课程目标

- [ ] 了解Spring结构体系
- [ ] 熟练掌握Spring开发环境搭建
- [ ] 理解**控制反转**和**依赖注入**的概念
- [ ] 熟练掌握常见的属性注入方式
- [ ] 熟练掌握Spring XML开发方式
- [ ] 熟练掌握Spring 注解开发方式

1.Spring 概述

- Spring 是最受欢迎的**JavaEE应用程序开发框架**.
- Spring 框架是一个基于Java的**开源**项目，由Rod Johnson开发.

- Spring 框架的**目标是简化JAVAEE应用程序开发**. EJB

1.1 框架优势

- Spring 框架提供一个设 `计良好的MVC框架` ,让JavaWEB开发变得更加容易(SpringMVC)。
- Spring 框架提供优秀的 `数据访问集成方案` ,让数据访问变得轻松容易。
- Spring 框架提供了 `完整的事务解决方案` 。
- Spring 框架让软件测试变得更容易。
- ……

1.2 控制反转(IOC)

`IOC` 是 `Inverse of Control` 的缩写,意思是控制反转. 是降低对象之间耦合关系的设计思想.

通过 `IOC` ,开发人员不需要关心对象的创建过程,该过程交给**Spring IOC容器**完成.**Spring IOC容器**通过**反射**机制创建类的实例.

> 将对象的创建交给Spring容器管理,转移控制权

1.3 依赖注入(DI)

`DI` 是 `Dependency Injection` 的缩写,意思是 `Spring IOC 容器` 创建对象的时候,同时为这个对象**注入**它所**依赖**的属性的值. 这个对象与对象之间的关系也交给了 `Spring IOC 容器` 来维护.

> 将对象与对象之间的关系交给Spring IOC容器维护,转移控制权

总结:

依赖注入是 Spring IOC 容器在运行期间, 动态的将依赖关系注入到对象中.

所以依赖注入是控制反转的实现方式.

依赖注入和控制反转描述是从不同角度描述的同一件事情, 就是通过引入Spring IOC容器,利用依赖关系注入的方式, **实现对象与对象之间的解耦.**(高内聚,低耦合)

1.3 面向切面编程(AOP)

