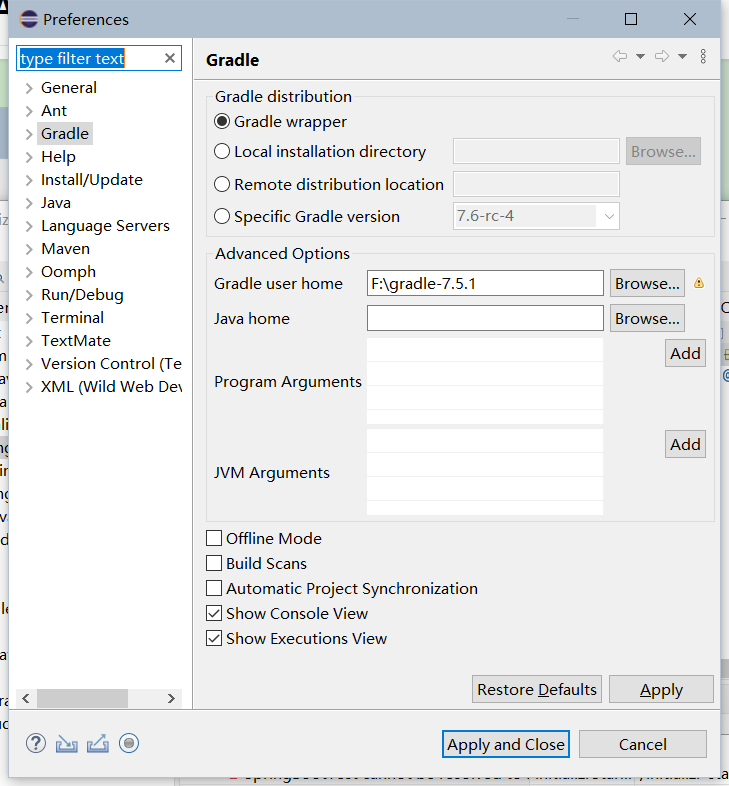
# Proj

<https://github.com/Rox0617/Final-project>

# 简介、环境搭建、入门项目

## Eclipse Gradle开发环境配置



## Spring Initializr开源项目：

.gitignore：版本管理的忽略文件

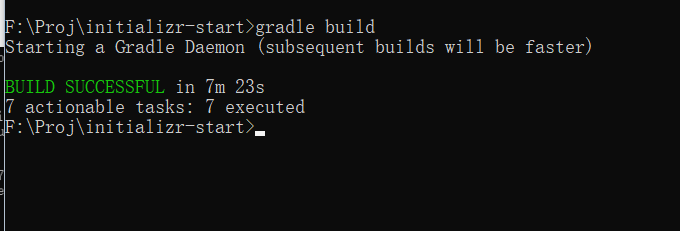
build.gradle：gradle项目的配置文件（核心）

gradlew：在Linux下的执行脚本

gradlew.bat：在Windows下的执行脚本



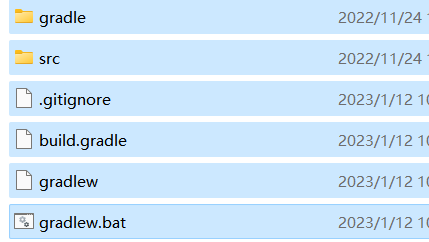
对项目进行编译 Gradle build - 编译成功

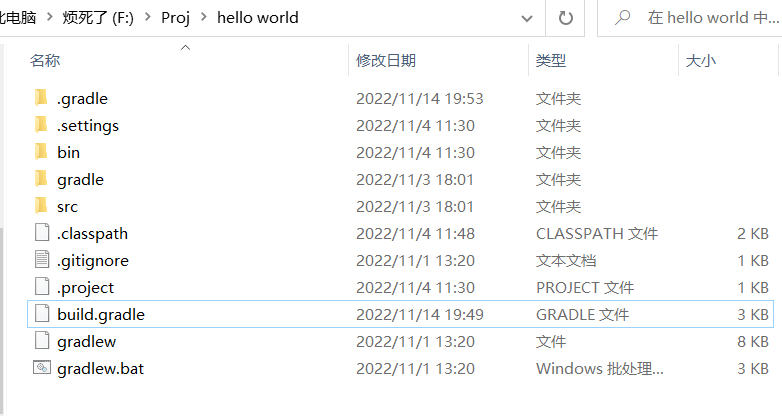


## Hello World

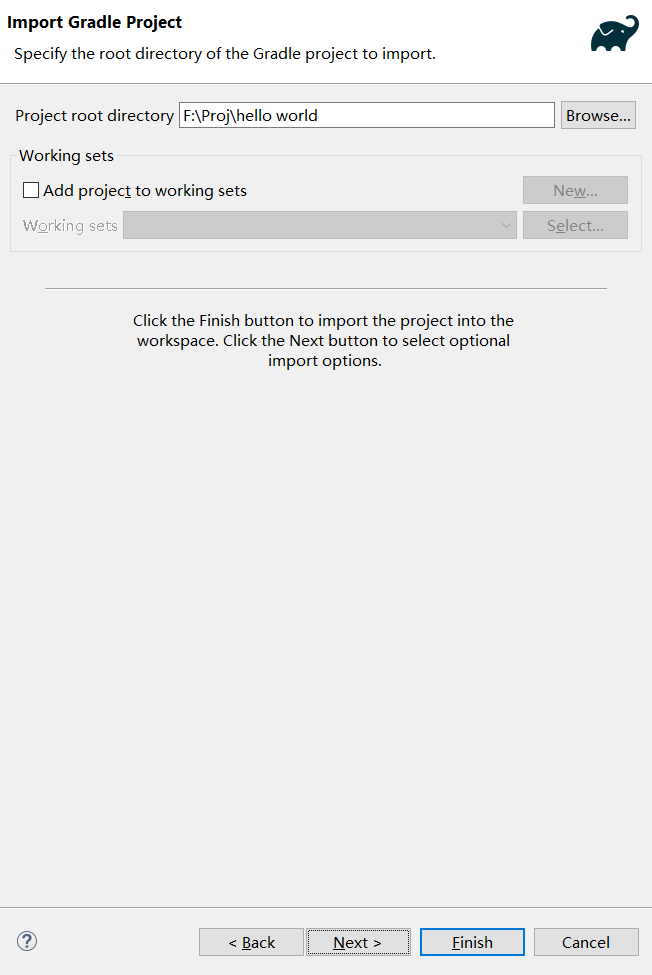
### 创建项目

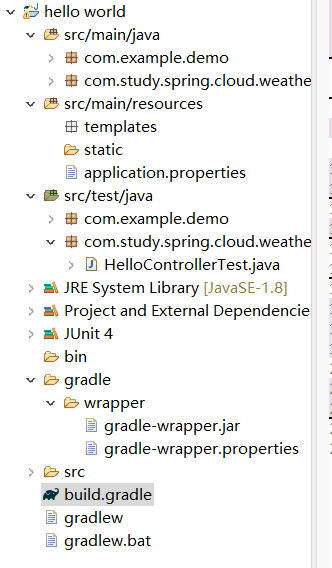
复制的Spring - initializr - start项目，将副本重命名为hello-world：





### Improt Gradle Project





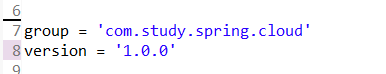
### 开始项目：

### 修改源码：

Application.java一定要处于整个工程的根目录下，这样它才能根据配置去扫描子节点下的Spring的Bean。

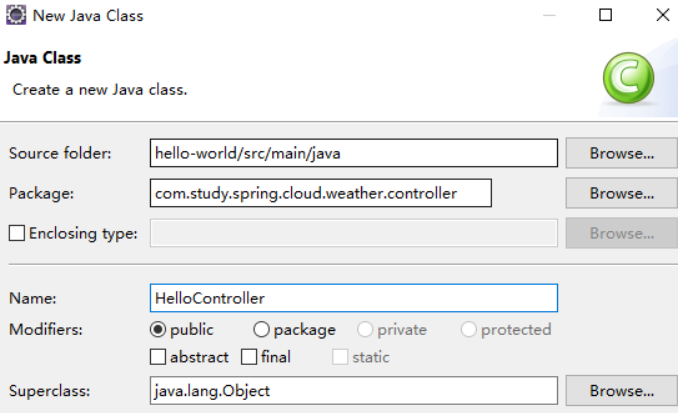
修改build.gradle配置：Application.java一定要处于整个工程的根目录下，这样它才能根据配置去扫描子节点下的Spring的Bean。

