本章学习目标

- Spring 框架简介
- SpringIOC 的概念和作用
- 工厂模式设计一个简单的 IOC 容器
- SpringIOC 的 XML 方式 HelloWorld
- SpringIOC 的 XML 方式创建对象配置细节
- SpringIOC 的 XML 方式依赖注入
- SpringIOC 的注解方式
- Spring 整合 Junit 简化测试类编写

1. Spring 框架简介

Spring框架一个一站式的Java开源框架，能够解决分层开发中各个方面的问题！

2. SpringIOC 的概念和作用

2.1. IoC 是什么？

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在 Java 开发中，Ioc 意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●谁控制谁，控制什么：传统 Java SE 程序设计，我们直接在对象内部通过

new 进行创建对象，是程序主动去创建依赖对象；而 IoC 是有专门一个容器来创建这些对象，即由 Ioc 容器来控制对象的创建；谁控制谁？当然是 IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

2.2. IoC 能做什么？

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实 IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

IoC 很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由 IoC 容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

3. 工厂模式设计一个简单的 IOC 容器

IOC 容器就是一个底层设计思想：工厂模式

3.1. 设计一个 BeanFactory（模拟 IOC 容器）

```
/**
 * 创建对象的工厂（模拟简单 IOC 容器）
 * @author lenovo
 *
 */
public class BeanFactory {


    /**
     * 初始化 beans.properties 文件
     */
    private static Properties props = new Properties();
    static{
        InputStream in =
BeanFactory.class.getResourceAsStream("/beans.properties");
        try {
            props.load(in);
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("加载 beans.properties 文件失败");
        }
    }
```

```
    /**
     * 从工厂获取一个对象
     * @return
     */
    public static Object getBean(String name){
        //根据 name 创建不同的对象
        //1.通过 name 在 properties 文件找到类名称
        String className = props.getProperty(name);
        //2.通过反射构造类的对象
        try {
            return Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

3.2. beans.properties

```
customerDao=cn.sm1234.dao.impl.CustomerDaoImpl2
customerService=cn.sm1234.service.impl.CustomerServiceImpl
```

3.3. CustomerServiceImpl

```
public class CustomerServiceImpl implements CustomerService {
    //1.传统方法，直接 new 对象.（弊端：耦合性太高，修改源代码）
    //private CustomerDao customerDao = new CustomerDaoImpl();
```

```
    //private CustomerDao customerDao = new CustomerDaoImpl2();


    //2.IOC 容器
    private CustomerDao customerDao = (CustomerDao)
BeanFactory.getBean("customerDao");


    @Override
    public void save() {
        customerDao.save();
    }


}
```

3.4. ActionDemo

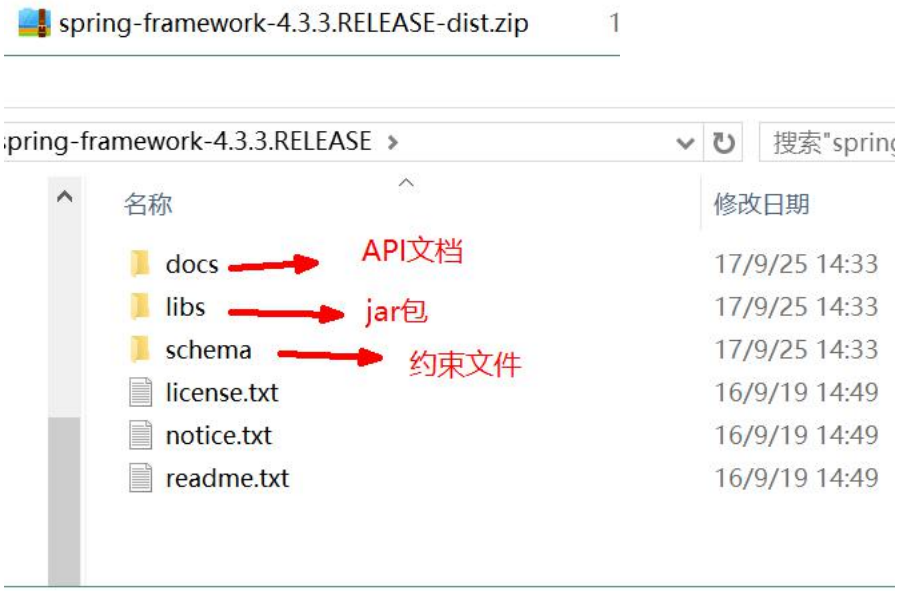
```
public class ActionDemo {


