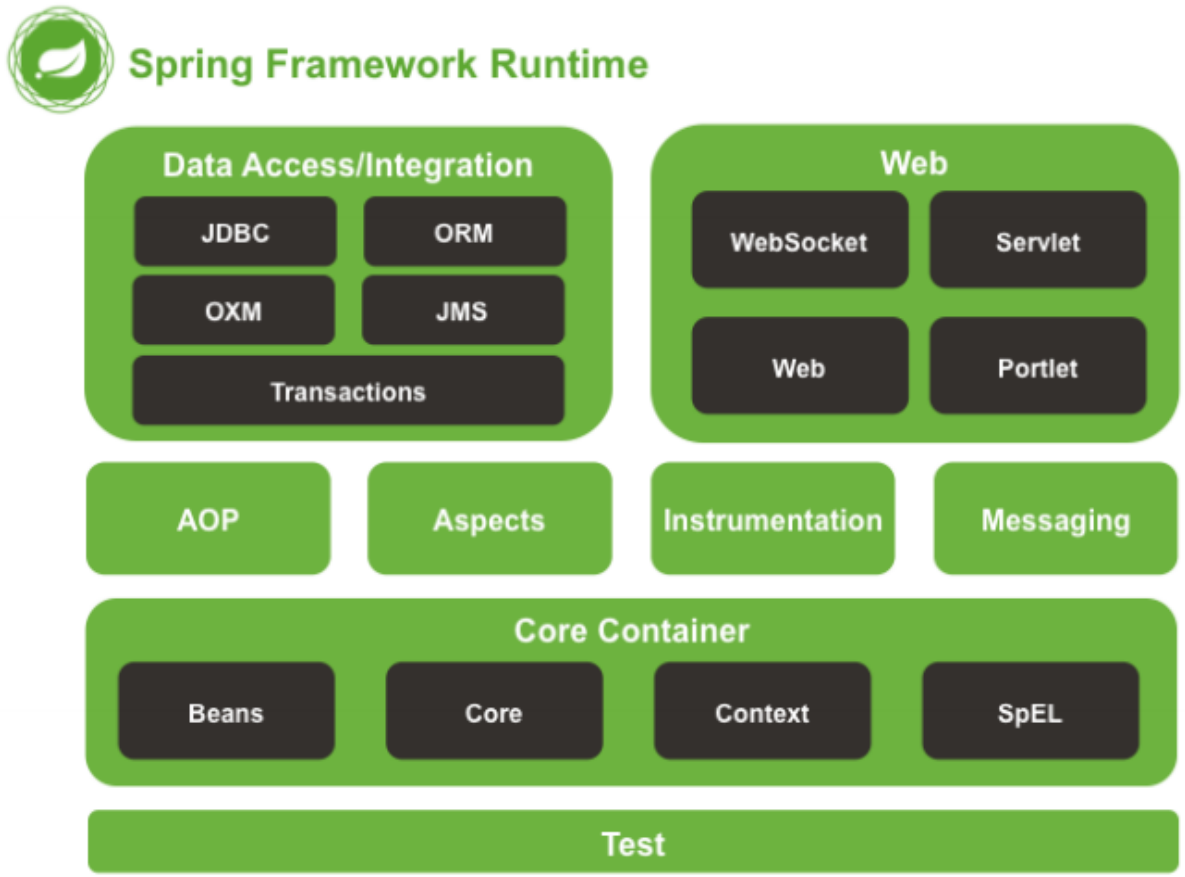
Spring概述

简介

Spring是于2003年兴起的一个轻量级的Java开发框架，它是为了解决企业应用开发的复杂性而创建的。Spring的核心是控制反转（IOC）和面向切面编程（AOP）。简单来说，Spring是一个分层的JavaSE/EE full-stack（一站式）轻量级开源项目。 Spring 的主要作用是为了代码“解耦”，降低代码间的耦合度。 根据功能的不同，可以将一个系统中的代码分为主业务逻辑两类。它们各自具有鲜明的特点：主业务代码间逻辑联系紧密，有具体的专业业务应用场景，复用性相对较低；系统级业务相对功能独立，没有具体的专业业务应用场景，主要是为主业务提供系统及服务，如日志，安全，事务等，复用性强。 Spring根据代码的功能特点，将降低耦合度的方式分为两类，IOC和AOP。 IOC使得主业务在相互调用中，不用维护关系了，即不用再自己创建要使用得对象了。而是由Spring容器统一管理，自动“注入”。 而AOP使得系统级服务得到最大复用，且不用再由程序员手工将系统级服务“混杂”到主业务逻辑中了，而是由Spring容器统一完成“织入”。

Spring体系结构



Spring由20多个模块组成，它们可以分为数据访问/继承（Data Access/Intergration）、Web、面向切面编程（AOP,Aspects）、应用服务器设备管理（Instrumentation）、信息发送（Messaging）、核心容器（Core Container）和测试（Test）。

Spring的特点

> **非侵入式**

所谓非侵入式是指，Spring框架的API不会在业务逻辑出现，即业务逻辑是POJO。由于业务逻辑中没有Spring 的API，所以物业逻辑可以从Spring框架快速地移植到其他框架，即与环境无关。POJO（Plain Old Java Object）:最存粹的类，无需其他引用项目的API。

> **容器**

Spring作为一个容器，可以管理对象的生命周期、对象与对象之间的依赖关系。可以通过配置文件，来定义对象，一以及设置与其他对象的依赖关系。

> IOC

控制反转（Inversion of Control），即创建被调者的实例不是由调用者完成，而是由Spring 容器完成，并注入给调用者。 当应用了IOC，一个对象依赖的其他对象通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。即，不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

> AOP

面向切面编程（Aspect Orient Programming）,是一种编程思想，是面向对象编程OOP得补充。很多框架都实现了AOP编程思想得实现。Spring也提供了面向切面编程得丰富支持，允许通过分离应用得业务逻辑与系统级服务（例如日志和事务管理）进行开发。应用对象只实现它应该做得（完成业务逻辑）仅此而已。它们并不负责其他得系统级关注点，例如日志和事务支持。 我们可以把日志、安全、事务管理等服务理解成一个“切面”，那么以前这些服务一直是直接写在业务逻辑得代码当中。 这由两点不好，首先业务逻辑不纯净；其次这些服务被很多业务与逻辑反复使用，完全可以剥离出来做到服用。那么AOP就是解决这些问题的解决方案，可以把这些服务剥离出来形成一个“切面”，以期复用，然后将“切面”动态的“织入”到业务逻辑中，让业务逻辑能够享受到此“切面”的服务。

IOC：控制反转

前言

控制反转是一个概念，是一种思想。即将传统上由程序代码直接操控的对象调用权交给容器，通过容器来实现对象的装配和管理。控制反转就是对对象控制权的转移，从程序代码本身转到了外部容器。 控制反转的实现由多种方式，现在主要流行的方式由两种：**依赖注入**和**依赖查找**，依赖注入方式应用更为广泛。也是Spring采用的方式。

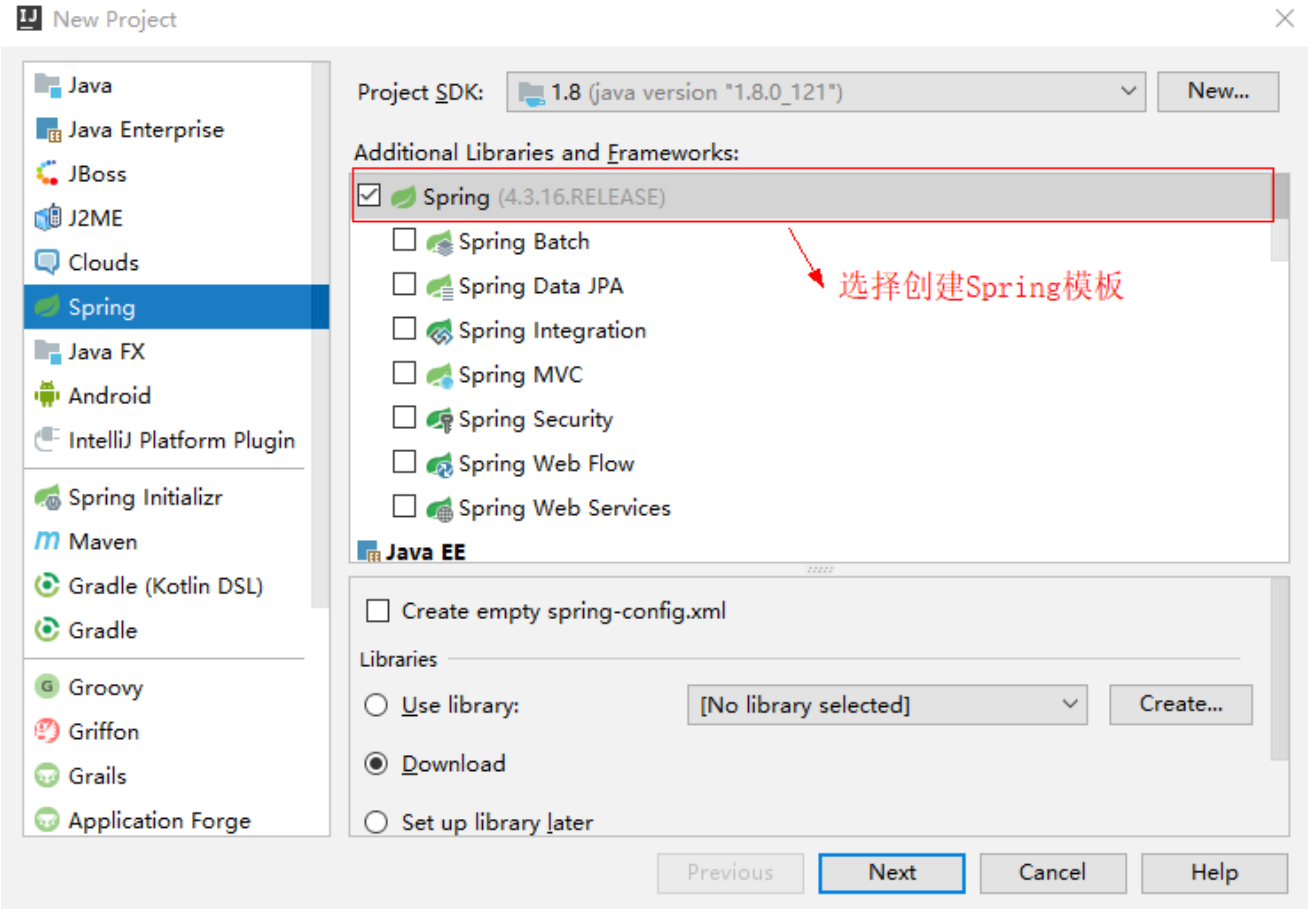
- **依赖查找**：容器提供回调接口和上下文环境给组件，程序代码则需要提供具体的查找方式。比较典型的是依赖于JNDI服务接口的查找。
- **依赖注入**：程序代码不做定位查询，这些工作由容器自行完成。

  依赖注入DI是指程序运行过程中，若需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部容器，由外部容器创建后传递给程序。

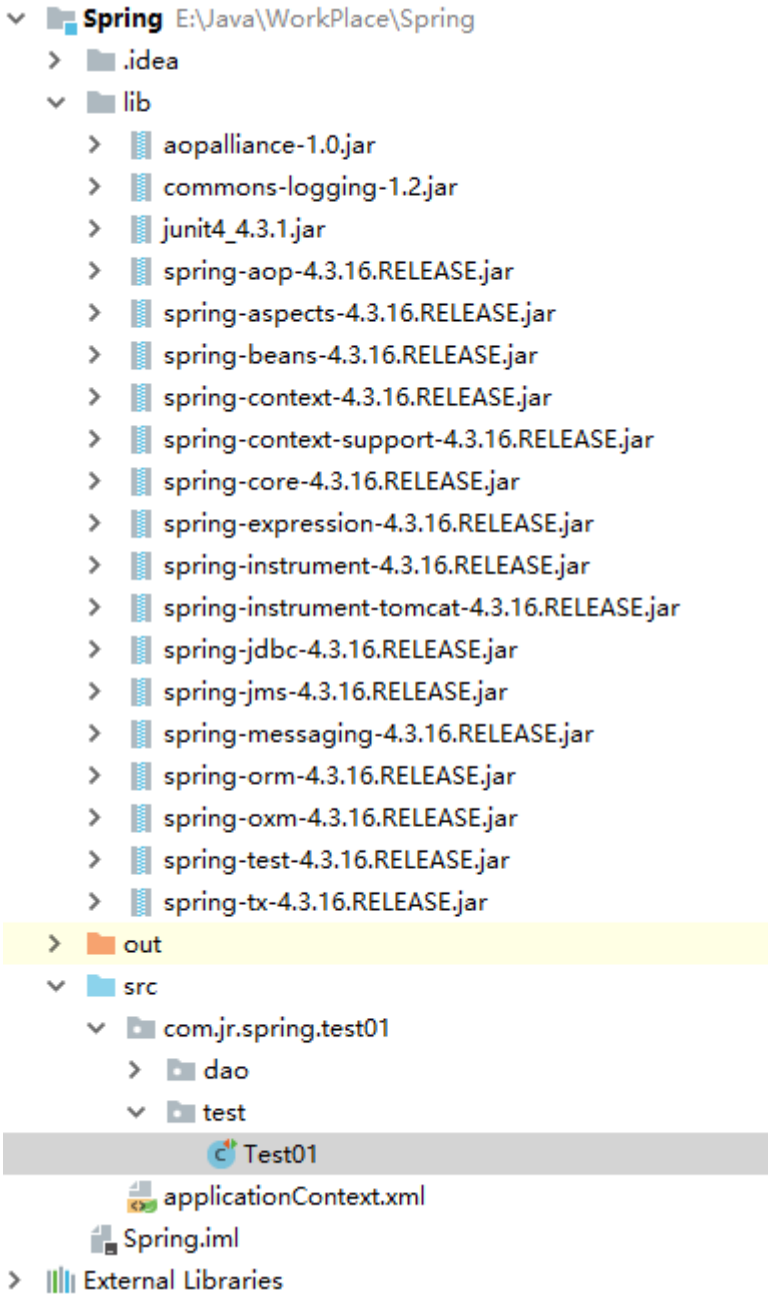
  依赖注入目前时最优秀的解耦方式。依赖注入让Sping和Bean之间以配置文件的方式在一起，而不是以硬解码的方式耦合在一起的。

