目录

[1 Spring项目结构 2](#_Toc42636519)

[2 IoC容器 2](#_Toc42636520)

[2.1 理论介绍 2](#_Toc42636521)

[2.2 注入方法 2](#_Toc42636522)

[2.3 依赖关系的管理 3](#_Toc42636523)

[2.4 Spring的IoC容器 3](#_Toc42636524)

[2.4.1 BeanFactory 4](#_Toc42636525)

[2.4.2 ApplicationContext 4](#_Toc42636526)

[3 Bean 6](#_Toc42636527)

[3.1 Bean介绍 6](#_Toc42636528)

[3.2 <bean></bean> 6](#_Toc42636529)

[3.2.1 <bean>的基本属性 6](#_Toc42636530)

[3.2.2 <bean>对象的参数注入方式 6](#_Toc42636531)

[3.2.3 <bean>对象的参数注入类型 7](#_Toc42636532)

[3.2.4 <bean>的继承 9](#_Toc42636533)

[3.2.5 <bean>的生命周期 scope 9](#_Toc42636534)

[3.3 <beans> 10](#_Toc42636535)

[3.4 FactoryBean 10](#_Toc42636536)

[4 Spring中的一些概念 12](#_Toc42636537)

[4.1 Aware 12](#_Toc42636538)

[5 Spring 数据访问 13](#_Toc42636539)

[5.1 数据源 13](#_Toc42636540)

[5.1.1 DBCP 14](#_Toc42636541)

[5.2 JPA 14](#_Toc42636542)

[6 AOP 15](#_Toc42636543)

[6.1 理论 15](#_Toc42636544)

[6.2 Spring AOP 15](#_Toc42636545)

[6.3 Spring Boot基于注解的AOP编程 16](#_Toc42636546)

[6.3.1 实现流程 16](#_Toc42636547)

[6.3.2 切面方法 16](#_Toc42636548)

# Spring项目结构

# IoC容器

## 理论介绍

1. IoC(Inversion of Control)控制反转，别名叫依赖注入(Dependency Injection)
2. 一般情况下，
   1. 我们想要使用类A中的方法，就需要先创建一个类A的对象实例。
   2. 如果类A还依赖于类B，那么一般会在类A的构造方法中创建一个类B的对象实例，依次类推。
   3. 所以我们想要使用A中的方法，就得创建一堆依赖对象
3. IoC理念下：
   1. 将所有类的信息和类之间的依赖关系传递给IoC容器，IoC容器负责所有类的对象创建
   2. IoC容器对外提供获取类对象实例的方法，我们需要使用哪个类，就通过该方法获取类的实例化对象
4. IoC Service Provider的基本功能是业务对象的创建管理和业务对象间的依赖绑定。IoC容器包含了上述功能，同时还需要实现更多的管理功能

## 注入方法

被注入对象如何通知IoC容器自己需要使用哪些类的实例对象，让IoC容器为自己提供服务。

1. 构造注入
   1. 构造方法的形参就是需要使用的对象，将其赋值给本地变量this…
2. Setter注入
   1. 使用setXXX()方法为本地变量赋值，形参就是所需对象
3. 接口注入
   1. 比较复杂，不提倡使用

## 依赖关系的管理

如何告知IoC容器业务对象之间的依赖关系

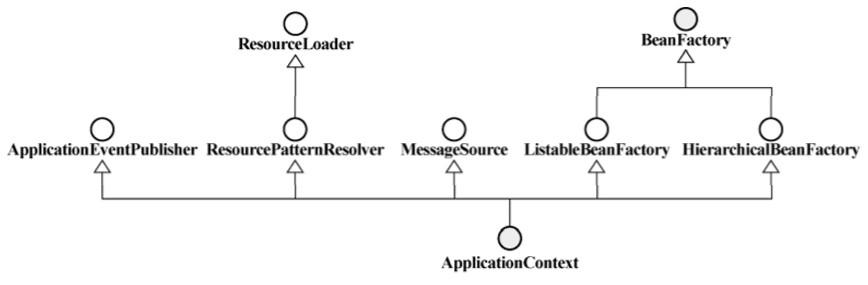
1. 直接编码方式
   1. 创建一个IoC容器对象，然后使用编码的方式添加一个个类与类间的依赖关系
2. 配置文件方式
   1. 最常用的方式，比如XML文件、properties
   2. 每个类都是一个<bean>，类间的依赖关系通过<property>关键字指定
3. 元数据/注解方式

