**Spring Boot开发**

20191230

目录

[第1章 SpringBoot简介 5](#_Toc22413)

[1.1. SpringBoot简介 5](#_Toc30179)

[1.1.1. Spring、SpringMVC、SpringBoot的关系 5](#_Toc9146)

[1.1.2. SpringBoot的优点 5](#_Toc31195)

[1.2. 微服务 5](#_Toc10745)

[1.3. 学习本书的建议 6](#_Toc19921)

[第2章 开发环境搭建 6](#_Toc8865)

[2.1. 搭建Java开发环境 6](#_Toc3160)

[2.2. Maven安装与配置 6](#_Toc23344)

[2.3. IDEA安装与配置 6](#_Toc15714)

[2.4. 第一个SpringBoot应用程序 6](#_Toc4750)

[第3章 SpringBoot基础 6](#_Toc10221)

[3.1. SpringBoot项目结构 6](#_Toc20633)

[3.1.1. pom.xml 7](#_Toc26097)

[3.1.2. src 7](#_Toc23235)

[3.1.3. Target 8](#_Toc29928)

[3.2. SpringBoot的Starter 8](#_Toc14102)

[3.2.1. Starter是什么 8](#_Toc3943)

[3.2.2. 怎么使用Starter 8](#_Toc30677)

[3.2.3. Starter原理介绍 9](#_Toc16538)

[3.3. SpringBoot常用注解 9](#_Toc7563)

[3.3.1. @Component 9](#_Toc26798)

[3.3.2. @Bean 9](#_Toc2643)

[3.3.3. @Import 10](#_Toc24934)

[3.3.4. @Configuration 12](#_Toc6298)

[3.3.5. @ComponentScan 13](#_Toc11369)

[3.3.6. @Conditional 13](#_Toc14504)

[3.4. 了解自动配置 15](#_Toc8471)

[3.5. HelloWorld程序解析 17](#_Toc21425)

[3.5.1. 依赖引入 17](#_Toc32226)

[3.5.2. 程序入口 17](#_Toc26691)

[第4章 配置文件 17](#_Toc8566)

[4.1. YAML简介 17](#_Toc13127)

[4.1.1. 基本语法 17](#_Toc4675)

[4.1.2. 数据类型 17](#_Toc4597)

[4.2. 配置文件加载顺序 17](#_Toc28445)

[4.3. Spring Boot基本配置介绍 17](#_Toc27746)

[4.4. 自动配置原理 18](#_Toc31242)

[第5章 日志 18](#_Toc13722)

[5.1. 日志简述 18](#_Toc13958)

[5.1.1. 日志的抽象层 18](#_Toc13661)

[5.1.2. 日志的实现层 18](#_Toc7863)

[5.2. SpringBoot使用的日志-SLF4j 18](#_Toc12539)

[5.2.1. SpringBoot日志依赖关系 18](#_Toc20327)

[5.2.2. 使用SLF4j 18](#_Toc29672)

[5.2.3. 更改日志配置 18](#_Toc18631)

[5.3. 更改日志框架 18](#_Toc32243)

[第6章 SpringBoot Web开发 18](#_Toc4674)

[6.1. Spring核心技术 18](#_Toc3152)

[6.1.1. 控制反转IOC 18](#_Toc16870)

[6.1.2. 面向切面AOP 19](#_Toc7121)

[6.2. SpringMVC 19](#_Toc11319)

[6.2.1. SpringMVC架构 19](#_Toc13825)

[6.2.2. 转发与重定向 19](#_Toc26465)

[6.2.3. 拦截器 19](#_Toc10238)

[6.2.4. 类型转换器 19](#_Toc25886)

[6.2.5. 异常处理 19](#_Toc31049)

[6.2.6. SpringMVC的自动配置 19](#_Toc16602)

[6.3. Thymeleaf模板引擎 19](#_Toc27104)

[6.3.1. Thymeleaf简介 19](#_Toc3026)

[6.3.2. 引入Thymeleaf 19](#_Toc14942)

[6.3.3. Thymeleaf语法规则 19](#_Toc28479)

[6.3.4. Thymeleaf应用实例 19](#_Toc22013)

[6.4. SpringBoot对JSP的支持 19](#_Toc31951)

[第7章 第七章 SpringBoot 数据访问 20](#_Toc30115)

[7.1. Mybatis 20](#_Toc2045)

[7.1.1. 简述Mybatis 20](#_Toc19556)

[7.1.2. SpringBoot使用Mybatis 20](#_Toc30464)

[7.1.3. Mybatis注解方式开发 25](#_Toc4391)

[7.2. MongoDB 27](#_Toc12400)

[7.2.1. 简述 27](#_Toc5705)

[7.2.2. 配置MongoDB 27](#_Toc13010)

[7.2.3. 基本用法 29](#_Toc26478)

[7.2.4. SpringBoot使用MongoDB 33](#_Toc3486)

[7.3. Redis 36](#_Toc26351)

[7.3.1. Redis简介 36](#_Toc28557)

[7.3.2. Redis安装与配置 37](#_Toc28533)

[7.3.3. Redis数据类型 39](#_Toc23590)

[7.3.4. Redis命令 41](#_Toc25547)

[7.3.5. SpringBoot使用Redis 46](#_Toc27364)

[第8章 SpringBoot Actuator 49](#_Toc9712)

[8.1. SpringBoot Actuator 简介 49](#_Toc17523)

[8.2. 原生端点使用 49](#_Toc30694)

[8.3. 定制 Actuator 49](#_Toc30411)

[第9章 Spring Security 49](#_Toc7126)

[9.1. Spring Security简介 49](#_Toc16181)

[9.2. 配置Spring Security 49](#_Toc30339)

[9.3. Spring Security入门 49](#_Toc8584)

[9.3.1. 框架原理 49](#_Toc19999)

[9.3.2. 核心组件 49](#_Toc7698)

[9.4. 实例 49](#_Toc20738)

[第10章 SpringBoot部署与测试 49](#_Toc9229)

[10.1. 传统java web应用的部署方式 50](#_Toc10957)

[10.1.1. 方式一：直接拷贝 50](#_Toc3224)

[10.1.2. 方式二：添加XML文件 51](#_Toc25604)

[10.1.3. 方式三：隐式部署 52](#_Toc2693)

[10.2. Spring Boot 部署方式 58](#_Toc26409)

[10.2.1. Spring Boot快速部署 58](#_Toc7543)

[10.2.2. Spring Boot 部署原理 59](#_Toc8144)

[10.3. Docker 63](#_Toc19245)

[10.3.1. 什么是Docker？ 64](#_Toc2895)

[10.3.2. 安装Docker 64](#_Toc14317)

[10.3.3. Docker运行原理 65](#_Toc4085)

[10.3.4. Docker常用命令介绍 67](#_Toc16229)

[10.3.5. Docker部署tomcat 69](#_Toc22252)

[10.3.6. Docker部署mysql 70](#_Toc31105)

[10.4. 数据迁移 72](#_Toc1585)

[10.4.1. 平滑地迁移数据库 72](#_Toc13063)

[10.4.2. 我们的解决方案 73](#_Toc25415)

[10.5. 单元测试 73](#_Toc24564)

[10.5.1. 无框架下的测试 74](#_Toc27794)

[10.5.2. JUnit 75](#_Toc32247)

[10.5.3. Spring Boot中的测试 78](#_Toc9854)

[第11章 项目实战 81](#_Toc20341)

# SpringBoot简介

## SpringBoot简介

来看Spring官网对Spring Boot的介绍：Spring Boot使您可以轻松地创建独立的、生产级的、基于Spring的应用程序，您可以“只是运行”。我们对Spring平台和第三方库有一个独到的见解，这样您就可以从最少的麻烦开始了。大多数Spring引导应用程序需要最少的Spring配置。

Spring Boot对一些常用的功能、一些第三方库做了自动配置，使得你在使用Spring Boot的时候无需先进行重复而又烦人的工作，你只需要对项目做一些简单的、少量的配置就可以像运行一个最简单的jar程序一样运行一个Spring Boot应用。

### Spring、SpringMVC、SpringBoot的关系

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring这个轻量级的框架已经成为web开发事实上的标准。

Spring为javaEE开发主要提供了以下功能：

IOC：资源集中管理，实现资源的可配置和易管理。降低了使用资源双方的依赖程度，也就是我们说的耦合度。

AOP：的将一些通用任务，如安全、事务、日志等集中进行管理，提高了复用性和管理的便捷性

ORM和DAO提供了与第三方持久层框架的良好整合，简化了底层数据访问。

Web MVC：使得复杂项目更加容易维护，减少项目之间的耦合。

Spring Boot集成了Spring的所有组件，你可以在程序中使用这些组件。

Spring Boot在Spring的基础上扩充了许多功能，许多特性。而spring mvc 是只是spring 处理web层请求的一个模块。

因此他们的关系可以描述成这样：

spring mvc < spring < springboot。

### SpringBoot的优点

## 微服务

## 学习本书的建议

# 开发环境搭建

## 搭建Java开发环境

## Maven安装与配置

## IDEA安装与配置

## 第一个SpringBoot应用程序

# SpringBoot基础

## SpringBoot项目结构

Spring Boot框架对项目的结构并没有做强制约束，但是分工良好、结构清晰的项目结构可以规范开发的模块分工，增强代码的可读性，降低维护的难度与成本。维护一个结构清晰的项目结构是十分有必要的。

下面介绍一个典型的Spring Boot开发Web应用的项目结构。

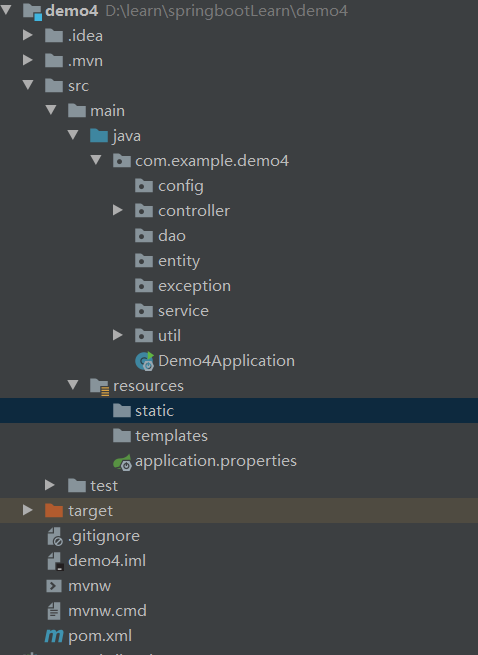


图3-1

### pom.xml

POM是项目对象模型(Project Object Model)的简称,它是Maven项目中的文件.主要用来管理项目的依赖关系，再这个文件中可以选择引入项目所需要的依赖。还可以用于管理：源代码、配置文件、开发者的信息和角色、问题追踪系统、组织信息、项目授权、项目的url等等。

### src

1. **main**

main层由java目录和resources组成，java目录存放java代码，resources目录存放资源。

**代码层java:**

1. 启动类Demo4Application,项目的入口类。
2. Config:存放项目的配置类
3. Controller: 存放作为controller的类
4. Dao:数据接口访问层
5. entity:实体层
6. Exception:存放自定义的异常类
7. Serivce:数据服务层
8. Util:放一些常用的工具类

**资源目录resources**

1. 项目配置文件application.yml/application.properties
2. 静态资源目录static: 用于存放html、css、js、图片等资源

3.放视图模板，springBoot的动态页面

1. **test**

test层的作用是在项目的开发阶段进行测试工作。

### Target

target用来存放项目构建后的文件和目录、jar包、war包、编译的class文件。

## SpringBoot的Starter

### Starter是什么

starter可以称为场景启动器。一个starter包含了这个应用这个场景所需要的jar包，和这个场景一些初始化配置。

### 怎么使用Starter

要使用starter,只需要将这个starter再pom文件中引入即可。例如我们想要使用web相关的功能。只需要：

1.在pom.xml文件中加入

<dependency>

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

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

</dependency>

这样就在项目总引入了web相关的jar包以及初始配置。

2.想要指定自己想要的配置，只要在配置文件中写入即可。例如想要指定程序的启动端口号，可以在application.properties写入server.port=”端口号”或在application.yml中写入:

server:

Port=”端口号”

1. 编写自己的业务代码

### Starter原理介绍

下面通过实现一个自定义的Starter来介绍Starter的原理。

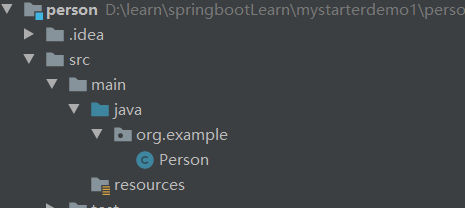
通过以下步骤来实现，

1. 编写要封装的组件
2. 编写自动配置组件
3. 编写starter
4. 导入starter并测试

下面完成上述四个步骤来实现一个自定义的starter

1. 编写要封装的组件

创建一个maven项目



**pom文件**

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

<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>org.example</groupId>

<artifactId>person</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

**Person.class**

public class Person {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

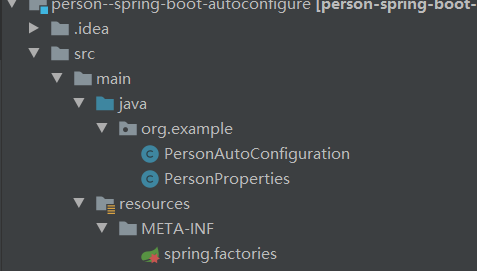
}

}

然后执行命令mvn clear install将组件打包到maven仓库。

2.编写自动配置组件

创建一个maven项目



**pom文件**

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

<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>org.example</groupId>

<artifactId>person-spring-boot-autoconfigure</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

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

<artifactId>spring-boot-autoconfigure</artifactId>

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.example</groupId>

<artifactId>person</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

<build>

<resources>

<resource>

<directory>src/main/resources</directory>

<includes>

<include>META-INF/\*</include>

</includes>

</resource>

</resources>

</build>

</project>

引入spring-boot-autoconfigure、spring-boot-configuration-processor还有person组件依赖。

**PersonAutoConfiguration.class**

@Configuration

@EnableConfigurationProperties(PersonProperties.class)

public class PersonAutoConfiguration {

@Bean

public Person person(PersonProperties properties){

Person person = new Person();

person.setName(properties.getName());

person.setAge(properties.getAge());

return person;

}

}

这是一个自动配置类，可以为容器添加一个person组件（bean），这里用@EnableConfigurationProperties(PersonProperties.class)注解使得PersonProperties.class注入容器，person对象的属性值从PersonProperties.class中获得。

**PersonProperties.class**

@ConfigurationProperties(prefix = "person")

public class PersonProperties {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

使用@ConfigurationProperties注解将配置文件（yml/properties）中指定前缀的配置（这里是 "person"）转为PersonProperties对象中属性的值。

**spring.factories**

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

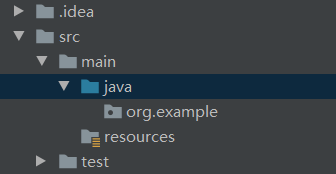
org.example.PersonAutoConfiguration

类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值被加入到了容器中，用他们来做各种功能的自动配置。

然后执行命令mvn clear install将这个自动配置组件打包到maven仓库。

3.编写starter

新建一个maven项目



**pom文件**

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

<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>org.example</groupId>

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

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

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

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

<version>2.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.example</groupId>

<artifactId>person-spring-boot-autoconfigure</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

</project>

可以看到这个starter项目是没有java代码的，只有一个pom.xml文件，在pom.xml文件中引入了org.example.person-spring-boot-autoconfigure依赖。

1. 导入starter并测试

新建一个springBoot工程。在pom.xml文件中引入依赖：

<dependency>

<groupId>org.example</groupId>

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

<version>1.0-SNAPSHOT</version>

</dependency>

在配置文件application.properties中写入：

person.name=zhangsan

person.age=12

运行测试方法personTest()：

@SpringBootTest

class DemoApplicationTests {

@Autowired

Person person;

@Test

void personTest(){

System.out.println(person);

System.out.println(person.getName()+person.getAge());

}

}

运行结果如下：

starter4

这个运行结果说明了此person组件（bean）成功注入了容器，间接说明了自动配置类生效，自定义starter成功实现。

总结：starter本身没有java,它的作用是导入一些依赖包，依赖包中有自动配置类，向应用提供各种功能组件。

## SpringBoot常用注解

### @Component

带此注解的类看为组件，当使用基于注解的配置和类路径扫描的时候，这些类就会被实例化。

@Component

Class Person{

String name;

}

上例中这个Person就会被实例化，由Spring IOC容器管理。使用时可以用@Autowired注解来注入。

@Autowired

Person person;

