Spring-IOC与DI

第一章. Spring简介

1.1 Spring介绍

> Spring框架主页: Spring官网
>
> Spring资源地址:下载地址

- Spring框架，由Rod Johnson开发
- Spring是一个非常活跃的开源框架, 基于IOC和AOP来构架多层JavaEE系统，以帮助分离项目组件之间的依赖关系
- 它的主要目地是简化企业开发

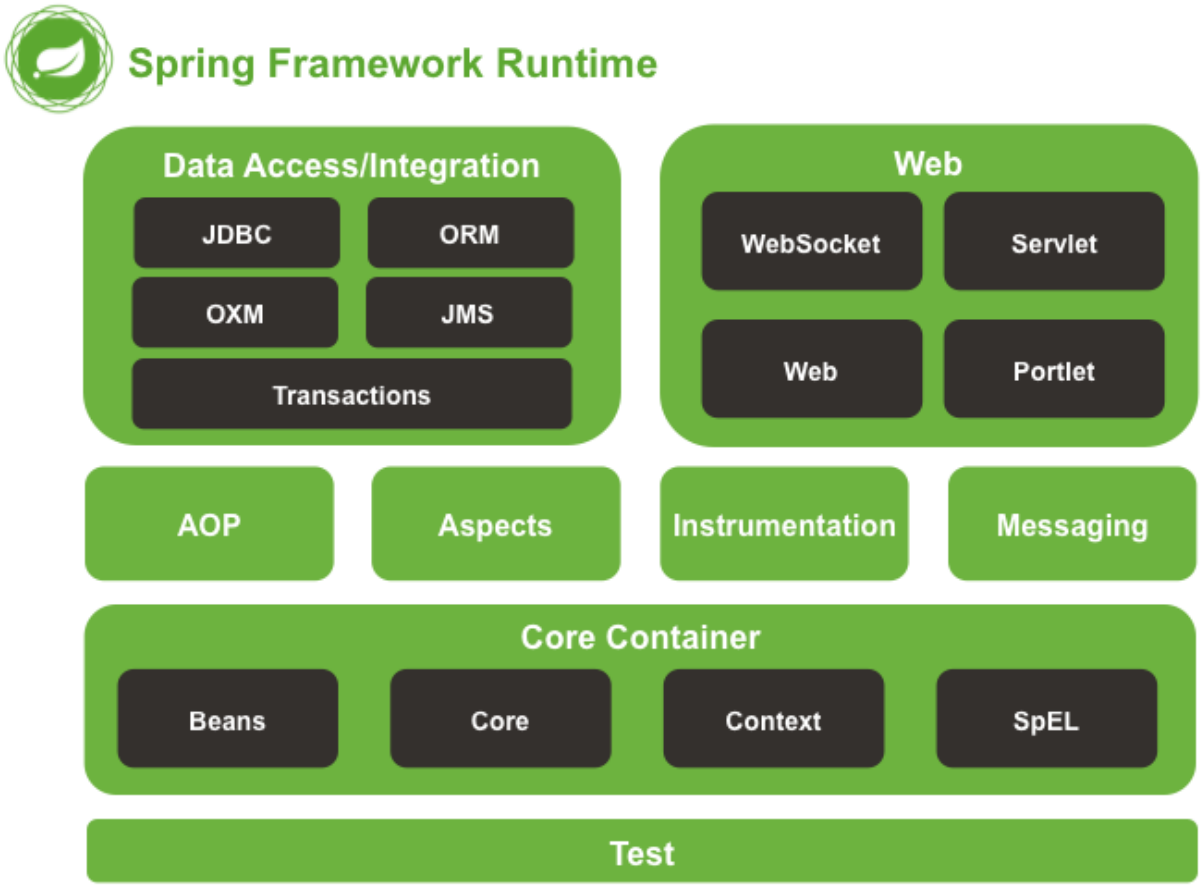
1.2 Spring解决的问题

- 方便解耦，简化开发：Spring 就是一个大工厂，可以将所有对象创建和依赖关系维护，交给 Spring 管理
- AOP 编程的支持：Spring 提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能
- 声明式事务的支持：只需要通过配置就可以完成对事务的管理，而无需手动编程
- 方便程序的测试：Spring 对 Junit4 支持，可以通过注解方便的测试 Spring 程序
- 方便集成各种优秀框架：Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz 等）的直接支持
- 降低 JavaEE API 的使用难度：Spring对 JavaEE 开发中非常难用的API（JDBC、JavaMail、远程调用等），都提供了封装，使这些 API 应用难度大大降低

