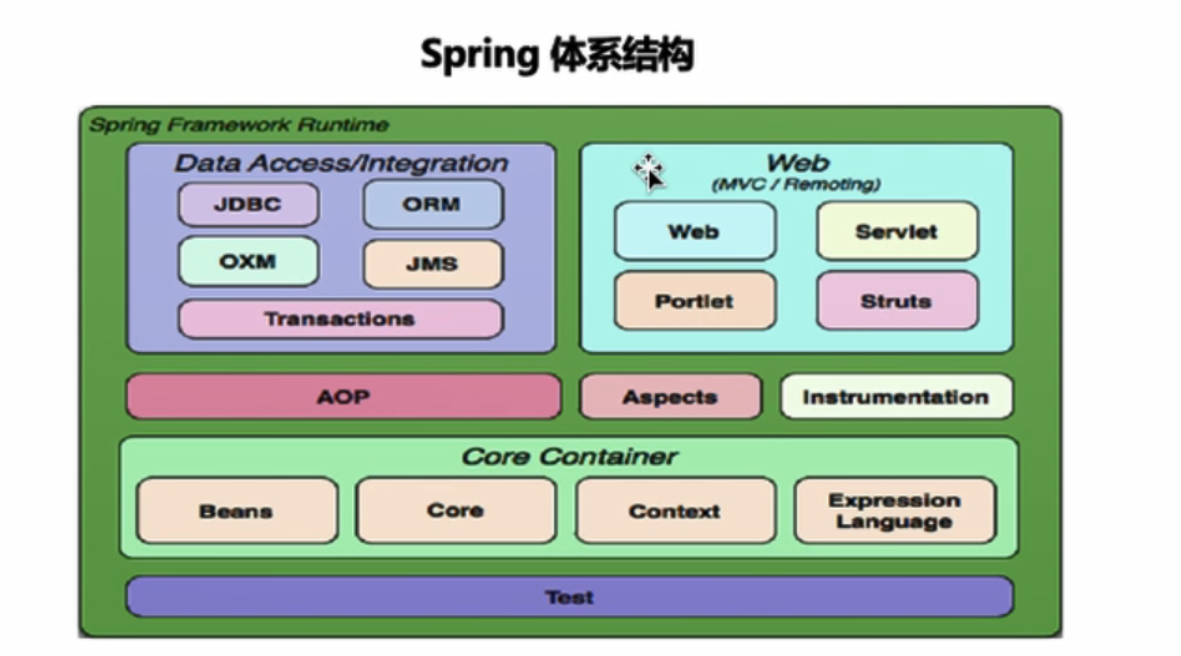
# Spring 学习总结和记录

## 一、概述




Spring两大核心机制

- **IoC（控制反转：Inversion of Control）**/DI（依赖注入：Dependency Injection）

  IoC产生对象、DI负责将对象之间做一个相关联的注入

- **AOP（面向切面编程：Aspect Oriented Programming)**

Spring：

，是**软件设计层面**的框架。他的优势是可以将应用程序进行分层，开发者可以自主选择组件。如：

- MVC：Structs2、Spring MVC
- ORMapping：Hibernate、MyBatis、Spring Data

Spring本身提供了各个层面的解决方案，比如Spring MVC、Spring Data、Spring Cloud

优点：

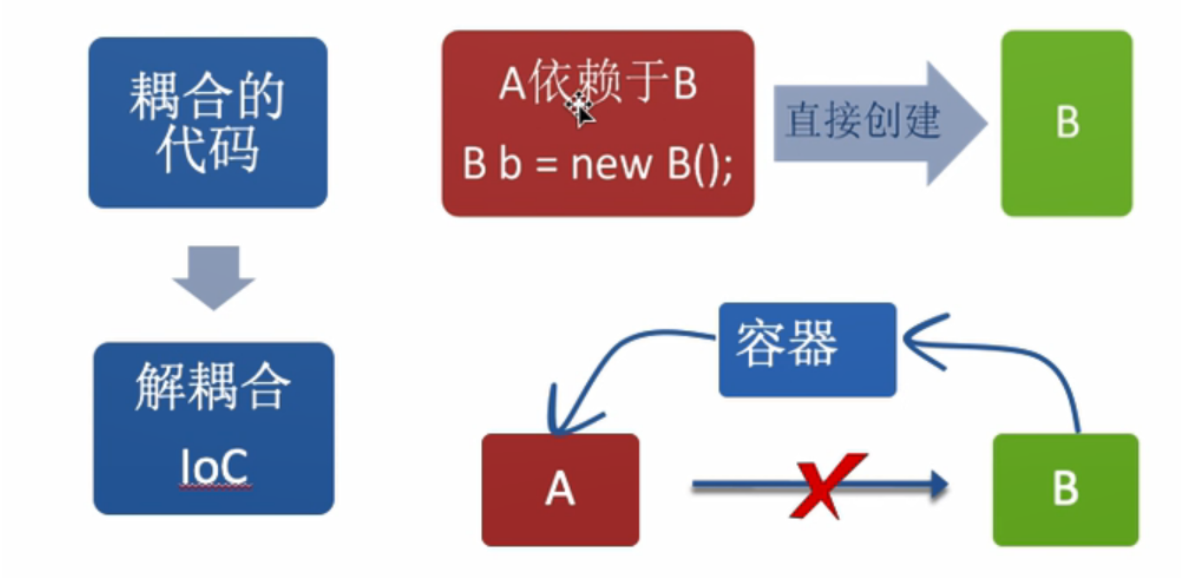
- 低浸入式设计
- 独立于各种应用服务器
- 依赖注入组件将组件关系透明化，降低耦合度
- 面向切面编程的特性允许将通用任务进行集中式处理
- 与第三方框架的良好整合

# 二、Spring IoC（控制反转）

## 1、什么是控制反转

在传统的程序开发中，需要调用对象时，通常由调用者来春卷关键被调用者的实例，即对象是由调用者主动 new 出来的。

但在Spring框架中创建对象的工作不再由调用者来完成，而是交给IoC容器来创建，再推送给调用者，整个流程完成反转。




## 2、IoC的使用方法

第一步，创建Maven工程，pom.xml添加依赖。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
```

```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.gloryh</groupId>
    <artifactId>spring-demo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

    </dependencies>

</project>
```

第二步，创建实体类

```
package com.gloryh.entity;

import lombok.Data;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
public class Student {
    private long id;
    private String name;
    private int age;
}
```

- 传统的开发方式（需要手动 new Student）：

```
Student student = new Student();
student.setId(1);
student.setName("张三");
student.setAge(19);
System.out.println(student);
```

- 通过IoC创建对象（在配置文件中添加需要管理的对象，XML格式文件，文件名自定义）：

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

xmlns:context="http://www.springframework.org/schem
a/context"

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

xsi:schemaLocation="http://www.springframework.org/
schema/beans
http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd
   http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/sprin
g-context-4.3.xsd
">
    <bean id="student"
class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三">
</property>
    </bean>
</beans>
```

  获取对象：
```

```
//加载配置文件
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
Student student=(Student)
applicationContext.getBean("student");
System.out.println(student);
```

## 3、IoC配置文件

IoC是通过配置bean标签完成对对象的管理。bean标签主要配置字段有：

- id：对象名
- class：对象的模板类（实体类）。注：所有交给IoC来管理的类必须有无参构造函数，否则会报错。因为Spring底层是通过反射机制来创建对象，而反射机制调用的是无参构造。

对象成员变量通过property标签完成赋值。property标签主要配置字段有：

- name：成员变量名
- value：成员变量的值。注：变量值为基本数据类型、String时可以直接赋值,如果是其他引用类型，作为需要通过ref来实现赋值。
- ref：将IoC中的另外一个bean赋值给当前成员变量（DI）。如下：

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

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

xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd
```

```
   http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-4.3.xsd
">
    <bean id="student" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="address" ref="address"></property>
    </bean>
    <bean id="address" class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
</beans>
```

- 集合注入list标签：用于list集合的注入，如：

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private long id;
    private String name;
    private int age;
    private List<Address> addresses ;
```

