第5章

SQLite基础

通常情况下，数据库作为软件应用程序的主要组成部分，与数据库管理系统一样，非常的庞大，并且占用了相当多的系统资源，增加了管理的复杂性。随着嵌入式系统的飞速发展以及软件应用程序逐渐模块化，采用嵌入式特点的新型数据库管理系统要比大型复杂的传统数据库管理系统更为适用。SQLite就是这样一个十分优秀的嵌入式数据库系统。在PHP5中已经集成了SQLite的嵌入式数据库产品。SQLite也被用于很多航空电子设备、建模和仿真程序、工业控制、智能卡、决策支持包、医药信息系统等。

SQLite简单易用，同时提供了丰富的数据库接口。它的设计思想是小型、快速和最小化的管理。这对于需要一个数据库用于存储数据但又不想花太多时间来调整数据性能的开发人员很适用。实际上在很多情况下嵌入式系统的数据管理并不需要存储程序或复杂的表之间的关联。SQLite在数据库容量大小和管理功能之间找到了理想的平衡点。而且SQLite的版权允许无任何限制的应用，包括商业性的产品。完全的开源代码这一特点更使其可以称得上是理想的“嵌入式数据库”。部分开源嵌入式数据库性能对比见表5-1。

表5-1　嵌入式数据库性能对比表

产品名称	速度	稳定性	数据库容量	SQL支持	Win32平台下最小体积	数据操纵
SQLite	最快	好	2TB	大部分SQL-92	374KB	SQL
Berkeley DB	快	好	256TB	不支持	840KB	仅应用程序接口
Firebird嵌入式服务器	快	好	64TB	完全SQL-92与大部分SQL-99	3.68MB	SQL

本章主要介绍SQLite数据库的基础知识，包括数据类型、常用命令、API函数、工具和实例。目前SQLite已经于2020年1月27日进入SQLite3.31.1版本。

5.1　SQLite的特点及适用场景


视频讲解

SQLite是一个开源的、内嵌式的关系型数据库。它是D. Richard Hipp采用C语言开发出来的、完全独立的、不具有外部依赖性的嵌入式数据库引擎。SQLite第一个版本发布于2000年5月，在便携性、易用性、紧凑性、有效性和可靠性方面有突出的表现。

SQLite 能够运行在 Windows、Linux、UNIX 等各种操作系统,同时支持多种编程语言如 Java、PHP、TCL、Python 等。SQLite 主要特点如下。

(1) 支持 ACID(Atomic,Consistent,Isolated,and Durable)事务。

(2) 零配置,即无须安装和管理配置。

(3) 储存在单一磁盘文件中的一个完整的数据库。

(4) 数据文件可在不同字节顺序的机器间自由共享。

(5) 支持数据库大小至 2TB。

(6) 程序体积小,全部 C 语言代码约 3 万行(核心软件,包括库和工具),250KB 大小。

(7) 相对于目前其他嵌入式数据库具有更快捷的数据操作。

(8) 支持事务功能和并发处理。

(9) 程序完全独立,不具有外部依赖性。

(10) 支持多种硬件平台,如 ARM/Linux、SPARC/Solaris 等。

SQLite 不同于其他大部分的 SQL 数据库引擎,因为它的首要设计目标就是尽量简单化以达到易于管理、易于使用、易于嵌入到其他的大型程序中、易于维护和配置的目的。SQLite 比较适用的场合主要包括网站,嵌入式设备和应用软件,应用程序文件格式,替代某些特别的文件格式,内部的或临时的数据库,命令行数据集分析工具,企业级数据库的替代品,数据库教学等。

5.2 SQLite 的存储种类和数据类型

与其他传统关系型数据库使用的是静态数据类型不同的是,SQLite3 采用的是动态数据类型,即字段可以存储的数据类型是在表声明时就确定的,因此它们在数据存储方面存在着很大的差异。

SQLite 将数据值的存储划分为以下几种存储类型。

(1) NULL,空值。

(2) INTEGER,整型,根据大小可以使用 1、2、3、4、6、8 字节来存储。

(3) REAL,浮点型,用来存储 8 字节的 IEEE 浮点。

(4) TEXT,文本字符串,使用 UTF-8、UTF-16、UTF-32 等保存数据。

(5) BLOB(Binary Large Objects),二进制类型,按照二进制存储,不做任何改变。

在 SQLite 中,存储种类和数据类型有一定的差别,如 INTEGER 存储类别可以包含 6 种不同长度的 Integer 数据类型,然而这些 INTEGER 数据一旦被读入到内存后,SQLite 会将其全部视为占用 8 字节的无符号整型。因此对于 SQLite 而言,即使在表声明中明确了字段类型,仍然可以在该字段中存储其他类型的数据。然而需要特别说明的是,尽管 SQLite 提供了这种方便,但是一旦考虑到数据库平台的可移植性问题,用户在实际的开发中还是应该尽可能地保证数据类型的存储和声明的一致性。除非有极为充分的理由,同时又不再考虑数据库平台的移植问题,在此种情况下确实可以使用 SQLite 提供的此种特征。

需要注意的是,实际上,SQLite3 也接受如下的扩展的数据类型,见表 5-2。

表 5-2 SQLite3 接受的数据类型(扩展)

数据类型	说 明
smallint	16 位元的整数
integer	32 位元的整数
decimal(p,s)	精确值 p 是指该数根据权的大小排列的数位集合,s 是指小数点后有几位数。如未特别指定,则默认设为 p=5; s=0
float	32 位元的实数
double	64 位元的实数
char(n)	n 长度的字串,n 不能超过 254
varchar(n)	长度不固定且其最大长度为 n 的字串,n 不能超过 4000
graphic(n)	类似 char(n),但其单位是两个字元 double-bytes,n 不能超过 127
vargraphic(n)	可变长度且其最大长度为 n 的双字元字串,n 不能超过 2000
date	包含年份、月份、日期
time	包含小时、分钟、秒

为了最大化 SQLite 和其他数据库引擎之间的数据类型兼容性,SQLite 提出了“类型亲和性”(Type Affinity)的概念。所谓“类型亲和性”指的是在表字段被声明之后,SQLite 都会根据该字段声明时的类型为其选择一种亲和类型,当数据插入时,该字段的数据将会优先采用亲和类型作为该值的存储方式,除非亲和类型不匹配或无法转换当前数据到该亲和类型,这样 SQLite 才会考虑其他更适合该值的类型存储该值。

在 SQLite3 版数据库中的列类型有五种类型亲和性:文本类型、数字类型、整数类型、浮点类型、NULL 无类型。

(1) 一个具有文本类型亲和性的列,可以使用 NULL、TEXT、BLOB 值类型保存数据。例如,数字数据被插入一个具有文本类型亲和性的列,在存储之前数字将被转换成文本。

(2) 一个具有数字类型亲和性的列,可以使用 NULL、INTEGER、REAL、TEXT、BLOB 五种值类型保存数据。例如,一个文本类型数据被插入到一个具有数字类型亲和性的列,在存储之前将被转变成整型或浮点型。

(3) 一个具有整数亲和性的列,在转换方面和具有数字亲和性的列是相同的,但也有些区别,如浮点型的值将被转换成整型。

(4) 一个具有浮点亲和性的列,可以使用 REAL、FLOAT、DOUBLE 值类型保存数据。

(5) 一个具有无类型亲和性的列,不会选择用哪个类型保存数据,数据不会进行任何转换。

5.3 SQLite 语法

SQLite 库可以解析大部分的标准 SQL 语言,但同时它也省去了一些原有的特性,并且加入了一些自己的新特性。SQLite 的语法主要针对数据表、视图、索引、事务等对象设计规则,包括结构定义、结构删除、数据操作、事务处理、其他操作等几个部分。

5.3.1 数据表操作

1. 创建表

该命令的语法规则和使用方式与大多数关系型数据库基本相同,因此本章还是以示例

的方式来演示 SQLite 中创建表的各种规则。但是对于一些 SQLite 特有的规则,本章会给予额外的说明。

1) 最简单的数据表

```
sqlite> CREATE TABLE testable (num integer);
```

这里需要说明的是,对于自定义数据表表名,如 testtable,不能以 sqlite_开头,因为以该前缀定义的表名都用于 SQLite 内部。

2) 创建带有默认值的数据表

```
sqlite> CREATE TABLE testtable (num integer DEFAULT 0, description varchar
DEFAULT 'hello');
```

3) 在指定数据库创建表

```
sqlite> ATTACH DATABASE '/home/work/proj/mydb.db' AS mydb;
sqlite> CREATE TABLE mydb.testtable (num integer);
```

这里先通过 ATTACH DATABASE 命令将一个已经存在的数据库文件 attach 到当前的连接中(attach 命令将在后文介绍),之后再通过指定数据库名的方式在目标数据库中创建数据表,如 mydb.testtable。关于该规则还需要给出一些额外的说明,如果我们在创建数据表时没有指定数据库名,那么将会在当前连接的 main 数据库(主数据库)中创建该表,在一个连接中只能有一个 main 数据库。如果需要创建临时表,就无须指定数据库名,见如下示例。

创建两个表,一个临时表和普通表。

```
sqlite> CREATE TEMP TABLE temptable(num integer);
sqlite> CREATE TABLE testtable (num integer);
```

将当前连接中的缓存数据导出到本地文件,同时退出当前连接。

```
sqlite> .backup /home/work/proj/mydb.db
sqlite> .exit
```

重新建立 SQLite 的连接,并将刚刚导出的数据库作为主库重新导入。

查看该数据库中的表信息,通过结果可以看出临时表并没有被持久化到数据库文件中。

```
sqlite> .tables
testtable
```

4) IF NOT EXISTS 从句

如果当前创建的数据表名已经存在,即与已经存在的表名、视图名或索引名冲突,那么本次创建操作将失败并报错。然而如果在创建表时加上 IF NOT EXISTS 从句,那么本次创建操作将不会有任何影响,即不会有错误抛出,除非当前的表名或某一索引名冲突。

```
sqlite> CREATE TABLE testtable (num integer);
Error: table testtable already exists
sqlite> CREATE TABLE IF NOT EXISTS testtable (numl integer);
```

5）主键约束

直接在字段的定义上指定主键。

```
sqlite> CREATE TABLE testtable (num integer PRIMARY KEY ASC);
```

在所有字段已经定义完毕后，再定义表的数约束，这里定义的是基于 num 和 description 的联合主键。

```
sqlite> CREATE TABLE testtable2 (
...>    num integer,
...>    description integer,
...>    PRIMARY KEY (num,description)
...> );
```

和其他关系型数据库一样，主键必须是唯一的。

6）唯一性约束

直接在字段的定义上指定唯一性约束。

```
sqlite> CREATE TABLE testtable (num integer UNIQUE);
```

在所有字段已经定义完毕后，再定义表的唯一性约束，这里定义的是基于两个列的唯一性约束。

```
sqlite> CREATE TABLE testtable2 (
...>    num integer,
...>    description integer,
...>    UNIQUE (num, description)
...> );
```

