数据库高级应用

视图（view）

视图：一种虚拟的表，它的行为和真实的表一样，但并不包含真实的数据。你可以这样理解：视图是用真实的表以及其它的视图定义出来的“假”数据表，用来查看表的数据，这样可以从繁杂的查询语句解放出来，不必关心表与表之间的复杂联系。

PS：给出一个产品名字：“HTC T528d”，请查询买过它的用户资料。

```
SELECT customers.cust_name AS '客户姓名',
customers.cust_contact AS '客户联系名'
FROM customers
INNER JOIN orders
ON orders.cust_id = customers.cust_id
INNER JOIN orderitems
ON orderitems.order_num = orders.order_num
INNER JOIN products
ON products.prod_id = orderitems.prod_id
WHERE products.prod_name = 'HTC T528d';
```

分析上述查询：

输入条件：产品的名字：HTC T528d；

输出结果：购买过它的用户名与联系名；

products <-> orderitems <-> orders <-> customers

对于使用者来说，只关心***输入条件与输出结果***，如果对真实表操作，使用者必须掌握表之间的联结关系，才能写出上述联结查询关系。那么，有没有办法使得这个查询变得简答？答案是有的。我们可以把这个查询包装成一个 ***productscustomers*** 的视图（虚拟表），轻松的检索出我们需要的结果。

PS：

```
SELECT cust_name, cust_contact FROM productscustomers
WHERE prod_name = 'HTC T528d';
```

使用视图

创建视图

创建视图：

CREATE VIEW 视图名 AS 视图语句

PS：

注意，以上述示例来说，最后视图 productscustomers 一共出现了

3 个列名：cust_name、cust_contact、prod_name；所以，在视图语句中，这三个是必须出现的，否则，视图会出现无法识别列名的错误。

```
CREATE VIEW productscustomers AS
SELECT customers.cust_name,
customers.cust_contact,
products.prod_name
FROM customers
INNER JOIN orders
ON orders.cust_id = customers.cust_id
INNER JOIN orderitems
ON orderitems.order_num = orders.order_num
INNER JOIN products
ON products.prod_id = orderitems.prod_id;
```

使用视图

视图创建后，你可以把这个虚拟表当成真正的表来用。现在：productscustomers 就是一个新表。

PS：

```
SELECT cust_name AS '客户姓名',
cust_contact AS '客户联系名'
```

```
FROM productscustomers
WHERE prod_name = 'HTC T528d';
```

查看创建的视图

查看创建的视图：

```
SHOW CREATE VIEW 视图名;
```

删除视图

删除视图：

```
DROP VIEW 视图名;
```

更新视图

当视图创建后，发现需要修改，可以更新视图，两种方式：

1、 先 DROP（删除）再重新 CREATE（创建）一次。

2、 使用 CREATE OR REPLACE VIEW 视图名 AS 视图语句

（1）如果更新的视图不存在，将创建一个新的视图。

（2）如果更新的视图存在，AS 后面的视图语句将覆盖原来的视图语句，更新视图。

为什么要使用视图

前面的例子我们已经看出使用视图的好处，下面列出视图使用的常见应用。

1、 重用 SQL 语句

2、 简化复杂的 SQL 的使用，使用者可以不必知道它的细节。

3、 视图公开只需要使用的表的部分（列），而不是表的全部。

4、 保护数据。视图可以通过 DBA 设定访问权限，而不是表的访问权限。

5、 更改数据的返回样式和表示：视图可以返回和原本表不同的样式和表示。

视图的规则和限制

1、 视图名唯一。

2、 视图的数目通常没有限制。

3、 创建视图需要足够的访问权限，访问视图也可以设置权限。

4、 视图可以嵌套，联结其它的表或视图，形成新的联结查询，但通常不推荐这么做。

5、 ORDER BY 可以在视图后使用，但如果该 ORDER BY 语句在创建视图中的语句已经有，那么视图语句中的 ORDER BY 将被视图后的覆盖。

6、 视图创建后，你可以将它当成真实的表来使用，执行 SELECT、过滤、排序等操作，但请注意它并不是真实的表，它的数据都来自于真实的表。如果真实表的数据发生变化，视图获取的数据也将发生变化。

7、 视图仅仅是用来查看存储在别处数据的一种手段和措施。视图甚至能添加和更新数据，但存在一些限制，通常也不推荐这么做，视图一般只用于数据的检索，增加、删除、更新通常不通过视图完成。

视图的应用

简化复杂的联结查询

使用视图格式化检索数据

PS：

```
SELECT CONCAT(RTRIM(vend_name), ' (', RTRIM(vend_country), ')')
AS '设备厂商 (国家|地区)'
FROM vendors
ORDER BY vend_name;
```

像这种需要格式化输出最后检索的信息，可以使用视图来完成。

PS：

```
CREATE OR REPLACE VIEW vendorslocations AS
SELECT       CONCAT(RTRIM(vend_name),       '       (',
RTRIM(vend_country), ')')
AS '设备厂商 (国家|地区)'
FROM vendors
ORDER BY vend_name;

SELECT * FROM vendorslocations;
```

使用视图过滤不需要的数据

比如：有一个应用需要用到每个客户的邮箱地址，以便程序能自动给该客户发一封邮件，这时候，需要把没有邮箱地址的用户数据过滤掉。如果使用视图预处理这一切，就不容易出错。

PS：

```
CREATE OR REPLACE VIEW customersemaillist AS
SELECT cust_id, cust_name, cust_email
FROM customers
WHERE cust_email IS NOT NULL;

SELECT * FROM customersemaillist;
```