这个注解也可以当作元注解使用，来注解其他注解。

如@Controller注解就是一个被@Component注解的注解。当@Controller被@Component注解了后，@Controller也就有了@Component注解的特性。所有被@Controller注解的类也会被IOC容器管理。

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Controller {

### @Bean

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。

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。

例如：

public class E2 {

public E2(){

System.out.println("E2 initialize");

}

}

@Configuration

public class BeanConfig {

@Bean

public E2 myBean(){

return new E2();

}

}

@Test  
public void BeanTest(){  
 ApplicationContext context = new AnnotationConfigApplicationContext(BeanConfig.class);

//bean名称为指定时，默认的时方法的名称,可以用@Bean(name = "xx")来指定bean的名称  
 Object myBean = context.getBean("myBean");  
}

### @Import

有时没有把某个类注入到IOC容器中，但在使用的时候需要获取该类对应的bean，这时就可以用到@Import注解。

@Import的用法有以下4种：

1.直接用类名导入bean

2.通过@Configuration注解的配置类导入bean

3.通过ImportSelector的实现类导入bean

4.通过ImportBeanDefinitionRegistrar的实现类导入bean

下面给出示例，四种方式的用法分别如下：

1.导入一个bean  
public class Bean1 {

}

@Configuration

@Import({Bean1.class})

public class MyConfig {

}

2.导入@Configuration注解的配置类

public class Bean2 {

}

@Configuration

public class BeanConfig {

@Bean

public Bean2 getBean2(){

return new Bean2();

}

}

@Configuration

@Import({BeanConfig.class})

public class MyConfig {

}

3.导入ImportSelector的实现类

public class Bean3 {

}

public class MyImportSelector implements ImportSelector {

@Override

public String[] selectImports(AnnotationMetadata annotationMetadata) {

return new String[]{"com.example.demo4.ImportDemo.Bean3"};

}

}

@Configuration

@Import({MyImportSelector.class})

public class MyConfig {

}

4.导入ImportBeanDefinitionRegistrar的实现类

public class Bean4 {

}

public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {

@Override

public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(Bean4.class);

registry.registerBeanDefinition("Bean4", rootBeanDefinition);

}

}

@Configuration

@Import({MyImportBeanDefinitionRegistrar.class})

public class MyConfig {

}

在此我们写一个统一的配置类MyConfigAll,和一个测试类同一测试以上4种方式。

@Configuration

@Import({Bean1.class,BeanConfig.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})

public class MyConfigAll {

}

再spring Boot的测试类里新建一个方法

@Test

public void ImportTest(){

AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext(MyConfig.class);

String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNames();

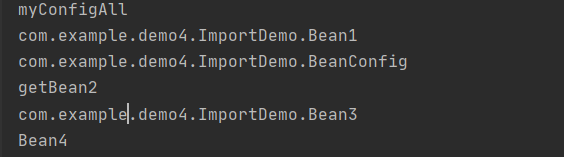
for (String beanDefinitionName: beanDefinitionNames) {

System.out.println(beanDefinitionName);

}

}

测试这个方法，结果如下：



可以看出Bean1,Bean2,Bean3,Bean4通过不同的4种导入方法被导入SpringIOC容器中。

### @Configuration

按住Ctrl点入@Configuration，可以进入此注解的定义

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Configuration {...}

可见它被@Component注解了，因此@Configuration注解有@Component的所有特性，意味也将会注册为bean, 也可以依赖注入。

被@Configuration注解的类内部包含有一个或者多个被@Bean注解的方法，这些方法将会被ApplicationContext上下文类进行扫描，并构建对应的bean,加入到Spring容器之中进行管理。

@Configuration

public class TestConfiguration {

@Bean

public Bean getBean() {

return new Bean();

}

}

### @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

@ComponentScan的主要用法是：

1. 扫描路径下边带有@Component注解的类加入spring容器（@Controller，@Service，@Repository等被@Component注解的注解也是）

例：@ComponentScan(value = "com.demo.controller")

会扫描com.demo.controller包下带有@Component注解的类加入spring容器

如果只写@ComponentScan后面不指定特定的包，则会扫描此@ComponentScan注解所在包下的带有@Component注解的类加入spring容器。

1. 通过includeFilters来按照规则只包含某些包的扫描。

@ComponentScan(value = "com.demo",

includeFilters = {@Filter(type = FilterType.ANNOTATION, classes = {Controller. class})},useDefaultFilters = false)

public class MyBeanConfig {...}

以上，指定被 @Controller 注解的类才会被注册到容器中。includeFilters的参数是一个 Filter[] 数组，然后指定 FilterType 的类型为 ANNOTATION，也就是通过注解来过滤，最后的 value 则是Controller 注解类。配置之后，在 spring 扫描的时候，就会将 com.demo 包下，所有被 @Controller 注解标注的类加入容器。这里的useDefaultFilters的属性要设置为false。因为useDefaultFilters 的默认值是true,它的作用是会自动发现被 @Component、@Repository、@Service 和 @Controller 标注的类，所有这里把useDefaultFilters 的值设为false来关闭这个功能。

3.通过excludeFilters过滤出不用加入spring容器的类

用excludeFilters可以排除来按照规则排除某些包的扫描，它的用法和includeFilters 类似。

@ComponentScan(value = "com.demo",

excludeFilters = {@Filter(type = FilterType.ANNOTATION,

value = {Controller.class})})

public class MyBeanConfig {

在 spring 扫描的时候，就会跳过 com.demo 包下，所有被 @Controller 注解标注的类。

### @Conditional

@Conditional的作用是根据某个条件来决定是否创建Bean，通过实现Condition接口，并重写matches接口来构造判断条件，我们可以利用这个特性进行一些自动的配置。

下面将以不同的操作系统为条件，通过实现Condition接口，并重写其matches方法来构造判断条件。若在Windows系统下运行程序，则注入cat bean；若在Linux系统下运行程序，则注入dog bean。

1.创建Dog和Cat类

public class Cat {

}

public class Dog {

}

2.创建MyBeanConfig 配置类，来根据条件注入不同的bean

@Configuration

public class MyBeanConfig {

@Conditional(WindosCondition.class)

@Bean("cat")

public Cat getCat(){

return new Cat();

}

@Conditional(LinuxCondition.class)

@Bean("dog")

public Dog getDog(){

return new Dog();

}

}

3.创建LinuxCondition 和WindosCondition

public class LinuxCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata){

return context.getEnvironment().getProperty("os.name").contains("Linux");

}

}

public class WindosCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata){

return context.getEnvironment().getProperty("os.name").contains("Windows");

}

}

4.在测试类种创建一个测试方法，来看有注入了哪些bean

@Test

public void beanConditionTest(){

AnnotationConfigApplicationContext annotationConfigApplicationContext = new Annotati onConfigApplicationContext(MyBeanConfig.class);

String[] beanDefinitionNames = annotationConfigApplicationContext.getBeanDefinitionNa mes();

for (String beanDefinitionName: beanDefinitionNames) {

System.out.println(beanDefinitionName);

}

String osName = annotationConfigApplicationContext.getEnvironment().getProperty("os.n ame");

System.out.println("当前系统为：" + osName);

}

可以看到注入了cat这个bean:

beancondition

## 了解自动配置

在springboot启动的时候,加载主配置类过程中,开启了自动配置的功能.@EnableAutoConfiguration。

@SpringBootApplication是一个组合注解：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),

@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {...}

主要起作用的就是@EnableAutoConfiguration注解

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage

@Import(AutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {...}

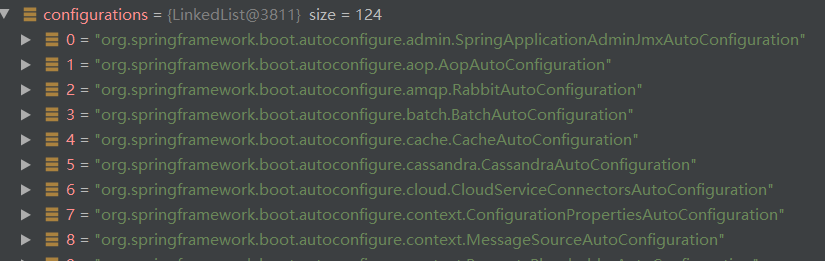
这里利用AutoConfigurationImportSelector给容器中导入一些,自动配置的组件。

看到AutoConfigurationImportSelector中的getCandidateConfigurations(annotationMetadata, attributes)方法，

这个方法中的SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),getBeanClassLoader())方法会扫描所有jar包类路径下 META-INF/spring.factories

把扫描到的这些文件的内容包装成properties对象，再从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中。

在SpringFactoriesLoader.loadFactoryNames(...)处打一个断点，再以debug模式启动程序可以看到：



类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值被加入到了容器中，这些用他们来做各种功能的自动配置。

这样各种XxxAutoConfiguratio被加载到了容器中，它们会向容器中加入各种功能的组件。

下面以HttpEncodingAutoConfiguration来阐述XxxAutoConfiguratio是怎么进行自动配置的。

@Configuration(proxyBeanMethods = false)

//将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中

@EnableConfigurationProperties(HttpProperties.class)

//判断当前应用是否是web应用，如果是，当前配置类生效

@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)

//判断应用中是否有CharacterEncodingFilter类,如果有 ,当前配置类生效

@ConditionalOnClass(CharacterEncodingFilter.class)

//判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的（matchIfMissing = true的作用）

@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true)

public class HttpEncodingAutoConfiguration {

private final HttpProperties.Encoding properties;

public HttpEncodingAutoConfiguration(HttpProperties properties) {

this.properties = properties.getEncoding();

}

@Bean//给容器中添加一个组件，此组件的一些属性的值可以从properties中获取

@ConditionalOnMissingBean///若容器中无组件CharacterEncodingFilter，这个bean才加入容器

public CharacterEncodingFilter characterEncodingFilter() {

CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();

filter.setEncoding(this.properties.getCharset().name());

filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));

return filter;

}

...

}

上面这个组件中属性的值可以从properties中获取，这个properties是开头@EnableConfigurationProperties(HttpProperties.class)中引入的HttpProperties的HttpProperties.Encoding属性注入的，而HttpProperties中的属性可以和配置文件中的值相绑定。这样配置文件中的值就注入到了这些组件的属性中。

Spring Boot启动时会加载很多自动配置类，这些自动配置类向容器中加入了很多功能组件。当我们需要使用这些组件时，可以查看相应的自动配置类，看看这些组件有哪些功能。当你要指定组件中属性的值时，可以查看此配置类绑定的XxxProperties类，然后再配置文件中按照 XxxProperties类的属性来进行配置。

## HelloWorld程序解析

### 依赖引入

#### 父项目

#### 启动器

### 程序入口

# 配置文件

## YAML简介

### 基本语法

### 数据类型

## 配置文件加载顺序

## Spring Boot基本配置介绍

## 自动配置原理

# 日志

## 日志简述

### 日志的抽象层

### 日志的实现层

## SpringBoot使用的日志-SLF4j

### SpringBoot日志依赖关系

### 使用SLF4j

### 更改日志配置

## 更改日志框架

# SpringBoot Web开发

## Spring核心技术

### 控制反转IOC

#### Bean属性

#### 基于注解的依赖注入

### 面向切面AOP

#### 注解式AOP开发

## SpringMVC

### SpringMVC架构

### 转发与重定向

### 拦截器

### 类型转换器

### 异常处理

### SpringMVC的自动配置

## Thymeleaf模板引擎

### Thymeleaf简介

### 引入Thymeleaf

### Thymeleaf语法规则

### Thymeleaf应用实例

## SpringBoot对JSP的支持

# 第七章 SpringBoot 数据访问

## Mybatis

### 简述Mybatis

Mybatis是当前Java Web开发中流行的持久化ORM框架，它对JDBC进行了封装和简化，无需JDBC的注册驱动，创建Connection连接，配置Statement等繁琐过程，大大减少了JDBC代码，使得开发者只需专注于SQL语句设计即可。Mybatis通过内部机制将Java类（对象）持久化为数据库表中的记录，反之也可以将数据库中的记录转化为Java类（对象）。在三层架构开发中，Mybatis作用在数据访问层，它让数据访问层的开发变得简单，高效。

7.1.2 ORM框架原理

Java程序常常要连接并操作数据库，但是两者的数据类型往往并不匹配，Java是面向对象的语言，Java语言中操作的单元是类与对象，而数据库的数据格式是关系类型。为了匹配Java面向对象与关系型数据库的数据类型，人们发明了ORM框架（Object Relational Mapping，对象关系映射）,用于将Java中的对象映射成数据库中的记录，对象中的属性映射为数据库中的字段，程序员可以使用面向对象的编程方式来操作数据库。例如，在程序代码中添加一个对象，则数据库中相应添加一行记录，在程序代码中删除一个对象，则数据量中相应删除了一条记录。比较常见的ORM框架有Hibernate和Mybatis。

ORM框架的原理如图7.1所示。其中，CRUD是在做计算处理时的增加（Create），读取查询（Retrieve），更新（Update）和删除（Delete）4个单词的首字母缩写，POJO(Plain Ordinary Java Object)为简单的Java对象，实际就是普通的JavaBean。

数据库表

ORM框架

POJO对象

Java程序

图7.1

### SpringBoot使用Mybatis

此处通过演示一个实例，在MySQL数据中的user表中增加和查询用户信息。同时为了简化实例，省略了service层，将具体操作整合在了Controller中。

1.导入Mybatis相关依赖

<dependency>

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

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

<version>x.x.x</version>

</dependency>

MyBatis-Spring-Boot-Starter依赖将会提供：自动检测现有的DataSoure，创建并注册SqlSessionFactory的实例，该实例使用SqlSessionFactoryBean将该DataSource作为输入进行传递，创建并注册从SqlSessionFactory中获取的SqlSessionTemplate的实例。自动扫描项目的mappers，将它们连接到SqlSessionTemplate并将其注册到Spring上下文，以便将它们注入到项目的bean中。使用了该Starter之后，只需要定义一个DataSource即可（application.properties或application.yml中可配置），它会自动创建使用该DataSource的SqlSessionFactoryBean以及SqlSessionTemplate，也会自动扫描项目的Mappers，连接到SqlSessionTemplate，并注册到Spring上下文中。

2.在application.yml中配置数据库连接信息

数据库配置

spring:

datasource:

username: root

password: 123456

url:jdbc:mysql://localhost:3306/springboot?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8

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

整合mybatis

mybatis:

type-aliases-package: com.example.pojo

mapper-locations: classpath:mybatis/mappper/\*.xml

mapper-locations: 是指mapper的位置，也就是.xml文件的位置，如果没有配置，就会默认去mapper接口类所在的包中查找。但是注意，如果在编译之后，Java文件夹和resources文件夹的文件都会到同一个包中。

type-aliases-package:实体类的别名。这个包下面的实体类都可以使用别名，例如：

com.example.pojo.User 可以用User代替

3.在com.example.pojo包下创建一个实体类User

package com.example.pojo;

public class User {

private int id;

private String name;

private String gender;

//省略了getter，setter方法

}

4.创建mapper包以及和实体类对应的Usermapper接口

package com.example.mapper;

@Mapper//这个注解表示这是一个mybatis 的mapper类，不可省略。

@Repository

public interface UserMapper {

List<User> queryUserList();

User queryUserById(int id);

int addUser(User user);

int deleteUser(int id);

int updateUser(User user);

User selectUserById(int id);

}

5.编写mapper接口的映射文件

Mapper接口的映射文件可以和mapper接口放在同一个包中，但是为了方便管理，可以将映射文件放到resources资源文件夹中，在resources目录下新建一个mapper文件，然后新建一个UserMapper.xml文件。

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

<!DOCTYPE mapper

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

命名空间

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

<mapper namespace="com.example.UserMapper">

SQL语句查询结果映射为User类型。注意，如果在整合Mybatis时没有设置别名，则需要写全称类名

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

select \* from user

</select>

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

select \* from user

</select>

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

select \* from user where id = #{id}

</select>

<insert id="addUser" parameterType="User">

insert into user (id,name,gender) values (#{id},#{name},#{gender})

</insert>

<update id="updateUser" parameterType="User">

update user set name=#{name},pwd=#{gender} where id = #{id}

</update>

<delete id="deleteUser" parameterType="int">

delete from user where id = #{id}

</delete>

</mapper>

映射文件主要实现SQL语句与Java对象之间的映射，使得SQL语句查询出来的关系型数据能够封装成Java对象。映射文件通常用POJO+Mapper命名。

映射文件中的一级标签<mapper></mapper>里面可以包含多对<select>标签或者<insert>等其他标签。<mapper>标签的namespace属性，用于标识映射文件，通常其值设置成对应接口的全路径名称。

6.在controller包中编写UserController类，

package com.example.controller;