```
}
```

因为是一对多的关系，此时就不能使用`<property name="address" ref="address"></property>`这样的注入方式，而要使用list标签，多个ref注入。如：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student"
class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三">
</property>
        <property name="addresses">
            <list>
                <ref bean="address"></ref>
                <ref bean="address2"></ref>
            </list>
        </property>
    </bean>

    <bean id="student2"
class="com.gloryh.entity.Student">
        <constructor-arg index="0" value="2">
</constructor-arg>
        <constructor-arg index="2" value="20">
</constructor-arg>
        <constructor-arg index="1" value="李四">
</constructor-arg>
        <constructor-arg index="3" ref="address">
</constructor-arg>
```

```
    </bean>
    <bean id="student3"
class="com.gloryh.entity.Student">
        <property name="name" value="王五">
</property>
    </bean>
    <bean id="address"
class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
    <bean id="address2"
class="com.gloryh.entity.Address">
        <property name="name" value="河南省开封市">
</property>
        <property name="id" value="2"></property>
    </bean>
</beans>
```

# 4、IoC底层原理

## 1.读取配置文件，解析XML。

```
package com.gloryh.ioc;

import com.gloryh.entity.Student;

/**
 * 测试手写的ioc实现原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
public class Test {
  public static void main(String[] args) {
    ApplicationContext applicationContext =new
ClassPathXmlApplicationContext("spring.xml");
    Student student =(Student)
applicationContext.getBean("student");
    System.out.println(student);
```

```
  }
}
```

```
package com.gloryh.ioc;

/**
 * 手动实现IoC实现原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
public interface ApplicationContext {
    public Object getBean(String id);
}
```

```
package com.gloryh.ioc;

import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;

import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * Application接口实现类，用于手动实现IoC原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
public class ClassPathXmlApplicationContext implements
ApplicationContext {
  private Map<String, Object> ioc = new HashMap<String,
Object>();

  public ClassPathXmlApplicationContext(String path) {
    try {
      SAXReader saxReader = new SAXReader();
```

```
        Document document =
saxReader.read("./src/main/resources/" + path);
        Element root = document.getRootElement();
        Iterator<Element> iterator = root.elementIterator();
        while (iterator.hasNext()) {
          Element element = iterator.next();
          String id =element.attributeValue("id");
          String className =
element.attributeValue("class");
          System.out.println(id);
          System.out.println(className);
          //接下来通过反射机制创建对象
          Class clazz = Class.forName(className);
          //获取无参构造函数
          Constructor constructor=clazz.getConstructor();
          System.out.println(constructor);
          //调用函数获取对象
          Object object=constructor.newInstance();
          System.out.println(object);
          //存入数据
          ioc.put(id,object);
        }
        System.out.println(document);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    public Object getBean(String id) {
      return ioc.get(id);
    }
}
```

## 2.通过反射机制来实例化配置文件中所配置的所有的bean。

测试方法：

```
package com.gloryh.ioc;

import com.gloryh.entity.Student;
```

```
/**
 * 测试手写的ioc实现原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
public class Test {
  public static void main(String[] args) {
    ApplicationContext applicationContext =new
ClassPathXmlApplicationContext("spring.xml");
    Student student =(Student)
applicationContext.getBean("student");
    System.out.println(student);
  }
}
```

IoC接口类和实现类：

```
package com.gloryh.ioc;

/**
 * 手动实现IoC实现原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
public interface ApplicationContext {
  /**
   * @param id
   * @return Object
   */
  public Object getBean(String id);
}
```

```
package com.gloryh.ioc;

import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
```

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * Application接口实现类，用于手动实现IoC原理
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
public class ClassPathXmlApplicationContext implements
ApplicationContext {
  private Map<String, Object> ioc = new HashMap<String,
Object>();

  public ClassPathXmlApplicationContext(String path) {
    try {
      SAXReader saxReader = new SAXReader();
      Document document =
saxReader.read("./src/main/resources/" + path);
      Element root = document.getRootElement();
      Iterator<Element> iterator = root.elementIterator();
      while (iterator.hasNext()) {
        Element element = iterator.next();
        String id =element.attributeValue("id");
        String className =
element.attributeValue("class");
        System.out.println("当前bean的id为："+id);
        System.out.println("对应的实体类全限定类名
为："+className);
        //接下来通过反射机制创建对象
        Class clazz = Class.forName(className);
        //获取无参构造函数
        Constructor constructor=clazz.getConstructor();
        System.out.println("当前实体类的为无参构造方法
为："+constructor);
        //调用函数创建目标对象
```

```
            Object object=constructor.newInstance();
            System.out.println("当前实体类的初始化对象
为："+object);
            //给目标对象赋值（遍历下一个节点）
            Iterator<Element> beanIterator
=element.elementIterator();
            while (beanIterator.hasNext()){
              Element property =beanIterator.next();
              String name =property.attributeValue("name");
              String valueStr
=property.attributeValue("value");
              //取出ref标签
              String ref=property.attributeValue("ref");
              if(ref == null){
                System.out.println("由于当前property的ref标签为
null，执行普通赋值操作");
                getValue(name,valueStr,clazz,object);
              }else{
                System.out.println("由于当前property的ref标签不为
null，需要加载bean");
                //这里可以使用递归进行赋值，或者在getBean方法中调用进
行赋值
              }
              ioc.put(id,object);
            }

          }
          System.out.println(ioc);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }

      public Object getBean(String id) {
        return ioc.get(id);
      }
      public Method getValue(String name ,String
valueStr,Class clazz,Object object){
        //需要将name对应字符串首字符变大写，然后添加set（如id对应构造方
法setId（））
```

```
    String methodName
="set"+name.substring(0,1).toUpperCase()+name.substring(1)
;
    System.out.println("当前参数对应的set方法为
为："+methodName);
    //通过反射获取方法以及所需要的参数类型
    Field field = null;
    try {
      field = clazz.getDeclaredField(name);
    } catch (NoSuchFieldException e) {
      e.printStackTrace();
    }
    System.out.println("当前set方法的参数类型
为："+field.getType().getName());
    Method method = null;
    try {
      method =
clazz.getDeclaredMethod(methodName,field.getType());
    } catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
    System.out.println("当前实体类的完整set方法名
为："+method);
    //给方法赋值,直接使用valueStr的话会抛出异常（类型不匹配），因为
valueStr为String类型。
    //要根据成员变量类型对valueStr进行数据转换
    Object value=null;
    if(field.getType().getName() == "long"){
      value  =Long.parseLong(valueStr);
    }
    if(field.getType().getName() == "int"){
      value = Integer.parseInt(valueStr);
    }
    if (field.getType().getName() == "java.lang.String"){
      value=valueStr;
    }
    //类型转换完成，进行赋值
    try {
      method.invoke(object,value);
    } catch (IllegalAccessException e) {
      e.printStackTrace();
```

```
    } catch (InvocationTargetException e) {
      e.printStackTrace();
    }
    return method;
  }
}
```

## 3. 获取bean的方式

### 1.通过id获取

```
package com.gloryh.test;

import com.gloryh.entity.Student;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * TODO
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
public class Test {
  public static void main(String[] args) {
    //加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
    Student student=(Student)
applicationContext.getBean("student");
    System.out.println(student);
  }
}
```

### 2.通过时类获取

```
package com.gloryh.test;

import com.gloryh.entity.Student;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * TODO
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
public class Test {
  public static void main(String[] args) {
    //加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
    Student student=
applicationContext.getBean(Student.class);
    System.out.println(student);
  }
}
```

注：这种获取的方式有一个弊端，那就是在配置文件中，此类对应的bean只能有一个，有多个的话就会出错。

例如（此时spring.xml中有两个bean对应实体类Student.class）：

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

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

xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd
```

```
   http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-4.3.xsd
">
    <bean id="student" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="address" ref="address"></property>
    </bean>
    <bean id="address" class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
    <bean id="student2" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="address" ref="address"></property>
    </bean>
</beans>
```

此时运行Test就会报错：

```
Exception in thread "main"
org.springframework.beans.factory.NoUniqueBeanDefinitionEx
ception:
No qualifying bean of type 'com.gloryh.entity.Student'
available: expected single matching bean but found 2:
student,student2
```

而通过id进行赋值就存在唯一性，不会出现此异常。

## 4.调用有参构造创建bean

首先在实体类中要有对应的有参构造方法：

- 通过注释创建

```
package com.gloryh.entity;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private long id;
    private String name;
    private int age;
    private Address address ;
}
```

