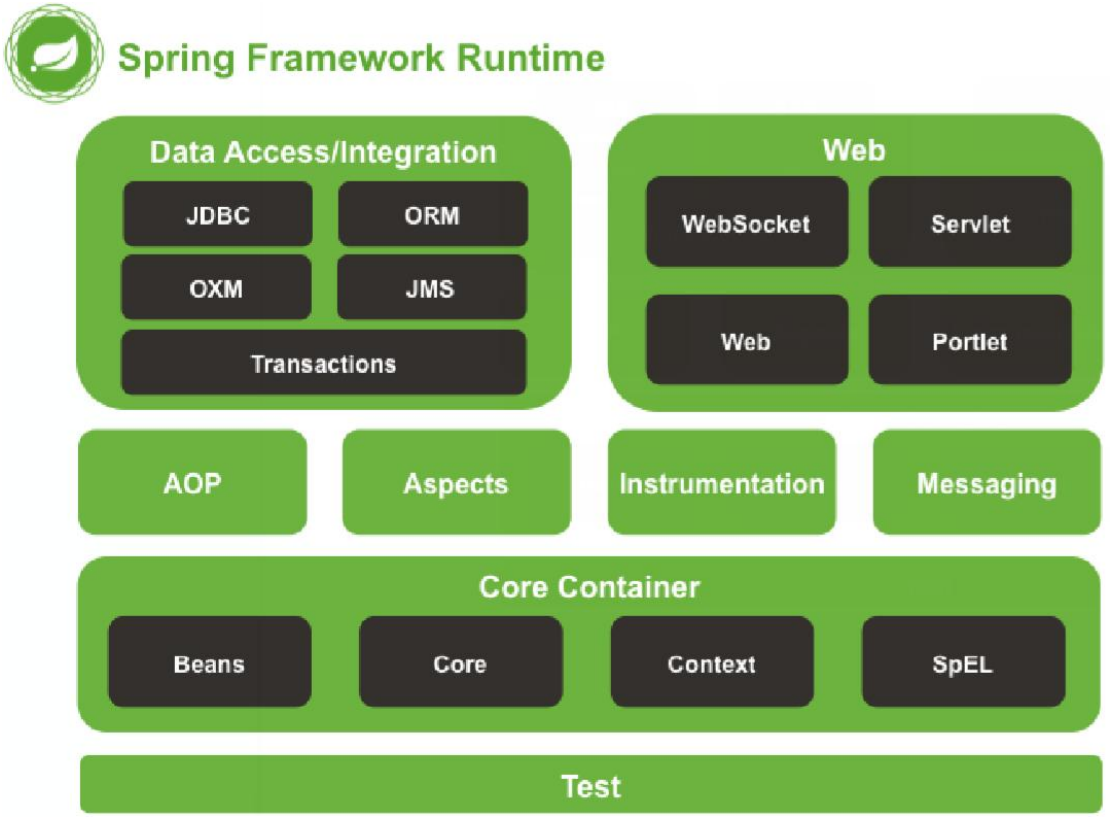
第四讲 Spring 框架之 IOC 容器

一、Spring 概述

1. spring 是什么

Spring 是分层的 Java SE/EE 应用 full-stack 轻量级开源框架，以 IoC（Inverse Of Control： 反转控制）和 AOP（Aspect Oriented Programming：面向切面编程）为内核，提供了展现层 Spring MVC 和持久层 Spring JDBC 以及业务层事务管理等众多的企业级应用技术，还能整合开源世界众多 著名的第三方框架和类库，逐渐成为使用最多的 Java EE 企业应用开源框架。

2. spring 的体系结构



二、IoC 的概念和作用

1. 程序的耦合和解耦

① 什么是程序的耦合

耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差(降低耦合性，可以提高其独立性)。耦合性存在于各个领域，而非软件设计中独有的，但是我们只讨论软件工程中的耦合。