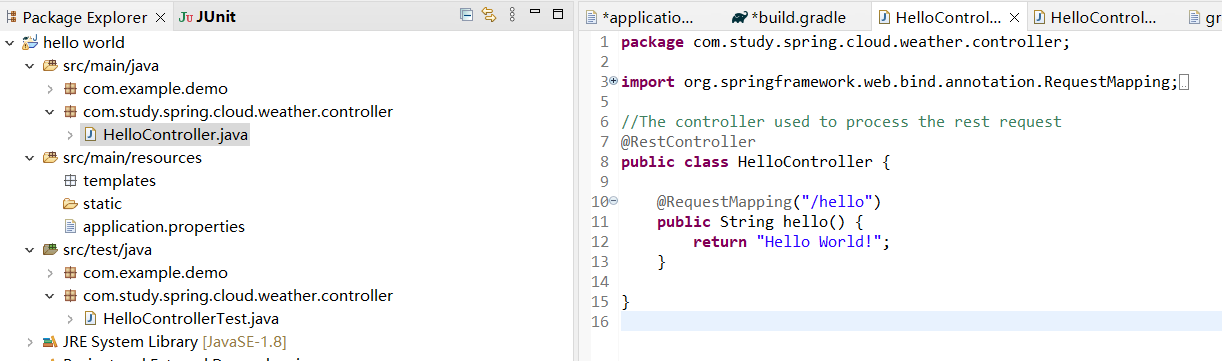
修改build.gradle配置：



### 开发controller

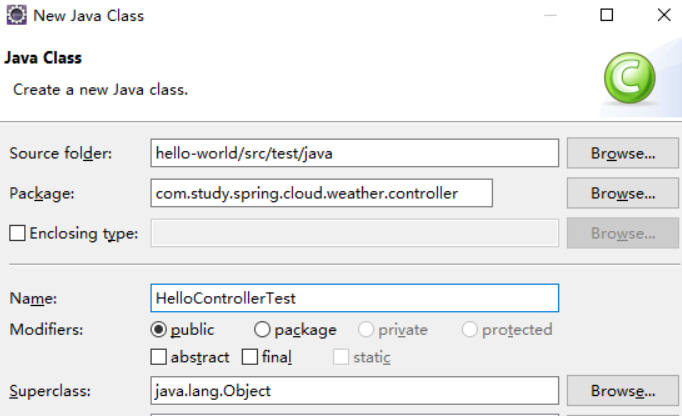
新建一个controller的包：

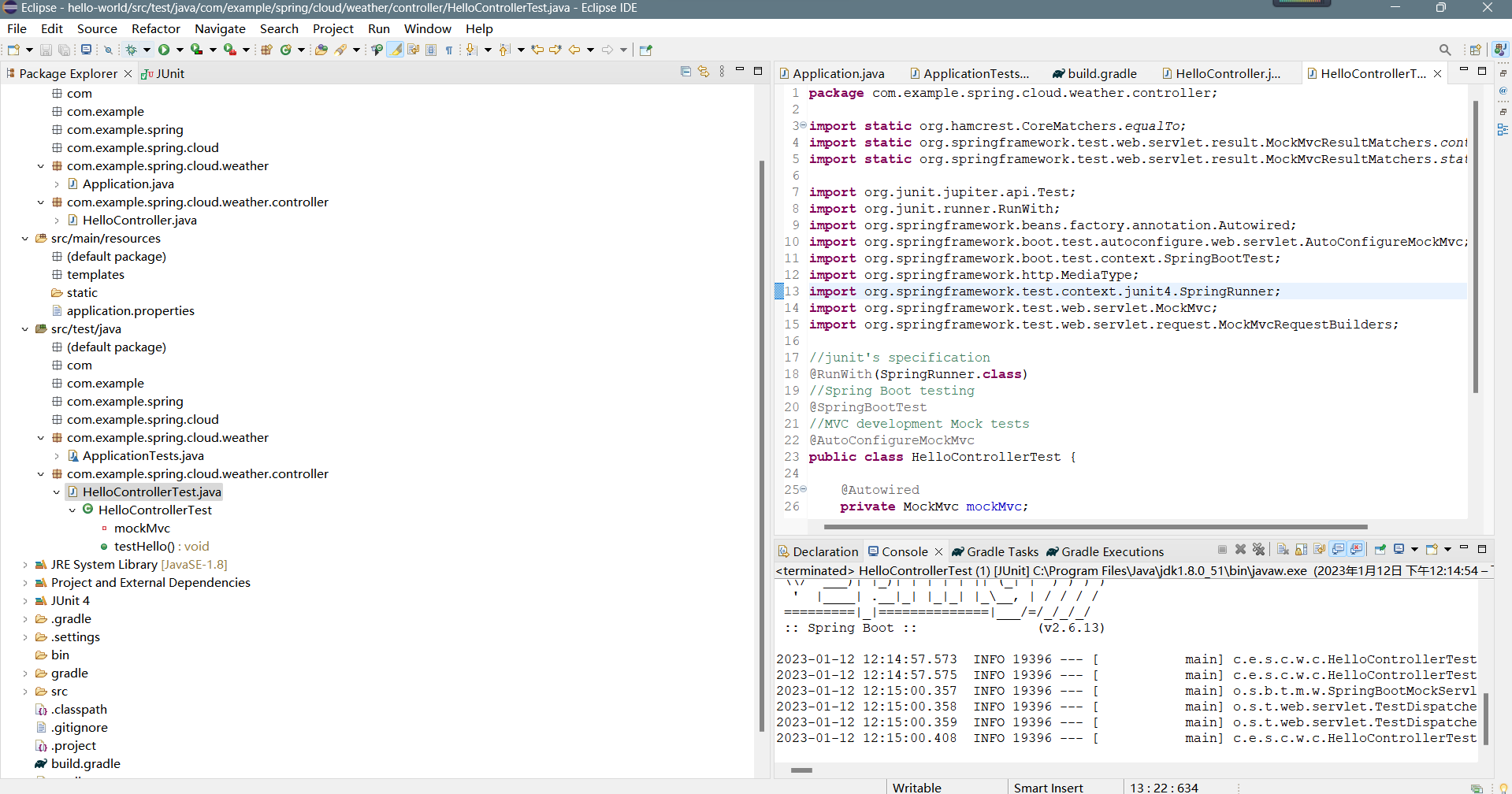




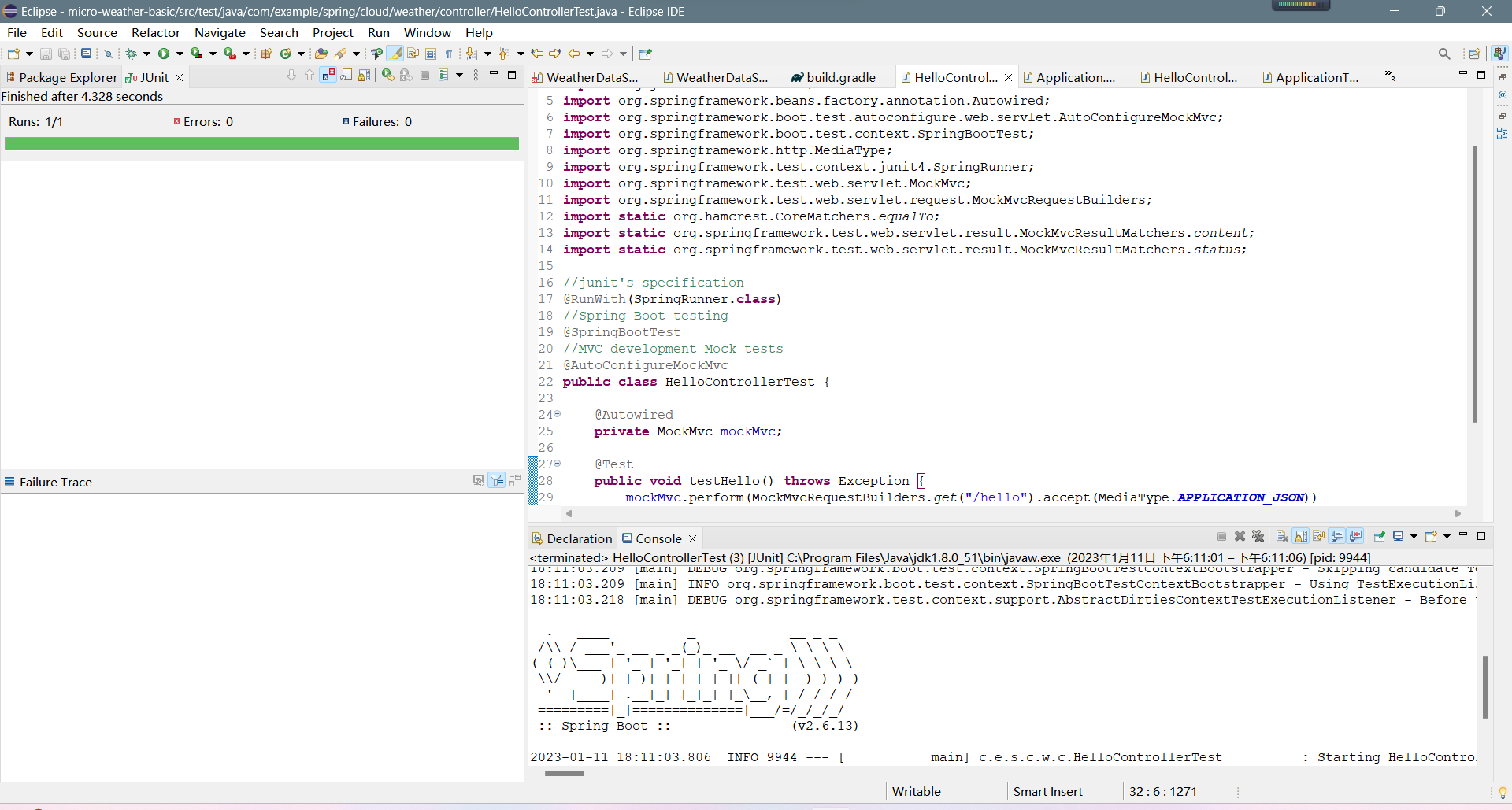
### 编写测试用例

### 新建一个controller的包：



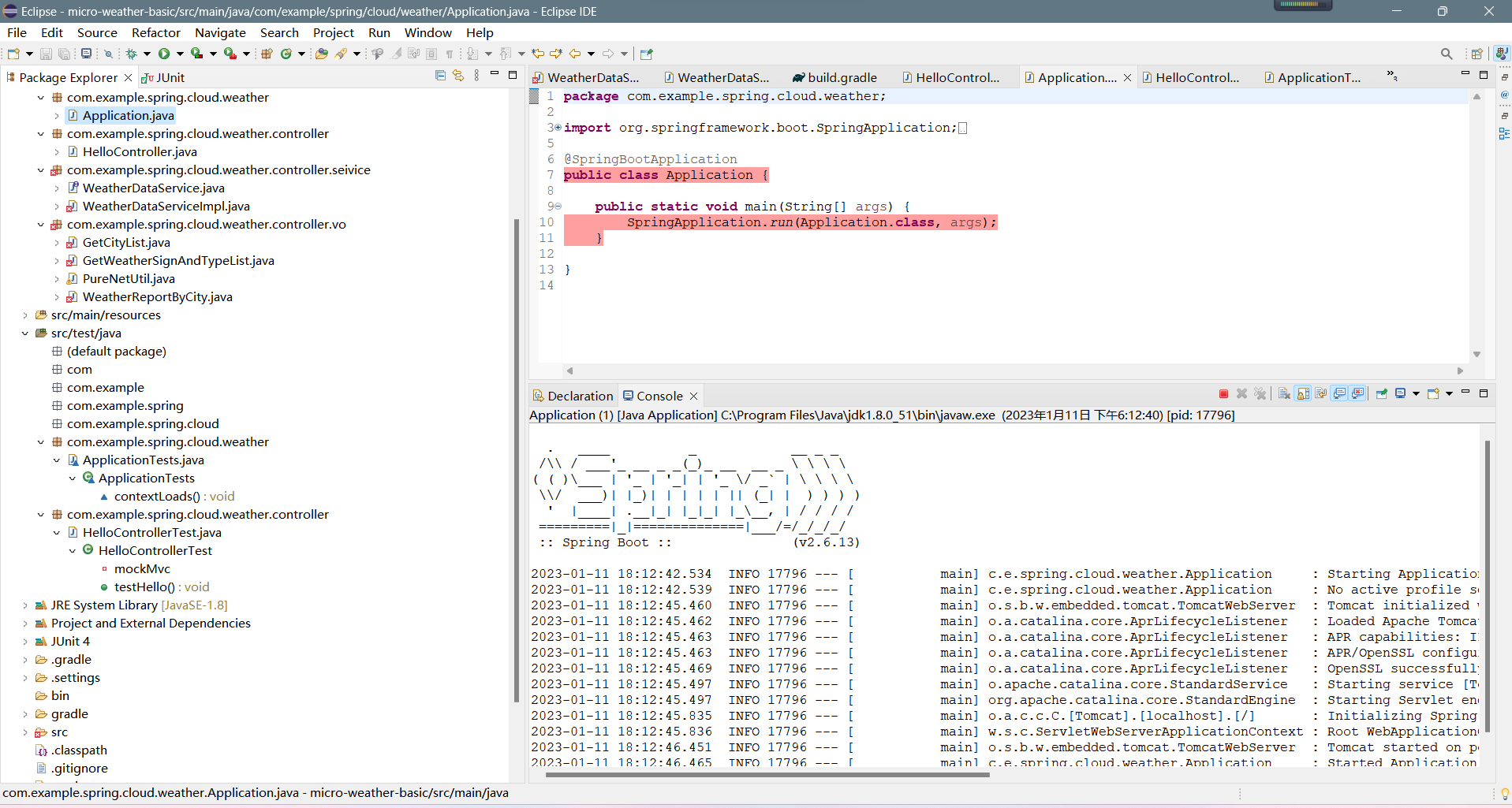


### 运行测试类结果展示：

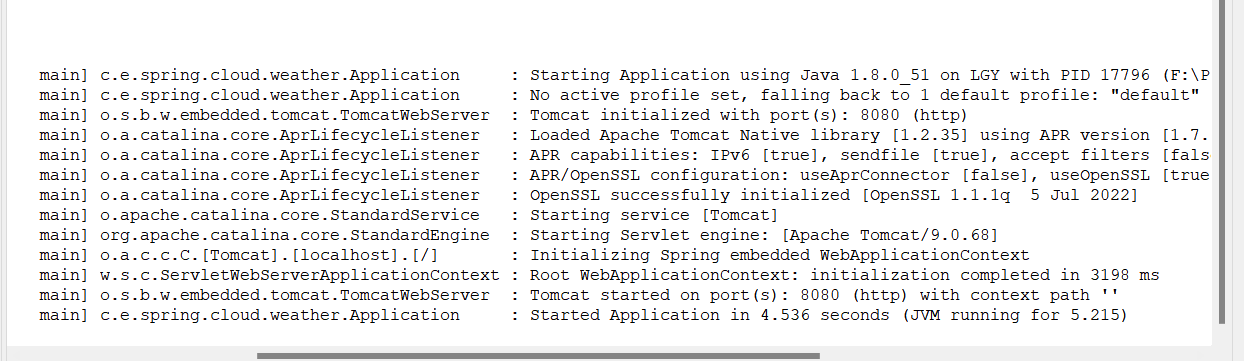


### 运行Spring Boot

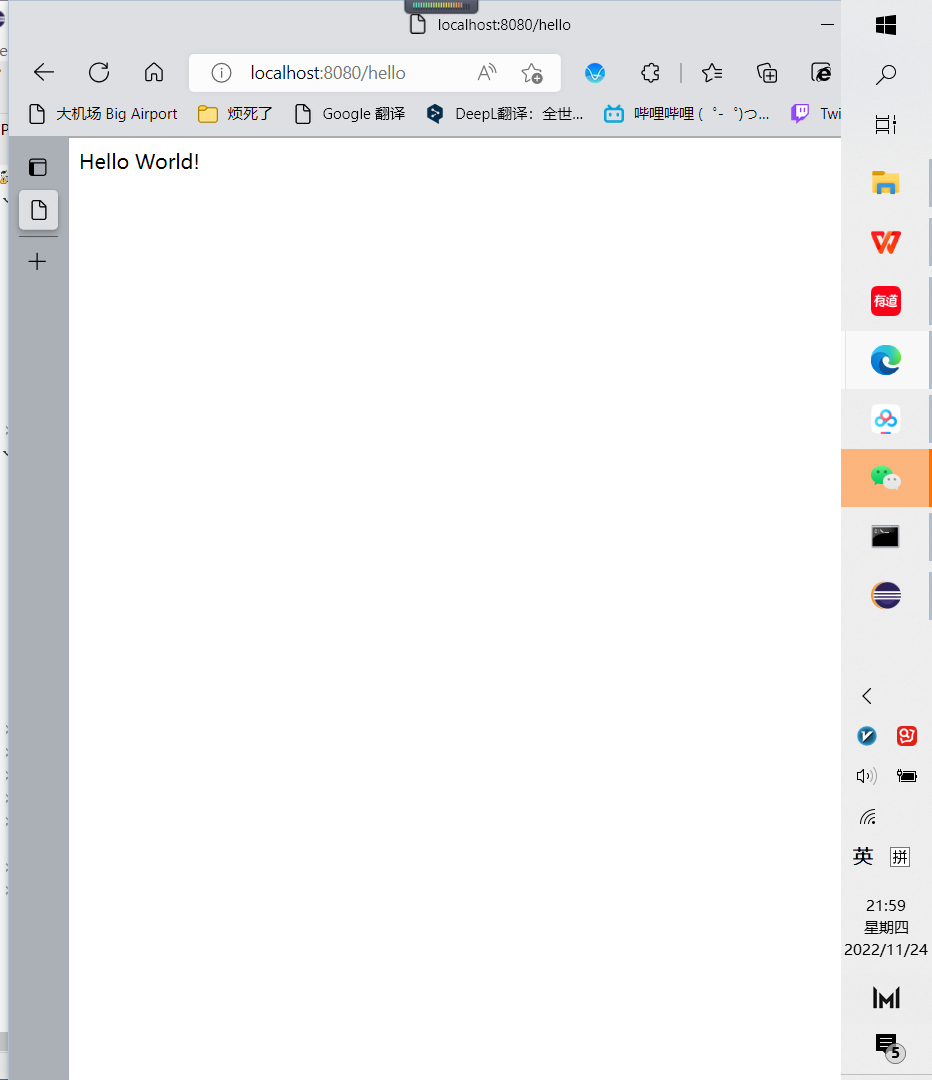
右键Run as Java Application



Tomcat在8080端口：

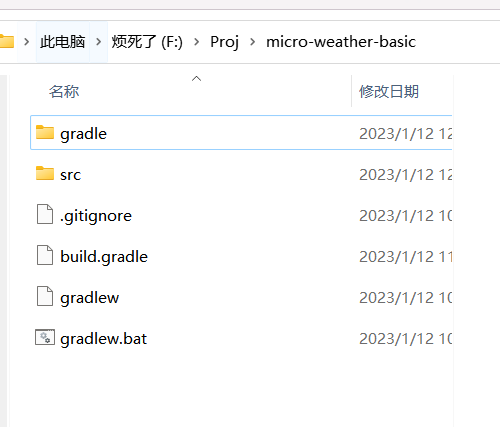


[localhost:8080/hello](http://localhost:8080/hello)



# 一个简单的天气预报系统

## 创建项目micro-weather-basic



### 修改源码

配置文件

Spring Boot的默认配置文件为properties，根据我们引入的不同Starter模块，可以在这里定义容器端口号、数据库连接信息、日志级别等各种配置信息。此外我还发现，其配置文件还支持使用YAML文件。在经过对比后明显发现YAML配置信息利用阶梯化缩进的方式，其结构更为清晰易读，同时配置内容的字符量也得到显著减少。

在一些特殊情况下，我们希望有些参数每次被加载的时候不是一个固定的值，比如密钥、服务端口等。在Spring Boot的属性配置文件中，可以通过使用${random}配置来产生随机的int值、long值或者string字符串，这样我们就可以容易地通过配置随机生成属性，而不是在程序中通过编码来实现这些逻辑。该配置方式可以设置应用端口等场景，以避免在本地调试时出现端口冲突的麻烦。

application-dev.properties：开发环境

application-test.properties：测试环境

application-prod.properties：生产环境

Spring Boot使用了下面这种较为特别的属性加载顺序：

1.在命令行中传入的参数。

2.SPRING\_APPLICATION\_JSON中的属性。SPRING\_APPLICATION\_JSON是以JSON格式配置在系统环境变量中的内容。

3.java:comp/env中的JNDI属性。

4.Java的系统属性，可以通过System.getProperties()获得的内容。

5.操作系统的环境变量。

6.通过random.\*配置的随机属性。

7.位于当前应用jar包之外，针对不同{profile}环境的配置文件内容，例如application-{profile}.properties或是YAML定义的配置文件。

8.位于当前应用jar包之内，针对不同{profile}环境的配置文件内容，例如application-{profile}.properties或是YAML定义的配置文件。

9.位于当前应用jar包之外的application.properties和YAML配置内容。

10.位于当前应用jar包之内的application.properties和YAML配置内容。

11.在@Configuration注解修改的类中，通过@PropertySource注解定义的属性。

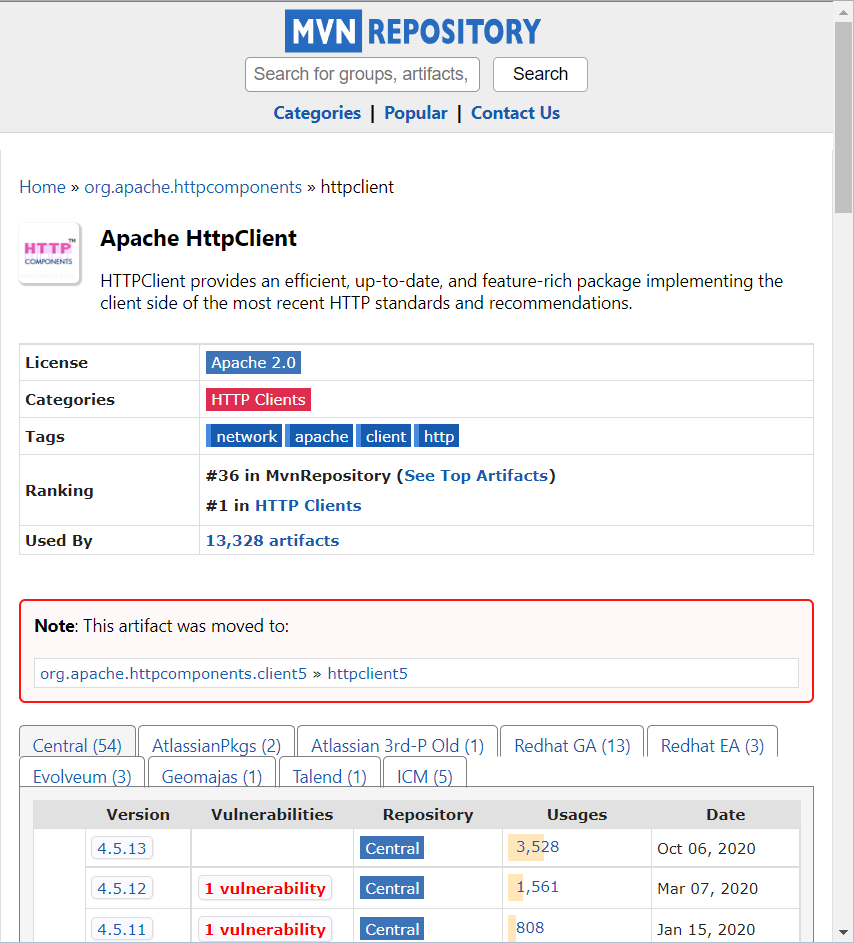
12应用默认属性，使用SpringApplication.setDefaultProperties定义的内容.

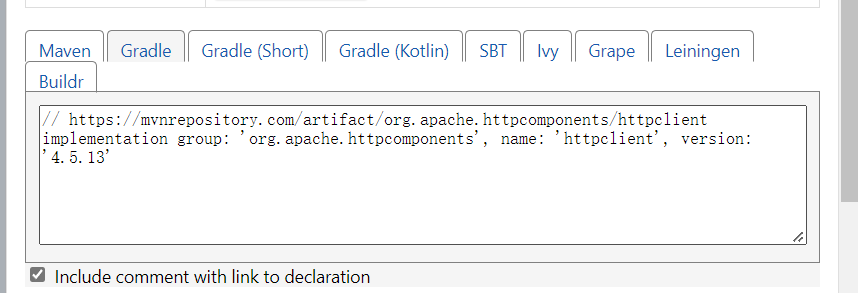
当我们决定用Spring Boot来作为微服务框架时，除了它强大的快速开发功能之外，还因为它在Starter POMs中提供了一个特殊依赖模块spring-boot-starter-actuator，引入该模块能够自动为Spring Boot构建的应用提供一系列用于监控的端点。同时，Spring Cloud在实现各个微服务组件的时候，进一步为该模块做了不少扩展，比如，为原生端点增加了更多的指标和度量信息（比如在整合Eureka的时候会为/health端点增加相关的信息），并且根据不同的组件还提供了更多有空的端点（比如，为API网关组件Zuul提供了/routes端点来返回路由信息）。

spring-boot-starter-actuator模块的实现对于实施微服务的中小团队来说，可以有效地省去或大大减少监控系统在采集应用指标时的开发量。当然，它也并不是万能的，有时候也需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。

