第七章：MySQL高级

第1节：视图

1.1 概念

1. 视图是简化查询过程，提高数据库安全性的虚拟表。
2. 视图中保存的仅仅是一条select语句，保存的是视图的定义，并没有保存真正的数据。视图中的源数据都来自于数据库表，数据库表称为基本表或者基表，视图称为虚拟表。

1.2 作用

1. 防止未经许可的用户访问敏感数据，确保数据的安全性
2. 封装sql语句，简化查询过程
3. 视图可对用户屏蔽真实表结构

1.3 语法

格式：

① 创建视图：CREATE VIEW 视图名字 AS SELECT 语句;
② 查询视图：SELECT * FROM 视图名;
③ 查看视图结构：desc 视图名;
④ 查看创建视图的文本信息：SHOW CREATE VIEW 视图名;
⑤ 修改视图：ALTER VIEW 视图名称 AS SQL语句 ;
⑥ 删除视图：DROP VIEW 视图名称;

注意：

查看当前用户是否有创建视图的权限

```
SELECT Select_priv,Create_view_priv FROM MySQL.user WHERE user='用户名';
```

参数说明：

(1)Select_priv：属性表示用户是否具有SELECT权限，Y表示拥有SELECT权限，N表示没有。
(2)Create_view_priv：属性表示用户是否具有CREATE VIEW权限，Y表示拥有CREATE 权限，N表示没有;
(3)MySQL.user：表示MySQL数据库下面的user表。
(4)用户名：参数表示要查询是否拥有权限的用户，该参数需要用单引号引起来。

第2节：存储过程

2.1 概念

1. 存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。类似于java中的方法。
2. 存储过程是为了完成特定功能的SQL语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数(需要时)来调用执行。
3. 存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用。

2.2 优点

1. 存储过程在创建的时候直接编译，而sql语句每次使用都要编译，提高执行效率
2. 一个存储过程可以被重复使用。（其实sql语句也可以，没什么卵用）
3. 一条sql语句，可能需要访问几张表，对数据库连接好几次，存储过程只会连接一次
4. 存储的程序是安全的。数据库管理员可以向访问数据库中存储过程的应用程序授予适当的权限，而不向基础数据库表提供任何权限。

2.3 缺点

1. 可移植性太差了
2. 对于简单的sql语句，毫无意义
3. 对于只有一类用户的系统安全性毫无意义
4. 团队开发，标准不定好的话，后期维护很麻烦
5. 对于开发和调试都很不方便
6. 复杂的业务逻辑，用存储过程还是很吃力的

2.4 使用

1. 创建存储过程

```
DELIMITER $$
CREATE PROCEDURE 存储过程名(参数和返回值)
BEGIN
SELECT * FROM users;
END$$
DELIMITER $$
```

