Spring 框架

尚硅谷 java 研究院

版本：V 1.0

第0章　简单了解框架

框架，即 framework。其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

框架是对特定应用领域中的应用系统的部分设计和实现的整体结构。

因为软件系统发展到今天已经很复杂了，特别是服务器端软件，涉及到的知识，内容，问题太多。在某些方面使用别人成熟的框架，就相当于让别人帮你完成一些基础工作，你只需要集中精力完成系统的业务逻辑设计。而且框架一般是成熟，稳健的，他可以处理系统很多细节问题，比如，事务处理，安全性，数据流控制等问题。还有框架一般都经过很多人使用，所以结构很好，所以扩展性也很好，而且它是不断升级的，你可以直接享受别人升级代码带来的好处。

第1章　Spring 概述

1.1 Spring 概述

1) Spring 是一个开源框架

2) Spring 为简化企业级开发而生，使用 Spring，JavaBean 就可以实现很多以前要靠 EJB 才能实现的功能。同样的功能，在 EJB 中要通过繁琐的配置和复杂的代码才能够实现，而在 Spring 中却非常的优雅和简洁。

3) Spring 是一个 **IOC**(DI)和 **AOP** 容器框架。

4) Spring 的优良特性

① **非侵入式**：基于 Spring 开发的应用中的对象可以不依赖于 Spring 的 API

② **依赖注入**：DI——Dependency Injection，反转控制(IOC)最经典的实现。

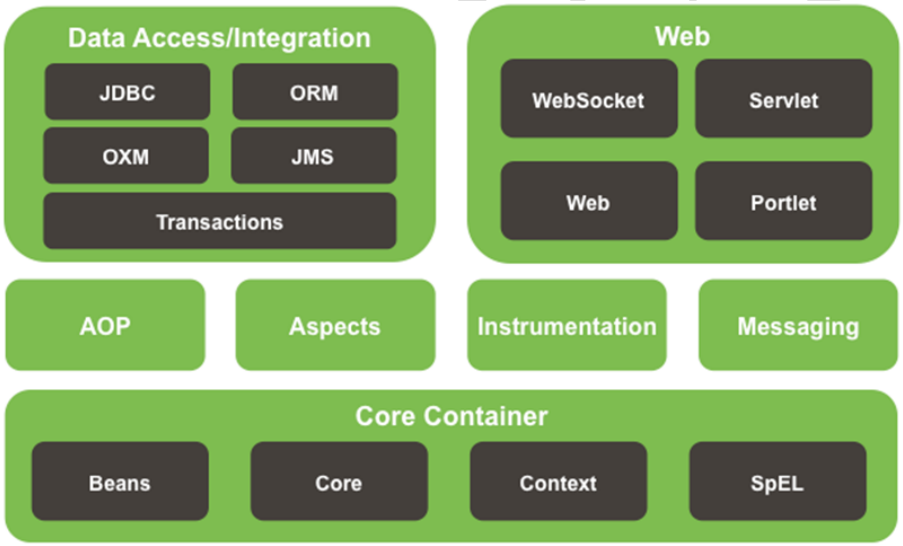
③ **面向切面编程**：Aspect Oriented Programming——AOP

④ **容器**：Spring 是一个容器，因为它包含并且管理应用对象的生命周期

⑤ **组件化**：Spring 实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用 XML 和 Java 注解组合这些对象。

⑥ **一站式**：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上 Spring 自身也提供了表述层的 SpringMVC 和持久层的 Spring JDBC）。

5) Spring 模块



1.2 搭建 Spring 运行时环境

1) 加入 JAR 包

① Spring 自身 JAR 包：spring-framework-4.0.0.RELEASE\libs 目录下

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELE2ASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

② commons-logging-1.1.1.jar

2) 在 Spring Tool Suite 工具中通过如下步骤创建 Spring 的配置文件

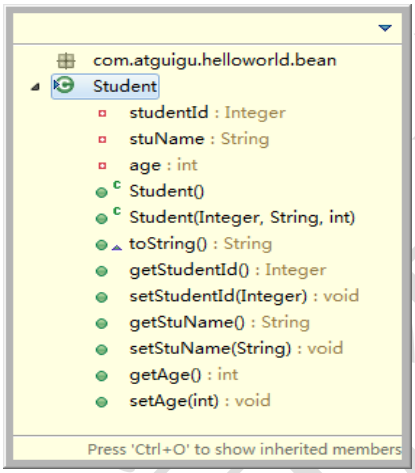
① File->New->Spring Bean Configuration File

② 为文件取名字 例如：applicationContext.xml

1.3 HelloWorld

1) 目标：使用 Spring 创建对象，为属性赋值

2) 创建 Student 类



3) 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>
```

4) 测试：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);
```

第 2 章 IOC 容器和 Bean 的配置

2.1 IOC 和 DI

2.1.1 IOC(Inversion of Control)：反转控制

在应用程序中的组件需要获取资源时，传统的方式是组件主动的从容器中获取所需要的资源，在这样的模式下开发人员往往需要知道在具体容器中特定资源的获取方式，增加了学习成本，同时降低了开发效率。

反转控制的思想完全颠覆了应用程序组件获取资源的传统方式：反转了资源的获取方向——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

反转控制: 我想吃饭 饭来张口

2.1.2 DI(Dependency Injection)：依赖注入

IOC 的另一种表述方式：即组件以一些预先定义好的方式(例如：setter 方法)接受来自于容器的资源注入。相对于 IOC 而言，这种表述更直接。

总结: IOC 就是一种反转控制的思想， 而 DI 是对 IOC 的一种具体实现。

2.1.3 IOC 容器在 Spring 中的实现

前提: Spring 中有 IOC 思想， 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.

1）在通过 IOC 容器读取 Bean 的实例之前，需要先将 IOC 容器本身实例化。

2）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。

2.1.4 ApplicationContext 的主要实现类

1) ClassPathXmlApplicationContext：对应类路径下的 XML 格式的配置文件
2) FileSystemXmlApplicationContext：对应文件系统中的 XML 格式的配置文件
3) 在初始化时就创建单例的 bean，也可以通过配置的方式指定创建的 Bean 是多实例的。

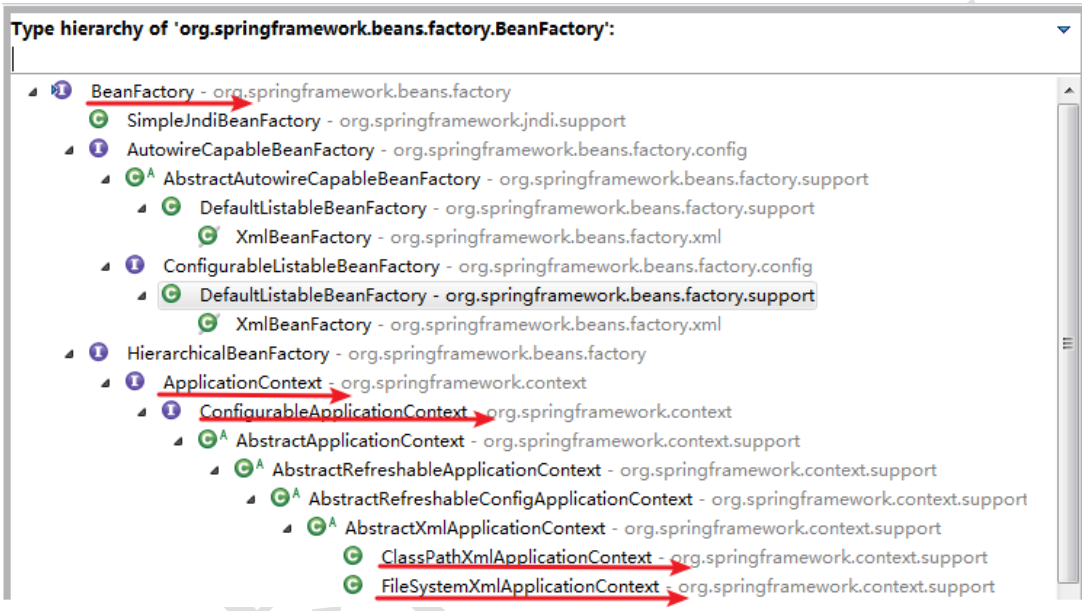
2.1.5 ConfigurableApplicationContext

1) 是 ApplicationContext 的子接口，包含一些扩展方法
2) refresh()和 close()让 ApplicationContext 具有启动、关闭和刷新上下文的能力。

2.1.6 WebApplicationContext

1) 专门为 WEB 应用而准备的，它允许从相对于 WEB 根目录的路径中完成初始化工作

2.1.7 容器的结构图



2.2 通过类型获取 bean

1) 从 IOC 容器中获取 bean 时，除了通过 id 值获取，还可以通过 bean 的类型获取。但如果同一个类型的 bean 在 XML 文件中配置了多个，则获取时会抛出异常，所以同一个类型的 bean 在容器中必须是唯一的。

```
HelloWorld helloWorld = cxt.getBean(HelloWorld. class);
```

2) 或者可以使用另外一个重载的方法，同时指定 bean 的 id 值和类型

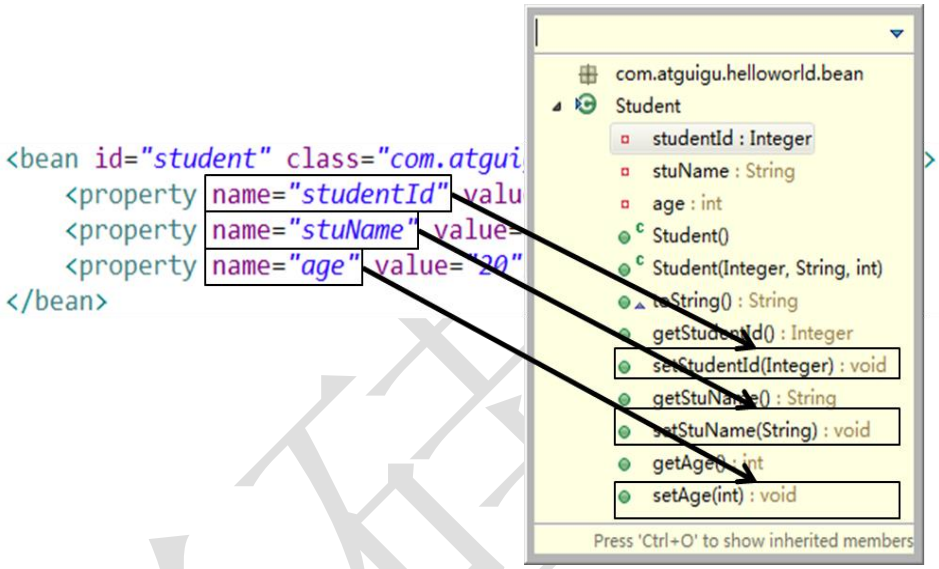
```
HelloWorld helloWorld = cxt.getBean("helloWorld",HelloWorld. class);
```

