Table of Contents

使用Spring Cloud和Docker构建微服务

本文主要是对Spring Cloud学习的一些总结，探讨的话题主要有：

1. 什么是微服务
2. 注册中心Eureka
3. 服务提供者
4. 服务消费者
    i. 客户端负载均衡Ribbon
    ii. 简化的Http客户端Feign
5. 熔断器
    i. Hystrix
    ii. Hystrix监控界面Hystrix Dashboard
    iii. Hystrix集群监控工具Turbine
6. 配置中心
7. API Gateway

目前API Gateway已经完成。基于Spring Cloud构建微服务的必要组件已经讲解完成。下一步是讲解Docker的使用，以及如何使用Docker部署Spring Cloud应用。

持续更新。

序言

什么是微服务架构

近年来，在软件开发领域关于微服务的讨论呈现出火爆的局面，有人倾向于在系统设计与开发中采用微服务方式实现软件系统的松耦合、跨部门开发；同时，反对之声也很强烈，持反对观点的人表示微服务增加了系统维护、部署的难度，导致一些功能模块或代码无法复用，同时微服务允许使用不同的语言和框架来开发各个系统模块，这又会增加系统集成与测试的难度，而且随着系统规模的日渐增长，微服务在一定程度上也会导致系统变得越来越复杂。尽管一些公司已经在生产系统中采用了微服务架构，并且取得了良好的效果；但更多公司还是处在观望的态度。

什么是微服务架构呢？简单说就是将一个完整的应用（单体应用）按照一定的拆分规则（后文讲述）拆分成多个不同的服务，每个服务都能独立地进行开发、部署、扩展。服务于服务之间通过注入RESTful api或其他方式调用。大家可以搜索到很多相关介绍和文章。本文暂不细表。

在此推荐两个比较好的博客：

> http://microservices.io/ http://martinfowler.com/articles/microservices.html

Spring Cloud 简介

Spring Cloud是在Spring Boot的基础上构建的，为开发人员提供快速建立分布式系统中的一些常见的模式

> 例如：配置管理（configuration management），服务发现（service discovery），断路器（circuit breakers），智能路由（ intelligent routing），微代理（micro-proxy），控制总线（control bus），一次性令牌（ one-time tokens），全局锁（global locks），领导选举（leadership election），分布式会话（distributed sessions），集群状态（cluster state）。

Spring Cloud 包含了多个子项目：

> 例如：Spring Cloud Config、Spring Cloud Netflix等

Spring Cloud 项目主页：http://projects.spring.io/spring-cloud/

Talk is cheep, show me the code.下面我们将以代码与讲解结合的方式，为大家讲解Spring Cloud中的各种组件。

准备

环境准备：

工具	版本或描述
JDK	1.8
IDE	STS 或者 IntelliJ IDEA
Maven	3.x

主机名配置：

主机名配置（**C:\Windows\System32\drivers\etc\hosts**文件）
127.0.0.1 discovery config-server gateway movie user feign ribbon

主机规划：

项目名称	端口	描述	URL
microservice-api-gateway	8050	API Gateway	详见文章
microservice-config-client	8041	配置服务的客户端	详见文章
microservice-config-server	8040	配置服务	详见文章
microservice-consumer-movie-feign	8020	Feign Demo	/feign/1
microservice-consumer-movie-feign-with-hystrix	8021	Feign Hystrix Demo	/feign/1
microservice-consumer-movie-feign-with-hystrix-stream	8022	Hystrix Dashboard Demo	/feign/1
microservice-consumer-movie-ribbon	8010	Ribbon Demo	/ribbon/1
microservice-consumer-movie-ribbon-with-hystrix	8011	Ribbon Hystrix Demo	/ribbon/1
microservice-discovery-eureka	8761	注册中心	/
microservice-hystrix-dashboard	8030	hystrix监控	/hystrix.stream
microservice-hystrix-turbine	8031	turbine	/turbine.stream
microservice-provider-user	8000	服务提供者	/1

Spring Cloud所有的配置项：

> http://cloud.spring.io/spring-cloud-static/Brixton.SR5/#_appendix_compendium_of_configuration_properties

在进入主题之前，我们首先创建一个父项目（spring-cloud-microservice-study），这样可以对项目中的Maven依赖进行统一的管理。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>microservice-discovery-eureka</module>
        <module>microservice-provider-user</module>
        <module>microservice-consumer-movie-ribbon</module>
        <module>microservice-consumer-movie-feign</module>
        <module>microservice-consumer-movie-ribbon-with-hystrix</
module>
        <module>microservice-consumer-movie-feign-with-hystrix</
module>
        <module>microservice-hystrix-dashboard</module>
        <module>microservice-consumer-movie-feign-with-hystrix-s
tream</module>
        <module>microservice-hystrix-turbine</module>
    </modules>

    <!-- 使用最新的spring-boot版本 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.0.RELEASE</version>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourc
eEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId
>
                <version>Brixton.SR4</version>
```

```
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

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

            </plugin>
        </plugins>
    </build>
</project>
```

关于服务发现

在微服务架构中，服务发现（Service Discovery）是关键原则之一。手动配置每个客户端或某种形式的约定是很难做的，并且很脆弱。Spring Cloud提供了多种服务发现的实现方式，例如：Eureka、Consul、Zookeeper。本文暂时只讲述基于Eureka的服务发现。后续会补上基于Consul和Zookeeper的服务发现。

Eureka Server示例

创建一个Maven工程（microservice-discovery-eureka），并在pom.xml中加入如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-discovery-eureka</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

编写Spring Boot启动程序：通过@EnableEurekaServer申明一个注册中心

```
/**
 * 使用Eureka做服务发现.
 * @author eacdy
 */
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

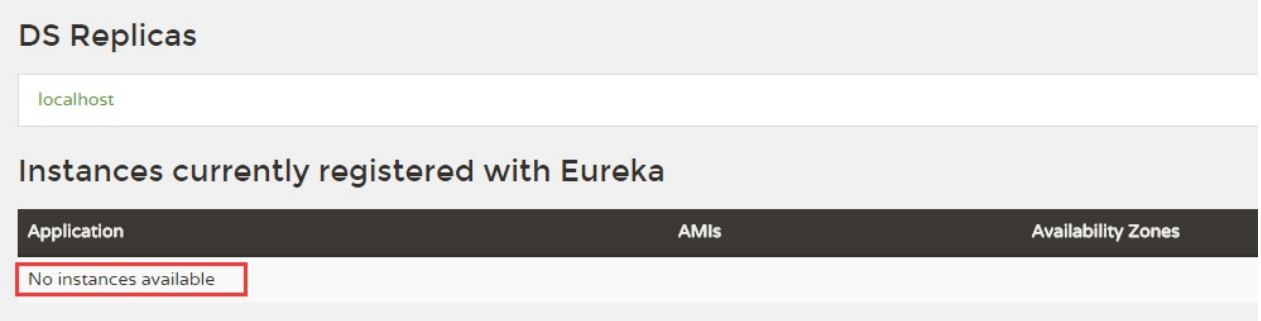
在默认情况下，Eureka会将自己也作为客户端尝试注册，所以在单机模式下，我们需要禁止该行为，只需要在application.yml中如下配置：

```
server:
  port: 8761                    # 指定该Eureka实例的端口

eureka:
  instance:
    hostname: discovery         # 指定该Eureka实例的主机名
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.p
ort}/eureka/

# 参考文档：http://projects.spring.io/spring-cloud/docs/1.0.3/spri
ng-cloud.html#_standalone_mode
# 参考文档：http://my.oschina.net/buwei/blog/618756
```