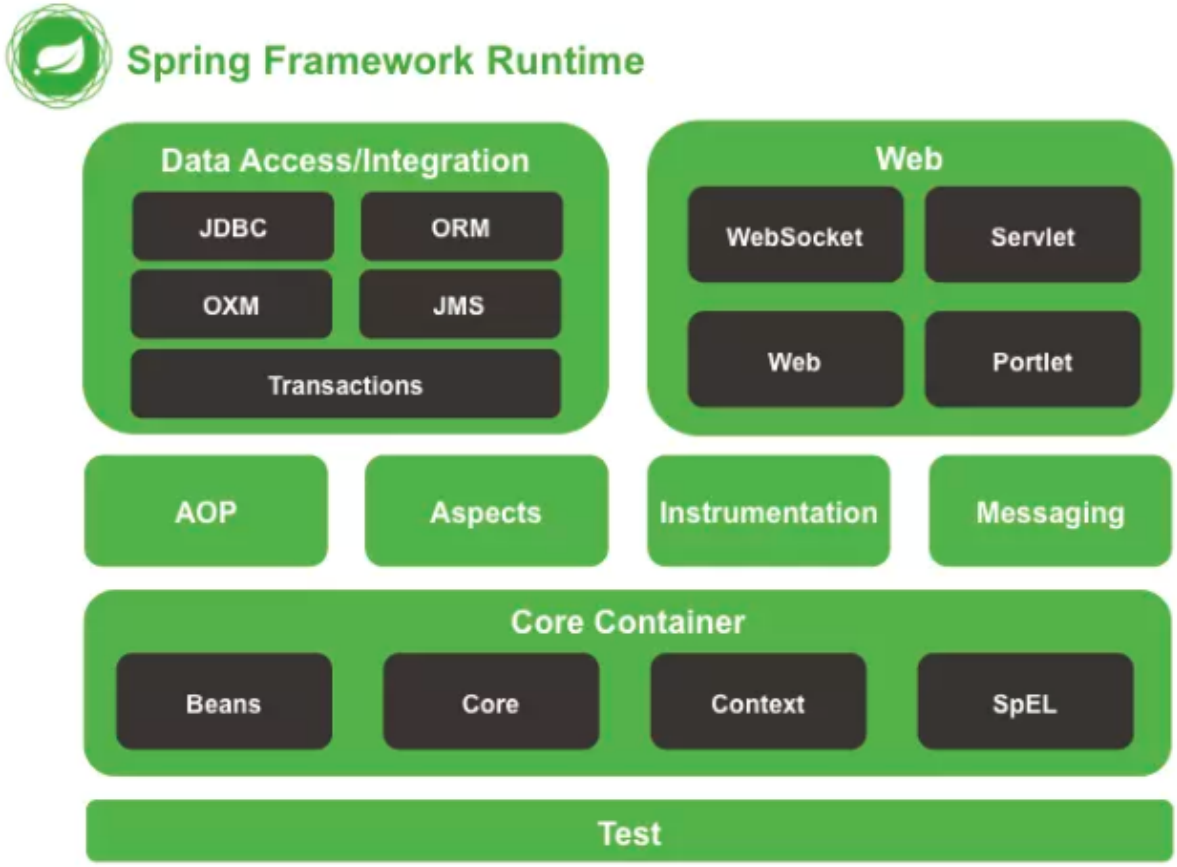
Spring 框架的一个关键组件是面向切面的程序设计（AOP）框架。

一个程序中跨越多个点的功能被称为横切关注点，这些横切关注点在概念上独立于应用程序的业务逻辑。有各种各样常见的很好的关于方面的例子，比如**日志记录、声明性事务、安全性，和缓存**等等。(Filter)

Spring 框架的 AOP 模块提供了面向切面的程序设计实现，允许你定义拦截器方法和切入点，**可以实现将应该被分开的代码干净的分开的功能。**

2.Spring 体系结构

Spring 框架提供约 20 个模块，可以根据应用程序的要求来使用。



2.1 核心容器

由 `spring-beans、spring-core、spring-context 和 spring-expression` 4 个模块组成。

- `spring-beans` 和 `spring-core` 模块是 Spring 框架的核心模块

  包含了 `控制反转` （Inversion of Control, IOC） 和 `依赖注入` （Dependency Injection, DI）

  `BeanFactory` 接口是 Spring 框架中的核心接口， 它是工厂模式的具体实现。

  `BeanFactory` 使用**控制反转**对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。

- `spring-context` 模块构架于核心模块之上， 它扩展了 BeanFactory， 为它添加了 Bean 生命周期控制、 框架事件体系以及资源加载透明化等功能·。 此外该模块还提供了许多企业级支持， 如邮件访问、远程访问、 任务调度等。

  `ApplicationContext` 是该模块的核心接口，它是 BeanFactory 的超类， 与BeanFactory 不同，`ApplicationContext` 容器实例化后会自动对所有的单实例 Bean 进行实例化与依赖关系的装配， 使之处于待用状态。

  `ApplicationContext` 接口的常用子类:

  `FileSystemXmlApplicationContext` 从磁盘绝对路径加载配置文件初始化IOC容器

  `ClassPathXmlApplicationContext` 从类路径加载配置文件初始化IOC容器

- `spring-expression 模块是统一表达式语言（EL） 的扩展模块， 可以查询、 管理运行中的对象，同时也方便的可以调用对象方法、 操作数组、 集合等` 。 它的语法类似于传统 EL， 但提供了额外的功能，最出色的要数函数调用和简单字符串的模板函数。

> ApplicationContext 重点掌握

2.2 数据访问及集成

由 `spring-jdbc、 spring-tx、 spring-orm、 spring-jms 和 spring-oxm` 5 个模块组成。

- `spring-jdbc` 模块是 Spring 提供的 JDBC 抽象框架的主要实现模块， 用于简化 Spring JDBC。

  主要是提供 JDBC 模板方式、 关系数据库对象化方式、 SimpleJdbc 方式、 事务管理来简化 JDBC 编程，

  主要实现类是 `JdbcTemplate` 、 `SimpleJdbcTemplate` 以及 `NamedParameterJdbcTemplate` 。
- `spring-tx` 模块是 Spring JDBC 事务控制实现模块。