1.3 Spring的组成

> Spring框架包含的功能大约由20个模块组成。这些模块按组可分为核心容器、数据访问/集成，Web，AOP(面向切面编程)、设备、消息和测试

1.3.1 Spring组成图



1.4 Core核心模块

- spring-core：依赖注入IoC与DI的最基本实现
- spring-beans：Bean工厂与bean的装配
- spring-context：spring的context上下文即IoC容器
- spring-context-support
- spring-expression：spring表达式语言

详细说明

（1）spring-core

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

（2）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 of Control / 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

（3）spring-context

Spring核心提供了大量扩展，这样使得由 Core 和 Beans 提供的基础功能增强：这意味着Spring 工程能以框架模式访问对象。Context 模块继承了Beans 模块的特性并增加了对国际化（例如资源绑定）、事件传播、资源加载和 context 透明化（例如 Servlet container）。同时，也支持JAVA EE 特性，例如 EJB、 JMX 和 基本的远程访问。Context 模块的关键是 ApplicationContext 接口。spring-context-support 则提供了对第三方库集成到 Spring-context 的支持，比如缓存（EhCache, Guava, JCache）、邮件（JavaMail）、调度（CommonJ, Quartz）、模板引擎（FreeMarker, JasperReports, Velocity）等。

（4）spring-expression

为在运行时查询和操作对象图提供了强大的表达式语言。它是JSP2.1规范中定义的统一表达式语言的扩展，支持 set 和 get 属性值、属性赋值、方法调用、访问数组集合及索引的内容、逻辑算术运算、命名变量、通过名字从Spring IoC容器检索对象，还支持列表的投影、选择以及聚合等。

数据访问与集成层包含 JDBC、ORM、OXM、JMS和事务模块。

（1）spring-jdbc

提供了 JDBC抽象层，它消除了冗长的 JDBC 编码和对数据库供应商特定错误代码的解析。

（2）spring-tx

支持编程式事务和声明式事务，可用于实现了特定接口的类和所有的 POJO 对象。编程式事务需要自己写 beginTransaction()、commit()、rollback()等事务管理方法，声明式事务是通过注解或配置由 spring 自动处理，编程式事务粒度更细。

（3）spring-orm

提供了对流行的对象关系映射 API的集成，包括 JPA、JDO 和 Hibernate 等。通过此模块可以让这些 ORM 框架和 spring 的其它功能整合，比如前面提及的事务管理。

（4）spring-oxm

模块提供了对 OXM 实现的支持，比如JAXB、Castor、XML Beans、JiBX、XStream等。

（5）spring-jms

模块包含生产（produce）和消费（consume）消息的功能。从Spring 4.1开始，集成了 spring-messaging 模块

Spring 处理Web层jar

Web 层包括 spring-web、spring-webmvc、spring-websocket、spring-webmvc-portlet 等模块。

详细说明

（1）spring-web

提供面向 web 的基本功能和面向 web 的应用上下文，比如 multipart 文件上传功能、使用 Servlet 监听器初始化 IoC 容器等。它还包括 HTTP 客户端以及 Spring 远程调用中与 web 相关的部分

（2）spring-webmvc

为 web 应用提供了模型视图控制（MVC）和 REST Web 服务的实现。Spring 的 MVC 框架可以使领域模型代码和 web 表单完全地分离，且可以与 Spring 框架的其它所有功能进行集成

（3）spring-webmvc-portlet

（即Web-Portlet模块）提供了用于 Portlet 环境的 MVC 实现，并反映了 pring-webmvc 模块的功能

Spring AOP涉及jar

（1）spring-aop

提供了面向切面编程（AOP）的实现，可以定义诸如方法拦截器和切入点等，从而使实现功能的代码彻底的解耦。使用源码级的元数据。

（2）spring-aspects

提供了对 AspectJ 的集成

Instrumentation 模块涉及jar

（1）spring-instrument

模块提供了对检测类的支持和用于特定的应用服务器的类加载器的实现。

（2）spring-instrument-tomcat

模块包含了用于 Tomcat 的Spring 检测代理。

Messaging消息处理 涉及jar

spring-messaging 模块

从 Spring 4 开始集成，从一些 Spring 集成项目的关键抽象中提取出来的。这些项目包括 Message、MessageChannel、MessageHandler 和其它服务于消息处理的项目。这个模块也包含一系列的注解用于映射消息到方法

