POJO：一个简单的Java类，这个类没有实现/继承任何特殊的java接口或者类，不遵循任何主要java模型，约定或者框架的java对象。在理想情况下，POJO不应该有注解。

JavaBean：

- JavaBean是可序列化的，实现了serializable接口
- 具有一个无参构造器
- 有按照命名规范的setter和getter，is（可以用于访问布尔类型的属性）方法

ORM：对象关系映射

# 一、IOC

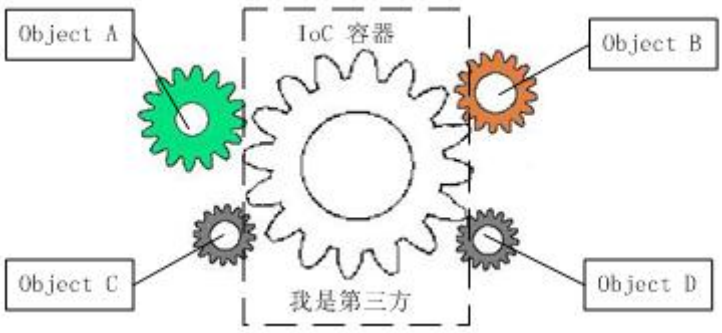
https://www.cnblogs.com/wang-meng/p/5597490.html

随着工业级应用的规模越来越庞大，对象之间的依赖关系也越来越复杂，经常会出现对象之间的多重依赖性关系，因此，架构师和设计师对于系统的分析和设计，将面临更大的挑战。对象之间耦合度过高的系统，必然会出现牵一发而动全身的情形。

耦合关系不仅会出现在对象与对象之间，也会出现在软件系统的各模块之间，以及软件系统和硬件系统之间。如何降低系统之间、模块之间和对象之间的耦合度，是软件工程永远追求的目标之一。**为了解决对象之间的耦合度过高的问题**，软件专家Michael Mattson提出了IOC理论，用来实现对象之间的“解耦”，目前这个理论已经被成功地应用到实践当中，很多的J2EE项目均采用了IOC框架产品Spring。

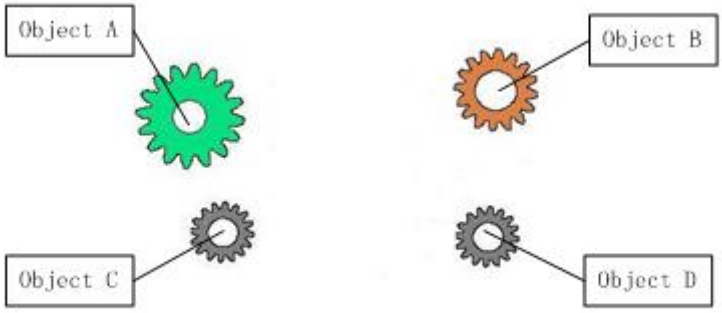
## 1.什么是控制反转IOC

IOC是Inversion of Control的缩写，多数书籍翻译成“控制反转”，还有些书籍翻译成为“控制反向”或者“控制倒置”。简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦，如下图：




由于引进了中间位置的“第三方”，也就是IOC容器，使得A、B、C、D这4个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去

联系，这就是有人把IOC容器比喻成“粘合剂”的由来。




上图就是拿掉IOC容器之后的系统。

获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖注入（Dependency Injection）”。他的这个答案，实际上给出了实现IOC的方法：注入。所谓依赖注入，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。

所以，依赖注入(DI)和控制反转(IOC)是从不同的角度的描述的同一件事情，就是指**通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦**。

对象A依赖于对象B,当对象 A需要用到对象B的时候，IOC容器就会立即创建一个对象B送给对象A。IOC容器就是一个对象制造工厂，你需要什么，它会给你送去，你直接使用就行了，而再也不用去关心你所用的东西是如何制成的，也不用关心最后是怎么被销毁的，这一切全部由IOC容器包办。 在传统的实现中，由程序内部代码来控制组件之间的关系。我们经常使用new关键字来实现两个组件之间关系的组合，这种实现方式会造成组件之间耦合。IOC很好地解决了该问题，它将实现组件间关系从程序内部提到外部容器，也就是说由容器在运行期将组件间的某种依赖关系动态注入组件中。