## Spring的IoC容器



Spring提供了两种容器类型：BeanFactory和ApplicationContext

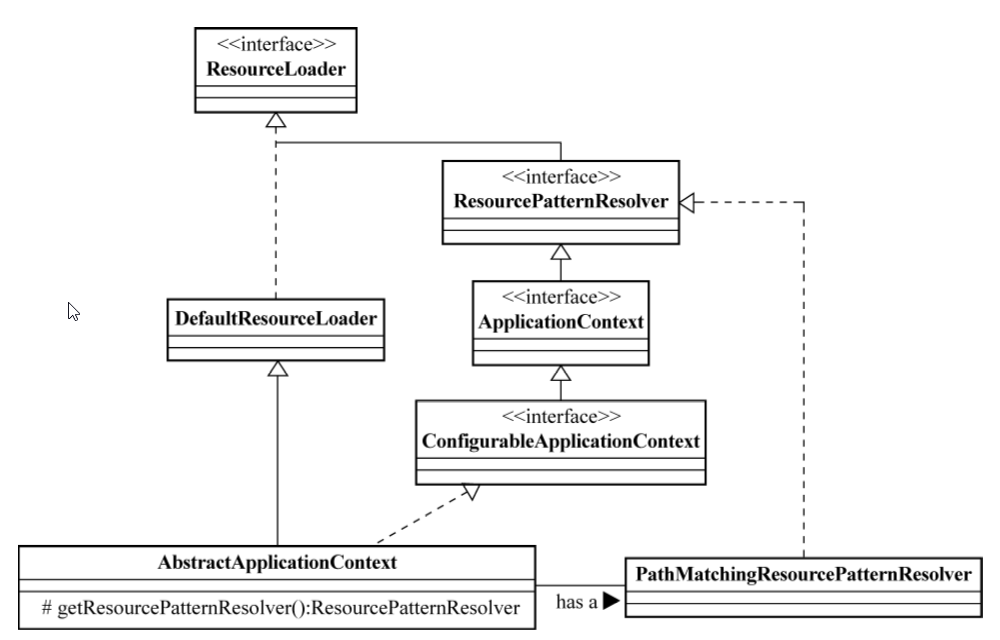
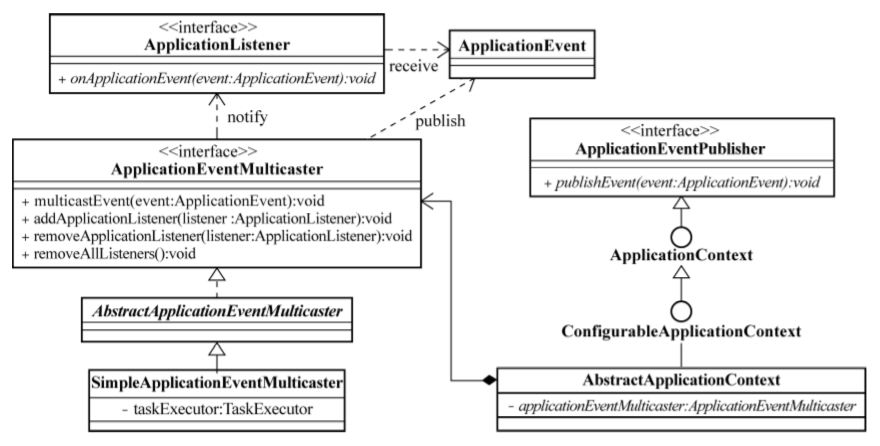
1. BeanFactory：默认采用延迟初始化策略，对象需要使用时才进行初始化，启动初期较快
2. ApplicationContext：默认全部初始化并绑定，在BeanFactory的基础上增加了更多功能