@AllArgsConstructor有参构造

@NoArgsConstructor无参构造

- 通过方法创建

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
public class Student {
    private long id;
    private String name;
```

```
    private int age;
    private Address address ;
    public Student(long id, String name, int age, Address
address) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

然后配置文件（参考spring.xml中的student2）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="address" ref="address"></property>
    </bean>
    <bean id="address" class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
    <bean id="student2" class="com.gloryh.entity.Student">
        <constructor-arg name="id" value="2">
</constructor-arg>
        <constructor-arg name="name" value="李四">
</constructor-arg>
        <constructor-arg name="age" value="20">
</constructor-arg>
```

```
        <constructor-arg name="address" ref="address">
</constructor-arg>
    </bean>
    <bean id="student2" class="com.gloryh.entity.Student">
        <property name="name" value="王五"></property>

    </bean>
</beans>
```

name字段可以省略不写，省略的话，按照构造的方法中字段的顺序进行赋值。

```
<bean id="student2" class="com.gloryh.entity.Student">
    <constructor-arg  value="2"></constructor-arg>
    <constructor-arg  value="李四"></constructor-arg>
    <constructor-arg  value="20"></constructor-arg>
    <constructor-arg  ref="address"></constructor-arg>
</bean>
```

位置不匹配可能会造成赋值不匹配或者类型不匹配而抛出异常，如：

```
<bean id="student2" class="com.gloryh.entity.Student">
    <constructor-arg  value="2"></constructor-arg>
    <constructor-arg  value="20"></constructor-arg>
    <constructor-arg  value="李四"></constructor-arg>
    <constructor-arg  ref="address"></constructor-arg>
</bean>
```

异常如下：

```
警告: Exception encountered during context initialization
- cancelling refresh attempt:
org.springframework.beans.factory.UnsatisfiedDependencyExc
eption: Error creating bean with name 'student2' defined
in class path resource [spring.xml]: Unsatisfied
dependency expressed through constructor parameter 2:
Could not convert argument value of type
[java.lang.String] to required type [int]: Failed to
convert value of type 'java.lang.String' to required type
'int'; nested exception is
java.lang.NumberFormatException: For input string: "李四"
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyExc
eption: Error creating bean with name 'student2' defined
in class path resource [spring.xml]: Unsatisfied
dependency expressed through constructor parameter 2:
Could not convert argument value of type
[java.lang.String] to required type [int]: Failed to
convert value of type 'java.lang.String' to required type
'int'; nested exception is
java.lang.NumberFormatException: For input string: "李四"
```

当我们不使用name字段进行属性字段匹配时，不按顺序写就会抛出此异常。此时也可以通过下标的方式解决。如：

```
<bean id="student2" class="com.gloryh.entity.Student">
    <constructor-arg index="0"  value="2"></constructor-
arg>
    <constructor-arg index="2" value="20"></constructor-
arg>
    <constructor-arg index="1" value="李四"></constructor-
arg>
    <constructor-arg index="3" ref="address">
</constructor-arg>
</bean>
```

# 5、Spring IoC的特性

## 1.scope作用域

Spring 管理的bean是根据 scope 来生成的，表示bean的作用域，一共4种：

- singleton：单例，表示通过 IoC 容器获取的bean是唯一的。
- prototype：原型，表示通过 IoC 容器获取的bean是不同的。
- request：请求，表示在一次HTTP请求内有效。
- session：会话，表示在一次用户会话内有效。

request 和 session 只适用于 Web 项目，大多数情况下都是使用单例和原型。

prototype模式时，当业务代码获取IoC容器中的bean时，Spring才会去调用对应实体类的无参构造方法去创建对应的bean。

- 优点：当不需要使用bean时，可以不创建bean，节省资源。
- 缺点：当多次使用bean时，会创建多个相同的bean，浪费内存。

singleton模式时，无论业务代码是否获取/使用IoC容器里的bean，Spring在加载配置文件时（spring.xml）都会创建bean。

- 优点：当多次使用bean时，可以只创建一个bean，节省内存。
- 缺点：当不需要使用bean时，也会创建bean，浪费资源。

## 2.Spring的继承

与Java的继承不同的是，Java是类层面的继承，子类可以继承父类的内部结构信息；而Spring是对象层面的继承，子对象可以继承父对象的属性值。

测试方法：

```
package com.gloryh.test;

import com.gloryh.entity.Student;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * TODO
 *
```

```
 * @author 黄光辉
 * @since 2020/8/26
 */
public class Test {
  public static void main(String[] args) {
    //加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
    Student student=(Student)
applicationContext.getBean("student");
    System.out.println("student:"+student);
    Student student2=(Student)
applicationContext.getBean("student2");
    System.out.println("student2:"+student2);

  }
}
```

继承前（student和student2）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="addresses">
            <list>
                <ref bean="address"></ref>
                <ref bean="address2"></ref>
            </list>
        </property>
    </bean>
```

```
    <bean id="student2" class="com.gloryh.entity.Student">
    </bean>
    <bean id="address" class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
    <bean id="address2" class="com.gloryh.entity.Address">
        <property name="name" value="河南省开封市">
</property>
        <property name="id" value="2"></property>
    </bean>
</beans>
```

运行结果：

```
"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
student:Student(id=1, name=张三, age=19, addresses=[Address(id=1, name=河南省郑州市), Address(id=2, name=河南省开封市)])
student2:Student(id=0, name=null, age=0, addresses=null)

进程已结束,退出代码0
```

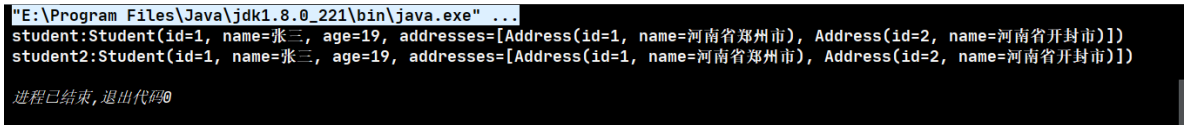
继承的方法是给继承的bean添加parent字段值。如下（student和student2）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student">
        <property name="age" value="19"></property>
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="addresses">
            <list>
                <ref bean="address"></ref>
                <ref bean="address2"></ref>
            </list>
        </property>
    </bean>
```

```
    <bean id="student2" class="com.gloryh.entity.Student"
parent="student">
    </bean>
    <bean id="address" class="com.gloryh.entity.Address">
        <property name="name" value="河南省郑州市">
</property>
        <property name="id" value="1"></property>
    </bean>
    <bean id="address2" class="com.gloryh.entity.Address">
        <property name="name" value="河南省开封市">
</property>
        <property name="id" value="2"></property>
    </bean>
</beans>
```

继承后运行结果：



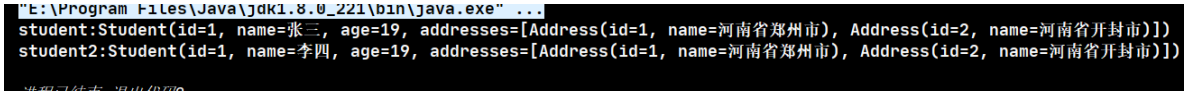

在继承的基础上还可以进行覆盖，比如将name修改为李四：

xml文件中：

```
<bean id="student" class="com.gloryh.entity.Student">
    <property name="age" value="19"></property>
    <property name="id" value="1"></property>
    <property name="name" value="张三"></property>
    <property name="addresses">
        <list>
            <ref bean="address"></ref>
            <ref bean="address2"></ref>
        </list>
    </property>
</bean>

<bean id="student2" class="com.gloryh.entity.Student"
parent="student">
    <property name="name" value="李四"></property>
</bean>
```

运行结果：




不同的类也可以进行继承（User和Student）：

Student.Class

```
package com.gloryh.entity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
    private long id;
    private String name;
    private int age;
    private List<Address> addresses ;
}
```

User.Class

```
package com.gloryh.entity;
import lombok.Data;
import java.util.List;
/**
 * User实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
```

```
@Data
public class User {
    private long id;
    private String name;
    private int age;
    private List<Address> addresses;
}
```

xml文件中编写继承关系

