第一章 概览

过去的几十年中涌现了许多数据库管理系统(DBMSs)，例如DB-2,Informix,Ingres,MySQL,Oracle,SQL Server, Sybase 等。SQLite 近几年才加入到DBMS 的大家庭中，并且在商业应用上取得了成功。2000 年 5 月 9 日，SQLite 首次发布。该数据库是一个具有以下特性的关系型数据库：

无需配置(Zero configuration)

在使用 SQLite 之前无需进行安装或者配置。数据库只需要很少的管理或者不需要管理，你可以在官方主页 http://www.sqlite.org 下载源代码，使用 C 语言编译器编译成库文件就可以使用了。

嵌入在应用中(Embeddable)

无需为 SQLite 维护一个单独的进程，SQLite 是嵌入在应用程序中的。

为应用程序提供接口(Application interface)

SQLite 为 C 语言应用程序提供 SQL 语言环境来操作数据库。这是一种调用级别的接口库(call-level interface library)。用户可以书写大段的 SQL 语句并把它们通过接口传递给 SQLite。(除了使用 SQL 语言，其他的方式无法操作数据库)。这并不会对应用程序提出额外的预处理和编译要求，一个普通的 C 编译器就可以完成所有的工作。

事务功能(Transactional support)

SQLite 具有事务功能，也即支持原子性，一致性，独立性和持久性(ACID)。当 SQLite 遇到系统崩溃和掉电的时候，无需用户手动修复数据库。当 SQLite 读取一个数据库时，如果数据库是错误的，SQLite 将自动执行修复功能，修复过程对用户来说是透明的。

线程安全(Thread-safe)

SQLite 是一个线程安全库，同一个进程中的多个线程可以同时进入一个或多个数据库。SQLite 维护数据库级别的线程并发控制。

轻量级(Lightweight)

SQLite 大小只有 250KB，如果在编译源代码的时候去掉一些不必要的功能，SQLite 还可以变得更小。SQLite 可以运行在 Linux, Windows, Mac OS X, OpenBSD 和其他的一些操作系统上。

可定制(Customizable)

SQLite 提供了一个完善的框架，用户可以通过这个框架定义和使用 SQL 函数、合计函数(aggregate function)，排序函数。SQL 也支持 UTF-8 和 UTF-16 标准编码格式。

跨平台(Cross-platform)

SQLite 使你能够在平台之间移动数据库文件。例如：你可以在一台 Linux X86 机器上创建一个数据库文件，然后把数据库文件移动到一个 ARM 平台，该数据库文件依然有效。

SQLite 与许多其他的现代 SQL 数据库不同，SQLite 的首要设计目标是简洁，尽管这有时候会导致无法有效支持一些特性。但是维护、商业化、操作、管理、和在应用中嵌入 SQLite 非常简单。SQLite 使用简单的技巧来实现 ACID 特性。

SQLite 支持 SQL-92 标准的众多子标准和操作特性。你可以创建表、索引、触发器和视图，也可以通过 INSERT、DELETE、UPDATE、SELECT 来操作存储的信息，下表是 SQLite3.3.6 发布版支持的额外特性：

- 部分地支持 ALTER TABLE（重命名 table，添加 column）
- UNIQUE,NOT NULL.CHECK 约束
- 子查询，包括关联子查询，INNER JOIN, LEFT OUTER JOIN, NATURAL JOIN, UNION, UNION ALL, INTERSECT, EXCEPT
- 事务命令:BEGIN, COMMIT, ROLLBACK
- SQLite 命令，包括 reindex, attach, detach, pragma

SQLite3.3.6 发布版还不支持下面的 SQL-92 特性：

- 外键约束
- 许多 ALTER TABLE 特性
- 一些触发器相关的特性
- 更新一个 VIEW
- 授予权限(GRANT)和撤销权限(REVOKE)

SQLite 把整个数据库存储在单个普通的本地文件中，该文件可以放置在本地文件系统的任一个路径下。任何一个有读/写该文件权限的用户都可以读/写该文件。SQLite 使用一个独立的日志文件来保存事务回滚信息，当事务 abort 或者系

统崩溃的时候，这些信息被用于修复数据库。Attach 命令帮助事务在不同的数据库中同时工作。这些事务也是具有 ACID 特性的。Pragma 命令使你可以更改 SQLite 库的行为。

SQLite 允许多个应用同时进入同一个数据库，并通过事务管理来支持有限形式的并发访问。SQLite 允许多个读事务同时存在，但是某时刻只能有一个写事务存在。

SQLite 被广泛应用于底层到中层的数据库应用程序，例如网址服务器（SQLite 平均每天支持 100,000 次访问，SQLite 的开发团队表示 SQLite 平均每天可以支持 1,000,000 次访问）、移动电话、PDA、机顶盒、独立应用程序。你甚至可以把 SQLite 作为初级数据库课程的教材，因为源代码已经有了完善的注释和相关技术文档，而且 SQLite 是完全开源的，没有任何证书约束。

1.1 应用实例

1.1.1 一个简单的 SQLite 应用

```
#include <stdio.h>
#include "sqlite3.h"
int main(void)
{
    sqlite3*      db = 0;
    sqlite3_stmt* stmt = 0;
    int retcode;
    retcode = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (retcode != SQLITE_OK){
        sqlite3_close(db);
        fprintf(stderr, "Could not open MyDB\n");
        return retcode;
    }
    retcode = sqlite3_prepare(db, "select SID from Students order by SID",
                                        -1, &stmt, 0);
    if (retcode != SQLITE_OK){
        sqlite3_close(db);
        fprintf(stderr, "Could not execute SELECT\n");
        return retcode;
    }
    while (sqlite3_step(stmt) == SQLITE_ROW){
        int i = sqlite3_column_int(stmt, 0);
        printf("SID = %d\n", i);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return SQLITE_OK;
}
```

你可以编译并执行上面的例子。在本书中，例程的输出是由 Linux 系统产生的，但是这些例程也可以运行在其他平台上。

这些例程假定你已经准备好了 sqlite3 可执行程序、libsqlite3.so(windows 平台的 sqlite3.dll 或者 Mac OS X 平台的 libsqlite3.dylib)和 sqlite3.h。你可以从 http://www.sqlite.org 获取这些源代码或者二进制可执行文件。把 sqlite3、库文件、sqlite3.h 这三个文件放在同一个目录下会让工作变得简单很多。

例如，在 Linux 环境下，将 app1.c、libsqlite3.so、sqlite3、sqlite3.h 放在同一个目录下，用以下命令编译可以得到二进制可执行文件：

```
gcc app1.c -o ./app1 -lsqlite3 –L
```

NOTE

SQLite 的源代码和应用程序的源代码必须用同一个编译器编译。如果你已经把 SQLite 作为一个包安装到电脑上，编译的命令有所不同。例如，在 ubuntu 上可以使用 sudo aptitude install sqlite3 libsqlite3-dev 来安装 SQLite，安装完成后使用 cc app1.c -o ./app1 -lsqlite3 编译文件。

因为 SQLite 已经支持最近的 Mac OS X 版本，所以上面的命令在 Mac OS X 上也能运行。

上例打开了当前目录下的数据库文件 MyDB。一个数据库至少需要一个 table，即 student；这个 table 必须至少有一个整数列即 SID。在下一个例子中，你将学习到如何创建一个新的 table，并在其中插入一行数据，通过下面的命令，你可以创建并更新表格：

```
sqlite3 MyDB "create table students (SID integer)"
sqlite3 MyDB "insert into students values (200)"
sqlite3 MyDB "insert into students values (100)"
sqlite3 MyDB "insert into students values (300)"
```

运行 app1（Linux 下可能需要配置环境变量），将有如下的输出：

```
SID = 100
SID = 200
SID = 300
```

NOTE

在 Linux、Unix、Mac OS X 系统下，可能需要在命令行编辑器中加入前缀./:

./app1

该例程将执行 SQL 语言 select SID from Students order by SID。依次查找 SID 的值并打印出来，最后关闭数据库。

SQLite 是一种嵌入应用程序中的调用级别的接口库(call-level interface library)。这个库中所有的 SQLite API 前缀都是 sqlite3_，它们的声明在 sqlite3.h 中。在上例中我们用到了一些 SQLite API 例如 sqlite3_open，sqlite3_prepare，sqlite3_step，sqlite3_column_int，sqlite3_finalize 和 sqlite3_close。上例中也用了一些助记符用来作为 API 的返回值，例如 SQLITE_OK，SQLITE_ROW。所以助记符的定义也在 sqlite3.h 中。

下面的几小节将介绍一些关键的 API。

1.1.1.1sqlite3_open

通过运行这个函数，应用程序通过 SQLite 库建立一个与数据库文件的连接。（该应用程序对于这个数据库文件或者其他的数据库文件可能还有另外的连接。SQLite 区别对待每个连接，每个连接都是独立的。）如果数据库文件不存在，SQLite 自动创建一个数据库文件。

NOTE

当打开或者创建一个文件时，SQLite 遵循懒惰原则：实际上文件的打开或者创建被推迟，直到 SQLite 要进入文件并读取数据。

SQLite_open 返回一个指向 sqlite3 类型实体的句柄。这个句柄完全代表了这个连接，通过这个句柄可以对数据库连接进行操作。

1.1.1.2sqlite3_prepare

Sqlite3_prepare 编译一条 SQL 语言，并产生一个等价内部字节码。SQLite 内部的虚拟机负责读取并运行这个字节码。

Sqlite3_prepare 返回一个语句句柄(Sqlite3_stmt)，通过这个句柄可以操作字节码。在上面的例子中，将 select SID from Students order by SID 编译成了字节码，

并返回了句柄 stmt。这个句柄的作用就像一个游标，可以获取 table 中每一行的信息。

1.1.1.3sqlite3_step

Sqlite3_step 执行字节码，直到遇到断点（找到新的行）或者停止（没有更多的行），并分别返回 SQLITE_ROW 和 SQLITE_DONE。对于不返回行的 SQL 语句（例如 UPDATE, INSERT, DELETE, CREATE）,该函数返回 SQLITE_DONE。初始化时，游标指向要输出的行的集合的第一行之前。每执行一次 Sqlite3_step 游标就向下移动一行。游标只能向前移动，不能后退。

1.1.1.4sqlite3_column_int

如果 sqlite3_step 返回 SQLITE_ROW，通过 API 函数 sqlite3_conlumn_*就能得到指定的数据。如果 SQL 语句请求读取的数据类型和数据库中的存储类型不一样，sqlite3_conlumn_*会把数据类型转换为 SQL 语句请求的数据类型。在上面的例子中，请求的数据是整型，使用 sqlite3_conlumn_int 函数后得到整型的数据。

1.1.1.5sqlite3_finalize

Sqlite3_finalize 销毁预处理的 SQL 语句。即，擦除字节码程序，释放分配给 SQL 语句句柄的所有资源，该句柄变为无效。

1.1.1.6 sqlite3_close

Sqlite3_close 关闭数据库连接，释放分配给该连接的所有资源。该连接句柄变为无效。

1.1.1.7 其他有用的函数

其他有用的函数是 sqlite3_bind_*和 sqlite3_reset。在输入的 SQL 语句中，一个或者多个参数可以被替换成“?”。替换之后的结果可以作为 sqlite3_prepare 的输入。可以通过 sqlite3_bind_*来设置这些参数的值。如果一个参数没有绑定任何值，那么要么将其设定为默认值，要么将其设置为 SQL NULL（当数据库的 schema 没有生命默认值时）。sqlite3_reset 重置 SQL 语句句柄为其初始状态，但绑定了值的参数并不会被重置，重置之后的句柄可以重复使用，sqlite3_bind_* 函数也可以改变参数绑定的值。

1.1.1.8 返回值

API 返回 0 或者正整数，这些返回值以助记符的形式给出。SQLITE_OK 表示执行成功，SQLITE_ROW 表示 sqlite3_step 函数找到了一个新行，SQLITE_DONE 表示 SQL 语句执行完毕。

1.1.2 SQLite 命令行例程

运行下面的例子后，就可以直接在命令行下操作数据库了。该程序使用两个参数作为输入，第一个参数是数据库文件名字，第二个参数是 SQL 语句。程序的执行过程为：打开数据库文件，使用 sqlite3_exec 执行 SQL 语句，关闭数据库。sqlite3_exec 中包含了 sqlite3_prepare 和 sqlite3_step。

```
#include <stdio.h>
#include "sqlite3.h"
static int callback(void *NotUsed, int argc, char **argv, char **colName)
{
    // Loop over each column in the current row
    int i;
    for (i = 0; i < argc; i++){
        printf("%s = %s\n", colName[i], argv[i] ? argv[i] : "NULL");
    }
    return 0;
}
int main(int argc, char **argv)
{
```

```
    sqlite3* db = 0;
    char* zErrMsg = 0;
    int rc;
    if (argc != 3){
        fprintf(stderr, "Usage: %s DATABASE-FILE-NAME SQL-STATEMENT\n",
argv[0]);
        return -1;
    }
    rc = sqlite3_open(argv[1], &db);
    if (rc != SQLITE_OK){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return -2;
    }
    rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
    if (rc != SQLITE_OK){
        fprintf(stderr, "SQL error: %s\n", zErrMsg);
    }
    sqlite3_close(db);
    return rc;
}
```

将上面的代码编译成 app2，在命令行中输入下列命令就可以在 Student 表中插入新的行了。

```
./app2 MyDB "insert into Students values(100)"
./app2 MyDB "insert into Students values(10)"
./app2 MyDB "insert into Students values(1000)"
```

然后运行 app1，就能得到下面的结果：

```
SID = 10
SID = 100
SID = 100
```

SID = 200

SID = 300

SID = 1000

你也可以在一个数据库中创建新的表，例如，执行./app2 MyDBExtn "create table Courses(name varchar, SID integer)"就能在 MyDBExtn 数据库中创建 Courses 表。

NOTE

SQLite 有一个内部的命令行程序（sqlite3），你可以从官网上下载这个程序的可执行文件或者源代码，app2 是这个程序的核心代码，通过使用这个程序，你就能对数据库进行各种操作了。

1.1.3 多线程

SQLite 是一个线程安全的库，所以同一应用程序中的多个线程可以并发地进入一个或多个数据库。

NOTE

为了实现线程安全，编译源代码时必须将预处理宏 THREADSAFE 设置为 1。（编译时在配置脚本中写入--enable-threadsafe）。Linux 平台上的编译设置默认是禁止多线程并发的，但是 windows 平台上是默认打开的。SQLite 库是否具有并发特性，这在程序中无法查询得到的。