注意：如果视图语句中有 WHERE 条件，使用的时候再加 WHERE 条件，那么，最终结果将是两个 WHERE 条件的组合。

使用视图计算字段

例如：给一个订单号，想直接得到这个订单所有物品的花费价格（个数与单价的乘积），可以使用视图来简化查询。

PS：

```
SELECT prod_id, quantity, item_price, (quantity * item_price) AS 'expanded_price'
FROM orderitems
WHERE order_num = 1;
```

PS：

```
CREATE OR REPLACE VIEW orderitemsexpanded AS
SELECT order_num, prod_id, quantity, item_price, (quantity * item_price) AS 'expanded_price'
FROM orderitems;

SELECT * from orderitemsexpanded WHERE order_num = 1;
SELECT sum(expanded_price) AS '总价' FROM orderitemsexpanded WHERE order_num = 1;
```

对视图的增加、删除、更新数据

到目前为止，我们所有的例题针对视图都是 SELECT 语句的使用，那么是否可以对视图增加、删除、更新数据。答案是肯定的。视图可以 INSERT、UPDATE、DELETE。对视图的增、删、改实际上是对其关联的基表（真实表）的数据的操作。

但是，遗憾的是，这一切操作是有限制的，视图定义语句中如果有以下操作，则不能通过对视图增、删、改：

1、 分组（GROUP HAVING 出现）

2、 联结

3、 子查询

4、 并操作

5、 聚集函数

6、 DISTINCT

7、 计算列

可以看出，视图的操作限制多多，这就严重的虚弱了对视图增、删、改。所以，在绝大部分情况下，视图只应该应用在 SELECT 中，而避免 INSERT、UPDATE、DELETE。

存储过程（stored procedure）

到目前为止，我们的所有 SQL 学习都是针对一个表或多个表执行的单条语句，但实际并非所有的操作都这么简单，有时候会有一个完整的操作需要多条语句的配合才能完成，并且，多条语句的执行也不是固定的，它有可能需要前面的语句结果变化。

存储过程简单来讲，是为以后的使用而保存的一条或多条 SQL 语句的集合，可以将其视为批处理文件，虽然它的作用不仅仅限于批处理功能。它们存放在 DB 中，然后根据调用者的情况，做出不同的执行，将结果返回。

知识扩展：

1、 视图 view ：虚拟表，复杂的查询变成简单查询。

2、 存储过程 PROCEDURE：多条 SQL 命令集。

3、 触发器 trigger：一条 SQL 操作去触发另外一个 SQL 操作。

4、 存储函数 function：自定义的 SQL 函数。

5、 事件 event：在一段时间或周期，自动在 DB 上执行一系列 SQL，维护 DB 数据。

创建存储过程

存储过程是一个预先编辑好的预处理文件，你需要先创建，然后再调用。

```
   CREATE PROCEDURE 存储过程名(参数1，参数2，……)
BEGIN
   存储过程主语句
END;
```

PS：创建一个返回产品平均价格的存储过程。

```
CREATE PROCEDURE productpricing()
BEGIN
   SELECT AVG(prod_price) AS priceavg
   FROM products;
END;
```

注意：该存储过程没有参数，但()要写，BEGIN 是开始标志，END 是结束标志。

调用存储过程

SQL 称存储过程的执行为调用，因此，SQL 执行存储过程的语句的关键字是 CALL。CALL 接受存储过程的名字以及给它的参数。

CALL 存储过程名(参数 1，参数 2，……);

PS:

```
CALL productpricing();
```

注意：调用存储过程，也需要跟()。

删除存储过程

存储过程创建后就保留在数据库上，可以一直调用。存储过程也可以被删除。

DROP PROCEDURE 存储过程名;

注意：存储过程名后面不跟()。

PS:

```
DROP PROCEDURE productpricing;
```

PS:

```
DROP PROCEDURE IF EXISTS productpricing;
```

查看存储过程的创建

可以使用 SHOW 语句来查看存储过程创建的信息。

SHOW CREATE PROCEDURE 存储过程名;

另外：SHOW PROCEDURE STATUS;可以查看所有存储过程创建的信息。

存储过程的参数

上述例题是一个很简单的存储过程，简单返回 SELECT 的结果。通常，存储过程并不直接显示结果，而是把结果返回给你指定的参数。参数是一个变量，变量是一个在内存中特殊的位置，用来存储临时数据。

〔例题〕

PS：创建存储过程

```
CREATE PROCEDURE productpricing(
   OUT p1 DECIMAL(8,2),
   OUT p2 DECIMAL(8,2),
   OUT p3 DECIMAL(8,2)
)
BEGIN
   SELECT MIN(prod_price) INTO p1
   FROM products;
```

```
    SELECT MAX(prod_price) INTO p2
    FROM products;
    SELECT AVG(prod_price) INTO p3
    FROM products;
END;
```

PS：调用存储过程

```
CALL productpricing(
    @priceMin,
    @priceMax,
    @priceAvg
);
```

PS：查看存储过程返回参数值

```
SELECT @priceMin;
SELECT @priceMax;
SELECT @priceAvg;
```

or：

```
SELECT @priceMin, @priceMax, @priceAvg;
```

注意：上述例题的**返回参数是三个变量**，存储过程返回的不是结果集，即不能是一个多行多列的结果，而是一个值才能赋给输出参数。并且，OUT 修饰已经表明此参数是一个返回值（输出参数）。在 MySQL 中还有 IN：输入参数；INOUT：输入输出参数。INTO 关键字是将输出值给输出变量的。

〔例题〕

PS：创建存储过程

```
CREATE PROCEDURE ordertotal(
   IN onumber INT,
   OUT ototal DECIMAL(8,2)
)
BEGIN
   SELECT sum(item_price * quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO ototal;
END;
```

