**尚硅谷大数据技术SpringBoot**

# 简单了解框架

框架，即framework。其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

框架是对特定应用领域中的应用系统的部分设计和实现的整体结构。

因为软件系统发展到今天已经很复杂了，特别是服务器端软件，涉及到的知识，内容，问题太多。在某些方面使用别人成熟的框架，就相当于让别人帮你完成一些基础工作，你只需要集中精力完成系统的业务[逻辑设计](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E8%AE%BE%E8%AE%A1)。而且框架一般是成熟，稳健的，他可以处理系统很多细节问题，比如，事务处理，安全性，数据流控制等问题。还有框架一般都经过很多人使用，所以结构很好，所以扩展性也很好，而且它是不断升级的，你可以直接享受别人升级代码带来的好处。

# 第1章 Spring

## 1.1 Spring 概述

1. Spring是一个开源框架
2. Spring为简化企业级开发而生，使用Spring，JavaBean就可以实现很多以前要靠EJB才能实现的功能。同样的功能，在EJB中要通过繁琐的配置和复杂的代码才能够实现，而在Spring中却非常的优雅和简洁。
3. Spring是一个**IOC**(DI)和**AOP**容器框架。
4. Spring的优良特性

① **非侵入式**：基于Spring开发的应用中的对象可以不依赖于Spring的API

② **依赖注入**：DI——Dependency Injection，反转控制(IOC)最经典的实现。

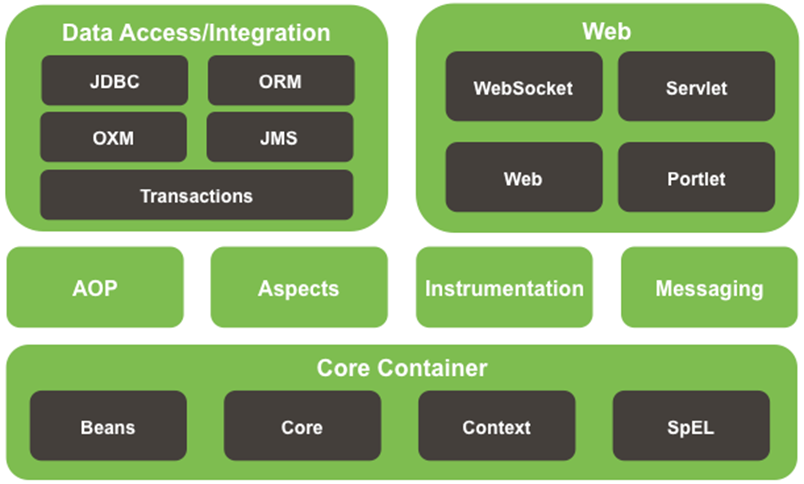
③ **面向切面编程**：Aspect Oriented Programming——AOP

④ **容器**：Spring是一个容器，因为它包含并且管理应用对象的生命周期

⑤ **组件化**：Spring实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用XML和Java注解组合这些对象。

⑥ **一站式**：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的Spring JDBC）。

1. Spring模块



## 1.2 Spring HelloWorld

1. 创建一个Maven版的Java工程
2. 在pom.xml中加入对Spring的依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>4.0.0.RELEASE</version>  </dependency> |

1. 创建Spring的核心配置文件

* 为文件取名字 例如：applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  </beans> |

1. 编写组件

* 创建控制层组件

|  |
| --- |
| package com.atguigu.spring.controller;  /\*\*  \* 控制层组件 处理客户端的请求，给客户端响应  \*/  import com.atguigu.spring.service.UserService;  public class UserController {  public void listAllUsers() {  }  } |

* 创建业务层组件接口

|  |
| --- |
| package com.atguigu.spring.service;  /\*\*  \* 业务层组件 处理业务逻辑  \*/  public interface UserService {  public void doGetAllUser();  } |

* 创建业务层组件实现类

|  |
| --- |
| package com.atguigu.spring.service;  import com.atguigu.spring.dao.UserDao;  public class UserServiceImpl implements UserService{    @Override  public void doGetAllUser() {  }  } |

* 创建持久层组件接口

|  |
| --- |
| package com.atguigu.spring.dao;  /\*\*  \* 持久层组件 负责数据库的CRUD操作  \*  \*/  public interface UserDao {    public void selectAllUsers();  } |

* 创建持久层组件实现类

|  |
| --- |
| package com.atguigu.spring.dao;  public class UserDaoJdbcImpl implements UserDao {    @Override  public void selectAllUsers() {  System.out.println("UserDaoJdbcImpl selectAllUsers Success .....");  }  } |