下面的例子用 pthreads 库，该库只存在于 Linux 系统下。在 windows 下你有两个选择，使用 Cygwin 提供的工具和编译器(http://www.cygwin.com)，或者从官网(http://sourceware.org/pthreads-win32/)下载 pthreads for windows。

```
#include <stdio.h>
#include <pthread.h>
#include "sqlite3.h"
void* myInsert(void* arg)
{
    sqlite3*      db = 0;
    sqlite3_stmt* stmt = 0;
    int val = (int)arg;
    char SQL[100];
```

```
    int rc;
    rc = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (rc != SQLITE_OK) {
       fprintf(stderr, "Thread[%d] fails to open the database\n",      val);
       goto errorRet;
    }
    /* Create the SQL string. If you were using string values,
         you would need to use sqlite3_prepare() and sqlite3_bind_*
         to avoid an SQL injection vulnerability. However %d
         guarantees an integer value, so this use of sprintf is
         safe.
     */
    sprintf(SQL, "insert into Students values(%d)", val);
    /* Prepare the insert statement */
    rc = sqlite3_prepare(db, SQL, -1, &stmt, 0);
    if (rc != SQLITE_OK) {
       fprintf(stderr, "Thread[%d] fails to prepare SQL: %s ->
return code %d\n", val, SQL, rc);
       goto errorRet;
    }
    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
       fprintf(stderr,
          "Thread[%d] fails to execute SQL: %s -> return code %d\n", val, SQL, rc);
    }
    else {
       printf("Thread[%d] successfully executes SQL: %s\n", val,
          SQL);
    }
    sqlite3_close(db);
```

```
        return (void*)rc;
}
int main(void)
{
    pthread_t t[10];
    int i;
    for (i=0; i < 10; i++)
       pthread_create(&t[i], 0, myInsert, (void*)i); /* Pass the value of i */
    /* wait for all threads to finish */
    for (i=0; i<10; i++)    pthread_join(t[i], 0);
    return 0;
}
```

这个例子创建 10 个线程，每个线程都要在 MyDB 数据库的 Student 表中插入一行。SQLite 使用的是基于锁的并发控制策略，所以一些或者全部的 INSERT 语句可能执行失败。应用程序无需关心并发控制，只需要检查失败的 INSERT 语句，并且在代码中处理这些失败情况（例如，当 SQL 语句执行失败后，你可以重新执行该语句，或者通知用户 SQL 语句执行失败）。

每个线程都需要创建数据库连接并建立自己的 SQLite 句柄。SQLite 不推荐在线程间分享 SQLite 句柄。尽管分享句柄后程序依然可以运行，但是无法保证得到正确的结果。事实上，如果这样做的话，在一些 Linux 平台上 SQLite 可能会崩溃。同样的，在一些 Unix/Linux 平台上，不能通过 fork 函数将 SQLite 句柄传给子进程，否则也将导致 SQLite 崩溃。

NOTE

在 SQLite 3.3.1 以及之后的版本，放松了对“在线程间分享 SQLite 句柄”的限制。你可以将一个 SQLite 连接从一个线程转移给另一个线程，但是前提条件是前一个线程不再持有任何关于这个 SQLite 连接的本地文件锁。如果没有任何 pending 的事务、所有的 SQL 语句都被 reset 和 finalized，那么这个线程就不持有任何本地文件锁。

1.1.4 操作多个数据库

下面的例程操作了两个数据库：

```
#include <stdio.h>
#include "sqlite3.h"
int main(void)
{
    sqlite3* db = 0;
    sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    sqlite3_exec(db, "attach database MyDBExtn as DB1", 0, 0, 0);
    sqlite3_exec(db, "begin", 0, 0, 0);
    sqlite3_exec(db, "insert into Students values(2000)", 0, 0, 0);
    sqlite3_exec(db, "insert into Courses values('SQLite Database', 2000)", 0, 0,
0);
    sqlite3_exec(db, "commit", 0, 0, 0);
    sqlite3_close(db);
    return SQLITE_OK;
}
```

上面的例子没有包括对错误的处理。该例子首先打开了 MyDB 数据库，然后将 MyDBExtn 附件到当前的连接上。MyDB 需要有一个 Student（SID）表，MyDBExtn 需要有一个 Courses（name，SID）表。Begin 命令开始了一项事务，在 student 表中插入一行，然后在 courses 表中插入一行，最后通过 commit 命令提交事务。INSERT 命令不需要回调函数，所以传给 sqlite3_exec 的回调函数参数是 0。

NOTE

SQLite 允许向单个 sqlite3_exec 函数传入多个 SQL 语句，例如：begin; insert into Students values(2000); insert into Courses values('SQLite Database', 2000); commit.

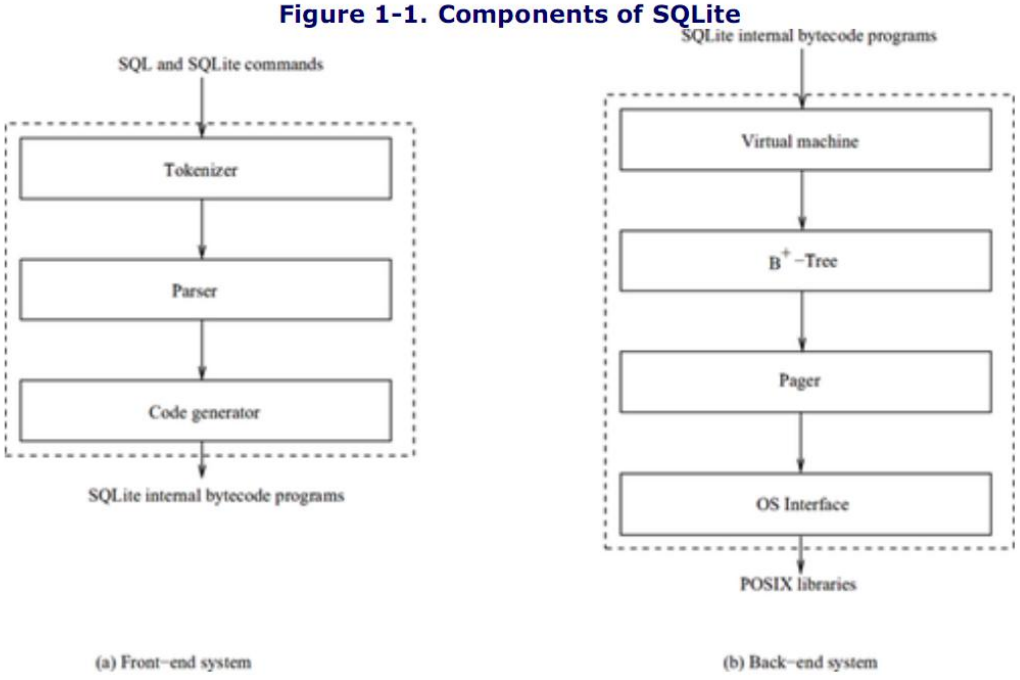
1.1.5 目录（catalog）

对于每个数据库，SQLite 自己创建并维护一个叫做 sqlite_master 的主目录，里面存储着数据库的元信息（模型，表，索引，触发器，视图）。你可以向主目录申请查询（例如，select * from sqlite_master），但是不能直接修改主目录。所有的目录都以 sqlite_为前缀，用户不能使用这个前缀命名变量。

1.2 架构

SQLite 有七个主要的模块，这七个模块被分成两部分：前端解析系统和后台引擎。下面的两个结构图展示了模块之间时怎样交互的。

Figure 1-1. Components of SQLite



1.2.1 前端

应用程序将 SQL 语句或者 SQLite 命令传给 SQLite 的前端系统，该系统负责对传入的语句（命令）进行分析，优化，并转化成后台可以接收并执行的字节码，前端系统由三个模块组成：

标记解析器（tokenizer）

标记解析器把输入的 SQL 语句分解为单个的标记。

语法分析器（parser）

语法分析器通过分析标记解析器生成的标记来确定 SQL 语句的结构，并生成语法树。语法分析器也包含了一个优化器，用来优化和重构语法树，并挑选出一颗合适的语法树用来生成字节码。

字节码生成器（the code generator）

字节码生成器遍历语法树，生成字节码。

所有的前端实现都在 API 函数 sqlite3_prepare 中。

1.2.2 后台

后台是一个解释字节码的引擎，引擎完成实际操作数据库的工作。后台主要

由 4 个模块组成：

虚拟机（The Virtual Machine）

虚拟机是字节码解释器，被用来执行字节码代表的 SQL 语句。虚拟机是数据库的最后一道操作，它将数据库文件视为表和索引的集合。

B/B+树（The B/B+-tree）

B/B+树模块将表和索引分别存储在 B+树和 B 树中。该模块既能帮助虚拟机模块查询、插入和删除树中的某个元组，也能创建新树和删除旧的树。

页管理器（The pager）

Pager 模块在本地文件之上实现了基于页的数据库文件抽象。它维护一个数据库文件页的 cache 在内存中，供 B/B+树模块使用。Pager 模块也管理文件锁和日志文件，实现事务的 ACID 特性。

操作系统接口（The operating system interface）

操作系统接口模块针对不同的本地操作系统提供统一的交互接口。

后端通过 sqlite3_bind_*, sqlite3_step, sqlite3_column_*, sqlite3_reset 和 sqlite3_finalize API functions 这些函数实现。

NOTE

由于篇幅所限，本小节只讨论了 SQLite 引擎，没有讨论前端解析系统。本小节中的例子是运行在 Linux 系统下的，但是 SQLite 支持各种不同的平台，所以这些例子也可以运行在其他平台上。

1.3 SQLite 的局限

与其他许多 SQL 数据库不同，SQLite 的首要设计目标是尽可能保证简单，即使因此损失了一些不是很重要的特性。下面的表列出了 SQLite 不支持的特性：

SQL-92 特性

如前文所述，SQLite 不支持一些 SQL-92 特性，你可以在官网获得最近版本的支持信息。

低并发性

SQLite 只支持平面事务处理，不支持嵌套事务和保存点（savepoint）（嵌套事务指一个事务可以包含几个子事务，保存点允许事务回滚到之前的状态）。SQLite 也不支持高并发的事务处理，只允许读事务并发，不允许写事务并发。即，如果已经存在一个读事务正在读数据库文件，那么就不允许任何写事务写数据库文件。同样，如果已经存在一个写事务正在写数据库文件，那么就不允许任何读/写事务访问数据库文件。

应用限制

由于 SQLite 事务并发的局限性，SQLite 只适应于小型的事务。在许多情况下，这并没有什么问题。每一个应用非常快地操作数据库然后继续下一步，所以没有应用相对长时间的持有一个数据库。但是有一些应用不一样，尤其是主要功能是写入数据库文件的应用程序，它们要求更高的事务并发性（表和列级别的事务并发性而不是数据库级别的），所以推荐使用其他的数据库。SQLite 并不是一个企业级的数据库，它在简单数据库的应用、维护、管理方面的优秀表现要比企业级数据库提供的一些复杂功能更重要。

网络文件系统问题（NFS problem）

SQLite 用本地文件锁控制事务的并发。如果数据库文件在网络上，这种控制方式可能导致问题。许多 NFS 系统的文件逻辑锁都包含一些 bug。如果文件锁没有正常工作，有可能导致不同的应用在同一时刻修改同一数据库的相同部分，这会使得数据库发生崩溃。因为这个问题是 NFS 系统的 bug 导致的，所以 SQLite 并没有什么解决办法。

另外，由于与许多的网络文件系统有关，SQLite 的表现并不会很好。在这种情况下，需要跨网络访问数据库文件，所以一些其他的客户/服务器架构的数据库比 SQLite 更加适合。

数据库大小

对于很大的数据量，SQLite 并不是一个很好的选择。理论上来说，一个数据库文件的大小可以达到 2TB。但是日志系统有内存限制。对于每个写事务，SQLite 为每个数据页在内存中维护一个 1bit 的相关信息，无论该数据页是否被访问（读/写）。（页的默认大小是 1024bytes）所以对于拥有很多页的数据库来说，内存限制是一个很重要的瓶颈。

实体的数量和类型

表或者索引最多只能有 2^{64}-1 个。（当然，由于 2TB 的限制，你无法维护这么多。）单个的表或者索引可以包含 2^{30}bytes 的数据。当打开一个数据库文件时，SQLite 从主目录（master catalog）中读取所有的表或者索引并在内存中创建一个目录（catalog）实体。所以，为了提升数据库的性能，最好能降低表、索引、视图、触发器的数量。同样的，尽管列的数量没有限制，但是太多的列也会影响性能。优化查询不会应用于超过 30 列的列。

变量引用

在一些嵌入式数据库中，SQL 语句可以持有变量（例如，来自应用程序的变量。）SQLite 不支持这一功能。但是 SQLite 允许将变量通过 sqlite3_bind_*函数绑定到 SQL 语句上，但是只能绑定输入参数，不能绑定输出参数。这种方法比直接进入数据库要好，因为直接进入数据库需要一个特殊的预处理来把 SQL 语句转化成 API 调用。

存储过程

许多数据库可以创建并存储一种叫做存储过程的东西。存储过程是一组为了完成特定功能的 SQL 语句。SQL 请求可以使用这些过程。SQLite 并不支持这种功能。

第二章 数据库文件格式

在详细介绍 SQLite 引擎之前，我们将先介绍数据库命名规则和数据库文件结构。

2.1 数据库命名规则

应用程序通过向 sqlite3_open 传入数据库文件名来打开数据库。文件名可以是相对路径也可以是绝对路径。SQLite 支持常见的文件系统支持的文件名，但是有两条例外：

- 如果文件名是 c 语言 NULL 指针（例如：0），SQLite 打开一个新的临时文件
- 如果文件名是":memory:"，SQLite 创建一个只在内存中存在的数据库(in-memory database)

在这两种情况下创建的数据库文件是临时的，只要数据库关闭，文件就会消失。

NOTE

SQLite 为临时文件随机命名。文件名的前缀是 sqlite_，后面紧跟着 16 个随机字母或数字。文件存储在本地系统默认的临时文件夹中。SQLite 会依次尝试以下的存储路径：(1)/var/tmp, (2)/usr/tmp, (3)/tmp, (4)当前工作路径。

无论以哪种方式打开数据库（文件方式，临时文件方式，内存数据库方式），在 SQLite 内部，打开的数据库都被命名为 main。

NOTE

在 SQLite 内部，数据库文件的名字不是数据库名。他们是两个不同的概念。通过使用 attach 命令，你可以将一个同样的数据库文件以一个不同的数据库名称连接到一个数据库连接上。你可以通过这些数据库名称对数据库文件进行操作。更多有关 attach 的信息请查询官网。

每当应用程序使用 sqlite3_open 打开一个数据库连接，SQLite 就为这个数据库连接维护一个独立的临时数据库，这个临时数据库被命名为 temp。temp 数据库存储临时实体，例如：表和索引。（应用程序可以在 SQL 语句中使用 main 或者 temp，例如：select * from temp.table1 将返回 temp 数据库的 table1 的所有行，而不是从 main 数据库返回这些信息。Temp 数据库的目录名是 sqlite_temp_master）临时实体只对本数据库连接是可见的（而不是指向同一个文件的，同一个线程、