创建测试项目

1. **使用IDEA创建Spring项目**

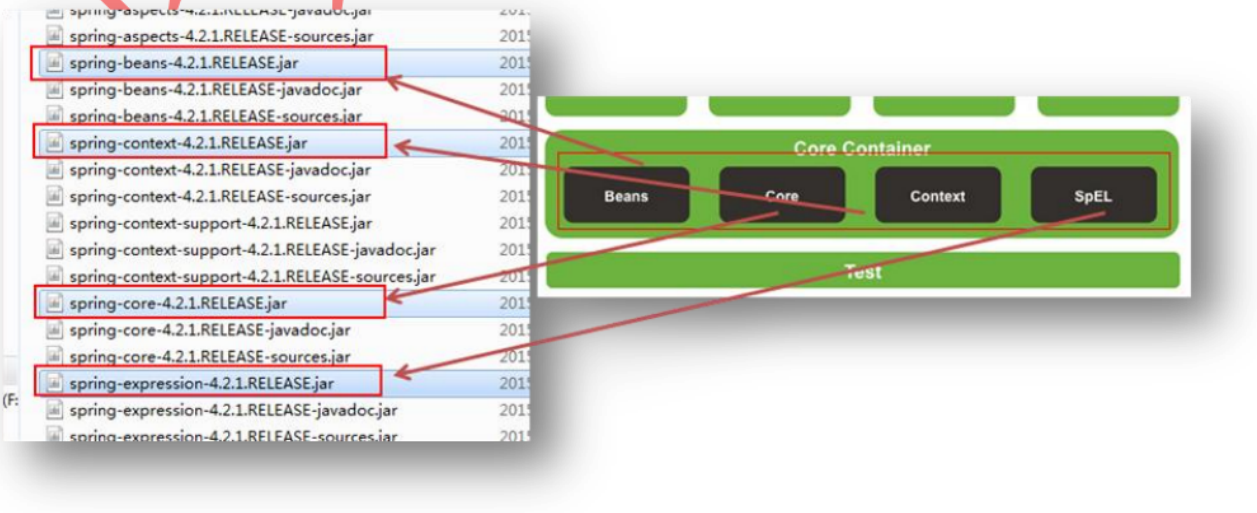


- **IDEA会自动下载Spring框架所需的Jar包**


Spring E:\Java\WorkPlace\Spring
.idea
lib
aopalliance-1.0.jar
commons-logging-1.2.jar
junit4_4.3.1.jar
spring-aop-4.3.16.RELEASE.jar
spring-aspects-4.3.16.RELEASE.jar
spring-beans-4.3.16.RELEASE.jar
spring-context-4.3.16.RELEASE.jar
spring-context-support-4.3.16.RELEASE.jar
spring-core-4.3.16.RELEASE.jar
spring-expression-4.3.16.RELEASE.jar
spring-instrument-4.3.16.RELEASE.jar
spring-instrument-tomcat-4.3.16.RELEASE.jar
spring-jdbc-4.3.16.RELEASE.jar
spring-jms-4.3.16.RELEASE.jar
spring-messaging-4.3.16.RELEASE.jar
spring-orm-4.3.16.RELEASE.jar
spring-oxm-4.3.16.RELEASE.jar
spring-test-4.3.16.RELEASE.jar
spring-tx-4.3.16.RELEASE.jar
out
src
com.jr.spring.test01
dao
test
Test01
applicationContext.xml
Spring.iml
External Libraries

- **其中基本Jar包**

首先，导入 Spring 程序开发的四个基本 jar 包。



其次，导入日志相关的 Jar 包。

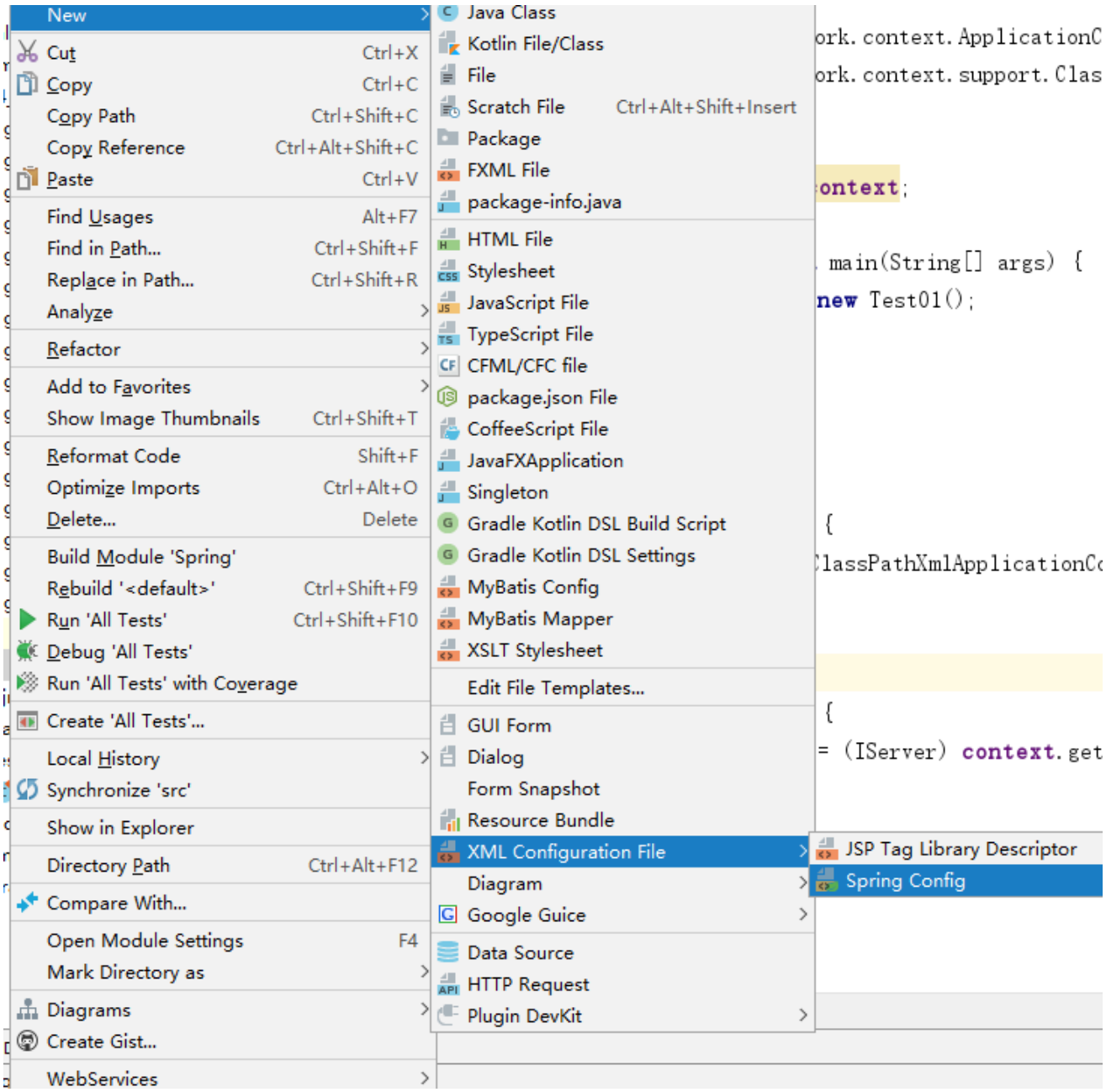
在依赖库 spring-framework-3.0.2.RELEASE-dependencies.zip 解压目录下：\org.apache.commons\com.springsource.org.apache.commons.logging\1.1.1 下的 com.springsource.org.apache.commons.logging-1.1.1.jar 文件。该文件只是日志记录的实现规范，并没有具体的实现。相当于 slf4j.jar 的作用。

这里日志的实现使用 log4j，故还需要 log4j.jar。在依赖库解压目录下：\org.apache.log4j\com.springsource.org.apache.log4j\1.2.15 中的 com.springsource.org.apache.log4j-1.2.15.jar

最后，导入 JUnit 测试 Jar 包 junit-4.9.jar。

Spring 基本编程，共需 7 个 Jar 包即可。

2. **创建Spring配置文件**



3. **定义接口和实现类**

- 接口

```
public interface IServer {
    void print();
}
```

- 实现类

```java
public class IServerImp implements IServer {
    @Override
    public void print() {
        System.out.println("HelloWorld");
    }
}
```

4. 定义测试类

```java
public class Test01 {
    ApplicationContext context;

    @Before
    public void init() {
        context = new ClassPathXmlApplicationContext("applicationContext.xml");
    }

    @Test
    public void test() {
        IServer iServer = (IServer) context.getBean("server");
        iServer.print();
    }
}
```

5. 运行结果

```
E:\Java\jdk1.8.0_121\bin\java ...
六月 08, 2018 9:01:31 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76ccd017: startup date [Fri Jun 08 21:01:31 CST 2018]; root of context hierarchy
六月 08, 2018 9:01:31 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
HelloWorld

Process finished with exit code 0
```

配置文件解析

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="server" class="com.jr.spring.test01.dao.imp.IServerImp"></bean>
</beans>
```

id与name的区别

一般情况下，命名<bean/>使用id属性，而不使用name属性。在没有id属性的情况下，name属性与id属性的作用是相同的，当<bean/>中含有一些特殊字符时，就需要使用那么属性了。 id的命名需要满足XML对id属性命名规范：必须以字母开头名可以包含字母、数字、下划线、连字符、句号、冒号。 name属性值则可以包含各种符号。

配置文件存放路径问题

根据获取容器实例（ApplicationContext）的对象不同，配置文件可以存放到随意路径。

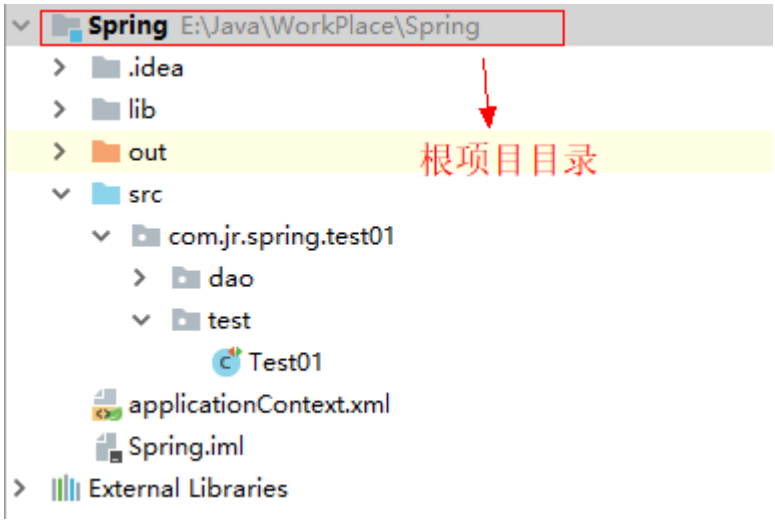
> 存放到源代码根目录

若存放到源代码根目录，那么创建容器实例对象必须为**ClassPathXmlApplictionContext**

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

该对象会在源代码根目录下查找指定目录的配置文件资源

> 存放到本地目录中



若存放到项目根目录，那么创建容器实例对象必须为**FileSystemXmlApplictionContext**

```
ApplicationContext context = new
javaFileSystemtXmlApplicationContext("applicationContext.xml");
```

该对象会从项目根目录下或磁盘空间下查找配置文件资源

BeanFactory接口容器

BeanFactory接口对象也可以作为Spring容器出现。BeanFactory接口是ApplicationContxt接口的父类。 其中XmlBeanFactory作为其中的实现类之一。

1. 创建XmlBeanFactory实例

```
public class Test02 {
    BeanFactory beanFactory;

    @Before
    public void init() {
        beanFactory =new
 XmlBeanFactory(newClassPathResource("applicationContext.xml"));
    }

    @Test
    public void test() {
        IServer iServer = (IServer) beanFactory.getBean("server");
        iServer.print();
    }
}
```

而Spring配置文件以资源Resouce的形式出现XmlBeanFactory类的构造器参数中。 Resouce是一个接口，其具有两个实现类：

- ClassPathResource：指定类路径下的资源文件
- FileSystemResource：指定项目根路径或本地磁盘路径下的资源文件

两个接口容器的区别

> **区别**

BeanFactory容器，对容器中对象的装配与加载采用延迟加载策略，即在第一次调用getBean()时，才真正装配该对象 而ApplicationContext容器，在容器初始化时就已经装配该对象。

Bean的装配，即Bean对象的创建

> **优势**

- **BeanFactory**：因为Bean对象是在调用者第一次调用getBean()方法时才装配Bean，所以BeanFactory相对不占用资源（内存和CUP），但响应速度慢。
- **ApplicationContext**：因为Bean对象在容器初始化时就已经装配了，所以使用ApplicationContext容器对资源的占用比较大，但响应速度快。

Bean的装配

> 默认的装配方式

代码举例test01

接口

```
public interface IServer {
    void print();
}
```

实现类

```
public class IServerImp implements IServer {
    public IServerImp() {
        System.out.println("实例化IServerImp");
    }

    @Override
    public void print() {
        System.out.println("HelloWorld");
    }

}
```

配置文件

```
<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="server" class="com.jr.spring.test01.dao.imp.IServerImp"></bean>

</beans>
```

测试类

```
public class Test01 {
    ApplicationContext context;

    @Before
    public void init() {
        context = new
ClassPathXmlApplicationContext("com/jr/spring/test01/applicationContext.xml");
    }

    @Test
    public void test() {
        IServer iServer = (IServer) context.getBean("server");
        iServer.print();
    }
}
```

代码通过getBean()方法从容器获取指定的bean实例，容器首先会调用Bean类的无参构造器，创造空值的实例对象。

> Spring的工厂Bean

- **动态**

代码举例test02

Spring对于使用动态工厂来创建的Bean，有专门的属性定义。factory-bean指定相应的工厂Bean，由factory-method指定创建所用方法。此时配置文件中至少有两个Bean的指定：工厂类的Bean，与工厂类所要创建的目标Bean。而测试类中不需要获取工厂Bean对象了，可以直接获取目标Bean对象。实现测试类与工厂类间的解耦。

工厂类

```
public class ServerFactory {

    public IServer getServer() {
        return new IServerImp();
    }
}
```

配置文件

```
<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="serverFactory" class="com.jr.spring.test02.dao.ServerFactory"></bean>
<bean id="serverDynamic" factory-bean="serverFactory" factory-method="getServer">
</bean>

</beans>
```

- **静态**

代码举例test02

使用工厂模式中的静态工厂来创建实例Bean。 此时需要注意，静态工厂无需工厂实例，所以不需要定义静态工厂<bean/>。 而对于工厂所要创建的Bean，其不是由自己的类对象创建的，所以无需指定自己的类对象。但其是由工厂类创建的，所以需要指定所用工厂类。股class属性指定的是工厂类而非自己的类对象。当然，还需要通常factory-method属性指定工厂方法。

工厂类

```
public class ServerStaticFactory {

    public static IServer getServer() {
        return new IServerImp();
    }
}
```

配置文件

```
<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">
    <!--静态工厂-->
    <bean id="serverStatic" class="com.jr.spring.test02.dao.ServerStaticFactory"
factory-method="getServer"></bean>

</beans>
```

> 容器中Bean的作用域

当通过Spring容器创建一个Bean实例时，不仅可以完成Bean的实例化，还可以通过**scope**属性，为Bean指定特定的作用域。Spring支持5种作用域。

1. **singleton**：单例模式。即每次使用getBean()方法获取的同一个<bean/>的实例都是一个新的实例。
2. **prototype**：原型模式。即每次使用getBean()方法获取的同一个<bean/>的实例都是一个新的实例。
3. **request**：对于每次的HTTP请求，都将会产生一个不同的Bean实例。

4. **session**：对于每个不同的HttpSession，都将会产生一个不同的Bean实例。

注意：

1）.对于scope的值request，session与globle session，只有Web应用中使用Spring时，该作用域才有效。

2）.对于scope为singleton的单例模式，该Bean是在容器被创建时即被装配好了。

3）.对于scope为protoype的原型模式，Bean实例时在代码使用Bean实例时才进行装配的。

> Bean后处理器

Bean后处理器是一种特殊的Bean，容器中所有的Bean在初始化时，均会自动执行该类的两个方法。由于该Bean是由其他Bean自动调用执行，而不是程序员手工调用，故此Bean无需id属性或name属性。 需要做的是，在Bean后处理器类方法中，只要对Bean类与Bean类中的方法进行判断就可实现指定的Bean的指定方法进行功能扩展和增强。方法返回的Bean对象，即是增强过的对象。

举例test03

程序中由一个业务接口，其有两个业务方法，some（）与other（）。有两个Bean，StudentServiceImpl()和TeacherServiceImpl()，均实现了IService接口。

要求：对StudentServiceImpl的some方法进行增强，输出其开始执行时间与执行结束时间。

接口