* 在spring的核心配置文件中管理Bean

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">    <!-- 管理组件 -->  <!--  bean: 对应一个被Spring管理的组件对象  id: bean的唯一标识  class: 组件对象对应的类的全类名  -->  <bean id="userController" class="com.atguigu.spring.controller.UserController">  </bean>  <bean id="userServiceImpl" class="com.atguigu.spring.service.UserServiceImpl">  </bean>  <bean id="userDaoJdbcImpl" class="com.atguigu.spring.dao.UserDaoJdbcImpl"></bean>  </beans> |

* 编写测试类

|  |
| --- |
| package com.atguigu.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.atguigu.spring.controller.UserController;  import com.atguigu.spring.dao.UserDao;  import com.atguigu.spring.service.UserService;  public class TestSpring {    @Test  public void testSpringXML() {  //1. 先创建Spring的容器对象  ApplicationContext ctx =  new ClassPathXmlApplicationContext("applicationContext.xml");    //2. 从Spring的容器中获取Bean对象  UserController uc = ctx.getBean("userController", UserController.class);  System.out.println("uc:" + uc);  UserService us = ctx.getBean("userServiceImpl",UserService.class);  System.out.println("us:" + us );  UserDao ud = ctx.getBean("userDaoJdbcImpl",UserDao.class);  System.out.println("ud:" + ud );  }  } |

1. 组件装配

* 在控制层组件中定义业务层组件类型的属性

|  |
| --- |
| package com.atguigu.spring.controller;  /\*\*  \* 控制层组件 处理客户端的请求，给客户端响应  \*/  import com.atguigu.spring.service.UserService;  public class UserController {  private UserService userService ;  public void setUserService(UserService userService) {  this.userService = userService;  }    public void listAllUsers() {  userService.doGetAllUser();  }  } |

* 在业务层组件中定义持久层组件类型的属性

|  |
| --- |
| package com.atguigu.spring.service;  import com.atguigu.spring.dao.UserDao;  public class UserServiceImpl implements UserService{    private UserDao userDao ;  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }    @Override  public void doGetAllUser() {  userDao.selectAllUsers();  }  } |

* 在Spring的核心配置文件中完成组件装配

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">    <!-- 管理组件 -->  <!--  bean: 对应一个被Spring管理的组件对象  id: bean的唯一标识  class: 组件对象对应的类的全类名  -->  <bean id="userController" class="com.atguigu.spring.controller.UserController">  <!-- 给属性注入值 -->  <property name="userService" ref="userServiceImpl"></property>  </bean>    <bean id="userServiceImpl" class="com.atguigu.spring.service.UserServiceImpl">  <property name="userDao" ref="userDaoJdbcImpl"></property>  </bean>  <bean id="userDaoJdbcImpl" class="com.atguigu.spring.dao.UserDaoJdbcImpl"></bean>  </beans> |

* 测试控制层 业务层 持久层的调用

|  |
| --- |
| package com.atguigu.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.atguigu.spring.controller.UserController;  import com.atguigu.spring.dao.UserDao;  import com.atguigu.spring.service.UserService;  public class TestSpring {    @Test  public void testSpringXML() {  //1. 先创建Spring的容器对象  ApplicationContext ctx =  new ClassPathXmlApplicationContext("applicationContext.xml");    //2. 从Spring的容器中获取Bean对象  UserController uc = ctx.getBean("userController", UserController.class);  System.out.println("uc:" + uc);  UserService us = ctx.getBean("userServiceImpl",UserService.class);  System.out.println("us:" + us );  UserDao ud = ctx.getBean("userDaoJdbcImpl",UserDao.class);  System.out.println("ud:" + ud );    uc.listAllUsers();  }  } |

## 1.3 基于注解开发Spring应用

### 1.3.1 常用注解标识组件

1. 普通组件：

@Component 标识一个受Spring IOC容器管理的组件

1. 持久化层组件：

@Repository 标识一个受Spring IOC容器管理的持久化层组件

1. 业务逻辑层组件：
2. @Service 标识一个受Spring IOC容器管理的业务逻辑层组件
3. 表述层控制器组件：

@Controller 标识一个受Spring IOC容器管理的表述层控制器组件

### 1.3.2 组件命名规则

1. 默认情况：使用组件的简单类名首字母小写后得到的字符串作为bean的id
2. 使用组件注解的value属性指定bean的id
3. 注意：事实上Spring并没有能力识别一个组件到底是不是它所标记的类型，即使将 @Respository注解用在一个表述层控制器组件上面也不会产生任何错误，所以 @Respository、@Service、@Controller这几个注解仅仅是为了让开发人员自己 明