PS：调用存储过程

```
CALL ordertotal(1, @total);
```

调用要给出一个输入参数和输出参数。

PS：参考输出参数值

```
SELECT @total;
```

高阶存储过程

从前面两个例题来看，存储过程还是太简单了，基本上都是封装

一个 SELECT 语句。但这不仅仅是存储过程的全部，存储过程能完成复杂的业务规则和智能处理，只有完成这一切你才会发现使用存储过程的好处。

〔例题〕

场景：你需要完成一个订单统计，但要对统计的金额附加上一定的营业税款，但是，税款又不是对每个订单有效（有可能有，有可能无）。最后得出这个订单统计，**有含税的也有不含税**的情况。

PS：创建**高阶存储过程**

```
CREATE PROCEDURE ordertotaltax(
   IN onumber BIGINT, -- 订单编号
   IN taxable BOOLEAN, -- 是否加税
   OUT ototal DECIMAL(8,2) -- 最后统计的订单金额
)
BEGIN
   DECLARE total DECIMAL(8,2); -- 声明一个订单总金额变量
   DECLARE taxrate INT DEFAULT 6; -- 声明一个税收百分比，默认是 6%

   SELECT SUM(item_price * quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO total;
```

```
   IF taxable THEN
     SELECT total + (total * taxrate / 100) INTO ototal;
   END IF;
      SELECT total INTO ototal;
END;
```

PS：调用存储过程和参考存储过程返回参数值

```
CALL ordertotaltax(1, 0, @total);
SELECT @total;
CALL ordertotaltax(2, 1, @total);
SELECT @total;
```

注意：BOOLEAN 值，0 表是假，1 表示真。（在编程中，通常 0 为 false，非 0 为 true）IF、THEN、END IF 是 MySQL 中逻辑语句，这种语句被称为复合 SQL 语句。有关复合 SQL 语句的详细使用请参阅相关资料。

〔示例〕

PS：

```
CREATE PROCEDURE greeting()
BEGIN
   DECLARE user CHAR(50) CHARACTER SET utf8;
```

```
   SET user = (SELECT CURRENT_USER());
   IF INSTR(user, '@') > 0 THEN -- 判断当前登录用户名是否包含@符号
      SET user = SUBSTRING_INDEX(user,'@', 1); -- 截取@前面的字符串
   END IF;
   IF user = '' THEN -- 匿名用户登录，没有登录名
      SET user = 'earthing';
   END IF;
   SELECT CONCAT('Greeting, ', user, '!') AS greeting;
END;

CALL greeting;
```

〔示例〕

PS：

```
drop procedure if exists pr_param_inout;

create procedure pr_param_inout
(
   inout id int
)
```

```
begin
   select id as id_inner_1;  -- id 值为调用者传进来的值

   if (id is not null) then
      set id = id + 1;

      select id as id_inner_2;
   else
      select 1 into id;
   end if;

   select id as id_inner_3;
end;

SET @id = 10;
CALL pr_param_inout(@id);
SELECT @id;
```

触发器（trigger）

我们前面学过的 SQL 语句都是需要调用的时候执行的，这被称为一个被执行点，一个被触发事件。那么，有没有 SQL 语句在一种条件

或事件下（关联）被自动执行呢？下面列举一些场景应用：

1、 在向 customers 表插入一条数据的时候，检查其邮箱地址是否合法。

2、 如果产品有一个库存量字段，每当生成订单后，这个库存量要自动减少为相应的值。

3、 在删除一张表的一条数据的时候，自动备份这条数据到其它副本表中去。

上面这一切场景应用都基于一个共同条件：表中数据发生一定的变化时候，自动触发其它 SQL 命令的执行以达到自动变化。这就是触发器。

触发器定义：MySQL 响应以下任意语句而自动执行的其它 MySQL 语句。

1、 delete

2、 insert

3、 update

其它语句不支持触发器。

创建触发器

创建触发器，需要给出以下 4 个方面的信息代码：

1、 一个唯一的触发器名。

2、 触发器关联的表。

3、 触发器响应的活动（delete、insert、update）。

4、 触发器应该在何时被执行（处理之前还是处理之后）。

触发器创建的关键字：CREATE TRIGGER 触发器名

〔示例〕

PS：

```
DROP TRIGGER IF EXISTS newproduct;


CREATE TRIGGER newproduct AFTER INSERT ON products
FOR EACH ROW
BEGIN
SELECT  MAX(prod_id) FROM products  INTO @newID ;
END;


SELECT @newID;
INSERT INTO products
(prod_name, prod_price, prod_desc, vend_id)
VALUES
('三星 S5830I ', 858, 'WCDMA|GSM', 2);
SELECT @newID;
```

解释：CREATE TRIGGER 为创建触发器的关键字，newproduct 为触发器的名字。触发器可以在一个操作发生之前或之后，此例为 AFTER INSERT 表示插入数据成功之后。触发器还指定了 FOR EACH ROW 每插入一行新数据。

注意：

1、 只有表才支持触发器，视图不支持。

2、 触发器是按照每个表每个事件每次地定义，所以，每个表每个事件每次只允许 1 个触发器。那么，每个表最多支持 6 个触发器：每条 INSERT、UPDATE、DELETE 的之前或之后。

3、 单一的触发器不能与多个事件或多个表相关联，如果你需要一个对 INSERT 和 UPDATE 操作的触发器，则应该定义 2 个触发器。

4、 如果 BEFORE 触发器失败，则 MySQL 将不再执行请求操作。此外，如果 BEFORE 触发器或语句本身失败，MySQL 将不再执行 AFTER 触发器（如果有）。

删除触发器

PS：DROP TRIGGER 触发器名;

PS：DROP TRIGGER IF EXISTS 触发器名;

