1、Spring

1.1 简介

- Spring：春天，给软件行业带来了春天！
- 2002，首次推出了Spring框架的雏形：interface21框架
- Spring框架以interface21框架为基础，经过重新设计，并不断丰富其内涵，于2004年3月24日，发布了1.0正式版本。
- Rod Johnson，Srping FrameWork创始人，著名作者。很难想象Rod Johnson的学历？他是悉尼大学的博士，然而他的专业不是计算机，而是音乐学
- Spring理念：使现有的技术更加容易使用，本身是一个大杂烩，整合了现有的技术框架！
- SSH：Struct2+Spring+Hibernate
- SSM：SpringMVC+Spring+Mybatis

官网：https://spring.io/projects/spring-framework#overview

官网下载地址：https://repo.spring.io/release/org/springframework/spring/

GitHub:https://github.com/spring-projects/spring-framework

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-
webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-
webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

1.2 优点

- spring是一个免费的开源的框架（容器）
- spring是一个轻量级的、非入侵式的框架
- 控制反转（IOC），面向切面编程（AOP）
- 支持事务的处理，对框架整合的支持

总结一句话：Spring就是一个轻量级的控制反转（IOC）和面向切面编程（AOP）的框架！

1.3 组成

1.4 拓展

在Spring的官网中有这个介绍：现代化的java开发，说白就是基于spring的开发

- Sprint Boot
  - 一个快速开发的脚手架
  - 基于SpringBoot可以快速的开发单个微服务
  - 约定大于配置
- Spring Cloud

- SpringCloud是基于SpringBoot实现的

因为现在大多数公司都在使用SpringBoot进行快速开发，学习SpringBoot的前提，需要完全掌握Spring及SpringMVC！承上启下的作用

弊端：发展了太久之后，违背了原来的理念！配置十分繁琐，人称“配置地狱”！

2、IOC理论指导

1、UserDao接口

2、UserDaoImpl实现类

3、UserService业务接口

4、UserServiceImpl业务实现类

在我们之前的业务中，用户的需求可能会影响我们原来的代码，我们需要根据用户的需求去修改原代码，如果程序代码量十分大，修改一次的成本代价十分昂贵！

```
    private UserDao userDao = new UserDaoSqlserverImpl();
```

我们使用一个Set接口实现。已经发生了革命性的变化！！！

```
    private UserDao userDao;

    //利用set进行动态实现值的注入;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
```

```
        //用户实际调用的是业务层，dao层不需要直接接触！
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDao(new UserDaoSqlserverImpl());

        userService.getUser();
```

- 之前，程序是主动创建对象！控制权在程序员手上
- 使用set注入后，程序不在具有主动性，而是变成了被动接受对象！

这种思想，从本质上解决了问题，我们程序员不用再去管理对象的创建了。系统的耦合性大大降低，可以更加专注的在业务的实现上！这是IOC的原型！

2.1 IOC本质

控制反转IoC(Inversion of Control)，是一种设计思想，DI(依赖注入)是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：获得依赖对象的方式反转了。

采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入（Dependency Injection,DI）

3、HelloSpring

思考问题?

- Hello对象是谁创建的？ Spring创建
- Hello对象的属性是怎么设置的？ 属性是由Spring容器设置的

```
<!--    使用Spring来创建对象，在Spring这些都称为Bean

    类型 变量名= new 类型();
    Hello hello = new Hello();
    id = 变量名
    class = 类型
    property 相当于给对象中的属性设置一个值!
    -->

    <bean id="hello" class="com.example.pojo.Hello">
        <property name="str" value="Spring"/>
    </bean>
```

这个过程就叫控制反转：

控制：谁来控制对象的创建，传统应用程序的对象是由程序本身控制创建的，使用Spring后，对象是由Spring来创建的

反转：程序本身不创建对象，而变成被动的接收对象

依赖注入：就是利用set方法进行注入的

IOC是一种编程思想，由主动的编程变成被动的接收

可以通过newClassPathXmlApplicationContext去浏览一下底层源码。

Ok！到了现在，我们彻底不用再去程序中改动去了，要实现不同的操作，只需要在xml配置文件中进行修改，所谓的IoC，一句话搞定：对象由Spring来创建，管理，装配

```
        <bean id="mysqlImpl" class="com.example.dao.UserDaoMysqlImpl"/>
        <bean id="oracleImpl" class="com.example.dao.UserDaoOracleImpl"/>
        <bean id="SqlserverImpl"
    class="com.example.dao.UserDaoSqlserverImpl"/>

        <bean id="UserServiceImpl"
    class="com.example.service.UserServiceImpl">
    <!--        ref: 引用Spring容器中创建好的对象-->
    <!--        val: 具体的值，如：基本类型-->
            <property name="userDao" ref="SqlserverImpl"/>
        </bean>
```