  使用 Spring 框架， 它**对事务做了很好的封装**
- `spring-orm` 模块是 ORM 框架支持模块

  主要集成 Hibernate, Java Persistence API (JPA) 和Java Data Objects (JDO) 用于资源管理、 数据访问对象(DAO)的实现和事务策略。
- `spring-jms` 模块（Java Messaging Service） 能够发送和接受信息， 自 Spring Framework 4.1以后， 他还提供了对 spring-messaging 模块的支撑。
- `spring-oxm` 模块主要提供一个抽象层以支撑 OXM（OXM 是 Object-to-XML-Mapping 的缩写， 它是一个 O/M-mapper， 将 java 对象映射成 XML 数据， 或者将 XML 数据映射成 java 对象）

2.3 WEB

由 `spring-web、 spring-webmvc、 spring-websocket 和 spring-webflux` 4 个模块组成。

- `spring-web` 模块为 Spring 提供了最基础 Web 支持， 主要建立于核心容器之上，

  通过 Servlet 或者 Listener 来初始化 IOC 容器， 也包含一些与 Web 相关的支持。
- `spring-webmvc` 模块众所周知是一个的Web-Servlet模块，

  实现了Spring MVC（Model-View-Controller）的Web应用。
- `spring-websocket` 模块主要是与 Web 前端的全双工通讯的协议。
- `spring-webflux` 是一个新的非堵塞函数式 Reactive Web 框架

  可以用来建立异步的， 非阻塞，事件驱动的服务， 并且扩展性非常好。

2.4 AOP

由 `spring-aop、 spring-aspects 和 spring-instrument` 3 个模块组成。

- `spring-aop` 是 Spring 的另一个核心模块， 是 AOP 主要的实现模块。 作为继 OOP 后， 对程序员影响最大的编程思想之一， AOP 极大地开拓了人们对于编程的思路。 在 Spring 中， 他是 `以 JVM 的动态代理技术` 为基础， 然后 `设计出了一系列的 AOP 横切实现` ， 比如 `前置通知、 返回通知、 异常通知` 等， 同时， Pointcut 接口来匹配切入点， 可以使用现有的切入点来设计横切面， 也可以扩展相关方法根据需求进行切入。
- `spring-aspects` 模块集成自 `AspectJ` 框架， 主要是为 Spring AOP 提供多种 AOP 实现方法。
- `spring-instrument` 模块是基于 JAVA SE 中的"java.lang.instrument"进行设计的， 应该算是AOP的一个支援模块， 主要作用是在 JVM 启用时， 生成一个代理类， 程序员通过代理类在运行时修改

类的字节， 从而改变一个类的功能， 实现 AOP 的功能。 在分类里， 我把他分在了 AOP 模块下，在 Spring 官方文档里对这个地方也有点含糊不清， 这里是纯个人观点。

2.4 Test

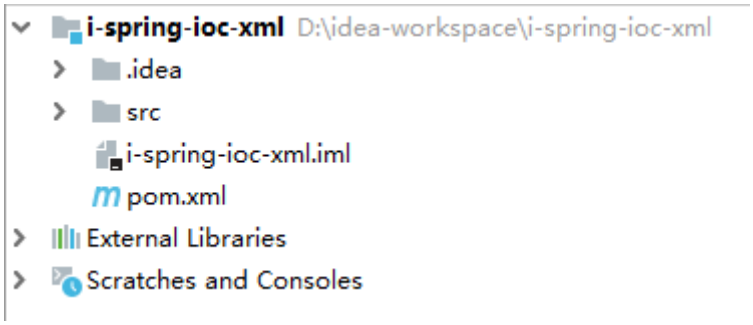
即 spring-test 模块。

`spring-test` 模块主要为测试提供支持的， 毕竟在不需要发布（程序） 到你的应用服务器或者连接到其他企业设施的情况下能够执行一些集成测试或者其他测试对于任何企业都是非常重要的。

.....