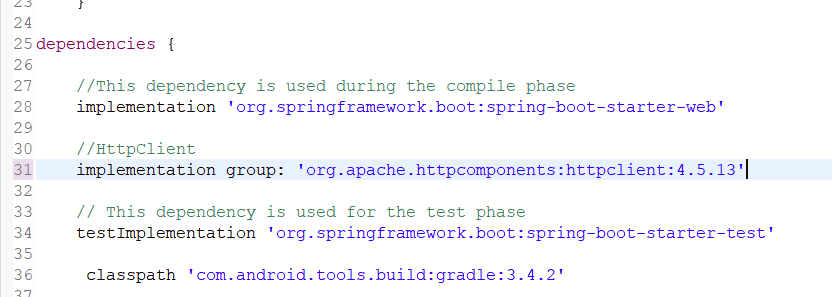
修改源码：

<https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient>



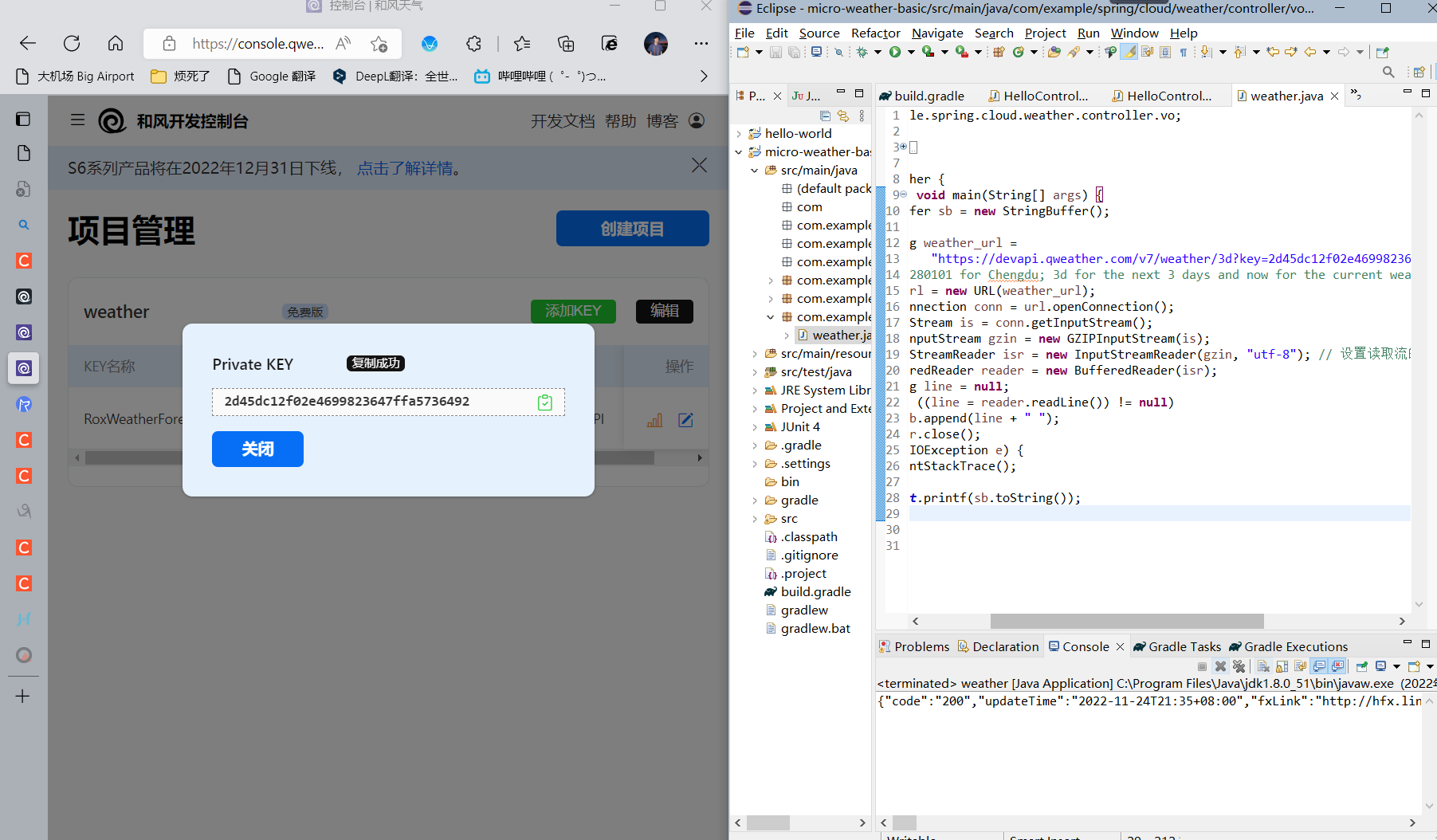


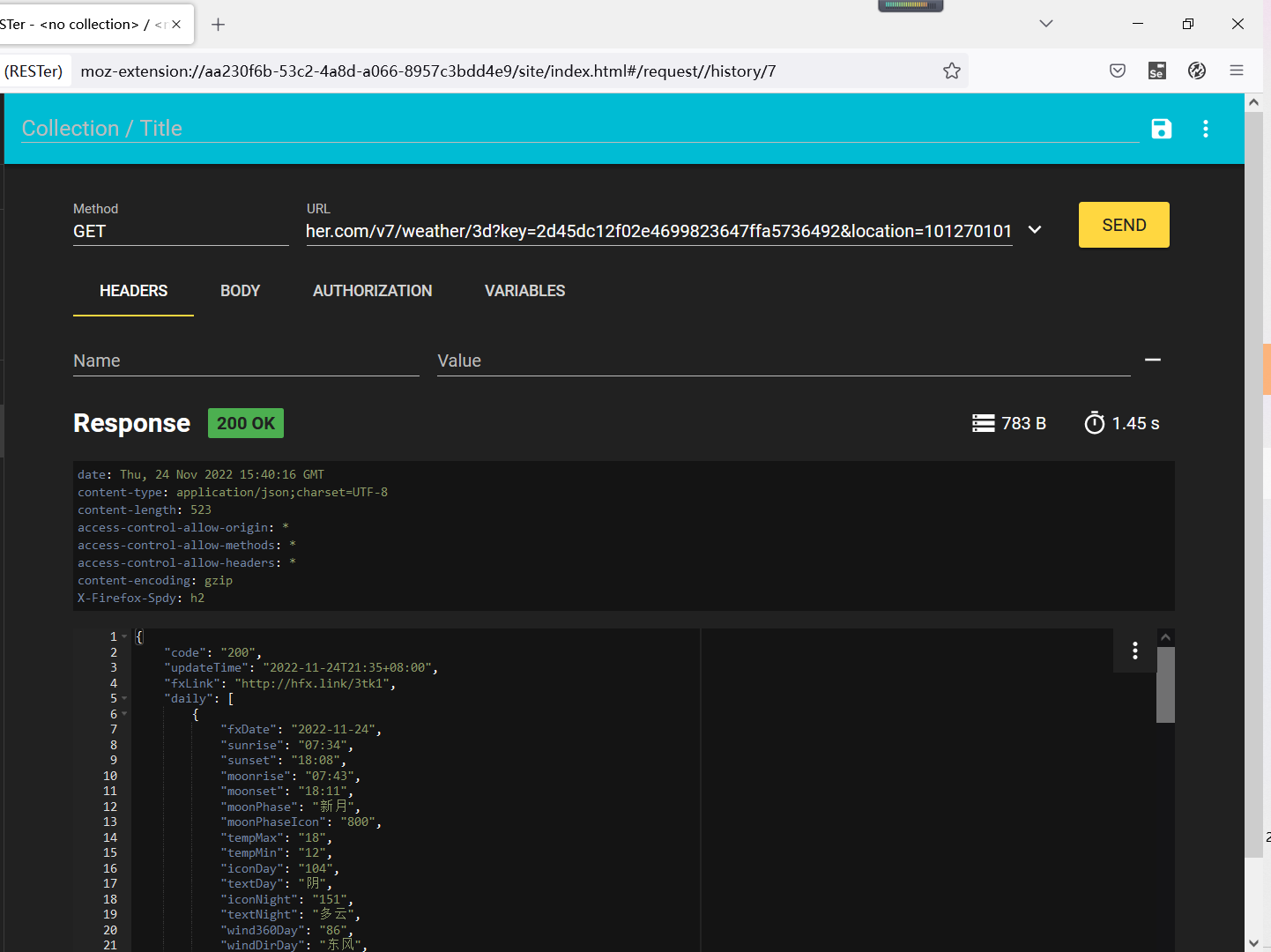
添加依赖



### 数据来源

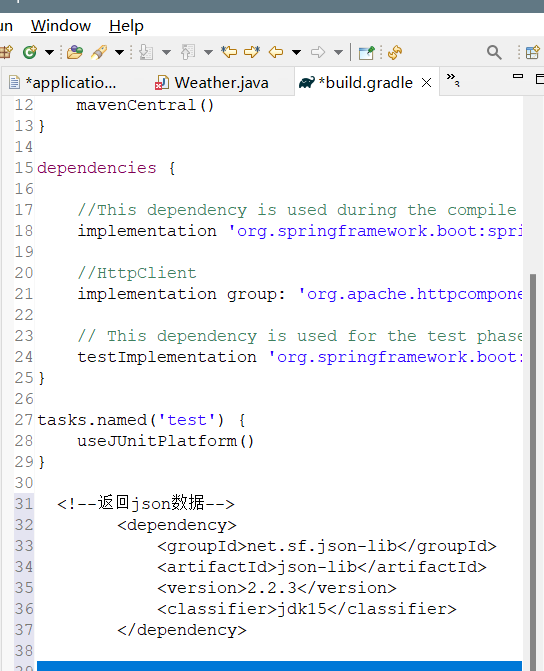
**测试1和风开发api：(否）**

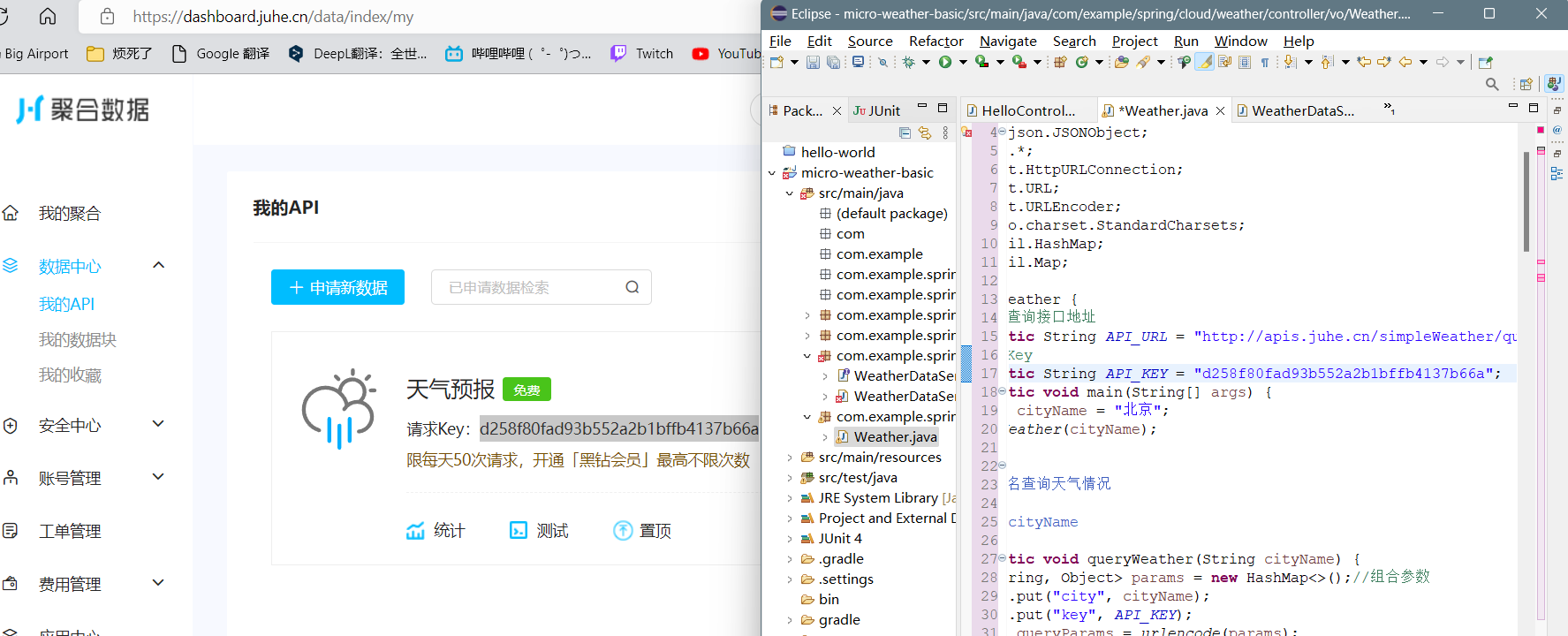


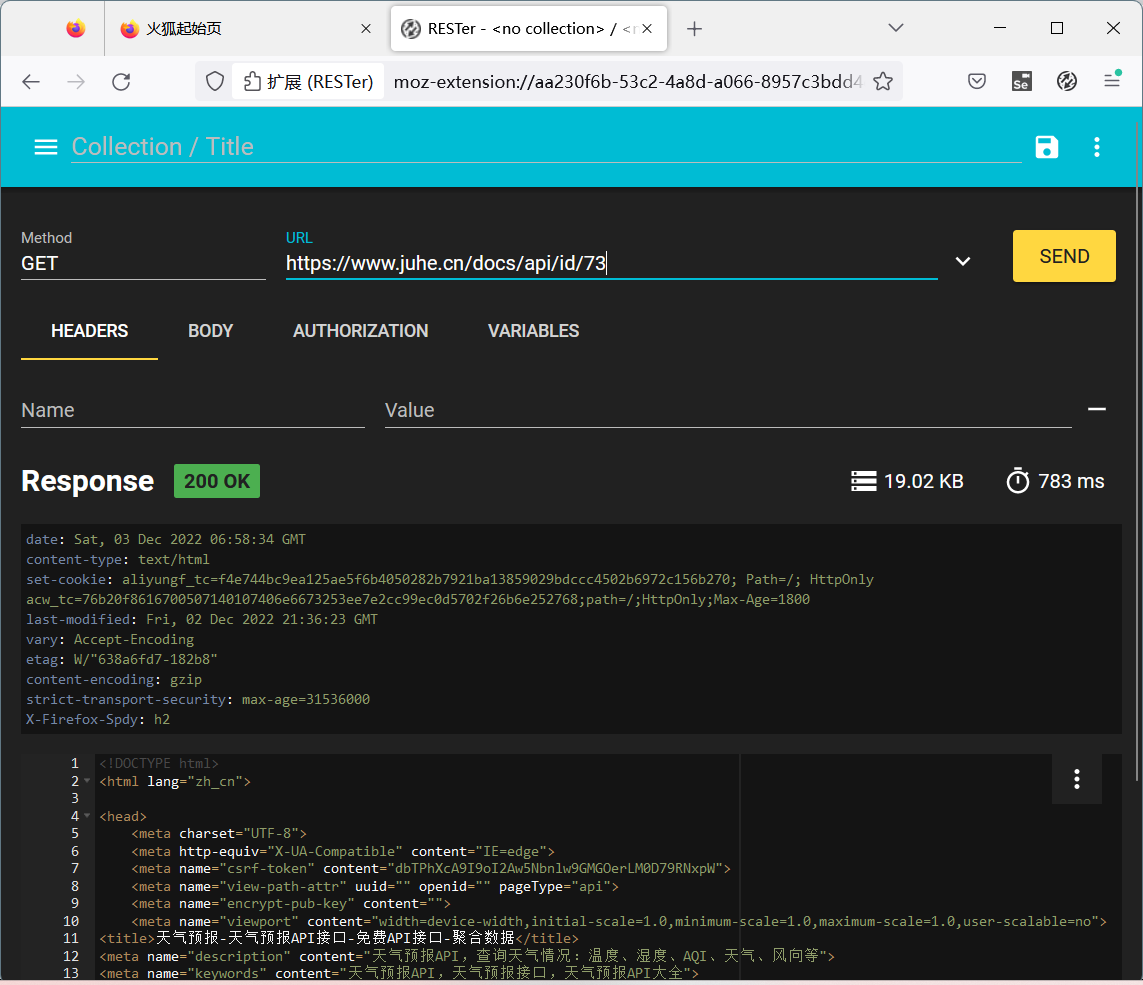


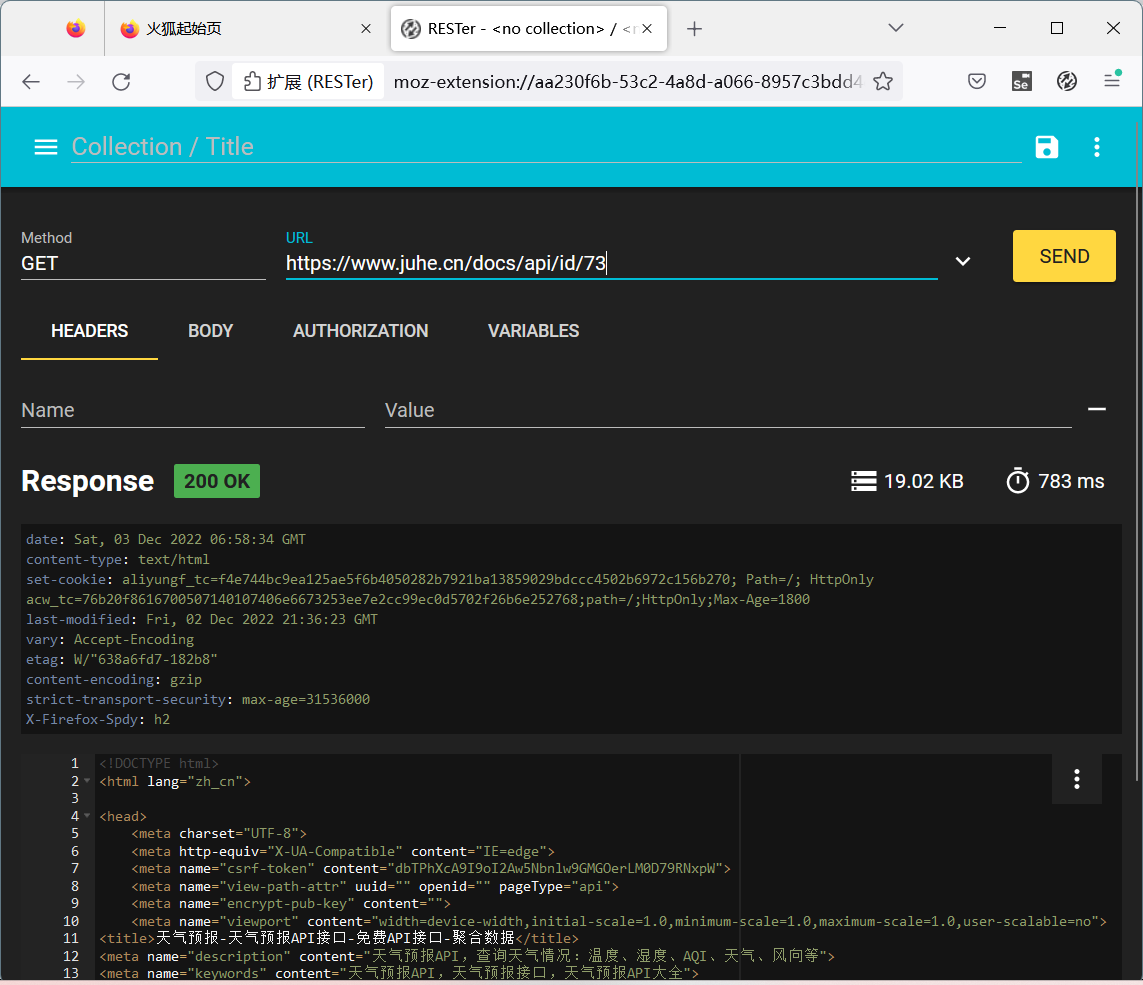
**测试2聚合开发api：（否）**

添加依赖







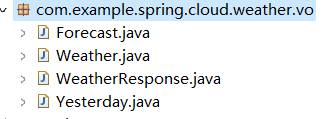


**测试3api数据来源：（允许）**

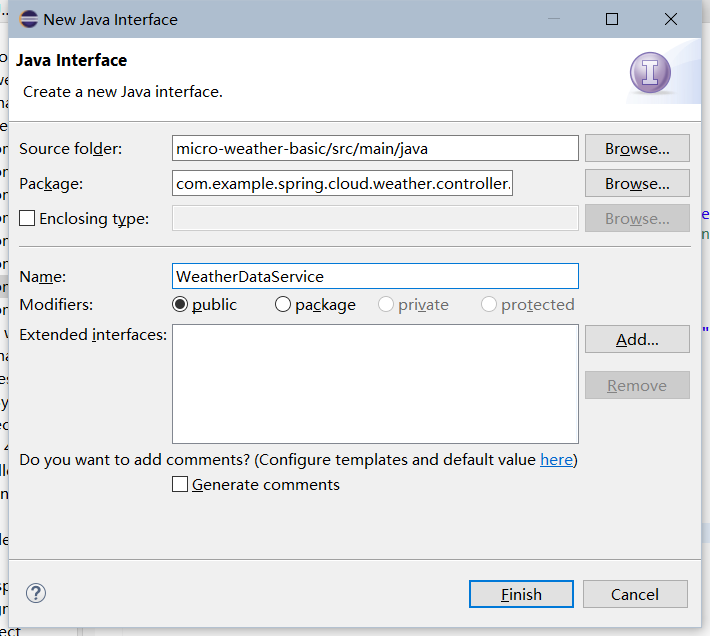


### 新建vo包：

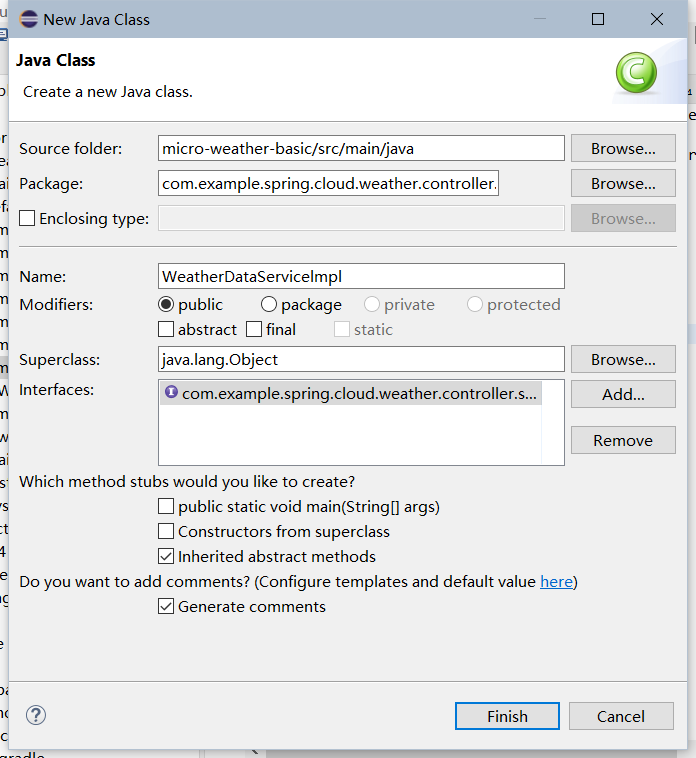
根据数据来源显示的回应字段新建四个类：Weather、Yesterday、Forecast、WeatherResponse：