### BeanFactory

1. 是一个接口，需要调用具体的实现类(如DefaultListableBeanFactory)完成容器功能
2. 提供多个getBean()方法用于通过Bean name或者Bean类型获取容器创建管理的对象实例
3. FactoryBean：一种特殊的Bean，用于生成所需类的实例化对象并返回
   1. 一般来说，某个bean A依赖于另一个bean B，说明A中需要使用B的实例化对象
   2. 但是对于FactoryBean B，是A需要使用B生成的一个其他类的实例化对象

### ApplicationContext

1. ApplicationContext在BeanFactory的基础上增加了更多功能
   1. 统一资源加载策略
   2. 国际化信息支持
   3. 容器内部事件发布
2. 统一资源加载策略
   1. 为了统一资源定位和加载的规范，Spring提供了各种Resource接口，比如ResourceLoader、ResourcePatternResolver (批量查找的ResourceLoader)等
   2. ApplicationContext接口继承了ResourcePatternResolver接口，因此拥有了资源定位和加载的功能
   3. 
3. 国际化信息支持
   1. JAVA提供了Locale类来表示不同的国家，ResourceBundle类来保存和获取不同Locale(国家)的信息
   2. Spring提供MessageSource接口统一了国际化信息的访问方式，不用先根据Locale取得ResourceBundle，然后再从ResourceBundle查询信息了。
   3. ApplicationContext接口继承了MessageSource接口
4. 容器内部事件发布
   1. JAVA自带：EventObject接口、EventListener接口和EventPublisher接口来完成事件定义、事件监听、事件发布
   2. Spring也提供了相应的相应的抽象和接口继承与JAVA自带的事件接口
   3. ApplicationContext也继承于了ApplicationEventPublisher接口
   4. 

由此可见，ApplicationContext通过继承众多不同功能的接口来实现IoC容器的不同功能

# Bean

## Bean介绍

1. 所有注册到容器中需要管理的业务对象，在Spring中统称为Bean
2. Bean常见注册方法有三种：源代码注册、XML配置文件注册、注解注册
3. 本章主要介绍XML文件中的Bean定义

## <bean></bean>

在XML文件中，每个业务对象都由一个<bean>元素描述

### <bean>的基本属性

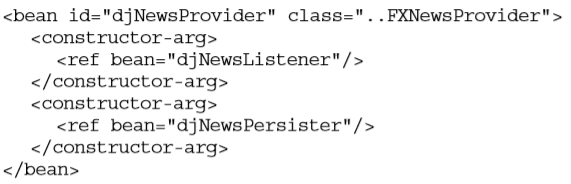


1. id : 唯一标识
2. name：别名，可以使用id中不允许使用的字符
3. class：<bean>对象的类型
4. depends-on：显示声明该Bean依赖于其他Bean，初始化时需要先初始化依赖Bean
5. autowire：设置自动配置依赖关系的模式，默认是no，需要用户自己手动配置Bean之间的依赖关系
6. lazy-init：延迟初始化，只有用到该对象时再初始化Bean

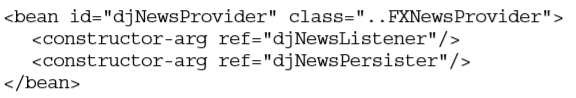
### <bean>对象的参数注入方式

容器在实例化<bean>对象时需要为对象内部的一些属性进行赋值(注入)，赋值的方式有两种：在构造方法中赋值、在setter方法中赋值。因此，XML中<bean>的赋值也有两种方式：

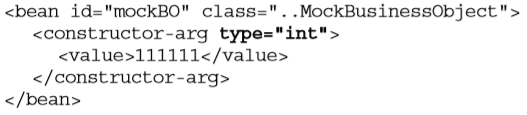
1. 构造注入<constructor-arg >
   1. 依次注入两个Bean对象，通过唯一id确定