```
<bean id="student" class="com.gloryh.entity.Student">
    <property name="age" value="19"></property>
    <property name="id" value="1"></property>
    <property name="name" value="张三"></property>
    <property name="addresses">
        <list>
            <ref bean="address"></ref>
            <ref bean="address2"></ref>
        </list>
    </property>
</bean>
<bean id="user" class="com.gloryh.entity.User"
parent="student">
</bean>
```

测试方法:

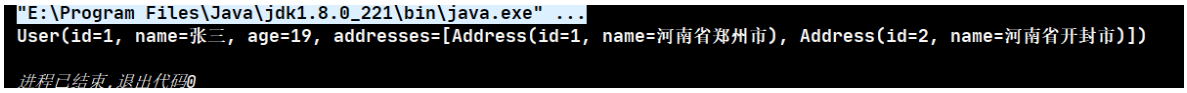
```
package com.gloryh.test;

import com.gloryh.entity.User;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * TODO
 *
 * @author 黄光辉
 * @since 2020/8/26
```

```
 */
public class Test {
  public static void main(String[] args) {
    //加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
    User  user=(User) applicationContext.getBean("user");
    System.out.println(user);
  }
}
```

运行结果：




注：继承的类必须含有被继承类的属性（只多不少）。例如：

```
<bean id="address" class="com.gloryh.entity.Address">
    <property name="name" value="河南省郑州市"></property>
    <property name="id" value="1"></property>
</bean>
<bean id="user" class="com.gloryh.entity.User"
parent="address">
</bean>
```

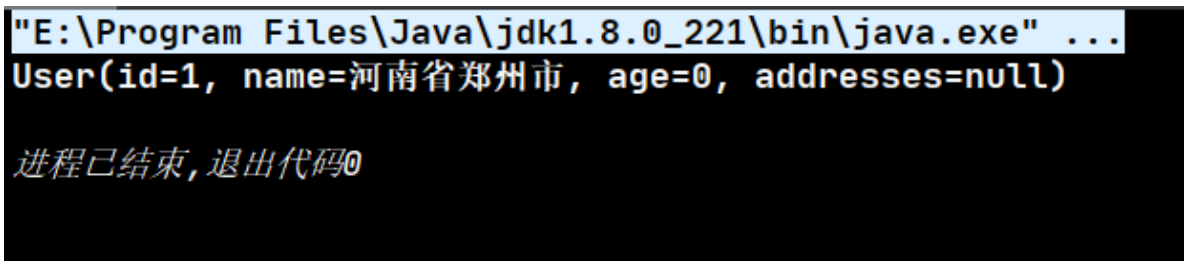
Address.class（只对应User里的id和name）

```
package com.gloryh.entity;

import lombok.Data;

/**
 * 地址实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
@Data
public class Address {
    private long id;
    private String name;
}
```

运行结果：




所以，Spring 的继承关注的在于具体的对象，而不在于类。即两个不同的类的实例化对象可以完成继承，前提是子对象必须含有父对象的所有属性，同时可以在此基础上添加其他的属性（参考User和Address的继承例子）。

## 3.Spring的依赖

与继承类似，依赖也是描述 bean 和 bean 之间的一种关系，配置依赖之后，被依赖的 bean 一定先创建，然后再创建依赖的 bean（A 依赖于 B ，则先创建B ，再创建 A）。

Student.class

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;

import java.util.List;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
@Data
@AllArgsConstructor
public class Student {
  private long id;
  private String name;
  private int age;
  private List<Address> addresses;
```

```
  public Student() {
    System.out.println("创建了Student对象");
  }
}
```

User.class

```
package com.gloryh.entity;

import lombok.Data;

import java.util.List;

/**
 * User实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
@Data
public class User {
  private long id;
  private String name;
  private int age;
  private List<Address> addresses;

  public User() {
    System.out.println("创建了User对象");
  }
}
```

测试方法:

```
package com.gloryh.test;


import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;
```

```
/**
 * 测试2
 *
 * @author 黄光辉
 * @since 2020/8/26
 **/
public class Test2 {
  public static void main(String[] args) {
      ApplicationContext applicationContext =new
ClassPathXmlApplicationContext("spring-dependency.xml");

  }
}
```

默认情况下，会按照bean的顺序从上到下执行创建。

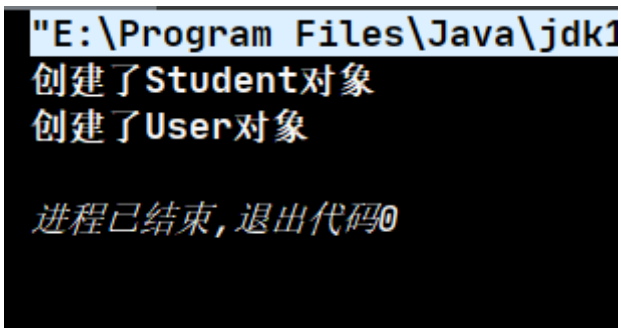
spring-dependency.xml中：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student">
</bean>

    <bean id="user" class="com.gloryh.entity.User"></bean>
</beans>
```

执行结果：



依赖之后会先创建被依赖对象，在创建依赖对象（student依赖于user）。

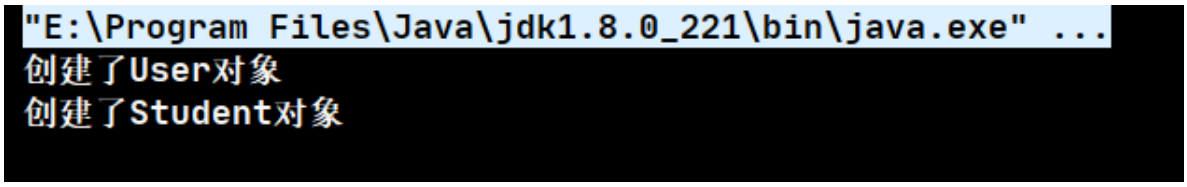
spring-dependency.xml中（关键字：depends-on）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student"
depends-on="user"></bean>

    <bean id="user" class="com.gloryh.entity.User"></bean>
</beans>
```

运行结果：




## 4.Spring 的 p 命名空间

p 命名空间是对 IoC / DI 的简化操作，使用 p 命名空间可以更加方便的完成本案的配置以及bean之间的依赖注入。即 p 命名空间可以代替property的作用。

spring-p.xml（p命名空间用法）

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

xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd">
    <bean id="student" class="com.gloryh.entity.Student"
p:id="1" p:name="张三" p:age="20" p:user-ref="user">
</bean>

    <bean id="user" class="com.gloryh.entity.User"
p:id="2" p:name="李四" p:age="19"></bean>
</beans>
```

Student.class

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;

/**
 * 学生实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
@Data
@AllArgsConstructor
public class Student {
  private long id;
  private String name;
  private int age;
  private User user;
  public Student() {
    System.out.println("创建了Student对象");
  }
```

```
}
```

User.class

```
package com.gloryh.entity;

import lombok.Data;

/**
 * User实体类
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
@Data
public class User {
  private long id;
  private String name;
  private int age;

  public User() {
    System.out.println("创建了User对象");
  }
}
```

测试方法

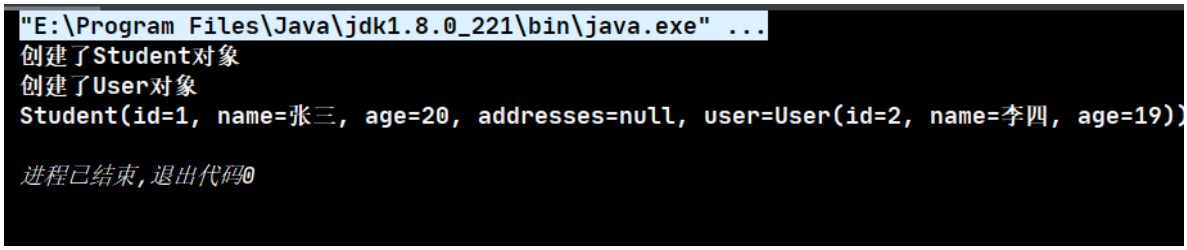
```
package com.gloryh.test;

import com.gloryh.entity.Student;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * 测试3
 *
 * @author 黄光辉
 * @since 2020/8/26
 */
```

```
public class Test3 {
  public static void main(String[] args) {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring-p.xml");
    Student student = (Student)
applicationContext.getBean("student");
    System.out.println(student);
  }
}
```

