SpringCloud，主讲：汤小洋

一、微服务

1. 微服务简介

将单一应用程序划分成多个微小的服务， 每个服务完成单一功能，这样的每个服务叫做一个微服务

2. 微服务架构

是一种架构模式

- 将应用的每个功能放到一个独立的服务中，每个服务对应一个进程
- 使用一组小型服务来开发单个应用，每个服务运行在独立的进程中，服务与服务之间通过**HTTP的方式**进行互相通信
- 每个服务都是一个可独立替换和独立升级的软件单元， 并且能够被独立的部署到生产环境

优点：

- 分而治之：单个服务功能内聚，复杂性低，方便团队的拆分和管理
- 可伸缩：能够单独的对指定的服务进行伸缩
- 迭代周期短：支持快速的迭代开发
- 独立部署，独立开发

缺点：

- 运维要求高：应用流程通常跨多个微服务，不易进行问题的定位
- 分布式的复杂性：微服务需要使用分布式，而由于分布式本身的复杂性，导致微服务架构也变得复杂起来

二、SpringCloud

1. 简介

SpringCloud是一套完整的微服务解决方案，基于SpringBoot框架

SpringCloud是一系列框架的有序集合，它利用SpringBoot的开发便利性简化了分布式系统的开发

SpringCloud为开发人员提供了快速构建分布式系统的一些工具，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等

2. 技术栈

微服务内容	技术实现
服务的注册与发现	Eureka
服务的接口调用	Feign
服务的熔断器	Hystrix
服务网关	Zuul
负载均衡	Ribbon
服务监控	Zabbix
全链路跟踪	ZipKin
配置管理	Archaius
服务的配置中心	SpringCloud Config
数据流操作	SpringCloud Stream
事件、消息总线	SpringCloud Bus

微服务之间通过HTTP的方式进行互相通信，此时Web API的设计就显得非常重要，会使用Restful API设计方式

三、Restful API

1. 简介

Representational State Transfer，简称为REST， 即表现层状态转化

- 资源 Resources

  指的是网络上的某个数据， 如一个文件、一种服务等
- 表现层 Representational

  资源的表现层，指的是资源的具体呈现形式，如HTML、JSON等
- 状态转化 State Transfer

  指的是状态变化，通过HTTP方法来实现：

  > *GET* *获取资源*
  > *POST* *新建资源*
  > *PUT* *更新资源*
  > *DELETE* *删除资源*

简单来说，客户端通过HTTP方法对服务器的资源进行操作， 实现表现层状态转化

2. 设计原则

Restful 是目前最流行的 API 设计规范，用于 Web 数据接口的设计

Restful API设计原则：

- 尽量将API 部署在一个专用的域名下，如 `http://api.itany.com`
- API的版本应该在URL中体现，如 `http://api.itany.com/v2`
- URL中不要使用动词，应使用资源名词，且使用名词的复数形式，如

功能说明	请求类型	URL
获取用户列表	GET	http://api.itany.com/v2/users
根据id获取用户	GET	http://api.itany.com/v2/users/id
添加用户	POST	http://api.itany.com/v2/users
根据id删除用户	DELETE	http://api.itany.com/v2/users/id
修改用户	PUT	http://api.itany.com/v2/users

注：简单来说，可以使用同一个 URL ，通过约定不同的 HTTP 方法来实施不同的业务

- 服务器响应时返回JSON对象，应包含HTTP状态码、消息、结果等，如 `{code:200,message:"success",result:{id:1001,name:"tom"}}`
- 最好在返回的结果中提供链接， 指向其他的API方法，使得用户不用查文档， 就能知道下一步该怎么做

