第1章 MySQL 概述

1.1 数据库基础知识

1.1.1 数据管理技术的发展

数据管理是指对数据进行分类,组织,存储,查询,维护等.数据管理大概经历三个阶段: 人工管理阶段; 文件系统阶段; 数据库系统阶段.

1.1.2 数据库的相关概念

数据库(Database,DB)保存在计算机存储设备上,按照一定规则组织起来,可以被用户和应用程序共享的数据的集合. 就是数据的仓库

数据库管理系统(Database Management System, DBMS)是用来管理数据库的的软件,可以建立,使用和维护数据库,对数据库管理与控制以保证数据库的安全性与完整性.常用的数据管理系统有:MySQL, Oracle, DB2 , Sybase, SQL Server...

数据库系统(DatabaseSystem, DBS)包括安装 DBMS 的硬件, DBMS, 数据库和数据库管理员.

1.1.3 SQL 语言

SQL(Structured Query Language,结构化查询语言)对数据库进行查询与操作的语言. SQL 不是某个特定数据库专用商的语言,几乎所有的数据库管理系统都支持 SQL

SQL 语言包括 4 部分:

数据定义语言 DDL, 包括 CREATE,ALTER,DROP 等语句

数据操作语言 DML, 包括 INSERT,UPDATE,DELETE 等语句

数据查询语言 DQL,包括 SELECT 语句

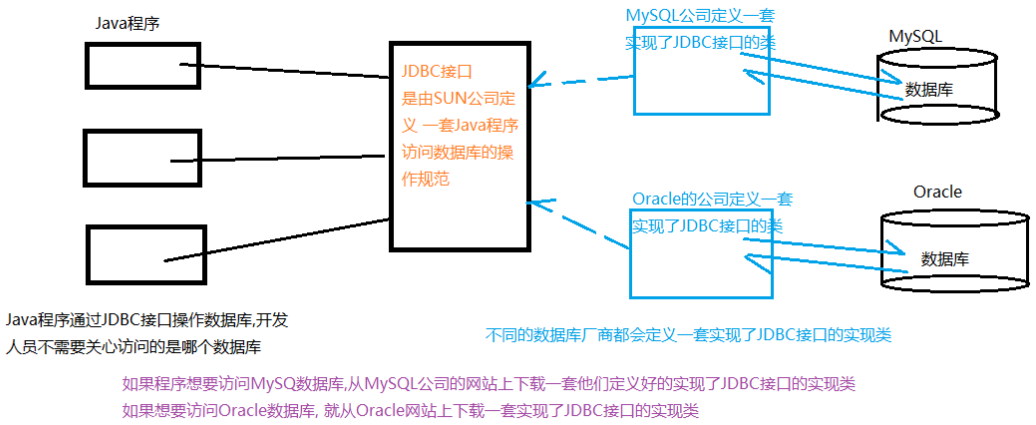
数据控制语言 DCL, GRANT,REVOKE,COMMIT,ROLLBACK 等语句

1.1.4 数据库访问接口

不同的计算机语言有不同的数据库访问接口,程序通过 这些接口执行 SQL 语句.主要的数据库访问接口有:

ODBC(Open Database Connectivity)开放数据库互连技术为访问不同的 SQL 数据库提供了一个共同的接口.ODBC 使用 SQL 作为访问数据库的标准,通过 ODBC 可以使程序访问不同的数据库

JDBC(Java Database Connectivity)Java 数据库互连用于 Java 程序连接数据库



1.2 什么是 MySQL

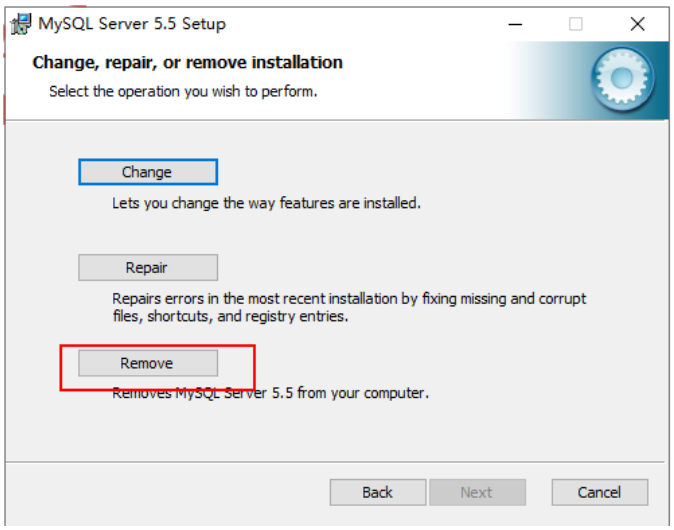
MySQL 原来是瑞典 MySQL AB 公司的，在 2008 年被 SUN 公司收购, 2009 年 SUN 被 Oracle 收购

MySQL 是一个小型的关系数据库管理系统. 规模小,速度快,成本低

第2章 MySQL 的安装与配置

2.1 卸载 MySQL

1) 双击安装文件,选择 Remove 移除



删除安装路径文件夹

C:\Program Files\mysql

删除在 c:/ProgramData 目录中的 mysqL 程序数据

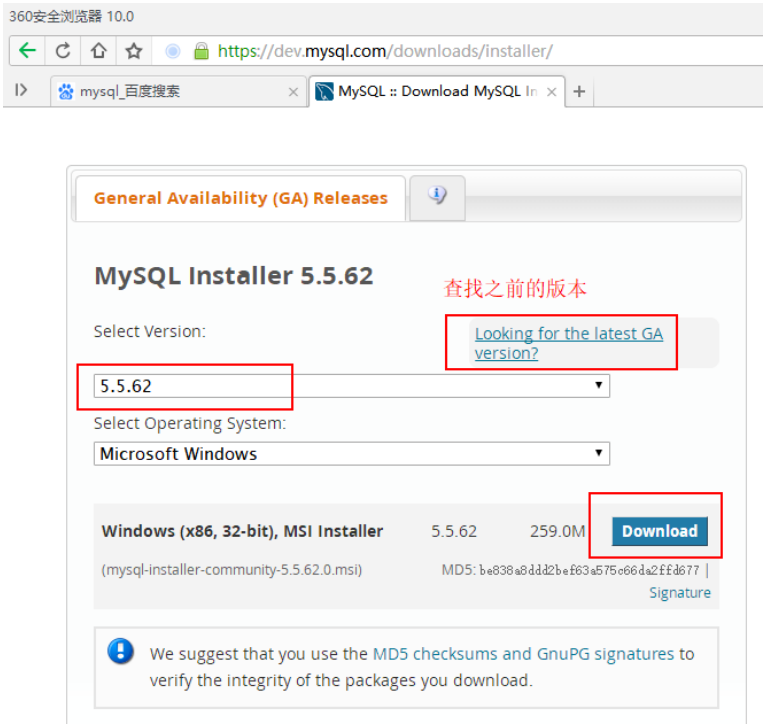
清除注册表,运行输入 regedit 打开注册表,删除计算机\HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001 \Services \EventLog \Application\MySQL 目录

2.2 MySQL 的安装与配置

1) 下载

MySQL 分为社区版(Community)与企业版(Enterprise). 社区版可以自由下载并且免费,适用于多数用户. 企业收收费,功能多.

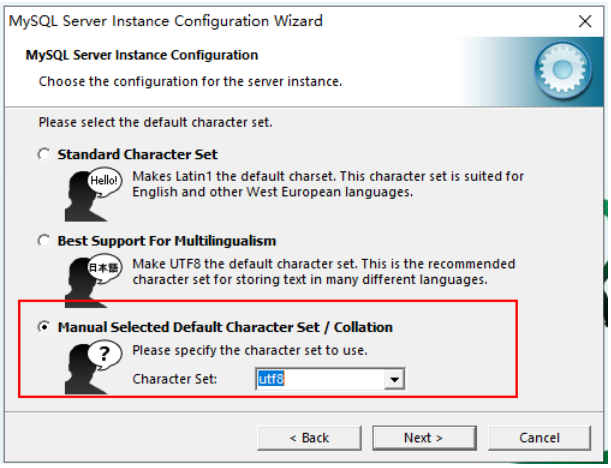
MySQL 版本有 GA(General Availabity)表示广泛使用的版本, RC(Release Candidate)候选版; Alpha 内测版与 Beta 公测版.



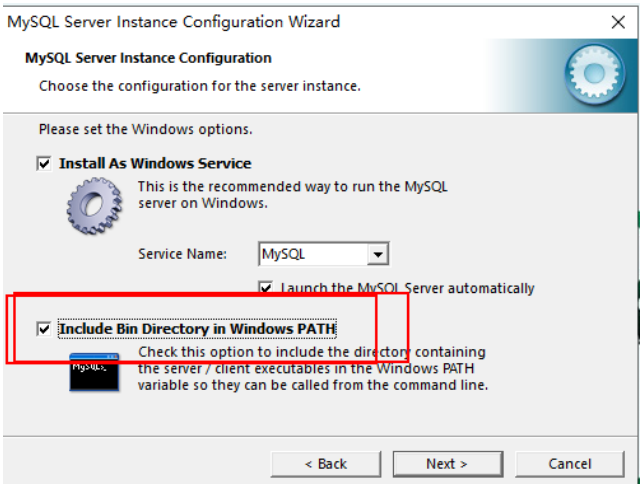
2)安装

采用默认值安装

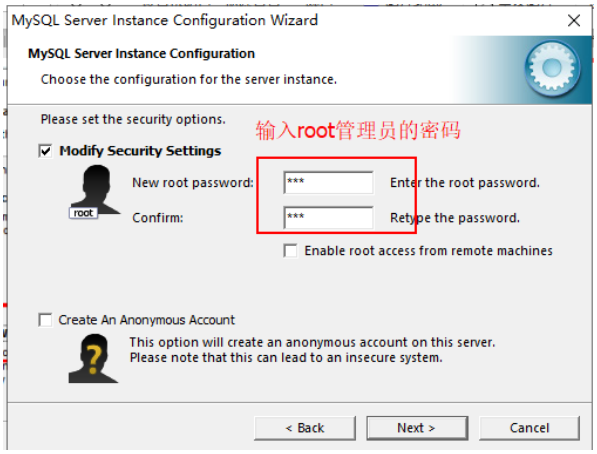
在配置默认的语言时,选择 utf8



把 MySQL 的 bin 目录添加到 path 环境变量中



安装 MySQL 后,会有一个 root 管理员帐号,输入 root 用户的密码



安装完成后,打开 C:\Program Files\MySQL\MySQL Server 5.5 目录,包括:

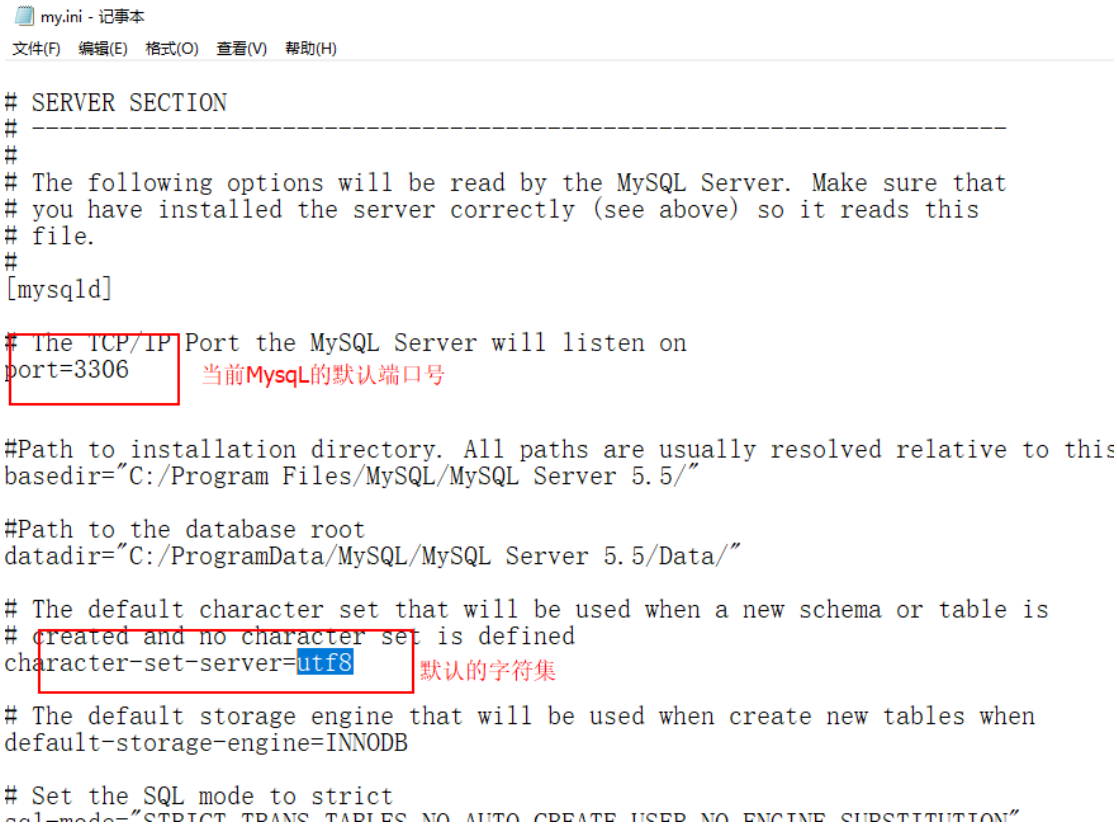
bin 就是 MySQL 的一些工具命令

include 存储头文件

lib 存储库文件

share 存放字符集,语言等信息

目录中的 my.ini 是当前 MySQL 数据库的配置文件,可以修改端口号,字符集等信息



2.3 启动与登录 MySQL

通过服务管理器启动

以管理员身份打开命令窗口,输入 net start mysql

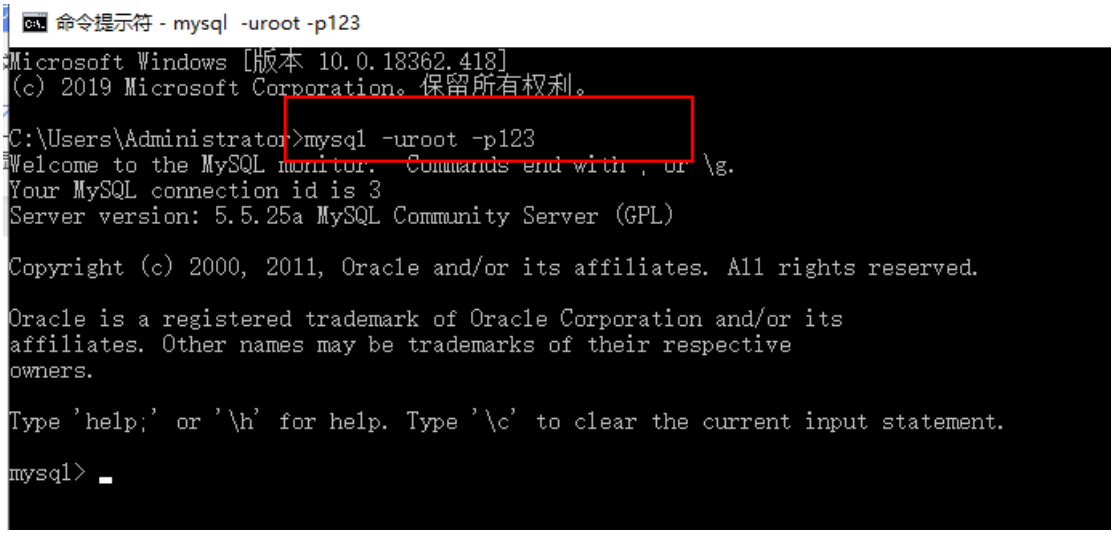
停止 就是 net stop mysql

登录 mysql.在命令窗口中输入:

mysql -uroot -p123

-u 参数指定用户名为 root

-p 参数指定密码为 123



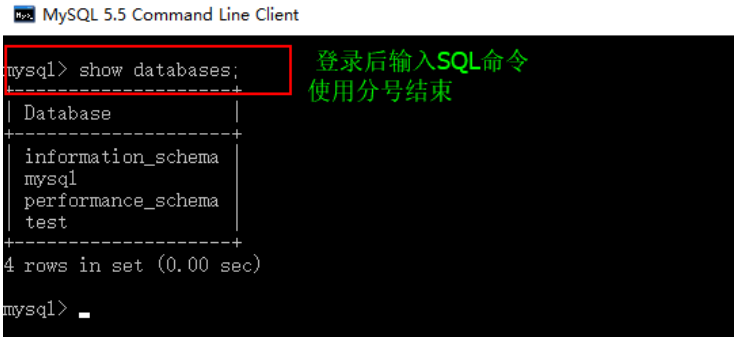
退出输入 exit, \q, Ctrl+C

第3章 数据库的操作

3.1 数据库与数据库对象

数据库就是数据库对象的容器,所谓的数据库对象包含表,视图,存储过程,函数等.