确当前的组件扮演的角色。

### 1.3.3 Spring HelloWorld 注解版

1. 在控制层 业务层 持久层组件标注对应的注解

* 在控制层组件标注注解

|  |
| --- |
| @Controller  public class UserController {  } |

* 在业务层组件标注注解

|  |
| --- |
| @Service  public class UserServiceImpl implements UserService{  } |

* 在持久层组件标注注解

|  |
| --- |
| @Repository  public class UserDaoJdbcImpl implements UserDao {  } |

1. 在Spring的核心配置文件中开启组件扫描

* 在Spring的核心配置文件中开启组件扫描

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 组件扫描  base-package: 基本包  Spring会扫描通过base-package指定的包下以及子包下的组件，将带有Spring相关  注解的类管理到IOC容器中。  -->  <context:component-scan base-package="com.atguigu.spring"></context:component-scan>  </beans> |

1. 编写测试类

|  |
| --- |
| package com.atguigu.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.atguigu.spring.controller.UserController;  import com.atguigu.spring.dao.UserDao;  import com.atguigu.spring.service.UserService;  public class TestSpring {    @Test  public void testSpringXML() {  //1. 先创建Spring的容器对象  ApplicationContext ctx =  new ClassPathXmlApplicationContext("applicationContext.xml");  //2. 从Spring的容器中获取Bean对象  UserController uc = ctx.getBean("userController", UserController.class);  System.out.println("uc:" + uc);  UserService us = ctx.getBean("userServiceImpl",UserService.class);  System.out.println("us:" + us );  UserDao ud = ctx.getBean("userDaoJdbcImpl",UserDao.class);  System.out.println("ud:" + ud );  }  } |

### 1.3.4 @Autowired注解

1. @Autowired的工作机制

* 首先会通过当前被装配的属性的类型到IOC容器中去匹配对应的Bean对象,如果能唯一确定一个bean对象，则装配成功
* 当通过当前被装配的属性的类型匹配在IOC容器中匹配到多个对应的Bean对象时， 会再使用当前被装配的属性的名字与匹配到的Bean对象的id值再进行唯一确定，如果能确定唯一一个，则装配 成功，否则，抛出异常

expected single matching bean but found 2: userDaoJdbcImpl,userDaoMyBatisImpl

* 如果被装配的属性在IOC容器中匹配不到任何一个Bean对象，也会抛出异常

expected at least 1 bean which qualifies as autowire candidate for this dependency.

Dependency annotations:

@org.springframework.beans.factory.annotation.Autowired(required=true)}

* 如果匹配到多个Bean的情况，并且通过属性名也无法唯一确定一个Bean的时候，可以手动通过@Qualifier注解来具体指定装配哪个Bean对象.

* @Autowired 注解中required的属性默认是true，表示属性必须被装配，可以改为false，表示可选.也就是 有就装配，没有就不装配.

* @Autowired 和 @Qualifier 注解可以加在属性上，也可以加在方法上。

### 1.3.5 基于注解装配

1. 在 控制层 和 业务层分别定义需要被装配的组件类型的属性,并在属性上标注注解

* 在控制层中定义业务层类型的属性 和 相关方法

|  |
| --- |
| @Controller  public class UserController {  @Autowired  private UserService userService ;    public void listAllUsers() {  userService.doGetAllUser();  }  } |

* 在业务层中定义持久层类型的属性 和相关方法

|  |
| --- |
| package com.atguigu.spring.service;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.stereotype.Service;  import com.atguigu.spring.dao.UserDao;  @Service  public class UserServiceImpl implements UserService{  @Autowired  private UserDao userDao ;  @Override  public void doGetAllUser() {  userDao.selectAllUsers();  }  } |

* 在持久层中定义相关方法

|  |
| --- |
| @Repository  public class UserDaoJdbcImpl implements UserDao {    @Override  public void selectAllUsers() {  System.out.println("UserDaoJdbcImpl selectAllUsers Success .....");  }  } |

1. 在测试方法中测试 控制层 业务层 持久层的调用

|  |
| --- |
| package com.atguigu.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.atguigu.spring.controller.UserController;  import com.atguigu.spring.dao.UserDao;  import com.atguigu.spring.service.UserService;  public class TestSpring {  @Test  public void testSpringXML() {  //1. 先创建Spring的容器对象  ApplicationContext ctx =  new ClassPathXmlApplicationContext("applicationContext.xml");    //2. 从Spring的容器中获取Bean对象  UserController uc = ctx.getBean("userController", UserController.class);  System.out.println("uc:" + uc);  UserService us = ctx.getBean("userServiceImpl",UserService.class);  System.out.println("us:" + us );  UserDao ud = ctx.getBean("userDaoJdbcImpl",UserDao.class);  System.out.println("ud:" + ud );  uc.listAllUsers();    }  } |

