Spring5

Spring5 课程介绍

1. IOC 容器；
2. AOP 面向切面编程；
3. 声明事务；
4. 注解的方式启动 对我们后期学习 SpringBoot 有非常大帮助；
5. 整合 SpringMVC 和 Mybatis；
6. Spring5 新特性；

JDK 最低版本要求 1.8

Spring 概念

Spring 是一个 JavaEE 开源的轻量级别的框架，可以解决我们企业开发中遇到的难题，能够让编码变的更加简单，核心组件 IOC 容器和 Aop 面向切面编程。

1. IOC 控制反转：把整个对象创建的过程，统一交给我们 SpringIOC 容器来实现管理，底层使用反射+工厂模式实现。
2. Aop 面向切面编程：对我们功能（方法）前后实现增强，比如打印日志、事务原理、权限管理，底层是基于动态代理模式实现的。

减少到我们的代码的冗余性问题。

Spring 优势

1. 方法的解耦，简化开发；
2. Aop 技术的支持；
3. 提供声明事务支持
4. Junit 单元测试
5. 方便整合其他框架（Mybatis、SpringMVC、SpringBoot、SpringCloud、Redis 等）
6. 降低我们的 JavaEEapi 开发使用的难度（Spring 对很多复杂的 api 接口实现了封装）

Spring 与 SpringBoot 关系

SpringBoot 直接采用注解化的方式启动，底层会依赖于 Spring/SpringMVC 注解方式启动。

总结：SpringBoot 底层基于 Spring/SpringMVC 注解化方式实现包装。

比如：
1.@RestController
2.@ComponentScan("com.mayikt.aop")
3. @Configuration
4. @Component
5. @Scheduled
6. @Value
7. @Bean

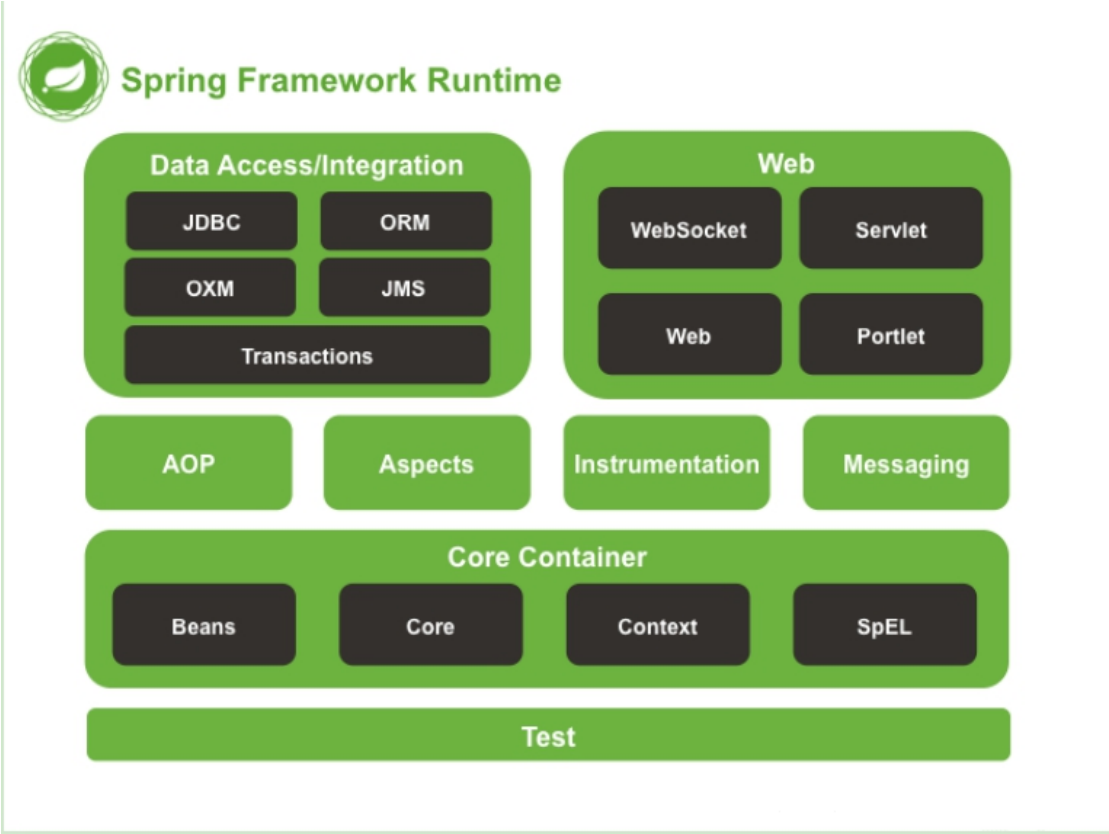
SpringIOC 底层容器原理

Spring 框架快速入门

https://spring.io/ spring 的官网
Spring 官方下载依赖 jar 包地址：
https://repo.spring.io/webapp/#/artifacts/browse/tree/General/libs-release-local/org/springframework/spring

