1 数据库简介

按照某种规则对数据进行收集和其他操作的一种软件

- 数据库管理系统

  DBMS存放数据的容器,完成数据的创建,读取,更新,删除等操作

- 数据库应用程序

  DBMS对用户不友好,所以我们使用应用程序作为媒介是的用户可以良好的体验到数据库的便利

2 MySQL细节

1. 必须要在语句后加;
2. 大小写不敏感
3. 基础语句

```
# show命令通常用来显示数据库内的信息
show databases;     # 显示所有数据库
show tables;     # 显示某个数据库下的所有的表
drop database database_name;     # 删除数据库
drop table table_name;     # 删除表
create database database_name;     # 创建数据库
use database_name;     # 使用某一个数据库
grant all privileges on database_name.* to name@localhost identified by 'xxxxxxx'     #
all privileges可以替换成具体的数据库操作,亦可以对数据库的某个表加权限,创建完之后需要退出使用 mysql -
u ... -p 登录就可以
select database();     # 查看现在使用的是什么数据库
des table_name;     # 显示建立后的表的结构

# 建表操作
create table table_name(
  column_name type [...] ,
  ...
)auto_increment=n,charset=utf8;     # charset选项设定表的编码方式
# 创建数据表的操作,制定域名和数据类型以及相应的完整性约束

# DML插入数据和查询数据
insert into table_name(1...) values(2...);     # 1是可选的,1如果不写的话2必须要使用建表的顺序填
写数据
select * from table_name where ... group by .... order by ... # 查询语句
delete from table_name where ...     # 删除数据

# 注释
/**/
```

4. 基础数据库
   - mysql : 存储运行的相关信息和用户等重要信息
   - information_schema :信息架构,管理着数据库的组成信息
5. MySQL基本的数据类型
   - 数值(unsigned属性设置无符号)
     - tinyint :-128~127
     - smallint : -32768~32767
     - mediumint : -8388608 ~ 8388607
     - int : 21亿

        - bigint
        - float : -E^38~ E^38
        - double : -E^308~ E^308
        - decimal : 精确计算的数据类型
    - 字符串类型 引号确认
        - char(size) : 固定长度字符串
        - varchar(size) : 可变长字符串
        - longtext : 2^64-1可变长字符串,存储网页在再合适不过
        - tinytext : 2^8-1可变长
        - text : 2^16-1
        - mediumtext : 2^24-1可变长
    - 日期类型 引号确认
        - datetime : 精确日期类型 xxxx-xx-xx xx:xx:xx
        - date : xxxx-xx-xx
    - 二进制类型
        - longblob : 2^32-1字节
        - tinyblob : 2^8-1
        - blob : 2^16-1
        - mediumblob : 2^24-1
6. MySQL基本列选项(表级完整性约束)
    - auto_increment :

      自增序列,建表的时候制定完,可以在插入的时候插入null数据,默认会开始自增长建立

      auto_increment一个表只能有一个并且必须是主键
        - 数据必须是int类型(tinyint,into,smallint,longint,mediumint)
        - 列后必须要加auto_incremen属性
        - 必须是唯一的,primary key或者unique限制

      删除之前的数据之后,自增长id会自动的调整
    - default 'xxx' : 设定默认值,没有默认值的NULL会被自动的转成default
    - index :定义索引
    - unique : 唯一,唯一可以是NULL
    - check : 约束
    - null / not null : 是否空,未指定默认是可以为NULL
    - primary key : 定义主键

```
create table people(
  name varchar(20) unique ,
  birthday datetime not null,
  age int,
  html longtext,
  id int auto_increment primary key)auto_increment=n,charset=utf8;     # 建表之后使用语句设定自增长的值

# 设定auto_increment的两种办法
1.就是上面的建表指定
2.alter table table_name auto_increment = x;
```

3 SQL

1. DML数据操作语言

   select / insert / update / delete

2. DDL数据定义语言

   create / drop / alter

3. DCL数据控制语言

   grant / revoke / ...

