Mysql 数 据 库 精 深

## Mysql基础常识

### 无符号与有符号

unsigned与signed代表无符号和有符号两种状态。当定义一个字段类型的时候，默认状态下是有符号的，比如一个有符号类型变量他的长度范围可以是：-128 ~ 127；但是你定义的时候可以特别指出他是一个无符号类型的变量，声明他为unsigned，那么他的表示范围就就变成了：0 ~ 255。

### 字段类型

**char**：代表固定长度字符串，如name字段定义成char(10)，但实际只存了6个，那也占10个字节。好处是查询会快，缺点是空间占得大。

**varchar**：代表可变长度字符串，如name字段定义成 varchar(10) ,但实际存了6个字节，那么将会占据6字节而不是10字节。好处是占空间小，但查询效率没有char好。varchar适合存储备注信息、地址这类字段。

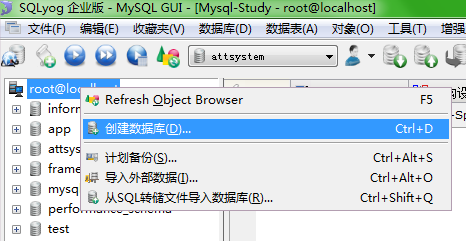
**text**：用于存储大文本文件，如新闻稿件、文章之类的内容。

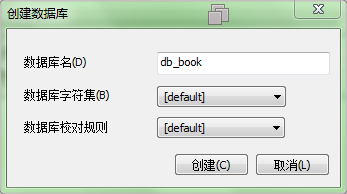
**DECIMAL(M,D)**：decimal，小数的；M表示数据的总长度，D表示小数位；例如：decimal(5,2) 123.45；存入数据的时候按照四舍五入计算。

关于二进制类型数据，一般用来存储图片信息、音频等。例如一些需要加密保存的图片。

### 数据库的创建

可以使用命令行创建，但一般都使用图形界面来创建数据库，这样可以加快开发效率，而且更加直观。方式截图如下，很简单。





## 数据库表基本操作

### 约束条件

PRIMARY KEY：primary key，一个表的主键，唯一标识一条对应的记录；

FOREIGN KEY：foreign key，外键，与另一个表的主键进行关联；

NOT NULL：非空约束，标识该属性不能为空；

UNIQUE：unique，标识该属性是唯一的；

AUTO\_INCREMENT：auto\_increment，标识该属性自动增加；

DEFAULT：default，为该属性设置默认值。

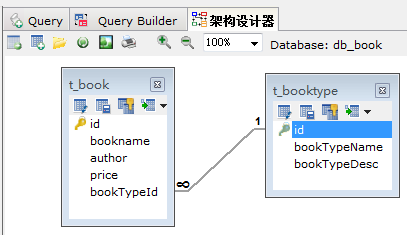
### 创建表

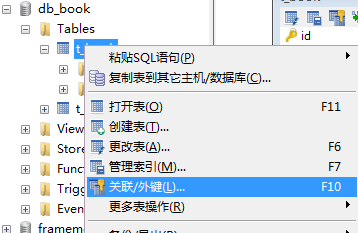
1. **CREATE** **TABLE** t\_bookType(
2. id **int** **primary** **key** auto\_increment,
3. bookTypeName **varchar**(20),
4. bookTypeDesc **varchar**(200)
5. );

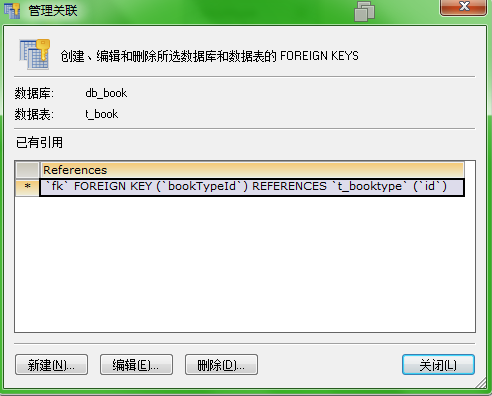
### 主外键关联

1. **CREATE** **TABLE** t\_book(
2. id **int** **primary** **key** auto\_increment,
3. bookName **varchar**(20),
4. author **varchar**(10),
5. price **decimal**(6,2),
6. bookTypeId **int**,
7. **constraint** **`**fk**`** **foreign** **key** (**`**bookTypeId**`**) **references** **`**t\_bookType**`**(**`**id**`**)
8. );

在架构设计器中查看主外键关联情况，把关联的两个表拖进去就可以了：

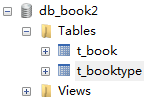




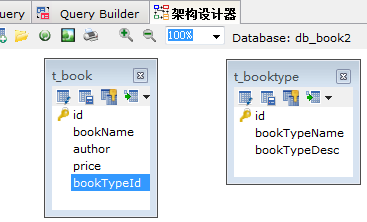


### 工具创建主外键关联

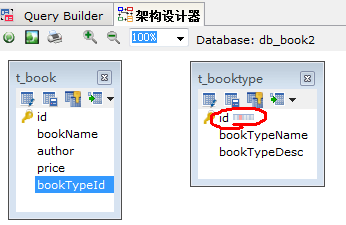
首先创建出相应的两个表，在从表中创建一个对应的外键，等待去关联主表的主键；

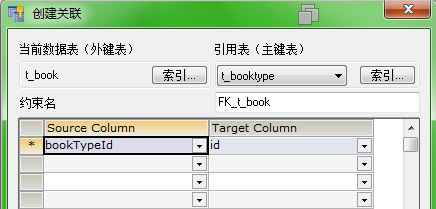


完成后将两个表拖入架构设计器：

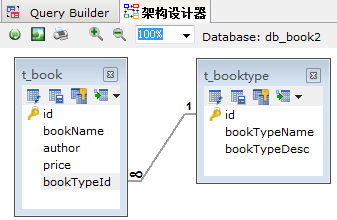


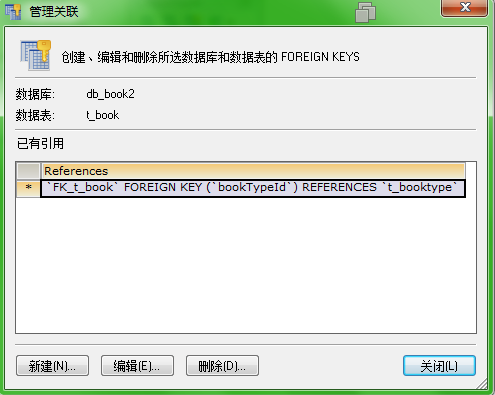
在这里t\_book表作为从表，他的外键是bookTypeId；t\_booktype作为主表，其ID为主键，下一步用鼠标左键将bookTypeId拖入到t\_booktype表的主键上。





点击对话框的【创建】按钮，则关联创建如下：

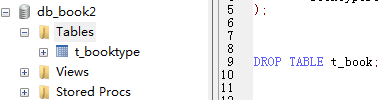




在查看这个关联的时候，与手动创建唯一不同的是他的关联名字：`FK\_t\_book`

### 删除表

**DROP TABLE** t\_book;



刷新后，t\_book表被删除。

## 查询数据 - 单表查询

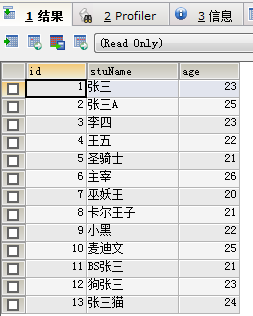
### 查询所有字段

**SELECT** \* **FROM** t\_student;



### 查询指定字段

**SELECT** id,stuName,age **FROM** t\_student



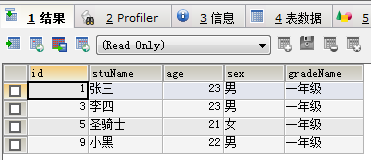
### where条件查询

where条件查询后面跟随一个条件表达式。

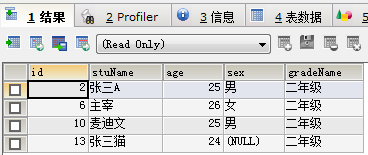
SELECT 字段1，字段2，字段3... FROM 表名 WHERE 条件表达式；

如下两个例子所示：

**SELECT** \* **FROM** t\_student **WHERE** gradeName = '一年级' ;



**SELECT** \* **FROM** t\_student **WHERE** age>23;



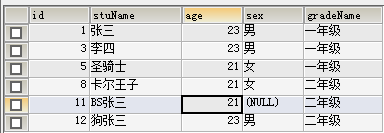
### in条件关键字查询

in关键字是条件表达式的一种，后面跟随一个条件范围集合，基本语法如下：

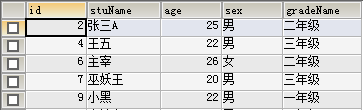
SELECT 字段1，字段2... FROM 表名 WHERE 字段 [NOT]IN(元素1，元素2 …)；

如下两个例子所示：

**SELECT** \* **FROM** t\_student **WHERE** age **IN** (21,23);



**SELECT** \* **FROM** t\_student **WHERE** age **NOT IN** (21,23);

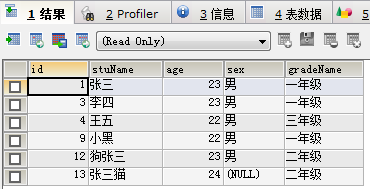


### BETWEEN AND 范围查询

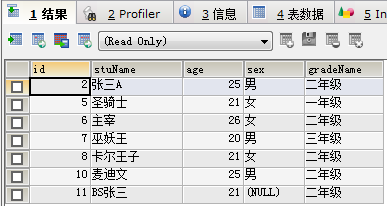
between and 将查询结果控制在一个范围，比如查询年龄在22 ~ 24岁之间的所有人信息。他的基本语法格式如下：

SELECT 字段1，字段2，字段3... FROM 表名 WHERE 字段 [NOT]BETWEEN 取值1 AND 取值2；如下两个例子所示：

**SELECT** \* **FROM** t\_student **WHERE** age  **BETWEEN** 22 **AND** 24;



**SELECT** \* **FROM** t\_student **WHERE** age **NOT BETWEEN** 22 **AND** 24;

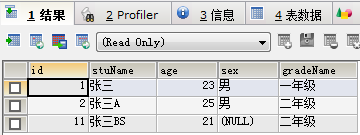


### LIKE关键字与模糊查询

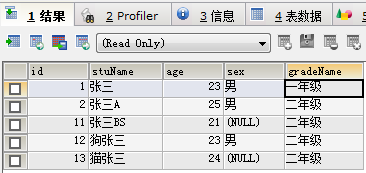
like关键字用于模糊查询，比如查询出姓名中所有带“张三”两个字的所有记录信息。他的基本语法如下所示：

SELECT 字段 1，字段2，字段3...FROM 表名 WHERE 字段 [NOT]LIKE‘字符串’；“%”代表任意字符；“\_” 代表单个字符；如：'%张三%';、'张三\_\_';、'张三%';、'张三'; 等。如下几个例子所示：

**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '张三%';



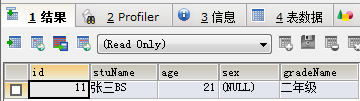
**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '%张三%';



**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '%张三';



**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '张三\_ \_';



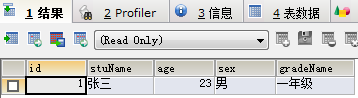
**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '张三\_';