在 SQLite 中，NULL 值被视为和其他任何值都是不同的，如下例：

```
sqlite> DELETE FROM testtable;
sqlite> SELECT count( * ) FROM testtable;
count( * )
 ----------
0
sqlite> INSERT INTO testtable VALUES(NULL);
sqlite> INSERT INTO testtable VALUES(NULL);
sqlite> SELECT count( * ) FROM testtable;
count( * )
 ----------
2
```

由此可见，两次插入的 NULL 值均插入成功。

2. 表的修改

SQLite 对 ALTER TABLE 命令支持得非常有限，仅支持修改表名和添加新字段。其他的功能，如重命名字段、删除字段和添加、删除约束等均未提供支持。

1) 修改表名

需要先说明的是,SQLite 中表名的修改只能在同一个数据库中,不能将其移动到 Attached 数据库中。而且一旦表名被修改后,该表已存在的索引将不会受到影响,然而依赖该表的视图和触发器将不得不重新修改其定义。

```
sqlite> CREATE TABLE testtable (num integer);
sqlite> ALTER TABLE testtable RENAME TO testtable2;
sqlite> .tables
testtable2
```

通过.tables 命令的输出可以看出,表 testtable 已经被修改为 testtable2。

2) 新增字段

```
sqlite> CREATE TABLE testtable (num integer);
sqlite> ALTER TABLE testtable ADD COLUMN description integer;
sqlite> .schema testtable
CREATE TABLE "testtable" (num integer, description integer);
```

通过.schema 命令的输出可以看出,表 testtable 的定义中已经包含了新增字段。关于 ALTER TABLE 最后需要说明的是,在 SQLite 中该命令的执行时间不会受到当前表行数的影响。

3. 表的删除

在 SQLite 中如果某个表被删除了,那么与之相关的索引和触发器也会被随之删除。而在很多其他的关系型数据库中是不可以这样的,如果必须要删除相关对象,只能在删除表语句中加入 WITH CASCADE 从句。见如下示例:

```
sqlite> CREATE TABLE testtable (num integer);
sqlite> DROP TABLE testtable;
sqlite> DROP TABLE testtable;
Error: no such table: testtable
sqlite> DROP TABLE IF EXISTS testtable;
```

从上面的示例中可以看出,如果删除的表不存在,SQLite 将会报错并输出错误信息。如果希望在执行时不抛出异常,可以添加 IF EXISTS 从句,该从句的语义和 CREATE TABLE 中的完全相同。

5.3.2 视图的操作

1. 创建视图

这里只是给出简单的 SQL 命令示例,具体的含义和技术细节可以参照上面的创建数据表部分,如临时视图、IF NOT EXISTS 从句等。

1) 最简单的视图

```
sqlite> CREATE VIEW testview AS SELECT * FROM testtable WHERE num > 10;
```

2）创建临时视图

```
sqlite> CREATE TEMP VIEW tempview AS SELECT * FROM testtable WHERE num > 10;
```

3）IF NOT EXISTS 从句

```
sqlite> CREATE VIEW testview AS SELECT * FROM testtable WHERE num > 10;
Error: table testview already exists
sqlite> CREATE VIEW IF NOT EXISTS testview AS SELECT * FROM testtable WHERE
numl > 10;
```

2. 删除视图

该操作的语法和删除表基本相同，因此这里只是给出示例：

```
sqlite> DROP VIEW testview;
sqlite> DROP VIEW testview;
Error: no such view: testview
sqlite> DROP VIEW IF EXISTS testview;
```

5.3.3 索引的操作

1. 创建索引

在 SQLite 中，创建索引的 SQL 语法和其他大多数关系型数据库基本相同，因而这里也仅仅是给出示例用法，在命令行中输入以下指令，首先创建表。

```
sqlite> CREATE TABLE testtable (name text,num integer);
```

创建最简单的索引，该索引基于某个表的一个字段。

```
sqlite> CREATE INDEX testtable_idx ON testtable(name);
```

创建联合索引，该索引基于某个表的多个字段，同时可以指定每个字段的排序规则（升序/降序）。

```
sqlite> CREATE INDEX testtable_idx2 ON testtable(name ASC,num DESC);
```

创建唯一性索引，该索引规则和数据表的唯一性约束的规则相同，即 NULL 和任何值都不同，包括 NULL 本身。

```
sqlite> CREATE UNIQUE INDEX testtable_idx3 ON testtable(num DESC);
sqlite> .indices testtable
```

上述命令的输出结果如图 5-1 所示。

在图 5-1 中从.indices 命令的输出可以看出，三个索引均已成功创建。

2. 删除索引

索引的删除和视图的删除非常相似，含义也是如此，因此这里也只是给出示例：

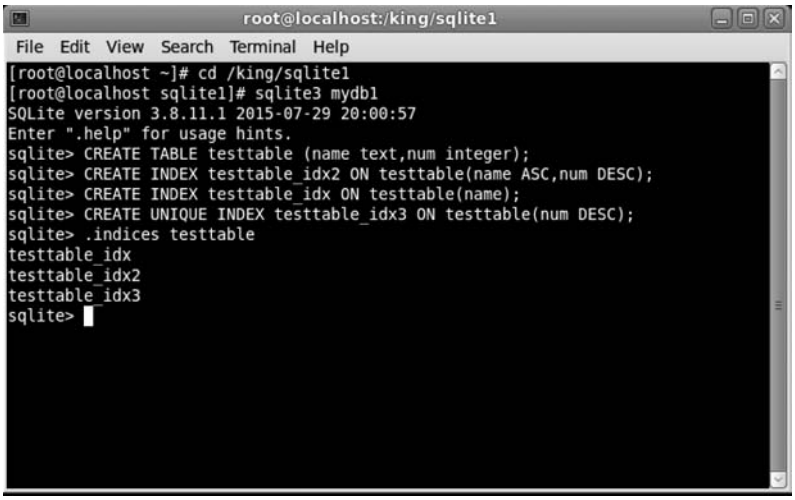

图 5-1 创建索引

```
sqlite> DROP INDEX testtable_idx;
```

如果删除的索引不存在将会导致操作失败,在不确定索引是否存在但又不希望错误被抛出的情况下,可以使用 IF EXISTS 从句。

```
sqlite> DROP INDEX testtable_idx;
Error: no such index: testtable_idx
sqlite> DROP INDEX IF EXISTS testtable_idx;
```

3. 重建索引

重建索引用于删除已经存在的索引,同时基于其原有的规则重建该索引。这里需要说明的是,如果在 REINDEX 语句后面没有给出数据库名,那么当前连接下所有 Attached 数据库中所有索引都会被重建。如果指定了数据库名和表名,那么该表中的所有索引都会被重建,如果只是指定索引名,那么当前数据库的指定索引被重建。

当前连接 Attached 所有数据库中的索引都被重建。

```
sqlite> REINDEX;
```

重建当前主数据库中 testtable 表的所有索引。

```
sqlite> REINDEX testtable;
```

重建当前主数据库中名称为 testtable_idx2 的索引。

```
sqlite> REINDEX testtable_idx2;
```

5.3.4 触发器的操作

触发器是一种特殊的存储过程,在用户试图对指定的表执行指定的数据修改语句时自

动执行。SQLite 的触发器操作见表 5-3。

表 5-3 SQLite 的触发器操作

名 称	描 述	基 本 语 法
Create trigger	创建一个触发器	Create triggertrigger_name Database evevt ON [database_name] table_name trigger_action Database evevt: insert/delete/update/update of trigger_ action: BEGIN/select _ statement/ insert _ statement/ delete_statement/update_statement/END
Drop trigger	删除一个触发器	Drop trigger [database_name] trigger_name

5.3.5 日期和时间函数

SQLite 主要支持以下四种与日期和时间相关的函数。

(1) date(timestring, modifier, modifier, ...)。

(2) time(timestring, modifier, modifier, ...)。

(3) datetime(timestring, modifier, modifier, ...)。

(4) strftime(format, timestring, modifier, modifier, ...)。

以上四个函数都接受一个时间字符串作为参数,其后再跟有 0 个或多个修改符。其中,strftime()函数还接受一个格式字符串作为其第一个参数。strftime()和 C 运行时库中的同名函数完全相同。至于其他三个函数,date 函数的默认格式为:“YYYY-MM-DD”,time 函数的默认格式为:“H:MM:SS”,datetime 函数的默认格式为:“YYYY-MM-DD HH:MM:SS”。

1. strftime 函数的格式信息

strftime 函数的格式见表 5-4。

表 5-4 strftime 函数的格式

strftime 函数的格式	描 述
%d	day of month: 00
%f	fractional seconds: SS.SSS
%H	hour: 00-24
%j	day of year: 001-366
%J	Julian day number
%m	month: 01-12
%M	minute: 00-59
%s	seconds since 1970-01-01
%S	seconds: 00-59
%w	day of week 0-6 with Sunday==0
%W	week of year: 00-53
%Y	year: 0000-9999
%%	%

需要指出的是,其余三个时间函数均可用 strftime 来表示,如:

date(...) trftime('%Y-%m-%d', ...)

time(...) strftime('%H: %M: %S', ...)

datetime(...) strftime('%Y-%m-%d %H: %M: %S', ...)。

2. 时间字符串的格式

时间字符串的格式如下。

(1) YYYY-MM-DD。

(2) YYYY-MM-DD HH:MM。

(3) YYYY-MM-DD HH:MM:SS。

(4) YYYY-MM-DD HH:MM:SS.SSS。

(5) HH:MM。

(6) HH:MM:SS。

(7) HH:MM:SS.SSS。

(8) now。

说明:(5)~(7)中只是包含了时间部分,SQLite 将假设日期为 2000-01-01。(8)表示当前时间。

3. 修改符

修改符有如下格式。

(1) NNN days。

(2) NNN hours。

(3) NNN minutes。

(4) NNN.NNNN seconds。

(5) NNN months。

(6) NNN years。

(7) start of month。

(8) start of year。

(9) start of day。

(10) weekday N。

其中,(1)~(6)将只是简单地加减指定数量的日期或时间值,如果 NNN 的值为负数,则减,否则加。(7)~(9)则将时间串中的指定日期部分设置到当前月、年或日的开始。(10)则将日期前进到下一个星期 N,其中星期日为 0。需要说明的是:修改符的顺序极为重要,SQLite 将会按照从左到右的顺序依次执行修改符。

4. 示例

返回当前日期。

```
sqlite> SELECT date('now');
2020-02-28
```

返回当前月的最后一天。

```
sqlite> SELECT date('now','start of month','1 month','-1 day');
2020-02-29
```

返回从 1970-01-01 00：00：00 到当前时间所流经的秒数。

```
sqlite> SELECT strftime('%s','now');
1582877455
```

返回当前年中 10 月份的第一个星期二的日期。

```
sqlite> SELECT date('now','start of year','+9 months','weekday 2');
2020-10-06
```

