# 简介

SQLite是一个进程内的库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。一个零配置的数据库，不需要在系统中配置。

SQLite 引擎不是一个独立的进程，可以按应用程序需求进行静态或动态连接。SQLite 直接访问其存储文件。

官网：http://www.sqlite.org/index.html

# 特点

1. 不需要一个单独的服务器进程或操作的系统（无服务器的）。
2. SQLite 不需要配置，这意味着不需要安装或管理。
3. 一个完整的 SQLite 数据库是存储在一个单一的跨平台的磁盘文件。
4. SQLite 是非常小的，是轻量级的，完全配置时小于 400KiB，省略可选功能配置时小于250KiB。
5. SQLite 是自给自足的，这意味着不需要任何外部的依赖。
6. SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问。
7. SQLite 支持 SQL92（SQL2）标准的大多数查询语言的功能。
8. SQLite 使用 ANSI-C 编写的，并提供了简单和易于使用的 API。
9. SQLite 可在 UNIX（Linux, Mac OS-X, Android, iOS）和 Windows（Win32, WinCE, WinRT）中运行。

# 局限

在SQLite中，SQL92不支持的特性如下所示：

特性 描述

RIGHT OUTER JOIN 只实现了 LEFT OUTER JOIN。

FULL OUTER JOIN 只实现了 LEFT OUTER JOIN。

ALTER TABLE 支持 RENAME TABLE 和 ALTER TABLE 的 ADD COLUMN variants 命 令，不支持 DROP COLUMN、ALTER COLUMN、ADD CONSTRAINT。

Trigger 支持 支持 FOR EACH ROW 触发器，但不支持 FOR EACH STATEMENT 触发 器。

VIEWs 在SQLite中，视图是只读的。不可以在视图上执行DELETE、INSERT 或 UPDATE 语句。

GRANT和REVOKE 可以应用的唯一的访问权限是底层操作系统的正常文件访问权限。

# 命令分类

与关系数据库进行交互的标准SQLite命令类似于 SQL。命令包括 CREATE、SELECT、INSERT、UPDATE、DELETE 和 DROP。这些命令基于它们的操作性质可分为以下几种。

## DDL - 数据定义语言

命令 描述

CREATE 创建一个新的表，一个表的视图，或者数据库中的其他对象。

ALTER 修改数据库中的某个已有的数据库对象，比如一个表。

DROP 删除整个表，或者表的视图，或者数据库中的其他对象。

## DML - 数据操作语言

命令 描述

INSERT 创建一条记录。

UPDATE 修改记录。

DELETE 删除记录。

## DQL - 数据查询语言

命令 描述

SELECT 从一个或多个表中检索某些记录。

# 安装

目前，几乎所有版本的Linux操作系统都附带SQLite。所以，只要使用下面的命令来检查您的机器上是否已经安装了 SQLite。

rpm -qa sqlite

$sqlite3

SQLite version 3.7.15.2 2013-01-09 11:53:05

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

sqlite>

如果没有看到上面的结果，那么就意味着没有在 Linux 机器上安装 SQLite。按照下面的步骤安装 SQLite：

访问 SQLite 下载页面，从源代码区下载 sqlite-autoconf-\*.tar.gz。

步骤如下：

$tar xvfz sqlite-autoconf-3071502.tar.gz

$cd sqlite-autoconf-3071502

$./configure --prefix=/usr/local

$make

$make install

# 命令

点命令的不同之处在于不以分号 ; 结束。获取可用的点命令的清单，可以在任何时候输入 ".help"。例如：

sqlite>.help

上面的命令会显示各种重要的 SQLite 点命令的列表，如下所示：

命令 描述

.backup ?DB? FILE 备份 DB 数据库（默认是 "main"）到 FILE 文件。

.bail ON|OFF 发生错误后停止。默认为 OFF。

.databases 列出附加数据库的名称和文件。

.dump ?TABLE? 以 SQL 文本格式转储数据库。如果指定了 TABLE 表，则只转储匹配 LIKE 模式的 TABLE 表。

.echo ON|OFF 开启或关闭 echo 命令。

.exit 退出 SQLite 提示符。

.explain ON|OFF 开启或关闭适合于 EXPLAIN 的输出模式。如果没有带参数，则为 EXPLAIN on，及开启 EXPLAIN。

.header(s) ON|OFF 开启或关闭头部显示。

.help 显示消息。

.import FILE TABLE 导入来自 FILE 文件的数据到 TABLE 表中。

.indices ?TABLE? 显示所有索引的名称。如果指定了 TABLE 表，则只显示匹配 LIKE 模 式的 TABLE 表的索引。

.load FILE ?ENTRY? 加载一个扩展库。

.log FILE|off 开启或关闭日志。FILE 文件可以是 stderr（标准错误）/stdout（标准 输出）。

.mode MODE 设置输出模式，MODE 可以是下列之一：

csv 逗号分隔的值

column 左对齐的列

html HTML 的 <table> 代码

insert TABLE 表的 SQL 插入（insert）语句

line 每行一个值

list 由 .separator 字符串分隔的值

tabs 由 Tab 分隔的值

tcl TCL 列表元素

.nullvalue STRING 在 NULL 值的地方输出 STRING 字符串。

.output FILENAME 发送输出到 FILENAME 文件。

.output stdout 发送输出到屏幕。

.print STRING... 逐字地输出 STRING 字符串。

.prompt MAIN CONTINUE 替换标准提示符。

.quit 退出 SQLite 提示符。

.read FILENAME 执行 FILENAME 文件中的 SQL。

.schema ?TABLE? 显示 CREATE 语句。如果指定了 TABLE 表，则只显示匹配 LIKE 模 式的 TABLE 表。

.separator STRING 改变输出模式和 .import 所使用的分隔符。

.show 显示各种设置的当前值。

.stats ON|OFF 开启或关闭统计。

.tables ?PATTERN? 列出匹配 LIKE 模式的表的名称。

.timeout MS 尝试打开锁定的表 MS 毫秒。

.width NUM NUM 为 "column" 模式设置列宽度。

.timer ON|OFF 开启或关闭 CPU 定时器。

## 当前配置

使用 .show 命令，来查看 SQLite 命令提示符的默认设置。确保 sqlite> 提示符与点命令之间没有空格，否则将无法正常工作。

sqlite>.show

echo: off

explain: off

headers: off

mode: column

nullvalue: ""

output: stdout

separator: "|"

width:

sqlite>

## 格式化输出

使用下列的点命令来格式化输出为下面所列出的格式。

sqlite>.header on

sqlite>.mode column

sqlite>.timer on

sqlite>

上面设置将产生如下格式的输出：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

CPU Time: user 0.000000 sys 0.000000

## sqlite\_master表格

主表中保存数据库表的关键信息，并把它命名为 sqlite\_master。如要查看表概要，可按如下操作：

sqlite>.schema sqlite\_master

CREATE TABLE sqlite\_master (

type text,

name text,

tbl\_name text,

rootpage integer,

sql text

);

# 语法

## 大小写敏感性

有个重要的点值得注意，SQLite 是不区分大小写的，但也有一些命令是大小写敏感的，比如 GLOB 和 glob 在 SQLite 的语句中有不同的含义。

## 注释

SQLite注释是附加的注释，可以在SQLite代码中添加注释以增加其可读性，他们可以出现在任何空白处，包括在表达式内和其他SQL语句的中间，但它们不能嵌套。

SQL注释以两个连续的 "-" 字符（ASCII 0x2d）开始，并扩展至下一个换行符（ASCII 0x0a）或直到输入结束，以先到者为准。

也可以使用 C 风格的注释，以 "/\*" 开始，并扩展至下一个 "\*/" 字符对或直到输入结束，以先到者为准。SQLite的注释可以跨越多行。

sqlite>.help -- This is a single line comment

## 语句

所有的 SQLite 语句可以以任何关键字开始，如 SELECT、INSERT、UPDATE、DELETE、ALTER、DROP 等，所有的语句以分号（;）结束。

### ANALYZE

ANALYZE;

or

ANALYZE database\_name;

or

ANALYZE database\_name.table\_name;

### AND/OR

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

### ALTER TABLE

ALTER TABLE table\_name ADD COLUMN column\_def...;

ALTER TABLE table\_name RENAME TO new\_table\_name;

### ATTACH DATABASE

ATTACH DATABASE 'DatabaseName' As 'Alias-Name';

### BEGIN TRANSACTION

BEGIN;

or

BEGIN EXCLUSIVE TRANSACTION;

### BETWEEN

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

### COMMIT

COMMIT;

### CREATE INDEX

CREATE INDEX index\_name

ON table\_name ( column\_name COLLATE NOCASE );

SQLite CREATE UNIQUE INDEX 语句：

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

SQLite CREATE TABLE 语句：

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

SQLite CREATE TRIGGER 语句：

CREATE TRIGGER database\_name.trigger\_name

BEFORE INSERT ON table\_name FOR EACH ROW

BEGIN

stmt1;

stmt2;

....

END;

SQLite CREATE VIEW 语句：

CREATE VIEW database\_name.view\_name AS

SELECT statement....;

SQLite CREATE VIRTUAL TABLE 语句：

CREATE VIRTUAL TABLE database\_name.table\_name USING weblog( access.log );

or

CREATE VIRTUAL TABLE database\_name.table\_name USING fts3( );

SQLite COMMIT TRANSACTION 语句：

COMMIT;

SQLite COUNT 子句：

SELECT COUNT(column\_name)

FROM table\_name

WHERE CONDITION;

SQLite DELETE 语句：

DELETE FROM table\_name

WHERE {CONDITION};

SQLite DETACH DATABASE 语句：

DETACH DATABASE 'Alias-Name';

SQLite DISTINCT 子句：

SELECT DISTINCT column1, column2....columnN

FROM table\_name;

SQLite DROP INDEX 语句：

DROP INDEX database\_name.index\_name;

SQLite DROP TABLE 语句：

DROP TABLE database\_name.table\_name;

SQLite DROP VIEW 语句：

DROP VIEW view\_name;

SQLite DROP TRIGGER 语句：

DROP TRIGGER trigger\_name

SQLite EXISTS 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name EXISTS (SELECT \* FROM table\_name );

SQLite EXPLAIN 语句：

EXPLAIN INSERT statement...;

or

EXPLAIN QUERY PLAN SELECT statement...;

SQLite GLOB 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name GLOB { PATTERN };

SQLite GROUP BY 子句：

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name;

SQLite HAVING 子句：

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

SQLite INSERT INTO 语句：

INSERT INTO table\_name( column1, column2....columnN)

VALUES ( value1, value2....valueN);

SQLite IN 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

SQLite Like 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name LIKE { PATTERN };

SQLite NOT IN 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name NOT IN (val-1, val-2,...val-N);

SQLite ORDER BY 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

SQLite PRAGMA 语句：

PRAGMA pragma\_name;

For example:

PRAGMA page\_size;

PRAGMA cache\_size = 1024;

PRAGMA table\_info(table\_name);

SQLite RELEASE SAVEPOINT 语句：

RELEASE savepoint\_name;

SQLite REINDEX 语句：

REINDEX collation\_name;

REINDEX database\_name.index\_name;

REINDEX database\_name.table\_name;

SQLite ROLLBACK 语句：

ROLLBACK;

or

ROLLBACK TO SAVEPOINT savepoint\_name;

SQLite SAVEPOINT 语句：

SAVEPOINT savepoint\_name;

SQLite SELECT 语句：

SELECT column1, column2....columnN

FROM table\_name;

SQLite UPDATE 语句：

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

SQLite VACUUM 语句：

VACUUM;

SQLite WHERE 子句：

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION;

# 数据类型

SQLite 数据类型是一个用来指定任何对象的数据类型的属性。SQLite 中的每一列，每个变量和表达式都有相关的数据类型。

可以在创建表的同时使用这些数据类型。SQLite 使用一个更普遍的动态类型系统。在 SQLite 中，值的数据类型与值本身是相关的，而不是与它的容器相关。

## 存储类

每个存储在 SQLite 数据库中的值都具有以下存储类之一。

存储类 描述

NULL 值是一个 NULL 值。

INTEGER 值是一个带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。

REAL 值是一个浮点值，存储为 8 字节的 IEEE 浮点数字。

TEXT 值是一个文本字符串，使用数据库编码（UTF-8、UTF-16BE 或 UTF-16LE）存储。

BLOB 值是一个 blob 数据，完全根据它的输入存储。

SQLite 的存储类稍微比数据类型更普遍。INTEGER 存储类，例如，包含 6 种不同的不同长度的整数数据类型。

## 亲和(Affinity)类型

SQLite支持列的亲和类型概念。任何列仍然可以存储任何类型的数据，当数据插入时，该字段的数据将会优先采用亲缘类型作为该值的存储方式。SQLite目前的版本支持以下五种亲缘类型。

亲和类型 描述

TEXT 数值型数据在被插入之前，需要先被转换为文本格式，之后再插入到目标 字段中。

NUMERIC 当文本数据被插入到亲缘性为NUMERIC的字段中时，如果转换操作不会 导致数据信息丢失以及完全可逆，那么SQLite就会将该文本数据转换为 INTEGER或REAL类型的数据，如果转换失败，SQLite仍会以TEXT方式存 储该数据。对于NULL或BLOB类型的新数据，SQLite将不做任何转换， 直接以NULL或BLOB的方式存储该数据。需要额外说明的是，对于浮点 格式的常量文本，如"30000.0"，如果该值可以转换为INTEGER同时又不 会丢失数值信息，那么SQLite就会将其转换为INTEGER的存储方式。

INTEGER 对于亲缘类型为INTEGER的字段，其规则等同于NUMERIC，唯一差别是 在执行CAST表达式时。

REAL 其规则基本等同于NUMERIC，唯一的差别是不会将"30000.0"这样的文本 数据转换为INTEGER存储方式。

NONE 不做任何的转换，直接以该数据所属的数据类型进行存储。

下表列出了当创建 SQLite3 表时可使用的各种数据类型名称，同时也显示了相应的亲和类型。

数据类型 亲和类型

INT INTEGER

INTEGER

TINYINT

SMALLINT

MEDIUMINT

BIGINT

UNSIGNED BIG INT

INT2

INT8

CHARACTER(20) TEXT

VARCHAR(255)

VARYING CHARACTER(255)

NCHAR(55)

NATIVE CHARACTER(70)

NVARCHAR(100)

TEXT

CLOB

BLOB NONE

no datatype specified

REAL REAL

DOUBLE

DOUBLE PRECISION

FLOAT

NUMERIC NUMERIC

DECIMAL(10,5)

BOOLEAN

DATE

DATETIME

## Boolean 数据类型

SQLite 没有单独的 Boolean 存储类。相反，布尔值被存储为整数 0（false）和 1（true）。

## Date 与 Time 数据类型

SQLite 没有一个单独的用于存储日期和/或时间的存储类，但 SQLite 能够把日期和时间存储为 TEXT、REAL 或 INTEGER 值。

存储类 日期格式

TEXT 格式为 "YYYY-MM-DD HH:MM:SS.SSS" 的日期。

REAL 从公元前 4714 年 11 月 24 日格林尼治时间的正午开始算起的天数。

INTEGER 从 1970-01-01 00:00:00 UTC 算起的秒数。

可以以任何上述格式来存储日期和时间，并且可以使用内置的日期和时间函数来自由转换不同格式。

# 创建数据库

SQLite 的 sqlite3 命令被用来创建新的 SQLite 数据库。不需要任何特殊的权限即可创建一个数据。

## 语法

$sqlite3 DatabaseName.db

通常情况下，数据库名称在 RDBMS 内应该是唯一的。

## 实例

如果想创建一个新的数据库 <testDB.db>，SQLITE3 语句如下所示：

$sqlite3 testDB.db

SQLite version 3.7.15.2 2013-01-09 11:53:05

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

sqlite>

上面的命令将在当前目录下创建一个文件 testDB.db。该文件将被 SQLite 引擎用作数据库。如果您已经注意到 sqlite3 命令在成功创建数据库文件之后，将提供一个 sqlite> 提示符。

一旦数据库被创建，就可以使用 SQLite 的 .databases 命令来检查它是否在数据库列表中，如下所示：

sqlite>.databases

seq name file

--- --------------- ----------------------

0 main /home/sqlite/testDB.db

可以使用 SQLite .quit 命令退出 sqlite 提示符，如下所示：

sqlite>.quit

$

可以在命令提示符中使用 SQLite .dump 点命令来导出完整的数据库在一个文本文件中，如下所示：

$sqlite3 testDB.db .dump > testDB.sql

上面的命令将转换整个 testDB.db 数据库的内容到 SQLite 的语句中，并将其转储到 ASCII 文本文件 testDB.sql 中。可以通过简单的方式从生成的 testDB.sql 恢复，如下所示：

$sqlite3 testDB.db < testDB.sql

# 附加数据库

假设这样一种情况，当在同一时间有多个数据库可用，想使用其中的任何一个。ATTACH DATABASE 语句是用来选择一个特定的数据库，使用该命令后，所有的 SQLite 语句将在附加的数据库下执行。

## 语法

ATTACH DATABASE 'DatabaseName' As 'Alias-Name';

如果数据库尚未被创建，上面的命令将创建一个数据库，如果数据库已存在，则把数据库文件名称与逻辑数据库 'Alias-Name' 绑定在一起。

## 实例

如果想附加一个现有的数据库 testDB.db，则 ATTACH DATABASE 语句将如下所示：

sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';

使用 SQLite .database 命令来显示附加的数据库。

sqlite> .database

seq name file

--- --------------- ----------------------

0 main /home/sqlite/testDB.db

2 test /home/sqlite/testDB.db

数据库名称 main 和 temp 被保留用于主数据库和存储临时表及其他临时数据对象的数据库。这两个数据库名称可用于每个数据库连接，且不应该被用于附加，否则将得到一个警告消息，如下所示：

sqlite> ATTACH DATABASE 'testDB.db' as 'TEMP';

Error: database TEMP is already in use

sqlite> ATTACH DATABASE 'testDB.db' as 'main';