**IOC带来的好处：**

- 在软件工程中，就是可维护性比较好，非常便于进行单元测试，便于调试程序和诊断故障。代码中的每一个Class都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。
- 在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。
- 可复用性好，我们可以把具有普遍性的常用组件独立出来，反复利用到项目中的其它部分，或者是其它项目，当然这也是面向对象的基本特征。
- 模块具有热插拔特性，只要修改配置文件就可以了，完全具有热插拔的特性。

**我们可以把IOC容器的工作模式看做是工厂模式的升华，可以把IOC容器看作是一个工厂，这个工厂里要生产的对象都在配置文件中给出定义，然后利用编程语言的的反射编程，根据配置文件中给出的类名生成相应的对象。从实现来看，IOC是把以前在工厂方法里写死的对象生成代码，改变为由配置文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。**

**IOC的缺点：**

- 引入新框架需要学习成本
- 由于IOC容器生成对象是通过反射方式，在运行效率上有一定的损耗。如果你要追求运行效率的话，就必须对此进行权衡。
- 具体到IOC框架产品(比如：Spring)来讲，需要进行大量的配制工作，比较繁琐，对于一些小的项目而言，客观上也可能加大一些工作成本。

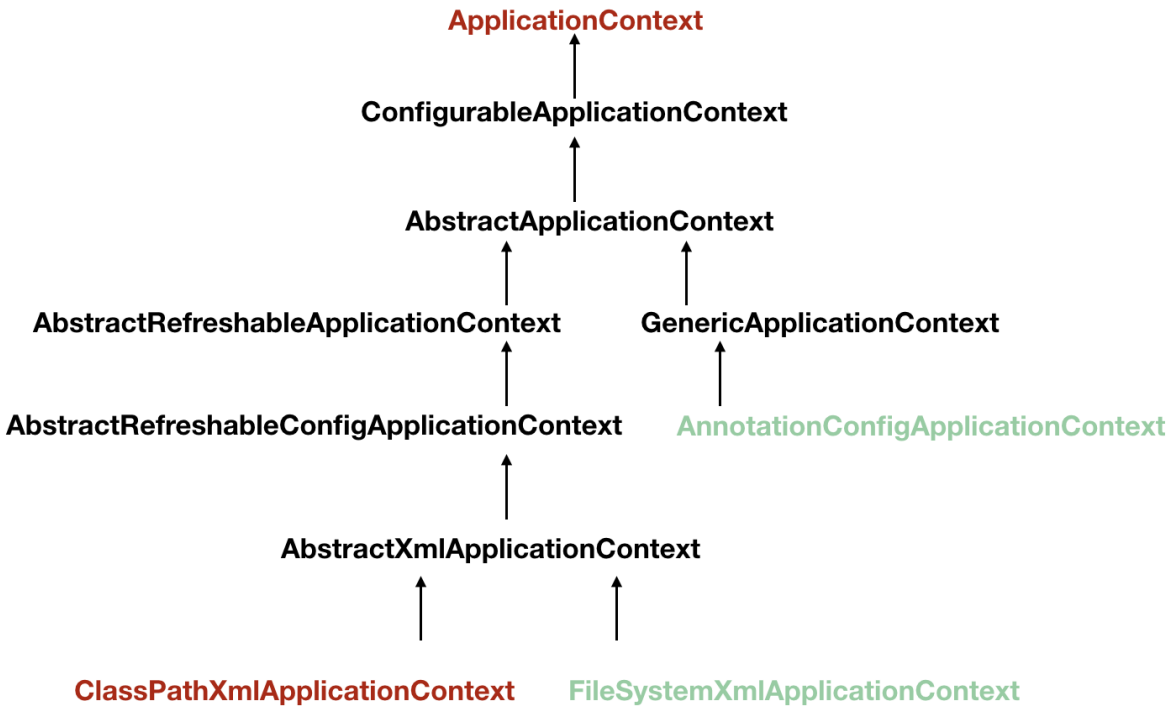
- IOC框架产品本身的成熟度需要进行评估，如果引入一个不成熟的IOC框架产品，那么会影响到整个项目，所以这也是一个隐性的风险。