2.3 给 bean 的属性赋值

2.3.1 依赖注入的方式

1. 通过 bean 的 setXxx()方法赋值

Hello World 中使用的就是这种方式



2. 通过 bean 的构造器赋值

1) Spring 自动匹配合适的构造器

```
<bean id="book" class="com.atguigu.spring.bean.Book" >
  <constructor-arg value= "10010"/>
  <constructor-arg value= "Book01"/>
  <constructor-arg value= "Author01"/>
  <constructor-arg value= "20.2"/>
</bean >
```

2) 通过索引值指定参数位置

```
<bean id="book" class="com.atguigu.spring.bean.Book" >
```

```
        <constructor-arg value= "10010" index ="0"/>
        <constructor-arg value= "Book01" index ="1"/>
        <constructor-arg value= "Author01" index ="2"/>
        <constructor-arg value= "20.2" index ="3"/>
    </bean >
```

3) 通过类型区分重载的构造器

```
<bean id="book" class="com.atguigu.spring.bean.Book" >
    <constructor-arg value= "10010" index ="0" type="java.lang.Integer" />
    <constructor-arg value= "Book01" index ="1" type="java.lang.String" />
    <constructor-arg value= "Author01" index ="2" type="java.lang.String" />
    <constructor-arg value= "20.2" index ="3" type="java.lang.Double" />
</bean >
```

2.3.2 p 名称空间

为了简化 XML 文件的配置，越来越多的 XML 文件采用属性而非子元素配置信息。Spring 从 2.5 版本开始引入了一个新的 p 命名空间，可以通过<bean>元素属性的方式配置 Bean 的属性。

使用 p 命名空间后，基于 XML 的配置方式将进一步简化。

```
<bean
    id="studentSuper"
    class="com.atguigu.helloworld.bean.Student"
    p:studentId="2002" p:stuName="Jerry2016" p:age="18" />
```

2.3.3 可以使用的值

1. 字面量

1) 可以使用字符串表示的值，可以通过 value 属性或 value 子节点的方式指定

2) 基本数据类型及其封装类、String 等类型都可以采取字面值注入的方式

3) 若字面值中包含特殊字符，可以使用<![CDATA[]]>把字面值包裹起来

2. null 值

```
<bean class="com.atguigu.spring.bean.Book" id="bookNull" >
  <property name= "bookId" value ="2000"/>
  <property name= "bookName">
    <null/>
  </property>
  <property name= "author" value ="nullAuthor"/>
  <property name= "price" value ="50"/>
</bean >
```

3. 给 bean 的级联属性赋值

```
<bean id="action" class="com.atguigu.spring.ref.Action">
        <property name="service" ref="service"/>
        <!-- 设置级联属性(了解) -->
        <property name="service.dao.dataSource" value="DBCP"/>
</bean>
```

4. 外部已声明的 bean、引用其他的 bean

```
<bean id="shop" class="com.atguigu.spring.bean.Shop" >
  <property name= "book" ref ="book"/>
</bean >
```

5. 内部 bean

当 bean 实例仅仅给一个特定的属性使用时，可以将其声明为内部 bean。内部 bean 声明直接包含在<property>或<constructor-arg>元素里，不需要设置任何 id 或 name 属性

内部 bean 不能使用在任何其他地方

```
<bean id="shop2" class="com.atguigu.spring.bean.Shop" >
  <property name= "book">
    <bean class= "com.atguigu.spring.bean.Book" >
      <property name= "bookId" value ="1000"/>
      <property name= "bookName" value="innerBook" />
      <property name= "author" value="innerAuthor" />
      <property name= "price" value ="50"/>
    </bean>
  </property>
</bean >
```

2.4 集合属性

在 Spring 中可以通过一组内置的 XML 标签来配置集合属性，例如：<list>，<set>或<map>。

2.4.1 数组和 List

配置 java.util.List 类型的属性，需要指定<list>标签，在标签里包含一些元素。这些标签可以通过<value>指定简单的常量值，通过<ref>指定对其他 Bean 的引用。通过<bean>指定内置 bean 定义。通过<null/>指定空元素。甚至可以内嵌其他集合。

数组的定义和 List 一样，都使用<list>元素。

配置 java.util.Set 需要使用<set>标签，定义的方法与 List 一样。

```
<bean id="shop" class="com.atguigu.spring.bean.Shop" >
  <property name= "categoryList">
    <!-- 以字面量为值的 List 集合 -->
    <list>
      <value> 历史</value >
      <value> 军事</value >
    </list>
  </property>
  <property name= "bookList">
    <!-- 以 bean 的引用为值的 List 集合 -->
    <list>
      <ref bean= "book01"/>
      <ref bean= "book02"/>
    </list>
  </property>
</bean >
```

2.4.2 Map

Java.util.Map 通过<map>标签定义，<map>标签里可以使用多个<entry>作为子标签。每个条目包含一个键和一个值。

必须在<key>标签里定义键。

因为键和值的类型没有限制，所以可以自由地为它们指定<value>、<ref>、<bean>或<null/>元素。

可以将 Map 的键和值作为<entry>的属性定义：简单常量使用 key 和 value 来定义；bean 引用通过 key-ref 和 value-ref 属性定义。

```
<bean id="cup" class="com.atguigu.spring.bean.Cup">
    <property name="bookMap">
        <map>
            <entry>
                <key>
                    <value>bookKey01</value>
                </key>
                <ref bean="book01"/>
            </entry>
            <entry>
                <key>
                    <value>bookKey02</value>
                </key>
                <ref bean="book02"/>
            </entry>
        </map>
    </property>
</bean>
```

2.4.3 集合类型的 bean

如果只能将集合对象配置在某个 bean 内部，则这个集合的配置将不能重用。我们需要将集合 bean 的配置拿到外面，供其他 bean 引用。

配置集合类型的 bean 需要引入 util 名称空间

```
<util:list id="bookList">
    <ref bean="book01"/>
    <ref bean="book02"/>
    <ref bean="book03"/>
    <ref bean="book04"/>
    <ref bean="book05"/>
</util:list>

<util:list id="categoryList">
    <value>编程</value>
    <value>极客</value>
    <value>相声</value>
    <value>评书</value>
</util:list>
```

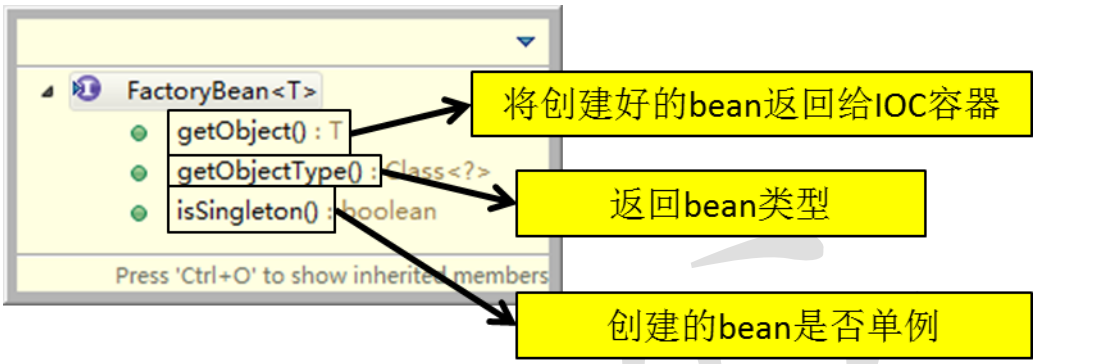
2.5 FactoryBean

2.5.1 FactoryBean

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，其返回的对象不是指定类的一个实例，其返回的是该工厂 bean 的 getObject 方法所返回的对象。

工厂 bean 必须实现 org.springframework.beans.factory.FactoryBean 接口。




```
<bean id="product" class="com.atguigu.spring.bean.ProductFactory">
    <property name="productName" value="Mp3" />
</bean>
```

2.6 bean 的作用域

在 Spring 中，可以在<bean>元素的 scope 属性里设置 bean 的作用域，以决定这个 bean 是单实例的还是多实例的。

默认情况下，Spring 只为每个在 IOC 容器里声明的 bean 创建唯一一个实例，整个 IOC 容器范围内都能共享该实例：所有后续的 getBean()调用和 bean 引用都将返回这个唯一的 bean 实例。该作用域被称为 singleton，它是所有 bean 的默认作用域。

类别	说明
singleton	在 SpringIOC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在
prototype	每次调用 getBean() 时都会返回一个新的实例
request	每次 HTTP 请求都会创建一个新的 Bean，该作用域仅适用于 WebApplicationContext 环境
session	同一个 HTTP Session 共享一个 Bean，不同的 HTTP Session 使用不同的 Bean。该作用域仅适用于 WebApplicationContext 环境

当 bean 的作用域为单例时，Spring 会在 IOC 容器对象创建时就创建 bean 的对象实例。

而当 bean 的作用域为 prototype 时，IOC 容器在获取 bean 的实例时创建 bean 的实例对象。

2.7 bean 的生命周期

1) Spring IOC 容器可以管理 bean 的生命周期，Spring 允许在 bean 生命周期内特定的时间点执行指定的任务。

2) Spring IOC 容器对 bean 的生命周期进行管理的过程：

① 通过构造器或工厂方法创建 bean 实例

② 为 bean 的属性设置值和对其他 bean 的引用

③ 调用 bean 的初始化方法

④ bean 可以使用了

⑤ 当容器关闭时，调用 bean 的销毁方法

3) 在配置 bean 时，通过 init-method 和 destroy-method 属性为 bean 指定初始化和销毁方法

4) bean 的后置处理器

① bean 后置处理器允许在调用**初始化方法前后**对 bean 进行额外的处理

② bean 后置处理器对 IOC 容器里的所有 bean 实例逐一处理，而非单一实例。

其典型应用是：检查 bean 属性的正确性或根据特定的标准更改 bean 的属性。

③ bean 后置处理器需要实现接口：

org.springframework.beans.factory.config.BeanPostProcessor。在初始化方法被调用前后，Spring 将把每个 bean 实例分别传递给上述接口的以下两个方法：

●postProcessBeforeInitialization(Object, String)

●postProcessAfterInitialization(Object, String)

5) 添加 bean 后置处理器后 bean 的生命周期

①通过构造器或工厂方法**创建 bean 实例**

②为 bean 的**属性设置值**和对其他 bean 的引用

③将 bean 实例传递给 bean 后置处理器的 postProcessBeforeInitialization()方法

④调用 bean 的**初始化**方法

⑤将 bean 实例传递给 bean 后置处理器的 postProcessAfterInitialization()方法

⑥bean 可以使用了

⑦当容器关闭时调用 bean 的销毁方法

2.8 引用外部属性文件

当 bean 的配置信息逐渐增多时，查找和修改一些 bean 的配置信息就变得愈加困难。这时可以将一部分信息提取到 bean 配置文件的外部，以 properties 格式的属性文件保存起来，同时在 bean 的配置文件中引用 properties 属性文件中的内容，从而实现一部分属性值在发生变化时仅修改 properties 属性文件即可。这种技术多用于连接数据库的基本信息的配置。

