# 视图-View

## 简介

**视图是一个从一张或几张数据表或视图中导出的虚拟表，它的作用类似于对数据表进行筛选，必须使用SQL语句中的SELECT语句实现构成**。在定义视图时，只是把视图的定义存放在数据库，并不保存视图的数据，直到用户使用视图时才进行数据的查询并返回操作，当需要从不同的服务器中获得数据时，使用视图可以很好的将结构相同的数据组织并返回。

## 视图的特点

1. 简化数据的操作

可以将**经常使用**的连接，联合查询、及选择查询定义为视图，只需要简单的调用视图就行。视图可以隐藏表表间的复杂关系。

1. 可作为安全机制-限制用户访问

用户可以通过设置视图，**使特定的用户只能查看或修改他们权限内的数据**，其它的数据库或表不能进行操作。视图的安全性可以防止未授权的用户查看或操作特定的列或行，通过在你表中设置一个用户的标志来建立视图，使用户只能查看或操作标志自己标志的列或行，从而保证数据的安全性。

1. 合并及分割数据

应用场景：表中的数据量太大，需要对表进行拆分，这样会导致表的结构发生变化。

可以使用视图来屏蔽实体表间的逻辑关系，去构建应用程序所需要的原始表关系。

1. 数据的导入导出

利用视图导入导出特定的数据表，就是一个临时表的作用。

## 视图的类型

1. MERGE

将视图的sql语句和引用视图的sql语句合并在一起，最后一起执行。

1. TEMPTABLE

将视图的结果集存放在临时表中，每次执行时从临时表中操作。

1. UNDEFINED

默认的视图类型，DBMS倾向于选择而不是必定选择MERGE，因为MERGE的效率更高，更重要的是**临时表视图不能更新**。

## 视图的使用

1. 创建视图

具体格式：

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

VIEW 视图名称 [(column\_list)] AS SELECT\_statement（查询语句）

[WITH [CASCAD|LOCAL] CHECKOPTION]

简略格式：

CREATE VIEW 视图名(列1，列2...) AS SELECT (列1，列2...) FROM ...;

1. 使用视图-和表的使用方式差不多

SELECT (列1，列2，列3) FROM 视图名称 WHERE 条件;

1. 修改视图 CREATE OR REPLACE VIEW 视图名 AS SELECT [...] FROM [...];

具体格式：

ALTER ALGORITHM =MERGE VIEW 视图名称

AS （查询语句）SELECTpt.ptno,pt.ptname,pi.pino,pi.piname,pi.piprice

FROM t\_product\_typept INNER JOIN t\_product\_item pi ON pt.id = pi.pifid;

1. 删除视图

DROP VIEW IF EXISTS 视图名称;

1. 查看数据库已有视图-查看表一样

SHOW TABLES [like...];（可以使用模糊查找）

1. 查看视图详情-查看表详情

DESC 视图名

或者

SHOW FIELDS FROM 视图名

1. 视图条件限制 [WITH CHECK OPTION]

如果在创建视图的时候制定了“WITH CHECK OPTION”，那么更新数据时不能插入或更新不符合视图限制条件的记录。

限制的条件是SELECT语句中的WHERE决定

## 视图的更新-对于可更新视图才能进行CUD操作

可更新视图的条件：

1. 对于可更新的视图在视图中的行和基表中的行之间必须具有一对一的关系。
2. 还有一些特定的其他结构，这类结构会使得视图不可更新。

聚合函数、DISTINCT关键字、GROUP BY子句、ORDER BY子句、HAVING子句、UNION运算符、位于选择列表中的子查询、FROM子句中包含多个表、SELECT语句中引用了不可更新视图、WHERE子句中的子查询，引用FROM子句中的表、ALGORITHM 选项指定为TEMPTABLE（使用临时表总会使视图成为不可更新的）

# 存储过程 - MySQL5.0才开始支持存储过程

## 定义 - 编译以后的SQL语句

1. **SQL语句需要先编译然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中**，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。
2. **存储过程是可编程的函数**，在数据库中创建并保存，可以**由SQL语句和控制结构组成**。当想要在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。**数据库中的存储过程可以看做是对编程中面向对象方法的模拟，它允许控制数据的访问方式**。

## 存储过程的特点

1. 增强SQL语言的功能和灵活性

储过程可以用控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

1. 标准组件式编程 - 耦合度降低

存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

1. 性能方面 - 较快的执行速度

因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

1. 减少网络流量

针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织进存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大减少网络流量并降低了网络负载。