3. 用法

cloud-provider

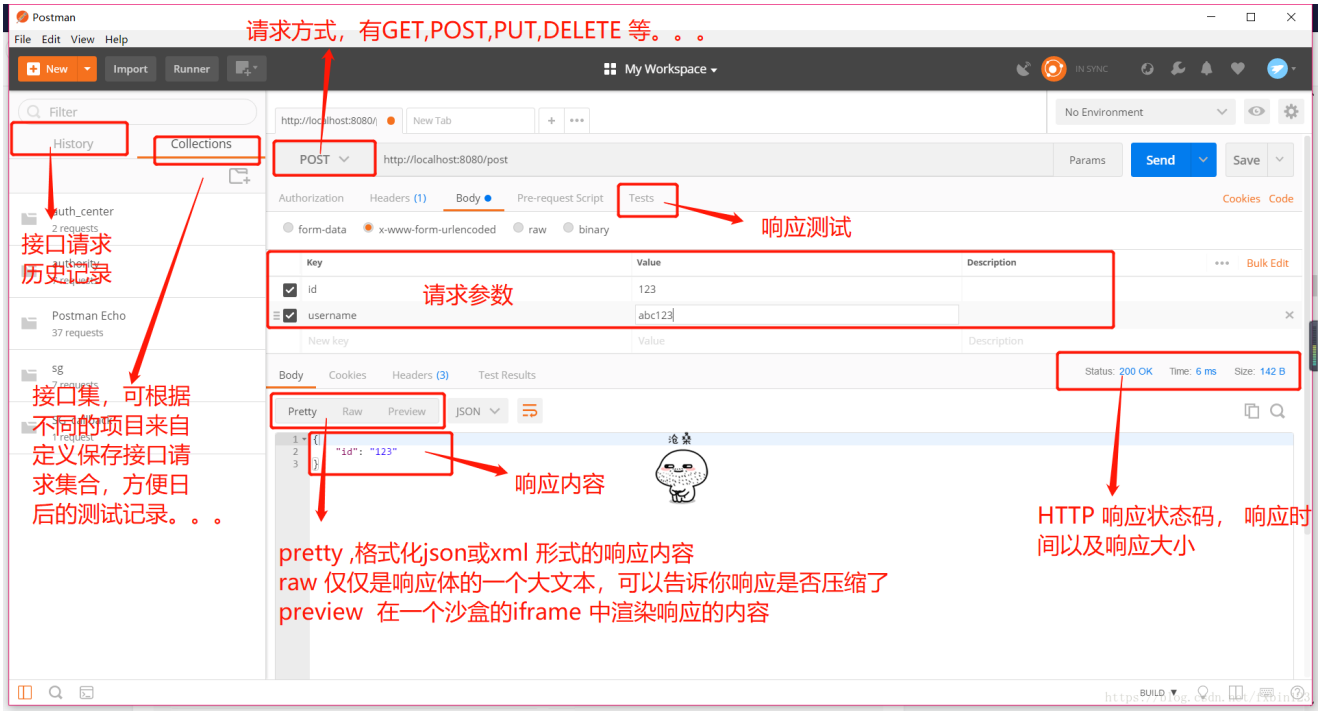
- 构建Restful API服务

cloud-consumer

- 使用RestTemplate调用Rest服务
- RestTemplate是Spring提供的用于访问Rest服务的客户端，提供了访问远程Http服务的方法

4. 使用postman

Postman是一款非常优秀的调试工具，可以用来模拟发送各类HTTP请求，进行接口测试。



5. 使用Swagger2

通常情况下，我们会创建一份Restful API文档来记录所有的接口细节，供其他开发人员使用提供的接口服务，但会存在以下的问题：

- 接口众多，并且细节复杂
- 需要根据接口的变化，不断修改API文档，非常麻烦，费时费力

Swagger2的出现就是为了解决上述的这些问题，减少创建API文档的工作量

- 后端人员在代码里添加接口的说明内容，就能够生成可预览的API文档，无须再维护Word文档
- 让维护文档和修改代码整合为一体，在修改代码逻辑的同时方便的修改文档说明
- 提供了强大的页面测试功能，便于对接口进行测试