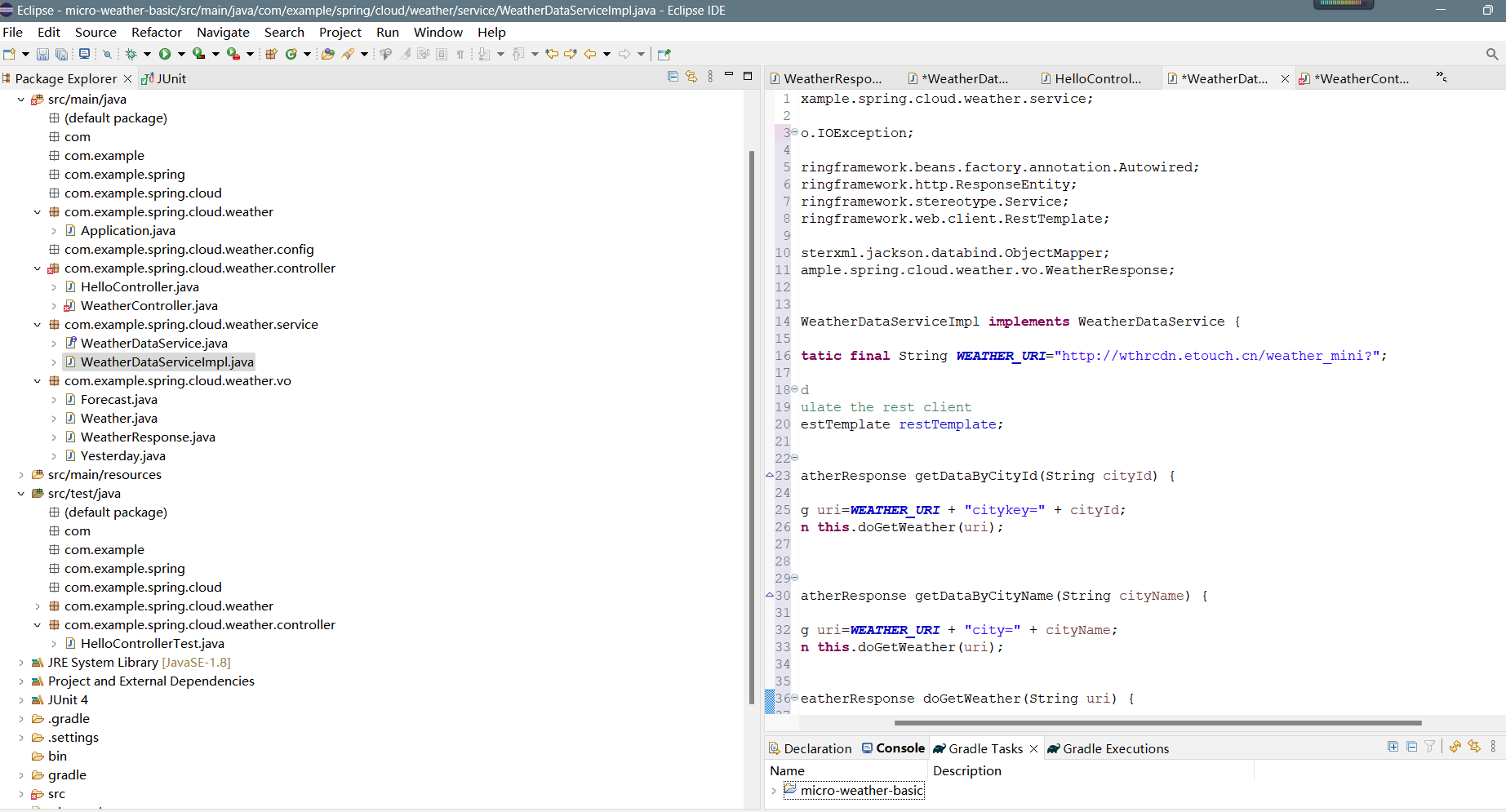


### 新建service接口：



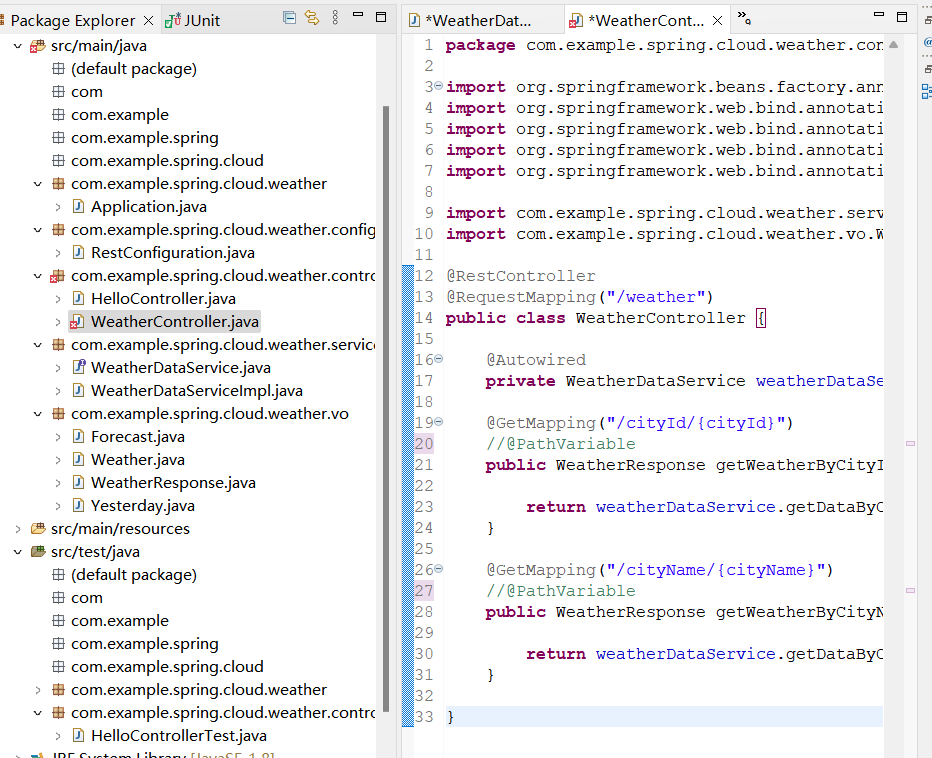
对这个接口创建一个是实现类WeatherDataServiceImpl：





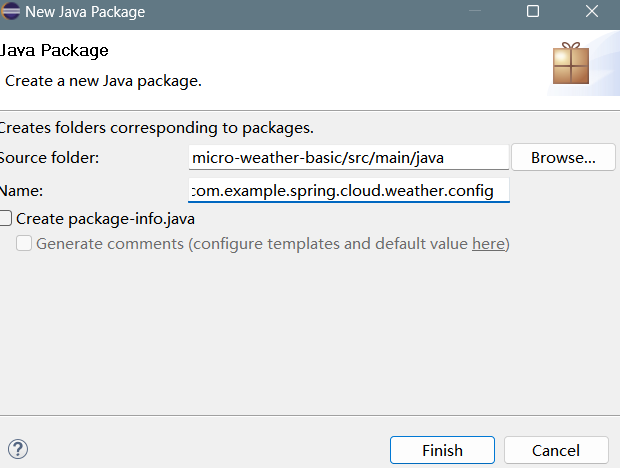
### Controller

新建类

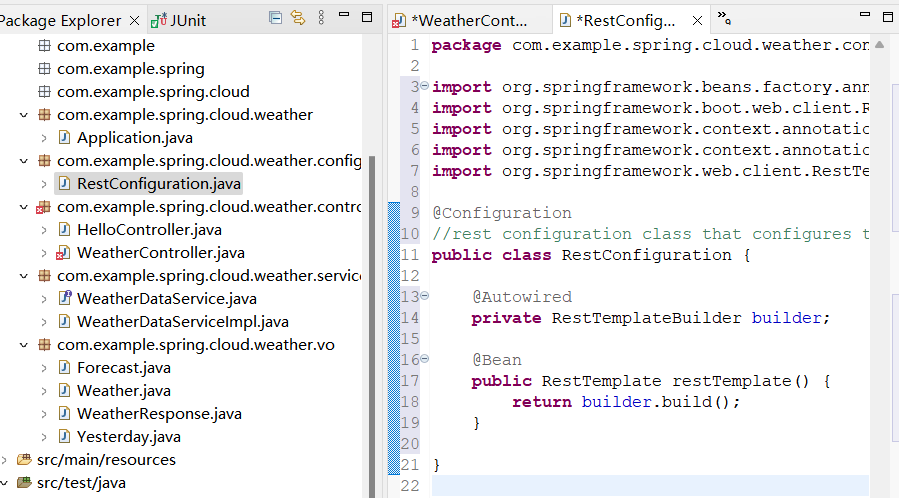


### Config

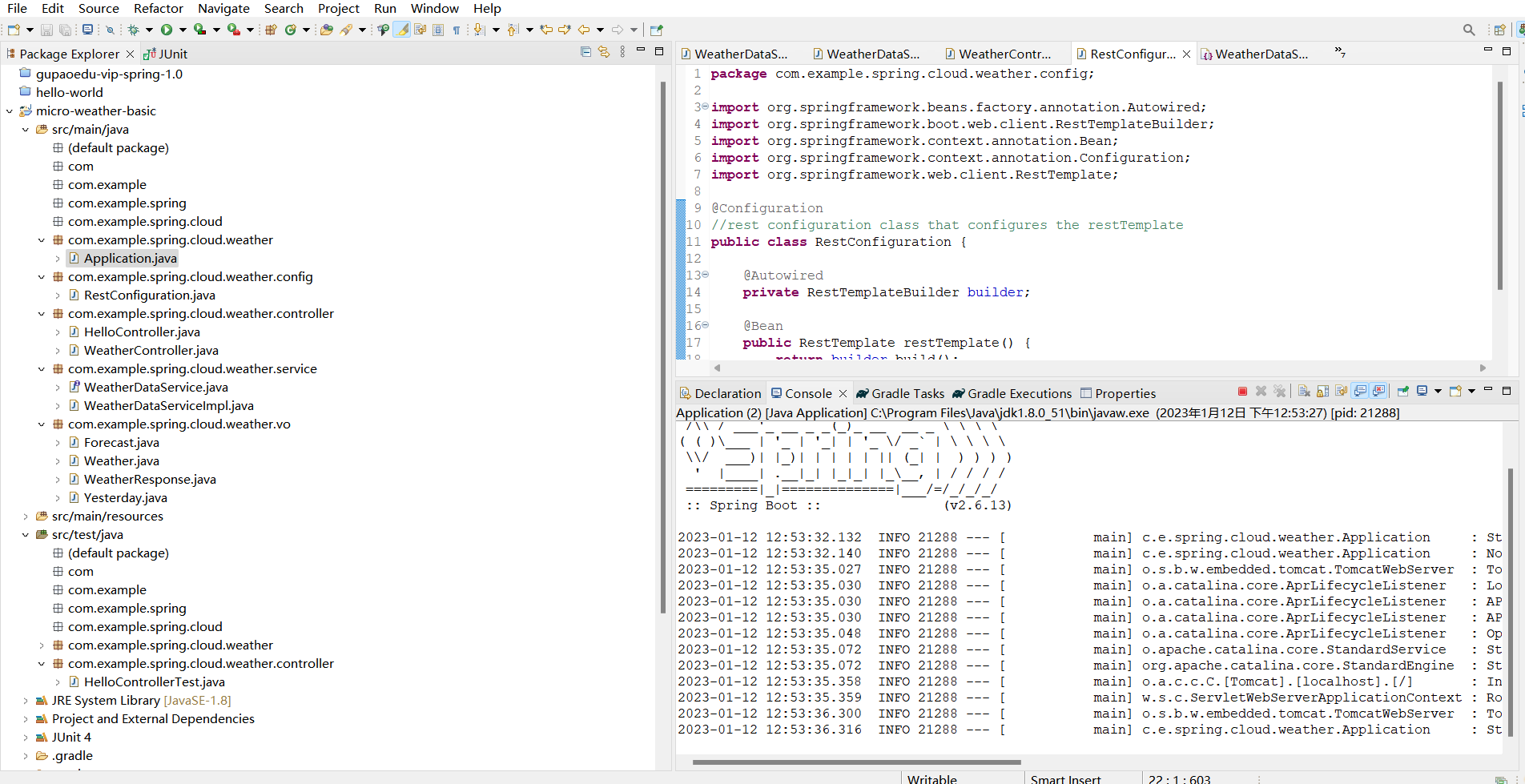
新建包：



新建类：



### 运行Java application

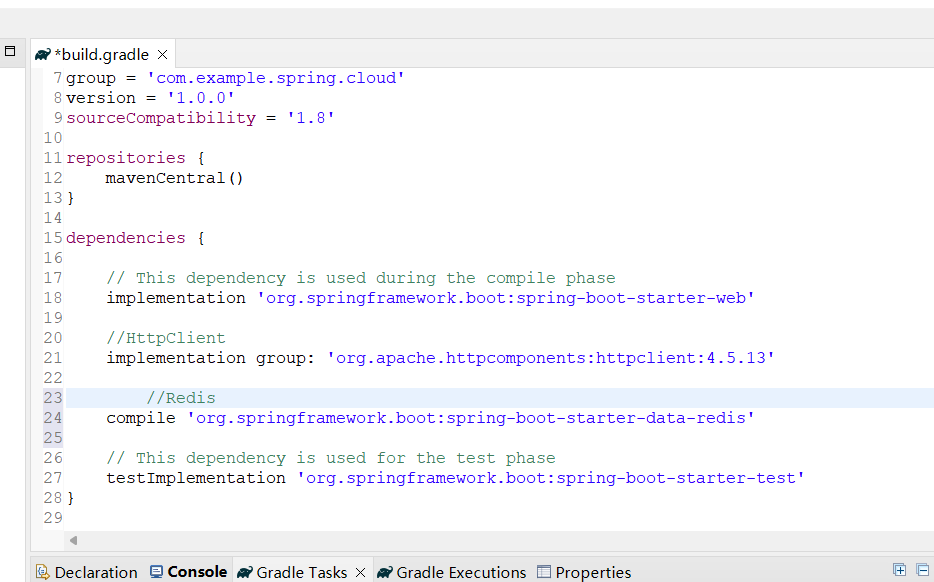


**访问Localhost:8080**

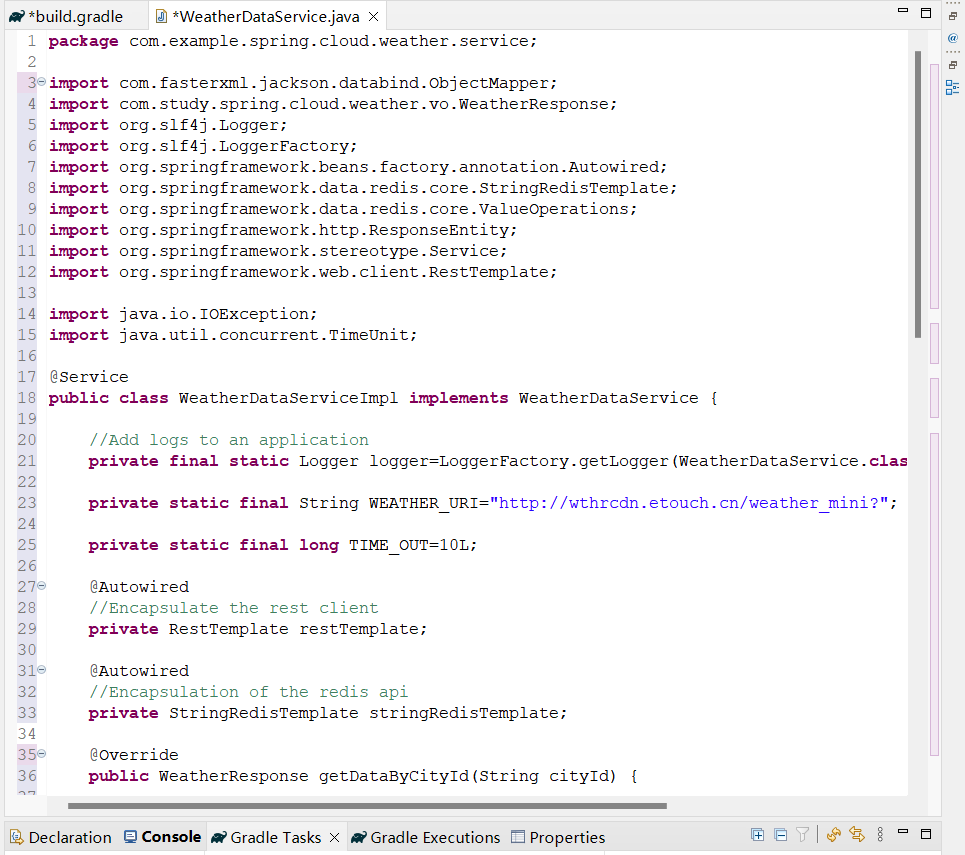
# 用redis提升应用的并发访问能力

### 新建项目micro-weather-redis

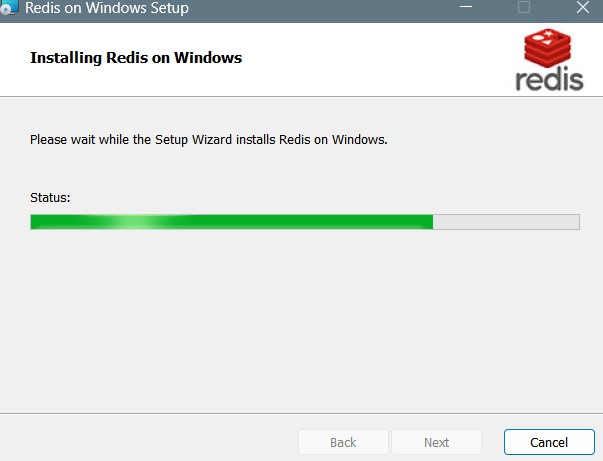
### 加入redis依赖：

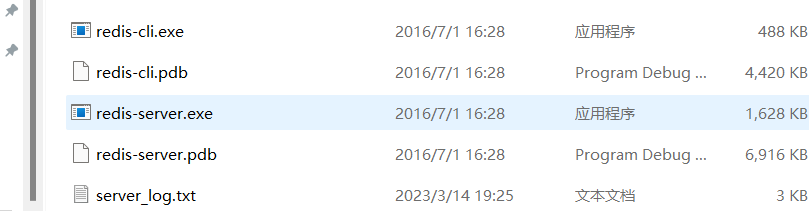


修改WeatherDataServiceImpl.java：



### 安装并运行Redis



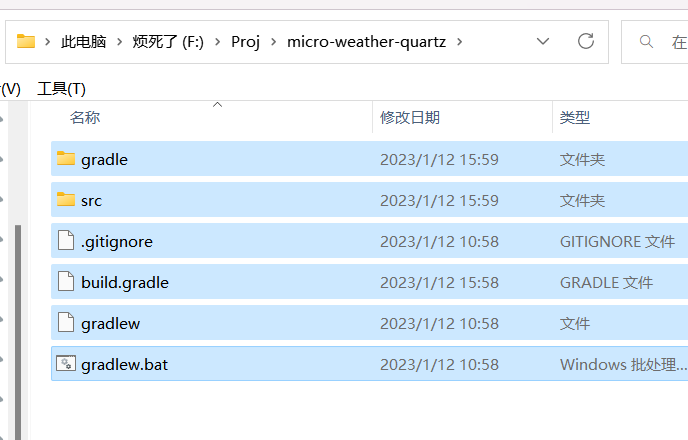


访问结果：



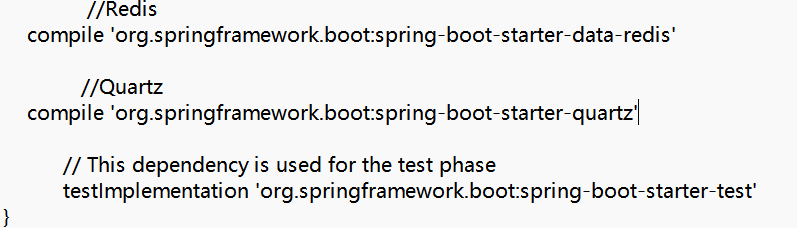
# quartz scheduler定时获取天气数据

### 新建项目文件夹：micro-weather-quartz

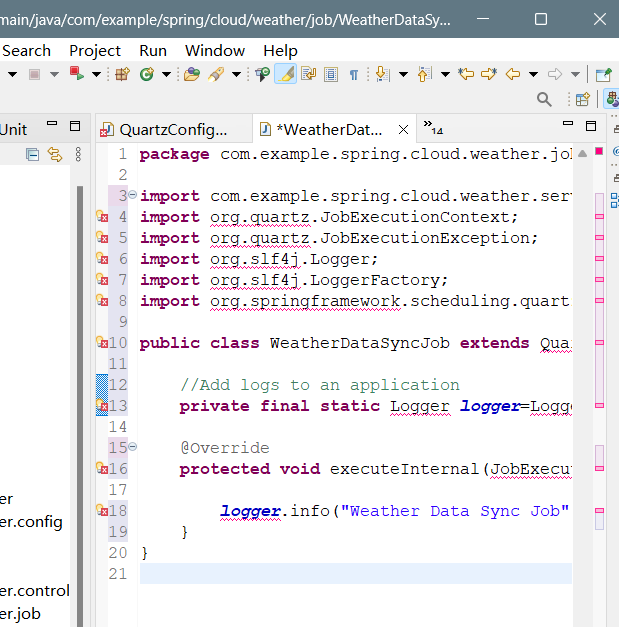


### 修改源码：

添加quartz依赖：



新建包com.example.spring.cloud.weather.job，在包下新建类WeatherDataSyncJob:



在com.example.spring.cloud.weather.config包下新建配置类QuartzConfiguration:



### 准备城市数据

从<http://mobile.weather.com.cn/js/citylist.xml>文件中选取部分城市数据整理成xml文件，如：

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

<c c1="0">

<d d1="101010100" d2="北京" d3="beijing" d4="北京"/>

<d d1="101010200" d2="海淀" d3="haidian" d4="北京"/>

<d d1="101010300" d2="朝阳" d3="chaoyang" d4="北京"/>

<d d1="101010400" d2="顺义" d3="shunyi" d4="北京"/>

<d d1="101010500" d2="怀柔" d3="huairou" d4="北京"/>

<d d1="101010600" d2="通州" d3="tongzhou" d4="北京"/>

