# Spring

## 1. Spring入门

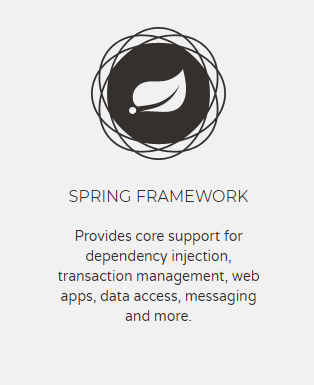
### 1.1 Spring Framework

**框架**（Framework）：提供某一领域完整的解决方案，使开发更具有工程性、简便性、稳定性，更有效率。

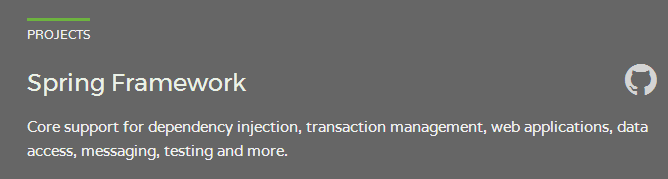
**Spring**是一个开源的简化企业级开发的容器框架。

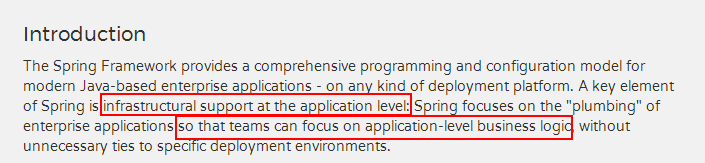
**目的**：简化JavaEE的开发

Spring发展至今，已经产生了关乎JavaEE企业级开发方方面面的框架，我们学习的是Spring Framework



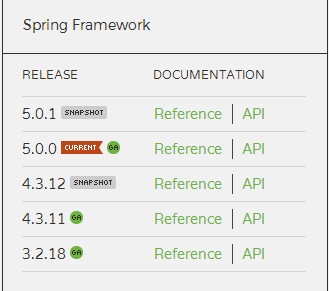
**地址**：https://spring.io/



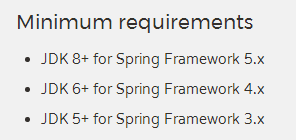


在应用程序层面提供最基础的支持；使开发团队可以专注于应用的业务逻辑开发。

**发行版**：



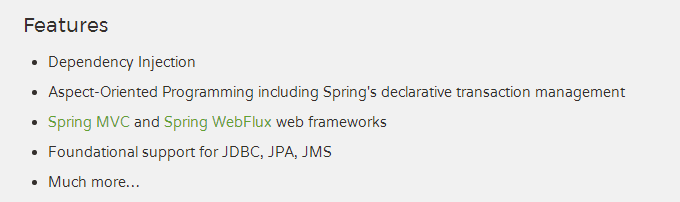
**版本要求**：



### 1.2 显著特点

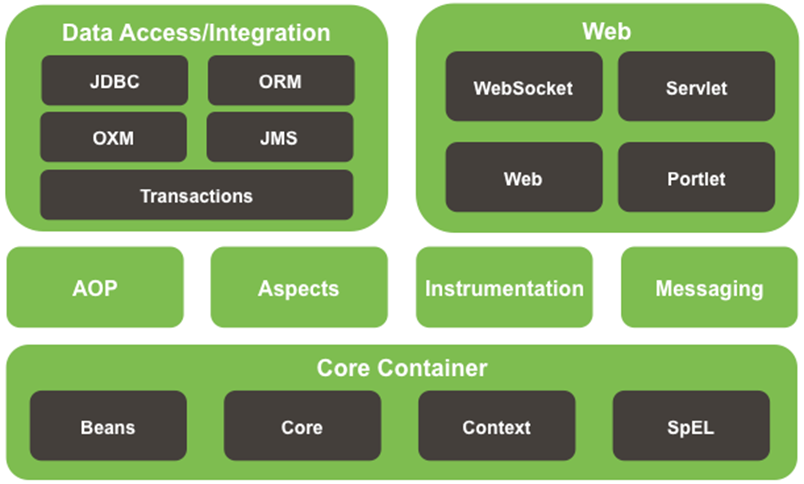
**IOC**（Inversion of control）控制反转 DI

**AOP**(Aspect Oriented Programming)的容器框架



### 1.3 功能丰富

几乎覆盖JavaEE开发的方方面面



### 1.4 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。

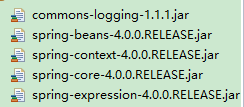
[6]**一站式**：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的Spring JDBC）。

### 1.5 Jar包描述：

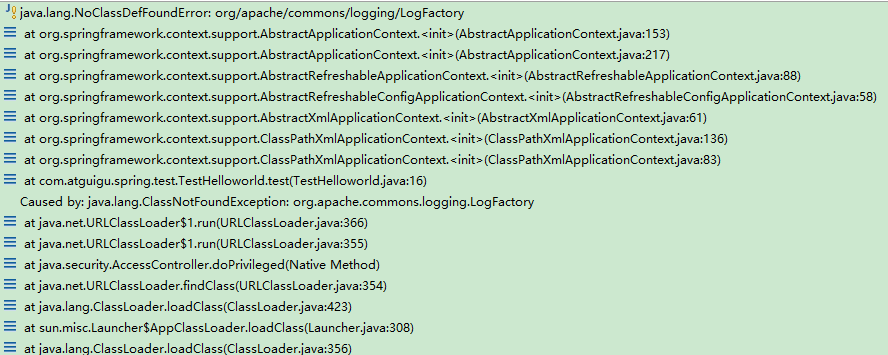
|  |  |
| --- | --- |
| 核心容器模块 | |
| org.springframework spring-beans | bean支持，包括Groovy |
| org.springframework spring-context | 运行时上下文，包括调度和远程调用抽象 |
| org.springframework spring-core | 核心库，被许多其它模块使用 |
| org.springframework spring-expression | Spring表达式语言 |
| AOP+Aspects模块 | |
| org.springframework spring-aop | 基于代理的AOP |
| org.springframework spring-aspects | 基于切面的AspectJ |
| 数据库访问模块 | |
| org.springframework spring-jdbc | JDBC支持包 |
| org.springframework spring-orm | 对象关系映射，包括对JPA和Hibernate支持 |
| org.springframework spring-oxm | 对象XML映射 |
| org.springframework spring-tx | 事务基础，包括对DAO的支持及JCA的集成 |
| org.springframework spring-jms | JMS支持包，包括发送和接收JMS消息的帮助类 |
| web应用开发模块 | |
| org.springframework spring-web | web支持包，包括客户端及web远程调用 |
| org.springframework spring-webmvc | REST web服务及web应用的MVC实现 |
| org.springframework spring-webmvc-portlet | 用于Portlet环境的MVC实现 |
| org.springframework spring-websocket | WebSocket和SockJS实现，包括对STOMP的支持 |
| 单元测试模块 | |
| org.springframework spring-test | 单元测试和集成测试组件 |
| 第三方集成 | |
| org.springframework spring-context-support | 包含用于集成第三方库到Spring上下文的类 |
| 消息框架 | |
| org.springframework spring-messaging | 消息处理的架构和协议 |
| 检测代理 | |
| spring-instrument-4.0.0.RELEASE | JVM引导的检测代理 |
| org.springframework spring-instrument-tomcat | tomcat的检测代理 |

### 1.6 Helloworld

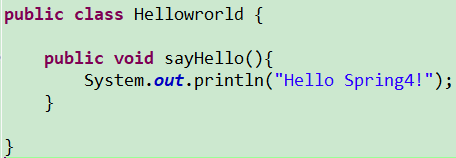
①导包



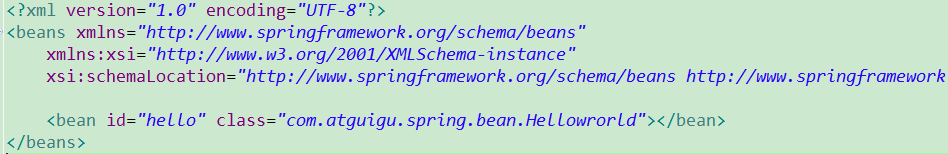
导入的jar包中包括，4个Spring的核心Jar包与其依赖的一个日志框架commons-logging，如果不加入，会报错如下：