5.3.6 数据库和事物

1. Attach 数据库

ATTACH DATABASE 语句添加另外一个数据库文件到当前的连接中，如果文件名为“:memory:”，我们可以将其视为内存数据库，内存数据库无法持久化到磁盘文件上。如果操作 Attached 数据库中的表，则需要在表名前加数据库名，如 dbname. table_name。最后需要说明的是，如果一个事务包含多个 Attached 数据库操作，那么该事务仍然是原子的。

见如下示例：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> INSERT INTO testtable VALUES(1);
sqlite> .backup 'D:/mydb.db'        -- 将当前连接中的主数据库备份到指定文件
sqlite> .exit
```

重新登录 SQLite 命令行工具：

```
sqlite> CREATE TABLE testtable (first_col integer);
sqlite> INSERT INTO testtable VALUES(2);
sqlite> INSERT INTO testtable VALUES(1);
sqlite> ATTACH DATABASE 'D:/mydb.db' AS mydb;
sqlite> .header on              -- 查询结果将字段名作为标题输出
sqlite> SELECT t1.first_col FROM testtable t1, mydb.testtable t2 WHERE t.first_col =
t2.first_col;
first_col
-----------
1
```

2. Detach 数据库

DETACH DATABASE 语句卸载当前连接中的指定数据库，注意 main 和 temp 数据库无法被卸载。

该示例承载上面示例的结果，即 mydb 数据库已经被 Attach 到当前的连接中。

```
sqlite> DETACH DATABASE mydb;
sqlite> SELECT t1.first_col FROM testtable t1, mydb.testtable t2 WHERE t.first_col =
t2.first_col;
Error: no such table: mydb.testtable
```

3. 事务

事务处理是由一条或多条 SQL 语句序列结合在一起所形成的一个逻辑处理单元。事务处理中的每条语句都只完成整个任务中的一部分工作,所有的语句组织在一起才能够完成某个特定的任务。支持事务处理以保证一个事务内的所有操作都能完成,否则就全部取消并回复到原状态。事务处理的语法包括事务开始、事务终止、事务结束以及事务回滚四个主要部分,其具体内容见表 5-5。

表 5-5 SQLite 事务处理语法表

事 务 分 类	作 用	基 本 语 法
Begin transaction	标记一个事务的起点	BEGIN[TRANSACTION [name]]
End transaction	标记一个事务的结束	END[TRANSACTION [name]]
Commit transaction	标记一个事务的结束,功能同 End transaction	COMMIT[TRANSACTION [name]]
Roll back transaction	将事务回滚到起点	ROLL BACK[TRANSACTION [name]]

在 SQLite 中,如果没有为当前的 SQL 命令(SELECT 除外)显式地的指定事务,那么 SQLite 会自动为该操作添加一个隐式的事务,以保证该操作的原子性和一致性。当然,SQLite 也支持显式的事务,其语法与大多数关系型数据库相比基本相同。见如下示例:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO testtable VALUES(1);
sqlite> INSERT INTO testtable VALUES(2);
sqlite> COMMIT TRANSACTION;           -- 显式事务被提交,数据表中的数据也发生了变化
sqlite> SELECT COUNT( * ) FROM testtable;
COUNT( * )
 ----------
2
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO testtable VALUES(1);
sqlite> ROLLBACK TRANSACTION;         -- 显式事务被回滚,数据表中的数据没有发生变化
sqlite> SELECT COUNT( * ) FROM testtable;
COUNT( * )
 ----------
2
```

5.4 SQLite 的内置函数

本节介绍 SQLite 的内置函数。SQLite 本身集成了大量的内置函数以方便使用者快捷地操作、使用数据库。其内置函数(主要函数)主要分为算术函数、字符串处理函数、条件判断函数、聚合函数以及其他函数五大类,其声明和描述见表 5-6。

表 5-6 典型内置函数

函 数	说 明
算术函数	
abs(X)	该函数返回数值参数 X 的绝对值，如果 X 为 NULL，则返回 NULL；如果 X 为不能转换成数值的字符串，则返回 0；如果 X 值超出 Integer 的上限，则抛出 Integer Overflow 的异常
max(X,Y,...)	返回函数参数中的最大值，如果有任何一个参数为 NULL，则返回 NULL
min(X,Y,...)	返回函数参数中的最小值，如果有任何一个参数为 NULL，则返回 NULL
round(X[,Y])	返回数值参数 X 被四舍五入到 Y 刻度的值，如果参数 Y 不存在，默认参数值为 0
random()	返回整型的伪随机数
条件判断函数	
coalesce(X,Y,...)	返回函数参数中第一个非 NULL 的参数，如果参数都是 NULL，则返回 NULL。该函数至少 2 个参数
ifnull(X,Y)	该函数等同于两个参数的 coalesce()函数，即返回第一个不为 NULL 的函数参数。如果两个均为 NULL，则返回 NULL
nullif(X,Y)	如果函数参数相同，返回 NULL，否则返回第一个参数
字符串处理函数	
length(X)	如果参数 X 为字符串，则返回字符的数量；如果为数值，则返回该参数的字符串表示形式的长度；如果为 NULL，则返回 NULL
lower(X)	返回函数参数 X 的小写形式，默认情况下，该函数只能应用于 ASCII 字符
ltrim(X[,Y])	如果没有可选参数 Y，该函数将移除参数 X 左侧的所有空格符；如果有参数 Y，则移除 X 左侧的任意在 Y 中出现的字符，最后返回移除后的字符串
replace(X,Y,Z)	字符串类型的函数参数 X 中所有子字符串 Y 替换为字符串 Z，最后返回替换后的字符串，源字符串 X 保持不变
rtrim(X[,Y])	如果没有可选参数 Y，该函数将移除参数 X 右侧的所有空格符；如果有参数 Y，则移除 X 右侧的任意在 Y 中出现的字符，最后返回移除后的字符串
substr(X,Y[,Z])	返回函数参数 X 的子字符串，从 X 中截取 Z 长度的字符，如果忽略 Z 参数，则取第 Y 个字符后面的所有字符。如果 Z 的值为负数，则从第 Y 位开始，向左截取 abs(Z)个字符。如果 Y 值为负数，则从 X 字符串的尾部开始计数到第 abs(Y)的位置开始
trim(x[,y])	如果没有可选参数 Y，该函数将移除参数 X 两侧的所有空格符；如果有参数 Y，则移除 X 两侧的任意在 Y 中出现的字符，最后返回移除后的字符串
upper(X)	返回函数参数 X 的大写形式，默认情况下，该函数只能应用于 ASCII 字符
聚合函数	
avg(x)	该函数返回在同一组内参数字段的平均值。对于不能转换为数字值的 STRING 和 BLOB 类型的字段值，如'HELLO'，SQLite 会将其视为 0。avg 函数的结果总是浮点型，唯一的例外是所有的字段值均为 NULL，那样该函数的结果也为 NULL
count(x|*)	count(x)函数返回在同一组内，x 字段中值不等于 NULL 的行数。count(*)函数返回在同一组内的数据行数
group_concat(x[,y])	该函数返回一个字符串，该字符串将会连接所有非 NULL 的 x 值。该函数的 y 参数将作为每个 x 值之间的分隔符，如果在调用时忽略该参数，在连接时将使用默认分隔符“,”

续表

函　数	说　明
聚合函数	
max(x)	该函数返回同一组内的 x 字段的最大值，如果该字段的所有值均为 NULL，该函数也返回 NULL
min(x)	该函数返回同一组内的 x 字段的最小值，如果该字段的所有值均为 NULL，该函数也返回 NULL
sum(x)	该函数返回同一组内的 x 字段值的总和，如果字段值均为 NULL，该函数也返回 NULL。如果所有的 x 字段值均为整型或者 NULL，该函数返回整型值，否则就返回浮点型数值。如果所有的数据值均为整型，一旦结果超过上限时将会抛出 integer overflow 的异常
total(x)	该函数不属于标准 SQL，其功能和 sum 基本相同，只是计算结果比 sum 更为合理。例如，当所有字段值均为 NULL 时，和 sum 不同的是，该函数返回 0.0。另外该函数始终返回浮点型数值。该函数始终都不会抛出异常
其他函数	
changes()	该函数返回最近执行的 INSERT、UPDATE 和 DELETE 语句所影响的数据行数，也可以通过执行 C/C++函数 sqlite3_changes()得到相同的结果
total_changes()	该函数返回自从该连接被打开时起，INSERT、UPDATE 和 DELETE 语句总共影响的行数，也可以通过 C/C++接口函数 sqlite3_total_changes()得到相同的结果
typeof(X)	返回函数参数数据类型的字符串表示形式，如 INTEGER、TEXT、REAL、NULL 等

这里还需要进一步说明的是，对于所有聚合函数而言，distinct 关键字可以作为函数参数字段的前置属性，以便在进行计算时忽略所有重复的字段值，如 count(distinct x)。图 5-2 显示了 SQLite 官网上对主要内置函数的列表显示。

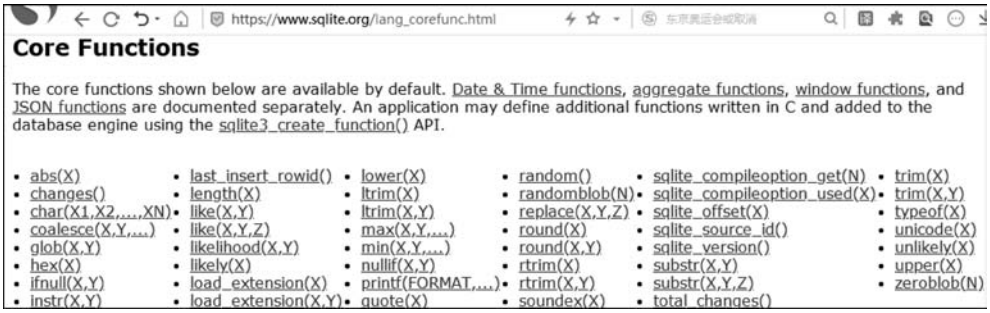

图 5-2　SQLite 的内置函数

5.5　SQLite 的运算符

SQLite 的运算符是一个保留字或字符，主要用于在 SQLite 语句中的 WHERE 子句中执行操作，如比较和算术运算等。SQLite 主要有数学运算符、比较运算符、逻辑运算符和位运算符四种。

5.5.1 数学运算符

所有的数学运算符(+,-,*,/,%,<<,>>,& 和|)在执行之前都会先将操作数转换为NUMERIC 存储类型,即使在转换过程中可能会造成数据信息的丢失。此外,如果其中一个操作数为 NULL,那么它们的结果也为 NULL。在数学操作符中,如果其中一个操作数看上去并不像数值类型,那么它们结果为 0 或 0.0。图 5-3 显示了使用了数学运算符的实例图。

