Spring

Spring 是一个 IOC(DI)和 AOP 容器框架。

IOC：(Inversion of Control) 中文名是控制反转，是一种思想。控制反转的思想完全颠覆了应用程序组件获取资源的传统方式。

控制反转思想的内容：反转资源的获取方向——改由容器主动的将资源推送给需要的组件，开发人员不需要知道容器是如何创建资源对象的， 只需要提供接收资源的方式即可， 极大的降低了学习成本， 提高了开发的效率。 这种行为也称为查找的被动形式。

传统方式: 我想吃饭 我需要买菜做饭
控制反转: 我想吃饭 饭来张口

DI(Dependency Injection)： 依赖注入。DI 是对 IOC 的一种具体实现。

AOP(Aspect-Oriented Programming， 面向切面编程)： 是一种新的方法论， 是对传统 OOP(Object-Oriented Programming， 面向对象编程)的补充。

容器：

IOC思想必须基于IOC容器来完成， 而IOC容器在最底层实质上就是一个对象工厂。

IOC容器是一个创建对象、配置对象、连接对象、并管理对象的生命周期的一个容器，这里的对象即bean对象。

在Spring中说的Bean，说的是被容器管理的Bean。

Spring提供了IOC容器的两种实现方式：

① BeanFactory：IOC容器的基本实现，是Spring内部的基础设施，是面向Spring本身的，不是提供给开发人员使用的。
② ApplicationContext：BeanFactory的子接口，提供了更多高级特性。面向Spring的使用者，几乎所有场合都使用ApplicationContext而不是底层的BeanFactory。

Spring的使用：

HelloWorld

目标： 使用 Spring 创建Student对象， 为属性赋值

1. 创建 Student 类
2. 创建 Spring 配置文件

```
<!-- 使用bean元素定义一个由IOC容器创建的对象 -->
<!-- class属性指定用于创建bean的全类名 -->
<!-- id属性指定用于引用bean实例的标识 -->
<bean id="student" class="com.atguigu.helloworld.bean.Student">
<!-- 使用property子元素为bean的属性赋值 -->
<property name="studentId" value="1001"/>
<property name="stuName" value="Tom2015"/>
<property name="age" value="20"/>
</bean>
```

3. 通过 Spring 的 IOC 容器创建 Student 类实例

```
//1.创建IOC容器对象
ApplicationContext iocContainer =
new ClassPathXmlApplicationContext("helloworld.xml");
//2.根据id值获取bean实例对象
Student student = (Student) iocContainer.getBean("student");
//3.打印bean
System.out.println(student)
```

Spring中有两种类型的bean：

一种是普通bean，另一种是工厂bean，即FactoryBean。

> 工厂bean跟普通bean不同，对于工厂bean，容器调用getBean方法返回的不是工厂Bean的一个实例，其返回的是工厂bean的getObject方法所返回的对象。
> 对于普通bean，容器调用getBean方法返回的是普通Bean的一个实例。

AOP：

AOP(面向切面编程)：

将非核心代码写到一个类中，而不和核心代码写在同一个类中。然后利用动态代理在invoke方法中让核心代码所属的类的对象去调用核心代码，让非核心代码所属的类的对象去调用非核心代码。

术语：

目标类：

核心代码所属的类。一个核心代码对应于目标类中的一个目标方法。

切面类：

非核心代码所属的类。IOC容器扫描到一个类中的@Aspect注解时，IOC容器才会将这个类当作一个切面类。

增强(也叫通知Advice)：

切面类中的方法，一个非核心代码对应于一个增强。IOC容器扫描到切面中的@Before、@AfterReturning、@After、@Around、@AfterThrowing注解时，IOC容器才会将方法当作一个通知。

连接点：

如果在通知的参数列表添加JoinPoint参数(JoinPoint joinPoint)，则这个通知被调用时，IOC容器会将当前和这个通知合作的目标方法的信息封装成一个JoinPoint对象传进去。

示例：

```
@Before(value="test()")
    public void beforeMethod(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();//获取方法的参数
        String methodName = joinPoint.getSignature().getName();//获取方法名

System.out.println("method:"+methodName+",arguments:"+Arrays.toString(args));
    }
```

切入点：

通知(@Advice)和切入点(@Pointcut)共同组成了切面(@Aspect)。

切入点表达式：

切入点表达式的语法格式

```
execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表]))
```

（1）在编写 AspectJ 切面时，可以直接在通知注解中书写切入点表达式，但同一个切入点表达式可能会在多个通知中重复出现；

（2）在 AspectJ 切面中，可以通过 @PointCut 注解将一个切入点声明或简单方法。切入点的方法体通常是空的，因为将切入点定义与应用程序逻辑混在一起是不合理的；

（3）切入点方法的访问控制符同时也控制着这个切入点的可见性。如果切入点要在多个切面中共用，最好将它们集中在一个公共的类中，在这种情况下，它们必须被声明为 public，在引入这个切入点时，必须将类名也包括在内。如果类没有与这个切面放在同一个包中，还必须包含包名。

（4）其他通知可以通过方法名称引入该切入点；

实现步骤：

1. 随便声明一个没有实现的返回void的空方法
2. 给方法上标注@Pointcut注解
3. 在需要用到这个内容的注解execution属性内写入这个空方法的方法名

如下图：

ApplicationContext.getBean返回的对象是代理对象还是原生对象?

ApplicationContext.getBean方法默认返回的是原生对象。

但是如果一个JavaBean对象是有代理类对象的，那么ApplicationContext.getBean("这个JavaBean对象对应的id属性值")就不是返回这个JavaBean对象，而是返回这个javaBean对象的代理对象。

DI:

注入分为：自动注入和手动注入。自动注入就是不需要指定注入的内容，容器会自己去找。手动注入就是需要指定注入的内容。

手动注入：按注入方式分，可分为：1. set注入 2.构造方法注入
按要注入的属性的类型分，可分为：1. 对简单类型属性的手动注入 2. 对引用类型属性的手动注入。

简单类型就是 基本数据类型 + String类型
对简单类型的手动注入除了set注入和构造方法注入，还有另外一种方式：就是用注解@value。

自动注入：只有引用类型属性有自动注入，简单类型属性是没有自动注入的。
自动注入按注入方式可分为：1. byName方式 2.byType方式

配置自动注入有两种方式：1.在配置文件上配置
byName方式: autowire="byName"
byType方式: autowire="byType"
2.注解
byName方式: @Qualifier
byType方式: @Autowired

自动注入的流程：IOC容器发现成员变量是要被自动注入的时候，会给这个成员变量到容器中找Bean对象。然后将找到的对象的地址直接赋值给这个成员变量，而不是通过这个成员变量的set方法，所以不需要定义这个成员变量的set方法。

@Autowired注解

@Autowired注解的位置： 1. 成员变量上 2.方法上 3.注解上
IOC容器扫描到在成员变量上的@Autowired注解时，会给这个成员变量去找JavaBean对象。然后将找到的对象的地址直接赋值给这个成员变量，而不是通过这个成员变量的set方法，所以不需要定义这个成员变量的set方法。找的规则是找类型与这个成员变量的类型相同或是成员变量类型子类的对象。如果找不到则报错。如果找到了多个，则从这些对象中找id是与这个成员变量名相同的对象，如果找不到则报错。
如果想"找类型与这个成员变量的类型相同或是成员变量类型子类的对象"找不到时不报错，就将@Autowired的required属性值设为false。