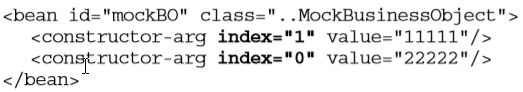
* 1. 可以简写为



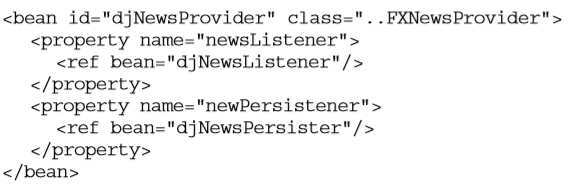
* 1. 可以通过type属性指定注入对象的数据类型



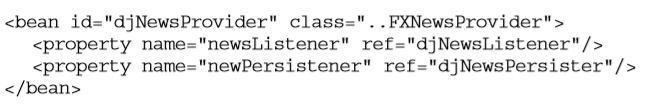
* 1. 可以通过index属性指定注入对象对应的形参位置



1. Setter注入<property >
   1. <property name=“形参名”> 赋值</property>

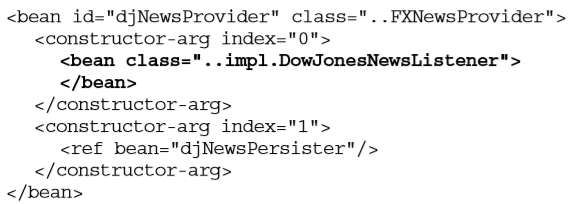


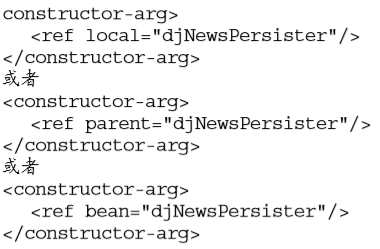
* 1. 简写



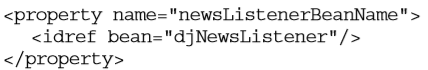
### <bean>对象的参数注入类型

注入方式有两种，每种方式都可以注入不同类型的数据对象，包括<bean>、<ref>、<idref>、<value>、<null>、<list>、<set>、<map>等

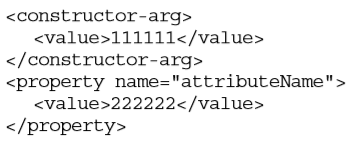
1. 内部<bean>
   1. 如果一个Bean只被本Bean依赖，可以定义成本Bean对象的内部Bean，其他Bean无法访问
   2. 
2. <ref>
   1. 通过Bean对象的id引用容器中注入的其他Bean对象
   2. local:引用本文件中定义的Bean对象；parent：引用父容器中定义的Bean对象；bean:引用所有已定义的Bean对象



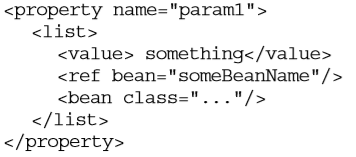
1. <idref>
   1. 注入的是Bean对象的名称，而不是对象本身



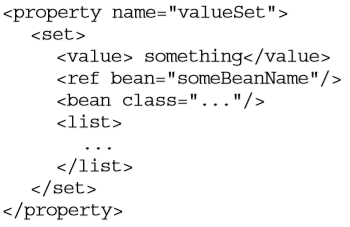
1. <value>
   1. 注入的是Java基本数据类型



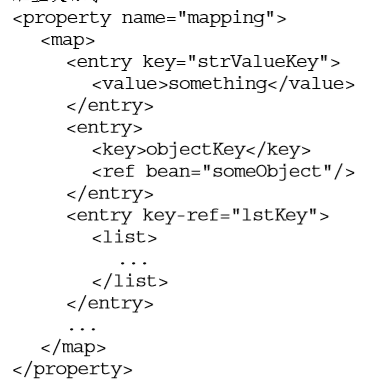
1. <null>
   1. 注入空值，像<value></value>注入的是空字符串“”,不是空值null
2. <list>
   1. 注入一个列表，列表中每个元素都可以是上述介绍的几种基本类型