运行结果：




# 三、Spring IoC 的工厂模式

IoC通过工厂模式创建 bean 有两种方式：

- 静态工厂方法
- 实例工厂方法

## 1、静态工厂方法

测试实体类（Car）

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * Car实体类
 *
 * @author 黄光辉
 * @since 2020/8/27
 **/
@Data
```

```
@AllArgsConstructor
@NoArgsConstructor
public class Car {
    private long id;
    private String name;
}
```

## 1.普通的静态工厂方法

以Car为例，我们的工厂办法中使用一个Map集合以键值对方式存值和取值，创建一个静态代码块来作为工厂生产我们的Car实体类。然后通过getCar方法，用 id 的值从Map总取值。

静态工厂方法实现：

```
package com.gloryh.factory;

import com.gloryh.entity.Car;

import java.util.HashMap;
import java.util.Map;

/**
 * 静态工厂类
 *
 * @author 黄光辉
 * @since 2020/8/27
 **/
public class StaticCarFactory {
    private static Map<Long, Car> cars;
    static{
        cars = new HashMap<Long, Car>();
        cars.put(1L,new Car(1L,"宝马"));
        cars.put(2L,new Car(2L,"奔驰"));
    }
    public static Car getCar(Long id){
        return cars.get(id);
    }
}
```

测试方法：

```java
package com.gloryh.test;

import com.gloryh.entity.Car;
import com.gloryh.factory.StaticCarFactory;

/**
 * 测试四
 *
 * @author 黄光辉
 * @since 2020/8/27
 */
public class Test4 {
  public static void main(String[] args) {
    // 调用静态工厂类，获取car
    Car car = StaticCarFactory.getCar(1L);
    System.out.println(car);
  }
}
```

## 2.将静态工厂方法交给IoC管理

在 spring 配置文件中配置（spring-factory.xml）：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <!-- 配置静态工厂创建Car -->
    <bean id="car"
class="com.gloryh.factory.StaticCarFactory" factory-
method="getCar">
        <constructor-arg value="1"></constructor-arg>
    </bean>
</beans>
```

测试方法：

```
package com.gloryh.test;

import com.gloryh.entity.Car;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * 测试四
 *
 * @author 黄光辉
 * @since 2020/8/27
 */
public class Test4 {
  public static void main(String[] args) {
    // 调用IoC配置的工厂类获取bean
    ApplicationContext applicationContext=new
ClassPathXmlApplicationContext("spring-factory.xml");
    Car car =(Car) applicationContext.getBean("car");
    System.out.println(car);
  }
}
```

# 2、实例工厂方法

## 1.普通的实例工厂方法

同样以Car实体类为例，此时创建的为实例工厂方法来创建Car实体类，我们依然通过键值对方式来进行存取，但是因为是实例方法而不是静态方法，所以可以通过该工厂方法的无参构造方法来创建Car，并通过getCar来获取。

实例工厂实现方法：

```
package com.gloryh.factory;

import com.gloryh.entity.Car;

import java.util.HashMap;
import java.util.Map;
```

```
/**
 * 实例工厂方法
 *
 * @author 黄光辉
 * @since 2020/8/27
 */
public class InstanceCarFactory {
  private Map<Long, Car> cars;

  public InstanceCarFactory() {
    cars = new HashMap<Long, Car>();
    cars.put(1L, new Car(1L, "宝马"));
    cars.put(2L, new Car(2L, "奔驰"));
  }

  public Car getCar(Long id) {
    return cars.get(id);
  }
}
```

测试方法:

```
package com.gloryh.test;

import com.gloryh.entity.Car;
import com.gloryh.factory.InstanceCarFactory;

/**
 * 测试四
 *
 * @author 黄光辉
 * @since 2020/8/27
 */
public class Test4 {
  public static void main(String[] args)
    //1.实例化工厂类
    InstanceCarFactory instanceCarFactory = new
InstanceCarFactory();
    //2.通过id获取实体类
    Car car=instanceCarFactory.getCar(1L);
```

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

## 2.将实例工厂方法交给 IoC 管理

在 spring 配置文件中配置（spring-factory.xml）：

```
<!-- 配置实例工厂方法 -->
<!-- 第一步配置实例工厂 bean -->
<bean id="carFactory"
class="com.gloryh.factory.InstanceCarFactory"></bean>
<!-- 第二步，配置实例工厂创建Car -->
<bean id="car2" factory-bean="carFactory" factory-
method="getCar">
    <constructor-arg value="1"></constructor-arg>
</bean>
```

测试方法：

```
package com.gloryh.test;

import com.gloryh.entity.Car;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

/**
 * 测试四
 *
 * @author 黄光辉
 * @since 2020/8/27
 */
public class Test4 {
  public static void main(String[] args) {
    // 调用IoC配置的工厂类获取bean
    ApplicationContext applicationContext=new
ClassPathXmlApplicationContext("spring-factory.xml");
    Car car =(Car) applicationContext.getBean("car2");
    System.out.println(car);
```

```
    }
}
```

# 四、IoC的自动装载（Autowire）

Person实体类：

```
package com.gloryh.entity;

import lombok.Data;

/**
 * Person实体类
 *
 * @author 黄光辉
 * @since 2020/8/27
 **/
@Data
public class Person {
    private long id;
    private String name;
    private Car car;
}
```

Car实体类：

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * Car实体类
 *
 * @author 黄光辉
 * @since 2020/8/27
 **/
@Data
@AllArgsConstructor
@NoArgsConstructor
```

```
public class Car {
    private long id;
    private String name;
}
```

IoC 负责创建对象，DI负责依赖的注入，这是通过配置 property 标签的 ref 属性来完成的。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="car" class="com.gloryh.entity.Car">
        <property name="id" value="1"></property>
        <property name="name" value="宝马"></property>
    </bean>

    <bean id="person" class="com.gloryh.entity.Person">
        <property name="name" value="张三"></property>
        <property name="id" value="11"></property>
        <property name="car" ref="car"></property>
    </bean>
</beans>
```

我们也可以不使用这种方法而使用另外一种 Spring 提供的更加简便的依赖注入方式，即**自动装载**。通过这种方式可以不需要手动配置 property，IoC 容器会自动选择bean完成注入。

自动装载有两种方式：

- byName：通过属性名来自动装载。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="car" class="com.gloryh.entity.Car" >
        <property name="id" value="1"></property>
        <property name="name" value="宝马"></property>
    </bean>

    <bean id="person" class="com.gloryh.entity.Person"
autowire="byName">
        <property name="name" value="张三"></property>
        <property name="id" value="11"></property>
    </bean>
</beans>
```

当添加了`autowire="byName"`字段，Spring 的自动装载机制就会根据实体类中的字段自动选择bean来注入。bean 的 id 对应 实体类 的 名称，即此时 Person 实体类中 Car 实体类的引用名为car（即:`private Car car;`），xml 文件中 存在 bean 的 id 与之相同（即：`<bean id="car" class="com.gloryh.entity.Car" >`）,所以Spring识别到之后会自动完成装载。

- byType：通过属性的数据类型来自动装载。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringXmlModelInspection -->
<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-3.2.xsd">
    <bean id="a" class="com.gloryh.entity.Car" >
        <property name="id" value="1"></property>
        <property name="name" value="宝马"></property>
    </bean>

    <bean id="person" class="com.gloryh.entity.Person"
autowire="byType">
```

```
        <property name="name" value="张三"></property>
        <property name="id" value="11"></property>
    </bean>
</beans>
```

此时，由于我们使用的为`autowire="byType"`，那么就不再根据 bean 的 id 和 实体类属性名来判断进行自动装载，而是根据其被引用实体类的类型进行判断和自动装载，即此时 Person 实体类中的 成员变量 car 的类型是 Car（即：`private Car car;`），而在 Spring 配置文件中存在一个 bean 的 class 与之对应（即：`<bean id="a" class="com.gloryh.entity.Car" >`），那么也可以完成自动装载。