4、IOC创建对象的方式

1、使用无参构造创建对象，默认！

2、假设我们使用有参构造创建对象

- 下标赋值

```
        <bean id="user" class="com.example.pojo.User">
            <constructor-arg index="0" value="小白和小鱼"/>
        </bean>
```

- 类型(不建议使用)

```
    <bean id="user" class="com.example.pojo.User">
        <constructor-arg type="java.lang.String" value="小白"/>
    </bean>
```

- 参数名

```
    <bean id="user" class="com.example.pojo.User">
        <constructor-arg name="name" value="小鱼"/>
    </bean>
```

总结：在配置文件加载的时候，容器中管理的对象就已经初始化了

5、Spring配置

5.1 别名

```
<!--    别名，如果添加了别名，我们也可以使用别名获取到这个对象-->
    <alias name="user" alias="userNew"/>
```

5.2 Bean的配置

```
<!--
    id：bean的唯一标示符，也就是相当于我们学的对象名
    class：bean对象所对应的全限定名：包名+类型
    name：也是别名,而且name 可以同时取多个别名,允许使用多种分隔符进行分割
-->
    <bean id="userT" class="com.example.pojo.UserT" name="user2,u2
u3;u4">
        <property name="name" value="哈哈哈"/>
    </bean>
```

5.3 import

这个import，一般用于团队开发使用，他可以将多个配置文件，导入合并为一个

假设，现在项目中有多个人开发，这三个人复制不同的类的开发，不同的类需要注册在不同的bean中，我们可以利用import将其合并

```
<import resource="beans.xml"/>
<import resource="beans2.xml"/>
<import resource="beans3.xml"/>
```

总的是：applicationContext.xml

6、依赖注入

6.1、构造器注入

前面已经讲过

6.2、Set方式注入【重点】

- 依赖注入：Set注入！
  - 依赖：bean对象的创建依赖于容器！
  - 注入：bean对象中的所有属性，由容器来注入！

【环境搭建】

1. 普通类型

```
public class Address {
    private String address;
}
```

2. 复杂类型

```
public class Student {
    private String name;
    private Address address;
    private String[] books;
    private List<String> hobby;
    private Map<String, String> card;
    private Set<String> games;
    private String wife;
    private Properties info;
}
```

3. beans.xml

```
    <bean id="student" class="com.example.pojo.Student">
        <!--第一种，普通注入-->
        <property name="name" value="小白"/>
    </bean>
```

4. 测试

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        Student student = (Student) context.getBean("student");
        System.out.println(student.getName());
    }
}
```

5、完善注入信息

```
    <bean id="address" class="com.example.pojo.Address">
        <property name="address" value="西安"/>
    </bean>
```

```xml
<bean id="student" class="com.example.pojo.Student">
    <!--value-->
    <property name="name" value="小白"/>

    <!--ref-->
    <property name="address" ref="address"/>

    <!--数组-->
    <property name="books">
        <array>
            <value>红楼梦</value>
            <value>西游记</value>
            <value>水浒传</value>
            <value>三国演义</value>
        </array>
    </property>

    <!--list-->
    <property name="hobby">
        <list>
            <value>听歌</value>
            <value>敲代码</value>
            <value>看电影</value>
        </list>
    </property>

    <!--Map-->
    <property name="card">
        <map>
            <entry key="身份证" value="123121212121212"/>
            <entry key="银行卡" value="129283729198938 39"/>
        </map>
    </property>

    <!--Set-->
```

```
        <property name="games">
            <set>
                <value>LOL</value>
                <value>COC</value>
                <value>BOB</value>
            </set>
        </property>

        <!--null-->
        <property name="wife">
            <null/>
        </property>

        <!--Properties-->
        <property name="info">
            <props>
                <prop key="driver">20190526</prop>
                <prop key="url">男</prop>
                <prop key="username">root</prop>
                <prop key="password">12345666</prop>
            </props>
        </property>
    </bean>
```

6.3、拓展注入

https://docs.spring.io/spring-framework/docs/4.3.30.RELEASE/spring-framework-reference/htmlsingle/#beans

使用：