使用步骤：

1. 添加Swagger2依赖

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-annotations</artifactId>
    <version>1.5.21</version>
</dependency>
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-models</artifactId>
    <version>1.5.21</version>
</dependency>
```

2. 创建Swagger2配置类

```
@Configuration
@EnableSwagger2  // 启用Swagger2
public class Swagger2Config {

    /**
     * 创建Restful API文档内容
     */
    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo())
                .select()
                // 指定要暴露给Swagger来展示的接口所在的包

.apis(RequestHandlerSelectors.basePackage("com.itany.controller"))
                .paths(PathSelectors.any())
                .build();
    }

    /**
     * 创建API的基本信息，这些信息会展现在文档页面中
     */
    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                // 标题
                .title("使用Swagger2构建Restful API文档")
                // 描述
                .description("欢迎访问后端API接口文档")
                // 联系人
                .contact(new
Contact("tangxiaoyang","https://github.com/tangyang8942","1049901079@qq.com"))
                // 版本号
                .version("1.0")
                .build();
    }

}
```

3. 添加文档内容

使用Swagger2提供的注解对接口进行说明，常用注解：

- @Api 标注在类上，对类进行说明
- @ApiOperation 标注在方法上，对方法进行说明
- @ApiImplicitParams 标注在方法上，对方法的多个参数进行说明
- @ApiImplicitParam 标注在方法上，对方法的一个参数进行说明
- @ApiModel 标注在模型Model上，对模型进行说明
- @ApiModelProperty 标注在属性上，对模型的属性进行说明
- @ApiIgnore 标注在类或方法上，表示忽略这个类或方法

4. 查看Restful API的页面，并测试接口

   启动SpringBoot程序，访问 `http://localhost:端口/swagger-ui.html`

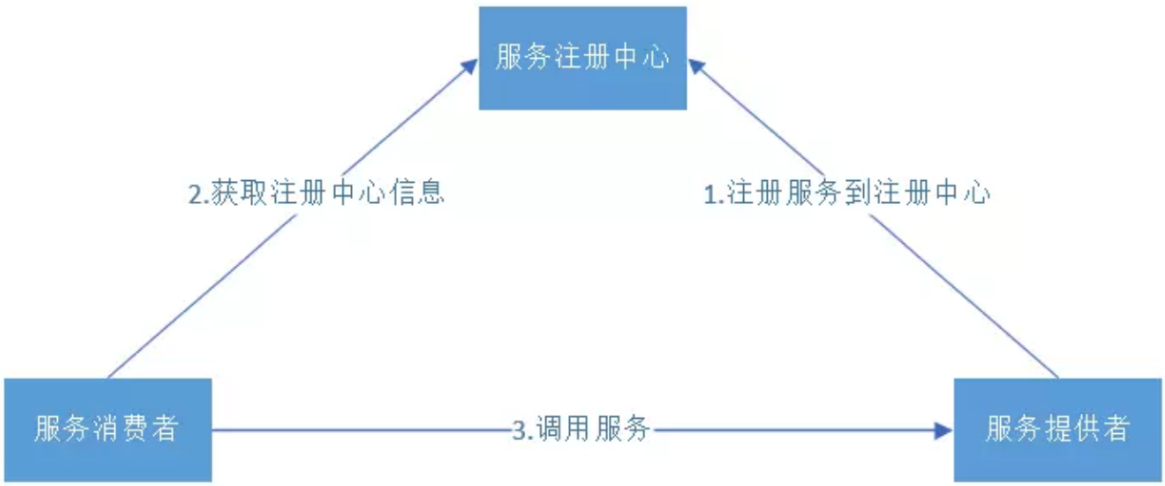
四、Eureka

1. 简介

Eureka是一个基于REST的服务，主要用于服务的注册和发现，以达到负载均衡和中间层服务故障转移的目的。

作用与zookeeper类似，都可以作为服务注册中心

2. 体系结构



执行流程：

1. 服务提供者在启动时，向注册中心注册自己提供的服务
2. 服务消费者在启动时，向注册中心订阅自己所需的服务
3. 注册中心返回服务提供者地址给消费者
4. 服务消费者从提供者地址中调用消费者

两个组件：

- Eureka Server 服务端

  指的是服务注册中心，提供服务的注册和发现

  注册中心会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到

- Euraka Client 客户端

  指的是服务提供者和消费者，在应用程序启动时会和服务端进行交互，注册或订阅服务

3. 搭建服务注册中心

步骤：

1. 创建项目，勾选Eureka Server
2. 编辑pom.xml文件，配置依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <!-- 修改springboot版本为2.0.6.RELEASE -->
  <version>2.0.6.RELEASE</version>
  <relativePath/>
</parent>

<properties>
  <!-- 修改springcloud版本为Finchley.SR1 -->
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>
```