2.8.1 直接配置

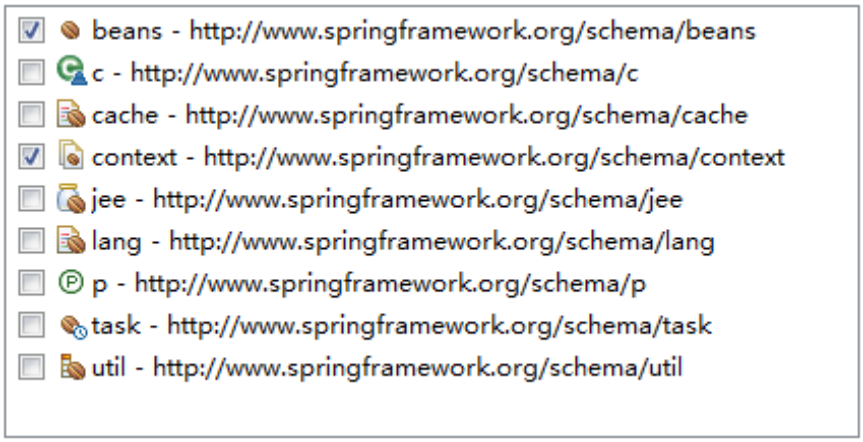
```
<!-- 直接配置 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="user" value="root"/>
    <property name="password" value="root"/>
    <property name="jdbcUrl" value="jdbc:mysql:///test"/>
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
</bean>
```

2.8.2 使用外部的属性文件

1. 创建 properties 属性文件

```
prop.userName=root
prop.password=root
prop.url=jdbc:mysql:///test
prop.driverClass=com.mysql.jdbc.Driver
```

2. 引入 context 名称空间



3.指定 properties 属性文件的位置

```
<!-- 指定properties属性文件的位置 -->
<!-- classpath:xxx 表示属性文件位于类路径下 -->
<context:property-placeholder location="classpath:jdbc.properties"/>
```

4.从 properties 属性文件中引入属性值

```
<!-- 从properties属性文件中引入属性值 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="user" value="${prop.userName}"/>
    <property name="password" value="${prop.password}"/>
    <property name="jdbcUrl" value="${prop.url}"/>
    <property name="driverClass" value="${prop.driverClass}"/>
</bean>
```

2.9 自动装配

2.9.1 自动装配的概念

1) 手动装配：以 value 或 ref 的方式明确指定属性值都是手动装配。
2) 自动装配：根据指定的装配规则，不需要明确指定，Spring 自动将匹配的属性值注入 bean 中。

2.9.2 装配模式

1) 根据类型自动装配：将类型匹配的 bean 作为属性注入到另一个 bean 中。若 IOC 容器中有多个与目标 bean 类型一致的 bean，Spring 将无法判定哪个 bean 最合适该属性，所以不能执行自动装配
2) 根据名称自动装配：必须将目标 bean 的名称和属性名设置的完全相同
3) 通过构造器自动装配：当 bean 中存在多个构造器时，此种自动装配方式将会很复杂。不推荐使用。

2.9.3 选用建议

相对于使用注解的方式实现的自动装配，在 XML 文档中进行的自动装配略显笨拙，在项目中更多的使用注解的方式实现。

2.10 通过注解配置 bean

2.10.1 概述

相对于 XML 方式而言，通过注解的方式配置 bean 更加简洁和优雅，而且和 MVC 组件化开发的理念十分契合，是开发中常用的使用方式。

2.10.2 使用注解标识组件

1) 普通组件：@Component
   标识一个受 Spring IOC 容器管理的组件
2) 持久化层组件：@Repository
   标识一个受 Spring IOC 容器管理的持久化层组件

3) 业务逻辑层组件：@Service

标识一个受 Spring IOC 容器管理的业务逻辑层组件

4) 表述层控制器组件：@Controller

标识一个受 Spring IOC 容器管理的表述层控制器组件

5) 组件命名规则

①默认情况：使用组件的简单类名首字母小写后得到的字符串作为 bean 的 id

②使用组件注解的 value 属性指定 bean 的 id

注意：事实上 Spring 并没有能力识别一个组件到底是不是它所标记的类型，即使将 @Respository 注解用在一个表述层控制器组件上面也不会产生任何错误，所以 @Respository、@Service、@Controller 这几个注解仅仅是为了让开发人员自己明确当前的组件扮演的角色。

2.10.3 扫描组件

组件被上述注解标识后还需要通过 Spring 进行扫描才能够侦测到。

1) 指定被扫描的 package

```
<context:component-scan base-package="com.atguigu.component"/>
```

2) 详细说明

①**base-package** 属性指定一个需要扫描的基类包，Spring 容器将会扫描这个基类包及其子包中的所有类。

②当需要扫描多个包时可以使用逗号分隔。

③如果仅希望扫描特定的类而非基包下的所有类，可使用 resource-pattern 属性过滤特定的类，示例：

```
<context:component-scan
    base-package="com.atguigu.component"
    resource-pattern="autowire/*.class"/>
```

④包含与排除

●<context:include-filter>子节点表示要包含的目标类

注意：通常需要与 use-default-filters 属性配合使用才能够达到“仅包含某些组件”这样的效果。即：通过将 use-default-filters 属性设置为 false，禁用默认过滤器，然后扫描的就只是 include-filter 中的规则指定的组件了。

●<context:exclude-filter>子节点表示要排除在外的目标类
●component-scan 下可以拥有若干个 include-filter 和 exclude-filter 子节点
●过滤表达式

类别	示例	说明
annotation	com.atguigu.XxxAnnotation	过滤所有标注了 XxxAnnotation 的类。这个规则根据目标组件是否标注了指定类型的注解进行过滤。
assignable	com.atguigu.BaseXxx	过滤所有 BaseXxx 类的子类。这个规则根据目标组件是否是指定类型的子类的方式进行过滤。
aspectj	com.atguigu.*Service+	所有类名是以 Service 结束的，或这样的类的子类。这个规则根据 AspectJ 表达式进行过滤。
regex	com\.atguigu\.anno\.*	所有 com.atguigu.anno 包下的类。这个规则根据正则表达式匹配到的类名进行过滤。
custom	com.atguigu.XxxTypeFilter	使用 XxxTypeFilter 类通过编码的方式自定义过滤规则。该类必须实现 org.springframework.core.type.filter.TypeFilter 接口

3) JAR 包

必须在原有 JAR 包组合的基础上再导入一个：spring-aop-4.0.0.RELEASE.jar

2.10.4 组件装配

1) 需求

Controller 组件中往往需要用到 Service 组件的实例，Service 组件中往往需要用到 Repository 组件的实例。Spring 可以通过注解的方式帮我们实现属性的装配。

2) 实现依据

在指定要扫描的包时，<context:component-scan> 元素会自动注册一个 bean 的后置处理器：AutowiredAnnotationBeanPostProcessor 的实例。该后置处理器可以自动装配标记了**@Autowired**、@Resource 或@Inject 注解的属性。

3) @Autowired 注解

①根据类型实现自动装配。

②构造器、普通字段(即使是非 public)、一切具有参数的方法都可以应用@Autowired 注解

③默认情况下，所有使用@Autowired 注解的属性都需要被设置。当 Spring 找不到匹配的 bean 装配属性时，会抛出异常。

④若某一属性允许不被设置，可以设置@Autowired 注解的 required 属性为 false

⑤默认情况下，当 IOC 容器里存在多个类型兼容的 bean 时，Spring 会尝试匹配 bean 的 id 值是否与变量名相同，如果相同则进行装配。如果 bean 的 id 值不相同，通过类型的自动装配将无法工作。此时可以在@Qualifier 注解里提供 bean 的名称。Spring 甚至允许在方法的形参上标注@Qualifiter 注解以指定注入 bean 的名称。

⑥@Autowired 注解也可以应用在数组类型的属性上，此时 Spring 将会把所有匹配的 bean 进行自动装配。

⑦@Autowired 注解也可以应用在集合属性上，此时 Spring 读取该集合的类型信息，然后自动装配所有与之兼容的 bean。

⑧@Autowired 注解用在 java.util.Map 上时，若该 Map 的键值为 String，那么 Spring 将自动装配与值类型兼容的 bean 作为值，并以 bean 的 id 值作为键。

4) @Resource

@Resource 注解要求提供一个 bean 名称的属性，若该属性为空，则自动采用标注处的变量或方法名作为 bean 的名称。

5) @Inject

@Inject 和@Autowired 注解一样也是按类型注入匹配的 bean，但没有 reqired 属性。

第 3 章 AOP 前奏

3.1 提出问题

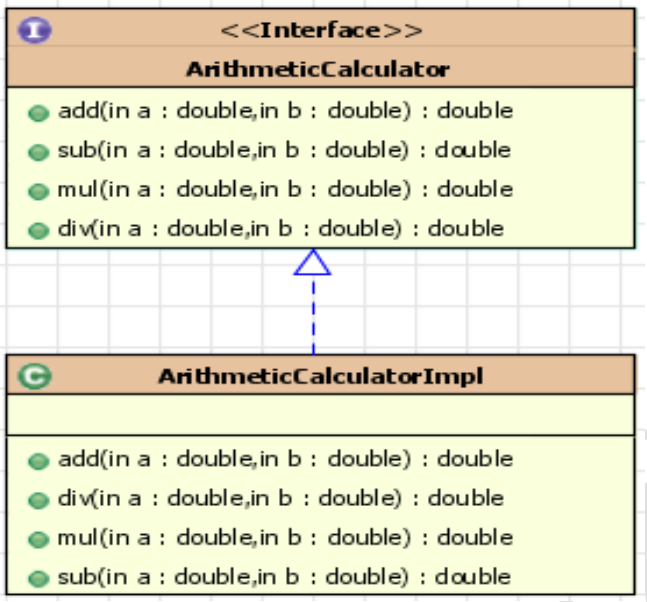
3.1.1 情景：数学计算器

1) 要求

①执行加减乘除运算

②日志：在程序执行期间追踪正在发生的活动

③验证：希望计算器只能处理正数的运算



2) 常规实现

```
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    @Override
    public void add(int i, int j) {

        System.out.println("日志:The method add begins with [" +
                i + ", " +  j + "]" );

        int result = i + j;
        System.out.println("result: " + result);

        System.out.println("日志:The method add ends with " + result);
    }

    @Override
    public void sub(int i, int j) {

        System.out.println("日志:The method sub begins with [" +
                i + ", " +  j + "]" );

        int result = i - j;
        System.out.println("result: " + result);

        System.out.println("日志:The method sub ends with " + result);
    }
```