进程或者其他进程中的数据库连接）。SQLite 将 temp 数据库存储在一个单独的临时文件中，而不是存储在 main 数据库文件中。当应用程序关闭数据库连接的时候，temp 文件将被删除。

2.2 数据库文件结构

除了内存数据库(in-memory)，SQLite 在单独的数据库文件中存储数据库实体(main 或者 temp)。

2.2.1 页

为了方便原地操作和从数据库读写数据，SQLite 把每个数据库(包括内存数据库)分成大小固定的多个页。页的大小是 2 的指数，可以在 512 到 32,768 之间，默认大小是 1024B。（上界是由存储页大小的 2-byte signed int 变量决定的）。数据库（可以扩展和压缩）是一个页组成的数组。页的索引叫做页号（page number）。页号从 1 开始，上界是 2,147,483,674（2^{31}-1）。（上界可以更大，这取决与文件系统的限制）。第 0 页是虚页。第 1 页和之后的页存储在数据库文件中，存储偏移量是 0，即从文件开头开始存储第 1 页。

NOTE

每当一个数据库文件被创建，SQLite 使用默认页大小（在编译时指定的大小），在创建数据库的第 1 个表之前，可以使用预处理命令修改页的大小。

2.2.2 页的种类

页的种类有 4 种：叶子页，内部页，溢出页，自由页。自由页是非激活页（还没有使用的页），其他的是激活页。B+树内部页包含搜索信息（B 树内部页包含搜索信息和数据）。B+树的叶子页存储实际的数据（例如：表中每行的数据）。如果一页不能盛放一行数据，数据的一部分会存储在 B+树的叶子页，另一部分存储在溢出页中。

2.2.3 文件头

SQLite 可以使用任何类型的页，除了页 1，页 1 永远是 B+树的内部页，保存有 100byte 的文件头信息，存储偏移量是 0。文件头信息决定了该数据库文件的结构。当数据库文件被创建时，SQLite 初始化文件头信息。文件头的格式如下表所示，前两列的存储单位是 byte。

Table 2-1. Structure of database file header

Offset	Size	Description
0	16	Header string
16	2	Page size in bytes
18	1	File format write version
19	1	File format read version
20	1	Bytes reserved at the end of each page
21	1	Max embedded payload fraction
22	1	Min embedded payload fraction
23	1	Min leaf payload fraction
24	4	File change counter
28	4	Reserved for future use
32	4	First freelist page
36	4	Number of freelist pages
40	60	15 4-byte meta values

Header string

这是一个 16byte 的 string：”SQLite format 3.”

Page size

页的大小。

File format

偏移量为 18、19 的两个 byte 用来存储文件格式版本，在当前版本中，这两个值都是 1，否则将返回一个 error 值。如果将来文件格式发生改变，这两个值会增长以表示新的文件格式版本。

Reserved space

处于一些原因，SQLite 在每一页底部保留一小块固定空间（<=255bytes），空间的大小保存在 reserveed space 变量中，默认值是 0。使用 SQLite 内建加密模式时，这个值会发生变化。

Embedded payload

最大碎片负载（偏移量是 21）指的是每页能够用于存储 B/B+树内部节点的空间。255 表示 100%。默认值是 64（25%），这个值用来限定 cell 的最大值，即每个节点最少有 4 个 cell。如果 cell 的大小超过允许的最大值，SQLite 就创建溢出页，把尽量多的 byte 移动到溢出页中，但是不能在分割的过程中导致 cell 的大小小于最小 embedded payload(偏移量是 22，默认值是 32，即 12.5%)。

叶子页最小负载（偏移量 23）和最小 embedded payload 相似，但是是针对 B+树的叶子页来说的。（默认值是 32,12.5%）。叶子页的最大负载百分比永远是

100%，文件头中并没有保存该值的变量。

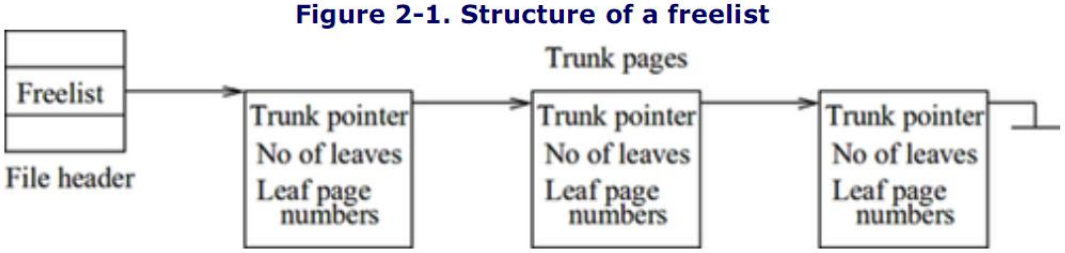
File change counter

文件更改次数（偏移量 24）被用于事务管理。每个事务都会使这个值加 1. 这个值用来指示什么时候数据库被改变，辅助 pager 模块避免盲目刷新 cache。截止到目前为止，这个功能还没有实现。Pager 模块负责增加这个值。

Free list

自由页链表（偏移量 32）存储未使用的页。自由页页数存储在偏移量 36。在自由页链表中页有两种类型：主干页和叶子页。文件头指出了自由页链表中的第一个主干页。每一个主干页里含有指向许多叶子页的指针。

Figure 2-1. Structure of a freelist



主干页的格式如下所示：

- 下一个主干页的页号（4bytes）
- 本页中含有的叶子页指针个数（4bytes）
- 0 或者更多 4bytes 叶子页页号

当一个页变为非激活态，SQLite 将该页添加到自由页链表中，并不会把页释放给本地文件系统。当你向数据库中添加新的数据的时候，SQLite 从自由页链表中取出一页用来存储用户添加的数据。如果自由页链表是空的，SQLite 向本地文件系统申请新的页，把新页附加到数据库文件上。

NOTE

vacuum 命令可以清空自由页链表。该命令在内存中建立一个数据库的副本（这个副本是通过 INSERT INTO…SELECT * FROM…命令实现的）。然后在事务系统的保护下，用内存中的副本重写原始数据库。

Meta variables

在偏移量 40 的地方，有 15 个关于 B+树和虚拟机模块的 4bytes 的整型变量。它们存储许多元信息，包括模式缓存号（偏移量 40），每次模式变化会改变这个值。视图层的文件格式信息（偏移量 44），page 缓存大小（偏移量 48），自动清

空标志（偏移量 52），文本编码（偏移量 56，1:UTF-8, 3:UTF-16 LE, 4:UTF-16 BE），用户版本号（偏移量 60）。在源文件 btree.c 中有关于这些变量更多的信息。

NOTE

SQLite 兼容版本 3.0.0 之前的数据库文件，这意味着任何版本的 SQLite 可以读写 3.0.0 之前创建的数据库文件。3.0.0 的 SQLite 可以读写大部分之后版本创建的数据库文件。无论如何，版本 3.0.0 之后添加了许多新的 SQLite3.0.0 不理解的特性，如果数据库文件包含这些新的特性，3.0.0 及之前的 SQLite 可能无法识别数据库文件。

在 page1 中，B+树内部节点之后是文件头。该节点是主目录表的根，即常规数据库的 sqlite_master，或者临时数据库的 sqlite_temp_master。

NOTE

所有的多字节整型变量都是大端存储的，这允许将数据库文件安全地从一个平台移动到另一个平台。

第三章 页缓存管理（Page Cache Management）

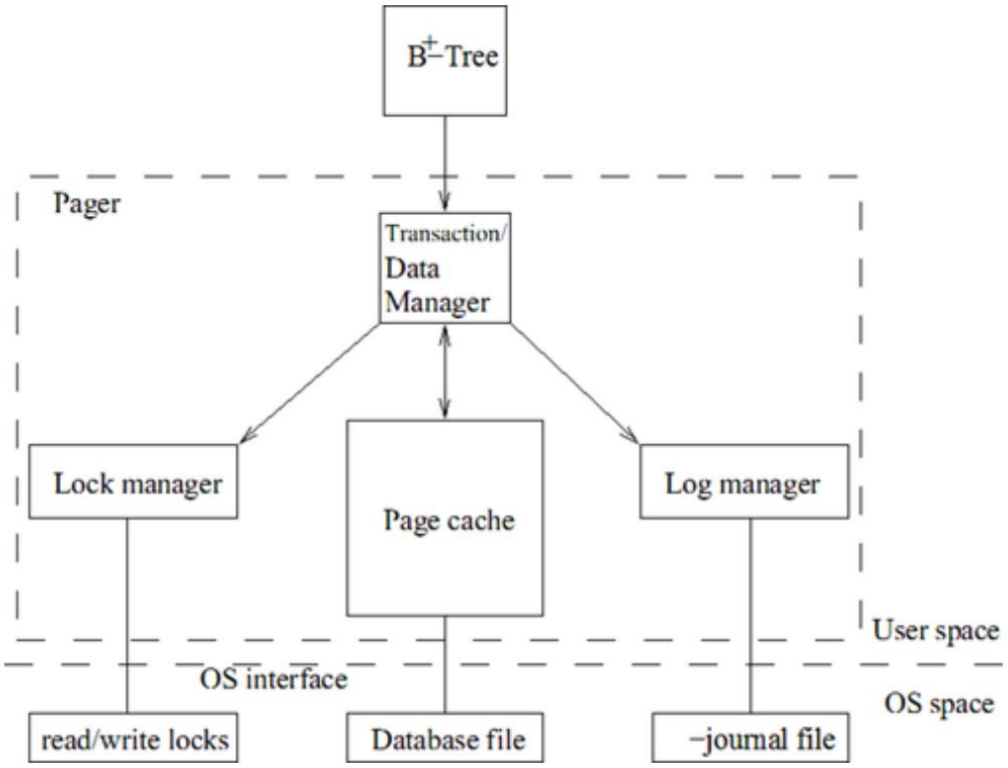
Pager 是进入数据库文件和日志文件的唯一模块（通过操作系统的本地 IO API）。但是它并不翻译数据库的内容，也不独立修改数据。（pager 可能修改一些文件头信息，比如文件修改次数）。Pager 基于随机存取/字节导向的文件系统操作，然后将其抽象为随机存取/页导向的文件系统操作。它定义了易用的、文件系统无关的进入数据库文件页的接口。B+树模块总是使用页接口来进入数据库，而不直接进入任何数据库文件或者日志文件。它将数据库文件视为统一大小的页组成的数组。

数据库（除了内存数据库）通常以常规文件的形式被存储在外存设备上（例如硬盘）。当 SQLite 需要一个数据项的时候，它将该数据项从数据库文件读取到内存中、在内存中对其进行操作、如果需要的话，将其写回文件。一般来说，数据库相比内存要大得多。由于内存资源的珍贵性，只有部分内存用来保存少量的来自数据库文件的数据，这部分内容通常被称为 database cache 或者 data buffer。SQLite 将其称为 page cache。Pager 就是页缓存管理器。

3.1 页缓存管理器的职责（Pager Responsibilities）

对于每个数据库文件，在文件和内存之间移动页是 pager 的基本功能。页的移动对于 B+树模块和更高层的模块来说是透明的，pager 是本地文件系统和其他功能模块的中间层。它的主要职责是保证数据库页的地址是可以查找的到的，从而使其他模块能获取页的内容。它也辅助将页写回数据库文件。Pager 创建了一个抽象，使得整个数据库文件看起来以一个页数组的形式存储在内存中。

除了内存管理，pager 还负责其他许多工作。它提供了事务处理系统的核心业务：事务管理、数据管理、日志管理、锁管理。作为一个事务管理器，pager 通过并发控制和数据库文件恢复保证了事务级别的 ACID 特性，负责实现原子提交和事务回滚。作为数据管理器，通过页缓存辅助读写数据库文件页。作为日志管理器，决定是否将日志记录写入日志文件。作为锁管理器，确保事务在进入数据库页之前获得合适的数据库文件锁。简而言之，pager 实现了存储的持久性和事务的原子性。Pager 子模块之间的连接如下图所示：



NOTE

Pager 之上的所有模块和底层的锁、日志管理是完全独立的。事实上，它们并不关心锁和日志行为。B+树模块通过事务进行操作，并不关心事务的 ACID 特性是如何实现的。Pager 模块把事务行为分为锁、日志、读写数据库文件。B+树模块通过传递页号向 pager 申请一页。Pager 返回一个指针，该指针指向页缓存中的一页。在修改一页之前，B+树模块通知 pager，pager 将相关数据写入日志文件，以备将来修复数据库用。然后 pager 请求获得合适的文件锁。B+树使用完页后会通知 pager，如果该页被修改过了，pager 负责将该页写回数据库文件。

3.2 pager 接口结构

Pager 模块实现了名为 pager 的数据结构。每个打开的数据库文件通过一个单独的 pager 实体管理，每个 pager 实体也有且只有一个相关联的数据库文件。如果 B+树模块要使用一个数据库文件，它就需要创建一个新的 pager 实体，通过该实体实现 pager 级别的数据库文件操作。Pager 可以跟踪到文件锁信息、日志文件信息、数据库状态信息、日志状态信息等。

3.3 缓存管理

SQLite 为每个打开的数据库文件维护了一个 page 缓存。如果一个线程多次打

开同一个数据库文件，只有在第一次打开文件的时候 pager 创建并初始化一个独立的页缓存。如果多个线程打开同一个文件，就会产生同一个文件的多个独立的页缓存。内存数据库也以文件的形式存储在内存中。所以对于任一种类型的数据库，B/B+树模块通过同样的接口访问数据库。