Error: database TEMP is already in use

# 分离数据库

SQLite的 DETACH DTABASE 语句是用来把命名数据库从一个数据库连接分离和游离出来，连接是之前使用 ATTACH 语句附加的。如果同一个数据库文件已经被附加上多个别名，DETACH 命令将只断开给定名称的连接，而其余的仍然有效。您无法分离 main 或 temp 数据库。

如果数据库是在内存中或者是临时数据库，则该数据库将被摧毁，且内容将会丢失。

## 语法

DETACH DATABASE 'Alias-Name';

在这里，'Alias-Name' 与您之前使用 ATTACH 语句附加数据库时所用到的别名相同。

## 实例

假设在前面的章节中您已经创建了一个数据库，并给它附加了 'test' 和 'currentDB'，使用 .database 命令，我们可以看到：

sqlite>.databases

seq name file

--- --------------- ----------------------

0 main /home/sqlite/testDB.db

2 test /home/sqlite/testDB.db

3 currentDB /home/sqlite/testDB.db

现在，让尝试把 'currentDB' 从 testDB.db 中分离出来，如下所示：

sqlite> DETACH DATABASE 'currentDB';

现在，如果检查当前附加的数据库，testDB.db 仍与 'test' 和 'main' 保持连接。

sqlite>.databases

seq name file

--- --------------- ----------------------

0 main /home/sqlite/testDB.db

2 test /home/sqlite/testDB.db

# 创建表

CREATE TABLE 语句用于在任何给定的数据库创建一个新表。创建基本表，涉及到命名表、定义列及每一列的数据类型。

## 语法

CREATE TABLE database\_name.table\_name(

column1 datatype PRIMARY KEY(one or more columns),

column2 datatype,

column3 datatype,

.....

columnN datatype,

);

CREATE TABLE 是告诉数据库系统创建一个新表的关键字。CREATE TABLE 语句后跟着表的唯一的名称或标识。也可以选择指定带有 table\_name 的 database\_name。

## 实例

下面是一个实例，它创建了一个 COMPANY 表，ID 作为主键，NOT NULL 的约束表示在表中创建纪录时这些字段不能为 NULL：

sqlite> CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

sqlite> CREATE TABLE DEPARTMENT(

ID INT PRIMARY KEY NOT NULL,

DEPT CHAR(50) NOT NULL,

EMP\_ID INT NOT NULL

);

create table employer(

id int primary key not null,

name text no null,

age int not null,

address char(50) not null,

slary real not null

);

可以使用 SQLIte 命令中的 .tables 命令来验证表是否已成功创建，该命令用于列出附加数据库中的所有表。

sqlite>.tables

COMPANY DEPARTMENT

在这里，可以看到刚创建的两张表 COMPANY、 DEPARTMENT。

可以使用 SQLite .schema 命令得到表的完整信息，如下所示：

sqlite>.schema COMPANY

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

# 删除表

DROP TABLE 语句用来删除表定义及其所有相关数据、索引、触发器、约束和该表的权限规范。使用此命令时要特别注意，因为一旦一个表被删除，表中所有信息也将永远丢失。

## 语法

可以选择指定带有表名的数据库名称，如下所示：

DROP TABLE database\_name.table\_name;

## 实例

先确认 COMPANY 表已经存在，然后我们将其从数据库中删除。

sqlite>.tables

COMPANY test.COMPANY

这意味着 COMPANY 表已存在数据库中，接下来让把它从数据库中删除，如下：

sqlite>DROP TABLE COMPANY;

sqlite>

如果尝试 .TABLES 命令，那么将无法找到 COMPANY 表了：

sqlite>.tables

sqlite>

显示结果为空，意味着已经成功从数据库删除表。

# Insert into

INSERT INTO 语句用于向数据库的某个表中添加新的数据行。

## 语法

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...valueN);

在这里，column1, column2,...columnN 是要插入数据的表中的列的名称。

如果要为表中的所有列添加值，也可以不需要在 SQLite 查询中指定列名称。但要确保值的顺序与列在表中的顺序一致。SQLite 的 INSERT INTO 语法如下：

INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);

## 实例

假设已经在 testDB.db 中创建了 COMPANY表，如下所示：

sqlite> CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

现在，下面的语句将在 COMPANY 表中创建六个记录：

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );

也可以使用第二种语法在 COMPANY 表中创建一个记录，如下所示：

INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );

上面的所有语句将在 COMPANY 表中创建下列记录。

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

使用一个表来填充另一个表

可以通过在一个有一组字段的表上使用 select 语句，填充数据到另一个表中。 INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

# Select

SELECT语句用于从数据库表中获取数据，以结果表的形式返回数据。这些结果表也被称为结果集。

## 语法

SELECT column1, column2, columnN FROM table\_name;

在这里，column1, column2...是表的字段，即要获取的值。如果想获取所有可用的字段，可以使用下面的语法。

SELECT \* FROM table\_name;

## 实例

假设COMPANY表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

使用SELECT语句获取并显示所有这些记录。在这里，前三个命令被用来设置正确格式化的输出。

sqlite>.header on

sqlite>.mode column

sqlite> SELECT \* FROM COMPANY;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

如果只想获取 COMPANY 表中指定的字段，则使用下面的查询：

sqlite> SELECT ID, NAME, SALARY FROM COMPANY;

ID NAME SALARY

---------- ---------- ----------

1 Paul 20000.0

2 Allen 15000.0

3 Teddy 20000.0

4 Mark 65000.0

5 David 85000.0

6 Kim 45000.0

7 James 10000.0

由于要显示的列的默认宽度导致.mode column输出被截断。使用 .width num, num.... 命令设置显示列的宽度，如下所示。

sqlite>.width 10, 20, 10

sqlite>SELECT \* FROM COMPANY;

.width命令设置第一列的宽度为 10，第二列的宽度为 20，第三列的宽度为 10。

ID NAME AGE ADDRESS SALARY

---------- -------------------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

因为所有的点命令只在SQLite提示符中可用，所以进行带有SQLite的编程时，要使用下面的带有sqlite\_master表的SELECT语句来列出所有在数据库中创建的表：

sqlite> SELECT tbl\_name FROM sqlite\_master WHERE type = 'table';

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

tbl\_name

----------

COMPANY

可以列出关于 COMPANY 表的完整信息，如下所示：

sqlite> SELECT sql FROM sqlite\_master WHERE type = 'table' AND tbl\_name = 'COMPANY';

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

)

# 运算符

运算符是一个保留字或字符，主要用于 SQLite 语句的 WHERE 子句中执行操作，如比较和算术运算。运算符用于指定 SQLite 语句中的条件，并在语句中连接多个条件。

## 算术运算符

假设变量 a=10，变量 b=20，则：

运算符 描述 实例

+ 加法 - 把运算符两边的值相加 a + b 将得到 30

- 减法 - 左操作数减去右操作数 a - b 将得到 -10

\* 乘法 - 把运算符两边的值相乘 a \* b 将得到 200

/ 除法 - 左操作数除以右操作数 b / a 将得到 2

% 取模 - 左操作数除以右操作数后得到的余数 b % a will give 0

### 实例

sqlite> .mode line

sqlite> select 10 + 20;

10 + 20 = 30

sqlite> select 10 - 20;

10 - 20 = -10

sqlite> select 10 \* 20;

10 \* 20 = 200

sqlite> select 10 / 5;

10 / 5 = 2

sqlite> select 12 % 5;

12 % 5 = 2

## 比较运算符

假设变量 a=10，变量 b=20，则：

运算符 描述 实例

== 检查两个操作数的值是否相等，如果相等则条件为真。 (a == b) 不为真。

= 检查两个操作数的值是否相等，如果相等则条件为真。 (a = b) 不为真。

!= 检查两个操作数的值是否相等，如果不相等则条件为真。 (a != b) 为真。

<> 检查两个操作数的值是否相等，如果不相等则条件为真。 (a <> b) 为真。

> 检查左操作数的值是否大于右操作数的值，如果是则条件为真。 (a > b) 不为真。

< 检查左操作数的值是否小于右操作数的值，如果是则条件为真。 (a < b) 为真。

>= 检查左操作数的值是否大于等于右操作数的值，如果是则条件为真。(a >= b) 不为真。

<= 检查左操作数的值是否小于等于右操作数的值，如果是则条件为真。(a <= b) 为真。

!< 检查左操作数的值是否不小于右操作数的值，如果是则条件为真。 (a !< b) 为假。

!> 检查左操作数的值是否不大于右操作数的值，如果是则条件为真。 (a !> b) 为真。

### 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

WHERE 子句是用来设置 SELECT 语句的条件语句。

下面的 SELECT 语句列出了 SALARY 大于 50,000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE SALARY > 50000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面的 SELECT 语句列出了 SALARY 等于 20,000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE SALARY = 20000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

3 Teddy 23 Norway 20000.0

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE SALARY != 20000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE SALARY <> 20000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 SALARY 大于等于 65,000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE SALARY >= 65000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

## 逻辑运算符

运算符 描述

AND AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。

BETWEEN BETWEEN 运算符用于在给定最小值和最大值范围内的一系列值中搜索值。

EXISTS EXISTS 运算符用于在满足一定条件的指定表中搜索行的存在。

IN IN 运算符用于把某个值与一系列指定列表的值进行比较。

NOT IN IN 运算符的对立面，用于把某个值与不在一系列指定列表的值进行比较。

LIKE LIKE 运算符用于把某个值与使用通配符运算符的相似值进行比较。

GLOB GLOB 运算符用于把某个值与使用通配符运算符的相似值进行比较。GLOB 与 LIKE 不同之处在于，它是大小写敏感的。

NOT NOT 运算符是所用的逻辑运算符的对立面。比如 NOT EXISTS、NOT BETWEEN、NOT IN，等等。它是否定运算符。

OR OR 运算符用于结合一个 SQL 语句的 WHERE 子句中的多个条件。

IS NULL NULL 运算符用于把某个值与 NULL 值进行比较。

IS IS 运算符与 = 相似。

IS NOT IS NOT 运算符与 != 相似。

|| 连接两个不同的字符串，得到一个新的字符串。

UNIQUE UNIQUE 运算符搜索指定表中的每一行，确保唯一性（无重复）。

### 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面的 SELECT 语句列出了 AGE 不为 NULL 的所有记录，结果显示所有的记录，意味着没有一个记录的 AGE 等于 NULL：

sqlite> SELECT \* FROM COMPANY WHERE AGE IS NOT NULL;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

sqlite> SELECT \* FROM COMPANY WHERE NAME LIKE 'Ki%';

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

6 Kim 22 South-Hall 45000.0

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

sqlite> SELECT \* FROM COMPANY WHERE NAME GLOB 'Ki\*';

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

6 Kim 22 South-Hall 45000.0

下面的 SELECT 语句列出了 AGE 的值为 25 或 27 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE IN ( 25, 27 );

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面的 SELECT 语句列出了 AGE 的值既不是 25 也不是 27 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE NOT IN ( 25, 27 );

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

3 Teddy 23 Norway 20000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 AGE 的值在 25 与 27 之间的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE BETWEEN 25 AND 27;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 EXISTS 运算符一起使用，列出了外查询中的 AGE 存在于子查询返回的结果中的所有记录：

sqlite> SELECT AGE FROM COMPANY

WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);

AGE

----------

32

25

23

25

27

22

24

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 > 运算符一起使用，列出了外查询中的 AGE 大于子查询返回的结果中的年龄的所有记录：

sqlite> SELECT \* FROM COMPANY

WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

## 位运算符

位运算符作用于位，并逐位执行操作。真值表 & 和 | 如下：

p q p & q p | q

0 0 0 0

0 1 0 1

1 1 1 1

1 0 0 1

假设如果 A = 60，且 B = 13，现在以二进制格式，它们如下所示：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

~A = 1100 0011

假设变量 A=60，变量 B=13，则：

运算符 描述

& 如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。 (A & B) 将得到 12，即为 0000 1100

| 如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。 (A | B) 将得到 61，即为 0011 1101

~ 二进制补码运算符是一元运算符，具有"翻转"位效应，即0变成1，1变成0。 (~A ) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。

<< 二进制左移运算符。左操作数的值向左移动右操作数指定的位数。 A << 2 将得到 240，即为 1111 0000

>> 二进制右移运算符。左操作数的值向右移动右操作数指定的位数。 A >> 2 将得到 15，即为 0000 1111

### 实例

sqlite> .mode line

sqlite> select 60 | 13;

60 | 13 = 61

sqlite> select 60 & 13;

60 & 13 = 12

sqlite> select 60 ^ 13;

10 \* 20 = 200

sqlite> select (~60);

(~60) = -61

sqlite> select (60 << 2);

(60 << 2) = 240

sqlite> select (60 >> 2);

(60 >> 2) = 15

# 表达式

表达式是一个或多个值、运算符和计算值的SQL函数的组合。SQL 表达式与公式类似，都写在查询语言中。还可以使用特定的数据集来查询数据库。

## 语法

假设 SELECT 语句的基本语法如下：

SELECT column1, column2, columnN

FROM table\_name

WHERE [CONTION | EXPRESSION];

## 布尔表达式

SQLite 的布尔表达式在匹配单个值的基础上获取数据。语法如下：

SELECT column1, column2, columnN

FROM table\_name

WHERE SINGLE VALUE MATCHTING EXPRESSION;

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的实例演示了 SQLite 布尔表达式的用法：

sqlite> SELECT \* FROM COMPANY WHERE SALARY = 10000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 James 24 Houston 10000.0

## 数值表达式

这些表达式用来执行查询中的任何数学运算。语法如下：

SELECT numerical\_expression as OPERATION\_NAME

[FROM table\_name WHERE CONDITION] ;

numerical\_expression 用于数学表达式或任何公式。

sqlite> SELECT (15 + 6) AS ADDITION

ADDITION = 21

有几个内置的函数，比如 avg()、sum()、count()，等等，执行被称为对一个表或一个特定的表列的汇总数据计算。

sqlite> SELECT COUNT(\*) AS "RECORDS" FROM COMPANY;

RECORDS = 7

## 日期表达式

日期表达式返回当前系统日期和时间值，这些表达式将被用于各种数据操作。

sqlite> SELECT CURRENT\_TIMESTAMP;

CURRENT\_TIMESTAMP = 2013-03-17 10:43:35

# Where子句

SQLite的 WHERE 子句用于指定从一个表或多个表中获取数据的条件。

如果满足给定的条件，即为真（true）时，则从表中返回特定的值。可以使用 WHERE 子句来过滤记录，只获取需要的记录。

WHERE 子句不仅可用在 SELECT 语句中，它也可用在 UPDATE、DELETE 语句中，等等。

# AND/OR 运算符

SQLite 的 AND 和 OR 运算符用于编译多个条件来缩小在 SQLite 语句中所选的数据。这两个运算符被称为连接运算符。

这些运算符为同一个 SQLite 语句中不同的运算符之间的多个比较提供了可能。

## AND

AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。使用 AND 运算符时，只有当所有条件都为真（true）时，整个条件为真（true）。例如，只有当 condition1 和 condition2 都为真（true）时，[condition1] AND [condition2] 为真（true）。

### 语法

带有 WHERE 子句的 AND 运算符的基本语法如下：

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition1] AND [condition2]...AND [conditionN];

使用 AND 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，所有由 AND 分隔的条件都必须为真（TRUE）。

### 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

## OR

OR 运算符也用于结合一个 SQL 语句的 WHERE 子句中的多个条件。使用 OR 运算符时，只要当条件中任何一个为真（true）时，整个条件为真（true）。例如，只要当 condition1 或 condition2 有一个为真（true）时，[condition1] OR [condition2] 为真（true）。

### 语法

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition1] OR [condition2]...OR [conditionN]

可以使用 OR 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，只要任何一个由 OR 分隔的条件为真（TRUE）即可。

### 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

# Update

SQLite 的 UPDATE 查询用于修改表中已有的记录。可以使用带有 WHERE 子句的 UPDATE 查询来更新选定行，否则所有的行都会被更新。

## 语法

UPDATE table\_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

使用 AND 或 OR 运算符来结合 N 个数量的条件。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它会更新 ID 为 6 的客户地址：

sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;

现在，COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 Texas 45000.0

7 James 24 Houston 10000.0

如果想修改 COMPANY 表中 ADDRESS 和 SALARY 列的所有值，则不需要使用 WHERE 子句，UPDATE 查询如下：

sqlite> UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00;

现在，COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 Texas 20000.0

2 Allen 25 Texas 20000.0

3 Teddy 23 Texas 20000.0

4 Mark 25 Texas 20000.0

5 David 27 Texas 20000.0

6 Kim 22 Texas 20000.0

7 James 24 Texas 20000.0

# Delete

SQLite 的 DELETE 查询用于删除表中已有的记录。可以使用带有 WHERE 子句的 DELETE 查询来删除选定行，否则所有的记录都会被删除。

## 语法

DELETE FROM table\_name

WHERE [condition];

使用 AND 或 OR 运算符来结合 N 个数量的条件。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它会删除 ID 为 7 的客户：

sqlite> DELETE FROM COMPANY WHERE ID = 7;

现在，COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

如果想要从 COMPANY 表中删除所有记录，则不需要使用 WHERE 子句，DELETE 查询如下：

sqlite> DELETE FROM COMPANY;

现在，COMPANY 表中没有任何的记录，因为所有的记录已经通过 DELETE 语句删除。

# Like 子句

SQLite 的 LIKE 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，LIKE 运算符将返回真（true），也就是 1。这里有两个通配符与 LIKE 运算符一起使用：

百分号（%）代表零个、一个或多个数字或字符。下划线（\_）代表一个单一的数字或字符。这些符号可以被组合使用。

## 语法

SELECT column\_list

FROM table\_name

WHERE column LIKE 'XXXX%'

SELECT column\_list

FROM table\_name

WHERE column LIKE '%XXXX%'

SELECT column\_list

FROM table\_name

WHERE column LIKE 'XXXX\_'

SELECT column\_list

FROM table\_name

WHERE column LIKE '\_XXXX'

SELECT column\_list

FROM table\_name

WHERE column LIKE '\_XXXX\_'

使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

## 实例

语句 描述

WHERE SALARY LIKE '200%' 查找以 200 开头的任意值

WHERE SALARY LIKE '%200%' 查找任意位置包含 200 的任意值

WHERE SALARY LIKE '\_00%' 查找第二位和第三位为 00 的任意值

WHERE SALARY LIKE '2\_%\_%' 查找以 2 开头，且长度至少为 3 个字符的任意值

WHERE SALARY LIKE '%2' 查找以 2 结尾的任意值

WHERE SALARY LIKE '\_2%3' 查找第二位为 2，且以 3 结尾的任意值

WHERE SALARY LIKE '2\_\_\_3' 查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE LIKE '2%';

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE ADDRESS LIKE '%-%';

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

6 Kim 22 South-Hall 45000.0

# Glob 子句

SQLite 的 GLOB 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，GLOB 运算符将返回真（true），也就是 1。与 LIKE 运算符不同的是，GLOB 是大小写敏感的，对于下面的通配符，它遵循 UNIX 的语法。

星号（\*）代表零个、一个或多个数字或字符。问号（?）代表一个单一的数字或字符。这些符号可以被组合使用。

## 语法

SELECT FROM table\_name

WHERE column GLOB 'XXXX\*'

SELECT FROM table\_name

WHERE column GLOB '\*XXXX\*'

SELECT FROM table\_name

WHERE column GLOB 'XXXX?'