    public static void main(String[] args) {
        //调用业务,直接 new 对象
        //CustomerService customerService = new CustomerServiceImpl();
        //使用 IOC 容器去获取对象
        CustomerService customerService = (CustomerService)
BeanFactory.getBean("customerService");
        customerService.save();
    }
}
```

4. SpringIOC 的 XML 方式

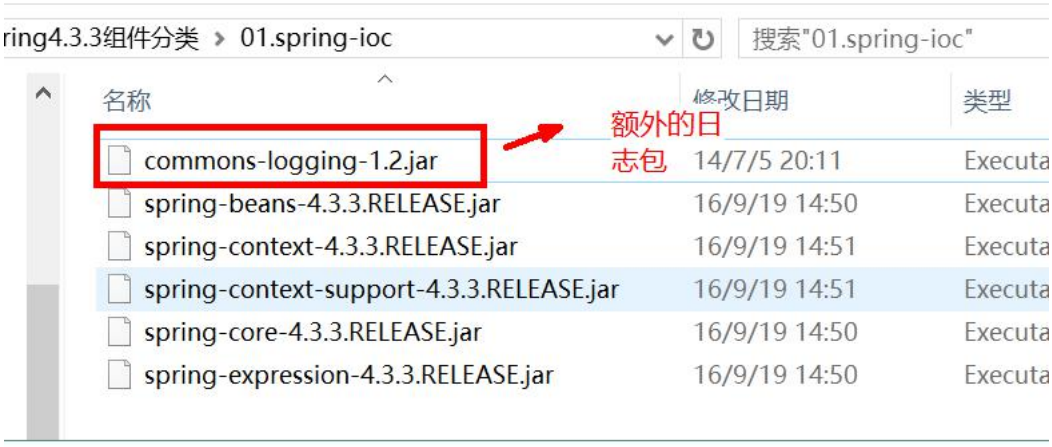
4.1. 创建对象

4.1.1. HelloWorld 程序

4.1.1.1. 下载 Spring 的资源



4.1.1.2. 把必须到的 jar 包导入到项目



4.1.1.3. 编写 Dao 接口和实现类

接口：

```
public interface CustomerDao {


    public void save();

}
```

实现类：

```
public class CustomerDaoImpl implements CustomerDao {


    @Override
    public void save() {
        System.out.println("把客户数据保存到 mysql 数据");
    }


}
```

4.1.1.4. 编写 applicationContext.xml（重点）

```
<?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">


    <!-- 创建 CustomerDaoImpl 对象-->
    <bean id="customerDao"
class="cn.sm1234.dao.impl.CustomerDaoImpl"></bean>
```

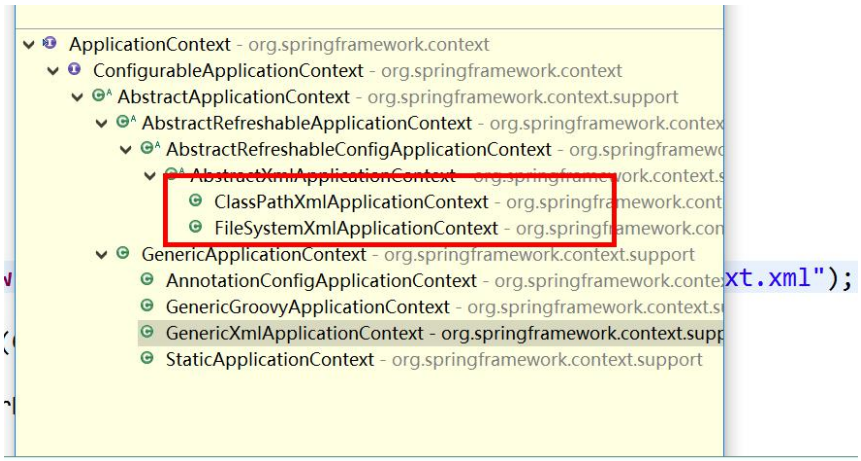
```
</beans>
```

4.1.1.5. 编写测试代码