# 2.IOC容器

首先看一下如何启动Spring容器：

```
1 public static void main(String[] args) {
2     ApplicationContext context = new
  ClassPathXmlApplicationContext("classpath:applicationfile.xml");
3 }
```

`ApplicationContext context = new ClassPathXmlApplicationContext(...)` 其实很好理解，从名字上就可以猜出一二，就是在 ClassPath 中寻找 xml 配置文件，根据 xml 文件内容来构建 ApplicationContext。当然，除了 ClassPathXmlApplicationContext 以外，我们也还有其他构建 ApplicationContext 的方案可供选择，我们先来看看大体的继承结构是怎么样的：




我们可以看到，ClassPathXmlApplicationContext 兜兜转转了好久才到 ApplicationContext 接口，同样的，我们也可以使用绿颜色的 **FileSystemXmlApplicationContext** 和 **AnnotationConfigApplicationContext** 这两个类。

**FileSystemXmlApplicationContext** 的构造函数需要一个 xml 配置文件在系统中的路径，其他和 ClassPathXmlApplicationContext 基本上一样。

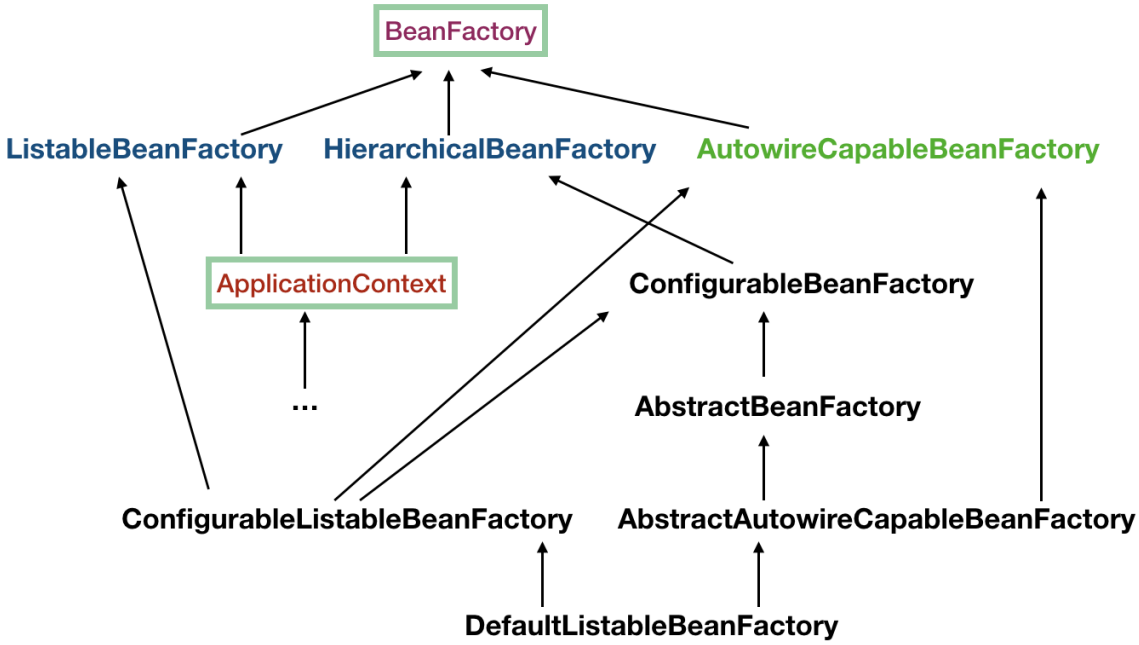
**AnnotationConfigApplicationContext** 是基于注解来使用的，它不需要配置文件，采用 java 配置类和各种注解来配置，是比较简单的方式，也是大势所趋吧。

