# 第三讲 MySQL数据库

|  |  |  |  |
| --- | --- | --- | --- |
| **课程大纲** | **课程内容** | **学习效果** | **掌握目标** |
| MySQL | 数据库事务 | 掌握 | 掌握什么是数据库事务、事务的ACID特性、并发异常和隔离级别 |
| DCL | 了解 | 了解DCL语句来创建用户、分配权限、修改密码等 |
| 索引 | 掌握 | 掌握索引概念、优缺点、分类、创建方式等 |
| 存储过程 | 了解 | 了解存储过程的含义、作用、编写和调用方式等。 |
| 存储引擎 | 了解 | 了解MySQL的常见存储引擎和InnoDB存储引擎的特点 |

## 数据库事务

### （一）什么是数据库事务

数据库事务是访问并可能操作各种数据项的一个数据库操作（包括读、写）序列，这些操作要么全部执行,要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

例如：银行转账。核心操作包括更新一个账户减去转账金额，更新另一个账户加上转账金额，要求要同时成功或同时失败。

### （二）事务的目的

1. 为数据库操作提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。
2. 当多个应用程序在并发方法数据库时，可以再这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

总结：事务的目的就是要提供3种方法：1、失败恢复方法。 2、保持一致性的方法。3、操作隔离的方法。

**成功情况下**

能将数据从一种状态变为另一种状态，并能够持久化。

**异常情况下**

1. 能将数据恢复到正常状态。
2. 要能保证一致性，包含数据的一致性和约束的一致性。

**并发情况下**

并发的操作之间不能产生相互影响。

### 事务特性

事务包括ACID四个特性，分别是原子性、一致性、隔离性、持久性。

#### 原子性（Actomicity）

表示一个事务内的所有操作是一个整体，是一个不可分割的工作单位，要么全部成功，要么全部失败。

#### 一致性（Consistency）

事务前后数据的完整性必须保持一致。

#### 隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

#### 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

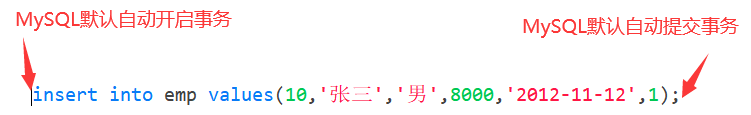
### 操作事务

在数据库中，一个事务可以由多条SQL语句组成，当然一个事务中也可以只包含一条SQL语句。

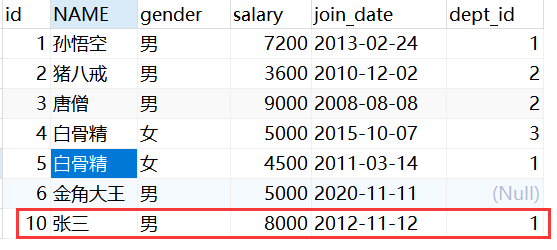
MYSQL中可以有两种方式进行事务的操作：1、自动提交事务。2、手动提交事务。

#### 自动提交事务

MySQL的每一条DML(增删改)语句都是一个单独的事务，MySQL默认情况下，在执行每条语句时都会**自动开启**一个事务，执行完毕**自动提交事务**。



当语句执行完成后，数据库中添加了数据。



#### 手动提交事务

事务有关的SQL语句

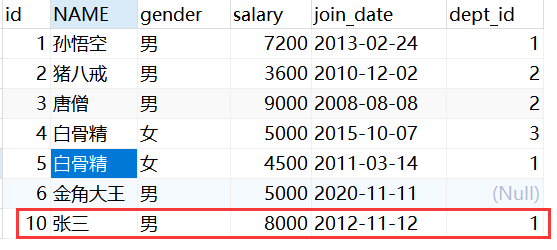
| SQL语句 | 描述 |
| --- | --- |
| start transaction; | 开启事务 |
| commit; | 提交事务 |
| rollback; | 回滚事务 |

**示例**

start transaction; --开启事务

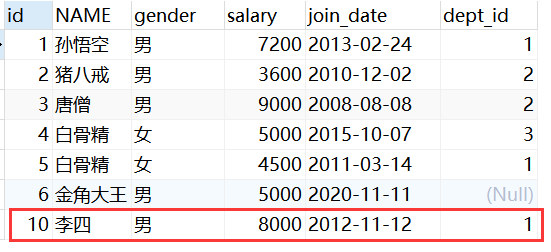
update emp set name='李四' where id=10; -- 更新数据

在没有提交事务时，数据库中的数据不会真正的发生变化



commit;-- 提交事务

事务提交后，数据库中的数据才会真正的变化



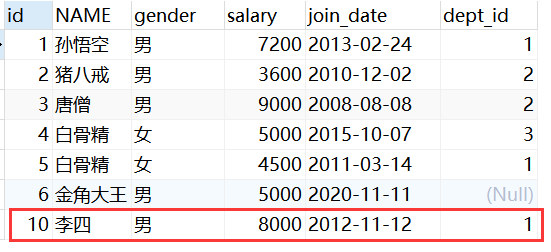
**示例2**

start transaction; --开启事务

update emp set name='王五' where id=10; -- 更新数据

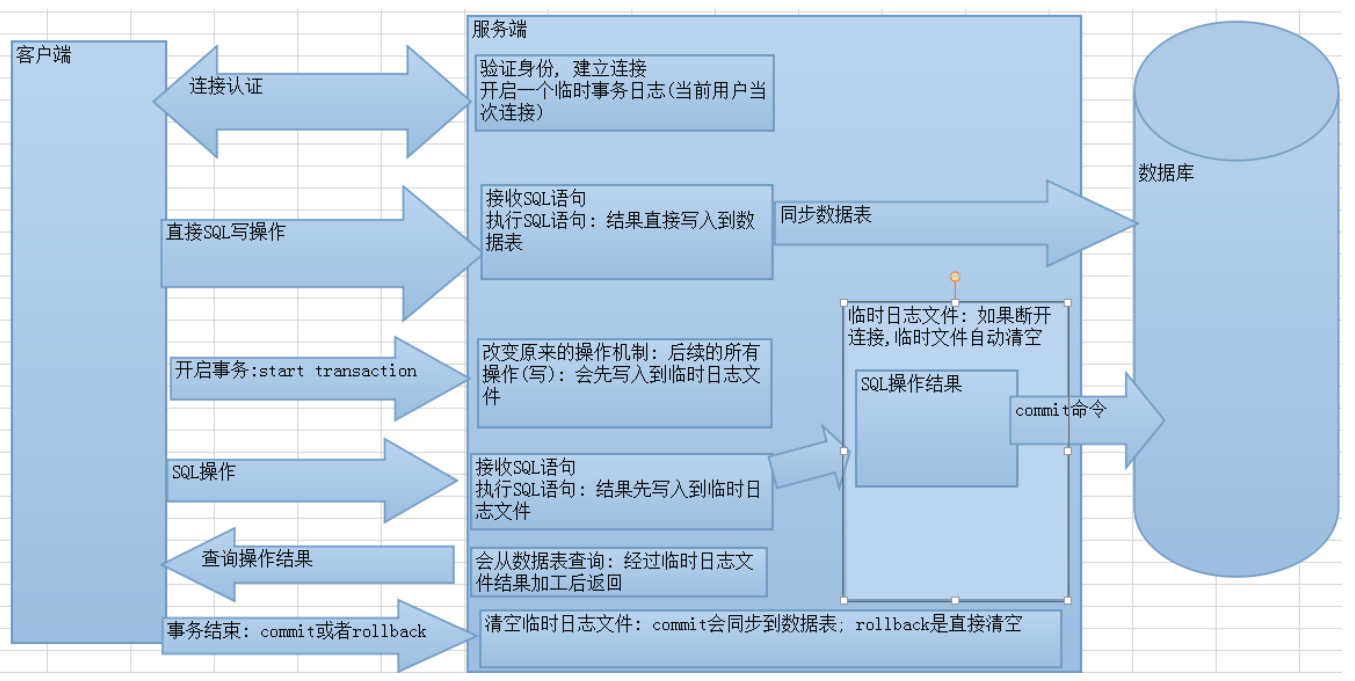
如果不提交事务，而是回滚事务，数据库中的数据将不会变化

rollback;