数据库分为系统数据库与用户数据库两大类.安装完 MySQL 后,登录后,输入 show databases; 显示当前的所有数据库



infomation_schema 数据库主要存储系统中的数据库对象信息，包括用户表信息，列信息，权限信息。

mysql 主要存储系统的用户权限信息

performance_schema 存储数据库服务器的性能参数

test 自动创建的测试数据库

用户可以根据自己的需求创建自己的数据库，即用户数据库

3.2 创建数据库

CREATE DATABASE 数据库名；

```
mysql> CREATE DATABASE test_db;        创建数据库
Query OK, 1 row affected (0.00 sec)

mysql> SHOW CREATE DATABASE test_db;      查看数据库的定义
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| test_db  | CREATE DATABASE `test_db` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+----------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SHOW CREATE DATABASE test_db\G;
*************************** 1. row ***************************
       Database: test_db
Create Database: CREATE DATABASE `test_db` /*!40100 DEFAULT CHARACTER SET utf8 */
1 row in set (0.00 sec)

ERROR:
No query specified

mysql>
```

3.3 查看数据库

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
| test_db            |
+--------------------+
5 rows in set (0.00 sec)
```

3.4 删除数据库

删除数据库会把所有的数据一起删除，是不可恢复的，在删除时一定要非常谨慎。

```
mysql> DROP DATABASE test_db;
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

删除数据库

3.5数据库存储引擎

存储引擎是 MySQL 数据库管理系统特有的一个特征。存储引擎指定了表的类型，即如何存储和检索数据，是否支持事务等。

不需要在整个数据库服务器中都使用同一个存储引擎，必要时，可以对每一个表使用不同的存储引擎。

```
mysql> SHOW ENGINES\G;
*************************** 1. row ***************************
```

```
*************************** 6. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 9. row ***************************
```

3.5.1 InnoDB 存储引擎

InnoDB 存储引擎支持事务，支持行级锁和外键， MySQL 默认的存储引擎

3.5.2 MyISAM 存储引擎

MyISAM(Indexed Sequential Access Method)拥有较高的插入，查询速度， 不支持事务

3.5.3 Memory 存储引擎

Memory 存储引擎会把表中的数据存储到内存中。

3.5.4 如何选择存储引擎

一般情况下采用默认的 InnoDB 存储引擎即可，支持事务，外键。 如果数据表主要用来插入和查询记录，可以选择 MyISAM; 如果只是临时存储数据，并且数据量不大可以选择 Memory 存储引擎。

第4章 数据类型

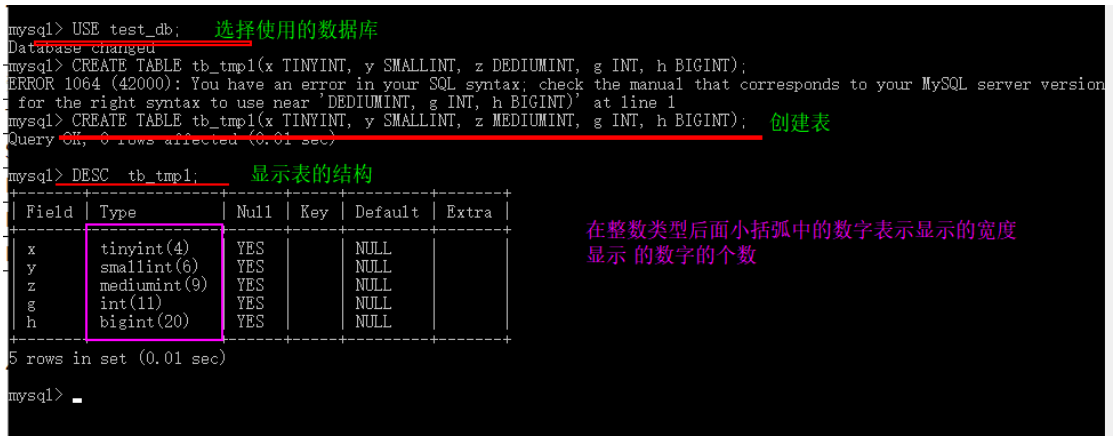
数据库中的数据存储在数据表中，数据表就是由行与列组成的二维表，列又称为字段，每一列都需要指定数据类型，数据类型决定插入数据的内容。数据类型包括 数值类型，日期、时间类型和字符串类型等

创建表的简单语法：

CREATE TABLE 表名（ 字段名 字段类型， 字段名 字段类型。。。）；

4.1 整数类型

数据类型名称	存储需求	有符号的取值范围	无符号取值范围
TINYINT	1 字节	-128~127	0~255
SMALLINT	2 字节	-32768~32767	0~65535
MEDIUMINT	3 字节		
INT	4 字节	-2147483648~2147483647	0~4294967295
BIGINT	8 字节		



4.2 小数类型

数据类型名称	说明	存储需求
FLOAT	单精度浮点小数,有 7 位有效数字	4 字节
DOUBLE	双精度浮点小数,有 15 位有效数字	8 字节
DECIMAL(M,D)	定点小数,M 表示精度,D 是小数的位数. DECIMAL 是以字符串形式存储的,主要用于货币计算	M+2 个字节

注意:在 MySQL 中,不管是浮点小数还是定点小数,如果数据超出了数据类型的精度范围都采用四舍五入进行处理

```
mysql> CREATE TABLE tb_tmp2( x FLOAT, y DOUBLE, z DECIMAL(20,4) );      创建表
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO tb_tmp2 VALUES(123456789, 12345678901234567890123456789,12345678901234567890123456789);
ERROR 1264 (22003): Out of range value for column 'z' at row 1
mysql> INSERT INTO tb_tmp2 VALUES(123456789, 12345678901234567890123456789,1234567890123456.7890);
Query OK, 1 row affected (0.01 sec)          向表中插入数据

mysql> SELECT * FROM tb_tmp2;          查询表中所有的数据
+-----------+----------------------+----------------------+
| x         | y                    | z                    |
+-----------+----------------------+----------------------+
| 123457000 | 1.2345678901234568e29 | 1234567890123456.7890 |
+-----------+----------------------+----------------------+
1 row in set (0.00 sec)
```

4.3 日期与时间类型

数据类型名称	格式	范围	存储需求
YEAR	YYYY	1901~2155	1 字节
TIME	HH:MM:SS	-838:59:59~838:59:59	3 字节
DATE	YYYY-MM-DD	1000-01-01~9999-12-31	3 字节
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 ~ 9999-12-31 23:59:59	8 字节
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:00 ~ 2038	4 字节

4.3.1 YEAR

YEAR 表示年,经常以四位字符串的形式表示,如:'2019', 也可以是数字,如 2019

有时也可以使用两位字符串表示年,范围在'00'~'99'. 注意:'00'~'69'范围的值会被转换为 2000~2069 年, '70'~'99'范围的值会被转换为 1970~1999 年. '0'被转换为 2000 年

有时也会使用两位数字表示年, 1~69 范围的值表示 2001~2069 年, 70~99 范围内的值被转换为 1970~1999 年, 0 被转换为 0000 年

```
mysql> CREATE TABLE tb_tmp3( y YEAR);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO tb_tmp3
    -> VALUES(2019),
    -> ('2019'),
    -> ('66'),
    -> (66),
    -> ('88'),
    -> (88),
    -> ('00');
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

向表中插入多条记录
insert into 表名
values (记录1), (记录2), (记录3).....(记录n);

```
mysql> SELECT * FROM tb_tmp3;
+------+
| y    |
+------+
| 2019 |
| 2019 |
| 2066 |
| 2066 |
| 1988 |
| 1988 |
| 2000 |
+------+
7 rows in set (0.00 sec)
```

4.3.2 TIME

TIME 表示时间, HH:MM:SS 格式中的 HH 小时数不是一天的时间 ,可能是一个时间间隔.

```
mysql> CREATE TABLE tb_tmp4( t TIME);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO tb_tmp4
    -> VALUES('15:34:08'),
    -> ('20:20'),
    -> ('10 11:12'),
    -> ('10'),
    -> (101112),
    -> ('0'),
    -> (current_time),
    -> (now());
Query OK, 8 rows affected (0.01 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

当前时间

```
mysql> SELECT * FROM tb_tmp4;
+-----------+
| t         |
+-----------+
| 15:34:08  |
| 20:20:00  |
| 251:12:00 |
| 00:00:10  |
| 10:11:12  |
| 00:00:00  |
| 15:35:49  |
| 15:35:49  |
+-----------+
8 rows in set (0.00 sec)
```

4.3.3 DATE

DATE 日期,格式是 YYYY-MM-DD

```
mysql> CREATE TABLE tb_tmp5( d DATE);
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO tb_tmp5
    -> VALUES('2019-11-03'),
    -> ('20191103'),
    -> ('19-11-03'),
    -> ('191101'),
    -> ('89-11-03'),
    -> (current_date),
    -> (now());
Query OK, 7 rows affected, 1 warning (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 1
```

```
mysql> SELECT * FROM tb_tmp5;
+------------+
| d          |
+------------+
| 2019-11-03 |
| 2019-11-03 |
| 2019-11-03 |
| 2019-11-01 |
| 1989-11-03 |
| 2019-11-03 |
| 2019-11-03 |
+------------+
7 rows in set (0.00 sec)
```

4.3.4 DATETIME

既有日期又有时间,占 8 字节

```
mysql> CREATE TABLE tb_tmp6( dt DATETIME);
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO tb_tmp6
    -> VALUES('2019-11-03 15:45:36'),
    -> ('20191103154536'),
    -> ('19-11-03 15:45:36'),
    -> ('88-11-03 15:45:36'),
    -> (now());
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM tb_tmp6;
+---------------------+
| dt                  |
+---------------------+
| 2019-11-03 15:45:36 |
| 2019-11-03 15:45:36 |
| 2019-11-03 15:45:36 |
| 1988-11-03 15:45:36 |
| 2019-11-03 15:46:58 |
+---------------------+
5 rows in set (0.00 sec)
```

4.3.5 TIMESTAMP

与 DATETIME 一样,也是用来表示日期的,占 4 字节

与 DATETIME 区别取值范围小:1970~2037; 另外一个区别是 TIMESTAMP 以世界协调时的格式存储的,如果当前时间是北京时间的 15:52:10 秒,如果是以 TIMESTAMP 类型保存时,实际在数据表存储的时间是 7:52:10 秒,在检索时再把时间转换为当前时区, 即以 TIMESTAMP 存储一个时间,在不同时区检索出来的时间值是不同的

4.4 字符串类型

数据类型名称	说明	存储需求
CHAR(M)	固定长度的字符串	M 字节
VARCHAR(M)	变长的字符串	L+1,L 是字符的数量,L<=M
TINYTEXT	小的字符串	L+1 字节, $L < 2^8$
TEXT	字符串	L+2 字节, $L<2^{16}$
MEDIUMTEXT	字符串	L+3 字节, $L<2^{24}$
LONGTEXT	字符串	L+4 字节, $L<2^{32}$
ENUM	枚举	
SET	集合	

4.4.1 CHAR 与 VARCHAR

CHAR(M)为固定长度的字符串,M 取值 0~255 个字符, VARCHAR(M)是变长字符串,实际长度由字符数决定,需要多一个字节存储字符的数量

字符串	CHAR(4)	存储需求	VARCHAR(4)	存储需求
‘’	‘ ‘	4 字节	‘’	1 字节
‘ab’	‘ab ‘	4 字节	‘ab’	3 字节
‘abc’	‘abc ‘	4 字节	‘abc’	4 字节

‘abcd’	‘abcd’	4 字节	‘abcd’	5 字节
‘abcdef’	‘abcd’	4 字节	‘abcd’	5 字节

4.4.2 TEXT

TEXT 用来存储字符串,如文章内容,评论内容.
TINYTEXT 可以存储 255 个字符, TEXT 存储 65535 个字符,LONGTEXT 可以存储 4GB 个字符

4.4.3 ENUM

定义字段时,可以这样:
字段名 ENUM(‘值 1’, ‘值 2’, ‘值 3’,….’值 n’)
字段只能取值 ENUM 列表中的某个值

```
1  CREATE TABLE tb_tmp7(  data ENUM('EXCELLENT','GOOD','BAD') );
2
3  INSERT INTO tb_tmp7 VALUES('EXCELLENT'),('GOOD'),(1);
4
5  SELECT * FROM tb_tmp7;
6
```

信息 | 结果 1 | 剖析 | 状态

data
EXCELLENT
GOOD
EXCELLENT

4.4.4 SET

```
1  CREATE TABLE tb_tmp8( data SET('a','b','c','d'));
2
3  INSERT INTO tb_tmp8 VALUES('a'), ('b,d,b'),('d,c,a');
4
5  SELECT * FROM tb_tmp8;
6
7  INSERT INTO tb_tmp8 VALUES('a,e,f');
```

自动去掉重复的数据值
会自动调整数值的顺序

信息 | 结果 1 | 剖析 | 状态

data
a
b,d
a,c,d

4.4.5 二进制类型

MySQL 可以使用二进制类型存储非文本数据,如 BIT(M), BINARY(M),BLOB 等.

虽然使用 BLOB 或者 TEXT 可以存储大容量的数据,对这些大数据的处理会降低数据库的性能.一般情况下,只存储文件的路径,而不是文件的内容.

4.5 如何选择数据类型

浮点数与定点数. 在长度一定的情况下,浮点小数可以表示更大的数据范围,但是有误差,对于精度要求较高时采用定点小数 DECIMAL.

日期时间类型,如果表示年使用 YEAR,表示时间使用 TIME,表示日期使用 DATE,表示日期与时间使用 TIMESTAMP 或 DATETIME, DATETIME 表示范围大, TIMESTAMP 实际存储的是 UTC 世界协调时. 如果使用 TIMESTAMP 定义字段类型时,在插入数据时如果没有这个列值,系统会把当前时间插入.

CHAR 与 VARCHAR,CHAR 是固定长度的,处理速度比 VARCHAR 快,但是浪费空间. 但是在 MySQL 中,存储引擎会影响 CHAR 与 VARCHAR,在 InnoDB 存储引擎中,不区分固定长度与可变长度,使用 CHAR 不一定比 VARCHAR 快,使用 VARCHAR 可以节省存储空间; 在 MyISAM 存储引擎中,使用 CHAR 类型查询速度比 VARCHAR 快很多.

第5章 数据表的操作

5.1 创建数据表

```
CREATE TABLE  数据表名(
    字段名  数据类型  [列级约束],
    字段名  数据类型  [列级约束],
    [表级约束]
)
```

如创建员工表:

```
CREATE TABLE tb_emp1(
    id    INT,
    name VARCHAR(30),
    deptid INT,
    salary FLOAT
);
```

使用 DESCRIBE/DESC 查看表的结构

```
10  DESCRIBE tb_emp1;
11
```

信息 | 结果 1 | 剖析 | 状态

Field	Type	Null	Key	Default	Extra
id	int(11)	YES		(Null)	
name	varchar(30)	YES		(Null)	
deptid	int(11)	YES		(Null)	
salary	float	YES		(Null)	

FIELD 就是列名，TYPE 列的数据类型，NULL 是否允许为空，Key 是否索引列，Default 默认值,Extra 表示可以获得与列有关的附加信息

SHOW CREATE TABLE 查看表的定义语句

```
8  SHOW CREATE TABLE tb_emp1;
```

信息 | 结果 1 | 剖析 | 状态

Table	Create Table
tb_emp1	CREATE TABLE `tb_emp1` (`id` int(11) DEFAULT NULL, `name` varchar(30) DEFAULT NULL, `deptid` int(11) DEFAULT NULL, `salary` fl

5.2 创建表约束

数据的完整性是指数据的准确性与一致性，完整性检查就是检查数据的准确性与一致性. 在数据库中,提供了一套约束机制确保数据的完整性,这些约束包括:

主键约束,保证记录的唯一性

外键约束,约束的字段为表的外键,外键一般是其他表中是主键

非空约束,约束字段的值不为 NULL

唯一约束,约束字段的值是唯一 的

默认值约束,用来设置字段的默认值

字段值的自增增长约束

在有的数据库管理系统中还有 CHECK 检查约束,但是 MySQL 不支持

这些约束可以简单的分为单列约束,即每个约束只约束一列数据；多列约束,即每个约束可以约束多列数据.