本次课以 idea 构建 maven 项目方式讲解；
javadoc Api 文档的介绍
Sources jar 的源代码 .java
直接命名为.jar 包的格式 就是 class 文件。

七大核心模块



Test

对应 spring-test.jar. Spring 提供的测试工具，可以整合 JUnit 测试，简化测试环节.

Core Container

Spring 的核心组件，包含了 Spring 框架最基本的支撑.

Beans，对应 spring-beans.jar. Spring 进行对象管理时依赖的 jar 包.
Core，对应 spring-core.jar, Spring 核心 jar 包.
Context，对应 spring-context.jar, Spring 容器上下文对象.
SpEL，对应 spring-expression.jar, Spring 表达式语言.

AOP

面向切面编程，对应 spring-aop.jar.

Data Access

Spring 对数据访问层的封装

JDBC，对应 spring-jdbc.jar. Spring 对 jdbc 的封装，当需要使用 spring 连接数据库时使用. spring-jdbc.jar 需要依赖 spring-tx.jar.
Transactions，对应 spring-tx.jar. 事务管理

ORM，对应 spring-orm.jar. spring 整合第三方 orm 框架需要使用的 jar 包，例如 Hibernate 框架.
Web
Spring 对 javax 下的接口或类做的扩展功能.

spring-web.jar，对 Servlet, filter, Listener 等做的增强.
spring-webmvc.jar，实际上就是 SpringMVC 框架. 需要依赖 spring 环境和 spring-web.jar.

Spring Core

核心容器提供 Spring 框架的基本功能。Spring 以 bean 的方式组织和管理 Java 应用中的各个组件及其关系。Spring 使用 BeanFactory 来产生和管理 Bean，它是工厂模式的实现。BeanFactory 使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开。
Maven 依赖：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```


Spring-Beans

这个 jar 文件是所有应用都要用到的，它包含访问配置文件、创建和管理 bean 以及进行 Inversion ofControl / Dependency Injection（IoC/DI）操作相关的所有类。如果应用只需基本的 IoC/DI 支持，引入 spring-core.jar 及 spring-beans.jar 文件就可以了。
外部依赖 spring-core，(CGLIB)。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

Spring Context

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

Spring-Expression

模块提供了一个强大的表达式语言，用于在运行时查询和处理对象图。该语言支持设置和获取属性值；属性赋值，方法调用，访问数组的内容，收集和索引器，逻辑和算术运算，命名变量，并从 Spring 的 IOC 容器的名字对象检索，它也支持列表选择和投影以及常见的列表聚合。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-expression</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```


Spring AOP

通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

JDBC 和 DAO 模块（Spring DAO）

JDBC、DAO 的抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理，和不同数据库供应商所抛出的错误信息。异常层次结构简化了错误处理，并且极大的降低了需要编写的代码数量，比如打开和关闭链接。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

spring-transaction

以前是在这里 org.springframework.transaction
为 JDBC、Hibernate、JDO、JPA、Beans 等提供的一致的声明式和编程式事务管理支持。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```


Spring ORM

Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 对象的关系工具，其中包括了 Hibernate、JDO 和 IBatis SQL Map 等，所有这些都遵从 Spring 的通用事物和 DAO 异常层次结构。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

Spring Web MVC

MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的。MVC 容纳了大量视图技术，其中包括 JSP、POI 等，模型来有 JavaBean

来构成，存放于 m 当中，而视图是一个街口，负责实现模型，控制器表示逻辑代码，由 c 的事情。Spring 框架的功能可以用在任何 J2EE 服务器当中，大多数功能也适用于不受管理的环境。Spring 的核心要点就是支持不绑定到特定 J2EE 服务的可重用业务和数据的访问的对象，毫无疑问这样的对象可以在不同的 J2EE 环境，独立应用程序和测试环境之间重用。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