1. 安全机制

通过对执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

## 存储过程的使用

* 1. 关键字DELIMITER -- 作用是指定新的MySQL分隔符

DELIMITER // -- 将分隔符；替换为//

DELIMITER ； -- 将分隔符// 替换为；

* 1. 存储过程的创建

1. 简略格式：

CREATE PROCEDURE 存储过程名称 ([ [IN|OUT|INOUT] 参数名 数据类型

[,[IN|OUT|INOUT] 参数名 数据类型…] ]) [特性 ...] (BEGIN)过程体(END)

1. 具体格式：

DELIMITER //

CREATE PROCEDURE selsing(IN sno INT)

BEGIN

SELECT id , city

FROM student

WHERE id = sno;

END//

DELIMITER;

1. 参数说明

存储过程根据需要可能会有输入、输出、输入输出参数，如果有多个参数用","分割开。MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT:

IN参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

OUT:该值可在存储过程内部被改变，并可返回

INOUT:调用时指定，并且可被改变和返回

3.3 调用存储过程

(1) 具体格式

#调用

SET @p\_inout=1; //@变量名称 用户变量

CALL inout\_param(@p\_inout) ;

SELECT @p\_inout;

CALL 存储过程名(参数) -- 输入参数/输出参数/输入输出参数

(2) CALL selsing(2) 这样调用就可以

## 查询MySQL中的存储过程

#查询存储过程

SELECT name FROM mysql.proc WHERE db='数据库名';

SELECT routine\_name FROM information\_schema.routines WHERE routine\_schema='数据库名';

SHOW PROCEDURE STATUS WHERE db='数据库名';

#查看存储过程详细信息

SHOW CREATE PROCEDURE 数据库.存储过程名;

## 存储过程的修改

1. 具体格式

ALTER {PROCEDURE | FUNCTION} sp\_name [characteristic ...]

characteristic:

{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }

| SQL SECURITY { DEFINER | INVOKER }

| COMMENT 'string'

1. 参数说明

sp\_name参数表示存储过程或函数的名称；

characteristic参数指定存储函数的特性。

CONTAINS SQL表示子程序包含SQL语句，但不包含读或写数据的语句；

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。

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；INVOKER表示调用者可以执行。

COMMENT 'string'是注释信息。

1. 实例代码

#将读写权限改为MODIFIES SQL DATA，并指明调用者可以执行。

ALTER PROCEDURE 存储过程名称

MODIFIES SQL DATA

SQL SECURITY INVOKER ;

#将读写权限改为READS SQL DATA，并加上注释信息'FIND NAME'。

ALTER PROCEDURE name\_from\_employee

READS SQL DATA

COMMENT 'FIND NAME' ;

## 存储过程的删除

DROP PROCEDURE 存储过程名称；

## 存储过程的控制语句

1. 变量作用域

内部变量在其作用域范围内享有更高的优先权，当执行到end时，内部变量消失，不再可见了，在存储过程外再也找不到这个内部变量，但是可以通过out参数或者将其值指派给会话变量来保存其值。

实例代码：

#变量作用域

DELIMITER //

CREATE PROCEDURE proc()

BEGIN

DECLARE x1 VARCHAR(5) DEFAULT 'outer';

BEGIN

DECLARE x1 VARCHAR(5) DEFAULT 'inner';

SELECT x1;

END;

SELECT x1;

END;

//

DELIMITER ;

#调用

CALL proc();

* 关键字DECLARE 用于在符合语句中定义一个变量

1. 条件语句 - IF-THEN-ELSE语句

#条件语句IF-THEN-ELSE

DROP PROCEDURE IF EXISTS proc3;

DELIMITER //

CREATE PROCEDURE proc3(IN parameter int)

BEGIN

DECLARE var int;

SET var=parameter+1;

IF var=0 THEN

INSERT INTO t VALUES (17);

END IF ;

IF parameter=0 THEN

UPDATE t SET s1=s1+1;

ELSE

UPDATE t SET s1=s1+2;

END IF ;

END ;

//

DELIMITER ;

1. CASE-WHEN-THEN-ELSE语句

#CASE-WHEN-THEN-ELSE语句

DELIMITER //

CREATE PROCEDURE proc4 (IN parameter INT)

BEGIN

DECLARE var INT;

SET var=parameter+1;