说明：

IN 表示这个存储过程需要的输入参数

out表示这个存储过程需要的输出参数

inout是可以接受一个参数并输出一个参数

2. 调用存储过程

```
CALL 存储过程名(参数);
```

3. 删除存储过程

```
DROP PROCEDURE 存储过程名;
```

2.5 案例

```
# 1. 选中库
USE myemployees;

# 2. 创建存储过程
DELIMITER $$ # 声明结束标识
CREATE PROCEDURE emp_pro1()
  BEGIN
    SELECT * FROM employees;
  END $$
DELIMITER $$
```

```
# 3. 调用存储过程
CALL emp_pro1();

# 4. 创建传参的存储过程  in
DELIMITER $$
CREATE PROCEDURE emp_pro2(IN eid INT)
  BEGIN
    SELECT * FROM employees WHERE employee_id = eid;
  END $$
DELIMITER $$

CALL emp_pro2(103);

DELIMITER $$
  CREATE PROCEDURE emp_pro3(IN eid INT,IN jobid VARCHAR(10))
  BEGIN
    SELECT * FROM employees WHERE employee_id=eid AND job_id=jobid;
  END $$
DELIMITER $$

CALL emp_pro3(101,'AD_VP');

# 5. 创建传参的存储过程  out
# 案例1：封装存储过程：计算1+2+...+n结果 出参 out
DELIMITER $$
  CREATE PROCEDURE sum_pro4(IN n INT,OUT sums INT)
  BEGIN
    DECLARE i INT DEFAULT 1;# 声明一个变量初始值为1
    DECLARE temp INT DEFAULT 0;
    WHILE i<=n DO
      SET temp = temp + i;
      SET i = i+1;
    END WHILE;
    SET sums = temp;
  END $$
DELIMITER $$

CALL sum_pro4(100,@sum);
SELECT @sum;

# 6. 创建传参的存储过程  inout即可以入参也可出参
DELIMITER $$
  CREATE PROCEDURE emp_pro5(INOUT xx VARCHAR(32))
  BEGIN
    SELECT last_name INTO xx FROM employees WHERE employee_id=xx;

  END $$
DELIMITER $$
```

```
SET @id = 101;
CALL emp_pro5(@id);
SELECT @id;

# 7. 删除存储过程
DROP PROCEDURE emp_pro1;
```

第3节：触发器

3.1 概念

1. 触发器的这种特性可以协助应用在数据库端确保数据的完整性。也可以把触发器理解成一个特殊的存储过程，不需要显示调用，是自动被调用的存储过程。
2. 类似于servlet中的监听器
3. 监视某种情况，并触发某种操作，它是提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，例如当对一个表进行操作（insert，delete， update）时就会激活它执行。

3.2 语法

语法：

```
DELIMITER $$
CREATE TRIGGER 触发器名 触发时机（BEFORE|AFTER） 触发事件（INSERT|UPDATE|DELETE）
ON 表名 FOR EACH ROW
BEGIN
  执行语句列表;
END $$

DELIMITER
```

说明：

- BEFORE和AFTER参数指定了触发执行的时间，在事件之前或是之后。
- FOR EACH ROW表示任何一条记录上的操作满足触发事件都会触发该触发器，也就是说触发器的触发频率是针对每一行数据触发一次。
- 触发事件参数详解：
  1. INSERT型触发器：插入某一行时激活触发器，可能通过INSERT、REPLACE 语句触发；
  2. UPDATE型触发器：更改某一行时激活触发器，可能通过UPDATE语句触发；
  3. DELETE型触发器：删除某一行时激活触发器，可能通过DELETE、REPLACE语句触发。

介绍：

```
BEFORE INSERT：在添加之前激活触发器
BEFORE DELETE：在删除之前激活触发器
BEFORE UPDATE：在修改之前激活触发器
AFTER INSERT：在添加之后激活触发器
AFTER DELETE：在删除之后激活触发器
AFTER UPDATE：在修改之后激活触发器
```

3.3 使用

1. 表准备(创建一个times表)

```
CREATE TABLE `times` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `createTime` datetime DEFAULT NULL,
  `state` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`)
)
```

字段说明:

createTime:为添加数据的当前时间

state:是状态信息

2. 创建触发器

```
DELIMITER $$
  CREATE TRIGGER trigger2 AFTER INSERT ON jobs FOR EACH ROW
  BEGIN
    INSERT INTO emp_log(createtime,info) VALUES(NOW(),CONCAT('jobs表中插入一条数据:',new.job_title));
  END $$
DELIMITER $$


DELIMITER $$
  CREATE TRIGGER trigger3 AFTER UPDATE ON jobs FOR EACH ROW
  BEGIN
    INSERT INTO emp_log(createtime,info) VALUES(NOW(),CONCAT('jobs表中插入一条数据:',old.job_title));
  END $$
DELIMITER $$
```

说明:

MySQL 中定义了 NEW 和 OLD,用来表示触发器的所在表中,触发了触发器的那一行数据,来引用触发器中发生变化的记录内容,具体地:

① 在INSERT型触发器中,NEW用来表示将要(BEFORE)或已经(AFTER)插入的新数据;