1. <set>
   1. 与<list>相同，不过是无序的

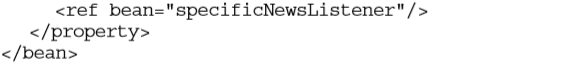
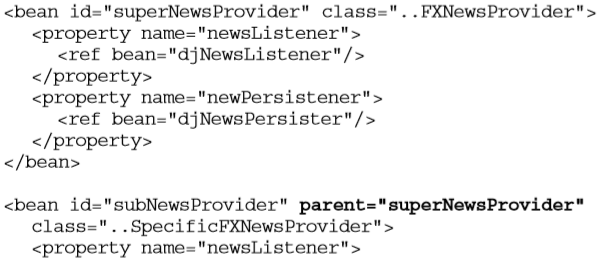


1. <map>
   1. 每个元素都是一个<entry>，拥有key和value



### <bean>的继承

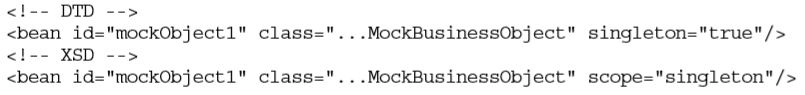
多个<bean>的注入配置重复度较高时，可以使用**parent**属性进行继承，对于继承来的注入配置也可以重写。



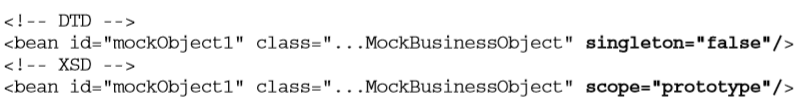
### <bean>的生命周期 scope

<bean>通过scope属性定义其存活时间，常见的取值有：singleton、prototype、request、session、global session

1. Singleton
   1. 全局共享对象，整个容器只存在一个该Bean对象实例，所有依赖都引用这一单一实例，存活时间一般与容器存活时间相同。



1. Prototype
   1. 容器在接到该类型对象的请求的时候，会每次都重新生成一个新的对象实例给请求方，请求方需要自己负责当前返回对象的后继生命周期的管理工作

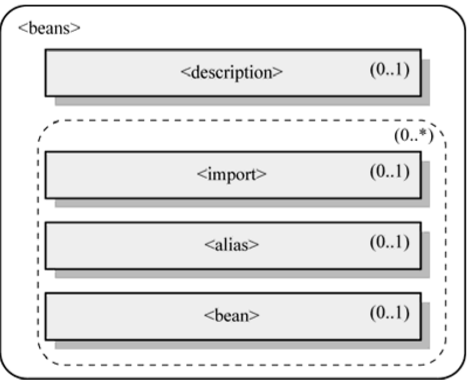


1. request、session、global session
   1. 和prototype类型相同，可以看作prototype的一种特例，只不过应用场景不同





## <beans>



1. 由一组<bean>组成，可以定义这些<bean>共有的属性，每个<bean>再自己定义自己的私有属性，包括：default-lazy-init、default-autowire、default-dependency-check等
2. <description>：整组<beans>的描述信息
3. <import>：导入其他文件定义的<bean>
4. <alias>：为<bean>取别名，和name属性作用相同

## FactoryBean

1. 为工厂模式设计的一种特殊的Bean，该类提供了一个factory method用于返回一个其他类的对象实例
2. 引用FactoryBean的Bean，所依赖的并不是FactoryBean的对象实例，而是FactoryBean中factory method生成的对象实例。比如Bean foo所依赖的是getInstance()方法所返回的对象实例