启动工程后，访问：http://discovery:8761/，如下图。我们会发现此时还没有服务注册到Eureka上面。



代码地址（任选其一）：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-discovery-eureka https://github.com/eacdy/spring-cloud-study/tree/master/microservice-discovery-eureka

服务提供者和服务消费者

下面这张表格，简单描述了服务提供者/消费者是什么：

名词	概念
服务提供者	服务的被调用方（即：为其他服务提供服务的服务）
服务消费者	服务的调用方（即：依赖其他服务的服务）

服务提供者代码示例

这是一个稍微有点复杂的程序。我们使用spring-data-jpa操作h2数据库，同时将该服务注册到注册中心Eureka中。

创建一个Maven工程，并在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-provider-user</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <dependencies>
        <!-- 添加Eureka的依赖 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
```

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

        </dependency>

        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
        </dependency>

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

        </dependency>
    </dependencies>
</project>
```

配置文件：application.yml

```
server:
  port: 8000
spring:
  application:
    name: microservice-provider-user    # 项目名称尽量用小写
  jpa:
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
  datasource:                           # 指定数据源
    platform: h2                        # 指定数据源类型
    schema: classpath:schema.sql        # 指定h2数据库的建表脚本
    data: classpath:data.sql            # 指定h2数据库的insert脚本
logging:
  level:
    root: INFO
    org.hibernate: INFO
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
    com.itmuch.youran.persistence: ERROR
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/    # 指定注册中心
的ip
  instance:
    preferIpAddress: true
```

建表语句：schema.sql

```
drop table user if exists;
create table user (id bigint generated by default as identity, u
sername varchar(255), age int, primary key (id));
```

插库语句：data.sql

```
insert into user (id, username, age) values (1,'Tom',12);
insert into user (id, username, age) values (2,'Jerry', 23);
insert into user (id, username, age) values (3,'Reno', 44);
insert into user (id, username, age) values (4,'Josh', 55);
```

DAO：

```
@Repository
public interface UserRepository extends JpaRepository<User, Long
> {
}
```

Controller：

```
/**
 * 作用：
 * ① 测试服务实例的相关内容
 * ② 为后来的服务做提供
 * @author eacdy
 */
@RestController
public class UserController {
    @Autowired
    private DiscoveryClient discoveryClient;
    @Autowired
    private UserRepository userRepository;

    /**
     * 注：@GetMapping("/{id}")是spring 4.3的新注解等价于：
     * @RequestMapping(value = "/id", method = RequestMethod.GET
)
     * 类似的注解还有@PostMapping等等
     * @param id
     * @return user信息
     */
    @GetMapping("/{id}")
    public User findById(@PathVariable Long id) {
        User findOne = this.userRepository.findOne(id);
        return findOne;
    }

    /**
     * 本地服务实例的信息
     * @return
     */
    @GetMapping("/instance-info")
    public ServiceInstance showInfo() {
        ServiceInstance localServiceInstance = this.discoveryCli
ent.getLocalServiceInstance();
        return localServiceInstance;
    }
}
```

实体类：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column
    private String username;
    @Column
    private Integer age;
    ...
    // getters and setters
}
```

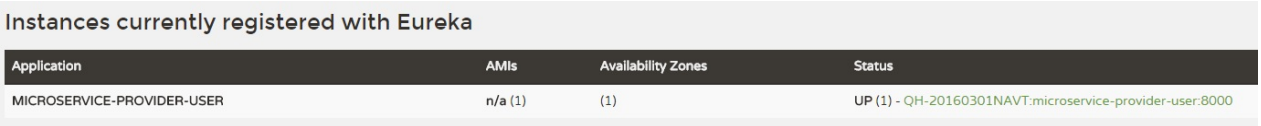
编写Spring Boot启动程序，通过@EnableDiscoveryClient注解，即可将microservice-provider-user服务注册到Eureka上面去

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

至此，代码编写完成。

我们依次启动Eureka服务和microservice-provider-user服务。

访问：http://localhost:8761，如下图。我们会发现microservice-provider-user服务已经被注册到了Eureka上面了。



访问：http://localhost:8000/instance-info，返回结果：

```
{
    "host": "192.168.0.59",
    "port": 8000,
    "metadata": {},
    "uri": "http://192.168.0.59:8000",
    "secure": false,
    "serviceId": "microservice-provider-user"
}
```

访问：http://discovery:8000/1，返回结果：

```
{
    "id": 1,
    "username": "Tom",
    "age": 12
}
```

代码地址（任选其一）：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-provider-user https://github.com/eacdy/spring-cloud-study/tree/master/microservice-provider-user

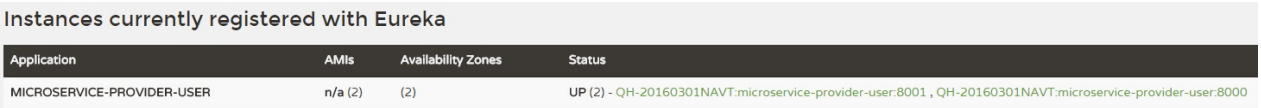
上文我们创建了注册中心，以及服务的提供者microservice-provider-user，并成功地将服务提供者注册到了注册中心上。

要想消费microservice-provider-user的服务是很简单的，我们只需要使用RestTemplate即可，或者例如HttpClient之类的http工具也是可以的。但是在集群环境下，我们必然是每个服务部署多个实例，那么服务消费者消费服务提供者时的负载均衡又要如何做呢？

准备工作

1. 启动注册中心：microservice-discovery-eureka
2. 启动服务提供方：microservice-provider-user
3. 修改microservice-provider-user的端口为8001，另外启动一个实例

此时，访问http://discovery:8761



可以在Eureka中看到microservice-provider-user有两个实例在运行。

下面我们创建一个新的微服务（microservice-consumer-movie-*），负载均衡地消费microservice-provider-user的服务。

第1节：Ribbon

Ribbon介绍

Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层服务连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随即连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。简单地说，Ribbon是一个客户端负载均衡器。

Ribbon代码示例

创建一个Maven项目，并在pom.xml中加入如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-consumer-movie-ribbon</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

        <!-- 整合ribbon -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-ribbon</artifactId>
        </dependency>

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

        </dependency>
    </dependencies>
</project>
```

启动类：MovieRibbonApplication.java。使用@LoadBalanced注解，为RestTemplate开启负载均衡的能力。

```
@SpringBootApplication
@EnableDiscoveryClient
public class MovieRibbonApplication {
    /**
     * 实例化RestTemplate，通过@LoadBalanced注解开启均衡负载能力.
     * @return restTemplate
     */
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

实体类：User.java

