## 环境配置

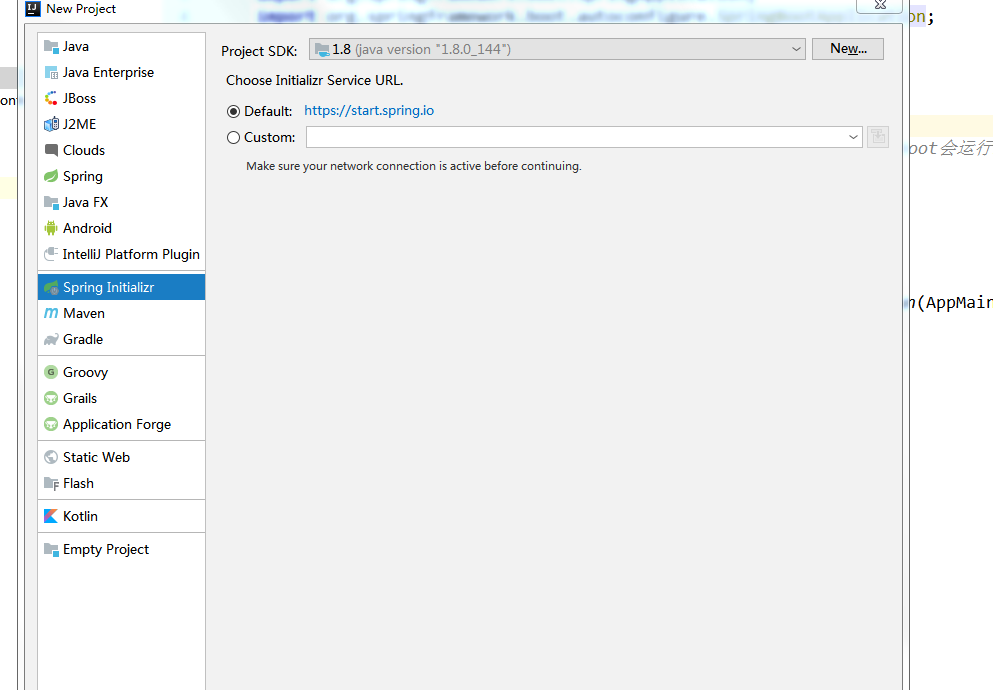
### 1.1手动配置

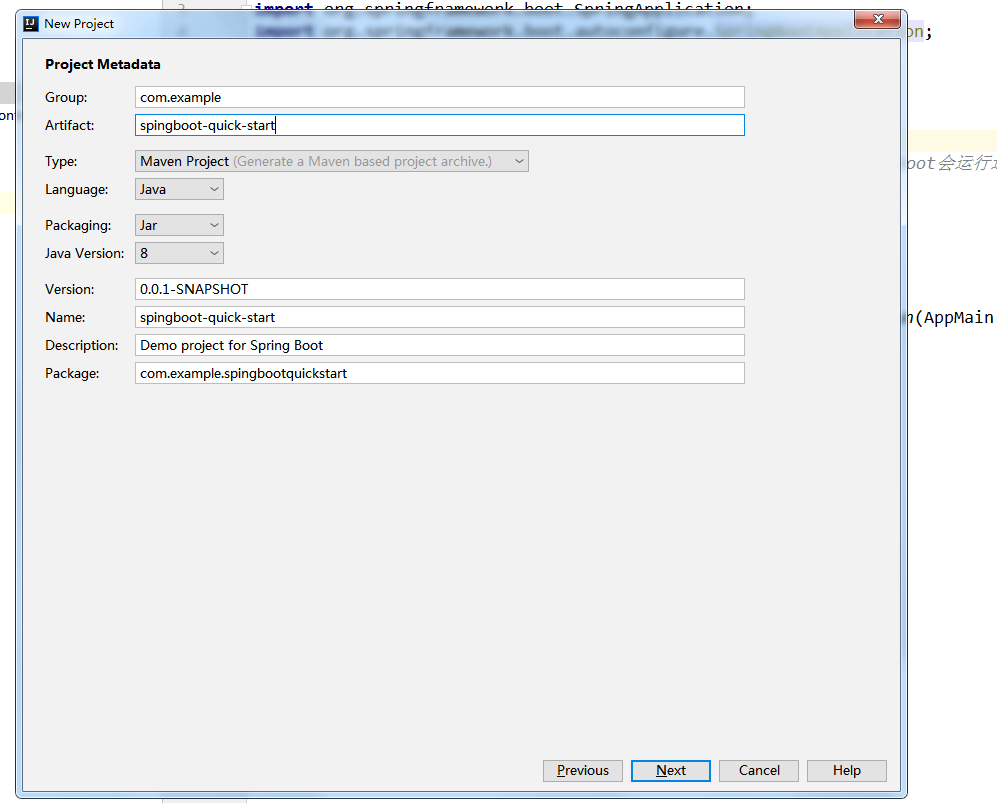
见 sping boot pdf

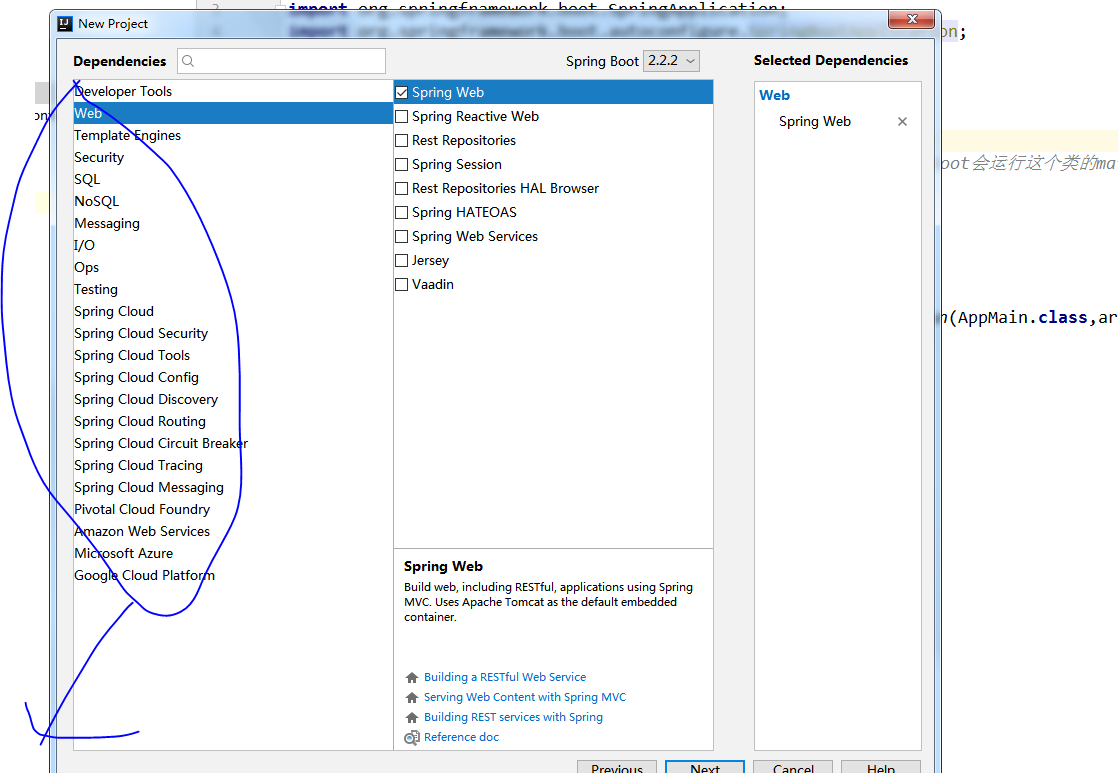
Poml jar包依赖

http://mvnrepository.com/

### 1.2快速配置







模块功能

## 构建系统

### 2.1

使用该设置，您还可以通过覆盖自己项目中的属性来覆盖各个依赖项。例如，要升级到另一个Spring Data发布系列，您可以将以下内容添加到您的pom.xml：

<properties>

<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version></properties>

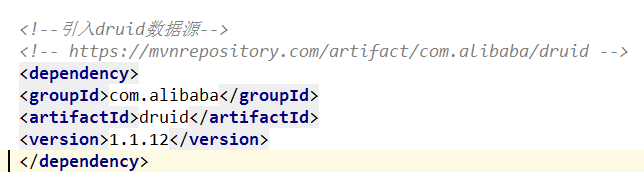
## 数据库

对于数据增删改是否成功的判断，我们一般将增删改的函数设成void类型，用自定义异常类来最终操作是否成功。但并不是绝对

### Jdbc---深入了解之后

jdbc是java与数据库连接的桥梁，用来连接数据库

### Druid

引入依赖：

Yml配置



# 初始化大小，最小，最大  
initialSize: 1  
minIdle: 3  
maxActive: 20  
# 配置获取连接等待超时的时间  
maxWait: 60000  
# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒  
timeBetweenEvictionRunsMillis: 60000  
# 配置一个连接在池中最小生存的时间，单位是毫秒  
minEvictableIdleTimeMillis: 30000  
validationQuery: select 'x'  
testWhileIdle: true  
testOnBorrow: false  
testOnReturn: false  
# 打开PSCache，并且指定每个连接上PSCache的大小  
poolPreparedStatements: true  
maxPoolPreparedStatementPerConnectionSize: 20  
# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙  
filters: stat,wall,slf4j  
# 通过connectProperties属性来打开mergeSql功能；慢SQL记录  
connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000

自定义数据源 DruidDataSource

<https://blog.csdn.net/wangmx1993328/article/details/81865153>

#### 2.1数据连接池

对于简单的数据库应用，由于数据库的访问不是很频繁，因此我们可以每次访问的时候连接数据库，不用的时候再断开数据库连接。如何我们的应用对于数据库的使用很复杂我们这样每次操作都关闭连接，会降低系统的性能。

数据连接池的作用就是为了解决上面问题，

数据库连接池负责分配，管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不用再重新建立一个

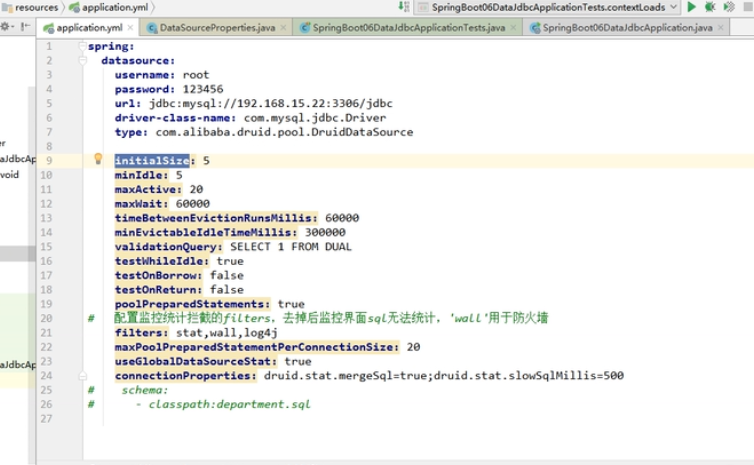
druid数据库连接池

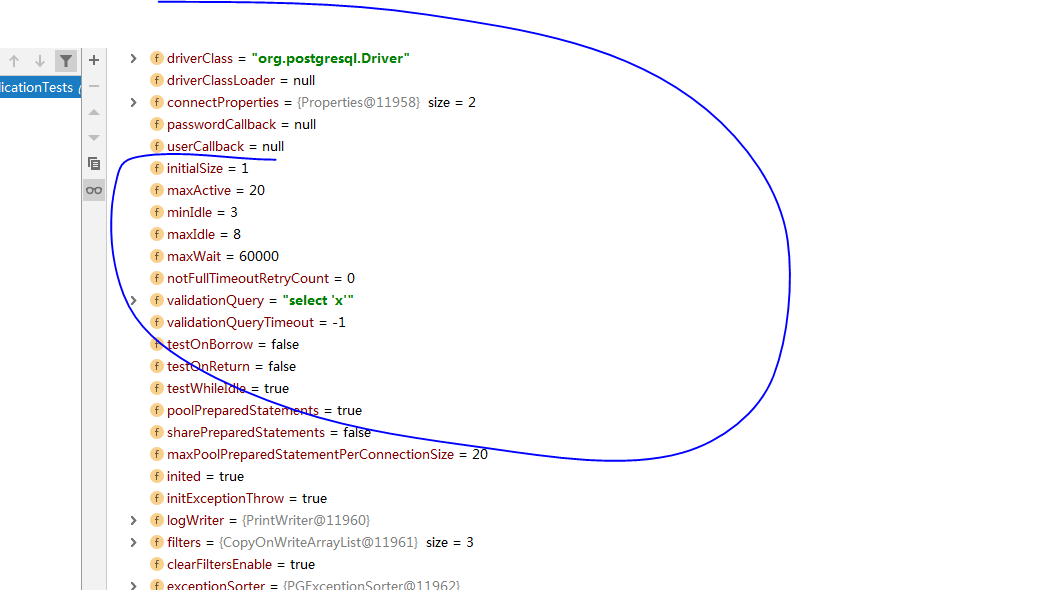
### Mybatis

#### 3.1引入依赖

*<!--引入mybits-->*<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>2.1.1</version>  
</dependency>

3.2





### springdata jpa

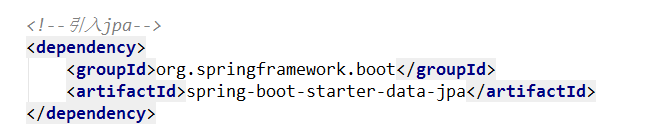
官方文档：

[https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/#jpa.query-methods](https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/" \l "jpa.query-methods)

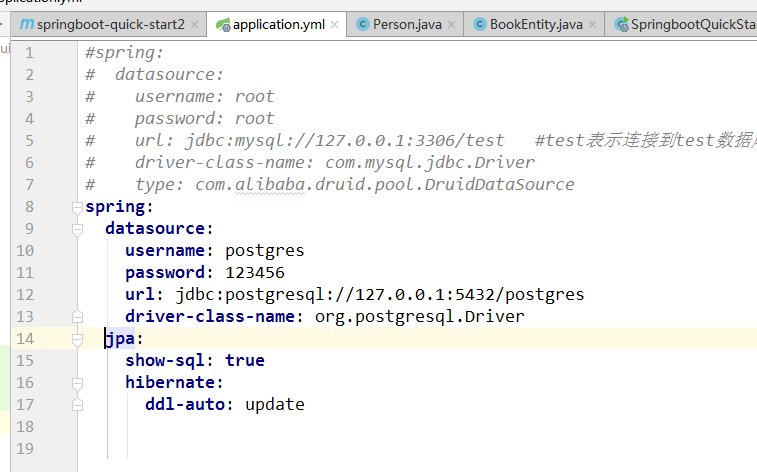
<https://blog.csdn.net/fly910905/article/details/78557110>