在软件工程中，耦合指的就是就是对象之间的依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

它有如下分类：

- 内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。
- 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。在具有大 量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。
- 外部耦合 。一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传 递该全局变量的信息，则称之为外部耦合。
- 控制耦合 。一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进 行适当的动作，这种耦合被称为控制耦合。
- 标记耦合 。若一个模块 A 通过接口向两个模块 B 和 C 传递一个公共参数，那么称模块 B 和 C 之间 存在一个标记耦合。
- 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。数据耦合是最低的一种耦合形 式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。
- 非直接耦合 。两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

总结： 耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上应采用以下原则：

如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合。

② 内聚与耦合

内聚标志一个模块内各个元素彼此结合的紧密程度，它是信息隐蔽和局部化概念的自然扩展。内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当恰好做一件事。它描述的是模块内的功能联系。耦合是软件结构中各模块之间相互连接的一种度量，耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。 程序讲究的是低耦合，高内聚。就是同一个模块内的各个元素之间要高度紧密，但是各个模块之间的相互依存度却要不那么紧密。

内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。在进行软件设计时，应力争做到高内聚，低耦合。

在开发中，有些依赖关系是必须的，有些依赖关系可以通过优化代码来解除的。

请看下面的示例代码：

```
public class AccountServiceImpl implements IAccountService {
   private IAccountDao accountDao = new AccountDaoImpl();
 }
```

业务层调用持久层，并且此时业务层在依赖持久层的接口和实现类。如果此时没有持久层实现类，编译将不能通过。这种编译期依赖关系，应该在开发中杜绝。需要优化代码解决。

再比如：

```
public class JdbcDemo1 {
   public static void main(String[] args) throws Exception {
    //1.注册驱动
    //DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    Class.forName("com.mysql.jdbc.Driver");
   ......
   }
}
```

JdbcDemo1 依赖了数据库的具体驱动类（MySQL），如果这时候更换了数据库品牌（比如 Oracle），需要修改源码来重新数据库驱动。这显然不是我们想要的。

解决程序耦合的思路

③ 工厂模式解耦

在实际开发中我们可以把三层的对象都使用配置文件配置起来，当启动服务器应用加载的时候，让一个类中的方法通过读取配置文件，把这些对象创建出来并存起来。在接下来的使用的时候，直接拿过来用就好了。 那么，这个读取配置文件，创建和获取三层对象的类就是工厂。

