课堂主题

Spring IoC和AOP的应用、事务支持

课堂目标

- 搞明白什么是IoC、什么是DI、什么是AOP、什么是Spring容器
- 掌握Spring IoC/DI基于XML方式的应用
- 掌握Spring IoC/DI基于XML和注解混合的使用
- 掌握Spring IoC/DI基于纯注解的使用
- 掌握Spring基于AspectJ的AOP使用之XML方式
- 掌握Spring基于AspectJ的AOP使用之ML和注解混合方式
- 掌握Spring基于AspectJ的AOP使用之纯注解方式的应用
- 掌握Spring声明式事务的使用之XML方式
- 掌握Spring声明式事务的使用之XML和注解混合方式
- 掌握Spring声明式事务的使用之纯注解方式的应用

知识要点

- IoC注解使用方法
- 常用注解
  - IoC注解（创建对象）
    - Component注解
    - Controller&Service&Repository注解
  - DI注解（依赖注入）
    - @Autowired
    - @Qualifier
    - @Resource
    - @Inject
    - @Value
    - @Autowired、@Resource、@Inject区别
    - 改变Bean作用范围的注解
    - 生命周期相关注解
- 关于注解和XML的选择问题

- 基于纯注解方式的使用
- 注解和XML混合开发遗留的问题
- @Configuration
- @Bean
- @ComponentScan
- @PropertySource
- @Import
- 创建纯注解方式上下文容器

核心高级篇

- AOP介绍
- 什么是AOP
- 为什么使用AOP
- AOP相关术语介绍
  - 术语解释
  - 图示说明
- AOP实现之AspectJ（了解）
- AOP实现之Spring AOP(了解)
  - 实现原理分析
    - JDK动态代理
    - Cglib动态代理
  - 使用

- 基于AspectJ的AOP使用
- 添加依赖
- 编写目标类和目标方法
- 使用XML实现
  - 实现步骤
  - 切入点表达式
  - 通知类型
- 使用注解实现
  - 实现步骤
  - 环绕通知注解配置
  - 定义通用切入点
- 纯注解方式

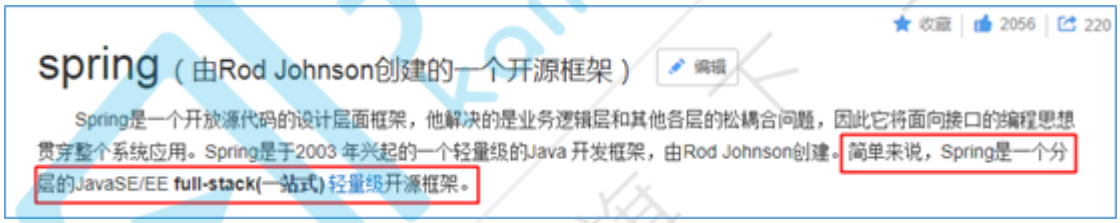
组件支撑篇

- 整合Junit
- 单元测试问题
- 解决思路分析
- 具体实现

介绍篇

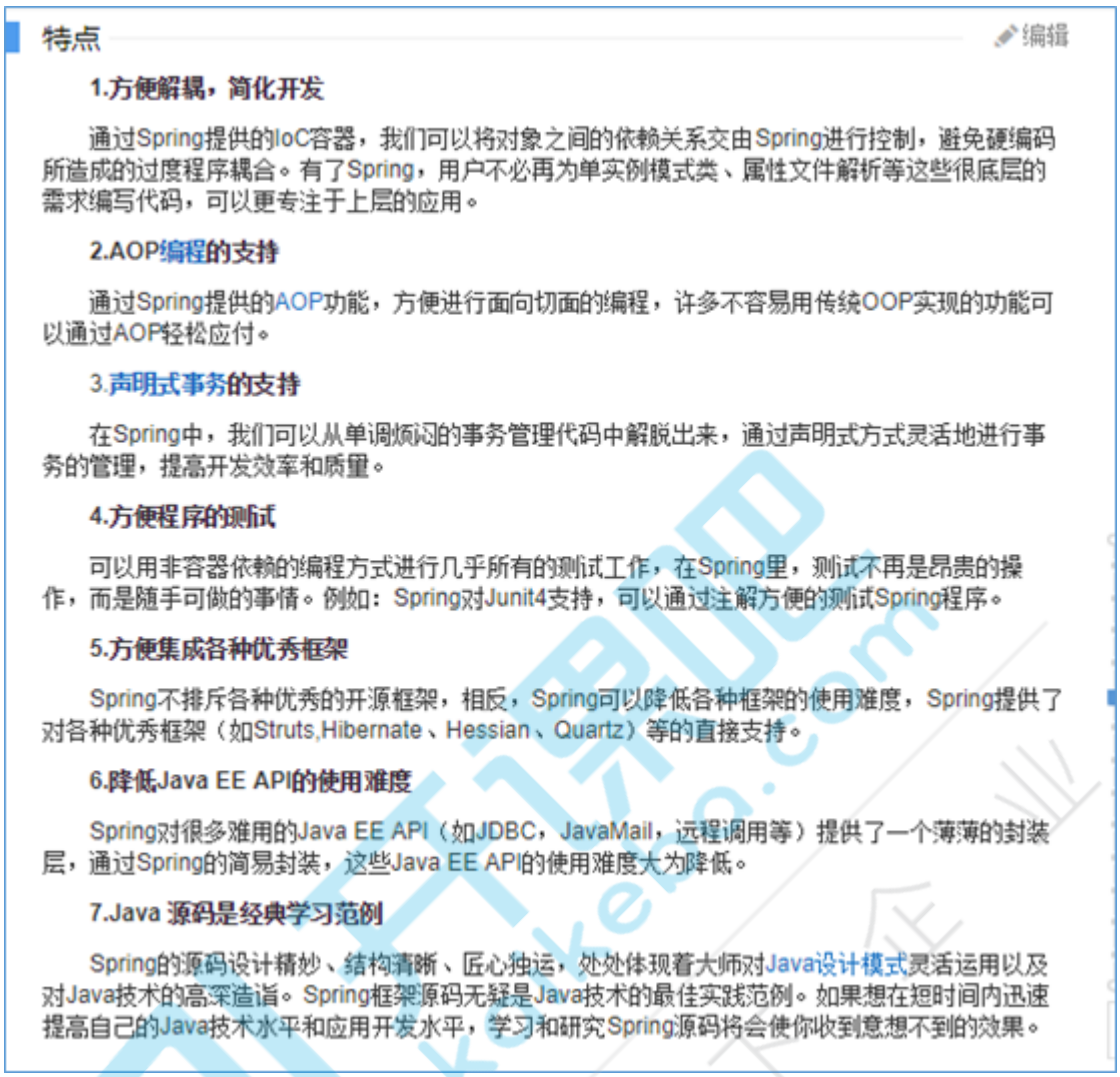
什么是Spring

- 百度百科中的介绍



- Spring官方网址：http://spring.io/
- 我们经常说的Spring其实指的是 `Spring Framework`（spring 框架）。

为什么学习Spring



特点 编辑

1.方便解耦，简化开发

通过Spring提供的IoC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

2.AOP编程的支持

通过Spring提供的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。

3.声明式事务的支持

在Spring中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

4.方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不再是昂贵的操作，而是随手可做的事情。例如：Spring对Junit4支持，可以通过注解方便的测试Spring程序。

5.方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架（如Struts,Hibernate、Hessian、Quartz）等的直接支持。

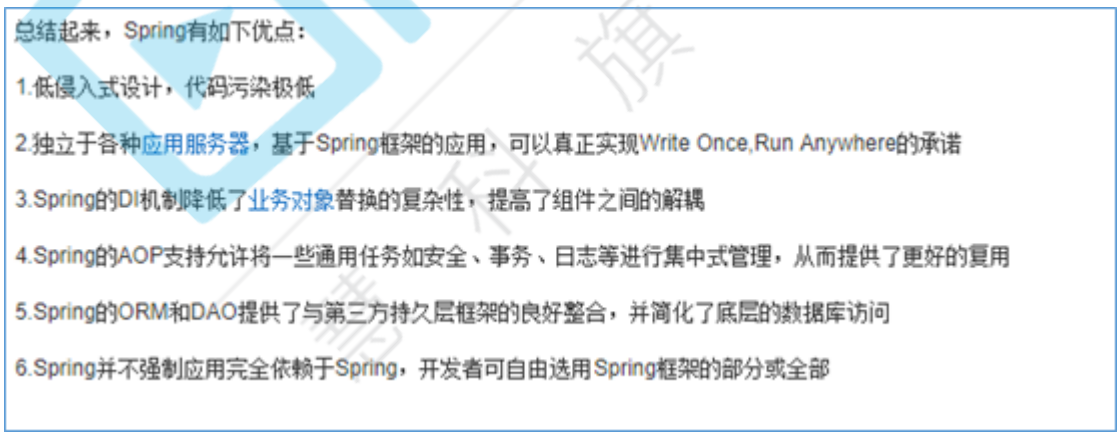
6.降低Java EE API的使用难度

Spring对很多难用的Java EE API（如JDBC，JavaMail，远程调用等）提供了一个薄薄的封装层，通过Spring的简易封装，这些Java EE API的使用难度大为降低。

7.Java 源码是经典学习范例

Spring的源码设计精妙、结构清晰、匠心独运，处处体现着大师对Java设计模式灵活运用以及对Java技术的高深造诣。Spring框架源码无疑是Java技术的最佳实践范例。如果想在短时间内迅速提高自己的Java技术水平和应用开发水平，学习和研究Spring源码将会使你收到意想不到的效果。