4.1配置与使用

#### 4.1引入jpaPom.xml

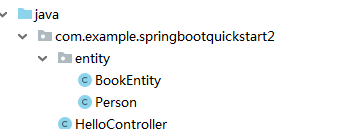


#### 4.2配置jpaYml

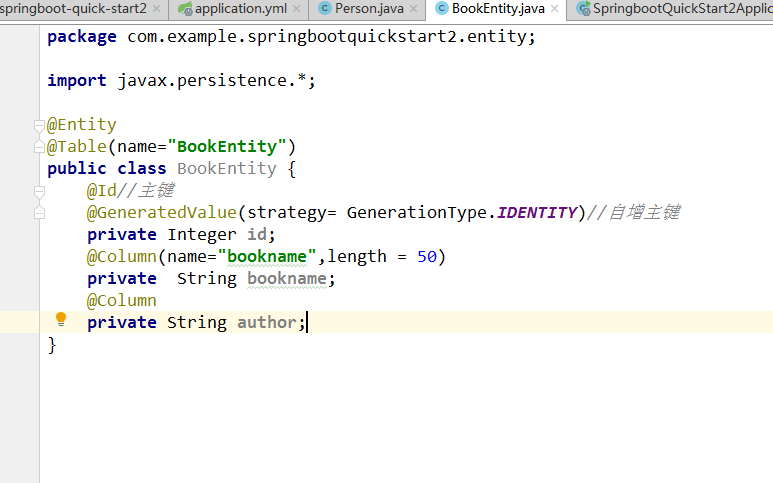


空格一定要对齐！！！！！！

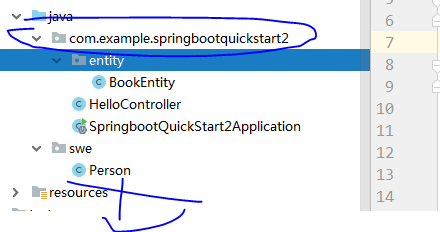
#### 4.3创建实体



自定义的jpa repository的使用



注意：当创建的实体类不在启动类所在包下或子目录下时



图中的person实体不在启动类所在包下或子目录下

在启动类添加 @EntityScan注解

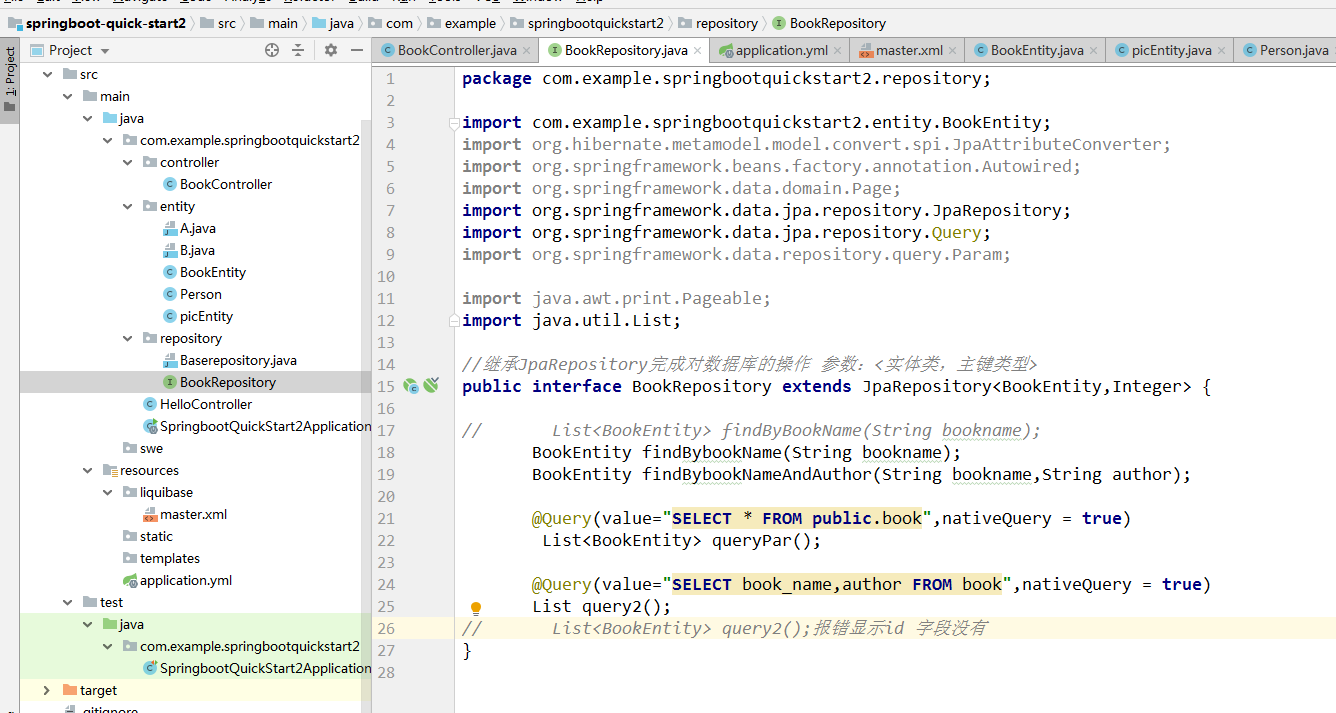
实体类注解使用说明

<https://www.kancloud.cn/cxr17618/springboot/435779>

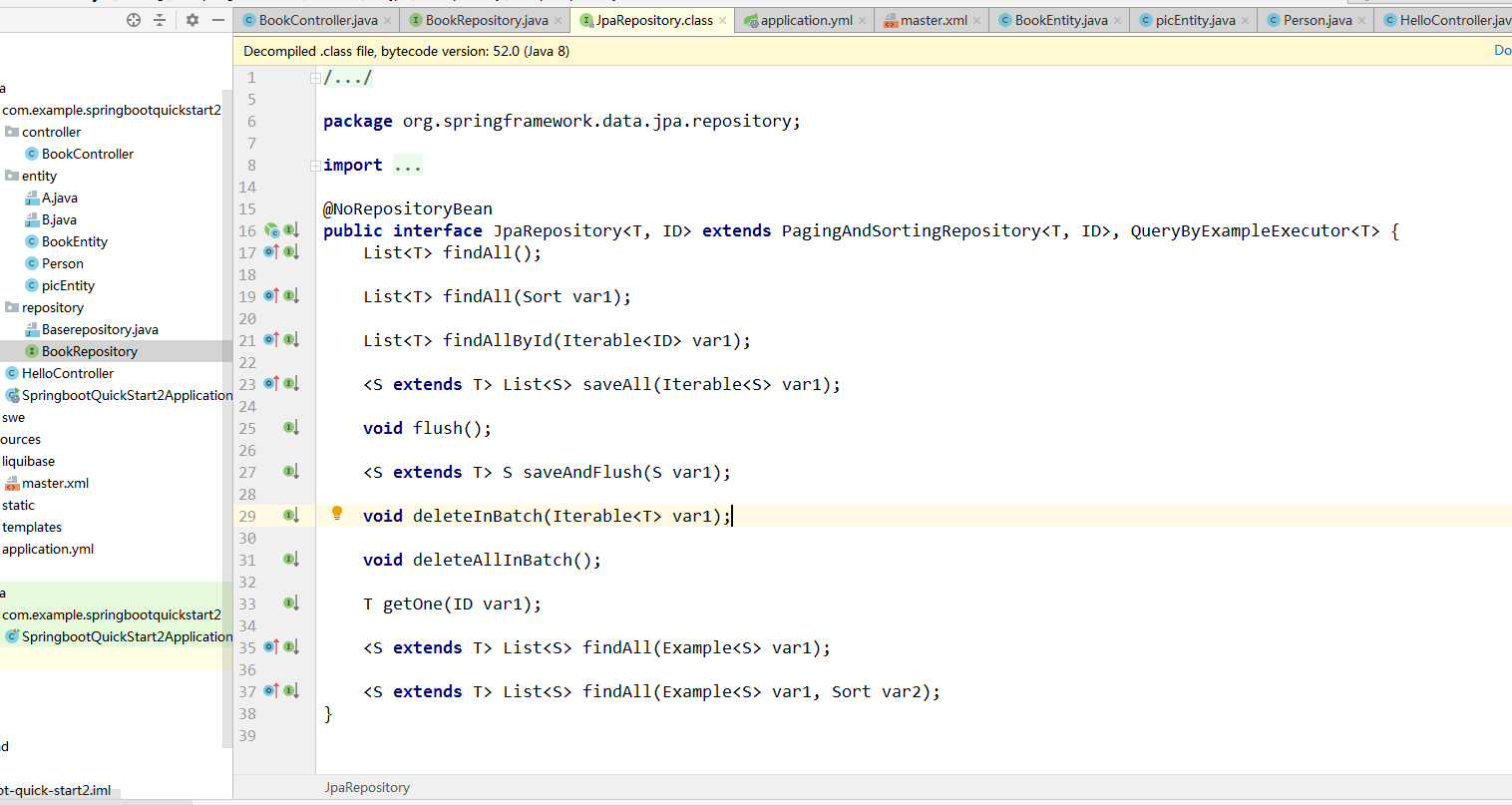
实体类添加字段注释？？？？



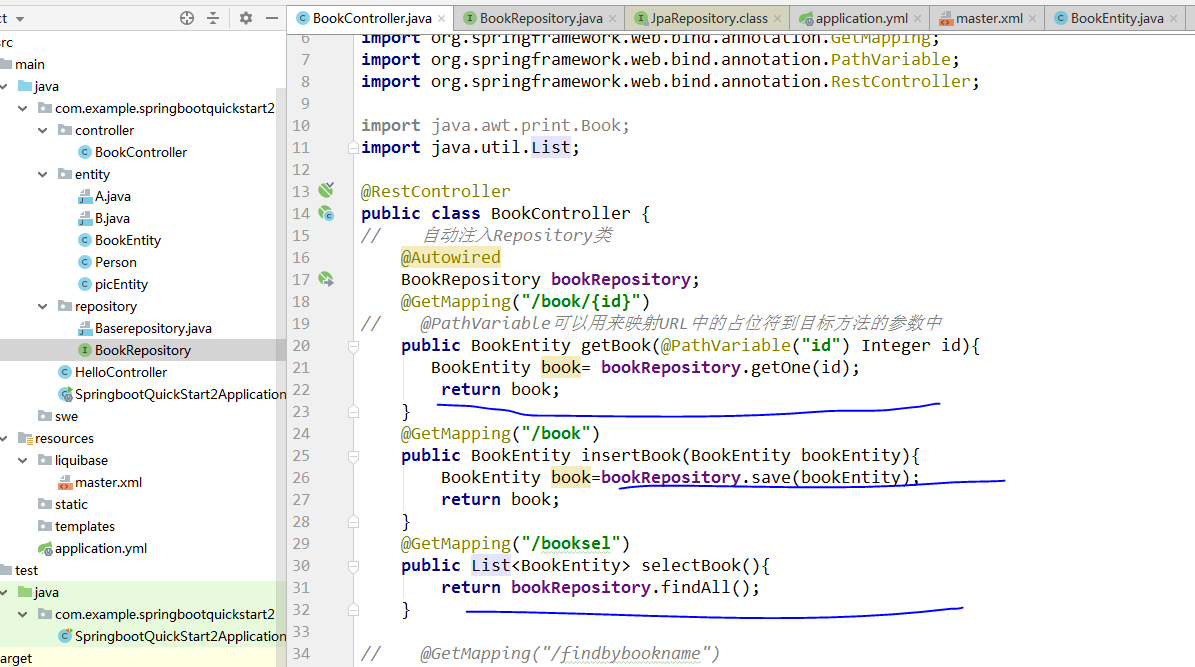
#### 4.4dao层创建repository仓库类



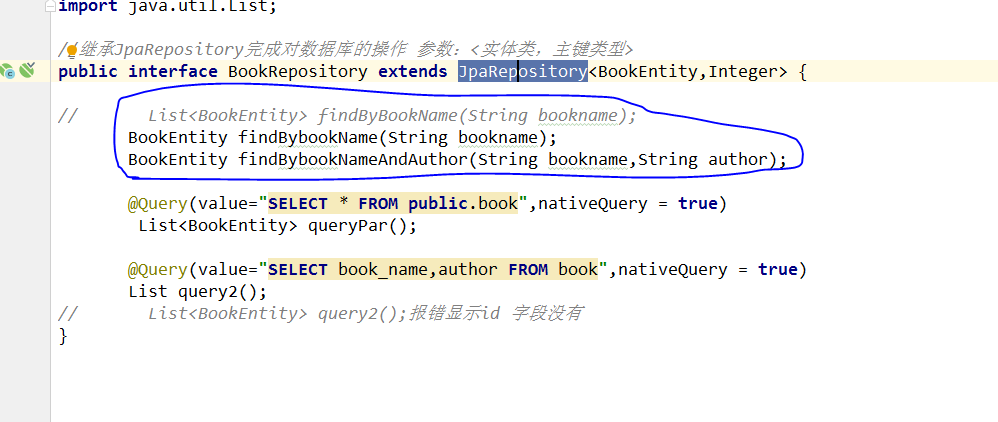
JpaRepository里面封装了一些方法，



##### 我们可以在使用的时候直接调用里面的接口



##### 还可以在repository仓库类中自定义方法



需要注意的是：自定义查询时findBybookName====>字段book\_name===实体属性==》bookName

findByAothor==>字段author

##### Jpa repository自定义方法命名规范

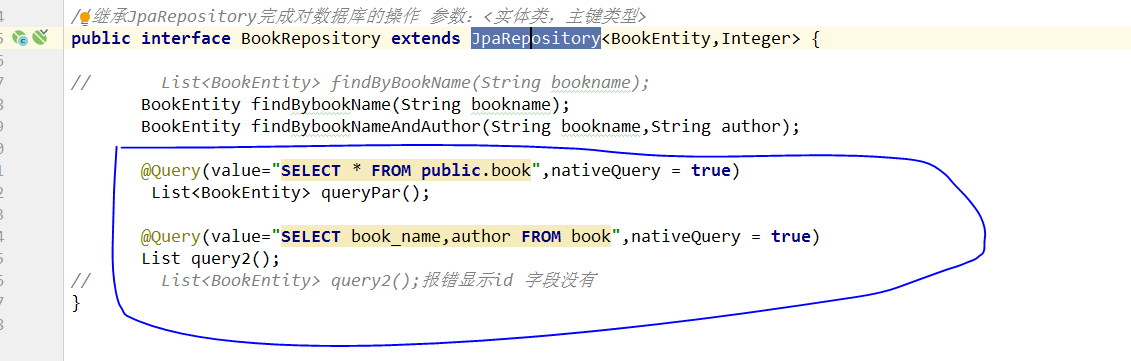
<https://blog.csdn.net/liyang_nash/article/details/80704089>

jpa默认只有save(Entity)方法，如果数据库中没有记录就新增，如果数据库中有记录就更新记录。

Jpa 增删改查使用方法

<https://www.cnblogs.com/lsl1314520/p/10598346.html>

##### [还可以利用@Query进行查询](https://www.cnblogs.com/powerwu/articles/10717970.html)



@Query使用方法

<https://blog.csdn.net/yingxiake/article/details/51016234>

<https://blog.csdn.net/qq_26434357/article/details/95513073>



### [JPA中@Query自定义的插入、更新、删除方法添加@Modifying注解和@Transactional注解？](https://www.cnblogs.com/wuhenzhidu/p/jpa.html)

参考：<https://www.cnblogs.com/wuhenzhidu/p/jpa.html>

 @Transactional 注解

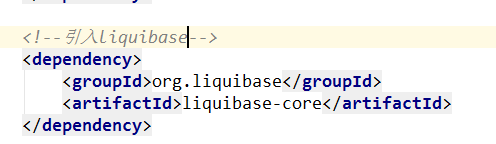
抛出异常之后，事务会自动回滚，数据不会插入到数据库。

### Liquibase

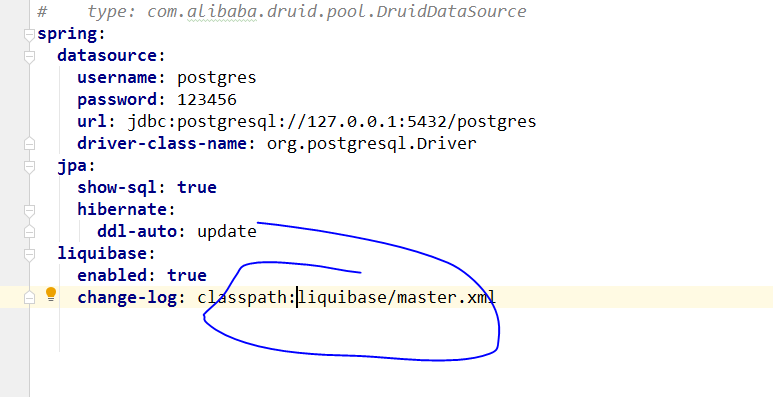
官方文档

<http://www.liquibase.org/documentation/index.html>

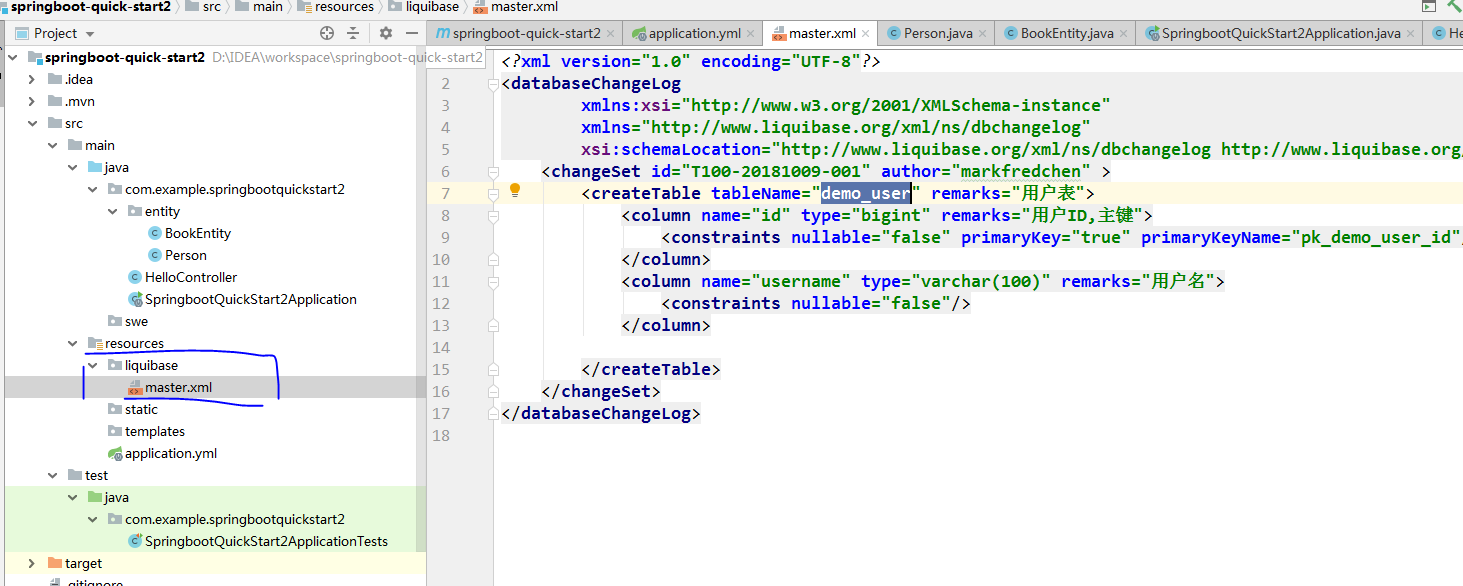
#### 配置：



Yml



标记部分没有空格



以xml文件生成数据表

运行时自动生成



#### 简介

Liquibase是一个用于数据库重构和迁移的开源工具，类似于php 中使用artisan

自动创建生成数据表，artisan将数据表的变化放在php文件中，liquibase放在XMl文件中。可实现代码的迁移和回滚。

Liquibase是一个用于跟踪、管理和应用数据库变化的开源数据库重构工具。它将所有数据库的变化保存在XML文件中，便于版本控制和项目部署升级。在快速搭建项目的JHipster框架中集成了该工具，使用该工具可以稳定、高效率地实现代码迁移与回滚。

本文使用Java Hipster快速自动生成的项目目录，简要说明liquibase使用方法。XML文件所在目录：resources\config\liquibase。

<https://blog.csdn.net/solocoder/article/details/99696715>

### 6.JDBC与 JdbcTemplate

JDBC（Java DataBase Connectivity,java数据库连接）是一种用于执行SQL语句的Java API，

可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。

DBCTemplate就是Spring对JDBC的封装,通俗点说就是Spring对jdbc的封装的模板

参数：

Spring Boot组件提供的全部配置属性。请注意，这些属性是否生效取决于对应

的组件是否声明为Spring应用程序上下文里的Bean ，为一个不生效的组件

设置属性是没有用的。

liquibase.change-log

变更日志配置路径。 （默认值： classpath:/db/changelog/db.changelog-master.

yaml 。 ）

liquibase.check-change-log-location

检查变更日志位置是否存在。 （默认值： true 。 ）

liquibase.contexts

要使用的运行时上下文列表，用逗号分隔。

liquibase.default-schema

默认的数据库Schema。

liquibase.drop-first

先删除数据库Schema。 （默认值： false 。 ）

liquibase.enabled

开启Liquibase支持。 （默认值： true 。 ）

liquibase.password

待迁移数据库的登录密码。

liquibase.url

待迁移数据库的JDBC URL。如果没有设置，就使用配置的主数据源。

liquibase.user

待迁移数据库的登录用户。

## 日志

### 4.1配置

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring‐boot‐starter</**artifactId**>  
</**dependency**>  
 *<!--引入日志-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-logging</**artifactId**>  
</**dependency**>



### 4.2使用

//日志的级别；

//由低到高 trace<debug<info<warn<error

//可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效

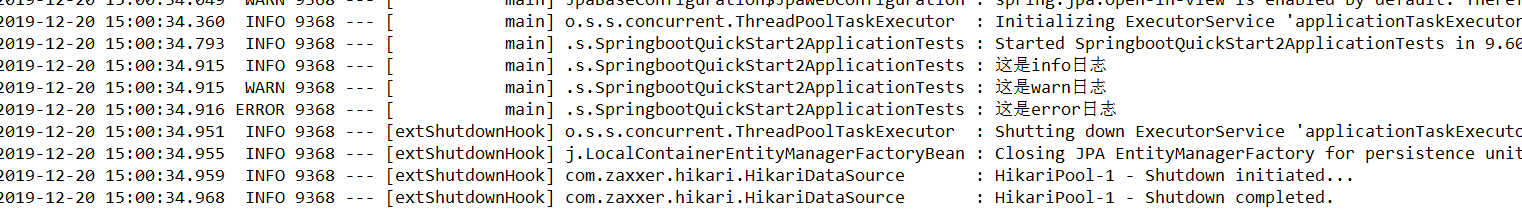
Logger logger= LoggerFactory.*getLogger*(getClass());  
logger.trace(**"这是trace日志"**);  
logger.debug(**"这是debug日志"**);

//SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的级别；root

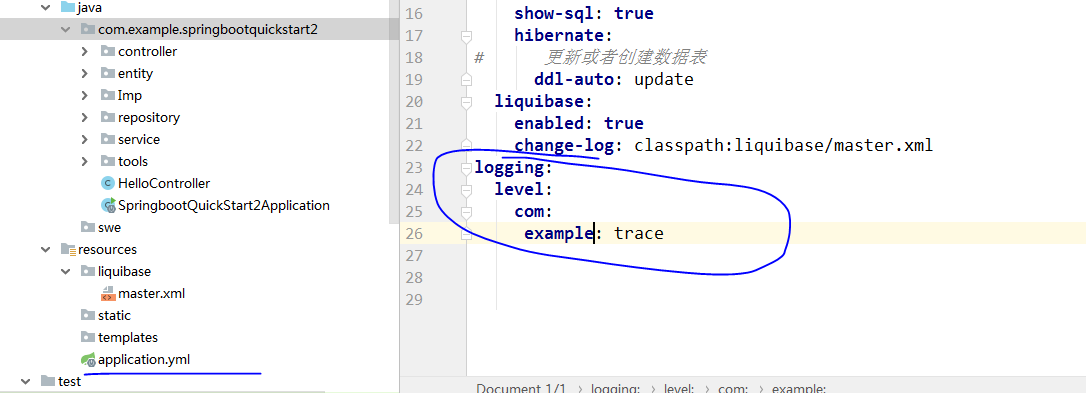
级别

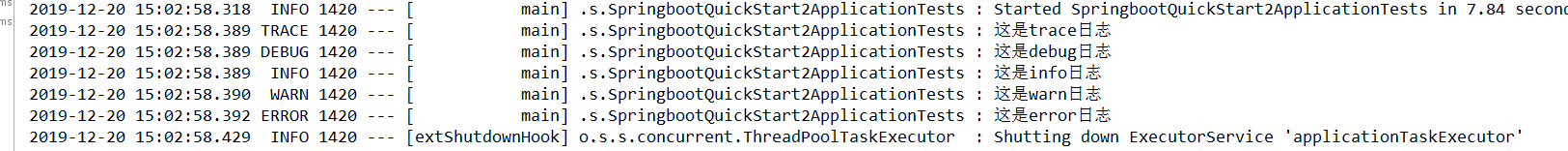
控制台只输出info 及以上日志

logger.info(**"这是info日志"**);  
logger.warn(**"这是warn日志"**);  
logger.error(**"这是error日志"**);



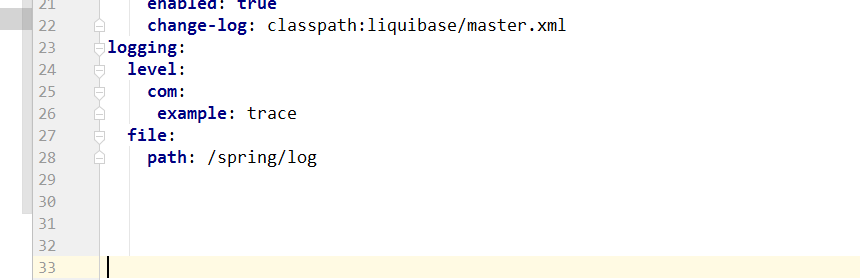
修改日志级别

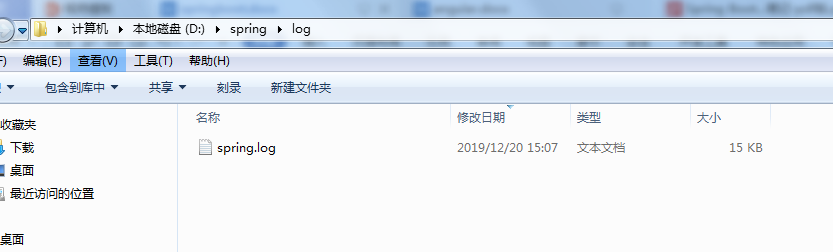




指定日志文件路径

/spring/log 项目根路径下的文件





### 4.3修改日志配置

logging.level.com.atguigu=trace

#logging.path=

# 不指定路径在当前项目下生成springboot.log日志

# 可以指定完整的路径；

#logging.file=G:/springboot.log

# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件 logging.path=/spring/log

## 五．Docker

账号：root  
密码：123456

### 5.1介绍

Docker让开发者将应用或依赖包打包到一个轻量级，可移植的容器中，然后发布到任何流行的Linux机器上。

简单的说就是我在一台windows上做了mysql 等一些配置，想把软件安在linux上，如果不用docker的话就是要在每一台linux上重新配置一遍环境。而Docker类似于镜像，只需要发布Docker就可以直接在linux上安装配置了不需要重新配置。

### 使用客户端连接linux服务器进行命令操作；

SmarTTY

### 5.2虚拟机安装配置

参考

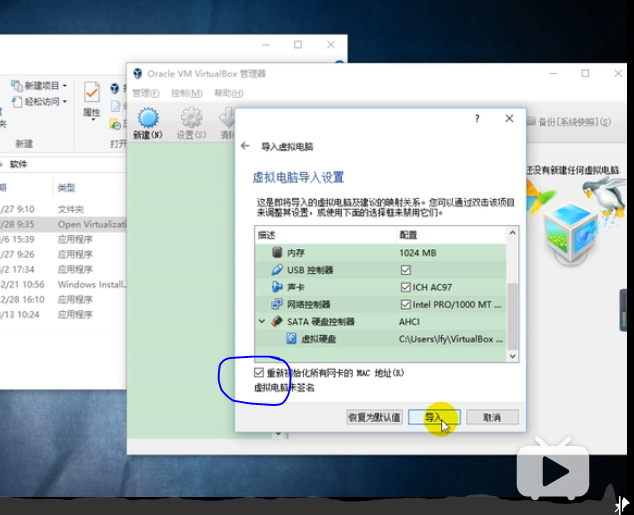
<https://www.cnblogs.com/Zeng02/p/11637847.html>

5.21安装linux虚拟机 VirtualBox

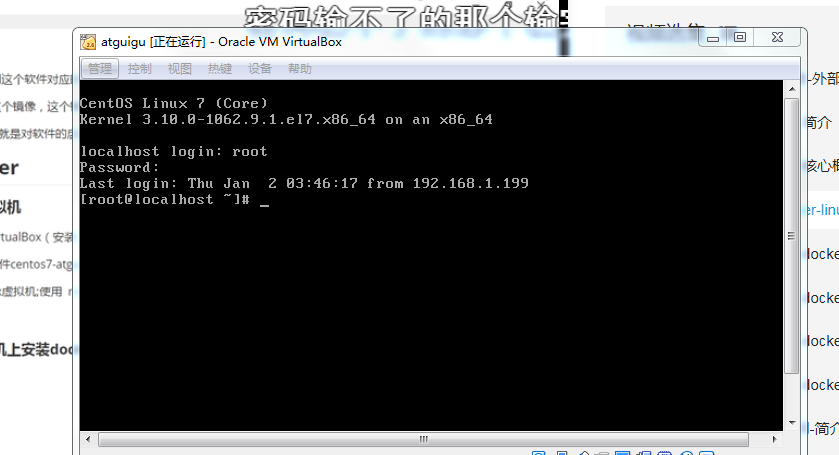
**1.点击管理---》导入虚拟电脑**

导入linux系统

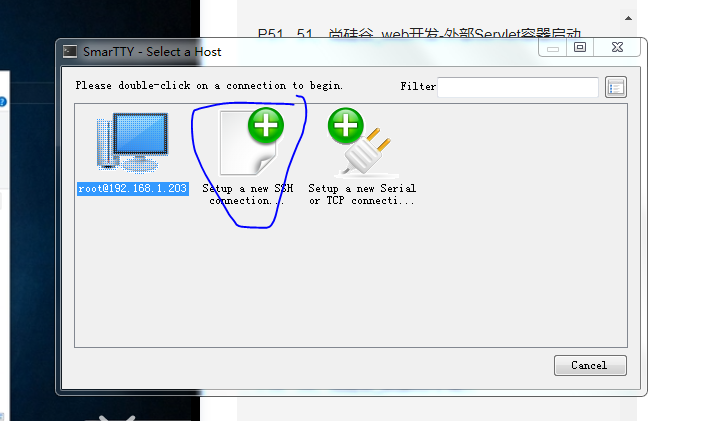


**2.打钩重新初始化所有网卡地址**

1. **输入账号密码登录**

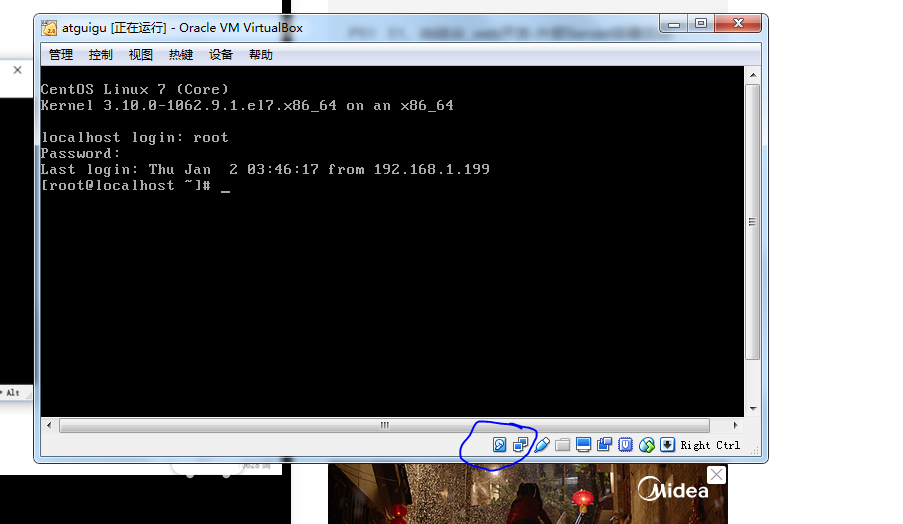


1. **使用linux客户端---SmarTTY**

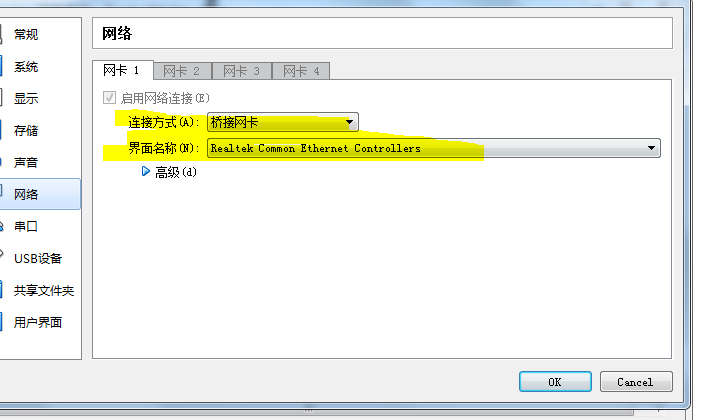


1. **设置虚拟机网络**

在virtualBox中



右键点击圈中---》网络

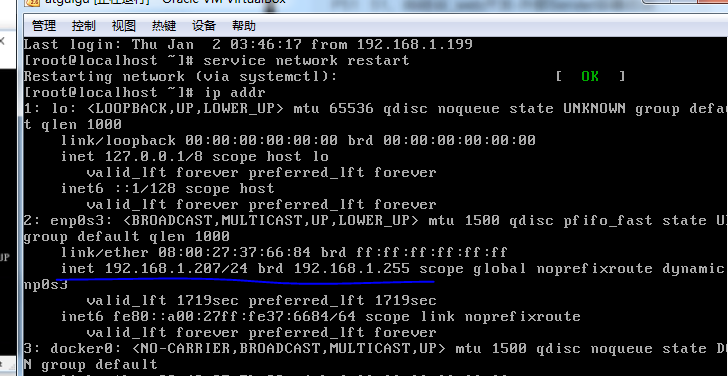


连接方式为桥接网卡

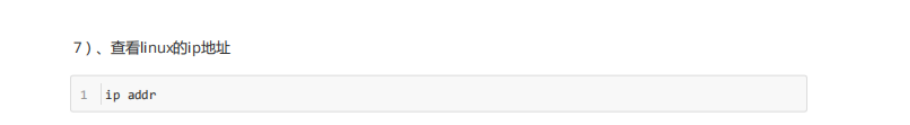
界面名称有线为图中所示

无线网络为无线网络的选择 关键字样 wireless

1. **service network restart重启网卡**
2. **输入ip addr 查看ip**







### 5.3在linux虚拟机安装Docker

检查内核版本必须3.10以上

Uname -r

1. 安装docker

Yum install docker

1. 输入y确认安装
2. 启动docker

systemctl start docker

docker -v 查看版本

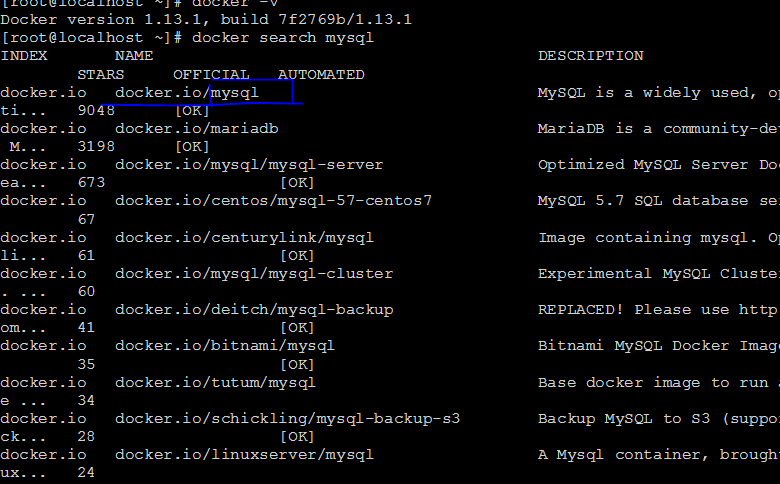
1. 设置开机自启动

systemctl enable docker

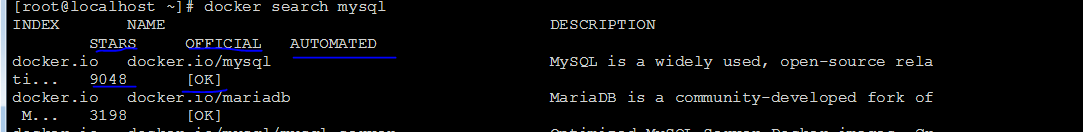
Systemctl stop docker 停止docker

### 5.4docker拉取镜像

1. **docker search 命令搜索镜像**



圈中部分就是镜像名称



Stars关注人数

Official:是否为官方

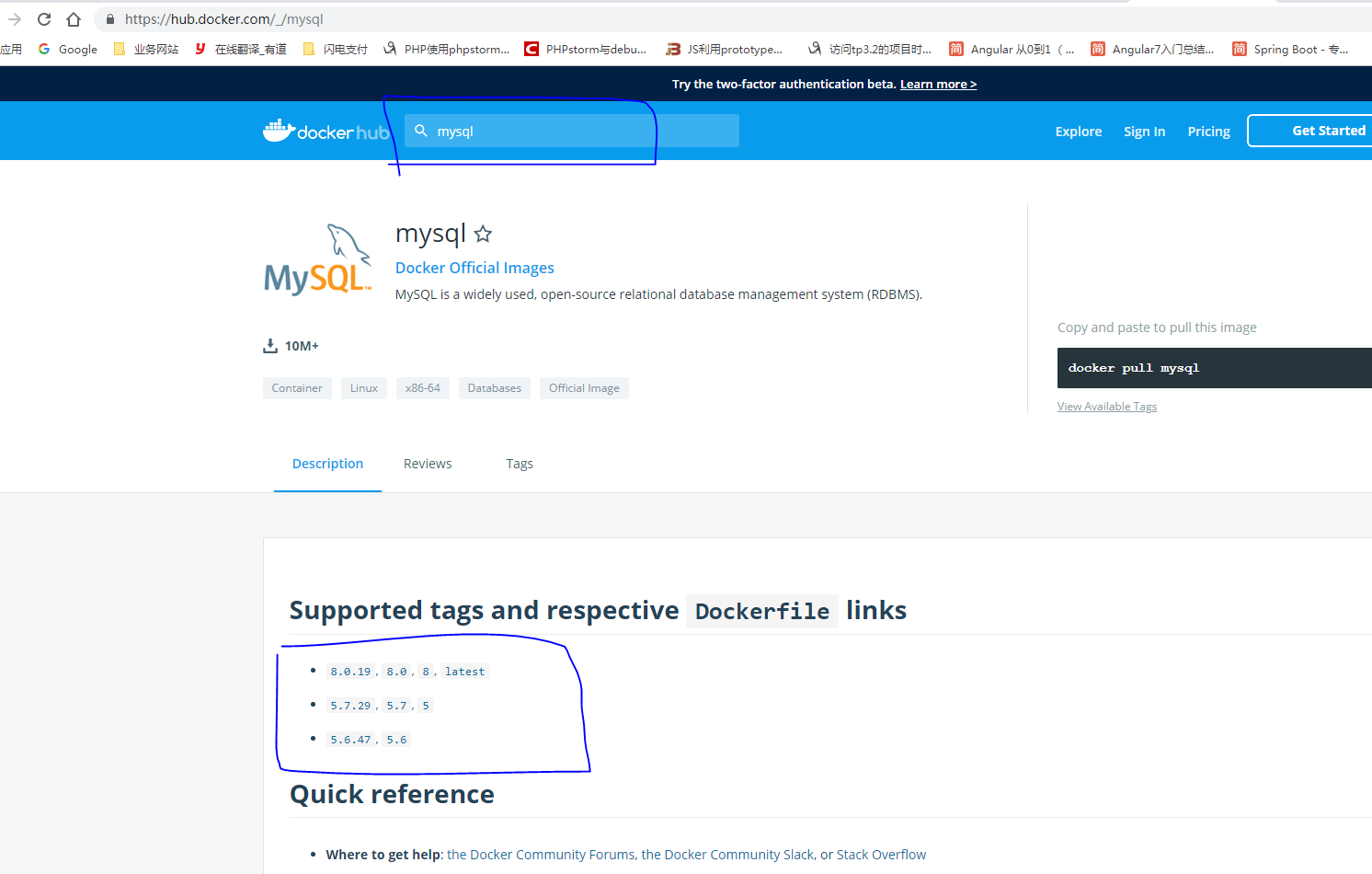
AtoMated:自动配置 ok

Name:镜像名称

docker pull mysql 下载镜像（默认下载最新版本）

2.Docker的镜像仓库网站：搜索镜像

<https://hub.docker.com/>



圈中为版本号

在SmarTTY中

输入docker pull mysql:8.019 下载具体版本

docker run --name mysql01 -d mysql 运行mysql容器

### 5.5docker 安装mysql

常见的镜像操作：

Docker运行一个镜像会产生一个容器

运行mysql

docker run --name mysql01 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql

停止容器

Docker stop 容器id

删除容器

Docker rm 容器id

docker ps 查看运行中的容器

docker ps -a 查看所有容器

docker logs container\_id（容器id ）



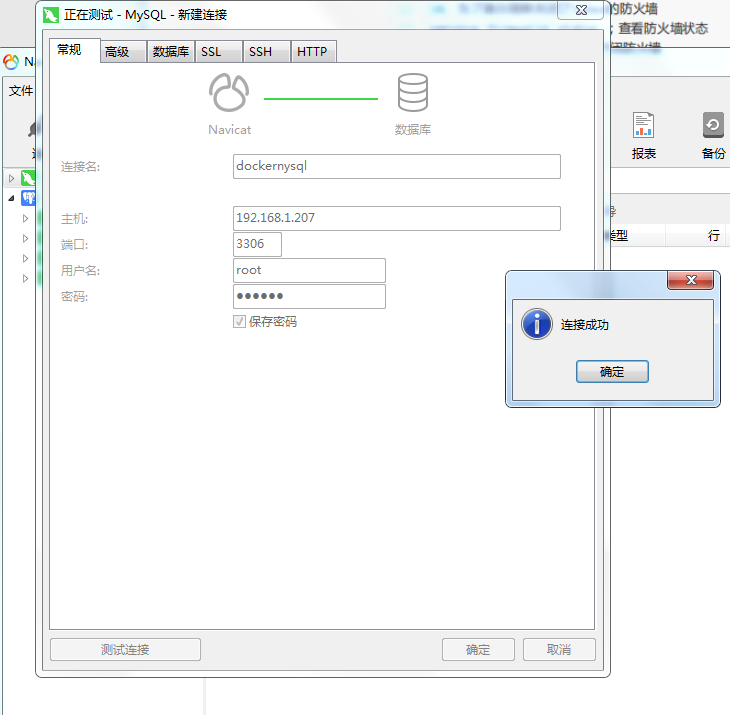
### 5.6navicat配置docker安装的mysql

开放端口运行mysql

docker run -p 3306:3306 --name mysql01 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql

运行postgresql

docker run -p 5432:5432--name postgresql01-e POSTGRES\_PASSWORD=mysecretpassword -d postgres



### navicat安装激活

参考：

<https://www.52pojie.cn/thread-952490-1-1.html>

## Security登录认证

官网：

<https://docs.spring.io/spring-security/site/docs/5.2.2.BUILD-SNAPSHOT/reference/htmlsingle/#prerequisites>

### 6.1引入依赖

*<!--引入spring security-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-security</**artifactId**>  
</**dependency**>

## 6.2编写springsecurity配置类

## 七．Spring boot mvc 的理解

Service

## 拦截器

拦截相关请求

### 8.1创建拦截器



### 8.2注入拦截器



## @注解

### 9.0 bean的理解

个人理解，spring系列框架依托于ioc这个容器，bean就是代理这些类放到ioc中,注册的时候用@bean（之前用的xml）使用的时候就用自动注入的注解。（暂时的理解，深入理解后续补上）





### @Bean 注解全解析

随着SpringBoot的流行，基于注解式开发的热潮逐渐覆盖了基于XML纯配置的开发，而作为Spring中最核心的bean当然也能够使用注解的方式进行表示。所以本篇就来详细的讨论一下作为Spring中的Bean到底都有哪些用法。

### @Bean 基础声明

Spring的@Bean注解用于告诉方法，产生一个Bean对象，然后这个Bean对象交给Spring管理。产生这个Bean对象的方法Spring只会调用一次，随后这个Spring将会将这个Bean对象放在自己的IOC容器中。

SpringIOC 容器管理一个或者多个bean，这些bean都需要在@Configuration注解下进行创建，在一个方法上使用@Bean注解就表明这个方法需要交给Spring进行管理。

### 9.1@Component

把简单的java对象实例化到spring容器中

Pojo(简单的java对象)

### 9.2@Configuration

个人理解就是加上这个注解之后项目启动时自动解析你的配置

### 9.3@Cacheable

<https://www.cnblogs.com/MagicAsa/p/10762356.html>

原理

@Cacheable(cacheNames = {"emp"},keyGenerator = "myKeyGenerator",condition = "#a0>1",unless = "#a0==2")   
意思为 放在的缓存名称为emp中，key的生成方式为配置的myKeyGenerator类，当第一个参数大于1时候缓存，当第一个参数等于2时不缓存

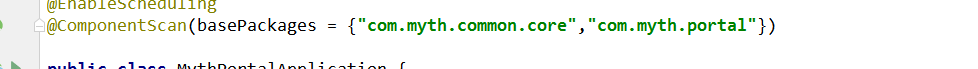
@Cacheable(value={"users"}, key="#user.id", condition="#user.id%2==0")



### 9.4@ComponentScan

@ComponentScan主要就是定义扫描的路径从中找出标识了需要装配的类自动装配到spring的bean容器中

使用方法：在启动类上加上这一注解并定义路径



然后将其他模块通过自动注解加载的类，放到这个启动类容器中

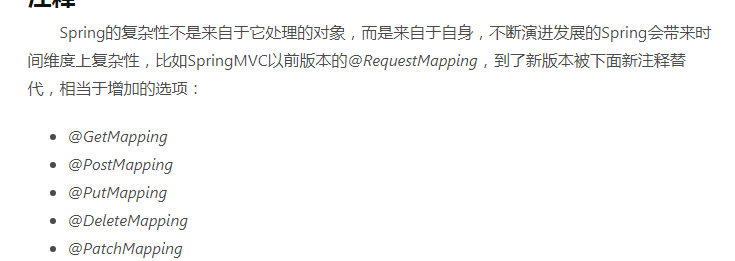
简单的说就是你可以使用其他模块里的东西了。

参考：<https://www.cnblogs.com/jpfss/p/11171655.html>

### 9.5@RequestMapping与@GetMapping和@PostMapping

@RequestMapping(value = "/get/{id}", method = RequestMethod.GET)

@GetMapping用于处理请求方法的GET类型，@ PostMapping用于处理请求方法的POST类型



### 9.6@SequenceGenerator

参考：<https://blog.csdn.net/langjian2012/article/details/38894195>

## sprinboot其他

### 10.1session使用

request.getSession().setAttribute(ConstantUtil.USER\_SESSION\_KEY, userEntity.getUserName());

request.getSession().getAttribute(ConstantUtil.USER\_SESSION\_KEY)

## 自动配置

## Bean

Bean是Spring框架中****最核心****的两个概念之一（另一个是****面向切面编程AOP****）

官方翻译：

在 Spring 中，构成应用程序****主干****并由****Spring IoC容器****管理的****对象****称为****bean****。bean是一个由Spring IoC容器实例化、组装和管理的对象

概念简单明了，我们提取处关键的信息：

1. bean是对象，一个或者多个不限定
2. bean由Spring中一个叫IoC的东西管理
3. 我们的应用程序由一个个bean构成

理解Spring Boot的自动配置先要理解@Conditional注解

@Bean: 将当前对象放入到Spring的IOC容器中来管理   
@Conditional：满足特定条件时才会创建一个Bean放入到IOC容器，SpringBoot就是利用这个特性进行自动配置的。

## 缓存---Redis

### 12.0springboot缓存

快速体验缓存

步骤：

1. 开启基于注解的缓存@EnableCaching
2. 标注缓存注解

@Cacheable

@CacheEvict

@Cacheput

Springboot默认使用ConcurrentMapCacheManager缓存

Redis中文文档

<http://www.redis.cn/documentation.html>

里面有一些命令



String,List(列表),Set(集合),Hash(散列),zSet(有序集合)  
\* stringRedisTemplate.opsForValue()[String(字符串)] //操作字符串  
\* stringRedisTemplate.opsForList()[List(列表)] //操作list  
\* stringRedisTemplate.opsForSet()[Set(集合)]   //操作set  
\* stringRedisTemplate.opsForHash()[Hash(散列)] //操作hash  
\* stringRedisTemplate.opsForZSet()[ZSet(有序集合)]//操作有序set

// ValueOperations<String, String> valueOperations = stringRedisTemplate.opsForValue();  
// stringRedisTemplate.opsForValue().append("msg","asda");  
// 等价于  
// redisTemplate.opsForValue().set("asdasd",set);

### 12.1安装配置

windows下载

<https://github.com/microsoftarchive/redis/releases>

**Windows下使用redisDesktop操作redis**

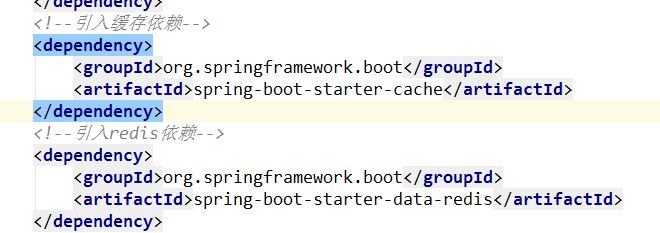
redis菜鸟教程

<https://www.runoob.com/redis/redis-install.html>

打开一个 **cmd** 窗口 使用 cd 命令切换目录到 **C:\redis** 运行：

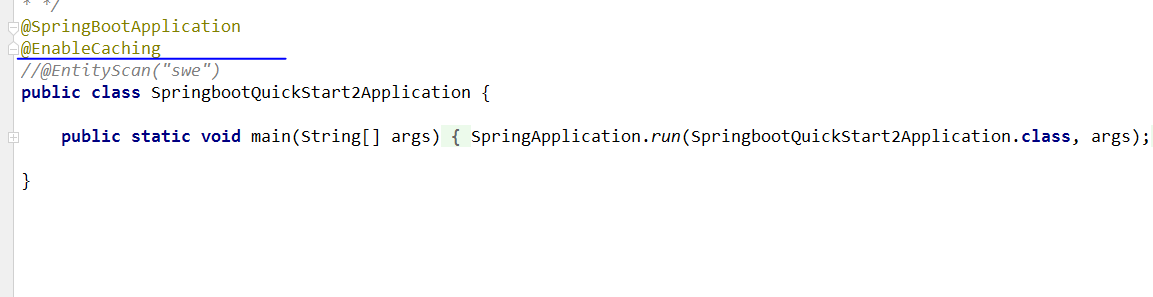
redis-server.exe redis.windows.conf

引入依赖

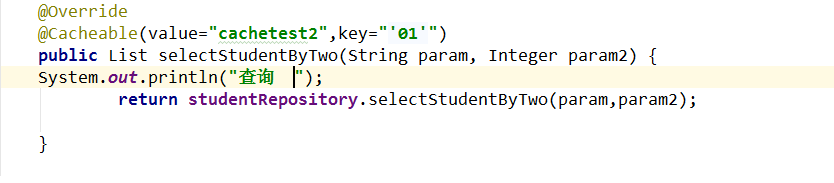




**开启缓存**



**在方法上使用@Cacheable注解**



设置序列化



@Bean  
**public** CacheManager cacheManager(RedisConnectionFactory factory) {  
 RedisSerializer<String> redisSerializer = **new** StringRedisSerializer();  
 Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = **new** Jackson2JsonRedisSerializer(Object.**class**);  
  
 *//解决查询缓存转换异常的问题* ObjectMapper om = **new** ObjectMapper();  
 om.setVisibility(PropertyAccessor.***ALL***, JsonAutoDetect.Visibility.***ANY***);  
 om.enableDefaultTyping(ObjectMapper.DefaultTyping.***NON\_FINAL***);  
 jackson2JsonRedisSerializer.setObjectMapper(om);  
  
 *// 配置序列化（解决乱码的问题）* RedisCacheConfiguration config = RedisCacheConfiguration.*defaultCacheConfig*()  
 .entryTtl(**timeToLive**)  
 .serializeKeysWith(RedisSerializationContext.SerializationPair.*fromSerializer*(redisSerializer))  
 .serializeValuesWith(RedisSerializationContext.SerializationPair.*fromSerializer*(jackson2JsonRedisSerializer))  
 .disableCachingNullValues();  
  
 RedisCacheManager cacheManager = RedisCacheManager.*builder*(factory)  
 .cacheDefaults(config)  
 .build();  
 **return** cacheManager;  
}

### 12.2使用

序列化

<https://www.cnblogs.com/wslook/p/9401134.html>

## 十三．Springboot 消息队列

理解与使用场景

参考：

<https://www.jianshu.com/p/36a7775b04ec>

### 什么是消息队列？

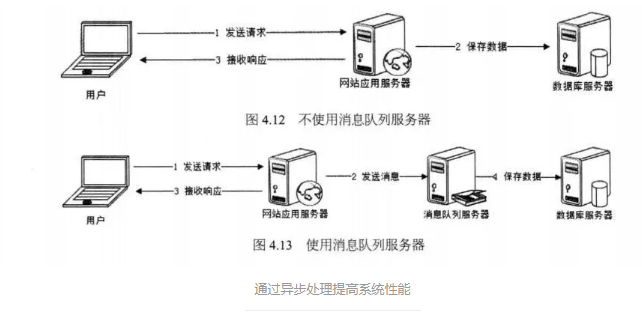
在数据结构中我们了解到队列先进先出，而消息队列我们可以理解成是一个存放消息的容积，在我们需要时从容器中取出使用。

### 为什么要使用消息队列？

1. 通过异步提高系统性能。

场景：

秒杀活动，抢票，促销活动。

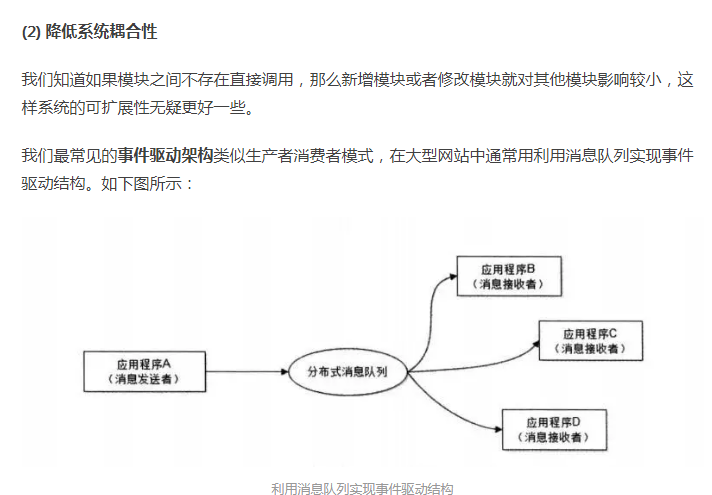


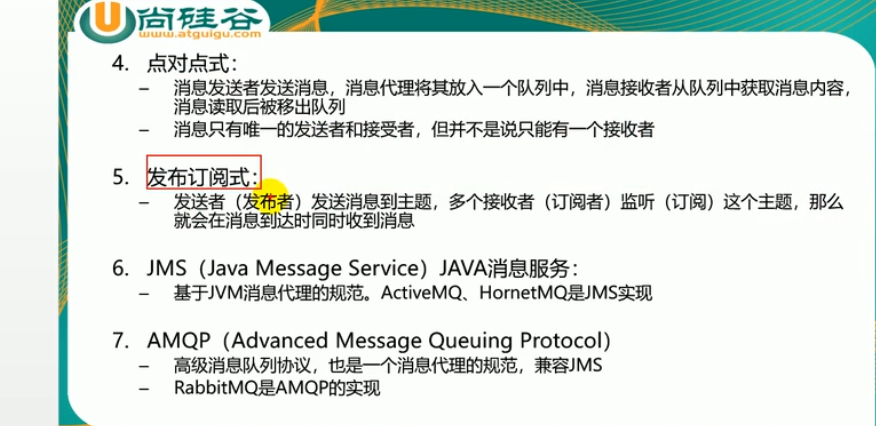
在这个图中我们可以看到，

不使用消息队列的时候用户的请求直接写入数据库，当在像秒杀这种千万次的请求下（高并发），服务器响应速度会变慢

使用消息队列之后，用户发出的请求会先到消息队列服务器，然后消息队列服务器立刻返回，然后消息队列的消费者进程从消息队列中获取数据异步写入数据库。由于消息队列服务器处理速度比数据库快，因此响应速度得到提升。

2.降低系统耦合性





### JMS和AMQP 消息服务规范

基于JVM消息代理的规范。ActiveMQ,HornetMQ是JMS实现

AMQP是高级消息队列协议，也是一个消息代理的规范。兼容JMS

RabbitMQ是AMQP的实现

**对比**



### 整合RabbitMQ

#### Docker安装启动rabbitmq

**安装**：docker pull rabbitmq:3-management

参考：

<https://blog.csdn.net/hsg_happyLearning/article/details/82314082>

**启动：**

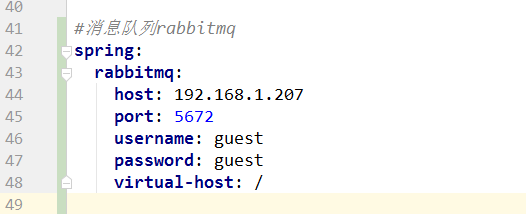
# -d是后台启动；-p是端口映射，第一个端口是linux服务器的端口，第二个端口是docker容器的端口，将linux服务器的端口与创建的rabbitmq容器的端口进行映射，5672是rabbitmq默认的端口，15672是rabbitmq web服务的端口；--name是给容器取名字；df80af9ca0c9是rabbitmq镜像的id

docker run -d -p 5672:5672 -p 15672:15672 --name myRabbitMQ df80af9ca0c9

引用依赖

<!--引入消息队列rabbitmq-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>

#### 配置



配置类介绍：

队列关联绑定交换器

\* RabbitAutoConfiguration为rabbitmq的自动配置类  
\*有自动配置了连接工厂connectionfactory  
\* RabbitProperties封装了rabbitmq的所有配置  
\*RabbitTemplate给rabbitmq发送和接收消息