```
/**
     * Spring 的 IOC 的入门程序
     */
    @Test
    public void test1(){
        //1.初始化 SpringIOC 容器
        ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //2.从 IOC 容器取出对象
        CustomerDao customerDao = (CustomerDao)ac.getBean("customerDao");

        System.out.println(customerDao);
    }
```

4.1.2. ApplicationContext 接口的用法

AppicationContext 接口，用于初始化 SpringIOC 容器。



ClassPathXmlApplicationContext: 使用类路径方式初始化 ioc 容器（推荐）

FileSystemApplicationContext: 使用文件系统的方式初始化 ioc 容器

```
/**
     * 演示 ApplicationContext 的用法
     */
    @Test
    public void test2(){
        //1.ClassPathXmlApplicationContext: 使用类路径方式初始化 ioc 容器
        //ApplicationContext ac = new
ClassPathXmlApplicationContext("cn/sm1234/test/applicationContext.xml")
;


        //2.FileSystemApplicationContext:  使用文件系统的方式初始化 ioc 容器
        /**
         * 绝对路径
         * 相对路径
         */
        //2.1 绝对路径
        //ApplicationContext ac = new
FileSystemXmlApplicationContext("E:\\workspaces\\sm1234_spring\\ch01_02
```

```
_spring_ioc_xml\\src\\applicationContext.xml");
        //2.2 相对路径
        ApplicationContext ac = new
FileSystemXmlApplicationContext("./src/applicationContext.xml");

        CustomerDao customerDao = (CustomerDao)ac.getBean("customerDao");

        System.out.println(customerDao);
    }
```

4.1.3. 单例和多例问题

scope 配置

包括：

singleton： 创建一个单例的对象

prototype： 每次创建一个对象（多例的）

request： 用在 web 项目中，保证一个请求创建一个对象

session： 用在 web 项目中，保证一个会话创建一个对象

```
/**
     * scope 配置
     */
    @Test
    public void test1(){
        ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
        for (int i = 0; i < 5; i++) {
            CustomerDao customerDao =
```

```
(CustomerDao)ac.getBean("customerDao");
            System.out.println(customerDao);
        }
    }
```

scope 配置实际应用场景：

1）action 对象： 必须是多例的

2）service 对象：单例

3）dao 对象：单例

4.1.4. 对象生命周期方法

init-method: 初始化方法

destroy-method: 销毁方法

```
<!-- 创建 CustomerDaoImpl 对象-->
    <bean id="customerDao" class="cn.sm1234.dao.impl.CustomerDaoImpl"
            init-method="init" destroy-method="destroy"></bean>
```

注意：单例的情况时使用！

4.2. 依赖注入

4.2.1. 构造方式参数注入

```
public class CustomerServiceImpl implements CustomerService {

    private CustomerDao customerDao;

    //使用 SpirngIOC 容器的依赖注入
```

```
    //1.构造方法参数注入
    public CustomerServiceImpl(CustomerDao customerDao) {
        super();
        this.customerDao = customerDao;
    }
```

```
    <!-- 创建 CustomerDaoImpl对象-->
    <bean id="customerDao" class="cn.sm1234.dao.impl.CustomerDaoImpl"></bean>

    <bean id="customerService" class="cn.sm1234.service.impl.CustomerServiceImpl">
        <!-- 1.构造方法 -->
        <!--
            index: 参数索引，从0开始
         -->
        <constructor-arg index="0" ref="customerDao"/>
    </bean>
/beans>
```

4.2.2. setter 方法注入（推荐使用）

```
//2.setter 方法注入
    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }
```

```
    <!-- 创建 CustomerDaoImpl对象-->
    <bean id="customerDao" class="cn.sm1234.dao.impl.CustomerDaoImpl"></bean>

    <bean id="customerService" class="cn.sm1234.service.impl.CustomerServiceImpl">
        <!-- 1.构造方法 -->
        <!--
            index: 参数索引，从0开始
         -->
        <!-- <constructor-arg index="0" ref="customerDao"/> -->