3. 编辑application.yml文件，配置eureka

```
server:
  port: 7001

eureka:
  client:
    # 是否将自己注册到Eureka-Server中，默认的为true
    register-with-eureka: false
    # 是否从Eureka-Server中获取服务提供者的注册信息，默认为true
    fetch-registry: false
    # 设置服务注册中心的地址
    service-url:
      defaultZone: http://localhost:${server.port}/eureka/
```

Spring Cloud Eureka 常用配置及说明

4. 编辑启动类，启用Eureka服务器

```
@SpringBootApplication
@EnableEurekaServer // 启用Eureka服务器
public class CloudEureka7001Application {
    public static void main(String[] args) {
        SpringApplication.run(CloudEureka7001Application.class, args);
    }
}
```

5. 访问Eureka-Server服务管理平台

   通过浏览器访问 `localhost:7001`

4. 注册服务

步骤：

1. 编辑pom.xml文件，配置Eureka-Client依赖

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

注：如果是新建项目，可以勾选Eureka Discovery Client

2. 编辑application.yml文件，配置eureka

```
server:
  port: 8001

spring:
  application:
    # 应用名，在注册中心显示的服务名
    name: user-provider

eureka:
  client:
    # 指定服务注册中心的地址
    service-url:
      defaultZone: http://localhost:7001/eureka/
```

3. 编辑启动类，启用Eureka客户端

```
@SpringBootApplication
@EnableEurekaClient // 启用Eureka客户端
public class CloudProvider8001Application {
    public static void main(String[] args) {
        SpringApplication.run(CloudProvider8001Application.class, args);
    }
}
```

5. 自我保护机制

在某个时刻，如果某个服务不可用了，Eureka不会立即的清理该服务， 依旧会对该服务的信息进行保存

> *默认情况下，微服务在Eureka上注册后，会每30秒发送心跳包，Eureka通过心跳来判断服务是否健康，如果Eureka的Server在一定时间内（默认90s），没有接收到某个微服务实例的心跳，将会注销该实例。*
>
> *但是当网络发生故障时，通常会导致Eureka Server在短时间内无法收到大批微服务的心跳，但微服务自身是正常的，只是网络通信出现了故障。*

> *考虑到这种情况，Eureka设置了一个阀值，当心跳失败的比例在15分钟之内低于85%时，Eureka Server认为很大程度上出现了网络故障，将不再删除心跳过期的微服务，尽可能的保护这些注册信息，自动进入 自我保护模式。*
>
> *当网络故障被解决时，服务将自动退出 自我保护模式*

可以关闭自我保护机制 `eureka.server.enable-self-preservation=false`

五、Feign

1. 简介

Feign是一个HTTP客户端，可以更快捷、优雅地调用HTTP服务，使编写HTTPClient变得更简单。

在Spring Cloud中使用Feign非常简单，只需要创建一个接口，然后在接口上添加一些注解就可以了。

2. 用法

步骤：

1. 编辑pom.xml文件，配置Feign和Eureka-Client依赖

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

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

2. 编辑application.yml文件，配置eureka

```
eureka:
  client:
    # 是否将自己注册到Eureka-Server中，默认的为true
    register-with-eureka: false
    # 指定服务注册中心的地址
    service-url:
      defaultZone: http://localhost:7001/eureka/
```