\* AmqpAdmin：rabbitmq系统管理功能组件：创建删除Queue,Exchange,Binding

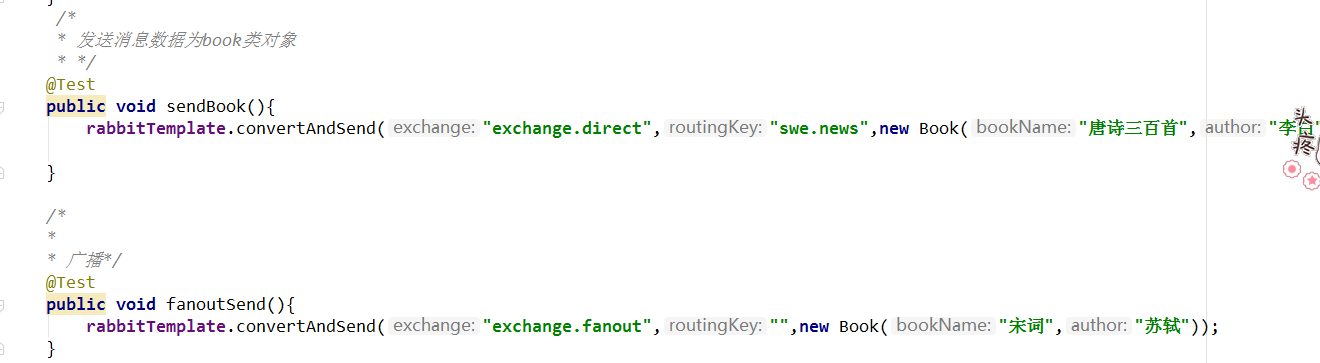
使用程序创建消息队列

@RabbitListener监听

开启基于注解的模式@EnableRabbit  
**测试类测试消息的发送与接受：**







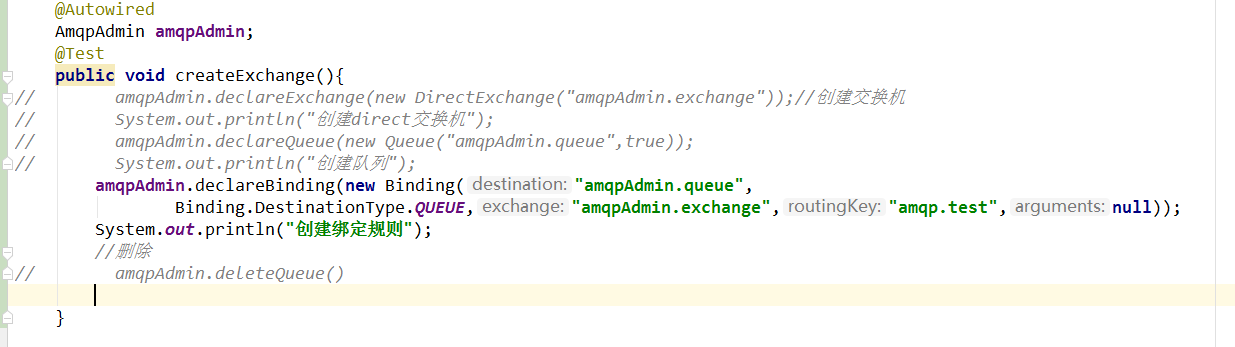
**service监听消息队列**



在启动类中加上

@EnableRabbit *//开启基于注解的rabbit模式*

**测试类使用**AmqpAdmin**测试创建删除exchange,queue,bind**



### 集成kafka消息队列

安装[wurstmeister / kafka-docker](https://www.github.com/wurstmeister/kafka-docker" \t "https://hub.docker.com/r/wurstmeister/_blank)文档：

<https://hub.docker.com/r/wurstmeister/kafka>

安装 docker composer

<https://docs.docker.com/compose/install/>

### 新建修改**docker-compose.yml**

### Vi **docker-compose.yml**

version: '2'

services:

zookeeper:

image: wurstmeister/zookeeper ## 镜像

ports:

- "2181:2181" ## 对外暴露的端口号

kafka:

image: wurstmeister/kafka ## 镜像

volumes:

- /etc/localtime:/etc/localtime ## 挂载位置（kafka镜像和宿主机器之间时间保持一直）

ports:

- "9092:9092"

environment:

KAFKA\_ADVERTISED\_HOST\_NAME: 182.92.217.111 ## 修改:宿主机IP

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181 ## 卡夫卡运行是基于zookeeper的

KAFKA\_ADVERTISED\_PORT: 9092

KAFKA\_LOG\_RETENTION\_HOURS: 120

KAFKA\_MESSAGE\_MAX\_BYTES: 10000000

KAFKA\_REPLICA\_FETCH\_MAX\_BYTES: 10000000

KAFKA\_GROUP\_MAX\_SESSION\_TIMEOUT\_MS: 60000

KAFKA\_NUM\_PARTITIONS: 3

KAFKA\_DELETE\_RETENTION\_MS: 1000

kafka-manager:

image: sheepkiller/kafka-manager ## 镜像：开源的web管理kafka集群的界面

environment:

ZK\_HOSTS: 182.92.217.111 ## 修改:宿主机IP

ports:

- "9001:9000" ## 暴露端口

docker pull sheepkiller/kafka-manager

* 启动集群：docker-compose up -d

访问kafka管理界面

[http://192.168.1.207:9001](http://192.168.1.207:9001/)

## 十四．Springboot与检索

### Elasticsearch

Elasticsearch默认占用2g的内存

Docker镜像：

**拉取命令**：docker pull elasticsearch

**开发模式运行：**

创建用户定义的网络（用于连接到连接到同一网络的其他服务（例如，Kibana））：

$ docker network create somenetwork

**启动命令**：

docker run -d --name elasticsearch --net somenetwork -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" elasticsearch:7.6.0

**安装成功标志**：

Ip:9200浏览器访问有json数据响应

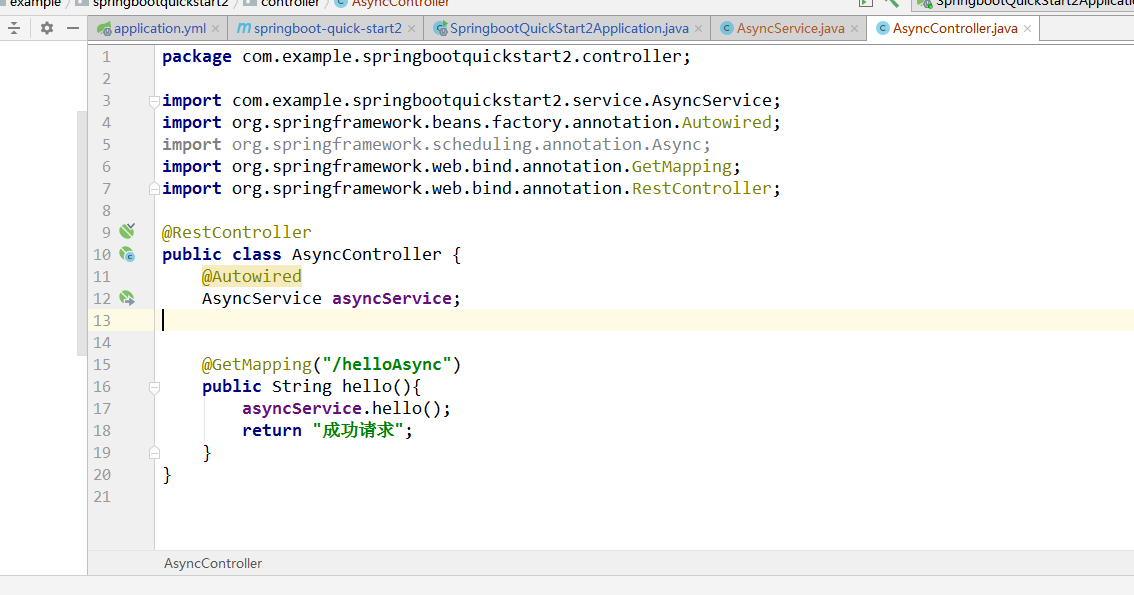
## Sprinboot与任务

### 异步任务

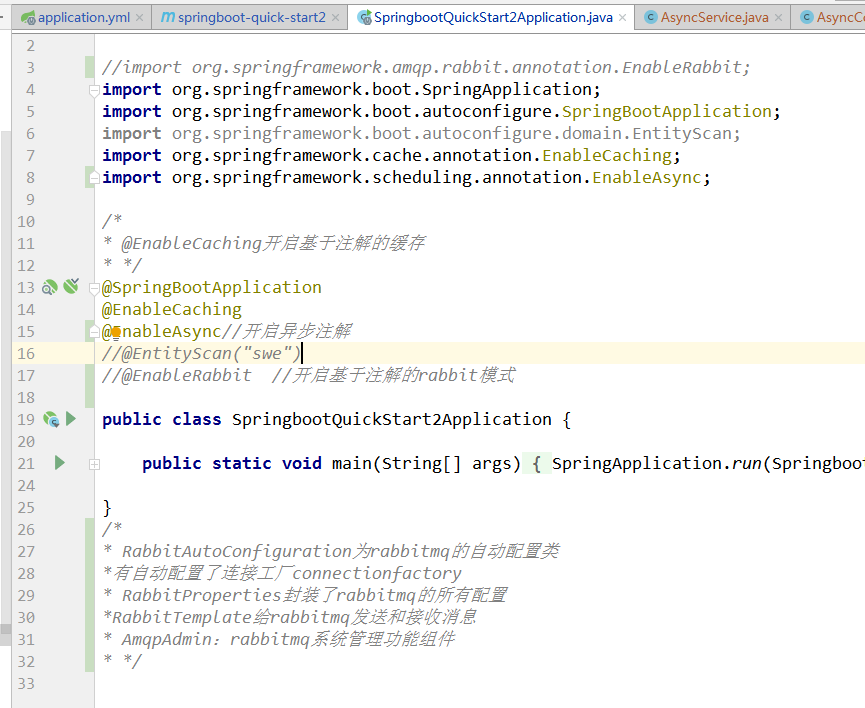
@EnableAsync @Aysnc注解

在方法上加上@Aysnc注解，告诉spring这是个异步方法，spring自己开一个线程池进行调用





然后再启动类加上@EnableAsync



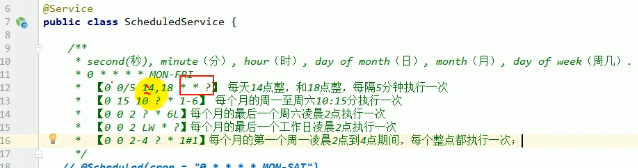
### 定时任务

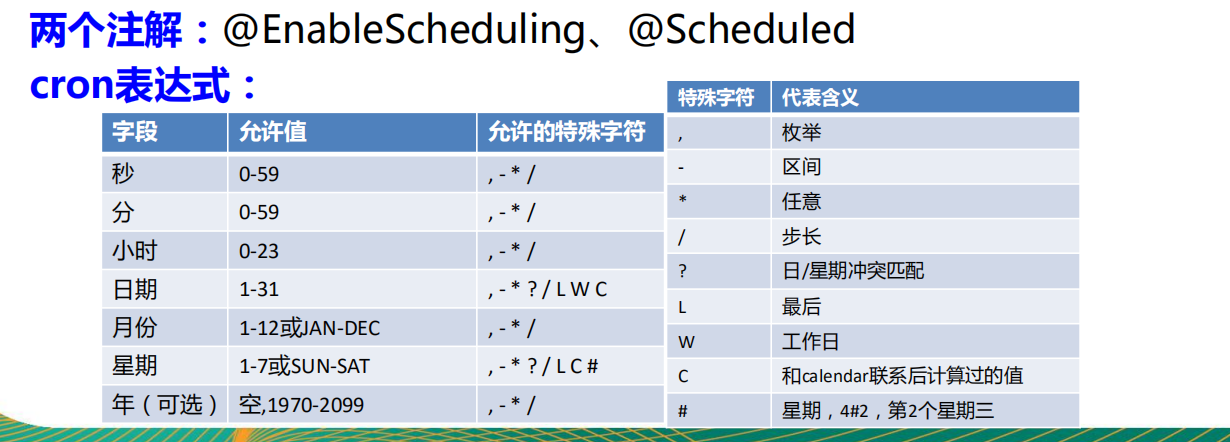
**开发场景：固定时间进行日志分析，数据分析**

**@EnableScheduling @Scheduled**

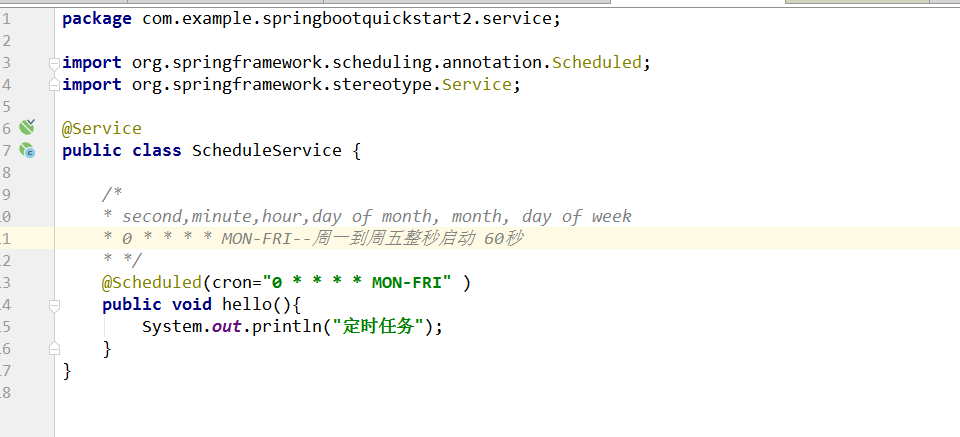
**@Scheduled cron表达式**

\* 枚举 0，1，2，3，4秒运行  
\* \*/  
@Scheduled(cron="0，1，2，3，4 \* \* \* \* MON-FRI" )  
//指定区间@Scheduled(cron="0-4 \* \* \* \* MON-FRI" )  
//步长@Scheduled(cron="0-4 \* \* \* \* MON-FRI" )每4秒齿执行一次

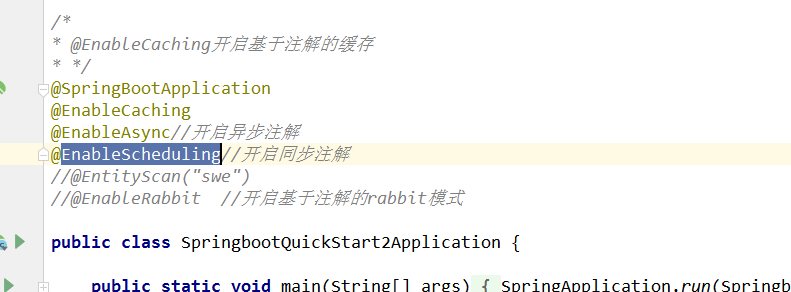




*/\*  
\* second,minute,hour,day of month, month, day of week  
\* 0 \* \* \* \* MON-FRI--周一到周五整秒启动 60秒  
\* \*/*



启动类开启注解



### 邮件任务

**引入依赖**

<dependency>

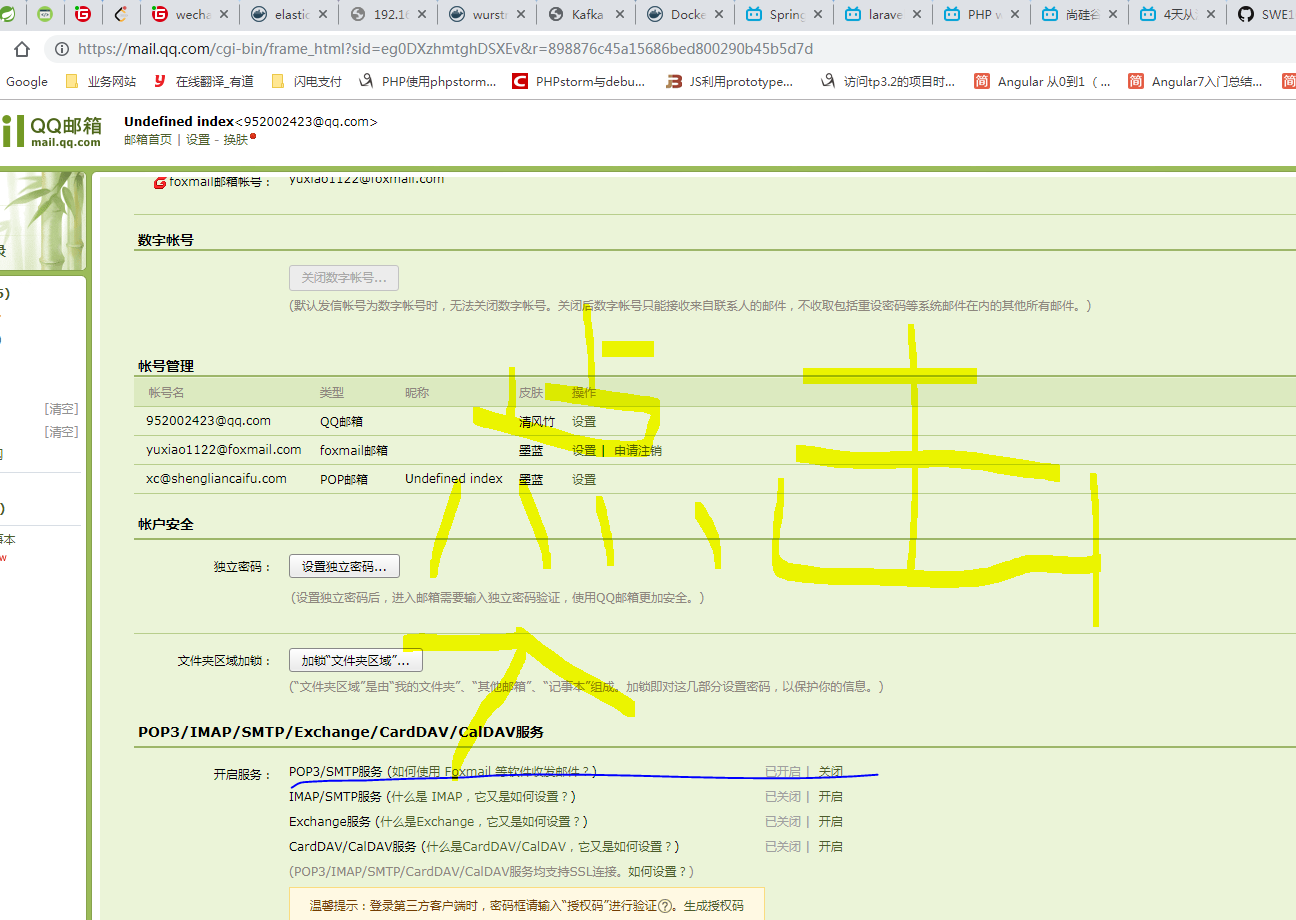
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-mail</artifactId></dependency>