<d d1="101010700" d2="昌平" d3="changping" d4="北京"/>

<d d1="101010800" d2="延庆" d3="yanqing" d4="北京"/>

<d d1="101010900" d2="丰台" d3="fengtai" d4="北京"/>

<d d1="101011000" d2="石景山" d3="shijingshan" d4="北京"/>

<d d1="101011100" d2="大兴" d3="daxing" d4="北京"/>

<d d1="101011200" d2="房山" d3="fangshan" d4="北京"/>

<d d1="101011300" d2="密云" d3="miyun" d4="北京"/>

<d d1="101011400" d2="门头沟" d3="mentougou" d4="北京"/>

<d d1="101011500" d2="平谷" d3="pinggu" d4="北京"/>

<d d1="101020100" d2="上海" d3="shanghai" d4="上海"/>

<d d1="101020200" d2="闵行" d3="minhang" d4="上海"/>

<d d1="101020300" d2="宝山" d3="baoshan" d4="上海"/>

<d d1="101020500" d2="嘉定" d3="jiading" d4="上海"/>

<d d1="101020600" d2="南汇" d3="nanhui" d4="上海"/>

<d d1="101020700" d2="金山" d3="jinshan" d4="上海"/>

<d d1="101020800" d2="青浦" d3="qingpu" d4="上海"/>

<d d1="101020900" d2="松江" d3="songjiang" d4="上海"/>

<d d1="101021000" d2="奉贤" d3="fengxian" d4="上海"/>

<d d1="101021100" d2="崇明" d3="chongming" d4="上海"/>

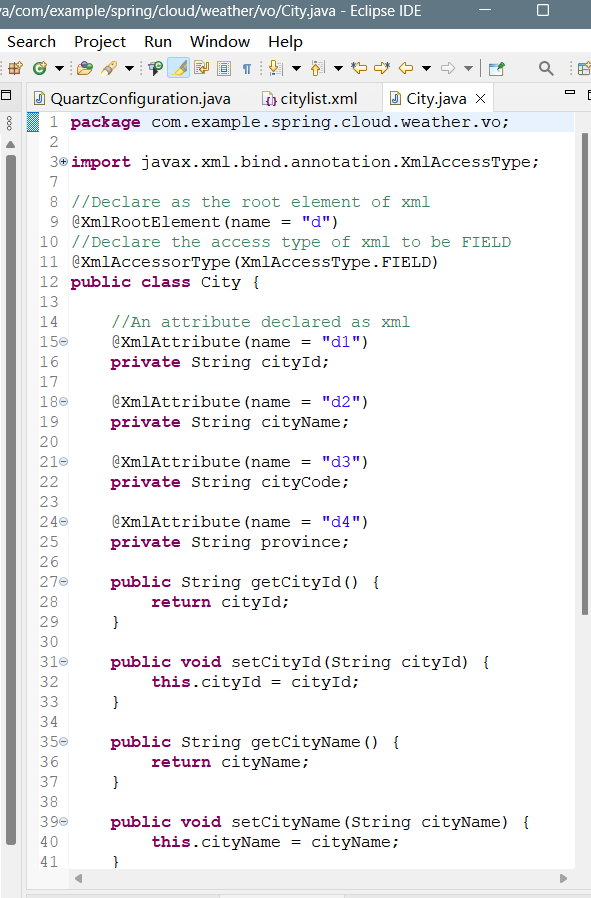
<d d1="101021200" d2="徐家汇" d3="xujiahui" d4="上海"/>

<d d1="101021300" d2="浦东" d3="pudong" d4="上海"/>

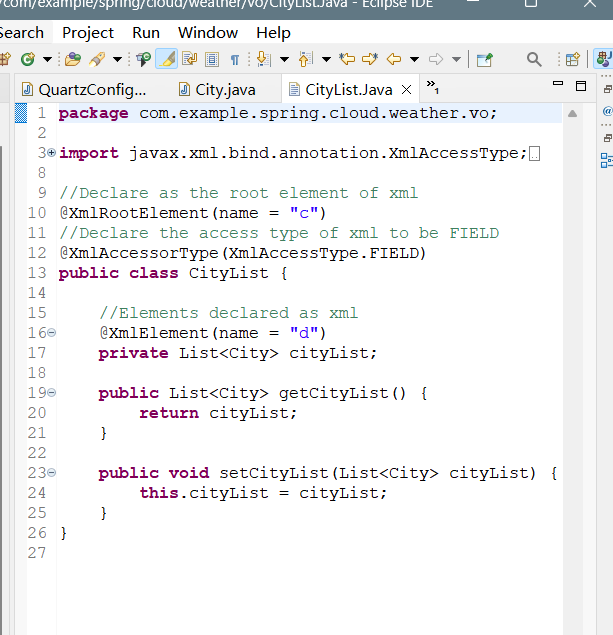
</c>

### 获取城市数据：

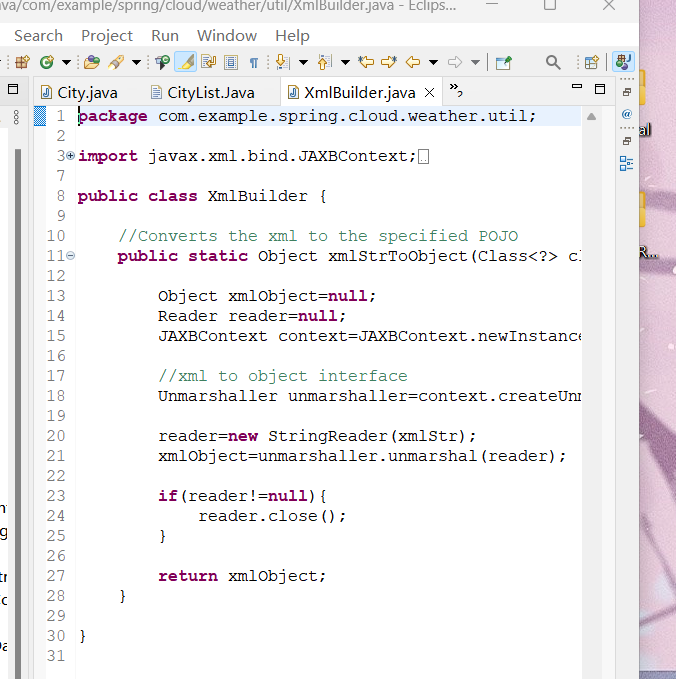
在com.example.spring.cloud.weather.vo包下新建类City：



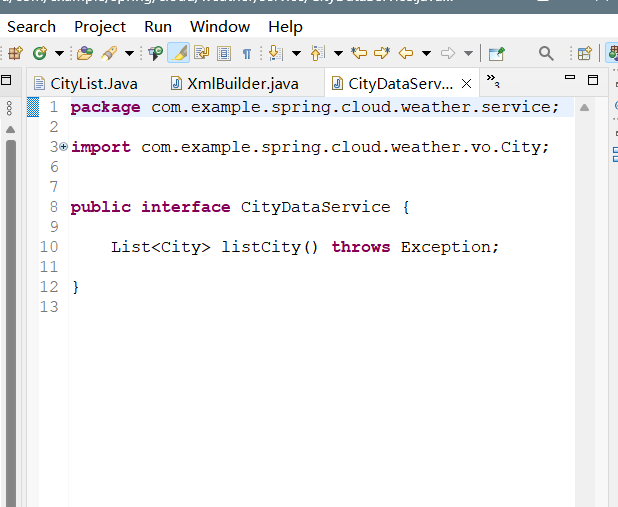
在com.example.spring.cloud.weather.vo包下新建类CityList：



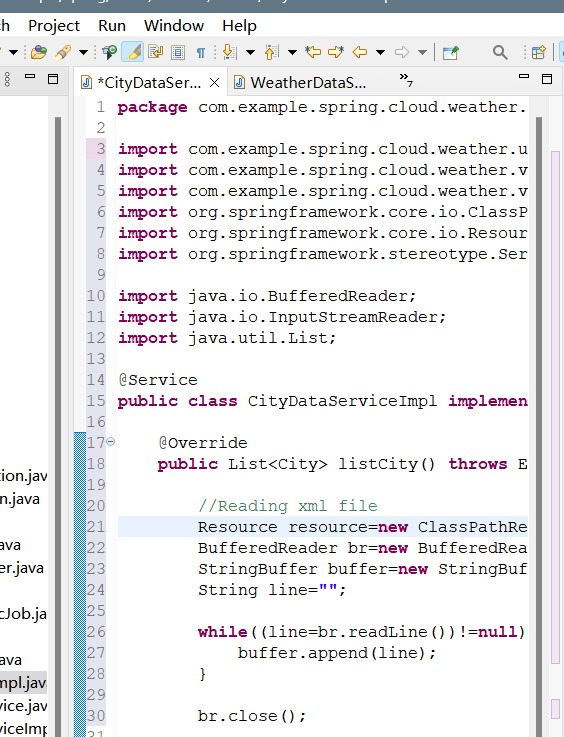
新建包com.example.spring.cloud.weather.util，在包下新建类XmlBuilder：