3. 编辑启动类，启用Eureka客户端

```
@SpringBootApplication
@EnableEurekaClient
// 启用Feign客户端，扫描指定包下所有的feign注解
@EnableFeignClients(basePackages = "com.itany.service")
public class CloudConsumer8080Application {
    public static void main(String[] args) {
        SpringApplication.run(CloudConsumer8080Application.class, args);
    }
}
```

4. 创建接口并配置

```
// 调用的服务名，到Eureka中寻找对应的微服务
@FeignClient("user-provider")
public interface UserService {

    @GetMapping("/users")
    public ResponseResult getUserList();

    @GetMapping("/users/{id}")
    public ResponseResult getUser(@PathVariable(value = "id") Integer id);

    @PostMapping("/users")
    public ResponseResult postUser(@RequestParam("username") String
username,@RequestParam("password") String password);

    @DeleteMapping("/users/{id}")
    public ResponseResult deleteUser(@PathVariable(value = "id") Integer id);

    @PutMapping("/users")
    public ResponseResult putUser(@RequestParam Map<String,Object> map);

}
```

3. 传参

feign传递对象参数的解决方式：

- 方式一：将对象参数拆为多个简单类型参数，且必须添加@RequestParam注解
- 方式二：使用Map替代对象参数，且必须添加@RequestParam注解

六、Hystrix

1. 服务熔断和服务降级

服务雪崩：在微服务架构中服务之间会相互调用和依赖，如果某个服务发生故障，可能会导致多个服务故障，从而导致整个系统故障

解决服务雪崩的方式：

- 服务熔断

  当服务出现不可用或响应超时时，为了防止整个系统出现雪崩， 暂时停止对该服务的调用，直接返回一个结果，快速释放资源。

  如果检测到目标服务情况好转，则恢复对目标服务的调用。

- 服务降级

  为了防止核心业务功能出现负荷过载或者响应慢的问题，将非核心服务进行降级，暂时性的关闭或延迟使用，保证核心服务的正常运行

2. 简介

Hystrix就是用来实现服务熔断的，其实就是一种机制，当某个服务不可用时，可以阻断故障的传播，称为断路器或熔断器

- Hystrix负责监控服务之间的调用情况，当出现连续多次调用失败的情况时会进行熔断保护
- 该服务的断路器就会打开，直接返回一个由开发者设置的fallback（退路）信息
- Hystrix会定期再次检查故障的服务，如果故障服务恢复，将继续使用服务

3. 用法

断路器是安装在服务消费者上的，我们需要做的是在服务消费者上开启断路器并配置。

在Feign中使用Hystrix是非常简单的，已经包含了整合Hystrix的依赖

步骤：

1. 编辑application.yml文件，启用断路器

```
# 启用断路器
feign:
  hystrix:
    enabled: true
```

2. 设置fallback信息

   在@FeignClient注解中指定fallback参数

```
// 调用的服务名，到Eureka中寻找对应的微服务，找到的是：微服务的ip:port
@FeignClient(value = "user-provider",fallback = UserServiceFallback.class)
public interface UserService {
```

   创建UserService接口的实现类，并配置返回的信息

```
@Service
public class UserServiceFallback implements UserService {

    @Override
    public ResponseResult getUserList() {
        System.out.println("断路器开启。。。。。。
UserServiceFallback.getUserList");
        return ResponseResult.fail("获取用户列表失败");
    }

    @Override
    public ResponseResult getUser(Integer id) {
        return ResponseResult.fail("获取指定用户失败");
    }

    @Override
    public ResponseResult postUser(String username, String password) {
        return ResponseResult.fail("添加用户失败");
    }

}
```