- 好处：



总结起来，Spring有如下优点：

1.低侵入式设计，代码污染极低

2.独立于各种应用服务器，基于Spring框架的应用，可以真正实现Write Once,Run Anywhere的承诺

3.Spring的DI机制降低了业务对象替换的复杂性，提高了组件之间的解耦

4.Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中式管理，从而提供了更好的复用

5.Spring的ORM和DAO提供了与第三方持久层框架的良好整合，并简化了底层的数据库访问

6.Spring并不强制应用完全依赖于Spring，开发者可自由选用Spring框架的部分或全部

什么是耦合和内聚

- 耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。

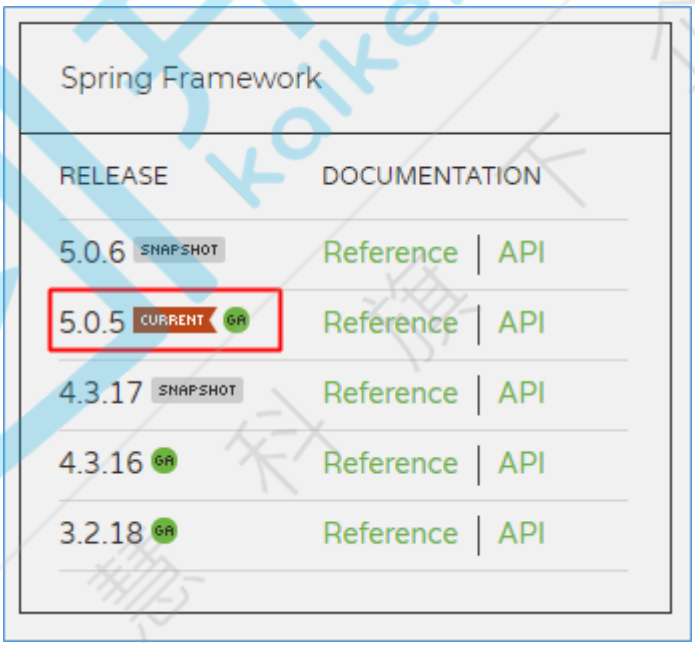
```
在软件工程中，耦合指的就是就是对象之间的依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。

软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。

划分模块的一个准则就是高内聚低耦合。
```

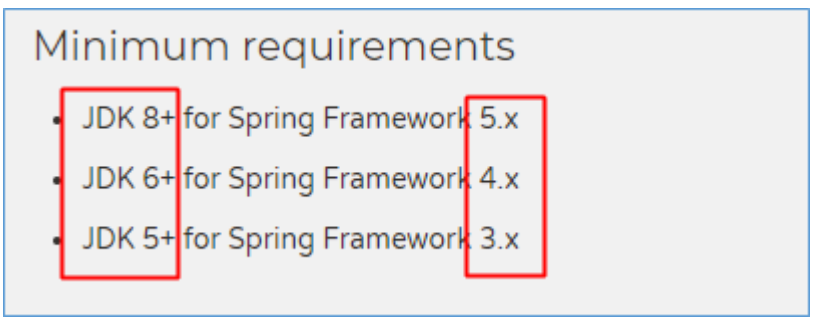
- 内聚标志一个模块内各个元素彼此结合的紧密程度，它是信息隐蔽和局部化概念的自然扩展。

```
内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当恰好做一件事。

内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。

在进行软件设计时，应力争做到高内聚，低耦合。
```

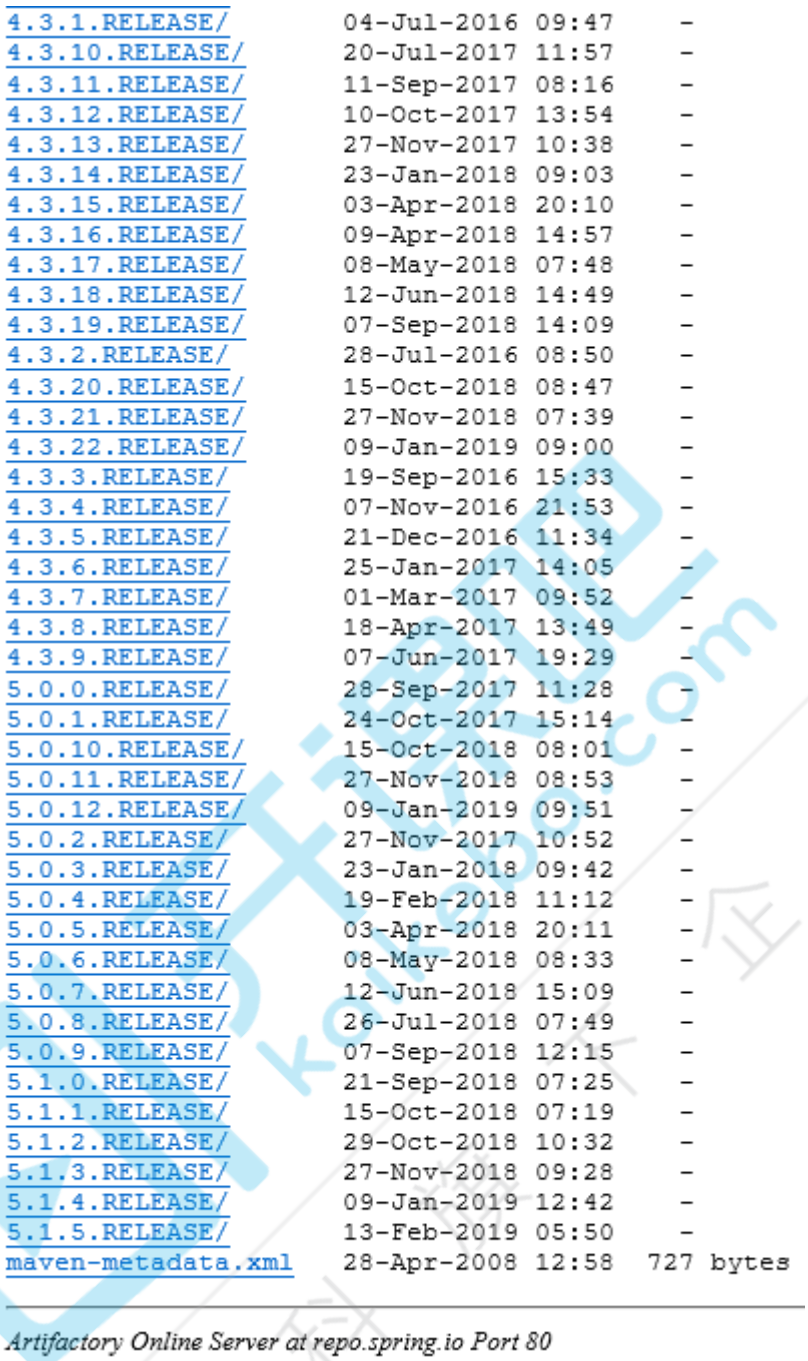
Spring版本介绍

- spring的官网：http://spring.io/
- spring framework的网址：https://projects.spring.io/spring-framework/





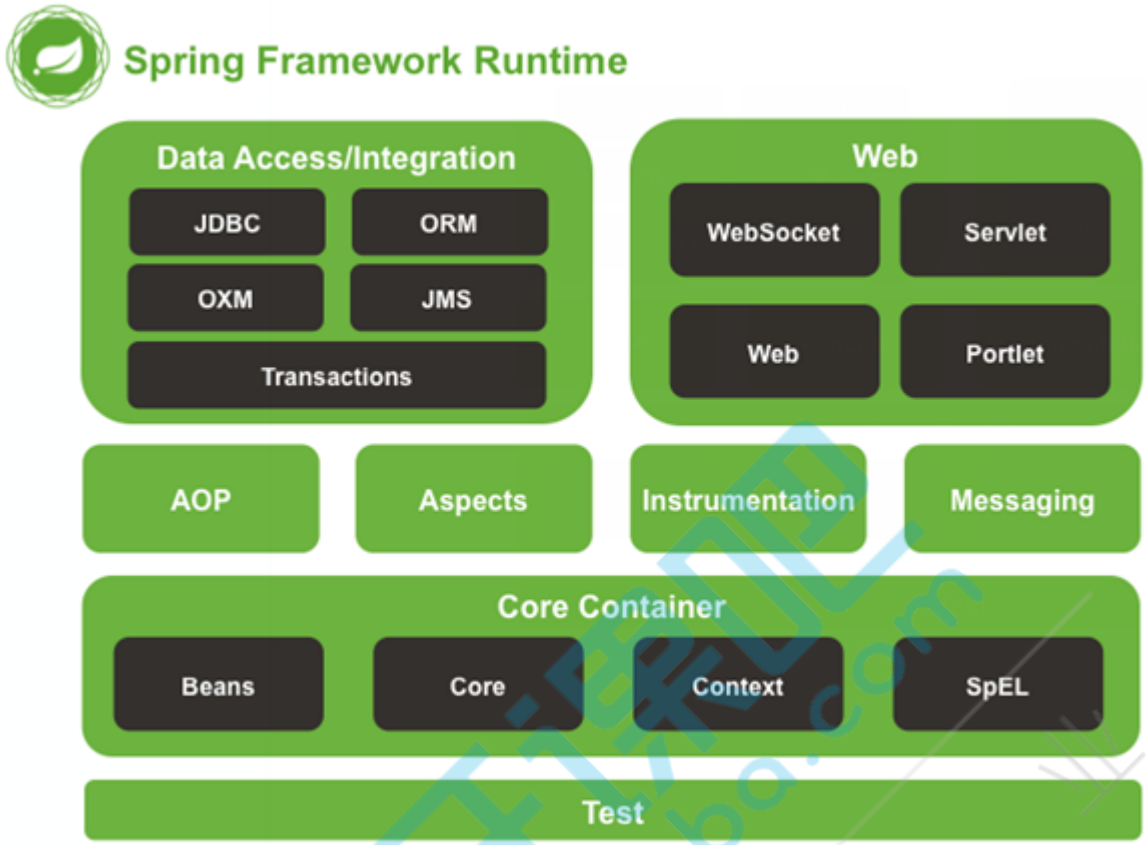
- jar包下载地址：http://repo.spring.io/release/org/springframework/spring/