```
<?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:p="http://www.springframework.org/schema/p"
```

```
        xmlns:c="http://www.springframework.org/schema/c"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--p命名空间注入，可以直接注入简单属性：property-->
    <bean id="user" class="com.example.pojo.User" p:name="小白"
p:age="18"/>

    <!--c命名空间注入，通过构造器注入：construct-args-->
    <bean id="user2" class="com.example.pojo.User" c:age="18" c:name="小
白白"/>
</beans>
```

测试：

```
    @Test
    public void test2(){
        ApplicationContext context = new
ClassPathXmlApplicationContext("userbeans.xml");
        User user = context.getBean("user2", User.class);

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

注意点：p命名和c命名不能直接输入，需要导入依赖

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

6.4、 bean的作用域

1、单例模式（Spring默认机制）

```
    <!--c命名空间注入，通过构造器注入：construct-args-->
    <bean id="user2" class="com.example.pojo.User" c:age="18" c:name="小
白白" scope="singleton"/>
```

2、原型模式：每次从容器中get的时候，都会产生一个新对象

```
    <!--c命名空间注入，通过构造器注入：construct-args-->
    <bean id="user2" class="com.example.pojo.User" c:age="18" c:name="小
白白" scope="prototype"/>
```

3、其余的request、session、application这些个只能在web开发中使用到！

7、Bean的自动装配

- 自动装配是Spring满足bean依赖一种方式！
- Spring会在上下文中自动寻找，并自动给bean装配属性！

在Spring中有三种装配的方式

1、在xml中显示的配置

2、在java中显示配置

3、隐式的自动装配bean【重要】

7.1 测试

测试：一个人两个动物

7.2 ByName自动装配

```
<!--    byName:会自动在容器的上下文中查找，和自己对象set方法后面的值对应的beanid-
->
<!--    byName:会自动在容器的上下文中查找，和自己对象类型相同的的beanid-->
    <bean id="people" class="com.example.pojo.People" autowire="byName">
        <property name="name" value="小白白啊"/>
    </bean>
```

7.3 ByType自动装配

```
    <bean class="com.example.pojo.Cat"/>
    <bean class="com.example.pojo.Dog"/>

<!--    byName:会自动在容器的上下文中查找，和自己对象set方法后面的值对应的beanid-
->
    <!--    byName:会自动在容器的上下文中查找，和自己对象set方法后面的值对应的
beanid-->
    <bean id="people" class="com.example.pojo.People" autowire="byType">
        <property name="name" value="小白白啊"/>
    </bean>
```

小结：

- byname的时候，需要保证bean的id唯一，并且这个bean需要和自动注入的属性的set方法值一致！
- Bytype的时候，需要保证所有bean的class唯一，并且这个bean需要和自动注入的属性的类型一致！

7.4 使用注解实现自动装配

jdk1.5支持注解，Spring2.5支持注解！

要使用注解须知：

1、导入约束

2、配置注解的支持

```
<?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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-
context.xsd">

    <context:annotation-config/>
</beans>
```

@Autowired

直接在属性上面使用即可，也可以在set方式上使用！

使用Autowired我们可以不用编写set方法，前提是你这个自动装载的属性在IOC（Spring）容器中存在，且符合名字byname！

科普：

@Nullable：字段标记了这个注解，说明该字段可以为null

测试代码：

```
    @Autowired
    private Cat cat;

    @Autowired
    @Qualifier(value = "dog2")
    private Dog dog;
```

如果Autowire自动装配环境复杂，可以使用Qualifier配合使用

@Resource

```
    @Resource(name = "cat1")
    private Cat cat;

    @Resource
    private Dog dog;
```

小结：

Resource和Autowired的区别：

- 都可以用来自动装配，都可以放在属性字段上面
- Autowired通过bytype实现
- Resource默认通过byname实现，如果找不到会根据类型进行查找

8、使用注解开发

在Spring4之后，要使用注解开发，必须保证aop的包导入了

使用注解需要导入context约束，增加注解的支持！

```
<?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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-
context.xsd">

    <context:annotation-config/>

</beans>
```

1、bean

2、组建Component

```
//等价于<bean id="user" class="com.example.pojo.User"/>
//Component 组建

@Component
public class User {
    public String name="小鱼儿";
}
```