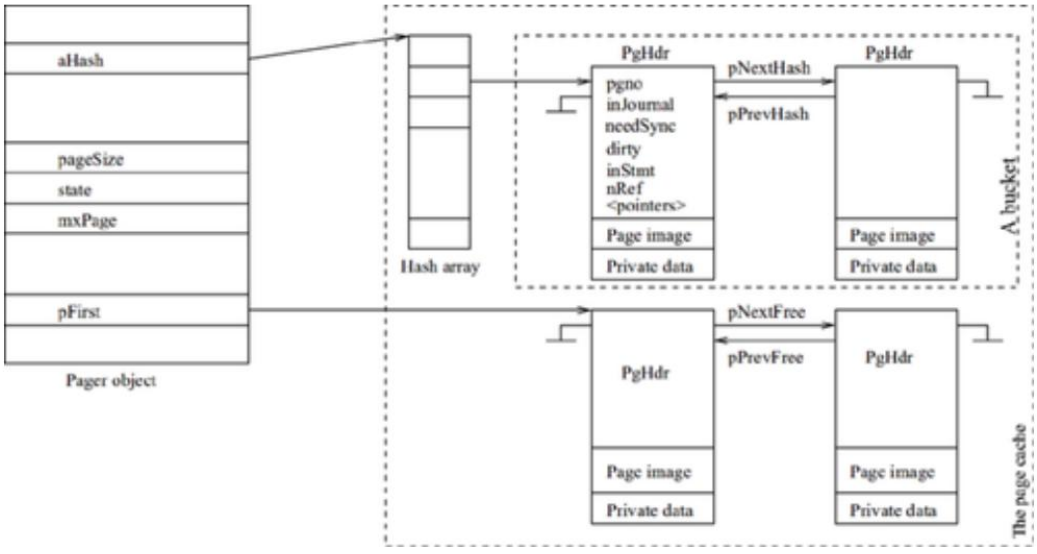
NOTE

页缓存存在于应用程序的内存空间中。操作系统本身也会创建一份页缓存。当应用程序从文件读取数据时，操作系统通常创建一份数据备份，然后应用程序创建自己的页缓存。SQLite 页缓存的组织和管理和操作系统中的管理是独立的。

页缓存的管理对系统性能的影响非常大。接下来的小节将讨论 pager 如何组织和维护页缓存，以及缓存的使用者如何读取和修改缓存。

3.3.1 内存组织

为了加快内存搜索速度，内存能很好地组织当前持有的数据项。SQLite 使用一个哈希表和页插槽来组织缓存页。插槽是通用的，即任意类型的插槽可以是通用的，即任意类型的插槽可以存放任意类型的页。随着页的增长，pager 创建新的插槽并将它们添加到哈希表中。一个缓存可以含有最多的插槽的个数是限定的。（内存数据库没有这个限制，只要操作系统允许应用程序的内存空间增加）



上图描述了缓存层。哈希表中的每一页由一个 PgHdr 类型的实体表示。页的数据存储在 PgHdr 实体后面。页数据后面是一些私有信息，用来供 B+树控制页缓存。（内存数据库没有日志文件，所以它们的恢复信息存储在内存实体中。指

向这些实体的指针存储在上面提到的私有信息之后，只有 pager 能使用这些指针。）当 pager 把页拉到缓存中时，上述私有信息占用的空间被初始化为 0。所有的 page 可以通过哈希表来访问，即 pager 中的 aHash。哈希表的大小在 SQLite 库被编译后就是固定的了。哈希表中的每个元素指向一个 page 组成的桶，每个桶中的页通过一个双向链表组织起来。

PgHdr 只对 pager 模块是可见的，对 B+树模块和其他高层模块是不可见的。PgHdr 的头部有很多控制变量。Pgno 是页号。当一页已经被写入回滚日志时，injournal 为真。如果在 page 写回数据库文件之前，日志需要被刷新到磁盘，那么 needSync 为真。如果页被修改，而且新值还没有被写回数据库文件，那么 dirty 为真。如果当前页在当前 statement 日志中，inStmt 为真。nRef 表示引用当前页的数量。如果 nRef 大于 0，该页就是激活的，我们称该页被钉住；否则，该页就是未被钉住的，是自由的。PgHdr 实体中有很多指针变量（上图并没有显示所有的指针变量）。pNextHash 和 pPrevHash 将同一个桶中的页连接起来。pNextStmt 和 pPrevStmt 把在 statement 日志中的页连接起来（接下来我们将介绍 statement 日志）。pNextFree 和 pPrevFree 指针用来连接所有的自由页。缓存中所有的页（自由页或非自由页）通过 pNextAll 指针连接起来。pDirty 指针连接所有的脏页。注意，自由页也可能是脏页。

3.3.2 读取缓存

缓存通过页号来搜索页。B+树模块向 sqlite3pager_get 函数传入页号来读取页。该 API 读取页的时候按照以下的步骤进行：

1.

搜索缓存空间。

a.

使用哈希函数对页 P 进行处理，得到一个索引值。（SQLite 使用很简单的哈希函数来确定索引值：页号对哈希数组的大小取模）

b.

使用索引在哈希表中查找，得到哈希值对应的桶。

c.

通过 pNextHash 指针搜索桶。如果找到 P，页 P 被钉住（nRef 增加 1），

返回页的地址。

2.

如果没有找到页 P，该函数寻找一个自由插槽来插入期望页。（如果插槽没有到达最大数量，创建一个新的插槽）

3.

如果没有自由插槽也不能创建新的插槽，就按照内存置换算法找到一页，释放掉该页，重新使用这个插槽。这种插槽称为受害者插槽。

4.

如果受害者页或者自由页是脏页，将该页写回数据库文件（遵循 WAL 原则，即在写回该页之前确保该页的日志信息已经刷新到磁盘）。

5.

从数据库文件读取页 P 到自由插槽，钉住该页（例如，将 nRef 赋值为 1），返回页的地址。如果 P 大于当前文件的最大页数，不读取该页，转而初始化该页为 0。当从文件读取页时，也初始化 pgHdr 底部的私有变量为 0。

SQLite 严格遵循按需获取原则，使得页缓存的获取逻辑非常简单。

3.3.3 写缓存

当某个页的地址被返回给 B+树模块时，pager 不知道使用者什么时候操作页。每个页遵循标准协议：使用者申请页，使用页，释放页。申请到页之后，使用者可以直接修改页的内容，但是在做任何修改之前必须调用 sqlite3-pager_write。该函数返回后，使用者可以原地修改页。

当 write 函数第一次被调用时，pager 把页的原始信息写入回滚日志中，并把 injournal 和 needSync 这两个标志位置为真。（SQLite 遵循 WAL 原则：直到 needSync 标志位清零后，才将被修改过的页写回数据库文件）每当 write 函数操作一页时，dirty 标志位被置为真，只有当 pager 把页写回数据库文件后，dirty 标志位才被清零。因为 pager 不知道什么时候使用者会修改页，所以页并不会马上被刷新到数据库文件。Pager 遵循延迟写回原则：页的刷新被延迟，直到 pager 进行强制刷新操作或者该页需要被回收以空出插槽供新的页使用（即该页是受害者页的情况）。

3.3.4 页置换

当缓存已经满了的时候就会发生页置换，旧的页被去掉，为新页的加入腾出空间。就如在读取页小节中提到的，当申请的页不在缓存中而且没有可以取得的自由插槽时，pager 选择一页作为受害者页，将其置换。之前我们也介绍过，所有的插槽都是通用的，所以新页可以存放在任意一个插槽中。SQLite 使用近期最少使用算法（LRU）进行页的置换。

SQLite 将自由页通过优先队列组织起来。当某个页是未被钉住（自由）的状态时，pager 将该页添加到队列的末尾。一般选择队列开头的那页作为受害者页，但是纯粹的 LRU 算法并不总是选择队首的页作为受害者页。SQLite 尝试从队首开始寻找这样一页：置换该页时不需要刷新该页到磁盘。（因为遵循 WAL 原则，刷新页到数据库文件之前要把页的内容刷新到日志文件中，刷新到磁盘是一个费时的操作，SQLite 希望尽可能的推迟这个操作。）如果找到符合这个要求的受害者页，就将该页所在的插槽回收。否则，SQLite 先刷新日志文件到磁盘，再刷新队首页到磁盘，最后回收队首插槽。

NOTE

页被钉住表示该页目前正在被使用，不能被回收。为了避免出现缓存中所有的页都被钉住的情况，SQLite 需要保留一些插槽，对于 SQLite3.3.6 发行版，最少保留插槽个数是 10 个。

第四章 事务管理

事务管理是数据库能够并发的关键。SQLite 依赖本地文件锁和页日志来保证

ACID 性质。SQLite 只支持平面事务，没有鸟巢事务结构（nestting）或者保存点（savepoint）能力。

4.1 事务类型

SQLite 在事务中执行每条 SQL 语句。支持读事务和写事务。应用程序不能通过读/写事务之外的方式读/写数据库。对于一个 SELECT 语句，SQLite 创建一个读事务，然后将其转换为一个写事务。SQL 语句执行完成后，事务自动提交。应用程序不知道事务具体的执行过程，它们只需要把 SQL 语句提交给 SQLite，由 SQLite 负责剩下的工作。一个程序可以在同一个数据库连接上并行地执行 SELECT 语句（读事务），但是在同一个数据库连接上只能执行一个非 SELECT 语句（写事务）。

对于一些应用程序，尤其是写入操作很频繁的程序来说，自动提交事务的成本非常高，因为对于每条非 SELECT 语句，SQLite 需要执行：打开日志文件，写入，关闭日志文件三个操作。自动提交事务模式下，每次语句执行完毕后（包括 SELECT 语句），SQLite 丢弃 page 缓存。缓存的重建导致磁盘的读取/写入操作，是一个很费资源的操作。另外，并发控制还需要频繁地获取和释放文件锁。这些问题会导致 SQLite 性能下降。解决方法是建立用户级别的事务（对应的，上面提到的是系统级别的事务）：

```
BEGIN;
        INSERT INTO tablel values(100);
        INSERT INTO table2 values(20, 100);
        UPDATE tablel SET x=x+1 WHERE y> 10;
        INSERT INTO table3 VALUES (1,2,3);
COMMIT;
```

应用程序可以通过显式使用 BEGIN 命令手动创建一个事务。该事务是用户级别的事务。用户事务创建完成后，SQLite 放弃自动提交模式，即每个 SQLite 语句结束后不自动执行提交（commit）或放弃（abort），也不会丢弃 page 缓存。接下来的 SQL 语句变成用户事务的一部分。当应用程序执行 commit（或者 rollback）命令的时候，SQLite commit（或者 abort）事务。如果事务被 abort 或者执行失败，或者应用程序关闭了数据库连接，整个事务就会回滚。用户事务结束后，SQLite 返回自动提交模式。

NOTE

SQLite 只支持平面事务。应用程序不能对于同一个数据库连接在同一时刻创建多个用户事务。如果在用户事务中执行 BEGIN 命令，SQLite 会返回错误。

用户进程又不仅仅是平面事务。每个非 SELECT 语句都是被一个单独的语句级别的子事务执行的。尽管并没有存储点功能，SQLite 还是在当前语句级别的子事务中实现了存储点。如果当前事务执行失败，SQLite 不会放弃（abort）用户事务，而是把数据库状态恢复到该事务执行之前的状态，接下来的事务从这里开始执行。执行失败的事务不会影响其他子事务执行的结果，SQLite 帮助一个很长的用户事务实现了一定的容错能力。

在上面的例子中，每个 SQL 语句都是一个独立的子事务。如果 UPDATE 的第 10 列发生了严重的错误，UPDATE 对前 9 列的修改都会回滚。但是三个 INSERT 子事务会被提交。

4.2 锁管理

SQLite 为了保证事务的串行执行，使用文件锁策略来保证事务进入数据库的请求。SQLite 严格遵守两相锁协议，即只有事务完成才释放锁。SQLite 实行数据库级别的锁机制，而不是列、页或者表级别的锁机制，它锁住整个数据库文件，而不是数据库文件的一部分。

NOTE

子事务通过所属的用户事务申请锁。直到用户事务的子事务都提交或者放弃，锁才被释放。

4.2.1 锁的种类和兼容性

对于单个事务，数据库文件可以是以下五种锁状态中的一种：

NOLOCK

事务不会持有数据库文件的锁。该事务可以读/写数据库。只要其他事务的锁状态允许，其他事务也可以读/写数据库文件。事务初始化之后的默认状态就是无锁状态。

SHARED

SHARED LOCK 只允许读数据库文件。任意数量的事务在同一时间可以获取同一个数据库文件的 SHARED LOCK，所以可以有许多并发的读事务。当一个或者多个事务持有 SHARED LOCK 时，不允许事务写数据库文件。

RESERVED LOCK

RESERVED LOCK 允许从数据库文件读取数据。RESERVED LOCK 意味着事务有写数据库文件的意图，但是该事务现在只是在读数据库文件。一个数据库文件最多只能有一个 RESERVED LOCK，但是 RESERVED LOCK 可以和许多 SHARED LOCK 并存。RESERVED LOCK 存在时，其他事务可以获得新的 SHARED LOCK 锁，但是不能获得新的其他类型的锁了。

PENDING

PENDING LOCK 允许读数据库文件。PENDING LOCK 意味着事务想要尽快写数据库文件。该事务只是在等待所有的读取锁释放，从而可以获得一个 EXCLUSIVE LOCK。一个数据库文件最多只能有一个 PENDING LOCK，PENDING LOCK 可以和已经存在的 SHARED LOCK 并存，但是在这期间，其他事务不能获得新的锁。

EXCLUSIVE

EXCLUSIVE LOCK 是唯一允许写操作的锁。一个文件只允许一个 EXCLUSIVE LOCK 锁存在，其他任何种类的锁都不能和 EXCLUSIVE LOCK 并存。

锁的兼容性表格如下表所示。列是一个事务已经拥有的锁的类型，行是另一个新请求的锁的类型，每个单元格表示两个锁的兼容性。

Table 4-1. Lock compatibility matrix

Requested lock → / Existing lock ↓	SHARED	RESERVED	PENDING	EXCLUSIVE
SHARED	Y	Y	Y	N
RESERVED	Y	N	N	N
PENDING	N	N	N	N
EXCLUSIVE	N	N	N	N

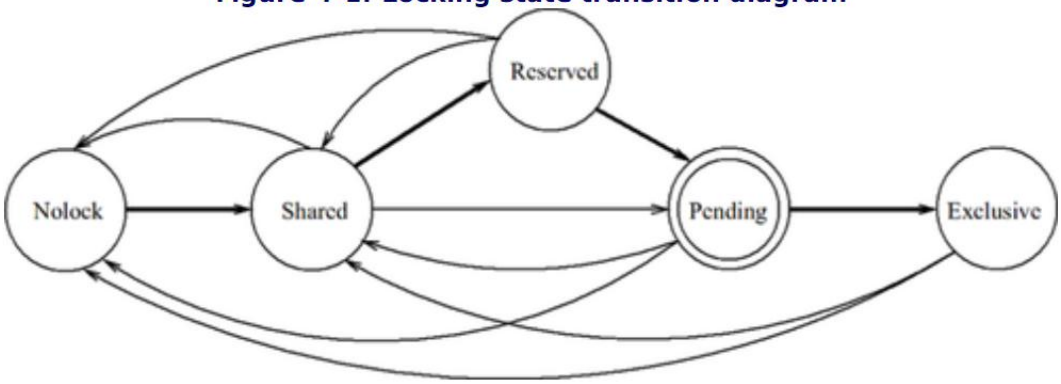
NOTE

对于 SQLite 这种数据库来说，EXCLUSIVE LOCK 是必须的，其他类型的锁只是为了增加事务的并发性。如果只有 EXCLUSIVE LOCK，SQLite 可以串行地执行地事务。如果只有 SHARED LOCK 和 EXCLUSIVE LOCK，SQLite 可以执行读事务的并发。在实际中，一个事务在 SHARED LOCK 的保护下读数据库文件，修改数据项，申请 EXCLUSIVE LOCK 来把数据项写回数据库文件。如果两个事务同时进行上述的操作，就会有可能发生死锁。RESERVED LOCK 和 PENDING LOCK 就是用来大幅降低这种死锁的可能性的。这两种锁也提高了事务的并发性，改善了写饥饿问题（读操作一直排挤写操作）。