4. 比较运算符

   - =,>,<,>=,<=,<>,IS,NOT,LIKE,BETWEEN,IN

   LIKE模糊匹配

```
select * from people where name like "李%"    # 查找李性同学,%类似于*,_类似与.
```

5. NULL的where的特殊性

   匹配**NULL**的时候,我们不可以使用**[=],**只能使用**IS**操作符号

6. select order by排序语句 + limit语句

```
select * from people order by sex ASC , birth DESC;    # ASC升序,DESC降序,NULL最小值
# limit语句类似于Linux中的head语句
```

7. select group by语句 + 集函数

   - AVG
   - COUNT : count比较特殊,我们最好使用COUNT(*),否则使用具体的列count只会统计值不是NULL的数据
   - MAX
   - MIN
   - SUM

8. Select指定别名

```
select sex , count(mid) as total from people group by ...;
```

9. SQL运算符和数据库函数

   - +,-,*,DIV(DIV获取结果的整数部分),%
   - length()返回字节数 , concat(),trim(类似python的strip),rand(seed随机数产生,extract(type from dat)抽取某一个日期具体的时间描述,datediff(dat1,dat2)时间差,MD5(获取哈希值),date_add(date , interval +/-n type)时间加运算
   - extract和date_add运算的type类型常用:year , month,day,hour,minute,second
   - case语句

```
# case语句首先执行第一个条件符合的when语句
select nam,
    case sex
        when 条件表达式1 then ...
        when .....     then ...
        else ...
    end as sex    # 重命名
from people;
```

10. 多表连接查询操作

```
# 实例
# 内连接 / 外连接
select * from table1
inner join table2    # left outer join / right outer join
on table1.p = table2.p
where ...
group by ...
o`rder by...

针对for_examl的数据库
# 子查询
# 基本子查询
select COUNT(*) from `BOOK` where `Price` > (sele
                                ct avg(Price) from `BORROW` inner join `BOOK` on
                                `BOOK`.`Bno`= `BORROW`.`Bno`)
# IN / ONT IN 在查询中的应用
# 可以判断结果是否在子查询中

# EXIESTS / NOT EXIEST 的作用
# 用来判断子查询是否存在查询结果
```

```
insert into table_name(column1 , ...) values (...);    # 后者和前者一一对应
update table_name set column1 = ,column2 = ,... where name = '...';    # 按照set之后的更新
# 对这里的update有必要说一下,我们where指定的内容实际上是杜表中所有的数据进行遍历,对where后面的表达式返回为true的数据进行更改,默认没有where全都返回true
delete from table_name where ...;
select * from table_name where ... ;  # 查询操作
```

4 表的更新

1. 修改表的列结构

```
alter table table_name modify column_name type    # 修改列的定义，修改之后的猎德累心个必须要匹配否则会发生错误，建议在修改敏感数据之前先进行备份，如果数据中存在数据，最好不要尽心数据类型转换
alter table table_name add column_name type first / after   # 增加列,first,after用于将列追加到某一个特定的列之后
alter table table_name change old_column_name new_column_name type   # 修改列名和定义
alter table table_name drop column_name    # 删除对应的列,原数据一并删除
```

2. 赋值和删除表
    1. 复制表及其数据

```
create table new_table select * from old_table where ...    # 利用已经存在的表创建新表数据，利用where参数可以选择只满足条件的数据复制
# 但是需要注意，这样的方法有可能会造成数据类型的变化和INDEX的设定丢失
```

    2. 复制表的列构造不复制数据

```
create table new_table like old_table    # 该方法不复制数据，但是auto_increment和primary key属性会复制
```

    3. 向空表批量插入数据

```
insert into new_table(column_name / *) select * from old_table where ... limite
... # 利用where筛选挑选的数据
# 该方法的可以选定插入表数据或者列数据，但是直插入列数据的时候注意其他的数据都是NULL，这时候就要判断其他的数据是否可以为空，否则插入失败
```

4. 删除表

```
drop table table_name if exists table_name    # 表存在就删除，危险操作，无法恢复
```

5 事务处理和锁定

事务处理可以讲多个更新命令当成一个整体来执行，保证数据库的一致性和数据库性能

存储引擎

1. MySQL存在一个重要的特征就是存储引擎。MySQL Server整体分为两部分，外层进行客户端的连接和语法检查，内部(存储引擎)进行数据的输入输出和文件操作
2. MySQL的存储引擎众多，可随意选择
3. 引擎分类

存储引擎	特征
* MyISAM	高速，不支持事务处理
* InnoDB	支持行锁定和事务处理，速度稍慢
ISAM	几部不在标准安装，比较落后
MERGE	将多个MyISAM作为一个表处理的引擎
MEMORY,HEAP	内存存储数据结构
* Falcon	新的存储引擎，支持事务处理
ARCHIVE	数据压缩存储引擎
CSV	用CSV的文本形式存储数据，适用于跨平台