SELECT FROM table\_name

WHERE column GLOB '?XXXX'

SELECT FROM table\_name

WHERE column GLOB '?XXXX?'

SELECT FROM table\_name

WHERE column GLOB '????'

使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

## 实例

语句 描述

WHERE SALARY GLOB '200\*' 查找以 200 开头的任意值

WHERE SALARY GLOB '\*200\*' 查找任意位置包含 200 的任意值

WHERE SALARY GLOB '?00\*' 查找第二位和第三位为 00 的任意值

WHERE SALARY GLOB '2??' 查找以 2 开头，且长度至少为 3 个字符的任意值

WHERE SALARY GLOB '\*2' 查找以 2 结尾的任意值

WHERE SALARY GLOB '?2\*3' 查找第二位为 2，且以 3 结尾的任意值

WHERE SALARY GLOB '2???3' 查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE AGE GLOB '2\*';

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

sqlite> SELECT \* FROM COMPANY WHERE ADDRESS GLOB '\*-\*';

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

6 Kim 22 South-Hall 45000.0

# Limit 子句

SQLite 的 LIMIT 子句用于限制由 SELECT 语句返回的数据数量。

## 语法

SELECT column1, column2, columnN

FROM table\_name

LIMIT [no of rows]

LIMIT 子句与 OFFSET 子句一起使用时的语法：

SELECT column1, column2, columnN

FROM table\_name

LIMIT [no of rows] OFFSET [row num]

SQLite 引擎将返回从下一行开始直到给定的 OFFSET 为止的所有行，如下面的最后一个实例所示。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面是一个实例，它限制了您想要从表中提取的行数：

sqlite> SELECT \* FROM COMPANY LIMIT 6;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

但是，在某些情况下，可能需要从一个特定的偏移开始提取记录。下面是一个实例，从第三位开始提取 3 个记录：

sqlite> SELECT \* FROM COMPANY LIMIT 3 OFFSET 2;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

# Order By

SQLite 的 ORDER BY 子句是用来基于一个或多个列按升序或降序顺序排列数据。

## 语法

SELECT column-list

FROM table\_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

可以在 ORDER BY 子句中使用多个列。确保您使用的排序列在列清单中。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

面是一个实例，它会将结果按 SALARY 升序排序：

sqlite> SELECT \* FROM COMPANY ORDER BY SALARY ASC;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

7 James 24 Houston 10000.0

2 Allen 25 Texas 15000.0

1 Paul 32 California 20000.0

3 Teddy 23 Norway 20000.0

6 Kim 22 South-Hall 45000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面是一个实例，它会将结果按 NAME 和 SALARY 升序排序：

sqlite> SELECT \* FROM COMPANY ORDER BY NAME, SALARY ASC;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

5 David 27 Texas 85000.0

7 James 24 Houston 10000.0

6 Kim 22 South-Hall 45000.0

4 Mark 25 Rich-Mond 65000.0

1 Paul 32 California 20000.0

3 Teddy 23 Norway 20000.0

下面是一个实例，它会将结果按 NAME 降序排序：

sqlite> SELECT \* FROM COMPANY ORDER BY NAME DESC;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

3 Teddy 23 Norway 20000.0

1 Paul 32 California 20000.0

4 Mark 25 Rich-Mond 65000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

5 David 27 Texas 85000.0

2 Allen 25 Texas 15000.0

# Group By

SQLite 的 GROUP BY 子句用于与 SELECT 语句一起使用，来对相同的数据进行分组。在 SELECT 语句中，GROUP BY 子句放在 WHERE 子句之后，放在 ORDER BY 子句之前。

## 语法

SELECT column-list

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2....columnN

ORDER BY column1, column2....columnN

可以在 GROUP BY 子句中使用多个列。确保您使用的分组列在列清单中。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

如果想了解每个客户的工资总额，则可使用 GROUP BY 查询，如下所示：

sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;

这将产生以下结果：

NAME SUM(SALARY)

---------- -----------

Allen 15000.0

David 85000.0

James 10000.0

Kim 45000.0

Mark 65000.0

Paul 20000.0

Teddy 20000.0

使用下面的 INSERT 语句在 COMPANY 表中另外创建三个记录：

INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00 );

INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00 );

INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00 );

现在表具有重复名称的记录，如下所示：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

8 Paul 24 Houston 20000.0

9 James 44 Norway 5000.0

10 James 45 Texas 5000.0

sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME ORDER BY NAME;

这将产生以下结果：

NAME SUM(SALARY)

---------- -----------

Allen 15000

David 85000

James 20000

Kim 45000

Mark 65000

Paul 40000

Teddy 20000

sqlite> SELECT NAME, SUM(SALARY)

FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;

这将产生以下结果：

NAME SUM(SALARY)

---------- -----------

Teddy 20000

Paul 40000

Mark 65000

Kim 45000

James 20000

David 85000

Allen 15000

# Having 子句

HAVING 子句允许指定条件来过滤将出现在最终结果中的分组结果。WHERE 子句在所选列上设置条件，而 HAVING 子句则在由 GROUP BY 子句创建的分组上设置条件。

## 语法

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

在一个查询中，HAVING 子句必须放在 GROUP BY 子句之后，必须放在 ORDER BY 子句之前。

SELECT column1, column2

FROM table1, table2

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

ORDER BY column1, column2

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

8 Paul 24 Houston 20000.0

9 James 44 Norway 5000.0

10 James 45 Texas 5000.0

下面是一个实例，显示名称计数小于 2 的所有记录：

sqlite > SELECT \* FROM COMPANY GROUP BY name HAVING count(name) < 2;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000

5 David 27 Texas 85000

6 Kim 22 South-Hall 45000

4 Mark 25 Rich-Mond 65000

3 Teddy 23 Norway 20000

下面是一个实例，它将显示名称计数大于 2 的所有记录：

sqlite > SELECT \* FROM COMPANY GROUP BY name HAVING count(name) > 2;

这将产生以下结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

10 James 45 Texas 5000

# Distinct 关键字

SQLite 的 DISTINCT 关键字与 SELECT 语句一起使用，来消除所有重复的记录，并只获取唯一一次记录。有可能出现一种情况，在一个表中有多个重复的记录。当提取这样的记录时，DISTINCT 关键字就显得特别有意义，它只获取唯一一次记录，而不是获取重复记录。

## 语法

SELECT DISTINCT column1, column2,.....columnN

FROM table\_name

WHERE [condition]

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

8 Paul 24 Houston 20000.0

9 James 44 Norway 5000.0

10 James 45 Texas 5000.0

sqlite> SELECT name FROM COMPANY;

这将产生以下结果：

NAME

----------

Paul

Allen

Teddy

Mark

David

Kim

James

Paul

James

James

sqlite> SELECT DISTINCT name FROM COMPANY;

这将产生以下结果，没有任何重复的条目：

NAME

----------

Paul

Allen

Teddy

Mark

David

Kim

James

# PRAGMA

SQLite 的 PRAGMA 命令是一个特殊的命令，可以用在 SQLite 环境内控制各种环境变量和状态标志。一个 PRAGMA 值可以被读取，也可以根据需求进行设置。

## 语法

要查询当前的 PRAGMA 值，只需要提供该 pragma 的名字：

PRAGMA pragma\_name;

要为 PRAGMA 设置一个新的值，语法如下：

PRAGMA pragma\_name = value;

设置模式，可以是名称或等值的整数，但返回的值将始终是一个整数。

## auto\_vacuum

auto\_vacuum Pragma 获取或设置 auto-vacuum 模式。语法如下：

PRAGMA [database.]auto\_vacuum;

PRAGMA [database.]auto\_vacuum = mode;

其中，mode 可以是以下任何一种：

Pragma 值 描述

0 或 NONE 禁用 Auto-vacuum。这是默认模式，意味着数据库文件尺寸大小不会缩小，除非手动使用 VACUUM 命令。

1 或 FULL 启用 Auto-vacuum，是全自动的。在该模式下，允许数据库文件随着数据从数据库移除而缩小。

2 或 INCREMENTAL 启用 Auto-vacuum，但是必须手动激活。在该模式下，引用数据被维持，免费页面只放在免费列表中。这些页面可在任何时候使用 incremental\_vacuum pragma 进行覆盖。

## cache\_size

cache\_size Pragma 可获取或暂时设置在内存中页面缓存的最大尺寸。语法如下：

PRAGMA [database.]cache\_size;

PRAGMA [database.]cache\_size = pages;

pages 值表示在缓存中的页面数。内置页面缓存的默认大小为 2,000 页，最小尺寸为 10 页。

## case\_sensitive\_like

case\_sensitive\_like Pragma 控制内置的 LIKE 表达式的大小写敏感度。默认情况下，该 Pragma 为 false，这意味着，内置的 LIKE 操作符忽略字母的大小写。语法如下：

PRAGMA case\_sensitive\_like = [true|false];

目前没有办法查询该 Pragma 的当前状态。

## count\_changes

count\_changes Pragma 获取或设置数据操作语句的返回值，如 INSERT、UPDATE 和 DELETE。语法如下：

PRAGMA count\_changes;

PRAGMA count\_changes = [true|false];

默认情况下，该 Pragma 为 false，这些语句不返回任何东西。如果设置为 true，每个所提到的语句将返回一个单行单列的表，由一个单一的整数值组成，该整数表示操作影响的行。

## database\_list

database\_list Pragma 将用于列出了所有的数据库连接。语法如下：

PRAGMA database\_list;

该 Pragma 将返回一个单行三列的表格，每当打开或附加数据库时，会给出数据库中的序列号，它的名称和相关的文件。

## encoding

encoding Pragma 控制字符串如何编码及存储在数据库文件中。语法如下：

PRAGMA encoding;

PRAGMA encoding = format;

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

## freelist\_count

freelist\_count Pragma 返回一个整数，表示当前被标记为免费和可用的数据库页数。语法如下：

PRAGMA [database.]freelist\_count;

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

## index\_info

index\_info Pragma 返回关于数据库索引的信息。语法如下：

PRAGMA [database.]index\_info( index\_name );

结果集将为每个包含在给出列序列的索引、表格内的列索引、列名称的列显示一行。

## index\_list

index\_list Pragma 列出所有与表相关联的索引。语法如下：

PRAGMA [database.]index\_list( table\_name );

结果集将为每个给出列序列的索引、索引名称、表示索引是否唯一的标识显示一行。

## journal\_mode

journal\_mode Pragma 获取或设置控制日志文件如何存储和处理的日志模式。语法如下：:

PRAGMA journal\_mode;

PRAGMA journal\_mode = mode;

PRAGMA database.journal\_mode;

PRAGMA database.journal\_mode = mode;

这里支持五种日志模式：

Pragma 值 描述

DELETE 默认模式。在该模式下，在事务结束时，日志文件将被删除。

TRUNCATE 日志文件被阶段为零字节长度。

PERSIST 日志文件被留在原地，但头部被重写，表明日志不再有效。

MEMORY 日志记录保留在内存中，而不是磁盘上。

OFF 不保留任何日志记录。

## max\_page\_count

max\_page\_count Pragma 为数据库获取或设置允许的最大页数。语法如下：

PRAGMA [database.]max\_page\_count;

PRAGMA [database.]max\_page\_count = max\_page;

默认值是 1,073,741,823，这是一个千兆的页面，即如果默认 1 KB 的页面大小，那么数据库中增长起来的一个兆字节。

## page\_count

page\_count Pragma 返回当前数据库中的网页数量。语法如下：

PRAGMA [database.]page\_count;

数据库文件的大小应该是 page\_count \* page\_size。

## page\_size

page\_size Pragma 获取或设置数据库页面的大小。语法如下：

PRAGMA [database.]page\_size;

PRAGMA [database.]page\_size = bytes;

默认情况下，允许的尺寸是 512、1024、2048、4096、8192、16384、32768 字节。改变现有数据库页面大小的唯一方法就是设置页面大小，然后立即 VACUUM 该数据库。

## parser\_trace

parser\_trace Pragma 随着它解析 SQL 命令来控制打印的调试状态，语法如下：

PRAGMA parser\_trace = [true|false];

默认情况下，它被设置为 false，但设置为 true 时则启用，此时 SQL 解析器会随着它解析 SQL 命令来打印出它的状态。

## recursive\_triggers

recursive\_triggers Pragma 获取或设置递归触发器功能。如果未启用递归触发器，一个触发动作将不会触发另一个触发。语法如下：

PRAGMA recursive\_triggers;

PRAGMA recursive\_triggers = [true|false];

## schema\_version

schema\_version Pragma 获取或设置存储在数据库头中的的架构版本值。语法如下：

PRAGMA [database.]schema\_version;

PRAGMA [database.]schema\_version = number;

这是一个 32 位有符号整数值，用来跟踪架构的变化。每当一个架构改变命令执行（比如 CREATE... 或 DROP...）时，这个值会递增。

## secure\_delete

secure\_delete Pragma 用来控制内容是如何从数据库中删除。语法如下：

PRAGMA secure\_delete;

PRAGMA secure\_delete = [true|false];

PRAGMA database.secure\_delete;

PRAGMA database.secure\_delete = [true|false];

安全删除标志的默认值通常是关闭的，但是这是可以通过 SQLITE\_SECURE\_DELETE 构建选项来改变的。

## sql\_trace

sql\_trace Pragma 用于把 SQL 跟踪结果转储到屏幕上。语法如下：

PRAGMA sql\_trace;

PRAGMA sql\_trace = [true|false];

SQLite 必须通过 SQLITE\_DEBUG 指令来编译要引用的该 Pragma。

## synchronous

synchronous Pragma 获取或设置当前磁盘的同步模式，该模式控制积极的 SQLite 如何将数据写入物理存储。语法如下：

PRAGMA [database.]synchronous;

PRAGMA [database.]synchronous = mode;

SQLite 支持下列同步模式：

Pragma 值 描述

0 或 OFF 不进行同步。

1 或 NORMAL 在关键的磁盘操作的每个序列后同步。

2 或 FULL 在每个关键的磁盘操作后同步。

## temp\_store

temp\_store Pragma 获取或设置临时数据库文件所使用的存储模式。语法如下：

PRAGMA temp\_store;

PRAGMA temp\_store = mode;

SQLite 支持下列存储模式：

Pragma 值 描述

0 或 DEFAULT 默认使用编译时的模式。通常是 FILE。

1 或 FILE 使用基于文件的存储。

2 或 MEMORY 使用基于内存的存储。

## temp\_store\_directory

temp\_store\_directory Pragma 获取或设置用于临时数据库文件的位置。语法如下：

PRAGMA temp\_store\_directory;

PRAGMA temp\_store\_directory = 'directory\_path';

## user\_version

user\_version Pragma 获取或设置存储在数据库头的用户自定义的版本值。语法如下：

PRAGMA [database.]user\_version;

PRAGMA [database.]user\_version = number;

这是一个 32 位的有符号整数值，可以由开发人员设置，用于版本跟踪的目的。

## writable\_schema

writable\_schema Pragma 获取或设置是否能够修改系统表。语法如下：

PRAGMA writable\_schema;

PRAGMA writable\_schema = [true|false];

如果设置了该 Pragma，则表以 sqlite\_ 开始，可以创建和修改，包括 sqlite\_master 表。使用该 Pragma 时要注意，因为它可能导致整个数据库损坏。

# 约束

约束是在表的数据列上强制执行的规则。这些是用来限制可以插入到表中的数据类型。这确保了数据库中数据的准确性和可靠性。

约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。以下是在 SQLite 中常用的约束。

NOT NULL 约束：确保某列不能有 NULL 值。

DEFAULT 约束：当某列没有指定值时，为该列提供默认值。

UNIQUE 约束：确保某列中的所有值是不同的。

PRIMARY Key 约束：唯一标识数据库表中的各行/记录。

CHECK 约束：CHECK 约束确保某列中的所有值满足一定条件。

## NOT NULL

默认情况下，列可以保存 NULL 值。如果您不想某列有 NULL 值，那么需要在该列上定义此约束，指定在该列上不允许 NULL 值。

NULL 与没有数据是不一样的，它代表着未知的数据。

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列，其中 ID、NAME 和 AGE 三列指定不接受 NULL 值：

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

## DEFAULT

DEFAULT 约束在 INSERT INTO 语句没有提供一个特定的值时，为列提供一个默认值。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，SALARY 列默认设置为 5000.00。所以当 INSERT INTO 语句没有为该列提供值时，该列将被设置为 5000.00。

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL DEFAULT 50000.00

);

## UNIQUE

UNIQUE 约束防止在一个特定的列存在两个记录具有相同的值。在 COMPANY 表中，例如，您可能要防止两个或两个以上的人具有相同的年龄。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，AGE 列设置为 UNIQUE，所以不能有两个相同年龄的记录：

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL UNIQUE,

ADDRESS CHAR(50),

SALARY REAL DEFAULT 50000.00

);

