# 订单业务

## 定时任务实现

### 需求说明

例子:当订单在指定的时间内(30分钟),没有完成支付.则将订单关闭.

问题:如何监控???

对比条件: 当前时间 – 创建时间 > 30分钟 1未支付 6交易关闭

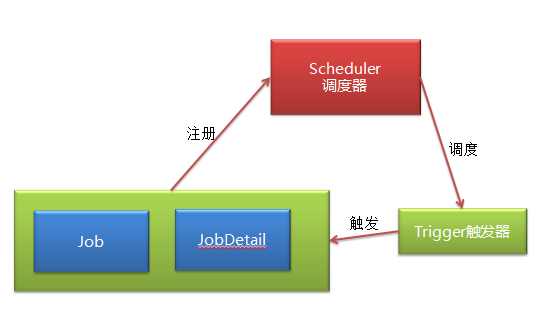
工具:timer java原生提供的 功能单一

## Quartz

### 介绍

Quartz是OpenSymphony开源组织在Job scheduling领域又一个开源项目，它可以与J2EE与J2SE应用程序相结合也可以单独使用。Quartz可以用来创建简单或为运行十个，百个，甚至是好几万个Jobs这样复杂的程序。Jobs可以做成标准的Java组件或 EJBs。Quartz的最新版本为Quartz 2.3.0.

### Quartz调用原理



1. 将任务注册到调度器中.包含任务执行具体的类路径和任务执行的时间
2. 调度器管理全部的任务.内部有时钟,监控程序的执行.
3. 当程序到了执行的时间 ,则调度器则会通过触发器去执行任务.

### 添加jar包

<!--添加Quartz的支持 -->

<dependency>

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

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

</dependency>

### 编辑Quartz配置类

@Configuration

**public** **class** OrderQuartzConfig {

//定义任务详情

@Bean

**public** JobDetail orderjobDetail() {

//指定job的名称和持久化保存任务

**return** JobBuilder

.*newJob*(OrderQuartz.**class**) //任务处理类

.withIdentity("orderQuartz") //定义任务名称

.storeDurably()

.build();

}

//定义触发器

@Bean

**public** Trigger orderTrigger() {

/\*SimpleScheduleBuilder builder = SimpleScheduleBuilder.simpleSchedule()

.withIntervalInMinutes(1) //定义时间周期

.repeatForever();\*/

CronScheduleBuilder scheduleBuilder

= CronScheduleBuilder.*cronSchedule*("0 0/1 \* \* \* ?");

**return** TriggerBuilder

.*newTrigger*()

.forJob(orderjobDetail())

.withIdentity("orderQuartz")

.withSchedule(scheduleBuilder).build();

}

}

### 编辑业务处理类

//准备订单定时任务

@Component

**public** **class** OrderQuartz **extends** QuartzJobBean{

@Autowired

**private** OrderService orderService;

//将超时订单修改状态

@Override

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

orderService.updateOrderStatus();

System.***out***.println("定时任务执行");

}

}

### 编辑业务Service

//定时关闭超时任务

//当前时间 – 创建时间 > 30分钟 1未支付

@Override

**public** **void** updateOrderStatus() {

//获取是当前时间

Calendar calendar = Calendar.*getInstance*();

calendar.add(Calendar.***MINUTE***, -30);

Date date = calendar.getTime();

Order tempOrder = **new** Order();

tempOrder.setStatus(6); //表示交易关闭

tempOrder.setUpdated(**new** Date());

UpdateWrapper<Order> updateWrapper =

**new** UpdateWrapper<>();

updateWrapper.eq("status",1)

.lt("created",date);

orderMapper.update(tempOrder, updateWrapper);

}

# 微服务

## 传统项目问题

### 问题说明

1. 单体项目中所有的默认都写到一个tomcat服务器中,如果有某个节点出现问题,则直接影响整个的服务运行.
2. 分布式系统中,虽然实现了架构的松耦合.但是在不引入插件时无法实现高可用.