3) 问题

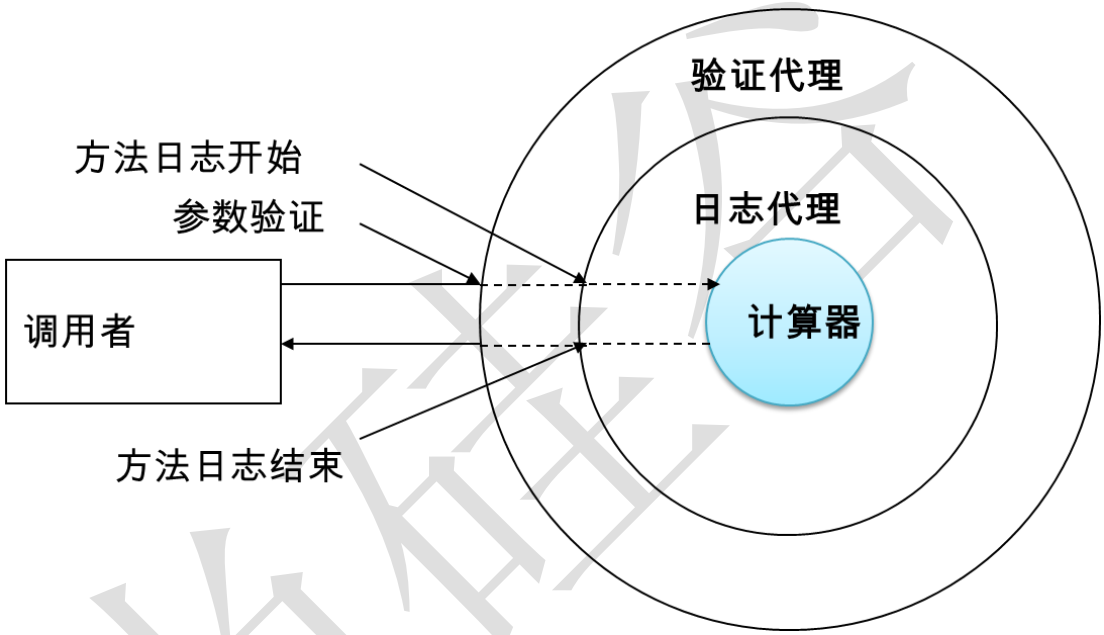
①代码混乱：越来越多的非业务需求(日志和验证等)加入后，原有的业务方法急剧膨胀。每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。

②代码分散：以日志需求为例，只是为了满足这个单一需求，就不得不在多个模块（方法）里多次重复相同的日志代码。如果日志需求发生变化，必须修改所有模块。

3.2 动态代理

3.2.1 动态代理的原理

代理设计模式的原理：**使用一个代理将原本对象包装起来**，然后用该代理对象"取代"原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。



3.2.2 动态代理的方式

1) 基于接口实现动态代理： JDK 动态代理
2) 基于继承实现动态代理： Cglib、Javassist 动态代理

3.3 数学计算器的改进

3.3.1 日志处理器

```
public class CalculatorLoggingHandler implements InvocationHandler {

    private Log log = LogFactory.getLog(this.getClass());

    private Object target;

    public CalculatorLoggingHandler(Object target) {
        super();
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        log.info("The method " + method.getName() + "() begins with " + Arrays.toString(args));
        Object result = method.invoke(target, args);
        log.info("The method " + method.getName() + "() ends with " + result);
        return result;
    }

    public static Object createProxy(Object target){
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new CalculatorLoggingHandler(target));
    }

}
```

3.3.2 验证处理器

```
public class CalculatorValidationHandler implements InvocationHandler {
    private Object target;

    public CalculatorValidationHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        for(Object arg : args){
            validate((Double)arg);
        }
        Object result = method.invoke(target, args);
        return result;
    }

    public static Object createProxy(Object target){
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new CalculatorValidationHandler(target));
    }

    private void validate(double a){
        if(a < 0)
            throw new IllegalArgumentException("Positive numbers only");
    }

}
```

3.3.3 测试代码

```
public class Main {
    public static void main(String[] args) {
        ArithmeticCalculator arithmeticCalculatorImpl =
            new ArithmeticCalculatorImpl();

        ArithmeticCalculator arithmeticCalculator
            = (ArithmeticCalculator) CalculatorValidationHandler
                .createProxy(CalculatorLoggingHandler
                    .createProxy(arithmeticCalculatorImpl));
        System.out.println(arithmeticCalculator.add(-12, 13));
    }
}
```

3.3.4 保存生成的动态代理类

在测试方法中加入如下代码：

Properties properties = System.getProperties();

properties.put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");

第 4 章　AOP 概述

4.1 AOP 概述

1) 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。

面向对象　纵向继承机制

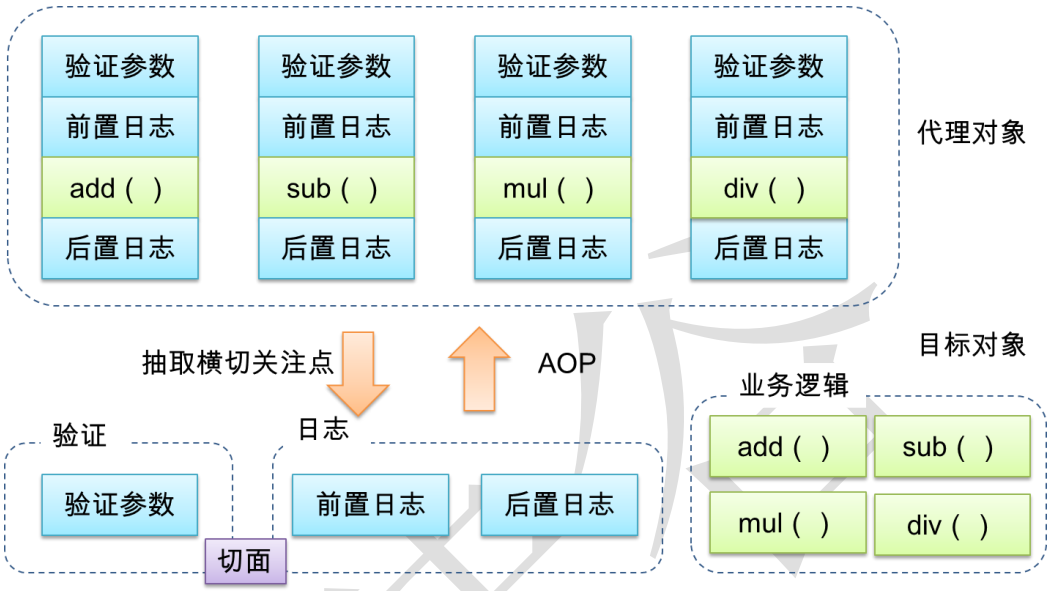
面向切面　横向抽取机制

2) AOP 编程操作的主要对象是切面(aspect)，而切面用于模块化横切关注点（公共功能）。
3) 在应用 AOP 编程时，仍然需要定义公共功能，但可以明确的定义这个功能应用在哪里，以什么方式应用，并且不必修改受影响的类。这样一来横切关注点就被模块化到特殊的类里——这样的类我们通常称之为“切面”。
4) AOP 的好处：

① 每个事物逻辑位于一个位置，代码不分散，便于维护和升级

② 业务模块更简洁，只包含核心业务代码

③ AOP 图解



4.2 AOP 术语

4.2.1 横切关注点

从每个方法中抽取出来的同一类非核心业务。

4.2.2 切面(Aspect)

封装横切关注点信息的类，每个关注点体现为一个通知方法。

4.2.3 通知(Advice)

切面必须要完成的各个具体工作

4.2.4 目标(Target)

被通知的对象

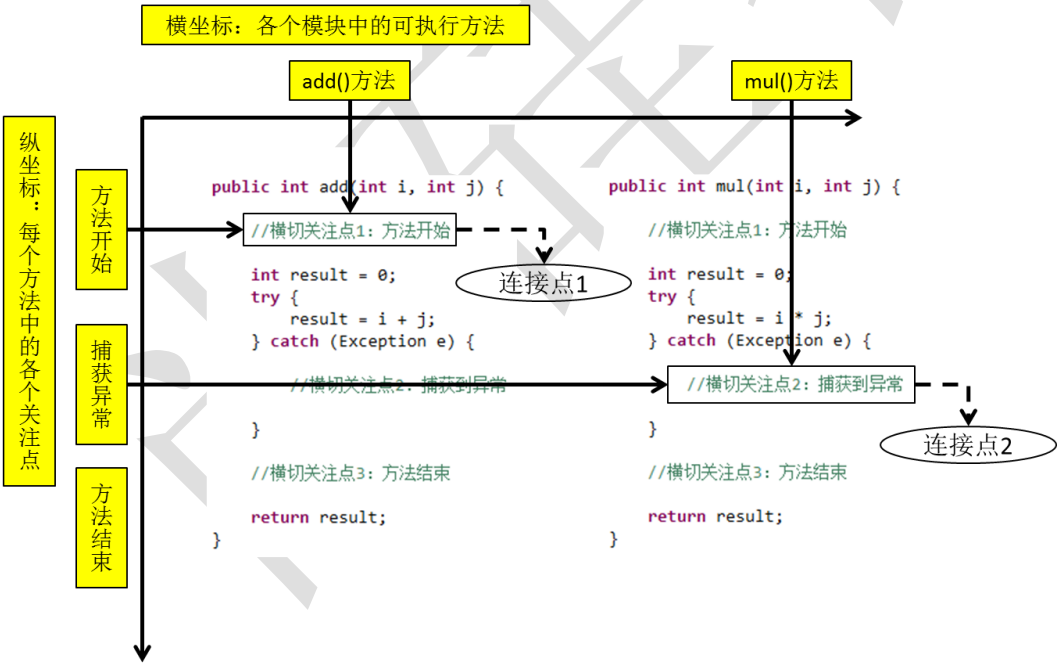
4.2.5 代理(Proxy)

向目标对象应用通知之后创建的代理对象

4.2.6 连接点(Joinpoint)