使用触发器

INSERT 触发器

INSERT 触发器分为在 INSERT 语句执行前或执行后，要点如下：

1、 在 INSERT 触发器代码内部，可以引用一个名为 NEW 的虚拟表，通过它可以访问被插入的行。

2、 在 BEFORE INSERT 触发器中，NEW 中的值也可以被更新（即允许更改被插入的值）。

〔示例〕

PS：

```
DROP TRIGGER IF EXISTS neworder;
CREATE TRIGGER neworder AFTER INSERT ON orders
FOR EACH ROW
BEGIN
   SELECT NEW.order_num INTO @neworderid;
   SELECT NEW.order_date INTO @neworderdate;
   SELECT NEW.cust_id INTO @newcustid;
END;

INSERT INTO orders (order_date, cust_id)
VALUES(NOW(), 2);
SELECT @neworderid, @neworderdate, @newcustid;
```

〔示例〕

```
DROP TRIGGER IF EXISTS neworder;
CREATE TRIGGER neworder BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
   IF NEW.cust_id = 3 THEN
      SET NEW.cust_id = 1;
   END IF;
END;

INSERT INTO orders (order_date, cust_id)
VALUES(NOW(), 3);
SELECT * FROM orders ORDER BY order_num DESC LIMIT 1;
```

BEFORE 和 AFTER 的区别

BEFORE 通常用于数据的验证和净化（目的在于检查插入数据是否正确，以便及时修改）。

AFTER 通常用于插入数据后，立即获得该数据值，以便其它使用。

BEFORE 和 AFTER 的规则也适合 UPDATE 触发器。

〔示例〕

PS：

```
DROP TRIGGER IF EXISTS newdepartmentandemployee;
CREATE TRIGGER newdepartmentandemployee AFTER INSERT ON department
FOR EACH ROW
BEGIN
   SELECT NEW.id INTO @newdepartmentid;
END;

INSERT INTO department(name) VALUES ('开发部');
INSERT INTO employee(name, fk_department) VALUES ('TMAC', @newdepartmentid);
```

DELETE 触发器

DELETE 触发器同样分为 DELETE 语句执行之前或执行以后，要点如下：

1、 在 DELETE 触发器代码内部，可以引用一个名为 OLD 的虚拟表，访问被删除的行。

2、 OLD 表中的值全部是只读，不能更新。

〔示例〕

PS：

```
CREATE TRIGGER orderbackup BEFORE DELETE ON orders
FOR EACH ROW
BEGIN
      INSERT INTO orders_backup (order_num, order_date,
cust_id)
      VALUES      (OLD.order_num,      OLD.order_date,
OLD.cust_id);
END;

DELETE FROM orders WHERE order_num = 7;
```

UPDATE 触发器

UPDATE 触发器同样分为 UPDATE 执行之前或执行之后，要点如下：

1、 在 UPDATE 触发器代码中，可以引用一个名为 OLD 的虚拟表访问更新前的值，可以引用一个名为 NEW 的虚拟表访问更新后的值。

2、 在 BEFORE UPDATE 触发器中，NEW 中的值仍然可以被更改（即可以更改将要更新的值）。

3、 OLD 中的值全部是只读，不能更新。

〔示例〕

```
CREATE TRIGGER upperemployee BEFORE UPDATE ON employee
FOR EACH ROW
BEGIN
      SET NEW.name = UPPER(NEW.name);
END;

UPDATE employee SET name = 'ivy' WHERE id = 3;
```

〔示例〕

```
DROP PROCEDURE  IF EXISTS getusername;
CREATE PROCEDURE getusername(
   OUT username CHAR(20)
)
BEGIN
   SELECT CURRENT_USER() INTO username;
END;

DROP TRIGGER IF EXISTS upperemployee;
CREATE TRIGGER upperemployee BEFORE UPDATE ON employee
FOR EACH ROW
BEGIN
```

```
--       SET NEW.name = UPPER(NEW.name);
      CALL  getusername(@username);
      SET    NEW.name    =    CONCAT(UPPER(@username),
'_',NEW.name) ;
END;

UPDATE employee SET name = 'ivy' WHERE id = 3;
```

MySQL 触发器小结

1. 与其它 DBMS 相比，MySQL 5 中对触发器的支持比较弱，在未来版本可能会升级功能，请关注每个新版本的新功能。
2. 触发器的执行是自动的，只要是 INSERT、UPDATE、DELETE 语句执行就能触发触发器工作。
3. 可以应用触发器来保证数据的一致性（如：大小写，格式等），这种由触发器来完成的功能是透明的，也总是会执行的，与客户机应用无关。
4. 触发器是一种非常有用的审计跟踪。使用它，可以轻松的做备份，移植，更改工作。

存储函数（stored function）

存储函数又名自定义函数。存储函数像 MySQL 内建函数一样，定

义一个函数直接使用，下面列举一道例题：

PS：

```
DROP FUNCTION IF EXISTS ORDER_COUNT;
CREATE FUNCTION ORDER_COUNT(
   startdatetime datetime
)
RETURNS INT
BEGIN
   RETURN(SELECT count(*) FROM orders WHERE order_date
BETWEEN startdatetime AND NOW());
END;

SELECT ORDER_COUNT('2010-10-10 12:00:00');
```

事件（event）

在 MySQL 5.1.6 版本后，提供了一个事件调度器，其作用是我们可以通过它把数据库操作安排在预订的时间执行。事件是一个与时间表相关联的存储程序，时间表定义时间发生的时间、次数以及何时停止（消失）。

事件非常适合用来执行无人看守的系统关联任务、定期数据汇总、定期清理失效数据、日志数据。

首先，你可以使用以下语句查看你的 DB 是否开启事件调度器，默认安装后，事件调度器是关闭的。

```
SHOW VARIABLES LIKE 'event_scheduler';
```