@RestController

public class UserController {

@Autowired

private UserMapper userMapper;

//查询所有用户

@GetMapping("/queryUserList")

public List<User> queryUserList(){

List<User> usersList = userMapper.queryUserList();

for (User user:usersList){

System.out.println(user);

}

return usersList;

}

//根据id选择用户，并在控制台输出

@GetMapping("/selectUserById")

public void selectUserById(){

User user = userMapper.selectUserById(5);

System.out.println(user);

}

//添加一个用户

@GetMapping("/addUser")

public String addUser(){

userMapper.addUser(new User(5,"小白","女"));

return "ok";

}

}

在浏览器中现后输入http://localhost:8080/addUser和http://localhost:8080/selectUserById。可以在控制台看到如图7.2的内容



图7.2

### Mybatis注解方式开发

Springboot整合Mybatis注解方式开发，可以减少让我们的开发变得更加简单，不需要去配置Mapper.XML，。本节将用一个实例来解释如何在springboot使用mybatis注解开发，为了简便，省略了service层。

注意：注解方式开发发无需添加新的依赖。

1. 在application.yml中配置数据源信息

数据库连接信息和上一节完全相同，此处不再赘述。

整合mybatis

mybatis:

type-aliases-package: com.example.pojo

~~mapper-locations: classpath:mybatis/mappper/\*.xml~~

此处不再需要配置mapper-locations，因为在注解方式开发下不再需要对应mapper.xml

1. 新建一个pojo包并新建一个User类

public class User {

private int id;

private String name;

private String gender;

public User() {

}

public User(int id, String name, String gender) {

this.id = id;

this.name = name;

this.gender = gender;

}

//省略getter（）和setter方法

}

1. 创建mapper包以及和实体类对应的Usermapper接口

@Mapper

@Repository

public interface UserMapper {

//查询所有用户

@Select("select id,name,gender from user")

List<User> getAllUser();

//根据id查询用户

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

User selectUserById(Integer id);

//添加一个用户

@Insert("insert into user(id,name, gender) values(#{id},#{name}.#{gender})")

int addUser(User user);

}

mapper类中在对应的方法中添加@select()，@Insert()，@update()，@delete()的注解，将SQL语句写入到对应的注解中，就不需要去\*\*\*mapper.xml映射文件中配置SQL等信息了。

1. 在controller包中编写UserController类

@RestController

public class UserController {

@Autowired

private UserMapper userMapper;

@GetMapping("getAll")

public List<User> getAllUser(){

List<User> userList = userMapper.getAllUser();

return userList;

}

@GetMapping("/addUser")

public void addUser(){

userMapper.addUser(new User(6,"小红","女"));

}

@GetMapping("selectUserById")

public User selectUserById(){

User user = userMapper.selectUserById(6);

return user;

}

}

Controller层和上一节编写方式没有区别。

1. 测试

运行程序，然后在浏览器先后输入localhost:8080/addUser和localhost:8080/ selectUserById 可以看到如图7.3所示内容



图7.3

至此测试成功。

## MongoDB

### 简述

MongoDB 是由C++语言编写的，是一个**基于分布式文件存储**的开源数据库系统。它旨在为WEB应用提供可扩展的高性能数据存储解决方案，MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组。它作为一种NoSQL存储介质，存储读取快，但实质上它又是一种介于关系型数据库（如mysql，数据存在磁盘中）和非关系型数据库（如redis，数据存在内存中）的介质，它数据存在磁盘，但读取又借助内存机制映射进行，所以集成了关系型和非关系型的各自优点。

### 配置MongoDB

从官方网站[https://www.mongodb.com](https://www.mongodb.com/download-center" \l "community) 中下载MongoDB。此处演示下载的是MongoDB4.2版本。安装后目录如下：

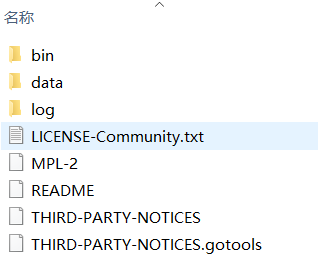


图7.2.1 MongoDB文件结构

配置并从配用路径开启：

在安装目录\data\下创建一个db目录，作为安装目录\data\的备用目录，在log文件夹下创建新文件 mongo.config(安装目录\log\mongo.config)

在启动时，使用的是安装目录\bin\下的mongod.exe，如果直接启动，实际使用的是C:\data\db\为数据库文件存储目录，需要确保该目录存在。

可以使用--dbpath "安装目录\data\db"显式指定该目录为存储位置，来启动MongoDB，如：

D:\ProgramFiles\MongoDB\Server\4.2\bin>mongod.exe

--dbpath "D:\Program Files\MongoDB\Server\4.2\data\db"

测试是否成功启动服务，MongoDB默认的监听端口是：27017。在浏览器中输入<http://localhost:27017/，若页面出现> It looks like you are trying to access MongoDB over HTTP on the native driver port. 则表明服务已经启动

测试使用：使用安装目录\bin\下的mongo.exe 启动MongoDB的操作终端：

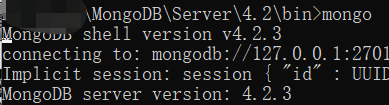


图7.2.2操作演示

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>

测试使用成功。

### 基本用法

创建数据库：use database\_name

实例：

> use first

switched to db firstdb

查看当前正在使用的数据库：db

> db

firstdb

>

查看所有数据库：show dbs

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>

此时新建的firstdb不在数据库的列表中，只有当集合内容插入到数据库firstdb中才会正在被创建。

注意：刚开始创建的数据库是存在内存中的，真正创建后数据库被存在磁盘中。 MongoDB 中默认的数据库为 test，如果你没有创建新的数据库，集合将存放在 test 数据库中。

删除数据库：db.dropDatabase()

删除当前数据库，默认为 test，可以先使用 db 命令查看当前数据库名，再决定是否执行删除命令。

集合的创建

当向一个集合中插入内容的时候，如果集合不存在则为自动创建。

此外，也可以使用显式创建集合的方式

基本语法格式：db.createCollection(name，options)

参数说明：

name ：要创建的集合名称

options: 可选参数, 指定有关内存大小及索引的选项

删除集合：db.collection.drop()

collection就是我们想删除集合名字，执行完命令后会出现返回值：

删除成功，返回值是true；否则返回false

例如：删除firstacollection集合

执行：db.firstcollection.drop()

查看当前数据库中的表：show collections或show tables

插入文档

注意：存储再集合中的数据都是bson格式，bson数据结构和json基本一致。

语法：db.collection\_name.insert(document) 或者 db.col.save(document)

document为所所要插入的文档内容，格式是bson。

更新文档

MongoDB 使用 update() 和 save() 方法来更新集合中的文档，

update()方法

语法格式：db.collection\_name.update(

<query>,

<update>,

{

upsert: <boolean>,

multi: <boolean>,

writeConcern: <document>

}

)

参数说明：

query : update的查询条件，类似sql update查询内where后面的。

update : update的对象和一些更新的操作符（如$,$inc...）等，可以理解为sql update查询内set后面的

upsert : 可选，它表示如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。

multi : 可选，mongodb 默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

writeConcern :可选，抛出异常的级别。

save方法

此方法通过传入文档来替换已有文档。

语法格式：db.collection\_name.save(

<document>,

{

writeConcern: <document>

}

)

参数说明：

document : 文档数据。

writeConcern :可选，抛出异常的级别

删除文档

删除文档使用remove函数

db.collection\_name.remove(

<query>,

{

justOne: <boolean>,

writeConcern: <document>

}

)

参数说明：

query :（可选）删除的文档的条件。

justOne : （可选）如果设为 true 或 1，则只删除一个文档，若不设置该参数，或使用默认值 false，则删除所有匹配条件的文档。

writeConcern :（可选）抛出异常的级别。

若要删除所有数据，则执行以下操作

db.collection\_name.remove({ })

查询文档

查询文档使用find()方法，此方法以非结构化的方式来显示所有文档。

语法格式：db.collection\_name.find(query, projection)

参数说明：

query ：可选，使用查询操作符指定查询条件

projection ：可选，使用投影操作符指定返回的键。查询时返回文档中所有键值， 只需省略该参数即可（默认省略）。

### SpringBoot使用MongoDB

Spring Boot中可以通过在pom.xml中加入spring-boot-starter-data-mongodb引入对mongodb的访问支持依赖

<dependency>

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

配置数据源，在application.yml中配置

本地安装的mongodb，没有用户名和密码

spring:

data:

mongodb:

uri: mongodb://localhost:27017/test

有用户名和密码的写法

spring:

data:

mongodb:

uri: mongodb://name:password@localhost:27017/test

MongoDB存储的是bson数据，所以需要创建一个实体类与MongoDB存储的bson数据形成对应，即bson转成实体对象，实体对象可以转换成bson的中间数据模型。

此处创建一个MongodbTest作为实体类

public class MongodbTest {

private Integer id;

private Integer age;

private String name;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public String getName() {

return name;

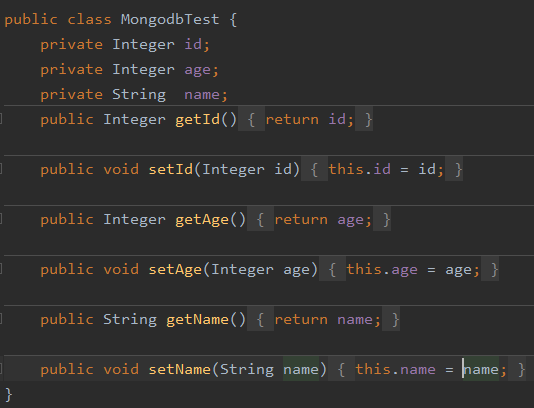
}

public void setName(String name) {

this.name = name;

}

}



在dao层中创建一个MongodbTestDao类

@Component

public class MongodbTestDao {

@Autowired

private MongoTemplate mongoTemplate;

/\*

创建对象

\*/

public void saveTest(MongodbTest test){

mongoTemplate.save(test);

}

/\*\*

\* 根据用户名查询对象

\* @param name

\* @return

\*/

public MongodbTest findTestByName(String name){

Query query = new Query(Criteria.where("name").is(name));

MongodbTest mgt = mongoTemplate.findOne(query,MongodbTest.class);

return mgt;

}

}

此处作为一个简单实例，省略了service层，相关操作在Controller中进行。

在Controller层中新建MongodbForTest类

@RestController

public class MongodbForTest {

@Autowired

private MongodbTestDao mtdao;

@RequestMapping(value = "/test1")

public void saveTest() throws Exception{

MongodbTest mgtest = new MongodbTest();

mgtest.setId(2);

mgtest.setAge(33);

mgtest.setName("Jony");

mtdao.saveTest(mgtest);

}

@RequestMapping(value = "/test2")

public MongodbTest findTestByName(){

MongodbTest mgtest = mtdao.findTestByName("Jony");

System.out.println("mgtest is"+mgtest);

return mgtest;

}

}

运行程序，再浏览器输入http://localhost:8080/test1

结束后,重新输入<http://localhost:8080/test2>，可以看到浏览器中出现下面的数据：

{"id":2,"age":33,"name":"Jony"}

则表明程序执行成功。

## Redis

### Redis简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。它支持多种类型的数据结构，如字符串（strings），散列（hashs），列表（lists），集合（sets），有序集合（sorted sets）。这些数据类型都支持push，pop，add，remove以及取交集并集和差集等丰富的操作。在此基础上，Redis支持各种不同方式的排序。

Redis支持数据的持久化同时也具有很高的效率，它的数据都是缓存在内存中，同时会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件。它的操作都是原子性的，所谓原子性就是指要么成功执行操作，要么失败完全不执行操作。它也有丰富的特性，可以用于缓存，实现发布/订阅机制，按key设置过期时间，过期后将会自动删除。

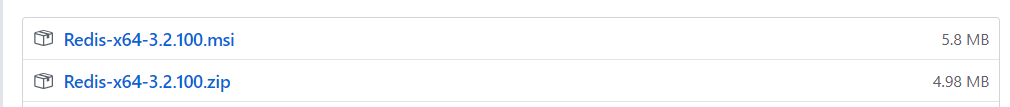
不过需要注意的是，Redis运行在内存中，所以在对不同数据集进行读写时需要权衡内存的大小，因为数据量不能大于内存。

### Redis安装与配置

1.window10上安装Redis

可从Redis官网：<https://redis.io/> 上下载对应的版本，

或者从github：https://github.com/MicrosoftArchive/redis/releases下载。Redis支持32位和64位，根据自身系统平台进行选择。



msi后缀的文件是安装版，zip后缀的文件是解压版。

此处下载msi版本，下载后打开文件，一路点击next，直到出现图7.3.1

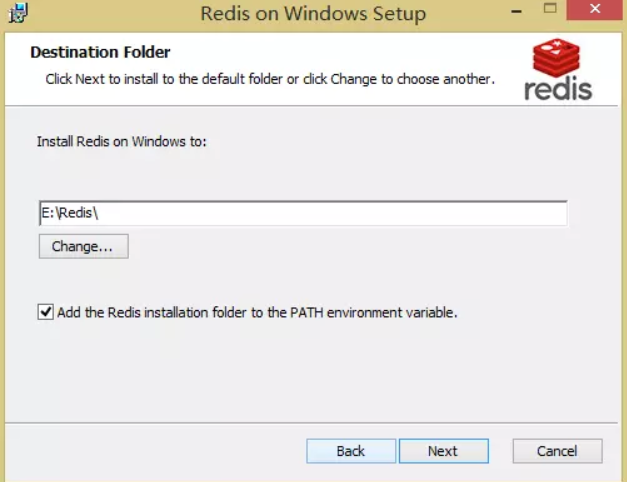


图7.3.1

选择文件安装位置，勾选上添加安装目录到 PATH 环境变量的选项。继续点击next，出现图7.3.2



图7.3.2

选择端口，Redis默认是6739端口，若不想用6379或者6379端口已经被占用，可以修改成其他端口。下方是添加到windows防火墙的选项，默认勾选。继续点击next。出现图7.3.3

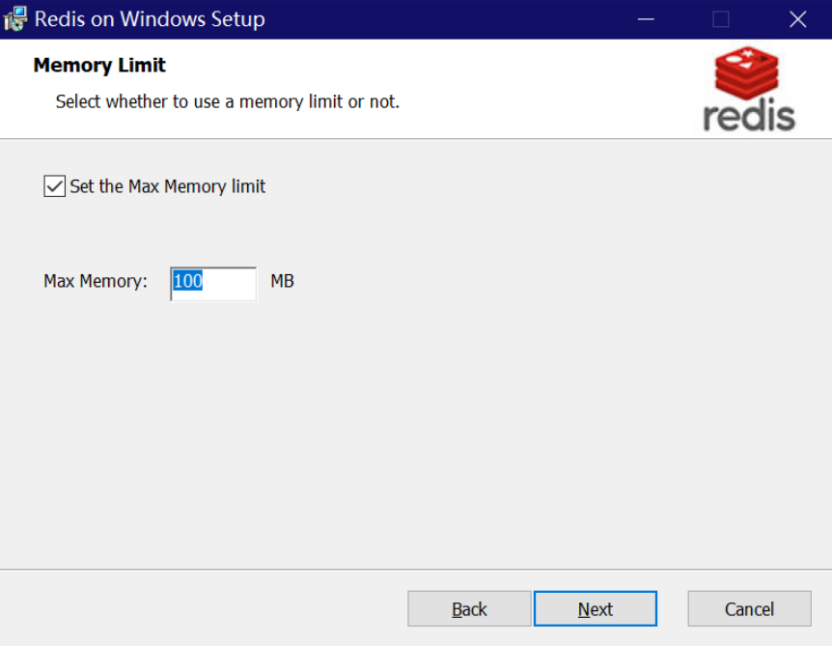


图7.3.3

此页面是设置Redis最大内存限制，默认是100MB内存，根据需要可以修改大小。继续点击next，然后点击Install，等到片刻，点击finsh，安装完成。

2.测试安装的Redis

若是采用msi文件的安装

第一种方式，可以从计算机管理->服务与应用->服务 查看Redis是否正在运行。如图7.3.4

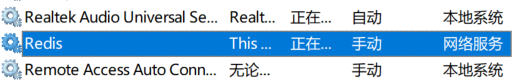


图7.3.4

若正在运行，则说明Redis服务启动成功。

第二种方式，可以在命令行窗口进入到安装路径的根目录。

输入命令redis-server redis.windows.conf，若出现图7.3.5所示内容，则说明服务启动成功。

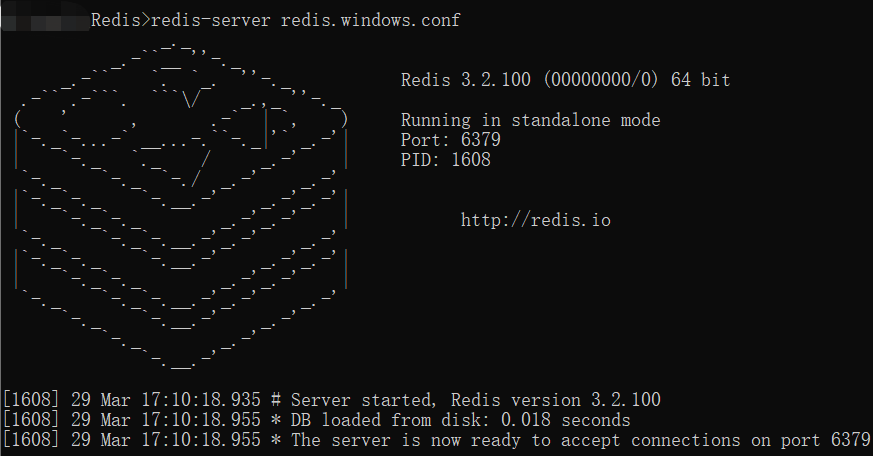


图7.3.5

测试使用（注意必须在启动服务后才能使用），可以从Redis的安装根目录中找到redis-cli.exe文件，点击启动。设置键值对，进行测试。例如图7.3.6

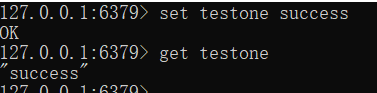


图7.3.6

至此，测试成功。

### Redis数据类型

1.字符串（string）类型

字符串类型是 Redis 中最基本的数据类型，它能存储任何形式的字符串，包括二进制数据，序列化后的数据，JSON 化的对象甚至是一张图片。

|  |  |
| --- | --- |
| key | value |
| Age | 18 |

2.哈希（hash）类型

hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。Redis 中每个 hash 可以存储 232 - 1 键值对。

|  |  |
| --- | --- |
| key | user |
| field | value |
| name | 小明 |
| age | 18 |