横切关注点在程序代码中的具体体现，对应程序执行的某个特定位置。例如：类某个方法调用前、调用后、方法捕获到异常后等。

在应用程序中可以使用横纵两个坐标来定位一个具体的连接点：

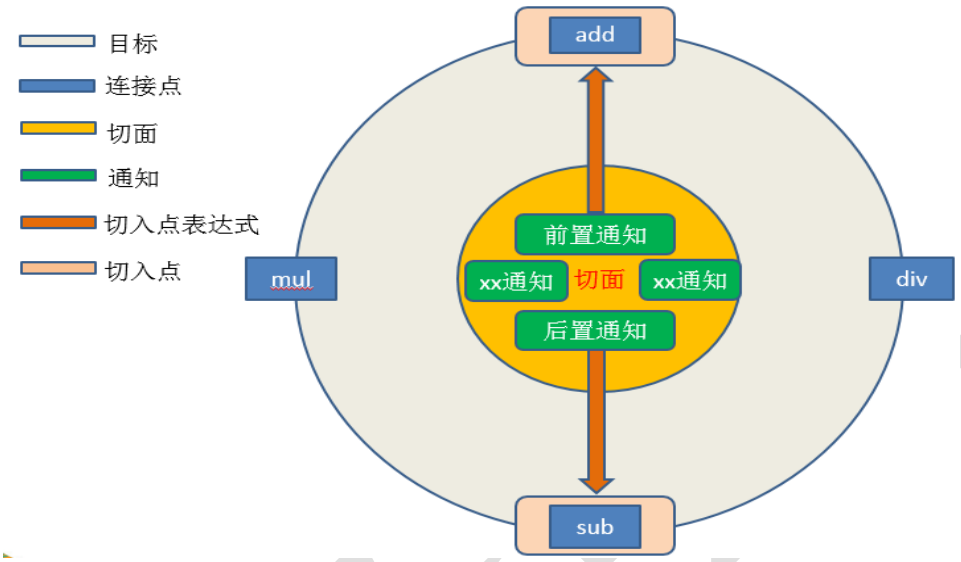


4.2.7 切入点(pointcut)：

定位连接点的方式。每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物。如果把连接点看作数据库中的记录，那么切入点就是查询条件——AOP 可以通过切

入点定位到特定的连接点。切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

4.2.8 图解



4.3 AspectJ

4.3.1 简介

AspectJ：Java 社区里最完整最流行的 AOP 框架。

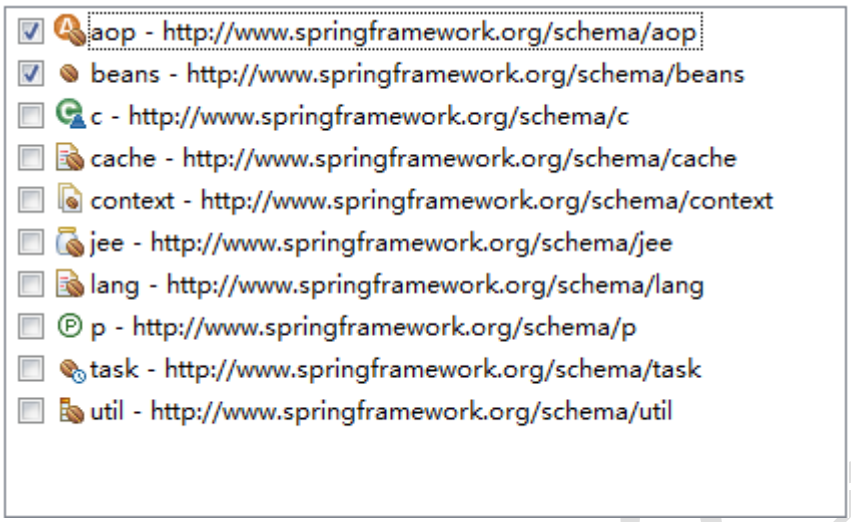
在 Spring2.0 以上版本中，可以使用基于 AspectJ 注解或基于 XML 配置的 AOP。

4.3.2 在 Spring 中启用 AspectJ 注解支持

1) 导入 JAR 包

- com.springsource.net.sf.cglib-2.2.0.jar
- com.springsource.org.aopalliance-1.0.0.jar
- com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
- spring-aop-4.0.0.RELEASE.jar
- spring-aspects-4.0.0.RELEASE.jar

2) 引入 aop 名称空间



3) 配置

<aop:aspectj-autoproxy>

当 Spring IOC 容器侦测到 bean 配置文件中的<aop:aspectj-autoproxy>元素时，会自动为与 AspectJ 切面匹配的 bean 创建代理

4.3.3 用 AspectJ 注解声明切面

1) 要在 Spring 中声明 AspectJ 切面，只需要在 IOC 容器中将切面声明为 bean 实例。
2) 当在 Spring IOC 容器中初始化 AspectJ 切面之后，Spring IOC 容器就会为那些与 AspectJ 切面相匹配的 bean 创建代理。
3) 在 AspectJ 注解中，切面只是一个带有@Aspect 注解的 Java 类，它往往要包含很多通知。
4) 通知是标注有某种注解的简单的 Java 方法。
5) AspectJ 支持 5 种类型的通知注解：

① @Before：前置通知，在方法执行之前执行

② @After：后置通知，在方法执行之后执行

③ @AfterRunning：返回通知，在方法返回结果之后执行

④ @AfterThrowing：异常通知，在方法抛出异常之后执行

⑥ @Around：环绕通知，围绕着方法执行

第 5 章 AOP 细节

5.1 切入点表达式

5.1.1 作用

通过表达式的方式定位一个或多个具体的连接点。

5.1.2 语法细节

1) 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]))

2) 举例说明

表达式	execution(* com.atguigu.spring.ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口中声明的所有方法。 第一个“*”代表任意修饰符及任意返回值。 第二个“*”代表任意方法。 “..”匹配任意数量、任意类型的参数。 若目标类、接口与该切面类在同一个包中可以省略包名。

表达式	execution(public * ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口的所有公有方法

表达式	execution(public double ArithmeticCalculator.*(..))
含义	ArithmeticCalculator 接口中返回 double 类型数值的方法

表达式	execution(public double ArithmeticCalculator.*(double, ..))
含义	第一个参数为 double 类型的方法。

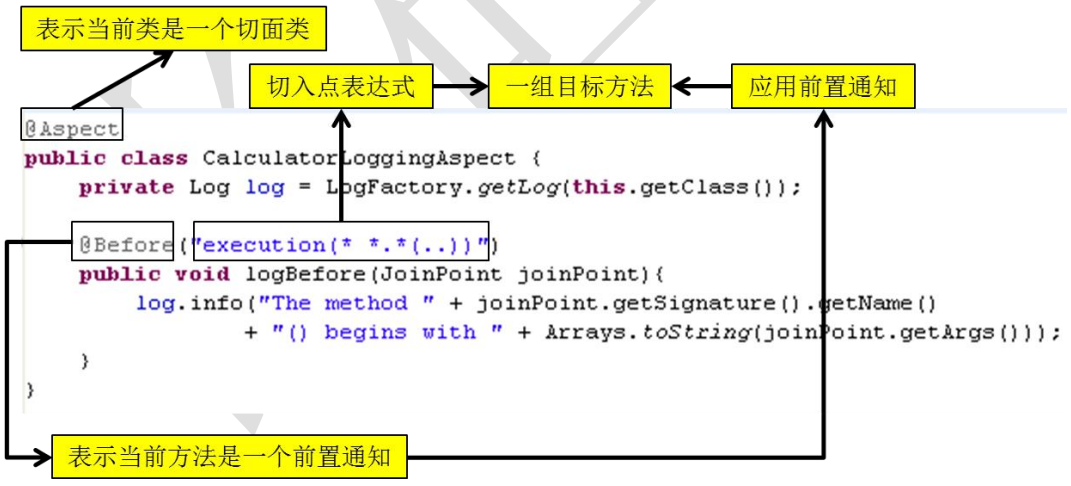
	“..” 匹配任意数量、任意类型的参数。

表达式	execution(public double ArithmeticCalculator.*(**double**, **double**))
含义	参数类型为 double，double 类型的方法

3）在 AspectJ 中，切入点表达式可以通过 “&&”、“||”、“!”等操作符结合起来。

表达式	execution (* *.add(int,..)) **||** execution(* *.sub(int,..))
含义	任意类中第一个参数为 int 类型的 add 方法或 sub 方法
表达式	!execution (* *.add(int,..))
含义	匹配不是任意类中第一个参数为 int 类型的 add 方法

5.1.3 切入点表达式应用到实际的切面类中

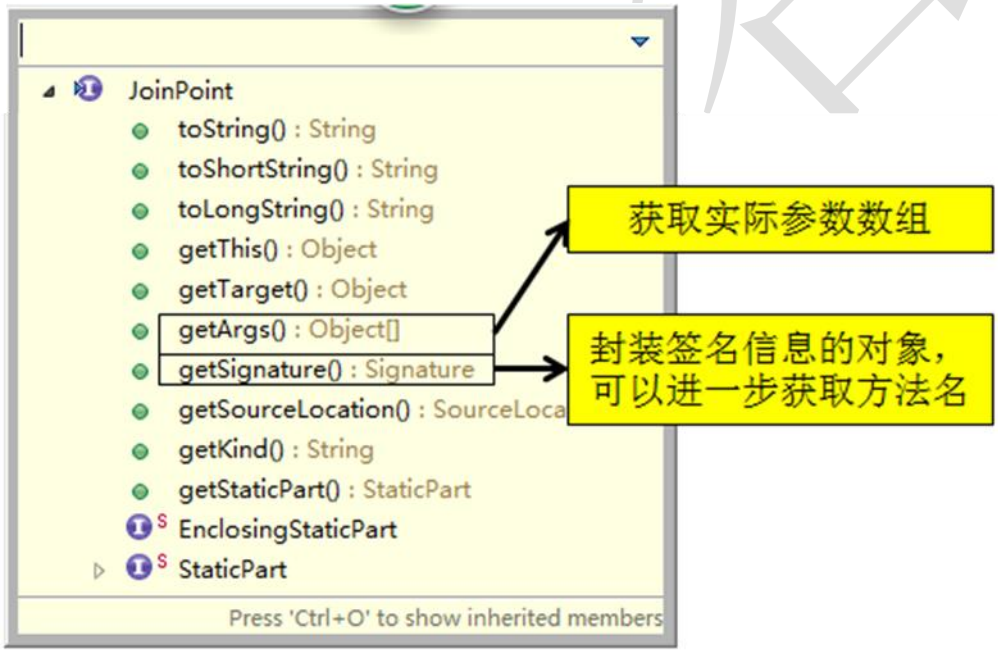


5.2 当前连接点细节

5.2.1 概述

切入点表达式通常都会是从宏观上定位一组方法，和具体某个通知的注解结合起来就能够确定对应的连接点。那么就一个具体的连接点而言，我们可能会关心这个连接点的一些具体信息，例如：当前连接点所在方法的方法名、当前传入的参数值等等。这些信息都封装在 JoinPoint 接口的实例对象中。

5.2.2 JoinPoint



5.3 通知

5.3.1 概述

1) 在具体的连接点上要执行的操作。
2) 一个切面可以包括一个或者多个通知。
3) 通知所使用的注解的值往往是切入点表达式。

5.3.2 前置通知

1) 前置通知：在方法执行之前执行的通知
2) 使用@Before 注解

5.3.3 后置通知

1) 后置通知：后置通知是在连接点完成之后执行的，即连接点返回结果或者抛出异常的时候
2) 使用@After 注解

5.3.4 返回通知

1) 返回通知：无论连接点是正常返回还是抛出异常，后置通知都会执行。如果只想在连接点返回的时候记录日志，应使用返回通知代替后置通知。
2) 使用@AfterReturning 注解,在返回通知中访问连接点的返回值