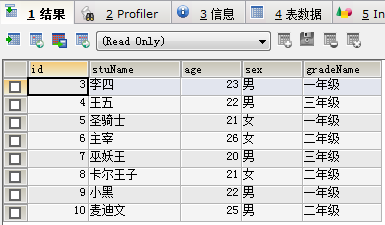
**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '\_张三';



**SELECT** \* **FROM** t\_student **WHERE** stuName **LIKE** '张三';



**SELECT** \* **FROM** t\_student **WHERE** stuName **NOT LIKE** '%张三%';



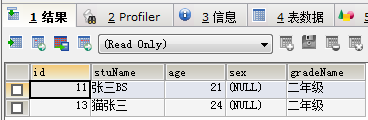
查询出所有stuName字段中不包含张三的信息会用到 **NOT LIKE**。

### 空值查询 IS NULL

用于查询某一个字段是否为空的情况。其语法格式如下：

SELECT 字段1，字段2，字段3... FROM 表名 WHERE 字段 IS [NOT] NULL；示例如下：

**SELECT** \* **FROM** t\_student **WHERE** sex **IS NOT NULL**;



### 多条件查询AND / OR

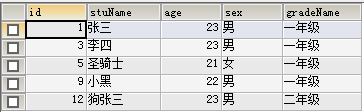
即多条件表达式查询。其语法格式如下：

SELECT 字段 1，字段2... FROM 表名 WHERE 条件表达式1 AND 条件表达式2[...AND 条件表达式n] 。或“AND”换成“OR”。示例如下：

**SELECT** \* **FROM** t\_student **WHERE** gradeName='一年级' **AND** age=23



**SELECT** \* **FROM** t\_student **WHERE** gradeName='一年级' **OR** age=23



### DISTINCT 去重复查询

distinct 释义：不同的|清楚的|明显的；用于去除重复信息。其语法格式如下：

SELECT DISTINCT 字段名 FROM 表名；示例如下。

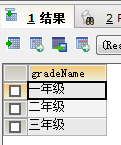
当我们不使用distinct关键字的时候，会出现重复字段：

**SELECT** gradeName  **FROM** t\_student



当我们 使用distinct关键字的时候，可以去除重复字段：

**SELECT DISTINCT** gradeName  **FROM** t\_student



### GROUP BY 分组查询

group by 分组，Select一个字段可以跟一个函数，但是不能再加入其他字段，因为Group By分组的依据是那个被Select的字段，group by可以单独使用。语法格式如下：

GROUP BY 属性名 [HAVING 条件表达式][WITH ROLLUP]

1，单独使用(毫无意义)；

2，与GROUP\_CONCAT()函数一起使用；concat，n，合并多个字符串或数组。

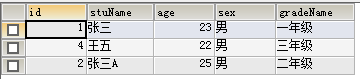
3，与聚合函数一起使用；如COUNT()函数

4，与HAVING 一起使用(限制输出的结果)； having；

5，与WITH ROLLUP一起使用(最后加入一个总和行)；with rollup：与汇总。

使用举例如下：

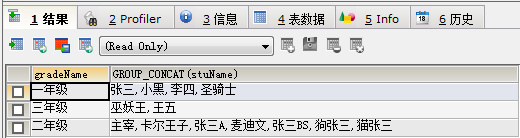
**SELECT** \* **FROM** t\_student **GROUP** **BY** gradeName;



可以看到，数据丢失了,是因为 **\*** 代表多条件，导致。即只能有一个字段被查询。

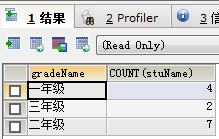
**SELECT** gradeName,GROUP\_CONCAT(stuName) **FROM** t\_student

**GROUP** **BY** gradeName;



中间以逗号分隔开，在Java或C#中用split()函数就可以很容易的取出这些数据。针对聚合函数的结合使用，举例如下：查询每个年级的学生总数。

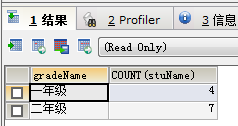
**SELECT** gradeName,COUNT(stuName) **FROM** t\_student **GROUP** **BY** gradeName;



利用HAVING筛选查询结果。

**SELECT** gradeName,COUNT(stuName) **FROM** t\_student **GROUP** **BY**

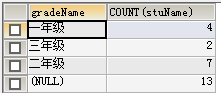
gradeName **HAVING** COUNT(stuName)>3;



WITH ROLLUP动态的在最后一行加入一个总和的计算或字符叠加，不是很常用。

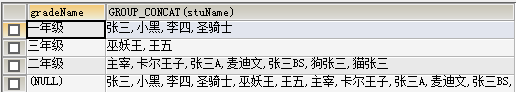
**SELECT** gradeName,COUNT(stuName) **FROM** t\_student

**GROUP** **BY** gradeName **WITH** **ROLLUP**;



**SELECT** gradeName,GROUP\_CONCAT(stuName) **FROM** t\_student

**GROUP** **BY** gradeName **WITH** **ROLLUP**;



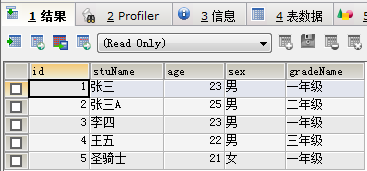
### LIMIT 分页查询

limit 限制，界限。分页查询，其语法格式如下：

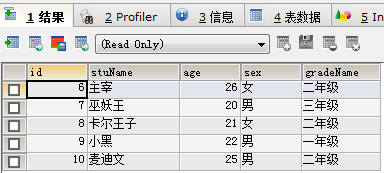
SELECT 字段 1，字段2... FROM 表名 LIMIT 初始位置，记录数；

使用举例如下：

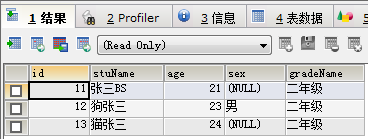
**SELECT** \* **FROM** t\_student **LIMIT** 0,5;



**SELECT** \* **FROM** t\_student **LIMIT** 5,5;



**SELECT** \* **FROM** t\_student **LIMIT** 10,5;



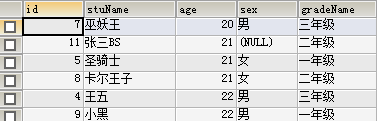
### 查询结果排序 ASC / DESC

对查询结果按照升序或降序进行排序。其中ASC代表升序排列，默认情况下是按照升序排列的；DESC代表降序排列，需要指明。其语法格式如下：

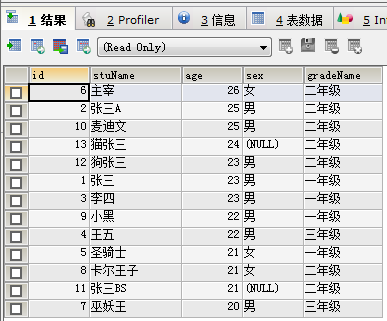
SELECT 字段1，字段2... FROM 表名 ORDER BY 属性名 [ASC|DESC]

使用举例如下：

**SELECT** \* **FROM** t\_student **ORDER** **BY** age **ASC**;

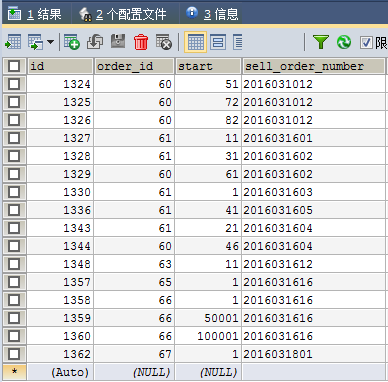


**SELECT** \* **FROM** t\_student **ORDER** **BY** age **DESC**;



### 高级应用 - DISTINCT与一维分组

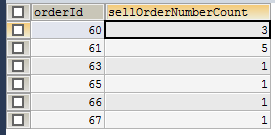
场景描述如下：以order\_id分组，计算出sell\_order\_number的数量。这里需要Distinct和Group by进行综合使用。



Sql 代码如下：

1. **SELECT**
2. order\_id **AS** orderId,
3. COUNT(**DISTINCT** sell\_order\_number) **AS** sellOrderNumberCount
4. **FROM**
5. coupon\_barcode\_segment\_used
6. **GROUP** **BY** order\_id ;

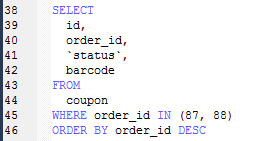
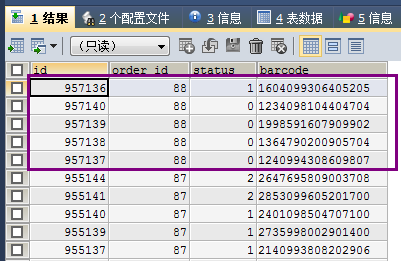
结果如下：



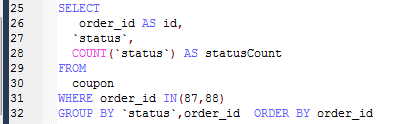
这里需要注意Distinct关键字的用法。

### 高级应用 – 二维分组

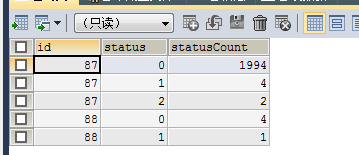
场景描述如下：根据order\_id和status分为2组。示例数据如下：

二维分组应用举例如下：



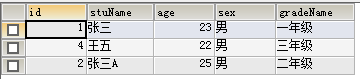
结果如下：



对比3.10节中的第一个示例，可以更容易的理解二维分组与多维分组的原理

重现代码如下：

**SELECT** \* **FROM** t\_student **GROUP** **BY** gradeName;

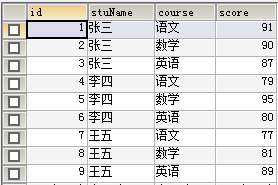


可以看到，数据丢失了,是因为 **\*** 代表多条件，导致。即只能有一个字段被查询。

这是因为你分组的依据是一个字段，如果你查询的内容是stuName，age和sex三个字段，分组也以这三个字段为依据，那么就不会出错。总之，你查询的**表字段**要和你的分组依据相对应。注意，这里指明的是表字段，没写sql函数。从这里就可以引申出多维分组的应用。

## 聚合函数查询

新建一个t\_grade表，表数据如下：



### COUNT() 函数

COUNT()函数用来统计记录的条数；与GOUPE BY关键字一起使用。示例如下：

**SELECT** COUNT(\*) **FROM** t\_grade;



**SELECT** COUNT(\*) **AS** total **FROM** t\_grade; 为他取一个名字：total。

AS total是为这个数据列取一个名字。

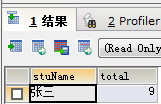


**SELECT** stuName **FROM** t\_grade

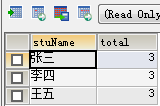


如果不使用GROUP BY，会出现查询错误：

**SELECT** stuName,COUNT(\*) **AS** total **FROM** t\_grade ;



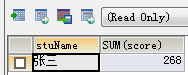
**SELECT** stuName,COUNT(\*) **AS** total **FROM** t\_grade **GROUP** **BY** stuName;