## PRIMARY KEY

PRIMARY KEY 约束唯一标识数据库表中的每个记录。在一个表中可以有多个 UNIQUE 列，但只能有一个主键。在设计数据库表时，主键是很重要的。主键是唯一的 ID。

我们使用主键来引用表中的行。可通过把主键设置为其他表的外键，来创建表之间的关系。由于"长期存在编码监督"，在 SQLite 中，主键可以是 NULL，这是与其他数据库不同的地方。

主键是表中的一个字段，唯一标识数据库表中的各行/记录。主键必须包含唯一值。主键列不能有 NULL 值。

一个表只能有一个主键，它可以由一个或多个字段组成。当多个字段作为主键，它们被称为复合键。

如果一个表在任何字段上定义了一个主键，那么在这些字段上不能有两个记录具有相同的值。

已经看到了我们创建以 ID 作为主键的 COMAPNY 表的各种实例：

CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

## CHECK

CHECK 约束启用输入一条记录要检查值的条件。如果条件值为 false，则记录违反了约束，且不能输入到表。

实例

例如，下面的 SQLite 创建一个新的表 COMPANY，并增加了五列。在这里，我们为 SALARY 列添加 CHECK，所以工资不能为零：

CREATE TABLE COMPANY3(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL CHECK(SALARY > 0)

);

## 删除约束

SQLite 支持 ALTER TABLE 的有限子集。在 SQLite 中，ALTER TABLE 命令允许用户重命名表，或向现有表添加一个新的列。重命名列，删除一列，或从一个表中添加或删除约束都是不可能的。

# Joins

SQLite 的 Joins 子句用于结合两个或多个数据库中表的记录。JOIN 是一种通过共同值来结合两个表中字段的手段。

SQL 定义了三种主要类型的连接：

交叉连接 - CROSS JOIN

内连接 - INNER JOIN

外连接 - OUTER JOIN

在我们继续之前，让我们假设有两个表 COMPANY 和 DEPARTMENT。我们已经看到了用来填充 COMPANY 表的 INSERT 语句。现在让我们假设 COMPANY 表的记录列表如下：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

另一个表是 DEPARTMENT，定义如下：

CREATE TABLE DEPARTMENT(

ID INT PRIMARY KEY NOT NULL,

DEPT CHAR(50) NOT NULL,

EMP\_ID INT NOT NULL

);

下面是填充 DEPARTMENT 表的 INSERT 语句：

INSERT INTO DEPARTMENT (ID, DEPT, EMP\_ID)

VALUES (1, 'IT Billing', 1 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP\_ID)

VALUES (2, 'Engineering', 2 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP\_ID)

VALUES (3, 'Finance', 7 );

最后，我们在 DEPARTMENT 表中有下列的记录列表：

ID DEPT EMP\_ID

---------- ---------- ----------

1 IT Billing 1

2 Engineerin 2

3 Finance 7

## CROSS JOIN

交叉连接（CROSS JOIN）把第一个表的每一行与第二个表的每一行进行匹配。如果两个输入表分别有 x 和 y 列，则结果表有 x+y 列。由于交叉连接（CROSS JOIN）有可能产生非常大的表，使用时必须谨慎，只在适当的时候使用它们。

下面是交叉连接（CROSS JOIN）的语法：

SELECT ... FROM table1 CROSS JOIN table2 ...

基于上面的表，我们可以写一个交叉连接（CROSS JOIN），如下所示：

sqlite> SELECT EMP\_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;

上面的查询会产生以下结果：

EMP\_ID NAME DEPT

---------- ---------- ----------

1 Paul IT Billing

2 Paul Engineerin

7 Paul Finance

1 Allen IT Billing

2 Allen Engineerin

7 Allen Finance

1 Teddy IT Billing

2 Teddy Engineerin

7 Teddy Finance

1 Mark IT Billing

2 Mark Engineerin

7 Mark Finance

1 David IT Billing

2 David Engineerin

7 David Finance

1 Kim IT Billing

2 Kim Engineerin

7 Kim Finance

1 James IT Billing

2 James Engineerin

7 James Finance

## INNER JOIN

内连接（INNER JOIN）根据连接谓词结合两个表（table1 和 table2）的列值来创建一个新的结果表。查询会把 table1 中的每一行与 table2 中的每一行进行比较，找到所有满足连接谓词的行的匹配对。当满足连接谓词时，A 和 B 行的每个匹配对的列值会合并成一个结果行。

内连接（INNER JOIN）是最常见的连接类型，是默认的连接类型。INNER 关键字是可选的。

下面是内连接（INNER JOIN）的语法：

SELECT ... FROM table1 [INNER] JOIN table2 ON conditional\_expression ...

为了避免冗余，并保持较短的措辞，可以使用 USING 表达式声明内连接（INNER JOIN）条件。这个表达式指定一个或多个列的列表：

SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...

自然连接（NATURAL JOIN）类似于 JOIN...USING，只是它会自动测试存在两个表中的每一列的值之间相等值：

SELECT ... FROM table1 NATURAL JOIN table2...

基于上面的表，我们可以写一个内连接（INNER JOIN），如下所示：

sqlite> SELECT EMP\_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID;

上面的查询会产生以下结果：

EMP\_ID NAME DEPT

---------- ---------- ----------

1 Paul IT Billing

2 Allen Engineerin

7 James Finance

# OUTER JOIN

外连接（OUTER JOIN）是内连接（INNER JOIN）的扩展。虽然 SQL 标准定义了三种类型的外连接：LEFT、RIGHT、FULL，但 SQLite 只支持 左外连接（LEFT OUTER JOIN）。

外连接（OUTER JOIN）声明条件的方法与内连接（INNER JOIN）是相同的，使用 ON、USING 或 NATURAL 关键字来表达。最初的结果表以相同的方式进行计算。一旦主连接计算完成，外连接（OUTER JOIN）将从一个或两个表中任何未连接的行合并进来，外连接的列使用 NULL 值，将它们附加到结果表中。

下面是左外连接（LEFT OUTER JOIN）的语法：

SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional\_expression ...

为了避免冗余，并保持较短的措辞，可以使用 USING 表达式声明外连接（OUTER JOIN）条件。这个表达式指定一个或多个列的列表：

SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,... ) ...

基于上面的表，我们可以写一个外连接（OUTER JOIN），如下所示：

sqlite> SELECT EMP\_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID;

上面的查询会产生以下结果：

EMP\_ID NAME DEPT

---------- ---------- ----------

1 Paul IT Billing

2 Allen Engineerin

Teddy

Mark

David

Kim

7 James Finance

# Unions

SQLite的 UNION 子句/运算符用于合并两个或多个 SELECT 语句的结果，不返回任何重复的行。

为了使用 UNION，每个 SELECT 被选择的列数必须是相同的，相同数目的列表达式，相同的数据类型，并确保它们有相同的顺序，但它们不必具有相同的长度。

## 语法

UNION 的基本语法如下：

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

UNION

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

这里给定的条件根据需要可以是任何表达式。

## 实例

假设有下面两个表，（1）COMPANY 表如下所示：

sqlite> select \* from COMPANY;

ID NAME AGE ADDRESS SALARY

---------- -------------------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

（2）另一个表是 DEPARTMENT，如下所示：

ID DEPT EMP\_ID

---------- -------------------- ----------

1 IT Billing 1

2 Engineering 2

3 Finance 7

4 Engineering 3

5 Finance 4

6 Engineering 5

7 Finance 6

现在，让我们使用 SELECT 语句及 UNION 子句来连接两个表，如下所示：

sqlite> SELECT EMP\_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID

UNION

SELECT EMP\_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID;

这将产生以下结果：

EMP\_ID NAME DEPT

---------- -------------------- ----------

1 Paul IT Billing

2 Allen Engineerin

3 Teddy Engineerin

4 Mark Finance

5 David Engineerin

6 Kim Finance

7 James Finance

## UNION ALL

UNION ALL 运算符用于结合两个 SELECT 语句的结果，包括重复行。

适用于 UNION 的规则同样适用于 UNION ALL 运算符。

### 语法

UNION ALL 的基本语法如下：

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

这里给定的条件根据需要可以是任何表达式。

### 实例

现在，让我们使用 SELECT 语句及 UNION ALL 子句来连接两个表，如下所示：

sqlite> SELECT EMP\_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID

UNION ALL

SELECT EMP\_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID;

这将产生以下结果：

EMP\_ID NAME DEPT

---------- -------------------- ----------

1 Paul IT Billing

2 Allen Engineerin

3 Teddy Engineerin

4 Mark Finance

5 David Engineerin

6 Kim Finance

7 James Finance

1 Paul IT Billing

2 Allen Engineerin

3 Teddy Engineerin

4 Mark Finance

5 David Engineerin

6 Kim Finance

7 James Finance

# NULL值

SQLite 的 NULL 是用来表示一个缺失值的项。表中的一个 NULL 值是在字段中显示为空白的一个值。

带有 NULL 值的字段是一个不带有值的字段。NULL 值与零值或包含空格的字段是不同的。

## 语法

创建表时使用 NULL 的基本语法如下：

SQLite> CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

在这里，NOT NULL 表示列总是接受给定数据类型的显式值。

带有 NULL 值的字段在记录创建的时候可以保留为空。

## 实例

NULL 值在选择数据时会引起问题，因为当把一个未知的值与另一个值进行比较时，结果总是未知的，且不会包含在最后的结果中。假设有下面的表，COMPANY 的记录如下所示：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

让我们使用 UPDATE 语句来设置一些允许空值的值为 NULL，如下所示：

sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID IN(6,7);

现在，COMPANY 表的记录如下所示：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22

7 James 24

接下来，让我们看看 IS NOT NULL 运算符的用法，它用来列出所有 SALARY 不为 NULL 的记录：

sqlite> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM COMPANY

WHERE SALARY IS NOT NULL;

上面的 SQLite 语句将产生下面的结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

下面是 IS NULL 运算符的用法，将列出所有 SALARY 为 NULL 的记录：

sqlite> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM COMPANY

WHERE SALARY IS NULL;

上面的 SQLite 语句将产生下面的结果：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

6 Kim 22

7 James 24

# 别名

可以暂时把表或列重命名为另一个名字，这被称为别名。

使用表别名是指在一个特定的 SQLite 语句中重命名表。重命名是临时的改变，在数据库中实际的表的名称不会改变。

列别名用来为某个特定的 SQLite 语句重命名表中的列。

## 语法

表 别名的基本语法如下：

SELECT column1, column2....

FROM table\_name AS alias\_name

WHERE [condition];

列 别名的基本语法如下：

SELECT column\_name AS alias\_name

FROM table\_name

WHERE [condition];

## 实例

假设有下面两个表，（1）COMPANY 表如下所示：

sqlite> select \* from COMPANY;

ID NAME AGE ADDRESS SALARY

---------- -------------------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

（2）另一个表是 DEPARTMENT，如下所示：

ID DEPT EMP\_ID

---------- -------------------- ----------

1 IT Billing 1

2 Engineering 2

3 Finance 7

4 Engineering 3

5 Finance 4

6 Engineering 5

7 Finance 6

现在，下面是 表别名 的用法，在这里我们使用 C 和 D 分别作为 COMPANY 和 DEPARTMENT 表的别名：

sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT

FROM COMPANY AS C, DEPARTMENT AS D

WHERE C.ID = D.EMP\_ID;

上面的 SQLite 语句将产生下面的结果：

ID NAME AGE DEPT

---------- ---------- ---------- ----------

1 Paul 32 IT Billing

2 Allen 25 Engineerin

3 Teddy 23 Engineerin

4 Mark 25 Finance

5 David 27 Engineerin

6 Kim 22 Finance

7 James 24 Finance

列别名 的实例，在这里 COMPANY\_ID 是 ID 列的别名，COMPANY\_NAME 是 name 列的别名：

sqlite> SELECT C.ID AS COMPANY\_ID, C.NAME AS COMPANY\_NAME, C.AGE, D.DEPT

FROM COMPANY AS C, DEPARTMENT AS D

WHERE C.ID = D.EMP\_ID;

上面的 SQLite 语句将产生下面的结果：

COMPANY\_ID COMPANY\_NAME AGE DEPT

---------- ------------ ---------- ----------

1 Paul 32 IT Billing

2 Allen 25 Engineerin

3 Teddy 23 Engineerin

4 Mark 25 Finance

5 David 27 Engineerin

6 Kim 22 Finance

7 James 24 Finance

# Trigger触发器

SQLite 触发器（Trigger）是数据库的回调函数，它会在指定的数据库事件发生时自动执行/调用。

## 要点

SQLite 的触发器（Trigger）可以指定在特定的数据库表发生 DELETE、INSERT 或 UPDATE 时触发，或在一个或多个指定表的列发生更新时触发。

SQLite 只支持 FOR EACH ROW 触发器（Trigger），没有 FOR EACH STATEMENT 触发器（Trigger）。因此，明确指定 FOR EACH ROW 是可选的。

WHEN 子句和触发器（Trigger）动作可能访问使用表单 NEW.column-name 和 OLD.column-name 的引用插入、删除或更新的行元素，其中 column-name 是从与触发器关联的表的列的名称。

如果提供 WHEN 子句，则只针对 WHEN 子句为真的指定行执行 SQL 语句。如果没有提供 WHEN 子句，则针对所有行执行 SQL 语句。

BEFORE 或 AFTER 关键字决定何时执行触发器动作，决定是在关联行的插入、修改或删除之前或者之后执行触发器动作。

当触发器相关联的表删除时，自动删除触发器（Trigger）。

要修改的表必须存在于同一数据库中，作为触发器被附加的表或视图，且必须只使用 tablename，而不是 database.tablename。

一个特殊的 SQL 函数 RAISE() 可用于触发器程序内抛出异常。

## 语法

创建 触发器（Trigger） 的基本语法如下：

CREATE TRIGGER trigger\_name [BEFORE|AFTER] event\_name

ON table\_name

BEGIN

-- Trigger logic goes here....

END;

在这里，event\_name 可以是在所提到的表 table\_name 上的 INSERT、DELETE 和 UPDATE 数据库操作。可以在表名后选择指定 FOR EACH ROW。

以下是在 UPDATE 操作上在表的一个或多个指定列上创建触发器（Trigger）的语法：

CREATE TRIGGER trigger\_name [BEFORE|AFTER] UPDATE OF column\_name

ON table\_name

BEGIN

-- Trigger logic goes here....

END;

## 实例

假设一个情况，要为被插入到新创建的 COMPANY 表（如果已经存在，则删除重新创建）中的每一个记录保持审计试验：

sqlite> CREATE TABLE COMPANY(

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

为了保持审计试验，我们将创建一个名为 AUDIT 的新表。每当 COMPANY 表中有一个新的记录项时，日志消息将被插入其中：

sqlite> CREATE TABLE AUDIT(

EMP\_ID INT NOT NULL,

ENTRY\_DATE TEXT NOT NULL

);

在这里，AUDIT 记录的EMP\_ID 是来自 COMPANY 表的 ENTRY\_DATE 将保持 COMPANY 中记录被创建时的时间戳。所以，现在 COMPANY 表上创建一个触发器，如下所示：

sqlite> CREATE TRIGGER audit\_log AFTER INSERT

ON COMPANY

BEGIN

INSERT INTO AUDIT(EMP\_ID, ENTRY\_DATE) VALUES (new.ID, datetime('now'));

END;

现在，我们将开始在 COMPANY 表中插入记录，这将导致在 AUDIT 表中创建一个审计日志记录。因此，让我们在 COMPANY 表中创建一个记录，如下所示：

sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Paul', 32, 'California', 20000.00 );

这将在 COMPANY 表中创建如下一个记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

同时，将在 AUDIT 表中创建一个记录。这个纪录是触发器的结果，这是在 COMPANY 表上的 INSERT 操作上创建的触发器（Trigger）。类似的，可以根据需要在 UPDATE 和 DELETE 操作上创建触发器（Trigger）。

EMP\_ID ENTRY\_DATE

---------- -------------------

1 2013-04-05 06:26:00

## 列出

可以从 sqlite\_master 表中列出所有触发器，如下所示：

sqlite> SELECT name FROM sqlite\_master

WHERE type = 'trigger';

上面的 SQLite 语句只会列出一个条目，如下：

name

----------

audit\_log

如果想要列出特定表上的触发器，则使用 AND 子句连接表名，如下所示：

sqlite> SELECT name FROM sqlite\_master

WHERE type = 'trigger' AND tbl\_name = 'COMPANY';

上面的 SQLite 语句只会列出一个条目，如下：

name

----------

audit\_log

## 删除

下面是 DROP 命令，可用于删除已有的触发器：

sqlite> DROP TRIGGER trigger\_name;

# Index索引

索引（Index）是一种特殊的查找表，数据库搜索引擎用来加快数据检索。简单地说，索引是一个指向表中数据的指针。一个数据库中的索引与一本书后边的索引是非常相似的。

例如，如果想在一本讨论某个话题的书中引用所有页面，首先需要指向索引，索引按字母顺序列出了所有主题，然后指向一个或多个特定的页码。

索引有助于加快 SELECT 查询和 WHERE 子句，但它会减慢使用 UPDATE 和 INSERT 语句时的数据输入。索引可以创建或删除，但不会影响数据。

使用 CREATE INDEX 语句创建索引，它允许命名索引，指定表及要索引的一列或多列，并指示索引是升序排列还是降序排列。

索引也可以是唯一的，与 UNIQUE 约束类似，在列上或列组合上防止重复条目。

## 命令

CREATE INDEX index\_name ON table\_name;

### 单列索引

单列索引是一个只基于表的一个列上创建的索引。基本语法如下：

CREATE INDEX index\_name

ON table\_name (column\_name);

### 唯一索引

使用唯一索引不仅是为了性能，同时也为了数据的完整性。唯一索引不允许任何重复的值插入到表中。基本语法如下：

CREATE UNIQUE INDEX index\_name

on table\_name (column\_name);

### 组合索引

组合索引是基于一个表的两个或多个列上创建的索引。基本语法如下：

CREATE INDEX index\_name

on table\_name (column1, column2);

是否要创建一个单列索引还是组合索引，要考虑到在作为查询过滤条件的 WHERE 子句中使用非常频繁的列。

如果值使用到一个列，则选择使用单列索引。如果在作为过滤的 WHERE 子句中有两个或多个列经常使用，则选择使用组合索引。

### 隐式索引

隐式索引是在创建对象时，由数据库服务器自动创建的索引。索引自动创建为主键约束和唯一约束。

实例

下面是一个例子，在 COMPANY 表的 salary 列上创建一个索引：

sqlite> CREATE INDEX salary\_index ON COMPANY (salary);

现在，使用 .indices 命令列出 COMPANY 表上所有可用的索引，如下所示：

sqlite> .indices COMPANY

这将产生如下结果，其中 sqlite\_autoindex\_COMPANY\_1 是创建表时创建的隐式索引。

salary\_index

sqlite\_autoindex\_COMPANY\_1

可以列出数据库范围的所有索引，如下所示：

sqlite> SELECT \* FROM sqlite\_master WHERE type = 'index';

DROP INDEX 命令

一个索引可以使用 SQLite 的 DROP 命令删除。当删除索引时应特别注意，因为性能可能会下降或提高。

DROP INDEX index\_name;

可以使用下面的语句来删除之前创建的索引：

sqlite> DROP INDEX salary\_index;

## 准则

虽然索引的目的在于提高数据库的性能，但这里有几个情况需要避免使用索引。使用索引时，应重新考虑下列准则：

索引不应该使用在较小的表上。

索引不应该使用在有频繁的大批量的更新或插入操作的表上。

索引不应该使用在含有大量的 NULL 值的列上。

索引不应该使用在频繁操作的列上。

# Indexed By

"INDEXED BY index-name" 子句规定必须需要命名的索引来查找前面表中值。如果索引名 index-name 不存在或不能用于查询，然后 SQLite 语句的准备失败。

"NOT INDEXED" 子句规定当访问前面的表（包括由 UNIQUE 和 PRIMARY KEY 约束创建的隐式索引）时，没有使用索引。

然而，即使指定了 "NOT INDEXED"，INTEGER PRIMARY KEY 仍然可以被用于查找条目。

## 语法

下面是 INDEXED BY 子句的语法，它可以与 DELETE、UPDATE 或 SELECT 语句一起使用：

SELECT|DELETE|UPDATE column1, column2...