5.2.1 使用主键约束

主键,也称为主码，要求主键列的数据必须是唯一的,并且不能为 NULL. 通过主键约束可以唯一的标识一条记录. 也可以结合外键定义不同数据表之间的关系，通过主键约束可以提高数据的查询速度. 主键与记录是一对一关系，主键可以为单字段主键,也可以为多字段联合主键. 在实际开发中,经常单独创建一个自动增长列作为主键.

1)在定义列的同时指定主键

```
CREATE TABLE tb_emp2(
    id INT PRIMARY KEY,     -- 在定义列时,在数据类型后面使用 PRIMARY KEY 表示主键
    name VARCHAR(30),
    deptid INT,
    salary FLOAT
);
```

2)在定义完所有的字段后指定主键

```
CREATE TABLE tb_emp3(
    id INT ,
    name VARCHAR(30),
    deptid INT,
    salary FLOAT,
    PRIMARY KEY( id )
);
```

```
INSERT INTO tb_emp3( id, name, deptid, salary) VALUES (1001, 'lisi', 9001,7865);
INSERT INTO tb_emp3( id, name, deptid, salary) VALUES (1001, 'wangwu', 9002,1235);
SELECT * FROM tb_emp3;
```

在id列上创建主键,则id字段的值不能重复,也不能为NULL

插入记录时,如果id重复了,插入失败

信息 状态

```
INSERT INTO tb_emp3( id, name, deptid, salary) VALUES (1001, 'wangwu', 9002,1235)
> 1062 - Duplicate entry '1001' for key 'PRIMARY'
> 时间: 0s
```

3)在多字段上创建主键

也叫联合主键,是定义完所有字段后指定主键

```
CREATE TABLE tb_emp4(
    id INT ,
    name VARCHAR(30),
    deptid INT,
    salary FLOAT,
    PRIMARY KEY( id , name)
);
```

```
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1001, 'lisi', 9001,7865);
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1001, 'wangwu', 9002,1235);
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1002, 'wangwu', 9002,1235);
SELECT * FROM tb_emp4;
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1001	lisi	9001	7865
1001	wangwu	9002	1235
1002	wangwu	9002	1235

```
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1002, 'wangwu', 9002,1235);
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1002, 'wangwu', 9003,6547);
```

信息 状态

```
INSERT INTO tb_emp4( id, name, deptid, salary) VALUES (1002, 'wangwu', 9003,6547)
> 1062 - Duplicate entry '1002-wangwu' for key 'PRIMARY'
> 时间: 0.001s
```

5.2.2 查看约束

```
42
43  SELECT * FROM information_schema.TABLE_CONSTRAINTS
44
```

信息 | 结果 1 | 剖析 | 状态

CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME	TABLE_SCHEMA	TABLE_NAME	CONSTRAINT_TYPE
def	mysql	PRIMARY	mysql	time_zone_transitio	PRIMARY KEY
def	mysql	PRIMARY	mysql	time_zone_transitio	PRIMARY KEY
def	mysql	PRIMARY	mysql	user	PRIMARY KEY
def	test_db	PRIMARY	test_db	tb_emp	PRIMARY KEY
def	test_db	PRIMARY	test_db	tb_emp2	PRIMARY KEY
def	test_db	PRIMARY	test_db	tb_emp3	PRIMARY KEY
def	test_db	PRIMARY	test_db	tb_emp4	PRIMARY KEY

5.2.3 外键约束

外键用来在两个表之间建立关联,一个表的外键必须是另外一个表的主键，外键约束主要作用是保证数据引用的完整性,也叫参考完整性，定义外键后,不允许删除另外一个表的关联行.

这两个表中,关联字段是主键的表称为主表(父表),关联字段是外键的表称为从表(子表).

如:先定义一个部门表,部门编号是主键

```
CREATE TABLE tb_dept(
    id INT PRIMARY KEY,
    name VARCHAR(100)
    );
```

再定义一个员工表,员工表中有部门编号,员工表中部门编号是外键，外键定义语法:

```
[CONSTRAINT  外键约束名]  FOREIGN KEY (字段名) REFERENCES  主表名(主键名)
CREATE TABLE tb_emp5 (
    id INT PRIMARY KEY,
    name VARCHAR(30),
    deptid INT,
    salary FLOAT,
    CONSTRAINT fk_emp5_dept FOREIGN KEY (deptid) REFERENCES tb_dept(id)
)
```

员工表的 deptid 部门编号字段引用了部门表 tb_dept 中的 id 字段，员工表 tb_emp5 称为子表(从表)，部门表 tb_dept 称为父表(主表)．在这两个表中建立参考完整性约束.向员工表中插入记录时，外键字段 deptid 可以为 NULL,如果不为 NULL,则外键 deptid 字段的值必须是 tb_dept 部门表中的有的 id 值；一旦部门表 tb_dept 中的 id 字段被其他表引用了,则该条记录不能删除

当向 tb_emp5 表中插入记录时，外键字段的值可以为 NULL

```
INSERT INTO tb_emp5(id, name,  salary) VALUES (1001, 'lisi', 7865);
INSERT INTO tb_emp5(id, name,  salary) VALUES (1002, 'wangwu', 1234);
SELECT * FROM tb_emp5
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1001	lisi	(Null)	7865
1002	wangwu	(Null)	1234

如果向 tb_emp5 表中插入记录时,如果想要插入 deptid 字段的值,deptid 部门编号字段是外键, 引用了 tb_dept 表的 id 字段, 向 tb_emp5 表中插入的 deptid 字段必须是 部门表 tb_dept 中有的 id,否则插入失败

```
INSERT INTO tb_emp5(id, name, deptid, salary) VALUES (1003, 'zhaoliu', 9001, 1234);
```

当前tb_dept部门表中没有9001编号的部门,插入失败

信息 | 状态

```
INSERT INTO tb_emp5(id, name, deptid, salary) VALUES (1003, 'zhaoliu', 9001, 1234)
> 1452 - Cannot add or update a child row: a foreign key constraint fails (`test_db`.`tb_emp5`, CONSTRAINT `fk_emp5_dept`
FOREIGN KEY (`deptid`) REFERENCES `tb_dept` (`id`))
> 时间: 0.008s
```

先向 tb_dept 部门表中插入三条记录,再向 tb_emp5 表中插入记录,如果员工表中 deptid 在部门表 tb_dept 中存在,插入成功

```
INSERT INTO tb_dept (id, name) VALUES(9001, '开发部');
INSERT INTO tb_dept (id, name) VALUES(9002, '测试部');
INSERT INTO tb_dept (id, name) VALUES(9003, '销售部');

INSERT INTO tb_emp5(id, name, deptid, salary) VALUES (1003, 'zhaoliu', 9001, 1234);
SELECT * FROM tb_emp5
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1001	lisi	(Null)	7865
1002	wangwu	(Null)	1234
1003	zhaoliu	9001	1234

删除部门表 tb_dept 中的数据时,如果这条记录没有被引用,可以删除

```
DELETE FROM tb_dept WHERE id = 9002
```

信息 | 剖析 | 状态

```
DELETE FROM tb_dept WHERE id = 9002
> Affected rows: 1
> 时间: 0.007s
```

如果部门表中的记录被引用了, 删除失败, 9001 部门编号被 tb_emp5 表的记录引用,不能删除

```
DELETE FROM tb_dept WHERE id = 9001
```

信息 | 状态

```
DELETE FROM tb_dept WHERE id = 9001
> 1451 - Cannot delete or update a parent row: a foreign key constraint fails (`test_db`.`tb_emp5`, CONSTRAINT
`fk_emp5_dept` FOREIGN KEY (`deptid`) REFERENCES `tb_dept` (`id`))
> 时间: 0.008s
```

如果想要删除部门表中 9001 编号的记录,需要先把引用 9001 编号的员工表记录删除后才可以.当前案例中,先删除员工表中部门编号是 9001 的记录, 再删除部门表中 9001 号记录.

5.2.4 非空约束

非空约束指字段的值不能为 NULL,在添加数据时,必须给该字段插入数据

```
77   id INT PRIMARY KEY,
78   name VARCHAR(30) NOT NULL,
79   deptid INT,
80   salary FLOAT
81   );
82
83   INSERT INTO tb_emp6( id, name ) VALUES (1234, 'lisi');
84   INSERT INTO tb_emp6( id ) VALUES (4567);
```

在插入数据时,如果没有给name字段插入数据,则插入失败

信息 状态

```
INSERT INTO tb_emp6( id ) VALUES (4567)
> 1364 - Field 'name' doesn't have a default value
> 时间: 0.001s
```

5.2.5 唯一约束

唯一约束要求该列的值是唯一的,允许为 NULL

```
86   CREATE TABLE tb_emp7 (
87   id INT PRIMARY KEY,
88   name VARCHAR(30) UNIQUE,
89   deptid INT,
90   salary FLOAT
91   );
92  INSERT INTO tb_emp7( id, name ) VALUES (1234, 'lisi');
93  INSERT INTO tb_emp7( id ) VALUES (4567);
94  INSERT INTO tb_emp7( id ) VALUES (3698);
95  SELECT * FROM tb_emp7;
96  INSERT INTO tb_emp7( id, name ) VALUES (2587, 'lisi');
```

信息 状态

```
INSERT INTO tb_emp7( id, name ) VALUES (2587, 'lisi')
> 1062 - Duplicate entry 'lisi' for key 'name'
> 时间: 0.003s
```

PRIMARY KEY 主键约束也要求字段值是唯一的, 但是一个表中只能有一个主键; UNIQUE 唯一约束可以有多个; 主键不允许为 NULL,唯一约束可以为 NULL

5.2.6 默认值约束

可以给指定的列指定默认值,在插入记录时,如果没有给字段赋值就采用默认值.整数,小数,字符串类型的字段都可以设置默认值,日期类型的字段,只有 TIMESTAMP 类型的日期才能设置当前时期默认值,

```
111
112  CREATE TABLE tb_emp9 (
113  name VARCHAR(30),
114  gender char(1) DEFAULT '男',
115  firedate TIMESTAMP DEFAULT now()
116  );
117
118 INSERT INTO tb_emp9 (name) VALUES ('lisi');
119 SELECT * FROM tb_emp9;
120
```

性别字段默认为男
解聘时间默认为当前时间
只能TIMESTAMP时间类型有默认值,其他时间类型没有默认值

在插入数据时,如果不插入数据,字段的值就是默认值

信息 结果 1 剖析 状态

name	gender	firedate
lisi	男	2019-11-04 21:50:58

5.2.7 自动增长约束

在数据实际使用中，经常在每次插入记录时，让系统自动生成的字段值作为主键.

```
CREATE TABLE tb_emp10 (
id INT PRIMARY KEY auto_increment,
name VARCHAR(30),
gender char(1) DEFAULT '男',
deptid INT,
salary FLOAT
);
```

```
129 INSERT INTO tb_emp10(id, name, deptid, salary) VALUES (1003, 'zhaoliu', 9001, 1234);
130 INSERT INTO tb_emp10( id, name ) VALUES (1234, 'lisi');
131 INSERT INTO tb_emp10( name, deptid, salary) VALUES ( 'zhaoliu', 9001, 1234);
132 INSERT INTO tb_emp10( name, deptid, salary) VALUES ( 'chenqi', 9001, 798);
133 SELECT * FROM tb_emp10;
```

信息 结果 1 剖析 状态

id	name	gender	deptid	salary
1003	zhaoliu	男	9001	1234
1234	lisi	男	(Null)	(Null)
1235	zhaoliu	男	9001	1234
1236	chenqi	男	9001	798

在插入记录时,可以插入id字段值,
经常采用自动增长,即插入记录时不指定id字段的值

5.3 修改数据表

修改表是指修改数据表的结构,如:修改表名，修改字段的名称或类型，添加/删除字段，更改表的存储引擎，删除外键约束等..

5.3.1 修改表名

ALTER TABLE tb_tmp4 **RENAME TO** tb_tmp444

5.3.2 修改字段名

ALTER TABLE 表名 CHANGE 旧的字段名 新的字段名 [新数据类型]

如果表中有数据,修改了数据类型可能会影响到表中已有的数据.当表中已有数据时,不要轻易修改字段的数据类型.

ALTER TABLE tb_emp1 **CHANGE name e_name VARCHAR(50)**;

5.3.3 修改字段的数据类型

ALTER TABLE 表名 MODIFY 字段名 数据类型

ALTER TABLE tb_emp1 **MODIFY salary DECIMAL(10,2)**

5.3.4 添加字段

ALTER TABLE 表名 ADD 新字段名 数据类型 [约束] [FIRST|LAST|AFTER 字段]

```
ALTER TABLE tb_emp1 ADD gender char(1)    -- 默认新字段在最后
ALTER TABLE tb_emp1 ADD age TINYINT AFTER e_name;    -- 在 e_name 字段后面插入 age
ALTER TABLE tb_emp1 ADD managerid INT NOT NULL;  -- 添加带有约束的字段
-- 添加外键约束, 测试发现如果从表中有数据添加失败
ALTER TABLE tb_emp1 ADD CONSTRAINT fk_emp1_dept FOREIGN KEY(managerid) REFERENCES tb_dept(id);
```

5.3.5 删除字段

ALTER TABEL 表名 drop 字段名

ALTER TABLE tb_emp1 DROP id;

5.3.6 修改表的存储引擎

ALTER TABLE tb_emp2 **ENGINE = myisam** -- 修改存储引擎

SHOW CREATE TABLE tb_emp2; -- 查看表的定义,存储引擎就改为 MyISAM 了

注意,如果该表中有外键约束,不能修改存储引擎.

5.3.7 删除外键约束

ALTER TABLE tb_emp1 DROP FOREIGN KEY fk_emp1_dept

5.4 删除数据表

删除数据表,表的定义与表中的数据也会被删除. 在进行删除表操作前,最好对表中的数据进行备份.

DROP TABLE 表名

如果数据表之间存在外键关联关系,不能直接删除父表(主表),如果想要删除父表,需要先删除外键关联关系

5.5 综合案例

1) 创建 bjpowernode 数据库
2) 在 bjpowernode 数据库中创建两个表: 员工表与部门表

员工表 employee:

字段名	数据类型	约束
id	INT	自动增长, 主键
name	VARCHAR(30)	非空
salary	float	
birtyday	TIMESTAMP	
deptid	INT	外键
title	VARCHAR(30)	

部门表 dept

字段名	数据类型	约束
id	INT	主键
name	VARCHAR(50)	非空
address	VARCHAR(100)	

3) 修改 employee 员工表,添加 mobile 手机号字段,
4) 修改 employee 员工表,添加 gender 性别字段,非空,默认值为男

第6章 数据的插入,更新与删除

数据的基本操作(CRUD),包括插入数据 CREATE,查询数据的操作 READ,更新数据的操作 UPDATE,删除数据的操作 DELETE

6.1 插入数据

向数据库表中插入新的数据,可以插入完整的记录，也可以插入一部分，插入多条记录,插入另外一个查询结果

6.1.1 为表的所有字段都插入数据

INSERT INTO 表名(字段列表) VALUES(值列表);
注意值列表要与字段列表匹配,包括数量与类型都要匹配

```
INSERT INTO tb_emp3 ( id, name, deptid, salary) VALUES( 1002, 'wangwu', 9002, 789);
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1001	lisi	9001	7865
1002	wangwu	9002	789