# 第2章 SpringBoot

## 2.1 Spring分布式架构



## 2.2 SpringBoot 概述

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。

该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

## 2.3 为什么要使用SpringBoot

说到为什么使用Spring Boot, 就不得不提到Spring框架的前世今生

Spring框架由于其繁琐的配置，一度被人认为“配置地狱”，各种XML、Annotation配置混合使用，让人眼花缭乱，而且如果出错了也很难找出原因。

通过SpringMVC框架部署和发布web程序，需要和系统外服务器进行关联，操作繁琐不方便。

Spring Boot是由Spring官方推出的一个新框架，对Spring进行了高度封装，是Spring未来的发展方向。使用Spring Boot框架后，可以帮助开发者快速搭建Spring框架，也可以帮助开发者快速启动一个Web服务，无须依赖外部Servlet容器，使编码变得简单，使配置变得简单，使部署变得简单，使监控变得简单。

## 2.4 Spring 前世今生

1. Spring1.x 时代

在Spring1.x时代，都是通过xml文件配置bean

随着项目的不断扩大，需要将xml配置分放到不同的配置文件中

需要频繁的在java类和xml配置文件中切换。

1. Spring2.x时代

随着JDK 1.5带来的注解支持，Spring2.x可以使用注解对Bean进行申明和注入，大大的 减少了xml配置文件，同时也大大简化了项目的开发。

那么，问题来了，究竟是应该使用xml还是注解呢？

|  |
| --- |
| 最佳实践：  应用的基本配置用xml，比如：数据源、资源文件等；  业务开发用注解，比如：Service中注入bean等； |

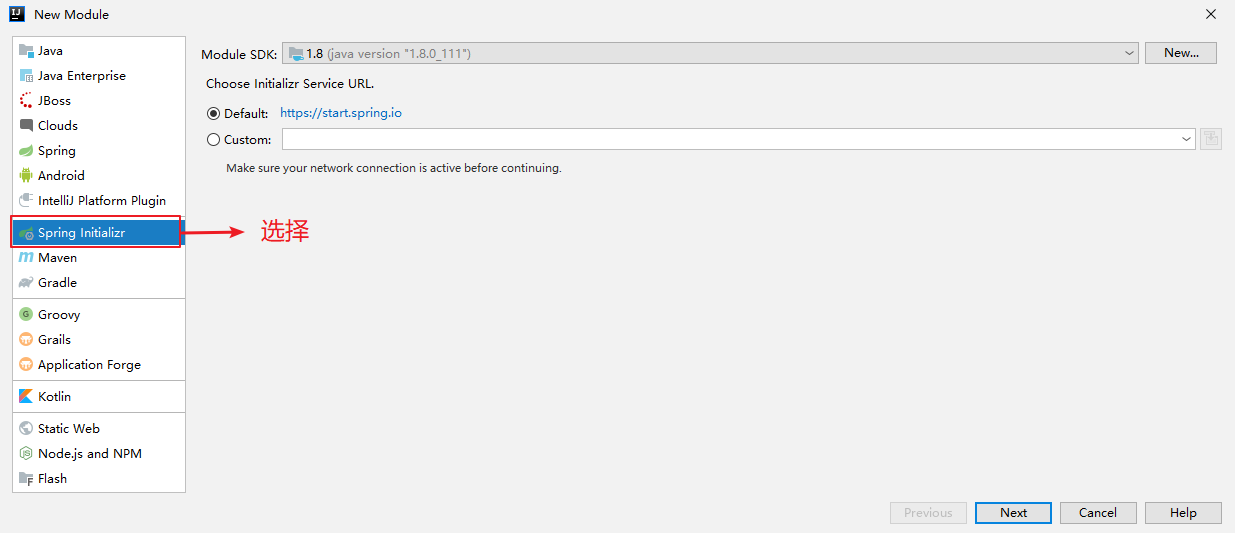
1. Spring3.x到Spring4.x

从Spring3.x开始提供了Java配置方式，使用Java配置方式可以更好的理解你配置的 Bean，现在我们就处于这个时代，并且Spring4.x和Spring boot都推荐使用java配置的 式。