4.2.2 锁的申请原则

在读取数据库的页之前，事务申请一个该页的 SHARED LOCK，表示该事务想要读取该页。在对页进行任何修改之前，事务需要申请一个 RESERVED LOCK，表示该事务在不久的将来要对该页进行写操作。当持有一个 RESERVED LOCK 的时候，事务可以修改内存中的页。在将页写会数据库文件之前，事务需要获得一个 EXCLUSIVE LOCK。锁的状态转换图如下所示。

Figure 4-1. Locking state transition diagram



一般的锁转换步骤是：SHARED LOCK--RESERVED LOCK--PENDING LOCK--EXCLUSIVE LOCK。直接从 SHARED LOCK 转化为 PENDING LOCK 只会在某个日志需要会滚的时候才会发生。但是一旦发生这种情况，其他事务都不能从 SHARED LOCK 转化为 RESERVED LOCK。

NOTE

PENDING LOCK 是一个内部锁，在锁管理系统之外是不可见的。pager 不能通过锁管理器获取一个 PENDING LOCK。pager 可以申请一个 EXCLUSIVE LOCK 锁，但是在锁管理器内部总是现获取一个 PENDING LOCK，然后转化为一个 EXCLUSIVE LOCK。所以 PENDING LOCK 是获取 EXCLUSIVE LOCK 之前的一个暂时的步骤。获得 PENDING LOCK 之后，为了防止申请 EXCLUSIVE LOCK 失败，pager 随后会发送一个 EXCLUSIVE LOCK 的请求，该请求将 PENDING LOCK 升级为 EXCLUSIVE LOCK。

NOTE

尽管锁机制解决了并发控制问题，但是又导致了新的问题。假设两个事务持有一个数据库文件的 SHARED LOCK。它们都申请 RESERVED LOCK。其中一个得到了 RESERVED LOCK，另一个进入等待状态。一会儿之后，拥有 RESERVED LOCK 的事务申请一个 EXCLUSIVE LOCK，这样的话就需要处于等待状态的事务释放 SHARED LOCK。但是 SHARED LOCK 永远不会被释放，因为拥有 SHARED LOCK 的事

务处于等待状态。这种情况称之为死锁。

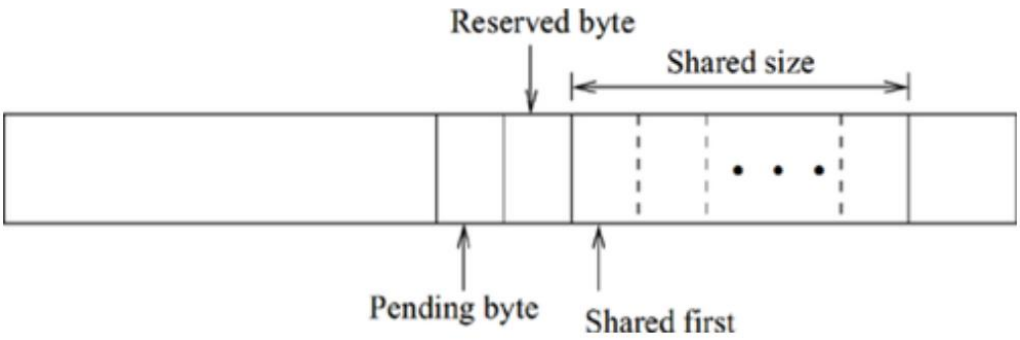
死锁是一个恼人的问题。解决死锁的方法有两种：1、预防；2、检测并去除。SQLite 采取预防的方法。如果事务获取锁失败，该事务将进行有限次的尝试（尝试次数可以在主程序运行时设定，默认次数是 1）。如果所有的尝试都失败了，SQLite 返回 SQLITE_BUSY。主程序可以在稍候重试或者放弃事务。最终，系统中将不会出现死锁。无论如何，理论上，系统中会发生饥饿现象，即一个事务一直申请一个锁但是永远也获得不了这个锁。但是 SQLite 并不是一个企业级别的数据库，所以饥饿现象不是一个大问题。

4.2.3 锁的实现

SQLite 使用操作系统提供的文件锁函数实现锁机制（文件锁的实现是依赖于操作系统的。本书使用 Linux 系统演示文件锁是如何实现的）。Linux 只有两种锁模式，即读取锁和写入锁，用来锁定文件的某一段连续的部分。为了避免术语混淆，接下来用读取锁代替 SHARED LOCK，用写入锁代替 EXCLUSIVE LOCK。Linux 为进程和线程分配锁，为了避免混淆进程和线程，下文将一直使用线程。许多父辈线程或者同辈线程可能都持有文件某一段读读取锁，但是只有一个线程可能取得文件该段的写入锁。写入锁排斥其他所有的锁，包括读取锁和写入锁。在同一个文件中读取锁和写入锁可以并存，但是两种锁必须在文件的不同段。单独的线程在文件的某一段只能持有一种类型的锁。如果该线程要向一个已经有锁的段申请锁，那么这个锁将会变成新的锁。

- SQLite 使用操作系统提供的文件锁实现自己的四种锁模式：
- SHARED LOCK 是通过在数据库文件的某一段设置读取锁实现的。
- EXCLUSIVE LOCK 是通过在数据库文件的某一段设置写入锁实现的。
- RESERVED LOCK 是通过在数据库文件的某一个 byte 设置写入锁实现的（该 byte 在 SHARED LOCK 之外的范围），该 byte 被设计为 RESERVED LOCK byte。

PENDING LOCK 是通过在数据库文件的某一个 byte 设置写入锁实现的（这个 byte 与 RESERVED LOCK 所在的 byte 不同，而且也在 SHARED LOCK 范围之外）。下图表示了锁的分配。



SQLite 保留 510bytes（这个数值在源代码的头文件中被 SHARED_SIZE 宏定义）作为 RESERVED LOCK byte。这个范围从 SHARED_FIRST 开始。PENDING_BYTE 宏（0x40000000,第一个 byte 定义了锁 byte 的起始位置）被用来设定 PENDING LOCK。RESERVED_BYTE 宏用来设置 PENDING_BYTE 之后的 byte。SHARED_FIRST 用来设置 PENDING_BYTE 之后的第二个 byte。所有的 bytes 都装在一页中，尽管这一页是最小的一页（512bytes）。

NOTE

windows 中的锁是强制性的，这意味着锁的有效范围是所有的进程，即使进程之间并不相关。被锁住的空间是被操作系统持有的。因此，SQLite 实际上并不能在被锁住的空间中存储数据。所以，pager 从不分配锁 page（上一段提到的那个 page）。

为了获取一个数据库文件的 SHARED LOCK，一个线程首先获取一个位于 PENDING_BYTE 上的读取锁来保证没有其他任何进程或者线程持有这个文件的 PENDING LOCK。（上文提到过已经存在的 PENDING LOCK 和新创建的 SHARED LOCK 是不兼容的）。如果操作成功，SHARED_SIZE 所确定的范围就被读取锁锁住了，最后，PENDING_BYTE 上的读取锁释放。

NOTE

windows 的一些版本只支持写入锁。为了获取一个 SHARED LOCK，特定 bytes 范围之内的一个 byte 被设计为写入锁。这个 byte 是随机选取的，所以有可能两个独立的读取者同时进入同一个数据库文件，除非它们很不巧地选中了同一个 byte 作为写入锁。在这种系统中，并发性受 SHARED_SIZE 大小的影响。

一个线程可能在获得一个 SHARED LOCK 之后又获得一个 RESERVED LOCCK。为了获取 SHARED LOCK，一个写入锁被获取（在 RESERVED LOCK bytes 范围内）。

注意到线程不会释放文件的 RESERVED LOCK。（这保证了另一个线程不能获得这个文件的 EXCLUSIVE LOCK）

一个线程可能在获得 SHARED LOCK 之后仅仅想要获得一个 PENDING LOCK。为了获得 PENDING LOCK，需要在 PENDING_BYTE 中获取一个写入锁。（这保证了没有新的 SHARED LOCK 可以被获取，但是已经存在的 SHARED LOCK 可以并存）注意线程并不释放自己的 SHARED LOCK。（这保证了另一个线程不能获得 EXCLUSIVE LOCK）

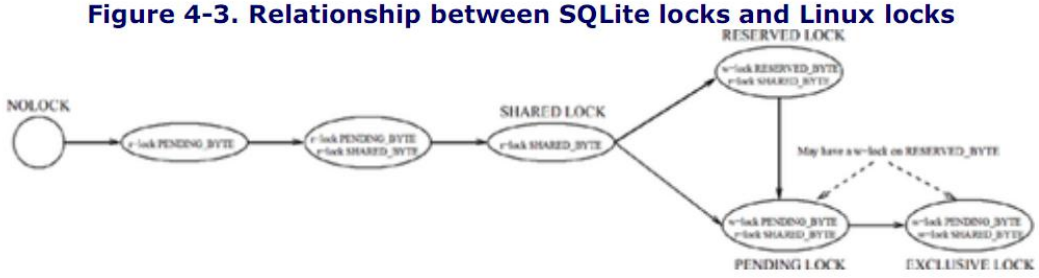
NOTE

一个线程在获取 PENDING LOCK 之前并不是必须获取一个 RESERVED LOCK。当系统崩溃时，这条性质被 SQLite 用来回滚日志文件。如果线程通过 SHARED--RESERVED--PENDING 这样的顺序获取 PENDING LOCK，它为了获取 PENDING LOCK 将不会释放前两个锁。

一个线程在获取 PENDING LOCK 之后可能只获取了一个 EXCLUSIVE LOCK.为了获取 EXCLUSIVE LOCK，首先要在 shared byte range 上获取一个写入锁。因为所有的 SQLite 锁都需要在 shared byte range 上获取读取锁（至少一个），这保证了当线程持有 EXCLUSIVE LOCK 的时候，没有其他的锁。

锁的状态转换如下图所示。下图展示了 SQLite 锁和本地锁的关系。PENDING LOCK 和 EXCLUSIVE LOCK 的表示有些笨拙，它们在 RESERVED_BYTE 上有没有写入锁取决于 SQLite 是通过那条路径获取锁的。

Figure 4-3. Relationship between SQLite locks and Linux locks



4.3 日志管理

当事务或者子事务被撤销的时候，或者应用程序、系统崩溃的时候，可以通过日志存储的信息恢复数据库。SQLite 为每个数据库维护一个日志文件。（SQLite 不会为内存数据库维护日志文件。）SQLite 假设恢复只负责事务的回滚，所以日志文件又叫做回滚日志。日志文件总是和数据库文件放在同一个路径下，并且有

同样的名字，但是后缀是 journal。

NOTE

SQLite 同一时刻只允许一个写事务访问数据库文件，并且为每个写事务动态地创建日志文件，当事务完成后这个日志文件会被删除。

4.3.1 日志文件结构

Figure 4-4. Structure of journal segment header

SQLite 把回滚日志分成许多大小不同的段。每个段有一个段头，之后是日志记录。段头的格式如上图所示。首先是一个 8bytes 的 magic number：0xD9, 0xD5, 0x05, 0xF9, 0x20, 0xA1, 0x63, and 0xD7。magic number 被用来进行日志文件完整性校验。日志条目的数量（nRec）表示在这个段中有多少个有效日志条目。Random number 用来确定每个日志条目的校验值。不同的段可能有不同的 Random number。initial database page count 表示当前日志创建时，数据库文件中一共有多少页。sector size 表示日志文件所在磁盘的扇道大小。段头占一个完整的扇道。段头中未使用的空间被保留。

NOTE

SQLite 支持异步事务模式，这种模式比普通的事务模式快。SQLite 不推荐使用异步模式，但是你可以通过预处理指令来开启这个功能。异步模式通常用在软件的开发阶段来缩短开发时间。该模式也适用于一些不需要测试数据库恢复的测试程序。异步模式既不会刷新日志到磁盘，也不会刷新数据库文件到磁盘。日志文件将只有一个日志段。nRec 的值是-1，（例如：0xFFFFFFFF 作为一个有符号数），实际值与文件的大小有关。

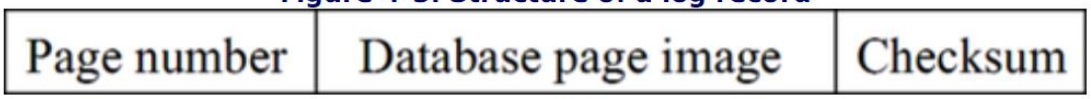
通常来说，回滚日志只有一个日志段。但是在一些情况下，日志文件会有很多日志段，SQLite 多次写入段头（在接下来的缓存刷新小节里你会看到这种情况）。每次写入段头都是从扇区边界开始的。在多日志段文件中，每个段的 nRec 都不能是 0xFFFFFFFF。

4.3.2 日志记录结构

当前写事务中的非 SELECT 命令创建日志记录。SQLite 使用旧值记录技术实现页级别的粗粒度日志控制。在第一次替换任何页之前，页的原始信息作为一个新

的日志条目被写入日志文件。该条记录包含一个 32-bit 的校验值。校验值包含了页号信息和页中的数据的信息。段头的 32-bit random number 是生成校验值的依据。random number 非常重要，因为文件尾部的垃圾数据可能是其他文件删除的数据。（垃圾数据会被识别出来，因为两个数据库文件的 random number 不一样，所以日志记录生成的校验值也就不一样）。如果垃圾数据来自一个已经被删除的日志文件，那么垃圾数据的校验值有很小的可能是正确的。SQLite 通过随机初始化为与其他数据库文件不同的校验值，最小化了这种可能。

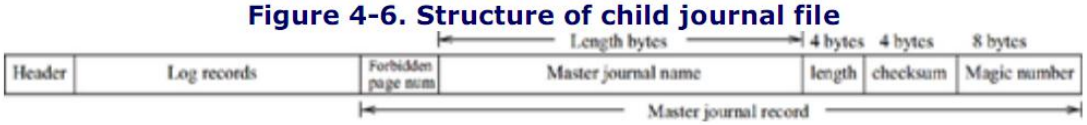
Figure 4-5. Structure of a log record



4.3.3 多数据事务日志

应用程序可以使用 ATTACH 命令把数据库附加到一个已经打开的数据库连接上。如果一个事务修改多个数据库，每个数据库都会有自己的回滚日志，并且彼此之间并不知道对方的存在。为了打破这种隔阂，SQLite 额外维护一个集合日志，叫做 master journal。master journal 不包含用于回滚的任何日志记录。它记录同一个事务下的所有回滚日志文件的名字。相对的，每个回滚日志文件也记录了 master journal 的名字。如果没有附加数据库，或者附加数据库没有加入当前事务（写事务），就不会有 master journal 被创建，并且普通的回滚日志文件不会记录 master journal 的名字。