CASE var

WHEN 0 THEN

INSERT INTO t VALUES (17);

WHEN 1 THEN

INSERT INTO t VALUES (18);

ELSE

INSERT INTO t VALUES (19);

END CASE ;

END ;

//

DELIMITER ;

1. 循环语句 WHILE-DO…END-WHILE

DELIMITER //

CREATE PROCEDURE proc5()

BEGIN

DECLARE var INT;

SET var=0;

WHILE var<6 DO

INSERT INTO t VALUES (var);

SET var=var+1;

END WHILE ;

END;

//

DELIMITER ;

1. 循环语句REPEAT...END REPEAT此语句的特点是执行操作后检查结果

DELIMITER //

CREATE PROCEDURE proc6 ()

BEGIN

DECLARE v INT;

SET v=0;

REPEAT

INSERT INTO t VALUES(v);

SET v=v+1;

UNTIL v>=5

END REPEAT;

END;

//

DELIMITER ;

1. 循环语句LOOP...END LOOP

DELIMITER //

CREATE PROCEDURE proc7 ()

BEGIN

DECLARE v INT;

SET v=0;

LOOP\_LABLE:LOOP

INSERT INTO t VALUES(v);

SET v=v+1;

IF v >=5 THEN

LEAVE LOOP\_LABLE;

END IF;

END LOOP;

END;

//

DELIMITER ;

LABLES标号可以用在begin repeat while 或者loop 语句前，语句标号只能在合法的语句前面使用。可以跳出循环，使运行指令达到复合语句的最后一步。

# 触发器

## 简介

**触发器是一种特殊的存储过程。**触发器是一种与表操作有关的数据库对象，**当触发器所在表上出现指定事件时，将调用该对象，即表的操作事件触发表上的触发器的执行。**

## 触发器的创建

1. 具体格式

CREATE TRIGGER 触发器名称

trigger\_time

trigger\_event ON tbl\_name

FOR EACH ROW

trigger\_stmt

1. 参数说明：

trigger\_name：标识触发器名称，用户自行指定；

trigger\_time：标识触发时机，取值为 BEFORE 或 AFTER；

trigger\_event：标识触发事件，取值为 INSERT、UPDATE 或 DELETE；

tbl\_name：标识建立触发器的表名，即在哪张表上建立触发器；

trigger\_stmt：触发器程序体，可以是一句SQL语句，或者用 BEGIN 和 END 包含的多条语句。

由此可见，可以建立6种触发器，即：BEFORE INSERT、BEFORE UPDATE、BEFORE DELETE、AFTER INSERT、AFTER UPDATE、AFTER DELETE。

另外有一个限制是不能同时在一个表上建立2个相同类型的触发器，因此在一个表上最多建立6个触发器。

1. trigger\_event 详解

MySQL 除了对 INSERT、UPDATE、DELETE 基本操作进行定义外，还定义了 LOAD DATA 和 REPLACE 语句，这两种语句也能引起上述6中类型的触发器的触发。

LOAD DATA 语句用于将一个文件装入到一个数据表中，相当与一系列的 INSERT 操作。

REPLACE 语句一般来说和 INSERT 语句很像，只是在表中有 primary key 或 unique 索引时，如果插入的数据和原来 primary key 或 unique 索引一致时，会先删除原来的数据，然后增加一条新数据，也就是说，一条 REPLACE 语句有时候等价于一条。

INSERT 语句，有时候等价于一条 DELETE 语句加上一条 INSERT 语句。

INSERT 型触发器：插入某一行时激活触发器，可能通过 INSERT、LOAD DATA、REPLACE 语句触发；

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；

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。

1. BEGIN … END 详解

BEGIN

[statement\_list]

END

利用DELIMITER指定不同的分隔符

## 触发器实例

1. 假设系统中有两个表

班级表 class(班级号 classID, 班内学生数 stuCount)

学生表 student(学号 stuID, 所属班级号 classID)

要创建触发器来使班级表中的班内学生数随着学生的添加自动更新，代码如下：

DELIMITER //

CREATE TRIGGER tri\_stuInsert AFTER INSERT

ON student FOR EACH ROW

BEGIN

DECLARE c INT;

SET c = (select stuCount from class where classID=new.classID);

UPDATE class SET stuCount = c + 1 WHERE classID = new.classID;

END //

DELIMITER ;

