Spring 框架两大核心机制（IoC、AOP）

- IoC（控制反转）/ DI（依赖注入）
- AOP（面向切面编程）

Spring 是一个企业级开发框架，是软件设计层面的框架，优势在于可以将应用程序进行分层，开发者可以自主选择组件。

MVC：Struts2、Spring MVC

ORMapping：Hibernate、MyBatis、Spring Data

如何使用 IoC

- 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.southwind</groupId>
    <artifactId>aispringioc</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.11.RELEASE</version>
        </dependency>
    </dependencies>

</project>
```

- 创建实体类 Student

```java
package com.southwind.entity;

import lombok.Data;

@Data
public class Student {
    private long id;
    private String name;
    private int age;
}
```

- 传统的开发方式，手动 new Student

```java
Student student = new Student();
student.setId(1L);
student.setName("张三");
student.setAge(22);
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 格式的配置文件，文件名可以自定义。

```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"
       xmlns:context="http://www.springframework.org/schema/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/spring-context-4.3.xsd
">

    <bean id="student" class="com.southwind.entity.Student">
        <property name="id" value="1"></property>
        <property name="name" value="张三"></property>
        <property name="age" value="22"></property>
    </bean>

</beans>
```

- 从 IoC 中获取对象，通过 id 获取。

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

配置文件

- 通过配置 `bean` 标签来完成对象的管理。
  - `id`：对象名。
  - `class`：对象的模版类（所有交给 IoC 容器来管理的类必须有无参构造函数，因为 Spring 底层是通过反射机制来创建对象，调用的是无参构造）
- 对象的成员变量通过 `property` 标签完成赋值。
  - `name`：成员变量名。
  - `value`：成员变量值（基本数据类型，String 可以直接赋值，如果是其他引用类型，不能通过 value 赋值）
  - `ref`：将 IoC 中的另外一个 bean 赋给当前的成员变量（DI）

```
<bean id="student" class="com.southwind.entity.Student">
    <property name="id" value="1"></property>
    <property name="name" value="张三"></property>
    <property name="age" value="22"></property>
    <property name="address" ref="address"></property>
</bean>

<bean id="address" class="com.southwind.entity.Address">
    <property name="id" value="1"></property>
    <property name="name" value="科技路"></property>
</bean>
```

IoC 底层原理

- 读取配置文件，解析 XML。
- 通过反射机制实例化配置文件中所配置所有的 bean。

```
package com.southwind.ioc;

import org.dom4j.Document;
import org.dom4j.DocumentException;
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;

public class ClassPathXmlApplicationContext implements ApplicationContext {
    private Map<String,Object> ioc = new HashMap<String, Object>();
    public ClassPathXmlApplicationContext(String path){
        try {
            SAXReader reader = new SAXReader();
            Document document = reader.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");
                //通过反射机制创建对象
                Class clazz = Class.forName(className);
                //获取无参构造函数，创建目标对象
                Constructor constructor = clazz.getConstructor();
                Object object = constructor.newInstance();
                //给目标对象赋值
                Iterator<Element> beanIter = element.elementIterator();
                while(beanIter.hasNext()){
                    Element property = beanIter.next();
                    String name = property.attributeValue("name");
                    String valueStr = property.attributeValue("value");
                    String ref = property.attributeValue("ref");
                    if(ref == null){
                        String methodName =
"set"+name.substring(0,1).toUpperCase()+name.substring(1);
                        Field field = clazz.getDeclaredField(name);
                        Method method =
clazz.getDeclaredMethod(methodName,field.getType());
                        //根据成员变量的数据类型将 value 进行转换
                        Object value = null;
                        if(field.getType().getName() == "long"){
                            value = Long.parseLong(valueStr);
                        }
                        if(field.getType().getName() == "java.lang.String"){
                            value = valueStr;
                        }
                        if(field.getType().getName() == "int"){
                            value = Integer.parseInt(valueStr);
                        }
                        method.invoke(object,value);
```

```
                    }
                    ioc.put(id,object);
                }
            }
        } catch (DocumentException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e){
            e.printStackTrace();
        } catch (NoSuchMethodException e){
            e.printStackTrace();
        } catch (InstantiationException e){
            e.printStackTrace();
        } catch (IllegalAccessException e){
            e.printStackTrace();
        } catch (InvocationTargetException e){
            e.printStackTrace();
        } catch (NoSuchFieldException e){
            e.printStackTrace();
        }
    }

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

通过运行时类获取 bean

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
Student student = (Student) applicationContext.getBean(Student.class);
System.out.println(student);
```