ApplicationContext 启动过程中，会负责创建实例 Bean，往各个 Bean 中注入依赖等。

## （1）BeanFactory 简介

BeanFactory，从名字上也很好理解，生产 bean 的工厂，它负责生产和管理各个 bean 实例。

初学者可别以为我之前说那么多和 BeanFactory 无关，前面说的 ApplicationContext 其实就是一个 BeanFactory。我们来看下和 BeanFactory 接口相关的主要的继承结构：




依赖注入（DI）

Spring 最认同的技术是控制反转的依赖注入（DI）模式。控制反转（IoC）是一个通用的概念，它可以用许多不同的方式去表达，依赖注入仅仅是控制反转的一个具体的例子。

当编写一个复杂的 Java 应用程序时，应用程序类应该尽可能的独立于其他的 Java 类来增加这些类可重用可能性，当进行单元测试时，可以使它们独立于其他类进行测试。依赖注入（或者有时被称为配线）有助于将这些类粘合在一起，并且在同一时间让它们保持独立。

到底什么是依赖注入？让我们将这两个词分开来看一看。这里将依赖关系部分转化为两个类之间的关联。例如，类 A 依赖于类 B。现在，让我们看一看第二部分，注入。所有这一切都意味着类 B 将通过 IoC 被注入到类 A 中。

依赖注入可以以向构造函数传递参数的方式发生，或者通过使用 setter 方法 post-construction。

IOC：控制反转，为了降低程序对象之间的耦合度，spring把对象的创建和维护的控制权，由应用程序转移到外部第三方工厂或spring容器中，这样控制权的转移，称为控制反转。

又称为DI（依赖注入）,spring容器启动时，spring会根据依赖对象之间的关系，将依赖对象注入到组件中

DI能注入什么数据类型：

- 字面量（基本数据类型和字符串）
- null或空字符串

- 引用类型
- 数组、集合（list set map）

**spring把一切可以实例化的类 称为bean**

使用spring ioc降低对象之间的耦合度，方式有四种：

1）设值注入：通过setter方法注入依赖对象

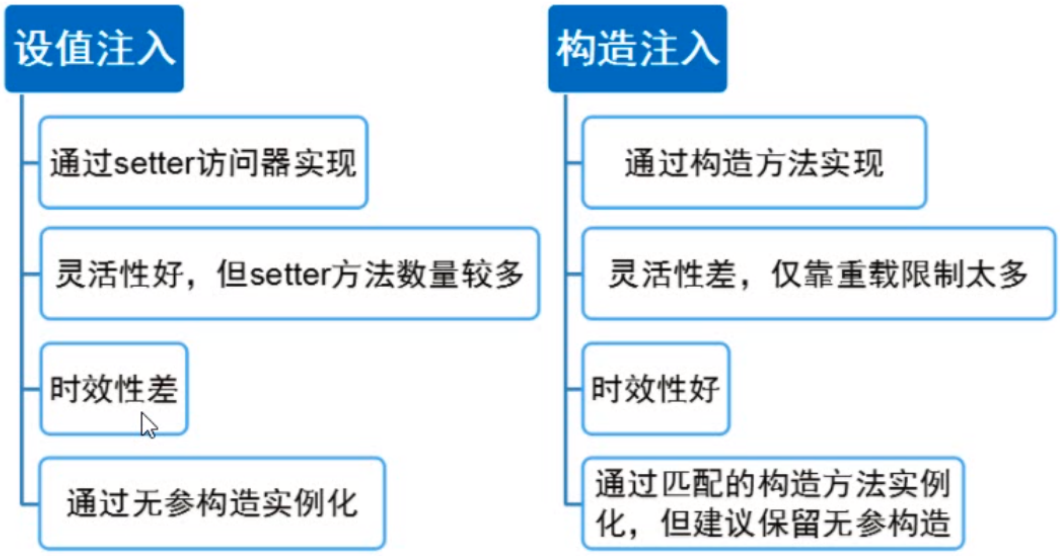
- 提供待注入属性的setter方法
- 在容器中使用property进行设置注入

```
<bean id="userDaos" class="com.dao.impl.UserDaoImpl">
</bean>
<bean id="userService"
class="com.service.impl.UserServiceImpl">
    <!-- 依赖的属性名，就是类中的属性 -->
    <property name="userDao" ref = "userDaos" />
    <property name="name" value = "姓名" />
    <property name="array">
        <array value-type="java.lang.String">
            <value>ka1</value>
            <value>ka2</value>
            <value>ka3</value>
        </array>
    </property>
</bean>
```