然后，你要打开它，需要修改 MySQL 安装根目录下的 my.ini 文件，在[mysqld]下面加上：

```
event_scheduler=on
```

重启服务器，然后再打开 MySQL 连接，运行刚刚的命令查看。

另外、在 DB 运行的时候也可以通过下面两个命令来开|关事件调度器，但是，它们在重启服务器后，还是以 my.ini 配置文件里面的选项为准。

```
SET GLOBAL event_scheduler = OFF;
SET GLOBAL event_scheduler = ON;
```

〔示例〕

场景：假设你有一个表 web_access，里面记录了网络用户登录的信息。如果你不想这个表的数据越来越多，你可以创建一个事件来清理它们。现在要这个事件每隔 4 小时执行一次，把超过一天的数据清掉，我们来定义事件。

PS：

```
CREATE EVENT expire_web_access
ON SCHEDULE EVERY 4 HOUR
DO
   DELETE FROM web_access
      WHERE last_visit < NOW() - INTERVAL 1 DAY;
```

索引

用于加快查询的技术有很多，其中最重要的是索引。

索引的工作原理

一个没有索引的表就是一个无序的数据行集合。如下：

company_num	ad_num
14	48
23	49
17	52
13	55
23	62
23	63
23	64
13	77
23	99

14	101
13	102
17	119

如果现在想要找到某个公司（company_num）的广告（ad_num），就必须从表的第一行开始搜索，一行一行的条件匹配，直到搜索完毕才能得出结果集。如果，现在给这个表的 company_num 加一个索引。

索引	company_num	ad_num
13	14	48
13	23	49
13	17	52
14	13	55
14	23	62
17	23	63
17	23	64
23	13	77
23	23	99
23	14	101
23	13	102
23	17	119

现在，我们要找寻公司编号 company_num 是 13 的所有行。搜索就会搜索索引，而不是内容，很容易找到 3 个连续的索引，通过索引马上找到 3 行数据，当我们再向下读取的时候，发现是 14 的索引，立即知道所对应的数据一定不匹配了，就停止向下搜索了。由此可见、索引可以提高搜索效率的一个原因就是我们可以知道匹配数据行在什么位置，从而跳过其余部分。

索引分类

普通索引（Normal）

这是基本的索引，没有什么限制，选项类型是 Normal。

```
PS：CREATE INDEX 索引名 ON 表名(列名);
PS：ALTER TABLE 表名 ADD INDEX 索引名 (列名);
PS：
CREATE TABLE 表名(
   name VARCHAR(50),
   INDEX 索引名 (列名)
);
```

唯一索引（Unique）

唯一索引要求索引的列的值不能重复，但允许有 NULL 值。

唯一索引的创建方式和上面的一样，只是加上 UNIQUE 的修饰，如：

```
CREATE UNIQUE INDEX 索引名 ON 表名(列名);
```

用户表：登录名、身份证、学号等等唯一。

主键索引（Primary Key）

一个表的一列被指定为主键后，会自动创建主键索引，主键索引要求该列值不能为 NULL，当然值也不能重复，它是一种特殊的唯一索引。该索引为指定主键时候建立。

全文索引（FullText）

全文索引是为了配合全文搜索功能而特有的。全文索引的表的引擎应该是 MyISAM，MySQL 默认引擎是 InnoDB；加上全文索引的列必须是 CHAR、VARCHAR、TEXT 类型，其它的类型无效。全文索引的建立和前面一样，加上 FULLTEXT 修饰，例如：

```
CREATE FULLTEXT INDEX 索引名 ON 表名(列名);
```

多列（组合）索引

除了为一列可以加上索引，还可以同时为多列加上一个索引，称为多列（组合）索引。

```
ALTER TABLE 表名 ADD INDEX 索引名 (列名1, 列名2);
```

删除索引

```
DROP INDEX 索引名ON 表名;
```

索引的缺点

目前，你知道为表建立索引，可以加快检索速度，但索引也有很多缺点，下面介绍：

1. 索引加快了检索速度，却降低了有索引列的插入、删除、更新值的速度。原因很简单：更改数据的同时，也要更改索引数据，索引数据和数据是并存的。
2. 索引要占用磁盘空间，索引越多，占用数据空间越大。InnoDB 的索引文件和数据文件在同一个存储空间；MyISAM 的索引文件和数据文件可以分开存储。

索引的使用原则

1. 尽量为用来搜索、分类、分组的数据列建立索引。意味：最适合

有索引的列是那些在 WHERE 子句中出现的数据列、联结条件给出的数据列、ORDER BY 或 GROUP BY 子句中出现的数据列。

2. 考虑数据列的维度。维度（cardinality）：它所容纳的非重复的个数。比如：有一个数据列的值是：1、3、7、4、3；它的维度就是 4。维度越高（越接近行的个数），它包含独一无二的值越多，重复值越少，索引效果越好。如果你的行里面只有两种值，比如：男或女，加上索引也无济于事。
3. 对数据列类型的小值进行索引。数据列类型通常容量有大有小，比如：整型就有 tinyint 和 bigint；如果使用索引在小值类型上的效果远远大于大值类型。使用小值类型还可以节约存储空间。
4. 为字符串的前缀加索引而不是全部。假设，你有一个 CHAR(200) 的字符串列，其中数据有一定规律，就是前 10-20 个字符都一样，后面的字符不一样。这时候，你给这列加索引，请指定为前 20-30 个字符加索引，而不是 200 个长度加索引。较小的值编入索引占用空间小，速度更快。