常用的存储引擎就是MyISAM,InnoDB(支持事务处理)，Falcon最为新秀也需要多了解

4. 设置存储引擎
    1. 查看存储引擎和表的详细信息

```
create table table_name (
  id int primary key,
  ...
);
# 记住必须是针对创建过的表才可以使用
show create table table_name;    # 常规打印，不排版
show create table table_name \G    # 规则打印

# Answer
***************************[ 1. row ]***************************
Table        | new_book
Create Table | CREATE TABLE `new_book` (
  `Bno` char(10) NOT NULL,
  `Bname` char(20) DEFAULT NULL,
  `Press` char(20) DEFAULT NULL,
  `Price` float DEFAULT NULL,
  PRIMARY KEY (`Bno`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

1 row in set
Time: 0.002s
```

2. 变更存储引擎

   我们之后可以对已经确定过的表的存储引擎进行变更

```
alter table table_name engine=new_engine_name    # 变更引擎,变更失败有可能是对应的存储引擎没有激活
```

事务处理

实际中的很多的操作都是组合到一起成为以这个整体作为一个事务做操作

1. 为什么需要事务

   以这个操作我们必须要保证数据库数据的一致性，所以必须引入事务的回滚和提交机制从而有效的保证数据的一致性，尽量避免不需要的错误

   事务是原子操作，只有回滚和提交两种结果(要么都成功要么都失败)

2. 事务处理

   命令 :BEGIN / COMMIT / ROLLBACK

```
BEGIN    # 声明事务开始
COMMIT    # 声明事务已经运行成功，COMMIT之前请先进行检查
ROLLBACK    # 事务失败开始回滚
# 可以写成sql脚本也可以在mycli命令行中执行
BEGIN;
SELECT * FROM table_name;    # 检查数据
delete from table_name;   # 删除数据
select * from table_name;    # 再次检查数据会否删除
select * from table_name;    # 查看是否删除数据
rollback    # 回滚到begin
select * from table_name;    # 检查是否回复删除数据
```

3. 自动提交机制

   1. AUTOCOMMIT = 1

在MySQL中，通常我们执行的命令都会被自动的提交是因为我们将全局变量AUTOCOMMIT设置为1，自动提交模式，所有的命令和语句都会自动的将数据内容写入硬盘中。

但是当我们对以InnoDB为存储引擎的表运行BEGIN命令之后只要没哟COMMIT都是可以ROLLBACK回顾我们的数据库状态的

2. AUTOCOMMIT = 0

用户可以讲强制提交的操作关闭，之后所有的操作只要没有COMMIT就可以ROLLBACK回顾恢复(只要这个表的存储引擎是支持事务的就可以,否则没有ROLLBAKC操作)

```
select @@autocommit;     # 检查当前是不是自动提交模式
set autocommit = 0;
set autocommit = 1;     # 设置自动提交环境
```

总结：

- 不支持事务操作的存储引擎是没有办法回滚恢复和(就不存在回滚和提交的能力)
- 支持的存储引擎如果是自动提交在BEGIN语句块中支持回滚和提交

  支持的存储引擎如果不是自动提交状态可以在任何时候回滚和提交

4. 部分回滚

上面的是没有确定开始标记的，我们都回滚的时间点是没有办法确认的

为此MySQL引入了保存点的概念，从而实现部分回滚的功能

```
# begin -> mysql command -> mysql command -> ... -> savepoint -> mysql command -> ... ->
rollback / rollbakc to some savepoint
begin ;    # 必须在begin语句块中才可以使用
savepoint point_name    # 定义保存点
rollback to savepoint point_name     #  回滚到保存点处，保存点之后的命令影响清空
```

5. 例外

在数据库的管理中并不是所有的命令操作都是可用事务回滚的，以下是几个例外，使用的时候要格外小心

```
# 删库，删表，改表操作是无法回复的
drop database database_name;
drop table table_name;
drop ;
alter table ;
```