② 在UPDATE型触发器中,OLD用来表示将要或已经被修改的原数据,NEW用来表示将要或已经修改为的新数据;

③ 在DELETE型触发器中,OLD用来表示将要或已经被删除的原数据;

3. 触发

```
INSERT INTO jobs VALUES('IT_javaEE','javaEE开发工程师',15000,100000);
UPDATE jobs SET job_title='JavaEE高级工程师' WHERE job_id = 'IT_javaEE';
```

4. 查看表

5. 查看触发器

```
SHOW TRIGGERS;
```

6. 删除触发器

```
DROP TRIGGER 触发器名
```

第4节：存储引擎

4.1 概念

思考：我们在mysql中创建的数据库、表以及表中的数据是保存在哪里的？（都是以文件的形式存储在硬盘上的）

1. 在关系型数据库中每一个数据表就对应一个文件，而这些文件是以什么方式存储在硬盘上的就取决于选择的存储引擎类型，不同的存储引擎存储文件的方式也是不一样的。
2. 我们可以认为数据库存储引擎是规定数据表如何存储数据，如何为存储的数据建立索引以及如何支持更新、查询等技术的实现。
3. 不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能
4. 由于在关系数据库中数据的存储是以表的形式存储的，所以存储引擎也可以称为表类型，也就是说存储引擎是针对表而言的
5. 查看MySQL支持的存储引擎类型：`SHOW ENGINES`
6. 查看当前默认的存储引擎类型：`SHOW VARIABLES LIKE '%storage_engine%'`

4.2 详解

4.2.1 MyISAM

特点：

1. 在做插入、查询时速度快，性能高
2. 支持全文索引，表级锁
3. 不支持事务，外键
4. 其数据的物理组织形式是非聚簇表。数据和索引分开存储，顶级节点只存索引，数据都存储在B+树的叶子节点

文件存储格式：

1. tb01.frm：存储tb01表结构信息（表中有哪些列，数据类型等）
2. tb01.MYI：MY表示MYISAM存储引擎，I是index索引，这里存储tb01的索引信息。
3. tb01.MYD：MY表示MYISAM存储引擎，D是data数据，这里存储tb01的数据信息（即表中的记录）

4.2.2 InnoDB

特点：

1. 支持外键、支持事务
2. 支持行级锁，因此在高并发量的情况下效率高
3. 不支持全文索引
4. 其数据的物理组织形式是聚簇表。数据和索引放在一块，都位于B+树的叶子节点上。

文件存储格式：

1. xx.frm:同MYISAM存储引擎，也是用来存储表结构信息。
2. ibdata1:共享表空间，且来存储所有InnoDB数据表的数据信息，包含索引信息

4.2.3 指定存储引擎

```
1. 建表时:
    create table 表名(id bigint(12),name varchar(200)) ENGINE=MyISAM;
    create table 表名(id int(4),cname varchar(50)) ENGINE=InnoDB;

2. 已建表修改
    alter table 表名 engine = innodb;
```

第5节：索引

5.1 概念

```
1. 索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。更通俗的说，数据库索引好比是一本书前面的目录，能加快数据库的查询速度
2. 索引分为聚簇索引和非聚簇索引两种
```

5.2 分类

```
1. 普通索引 ：仅加速查询
2. 唯一索引 ：加速查询 + 列值唯一（可以有null）
3. 组合索引 ：多列值组成一个索引，
4. 主键索引：加速查询 + 列值唯一 +  表中只有一个（不可以有null）
```

5.3 创建索引

```
1. 创建表的时候创建索引
  语法:
  CREATE TABLE tbl_name(
    字段名称 字段类型 [完整性约束条件],
    ....
    INDEX [索引名称](字段名称)
  );

2. 在已经存在的表上创建索引:
语法:
  1.CREATE INDEX 索引名称 ON 表名（字段名）
  2.ALTER TABLE 表名 ADD INDEX 索引名称(字段名称);

3. 查看索引
  show index from 表名;

4.  删除索引
  drop index 索引名 on 表名;
```