- 在类上面，说明这个类被Spring管理了，就是bean！！！

3、属性注入

```
//等价于<bean id="user" class="com.example.pojo.User"/>
//Component 组建

@Component
public class User {

    public String name;
    //相当于 <property name="name" value="xiaoyuer"/>
    @Value("xiaoyuer")
    public void setName(String name) {
        this.name = name;
    }
}
```

4、衍生注解

@Component有几个衍生注解，我们在web开发中，会按照mvc三层架构分层！！！

- dao【@Repository】
- Service【@Service】
- controller【@Controller】

  这四个注解功能都是一样的，都是代表将某个类注册到Spring中，装配Bean

5、自动装配

```
@Autowired:自动装配，通过类型
  - 如果Autowired不能唯一自动装配上属性，则需要通过@Qualifier（value="***"）
@Nullable: 字段标记了这个注解，说明这个字段可以为null
@Resource: 自动装配通过名字
```

6、小结

xml与注解：

- xml更加万能，适用于任何场合！维护简单方便
- 注解不是自己类使用不了，维护相对复杂！

xml与注解最佳实践

- xml用来管理bean；
- 注解只负责完成属性的注入；
- 我们在使用的过程中，只需要注意一个问题：必须让注解生效，就需要开启注解的支持

```
    <!--     指定要扫描的包，这个包下的注解就会生效-->
        <context:component-scan base-package="com.example"/>
        <context:annotation-config/>
```

9、使用java的方式配置Srping

我们现在要完全不使用Spring的xml配置了，全权交给java来做了！

JavaConfig是Spring的一个子项目，在Spring4之后，它成为了一个核心

配置文件：

```
    //这个也会被Spring容器托管，注册到容器中，因为他本来就是一个@Component
    //也代表这是一个配置类，就和我们之前看的beans.xml一样
    @Configuration
    @ComponentScan("com.example.pojo")
    @Import(BaiConfig2.class)
    public class BaiConfig {

        //注册一个bean，就相当于我们之前写的一个bean标签
        //这个方法的名字，就相当于bean标签中的id属性
        //这个方法的返回值，就相当于bean标签中的class属性
        @Bean
```

```
    public User getUser(){
        return new User();   //就是返回注入到bean的对象！！！
    }
}
```

实体类：

```
//等价于<bean id="user" class="com.example.pojo.User"/>
//Component 组建

//这里这个注解的意思，就是说明这个类被Spring接管了，注册到了容器中
@Component
public class User {

    public String name;

    public String getName() {
        return name;
    }

    //属性注入值
    @Value("小白白")
    public void setName(String name) {
        this.name = name;
    }

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

测试类

```
    @Test
    public void test1(){
        //如果完全使用了配置类方式去做，我们就只能通过AnnotationConfig上下文来获取容器，通过配置类的class对象进行加载
        ApplicationContext  context = new AnnotationConfigApplicationContext(BaiConfig.class);
        User getUser = (User) context.getBean("getUser");

        System.out.println(getUser.getName());
    }
```

这种纯java的方式，在Spring中随处可见

10、代理模式

中介：（黄牛、卖房子），帮一些人做一些事情

为什么要学习代理模式？因为这是SpringAOP的底层！【SpringAOP和SpringMVC】

代理模式的分类：

- 静态代理
- 动态代理

10.1 静态代理

角色分析：

- 抽象角色：一般会使用接口或者抽象类来解决
- 真实角色：被代理的角色
- 代理角色：代理真实角色，代理真实角色后，我们一般会做一些附属操作
- 客户：访问代理对象的人

接口/抽象

```
public interface Rent {
    public void rent();
}
```

真实人:

```
public class Host implements Rent{
    public void rent() {
        System.out.println("房东要出租房子! ");
    }
}
```

代理人:

```
public class Proxy implements Rent{
    Host host;

    public Proxy() {
    }

    public Proxy(Host host) {
        seeHouse();
        this.host = host;
        hetong();
        fare();
    }

    public void rent() {
        host.rent();
    }

    //看房
```

```
    public void seeHouse(){
        System.out.println("中介带看房子！");
    }

    //收中介费
    public void fare(){
        System.out.println("收中介费！");
    }

    //签合同
    public void hetong(){
        System.out.println("签租赁合同！");
    }
}
```

真是用户：

```
public class Client {
    public static void main(String[] args) {
        //房东要租房子
        Host host = new Host();
        //代理，中介帮房东找人，但是呢？代理角色一般会有一些附加操作
        Proxy proxy = new Proxy(host);