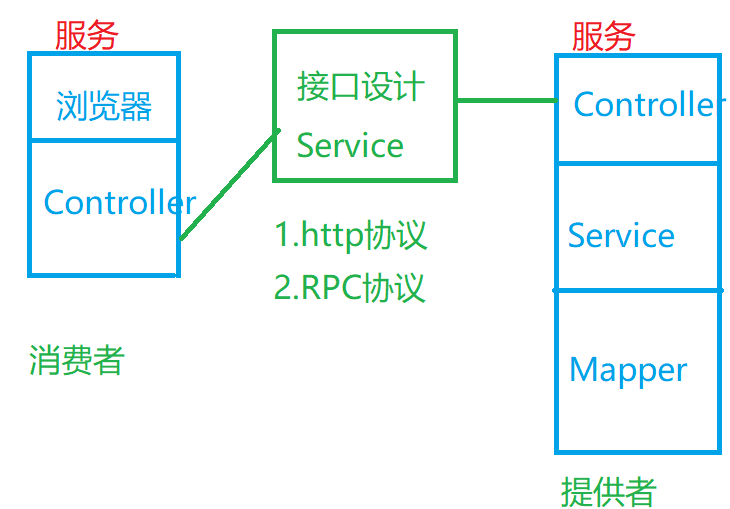
### 微服务概述

总结:微服务的前提条件是分布式系统架构.把架构中采用独立设计(小型化)的.并且能够独立运行的,并且内部能够实现自动化的故障迁移.

### 微服务程序编程方式---SOA思想

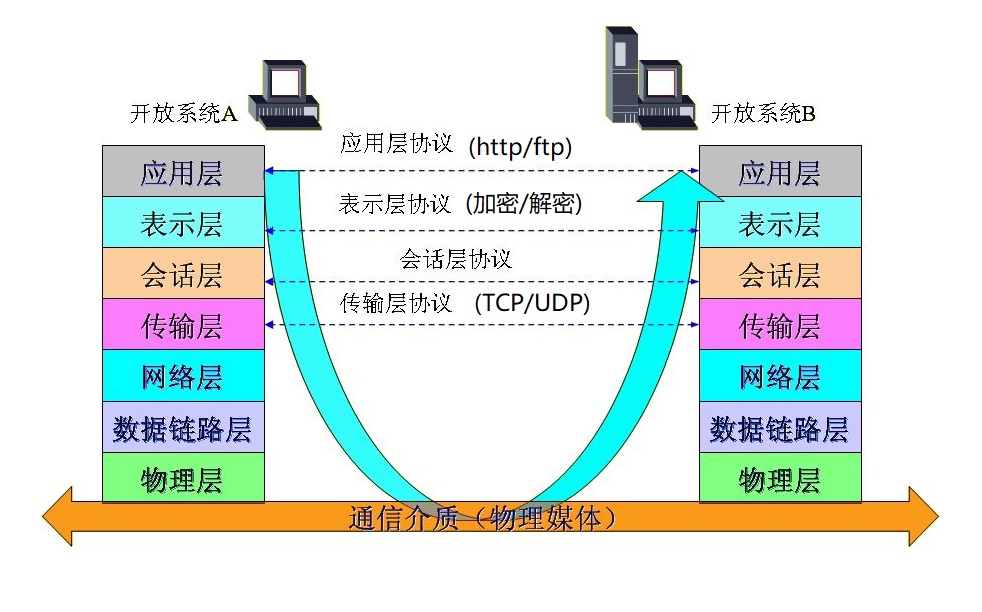
面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（称为服务）进行拆分，并通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

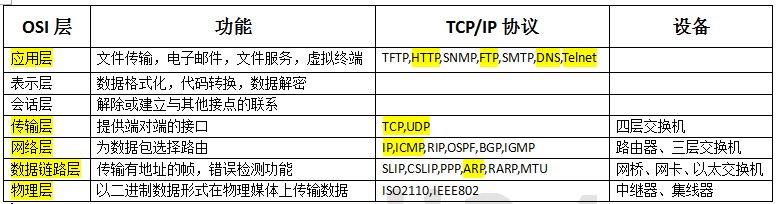
(写代码的方式)



服务和服务中间通过接口的方式调用,通过HTTP协议/RPC协议调用.调用接口的叫做消费者,实现接口的叫做提供者.