项目构建

Maven 依赖

```
    <dependencies>
        <!--
         这个jar 文件包含Spring 框架基本的核心工具类。Spring 其它组件要都要使用到这个包里的类，是其它组件的基本核心，当然你也可以在自己的应用系统中使用这些工具类。
外部依赖Commons Logging， (Log4J)。
         -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>
        <!--
        这个jar 文件是所有应用都要用到的，它包含访问配置文件、创建和管理bean 以及进行Inversion ofControl / Dependency Injection（IoC/DI）操作相关的所有类。如果应用只需基本的IoC/DI 支持，引入spring-core.jar 及spring-beans.jar 文件就可以了。
外部依赖spring-core，(CGLIB)。
        -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>
        <!--
这个jar 文件为Spring 核心提供了大量扩展。可以找到使用Spring ApplicationContext 特性时所需的全部类，JDNI 所需的全部类，instrumentation 组件以及校验Validation 方面的相关类。
外部依赖spring-beans，(spring-aop)。
```

```
        -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.1.RELEASE</version>
        </dependency>


    </dependencies>
```

创建 spring.xml 文件

```
<?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 SpringBean 对象
     -->
    <bean id="userEntity" class="com.mayikt.entity.UserEntity"></bean>

</beans>
```

获取 Bean 对象

```
//        new UserEntity()
        // 1.读取 xml 配置文件
        ClassPathXmlApplicationContext
classPathXmlApplicationContext = new
ClassPathXmlApplicationContext("spring.xml");
        // 2.根据 bean 的 id 获取 bean 对象
        UserEntity userEntity =
classPathXmlApplicationContext.getBean("userEntity",
UserEntity.class);
        System.out.println(userEntity);
        userEntity.addUser();
```

SpringIOC

IOC 容器底层实现原理；
1.IOC 容器中非常核心的接口 BeanFactory
BeanFactory
Bean 对象 Factory 工厂
2.IOC 容器基本的概念：控制反转
把对象的创建过程与使用统一都交给我们的 Spring 来进行原理。
不需要开发者自己去 new 对象
3. IOC 容器底层实现技术：反射技术、解析 xml、工厂模式
4. IOC 作用 降低我们代码的耦合度。

创建对象的方式有那些：
1. 单独 new 方式---耦合度太高了
每次单独 new 对象，没有实现统一管理对象，如果后期 userDao 的名称信息发生变化的情况下，需要改变的引用地方比较多，耦合度太高。
2. 工厂模式---降低我们耦合度
概念：统一的管理和维护我们每个对象创建与使用的过程。

不需要自己 new 对象。

3. 反射的方式
降低代码的-耦合度

Com.mayikt.dao---数据库访问层；
Com.mayikt.service---业务逻辑层；
业务逻辑层调用到数据库访问层

反射创建对象

SpringIOC 容器底层实现原理：
反射+工厂模式+解析 xml 技术实现
1.使用解析 xml 技术 解析 spring.xml 配置文件；
2.获取<bean id="" class=""/> 类的完整路径地址
3.使用到反射技术初始化对象
4.需要使用工厂模式封装初始化对象

IOC 核心的接口

1. IOC 的核心思想底层基于反射+工厂模式实现
2. Spring 提供 IOC 容器实现两种方式：

2.1 BeanFactory IOC 容器基本的实现，是 spring 内部自己使用的接口，不提供给开发者使用。加载配置文件过程的时候，不会创建对象，当我们在获取对象的时候才会获取创建对象。

2.2 ApplicationContext BeanFactory 接口的子接口，提供更多的强大功能，适合于开发者使用。

当我们在加载配置文件的过程中，就会将配置文件中的对象创建。

在做服务器端开发的时候，使用 ApplicationContext 比较多，因为所有 bean 初始化操作在项目启动完成之前都已经初始化了。

ApplicationContext 主要实现类

ClassPathXmlApplicationContext：对应类路径下的 XML 格式的配置文件
FileSystemXmlApplicationContext：对应文件系统中的 XML 格式的配置文件
ConfigurableApplicationContext 是 ApplicationContext 的子接口，包含一些扩展方法 refresh()和 close()让 ApplicationContext 具有启动、关闭和刷新上下文的能力。所以要关闭 ApplicationContext 需要 new 此接口的对象调用 close()方法
WebApplicationContext 专门为 WEB 应用而准备的，它允许从相对于 WEB 根目录的路径中完成初始化工作

SpringBean 的注入方式

创建对象和 set 方法注入属性

1. 什么是 Bean 管理
使用 spring 创建对象
使用 spring 注入属性
2. Bean 的管理有两种方式
1. 基于 XML 方式配置
基于 XML 方式创建对象

```
<bean id="userEntity"
class="com.mayikt.entity.UserEntity"></bean>
```