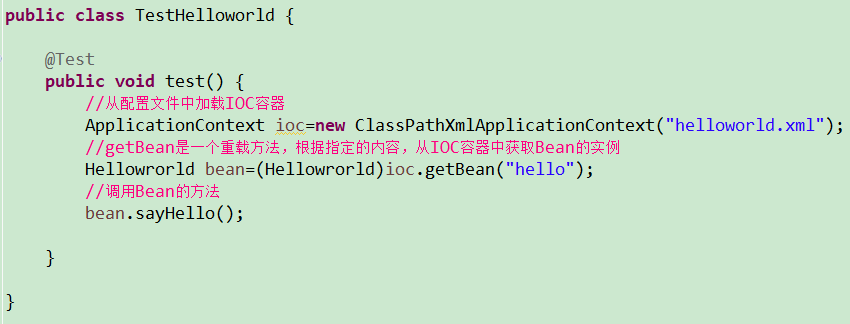
②编写一个类



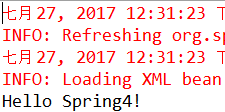
③编写配置文件，在配置文件中配置Bean



④进行测试



⑤测试通过



### 1.7 Helloworld细节

①src目录：源码路径，也称为类路径。所有源码包里面的东西都会被合并在类路径里面：

在 java 工程中是: 项目名/bin/

在 javaweb工程中是 项目名/build/classes/

②使用容器，即 Application（应用上下文）接口来管理组件。Application有两个常用的实现类：

ClassPathXmlApplicationContext：从类路径下的XML配置文件中创建应用上下文对象

FileSystemXmlApplicationContext：从文件系统中的XML配置文件中创建应用上下文对象

③容器一创建就会自动创建配置文件中配置的bean，且默认创建的 bean是单例的

④getBean():从容器中取出bean

getBean(String id):通过id取得Bean，获取Bean的返回值类型是Object，需要强转。如果id不存在，会报错NoSuchBeanDefinitionException

getBean(Class T)：通过类型获取Bean，会返回指定类型的Bean，但是要保证容器中只有唯一此类型的Bean存在，否则会报错NoSuchUniqueBeanException

getBean(String id,Class T):从容器中取出指定id，指定类型的bean。

## 2. IOC

### 2.1 IOC

**I**nversion **o**f **C**ontrol：控制反转；

控制:资源的获取方式

①传统: 主动地获取资源（需要自己去创建所需的资源）

BookServlet{

BookService bs = new BookService();

//复杂的系统功能可能同时需要创建多个资源

XXXService xx=new xxxService();

public void test01(){

bs.checkout();//

}

②IOC方式：资源不需要自动创建，而是交给一个由应用程序组件所在的容器创建，并自动的注入到需要资源的组件中。

BookServlet{

BookService bs;

public void test01(){

bs.checkout();//

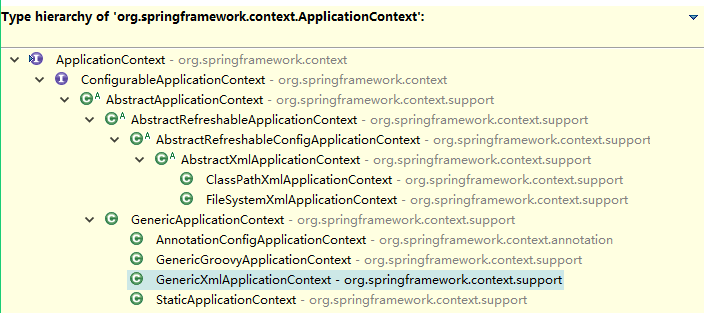
}

}

**优点**：开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率

IOC在Spring中的典型实现就是容器。容器是Spring框架的核心。在通过IOC容器读取Bean的实例之前，需要先将IOC容器本身实例化。

在Spring中，使用ApplicationContext作为容器的实现类



### 2.2 在容器中为Bean赋值

<property>标签通过setXXX()赋值，其中name属性对应的是setXxx()方法中的Xxx。

<bean class=*"com.dorr.spring.bean.Person"* id=*"gg"*>

<!--使用property标签赋值

name：属性名

value：属性值

-->

<property name=*"name"* value=*"国哥"*/>

<property name=*"age"* value=*"18"*/>

</bean>

### 2.3 可以用值的类型

#### 2.3.1 字面量

字面量表示可以使用字符串表示的值，包括基本数据类型及其包装类，String。

字面量可以通过value属性或<value>标签来指定。

#### 2.3.2 null值

null值不属于字面量，需要使用<null/>来赋值！

<bean class=*"com.dorr.spring.bean.Person"* id=*"cfl"*>

<!-- <property name="gender" value="null"></property> -->

<!-- 使用null标签-->

<property name=*"gender"*><null/></property>

</bean>

#### 2.3.3 外部bean

外部bean就是普通的bean，声明在外部，可以通过ref属性结合id直接引用！

<bean class=*"com.dorr.spring.bean.Person"* id=*"cfl"*>

<property name=*"name"* value=*"&lt;苍飞龙&gt;"*/>

<property name=*"phone"* ref=*"phone"*/>

</bean>

<bean class=*"com.dorr.spring.bean.Phone"* id=*"phone"*>

<property name=*"brand"* value=*"Mi"*/>

<property name=*"type"* value=*"7"*/>

</bean>

#### 2.3.4 内部bean

<bean class=*"com.dorr.spring.bean.Person"* id=*"fl"*>

<property name=*"phone"* >

<!-- 内部bean只能被当前的bean所使用，无法通过id获取 -->

<bean class=*"com.dorr.spring.bean.Phone"* >

<property name=*"brand"* value=*"Mi"*/>

<property name=*"type"* value=*"7"*/>

</bean>

</property>

</bean>

内部bean代表只能被这个bean内部使用的一个bean，无法被外部访问，因此id属性无意义。

### 3. DI(依赖注入)

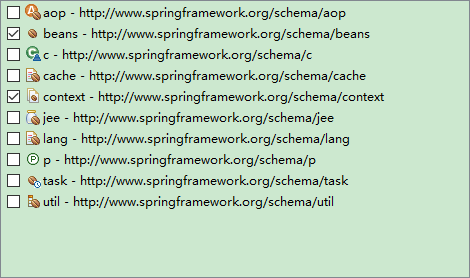
之前我们已经学习了通过配置文件的方式来装配组件。除了配置文件的方式，我们还可以使用注解来实现组件的自动创建和装配。

相对于xml方式的自动装配而言，通过注解的方式配置bean会更加的优雅和简洁，并与MVC的组件化开发理念契合，是开发汇总常用的使用方式

要使用自动装配，还需要导入 AO的包（spring-aop-4.0.0.RELEASE），否则，会报错：java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource

#### 3.1 容器自动创建Bean

要使用DI功能，需要首先开启context名称空间：



之后使用<context:component-scan>标签指定要扫描的父路径。

<context:component-scan base-package=*"com.dorr.spring"* >

之后容器会扫描指定的路径，为标有指定注解的bean，自动注册为spring的bean，支持以下注解：

@Component,： 标识一个普通的对象

@Repository：标识一个持久化层的组件

@Service： 标识一个服务层的组件

@Controller ：标识一个控制层的组件

注意：

①这四个注解功能是一样的。主要是为了给开发人员看，以区别每个类的功能

②base-package:指定扫描的父路径，会自动扫描其下所有子路径

③默认将类名首字母小写作为id，创建组件；还可以通过注解的value属性，手动指定id。

#### 3.2 容器自动装配Bean

①在需要自动赋值的属性上标注@Autowired注解，容器会自动扫描容器中的组件，并自动赋值。

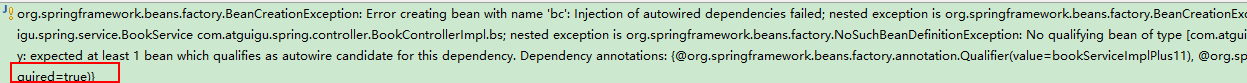


②自动装配的规则：

首先按照类型匹配，如果匹配到单个，就直接装配；

如果匹配到多个，继续按照属性名作为id进一步匹配；

如果匹配不到，报错！报错如下图，原因是@Autowired默认属性required=true，意思就是强制装配。



③取消强制装配：

修改@Autowired注解的required属性为false即可

④指定装配指定id的组件：

在要装配的组件上标注@Quliafier(xxx)后，将按照xxx作为id来进行装配！

⑤一般情况下，需要将@Quliafier和required=false注解结合使用，以防止容器中无法匹配到指定id的bean从而报错。

⑥@Autowired属性还可以标注在方法参数上，此时这个方法会在Bean创建后就立刻执行。

@Autowired

**public** **void** init(BookDao bookDao,@Qualifier("bookServiceImpl2")BookService bookService){

System.***out***.println("BookDao"+bookDao);

System.***out***.println("BookService"+bookService);

}

⑦@Autowired与@Resource的区别

@Resource JSR250 (Common Annotations for Java)这是jsr250规范的实现，通过 ‘CommonAnnotationBeanPostProcessor’ 类实现依赖注入。

与@Autowired的区别：

@Autowired：org.springframework.beans.factory.annotation.Autowired 是Spring框架自己的注解

@Resource：javax.annotation.Resource 是Java的一种标准，扩展性强，可以被EJB等容器框架识别

#### 3.3 容器扫描的过滤与包含

在<context:component-scan>中可以通过声明<include-filter>标签来指定只扫描哪些组件；

在<context:component-scan>中可以通过声明<exclude-filter>标签来指定不扫描哪些组件；

<include-filter>和<exclude-filter>可以存在多个；

<include-filter>必须结合use-default-filters="false"属性配合使用才有效；

常见的过滤属性如下：

type： 过滤规则的类型

* + - * annotation：注解，传入注解全类名，所有标注了这个注解的类，都会被过滤掉

按照@Component注解进行过滤，会过滤所有的组件！

* + - * assignable： 签名,传入全类名或者接口名
      * aspectJ ： 参考aspectJ表达式
      * regex： 参考正则表达式
      * custom ： 自定义，需要实现TypeFilter接口

expression：表达式

#### 3.4 配置文件和注解开发的选择

一些简单的业务逻辑，可以使用注解快速开发！

但是，比较重要的类，可能经常做修改的类，我们一般放到配置文件中！

第三方编写的类，只能在配置文件中配置！

具体参考团队开发约束和风格！

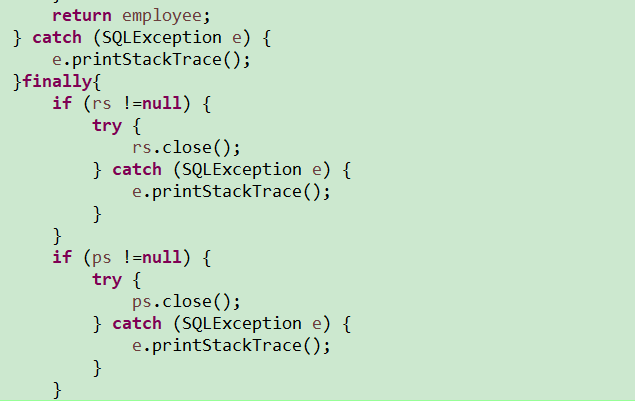
# 二、MyBatis

## 1. MyBatis简介

Mybatis是一个java编写的轻量级的半自动的ORM映射的Dao层框架。

在学习Mybatis之前，我们已经接触过原生JDBC的方式。





JDBC的弊端：SQL夹在Java代码块里，耦合度高导致硬编码内伤，维护不易。而实际开发需求中sql是有变化，频繁修改的情况多见。

Hibernate（全自动全映射的ORM框架）的弊端，自动生产的多长、难、复杂SQL，不容易做特殊优化。

对开发人员而言，核心sql还是需要自己优化+掌握在自己手上，简单来说，就是一句话，**sql和java编码分开，功能边界清晰；一个专注业务、一个专注数据；二者之间通过简单的XML或注解用于配置和原始映射，将接口和Java的POJO映射成数据库中的记录，完成业务代码+底层数据库的媒介**

**Mybatis的最大优点： Java代码与SQL分层解耦**

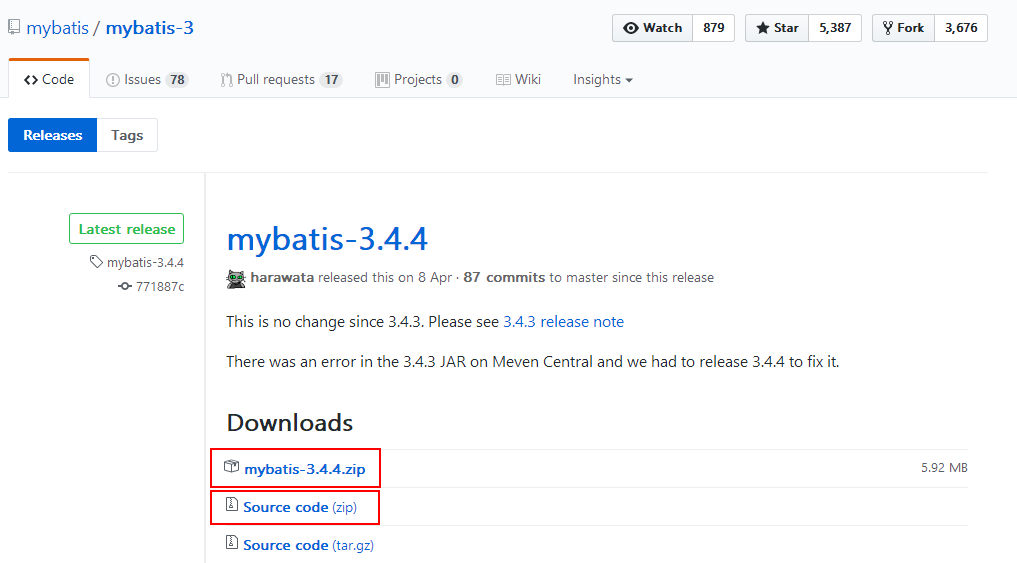
原是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，

随着开发团队转投Google Code旗下，ibatis3.x正式更名为Mybatis ，代码于2013年11月迁移到Github（下载地址：https://github.com/mybatis/mybatis-3/releases）。

iBATIS一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAO）

MyBatis消除了几乎所有的JDBC代码和参数的手工设置以及对结果集的检索封装。MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录.

本质是一个将SQL语句map到JAVA POJO的框架。



## 2. 准备工作

①建库建表

CREATE TABLE tbl\_employee(

id INT(11) PRIMARY KEY AUTO\_INCREMENT,

last\_name VARCHAR(255),

gender VARCHAR(10),

email VARCHAR(255)

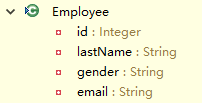
);

INSERT INTO tbl\_employee(last\_name,gender,email) VALUES('Tom','male','Tom@163.com');

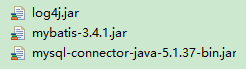
INSERT INTO tbl\_employee(last\_name,gender,email) VALUES('Jack','male','Jack@163.com');

INSERT INTO tbl\_employee(last\_name,gender,email) VALUES('Marry','female','Tom@163.com');

②POJO对象：Employee类



③建工程，导jar包



④加入log4j的配置文件

⑤加入mybatis的配置文件

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

<!DOCTYPE configuration

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

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

<configuration>

<!-- environmets标签里可以配置多个environment标签 -->

<environments default=*"development"*>

<!-- 一个environmet就代表一个数据库运行环境 -->

<environment id=*"development"*>

<!-- transactionManager：事务管理器 -->

<transactionManager type=*"JDBC"* />

<!--dataSource：数据源，需要配置用户名，密码，驱动，连接 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/1115\_mybatis"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"1234"* />

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource=*"helloworld.xml"* />

<mapper resource=*"EmployeeMapper.xml"*/>

<!-- 注册接口类型 -->

<mapper class=*"com.dorr.mybatis.mapper.EmployeeAnnotationMapper"*/>

</mappers>

</configuration>

⑥配置数据库连接信息

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"passw ord"* value=*"1234"*/>

## 3. Helloworld

mybatis的优点在于java代码和sql语句相分离。

sql语句编写在xxxMapper文件中，同时注意需要在全局配置文件中注册!