|  |
| --- |
| Spring 1.X 使用基本的框架类及配置文件（.xml）实现对象的声明及对象关系的整合。 org.springframework.core.io.ClassPathResource org.springframework.beans.factory.xml.XmlBeanFactory org.springframework.context.support.ClassPathXmlApplicationContext  Spring 2.X 使用注解代替配置文件中对象的声明。简化配置。 org.springframework.stereotype.@Component org.springframework.stereotype.@Controller org.springframework.stereotype.@Service org.springframework.stereotype.@Repository org.springframework.stereotype.@Scope org.springframework.beans.factory.annotation.@Autowired  Spring 3.X 使用更强大的注解完全代替配置文件。 org.springframework.context.annotation.AnnotationConfigApplicationContext org.springframework.context.annotation.@Configuration org.springframework.context.annotation.@Bean org.springframework.context.annotation.@Value org.springframework.context.annotation.@Import  Spring 4.X 使用条件注解强化之前版本的注解。 org.springframework.context.annotation.@Conditional |

## 2.5 自动创建一个SpringBoot项目

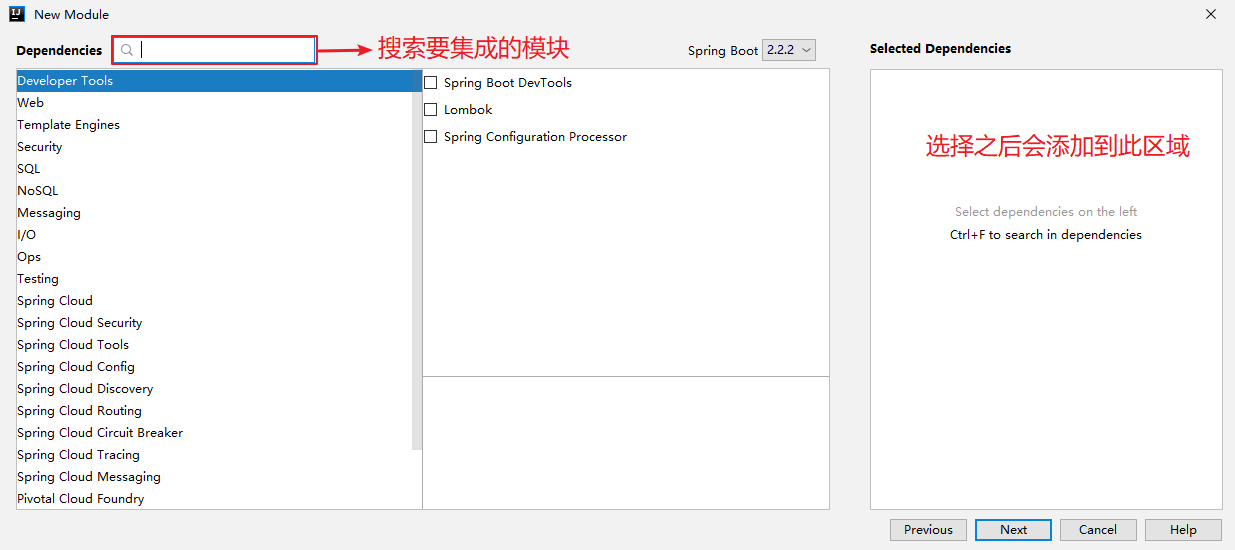
1. 在Idea中new→Module→Spring Initializr



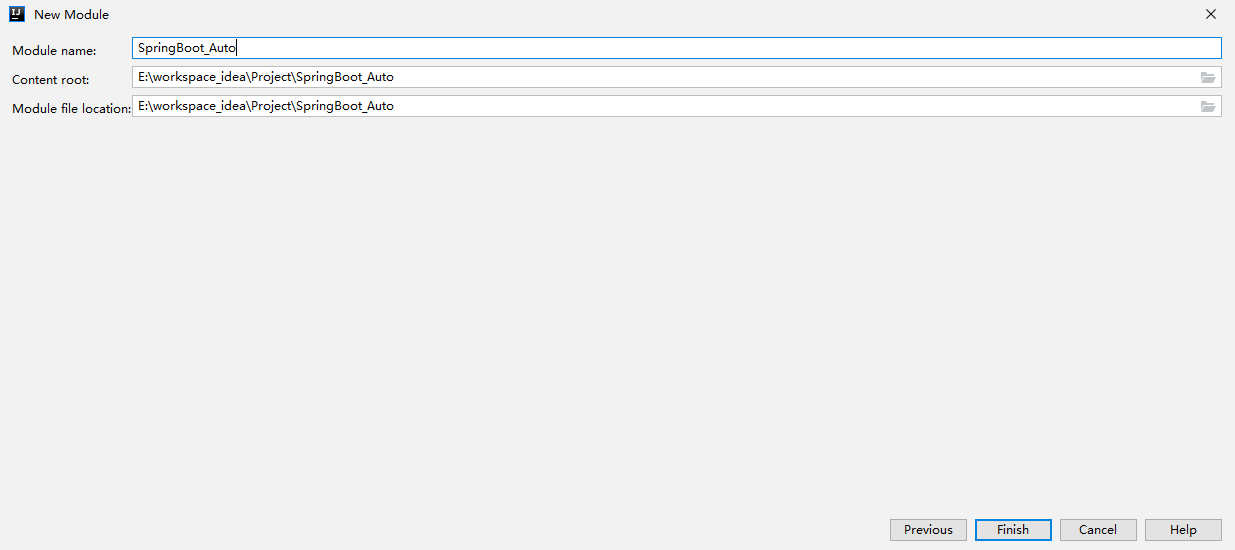
1. 给工程命名、设置包名等，其他默认即可



1. 选择工程的版本



1. 点击Next ，给工程命名，然后点击Finish



## 2.6 手动创建一个SpringBoot 项目

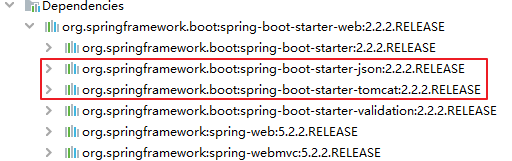
### 2.6.1 创建Maven项目

### 2.6.2 集成Spring Boot框架

* 修改pom.xml文件，增加Spring Boot框架的依赖关系及对Web环境的支持。

|  |
| --- |
| <project>  ...  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.5.RELEASE</version>  </parent>  ...  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies>  ... </project> |

* Spring Boot版本为官方最新正式版2.2.2.RELEASE
* 以往的项目中，所有类库的依赖关系都需要我们自己导入到pom.xml文件中，但是Spring Boot项目增加spring-boot-starter-web依赖后，会自动加载web环境配置相关依赖(SpringMVC, Tomcat)，简化了我们的操作。