3.Hello Spring

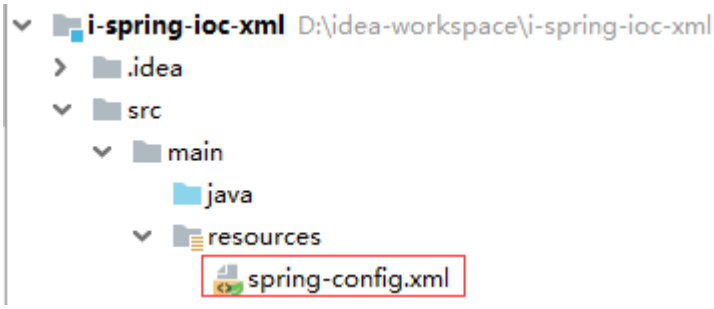
3.1 创建maven工程



3.2 项目依赖

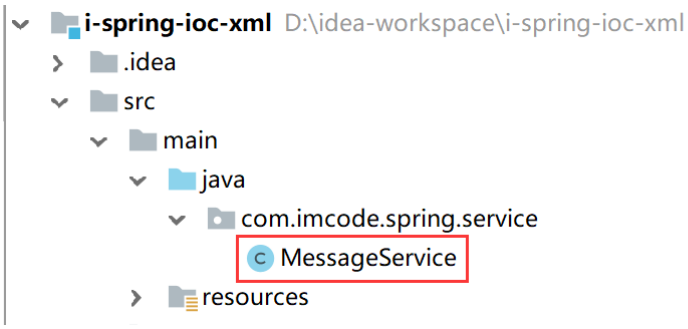
```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.10.RELEASE</version>
    </dependency>
</dependencies>
```

3.3 配置文件




```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>
```

3.4 编写JavaBean




```java
public class MessageService {
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

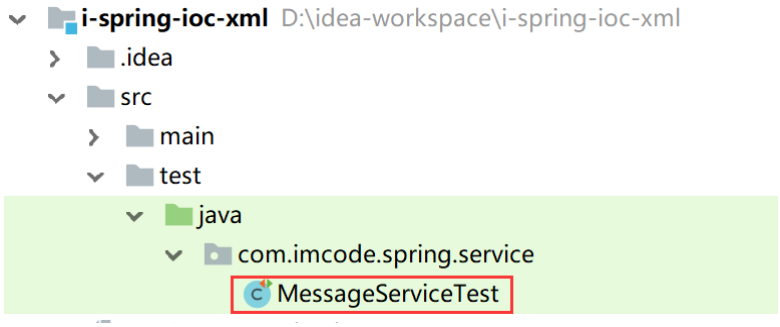
3.5 装配JavaBean

在spring-config.xml 添加如下内容:

```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">
    <!-- 配置baen
         <bean> 配置需要让Spring IOC容器创建的对象
         name : 用于之后从spring容器获得实例时使用
         class : 需要创建实例的全限定类名
         property:类中的成员
             property name  成员的名称
```

```
            property value 成员的值
        scope:
            singleton 单例
            prototype 多例
        lazy-init: true 按需创建对象 false 容器启动就创建对象 scope="singleton"时有效
    -->
    <bean id="messageService" class="com.imcode.spring.service.MessageService">
        <property name="message" value="Hello Spring"/>
    </bean>
</beans>
```

3.6 单元测试



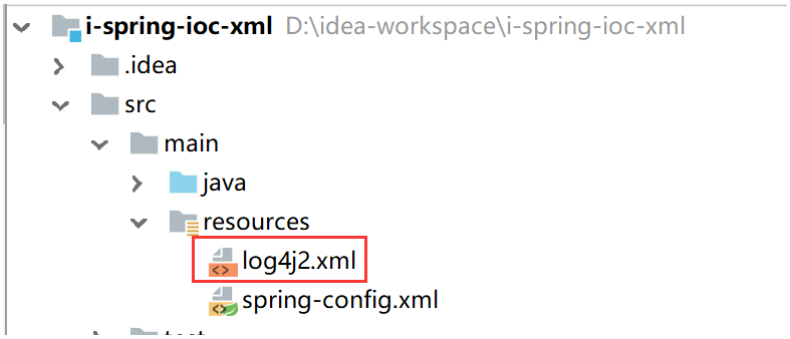

```
@Test
public void getMessage() {
    //获取应用上下文对象(Spring容器)
    ApplicationContext context =
        new ClassPathXmlApplicationContext("spring-config.xml");

    //从容器中获取java对象
    MessageService service =
        context.getBean("messageService",MessageService.class);

