# 微服务架构 SpringCloud

**内参保密不得外传**

**达内Java培优方向**

**陈子枢 主讲**

**QQ: 1920-9520-78**

**http://dobetter.tedu.cn**

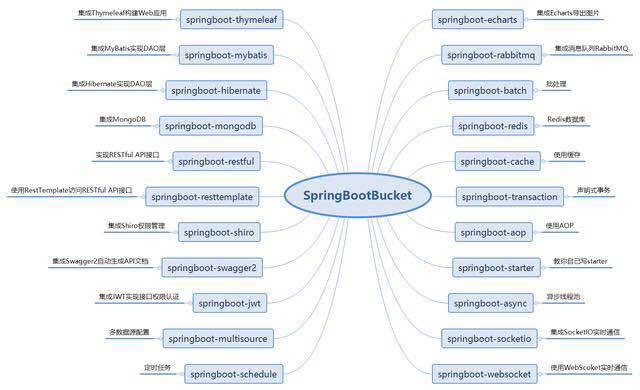
**2018**

# 第一天：SpringBoot全新开发方式+部署+Mybatis注解编程

## 介绍

### 定义

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。Spring Boot 现在已经成为Java 开发领域的一颗璀璨明 珠，它本身是包容万象的，可以跟各种技术集成。成为SpringBoot全家桶。



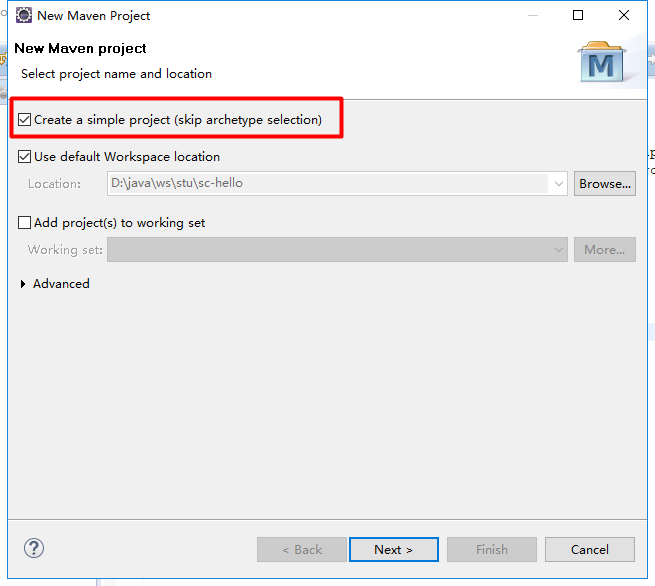
### 特点

* 创建独立的Spring应用程序
* 嵌入的Tomcat，无需部署WAR文件
* 简化Maven配置
* 自动配置Spring
* 提供生产就绪型功能，如指标，健康检查和外部配置

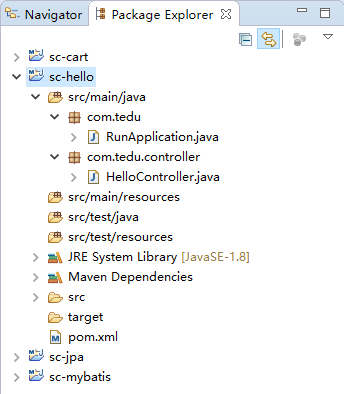
## 第一个SpringBoot例子

### 创建Maven工程

创建Maven工程，创建simple project，类型为jar



修改jdk版本必须1.8以上，springboot基本上废除了1.6、1.7



### 全新构建方式pom.xml

传统构建Maven项目，pom中的依赖包繁多，升级一个jar版本时，会引发新的冲突，调试许久。而SpringBoot接管了jar的版本，它构建好一套，这套中各jar的版本已经测试好，开发者再无需去关注每个依赖包的版本及冲突问题，从而简化开发。

再者，它启动也非常快，直接运行一个类，使用tomcat的maven插件。开发调试时效率提高。

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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.tedu</groupId>

<artifactId>sc-hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

</project>

### 创建HelloController

使用@RestController替代@Controller和@ResponseBody（返回json串）

package com.tedu.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@RequestMapping("hello")

public String hello(){

return "hello springboot";

}

}

### 全新运行方式

注意：必须加@SpringBootApplication注解，同时必须有包名

package com.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class RunApplication {

public static void main(String[] args) {

SpringApplication.run(RunApplication.class, args);

}

}

## 直接运行jar启动web服务

### pom.xml

在pom.xml中增加plugin插件，忘记加会报“清单找不到”异常。

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<groupId>com.tedu</groupId>

<artifactId>hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

### 执行jar包

进入到jar所在路径，然后执行java直接运行jar文件，启动web服务

D:\java\ws\stu\hello\target>D:\java\env\jdk1.8\bin\java -jar hello-0.0.1-SNAPSHOT.jar



## 打war包手动部署

### pom.xml

修改包类型为war，增加依赖spring-boot-starter-tomcat，<scope>的provided表示使用外部的tomcat，而不使用集成微缩版的tomcat。

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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.tedu</groupId>

<artifactId>hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<!-- 怎么加入父工程 -->

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 定义版本号变量，SpringBoot要求jdk1.8以上 -->

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<!-- 标识它的类型和jar依赖 -->

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

</dependencies>

<!-- jar运行方式 -->

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

### WebApplication

继承SpringBootServletInitializer覆盖基类的configure方法

package com.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.builder.SpringApplicationBuilder;

import org.springframework.boot.web.support.SpringBootServletInitializer;

public class WebApplication extends SpringBootServletInitializer{

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {

return builder.sources(WebApplication.class);

}

public static void main(String[] args) {

SpringApplication.run(WebApplication.class, args);

}

}

### 打包war

不能RunApplication，只能执行maven命令打包，手动把war考入到tomcat下。

D:\java\ws\stu\hello\target\hello-0.0.1-SNAPSHOT.war

## SpringBoot和mybatis整合

### 系统架构图



### 创建数据库表

创建mybatisdb数据库，设置utf8字符集。

创建user表：

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (

`ID` int(11) NOT NULL,

`NAME` varchar(30) DEFAULT NULL,

`BIRTHDAY` datetime DEFAULT NULL,

`ADDRESS` varchar(200) DEFAULT NULL,

PRIMARY KEY (`ID`)

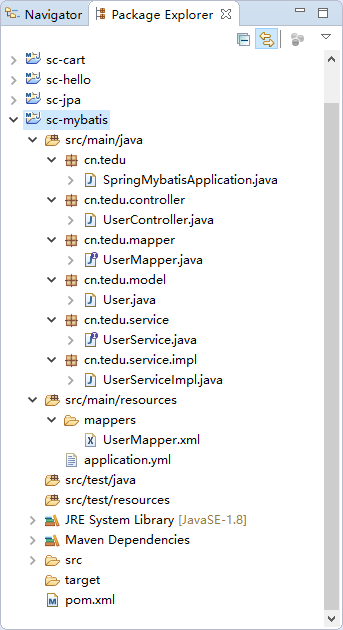
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

增加数据：

insert into `user`(`ID`,`NAME`,`BIRTHDAY`,`ADDRESS`) values (1,'夏言','1573-01-01 00:00:00','桂州村'),(2,'严嵩','1587-01-01 00:00:00','分宜县城介桥村'),(3,'徐阶','1580-01-01 00:00:00','明松江府华亭县'),(4,'高拱','1566-01-01 00:00:00','河南省新郑市高老庄村'),(5,'张居正','1558-01-01 00:00:00','江陵');

### 创建Maven工程

创建Maven工程，创建simple project，类型为jar



### pom.xml

额外需要的jar，还得自己依赖，例如：mysql驱动包，阿里的数据源druid包

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>sc-mybatis</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>sc-mybatis</name>

<url>http://maven.apache.org</url>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.3.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid-spring-boot-starter</artifactId>

<version>1.1.0</version>

</dependency>

</dependencies>

</project>

### User.java

创建pojo对象

package cn.tedu.model;

import java.util.Date;

public class User {

private Integer id;

private String name;

private Date birthday;

private String address;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Date getBirthday() {

return birthday;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

### UserMapper.xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace命名空间，唯一特性 -->

<mapper namespace="cn.tedu.mapper.UserMapper">

<select id="find" resultType="User">

select id,name,birthday,address from user

</select>

</mapper>

### UserMapper.java 接口

package cn.tedu.mapper;

import java.util.List;

import org.apache.ibatis.annotations.Param;

import org.apache.ibatis.annotations.Select;

import cn.tedu.model.User;

//注解和xml方式混合

public interface UserMapper {

//调用xml方式

public List<User> find();

//调用注解方式

@Select("select \* from user where id=#{id}")

public User get(@Param("id") Integer id);

}

### UserService.java

package cn.tedu.service;

import java.util.List;

import cn.tedu.model.User;

public interface UserService {

public List<User> find();

public User get(Integer id);

}

### UserServiceImpl.java

package cn.tedu.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import cn.tedu.mapper.UserMapper;

import cn.tedu.model.User;

import cn.tedu.service.UserService;

@Service

public class UserServiceImpl implements UserService{

@Autowired

private UserMapper userMapper;

public List<User> find() {

return userMapper.find();

}

public User get(Integer id){

return userMapper.get(id);

}

}

### UserController.java

使用@RestController替代@Controller和@ResponseBody（返回json串）

package cn.tedu.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import cn.tedu.model.User;

import cn.tedu.service.UserService;

@RestController

@RequestMapping(value = "/user")

public class UserController {

@Autowired

private UserService userService;

@RequestMapping("/find")

public List<User> find() {

return userService.find();

}

@RequestMapping("/get/{id}")

public User get(@PathVariable Integer id){

return userService.get(id);

}

}

### application.yml

全局配置文件，yml为新的配置文件方式，注意其中格式为空格，不能有tab。

配置端口，配置数据源，配置mybatis全局配置。

注意：如果端口，启动时日志显示8080，说明此文件未加载。检查原因一般是文件名或者路径不正确。

server:

port: 8070

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/mybatisdb

username: root

password: root

mybatis:

typeAliasesPackage: com.tedu.pojo

mapperLocations: classpath:mappers/\*.xml

logging:

level:

com.tedu.mapper: debug



### RunApplication.java

package cn.tedu;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

@MapperScan(basePackages="cn.tedu.mapper") //扫描Mybatis接口文件

public class RunApplication {

public static void main(String[] args) {

SpringApplication.run(RunApplication.class, args);

}

}

# 第二天：SpringBoot+MybatisPlus重构购物车

## MyBatisPlus

### 介绍



使用SpringBoot+Mybatis+MybatisPlus(MP)全新注解方式，自动产生SQL语句，替代旧的通用Mapper。旧的通用Mapper是基于Mybatis拦截器的机制，而新的MybatisPlus是基于注解扫描机制，在启动服务时就进行加载，所以性能比旧的通用Mapper方式高很多。

MyBatisPlus官网：http://mp.baomidou.com



### 实现方式

|  |  |  |
| --- | --- | --- |
| 方案 | 描述 | 备注 |
| MyBatis | CartMapper接口  CartMapper.xml映射文件 | 手写SQL语句 |
| 通用Mapper | CartMapper接口  继承SysMapper接口 | 自动生成SQL语句 |
| MP（MyBatisPlus） | CartMapper接口  继承BaseMapper接口 | 自动生成SQL语句 |

### BaseMapper底层接口

已经封装好了CRUD，单表、批量、分页等19个方法，极大减少开发者的工作量。



### MybatisPlus注解

Pojo对象上标识，自动驼峰规则

@TableName("tb\_cart") //类和表的映射

public class Cart extends BasePojo{

private static final long serialVersionUID = 1L;

//@TableField(exist=false) //数据库表中不存在的字段要标识

@TableId(value="id",type=IdType.AUTO) //主键自增

private Long id;

@TableField("user\_id") //属性和字段的映射

private Long userId;

* @TableName("tb\_cart") //类和表的映射
* @TableId(value="id",type=IdType.AUTO) //主键自增
* @TableField("user\_id") //属性和字段的映射
* @TableField(exist=false) //表中不存在的字段要标识

### 自动填充公共字段

@TableField(value = "created",fill = FieldFill.INSERT)

private Date created;

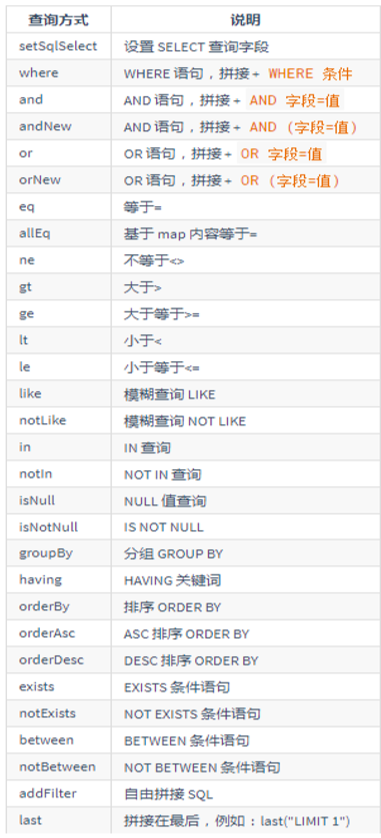
@TableField(value = "updated",fill = FieldFill.INSERT\_UPDATE)

private Date updated;

### 条件构造器

Mybatis提供QBC( Query By Criteria)，实现面向对象的查询方式。

实体包装器 EntityWrapper，用于处理 sql 拼接，排序，实体参数查询等。注意其使用的是数据库字段而不是java属性。



例如：翻页查询

public Page<T> selectPage(Page<T> page, EntityWrapper<T> entityWrapper) {

if (null != entityWrapper) {

entityWrapper.orderBy(page.getOrderByField(), page.isAsc());

}

page.setRecords(baseMapper.selectPage(page, entityWrapper));

return page;

}

**拼接SQL方式 一:**

EntityWrapper<User> ew = new EntityWrapper<User>();

ew.setEntity(new User(1));

ew.where("user\_name={0}", "'tonychen'").and("id=1")

.orNew("user\_status={0}", "0").or("status=1")

.notLike("user\_nickname", "tony")

.andNew("new=xx").like("hhh", "ddd")

.andNew("pwd=11").isNotNull("n1,n2").isNull("n3")

.groupBy("x1").groupBy("x2,x3")

.having("x1=11").having("x3=433")

.orderBy("name").orderBy("birthday,address");

System.out.println(ew.getSqlSegment());

**拼接 SQL方式 二:**

int buyCount = selectCount(Condition.create()

.setSqlSelect("sum(quantity)")

.isNull("order\_id")

.eq("user\_id", 1)

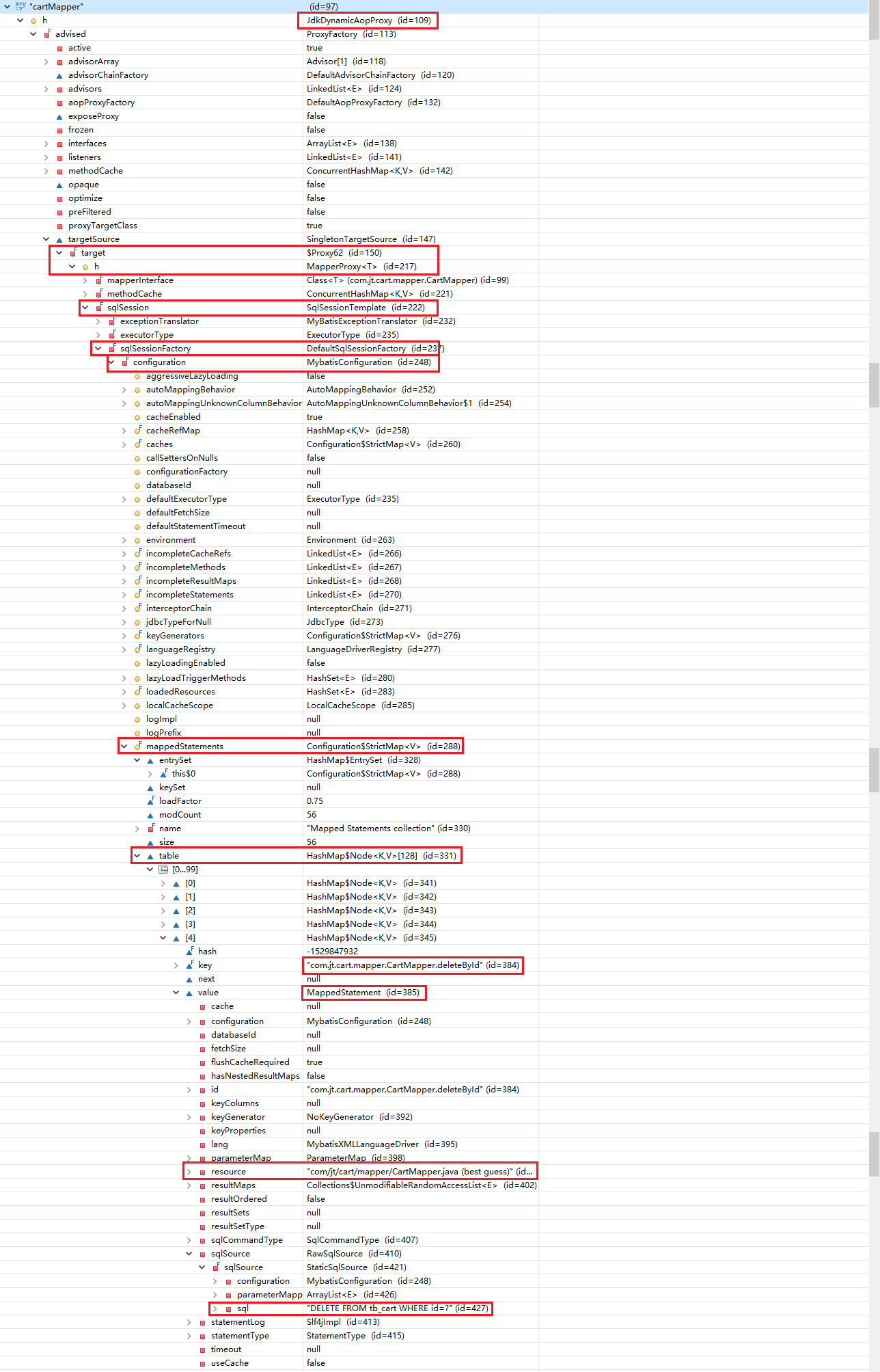
.eq("type", 1)

.in("status", new Integer[]{0, 1})

.eq("product\_id", 1)

.between("created\_time", startDate, currentDate);

### MP工作原理



## 购物车接口

### 我的购物车

根据用户ID查询购物车数据

|  |  |
| --- | --- |
| 请求方法 | GET |
| URL | http://cart.jt.com/cart/query/{userId} |
| 参数 | userId 用户ID |
| 示例 | http://cart.jt.com/cart/query/7 |
| 返回值 | {  status: 200 //200 成功，201 没有查到  msg: “OK” //返回信息消息  data:  [  1]  0:  {  created: 1418092628000  updated: 1418092628000  id:1  userId:1  itemId:39  itemTitle: “java核心技术”  itemImage: http://image.jt.com/images/2015/06/11/20150309118.jpg  itemPrice: 87200  num:1  }  } |