### SUM() 函数

SUM()函数是求和函数；与GOUPE BY关键字一起使用。如下所示：

**SELECT** stuName,SUM(score) **FROM** t\_grade **WHERE** stuName="张三";



**SELECT** stuName,SUM(score) **FROM** t\_grade **GROUP** **BY** stuName;

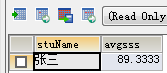


### AVG() 函数

AVG()函数是求平均值的函数；与GOUPE BY关键字一起使用。如下所示：

**SELECT** stuName,AVG(score) **AS avgsssFROM** t\_grade

**WHERE** stuName="张三";



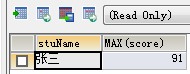
**SELECT** stuName,AVG(score) **FROM** t\_grade **GROUP** **BY** stuName;



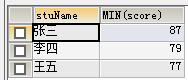
### MAX()函数 和 MIN()函数

MAX()函数是求最大值的函数；MIN()函数是求最小值的函数。与GOUPE BY关键字一起使用。示例如下：

**SELECT** stuName, **MAX**(score) **FROM** t\_grade **WHERE** stuName="张三";



**SELECT** stuName,**MIN**(score) **FROM** t\_grade **GROUP** **BY** stuName;



### 聚合函数左连接陷阱 TODO

同时使用左连接或者有链接会出现计算值翻倍的情况

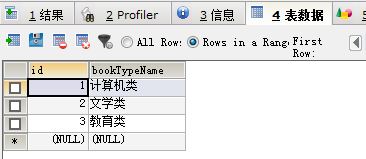
表结构一对多会出现。

两次左连接 计算值会翻倍。没有找到根本原因，请延伸此处

## 连接查询

连接查询是将两个或两个以上的表按照某个条件连接起来，从中选取需要的数据；内连接查询用的比较多，但外连接也会用，只是并不是非常多。创建示例数据表2个，如下：



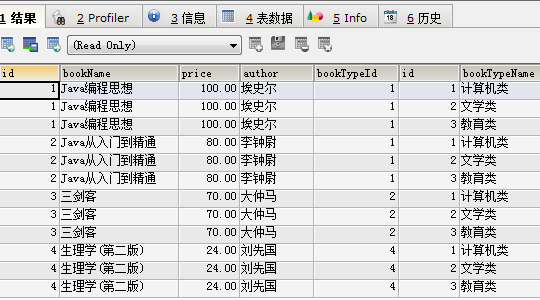


t\_book 和 t\_booktype

### 内连接查询 – 广义笛卡尔积

广义迪卡儿积没有限定语句，结果会产生N\*M条记录。示例如下：

**SELECT** \* **FROM** t\_book , t\_bookType;



### 内连接查询 – 条件限制

**SELECT** tb.id, tb.bookName,tb.price,tb.author,tby.bookTypeName **FROM**

t\_book tb, t\_bookType tby **WHERE** tb.bookTypeId=tby.id;



### 外连接查询

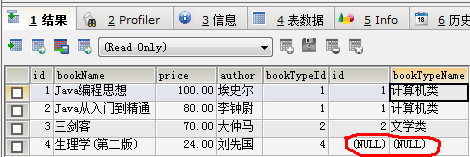
外连接可以查出某一张表的所有信息； 语法格式如下：

SELECT 属性名列表 FROM 表名 1LEFT|RIGHT JOIN 表名2 ON 表名 1.属性名1=表名2.属性名2；

### 外连接查询-左连接查询

可以查询出“表名1”的所有记录，而“表名2”中，只能查询出匹配的记录；

**SELECT** \* **FROM** t\_book tb LEFT JOIN t\_bookType tby **ON** tb.bookTypeId=tby.id;



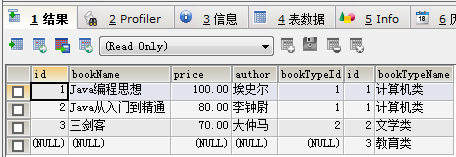
从结果可以看出，t\_bookType表中不符合查询条件的记录会被NULL所代替。

### 外连接查询-右连接查询

可以查询出“表名2”的所有记录，而“表名 1”中，只能查询出匹配的记录；

**SELECT** \* **FROM** t\_book tb RIGHT **JOIN** t\_bookType tby

**ON** tb.bookTypeId=tby.id;



### 多条件查询 AND

使用AND作为连接条件。示例如下：

**SELECT** tb.id, tb.bookName,tb.price,tb.author,tby.bookTypeName **FROM**

t\_book tb, t\_bookType tby **WHERE** tb.bookTypeId=tby.id **AND** tb.price>70;



### zid在一个表中而不在另一个表中

场景描述：A、B两表，找出ID字段中，存在A表，但是不存在B表的数据。

方法：使用左连接：

select

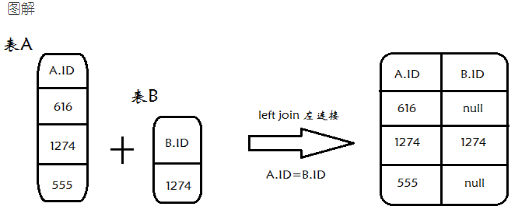
\*

from

managercenter.`mc\_user\_info` info left join managercenter.`mc\_user\_role` ur

on info.id = ur.mc\_user\_id

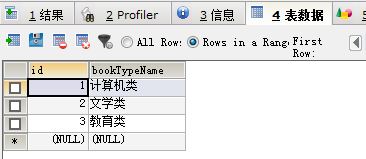
where ur.id is null



## 子查询

创建示例数据表3个，如下：







t\_book 、 t\_booktype和t\_pricelevel

### 子查询 - 关键字In

一个查询语句的条件可能落在另一个SELECT语句的查询结果中。表示一个集合数据。

**SELECT** \* **FROM** t\_book tb

**WHERE** tb.booktypeId **IN** (**SELECT** id **FROM** t\_booktype);

这个查询语句的意思是：查询t\_book表中的所有字段，其限定条件是t\_book表中的booktypeId字段需要在 IN这个范围内；而IN的范围是由一个查询条件给出的集合数据。如果我们单独执行这个句话：**SELECT** id **FROM** t\_booktype；那么会得到如下结果：



也就是说IN的范围是{1,2,3}，也就是说这个查询语句还可以表述成如下：

**SELECT** \* **FROM** t\_book tb **WHERE** tb.booktypeId **IN** (1，2，3);



**ELECT** \* **FROM** t\_book tb **WHERE** tb.booktypeId **NOT IN** (1，2，3);



### 子查询 - 比较运算符

子查询可以使用比较运算符。使用比较运算符进行子查询操作，一般其子语句的结果会是一个查询条件，你不要放一个集合数据进去，这样的逻辑上就有问题，自然也会报错。示例如下所示：查询t\_book中的所有信息，限定条件是price >= 80。

**SELECT** \* **FROM** t\_book **WHERE** price>= (

**SELECT** price **FROM** t\_pricelevel **WHERE** priceLevel=1);



### 子查询 - 关键字Exists

假如子查询查询到记录，则进行外层查询，否则，不执行外层查询；

**SELECT** \* **FROM** t\_book **WHERE** **EXISTS** (**SELECT** \* **FROM**

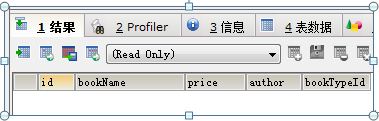
t\_booktype **WHERE** id <3);



将查询条件改变，子查询中没有结果，则不会执行外层查询：

**SELECT** \* **FROM** t\_book **WHERE** **EXISTS** (**SELECT** \* **FROM**

t\_booktype **WHERE** id >3);



### 子查询 - 关键字Any

ANY关键字表示满足其中任一条件；

**SELECT** \* **FROM** t\_book



**SELECT** \* **FROM** t\_book **WHERE** price>= **ANY** (

**SELECT** price **FROM** t\_pricelevel);



**注意！**这里不能等同于：**SELECT** \* **FROM** t\_book **WHERE** price>= **ANY** **(40,60,80)**;

这样的语法是错误的！

### 子查询 - All关键字

ALL关键字表示满足所有条件；

**SELECT** price **FROM** t\_pricelevel;



**SELECT** \* **FROM** t\_book;



当使用ALL关键字时，表示需要满足price != 40/60/80三个值。

**SELECT** \* **FROM** t\_book **WHERE** price != **ALL** (

**SELECT** price **FROM** t\_pricelevel);



### 多表联合查询

即2个以上的表进行联合查询，如三表联查。

**SELECT** priceLevel **AS** '第三层'  **FROM** t\_pricelevel **WHERE** id <2;



**SELECT** id **AS** '第二层' **FROM** t\_booktype **WHERE** id **IN** (

**SELECT** priceLevel **AS** '第三层' **FROM** t\_pricelevel **WHERE** id <2));



**SELECT** \* **FROM** t\_book tb **WHERE** tb.booktypeId **IN** (

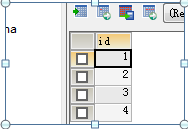
**SELECT** id **AS** '第二层' **FROM** t\_booktype **WHERE** id **IN** (

**SELECT** priceLevel **AS** '第三层' **FROM** t\_pricelevel **WHERE** id <2));

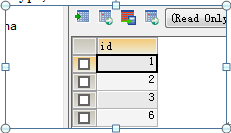


## 合并查询结果

**SELECT** id **FROM** t\_book;



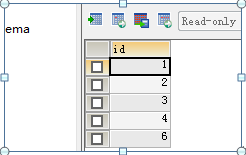
**SELECT** id **FROM** t\_booktype;



### UNION 去除相同记录

使用UNION关键字是，数据库系统会将所有的查询结果合并到一起，然后去除掉相同的记录；

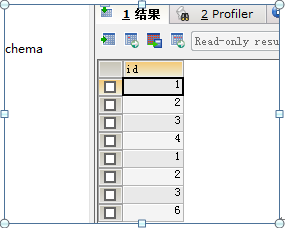
**SELECT** id **FROM** t\_book **UNION** **SELECT** id **FROM** t\_booktype;



### UNION ALL

使用UNION ALL，不会去除掉系统的记录；

**SELECT** id **FROM** t\_book **UNION ALL** **SELECT** id **FROM** t\_booktype;



### 为表取别名

格式： 表名 表的别名。

**SELECT** \* **FROM** t\_book t **WHERE** t.id>1;



### 为字段取别名

格式： 属性名 [AS] 别名。AS关键字可以加也可以不加。

为bookName列起别名为：bName。

**SELECT** t.id, t.bookName bName **FROM** t\_book t **WHERE** t.id>1;



添加AS关键字：

**SELECT** t.id, t.bookName **AS** bName **FROM** t\_book t **WHERE** t.id>1;

可见查询结果是一样的。

## 插入 - 更新 - 删除数据

这里使用到t\_book表，其原始数据显示如下：



### 所有字段插入

格式：INSERT INTO 表名 VALUES(值1，值2，值3，...，值n)；

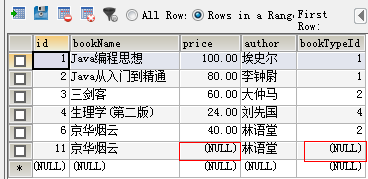
**INSERT** **INTO** t\_book **VALUES**(NULL,'京华烟云',40,'林语堂',2);



### 指定字段插入

格式：INSERT INTO 表名(属性1...属性n) VALUES(值 1...，值n)；

**INSERT** **INTO** t\_book(bookName,author) **VALUES**('京华烟云','林语堂');



从结果来看，没有被指定插入数据的字段，则被换成了NULL来代替。

### 加强版数据插入replace into

replace into是insert into的增强版。在向表中插入数据时，我们经常会遇到这样的情况：1、首先判断数据是否存在；2、如果不存在，则插入；3、如果存在，则更新。

在SQL Server中可以这样处理：

IF NOT EXISTS (SELECT 1 FROM t WHERE id = 1)?