INDEXED BY (index\_name)

table\_name

WHERE (CONDITION);

## 实例

假设有表 COMPANY，我们将创建一个索引，并用它进行 INDEXED BY 操作。

sqlite> CREATE INDEX salary\_index ON COMPANY(salary);

sqlite>

现在使用 INDEXED BY 子句从表 COMPANY 中选择数据，如下所示：

sqlite> SELECT \* FROM COMPANY INDEXED BY salary\_index WHERE salary > 5000;

# Alter

SQLite 的 ALTER TABLE 命令不通过执行一个完整的转储和数据的重载来修改已有的表。使用 ALTER TABLE 语句重命名表，使用 ALTER TABLE 语句还可以在已有的表中添加额外的列。

在 SQLite 中，除了重命名表和在已有的表中添加列，ALTER TABLE 命令不支持其他操作。

## 语法

用来重命名已有的表的 ALTER TABLE 的基本语法如下：

ALTER TABLE database\_name.table\_name RENAME TO new\_table\_name;

用来在已有的表中添加一个新的列的 ALTER TABLE 的基本语法如下：

ALTER TABLE database\_name.table\_name ADD COLUMN column\_def...;

## 实例

假设 COMPANY 表有如下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，尝试使用 ALTER TABLE 语句重命名该表，如下所示：

sqlite> ALTER TABLE COMPANY RENAME TO OLD\_COMPANY;

上面的 SQLite 语句将重命名 COMPANY 表为 OLD\_COMPANY。现在，让我们尝试在 OLD\_COMPANY 表中添加一个新的列，如下所示：

sqlite> ALTER TABLE OLD\_COMPANY ADD COLUMN SEX char(1);

现在，COMPANY 表已经改变，使用 SELECT 语句输出如下：

ID NAME AGE ADDRESS SALARY SEX

---------- ---------- ---------- ---------- ---------- ---

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

请注意，新添加的列是以 NULL 值来填充的。

# Truncate Table

在 SQLite 中，并没有 TRUNCATE TABLE 命令，但可以使用 SQLite 的 DELETE 命令从已有的表中删除全部的数据，但建议使用 DROP TABLE 命令删除整个表，然后再重新创建一遍。

## 语法

DELETE 命令的基本语法如下：

sqlite> DELETE FROM table\_name;

DROP TABLE 的基本语法如下：

sqlite> DROP TABLE table\_name;

如果您使用 DELETE TABLE 命令删除所有记录，建议使用 VACUUM 命令清除未使用的空间。

## 实例

假设 COMPANY 表有如下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

下面为删除上表记录的实例：

SQLite> DELETE FROM COMPANY;

SQLite> VACUUM;

现在，COMPANY 表中的记录完全被删除，使用 SELECT 语句将没有任何输出。

# View视图

视图（View）只不过是通过相关的名称存储在数据库中的一个 SQLite 语句。视图（View）实际上是一个以预定义的 SQLite 查询形式存在的表的组合。

视图（View）可以包含一个表的所有行或从一个或多个表选定行。视图（View）可以从一个或多个表创建，这取决于要创建视图的 SQLite 查询。

视图（View）是一种虚表，允许用户实现以下几点：

用户或用户组查找结构数据的方式更自然或直观。

限制数据访问，用户只能看到有限的数据，而不是完整的表。

汇总各种表中的数据，用于生成报告。

SQLite 视图是只读的，因此可能无法在视图上执行 DELETE、INSERT 或 UPDATE 语句。但是可以在视图上创建一个触发器，当尝试 DELETE、INSERT 或 UPDATE 视图时触发，需要做的动作在触发器内容中定义。

## 创建

SQLite 的视图是使用 CREATE VIEW 语句创建的。SQLite 视图可以从一个单一的表、多个表或其他视图创建。

CREATE VIEW 的基本语法如下：

CREATE [TEMP | TEMPORARY] VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE [condition];

可以在 SELECT 语句中包含多个表，这与在正常的 SQL SELECT 查询中的方式非常相似。如果使用了可选的 TEMP 或 TEMPORARY 关键字，则将在临时数据库中创建视图。

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，下面是一个从 COMPANY 表创建视图的实例。视图只从 COMPANY 表中选取几列：

sqlite> CREATE VIEW COMPANY\_VIEW AS

SELECT ID, NAME, AGE

FROM COMPANY;

现在，可以查询 COMPANY\_VIEW，与查询实际表的方式类似。下面是实例：

sqlite> SELECT \* FROM COMPANY\_VIEW;

这将产生以下结果:

ID NAME AGE

---------- ---------- ----------

1 Paul 32

2 Allen 25

3 Teddy 23

4 Mark 25

5 David 27

6 Kim 22

7 James 24

## 删除

要删除视图，只需使用带有 view\_name 的 DROP VIEW 语句。DROP VIEW 的基本语法如下：

sqlite> DROP VIEW view\_name;

下面的命令将删除在前面创建的 COMPANY\_VIEW 视图：

sqlite> DROP VIEW COMPANY\_VIEW;

# Transaction

事务（Transaction）是一个对数据库执行工作单元。事务（Transaction）是以逻辑顺序完成的工作单位或序列，可以是由用户手动操作完成，也可以是由某种数据库程序自动完成。

事务（Transaction）是指一个或多个更改数据库的扩展。例如，如果正在创建一个记录或者更新一个记录或者从表中删除一个记录，那么正在该表上执行事务。重要的是要控制事务以确保数据的完整性和处理数据库错误。

实际上，可以把许多的 SQLite 查询联合成一组，把所有这些放在一起作为事务的一部分进行执行。

事务（Transaction）具有以下四个标准属性，通常根据首字母缩写为 ACID：

原子性（Atomicity）,确保工作单位内的所有操作都成功完成，否则，事务会在出现故障时终止，之前的操作也会回滚到以前的状态。

一致性（Consistency),确保数据库在成功提交的事务上正确地改变状态。

隔离性（Isolation）,使事务操作相互独立和透明。

持久性（Durability）,确保已提交事务的结果或效果在系统发生故障的情况下仍然存在。

使用下面的命令来控制事务：

BEGIN TRANSACTION：开始事务处理。

COMMIT：保存更改，或者可以使用 END TRANSACTION 命令。

ROLLBACK：回滚所做的更改。

事务控制命令只与 DML 命令 INSERT、UPDATE 和 DELETE 一起使用。他们不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

## BEGIN TRANSACTION

事务（Transaction）可以使用 BEGIN TRANSACTION 命令或简单的 BEGIN 命令来启动。此类事务通常会持续执行下去，直到遇到下一个 COMMIT 或 ROLLBACK 命令。不过在数据库关闭或发生错误时，事务处理也会回滚。以下是启动一个事务的简单语法：

BEGIN;

or

BEGIN TRANSACTION;

## COMMIT

COMMIT 命令是用于把事务调用的更改保存到数据库中的事务命令。

COMMIT 命令把自上次 COMMIT 或 ROLLBACK 命令以来的所有事务保存到数据库。

COMMIT 命令的语法如下：

COMMIT;

or

END TRANSACTION;

## ROLLBACK

ROLLBACK 命令是用于撤消尚未保存到数据库的事务的事务命令。

ROLLBACK 命令只能用于撤销自上次发出 COMMIT 或 ROLLBACK 命令以来的事务。

ROLLBACK 命令的语法如下：

ROLLBACK;

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，让开始一个事务，并从表中删除 age = 25 的记录，最后，我们使用 ROLLBACK 命令撤消所有的更改。

sqlite> BEGIN;

sqlite> DELETE FROM COMPANY WHERE AGE = 25;

sqlite> ROLLBACK;

检查 COMPANY 表，仍然有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，让开始另一个事务，从表中删除 age = 25 的记录，最后我们使用 COMMIT 命令提交所有的更改。

sqlite> BEGIN;

sqlite> DELETE FROM COMPANY WHERE AGE = 25;

sqlite> COMMIT;

检查 COMPANY 表，有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

3 Teddy 23 Norway 20000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

# 子查询

子查询或内部查询或嵌套查询是在另一个 SQLite 查询内嵌入在 WHERE 子句中的查询。

使用子查询返回的数据将被用在主查询中作为条件，以进一步限制要检索的数据。

子查询可以与 SELECT、INSERT、UPDATE 和 DELETE 语句一起使用，可伴随着使用运算符如 =、<、>、>=、<=、IN、BETWEEN 等。

## 规则

子查询必须用括号括起来。

子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。

ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使用 GROUP BY，功能与 ORDER BY 相同。

子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。

BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内使用。

## SELECT 中使用

子查询通常与 SELECT 语句一起使用。基本语法如下：

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

### 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，检查 SELECT 语句中的子查询使用：

sqlite> SELECT \*

FROM COMPANY

WHERE ID IN (SELECT ID

FROM COMPANY

WHERE SALARY > 45000) ;

这将产生以下结果:

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

## INSERT中使用

子查询也可以与 INSERT 语句一起使用。INSERT 语句使用子查询返回的数据插入到另一个表中。在子查询中所选择的数据可以用任何字符、日期或数字函数修改。

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ]

### 实例

假设 COMPANY\_BKP 的结构与 COMPANY 表相似，且可使用相同的 CREATE TABLE 进行创建，只是表名改为 COMPANY\_BKP。现在把整个 COMPANY 表复制到 COMPANY\_BKP，语法如下：

sqlite> INSERT INTO COMPANY\_BKP

SELECT \* FROM COMPANY

WHERE ID IN (SELECT ID

FROM COMPANY) ;

## UPDATE中使用

子查询可以与 UPDATE 语句结合使用。当通过 UPDATE 语句使用子查询时，表中单个或多个列被更新。

UPDATE table

SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

### 实例

假设，有 COMPANY\_BKP 表，是 COMPANY 表的备份。下面的实例把 COMPANY 表中所有 AGE 大于或等于 27 的客户的 SALARY 更新为原来的 0.50 倍：

sqlite> UPDATE COMPANY

SET SALARY = SALARY \* 0.50

WHERE AGE IN (SELECT AGE FROM COMPANY\_BKP

WHERE AGE >= 27 );

这将影响两行，最后 COMPANY 表中的记录如下：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 10000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 42500.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

## DELETE中使用

子查询可以与 DELETE 语句结合使用，就像上面提到的其他语句一样。

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

### 实例

假设，有 COMPANY\_BKP 表，是 COMPANY 表的备份。下面的实例删除 COMPANY 表中所有 AGE 大于或等于 27 的客户记录：

sqlite> DELETE FROM COMPANY

WHERE AGE IN (SELECT AGE FROM COMPANY\_BKP

WHERE AGE > 27 );

这将影响两行，最后 COMPANY 表中的记录如下：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 42500.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

# Autoincrement

SQLite 的 AUTOINCREMENT 是一个关键字，用于表中的字段值自动递增。可以在创建表时在特定的列名称上使用 AUTOINCREMENT 关键字实现该字段值的自动增加。

关键字 AUTOINCREMENT 只能用于整型（INTEGER）字段。

## 语法

AUTOINCREMENT 关键字的基本用法如下：

CREATE TABLE table\_name(

column1 INTEGER AUTOINCREMENT,

column2 datatype,

column3 datatype,

.....

columnN datatype,

);

## 实例

假设要创建的 COMPANY 表如下所示：

sqlite> CREATE TABLE COMPANY(

ID INTEGER PRIMARY KEY AUTOINCREMENT,

NAME TEXT NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL

);

现在，向 COMPANY 表插入以下记录：

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ('Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ('Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'James', 24, 'Houston', 10000.00 );

这将向 COMPANY 表插入 7 个元组，此时 COMPANY 表的记录如下：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

# 注入

如果站点允许用户通过网页输入，并将输入内容插入到 SQLite 数据库中，这个时候就面临着一个被称为 SQL 注入的安全问题。

注入通常在请求用户输入时发生，比如需要用户输入姓名，但用户却输入了一个 SQLite 语句，而这语句就会在不知不觉中在数据库上运行。

永远不要相信用户提供的数据，所以只处理通过验证的数据，这项规则是通过模式匹配来完成的。在下面的实例中，用户名 username 被限制为字母数字字符或者下划线，长度必须在 8 到 20 个字符之间 - 请根据需要修改这些规则。

if (preg\_match("/^\w{8,20}$/", $\_GET['username'], $matches)){

$db = new SQLiteDatabase('filename');

$result = @$db->query("SELECT \* FROM users WHERE username=$matches[0]");

}else{

echo "username not accepted";

}

为了演示这个问题，假设考虑此摘录：To demonstrate the problem, consider this excerpt:

$name = "Qadir'; DELETE FROM users;";

@$db->query("SELECT \* FROM users WHERE username='{$name}'");

函数调用是为了从用户表中检索 name 列与用户指定的名称相匹配的记录。正常情况下，$name 只包含字母数字字符或者空格，比如字符串 ilia。但在这里，向 $name 追加了一个全新的查询，这个对数据库的调用将会造成灾难性的问题：注入的 DELETE 查询会删除 users 的所有记录。

虽然已经存在有不允许查询堆叠或在单个函数调用中执行多个查询的数据库接口，如果尝试堆叠查询，则会调用失败，但 SQLite 和 PostgreSQL 里仍进行堆叠查询，即执行在一个字符串中提供的所有查询，这会导致严重的安全问题。

防止 SQL 注入

在脚本语言中，比如 PERL 和 PHP，您可以巧妙地处理所有的转义字符。编程语言 PHP 提供了字符串函数 sqlite\_escape\_string() 来转义对于 SQLite 来说比较特殊的输入字符。

if (get\_magic\_quotes\_gpc())

{

$name = sqlite\_escape\_string($name);

}

$result = @$db->query("SELECT \* FROM users WHERE username='{$name}'");

虽然编码使得插入数据变得安全，但是它会呈现简单的文本比较，在查询中，对于包含二进制数据的列，LIKE 子句是不可用的。

请注意，addslashes() 不应该被用在 SQLite 查询中引用字符串，它会在检索数据时导致奇怪的结果。

# Explain解释

在 SQLite 语句之前，可以使用 "EXPLAIN" 关键字或 "EXPLAIN QUERY PLAN" 短语，用于描述表的细节。

如果省略了 EXPLAIN 关键字或短语，任何的修改都会引起 SQLite 语句的查询行为，并返回有关 SQLite 语句如何操作的信息。

来自 EXPLAIN 和 EXPLAIN QUERY PLAN 的输出只用于交互式分析和排除故障。

输出格式的细节可能会随着 SQLite 版本的不同而有所变化。

应用程序不应该使用 EXPLAIN 或 EXPLAIN QUERY PLAN，因为其确切的行为是可变的且只有部分会被记录。

## 语法

EXPLAIN 的语法如下：

EXPLAIN [SQLite Query]

EXPLAIN QUERY PLAN 的语法如下：

EXPLAIN QUERY PLAN [SQLite Query]

## 实例

假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

现在，检查 SELECT 语句中的 Explain 使用：

sqlite> EXPLAIN SELECT \* FROM COMPANY WHERE Salary &gt= 20000;

这将产生以下结果：

addr opcode p1 p2 p3

---------- ---------- ---------- ---------- ----------

0 Goto 0 19

1 Integer 0 0

2 OpenRead 0 8

3 SetNumColu 0 5

4 Rewind 0 17

5 Column 0 4

6 RealAffini 0 0

7 Integer 20000 0

8 Lt 357 16 collseq(BI

9 Rowid 0 0

10 Column 0 1

11 Column 0 2

12 Column 0 3

13 Column 0 4

14 RealAffini 0 0

15 Callback 5 0

16 Next 0 5

17 Close 0 0

18 Halt 0 0

19 Transactio 0 0

20 VerifyCook 0 38

21 Goto 0 1

22 Noop 0 0

现在，检查 SELECT 语句中的 Explain Query Plan 使用：

SQLite> EXPLAIN QUERY PLAN SELECT \* FROM COMPANY WHERE Salary &gt= 20000;

order from detail

---------- ---------- -------------

0 0 TABLE COMPANY

# Vacuum

VACUUM 命令通过复制主数据库中的内容到一个临时数据库文件，然后清空主数据库，并从副本中重新载入原始的数据库文件。这消除了空闲页，把表中的数据排列为连续的，另外会清理数据库文件结构。