4.3.1.RELEASE/	04-Jul-2016 09:47	-
4.3.10.RELEASE/	20-Jul-2017 11:57	-
4.3.11.RELEASE/	11-Sep-2017 08:16	-
4.3.12.RELEASE/	10-Oct-2017 13:54	-
4.3.13.RELEASE/	27-Nov-2017 10:38	-
4.3.14.RELEASE/	23-Jan-2018 09:03	-
4.3.15.RELEASE/	03-Apr-2018 20:10	-
4.3.16.RELEASE/	09-Apr-2018 14:57	-
4.3.17.RELEASE/	08-May-2018 07:48	-
4.3.18.RELEASE/	12-Jun-2018 14:49	-
4.3.19.RELEASE/	07-Sep-2018 14:09	-
4.3.2.RELEASE/	28-Jul-2016 08:50	-
4.3.20.RELEASE/	15-Oct-2018 08:47	-
4.3.21.RELEASE/	27-Nov-2018 07:39	-
4.3.22.RELEASE/	09-Jan-2019 09:00	-
4.3.3.RELEASE/	19-Sep-2016 15:33	-
4.3.4.RELEASE/	07-Nov-2016 21:53	-
4.3.5.RELEASE/	21-Dec-2016 11:34	-
4.3.6.RELEASE/	25-Jan-2017 14:05	-
4.3.7.RELEASE/	01-Mar-2017 09:52	-
4.3.8.RELEASE/	18-Apr-2017 13:49	-
4.3.9.RELEASE/	07-Jun-2017 19:29	-
5.0.0.RELEASE/	28-Sep-2017 11:28	-
5.0.1.RELEASE/	24-Oct-2017 15:14	-
5.0.10.RELEASE/	15-Oct-2018 08:01	-
5.0.11.RELEASE/	27-Nov-2018 08:53	-
5.0.12.RELEASE/	09-Jan-2019 09:51	-
5.0.2.RELEASE/	27-Nov-2017 10:52	-
5.0.3.RELEASE/	23-Jan-2018 09:42	-
5.0.4.RELEASE/	19-Feb-2018 11:12	-
5.0.5.RELEASE/	03-Apr-2018 20:11	-
5.0.6.RELEASE/	08-May-2018 08:33	-
5.0.7.RELEASE/	12-Jun-2018 15:09	-
5.0.8.RELEASE/	26-Jul-2018 07:49	-
5.0.9.RELEASE/	07-Sep-2018 12:15	-
5.1.0.RELEASE/	21-Sep-2018 07:25	-
5.1.1.RELEASE/	15-Oct-2018 07:19	-
5.1.2.RELEASE/	29-Oct-2018 10:32	-
5.1.3.RELEASE/	27-Nov-2018 09:28	-
5.1.4.RELEASE/	09-Jan-2019 12:42	-
5.1.5.RELEASE/	13-Feb-2019 05:50	-
maven-metadata.xml	28-Apr-2008 12:58	727 bytes

Artifactory Online Server at repo.spring.io Port 80

Spring体系结构



Spring核心概念介绍

- IoC（核心中的核心）：Inverse of Control，控制反转。对象的创建权力由程序反转给Spring框架。
- DI：Dependency Injection，依赖注入。在Spring框架负责创建Bean对象时，动态的将依赖对象注入到Bean组件中！！
- AOP：Aspect Oriented Programming，面向切面编程。在不修改目标对象的源代码情况下，增强IoC容器中Bean的功能。
- Spring容器：指的就是IoC容器，底层也就是一个BeanFactory。

核心基础篇

基于XML的使用

IoC配置

在Spring 的XML文件中通过一个bean标签，完成IoC的配置。

bean标签介绍

- bean标签作用：

  用于配置被spring容器管理的bean的信息。

默认情况下它调用的是类中的【无参构造函数】。如果没有无参构造则不能创建成功。

- bean标签属性：
  - id：给对象在**容器中**提供一个唯一标识。用于获取对象。
  - class：指定类的全限定名。用于反射创建对象。默认情况下**调用无参构造函数。**
  - **init-method**：指定类中的初始化方法名称。
  - **destroy-method**：指定类中销毁方法名称。比如DataSource的配置中一般需要指定destroy-method=“close”。
  - scope：指定对象的作用范围。
    - singleton :**默认值**，单例的（在整个容器中只有一个对象）,生命周期如下：
      - 对象出生：当应用加载，创建容器时，对象就被创建了。
      - 对象活着：只要容器在，对象一直活着。
      - 对象死亡：当应用卸载，销毁容器时，对象就被销毁了。
    - prototype :多例的.每次访问对象时，都会重新创建对象实例。生命周期如下：
      - 对象出生：当使用对象时，创建新的对象实例。
      - 对象活着：只要对象在使用中，就一直活着。
      - 对象死亡：当对象长时间不用时，被 java 的垃圾回收器回收了。
    - request :将Spring 创建的 Bean 对象存入到 request 域中.
    - session :将Spring 创建的 Bean 对象存入到 session 域中.
    - global session :WEB 项目中,应用在 Portlet 环境.如果没有 Portlet 环境那么 globalSession 相当于 session.

bean实例化的三种方式

- **第一种：使用默认无参构造函数（重点）**

  在默认情况下：它会根据默认无参构造函数来创建类对象。

  如果 bean 中没有默认无参构造函数，将会创建失败。

```
<bean id="userService" class="com.kkb.spring.service.UserServiceImpl"/>
```

- 第二种：静态工厂（了解）

  使用 StaticFactory 类中的静态方法 createUserService 创建对象，并存入spring 容器：

```
/**
* 模拟一个静态工厂，创建业务层实现类
*/
public class StaticFactory {
    public static UserService createUserService(){
        return new UserServiceImpl();
    }
}

```

```xml
<bean id="userService" class="com.kkb.spring.factory.StaticFactory" factory-method="createUserService"></bean>
```

配置说明：

```
<!-- 此种方式是:
使用 StaticFactory 类中的静态方法 createUserService 创建对象，并存入 spring 容器
    id 属性：指定 bean 的 id，用于从容器中获取
    class 属性：指定静态工厂的全限定类名
    factory-method 属性：指定生产对象的静态方法
-->
```

- 第三种：实例工厂（了解）

```java
/**
*   模拟一个实例工厂，创建业务层实现类
*   此工厂创建对象，必须现有工厂实例对象，再调用方法
*/
public class InstanceFactory {
    public UserService createUserService(){
        return new UserServiceImpl();
    }
}

```

```xml
<bean id="instancFactory" class="com.kkb.factory.InstanceFactory"></bean>

<bean id="userService" factory-bean="instancFactory" factory-method="createUserService"></bean>
```

```
<!-- 此种方式是：

    * 先把工厂的创建交给 spring 来管理。
    * 然后在使用工厂的 bean 来调用里面的方法

        factory-bean 属性：用于指定实例工厂 bean 的 id。
        factory-method 属性：用于指定实例工厂中创建对象的方法。
-->
```

DI配置

概述

- 什么是依赖

```
依赖指的就是Bean实例中的属性

依赖（属性）分为：简单类型（8种基本类型和String类型）的属性、POJO类型的属性、集合数组类型的属性。
```

- 什么是依赖注入

```
依赖注入：Dependency Injection。它是 spring 框架核心 IoC 的具体实现。
```

- 为什么要进行依赖注入

```
我们的程序在编写时，通过控制反转，把对象的创建交给了 spring，但是代码中不可能出现没有依赖的情况。

那如果一个bean中包含了一些属性，那么spring帮我们实例化了bean对象之后，也需要将对应的属性信息进行赋值操作，这种属性赋值操作，就是所谓的依赖注入（获取值、注入属性）

```

依赖注入的方式

构造函数注入

*顾名思义，就是使用类中的构造函数，给成员变量赋值。*注意：赋值的操作不是我们自己做的，而是通过配置的方式，让 spring 框架来为我们注入。

```java
public class UserServiceImpl implements UserService {
    private int id;
    private String name;

    public UserServiceImpl(int id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public void saveUser() {
        System.out.println("保存用户:id为"+id+"，name为"+name+"    Service实现");
    }
}
```

```xml
<bean id="userService" class="com.kkb.spring.service.UserServiceImpl">
        <constructor-arg name="id" value="1"></constructor-arg>
        <constructor-arg name="name" value="zhangsan"></constructor-arg>
</bean>
```

```
<!-- 使用构造函数的方式，给 service 中的属性传值要求：类中需要提供一个对应参数列表的构造函数。
    涉及的标签：constructor-arg
        * index:指定参数在构造函数参数列表的索引位置
        * name:指定参数在构造函数中的名称
        * value:它能赋的值是基本数据类型和 String 类型
        * ref:它能赋的值是其他 bean 类型，也就是说，必须得是在配置文件中配置过的 bean
--!>
```

set方法注入（重点）

- 手动装配方式（XML方式）

```
- 需要配置bean标签的子标签property
- 需要配置的bean中指定setter方法。
```

- 自动装配方式（注解方式，后面讲解）

```
- @Autowired：
  - 作用一：查找实例，从spring容器中根据Bean的类型（byType）获取实例。
  - 作用二：赋值，将找到的实例，装配给另一个实例的属性值。
  - 注意事项：一个java类型在同一个spring容器中，只能有一个实例
- @Resource：
  - 作用一：查找实例，从spring容器中根据Bean的名称（byName）获取实例。
  - 作用二：赋值，将找到的实例，装配给另一个实例的属性值。
- @Inject：