④ 控制反转-Inversion Of Control

- 存哪去？

  分析：由于我们是很多对象，肯定要找个集合来存。这时候有 Map 和 List 供选择。

  到底选 Map 还是 List 就看我们有没有查找需求。有查找需求，选 Map。

  所以答案就是

  在应用加载时，创建一个 Map，用于存放三层对象。

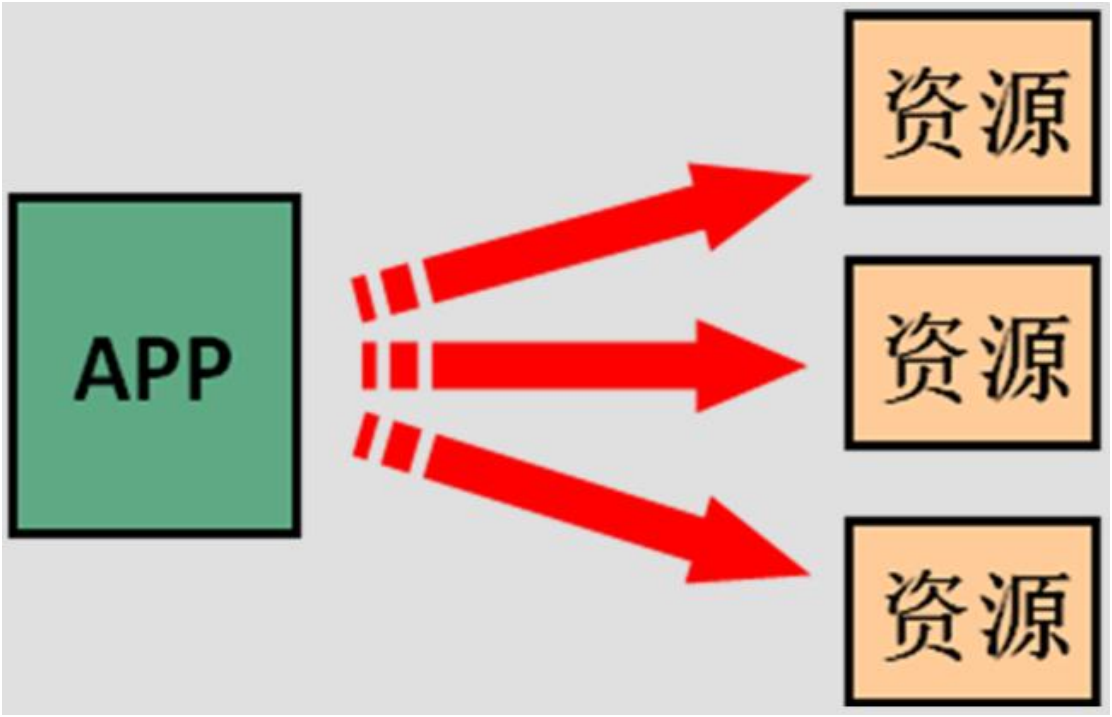
  把这个 map 称之为容器。

- 什么是工厂？

  工厂就是负责从容器中获取指定对象的类。这时候我们获取对象的方式发生了改变。

  原来：

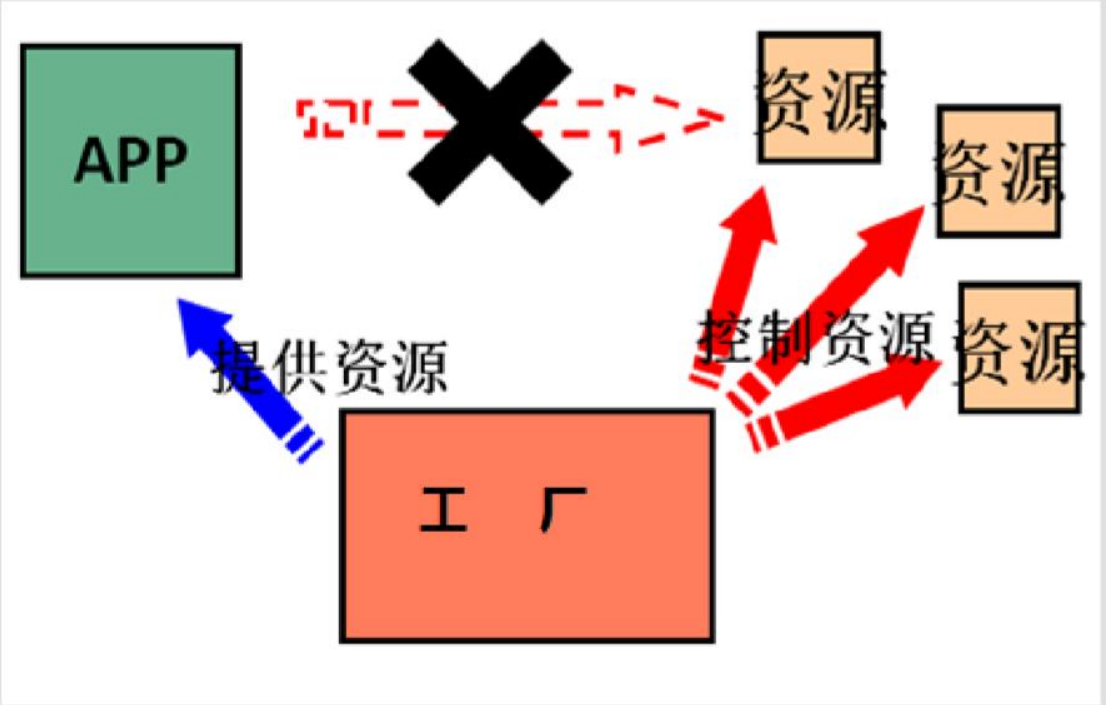
  在获取对象时，都是采用 new 的方式。是主动的。



现在：

获取对象时，跟工厂要，有工厂为我们查找或者创建对象。是被动的。

这种被动接收的方式获取对象的思想就是控制反转，它是 spring 框架的核心之一。



明确 ioc 的作用： 削减计算机程序的耦合(解除我们代码中的依赖关系)。

2. 使用 spring 的 IOC 解决程序耦合

案例:账户的业务层和持久层的依赖关系解决。在开始 spring 的配置之前，我们要先准备 一下环境。由于我们是使用 spring 解决依赖关系，并不是真正的要做增删改查操作，所以此时没必要写实体类，使用的是 java 工程，不是 java web 工程。