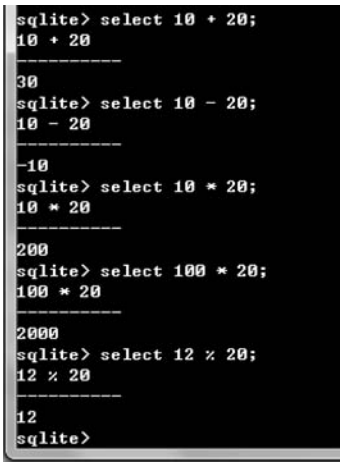

图 5-3 SQLite 的数学运算符应用

5.5.2 比较运算符

在 SQLite 中支持的比较运算符有=,==,<,<=,>,>=,!=,<>,IN,NOT IN,BETWEEN,IS 和 IS NOT。

数据的比较结果主要依赖于操作数的存储方式,其规则如下。

(1) 存储方式为 NULL 的数值小于其他存储类型的值。

(2) 存储方式为 INTEGER 和 REAL 的数值小于 TEXT 或 BLOB 类型的值,如果同为 INTEGER 或 REAL,则基于数值规则进行比较。

(3) 存储方式为 TEXT 的数值小于 BLOB 类型的值,如果同为 TEXT,则基于文本规则(ASCII 值)进行比较。

(4) 如果是两个 BLOB 类型的数值进行比较,其结果为 C 运行时函数 memcmp()的结果。

下面给出一个实例。假设变量 a=10,变量 b=20,则表 5-7 列举了各比较运算符的应用情况。

表 5-7 比较运算符

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)为假
<	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或等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小于或等于右操作数的值,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

5.5.3 逻辑运算符

SQLite 逻辑运算符见表 5-8。

表 5-8 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,如 NOT EXISTS、NOT BETWEEN、NOT IN,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)

5.5.4 位运算符

如果 $A=60$,且 $B=13$,则表 5-9 列举了各位运算符的应用情况。

表 5-9 位运算符

运算符	描 述	实 例
&	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。	(~A)将得到-61,即为 1100 0011,2 的补码形式,带符号的二进制数
<<	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

5.6 SQLite 的常用命令

用户可以在任何时候输入“.help”,列出可用的命令。

```
sqlite> .help
.bail ON|OFF              Stop after hitting an error.    Default OFF
.databases                List names and files of attached databases
.dump ?TABLE? ...         Dump the database in an SQL text format
.echo ON|OFF              Turn command echo on or off
.exit                     Exit this program
.explain ON|OFF           Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF         Turn display of headers on or off
.help                     Show this message
.import FILE TABLE        Import data from FILE into TABLE
.indices TABLE            Show names of all indices on TABLE
.load FILE ?ENTRY?        Load an extension library
.mode MODE ?TABLE?        Set output mode where MODE is one of:
                            csv      Comma - separated values
                            column   Left - aligned columns.    (See .width)
                            html     HTML < table > code
                            insert   SQL insert statements for TABLE
                            line     One value per line
                            list     Values delimited by .separator string
                            tabs     Tab - separated values
                            tcl      TCL list elements
.nullvalue STRING         Print STRING in place of NULL values
.output FILENAME          Send output to FILENAME
.output stdout            Send output to the screen
.prompt MAIN CONTINUE     Replace the standard prompts
.quit                     Exit this program
.read FILENAME            Execute SQL in FILENAME
.schema ?TABLE?           Show the CREATE statements
.separator STRING         Change separator used by output mode and .import
.show                     Show the current values for various settings
.tables ?PATTERN?         List names of tables matching a LIKE pattern
.timeout MS               Try opening locked tables for MS milliseconds
.width NUM NUM ...        Set column widths for "column" mode
```

表 5-10 给出 SQLite 一些常用命令及说明。

表 5-10 SQLite 常用命令及说明

命 令	说 明
.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,即开启 EXPLAIN
.header(s) ON|OFF	开启或关闭头部显示

续表

命　令	说　明
.help	显示帮助消息
.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(μs)
.width NUM NUM	为 column 模式设置列宽度
.timer ON|OFF	开启或关闭 CPU 定时器测量

值得注意的是,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。

下面举几个命令行的例子(本章例子均在 Linux 下运行,内核版本 2.6.38)。

(1) 备份和还原数据库。

在当前连接的 main 数据库中创建一个数据表,之后再通过.backup 命令将 main 数据库备份到/sqlite1/my.db 文件中。

```
sqlite > CREATE TABLE mytable (first_col integer);
sqlite > .backup '/sqlite1/my.db '
sqlite > .exit
```

(2) DUMP 数据表的创建语句到指定文件。

先将命令行当前的输出重定向到/sqlite1/myoutput.txt,之后再将之前创建的 mytable 表的声明语句输出到该文件。

```
sqlite> .output /sqlite1/myoutput.txt
sqlite> .dump mytabl%
sqlite> .exit
```

(3) 显示当前连接的所有 Attached 数据库和 main 数据库。

```
sqlite> ATTACH DATABASE '/sqlite1/my.db' AS mydb;
sqlite> .databases
seq    name                          file
---    ----------------   -------------------------
0      main
2      mydb                          /sqlite1/my.db
```

(4) 显示 main 数据库中的所有数据表。

```
sqlite> .tables
mytable
```

(5) 显示匹配表名 mytabl%的数据表的所有索引。

```
sqlite> CREATE INDEX myindex on mytable(first_col);
sqlite> .indices mytabl%
myindex
```

(6) 显示匹配表名 mytable%的数据表的 Schema 信息,依赖该表的索引信息也被输出。

```
sqlite> .schema mytabl%
CREATE TABLE mytable (first_col integer);
CREATE INDEX myindex on mytable(first_col);
```

5.7 SQLite 的 C/C++接口

本节介绍 C/C++接口的相关知识。SQLite 有超过 225 个 API,此外还有一些数据结构和预定义。然而,大多数 API 是可选的,非常专业。核心应用编程接口小,简单易于学习。

从功能的角度来区分,SQLite 的 API 可分为两类:核心 API 和扩充 API。核心 API 由所有完成基本数据库操作的函数构成,主要包括连接数据库、执行 SQL 和遍历结果集等。它还包括一些功能函数,用来完成字符串格式化、操作控制、调试和错误处理等任务。扩充 API 提供不同的方法来扩展 SQLite,它向用户提供创建自定义的 SQL 扩展,并与 SQLite 本身的 SQL 相集成等功能。

以下 2 个对象和 8 个函数构成了 SQLite 接口的关键元素。

sqlite3:数据库连接对象。由 sqlite3_open()创建,由 sqlite3_close()销毁。

sqlite3_stmt：准备好的语句对象。由 sqlite3_prepare()创建，并由 sqlite3_finalize()销毁。sqlite3_stmt 对象的一个实例表示一条已编译成二进制形式并准备好进行计算的 SQL 语句。也就是说，把每一条 SQL 语句看作一个独立的计算机程序，原始的 SQL 文本是源代码，准备好的语句对象 sqlite3_stmt 是编译后的目标代码，在运行之前，所有的 SQL 都必须转换成一个准备好的语句。

sqlite3_open()：打开与新的或现有的 SQLite 数据库的连接。SQLite3 的构造函数。

sqlite3_prepare()：将 SQL 文本编译成字节码，用于查询或更新数据库。sqlite3_stmt 的构造函数。

sqlite3_bind()：将应用程序数据存储到原始 SQL 的参数中。

sqlite3_step()：将 sqlite3_stmt 推进到下一个结果行或完成。

SqLite3_column()：SQLite3_stmt 的当前结果行中的列值。

Sqlite3_finalize()：SQLite3_stmt 的析构函数。

SqLite3_close()：SQLite3 的析构函数。

sqlite3_exec()：为一个或多个 SQL 语句的字符串执行 sqlite3_prepare()、sqlite3_step()、sqlite3_column()和 sqlite3_finalize()的包装函数。


视频讲解

5.7.1 核心 C API 函数

1. 预编译查询

核心 C API 主要与执行 SQL 命令有关。核心 C API 大约有 10 个，它们分别如下。

```
sqlite3_open()
sqlite3_prepare()
sqlite3_step()
sqlite3_column()
sqlite3_finalize()
sqlite3_close()
sqlite3_exec()
sqlite3_get_table()
sqlite3_reset()
sqlite3_bind()
```

有两种方法执行 SQL 语句：预编译查询(Prepared Query)和封装查询。预编译查询由三个阶段构成：准备(preparation)、执行(execution)和定案(finalization)。封装查询只是对预编译查询的三个过程进行了封装，最终也会转化为预编译查询来执行。

预处理查询是 SQLite 执行所有 SQL 命令的方式，主要包括以下三个步骤：

1) 准备

分词器、分析器和代码生成器把 SQL 语句编译成虚拟机字节码，编译器会创建一个语句句柄(sqlite3_stmt)，它包括字节码以及其他执行命令和遍历结果集所需的全部资源。相应的 C API 为 sqlite3_prepare()，位于 prepare.c 文件中，有多种类似的形式，如 sqlite3_prepare()、sqlite3_prepare16()、sqlite3_prepare_v2()等。完整的 API 语法如下：

```
int sqlite3_prepare(
  sqlite3 * db,
        /* db 为 sqlite3 的句柄 */
  const char * zSql,
    /* zSql 为要执行的 SQL 语句 */
  int nByte,
         /* nByte 为要执行语句在 zSql 中的最大长度,如果是负数,那么就需要重新自动计算 */
  sqlite3_stmt * * ppStmt,  /* ppStmt 为预编译后的句柄 */
  const char * * pzTail     /* pzTail 预编译后剩下的字符串(未预编译成功或者多余的)的指
                               针,一般传入 0 或者 NULL 即可 */
);
```

sqlite3_prepare 接口把一条 SQL 语句编译成字节码留给后面的执行函数。使用该接口访问数据库是当前比较好的一种方法。

sqlite3_prepare16()原型如下：

```
int sqlite3_prepare16(sqlite3 * ,const void * ,int,sqlite3_stmt ** ,const void ** );
```

sqlite3_prepare()处理的 SQL 语句是 UTF-8 编码的，而 sqlite3_prepare16()则要求是 UTF-16 编码的。

相对于 sqlite3_prepare()来说，sqlite3_prepare_v2()提供了一个更好的接口，保留旧的 sqlite3_prepare()是为了向后兼容。建议使用 sqlite3_prepare_v2()，而不是 sqlite3_prepare()。

sqlite3_prepare_v2()原型如下：

```
int sqlite3_prepare_v2(
  sqlite3 * db,             /* db 为 sqlite3 的句柄 */
  const char * zSql,        /* zSql 为要执行的 SQL 语句 */
  int nByte,                /* nByte 为要执行语句在 zSql 中的最大长度,如果是负数,那么就需
                               要重新自动计算 */
  sqlite3_stmt * * ppStmt,  /* ppStmt 为预编译后的句柄 */
  const char * * pzTail     /* pzTail 预编译后剩下的字符串(未预编译成功或者多余的)的指针 */
);
```