```
public class User {
    private Long id;
    private String username;
    private Integer age;
    ...
    // getters and setters
}
```

Ribbon的测试类：RibbonService.java

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

    public User findById(Long id) {
        // http://服务提供者的serviceId/url
        return this.restTemplate.getForObject("http://microservi
ce-provider-user/1", User.class);
    }
}
```

controller：RibbonController.java

```
@RestController
public class RibbonController {
    @Autowired
    private RibbonService ribbonService;

    @GetMapping("/ribbon/{id}")
    public User findById(@PathVariable Long id) {
        return this.ribbonService.findById(id);
    }
}
```

application.yml

```
server:
  port: 8010
spring:
  application:
    name: microservice-consumer-movie-ribbon
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    preferIpAddress: true
```

启动后，访问多次http://localhost:8010/ribbon/1，返回结果：

```
{
    "id": 1,
    "username": "Tom",
    "age": 12
}
```

然后打开两个microservice-provider-user实例的控制台，发现两个实例都输出了类似如下的日志内容：

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_
, user0_.username as username3_0_0_ from user user0_ where user0
_.id=?
2016-09-01 16:37:01.819 TRACE 22164 --- [nio-8001-exec-1] o.h.ty
pe.descriptor.sql.BasicBinder      : binding parameter [1] as [B
IGINT] - [1]
2016-09-01 16:37:01.829 TRACE 22164 --- [nio-8001-exec-1] o.h.ty
pe.descriptor.sql.BasicExtractor   : extracted value ([age2_0_0_
] : [INTEGER]) - [12]
2016-09-01 16:37:01.829 TRACE 22164 --- [nio-8001-exec-1] o.h.ty
pe.descriptor.sql.BasicExtractor   : extracted value ([username3
_0_0_] : [VARCHAR]) - [Tom]
```

至此，我们已经通过Ribbon在客户端侧实现了均衡负载。

代码地址（任选其一）：

Ribbon代码地址：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon

第2节：Feign

Feign介绍

Feign是一个声明式的web service客户端，它使得编写web service客户端更为容易。创建接口，为接口添加注解，即可使用Feign。Feign可以使用Feign注解或者JAX-RS注解，还支持热插拔的编码器和解码器。Spring Cloud为Feign添加了Spring MVC的注解支持，并整合了Ribbon和Eureka来为使用Feign时提供负载均衡。

> 翻译自：http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#spring-cloud-feign

Feign示例

创建一个Maven项目，并在pom.xml添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-consumer-movie-feign</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

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

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

        </dependency>
    </dependencies>
</project>
```

启动类：MovieFeignApplication.java

```
/**
 * 使用@EnableFeignClients开启Feign
 * @author eacdy
 */
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
public class MovieFeignApplication {
    public static void main(String[] args) {
        SpringApplication.run(MovieFeignApplication.class, args)
;
    }
}
```

实体类：User.java

```
public class User {
    private Long id;
    private String username;
    private Integer age;
    ...
    // getters and setters
}
```

Feign测试类：UserFeignClient.java。

```
/**
 * 使用@FeignClient("microservice-provider-user")注解绑定microserv
ice-provider-user服务，还可以使用url参数指定一个URL。
 * @author eacdy
 */
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
    @RequestMapping("/{id}")
    public User findByIdFeign(@RequestParam("id") Long id);
}
```

Feign的测试类：FeignController.java

```
@RestController
public class FeignController {
    @Autowired
    private UserFeignClient userFeignClient;

    @GetMapping("feign/{id}")
    public User findByIdFeign(@PathVariable Long id) {
        User user = this.userFeignClient.findByIdFeign(id);
        return user;
    }
}
```

application.yml

```
server:
  port: 8020
spring:
  application:
    name: microservice-consumer-movie-feign
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    preferIpAddress: true
ribbon:
  eureka:
    enabled: true          # 默认为true。如果设置为false，Ribbon将不会从Eureka中获得服务列表，而是使用静态配置的服务列表。静态服务列表可使用：<client>.ribbon.listOfServers来指定。参考：http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#spring-cloud-ribbon-without-eureka

### 参考：https://spring.io/guides/gs/client-side-load-balancing/
```

同样的，启动该应用，多次访问http://192.168.0.59:8020/feign/1，我们会发现和Ribbon示例一样实现了负载均衡。

代码地址（任选其一）：

Feign代码地址：

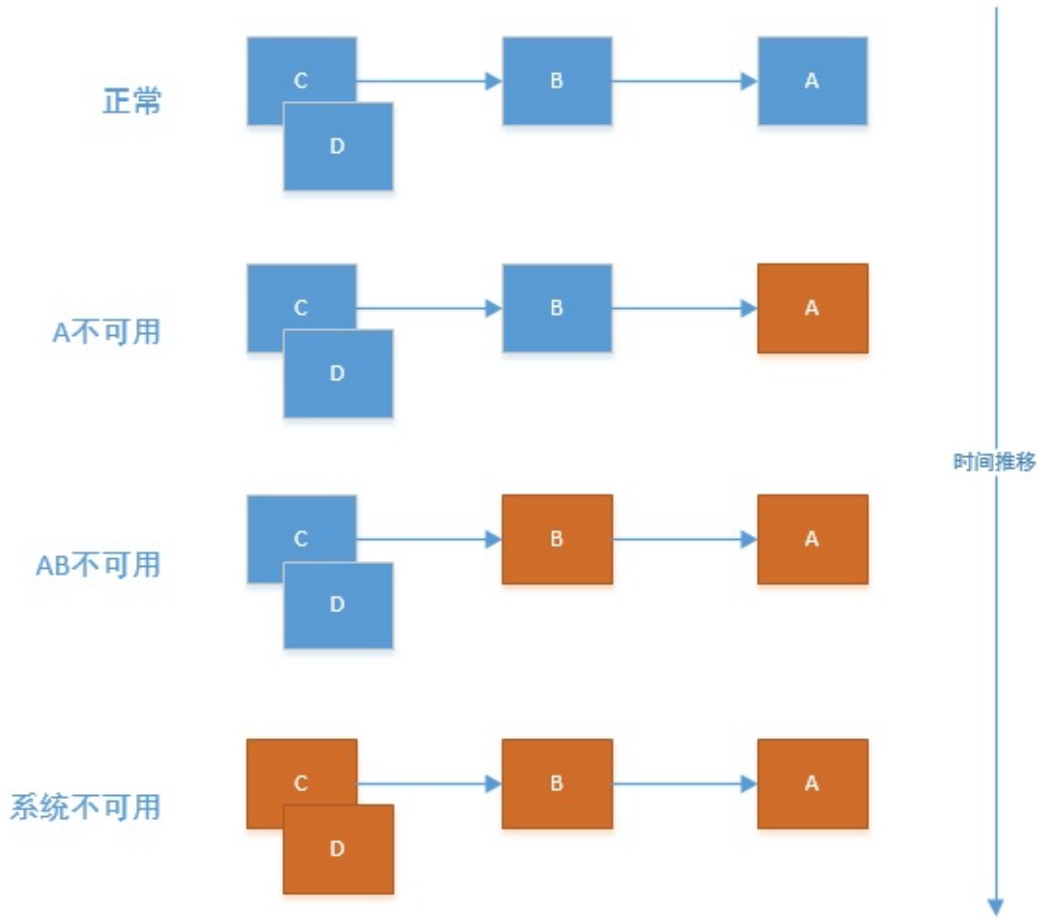
> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign

第5章：熔断器

雪崩效应

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。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。

如果下图所示：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。

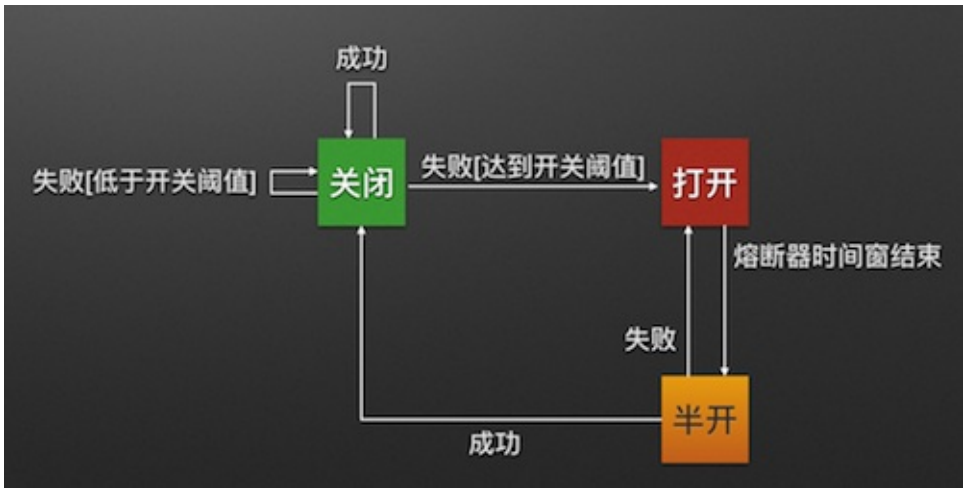


熔断器（CircuitBreaker）

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

熔断器开关相互转换的逻辑如下图：



第1节 ：Hystrix

Hystrix

在Spring Cloud中使用了Netflix开发的Hystrix来实现熔断器。下面我们依然通过几个简单的代码示例，进入Hystrix的学习：

通用方式使用Hystrix

代码示例：

pom.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-consumer-movie-ribbon-with-hystrix</
artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

```
        <!-- 整合ribbon -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-ribbon</artifactId>
        </dependency>

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

        </dependency>

        <!-- 整合hystrix -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-hystrix</artifactId>

        </dependency>
    </dependencies>
</project>
```

启动类：MovieRibbonHystrixApplication.java，使用@EnableCircuitBreaker注解开启断路器功能：

```
/**
 * 使用@EnableCircuitBreaker注解开启断路器功能
 * @author eacdy
 */
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class MovieRibbonHystrixApplication {
    /**
     * 实例化RestTemplate，通过@LoadBalanced注解开启均衡负载能力.
     * @return restTemplate
     */
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(MovieRibbonHystrixApplication.clas
s, args);
    }
}
```

实体类：User.java

```
public class User {
    private Long id;
    private String username;
    private Integer age;
    ...
    // getters and setters
}
```

Hystrix业务类：RibbonHystrixService.java，使用@HystrixCommand注解指定当该方法发生异常时调用的方法

```
@Service
public class RibbonHystrixService {
    @Autowired
    private RestTemplate restTemplate;
    private static final Logger LOGGER = LoggerFactory.getLogger
(RibbonHystrixService.class);

    /**
     * 使用@HystrixCommand注解指定当该方法发生异常时调用的方法
     * @param id id
     * @return 通过id查询到的用户
     */
    @HystrixCommand(fallbackMethod = "fallback")
    public User findById(Long id) {
        return this.restTemplate.getForObject("http://microservi
ce-provider-user/1", User.class);
    }

    /**
     * hystrix fallback方法
     * @param id id
     * @return 默认的用户
     */
    public User fallback(Long id) {
        LOGGER.info("异常发生，进入fallback方法，接收的参数：id = {}"
, id);
        User user = new User();
        user.setId(-1L);
        user.setUsername("default username");
        user.setAge(0);
        return user;
    }
}
```

controller：RibbonHystrixController.java

```
@RestController
public class RibbonHystrixController {
    @Autowired
    private RibbonHystrixService ribbonHystrixService;

    @GetMapping("/ribbon/{id}")
    public User findById(@PathVariable Long id) {
        return this.ribbonHystrixService.findById(id);
    }
}
```

application.yml

```
server:
  port: 8011
spring:
  application:
    name: microservice-consumer-movie-ribbon-with-hystrix
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    hostname: ribbon          # 此处，preferIpAddress不设置或者设为false，不能设为false，否则影响turbine的测试。turbine存在的问题：eureka.instance.hostname一致时只能检测到一个节点，会造成turbine数据不完整
```

验证：

1. 启动注册中心：microservice-discovery-eureka
2. 启动服务提供方：microservice-provider-user
3. 启动服务消费方：microservice-consumer-movie-ribbon-with-hystrix
4. 访问：http://localhost:8011/ribbon/1，获得结果：`{"id":1,"username":"Tom","age":12}`

5. 关闭服务提供方：microservice-provider-user，访问 http://localhost:8011/ribbon/1，获得结果：`{"id":-1,"username":"default username","age":0}`，另外日志打印：`c.i.c.s.u.service.RibbonHystrixService : 异常发生，进入fallback方法，接收的参数：id = 1`

注意：

> 1. 本示例代码在microservice-consumer-movie-ribbon基础上修改而来
> 2. 如对本示例涉及的知识点有疑难，请查看上一章《服务消费者》

代码地址（任选其一）：

> 1. http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon-with-hystrix
> 2. https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon-with-hystrix

Feign使用Hystrix

代码示例

在Feign中使用Hystrix是非常简单的事情，因为Feign已经集成了Hystrix。我们使用microservice-consumer-movie-feign-with-hystrix项目的代码做一点修改，将其中的UserClient.java修改为如下即可：

```
/**
 * 使用@FeignClient注解的fallback属性，指定fallback类
 * @author eacdy
 */
@FeignClient(name = "microservice-provider-user", fallback = Hys
trixClientFallback.class)
public interface UserFeignHystrixClient {
    @RequestMapping("/{id}")
    public User findByIdFeign(@RequestParam("id") Long id);

    /**
     * 这边采取了和Spring Cloud官方文档相同的做法，将fallback类作为内部
类放入Feign的接口中，当然也可以单独写一个fallback类。
     * @author eacdy
     */
    @Component
    static class HystrixClientFallback implements UserFeignHystr
ixClient {
        private static final Logger LOGGER = LoggerFactory.getLo
gger(HystrixClientFallback.class);

        /**
         * hystrix fallback方法
         * @param id id
         * @return 默认的用户
         */
        @Override
        public User findByIdFeign(Long id) {
            LOGGER.info("异常发生，进入fallback方法，接收的参数：id =
 {}", id);
            User user = new User();
            user.setId(-1L);
            user.setUsername("default username");
            user.setAge(0);
            return user;
        }
    }
}
```