并行事务处理

1. 锁
   - 共享锁

     拥有共享锁的进程可以读不可写，其他的进程可以在共享锁上继续加共享锁但是不能加排它锁，读取锁定
   - 排它锁

     拥有排它锁的进程可读可写，别的进程不可读不可写
2. 封锁粒度
   - 记录(行)
   - 表
   - 数据库

封锁粒度越小开销大并发好，相反开销小并发差，**MySQL**只支持行表粒度并且不支持锁定提升(大量行锁向表锁自动转化)

日志

- UNDO日志

  UNDO日志涉及到我们的回滚操作的内部实现，我们实际上对表操作之后会保存对表操作的指针，这个指针指向UNDO日志，回滚的时候会自动的用旧数据覆盖新的数据，赋值实现数据库一致性(给其他的用户保留变更前的数据)

- REDO日志

  事务处理日志，提供了回复数据库的手段

6 索引

内部使用B树作为索引加快我们的SELECT的操作的速度

1. 创建索引

```
create index index_name on table_name(employee_name)    # 建立索引
show index from table_name \G   # 查看改表上的所有的索引,一行一个索引，索引的类型默认是B树
# 索引的显示项目
Table          | READRES     #  索引的目标表
Non_unique     | 1    # 是否允许重复
Key_name       | no_index    # 索引的名字
Seq_in_index   | 1    # 索引列序号
Column_name    | Rno    # 索引的列名
Collation      | A    # 排序方式 ASC / AFTER
Cardinality    | 6    # 索引的非重复值的个数,这个由表内数据决定
Sub_part       | None
Packed         | None
Null           | YES    # 是否允许空
Index_type     | BTREE    # 索引算法
Comment        |     # 备注
Index_comment |
# MySQL建立的所以会对主键自动建立聚簇索引
```

   主键自动建立聚簇索引，一个表只有一个聚簇索引

2. 删除索引

```
drop index index_name on table_name    # 删除标的制定索引
```

3. 多列复合索引

```
create index index_name on table_name(column_name1,column_name2,...)    # 建立多个所以，复合索引
```

   复合索引的本质就是建立了多个相同名字的索引，索引针对的列不同，但是使用的时候是一块使用索引

4. 唯一索引

   唯一索引约束了指定的列的唯一属性

   - insert操作的时候插入该列重复数据会失败
   - 已经存在该列重复数据的时候也会失败
   - 唯一性符合索引只要求我们整体记录对应的符合字段不重复就可以

5. 评判索引，非常重要

```
explain select * from ...     # 生成表格记录我们的查询的状况
# 最重要的几个参数
# rows - 本次检索的搜索上的记录的数目，filtered过滤情况，记录有效的反应了我们的索引的工作情况，差距和没有索引的额时候越大越好，否则要考虑更换索引的列
```

6. 错误使用导致没有办法使用索引的情况
    - LIKE模糊搜索，后方一致或者部分一致搜索不会动用索引

      ```
      select * from table_name where column like "%w%"
      select * from table_name where column like "%w"
      # 索引使用我们的字段数据的开头作为索引使用的，一旦使用上述的模糊匹配会导致放弃索引，除非使用前方一致或者全部一致的匹配
      ```

    - IS NOT NULL / <>

      上述是无法利用索引的，因为匹配的记录不清晰
    - 使用函数或者运算的时候

      使用函数或者运算并没有办法直接得到我们的要索引的确切信息，放弃索引
    - 复合索引丢失第一个索引字段

      复合索引比较特殊，想要使用复合索引，第一个字段必须在WHERE语句中作为查询条(使用OR连接使用其他的复合字段也是不可以的，因为还是要面对对其他的字段的索引的单独检查依然放弃了第一个复合索引字段)

7 视图

1. 为用户提供良好的统一的数据接口但是内部使用分离的数据库规范化手段

   将数据从基本表中抽离模拟构建虚拟表呈献给用户
2. 视图的本质就是将SELECT语句的检索结果利用表的形式保留下来(SELECT语句的多表条件检索)，视图本身不含有任何的数据仅仅是从表中动态的抽取数据并将数据组织起来