    String message = service.getMessage();
    System.out.println(message);
}
```

3.7 日志配置

依赖类库

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.12.1</version>
</dependency>
```

配置文件




```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--先定义所有的appender-->
    <appenders>
        <!--输出控制台的配置-->
        <console name="Console" target="SYSTEM_OUT">
            <!--输出日志的格式-->
            <patternlayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%p] %c %m %n"/>
        </console>
    </appenders>

    <!-- 然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <!-- 日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->
    <loggers>
        <!--org.springframework
        <logger name="org.springframework" level="INFO"/>-->
        <root level="INFO">
            <!--输出到控制台-->
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>
```

4.IOC 控制反转

4.1 IOC 容器

- Spring IOC 容器是 Spring 框架的核心。**容器将创建对象**，把它们连接在一起，配置它们，并管理他们的整个生命周期从创建到销毁。
- Spring IOC 容器使用**依赖注入（DI）**来管理组成一个应用程序的组件。这些对象被称为 `Spring Bean`
- 通过配置，容器知道对哪些对象进行实例化，配置和组装。
- 配置信息可以通过 **XML，Java注解或 Java 代码**来表示。Hello Spring案例使用了xml配置。

Spring 提供了以下两种不同类型的容器：

- **BeanFactory 容器**

  它是最简单的容器，给 **依赖注入(DI)** 提供了基本的支持，

  通过 `org.springframework.beans.factory.BeanFactory` 接口定义。

- **ApplicationContext 容器**

该容器添加了更多的企业特定的功能，例如从一个属性文件中解析文本信息的能力，发布应用程序事件给感兴趣的事件监听器的能力。

该容器是由 `org.springframework.context.ApplicationContext` 接口定义。

> ApplicationContext 容器包括 BeanFactory 容器的所有功能

4.2 对象的生命周期

Bean 是一个被实例化，组装，并通过 Spring IOC 容器所管理的**JAVA对象。**

这些 bean 是由给容器提供的配置信息创建的。

容器需要知道bean的如下信息：

对象实例化

```
1    <bean id="messageService" class="com.imcode.spring.MessageService"/>
```

该配置告诉容器使用 `MessageService` 类的无参构造方法实例化Bean

- `lazy-init="false"` 默认值,启动容器的时候就创建对象.
- `lazy-init="true"` ,使用对象的时候才创建对象.

该属性只对单例 `scope="singleton"` 有效.

```
1    <bean id="messageService" class="com.imcode.spring.service.MessageService"
2            scope="singleton" lazy-init="true">
```

对象的作用域

通过scope属性设置bean的作用域

```
1    <bean id="messageService" class="com.imcode.spring.service.MessageService"
     scope="singleton">
```

作用域	描述
singleton	该作用域将 bean 的定义的限制在每一个 Spring IoC 容器中的一个单一实例(默认)。
prototype	该作用域将单一 bean 的定义限制在任意数量的对象实例(多例)。
request	该作用域将 bean 的定义限制为 HTTP 请求。只在 WebApplicationContext 的上下文中有效。
session	该作用域将 bean 的定义限制为 HTTP 会话。 只在 WebApplicationContext 的上下文中有效。
global-session	该作用域将 bean 的定义限制为全局 HTTP 会话。只在 WebApplicationContext 的上下文中有效。

对象的初始化和销毁

在 `MessageService` 类中增加如下方法:

```
public void init() {
    System.out.println("初始化对象...");
}
public void destroy() {
    System.out.println("销毁对象...");
}
```

在spring-config.xml添加如下配置:

```
<bean name="messageService"
          class="com.imcode.spring.service.MessageService"
          scope="singleton"
          lazy-init="true"
          init-method="init"
          destroy-method="destroy">
        <property name="message" value="Hello Spring"/>
</bean>
```

对象的依赖关系

使用依赖注入来维护对象与对象之间的依赖关系

5.DI 依赖注入

5.1 `setXxx()` 方法注入

```
<bean id="messageService" class="com.imcode.spring.service.MessageService">
    <property name="message" value="Hello Spring"/>
</bean>
```

5.2 构造方法注入

```
public class MessageService  {

    private String message;

    public MessageService(String message) {
        this.message = message;
    }
```