### 加入购物车

|  |  |
| --- | --- |
| 请求方法 | POST |
| URL | http://cart.jt.com/cart/save |
| 参数 | userId 用户ID  itemId 商品ID  itemTitle 商品标题  itemImage 商品主图  itemPrice 商品价格  num 商品数量 |
| 示例 |  |
| 返回值 | {  status: 202 //200成功，201失败，202已经存在，400参数不合法  msg: “该商品已经存在购物车中！” //对应的消息  data: null  } |

### 更新商品数量

|  |  |
| --- | --- |
| 请求方法 | GET |
| URL | http://cart.jt.com/cart/update/num/{userId}/{itemId}/{num} |
| 参数 | userId 用户名ID  itemId 商品ID  num 数量 |
| 示例 | http://cart.jt.com/cart/update/num/001/002/2 |
| 返回值 | {  status: 200  msg: “OK”  data:null  } |
|  |  |

### 删除购物车中商品

|  |  |
| --- | --- |
| 请求方法 | GET |
| URL | http://cart.jt.com/cart/delete/{userId}/{itemId} |
| 参数 | userId 用户ID  itemId 商品ID |
| 示例 | http://cart.jt.com/cart/delete/2/6 |
| 返回值 | {  status: 200  msg: “OK”  data:null  } |
|  |  |

## 购物车

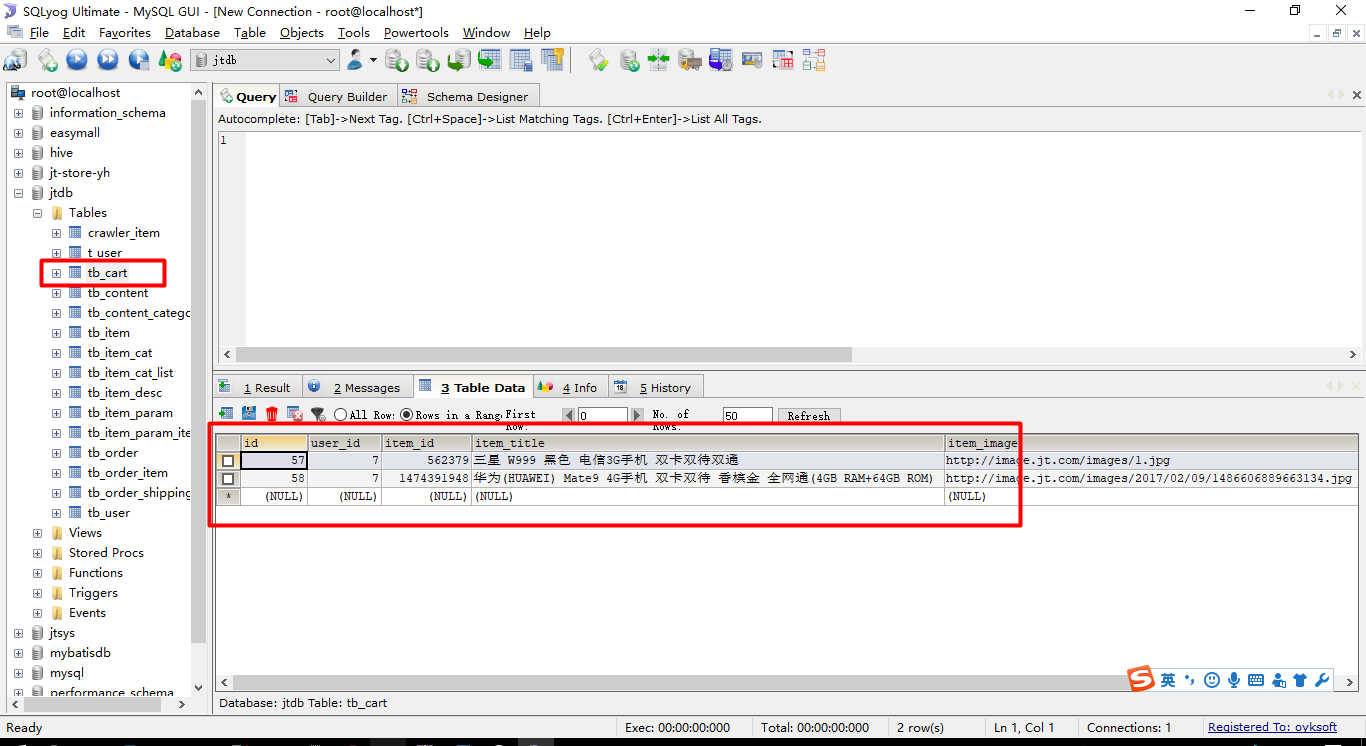
### 全新技术栈



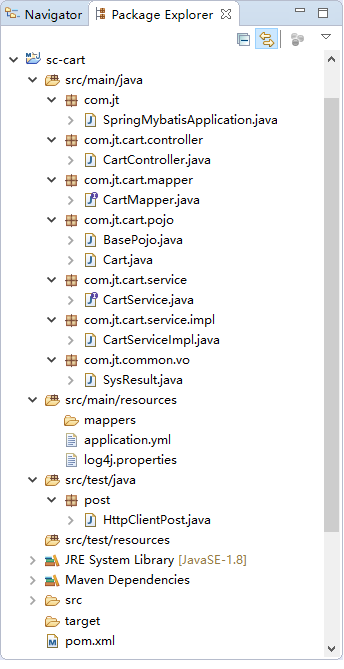
### 系统架构图



### 使用到京淘的购物车表tb\_cart



### 创建Maven工程



### pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>sc-mybatis</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>sc-mybatis</name>

<url>http://maven.apache.org</url>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.3.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<!-- mybatisplus与springboot整合 -->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatisplus-spring-boot-starter</artifactId>

<version>1.0.5</version>

</dependency>

<!-- MP 核心库 -->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus</artifactId>

<version>2.1.8</version>

</dependency>

<dependency>

<groupId>com.github.pagehelper</groupId>

<artifactId>pagehelper-spring-boot-starter</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.datatype</groupId>

<artifactId>jackson-datatype-joda</artifactId>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.module</groupId>

<artifactId>jackson-module-parameter-names</artifactId>

</dependency>

<!-- alibaba的druid数据库连接池 -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid-spring-boot-starter</artifactId>

<version>1.1.0</version>

</dependency>

</dependencies>

</project>

### BasePojo.java

package com.jt.cart.pojo;

import java.io.Serializable;

import java.util.Date;

import com.baomidou.mybatisplus.annotations.TableField;

import com.baomidou.mybatisplus.enums.FieldFill;

public class BasePojo implements Serializable {

private static final long serialVersionUID = 1L;

@TableField(value = "created",fill = FieldFill.INSERT)

private Date created;

@TableField(value = "updated",fill = FieldFill.INSERT\_UPDATE)

private Date updated;

public Date getCreated() {

return created;

}

public void setCreated(Date created) {

this.created = created;

}

public Date getUpdated() {

return updated;

}

public void setUpdated(Date updated) {

this.updated = updated;

}

}

### Cart.java

package com.jt.cart.pojo;

import com.baomidou.mybatisplus.annotations.TableField;

import com.baomidou.mybatisplus.annotations.TableId;

import com.baomidou.mybatisplus.annotations.TableName;

import com.baomidou.mybatisplus.enums.IdType;

@TableName("tb\_cart") //类和表的映射

public class Cart extends BasePojo{

private static final long serialVersionUID = 1L;

//@TableField(exist=false) //数据库表中不存在的字段要标识

@TableId(value="id",type=IdType.AUTO) //主键自增

private Long id;

@TableField("user\_id") //属性和字段的映射

private Long userId;

@TableField("item\_id")

private Long itemId;

@TableField("item\_title")

private String itemTitle;

@TableField("item\_image")

private String itemImage;

@TableField("item\_price")

private Long itemPrice;

private Integer num;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public Long getUserId() {

return userId;

}

public void setUserId(Long userId) {

this.userId = userId;

}

public Long getItemId() {

return itemId;

}

public void setItemId(Long itemId) {

this.itemId = itemId;

}

public String getItemTitle() {

return itemTitle;

}

public void setItemTitle(String itemTitle) {

this.itemTitle = itemTitle;

}

public String getItemImage() {

return itemImage;

}

public void setItemImage(String itemImage) {

this.itemImage = itemImage;

}

public Long getItemPrice() {

return itemPrice;

}

public void setItemPrice(Long itemPrice) {

this.itemPrice = itemPrice;

}

public Integer getNum() {

return num;

}

public void setNum(Integer num) {

this.num = num;

}

}

### SysResult.java

package com.jt.common.vo;

/\*\*

\* 京淘商城自定义响应结构

\*/

public class SysResult {

// 响应业务状态

/\*

\* 200 成功

\* 201 错误

\* 400 参数错误

\*/

private Integer status;

// 响应消息

private String msg;

// 响应中的数据

private Object data;

public static SysResult build(Integer status, String msg, Object data) {

return new SysResult(status, msg, data);

}

public static SysResult ok(Object data) {

return new SysResult(data);

}

public static SysResult ok() {

return new SysResult(null);

}

public SysResult() {

}

public static SysResult build(Integer status, String msg) {

return new SysResult(status, msg, null);

}

public SysResult(Integer status, String msg, Object data) {

this.status = status;

this.msg = msg;

this.data = data;

}

public SysResult(Object data) {

this.status = 200;

this.msg = "OK";

this.data = data;

}

public Boolean isOK() {

return this.status == 200;

}

public Integer getStatus() {

return status;

}

public void setStatus(Integer status) {

this.status = status;

}

public String getMsg() {

return msg;

}

public void setMsg(String msg) {

this.msg = msg;

}

public Object getData() {

return data;

}

public void setData(Object data) {

this.data = data;

}

}

### CartMapper.java

继承MybatisPlus提供的BaseMapper，它会自动产生SQL语句。采用注解开发方式，SQL直接写在方法上面。

特殊方法通过自定义接口实现CartMapper，通用的方法通过BaseMapper实现。

这里的方法和业务接口的方法不一致，业务接口中的部分方法基类BaseMapper已经实现，其他方法按业务需要规划。

package com.jt.cart.mapper;

import java.util.Map;

import org.apache.ibatis.annotations.Delete;

import org.apache.ibatis.annotations.Select;

import org.apache.ibatis.annotations.Update;

import org.springframework.stereotype.Repository;

import com.baomidou.mybatisplus.mapper.BaseMapper;

import com.jt.cart.pojo.Cart;

@Repository

public interface CartMapper extends BaseMapper<Cart>{

/\*\*

\* 按用户id和商品的id来查询商品是否存在

\*/

@Select("select count(1) from tb\_cart where user\_id=#{userId} and item\_id=#{itemId}")

public Integer queryByUserIdItemId(Map<String,Long> map);

/\*\*

\* 根据用户id和商品id，更新数据

\* 注意updateNum方法名称不能叫update，会和BaseMapper中重复，无法正确执行

\*/

@Update("update tb\_cart"

+ " set num = #{num}"

+ " where user\_id=#{userId} and item\_id=#{itemId}")

public void updateNum(Cart cart);

/\*\*

\* 删除某个人的商品

\*/

@Delete("delete from tb\_cart where user\_id=#{userId} and item\_id=#{itemId}")

public void deleteItem(Cart cart);

}

### CartService.java

package com.jt.cart.service;

import com.jt.cart.pojo.Cart;

import com.jt.common.vo.SysResult;

public interface CartService {

public SysResult save(Cart cart);

public SysResult query(Long userId, Integer page, Integer rows);

public SysResult updateNum(Cart cart);

public SysResult deleteItem(Cart cart);

}

### CartServiceImpl.java

业务逻辑：如果某用户的商品存在在购物车中，则修改商品数量；如果某用户的商品不存在在购物车中，则新增此商品。

package com.jt.cart.service.impl;

import java.util.Date;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.baomidou.mybatisplus.mapper.EntityWrapper;

import com.github.pagehelper.PageHelper;

import com.github.pagehelper.PageInfo;

import com.jt.cart.mapper.CartMapper;

import com.jt.cart.pojo.Cart;

import com.jt.cart.service.CartService;

import com.jt.common.vo.SysResult;

@Service

public class CartServiceImpl implements CartService {

@Autowired

private CartMapper cartMapper;

//将某个人的商品加入到购物车

public SysResult save(Cart cart){

try{

//判断，这个货物是否已经在本购物车中。必须按itemId+userId

Map<String,Long> mapParam = new HashMap<String,Long>();

mapParam.put("userId", cart.getUserId());

mapParam.put("itemId", cart.getItemId());

Integer count = cartMapper.queryByUserIdItemId(mapParam);

if(count>0){

//查询某人的购物车

EntityWrapper<Cart> wrapper = new EntityWrapper<Cart>();

wrapper.eq("user\_id", cart.getUserId());

List<Cart> cartList = cartMapper.selectList(wrapper);

if(cartList != null && cartList.size()>0){

Cart oldCart = cartList.get(0);

oldCart.setNum(oldCart.getNum()+1);

this.updateNum(oldCart);

}

return SysResult.build(202, "该商品已经存在购物车中！");

}else{

cart.setCreated(new Date());

cart.setUpdated(cart.getCreated());

cartMapper.insert(cart);

return SysResult.ok();

}

}catch(Exception e){

e.printStackTrace();

return SysResult.build(201, "添加商品到购物车失败!");

}

}

//查询某个人的购物车数据

public SysResult query(Long userId, Integer page, Integer rows){

Cart param = new Cart();

param.setUserId(userId);

EntityWrapper<Cart> wrapper = new EntityWrapper<Cart>();

wrapper.setEntity(param);

PageHelper.startPage(page, rows);

List<Cart> cartList = cartMapper.selectList(wrapper);

PageInfo<Cart> info = new PageInfo<Cart>(cartList);

return SysResult.ok(info);

}

//更新数量

public SysResult updateNum(Cart cart){

try{

cartMapper.updateNum(cart);

return SysResult.ok();

}catch(Exception e){

e.printStackTrace();

return SysResult.build(201, "更新商品数量错误! "+cart.getItemId());

}

}

//删除某人某个商品

public SysResult deleteItem(Cart cart){

try{

cartMapper.deleteItem(cart);

return SysResult.ok();

}catch(Exception e){

e.printStackTrace();

return SysResult.build(201, "删除商品失败!");

}

}

}

### CartController.java

package com.jt.cart.controller;

import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import com.jt.cart.pojo.Cart;

import com.jt.cart.service.CartService;

import com.jt.common.vo.SysResult;

@RestController

@RequestMapping("/cart")

public class CartController {

@Autowired

private CartService cartService;

@RequestMapping("/save")

public SysResult save(Cart cart){

return cartService.save(cart);

}

@RequestMapping("/query/{userId}")

public SysResult query(@PathVariable Long userId,

@RequestParam(value="page",defaultValue="1") Integer page,

@RequestParam(value="rows",defaultValue="20") Integer rows){

return cartService.query(userId, page, rows);

}

@RequestMapping("/update/num/{userId}/{itemId}/{num}")

public SysResult update(@PathVariable Long userId, @PathVariable Long itemId, @PathVariable Integer num){

Cart param = new Cart();

param.setUserId(userId);

param.setItemId(itemId);

param.setNum(num);

param.setUpdated(new Date());

return cartService.updateNum(param);

}

@RequestMapping("/delete/{userId}/{itemId}")

public SysResult deleteItem(@PathVariable Long userId, @PathVariable Long itemId){

Cart param = new Cart();

param.setUserId(userId);

param.setItemId(itemId);

return cartService.deleteItem(param);

}

}

### application.yml

server:

port: 8070

spring:

datasource:

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/jtdb?jdbc:mysql://127.0.0.1:3306/jtdb?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

username: root

password: root

mybatis:

mapUnderscoreToCamelCase: true

typeAliasesPackage: com.jt.cart.pojo

mapperLocations: classpath:mappers/\*.xml

mybatis-plus:

configuration:

map-underscore-to-camel-case: true

logging:

level:

com.jt.cart.mapper: debug

### log4j.properties

打印mybatis执行的SQL语句，方便检查错误。

log4j.rootLogger=DEBUG, Console

#Console

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n

log4j.logger.java.sql.ResultSet=INFO

log4j.logger.org.apache=INFO

log4j.logger.java.sql.Connection=DEBUG

log4j.logger.java.sql.Statement=DEBUG

log4j.logger.java.sql.PreparedStatement=DEBUG

### RunApp.java

注意需要扫描的目录必须是它的子目录！

package com.jt;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

@MapperScan("com.jt.cart.mapper") //扫描Mybatis接口文件

public class RunApplication {

public static void main(String[] args) {

SpringApplication.run(RunApplication.class, args);

}

}

### 模拟POST请求工具类

模拟POST请求工具类，进行save方法的测试

package post;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.apache.http.HttpEntity;

import org.apache.http.HttpResponse;

import org.apache.http.client.entity.UrlEncodedFormEntity;

import org.apache.http.client.methods.HttpPost;

import org.apache.http.entity.StringEntity;

import org.apache.http.impl.client.CloseableHttpClient;

import org.apache.http.impl.client.HttpClients;

import org.apache.http.message.BasicNameValuePair;

import org.apache.http.util.EntityUtils;

import org.json.JSONObject;

import org.junit.Test;

public class HttpClientPost {

@Test

public void form() throws Exception {

String url = "http://localhost:8070/cart/save";

Map<String, String> paraMap = new HashMap<String,String>();

paraMap.put("userId", "7");

paraMap.put("itemId", "562379");

paraMap.put("itemTitle", "title");

paraMap.put("itemImage", "image");

paraMap.put("itemPrice", "100");

paraMap.put("num", "1");

String result = httpPostWithForm(url,paraMap);

System.out.println(result);

}

public static String httpPostWithForm(String url, Map<String, String> paraMap) throws Exception {

HttpPost httpPost = new HttpPost(url);

CloseableHttpClient client = HttpClients.createDefault();

String respContent = null;

//遍历参数map，设置表单参数

List<BasicNameValuePair> pairList = new ArrayList<BasicNameValuePair>();

for (Map.Entry<String, String> entry : paraMap.entrySet()) {

pairList.add(new BasicNameValuePair(entry.getKey(), entry.getValue()));

}

httpPost.setEntity(new UrlEncodedFormEntity(pairList, "utf-8"));

HttpResponse resp = client.execute(httpPost);

if (resp.getStatusLine().getStatusCode() == 200) {

HttpEntity he = resp.getEntity();

respContent = EntityUtils.toString(he, "UTF-8");

}

return respContent;

}

@Test

public void json() throws Exception {

String url = "http://localhost:8070/cart/query/7?page=1&rows=1";

String result = httpPostWithJSON(url);

System.out.println(result);

}

public static String httpPostWithJSON(String url) throws Exception {

HttpPost httpPost = new HttpPost(url);

CloseableHttpClient client = HttpClients.createDefault();

String respContent = null;

// json方式

JSONObject jsonParam = new JSONObject();

jsonParam.put("name", "admin");

jsonParam.put("pass", "123456");