        //用户不用面对房东，直接找中介租房子
        proxy.rent();
    }
}
```

代理模式的好处：

- 可以使真实角色的操作更加纯粹，不用关注一些公共的业务
- 公共就交给了代理，实现了业务分工
- 公共业务发生扩张的时候，方便集中管理

缺点：

- 一个真实的角色就会产生一个代理角色，代码量翻倍，开发效率会变低

10.2 加深理解

10.3 动态代理

- 动态代理和静态代理角色一样
- 动态代理的代理类是动态生成的，不是我们直接写好的
- 动态代理分为两大类：基于接口的动态代理，基于类的动态代理
  - 基于接口：JDK动态代理【我们在这里使用】
  - 基于类：cglib
  - java字节码实现：javasist

需要了解两个类：Proxy（代理），InvocationHandler（调用处理程序）

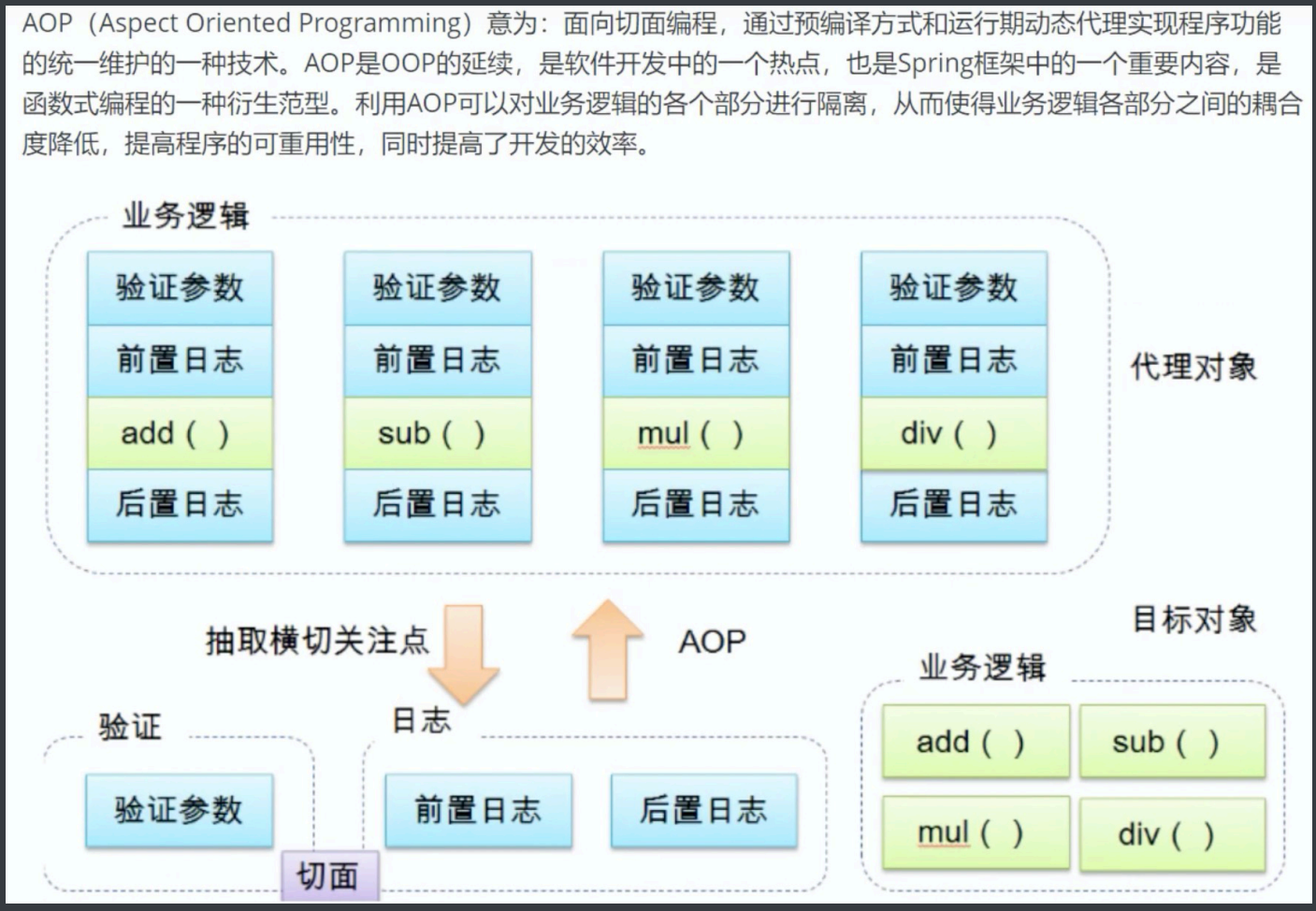
动态代理的好处：

- 可以使真实角色的操作更加纯粹！不用去关注一些公共的业务
- 公共也就交给代理角色，实现了业务的分工
- 公共业务发生扩展的时候，方便集中管理
- 一个动态代理类代理的是一个接口，一般就是对应一类业务
- 一个动态代理可以代理多个类，只需要实现一个接口即可

11、AOP

11.1 什么是AOP

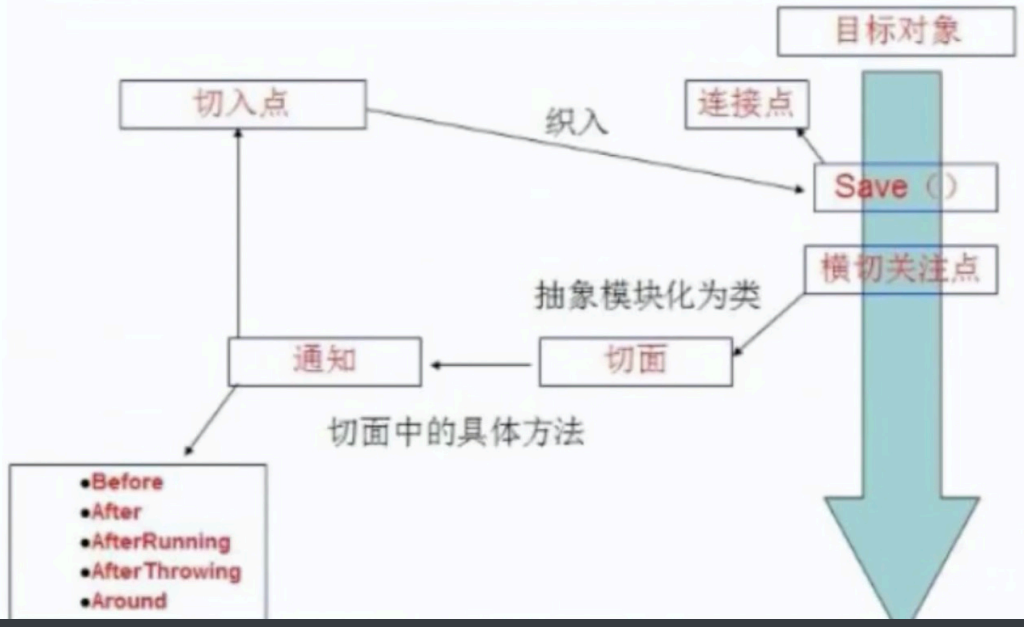
AOP（Aspect Oriented Programming）意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。



11.2 AOP在Spring中的作用

提供声明式事务；允许用户自定义切面

- 横切关注点：跨越应用程序多个模块的方法或功能。即是，与我们业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。如日志 , 安全 , 缓存 , 事务等等 ....