如果表中没有明确的整型主键（INTEGER PRIMARY KEY），VACUUM 命令可能会改变表中条目的行 ID（ROWID）。VACUUM 命令只适用于主数据库，附加的数据库文件是不可能使用 VACUUM 命令。

如果有一个活动的事务，VACUUM 命令就会失败。VACUUM 命令是一个用于内存数据库的任何操作。由于 VACUUM 命令从头开始重新创建数据库文件，所以 VACUUM 也可以用于修改许多数据库特定的配置参数。

## 手动 VACUUM

下面是在命令提示符中对整个数据库发出 VACUUM 命令的语法：

$sqlite3 database\_name "VACUUM;"

也可以在 SQLite 提示符中运行 VACUUM，如下所示：

sqlite> VACUUM;

也可以在特定的表上运行 VACUUM，如下所示：

sqlite> VACUUM table\_name;

## 自动 VACUUM

SQLite 的 Auto-VACUUM 与 VACUUM 不大一样，它只是把空闲页移到数据库末尾，从而减小数据库大小。通过这样做，它可以明显地把数据库碎片化，而 VACUUM 则是反碎片化。所以 Auto-VACUUM 只会让数据库更小。

在 SQLite 提示符中，可以通过下面的编译运行，启用/禁用 SQLite 的 Auto-VACUUM：

sqlite> PRAGMA auto\_vacuum = NONE; -- 0 means disable auto vacuum

sqlite> PRAGMA auto\_vacuum = INCREMENTAL; -- 1 means enable incremental vacuum

sqlite> PRAGMA auto\_vacuum = FULL; -- 2 means enable full auto vacuum

可以从命令提示符中运行下面的命令来检查 auto-vacuum 设置：

$sqlite3 database\_name "PRAGMA auto\_vacuum;"

# 日期 & 时间

SQLite 支持以下五个日期和时间函数：

序号 函数 实例

1 date(timestring, modifier, modifier, ...) 以 YYYY-MM-DD 格式返回日期。

2 time(timestring, modifier, modifier, ...) 以 HH:MM:SS 格式返回时间。

3 datetime(timestring, modifier, modifier, ...) 以 YYYY-MM-DD HH:MM:SS 格式返回。

4 julianday(timestring, modifier, modifier, ...) 这将返回从格林尼治时间的公元前 4714 年 11 月 24 日正午算起的天数。

5 strftime(format, timestring, modifier, modifier, ...) 这将根据第一个参数指定的格式字符串返回格式化的日期。具体格式见下边讲解。

上述五个日期和时间函数把时间字符串作为参数。时间字符串后跟零个或多个 modifier 修饰符。strftime() 函数也可以把格式字符串 format 作为其第一个参数。下面将详细讲解不同类型的时间字符串和修饰符。

## 时间字符串

一个时间字符串可以采用下面任何一种格式：

序号 时间字符串 实例

1 YYYY-MM-DD 2010-12-30

2 YYYY-MM-DD HH:MM 2010-12-30 12:10

3 YYYY-MM-DD HH:MM:SS.SSS 2010-12-30 12:10:04.100

4 MM-DD-YYYY HH:MM 30-12-2010 12:10

5 HH:MM 12:10

6 YYYY-MM-DDTHH:MM 2010-12-30 12:10

7 HH:MM:SS 12:10:01

8 YYYYMMDD HHMMSS 20101230 121001

9 now 2013-05-07

可以使用 "T" 作为分隔日期和时间的文字字符。

## 修饰符（Modifier）

时间字符串后边可跟着零个或多个的修饰符，这将改变有上述五个函数返回的日期和/或时间。任何上述五大功能返回时间。修饰符应从左到右使用，下面列出了可在 SQLite 中使用的修饰符：

NNN days

NNN hours

NNN minutes

NNN.NNNN seconds

NNN months

NNN years

start of month

start of year

start of day

weekday N

unixepoch

localtime

utc

## 格式化

SQLite 提供了非常方便的函数 strftime() 来格式化任何日期和时间。您可以使用以下的替换来格式化日期和时间：

替换 描述

%d 一月中的第几天，01-31

%f 带小数部分的秒，SS.SSS

%H 小时，00-23

%j 一年中的第几天，001-366

%J 儒略日数，DDDD.DDDD

%m 月，00-12

%M 分，00-59

%s 从 1970-01-01 算起的秒数

%S 秒，00-59

%w 一周中的第几天，0-6 (0 is Sunday)

%W 一年中的第几周，01-53

%Y 年，YYYY

%% % symbol

## 实例

现在使用 SQLite 提示符尝试不同的实例。下面是计算当前日期：

sqlite> SELECT date('now');

2013-05-07

下面是计算当前月份的最后一天：

sqlite> SELECT date('now','start of month','+1 month','-1 day');

2013-05-31

下面是计算给定 UNIX 时间戳 1092941466 的日期和时间：

sqlite> SELECT datetime(1092941466, 'unixepoch');

2004-08-19 18:51:06

下面是计算给定 UNIX 时间戳 1092941466 相对本地时区的日期和时间：

sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');

2004-08-19 11:51:06

下面是计算当前的 UNIX 时间戳：

sqlite> SELECT strftime('%s','now');

1367926057

下面是计算美国"独立宣言"签署以来的天数：

sqlite> SELECT julianday('now') - julianday('1776-07-04');

86504.4775830326

下面是计算从 2004 年某一特定时刻以来的秒数：

sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02:34:56');

295001572

下面是计算当年 10 月的第一个星期二的日期：

sqlite> SELECT date('now','start of year','+9 months','weekday 2');

2013-10-01

下面是计算从 UNIX 纪元算起的以秒为单位的时间（类似 strftime('%s','now') ，不同的是这里有包括小数部分）：

sqlite> SELECT (julianday('now') - 2440587.5)\*86400.0;

1367926077.12598

在 UTC 与本地时间值之间进行转换，当格式化日期时，使用 utc 或 localtime 修饰符，如下所示：

sqlite> SELECT time('12:00', 'localtime');

05:00:00

sqlite> SELECT time('12:00', 'utc');

19:00:00

# 常用函数

SQLite 有许多内置函数用于处理字符串或数字数据。下面列出了一些有用的 SQLite 内置函数，且所有函数都是大小写不敏感。欲了解更多查看 SQLite 的官方文档：

序号 函数 & 描述

1 SQLite COUNT 函数

SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。

2 SQLite MAX 函数

SQLite MAX 聚合函数允许我们选择某列的最大值。

3 SQLite MIN 函数

SQLite MIN 聚合函数允许我们选择某列的最小值。

4 SQLite AVG 函数

SQLite AVG 聚合函数计算某列的平均值。

5 SQLite SUM 函数

SQLite SUM 聚合函数允许为一个数值列计算总和。

6 SQLite RANDOM 函数

SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。

7 SQLite ABS 函数

SQLite ABS 函数返回数值参数的绝对值。

8 SQLite UPPER 函数

SQLite UPPER 函数把字符串转换为大写字母。

9 SQLite LOWER 函数

SQLite LOWER 函数把字符串转换为小写字母。

10 SQLite LENGTH 函数

SQLite LENGTH 函数返回字符串的长度。

11 SQLite sqlite\_version 函数

SQLite sqlite\_version 函数返回 SQLite 库的版本。

先假设 COMPANY 表有以下记录：

ID NAME AGE ADDRESS SALARY

---------- ---------- ---------- ---------- ----------

1 Paul 32 California 20000.0

2 Allen 25 Texas 15000.0

3 Teddy 23 Norway 20000.0

4 Mark 25 Rich-Mond 65000.0

5 David 27 Texas 85000.0

6 Kim 22 South-Hall 45000.0

7 James 24 Houston 10000.0

## COUNT

SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。下面是实例：

sqlite> SELECT count(\*) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

count(\*)

----------

7

## MAX

SQLite MAX 聚合函数允许我们选择某列的最大值。下面是实例：

sqlite> SELECT max(salary) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

max(salary)

-----------

85000.0

## MIN

SQLite MIN 聚合函数允许我们选择某列的最小值。下面是实例：

sqlite> SELECT min(salary) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

min(salary)

-----------

10000.0

## AVG

SQLite AVG 聚合函数计算某列的平均值。下面是实例：

sqlite> SELECT avg(salary) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

avg(salary)

----------------

37142.8571428572

## SUM

SQLite SUM 聚合函数允许为一个数值列计算总和。下面是实例：

sqlite> SELECT sum(salary) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

sum(salary)

-----------

260000.0

## RANDOM

SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。下面是实例：

sqlite> SELECT random() AS Random;

上面的 SQLite SQL 语句将产生以下结果：

Random

-------------------

5876796417670984050

## ABS

SQLite ABS 函数返回数值参数的绝对值。下面是实例：

sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");

上面的 SQLite SQL 语句将产生以下结果：

abs(5) abs(-15) abs(NULL) abs(0) abs("ABC")

---------- ---------- ---------- ---------- ----------

5 15 0 0.0

## UPPER

SQLite UPPER 函数把字符串转换为大写字母。下面是实例：

sqlite> SELECT upper(name) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

upper(name)

-----------

PAUL

ALLEN

TEDDY

MARK

DAVID

KIM

JAMES

## LOWER

SQLite LOWER 函数把字符串转换为小写字母。下面是实例：

sqlite> SELECT lower(name) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

lower(name)

-----------

paul

allen

teddy

mark

david

kim

james

## LENGTH

SQLite LENGTH 函数返回字符串的长度。下面是实例：

sqlite> SELECT name, length(name) FROM COMPANY;

上面的 SQLite SQL 语句将产生以下结果：

NAME length(name)

---------- ------------

Paul 4

Allen 5

Teddy 5

Mark 4

David 5

Kim 3

James 5

## sqlite\_version

SQLite sqlite\_version 函数返回 SQLite 库的版本。下面是实例：

sqlite> SELECT sqlite\_version() AS 'SQLite Version';

上面的 SQLite SQL 语句将产生以下结果：

SQLite Version

--------------

3.6.20

# C/C++

在 C/C++ 程序中使用 SQLite 之前，需要确保机器上已经有 SQLite 库。

以下是重要的 C&C++ / SQLite 接口程序，了解更多细节，请查看 SQLite 官方文档。

序号 API & 描述

1 sqlite3\_open(const char \*filename, sqlite3 \*\*ppDb)

该例程打开一个指向 SQLite 数据库文件的连接，返回一个用于其他 SQLite 程序的数据库连接对象。

如果 filename 参数是 NULL 或 ':memory:'，那么 sqlite3\_open() 将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。

如果文件名 filename 不为 NULL，那么 sqlite3\_open() 将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，sqlite3\_open() 将创建一个新的命名为该名称的数据库文件并打开。

2 sqlite3\_exec(sqlite3\*, const char \*sql, sqlite\_callback, void \*data, char \*\*errmsg)

该例程提供了一个执行 SQL 命令的快捷方式，SQL 命令由 sql 参数提供，可以由多个 SQL 命令组成。在这里，第一个参数 sqlite3 是打开的数据库对象，sqlite\_callback 是一个回调，data 作为其第一个参数，errmsg 将被返回用来获取程序生成的任何错误。

sqlite3\_exec() 程序解析并执行由 sql 参数所给的每个命令，直到字符串结束或者遇到错误为止。

3 sqlite3\_close(sqlite3\*)

该例程关闭之前调用 sqlite3\_open() 打开的数据库连接。所有与连接相关的语句都应在连接关闭之前完成。

如果还有查询没有完成，sqlite3\_close() 将返回 SQLITE\_BUSY 禁止关闭的错误消息。

## 连接数据库

下面的 C 代码段显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

#include <stdio.h>

#include <sqlite3.h>

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stderr, "Opened database successfully\n");

}

sqlite3\_close(db);

}

现在，来编译和运行上面的程序，在当前目录中创建我们的数据库 test.db。您可以根据需要改变路径。

$gcc test.c -l sqlite3

$./a.out

Opened database successfully

如果要使用 C++ 源代码，可以按照下列所示编译代码：

$g++ test.c -l sqlite3

在这里，把我们的程序链接上 sqlite3 库，以便向 C 程序提供必要的函数。这将在您的目录下创建一个数据库文件 test.db，您将得到如下结果：

-rwxr-xr-x. 1 root root 7383 May 8 02:06 a.out

-rw-r--r--. 1 root root 323 May 8 02:05 test.c

-rw-r--r--. 1 root root 0 May 8 02:06 test.db

## 创建表

下面的 C 代码段将用于在先前创建的数据库中创建一个表：

#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>

static int callback(void \*NotUsed, int argc, char \*\*argv, char \*\*azColName){

int i;

for(i=0; i<argc; i++){

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");

}

printf("\n");

return 0;

}

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

char \*sql;

/\* Open database \*/

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stdout, "Opened database successfully\n");

}

/\* Create SQL statement \*/

sql = "CREATE TABLE COMPANY(" \

"ID INT PRIMARY KEY NOT NULL," \

"NAME TEXT NOT NULL," \

"AGE INT NOT NULL," \

"ADDRESS CHAR(50)," \

"SALARY REAL );";

/\* Execute SQL statement \*/

rc = sqlite3\_exec(db, sql, callback, 0, &zErrMsg);

if( rc != SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}else{

fprintf(stdout, "Table created successfully\n");

}

sqlite3\_close(db);

return 0;

}

上述程序编译和执行时，它会在 test.db 文件中创建 COMPANY 表，最终文件列表如下所示：

-rwxr-xr-x. 1 root root 9567 May 8 02:31 a.out

-rw-r--r--. 1 root root 1207 May 8 02:31 test.c

-rw-r--r--. 1 root root 3072 May 8 02:31 test.db

## INSERT

下面的 C 代码段显示了如何在上面创建的 COMPANY 表中创建记录：

#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>

static int callback(void \*NotUsed, int argc, char \*\*argv, char \*\*azColName){

int i;

for(i=0; i<argc; i++){

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");

}

printf("\n");

return 0;

}

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

char \*sql;

/\* Open database \*/

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stderr, "Opened database successfully\n");

}

/\* Create SQL statement \*/

sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " \

"VALUES (1, 'Paul', 32, 'California', 20000.00 ); " \

"INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " \

"VALUES (2, 'Allen', 25, 'Texas', 15000.00 ); " \

"INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \

"VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );" \

"INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \

"VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";

/\* Execute SQL statement \*/

rc = sqlite3\_exec(db, sql, callback, 0, &zErrMsg);

if( rc != SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}else{

fprintf(stdout, "Records created successfully\n");

}

sqlite3\_close(db);

return 0;

}

上述程序编译和执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

Opened database successfully

Records created successfully

## SELECT

在我们开始讲解获取记录的实例之前，让我们先了解下回调函数的一些细节，这将在我们的实例使用到。这个回调提供了一个从 SELECT 语句获得结果的方式。它声明如下：

typedef int (\*sqlite3\_callback)(

void\*, /\* Data provided in the 4th argument of sqlite3\_exec() \*/

int, /\* The number of columns in row \*/

char\*\*, /\* An array of strings representing fields in the row \*/

char\*\* /\* An array of strings representing column names \*/

);

如果上面的回调在 sqlite\_exec() 程序中作为第三个参数，那么 SQLite 将为 SQL 参数内执行的每个 SELECT 语句中处理的每个记录调用这个回调函数。

下面的 C 代码段显示了如何从前面创建的 COMPANY 表中获取并显示记录：

#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>

static int callback(void \*data, int argc, char \*\*argv, char \*\*azColName){

int i;

fprintf(stderr, "%s: ", (const char\*)data);

for(i=0; i<argc; i++){

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");

}

printf("\n");

return 0;

}

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

char \*sql;

const char\* data = "Callback function called";

/\* Open database \*/

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stderr, "Opened database successfully\n");

}

/\* Create SQL statement \*/

sql = "SELECT \* from COMPANY";

/\* Execute SQL statement \*/

rc = sqlite3\_exec(db, sql, callback, (void\*)data, &zErrMsg);

if( rc != SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}else{

fprintf(stdout, "Operation done successfully\n");

}

sqlite3\_close(db);

return 0;

}

上述程序编译和执行时，它会产生以下结果：

Opened database successfully

Callback function called: ID = 1

NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 20000.0

Callback function called: ID = 2

NAME = Allen

AGE = 25

ADDRESS = Texas

SALARY = 15000.0

Callback function called: ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0

Callback function called: ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0

Operation done successfully

## UPDATE

下面的 C 代码段显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>

static int callback(void \*data, int argc, char \*\*argv, char \*\*azColName){

int i;

fprintf(stderr, "%s: ", (const char\*)data);

for(i=0; i<argc; i++){

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");

}

printf("\n");

return 0;

}

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

char \*sql;

const char\* data = "Callback function called";

/\* Open database \*/

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stderr, "Opened database successfully\n");

}

/\* Create merged SQL statement \*/

sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1; " \

"SELECT \* from COMPANY";

/\* Execute SQL statement \*/

rc = sqlite3\_exec(db, sql, callback, (void\*)data, &zErrMsg);

if( rc != SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}else{

fprintf(stdout, "Operation done successfully\n");

}

sqlite3\_close(db);

return 0;

}

上述程序编译和执行时，它会产生以下结果：

Opened database successfully

Callback function called: ID = 1

NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 25000.0

Callback function called: ID = 2

NAME = Allen

AGE = 25

ADDRESS = Texas

SALARY = 15000.0

Callback function called: ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0

Callback function called: ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0

Operation done successfully

## DELETE

下面的 C 代码段显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>

static int callback(void \*data, int argc, char \*\*argv, char \*\*azColName){

int i;

fprintf(stderr, "%s: ", (const char\*)data);

for(i=0; i<argc; i++){

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");

}

printf("\n");

return 0;

}

int main(int argc, char\* argv[])

{

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

char \*sql;

const char\* data = "Callback function called";

/\* Open database \*/

rc = sqlite3\_open("test.db", &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

exit(0);