Figure 4-6. Structure of child journal file



master journal 总是和主数据库文件在同一个路径下，并且和主数据库文件有同样的名字，但是带有-mj 后缀，后面还会跟着 8 个随机的字母或数字。master journal 总是暂时性的文件。当事务想要提交的时候，它被创建，提交完成后，它被删除。

4.3.4 Statement journal

在用户事务中，SQLite 为最近的非 SELECT 命令维护一个 Statement subjournal。当出现 Statement fail 后，这个日志可以用来修复数据库。Statement journal 是一个独立的回滚日志文件，文件名字是随机确定的（前缀是 sqlite_）。该文件不是

用于修复系统崩溃引起的数据库错误的，当出现 statement 撤销的时候，这个文件才会派上用场。当 statement 完成后，SQLite 删除这个文件。该文件没有段头记录，nRec 的值被保存在一个在内存中的数据结构里，这个数据结构还保存了 statement 执行前数据库文件的大小。这些日志记录没有校验信息。

4.3.5 日志协议

SQLite 遵循 WAL 协议来保证数据库可恢复性。SQLite 不会立刻强制把日志记录刷新到磁盘，在把下一页写回数据库文件的时候，SQLite 才会刷新所有的日志。日志刷新保证了被写入日志的所有页都到达了磁盘。如果在日志刷新之前，数据库被修改了，然后发生了断电，未刷新的日志就会丢失，SQLite 就不能回滚事务造成的影响，这会导致数据库崩溃。

4.3.6 提交协议

当应用程序提交事务时，SQLite 确保所有的回滚日志都刷新到了磁盘。提交完成后，回滚日志被删除，事务执行完毕。如果在这之前发生了系统崩溃，事务提交失败，下次数据库被读取的时候数据库就会回滚。无论如何，在删除日志文件之前，所有的数据库修改都被写到了磁盘上。

NOTE

SQLite 不会为异步事务提供回滚日志。所以，如果在异步模式下出现错误，数据库有可能崩溃。书写异步事务的人需要注意这一点。

4.4 事务的执行

和其他的数据库一样，SQLite 的事务管理有两个组件：1）一般过程 2）回滚过程。在一般过程中，pager 把回滚信息存到日志文件，如果需要的话，使用回滚信息修复数据库。

一般过程包括以页的形式读/写数据库文件，并提交事务和子事务。另外，作为一般过程的一部分，pager 把页缓存刷新到磁盘。

大多数事务和 statement 子事务提交自己。但是在一些情况下，有的事务或者子事务撤销自己。有时可能发生断电或者系统崩溃。无论这两种情况哪一种发生了，SQLite 都需要对数据库进行回滚。如果只是撤销事务，内存中的信息就能帮助回滚，如果断电或者系统崩溃，数据库也可能崩溃，而且无法获取内存中的信息。

4.4.1 读操作

为了对数据库的页进行操作，B+树模块需要使用 sqlite3pager_get 函数。如果请求的页在数据库文件中不存在，pager 会创建这个页。sqlite3pager_get 函数首先取得一个 SHARED LOCK，如果获取锁失败，就返回 SQLITE_BUSY。否则，该函数执行一个内存读取操作，并把读取的页返回给调用者。读取内存的操作会把页钉住。

Pager 第一次申请 SHARED LOCK 的时候，意味着 pager 开始了一个隐性的读事务，所以它会判读是否需要对数据进行恢复（回滚）。如果数据库需要回滚，pager 在返回读取页之前执行回滚操作。

4.4.2 写操作

修改一页之前，B+树模块必须把该页钉住（通过使用 sqlite3pager_get 函数）。通过使用 sqlite3pager_write，页会变成可写页。第一次 sqlite3pager_write 被用于页的时候，pager 需要申请一个 RESERVED LOCK。如果申请失败，pager 返回 SQLITE_BUSY。

Pager 第一次申请 RESERVED LOCK 的时候，我们称它将读事务逐步升级为了写事务。同时，pager 创建并打开回滚日志、初始化日志第一段的段头，记录下数据库文件的当前大小。

为了使一页变得可写，pager 把该页的原始内容写入到回滚文件中（以一个新的日志条目的形式）。一但页变得可写，用户可以任意次地修改该页，而不用通知 pager，对该页的改动不会立即写回到数据库文件。

NOTE

一但一页的内容被复制到回滚日志中，该页就永远不会成为回滚日志文件的新的日志条目，即使当前事务多次对该页调用了写函数。这样做的好处是该页可以从日志文件中复制出来实现页的恢复。无论事务对该页做了一次修改还是多次修改，回滚后，该页的内容都是一样的，所以回滚不会产生任何的弥补日志记录。

SQLite 从来不会把新创建的页（当前事务添加的数据库中的页）存储在日志文件中。因为数据库中没有该页的更旧的版本。日志文件转而记录数据库文件的当前大小，这样当事务回滚的时候，直接将数据库文件裁剪到原大小就可以了。

4.4.3 缓存刷新

缓存刷新是 pager 模块的内部操作，其他模块不能直接执行这个操作。如果内存满了需要进行页置换，或者事务准备好提交更新，pager 就把页写回数据库文件。Pager 按照下面的顺序执行：

1.

如果事务不是异步的且新的数据已经写入到日志中，并且数据库不是一个临时数据库，那么 pager 会把内存中的日志信息刷新到日志文件中。Pager 会先执行一个 fsync 调用，确保这之前的所有日志条目都已经被刷新到了日志文件（磁盘）。日志文件刷新完成后，pager 修改当前日志段头的 nRec 值，然后再进行一次刷新。上述操作不是原子性的，如果在第二次刷新之前系统崩溃，数据库可能面临一些风险。

2.

尝试获取 EXCLUSIVE LOCK，如果获取失败，返回 SQLITE_BUSY。

3.

把所有修改过的页写回数据库文件，这一过程是原地进行的，内存中的被修改的页会被清空。

如果是因为内存置换把页写回数据库文件，pager 不会提交事务，而是继续执行事务。在所有的修改完成之前，上述的 3 步重复执行。

NOTE

Pager 获取的 EXCLUSIVE LOCK 直到事务完成后才被释放。这意味着其他事务不能在此期间创建读/写事务。

4.4.4 提交操作

一个事务修改一个数据库或者修改多个数据库，这两种情况下 SQLite 的提交原则有一点不同。

4.4.4.1 单数据库的情况

提交读事务非常简单。Pager 释放 SHARED LOCK，清除页缓存。提交写事务的话，pager 执行以下的步骤：

1.

获取 EXCLUSIVE LOCK。（如果获取失败，返回 SQLITE_BUSY。）把所有修改过的页写回数据库文件。

2.

Pager 执行一个 fsync 系统调用，把数据库文件刷新到磁盘上。

3.

删除日志文件。

4.

释放 EXCLUSIVE LOCK，清空内存。

4.4.4.2 多数据库的情况

相比之下提交协议有些复杂，多数据库系统中的事务提交是类似的。虚拟机模块实际操作提交过程。每个 pager 执行自己负负责的事务提交。对于读/写事务来说，该原则保证了对所有包含进来的数据库都进行提交，具体步骤如下：

1.

释放相关所有数据库的 SHARED LOCK。

2.

申请相关所有数据库的 EXCLUSIVE LOCK。

3.

创建一个新的 master journal 文件，在该文件中写入所有的回滚文件，将 master journal 和所有的普通日志文件刷新到磁盘。（master journal 中不包含临时数据库的名字）

4.

把 master journal 文件的名字写入所有的普通日志文件中，刷新回滚日志到磁盘。（直到事务提交的时候，pager 才会知道这是一个涉及多个数据库的事务。）

5.

刷新每个数据库文件。

6.

删除 master journal 文件，刷新日志目录。

7.

释放所有数据库的 EXCLUSIVE LOCK，清空所有页缓存。

NOTE

master journal 被删除的时候系统就认为事务已经成功提交了，如果在这之前发生断电，就认为事务提交失败了。当下次读取这些数据库的时候，SQLite 会对

所有的数据库进行恢复。

如果主数据库是一个临时数据库或者一个内存数据库，SQLite 不能保证多数据库事务操作的原子性。这也意味着全局恢复可能无法实现。SQLite 不会创建 master journal。虚拟机模块一个接一个单独执行事务提交操作。所以对于每个数据库来说，事务是原子性的。所以，如果多数据库事务提交失败，那么可能有些数据库会发生变化，有的不会。

4.4.5 子事务操作

子事务一般有三种操作，读、写、提交。接下来我们会一一讨论。

4.4.5.1 读操作

用户事务对数据库文件进行页缓存，子事务通过用户事务创建的页缓存读取数据库文件，所有的读取规则和用户事务相同。

4.4.5.2 写操作

写操作由两部分组成：锁操作和日志操作。子事务通过所属的用户事务获取锁。但是子事务的日志是一个单独的临时日志文件。SQLite 在子事务日志文件中写入一部分日志记录，还有一部分相关的日志记录被写入主回滚日志。当一个子事务试图通过 sqlite3pager_write 使某页变得可写的时候，pager 执行如下的两个动作（这两个动作是等价的）：

1.

如果该页不在主回滚日志中，就在主回滚日志中添加该页。

2.

否则，在子事务的回滚日志中添加这一页。

pager 永远不会刷新子事务日志，因为数据库恢复不需要子事务日志。如果数据库崩溃，主回滚日志会负责数据库的恢复。你可能会注意到内存中的页有可能既是主回滚日志页，又是子事务日志页，主回滚日志保存着页的内容的最近的版本。

4.4.5.3 提交操作

子事务的提交非常简单，提交完成后，pager 删除子事务日志。

4.4.6 事务撤销

事务撤销后，数据库的恢复非常简单。有的时候 pager 需要移除事务产生的

影响，有的时候不需要。如果事务只持有一个 RESERVED LOCK 或者一个 PENDING LOCK，这保证了数据库文件没有被修改，所以 pager 会删除所有的日志文件，放弃页内存中所有的脏页。否则的话，由于事务已经向数据库文件写入了一些页，pager 将执行以下的回滚操作：

Pager 从日志文件中读取记录，恢复日志中记录的页。（同一个事务只在日志文件中记录一次同一个页，该记录是该页的最近版本的镜像。）所以，恢复完成后，数据库恢复的事务执行之前的状态。如果事务扩展了数据库，pager 就把数据库文件截断为事务开始之前的大小。然后 pager 刷新数据库文件，删除回滚日志文件，释放 EXCLUSIVE LOCK，清空内存。

4.4.7 子事务撤销

就像在“子事务操作”小节中提到的，子事务有可能既向回滚日志中写入记录，也向子事务日志中写入记录。SQLite 需要回滚所有的子事务日志，还需要回滚主回滚日志的一部分。当子事务在主回滚日志中写下第一条日志记录的时候，pager 把这条记录的位置保存在内存的一个数据结构中。Pager 从这些记录中恢复数据库，然后删除子事务日志文件，但是保持主回滚日志不变。当子事务开始的时候，pager 也会记录数据库文件的大小，这样当子事务撤销的时候，pager 截断数据库文件至初始大小。

4.4.8 从错误中恢复

当发生系统崩溃或者断电的时候，不一致的数据可能被留在数据库文件中。如果没有应用程序更新数据库，但是存在一个回滚日志文件，这意味着之前的事务可能执行失败了，数据库需要进行恢复操作。如果数据库文件未被锁住或者只有 SHARED LOCK，这时我们称回滚日志文件是 hot 的。当写事务执行到一半并且发生失败的时候，我们称回滚日志文件是 hot 的。无论如何，如果回滚日志文件是多数据库事务产生的并且不存在 master journal file，这意味着事务在执行失败之前已经成功提交了。一个处于 hot 状态的日志文件表示 SQLite 需要对数据库文件进行恢复操作。

NOTE

如果不存在 master journal，存在一个回滚日志，并且数据库文件没有 RESERVED LOCK 或者更强的锁，那么这个回滚日志就是 hot 的。（持有 RESERVED

LOCK 的事务创建回滚日志文件，这时日志文件不是 hot 的）如果回滚日志文件中出现了一个 master journal 的名字，并且 master journal 文件存在，并且数据库文件没有 RESERVED LOCK，那么这个回滚日志文件就是 hot 的。

对于一般的数据库系统来说，数据库被打开的时候执行恢复操作。但是 SQLite 与它们不同，SQLite 在第一次读取操作发生的时候进行数据库文件正确性检查并且决定是否进行恢复操作。

如果当前的应用对数据库文件只有读取的权限，而没有写入的权限，那么回滚操作就会失败，SQLite 返回一个意外错误码。

在实际读取操作之前，SQLite 执行如下的恢复操作步骤：

1.

获取一个 SHARED LOCK，如果获取失败，返回 SQLITE_BUSY。

2.

检查是否存在一个处于 hot 状态的日志。如果不存在一个 hot 日志，恢复操作结束。如果存在一个 hot 日志，数据库通过下面的步骤进行回滚。

3.

申请一个 EXCLUSIVE LOCK（通过 PENDING LOCK）。（pager 不会申请 RESERVED LOCK，因为这将会导致其他事务认为这个日志文件不是 hot 的并且读取数据库中的错误数据。因为回滚操作要想数据库文件中写入数据，所以需要一个 EXCLUSIVE LOCK）如果申请失败，这意味着另一个 pager 正在尝试恢复操作。这种情况下，本 pager 放弃所有的锁，关闭数据库文件，返回 SQLITE_BUSY。

4.

从回滚日志读取所有的记录并撤销这些记录。这使数据库文件恢复到事务执行之前的状态。

5.

刷新数据库文件，防止又一次发生断电或者系统崩溃。

6.

删除回滚日志文件。

7.

如果删除 master journal 是安全的，就删除 master journal。（这一步是可选的。）

8.