2）构造注入：通过构造函数注入属性

- 提供 依赖属性 的构造函数和无参构造函数
- 在spring容器中（xml中）

```
<bean id="userDaos" class="com.dao.impl.UserDaoImpl">
</bean>
<bean id="userService"
class="com.service.impl.UserServiceImpl">
    <!-- constructor-arg 对应一个构造函数的一个参数 -->
    <!-- index表示参数顺序，同样也可以通过 name 来指定 -->
    <constructor-arg index ="0" ref="product" />
</bean>
```




3）p命名空间注入：通过p命名空间属性 进行依赖对象注入。能够减少配置量，需要属性提供setter方法；无法处理特殊字符。

- 引入p命名空间

  在xml配置文件中引入p命名空间，xmlns:p="http://www.springframework.org/schema/p"

  p命名空间的特点：使用 **属性** 而不是子元素的形式配置Bean的属性，从而简化了配置代码。语法为：

  > *对于直接量（基本数据类型、字符串）属性： p:属性名="属性值"*
  >
  > *对于引用Bean的属性：p:属性名-ref="Bean的id"*

- 使用p命名空间注入属性值：

  >

4）ioc注解注入

# 二、AOP

Spring 框架的一个关键组件是**面向方面的程序设计（AOP）**框架。一个程序中跨越多个点的功能被称为**横切关注点**（面向切面编程），这些横切关注点在概念上独立于应用程序的业务逻辑。有各种各样常见的很好的关于方面的例子，比如日志记录、声明性事务、安全性，和缓存等等。

在 OOP 中模块化的关键单元是类，而在 AOP 中模块化的关键单元是方面。AOP 帮助你将横切关注点从它们所影响的对象中分离出来，然而依赖注入帮助你将你的应用程序对象从彼此中分离出来。

Spring 框架的 AOP 模块提供了面向方面的程序设计实现，可以定义诸如方法拦截器和切入点等，从而使实现功能的代码彻底的解耦出来。

切入点选择时机：

| 通知 | 描述 |
| --- | --- |
| 前置通知aop:before | 在一个方法执行之前，执行通知。 |
| 后置通知aop:after | 在一个方法执行之后，不考虑其结果，执行通知。 |
| 返回后通知after-returning | 在一个方法执行之后，只有在方法成功完成时，才能执行通知。 |
| 抛出异常后通知after-throwing | 在一个方法执行之后，只有在方法退出抛出异常时，才能执行通知。 |
| 环绕通知aop:around | 在建议方法调用之前和之后，执行通知。 |