2) 执行

虚拟机执行字节码的执行过程是一个步进(stepwise)的过程，每一步由 sqlite3_step()启动，并由虚拟机执行一段字节码。当第一次调用 sqlite3_step()时，一般会获得一种锁，锁的种类由命令要做什么(读或写)决定。对于 SELECT 语句，每次调用 sqlite3_step()使用语句句柄的游标移到结果集的下一行。对于其他 SQL 语句(INSERT、UPDATE、DELETE 等)，第一次调用 sqlite3_step()会导致 VDBE 执行整个命令。在 SQL 声明准备好之后，需要调用以下的方法来执行：

```
int sqlite3_step(sqlite3_stmt * );
```

如果 SQL 返回了一个单行结果集，sqlite3_step() 函数将返回 SQLITE_ROW；如果 SQL 语句执行成功或者正常将返回 SQLITE_DONE，否则将返回错误代码。如果不能打开

数据库文件,则会返回 SQLITE_BUSY;如果函数的返回值是 SQLITE_ROW,那么下列方法可以用来获得记录集行中的数据。

```
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);
int sqlite3_column_count(sqlite3_stmt * );
const char * sqlite3_column_decltype(sqlite3_stmt * , int iCol);
const void * sqlite3_column_decltype16(sqlite3_stmt * , int iCol);
double sqlite3_column_double(sqlite3_stmt * , int iCol);
int sqlite3_column_int(sqlite3_stmt * , int iCol);
long long int sqlite3_column_int64(sqlite3_stmt * , int iCol);
const char * sqlite3_column_name(sqlite3_stmt * , int iCol);
const void * sqlite3_column_name16(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);
```

sqlite3_column_count()函数返回结果集中包含的列数。sqlite3_column_count()可以在执行了 sqlite3_prepare()之后的任何时刻调用。sqlite3_data_count()除了必须要在 sqlite3_step()之后调用之外,其他与 sqlite3_column_count()大同小异。如果调用 sqlite3_step()返回值是 SQLITE_DONE 或者一个错误代码,则此时调用 sqlite3_data_count()将返回 0,然而 sqlite3_column_count()仍然会返回结果集中包含的列数。

返回的记录集通过使用其他的几个 sqlite3_column_ * ()函数来提取。所有的这些函数都把列的编号作为第二个参数。列编号从左到右以零起始,请注意它和之前那些从 1 起始的参数的不同。

sqlite3_column_type()函数返回第 N 列的值的数据类型。具体的返回值如下:

```
#define SQLITE_INTEGER      1
#define SQLITE_FLOAT        2
#define SQLITE_TEXT         3
#define SQLITE_BLOB         4
#define SQLITE_NULL         5
```

sqlite3_column_decltype()用来返回该列在 CREATE TABLE 语句中声明的类型。它可以用在当返回类型是空字符串时,sqlite3_column_name()返回第 N 列的字段名。sqlite3_column_bytes()用来返回 UTF-8 编码的 BLOBs 列的字节数或者 TEXT 字符串的字节数。sqlite3_column_bytes16()对于 BLOBs 列返回同样的结果,但是对于 TEXT 字符串则按 UTF-16 的编码来计算字节数。sqlite3_column_blob()返回 BLOB 数据。sqlite3_column_text()返回 UTF-8 编码的 TEXT 数据。sqlite3_column_text16()返回 UTF-16 编码的 TEXT 数据。sqlite3_column_int()以本地主机的整数格式返回一个整数值。sqlite3_column_int64()返回一个 64 位的整数。最后 sqlite3_column_double()返回浮点数。

需要注意的是,不一定非要按照 sqlite3_column_type()接口返回的数据类型来获取数据,数据类型不同时软件将自动转换。

3）定案

虚拟机关闭语句，释放资源。相应的 C API 为 sqlite3_finalize()，它导致虚拟机结束程序运行并关闭语句句柄。如果事务是由人工控制开始的，它必须由人工控制进行提交或回卷，否则 sqlite3_finalize()会返回一个错误。当 sqlite3_finalize()执行成功，所有与语句对象关联的资源都将被释放。在自动提交模式下，还会释放关联的数据库锁。

综合来看，预编译查询的执行流程如图 5-4 所示。

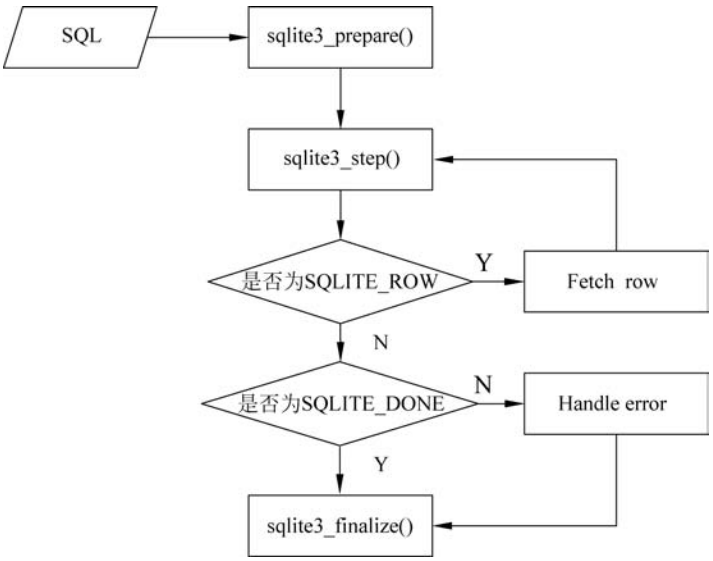

图 5-4 预编译查询执行流程图

封装查询简单地将上述三个步骤封装成一个函数引用，使得系统在某些环境下执行特定指令时非常便利。但无论是哪一种方法，都遵循这样一个规则：方法的封装性越强，它在执行和获取结果方面上的控制性将越差。因此，预编译查询与封装查询相比，提供了更多的特征、更多的控制以及更多的信息。

另一方面，在实际应用中，每一种查询都有适合它自己的用途。函数更适合执行如创建、删除、插入以及更新等修改类命令，而预编译查询则更为适合执行查询类的命令。

2. sqlite3_open()或 sqlite3_open16()函数

打开数据库用 sqlite3_open()或 sqlite3_open16()函数，它们的声明如下：

```
int sqlite3_open(
    const char *filename,                    /* 数据库文件名 (UTF-8) */
    sqlite3 **ppDb                           /* 输出数据库句柄 */
);
int sqlite3_open16(
    const void *filename,                    /* 数据库文件名 (UTF-16) */
    sqlite3 **ppDb                           /* 输出数据库句柄 */
);
```

其中，filename 参数可以是一个操作系统文件名，或字符串，或一个空指针(NULL)。如果使用后两者将创建内存数据库。当 filename 不为空时，函数先尝试打开，如果文件不

存在,则用该名字创建一个新的数据库。

在 SQLite 中,数据库通常是存储在磁盘文件中的。然而在有些情况下,可以让数据库始终驻留在内存中。最常用的一种方式是在调用 sqlite3_open()时,数据库文件名参数传递":memory:",如:

```
rc = sqlite3_open(":memory:", &db);
```

在调用完以上函数后,不会有任何磁盘文件被生成,取而代之的是,一个新的数据库在纯内存中被成功创建了。由于没有持久化,该数据库在当前数据库连接被关闭后就会立刻消失。需要注意的是,尽管多个数据库连接都可以通过上面的方法创建内存数据库,然而它们却是不同的数据库,相互之间没有任何关系。事实上,我们也可以通过 ATTACH 命令将内存数据库像其他普通数据库一样,附加到当前的连接中,如:

```
ATTACH DATABASE ':memory:' AS aux1;
```

在调用 sqlite3_open()函数或执行 ATTACH 命令时,如果数据库文件参数传的是空字符串,那么一个新的临时文件将被创建作为临时数据库的底层文件,如:

```
rc = sqlite3_open("", &db);
```

或

```
ATTACH DATABASE '' AS aux2;
```

和内存数据库非常相似,两个数据库连接创建的临时数据库也是各自独立的,在连接关闭后,临时数据库将自动消失,其底层文件也将被自动删除。尽管磁盘文件被创建用于存储临时数据库中的数据信息,但是实际上临时数据库也会和内存数据库一样通常驻留在内存中。唯一不同的是,当临时数据库中数据量过大时,SQLite 为了保证有更多的内存可用于其他操作,会将临时数据库中的部分数据写到磁盘文件中,而内存数据库则始终会将数据存放在内存中。

3. sqlite3_close()函数

关闭数据库用 sqlite3_close()函数,声明如下:

```
int sqlite3_close(sqlite3 * );
```

为了 sqlite3_close()能够成功执行,所有与连接所关联的且已编译的查询必须被定案。如果仍然有查询没有定案,sqlite3_close()将返回 SQLITE_BUSY 和错误信息。

4. sqlite3_exec()函数

对于用户而言,在 SQLite C/C++ API 中使用频率最高的 3 个函数是:sqlite3_open(),sqlite3_close()和 sqlite3_exec()。sqlite3_exec()的作用是解析并执行由 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。大部分 SQL 操作都可以通过 sqlite3_exec 来完成,它的 API 形式如下:

```
int sqlite3_exec(
sqlite3 * ,                                  / * 数据库句柄 * /
const char * sql,                            / * 要执行的 SQL 语句 * /
int ( * callback)(void * ,int,char ** ,char ** ),    / * callback 回调函数 * /
void * ,                                     / * void * 回调函数的第一个参数 * /
char ** errmsg                               / * errmsg 错误信息,如果没有 SQL 问题则值为 NULL * /
);
```

回调函数是一个比较复杂的函数。原型如下：

```
int callback(void * params,int column_size,char ** column_value,char ** column_name)
```

参数说明如下。

params 是 sqlite3_exec 传入的第四个参数。

column_size 是结果字段的个数。

column_value 是返回记录的一位字符数组指针。

column_name 是结果字段的名称。

通常情况下 callback 在 select 操作中会使用到，尤其是处理每一行记录数时。返回的结果每一行记录都会调用下“回调函数”。如果回调函数返回了非 0，那么 sqlite3_exec 将返回 SQLITE_ABORT，并且之后的回调函数也不会执行，同时未执行的子查询也不会继续执行。

对于更新、删除、插入等不需要回调函数的操作，sqlite3_exec 的第三、第四个参数可以传入 0 或者 NULL。

通常情况下，sqlite3_exec 返回 SQLITE_OK＝0 的结果，非 0 结果可以通过 errmsg 来获取对应的错误描述。在 SQLite3 里 sqlite3_exec 可以被接口封装起来使用。

5. sqlite3_bind

SQL 声明可以包含一些型如“?”或“? nnn”或“:aaa”的标记，其中“nnn”是一个整数，“aaa”是一个字符串。这些标记代表一些不确定的字符值(或者通配符)，用户可以在后面用 sqlite3_bind 接口来填充这些值。每一个通配符都被分配了一个编号(由它在 SQL 声明中的位置决定，从 1 开始)。相同的通配符可以在同一个 SQL 声明中出现多次。在这种情况下所有相同的通配符都会被替换成相同的值，没有被绑定的通配符将自动取 NULL 值。

```
int sqlite3_bind_blob(sqlite3_stmt * , int, const void * , int n, void( * )(void * ));
int sqlite3_bind_double(sqlite3_stmt * , int, double);
int sqlite3_bind_int(sqlite3_stmt * , int, int);
int sqlite3_bind_int64(sqlite3_stmt * , int, long long int);
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 n, void( * )(void * ));
int sqlite3_bind_value(sqlite3_stmt * , int, const sqlite3_value * );
```

