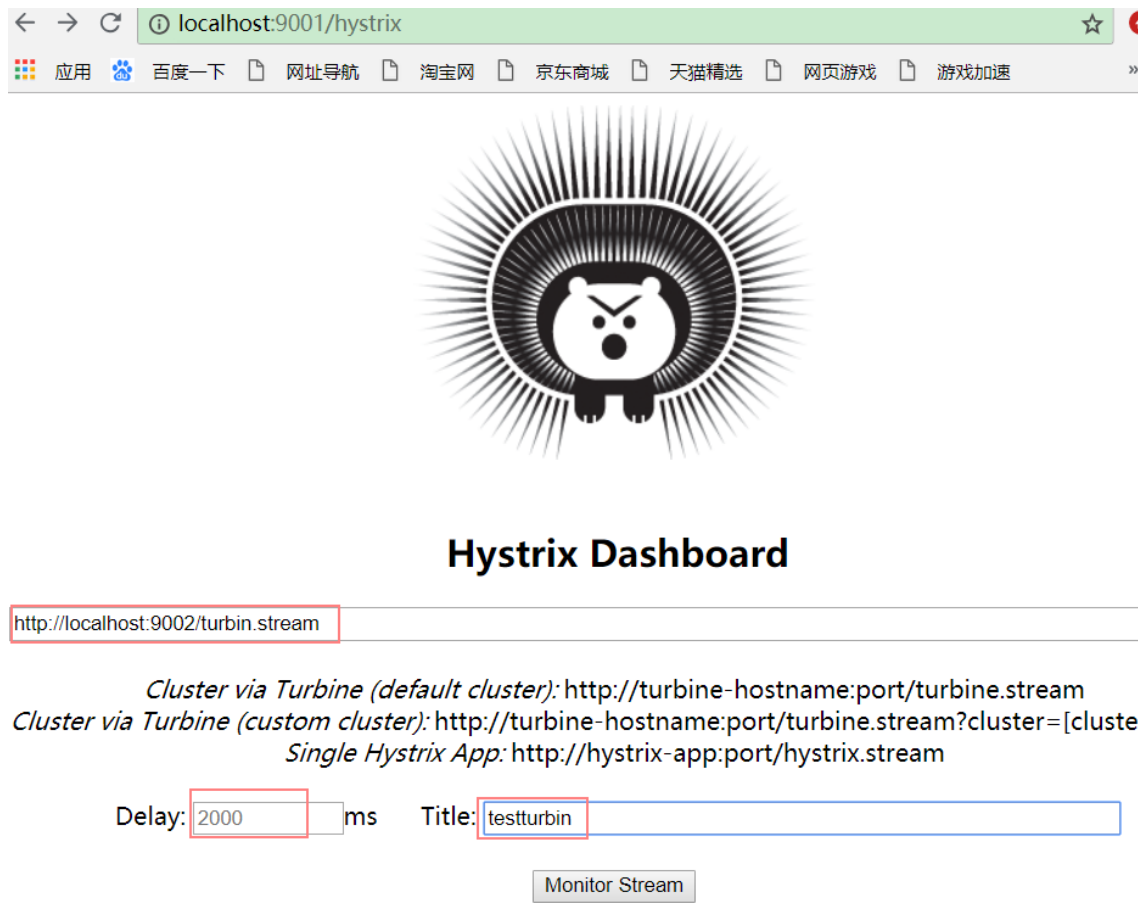
eureka3.com

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PROVIDER	n/a (3)	(3)	UP (3) - user-provider1 , user-provider2 , user-provider3
TURBINE	n/a (1)	(1)	UP (1) - Admin-18906:turbine:9002

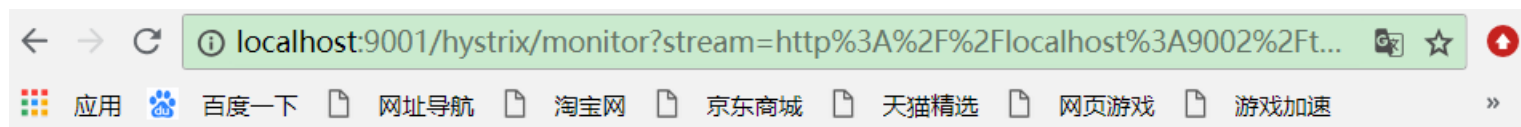
集群监控测试

- ❖ 在浏览器输入http://localhost:9001/hystrix进入监控面板主页面
 - ▶ 输入http://localhost:9002/turbine.stream进入默认集群面板页



集群监控测试

❖ 默认集群面板监控



Hystrix Stream: testturbin



HYSTRIX
DEFEND YOUR APP

Circuit

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)



findUserById
10 | 0 | 0.0 %
0 | 0
0 | 0

Host: 0.3/s

Cluster: 0.5/s

Circuit **Closed**

Hosts	2	90th	1ms
Median	1ms	99th	1ms
Mean	1ms	99.5th	1ms

Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |

UserController

Host: 0.3/s

Cluster: 0.5/s

Active	0	Max Active	2
Queued	0	Executions	10
Pool Size	20	Queue Size	5

本章重点总结

- ❖ 了解分布式系统面临的问题、雪崩效应；
- ❖ 了解Hystrix简介；
- ❖ 理解服务容错、服务熔断与服务降级的区别；
- ❖ 掌握服务熔断；
- ❖ 掌握服务降级；
- ❖ 掌握Hystrix Dashboard监控数据；
- ❖ 掌握Turbine监控集群数据；

课后作业【必做任务】

- ❖ 1、独立完成课件中的示例