* spring-boot-starter-parent：继承Spring Boot的相关参数
* spring-boot-starter-xxx：代表一个Spring Boot模块([参考附录1.Spring Boot相关模块](onenote:PN1.one#参考附录1.Spring Boot相关模块&section-id={9044B196-8600-4CA5-9DCA-52597144F9F4}&page-id={A4B380C1-3C9D-4BE5-8A77-38D6E8AE3A7B}&base-path=//D:/OneNote/Teacher_atcrowdfunding_new))
* spring-boot-starter-web：代表Web模块，在这个模块中包含了许多依赖的JAR包



### 2.6.3 增加程序代码

* 在src/main/java目录中增加类com.atguigu.springboot.SpringBootSelfApplication，并增加相应代码。

|  |
| --- |
| package com.atguigu.springboot;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  public class SpringBootSelfApplication {  public static void main(String[] args) {  SpringApplication.run(SpringBootSelfApplication.class, args);  }  } |

* Spring Boot项目中都会有一个以Application结尾的应用类，然后有一个标准的Java入口方法main方法。通过这个方法启动Spring Boot项目，方法中无需放入任何业务逻辑。
* @SpringBootApplication注解是Spring Boot核心注解
* 右键点击项目或项目中的SpringBootSelfApplication类, 选择菜单Run as Spring Boot App，启动SpringBoot项目.

### 2.6.4 集成Tomcat服务器

* SpringBoot内置了Tomcat，当增加Web依赖后执行main方法，等同于启动Tomcat服务器, 默认端口号为8080。如果想具体指定,通过server.port来指定
* 默认情况下SpringBoot启动后，默认的context-path的值为/，从浏览器端访问项目时，,不需要加项目名，直接通过 <http://localhost:8080/>请求名 来访问， 如果想具体指定，通过server.servlet.context-path来指定
* 例如:在src/main/resources/目录中增加application.properties文件。

|  |
| --- |
| server.servlet.context-path=/ server.port=80 |

* Spring Boot会自动读取src/main/resources/路径或着src/main/resources/config路径中的application.properties文件或application.yml文件。

### 2.6.5 为什么还会有配置文件

Spring Boot我们称之为微框架，这里的“微”不是小和少的意思，而是“简”的意思，简单，简洁。

项目中大部分的基础配置由Spring Boot框架帮我们自动集成，简化了我们的配置，但是框架自身为了扩展性，依然需要提供配置文件。

上面的代码中只是简单的应用了Spring Boot框架，但是我们真正要做的是将Spring Boot应用到项目中，所以接下来我们增加对SpringMVC框架，Mybatis框架的集成。