```

使用p名称空间注入数据

本质上还是调用set方法，自行了解

1. 步骤一：需要先引入 p 名称空间

   ```
   在schema的名称空间中加入该行：xmlns:p="http://www.springframework.org/schema/p"
   ```

2. 步骤二：使用p名称空间的语法

   ```
   p:属性名 = ""

   p:属性名-ref = ""
   ```

3. 步骤三：测试

```xml
<bean id="person" class="com.kkb.spring.demo.Person" p:pname="老王" p:car2-ref="car2"/>
<bean id="car2" class="com.kkb.spring.demo.Car2" />
```

依赖注入不同类型的属性

简单类型（value）

```xml
<bean id="userService" class="com.kkb.spring.service.UserServiceImpl">
        <property name="id" value="1"></property>
        <property name="name" value="zhangsan"></property>
</bean>
```

引用类型（ref）

ref就是reference的缩写，是引用的意思。

```xml
<bean id="userService" class="com.kkb.spring.service.UserServiceImpl">
        <property name="userDao" ref="userDao"></constructor-arg>
</bean>
<bean id="userDao" class="com.kkb.spring.dao.UserDaoImpl"></bean>
```

集合类型（数组）

1. 如果是数组或者List集合，注入配置文件的方式是一样的

```xml
    <bean id="collectionBean" class="com.kkb.demo5.CollectionBean">
        <property name="arrs">
            <list>
                <!-- 如果集合内是简单类型，使用value子标签，如果是POJO类型，则使用bean标签 -->
                <value>美美</value>
                <value>小风</value>
              <bean></bean>
            </list>
        </property>
    </bean>
```

2. 如果是Set集合，注入的配置文件方式如下：

```xml
    <property name="sets">
        <set>
            <!-- 如果集合内是简单类型，使用value子标签，如果是POJO类型，则使用bean标签 -->
            <value>哈哈</value>
            <value>呵呵</value>
        </set>
    </property>
```

3. 如果是Map集合，注入的配置方式如下：

```
    <property name="map">
        <map>
            <entry key="老王2" value="38"/>
            <entry key="凤姐" value="38"/>
            <entry key="如花" value="29"/>
        </map>
    </property>
```

4. 如果是Properties集合的方式，注入的配置如下：

```
    <property name="pro">
        <props>
            <prop key="uname">root</prop>
            <prop key="pass">123</prop>
        </props>
    </property>
```

基于注解和XML混合方式的使用

```
- 学习基于注解的 IoC 配置，大家脑海里首先得有一个认知，即注解配置和 xml 配置要实现的功能都是一样的，都是要降低程序间的耦合。只是配置的形式不一样。
- 关于实际的开发中到底使用xml    还是注解，每家公司有着不同的使用习惯。所以这两种配置方式我们都需要掌握。
- 我们在讲解注解配置时，采用上一章节的案例，把 spring 的 xml 配置内容改为使用注解逐步实现。
```

IoC注解使用方法

- 第一步：spring配置文件中，配置context:component-scan标签

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

    <!-- 开启注解并扫描指定包中带有注解的类 -->
    <context:component-scan base-package="com.kkb.spring.service"></context:component-scan>
</beans>
```

- 第二步：类上面加上注解@Component，或者它的衍生注解@Controller、@Service、@Repository

```
@Service
public class UserServiceImpl implements UserService {

    private int id;
```

常用注解

IoC注解（创建对象）

相当于：

```
<bean id="" class="">
```

Component注解

- 作用：

```
把资源让 spring 来管理。相当于在 xml 中配置一个 bean。
```

- 属性：

```
value：
    指定 bean 的 id。如果不指定 value 属性，默认 bean 的 id 是当前类的类名，首字母小写。
```

Controller&Service&Repository注解

他们三个注解都是针对@Component的衍生注解，他们的作用及属性都是一模一样的。他们只不过是提供了更加明确的语义化。

注意：如果注解中有且只有一个属性要赋值时，且名称是 value，value 在赋值时可以不写。

```
 - @Controller：一般用于表现层的注解。
 - @Service：一般用于业务层的注解。
 - @Repository：一般用于持久层的注解。
```

DI注解（依赖注入）

相当于：

```
<property name="" ref="">
<property name="" value="">
```

@Autowired

- @Autowired默认按类型装配（byType），
- @Autowired是由AutowiredAnnotationBeanPostProcessor类实现
- @Autowired是spring自带的注解

- @Autowired默认情况下要求依赖对象必须存在，如果需要允许null值，可以设置它的required属性为false，如：@Autowired(required=false)
- 如果我们想按名称装配（byName）可以结合 `@Qualifier` 注解进行使用

@Qualifier

- 在自动按照类型注入的基础之上，再按照 Bean 的 id 注入。
- 它在给字段注入时不能独立使用，必须和@Autowire 一起使用；
- 但是给方法参数注入时，可以独立使用。

@Resource

- @Resource默认按名称装配（byName），可以通过@Resource的name属性指定名称，如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，当找不到与名称匹配的bean时才按照类型进行装配。
- @Resource属于J2EE JSR250规范的实现
- 但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

推荐使用@Resource注解，因为它是属于J2EE的，减少了与spring的耦合。这样代码看起就比较优雅。

@Inject

- @Inject是根据类型进行自动装配的，如果需要按名称进行装配，则需要配合@Named；
- @Inject是JSR330中的规范，需要导入javax.inject.Inject;实现注入。
- @Inject可以作用在变量、setter方法、构造函数上。

@Value

- 给基本类型和String类型注入值
- 可以使用占位符获取属性文件中的值。

```
@value(“${name}”)//name是properties文件中的key
private String name;
```

@Autowired、@Resource、@Inject区别

1. @Autowired是spring自带的，@Inject是JSR330规范实现的，@Resource是JSR250规范实现的，需要导入不同的包
2. @Autowired、@Inject用法基本一样，不同的是@Autowired有一个request属性
3. @Autowired、@Inject是默认按照类型匹配的，@Resource是按照名称匹配的
4. @Autowired如果需要按照名称匹配需要和@Qualifier一起使用，@Inject和@Name一起使用

改变Bean作用范围的注解

- @Scope：指定 bean 的作用范围，相当于下面的配置：

```
<bean id="" class="" scope="">
```

- 属性：

```
value：指定范围的值。取值：singleton prototype request session globalsession
```

生命周期相关注解

- @PostConstruct
- @PreDestroy

相当于：

```
<bean id="" class="" init-method="" destroy-method="" />
```

关于注解和XML的选择问题

- 注解的优势：

配置简单，维护方便（我们找到类，就相当于找到了对应的配置）。

- XML 的优势：

修改时，不用改源码。不涉及重新编译和部署。

- Spring 管理 Bean 方式的比较：

	基于XML配置	基于注解配置
Bean定义	<bean id="..." class="..." />	@Component 衍生类@Repository @Service @Controller
Bean名称	通过 id或name 指定	@Component("person")
Bean注入	<property> 或者 通过p命名空间	@Autowired 按类型注入 @Qualifier按名称注入
生命过程、Bean作用范围	init-method destroy-method 范围 scope属性	@PostConstruct 初始化 @PreDestroy 销毁 @Scope设置作用范围
适合场景	Bean来自第三方，使用其它	Bean的实现类由用户自己开发

基于纯注解方式的使用

注解和XML混合开发遗留的问题

想一想能不能将以下这些bean的配置都从xml中去掉，并且最终将XML也去掉。如果可以，那么我们就可以脱离xml配置了。

- 注解扫描配置（能不能去掉）

```
<!-- 开启注解并扫描指定包中带有注解的类 -->
<context:component-scan base-package="com.kkb.spring.service"/>
<context:property-placeholder src=""></context:property-placeholder>
```

- 非自定义的Bean配置（比如：SqlSessionFactory和BasicDataSource配置）

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" value="dataSource"></property>
</bean>
```

- 去掉XML后，如何创建ApplicationContext

之前创建ApplicationContext都是通过读取XML文件进行创建的。

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

@Configuration

```
相当于spring的XML配置文件。

从Spring3.0开始可以使用@Configuration定义配置类，可替换xml配置文件

配置类内部包含有一个或多个被@Bean注解的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义对象，初始化Spring容器。
```

- 属性：

  value:用于指定配置类的字节码

- 示例代码：

```
    @Configuration
    public class SpringConfiguration {
        //spring容器初始化时，会调用配置类的无参构造函数
        public SpringConfiguration(){
            System.out.println("容器启动初始化。。。");
        }
    }
```

@Bean

```
相当于<bean>标签

作用为：注册bean对象，主要用来配置非自定义的bean，比如DruidDataSource、SqlSessionFactory

@Bean标注在方法上(返回某个实例的方法)
```

- 属性：

```
name：给当前@Bean 注解方法创建的对象指定一个名称(即 bean 的 id)。如果不指定，默认与标注的方法名相同。

@Bean注解默认作用域为单例singleton作用域，可通过@Scope(“prototype”)设置为原型作用域；
```