3.列表（list）类型

Redis 列表是简单的字符串列表，按照插入顺序排序，可以通过添加一个元素到列表的头部（左边）或者尾部（右边）。

|  |  |
| --- | --- |
| key | value |
| students | 小明 小红 小王 |

4.集合（set）类型

Redis 的 集合类型 是 string 类型的无序集合，集合成员是唯一的，即集合中不能出现重复的数据。

|  |  |
| --- | --- |
| Key | Value |
| Students | 小明 |
| 小红 |
| 小王 |

6.有序集合（sorted set）类型

Redis 有序集合和集合一样也是 string 类型元素的集合，且不允许重复的成员。不同的是 有序集合的每个元素都会关联一个分数（分数可以重复），redis 通过分数来为集合中的成员进行从小到大的排序。

|  |  |
| --- | --- |
| Key | Value |
| Students | 小明 5000 |
| 小红 6000 |
| 小王 7000 |

### Redis命令

1.基本操作命令

redis 默认为 16 个库 ，redis 默认自动使用 0 号库。

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| **ping** | Pong | 检查Redis服务状态，返回pong表示服务运行正常 |
| **dbsize** | 数字 | 查看当前数据库中 key 的数目,返回值表示当前数据库中key的数量 |
| **select** db | OK | 切换数据库命令 |
| **flush** db | OK | 删除当前库的数据 |
| **exit**或**quit** |  | 退出当前Redis连接 |

2.key的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| keys **pattern** |  | 查找所有符合模式pattern的key。 |
| **exists** key[key…] | 存在key返回1，否则返回0,使用多个key，返回存在key的数量。 | 判断key是否存在。 |
| **expire** key seconds | 设置成功返回数字 1，否则返回1。 | 设置 key 的生存时间，超过时间，key 自动删除，单位是秒。 |
| **ttl** key | Key永不过期返回-1，key不存在返回-2，其他数字表示key的剩余时间 | 查看key的剩余生存时间 |
| **type** key | 字符串表示的数据类型 | 查看key所存储值的数据类型 |
| **del** key [key…] | 数字，表示删除的key的数量 | 删除存在的key，不存在的key忽略 |

3.字符串类型（string）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| set key value |  | 将字符串值value设置到key中 |
| get key | 返回key对应的value | 获取key中设置的字符串值 |
| incr key | 数字 | 将key中存储的数据值加1，若key不存在，则key先被初始化为0，再加1，注意：只能对数字类型的数据操作 |
| decr key | 数字 | 将key中存储的数据值减1，若key不存在，则key先被初始化为0，再减1，注意：只能对数字类型的数据操作 |
| append key value | 追加字符串之后的总长度 | 如果 key 存在，则将 value 追加到 key 原来值的末尾；如果 key 不存在，则将 key 设置值为 value |
| strlen key | 若key存在，返回字符串值的长度，否则返回0 | 查询key所储存的字符串值的长度 |
| getrange key start end | 截取的自字符串 | 获取 key 中字符串值从 start 开始 到 end 结束 的子字符串,包括 start 和 end, 负数表示从字符串的末尾开始， -1 表示最后一个字符 |
| setrange key offset value | 修改后的字符串的长度 | 用 value 覆盖key 的存储的值从 offset 开始,不存在的 key 做空白字符串。 |
| mset key value [key value…] | OK | 同时设置一个或多个 key-value 对返回值 |
| mget key [key …] | 包含所有 key 的列表 | 获取所有(一个或多个)给定 key 的value |

4.哈希类型（hash）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| hset hash key field value | 若field是hash表中的新field，设置成功返回1；若field已经存在，旧值覆盖新值，返回0 | 将哈希表 key 的域 field 的值设为 value ，如果 key 不存在，则新建 hash 表，执行赋值，如果有 field ,则覆盖值。 |
| hget key field | 若存在，则返回field域的值；若key或者field不存在，则返回nil | 获取哈希表中key给定域field的值 |
| hmset key field value [field value…] | 设置成功返回OK；失败返回错误 | 将多个 field-value (域-值)设置到哈希表 key 中，会覆盖已经存在的 field，若hash 表 key 不存在，创建空的 hash 表，执行 hmset |
| hmget key field [field…] | 返回与field顺序对应的值；若相应位置的field不存在,返回nil | 查询哈希表 key 中一个或多个给定域的值 |
| hgetall key | 以列表形式返回 hash 中域和域的value ；若key 不存在，返回empty list or set | 查询哈希表key中的所有域和值 |
| hdel key field [field…] | 成功删除的field的数量 | 删除哈希表 key 中的一个或多个指定域 field |
| hkeys key | 包含所有 field 的列表，key 不存在返回空列表 | 查看哈希表key中所有field |
| hvals key | 哈希表中所有域的值；key不存在则返回空列表 | 查询哈希表中所有域的值 |
| hexists key field | 若存在返回值是1；否则返回0 | 查看哈希表key中，给定域field是否存在 |

6.列表（list）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| lpush key value [value…] | 新列表的长度 | 将一个或多个 value 插入到列表 key 的表头（最左边），从左到右的顺序依次插入到表头 |
| rpush key value [value…] | 新列表的长度 | 将一个或多个 value 插入到列表 key 的表尾（最右边），从左到右的顺序依次插入到表尾 |
| lrange key start stop | 指定区间的列表 | 获取列表 key 中指定区间内的元素 |
| lindex key index | 指定下标的元素；index不在列表范围内，返回nil | 查询列表 key 中下标为指定 index 的元素 |
| llen key | 列表的长度；key不存在返回0 | 查询列表key的长度 |
| lrem key count value | 移除的元素的个数 | 移除列表中与参数 value 相等的元素， count >0 ，从列表的左侧向右开始移除； count < 0 从列表的尾部开始移除；count = 0 移除表中所有与 value 相等的值。 |
| lset key index value | 设置成功返回OK，否则返回错误信息 | 设置key下表为index的元素的值为value |
| linsert key BEFORE|ALFTER pivot value | 若命令成功执行，返回新列表的长度；若不存在pivot，返回-1；若key不存在，返回0 | 将value插入到列表key中位于值 pivot 之前或之后的位置；若key不存或者pivot不在列表中，不执行任何操作。 |

7.集合类型（set）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| sadd key member [member…] | 加入到集合的新元素的个数 | 将一个或多个元素加入到集合key中，已经存在的元素被忽略 |
| smembers key | 集合中成员的列表 | 查询集合中所有成员元素，不存在的key被视为空集合 |
| sismember key member | 是集合的成员返回1；否则返回0 | 检查元素是否是集合里面的成员 |
| scard key | key的元素个数；不存在的key则返回0 | 查询集合里面的元素个数 |
| srem key member [member…] | 成功删除元素的个数 | 删除集合key中的一个或多个成员，不存在的元素被忽略 |
| srandmember key [count] | 一个元素或者多个元素的集合 | 返回集合中一个或多个随机元素 |
| spop key [count] | 被删除的元素；key不存在或者空集合则返回nil | 从集合中随机删除一个或者删除指定个数的元素, count 是删除的元素个数 |

8.有序集合类型（sorted set）的操作命令

|  |  |  |
| --- | --- | --- |
| 语法 | 返回值 | 作用 |
| zadd key score member [score member…] | 新添加元素的个数 | 将一个或多个元素及其score值加入到有序集合 key中 |
| zrange key start stop [WITHSCORES] | 自定区间的成员集合 | 查询有序集合内指定区间的元素 |
| zrevrange key start stop [WITHSCORES] | 自定区间的成员集合 | 查询有序集合key内指定区间的元素 |
| zrem key member [member…] | 删除成功的元素数量 | 删除有序集合key中的一个或多个成员 |
| zcard key | Key存在则返回集合元素的个数；key不存在则返回0 | 查询有序集合key的元素个数 |
| zcount key min max | 返回成员数量 | 查询有序集合key中，score值在min和max之间的成员数量； |

### SpringBoot使用Redis

要在Springboot项目中使用Redis，需要在pom.xml中添加spring-boot-starter-data-redis依赖，并在application.properties中配置相关属性，配置结束后就可以使用StringRedisTemplate模板类来操作Redis。StringRedisTemplate继承了RedisTemplate，它封装了对Redis的一些常用操作，但是只能对key和value都是String的键值对进行操作。

下面将通过一个实例来介绍在Spring Boot中如何使用Redis来实现储存字符串和对象。该项目是继续对上一个小节中项目的扩展。

1.在pom.xml中添加spring-boot-starter-data-redis依赖。

<!--Redis依赖-->

<dependency>

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

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

</dependency>

2.在application.properlies中添加Redis相关配置

# Redis数据库索引（默认为0）

spring.redis.database=0

# Redis服务器地址

spring.redis.host=127.0.0.1

# Redis服务器连接端口

spring.redis.port=6379

# Redis服务器连接密码（默认为空）

spring.redis.password=

# 连接池最大连接数（使用负值表示没有限制）

spring.redis.jedis.pool.max-active=-1

# 连接池最大阻塞等待时间（使用负值表示没有限制）

spring.redis.jedis.pool.max-wait=-1

# 连接池中的最大空闲连接

spring.redis.jedis.pool.max-idle=8

# 连接池中的最小空闲连接

spring.redis.jedis.pool.min-idle=0

3.新建Utils工具包并新建JsonUtils类

public class JsonUtils {

//定义Json对象

private static final ObjectMapper MAPPER=new ObjectMapper();

/\*\*

\* 将对象转换成Json字符串

\* @param data

\* @return

\*/

public static String objectToJson(Object data){

try {

String string = MAPPER.writeValueAsString(data);

return string;

}catch (JsonProcessingException e){

e.printStackTrace();

}

return null;

}

public static <T> T jsonToPojo(String jsonData,Class<T>beanType){

try {

T t = MAPPER.readValue(jsonData,beanType);

return t;

}catch (Exception e){

e.printStackTrace();

}

return null;

}

}

4.在controller层的UserController类中添加下面两个方法

@RequestMapping("/setUserPojo")

public void setUserPojo(){

User user = new User(10, "小明", "男");

stringRedisTemplate.opsForValue().set("jsonUser", JsonUtils.objectToJson(user));

}

@RequestMapping("/getUserPojo")

public User getUserPojo( ){ User user =

JsonUtils.jsonToPojo(stringRedisTemplate.opsForValue().get("jsonUser"),User.class);

return user;

}

5.运行程序

在浏览器中先输入<http://localhost:8080/setUserPojo>；

后输入<http://localhost:8080/getUserPojo>，会看到页面中有如下的内容：

{"id":10,"name":"小明","gender":"男"}

说明项目运行成功。

6.总结

Redis的使用场景十分丰富，以上的例子仅仅只是冰山一角，Redis可以用作全页缓存，会话缓存（Session Cache），快速计数，限速，消息队列，发布订阅等等。更多关于Redis的知识不再此作详细介绍。

# SpringBoot Actuator

## SpringBoot Actuator 简介

## 原生端点使用

## 定制 Actuator

# Spring Security

## Spring Security简介

## 配置Spring Security

## Spring Security入门

### 框架原理

### 核心组件

## 实例

# SpringBoot部署与测试

在前面的章节中我们已经能够生成一个完整的SpringBoot应用了，它将包括SpringBoot的基本环境和一些简单代码逻辑，我们的数据库也使用了目前主流的Mybatis，甚至我们还可以利用Redis缓存让我们的服务器运行更加高效。

万事开头难，我们已经顺利完成了开始的部分，但是虽然前面的铺垫十分精彩，但一部好的电影的结尾依然十分重要。没有它我们前面的努力都将化为乌有。

在本章，我们将在SpringBoot开发应用程序的基础上更进一步，讨论如何部署我们的应用程序，同时我们也将对比各种部署方式的优缺点让我们能更好地斟酌我们的部署方式。虽然以往的java程序部署方式没有太多的变化，但是SpringBoot的新特性将让我们与众不同一些。

## **传统java web应用的部署方式**

### 方式一：直接拷贝

为了方便读者理解，我们将用一个实例对各种方式进行实战演示操作。这个小例子是我初学时的简单作品，源码部分可以在本书末尾进行下载。下面我们开始方式一的学习。

顾名思义我们在完成一个项目的开发后，传统的java web工作目录下将出现web app目录或者web Content目录。本例子的目录结构如图10-1。

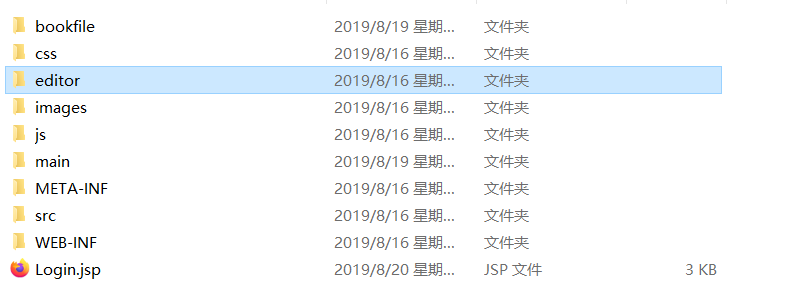
****

图 10-1 项目目录

我们可以看到该目录下已经有了我们需要的js文件，源码，以及各种css文件等等。部署的方式也十分简单：

1. 复制我们的web Content文件夹到我们的tomcat安装目录下的webapps下。如图10-2

我们将项目名称重命名为了”student” 。

****

图10-2 复制目录

2，完成复制后我们需要修改conf目录下的server.xml文件，在conf/server.xml文件中的Host标签下添加一个Context元素

<Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true">

<Context path="/student" docBase="yourpath" reloadable="true"></Context>

</Host>

path：该路径为刚才复制的文件名

docBace：该项目的物理地址全路径

修改完成后我们的访问的URL就为：http://localhost:8080/student，使用浏览器访问结果如图10-3：

****

图10-3 项目首页

然而一般来说，不建议通过server.xml来管理上下文，因为修改了Tomcat配置文件。不过，如果有多个应用程序需要测试，使用这种方式或许更为方便，因为可以在一个文件中同时管理所有的应用程序。

### 方式二：添加XML文件

在Tomcat的conf/Catalina/localhost目录下创建一个XML文件并修改它，

如果决定给每一个上下文都创建一个XML文件，那么这文件名就很重要了，因为上下文路径就是从这个文件名的衍生得到的。例如，将一个student.xml文件放Catalina/localhost目录下，那么这个应用程序的上下文路径就是student，访问的URL就为： http://localhost:8080/student这个上下文文件只有一行代码：

<?xml version="1.0" encoding="UTF-8"?><Context docBase="yourpath" reloadable="true"></Context> ****

图10-4

这里的docBase是必须的属性，它定义了应用程序的位置。reloadable属性是可选的，如果存在并且值为true，那么一旦程序中Java类文件或者是其他资源文件有任何添加，减少或者是更新，Tomcat都可以检测到，并且重新加载该应用程序