#### 事务的原理

事务开启之后, 所有的操作都会临时保存到事务日志, 事务日志只有在得到commit命令才会同步到数据表中，其他任何情况都会清空事务日志(rollback，断开连接)



### （五）事务的并发异常

在事务并发操作时（例如多个客户端来操作同一个数据库中的同一张表）。可能会出现如下的问题。

|  |  |
| --- | --- |
| **异常** | **描述** |
| 回滚丢失 | 此种更新丢失是因为回滚的原因，所以叫做回滚丢失。 |
| 覆盖丢失 | 此种更新是因为更新被其他事务覆盖了，所以叫做覆盖更新 |
| 脏读 | 一个事务读取了另一个事务尚未提交的数据 |
| 不可重复读 | 一个事务对同一行数据执行了两次或以上的查询，结果却不同 |
| 幻读 | 幻读和不可重复读有点像，只是针对的一般是数据的数量 |

当然这些并发的异常可以说是由数据库的隔离级别引起的，或者说通过调整数据库的隔离级别可以防止以上并发异常的发生。

### （六）隔离级别

数据库的事务隔离级别主要包括：读未提交(Read Uncommitted)、读已提交(Read Committed)、可重复读(Repeatable Read)、串行化(Serializable)。隔离级别越高，数据库的效率就越低。

MySQL数据库InnoDB存储引擎（默认）的默认隔离级别是**Repeatable Read**

Oracle数据库默认的隔离级别是**Read Committed**

各种隔离级别能解决哪些并发异常

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **隔离级别/并发异常** | **回滚丢失** | **脏读** | **不可重复读** | **覆盖丢失** | **幻读** |
| **读未提交** | **×** | **×** | **×** | **×** | **×** |
| **读已提交** | **√** | **√** | **×** | **×** | **×** |
| **可重复读** | **√** | **√** | **√** | **√** | **×** |
| **串行化** | **√** | **√** | **√** | **√** | **√** |

#### 1、隔离级别的查看和修改

##### （1）MySQL8以前版本

**查看**

select @@session.tx\_isolation

**修改**

set @@session.tx\_isolation=’隔离级别’

##### MySQL8

**查看**

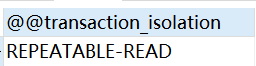
select @@transaction\_isolation

**修改**

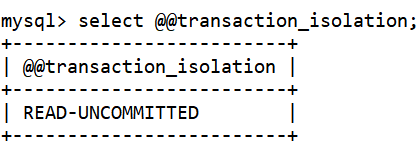
set session transaction isolation level 隔离级别

**示例**

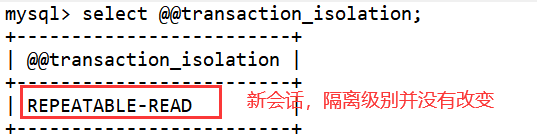
select @@transaction\_isolation



set session transaction isolation level read uncommitted



当然，修改隔离级别，只是修改当前会话的隔离级别。如果打开另外一个会话查看，隔离级别没有改变。



#### 2、隔离级别应用

**准备数据**

create database tx character set utf8;

create table stu(

id int primary key auto\_increment,

name varchar(20),

age int

);

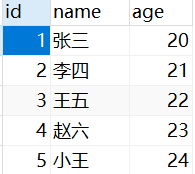
insert into stu values(null,'张三',20);

insert into stu values(null,'李四',21);

insert into stu values(null,'王五',22);

insert into stu values(null,'赵六',23);

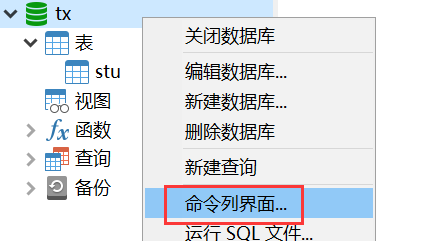
insert into stu values(null,'小王',24);

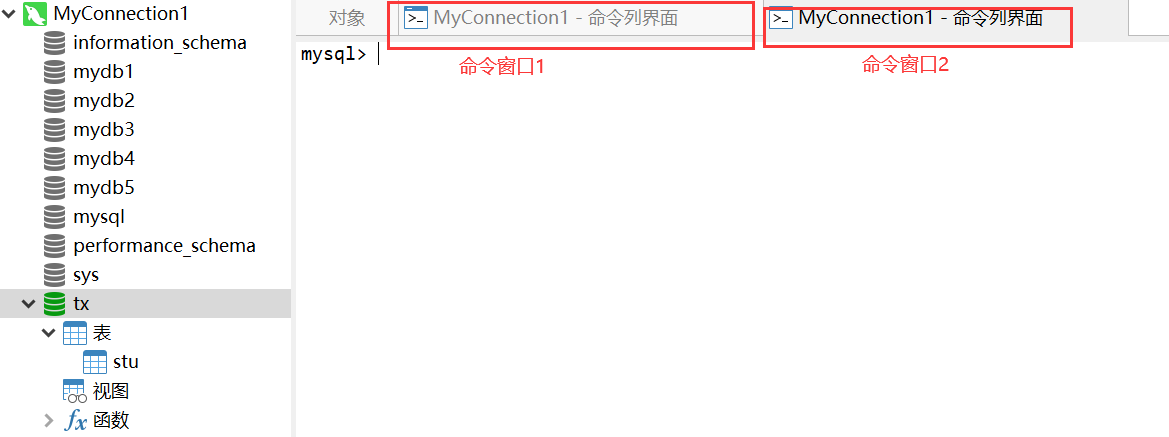


##### （1）Read Uncommitted

###### a、测试脏读

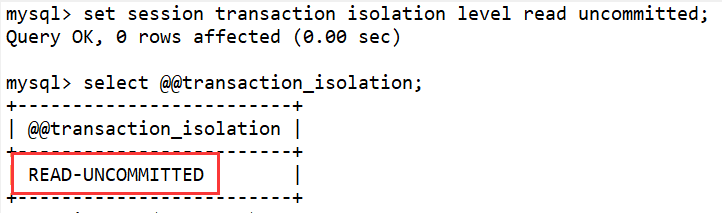
在Navicat中开启两个命令窗口，每一个窗口就相当于一个线程。





**1、把两个命令窗口的隔离级别都修改成 Read Uncommitted**

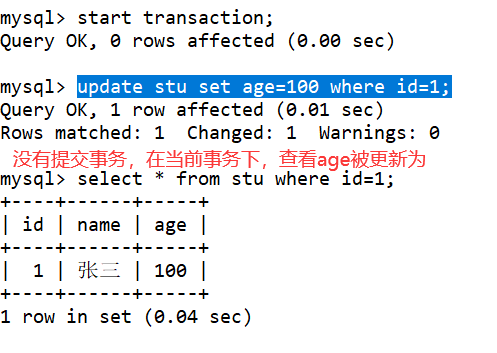
set session transaction isolation level read uncommitted