**邮件授权码:huzqxfirjzgfbfhd**

**配置：**





Host为：

点击上面的链接

**使用**：

@Autowired  
JavaMailSenderImpl javaMailSender;  
@Test  
public void mailtest(){  
 SimpleMailMessage simpleMailMessage=new SimpleMailMessage();  
 //邮件设置  
 simpleMailMessage.setSubject("邮件主题");//邮件主题  
 simpleMailMessage.setText("邮件内容");//邮件内容  
 simpleMailMessage.setTo("3505454117@qq.com");//收件人  
 simpleMailMessage.setFrom("952002423@qq.com");//发件人  
  
 javaMailSender.send(simpleMailMessage);  
}

*//复杂邮件测试*@Test  
public void complexMialTest()throws Exception{  
 //创建复杂消息邮件  
 MimeMessage mimeMessage=javaMailSender.createMimeMessage();  
 MimeMessageHelper mimeMessageHelper=new MimeMessageHelper(mimeMessage,true);  
 //邮件设置  
 mimeMessageHelper.setSubject("复杂邮件主题");//邮件主题  
 mimeMessageHelper.setText("<p style='color:red'>今天开会阿萨德加拉会计师大立科技</p>",true);//邮件内容  
 mimeMessageHelper.setTo("3505454117@qq.com");//收件人  
 mimeMessageHelper.setFrom("952002423@qq.com");//发件人  
 //上传文件  
 mimeMessageHelper.addAttachment("1.jpg",new File("C:\\Users\\Public\\Pictures\\Sample Pictures\\a.png"));  
 mimeMessageHelper.addAttachment("2.jpg",new File("C:\\Users\\Public\\Pictures\\Sample Pictures\\a.png"));  
 javaMailSender.send(mimeMessage);  
}

## Springboot与安全

？？

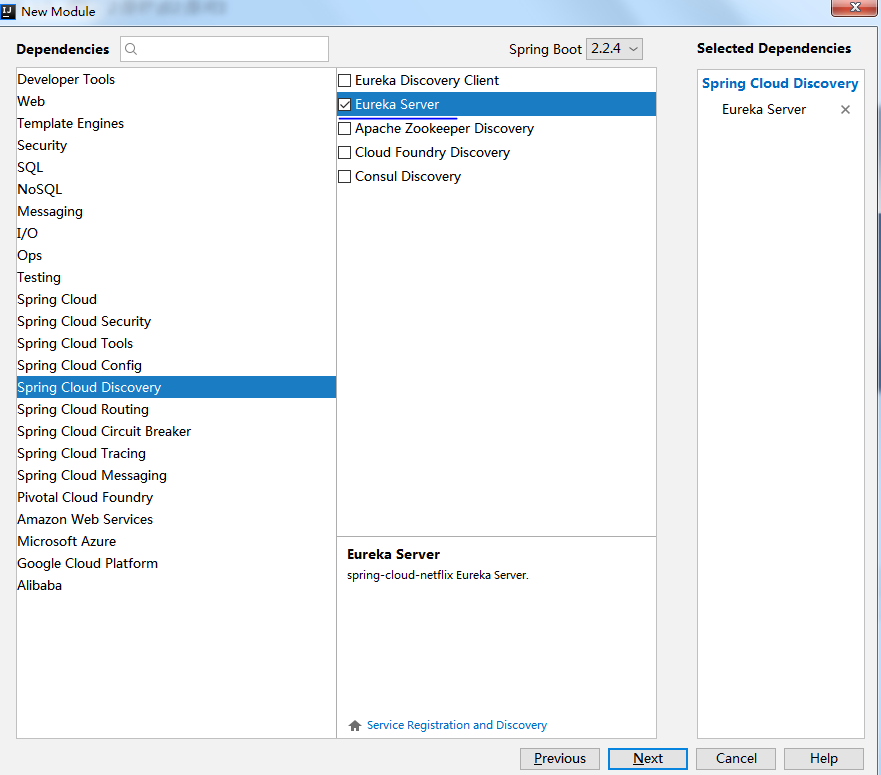
## 分布式

### Springcloud eureka注册中心

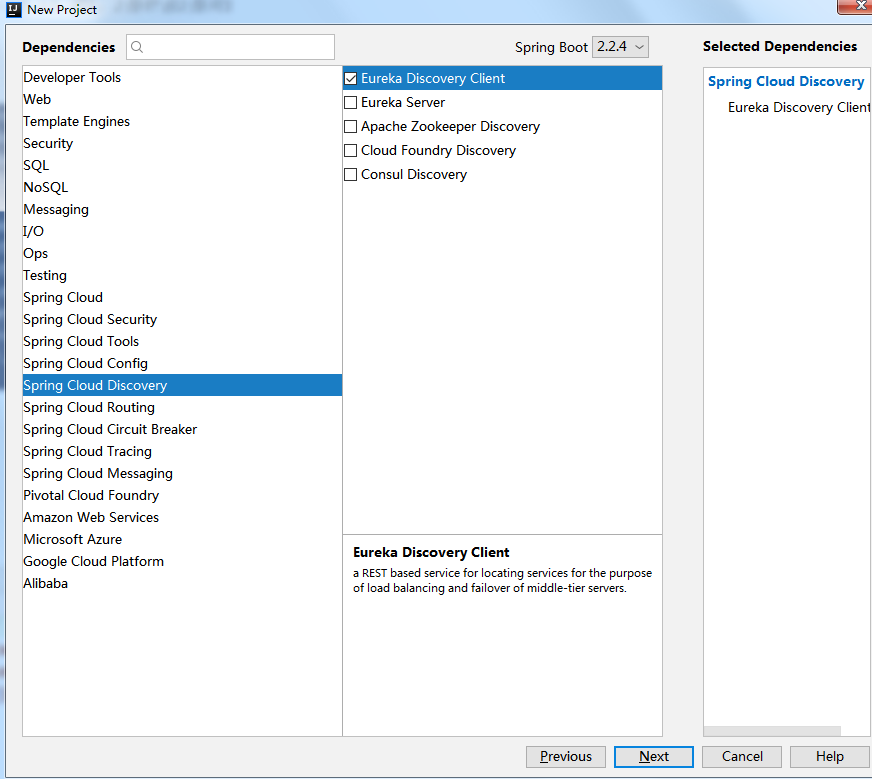
#### 新建一个空项目



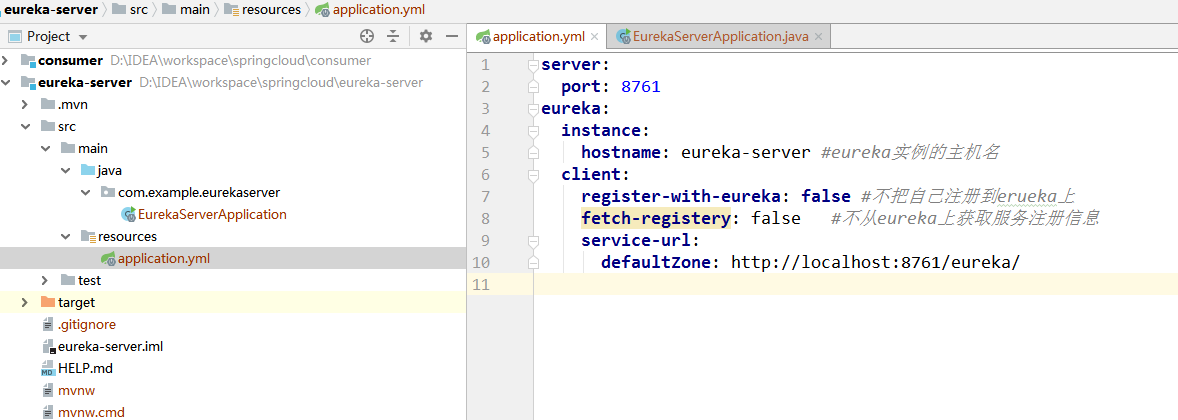
#### 新建注册中心模块



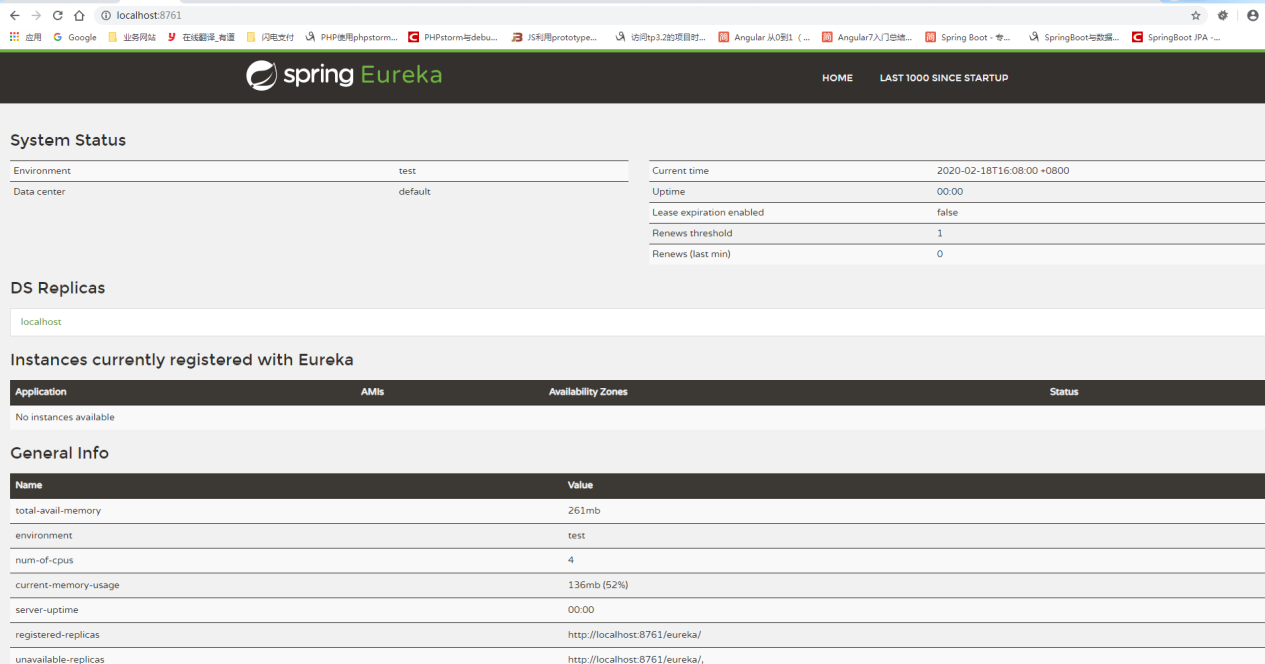
#### 新建provider和consumer模块



#### 配置注册中心yml



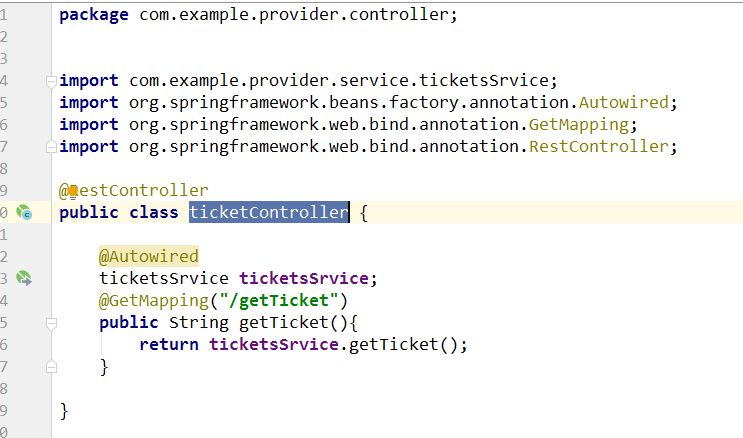
浏览器输入localhost:8761



#### 编写服务提供者并注册到注册中心

##### 在provider中建立ticketsSrvice,ticketController

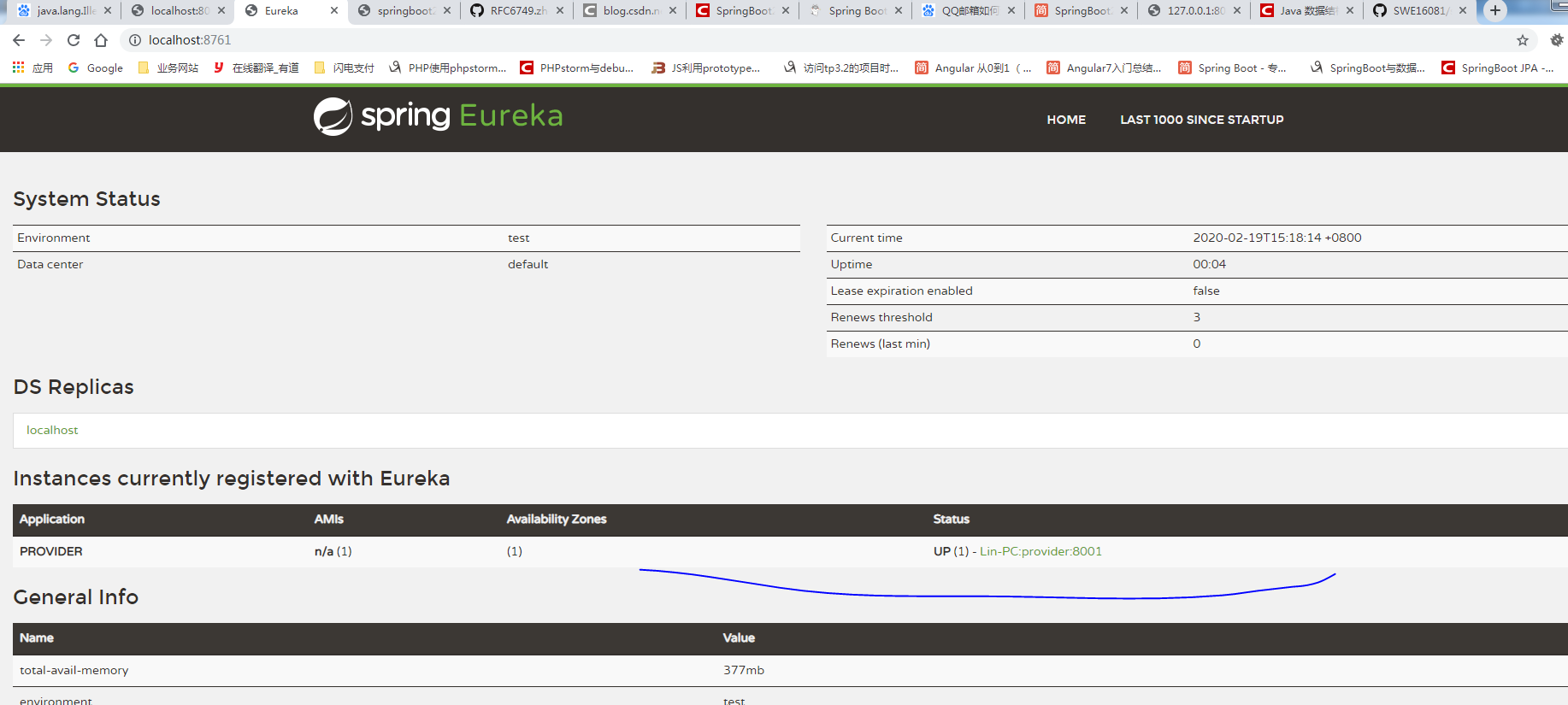




##### 配置provider yml



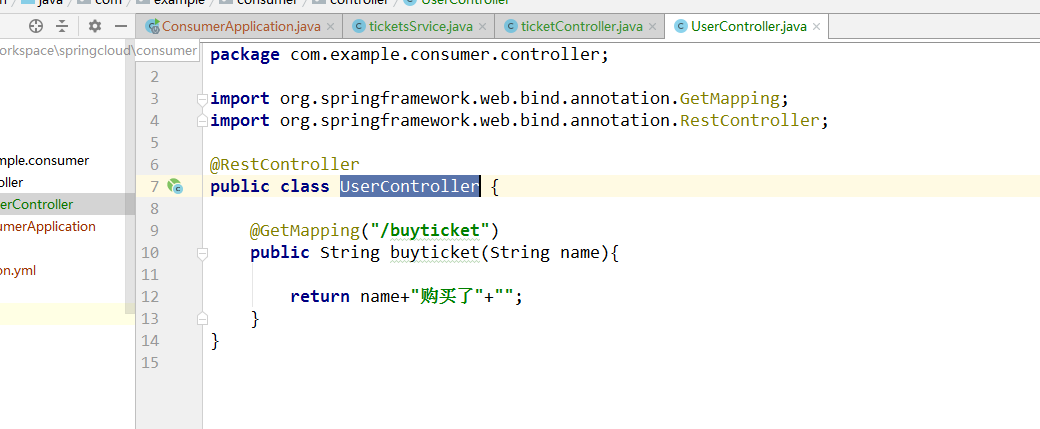
##### 运行 provider 浏览器输入localhost:8001/getTicket