1. NEW 和 OLD 详解

触发器的所在表中，触发了触发器的那一行数据。

具体地：

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；

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；

在 DELETE 型触发器中，OLD 用来表示将要或已经被删除的原数据；

使用方法： NEW.columnName （columnName 为相应数据表某一列名）

另外，OLD 是只读的，而 NEW 则可以在触发器中使用 SET 赋值，这样不会再次触发触发器，造成循环调用（如每插入一个学生前，都在其学号前加“2013”）。

## 查看触发器

SHOW TRIGGERS [FROM schema\_name];

1. 在MySQL中schema和database差不多

## 删除触发器

DROP TRIGGER [IF EXISTS] [schema\_name.]trigger\_name

## 触发器的执行顺序

我们建立的数据库一般都是 InnoDB 数据库，其上建立的表是事务性表，也就是事务安全的。这时，若SQL语句或触发器执行失败，MySQL 会回滚事务，有：

1. 如果 BEFORE 触发器执行失败，SQL 无法正确执行。
2. SQL 执行失败时，AFTER 型触发器不会触发。
3. AFTER 类型的触发器执行失败，SQL 会回滚。

# 游标

## 定义

游标（cursor）就是游动的标识，**一条sql取出对应n条结果资源的接口/句柄，就是游标，沿着游标可以一次取出一行。对查询出来的结果集，一行一行的查看，这个过程就是游标。**

## 游标的特点

(1) 游标是**只读**的，也就是不能更新它；

(2) 游标是不能滚动的，也就是**只能在一个方向上进行遍历**，不能在记录之间随意进退，不能跳过某些记录；

(3) 避免在已经打开游标的表上更新数据。

## 游标的使用 -- 利用存储过程

delimiter //

drop procedure if exists StatisticStore;

CREATE PROCEDURE StatisticStore()

BEGIN

--创建接收游标数据的变量

declare c int;

declare n varchar(20);

--创建总数变量

declare total int default 0;

--创建结束标志变量

declare done int default false;

--创建游标

declare cur cursor for select name,count from store where name = 'iphone';

--指定游标循环结束时的返回值

declare continue HANDLER for not found set done = true;

--设置初始值

set total = 0;

--打开游标

open cur;

--开始循环游标里的数据

read\_loop:loop

--根据游标当前指向的一条数据

fetch cur into n,c;

--判断游标的循环是否结束

if done then

leave read\_loop; --跳出游标循环

end if;

--获取一条数据时，将count值进行累加操作，这里可以做任意你想做的操作，

set total = total + c;

--结束游标循环

end loop;

--关闭游标

close cur;

--输出结果

select total;

END;

--调用存储过程

call StatisticStore();

# 事务管理

## 简介

**一个事务是一个连续的一组数据库操作**，就好像它是一个单一的工作单元进行。换言之，永远不会是完整的事务，除非该组内的每个单独的操作是成功的。如果在事务的任何操作失败，则整个事务将失败。

## MySQL事务和存储引擎

**MySQL的事务支持不是绑定在MySQL服务器本身，而是与存储引擎相关**  
(1) MyISAM：不支持事务，用于只读程序提高性能

(2) InnoDB：支持ACID事务、行级锁、并发

(3) Berkeley DB：支持事务

## 事务的特性

事务有以下四个标准属性的缩写ACID，通常被称为：

1. A（atomicity），原子性。原子性指整个数据库事务是不可分割的工作单位。只有使事务中所有的数据库操作都执行成功，整个事务的执行才算成功。事务中任何一个SQL语句执行失败，那么已经执行成功的SQL语句也必须撤销，数据库状态应该退回到事务前的状态。
2. C（consistency），一致性。一致性是指事务将数据库从一种状态转变为另一种状态。在事务的开始之前和事务结束以后，数据库的完整性约束没有被破坏。
3. I（isolation），隔离性。隔离性还有其他的称呼，如并发控制、可串行化、锁。事务的隔离性要求每个事务读写的对象与其他事务的操作对象能互相分离，即该事务提交前对其他事务都不可见，这通常使用锁来实现。数据库系统中提供了一种粒度锁的策略，允许事务仅锁住一个实体对象的子集，以此来提高事务之间的并发度。（如果是全表锁，事务之间基本就无法实现并发，但是如果只锁住表中处理的行，可以提高事务的并发度）
4. D（durability），持久性。事务一旦提交，其结果就是永久性的**。**即使发生宕机等故障，数据库也能将数据恢复。需要注意的是，持久性只能从事务本身的角度来保证结果的永久性，如事务提交后，所有的变化都是永久的，即使当数据库由于崩溃而需要恢复时，也能保证恢复后提交的数据都不会丢失。