在 spring 的配置文件中，会配置一个 bean 标签，注入 bean 的信息 创建 bean 对象
Id：获取 bean 对象 唯一 bean 对象的名称； bean 的名称不允许重复
Class 属性： 类的完整路径地址（类名称+包名称）
默认底层使用反射技术执行无参数构造函数

2. 基于 xml 方式注入属性

DI 依赖注入： 对象的属性注入值； （spring 实现）

1. 第一种实现方式：基于对象属性 set 方法实现

```
<bean id="bookEntity" class="com.mayikt.entity.BookEntity">
    <property name="bookName" value="蚂蚁课堂面试宝典
"></property>
    <property name="bookPrice" value="108.00"></property>
</bean>
```

在 Bean 标签下 在定义一个属性<property>标签

Name：类中的属性名称

Value：需要注入属性值

有参构造函数注入属性

实例类

```
public class OrderEntity {
    private String orderId;
    private String orderName;

    public OrderEntity(String orderId, String orderName) {
        this.orderId = orderId;
        this.orderName = orderName;
    }

    @Override
    public String toString() {
        return "OrderEntity{" +
                "orderId='" + orderId + '\'' +
                ", orderName='" + orderName + '\'' +
                '}';
    }
}
```

Xml 配置文件

```
<bean id="orderEntity"
class="com.mayikt.entity.OrderEntity">
 <!--
        <constructor-arg name="orderId"
value="123456"></constructor-arg>

    <constructor-arg name="orderName" value="蚂蚁课堂第八期

订单"></constructor-arg>

  -->
    <constructor-arg index="0"
```

```
value="123456"></constructor-arg>
    <constructor-arg index="1" value="蚂蚁课堂第八期订单
"></constructor-arg>
</bean>
```

<constructor-arg name 指定参数列表名称

<constructor-arg index 指定参数列表索引

p 名称空间注入

1. Xml 头部引入 P 标签：

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"></bean>
```

2. 使用 p 标签注入属性：

```
<bean id="bookEntity" class="com.mayikt.entity.BookEntity"
p:bookName="mayikt" p:bookPrice="66">
</bean>
```

使用 p 标签为属性注入值：调用 set 方法注入值

注入空值和特殊符号

注入空值属性

```
<bean id="bookEntity2"
class="com.mayikt.entity.BookEntity">
    <property name="bookName" value="mayikt">
    </property>
    <property name="bookPrice" >
        <null></null>
    </property>
```

注入特殊符号

转移注入方式

<< 转移为：<<

>>转移为：>>

```
<bean id="bookEntity3" class="com.mayikt.entity.BookEntity">
    <!--    <property name="bookName" value="<<武汉>>"></property> -->
    <property name="bookName" value="&lt;&lt;武汉&gt;&gt;"></property>
    <property name="bookPrice">
        <null></null>
    </property>
</bean>
```

Cdata 注入方式

<![CDATA[<<>>]]>

```
<bean id="bookEntity4" class="com.mayikt.entity.BookEntity">
    <!--    <property name="bookName" value="<<武汉>>"></property> -->
    <property name="bookName">
        <value><![CDATA[<<武汉>>]]></value>
    </property>
    <property name="bookPrice">
        <null></null>
    </property>
</bean>
```

注入属性外部 bean

Com.mayikt.controller---控制层

Com.mayikt.service----业务逻辑层

MemberService ##new MemberDao().

Com.mayikt.dao----数据库访问层

MemberDao----

Com.mayikt.service

调用：memberService

Com.mayikt.dao

MemberDaoImpl

```
public interface MemberDao {
    void addMember();
}
public class MemberDaoImpl implements  MemberDao {
   public void addMember() {
       System.out.println("dao member");
    }
}
import com.mayikt.dao.MemberDao;
import com.mayikt.dao.MemberDaoImpl;

public class MemberService {
    private MemberDao memberDao;

    public MemberDao getMemberDao() {
        return memberDao;
    }

    public void setMemberDao(MemberDao memberDao) {
        this.memberDao = memberDao;
    }

    public void addMember() {
        System.out.println("<<<Service Service>>");
        // 原始的方式
//          MemberDao memberDao = new MemberDaoImpl();
//          memberDao.addMember();
        memberDao.addMember();
    }
}