- 示例代码：

```
    @Configuration
    public class SpringConfiguration {
        //spring容器初始化时，会调用配置类的无参构造函数
        public SpringConfiguration(){
            System.out.println(“容器启动初始化。。。”);
        }
        @Bean
        @Scope(“prototype”)
        public SqlSessionFactory userService(){
            SqlSessionFactory sqlSessionFactory = new DefaultSqlSessionFactory();
            sqlSessionFactory.setxxx();
            return sqlSessionFactory;
        }
        @Bean
        @Scope(“prototype”)
        public SqlSessionFactory userService(){
            SqlSessionFactory sqlSessionFactory = new DefaultSqlSessionFactory();
            sqlSessionFactory.setxxx();
            return sqlSessionFactory;
        }
    }
```

@ComponentScan

```
相当于context:component-scan标签

组件扫描器，扫描@Component、@Controller、@Service、@Repository注解的类。

该注解是编写在类上面的，一般配合@Configuration注解一起使用。
```

- 属性：

```
basePackages：用于指定要扫描的包。

value：和basePackages作用一样。
```

- 示例代码：

  Bean类（Service类）：

```java
@Service
public class UserServiceImpl implements UserService {
    @Override
    public void saveUser() {
        System.out.println("保存用户    Service实现");
    }
}
```

配置类：

```java
@Configuration
@ComponentScan(basePackages="com.kkb.spring.service")
public class SpringConfiguration {
    public SpringConfiguration() {
        System.out.println("容器初始化...");
    }

//  @Bean
//  @Scope("prototype")
//  public UserService userService() {
//      return new UserServiceImpl(1,"张三");
//  }
}
```

@PropertySource

```
相当于context:property-placeholder标签

编写在类上面，作用是加载properties配置文件
```

- 属性

```
value[]：用于指定properties文件路径，如果在类路径下，需要写上classpath
```

- 示例代码

配置类：

```java
@Configuration
@PropertySource("classpath:jdbc.properties")
public class JdbcConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
```

```
    private String password;

    /**
     * 创建一个数据源，并存入 spring 容器中
     *
     * @return
     */
    @Bean(name = "dataSource")
    public DataSource createDataSource() {
        try {
            ComboPooledDataSource ds = new ComboPooledDataSource();
            ds.setDriverClass(driver);
            ds.setJdbcUrl(url);
            ds.setUser(username);
            ds.setPassword(password);
            return ds;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

properties文件：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql:///spring
jdbc.username=root
jdbc.password=root
```

- 问题：

```
当系统中有多个配置类时怎么办呢？想一想之前使用XML配置的时候是如何解决该问题的。
```

@Import

```
相当于spring配置文件中的<import>标签

用来组合多个配置类,在引入其他配置类时，可以不用再写@Configuration 注解。当然，写上也没问题。
```

- 属性

```
value：用来指定其他配置类的字节码文件
```

- 示例代码：

```java
@Configuration
@ComponentScan(basePackages = "com.kkb.spring")
@Import({ JdbcConfig.class })
public class SpringConfiguration {
}

@Configuration
@PropertySource("classpath:jdbc.properties")
public class JdbcConfig {
}
```

创建纯注解方式上下文容器

- Java应用（AnnotationConfigApplicationContext）

```java
ApplicationContext context = new
 AnnotationConfigApplicationContext(SpringConfiguration.class);
UserService service = context.getBean(UserService.class);
service.saveUser();
```

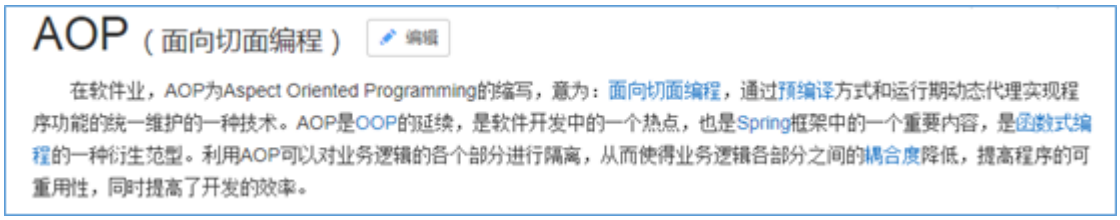
- Web应用（AnnotationConfigWebApplicationContext，后面讲解）

```xml
<web-app>
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.
            support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            com.kkb.spring.test.SpringConfiguration
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

核心高级篇

AOP介绍

什么是AOP



- 在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程
- 作用：在不修改目标类代码的前提下，可以通过AOP技术去增强目标类的功能。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是一种编程范式，隶属于软工范畴，指导开发者如何组织程序结构
- AOP最早由AOP联盟的组织提出的,制定了一套规范.Spring将AOP思想引入到框架中,必须遵守AOP联盟的规范
- 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可以对业务代码中【业务逻辑】和【系统逻辑】进行隔离，从而使得【业务逻辑】和【系统逻辑】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。

为什么使用AOP

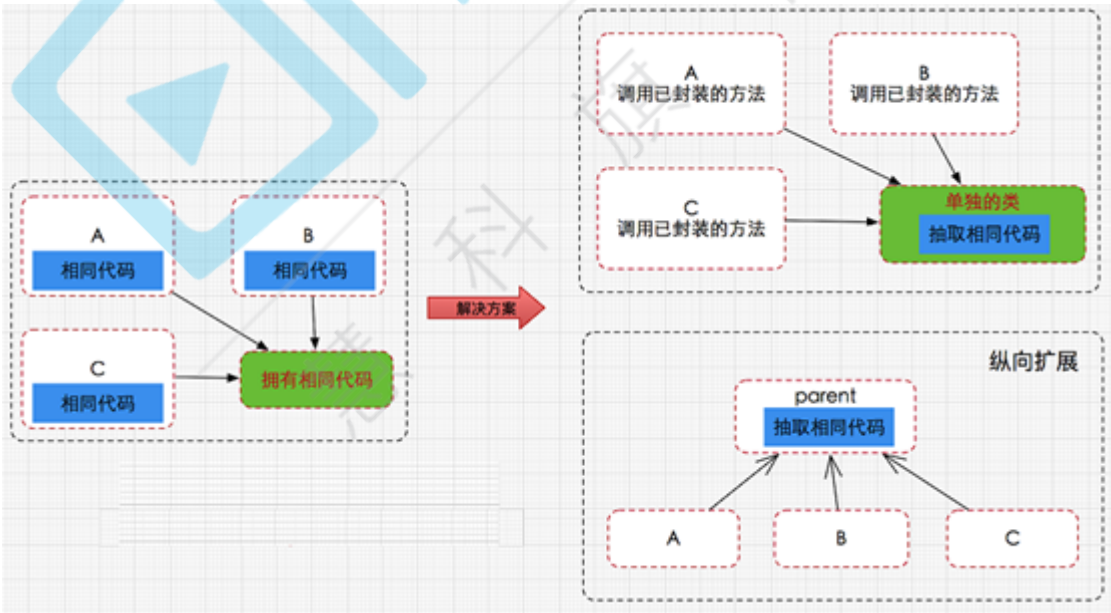
- 作用：

  AOP采取横向抽取机制，补充了传统纵向继承体系（OOP）无法解决的重复性代码优化（性能监视、事务管理、安全检查、缓存），将业务逻辑和系统处理的代码（关闭连接、事务管理、操作日志记录）解耦。

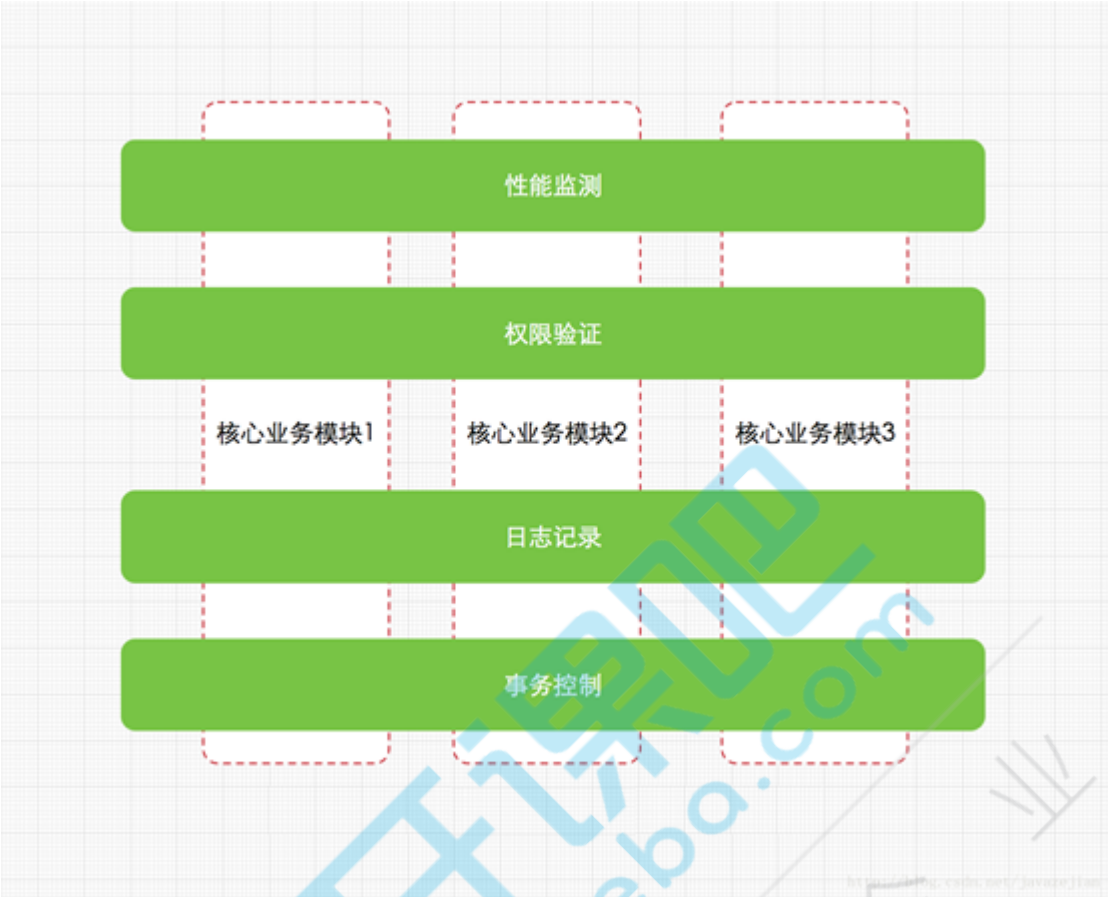
- 优势：

  重复性代码被抽取出来之后，维护更加方便

- 纵向继承体系：



- 横向抽取机制：



AOP相关术语介绍

术语解释

- Joinpoint(连接点)