}else{

fprintf(stderr, "Opened database successfully\n");

}

/\* Create merged SQL statement \*/

sql = "DELETE from COMPANY where ID=2; " \

"SELECT \* from COMPANY";

/\* Execute SQL statement \*/

rc = sqlite3\_exec(db, sql, callback, (void\*)data, &zErrMsg);

if( rc != SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}else{

fprintf(stdout, "Operation done successfully\n");

}

sqlite3\_close(db);

return 0;

}

上述程序编译和执行时，它会产生以下结果：

Opened database successfully

Callback function called: ID = 1

NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 20000.0

Callback function called: ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0

Callback function called: ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0

Operation done successfully

补充说明

使用的过程根据使用的函数大致分为如下几个过程：

* sqlite3\_open()
* sqlite3\_prepare()
* sqlite3\_step()
* sqlite3\_column()
* sqlite3\_finalize()
* sqlite3\_close()

这几个过程是概念上的说法，而不完全是程序运行的过程，如sqlite3\_column()表示的是对查询获得一行里面的数据的列的各个操作统称，实际上在sqlite中并不存在这个函数。

1．  sqlite3\_open()：打开数据库

在操作数据库之前，首先要打开数据库。这个函数打开一个sqlite数据库文件的连接并且返回一个数据库连接对象。这个操作同时程序中的第一个调用的sqlite函数，同时也是其他sqlite api的先决条件。许多的sqlite接口函数都需要一个数据库连接对象的指针作为它们的第一个参数。

函数定义

int sqlite3\_open(

  const char \*filename,   /\* Database filename (UTF-8) \*/

  sqlite3 \*\*ppDb          /\* OUT: SQLite db handle \*/

);

int sqlite3\_open16(

  const void \*filename,   /\* Database filename (UTF-16) \*/

  sqlite3 \*\*ppDb          /\* OUT: SQLite db handle \*/

);

int sqlite3\_open\_v2(

  const char \*filename,   /\* Database filename (UTF-8) \*/

  sqlite3 \*\*ppDb,         /\* OUT: SQLite db handle \*/

  int flags,              /\* Flags \*/

  const char \*zVfs        /\* Name of VFS module to use \*/

);

说明：

假如这个要被打开的数据文件不存在，则一个同名的数据库文件将被创建。如果使用sqlite3\_open和sqlite3\_open\_v2的话，数据库将采用UTF-8的编码方式，sqlite3\_open16采用UTF-16的编码方式

返回值：

如果sqlite数据库被成功打开（或创建），将会返回SQLITE\_OK，否则将会返回错误码。Sqlite3\_errmsg()或者sqlite3\_errmsg16可以用于获得数据库打开错误码的英文描述，这两个函数定义为：

const char \*sqlite3\_errmsg(sqlite3\*);

const void \*sqlite3\_errmsg16(sqlite3\*);

无论是否成功打开数据库, 都应该使用 sqlite3\_close() 关闭数据库连接.

参数说明：

filename：需要被打开的数据库文件的文件名，在sqlite3\_open和sqlite3\_open\_v2中这个参数采用UTF-8编码，而在sqlite3\_open16中则采用UTF-16编码

ppDb：一个数据库连接句柄被返回到这个参数，即使发生错误。唯一的一场是如果sqlite不能分配内存来存放sqlite对象，ppDb将会被返回一个NULL值。

flags：作为数据库连接的额外控制的参数，可以是SQLITE\_OPEN\_READONLY，SQLITE\_OPEN\_READWRITE和SQLITE\_OPEN\_READWRITE|SQLITE\_OPEN\_CREATE中的一个，用于控制数据库的打开方式，可以和[SQLITE\_OPEN\_NOMUTEX](http://www.sqlite.org/c3ref/c_open_autoproxy.html" \t "https://blog.csdn.net/u014028070/article/details/_blank)，[SQLITE\_OPEN\_FULLMUTEX](http://www.sqlite.org/c3ref/c_open_autoproxy.html" \t "https://blog.csdn.net/u014028070/article/details/_blank)， [SQLITE\_OPEN\_SHAREDCACHE](http://www.sqlite.org/c3ref/c_open_autoproxy.html" \t "https://blog.csdn.net/u014028070/article/details/_blank)，以及[SQLITE\_OPEN\_PRIVATECACHE](http://www.sqlite.org/c3ref/c_open_autoproxy.html" \t "https://blog.csdn.net/u014028070/article/details/_blank)结合使用，具体的详细情况可以查阅文档

=====================================================================================

2．  Sqlite3\_prepare()

这个函数将sql文本转换成一个准备语句（prepared statement）对象，同时返回这个对象的指针。这个接口需要一个数据库连接指针以及一个要准备的包含SQL语句的文本。它实际上并不执行（evaluate）这个SQL语句，它仅仅为执行准备这个sql语句

函数定义（仅列出UTF-8的）

int sqlite3\_prepare(

  sqlite3 \*db,            /\* Database handle \*/  /\* 成功打开的数据库句柄 \*/

  const char \*zSql,       /\* SQL statement, UTF-8 encoded \*/ /\* UTF8编码的 SQL 语句 \*/

  int nByte,              /\* Maximum length of zSql in bytes. \*/   /\* 参数 sql 的字节数, 包含 '\0' \*/

  sqlite3\_stmt \*\*ppStmt,  /\* OUT: Statement handle \*/ /\* 输出:预编译语句句柄 \*/

  const char \*\*pzTail     /\* OUT: Pointer to unused portion of zSql \*/ /\* 输出:指向 sql 语句中未使用的部分 \*/

);

int sqlite3\_prepare\_v2(

  sqlite3 \*db,            /\* Database handle \*/

  const char \*zSql,       /\* SQL statement, UTF-8 encoded \*/

  int nByte,              /\* Maximum length of zSql in bytes. \*/

  sqlite3\_stmt \*\*ppStmt,  /\* OUT: Statement handle \*/

  const char \*\*pzTail     /\* OUT: Pointer to unused portion of zSql \*/

);

参数：

db：数据指针

zSql：sql语句，使用UTF-8编码

nByte：如果nByte小于0，则函数取出zSql中从开始到第一个0终止符的内容；如果nByte不是负的，那么它就是这个函数能从zSql中读取的字节数的最大值。如果nBytes非负，zSql在第一次遇见’/000/或’u000’的时候终止.如果用户知道被传入的 sql 语句是以 '\0' 结尾的, 那么有一个更好的做法是:把nbytes的值设为该字符串的长度(包含'\0'), 这样可以避免 SQLite 复制该字符串的一份拷贝, 以提高程序的效率.

pzTail：上面提到zSql在遇见终止符或者是达到设定的nByte之后结束，假如zSql还有剩余的内容，那么这些剩余的内容被存放到pZTail中，不包括终止符。如果 pszTail 不为 NULL, 则 \*pszTail 指向 sql 中第一个被传入的 SQL 语句的结尾. 该函数只编译 sql 的第一个语句, 所以 \*pszTail 指向的内容则是未被编译的.

ppStmt：能够使用sqlite3\_step()执行的编译好的准备语句的指针，如果错误发生，它被置为NULL，如假如输入的文本不包括sql语句。调用过程必须负责在编译好的sql语句完成使用后使用sqlite3\_finalize()删除它。

说明

如果执行成功，则返回SQLITE\_OK，否则返回一个错误码。推荐在现在任何的程序中都使用sqlite3\_prepare\_v2这个函数，sqlite3\_prepare只是用于前向兼容

备注

<1>准备语句（prepared statement）对象

typedef struct sqlite3\_stmt sqlite3\_stmt;

准备语句（prepared statement）对象一个代表一个简单SQL语句对象的实例，这个对象通常被称为“准备语句”或者“编译好的SQL语句”或者就直接称为“语句”。

         语句对象的生命周期经历这样的过程：

l  使用sqlite3\_prepare\_v2或相关的函数创建这个对象

l  使用sqlite3\_bind\_\*()给宿主参数（host parameters）绑定值

l  通过调用sqlite3\_step一次或多次来执行这个sql

l  使用sqlite3——reset()重置这个语句，然后回到第2步，这个过程做0次或多次

l  使用sqlite3\_finalize()销毁这个对象

在sqlite中并没有定义sqlite3\_stmt这个结构的具体内容，它只是一个抽象类型，在使用过程中一般以它的指针进行操作，而sqlite3\_stmt类型的指针在实际上是一个指向Vdbe的结构体得指针

<2>宿主参数(host parameters)

在传给sqlite3\_prepare\_v2()的sql的语句文本或者它的变量中，满足如下模板的文字将被替换成一个参数：

l  ?

l  ?NNN，NNN代表数字

l  :VVV，VVV代表字符

l  [@VVV](http://my.oschina.net/alivvv" \t "https://blog.csdn.net/u014028070/article/details/_blank)

l  $VVV

在上面这些模板中，NNN代表一个数字，VVV代表一个字母数字标记符（例如:222表示名称为222的标记符），sql语句中的参数（变量）通过上面的几个模板来指定，如

“select ? from ? “这个语句中指定了两个参数，sqlite语句中的第一个参数的索引值是1，这就知道这个语句中的两个参数的索引分别为1和2，使用”?”的话会被自动给予索引值，而使用”?NNN”则可以自己指定参数的索引值，它表示这个参数的索引值为NNN。”:VVV”表示一个名为”VVV”的参数，它也有一个索引值，被自动指定。

例子：

INSERT INTO people (id, name) VALUES ( ?, ? );

INSERT INTO people (id, id2,name) VALUES ( ?1, ?1.?2 );作用：可以用同一个值绑定几个变量

INSERT INTO people (id, name) VALUES ( :id, :name );

INSERT INTO people (id, name) VALUES ( @id, @name );

INSERT INTO people (id, name) VALUES ( $id, $name );用来支持Tcl变量的扩展语法，除非使用Tcl编程，否则推荐使用“:<name>”版本。

可以使用sqlite3\_bind\_\*()来给这些参数绑定值

=====================================================================================

3.

int sqlite3\_bind\_text(sqlite3\_stmt\*, int, const char\*, int n,void(\*)(void\*));

sqlite3\_bind\_\*系列函数有好多，这里只对sqlite3\_bind\_text进行一下说明，sqlite3\_bind\_text的第二个参数为序号（从1开始），第三个参数为字符串值，第四个参数为字符串长度。sqlite3\_bind\_text的第五个参数为一个函数指针，SQLITE3执行完操作后回调此函数，通常用于释放字符串占用的内存。此参数有两个常数，SQLITE\_STATIC告诉sqlite3\_bind\_text函数字符串为常量，可以放心使用；而SQLITE\_TRANSIENT会使得sqlite3\_bind\_text函数对字符串做一份拷贝。一般使用这两个常量参数来调sqlite3\_bind\_text。statement准备好了以后，就是操作的执行了。

int sqlite3\_bind\_blob(sqlite3\_stmt\*, int, const void\*, int n, void(\*)(void\*));  
功能:为查询结构体绑定数据  
PARAM1(IN):查询结构体  
PARAM2(IN):绑定第几个数据(sql语句中的第几个问号(?))  
PARAM3(IN):二进制数据指针  
PARAM4(IN):二进制数据长度  
PARAM5(IN):析构回调函数,通常设置NULL,结束后自己释放  
返回值:SQLITE\_OK成功.  
说明:绑定函数必须要在sqlite3\_step前,sqlite3\_prepare或者sqlite3\_reset之后调用.  
未绑定的参数缺省为NULL,类似的绑定函数还有以下  
int sqlite3\_bind\_double(sqlite3\_stmt\*, int, double);  
int sqlite3\_bind\_int(sqlite3\_stmt\*, int, int);  
int sqlite3\_bind\_int64(sqlite3\_stmt\*, int, sqlite\_int64);  
int sqlite3\_bind\_null(sqlite3\_stmt\*, int);  
int sqlite3\_bind\_text(sqlite3\_stmt\*, int, const char\*, int n, void(\*)(void\*));  
int sqlite3\_bind\_text16(sqlite3\_stmt\*, int, const void\*, int, void(\*)(void\*));  
int sqlite3\_bind\_value(sqlite3\_stmt\*, int, const sqlite3\_value\*);

=====================================================================================

4.  sqlite3\_step()

这个过程用于执行有前面sqlite3\_prepare创建的准备语句。这个语句执行到结果的第一行可用的位置。继续前进到结果的第二行的话，只需再次调用sqlite3\_setp()。继续调用sqlite3\_setp()知道这个语句完成，那些不返回结果的语句（如：INSERT，UPDATE，或DELETE），sqlite3\_step()只执行一次就返回

函数定义

int sqlite3\_step(sqlite3\_stmt\*);

返回值

函数的返回值基于创建sqlite3\_stmt参数所使用的函数，假如是使用老版本的接口sqlite3\_prepare()和sqlite3\_prepare16()，返回值会是 SQLITE\_BUSY， SQLITE\_DONE， SQLITE\_ROW， SQLITE\_ERROR 或 SQLITE\_MISUSE，而v2版本的接口sqlite3\_prepare\_v2()和sqlite3\_prepare16\_v2()则会同时返回这些结果码和扩展结果码。

对所有V3.6.23.1以及其前面的所有版本，需要在sqlite3\_step()之后调用sqlite3\_reset()，在后续的sqlite3\_ step之前。如果调用sqlite3\_reset重置准备语句失败，将会导致sqlite3\_ step返回SQLITE\_MISUSE，但是在V3. 6.23.1以后，sqlite3\_step()将会自动调用sqlite3\_reset。

说明:  
    当一条语句被 sqlite3\_prepare() 或其相关的函数预编译后, sqlite3\_step() 必须被调用一次或多次来评估该预编译语句.  
    该函数的详细行为依赖于由 sqlite3\_prepare()(或其相关的函数) 产生的是一条怎样的预编译语句.  
    函数将返回一个以下的结果来标识其执行结果:  
        SQLITE\_BUSY:忙碌. 数据库引擎无法锁定数据去完成其工作. 但可以多次尝试.  
        SQLITE\_DONE:完成. sql 语句已经被成功地执行. 在调用 sqlite\_reset() 之前, 当前预编译的语句不应该被 sqlite3\_step() 再次调用.  
        SQLITE\_ROW:查询时产生了结果. 此时可以通过相关的"数据访问函数(column access functions)"来取得数据. sqlite3\_step() 的再一次调用将取得下一条查询结果.  
        SQLITE\_ERROR:发生了错误. 此时可以通过 sqlite3\_errmmsg() 取得相关的错误信息. sqlite3\_step() 不能被再次调用.  
        SQLITE\_MISUSE:不正确的库的使用. 该函数使用不当.  
其它:  
    有关预编译的使用, 将在以后的函数介绍中详细说明.

=====================================================================================

5.int sqlite3\_reset(sqlite3\_stmt \*pStmt);

sqlite3\_reset用于重置一个准备语句对象到它的初始状态，然后准备被重新执行。所有sql语句变量使用sqlite3\_bind\*绑定值，使用sqlite3\_clear\_bindings重设这些绑定。Sqlite3\_reset接口重置准备语句到它代码开始的时候。sqlite3\_reset并不改变在准备语句上的任何绑定值，那么这里猜测，可能是语句在被执行的过程中发生了其他的改变，然后这个语句将它重置到绑定值的时候的那个状态。

它的返回值相对有些特殊。返回SQLITE\_BUSY表示暂时无法执行操作，SQLITE\_DONE表示操作执行完毕，SQLITE\_ROW表示执行完毕并且有返回（执行select语句时）。当返回值为SQLITE\_ROW时，我们需要对查询结果进行处理，SQLITE3提供sqlite3\_column\_\*系列函数。

const unsigned char \*sqlite3\_column\_text(sqlite3\_stmt\*, int iCol);

其中参数iCol为列的序号，从0开始。如果返回值有多行，则可以再次调用sqlite3\_step函数，然后由sqlite3\_column\_\*函数取得返回值。使用上述这些函数基本上可以完成对SQLITE3数据库的操作了（不过我这里只针对text数据类型做了说明）。

=====================================================================================

6.  sqlite3\_column()

这个过程从执行sqlite3\_step()执行一个准备语句得到的结果集的当前行中返回一个列。每次sqlite3\_step得到一个结果集的列停下后，这个过程就可以被多次调用去查询这个行的各列的值。对列操作是有多个函数，均以sqlite3\_column为前缀

const void \*sqlite3\_column\_blob(sqlite3\_stmt\*, int iCol);

int sqlite3\_column\_bytes(sqlite3\_stmt\*, int iCol);

int sqlite3\_column\_bytes16(sqlite3\_stmt\*, int iCol);

double sqlite3\_column\_double(sqlite3\_stmt\*, int iCol);

int sqlite3\_column\_int(sqlite3\_stmt\*, int iCol);

sqlite3\_int64 sqlite3\_column\_int64(sqlite3\_stmt\*, int iCol);

const unsigned char \*sqlite3\_column\_text(sqlite3\_stmt\*, int iCol);

const void \*sqlite3\_column\_text16(sqlite3\_stmt\*, int iCol);

int sqlite3\_column\_type(sqlite3\_stmt\*, int iCol);

int sqlite3\_column\_numeric\_type(sqlite3\_stmt\*, int iCol);

sqlite3\_value \*sqlite3\_column\_value(sqlite3\_stmt\*, int iCol);

说明

第一个参数为从sqlite3\_prepare返回来的prepared statement对象的指针，第二参数指定这一行中的想要被返回的列的索引。最左边的一列的索引号是0，行的列数可以使用sqlite3\_colum\_count()获得。

这些过程会根据情况去转换数值的类型，sqlite内部使用sqlite3\_snprintf()去自动进行这个转换，下面是关于转换的细节表：

|  |  |  |
| --- | --- | --- |
| 内部类型 | 请求的类型 | 转换 |
| NULL | INTEGER | 结果是0 |
| NULL | FLOAT | 结果是0.0 |
| NULL | TEXT | 结果是NULL |
| NULL | BLOB | 结果是NULL |
| INTEGER | FLOAT | 从整形转换到浮点型 |
| INTEGER | TEXT | 整形的ASCII码显示 |
| INTEGER | BLOB | 同上 |
| FLOAT | INTEGER | 浮点型转换到整形 |
| FLOAT | TEXT | 浮点型的ASCII显示 |
| FLOAT | BLOB | 同上 |
| TEXT | INTEGER | 使用atoi() |
| TEXT | FLOAT | 使用atof() |
| TEXT | BLOB | 没有转换 |
| BLOB | INTEGER | 先到TEXT，然后使用atoi |
| BLOB | FLOAT | 先到TEXT，然后使用atof |
| BLOB | TEXT | 如果需要的话添加0终止符 |