## 2.7 SpringBoot 集成 Spring & Spring Web MVC

* 基本的Spring Boot环境已经构建好了，现在需要配置Spring框架及SpringMVC框架的业务环境

### 2.7.1 @ComponentScan注解

* 通过@ComponentScan注解指定扫描的包

|  |
| --- |
| package com.atguigu.springboot;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.context.annotation.ComponentScan;  @ComponentScan(basePackages="com.atguigu")  @SpringBootApplication  public class SpringBootSelfApplication {    public static void main(String[] args) {  SpringApplication.run(SpringBootSelfApplication.class, args);  }  } |

* 默认扫描

默认扫描当前包com.atguigu.springboot和子包com.atguigu.springboot.\*

如果还需要扫描其他的包，那么需要增加@ComponentScan注解,指定包名进行扫描。

### 2.7.2 增加控制器代码

在src/main/java目录中增加类com.atguigu.springboot.controller.UserController，并增加相应代码。

|  |
| --- |
| package com.atguigu.springboot.controller;  import java.util.HashMap;  import java.util.Map;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.ResponseBody;  @Controller  public class UserController {    @ResponseBody //返回Json数据  @RequestMapping("/getAllUser") //指定请求URL  public Object getAllUser() {  Map<String,String> map = new HashMap<>();  map.put("username", "张三");  return map;  }  } |

### 2.7.3 执行main方法启动应用

访问路径http://localhost:8080[/应用路径名称]/ getAllUser页面打印JSON字符串即可



### 2.7.4 @Controller和@RestController区别

@RestController等同于@Controller + @ResponseBody，所以上面的代码可以变为：

|  |
| --- |
| package com.atguigu.springboot.controller;  import java.util.HashMap;  import java.util.Map;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class UserController {    @RequestMapping("/getAllUser") //指定请求URL  public Object getAllUser() {  Map<String,String> map = new HashMap<>();    map.put("username", "张三");    return map;  }  } |

## 2.8 SpringBoot集成通用Mapper

### 2.8.1 通用Mapper简介

通用mapper 可以极大的方便开发人员进行CRUD操作，提供极其方便的单表增删改查。一句话简单说，它就是个辅助mybatis极简单表开发的组件。它不是为了替代mybatis，而是让mybatis的开发更方便。

### 2.8.2 集成通用Mapper

1. 在pom.xml中加入通用Mapper的starter

|  |
| --- |
| <dependency>  <groupId>tk.mybatis</groupId>  <artifactId>mapper-spring-boot-starter</artifactId>  <version>2.0.2</version>  </dependency>    <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

1. 添加持久层代码

* 通用Mapper提供了Mapper接口，该接口中提供了常用的CRUD方法.
* 用户可以自己定义自己的Mapper接口，继承通用Mapper提供的Mapper接口，

|  |
| --- |
| package com.atguigu.springboot.mapper;  import com.atguigu.springboot.beans.User;  import tk.mybatis.mapper.common.Mapper;  public interface UserMapper extends Mapper<User> {  } |

1. 在src/main/resources下创建application.yml文件,配置数据源

|  |
| --- |
| # jdbc配置  spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://localhost:3306/bigdata?serverTimezone=UTC  username: root  password: 1234 |

## 2.9 整合测试

### 2.9.1 增加业务层代码

1. 增加业务层接口

|  |
| --- |
| package com.atguigu.springboot.service;  import java.util.List;  import com.atguigu.springboot.beans.User;  public interface UserService {  /\*\*  \* 查询所有的用户  \*/  public List<User> selectAllUser();  } |

1. 增加业务层实现类

|  |
| --- |
| package com.atguigu.springboot.service;  import java.util.List;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.atguigu.springboot.beans.User;  import com.atguigu.springboot.mapper.UserMapper;  @Service  public class UserServiceImpl implements UserService {    @Autowired  private UserMapper userMapper;    @Override  public List<User> selectAllUser() {  return userMapper.selectAll();  }  } |

### 2.9.2 增加控制层方法

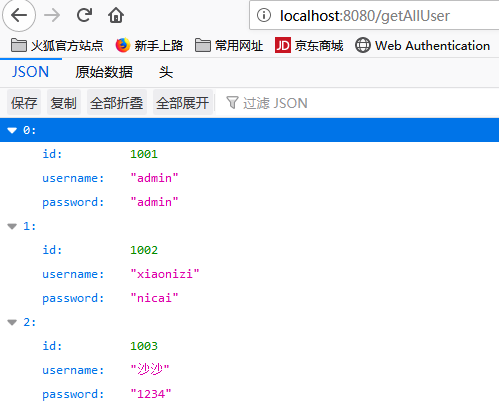
|  |
| --- |
| package com.atguigu.springboot.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  import com.atguigu.springboot.service.UserService;  @RestController  public class UserController {    @Autowired  private UserService userService;      @RequestMapping("/getAllUser") //指定请求URL  public Object getAllUser() {    return userService.selectAllUser();  }  } |