StringEntity entity = new StringEntity(jsonParam.toString(), "utf-8");// 解决中文乱码问题

entity.setContentEncoding("UTF-8");

entity.setContentType("application/json");

httpPost.setEntity(entity);

HttpResponse resp = client.execute(httpPost);

if (resp.getStatusLine().getStatusCode() == 200) {

HttpEntity he = resp.getEntity();

respContent = EntityUtils.toString(he, "UTF-8");

}

return respContent;

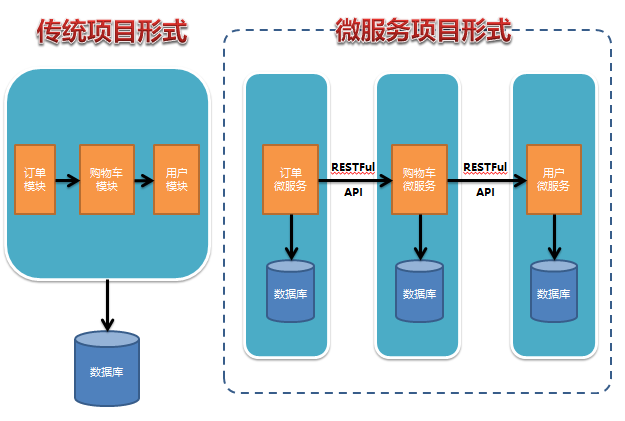
}

}

# 第三天：Euraka注册中心+Ribbon负载均衡

## SpringCloud介绍

### 微服务架构演化



简而言之，微服务就是开发一组小型服务的方式来开发一个独立的应用系统，每个小型服务都运行在自己的进程中，并采用HTTP资源API轻量级的机制来互相通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据库存储技术。

### 微服务的优点

1. **易于开发和维护**

业务分拆，一个微服务只关注一个特定的业务功能，所以它的业务清晰、代码量较少。开发和维护单个微服务相对简单。每个微服务业务复杂度低，方便理解、维护和调试。整个应用由若干个微服务构成。

1. **单个微服务启动快**

单个微服务代码量少，启动比庞大的项目要快。

1. **局部修改容易部署**

传统单体项目修改一个功能就需要重新部署整个应用，而微服务只需对需要修改的服务重新部署，其他服务无需停止，甚至不相关的业务仍然可以继续执行。

1. **技术栈不受限**

在微服务中，支持技术异构，可以根据软件团队擅长的技术去实现，如java、c#、c、php等，也支持异构数据库mysql、oracle、sqlServer等。

### 定义

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Spring Cloud就是一个全家桶，整合了市面上最好最先进的技术，形成一个工具集，并简化其中的操作，引领编程新方式。

### 官网

官网： http://projects.spring.io/spring-cloud/

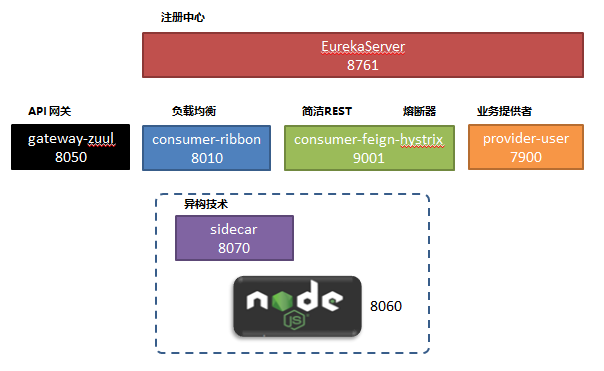
手册： http://cloud.spring.io/spring-cloud-static/Dalston.SR2/

中文： https://springcloud.cc/

### 核心功能

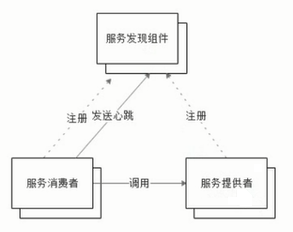
* configuration management 配置中心
* service discovery 服务发现
* circuit breakers 断路器
* intelligent routing 智能路由
* micro-proxy 微代理
* control bus 控制总线
* one-time tokens 一次性令牌
* global locks 全局锁
* leadership election 选举算法
* distributed sessions 分布式会话
* cluster state 集群状态

### 核心组件关系图



## 注册中心Eureka

### 注册中心



注意它的特点，结构类似于MessageQueue消息队列，服务（提供者、消费者）先都注册到注册中心。它的特点在于，不会每次都去注册中心获取，而是有本地缓存，加快访问性能。内部含有心跳机制，当注册中心信息改变，自动快速获取新的信息到本地。心跳机制还保证分布式环境下，某个服务失败后，自动列表从注册中心移除。注册中心中保证所有可用的链接。

### 拓展：CAP定理

CAP原则又称CAP定理，指的是在一个分布式系统中，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。它是分布式系统中最核心最重要的理论。

分布式系统的CAP理论：理论首先把分布式系统中的三个特性进行了如下归纳：

* 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）
* 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
* 分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

CAP理论就是说在分布式系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。所以我们只能在一致性和可用性之间进行权衡，要么选择CP要么选择AP，没有分布式系统能同时保证这三点。

### 拓展：分布式对关系型数据库的冲击

对于web网站来说，关系数据库的很多主要特性却往往无用武之地

* 数据库事务一致性需求

很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。

* 数据库的写实时性和读实时性需求

对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。如：MQ消息队列机制，意义，可以解决瞬时的高并发，消峰去谷作用。

* 对复杂的SQL查询，特别是多表关联查询的需求

任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

SNS：全称Social Networking Services，专指社交网络服务，包括了社交软件和社交网站。例如：脸谱facebook、腾讯QQ、微信等。

### ZooKeeper和Eureka对比

Eureka本身是Netflix开源的一款提供服务注册和发现的产品，并且提供了相应的Java封装。在它的实现中，节点之间相互平等，部分注册中心的节点挂掉也不会对集群造成影响，即使集群只剩一个节点存活，也可以正常提供发现服务。哪怕是所有的服务注册节点都挂了，Eureka Clients（客户端）上也会缓存服务调用的信息。这就保证了我们微服务之间的互相调用足够健壮。

Zookeeper主要为大型分布式计算提供开源的分布式配置服务、同步服务和命名注册。曾经是Hadoop项目中的一个子项目，用来控制集群中的数据，目前已升级为独立的顶级项目。很多场景下也用它作为Service发现服务解决方案。

**对比**

根据著名的CAP定理（C-数据一致性；A-服务可用性；P-服务对网络分区故障的容错性CAP这三个特性在任何分布式系统中不能同时满足，最多同时满足两个CP或者AP）。

**ZooKeeper**

Zookeeper是基于CP来设计的，即任何时刻对Zookeeper的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性，但是它不能保证每次服务请求的可用性。从实际情况来分析，在使用Zookeeper获取服务列表时，如果zookeeper正在选主，或者Zookeeper集群中半数以上机器不可用，那么将无法获得数据。所以说，Zookeeper不能保证服务可用性。

诚然，在大多数分布式环境中，尤其是涉及到数据存储的场景，数据一致性应该是首先被保证的，这也是zookeeper设计成CP的原因。但是对于服务发现场景来说，情况就不太一样了：针对同一个服务，即使注册中心的不同节点保存的服务提供者信息不尽相同，也并不会造成灾难性的后果。因为对于服务消费者来说，能消费才是最重要的——拿到可能不正确的服务实例信息后尝试消费一下，也好过因为无法获取实例信息而不去消费。（尝试一下可以快速失败，之后可以更新配置并重试）所以，对于服务发现而言，可用性比数据一致性更加重要——AP胜过CP。

**Eureka**

而Spring Cloud Netflix在设计Eureka时遵守的就是AP原则。Eureka Server也可以运行多个实例来构建集群，解决单点问题，但不同于ZooKeeper的选举leader的过程，Eureka Server采用的是Peer to Peer对等通信。这是一种去中心化的架构，无master/slave区分，每一个Peer都是对等的。在这种架构中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的serviceUrl指向其他节点。每个节点都可被视为其他节点的副本。

如果某台Eureka Server宕机，Eureka Client的请求会自动切换到新的Eureka Server节点，当宕机的服务器重新恢复后，Eureka会再次将其纳入到服务器集群管理之中。当节点开始接受客户端请求时，所有的操作都会进行replicateToPeer（节点间复制）操作，将请求复制到其他Eureka Server当前所知的所有节点中。

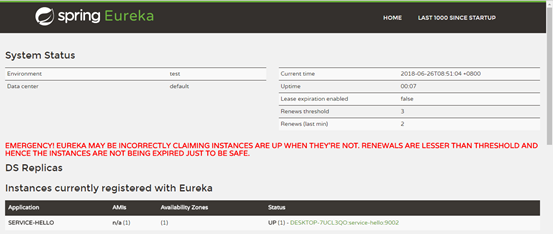
一个新的Eureka Server节点启动后，会首先尝试从邻近节点获取所有实例注册表信息，完成初始化。Eureka Server通过getEurekaServiceUrls()方法获取所有的节点，并且会通过心跳续约的方式定期更新。默认配置下，如果Eureka Server在一定时间内没有接收到某个服务实例的心跳，Eureka Server将会注销该实例（默认为90秒，通过eureka.instance.lease-expiration-duration-in-seconds配置）。当Eureka Server节点在短时间内丢失过多的心跳时（比如发生了网络分区故障），那么这个节点就会进入自我保护模式。

**总结**

ZooKeeper基于CP，不保证高可用，如果zookeeper正在选主，或者Zookeeper集群中半数以上机器不可用，那么将无法获得数据。Eureka基于AP，能保证高可用，即使所有机器都挂了，也能拿到本地缓存的数据。作为注册中心，其实配置是不经常变动的，只有发版（发布新的版本）和机器出故障时会变。对于不经常变动的配置来说，CP是不合适的，而AP在遇到问题时可以用牺牲一致性来保证可用性，既返回旧数据，缓存数据。

所以理论上Eureka是更适合作注册中心。而现实环境中大部分项目可能会使用ZooKeeper，那是因为集群不够大，并且基本不会遇到用做注册中心的机器一半以上都挂了的情况。所以实际上也没什么大问题。

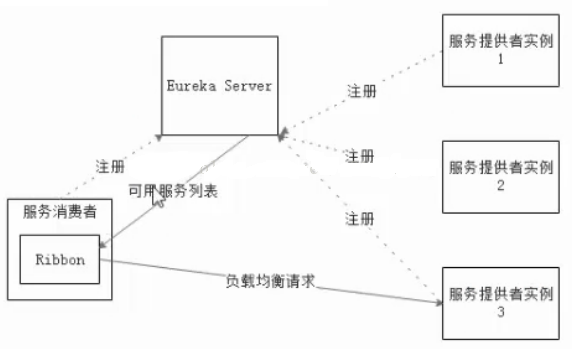
### 自我保护模式



什么是自我保护模式？默认配置下，如果Eureka Server每分钟收到心跳续约的数量低于一个阈值（instance的数量(60/每个instance的心跳间隔秒数)自我保护系数），并且持续15分钟，就会触发自我保护。在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。当它收到的心跳数重新恢复到阈值以上时，该Eureka Server节点就会自动退出自我保护模式。它的设计哲学前面提到过，那就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。该模式可以通过eureka.server.enable-self-preservation = false来禁用，同时eureka.instance.lease-renewal-interval-in-seconds可以用来更改心跳间隔。

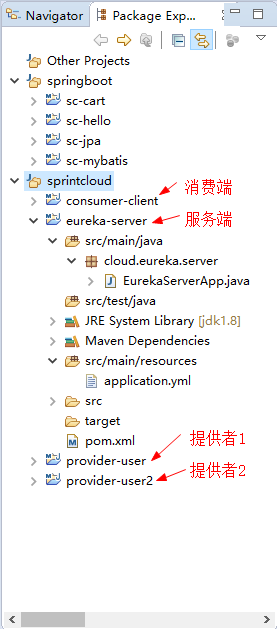
### 调用关系图

常见的服务发现产品：Eureka/ZooKeeper/Consul，Eureka轻量级。



## Eureka服务端

### 创建Maven工程



### pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>eureka.server</groupId>

<artifactId>eureka-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath />

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

security:

basic:

enabled: true

user:

name: user

password: password123

server:

port: 8761

eureka:

client:

register-with-eureka: false

fetch-registry: false

service-url:

defaultZone: http://user:password123@localhost:8761/eureka

logging:

level:

root: INFO

### EurekaServerApp.java

package cloud.eureka.server;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer

@SpringBootApplication

public class EurekaServerApp {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApp.class, args);

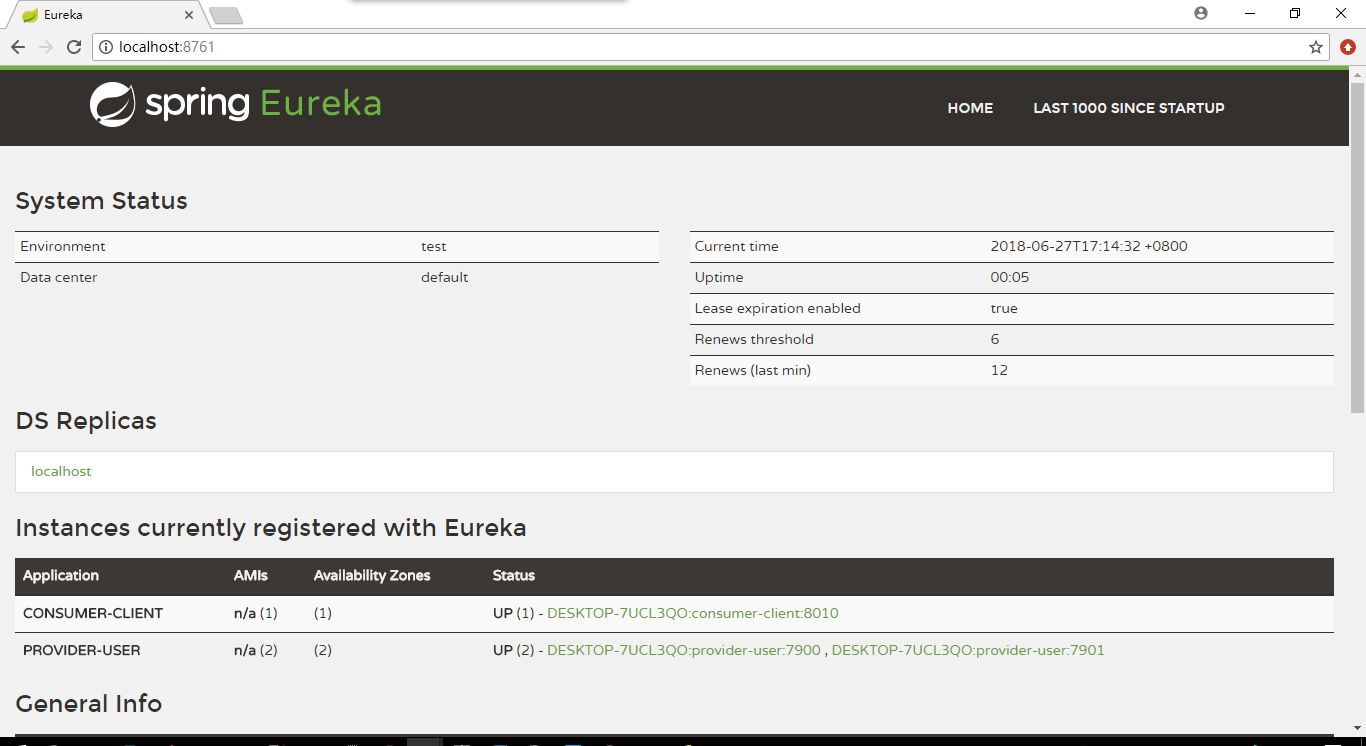
}

}

### 监控地址

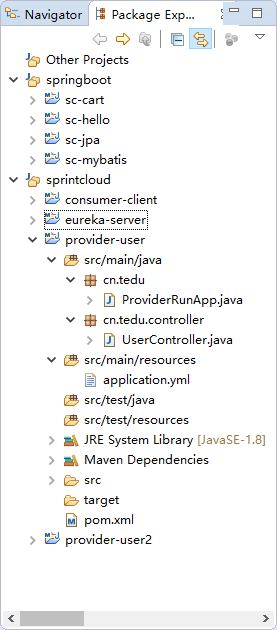
地址：http://localhost:8761

启动后形成一个控制台



## 服务提供者

### 创建Maven工程



### pom.xml

注意和Eureka服务端不同

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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.tedu</groupId>

<artifactId>h1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

注意多个提供端的差异在于端口不同，将来同一个业务服务多实例如何标识呢？就是通过端口来标识。

server:

port: 7900

spring:

application:

name: provider-user

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

logging:

level:

root: INFO

### HelloController.java

package cn.tedu.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@GetMapping("/hello/{name}")

public String hello(@PathVariable String name) {

return "1:"+name;

}

}

### ProviderRunApp.java

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class ProviderRunApp {

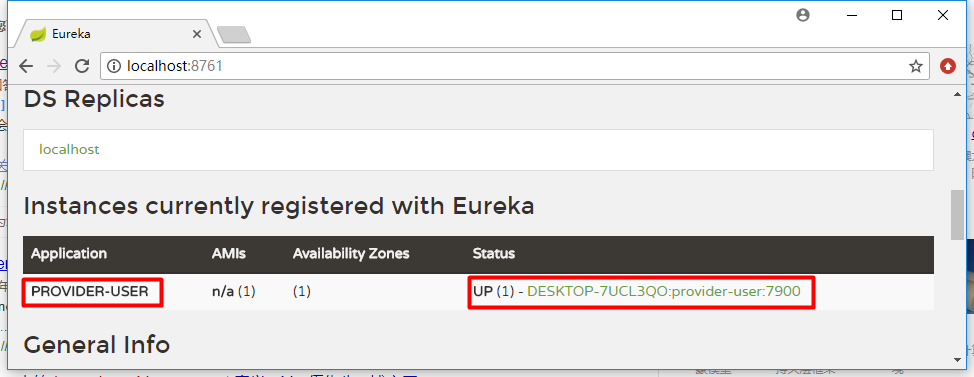
public static void main(String[] args) {

SpringApplication.run(ProviderRunApp.class, args);

}

}

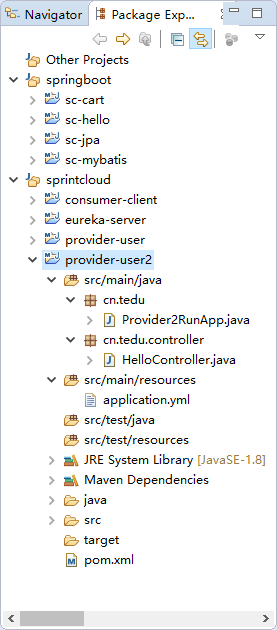
### 查看服务



## 服务提供者2

### 创建Maven工程

创建项目，复制文件进行修改。主要



### application.yml

修改端口为7901

server:

port: 7901

spring:

application:

name: provider-user

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

logging:

level:

root: INFO

### HelloController.java

修改打印的页面展现的值，方便观察是哪个提供者响应的。

package cn.tedu.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@GetMapping("/hello/{name}")

public String hello(@PathVariable String name) {

return "2:"+name;

