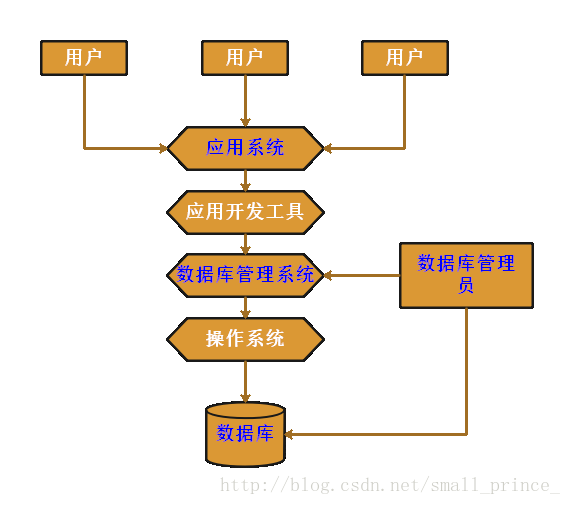
数据库

## 数据库概念

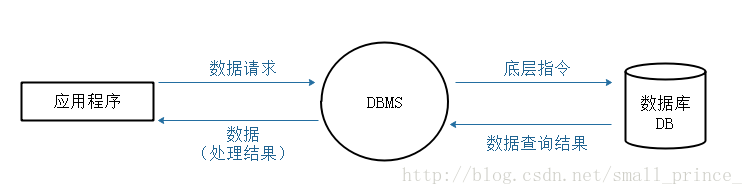
Database：简而言之可视为[电子化](https://baike.baidu.com/item/%E7%94%B5%E5%AD%90%E5%8C%96" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)的[文件柜](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%9F%9C" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)——存储电子[文件](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)的处所，用户可以对文件中的数据进行新增、截取、更新、删除等操作。所谓“数据库”是以一定方式储存在一起、能与多个用户[共享](https://baike.baidu.com/item/%E5%85%B1%E4%BA%AB" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)、具有尽可能小的[冗余度](https://baike.baidu.com/item/%E5%86%97%E4%BD%99%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)、与应用程序彼此独立的数据[集合](https://baike.baidu.com/item/%E9%9B%86%E5%90%88" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)，这也是与文件系统的本质区别。

数据库中的几个术语：数据，数据库（DB），数据库管理系统（DBMS），数据库系统（DBS），数据库管理员（DBA）。



DBMS：位于用户应用程序与操作系统之间的一层数据管理软件，是数据库系统的核心组成部分。为用户或应用程序提供访问数据库的方法：包括数据库的建立、查询、更新以及各种数据控制。

数据库管理系统是为了科学地组织和存储数据、高效地获取和维护数据。



## 数据库类型

关系数据库（sql）：mysql、oracle...

非关系型数据库（nosql）：redis、mongodb...

## 数据库索引

### 概念

[索引](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95" \t "https://baike.baidu.com/item/_blank)（index）是对数据库表中一个或多个列（组合索引）的值进行排序的数据结构。

### 作用

好比是一本书前面的目录，加快数据库的查询速度

### 类型

UNIQUE唯一索引,不可以出现相同的值,可以有NULL值

INDEX普通索引,允许出现相同的索引内容

PRIMARY KEY主键索引

### 缺点

* 在数据库建立过程中，需花费较多的时间去建立并维护索引，特别是随着数据总量的增加，所花费的时间将不断递增。
* 在数据库中创建的索引需要占用一定的物理存储空间，这其中就包括数据表所占的数据空间以及所创建的每一个索引所占用的物理空间，如果有必要建立起聚簇索引，所占用的空间还将进一步的增加
* 在对表中的数据进行修改时，例如对其进行增加、删除或者是修改操作时，索引还需要进行动态的维护，对索引文件进行重新排序或更新。

### 适合加索引的列

* 维度高的列创建索引，如user表中性别列就维度低不适合加索引，即使有很多数据但维度很低，加索引也是全表查询
* 对 where,on,group by,order by 中出现的列使用索引
* 对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
* 对于那些定义为text, image和[bit数据类型](https://baike.baidu.com/item/bit%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B" \t "https://baike.baidu.com/item/_blank)的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。

### 不走索引的sql

* 条件字段选择性弱，查出的结果集较大，不走索引；
* 索引列上有函数运算，导致不走索引，如SELECT \* FROM T WHERE FUN(Y) = XXX
* 隐式转换导致不走索引，如SELECT \* FROM T WHERE Y = 5但是Y列是VARCHAR2会隐式的转换SELECT \* FROM T WHERE TO\_NUMBER(Y) = 5；
* 使用不等于操作符如：<>、!= 等不走索引；
* like 后面的字符当首位为通配符时不走索引；
* 对于count(\*)当索引字段有not null约束时走索引，否则不走索引；