INSERT INTO t(id, update\_time) VALUES(1, getdate())

ELSE

UPDATE t SET update\_time = getdate() WHERE id = 1

MySQL中有更简单的方法：replace into。这个语法是MySQL独有的。

replace into t(id, update\_time) values(1, now());

replace into 跟 insert 功能类似，不同点在于：replace into 首先尝试插入数据到表中， 1. 如果发现表中已经有此行数据（根据主键或者唯一索引判断）则先删除此行数据，然后插入新的数据。 2. 否则，直接插入新数据。要**注意**的是：插入数据的表必须有主键或者是唯一索引！否则的话，replace into 会直接插入数据，这将导致表中出现重复的数据。

尤其对于大数据量的插入，比如批量插入数据库5000条、1万条等等，通过像MyBatis这样的框架，Java在Service层完成List数据组织(有效数据过滤)，但不能保证过滤完全没问题，所以都会选择以replace into的方式插入数据。

开发中的实例如下：

**Controller层：**

List<MovieRemote> movieRemoteList =mrService.getMovieRemoteList();

**Service层接口：**

**public** List<MovieRemote> getMovieRemoteList();

**Service层实现类：**

**public** **int** insertEntityList(List<MovieRemote> insertList) {

**return** mrMapper.insertEntityList(insertList);

}

**Dao层：**

**public** **int** insertEntityList(List<MovieRemote> insertList);

**Mapper.xml配置文件：**

<insert id=*"insertEntityList"* useGeneratedKeys=*"true"* parameterType=*"java.util.List"*>

replace into movie\_remote (number, name, type) values

<foreach collection=*"list"* item=*"item"* index=*"index"* separator=*","* >

( #{item.number,jdbcType=VARCHAR},

#{item.name,jdbcType=VARCHAR},

#{item.type,jdbcType=VARCHAR})

</foreach>

</insert>

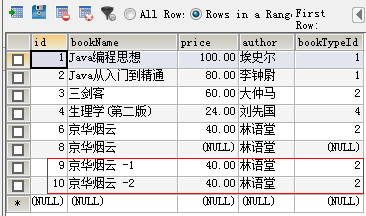
注意：这里的主键是number，他是String类型，不是自增的，如果主键自增需要查下最后生成的id，百度都有。

### 同时插入多条记录

格式：INSERT INTO 表名 [(属性列表)] VALUES(取值列表1)，(取值列表2)...， (取值列表n)；

**INSERT** **INTO** t\_book(id,bookName,price,author,bookTypeId)

**VALUES** (NULL,'京华烟云 -1',40,'林语堂',2),(NULL,'京华烟云 -2',40,'林语堂',2);



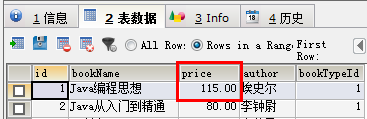
### 更新数据

格式：UPDATE 表名 SET 属性名1=取值1，属性名2=取值2，...，属性名n=取值n WHERE 条件表达式；

我们准备更新第一条记录的价格，将100.00更新成115.00。



**UPDATE** t\_book **SET** price=115 **WHERE** id=1;



**UPDATE** t\_book **SET** bookName='Java编程思想(第八版)',price=125 **WHERE** id=1;

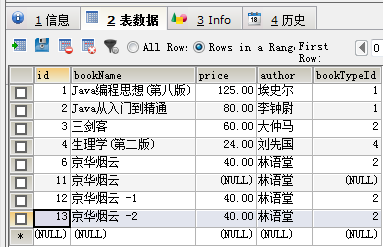


### 同时更新多条记录

**UPDATE** t\_book **SET** bookName='读者',price=4,author='中共'

**WHERE** bookName **LIKE** '%京华%';

更新前表的内容如下：



更新后表的内容变化如下：



### 删除数据

DELETE FROM 表名 [WHERE 条件表达式]；

**DELETE**  **FROM**  t\_book  **WHERE**  id=5;

根据上图所示，id=6的那条数据会被删除，其结果如下：



**DELETE**  **FROM**  t\_book  **WHERE**  bookName='读者';

删除所有bookName=’读者’的记录，其结果如下：

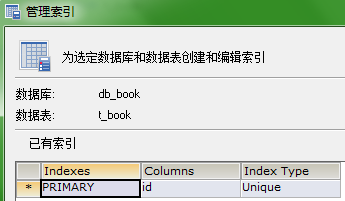


## 索引的使用

### 索引常识

索引定义：索引是由数据库表中一列或者多列组合而成，其作用是提高对表中数据的查询速度；类似于图书的目录，方便快速定位，寻找指定的内容；

一个表的主键字段是他的默认索引。如下图：

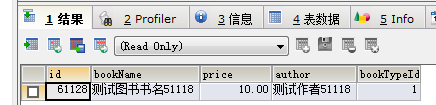


优点：提高查询数据的速度；

缺点：创建和维护索引的时间增加了；

其效率对比如下，首先针对bookName字段不创建索引进行查询：

**SELECT** \* **FROM** t\_book **WHERE** bookName='测试图书书名51118';

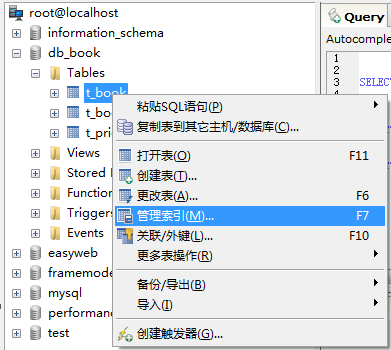


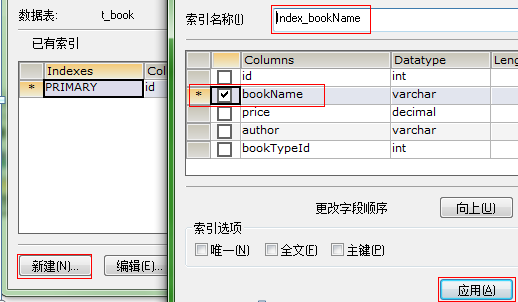


从其查询结果显示来看，在11万条数据中进行查询，一共耗时78毫秒，一条记录。MySql的执行效率还是很不错的，下面将演示对bookName字段创建索引后的效果。

### 创建示例

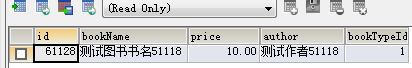
利用图形界面创建索引：





点击【应用】，bookName字段索引创建完毕。

再次执行上面的查询语句，结果如下。查询耗时已经降到毫秒级以下。





## 索引高级特性与实践

### 索引分类

**1 - 普通索引**

这类索引可以创建在任何数据类型中；

**2 - 唯一性索引**

使用UNIQUE参数可以设置，在创建唯一性索引时，限制该索引的值必须是唯一的；

**3 - 全文索引**

使用FULLTEXT参数可以设置，全文索引只能创建在CHAR，VARCHAR，TEXT类型的字段上。主要作用 就是提高查询较大字符串类型的速度；只有**MyISAM** 引擎支持该索引，Mysql默认引擎不支持；

**4 - 单列索引**

在表中可以给单个字段创建索引，单列索引可以是普通索引，也可以是唯一性索引，还可以是全文索引；

**5 - 多列索引**

多列索引是在表的多个字段上创建一个索引；

**6 - 空间索引**

使用SPATIAL参数可以设置空间索引。空间索引只能建立在空间数据类型上，这样可以提高系统获取空间数 据的效率；只有MyISAM 引擎支持该索引，Mysql默认引擎不支持；

### 建表同时创建索引

语法格式如下：

**CREATE TABLE** 表名 (属性名 数据类型 [完整性约束条件],

属性名 数据类型 [完整性约束条件],

....

属性名 数据类型

[UNIQUE|FULLTEXT|SPATIAL] **INDEX**|**KEY**

[别名](属性名1[(长度)][ASC|DESC])

);

**1 - 创建普通索引**

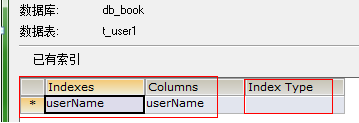
**CREATE** **TABLE** t\_user1(id **INT** ,

userName **VARCHAR**(20),

**password** **VARCHAR**(20),

**INDEX** (userName)

);



Index Type栏是空的，因为没有指定索引类型。

**2 - 创建唯一性索引**

**CREATE** **TABLE** t\_user1(id **INT** ,

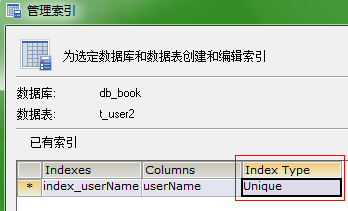
userName **VARCHAR**(20),

**password** **VARCHAR**(20),

**UNIQUE INDEX**  index\_userName(userName)

);

使用UNIQUE(unique)关键字创建唯一性索引。其中index\_userName是别名。



ALTER TABLE `cps\_ironic`.`interface` ADD UNIQUE INDEX `i\_interface\_ssi` (`switch\_ip`, `switch\_port`,`is\_del`)

**3 - 创建全文索引**

创建全文索引需要声明：FULLTEXT关键字，在Mysql中只有**MyISAM**引擎支持全文索引。这里不再举例。

**4 - 创建单列索引**

上述两个例子创建的都是单列索引。

**5 - 创建多列索引**

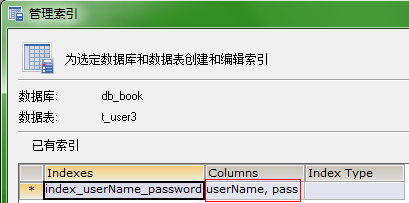
**CREATE** **TABLE** t\_user1(id **INT** ,

userName **VARCHAR**(20),

**pass** **VARCHAR**(20),

**INDEX**  index\_userName\_password(userName,pass)

);



可见Columns列有2个值。

**6 - 创建空间索引**

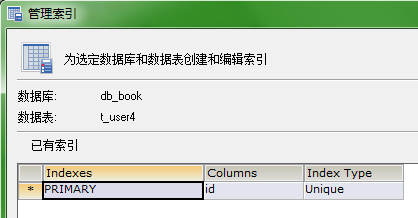
使用SPATIAL参数可以设置空间索引。只有MyISAM 引擎支持该索引。

### 建表后创建索引

语法格式如下：

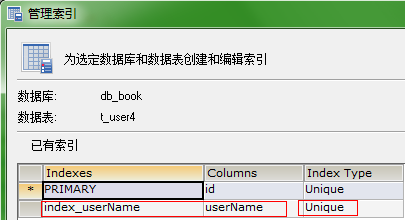
CREATE [UNIQUE |FULLTEXT|SPATIAL] INDEX 索引名 ON 表名 (属性名 [(长度)][ASC |DESC])；