**2、在窗口1开启事务，更新id为1的年龄为100，不提交事务。**

start transaction;

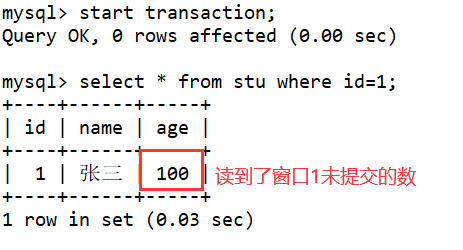
update stu set age=100 where id=1;



1. **在命令窗口2中，查看id为1的数据**

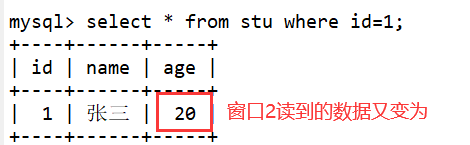
start transaction;

select \* from stu where id=1;



1. **如果窗口1回滚，窗口2读到的数据又会发生变化**



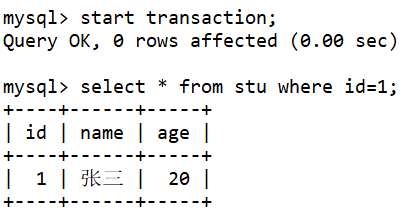


###### b、测试不可重复读

**窗口1开启事务查询id为1的数据**

start transaction;

select \* from stu where id=1;

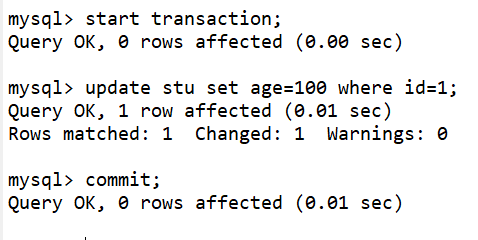


**2、窗口2开启事务，更新id为1的age为100**

start transaction;

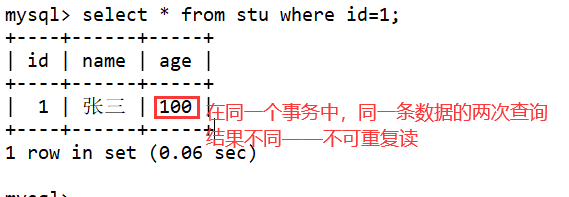
update stu set age=100 where id=1;

commit;



1. **回到窗口1，还是在原来的事务中，查询id为1的数据**

select \* from stu where id=1;

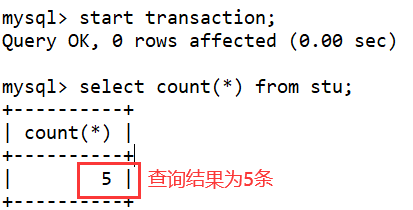


###### c、测试幻读

**1、在窗口1中开启事务，查询stu表的数据总条数**

start transaction;

select count(\*) from stu;

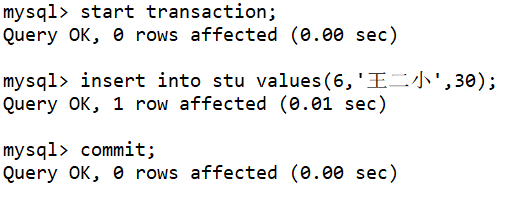


**2、在窗口2中开启事务，插入一条新数据**

start transaction;

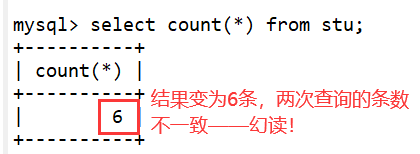
insert into stu values(6,'王二小',30);

commit;



**3、回到窗口1，还是在当前事务中，再次查询数据条数**

select count(\*) from stu;



###### d、测试丢失更新

关于丢失更新的说明

**回滚丢失**

|  |  |
| --- | --- |
| **事务A** | **事务B** |
| 查询数据得到初始值20 | 查询数据得到初始值20 |
| 更新为20+1=21 | 更新为20+2=22 |
| 提交 | 失败回滚 |
|  | 数据回到了初始状态20，导致事务A的更新丢失了，由于回滚导致的丢失叫做回滚丢失 |

**覆盖丢失**

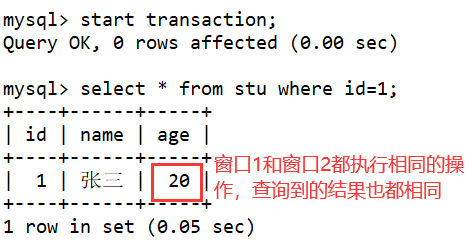
|  |  |
| --- | --- |
| **事务A** | **事务B** |
| 查询数据得到初始值20 | 查询数据得到初始值20 |
| 更新为20+1=21 | 更新为20+2=22 |
| 提交 | 提交 |
|  | 最终结果变为22，由于事务B是从20开始计算，并不知道数据已经发生了变化，所以导致事务A的更新被覆盖了，所以叫做覆盖更新 |

**测试回滚丢失**

1. **窗口1和窗口2都开启事务，查询id为1的数据的信息**

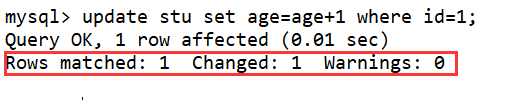
start transaction;

select \* from stu where id=1;



1. **窗口1更新id为1的age=age+1**

update stu set age=age+1 where id=1;



操作成功！！！

1. **窗口2更新id为1的age=age+2**

update stu set age=age+2 where id=1;



**会发现窗口2的更新会阻塞！这是因为在mysql数据库中，在窗口1更新这条数据没有提交时，对该条数据加锁了，导致窗口2无法更新这条数据。只有当窗口1条件是，窗口2的更新会自动执行。**

**窗口1**

**commit;**

**窗口2**

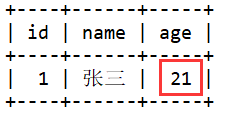


1. **窗口2回滚**

rollback;

1. **回到窗口1查询数据**

select \* from stu where id=1;



**我们会发现，并没有出现我们预期的丢失更新的情况！**

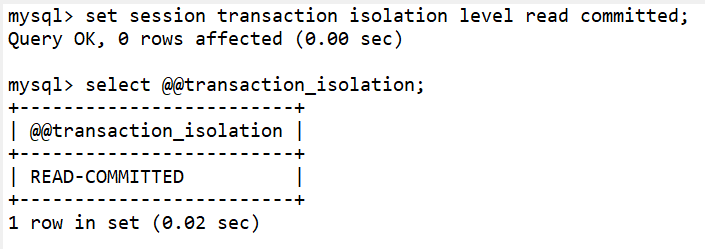
**这是因为MySQL的锁引起的，无论使用哪种隔离级别都无法掩饰出丢失更新的情况！除非关闭MySQL的锁。**