```
public interface IServer {
    void some();

    void other();
}
```

实现类

```
public class StudentServiceImp implements IServer {
    @Override
    public void some() {
        System.out.println("StudentServiceImp:some()");
    }

    @Override
    public void other() {
        System.out.println("StudentServiceImp:other()");
    }
}
```

Bean后处理器类

```
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
```

```
    public Object postProcessBeforeInitialization(Object bean, String id) throws
BeansException {
        if (bean instanceof StudentServiceImp) {
            //通过动态代理增强Bean方法
            IServer studentService = (IServer)
Proxy.newProxyInstance(bean.getClass().getClassLoader(),
bean.getClass().getInterfaces(), new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                    if (method.getName().equals("some")) {
                        System.out.println("Bean后处理器通过动态代理增强的方法:当前执行方法前的
时间" + System.currentTimeMillis());
                        Object object = method.invoke(bean, args);
                        System.out.println("Bean后处理器通过动态代理增强的方法:当前执行方法后的
时间" + System.currentTimeMillis());
                        return object;
                    }
                    return method.invoke(bean, args);
                }
            });
            return studentService;
        }
        return null;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String id) throws
BeansException {
        return bean;
    }
}
```

代码中需要定义Bean后处理器类。该类就是实现了BeanPostProcessor的类。该接口中包含两个方法，分别在目标Bean**初始化之前和之后**执行，它们的返回值为功能被扩展后的Bean对象。

配置文件

```
<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">
    <!--定义Bean-->
    <bean id="StudentServer" class="com.jr.spring.test03.dao.imp.StudentServiceImp">
</bean>
    <bean id="TeacherServer" class="com.jr.spring.test03.dao.imp.TeacherServiceImp">
</bean>
    <!--定义Bean后处理器-->
    <bean class="com.jr.spring.test03.dao.MyBeanPostProcessor"></bean>
</beans>
```

Bean初始化完毕有一个标志：在生命周期中的一个方法会被执行（下面会讲到Bean的生命周期），即当该方法被执行，表示该bean被初始化完毕。 所以Bean后处理器中两个方法的执行，是在这个方法之前和之后执行。

> 定制Bean的生命始末

可以为Bean定制初始化后的生命行为，也可以为Bean定制销毁前的生命行为

举例test04

实现类

```
public class StudentServiceImp implements IServer {
    @Override
    public void some() {
        System.out.println("StudentServiceImp:some()");
    }

    @Override
    public String other() {
        System.out.println("StudentServiceImp:other()");
        return null;
    }

    public void init() {
        System.out.println("init-method");
    }

    public void destory() {
        System.out.println("destory-method");
    }
}
```

配置文件

```
<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">
    <!--定义Bean-->
    <bean id="StudentServer" class="com.jr.spring.test04.dao.imp.StudentServiceImp"
init-method="init"
          destroy-method="destory"></bean>
</beans>
```

若要看到Bean的destory-method的执行。需要满足两个条件

1. Bean为singleton，即单例
2. 要确保容器关闭，接口ApplicationContext没有close()方法，但其实现类有，所以，可以将ApplicationContext强转为其实现类对象，或者直接创建就是实现类对象

> Bean的生命周期

Bean实例从创建到最后销毁，需要经常很多过程，执行很多生命周期方法。

举例test05

1. 调用无参构造器，创建实例对象。
2. 调用参数的setter，为属性注入值。
3. 若Bean实现了BeanNameAware接口，则会执行接口方法setBeanName(String beanId)，使Bean类可以获取在其在容器的id名称。
4. 若Bean实现了BeanFactoryAware接口，则执行接口方法的setBeanFactory（BeanFactory factory），使Bean类可以获取BeanFactory对象。
5. 若定义并注册了Bean后处理器BeanPostProcessor，则执行接口的postProcessBeforInitialzation()。
6. 若Bean实现了InitalizingBean接口，则执行接口方法的afterPropertiesSet()。该方法在Bean的**所有属性的set方法执行完毕后执行，使Bean初始化结束的标志，即Bean实例化结束。**
7. 若设置了init-method方法，则执行其中定义的方法。
8. 若定义并注册了Bean后处理器BeanPostProcessor，则执行接口方法的postProcessAfterInitialzation()。
9. 执行业务方法。
10. 若Bean实现了DisposableBean接口，则执行接口方法destory()。
11. 若设置了destory-method方法。则执行其中定义的方法。

依赖注入

基于XML的DL

Bean实例在调用无参构造器创建了空值对象后，就要对Bean对象的属性进行初始化。初始化是容器自动完成的，称为注入。根据注入方式的不同，常用的有两种：**设置注入**、**构造注入。** 还有另外一种，实现特定接口注入。由于这种方式采用侵入式编程，污染了代码，所以几乎不用。

设值注入

设置注入是指，通过setter方法传入被调用者的实例。这种注入方式简单、直观、因而在Spring的依赖注入中大量使用。

举例test06

实体类

```
public class Student {
    private String name;
    private String age;

    public String getName() {
        return name;
    }

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

    public String getAge() {
        return age;
    }
```

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

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

实现类

```
public class StudentServicelmp implements IServer {
    private String name;
    private int age;
    private Student student;

    public Student getStudent() {
        return student;
    }

    public void setStudent(Student student) {
        this.student = student;
    }

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

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

配置文件

```xml
<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">
    <!--定义Bean-->
    <bean id="Student" class="com.jr.spring.test06.dao.bean.Student">
        <property name="name" value="SB"></property>
        <property name="age" value="18"></property>
    </bean>

    <bean id="StudentServer" class="com.jr.spring.test06.dao.imp.StudentServiceImp">
        <property name="name" value="JR"></property>
        <property name="age" value="18"></property>
        <property name="student" ref="Student"></property>
    </bean>
</beans>
```

构造注入

构造注入是指，在构造调用者实例的同时，完成被调用者的实例化。即，使用构造器设置依赖关系。

举例test07

实体类

```java
public class School {
    private String name;
    private String adress;

    public School(String name, String adress) {
        this.name = name;
        this.adress = adress;
    }

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

实现类

```java
public class StudentServiceImp implements IServer {
    private String name;
    private int age;

    private School school;
    public StudentServiceImp(String name, int age, School school) {
```

```
        this.name = name;
        this.age = age;
        this.school = school;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public School getSchool() {
        return school;
    }
}
```

配置文件

```
<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">
    <!--定义Bean-->
    <bean id="school" class="com.jr.spring.test07.dao.bean.School">
        <constructor-arg index="0" value="海丰中学"/>
        <constructor-arg index="1" value="海城"/>
    </bean>

    <bean id="StudentServer" class="com.jr.spring.test07.dao.imp.StudentServiceImp">
       <constructor-arg index="0" value="SB"></constructor-arg>
        <constructor-arg index="1" value="18"></constructor-arg>
        <constructor-arg index="2" ref="school"
type="com.jr.spring.test07.dao.bean.School"></constructor-arg>
    </bean>
</beans>
```

集合类型注入

举例test08

实体类

```
public class Student {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
```

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

    public int getAge() {
        return age;
    }

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

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

```
public class School {
    private String[] dateArray;
    private List<String> dataList;
    private Set<String> dataSet;
    private Map<String, Object> dataMap;
    private Properties dataProperties;

    public String[] getDateArray() {
        return dateArray;
    }

    public void setDateArray(String[] dateArray) {
        this.dateArray = dateArray;
    }

    public List<String> getDataList() {
        return dataList;
    }

    public void setDataList(List<String> dataList) {
        this.dataList = dataList;
    }

    public Set<String> getDataSet() {
        return dataSet;
    }

    public void setDataSet(Set<String> dataSet) {
        this.dataSet = dataSet;
    }
```

```
    public Map<String, Object> getDataMap() {
        return dataMap;
    }

    public void setDataMap(Map<String, Object> dataMap) {
        this.dataMap = dataMap;
    }

    public Properties getDataProperties() {
        return dataProperties;
    }

    public void setDataProperties(Properties dataProperties) {
        this.dataProperties = dataProperties;
    }
}
```

配置文件

```
<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">
    <!--定义Bean-->
   <bean id="student" class="com.jr.spring.test08.dao.bean.Student">
       <property name="age" value="18"></property>
       <property name="name" value="JR"></property>
   </bean>

    <bean id="school" class="com.jr.spring.test08.dao.bean.School">
        <!--数值类型注入-->
        <property name="dateArray">
            <array>
                <value>SB</value>
                <value>18</value>
            </array>
        </property>

        <!--List集合类型注入-->
        <property name="dataList">
            <list>
                <value>SB</value>
                <value>18</value>
            </list>
        </property>

        <!--Set集合类型注入-->
        <property name="dataSet">
            <set>
                <value>SB</value>
                <value>18</value>
            </set>
        </property>
```

```
        <!--Map集合类型注入-->
        <property name="dataMap">
            <map>
                <entry key="name" value="SB"></entry>
                <entry key="student" value-ref="student"></entry>
            </map>
        </property>

        <!--Properties类型注入-->
        <property name="dataProperties">
            <props>
                <prop key="name">SB</prop>
                <prop key="age">18</prop>
            </props>
        </property>
    </bean>

</beans>
```

在<property/>标签下，根据属性类型创建不同类型的嵌入类型标签。

- 数组类型： <array/>
- List类型：<List/>
- Set类型：<Set/>
- Map类型：<Map/>
- Properties类型：<props/>

  而后在根据每个元素的类型设置不同类型名。

  如果是普通类型用value，自定义类型用ref，若继续嵌套集合类型，可在创建嵌套的类型标签。

抽象继承注入

如果多个Bean中相同成员变量拥有相同的值，那么我们可以将这些共同的属性值在放在统一一个Bean中，然后其他Bean继承这个Bean，进而拥有了该Bean中了属性值。

实体类

```
public class Student {
    private String name;
    private int age;
    private Properties properties;

    public String getName() {
        return name;
    }

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

    public int getAge() {
```

```
        return age;
    }

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

    public Properties getProperties() {
        return properties;
    }

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

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

配置文件

```
<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">

    <!--抽象Bean01-->
    <bean id="studentAbstract01" class="com.jr.spring.test09.dao.Student"
abstract="true">
        <property name="name" value="SB"></property>
        <property name="age" value="18"></property>
    </bean>

    <!--抽象Bean02继承抽象Bean01-->
    <bean id="studentAbstract02" class="com.jr.spring.test09.dao.Student"
abstract="true" parent="studentAbstract01">
        <property name="properties">
            <props>
                <prop key="name">SB</prop>
                <prop key="age">18</prop>
                <prop key="shcool">广州科技职业学院</prop>
            </props>
        </property>
    </bean>

    <!--实体Bean继承抽象Bean02-->
```

```
    <bean id="student" class="com.jr.spring.test09.dao.Student"
parent="studentAbstract02">
        <property name="properties">
            <props merge="true">
                <prop key="name">JR</prop>
                <prop key="shcool">广东理工职业学院</prop>
            </props>
        </property>
    </bean>
</beans>
```

继承注入两个bean类可以为相同类型。也可不同，但不同的话，只能注入相同的属性，也就是属性数据类型和成员变量相同的属性。

对域属性（自定义对象属性）的注入

举例test10

- **对域属性的自动注入**

  对于域属性的注入。也不可再配置文件中显示的注入。可以通过为<bean/>标签设置autowire属性值，为域属性进行隐式自动注入。根据自动注入判断标准的不同，可以分为两种。

  1. **byName**：根据名称自动注入

  当配置文件中被调用者Bean的id值与代码中调用者Bean类的属性名相同时，可使用byName方式，让容器自动将被调用者Bean注入给调用者Bean，容器是通过**调用者的Bean类的属性名**与配置文件的**被调用这Bean的id**进行比较而实现自动注入的。

  实体类

```
/**
*调用者实体类
*/
public class Student {
    private String name;
    private int age;
    private Course course;//调用者的Bean类的属性名

    public Course getCourse() {
        return course;
    }

    public void setCourse(Course course) {
        this.course = course;
    }

    public String getName() {
        return name;
    }

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

```
    }

    public int getAge() {
        return age;
    }

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

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

```
/**
*被调用者实体类
*/
public class Course {
    private String name;
    private int number;

    public String getName() {
        return name;
    }

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

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

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

配置文件

```
<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">

    <!--被调用者bean-->
    <bean id="course" class="com.jr.spring.test10.dao.Course">
        <property name="name" value="海丰中学"></property>
        <property name="number" value="18"></property>
    </bean>
    <!--调用者bean-->
    <bean id="student" class="com.jr.spring.test10.dao.Student" autowire="byName">
        <property name="name" value="JR"></property>
        <property name="age" value="18"></property>
    </bean>
</beans>
```

被调用者的bean的id与调用者bean类中的属性名一致时，通过byName会自动注入该域对象。

2. **byType**：根据类型自动注入

   使用byType方式自动注入，要求配置文件中被调用者bean的class属性指定的类，要与代码中调用者Bean类的某域属性类型同源。即要么相同，要么有is-a关系（子类或者实现类）。但这样的同源的被调用Bean只能有一个。，多一个，容器就不知道匹配那样一个了。

调用者Bean实体类

```
/**
*调用者实体类
*/
public class Student {
    private String name;
    private int age;
    private Course course;//调用者的Bean类的属性名

    public Course getCourse() {
        return course;
    }

    public void setCourse(Course course) {
        this.course = course;
    }

    public String getName() {
        return name;
    }

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

```
    public int getAge() {
        return age;
    }

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

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

配置文件

```
<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">

    <!--被调用者bean-->
    <bean id="myCourse" class="com.jr.spring.test10.dao.Course">
        <property name="name" value="海丰中学"></property>
        <property name="number" value="18"></property>
    </bean>
    <!--调用者bean-->
    <bean id="student" class="com.jr.spring.test10.dao.Student" autowire="byType">
        <property name="name" value="JR"></property>
        <property name="age" value="18"></property>
    </bean>
</beans>
```

调用者bean实体类中的域属性名与定义的被调用者bean的id就算不一样。通过byType也可以自动注入

使用SPEL注入

SPEL：Spring Expression Language，即Spring EL表达式语言。即，在Spring配置文件中为Bean的属性诸如之时，可直接使用SPEL表达式计算的结果。用法：<bean id="abc" value="#{...}"/>。

举例test11

- 动态注入静态方法返回值

```
#{T{类路径名}.静态方法}
```

- 动态注入其他bean的属性值

```
#{bean的id.成员变量名}
```

- 动态注入其他bean的方法返回值

```
#{bean的id.方法名}
```

实体类

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public int calculateAge() {
        return age > 15 ? 15 : age;
    }

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

```
public class Student {
    private String name;
    private int age;
    private String courseName;

    public String getCourseName() {
        return courseName;
    }
```

```
    public void setCourseName(String courseName) {
        this.courseName = courseName;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