①在返回通知中，只要将 returning 属性添加到@AfterReturning 注解中，就可以访问连接点的返回值。该属性的值即为用来传入返回值的参数名称

②必须在通知方法的签名中添加一个同名参数。在运行时 Spring AOP 会通过这个参数传递返回值

③原始的切点表达式需要出现在 pointcut 属性中

```
@AfterReturning(pointcut="execution(* *.*(..))", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result){
    log.info("The method " + joinPoint.getSignature().getName()
            + "() ends with " + result);
}
```

5.3.5 异常通知

1) 异常通知：只在连接点抛出异常时才执行异常通知
2) 将 throwing 属性添加到@AfterThrowing 注解中，也可以访问连接点抛出的异常。Throwable 是所有错误和异常类的顶级父类，所以在异常通知方法可以捕获到任何错误和异常。
3) 如果只对某种特殊的异常类型感兴趣，可以将参数声明为其他异常的参数类型。然后通知就只在抛出这个类型及其子类的异常时才被执行

5.3.6 环绕通知

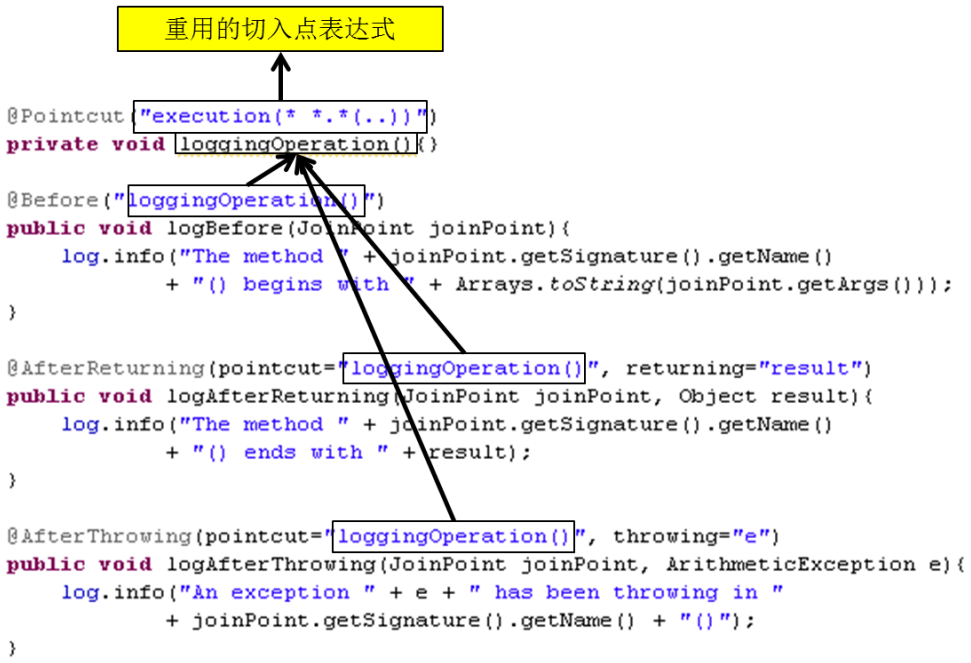
1) 环绕通知是所有通知类型中功能最为强大的，能够全面地控制连接点，甚至可以控制是否执行连接点。
2) 对于环绕通知来说，连接点的参数类型必须是 ProceedingJoinPoint。它是 JoinPoint 的子接口，允许控制何时执行，是否执行连接点。
3) 在环绕通知中需要明确调用 ProceedingJoinPoint 的 proceed()方法来执行被代理的方法。如果忘记这样做就会导致通知被执行了，但目标方法没有被执行。
4) 注意：环绕通知的方法需要返回目标方法执行之后的结果，即调用 joinPoint.proceed(); 的返回值，否则会出现空指针异常。

```
@Around("execution(* *.*(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable{
    log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));

    try {
        joinPoint.proceed();
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends");
    } catch (Throwable e) {
        log.info("An exception " + e + " has been throwing in "
                + joinPoint.getSignature().getName() + "()");
        throw e;
    }
}
```

5.4 重用切入点定义

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成简单的方法。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



5.4 指定切面的优先级

1) 在同一个连接点上应用不止一个切面时，除非明确指定，否则它们的优先级是不确定的。
2) 切面的优先级可以通过实现 Ordered 接口或利用@Order 注解指定。
3) 实现 Ordered 接口，getOrder()方法的返回值越小，优先级越高。
4) 若使用@Order 注解，序号出现在注解中

```
@Aspect
@Order(0)
public class CalculaotorValidationAspect {

@Aspect
@Order(1)
public class CalculatorLoggingAspect {
```

第 6 章 以 XML 方式配置切面

6.1 概述

除了使用 AspectJ 注解声明切面，Spring 也支持在 bean 配置文件中声明切面。这种声明是通过 aop 名称空间中的 XML 元素完成的。

正常情况下，基于注解的声明要优先于基于 XML 的声明。通过 AspectJ 注解，切面可以与 AspectJ 兼容，而基于 XML 的配置则是 Spring 专有的。由于 AspectJ 得到越来越多的 AOP 框架支持，所以以注解风格编写的切面将会有更多重用的机会。

6.2 配置细节

在 bean 配置文件中，所有的 Spring AOP 配置都必须定义在<aop:config>元素内部。对于每个切面而言，都要创建一个<aop:aspect>元素来为具体的切面实现引用后端 bean 实例。

切面 bean 必须有一个标识符，供<aop:aspect>元素引用。

```
<bean id="calculatorLoggingAspect"
    class="org.simpleit.CalculatorLoggingAspect"></bean>

<bean id="calculatorValidationAspect"
    class="org.simpleit.CalculaotorValidationAspect"></bean>

<aop:config>
    <aop:aspect id="loggingAspect"
        ref="calculatorLoggingAspect"></aop:aspect>

    <aop:aspect id="validationAspect"
        ref="calculatorValidationAspect"></aop:aspect>
</aop:config>
```

6.3 声明切入点

1) 切入点使用<aop:pointcut>元素声明。
2) 切入点必须定义在<aop:aspect>元素下，或者直接定义在<aop:config>元素下。
   ① 定义在<aop:aspect>元素下：只对当前切面有效
   ② 定义在<aop:config>元素下：对所有切面都有效
3) 基于 XML 的 AOP 配置不允许在切入点表达式中用名称引用其他切入点。

```
<aop:config>
    <aop:pointcut id="testOperation"
        expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>

    <aop:aspect id="loggingAspect"
        ref="calculatorLoggingAspect">
    </aop:aspect>

    <aop:aspect id="validationAspect"
        ref="calculatorValidationAspect">
    </aop:aspect>
</aop:config>
```

6.4 声明通知

1) 在 aop 名称空间中，每种通知类型都对应一个特定的 XML 元素。
2) 通知元素需要使用<pointcut-ref>来引用切入点，或用<pointcut>直接嵌入切入点表达式。
3) method 属性指定切面类中通知方法的名称

```
<aop:config>
    <aop:pointcut id="testOperation"
        expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>

    <aop:aspect id="loggingAspect"
        ref="calculatorLoggingAspect">
        <aop:after method="logBefore"
            pointcut-ref="testOperation"/>
    </aop:aspect>

    <aop:aspect id="validationAspect"
        ref="calculatorValidationAspect">
        <aop:before method="validateBefore"
            pointcut-ref="testOperation"/>
    </aop:aspect>
</aop:config>
```

第 7 章 JdbcTemplate

7.1 概述

为了使 JDBC 更加易于使用，Spring 在 JDBC API 上定义了一个抽象层，以此建立一个 JDBC 存取框架。

作为 Spring JDBC 框架的核心，JDBC 模板的设计目的是为不同类型的 JDBC 操作提供模板方法，通过这种方式，可以在尽可能保留灵活性的情况下，将数据库存取的工作量降到最低。

可以将 Spring 的 JdbcTemplate 看作是一个小型的轻量级持久化层框架，和我们之前使用过的 DBUtils 风格非常接近。

7.2 环境准备

7.2.1 导入 JAR 包

1) IOC 容器所需要的 JAR 包

commons-logging-1.1.1.jar

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELEASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

2) JdbcTemplate 所需要的 JAR 包

spring-jdbc-4.0.0.RELEASE.jar

spring-orm-4.0.0.RELEASE.jar

spring-tx-4.0.0.RELEASE.jar

3) 数据库驱动和数据源

druid-1.1.9.jar

mysql-connector-java-5.1.7-bin.jar

7.2.2 创建连接数据库基本信息属性文件

```
user=root
password=root
jdbcUrl=jdbc:mysql:///query_data
driverClass=com.mysql.jdbc.Driver
```

```
initialPoolSize=30
minPoolSize=10
maxPoolSize=100
acquireIncrement=5
maxStatements=1000
maxStatementsPerConnection=10
```

7.2.3 在 Spring 配置文件中配置相关的 bean

1) 数据源对象

```
<context:property-placeholder location="classpath:jdbc.properties"/>

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="user" value="${user}"/>
    <property name="password" value="${password}"/>
    <property name="jdbcUrl" value="${jdbcUrl}"/>
    <property name="driverClass" value="${driverClass}"/>
    <property name="initialPoolSize" value="${initialPoolSize}"/>
    <property name="minPoolSize" value="${minPoolSize}"/>
    <property name="maxPoolSize" value="${maxPoolSize}"/>
    <property name="acquireIncrement" value="${acquireIncrement}"/>
    <property name="maxStatements" value="${maxStatements}"/>
    <property name="maxStatementsPerConnection"
                    value="${maxStatementsPerConnection}"/>
</bean>
```

2) JdbcTemplate 对象

```
<bean id="template"
```

```
        class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

7.3 持久化操作

1) 增删改

JdbcTemplate.update(String, Object...)

2) 批量增删改

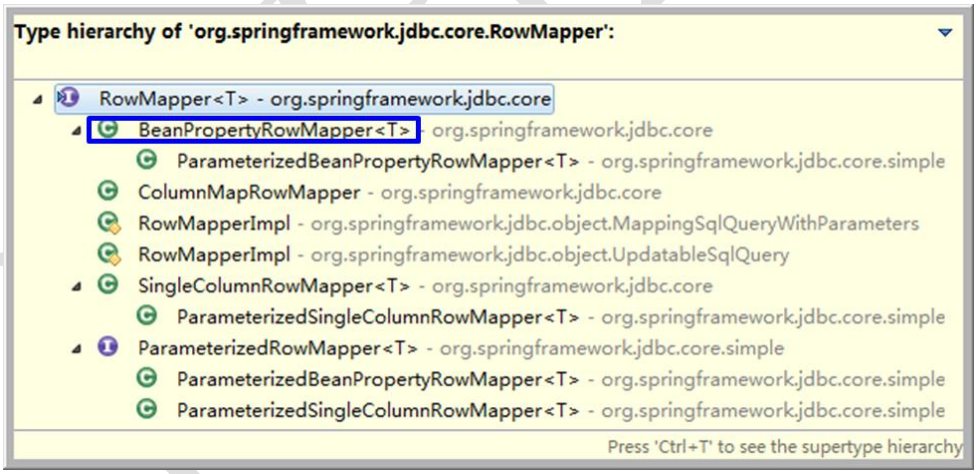
JdbcTemplate.batchUpdate(String, List<Object[]>)

Object[]封装了 SQL 语句每一次执行时所需要的参数

List 集合封装了 SQL 语句多次执行时的所有参数

3) 查询单行

JdbcTemplate.queryForObject(String, RowMapper<Department>, Object...)



4) 查询多行

JdbcTemplate.query(String, RowMapper<Department>, Object...)

RowMapper 对象依然可以使用 BeanPropertyRowMapper

5) 查询单一值

JdbcTemplate.queryForObject(String, Class, Object...)

7.5 使用 JdbcTemplate 实现 Dao

1) 通过 IOC 容器自动注入

JdbcTemplate 类是线程安全的，所以可以在 IOC 容器中声明它的单个实例，并将这个实例注入到所有的 Dao 实例中。