```
所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点
```

- Pointcut(切入点)

```
所谓切入点是指我们要对哪些Joinpoint进行拦截的定义
```

- Advice(通知/增强)

```
所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)
```

- Introduction(引介)

```
引介是一种特殊的通知在不修改类代码的前提下， Introduction可以在运行期为类动态地添加一些方法或Field
```

- Target(目标对象)

```
1 代理的目标对象
```

- Weaving(织入)

```
1 是指把增强应用到目标对象来创建新的代理对象的过程
```

- Proxy（代理）

```
1 一个类被AOP织入增强后，就产生一个结果代理类
```

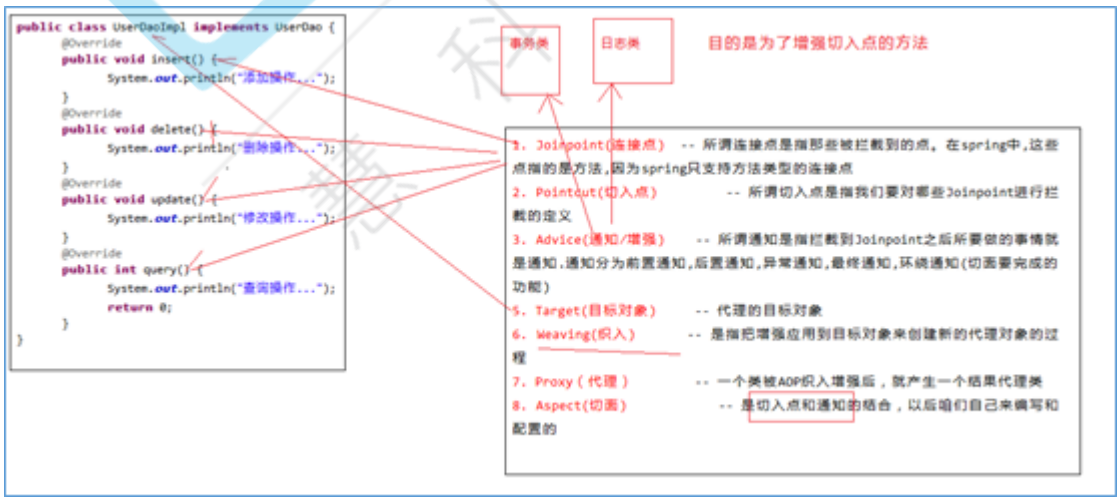
- Aspect(切面)

```
1 是切入点和通知的结合，以后咱们自己来编写和配置的
```

- Advisor（通知器、顾问）

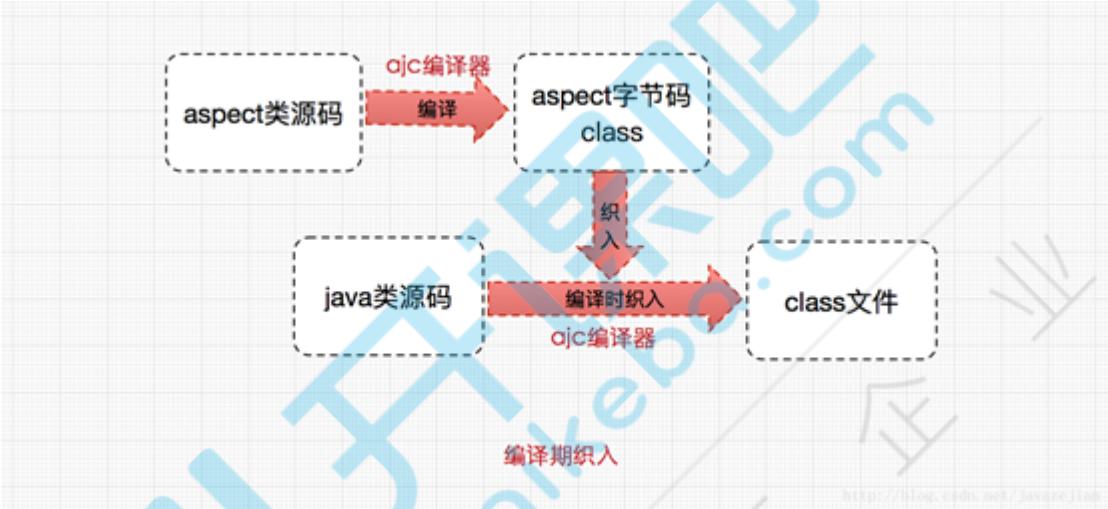
```
1 和Aspect很相似
```

图示说明



AOP实现之AspectJ（了解）

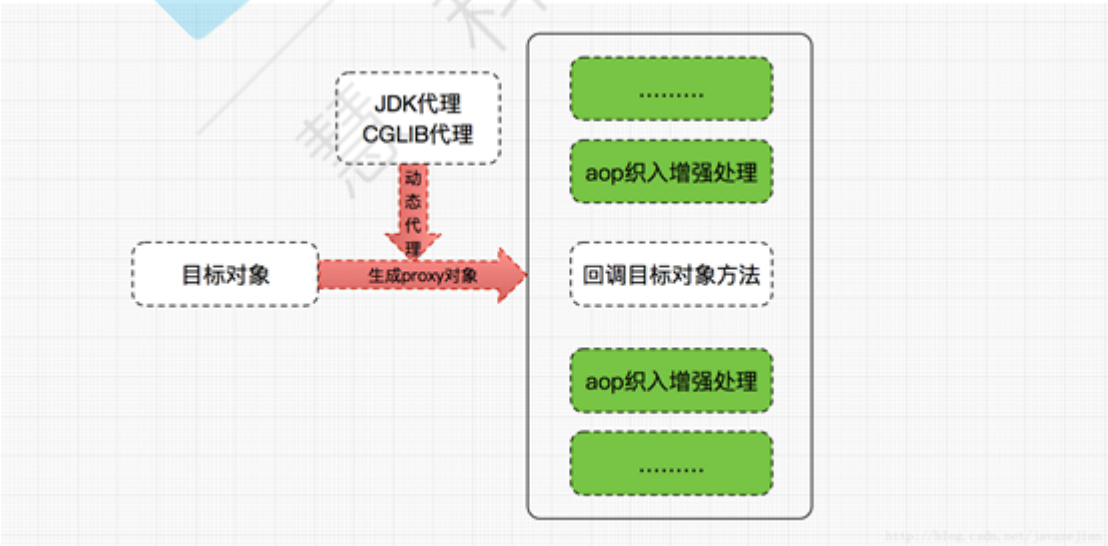
- AspectJ是一个Java实现的AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器）
- 可以这样说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言。更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联，因此对于有java编程基础的工程师，上手和使用都非常容易。
- 了解AspectJ应用到java代码的过程（这个过程称为织入），对于织入这个概念，可以简单理解为aspect(切面)应用到目标函数(类)的过程。
- 对于织入这个过程，一般分为动态织入和静态织入，动态织入的方式是在运行时动态将要增强的代码织入到目标类中，这样往往是通过动态代理技术完成的，如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)，Spring AOP采用的就是基于运行时增强的代理技术
- ApectJ采用的就是静态织入的方式。ApectJ主要采用的是编译期织入，在这个期间使用AspectJ的acj编译器(类似javac)把aspect类编译成class字节码后，在java目标类编译时织入，即先编译aspect类再编译目标类。



AOP实现之Spring AOP(了解)

实现原理分析

- Spring AOP是通过动态代理技术实现的
- 而动态代理是基于反射设计的。（关于反射的知识，请自行学习）
- 动态代理技术的实现方式有两种：基于接口的JDK动态代理和基于继承的CGLib动态代理。



JDK动态代理

目标对象必须实现接口

```
1. 使用Proxy类来生成代理对象的一些代码如下：
    /**
     * 使用JDK的方式生成代理对象
     * @author Administrator
     */
    public class MyProxyUtils {
        public static UserService getProxy(final UserService service) {
            // 使用Proxy类生成代理对象
            UserService proxy =
                 (UserService) Proxy.newProxyInstance(
                    service.getClass().getClassLoader(),
                    service.getClass().getInterfaces(),
                    new InvocationHandler() {