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

配置文件

```
<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">

    <!--被调用者bean-->
    <bean id="person" class="com.jr.spring.test11.dao.Person">
        <property name="name" value="海丰中学"></property>
        <!--使用SPEL动态注入类静态方法返回值-->
        <property name="age" value="#{T(java.lang.Math).random()*30}"></property>
    </bean>
    <!--调用者bean-->
    <bean id="student" class="com.jr.spring.test11.dao.Student" autowire="byName">
        <property name="name" value="JR"></property>
        <!--使用SPEL动态注入其他bean的属性值-->
        <property name="courseName" value="#{person.name}"></property>
        <!--使用SPEL动态注入其他bean的方法返回值-->
        <property name="age" value="#{person.calculateAge()}"></property>
    </bean>
</beans>
```

注意！！！使用SPEL动态注入其他Bean对象属性值，是通过调用该成员变量的getter方法获取返回值。所以被调用者Bean必须为该成员变量设置getter方法。

使用内部Bean注入

若不希望代码直接访问某个bean，即，在代码中通过getBean方法获取该Bean实例，则可将该Bean的定义放在调用者bean定义的内部。

举例test12

```
<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="student" class="com.jr.spring.test12.dao.Student">
        <property name="name" value="JR"></property>
        <property name="age" value="18"></property>
        <property name="course">
            <!--内部定义bean注入-->
            <bean class="com.jr.spring.test12.dao.Course">
                <property name="name" value="红城中学"></property>
                <property name="number" value="6"></property>
            </bean>
        </property>
    </bean>
</beans>
```

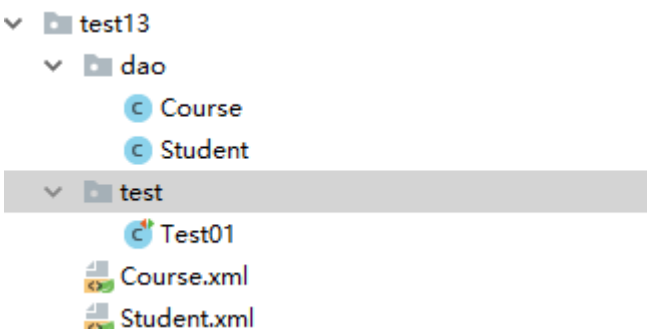
为应用指定多个Spring配值文件

在实际应用例，随着应用规模的增加，系统中的Bean数量也大量增加，导致配置文件变得非常庞大、臃肿。为了避免这种情况的产生，提高配置文件的可读性与维护性，可以将Spring配置文件分解成多个配置文件。

> 平等关系的配置文件

将配置文件分解为地位平等的多个配置文件。

举例test13



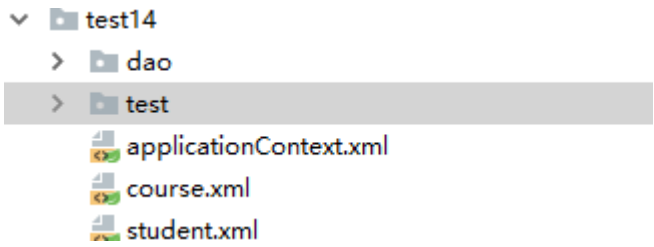
将所有配置文件的路径定义为一个String数组，将其作为容器初始化参数出现。

```
    @Before
    public void init() {
     String [] path=new String[]
{"com/jr/spring/test13/course.xml","com/jr/spring/test13/student.xml"};
     context = new ClassPathXmlApplicationContext(path);
    }
```

> 包含关系的配置文件

各配置文件中有一个总文件，总配置文件将各其它子文件通过<import/>引入。在Java代码中只需要使用总配置文件对容器进行初始化即可。

举例test14



配置文件

```
<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="course" class="com.jr.spring.test14.dao.Course">
        <property name="name" value="海丰中学"></property>
        <property name="number" value="18"></property>
    </bean>

</beans>
```

```
<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">
    <!--包含course配置文件-->
    <import resource="course.xml"></import>
    <bean id="student" class="com.jr.spring.test14.dao.Student">
        <property name="name" value="JR"></property>
        <property name="age" value="18"></property>
        <property name="course" ref="course"><!--注入course配置文件中定义的bean-->
        </property>
    </bean>
</beans>
```

引用上一个配置文件并使用定义id为course的bean

```
<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">
    <!--包含course, student配置文件-->
    <import resource="course.xml"></import>
    <import resource="student.xml"></import>
</beans>
```

将所有配置文件汇总在同一个配置文件

测试代码

```
@Before
public void init() {
    context = new
ClassPathXmlApplicationContext("com/jr/spring/test14/applicationContext.xml");
}
```

初始化加载配置文件的时候只需要载入汇总的配置文件。

在导入其他配置文件时，也可以使用通配符。但要求父配置文件名不能满足*所能匹配的格式，否则将会出现循环递归包含。

基于注解的DL

对于DL使用注解，将不再需要在Spring配置文件中声明Bean实例。

注解的后台实现用到了AOP编程，故需要用到AOP的Jar包。

使用注解步骤

1. **导入AOP的Jar包** spring-aop-4.3.16.RELEASE.jar
2. **XML配置文件添加新的约束**

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

3. **在配置文件中配置组建扫描器，用于在指定的基本包中扫描注解**

   ```
   <context:component-scan base-package="包路径"/>
   ```

注解

- @Component

```
//注解中省略了value属性
@Component("student")
public class Student {
    private String name;
    private int age;
}
```

该注解用于指定该Bean的id，此外Spring还提供了3个功能基本和@Component等效的注解：

之所以创建三个功能与@Component等效的注解，是为了以后对其进行功能上的扩展，式它们不在等效。

- @Scope

```
@Component(value="student")
@Scope(value="prototype")
public class Student {
    private String name;
    private int age;
}
```

其value属性用于指定该Bean作用域，默认为singleton。

- @Value

```
@Component(value="student")
@Scope(value="prototype")
public class Student {
    @Value("SB")
    private String name;
    @Value("18")
    private int age;
}
```

向基本类型属性注入，该注解的value属性用于指定要注的入值。

- @Autowired （byType）

调用者Bean

```
@Component( "student")
@Scope(value = "proptoype")
public class Student {
    @Value("SB")
    private String name;
    @Value("18")
    private int age;
    @Autowired(required = true)
    private Course course;
}
```

被调用者bean

```
@Component("course")
public class Course {
    private String name;
    private int number;
}
```

该注解默认使用**按类型自动装配Bean**的方式。

- @Autowired与@Qualifier （byName）

  @Autowired与@Qualfier联合是使用，就是指定Bean的id进行注入。

  调用者Bean

```
@Component( "student")
@Scope(value = "proptoype")
public class Student {
    @Value("SB")
    private String name;
    @Value("18")
    private int age;
    @Autowired(required = true)
    @Qualifier(value="course")
    private Course course;
}
```

  被调用者bean

```
@Component("course")
public class Course {
    private String name;
    private int number;
}
```

  @Qualifier中的value属性用于指定注入Bean的id

- @Resource

  Spring提供了对JSR-250规范中定义@Resource标准注解的支持。@Resource注解即可以按名称匹配Bean，也可以按类型匹配Bean。但使用该注解，要求JDK版本必须是6及以上。

  1. 按类型注入域属性

     调用者Bean

```
@Component("student")
@Scope(value = "proptoype")
public class Student {
    @Value("SB")
    private String name;
    @Value("18")
    private int age;
    @Resource
    private Course course;
}
```

被调用者bean

```xml
@Component("course")
public class Course {
    private String name;
    private int number;
}
```

若@Resource注解若不带任何参数，则会按照类型进行Bean的匹配注入。

2. 按名称注入域属性

被调用者bean

```
@Component("course")
public class Course {
    private String name;
    private int number;
}
```

调用者Bean

```
@Component( "student")
@Scope(value = "proptoype")
public class Student {
    @Value("SB")
    private String name;
    @Value("18")
    private int age;
    @Resource(name="course")
    private Course course;
}
```

若@Resource注解指定其name属性，则name的值即为按照名称进行匹配的Bean的id。

使用JUnit4测试Spring

使用Spring的JUnit4对Spring代码进行测试，将不在需要在程序的代码中直接写出创建Spring的容器，及从Spring容器中通过getBean()获取对象了。这些工作将有JUnit4注解，配合者域属性的自动注入注解共同完成。

测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations =
"classpath:com/jr/spring/test16/applicationContext.xml")
public class Test01 {
    @Autowired
    @Qualifier(value = "student")
    Student student;

    @Test
    public void test() {
        System.out.println(student);
    }
}
```

注意

> 出现找不到配置文件错误 java.io.FileNotFoundException: class path resource [com/jr/spring/test16/test/com/jr/spring/test16/applicationContext.xml] cannot be opened because it does not exist

解决办法：记得在注解@ContextConfiguration的locations属性值前面加上classpath，声明在类路径下查找。

> 如果出现Spring-test框架与JUnit框架整合问题，如：

- **提示SpringJUnit4ClassRunner requires JUnit 4.12 or higher.**

  解决办法：引入JUnit4.12的Jar包
- **在依赖JUnit4.12的jar包后，JUnit测试出现java.lang.NoClassDefFoundError: org/hamcrest/SelfDescribing错误**

  解决办法：引入低版本JUnit依赖的其他Jar包Hamcrest.Jar

注解与XML共同使用

注解的好处是，配置方便、直观、但其弊端也显而易见，以硬编码的方式写入到了Java代码，其修改时需要重新编译代码的。 XML配置方式的最大好处是，对其所修改，无需编译代码，只需要重启服务器即可将配置加载。 若注解与XML同用，**XML的优先级要高于注解。**

classpath:与classpath*:

- classpath：只会到你的class路径中查找找文件。
- classpath*：不仅包含class路径，还包括jar文件中（class路径）进行查找. 用classpath*:需要遍历所有的classpath，所以加载速度是很慢的；因此，在规划的时候，应该尽可能规划好资源文件所在的路径，尽量避免使用classpath。

classpath*:的使用

1. 从上面使用的场景看，可以在路径上使用通配符*进行模糊查找。比如：

```
<param-value>classpath:applicationContext-*.xml</param-value>
```

2. "**/"表示的是任意目录；**"/applicationContext-*.xml"表示任意目录下的以"applicationContext-"开头的XML文件。
3. 程序部署到tomcat后，src目录下的配置文件会和class文件一样，自动copy到应用的WEB-INF/classes目录下；classpath:与classpath*:的区别在于，前者只会从第一个classpath中加载，而 后者会从所有的classpath中加载。
4. 如果要加载的资源，不在当前ClassLoader的路径里，那么用classpath:前缀是找不到的，这种情况下就需要使用classpath*:前缀。
5. 在多个classpath中存在同名资源，都需要加载时，那么用classpath:只会加载第一个，这种情况下也需要用classpath*:前缀。

AOP：面向切面

简介

AOP（Aspect Orient Programming）,面向切面编程，是面向对象编程OOP的一种补充。面向对象编程是从静态角度考虑程序的结构，而面向切面编程时从动态角度考虑程序运行过程。 通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。也是Spring框架的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间耦合度降低，提高程序的可重用性，同时提高了开发的效率。 AOP底层就是采用**动态代理模式**实现的。采用了JDK的动态代理和CGLIB的动态代理两种代理。

AOP编程术语

- 切面（Aspect）

  切面泛指交叉业务逻辑。实际就是对主业务逻辑的一种增强，如日志处理和事务处理就可以理解为切面。常用的切面有**通知**和**顾问**。

- 织入（Weaving）

  织入是指将切面代码插入到目标对象的过程。

- 连接点（JoinPoint）

  连接点指**可以**被切面织入的方法。通常业务接口中的方法均为连接点。

- 切入点（Pointcut）

  切入点指切面**具体织入**的方法(已经织入了)。例如↓

接口

```
public interface IServuce{
    void doSome();
    void doOther();
}
```

实现类

```
public class StudentServiceImp implements IService{
    pulic void doSome(){

    }
    pulic void doOther(){

    }
}
```

若doSome()将被增强，而doOther()不被增强，则doSome()为切入点(已被织入)，而doOther()仅为连接点(还未并未织入)

- 目标对象（Target）

  目标对象指将要被增强的对象。即包含主业务逻辑的对象。

- 通知（Advice）

  通知是切面的一种实现，可以完成简单织入功能（织入功能就是在这里为基础完成的）。**通知定义了增强代码切入到目标代码的时间点**，是目标方法执行之前执行，还是之后执行等。同之类型不同，切入时间不同。

切入点定义切入位置，同之定义切入时间。

- 顾问（Advisor）

  顾问是切面的另一种实现，能够将通知以更为复杂的方式织入到目标对象中，是将通知包装为复杂切面的装配器。

AOP编程环境的搭配

AOP是由AOP联盟提出的一种编程思想，提出的一套编程规范。而Spring是这套AOP规范的一种实现。所以需要导入AOP联盟的规范（接口）包及Spring对其的兼容包。

通知(Advice)

通知（Advice），切面是一种实现，可以完成简单的织入功能。常用通知由前置通知，后置通知，环绕通知，异常处理通知。

> 前置通知（MethodBeforeAdvice）

定义前置通知，需要实现MethodBeforeAdvice接口。该接口中有一个方法before()，会在目标方法执行之前执行。前置通知的特点。

1. 在目标方法执行之前执行。
2. 不改变目标方法的执行流程，前置通知代码不能阻止目标代码执行。
3. 不改变目标代码执行结果。

举例：test1

主业务接口

```
public interface IService {
    //主业务方法
    void doSome();

    //主业务方法
    void doOther();
}
```

目标方法

```
public class StudentServiceImp implements IService {
    //    目标方法
    @Override
    public void doSome() {
        System.out.println("doSome");
    }

    //目标方法
    @Override
    public void doOther() {
        System.out.println("doOther");

    }
}
```

前置通知

```
public class MyMethodBeforeAdvice implements MethodBeforeAdvice{
    /**
     *
     * @param method    代表业务方法
     * @param objects   业务方法参数列表
     * @param o          目标对象
     * @throws Throwable
     */
    @Override
    public void before(Method method, Object[] objects, Object o) throws Throwable {
        System.out.println(method.getName()+"方法执行前切入");
    }
}
```

配置文件

```
<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">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test1.dao.StudentServiceImp"></bean>
    <!--配置切面：通知-->
    <bean id="myMethodBeforeAdvice"
class="com.jr.spring.aop.test1.dao.MyMethodBeforeAdvice"></bean>
    <!--配置代理-->
    <bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="studentServiceTarget"></property>
        <property name="interfaces" value="com.jr.spring.aop.test1.dao.IService">
</property>
   <property name="interceptorNames" value="myMethodBeforeAdvice"></property>
    </bean>
</beans>
```

因为AOP底层原理是通过代理实现，那么最终切面和目标对象的联合也就是代理产生的对象。需要通过Spring的代理工厂（ProxyFactoryBean）对象进行加工产生代理对象。

> 后置通知（AfterReturningAdvice）

定义后置通知，需要实现接口AfterReturningAdvice。该接口中有一个方法afterReturning()，会在目标方法执行后执行。后置通知的特点：

1. 在目标方法执行之后执行。
2. 不改变目标方法的执行流程，后置通知代码不能阻止目标方法执行。
3. 不改变目标方法执行的结果。

举例：test02

定义切面：通知