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

<!DOCTYPE mapper

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

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

<mapper namespace=*"helloworld"*>

<select id=*"selectEmployee"* resultType=*"com.dorr.mybatis.bean.Employee"*>

select id,last\_name lastName,gender,email from tbl\_employee where id=#{id}

</select>

</mapper>

@Test

**public** **void** test() **throws** IOException{

//从当前的类路径获取mybatis的配置文件

String resource = "mybatis\_config.xml";

//使用一个流读取配置文件

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//根据配置文件，创建一个SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

**try**{

Object emp = session.selectOne("helloworld.selectEmployee", "1");

System.***out***.println(emp);

}**finally**{

session.close();

}

}

## 4. 接口式编程

使用原生API进行开发有以下不便之处：

①我们更倾向于使用Dao---DaoImpl来分层解耦

②方法的返回值，是Object类型，不能直接使用，需要强转

③方法的入参无法进行严格的检查

因此，在后续的mybatis版本中，推荐使用接口式编程开发！

要求：

①编写接口，mybatis习惯上以mapper作为接口文件的命名

**public** **interface** EmployeeMapper {

**public** Employee getEmployeeById(Integer id);

}

②编写sql语句的配置文件，并与接口中的方法进行绑定

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

<!DOCTYPE mapper

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

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

<mapper namespace=*"com.dorr.mybatis.mapper.EmployeeMapper"*>

<select id=*"getEmployeeById"*

resultType=*"com.dorr.mybatis.bean.Employee"* >

select id,last\_name lastName,gender,email from tbl\_employee where id=#{id}

</select>

要求：

namespace:必须和接口的全类名一致

sql 的id： 必须和对应的方法名一致

select 标签中，返回值类型及参数类型也必须和对应的方法一致，parameterType参数类型也可以不写，mybatis会根据 typeHandler自动判断

③通过sqlsession的getMapper()方法获取接口的代理对象，执行其方法

@Test

**public** **void** testMapper(){

SqlSession sqlSession = sqlSessionFactory.openSession();

**try**{

//执行数据库的CRUD

EmployeeMapper mapper = sqlSession.getMapper(EmployeeMapper.**class**);

System.***out***.println(mapper.getClass().getName());//com.sun.proxy.$Proxy3

Employee employee = mapper.getEmployeeById(1);

System.***out***.println(employee);

}**finally**{

sqlSession.close();

}

}

## 5. CRUD

接口式编程流程 XXXMapper.java---->XXXMapper.xml---->在全局配置文件中注册---->接口式调用

①xxxMapper.java文件

**public** **interface** EmployeeMapper {

**public** Employee getEmployeeById(Integer id);

**public** **void** addEmployee(Employee emp);

**public** **void** deleteEmployee(Integer id);

**public** **void** updateEmployeeEmail(Employee emp);

**public** List<Employee> getAll();

}

②xxxMapper.xml文件

<mapper namespace=*"com.dorr.mapper.EmployeeMapper"*>

<select id=*"getEmployeeById"* resultType=*"com.dorr.bean.Employee"*>

SELECT `id`,`last\_name` lastName,`gender`,`email` FROM `tbl\_employee` WHERE `id`=#{id}

</select>

<insert id=*"addEmployee"* parameterType=*"com.dorr.bean.Employee"*>

INSERT INTO `tbl\_employee`(`last\_name`,`gender`,`email`) VALUES(#{lastName},#{gender},#{email})

</insert>

<delete id=*"deleteEmployee"* parameterType=*"int"*>

DELETE FROM `tbl\_employee` WHERE id=#{id}

</delete>

<update id=*"updateEmployeeEmail"* parameterType=*"com.dorr.bean.Employee"*>

UPDATE `tbl\_employee` SET `email`=#{email} WHERE `id`=#{id}

</update>

<select id=*"getAll"* resultType=*"com.dorr.bean.Employee"*>

SELECT `id`,`last\_name` lastName,`gender`,`email` FROM `tbl\_employee` </select>

③在全局配置文件中注册xxxMapper.xml配置文件

<mappers>

<mapper resource=*"com/dorr/bean/EmployeeMapper.xml"* />

</mappers>

④调用，以添加为例

@Test

**public** **void** testAddEmployee() **throws** IOException {

SqlSession sqlSession = sqlSessionFactory.openSession();

**try**{

EmployeeMapper mapper = sqlSession.getMapper(EmployeeMapper.**class**);

Employee employee=**new** Employee(**null**, "Jackie", "male", "Jackie@qq.com");

mapper.addEmployee(employee);

sqlSession.commit();

}**finally**{

sqlSession.close();