这样就完成了，是不是很简单呢？测试过程类似通用方式。

注意：

> 1. 本示例代码在microservice-consumer-movie-feign基础上修改而来
> 2. 如对本示例涉及的知识点有疑难，请查看上一章《服务消费者》

代码地址（任选其一）

> 1. http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix
> 2. https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix

参考文档：

> 1. https://msdn.microsoft.com/en-us/library/dn589784.aspx
> 2. http://martinfowler.com/bliki/CircuitBreaker.html
> 3. http://particular.net/blog/protect-your-software-with-the-circuit-breaker-design-pattern
> 4. https://github.com/Netflix/Hystrix/wiki/How-it-Works#flow-chart
> 5. https://segmentfault.com/a/1190000005988895

第2节：Hystrix Dashboard

Hystrix监控

除了隔离依赖服务的调用以外，Hystrix还提供了近实时的监控，Hystrix会实时、累加地记录所有关于HystrixCommand的执行信息，包括每秒执行多少请求多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。

上文提到的microservice-consumer-movie-ribbon-with-hystrix项目已经具备对Hystrix监控的能力，下面我们进入测试。

测试步骤

1. 启动：microservice-discovery-eureka
2. 启动：microservice-provider-user
3. 启动：microservice-consumer-movie-ribbon-with-hystrix
4. 访问：http://localhost:8011/ribbon/1，注意：该步骤不能省略，因为如果应用的所有接口都未被调用，将只会看到一个ping
5. 访问：http://localhost:8011/hystrix.stream，可以看到类似如下输出：

```
data: {"type":"HystrixCommand","name":"findById","group":"Ri
bbonHystrixService","currentTime":1472658867784,"isCircuitBr
eakerOpen":false,"errorPercentage":0,"errorCount":0,"request
Count":0,"rollingCountBadRequests":0....}
```

并且会不断刷新以获取实时的监控数据。但是纯文字的输出可读性实在是太差，运维人员很难一眼看出系统当前的运行状态。那么是不是有可视化的工具呢？

Hystrix Dashboard

Hystrix Dashboard可以可视化查看实时监控数据。我们可以下载hystrix-dashboard的war包部署到诸如Tomcat之类的容器就，本文不做赘述另外Spring Cloud也提供了Hystrix Dashboard的整合，下面我们看看Spring Cloud是怎么玩转Hystrix Dashboard的。

新建一个maven项目，在其中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-hystrix-dashboard</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

        </dependency>
    </dependencies>
</project>
```

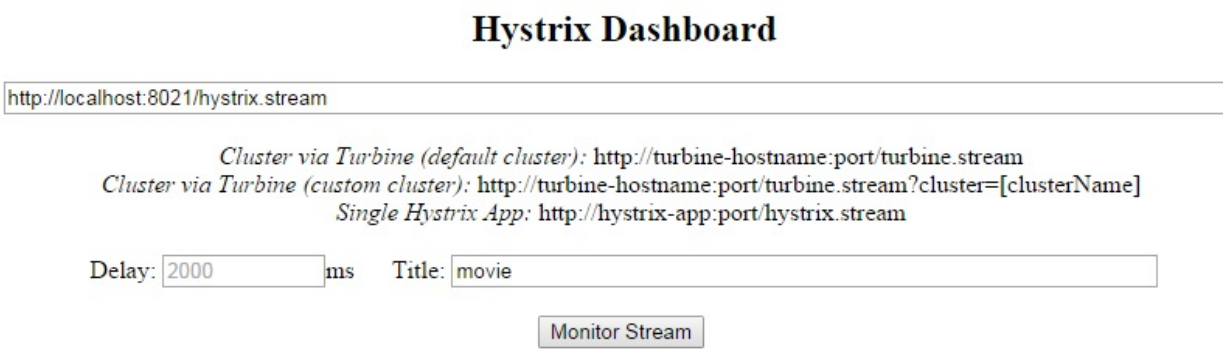
编写启动类：HystrixDashboardApplication.java

```
/**
 * 测试步骤:
 * 1. 访问http://localhost:8030/hystrix.stream 可以查看Dashboard
 * 2. 在上面的输入框填入: http://想监控的服务:端口/hystrix.stream进行测试
 * 注意：首先要先调用一下想监控的服务的API，否则将会显示一个空的图表.
 * @author eacdy
 */
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(HystrixDashboardApplication.class).web(true).run(args);
    }
}
```

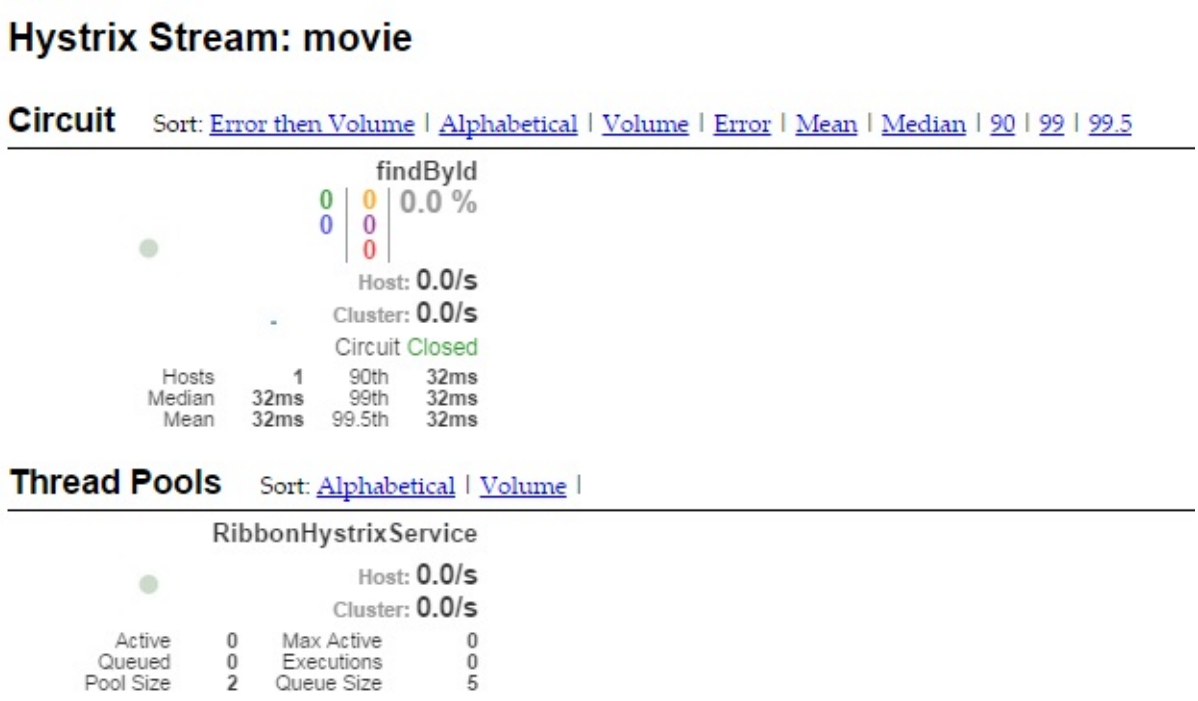
配置文件：application.yml