```
public class MyAfterReturningAdvice implements AfterReturningAdvice {


    /**
     * @param result  目标方法结果返回值
     * @param method  代表业务方法
     * @param objects 业务方法参数列表
     * @param bean       目标对象
     * @throws Throwable
     */
    @Override
    public void afterReturning(Object result, Method method, Object[] objects, Object
bean) throws Throwable {
        System.out.println("目标方法执行后切入");
        if (result!=null){
            System.out.println("目标方法结果返回值:" + result);
```

```
        }
    }
}
```

> 环绕通知（MethodInterceptor）

定义环绕通知，需要实现MethodInterceptor接口。环绕通知，也叫方法拦截器。可以在目标方法调用之前及之后做处理，可以该改变目标方法的返回值，也可以改变程序执行流程。

举例：test03

定义切面：通知

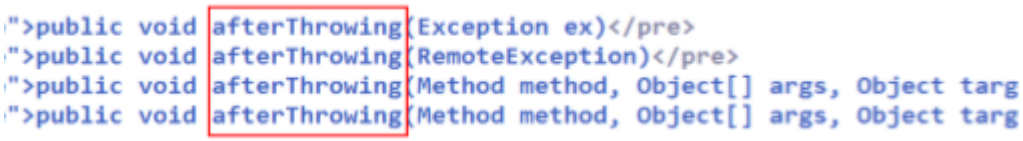
```
public class MyMethodInterceptor implements MethodInterceptor {

    /**
     *
     * @param methodInvocation 拦截的目标方法对象。通过该对象可对目标对象方法进行控制
     * @return                 返回目标方法值
     * @throws Throwable
     */
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        //目标方法执行前切入
        System.out.println(methodInvocation.getMethod().getName() + "目标方法执行前切入");
        //执行目标方法并获取返回值
        String result = (String) methodInvocation.proceed();
        //修改返回值
        if (result != null) {
            result = result.toLowerCase();
        }
        //目标方法执行后切入
        System.out.println(methodInvocation.getMethod().getName() + "目标方法执行后切入");
        return result;
    }
}
```

> 异常通知

定义异常通知，需要实现ThrowsAdvice接口。该接口的主要作用是，在目标方法抛出异常后，根据异常的不同做出相应的处理。当该接口处理完异常后，会简单地将异常在次抛出给目标方法。

不过，该接口较为特殊，从形式上看，该接口中没有必要实现的方法。但，这个接口确实有必须要实现的方法afterThrowing()。。这个方法重载了四种形式。由于使用时，一般只使用其中一种，若要都定义到接口中，则势必会使程序员在使用时必须要实现这四个方法。这是很麻烦的。所以将该接口定义为**标识接口（没有方法的接口）**

这四个方法在打开ThrowsAdvice源码后，上侧的注释部分可以看到



不过，在这四种形式中，常用的形式如下：

```
public void afterThrowing(自定义的异常类 e)
```

这里的参数e为，与具体业务相关的用户自定义的异常类对象。容器会根据异常类型的不同，自动选择（根据重载方法中的异常参数与方法抛出的异常存在is-a关系的方法）不同的该方法执行。这些方法的执行实在目标方法执行结束后执行的。

举例test4：

本例实现用户身份验证。当用户名不正确时，抛出用户名有误异常：当密码不正确时，抛出密码有误异常。在然，在抛出这些异常处理后，都要做一些其他处理。

定义异常处理类父类

```
public class UserException extends Exception {
    public UserException() {
        super();
    }

    public UserException(String message) {
        super(message);
    }
}
```

用户名错误异常类

```
public class UserException extends Exception {
    public UserException() {
        super();
    }

    public UserException(String message) {
        super(message);
    }
}
```

密码错误异常类

```
public class UserPasswordException extends UserException {
    public UserPasswordException() {
        super();
    }

    public UserPasswordException(String message) {
        super(message);
    }
}
```

主业务接口

```
public interface IService {
    boolean check(String userName, String password) throws UserException;
}
```

目标对象

```
public class UserService implements IService {
    @Override
    public boolean check(String userName, String password) throws UserException {
        if (!userName.equals("admin")) {
            throw new UserNameException("用户名有误");
        }
        if (!password.equals("admin")) {
            throw new UserPasswordException("密码有误");
        }
        return true;
    }
}
```

切面：通知

```
public class MyThrowsAdvice implements ThrowsAdvice {
    public void afterThrowing(UserNameException e) {
        System.out.println("异常通知捕捉到UserPasswordException异常，异常信息为:" +
e.getMessage());
    }

    public void afterThrowing(UserPasswordException e) {
        System.out.println("异常通知捕捉到UserPasswordException异常，异常信息为:" +
e.getMessage());
    }

}
```

> 通知的其他方式

- 给目标方法织入多个切面

若要给目标方法织入多个切面，则需要在配置代理对象的切面属性时，设定为list

举例：test5

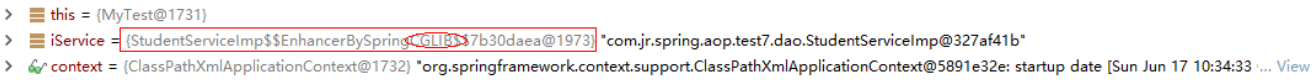
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test5.dao.StudentServiceImp"></bean>
    <!--配置切面：前置通知-->
    <bean id="myBeforeMethodAdvice"
class="com.jr.spring.aop.test5.dao.MyMethodBeforeAdvice"></bean>
    <!--配置切面：后置通知-->
    <bean id="myAfterReturningAdvice"
class="com.jr.spring.aop.test5.dao.MyAfterReturningAdvice"></bean>
    <!--配置代理-->
    <bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="studentServiceTarget"></property>
        <property name="interfaces" value="com.jr.spring.aop.test5.dao.IService">
</property>
        <property name="interceptorNames">
            <list>
                <value>myBeforeMethodAdvice</value>
                <value>myAfterReturningAdvice</value>
            </list>
        </property>
    </bean>
</beans>
```

> 无接口的CGLIB代理生成

若不存在接口，则ProxyFactoryBean会自动采用CGLIB方式生成动态代理 *举例：test6*

```
<bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="studentServiceTarget"></property>
    <property name="interceptorNames" value="myMethodBeforeAdvice"></property>
</bean>
```

由于没有了接口，所以在配置代理的时候，不需要在设定接口属性



> 有接口的CGLIB代理生成proxyTargetClass属性

若存在接口，但需要使用CGLIB生成代理对象，此时，需要在配置文件中增加一个proxyTargetclass属性设置，用于强制使用CGLIB代理机制

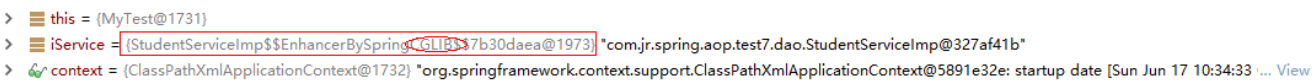
举例：test7

配置文件

```
<bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="studentServiceTarget"></property>
    <property name="interfaces" value="com.jr.spring.aop.test7.dao.IService">
</property>
    <property name="interceptorNames" value="myMethodBeforeAdvice"></property>
    <property name="proxyTargetClass" value="true"></property>
</bean>
```

也可指定optimize（优化）的值为true，强制使用GBLIB代理机制

```
<bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="studentServiceTarget"></property>
    <property name="interfaces" value="com.jr.spring.aop.test7.dao.IService">
</property>
    <property name="interceptorNames" value="myMethodBeforeAdvice"></property>
    <property name="optimize" value="true"></property>
</bean>
```



顾问(Advisor)

通知是Spring提供的一种切面。但其功能过于简单，只能将切面织入到目标类的所有目标方法中，无法完成将切面织入到指定目标方法中。

顾问是Spring提供的另一种切面。其可以完成更为复杂的切面织入功能。

顾问将通知进行了包装，会根据不同的通知类型，在不同的时间点，将切面织入到不同的切入点。

> NameMatchMethodPointCutAdvisor

NameMatchMethodPointcutAdvisor，即名称匹配方法切入点顾问。容器可以根据配置文件中指定的方法名来设置切入点。

代码不用修改，只在配置文件中注册一个顾问，然后使用通知属性advice与切入点的方法名属性mappedName对其进行配置。代理中的切面，使用这个顾问即可。

举例：test8

```
<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">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test8.dao.StudentServiceImp"></bean>
    <!--配置切面：通知-->
    <bean id="myMethodBeforeAdvice"
class="com.jr.spring.aop.test8.dao.MyMethodBeforeAdvice"></bean>
    <!--配置切面：顾问-->
    <bean id="myNameMatchMethodPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice" ref="myMethodBeforeAdvice"></property>
        <!--织入单个切入点-->
        <!--<property name="mappedName" value="doSome"></property>-->
        <!--织入多个切入点-->
        <property name="mappedNames">
           <array>
                <value>doSome</value>
                <value>doOther</value>
            </array>
        </property>
    </bean>
    <!--配置代理-->
    <bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="studentServiceTarget"></property>
        <property name="interfaces" value="com.jr.spring.aop.test8.dao.IService">
</property>
        <property name="interceptorNames" value="myNameMatchMethodPointcutAdvisor">
</property>
    </bean>
</beans>
```

> RegexpMethodPointcutAdvisor

RegexpMethodPointcutAdvisor，即正则表达式方法顾问，容器可根据正则表达式来设置切入点，注意，与正则表达式进行匹配的**对象是接口**中的方法名(全限定方法名)，而非目标类（接口中的实现类）的方法名。

举例test9

常用的正则表达式的运算符：

运算符	名称	意义
.	点号	表示任意单个字符
+	加号	表示前一个字符出现一次或者多次
*	星号	表示前一个字符出现0次或者多次

举例	分析
aa\.bb\.Hello\.do.*	表示aa.bb.Hello类中所有以do开头的方法作为切入点方法。由于点号即是包与包间的连接符，有实正则表达式运算符。为了指明这里的点号不是正则表达式运算符，所以要使用转义字符--""。
.*do.*	表示方法全名（包括包名、接口名、方法名）中包含do的方法作为切入点方法。
.do.*|.*S.*	表示方法全名中包含do或者S的方法均为切入点。
.*do.*.S.*	表示方法全名中即包含do又包含S的方法，作为切入点方法。

配置文件

```
<!--配置目标对象-->
<bean id="studentServiceTarget" class="com.jr.spring.aop.test9.dao.StudentServiceImp">
</bean>
<!--配置切面：通知-->
<bean id="myMethodBeforeAdvice"
class="com.jr.spring.aop.test9.dao.MyMethodBeforeAdvice"></bean>
<!--配置切面：顾问-->
<bean id="myRegexpMethodPointcutAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice" ref="myMethodBeforeAdvice"></property>
    <!--织入单个切入点-->
    <property name="pattern" value="com.jr.spring.aop.test9.dao.IService.do.*">
    </property>
</bean>
<!--配置代理-->
<bean id="studentServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="studentServiceTarget"></property>
    <property name="interfaces" value="com.jr.spring.aop.test9.dao.IService">
</property>
    <property name="interceptorNames" value="myRegexpMethodPointcutAdvisor"></property>
</bean>
```

- 正则表达式的写法

1. 通过patterns匹配一条正则表达式织入切入点

```
<property name="pattern" value="com.jr\spring.aop.test9.dao.IService.do.*">
```

2. 通过patterns匹配多条正则表达式织入切入点

```
<property name="patterns">
    <array>
        <value>com.jr.spring.aop.test9.dao.IService.doSome</value>
        <value>com.jr.spring.aop.test9.dao.IService.doOther</value>
    </array>
</property>
```

3. 过'|'符号匹配多条正则表达式织入切入点

```
<property name="pattern"
value="com.jr.spring.aop.test9.dao.IService.doSome|com.jr.spring.aop.test9.dao.ISer
vice.doOther"></property>
```

注意

NameMatchMethodPointCutAdvisor顾问根据匹配的**方法名**进行织入。

RegexpMethodPointcutAdvisor顾问匹配的是接口方法名而不是目标对象方法名。

RegexpMethodPointcutAdvisor顾问根据匹配的**全限定方法名（包名+接口名|类名+方法名）**进行织入。

自动代理生成器

前面代码中使用的代理对象，均是又ProxyFactoryBean代理工具类生成的。而该代理工具存在如下缺点：

1. 一个代理对象只能代理一个Bean，即如果有两个Bean同时都要织入同一个切面，这时，不仅要配置这两个Bean，即两个目标对象，同时还要配置两个代理对象。
2. 在客户类中获取Bean时，使用的是代理类的id，而非我们定义的目标对象Bean的ID。我们真正想要执行的应该是目标对象。从形式上看，不符合正常的逻辑。

   Spring提供了自动代理生成器，用于解决ProxyFactoryBean的问题。自动代理生成器均继承来自Bean后处理器BeanPostProcessor。容器中所有Bean在初始化时均自动执行Bean后处理器中的方法，故其无需id属性。所以自动代理生成器的Bean也没有id属性，调用者类直接使用目标对象的ID。

   常用的自动代理生成器有两个：

> 默认的advisor自动代理生成器

DefaultAdvisorAutoProxyCreator代理的生成方式是，将所有目标对象与Advisor自动结合，生成代理对象。无需给生成器做任何的注入配置。**注意，只能与Advisor（顾问）配置使用。**

举例test10

配置文件

```
<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">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test10.dao.StudentServiceImp"></bean>
    <!--配置切面：通知-->
    <bean id="myMethodBeforeAdvice"
class="com.jr.spring.aop.test10.dao.MyMethodBeforeAdvice"></bean>
    <!--配置切面：顾问-->
    <bean id="myNameMatchMethodPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice" ref="myMethodBeforeAdvice"></property>
        <property name="mappedName" value="doSome"></property>
    </bean>
    <!--配置自动代理生成器-->
```

```
    <bean
class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
</bean>
</beans>
```

测试类

```
public class MyTest {
    ApplicationContext context;

    @Before
    public void before() {
        context = new
ClassPathXmlApplicationContext("com/jr/spring/aop/test10/applicationContext.xml");
    }

    @Test
    public void Test() {
        IService iService= (IService) context.getBean("studentServiceTarget");
        iService.doSome();
        System.out.println("----------------");
        iService.doOther();
    }

}
```

调用者类直接获取注册Bean的ID即可，最后获取的Bean即使织入切面的对象Bean。

> Bean名称自动代理生成器

DefaultAdvisorAutoProxyCreator会为每一个目标对象织入所有匹配的Advisor，不具有选择性，**且切面只能是顾问Advisor**。而BeanNameAutoProxyCreator的代理生成方式是，根据Bean的ID，来为符合相应名称的类生成相应代理对象，且切面既可以是顾问Advisor有可以是通知Adivice。

举例test11

```
<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">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test11.dao.StudentServiceImp"></bean>
    <bean id="teacherServiceTarget"
class="com.jr.spring.aop.test11.dao.TeacherServiceImp"></bean>
    <!--配置切面：前置通知-->
    <bean id="myMethodBeforeAdvice"
class="com.jr.spring.aop.test11.dao.MyMethodBeforeAdvice"></bean>
    <!--配置切面：后置通知-->
    <bean id="myAfterReturningAdvice"
class="com.jr.spring.aop.test11.dao.MyAfterReturningAdvice"></bean>
    <!--配置切面：顾问-->
```

```
    <bean id="myNameMatchMethodPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice" ref="myMethodBeforeAdvice"></property>
        <!--织入单个切入点-->
        <!--<property name="mappedName" value="doSome"></property>-->
        <!--织入多个切入点-->
        <property name="mappedNames">
            <array>
                <value>doSome</value>
                <value>doOther</value>
            </array>
        </property>
    </bean>

    <!--配置自动代理生成器-->
    <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames" value="studentServiceTarget,teacherServiceTarget">
</property>
        <property name="interceptorNames"
value="myMethodBeforeAdvice,myAfterReturningAdvice,myNameMatchMethodPointcutAdvisor">
</property>
    </bean>
</beans>
```