向表中所有字段都插入数据时,表名后面的字段列表可以省略,但是不建议

```
INSERT INTO tb_emp3  VALUES( 1003, 'zhaoliu', 9002, 1234);
SELECT * FROM tb_emp3;
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1001	lisi	9001	7865
1002	wangwu	9002	789
1003	zhaoliu	9002	1234

6.1.2 为表的指定字段插入数据

```
INSERT INTO tb_emp3 ( id, name, deptid) VALUES( 1004, 'chenqi', 9003);
SELECT * FROM tb_emp3;
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1001	lisi	9001	7865
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1004	chenqi	9003	(Null)

6.1.3 同时插入多条记录

同时插入多条记录时，值列表之间使用逗号分隔，通过一个 INSERT 插入多条记录,比多条 INSERT 插入记录效率高

```
INSERT INTO tb_emp3 ( id, name, deptid, salary)
    VALUES( 1005, 'zhuba', 9002, 7089), (1006,'zhangsan',9001,456)
SELECT * FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1001	lisi	9001	7865
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1004	chenqi	9003	(Null)
1005	zhuba	9002	7089
1006	zhangsan	9001	456

6.1.4 将查询结果插入表中

INSERT INTO 表 1 (字段列表) SELECT 字段列表 FROM 表 2
从表 2 中查询数据，把查询结果插入到表 1 中

```
INSERT INTO tb_emp4 (id, name, deptid, salary)
SELECT id, name, deptid, salary FROM tb_emp3;
SELECT * FROM tb_emp4;
```

把tb_emp3表中的查询结果插入到tb_emp4表中

信息 | 剖析 | 状态

```
INSERT INTO tb_emp4 (id, name, deptid, salary)
SELECT id, name, deptid, salary FROM tb_emp3
> Affected rows: 5
> 时间: 0.007s
```

6.1.5 主从表数据插入

在从表中插入的数据,必须是在主表中有的数据值
主表 tb_dept 部门表有两条记录

```
SELECT * FROM tb_dept;
```

信息 | 结果 1 | 剖析 | 状态

id	name
9001	开发部
9003	销售部

从表 tb_emp6 的部门编号 detpid 引用了主表 tb_dept 的 id 字段

```
SHOW CREATE TABLE tb_emp6;
```

tb_emp6表有外键约束,表的deptid字段引用了tb_dept表的id字段

信息 结果 1 剖析 状态

Table	Create Table
tb_emp6	eptid`), CONSTRAINT `fk_emp_dept` FOREIGN KEY (`deptid`) REFERENCES `tb_dept` (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8

向 tb_emp6 表中插入数据时, deptid 字段的值必须是在主表 tb_dept 中存在的数据

```
INSERT INTO tb_emp6 ( id, name, deptid, salary)
    VALUES( 1005, 'zhuba', 9001, 7089), (1006,'zhangsan',9003,456)
INSERT INTO tb_emp6 ( id, name,  salary) VALUES( 1005, 'zhuba',  7089)
SELECT * FROM tb_emp6;
INSERT INTO tb_emp6 ( id, name, deptid, salary) VALUES( 1007, 'hehe', 9005, 7089)
```

向eb_emp6表中插入数据时, deptid字段的值必须 是引用表tb_dept中id字段的值
或者向tb_emp6表中插入数据是,deptid不插入数据,默认为NULL也可以

信息 状态

```
INSERT INTO tb_emp6 ( id, name, deptid, salary) VALUES( 1007, 'hehe', 9005, 7089)
> 1452 - Cannot add or update a child row: a foreign key constraint fails (`test_db`.`tb_emp6`, CONSTRAINT `fk_emp_dept`
FOREIGN KEY (`deptid`) REFERENCES `tb_dept` (`id`))
> 时间: 0.007s
```

6.2 更新数据

使用 UPDAE 语句更新表中的记录.

UPDATE 表名 SET 字段名 = 值 WHERE 条件

把表中符合条件记录的字段值更新为新的值,如 把工资低于 5000 元的员工,调整新的工资为原来工资的 1.5 倍

UPDATE tb_emp6 SET salary = salary * 1.5 WHERE salary < 5000;

```
SELECT * FROM tb_emp6;
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1005	zhuba	9001	7089
1006	zhangsan	9003	1000
1234	lisi	(Null)	(Null)

```
UPDATE tb_emp6 SET salary = salary * 1.5 WHERE salary < 5000;
SELECT * FROM tb_emp6;
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1005	zhuba	9001	7089
1006	zhangsan	9003	1500
1234	lisi	(Null)	(Null)

如果在更新数据时,没有指定 WHERE 条件会更新所有的记录

```
30  UPDATE tb_emp6 SET salary = 8888 ;
31  SELECT * FROM tb_emp6;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1005	zhuba	9001	8888
1006	zhangsan	9003	8888
1234	lisi	(Null)	8888

```
30  UPDATE tb_emp6 SET salary = salary * 2 ;
31  SELECT * FROM tb_emp6;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1005	zhuba	9001	17776
1006	zhangsan	9003	17776
1234	lisi	(Null)	17776
4567	wangwu	(Null)	(Null)

6.3 删除数据

DELETE FROM 表名 where 条件

如 tb_emp3 表中有如下记录

```
33  SELECT * FROM tb_emp3;
34
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1004	chenqi	9003	(Null)
1005	zhuba	9002	7089
1006	zhangsan	9001	456

把工资低于 500 的记录删除

```
34
35  DELETE FROM tb_emp3 WHERE salary < 500
36  SELECT * FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1004	chenqi	9003	(Null)
1005	zhuba	9002	7089

把工资为 NULL 的记录删除

```
34
35  DELETE FROM tb_emp3 WHERE salary IS NULL;
36  SELECT * FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1005	zhuba	9002	7089

第7章 运算符

7.1 算术运算符

```
37
38  SELECT salary , salary /10 , salary % 5 FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

salary	salary /10	salary % 5
789	78.9	4
1234	123.4	4
7089	708.9	4

```
37
38  SELECT salary , salary /0 , salary % 0  FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

salary	salary /0	salary % 0
789	(Null)	(Null)
1234	(Null)	(Null)
7089	(Null)	(Null)

7.2 比较运算符

= 等于
<=> 安全的等于
!= 或 <> 不等于
<= < >= >
IS NULL 判断值是否为 NULL
IS NOT NULL

BETWEEN AND 判断是否在某个范围内
IN 判断值是否在某个列表中
NOT IN 不在列表中
LIKE 匹配通配符

7.2.1 相等=

```
41  -- 使用=判断相等时，如果有一个参数为NULL,则结果为NULL;
    如果字符串与整数进行判等,MySQL会自动将字符串转换为数字
42  SELECT salary , salary = 789, salary = '1234' , salary = NULL FROM tb_emp3;
43
```

信息 | 结果 1 | 剖析 | 状态

salary	salary = 789	salary = '1234'	salary = NULL
789	1	0	(Null)
1234	0	1	(Null)
7089	0	0	(Null)
(Null)	(Null)	(Null)	(Null)
(Null)	(Null)	(Null)	(Null)

7.2.2 安全的等于<=>

与=操作结果是一样，不同的是<=>判断相等时,可以判断 NULL 值

```
43
44  SELECT salary , salary <=> 789, salary  <=> '1234' , salary  <=> NULL FROM tb_emp3;
45
```

信息 | 结果 1 | 剖析 | 状态

salary	salary <=> 789	salary <=> '1234'	salary <=> NULL
789	1	0	0
1234	0	1	0
7089	0	0	0
(Null)	0	0	1
(Null)	0	0	1

7.2.3 不等于<> 或 !=

```
43
44  SELECT salary , salary != 789, salary  <> '1234' , salary  <> NULL FROM tb_emp3;
45
```

信息 | 结果 1 | 剖析 | 状态

salary	salary != 789	salary <> '1234'	salary <> NULL
789	0	1	(Null)
1234	1	0	(Null)
7089	1	1	(Null)
(Null)	(Null)	(Null)	(Null)
(Null)	(Null)	(Null)	(Null)

7.2.4 IS NULL

```
43
44  SELECT salary , salary IS NOT NULL, salary IS NULL, ISNULL(salary) FROM tb_emp3;
45
```

信息 | 结果 1 | 剖析 | 状态

salary	salary IS NOT NULL	salary IS NULL	ISNULL(salary)
789	1	0	0
1234	1	0	0
7089	1	0	0
(Null)	0	1	1
(Null)	0	1	1

7.2.5 BETWEEN AND

```
44  SELECT salary , salary BETWEEN 1234 AND 7089,
45                  salary >=1234 AND salary <=7089   FROM tb_emp3;
```

信息 | 结果 1 | 剖析 | 状态

salary	salary BETWEEN 1234 AND 7089	salary >=1234 AND salary <=7089
789	0	0
1234	1	1
7089	1	1
(Null)	(Null)	(Null)
(Null)	(Null)	(Null)
6666	1	1

7.2.6 IN

判断数是否为 IN 指定列表中的一个

```
47  SELECT * FROM tb_emp3 WHERE deptid IN (9001,9003);
48  SELECT * FROM tb_emp3 WHERE deptid=9001 OR deptid=9003;
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1005	zhuba	9003	7089
1008	feifei	9001	6666

```
47  SELECT * FROM tb_emp3 WHERE deptid NOT IN (9001,9003);
48  SELECT * FROM tb_emp3 WHERE NOT deptid  IN (9001,9003);
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234

7.2.7 LIKE

LIKE 用来匹配字符串,可以使用两种通配符:

% 匹配任意数量的任意字符

_ 只能匹配一个字符

```
52  SELECT * FROM tb_emp3  WHERE name LIKE 'z%'     -- 匹配姓名以z开头的字符串
53  SELECT * FROM tb_emp3  WHERE name LIKE '%i'     -- 匹配姓名以i结尾的字符串
54  SELECT * FROM tb_emp3  WHERE name LIKE '_h%'    -- 匹配姓名中第二个字符是h的字符串
55  SELECT * FROM tb_emp3  WHERE name LIKE '%a%'    -- 匹配姓名中含有a的字符串
56
```

信息 | 结果 1 | 剖析 | 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1005	zhuba	9003	7089
1006	zhangsan	(Null)	(Null)

7.2.8 REGEXP

regular expression 正则表达式,用来匹配字符串

表达式 REGEXP 正则表达式, 判断前面表达式是否匹配指定的正则表达式.常用的通配符有:

^, 匹配以指定的字符开头的字符串

$, 匹配以指定的字符结尾的字符串

. 匹配任何一个字符

[abc] 匹配 a 或者 b 或者 c 中的一个,也可以使用范围[a-z], [0-9]

* 匹配任意次

```
SELECT * FROM tb_emp3  WHERE name REGEXP '^z'      -- 匹配姓名以 z 开头的字符串
SELECT * FROM tb_emp3  WHERE name REGEXP 'i$'      -- 匹配姓名以 i 结尾的字符串
SELECT * FROM tb_emp3  WHERE name REGEXP '.h'      -- 匹配姓名中第二个字符是 h 的字符串
SELECT * FROM tb_emp3  WHERE name REGEXP '.*a.*'   -- 匹配姓名中含有 a 的字符串
```

7.3逻辑运算符

NOT 或 ! 逻辑非
AND 或 && 逻辑与
OR 或 || 逻辑或
XOR 逻辑异或

```
64  SELECT * FROM tb_emp3  WHERE salary >= 2000 AND  salary <=8000
65  SELECT * FROM tb_emp3  WHERE salary >= 2000 &&  salary <=8000
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1005	zhuba	9003	7089
1008	feifei	9001	6666

```
67  SELECT * FROM tb_emp3  WHERE salary < 2000  OR  salary IS NULL
68  SELECT * FROM tb_emp3  WHERE salary < 2000  ||  salary IS NULL
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1006	zhangsan	(Null)	(Null)
1007	chenqi	(Null)	(Null)

```
70  SELECT * FROM tb_emp3 WHERE  NOT salary IS NULL
```

信息 结果 1 剖析 状态

id	name	deptid	salary
1002	wangwu	9002	789
1003	zhaoliu	9002	1234
1005	zhuba	9003	7089
1008	feifei	9001	6666

```
73  SELECT  10,  NOT 10,  NULL ,  NOT  NULL,  NOT 0
74
```

信息 结果 1 剖析 状态

10	NOT 10	NULL	NOT NULL	NOT 0
10	0	(Null)	(Null)	1

7.4 位运算符

位运算符的操作数是二进制位.

& 按位与
| 按位或
^ 按位异或
~ 按位取反
<< 按位左移
>> 按位右移

```
74
75  SELECT salary , salary << 1 , salary >> 1 , ~salary, salary & 7, salary | 7 FROM tb_emp3
76
```

信息 | 结果 1 | 剖析 | 状态

salary	salary << 1	salary >> 1	~salary	salary & 7	salary | 7
789	1578	394	18446744073709550826	5	791
1234	2468	617	18446744073709550381	2	1239
7089	14178	3544	18446744073709544526	1	7095
(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
6666	13332	3333	18446744073709544949	2	6671

7.5运算符优先级

! 逻辑非最高
-负号 ~按位取反
^ 按位异或
* / % 算术
+ - 算术加减
<< >> 移位
& 按位与
| 按位或
比较运算符 = <> >= < IS LIKE IN
NOT 逻辑非
AND
OR
= 赋值

算术 > 比较运算符 > 逻辑运算符 >赋值运算符
在使用时如果不确定就使用小括弧

第8章 函数

8.1数学函数

常用的数学函数有：绝对值函数，三角函数，随机数函数，四舍五入函数..

求绝对值函数

```
SELECT  ABS(-12), ABS(-6.78), PI()
```

信息 | 结果 1 | 剖析 | 状态

ABS(-12)	ABS(-6.78)	PI()
12	6.78	3.141593

平方根,求余

```
SELECT SQRT(100), MOD(100,7)
```

信息 | 结果 1 | 剖析 | 状态

SQRT(100)	MOD(100,7)
10	2

ceil(n) 返回大于等于 n 的最小整数

```
SELECT CEIL(5.12), CEILING(7.89), CEIL(-5.9)  -- 大于指定数的最小整数
```

信息 | 结果 1 | 剖析 | 状态

CEIL(5.12)	CEILING(7.89)	CEIL(-5.9)
6	8	-5

floor(n) 返回小于等于 n 的最大整数

```
SELECT FLOOR(5.12), FLOOR(7.89), FLOOR(-5.9)
```

信息 | 结果 1 | 剖析 | 状态

FLOOR(5.12)	FLOOR(7.89)	FLOOR(-5.9)
5	7	-6

随机数与四舍五入

```
SELECT RAND(),  ROUND(5.112),  ROUND(7.89)
```

信息 | 结果 1 | 剖析 | 状态

RAND()	ROUND(5.112)	ROUND(7.89)
0.16266302965829432	5	8

```
SELECT ROUND(12345.1278,1), ROUND(12345.1278,2), ROUND(12345.1278,-2)
```

保留小数点后1位　保留小数点后两位　保留小数点前两位

信息 | 结果 1 | 剖析 | 状态

ROUND(12345.1278,1)	ROUND(12345.1278,2)	ROUND(12345.1278,-2)
12345.1	12345.13	12300

```
SELECT TRUNCATE(12345.1278,1), TRUNCATE(12345.1278,2), TRUNCATE(12345.1278,-2)
```

截取小数点后1位

信息 | 结果 1 | 剖析 | 状态

TRUNCATE(12345.1278,1)	TRUNCATE(12345.1278,2)	TRUNCATE(12345.1278,-2
12345.1	12345.12	12300

8.2 字符串函数

8.2.1 字符的个数 char_length

```
SELECT CHAR_LENGTH('hi动力节点'), LENGTH('hi动力节点')
```

字符的数量　所占字节的数量

信息 | 结果 1 | 剖析 | 状态

CHAR_LENGTH('hi动力节点')	LENGTH('hi动力节点')
6	14

8.2.2 字符串连接 concat()

```
SELECT CONCAT("my","sql","good",5.7), CONCAT('my','sql',NULL)
```

把各个字符串连接起来

信息 | 结果 1 | 剖析 | 状态

CONCAT("my","sql","good",5.7)	CONCAT('my','sql',NULL)
mysqlgood5.7	(Null)

```
SELECT CONCAT_WS('-',"my","sql","good",5.7), CONCAT_WS('*','my','sql',NULL)
```