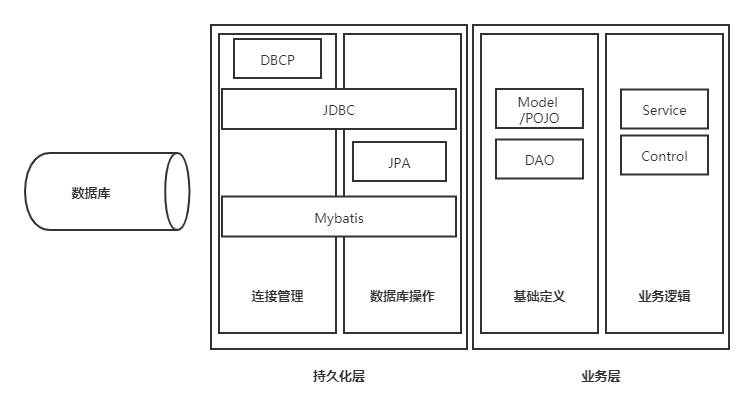
# Spring中的一些概念

## Aware

1. 本身是察觉、注意的意思，Aware接口是为了提醒实现类要实现某些东西的功能
2. 比如BeanNameAware，实现类要提供获取BeanName的方法；BeanFactoryAware，实现类要提供获取BeanFactory的方法等

# Spring 数据访问

Spring中数据库访问的各层次划分如下：



1. 持久化层主要包含两部分功能，一是与数据库之间连接的创建和管理，一个是封装具体的数据库操作方法
   1. DBCP是一个数据库连接池，负责第一部分功能
   2. JDBC是JAVA提供的数据库连接与操作库，两部分功能都有
   3. JPA（Java Persistence API，Java持久化API），定义了对象关系映射(ORM)以及实体对象持久化标准接口
   4. Mybatis是一种Java持久化层框架，封装了数据库连接与操作映射
2. 业务层主要由编程者实现，包括基础定义类和业务逻辑类
   1. Model/POJO层，定义POJO类，对应数据库中具体的表
   2. DAO层，封装了具体表的数据库操作
   3. Service层，通过调用DAO层对象，实现具体的业务逻辑
   4. Control层，调用Service层对象处理不同的请求

## 数据源

介绍：

1. 数据源保证了应用程序与目标数据之间交互的规范和协议，目标数据可以是数据库，文件系统等等。
2. 数据源定义了位置信息，用户验证信息和交互时所需的一些特性的配置，同时它封装了如何建立与数据源的连接，向外暴露获取连接的接口。
3. 数据源把传统中需要在代码里编写配置信息和获取连接等操作抽象出一个规范或者接口，这样不同的第三方可以自行实现该接口提供不同的策略。
4. 应用程序连接数据库无需关注其底层是如何如何建立的，也就是说应用业务逻辑与连接数据库操作是松耦合的。
5. 当需要更换数据库服务器或者更换数据库种类时，只需修改配置中的数据源信息即可，无需修改程序代码

数据源的种类：

1. 简单的DataSource
   1. 只提供ConnectionFactory的基本功能
2. 拥有连接缓冲池的DataSource
3. 支持分布式事务的DataSource