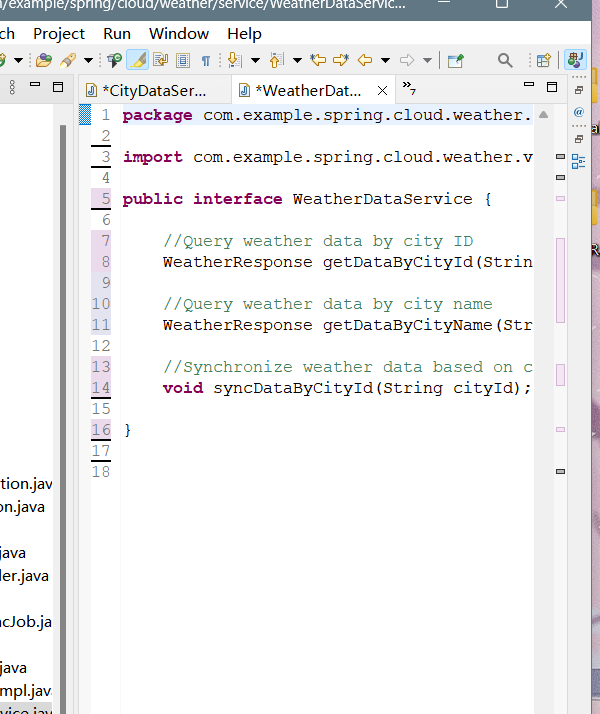
在com.example.spring.cloud.weather.service包下新建接口CityDataService：



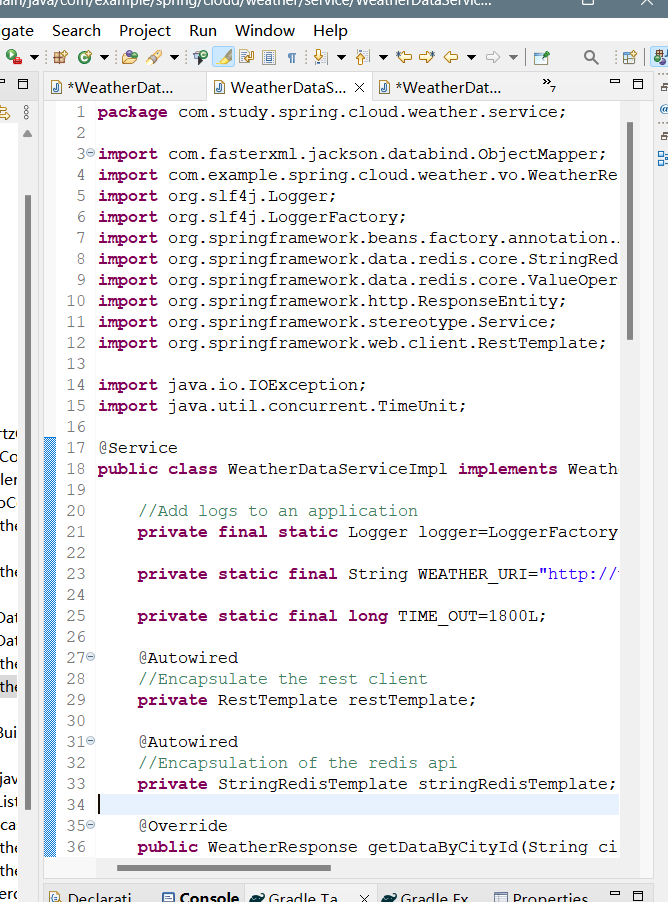
在com.example.spring.cloud.weather.service包下新建类CityDataServiceImpl：



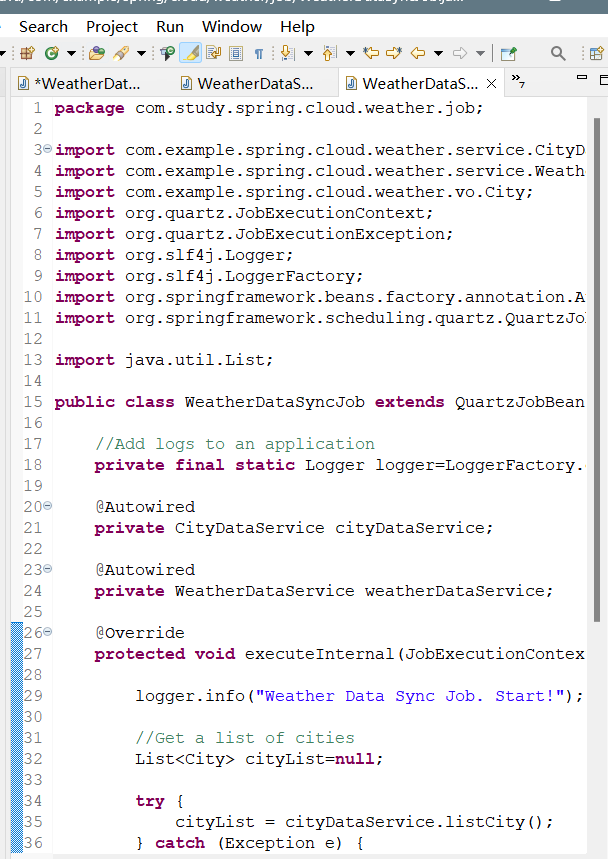
修改WeatherDataService，新增一个方法syncDataByCityId：



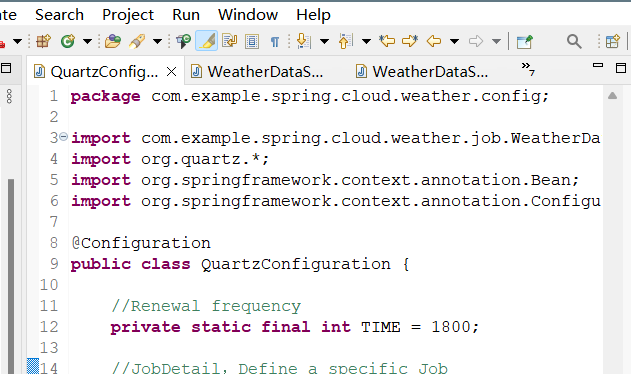
修改WeatherDataServiceImpl，新增syncDataByCityId方法的实现：



修改WeatherDataSyncJob：

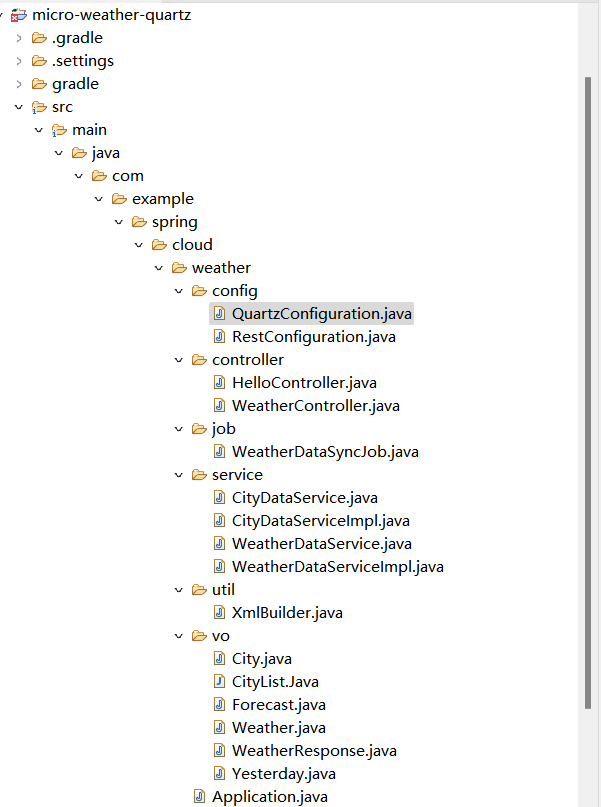


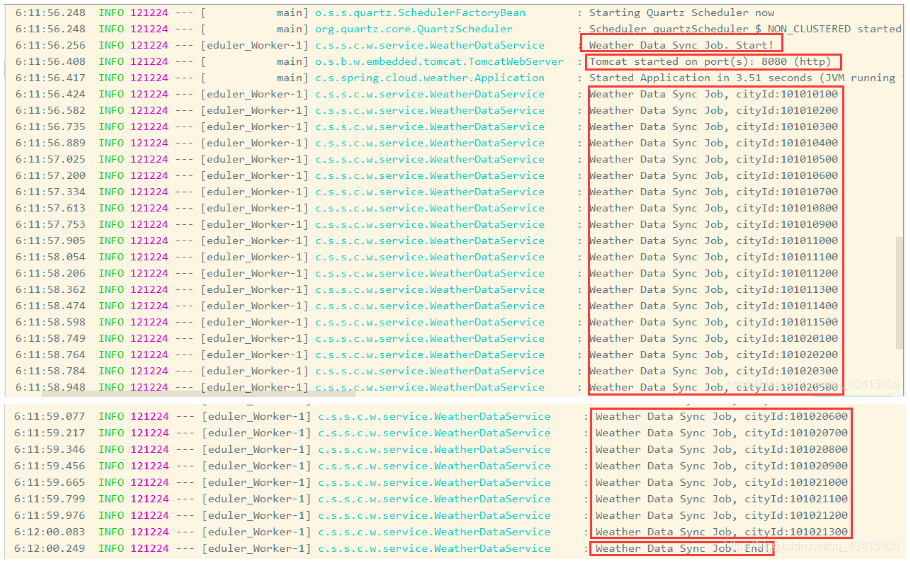
修改QuartzConfig ration的更新频率：

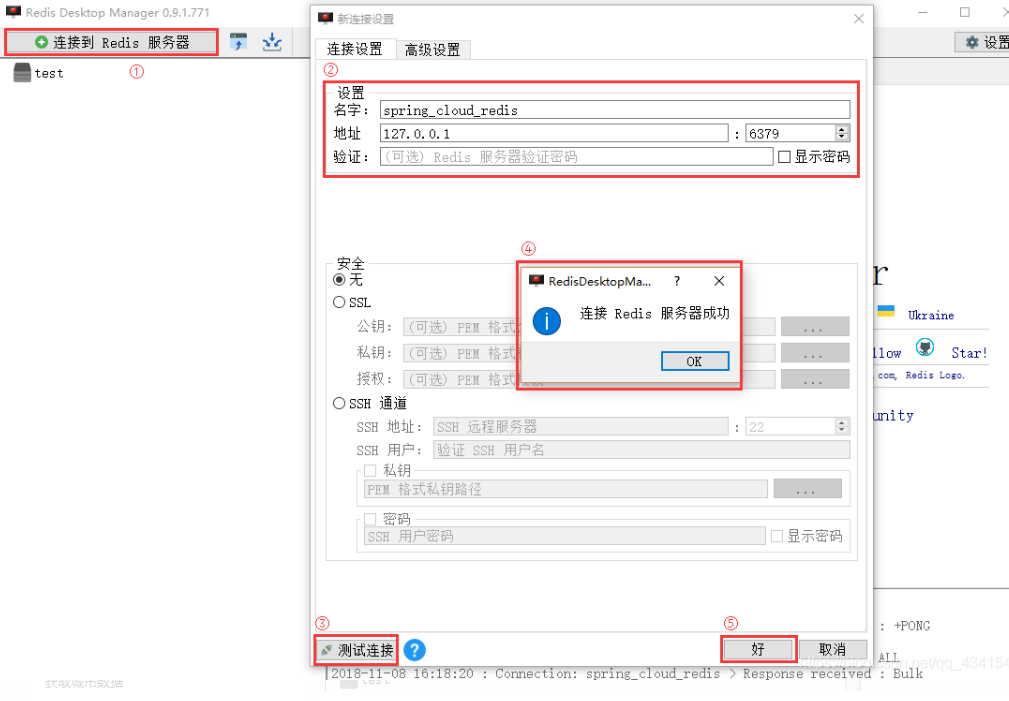


### 运行Redis

最终运行结果：







# 为天气预报制作 UI

天气预报服务的功能

按照不同的城市来进行查询

查询近几天的天气信息

界面简洁、优雅

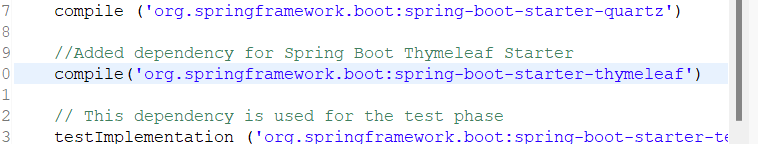
天气预报服务的API

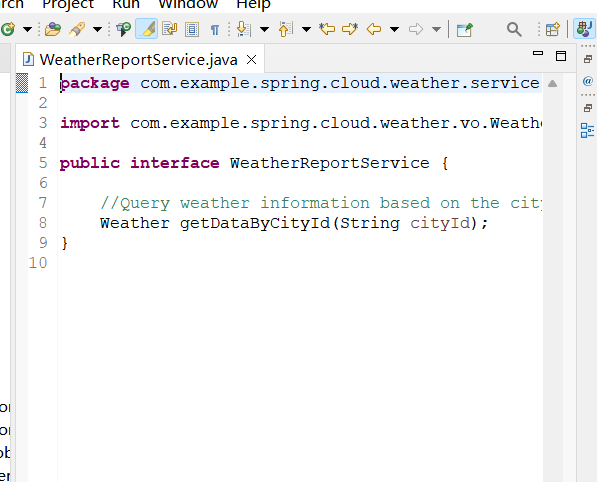
获取到该城市ID的天气预报信息：GET/report/cityId/{cityId}

创建新项目：

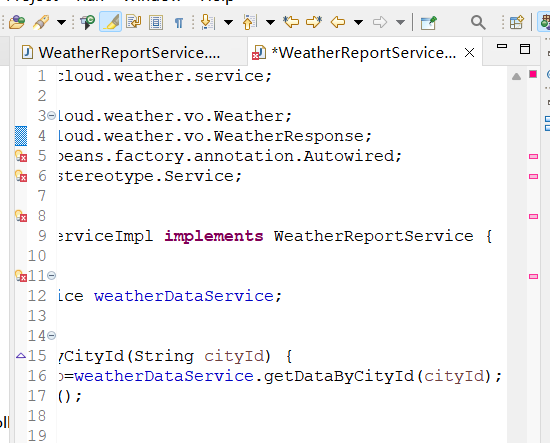


修改build.gradle配置，加入thymeleaf的依赖：

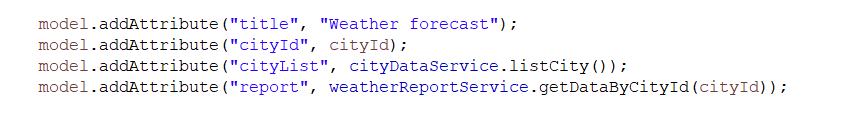


在com.example.spring.cloud.weather.service包下新建接口WeatherReportService：

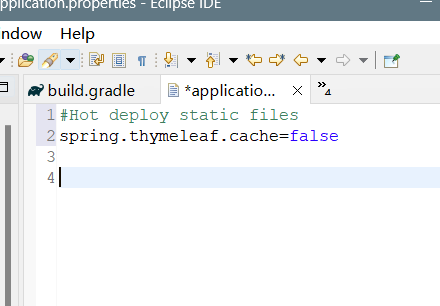
在com.example.spring.cloud.weather.service包下新建类WeatherReportServiceImpl：



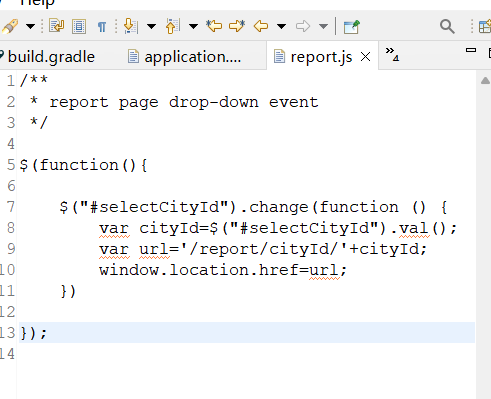
在com.example.spring.cloud.weather.controller包下新建类WeatherReportController：