当把上下文文件添加到Tomcat的指定目录下，Tomcat就会自动加载应用程序。当删除这个文件时，Tomcat就会自动卸载应用程序（Eclipse和Idea都是利用这种方式部署项目的，所以去Tomcat部署目录中是找不到项目根目录及资源文件的，这些文件存在于IDE项目的工作空间中）。

### 方式三：隐式部署

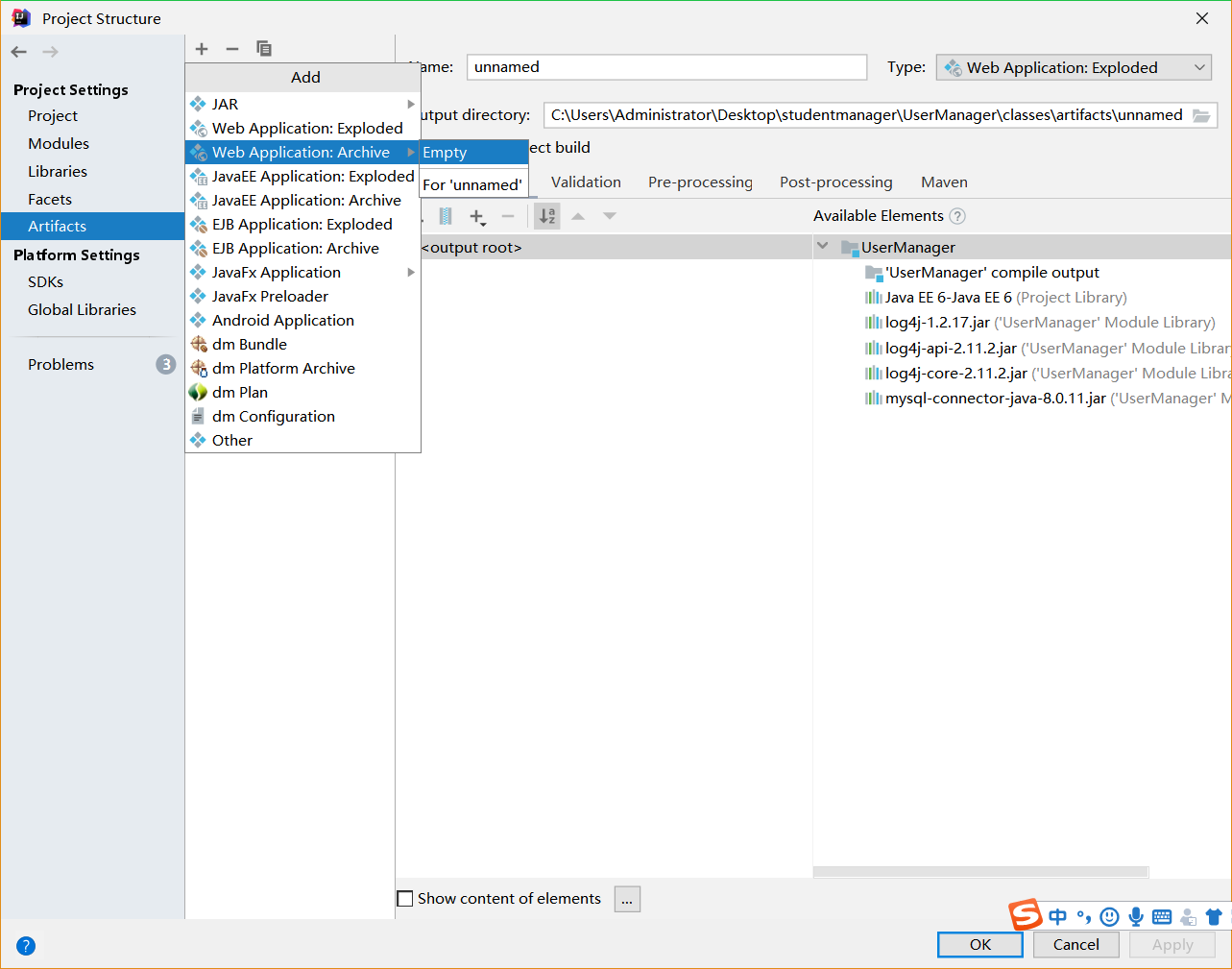
隐式部署相对于前面两种方式更加方便，因此推荐使用该方式。其主要方法是通过将一个war文件或者整个应用程序复制到Tomcat的webapps(Tomcat默认的部署项目位置，可以在server.xml文件中修改，但不建议修改)目录下，启动服务器之后就会解压这个war，解压到部署项目位置。（MyEclipse就是将项目根目录部署到Tomcat中，所以会在Tomcat中找到项目根目录及资源文件）。

让我们来看看部署的细节部分：

1. 打包项目为war包（以idea为例）

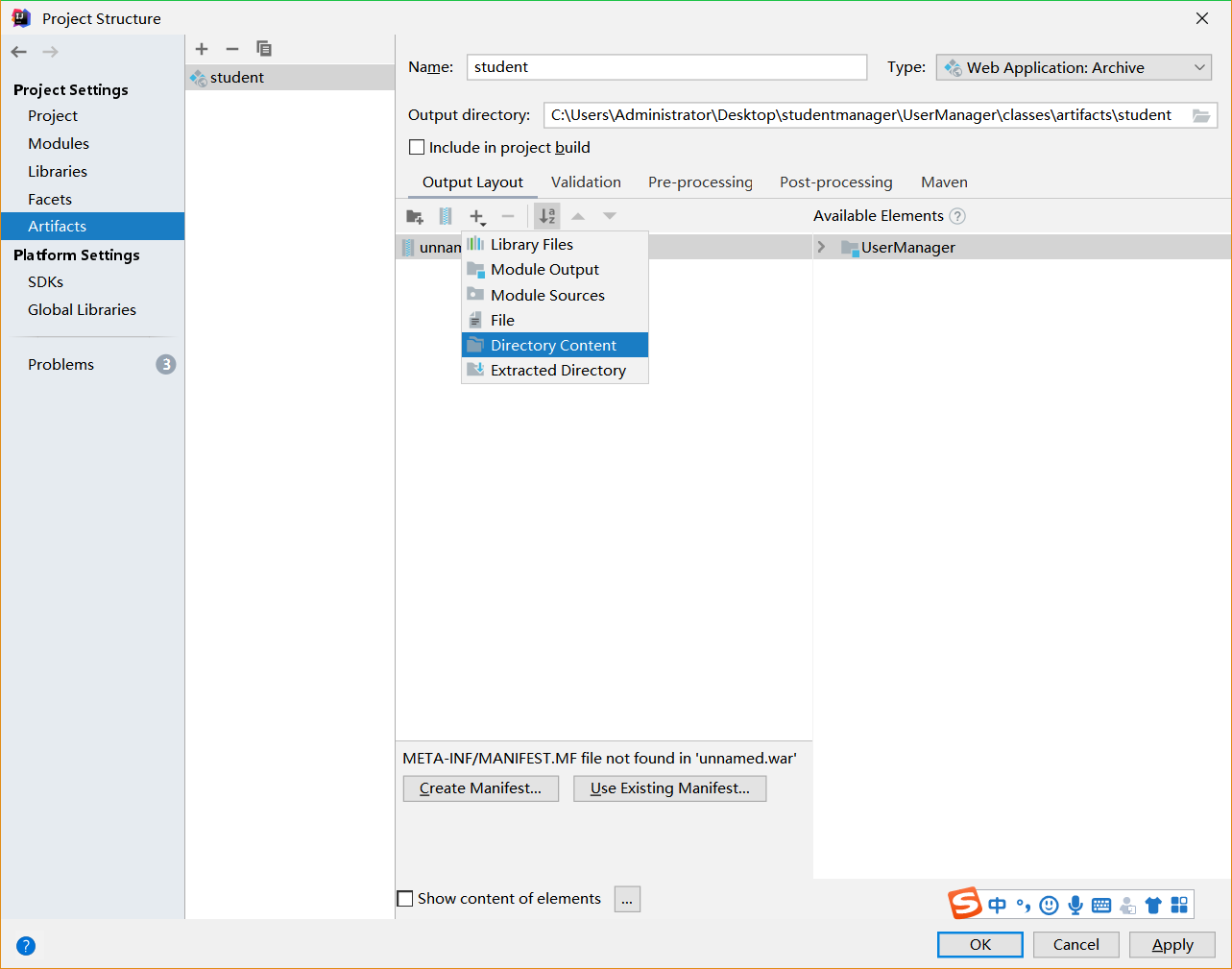
我们将使用idea的Artifacts 来对项目进行打包，在我们已经在idea中打开并配置好其他环境后，我们才可以进行下面的操作。

1. 依次打开File -> Project Structure ->Artifacts ，并点击“+”号添加新的Artifact 如图10-5所示

****

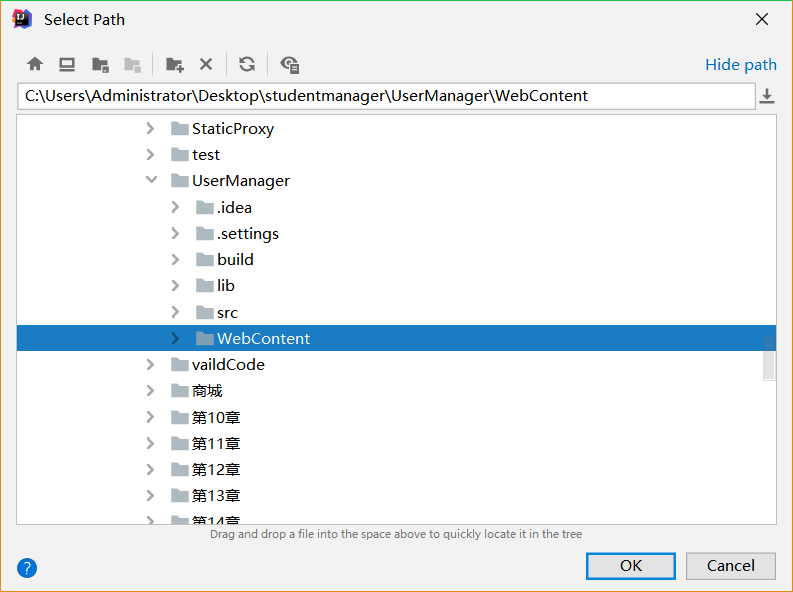
**图10-5 添加Artifact**

1. 点击Empty完成添加Artifact，接着我们需要继续为该Artifacts 添加目录（不然我们打包后的war文件都不知道放在哪里了）我们一般添加的目录为webapp目录或者和本例所示一样的WebContent目录。如图10-6所示

****

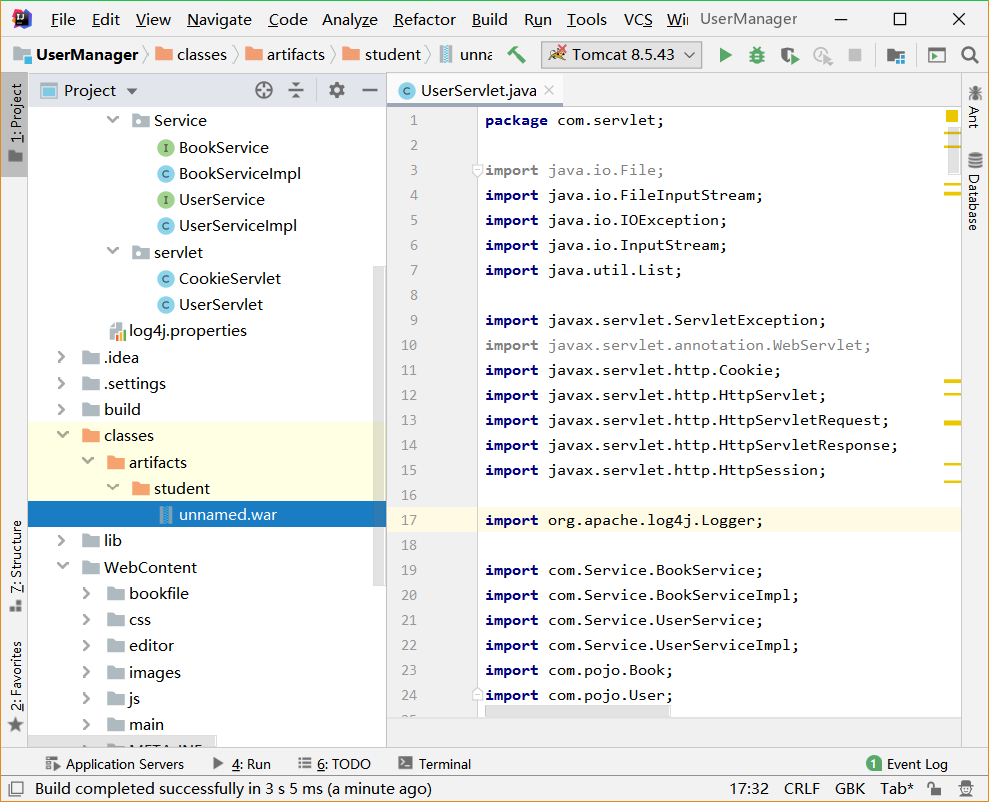
**图10-6添加目录**

点击Directory Content后添加我们的WebContent即可，如图10-7所示

****

**图10-7 添加目录**

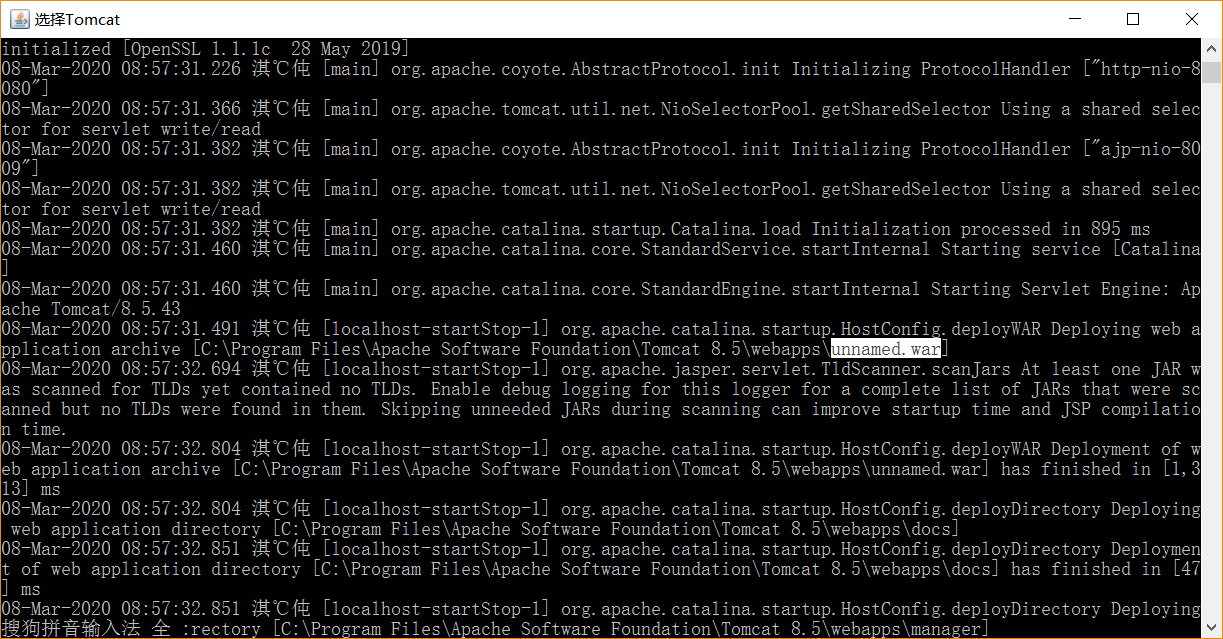
1. 完成Artifact 添加后我们就可以直接“Build”生成war文件了。依次点击Build-> Build Artifacts 即可完成war文件的打包。打包完成后的war文件如下图10-8

****

**图10-8 war文件**

1. 复制war包到根目录

终于完成了对项目的war文件的打包，接下来事情变得异常的简单和快捷了，我们只需将war文件放在Tomcat 的webapps 目录下，我们的Tomcat 在启动的时候回自动扫描该文件，并自动将该文件解压加载，我们的项目就顺利运行起来了。在使用命令行启动Tomcat后我们再控制台也可以看到Tomcat确实发现了我们的项目。如图10-9.