        <!-- 2.setter方法注入 -->
        <property name="customerDao" ref="customerDao"/>
    </bean>
```

4.2.3. p 名称空间注入

在 applicationContext.xml 引入 p 名称空间

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

    <!-- 创建 CustomerDaoImpl对象-->
    <bean id="customerDao" class="cn.sm1234.dao.impl.CustomerDaoImpl"></bean>
```

```
<!-- 3.使用 p 名称空间注入 -->
    <bean                                                    id="customerService"
class="cn.sm1234.service.impl.CustomerServiceImpl"
p:customerDao-ref="customerDao"/>
```

注意：p 名称空间简化 setter 方法注入，所以必须要有 setter 方法

p:属性 ： 注入普通的数据（例如 String）

p:属性-ref： 注入 JavaBean 对象（例如 CustomerDao）

4.2.4. spEL 表达式注入

spEL 表达式，spring EL 表达式，是 spring3.0 以后的新特性。

```
<!-- 4.spEL 表达式注入（可以使用在 setter 方法和构造方法上面的） -->
    <bean id="customerService"
class="cn.sm1234.service.impl.CustomerServiceImpl">
        <property name="customerDao" value="#{customerDao}"/>
        <property name="name" value="#{'eric'}"/>
    </bean>
```

4.2.5. 注入不同的数据类型（重点）

普通数据类型： <value>或者 value 属性

JavaBean 对象： <ref> 或者 ref 属性

数组： <array>

List 集合： <list>

Map 集合： <map>

Properties: <props>

```
<!-- 注入不同的数据类型 -->
    <bean id="customerService"
class="cn.sm1234.service.impl.CustomerServiceImpl">
        <!-- 1.普通类型 -->
        <property name="gender">
            <value>jack</value>
        </property>
        <!-- 2.JavaBean 类型 -->
        <property name="customer">
            <ref bean="c1"/>
        </property>
        <!-- 3.数组类型 -->
        <property name="addresses">
            <array>
                <value>北京</value>
                <value>上海</value>
                <value>广州</value>
            </array>
        </property>
        <!-- 4.List 集合 -->
        <property name="customerList">
            <list>
                <ref bean="c1"/>
                <ref bean="c2"/>
```

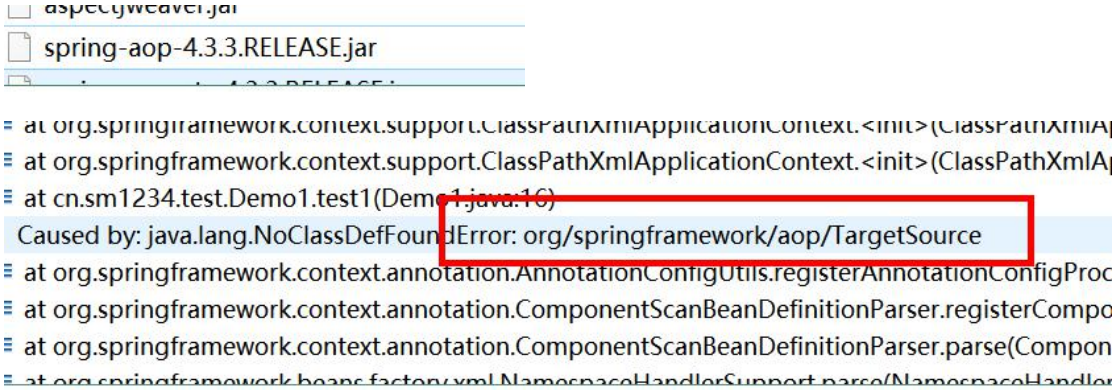
```
            <ref bean="c3"/>
        </list>
    </property>
    <!-- 5.Map 集合 -->
    <property name="customerMap">
        <map>
            <entry key="001" value-ref="c1"/>
            <entry key="002" value-ref="c2"/>
            <entry key="003" value-ref="c3"/>
        </map>
    </property>
    <!-- 6.Properties 类型 -->
    <property name="customerProps">
        <props>
            <prop key="001">rose</prop>
            <prop key="002">eric</prop>
            <prop key="003">jack</prop>
        </props>
    </property>
</bean>
```