以上是 sqlite3_bind 所包含的全部接口，其功能是给 SQL 声明中的通配符赋值。没有绑定的通配符则被认为是空值。绑定上的值不会被 sqlite3_reset()函数重置，但是在调用了 sqlite3_reset()之后所有的通配符都可以被重新赋值。sqlite3_reset()函数用来重置一个

SQL 声明的状态,使得它可以被再次执行。

6. sqlite3_get_table()函数

sqlite3_get_table 的说明如下:

```
int sqlite3_get_table(
 sqlite3 * db,
            /* db 是 sqlite3 的句柄 */
 const char * zSql,
    /* zSql 是要执行的 sql 语句 */
 char *** pazResult,
  /* pazResult 是执行查询操作的返回结果集 */
 int * pnRow,
         /* pnRow 是记录的行数 */
 int * pnColumn,          /* pnColumn 是记录的字段个数 */
 char * * pzErrmsg        /* pzErrmsg 是错误信息 */
);
```

由于 sqlite3_get_table 是 sqlite3_exec 的包装,因此返回的结果和 sqlite3_exec 类似。

5.7.2 扩充 C API 函数

SQLite 的扩充 API 用来支持用户自定义的函数、聚合和排序法。用户自定义函数是一个 SQL 函数,它对应于用 C 语言或其他语言实现的函数的句柄。使用 C API 时,这些句柄用 C/C++实现。用户自定义函数可以在注册之后像系统内置函数一样应用于语句中。自定义函数的使用类似于存储过程,不但方便了用户对常见功能的调用,也使得数据库执行的速度得到了较大的提高。

1. 简单函数和聚合函数

用户自定义函数从整体上可以分为两类:简单函数和聚合函数。其中,简单函数可以用在任何的表达式中,聚合函数经常用在 select 语句中。

sqlite3_create_function()函数用于注册或者删除用户自定义函数。其声明如下所示:

```
typedef struct sqlite3_value sqlite3_value;
  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 * )
  );
  int sqlite3_create_function16(
    sqlite3 * ,
    const void * zFunctionName,
    int nArg,
    int eTextRep,
    void * ,
```

```
    void ( * xFunc)(sqlite3_context * ,int,sqlite3_value * * ),
    void ( * xStep)(sqlite3_context * ,int,sqlite3_value * * ),
    void ( * xFinal)(sqlite3_context * )
);
#define SQLITE_UTF8       1
#define SQLITE_UTF16      2
#define SQLITE_UTF16BE    3
#define SQLITE_UTF16LE    4
#define SQLITE_ANY        5
```

sqlite3_create_function16()和 sqlite_create_function()的不同就在于自定义的函数名一个要求是 UTF-16 编码,而另一个则要求是 UTF-8 编码。

自定义函数传递参数有两种方式,第一种是在注册时用 pUserData 传入,第二种是在调用已经注册的函数时传入参数。

对于简单函数而言,只需要设置 xFunc 参数,而把 xStep 和 xFinal 设为 NULL。但是对于聚合函数而言,则需要设置 xStep 和 xFinal 参数,而把 xFunc 设为 NULL。

其他的用户自定义函数接口主要还有如下几种。

Void(* func)(sqlite3_context * ,int,sqlite3_value **)是一个回调函数,第一个参数表示用户自定义函数的格式,第二个参数表示自定义函数的参数个数,第三个参数表示自定义函数的值。

Void * sqlite3_user_data(sqlite3_context *)函数用以返回用户注册函数时传入的参数 void * pUserData。

用户自定义聚合函数的代码部分主要包含了两个回调函数的编写以及聚合函数的注册,其基本步骤如下:用户自定义所用聚合函数的状态结构,利用 sqlite3_aggregate_context (sqlite3_context * ,sizeof(struct_custom_agg))分配状态结构空间,多次调用 xStep(),在查询结果的每一行上运行 xStep()进行数据处理,在 xFinal()中,利用 sqlite3_aggregate_context (sqlite3_context * ,0)得到状态结构,并且设置返回值,利用 sqlite3_create_function()注册聚合函数,在 SQL 语句中调用聚合函数。

下面的函数用来从 sqlite3_value 结构体中提取数据:

```
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 * );
long long int sqlite3_value_int64(sqlite3_value * );
const unsigned char * sqlite3_value_text(sqlite3_value * );
const void * sqlite3_value_text16(sqlite3_value * );
int sqlite3_value_type(sqlite3_value * );
```

上面的函数调用以下的 API 来获得上下文内容和返回结果:

```
void * sqlite3_aggregate_context(sqlite3_context * , int nbyte);
void * sqlite3_user_data(sqlite3_context * );
void sqlite3_result_blob(sqlite3_context * , const void * , int n, void( * )(void * ));
```

```
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_error(sqlite3_context*, const char*, int);
void sqlite3_result_error16(sqlite3_context*, const void*, int);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_text16(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
void *sqlite3_get_auxdata(sqlite3_context*, int);
void sqlite3_set_auxdata(sqlite3_context*, int, void*, void (*)(void*))
```

用户自定义函数注册的流程如图 5-5 所示。

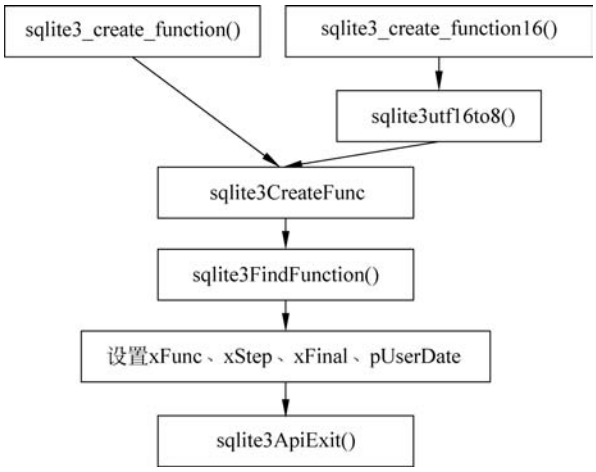

图 5-5　用户自定义函数注册流程图

2. 排序

总体来说,排序包含了对字符的排序以及对字符串的排序。排序通常采用排序序列的方法。一个排序序列就是一个字符清单,清单中的字符已经由确定位置的数字值安排好。这个按顺序排列的清单用于指明字符是如何排序的。排序可以辨别出系统任意给定的两个字符或字符串的先后顺序。

下面的函数用来实现用户自定义的排序规则:

```
int sqlite3_create_collation(
sqlite3* db,
const char*zName,
int pref16,
void* pUserData,
int(*xcompare)(void*,int,const void*,int,const void*)
);
```

sqlite3_create_collation()函数主要用来声明一个排序序列以及实现它的比较函数。其中,比较函数 int(*xcompare)(void*,int,const void*,int,const void*)只能用来进行文本的比较。

5.8 SQLite 工具

SQLite 提供了 7 个工具帮助用户更好地使用 SQLite。它们分别是：命令行 Shell(在 Windows 中是 sqlite3.exe,以下均是)、数据分析器 Analyzer(sqlite3_analyzer.exe)、RBU、数据库文件比较程序(sqldiff.exe)、数据库哈希(dbhash.exe)、Fossil 以及 SQLite 存档程序(sqlar.exe)。下面分别简要介绍这些工具。

5.8.1 命令行 Shell

启动 sqlite3 程序,仅仅需要敲入带有 SQLite 数据库名字的“sqlite3”命令即可。如果文件不存在,则创建一个新的数据库文件。然后 sqlite3 程序将提示输入 SQL,输入 SQL 语句(以分号“;”结束)并按 Enter 键之后,SQL 语句就会执行。

例如,创建一个名字为“test”的 SQLite 数据库,如下：

```
sqlite3 test
SQLite version 3.7.14
Enter ".help" for instructions
sqlite>
```

5.8.2 数据分析器

和 PostgreSQL 非常相似,SQLite 中的数据分析器 sqlite3_analyzer 也同样用于分析数据表和索引中的数据,并将统计结果存放于 SQLite 的内部系统表中,以便根据分析后的统计数据选择最优的查询执行路径,从而提高整个查询的效率。sqlite3_analyzer 程序是一个 TCL 程序,它使用 dbstat 虚拟表来收集关于数据库文件的信息,然后巧妙地格式化这些信息。该工具可以携带参数,如图 5-6 所示。

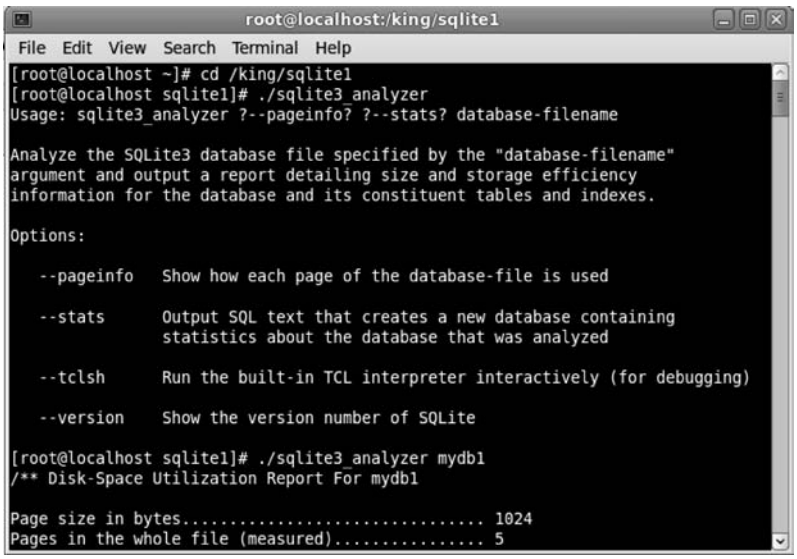

图 5-6 sqlite3_analyzer 参数

图 5-7～图 5-9 显示了对本章使用的 mydb1 数据库的分析信息。

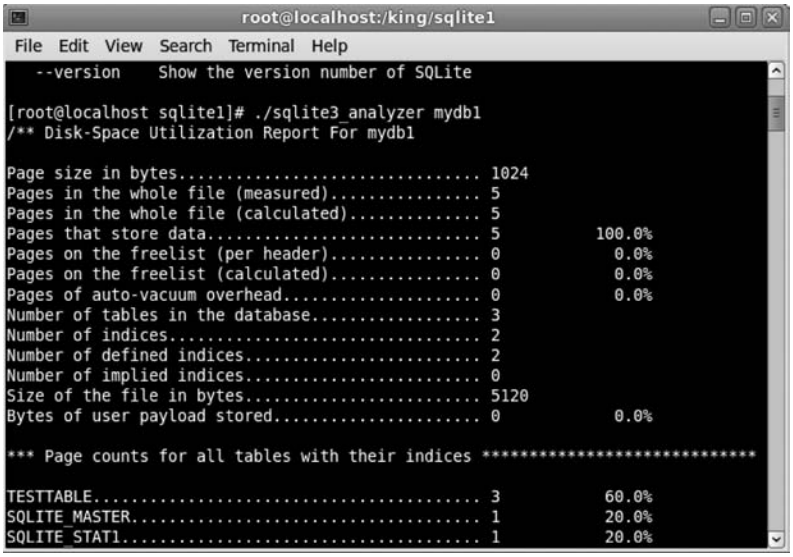

图 5-7 sqlite3_analyzer 对 mydb1 数据库分析情况(1)