```
@Repository
public class EmployeeDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public Employee get(Integer id){
        //...
    }
}
```

第 8 章　声明式事务管理

8.1 事务概述

1) 在 JavaEE 企业级开发的应用领域，为了保证数据的完整性和一致性，必须引入数据库事务的概念，所以事务管理是企业级应用程序开发中必不可少的技术。
2) 事务就是一组由于逻辑上紧密关联而合并成一个整体(工作单元)的多个数据库操作，这些操作要么都执行，要么都不执行。
3) 事务的四个关键属性(ACID)

①原子性(atomicity)：“原子”的本意是“不可再分”，事务的原子性表现为一个事务中涉及到的多个操作在逻辑上缺一不可。事务的原子性要求事务中的所有操作要么都执行，要么都不执行。

②一致性(consistency)：“一致”指的是数据的一致，具体是指：所有数据都处于满足业

务规则的一致性状态。一致性原则要求：一个事务中不管涉及到多少个操作，都必须保证事务执行之前数据是正确的，事务执行之后数据仍然是正确的。如果一个事务在执行的过程中，其中某一个或某几个操作失败了，则必须将其他所有操作撤销，将数据恢复到事务执行之前的状态，这就是回滚。

③隔离性(isolation)：在应用程序实际运行过程中，事务往往是并发执行的，所以很有可能有许多事务同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。隔离性原则要求多个事务在并发执行过程中不会互相干扰。

④持久性(durability)：持久性原则要求事务执行完成后，对数据的修改永久的保存下来，不会因各种系统错误或其他意外情况而受到影响。通常情况下，事务对数据的修改应该被写入到持久化存储器中。

8.2 Spring 事务管理

8.2.1 编程式事务管理

1) 使用原生的 JDBC API 进行事务管理
   ①获取数据库连接 Connection 对象
   ②取消事务的自动提交
   ③执行操作
   ④正常完成操作时手动提交事务
   ⑤执行失败时回滚事务
   ⑥关闭相关资源
2) 评价

   使用原生的 JDBC API 实现事务管理是所有事务管理方式的基石，同时也是最典型的编程式事务管理。编程式事务管理需要将事务管理代码嵌入到业务方法中来控制事务的提交和回滚。在使用编程的方式管理事务时，必须在每个事务操作中包含额外的事务管理代码。相对于核心业务而言，事务管理的代码显然属于非核心业务，如果多个模块都使用同样模式的代码进行事务管理，显然会造成较大程度的代码冗余。

8.2.2 声明式事务管理

大多数情况下声明式事务比编程式事务管理更好：它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。

事务管理代码的固定模式作为一种横切关注点，可以通过 AOP 方法模块化，进而借助 Spring AOP 框架实现声明式事务管理。

Spring 在不同的事务管理 API 之上定义了一个抽象层，通过配置的方式使其生效，从而让应用程序开发人员不必了解事务管理 API 的底层实现细节，就可以使用 Spring 的事务管理机制。

Spring 既支持编程式事务管理，也支持声明式的事务管理。

8.2.3 Spring 提供的事务管理器

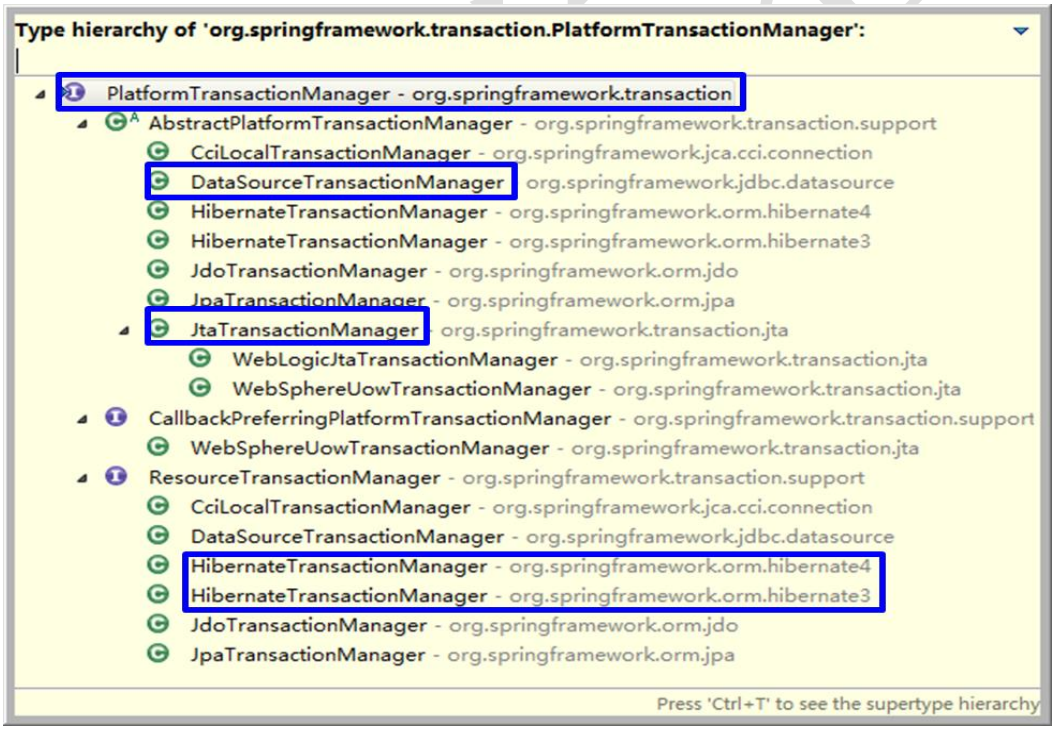
Spring 从不同的事务管理 API 中抽象出了一整套事务管理机制，让事务管理代码从特定的事务技术中独立出来。开发人员通过配置的方式进行事务管理，而不必了解其底层是如何实现的。

Spring 的核心事务管理抽象是它为事务管理封装了一组独立于技术的方法。无论使用 Spring 的哪种事务管理策略(编程式或声明式)，事务管理器都是必须的。

事务管理器可以以普通的 bean 的形式声明在 Spring IOC 容器中。

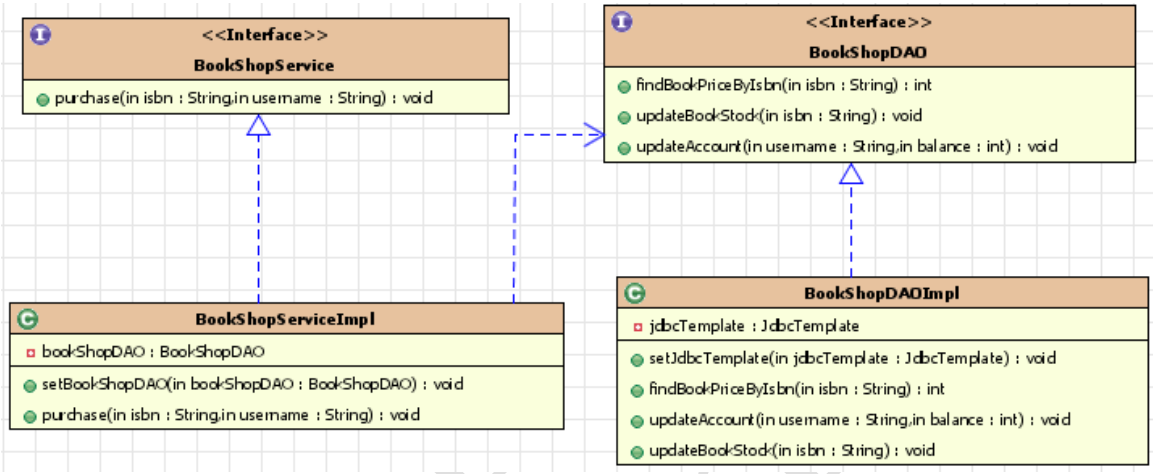
8.2.4 事务管理器的主要实现

1) DataSourceTransactionManager：在应用程序中只需要处理一个数据源，而且通过 JDBC 存取。
2) JtaTransactionManager：在 JavaEE 应用服务器上用 JTA(Java Transaction API)进行事务管理
3) HibernateTransactionManager：用 Hibernate 框架存取数据库



8.3 测试数据准备

8.3.1 需求



8.3.2 数据库表

```
CREATE TABLE book (
    isbn VARCHAR (50) PRIMARY KEY,
    book_name VARCHAR (100),
    price INT
) ;

CREATE TABLE book_stock (
    isbn VARCHAR (50) PRIMARY KEY,
    stock INT
    ) ;

CREATE TABLE account (
    username VARCHAR (50) PRIMARY KEY,
    balance INT
```

```
);

INSERT INTO account (`username`,`balance`) VALUES ('Tom',100000);
INSERT INTO account (`username`,`balance`) VALUES ('Jerry',150000);

INSERT INTO book (`isbn`,`book_name`,`price`) VALUES ('ISBN-001','book01',100);
INSERT INTO book (`isbn`,`book_name`,`price`) VALUES ('ISBN-002','book02',200);
INSERT INTO book (`isbn`,`book_name`,`price`) VALUES ('ISBN-003','book03',300);
INSERT INTO book (`isbn`,`book_name`,`price`) VALUES ('ISBN-004','book04',400);
INSERT INTO book (`isbn`,`book_name`,`price`) VALUES ('ISBN-005','book05',500);