##### （2）Read Committed

###### a、测试脏读

**1、把两个命令窗口的隔离级别都修改成 Read Committed**

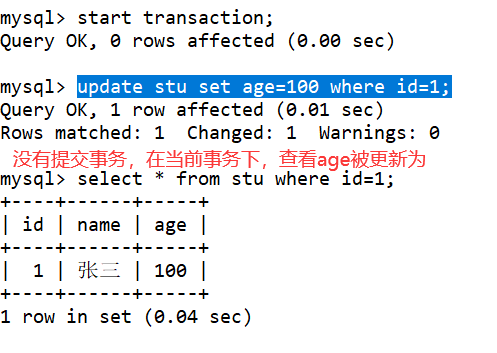
set session transaction isolation level read committed



**2、在窗口1开启事务，更新id为1的年龄为100，不提交事务。**

start transaction;

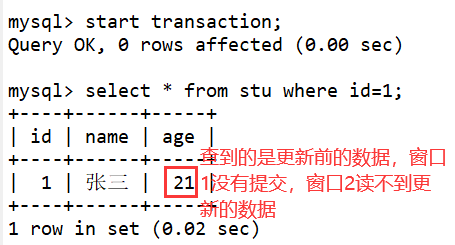
update stu set age=100 where id=1;



**3、在命令窗口2中，查看id为1的数据**

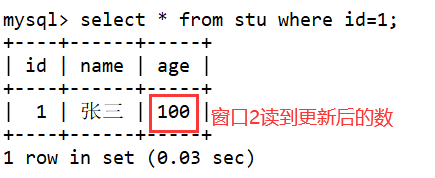
start transaction;

select \* from stu where id=1;



**4、窗口1提交，窗口2才能读到数据变化**



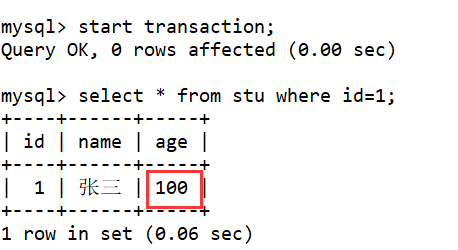


###### b、测试不可重复读

**窗口1开启事务查询id为1的数据**

start transaction;

select \* from stu where id=1;

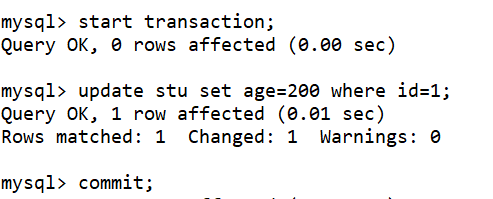


**2、窗口2开启事务，更新id为1的age为200**

start transaction;

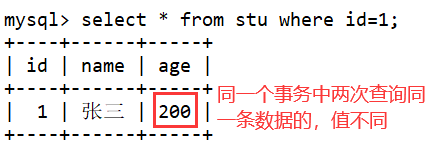
update stu set age=200 where id=1;

commit;



**3、回到窗口1，还是在原来的事务中，查询id为1的数据**

select \* from stu where id=1;



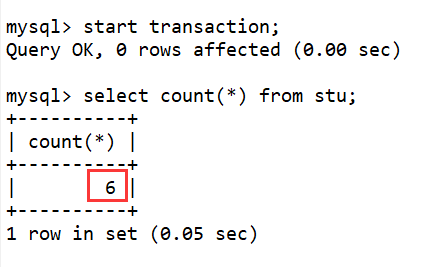
**不可重复读**

###### c、测试幻读

**1、在窗口1中开启事务，查询stu表的数据总条数**

start transaction;

select count(\*) from stu;

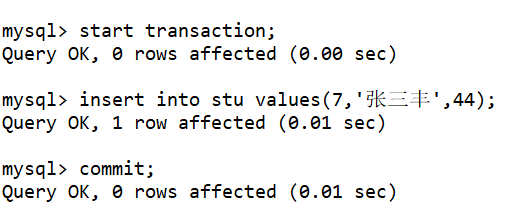


**2、在窗口2中开启事务，插入一条新数据**

start transaction;

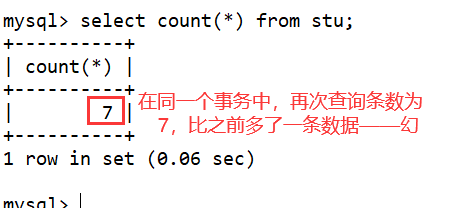
insert into stu values(7,'张三丰',44);

commit;



**3、回到窗口1，还是在当前事务中，再次查询数据条数**

select count(\*) from stu;

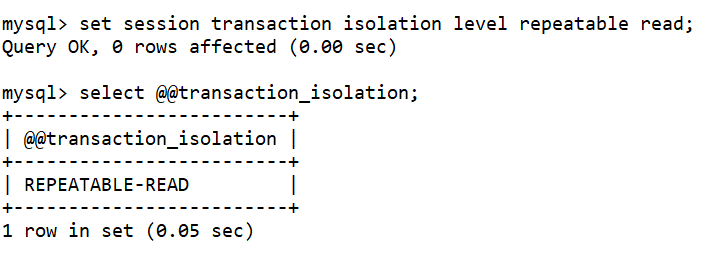


##### （3）Repeatable Read

###### a、测试脏读

**1、把两个命令窗口的隔离级别都修改成** Repeatable Read

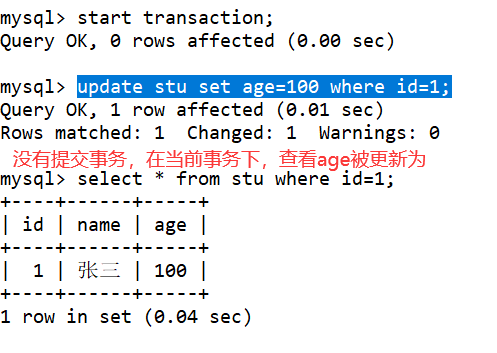
set session transaction isolation level repeatable read;



**2、在窗口1开启事务，更新id为1的年龄为100，不提交事务。**

start transaction;

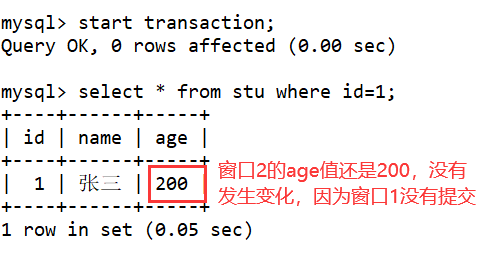
update stu set age=100 where id=1;



**3、在命令窗口2中，查看id为1的数据**

start transaction;

select \* from stu where id=1;



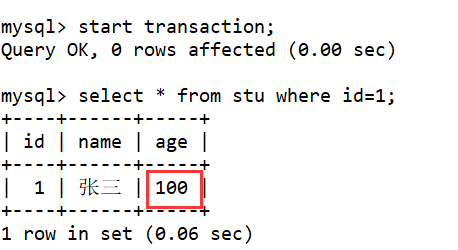
**不存在脏读**

###### b、测试不可重复读

**窗口1开启事务查询id为1的数据**

start transaction;

select \* from stu where id=1;

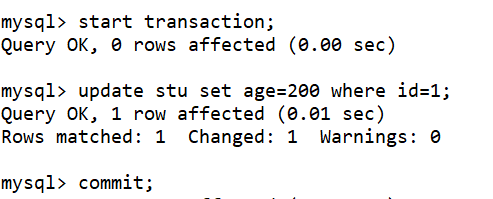


**2、窗口2开启事务，更新id为1的age为200**

start transaction;