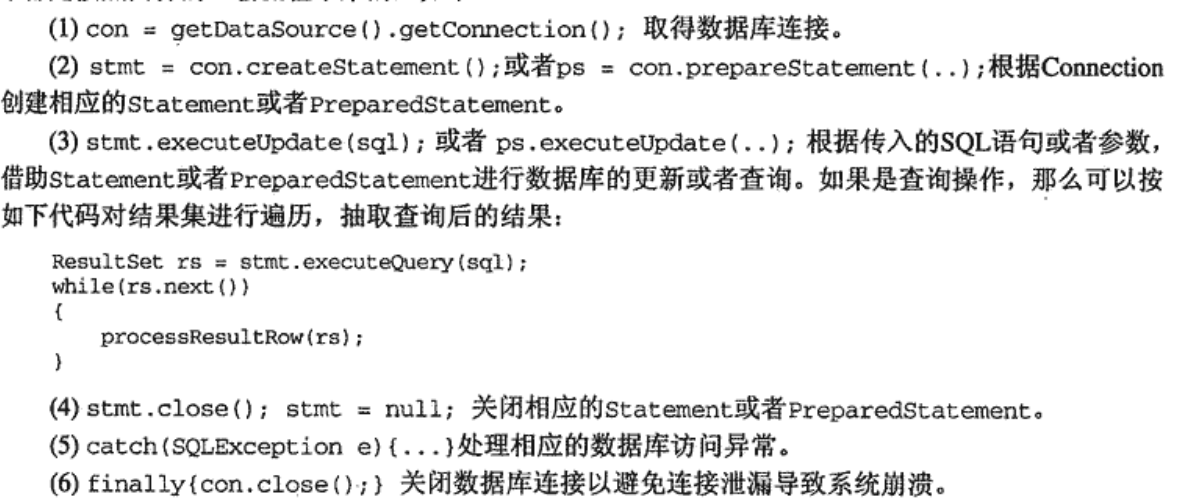
## JDBC

### JDBC介绍

1. Java平台访问关系数据库的标准API
2. 封装了数据库连接、执行、关闭等API
3. 封装了各种DAO异常，并自动将各种不同数据库的异常转译到封装好的DAO异常

### JDBC的缺点

所有访问数据库的DAO代码中，都得重复实现以下功能：



当DAO代码较多时，代码的冗余度会非常大，而且部分开发人员可能会遗忘实现部分功能，比如关闭连接，导致性能下降。

### jdbcTemplate

1. 为了解决上述问题，通过模板模式，将上述重复的DAO操作代码进行统一的封装
2. jdbcTemplate对外提供了各种SQL语句执行接口，便于开发者执行各种SQL语句

## DBCP

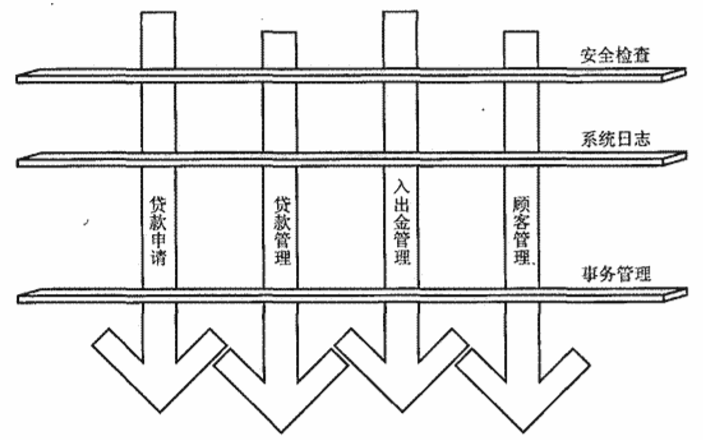
1. DataBase Connection Pool，数据库连接池，一种常用的第三方数据源
2. 数据库打开连接非常费时，而且为每一个用户打开一个连接对于网络应用来说是不现实的，因此开发者希望能够让所有用户共享一个“连接池”，内存中保存一定的数据库连接，当用户请求的时候就分配一个给他，当用户不需要的时候就将连接放回池中。

## JPA

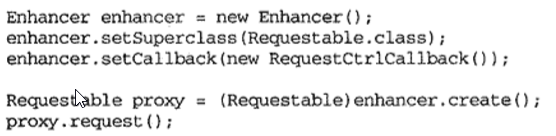
1. JPA（Java Persistence API，Java持久化API），定义了对象关系映射(ORM)以及实体对象持久化标准接口
2. 一般依赖于Hibernate实现。

# AOP

## 理论

1. 类似于代理类，当我们需要为各种不同的需求增加相似的新的需求，使用OOP编程方法，就需要修改每个原需求的所有类；而使用AOP编程方法，只需要修改代理类即可。
   1. 
2. 这种新的需求像横切面一样插入到众多原有需求中，所以叫做面向切面编程AOP，实现AOP这种思想的编程语言叫做AOL
3. JAVA平台的AOP实现机制
   1. 动态代理：基于反射，只对有接口的被代理类有效，动态编译，性能差一些
   2. 动态字节码增强：使用工具再运行期间，动态生成原有类的子类，子类中实现了新的需求，真正执行的是这些子类
   3. 自定义类加载器：在加载解析class文件时将横切逻辑织入
   4. 使用AOL语言编7A0B

## Spring AOP

1. 采用动态代理和动态字节码生成的方式，为目标对象创建一个代理对象，最终执行的是代理对象
   1. 当委托类有继承接口时，就使用动态代理，通过InvocationHandler实现横切逻辑，通过Proxy.newProxyInstance(委托类类加载器，委托类接口，横切逻辑处理对象)创建代理类
   2. 否则，使用CGLIB(Code Generation Library)的开源动态字节码生成类库生成代理对象实例
2. 动态字节码方法与动态代理相似，先通过proxy.Callback接口的实现类定义横切逻辑，然后使用CGLIB的Enhancer类的对象实例设置委托类(父类)和横切逻辑类(子类)，然后生成委托类的对象，不过该对象是一个子类对象，插入了横切逻辑
   1. 