### 网络模式OSI模型





HTTP和RPC区别:

1. 安全性

http协议更加的安全,经过表示层加密.

三次握手:目的保证数据传输的有效性.

http协议必须采用3次握手.

RPC协议:直接将数据发送,数据能否正确的传输,不能保证.

RPC内部也将数据进行加密处理.

1. 传输速度

RPC更快,它是第5层的协议.

RPC协议采用的是长链接的形式.

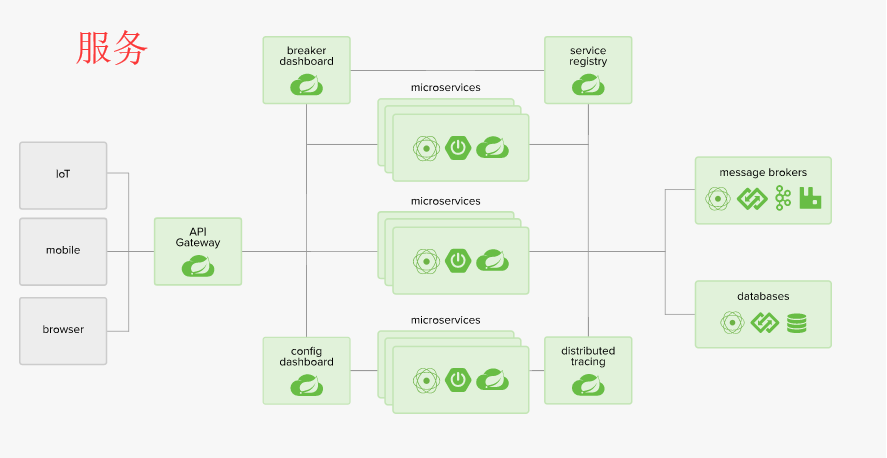
## SpringCloud

### springCloud介绍

Spring Cloud直接基于Spring Boot的创新企业Java方法，通过实施经过验证的模式，为您的微服务带来弹性，可靠性和协调，从而**简化了分布式微服务式架构**。

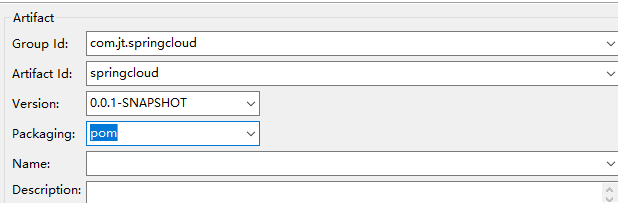
构建分布式系统不需要复杂且容易出错。Spring Cloud为最常见的分布式系统模式提供了一种简单易用的编程模型，可帮助开发人员构建弹性，可靠和协调的应用程序。Spring Cloud构建于Spring Boot之上，使开发人员可以轻松入门并快速提高工作效率。

### SpringCloud的组件模型



## 微服务入门案例

### 定义父级项目springcloud



### 编辑pom.xml文件

<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.jt.springcloud</groupId>

<artifactId>springcloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<!--导入springBoot依赖包 -->

<parent>

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

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

<version>2.1.1.RELEASE</version>

<relativePath />

</parent>

<properties>

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

</properties>

<!--依赖管理，用于管理spring-cloud的依赖 -->

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Finchley.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<dependencies>

<!--导入springCloudjar包 -->

<dependency>

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

<artifactId>spring-cloud-stream</artifactId>

</dependency>

<!--引入springBoot jar包 -->

<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>org.springframework.boot</groupId>

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

 </dependency> -->

<!--支持热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

<version>1.2.8.RELEASE</version>

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<!--整合redis -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