释放 EXCLUSIVE LOCK（和 PENDING LOCK），但是保留 SHARED LOCK。（因为 pager 在 sqlite3pager_get 函数中执行回滚操作。）

如果上述的过程成功执行，数据库文件就恢复成功了，接下来从数据库文件中读取数据就是安全的。

NOTE

如果没有回滚日志文件指向 master journal，那么这个 master journal 就是“陈腐”的。为了识别一个 master journal 是否是陈腐的，需要检查每个回滚日志。如果有一个回滚日志指向了 master journal，那么 master journal 就不是陈腐的。如果不存在回滚日志，或者回滚日志指向其他的 master journal，或者不指向 master journal，那么这个 master journal 就是陈腐的，pager 就会删除它。删除陈腐的 master journal 并不是必须的操作，删除它的唯一原因是释放磁盘空间。

4.4.9 存档点（check point）

为了减少恢复时的工作量，大多数数据库在运行间隙的时候会设置存档点。同一时刻对于同一个数据库文件 SQLite 只允许执行一个写事务，日志文件只包含这个事务的记录，事务完成后 SQLite 删除日志文件。所以 SQLite 不需要设置存档点。事务提交后到日志删除之前，SQLite 已经确定所有的修改已经写入到数据库文件中了。

第五章 表和索引管理

之前已经讨论过了 pager 模块怎么把数据库文件抽象为基于页的文件。在这一节中，我们将讨论 B+树模块如何在基于页的文件之上抽象出基于行的数据项。

表中的行可以通过很多种形式组织起来，例如输入顺序、相关性、哈希表、键值序列。SQLite 使用 B+树来组织一个表中所有的行，不同的表有不同的 B+树。SQLite 把索引视为一个表，并且把索引存储在一个 B 树里，不同的索引有不同的 B 树。B 树和 B+树很像，都是一种键值序列数据结构。SQLite 不使用其他的组织行（数据项）的方法。所以，一个数据库就是一个 B 树和 B+树的集合。所有的这些树都是基于数据库页的，它们相辅相成，不能被分开。但是，不能有数据库页存储来自两个不同表的行。B/B+树模块的职责是组织树中的页，使得可以有效地存储和读取数据项。

在接下来的小节中，主要介绍 B+树。B+树模块实现了从树中读取、插入、删除数据项，也实现了树的创建和删除，它把数据项视为变长度的字节字符串。

5.1 B+树结构

对许多数据库来说，B 树（B 代表平衡）是最重要的索引结构。它可以按照

键值有序地组织由相似数据项构成的数据集合。B 树是一种特殊的高度平衡的树，所有的叶子节点都处于同一层。数据实体信息和搜索信息即存储在内部节点中，也存储在叶子节点中。对于一般的增删查改操作，B 树的性能几乎是最优秀的。

B+树是 B 树的一个变体，B+树把所有的数据项都存储在叶子节点中。数据项（键值和数据）通过键值排序，内部节点只保存搜索信息（键值）和子节点指针。中间节点中的键值按照顺序存储，这些键值用来指示搜索的方向。

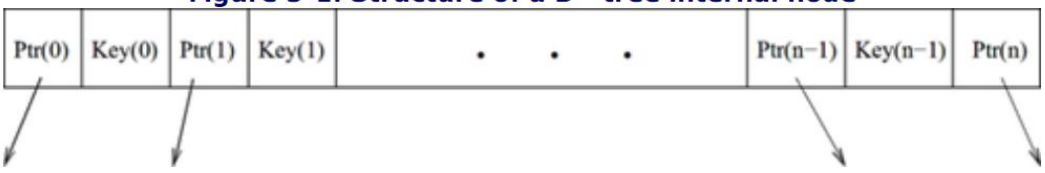
对于这两种树，内部节点可以包含指定范围内（存在上界和下界）的可变数量的子节点指针。下界通常大于或者等于上界的一半。根节点可以不遵守这条原则，它可以有任意数量的子节点指针（从 0 到上界）。所有的叶子节点都在同一层（最底层），有时候叶子节点还通过一个链表组织起来。对于 B+树来说，根节点永远是内部节点。

对于一个指定的上界 n+1(n>1)，在 B+树中，每个内部节点最多包含 n 个键值和最多 n+1 个子节点指针。键值和指针的组织方式如下图所示。对于任何内部节点：

- Ptr(0)指向的子树的所有键值都小于或者等于 key(0)
- Ptr(1)指向的子树的所有键值都小于或者等于 key(1)且大于等于 key(0)，以此类推
- Ptr(n)指向的子树的所有键值都大于或者等于 key(n-1)

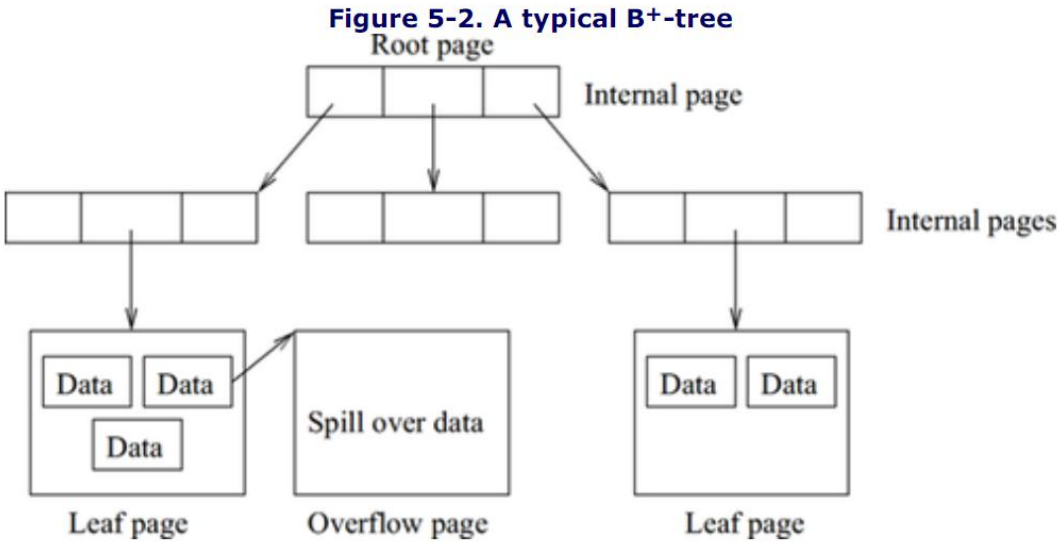
查找一个指定键值对应的数据的时间复杂度是 O(logm)，其中 m 是树中数据项的个数。

Figure 5-1. Structure of a B+-tree internal node



5.2 SQLite 内的 B+树

SQLite 通过分配根节点页来创建一颗树。根节点页不会重复分配，通过根节点页的页号来识别每棵树。页号存储在 master catalog table 中，master catalog table 的根存储在 page1 中。



一个页中只能存储一个节点（内部节点或者叶子节点），这个页就被视为节点页（内部节点页或者叶子节点页）。对于每个节点，每个数据项的键值和数据合在一起作为负载（payload）。固定数量的预先设置好的负载直接存储在页中。如果负载大于规定的数量，超过的字节存储到溢出页（overflow page）：超过的负载按照顺序存储在溢出页链表中，内部节点也可以有溢出页。

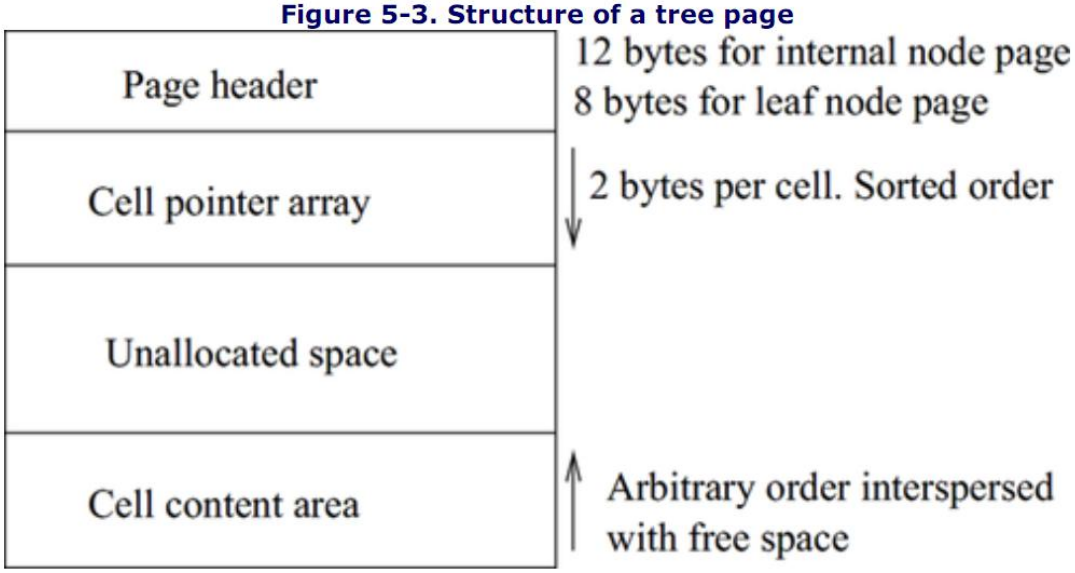
5.3 页结构

数据库文件被分为固定大小的页，所有的页由 B+树模块管理。每个页要么是树中的页（内部页或者叶子页），要么是溢出页，或者是自由页（自由页通过单链表组织起来）。在本节中我们将要学习内部页、叶子页和溢出页的结构。

5.3.1 树中页的结构

树中的页（内部页和叶子页）被分成许多 cell，一个 cell 包括一个（或者一部分）负载。Cell 是已分配或者已释放的磁盘空间集合。树中的页的格式如下图所示：

Figure 5-3. Structure of a tree page



每个页被分为 4 个部分：

1.

页头

2.

Cell 指针数组

3.

未分配空间

4.

Cell 内容

Cell 指针数组从上向下增长，cell 内容从下向上增长。Cell 指针数组作为页内的一种目录，帮助把 cell 组织起来。

页头只包含本页的管理信息，并且总是存储在页的开头。（page1 是一个例外:page1 的前 100 个 bytes 是文件头）页头的结构如下表所示，前两列的单位是 bytes。

Table 5-1. Structure of tree page header

Offset	Size	Description
0	1	Flags. 1: intkey, 2: zerodata, 4: leafdata, 8: leaf
1	2	Byte offset to the first free block
3	2	Number of cells on this page
5	2	First byte of the cell content area
7	1	Number of fragmented free bytes
8	4	Right child (the `Ptr(n)` value). Omitted on leaves.

偏移量是 0 的 flags 定义了页的格式。如果 leaf bit 为真，这意味着该页是一个叶子节点并且没有子节点。如果 zerodata 为真，那么该页只包含键值而没有任何数据。如果 intkey 为真，那么键值是一个整型变量，并且该键值被保存在 cell 头中而不是复杂区域（见之后的章节）。如果 leafdata 为真，那么树只在叶子节点存储数据。对于内部节点页，页头也包含最右边的孩子节点指针（offset 8）。

Cell 被保存在页的最底部，向页开头的方向增长。Cell 指针数组从页头之后的第一个 byte 开始，包含 0 个或者多个 cell 指针。每个 cell 指针是一个 2byte 整数，该整数指示了实际 cell 内容相对于页开头的偏移量。Cell 指针按照相应的键值排序，尽管 cell 本身的存储不是按照顺序的。Cell 指针数组的大小被存储在页头处偏移量为 3 的地方。

因为 cell 是被随机插入和删除的，存放 cell 的区域可能出现空闲空间。这些未被使用的空闲空间被一个链表收集起来，并且按照空间的地址升序排列。这个链表的头指针（一个 2byte 的偏移量）被保存在页头内偏移量为 1 的地方。每个空闲空间最小是 4bytes，因为空闲空间的头 4bytes 是用来存储控制信息的：前两个 bytes 存储下一个空闲空间的偏移量(为 0 的话表示没有下一个空闲空间了)，后两个 bytes 保存空闲空间的大小。因为空闲空间链表只能保存最小为 4bytes 的空间，所以小于 4bytes 的空间遗留在了 cell 部分的内部，这些空间被称为碎片。碎片的总大小记录在页头偏移量为 7 的地方（碎片总大小不能大于 255，在碎片总大小达到最大值之前，会对页进行一次碎片整理）。Cell 部分的第一个 byte（实际 cell 数据的起始位置）的位置存储在页头处偏移量为 5 的地方。该值作为未分配空间和 cell 部分的界限。

Cell 是变长的字节字符串。一个 cell 保存了一个负载。Cell 的结构如下图所示，size 列的单位是 byte。

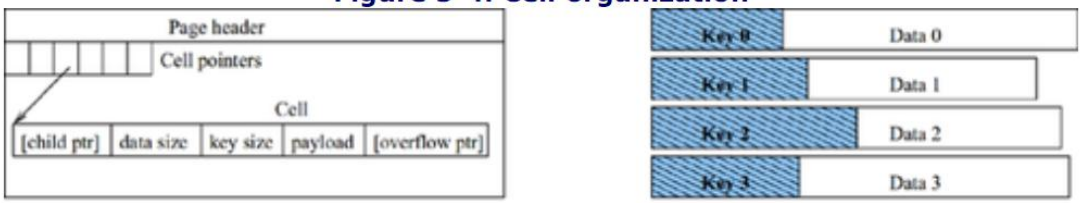
Table 5-2. Structure of a cell

Size	Description
4	Page number of the left child. Omitted if `leaf` flag bit is set.
var(1-9)	Number of bytes of data. Omitted if the `zerodata` flag bit is set.
var(1-9)	Number of bytes of key. Or the key itself if `intkey` flag bit is set.
*	Payload
4	First page of the overflow chain. Omitted if no overflow.