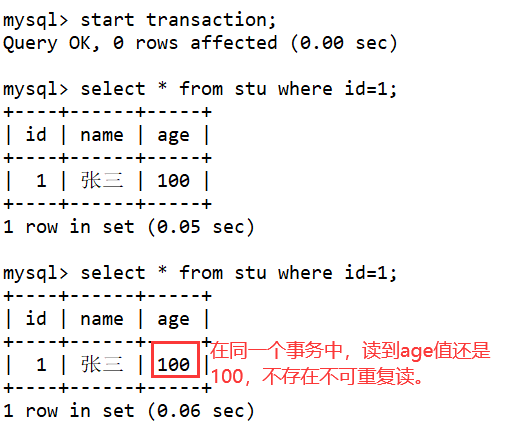
update stu set age=200 where id=1;

commit;



**3、回到窗口1，还是在原来的事务中，查询id为1的数据**

select \* from stu where id=1;

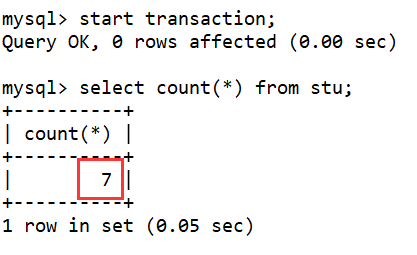


###### c、测试幻读

**1、在窗口1中开启事务，查询stu表的数据总条数**

start transaction;

select count(\*) from stu;

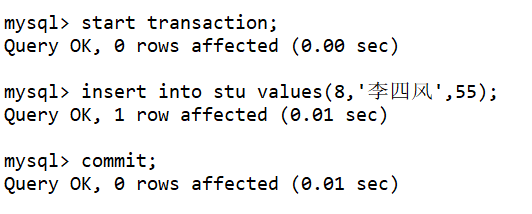


**2、在窗口2中开启事务，插入一条新数据**

start transaction;

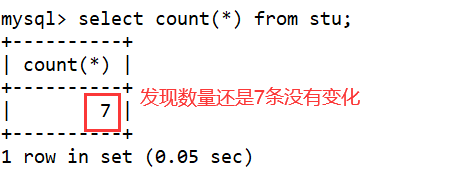
insert into stu values(8,'李四风',55);

commit;



**3、回到窗口1，还是在当前事务中，再次查询数据条数**

select count(\*) from stu;

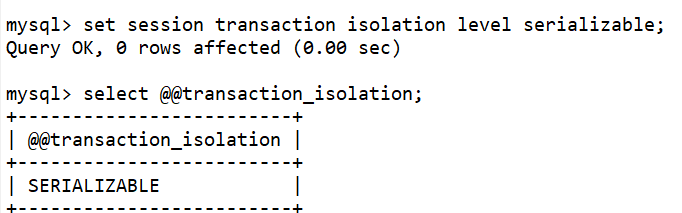


**我们发现，MySQL在Repeatable Read的隔离级别下，也不存在幻读了。这是因为MySQL中采用了next-key和gap-lock的机制的问题，这里不进行深入的探讨**

##### （4）Serializable

**1、窗口1和窗口2的隔离级别都设置为Serializable**

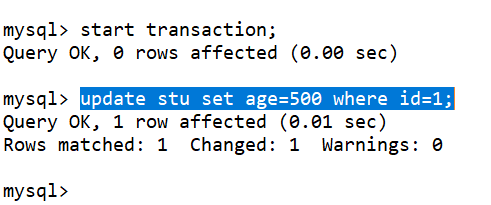
set session transaction isolation level serializable;



**2、窗口1开启事务，更新id为1的age为500，不提交事务**

start transaction;

update stu set age=500 where id=1;



**3、窗口2中开启事务查询id为1的数据**

select \* from stu where id=1；



**我们会发现，当窗口1不提或回滚结束事务时，窗口2不能进行任何操作。因此就也不存在脏读、不可重复读和幻读。**

## DCL

我们现在默认使用的都是root用户，超级管理员，拥有全部的权限。但是，一个公司里面的数据库服务器上面可能同时运行着很多个项目的数据库。所以，我们应该可以根据不同的项目建立不同的用户，分配不同的权限来管理和维护数据库。用户和权限的管理可以通过DCL（Data Control Language）来实现。

### 创建用户

**语法**

CREATE USER '用户名'@ '主机名' IDENTIFIED BY '密码'

1、`用户名`：将创建的用户名

2、`主机名`：指定该用户在哪个主机上可以登陆，如果是本地用户可用localhost，如果想让该用户可以从任意远程主机登陆，可以使用通配符%

3、`密码`：该用户的登陆密码，密码可以为空，如果为空则该用户可以不需要密码登陆服务器

**示例**

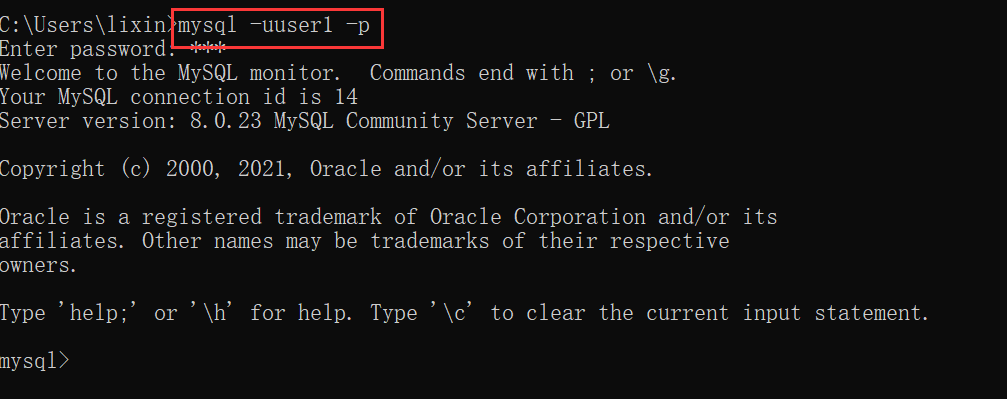
-- user1用户只能在localhost这个IP登录mysql服务器

CREATE USER 'user1'@'localhost' IDENTIFIED BY '123';

-- user2用户可以在任何电脑上登录mysql服务器

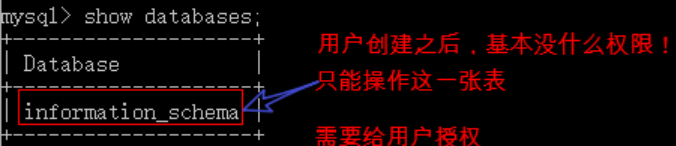
CREATE USER 'user2'@'%' IDENTIFIED BY '123';

使用新创建user1登录



### 授权用户

用户创建之后，基本没什么权限！需要给用户授权。



**语法**

GRANT 权限1, 权限2... ON 数据库名.表名 TO '用户名'@'主机名';

1、`GRANT` 授权关键字

2、授予用户的权限，如`SELECT`，`INSERT`，`UPDATE`等。如果要授予所的权限则使用`ALL`

3、 `数据库名.表名`：该用户可以操作哪个数据库的哪些表。如果要授予该用户对所有数据库和表的相应操作权限则可用表示，如`\*.\*`

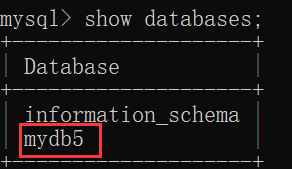
4、 `'用户名'@'主机名'`: 给哪个用户授权

**示例1**

给user1用户分配对mydb5这个数据库操作的权限。

GRANT CREATE,ALTER,DROP,INSERT,UPDATE,DELETE,SELECT ON mydb5.\* TO 'user1'@'localhost';