}

}

### Provider2RunApp.java

修改类名

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class Provider2RunApp {

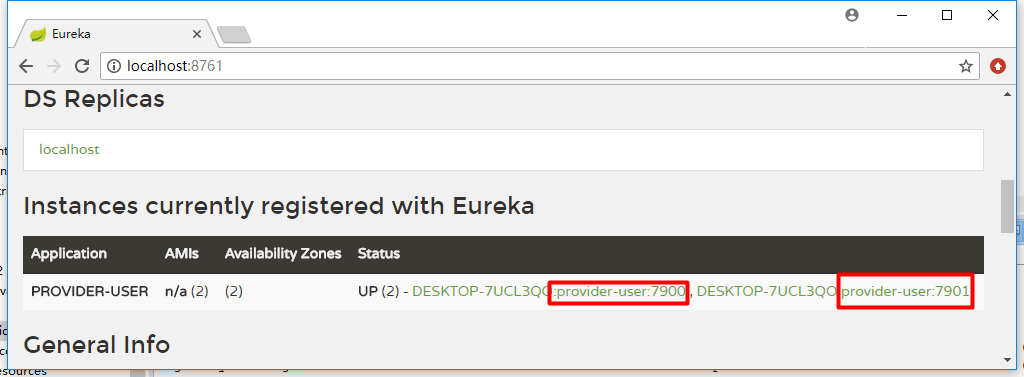
public static void main(String[] args) {

SpringApplication.run(Provider2RunApp.class, args);

}

}

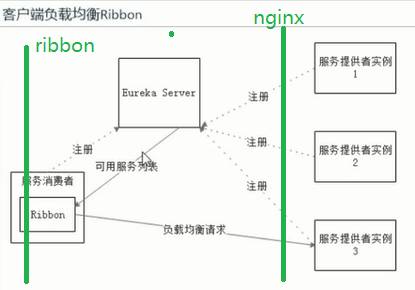
### 查看服务



## 负载均衡Ribbon

### Ribbon

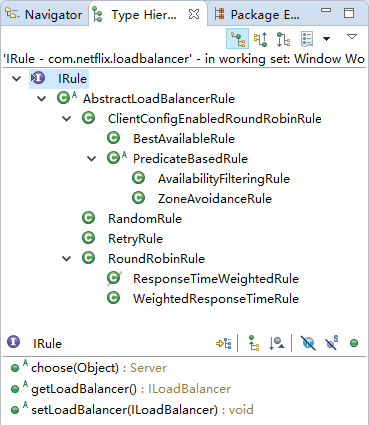
和nginx不同，它是客户端侧负载均衡。



### 负载均衡策略

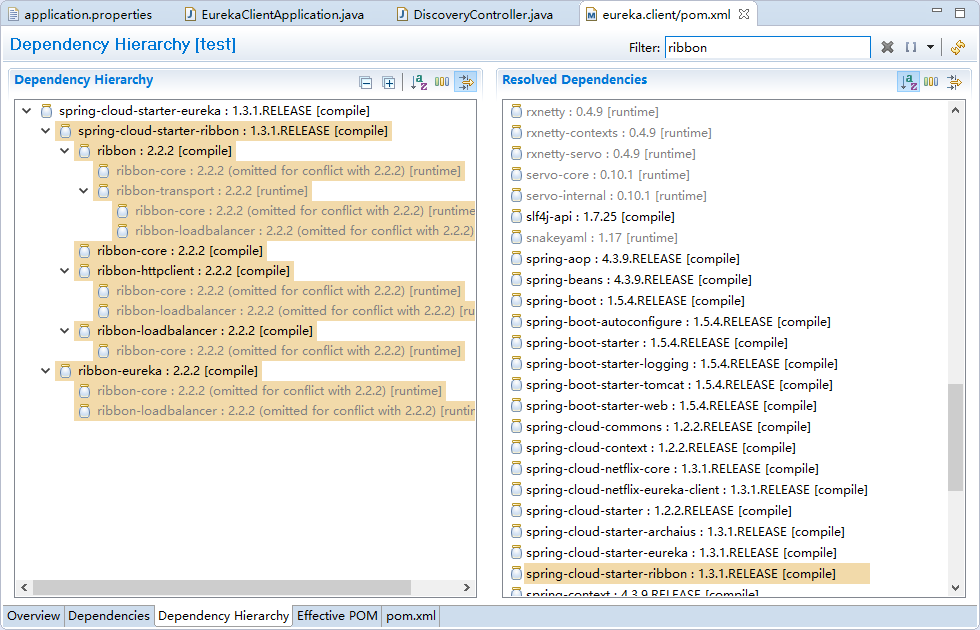
常见提供的负载均衡算法有三种：

* 第一种也是默认为轮询
* 第二种为random随机
* 第三种为WeightedResponseTimeRule，响应时间



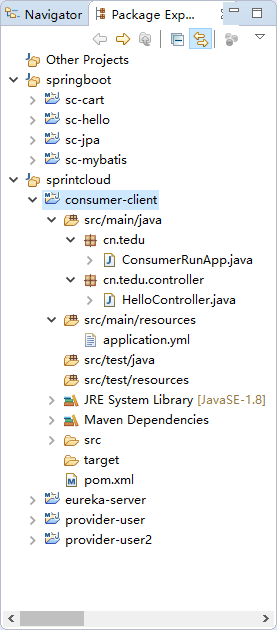
### 导包

无需引入jar包，在spring-cloud-start-euraka已经依赖了ribbon的jar包。



## 消费者Ribbon

### 创建Maven工程



### pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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.tedu</groupId>

<artifactId>h1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

server:

port: 8010

spring:

application:

name: consumer-client

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

logging:

level:

root: INFO

### HelloController.java

RestTemplate对象是在RunApp中声明并创建的，用它才可以实现负载均衡，同时注意url中的地址为VIP虚拟IP，为application.yml中配置的application-name。

package cn.tedu.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

@RestController

public class HelloController {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/hello/{name}")

@ResponseBody

public String hello(@PathVariable String name){

String url = "http://provider-user/hello/"+name; //VIP虚拟IP，提供者的application-name:provider-user

return this.restTemplate.getForObject(url, String.class);

}

}

### ConsumerRunApp.java

重点在启动时要初始化RestTemplate对象，同时设置@LoadBalanced负载均衡

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

@SpringBootApplication

@EnableEurekaClient

public class ConsumerRunApp {

@Bean

@LoadBalanced //Ribbon负载均衡

public RestTemplate restTemplate(){

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(ConsumerRunApp.class, args);

}

}

### 测试

执行顺序：

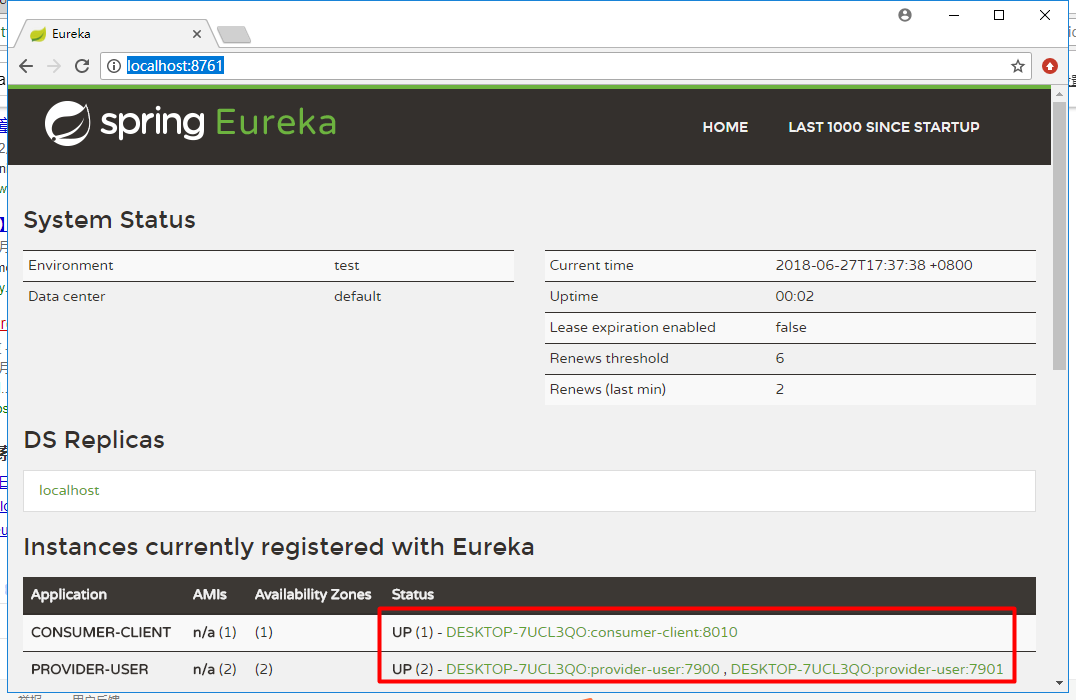
先启动服务端 8761 eureka-server EurekaServerApp

在启动提供者1 7900 provider-user ProviderRunApp

在启动提供者2 7901 provider-user2 Provider2RunApp

最后启动消费者 8010 consumer-client ConsumerRunApp

访问Eureka控制台： http://localhost:8761/



访问请求： http://localhost:8010/hello/tony

交替出现1:tony和2:tony，说明两个提供者交替执行。这里注意有时可能提供者2还未准备好不能工作，多刷一会就正常了。可以看出Ribbon默认的负载均衡策略是轮询。

## Ribbon随机负载算法

### RibbonRuleConfig.java

自定义规则扩展对象

package cn.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.netflix.loadbalancer.IRule;

import com.netflix.loadbalancer.RandomRule;

/\*\*

\*

\* 自定义Ribbon配置

\* 规定：这个类不能再@ComponentScan和@SpringBootApplication本包和子包下，否则引起@RibbonClients扫描冲突

\* 注意：随机第一次打断点进入，之后多次刷新就不进入，可能由于本地缓存原因

\*/

@Configuration

public class RibbonRuleConfig {

@Bean

public IRule ribbonRule(){

return new RandomRule();

}

}

### ConsumerRunApp.java

增加一个注解@RibbonClient即可

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.ribbon.RibbonClient;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

import cn.config.RibbonRuleConfig;

@SpringBootApplication

@EnableEurekaClient

@RibbonClient(name="provider-user", configuration=RibbonRuleConfig.class)

public class ConsumerRunApp {

@Bean

@LoadBalanced //Ribbon负载均衡

public RestTemplate restTemplate(){

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(ConsumerRunApp.class, args);

}

}

# 第四天：Feign简化REST+Hystrix断路器

## Feigh

### 概念

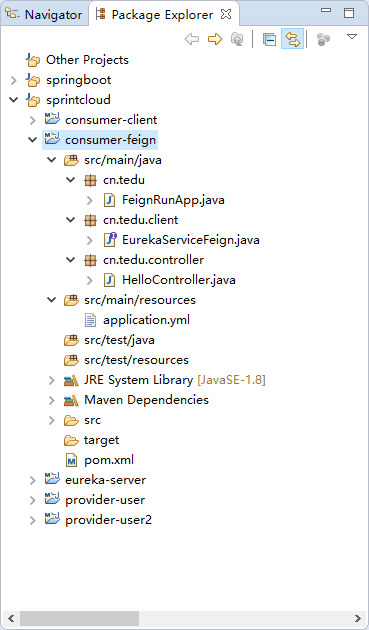
Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

Feigh是一个声明式web服务客户端。它能让开发web服务变得容易。使用Feign需要创建一个接口并注解它。它拥有包括Feign注解和JAX-RS注解的可插拔支持。它还支持可插拔的编码器和解码器。Spring Cloud拥有Spring MVC支持，并使用Spring Web中默认同样的HttpMessageConverters。在使用Feign时，Spring Cloud集成了Ribbon和Eureka来提供负载均衡的HTTP客户端。

总结：Feign简化HttpClient开发，封装了JAX-RS和springmvc的注解，学习成本很低。

## 消费者实现Feign

### 创建Maven工程



### pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>consumer-feign</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!-- Hystrix，Feign是基于Hystrix的 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<!-- Eureka依赖，连接注册中心的都需要有这个依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<!-- Feign依赖，声明式开发 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

server:

port: 9001

spring:

application:

name: consumer-feign

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

### EurekaServiceFeign.java

package cn.tedu.client;

import org.springframework.cloud.netflix.feign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

// 这个接口相当于把原来的服务提供者项目当成一个Service类

@FeignClient(value="provider-user")

public interface EurekaServiceFeign {

/\*

\* Feign中没有原生的@GetMapping/@PostMapping/@DeleteMapping/@PutMapping，要指定需要用method进行

\*/

@RequestMapping(value="/hello/{name}",method=RequestMethod.GET)

public String hello(@PathVariable("name") String name);

}

### HelloController.java

package cn.tedu.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import cn.tedu.client.EurekaServiceFeign;

@RestController

public class HelloController {

@Autowired

private EurekaServiceFeign eurekaServiceFeign;

@GetMapping("/hello/{name}")

@ResponseBody

public String hello(@PathVariable String name){

return eurekaServiceFeign.hello(name);

}

}

### FeignRunApp.java

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.cloud.client.SpringCloudApplication;

import org.springframework.cloud.netflix.feign.EnableFeignClients;

@SpringCloudApplication

@EnableFeignClients //开启Feign

public class FeignRunApp {

public static void main(String[] args) {

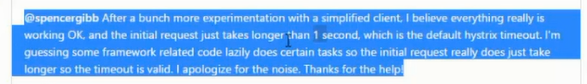
SpringApplication.run(FeignRunApp.class, args);

}

}

### 常见的坑

* 只支持旧的@RequestMapping不支持新的@GetMapping()等
* method=RequestMethod.GET 只支持POST方法，GET方法也会自动转到POST方法
* @PathVariable("name") String name 必须写名称“name”，springmvc可以不写
* 启动超过1s会报超时，再刷新就可以正常访问，原因在于hystrix断路器的影响，稍差的机器就会发生。



### 测试

执行顺序：

先启动服务端 8761 eureka-server EurekaServerApp

在启动提供者1 7900 provider-user ProviderRunApp

最后启动消费者 9001 consumer-client ConsumerRunApp

访问Eureka控制台： http://localhost:8761/

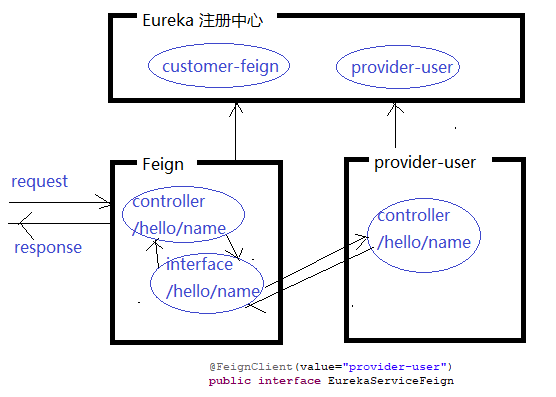


访问请求： http://localhost:9001/hello/hellen

出现1:hellen，页面出现，设置断点，发现提供者也进入了，Feign消费者也进入了。

## Feign小结

### 调用过程

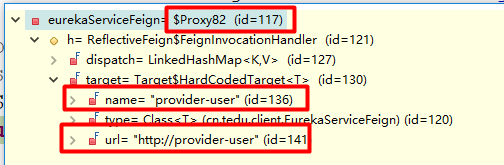


首先，提供者provider-user和消费者custorm-feign都注册到Eureka中。用户请求feign中的controller，feign中的controller调用feign定义的接口方法。接口的方法根据注解去找到eureka注册中心中的provider-user地址，然后请求远程provider-user所在服务器的地址，然后调用远程的provider-user提供者的具体服务。提供者响应返回json，json被feign封装传输给“接口”的返回值，“接口”在返回给feign的controller，最终响应给用户。

### 自动创建实现类

Feign是典型的基于接口，基于动态代理技术自动生成代理对象。

设置断点，可以清晰的看到这个过程。先访问consumer-feign的controller；观察EurekaServiceFeign接口，如下图其实一个jdk动态代理类。



动态代理类根据配置的注解信息去Eureka中找到对应调用的提供者的链接信息，进行访问，断点就进入到provider-user的controller中具体执行，执行完成层层返回。

## 断路器Hystrix

### 微服务设计引发新的问题



微服务的设计，服务分散在多个服务器上，服务之间互相调用，要调用的服务由于跨网络跨服务器调用，响应速度明显比传统项目单机调用慢很多，甚至由于网络涌动的不稳定的现象发生导致调用超时；还有类似级联失败、雪崩效应（依赖的基础服务宕机，关联的服务导致失败甚至宕机，就像滚雪球一样层层失败。）

如何解决这类新的问题呢？传统的机制就是超时机制。

### 超时机制

良好的设计，在通过网络请求其他服务时，都必须设置超时时间。正常情况下，一个远程调用几十毫秒内返回。当要调用的服务不可用时或者网络问题，响应时间要等超时，如HttpClient几十秒才超时返回。通常，一次远程调用对应一个线程/进程，如果大量的线程/进程得不到释放，并且越积越多，服务资源就会被耗尽，从而导致资深服务不可用。所以必须为每个请求设置超时时间。

例如：我们熟悉的tomcat就有超时设计

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" acceptCount="500" maxThreads="400" />

但我们发现传统的超时设计时间都比较久，面对今天互联网时代，用户对性能的极致要求时间显得太久了？20秒，用户要求网页秒级响应，等20秒，用户只能跟你说byebye了，扭头去找别的提供商了。

特别像微服务这样基于多个服务，服务之间都是远程调用，如果一个服务长时间等待，用户体验会极差的，那怎么办呢？断路器模式应运而生。

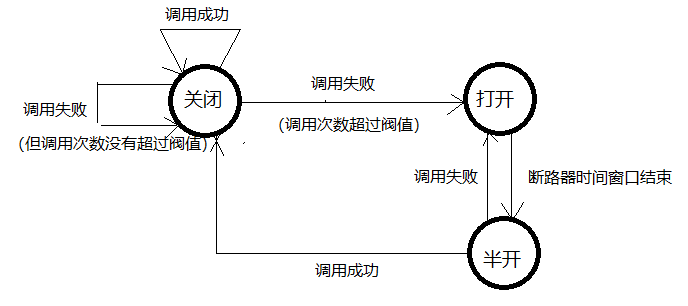
### 断路器模式

家里电表都有个断路器（俗称电闸），当使用的电器很多，用电巨大（例如功率过大、短路等），当电流过载时，电路就会升温，甚至烧断电路，引起火灾。有了这个断路器，我们及时拉闸，就不会造成严重后果了。

断路器可以实现快速失败，如果它在一段时间内检测到许多失败，如超时，就会强迫其以后的多个调用快速失败，不再请求所依赖的服务，从而防止应用程序不断地尝试执行可能会失败的操作，这样应用程序可以继续执行而不用等待修正错误，或者浪费CPU时间去等待长时间的超时。断路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。