****

**图10-9 开启Tomcat**

Unamed就是我们刚才打包完成的项目了。

1. 完成项目部署

接下来我们又可以在浏览器中输入localhost:8080/unamed即可打开我们的项目，现在我们应该很清楚我们Artifact的名字的意义了吧，对就是我们部署后项目的名字。如图10-10所示。

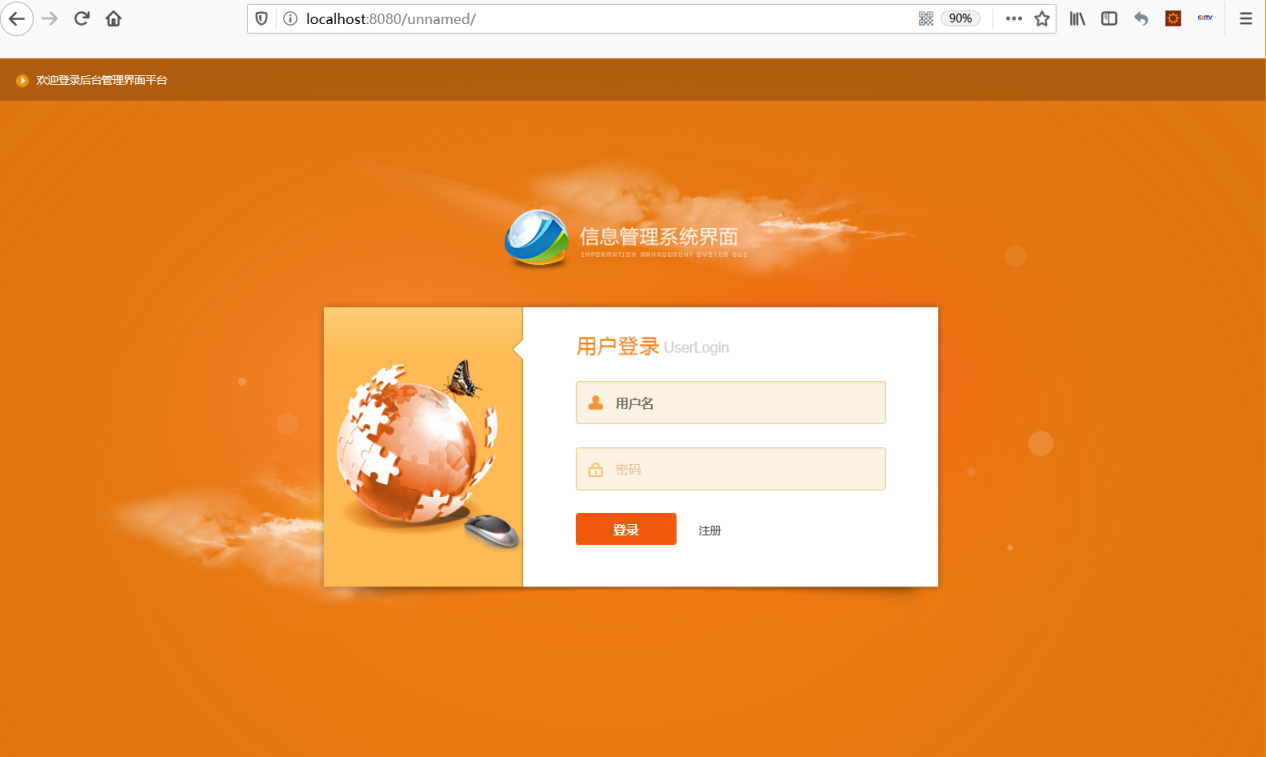
****

图10-10 开启项目

## Spring Boot 部署方式

### Spring Boot快速部署

前面已经介绍了三种部署方式，虽然在一定基础上已经进行了简化，然而得益于Spring Boot的内部简化，我们部署的方式将进一步得到简化。甚至我们可以直接输入一行命令即可完成项目的部署。

我们暂且先将其原理搁置一边，我们先来体验一下Spring Boot方式的简洁。在部署前我们应当有的准备工作是：

我们已经使用Spring Boot框架完成了我们的项目开发

部署的目标服务器上已经安装了java环境，并配置好了环境变量

准备工作就是这么多！

我们首先需要在我们的Idea中进行打包，这里我们使用的时maven进行打包，当然你也可以使用gradle打包工具进行打包。我们再pom.xml 中添加如下代码

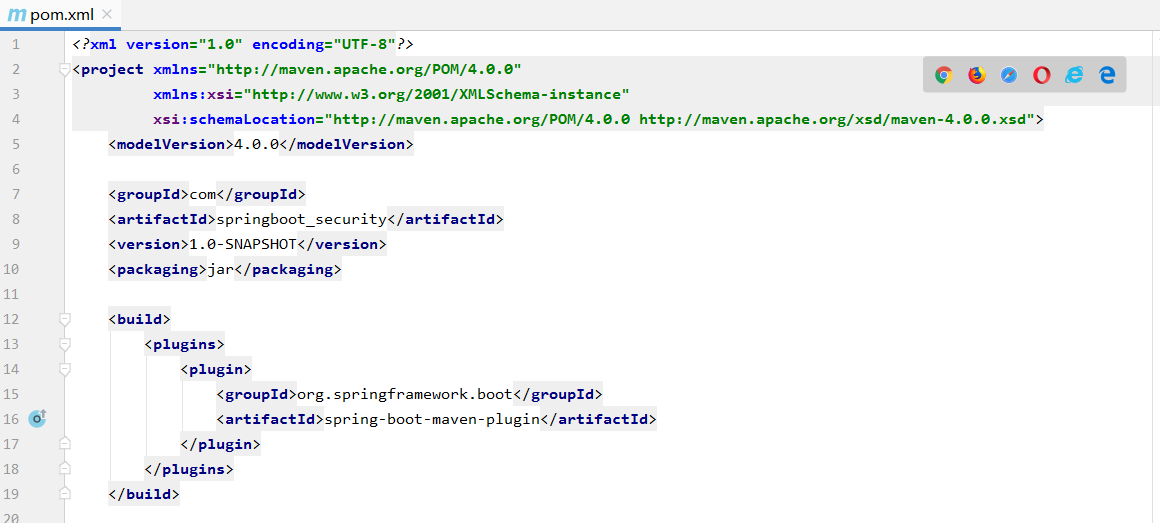


图 10-11 pom文件内容

其中<packaging>jar</packaging>代表我们的项目需要打包为jar文件，当然也可以是war文件。下面的<build></build>中是我们需要使用的maven插件。

打包完成我们直接打开cmd控制台窗口输入java -jar name.jar 回车，就会出现如下的界面：

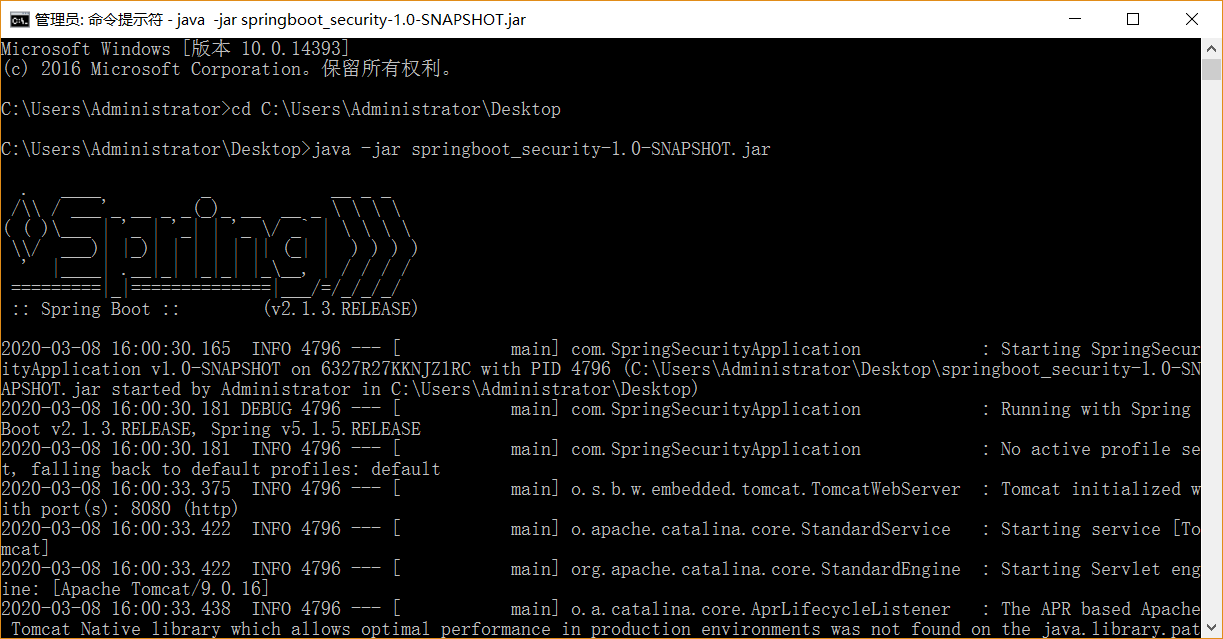


图10-12 开启项目

可以看到大大的Spring标志，我们的部署就完成了，多么的简洁明了啊。

### Spring Boot 部署原理

现实的效果是我们的部署方式变得极为简单，并且我们需要深入理解的是我们的项目中已经没有了传统项目的webapps 目录和web.xml文件了。要知道web.xml文件是配置我们的Servlet以及其他重要控制器的啊！而且我们也不需要服务器有Tomcat环境以及其他的配置文件了，为什么我们不需要这些了呢？（当然Spring Boot默认是不支持jsp文件的，这也是为了更好地进行前后端的分离）本节就来探知其中的原理，并且在看完本节后我们自己也可以自己实现这一过程。

我们先来看看原来的配置文件以及Tomcat都为我们做了什么，这样我们才可以知道我们应当做什么来替换这些操作，我们的配置文件主要分为web.xml，application.xml以及mvc.xml，我们回忆一下他们做的事情有哪些：

1. 配置DispathcherServlet，将其放入Tomcat中
2. 加载mvc.xml文件并且启动init（）方法以及拦截所有请求
3. 扫描controller，service，dao等组件

既然我们不需要外置的Tomcat，那么我们唯一的做法只能是将Tomcat嵌入我们的普通项目中来，实际上Spring Boot的做法便是如此。我们可以添加以下依赖：

<dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-core</artifactId>  
 <version>8.5.33</version>  
 </dependency>

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.2.4.RELEASE</version>  
 </dependency>

该依赖是Tomcat和Spring Mvc环境，我们将其加入我们的普通java项目中来，其中项目目录是这样的：如图10-13

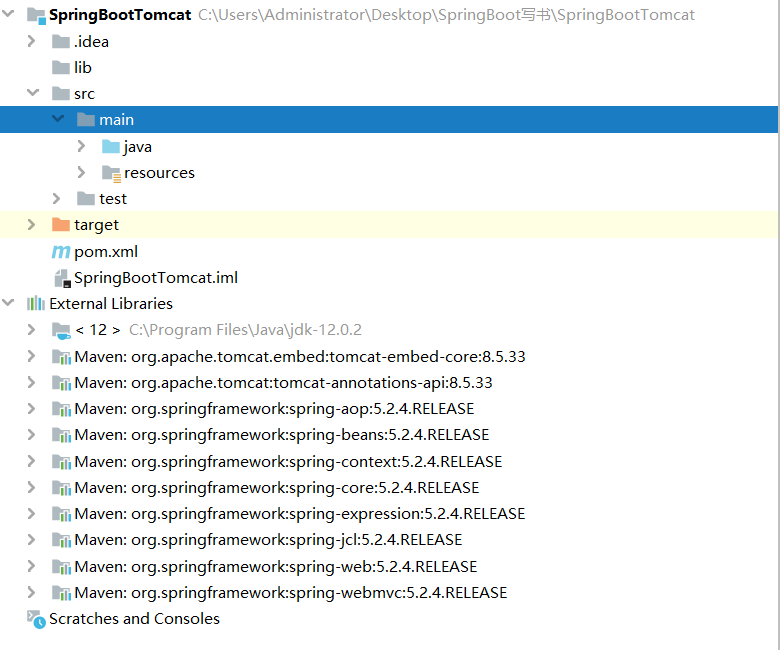
****

图10-13 项目目录

现在我们已经有了环境的配置，就让我们来看看SpringBoot的启动开始阶段是怎样的。通常是我们编写一个主类，然后在主类中调用SpringApplication的run方法，并在run方法中将我们的class文件传入即可。例如下面这样：

public class Test {  
 public static void main(String[] args) {  
 SpringApplication.run(Test.class);  
 }  
 }

我们都知道在一个java程序启动时都是先执行main方法，那么顺理成章Spring Boot的启动就应当从这个run方法中开始。我们首先可以确定的时在这个run方法中一定手动启动了内嵌的tomcat程序，并且接下来还配置好了Spring和Spring Mvc以及我们的DispatcherServlet，因为这些都是一个基本道德Spring Mvc程序都要有的。事实上的确如此。如果我们也能够仿照该过程编写一个run方法岂不是可以让一个普通java程序变成一个SpringBoot程序了？

我们添加SpringApplication类并新建run方法如下：

File base = new File(System.getProperty("java.io.tmpdir"));  
 //启动内嵌Tomcat  
 Tomcat tomcat = new Tomcat();  
 tomcat.setPort(8888);  
 //tomcat.addWebapp("/",base.getAbsolutePath());  
 tomcat.addContext("/",base.getAbsolutePath());  
 try {  
 tomcat.start();  
 System.out.println("----------init springmvc---------");  
 //初始化Spring环境  
 AnnotationConfigWebApplicationContext annotationConfigWebApplicationContext =  
 new AnnotationConfigWebApplicationContext();  
 annotationConfigWebApplicationContext.register(WebConfig.class);  
 annotationConfigWebApplicationContext.refresh();  
  
 //新建一个DispatcherServlet  
 DispatcherServlet dispatcherServlet = new DispatcherServlet(annotationConfigWebApplicationContext);  
 Wrapper wrapper = tomcat.addServlet("/","myservlet",dispatcherServlet);  
 wrapper.setLoadOnStartup(1);  
 wrapper.addMapping("/");  
 tomcat.getServer().await();  
 } catch (LifecycleException e) {  
 e.printStackTrace();  
 }

这段代码主要的功能只要三个：

1. 启动tomcat，配置好端口
2. 配置Spring环境
3. 新建DispatcherServlet 并加入tomcat

需要注意的是WebConfig.class这个类时为了替代我们的spring.xm配置文件而新建的，在该配置文件中主要就是扫描我们的控制器和其他组件。我们完全可以使用注解的形式进行替换。我们的WebConfig类很简单

import org.springframework.context.annotation.ComponentScan;  
 import org.springframework.context.annotation.Configuration;  
  
 @ComponentScan("mvc")  
 @Configuration  
 public class WebConfig {  
 }

@ComponentScan可以为我们扫描该目录下的其他控制器等组件。现在我们可以开始编写我们的控制器了，我们这里编写了一个简单的测试控制器TestController该控制器只是单纯地输出一句话。

@Controller  
 public class TestController {  
 @RequestMapping("/TestController")  
 public String Test(){  
 System.out.println("testcontroller is running !");  
 return "index";  
 }  
 }

完成以上的工作我们就可以开启我们的项目了，点击启动。令人惊喜的一幕出现了，果真在控制台部分启动了tomcat并且我们可以看到web.xml not found等字样，要知道只有tomcat将一个项目识别为web项目时才会去寻找web.xml。这说明我们的项目已经是一个web项目了。

我们可以试着访问一下该目录了：<http://localhost:8888/TestController，我们可以看我们控制器中的那句话被打印出来了。>如图10-14。

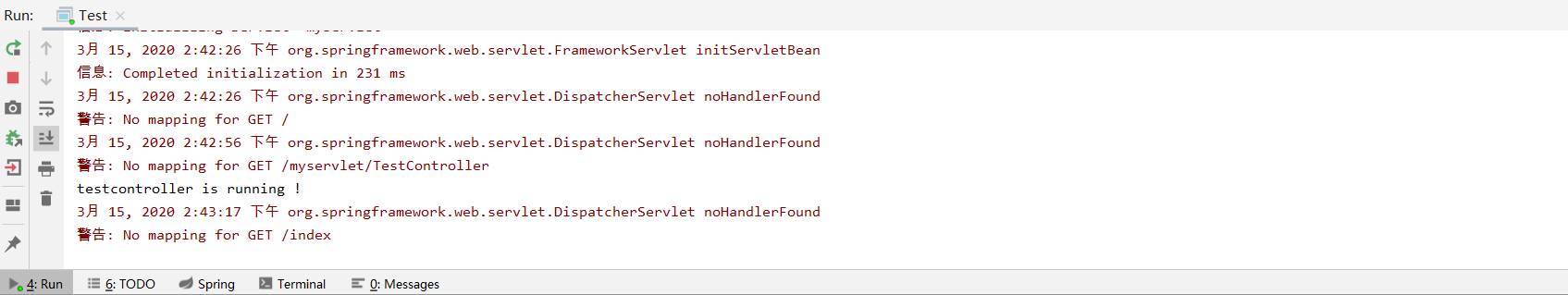


图 10-14 访问控制器

我也可以将该简单的测试项目打包成war文件或者jar文件，同样可以直接部署到我们的服务器上。

至此我们已经了解了Spring Boot是如何脱离tomcat 和Spring 配置文件的束缚并完成一个项目的启动部署的了。

## **Docker**

一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，不同版本环境的兼容，对运维人员都是考验

Docker之所以发展如此迅速，也是因为它对此给出了一个标准化的解决方案。

环境配置如此麻烦，换一台机器，就要重来一次，费力费时。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。开发人员利用 Docker 可以消除协作编码时“在我的机器上可正常工作”的问题。