默认情况下，主键是唯一索引，如下所示：



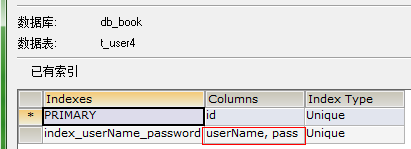
为一个字段创建一个唯一索引：

**CREATE**  **UNIQUE INDEX** index\_userName **ON** t\_user4(userName);



创建多列索引：

**CREATE**  **UNIQUE INDEX** index\_userName **ON** t\_user4(userName,pass);

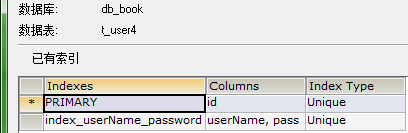


### ALTER TABLE语句来创建索引

ALTER TABLE 表名 ADD[UNIQUE |FULLTEXT|SPATIAL]INDEX索引名(属性名 [(长度)][ASC |DESC]);

**ALTER**  **TABLE** t\_user4  **ADD**  **UNIQUE**  **INDEX**

index\_userName\_password(userName,pass);



## 视图操作

### 视图的引入

1 视图是一种**虚拟的表**，是从数据库中一个或者多个表中导出来的数据组成的虚拟表。

2 数据库中只存放了视图的定义而并没有存放视图中的数据，这些数据存放在原来的表中。

3 使用视图查询数据时，数据库系统会从原来的表中取出对应的数据。

### 视图的作用

1 使操作简便化；

2 增加数据的安全性；

3 提高表的逻辑独立性；

### 创建视图

CREATE [ALGORITHM = { UNDEFIEND |MERGE|TEMPTABLE}]

VIEW 视图名 [( 属性清单)]

AS SELECT 语句

[WITH [ CASCADED|LOCAL] CHECK OPTION]；

----------------------------------------------------------------------------------------------------------------------

ALGORITHM(algorithm['æl gə' rɪ ðəm] 算法，运算法则)：可选参数，表示视图选择的算法，包括3个选项 UNDEFIEND(undefiend:默认的)| MERGE(merge:合并)|TEMPTABLE。 其中：

UNDEFINED选项表示MySQL将 自动选择所要使用的算法；**一般选默认**

MERGE选项表示将使用视图的语句与视图定义合并起来，使得视图定义的某一部分取代语句的对应部分；

TEMPTABLE选项表示将视图的结果存入临时表，然后使用临时表执行语句；

【视图名】参数表示要创建的视图的名称。

【属性清单】是可选参数，其指定了视图中各种属性的名词，默认情况下与SELECT语句中查询的属性相同。

SELECT 语句参数是一个完整的查询语句，标识从某个表查出某些满足条件的记录，将这些记录导入视图中。

WITH CHECK OPTION是可选参数，表似乎更新视图时要保证在该视图的权限范围之内。

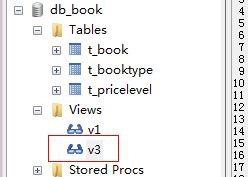
CASCADED (cascaded)是可选参数，表示更新视图时要满足所有相关视图和表的条件，该参数为默认值。

LOCAL表示更新视图时，要满足该视图本身的定义条件即可。

### 单表创建视图

**CREATE** **VIEW** v3(bbb,ppp)  **AS**  **SELECT** bookName,price **FROM** t\_book;

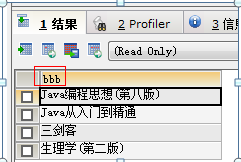
在单表上创建一个视图，视图中包含t\_book表中的两个字段，并重新在视图中定义这两个字段的名字。其显示结果如下：





查询视图v3中的一个字段的数据记录bbb。

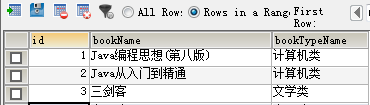
**SELECT**  bbb  **FROM** v3;



### 多表创建视图

**CREATE** **VIEW** v4 **AS** **SELECT** tb.id,tb.bookName,tby.bookTypeName

**FROM** t\_book tb,t\_booktype tby **WHERE** tb.bookTypeId=tby.id;



创建一个多表视图，包含t\_book和t\_booktype两个表，结果显示如上。

查询v4视图中的id和bookTypeName两个字段的所有数据

**SELECT**  id , bookTypeName  **FROM** v4;



### 查看视图

**1 - DESCRIBE 语句查看视图基本信息**

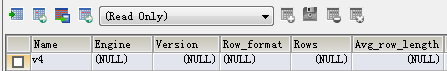
**DESC** v4;



**2 - SHOW TABLE STATUS 查看视图状态信息**

加上LIKE这是基本语法，这个语法不仅可以查看视图，也可以查看表。对比显示如下，查看视图如下：

SHOW **TABLE** STATUS LIKE 'v4';

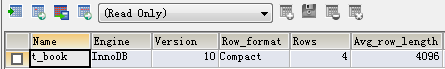






根据查询结果可以看到，表状态除了Name和Comment两个字段外，都是空的，这也印证了视图是一个虚表。对比查看表的结果，如下：

SHOW **TABLE** STATUS LIKE 't\_book';







由此可见，表的状态信息是存在的。

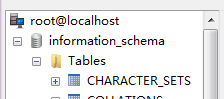
**3 - SHOW CREAT EVIEW 查看视图详细信息**

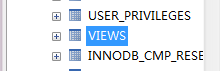
SHOW **CREATE** **VIEW** v5;

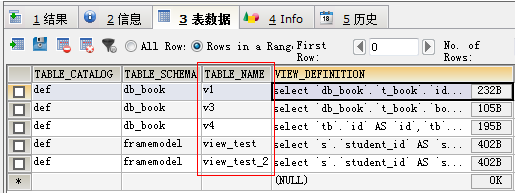


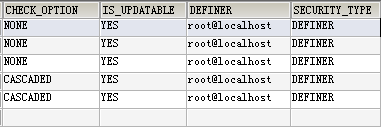
会显示出视图名称、创建视图的语句、字符集设置等等。

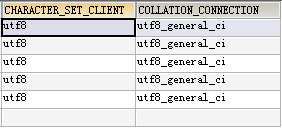
**4 - 在information\_schema的VIEWS表中查看视图详细信息**











### 修改视图

即修改以前定义的字段内容。准备工作：创建一个视图V1，首先用CREATE OR REPLACE语句修改它，然后用ALTER语句还原它。

**CREATE** **VIEW** v1 **AS** **SELECT** \* **FROM** t\_book;



新建视图V1如上图所示。

**1 – CREATE OR REPLACE VIEW修改视图**

CREATE OR REPLACE [ALGORITHM={UNDEFINED |MERGE |TEMPTABLE}]

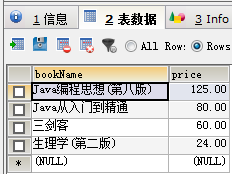
VIEW 视图名 [( 属性清单 )]

AS SELECT 语句

[WITH [CASCADED|LOCAL]CHECKOPTION]；

首先用CREATE OR REPLACE语句修改它。

**CREATE** **OR** REPLACE **VIEW** v1(bookName,price) **AS** **SELECT** bookName,price **FROM** t\_book;



**2 - ALTER 语句 修改视图**

ALTER [ALGORITHM={UNDEFINED |MERGE|TEMPTABLE}]

VIEW 视图名 [( 属性清单 )]

AS SELECT 语句

[WITH [CASCADED|LOCAL]CHECKOPTION]；

利用ALTER语句还原V1。这里要**注意**，ALTER语句要求很严格，空格多一个都不行。

**ALTER** **VIEW** v1 **AS** **SELECT** \* **FROM** t\_book;



**3 – CREATE OR REPLACE与ALTER的区别**

CREATE OR REPLACE 代表不存在则进行创建工作，如果存在则进行修改替换工作。而ALTER语句则完全是用于修改一个视图或者表。alter: ['ɔl tɚ]|vt|改变，更改。

### 更新视图

首先，【更新视图】是一个重要的概念，应用广泛。

其次，对视图的更新操作会影响到与该视图相关的表。

最后，视图更新包括了三种操作：INSERT / UPDATE / DELETE。

更新视图是指通过视图来插入(INSERT) 、更新(UPDATE) 和删除(DELETE)表中的数据。因为视图是一个虚拟的表，其中没有数据。通过视图更新时，都是转换基本表来更新。更新视图时，只能更新权限范围内的数据。超出了范围，就不能更新。

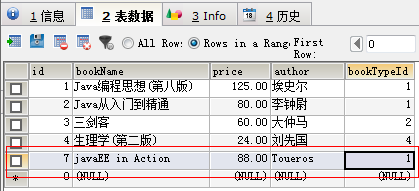
关于视图更新有如下需要注意的地方：

1. 单表视图可以更新。即此视图只对应一张表的数据。
2. 多表视图是不能更新的。
3. 综上两点即：视图不和基本表一一对应，是不能更新的。
4. 视图中虽然可以更新数据，但是有很多限制。一般情况下，最好将视图作为查询数据的虚拟表，而不要通过视图更新数据。
5. 对单表视图的更新会影响到其对应的基本数据表。如：视图插入一条记录，数据表中也会插入一条一样的记录。
6. 如果想达到多表数据更新的效果，需要使用“触发器”。

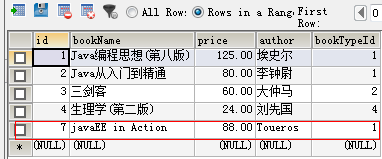
准备工作如下：继续利用视图V1，对他进行插入、更新和删除三种操作，并观察此三种行为给对应表中的数据带来的变化。

**1 – 插入(INSERT)**

**INSERT** **INTO** v1 **VALUES**(NULL,'javaEE in Action',88,'Toueros',1);



这里看到结果，视图V1中插入了一条数据。在对应的表t\_book中也出现了该条数据：



**2 – 更新(UPDATE)**

**UPDATE** v1 **SET** bookName='C# in Action',price=100,author='Microsoft'

**WHERE** id=7;

执行完，我们可以发现视图和基本表中的第七条记录出现变更。



**3 – 删除(DELETE)**

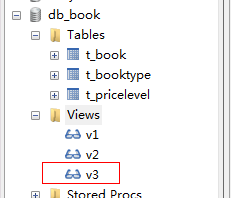
**DELETE** **FROM** v1 **WHERE** id=7;



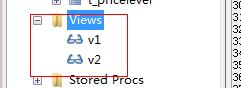
视图和基本表中的第七条记录被成功删除。

### 删除视图

没什么好说的，举个例子，删除视图V3。



**DROP** **VIEW** IF EXISTS v3;



视图V3被删除。

## 触发器介绍及其原理

### 触发器的引入及相关注意点

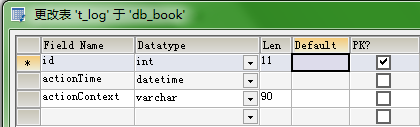
触发器：TRIGGER (trigger['trɪgə] |n. 扳机；[电子] 触发器；制滑机) 是由事件来触发某个操作。这些事件包括INSERT语句、UPDATE语句和DELETE语句。当数据库系统执行这些事件时，就会激活触发器执行相应的操作。

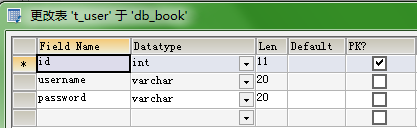
相关表：t\_book，t\_booktype。



 bookNum字段代表这类型图书数量。

新添加两个表：t\_log和t\_user。其字段如下。





### 创建只有一个执行语句的触发器

CREATE TRIGGER 触发器名 BEFORE|AFTER 触发事件

ON 表名 FOR EACH ROW [执行语句]