### EXPLAIN

执行计划：sql执行调优，查看执行具体情况，可以看到是否用到了索引。

## 数据库事务

### 概念

事务（Transaction），一般是指要做的或所做的事情。在计算机[术语](https://baike.baidu.com/item/%E6%9C%AF%E8%AF%AD" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)中是指访问并可能更新数据库中各种[数据项](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%A1%B9/3227309" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的一个程序执行单元(unit)。事务通常由[高级数据库](https://baike.baidu.com/item/%E9%AB%98%E7%BA%A7%E6%95%B0%E6%8D%AE%E5%BA%93/1439366" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)操纵语言或编程语言（如SQL，C++或Java）书写的[用户程序](https://baike.baidu.com/item/%E7%94%A8%E6%88%B7%E7%A8%8B%E5%BA%8F/7450916" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的执行所引起，并用形如begin transaction和end transaction语句（或[函数调用](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8/4127405" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)）来界定。事务由事务开始(begin transaction)和事务结束(end transaction)之间执行的全体操作组成。在[关系数据库](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)中，一个事务可以是一条SQL语句，一组SQL语句或整个程序。

### ACID特性

[事务](https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/5945882" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)是恢复和[并发控制](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91%E6%8E%A7%E5%88%B6" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的基本单位。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。

原子性（atomicity）:一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

一致性（consistency）:事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

隔离性（isolation）:一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。在同一个事务中，数据需要保证一致性，也就是说，在当前的事务中，删除了就是看不到了，插入了就是多了一条数据，但这些操作都是在当前事务内部的。对于外界是隔离的。如果我们开启事务插入一条数据，还没有提交，这时新开一个事务，去查询数据，是发现不了未提交数据的

持久性（durability）:持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

### 事务隔离性

当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题

#### 脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money=money+100 where name=’B’; (此时A通知B)

update account set money=money - 100 where name=’A’;

当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

#### 不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

#### 虚读(幻读)

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

MySQL数据库为我们提供的四种隔离级别：

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况

在MySQL数据库中查看当前事务的隔离级别：select @@tx\_isolation;

设置事务的隔离级别：set [glogal | session] transaction isolation level 隔离级别名称;

## 数据库锁

### 概念

锁是网络数据库中的一个非常重要的概念，当多个用户同时对数据库并发操作时，会带来数据不一致的问题，所以，锁主要用于多用户环境下保证数据库完整性和一致性。

### 目的

处理并发问题

### 技术手段

乐观锁、悲观锁和时间戳。

### 锁分类

从数据库系统角度分为三种：排他锁、共享锁、更新锁。

从程序员角度分为两种：一种是悲观锁，一种乐观锁。

### 悲观锁（Pessimistic Lock）

#### 概念

顾名思义，很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人拿这个数据就会block（阻塞），直到它拿锁。

正如其名，具有强烈的独占和排他特性。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

传统的关系数据库里用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在操作之前先上锁。

#### 性质分类

##### 共享锁（Share Lock）

S锁，也叫读锁，用于所有的只读数据操作。共享锁是非独占的，允许多个并发事务读取其锁定的资源。

特点

* 多个事务可封锁同一个共享页；
* 任何事务都不能修改该页；
* 通常是该页被读取完毕，S锁立即被释放。

##### 排他锁（Exclusive Lock）

X锁，也叫写锁，表示对数据进行写操作。如果一个事务对对象加了排他锁，其他事务就不能再给它加任何锁了。（某个顾客把试衣间从里面反锁了，其他顾客想要使用这个试衣间，就只有等待锁从里面打开了。）

特点

* 仅允许一个事务封锁此页；
* 其他任何事务必须等到X锁被释放才能对该页进行访问；
* X锁一直到事务结束才能被释放。

产生排他锁的SQL语句如下：select \* from ad\_plan for update;

##### 更新锁

U锁，在修改操作的初始化阶段用来锁定可能要被修改的资源，这样可以避免使用共享锁造成的死锁现象。

因为当使用共享锁时，修改数据的操作分为两步：

1. 首先获得一个共享锁，读取数据，

2. 然后将共享锁升级为排他锁，再执行修改操作。

这样如果有两个或多个事务同时对一个事务申请了共享锁，在修改数据时，这些事务都要将共享锁升级为排他锁。这时，这些事务都不会释放共享锁，而是一直等待对方释放，这样就造成了死锁。