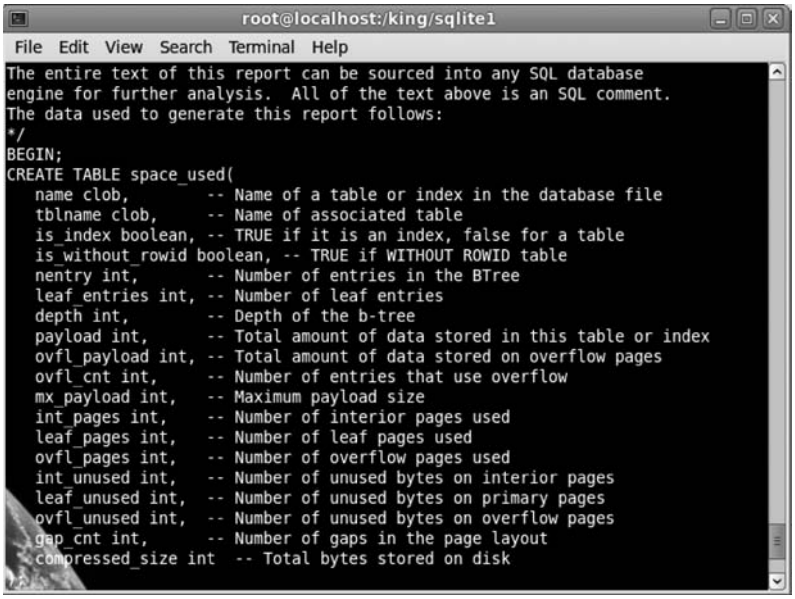

图 5-8 sqlite3_analyzer 对 mydb1 数据库分析情况(2)

5.8.3 可恢复批量更新

可恢复批量更新(Resumable Bulk Update,RBU)实用程序允许以可恢复且不中断正在进行的操作的方式,将一批更改应用于运行在嵌入式硬件上的远程数据库。

RBU 扩展是 SQLite 的一个附加工具,设计用于网络边缘的低功耗设备上的大型 SQLite 数据库文件。RBU 可以用于以下两个不同的任务。

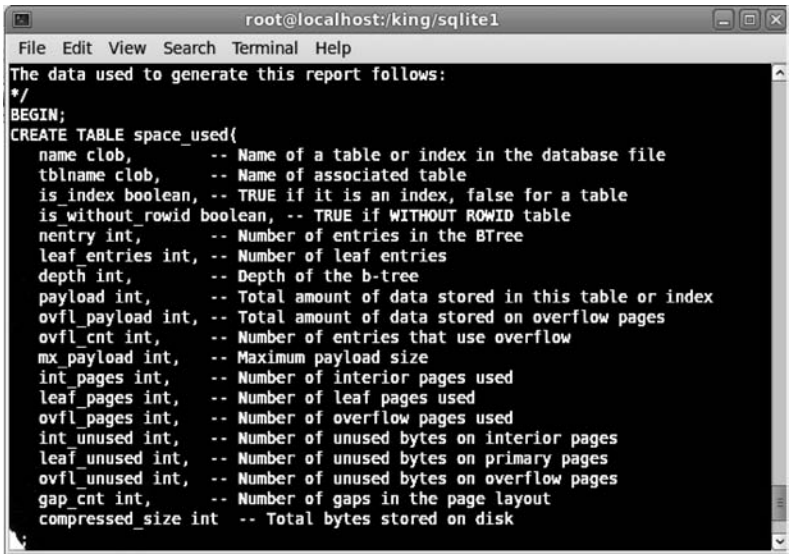

图 5-9 sqlite3_analyzer 对 mydb1 数据库分析情况(3)

(1) RBU 更新操作。RBU 更新是数据库文件的批量更新,可包括对一个或多个表的许多插入、更新和删除操作。

(2) RBU Vacuum 操作。RBU Vacuum 优化和重建整个数据库文件,其结果类似于 SQLite 的本机 Vacuum 命令。

5.8.4 数据库文件比较程序

数据库文件比较程序(SQLite Database Diff)比较两个 SQLite 数据库文件,并输出将一个数据库文件转换成另一个数据库文件所需的 SQL 脚本。下面是它的指令格式:

```
sqldiff [options] database1.sqlite database2.sqlite
```

通常的输出是一个将 database1.sqlite(源数据库)转换成 database2.sqlite(目标数据库)的 SQL 脚本。

SQLite Database Diff 的参数情况如图 5-10 所示。

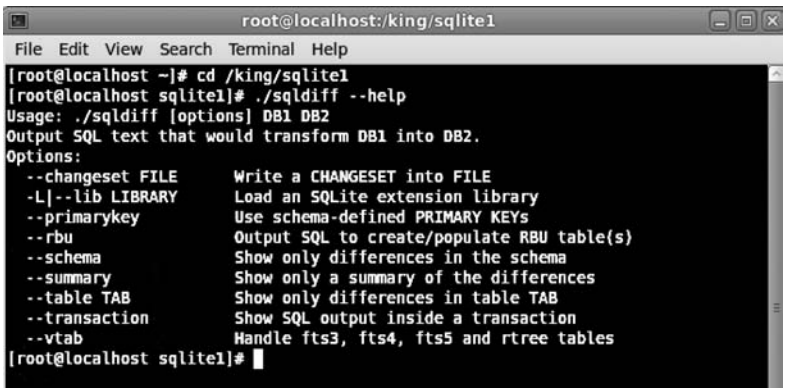

图 5-10 SQLite Database Diff 的参数

5.8.5 数据库哈希

数据库哈希(Database Hash,Dbhash)程序演示了如何计算 SQLite 数据库内容的散列。Dbhash 实用程序是一个命令行程序,用于计算 SQLite 数据库的模式和内容的 SHA1 哈希。

Dbhash 忽略无关的格式细节,只散列数据库模式和内容。因此,即使数据库文件被修改为:

```
VACUUM
PRAGMA page_size
PRAGMA journal_mode
REINDEX
ANALYZE
copied via the backup API
…
```

上述操作可能会导致原始数据库文件发生巨大变化,并因此导致文件级别的 SHA1 哈希非常不同。由于数据库文件中表示的内容通过这些操作没有改变,因此由 Dbhash 计算的散列也没有改变。

Dbhash 可以用来比较两个数据库,以确认它们是等价的,即使它们在磁盘上的表示是完全不同的。Dbhash 也可以用来验证远程数据库的内容,而不必通过慢速链接传输远程数据库的全部内容。

5.8.6 Fossil

Fossil 版本控制系统是一个分布式 VCS,专门设计用于支持 SQLite 开发。Fossil 使用 SQLite 作为存储。

5.8.7 SQLite 存档程序

SQLite 存档程序(SQLite Archiver)是一个使用 SQLite 进行存储的类似 ZIP 的归档程序。该程序(名为“sqlar”)的操作非常类似于“zip”,只是它构建的压缩存档文件存储在 SQLite 数据库中,而不是 ZIP 存档文件中。

5.9 实例代码

本节介绍几个 SQLite 的实例。

5.9.1 获取表的 Schema 信息

主要步骤如下。

(1) 动态创建表。

(2) 根据 SQLite3 提供的 API,获取表字段的信息,如字段数量以及每个字段的类型。

(3) 删除该表。

过程见以下代码及关键性注释:

```
#include <sqlite3.h>
#include <string>
using namespace std;
void doTest() {
sqlite3 * conn = NULL;
//1. 打开数据库
int result = sqlite3_open("/sqlite1/mytest.db",&conn);
if (result != SQLITE_OK) {
sqlite3_close(conn);
return;
}
const char * createTableSQL = "CREATE TABLE TESTTABLE (int_col INT, float_col
REAL, string_col TEXT)";
sqlite3_stmt * stmt = NULL;
int len = strlen(createTableSQL);
/*2. 准备创建数据表,如果创建失败,需要用 sqlite3_finalize 释放 sqlite3_stmt 对象,以防止内
存泄漏*/
if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
if (stmt)
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
/*3. 通过 sqlite3_step 命令执行创建表的语句. 对于 DDL 和 DML 语句而言,sqlite3_step 执行正确
的返回值只有 SQLITE_DONE,对于 SELECT 查询而言,如果有数据返回 SQLITE_ROW,当到达结果集末尾
时则返回 SQLITE_DONE*/
if (sqlite3_step(stmt) != SQLITE_DONE) {
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
//4. 释放创建表语句对象的资源
sqlite3_finalize(stmt);
printf("Succeed to create test table now.\n");
//5. 构造查询表数据的 sqlite3_stmt 对象
const char * selectSQL = "SELECT * FROM TESTTABLE WHERE 1 = 0";
sqlite3_stmt * stmt2 = NULL;
if (sqlite3_prepare_v2(conn,selectSQL,strlen(selectSQL),&stmt2,NULL)
!= SQLITE_OK) {
if (stmt2)
sqlite3_finalize(stmt2);
sqlite3_close(conn);
return;
}
//6. 根据 select 语句的对象,获取结果集中的字段数量
int fieldCount = sqlite3_column_count(stmt2);
printf("The column count is %d.\n",fieldCount);
//7. 遍历结果集中每个字段 meta 信息,并获取其声明时的类型
for (int i = 0; i < fieldCount; ++i) {
```

```
/*由于此时 Table 中并不存在数据,再有就是 SQLite 中的数据类型本身是动态的,所以在没有数据
//时无法通过 sqlite3_column_type 函数获取,此时 sqlite3_column_type 只会返回 SQLITE_NULL,直
//到有数据时才能返回具体的类型,因此这里使用了 sqlite3_column_decltype 函数来获取表声明时
//给出的声明类型*/
string stype = sqlite3_column_decltype(stmt2,i);
stype = strlwr((char*)stype.c_str());
if (stype.find("int") != string::npos) {
printf("The type of %dth column is INTEGER.\n",i);
} else if (stype.find("char") != string::npos || stype.find("text") != string::npos) {
printf("The type of %dth column is TEXT.\n",i);
} else if (stype.find("real") != string::npos || stype.find("floa") != string::npos
|| stype.find("doub") != string::npos ) {
printf("The type of %dth column is DOUBLE.\n",i);
}
}
sqlite3_finalize(stmt2);
/*8. 为了方便下一次测试运行,我们这里需要删除该函数创建的数据表,否则在下次运行时将无法
创建该表,因为它已经存在*/
const char* dropSQL = "DROP TABLE TESTTABLE";
sqlite3_stmt* stmt3 = NULL;
if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt3,NULL) != SQLITE_OK)
{
if (stmt3)
sqlite3_finalize(stmt3);
sqlite3_close(conn);
return;
}
if (sqlite3_step(stmt3) == SQLITE_DONE) {
printf("The test table has been dropped.\n");
}
sqlite3_finalize(stmt3);
sqlite3_close(conn);
}
int main() {
doTest();
return 0;
}
//输出结果为:
//Succeed to create test table now.
//The column count is 3.
//The type of 0th column is INTEGER.
//The type of 1th column is DOUBLE.
//The type of 2th column is TEXT.
//The test table has been dropped.
```