技巧

- 对于Bean的名称的指定，可以使用通配符“*”号

```
  <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test11.dao.StudentServiceImp"></bean>
    <bean id="teacherServiceTarget"
class="com.jr.spring.aop.test11.dao.TeacherServiceImp"></bean>
  <!--配置自动代理生成器-->
<property name="beanNames" value="*Service*"></property>
```

Aspectj对AOP的实现

简介

对于AOP这种编程思想，很多框架都进行了实现。Spring就是其中之一，可以完成面向切面编程，然而，Aspectj也实现了AOP的功能，且其实现方法更为简捷，使用更为方便，而且还支持注解式开发。所以，Spring又将Aspectj的对于AOP的实现也引入到了自己的框架中。

在Spring中使用AOP开发时，一般使用Aspectj的实现方法。

AspectJ对于AOP的实现有两种方式：**注解方式**和**XML方式**

开发环境

1. **导入Jar包**

AspectJ是专门针对AOP问题的，所有其运行是需要AOP环境的，即需要之前的AOP的两个Jar包（**AOP联盟**和**Spring整合AOP联盟的兼容包**）。另外，还需要**AspectJ自身的Jar**与**Spring整合AspectJ的兼容包**

一般情况下，使用weaver包下的jar即可。tools中的Jar除了包含weaver中类库外还包含其他工具，但一般用。所以，使用weaver包下的jar即可。

2. **引入AOP约束**

   在配置文件头部，要引入关于AOP的约束。

> 在前面Spring实现AOP时，并未引入AOP的约束，而在AspectJ实现AOP时，才提出要引入AOP的约束。说明，配置文件中使用的AOP约束中的标签，均是AspectJ框架使用的，而非Spring框架本身在实现AOP时使用的。

通知类型

Aspectj中常用的通知有五种类型：

1. 前置通知
2. 后置通知
3. 环绕通知
4. 异常通知
5. **最终通知**

其中最终通知是指，无论程序执行是否正常，该通知都会执行，类似于try...catch中的finally代码块。

切入点表达式

AspectJ除了提供了五种通知外，还定义了专门的表达式，**功能类似与顾问**用于指定切入点。。表达式的原型是：

execution([modifiers-pattern] 访问权限类型

```
            **ret-type-pattern**          返回值类型

            [declaring-type-pattern]      全限定性类名

            **name-pattern(param-pattern)**       方法名(参数名)

            [throws-pattern]              抛出异常类型

        )
```

切入点表达式要匹配的对象就是目标方法的方法名。所以，execution表达式中明显就是方法的签名。注意，表达式中加【】的部分表示可省略部分，各部分间空格分开。在其中可以使用以下符号：

符号	意义
*	0至多个任意字符
..	用在方法参数中，表示任意多个参数。 用在包名后，表示当前包及其子包路径
+	用在类名后，表示当前类及其子类 用在接口后，表示当前接口及其实现类 用在方法参数，表示参数类型为该参数类型于其子类类型

举例：

- **execution(public * *(..))**

  指定切入点为：任意公共方法

- **execution(* set * (..))**

  指定切入点为：任何一个以"set"开始的方法

- **execution(* com.xyz.service.*.*(..))**

  指定切入点为：定义在service包里的任意类的任意方法

- **execution(* com.xyz.service..*.*(..))**

  指定切入点为：定义在service包或者子包里的任意类的任意方法。“..”出现在类名中，后面必须跟“*”，表示包，子包下的所有类

- **execution(* *.service.*.*(..))**

  指定只有一级包下的service子包下所有类(或接口)中所有方法为切入点

- **execution(* *..service.*.*(..))**

  指定所有包下的service子包下所有类(或接口)中的所有方法为切入点

- **execution(* com.xyz.service.IAccountService.*(..))**

  指定切入点为：IAccountSerivce接口中的任意方法

- **execution(* com.xyz.service.IAccountService+.*(..))**

  指定切入点为：IAccountService若为接口，则接口中任意方法及其所有实现类中的任意方法为切入点。若为类，则该类及其子类中的任意方法为切入点

- **execution(* joke(String,int))**

  指定切入点为：所有的joke(String,int)方法，且joke()方法的第一个参数是String，第二个参数是int。如果方法中的参数类型是java.lang包下的类，可以直接使用类名，否则必须使用全限定类名，如joke(java.util.List,int)

- **execution(* joke(String,..))**

  指定切入点为：所有的joke()方法，该方法第一个参数为String，后面可以有任意个参数且参数类型不限，如joke(String s1)，joke(String s1,String s2)或joke(String s1,double d2 ,String s3)都行

- **execution(* joke(Object))**

  指定切入点：所有的joke()方法，方法拥有一个参数，且参数是Object类型。joke(Object obj)是，但，joke(String s)或joke(User user)均不是

- **execution(* joke(Object+))**

  指定切入点：所有的joke()方法，方法拥有一个参数，且参数是Object类型或该类的子类。joke(Object obj)是，但，joke(String s)或joke(User user)也是

基于注解的AOP实现

举例test12

实现步骤

- **Step1:定义业务接口与实现类**

```
public interface IService {
    //主业务方法
    void doSome();

    //主业务方法
    void doOther();
}
```

```
public class StudentServiceImp implements IService {
    //    目标方法
    @Override
    public void doSome() {
        System.out.println("doSome");
    }

    //目标方法
    @Override
    public void doOther() {
        System.out.println("doOther");
    }
}
```

- **Step2:定义切面POJO类**

  该类为一个POJO类，将作为切面出现。其中定义了若干个普通方法，将作为不同的通知方法。

```
public class MyAspect {
    public void before() {
        System.out.println("前置增强");
    }

    public void afterReturning() {
        System.out.println("后置增强");
    }
    //other method
}
```

- **Step3:在切面类上添加@Aspect注解**

  在定义的POJO类上添加@Aspect注解，指定当前POJO类将作为切面。

```
@Aspect
public class MyAspect {
    public void before() {
        System.out.println("前置增强");
    }

    public void afterReturning() {
        System.out.println("后置增强");
    }
    //other method
}
```

- **Step4:在POJO类的普通方法上添加通知注解**

  切面类是用于定义增强代码的，即用于定义增强目标类中的目标方法的增强方法。这些增强方法使用不同的"通知"注解，会在不同的时间点完成织入。当然，对于增强代码，还要通过execution表达式指定具体应用的目标类于目标方法，即切入点。

```
@Aspect
public class MyAspect {
    @Before(value = "execution(* com.jr.spring.aop.test12.dao.IService.do*())")
    public void before() {
        System.out.println("前置增强");
    }

    public void afterReturning() {
        System.out.println("后置增强");
    }
    //other method
}
```

- **Step5:注册目标对象与POJO切面类**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test12.dao.StudentServiceImp"></bean>
    <!--配置Aspectj切面-->
    <bean class="com.jr.spring.aop.test12.dao.MyAspect"></bean>
</beans>
```

- **Step6：注册AspectJ的自动代理**

  在定义好切面Aspect后，需要通知Spring容器，让容器生成“目标类+切面”的代理对象。这个代理是由容器自动生成的。只需要在Spring配置文件中注册一个基于aspectj的自动代理生成器。其就会自动扫描到@Aspect注解，并按通知类型与切入点，将其织入，并生成代理

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--配置aspectj的自动代理-->
    <aop:aspectj-autoproxy/>
    <!--配置目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test12.dao.StudentServiceImp"></bean>
    <!--配置Aspectj切面-->
    <bean class="com.jr.spring.aop.test12.dao.MyAspect"></bean>
</beans>
```

@Before方法由JoinPoint参数

在目标方法执行之前执行。被注解为前置通知的方法，可以包含一个JoinPoint类型参数。该类型的对象本身就是切入点表达式，可以获取切入点表达式、方法签名、目标对象等。

不光前置通知的方法可以包含一个JoinPoint类型参数，所有的通知放啊均可以包含该参数

```
@Before(value = "execution(* com.jr.spring.aop.test12.dao.IService.doOther(..))")
public void beforeOther(JoinPoint joinPoint) {
    System.out.println("前置增强，切入点表达式为:"+joinPoint);
    System.out.println("前置增强，方法签名为:"+joinPoint.getSignature());
    System.out.println("前置增强，目标对象为:"+joinPoint.getTarget());

    Object[]args=joinPoint.getArgs();
    if (args.length!=0) {
        System.out.println("前置增强，方法参数为: ");
        for (Object arg : args) {
            System.out.println(arg+"    ");
        }
    }
}
```

@AfterReturning注解有returning属性

在目标方法执行之后在执行。由于是目标方法之后执行，所以可以获取到目标方法的返回值。该注解的returing属性就是用于指定接受方法返回值的变量名的。所以，被注解为后置通知的方法，除了可以包含JoinPoint参数外，还可以包含用于接收返回值的变量。该变量最好为Object类型，因为目标方法的返回值可能是任何类型。

```
@AfterReturning(value = "execution(*
com.jr.spring.aop.test12.dao.IService.doThrid(int,int))", returning = "result")
public void afterReturning(int result) {
    System.out.println("后置增强,目标方法返回值: "+result);
}
```

@Around增强方法有ProceedingJoinPoint参数

在目标方法执行之前之后执行。被注解为环绕增强的方法要有返回值，Object类型。并且方法可以包含一个ProceedingJoinPoint类型的参数。

接口ProceedingJoinPoint其有一个proceed()方法，用于执行目标方法。若目标方法有返回值，则该方法的返回值就是目标方法的返回值。

最后，环绕增强方法将其返回值返回，该增强方法实际就是拦截了目标方法的执行。

```
@Around(value = "execution(* com.jr.spring.aop.test12.dao.IService.doThrid(int,int)))")
public Object afterReturning(ProceedingJoinPoint proceedingJoinPoint) {
    System.out.println("环绕增强：前");
    Object result = null;
    try {
        Object[] args = proceedingJoinPoint.getArgs();
        result = proceedingJoinPoint.proceed(args);
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
    System.out.println("环绕增强：前");
    return result;
}
```

@AfterThrowing注解中有throwing属性

在目标方法抛出异常后执行。该注解的throwing属性用于指定所发生的异常类对象。当然，被注解为异常通知的方法可以包含一个参数Throwable，参数名称为throwing指定名称，表示发生的异常对象。

```
@After(value = "execution (* com.jr.spring.aop.test12.dao.IService.doAfter()))")
public void after() {
    System.out.println("最终方法执行");
}
```

@Pointcut定义切入点

当较多的通知增强方法使用相同的切入带你表达式时，编写，维护均较为麻烦。

AspectJ提供了@Pointcut注解，同于定义execution切入点表达式。

```
@Pointcut(value = "execution (* com.jr.spring.aop.test12.dao.IService.doPointcut()))")
private void myExecution() {
}//表示方法

@Before(value = "myExecution()")
public void doPointcut() {
    System.out.println("使用了Pointcut作为execution表达式进行了前置增强");
}
```

其用法是，将@Pointcut注解在一个方法之上，以后所有的execution的value属性值均可使用该方法名作为切入点。代表的就是@Pointcut定义的切入点。这个使用@Pointcut注解的方法一般使用private的标识方法，即没有实际作用的方法。

基于XML的AOP实现

Aspectj除了提供了基于注解的AOP的实现外，还提供了以XML方式的实现。

切面就是一个POJO类，而用于增强的方法就是普通的方法。通过配置文件，将切面中的功能增强织入到了目标类的目标方法中。

举例：test13

实现步骤

Step1:定义业务接口与实现类

```
public interface IService {
    void doBefore();

    void doBeforeHaveJoinPoint(String name, int age);

    int doAfterReturning(int a, int b);

    int doAround(int a);

    void doThrowing();

    void doAfter();
}
```

```
public class StudentServiceImp implements IService {
    @Override
    public void doBefore() {
        System.out.println("目标方法执行doSome()");
    }

    @Override
    public void doBeforeHaveJoinPoint(String name, int age) {
        System.out.println("目标方法执行doOther()");
    }

    @Override
    public int doAfterReturning(int a, int b) {
        System.out.println("目标方法执行doThrid()");
        return a + b;
    }

    @Override
    public int doAround(int a) {
        System.out.println("目标方法执行doAround()");
        return a;
    }

    @Override
    public void doThrowing() {
        System.out.println("目标方法执行doThrowing()");
        int result = +(3 / 0);
    }
```

```
    @Override
    public void doAfter() {
        System.out.println("目标方法执行doAfter()");
    }

}
```

Step2:定义切面POJO类

该类为一个POJO类，将作为切面出现。其中定义了若干普通方法，将作为不同的通知方法。

```
public class MyAspect {

    //定义前置通知的增强方法
    public void before() {
        System.out.println("前置增强");
    }
    //定义具有JoinPoint形参的前置通知的增强方法
    public void beforeHaveJoinPoin(JoinPoint joinPoint) {
        System.out.println("前置增强，切入点表达式为:" + joinPoint);
        System.out.println("前置增强，方法签名为:" + joinPoint.getSignature());
        System.out.println("前置增强，目标对象为:" + joinPoint.getTarget());

        Object[] args = joinPoint.getArgs();
        if (args.length != 0) {
            System.out.println("前置增强，方法参数为: ");
            for (Object arg : args) {
                System.out.println(arg + "    ");
            }
        }
    }

    //定义后置通知的增强方法
    public void afterReturning(int result) {
        System.out.println("后置增强,目标方法返回值: " + result);
    }

    //定义环绕通知的增强方法
    public int around(ProceedingJoinPoint proceedingJoinPoint) {
        System.out.println("环绕增强: 前");
        int result = 1;
        try {
            Object[] args = proceedingJoinPoint.getArgs();
            result = (int) proceedingJoinPoint.proceed(args);
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("环绕增强: 后");
        return result * 100;
    }

    //定义异常通知的增强方法
```

```
    public void afterThrowing(Throwable throwable) {
        System.out.println(throwable.getMessage());
    }

    //定义最终通知的增强方法
    public void after() {
        System.out.println("最终方法执行");
    }
}
```

Step3:注册目标对象与POJO切面类

与使用注解实现AOP的方式不一样的是，此时注册切面时需要指定该切面的ID，在下面配置AOP时使用

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--注册目标对象-->
    <bean id="studentServiceTarget"
class="com.jr.spring.aop.test12.dao.StudentServiceImp"></bean>
    <!--注册Aspectj切面-->
    <bean id="myAspect" class="com.jr.spring.aop.test12.dao.MyAspect"></bean>
</beans>
```

Step4:在容器中定义AOP配置

配置文件中，除了要定义目标类与切面的Bean外，最主要的是在<aop-config/>中进行AOP的配置。而该标签的底层，会根据其子标签的配置，生成自动代理。