<?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">
```

```
    <!--
     注入 useService
     -->
    <bean id="memberService" class="com.mayikt.service.MemberService">
        <!--  注入 userDao
         name 属性值：类中属性的名称；
         ref 创建 MemberDaoImpl 类的 bean 的 id
         -->
        <property name="memberDao" ref="memberDao"></property>
    </bean>
    <bean id="memberDao" class="com.mayikt.dao.MemberDaoImpl"></bean>
</beans>
```


注入内部 bean

1. 数据库表 一对多或者一对一的关系
2. 部门--n 多个员工 一对多
3. 站在员工角度考虑 员工属于那个部门
4. 站在部门的角度考虑 部门下 n 多个员工

1. 在数据库中表中有一对一 一对多的关系；
2. 一对多关系； 部门与员工 一个部门会有多个员工 一个员工属于一个部门；
3. 实体类之间表示一对多的关系；

实体类 员工对象

```
public class EmpEntity {

    private String name;
    private Integer age;
    /**
     * 员工属于那个部门
     */
    private  DeptEntity deptEntity;
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public void setDeptEntity(DeptEntity deptEntity) {
        this.deptEntity = deptEntity;
    }

    @Override
    public String toString() {
        return "EmpEntity{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", deptEntity=" + deptEntity +
                '}';
    }
}
```

部门对象

```
public class DeptEntity {
    private String name;
    public void setName(String name) {
```

```
        this.name = name;
    }
    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "DeptEntity{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

Xml 相关配置

```
<!--内部 bean -->
<bean id="empEntity" class="com.mayikt.entity.EmpEntity">
    <!--设置属性 name/age -->
    <property name="name" value="mayikt"></property>
    <property name="age" value="21"></property>
    <!-- 嵌入部门 bean-->
    <property name="deptEntity">
        <bean id="deptEntity" class="com.mayikt.entity.DeptEntity">
            <property name="name" value="教育部门"></property>
        </bean>
    </property>

</bean>
```

注入级联赋值

写法 1

```
<bean id="empEntity" class="com.mayikt.entity.EmpEntity">
    <!--两个属性 -->
```

```
    <property name="name" value="mayikt"></property>
    <property name="addres" value="湖北省武汉市"></property>
    <!--级联赋值 -->
    <property name="deptEntity" ref="deptEntity"></property>
</bean>
<bean id="deptEntity" class="com.mayikt.entity.DeptEntity">
    <property name="name" value="教育部门"></property>
</bean>
```

写法 2

```
<bean id="empEntity" class="com.mayikt.entity.EmpEntity">
    <!--两个属性 -->
    <property name="name" value="mayikt"></property>
    <property name="addres" value="湖北省武汉市"></property>
    <!--级联赋值 -->
    <property name="deptEntity" ref="deptEntity"></property>
    <property name="deptEntity.name" value="教育部门"></property>
</bean>
<bean id="deptEntity" class="com.mayikt.entity.DeptEntity">
</bean>
```


注意：需要在员工实体类新增：deptEntity get 方法。

注入集合类型属性

1. 注入数组类型
2. 注入 list 集合类型
3. 注入 Map 集合类型属性
4. 注入 set 集合属性

实体类

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class StuEntity {
```

```
    //1.数组属性
    private String[] arrays;
    //2.list 集合属性
   private List<String> list;
    //3.Map
    private Map<String,String> map;
    //4.Set
    private Set<String> set;

    public void setArrays(String[] arrays) {
        this.arrays = arrays;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setMap(Map<String, String> map) {
        this.map = map;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }

    @Override
    public String toString() {
        return "StuEntity{" +
                "arrays=" + Arrays.toString(arrays) +
                ", list=" + list +
                ", map=" + map +
                ", set=" + set +
                '}';
    }
}
```

配置文件

```
<bean id="stuEntity" class="com.mayikt.entity.StuEntity">
    <!--数组类型注入 -->
    <property name="arrays">
        <array>
```

```
            <value>mayikt01</value>
            <value>mayikt02</value>
        </array>
    </property>
    <!--list-->
    <property name="list">
        <list>
            <value>语文</value>
            <value>数学</value>
        </list>
    </property>
    <!--Map-->
    <property name="map">
       <map>
           <entry key="余胜军" value="23"></entry>
           <entry key="小薇" value="25"></entry>
       </map>
    </property>
    <!--Set-->
    <property name="Set">
       <set>
           <value>01</value>
           <value>02</value>
       </set>
    </property>
</bean>
```

集合类型为对象

```
private List<CourseEntity> courses;
public void setCourses(List<CourseEntity> courses) {
    this.courses = courses;
}

public class CourseEntity {
    private String name;

    public void setName(String name) {
        this.name = name;
    }
```

```
    @Override
    public String toString() {
        return "CourseEntity{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