举例如下：在t\_book表中加入一条数据，然后自动让t\_bookType表中的bookNum字段值+1。执行脚本如下：

**CREATE TRIGGER** trig\_book **AFTER INSERT**

**ON** t\_book **FOR EACH ROW**

**UPDATE** t\_bookType **SET** bookNum=bookNum+1 **WHERE**

new.bookTypeId=t\_booktype.id;

---------------------------------------------------------------

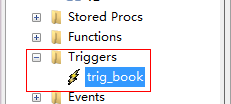
执行脚本的语义解释

在针对t\_book表执行INSERT(用insert语句插入一条数据，这条insert语句要单独调用)操作之后，接着对t\_bookType表的bookNum字段进行更新操作。

执行脚本额外补充知识

**过度变量：new | old**。代表具体的某一条数据，这里的new代表你刚刚插入的那条数据；在【执行语句】是INSERT和UPDATE两种情况下，一般使用new来作为过度变量；如果是DELETE的情况则使用old作为过度变量；new可以解释为刚刚的，old可以解释为以前的。

执行该触发器脚本后，结果显示如下：



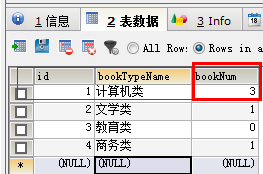
此时，向t\_book表中插入一条数据，对比t\_book，t\_booktype两个表中的数据变化，插入脚本如下：

**INSERT** **INTO** t\_book **VALUES**(NULL,' java设计模式',70,' kaifu.Li ',1);

t\_book表：



t\_booktype表：



可见插入t\_booktype表更新了bookNum字段。

### 创建有多个执行语句的触发器

语法如下：

CREATE TRIGGER 触发器名 BEFORE|AFTER 触发事件

ON 表名 FOR EACH ROW

BEGIN

执行语句列表

END

针对多执行语句的触发器，举例如下：在t\_book表中删除一条数据后，执行如下三个操作，包括更新t\_bookType表中的bookNum字段数量、在t\_log表中加入一条数据、在t\_user表中删除一条数据。

**DELIMITER** |

**CREATE TRIGGER** trig\_book2 **AFTER DELETE**

**ON** t\_book **FOR EACH ROW**

**BEGIN**

**UPDATE** t\_bookType **SET** bookNum=bookNum-1

**WHERE** old.bookTypeId=t\_booktype.id;

**INSERT INTO** t\_log **VALUES**(NULL,NOW(),'在book表里删除了一条数据');

**DELETE FROM** t\_user **WHERE** old.bookTypeId=t\_user.id;

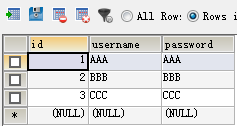
**END**

|

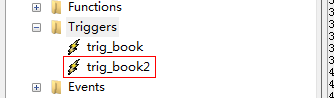
**DELIMITER** ;

**准备工作**

t\_user表中加入数据，如下图



执行上述脚本，创建触发器：trig\_book2。结果如下：

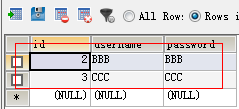


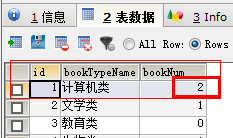
现在调用触发器，删除t\_book表中的一条记录

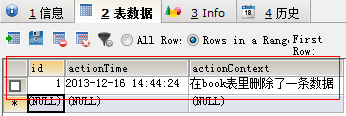
**DELETE** **FROM** t\_book **WHERE** id=8;

执行该脚本后，预期结果应该是t\_user表中的第一条记录被删除；t\_booktype表中第一条记录的bookNum字段变为2；t\_log表中有一条数据被插入。其结果显示如下：

 t\_book表。

 t\_user表 id = 1的记录被删除。

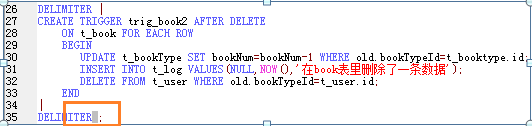
 t\_bookType表。

 t\_log表新增一条记录信息。

**额外注意点 DELIMITER**

delimiter [dɪ'lɪmɪtɚ]  [计] 定界符。

默认情况下，Mysql的默认结束符为 **";"** ，即：delimiter是分号 **;** 。在命令行客户端中，如果有一行命令以分号结束， 那么回车后，mysql将会执行该命令。但有时候，不希望MySQL这么做，因为可能输入较多的语句，且语句中包含有分号。这种情况下，就需要事先把delimiter换成其它符号，如//或$$。这样只有当//出现之后，mysql解释器才会执行这段语句。上面的例子中使用的是“|”。



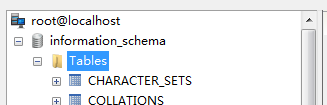
在小海豚中，编辑界面此处要有个空格，也是注意点。

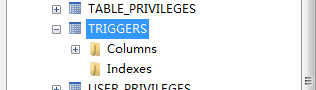
### 查看触发器

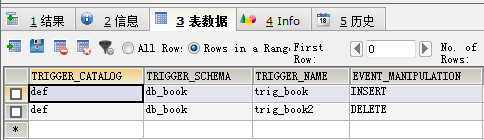
**SHOW TRIGGERS 语句查看触发器信息**



**在trigger表中查看触发器信息**







### 删除触发器

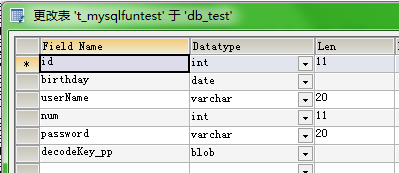
DROP TRIGGER 触发器名；如：**DROP** **TRIGGER** trig\_book ;

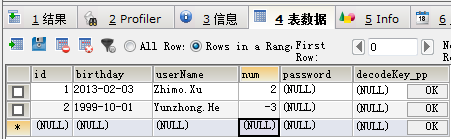
需要注意的是trig\_book后面要加一个空格，再加分号。



## 常用数据库函数

创建一张表：t\_mysqlFunTest，其结构如下：





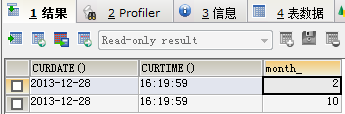
### 日期与时间函数

CURDATE() 返回当前日期； CURTIME() 返回当前时间；MONTH(d) 返回日期d 中的月份值，范围是1~12。

示例如下：

**SELECT** CURDATE(),CURTIME(),MONTH(birthday) **AS** month\_

**FROM** t\_mysqlFunTest;



### 字符串函数

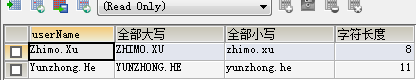
CHAR\_LENGTH(s) 计算字符串s的字符数；UPPER(s) 把所有字母变成大写字母； LOWER(s) 把所有字母变成小写字母。

示例如下：

**SELECT** userName,UPPER(userName) **AS** '全部大写',LOWER(userName)

**AS** '全部小写',CHAR\_LENGTH(userName)

**AS** '字符长度' **FROM** t\_mysqlFunTest;

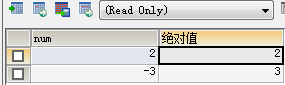


### 数学函数

**1 ABS(x) 求绝对值**

示例如下：

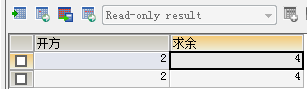
**SELECT** num,ABS(num) **AS** '绝对值' **FROM** t\_mysqlFunTest;



**2 SQRT(x) 求平方根 ；MOD(x,y) 求余**

示例如下：

**SELECT** SQRT(4) **AS** '开方',MOD(9,4) **AS** '求余' **FROM** t\_mysqlFunTest;



### 加密函数

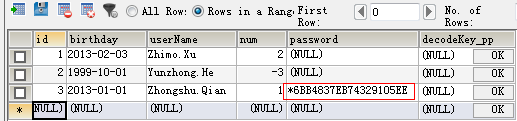
**1 PASSWORD(str) 加密函数**

一般对用户的密码加密不可逆，即加密了不可解密。

示例如下：

**INSERT** **INTO** t\_mysqlFunTest **VALUES**(NULL,'2013-1-1','Zhongshu.Qian',1,

**PASSWORD**('123456'), NULL);



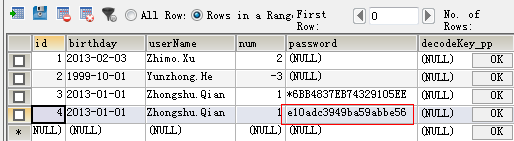
**2 MD5(str) 加密函数**

普通加密不可逆

示例如下：

**INSERT** **INTO** t\_mysqlFunTest **VALUES**(NULL,'2013-1-1','Zhongshu.Qian',1,

MD5('123456') , NULL);



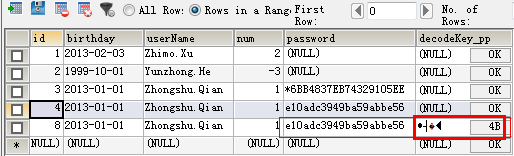
**3 ENCODE(str，pswd\_str) 加密函数**

使用字符串pswd\_str来加密字符串str。加密的结果是一个二进制数，必须使用BLOB类型的字段来保存它。这种加密机制类似于一把锁，符串pswd\_str是钥匙。encode()函数和decode()函数经常一起使用。

对比MD5加密与Encode加密的区别，示例如下：

**INSERT** **INTO** t\_mysqlFunTest **VALUES**(NULL,'2013-1-1','Zhongshu.Qian',1,

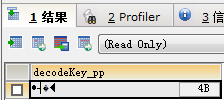
MD5('123456'), ENCODE('abcd','aa'));



**4 DECODE(crypt\_str，pswd\_str) 解密函数**

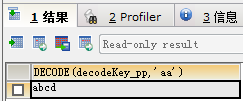
函数可以使用字符串pswd\_str来为crypt\_str解密，当使用普通查询，来查找id=8的记录时，结果显示如下：

**SELECT** decodeKey\_pp **FROM** t\_mysqlFunTest **WHERE** id=8;



当使用decode()方法解密后，查询结果如下：

**SELECT** DECODE(decodeKey\_pp,'aa') **FROM** t\_mysqlFunTest **WHERE** id=8;



### substring函数

查询指定日期到今天，每天晚上八点半到早上八点之间的订单数。

SELECT

create\_time

FROM

db\_matrix.oc\_orderinfo

WHERE

SUBSTRING(create\_time FROM 1 FOR 10) > '2016-03-25'

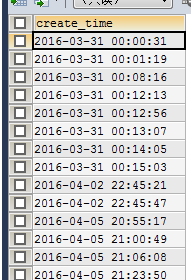
AND (

SUBSTRING(create\_time FROM 12 FOR 8) BETWEEN '20:30:00' AND '24:00:00'

OR

SUBSTRING(create\_time FROM 12 FOR 8) BETWEEN '00:00:00' AND '08:30:00'

) ;



## 存储过程和函数 介绍与初步

### 存储过程和函数介绍

存储过程和函数是在数据库中定义一些SQL语句的集合，然后直接调用这些存储过程和函数来执行已经定义好的SQL语句。存储过程和函数可以避免开发人员重复的编写相同的SQL语句，类似于Java中方法的封装。而且，存储过程和函数是在MySQL 服务器中存储和执行的，可以减少客户端和服务器端的数据传输；存储过程和函数在性能上很高。