```
<aop:config>
    <!--定义切入点-->
    <aop:pointcut id="doBeforePointcut" expression="execution(*
com.jr.spring.aop.test13.dao.IService.doBefore())"/>
    <!--定义切面-->
    <aop:aspect ref="myAspect">
        <aop:before method="before" pointcut-ref="doBeforePointcut"></aop:before>
    </aop:aspect>
</aop:config>
```

通过子标签<aop-aspect/>定义具体的织入规则：根据不同的通知类型，确定不同织入时间。

Step5:测试类中使用目标对象的ID

```
@Test
public void Test() {
    IService iService = (IService) context.getBean("studentServiceTarget");
    iService.doBefore();
    System.out.println("----------------");
```

```
        iService.doBeforeHaveJoinPoint("SB", 18);
        System.out.println("----------------");
        iService.doAfterReturning(1, 2);
        System.out.println("----------------");
        System.out.println(iService.doAround(1));
        System.out.println("----------------");
        iService.doAfter();  System.out.println("----------------");
        System.out.println("----------------");
        iService.doThrowing();
    }
```

后置通知

在XML的配置中，有一个属性returning，指定用于接收目标方法的返回值所使用的变量名，其可作为增强方法的参数出现

AOP配置

```
<aop:aspect ref="myAspect">
    <!-- 定义切入点-->
    <aop:pointcut id="doAfterReturningPointcut"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doAfterReturning(int,int))"/>
    <!--有返回值的后置通知-->
    <aop:after-returning method="afterReturning" pointcut-
ref="doAfterReturningPointcut" returning="result"/>
</aop:aspect>
```

切面类

```
public class MyAspect {
    //定义后置通知的增强方法
    public void afterReturning(int result) {
        System.out.println("后置增强,目标方法返回值: " + result);
    }
}
```

增强方法的参数名必须和AOP配置后置通知的returning的属性值一致。

环绕通知

AOP配置

```
<aop:config>
    <!--定义切入点-->
    <aop:pointcut id="doAround"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doAround(int))"/>
    <!--定义切面-->
    <aop:aspect ref="myAspect">
        <!--环绕通知-->
        <aop:around method="around" pointcut-ref="doAround"></aop:around>
    </aop:aspect>
</aop:config>
```

切面类

```
public int around(ProceedingJoinPoint proceedingJoinPoint) {
    System.out.println("环绕增强：前");
    int result = 1;
    try {
        Object[] args = proceedingJoinPoint.getArgs();
        result = (int) proceedingJoinPoint.proceed(args);
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
    System.out.println("环绕增强：后");
    return result * 100;
}
```

环绕通知的增强方法一般返回类型为Object，是目标方法的返回值。并且可以包含一个参数ProceedingJoinPoint，其方法proceed()可执行目标方法。

异常通知

在XML的配置中，有一个属性throwing，指定用于接收目标方法所抛出异常的变量名。其可作为增强方法的参数出现，该参数为Throwable类型。

AOP配置

```
    <aop:config>
        <!--定义切入点-->
        <aop:pointcut id="doThrowing"
                      expression="execution(*
com.jr.spring.aop.test13.dao.IService.doThrowing())"/>
        <!--定义切面-->
        <aop:aspect ref="myAspect">
            <!--异常通知-->
            <aop:after-throwing method="afterThrowing" pointcut-ref="doThrowing"
                                throwing="throwable"></aop:after-throwing>
        </aop:aspect>
    </aop:config>
```

切面类

```
public class MyAspect {
```

增强方法的参数名必须和AOP配置异常通知的throwing的属性值一致。

最终通知

AOP配置

```
<aop:config>
    <!--定义切入点-->
    <aop:pointcut id="doBeforePointcut" expression="execution(*
com.jr.spring.aop.test13.dao.IService.doBefore())"/>
    <aop:pointcut id="doBeforeHaveJoinPointPointcut"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doBeforeHaveJoinPoint())"/>
    <aop:pointcut id="doAfterReturningPointcut"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doAfterReturning(int,int))"/>
    <aop:pointcut id="doAround"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doAround(int))"/>
    <aop:pointcut id="doThrowing"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doThrowing())"/>
    <aop:pointcut id="doAfter"
                  expression="execution(*
com.jr.spring.aop.test13.dao.IService.doAfter())"/>
    <!--定义切面-->
    <aop:aspect ref="myAspect">
        <!--前置通知-->
        <aop:before method="before" pointcut-ref="doBeforePointcut"></aop:before>
        <!--有JoinPoin参数的前置通知-->
        <aop:before method="beforeHaveJoinPoin" pointcut-
ref="doBeforeHaveJoinPointPointcut"></aop:before>
        <!--有返回值的后置通知-->
        <aop:after-returning method="afterReturning" pointcut-
ref="doAfterReturningPointcut" returning="result"/>
        <!--环绕通知-->
        <aop:around method="around" pointcut-ref="doAround"></aop:around>
        <!--异常通知-->
        <aop:after-throwing method="afterThrowing" pointcut-ref="doThrowing"
                            throwing="throwable"></aop:after-throwing>
        <!--最终通知-->
        <aop:after method="after" pointcut-ref="doAfter"></aop:after>
    </aop:aspect>
</aop:config>
```

切面类

```
public void after() {
    System.out.println("最终增强通知方法执行");
}
```

Spring与Dao

前言

Spring与Dao部分，是Spring的两大核心技术IoC与AOP的典型应用体现。

对于JDBC模板的使用，是IoC的应用，是将JDBC模板对象注入给Dao层的实现类。

对于Spring的事务管理，是AOP的应用，将事务作为切面织入到了Service层的业务方法中。

Spring与JDBC模板

为了避免直接使用JDBC而带来的复杂也冗长的代码，Spring提供了一个强有力的模板类--JdbcTemplate来简化JDBC操作。并且，数据源DataSource对象与模板JdbcTemplate对象均可通过Bean的形式定义在配置文件中，充分发挥了依赖注入的能力。

举例:spring_dao

导入Jar包

除了Spring的基本Jar包，数据库驱动Jar外，还需要导入两个Jar包。它们均在Spring框架解压目录下的libs目录中。

- Spring的JDBC的Jar包

Spring-jdbc-4.2.1.RELEASE.jar

- Spring的事务Jar包

Spring-tx-4.2.1.RELEASE.jar

使用步骤

1.配置数据源

什么是数据库连接池

往往不同的的应用模块都会用不同服务器对其承载，如数据库有自己的数据库服务器，Web应用有Web服务器。

而不同服务器之间进行的通信可能是跨区域的。那么当Web服务器需要对数据库进行操作是，首先需要进行的是就数据库进行连接，然后在进行操作，而有用功只是耗费在了数据库的操作当中，但在连接数据库时却浪费了其他时间。并且在完成数据库操作之后，关闭数据库也是需要耗费时间的，所以整个过程存在许多耗费时间的地方。

数据库连接池的出现主要就是解决操作数据库中的Connection(即连接)的问题 。连接对象（Connection）采用池化的原因：采用池化的本意是通过减少对象生成的次数，减少花在对象初始化上面的开销，从而提高整体性能。

一般Spring操作数据库使用到的数据源有三种：**Spring的数据源、DBCP、C3P0或其他的数据库框架**

- 使用Spring默认的数据源

```
    <bean id="driverManagerDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/test"></property>
        <property name="username" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>
```

DriverManagerDataSource建立连接是只要能建立连接就新建一个connection，根本没有连接池的概念。

- 使用DBCP框架的数据源

```
    <bean id="basicDataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/test"></property>
        <property name="username" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>
```

- 使用C3P0框架的数据源

```
    <bean id="comboPooledDataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/test">
</property>
        <property name="user" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>
```

DBCP和C3P0才真正使用到了数据库连接池技术

数据源的配置其实有需要参数的配置，如数据源的创建时机、数据源创建的个数、销毁的时间和时机等。但一般程序源不配置这些，而交由经验丰富的管理员来配置。

不配置的话，默认采用框架的配置。

从配置文件中读取数据

为了便于维护，可以将数据库连接信息写入到属性文件中，使Spring配置文件从中读取数据。

Spring配置文件从属性文件中读取数据时，需要在<property/>的value属性中使用${},将属性文件中定义的key括起来，以引用指定属性的值。

```
    <bean id="driverManagerDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.user}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>
```

该属性文件若要被Spring配置文件读取，其必须在配置文件中进行注册。注册方式有两种：

- <bean/>方式

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="location" value="classpath:com/jr/spring/spring_dao/jdbc"></property>
</bean>
```

以PropertyPlaceholderConfigurer类的bean实例的方式进行注册，该类有一个属性location，用于指定属性文件的位置。这种方式不常用。

- 使用context方式

```
<context:property-placeholder location="classpath:com/jr/spring/spring_dao/jdbc">
</context:property-placeholder>
```

该种方式省去了注册Bean的麻烦。只需要在location属性中声明配置文件的位置。

2.数据源与Jdbc模板进行连接

在Jdbc模板中定义操作的数据源

接口实现类

```
public class UserServiceImp extends JdbcTemplate implements IService {

}
```

接口实现类需要继承JdbcTemplate

注册实现类

```
<bean id="IService" class="com.jr.spring.spring_dao.dao.imp.UserServiceImp">
    <property name="dataSource" ref="driverManagerDataSource"></property>
</bean>
```

实现类继承了JdbcTemplate的属性dataSource，该属性定义Jdbc模板操作的数据源。ref表示注册的数据源

数据库操作

对数据库的增、删、改都是通过update()方法实现的。该刚方法常用的重载方法有两个：

public int update(String sql);

public int update(String sql,Object...args);

返回值为所影响的记录条数。

> 增

```
@Override
public void insert(User user) {
    String sql = "insert into user(name,sex,age) values(?,?,?);";
    System.out.println("insert返回值: " + update(sql, user.getName(), user.getSex(),
user.getAge()));
}
```

> 删

```
@Override
public void delete(User user) {
    String sql = "delete from user where number=?";
    System.out.println("delete返回值: " + this.update(sql, user.getNumber()));
}
```

> 改

```
@Override
public void update(User user) {
    String sql = "update user set name=?,sex=?,age=? where number=?";
    System.out.println(this.update(sql, user.getName(), user.getSex(), user.getAge(),
user.getNumber()));
}
```

> 查

需要关注的是JDBC模板对数据的查询操作。

JDBC模板的查询结果均是以对象的形式返回。根据返回对象类型的不同，可以将查询分为两类：**简单对象查询**与**自定义对象查询。** 简单对象查询：查询结果为String、Integer、等简单对象类型，或该类型做为元素的集合类型，如List<String>等。 自定义对象查询：查询结果为自定义类型，如User等，或改类型作为元素的集合类型，如List<User>等。

- 简单对象查询
    1. 查询结果为String,Integer等简单数据类型

    ```
    @Override
    public String seleteNameById(User user) {
        String sql = "select name from user where number=?";
        return queryForObject(sql, String.class, user.getNumber());
    }
    ```

    2. 查询结果为分装了简单数据类型的List集合

```
@Override
public List<String> selectNames() {
    String sql = "select name from user";
    return queryForList(sql, String.class);
}
```

- 自定义对象查询

  需要将查询结果分装成自定义对象的话，可以通过实现RowMapper接口，将查询的每列数据获取到并封装到自定义对象中。如：

```
public class UserMapper implements RowMapper<User> {
    @Override
    public User mapRow(ResultSet resultSet, int rowNum) throws SQLException {
        int number = resultSet.getInt(1);
        String name = resultSet.getString("name");
        String sex = resultSet.getString("sex");
        int age = resultSet.getInt("age");
        System.out.println("rowNum:" + rowNum);
        return new User(number, name, sex, age);
    }
}
```

每次查询获取到每列数据都会调用mapRow()方法，在该方法中可获取到查询的每列数据，从而对其进行分装成自定义对象并返回。

1. 查询结果为单个对象

```
@Override
public User seleteById(int id) {
    String sql = "select number, name ,sex , age from user where number =?";
    User user = queryForObject(sql, new UserMapper(), id);
    return user;
}
```

2. 查询结果为多个对象集合

```
@Override
public List<User> selectAll() {
    String sql = "select number,name,sex,age from user";
    List<User> users = query(sql, new UserMapper());
    return users;
}
```

事务管理

Spring的事务管理，主要用到了两个事务相关的接口。

事务管理器接口

事务管理器是PlatformTransactionManager接口对象。其主要用于完成事务的提交，回滚，及获取事务的状态信息。

> 常见的两个实现类

PlatformTransactionManager接口有两个常用的实现类：

- DataSourceTranscationManager：使用JDBC或iBatis(MyBatis)进行持久化数据时使用。
- HibernateTransactionManager：使用Hibernate进行持久化数据时使用。

Spring的回滚方式

Spring事务的默认回滚方式是：**发生运行时异常时回滚，发生受查异常时提交。** 不过，对于受查异常，程序员也可以手工设置其回滚方式。通过 ” - “ 异常方式，可使发生指定的异常时事务回滚；通过 ” + “ 异常方式，可使发生指定的异常时事务提交。

事务定义接口

事务定义接口TransactionDefinition中定义了事务描述相关的三类常量：事务隔离级别、事务传播行为、事务默认超时时限，及对它们的操作。

> 事务隔离级别

这些常量均是以 ISOLATION_开头。即形如：ISOLATION_DEFAULT。

- DEFAULT：采用数据库默认的事务隔离级别。Mysql的默认为REPEATABLE_READ; Oracle默认为READ_COMMITTED。
- READ_UNCOMMITTED：读未提交。未解决任何并发问题。
- READ_COMMITTED：读已提交。解决脏读，存在不可重复读与欢读。
- REPEATABLE_READ：可重复读。解决脏读、不可重复读、存在幻读
- SERIALIZABLE：串行化。不存在并发问题。

> 事务传播行为

所谓事务传播行为是指，处于不同事务中的方法在相互调用时，执行期间事务的维护情况。如，A事务中的方法doSome()调用B事务中的doOther()，在调用期间事务的维护情况，就成为事务传播行为。事务传播行为时加在方法上的。 事务传播行为常量都是以PROPAGATION_开头，形如PROPAGATION_REQUIRED。

- REQUIRED

  指定的方法必须在事务内执行。若当前存在事务，就加在当前事务中；若当前没有事务，则创建一个新事务。这种事务传播行为时最常见的选择，也是Spring默认的事务传播行为。

- SUPPORTS

  指定的方法支持当前事务，但若当前有没事务，也可以非事务方式执行。

- MANDATORY

  指定的方法必须在当前事务内执行，若当前没有事务，则直接抛出异常。

- REQUIRES_NEW

  总是新建一个事务，若当前存在事务，就将当前事务挂起，知道新事务执行完毕。

- NOT_SUPPORTED

  指定的方法不能再事务环境中执行。若当前存在事务，就将当前事务挂起。、

- NEVER

指定的方法不能在事务环境下执行，若当前存在事务，就直接抛出异常。

- NESTED

  指定的 方法必须在事务内执行。若当前存在事务，则在嵌套事务内执行；若当前没有事务，则创建一个新事务。

> 事务超时时限

常量TIMEOUT_DEFAULT定义了事务底层默认的超时时限，及不支持事务超时时限设置的nono值。 注意，事务的超时时限起作用的条件比较多，且超时的时间点复杂。所以，该值一般就使用默认即可。

Spring代理工厂管理事务

该方式是，需要为目标类，即Service的实现类创建事务代理。事务代理使用的类是TranscationProxyFractoryBean，该类需要初始化如下一些属性：