方式一

使用构造方法中参数的索引(常用)

```
<bean id="messageService" class="com.imcode.spring.service.MessageService">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

方式二

使用构造方法中参数的名称(常用)

```
<bean id="messageService" class="com.imcode.spring.service.MessageService">
    <constructor-arg name="message" value="Hello Spring"/>
</bean>
```

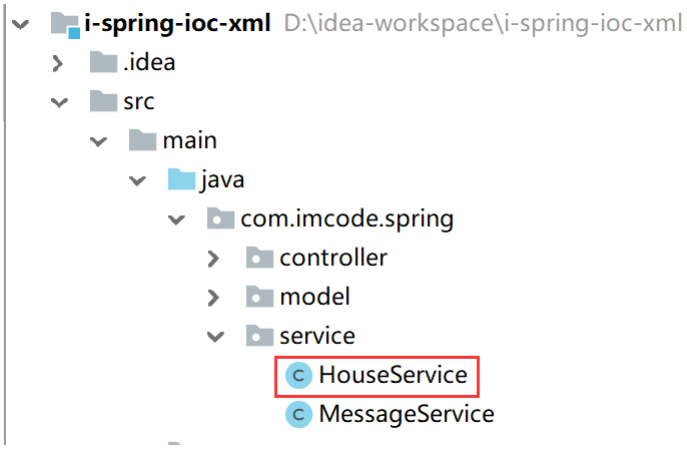
方式三

使用构造方法中参数的类型(不常用)

```
<bean id="messageService" class="com.imcode.spring.service.MessageService">
    <constructor-arg type="java.lang.String" value="Hello Spring"/>
</bean>
```

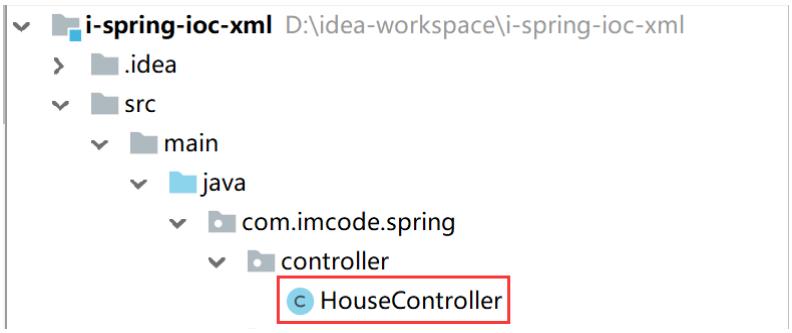
5.3 注入对象

HouseService




```
public class HouseService {
    public void deleteById(Integer id){
        System.out.println("删除成功 HouseId = " + id);
    }
}
```

HouseController




```
public class HouseController {
    private HouseService houseService;
    /**
```

```
     * 构造方法
     * @param HouseService
     */
    public HouseController(HouseService houseService) {
        this.houseService = houseService;
    }

    /**
     * set 方法
     * @param HouseService
     */
    public void setHouseService(HouseService houseService) {
        this.HouseService = houseService;
    }

    /**
     * servlet 伪代码
     * 模拟响应客户端请求
     */
    public void doGet() {
        Integer id = 100;// 模拟从request 获取到的id
        houseService.deleteById(id);
    }
}
```

构造方法注入

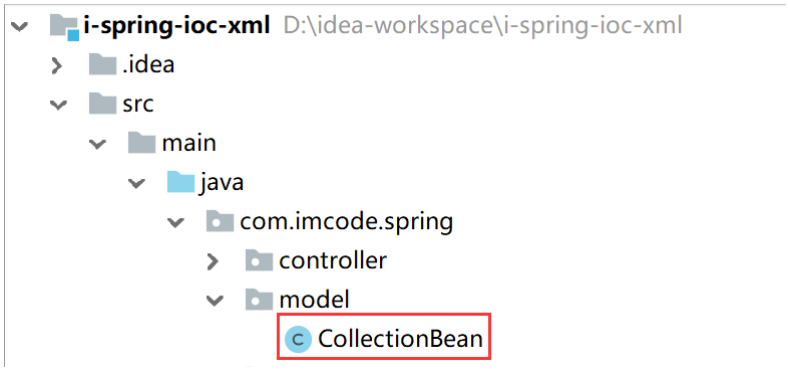
```
<bean id="houseService" class="com.imcode.spring.service.HouseService"/>
<bean id="houseController" class="com.imcode.spring.controller.HouseController">
    <!--<constructor-arg index="0" ref="houseService"/>-->
    <!--<constructor-arg name="houseService" ref="houseService"/>-->
    <constructor-arg type="com.imcode.spring.service.HouseService"
ref="houseService"/>
</bean>
```

setXxx() 方法注入