使用xml进行配置：

例： **execution (* com.sample.service..*. *(..))**

整个表达式可以分为五个部分：

1、execution()::表达式主体。

2、第一个*号：表示返回类型， *号表示所有的类型。

3、包名：表示需要拦截的包名，后面的两个句点表示当前包和当前包的所有子包，com.sample.service包、子孙包下所有类的方法。

4、第二个*号：表示类名，*号表示所有的类。

5、*(..)：最后这个星号表示方法名，*号表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数

# 三、Spring事务

https://juejin.im/post/5b00c52ef265da0b95276091

## 1.Spring事务管理接口：

- PlatformTransactionManager：（平台）事务管理器
- TransactionDefinition：事务定义信息（事务隔离级别、传播行为、超时、只读、回滚规则）
- TransactionStatus：事务运行状态

所谓事务管理，其实就是“按照给定的事务规则来执行提交或者回滚操作”。

### （1）PlatformTransactionManager

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。 Spring事务管理器的接口是： **org.springframework.transaction.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。

PlatformTransactionManager接口代码如下：

```
Public interface PlatformTransactionManager()...{
    // Return a currently active transaction or create a new one,
    // according to the specified propagation behavior
    // 根据指定的传播行为，返回当前活动的事务或创建一个新事务。
    TransactionStatus getTransaction(TransactionDefinition
definition) throws TransactionException;
    // Commit the given transaction, with regard to its status
    // 使用事务目前的状态提交事务
    Void commit(TransactionStatus status) throws
TransactionException;
    // Perform a rollback of the given transaction
    // 对执行的事务进行回滚
    Void rollback(TransactionStatus status) throws
TransactionException;
}
```

Spring中PlatformTransactionManager根据不同持久层框架所对应的接口实现类几个比较常见的如下图所示：

| 事务 | 说明 |
| --- | --- |
| or.springframework.jdbc.datasource.DataSourceTransactionManager | 使用Spri JDBC或iI 进行持久 据时使用 |
| or.springframework.orm.hibernate3.HibernateTransactionManager | 使用 Hiberna 版本进行 化数据时 |
| or.springframework.orm.jpa.JpaTransactionManager | 使用JPA 数据持久 使用 |
| or.springframework.transaction.jta.JtaTransactionManager | 使用一个 实现来管 务，在一 务跨越多 源时使用 |

比如我们在使用JDBC或iBatis（就是Mybatis）进行数据持久化操作时，我们的xml配置通常如下：

```xml
<!-- 事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionMana
ger">
    <!-- 数据源 -->
    <property name="dataSource" ref="dataSource" />
</bean>
```

druid和jdbc连接方式相同，配置数据源即可

## （2）TransactionDefinition接口介绍

事务管理器接口PlatformTransactionManager通过getTransaction(TransactionDefinition definition)方法来得到一个事务，这个方法里面的参数是TransactionDefinition类，这个类就定义了一些基本的事务属性。

那么什么是事务属性呢?

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了5个方面。


TransactionDefinition接口中的方法如下:

TransactionDefinition接口定义了5个方法以及一些事务属性的常量比如隔离级别、传播行为等等的常量。

```java
public interface TransactionDefinition {
    // 返回事务的传播行为
    int getPropagationBehavior();
    // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
    int getIsolationLevel();
    // 返回事务的名字
    String getName();
    // 返回事务必须在多少秒内完成
    int getTimeout();
    // 返回是否优化为只读事务。
    boolean isReadOnly();
}
```

### 1）事务隔离级别（定义了一个事务可能受其它并发事务影响的程度）：

并发事务带来的问题：脏读、丢失修改、不可重复读、幻读

**TransactionDefinition**接口定义了五个表示隔离级别的常量：

- TransactionDefinition.ISOLATION_DEFAULT：使用数据库默认的隔离级别，MySQL默认采用REPEATABLE_READ隔离级别，Oracle默认采用的READ_COMMITTED隔离级别。
- TransactionDefinition.ISOLATION_READ_UNCOMMITED：最低隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、不可重复读或幻读。
- **TransactionDefinition.ISOLATION_READ_COMMITTED:** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- **TransactionDefinition.ISOLATION_REPEATABLE_READ:** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
- **TransactionDefinition.ISOLATION_SERIALIZABLE:** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