```
spring:
  application:
    name: hystrix-dashboard
server:
  port: 8030
```

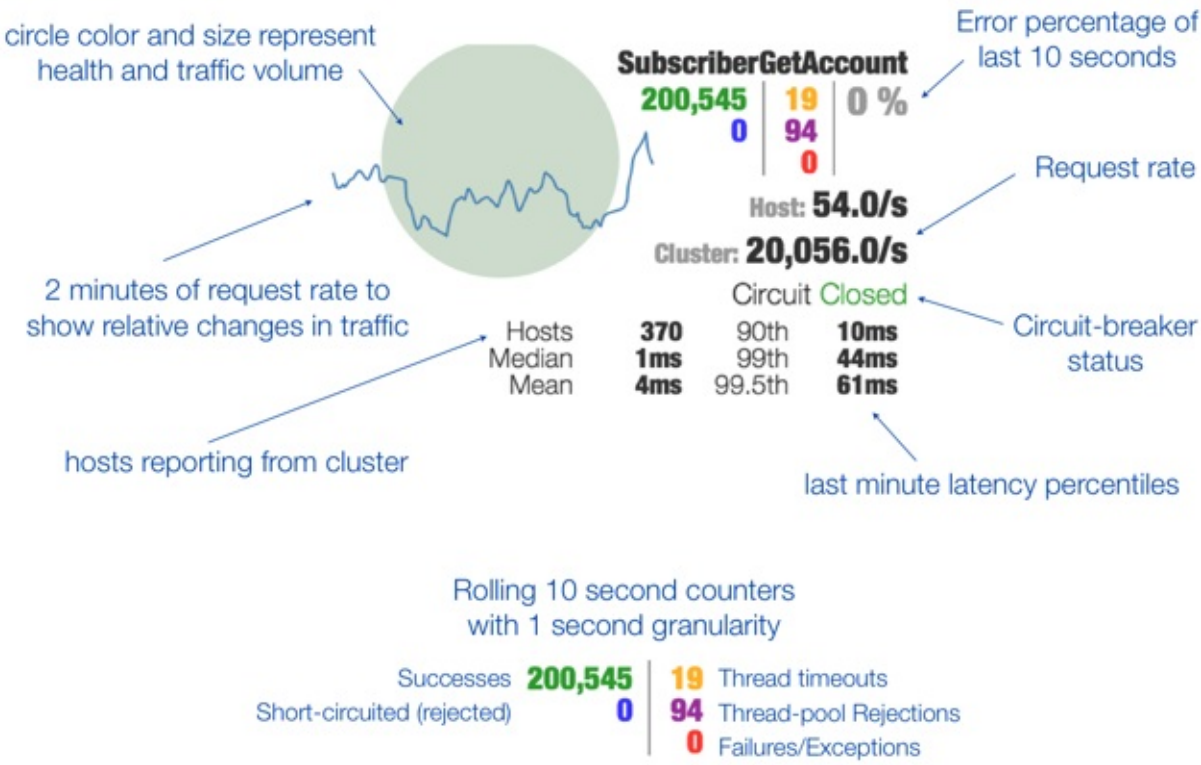
启动后，将会看到如下界面：



此时，我们在输入框中输入http://localhost:8030/hystrix.stream，并随意设置一个Title后，点击Monitor Stream按钮，会出现如下界面：



此时我们会看到findById这个API的各种指标。Hystrix Dashboard Wiki上详细说明了图上每个指标的含义，如下图：



此时，我们可以尝试将microservice-provider-user停止，然后重复访问多次http://localhost:8011/ribbon/1（20次以上），会发现断路器状态会变为开启。

代码地址（任选其一）：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-hystrix-dashboard
>
> https://github.com/eacdy/spring-cloud-study/tree/master/microservice-hystrix-dashboard

TIPS

我们启动前文的microservice-consumer-movie-feign-with-hystrix项目后发现其访问 localhost:8021/hystrix.stream ，是404，查看pom.xml依赖树，发现其没有依赖hystrix-metrics-event-stream项目。故而添加依赖：

```
        <!-- 整合hystrix，其实feign中自带了hystrix，引入该依赖主要是为了使用其中的hystrix-metrics-event-stream，用于dashboard -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-hystrix</artifactId>

        </dependency>
```

并在启动类上添加@EnableCircuitBreaker注解即可。

详见项目microservice-consumer-movie-feign-with-hystrix-stream，因为这不是本文的讨论重点，故而只做扩展阅读。

microservice-consumer-movie-feign-with-hystrix-stream代码地址：

> https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix-stream
>
> https://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix-stream

第3节 ：Turbine

Turbine

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。

和Hystrix Dashboard一样，Turbine也可以下载war包部署到Web容器，本文不做赘述。下面讨论Spring Cloud是怎么使用Turbine的。

编码

新建Maven项目，并在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-hystrix-turbine</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-turbine</artifactId>

        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-netflix-turbine</artifactId>

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

        </dependency>
    </dependencies>
</project>
```

启动类：TurbineApplication.java

```
/**
 * 通过@EnableTurbine接口，激活对Turbine的支持。
 * @author eacdy
 */
@SpringBootApplication
@EnableTurbine
public class TurbineApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(TurbineApplication.class).w
eb(true).run(args);
    }
}
```

配置文件：application.yml

```
spring:
  application.name: microservice-hystrix-turbine
server:
  port: 8031
security.basic.enabled: false
turbine:
  aggregator:
    clusterConfig: default   # 指定聚合哪些集群，多个使用",",分割，默认为default。可使用http://.../turbine.stream?cluster={clusterConfig之一}访问
  appConfig: microservice-consumer-movie-feign-with-hystrix-stream,microservice-consumer-movie-ribbon-with-hystrix  ### 配置Eureka中的serviceId列表，表明监控哪些服务
  clusterNameExpression: new String("default")
  # 1. clusterNameExpression指定集群名称，默认表达式appName；此时：turbine.aggregator.clusterConfig需要配置想要监控的应用名称
  # 2. 当clusterNameExpression: default时，turbine.aggregator.clusterConfig可以不写，因为默认就是default
  # 3. 当clusterNameExpression: metadata['cluster']时，假设想要监控的应用配置了eureka.instance.metadata-map.cluster: ABC，则需要配置，同时turbine.aggregator.clusterConfig: ABC
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/


### 参考：http://blog.csdn.net/liaokailin/article/details/51344281

### 参考：http://blog.csdn.net/zhuchuangang/article/details/51289593
```