## 事务隔离性实现原理

数据库事务会导致脏读、不可重复读和幻影读等问题。

1. 脏读：一个事务可以读取另一个尚未提交事务的修改数据。
2. 不可重复读：在同一个事务中，同一个查询在T1时间读取某一行，在T2时间重新读取这一行时候，这一行的数据已经发生修改，可能被更新了（update），也可能被删除了（delete）（具体某一行数据）
3. 幻影数据：在同一事务中，同一查询多次进行时候，由于其他插入操作（insert）的事务提交，导致每次返回不同的结果集。（范围数据）

针对可能的问题，InnoDB提供了四种不同级别的机制保证数据隔离性。

事务的隔离用是通过锁机制实现的，不同于MyISAM使用表级别的锁，InnoDB采用更细粒度的行级别锁，提高了数据表的性能。InnoDB的锁通过锁定索引来实现，如果查询条件中有主键则锁定主键，如果有索引则先锁定对应索引然后再锁定对应的主键（可能造成死锁），如果连索引都没有则会锁定整个数据表。

## 事务的隔离级别

5.1 ANSI SQL标准定义的四个隔离级别为：

(1) READ UNCOMMITTED（未提交读）脏读

事务中的修改，即使没有提交，在其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读。

原理：READ UNCOMMIT不会采用任何锁。

1. READ COMMITTED（提交读）解决脏读

一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。这个级别有时候也叫做不可重复读，因为两次执行相同的查询，可能会得到不一样的结果。因为在这2次读之间可能有其他事务更改这个数据，每次读到的数据都是已经提交的。

原理：数据的读是不加锁的，但是数据的写入、修改、删除加锁，避免了脏读。

1. REPEATABLE READ（可重复读），

解决了脏读，也保证了**在同一个事务**中多次读取同样记录的结果是一致的。但是理论上，可重读读隔离级别还是无法解决另外一个幻读的问题，指的是当某个事务在读取某个范围内的记录时，另外一个事务也在该范围内插入了新的记录，当之前的事务再次读取该范围内的记录时，会产生幻行。

原理：数据的读、写都会加锁，当前事务如果占据了锁，其他事务必须等待本次事务提交完成释放锁后才能对相同的数据行进行操作。

1. SERIALIZABLE（可串行化）

它通过强制事务串行执行，同步执行，避免了前面说的幻读的问题

5.2 注意

(1) InnoDB采用MVCC来支持高并发，并实现了四个标准的隔离级别。其默认级别是REPEATABLE READ（可重复读），并且通过间隙锁（next-key locking）策略防止幻读的出现。间隙锁使得InnoDB不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影的插入。

1. 隔离级别越低，事务请求的锁越少或保持锁的时间就越短。所以很多数据库系统默认的事务隔离级别是READ COMMITTED。质疑SERIALIZABLE隔离级别的性能，但是InnoDB存储引擎认为两者的开销是一样的，所以默认隔离级别使用REPEATABLE READ。
   1. 隔离级别的设置
2. 用命令设置当前会话或全局会话的事务隔离级别。

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

{

READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE

}

1. 如果想启动时就设置事务的默认隔离级别，修改MYSQL的配置文件，在[mysqld]中添加如下行：

[mysqld]

transaction-isolation = READ-COMMITTED

## 原子性、稳定性和持久性实现原理

原子性、稳定性和持久性是通过redo 和 undo 日志文件实现的，不管是redo还是undo文件都会有一个缓存我们称之为redo\_buf和undo\_buf。同样，数据库文件也会有缓存称之为data\_buf。

* 1. undo 日志文件

undo记录了数据在事务开始之前的值，当事务执行失败或者ROLLBACK时可以通过undo记录的值来恢复数据。例如 AA和BB的初始值分别为3，5。

1. 具体步骤

A 事务开始

B 记录AA=3到undo\_buf

C 修改AA=1

D 记录BB=5到undo\_buf

E 修改BB=7

F 将undo\_buf写到undo(磁盘)

G 将data\_buf写到datafile(磁盘)