第一个参数是连接符，使用连接符把各个字符串连接起来
会忽略NULL

信息 | 结果 1 | 剖析 | 状态

CONCAT_WS('-',"my","sql","good",5.7)	CONCAT_WS('*','my','sql',NULL)
my-sql-good-5.7	my*sql

8.2.3 字符串替换 insert

```
24  SELECT INSERT("helloworld", 3,2,'*');
25
```

把helloworld字符串中从3开始的2个字符使用*替换
注意：mysql字符串索引值是从1开始的

信息 | 结果 1 | 剖析 | 状态

INSERT("helloworld", 3,2,'*')
he*oworld

8.2.4 大小写转换

```
26  SELECT LOWER("Good"), UPPER('Good'), LCASE('Good'), UCASE('Good')
```

信息 | 结果 1 | 剖析 | 状态

LOWER("Good")	UPPER('Good')	LCASE('Good')	UCASE('Good')
good	GOOD	good	GOOD

8.2.5 截取指定长度的字符串 Left,Right

```
28  SELECT LEFT('helloworld',5), RIGHT('helloworld',5);
29
```

信息 | 结果 1 | 剖析 | 状态

LEFT('helloworld',5)	RIGHT('helloworld',5)
hello	world

8.2.6 填充字符串 lpad,rpad

```
30  SELECT LPAD('hello',10,'*'), RPAD('hello',10,'*')
```

信息 | 结果 1 | 剖析 | 状态

指定字符串长度，不足的位数使用*填充

LPAD('hello',10,'*')	RPAD('hello',10,'*')
*****hello	hello*****

8.2.7 删除前后空格 trim

```
SELECT TRIM('   hello   '), LTRIM('   hello   '), RTRIM('   hello   ');
```

信息 | 结果 1 | 剖析 | 状态

TRIM(' hello ')	LTRIM(' hello ')	RTRIM(' hello ')
hello	hello	hello

```
SELECT TRIM('xx' FROM 'xxooxxoo' )
```

把'xxooxxoo'字符串左右两端的'xx'字符串删除

信息 | 结果 1 | 剖析 | 状态

TRIM('xx' FROM 'xxooxxo
ooxxoo

8.2.8 重复生成字符串 repeat()

```
SELECT REPEAT('abc',3);
```

信息 | 结果 1 | 剖析 | 状态

REPEAT('abc',3)
abcabcabc

8.2.9 字符串大小比较

```
SELECT STRCMP('abc','def'), STRCMP('xx','XX')
```

信息 | 结果 1 | 剖析 | 状态

STRCMP('abc','def')	STRCMP('xx','XX')
-1	0

8.2.10 取子串 substring,mid

```
SELECT SUBSTRING('helloworld',1,5), SUBSTRING('helloworld',-5),SUBSTRING('helloworld',5);
```

从1开始取5个字符　　从结尾向前取5个字符　　从第5个字符开始取到最后

信息 | 结果 1 | 剖析 | 状态

SUBSTRING('helloworld',1	SUBSTRING('helloworld',-	SUBSTRING('helloworld',5
hello	world	oworld

```
39
40  SELECT MID('helloworld',1,5), MID('helloworld',-5),MID('helloworld',5);
41
```

信息 | 结果 1 | 剖析 | 状态

MID('helloworld',1,5)	MID('helloworld',-5)	MID('helloworld',5)
hello	world	oworld

8.2.11 匹配子串开始的位置

```
41
42  SELECT LOCATE('o','helloworld'), POSITION('o' IN 'helloworld'), INSTR('helloworld','o')
```

信息 | 结果 1 | 剖析 | 状态

LOCATE('o','helloworld')	POSITION('o' IN 'hellowo	INSTR('helloworld','o')
5	5	5

8.3 日期函数

```
43
44  SELECT CURRENT_DATE, CURDATE(), CURRENT_DATE();
45
```

信息 | 结果 1 | 剖析 | 状态

CURRENT_DATE	CURDATE()	CURRENT_DATE()
2019-11-07	2019-11-07	2019-11-07

```
46  SELECT CURRENT_TIME, CURTIME(), CURRENT_TIME();
47
```

信息 | 结果 1 | 剖析 | 状态

CURRENT_TIME	CURTIME()	CURRENT_TIME()
20:55:49	20:55:49	20:55:49

```
48  SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), NOW(), SYSDATE(), LOCALTIME()
49
50
```

信息 | 结果 1 | 剖析 | 状态

CURRENT_TIMESTAMP	CURRENT_TIMESTAMP()	NOW()	SYSDATE()	LOCALTIME()
2019-11-07 20:57:12	2019-11-07 20:57:12	2019-11-07 20:57:12	2019-11-07 20:57:12	2019-11-07 20:57:12

8.4 条件判断函数

8.4.1 IF 函数

```
50  SELECT IF( 3 > 2 ,3 ,2), IF(3>2,'yes','no'), IF(STRCMP('xx','YY'),'yes','no')
51
```

STRCMP比较两个字符串,相等返回0

信息 | 结果 1 | 剖析 | 状态

IF(3 > 2 ,3 ,2)	IF(3>2,'yes','no')	IF(STRCMP('xx','YY'),'yes','no')
3	yes	yes

8.4.2 IFNULL

```
51
52  SELECT IFNULL(NULL,0), IFNULL(123,1), IFNULL(3/0,'error');
53
```

如果前面为NULL,返回第二个表达式的值,如果第一个不为NULL就返回第一个表达式的值

信息 | 结果 1 | 剖析 | 状态

IFNULL(NULL,0)	IFNULL(123,1)	IFNULL(3/0,'error')
0	123	error

8.4.3 CASE 函数

```
54  SELECT CASE 2       -- 表达式
55    WHEN 1 THEN       -- 当表达式的值是1 ,就返回one
56      'one'
57      WHEN 2 THEN     -- 当表达式的值是2 ,就返回two
58      'two'
59      WHEN 3 THEN
60      'three'
61    ELSE
62      'more'
63  END ;
```

信息 | 结果 1 | 剖析 | 状态

CASE 2 -- 表达式WHEN 1
two

8.5 获得系统信息的函数

获得 MySQL 的版本号

```
64
65  SELECT VERSION()
66
```

信息 | 结果 1 | 剖析 | 状态

VERSION()
5.5.25a

获得当前连接的 ID

```
SELECT CONNECTION_ID()
```

用户登录 MySQL时， 系统会为每个连接分配一个id

CONNECTION_ID()
14

显示当前所有的连接

```
mysql> show processlist;
```

显示当前所有的连接数

```
+----+------+-----------------+---------+---------+------+-------+------------------+
| Id | User | Host            | db      | Command | Time | State | Info             |
+----+------+-----------------+---------+---------+------+-------+------------------+
| 14 | root | localhost:65474 | test_db | Sleep   |   69 |       | NULL             |
| 15 | root | localhost:65476 | test_db | Sleep   | 4943 |       | NULL             |
| 17 | root | localhost:50188 | NULL    | Query   |    0 | NULL  | show processlist |
+----+------+-----------------+---------+---------+------+-------+------------------+
3 rows in set (0.00 sec)
```

显示当前的数据库

```
SELECT  DATABASE(),  SCHEMA()
```

DATABASE()	SCHEMA()
test_db	test_db

显示当前用户

```
SELECT USER(), CURRENT_USER, CURRENT_USER(), SYSTEM_USER(), SESSION_USER()
```

USER()	CURRENT_USER	CURRENT_USER()	SYSTEM_USER()	SESSION_USER()
root@localhost	root@localhost	root@localhost	root@localhost	root@localhost

8.6加密函数

```
SELECT PASSWORD('123')
```

PASSWORD('123')
*23AE809DDACAF96AF0FD78ED04B6A265E05AA257

```
SELECT MD5('123')
```

MD5('123')
202cb962ac59075b964b07152d234b70

SELECT ENCODE('bjpowernode','hehe') -- 使用’hehe’密钥对前面的字符串加密

SELECT DECODE(ENCODE('bjpowernode','hehe'),'hehe') -- 使用同一个密钥'hehe'对密文进行解密

第9章 查询

```
SELECT
    {字段列表}
    [
        FROM 表 1, 表 2,...
        [WHERE  查询条件]
        [GROUP BY 分组字段 [ HAVING  筛选分组条件]]
        [ORDER BY 排序字段]
        [LIMIT  offset,  count]
    ]
```

9.1 单表查询

从单个 表中查询数据

9.1.1 查询指定的字段

```
1  SELECT f_id, f_name, f_price FROM fruits;
2
```

信息 | 结果 1 | 剖析 | 状态

f_id	f_name	f_price
a1	blackberry	10.2
a2	apricot	2.2
b1	blackberry	10.2
b2	berry	7.6
b5	xxxx	3.6
bs1	orange	11.2
bs2	melon	8.2
c0	chenry	3.2
l2	lemon	6.4
m1	mango	15.6
m2	xbabay	2.6

9.1.2 查询所有字段

```
1  SELECT  f_id, s_id, f_name,f_price FROM fruits;
2
3  SELECT * FROM fruits;      -- 一般用在测试时,在实际开发时,一般指定具体字段
4
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
a2	103	apricot	2.2
b1	101	blackberry	10.2
b2	104	berry	7.6
b5	107	xxxx	3.6
bs1	102	orange	11.2
bs2	105	melon	8.2

9.1.3 查询符合条件的记录

```
5  SELECT * FROM fruits WHERE  f_price > 10
6
```

把符合条件 记录筛选出来

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
t1	102	banana	10.3

```
5  SELECT * FROM fruits WHERE  f_price  BETWEEN 5 AND 10
6
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
b2	104	berry	7.6
bs2	105	melon	8.2
l2	104	lemon	6.4
o2	103	coconut	9.2
t2	102	grape	5.3

```
SELECT * FROM fruits WHERE f_name LIKE '%a%';
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
a2	103	apricot	2.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m2	105	xbabay	2.6
t1	102	banana	10.3
t2	102	grape	5.3
t4	107	xbababa	3.6

9.1.4 带 IN 的查询

```
-- 把供应商编号s_id为101, 102 , 103 的记录查询出来
SELECT * FROM fruits  WHERE s_id IN( 101, 102, 103 );
SELECT * FROM fruits  WHERE s_id=101 OR s_id=102 OR s_id=103 ;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
a2	103	apricot	2.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
c0	101	chenry	3.2

```
-- 把供应商编号s_id 不是 101, 102 , 103 的记录查询出来
SELECT * FROM fruits  WHERE s_id NOT IN( 101, 102, 103 );
SELECT * FROM fruits  WHERE NOT  s_id NOT IN( 101, 102, 103 );
SELECT * FROM fruits  WHERE s_id!=101 AND s_id!=102 AND s_id!=103 ;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
b2	104	berry	7.6
b5	107	xxxx	3.6
bs2	105	melon	8.2
l2	104	lemon	6.4
m1	106	mango	15.6
m2	105	xbabay	2.6
m3	105	xxxtt	11.6
t4	107	xbababa	3.6

9.1.5 带有 BETWEEN...AND 的查询

```
16  -- 查询价格在5元到10元之间的水果
17  SELECT * FROM fruits  WHERE f_price BETWEEN 5 AND 10
18  SELECT * FROM fruits  WHERE f_price >= 5 AND f_price <= 10
19
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
b2	104	berry	7.6
bs2	105	melon	8.2
l2	104	lemon	6.4
o2	103	coconut	9.2
t2	102	grape	5.3

9.1.6 带有 LIKE 的查询

```
20  -- 查询水果名称第一个字母 是b的记录
21  SELECT * FROM fruits WHERE f_name LIKE 'b%';
22
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
b2	104	berry	7.6
t1	102	banana	10.3

```
22
23  -- 查询水果名称中包含a的记录
24  SELECT * FROM fruits WHERE f_name LIKE '%a%';
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
a2	103	apricot	2.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m2	105	xbabay	2.6
t1	102	banana	10.3
t2	102	grape	5.3

```
26  -- 查询水果名称中 不 包含a的记录
27  SELECT * FROM fruits WHERE f_name NOT LIKE '%a%';
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
b2	104	berry	7.6
b5	107	xxxx	3.6
bs2	105	melon	8.2
c0	101	chenry	3.2
l2	104	lemon	6.4
m3	105	xxxtt	11.6
o2	103	coconut	9.2

9.1.7 查询空值

```
29  -- 查询email为空的客户
30  SELECT * FROM customers WHERE c_email IS NULL
31  SELECT * FROM customers WHERE c_email <=> NULL
```

信息 | 结果 1 | 剖析 | 状态

c_id	c_name	c_address	c_city	c_zip	c_contact	c_email
10003	Nethood	400 Street	Qingdao	266000	Loucong	(Null)
10005	张三	大族企业湾	北京	100010	zhangxiaosan	(Null)
100006	李四	天通苑	北京	100020	lisisi	(Null)

```
34  -- 查询email 不为空的客户
35  SELECT * FROM customers WHERE c_email IS NOT NULL
36  SELECT * FROM customers WHERE NOT c_email IS  NULL
37
```

信息 | 结果 1 | 剖析 | 状态

c_id	c_name	c_address	c_city	c_zip	c_contact	c_email
10001	RedHook	200 street	tianjin	300000	LiMing	liming@163.
10002	Stars	300 Street	Dalian	1100000	Zhangbo	zhangbo@q
10004	Joto	500 Street	Haikou	270000	Yangshan	yangsan@ho

9.1.8 AND 连接多个条件

```
38  -- 查询供应商编号为101，水果价格小于10的记录
39  SELECT * FROM fruits WHERE s_id = 101 AND f_price < 10
40
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
c0	101	chenry	3.2

9.1.9 带 OR 的多条件查询

```
41  -- 查询供应商编号是101 或者 水果价格大于10的记录
42  SELECT * FROM fruits WHERE s_id = 101 OR f_price > 10
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
c0	101	chenry	3.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
t1	102	banana	10.3

9.1.10 使用 DISTINCT 去掉重复的记录

```
44  -- 查询fruits水果表中所有的供应商编号
45  SELECT s_id FROM fruits;
46  -- 查询fruits水果表中所有的供应商编号，去掉重复的记录
47  SELECT DISTINCT s_id FROM fruits;
48
```

信息 | 结果 1 | 剖析 | 状态

s_id
101
103
104
107
102
105
106

9.1.11 对查询结果排序

```
48
49  -- 对fruits水果表按价格升序排序
50  SELECT * FROM fruits ORDER BY f_price;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a2	103	apricot	2.2
m2	105	xbabay	2.6
c0	101	chenry	3.2
t4	107	xbababa	3.6
b5	107	xxxx	3.6
t2	102	grape	5.3
l2	104	lemon	6.4

```
51
52  -- 对fruits水果表按价格降序排序
53  SELECT * FROM fruits ORDER BY f_price DESC;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
m1	106	mango	15.6
m3	105	xxxtt	11.6
bs1	102	orange	11.2
t1	102	banana	10.3
a1	101	blackberry	10.2
b1	101	blackberry	10.2
o2	103	coconut	9.2

```
54
55  SELECT * FROM fruits WHERE s_id IN (101,102,103) ORDER BY f_price DESC;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
bs1	102	orange	11.2
t1	102	banana	10.3
a1	101	blackberry	10.2
b1	101	blackberry	10.2
o2	103	coconut	9.2
t2	102	grape	5.3
c0	101	chenry	3.2

9.1.12 使用 LIMIT 限制数量

```
57  -- 查询最贵的5条水果记录
58  SELECT * FROM fruits ORDER BY f_price DESC LIMIT 5 ;
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
m1	106	mango	15.6
m3	105	xxxtt	11.6
bs1	102	orange	11.2
t1	102	banana	10.3
a1	101	blackberry	10.2

```
60  -- 查询第6到第10贵的水果记录
61  SELECT * FROM fruits ORDER BY f_price DESC LIMIT 5,5 ;
62
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
b1	101	blackberry	10.2
o2	103	coconut	9.2
bs2	105	melon	8.2
b2	104	berry	7.6
l2	104	lemon	6.4

limit 位置偏移量, 行数
第一条记录的位置偏移量是0
limt 5表示从0开始取5条记录
limit 5,5表示从5开始取5条记录
limit 10,5 表示从第10条记录开始取5条记录
将来可以使用limt offset, count实现分页查询