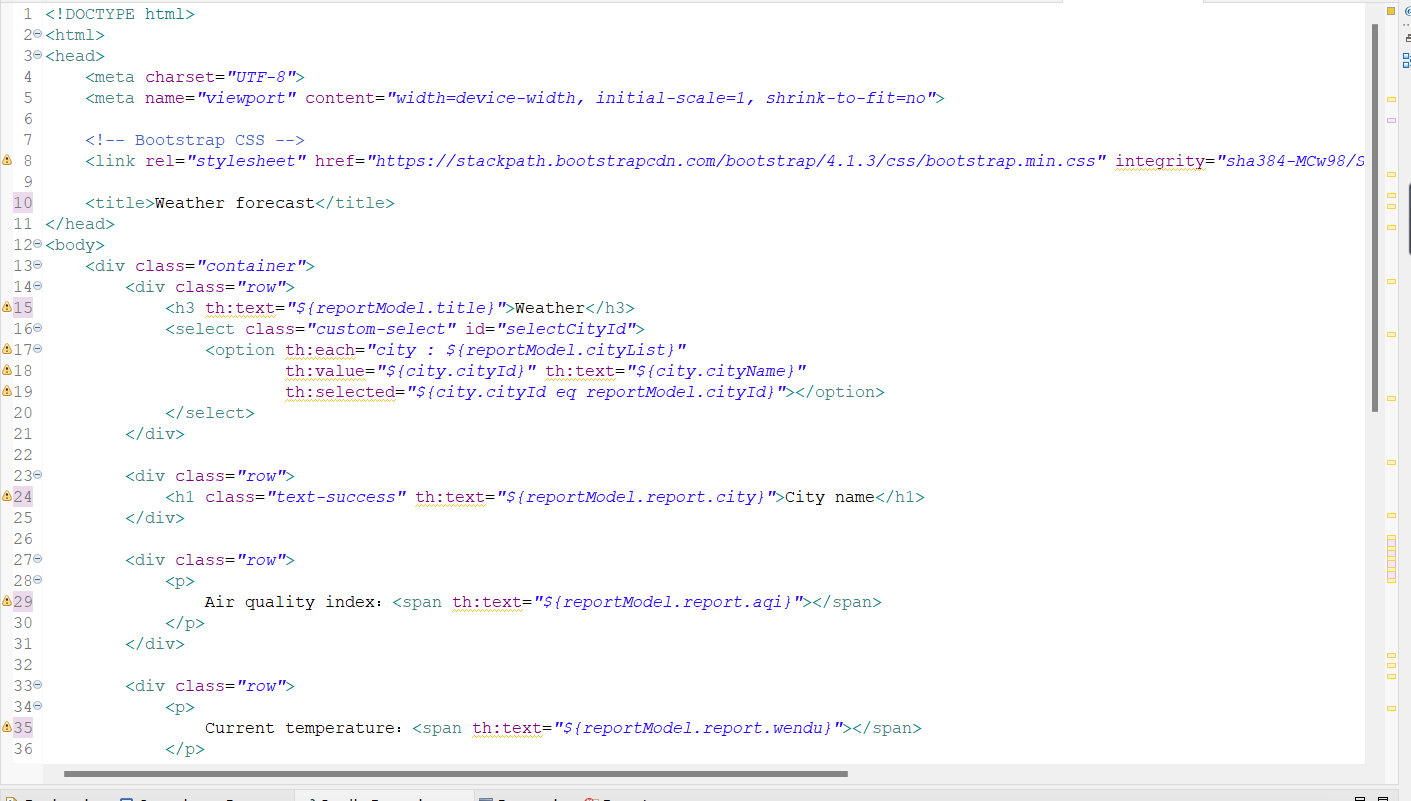
修改application.properties配置：



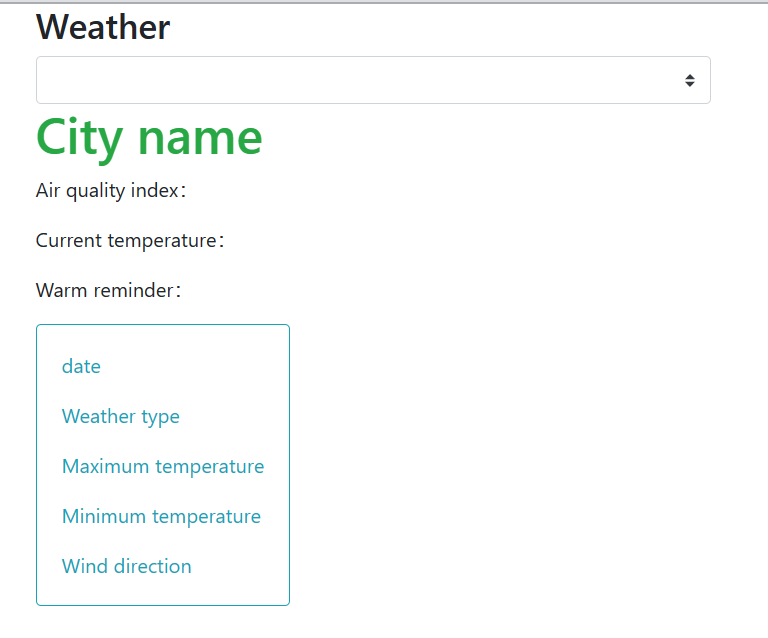
在resources的static目录下创建js目录，再在js目录下创建weather目录，在weather目录下新建report.js文件：



在resources的templates目录下创建weather目录，在weather目录下新建前端页面report.html：

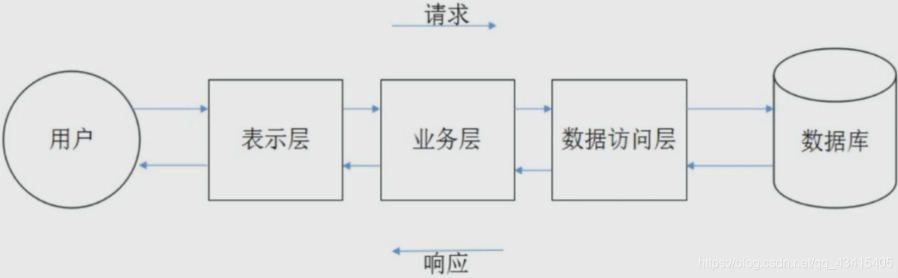


运行结果：



# 单块架构与微服务架构

## 单块架构：



功能代码和数据集中在一起，所有编译文件打成一个发布包，部署在同一个进程中，单块架构是所有大型系统演变的基础和前提。

## 单块架构的优缺点

### 优点

功能划分清楚

层次关系良好

每一层独立

部署简单

技术单一

用人成本低

### 缺点

功能仍然太大

升级风险高，需要升级一个服务时，不得不对整个系统做升级（整个系统一起打发布包）

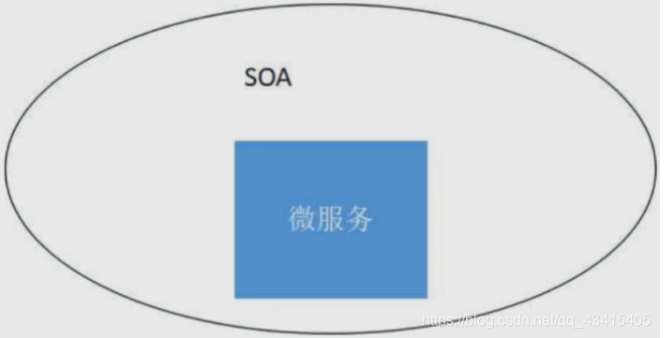
维护成本增加

交付周期变长，必须全部完成才能打包交付

可伸缩性差，因为所有功能点都在同一个系统中，要扩展某一个功能点时必须把所有功能点也扩展

监控困难，不同功能点都在统同一进程中，而很多监控都只能细到进程级别，难以再细化

## 将单块架构转为微服务架构



SOA的出发点是将整个系统打散，成为不同的功能单元，每个单元称为服务，服务之间需要中立的、与平台无关的方式定义的接口，以跨越不同的硬件平台、操作系统平台、编程语言，使得构建在各式各样的系统中可以以一种统一、通用的方式进行交互。

现在将SOA进一步拆分，就成了微服务架构。

微服务是SOA的一种特例，比传统SOA的粒度更小。

## 微服务架构是什么

微服务是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于HTTP的RESTful API进行通信协作。被拆分成的每一个小型服务都围绕着系统中的某一项或一些耦合度较高的业务功能进行构建，并且每个服务都维护着自身的数据存储、业务开发、自动化测试案例以及独立部署机制。由于有了轻量级的通信协作基础，所以这些微服务可以使用不同的语言来编写。

微服务架构就是将一个完整的应用从数据存储开始垂直拆分成多个不同的服务，每个服务都能独立部署、独立维护、独立扩展，服务与服务间通过诸如RESTful API的方式互相调用。

微服务架构是用来开发应用的，这种应用会有很多小的服务来组成，这些服务都可以运行在自己的进程中，即可以实现独立部署，这些服务之间通过轻量级的通讯机制来进行交互，例如http的交互方式，这些服务之间并没有非常强的技术上的关联，可以采用不同的技术栈、使用不同的数据存储形式。

## 微服务架构的设计原则

拆分足够微，颗粒度太大的话无法发挥微服务的优势，颗粒度太细的话会导致服务数量太多引起服务管理问题

轻量级通信

领域驱动原则，要深刻地了解当前要开发的业务功能

单一职责原则

DevOps及两个披萨，每个服务的团队应该小而精并且具备完全自治的全栈能力

不限于技术栈

## 如何来设计微服务系统

服务拆分

服务注册，微服务启动时将微服务信息注册到服务注册中心（或服务注册表）中，服务注册中心可以获取每个微服务的状态

服务发现，想调用另一个服务的功能时，去服务注册表中获取需要调用的服务的信息（如：接口、服务名称等），调用服务

服务消费，指调用其他服务

统一入口，只要知道入口的名称，不需要知道具体的某个服务的名称

配置管理

熔断机制

自动扩展

## 微服务的拆分的意义

易于实现

易于维护

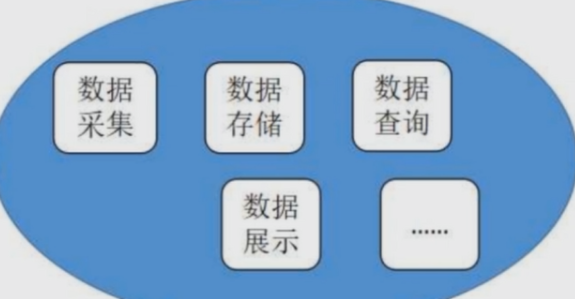
易于部署

易于更新

## 拆分的方法

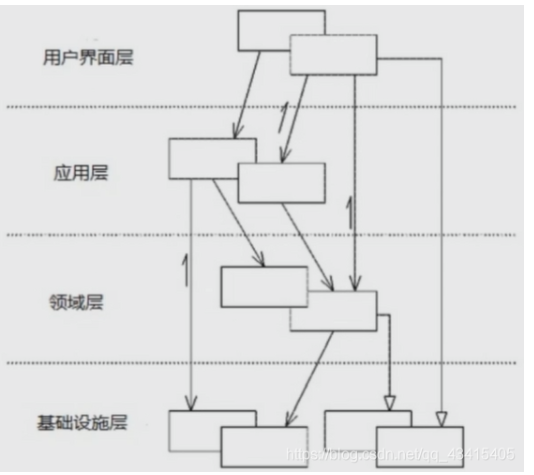
### 横向拆分

按照不同的业务功能拆分成不同的微服务



### 纵向拆分

把一个业务功能里面不同的模块或者组件进行拆分，例如把公共组件拆分成独立的基础设施下沉到底层，形成相对独立的基础设施层



### 使用DDD

DDD就是驱动设计原则来进行拆分。一个微服务应该能够反映出业务的领域模型，使用领域驱动设计不但可以减少微服务环境中通用语言的复杂性，而且还可以帮助团队弄清楚领域的边界、理清上下文的边界

# 天气预报系统微服务架构设计

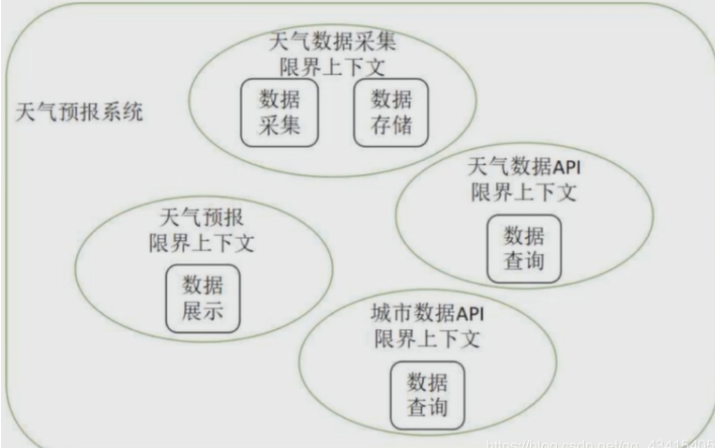
## 现有天气预报系统的弊端

* 大而全
* 混杂了太多的功能
* 难以理解
* 难以维护
* 难以扩展

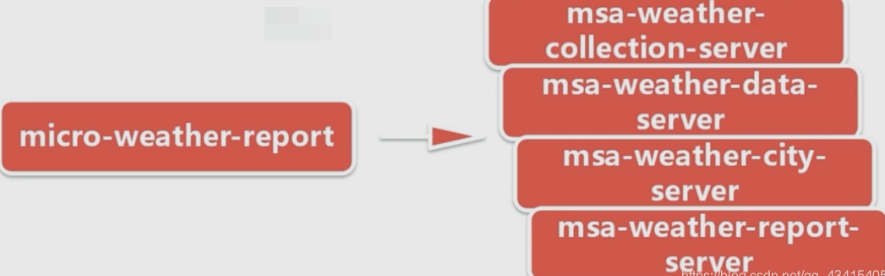
## 改造需求

* 微服务的拆分应该足够的小，每个微服务的业务是非常单一的
* 微服务应能支持水平扩展
* 如果有需要，应能够实现微服务间的相互调用

## 天气预报系统的[微服务](https://so.csdn.net/so/search?q=%E5%BE%AE%E6%9C%8D%E5%8A%A1&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_43415405/article/details/_blank)拆分

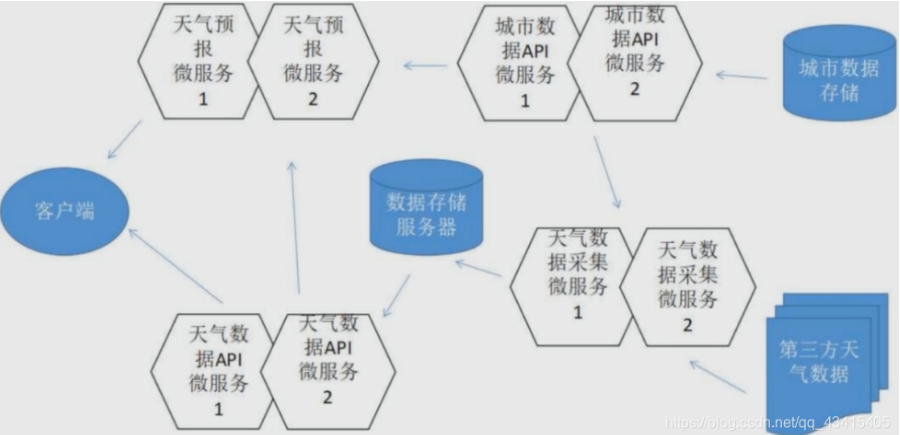


## 微服务代码的拆分



* msa-weather-collection-server：天气数据采集微服务
* msa-weather-data-server：天气数据API微服务
* msa-weather-city-server：城市数据API微服务
* msa-weather-report-server：天气预报微服务

## 系统的数据流向



## 系统的通信设计

### 第三方天气接口

调用方式：GET

http://wthrcdn.etouch.cn/weather\_mini?citykey={cityId}

参数：cityId为城市ID

### 天气数据接口

调用方式：GET/weather/cityId/{cityId}

参数：cityId为城市ID

调用方式：GET/weather/cityName/{cityName}

参数：cityName为城市名称

### 天气预报接口

调用方式：GET/report/cityId/{cityId}

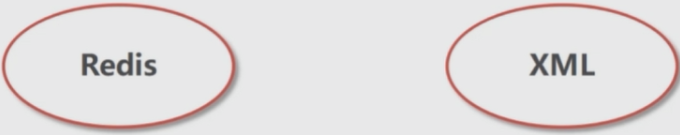
参数：cityId为城市ID

### 城市数据接口

# 调用方式：GET/cities

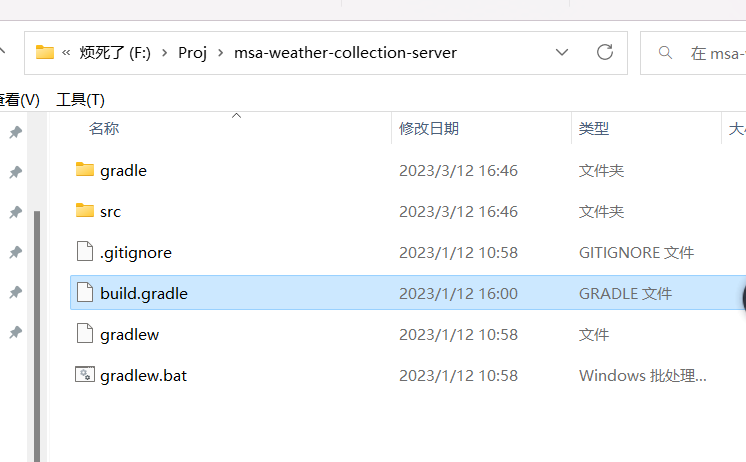
参数：无

## 系统的存储设计

NoSQL

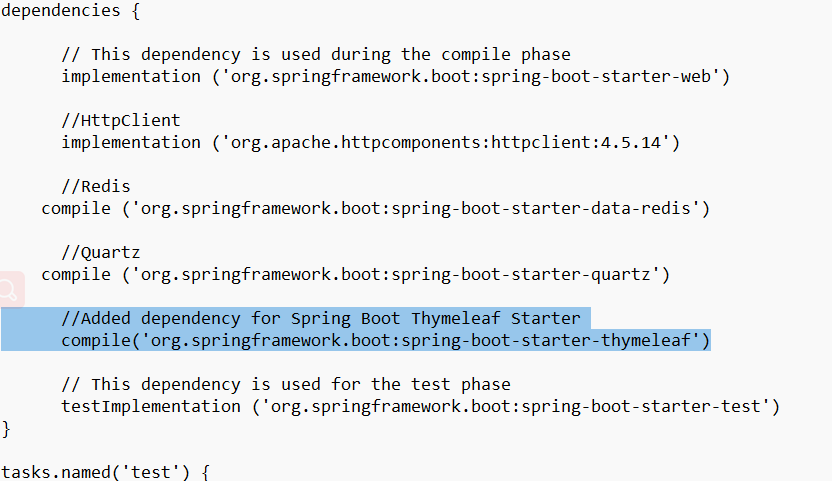
# 天气预报系统微服务 天气数据采集微服务

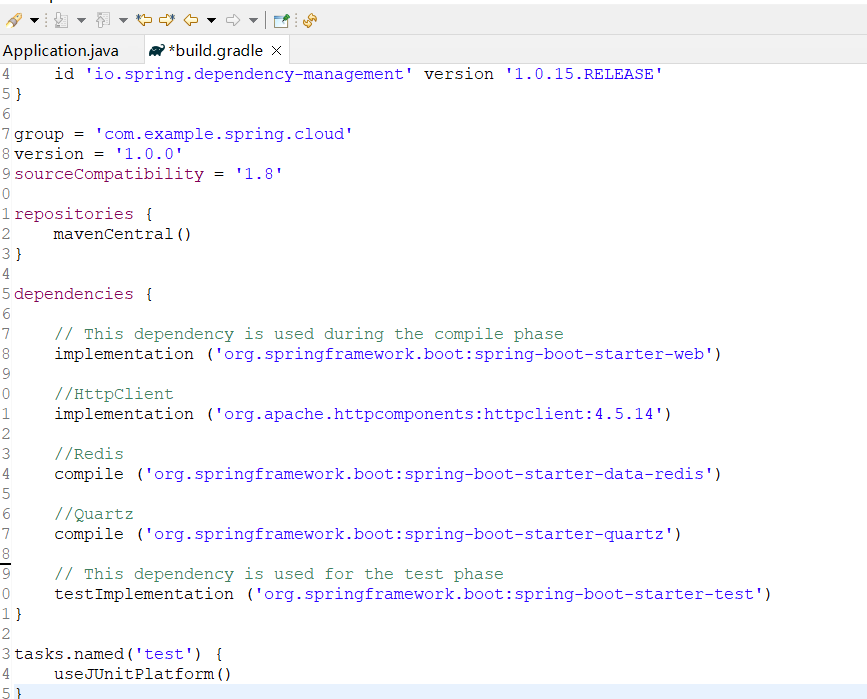
## 新建项目文件夹：msa-weather-collection-server