断路器模式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。

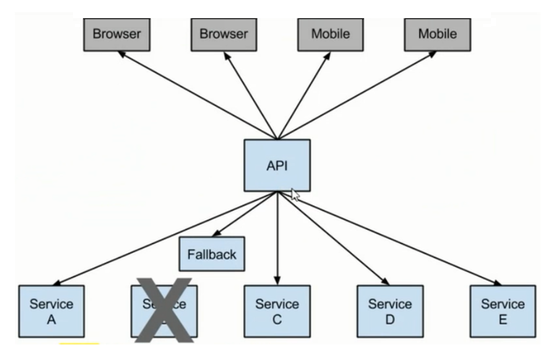
### 状态



断路器有三种状态：

1. 关闭：当访问没有问题时，断路器处于关闭未使用。
2. 打开：当访问开始出现异常，错误次数增多，达到阀值时就会打开断路器，这样服务直接访问断路器，进行快速失败返回。
3. 半开：那服务一直走断路器，系统就没法用了，万一被调用的服务以及稳定了呢。断路器的优势就来了，过一定时间窗口后（若干秒）它就会自动分流一部分服务再去尝试访问之前失败的服务。如果继续失败，那就不再转发，如果成功了，成功率高了，那会关闭断路器。

### 结构图



当服务B不可用时，开发人员需要写一个Fallback快速失败响应。可以设置为一个固定的值或者一个空值。

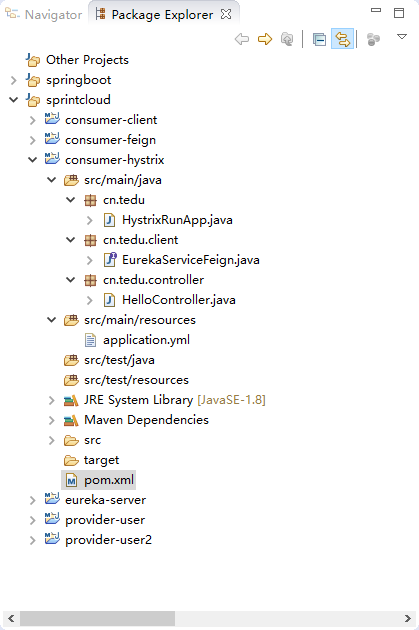
### 小结

* 断路器快速失效，调用者快速返回失败信息毫不犹豫
* 不是立即宣布死亡，而是设置阀值，超越阀值才宣布死亡
* 传统方式死亡了就不管了，可断路器超级厉害，“死了也不放过”。自带心跳机制，自动测试路径是否可用，如发现又“活了”，会自动恢复调用关系。这一切开发者都无需编写代码

这样的设计太牛了，考虑非常细致全面。不可用时立即响应，可用时自动恢复。不是网络抖动下，就永远宣布其死亡。这样的设计思想非常适合网络这种不稳定的应用场景。

## 消费者实现Hystrix

### 创建Maven项目



### pom.xml

增加hystrix依赖

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>consumer-feign</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!-- Hystrix，Feign是基于Hystrix的 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<!-- Eureka依赖，连接注册中心的都需要有这个依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<!-- Feign依赖，声明式开发 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

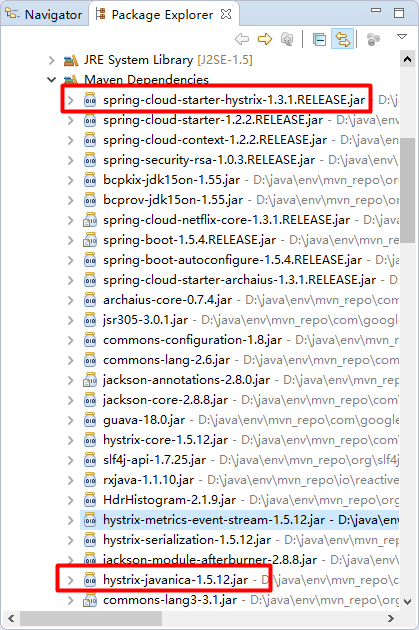
</dependency>

</dependencies>

</dependencyManagement>

</project>

### 导入hystrix支持



Spring cloud没有直接使用NetFlix开源的Hystrix，而使用的是javanica，javanica在Hystrix基础上利用反射和注解技术增强和简化了Hystrix的开发。

### application.yml

server:

port: 9001

spring:

application:

name: consumer-feign-hystrix

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

### EurekaServiceFeign.java

package cn.tedu.client;

import org.springframework.cloud.netflix.feign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

// 这个接口相当于把原来的服务提供者项目当成一个Service类

@FeignClient(value="provider-user")

public interface EurekaServiceFeign {

/\*

\* Feign中没有原生的@GetMapping/@PostMapping/@DeleteMapping/@PutMapping，要指定需要用method进行

\*/

@RequestMapping(value="/hello/{name}",method=RequestMethod.GET)

public String hello(@PathVariable("name") String name);

}

### HelloController.java

package cn.tedu.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

import cn.tedu.client.EurekaServiceFeign;

@RestController

public class HelloController {

@Autowired

private EurekaServiceFeign eurekaServiceFeign;

@GetMapping("/hello/{name}")

@ResponseBody

@HystrixCommand(fallbackMethod = "helloFallback")

public String hello(@PathVariable String name){

return eurekaServiceFeign.hello(name);

}

//对应上面的方法，参数必须一致，当访问失败时，hystrix直接回调用此方法

public String helloFallback(String name){

return "tony"; //失败调用时，返回默认值

}

}

### HystrixRunApp.java

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.cloud.client.SpringCloudApplication;

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

import org.springframework.cloud.netflix.feign.EnableFeignClients;

@SpringCloudApplication

@EnableFeignClients //开启feign

@EnableCircuitBreaker //开启hystrix

public class HystrixRunApp {

public static void main(String[] args) {

SpringApplication.run(HystrixRunApp.class, args);

}

}

### 测试

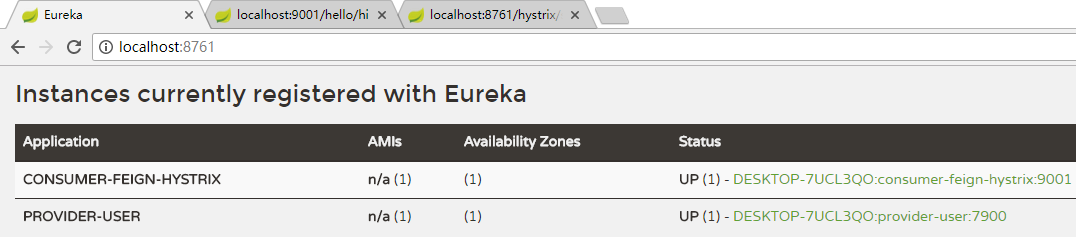
执行顺序：

先启动服务端 8761 eureka-server EurekaServerApp

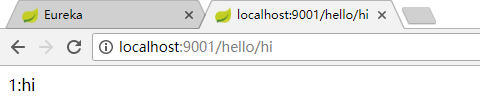
在启动提供者1 7900 provider-user ProviderRunApp

最后启动消费者 9001 consumer-feign-hystrix HystrixRunApp

访问Eureka控制台： http://localhost:8761/



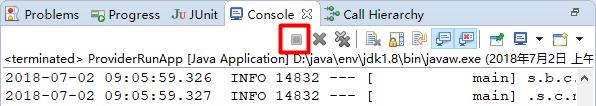
访问请求： http://localhost:9001/hello/hi



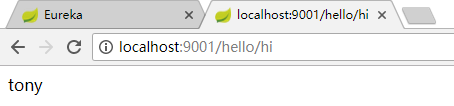
注意：可能访问太快时第一次就出现断路器实现的”tony”，这时访问太快，服务还未生效，多刷新几次，业务正常时应该出现上面的提示。

### 测试断路器

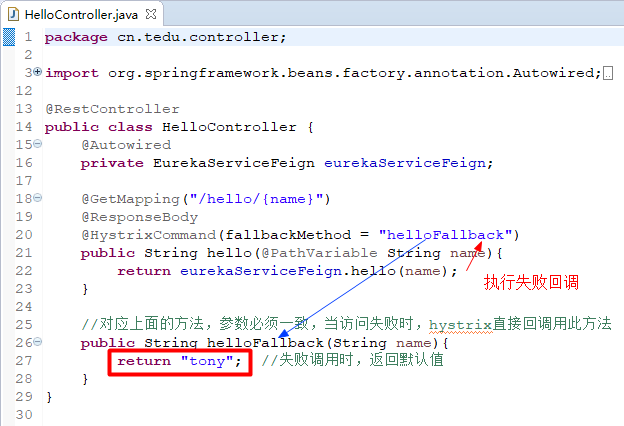
关闭provider-user，模拟服务失败宕机场景。



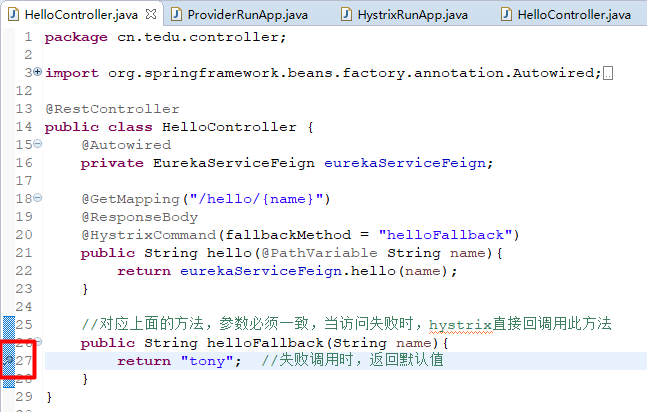
访问请求： http://localhost:9001/hello/hi



可以看到已经走了fallback方法。



重启provider-user服务，模拟服务修复，可以看到立即返回正确结果，说明服务正常时不会走断路器方法。



设置断点，和我们预想的相同，会发现只有异常时才走断路器的回调方法。

## HystrixDashBoard

### 需要依赖jar包支持

只针对当前实例，actuator是SpringBoot提供的一些监控的扩展支持

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

### pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>consumer-feign</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!-- Hystrix，Feign是基于Hystrix的 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<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>

<!-- Eureka依赖，连接注册中心的都需要有这个依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<!-- Feign依赖，声明式开发 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

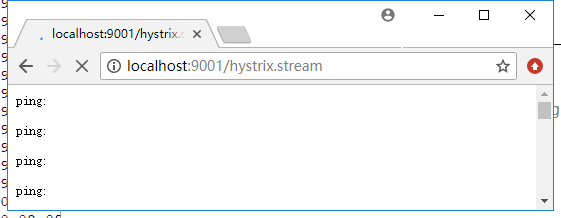
</dependencies>

</dependencyManagement>

</project>

### 查看监控日志

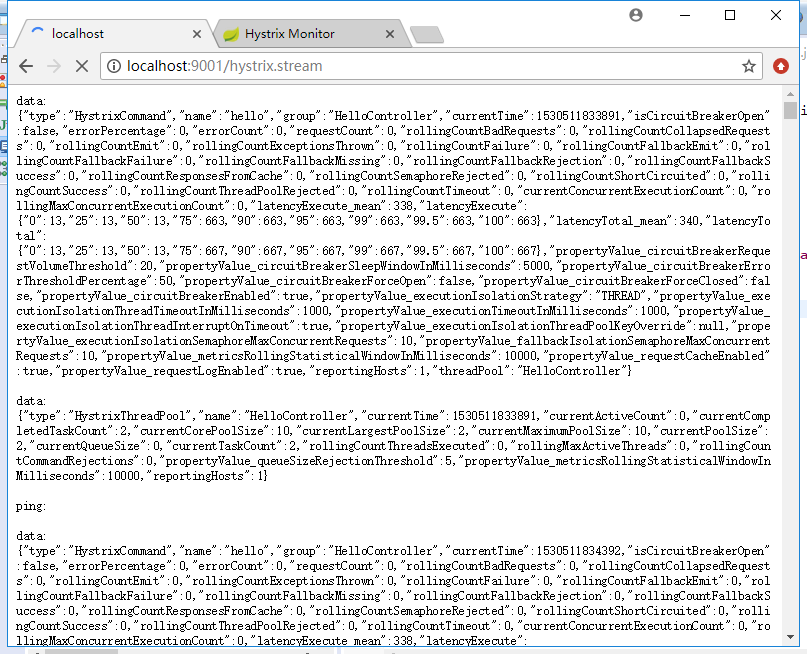
访问链接：http://localhost:9001/hystrix.stream



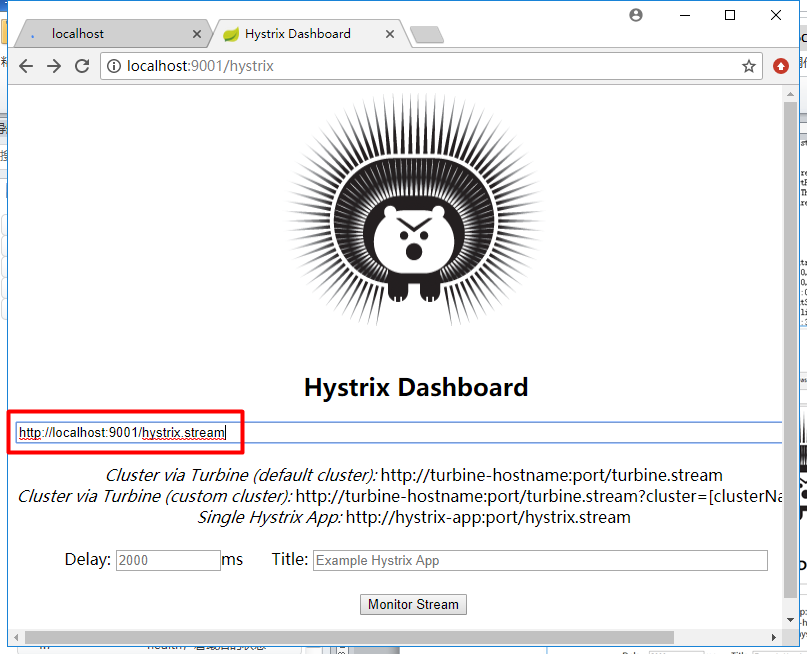
访问9001，消费者链接，展示ping:一直打印，转个不停。

访问一次业务 http://localhost:9001/hello/t

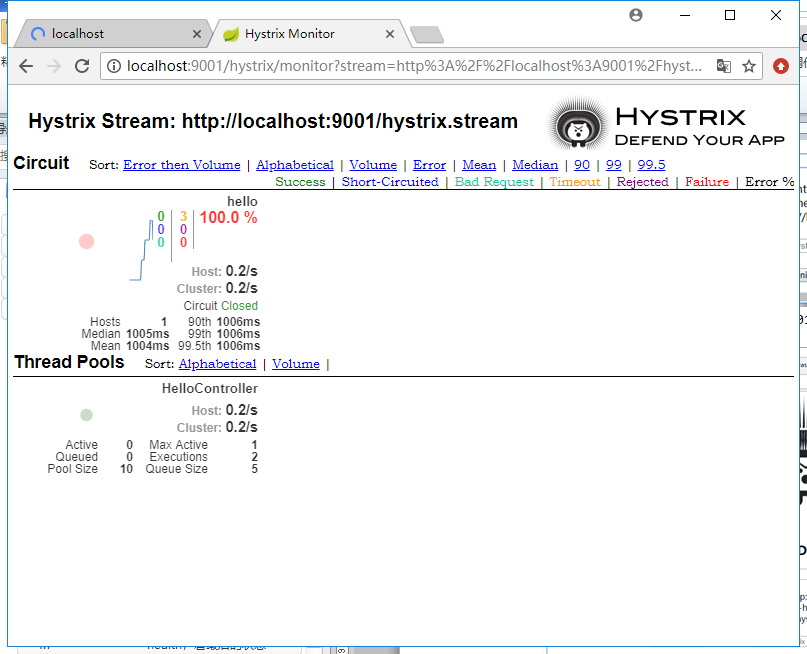
停止提供者，刷新消费者，开始打印日志信息



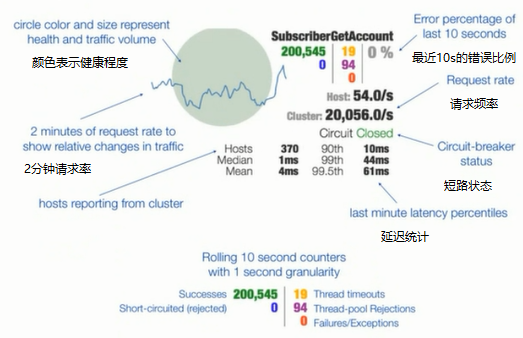
可以基于控制台访问



输入上面的链接：http://localhost:9001/hystrix.stream，点击monitor按钮



具体指标的含义为：



在监控的界面有两个重要的图形信息：一个实心圆和一条曲线。

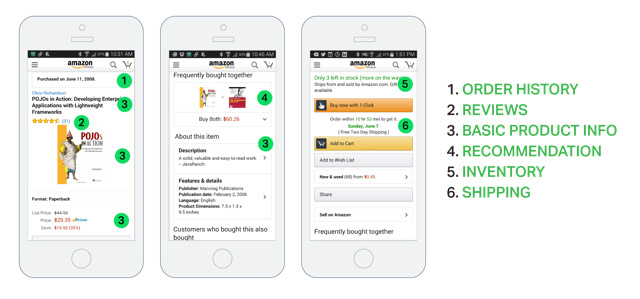
* 实心圆：1、通过颜色的变化代表了实例的健康程度，健康程度从绿色、黄色、橙色、红色递减。2、通过大小表示请求流量发生变化，流量越大该实心圆就越大。所以可以在大量的实例中快速发现故障实例和高压实例。
* 曲线：用来记录2分钟内流浪的相对变化，可以通过它来观察流量的上升和下降趋势。

# 第五天：Zull实现API网关+Sidecar异构系统调用NodeJS

## API网关

### 概念

Amazon Andriod 应用的商品详情页显示内容



* 购物车中的商品数量
* 历史订单
* 客户评论
* 低库存预警
* 送货选项
* 各种推荐，包括经常与该商品一起购买的其它商品、购买该商品的客户购买的其它商品、购买该商品的客户看过的其它商品
* 其它的购物选择

使用单体应用程序架构时，移动客户端通过向应用程序发起一次REST请求，来获取这些数据，负载均衡器将请求路由给N个相同的应用程序实例中的其中之一，然后应用程序会查询各种数据库表，并将响应返回给客户端。