- 准备 spring 的开发包
- 创建业务层接口和实现类

```
/**
 * 账户的业务层接口
  */
 public interface IAccountService {
  /**
  * 保存账户（此处只是模拟，并不是真的要保存）
   */
  void saveAccount();
 }
```

```
/**
 * 账户的业务层实现类
 */
public class AccountServiceImpl implements IAccountService {

private IAccountDao accountDao = new AccountDaoImpl();//此处的依赖关系有待解决

@Override
  public void saveAccount() {
    accountDao.saveAccount();
}
}
```

- 创建持久层接口和实现类

```
/**
 * 账户的持久层接口
*/
public interface IAccountDao {
/**
  * 保存账户
   */
  void saveAccount();
}

/**
 * 账户的持久层实现类
*/
public class AccountDaoImpl implements IAccountDao {
  @Override
  public void saveAccount() {
    System.out.println("保存了账户");
  }
}
```

- 给配置文件导入约束

在类的根路径下创建一个任意名称的 xml 文件（不能是中文）

/spring-framework-5.0.2.RELEASE/docs/spring-framework-reference/html5/core.html

```
<?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>
```

- 在配置文件中配置 service 和 dao

```
<!-- bean 标签：用于配置让 spring 创建对象，并且存入 ioc 容器之中
          id 属性：对象的唯一标识。
          class 属性：指定要创建对象的全限定类名
 -->
<!-- 配置 service -->
      <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"> </bean>
<!-- 配置 dao -->
 <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl"></bean>
```

- 测试配置是否成功

```
/**
 * 模拟一个表现层
 */
  public class Client {
  /**
        * 使用 main 方法获取容器测试执行
  */
  public static void main(String[] args) {
    //1.使用 ApplicationContext 接口，就是在获取 spring 容器
    ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
    //2.根据 bean 的 id 获取对象
    IAccountService aService = (IAccountService) ac.getBean("accountService");
    System.out.println(aService);
  IAccountDao aDao = (IAccountDao) ac.getBean("accountDao");
    System.out.println(aDao);
  }
 }
```

运行结果：

3. Spring 基于 XML 的 IOC 细节

① spring 中工厂的类结构图

② BeanFactory 和 ApplicationContext 的区别

BeanFactory 才是 Spring 容器中的顶层接口。

ApplicationContext 是它的子接口。

BeanFactory 和 ApplicationContext 的区别： 创建对象的时间点不一样。

ApplicationContext：只要一读取配置文件，默认情况下就会创建对象。

BeanFactory：什么使用什么时候创建对象。

③ ApplicationContext 接口的实现类

ClassPathXmlApplicationContext： 它是从类的根路径下加载配置文件 推荐使用这种

FileSystemXmlApplicationContext： 它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。

AnnotationConfigApplicationContext:

当我们使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。

4. IOC 中 bean 标签和管理对象细节

① bean 标签

- 作用：

用于配置对象让 spring 来创建的。

默认情况下它调用的是类中的无参构造函数。如果没有无参构造函数则不能创建成功。

- 属性：

id：给对象在容器中提供一个唯一标识。用于获取对象。

class：指定类的全限定类名。用于反射创建对象。默认情况下调用无参构造函数。

scope：指定对象的作用范围。

* singleton :默认值，单例的.

* prototype :多例的.

* request :WEB 项目中,Spring 创建一个 Bean 的对象,将对象存入到 request 域中.

* session :WEB 项目中,Spring 创建一个 Bean 的对象,将对象存入到 session 域中.