Test模块涉及jar

spring-test 模块

通过 JUnit 和 TestNG 组件支持单元测试和集成测试。它提供了一致性地加载和缓存 Spring 上下文，也提供了用于单独测试代码的模拟对象（mock object）

第二章. 入门程序和IOC简介

依赖注入或控制反转的定义中，调用者不负责被调用者的实例创建工作，该工作由Spring框架中的容器来负责，它通过开发者的配置来判断实例类型，创建后再注入调用者。由于Spring容器负责被调用者实例，实例创建后又负责将该实例注入调用者，因此称为依赖注入。而被调用者的实例创建工作不再由调用者来创建而是由Spring来创建，控制权由应用代码转移到了外部容器，控制权发生了反转，因此称为控制反转

2.1 IOC-控制反转

2.1.1 IOC-控制反转

IOC是 Inverse of Control 的简写，意思是控制反转。是降低对象之间的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，交给Spring容器完成。具体的过程是，程序读取Spring 配置文件，获取需要创建的 bean 对象，

通过反射机制创建对象的实例。

缺点：对象是通过反射机制实例化出来的，因此对系统的性能有一定的影响。

将对象的创建权利翻转给Spring容器.

2.1.2 DI-依赖注入

Dependency Injection，说的是创建对象实例时，同时为这个对象注入它所依赖的属性。相当于把每个bean与bean之间的关系交给容器管理。而这个容器就是spring。

例如我们通常在 Service 层注入它所依赖的 Dao 层的实例；在 Controller层注入 Service层的实例。

2.1.3 IOC和DI

IOC的别名,2004年，Martin Fowler探讨了同一个问题，既然IOC是控制反转，那么到底是"哪些方面的控制被反转了呢？"，经过详细地分析和论证后，他得出了答案："获得依赖对象的过程被反转了"。控制被反转之后，获得依赖对象的过程由自身管理对象变为由IOC容器主动注入。于是，他给"控制反转"取了一个更合适的名字叫做"依赖注入（Dependency Injection，DI）"。他的这个答案，实际上给出了实现IOC的方法：注入。

所谓依赖注入，就是由IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。

所以，依赖注入（DI）和控制反转（IOC）是从不同的角度描述的同一件事情，就是指通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦.

2.2 入门练习

2.2.1 创建web项目

项目名: spring-01-IOC

2.2.2 引入jar包

> 如果练习IOC ,需要引入 `org.springframework.beans` , `org.springframework.context`

jar包位置!Spring下载文件中/libs文件下!: beans,context,core,context-support,spring-expression.

Maven项目,pom.xml添加一下内容:

```
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.1.3.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.1.3.RELEASE</version>
    </dependency>

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

```
      <version>5.1.3.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context-support</artifactId>
      <version>5.1.3.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-expression</artifactId>
      <version>5.1.3.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.2</version>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
  </dependency>
```

2.2.3 引入log4j.properties

文件位置:src目录下

maven项目:resources目录下

```
log4j.rootLogger=ERROR,A1
log4j.logger.org.mybatis = ERROR
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n
```

2.2.4 创建一个操作类

```
package com.itqf.spring.bean;


public class Person {

    private String name;
    private Integer age;

    public Person() {
        System.out.println("------------> Person.Person");
    }
    //getter,setter
}
```

2.2.5 创建配置文件

建议命名: applicationContext.xml

建议位置: 普通项目src下 / maven项目 resources下

```
<?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">

        <!--
             id 和 name 可以同时存在,作为bean的标识
             class添加的应该是class的全路径
         -->
       <bean
           id="personId" name="personName" class="com.itqf.spring.bean.Person"
        />

</beans>
```

2.2.6 测试用例

```
 @Test
 public void test1(){
        //TODO 测试IOC
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //getBean 可以使用配置文件中的id值,也可以使用配置文件的name值.
        Person person = (Person) applicationContext.getBean("personId");
        System.out.println("person = " + person);
  }
```

第三章. 对象创建的细节

> 准备工作: 创建配置文件: applicationContext-bean.xml
>
> 创建测试代码: BeanTest.java

3.1 bean标签和属性讲解

bean标签

bean标签,是根标签beans内部必须包含的标签,它是用于声明具体的类 的对象!

bean标签对应属性

Property	属性解释
class	指定bean对应类的全路径
name	name是bean对应对象的一个标识
scope	执行bean对象创建模式和生命周期
id	id是bean对象的唯一标识,不能添加特别字符
lazy-init	是否延时加载 默认值:false
init-method	对象初始化方法
destory	对象销毁方法

3.2 创建对象工厂

3.2.1 FileSystemXmlApplicationContext

从硬盘绝对路径下加载配置文件