这样一个Turbine微服务就编写完成了。

Turbine测试

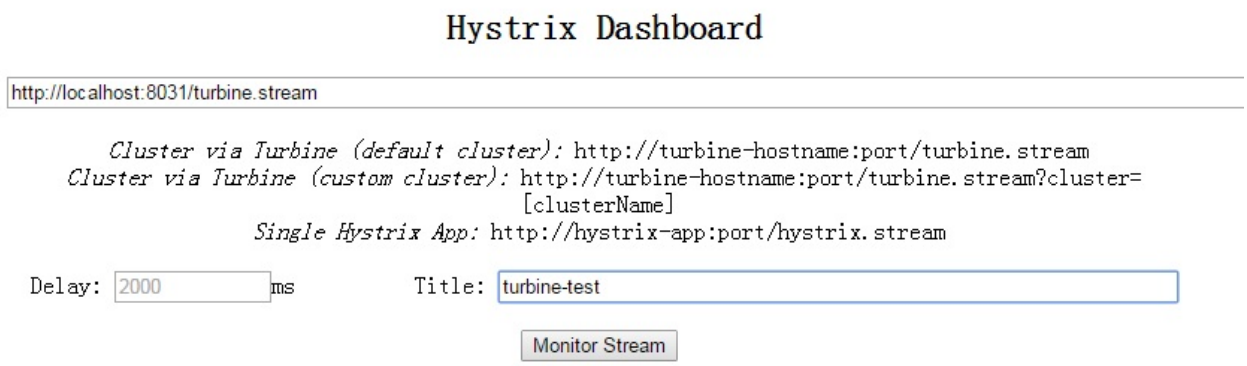
1. 启动项目：microservice-discovery-eureka
2. 启动项目：microservice-provider-user

3. 启动项目：microservice-consumer-movie-ribbon-with-hystrix
4. 启动项目：microservice-consumer-movie-feign-with-hystrix-stream
5. 启动项目：microservice-hystrix-dashboard
6. 启动项目：microservice-hystrix-turbine（即本例）
7. 访问：http://192.168.0.59:8011/ribbon/1，调用ribbon接口
8. 访问：http://localhost:8022/feign/1，调用feign接口
9. 访问：http://localhost:8031/turbine.stream，可查看到和Hystrix监控类似的内容：

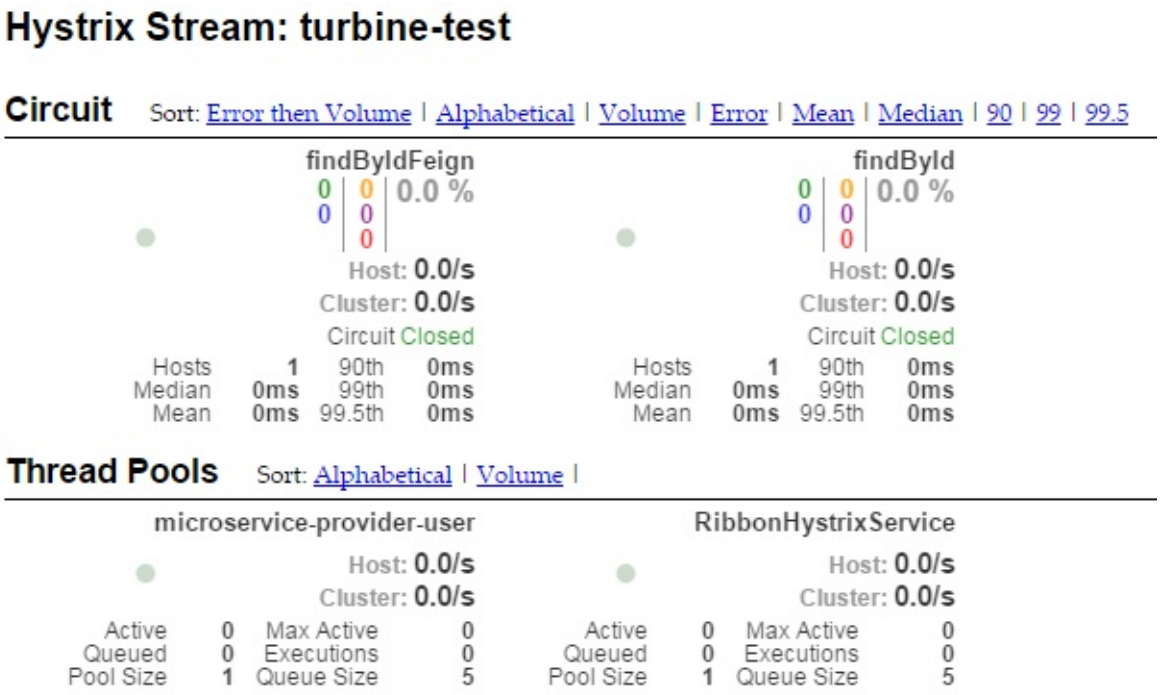
```
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFail
ure":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"
p
```

并且会不断刷新以获取实时的监控数据。同样的，我们可以将这些文本内容放入到Dashboard中展示。

1. 访问Hystrix Dashboard：http://localhost:8030/hystrix.stream，并将http://localhost:8031/turbine.stream输入到其上的输入框，并随意指定一个Title，如下图：



将会查看到如下的图表，有图可见，我们把两个项目的API都监控了：



TIPS

1. 项目microservice-consumer-movie-ribbon-with-hystrix 和 microservice-consumer-movie-feign-with-hystrix-stream 需要配置不同的主机名，并将preferIpAddress设为false或者不设置，否则将会造成在单个主机上测试，Turbine只显示一个图表的情况。项目的代码已经修改好并注释了，这边友谊提示一下。

2. turbine.clusterNameExpression 与 turbine.aggregator.clusterConfig的关系

   | turbine.clusterNameExpression取值 | turbine.aggregator.clusterConfig 取值 | | ---------- | ---------- | | 默认（appName） | 配置想要聚合的项目，此时使用turbine.stream?cluster=项目名称大写访问监控数据 | | new String("default") 或者"'default'" | 不配置，或者配置default，因为默认就是default | | metadata['cluster']；同时待监控的项目配置了类似：eureka.instance.metadata-map.cluster: ABC | 也设成ABC，需要和待监控的项目配置的eureka.instance.metadata-map.cluster一致。 |

具体可以关注 org.springframework.cloud.netflix.turbine.CommonsInstanceDiscovery和 org.springframework.cloud.netflix.turbine.EurekaInstanceDiscovery两个类。特别

关注一下
org.springframework.cloud.netflix.turbine.EurekaInstanceDiscovery.marshall(InstanceInfo)方法。