- 切面（ASPECT）：横切关注点 被模块化 的特殊对象。即，它是一个类。
- 通知（Advice）：切面必须要完成的工作。即，它是类中的一个方法。
- 目标（Target）：被通知对象。
- 代理（Proxy）：向目标对象应用通知之后创建的对象。
- 切入点（PointCut）：切面通知 执行的 “地点”的定义。
- 连接点（JointPoint）：与切入点匹配的执行点。



SpringAOP中，通过Advice定义横切逻辑，Spring中支持5种类型的Advice:

通知类型	连接点	实现接口
前置通知	方法方法前	org.springframework.aop.MethodBeforeAdvice
后置通知	方法后	org.springframework.aop.AfterReturningAdvice
环绕通知	方法前后	org.aopalliance.intercept.MethodInterceptor
异常抛出通知	方法抛出异常	org.springframework.aop.ThrowsAdvice
引介通知	类中增加新的方法属性	org.springframework.aop.IntroductionInterceptor

即 Aop 在 不改变原有代码的情况下，去增加新的功能 .

11.3 使用Spring实现AOP

【重点】使用AOP，需要先导入一个依赖

```
        <!--
    https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.4</version>
            <scope>runtime</scope>
        </dependency>
```

方式一：使用Spring的API接口【主要是SpringAPI接口实现】

方式二：使用自定义类实现【主要是切面定义】