```
    @Test
    public void testBeanFactory1(){
      //通过绝对路径加载配置文件
      ApplicationContext context = new
FileSystemXmlApplicationContext("D:\\workspace\\spring\\spring-01-
IOC\\src\\applicationContext-bean.xml");
    }
```

3.2.2 ClassPathXmlApplicationContext

从类路径下加载配置文件

普通项目: src目录下

maven项目: resources目录下

```
@Test
public void testBeanFactory2(){
    ApplicationContext context2 = new
 ClassPathXmlApplicationContext("applicationContext-bean.xml");
}
```

3.3 练习bean标签属性

3.3.1 name属性

可以重复,可以使用特殊字符

3.3.2 id属性

id属性作用和name几乎相同,但是也有细微的差别,id不可重复,且不能使用特殊字符

```
<!--同时添加name和id -->
<bean name="testBeanName"  id="testBeanId"  class="com.itqf.spring.bean.TestBean" />
```

Java代码

```
 ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext-bean.xml");
        /**
         * 参数1: name/id
         * 参数2(可选): 可以指定生成对象类型,如果不填此参数,需进行强转
         * 两种方式都可以获取!
         */
        TestBean testBeanName = applicationContext.getBean("testBeanName",  TestBean.class);
        TestBean testBeanId = applicationContext.getBean("testBeanId",  TestBean.class);
```

3.3.3 scope属性

bean标签中添加scope属性,设置bean对应对象生成规则.

scope = "singleton"

单例,默认值,适用于实际开发中的绝大部分情况.

配置:

```
<bean name="testBeanName" scope="singleton" id="testBeanId"
class="com.itqf.spring.bean.TestBean" />
```

测试:

```
@Test
public void test2(){
        //TODO 测试bean标签中 scope = singleton
```

```
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext-bean.xml");
        /**
         * 参数1: name/id
         * 参数2(可选): 可以指定生成对象类型,如果不填此参数,需进行强转
         * 两种方式都可以获取!
         */
        TestBean testBeanName = applicationContext.getBean("testBeanName", TestBean.class);
        TestBean testBeanId = applicationContext.getBean("testBeanId", TestBean.class);

        System.out.println(testBeanName == testBeanId); //打印 true

    }
```

scope="prototype"

多例,适用于struts2中的action配置

配置:

```
<bean name="testBeanName" scope="prototype" id="testBeanId"
class="com.itqf.spring.bean.TestBean" />
```

测试

```
@Test
public void test2(){
        //TODO 测试bean标签中 scope = prototype

        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext-bean.xml");
        /**
         * 参数1: name/id
         * 参数2(可选): 可以指定生成对象类型,如果不填此参数,需进行强转
         * 两种方式都可以获取!
         */
        TestBean testBeanName = applicationContext.getBean("testBeanName", TestBean.class);
        TestBean testBeanId = applicationContext.getBean("testBeanId", TestBean.class);

        System.out.println(testBeanName == testBeanId); //打印 false
}
```

3.3.4 lazy-init属性

注意: 只对单例有效,设置scope="singleton"时测试

延时创建属性.

lazy-init="false" 默认值,不延迟创建,即在启动时候就创建对象.

lazy-init="true" 延迟初始化,在用到对象的时候才会创建对象.

配置:

```
<bean name="testBeanName" id="testBeanId"  scope="singleton" lazy-init="false"
class="com.itqf.spring.bean.TestBean" />
```

测试1: lazy-init="false"

```
@Test
public void test2(){
   //TODO 测试bean标签中的 lazy-init="false" 默认值
   ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext-bean.xml");
   System.out.println("获取数据之前!");
    /**
      * 参数1: name/id
      * 参数2(可选): 可以指定生成对象类型,如果不填此参数,需进行强转
      * 两种方式都可以获取!
      */
    TestBean testBeanName = applicationContext.getBean("testBeanName", TestBean.class);
    TestBean testBeanId = applicationContext.getBean("testBeanId", TestBean.class);
        System.out.println("获取数据之后!");
         //测试结果: 先输出 实体类的构造方法 --> 获取数据之前  --> 获取数据之后
     //证明: false 不延迟创建,在创建ApplicationContext的时候就创建了对象!
}
```

测试2:lazy-init="true"