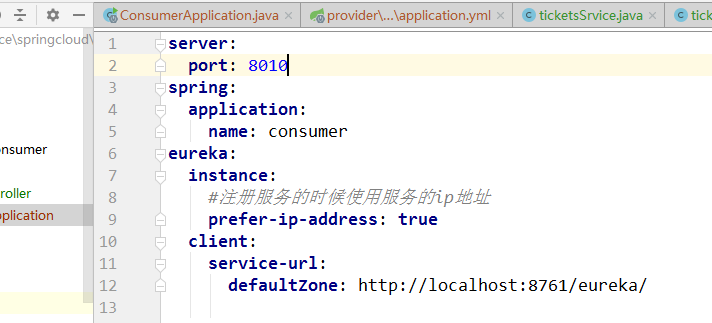
显示有服务提供者注册到注册中心

#### 服务消费者注册到注册中心

##### 编写UserController

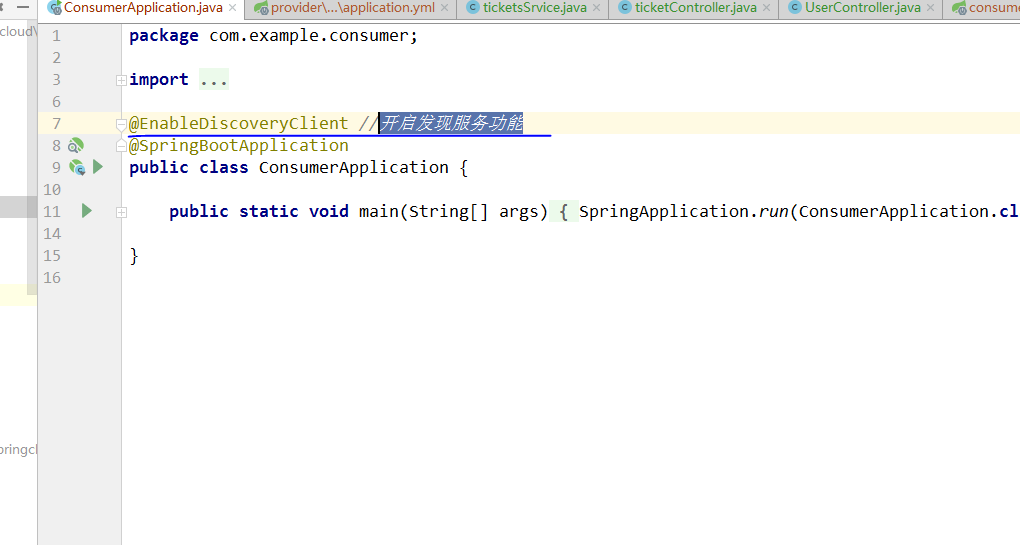


##### 配置yml



##### 启动类开启发现服务功能

*开启发现服务功能,从eureka中获取其他服务并调用*



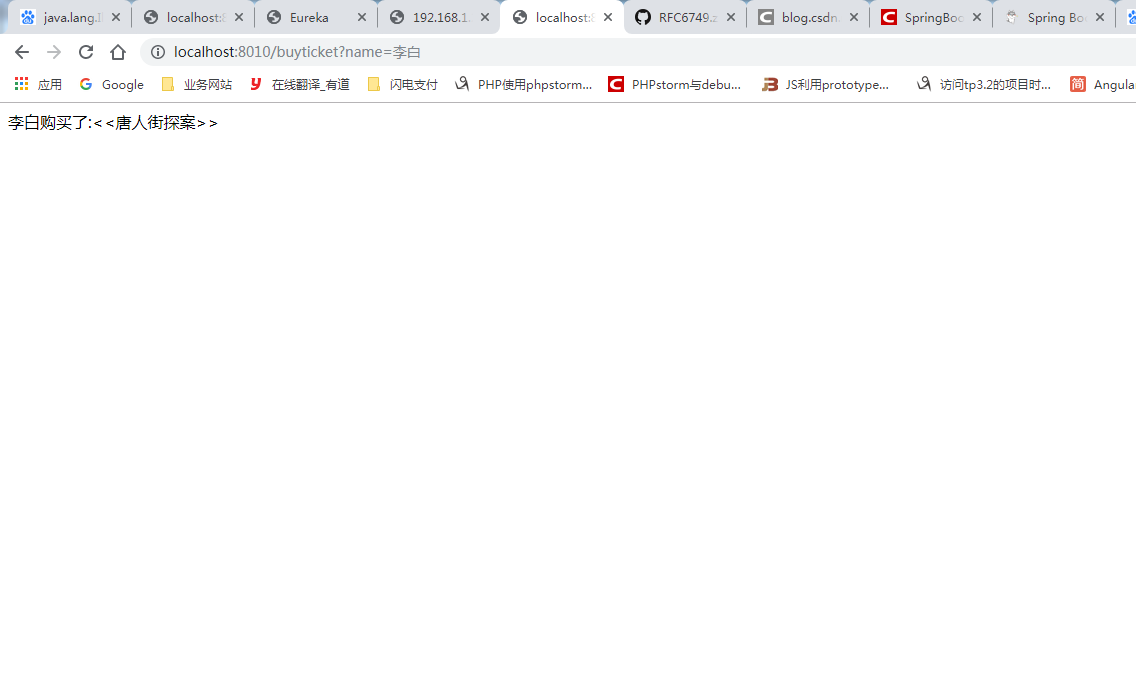
##### 结合resttemplate发送http请求从eureka中获取其他服务



##### 修改UserController



##### 启动



## 配置嵌入式Servlet容器

### 13.1web容器，Servelt容器，servlet

**Web容器(web服务器)**：web容器是一种服务程序，用来处理客户端发出的请求。

主要有：apache,tomcat,IIS,webLogic等

简单的时候就是客户端向服务器发送请求时,服务器上用来接收这些请求并返回相关的静态资源的就是web容器干的事情。

**Servlet**(java服务器的小程序)，一个基于 java技术的Web组件，运行在服务器端，由Servlet容器所管理，用于生成动态内容。

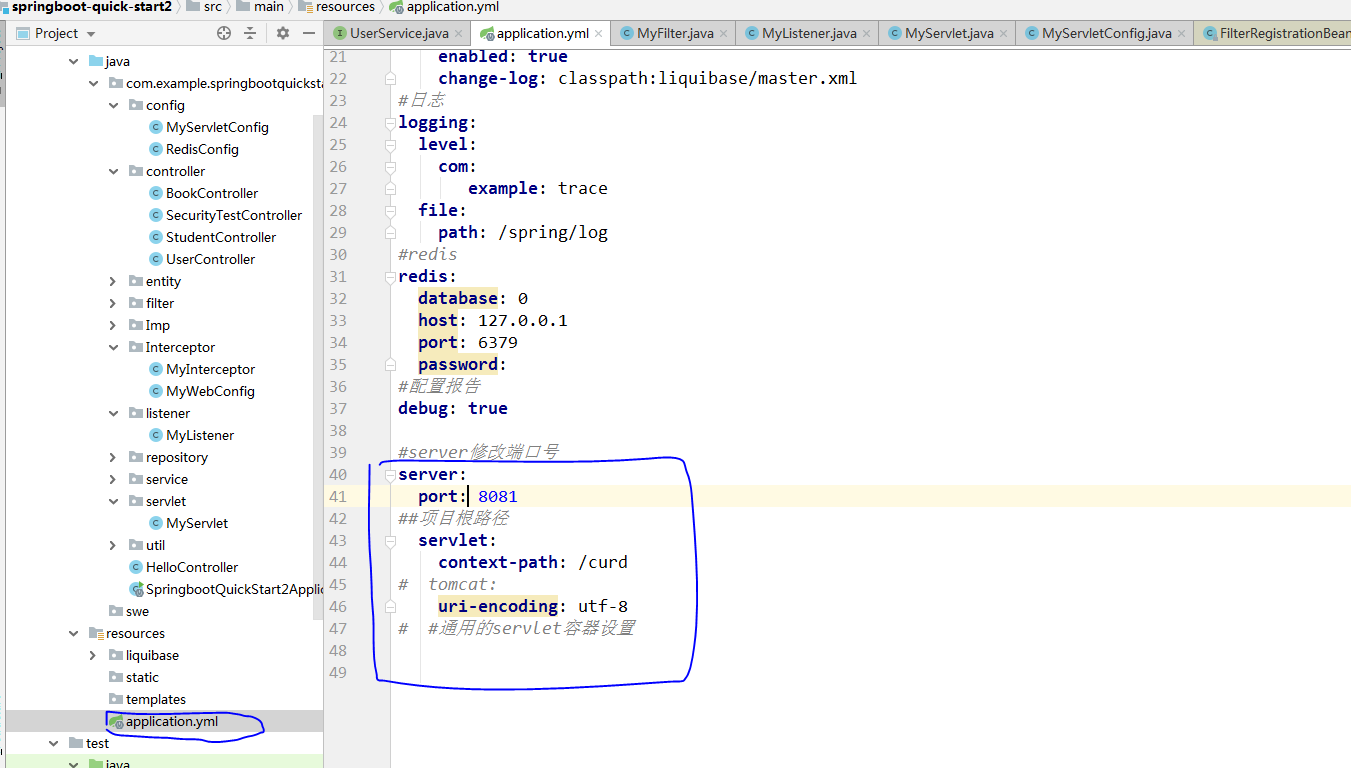
它是一个类，不能独立运行，必须被部署到servlet容器中。只能有servlet来实例化和调用。

**Servlet容器：**

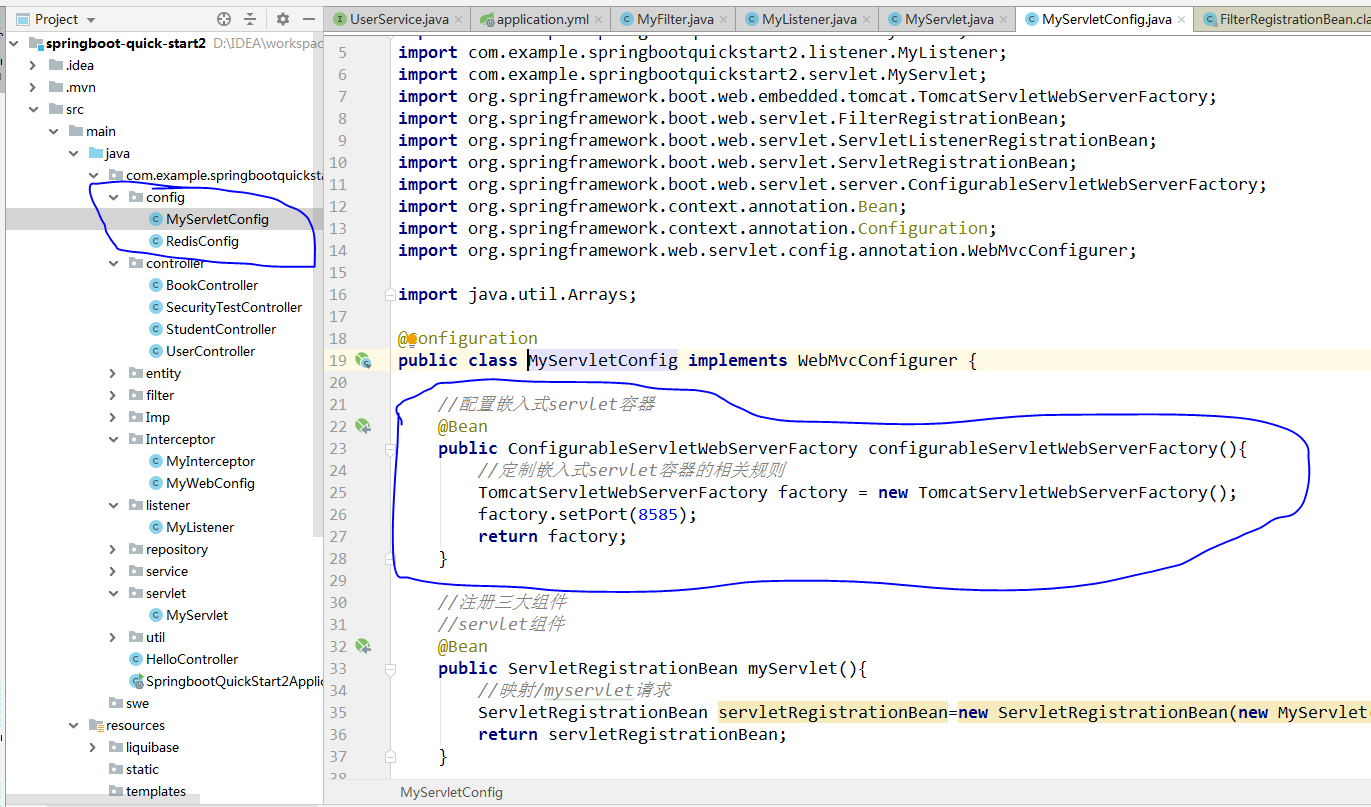
有了servlet之后，用户通过单击某个链接或者直接在浏览器的地址栏中输入URL来访问Servlet，Web服务器接收到该请求后，并不是将 请求直接交给Servlet，而是交给Servlet容器。Servlet容器实例化Servlet，调用Servlet的一个特定方法对请求进行处理， 并产生一个响应。这个响应由Servlet容器返回给Web服务器，Web服务器包装这个响应，以HTTP响应的形式发送给Web浏览器。

springboot默认使用的是Tomcat作为嵌入式的Servlet容器,在外置的Tomcat中我们通过web.xml进行配置。对于springboot而言我们可以通过下面方法修改相关配置

### 13.2配置文件yml 中修改



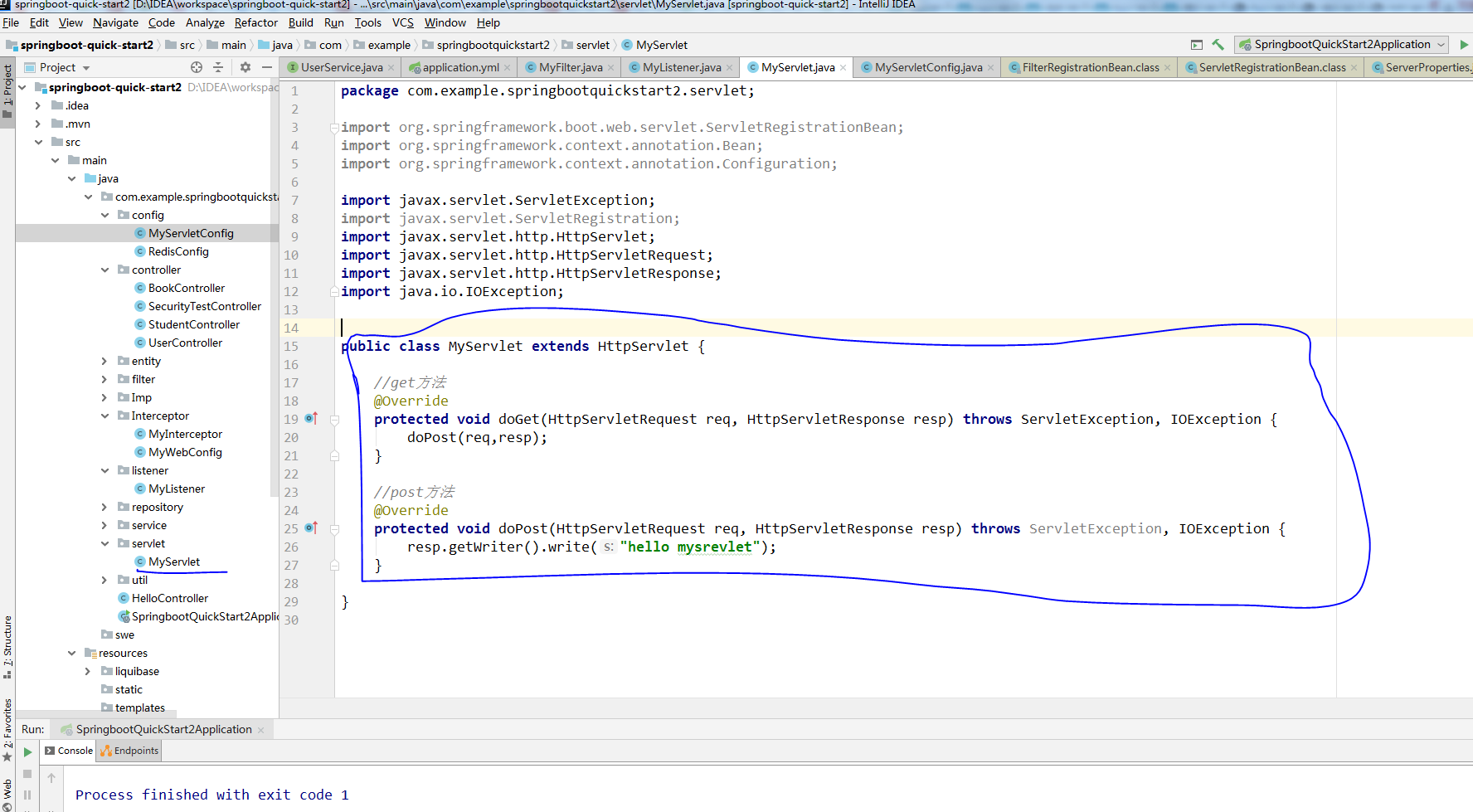
### 13.3编写配置类

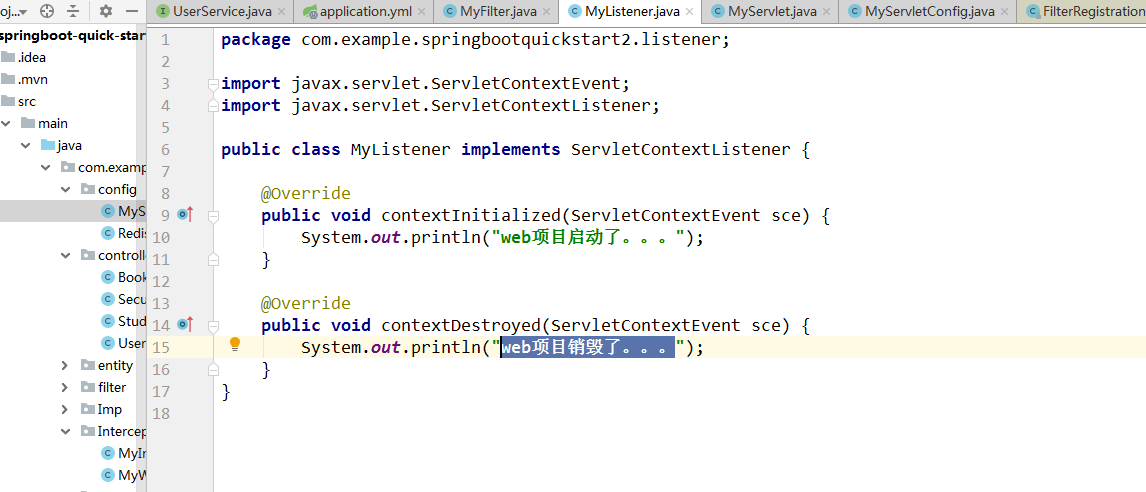
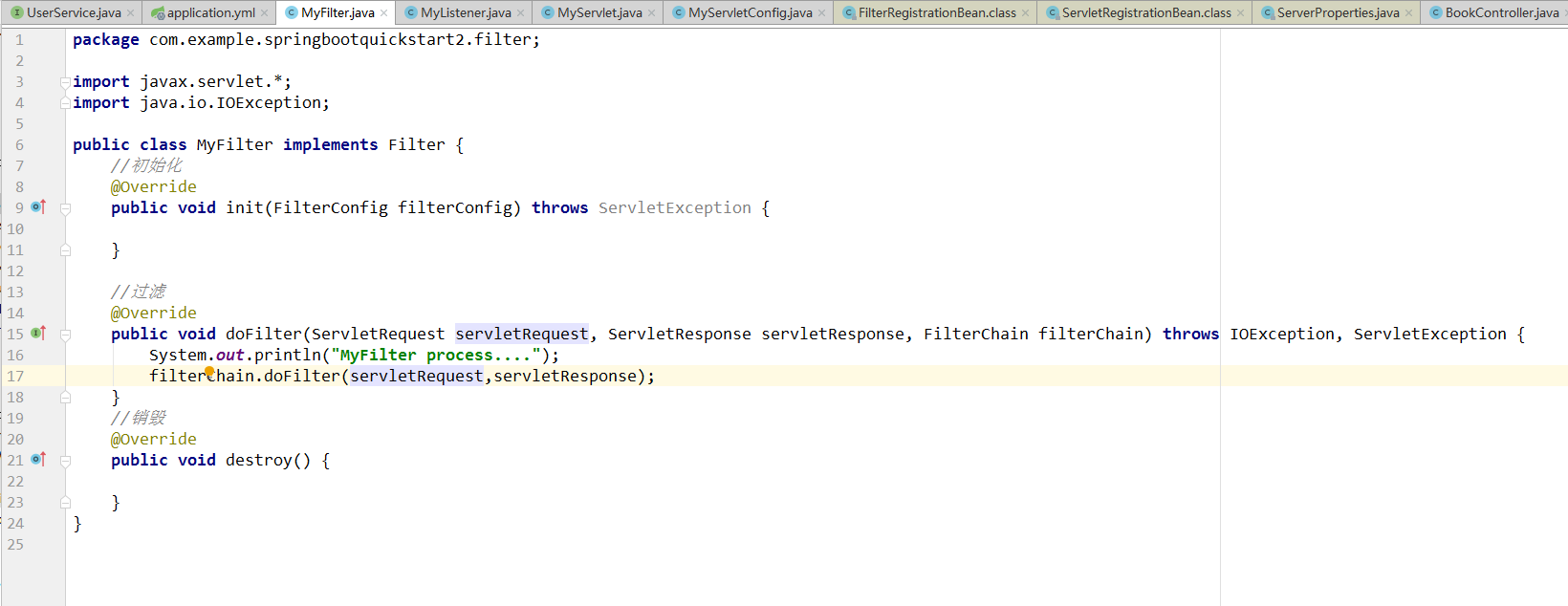