但是当 Spring 配置文件中有多个 bean 的类型（class）都与实体类成员变量对应时，IoC 自动装载会因为不知道装在哪个而抛出异常。

所以：

- 使用 byName 的话，Spring 配置文件里面可以有多个相同类型的 bean
- 使用 byType 的话，Spring 配置文件里面只能有一个相同类型的 bean

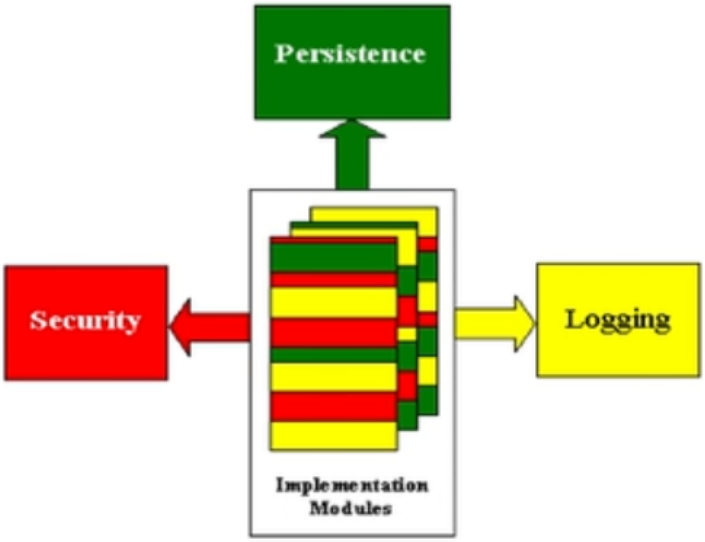
# 五、Spring AOP (面向切面编程)

## 1、什么是面向切面编程

面向切面编程（Aspect Oriented Programming ，简称AOP）使对面向对象编程的一个补充，在系统运行的时候，动态的将代码切入到类的指定方法、指定位置上的编程思想。

AOP的优点：

- 降低模块之间的耦合度
- 使系统更容易拓展
- 更好的代码复用
- 非业务代码更加集中，便于统一管理
- 业务代码更简洁纯粹，没有其他代码的影响
- 将复杂的需求分解出不同方面，将散布在系统中的公共功能集中解决




## 2、AOP的使用方法（以计算器为例）

第一步，在pom.xml中添加相关依赖。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
```

第二步，创建一个计算器接口Cal，定义四个方法（加减乘除）。

```
package com.gloryh.aop.utils;

/**
 * 计算器接口，实现加减乘除
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
public interface Cal {
    public int add (int num1,int num2);
```

```
    public int sub (int num1,int num2);
    public int mul (int num1,int num2);
    public int div (int num1,int num2);

}
```

第三步，创建接口实现类CalImpl，实现这个接口并打印参数日志和结果日志。

- 不采用面向切面思想实现

```
package com.gloryh.aop.impl;

import com.gloryh.aop.utils.Cal;

/**
 * Cal接口的实现类，实现加减乘除算法
 *
 * @author 黄光辉
 * @since 2020/8/28
 */
public class CalImpl implements Cal {
  public int add(int num1, int num2) {
    // 打印日志
    System.out.println("add方法的参数是
["+num1+","+num2+"]");
    System.out.println("add方法的结果是:"+(num1+num2));
    return num1 + num2;
  }

  public int sub(int num1, int num2) {
    // 打印日志
    System.out.println("sub方法的参数是
["+num1+","+num2+"]");
    System.out.println("sub方法的结果是:"+(num1-num2));
    return num1 - num2;
  }

  public int mul(int num1, int num2) {
    // 打印日志
    System.out.println("mul方法的参数是
["+num1+","+num2+"]");
```

```
    System.out.println("mul方法的结果是:"+(num1*num2));
    return num1 * num2;
  }

  public int div(int num1, int num2) {
    // 打印日志
    System.out.println("div方法的参数是
["+num1+","+num2+"]");
    System.out.println("div方法的结果是:"+(num1/num2));
    return num1 / num2;
  }
}
```

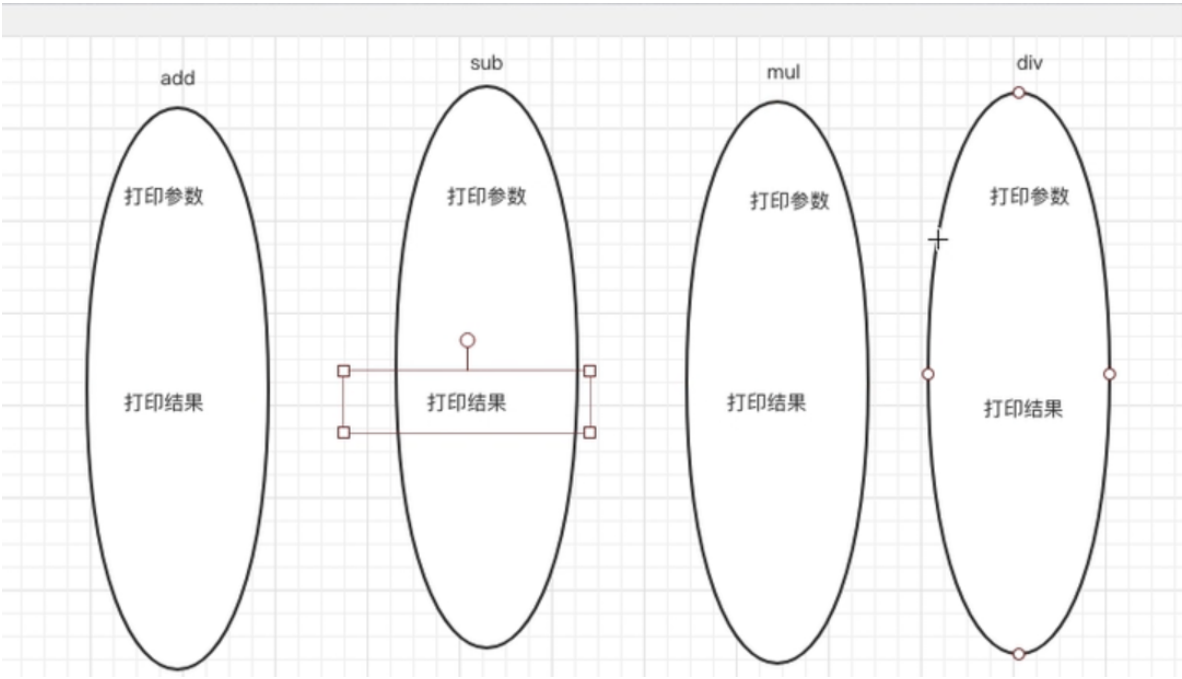
测试方法:

```
package com.gloryh.aop.test;

import com.gloryh.aop.impl.CalImpl;
import com.gloryh.aop.utils.Cal;