5. SpringIOC 的注解方式

5.1. 创建对象

5.1.1. HelloWorld 程序

5.1.1.1. 导入 spring-aop 的包



注意：如果不加上 aop 的包会报错！

5.1.1.2. 编写 Dao 接口和实现类

接口：

```
public interface CustomerDao {


    public void save();

}
```

实现类：

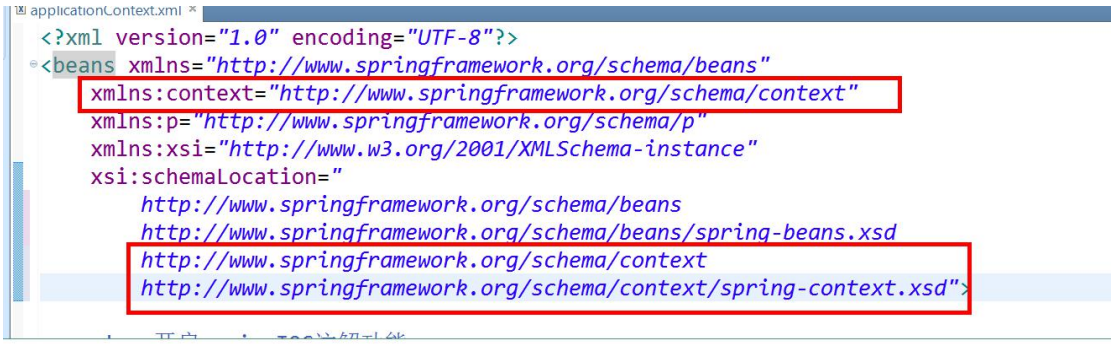
```
public class CustomerDaoImpl implements CustomerDao {


    @Override
    public void save() {
```

```
        System.out.println("把客户数据保存到 mysql 数据");
    }


}
```

5.1.1.3. 编写 applicationContext.xml

开启 springIOC 的注解功能:

注意：要引入 context 的名称空间：




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

<beans xmlns="http://www.springframework.org/schema/beans"

    xmlns:context="http://www.springframework.org/schema/context"

    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

        http://www.springframework.org/schema/context


http://www.springframework.org/schema/context/spring-context.xsd">


    <!-- 开启 springIOC 注解功能 -->
```

```
    <!-- base-package: IOC 注解的类所在包 -->
    <context:component-scan base-package="cn.sm1234"/>
</beans>
```

5.1.1.4. 在需要创建的类顶部加上注解

```
@Component  // 默认名称为类名（首字母小写）：  customerDaoImpl  <bean
id="customerDaoImpl" class="xxx"/>
public class CustomerDaoImpl implements CustomerDao {
```

5.1.1.5. 编写测试代码

```
/**
     * IOC 注解的入门程序
     */
    @Test
    public void test1(){
        ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
        CustomerDao customerDao =
(CustomerDao)ac.getBean("customerDaoImpl");
        System.out.println(customerDao);
    }
```

5.1.2. IOC 相关的注解

@Component ： 创建对象 （普通的 Spring 项目）

@Repository： 和@Component 的作用是一样，创建对象（分层项目，Dao 层）

@Service：和@Component 的作用是一样，创建对象（分层项目，Service 层）

@Controller：和@Component 的作用是一样，创建对象（分层项目，Web 层）

@Scope： 单例和多例（XML 方式里 scope 配置）

```
@Scope("prototype")
```

@PostConstruct: 初始化方法

@PreDestroy：销毁方法

5.2. 依赖注入

5.2.1. @Value

作用： 注入普通数据类型

jdbc.properties

```
url=jdbc:mysql://localhost:3306/ecshop
```

```
<!-- 加载 properties 文件 -->
    <context:property-placeholder location="classpath:jdbc.properties"/>
```

```
@Service(value="customerService")
public class CustomerServiceImpl implements CustomerService {


    //1.注入普通类型
    @Value("eric")
    private String name;
    @Value("${url}")
    private String url;
```

5.2.2. @Autowired

@Autowired: 自动根据类型注入