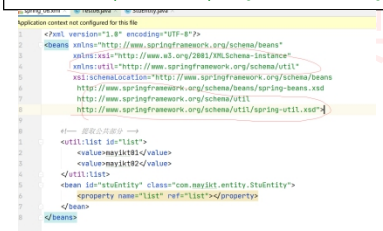
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="stuEntity" class="com.mayikt.entity.StuEntity">
        <!--对我们的 list 属性赋值 -->
        <property name="list">
            <list>
                <value>list01</value>
                <value>list02</value>
            </list>
        </property>
        <!--对我们的 arrays 属性赋值 -->
        <property name="arrays">
            <array>
                <value>mayikt01</value>
                <value>mayikt02</value>
            </array>
        </property>
        <!--对我们的 map 属性赋值 -->
        <property name="map">
            <map>
                <entry key="mayikt" value="余胜军"></entry>
                <entry key="xiaowei" value="小薇"></entry>
            </map>
        </property>
        <!--对我们的 set 属性赋值 -->
        <property name="set">
            <set>
                <value>list01</value>
                <value>list02</value>
```

```
            </set>
        </property>
        <property name="courses" >
            <list>
                <ref bean="courseEntity1"></ref>
```

集合注入部分提取公共

1. 需要先引入一个 util 名称空间

```
<?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:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">
```



2. 使用 util 标签 注入

```
<?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:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

    <!-- 提取公共部分 -->
    <util:list id="list">
        <value>mayikt01</value>
        <value>mayikt02</value>
    </util:list>
    <bean id="stuEntity" class="com.mayikt.entity.StuEntity">
```

```
        <property name="list" ref="list"></property>
    </bean>
</beans>
```

IOC 操作 Bean 的管理

1. Spring 中两种类型 bean，一种是为普通的 bean，另外一种是工厂 bean FactoryBean
2. 普通 Bean：在配置文件中定义什么类型与返回的类型需一致；
3. 工厂 Bean：在配置文件中定义 Bean 类型与返回类型可以不一致；

创建一个类，这个类是为工厂 Bean，实现 FactoryBean 接口

```
import com.mayikt.entity.UserEntity;
import org.springframework.beans.factory.FactoryBean;

public class MayiktBean implements FactoryBean<UserEntity> {
    /**
     * 定义返回 bean
     *
     * @return
     * @throws Exception
     */
    public UserEntity getObject() throws Exception {
        return new UserEntity();
    }

    public Class<?> getObjectType() {
        return null;
    }
}
```

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext app =
            new ClassPathXmlApplicationContext("spring_08.xml");
    UserEntity mayiktBean = (UserEntity) app.getBean("mayiktBean");
    System.out.println(mayiktBean);
}
```

Spring 的工厂 Bean

SpringBean 的作用域

什么是作用域？
设定 bean 作用域是为单例还是多例
作用域单例与多例有什么区别呢？
1. 单例的作用域：每次在调用 getbean 方法获取对象都是为同一个对象；
2. 多例的作用域：每次在调用 getbean 方法获取对象都是一个
新的对象。
注意：在 spring 默认的情况下，bean 的作用域就是为单例 节约服务器内存。
单例：
在同一个 jvm 中，该 bean 对象只会创建一次。
多例：
在同一个 jvm 中，该 bean 对象可以被创建多次。

设定对象单例还是多例
在 spring 的默认的情况下，springbean 的作用域为单例。
1.单例就是每次获取 bean 都是同一个对象；
2.多例就是每次获取 bean 都是新的一个对象；
单例：在同一个 jvm 中该 bean 只能存在一个实例；
多例子：在同一个 jvm 中该 bean 存在多个实例；
证明：如果是为单例，则两个对象地址都是一样的，
多例子对象则两个对象地址不一样。

单例配置：

```
<bean id="userEntity" class="com.mayikt.entity.UserEntity" scope="singleton"></bean>
```

默认就是为单例子；
多例配置：

```
<bean id="userEntity"
```

```
class="com.mayikt.entity.UserEntity
" scope="prototype"></bean>
```