### 2）事务传播行为（为了解决业务层方法之间互相调用的事务问题）：

当事务方法被另一个事务方法调用时，必须制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。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRED**： 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- **TransactionDefinition.PROPAGATION_SUPPORTS**： 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- **TransactionDefinition.PROPAGATION_MANDATORY**： 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

不支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRES_NEW**： 创建一个新的事务，如果当前存在事务，则把当前事务挂起。

- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED**：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NEVER**：以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

- **TransactionDefinition.PROPAGATION_NESTED**：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION_REQUIRED。

这里需要指出的是，前面的六种事务传播行为是 Spring 从 EJB 中引入的，他们共享相同的概念。而 **PROPAGATION_NESTED** 是 Spring 所特有的。以 PROPAGATION_NESTED 启动的事务内嵌于外部事务中（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。如果熟悉 JDBC 中的保存点（SavePoint）的概念，那嵌套事务就很容易理解了，其实嵌套的子事务就是保存点的一个应用，一个事务中可以包括多个保存点，每一个嵌套子事务。另外，外部事务的回滚也会导致嵌套子事务的回滚。

### 3）事务超时属性（一个事务允许执行的最长时间）

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

### 4）事务只读属性（对事务资源是否执行只读操作）

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。所谓事务性资源就是指那些被事务管理的资源，比如数据源、 JMS 资源，以及自定义的事务性资源等等。如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。在 TransactionDefinition 中以 boolean 类型来表示该事务是否只读。

### 5）回滚规则（定义事务回滚规则）

这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

## （3）TransactionStatus接口介绍

TransactionStatus接口用来记录事务的状态 该接口定义了一组方法,用来获取或判断事务的相应状态信息。

PlatformTransactionManager.getTransaction(...) 方法返回一个 TransactionStatus 对象。返回的TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务）。

**TransactionStatus**接口接口内容如下：

```java
public interface TransactionStatus{
    boolean isNewTransaction(); // 是否是新的事物
    boolean hasSavepoint(); // 是否有恢复点
    void setRollbackOnly();  // 设置为只回滚
    boolean isRollbackOnly(); // 是否为只回滚
    boolean isCompleted; // 是否已完成
}
```

# 2.Spring编程式和声明式事务实例讲解

## （1）Spring事务管理

**Spring**支持两种方式的事务管理：

- 编程式事务管理：通过Transaction Template手动管理事务，在实际应用中很少使用
- 使用XML配置声明式事务：推荐使用（代码侵入性最小），实际是通过AOP实现

实现声明式事务的四种方式：

- 基于 **TransactionInterceptor** 的声明式事务: Spring 声明式事务的基础，通常也不建议使用这种方式，但是与前面一样，了解这种方式对理解 Spring 声明式事务有很大作用。
- 基于 **TransactionProxyFactoryBean** 的声明式事务: 第一种方式的改进版本，简化的配置文件的书写，这是 Spring 早期推荐的声明式事务管理方式，但是在 Spring 2.0 中已经不推荐了。
- 基于**< tx>** 和**< aop>**命名空间的声明式事务管理： 目前推荐的方式，其最大特点是与 Spring AOP 结合紧密，可以充分利用切点表达式的强大支持，使得管理事务更加灵活。
- 基于 **@Transactional** 的全注解方式： 将声明式事务管理简化到了极致。开发人员只需在配置文件中加上一行启用相关后处理 Bean 的配置，然后在需要实施事务管理的方法或者类上使用 @Transactional 指定事务规则即可实现事务管理，而且功能也不必其他方式逊色。

我们今天要将的是使用编程式以及基于**AspectJ**的声明式和基于注解的事务方式，实现烂大街的转账业务。

再来说一下这个案例的思想吧，我们在两次转账之间添加一个错误语句（对应银行断电等意外情况），如果这个时候两次转账不能成功，则说明事务配置正确，否则，事务配置不正确。