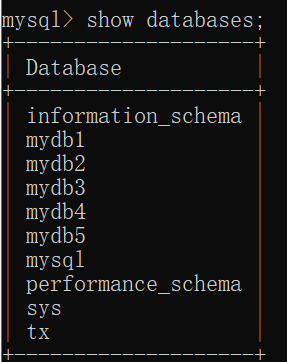
**注意要使用root用户进行授权**



**示例2**

给user2用户分配对所有数据库操作的权限。

GRANT ALL ON \*.\* TO 'user2'@'%';



### 查看权限

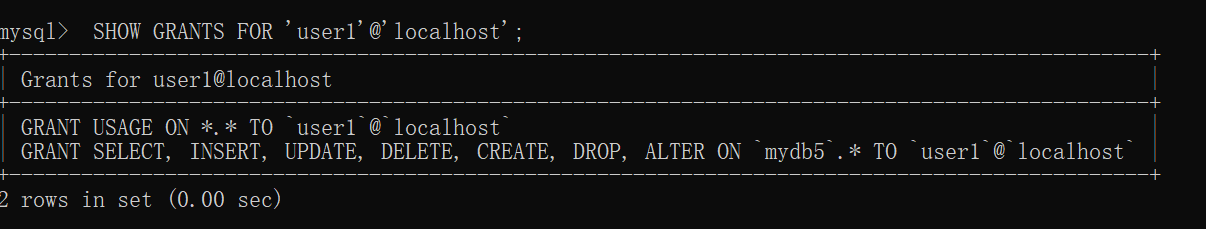
**语法**

SHOW GRANTS FOR '用户名'@'主机名';

**示例**

查看user1用户的权限

SHOW GRANTS FOR 'user1'@'localhost';



### 删除用户

**语法**

DROP USER '用户名'@'主机名';

**示例**

删除user2

DROP USER 'user2'@'localhost';

### 修改管理员密码

**语法**

mysqladmin -uroot -p password 新密码 -- 新密码不需要加上引号

**示例**

mysqladmin -uroot -p password 123456

输入老密码

**注意：要在未登录的情况下操作**

### 修改普通用户密码

**语法**

set password for '用户名'@'主机名' ='新密码';

**示例**

set password for 'user1'@'localhost' = '666666';

**注意：要在登录情况下操作**

## 索引

### 索引概述

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。

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。

一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。

### 优缺点

#### 1、优点

类似大学图书馆建书目索引，提高数据检索的效率，降低数据库的IO成本。

通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗。

#### 2、缺点

实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行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每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句。

### 分类

分为普通索引、唯一索引和组合索引（多列索引）。

#### 普通索引

##### 特点

普通索引使用没有限制，允许在所有数据类型上创建，可以有重复值和空值。

一个索引只包含单个列，一个表可以有多个单列索引。

##### 创建普通索引

###### 建表时创建

**语法**

INDEX 索引名 (列名)

**示例**

create table student(

id int primary key,

name varchar(20),

age int,

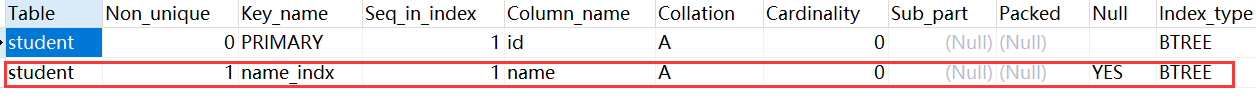
index name\_indx(name)

)



**测试是否创建成功**

show index from student



###### 修改表方式创建

**语法**

ALTER TABLE 表名 ADD INDEX 索引名(列名)

**示例**

create table student2(

id int primary key,

name varchar(20),

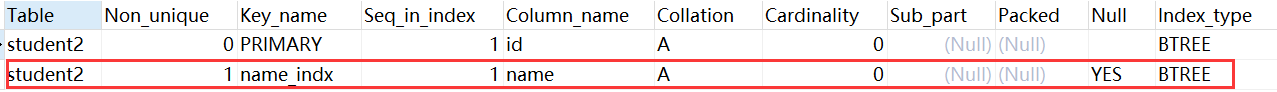
age int

)

ALTER TABLE student2 ADD INDEX name\_indx(name)

**测试是否创建成功**

show index from student2



###### 直接创建

**语法**

CREATE INDEX 索引名 ON 表名(列名)

**示例**

create table student3(

id int primary key,

name varchar(20),

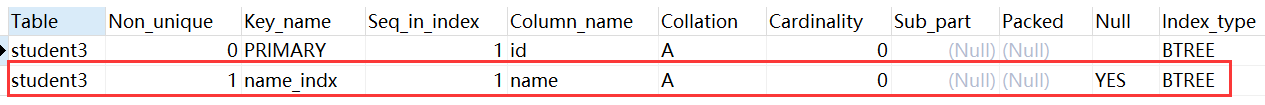
age int

)

CREATE INDEX name\_indx ON student3(name)

**测试是否创建成功**

show index from student3



#### 唯一索引

##### 特点

与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须是唯一的，创建方法和普通索引类似。

##### 创建唯一索引

###### 建表时创建

**语法**

UNIQUE INDEX 索引名 (列名)

**示例**

create table student4(

id int primary key,

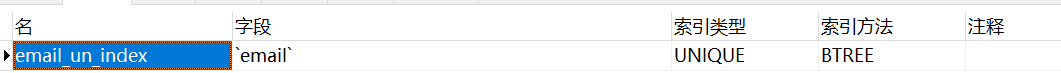
name varchar(20),

age int,

email varchar(200),

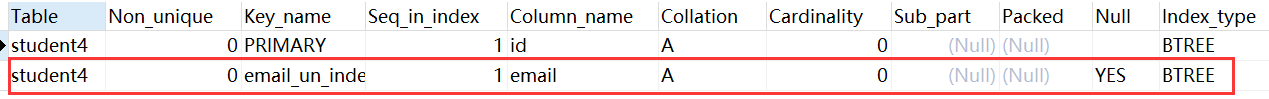
UNIQUE INDEX email\_un\_index(email)

)



**测试是否创建成功**

show index from student4



###### 修改表方式创建

**语法**

ALTER TABLE 表名 ADD UNIQUE INDEX 索引名(列名)

**示例**

create table student5(

id int primary key,

name varchar(20),

age int,

email varchar(200)

)

ALTER TABLE student5 ADD UNIQUE INDEX email\_un\_indx(email)



###### 直接创建

**语法**

CREATE UNIQUE INDEX 索引名 ON 表名(列名)

**示例**

create table student6(

id int primary key,

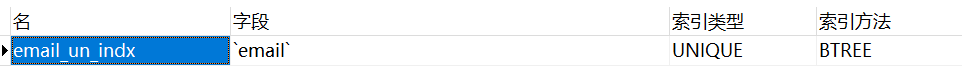
name varchar(20),

age int,

email varchar(200)

)

CREATE UNIQUE INDEX email\_un\_indx ON student6(email)



###### 添加唯一约束方式

**示例**

create table student7(

id int primary key,

name varchar(20),

age int,

email varchar(200) unique

)