/**
 * 计算器测试方法
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
public class Test {
  public static void main(String[] args) {
      Cal cal = new CalImpl();
      cal.add(1,1);
      cal.sub(2,1);
      cal.mul(2,5);
      cal.div(8,4);
  }
}
```

打印日志:

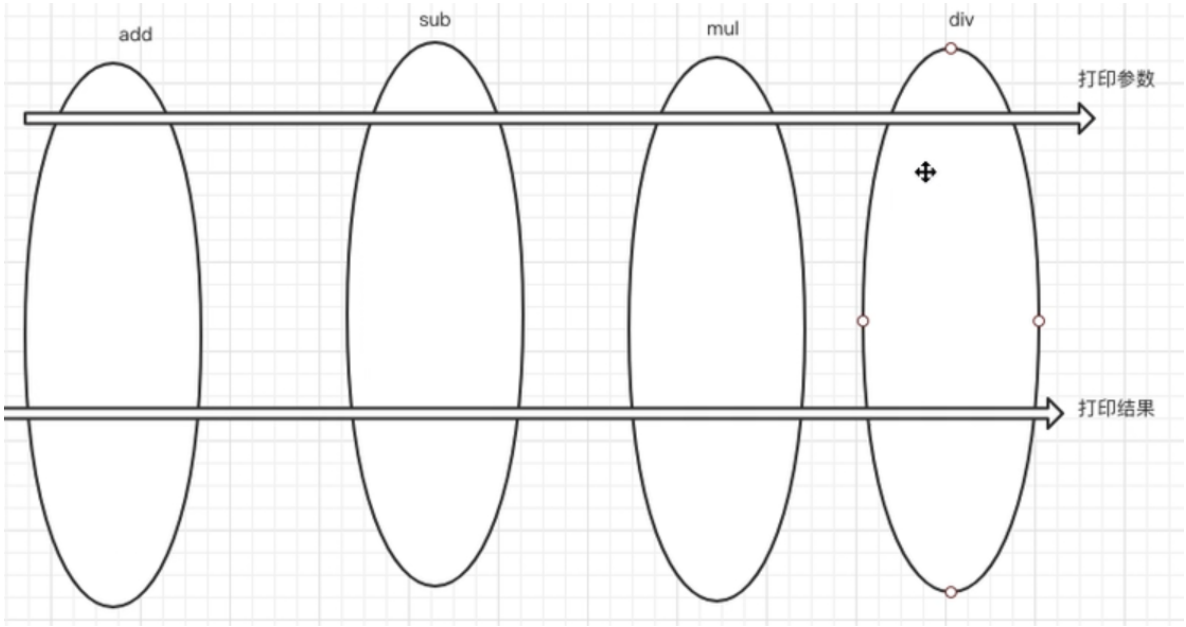
```
add方法的参数是[1,1]
add方法的结果是:2
sub方法的参数是[2,1]
sub方法的结果是:1
mul方法的参数是[2,5]
mul方法的结果是:10
div方法的参数是[8,4]
div方法的结果是:2
```

此时，我们的业务代码中掺杂着打印日志的代码，同时，日志打印的方式也一样，用法也重复，代码编写也重复，费时费力。




而且还不好维护代码，比如当我们更换打印参数的显示方法时或者更换语句的结构或中英文切换时，我们需要在业务代码中定位日志代码再进行修改。当业务量大时，就很麻烦，而且每个方法都需要修改。

所以我们需要引用面向切面编程思想，将我们重复的非业务代码提取出来，统一使用，这样便于维护，业务代码也更加简洁。




- 通过动态代理实现AOP。

  给业务代码找一个代理，打印日志信息的工作交给代理来做，这样的话业务代码就只需要关注自身的业务即可。

  实现方法：

  1.创建一个工具类MyInvocationHandler，实现InvcationHandler接口（这个接口是 Java 的反射机制提供的，该接口用于实现动态代理的功能），这个工具类就是实现创建动态代理类的类。

```
package com.gloryh.aop.utils;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

/**
 * 实现创建动态代理类功能
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
public class MyInvocationHandler implements
InvocationHandler {
    //接收委托对象
    private Object object =null;
```

```
    //返回代理对象
    public Object bind(Object obj){
        this.object=obj;
        return
Proxy.newProxyInstance(obj.getClass().getClassLoade
r(),obj.getClass().getInterfaces(),this );
    }

    public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
        //解耦合，将日志代码分离出来
        System.out.println(method.getName()+"方法的
参数是"+ Arrays.toString(args));
        //通过反射调用业务代码
        Object result =
method.invoke(this.object,args);
        System.out.println(method.getName()+"方法的
结果是:"+result);
        return result;
    }
}
```

使用代理对象解耦合之后我们的业务代码就不会再存在日志打印的非业务代码，而利用代理打印日志，此时业务代码整洁度提高。

```
package com.gloryh.aop.utils.impl;

import com.gloryh.aop.utils.Cal;

/**
 * Cal接口的实现类，实现加减乘除算法
 *
 * @author 黄光辉
 * @since 2020/8/28
 */
public class CalImpl implements Cal {
  public int add(int num1, int num2) {

    return num1 + num2;
  }
```

```
  public int sub(int num1, int num2) {

    return num1 - num2;
  }

  public int mul(int num1, int num2) {

    return num1 * num2;
  }

  public int div(int num1, int num2) {

    return num1 / num2;
  }
}
```

测试方法：

```
package com.gloryh.aop.test;

import com.gloryh.aop.utils.MyInvocationHandler;
import com.gloryh.aop.utils.impl.CalImpl;
import com.gloryh.aop.utils.Cal;

/**
 * 计算器测试方法
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
public class Test {
  public static void main(String[] args) {
      //首先实例化一个委托对象
      Cal cal = new CalImpl();
      //然后实例化MyInvocationHandler方法
      MyInvocationHandler myInvocationHandler =new
MyInvocationHandler();
      //根据MyInvocationHandler方法得到代理对象
      Cal cal1 = (Cal)
myInvocationHandler.bind(cal);
      //使用代理对象实现方法
```

```
        cal1.add(1,1);
        cal1.sub(2,1);
        cal1.mul(2,5);
        cal1.div(8,4);

    }
}
```

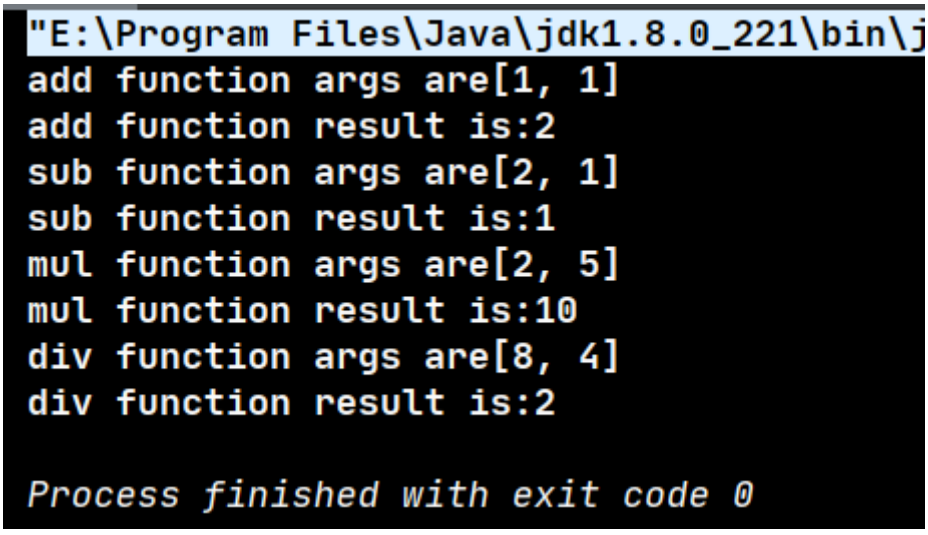
日志打印结果：


如果我们想把日志信息进行修改，就只需要修改一次就行了，比如修改为英文：

```
package com.gloryh.aop.utils;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

/**
 * 实现创建动态代理类功能
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
public class MyInvocationHandler implements
InvocationHandler {
    //接收委托对象
    private Object object =null;
```

```
    //返回代理对象
    public Object bind(Object obj){
        this.object=obj;
        return
Proxy.newProxyInstance(obj.getClass().getClassLoade
r(),obj.getClass().getInterfaces(),this );
    }

    public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
        //解耦合，将日志代码分离出来
        System.out.println(method.getName()+"
function args are"+ Arrays.toString(args));
        //通过反射调用业务代码
        Object result =
method.invoke(this.object,args);
        System.out.println(method.getName()+"
function result is:"+result);
        return result;
    }
}
```

日志打印：




但是我们通过这种代理方法实现过于复杂，不太符合我们面向对象的思想。而 Spring 框架对AOP进行了封装，我们使用 Spring 框架可以用面向对象的思想来实现 AOP 。

- 通过 Spring 框架 + AOP 实现

Spring框架中不需要创建InvocationHandler，只需要创建一个切面对象，将所有的非业务代码在切面对象中完成。

但是，Spring框架底层还是使用动态代理的方法，不过它会自动根据切面类以及目标了生成一个代理对象。

第一步，创建切面类（LoggerAspect.class）

为它添加Component注解，将其交给 IoC 容器管理。

为它添加Aspect注解，使其从普通对象转化为切面对象。

```
package com.gloryh.aop.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Arrays;

/**
 * 日志打印切面类
 *
 * @author 黄光辉
 * @since 2020/8/28
 **/
@Component
@Aspect
public class LoggerAspect {