注：BLOB数据类型是指二进制的数据块，比如要在数据库中存放一张图片，这张图片就会以二进制形式存放，在sqlite中对应的数据类型就是BLOB

int sqlite3\_column\_bytes(sqlite3\_stmt\*, int iCol)int sqlite3\_column\_bytes16(sqlite3\_stmt\*, int iCol)两个函数返回对应列的内容的字节数，这个字节数不包括后面类型转换过程中加上的0终止符。

下面是几个最安全和最简单的使用策略

* 先sqlite3\_column\_text() ，然后 sqlite3\_column\_bytes()
* 先sqlite3\_column\_blob()，然后sqlite3\_column\_bytes()
* 先sqlite3\_column\_text16()，然后sqlite3\_column\_bytes16()

(1)

得到数据行中某个列的数据

sqlite3\_column\_xxx(sqlite3\_stmt\*, int iCol);

在sqlite3\_step返回SQLITE\_ROW后，使用它得到第iCol列的数据。

其中的xxx代表：

blob：指向保存数据内存的指针

bytes, bytes16: 得到该blob类型数据的大小，或者text转换为UTF8/UTF16的字符串长度。

double, int, int64： 数值

text,text16：字符串指针

type:该列的数据类型（SQLITE\_INTEGER，SQLITE\_FLOAT，SQLITE\_TEXT，SQLITE\_BLOB，SQLITE\_NULL）

注意：如果对该列使用了不同与该列本身类型适合的数据读取方法，得到的数值将是转换过的结果。

(2)

得到数据行中某个列的数据的类型

int sqlite3\_column\_type(sqlite3\_stmt\*, int iCol);

返回值：SQLITE\_INTEGER，SQLITE\_FLOAT，SQLITE\_TEXT，SQLITE\_BLOB，SQLITE\_NULL

使用的方法和sqlite3\_column\_xxx()函数类似。

(3)

int sqlite3\_column\_count(sqlite3\_stmt \*pStmt);  
功能:获取列数

如果过程没有返回值，如update，将返回0

const char \*sqlite3\_column\_name(sqlite3\_stmt\*,int);  
功能:获取列名

const char \*sqlite3\_column\_decltype(sqlite3\_stmt \*, int i);  
功能:返回列数据类型

类似：

得到当前行中包含的数据个数

int sqlite3\_data\_count(sqlite3\_stmt \*pStmt);

如果sqlite3\_step返回SQLITE\_ROW，可以得到列数，否则为零。

=====================================================================================

7.int sqlite3\_reset(sqlite3\_stmt \*pStmt);  
功能:重置所有绑定的值,回到刚刚调用sqlite3\_prepare后的状态

=====================================================================================

8.sqlite3\_finalize

    int sqlite3\_finalize(sqlite3\_stmt \*pStmt);

这个过程销毁前面被sqlite3\_prepare创建的准备语句，每个准备语句都必须使用这个函数去销毁以防止内存泄露。

在空指针上调用这个函数没有什么影响，同时可以准备语句的生命周期的任一时刻调用这个函数：在语句被执行前，一次或多次调用sqlite\_reset之后，或者在sqlite3\_step任何调用之后不管语句是否完成执行

示例:  
    sqlite3\_finalize(pStmt);  
    pStmt = NULL;

=====================================================================================

9.sqlite3\_close

这个过程关闭前面使用sqlite3\_open打开的数据库连接，任何与这个连接相关的准备语句必须在调用这个关闭函数之前被释放

int sqlite3\_close(  
    sqlite3\* pDB    /\* 由 sqlite3\_open 或基相关的函数打开的 SQLite 对象句柄 \*/  
    );  
说明:  
    该函数用来析构 sqlite3 对象. 返回 SQLITE\_OK 表示对象被成功析构, 以及所有相关的资源被成功回收  
应用程序必须在关闭之前 "完成(finalize)" 所有的 "预编译语句(prepared statements)", 并且关闭所有的 "二进制句柄绑定(BLOB handle)", 如果在关闭时还有未完成的预编译语句或二进制句柄, 那么函数返回 SQLITE\_BUSY(5).  
示例:  
    if(pDB != NULL)  
    {  
        sqlite3\_close(pDB);  
        pDB = NULL;

    }

=====================================================================================

10.int sqlite3\_exec(  
    sqlite3\* pDB,        /\* sqlite3句柄 \*/  
    const char\* sql,    /\* 被执行的 SQL 语句 \*/  
    int (\*callback)(void\*,int,char\*\*,char\*\*),  /\* 执行/查询回调函数 \*/  
    void\* pvoid,    /\* 传递给回调函数的第一个参数 \*/  
    char\*\*errmsg    /\* 错误输出信息 \*/  
);  
  
说明:  
    该函数用来执行若干条 SQL 语句.  
    该函数包裹了先前版本的 sqlite3\_prepare(), sqlte3\_step() 和 sqlite3\_finalize() 函数, 这样, 用户就可以执行简单的代码执行多条 SQL 语句.  
    sqlite3\_exec()接口执行多条以";"分隔的SQL语句. 如果回调函数不为 NULL, 则它对每一个行查询结果都会调用该回调函数. 如果没有回调函数被指定, sqlite3\_exec() 只是简单地忽略查询结果.  
    当在执行该SQL语句发生错误时, 执行将发生中断, 并且后面的语句也全部被忽略. 如果 errmsg 参数不为空任何错误信息将会被写进由 sqlite3\_malloc() 得到的的内存空间中, 即 errmsg 指向的内存. 为了避免内存泄漏, 应用程序应该在不需要该错误信息后立即调用 sqlite3\_free() 释放该内存空间. 如果 errmsg 参数不为 NULL, 并且没有错误发生, errmsg 被设置为 NULL.  
    如果回调函数返回非零, sqlite3\_exec() 立即中断查询, 并且不再执行后续的 SQL 语句, 也不再调用回调函数, sqlite3\_exec() 将返回 SQLITE\_ABORT 结束执行.  
示例:  
    sqlite3\_exec(pDB, to\_utf8("delete from tablename where id=123;"), NULL, NULL, NULL);  
    sqlite3\_exec(pDB, to\_utf8("create table if not exists tablename (id integer primary key,name text);"), NULL, NULL, NULL);  
    sqlite3\_exec(pDB, to\_utf8("insert into tablename (name) values ('女孩不哭');"), NULL, NULL, NULL);  
    if(sqlite3\_exec(pDB, to\_utf8("select \* from tablename;"), sqlite\_callback, NULL, &pszErrMsg) != SQLITE\_OK)  
    {  
        ...  
        sqlite3\_free(pszErrMsg);  
        pszErrMsg = NULL;  
    }  
参阅:在 sqlite3 中使用回调函数(http://www.cnblogs.com/nbsofer/archive/2012/05/29/2523807.html)

=====================================================================================

11.int sqlite3\_errcode(  
    sqlite3\* pDB    /\* SQLite3 数据库句柄 \*/  
);  
  
说明:  
    该函数返回最近一次调用 sqlite3\_ API时产生的错误码.  
示例:  
    int errcode = sqlite3\_errcode(pDB);  
=====================================================================================  
12.const char \*sqlite3\_errmsg(  
    sqlite3\* pDB    /\* SQLite3 数据库句柄 \*/  
);  
说明:  
    该函数返回与pDB数据库指针相关的错误信息, 由英语书写.  
    用户不必考虑内存的释放, 其由SQLite内部管理, 它也将会在下产次调用函数时被覆盖.  
示例:  
    printf("%s\n", sqlite3\_errmsg(pDB));

13.

=====================================================================================  
int sqlite3\_key(  
  sqlite3 \*db,                   /\* Database to be rekeyed \*/  
  const void \*pKey, int nKey     /\* The key \*/  
);  
功能:为加密的数据库指定密码,改函数在sqlite3\_open之后调用

14.

=====================================================================================  
int sqlite3\_rekey(  
  sqlite3 \*db,                   /\* Database to be rekeyed \*/  
  const void \*pKey, int nKey     /\* The new key \*/  
);  
功能:重设数据库密码,如果pKey = 0 或者 nKey = 0,这数据库不加密

=====================================================================================

15.

=====================================================================================  
int sqlite3\_get\_table(  
  sqlite3\*,               /\* An open database \*/  
  const char \*sql,       /\* SQL to be executed \*/  
  char \*\*\*resultp,       /\* Result written to a char \*[]  that this points to \*/  
  int \*nrow,             /\* Number of result rows written here \*/  
  int \*ncolumn,          /\* Number of result columns written here \*/  
  char \*\*errmsg          /\* Error msg written here \*/  
);  
功能:查询表  
PARAM1(IN):数据库句柄  
PARAM2(IN):sql语句,以\0结尾.  
PARAM3(OUT):查询结果  
PARAM4(OUT):返回行数(多少条数据)  
PARAM5(OUT):返回列数(多少字段)  
PARAM6(OUT):返回错误信息  
返回值:SQLITE\_OK成功,房子失败见错误信息

16.

=====================================================================================  
void sqlite3\_free\_table(char \*\*result);  
功能:释放通过sqlite3\_get\_table查询保存的结果数据  
PARAM1(IN):要释放的数据指针

=====================================================================================

17.int sqlite3\_create\_function(  
  sqlite3 \*,  
  const char \*zFunctionName,  
  int nArg,  
  int eTextRep,  
  void\*,  
  void (\*xFunc)(sqlite3\_context\*,int,sqlite3\_value\*\*),  
  void (\*xStep)(sqlite3\_context\*,int,sqlite3\_value\*\*),  
  void (\*xFinal)(sqlite3\_context\*)  
);  
功能:

const void \*sqlite3\_value\_blob(sqlite3\_value\*);  
int sqlite3\_value\_bytes(sqlite3\_value\*);  
int sqlite3\_value\_bytes16(sqlite3\_value\*);  
double sqlite3\_value\_double(sqlite3\_value\*);  
int sqlite3\_value\_int(sqlite3\_value\*);  
sqlite\_int64 sqlite3\_value\_int64(sqlite3\_value\*);  
const unsigned char \*sqlite3\_value\_text(sqlite3\_value\*);  
const void \*sqlite3\_value\_text16(sqlite3\_value\*);  
const void \*sqlite3\_value\_text16le(sqlite3\_value\*);  
const void \*sqlite3\_value\_text16be(sqlite3\_value\*);  
int sqlite3\_value\_type(sqlite3\_value\*);  
int sqlite3\_value\_numeric\_type(sqlite3\_value\*);  
功能:类似sqlite3\_column\_\*函数

其他：

const char \*sqlite3\_libversion(void);  
int sqlite3\_libversion\_number(void);  
功能:获取版本号

sqlite\_int64 sqlite3\_last\_insert\_rowid(sqlite3\*);  
功能:获取最后插入的行标示.

int sqlite3\_changes(sqlite3\*);  
功能:获取最近执行的sqlite3\_exec影响的行数.

int sqlite3\_total\_changes(sqlite3\*);  
功能:获取自从数据库打开后有改动的函数

void sqlite3\_interrupt(sqlite3\*);  
功能:打断或停止数据库当前操作.

int sqlite3\_complete(const char \*sql);  
功能:判断语句是否以分号(;)结尾

int sqlite3\_busy\_handler(sqlite3\*, int(\*)(void\*,int), void\*);  
功能:设置查询忙碌时的回调处理  
说明:缺省的回调函数为空,如果回调函数为空,表锁定后sqlite3\_exec()执行会直接返回SQLITE\_BUSY

int sqlite3\_busy\_timeout(sqlite3\*, int ms);  
功能:设置查询超时时间(毫秒)

char \*sqlite3\_mprintf(const char\*,...);  
char \*sqlite3\_vmprintf(const char\*, va\_list);  
char \*sqlite3\_snprintf(int,char\*,const char\*, ...);  
功能:格式化字符,需要用%q来代替%s.(主要是不用对分号'进行转义)

void \*sqlite3\_malloc(int);  
void \*sqlite3\_realloc(void\*, int);  
void sqlite3\_free(void\*);  
功能:内存函数

int sqlite3\_set\_authorizer(  
  sqlite3\*,  
  int (\*xAuth)(void\*,int,const char\*,const char\*,const char\*,const char\*),  
  void \*pUserData  
);  
功能:设置数据库授权

void \*sqlite3\_trace(sqlite3\*, void(\*xTrace)(void\*,const char\*), void\*);  
void \*sqlite3\_profile(sqlite3\*,  
   void(\*xProfile)(void\*,const char\*,sqlite\_uint64), void\*);

void sqlite3\_progress\_handler(sqlite3\*, int, int(\*)(void\*), void\*);  
功能:设置执行回调函数,sqlite3\_exec(),sqlite3\_step() ,sqlite3\_get\_table()时会调用

void \*sqlite3\_commit\_hook(sqlite3\*, int(\*)(void\*), void\*);  
功能:设置事务回调函数

int sqlite3\_errcode(sqlite3 \*db);  
const char \*sqlite3\_errmsg(sqlite3\*);  
功能:获取错误码和错误消息

int sqlite3\_bind\_parameter\_count(sqlite3\_stmt\*);  
功能:返回需要绑定的参数数目(sql语句中问号?数量)

const char \*sqlite3\_bind\_parameter\_name(sqlite3\_stmt\*, int);  
功能:获取绑定参数名字,?参数返回NULL

int sqlite3\_bind\_parameter\_index(sqlite3\_stmt\*, const char \*zName);  
功能:获取绑定参数索引

int sqlite3\_clear\_bindings(sqlite3\_stmt\*);  
功能:清除绑定的参数

参考代码：

#include <stdio.h>  
#include "sqlite3.h"

int main(){  
    sqlite3 \*db;  
    char \*zErrMsg = NULL;  
    int rv;  
    char szSql[128] = { 0 };  
    sqlite3\_stmt \*stmt;  
    rv = sqlite3\_open("test.db", &db);  
    if(rv){  
        fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));  
        sqlite3\_close(db);  
        return 1;  
    }  
    else  
       fprintf(stderr,"sqlite3\_open OK\n");  
       //这里查询时，假定数据库中存在表test，并且有一列为COL1(text类型)  
       strcpy(szSql,"SELECT COL1 FROM test where COL1=?");   
       rv = sqlite3\_prepare(db, szSql, 128, &stmt, NULL);  
       if( rv != SQLITE\_OK ) {  
         fprintf(stderr, "sqlite3\_prepare(%d): %s\n", rv, sqlite3\_errmsg(db));  
         return -1;  
       }      
       rv = sqlite3\_bind\_text(stmt, 1, "VALUE1", strlen("VALUE1"), SQLITE\_STATIC);  
       if( rv != SQLITE\_OK ) {  
         fprintf(stderr, "sqlite3\_bind\_text(%d): %s\n", rv, sqlite3\_errmsg(db));  
         return -1;  
       }  
       rv = sqlite3\_step(stmt);  
       if( (rv != SQLITE\_OK) && (rv != SQLITE\_DONE) && (rv != SQLITE\_ROW) ) {  
         fprintf(stderr, "sqlite3\_step(%d): %s\n", rv, sqlite3\_errmsg(db));  
         return -1;  
       }  
       while( rv == SQLITE\_ROW ){  
          fprintf(stderr, "result: %s\n", sqlite3\_column\_text(stmt,0));  
          rv = sqlite3\_step(stmt);  
       }  
       rv = sqlite3\_finalize(stmt);  
       if( rv != SQLITE\_OK ) {  
         fprintf(stderr, "sqlite3\_finalize(%d): %s\n", rv, sqlite3\_errmsg(db));  
         return -1;  
       }  
       sqlite3\_close(db);  
       return 0;  
}

=====================================================================================

#include "stdafx.h"

#include "sqlite3.h"

static int callback( void\*NotUsed,int argc, char \*\*argv, char \*\*azColName)

{

     int i;

     for (i=0; i<argc; i++){

        printf ( "%s = %s/n", azColName[i], argv[i] ? argv[i] :"NULL" );

     }

     printf ( "/n" );

     return 0;

}

#define CHECK\_RC(rc,szInfo,szErrMsg,db) if(rc!=SQLITE\_OK) /

            { printf ( "%s error!/n" ,szInfo);/

            printf ( "%s/n",szErrMsg);    /

            sqlite3\_free(szErrMsg);         /

            sqlite3\_close(db);              /

            return 0;}

int \_tmain( int argc, \_TCHAR\* argv[])

{

     sqlite3 \*db;

     char \*dbPath= "f:/test.db";

     char \*szErrMsg = 0;

     int rc= sqlite3\_open(dbPath, &db);

     CHECK\_RC(rc, "open database" ,db);

     char \*szSql= "create table UserInfo(ID int primary key , UserName char, PassWord char);";

     rc=sqlite3\_exec(db,szSql,0,0,&szErrMsg);

     CHECK\_RC(rc, "create table" ,szErrMsg,db);

     rc=sqlite3\_exec(db, "insert into UserInfo(ID,UserName,PassWord) values(1,'kfqcome','123456')",0,0,&szErrMsg);

     CHECK\_RC(rc, "insert info" ,szErrMsg,db);

     rc=sqlite3\_exec(db, "insert into UserInfo(ID,UserName,PassWord) values(2,'miss wang','654321')",0,0,&szErrMsg);

     CHECK\_RC(rc, "insert info" ,szErrMsg,db);

     szSql= "select \* from UserInfo" ;

     rc = sqlite3\_exec(db,szSql, callback, 0, &szErrMsg);

     CHECK\_RC(rc, "query values" ,szErrMsg,db);

     sqlite3\_close(db);

     getchar ();

     return 0;

}

　　输出的结果：

ID = 1

UserName = kfqcome

PassWord = 123456

ID = 2

UserName = miss wang

PassWord = 654321