3. 测试

   当服务提供者不可用或出现异常时，会暂时停止对该服务的调用

七、Zuul

1. 简介

Zuul是一个路由网关Gateway，包含两大功能：

- 对请求的路由

  将外部的请求转发到具体的微服务实例上，是实现外部访问统一入口的基础

- 对请求的过滤

  对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础

将Zuul和Eureka进行整合，把Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的信息，对于微服务的访问都要通过Zuul进行跳转

2. 用法

Zuul本身也是一个项目，一个最终也会注册到Eureka中的微服务

步骤：

1. 创建项目，勾选Zuul和Eureka Discovery Client

2. 编辑pom.xml文件，配置依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

3. 编辑application.yml文件，配置eureka和zuul

```
server:
  port: 6001

spring:
  application:
    name: zuul-gateway

eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/

# 路由相关配置
zuul:
  # 请求前缀
  prefix: /v2
  # 配置路由表
  routes:
    # 对于每个微服务，可以指定一个唯一的key值，该值可以任意指定
    user:
      # 将 /user-service/ 开头的请求映射到 user-provider 这个微服务上
      path: /user-service/**
      serviceId: user-provider
```

4. 编辑启动类，启用Zuul

```
@SpringBootApplication
@EnableZuulProxy // 启用Zuul
public class CloudZuul6001Application {
    public static void main(String[] args) {
        SpringApplication.run(CloudZuul6001Application.class, args);
    }
}
```

5. 修改cloud-consumer的UserService，通过Zuul访问微服务，相当于是代理

```
// 调用的服务名，到Eureka中寻找对应的微服务，找到的是：微服务的ip:port
// @FeignClient(value = "user-provider",fallback = UserServiceFallback.class)
// 所有微服务的访问都要通过Zuul进行路由跳转
@FeignClient(value = "zuul-gateway",fallback = UserServiceFallback.class)
// 避免idea提示找不到该组件
@Service
public interface UserService {

    // @GetMapping("/users")
    @GetMapping("/v2/user-service/users")
    public ResponseResult getUserList();
```

6. 测试

八、Ribbon

1. 简介

Ribbon是一套客户端负载均衡的工具，用于实现微服务的负载均衡 Load Balance

Ribbon不需要独立部署，Feign集成了Ribbon，自动的实现了负载均衡

2. 用法

步骤：

1. 搭建Provider集群

   拷贝 `cloud-provider-8001` 为 `cloud-provider-8002` 和 `cloud-provider-8003`

   - 修改pom.xml中名称
   - 修改主启动类的类名
   - 修改application.yml配置：端口号和instance-id不同相同

```yaml
server:
  port: 8002  # 不同的端口

spring:
  application:
    name: user-provider  # 相同的服务名

eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/
  instance:
    instance-id: user-provider8002  # 不同的实例id
```

2. 测试负载均衡

   默认使用的是 `轮询` 的策略

   常见的策略：

   - 轮询 (RoundRobinRule) （默认）
   - 随机 (RandomRule)
   - 响应时间权重（WeightedResponseTimeRule）响应时间越短的服务器被选中的可能性大
   - 并发量最小可用（BestAvailableRule）选取最少并发量请求的服务器

3. 改变负载均衡策略

   在Zuul服务中，通过配置类指定要应用的负载均衡策略

```java
@Configuration
public class RibbonConfig {
    @Bean
    public IRule ribbonRule(){
        return new RandomRule();
    }
}
```

九、面试题

1. 什么是微服务
2. 什么是微服务架构
3. 微服务之间是如何通信的
4. SpringCloud 和 Dubbo 之间有什么区别
5. Eureka Server 和 ZooKeeper 之间有什么区别
6. StringCloud 和 SpringBoot， 谈一谈你的理解
7. 什么是服务熔断？什么是服务降级
8. 微服务的优缺点是什么？你在实际开发中遇到过什么样的问题
9. 简述微服务的技术栈
10. Restful API 是什么？ 有哪些要求