    @Before("execution(public int
com.gloryh.aop.utils.impl.CalImpl.*(..))")
    public void before(JoinPoint joinPoint){
        //首先获取方法名
        String name
=joinPoint.getSignature().getName();
        //获取参数
        String args
=Arrays.toString(joinPoint.getArgs());
        System.out.println(name+"方法的参数是: "+
args);
    }
```

```
    @After("execution(public int
com.gloryh.aop.utils.impl.CalImpl.*(..))")
    public void after(JoinPoint joinPoint){
        //首先获取方法名
        String name
=joinPoint.getSignature().getName();
        System.out.println(name+"方法执行完毕");
    }
    @AfterReturning(value = "execution(public int
com.gloryh.aop.utils.impl.CalImpl.*(..))",returning
= "result")
    public void afterReturning(JoinPoint
joinPoint,Object result){
        //首先获取方法名
        String name
=joinPoint.getSignature().getName();

        System.out.println(name+"方法的结果是: "+
result);
    }
    @AfterThrowing(value = "execution(public int
com.gloryh.aop.utils.impl.CalImpl.*(..))",throwing
= "exception")
    public void afterThrowing(JoinPoint joinPoint
,Exception exception){
        String name
=joinPoint.getSignature().getName();

        System.out.println(name+"方法的执行出现异常:\n
"+exception );
    }
}
```

第二步，为实现类添加Component注解，将实现类也交给IoC容器管理

```
package com.gloryh.aop.utils.impl;

import com.gloryh.aop.utils.Cal;
import org.springframework.stereotype.Component;
```

```java
/**
 * Cal接口的实现类，实现加减乘除算法
 *
 * @author 黄光辉
 * @since 2020/8/28
 */
@Component
public class CalImpl implements Cal {
  public int add(int num1, int num2) {

    return num1 + num2;
  }

  public int sub(int num1, int num2) {

    return num1 - num2;
  }

  public int mul(int num1, int num2) {

    return num1 * num2;
  }

  public int div(int num1, int num2) {

    return num1 / num2;
  }
}
```

第三步，在Spring配置文件中添加Component注解扫描配置和Aspect注解生效配置

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">


    <!-- 自动扫描component注解 -->
    <context:component-scan base-package="com.gloryh"></context:component-scan>

    <!-- 使 Aspect 注解生效，为目标类自动生成代理对象 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
</beans>
```

`<context:component-scan base-package="com.gloryh"></context:component-scan>`:这句话的意思是将包 com.gloryh 中的所有类进行扫描，把添加了Component注解的类扫描到 IoC 容器中，即让IoC来管理这些对象。

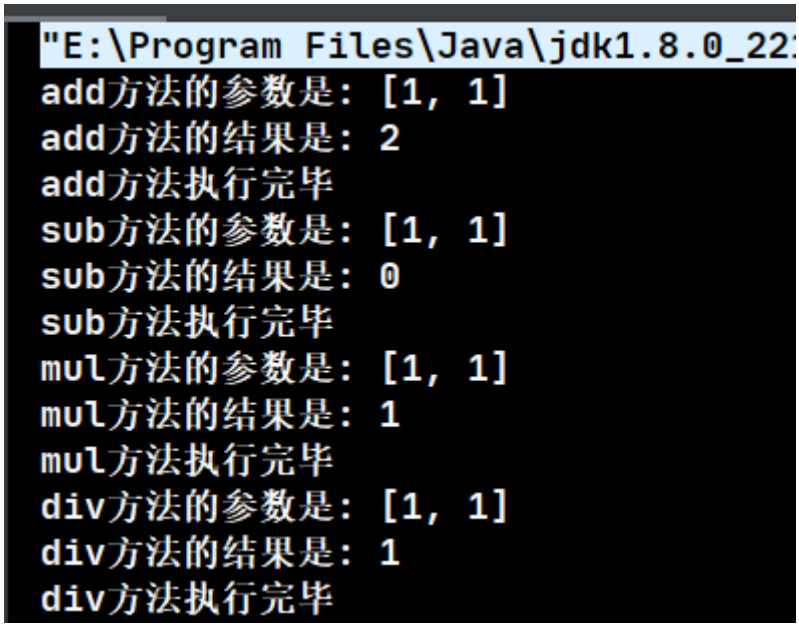
`<aop:aspectj-autoproxy></aop:aspectj-autoproxy></beans>`:这句话的意思是让Spring 框架结合切面类和目标类自动生成动态代理对象。（相当于创建了之前我们编写的 ---MyInvocationHandler ---这个类)

测试方法：

```
package com.gloryh.aop.test;
```

```
import com.gloryh.aop.utils.Cal;
import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;

/**
 * 测试Spring 动态代理实现面向切面编程思想 ------AOP
 *
 * @author 黄光辉
 * @since 2020/8/28
 */
public class Test2 {
  public static void main(String[] args) {
    // 加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring-aop.xml");
    // 获取代理对象
    Cal proxy = (Cal)
applicationContext.getBean("calImpl");
    proxy.add(1, 1);
    proxy.sub(1, 1);
    proxy.mul(1, 1);
    proxy.div(1, 1);
  }
}
```

执行结果:




**注解解释:**

> `@Before("execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..)")`

切面方法执行时机，即在业务代码执行前，这里使用里省略的写法，全写应该是:

`@Before(value="execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..)")`

- execution(),填入要执行的方法，注：方法用全量名
- 星号 * ： 表示所有方法，因为我们定义的这个实现类里的加减乘除都有相同的日志打印操作
- 两个点 .. ： 代表参数, 这句注解的意思就是说我们要执行的是:`com.gloryh.aop.utils.impl.CalImpl` 这个实现类里的所有的 返回值是 int 被public修饰的 带参数方法, 同时在这个方法执行之前执行`before(JoinPoint joinPoint)`方法打印日志。

> `@After("execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..))")`

切面方法执行时机，即在业务代码执行完毕后,这里使用里省略的写法，全写应该是:

`@After(value="execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..)")`

- execution(),填入要执行的方法，注：方法用全量名

- 星号 * ： 表示所有方法，因为我们定义的这个实现类里的加减乘除都有相同的日志打印操作
- 两个点 .. ： 代表参数, 这句注解的意思就是说我们要执行的是:`com.gloryh.aop.utils.impl.CalImpl` 这个实现类里的所有的 返回值是 int 被public修饰的 带参数方法, 同时在这个方法执行之后执行`after(JoinPoint joinPoint)`方法打印日志。

> `@AfterReturning(value = "execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..))",returning = "result")`

切面方法执行的时机，即在方法执行到数据返回后

- returning 来获取return的值。
- execution(),填入要执行的方法，注：方法用全量名
- 星号 * ： 表示所有方法，因为我们定义的这个实现类里的加减乘除都有相同的日志打印操作
- 两个点 .. ： 代表参数, 这句注解的意思就是说我们要执行的是:`com.gloryh.aop.utils.impl.CalImpl`这个实现类里的所有的 返回值是 int 被public修饰的 带参数方法, 同时在这个方法执行到数据返回之后执行`afterReturning(JoinPoint joinPoint,Object result)`方法打印日志。

> `@AfterThrowing(value = "execution(public int com.gloryh.aop.utils.impl.CalImpl.*(..))",throwing = "exception")`

切面方法执行的时机，即在方法出现异常时弹出异常打印。

- throwing 来捕获异常。
- execution(),填入要执行的方法，注：方法用全量名
- 星号 * ： 表示所有方法，因为我们定义的这个实现类里的加减乘除都有相同的日志打印操作
- 两个点 .. ： 代表参数, 这句注解的意思就是说我们要执行的是:`com.gloryh.aop.utils.impl.CalImpl`这个实现类里的所有的 返回值是 int 被public修饰的 带参数方法, 同时在这个方法执行出现异常之后执行

`afterThrowing(JoinPoint joinPoint,Exception exception)`方法打印日志。

**几个知识点：**

- 切面：横切关注点被模块化的抽象对象。（打印日志的位置）
- 通知：切面对象完成的工作。（打印日志）
- 目标：被通知/被横切的对象。（CalImpl实现方法类）
- 代理：切面、通知、目标混合之后的对象。（被spring 配置文件连接起来的CalImpl方法实现类，Cal接口类，LoggerAspect方法类的组合）
- 连接点：通知要插入业务代码的具体位置。（JoinPointer）
- 切点：AOP 通过切点来定位连接点。