相反，若是采用微服务架构，显示在产品页上的数据会分布在不同的微服务上，下面列举了可能与产品详情页数据有关的一些微服务：

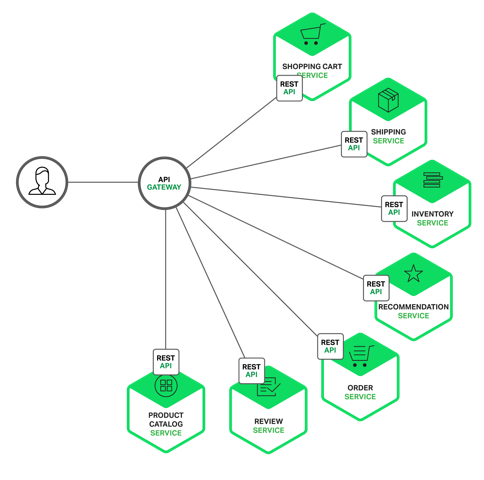
* 购物车服务~购物车中的件数
* 订单服务~历史订单
* 目录服务~商品基本信息、如名称、图片和价格
* 评论服务~客户的评论
* 库存服务~低库存预警
* 送货服务~送货选项、期限和费用，这些信息单独从送货方API获取
* 推荐服务~推荐商品

### 客户端与微服务直接通信的问题

1. 客户端需求和每个微服务暴露的细粒度 API 不匹配。在这个例子中，客户端需要发送 7 个独立请求。在更复杂的应用程序中，可能要发送更多的请求；按照 Amazon 的说法，他们在显示他们的产品页面时就调用了数百个服务。然而，客户端通过 LAN 发送许多请求，这在公网上可能会很低效，在移动网络上就根本不可行。这种方法还使得客户端代码非常复杂。
2. 客户端直接调用微服务的另一个问题是，部分服务使用的协议对 web 并不友好。一个服务可能使用 Thrift 二进制 RPC，而另一个服务可能使用 AMQP 消息传递协议。不管哪种协议对于浏览器或防火墙都不够友好，最好是内部使用。在防火墙之外，应用程序应该使用诸如 HTTP 和 WebSocket 之类的协议。
3. 另一个缺点是，它会使得微服务难以重构。随着时间推移，我们可能想要更改系统拆分服务的方式。例如，我们可能合并两个服务，或者将一个服务拆分成两个或更多服务。然而，如果客户端与微服务直接通信，那么执行这类重构就非常困难了。

### 使用 API 网关构建微服务

通常来说，使用 API 网关是更好的解决方式。API 网关是一个服务器，也可以说是进入系统的唯一节点。API 网关封装内部系统的架构，并且提供 API 给各个客户端。它还可能还具备授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等功能。下图展示了一个适应当前架构的 API 网关。



API 网关负责服务请求路由、组合及协议转换。客户端的所有请求都首先经过 API 网关，然后由它将请求路由到合适的微服务。API 网关经常会通过调用多个微服务并合并结果来处理一个请求。它可以在 web 协议（如 HTTP 与 WebSocket）与内部使用的非 web 友好协议之间转换。

API 网关还能为每个客户端提供一个定制的 API。通常，它会向移动客户端暴露一个粗粒度的 API。以产品详情的场景为例，API 网关可以提供一个端点（/productdetails?productid=xxx），使移动客户端可以通过一个请求获取所有的产品详情。API 网关通过调用各个服务（产品信息、推荐、评论等等）并合并结果来处理请求。

Netflix API 网关是一个很好的 API 网关实例。Netflix 流媒体服务提供给成百上千种类型的设备使用，包括电视、机顶盒、智能手机、游戏系统、平板电脑等等。

最初，Netflix 试图为他们的流媒体服务提供一个通用的 API。然而他们发现，由于各种各样的设备都有自己独特的需求，这种方式并不能很好地工作。如今，他们使用一个 API 网关，通过运行与针对特定设备的适配器代码，来为每种设备提供定制的 API。通常，一个适配器通过调用平均 6 到 7 个后端服务来处理每个请求。Netflix API 网关每天处理数十亿请求。

### API 网关的优点和缺点

如你所料，使用 API 网关有优点也有不足。使用 API 网关的最大优点是，它封装了应用程序的内部结构。客户端只需要同网关交互，而不必调用特定的服务。API 网关为每一类客户端提供了特定的 API，这减少了客户端与应用程序间的交互次数，还简化了客户端代码。

API 网关也有一些不足。它增加了一个我们必须开发、部署和维护的高可用组件。还有一个风险是，API 网关变成了开发瓶颈。为了暴露每个微服务的端点，开发人员必须更新 API 网关。API网关的更新过程要尽可能地简单，这很重要；否则，为了更新网关，开发人员将不得不排队等待。不过，虽然有这些不足，但对于大多数现实世界的应用程序而言，使用 API 网关是合理的。

### Zuul

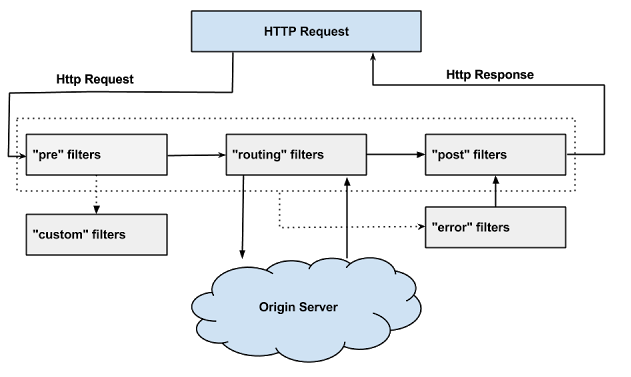
Zuul提供了一个框架，可以对过滤器进行动态的加载，编译，运行。

Zuul可以通过加载动态过滤机制，从而实现以下各项功能：

* 验证与安全保障: 识别面向各类资源的验证要求并拒绝那些与要求不符的请求。
* 审查与监控: 在边缘位置追踪有意义数据及统计结果，从而为我们带来准确的生产状态结论。
* 动态路由: 以动态方式根据需要将请求路由至不同后端集群处。
* 压力测试: 逐渐增加指向集群的负载流量，从而计算性能水平。
* 负载分配: 为每一种负载类型分配对应容量，并弃用超出限定值的请求。
* 静态响应处理: 在边缘位置直接建立部分响应，从而避免其流入内部集群。
* 多区域弹性: 跨越AWS区域进行请求路由，旨在实现ELB使用多样化并保证边缘位置与使用者尽可能接近。

除此之外，Netflix公司还利用Zuul的功能通过金丝雀版本实现精确路由与压力测试。

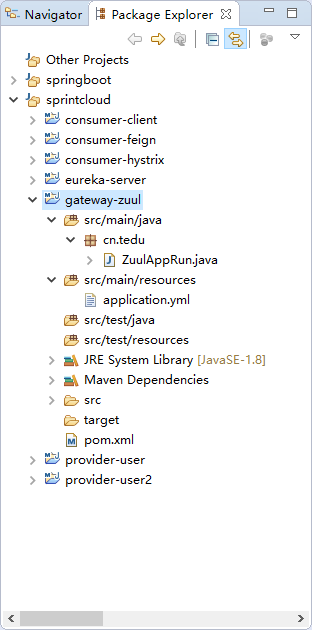
### Zuul生命周期



## Zuul实现网关



### 创建Maven项目



### pom.xml

注意zuul也要注册到注册中心Eureka上

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>consumer-feign</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

zuul的配置：app-a路由名称随意，path映射路径，访问路径就无需敲入serviceId那么长，可以利用path映射路径简写。

旧的方式：http://localhost:8050/provider-user/hello/tony

新的方式：http://localhost:8050/user/hello/tony

server:

port: 8050

spring:

application:

name: gateway-zuul

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

zuul:

routes:

app-a:

path: /user/\*\*

serviceId: provider-user

默认会把所有注册在eureka上的微服务都反向代理

server:

port: 8050

spring:

application:

name: gateway-zuul

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

zuul:

ignoredServices: '\*'

routes:

app-a:

path: /user/\*\*

serviceId: provider-user

这样配置后，只有provider-user的被映射到user，其它的服务访问形式不变，如果有多个之间用逗号隔开

直接访问： http://localhost:7900/hello/tony

未映射访问（无法访问）： http://localhost:8050/provider-user/hello/tony

ZUUL访问： http://localhost:8050/user/hello/tony

### ZuulRunApp.java

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication

@EnableZuulProxy

@EnableEurekaClient //zuul服务要注册到Eureka上

public class ZuulAppRun {

public static void main(String[] args) {

SpringApplication.run(ZuulAppRun.class, args);

}

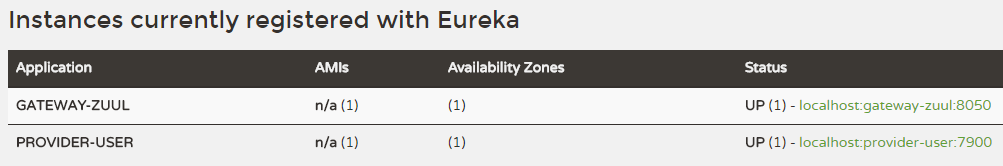
}

### 测试

启动EurekaServerApp

启动ProviderRunApp

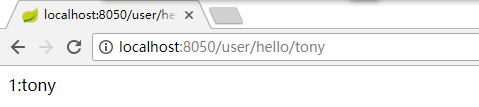
启动ZuulAppRun



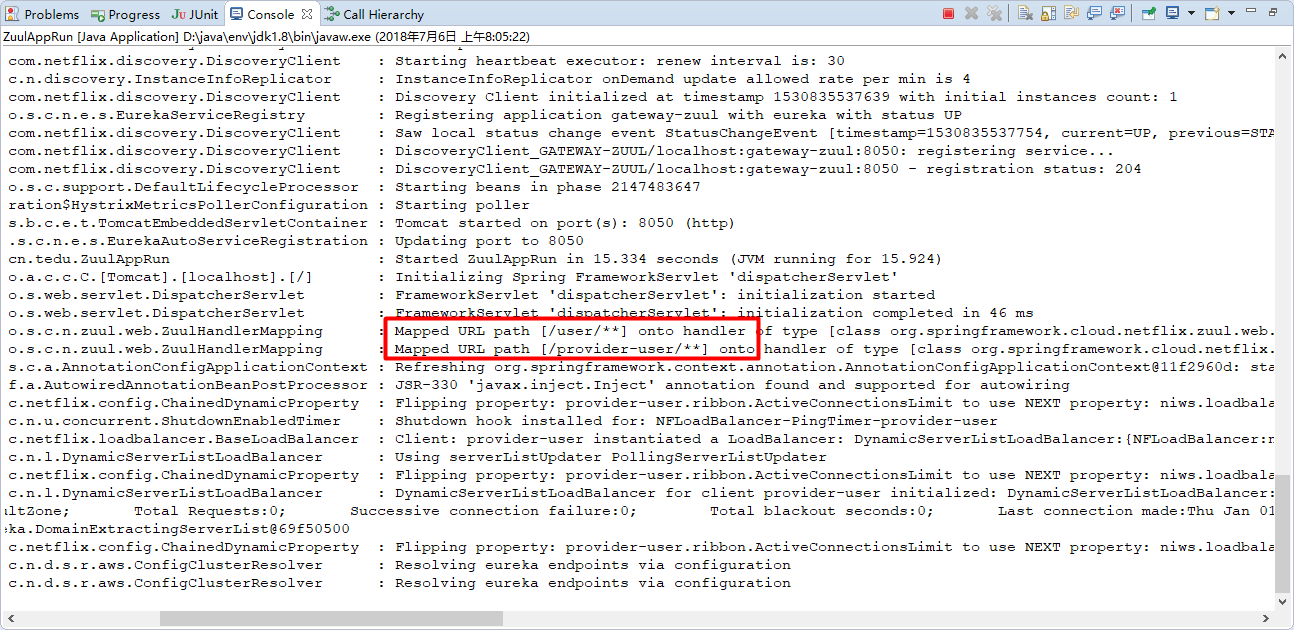
直接访问： http://localhost:7900/hello/tony

未映射访问： http://localhost:8050/provider-user/hello/tony

ZUUL访问： http://localhost:8050/user/hello/tony



### 日志



## Zuul中实现断路器

### HelloFallback.java

package cn.tedu.fallback;

import java.io.ByteArrayInputStream;

import java.io.IOException;

import java.io.InputStream;

import org.springframework.cloud.netflix.zuul.filters.route.ZuulFallbackProvider;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.MediaType;

import org.springframework.http.client.ClientHttpResponse;

import org.springframework.stereotype.Component;

@Component //Zuul实现熔断机制

public class HelloFallback implements ZuulFallbackProvider{

@Override

public String getRoute() {

return "provider-user";

}

@Override

public ClientHttpResponse fallbackResponse() {

return new ClientHttpResponse() {

@Override

public HttpHeaders getHeaders() {

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON\_UTF8);

return headers;

}

@Override

public InputStream getBody() throws IOException {

return new ByteArrayInputStream(("fallback "+HelloFallback.this.getRoute()).getBytes());

}

@Override

public String getStatusText() throws IOException {

return HttpStatus.BAD\_REQUEST.getReasonPhrase();

}

@Override

public HttpStatus getStatusCode() throws IOException {

return HttpStatus.BAD\_REQUEST;

}

@Override

public int getRawStatusCode() throws IOException {

return HttpStatus.BAD\_REQUEST.value();

}

@Override

public void close() {

}

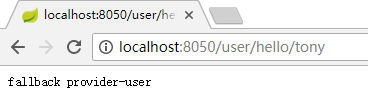
};

}

}

### 测试

执行业务正常，停掉provider-user服务，断路器生效



但这样实现的断路器刷新时间比较长，一旦服务正常，刷新正常。

## NodeJS

### 简介

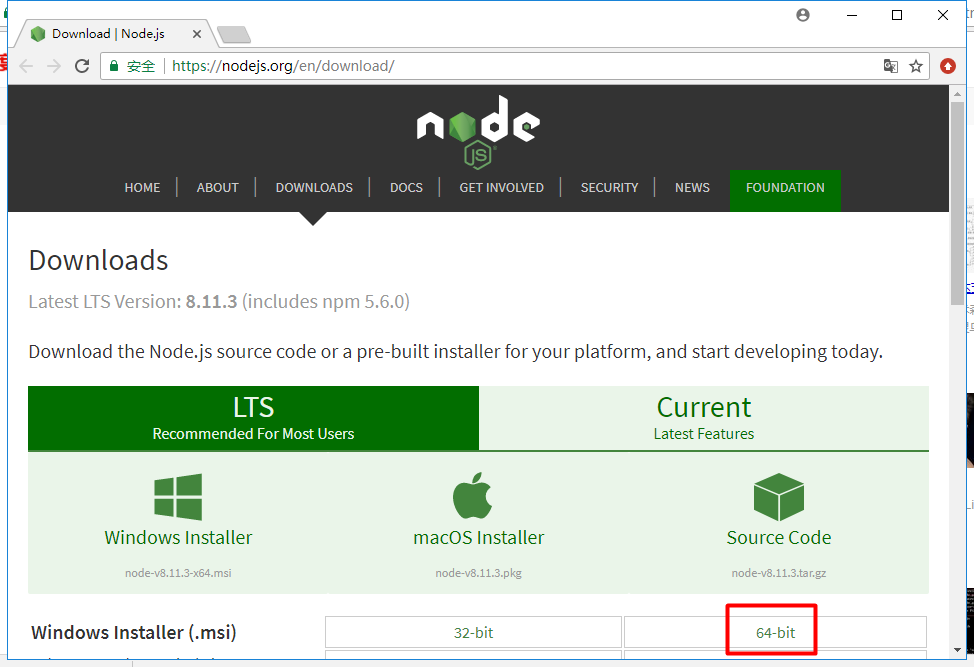


V8引擎执行JavaScript的速度非常快，性能非常好。 Node.js是一个基于Chrome JavaScript运行时建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。Node.js 使用事件驱动，非阻塞I/O 模型而得以轻量和高效，非常适合在分布式设备上运行数据密集型的实时应用。

### 安装

Windows下非常好安装，一路NEXT即可。

node-v8.11.3-x64.msi



### NodeJs代码

var http = require('http');

var url = require('url');

var path = require('path');

//创建server

var server = http.createServer(function(req, res){

//获得请求路径

var pathname = url.parse(req.url).pathname;

res.writeHead(200, {'Content-Type':'application/json; charset=utf-8'});

if(pathname === '/'){

res.end(JSON.stringify({ "index":"欢迎" }));

}else if(pathname === '/health.json'){

res.end(JSON.stringify({ "status":"UP" }));

}else{

res.end("404");

}

});

//创建监听，并打印日志

server.listen(8060, function(){

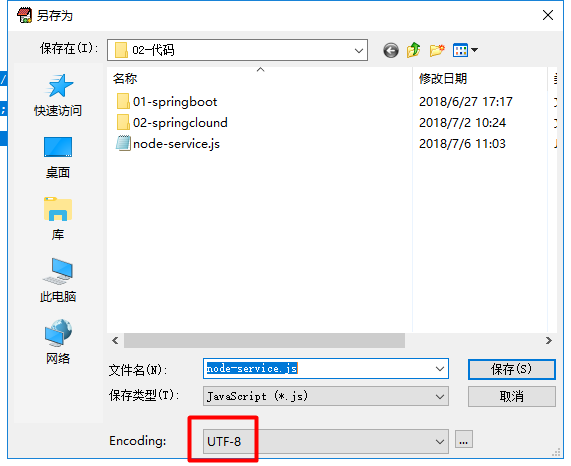
console.log('listening on localhost:8060');

});

注意：开发的Node.js应用，必须去实现一个健康检查接口，来让Sidecar可以把这个服务实例的健康状况报告给Eureka。

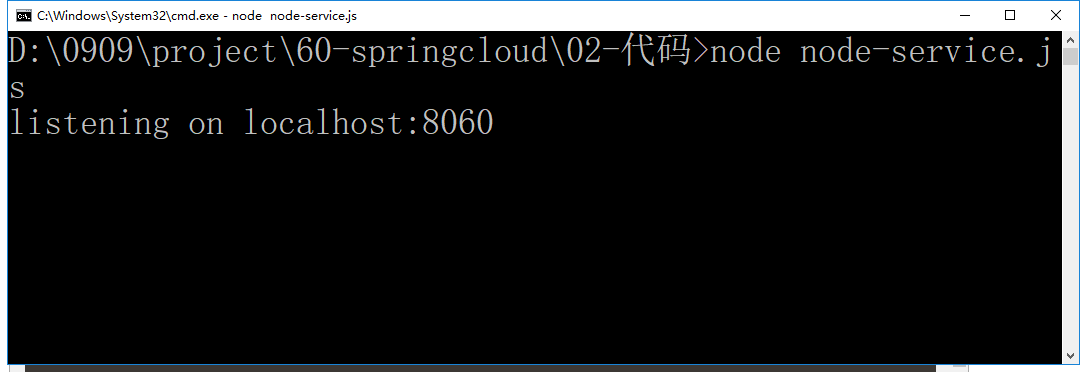
### 中文乱码

注意：文件编码必须是UTF-8，否则中文乱码



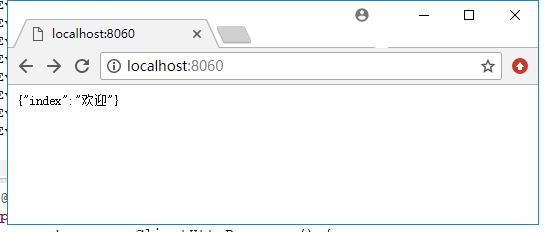
### 启动nodeServer

安装完成nodejs，就可以在dos窗口中执行node命令，后面跟编写的node-service.js文件。

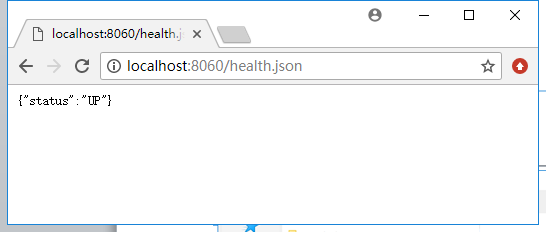


### 测试

访问首页



访问监控检查



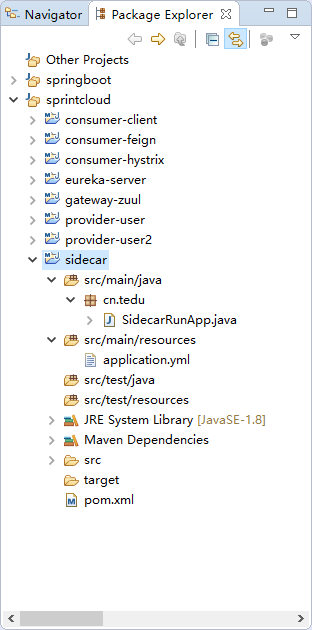
访问不存在页面



## Sidecar异构系统

通过sidecar来调用nodejs的服务。

### 创建Maven项目



### pom.xml

注意引入spring-cloud-netflix-sidecar，不是start开头的

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://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>cn.tedu</groupId>

<artifactId>consumer-feign</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-sidecar</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

### application.yml

server:

port: 8070

spring:

application:

name: sidecar

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

sidecar:

port: 8060

health-uri: http://localhost:8060/health.json

### SidecarRunApp.java

package cn.tedu;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.sidecar.EnableSidecar;

@SpringBootApplication

@EnableSidecar

public class SidecarRunApp {

public static void main(String[] args) {

SpringApplication.run(SidecarRunApp.class, args);

}

}

注意：

To enable the Sidecar, create a Spring Boot application with @EnableSidecar. This annotation includes @EnableCircuitBreaker,

@EnableDiscoveryClient, and @EnableZuulProxy. Run the resulting application on the same host as then non-jvm application.