9.1.13 分组

根据供应商编号进行分组,查看每个供应商供应水果的数量

```
3  -- 供应商编号及供应商供应水果的数量
4  SELECT s_id , COUNT(*) FROM fruits GROUP BY s_id;
5
```

信息 | 结果 1 | 剖析 | 状态

s_id	COUNT(*)
101	3
102	3
103	2
104	2
105	3
106	1
107	2

根据供应商编号分组,查看每个供应商供应水果的名称

```sql
-- 根据s_id分组，将每个供应商的水果名称显示出来
SELECT s_id, GROUP_CONCAT(f_name) FROM fruits GROUP BY s_id;
```

信息 | 结果 1 | 剖析 | 状态

s_id	GROUP_CONCAT(f_name)
101	blackberry,blackberry,chenry
102	grape,banana,orange
103	apricot,coconut
104	lemon,berry
105	xbabay,xxxtt,melon
106	mango
107	xxxx,xbababa

对分组结果进行过滤,之前 WHER 是对分组之前的数据进行过滤, HAVING 是对分组之后 的数据进行过滤

```sql
-- 根据s_id分组，将供应商的水果名称显示出来,只显示供应水果数量超过2种的
SELECT s_id, GROUP_CONCAT(f_name) FROM fruits
GROUP BY s_id HAVING COUNT(f_name) > 2
```

信息 | 结果 1 | 剖析 | 状态

s_id	GROUP_CONCAT(f_name)
101	blackberry,blackberry,chenry
102	grape,banana,orange
105	xbabay,xxxtt,melon

对分组结果排序

```sql
-- 供应商编号及供应商供应水果的数量,根据数量升序排序
SELECT s_id , COUNT(*) AS total
FROM fruits
GROUP BY s_id
ORDER BY total
```

给显示的字段起个别名

信息 | 结果 1 | 剖析 | 状态

s_id	total
106	1
103	2
104	2
107	2
102	3
105	3

注意:

在 GROUP BY 分组时, SELECT 后面的字段要么是分组字段,要么是聚合函数

9.2 使用聚合函数

9.2.1 count()

```
19
20  -- count()函数统计记录的行数, count(*)统计所有行数,包括空值
21  -- 查看customers表中记录总数
22  SELECT COUNT(*) FROM customers
23
```

信息 | 结果 1 | 剖析 | 状态

COUNT(*)
6

```
24  -- count(字段名)统计指定列的总的行数, 忽略空值行
25  -- 查看customers表中有email的记录数
26  SELECT COUNT(c_email) FROM customers;
27
```

信息 | 结果 1 | 剖析 | 状态

COUNT(c_email)
3

```
3  -- 对分组结果进行统计
4  -- 供应商编号及供应商供应水果的数量
5  SELECT s_id , COUNT(*) FROM fruits GROUP BY s_id;
```

信息 | 结果 1 | 剖析 | 状态

s_id	COUNT(*)
101	3
102	3
103	2
104	2
105	3
106	1
107	2

9.2.2 sum()

```
29  -- SUM(字段名) 对指定的列求和
30  -- 供应商编号是101 供应水果总价
31  SELECT SUM(f_price) FROM fruits WHERE s_id = 101
32
```

信息 | 结果 1 | 剖析 | 状态

SUM(f_price)
23.6

```
33  -- 计算每个供应商的水果总价
34  SELECT s_id , SUM(f_price) FROM fruits GROUP BY s_id;
35
```

信息 | 结果 1 | 剖析 | 状态

s_id	SUM(f_price)
101	23.6
102	26.8
103	11.4
104	14
105	22.4
106	15.6
107	7.2

9.2.3 avg()

```
36  -- 计算每个供应商的水果的平均价
37  SELECT s_id , AVG(f_price) FROM fruits GROUP BY s_id;
```

信息 | 结果 1 | 剖析 | 状态

s_id	AVG(f_price)
101	7.866667
102	8.933333
103	5.7
104	7
105	7.466667
106	15.6
107	3.6

9.2.4 max()

```
39  -- 查询水果的最高价
40  SELECT MAX(f_price) FROM fruits
```

信息 | 结果 1 | 剖析 | 状态

MAX(f_price)
15.6

```
42  -- 查询不同供应商水果的最高价
43  SELECT s_id, MAX(f_price) FROM fruits GROUP BY s_id;
```

信息 | 结果 1 | 剖析 | 状态

s_id	MAX(f_price)
101	10.2
102	11.2
103	9.2
104	7.6
105	11.6
106	15.6
107	3.6

9.2.5 min()

```
45  -- 查询不同供应商水果的最低价
46  SELECT s_id, MIN(f_price) FROM fruits GROUP BY s_id;
```

信息 | 结果 1 | 剖析 | 状态

s_id	MIN(f_price)
101	3.2
102	5.3
103	2.2
104	6.4
105	2.6
106	15.6
107	3.6

9.3 连接查询(多表查询)

连接查询就是从多个表中查询数据,又称为多表查询,包括内连接,外连接等. 在 SQL 语言有一个关系数据操作叫做笛卡儿积操作, 就是没有连接条件的多个表返回的结果

```
48  -- 查看fruits表有16条记录
49  SELECT COUNT(*) FROM fruits;
50  -- 查看supplies表中有7条记录
51  SELECT COUNT(*) FROM supplies
52
53  SELECT * FROM fruits, supplies
```

笛卡儿积操作
没有连接条件的多表查询

信息 | 结果 1 | 剖析 | 状态

就是把fruits表的每一条记录与supplies表的每一条记录进行连接

f_id	s_id	f_name	f_price	s_id(1)	s_name	s_city	s_zip	s_call
a1	101	blackberry	10.2	101	Fastfrui	Tianjin	3000000	12345
a1	101	blackberry	10.2	102	LT suplies	Chongqing	40000	4455
a1	101	blackberry	10.2	103	ACME	Shanghai	200000	9000
a1	101	blackberry	10.2	104	FNK	Qingdao	266000	2545464
a1	101	blackberry	10.2	105	Good	Zhengzyhou	122200	798798
a1	101	blackberry	10.2	106	Just	Beijing	100101	78974
a1	101	blackberry	10.2	107	Dk	Shijiazhang	123400	13415456
a2	103	apricot	2.2	101	Fastfrui	Tianjin	3000000	12345
a2	103	apricot	2.2	102	LT suplies	Chongqing	40000	4455

SELECT * FROM fruits, supplies | 只读 | 查询时间: 0.022s | 第 1 条记录 (共 112 条)

所谓多表连接,就是在笛卡儿积数据记录中,按照相应字段值的比较条件选择生成一个新的关系.连接分为内连接(INNER JOIN), 外连接(OUTER JOIN)与交叉连接(CROSS JOIN). 交叉连接就是笛卡儿积

9.3.1 内连接查询

内连接查询就是在笛卡儿积的数据记录中, 保底表中匹配条件 的记录

```
55  -- 在水果表,和供应商表中查询供应商编号, 水果名称,水果价格,供应商名称
56  SELECT supplies.s_id, f_name, f_price, s_name
57  FROM fruits , supplies
58  WHERE fruits.s_id = supplies.s_id
59
```

两个表中的都有s_id字段,使用表名.列名的形式进行区分

信息 | 结果 1 | 剖析 | 状态

s_id	f_name	f_price	s_name
101	blackberry	10.2	Fastfrui
103	apricot	2.2	ACME
101	blackberry	10.2	Fastfrui
104	berry	7.6	FNK
107	xxxx	3.6	Dk
102	orange	11.2	LT suplies
105	melon	8.2	Good
101	chenry	3.2	Fastfrui
104	lemon	6.4	FNK
106	mango	15.6	Just

虽然使用 WHERE 条件感觉比较简单明了,但是 WHERE 子句在某些时候会影响查询的性能. 建议内连接时使用 INNER JOIN 规范

```
60  -- 使用INNER JOIN内连接语法
61  SELECT supplies.s_id, f_name, f_price, s_name
62  FROM fruits INNER JOIN supplies
63  ON fruits.s_id = supplies.s_id
64
```

信息 结果 1 剖析 状态

s_id	f_name	f_price	s_name
101	blackberry	10.2	Fastfrui
103	apricot	2.2	ACME
101	blackberry	10.2	Fastfrui
104	berry	7.6	FNK
107	xxxx	3.6	Dk
102	orange	11.2	LT suplies
105	melon	8.2	Good
101	chenry	3.2	Fastfrui
104	lemon	6.4	FNK
106	mango	15.6	Just

连接条件是相等就是等值连接,不相等就是不等连接,如果在连接查询中,两个 表都是同一个表就是自然连接查询, 物理上是同个表,逻辑上分为两个表.

```
65  -- 查询供应a1水果的供应商供应的所有水果
66  SELECT  f1.f_id, f2.s_id, f2.f_name
67  FROM fruits AS f1 , fruits AS f2
68  WHERE f1.s_id = f2.s_id AND f1.f_id = 'a1'
69
70  -- 改为INNER JOIN
71  SELECT  f1.f_id, f2.s_id, f2.f_name
72  FROM fruits AS f1 INNER JOIN fruits AS f2
73  ON f1.s_id = f2.s_id AND f1.f_id = 'a2'
```

信息 结果 1 剖析 状态

f_id	s_id	f_name
a2	103	apricot
a2	103	coconut

9.3.2 外连接查询

在多表连接时,除了返回符合条件 的数据行外,还有可能需要返回左表(左外连接或左连接), 右表(右外连接,右连接)的数据行

LEFT JOIN 左外连接, 返回符合条件的 记录, 和左表所有的记录

RIGTH JOIN 右外连接, 返回右表所有的记录和左表中符合连接条件的记录

```
79  -- 在customers客户表和orders订单表中, 查看所有的客户,包含没有订单的客户
80  SELECT customers.c_id, customers.c_name, orders.o_num
81  FROM customers LEFT JOIN orders
82  ON customers.c_id = orders.c_id
83  除了返回 符合 连接条件 记录外, 还把左表customers中没有订单的记录显示出来了
84
```

信息 结果 1 剖析 状态

c_id	c_name	o_num
10001	RedHook	300001
10001	RedHook	300005
10002	Stars	(Null)
10003	Nethood	300002
10004	Joto	300003
10005	张三	300004
100006	李四	(Null)

```
79  -- 在customers客户表和orders订单表中, 查看所有的客户,包含没有订单的客户
80  SELECT customers.c_id, customers.c_name, orders.o_num
81  FROM orders RIGHT JOIN  customers
82  ON customers.c_id = orders.c_id
83      把符合连接条件的记录 与  右表中不符合条件的记录显示出来
84
```

信息 结果 1 剖析 状态

c_id	c_name	o_num
10001	RedHook	300001
10001	RedHook	300005
10002	Stars	(Null)
10003	Nethood	300002
10004	Joto	300003
10005	张三	300004
100006	李四	(Null)

```
79  -- 在customers客户表和orders订单表中, 查看所有的客户,包含没有订单的客户
80  SELECT customers.c_id, customers.c_name, orders.o_num
81  FROM orders RIGHT JOIN  customers
82  ON customers.c_id = orders.c_id AND o_num IN( 300001, 300002, 300003)
83  ORDER BY c_name                    连接时,还可以添加额外的过滤条件
84                                     也可以对结果进行排序
```

信息 结果 1 剖析 状态

c_id	c_name	o_num
10004	Joto	300003
10003	Nethood	300002
10001	RedHook	300001
10002	Stars	(Null)
10005	张三	(Null)
100006	李四	(Null)

9.4 子查询

虽然可以通过多表连接查询实现查询数据的功能,但是不建议使用连接查询. 因为在连接查询时,先对两个表做笛卡儿积,再筛选符合条件的记录, 所以连接查询性能很差. 使用子查询替代连接查询.

子查询就是在一个查询中嵌套了另外一个查询, 可以在 SELECT 语句 WHERE 或 FROM 子句中包含另外一个 SELECT 查询.外层的 SELECT 称为主查询,WHERE 或 FROM 子句中的 SELECT

查询语句称为子查询.

9.4.1 带有 ANY,SOME 关键字的子查询

ANY 与 SOME 一样 ,表示满足其中任何一个条件 ， 一个表达式与子查询的返回结果进行比较，只要满足任何一个条件即可

```
85  -- 返回价格比104供应商提供任何一个水果价格还高的水果信息
86  SELECT * FROM fruits
87  WHERE   f_price  >
88    ANY ( SELECT f_price FROM fruits WHERE s_id = 104)
89
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
b2	104	berry	7.6
bs1	102	orange	11.2
bs2	105	melon	8.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
o2	103	coconut	9.2
t1	102	banana	10.3

在执行时,先执行子查询
把fruits水果表中s_id为104的价格查询出来
-- 7.6 6.4
再执行主查询
从fruits表中遍历每一条记录,如果这条记录的f_price
大于7.6 或者 6.4 中的一个就检索出来

在执行子查询时,遍历一次fruits表中的所有记录
在执行主查询时, 再遍历一次fruits表中所有的记录
总共只需要遍历两次fruits表中的记录即可
如果使用连接查询， 先做笛卡儿积, 总的记录数是N*N

9.4.2 带有 ALL 的子查询

使用 ALL 时,表示满足所有的子查询的条件

```
85  -- 返回价格比103供应商提供所有水果价格还高的水果信息
86  SELECT * FROM fruits
87  WHERE   f_price  >
88    ALL ( SELECT f_price FROM fruits WHERE s_id = 103)
89
```

信息 | 结果 1 | 剖析 | 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
t1	102	banana	10.3

1) 先执行子查询， 把103供应商的水果价格查询出来
-- 2.2 9.2
如果使用ANY，只要水果价格大于 2.2 或者9.2其中的一个即可，
如果使用ALL， 要求水果价格必须既大于2.2 又大于9.2
ALL要大于所有的103的价格
ANY大于其中一个即可

9.4.3 带 EXISTS 的子查询

WHERE 条件后面是 EXISTS , EXISTS 后面可以是任何一个子查询，如果该子查询的结果至少返回一行,则 EXISTS 的结果就为 TRUE，就执行外层查询； 如果子查询没有返回任何记录,则 EXISTS 结果是 FALSE，不执行外层查询

```
-- 如果supplies供应商表中存在105编号的供应商,就查询fruits表中的记录
SELECT * FROM fruits
WHERE EXISTS
  ( SELECT * FROM supplies WHERE s_id = 105 )
```

信息 结果 1 剖析 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
a2	103	apricot	2.2
b1	101	blackberry	10.2
b2	104	berry	7.6
b5	107	xxxx	3.6
bs1	102	orange	11.2
bs2	105	melon	8.2

先执行子查询，从supplies表中查询s_id编号为105的记录
该子查询有返回记录，存在105的供应商,EXISTS的结果为TRUE
就执行外层 主查询

EXISTS 可以与其他条件一起使用

```
-- 如果supplies供应商表中存在105编号的供应商,就查询fruits表中的记录
SELECT * FROM fruits
WHERE f_price > 10 AND EXISTS
  ( SELECT * FROM supplies WHERE s_id = 105 )
```

信息 结果 1 剖析 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
t1	102	banana	10.3

NOT EXISTS 不存在的结果正好与 EXISTS 存在的结果相反

```
SELECT * FROM fruits
WHERE f_price > 10 AND NOT EXISTS
  ( SELECT * FROM supplies WHERE s_id = 105 )
```

信息 结果 1 剖析 状态

f_id	s_id	f_name	f_price
(Null)	(Null)	(Null)	(Null)

NOT EXISTS与EXISTS结果正好相反，
如果子查询至少返回一行，则NOT EXISTS为false

9.4.4 带有 IN 的子查询

在子查询中返回一个数据列，如果外层查询条件的数据在这个数据列中就表示 TRUE，不在这个数据列中就是 FALSE

```
95  -- 有订单的客户信息
96  SELECT * FROM customers
97  WHERE c_id IN
98  ( SELECT c_id FROM orders )
99
```

先执行子查询,从orders订单表中查询所有的客户编号
结果为10001, 10003, 10004, 10005

再执行主查询,遍历customers客户中所有的记录,如果customers客户表中的c_id在子查询的结果中就把记录检索出来

信息 结果 1 剖析 状态

c_id	c_name	c_address	c_city	c_zip	c_contact	c_email
10001	RedHook	200 street	tianjin	300000	LiMing	liming@163.com
10003	Nethood	400 Street	Qingdao	266000	Loucong	(Null)
10004	Joto	500 Street	Haikou	270000	Yangshan	yangsan@hotmial.com
10005	张三	大族企业湾	北京	100010	zhangxiaosan	(Null)