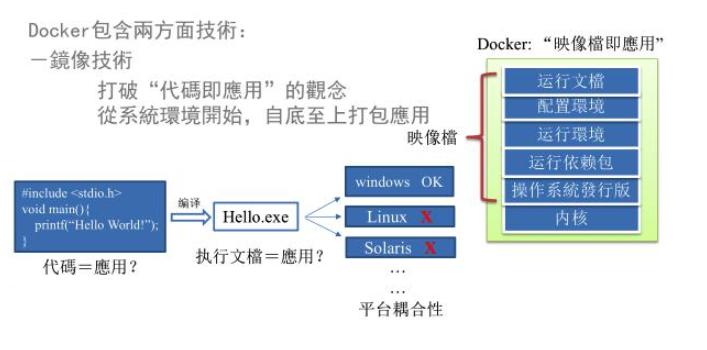


图10-16

之前在服务器配置一个应用的运行环境，要安装各种软件，例如一个普通的web项目需要安装Java/Tomcat/MySQL/JDBC驱动包等。安装和配置这些东西有多麻烦就不说了，它还不能跨平台。假如我们是在 Windows 上安装的这些环境，到了 Linux 又得重新装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。

传统上认为，软件编码开发/测试结束后，所产出的成果即是程序或是能够编译执行的二进制字节码等(java为例)。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让维运团队得以部署应用程式，开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，仍然常常发生部署失败的状况。Docker镜像的设计，使得Docker得以打破过去「程序即应用」的观念。透过镜像(images)将作业系统核心除外，运作应用程式所需要的系统环境，由下而上打包，达到应用程式跨平台间的无缝接轨运作。

### 什么是Docker？

Docker是一个Go语言实现的云开源项目。

Docker的主要目标是“Build，Ship and Run Any App,Anywhere”，也就是通过对应用组件的封装、分发、部署、运行等生命周期的管理，使用户的APP（可以是一个WEB应用或数据库应用等等）及其运行环境能够做到“一次封装，到处运行”。

Linux 容器技术的出现就解决了这样一个问题，而 Docker 就是在它的基础上发展过来的。将应用运行在 Docker 容器上面，而 Docker 容器在任何操作系统上都是一致的，这就实现了跨平台、跨服务器。只需要一次配置好环境，换到别的机子上就可以一键部署好，大大简化了操作，加快了我们开发者的开发进度。

Docker中最基本和必须了解的内容分别是镜像，容器和仓库。

镜像：我们打包好的一个环境，例如tomcat镜像，mysql镜像等。

容器：在我们的宿主机上容纳镜像的容器，根据具体镜像生产对应的容器为我们 服务。

仓库：存放我们的镜像的地方，有点类似GitHub的仓库。我们可以将配置好的镜 像传入仓库。

### 安装Docker

在讲述如何安装Docker之前我们需要首先安装一个linux系统，在网络以及各大论坛上都有详细的安装过程，在此不再赘述。我们使用的是centos6.8这个操作系统。

安装过程主要分为如下5个步骤：

1. 打开终端获取管理员权限，输入yum install -y epel -release
2. 继续输入yum install -y docker-io
3. 安装完成后我们可以查看docker的配置文件如图10-17

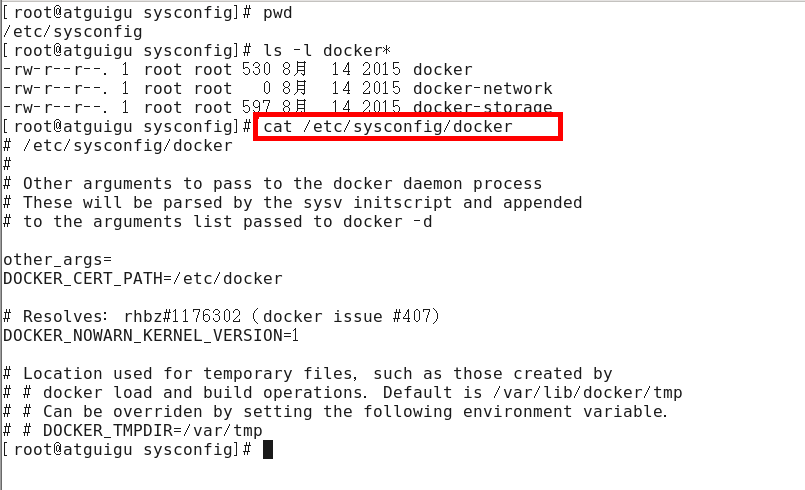


图10-17

1. 启动docker后台服务：在终端中输入 service docker start。
2. 我们可以输入docker version后会返回我们当前docker的版本号，这也说明我 们的docker安装成功了。
3. 由于docker仓库所在服务器在国外访问速度较慢，我们应当将地址替换为国内 阿里服务器。登录阿里云后选择镜像服务，下方有完整的替换方法。

### Docker运行原理

Docker的运行原理可以用如下图10-18描述。

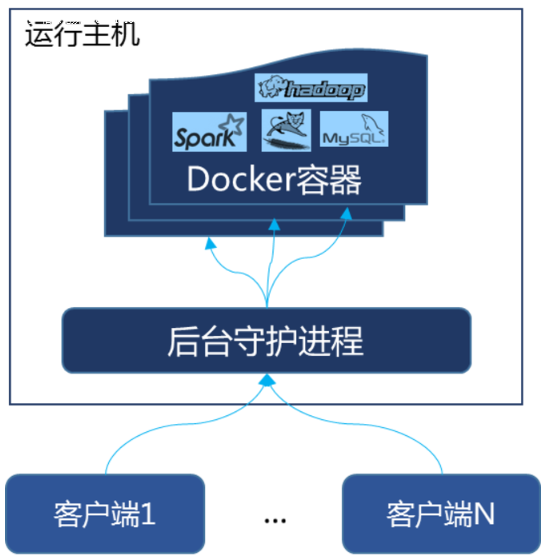


图10-18 Docker 结构

Docker是一个Client-Server结构的系统，Docker守护进程运行在主机上， 然后通过Socket连接从客户端访问，守护进程从客户端接受命令并管理运行在主机上的容器。 容器是从一个个的镜像中生成的，它们之间的关系就好像我们在编写java代码时，类的对象的关系。

看到这里你可能会想，这和我们的虚拟机是不是有点类似。事实上确实如此，但是虚拟机的启动缓慢，性能低下，十分的不方便。然而Docker却可以做到秒级别的启动时间，它是如何实现的呢？

首先，docker有着比虚拟机更少的抽象层。由亍docker不需要Hypervisor实现硬件资源虚拟化,运行在docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上docker将会在效率上有明显优势。

其次，docker利用的是宿主机的内核,而不需要Guest OS。因此,当新建一个容器时,docker不需要和虚拟机一样重新加载一个操作系统内核。仍而避免引寻、加载操作系统内核返个比较费时费资源的过程,当新建一个虚拟机时,虚拟机软件需要加载Guest OS,返个新建过程是分钟级别的。而docker由于直接利用宿主机的操作系统,则省略了返个过程,因此新建一个docker容器只需要几秒钟。

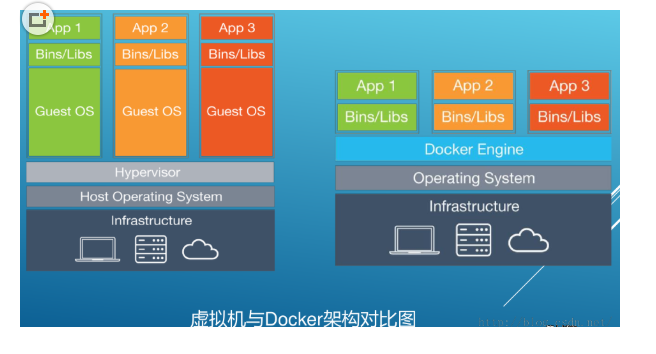
****

图10-19 虚拟机与Docker对比

镜像的分层加载不同于普通的程序的加载，Docker的镜像实际上是一层一层地进行加载并且docker的镜像实际上也是由一层一层的文件系统组成。

我们可以仔细观察我们在下载某个镜像的过程，例如我们使用pull命令下载MongoDB的过程如下图10-20.

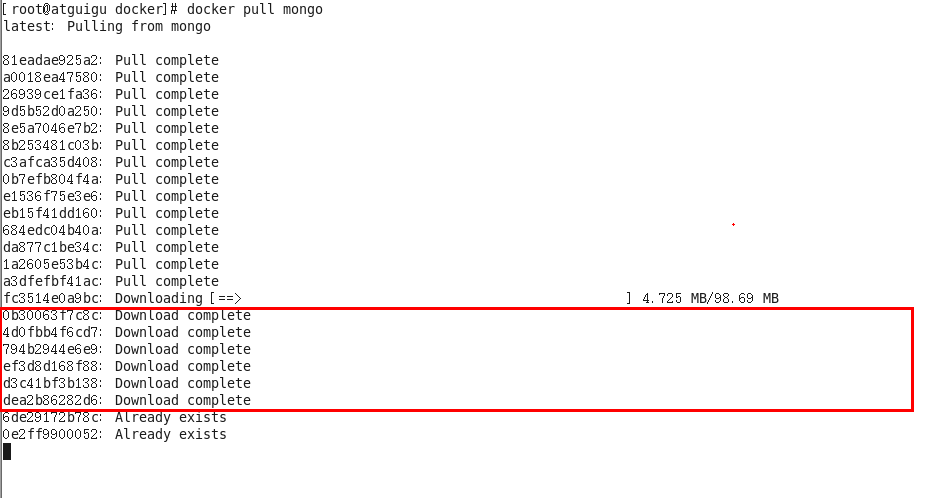
****

图10-20 下载MongoDB过程

我们可以看到下载过程中是经过一层一层地下载，我们需要先下载一个centos镜像，然后继续下载MongoDB，其间还有很多其他的层，我就不一一列出了。完成后，我们加载过程也是按照层级开始加载。

Docker这么做的最大的好处就是 - 共享资源

比如：有多个镜像都从相同的 base 镜像构建而来，那么宿主机只需在磁盘上保存一份base镜像，

同时内存中也只需加载一份 base 镜像，就可以为所有容器服务了。而且镜像的每一层都可以被共享。

### Docker常用命令介绍

在这一小节我们开始介绍Docker的一些常用命令，帮助我们更好地了解Docker。我们的命令主要分为帮助命令，镜像命令以及容器命令。

**Docker常用命令**

**帮助命令**

**容器命令**

**镜像命令**

我们首先来介绍帮助命令，帮助命令使用还是相对简单，主要包括以下3中命令。

docker version //用来查看当前docker的版本号

docker info //查看当前docker的相关信息

docker -help //我们可以使用该命令查看docker的所有命令语法等

对于镜像的操作是我们需要学习的重点，主要的命令如下

|  |  |
| --- | --- |
| 操作命令 | 功能 |
| Docker images +image name | 查看本机所有镜像 |
| Docker search +image name | 查找镜像 |
| Docker pull +image name | 下载镜像 |
| Docker rmi +image name | 移除镜像 |

除了基本操作指令外，很多指令都还有选项，例如在“docker images”后加上“-a”即可列出所有的镜像（包含中间镜像）。“-q”可以只显示镜像的ID信息，“-digests”可以只显示摘要信息。

同理其他的指令也都有各种各样的选项，期待读者可以自行依次验证并实验。

接下来就是最重要的容器相关命令了。

Docker run [OPTION] image name or ID [COMMAND]这条命令就是启动我们从仓库中下载的镜像，其中的option是十分重要的，我们一一来了解。

--name：为容器指定一个名称

-d：后台运行容器，并返回容器ID，也即启动守护式容器；

-i：以交互模式运行容器，通常与 -t 同时使用；

-p：: 指定端口映射，某些镜像例如tomcat需要知道端口我们才可以顺利访问；

Docker ps 命令是列出正在运行的所有容器和其对应的详细信息；

Docker start；docker restart；docker stop；docker kill；docker rm等命令我们都可以很轻松地从其命名知道它的作用，不再一一赘述。

前面已经提及我们的docker容器是层状结构，也就是说任何一个镜像实例出来的容器都含有一个最基本的centos 然后在其上一层层叠加其他环境。那么也就是说所有容器都可以有一个command窗口用于交互。例如当我们运行tomcat时，我们会得到一个容器的shell然后我们可以手动控制tomcat等，那么我们如果要返回宿主机shell呢？我们可以按住ctrl+p+q返回宿主机，但tomcat不会停止，此外我们可以输入exit完全退出我们的tomcat。

以上就涵盖了一些简单常用的命令，真正的命令大全还需要读者去docker官网查询并熟练使用。

### Docker部署tomcat

这一小节我们来实际操作部署一个镜像，以便我们熟悉整个安装过程。整个过程主要分为4个步骤。

1. 搜索镜像，输入docker search tomcat回车就可以在服务器上搜索tomcat镜像

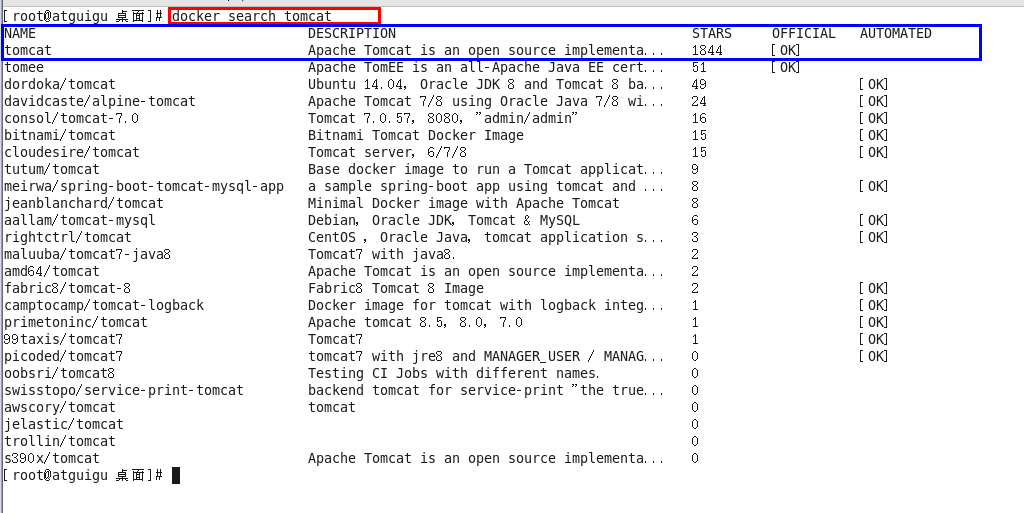


图10-20 搜索镜像

1. 拉取镜像，输入docker pull tomcat 默认拉取最新镜像

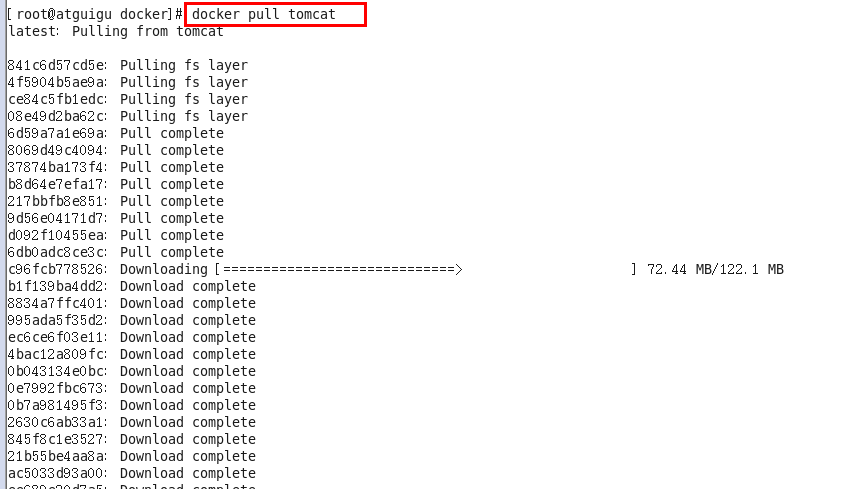


图10-21 拉取镜像

1. 查看是否拉取到镜像，输入docker images

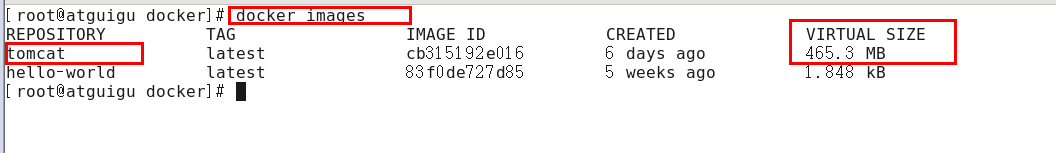


图10-22 确认镜像

1. 运行镜像 输入 docker tun -it -p 8080:8080 tomcat，代表容器8080端口映射我 们宿主机8080端口

在我们运行镜像后终端会显示和我们再idea中开发web应用一样的tomcat日志信息，这也说明我们的镜像部署成功了，至于我们的项目如何部署到容器中，我们可以通过tomcat容器的shell打开webapps 目录和前面讲述的部署方式没有什么区别。