</dependency>

<!--导入pojo插件 -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

<!--使用druid整合mysql数据库 -->

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

<!--阿里数据源 -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.12</version>

</dependency>

<!--mybatis-plus配置 -->

<dependency>

<groupId>com.baomidou</groupId>

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

<version>3.0.6</version>

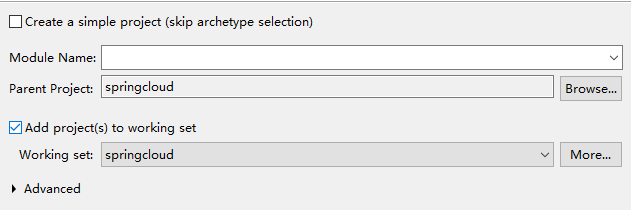
</dependency>

</dependencies>

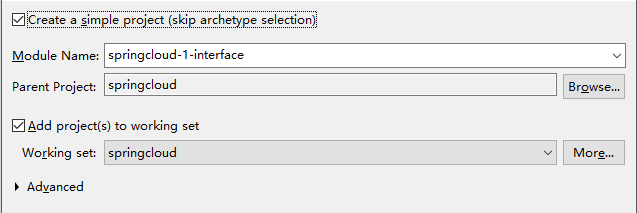
</project>

### 编辑接口项目

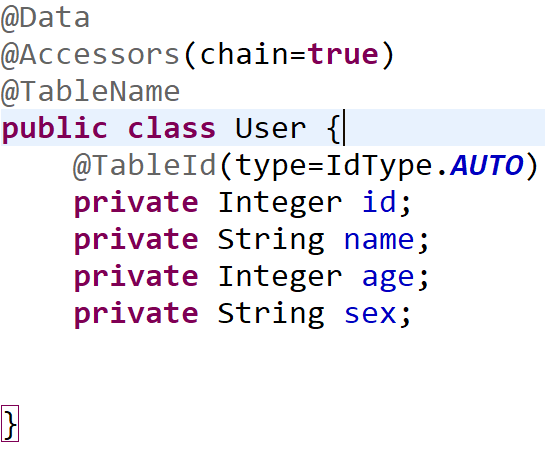
1. 创建项目



1. 创建项目2



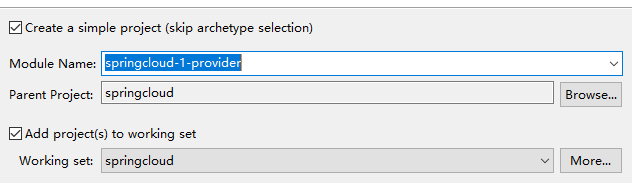
### 编辑POJO文件



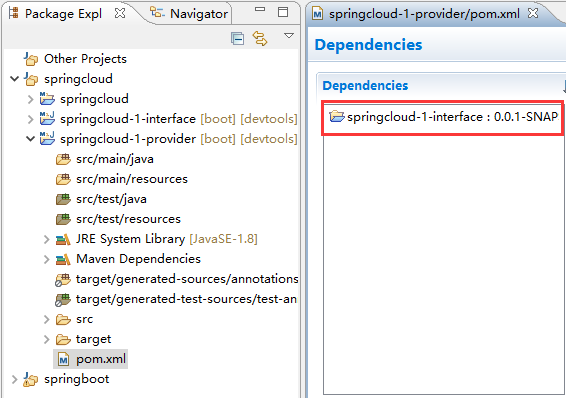
说明:如果编辑完成后,将接口文件打包处理.

## 创建服务的提供者

### 创建项目



### 依赖接口

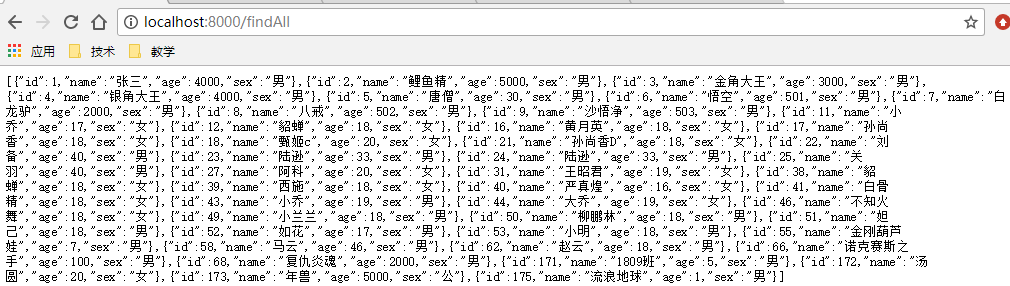


### 实现user信息查询

编辑:说明根据springBoot配置,实现数据库查询,并且JSON数据页面展现.