5.9.2 数据插入

主要步骤如下。

(1) 创建测试数据表。

(2) 通过 INSERT 语句插入测试数据。

(3) 删除测试表。

过程见以下代码及关键性注释：

```
#include <sqlite3.h>
#include <string>
#include <stdio.h>
using namespace std;
void doTest() {
sqlite3 * conn = NULL;
//1. 打开数据库
int result = sqlite3_open("/sqlite1/mytest.db",&conn);
if (result != SQLITE_OK) {
sqlite3_close(conn);
return;
}
const char * createTableSQL = "CREATE TABLE TESTTABLE (int_col INT,
float_col REAL, string_col TEXT)";
sqlite3_stmt * stmt = NULL;
int len = strlen(createTableSQL);
/*2. 准备创建数据表,如果创建失败,需要用 sqlite3_finalize 释放 sqlite3_stmt 对象,以防止内
存泄漏*/
if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
if (stmt)
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
/*3. 通过 sqlite3_step 命令执行创建表的语句.对于 DDL 和 DML 语句而言,sqlite3_step 执行正确
的返回值只有 SQLITE_DONE,对于 SELECT 查询而言,如果有数据返回 SQLITE_ROW,当到达结果集末尾
时则返回 SQLITE_DONE*/
if (sqlite3_step(stmt) != SQLITE_DONE) {
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
//4. 释放创建表语句对象的资源
sqlite3_finalize(stmt);
printf("Succeed to create test table now.\n");
int insertCount = 10;
//5. 构建插入数据的 sqlite3_stmt 对象
const char * insertSQL = "INSERT INTO TESTTABLE VALUES(%d,%f,'%s')";
const char * testString = "this is a test.";
char sql[1024];
sqlite3_stmt * stmt2 = NULL;
for (int i = 0; i < insertCount; ++i) {
sprintf(sql,insertSQL,i,i * 1.0,testString);
if (sqlite3_prepare_v2(conn,sql,strlen(sql),&stmt2,NULL) != SQLITE_OK) {
if (stmt2)
```

```
sqlite3_finalize(stmt2);
sqlite3_close(conn);
return;
}
if (sqlite3_step(stmt2) != SQLITE_DONE) {
sqlite3_finalize(stmt2);
sqlite3_close(conn);
return;
}
printf("Insert Succeed.\n");
}
sqlite3_finalize(stmt2);
/*6. 为了方便下一次测试运行,我们这里需要删除该函数创建的数据表,否则在下次运行时将无法
创建该表,因为它已经存在*/
const char* dropSQL = "DROP TABLE TESTTABLE";
sqlite3_stmt* stmt3 = NULL;
if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt3,NULL) != SQLITE_OK)
{
if (stmt3)
sqlite3_finalize(stmt3);
sqlite3_close(conn);
return;
}
if (sqlite3_step(stmt3) == SQLITE_DONE) {
printf("The test table has been dropped.\n");
}
sqlite3_finalize(stmt3);
sqlite3_close(conn);
}
int main() {
doTest();
return 0;
}
//输出结果如下:
//Succeed to create test table now.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//Insert Succeed.
//The test table has been dropped.
```

5.9.3 数据查询

数据查询是每个关系型数据库都会提供的最基本功能,下面的代码示例将给出如何通过

SQLite API 获取数据。

(1) 创建测试数据表。

(2) 插入一条测试数据到该数据表,以便于后面的查询。

(3) 执行 SELECT 语句检索数据。

(4) 删除测试表。

见以下示例代码和关键性注释:

```
#include <sqlite3.h>
#include <string>
#include <stdio.h>
using namespace std;
void doTest() {
sqlite3 * conn = NULL;
//1. 打开数据库
int result = sqlite3_open("/sqlite1/mytest.db",&conn);
if (result != SQLITE_OK) {
sqlite3_close(conn);
return;
}
const char * createTableSQL =
"CREATE TABLE TESTTABLE (int_col INT, float_col REAL, string_col TEXT)";
sqlite3_stmt * stmt = NULL;
int len = strlen(createTableSQL);
/*2. 准备创建数据表,如果创建失败,需要用 sqlite3_finalize 释放 sqlite3_stmt 对象,以防止内存泄漏*/
if (sqlite3_prepare_v2(conn,createTableSQL,len,&stmt,NULL) != SQLITE_OK) {
if (stmt)
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
/*3. 通过 sqlite3_step 命令执行创建表的语句.对于 DDL 和 DML 语句而言,sqlite3_step 执行正确的返回值只有 SQLITE_DONE,对于 SELECT 查询而言,如果有数据返回 SQLITE_ROW,当到达结果集末尾时则返回 SQLITE_DONE*/
if (sqlite3_step(stmt) != SQLITE_DONE) {
sqlite3_finalize(stmt);
sqlite3_close(conn);
return;
}
//4. 释放创建表语句对象的资源
sqlite3_finalize(stmt);
printf("Succeed to create test table now.\n");
//5. 为后面的查询操作插入测试数据
```

```
sqlite3_stmt * stmt2 = NULL;
const char * insertSQL = "INSERT INTO TESTTABLE VALUES(20,21.0,'this is a
test.')";
if (sqlite3_prepare_v2(conn,insertSQL,strlen(insertSQL),&stmt2,NULL) !=
SQLITE_OK) {
if (stmt2)
sqlite3_finalize(stmt2);
sqlite3_close(conn);
return;
}
if (sqlite3_step(stmt2) != SQLITE_DONE) {
sqlite3_finalize(stmt2);
sqlite3_close(conn);
return;
}
printf("Succeed to insert test data.\n");
sqlite3_finalize(stmt2);
//6. 执行 SELECT 语句查询数据
const char * selectSQL = "SELECT * FROM TESTTABLE";
sqlite3_stmt * stmt3 = NULL;
if (sqlite3_prepare_v2(conn,selectSQL,strlen(selectSQL),&stmt3,NULL) !=
SQLITE_OK) {
if (stmt3)
sqlite3_finalize(stmt3);
sqlite3_close(conn);
return;
}
int fieldCount = sqlite3_column_count(stmt3);
do {
int r = sqlite3_step(stmt3);
if (r == SQLITE_ROW) {
for (int i = 0; i < fieldCount; ++i) {
/*这里需要先判断当前记录当前字段的类型,再根据返回的类型使用不同的 API 函数获取实际的数
据值*/
int vtype = sqlite3_column_type(stmt3,i);
if (vtype == SQLITE_INTEGER) {
int v = sqlite3_column_int(stmt3,i);
printf("The INTEGER value is %d.\n",v);
} else if (vtype == SQLITE_FLOAT) {
double v = sqlite3_column_double(stmt3,i);
printf("The DOUBLE value is %f.\n",v);
} else if (vtype == SQLITE_TEXT) {
const char * v = (constch
ar*)sqlite3_column_text(stmt3,i);
printf("The TEXT value is %s.\n",v);
} else if (vtype == SQLITE_NULL) {
printf("This value is NULL.\n");
}
}
} else if (r == SQLITE_DONE) {
```

```
printf("Select Finished.\n");
break;
} else {
printf("Failed to SELECT.\n");
sqlite3_finalize(stmt3);
sqlite3_close(conn);
return;
}
} while (true);
sqlite3_finalize(stmt3);
/*7. 为了方便下一次测试运行,我们这里需要删除该函数创建的数据表,否则在下次运行时将无法
创建该表,因为它已经存在*/
const char* dropSQL = "DROP TABLE TESTTABLE";
sqlite3_stmt* stmt4 = NULL;
if (sqlite3_prepare_v2(conn,dropSQL,strlen(dropSQL),&stmt4,NULL) != SQLITE_OK)
{
if (stmt4)
sqlite3_finalize(stmt4);
sqlite3_close(conn);
return;
}
if (sqlite3_step(stmt4) == SQLITE_DONE) {
printf("The test table has been dropped.\n");
}
sqlite3_finalize(stmt4);
sqlite3_close(conn);
}
int main() {
doTest();
return 0;
}
//输出结果如下:
//Succeed to create test table now.
//Succeed to insert test data.
//The INTEGER value is 20.
//The DOUBLE value is 21.000000.
//The TEXT value is this is a test..
//Select Finished.
//The test table has been dropped.
```

5.10 小结

本章介绍了嵌入式数据库 SQLite 的功能、特点和 SQLite 数据库的相关基础应用情况。当前嵌入式系统软件开发的重要环节之一就是对各种数据的管理,而嵌入式数据库是实现该目标的重要手段。SQLite 数据库的特点十分适合嵌入式产品开发,而且完全免费开源,值得在日常学习中多实践多研究。

习题

1. 简要叙述 SQLite 数据库的主要特点。

2. 下载 SQLite 源码并尝试在指定嵌入式系统中安装 SQLite。

3. SQLite 与其他数据库最大的不同是它对数据类型的支持,简述 SQLite 数据库支持的数据类型。

4. SQLite 拥有一个模块化的体系结构,请简述它的构成子系统。

5. 设计一个数据库,包含学生信息表、课程信息表和成绩信息表。请写出各个表的数据结构的 SQL 语句,以 CREATE TABLE 开头。

6. 向学生信息表和课程信息表各增加 5 条记录数据。请写出增加数据的 SQL 语句,以 INSERT INTO 开头。

7. 删除学生信息表和课程信息表的个别记录数据。请写出删除数据的 SQL 语句,以 DELETE FROM 开头。

8. 修改学生信息表和课程信息表的个别记录数据。请写出修改数据的 SQL 语句,以 UPDATE 开头。

9. 向成绩信息表增加 10 条记录数据。写出增加数据的 SQL 语句,以 INSERT INTO 开头。

10. 完成以下查询,请写出 SQL 语句,以 SELECT FROM 开头:

(1) 学生信息表中有几位学生;

(2) 成绩信息表中有几位学生是满分;

(3) 没有成绩的学生有哪些;

(4) 至少有一位学生选的课程有哪些;

(5) 查询选了三门课并且平均成绩在 85 分以上的学生名单。