```
95  -- 查询没有订单的客户信息
96  SELECT * FROM customers
97  WHERE c_id NOT IN
98  ( SELECT c_id FROM orders )
99
```

信息 结果 1 剖析 状态

c_id	c_name	c_address	c_city	c_zip	c_contact	c_email
10002	Stars	300 Street	Dalian	1100000	Zhangbo	zhangbo@qq.com
100006	李四	天通苑	北京	100020	lisisi	(Null)

9.4.5 带有比较运算符的子查询

```
100 -- 查询水果价格比103供应商所有价格还高的水果
101 SELECT * FROM fruits
102 WHERE f_price >
103 ( SELECT MAX(f_price) FROM fruits WHERE s_id = 103)
104
```

1)先执行子查询, 把103供应商所有水果的最高价格查询出来9.2
2)再执行主查询, 再次遍历fruits表,把价格大于9.2的水果信息显示出来

信息 结果 1 剖析 状态

f_id	s_id	f_name	f_price
a1	101	blackberry	10.2
b1	101	blackberry	10.2
bs1	102	orange	11.2
m1	106	mango	15.6
m3	105	xxxtt	11.6
t1	102	banana	10.3

9.5 合并查询结果

利用 UNION 可以把多个 SELECT 查询结果合并为一个结果集, 在合并时,两个表对应的列数,数据类型必须相同,

```
105  SELECT s_id, f_name , f_price
106  FROM fruits
107  WHERE s_id = 101
108  UNION ALL
109  SELECT s_id, f_name, f_price
110  FROM fruits
111  WHERE f_price < 10
112
```

第一个SELECT查询s_id为101的水果信息
第二个SELECT查询价格小于10的水果信息
使用UNION可以把这两个 结果集合并为一个，
会去掉重复的记录

使用UNION ALL在合并时,会保留重复的记录

信息 | 结果 1 | 剖析 | 状态

s_id	f_name	f_price
101	blackberry	10.2
101	blackberry	10.2
101	chenry	3.2
103	apricot	2.2
104	berry	7.6
107	xxxx	3.6
105	melon	8.2
101	chenry	3.2

SELECT s_id, f_name , f_price FROM fruits WHERE s_id = 101 UNION ALL | 只读 | 查询时间: 0.027s | 第 1 条记录 (共 13 条)

9.6 为表或字段起别名

可以使用 AS 关键字给表或者字段起别名

9.6.1 为表起别名

```
1  -- 查询客户id, 客户名称, 订单编号,
   在两个表中都有c_id字段,在使用c_id字段时需要使用表名作为前缀进行区分
2  -- 为了在多次使用相同的表时,为了使用更加的方便,可以为表指定别名
3  SELECT c.c_id, c.c_name, o.o_num
4  FROM customers AS c JOIN orders AS o
5  ON c.c_id = o.c_id
```

为表指定别名

信息 | 结果 1 | 剖析 | 状态

c_id	c_name	o_num
10001	RedHook	300001
10003	Nethood	300002
10004	Joto	300003
10005	张三	300004
10001	RedHook	300005

9.6.2 为字段起别名

在 SELECT 查询显示结果时，显示的列名可能不够直观,或者列名太长，可以指定列名的别名.

```
7  -- 当显示的列名太长,或者列名不直观时,可以给字段起别名
8  SELECT c_id AS 客户编号, c_name AS 客户名称
9  FROM customers
```

信息 | 结果 1 | 剖析 | 状态

客户编号	客户名称
10001	RedHook
10002	Stars
10003	Nethood
10004	Joto
10005	张三
100006	李四

第10章　索引

索引用于快速找出某个列中的有特定值的行.如果不使用索引,MySQL 是从第 1 条记录开始读完整个表,直到找到相关的行. 表的记录行越大,查询花费的时间就越大. 如果表中的查询列有一个索引, MySQL 可以快速定位到某个位置去查询,不需要查看所有的数据.索引就类似于书的目录.索引是对数据表中一列或多列值进行排序的一种结构, 使用索引可以提高数据库中特定数据的查询速度.

MySQL 中每个存储引擎的索引都不一定完全相同,MySQL 中索引的存储类型有两种:BTREE 和 HASH, InnoDB 与 MyISAM 存储引擎支持 BTREE 索引, MEMORY 存储引擎支持 HASH 和 BTREE 索引.

索引可以提高数据的查询速度,但是创建索引,查询索引也需要消耗时间, 对表中的数据进行添加,删除,修改的同时也需要维护索引,也会降低维护速度.索引设计不合理会影响程序的性能,索引一般的设计原则:

1) 索引并不是越多越好,大量的索引会影响 INSERT,UPDATE,DELETE 语句的性能
2) 避免对经常更新的表进行过多的索引, 并且索引的列尽可能少.
3) **对经常查询的列创建索引,**即经常用于 WHERE 查询的列
4) 数量量小的表最好不使用索引
5) 在频繁进行排序或分组的列上创建索引

10.1　创建索引

10.1.1 在创建表时创建索引

在 CREATE TABLE 创建表时,可以定义列的主键约束,外键约束,唯一约束, 不管创建哪种约束,在定义约束的同时就相当于在指定的列上创建了一个索引

1 创建普通的索引

```
CREATE TABLE book(
    id    INT PRIMARY KEY auto_increment,
```

```
    name VARCHAR(50),
    author VARCHAR(50),
    price    FLOAT,
    publiccation_year YEAR,
    INDEX( publiccation_year )
);
```

```
22
23  EXPLAIN SELECT * FROM book WHERE publiccation_year=1990
24
```

信息 | 结果 1 | 剖析 | 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	book	ref	publiccation_year	publiccation_year	2	const	1	Using where

SIMPLLE表示简单查询
possible_keys 表示可选 的索引
key表示实际使用的索引
key_len索引长度越小,表示越快

2 创建唯一索引

```
CREATE TABLE book2(
    id    INT PRIMARY KEY auto_increment,
    name VARCHAR(50),
    author VARCHAR(50),
    price    FLOAT,
    publiccation_year YEAR,
    isbn VARCHAR(50),
    UNIQUE INDEX( isbn )
);
```

10.1.2 在已存在的表上创建索引

使用 ALTER TABEL 或者 CREATE INDEX 在已存在的表中创建索引

```
37  -- 查看book表中已创建的索引
38  SHOW INDEX FROM book
39
```

信息 | 结果 1 | 剖析 | 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
book	0	PRIMARY	1	id	A	0	(Null)	(Null)		BTREE
book	1	publiccation_year	1	publiccation_year	A	0	(Null)	(Null)	YES	BTREE

```
40  -- 在书名列中添加唯一索引
41  ALTER TABLE book ADD UNIQUE INDEX bknameIdex( name );
42  -- 查看book表中已创建的索引
43  SHOW INDEX FROM book
```

信息 | 结果 1 | 剖析 | 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
book	0	PRIMARY	1	id	A	0	(Null)	(Null)		BTREE
book	0	bknameIdex	1	name	A	0	(Null)	(Null)	YES	BTREE
book	1	publiccation_year	1	publiccation_year	A	0	(Null)	(Null)	YES	BTREE

```
44
45  CREATE INDEX bkauthorIndex ON book ( author)
46  -- 查看book表中已创建的索引
47  SHOW INDEX FROM book
48
```

信息 | 结果 1 | 剖析 | 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
book	0	PRIMARY	1	id	A	0	(Null)	(Null)		BTREE
book	0	bknameIdex	1	name	A	0	(Null)	(Null)	YES	BTREE
book	1	publiccation_year	1	publiccation_year	A	0	(Null)	(Null)	YES	BTREE
book	1	bkauthorIndex	1	author	A	0	(Null)	(Null)	YES	BTREE

10.2 删除索引

```
49  -- 删除索引
50  ALTER TABLE book DROP INDEX bknameIdex
51
52  DROP INDEX bkauthorIndex ON book
53
54  -- 查看book表中已创建的索引
55  SHOW INDEX FROM book
56
57
```

信息 | 结果 1 | 剖析 | 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
book	0	PRIMARY	1	id	A	0	(Null)	(Null)		BTREE
book	1	publiccation_year	1	publiccation_year	A	0	(Null)	(Null)	YES	BTREE

第11章 视图

为了提高复杂SQL语句的复用性和表操作的安全性，可以使用视图. 视图是一种虚拟表，不存储数据，视图就是一组 SQL 语句. 表是存储具体的数据的,视图看作是表的窗口,是查看表中内容的一种方法

11.1 创建视图

```
56
57  -- 在单表中创建视图
58  CREATE VIEW view_fruits( 水果名称,水果价格)
59  AS  SELECT f_name, f_price  FROM fruits;
60
61  -- 在多表上创建视图
62  CREATE VIEW view_fruits2( 水果名称,供应商名称,水果价格)
63  AS
64  SELECT f_name, s_name , f_price
65  FROM fruits INNER JOIN supplies
66  ON fruits.s_id = supplies.s_id
67
```

11.2 查看视图

```
67
68  -- 查看视图的结构
69  DESC view_fruits
70
```

信息 结果 1 剖析 状态

Field	Type	Null	Key	Default	Extra
水果名称	varchar(255)	NO		(Null)	
水果价格	decimal(8,2)	NO		(Null)	

```
71  -- 查看视图的信息
72  SHOW TABLE STATUS LIKE 'view_f%'
73
```

信息 结果 1 剖析 状态

Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length	Max_data_length	Index_length	Da
view_fruits	(Null)	(Null)	(Null)		(Null)	(Null)	(Null)	(Null)	
view_fruits2	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	

```
73
74  -- 在views视图表中查看视图的信息
75  SELECT * FROM information_schema.VIEWS
76
77
```

信息 结果 1 剖析 状态

TABLE_CATALO	TABLE_SCHEM	TABLE_NAME	VIEW_DEFINITION	CHECK_OPTION	IS_UPDATABLE	DEFINER
def	fruits	view_fruits	select `fruits`.`fruits`.`f_name` AS `水果	NONE	YES	root@localhost
def	fruits	view_fruits2	select `fruits`.`fruits`.`f_name` AS `水果	NONE	YES	root@localhost

11.3 修改视图

-- 如果视图不存在就创建,如果视图已存在使用新的 SQL 语句替换原来的 SQL 语句

CREATE OR REPLACE VIEW view_f AS SELECT * from customers

-- 使用 ALTER View 修改视图

ALTER VIEW view_f AS SELECT * from fruits;

11.4 更新视图

通过视图向基表中插入,更新,删除数据. 视图就是一组SQL语句,没有具体数据,通过视图更新的基表中的数据

UPDATE view_f set f_price = f_price*1.1 WHERE s_id = 101

并不是所有的视图都可以更新基表数据,如:

1) 视图中没有基表的非空字段,插入失败,可以更新,删除

```
87  INSERT INTO view_f (f_id, f_name,f_price) VALUES('hh','xigua',8.6)
88  -- 向视图中插入记录就是相当于向基表中插入记录,
    s_id不允许为NULL,还没有默认值,在插入记录时必须给s_id插入数据
89  INSERT INTO fruits (f_id, f_name,f_price) VALUES('hh','xigua',8.6)
```

信息 状态

```
INSERT INTO fruits (f_id, f_name,f_price) VALUES('hh','xigua',8.6)
> 1364 - Field 's_id' doesn't have a default value
> 时间: 0.001s
```

2)在定义视图时使用了数学表达式或者聚合函数,插入失败

```
91  CREATE OR REPLACE  VIEW view_f (供应商编号, 平均价格)  AS
92  SELECT s_id, AVG(f_price)
93  FROM fruits
94  GROUP BY s_id;
95  SELECT * FROM view_f
96  INSERT INTO view_f(供应商编号, 平均价格) VALUES(109,6.6)
```

信息 状态

```
INSERT INTO view_f(供应商编号, 平均价格) VALUES(109,6.6)
> 1471 - The target table view_f of the INSERT is not insertable-into
> 时间: 0.004s
```

11.5 删除视图

DROP VIEW view_f

第12章　事务

12.1　事务概述

当多个用户同时请求修改某个数据时,通过事务可以保证数据从一个一致性状态变更为另外一个一致性的状态，如 A 用户给 B 用户转 10000 元，需要在 A 用户帐户中减少 10000 元,在 B 用户的帐户中增加 10000 元,这两个操作应该是一个原子操作,要么两个都修改成功,要么这两个操作都没发生.

事务具有四个 ACID 特性:

1)原子性(Atomicity)把事务中所有的操作看作是一个整体,事务对数据的修改要么完全提交要么回滚

2)一致性(Consistency)在事务完成时,必须使所有的数据从一个一致性状态变更为另外一个一致性状态

3)隔离性(Isolation)一个事务中的操作语句对数据所做的修改必须与其他事务所做的修改相隔离.

4)持久性(Durability)在事务完成后,所做的修改是永久的

注意,在 MySQL 数据库中的 InnoDB 存储引擎支持事务, MyISAM 不支持事务

12.2　事务的控制语句

```
START TRANSACTION   | BEGIN        -- 开始事务
COMMIT              -- 提交事务
ROLLBACK            -- 回滚事务
```

12.3　事务隔离级别

在 MySQL 中有四种隔离级别,指定了事务中哪些数据改变对其他事务是可见的,哪些数据改变对其他事务是不可变的. 低级别可以支持更高的并发处理,,占用系统资源少.设置事务隔离级别的语句是:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL   READ UNCOMMITTED; --读未提交
SET GLOBAL TRANSACTION ISOLATION LEVEL   READ   COMMITTED; --读已提交
SET GLOBAL TRANSACTION ISOLATION LEVEL   REPEATABLE   READ ; --  可重复读
SET GLOBAL TRANSACTION ISOLATION LEVEL   SERIALIZABLE; --可串行化
```

12.3.1 读未提交 READ UNCOMMITTED

在读未提交隔离级别中,所有事务可以看到其他事务未提交的执行结果,这种现象称为脏读.