### 2.9.3 扫描Mapper

|  |
| --- |
| @MapperScan(basePackages = "com.atguigu.springboot.mapper")  @SpringBootApplication  public class SpringBootSelfApplication {  public static void main(String[] args) {  SpringApplication.run(SpringBootSelfApplication.class, args);  }  } |

### 2.9.4 测试

访问路径http://localhost:8080[/应用路径名称]/ getAllUser页面打印JSON字符串即可



## 2.10 Restful风格URL

### 2.10.1 REST 简介

REST（Representational State Transfer）又被称作表现层状态转换。它涉及到三个重要名词：

* 资源

所谓资源简单讲就是服务所能提供的数据，可以是实体数据也可是媒体类型，图片、PDF、文本等

* 表现层  
  何为变现层？简单说就是将数据以某种方式展现给用户，或者给客户返回一张图片等等动作称之为表现，通常是已JSON或XML形式展现数据
* 状态转换

状态转换就是对数据进行一系列的操作，因为资源本身并非一尘不变，随着需求的变化而变化。一个资源可能会随着需求的变化而经历一个资源创建、修改、查询、删除等过程，REST风格正是基于HTTP协议运行的，HTTP协议又被称为无状态协议，所以资源的变化需要在服务端完成，

简单用一句话概括就是：REST风格使用URL定位资源，用HTTP动词（GET,POST,DELETE,PUT）描述操作。

### 2.10.1 REST 规定

* GET请求
* 获取资源

例如：/emp/1

获取id=1的员工信息

* POST请求
* 添加资源

例如：/emp

添加员工信息

* PUT请求
* 更新资源

例如：/emp/1

更新id=1的员工信息

* DELETE请求
* 删除资源

例如：/emp/1

删除id=1的员工信息

### 2.10.2 Resulful风格URL 和普通URL对比

|  |
| --- |
| 普通URL: localhost:8888/SpringBootSelf/selectUser?id=1001&username=zhangsan    Resetful: localhost:8888/SpringBootSelf/selectUser/1001/zhangsan |

### 2.10.3 如何在后台处理Restful风格URL中的参数

* 客户端的URL: localhost:8888/SpringBootSelf/selectUser/1001
* 在@RequestMapping注解中使用 {} 占位符对应实际URL中的参数

|  |
| --- |
| @RequestMapping("/selectUser/{ids}")  public User selectUser(@PathVariable("ids") Integer id ) {  return userService.doSelectUser(id);  } |

* 在方法中使用@PathVariable注解指定将占位符对应的URL中的参数值赋值给方法的形参.

|  |
| --- |
| @RequestMapping("/selectUser/{ids}")  public User selectUser(@PathVariable("ids") Integer id ) {  return userService.doSelectUser(id);  } |

### 2.10.4 转换PUT请求和DELETE请求[了解]

* PUT请求和DELETE请求需要通过POST请求来转换
* 发送POST请求我们需要在form表单中发送，所以我们需要使用SpringBoot的模板
* 转换的步骤：
  1. 添加Thymeleaf模块

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> |

添加该模块后在main/resources目录下创建templates目录

* 1. 在templates目录下创建index.html页面，添加form表单，请求方式设置为post，表单中设置一个隐藏域，name属性值为\_method,

value值为put（转换为PUT请求时的值）或delete（转换为DELETE请求时的值）

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**>  <**a href="/getEmp/1"**>获取员工</**a**>  <**form action="/emp/4" method="post"**>  <**input type="hidden" name="\_method" value="delete"**><**br**>  <**input type="submit" value="删除员工"**>  </**form**> </**body**> </**html**> |

* 1. 配置过滤器
* 创建一个类继承HiddenHttpMethodFilter
* 在类上添加@WebFilter

|  |
| --- |
| **import** org.springframework.web.filter.HiddenHttpMethodFilter;  **import** javax.servlet.annotation.WebFilter;  @WebFilter **public class** MyFilter **extends** HiddenHttpMethodFilter { } |

* 1. 在启动类上添加@ServletComponentScan注解

|  |
| --- |
| @ServletComponentScan @MapperScan(basePackages = **"com.atguigu.springboot.mapper"**) @SpringBootApplication **public class** SpringBootSelfApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootSelfApplication.**class**, args);  } } |

# 附录

## 附录1 SpringBoot相关模块