如果一个数据在修改前直接申请更新锁，在数据修改时再升级为排他锁，就可以避免死锁。

特点

1. 用来预定要对此页施加X锁，它允许其他事务读，但不允许再施加U锁或X锁；

2. 当被读取的页要被更新时，则升级为X锁；

3. U锁一直到事务结束时才能被释放。

#### 范围分类

##### 行锁

行锁，锁的作用范围是行级别。

##### 表锁

锁的作用范围是整张表。

##### 锁的选择

数据库能够确定那些行需要锁的情况下使用行锁，如果不知道会影响哪些行的时候就会使用表锁。

举个例子，一个用户表user，有主键id和用户生日birthday。 当你使用update … where id=?这样的语句时，数据库明确知道会影响哪一行，它就会使用行锁；

当你使用update … where birthday=?这样的的语句时，因为事先不知道会影响哪些行就可能会使用表锁。

### 乐观锁（Optimistic Lock）

#### 概念

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受，而乐观锁是在应用程序实现的锁机制，在一定程度上解决了这个问题。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

#### 实现方式

##### 版本号（version）

版本号（记为version）：就是给数据增加一个版本标识，在数据库上就是表中增加一个version字段，每次更新把这个字段加1，读取数据的时候把version读出来，更新的时候比较version，如果还是开始读取的version就可以更新了，如果现在的version比老的version大，说明有其他事务更新了该数据，并增加了版本号，这时候得到一个无法更新的通知，用户自行根据这个通知来决定怎么处理，比如重新开始一遍。这里的关键是判断version和更新两个动作需要作为一个原子单元执行，否则在你判断可以更新以后正式更新之前有别的事务修改了version，这个时候你再去更新就可能会覆盖前一个事务做的更新，造成第二类丢失更新，所以你可以使用update … where … and version=”old version”这样的语句，根据返回结果是0还是非0来得到通知，如果是0说明更新没有成功，因为version被改了，如果返回非0说明更新成功。

##### 时间戳

时间戳（timestamp）：和版本号基本一样，只是通过时间戳来判断而已，注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间。

##### 待更新字段

待更新字段：和版本号方式相似，只是不增加额外字段，直接使用有效数据字段做版本控制信息，因为有时候我们可能无法改变旧系统的数据库表结构。假设有个待更新字段叫count,先去读取这个count,更新的时候去比较数据库中count的值是不是我期望的值（即开始读的值），如果是就把我修改的count的值更新到该字段，否则更新失败。java的基本类型的原子类型对象如AtomicInteger就是这种思想。

##### 几种方式的区别

新系统设计可以使用version方式和timestamp方式，需要增加字段，应用范围是整条数据，不论那个字段修改都会更新version,也就是说两个事务更新同一条记录的两个不相关字段也是互斥的，不能同步进行。旧系统不能修改数据库表结构的时候使用数据字段作为版本控制信息，不需要新增字段，待更新字段方式只要其他事务修改的字段和当前事务修改的字段没有重叠就可以同步进行，并发性更高。

### 锁机制产生的问题

并发控制会造成活锁和死锁，就像操作系统那样，会因为互相等待而导致。

#### 活锁

##### 定义

活锁的情况：如果事务T1封锁了数据R，事务T2又请求封锁R，于是T2等待；T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，T2仍然等待；然后T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求…T2有可能永远等待。

##### 解决方法

避免活锁的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得所。

#### 死锁

##### 定义

死锁的情形：如果事务T1封锁了数据R1，T2封锁了数据R2，然后T1又请求封锁R2，因T2已封锁了R1，T2也只能等待T1释放R1上的锁。这样就出现了T1在等待T2，而T2又在等待T1的局面，T1和T2两个事务永远不能结束，死锁这种没有完全解决的方法，只能尽量预防。

##### 预防方法

采取一定措施来预防死锁的发生。在数据库中，产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已被其他事务封锁的数据对象加锁，从而出现死等待。防止死锁的发生其实就是要破坏产生死锁的条件

1. 一次封锁法：要求每个事务必须将所有要使用的数据全部加锁，否则就不能继续执行。这种方法虽然可以有效地预防死锁的方式，但是一次就将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度，而且，数据库是不断变化的，原来不要求封锁的数据在执行过程中可能会变成封锁对象，所以很难事先精确地确定每个事务所要封锁的数据对象，为此只能扩大封锁范围，将事务在执行过程中可能要鞥说的数据对象全部加锁，这就进一步降低了并发度。
2. 顺序封锁法：顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实施封锁。例如在B树结构的索引中，可规定封锁的顺序必须是从根节点开始，然后是下一级的子节点，逐级封锁。顺序封锁法可以有效地防止死锁，但是数据库系统中封锁的数据对象极多，并且随数据的插入、删除等操作而不断变化，要维护这样的组员的封锁顺序非常困难，成本很高，而且事务的封锁请求可以随着事务的执行而动态地决定，很难实现确定每一个事务要封锁哪些对象，因此也就很难规定的顺序去施加封锁。