5.4 索引使用

1. 建议使用
   ① 主键自动建立唯一索引，任何表一定要建主键
   ② 频繁作为查询条件的字段应该创建索引
   ③ 查询中与其它表关联的字段，外键关系建立索引
   ④ 组合索引的选择问题， 组合索引性价比更高
   ⑤ 查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度
   ⑥ 查询中统计或者分组字段

2. 不建议使用
   ① 表记录太少
   ② 经常增删改的表或者字段，因为对表进行INSERT、UPDATE和DELETE。
   因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。
   ③ Where条件里用不到的字段不创建索引

3. 注意事项
   ① 如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。
   ② 不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描
   ③ 存储引擎不能使用索引中范围条件右边的列
   ④ mysql 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描
   ⑤ is not null 也无法使用索引,但是is null是可以使用索引的(和非空约束有关系)
   ⑥ like以通配符开头('%abc...')mysql索引失效会变成全表扫描的操作
   ⑦ 字符串不加单引号索引失效

第6节：锁机制

6.1 概念

1. 锁是计算机协调多个进程或线程并发访问某一资源的机制。
2. 在数据库中，除传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

6.2 锁分类

1. 从对数据操作的类型（读\写）分
   ① 读锁(共享锁)：针对同一份数据，多个读操作可以同时进行而不会互相影响。
   ② 写锁（排它锁）：当前写操作没有完成前，它会阻断其他写锁和读锁。

2. 从对数据操作的粒度分
   ① 表锁
   特点：偏向MyISAM存储引擎，开销小，加锁快；无死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。
   ② 行锁
   特点：
   1). 偏向InnoDB存储引擎，开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

2).InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁

3. 锁的用法划分

① 悲观锁

我们使用悲观锁的话其实很简单(手动加行锁就行了)：select * from xxxx for update，在select 语句后边加了for update相当于加了排它锁(写锁)，加了写锁以后，其他事务就不能对它修改了！需要等待当前事务修改完之后才可以修改.也就是说，如果操作1使用select ... for update，操作2就无法对该条记录修改了，即可避免更新丢失。

② 乐观锁

乐观锁不是数据库层面上的锁，需要用户手动去加的锁。一般我们在数据库表中添加一个版本字段version来实现，例如操作1和操作2在更新User表的时，执行语句如下：

update A set Name=lisi,version=version+1 where ID=#{id} and version=#{version},

此时即可避免更新丢失。

③ 间隙锁GAP

当我们用范围条件检索数据而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合范围条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在 的记录，叫做“间隙(GAP)”。

InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁。

例子：假如emp表中只有101条记录，其empid的值分别是1,2,...,100,101

```
Select * from emp where empid > 100 for update;
```

上面是一个范围查询，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101(这些记录并不存在)的“间隙”加锁

InnoDB使用间隙锁的目的有2个：

为了防止幻读(上面也说了，Repeatable read隔离级别下再通过GAP锁即可避免了幻读)

满足恢复和复制的需要：MySQL的恢复机制要求在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读

④ 死锁

1、产生原因

所谓死锁：是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用，它们都将无法推进下去.此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的InnoDB。

死锁的关键在于：两个(或以上)的Session加锁的顺序不一致。

那么对应的解决死锁问题的关键就是：让不同的session加锁有次序

2、产生示例

需求：将投资的钱拆成几份随机分配给借款人。

起初业务程序思路是这样的：

投资人投资后，将金额随机分为几份，然后随机从借款人表里面选几个，然后通过一条条select for update 去更新借款人表里面的余额等。

例如：两个用户同时投资，A用户金额随机分为2份，分给借款人1，2

B用户金额随机分为2份，分给借款人2，1，由于加锁的顺序不一样，死锁当然很快就出现了。

对于这个问题的改进很简单，直接把所有分配到的借款人直接一次锁住就行了。

```
Select * from xxx where id in (xx,xx,xx) for update
```

在in里面的列表值mysql是会自动从小到大排序，加锁也是一条条从小到大加的锁