* global session :WEB 项目中,应用在 Portlet 环境.如果没有 Portlet 环境那么 globalSession 相当于 session.

init-method：指定类中的初始化方法名称。

destroy-method：指定类中销毁方法名称。

② bean 的作用范围和生命周期

- 单例对象：scope="singleton"

一个应用只有一个对象的实例。它的作用范围就是整个引用。

生命周期：

对象出生：当应用加载，创建容器时，对象就被创建了。

对象活着：只要容器在，对象一直活着。

对象死亡：当应用卸载，销毁容器时，对象就被销毁了。

- 多例对象：scope="prototype"

每次访问对象时，都会重新创建对象实例。

生命周期：

对象出生：当使用对象时，创建新的对象实例。

对象活着：只要对象在使用中，就一直活着。

对象死亡：当对象长时间不用时，被 java 的垃圾回收器回收了。

③ 实例化 Bean 的三种方式

- 第一种方式：使用默认无参构造函数

```
<!--在默认情况下：
    它会根据默认无参构造函数来创建类对象。如果 bean 中没有默认无参构造函数，将会创建失败。
 -->
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"/>
```

- 第二种方式：spring 管理静态工厂-使用静态工厂的方法创建对象

```
/**
 * 模拟一个静态工厂，创建业务层实现类  */
 public class StaticFactory {
    public static IAccountService createAccountService(){
    return new AccountServiceImpl();
   }
 }
<!-- 此种方式是:
    使用 StaticFactory 类中的静态方法 createAccountService 创建对象，并存入 spring 容器
    id 属性：指定 bean 的 id，用于从容器中获取
    class 属性：指定静态工厂的全限定类名
    factory-method 属性：指定生产对象的静态方法
-->
 <bean id="accountService" class="com.itheima.factory.StaticFactory" factory-method="createAccountService"></bean>
```

- 第三种方式：spring 管理实例工厂-使用实例工厂的方法创建对象

```
/**
 * 模拟一个实例工厂，创建业务层实现类
  * 此工厂创建对象，必须现有工厂实例对象，再调用方法
  */
 public class InstanceFactory {
    public IAccountService createAccountService(){
    return new AccountServiceImpl();
```

```
  }
}
<!-- 此种方式是:
     先把工厂的创建交给 spring 来管理。
   然后在使用工厂的 bean 来调用里面的方法
   factory-bean 属性:用于指定实例工厂 bean 的 id。
   factory-method 属性:用于指定实例工厂中创建对象的方法。
  -->
<bean id="instancFactory" class="com.itheima.factory.InstanceFactory"></bean>
  <bean id="accountService"
        factory-bean="instancFactory"
        factory-method="createAccountService"></bean>
```

三、spring 的依赖注入

1. 依赖注入的概念

依赖注入:Dependency Injection。它是 spring 框架核心 ioc 的具体实现。
我们的程序在编写时,通过控制反转,把对象的创建交给了 spring,但是代码中不可能出现没有依赖的情况。 ioc 解耦只是降低他们的依赖关系,但不会消除。例如:我们的业务层仍会调用持久层的方法。
那这种业务层和持久层的依赖关系,在使用 spring 之后,就让 spring 来维护了。
简单的说,就是坐等框架把持久层对象传入业务层,而不用我们自己去获取。

2. 构造函数注入

顾名思义,就是使用类中的构造函数,给成员变量赋值。注意,赋值的操作不是我们自己做的,而是通过配置的方式,让 spring 框架来为我们注入。具体代码如下:

```
public class AccountServiceImpl implements IAccountService {
    private String name;
   private Integer age;
   private Date birthday;
      public AccountServiceImpl(String name, Integer age, Date birthday) {
    this.name = name;
```

```
    this.age = age;
    this.birthday = birthday;
  }

 @Override
  public void saveAccount() {
    System.out.println(name+","+age+","+birthday);
    }
 }

<!-- 使用构造函数的方式，给 service 中的属性传值
  要求：      类中需要提供一个对应参数列表的构造函数。
  涉及的标签：
    constructor-arg
      属性：
        index:指定参数在构造函数参数列表的索引位置
        type:指定参数在构造函数中的数据类型
        name:指定参数在构造函数中的名称         用这个找给谁赋值
      =======上面三个都是找给谁赋值，下面两个指的是赋什么值的==============
      value:它能赋的值是基本数据类型和 String 类型
        ref:它能赋的值是其他 bean 类型，也就是说，必须得是在配置文件中配置过的 bean
-->
 <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
 <constructor-arg name="name" value="  张三  "></constructor-arg>
  <constructor-arg name="age" value="18"></constructor-arg>
 <constructor-arg name="birthday" ref="now"></constructor-arg>
 </bean>

<bean id="now" class="java.util.Date"></bean>
```