### Docker部署mysql

上一小节我们已经完成了tomcat的部署，mysql也是我们最常用的环境之一。这一小节我们来看看它的部署方法。

1. 搜索镜像，和tomcat一样。输入docker search mysql回车就可以在服务器上搜 索mysql镜像
2. 继续输入docker pull mysql 我们可以选择合适的版本号例如5.6.等待下载完成 即可。
3. 输入：

Docker run -p 12345:3306 --name mysql

-v /xxx/mysql/conf:/etc/mysql/conf.d

-v/xxx/mysql/logs:/logs

-v/xxx/mysql/data:/var/lib/mysql

-e MYSQL\_ROOT\_PASSWORD=123456 -d

mysql:5.6

我们对该命令进行说明

-p 12345:3306：将主机的12345端口映射到docker容器的3306端口。

--name mysql：运行服务名字

-v /xxx/mysql/conf:/etc/mysql/conf.d ：将主机/xxx/mysql录下的conf/my.cnf 挂载到容器的 /etc/mysql/conf.d

-v /xxx/mysql/logs:/logs：将主机/xxx/mysql目录下的 logs 目录挂载到容器的 /logs。

-v /xxx/mysql/data:/var/lib/mysql ：将主机/xxx/mysql目录下的data目录挂载到容器的 /var/lib/mysql

-e MYSQL\_ROOT\_PASSWORD=123456：初始化 root 用户的密码

-d mysql:5.6 : 后台程序运行mysql5.6

1. 运行成功后我们就可以使用我们熟悉的mysql命令了，例如我们需要登录可以 使用mysql -uroot -p命令登录数据库，输入密码即可登录。然后各种建库，建 表操 作都会轻而易举。



图10-24 登录docker中的mysql

至此为止，我们的docker部分已经讲述完毕，当然这只是一个docker入门的使用教程而已，对于更专业的内容可以参考其他书籍等，例如自己编写docker file等用代码的形式配置我们想要的环境。

## 数据迁移

### 平滑地迁移数据库

你可能会认为：数据迁移无非是将数据从一个数据库拷贝到另一个数据库。不是很简单吗？主要的方法如下：

1. 可以通过 MySQL 主从同步的方式做到准实时的数据拷贝；
2. 也可以通过 mysqldump 工具将源库的数据导出，再导入到新库。

这有什么复杂的呢？如果不考虑现实的生产情况的话，确实像你说的那样。但是我们的实际场景是我们的系统在运行，同时需要将旧库的数据 ，我们需要做到以下事情

* 1）在线的迁移：也就是在迁移的同时还会有数据的写入；  
   2）数据完整性：也就是说在迁移之后需要保证新的库和旧的库的数据是一致的；
* 3）支持回滚：这样一旦迁移的过程中出现问题，可以立刻回滚到源库，不会对系 统的可用性造成影响。

为了达到以上目的，我们的数据迁移就显得不那么简单了。

### 我们的解决方案

## 单元测试

当我们完成了一个个功能的开发后，最后一步就是需要运行每一个功能点位的测试，确保项目上线后能够具有很好的鲁棒性。测试主要分为黑盒测试和白盒测试，下面对该两种测试方法进行简单的介绍。

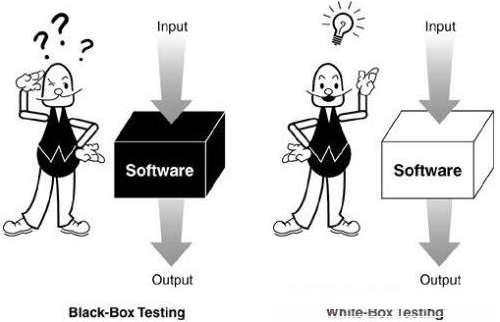


图10-25 黑盒测试和白盒测试

如图10-25可以直接看出来，黑盒测试就当整个程序是个黑盒子，我们看不到它里面做了些什么事情，只能通过输入输出看是否能得到我们所需的来测试。而白盒测试可以当盒子是透明的，里面的一切我们都看的清楚，从而我们可以通过去测内部结构来测试。

1. 黑盒测试 (Black-Box Testing)

　　黑盒测试又称为功能测试、数据驱动测试或基于规格说明书的测试，是一种从用户观点出发的测试。测试人员一般把被测程序当作一个黑盒子。

　　黑盒测试主要测到的错误类型有：不正确或遗漏的功能；接口、界面错误；性能错误；数据结构或外部数据访问错误；初始化或终止条件错误等等。

常用的黑盒测试方法有：等价类划分法；边界值分析法；因果图法；场景法；正交实验设计法；判定表驱动分析法；错误推测法；功能图分析法。

2. 白盒测试 (White-Box Testing)

　　为什么要考虑白盒测试呢？举个例子说明，设x=0,y=5, 程序设计z=y/x, 如果只考虑输入输出的黑盒测试，我们看不出程序内部的逻辑错误(分母不能为零)，即使我们使用黑盒测试的测试用例覆盖的再全也发现不了这个问题，这个时候就要考虑使用白盒测试了。

　　而什么是白盒测试呢？白盒测试也称为结构测试或逻辑驱动测试，是针对被测单元内部是如何进行工作的测试。它根据程序的控制结构设计测试用例，主要用于软件或程序验证。白盒测试法检查程序内部逻辑结构，对所有的逻辑路径进行测试，是一种穷举路径的测试方法，但即使每条路径都测试过了，但仍然有可能存在错误。因为：穷举路径测试无法检查出程序本身是否违反了设计规范，即程序是否是一个错误的程序；穷举路径测试不可能检查出程序因为遗漏路径而出错；穷举路径测试发现不了一些与数据相关的错误。

白盒测试需要遵循的原则有：

1）保证一个模块中的所有独立路径至少被测试一次；

2）所有逻辑值均需要测试真（true）和假（false）两种情况；

3）检查程序的内部数据结构，保证其结构的有效性；

4） 在上下边界及可操作范围内运行所有循环。

幸运的是现在已经有了很多专门为测试而设计的框架，其中最流行的就是Junit框架，此外我们还会介绍Spring Boot自带的测试模块。

### 无框架下的测试

我们试想当我们没有使用任何框架时我们该如何测试一个方法的正确性呢？我们理所应当地会想我们会编写一个main方法然后调用该方法，得出测试结果并比对即可。为此我们可以编写一个简单的实例，用于测试我们的计算器类的加方法是否正确。

package Test;  
 public class TestCacu {  
 public static void main(String[] args) {  
 Cacu cacu = new Cacu();  
 System.out.println(cacu.Add(1,2));  
 }  
 }  
 class Cacu{  
 public int Add(int a,int b){  
 return a+b;  
 }  
 }

这样编写其实没有本质的错误，但是这样的步骤太过于繁琐，而且不利于我们大规模的测试。每一次的测试都要修改配置，编写测试类等等缺点。一个较好的测试应当不需要我们写过多的代码，不需要改动配置直接运行就可以得出测试结果，并且观察输出语句的判断方式在参数较多时显得效率低下。Junit已经为我们解决了这些问题，下面我们来介绍JUnit的使用。

### JUnit

什么是JUnit？

JUnit是一个Java语言的单元测试框架。它由Kent Beck和Erich Gamma建立，逐渐成为源于Kent Beck的sUnit的xUnit家族中最为成功的一个。 JUnit有它自己的JUnit扩展生态圈。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。 [1]

JUnit是由 Erich Gamma 和 Kent Beck 编写的一个[回归测试](https://baike.baidu.com/item/%E5%9B%9E%E5%BD%92%E6%B5%8B%E8%AF%95/1925732" \t "https://baike.baidu.com/item/junit/_blank)框架（regression testing framework）。Junit测试是程序员测试，即所谓[白盒测试](https://baike.baidu.com/item/%E7%99%BD%E7%9B%92%E6%B5%8B%E8%AF%95/934440" \t "https://baike.baidu.com/item/junit/_blank)，因为程序员知道被测试的软件如何（How）完成功能和完成什么样（What）的功能。Junit是一套框架，继承TestCase类，就可以用Junit进行自动测试了。

Junit优点包括安装使用方便；可以同时执行多个测试方法；测试结果直观明了等。其使用步骤也十分简单。

我们先来介绍JUnit的各种注解然后再进行实际的测试。

1. @Before 注解：与junit3.x中的setUp()方法功能一样，在每个测试方法之前执行；
2. @After 注解：与junit3.x中的tearDown()方法功能一样，在每个测试方法之后执行；
3. @BeforeClass 注解：在所有方法执行之前执行；
4. @AfterClass 注解：在所有方法执行之后执行；
5. @Test(timeout = xxx) 注解：设置当前测试方法在一定时间内运行完，否则返回错 误；
6. @Test(expected = Exception.class) 注解：设置被测试的方法是否有异常抛出。抛出 异常类型为：Exception.class；

7）@Ignore 注解：注释掉一个测试方法或一个类，被注释的方法或类，不会被执行。

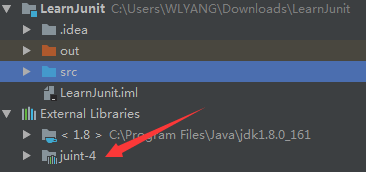


图10-26 导入JUnit依赖

1. 打开IDEA，新建java项目，在项目中导入junit包 File->Project Structure->libraries->点击‘+’ ->java->找到下载的junit-4.10->apply->ok 可以看到junit已经导入项目。
2. 在src中编写源代码（被测试的类及其方法），编写测试类和测试方法，项目目录结构如下：

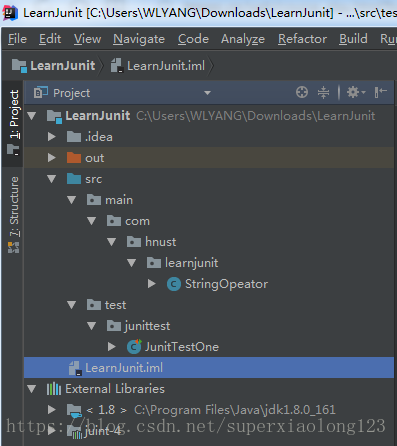


图10-27 项目目录结构

被测试代码：

package main.com.hnust.learnjunit;

public class StringOpeator {

public String sayHello(String str1,String str2){

return str1+str2;

}

}

测试类代码及其要求：

package test.junittest;

import main.com.hnust.learnjunit.StringOpeator;

import org.junit.Assert;

import org.junit.Test;

public class JunitTestOne {

@Test

public void test1(){

StringOpeator so =new StringOpeator();

String str=so.sayHello("Hello","Junit");

//断言语法

Assert.assertEquals("HelloJunit",str);

}

}

1. 在测试类中右键，选择“Run\*\*”,可以看到测试的结果报告，如果为红色则测试失败，反之绿色测试成功。
2. 以上使用到Assert断言机制，所谓断言，就是判断是或者不是，它的作用和优点是什么呢？就是比较预期结果和实际结果，以图形化的方式显示结果，就像上图中，出现原谅绿色勾就表示预期结果和实际结果一样，而出现红色叹号就说明测试用例没通过，就需要改善源代码，避免程序潜在漏洞了。

这就是Junit基本使用知识，以后在使用过程中再挖掘其更多的功能，熟练使用测试工具对于程序员写出高质量代码，减少程序bug有很大的帮助。

### Spring Boot中的测试

前面我们已经了解了测试的基本内容和大概流程，在本节我们就直接开始使用Spring Boot的test模块进行演示实战。

Spring Boot中引入单元测试很简单，依赖如下：

<dependency>

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

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

<scope>test</scope>

</dependency>

然后再编写创建好的测试类，具体代码如下：

@RunWith(SpringRunner.class)

@SpringBootTest

public class LearnServiceTest {

@Autowired

private LearnService learnService;

@Test

public void getLearn(){

LearnResource learnResource=learnService.selectByKey(1001L);

Assert.assertThat(learnResource.getAuthor(),is("Spring Boot Test"));

}

}

[上面就是最简单的单元测试写法，顶部只要@RunWith(SpringRunner.class)和SpringBootTest即可，想要执行的时候，鼠标放在对应的方法，右键选择run该方法即可。](mailto:上面就是最简单的单元测试写法，顶部只要@RunWith(SpringRunner.class)和SpringBootTest即可，想要执行的时候，鼠标放在对应的方法，右键选择run该方法即可。)

同理我们也可以在controller层进行单元测试，这时候就得用到MockMvc了，你可以不必启动工程就能测试这些接口。

MockMvc实现了对Http请求的模拟，能够直接使用网络的形式，转换到Controller的调用，这样可以使得测试速度快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且很方便。我们这里只是编写了伪代码演示，实际的例子可以参考项目实战部分。

@Controller

@RequestMapping("/learn")

public class LearnController{

@Autowired

private LearnService learnService;

/\*\*

\* 查询教程列表

\* @param page

\* @return

\*/

@RequestMapping(value = "/queryLeanList",method = RequestMethod.POST)

@ResponseBody

public AjaxObject queryLearnList(Page<LeanQueryLeanListReq> page){

List<LearnResource> learnList=learnService.queryLearnResouceList(page);

PageInfo<LearnResource> pageInfo =new PageInfo<LearnResource>(learnList);

return AjaxObject.ok().put("page", pageInfo);

}

这个类处于控制器层调用接口是/queryLeanList，我们可以为其创建一个测试类：

@Before

public void setupMockMvc(){

mvc = MockMvcBuilders.webAppContextSetup(wac).build(); //初始化MockMvc对象

session = new MockHttpSession();

User user =new User("root","root");

session.setAttribute("user",user); //拦截器那边会判断用户是否登录，所以这里注入一个用户

}

@Test

public void qryLearn() throws Exception {

mvc.perform(MockMvcRequestBuilders.get("/learn/resource/1001")

.contentType(MediaType.APPLICATION\_JSON\_UTF8)

.accept(MediaType.APPLICATION\_JSON\_UTF8)

.session(session)

)

.andExpect(MockMvcResultMatchers.status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.author").value("Spring Boot test"))

.andExpect(MockMvcResultMatchers.jsonPath("$.title").value("Spring Boot"))

.andDo(MockMvcResultHandlers.print());

}

需要注意的是这里我们使用了MockMvc，它可以自动地实现Http的请求，不然我们再测试时项目是未运行的，没有session没有用户的登录我们的测试代码时无法执行的。

新断言assertThat使用

JUnit 4.4 结合 Hamcrest 提供了一个全新的断言语法——assertThat。程序员可以只使用 assertThat 一个断言语句，结合 Hamcrest 提供的匹配符，就可以表达全部的测试思想，我们引入的版本是Junit4.12所以支持assertThat。

assertThat 的基本语法如下：

assertThat( [value], [matcher statement] );

Value：接下来想要测试的变量值；

Matcher： statement 是使用 Hamcrest 匹配符来表达的对前面变量所期望的值的声明，如果 value 值与 matcher statement 所表达的期望值相符，则测试成功，否则测试失败。

以前 JUnit 提供了很多的 assertion 语句，如：assertEquals，assertNotSame，assertFalse，assertTrue，assertNotNull，assertNull 等，现在有了 JUnit 4.4，一条 assertThat 即可以替代所有的 assertion 语句，这样可以在所有的单元测试中只使用一个断言方法，使得编写测试用例变得简单，代码风格变得统一，测试代码也更容易维护。

到此我们已经简单地介绍了一个java的项目的测试过程和使用的框架等知识，下一章我们将开始进行项目的实战。

# 项目实战