   PS：ALTER TABLE employee ADD INDEX index_name (name(5));
5. 多列索引尽可能使用“最左边”的列名检索。假设，你有一个组合索引加在 3 列上 index_name(state, city, zip)，索引的存储就是按照以上顺序从左到右存储的，其实它的索引排序就是：

   state, city, zip

   state, city

state

如果你要检索或者 ORDER BY 排序，应该按照上述 3 种情况使用，才能用到索引功能，而你使用 city，zip 或 city 或 zip 其实没有发挥到索引功能。MySQL 不能使用没有包含最左边前缀的搜索的索引功能。

6. 索引适可而止，不能多建。索引越多效果越好是一种错误，不但增加存储空间，另外对数据更新（增、删、改）带来影响，只应该对最常用的列的建立索引。
7. 索引类型是与引擎类型匹配的。在建立索引的时候，有一个选项：索引方式：BTREE 或 HASH。这属于 MySQL 性能优化的高阶功能，使用默认即可，不要随意改变。InnoDB 使用“B 树”索引；MyISAM 也使用“B 树”索引，但遇到空间数据类型又会是“R 树”索引。并且，这两种索引方式，要搭配数据列检索时候用到的比较符才会体现性能不同，比如：== <= >= != BETWEEN AND 等等在使用的时候，情况比较复杂，需要根据实际查询微调，这是 DB 性能优化要做的事情。

再谈 SQL 语言（DML DDL DCL）

SQL（Structured Query Language）：结构化查询语言，由 IBM 公司 1981 年推出。SQL 目前已经被确定为关系数据库（Relational Database）系统的国际标准。

SQL 根据功能应用可以分为以下四个类别

1、 ***DML（Data Manipulation Language）***: 数据操作语言。是指，像 SELECT、INSERT、UPDATE、DELETE 命令，用来对数据库的数据进行操作的语言。

2、 ***DDL（Data Definition Language）***: 数据定义语言。是指，像 CREATE、ALTER、DROP 等，DDL 主要是用在定义或改变表（TABLE）的结构，数据类型，表之间的链接和约束等初始化工作上，他们大多在建立表时使用。

3、 ***TCL（Transaction Control Language）***: 事务控制语言。是指：像 Transaction、RollBack、Commit、SavePoint 等这些用于数据库事务控制的命令。

4、 ***DCL（Data Control Language）***: 数据控制语言。是指：用来设置或更改数据库用户或角色权限的语句，包括（Grant，Revoke 等）命令。

注意：在有些地方分类将 TCL 和 DCL 合在一起，统一用 DCL 表示。大部分资料都只有 DML、DDL、DCL 分类。

事务处理

概念

事务（transaction）：作为一个不可分割的逻辑单元而执行的一组 SQL 语句，如有必要，它们的执行可以撤销。简单来说，一件事情分好几步来做，要么完全执行成功，要么完全不执行不成功。事务用来维护数据的完整性，针对批量的操作进行保证其完整性。

〔案例 1〕

张三给李四开了一张 100 元的支票，李四拿着这张支票去取钱，针对这件事情，李四的账户应该增加 100 元，而张三的账户应该减少 100 元。这两个步骤是一体不可分割的。

```
Update account Set balance = balance – 100 Where name = ‘张三’;
Update account Set balance = balance + 100 Where name = ‘李四’;
```

假设，在进行这个操作的时候，突然发生意外，导致整个操作不完整，造成第一条语句成功，第二条语句失败或第一条语句失败，第二条语句成功，这都将出现错误的数据。

要解决上述的困境，就需要有事务机制，将两条语句操作看成一个整体，这个整体要么全部成功，要么全部失败。事务中的回滚机制对数据中的局部操作不完整进行整体撤销；事务中的提交机制，保证

每一步都成功，才整体更新数据。事务的另外一个用途，确保某个操作所涉及的数据行在使用的时候不被其它客户修改。事务机制通过把多条语句定义为一个执行单元，防止在多用户的环境中可能发生的资源冲突的并发问题。

事务的原则

事务机制通常被概括为：ACID 原则。ACID 是 Atomic（原子性）、Consistent（稳定性）、Isolated（孤立性）、Durable（可靠性）的首字母缩写。它们分别代表事务机制应该具备的一个属性。

1、 原子性：构成一个事务的所有语句应该是一个独立的逻辑单元，要么全部执行成功，要么一个都不成功。你不能只执行它们当中的一部分。

2、 稳定性：数据库在事务开始之前或事务执行之后都必须是稳定的。简单讲，事务不应该影响你的数据库。

3、 隔离性：事务是单独执行的单元，不应该相互影响。

4、 可靠性：如果事务执行成功，数据将永久性的存储到数据库里。

事务相关术语

事务（Transaction）：一组 SQL 语句。

回滚〔退〕（Rollback）：撤销指定的 SQL 语句的过程。

提交（Commit）：将未存储的 SQL 语句结果写入到数据库中。

保存点（Savepoint）：事务处理中设置的临时占位点，回退功能可以选择回退到占位点，而不一定是开始点。

MySQL 的事务执行

并非所有的引擎都支持事务。在 MySQL 中 MyISAM 和 InnoDB 是两种最常用的引擎，前者不支持事务，后者才支持。可以使用：SHOW ENGINES;语句查看数据库引擎的有关信息。

在默认情况下，MySQL 的事务是自动提交（autocommit）模式运行，这意味着每条语句在执行完毕后，自动将执行结果永久性的保留到数据库中，相当于每一条语句就是一个隐含事务。如果你想明确事务的执行，需要禁用自动提交模式并告诉 MySQL 你想何时提交何时回滚。

使用事务 1

执行事务控制的第一个方法是：使用 Start Transaction 语句挂起自动提交模式，然后执行本次事务的多条语句，最后一条 Commit 语句结束事务并把所有语句的最后结果永久性的写入数据库。如果，中途语句碰见错误或不想语句执行成功，使用一条 Rollback 语句撤销事务并把数据恢复到事务开始之前的状态。Start Transaction 语句“挂起”自动提交模式的含义是：在事务提交或回滚后，该模式将恢复到本次开始事务 Start Transaction 语句执行前的模式－开