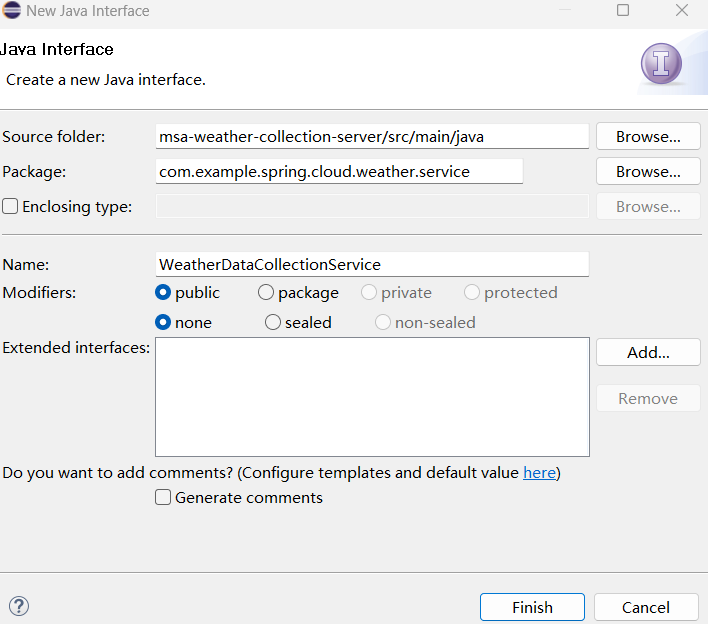
## 修改源码

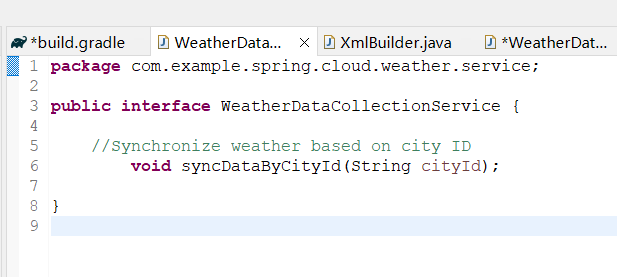
修改build.gradle配置，删除thymeleaf的依赖：

··



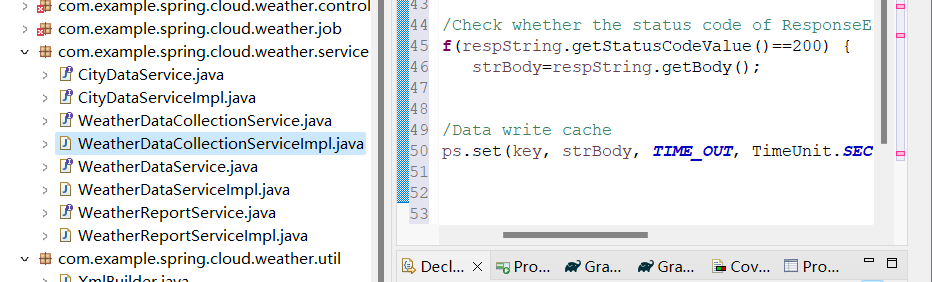
在com.example.spring.cloud.weather.service包下新建类WeatherDataCollectionService：

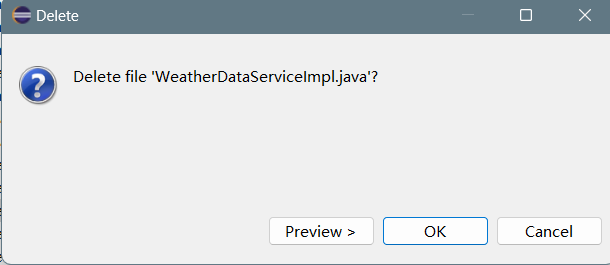
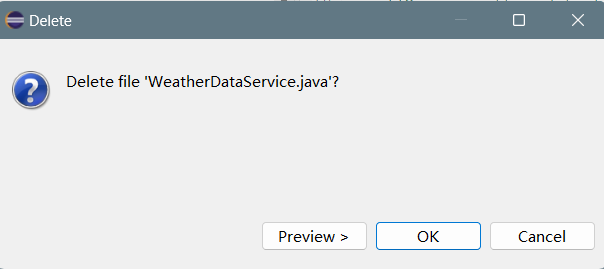




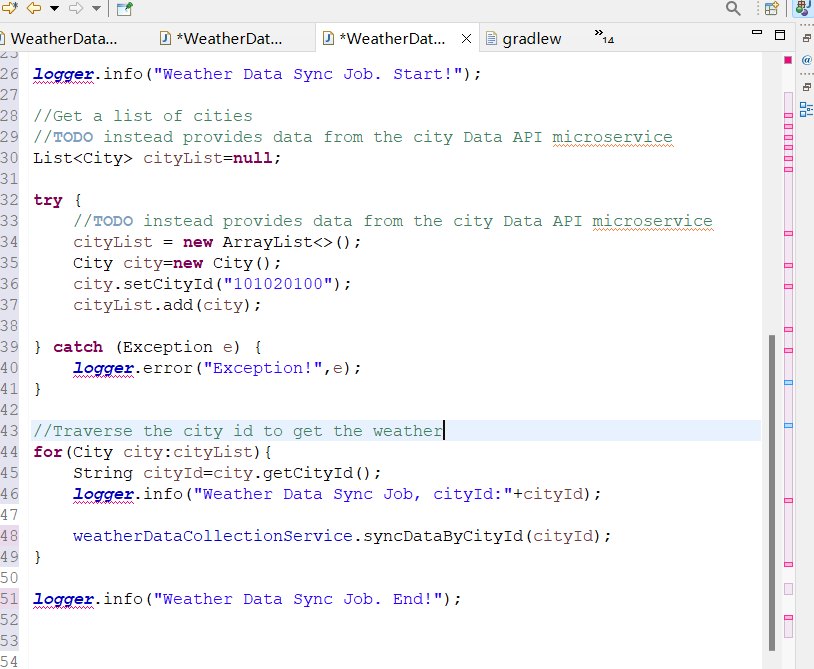
在com.example.spring.cloud.weather.service包下新建类WeatherDataCollectionServiceImpl：



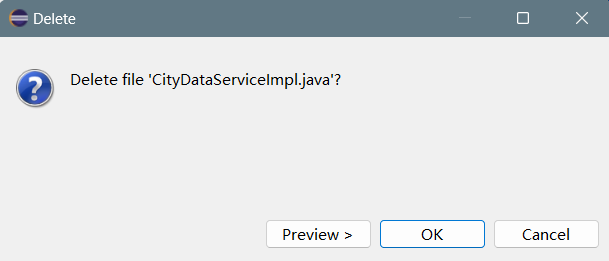
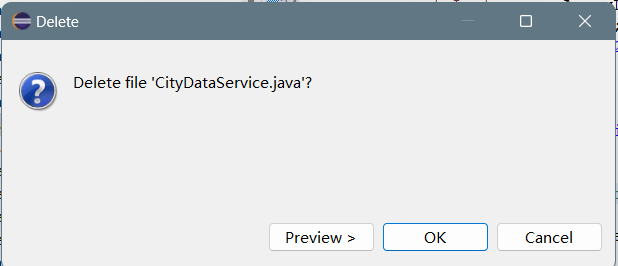


此处代码复制粘贴自WeatherDataServiceImpl类，此类完成后，com.example.spring.cloud.weather.service包中的WeatherDataService和WeatherDataServiceImpl两个类就可以删除了。

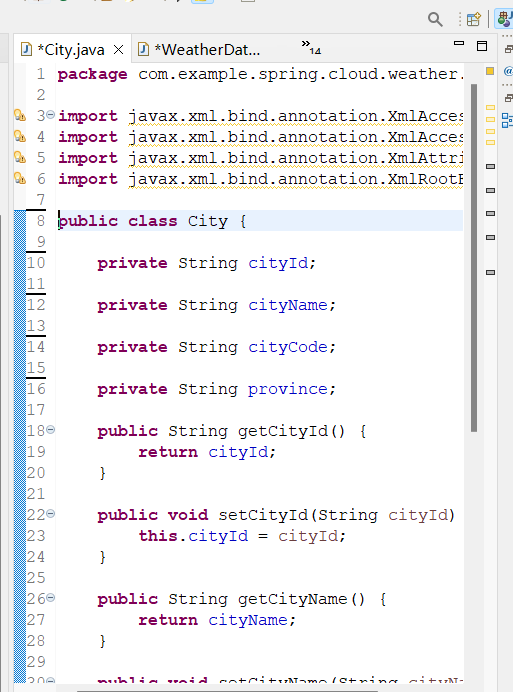
修改com.example.spring.cloud.weather.job包下的WeatherDataSyncJob类：



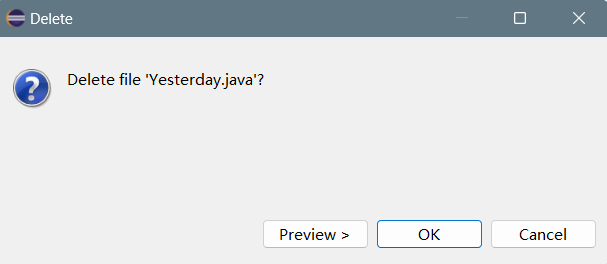
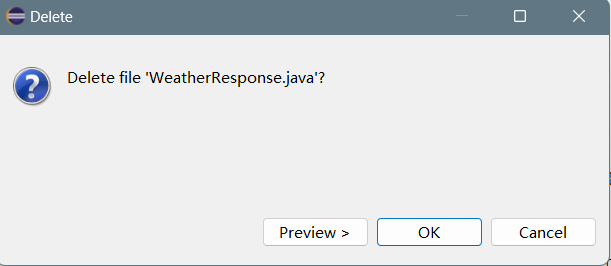
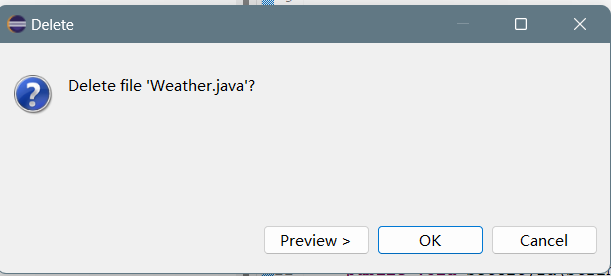
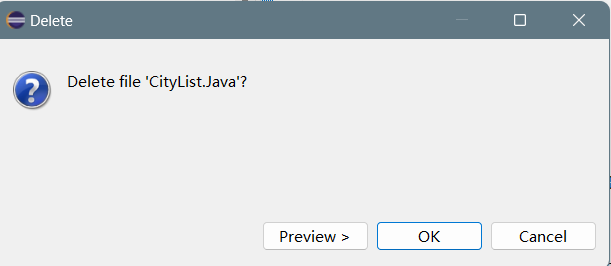
此类完成后，com.example.spring.cloud.weather.service包中的CityDataService和CityDataServiceImpl两个类就可以删除了。



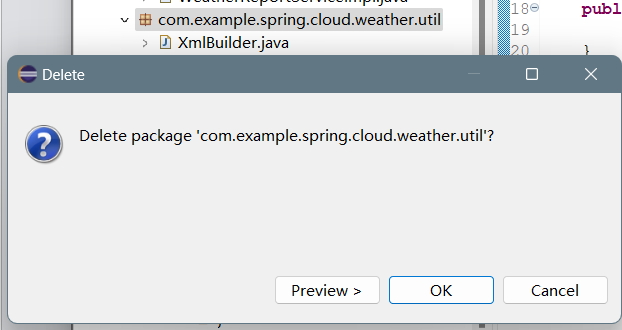
修改com.example.spring.cloud.weather.vo包下的City类，去掉其中xml相关解析代码：



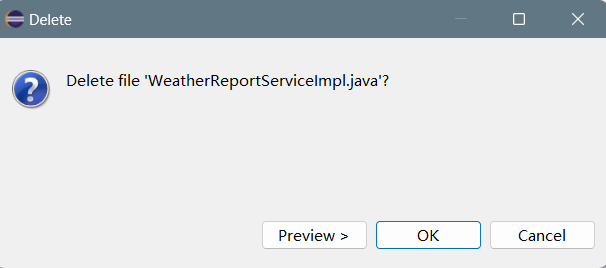
com.example.spring.cloud.weather.vo包下的其他类（CityList、Forecast、Weather、 WeatherResponse、Yesterday）全都删除。



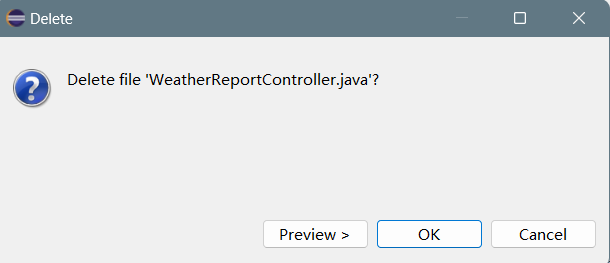
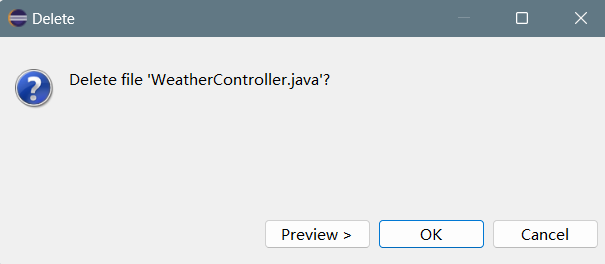
com.example.spring.cloud.weather.util包连同其下的类XmlBuilder删除。



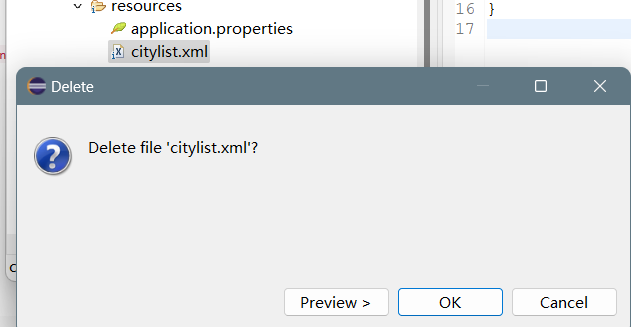
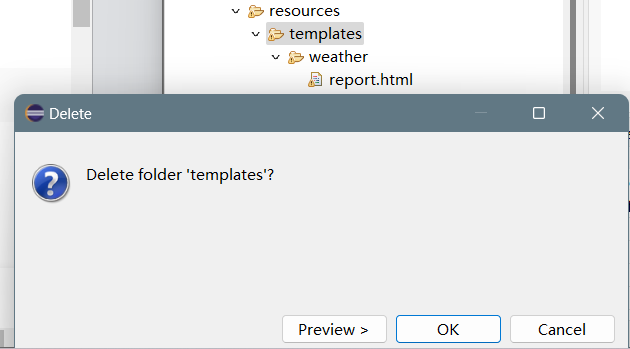
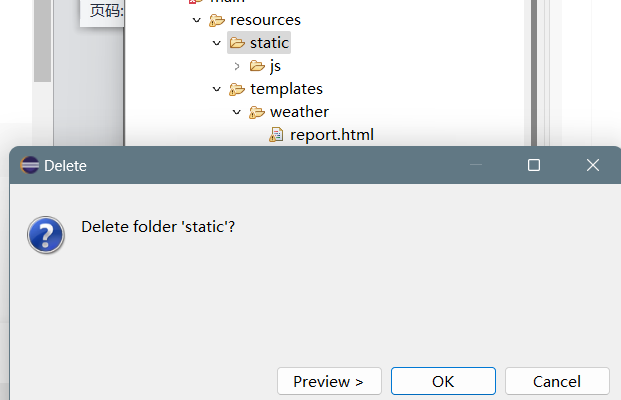
com.example.spring.cloud.weather.service包下的WeatherReportService和WeatherReportServiceImpl两个类删除。



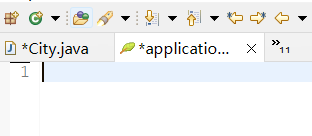
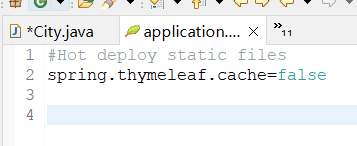
com.example.spring.cloud.weather.controller包下的WeatherController和WeatherReportController两个类删除。



删除前端目录下(resources/)static/js/weather/report.js和(resources/)templates/weather/report.html，并且删除resources包中citylist.xml文件。



修改application.properties配置文件，将以下图中内容删除



## 运行

先运行Redis

运行应用：Application

# Spring Cloud 简介、服务治理（Eureka）