**针对存储过程的 语法格式 和 参数含义 如下**：

CREATE PROCEDURE sp\_name(

[proc\_parameter[IN|OUT|INOUT] param\_name type])

[characteristic...] routine\_body

【sp\_name 】参数是 存储过程的名称；

【proc\_parameter】 表示 存储过程的参数列表；

【characteristic】 参数指定存储过程的特性；

【routine\_body】 参数是SQL代码的内容；用BEGIN|END来标识代码的开始和结束。

特别指明如下两个参数列：

**【proc\_parameter】**：中的每个参数由3部分组成。这3部分分别是输入输出类型、参数名称和参数类型。[IN|OUT|INOUT] param\_name type

其中，IN 表示输入参数；OUT表示输出参数；INOUT表示既可以是输入，也可以是输出；param\_name参数是存储过程的参数名称；

type参数指定存储过程的参数类型，该类型可以是MySQL数据库的任意数据类型；

**【Characteristic】**：特性，一种配置策略，参数有多个取值。其取值说明如下：

LANGUAGE SQL：说明routine\_body部分是由SQL语言的语句组成，这也是数据库系统默认的语言。

[NOT] DETERMINISTIC：(deterministic[dɪ,tɝmɪn'ɪstɪk]，确定的) 指明存储过程的执行结果是否是确定的。DETERMINISTIC表示结果是确定的。每次执行存储过程时，相同的输入会得到相同的输出。NOT DETERMINISTIC 表示结果是非确定的，相同的输入可能得到不同的输出。默认情况下，结果是非确定的。

{CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA} ：指明子程序使用SQL语句的限制。(**但测试发现，这些参数在测试的时候并没有明显区别…**)

CONTAINS SQL：表示子程序包含SQL语句，但不包含读或写数据的语句；(contains,n 包含) 默认情况下，系统会指定为CONTAINS SQL；

NO SQL：表示子程序中不包含SQL 语句；

READS SQL DATA：表示子程序中包含读数据的语句；

MODIFIES SQL DATA：表示子程序中包含写数据的语句。(modifies, n. 修改器)

SQL SECURITY {DEFINER|INVOKER}；指明谁有权限来执行。

DEFINER表示只有定义者自己才能够执行；默认情况下，系统指定的权限是DEFINER。

INVOKER表示调用者可以执行。

COMMENT‘string’：注释信息；

**针对存储函数的 语法格式 和 参数含义 如下**：

CREATE FUNCTION sp\_name([func\_parameter[param\_name type, . . .] ])

RETURNS type

[characteristic...] routine\_body

sp\_name：参数是存储函数的名称；

func\_parameter：表示存储函数的参数列表；RETURNStype指定返回值的类型；

characteristic：参数指定存储过程的特性，该参数的取值与存储过程中的取值是一样的；routine\_body：参数是SQL代码的内容，可以用BEGIN...END来标志SQL代码的开始和结束；

func\_parameter 可以由多个参数组成，其中每个参数由参数名称和参数类型组成，其形式如下：

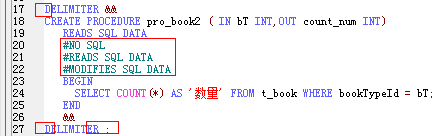
param\_name type

param\_name参数是存储函数的参数名称；

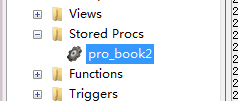
type参数指定存储函数的参数类型，该类型可以是MySQL数据库的任意数据类型；

### 创建简单 存储过程

创建一个简单的存储过程，并测试[CONTAINS SQL|NO SQL|READS SQL DATA| MODIFIES SQL DATA] 这些参数的区别。

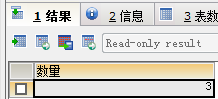


上述代码执行完成后，其显示如下：



此时，调用该存储过程：

**CALL** pro\_book2(1,@total);



**注意点：**

首先，pro\_book2的输出参数count\_num并没使用,**SELECT** COUNT(\*) **INTO** count\_num **FROM** t\_book **WHERE** bookTypeId =bT; 然后**CALL** pro\_book2(1,@total);再**Select** @total; 才正确; 第二点，存储过程的语法非常严格，红方框标识的地方要注意,该有空格的地方要有，不该有的不要有。第三点：@total是一个全局变量，后面会介绍。

### 创建简单 存储函数

创建函数的语句如下：

**DELIMITER** &&

**CREATE FUNCTION** func\_book (bookId **INT**)

**RETURNS** VARCHAR(20)

**BEGIN**

**RETURN** ( **SELECT** bookName **FROM** t\_book **WHERE** id=bookId );

**END**

&&

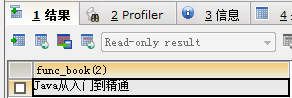
**DELIMITER** ;

执行该语句后，如下图所示：



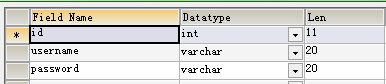
调用该存储函数，结果如下：

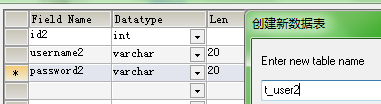
**SELECT** func\_book(2);



## 存储过程和函数 高级特性与实践

存储过程和函数在定义上极为相似，所以归并在一起来总结。涉及表：t\_user、t\_user2：

 t\_user表

 t\_user2表

### 变量定义与赋值

**定义变量**

语法格式如下：

DECLARE var\_name[,...] type [DEFAULT value] (declare |default)示例如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_user()

**BEGIN**

**DECLARE** a,b VARCHAR(20); # declart定义变量

**INSERT INTO** t\_user **VALUES**(NULL,a,b);

**END**

&&

**DELIMITER** ; #注意这里的空格

调用该存储过程：**CALL** pro\_user(); 结果向t\_user表中插入了一条记录。



**变量赋值**

* 方式1语法格式如下：

SET var\_name = expr[,var\_name = expr].. 示例如下所示：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userFuzhi1(IN username VARCHAR(20) ,

IN passwords VARCHAR(20) )

**BEGIN**

**DECLARE** a,b VARCHAR(20) ; # declart定义变量

**SET** a = username , b = passwords;

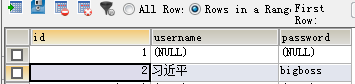
**INSERT INTO** t\_user **VALUES**(NULL,a,b);

**END**

&&

**DELIMITER** ; #注意这里的空格

调用该存储过程：**CALL** pro\_userFuzhi1('习近平','bigboss'); 结果向t\_user表中插入了一条记录。记录中username和password字段是有值的。

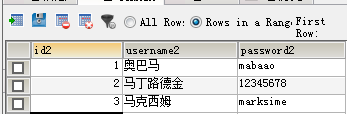


* 方式2语法格式如下：

SELECT col\_name[,...] INTO var\_name[,...]

FROM table\_name WHERE condition

示例：从t\_user2表中取一个数据，插入到t\_user表中。 t\_user2表内容如下：

 脚本代码如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userFuzhi2(IN id\_ INT)

**BEGIN**

**DECLARE** a,b VARCHAR(20) ; # declart定义变量

**SELECT** username2 ,password2 **INTO** a , b **FROM** t\_user2 **WHERE** id2 = id\_;

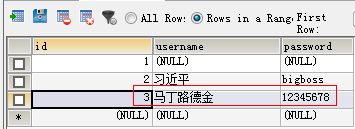
**INSERT INTO** t\_user **VALUES**(NULL,a,b);

**END**

&&

**DELIMITER** ; #注意这里的空格

调用该存储过程：**CALL** pro\_userFuzhi2(2); 结果向t\_user表中插入了一条记录。



方式2的这种变量赋值方式，在开发中貌似并不是很常用，至少目前我用的不多；但对比方式一来讲，还是方式一更加应用的更多更广泛。

### 游标声明使用和关闭

游标，英文：Cursor ['kɝsɚ]；很常用，尤其是在JDBC编程中，应用广泛。查询语句可能查询出多条记录，在存储过程和函数中使用游标来逐条读取查询结果集中的记录。游标的使用包括声明游标、打开游标、使用游标和关闭游标。游标必须声明在处理程序之前，并且声明在变量和条件之后。

**游标的使用步骤：**

* 声明游标

DECLARE cursor\_name CURSOR FOR select\_statement;

* 打开游标

OPEN cursor\_name;

* 使用游标

FETCH cursor\_name INTO var\_name [,var\_name...]； | fetch [fɛtʃ] 读取

* 关闭游标

CLOSE cursor\_name；

**简单游标使用示例**

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userCursorTest (IN \_id INT)

**BEGIN**

**DECLARE** a,b VARCHAR(20) ; # declart定义变量

# 声明游标 cursor\_t\_user2

**DECLARE** cursor\_t\_user2 **CURSOR FOR SELECT** userName2,password2 **FROM** t\_user2 **WHERE** id2=\_id;

**OPEN**  cursor\_t\_user2; #打开游标

**FETCH** cursor\_t\_user2 **INTO** a,b; #读取(使用)游标

**INSERT INTO**  t\_user **VALUES**(NULL,a,b);

**CLOSE** cursor\_t\_user2; #关闭游标cursor\_t\_user2，释放资源

**END**

&&

**DELIMITER** ; #注意这里的空格

调用该存储过程：**CALL** pro\_userCursorTest (3); 结果向t\_user表中插入了一条记录。



### 存储过程函数 与 流程控制

存储过程和函数中可以使用流程控制来控制语句的执行。MySQL 中可以使用IF语句、CASE语句、LOOP语句、LEAVE语句、ITERATE语句、REPEAT语句和WHILE语句来进行流程控制。

**1 IF语句**

IF search\_condition THEN statement\_list

[ELSE IF search\_condition THEN statement\_list ]...

[ELSE statement\_list ]

END IF

示例脚本如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userIf\_Else(IN \_id INT, IN \_username VARCHAR(20))

**BEGIN**

**SELECT** COUNT(\*) **INTO** @num **FROM** t\_user **WHERE** id=\_id;

#@num作为一个全局变量

IF @num>0 THEN **UPDATE** t\_user **SET** username=\_username

**WHERE** id=\_id; #更新

ELSE #t\_user表中不存在该id，则插入一条数据

**INSERT INTO** t\_user **VALUES**(\_id,'川端康成','japan');

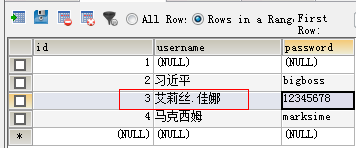
END IF ;

**END**

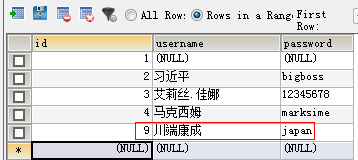
&&

**DELIMITER ;**

调用该存储过程：**CALL** pro\_userIf\_Else (3,’艾莉丝.佳娜’); 结果有两个如果存在id=3的记录，则更新这条记录的名字为艾莉丝.佳娜。如果不存在这个id值，则添加一条记录。这里第一次执行会id=3的记录会变，如下：



结果2将id改成9，**CALL** pro\_userIf\_Else (9,’艾莉丝.佳娜’);



**2 CASE语句**

CASE case\_value

WHEN when\_value THEN statement\_list

[WHEN when\_value THEN statement\_list]...