                        // 代理对象方法一执行，invoke方法就会执行一次
                        public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                            if("save".equals(method.getName())){
                                System.out.println("记录日志...");
                                // 开启事务
                            }
                            // 提交事务
                            // 让service类的save或者update方法正常的执行下去
                            return method.invoke(service, args);
                        }
                    });
            // 返回代理对象
            return proxy;
        }
    }
```

Cglib动态代理

- 目标对象不需要实现接口
- 底层是通过继承目标对象产生代理子对象（代理子对象中继承了目标对象的方法，并可以对该方法进行增强）

```
2. 编写相关的代码
    public static UserService getProxy(){
        // 创建CGLIB核心的类
        Enhancer enhancer = new Enhancer();
        // 设置父类
        enhancer.setSuperclass(UserServiceImpl.class);
        // 设置回调函数
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object obj, Method method, Object[] args,
                    MethodProxy methodProxy) throws Throwable {
                if("save".equals(method.getName())){
                    // 记录日志
```

```
                    System.out.println("记录日志了...");
                }
                return methodProxy.invokeSuper(obj, args);
            }
        });
        // 生成代理对象
        UserService proxy = (UserService) enhancer.create();
        return proxy;
    }

```

使用

- 其使用ProxyFactoryBean创建:
- 使用 <aop:advisor> 定义通知器的方式实现AOP则需要通知类实现Advice接口
- 增强（通知）的类型有：

```
- 前置通知：org.springframework.aop.MethodBeforeAdvice

- 后置通知：org.springframework.aop.AfterReturningAdvice

- 环绕通知：org.aopalliance.intercept.MethodInterceptor

- 异常通知：org.springframework.aop.ThrowsAdvice
```

基于AspectJ的AOP使用

其实就是指的Spring + AspectJ整合，不过Spring已经将AspectJ收录到自身的框架中了，并且底层织入依然是采取的动态织入方式。

添加依赖

```
        <!-- 基于AspectJ的aop依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>

```

编写目标类和目标方法

- 编写接口和实现类（目标对象）

```
UserService接口

UserServiceImpl实现类
```

- 配置目标类，将目标类交给spring IoC容器管理

```
<context:component-scan base-package="sourcecode.ioc" />
```

使用XML实现

实现步骤

- 编写通知（增强类，一个普通的类）

```
public class MyAdvice {

    public void log(){
        System.out.println("记录日志...");
    }
}
```

- 配置通知，将通知类交给spring IoC容器管理

```
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>
```

- 配置AOP 切面

```
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>

<!-- AOP配置 -->
<aop:config>
    <aop:aspect ref="myAdvice">
        <!-- method：指定要增强的方法，也就是指定通知类中的增强功能方法 -->
        <!-- pointcut：指定切入点，需要通过表达式来指定 -->
        <aop:before method="log"
            pointcut="execution(void cn.spring.dao.UserDaoImpl.insert())" />
    </aop:aspect>
</aop:config>
```

切入点表达式

- 切入点表达式的格式：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- 表达式格式说明：
  - execution：必须要
  - 修饰符：可省略
  - 返回值类型：**必须要**，但是可以使用*通配符
  - 包名 ：

```
** 多级包之间使用.分割

** 包名可以使用*代替，多级包名可以使用多个*代替

** 如果想省略中间的包名可以使用 ..
```

  - 类名

```
** 可以使用*代替

** 也可以写成*DaoImpl
```

  - 方法名：

```
** 也可以使用*好代替

** 也可以写成add*
```

  - 参数：

```
** 参数使用*代替

** 如果有多个参数，可以使用 ..代替
```

通知类型

通知类型（五种）：前置通知、后置通知、最终通知、环绕通知、异常抛出通知。

- 前置通知：

```
    * 执行时机：目标对象方法之前执行通知
    * 配置文件：<aop:before method="before" pointcut-ref="myPointcut"/>
    * 应用场景：方法开始时可以进行校验
```

- 后置通知：

```
    * 执行时机：目标对象方法之后执行通知，有异常则不执行了
    * 配置文件：<aop:after-returning method="afterReturning" pointcut-ref="myPointcut"/>
    * 应用场景：可以修改方法的返回值
```

- 最终通知：

```
    * 执行时机：目标对象方法之后执行通知，有没有异常都会执行
    * 配置文件：<aop:after method="after" pointcut-ref="myPointcut"/>
    * 应用场景：例如像释放资源
```

- 环绕通知：

```
    * 执行时机：目标对象方法之前和之后都会执行。
    * 配置文件：<aop:around method="around" pointcut-ref="myPointcut"/>
    * 应用场景：事务、统计代码执行时机
```

- 异常抛出通知：

```
    * 执行时机：在抛出异常后通知
    * 配置文件：<aop:after-throwing method=" afterThrowing " pointcut-ref="myPointcut"/>
    * 应用场景：包装异常
```

使用注解实现

实现步骤

- 编写切面类（注意不是通知类，因为该类中可以指定切入点）

```java
/**
 * 切面类（通知+切入点）
 *
 * @author think
 *
 */
// @Aspect：标记该类是一个切面类
@Component("myAspect")
@Aspect
public class MyAspect {

    //@Before：标记该方法是一个前置通知
    //value：切入点表达式
    @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    public void log() {
        System.out.println("记录日志...");
    }
}
```

- 配置切面类

```xml
<context:component-scan base-package="com.kkb.spring"/>
```

- 开启AOP自动代理

```xml
<!-- AOP基于注解的配置-开启自动代理 -->
<aop:aspectj-autoproxy />
```

环绕通知注解配置

@Around

```
作用：
    把当前方法看成是环绕通知。属性：
value：
    用于指定切入点表达式，还可以指定切入点表达式的引用。
```

```java
@Around(value = "execution(* *.*(..))")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) {
    // 定义返回值
    Object rtValue = null;
    try {
        // 获取方法执行所需的参数
        Object[] args = joinPoint.getArgs();
        // 前置通知：开启事务beginTransaction();
        // 执行方法
        rtValue = joinPoint.proceed(args);
        // 后置通知：提交事务commit();
    } catch (Throwable e) {
        // 异常通知：回滚事务rollback(); e.printStackTrace();
    } finally {
        // 最终通知：释放资源release();
    }
    return rtValue;

}
```

定义通用切入点

使用@PointCut注解在切面类中定义一个通用的切入点，其他通知可以引用该切入点

```java
// @Aspect：标记该类是一个切面类
@Aspect
public class MyAspect {

    // @Before：标记该方法是一个前置通知
    // value：切入点表达式
    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void log() {
        System.out.println("记录日志...");
    }

    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void validate() {
        System.out.println("进行后台校验...");
    }

    // 通过@Pointcut定义一个通用的切入点
    @Pointcut(value = "execution(* *..*.*DaoImpl.*(..))")
    public void fn() {
    }
}
```

纯注解方式

```java
@Configuration
@ComponentScan(basePackages="com.kkb")
@EnableAspectJAutoProxy
public class SpringConfiguration {
}
```

组件支撑篇

整合Junit

单元测试问题

在测试类中，每个测试方法都有以下两行代码：

```java
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");
UserService service1 = context.getBean(UserService.class);
```

- 我们使用单元测试要测试的是业务问题，以上两段代码明显不是业务代码。
- 但是这两行代码的作用是获取容器，如果不写的话，直接会提示空指针异常，所以又不能轻易删掉。

解决思路分析

针对上述问题，我们需要的是程序能自动帮我们创建容器。一旦程序能自动为我们创建 spring 容器，我们就无须手动创建了，问题也就解决了。

但紧接的问题就是junit它本身不认识spring，更无法帮助创建Spring容器了，不过好在Junit 给我们暴露了一个注解（@RunWith），可以让我们替换掉它的运行器。

这时，我们需要依靠 spring 框架，因为它提供了一个运行器，可以读取配置文件（或注解）来创建容器。我们只需要告诉它配置文件在哪就行了。

具体实现

- 第一步：添加依赖

```
添加spring-test包即可
```

- 第二步：通过@RunWith注解，指定spring的运行器

```
Spring的运行器是SpringJunit4ClassRunner
```

- 第三步：通过@ContextConfiguration注解，指定spring运行器需要的配置文件路径
- 第四步：通过@Autowired注解给测试类中的变量注入数据

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:applicationContext.xml")
public class TestSpringJunit {

    @Autowired
    private UserService service;

    @Test
    public void saveUser() {
        service.saveUser();
    }
}
```

事务支持

事务回顾

事务介绍

事务：指的是逻辑上一组操作，组成这个事务的各个执行单元，要么一起成功,要么一起失败！

事务的特性（ACID）：

- 原子性（Atomicity）

> 原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。

- **一致性（Consistency）**

> 一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。
>
> 拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

- **隔离性（Isolation）**

> 隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

- **持久性（Durability）**

> 持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

事务并发问题（隔离性导致）

在事务的**并发操作**中可能会出现一些问题：

- 脏读：一个事务读取到另一个事务**未提交**的数据。
- 不可重复读：一个事务因读取到另一个事务**已提交的数据**。导致*对同一条记录读取*两次以上的结果不一致。**update操作**
- 幻读：一个事务因读取到另一个事务**已提交的数据**。导致*对同一张表读取*两次以上的结果不一致。**insert、delete操作**