   使得我们可以将多个物理表的数据通过视图整理起来
3. 视图的作用
    - 权限控制

      我们只公开表中特定的数据给用户从而实现权限控制(一般都可以給列做限制)
    - 简化复杂的SELECT搜索语句生成

      可以放弃编写大量的多表条件SELECT的语句,转而使用视图来实现我们的查询效果
    - 限制更新数据的范围

      对新的假表(视图)可以增加约束限制从而实现正确的插入使用数据
4. 语法

   并不是我们在创建视图的时候所有的SELECT语法都可以使用
    - 不可以涉及子查询

```
create or replace view view_name(column_name1 , ...) as select ... [with check option]
# 创建视图
drop view view_name     # 删除视图
show tables;     # 查看当前的数据库中的所有表，基本表和视图混合起来了，为了区别使用v_前缀是一个好的习惯
show fields from view_name;     # 显示表或者视图的域的详细信息
```

5. 注意点
    1. 视图不存在具体的数据表，一切数据全部是动态抽取，但是我们可以使用基本的对表操作针对视图是一样可以使用的
    2. 修改基本表的话，视图的数据同步修改，更显示视图仅仅只是数据的一种组织形式
6. SELECT

```
select * from view_name where ...    # 在视图中检索
```

7. INSERT / UPDATE / DELETE操作的限制
    - 视图的列中含有统计函数
    - 视图定义的时候SELECT语句使用了GROUP BY / HAVING,DISTINCT,UNION的语句
    - 使用了子查询
    - 跨越多表进行数据变更

```
# 会报错,
Can not modify more than one base table through a join view 'for_exam.view_name
# 在insert语句中有很多需要注意的地方
insert into view_name(column1 , column2) values(...)    # 加入数据到视图中
# 需要注意的是，视图在insert的时候必须要指定表的域(column1,column2,...)，并且按照上面的说法，这些域必须只能在单表中才可以实现插入数据
```

8. WITH CHECK OPTION

    使用WITH CHECK OPTION选项我们可以对视图进行约束限制
    - 在更新和插入不符合视图搜索条件的数据将会拒绝执行操作

    当然，就算没有这个选项，我们操作的非法数据(这对视图而言的非法)在视图检索的时候也是不可见的(关联的基本表肯定已经改了但是视图不显示罢了)

8 存储过程

1. 什么是存储过程

    因为SQL语言是一种非程序的语言，所有的操作都是通过命令实现的，但是有时候我们必须要让SQL拥有程序设计原因的特性

    所以有了存储过程这个概念

    1. 存储过程是存储在数据库中的一些列的SQL命令的集合
    2. 存储过程也可看作是狠毒SQL命令构成的程序段

    功能：

    1. 统一输入输出，整体作为一个大的命令执行用户输入，提供用户输出
    2. 提供参数机制，可宽展性良好
    3. 提高执行性能(数据库中的命令都是需要解析和编译的，这里存储过程是已经编译过的处理二进制程序，执行效果高)
    4. 减轻网络负担:因为我们便利的用户，用户只需要提供参数即可，具体的执行过程发生在服务器，减少网络通信的负担
    5. 数据库处理黑盒子化
2. 语法

```
create procedure sp_name(IN / OUT / INOUT var TYPE)
begin
...
end
#  创建数据库，为了之后好区分，建议使用sp_前缀区分
# 参数是存储过程和调用者之间的信息交换的通道，参数主要有IN,OUT,INOUT,输入参数，输出参数，输入输出参数
```

1. delimiter

```
delimiter //    # 定义分隔符号，因为MySQL的监视器中使用了;作为命令的分隔符号，但是在我们的存储过程中使用;会发生冲突，所以重定义全局的分隔符号
# 该分隔符号只是针对存储过程的，存储过程的begin / end语句内部还是使用我们的;作为分隔符号
# 记得还原为;
delimiter ;
# mysql中允许启动，但是mycli等其他的只是别标准sql语法，这个不是标准的sql语法，所以一般在mycli别的环境中都会报错，在mysql中写吧
```

2. 控制语句
   - 分支语句

     记住我们的**IF / CASE**只会匹配执行第一个满足的语句块

     ```
     # if语句
     if ... then
         SQL ...
     elseif ... then
         SQL ...
     ...     # maybe more [elseif ... then]
     else
         SQL ...
     end if
     # case语句
     case ...
         when ... then ...
         when ... then ...
         else ...
     end case
     ```