始前是自动提交模式，将还是自动提交模式；开始前是手动模式，将还是手动模式。数据库是自动模式还是手动模式可以通过设置来完成，后面会讲。

〔示例〕

```
SELECT * FROM employee;

START  TRANSACTION;

DELETE FROM employee WHERE id > 3;

SELECT * FROM employee;

ROLLBACK;

SELECT * FROM employee;
```

使用事务 2

执行事务的第二种方式是：利用 SET 语句直接把自动提交模式改为手动模式。

```
Set autocommit = 0;   -- 手动

Set autocommit = 1;   -- 自动
```

autocommit 变量设置为 0 将禁用自动提交模式，意味着随后的任何语句都将成为当前事务的一部分，直到发出一条 Commit 或 RollBack 语句后为止，本次事务结束，接着有开启下次事务，依次类推。

注意：Set autocommit = 0;只针对 MySQL 数据库一次连接有

效，并不是对 DB 服务器永久有效，你断开本次连接，重新连接后，将恢复默认设置。

〔示例〕

```
SELECT * FROM employee;

Set autocommit = 0;

DELETE FROM employee WHERE id > 3;

SELECT * FROM employee;

ROLLBACK;

SELECT * FROM employee;
```

影响事务的语句

在 SQL 中有些语句不能成为事务的一部分，它们将会对事务产生隐式影响。一般来说：用来创建、改变或删除数据库或其中的 DDL 语句以及锁定有关的语句都不能成为事务的一部分。如果你在事务中有以下语句，那么 MySQL 将在执行这些语句之前提交当前事务。

```
ALTER TABLE

CREATE INDEX

DROP DATABASE

DROP INDEX

DROP TABLE

LOCK TABLES
```

```
RENAME TABLE

SET AUTOCOMMIT = 1

TRUNCATE TABLE

UNLOCK TABLE

……
```

对应不同的 MySQL 的版本，可能上述语句不完整，请查阅《MySQL 参考资料》。

使用事务保存点

事务保存点的意思就是说 SQL 可以对一个事务进行部分回滚，你在事务过程中不同的地方设置这个保存点标记，回滚可以选择回滚到某个保存点，实现部分回滚。

〔示例〕

```
SET autocommit = 0; # 只针对一次连接的自动提交事务取消

SELECT * FROM employee;

DELETE FROM employee WHERE id = 1;

SAVEPOINT del01;

SELECT * FROM employee;

DELETE FROM employee WHERE id = 2;

SAVEPOINT del002;
```

```
SELECT * FROM employee;
ROLLBACK TO del01;
SELECT * FROM employee;
COMMIT;
```

MySQL 的事务隔离级别

MySQL 是一个多用户使用的数据库系统，所以，不同的用户可能会在同一时间试图访问一个数据表。InnoDB 引擎（其它引擎暂时讨论）的策略是：使用了数据行级别的锁定机制为用户访问数据表提供了控制。在某个用户修改某个数据行的同时，另外一个用户可以读取和修改同一个表的其它数据行；如果有两个用户想同时修改某个数据行，先锁定的用户可以修改它。

多个事务同时运行，可能会产生以下问题：

1、 脏读（dirty read）：指某个事务所作出的修改在它尚未被提交的时候就可以被其它事务看到。其它事务会认为数据行已经被修改，但对数据行作出修改的事务还可以回滚，因此，导致数据混乱。

2、 不可重复读（nonrepeatable read）：指同一个事务使用同一条 Select 语句每次读到的结果不一样。比如：有一个事务有两次相同的 Select 语句，但另外一个事务在两次之间执行了修改数据的操作，就会发生这种问题。

3、 幻影数据行〔幻象读〕（phantom row）：指某个事务突然看到了一个以前它从来没有看见过的数据。比如：一个事务执行了 Select 语句后就有另外一个事务插入了一条新数据，前一个事务再次执行同一条 Select 语句就看到了一条新的数据行，这就是一个幻影数据行。

为了解决上述多个事务同时执行产生的问题，MySQL 的事务隔离级别的设置可以解决它们。下面是 MySQL 的 InnoDB 所支持的 4 个隔离级别：

1、 READ UNCOMMITTED：允许某个事务看到其它事务尚未提交的数据改动。

2、 READ COMMITTED：只允许某个事务看到其它事务已经提交的数据改动。

3、 REPEATABLE READ：如果某个事务执行两次相同的 Select，其结果是相同的，尽管在两次 Select 之间已经有其它事务插入或修改了数据行，这个事务看到的结果还是前后相同。

4、 SERIALIZABLE：某个事务正在 Select 查看某个数据行，其它事务都不允许修改，直到前个事务完成为止。

MySQL 隔离级别所允许的问题

隔离级别	脏读	不可重复读	幻影数据行
READ	Y	Y	Y

UNCOMMITTED			
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

InnoDB 引擎默认的隔离级别是 REPEATABLE READ。更改隔离级别的方式有以下几种：

1、 my.ini 中的 [mysqld] 下： transaction-isolation= REPEATABLE READ。-- 数据库服务器的配置文件，作用整个数据库。

2、 SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 全局隔离级别，设置作用于连接此数据库服务器的任何用户。

3、 SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 会话隔离级别，设置作用于本次用户会话的后续所有事务。

4、 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 只作用于下一个事务。

小结：不要认为隔离级别越高就一定越好，隔离级别高可以避免

很多事务的并发问题，但带来的是性能的损失；隔离级别低可以充分发挥并发处理。并发和事务永远是矛盾体，此消彼长，在没有特殊的情况下，一般不要更改数据库默认的隔离级别，既使使用 Java 以及 Spring 框架等，也应该遵循将隔离规则交由数据库本身处理，不同的数据库对隔离级别支持力度也有所不同。