事务隔离级别

为了避免上面出现的几种情况，在标准SQL规范中，定义了**4个事务隔离级别**，不同的隔离级别对事务的处理不同

- **四种隔离级别：**

  现在来看看MySQL数据库为我们提供的四种隔离级别（*由低到高*）：

  ① Read uncommitted（读未提交）：最低级别，任何情况都无法保证。

  ② Read committed（读已提交）：可避免脏读的发生。

  ③ Repeatable read（可重复读）：可避免脏读、不可重复读的发生。

  ④ Serializable（串行化）：可避免脏读、不可重复读、幻读的发生。

- **默认隔离级别**

  大多数数据库的默认隔离级别是Read Committed（RC），比如Oracle、DB2等。

  MySQL数据库的默认隔离级别是Repeatable Read（RR）。

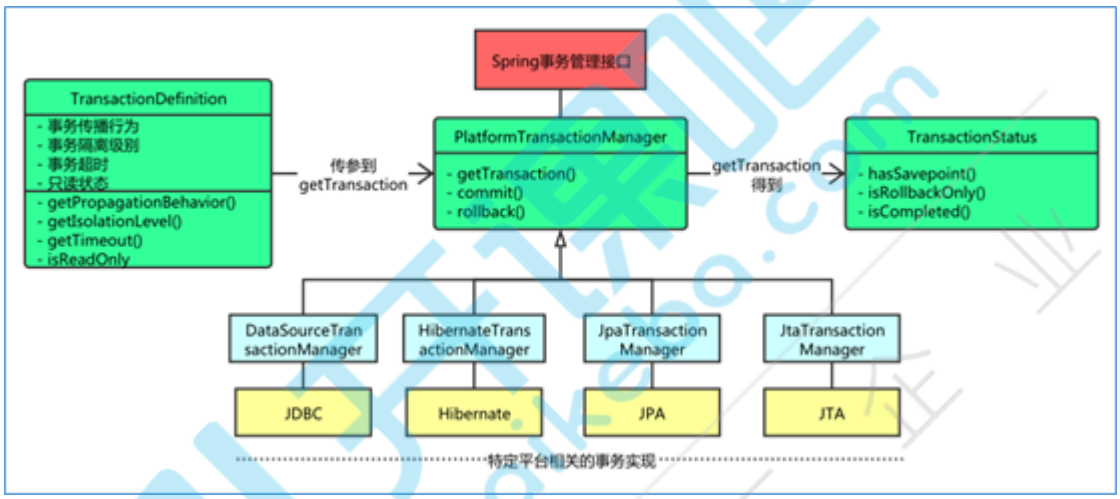
- 注意事项：

```
隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。
```

对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为Read Committed。它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、幻读这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

Spring框架事务管理相关接口

```
Spring并不直接管理事务，而是提供了事务管理接口是PlatformTransactionManager，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。
```



1. PlatformTransactionManager接口 -- 平台事务管理器.(真正管理事务的类)。该接口有具体的实现类，根据不同的持久层框架，需要选择不同的实现类！
2. TransactionDefinition接口 -- 事务定义信息.(事务的隔离级别,传播行为,超时,只读)
3. TransactionStatus接口 -- 事务的状态（是否新事务、是否已提交、是否有保存点、是否回滚）
4. 总结：上述对象之间的关系：平台事务管理器真正管理事务对象.根据事务定义的信息 TransactionDefinition 进行事务管理，在管理事务中产生一些状态.将状态记录到TransactionStatus中
5. PlatformTransactionManager接口中实现类和常用的方法

```
1. 接口的实现类
  * 如果使用的Spring的JDBC模板或者MyBatis（IBatis）框架，需要选择DataSourceTransactionManager实现类
  * 如果使用的是Hibernate的框架，需要选择HibernateTransactionManager实现类

2. 该接口的常用方法
  * void commit(TransactionStatus status)
  * TransactionStatus getTransaction(TransactionDefinition definition)
  * void rollback(TransactionStatus status)

```

6. TransactionDefinition

```
1. 事务隔离级别的常量
    * static int ISOLATION_DEFAULT              -- 采用数据库的默认隔离级别
```

```
    * static int ISOLATION_READ_UNCOMMITTED
    * static int ISOLATION_READ_COMMITTED
    * static int ISOLATION_REPEATABLE_READ
    * static int ISOLATION_SERIALIZABLE

2. 事务的传播行为常量（不用设置，使用默认值）
    * 先解释什么是事务的传播行为：解决的是业务层之间的方法调用！！
    * PROPAGATION_REQUIRED（默认值） -- A中有事务,使用A中的事务.如果没有，B就会开启一个新的事务,将A包含进来.(保证A,B在同一个事务中)，默认值！！
    * PROPAGATION_SUPPORTS           -- A中有事务,使用A中的事务.如果A中没有事务.那么B也不使用事务.
    * PROPAGATION_MANDATORY          -- A中有事务,使用A中的事务.如果A没有事务.抛出异常.

    * PROPAGATION_REQUIRES_NEW      -- A中有事务,将A中的事务挂起.B创建一个新的事务.(保证A,B没有在一个事务中)
    * PROPAGATION_NOT_SUPPORTED     -- A中有事务,将A中的事务挂起.
    * PROPAGATION_NEVER             -- A中有事务,抛出异常.

    * PROPAGATION_NESTED            -- 嵌套事务.当A执行之后,就会在这个位置设置一个保存点.如果B没有问题.执行通过.如果B出现异常,运行客户根据需求回滚(选择回滚到保存点或者是最初始状态)

```

Spring框架事务管理的分类

1. Spring的编程式事务管理（不推荐使用）

```
通过手动编写代码的方式完成事务的管理（不推荐）
```

2. Spring的声明式事务管理（底层采用AOP的技术）

```
通过一段配置的方式完成事务的管理
```

编程式事务管理（了解）

说明：Spring为了简化事务管理的代码:提供了模板类 TransactionTemplate，所以手动编程的方式来管理事务，只需要使用该模板类即可！！

手动编程方式的具体步骤如下：

1. 步骤一:配置一个事务管理器，Spring使用PlatformTransactionManager接口来管理事务，所以咱们需要使用到他的实现类！！

```xml
<!-- 配置事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

```

2. 步骤二:配置事务管理的模板

```xml
<!-- 配置事务管理的模板 -->
<bean id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
     <property name="transactionManager" ref="transactionManager"/>
</bean>
```

3. 步骤三:在需要进行事务管理的类中,注入事务管理的模板

```xml
        <bean id="accountService" class="com.kkb.spring.service.AccountServiceImpl">
            <property name="accountDao" ref="accountDao"/>
            <property name="transactionTemplate" ref="transactionTemplate"/>
        </bean>

```

4. 步骤四:在业务层使用模板管理事务:

```java
        // 注入事务模板对象
        private TransactionTemplate transactionTemplate;
        public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
            this.transactionTemplate = transactionTemplate;
        }

        public void pay(final String out, final String in, final double money) {
            transactionTemplate.execute(new TransactionCallbackWithoutResult() {

                protected void doInTransactionWithoutResult(TransactionStatus status)
{
                    // 扣钱
                    accountDao.outMoney(out, money);
                    int a = 10/0;
                    // 加钱
                    accountDao.inMoney(in, money);
                }
            });
        }
```

声明式事务管理（重点）

声明式事务管理又分成三种种方式

- 基于AspectJ的XML方式（重点掌握）
- 基于AspectJ的注解+XML混用方式（重点掌握）
- 基于AspectJ的纯注解方式（重点掌握）

事务管理之XML方式

准备转账环境：

业务层

```
AccountService
AccountServiceImpl
```

```
public void transfer(String in, String out, double money) {
    dao.outMoney(out, money);

    //异常代码
    System.out.println(1/0);

    dao.inMoney(in, money);
}
```

持久层

```
AccountDao
AccountDaoImpl
```

spring配置

```
<!-- 配置数据源连接池(C3P0) -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql:///web01" />
    <property name="user" value="root" />
    <property name="password" value="root" />
</bean>

<bean class="cn.spring.dao.AccountDaoImpl">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<context:component-scan base-package="cn.spring.service"></context:component-scan>
```

单元测试代码

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext-tx.xml" })
public class TransactionTest {

    @Autowired
    private AccountService service;

    @Test
    public void test01() {
        service.transfer("李聪", "陈阳", 100);
    }

}
```

配置事务管理的AOP

- 平台事务管理器：DataSourceTransactionManager

```xml
<!-- 配置平台事务管理器 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

- 事务通知：

```
<tx:advice id="" transaction-manager=""/>
```

```xml
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 配置事务相关属性 -->
    <tx:attributes>
        <!--对方法级别设置 事务的隔离级别、事务传播行为 -->
        <!-- 设置了默认的隔离级别和传播行为 -->
        <tx:method name="transfer*" />
        <!-- 提交订单 -->
        <tx:method name="submitOrder"/>
        <!-- 添加 -->
        <tx:method name="add*" />
        <tx:method name="insert*" />
        <!-- 删除 -->
        <tx:method name="delete*" />
        <tx:method name="remove*" />
        <!-- 修改 -->
        <tx:method name="update*" />
        <tx:method name="modify*" />
        <!-- 查询 -->
        <tx:method name="find*" read-only="true" />
        <tx:method name="get*" read-only="true" />
        <tx:method name="query*" read-only="true" />
    </tx:attributes>
</tx:advice>
```

- AOP配置：

```
        <aop:config>
            <aop:advisor advice-ref="" pointcut=""/>
        </aop:config>
```

```
<!-- 配置AOP -->
<aop:config>
    <!-- 切入点：所有的业务层实现类中方法 -->
    <aop:advisor advice-ref="txAdvice" pointcut="execution(* cn..service.*.*(..))"/>
</aop:config>
```

事务管理之混合方式

- service类上或者方法上加注解：

```
类上加@Transactional：表示该类中所有的方法都被事务管理
方法上加@Transactional：表示只有该方法被事务管理
```

- 开启事务注解：

```
<!-- 配置平台事务管理器 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 开启事务注解 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

事务管理之基于AspectJ的纯注解方式

```
@EnableTransactionManagement
```