添加了唯一约束的列，数据库会直接添加唯一索引。

#### 组合索引

##### （1）特点

同时在多个列上添加索引。当查询时，条件是添加了索引的列的组合，可以提高查询效率。

平时用的SQL查询语句一般都有比较多的限制条件，所以为了进一步榨取MySQL的效率，就要考虑建立组合索引。

##### （2）创建组合索引

###### 建表时创建

**语法**

INDEX 索引名 (列名1,列名2...,列名n)

**示例**

create table student8(

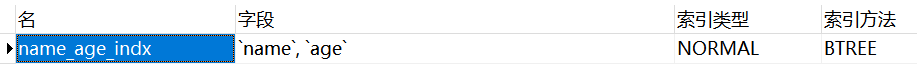
id int primary key,

name varchar(20),

age int,

**index name\_age\_indx(name,age)**

)



###### 修改表方式创建

**语法**

ALTER TABLE 表名 ADD UNIQUE INDEX 索引名 (列名1,列名2...,列名n)

**示例**

create table student9(

id int primary key,

name varchar(20),

age int

)

ALTER TABLE student9 ADD UNIQUE INDEX name\_age\_indx(name,age)



###### 直接创建

**语法**

CREATE INDEX 索引名 ON 表名(列名1,列名2...,列名n)

**示例**

create table student10(

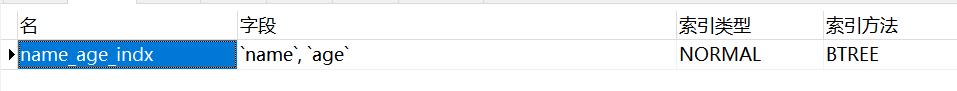
id int primary key,

name varchar(20),

age int

);

CREATE INDEX name\_age\_indx ON student10(name,age);



### 创建索引的条件

#### 1、创建索引

1、主键自动建立唯一索引。

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。

1. 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。

5、普通/组合索引的选择问题，在高并发下倾向创建组合索引。

#### 2、不创建索引

1、频繁更新的字段不适合创建索引。

因为每次更新不单单是更新了记录还会更新索引，加重了IO负担

2、Where条件里用不到的字段不创建索引

3、表记录太少。

4、经常增删改的表

5、如果某个数据列包含许多重复的内容，为它建立索引就没有太大的实际效果。

### （五）索引使用注意事项

#### 1、索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。Is null和is not null 无法使用索引。

#### 2、使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

#### 3、索引列排序

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序在符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

#### 4、like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

#### 5、不要在列上进行运算

例如：select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。关于这一点可以围观：一个单引号引发的MySQL性能损失。

#### 6、MySQL使用不等于(!= 或者<>)时无法使用索引会导致全表扫描

在MySQL的查询语句的查询条件中如果使用了不等于，那么将索引失效，进行全表扫描方式查询。

#### 7、最左前缀法则

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。

假设index(a,b,c)

|  |  |
| --- | --- |
| **where语句** | **索引是否被使用** |
| where a = 3 | Y,使用到a |
| where a = 3 and b = 5 | Y,使用到a，b |
| where a = 3 and b = 5 and c = 4 | Y,使用到a,b,c |
| where a = 3 and c = 5 | 使用到a， 但是c不可以，中间断了 |

## 存储过程

### （一）、什么是存储过程

存储过程是事先编译好并存储在数据库中的一段SQL语句的集合，可以重复直接调用。

### （二）、存储过程的优缺点

#### 1、存储过程优点

(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，因此提高了执行效率。

(2)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。

(3)存储过程是安全的。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、存储过程缺点

(1)可移植性差。

(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。

### （三）创建存储过程

**语法**

delimiter 符号

create procedure 名称([参数])

begin

sql语句

end

符号

delimiter;

#### 1、简单的存储过程

**创建查询dept表中数据条数的存储过程**

Delimiter$$

create procedure dept\_count\_pro()

comment '查询dept表数据条数的存储过程'

begin

select count(\*)from dept;

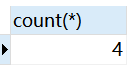
end;

$$

delimiter;

**调用存储过程：**

call dept\_count\_pro();



**说明**

comment '查询dept表数据条数的存储过程' 表示对存储过程的说明！

在MySQL中，默认使用;来表示一条语句的结束。在创建存储过程时，为了避免冲突，使用delimiter命令来改变MySQL数据库中结束符，将其改变为 $$。注意，使用delimiter命令可以自定义结束符，在创建完存储过程后使用delimiter;命令将结束符更改回MySQL原有的;。

#### 2、带输入参数的存储过程

向dept表中添加数据的存储过程

delimiter $$

create procedure dept\_insert\_pro(**in id int,in dept\_name varchar(20)**)

begin

declare v1 int default 0; /\* 定义变量\*/

declare v2 varchar(20); /\* 定义变量\*/

set v1=id; /\* 把参数赋给变量\*/

set v2 = dept\_name; /\* 把参数赋给变量\*/

insert into dept(id,name) values(v1,v2);

end

**$$**

delimiter;

call dept\_insert\_pro(5,'后勤部');



在声明和使用变量时，可以采用另外一种方式，可以省略declare

delimiter $$

create procedure dept\_insert\_pro2(**in id int,in dept\_name varchar(20)**)

begin

**set @v1=id;** /\* 声明变量并赋值\*/

**set @v2 = dept\_name;** /\* 声明变量并赋值\*/

insert into dept(id,name) values(**@v1,@v2**);

end

$$

delimiter;

call dept\_insert\_pro2(6,'安保部');



#### 3、带输出参数的存储过程

获取dept表中的id最大值

delimiter $$

create procedure dept\_getmaxid\_pro(**out maxid int**)

begin

select max(id) **into maxid** from dept;

end

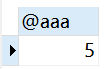
$$

delimiter;

call dept\_getmaxid\_pro(**@aaa**)

调用该存储过程，注意：输出参数必须是一个带@符号的变量

select **@aaa**



#### 4、带输入和输出参数的存储过程

delimiter $$

create procedure sp(in p1 int , out p2 int)

begin

**if** p1 = 1 **then**

set @v = 100;

**else**

set @v = 200;

**end if;**

/\* 语句体内可以执行多条sql，但必须以分号分隔 \*/

insert into dept(id) values(@v);

select max(id) into p2 from dept;

end

$$

delimiter;

call sp(1,@aaa)

select @aaa





**说明**

存储过程中，可以使用if then ... else ... end if; 结构来做条件判断语句。

#### 5、既做输入又做输出参数的存储过程

delimiter $$

create procedure sp4(inout p4 int)

begin

if p4 = 4 then

set @pg = 400;

else

set @pg = 500;

end if;

select @pg;

end

$$

delimiter;

set @pp = 4;

call sp4(@pp)



## 五、存储引擎

### （一）概述

存储引擎是MySQL中特有的术语，主要是指在MySQL中，用户可以根据应用的需求，选择如何存储、更新、查询和索引数据，是否使用事务等。

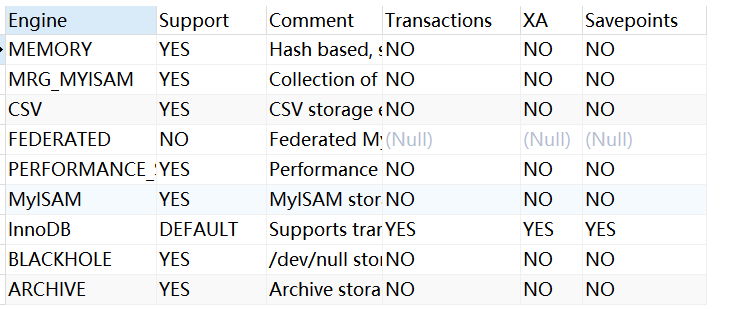
在MySQL中，支持多种不同类型的存储引擎，从而满足用户使用各种方式存储数据的需求。