}

}

## 6. SqlSession的细节

①sqlSession 代表和数据库的一次会话。sqlSession不是线程安全的，不能被共享！

因此它的范围最好是一个方法对应自己的sqlSession，或一次请求，创建一个sqlSession。

②SqlSession在每次方法执行完成之后，必须保证关闭，因此我们常常在方法的finally语句块中，执行其close()方法，将其关闭。

# 三、SpringMVC

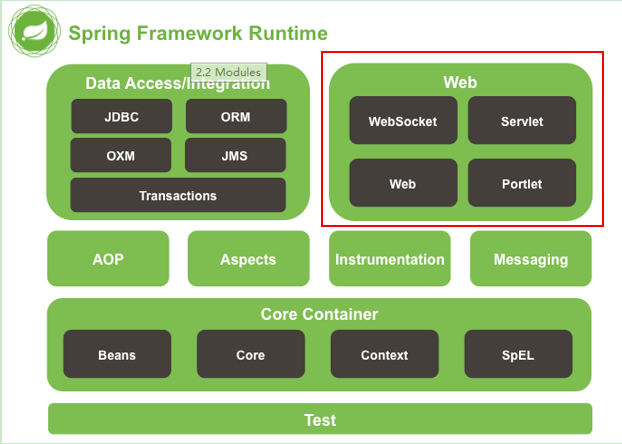
## 1. SpringMVC简介

### 1.1 SpringMVC概念

Spring: 是一个容器框架，用来简化开发！

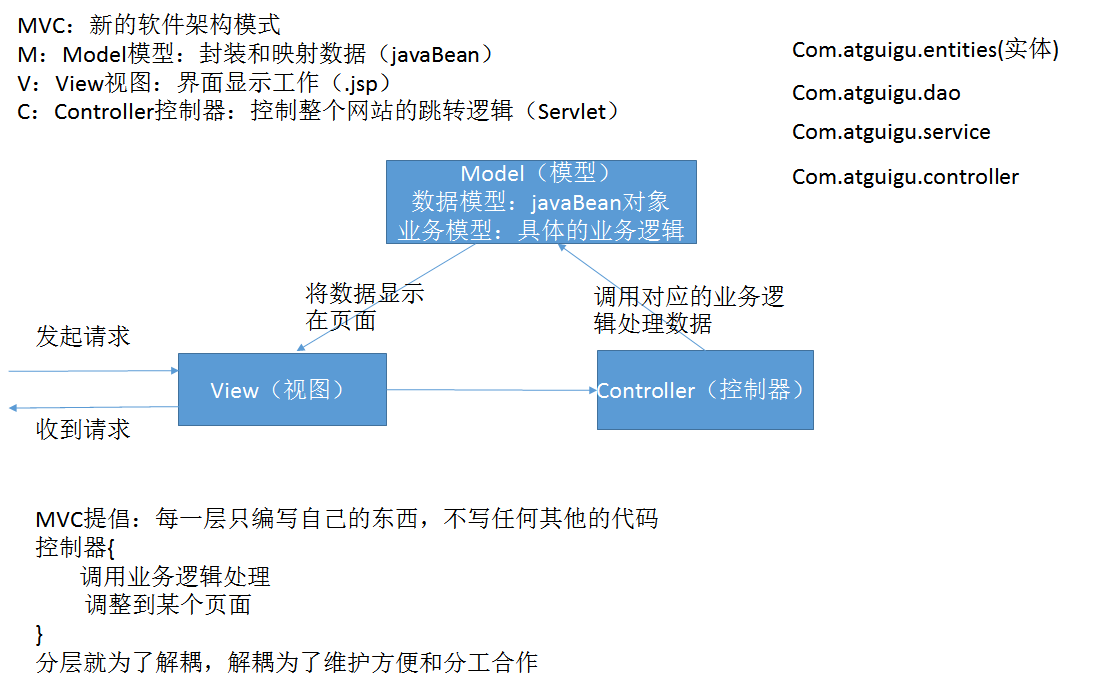
MVC： 是一种web开发的设计思想，分层解耦；

SpringMVC： Spring提供的符合MVC设计思想的，简化web开发的模块。



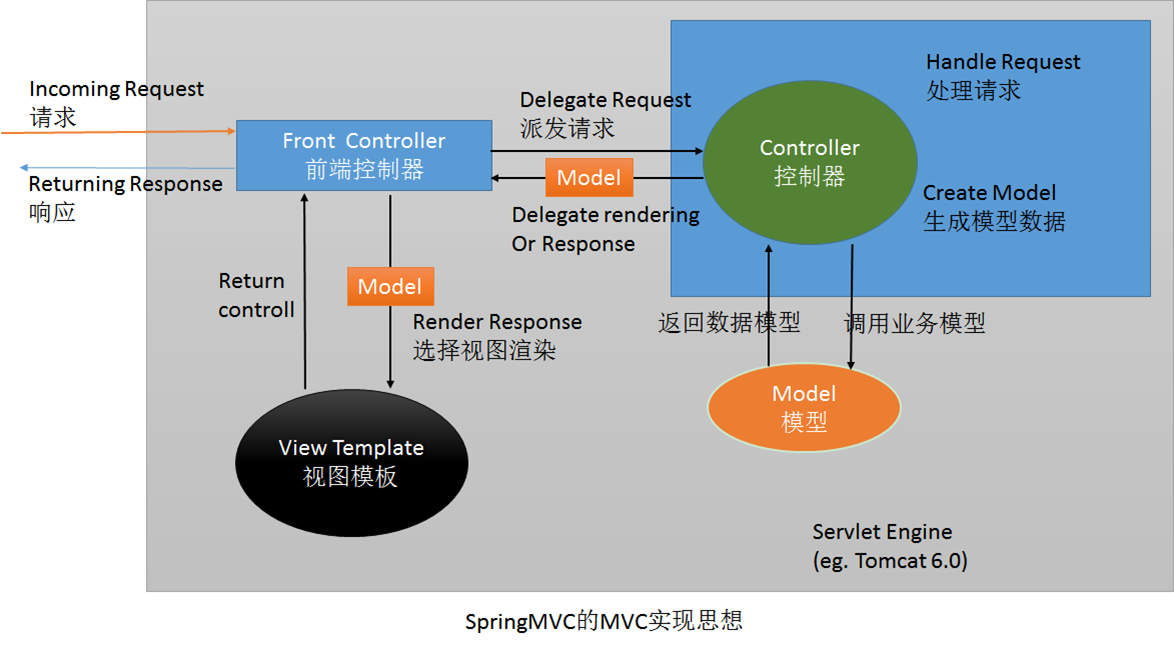
### 1.2 SpringMVC的核心

传统的MVC：



SpringMVC遵循了MVC的开发模式，但是做了一些改进：

在传统的MVC的基础上，增加了一个前端控制器来只能派发所有的请求！

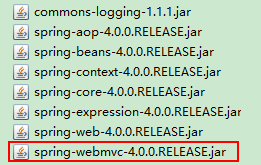


### 1.3 SpringMVC优点

* **Spring MVC 通过一套 MVC 注解**，让 POJO 成为处理请求的控制器，而无须实现任何接口。
* **支持 REST 风格的 URL 请求**
* 采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性

## 2. Helloworld

### 2.1. 建工程，导jar包

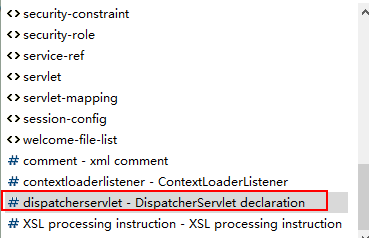


### 2.2 配置

#### 2.2.1配置前端控制器

SpringMVC的核心是前端控制器，所有的请求都需要用前端控制器来做智能派发。前端控制器实际上是一个Servlet！

使用alt+/，选择DispatcherServlet代码片段！



<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<!--配置文件的路径 -->

<param-name>contextConfigLocation</param-name>

<param-value>classpath:beans.xml</param-value>

</init-param>

<!-- servlet被创建的时机： 请求第一次访问的时候创建

load-on-startup：配置servlet创建的策略，随着web应用的一创建就创建

后边配置一个Int型的参数，代表创建的优先级，数字越小，优先级越高！

-->

<load-on-startup>1</load-on-startup>

</servlet>

<!-- 处理哪些请求 -->

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<!-- 配置拦截所有请求：

注意：只能写/

/\*:不能写

-->

<url-pattern>/</url-pattern>

</servlet-mapping>

#### 2.2.2 配置Spring的配置文件



### 2.3 代码编写

#### 2.3.1 页面、成功页面

#### 2.3.2 处理器

@Controller

**public** **class** HelloHandler {

/\*

\* 1. 路径分为相对路径和绝对路径（/）

\* 浏览器端解析： html http://ip:端口号

\* 服务器端解析： http://ip:端口号：项目名

\*/

@RequestMapping("/hello")

**public** String hello(){

System.***out***.println("进入了hello请求！");

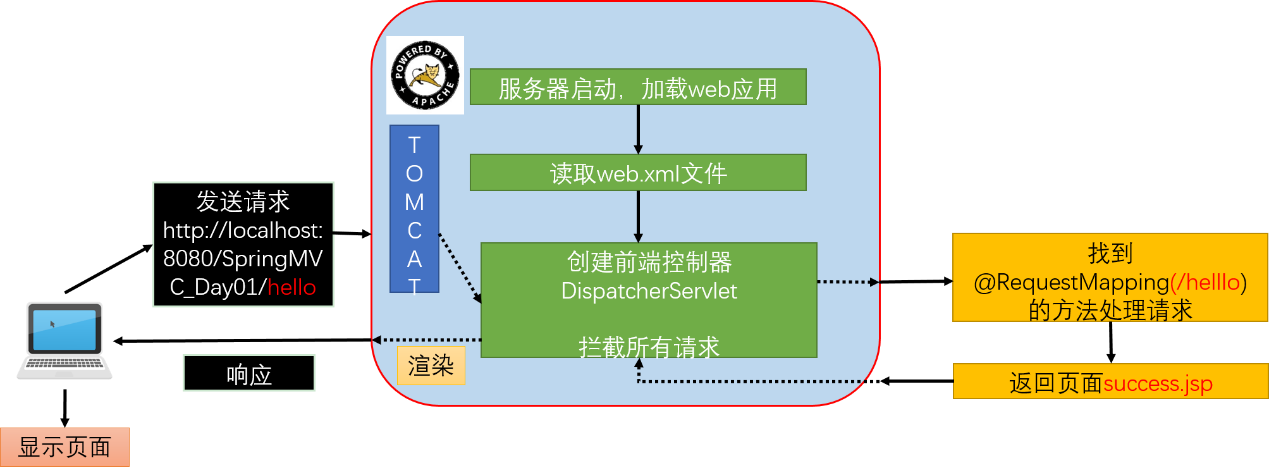
//request.getRequestDispatcher("xx").forward(request.response);