参考文档

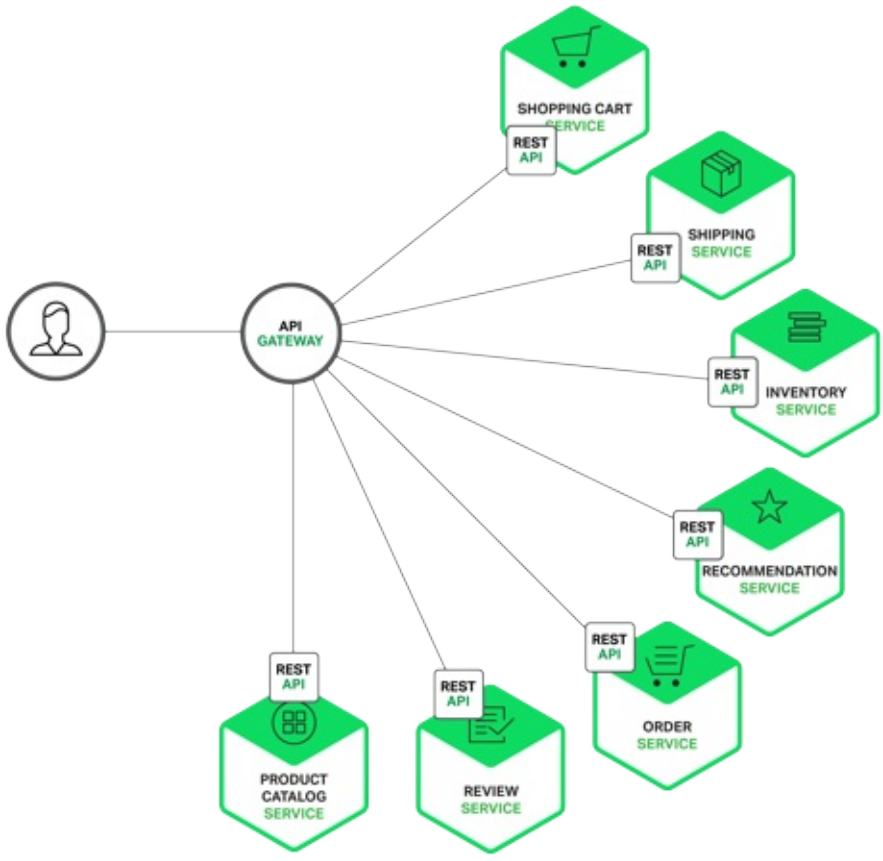
> http://blog.csdn.net/liaokailin/article/details/51344281
>
> http://stackoverflow.com/questions/31468227/whats-for-the-spring-cloud-turbine-clusternameexpression-config-mean

第7章：API Gateway

API Gateway是微服务架构中不可或缺的部分。API Gateway的定义以及存在的意义，Chris已经为大家描述过了，本文不再赘述，以下是链接：

> 中文版：http://dockone.io/article/482
>
> 英文版：https://www.nginx.com/blog/building-microservices-using-an-api-gateway/

使用API Gateway后，客户端和微服务之间的网络图变成下图：



通过API Gateway，可以统一向外部系统提供REST API。Spring Cloud中使用Zuul作为API Gateway。Zuul提供了动态路由、监控、回退、安全等功能。

下面我们进入Zuul的学习：

准备工作

1. 启动服务：microservice-discovery-eureka
2. 启动服务：microservice-provider-user

使用Zuul

创建Maven项目，在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
/maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <artifactId>microservice-api-gateway</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.itmuch.cloud</groupId>
        <artifactId>spring-cloud-microservice-study</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

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

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

启动类：

```
/**
 * 使用@EnableZuulProxy注解激活zuul，同时跟进入该注解可以看到该注解整合了
@EnableCircuitBreaker、@EnableDiscoveryClient，是个组合注解，目的是
简化配置。
 * @author eacdy
 */
@SpringBootApplication
@EnableZuulProxy
public class ZuulApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulApiGatewayApplication.class, a
rgs);
    }
}
```

配置文件：application.yml

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

这样，一个简单的API Gateway就完成了。

测试

启动microservice-api-gateway项目。还记得我们之前访问通过http://localhost:8000/1去访问microservice-provider-user服务中id=1的用户信息吗？

我们现在访问http://localhost:8050/microservice-provider-user/1试试。会惊人地看到：

```
{"id":1,"username":"Tom","age":12}
```

这不正是microservice-provider-user服务中id=1的用户信息吗？

所以我们可以总结出规律：访问

```
http://GATEWAY:GATEWAY_PORT/想要访问的Eureka服务id的小写/**
```

，将会访问到

```
http://想要访问的Eureka服务id的小写:该服务端口/**
```

自定义路径

上文我们已经完成了通过API Gateway去访问微服务的目的，是通过

```
http://GATEWAY:GATEWAY_PORT/想要访问的Eureka服务id的小写/**
```

的形式访问的，那么如果我们想自定义在API Gateway中的路径呢？譬如想使用

```
http://localhost:8050/user/1
```

就能够将请求路由到http://localhost:8000/1呢？

只需要做一点小小的配置即可：

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
# 下面整个树都非必须，如果不配置，将默认使用 http://GATEWAY:GATEWAY_POR
T/想要访问的Eureka服务id的小写/** 路由到：http://想要访问的Eureka服务id
的小写:该服务端口/**
zuul:
  routes:
    user:                                                    # 可以随
便写，在zuul上面唯一即可；当这里的值 = service-id时，service-id可以不写。

      path: /user/**                                         # 想要映
射到的路径
      service-id: microservice-provider-user                 # Eureka
中的serviceId
```

忽略某些服务

准备工作

1. 启动服务：microservice-discovery-eureka
2. 启动服务：microservice-provider-user
3. 启动服务：microservice-consumer-movie-ribbon

如果我们现在只想将microservice-consumer-movie-ribbon服务暴露给外部，microservice-provider-user不想暴露，那么应该怎么办呢？

依然只是一点小小的配置即可：

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
zuul:
  ignored-services: microservice-provider-user          # 需要忽
视的服务(配置后将不会被路由)
  routes:
    movie:                                              # 可以随
便写，在zuul上面唯一即可；当这里的值 = service-id时，service-id可以不写。

      path: /movie/**                                   # 想要映
射到的路径
      service-id: microservice-consumer-movie-ribbon    # Eureka
中的serviceId
```

这样microservice-provider-user服务就不会被路由，microservice-consumer-movie-ribbon服务则会被路由。

测试结果：

URL	结果	
http://localhost:8050/microservice-provider-user/1	404	说明 mic pro 未被
http://localhost:8050/movie/ribbon/1	{"id":1,"username":"Tom","age":12}	说明 mic con mov 被路

不使用Eureka使用Zuul

Zuul并不依赖Eureka，可以脱离Eureka运行，此时需要配置

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
zuul:
  routes:
    movie:                                                    # 可以随
便写
      path: /user/**
      url: http://localhost:8000/                             # path路
由到的地址，也就是访问http://localhost:8050/user/**会路由到http://loc
alhost:8010/**
```

不过笔者并不建议这么做，因为得自己配置URL，不是很方便。

其他使用

Zuul还支持更多的特性、更多的配置项甚至是定制，具体还得各位自行发掘。

其他API Gateway

Zuul只是API Gateway的一种，其他API Gateway有很多，譬如Nginx Plus、Kong等等。

参考文档

https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
http://microservices.io/patterns/apigateway.html