MySQL 安全管理和访问控制

MySQL 是一个多用户 DBMS，对于不同的用户应该有不同的权限。MySQL 的安全基础概括为：用户应该对他们需要的数据具有适当的访问权限，既不多也不少，合适就好。考虑到这个原则，你在给用户授予权限的时候应该参考以下内容：

1、 多数用户只需要对表进行读和写，少数用户需要创建和删除表。

2、 某些用户只需要读表，但不能更新表。

3、 你可以允许某些用户添加数据，但不能允许他们删除数据。

4、 只有 DBA 需要处理用户权限，多数用户不关心他人的权限受理。

5、 你可能允许用户通过存储过程或视图访问数据，但不允许他们直接访问数据。

6、 你可能想根据用户的登录地点来限制某些访问功能。

管理用户

MySQL 在安装时候创建的 root 账号，是最大权限的用户名，他对整个 MySQL DB 具有完全的控制权。在实际过程中，应该建立其它用户来操作数据库，而不是直接使用 root。

在 MySQL DB 中专门有一个名字叫做“mysql”的数据库，里面存储了关于本 DB 的所有信息，比如：你创建的存储过程、触发器、事件等等，其中的 user 表记录了所有该 DB 的登录用户信息及权限。

创建用户账号

创建一个用户账号，使用关键字：CREATE USER 语句。

PS：

```
CREATE USER 'jack' IDENTIFIED BY '123';
```

重命名用户账号

关键字：RENAME

PS：RENAME USER 'jack' TO 'tom';

更改口令

PS：SET PASSWORD FOR 'jack' = PASSWORD('jack');

删除用户账号

关键字：DROP

PS: DROP USER 'tom';

设置访问权限

光有用户账号，只能登录到 MySQL，但他没有任何权限访问其中任何数据库、数据，所以，必须立即给他授权。

查看用户账号的权限：

SHOW GRANTS FOR 'jack';

设置授权的关键字是 GRANT 语句。GRANT 要求你至少给出以下信息：

1 要授予的权限

2 被授予访问权限的数据库或表

3 用户名

PS：GRANT SELECT ON edu.* TO 'jack';

解释：为 jack 用户账号授权访问 edu 数据库的所有数据具有只读权限。

授权 Grant 的反操作是撤销权限，关键字是 REVOKE。

PS：REVOKE SELECT ON edu.* FROM 'jack';

注意：撤销的权限必须存在，否则报错；撤销权限并不会删除用户，

撤销权限和删除用户是两个概念。

GRANT 和 REVOKE 详解

GRANT 和 REVOKE 在几个层次上控制访问权限，如下：

- 整个服务器上：GRANT ALL，REVOKE ALL
- 整个数据库上：ON database.*
- 特定的表：ON database.table
- 特定的列
- 特定的存储过程

权限访问有一个权限说明表，请查看《MySQL》手册。

权限说明表

权限	说明
ALL	除 GRANT OPTION 以外的所有权限
ALTER	使用 ALTER TABLE
ALTER ROUTINE	使用 ALTER PROCEDURE 和 DROP PROCEDURE
CREATE	使用 CREATE TABLE
……	……

全文本搜索

在前面我们介绍了 LIKE 和正则表达式作为搜索方式来匹配关键字找到我们需要的资料，虽然这两种机制非常有用，但也存在着以下限值：

1、 性能：通配符和正则表达式匹配通常要求 MySQL 尝试匹配表中的所有行，因此，当搜索行数据不断增加后，这些搜索是非常耗时的。

2、 明确控制：使用前两种方式，很难精确控制匹配什么，不匹配什么。比如：指定一个词必须匹配，一个词必须不匹配；或一个词仅在第一个词确实匹配情况下才匹配或不匹配。

3、 智能化结果：使用前两种方式，它们不提供一种智能化的选择结果的方法。比如：一个特殊词的搜索将会返回所有包含该词的行，而不会区分包含单个匹配的行和包含多个匹配的行等等。

所有的上述限值以及更灵活的限值都可以通过全文本搜索来解决。使用全文本搜索的时候，MySQL 不需要每行查看，也不需要分别分析和处理每个单词，MySQL 是创建指定列中各词的一个索引，搜索功能是针对这些词进行的，所以，可以快速有效的解决词的匹配，不匹配，匹配频率等等。

要使用全文本搜索功能，需要具备以下条件：

1、 MySQL 数据库支持几种数据引擎，并非所有的数据引擎类型都支持全文本搜索。MySQL 中最常用的 MyISAM 和 InnoDB 引擎，前者支持全文本搜索，后者不支持。如果你想要支持全文本搜索功能，请注意你的表的搜索引擎。MySQL 5.5 默认的引擎为 InnoDB。

2、 全文本搜索的基础是 FULLTEXT 索引，这种索引只能对 CHAR、VARCHAR、TEXT 类型是数据列建立。

3、

```
# MyISAM 引擎的表无法加外键约束
DROP TABLE IF EXISTS productnotes;
CREATE TABLE productnotes (
   note_id BIGINT NOT NULL AUTO_INCREMENT,
   prod_id BIGINT NOT NULL,
   note_date DATETIME NOT NULL,
   note_text TEXT,
   PRIMARY KEY (note_id),
   FULLTEXT (note_text)
)ENGINE = MyISAM;
```

FULLTEXT 为支持全文本搜索的一种特殊索引。

全文本搜索的表达式

全文本搜索需要两个函数：Match 和 Against。Match()指定被搜索的列，Agains()指定要使用的搜索表达式。