**debug快捷键**

**关于IDEA的XML文件标签元素变红的原因？**

**数据库范式？**

[数据库设计三大范式](http://www.cnblogs.com/linjiqin/archive/2012/04/01/2428695.html)

为了建立冗余较小、结构合理的数据库，设计数据库时必须遵循一定的规则。在关系型数据库中这种规则就称为范式。范式是符合某一种设计要求的总结。要想设计一个结构合理的关系型数据库，必须满足一定的范式。

在实际开发中最为常见的设计范式有三个：

1．第一范式(确保每列保持原子性)

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，如下表所示。



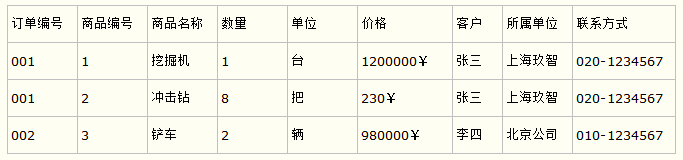
上表所示的用户信息遵循了第一范式的要求，这样在对用户使用城市进行分类的时候就非常方便，也提高了数据库的性能。

2．第二范式(确保表中的每列都和主键相关)

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键，如下表所示。

 订单信息表



这样就产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。

而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。如下所示。



这样设计，在很大程度上减小了数据库的冗余。如果要获取订单的商品信息，使用商品编号到商品信息表中查询即可。

3．第三范式(确保每列都和主键列直接相关,而不是间接相关)

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。



这样在查询订单信息的时候，就可以使用客户编号来引用客户信息表中的记录，也不必在订单信息表中多次输入客户信息的内容，减小了数据冗余。

**Mybaties的批处理？**

**Excel使用？**

**Decimal与bigdecimal?**

**mysql添加索引的原因？**

当某一列被查询比较频繁时，为该列添加索引，索引类型是不同的数据 ，如二叉树btree，这样在查询某一个值时查询速度回因为数据结构的原因而得到优化，如二叉树的查找方式比顺序查找快的多。

**Double类型计算时缺失精度解决方式**

BigDecimal在《Effective Java》这本书中也提到这个原则，float和double只能用来做科学计算或者是工程计算，在商业计算中我们要用 java.math.BigDecimal。BigDecimal一共有4个够造方法，我们不关心用BigInteger来够造的那两个，那么还有两个， 它们是：  
BigDecimal(double val)   
          Translates a double into a BigDecimal.   
BigDecimal(String val)   
          Translates the String repre sentation of a BigDecimal into a BigDecimal.  
上面的API简要描述相当的明确，而且通常情况下，上面的那一个使用起来要方便一些。我们可能想都不想就用上了，会有什么问题呢？等到出了问题的时候，才发现上面哪个够造方法的详细说明中有这么一段：  
Note: the results of this constructor can be somewhat unpredictable. One might assume that new BigDecimal(.1) is exactly equal to .1, but it is actually equal to .1000000000000000055511151231257827021181583404541015625. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed in to the constructor is not exactly equal to .1, appearances nonwithstanding.   
The (String) constructor, on the other hand, is perfectly predictable: new BigDecimal(".1") is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.  
原来我们如果需要精确计算，非要用String来够造BigDecimal不可！

案例：

BigDecimal c1 = new BigDecimal(Double.*toString*(0.02));  
BigDecimal c2 = new BigDecimal(Double.*toString*(0.18));  
System.*out*.println(c1.add(c2).doubleValue());

**Markdown使用？**