3. set 方法注入

是在类中提供需要注入成员的 set 方法。具体代码如下：

```
public class AccountServiceImpl implements IAccountService {
      private String name;
```

```
    private Integer age;
    private Date birthday;
        public void setName(String name) {
      this.name = name;
    }
    public void setAge(Integer age) {
      this.age = age;
    }
    public void setBirthday(Date birthday) {
      this.birthday = birthday;
    }

  @Override
    public void saveAccount() {
      System.out.println(name+","+age+","+birthday);
      }
  }

<!-- 通过配置文件给 bean 中的属性传值：使用 set 方法的方式
    涉及的标签：
      property
    属性：
        name：找的是类中 set 方法后面的部分
        ref：给属性赋值是其他 bean 类型的
        value：给属性赋值是基本数据类型和 string 类型的
    实际开发中，此种方式用的较多。
  -->
  <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
      <property name="name" value="test"></property>
    <property name="age" value="21"></property>
      <property name="birthday" ref="now"></property>
</bean>
  <bean id="now" class="java.util.Date"></bean>
```

4. 使用 p 名称空间注入数据（本质还是调用 set 方法）

此种方式是通过在 xml 中导入 p 名称空间，使用 p:propertyName 来注入数据，它的本质仍然是调用类中的 set 方法实现注入功能。

Java 类代码：

```
/**
 * 使用 p 名称空间注入，本质还是调用类中的 set 方法
  */
 public class AccountServiceImpl4 implements IAccountService {
      private String name;
   private Integer age;
   private Date birthday;
      public void setName(String name) {
    this.name = name;
   }
   public void setAge(Integer age) {
    this.age = age;
   }
   public void setBirthday(Date birthday) {
    this.birthday = birthday;
   }
 @Override
   public void saveAccount() {
    System.out.println(name+","+age+","+birthday);
    }
 }
```

配置文件代码：

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:p="http://www.springframework.org/schema/p"
              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="accountService"
      class="com.itheima.service.impl.AccountServiceImpl4"              p:name="test"  p:age="21"
p:birthday-ref="now"/>
```

```
</beans>
```

5. 注入集合属性

顾名思义，就是给类中的集合成员传值，它用的也是 set 方法注入的方式，只不过变量的数据类型都是集合。 我们这里介绍注入数组，List,Set,Map,Properties。具体代码如下：

```
public class AccountServiceImpl implements IAccountService {
    private String[] myStrs;
  private List<String> myList;
  private Set<String> mySet;
  private Map<String,String> myMap;
  private Properties myProps;
    public void setMyStrs(String[] myStrs) {
   this.myStrs = myStrs;
  }
  public void setMyList(List<String> myList) {
   this.myList = myList;
  }
  public void setMySet(Set<String> mySet) {
   this.mySet = mySet;
  }
  public void setMyMap(Map<String, String> myMap) {
   this.myMap = myMap;
  }
  public void setMyProps(Properties myProps) {
   this.myProps = myProps;
  }

 @Override
  public void saveAccount() {
   System.out.println(Arrays.toString(myStrs));
   System.out.println(myList);
   System.out.println(mySet);
   System.out.println(myMap);
  System.out.println(myProps);
```