##### 死锁的诊断与解除

超时法：如果某个事物的等待时间超过指定时限，则判定为出现死锁；

等待图法：如果事务等待图中出现了回路，则判断出现了死锁。

对于解决死锁的方法，只能是撤销一个处理死锁代价最小的事务，释放此事务持有的所有锁，同时对撤销的事务所执行的数据修改操作必须加以恢复。

## 数据库设计

### 三范式

范式（Database Normalization）

设计关系[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)时，遵从不同的规范[要求](https://baike.baidu.com/item/%E8%A6%81%E6%B1%82/3598753" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)，设计出合理的关系型数据库，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。

#### 第一范式（1NF）

数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性



应该设计为



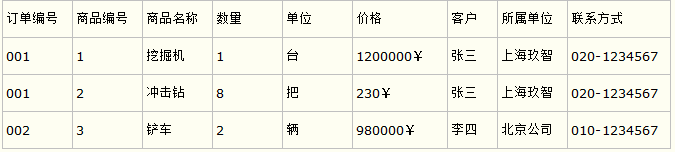
#### 第二范式（2NF）

表中每列与主键相关，而不能只与主键的某部分相关（主要针对联合主键），主键列与非主键列遵循完全函数依赖关系，也就是一个表只描述一件事情。

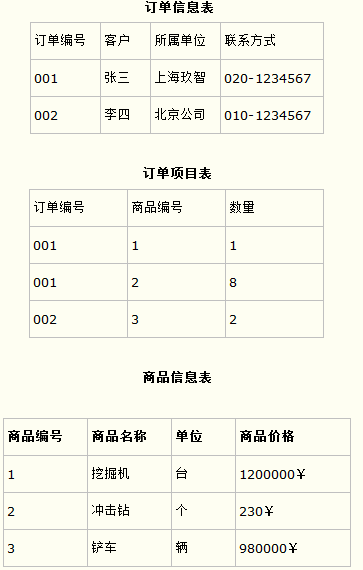
例子引入

需求描述：设计一个订单信息表，订单有多种商品，将订单编号和商品编号作为联合主键。

反范式设计



范式设计



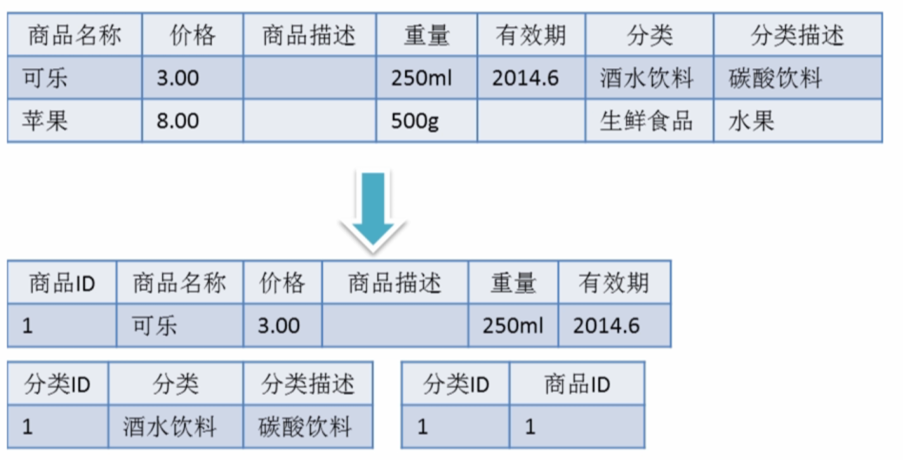
分析

第一种表设计不满足第二范式 ，订单编号和商品编号作为联合主键，由于商品名称，单位，价格这几列只与商品编号有关，与订单编号无关，因此与主键（联合主键）无关，违反范式第二原则；

第二种表设计满足第二范式，把第一种设计表进行拆分，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。

#### 第三范式（3NF）

在2NF基础上，任何非主[属性](https://baike.baidu.com/item/%E5%B1%9E%E6%80%A7" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)不依赖于其它非主属性（在2NF基础上消除传递依赖），简而言之，要求一个数据库表中不包含已在其它表中已包含的非主键字段。表的信息，如果能够被推导出来，就不应该单独的设计一个字段来存放(能尽量外键join就用外键join)。很多时候，我们为了满足第三范式往往会把一张表分成多张表



反范式

查询记录时，范式模式往往要进行多表连接，而反范式只需在同一张表中查询，当数据量很大的时候，显然反范式的效率会更好。

反范式有很多重复的数据，会占用更多的内存，查询时可能会较多地使用DROUP BY或DISTINCT等耗时耗性能的关键字。

当要修改更新数据时，范式更灵活，而反范式要修改全部的数据，且易出错。