官网上写了异构程序必须在一个host下部署，否则要配置${eureka.instance.hostName}

### Zuul的application.yml

增加app-sidecar的映射。#为备注，yml文件中不能使用/tab键，只能空格

server:

port: 8050

spring:

application:

name: gateway-zuul

eureka:

client:

serviceUrl:

defaultZone: http://user:password123@localhost:8761/eureka

zuul:

ignoredServices: '\*'

routes:

app-provider-user: #名称随意，唯一就好

path: /user/\*\*

serviceId: provider-user

app-sidecar: #通过sidecar调用nodejs的服务

path: /sidecar/\*\*

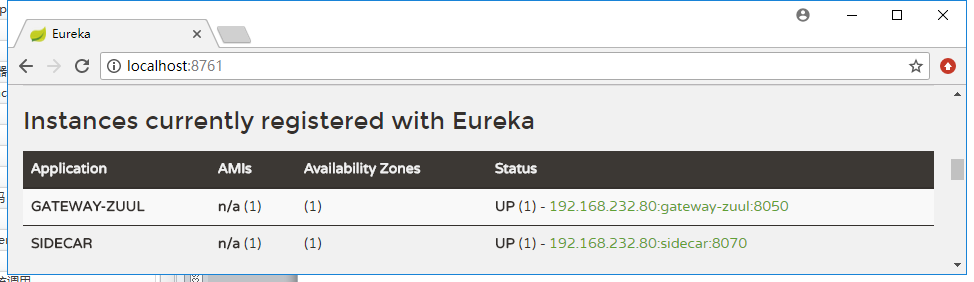
serviceId: sidecar

### 测试

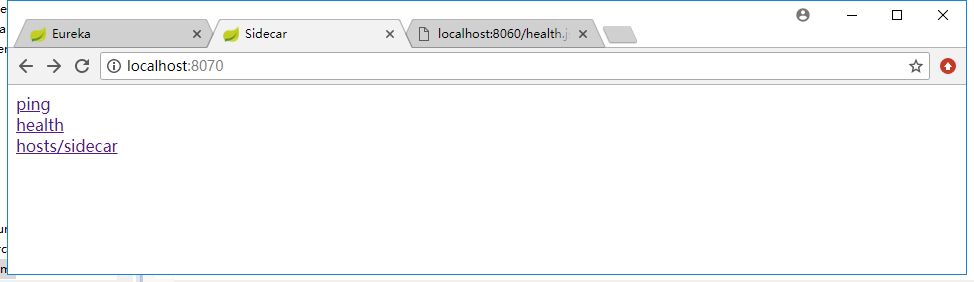
启动EurekaServerApp

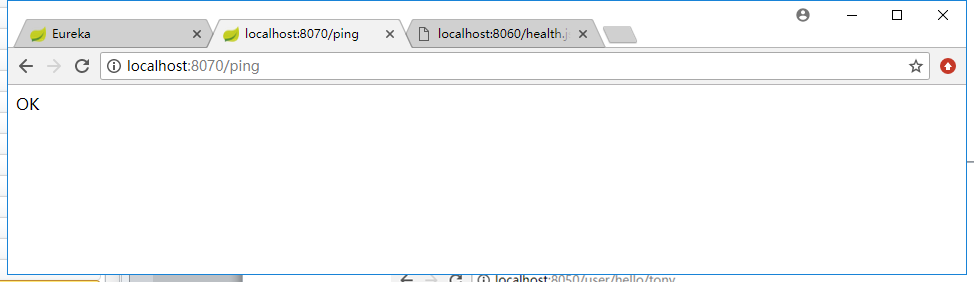
启动ZuulAppRun

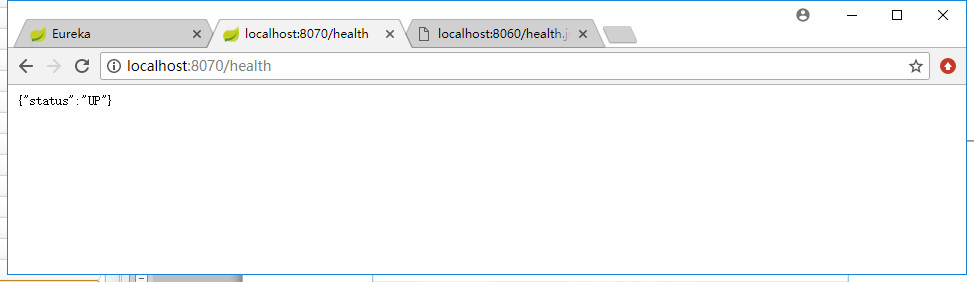
启动SidecarRunApp



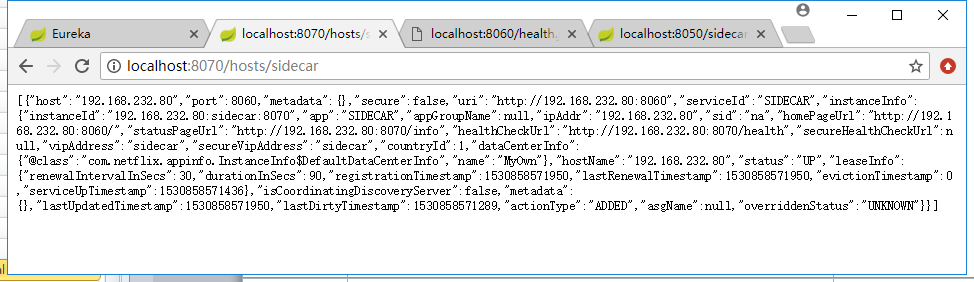
访问sidecar首页：http://localhost:8070





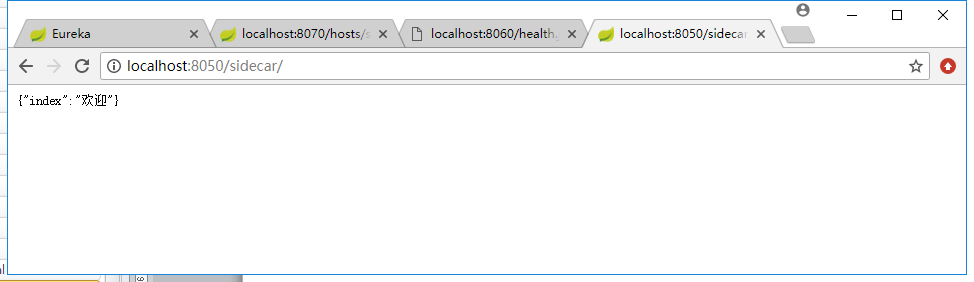


第一次访问为{}空，访问业务链接后，再访问如下内容：

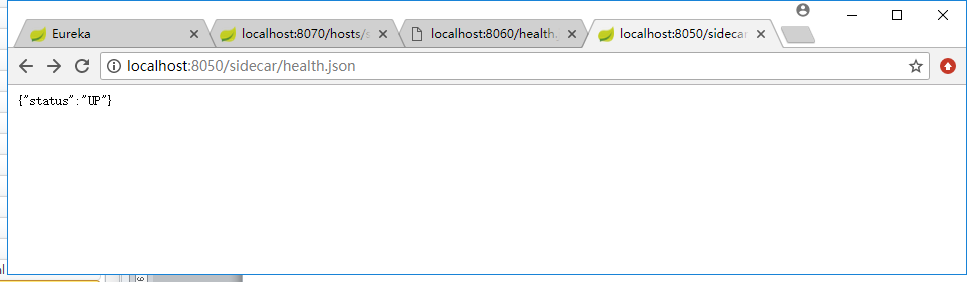


### 通过Zuul访问

访问健康链接： http://localhost:8050/sidecar/health.json



访问首页： http://localhost:8050/sidecar/



## Zuul+Sidecar实现异构系统调用

### 实现微服务的调用

改造Ribbon的Controller， 增加home方法

package cn.tedu.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

@RestController

public class HelloController {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/hello/{name}")

@ResponseBody

public String hello(@PathVariable String name){

String url = "http://provider-user/hello/"+name; //VIP虚拟IP，提供者的application-name:provider-user

return this.restTemplate.getForObject(url, String.class);

}

@GetMapping("/sidecar")

public String home(){

String url = "http://sidecar/"; //通过Zuul调用Nodejs的服务

return this.restTemplate.getForObject(url, String.class);

}

}

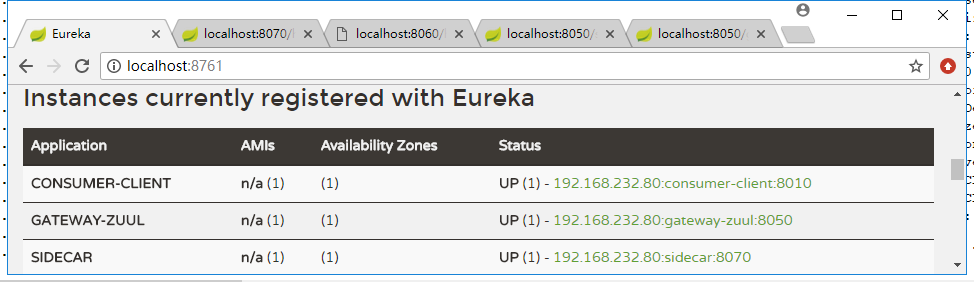
### 测试

启动EurekaServerApp

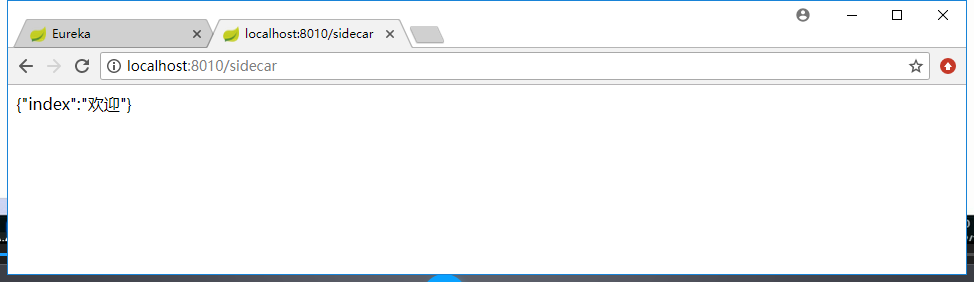
启动ZuulAppRun

启动SidecarRunApp

启动ConsumerRunApp



访问消费者链接：http://localhost:8010/sidecar



### 拓展：健康状态

修改node-server.js。nodejs服务器可以在linux下实现热部署，修改后会自动重新加载。

var http = require('http');

var url = require('url');

var path = require('path');

//创建server

var server = http.createServer(function(req, res){

//获得请求路径

var pathname = url.parse(req.url).pathname;

res.writeHead(200, {'Content-Type':'application/json; charset=utf-8'});

if(pathname === '/'){

res.end(JSON.stringify({ "index":"欢迎" }));

}else if(pathname === '/health.json'){

res.end(JSON.stringify({ "status":"DOWN" }));

}else{

res.end("404");

}

});

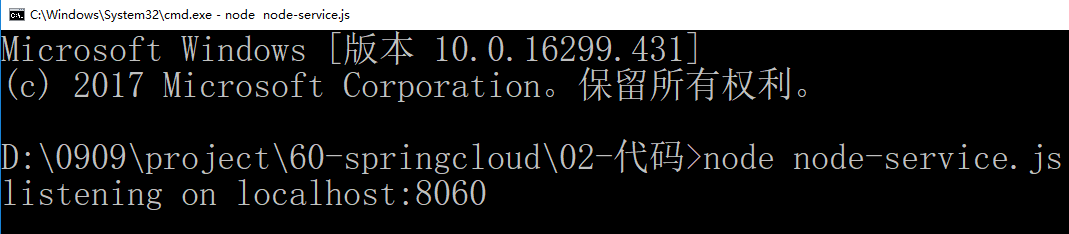
//创建监听，并打印日志

server.listen(8060, function(){

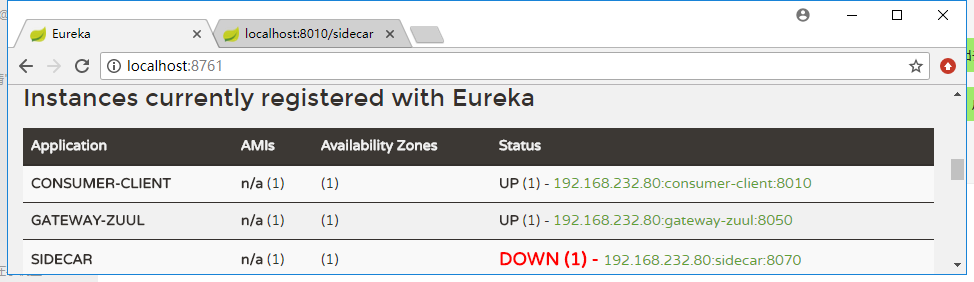
console.log('listening on localhost:8060');

});

重启服务：



访问Eureka：



# 第六天：Git版本控制

## Git版本控制

### 版本控制

版本控制（Version Control System）

* 记录文件的所有历史变化
* 错误恢复到某个历史版本
* 多人协作开发编辑同一个文件

### 主流的版本控制产品

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 名称 | 模型 | 并发模式 | 历史模式 | 变更范围 | 网络协议 | 原子提交性 |
| CVS | Client-server | Merge | Changeset | File | Pserver,ssh | No |
| SVN | Client-server | 3-way merge, recursive merge, octopus merge | Changeset and Snapshot | Tree | custom (svn), custom (svn) over ssh, HTTP and SSL (usingWebDAV) | Yes |
| Git | Distributed | Merge or lock | Snapshot | Tree | custom, custom over ssh, rsync, HTTP/HTTPS, email, bundles | Yes |

\* 版本库模型（Repository model）:描述了多个源码版本库副本间的关系，有C/S（客户端/服务器）模式和分布式两种模式。在客户端/服务器模式下，每一用户通过客户端访问位于服务器的主版本库，每一客户机只需保存它所关注的文件副本，对当前工作副本（working copy）的更改只有在提交到服务器之后，其它用户才能看到对应文件的修改。而在分布式模式下，这些源码版本库副本间是对等的实体，用户的机器除了保存他们的工作副本外，还拥有本地版本库的历史信息。

\* 并发模式（Concurrency model）:描述了当同时对同一工作副本/文件进行更改或编辑时，如何管理这种冲突以避免产生无意义的数据，有排它锁和合并模式。在排它锁模式下，只有发出请求并获得当前文件排它锁的用户才能对对该文件进行更改。而在合并模式下，用户可以随意编辑或更改文件，但可能随时会被通知存在冲突（两个或多个用户同时编辑同一文件），于是版本控制工具或用户需要合并更改以解决这种冲突。因此，几乎所有的分布式版本控制软件采用合并方式解决并发冲突。

\* 历史模式（History model）：描述了如何在版本库中存贮文件的更改信息，有快照和改变集两种模式。在快照模式下，版本库会分别存储更改发生前后的工作副本；而在改变集模式下，版本库除了保存更改发生前的工作副本外，只保存更改发生后的改变信息。

\* 变更范围（Scope of change）：描述了版本编号是针对单个文件还是整个目录树。

\* 网络协议（Network protocols）：描述了多个版本库间进行同步时采用的网络协议。

\* 原子提交性（Atomic commit）：描述了在提交更改时，能否保证所有更改要么全部提交或合并，要么不会发生任何改变。

简而言之，各有优缺点，git要配合hub，可以避免分布式损坏。svn有权限控制，避免全被clone走。git适合纯代码，svn适合综合性文档管理，结合起来就完美。显然最大的不同在于git是分布式的。

### 介绍



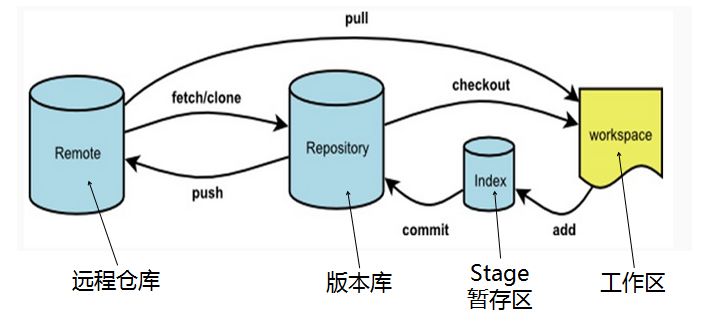
Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。Linus虽然创建了Linux的核心，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