```
@Test
public void test2(){
  //TODO 测试bean标签中的 lazy-init="true" 默认值

  ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext-bean.xml");
  System.out.println("获取数据之前!");
  /**
    * 参数1: name/id
    * 参数2(可选): 可以指定生成对象类型,如果不填此参数,需进行强转
    * 两种方式都可以获取!
    */
  TestBean testBeanName = applicationContext.getBean("testBeanName", TestBean.class);
  TestBean testBeanId = applicationContext.getBean("testBeanId", TestBean.class);
  System.out.println("获取数据之后!");
   //测试结果: 先输出 获取数据之前 ---> 实体类的构造方法 --> 获取数据之后
   //证明: true 延迟创建, 只有在获取的时候创建.
}
```

3.3.5 初始化/销毁

在TestBean类中添加初始化方法和销毁方法（名称自定义）：

```
public void init() {
   System.out.println("TestBean的初始化方法");
}

public void destroy() {
   System.out.println("TestBean的销毁方法");
}
```

第四章. 对象创建的几种方式

创建配置文件:applicationContext-create.xml

创建测试代码:CreateTest.java

4.1 无参构造函数

之前使用的方式调用了类的无参构造函数.

4.2 有参数构造函数

后面章节:对象的依赖-属性注入 (注入属性) 后面章节进行讲解.

4.3 静态工厂模式

创建工厂类:

```
/**
 * Person工厂类
 */
public class PersonFactory {

    public static Person  createPerson(){

        System.out.println("静态工厂创建Person");

        return  new Person();
    }
}
```

配置文件:applicationContext-create.xml

```
<bean name="personFactory" class="com.itqf.spring.utils.PersonFactory" factory-
method="createPerson" />
```

测试代码

```
 ApplicationContext context =new ClassPathXmlApplicationContext("applicationContext-
create.xml");
                //获取工场bean对应的name
 Person person = (Person) context.getBean("personFactory");

 System.out.println("person = " + person);
```

4.4 非静态工厂

创建工场类

```
/**
  * Person工厂类
  */
public class PersonFactory {
   /**
     * 非静态创建对象
     * @return Person
     */
 public  Person createPerson1(){
    System.out.println("非静态工厂创建Person");
    return new Person();
 } }
```

配置文件:applicationContext-create.xml

```
<bean name="personFactory1" class="com.itqf.spring.utils.PersonFactory" />
<bean name="personFactory2"  factory-bean="personFactory1" factory-method="createPerson1" />
```

测试代码

```
 //TODO 测试非静态工厂模式
 ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
create.xml");

//测试结果:会触发PersonFactory createPerson1方法 输出 System.out.println("非静态工厂创建Person");
```

第五章. 依赖注入

测试类:Person.java

创建配置文件：`applicationContext-injection.xml`

创建测试代码：`InjectionTest.java`

5.1 set方法注入

5.1.1 基本类型值注入使用value

配置:

```
  <!-- value值为基本类型 -->
  <bean name="person"  class="com.itqf.spring.bean.Person" >
       <property name="name" value="jeck" />
       <property name="age" value="11"/>
  </bean>
```

测试代码:

```
  @Test
  public  void  test1(){
        //TODO 测试基本数据类型注入数据
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
injection.xml");

        Person person = context.getBean("person", Person.class);

        System.out.println("person = " + person);
                //输出结果:------------> Person.Person
                //                person = Person{name='jeck', age=11}
 }
```

5.1.2 引入类型值注入ref

创建 Car.java:

```
public class Car {
    private String name;
    private String color;

    public Car() {
        super();
        System.out.println("Car的空参构造方法");
    }
    //getter、setter、toString
}
```

修改Person.java,在Person中引入Car:

```
public class Person {

    private String name;
    private Integer age;
    private Car car;
        //构造方法 getter setter toString方法
}
```

配置:利用ref属性给 person的car属性赋值

```
 <bean name="person1" class="com.itqf.spring.bean.Person">
     <property name="name" value="helen"></property>
     <property name="age" value="18"></property>
     <property name="car" ref="car"></property>
 </bean>

 <bean name="car" class="com.itqf.spring.bean.Car">
     <property name="name" value="MINI"></property>
     <property name="color" value="灰色" ></property>
 </bean>
```

测试: 使用之前测试用例即可!

5.2 构造函数注入

5.2.1 单个有参构造方法注入

在Person中创建有参构造函数:

```
public  Person(String  name , Car car){
       this.name = name;
       this.car = car;
       System.out.println("Person的有参构造方法:"+name+car);
}
```

配置:

```
<bean name="person" class="com.itqf.spring.bean.Person">
          <constructor-arg name="name" value="rose"/>
          <constructor-arg name="car"  ref="car"/>
 </bean>
 <!-- 构造函数car时候引入 -->
 <bean name="car"  class="com.itqf.spring.bean.Car" >
          <property name="name" value="mime"/>
          <property name="color" value="白色"/>
 </bean>
```

测试:

```
  @Test
  public  void test2(){
                //TODO 测试参构造方法
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
injection.xml");

        Person person = context.getBean("person", Person.class);

        System.out.println(person);
                //结果:调用有参数构造方法,输出
  }
```

5.2.2 index属性：按参数索引注入

参数名一致，但位置不一致时，使用 `index`

例如以下两个构造函数（第二个是新添加）：

```
    public Person(String name, Car car) {
        super();
        System.out.println("Person(String name, Car car)");
        this.name = name;
        this.car = car;
    }

    public Person(Car car, String name) {
        super();
        System.out.println("Person(Car car, String name)");
        this.name = name;
        this.car = car;
    }
```

配置：使用 `index` 确定调用哪个构造函数

```
    <bean name="person2" class="com.itqf.spring.bean.Person">
        <constructor-arg name="name" value="qf" index="0"></constructor-arg>
        <constructor-arg name="car" ref="car" index="1"></constructor-arg>
    </bean>
```

测试：

重新执行第一步的测试用例，执行结果调用了第一个构造函数

5.2.3 type属性：按参数类型注入

参数名和位置一致，但类型不一致时，使用`type`

例如以下两个构造函数（第二个是新添加）：

```
    public Person(Car car, String name) {
        super();
        System.out.println("Person(Car car, String name)");
        this.name = name;
        this.car = car;
    }

    public Person(Car car, Integer name) {
        super();
        System.out.println("Person(Car car, Integer name)");
        this.name = name + "";
        this.car = car;
    }
```

配置：使用`type`指定参数的类型

```
    <bean name="person2" class="com.itqf.spring.bean.Person">
        <constructor-arg name="name" value="988" type="java.lang.Integer"></constructor-arg>
        <constructor-arg name="car" ref="car" ></constructor-arg>
    </bean>
```

测试：

重新执行前面的测试用例，执行结果调用了第二个构造函数

5.3 p名称空间注入

导入p名称空间:

使用p:属性名 完成注入,走set方法

- 基本类型值: p:属性名="值"
- 引入类型值: P:属性名-ref="bean名称"

配置:

```
//1.第一步配置文件中 添加命名空间p
 xmlns:p="http://www.springframework.org/schema/p"

<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">

       //使用 p命称空间进行赋值
        <bean name="person" class="com.itqf.spring.bean.Person"  p:name="人名" p:age="11"
p:car-ref="car">

       </bean>

       <bean name="car"  class="com.itqf.spring.bean.Car" >
           <property name="name" value="mime" />
```

```
        <property name="color" value="白色"/>
    </bean>
```

测试:

```
@Test
public  void test2(){
        //TODO 测试p命名空间注入
    ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
injection.xml");

    Person person = context.getBean("person", Person.class);

    System.out.println(person);

 }
```

5.4 spel注入

spring Expression Language：spring表达式语言

配置:

```
  <bean name="car"  class="com.itqf.spring.bean.Car" >
          <property name="name" value="mime" />
          <property name="color" value="白色"/>
  </bean>

   <!--利用spel引入car的属性 -->
   <bean  name="person1"  class="com.itqf.spring.bean.Person"  p:car-ref="car">
           <property name="name" value="#{car.name}"/>
           <property name="age" value="#{person.age}"/>

   </bean>
```

测试

```
@Test
public  void test3(){
        //TODO 测试spel注入
    ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
injection.xml");

    Person person = context.getBean("person1", Person.class);

    System.out.println(person);

 }
```

5.5 复杂类型注入

创建配置文件:application-collection.xml

创建测试代码:CollectionTest.java

创建测试实体类:TestCollection

创建TestCollection:

```
/**
 * 练习:arr list map properties的注入
 */
public class TestCollection {

    private Object [] arrs;
    private List<Object> list;
    private Map<String,Object> map;
    private Properties properties;

    public Object[] getArrs() {
        return arrs;
    }

    public void setArrs(Object[] arrs) {
        this.arrs = arrs;
    }

    public List<Object> getList() {
        return list;
    }

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

    public Map<String, Object> getMap() {
        return map;
    }
```

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

    public Properties getProperties() {
        return properties;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }

    @Override
    public String toString() {
        return "TestCollection{" +
                "arrs=" + Arrays.toString(arrs) +
                ", list=" + list +
                ", map=" + map +
                ", properties=" + properties +
                '}';
    }
}
```