```
//2.注入 JavaBean
    @Autowired
    /**
     * 1)自动根据类型进入注入，直接赋值给变量（无需提供构造方法或者 setter 方法）
     * 2）如果 Spring 环境中没有任何一个 CustomerDao 的类型，那么会注入失败
     * 3）如果 Spring 环境中出现了多个 CustomerDao 的类型的对象，那么也会注入失
败
     */
    @Qualifier(value="customerDao2")
    private CustomerDao customerDao;
```

5.2.3. @Qualifier

@Qualifier： 指定注入的对象名称

当@Autowired: 出现多个相同类型的对象的时候，可以使用@Qulifier 指定需要注入的对象名称。

5.2.4. @Resource

@Resource： 既能够根据类型自动注入，也可以根据对象名称注入

```
//3.@Resource
    @Resource(name="customerDao2")
    private CustomerDao customerDao;
```

- @Autowired vs @Resource 注解的区别？

1）注入的方式不一样：

@Autowired：只能根据类型注入，如果需要根据名称进行注入需要@Qualfier 注解的配合。而@Resource 既能够根据类型自动注入，也可以根据对象名称注入

2）所属的标准不同

@Autowired 来自于 Spring 框架

@Resource 来自 JavaEE 标准

@Resource 注解比@Autowired 更加标准！

结论：推荐使用@Resource 注解来注入对象！

6. Spring 新注解（零配置注解）

6.1. @Configuration

作用：配置类，用于代替 applicationContext.xml

6.2. @ComponentScan

作用： <context:compnent-scan/>标签

```
/**
 * Spring 配置类（代替 applicationContext.xml 内容）
 * @author lenovo
 *
 */
```

```
@Configurable
@ComponentScan(basePackages={"cn.sm1234"})
public class SpringConfig {


}
```

```
    /**
     * 零配置注解的入门程序
     */
    @Test
    public void test1(){
        ApplicationContext ac = new
AnnotationConfigApplicationContext(SpringConfig.class);
        CustomerDao customerDao = (CustomerDao)ac.getBean("customerDao");
        System.out.println(customerDao);
    }
```

6.3. @PropertySource

作用：代替<context:property-placeholader/>配置，加载 properties 配置文件

```
@Configurable
@ComponentScan(basePackages={"cn.sm1234"})
@PropertySource(value="classpath:jdbc.properties")
public class SpringConfig {


}
```

6.4. @Import

作用：同时导入多个配置类，代替<import/> 配置

6.5. @Bean

作用：把一个对象放入 Spring 的 IOC 容器

注意：必须是用在方法的上面

```
/**
     * 1）可以执行该方法的代码
     * 2）把该方法的返回对象放入 Spring 的 IOC 容器
     * @return
     */
    @Bean(name="custDao")
    public CustomerDao getDao(){
        System.out.println("执行 getDao 方法");
        return new CustomerDaoImpl();
    }
```

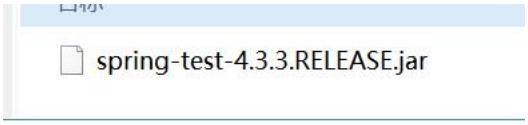
- @Bean vs @Resource 或@Autowired？

@Autowired 或@Resource： 从 SpringIOC 容器中获取一个指定类型或者指定名称的对象

@Bean： 执行某个方法，把方法的返回的对象，放入 Spring 的 IOC 容器。

7. Spring 整合 Junit 简化测试类编写

7.1. 导入 spring-test 包



7.2. 有配置文件的方式

```
/**
 * 演示有配置文件的方式 Spring 整合 Junit 简化测试
 * @author lenovo
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class Demo2 {


    @Resource(name="customerDao1")
    private CustomerDao customerDao;



    @Test
    public void test1(){
        System.out.println(customerDao);
    }
}
```

7.3. 零配置的方式

```
/**
 * 演示零配置方式 Spring 整合 Junit 简化测试
 * @author lenovo
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={SpringConfig.class})
public class Demo2 {


    @Resource(name="customerDao")
    private CustomerDao customerDao;


    @Test
    public void test1(){
        System.out.println(customerDao);
    }
}
```