```
    }
  }

<!-- 注入集合数据
      List 结构的：      array,list,set
  Map 结构的     map,entry,props,prop
-->
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
  <!-- 在注入集合数据时，只要结构相同，标签可以互换 -->
    <!-- 给数组注入数据 -->
    <property name="myStrs">
    <set>
        <value>AAA</value>
        <value>BBB</value>
        <value>CCC</value>
      </set>
  </property>
  <!-- 注入 list 集合数据 -->
    <property name="myList">
      <array>
        <value>AAA</value>
        <value>BBB</value>
        <value>CCC</value>
      </array>
    </property>
  <!-- 注入 set 集合数据 -->
    <property name="mySet">
    <list>
        <value>AAA</value>
        <value>BBB</value>
        <value>CCC</value>
      </list>
    </property>
  <!-- 注入 Map 数据 -->
    <property name="myMap">
      <props>
```

```
    <prop key="testA">aaa</prop>
      <prop key="testB">bbb</prop>
    </props>
  </property>
  <!-- 注入 properties 数据 -->
  <property name="myProps">
    <map>
      <entry key="testA" value="aaa"></entry>
    <entry key="testB">
       <value>bbb</value>
    </entry>
    </map>
  </property>
</bean>
```

四、基于注解的 IOC 配置

1. 环境搭建

2. 使用@Component 注解配置管理的资源

```
/**
 * 账户的业务层实现类
 */
@Component("accountService")
 public class AccountServiceImpl implements IAccountService {
      private IAccountDao accountDao;

 public void setAccountDao(IAccountDao accountDao) {
    this.accountDao = accountDao;
  }
 }

/**
```

```
 * 账户的持久层实现类
 */
@Component("accountDao")
 public class AccountDaoImpl    implements IAccountDao {


 private DBAssit dbAssit;
}
```

注意：　当我们使用注解注入时，set 方法不用写

3. 创建 spring 的 xml 配置文件并开启对注解的支持

注意：　基于注解整合时，导入约束时需要多导入一个 context 名称空间下的约束。

```
<?xml version="1.0" encoding="UTF-8"?> <beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
      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
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">


 <!-- 告知 spring 创建容器时要扫描的包 -->
   <context:component-scan base-package="com.itheima"></context:component-scan>


</beans>
```

4. 常用注解

① 用于创建对象的

相当于：<bean id="" class="">

- @Component

作用：

把资源让 spring 来管理。相当于在 xml 中配置一个 bean。

属性：

value：指定 bean 的 id。如果不指定 value 属性，默认 bean 的 id 是当前类的类名。首字母小写。

- @Controller @Service @Repository

他们三个注解都是针对一个的衍生注解，他们的作用及属性都是一模一样的。 他们只不过是提供了更加明确的语义化。

@Controller：一般用于表现层的注解。

@Service：一般用于业务层的注解。

@Repository：一般用于持久层的注解。

细节：如果注解中有且只有一个属性要赋值时，且名称是 value，value 在赋值是可以不写。

② 用于注入数据的

相当于：<property name="" ref="">

<property name="" value="">

- @Autowired

作用： 自动按照类型注入。当使用注解注入属性时，set 方法可以省略。它只能注入其他 bean 类型。当有多个 类型匹配时，使用要注入的对象变量名称作为 bean 的 id，在 spring 容器查找，找到了也可以注入成功。找不到 就报错。

- @Qualifier

作用： 在自动按照类型注入的基础之上，再按照 Bean 的 id 注入。它在给字段注入时不能独立使用，必须和 @Autowire 一起使用；但是给方法参数注入时，可以独立使用。

属性： value：指定 bean 的 id。

- @Resource

作用： 直接按照 Bean 的 id 注入。它也只能注入其他 bean 类型。

属性： name：指定 bean 的 id。

- @Value

作用： 注入基本数据类型和 String 类型数据的

属性： value：用于指定值

③ 用于改变作用范围的：

相当于：<bean id="" class="" scope="">

- @Scope

作用： 指定 bean 的作用范围。

属性： value：指定范围的值。

取值：singleton prototype request session globalsession

5. 新注解说明

① 待改造的问题

我们发现，之所以现在离不开 xml 配置文件，是因为有一句很关键的配置：