SpringBean 的生命周期

简单分为：实例化→属性赋值→初始化→销毁

生命周期概念：
1. 对象的创建与销毁的过程，类似之前学习 servlet 生命的周期过程。

生命周期的原理：
1. 通过构造函数创建 bean 对象（默认执行无参构造函数 底层基于反射实现）
2. 为 bean 的属性设置 （使用反射调用 set 方法）
3. 调用 bean 的初始化的方法（需要单独在类中配置初始化的方法）
4. 正常使用 bean 对象
5. Spring 容器关闭，调用该类的销毁回调的方法（需要单独在类中配置销毁的方法）

```
public class MemberEntity {
    private String name;
    public MemberEntity(){
        System.out.println("[第一步]-无参构造函数被执行---反射机制调用");
    }

    public void setName(String name) {
```

```
        System.out.println("[第二步]-set 方法初始化属性---反射机制调用");
        this.name = name;
    }

    /**
     * 回调调用 init 初始化方法
     */
    public void initMethod(){
        System.out.println("[第三步]-回调调用 init 初始化方法");
    }

    /**
     * destroyMethod
     */
    public void destroyMethod(){
        System.out.println("[第五步]-回调调用 destroyMethod 方法");
    }
}
```

```
<bean id="memberEntity" class="com.mayikt.entity.MemberEntity" init-method="initMethod"
destroy-method="destroyMethod">
   <property name="name" value="mayikt"></property>
</bean>
<bean id="mayiktBeanPost" class="com.mayikt.bean.MayiktBeanPost"></bean>
```

```
import com.mayikt.entity.MemberEntity;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test07 {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext app =
                new ClassPathXmlApplicationContext("spring_07.xml");
       MemberEntity memberEntity= app.getBean("memberEntity",MemberEntity.class);
        System.out.println("[第四步]-获取使用到的 memberEntity");
        System.out.println(memberEntity);
        // 手动让 bean 容器销毁
        app.close();
    }
}
```

Bean 的后置处理器 作用提供更多的扩展功能 BeanPostProcessor

相关演示代码

```
```

```java
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;


public class MayiktBeanPost implements BeanPostProcessor {
    /**
     * 调用初始化方法之前执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("在 bean 初始化方法之前执行");
        return bean;
    }

    /**
     * 调用初始化方法之后执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("在 bean 初始化方法之后执行");
        return bean;
    }
}
```

```xml
<bean id="mayiktBeanPost" class="com.mayikt.bean.MayiktBeanPost"></bean>
```

1.通过构造函数创建 bean 对象（默认执行无参构造函数 底层基于反射实现）
2.为 bean 的属性设置 （使用反射调用 set 方法）
3.将 bean 传递给后置处理器 调用初始化方法之前执行
4.调用 bean 的初始化的方法（需要单独在类中配置初始化的方法）
5.将 bean 传递给后置处理器 调用初始化方法之后执行
6.正常使用 bean 对象
7.Spring 容器关闭，调用该类的销毁回调的方法（需要单独在类中配置销毁的方法）

后置处理器底层原理

配置多个 BeanPostProcessor

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.Ordered;

public class MayiktBeanPost implements BeanPostProcessor, Ordered {
    /**
     * 调用初始化方法之前执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("调用该 bean 的 init 方法之前");
        return bean;
    }

    /**
     * 调用初始化方法之后执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("调用该 bean 的 init 方法之后");
        return bean;
    }

    public int getOrder() {
        return 1;
    }
}
```

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.Ordered;

public class MayiktBeanPost02 implements BeanPostProcessor, Ordered {
    /**
```

```
     * 调用初始化方法之前执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("[MayiktBeanPost02:]调用该 bean 的 init 方法之前");
        return bean;
    }

    /**
     * 调用初始化方法之后执行
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("[MayiktBeanPost02:]调用该 bean 的 init 方法之后");
        return bean;
    }

    public int getOrder() {
        return 0;
    }
}
```

```
<!-- 后置处理器-->
<bean id="mayiktBeanPost" class="com.mayikt.bean.MayiktBeanPost"></bean>
<bean id="mayiktBeanPost02" class="com.mayikt.bean.MayiktBeanPost02"></bean>
```

实现 Ordered 接口 getOrder 值越小越优先加载

SpringBean 的自动装配

什么是自动装配呢
根据装配的规则（属性的名称或者属性的类型）
Spring 根据装配的规则自动为属性注入值。

1. 什么是自动装配

A. 根据指定装配规则（属性名称或者属性的类型），spring 自动将匹配属性的值注入。

```
<?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:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util.xsd">

    <!-- spring ioc      <bean id="empEntity" class="com.mayikt.entity.EmpEntity">
         <property name="deptEntity" ref="deptEntity"></property>
    </bean> -->

    <!-- bean 标签中有一个属性 autowire
         1.根据属性的名称注入 bean 的 id 名称与属性的名称一致
         2.根据属性的类型注入 bean 的类型与属性类型一致
       -->
    <bean id="empEntity" class="com.mayikt.entity.EmpEntity" autowire="byType">

    </bean>
    <bean id="deptEntity" class="com.mayikt.entity.DeptEntity">
        <property name="name" value="教育部门"></property>
    </bean>

</beans>
```