- **配置事务管理器**

```
<bean id="myTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="basicDataSource"></property>
</bean>
```

  因为使用的JDBC或者IBatis的持久化数据，故使用的事务管理器类时DataSourceTransactionManager。

- **配置事务代理**

```
<bean id="IStockProcessServiceProxy"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTransactionManager"></property>
    <property name="target" ref="IStockProcessService"></property>
    <property name="transactionAttributes">
        <props>
            <prop key="open*">PROPAGATION_REQUIRED</prop>
            <prop key="find*">PROPAGATION_SUPPORTS,readOnly</prop>
            <prop key="buyStock">PROPAGATION_REQUIRED,-StockException</prop>
        </props>
    </property>
</bean>
```

配置解释：

为目标对象IStockProcessService中以open开头和以find开头的方法名即buyStock方法名注册事务管理。

- open*()方法：设置的事务传播行为PROGAGTION_REQUIRED，即该方法必须在事务下进行，即使调用方法无事务也会重新创建一个事务。
- find*()方法：设置的事务传播行为PROPAGATION_SUPPORTS，即该方法有无事务都能进行。其中readOnly表示该方法对数据库只进行读取操作。
- buyStock()方法：设置的事务传播行为同样为PROPAGATION_SUPPORTS，而-StockException表示当程序运行时出现StockException的自定异常时事务将回滚。

Spring注解管理事务

通过@Transactionl注解方式，也可将事务织入到相应方法中，而使用注解方式，只需要在配置文件中加一个tx标签，以告诉Spring使用注解来完成事务的织入。该标签只需指定一个属性，事务管理器。

- **开启事务注解驱动**

```
<tx:annotation-driven transaction-manager="myTransactionManager">
</tx:annotation-driven>
```

- **在Service实现类中添加事务注解**

```
public class StockProcessServiceImpl_Annotation implements IStockProcessService {
    private IAccount iAccount;
    private IStock iStock;

    public void setiAccount(IAccount iAccount) {
        this.iAccount = iAccount;
    }

    public void setiStock(IStock iStock) {
        this.iStock = iStock;
    }

    @Transactional(propagation = Propagation.REQUIRED)
    @Override
    public void openAccount(String aname, double money) {
        iAccount.insertAccount(aname, money);
    }

    @Transactional(propagation = Propagation.REQUIRED)
    @Override
    public void openStock(String aname, int count) {
        iStock.insertStock(aname, count);
    }

    @Transactional(propagation = Propagation.REQUIRED, rollbackFor =
StockException.class)
    @Override
    public void buyStock(String aname, String sname, double money, int amount)
throws StockException {
        iAccount.updateAccount(aname, money);
        if (1 == 1) {
            throw new StockException("购买股票异常");
        }
        iStock.updateStock(sname, amount);
    }

    @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
    @Override
    public Account findAccount(String aname) {
```

```
        return iAccount.selectAccount(aname);
    }

    @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
    @Override
    public Stock findStock(String sname) {
        return iStock.selectStock(sname);
    }
}
```

通过@Transactionl注解方式将事务织入到相应的方法中。@Transactionl的所有可选属性如下所示：

需要注意的是，@Transcationl若用在方法上，只能用于public方法上。对于其他非public方法，如果加上@Transactionl，虽然Spring不会报错，但不会将指定事务织入到该方法中。因为Spring会忽略所有非public方法上的@Transactionl注解。

若@Transaction注解在类上，则表示该类上的所有方法均将在执行期间织入事务。

- **测试类中指定获取目标对象**

```
@Before
public void before() {
    applicationContext = new
ClassPathXmlApplicationContext("com/jr/spring/spring_dao/applicationContext.xml");
    //使用注解事务管理的代理Service
    iStockProcessService = (IStockProcessService)
applicationContext.getBean("StockProcessServiceImpl_Annotation");
}
```

由于配置文件中不存在事务代理对象，所以测试类中要从容器中获取的将不再时事务代理对象，而是原来的目标对象。

AspectJ管理事务

使用XML的配置事务代理的方式的不足是，每个目标类都需要配置事务代理。当目标类较多，配置文件会表的非常臃肿。 使用XML配置顾问方式可以自动为每个符合切入点表达式的类生成事务代理。

- **配置事务通知**

  为事务通知设置相关属性。用于指定要将事务以什么方式织入给哪些方法。

```
    <tx:advice id="transactionAdvice" transaction-manager="myTransactionManager">
        <tx:attributes>
            <tx:method name="open*" propagation="REQUIRED"/>
            <tx:method name="find*" propagation="SUPPORTS" read-only="true">
</tx:method>
            <tx:method name="buyStock" propagation="REQUIRED" rollback-
for="StockException"></tx:method>
        </tx:attributes>
    </tx:advice>
```

应用到buyStock方法上的事务要求是必须的，且当buyStock方法发生StockException后，要回滚。

- **配置顾问**

  指定将配置好的事务通知，织入给谁

  ```
  <aop:config>
      <aop:pointcut id="myPointcut"
                    expression="execution(*
  com.jr.spring.spring_dao.dao.imp.StockProcessServiceImpl_AspectJ.*(..))">
  </aop:pointcut>
      <aop:advisor advice-ref="transactionAdvice" pointcut-ref="myPointcut">
  </aop:advisor>
  </aop:config>
  ```

  写切入点表达式时需要注意的是，不要同时将事务通知织入Service层和Dao层，因为Service层和Dao层均注入了数据源，而Service又调用了Dao，会出现循环调用的异常。

- **测试类中直接获取目标对象**

  测试类中要从容器获取的将不再是事务代理对象，而是目标对象。

注意

以上三种方式织入事务底层都是通过代理的原理植入事务代码，所以通过application得到的Bean也都是代理对象，所以对于Bean的引用不能是实现，而需要是接口。就像这样：

```
StudentServiceImp studentServiceImp=application =
application.getBean("studentServiceImp");
```

因为事务代理Bean与实现Bean不同类型，但同时实现接口类，所以必须这样写：

```
StudentService studentServiceImp =
application=application.getBean("studentServiceImp");
```

Spring与MyBatis

前言

将MyBatis与Spring进行整合，主要解决的问题就是将SqlSessionFactory对象交由Spring来管理。 SqlSession不能够交由Spring进行管理，SqlSession对象在每次进行操作之后都需要进行关闭（进行事务的提交和关闭），关闭后的SqlSession相当与没用了。所以在每次操作Dao方法时都需要重新创建SqlSession。而交由Spring管理的话，只能在一个Dao对象中存在一个SqlSession对象实例。 所以，该整合，只需将SqlSessionFactory的对象生成器SqlSessionFactoryBean注册在Spring容器中，在将其注入给Dao的实现类即可完成整合。

操作步骤

除了Spribu'bu'zhoung基本的Jar包外，还需要从MyBatis官网中下在Spring与MyBatis真个整合的Jar

链接

> 定义Sql映射文件mapper

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.jr.spring.spring_mybatis.dao.IStudent">
    <insert id="insert" parameterType="Student">
        insert into student(name,age,tid) values(#{name},#{age},#{tid})
    </insert>

    <delete id="delete">
        delete from student where id=#{id}
    </delete>

    <update id="update" parameterType="Student">
        update Student set name=#{name},age=#{age},tid=#{tid}
        <where>id=#{id}</where>
    </update>

    <select id="select" resultType="Student">
        select * from Student
        <where>id=#{id}</where>
    </select>
</mapper>
```

在Dao包下定义与接口名一致的SQL映射文件。

> 定义MyBatis主配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases>
        <package name="com.jr.spring.spring_mybatis.bean"></package>
    </typeAliases>
    <!-- mapping 文件路径配置 -->
    <mappers>
        <package name="com.jr.spring.spring_mybatis.dao"></package>
    </mappers>
</configuration>
```

因为数据源交给了Spring容器管理,主配置中不需要在配置数据源了

> 添加Log4j日志控制文件

> 定义Spring配置文件

- **配置数据源**

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driver}"></property>
    <property name="url" value="${jdbc.url}"></property>
    <property name="username" value="${jdbc.user}"></property>
    <property name="password" value="${jdbc.password}"></property>
</bean>
```

- **注册SqlSessionFactoryBean**

```
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <property name="configLocation"
value="classpath:com/jr/spring/spring_mybatis/mybatis.xml"></property>
    </bean>
```

将数据源和配置文件注入给SqlSessionFactory

- **生成Dao代理对象**

1. 使用MapperFactoryBean生成Dao代理对象

   由于使用Mapper动态代理方式没有Dao实现类，所以Dao的实现类对象有代理工厂生成.

```
    <bean id="studentDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
        <property name="mapperInterface"
value="com.jr.spring.spring_mybatis.dao.IStudent"></property>
    </bean>
```

2. 自动扫描的Mapper动态代理

   前面的方式在动态生成代理时存在一个缺点：MapperFactoryBean一次只能生成一个代理对象，即若有多个Dao接口需要代理对象，则需要配置多个Mapper动态代理。这样会使得配置文件变得臃肿。

而支持扫描的Mapper动态代理则会避免以上的缺点，其会对所配置的基础包所有的接口生成Mapper动态代理。在向Service注入Dao的实现类时，直接传入Dao的接口名即可。

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory">
</property>
    <property name="basePackage" value="com.jr.spring.spring_mybatis.dao">
</property>
</bean>
```

- **向Service注入接口名**

```
<bean id="studentService"
class="com.jr.spring.spring_mybatis.service.imp.StudentServiceImp">

<!--通过Mapper扫描配置器生成Mapper动态代理对象-->
    <property name="iStudent" ref="IStudent"></property>
</bean>
```

若使用Mapper扫描配置器生成的Mapper代理对象，则Mapper代理对象是没有名称的，所以向Service注入时可通过接口的简单类名注入

注意

如果通过Mapper扫描器自动生成Dao的代理实现类并注册给Spring，在注入给Service的时候，ref填写的对象需要注意的是：1.如果接口名前面两个都是大写字母的话，ref写的是接口的**简单类名**(即：接口名)；2.如果接口名开头只存在一个大写字母，那么ref写的是接口名的简单类名但**首字母小写。**

Spring与Web

前言

在Web项目中使用Spring框架，首先要解决在Web层（这里值Servlet）中获取Spring容器的问题。只要在web层获取到了Spring容器，便可容器中获取到Service对象。

获取Spring方式

> 直接在Htpp请求相应方法中直接获取Spring容器

请求Servlet

```
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ApplicationContext context=new ClassPathXmlApplicationContext("Spring配置文件路
径");
    }
}
```

将Spring容器的创建语句放在Servlet的doGet()或doPost()方法中是有问题的，因为每次的请求都会创建一个新的Spring容器,对于一个应用来说，只需要一个Spring容器即可,多余的将会耗费环境的资源。

此时，可以考虑将Spring容器的创建放在Servlet进行初始化进行，即执行init()方法时进行。并且，Servlet还是单例多线程的，即一个业务只有一个Servlet实例，所以执行该业务的用户执行的都是这一个Servlet实例。这样，Spring容器就具有了唯一性了。

但是，Servlet是一个业务一个Servlet实例，即LoginServlet只有一个，但还会有StudentServlet，TeacherServlet等。每个业务都会有一个Servlet都会执行自己的init()方法，也就都会创建一个Spring容器，这样依赖，Spring容器就又不唯一了。

> 使用Spring的Web插件

对于Web应用来说，ServletContext对象是唯一的，一个Web应用只有一个ServletContext对象，该对象是在Web应用装载时初始化的。若将Spring容器的创建时机放在ServletContext初始化时，就可以保证Spring容器的创建只执行一次，也就保证了Spring容器在整个应用中的唯一性。

当Spring容器创建好后，在整个应用的生命周期过程中，Spring容器应该时随时可以被访问的。即，Spring容器应具有全局性。而ServletContext对象的域属性，就具有应用的全局性。所以，将创建好的Spring容器，以域属性的形式放入到ServletContext的域空间中，就保证了Spring容器的全局性。

- **导入Jar包**

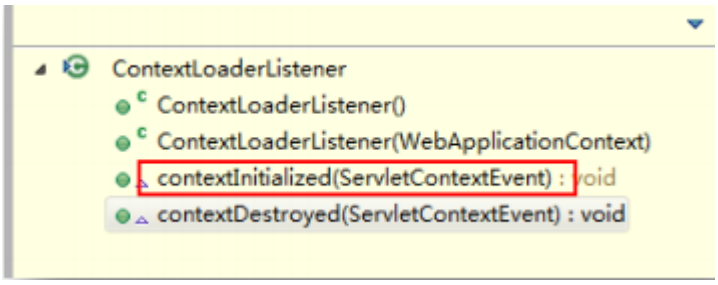
  在Web项目中使用Spring，需要导入Spring对Web的支持包：Spring-web-4.2.1.RELEASE.jar

- **注册监听器**

若要在ServletContext初始化时创建Spring容器，就需要使用监听器接口ServletContextListenner对ServletContext进行监听。在web.xml中注册该监听器。 Spring为该监听器接口定义了一个实现类ContextLoaderListenerm。

```
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
```

打开ContextLoaderLitener的源码。看到一共四个方法，两个构造方法，一个初始话方法，一个销毁方法。



在这四个方法中较重要的方法应该就是contextInitialized()，context(Spring容器)初始化方法。

跟踪initWebApplicationContext()方法，可以看到，在其中创建了容器对象

```
try {
    // Store context in local instance variable, to guarantee that
    // it is available on ServletContext shutdown.
    if (this.context == null) {
        this.context = createWebApplicationContext(servletContext);
    }
    if (this.context instanceof ConfigurableWebApplicationContext) {
```

创建好的容器对象放入到了ServletContext的域属性空间中，Key为一个常量；WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE。

```
            }
            configureAndRefreshWebApplicationContext(cwac, servletContext);
        }
    }
    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);

    ClassLoader ccl = Thread.currentThread().getContextClassLoader();
    if (ccl == ContextLoader.class.getClassLoader()) {
        currentContext = this.context;
    }
```

- **指定Spring配置文件的位置**

  Spring容器的创建需要加载配置文件，那么ContextLoaderListener在对Spring容器的创建时需要指定Spring配置文件的位置。

  其默认的Spring配置文件位置与名称为：WEB-INF/applicationContext.xml。即不配置参数的话，ContextLoaderLitener会根据该路径和文件名加载配置文件，找不到将抛出异常。

  但，一般会将配置文件放在项目的classpath下，即src下，所以需要在web.xml中对Spring配置文件的位置及名称进行指定。

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

从监听器ContextLoaderListener的父类ContextLoader的源码中可以看到其要读取的配置文件参数名称contextConfigLocation。

```
104     */
105    public static final String CONFIG_LOCATION_PARAM = "contextConfigLocation";
106
107    /**
108     * Config param for the root WebApplicationContext implementation class to us
```

所以<context-param/>的key必须指定为contextConfigLocation

- 获取Spring容器对象

在Servlet中获取容器对象的常用方式有两种：

1. 直接从ServletContext中获取

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

    String attr=WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE;
    WebApplicationContext ac=
(WebApplicationContext)this.getServletContext().getAttribute(attr);
}
```

该方法需要记住获取ServetContext属性的key，相对来说比较不方便。

2. 通过WebApplicationContextUtils获取

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

    WebApplicationContext
ac=WebApplicationContextUtils.getRequiredWebApplicationContext (this.getServletCont
ext()) ;
}
```