配置:

```
<?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 name="car" class="com.itqf.spring.bean.Car">
             <property name="name" value="保时捷"/>
             <property name="color" value="红色" />
       </bean>


       <bean name="testColl" class="com.itqf.spring.bean.TestCollection">

           <!-- 数组变量注入 -->
           <property name="arrs">
               <list>
                   <value>数组1</value>
                   <!--引入其他类型-->
                   <ref bean="car"/>
               </list>
           </property>

           <!-- 集合变量赋值-->
           <property name="list">
```

```
                <list>
                    <value>集合1</value>
                    <!--集合变量内部包含集合-->
                    <list>
                        <value>集合中的集合1</value>
                        <value>集合中的集合2</value>
                        <value>集合中的集合3</value>
                    </list>
                    <ref bean="car" />
                </list>
            </property>

            <!--map赋值 -->
            <property name="map">
                <map>
                    <entry key="car" value-ref="car" />
                    <entry key="name" value="保时捷" />
                    <entry key="age"  value="11"/>
                </map>

            </property>

            <!-- properties赋值 -->
            <property name="properties">
                 <props>
                     <prop key="name">pro1</prop>
                     <prop key="age">111</prop>
                 </props>
            </property>
        </bean>

</beans>
```

测试:

```
 @Test
 public  void  test4(){
        //TODO 复杂类型注入练习
     ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext-
collection.xml");

     TestCollection textColl = context.getBean("testColl", TestCollection.class);

     System.out.println("testColl = " + textColl);

   }
```

第六章.使用注解

> 使用注解的方式完成IOC

6.1 准备工作

- 创建项目: spring-02-annotation
- 导入jar包: spring-core,spring-context,spring-suppot-context,spring-beans,spring-expression, log4j,commons-logging,本次多加一个:spring-aop
- 引入日志配置文件:log4j.properties
- 实体类: 原项目 Person.java 和 Car.java即可
- 创建配置文件: applicationContext.xml
- 创建测试代码类:AnnotationTest.java

6.2 使用注解

6.2.1 引入Context的约束

参考文件位置:spring-framework-5.1.3.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html

配置:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--较之前多了 xmlns:context -->
<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"> <!-- bean definitions
here -->


</beans>
```

6.2.2 配置注解扫描

在applicationContext.xml中配置:

指定扫描 `包` 下所有类中的注解,扫描包时,会扫描包所有的子孙包.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--较之前多了 xmlns:context -->
<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"> <!-- bean definitions
here -->

     <!--扫描包设置-->
      <context:component-scan base-package="com.itqf.spring.bean">
</context:component-scan>
</beans>
```

6.2.3 使用注解

在Person类的头部添加如下注解

```
/**
  * @Component(person) == <bean name="person" class="com.itqf.spring.bean.Person" />
  */

@Component("person")
public class Person {
    private String name;
    private Integer age;
    private Car car;


    public  Person(){
        System.out.println("无参数构造方法!");
    }

  //getter,setter,toString
}
```

测试:

```
  @Test
  public  void  test1(){
     //TODO 测试注解入门
     ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

     Person person = context.getBean("person", Person.class);

     System.out.println("person = " + person);
 }
```

6.3 其他注解

介绍其他常用注解,测试方式同前

6.3.1 类头部可用的注解

```
@Service("person")              // service层
@Controller("person")   // controller层
@Repository("person")   // dao层
```

6.3.2 类头部可用的注解

指定对象作用域

```
@Scope(scopeName="singleton")
@Scope(scopeName="prototype")
```

6.3.3 注入属性value值

1.设置成员变量上:通过反射给变量赋值

```
@Value("name值")
private String name;
```

@Value("name值") 等同于 @Value(value="name值")

2.加在set方法上:通过set方法赋值

```
@Value("tom")
public void setName(String name)
{
  this.name = name;
}
```

6.3.4 自动装配

1. @Autowired

使用 `@Autowired` 自动装配对象类型的属性: 下面的Person中的Car使用了自动装配

```
//将Car定义成接口
@Component
public interface Car {
     void log();
}

//Baoma实现Car
@Component
public class Baoma implements Car {

    public void log() {
        System.out.println("宝马");
    }
}

//XianDai实现Car
@Component
public class XianDai implements Car {

    public void log() {
        System.out.println("现代");
    }
}
```