   - 循环语句

     ```
     while ... do
         SQL ...
     end while
     ```

3. 基本语句

   确定查找我们的所有的存储过程

   ```
   show procedure status \G    #  显示所有的存储过成
   show create procedure sp_name    # 详细罗列目标存储过程的信息
   ```

   删除存储过程

   ```
   drop procedure sp_name    # 删除存储过程
   ```

执行存储过程

```
call sp_name('lan%')    # 利用call命令滴啊用存储过程模糊匹配lan开头的人名
call sp_name(null)    # 有时候我们的存储过程必须要负空串和NULL进行考虑处理
```

4. 创建存储过程的要点 - 参数
    1. 变量

       本地变量，又名局部变量，只能在存储过程中使用的变量，存储在粗出过程中的临时值

       ```
       # 只能在存储过程中使用
       declare var_name type default n    # 在存储过程中定义我们的变量并设定默认值是n
       set var_name = ...    # 设定变量的值
       ```

    2. 写入变量

       ```
       # 在存储过程中，将select搜索的数据写入变量，也可以使用set，但是这里主要是为了调用
       found_rows()函数的方便性
       select found_rows() into var    # 将select检索到的记录的数目写入我们的变量
       # 其中的into是追加的语法之外，select...语法不变
       ```

       SELECT的FOUNT_ROWS函数，可以返回上一条SELECT`语句的执行的返回的记录的数目

       ```
       call sp_name('' , @num)    # @num声明out变量并带入存储过程以返回
       select @num    # 读取变量的值，这个是最重要的手段
       ```

    3. 实例 - 阶乘的存储过程计算

       ```
       delimter //
       create procedure sp_factorial(in p int , out r int)
       begin
           set r = 1;
           while p > 1 do
               set r = r * p;
               set p = p - 1;
           end while;
       end
       //
       delimiter ;
       ```

9 函数和触发器

存储函数

除了存储过程之外，**MySQL**还支持用户自定义存储函数

除了MySQL中自定义的多功能函数之外，自定函数也是一种单功能处理机制(第3节SQL中已经介绍了一批数据库函数)

1. 定义存储函数

```
# 阶乘函数
delimiter //
create function f_factorial(p int) returns int
begin
    declare result int default 1;
    while result > 1 do
        set result = result * p;
        set p = p - 1
    end while;
    return result;
end
//
delimiter ;
# 存储函数和存储过程很像，但是不存在输出参数，只需要用returns指定函数的返回值，在最后需要return一下，
其实和标准的程序设计语言是一个套路
```

2. 查看存储函数

```
show function status \G    # 检查当下所有的存储函数
show create function function_name    # 检查对应的存储函数的性质
```

3. 调用存储函数

```
# 和存储过程不同，存储函数可以直接调用
select f_factorial(5);
#  结果
+----------------+
| f_factorial(5) |
+----------------+
|            120 |
+----------------+
```

触发器

1. 出发器是针对数据库的特殊的存储过程，他的触发条件并不是调用而是事件触发
2. 触发器本质上实现了对表操作的自动化管理
3. 触发器基本语法

```
delimiter //
create trigger trigger_name before / after [insert , update , delete , ...]
on table_name for each row [ for each statement ]    #
begin
    SQL ...
end
//
delimiter ;
```

   - 并不是只有`INSERT / UPDATE / DELETE`操作才会进行出发启动，本质上类似的数据更新操作都会引起同样的触发条件
   - `BEFORE / AFTER`触发条件针对的是我们针对事件的动作执行的先后
   - `FOR EACH ROW`表示触发是一行为单位统计的，`FOR EACH STATEMENT`是以语句块为单位进行统计的

4. 确认触发器

```
show triggers \G    # 显示当前数据库的所有的触发器
```

5. 删除触发器

```
drop trigger trigger_name;    # 删除触发器
```

游标

1. 游标就是对SELECT的查询集合结果一个一个记录取出处理的方式
2. 指针：指针确定当前的记录的细腻系，内存中保存当前记录的地址位置
3. 使用方式

```
#  声明游标
declare cur_name cursor for select ...    # 声明针对select结果集合的游标
# 打开游标
# 定义的游标其实并没有实际执行select语句，需要打开游标才可以开始操作
open cur_name
# 从游标指针中取出记录
# 如果因为异常(比如取空)导致fetch失败我们的比那辆var_name是得不到正确的赋值的
fetch cur_name into var_name    # 将记录数据从游标中取出放入变量中
fetch cur_name into var1,var2,var3,...    # 记录数据批量取出

# 游标取出数据的中断处理，定义例外(异常)处理
# 针对我们的游标取尽元素的时候，会抛出NOT FOUND异常
declare [type] handler for [NOT FOUND] SQL ...     #  SQL语句针对异常的处理的方式
# type处理种类决定我们完成异常之后的处理之后如何行动，有两种处理(EXIT / CONTINUE)中断跳出存储过程和继续执行两种情况

# 关闭游标
close cur_name
```