**return** "/WEB-INF/pages/success.jsp";

}

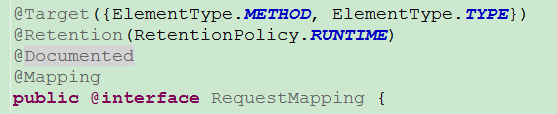
}

#### 2.3.3.流程



## 3. @RequestMapping

### 3.1 概述



@RequestMapping注解负责处理请求，可以标注在类和方法上！

①标注在方法上： 可以处理哪些请求

②标注在类上：相当于为这个类的所有方法的@RequestMapping属性，添加了一个基准路径（父路径）

通常不同的模块，可以使用不同的基准路径！

### 3.2 属性

#### 3.2.1 value

value: String[ ] ,可以映射的请求，多个请求用,隔开

#### 3.2.2 method

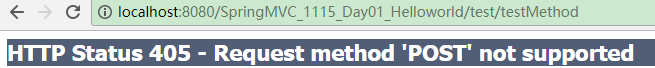
method: 限制这个请求的请求方式

请求方式：【GET】, HEAD, 【POST】, PUT, PATCH, DELETE, OPTIONS, TRACE

如果请求方式不匹配，报错405；

4xx: 客户的错误

5xx: 服务端错误



## 4. REST

### 4.1 @PathVariable

①@PathVariable:获取指定请求路径上的变量值！

②如果路径无法匹配，就报错！

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

**public** String handle(@PathVariable("id")Integer id){

System.***out***.println("路径上的id:"+id);

**return** "success";

}

### 4.2 REST概念

REST：即 Representational State Transfer。**（资源）表现层状态转化。是目前最流行的一种互联网软件架构**。

* **资源（Resources）**：网络上的一个实体，或者说是网络上的一个具体信息。

它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。

可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的 URI 。

获取这个资源，访问它的URI就可以，因此 URI 即为每一个资源的独一无二的识别符。

* **表现层（Representation）**：把资源具体呈现出来的形式，叫做它的表现层（Representation）。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。
* **状态转化（State Transfer）**：每发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。

而这种转化是建立在表现层之上的，所以就是 “表现层状态转化”。

* **具体说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。**

**它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。**

总结：

①REST是一种思想。REST推崇简洁的URL表达！

②REST这种思想，认为万物皆资源，所有的请求都是为了获取资源，通过HTTP协议中请求方式的不同，表达对资源的不同状态的修改！即GET对应查询、PUT对应修改、DELETE对应删除、POST对应添加。

REST推荐的URL格式：

/操作的资源类型/资源标识符 以请求方式的不同，区别对资源的不同操作（CRUD）

|  |  |  |
| --- | --- | --- |
| 以对Book的增删改查为例 | 过去URL | 使用REST的URL |
| 增 | /addBook? bookName=xx | /book（发送POST请求） |
| 删 | /deleteBook?bookId=1 | /book/1 (发送DELETE请求) |
| 改 | /updateBook?bookId=1&bookName=xx | /book/1(发送PUT请求) |
| 查 | /getBook?bookId=1 | /**book**/1(发送GET请求) |

### 4.3 简单的REST示例

问题：GET和POST请求很常见，而PUT和DELETE请求如何发？

解决：

①配置一个请求方式过滤器HiddenHttpMethodFilter，这个过滤器可以将POST请求转换为相应的PUT、DELETE

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name> <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class></filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

②创建一个method=post的表单

③表单中有名为\_method的属性，值为要发的请求方式（PUT,DELETE）不区分大小写

<a href=*"book/1"*>查询图书</a><br>

<form action=*"book"* method=*"post"*>

<input value=*"新增"* type=*"submit"*>

</form><br>

<form action=*"book/1"* method=*"post"*>

<input name=*"\_method"* value=*"delete"* type=*"hidden"*>

<input value=*"删除"* type=*"submit"*>

</form><br>

<form action=*"book/1"* method=*"post"*>

<input name=*"\_method"* value=*"put"* type=*"hidden"*>

<input value=*"修改"* type=*"submit"*>

</form><br>

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

**public** String getBook(@PathVariable("id")Integer id){

System.***out***.println("查询"+id+"号图书！");

**return** "success";

}

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

**public** String addBook(){

System.***out***.println("新增了一本图书！");

**return** "success";

}

@RequestMapping(value="/book/{id}",method=RequestMethod.***DELETE***)

**public** String deleteBook(@PathVariable("id")Integer id){

System.***out***.println("删除"+id+"号图书！");

**return** "success";

}

@RequestMapping(value="/book/{id}",method=RequestMethod.***PUT***)

**public** String updateBook(@PathVariable("id")Integer id){

System.***out***.println("修改"+id+"号图书！");

**return** "success";

}

### 4.4 HiddenHttpMethodFilter原理

HiddenHttpMethodFilter的doFilterInternal方法中，对请求方式进行了转换

@Override

**protected** **void** doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

**throws** ServletException, IOException {

//从请求参数中获取名称为 \_method的属性，这个属性就是我们自己写的PUT，DELETE

String paramValue = request.getParameter(**this**.methodParam);

//判断，当前是POST请求，且有\_method属性

**if** ("POST".equals(request.getMethod()) && StringUtils.*hasLength*(paramValue)) {

//将\_method属性转换为大写

String method = paramValue.toUpperCase(Locale.***ENGLISH***);

//将原来的请求，再次包装。在HttpMethodRequestWrapper里面将请求设置为我们的PUT，DELETE，并重写了getMethod()

HttpServletRequest wrapper = **new** HttpMethodRequestWrapper(request, method);

//放行包装后的请求

filterChain.doFilter(wrapper, response);

}

**else** {

//GET请求，和POST请求，直接放行

filterChain.doFilter(request, response);

}

}

**private** **static** **class** HttpMethodRequestWrapper **extends** HttpServletRequestWrapper {

**private** **final** String method;

**public** HttpMethodRequestWrapper(HttpServletRequest request, String method) {

**super**(request);

**this**.method = method;

}

@Override

**public** String getMethod() {

**return** **this**.method;

}

}

## 5. 处理请求

### 5.1 @RequestParam 获取请求参数

在处理方法入参处使用 @RequestParam 可以把请求参数传递给请求方法，功能类似request.getParameter("xx")。

**常见属性**：

value：参数名

required：是否必须。默认为 true, 表示请求参数中必须包含对应的参数，若不存在，将抛出异常

defaultValue:如果无法正确获取参数，就指定一个默认值。如果不指定，默认是null

**使用方法**：

① 默认情况下，只需要在处理方法的参数上，声明与请求参数名一致的形参即可。但是要注意方法参数名 与请求参数名要一致，否则无法自动赋值。

②如果遇到形参与请求参数不一致，可以使用@RequestParam("xx")明确指定要获取的参数名。

③一旦使用@RequestParam注解，如果无法找到指定的参数，就报错400。原因是默认@RequestParam(required=true)。可以通过修改@RequestParam(required=false)来解决。

### 5.2 @RequestHeader

**作用**：从请求中获取请求头对应的信息。功能类似于request.getHeader("");

### 5.3 @CookieValue

之前获取Cookie的方式较为麻烦！

Cookie[] cookies=request.getCookies();

String value="";

for(Cookie c: cookies){

if(c.getName().equals("JESSIONID")){

value=c.getValue();