装配类:

```
@Scope(scopeName = "prototype")
@Component("person")
public class Person {
    @Value("name值")
    private String name;
    private Integer age;
    @Autowired
    private Car car;  //自动装配 可以选择Car,如果Car是接口,找Car的实现类!
```

注意: 以上操作会出现一个问题,如果Car是接口,且Car只有一个实现类,那么@Autowired会自动将实现类装配给Person的car变量上,但是如果Car是接口,并且有两个以上实现类,那么自动装配就会报错,无法选择由哪个实现类赋值.所以需要配合另一个注释@Qualifier("bean name"), 这个属性可以将@Autowired按类型赋值改成按bean名字赋值.

6.3.5 @Qualifier

- 如果匹配多个类型一致的对象，将无法选择具体注入哪一个对象
- 使用`@Qualifier()`注解告诉spring容器自动装配哪个名称的对象。

```
@Scope(scopeName = "prototype")
@Component("person")
public class Person {
    @Value("name值")
    private String name;
    private Integer age;
    @Autowired
    @Qualifier("baoma")    //指定实现类
    private Car car;  //自动装配 可以选择Car,如果Car是接口,找Car的实现类!
```

6.3.6 @Resource

@Resource 是java的注释,但是Spring框架支持,@Resource指定注入哪个名称的对象

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

```
@Resource("baoma")
private Car car;
```

6.3.7 初始化和销毁方法

初始化和销毁方法等同于配置文件添加的init-method和destroy-method功能,

例:Person类中init方法和destroy方法添加如下注解:

```
@PostConstruct
public  void init(){
  System.out.println("初始化方法");
}

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

第七章. Spring整合JUnit测试

7.1 Spring各个Jar包作用

Spring Aspects：Spring提供的对AspectJ框架的整合 Spring Beans：Spring IOC的基础实现，包含访问配置文件、创建和管理bean等。 Spring Context：在基础IOC功能上提供扩展服务，此外还提供许多企业级服务的支持，有邮件服务、任务调度、JNDI定位，EJB集成、远程访问、缓存以及多种视图层框架的支持。 Spring Context Support：Spring context的扩展支持，用于MVC方面。 Spring Core：Spring的核心工具包 Spring expression：Spring表达式语言 Spring Framework Bom： Spring Instrument：Spring对服务器的代理接口 Spring Instrument Tomcat：Spring对tomcat连接池的集成 Spring JDBC：对JDBC 的简单封装 Spring JMS：为简化jms api的使用而做的简单封装 Spring Messaging： Spring orm：整合第三方的orm实现，如hibernate，ibatis，jdo以及spring 的jpa实现 Spring oxm：Spring对于object/xml映射的支持，可以让JAVA与XML之间来回切换 Spring test：对JUNIT等测试框架的简单

封装 Spring tx：为JDBC、Hibernate、JDO、JPA等提供的一致的声明式和编程式事务管理。 Spring web：包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类。 Spring webmvc：包含SpringMVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、Tiles、Velocity、XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类。 Spring webmvc portlet：Spring MVC的增强

7.2 Spring整合junit

为我们提供了方便的测试方式

1、导包：在spring-02-annotation项目中再加入如下包

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.3.RELEASE</version>
</dependency>
```

2、创建测试类

```
//创建容器
@RunWith(SpringJUnit4ClassRunner.class)
//指定创建容器时使用哪个配置文件
@ContextConfiguration("classpath:applicationContext.xml")
public class RunWithTest {
    //将名为user的对象注入到u变量中
    @Resource(name="person")
    private Person p;
    @Test
    public void testCreatePerson(){
        System.out.println(p);
    }
}
```

第八章 课后作业

```
1. 创建汽车类（Car）和引擎接口（Engine）
2. 创建两个引擎的实现类V6Engine和V8Engine
3. 使用依赖注入的方式显示V6的汽车和V8的汽车奔跑（running）的效果
4. 使用BeanFactory和ApplicationContext分别加载以上的类
5. 使用不同的生命周期创建上面的类并使用JUnit测试不同之处
6. 使用注解的方式再实现一次
```

第九章 面试题

```
1. IOC和DI是什么，它们之间有什么关系
2. Spring加载配置文件的方式的区别（ClassPath和FileSystem）
3. Spring加载bean的两种方式的区别（BeanFactory和ApplicationContext）
4. Spring创建的bean的生命周期（Singleton、Prototype、session、global-session、application）
```