```
<!-- 告知 spring 框架在，读取配置文件，创建容器时，扫描注解，依据注解创建对象，并存入容器中 -->
 <context:component-scan base-package="com.itheima"></context:component-scan>
```

如果他要也能用注解配置，那么就离脱离 xml 文件又进了一步。

② @Configuration

- 作用：

  用于指定当前类是一个 spring 配置类，当创建容器时会从该类上加载注解。获取容器时需要使用 AnnotationApplicationContext(有@Configuration 注解的类.class)。

- 属性：

  value:用于指定配置类的字节码

- 示例代码：

```
/**
* spring 的配置类，相当于 bean.xml 文件
*/
@Configuration
public class SpringConfiguration {
}
```

注意： 这里已经把配置文件用类来代替了

③ @ComponentScan

- 作用： 用于指定 spring 在初始化容器时要扫描的包。作用和在 spring 的 xml 配置文件中的： <context:component-scan base-package="com.itheima"/>是一样的。
- 属性： basePackages：用于指定要扫描的包。和该注解中的 value 属性作用一样。
- 示例代码：

```
/**
 * spring 的配置类，相当于 bean.xml 文件
```

```
 */
 @Configuration
@ComponentScan("com.itheima")
 public class SpringConfiguration {
 }
```

④ @Bean

- 作用： 该注解只能写在方法上，表明使用此方法创建一个对象，并且放入 spring 容器。
- 属性： name：给当前@Bean 注解方法创建的对象指定一个名称(即 bean 的 id)。
- 示例代码：

```
/**
 * 连接数据库的配置类
 */
public class JdbcConfig {

 /**
  * 创建一个数据源，并存入 spring 容器中
  */
  @Bean(name="dataSource")
  public DataSource createDataSource() {
   try {
    ComboPooledDataSource ds = new ComboPooledDataSource();
    ds.setUser("root");
    ds.setPassword("1234");
    ds.setDriverClass("com.mysql.jdbc.Driver");
   ds.setJdbcUrl("jdbc:mysql:///spring_day02");
   return ds;
   } catch (Exception e) {
    throw new RuntimeException(e);
   }
  }

 /**
  * 创建一个 DBAssit，并且也存入 spring 容器中
```

```
    * @param dataSource
    * @return
    */
   @Bean(name="dbAssit")
   public DBAssit createDBAssit(DataSource dataSource) {
    return new DBAssit(dataSource);
   }
   }
```

注意:　由于没有了配置文件，创建数据源的配置又都写死在类中了。

⑤ @PropertySource

- 作用:　用于加载.properties 文件中的配置。例如我们配置数据源时，可以把连接数据库的信息写到 properties 配置文件中, 就可以使用此注解指定 properties 配置文件的位置。
- 属性:　value[]: 用于指定 properties 文件位置。如果是在类路径下, 需要写上 classpath:
- 示例代码:

```
/**
 * 连接数据库的配置类
*/
 public class JdbcConfig {
   @Value("${jdbc.driver}")
   private String driver;
   @Value("${jdbc.url}")
   private String url;
   @Value("${jdbc.username}")
   private String username;
   @Value("${jdbc.password}")
   private String password;

  /**
   * 创建一个数据源，并存入 spring 容器中
*/
   @Bean(name="dataSource")
   public DataSource createDataSource() {
```

```
    try {
      ComboPooledDataSource ds = new ComboPooledDataSource();
      ds.setDriverClass(driver);
      ds.setJdbcUrl(url);
      ds.setUser(username);
      ds.setPassword(password);
      return ds;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

- jdbc.properties 文件

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/day44_ee247_spring
 jdbc.username=root
 jdbc.password=1234
```

⑥ @Import

- 作用： 用于导入其他配置类，在引入其他配置类时，可以不用再写@Configuration 注解。当然，写上也没问 题。
- 属性： value[]：用于指定其他配置类的字节码。
- 示例代码：

```
@Configuration
@ComponentScan(basePackages = "com.itheima.spring")
@Import({ JdbcConfig.class})
public class SpringConfiguration {
 }


@Configuration
@PropertySource("classpath:jdbc.properties")
public class JdbcConfig{
 }
```

⑦ 通过注解获取容器

```
ApplicationContext ac = new AnnotationConfigApplicationContext(SpringConfiguration.class);
```

⑧ 工程结构图