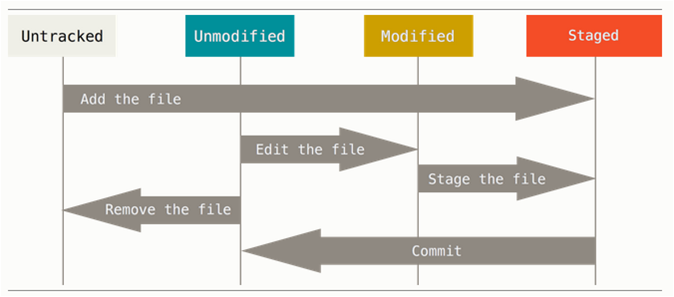
Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

### 组成结构



* 工作区：用来保存项目的元数据和对象数据库的地方。 这是 Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。
* 暂存区：保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。有时候也被称作“索引”，不过一般说法还是叫暂存区域。
* 版本库：也叫本地版本库，之所以说git 快，大部分提交都是对本地仓库而言的，不依赖网络，最后一次会推送的到远程仓库。
* 远程仓库：可以看做是github，它是一个远程仓库，它提供web服务的 供大家方便下载、查看、提交、存储。

### 文件的状态



新建文件状态为untracked，add命令执行后状态变为staged，已存在的文件状态为unmodified，修改文件内容，文件状态变为modified，commit提交，文件状态编程unmodifed。

### 命令速查



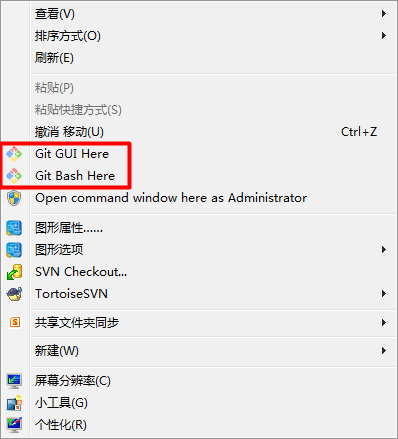
## 本地命令

### 注册账号

第一步：先官网注册账号：https://github.com

第二步：下载安装文件：

Git-2.12.0-64-bit.exe，一路next，安装完桌面右键菜单有下面两项，安装完成。选择Git Bash Here，进入git客户端。

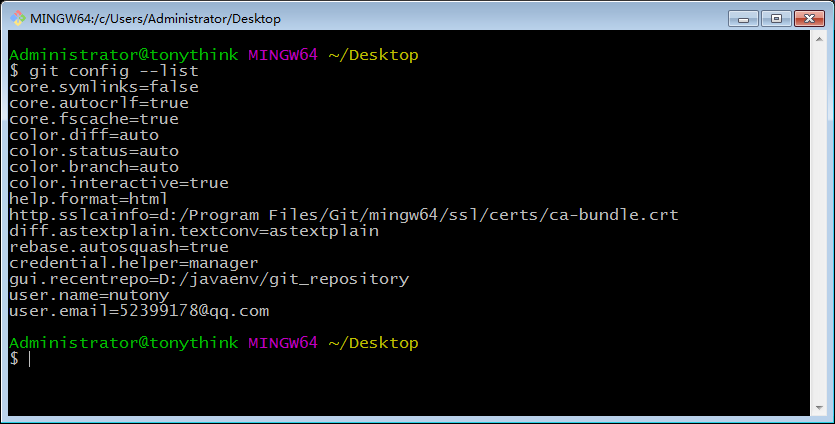


### 配置身份信息



提交文件时，就知道这个文件是谁提交的。出了问题，就知道谁干的！

### 查看配置信息



### 创建本地仓库

D:\javaenv\git\_repository

### 进入仓库

Administrator@tonythink MINGW64 ~/Desktop

$ cd d: #进入d盘

Administrator@tonythink MINGW64 /d

$ cd javaenv

$ mkdir git\_repository #创建目录

$ cd git\_repository #进入目录

### 初始化仓库

$ git init

Administrator@tonythink MINGW64 /d/javaenv/git\_repository (master)

当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。如果你没有看到.git目录，是因为默认这个目录是隐藏的，要显示修改其显示隐藏文件即可，一般无需修改。



### git的工作流程

* 克隆仓库
* 在工作目录中新增、修改、删除文件
* 暂存文件，将文件的快照放入暂存区
* 提交更新，把暂存区的内容提交到Git仓库中

### 提交文件

创建hi.txt

$ git add hi.txt #暂存单个文件

$ git add ./\* #批量暂存当前目录下所有内容

$ git status #查看文件状态

$ git commit -m "test" #提交

[master (root-commit) e522732] test

1 file changed, 1 insertion(+)

create mode 100644 hi.txt

$ git status #查看文件状态

### 查看提交记录

$ git log

commit e522732d94c440fdd750368ce937ac1c40dbd2ed #本次提交唯一标识，对于版本回退非常有用

Author: nutony <52399178@qq.com>

Date: Fri Jun 2 09:39:31 2017 +0800

test

### 比较当前文件和仓库文件

修改hi.txt文件内容，然后和仓库中已经提交的hi.txt比较

**$ git diff hi.txt**

diff --git a/hi.txt b/hi.txt

index 32f95c0..d7d0f3f 100644

--- a/hi.txt

+++ b/hi.txt

@@ -1 +1 @@

-hi

\ No newline at end of file

+<B1>Ƚϵ<B1>ǰ<CE>ļ<FE><BA>Ͳֿ<E2><CE>ļ<FE> #中文乱码

\ No newline at end of file

Administrator@tonythink MINGW64 /d/javaenv/git\_repository (master)

**$ git diff hi.txt**

diff --git a/hi.txt b/hi.txt

index 32f95c0..2b80830 100644

--- a/hi.txt

+++ b/hi.txt

@@ -1 +1 @@

-hi

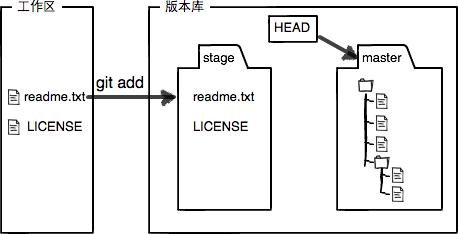
\ No newline at end of file

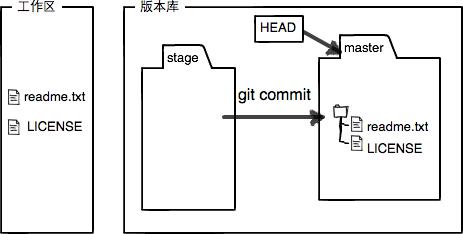
+<U+FEFF>比较当前文件和仓库文件 #文件格式改为UTF-8即可

\ No newline at end of file

### add和commit的区别

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

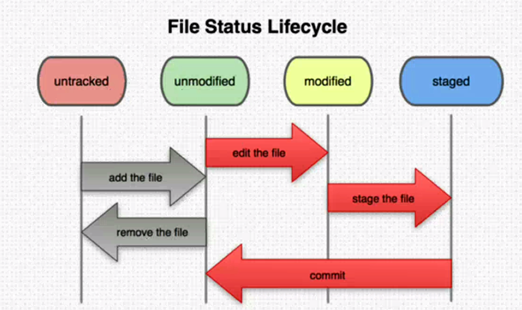




git的文件状态

文件新建完状态为untracked

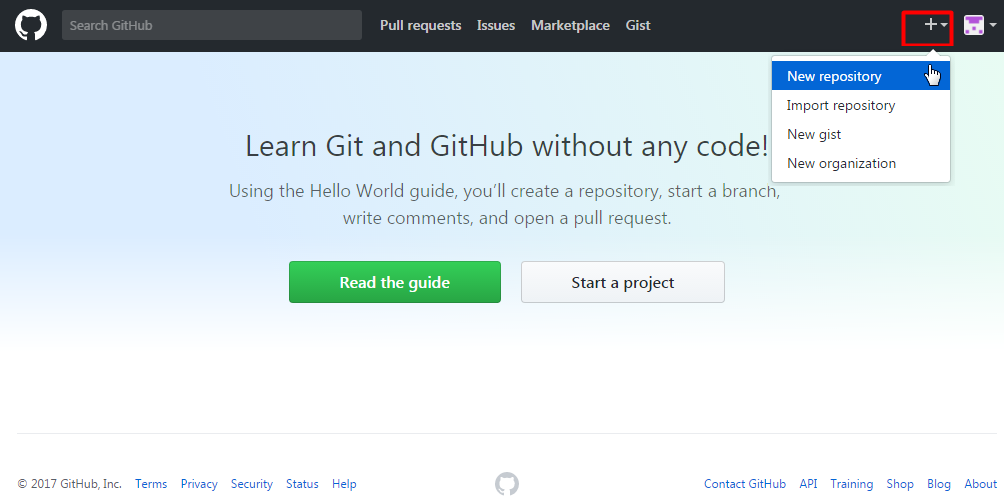
### 文件状态图



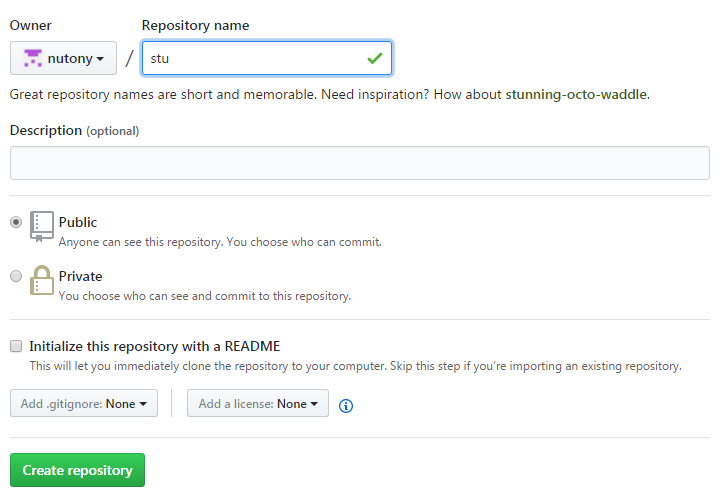
## 远程命令github

### 创建仓库

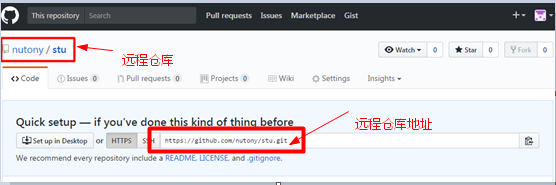
登录：https://github.com 在git上新建一个仓库



仓库名为stu，拥有者为：nutony。



创建成功



### 克隆远程仓库

首次需要把远程仓库克隆到本地仓库（D:\javaenv\git\_repository）

$ git clone https://github.com/nutony/stu.git

本地仓库目录中就多了stu目录

### 关联本地仓库

可以克隆仓库也可以关联本地仓库

运行 git remote add [shortname] [url]

git remote add stu git://github.com/nutony/stu

### 本地分支的更新推送到远程主机

复制文件到stu目录

$ cd stu #进入目录

$ ll #查看目录

total 11320

-rw-r--r-- 1 Administrator 197121 11571275 八月 12 2015 jt.sql

-rw-r--r-- 1 Administrator 197121 13879 一月 16 2016 jt-logo.png

Administrator@tonythink MINGW64 /d/javaenv/git\_repository/stu (master)

$ git add . #添加当前目录下文件

Administrator@tonythink MINGW64 /d/javaenv/git\_repository/stu (master)

$ ll

total 11320

-rw-r--r-- 1 Administrator 197121 11571275 八月 12 2015 jt.sql

-rw-r--r-- 1 Administrator 197121 13879 一月 16 2016 jt-logo.png

Administrator@tonythink MINGW64 /d/javaenv/git\_repository/stu (master)

$ git commit -m "jttest" #提交，设置备注

[master (root-commit) 4364109] jttest

2 files changed, 301 insertions(+)

create mode 100644 jt-logo.png

create mode 100644 jt.sql

Administrator@tonythink MINGW64 /d/javaenv/git\_repository/stu (master)

$ git push origin master #推送到远程仓库master默认分支，第一次需要登录，比较慢，弹出登录框输入用户名和密码。

执行结果：

Counting objects: 4, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 2.77 MiB | 31.00 KiB/s, done.

Total 4 (delta 0), reused 0 (delta 0)

To https://github.com/nutony/stu.git

\* [new branch] master -> master

Administrator@tonythink MINGW64 /d/javaenv/git\_repository/stu (master)

$

### 删除文件

$ rm \*.jpg #删除所有jpg结尾，删除本地文件

$ ll

$ git add .

$ git commit -m 'rm'

$ git push origin master

## 代码冲突

### 查看历史版本

$ git log

$ git log --stat #记录我们可以看到每一次commit，都有哪些文件发生了改变

$ git show 8ef96 #查询某次具体的修改

### 回滚

发现提交的内容错了，可以进行回滚操作。

$ vi a.txt #修改文件内容

$ git add a.txt

$ git commit –m 'a.txt'

$ git log #查看日志

commit 8ef960a0bfa137a0fadd7494823d5d1c101c427d

Author: nutony <52399178@qq.com>

Date: Sat Jun 3 09:13:32 2017 +0800

a.txt

commit 315e77e5cceb486a9bf63911cb8d3549a19b8255

Author: nutony <52399178@qq.com>

Date: Sat Jun 3 09:07:43 2017 +0800

Delete 31.jpg

$ git revert 8ef960a #日志的commitId的前几位即可

### 远程仓库覆盖本地

$ vi a.txt #创建a.txt，设置内容为haha 1

$ git add . #暂存

$ git commit -m '1' #提交

$ git pull origin master #推送，此时查看github，a.txt存在

$ vi a.txt #编辑本地文件a.txt，修改内容haha 2

$ git fetch --all #只是下载代码到本地，不进行合并操作

$ git reset --hard origin/master #把HEAD指向最新的下载版本

$ ll

$ cat a.txt #可以看到文件还是为1，被远程覆盖

### 解决冲突绝招

如果冲突，自己修改的文件自己知道，则把修改的文件复制到其它目录，然后强制用远程覆盖本地文件，然后把修改的文件考回来覆盖。这样的好处是多人修改后，多个文件，修改起来太复杂。这样的方式非常方便高效。

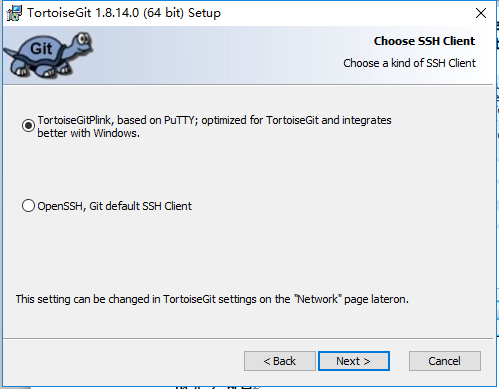
## TortoiseGit

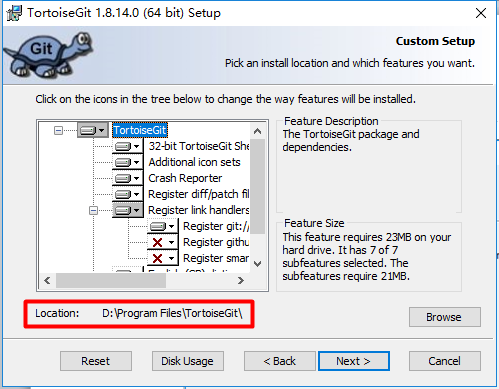
### 安装

TortoiseGit-1.8.14.0\_64bit.1436148947.msi

安装完成会自动获取登录的配置信息，同时在增加操作到右键菜单上。

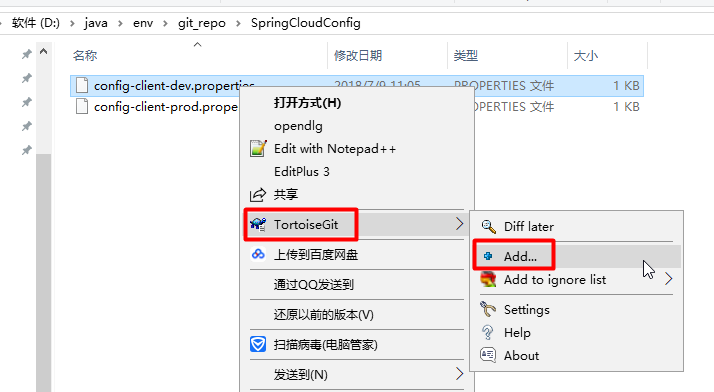


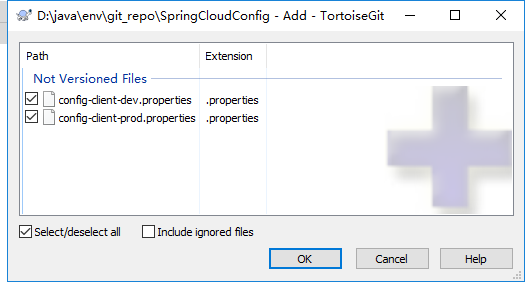




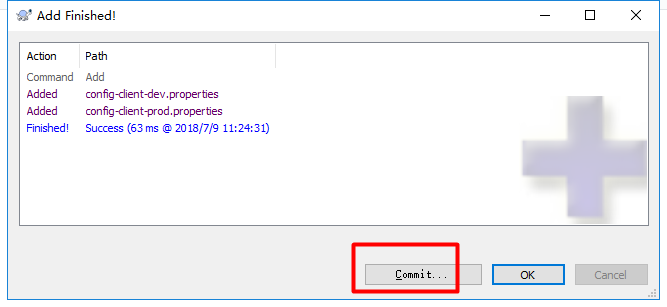
### 添加文件到本地仓库

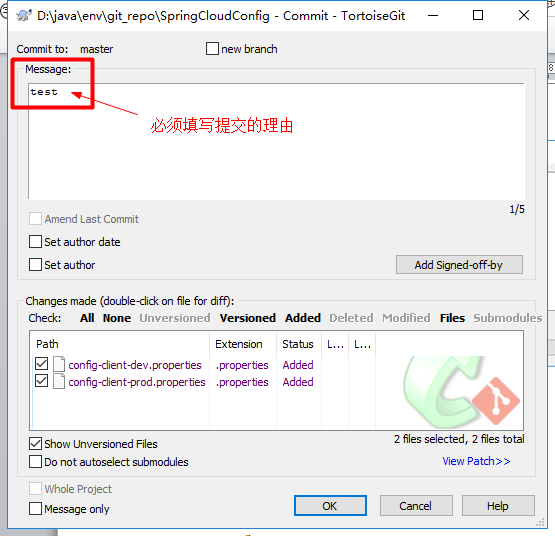
先创建好文件，然后右键选择add命令。可以批量选择。

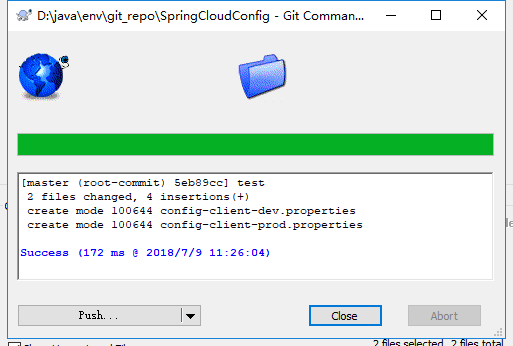




### 提交







### 查询