}

}

现在使用@CookieValue("xx")获取指定cookie的值

### 5.4 封装POJO

当页面传入的参数，恰好对应一个POJO的属性时，可以将这个属性封装为POJO。

做法：如果我们要提交的参数正好对应POJO的属性，只需要在处理方法的形参位置，声明一个POJO，SpringMVC会自动地帮我们封装参数

规则：参数名与 POJO的属性(setter)一一对应

级联属性的封装： 属性.属性。例如：<input name="phone.brand">

### 5.5 使用原生API

SpringMVC同时支持在处理方法的参数位置直接声明和使用原生的API。可用的原生API类型有HttpServletRequest、HttpServletResponse、HttpSession、java.security.Principal、Locale、InputStream、OutputStream

Reader、Writer。

实际上掌握前三种即可，后面的六大对象，可以通过前三个获取！

request.getLocale();

request.getUserPrincipal();

request.getInputStream();

request.getReader();

response.getWriter();

response.getOutputStream();

### 5.6 乱码问题的解决

乱码的分类：请求乱码和响应乱码。

请求乱码：

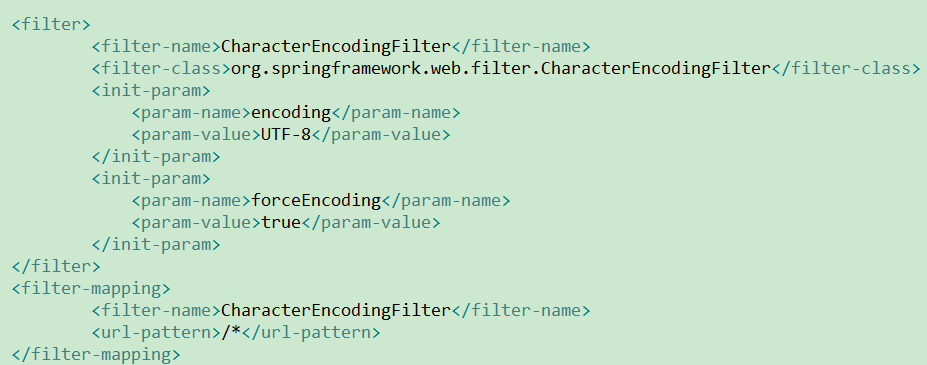
GET： 修改server.xml,在8080端口处添加属性 URIEncoding=UTF-8



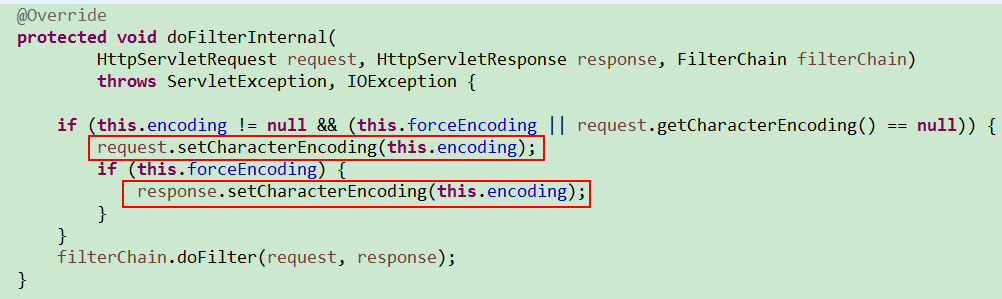
POST：（PUT，DELETE）也是POST。在第一次获取请求之前，设置request.setCharacterEncoding("UTF-8");

响应乱码：response.setContentType("text/html;charset=utf-8")

最终解决： SpringMVC提供了一个字符编码过滤器来解决；



原理：



我们在声明字符编码过滤器后，需要配置两个参数：encoding和forceEncoding。encoding代表设置请求和响应的编码。forceEncoding代表是否处理响应乱码。

注意：保证这个filter必须在其他filter之前配置在有效！

## 6. 输出数据

### 6.1 web.xml标配

<DispatcherServlet> 前端控制器

< CharacterEncodingFilter> 字符编码过滤器，要配在首位

< HiddenHttpMethodFilter> 支持RESTFUL请求

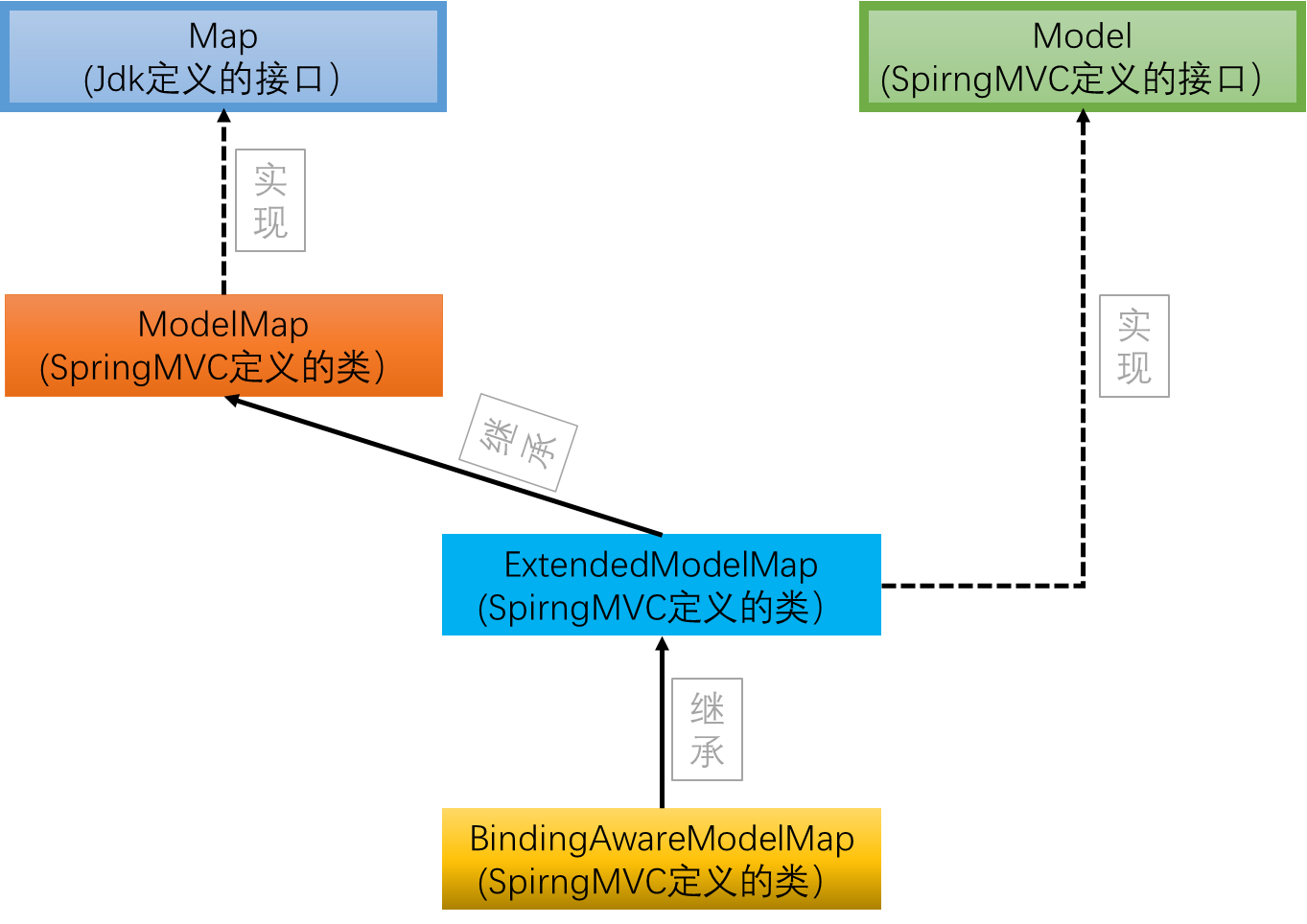
### 6.2 Map、ModelMap、Model

①SpringMVC提供了Model,Map,ModelMap来帮我们向页面传值

②这三种方式都可以向页面传值，且默认将数据，保存在请求域中！三种方式的最终实现类都是：

org.springframework.validation.support.BindingAwareModelMap

三种类型的关系：BindingAwareModelMap是Map、ModelMap、Model的实现类



### 6.3 ModelAndView

* 1. 控制器处理方法的返回值如果为 ModelAndView, 则其既包含视图信息，也包含模型数据信息。ModelAndView的数据也会放入到请求域中。
  2. 添加模型数据:

MoelAndView addObject(String attributeName, Object attributeValue)

ModelAndView addAllObject(Map<String, ?> modelMap)

* 1. 设置视图:

void setView(View view)

void setViewName(String viewName)

### 6.4 隐含模型

伴随着一次请求，不管在@ModelAttribute标注的方法中，声明的Model，或者在处理请求方法中标注的Model、Map、ModelMap，都指向一个贯穿始终的BindingAwareModelMap，这个BindingAwareModelMap的参数名在底层源码中为implicitModel，翻译过来，我们将其称为隐含模型。用来保存数据！

隐含模型中的数据，都会被放入到请求域中！

## 7. 视图

### 7.1. forward:

使用forward:可以完成一个转发操作；

格式：　forward:/; 转发的路径我们推荐使用绝对路径，即加上/

使用forward:后，转发的路径不会再经过我们配置的视图解析器来解析！

### 7.2. redirect:

使用redirect:完成重定向，SpringMVC会自动为我们拼上项目名

格式： redirect:/，重定向的路径也必须使用绝对路径，即加上/

使用redirect:后同样不会经过我们默认配置的视图解析器！

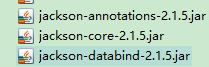
## 8. 处理Json

前台客户端，web客户端，安卓APP，IOSAPP；各种各样，但是与后台的通信方式都是Json！

### 8.1 使用@ResponseBody返回Json

将方法的返回值直接作为响应体返回！

①导包：



②后台：

@ResponseBody

@RequestMapping("/testJson")

**public** Collection<Employee> handleJson(){

Collection<Employee> all = empDao.getAll();

**return** all;

}

③前台提交ajax请求

$("#testJson").click(**function**(){

**var** url=$(**this**).attr("href");

$.ajax({

url:url,

type:"GET",

success:**function**(data){

console.log(data);

}

});

//取消跳转

**return** **false**;

});

### 8.2 使用@RequestBody将Json转换为POJO

①后台

@RequestMapping(value="/testJson4")

**public** String handleJson3(@RequestBody()Employee employee){

System.***out***.println(employee);

**return** "success";

}

②前台

$("#testSendJson").click(**function**(){

**var** url=$(**this**).attr("href");

//准备一个Json对象

**var** emp={lastName:"Jack",email:"JackMa@Alibaba.com",gender:1};

alert(**typeof** emp);

**var** empStr=JSON.stringify(emp);

alert(**typeof** empStr)

//data数据必须是一个JSON字符串

$.ajax({

url:url,

type:"POST",

data:empStr,

contentType:"application/json",

success:**function**(){

alert("success");

}

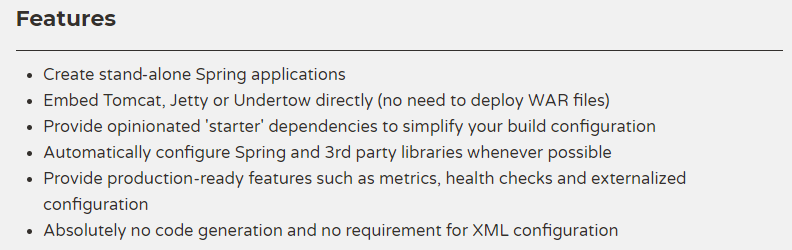
});

**return** **false**;

});

# 四、SpringBoot

## 1. SpringBoot简介



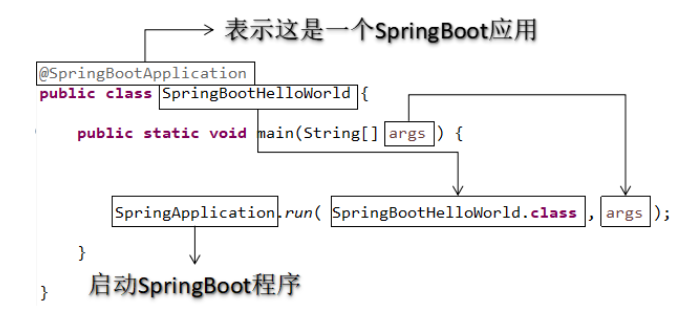
SpringBoot的作用是为了在使用Spring的应用进行开发时，简化配置。并且以最简单的方式来整合第三方的应用，例如Redis，Mybatis，SpringBoot等。

## 2. SpringBoot入门

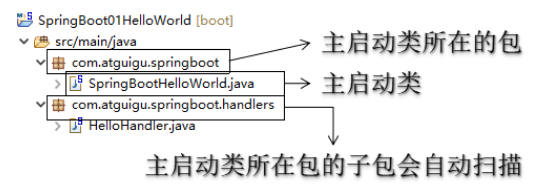
①创建Maven工程，并加入依赖

|  |
| --- |
| <!-- 继承SpringBoot官方指定的父工程 -->  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  </parent>  ​  <dependencies>  <!-- 加入Web开发所需要的场景启动器 -->  <dependency>  <!-- 指定groupId和artifactId即可，版本已在父工程中定义 -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

②创建SpringBoot项目的主启动类



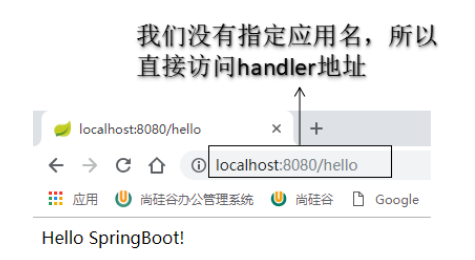
③创建Handler处理器



|  |
| --- |
| @Controller  public class HelloHandler {  @ResponseBody  @RequestMapping("/hello")  public String hello() {  return "Hello SpringBoot!";  }  } |

④运行主启动类中的main方法启动SpringBoot程序。

⑤通过浏览器访问应用



## 3. SpringBoot下的配置文件

SpringBoot环境下常用的配置文件有两种，一种是properties属性文件，一种是yml文件。二者各有特点，语法也有很大区别，但是最终效果基本一致。

### 3.1 使用Properties文件进行配置

文件名：application.properties

语法格式：属性名=属性值

### 3.2 yml语法配置

yml是YAML（YAML Ain't Markup Language）语言的文件，以数据为中心，比json、xml等更适合做配置文件。

yml语法：

* 使用缩进表示层级关系
* 缩进时不允许使用Tab键，只允许使用空格。
* 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
* 大小写敏感

YAML 支持的三种数据结构

* 对象：键值对的集合
* 数组：一组按次序排列的值
* 字面量：单个的、不可再分的值

示例：

spring:

application:

name: apple

server:

port: 8181

context-path: /banana

## 4. 整合Mybatis

①加入Mybatis的依赖

|  |
| --- |
| <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.5</version>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.1.1</version>  </dependency> |

②在application.yml中增加数据库环境配置信息

|  |
| --- |
| spring:  datasource:  name: mydb  type: com.alibaba.druid.pool.DruidDataSource  url: jdbc:mysql://127.0.0.1:3306/sb\_db  username: root  password: root  driver-class-name: com.mysql.jdbc.Driver  mybatis:  mapper-locations: classpath\*:/mybatis/\*Mapper.xml |

③在主启动类上，使用注解指定要扫描的包，会为此报下的接口创建代理对象

@MapperScan("com.dorr.springboot.mappers")