### 类型

MySQL提供多种存储引擎，但不同版本的MySQL所提供的存储引擎类型稍有不同。可以使用SQL语句查看当前版本数据库中都提供了哪些存储引擎。

语句

show engines;



当前8.0.23包含9种存储引擎。

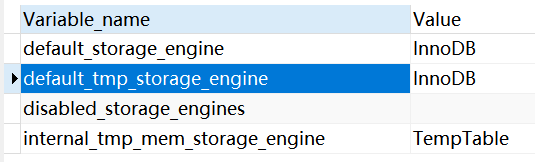
### 常见存储引擎简介

MySQL8.0.23中包括9种存储引擎。其中InnoDB、MyISAM、MRG\_MYISAM(又称为MERGE)、MEMORY、BLACKHOLE比较常见。

|  |  |
| --- | --- |
| **存储引擎** | **特点** |
| InnoDB | 支持事务处理，保证事务安全；支持外键；采用行锁 |
| MyISAM | 不支持事务的完整性和并发性，采用表锁；访问速度快；占用空间少 |
| MRG\_MYISAM | 将多个MyISAM表合成一个单表；不支持事务处理；采用表锁 |
| MEMORY | 使用存储在内存中的内容创建表，所有数据存放在内存中，因此读取速度快，但容易丢失数据 |
| BLACKHOLE | 不用于存储数据，而是将数据存入binlog中 |

查看当前系统使用的存储引擎

show variables like '%storage\_engine%';



### （四）InnoDB存储引擎介绍

在MySQL被Oracle收购之后，其开发重心转移到InnoDB存储引擎上，并且在MySQL5.5之后默认存储引擎改为了InnoDB。

#### 1、存储结构

在InnoDB存储引擎中，所有数据都被逻辑地存储在表空间（tablespace）中，表空间又由段（segment）、区（extent）、页（page）组成。其中，一个表空间包括多个段，一个段由多个区组成，一个区内又可以分为多个页。

##### 表空间

表空间可以分为系统表空间、独立表空间和临时表空间。

当数据库初始化时，一些初始信息都是被存在一个叫做ibdata1的表空间文件中。对于系统表空间，其大小随着数据量的增加会自动扩展，一般默认自动扩展大小是64M。

独立表空间意味着数据库中的每个表都有自己的表空间文件，使用独立表空间只需将innodb\_file\_per\_table参数设置为1即可。

临时表空间是从5.7版本开始从系统表空间中抽离出来而形成的，将临时表中的相关信息保存在其中。

##### 段

在InnoDB存储引擎中，常见的段主要有数据段、索引段、回滚段。

##### 区

区由连续的页构成，在物理存储上由系统分配一段连续的空间进行存储。每个区由64个连续的页构成，大小都是1MB。

##### 页

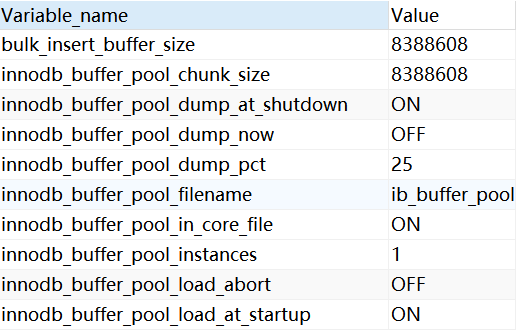
页是InnoDB存储引擎中最小的存储分配单位，默认大小为16KB。

#### 2、InnoDB内存结构

在MySQL中，当使用InnoDB存储引擎时，其内存结构与Oracle数据库的内存结构类似，分为系统全局区（SGA）和程序缓冲区（PGA）两部分。

用户可以查看InnoDB存储引擎的内存结构参数，SQL语句如下：

show variables like '%buffer%';



**系统全局区常用参数**

|  |  |
| --- | --- |
| **参数** | **作用** |
| innodb\_buffer\_pool | 缓存表中的数据、索引、插入缓冲、数据字典等信息 |
| innodb\_log\_buffer\_size | 设置事务在内存中的缓存大小 |
| innodb\_sort\_buffer\_size | 设置InnoDB进行排序时的缓存大小 |

**程序缓存区常用参数**

|  |  |
| --- | --- |
| **参数** | **作用** |
| sort\_buffer\_size | 设置用于SQL语句在内存中排序的缓存大小 |
| join\_buffer\_size | 设置用于表连接的缓存大小 |
| read\_buffer\_size | 设置用于表顺序扫描的缓存大小 |
| read\_rnd\_buffer\_size | 设置用于随机读的缓存大小 |

#### InnoDB特性

InnoDB存储引擎有三个主要特性分别是插入缓冲（change buffer）、两次写（double write）、自适应哈希索引（adaptive hash index）。

##### 插入缓冲

对于插入缓存，首先根据插入的普通索引页在缓冲池里查找，如果找到则直接将待插入页插入表中，如果没找到则先将待插入页放入缓冲池中，与之前的缓冲池进行合并操作，当有多个插入进行时，可以将其合并到一个操作中，从而提高插入的性能。

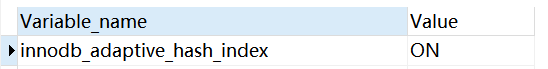
##### 两次写

当InnoDB缓冲池刷出新的“脏页”要写入磁盘文件时，由于两次写机制的存在，会首先将“脏页”写入两次写缓存中，随后两次写缓存以每次1MB的大小将数据写入两次写所在的磁盘表空间，并同时写入数据文件中。正是由于两次写的存在，保证了MySQL数据库中数据的安全性。

##### 自适应哈希索引

当使用InnoDB存储引擎时，会对索引的搜索进行监控，如果某个查询能使用哈希索引来优化查询索引速度，那么数据库系统将会自动完成这项操作，而不需要人为操作来完成。用户可以查看是否打开了该功能，SQL语句如下：

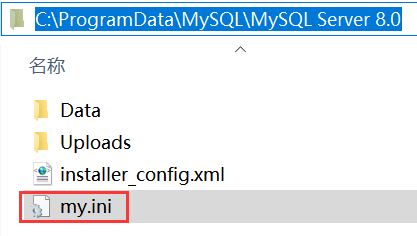
show variables like '%innodb\_adaptive\_hash\_index%';



### （五）修改MySQL的存储引擎

#### 修改my.ini文件

找到my.ini文件



通过文本编辑工具打开，可以进行修改



#### 建表时指定表的存储引擎

CREATE TABLE `emp` (

`id` int NOT NULL AUTO\_INCREMENT,

`NAME` varchar(10) CHARACTER SET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci NULL DEFAULT NULL,

`gender` char(1) CHARACTER SET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci NULL DEFAULT NULL,

`salary` double NULL DEFAULT NULL,

`join\_date` date NULL DEFAULT NULL,

`dept\_id` int NULL DEFAULT NULL,

PRIMARY KEY (`id`) USING BTREE

) **ENGINE = InnoDB** AUTO\_INCREMENT = 10 CHARACTER SET = utf8mb4 COLLATE = utf8mb4\_0900\_ai\_ci ROW\_FORMAT = Dynamic;