10 文件处理

1. 导入数据从文件

```
load data local infile 'path' into table table_name [type];    # 将外部CSV数据导入到数据库
# path使用绝对路劲或者相对路径
# 这里非常需要注明，本人亲测 Ubuntu16.04 MySQL 5.7.x 必须使用local关键字才可以成功的到入数据
# type 选项,选项可以复用
# fields terminated by ';'      指定CSV文件使用逗号作为分隔符号,默认是\t
# lines terminated by '\n'    指定换行符号，默认是\n
# ignore n lines    跳过多少行开始读数据，默认是0
```

2. 导出数据到文件

```
select * into outfile 'path' fields terminated by ',' from table_name;     # 将数据从table_name中导出数据到path制定的护具中
# 对与MySQL5.7.x以上的版本中，我们最新导入的新特性 secure_file_priv 会自动的禁止我们的MySQL对文件的导出导入做操作，上面的导入我们已经使用了LOCAL进行解决方案
# 我们的解决方案就是
#     1. select @@global.secure_file_priv     查看对应的该参数的全局变量的默认的地址
#     2. 切换到管理员用户 , 修改MySQLd的my.cnf(/etc/mysql/my.cnf)文件最后的[mysqld]配置，加入 secure-file-priv = 'path' , path注意这里使用的是我们制定的存储导入导出的文件路径
#     3. 进行完上述的操作之后我们还会发现，使用的导出导入操作会出现mysql用户没有读写权限的问题，这里我们需要对apparmor进行修改，加上对上面制定的path的读写权限
         vim /etc/apparmor.d/usr.sbin.mysqld 在其中最后加入
         "path/* rw,"     如果使用/home/lantian/*那么之后只要涉及到输出文件必须使用/home/lantian全程，使用~/也会错误
#     4. /etc/init.d/apparmor reload     对mysql重新加载配置
#     5. service mysql restart      对mysql服务进行重启
# 使用完上述的操作命令之后我们就可以争取的执行我们的 select into outfile 操作实现数据库的导出备份
```

3. 执行SQL文件命令

   我们将正确的SQL命令写入文件中，一次性执行节约效率

```
source path     # 执行外部的文件中的SQL命令，source并不是SQL命令。是引导如的MySQL处理方式
```

4. CLI执行MySQL命令

```
mysql -u root -p -e "source path ; ..."     # 在CLI中使用数据库文件执行
```

5. 保存MySQL的处理结果记录备份文件
   - 输出重定向(不建议使用，操作见过不显示，有操作危险)

```
mysql -u root -p > log     # 输出重定向导入
```

   - tee和notee命令

```
tee log     # log的path必须在我们的/etc/apparmor.d/usr.sbin.mysqld中写入mysql才有写入的权限
notee     # 停止写入，中间的所有的数据都会写入文件中备份一共存有备份检查
```

11 数据库备份

1. 数据库备份为了构建损坏的数据库的情况
2. 数据库的转储文件保存的是我们的数据库的创建所需要的SQL语句集合，这是本质，转储的输出就是数据库的本身
3. 应用
   - mysqldump

```
mysqldump -u root -p database_name > save.sql
# 查看我们的save.sql我们会发现其实就是我们的SQL构建语句
# 设置可以看出使用--做单行注释，/**/是多行注释
# 如果转储失败可以使用参数
mysqldump -u root -p database_name --default-character-set=utf8 > save.sql
```

   - 回复数据库

回复数据库首先我们需要手动的创建数据库在导入外部的转储.sql文件

```
mysql -u root -p -e 'create database database_name'    # 创建数据库
mysql -u root -p database_name < save.sql    # 将转储的sql文件存储转储copy到我们的
database_name数据库中
```