INSERT INTO book_stock (`isbn`,`stock`) VALUES ('ISBN-001',1000);
INSERT INTO book_stock (`isbn`,`stock`) VALUES ('ISBN-002',2000);
INSERT INTO book_stock (`isbn`,`stock`) VALUES ('ISBN-003',3000);
INSERT INTO book_stock (`isbn`,`stock`) VALUES ('ISBN-004',4000);
INSERT INTO book_stock (`isbn`,`stock`) VALUES ('ISBN-005',5000);
```

8.4 初步实现

1) 配置文件

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

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

2) 在需要进行事务控制的方法上加注解 @Transactional

8.5 事务的传播行为

8.5.1 简介

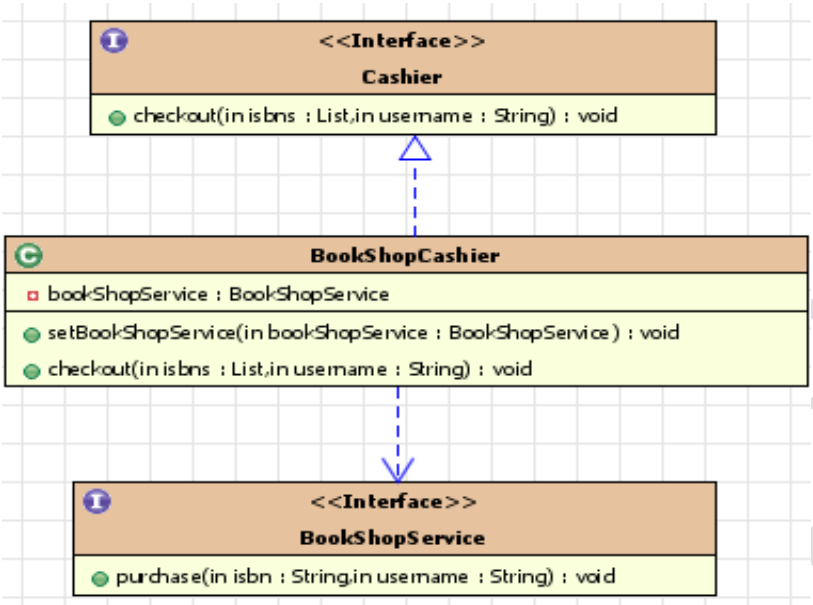
当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

事务的传播行为可以由传播属性指定。Spring 定义了 7 种类传播行为。

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行．如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行．否则它可以不运行在事务中．
NOT_SUPPORTE	当前的方法不应该运行在事务中．如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中．如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行．否则，就启动一个新的事务，并在它自己的事务内运行．

事务传播属性可以在@Transactional 注解的 propagation 属性中定义。

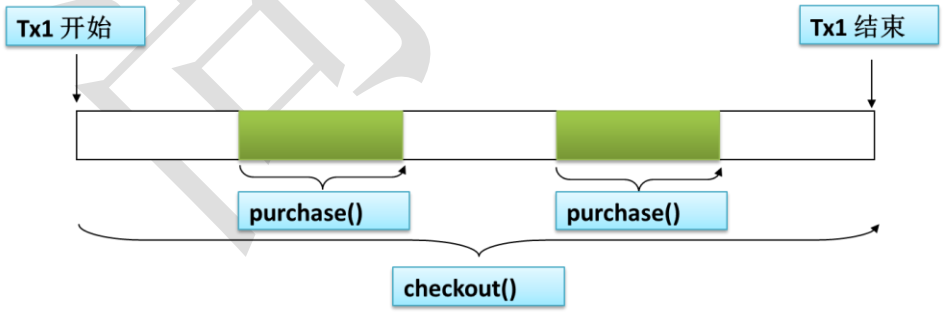
8.5.2 测试



1) . 说明

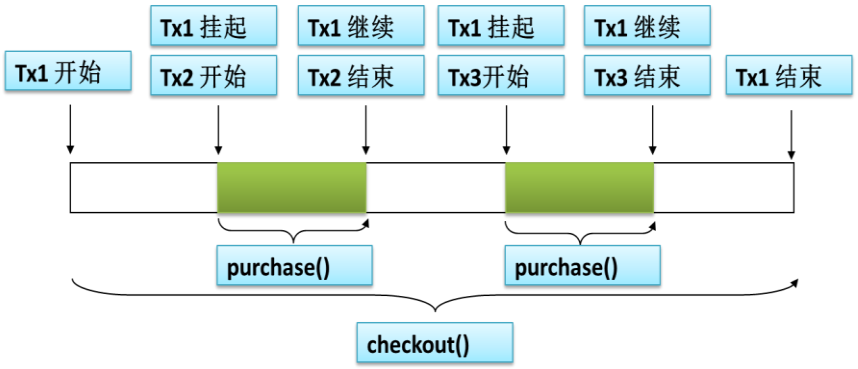
①REQUIRED 传播行为

当 bookService 的 purchase()方法被另一个事务方法 checkout()调用时，它默认会在现有的事务内运行。这个默认的传播行为就是 REQUIRED。因此在 checkout()方法的开始和终止边界内只有一个事务。这个事务只在 checkout()方法结束的时候被提交，结果用户一本书都买不了。



②. REQUIRES_NEW 传播行为

表示该方法必须启动一个新事务，并在自己的事务内运行。如果有事务在运行，就应该先挂起它。



8.5.3 补充

在 Spring 2.x 事务通知中，可以像下面这样在<tx:method>元素中设定传播事务属性。

```
<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="purchase" propagation="REQUIRES_NEW"/>
    </tx:attributes>
</tx:advice>
```

8.6 事务的隔离级别

8.6.1 数据库事务并发问题

假设现在有两个事务：Transaction01 和 Transaction02 并发执行。

1) 脏读

①Transaction01 将某条记录的 AGE 值从 20 修改为 30。

②Transaction02 读取了 Transaction01 更新后的值：30。

③Transaction01 回滚，AGE 值恢复到了 20。

④Transaction02 读取到的 30 就是一个无效的值。

2) 不可重复读

①Transaction01 读取了 AGE 值为 20。

②Transaction02 将 AGE 值修改为 30。

③Transaction01 再次读取 AGE 值为 30，和第一次读取不一致。

3) 幻读

①Transaction01 读取了 STUDENT 表中的一部分数据。

②Transaction02 向 STUDENT 表中插入了新的行。

③Transaction01 读取了 STUDENT 表时，多出了一些行。

8.6.2 隔离级别

数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。**一个事务与其他事务隔离的程度称为隔离级别**。SQL 标准中规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

1) **读未提交**：READ UNCOMMITTED

允许 Transaction01 读取 Transaction02 未提交的修改。

2) **读已提交**：READ COMMITTED

要求 Transaction01 只能读取 Transaction02 已提交的修改。

3) **可重复读**：REPEATABLE READ

确保 Transaction01 可以多次从一个字段中读取到相同的值，即 Transaction01 执行期间禁止其它事务对这个字段进行更新。

4) **串行化**：SERIALIZABLE

确保 Transaction01 可以多次从一个表中读取到相同的行，在 Transaction01 执行期间，禁止其它事务对这个表进行添加、更新、删除操作。可以避免任何并发问题，但性能十分低下。

5) 各个隔离级别解决并发问题的能力见下表

	脏读	不可重复读	幻读
READ UNCOMMITTED	有	有	有
READ COMMITTED	无	有	有
REPEATABLE READ	无	无	有
SERIALIZABLE	无	无	无

6) 各种数据库产品对事务隔离级别的支持程度

	Oracle	MySQL
READ UNCOMMITTED	×	√
READ COMMITTED	√(默认)	√
REPEATABLE READ	×	√(默认)
SERIALIZABLE	√	√

8.6.3 在 Spring 中指定事务隔离级别

1) 注解

用@Transactional 注解声明式地管理事务时可以在@Transactional 的 isolation 属性中设置隔离级别

2) XML

在 Spring 2.x 事务通知中，可以在<tx:method>元素中指定隔离级别

```
<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="purchase"
            propagation="REQUIRES_NEW"
            isolation="READ_COMMITTED"/>
    </tx:attributes>
</tx:advice>
```

8.7 触发事务回滚的异常

8.7.1 默认情况

捕获到 RuntimeException 或 Error 时回滚，而捕获到编译时异常不回滚。

8.7.2 设置途经

1) 注解@Transactional 注解

① rollbackFor 属性：指定遇到时必须进行回滚的异常类型，可以为多个

② noRollbackFor 属性：指定遇到时不回滚的异常类型，可以为多个

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
        isolation=Isolation.READ_COMMITTED,
        rollbackFor={IOException.class, SQLException.class},
        noRollbackFor=ArithmeticException.class)
public void purchase(String isbn, String username) {
```

2) XML

在 Spring 2.x 事务通知中，可以在<tx:method>元素中指定回滚规则。如果有不止

一种异常则用逗号分隔。

```
<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="purchase"
            propagation="REQUIRES_NEW"
            isolation="READ_COMMITTED"
            rollback-for="java.io.IOException, java.sql.SQLException"
            no-rollback-for="java.lang.ArithmeticException"/>
    </tx:attributes>
</tx:advice>
```

8.8 事务的超时和只读属性

8.8.1 简介

由于事务可以在行和表上获得锁，因此长事务会占用资源，并对整体性能产生影响。

如果一个事务只读取数据但不做修改，数据库引擎可以对这个事务进行优化。

超时事务属性：事务在强制回滚之前可以保持多久。这样可以防止长期运行的事务占用资源。

只读事务属性: 表示这个事务只读取数据但不更新数据, 这样可以帮助数据库引擎优化事务。

8.8.2 设置

1) 注解

@Transaction 注解

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
        isolation=Isolation.READ_COMMITTED,
        rollbackFor={IOException.class, SQLException.class},
        noRollbackFor=ArithmeticException.class,
        readOnly=true,
        timeout=30)
public void purchase(String isbn, String username) {
```

2) XML

在 Spring 2.x 事务通知中，超时和只读属性可以在<tx:method>元素中进行指定

```
<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="purchase"
            propagation="REQUIRES_NEW"
            isolation="READ_COMMITTED"
            rollback-for="java.io.IOException, java.sql.SQLException"
            no-rollback-for="java.lang.ArithmeticException"
            timeout="30"
            read-only="true"/>
    </tx:attributes>
</tx:advice>
```

8.9 基于 XML 文档的声明式事务配置

```
<!-- 配置事务切面 -->
<aop:config>
    <aop:pointcut
        expression="execution(*
com.atguigu.tx.component.service.BookShopServiceImpl.purchase(..))"
        id="txPointCut"/>
    <!-- 将切入点表达式和事务属性配置关联到一起 -->
    <aop:advisor advice-ref="myTx" pointcut-ref="txPointCut"/>
</aop:config>

<!-- 配置基于 XML 的声明式事务 -->
<tx:advice id="myTx" transaction-manager="transactionManager">
    <tx:attributes>
        <!-- 设置具体方法的事务属性 -->
        <tx:method name="find*" read-only="true"/>
        <tx:method name="get*" read-only="true"/>
```

```
            <tx:method name="purchase"
                isolation="READ_COMMITTED"
    no-rollback-for="java.lang.ArithmeticException,java.lang.NullPointerException"
                propagation="REQUIRES_NEW"
                read-only="false"
                timeout="10"/>
        </tx:attributes>
    </tx:advice>
```