> *你需要完成的任务：*
>
> - *使用编程式事务管理完成转账业务*
> - *使用基于AspectJ的声明式事务管理完成转账业务*
> - *使用基于@Transactional 的全注解方式事务管理完成转账业务*

## （2）编程式事务管理

注意： 通过添加/删除accountMoney() 方法中int i = 10 / 0这个语句便可验证事务管理是否配置正确。核心代码如下：

```java
public void accountMoney() {
```

```
    transactionTemplate.execute(new TransactionCallback<Object>() {
        @Override
        public Object doInTransaction(TransactionStatus status) {
            Object result = null;
            try {
                // 小马多1000
                ordersDao.addMoney();
                // 加入出现异常如下面int
                // i=10/0（银行中可能为突然停电等。。。）;
                // 结果：小马账户多了1000而小王账户没有少钱
                // 解决办法是出现异常后进行事务回滚
                System.out.println("已经修改过一个账户了");
                Thread.currentThread().sleep(10000);
                int i = 10 / 0;// 事务管理配置后异常已经解决
                // 小王 少1000
                ordersDao.reduceMoney();
            } catch (Exception e) {
                // 回滚
                status.setRollbackOnly();
                result = false;
                System.out.println("Transfer Error!");
            }
            return result;
        }
    });
}
```

## (3) 基于Aspect J的声明式事务管理

```
    <!-- 第一步：配置事务管理器 -->
    <bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionMan
ager">
        <!-- 注入dataSource -->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- 第二步：配置事务增强 -->
    <tx:advice id="txadvice" transaction-
manager="dataSourceTransactionManager">
        <!-- 做事务操作 -->
        <tx:attributes>
            <!-- 设置进行事务操作的方法匹配规则 -->
            <!-- account开头的所有方法 -->
            <!--
              propagation:事务传播行为;
              isolation: 事务隔离级别;
              read-only: 是否只读;
              rollback-for: 发生那些异常时回滚
              timeout:事务过期时间
             -->
```

```
            <tx:method name="account*" propagation="REQUIRED"
isolation="DEFAULT" read-only="false" rollback-for="" timeout="-1"
/>
        </tx:attributes>
    </tx:advice>
    <!-- 第三步：配置切面 切面即把增强用在方法的过程 -->
    <aop:config>
        <!-- 切入点 -->
        <aop:pointcut expression="execution(*
com.jdbc.aspectj.OrdersService.*(..))" id="pointcut1" />
        <!-- 切面 -->
        <aop:advisor advice-ref="txadvice" pointcut-ref="pointcut1"
/>
    </aop:config>
```

### (4) 基于注解的方式

核心步骤为:

```
<!-- 第一步：配置事务管理器 -->
<bean id="dataSourceTransactionManager"

 class="org.springframework.jdbc.datasource.DataSourceTransactionMan
ager">
    <!-- 注入dataSource -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!-- 第二步：开启事务注解 -->
<tx:annotation-driven transaction-
manager="dataSourceTransactionManager" />
```

为需要使用事务的类添加注解:

```
@Transactional(propagation = Propagation.REQUIRED, isolation =
Isolation.DEFAULT, readOnly = false, timeout = -1)
```

# 四、Mybatis

## 1.mybatis #和$区别

符号“#”:

- 是将传入的值当做字符串的形式，如：select id,name,age from student where name=#{name} -- name='cy'
- 使用#可以很大程度上防止sql注入。(语句的拼接)
- 会将传入的数据当成一个字符串，会自动加上双引号
- 解析为一个 JDBC 预编译语句（prepared statement）的参数标记符，一个 #{ } 被解析为一个参数占位符 。

符号“$”：

- 是将传入的数据直接显示生成sql语句，如：select id,name,age from student where name=${name} -- name=cy;
- 但是如果使用在order by 中就需要使用 $。
- 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换。

# 2.如何防止SQL注入

在Mybatis中最好使用 # 而不用 $ ，避免出现SQL注入。