H 事务提交

1. 通过undo可以保证原子性、稳定性和持久性

如果事务在F之前崩溃由于数据还没写入磁盘，所以数据不会被破坏。

如果事务在G之前崩溃或者回滚则可以根据undo恢复到初始状态。

数据在任务提交之前写到磁盘保证了持久性。

但是单纯使用undo保证原子性和持久性需要在事务提交之前将数据写到磁盘，浪费大量I/O。

* 1. redo/undo 日志文件

引入redo日志记录数据修改后的值，可以避免数据在事务提交之前必须写入到磁盘的需求，减少I/O。

1. 具体步骤

A 事务开始

B 记录AA=3到undo\_buf

C 修改AA=1 记录redo\_buf

D 记录BB=5到undo\_buf

E 修改BB=7 记录redo\_buf

F 将redo\_buf写到redo（磁盘）

G 事务提交

1. 通过undo保证事务的原子性，redo保证持久性。

F之前崩溃由于所有数据都在内存，恢复后重新冲磁盘载入之前的数据，数据没有被破坏。

FG之间的崩溃可以使用redo来恢复。

G之前的回滚都可以使用undo来完成。

## 事务的使用

* 1. 关键字COMMIT & ROLLBACK -- 提交和回滚

1. 当一个成功的事务完成后，发出COMMIT命令应使所有参与表的更改才会生效。
2. 如果发生故障时，应发出一个ROLLBACK命令返回的事务中引用的每一个表到以前的状态。
3. 可以控制的事务行为称为AUTOCOMMIT设置会话变量。如果AUTOCOMMIT设置为1（默认值），然后每一个SQL语句（在事务与否）被认为是一个完整的事务，并承诺在默认情况下，当它完成。 AUTOCOMMIT设置为0时，发出SET AUTOCOMMIT =0命令，在随后的一系列语句的作用就像一个事务，直到一个明确的COMMIT语句时，没有活动的提交。
   1. 关键字SAVEPOINT -- 设置回滚点

SAVEPOINT adqoo\_1

ROLLBACK TO SAVEPOINT adqoo\_1

发生在回滚点adqoo\_1之前的事务被提交，之后的事务没有生效。

* 1. 关键字RELEASE SAVEPOINT -- 删除回滚点

删除一个事务的保存点，当没有一个保存点执行这语句时，会抛出一个异常。

* 1. 事务实例

1. 利用BEGIN挂起，COMMIT提交

BEGIN; //开始事务，挂起自动提交

SELECT name FROM 表名称；

INSERT INTO 表名称 (列1 ，列2) VALUES (值1 ，值2)；

COMMIT; //提交事务，恢复自动提交

1. 利用修改autocommit模式，实现事务提交

set autocommit = 0; //挂起自动提交

SELECT name FROM 表名称；

INSERT INTO 表名称 (列1 ，列2) VALUES (值1 ，值2)；

COMMIT; //提交事务

set autocommit = 1; //恢复自动提交 -- 默认为1，（标识符为；）读取到标识符就提交。

## 分布式事务编程

1. 定义

InnoDB存储引擎提供了对于XA事务的支持，并通过XA事务来支持分布式事务的实现。**分布式事务指的是允许多个独立的事务资源参与到一个全局的事务中。**事务资源通常是关系型数据库系统，也可以是其他类型的资源。全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务的ACID要求又有了提高。在使用分布式事务时，InnoDB存储引擎的事务隔离级别必须设置成SERIALIZALE。

* 1. XA事务由一个或多个资源管理器、一个事务管理器、以及一个应用程序组成。

1. 资源管理器：提供访问事务资源的方法。通常一个数据库就是一个资源管理器。
2. 事务管理器：协调参与全局事务中的各个事务。需要和参与到全局事务中的所有资源管理器进行通信。
3. 应用程序：定义事务的边界，指定全局事务中的操作。
4. 在MYSQL数据库的分布式事务中，资源管理器就是MYSQL数据库，事务管理器为连接到MYSQL服务器的客户端。
   1. 分布式事务的提交

分布式事务使用两段式提交（two-phase commit）的方式。

1. 在第一阶段，所有参与全局事务的节点都会开始准备（PREPARE），告诉事务管理器他们准备好了。
2. 第二阶段，事务管理器告诉资源管理器执行ROLLBACK或COMMIT。如果任何一个节点显示不能提交，则所有的节点都会被告知需要回滚。与本地事务不同的是，需要多一次的PREPARE操作，待收到所有节点的同意信息后，再进行COMMIT或ROLLBACK操作。