SpringBean 的外部属性文件

SpringBean 的注解形式

Spring 的注解启动方式

Bean 的管理操作方式
1. 基于 XML 方式实现
2. 基于注解方式实现
什么是注解：注解是 JDK5 中推出的新特性，代码的特殊标记，
格式 注解名称“属性名称=属性值，属性名称=属性值”。
我们在后期学习 springboot 开发基本上都是使用注解，很少在使用 Xml 配置的方式。

```
@Bean
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor threadPoolTaskExecutor = new ThreadPoolTaskExecutor();
    threadPoolTaskExecutor.setCorePoolSize(corePoolSize);
    threadPoolTaskExecutor.setMaxPoolSize(maxPoolSize);
    threadPoolTaskExecutor.setQueueCapacity(queueCapacity);
    threadPoolTaskExecutor.setKeepAliveSeconds(keepAlive);
    threadPoolTaskExecutor.setThreadNamePrefix("MayiktThread-");
    threadPoolTaskExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
    threadPoolTaskExecutor.setWaitForTasksToCompleteOnShutdown(true);
    return threadPoolTaskExecutor;
}
```

注解可以使用在类、方法、属性、上面。
使用注解的目的，简化 xml 的配置方式。
Spring 提供的常用注解

1. @Component 将对象注入 Spring 容器中

2. @Service 注入业务逻辑对象

3. @Controller 控制器类

4. @Repository 注入 dao 对象

5. 以上该四个注解底层都是基于@Component 注解封装的，只是区分用于

在不同的场景下。

注解的使用方式

```
AnnotationConfigApplicationContext app = new AnnotationConfigApplicationContext();
app.register(BeanConfig.class);
app.refresh();
MemberEntity  memberEntity = (MemberEntity) app.getBean("memberEntity");
System.out.println(memberEntity);
```

SpringBean 的注解启动作用

SpringBean 的注解扫描配置

Autowired 与 Qualifier 注解

@Resource 用法

SpringBean 的 AOP

AOP 基本的概念

AOP 基本的作用

静态代理与动态代理

静态代理
动态代理

@AspectJ 注解用法

使用 aop 统一打印日志

SpringBean 的事务操作

事务的分类

手动事务

编程事务

Spring 事务传播行为

什么是 Spring 事务传播行为

1.事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。
2.例如：方法 A 事务方法调用方法 B 事务方法时，方法 B 是继续在调用者方法 A 的事务中运行，还是为自己开启一个新事务运行，这就是由方法 B 的事务传播行为决定的。

Spring 事务七种传播行为

* 保证同一个事务中

1.PROPAGATION_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

2.PROPAGATION_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。但是对于事务同步的事务管理器，PROPAGATION_SUPPORTS 与不使用事务有少许不同。

3.PROPAGATION_MANDATORY 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

* 保证没有在同一个事务中

4.PROPAGATION_REQUIRES_NEW 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

5.PROPAGATION_NOT_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。

6.PROPAGATION_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常

7.PROPAGATION_NESTED 如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按 TransactionDefinition.PROPAGATION_REQUIRED 属性执行

相关测试接口：

http://127.0.0.1:8080/insertOrderAndMember?orderName=mayikt&orderId=111&userName=mayikt&userAge=0

http://127.0.0.1:8080/insertMember?orderName=mayikt&orderId=111&userName=mayikt&userAge=22

通俗易懂方式：

1.PROPAGATION_REQUIRED-- 默认事务传播行为
当前线程如果存在事务，则加入当前线程的事务，如果当前线程不存在事务
，则创建一个新事务；

2.PROPAGATION_SUPPORTS
当前线程如果存在事务，则加入当前事务，如果当前线程不存在事务，则以非事务方式执行

3. PROPAGATION_MANDATORY
当前线程如果存在事务，则加入当前事务，如果当前线程存在事务则抛出异常

4. PROPAGATION_REQUIRES_NEW
当前线程如果存在事务，则会挂起当前事务，创建一个新的事务

5. PROPAGATION_NOT_SUPPORTED
以非事务的方式执行

6. PROPAGATION_NEVER
总是以非事务方式执行，如果当前线程存在事务则会抛出异常

7. PROPAGATION_NESTED
当前线程如果存在事务，则会嵌套一个事务。