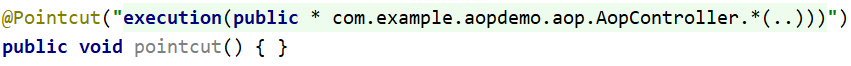
## Spring Boot基于注解的AOP编程

### 实现流程

1. 引入依赖
   1. spring-boot-starter-aop
2. 定义切面类，即代理类，该代理类可以通过注解实现多种AOP编程
   1. 切面类需要使用@Aspect和@Component进行注解
   2. 定义切入点，即指明哪些方法会被代理，常用实现方式有两种
      1. 使用@annotation指定自定义的注解，所有使用该注解的方法都会被代理

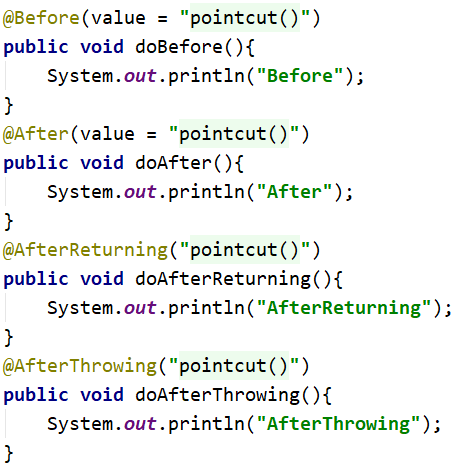
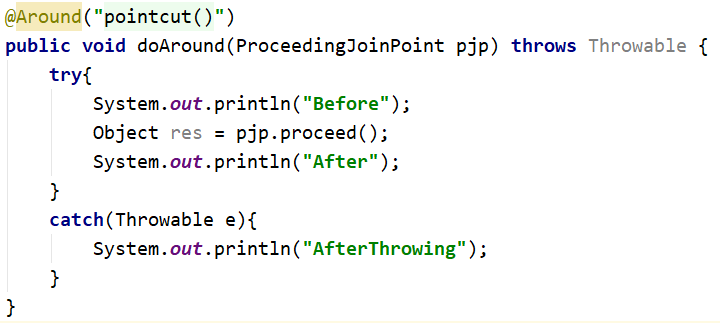


* + 1. 直接使用execution指明被代理方法



* 1. 定义切面方法，即被代理方法执行前后添加的额外操作
  2. 一个类内允许定义多个不同的切入点方法，每个切入点处理不同的被代理方法
  3. 每个切面方法也只针对某个切入点方法

### 切面方法

1. 使用注解声明切面方法
2. 常用5种切面方法
   1. @Before 前置通知，在被代理方法前调用
   2. @Around 环绕通知，将覆盖原有的被代理方法，在处理过程中可以通过反射调用原来的被代理方法
   3. @After 后置通知，在被代理方法后调用
   4. @AfterReturning 返回通知，在被代理方法执行并正常返回后调用，要求连接点方法在执行过程中没有发生异常
   5. @AfterThrowing 异常通知，当被代理方法异常时调用
3. @Around不影响其他切面方法的执行，可以理解原始被代理方法先被@Around修改之后再被其他切面方法处理
4. @Before、@After、@AfterReturning、@AfterThrowing
   1. 
   2. 注解参数指明对应的切入点的方法
5. @Around
   1. 
   2. 注解参数指明对应的切入点的方法
   3. 方法参数是ProceedingJoinPoint对象，通过调用proceed()方法执行被代理方法