```
<bean id="houseService" class="com.imcode.spring.service.HouseService"/>
<bean id="houseController" class="com.imcode.spring.controller.HouseController">
    <property name="houseService" ref="houseService"/>
</bean>
```

5.4 注入集合

编写类




```java
public class CollectionBean {
    private Object[] arrayData = new Object[5];
    private List<Object> listData;
    private Set<Object> setData;
    private Map<String, Object> mapData;
    private Properties propsData;
}
```

配置文件

```xml
<bean name="collectionBean" class="com.imcode.spring.model.CollectionBean">
    <property name="arrayData">
        <array>
            <value>100</value>
            <value>Hello Spring</value>
            <value>true</value>
            <value>20.10</value>
            <ref bean="houseService"></ref>
        </array>
    </property>
    <property name="listData">
        <list>
            <value>100</value>
            <value>Hello Spring</value>
            <value>true</value>
            <value>20.10</value>
            <ref bean="houseService"></ref>
        </list>
    </property>
    <property name="setData">
        <set>
            <value>100</value>
            <value>Hello Spring</value>
            <list>
                <value>A</value>
                <value>B</value>
                <value>C</value>
            </list>
            <ref bean="houseService"></ref>
        </set>
    </property>
    <property name="mapData">
        <map>
            <entry key="k1" value="100"/>
            <entry key="k2" value-ref="houseService"/>
        </map>
```

```
        </property>

        <property name="propsData">
            <props>
                <prop key="k1">100</prop>
                <prop key="k2">Spring</prop>
            </props>
        </property>
    </bean>
```

6.使用注解

6.1 启用注解扫描

开发中可以使用注解取代xml配置文件

注解使用前提，添加命名空间，让spring扫描含有注解类

```
<?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.xsd">
    <!-- 启动注解扫描 -->
    <context:component-scan base-package="com.imcode.spring.service"/>
</beans>
```

6.2 控制反转(IOC)

`@Component`

- `@Component` 等价于

  ```
  <bean class=""/>
  ```

- `@Component("name")` 等价于

  ```
  <bean name="" class=""/>
  ```

衍生注解

提供3个 `@Component` 注解衍生注解（功能一样）,分别用于数据访问层,服务层和控制层

`@Repository`

- 数据访问层

`@Service`

- 服务层

`@Controller`

- 控制层

6.3 依赖注入(DI)

注入简单数据

`@Value`

- 注入简单数据类型：`@Value("${}")`

注入对象

`@Autowired`

`@Qualifier("name")`

> @Autowired 如果注入的是具体类，不会有问题
>
> @Autowired 如果注入的是接口，该接口只有一个实现类，不会有问题
>
> @Autowired 如果注入的是接口，该接口有一个以上的实现类，会有问题，IOC容器不知道应该注入哪个实 现类了
>
> 使用
>
> @Autowired + @Qualifier("名称") 组合使用两个注解解决

`@Resource`

`@Resource(name = "bean的名称")`

该注解是JDK提供的注解

`@Resource("name")` == `@Autowired + @Qualifier("name")`

`@Resource == @Autowired`

6.4 作用域

注解在类上

`@Scope("prototype")`

多例

`@Scope("singleton")`

单例

6.5 初始化和销毁

`@PostConstruct`

`@PreDestroy`

```
public class MessageService {

     ......

    @PostConstruct
    public void init() {
        System.out.println("初始化对象...");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("销毁对象...");
    }
}
```

7.SpringTest

引入spring-test依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.10.RELEASE</version>
    <scope>test</scope>
</dependency>
```

测试用例

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:spring-config.xml")
public class SpringTest {
    @Autowired
    private ProductService productService;
    @Test
    public void test(){
        productService.deleteById(1L);
    }
}
```