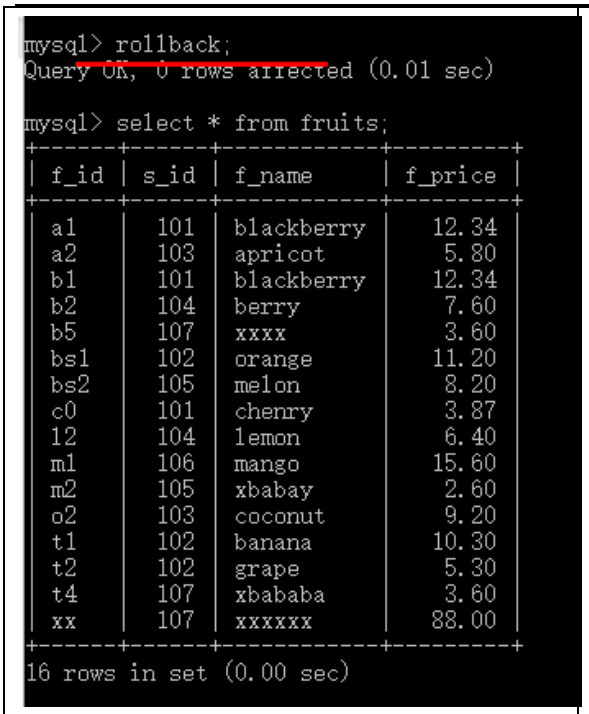
设备事务的隔离级别为读未提交 set global transaction isolation level read uncommitted;		
重新打开命令窗口，查看当前事务的隔离级别 mysql> show variables like 'tx_isolation'; +---------------+------------------+ | Variable_name | Value | +---------------+------------------+ | tx_isolation | READ-UNCOMMITTED | +---------------+------------------+ 1 row in set (0.00 sec)		
mysql> use fruits; Database changed mysql>	mysql> use fruits; Database changed mysql>	
mysql> select * from fruits; +------+------+------------+---------+ | f_id | s_id | f_name | f_price | +------+------+------------+---------+ | a1 | 101 | blackberry | 12.34 | | a2 | 103 | apricot | 5.80 | | b1 | 101 | blackberry | 12.34 | | b2 | 104 | berry | 7.60 | | b5 | 107 | xxxx | 3.60 | | bs1 | 102 | orange | 11.20 | | bs2 | 105 | melon | 8.20 | | c0 | 101 | chenry | 3.87 | | 12 | 104 | lemon | 6.40 | | m1 | 106 | mango | 15.60 | | m2 | 105 | xbabay | 2.60 | | o2 | 103 | coconut | 9.20 | | t1 | 102 | banana | 10.30 | | t2 | 102 | grape | 5.30 | | t4 | 107 | xbababa | 3.60 | | xx | 107 | xxxxxx | 88.00 | +------+------+------------+---------+ 16 rows in set (0.00 sec)	mysql> select * from fruits; +------+------+------------+---------+ | f_id | s_id | f_name | f_price | +------+------+------------+---------+ | a1 | 101 | blackberry | 12.34 | | a2 | 103 | apricot | 5.80 | | b1 | 101 | blackberry | 12.34 | | b2 | 104 | berry | 7.60 | | b5 | 107 | xxxx | 3.60 | | bs1 | 102 | orange | 11.20 | | bs2 | 105 | melon | 8.20 | | c0 | 101 | chenry | 3.87 | | 12 | 104 | lemon | 6.40 | | m1 | 106 | mango | 15.60 | | m2 | 105 | xbabay | 2.60 | | o2 | 103 | coconut | 9.20 | | t1 | 102 | banana | 10.30 | | t2 | 102 | grape | 5.30 | | t4 | 107 | xbababa | 3.60 | | xx | 107 | xxxxxx | 88.00 | +------+------+------------+---------+ 16 rows in set (0.00 sec)	
删除数据		

```
mysql> delete from fruits where f_id='xx';
Query OK, 1 row affected (0.00 sec)

mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | cherry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
+------+------+------------+---------+
15 rows in set (0.00 sec)
```

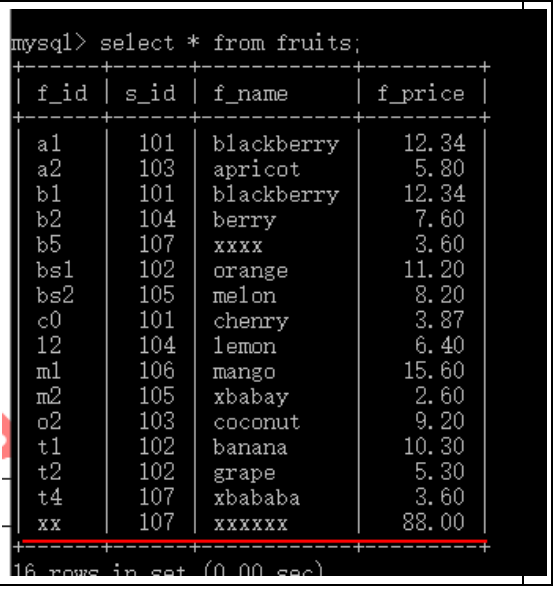
```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | cherry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
+------+------+------------+---------+
15 rows in set (0.00 sec)
```



```
mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
+------+------+------------+---------+
16 rows in set (0.00 sec)
```



```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
+------+------+------------+---------+
16 rows in set (0.00 sec)
```

如果设置隔离级别为读未提交，在 A 事务提交之前，B 事务可以看到 A 事务的操作的中间结果，如果 A 事务回滚，B 事务读到的数据就不是准备的，就是脏读

12.3.2 读已提交

大多数数据库管理系统默认的隔离级别.但是不是 MySQL 默认的隔离级别. 一个事务从开始到提交前所做的修改都其他事务来说是不可见的，事务只能看见其他事务已提交的改变

事务 A	事务 B

```
mysql> show variables like 'tx_isolation';
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tx_isolation  | READ-COMMITTED |
+---------------+----------------+
1 row in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> use fruits;
Database changed
```

```
mysql> show variables like 'tx_isolation';
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tx_isolation  | READ-COMMITTED |
+---------------+----------------+
1 row in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> use fruits;
Database changed
```

```
mysql> insert into fruits values('yy',107,'yyyy',66);
Query OK, 1 row affected (0.00 sec)

mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
| yy   |  107 | yyyy       |   66.00 |
+------+------+------------+---------+
```

```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
+------+------+------------+---------+
16 rows in set (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

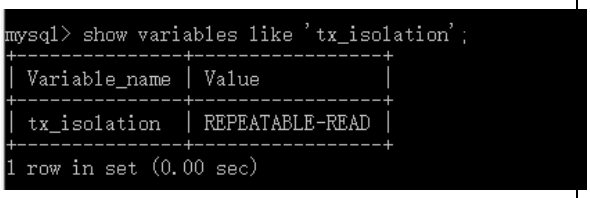
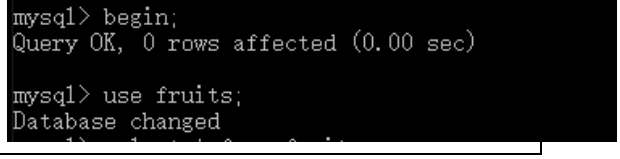
```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| l2   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
| yy   |  107 | yyyy       |   66.00 |
+------+------+------------+---------+
17 rows in set (0.00 sec)
```

开启事务 A 与事务 B，对于事务 A 未提交的更新,在事务 B 中无法读取的，当事务 A 已提交后,在事务 B 中可以读取到事务 A 的修改. 存在的问题：在事务 B 中，同一个 Select 检索语句,查出来了不同的结果,称为不可重复读现象. 即不能重复读,读多次的话可能会出现不一样的结果,即两次查询结果不一样

12.3.3 可重复读

REPEATABLE READ 这是 MySQL 默认的事务隔离级别，可以确保一个事务不管什么时候读取数据,检索出同样的数据.在理论上可能会导致幻读现象，在一个事务的两次查询中数据的记录不一致. 如在事务 B 中对表中所有的记录行进行了修改，在事务 A 中插入了新的数据行. 等事务 B 再次查询数据时,可能会发现多了一行没有修改的数据.

在 InnoDB 存储引擎中采用了 MVCC(Multi_Version Consistency Conntrol,多版本并发控制)机制解决了幻读问题

事务 A	事务 B
	
	

```
mysql> show variables like 'tx_isolation';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> use fruits;
Database changed
```

```
mysql> use fruits;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from fruits where s_id = 106;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | 1emon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
| yy   |  107 | yyyy       |   66.00 |
+------+------+------------+---------+
17 rows in set (0.00 sec)
```

```
mysql> delete from fruits where s_id = 106;
Query OK, 1 row affected (0.00 sec)

mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| 12   |  104 | 1emon      |    6.40 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
| yy   |  107 | yyyy       |   66.00 |
+------+------+------------+---------+
16 rows in set (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

<table>
<tr><td></td><td>

```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   12.34 |
| a2   |  103 | apricot    |    5.80 |
| b1   |  101 | blackberry |   12.34 |
| b2   |  104 | berry      |    7.60 |
| b5   |  107 | xxxx       |    3.60 |
| bs1  |  102 | orange     |   11.20 |
| bs2  |  105 | melon      |    8.20 |
| c0   |  101 | chenry     |    3.87 |
| l2   |  104 | lemon      |    6.40 |
| m1   |  106 | mango      |   15.60 |
| m2   |  105 | xbabay     |    2.60 |
| o2   |  103 | coconut    |    9.20 |
| t1   |  102 | banana     |   10.30 |
| t2   |  102 | grape      |    5.30 |
| t4   |  107 | xbababa    |    3.60 |
| xx   |  107 | xxxxxx     |   88.00 |
| yy   |  107 | yyyy       |   66.00 |
+------+------+------------+---------+
17 rows in set (0.00 sec)
```

</td></tr>
<tr><td></td><td>

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

</td></tr>
<tr><td></td><td></td></tr>
<tr><td>

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into fruits values ('zz',105,'zzzzz',2.5);
Query OK, 1 row affected (27.17 sec)
```

</td><td></td></tr>
<tr><td></td><td>

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
```

</td></tr>
<tr><td></td><td>

```
mysql> update fruits set f_price = f_price + 10;
Query OK, 16 rows affected (0.00 sec)
Rows matched: 16  Changed: 16  Warnings: 0
```

</td></tr>
</table>

```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   22.34 |
| a2   |  103 | apricot    |   15.80 |
| b1   |  101 | blackberry |   22.34 |
| b2   |  104 | berry      |   17.60 |
| b5   |  107 | xxxx       |   13.60 |
| bs1  |  102 | orange     |   21.20 |
| bs2  |  105 | melon      |   18.20 |
| c0   |  101 | chenry     |   13.87 |
| 12   |  104 | lemon      |   16.40 |
| m2   |  105 | xbabay     |   12.60 |
| o2   |  103 | coconut    |   19.20 |
| t1   |  102 | banana     |   20.30 |
| t2   |  102 | grape      |   15.30 |
| t4   |  107 | xbababa    |   13.60 |
| xx   |  107 | xxxxxx     |   98.00 |
| yy   |  107 | yyyy       |   76.00 |
+------+------+------------+---------+
16 rows in set (0.00 sec)

mysql> commit;
```

```
mysql> insert into fruits values ('zz',105,'zzzzz',2.5);
Query OK, 1 row affected (27.17 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from fruits;
+------+------+------------+---------+
| f_id | s_id | f_name     | f_price |
+------+------+------------+---------+
| a1   |  101 | blackberry |   22.34 |
| a2   |  103 | apricot    |   15.80 |
| b1   |  101 | blackberry |   22.34 |
| b2   |  104 | berry      |   17.60 |
| b5   |  107 | xxxx       |   13.60 |
| bs1  |  102 | orange     |   21.20 |
| bs2  |  105 | melon      |   18.20 |
| c0   |  101 | chenry     |   13.87 |
| 12   |  104 | lemon      |   16.40 |
| m2   |  105 | xbabay     |   12.60 |
| o2   |  103 | coconut    |   19.20 |
| t1   |  102 | banana     |   20.30 |
| t2   |  102 | grape      |   15.30 |
| t4   |  107 | xbababa    |   13.60 |
| xx   |  107 | xxxxxx     |   98.00 |
| yy   |  107 | yyyy       |   76.00 |
| zz   |  105 | zzzzz      |    2.50 |
+------+------+------------+---------+
17 rows in set (0.00 sec)
```

12.3.4 串行化

强制事务串行操作，事务在读取数据行上加锁，各个事务之间不可能有冲突，不建议使用,可能会导致大量的超时和锁竞争

第13章　存储过程与函数

存储过程和函数可以简单的理解为一条或多条 SQL 语句的集合,是经过预先编译并存储在数据库中的对象. 存储过程没有返回值,函数有返回值

13.1　创建存储过程

CREATE PROCEDURE　存储过程名 (参数列表)
　　BEGIIN
　　　　SQL 语句
　　END;

```
104  -- 创建简单的存储过程,没有参数
105  CREATE PROCEDURE proc_listFruits ()
106  BEGIN
107    SELECT * FROM fruits;
108  END
109
110  -- 调用存储过程
111  CALL proc_listFruits();
```

信息　结果 1　剖析　状态

f_id	s_id	f_name	f_price
a1	101	blackberry	22.34
a2	103	apricot	15.8
b1	101	blackberry	22.34
b2	104	berry	17.6
b5	107	xxxx	13.6
bs1	102	orange	21.2
bs2	105	melon	18.2

```
113  -- 创建有输入输出参数的存储过程, 根据水果名称查询水果价格
114  CREATE PROCEDURE proc_getPrice1(name VARCHAR(30), OUT price DECIMAL(8,2) )
115  BEGIN
116    SELECT f_price INTO price FROM fruits WHERE f_name = name;
117    END;
118  -- 调用有参数的存储过程
119  CALL proc_getPrice1('banana', @price);
120  -- 查看变量的值
121  SELECT @price;
122
```

信息　结果 1　剖析　状态

@price
20.3

13.2 创建函数

```
-- 创建函数,返回指定水果名称的价格
CREATE FUNCTION func_getPrice( name VARCHAR(30))
RETURNS  DECIMAL(8,2)    指定返回值类型
BEGIN
  RETURN ( SELECT f_price FROM fruits WHERE f_name = name );
END    返回值

-- 调用函数,查看结构
SELECT func_getPrice('banana');
```

信息 | 结果 1 | 剖析 | 状态

func_getPrice('banana')
20.3

第14章 用户管理

14.1 权限表

在 MySQL 数据库中,存储权限信息的表有: user, db, host, table_priv, columns_priv 等

1) user 表

存储登录服务器的用户信息

```
SELECT * FROM mysql.user;
```

信息 | 结果 1 | 剖析 | 状态

Host	User	Password	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv
localhost	root	*23AE809DDACAF96AF0FD	Y	Y	Y	Y	Y

2)db 数据库表与 host 主机表

db 表中存储了用户对某个数据库的操作权限，查看哪个用户可以访问哪个数据库

host 表存储了某个主机对数据库的操作权限

3)table_priv 表权限

tables_priv 设置表的权限, columns_priv 设置列的权限

14.2 账户管理

14.2.1 新建普通的账户

（1） 使用 CREATE USER 创建用户

```
-- 创建用户 CREATE USER 用户名@主机名 IDENTIFIED BY 密码
CREATE USER 'cui'@'localhost' IDENTIFIED BY '123'
-- 刚 CREATE USER 创建的用户没有任何权限，需要使用 GRANT 语句进行授权
```

（2） 使用 GRANT 创建用户

```
-- 直接使用GRANT创建用户
-- GRANT 权限 ON 库.表 TO 用户名@主机名 IDENTIFIED BY 密码
-- 在授权时,如果账户不存在就先创建一个账户
GRANT SELECT,UPDATE,DELETE ON fruits.* TO 'feifei'@'localhost' IDENTIFIED BY '123'

SELECT * from db;
```

信息 | 结果 1 | 剖析 | 状态

Host	Db	User	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	Grant_priv
localhost	fruits	feifei	Y	N	Y	Y	N	N	N

（3） 直接操作 MySQL 的 User 表

不管使用 CREATE USER 还是 GRANT 创建用户都会在 user 表中添加一条记录，也可以直接向 user 表添加记录的形式创建用户

```
-- 直接操作user表添加用户
INSERT INTO `user` (`host`, `user`,`Password`,ssl_cipher,x509_issuer,x509_subject) VALUES(
'localhost', 'bjpowernode', password('123') , '','','');
SELECT * FROM user;
```

信息 | 结果 1 | 剖析 | 状态

Host	User	Password	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	R
localhost	root	*23AE809DDACAF96AF0FD	Y	Y	Y	Y	Y	Y	Y
localhost	cui	*23AE809DDACAF96AF0FD	N	N	N	N	N	N	N
localhost	feifei	*23AE809DDACAF96AF0FD	N	N	N	N	N	N	N
localhost	bjpowern	*23AE809DDACAF96AF0FD	N	N	N	N	N	N	N

14.2.2 删除用户

```
DROP USER 'bjpowernode'@'localhost'
```

或

DELETE FROM mysql.`user` WHERE Host = 'host' AND User='bjpowernode'

14.2.3 修改普通用户密码

```
-- 修改密码
UPDATE mysql.`user` SET Password = password('456')
WHERE Host = 'localhost' AND User='feifei';
-- 刷新权限
FLUSH PRIVILEGES;
```

14.3 权限管理

对于登录数据库服务器的用户授予不同的权限,授予权限使用 GRANT，使用 REVOKE 可以收回权限

```
159
160  -- 收回feifei用户的Delete权限
161  REVOKE DELETE ON fruits.*  FROM 'feifei'@'localhost';
162  SELECT * FROM mysql.db;
```

信息 | 结果 1 | 剖析 | 状态

Host	Db	User	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	D
localhost	fruits	feifei	Y	N	Y	N	N	N

第15章 数据备份与恢复

保证数据安全的最重要的一个措施就是进行定期备份，一旦数据丢失或出现错误，可以使用备份的数据进行恢复 ，尽可能降低意外导致的损失.

15.1 备份

```
mysqldump -uroot -p123    fruits > d:/fruits.sql
```

15.2 恢复

登录 MySQL
创建 fruits 数据库

选择 fruits 数据库

source d:/fruits.sql

第16章　JDBC

16.1　JDBC 概述

在模块中添加 mysql 的 JDBC 的实现类