*//配置嵌入式servlet容器*@Bean  
**public** ConfigurableServletWebServerFactory configurableServletWebServerFactory(){  
 *//定制嵌入式servlet容器的相关规则* TomcatServletWebServerFactory factory = **new** TomcatServletWebServerFactory();  
 factory.setPort(8585);  
 **return** factory;  
}

### 13.4注册三大组件Servlet,Filter,Listener

参考：<https://www.cnblogs.com/Myarticles/articles/9628559.html>





代码：

*//注册三大组件  
 //servlet组件* @Bean  
 **public** ServletRegistrationBean myServlet(){  
 *//映射/myservlet请求* ServletRegistrationBean servletRegistrationBean=**new** ServletRegistrationBean(**new** MyServlet(),**"/myServlet"**);  
 **return** servletRegistrationBean;  
 }  
  
 *//filter组件* @Bean  
 **public** FilterRegistrationBean<MyFilter> filterRegistrationBean(){  
 FilterRegistrationBean<MyFilter> myFilterFilterRegistrationBean = **new** FilterRegistrationBean<>();  
 myFilterFilterRegistrationBean.setFilter(**new** MyFilter());  
*// myFilterFilterRegistrationBean.addUrlPatterns("/hello", "/myServlet");* myFilterFilterRegistrationBean.setUrlPatterns(Arrays.*asList*(**"/hello"**, **"/myServlet"**));  
 **return** myFilterFilterRegistrationBean;  
 }  
  
 *//listener组件* @Bean  
 **public** ServletListenerRegistrationBean myListener(){  
 ServletListenerRegistrationBean<MyListener> registrationBean = **new** ServletListenerRegistrationBean<>(**new** MyListener());  
 **return** registrationBean;  
 }

13.5切换其他servlet容器

## 十三．Webjar

什么是WebJars？

WebJars是将客户端（浏览器）资源（JavaScript，Css等）打成jar包文件，以对资源进行统一依赖管理。

## Thymeleaf模板使用

## 十五.引入 Kaptcha验证码

### 项目详情

前后端分离，angular做前端，springboot做后端，使用kaptcha进行验证码验证

### 步骤

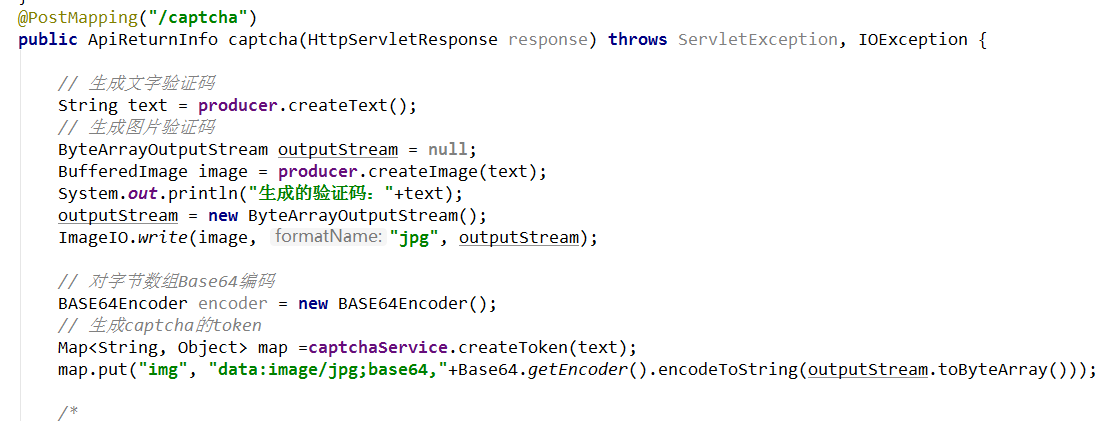
1. 后端生成base64验证码图片并将验证码以token为键值存入redis缓存
2. 前端请求springboot后端的bas64验证码图片以及token值
3. 前端发送携带账号密码验证码,token参数的请求
4. 后端接收请求并取出redis值并加以比对

### 后端springboot Kaptcha配置

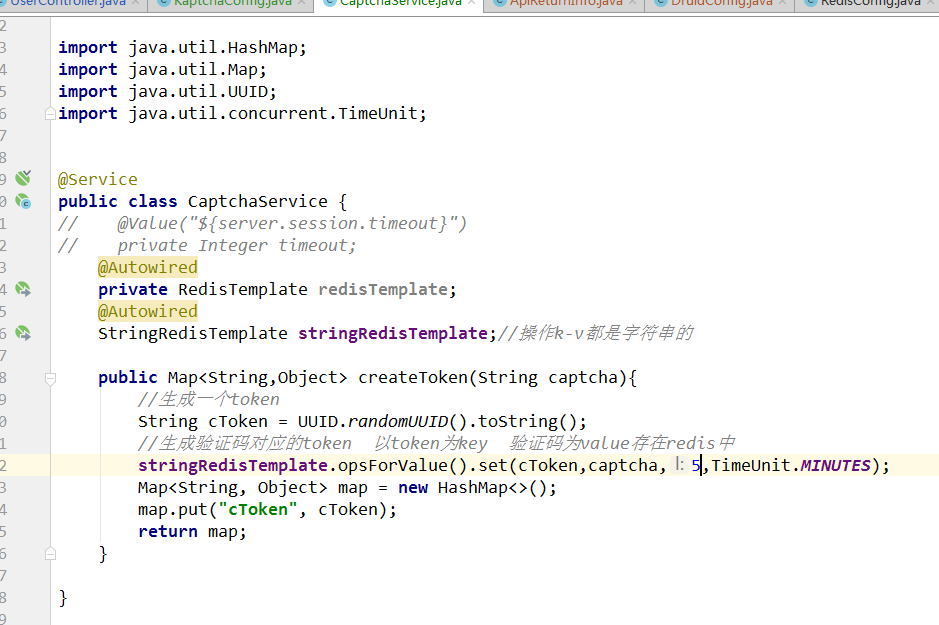
1. 自定义kapatcha验证码配置文件用于设置验证码图片的边框，背景，颜色，字体等样式



1. 在controller中生成验证码并存储于redis



1. 定义一个service用于自动存储到redis



参考

<https://blog.csdn.net/baidu_33969289/article/details/87473766>

<https://blog.csdn.net/m0_38121868/article/details/83416484>

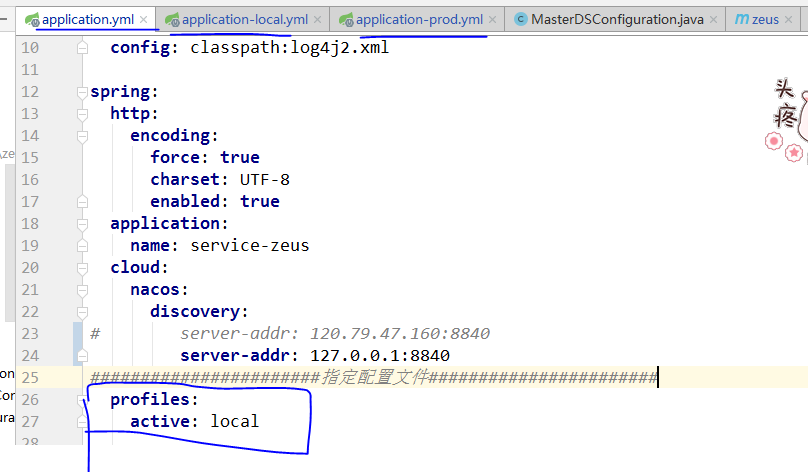
Kaptcha相关的配置表 字体 大小 背景

<https://www.jianshu.com/p/a3525990cd82>

## 配置文件详解

#### 1.yml

配置多个yml用于线上线下不同环境测试



使用

Profiles:

Active：local--》指定使用哪个yml

## feign模块间相互调用

服务之间调用使用feign。

### 17.0准备工作

#### 引入依赖：

<!-- feign微服务模块调用 -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
 <version>2.2.2.RELEASE</version>  
</dependency>

#### 启动类添加@EnableFeignClients注解

#### 配置yml（或定义配置个人不支持）

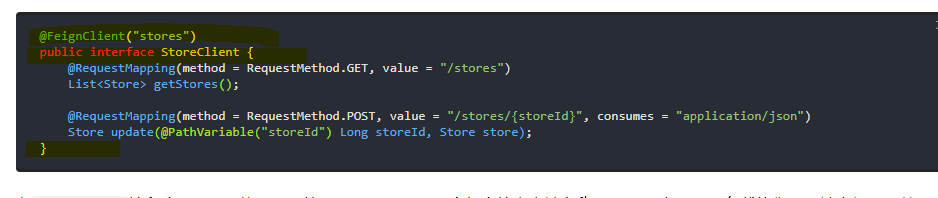
feign:  
 client:  
 config:  
 feignName:  
 connectTimeout: 5000  
 readTimeout: 5000  
 loggerLevel: full  
 errorDecoder: com.example.SimpleErrorDecoder  
 retryer: com.example.SimpleRetryer  
 requestInterceptors:  
 - com.example.FooRequestInterceptor  
 - com.example.BarRequestInterceptor  
 decode404: false  
 encoder: com.example.SimpleEncoder  
 decoder: com.example.SimpleDecoder  
 contract: com.example.SimpleContract

官方文档：[https://cloud.spring.io/spring-cloud-static/spring-cloud-openfeign/2.2.2.RELEASE/reference/html/#netflix-feign-starter](https://cloud.spring.io/spring-cloud-static/spring-cloud-openfeign/2.2.2.RELEASE/reference/html/" \l "netflix-feign-starter)

### 17.1注：

要使用feign就要先顶一个接口然后用@FeignClient进行注解

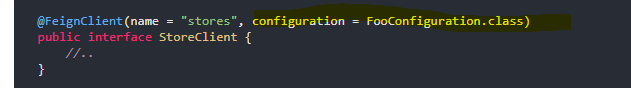
如：



### 17.2关于@FeignClient的使用

第一个参数为name注解名

还可以像这样



FooConfiguration.class是一个配置类，如何定义以后扩展。使用配置类配置和在yml里面配置是一样的,一般用yml

@Configuration

public class FooConfiguration {

    @Bean

    public Contract feignContract() {

        return new feign.Contract.Default();

    }

    @Bean

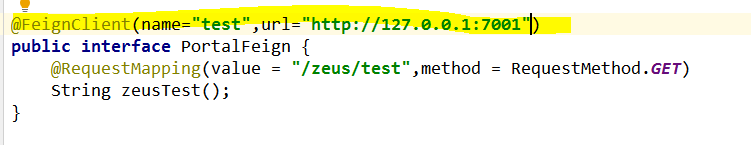
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

        return new BasicAuthRequestInterceptor("user", "password");

    }

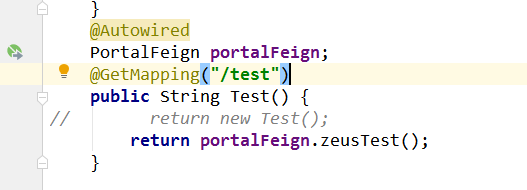
}

通常使用name,url配置像这样



@RequestMapping(value = "/zeus/test",method = RequestMethod.GET)//要访问模块的路径与方法  
String zeusTest();

**控制器的调用**

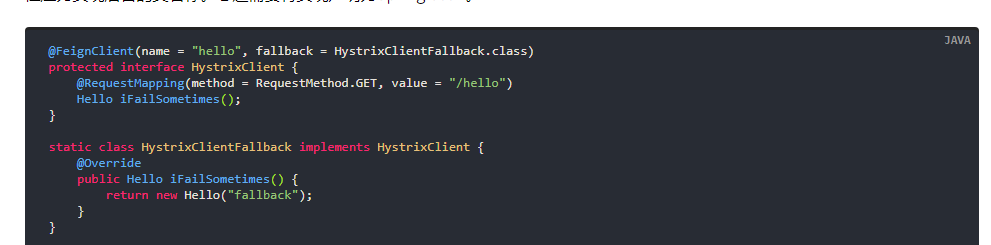


### 17.3断路由Hystrix

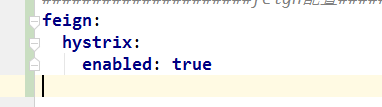
简单地说就是你请求其他模块出错时，Hystrix将执行它指定的代码，可能给你返回一个错误

信息。

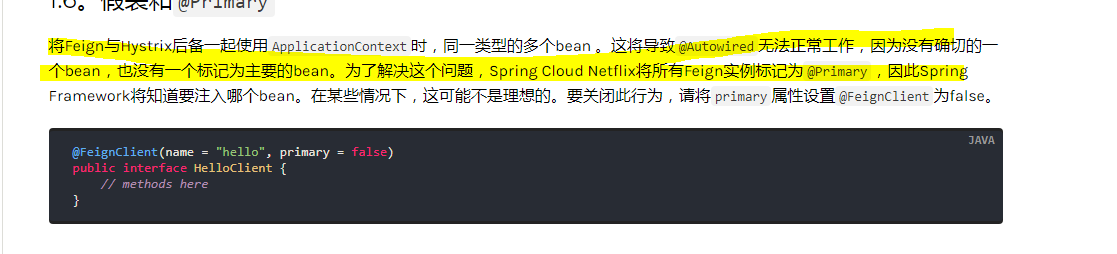
像这样

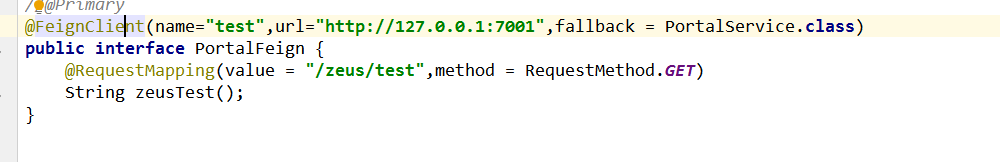


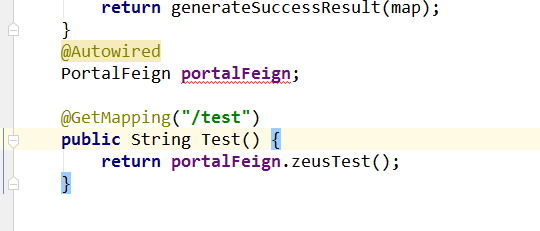
使用断路由要设置**hystrix**: **enabled**: **true**



@Autowired无法使用

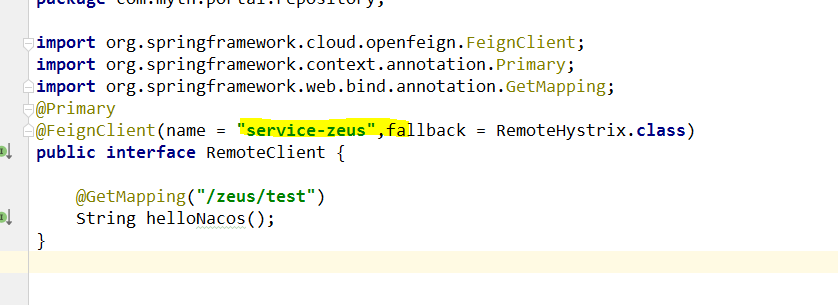






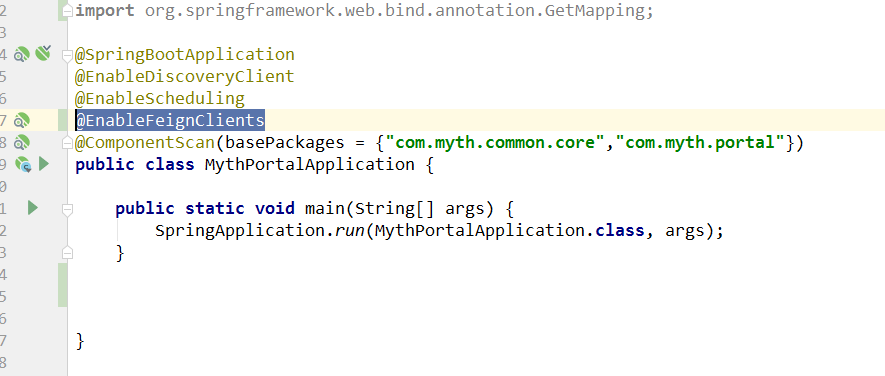
<https://blog.csdn.net/qq_33619378/article/details/95353326>

### 17.4结合nacos使用fegin



其他配置保持不变，@FignClient中name为模块名

然后再在启动类加上@EnableFeignClients



## 十八.jekins自动化各种任务，包括构建、测试和部署软件

# Java

Iterator，list，arraylist ，map

aop面向方面编程

Oop面向对象编程

Final关键字

用final修饰一个类，表示该类不能被继承

被final修饰的方法不能被重写

 如果父类中有final修饰的方法，那么子类不能去重写

Final修饰成员变量必须要赋初始值，而且是只能初始化一次

static修饰的属性强调它们只有一个，final修饰的属性表明是一个常数（创建后不能被修改）。static final修饰的属性表示一旦给值，就不可修改，并且可以通过类名访问

## 重载！！！

Java允许同名函数不同参数或者参数顺序不同

## Idea 快捷键使用

*//ctrl+o打开重写方法*

# Springboot idea导入项目

在idea中打开file->找到项目底下的pom.xml，as project 就会自动导入

Git别人的springboot多模块项目，file->project structure->