这种方式存在一个问题，配置文件中一个数据类型的对象只能有一个实例，否则会抛出异常，因为没有唯一的 bean。

通过有参构造创建 bean

- 在实体类中创建对应的有参构造函数。
- 配置文件

```
<bean id="student3" class="com.southwind.entity.Student">
    <constructor-arg name="id" value="3"></constructor-arg>
    <constructor-arg name="name" value="小明"></constructor-arg>
    <constructor-arg name="age" value="18"></constructor-arg>
    <constructor-arg name="address" ref="address"></constructor-arg>
</bean>
```

```
<bean id="student3" class="com.southwind.entity.Student">
    <constructor-arg index="0" value="3"></constructor-arg>
    <constructor-arg index="2" value="18"></constructor-arg>
    <constructor-arg index="1" value="小明"></constructor-arg>
    <constructor-arg index="3" ref="address"></constructor-arg>
</bean>
```

给 bean 注入集合

```
<bean id="student" class="com.southwind.entity.Student">
    <property name="id" value="2"></property>
    <property name="name" value="李四"></property>
    <property name="age" value="33"></property>
    <property name="addresses">
        <list>
            <ref bean="address"></ref>
            <ref bean="address2"></ref>
        </list>
    </property>
</bean>

<bean id="address" class="com.southwind.entity.Address">
    <property name="id" value="1"></property>
    <property name="name" value="科技路"></property>
</bean>

<bean id="address2" class="com.southwind.entity.Address">
    <property name="id" value="2"></property>
    <property name="name" value="高新区"></property>
</bean>
```

scope 作用域

Spring 管理的 bean 是根据 scope 来生成的，表示 bean 的作用域，共4种，默认值是 singleton。

- 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：回话，表示在一个用户会话内有效。

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

prototype 模式当业务代码获取 IoC 容器中的 bean 时，Spring 才去调用无参构造创建对应的 bean。

singleton 模式无论业务代码是否获取 IoC 容器中的 bean，Spring 在加载 spring.xml 时就会创建 bean。

Spring 的继承

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

```
<bean id="student2" class="com.southwind.entity.Student">
    <property name="id" value="1"></property>
    <property name="name" value="张三"></property>
    <property name="age" value="22"></property>
    <property name="addresses">
        <list>
            <ref bean="address"></ref>
            <ref bean="address2"></ref>
        </list>
    </property>
</bean>

<bean id="address" class="com.southwind.entity.Address">
    <property name="id" value="1"></property>
    <property name="name" value="科技路"></property>
</bean>

<bean id="address2" class="com.southwind.entity.Address">
    <property name="id" value="2"></property>
    <property name="name" value="高新区"></property>
</bean>

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

Spring 的继承关注点在于具体的对象，而不在于类，即不同的两个类的实例化对象可以完成继承，前提是子对象必须包含父对象的所有属性，同时可以在此基础上添加其他的属性。

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，再创建依赖的 bean，A 依赖于 B，先创建 B，再创建 A。

```
<?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-3.2.xsd
                           ">

    <bean id="student" class="com.southwind.entity.Student" depends-
on="user"></bean>

    <bean id="user" class="com.southwind.entity.User"></bean>

</beans>
```

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 bean 的配置以及 bean 之间的依赖注入。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       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.southwind.entity.Student" p:id="1"
p:name="张三" p:age="22" p:address-ref="address"></bean>

    <bean id="address" class="com.southwind.entity.Address" p:id="2"
p:name="科技路"></bean>

</beans>
```

Spring 的工厂方法

IoC 通过工厂模式创建 bean 的方式有两种：

- 静态工厂方法
- 实例工厂方法

> 静态工厂方法

```
package com.southwind.entity;

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

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

```
package com.southwind.factory;

import com.southwind.entity.Car;

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

public class StaticCarFactory {
    private static Map<Long, Car> carMap;
    static{
        carMap = new HashMap<Long, Car>();
        carMap.put(1L,new Car(1L,"宝马"));
        carMap.put(2L,new Car(2L,"奔驰"));
    }

    public static Car getCar(long id){
        return carMap.get(id);
    }
}
```

```
<!-- 配置静态工厂创建 Car -->
<bean id="car" class="com.southwind.factory.StaticCarFactory" factory-
method="getCar">
    <constructor-arg value="2"></constructor-arg>
</bean>
```

> 实例工厂方法

```java
package com.southwind.factory;

import com.southwind.entity.Car;

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

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

    public Car getCar(long id){
        return carMap.get(id);
    }
}
```

```xml
<!-- 配置实例工厂 bean -->
<bean id="carFactory" class="com.southwind.factory.InstanceCarFactory">
</bean>

<!-- 赔偿实例工厂创建 Car -->
<bean id="car2" factory-bean="carFactory" factory-method="getCar">
    <constructor-arg value="1"></constructor-arg>
</bean>
```

IoC 自动装载（Autowire）

IoC 负责创建对象，DI 负责完成对象的依赖注入，通过配置 property 标签的 ref 属性来完成，同时 Spring 提供了另外一种更加简便的依赖注入方式：自动装载，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：通过属性名自动装载
- byType：通过属性的数据类型自动装载

> byName

```
<bean id="cars" class="com.southwind.entity.Car">
    <property name="id" value="1"></property>
    <property name="name" value="宝马"></property>
</bean>