[ELSE statement\_list]

END CASE

IF ELSE语句和CASE语句和在Java或C#中的功能是一样的，针对CASE语句这里实现和上面IF ELSE语句一样的攻能。其脚本程序如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userCASE(IN \_id INT , IN \_username VARCHAR(20))

**BEGIN**

#@num作为一个全局变量

**SELECT** COUNT(\*) **INTO** @num **FROM** t\_user **WHERE** id=\_id;

CASE @num

WHEN 0 THEN **INSERT INTO** t\_user VALUES(\_id,'南怀瑾','123456');

WHEN 1 THEN **UPDATE** t\_user **SET** username=\_username

**WHERE** id=\_id;

ELSE **INSERT INTO** t\_user

**VALUES**(\_id,'procedure exception','warning!');

END CASE

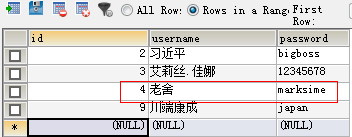
**END**

&&

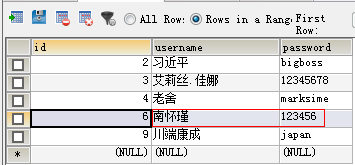
**DELIMITER ;**

调用该存储过程：**CALL** pro\_ userCASE (4,’ 老舍’); 同样结果有两个，分别如下：

情况1，将马克西姆更新成老舍。



情况2，插入一条数据，南怀瑾。



**3 LOOP、LEAVE语句 循环跳出组合 循环的一种**

LOOP语句可以使某些特定的语句重复执行，实现一个简单的循环。但是LOOP语句本身没有停止循环的语句，必须遇到LEAVE语句等才能停止循环，类似于跳出循环。LOOP语句的语法的基本形式如下：

[begin\_label：] LOOP

Statement\_list

END LOOP[end\_label]

LEAVE语句主要用于跳出循环控制。语法形式如下：

LEAVE label

脚本代码如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userLOOP\_LEAVE(IN totalNum INT)

**BEGIN**

ASDF:LOOP # ASDF 作为 begin\_label

**SET** totalNum=totalNum-1;

**IF** totalNum=2 **THEN** LEAVE ASDF ; #跳出循环的条件

**ELSE** **INSERT INTO** t\_user **VALUES**(totalNum,'测试1',totalNum\*10);

**END IF** ; #注意空格

END LOOP ASDF ; #注意空格

**END**

&&

**DELIMITER** ; #注意空格

此脚本的目的是不断的向表中加入测试数据，调用该存储过程，结果如下：

**CALL** pro\_userLOOP\_LEAVE(6);



从结果来看，当totalNum=2的时候，跳出了整个循环体，这证明LEAVE关键字类似于Java中的break，即：全部跳出。下面的iterate类似于continue。

**4 ITERATE语句**

ITERATE (iterate ['ɪtərət] vt. 迭代；重复)语句也是用来跳出循环的语句。但是，ITERATE语句是跳出本次循环，然后直接进入下一次循环。基本语法：ITERATE label;

脚本代码如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userITERATE (IN totalNum INT)

**BEGIN**

ASDF:LOOP # ASDF 作为 begin\_label

**SET** totalNum=totalNum-1;

**IF** totalNum=0 **THEN** **LEAVE** ASDF ; #注意ELSEIF是不能分开的

ELSEIF totalNum=3 THEN ITERATE ASDF ; #类似Java continue

**ELSE** **INSERT INTO** t\_user **VALUES**(totalNum,'测试1',totalNum\*10);

**END IF** ;

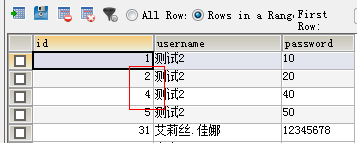
END LOOP ASDF ;

**END**

&&

**DELIMITER** ;

从此脚本中提到2点注意：1.ELSEIF不可分开写，会报错；2.ITERATE类似于Java中的continue。调用该存储过程**CALL** pro\_userITERATE(6); 结果如下：



结果显示跳出了第三个变量。

**5 REPEAT语句 循环的一种**

REPEAT(repeat [rɪ'pit] vt. 重复；复制 )语句是有条件控制的循环语句。当满足特定条件时，就会跳出循环语句，类似do{ }…while ()。REPEAT语句的基本语法形式如下：

[begin\_label :] REPEAT

Statement\_list

UNTIL search\_condition | until [ən'tɪl] 直到…时

END REPEAT [end\_label]

脚本代码如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userREPEAT (IN totalNum INT)

**BEGIN**

REPEAT

**SET** totalNum=totalNum-1;

**ELSE** **INSERT INTO** t\_user **VALUES**(totalNum,'测试repeat ',totalNum);

UNTIL totalNum=1 #值为1的时候，将其执行完，然后跳出

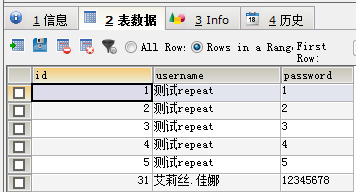
END REPEAT;

**END**

&&

**DELIMITER** ;

调用该存储过程**CALL** pro\_userREPEAT (6); 结果如下：



**6 WHILE DO语句**

纯粹的while() do{ }。其语法格式如下：

[begin\_label :] WHILE search\_condition DO

Statement\_list

END WHILE [end\_label]

脚本代码如下：

**DELIMITER** &&

**CREATE PROCEDURE** pro\_userWHILE (IN totalNum INT)

**BEGIN**

WHILE totalNum>0 DO

**ELSE** **INSERT INTO** t\_user **VALUES**(totalNum,'测试while ',totalNum);

**SET** totalNum=totalNum-1;

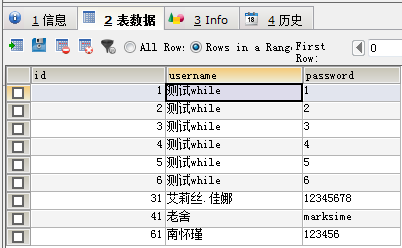
END WHILE;

**END**

&&

**DELIMITER** ;

调用该存储过程**CALL** pro\_userWHILE (6); 结果如下：

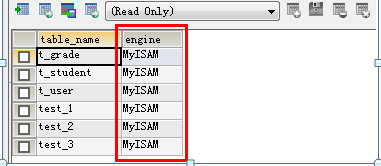


### 高级特性：游标与流程控制

举例：批量修改数据库表的引擎类型。创建数据库testaaa首先查看数据库表的类型：

**SELECT** table\_name,ENGINE **FROM** information\_schema.TABLES

**WHERE** table\_schema='testaaa' ;



数据库中包含的数据表名称和数据表引擎类型如上图所示，现在需要把engine列变成InnoDB，创建脚本如下：

**DELIMITER** &&

**CREATE PROCEDURE** changeEngineType(IN DB\_NAME VARCHAR(32),

IN Engine\_type VARCHAR(16)) # 数据库名称 引擎类型

**BEGIN**

**DECLARE** done INT DEFAULT 0; # 游标的标志位

**DECLARE** TB\_NAME VARCHAR(64); # 数据库的表名

**DECLARE** COMMAND VARCHAR(64); # 更改数据库引擎的命令

**DECLARE** cur1 **CURSOR FOR** **SELECT** table\_name

**FROM** information\_schema.TABLES **WHERE** table\_schema=DB\_NAME ;

**DECLARE** CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

#错误定义，标记循环结束

**OPEN** cur1;

**REPEAT** # 循环开始

**FETCH** cur1 **INTO** TB\_NAME;

**IF NOT** done **THEN**

**SET** COMMAND=CONCAT('ALTER TABLE ',DB\_NAME,'.',

TB\_NAME,' ENGINE = ',Engine\_type);

# 拼更改引擎的命令 ALTER TABLE test\_3 ENGINE = MYISAM ;

**SET** @E=COMMAND;

PREPARE stmt1 FROM @E;

EXECUTE stmt1; # 执行命令

DEALLOCATE PREPARE stmt1; # 释放对象 deallocate prepare

**END IF**;

**UNTIL** done **END REPEAT**; # 循环结束

**CLOSE** cur1; # 关闭游标

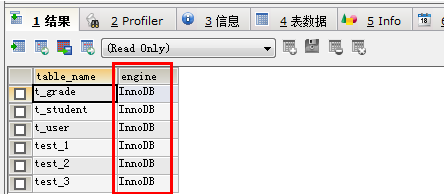
**END**;

&&

**DELIMITER** ;

执行完成该脚本后，调用该脚本如下：然后再次查看其引擎类型，如下图所示，已经改变。

**CALL** changeEngineType('testaaa','InnoDB');



## 附录：经验积累

### 查询区分大小写 - 建表时约束

CREATE TABLE user(

id int(14) primary key,

username VARCHAR(255) binary,

password VARCHAR(255) binary,

funcStatu int(4)

);

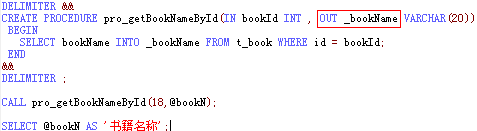
### 16.2 存储过程简单示例与调用

说明：这个例子弥补了14.2示例的不足。当一个存储过程包含返回值的时候我们应该如何调用。

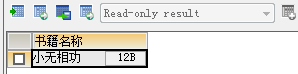
示例表数据如下：



存储过程脚本如下设计：



结果：



### 16.3 全局变量

// TODO

### 16.4 Java和MySQL数据类型对应一览

|  |  |  |
| --- | --- | --- |
| MySQL Type Name | Column Return Value | Returned as Java Class |
| BIT(1) | BIT | java.lang.Boolean |
| BIT( > 1) | BIT | byte[] |
| TINYINT | TINYINT | java.lang.Integer |
| BOOL, BOOLEAN | TINYINT | java.lang.Integer |
| smallint | smallint [unsigned] | java.lang.Integer |
| mediumint | mediumint[unsigned] | java.lang.Integer,  if unsigned java.lang.Long |
| int|Integer | int|Integer | java.lang.Integer,  if unsigned java.lang.Long |
| bigint | bigint[unsigned] | java.lang.Long,  if unsigned java.math.BigInteger |
| float | float | java.lang.Float |
| double | double | Java.lang.Double |
| decimal | decimal | java.math.BigDecimal |
| date | date | java.sql.Date |
| datetime | datetime | java.sql.Timestamp |
| timestamp | timestamp | java.sql.Timestamp |
| time | time | java.sql.time |
| year | year | If year Is Date Type configuration  property is set to false, then the  returned object type is java.sql.  Short. If set to true (the default), then the returned object is of type java.sql.Date with the date set to January 1st, at midnight. |
| char | char | java.lang.String |
| varchar | varchar | java.lang.String |
| binary | binary | byte[] |
| varbinary | varbinary | byte[] |
| TinyBlob | TinyBlob | byte[] |
| tinytext | varchar | java.lang.String |
| blob | blob | byte[] |
| text | varchar | java.lang.String |
| MediumBlob | MediumBlob | byte[] |
| mediumtext | varchar | java.lang.String |
| LongBlob | LongBlob | byte[] |
| longtext | varchar | java.lang.String |
| enum | char | java.lang.String |
| set | char | java.lang.String |

### 16.5 TODO