查询方法:localhost:8000/findAll

根据之前的资料自行完成.



## 新建服务消费者

### 编辑配置类

//一般使用springCloud时,如果是get请求采用Rest形式.

//如果请求是post提交时,后台直接接收JSON数据即可.

@Configuration

**public** **class** RestConfig {

@Bean

**public** RestTemplate getBean() {

**return** **new** RestTemplate();

}

}

说明:该类代替就框架使用的httpClient.

### 编辑消费者Controller

@RestController

**public** **class** UserController {

@Autowired

**private** RestTemplate restTemplate;

**private** String url = "http://localhost:8000";

/\*\*

\* 消费者

\* 参数介绍

\* url:消费者访问提供者的url地址

\* http://localhost:8000/findAll

\* responseType:返回值响应的数据类型

\* **@return**

\*/

@RequestMapping("/consumer/findAll")

**public** List<User> findUser(){

String getUrl = url+"/findAll";

List<User> userList =

restTemplate.getForObject(getUrl,List.**class**);

**return** userList;

}

}

### 编辑主启动类

//排除数据源之外加载

@SpringBootApplication(exclude=DataSourceAutoConfiguration.**class**)

**public** **class** SpringBoot\_consumer\_8020 {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBoot\_consumer\_8020.**class**, args);

}

}

### 关于数据源报错的说明

说明:如果引入数据源包.在包内有对应的spring的注解,所以在启动项目时,Spring容器要求必须实例化数据源.(驱动/用户名/密码….)

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:

If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.

If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

1. 添加数据源相关配置
2. 告诉spring容器,启动时暂时不要加载数据源.

### 测试效果



## 数据库新增

### 编辑提供者Controller

/\*

\* 服务提供者 用户新增

\* @ResponseBody 将对象转化为JSON

\* @RequestBody 将json串转化为对象

\*/

@RequestMapping("/saveUser")

**public** String saveUser(@RequestBody User user) {

**return** userService.saveUser(user);

}

### 编辑提供者Service

/\*

\* 服务提供者 用户新增

\* @ResponseBody 将对象转化为JSON

\* @RequestBody 将json串转化为对象

\*/

@RequestMapping("/saveUser")

**public** String saveUser(@RequestBody User user) {

**return** userService.saveUser(user);

}

### 编辑消费者Controller

/\*\*

\* 编辑消费者Controller

\* **@param** user

\* **@return**

\* request:请求发送的参数

\* responseType:返回值类型

\*

\*/

@RequestMapping("/consumer/saveUser/{name}/{age}/{sex}")

**public** String saveUser(User user) {

String postUrl = url + "/saveUser";

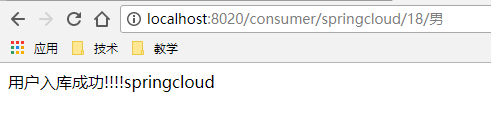
String result =

restTemplate.postForObject(postUrl,user,String.**class**);

**return** result;

}

### 页面访问效果



## 删除用户

### 编辑消费者Controller

//实现用户删除 id=176数据

@RequestMapping("/consumer/delUser/{id}")

**public** String delUser(@PathVariable Integer id) {

String delUrl = url+"/delUser/"+id;

**return** restTemplate.getForObject(delUrl, String.**class**);

}

### 编辑提供者Controller

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

**public** String delUser(@PathVariable Integer id) {

**return** userService.delUser(id);

}

### 编辑提供者Service

@Override

**public** String delUser(Integer id) {

userMapper.deleteById(id);

**return** "用户删除成功!!!"+id;

}