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

> byType

```
<bean id="car" class="com.southwind.entity.Car">
    <property name="id" value="2"></property>
    <property name="name" value="奔驰"></property>
</bean>

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

byType 需要注意，如果同时存在两个及以上的符合条件的 bean 时，自动装载会抛出异常。

AOP

AOP：Aspect Oriented Programming 面向切面编程。

AOP 的优点：

- 降低模块之间的耦合度。
- 使系统更容易扩展。
- 更好的代码复用。
- 非业务代码更加集中，不分散，便于统一管理。
- 业务代码更加简洁存粹，不参杂其他代码的影响。

AOP 是对面向对象编程的一个补充，在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面编程。将不同方法的同一个位置抽象成一个切面对象，对该切面对象进行编程就是 AOP。

如何使用?

- 创建 Maven 工程，pom.xml 添加

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
```

```
        <artifactId>spring-context</artifactId>
        <version>5.0.11.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>5.0.11.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.0.11.RELEASE</version>
    </dependency>
</dependencies>
```

- 创建一个计算器接口 Cal，定义4个方法。

```
package com.southwind.utils;

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);
}
```

- 创建接口的实现类 CalImpl。

```
package com.southwind.utils.impl;

import com.southwind.utils.Cal;

public class CalImpl implements Cal {
    public int add(int num1, int num2) {
        System.out.println("add方法的参数是["+num1+","+num2+"]");
        int result = num1+num2;
        System.out.println("add方法的结果是"+result);
        return result;
    }

    public int sub(int num1, int num2) {
        System.out.println("sub方法的参数是["+num1+","+num2+"]");
        int result = num1-num2;
        System.out.println("sub方法的结果是"+result);
        return result;
    }
```

```
    public int mul(int num1, int num2) {
        System.out.println("mul方法的参数是["+num1+","+num2+"]");
        int result = num1*num2;
        System.out.println("mul方法的结果是"+result);
        return result;
    }

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

上述代码中，日志信息和业务逻辑的耦合性很高，不利于系统的维护，使用 AOP 可以进行优化，如何来实现 AOP？使用动态代理的方式来实现。

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

```
package com.southwind.utils;

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

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 object){
        this.object = object;
        return
Proxy.newProxyInstance(object.getClass().getClassLoader(),object.getClass().
getInterfaces(),this);
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println(method.getName()+"方法的参数是： "+
Arrays.toString(args));
        Object result = method.invoke(this.object,args);
        System.out.println(method.getName()+"的结果是"+result);
```

```
        return result;
    }
}
```

以上是通过动态代理实现 AOP 的过程，比较复杂，不好理解，Spring 框架对 AOP 进行了封装，使用 Spring 框架可以用面向对象的思想来实现 AOP。

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即可，Spring 框架底层会自动根据切面类以及目标类生成一个代理对象。

LoggerAspect

```
package com.southwind.aop;

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

import java.util.Arrays;

@Aspect
@Component
public class LoggerAspect {

    @Before(value = "execution(public int com.southwind.utils.impl.CalImpl.*
(..))")
    public void before(JoinPoint joinPoint){
        //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(value = "execution(public int com.southwind.utils.impl.CalImpl.*
(..))")
    public void after(JoinPoint joinPoint){
        //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.southwind.utils.impl.CalImpl.*(..))",returning = "result")
    public void afterReturning(JoinPoint joinPoint,Object result){
        //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.southwind.utils.impl.CalImpl.*(..))",throwing = "exception")
    public void afterThrowing(JoinPoint joinPoint,Exception exception){
        //获取方法名
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法抛出异常："+exception);
    }

}
```

LoggerAspect 类定义处添加的两个注解：

- `@Aspect`：表示该类是切面类。
- `@Component`：将该类的对象注入到 IoC 容器。

具体方法处添加的注解：

`@Before`：表示方法执行的具体位置和时机。

CalImpl 也需要添加 `@Component`，交给 IoC 容器来管理。

```
package com.southwind.utils.impl;

import com.southwind.utils.Cal;
import org.springframework.stereotype.Component;

@Component
public class CalImpl implements Cal {
    public int add(int num1, int num2) {
        int result = num1+num2;
        return result;
    }

    public int sub(int num1, int num2) {
        int result = num1-num2;
        return result;
    }

    public int mul(int num1, int num2) {
        int result = num1*num2;
        return result;
    }

    public int div(int num1, int num2) {
        int result = num1/num2;
        return result;
    }
}
```

spring.xml 中配置 AOP。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
">

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

    <!-- 是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` 将 `com.southwind` 包中的所有类进行扫描，如果该类同时添加了 `@Component`，则将该类扫描到 IoC 容器中，即 IoC 管理它的对象。

`aop:aspectj-autoproxy` 让 Spring 框架结合切面类和目标类自动生成动态代理对象。

- 切面：横切关注点被模块化的抽象对象。
- 通知：切面对象完成的工作。
- 目标：被通知的对象，即被横切的对象。
- 代理：切面、通知、目标混合之后的对象。
- 连接点：通知要插入业务代码的具体位置。
- 切点：AOP 通过切点定位到连接点。