对于内部节点，每个 cell 包含一个 4byte 的子节点指针；对于叶子节点，cell 没有子节点指针。接下来是该 cell 存储的数据的大小（bytes）和存储的键值的大

小（bytes）（如果页头中的 intkey 为真，那么存储键值大小的地方就直接存储键值本身的整型值，如果 zerodata 为真，那么数据部分不存在）。下图展示了 cell 的结构，负载（payload）中可能不存储键值，或者不存储数据，或者两个都不存储。

Figure 5-4. Cell organization



(a) Structure of cell　　(b) Structure of payload

SQLite 使用变长整型数来表示整型数的大小（和整型键值）。可变长整数由 1~9 个字节组成，每个字节的低 7 位有效，第 8 位是标志位。在组成可变长整数的各字节中，前面字节(整数的高位字节)的第 8 位置 1，只有最低一个字节的第 8 位置 0，表示整数结束。可变长整数可以不到 9 个字节，即使使用了全部 9 个字节，也可以将它转换为一个 64-bit 整数。当可变长整数达到 9 个字节，第 9 个字节的第 8 位置不再是标志位，其 8 位均有效。这就是霍夫曼编码。使用霍夫曼编码可以大大节省存储空间。

之前我们提到过 SQLite 限制了每页中负载的数量。负载有可能不会把自身的全部存储在同一页中，尽管可能该页有足够大的空间。每页能够存储的最大单个负载由该页可用空间（可以被内部节点的单个 cell 使用）的总大小决定。这个限制值在文件头内偏移量 21 处（见数据库文件格式一节）。如果内部节点中的 cell 的负载大于最大负载限制，超出的部分就被分割并存储到溢出页链表中。一但分配了一个溢出页，要把尽可能多的字节转移到溢出页中，只要不导致 cell 的大小低于最小负载限制（该值存储在文件头内偏移量 22 处）就行。对于叶子节点，最小负载限制存储在文件头内偏移量 23 处，但是最大负载限制总是 100%，并且不会在文件头给出。

5.3.2 溢出页结构

一个 cell 的溢出页是一个单链表，每个溢出页（除了最后一个页）被填满数据，数据长度等于可用空间除以 4bytes：头 4 个 bytes 存储下一个溢出页的页号，最后一个溢出页可以小到只有 1byte 数据。不能在同一个溢出页存储来自两个 cell 的内容。

5.4 模块职能

本模块帮助虚拟机把所有的表和索引通过 B/B+树的形式组织起来：为每个表分配一个 B+树，为每个索引分配一个 B 树。每个树由一个或者多个页组成。虚拟机可以存储或者读取树中变长度的数据项，也可以在任何时间从树中删除一个数据项，并且拓展空间或者重用自由空间。对于一个含有 m 个数据项的树，本模块帮助虚拟机实现 O（logm）的查找时间复杂度。

5.4.1 空间管理

本模块接受来自虚拟机的插入或者删除 cell 的请求。插入操作需要对树上的页（和溢出页）分配空间，相对的，删除操作释放空间。每页自由空间的管理对于数据库有效利用空间是很关键的。

5.4.1.1 自由页的管理

当一页从树中移除的时候，该页被添加到自由页链表中，为稍后的重复使用做准备。当树需要扩展的时候，从自由页链表中取出一页添加到树上。如果自由页链表为空，就从本地文件系统中取出一页。（从本地文件系统取出的页总是添加在数据库文件的末尾。）

5.4.1.2 页内空间管理

树上的页有三种类型的自由空间：

1.

Cell 指针数组到 cell 实体之间的空间（称为未分配空间）。（可分配）

2.

cell 实体间的自由空间块（可分配）（通过一个自由空间块链表组织起来）

3.

Cell 间的碎片空间（不可分配）

每次分配和释放空间后，造成的自由空间变化遵循以下的原则：

为 cell 分配空间

空间分配器不会分配小于 4bytes 的空间，如果有小于 4bytes 的申请，那么空间分配器分配 4bytes 的空间。假设某页有 nFree bytes 的空间，对于该页有 nRequired bytes（nRequired>=4）的申请，nRequired <= nFree，那么空间分配器按照以下的步骤分配空间：

1.

遍历自由空间块链表，寻找是否有足够大的自由空间块，如果找到了，按照如下原则继续执行：

a.

如果自由空间块的大小小于 nRequired + 4，就把该空间块从自由空间块链表中取出来，用从空间块开头算起的 nRequired bytes 满足申请，剩下的空间（<=3 bytes）就成了碎片空间。

b.

否则，用从空间块末尾算起的 nRequired bytes 满足申请，剩下的自由空间依旧作为自由空间块存储于自由空间块链表中。

2.

如果没找到足够大的自由空间块，而且需要分配的空间超过了未分配空间，或者该页有太多的碎片空间，那么本模块就对该页进行碎片整理。通过执行压缩算法把碎片集合成更大的自由空间，并放到 Cell 指针数组与 cell 实体之间。在压缩过程中，需要一个接一个地把已经存在的 cell 移动到页的底部。

3.

从未分配空间的底部开始，分配 nRequired bytes 的空间。

释放 cell 占有的空间

假设有一个释放 nFree（>=4 bytes）（这 nFree 之前被分配器分配过）的请求。分配器创建一个新的大小为 nFree bytes 的自由空间块，把这块自由空间插入到自由空间块链表中合适的位置。然后尝试将该块和周围的空间块合并。如果两个空间块之间有碎片，这些碎片也会被合并。如果 cell 指针数组和 cell 实体之间有未分配空间，那么将释放的空间和未分配空间合并。

第六章 SQLite 引擎

后台系统的顶层模块是虚拟机。虚拟机是 SQLite 的核心，也是前后台系统的交互接口，核心信息处理在其中完成。虚拟机在本地操作系统上又抽象出一个机器，用来执行 SQLite 内部的字节码语言写成的程序。虚拟机接收前台生成的字节码程序（字节码程序就是预处理过的 SQL 命令），利用 B+树模块执行该程序，产生输出。

虚拟机不知道字节码程序执行了什么操作，它只是在必要时刻把数据从一种类型转换为另一种类型。动态数据转换是虚拟机的主要任务，其他事情由字节码程序控制。

字节码程序被 sqlite3_stmt（内部叫做 Vdbe）类型的内存实体包裹。运行字节码程序和取回结果的 API 有： sqlite3_bind_*, sqlite3_step, sqlite3_column_*, sqlite3_finalize.

一个 Vdbe 实体的内部状态包括：

- 一个字节码程序
- 所有结果列的名字和数据类型
- 绑定到输入参数上的值
- 一个程序计数器
- 一个操作数的执行栈
- 不定数量的编号过的内存单元
- 其他运行时状态信息（例如 B 树实体、排序器、链表、集合）

6.1 字节码编程语言

SQLite 定义了一种内部编程语言来预处理字节码程序，这种语言类似汇编语言，它会定义字节码的结构：<opcode, P1, P2, P3>。Opcode 指明了一个确定的操作，p1,p2,p3 是该操作的操作数。P1 是一个 32 位有符号整型变量。P2 是一个 31 位非负整型变量，对于可能导致跳转的操作，p2 总是存储跳转的目的地址。P3 是一个指向无结尾的字符串，或者指向结构体，或者指向 NULL。一些操作码使用这 3 个操作数，一些只使用一个或两个操作数，甚至不使用操作数。

NOTE

操作码是虚拟机内部操作的名字，并不是 SQLite 接口规格的一部分。所以，不同的版本操作码的语义可能不一样。SQLite 开发小组不推荐用户自己书写字节码。

下表展示了等价于 SELECT * FROM t1 的一个字节码程序。表 t1 有两列：x，y。表的第一行不是程序的一部分，所有其他行都是字节码指令。

Table 6-1. A typical bytecode program

Address	Opcode	P1	P2	P3
0	Goto	0	11	
1	Integer	0	0	
2	OpenRead	0	2	#t1
3	SetNumColumn	0	2	
4	Rewind	0	9	
5	Column	0	0	#x
6	Column	0	1	#y
7	Callback	2	0	
8	Next	0	5	
9	Close	0	0	
10	Halt	0	0	
11	Transaction	0	0	
12	VerifyCookie	0	1	
13	Goto	0	1	

虚拟机是一个翻译器，以下是它的结构：

```
for (; pc < nOp && rc == SQLITE_OK; pc++){
    switch (aOp[pc].opcode){
    case OP_Add:
        /* Implementation of the ADD operation here */
        break;
    case OP_Goto:
        pc = op[pc].p2-1;
        break;
    case OP_Halt:
        pc = nOp;
        break;
    /* other cases for other opcodes */
    }
}
```

翻译器是一个简单的循环，包括了大量的 switch 语句。每个 case 语句实现一个字节码指令。（操作码以 OP_为前缀）在每次迭代中，虚拟机从字节码程序中取到下一个字节码指令，例如，使用 pc 作为索引从 aOp 数组中取得（pc 和 aOp

都是 Vdbe 实体的成员）。虚拟机对指令进行解码并且执行该指令。虚拟机从指令编号为 0 的字节码程序开始执行。

虚拟机通过一个游标进入数据库，游标是指向单个 B+树（表）或者 B 树（索引）的指针。游标可以通过键值查找数据，也可以遍历整个树。虚拟机通过游标执行数据项的插入、读取、删除操作。

虚拟机使用操作数栈和一定数量的编号过的内存空间来保存中间结果。操作码使用栈中的操作数，计算结果也存储在栈中。

虚拟机依次执行字节码程序直到遇到 halt 指令或者发生错误（在翻译器程序中，rc 变量保存了指令执行的状态），或者程序计数器已经超过的最后一条指令的编号。如果是错误导致的执行终止，虚拟机会终止事务或者子事务，去除事务（子事务）对数据库的影响。

6.2 记录格式

虚拟机把数据项存储在 B/B+树中，每个数据项由键值和数据组成。只有虚拟机负责维护键值和数据的内部结构（尽管 B+树模块可能把数据项分割到溢出页中，虚拟机把数据项视为逻辑上连续的字节字符串）。对于表和索引，SQLite 使用两种相似但是不同的数据项格式。

有两种方法格式化数据项：固定长度的和变长度的。对于固定长度的格式，所有的数据项都使用相同的存储空间（无论是表还是索引），每个固定储存空间的大小在表（或索引）创建的时候就确定了。对于变长度的格式，存储空间的大小可能各不相同。变长度格式可以大大缩减数据库文件的大小，因为内存和磁盘之间需要迁移的数据变小了，所以整个系统运行地更加快速。另外，变长度数据项的使用使得 SQLite 可以支持动态类型。

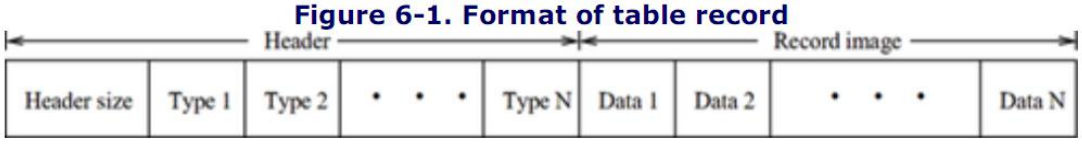
6.2.1 动态类型

每个存储在数据库的原始数据值都有一个指定的格式。大多数关系型数据库使用静态格式：每列只有一种数据格式，只有符合这种格式的数据才允许存储在该列中。SQLite 通过使用动态数据类型放松了这条限制。在动态类型中，数据类型变成了数据本身的一个特性，而不是列或者存储数据的变量的特性。SQLite 使用动态类型（尽管 SQLite 规格中称之为静态类型），数据类型作为数据的一部分

一起被存储。SQLite 允许用户在任何列中存储任何类型的数据，无论该列声明的是什么类型。（但是有一个例外：整型主键列只能存储整型数据）

6.2.2 表数据项格式

Figure 6-1. Format of table record



表的数据格式（行数据）如上图所示，数据项由数据头和数据体组成。数据头依次存储了大小（从数据项开头到 data1 之前，该变量是一个经过哈夫曼编码的 64bit 的变长度整型），数据类型（变长度无符号整型，最大值是 2 的 64 次方）。之后是数据本体，数据类型和数据本体一一对应。

虚拟机支持五种存储类型：有符号整型，有符号浮点型，字符串，二进制大对象，NULL。有些数据可能有多种类型，例如 123 可以是一个整数，或者一个浮点数，或者一个字符串。二进制大对象和 NULL 不可能有多种类型。SQLite 必须对数据类型进行隐式转换。

这种数据类型的编码形式如下表所示，这种编码方式的好处是数据长度变成了数据类型编码的一部分。

Table 6-2. Storage types and their meanings

Type value	Meaning	Length of data
0	NULL	0
N in {1..4}	Signed integer	N
5	Signed integer	6
6	Signed integer	8
7	IEEE float	8
8	Integer constant 0	0
9	Integer constant 1	0
10, 11	Reserved for expansion	N/A
N>=12 and even	Blob	(N-12)/2
N>=13 and odd	Text	(N-13)/2

NULL 类型就是 SQL NULL。对于整型，数据值是有符号整型数，根据数据的大小分别存储在 1，2，3，4，6，8bytes 中。浮点型数据按照 IEEE 浮点数标准存储在 8bytes 中。类型编码 8 和 9 代表整型常数 0 和 1.对于 TEXT 类型，使用默认的字符编码格式（UTF-8，UTF-16BE，UTF16-LE）存储在数据库文件中，对于 UTF-16BE 和 UTF16-LE 编码格式，分别是大端存储和小端存储的。（每个数据库文

件只能存储同一种 UTF 格式）对于二进制大对象（BLOB），数据值是一个二进制块，按照输入原样存储。

6.2.3 表的键值格式

在 SQLite 中，每个 B+树必须有一个主键。尽管定义一个关联表不能包含同样的行，事实上，用户可能在关联表中存储数据项的副本。数据库系统必须把数据项和它的副本区分开，即系统必须能够处理附加的用于区分的信息。这意味着系统要为关联表提供一个新的主键。所以，在内部，每个表都有一个独一无二的主键，该键值要么是表的创建者定义的，要么是被 SQLite 创建的。

对于每个表，SQLite 指定一个列作为行号（rowid，一个－2^63 到 2^63-1 的整型数），该列中的值唯一地确定一行。行号是表的隐式主键，是 B+树的唯一搜索键值。如果有一列被声明为整型主键列，该列就被视为所在表的行号列。所以，对于每个表，无论有没有声明一个整型主键列，都有一个唯一的整型值，即行号。对于后一种情况，行号在内部被视为表的整型主键。

下表展示了表的内容，该表是通过 create table t1(x ,y)命令创建的。行号列是通过 SQLite 添加的。行号值通常是由 SQLite 设定的。然而，你可以向行号列添加任何整型数，例如：insert into t1(rowid, x, y) values(100, 'hello', 'world')。

Table 6-3. A typical SQL table with rowid as key, and x and y as data

rowid	x	y
-5	abc	xyz
1	abc	12345
2	456	def
100	hello	world
54321	NULL	987

NOTE

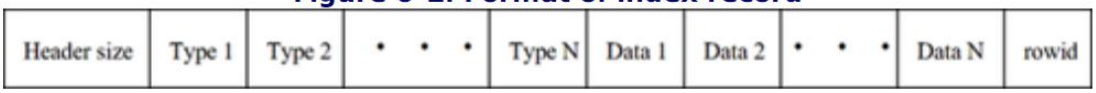
如果行号列是自定的（例如声明为整型主键），数据库使用者负责管理列的值。如果行号列是 SQLite 添加的，那么 SQLite 负责生成行号值并保证它们的唯一性。当插入新行的时候，SQLite 访问 B+树并找到一个未使用的整数来做为行号。

当行号作为数据的一部分被存储的时候，行号有一个内部整数类型。当作为一个键值存储的时候，行号是一个变长的哈夫曼编码。负数行号是允许的，但是需要用 9bytes 存储它们，所以并不推荐使用负数行号。SQLite 添加的行号总是非负的，尽管你可以显式声明负数行号，上面的-5 就是一个例子。

6.2.4 索引值格式

在前面的章节中，我们已经介绍了每个 B+树的键值是一个整数，数据是表中的一行。索引刚好相反。对于一个索引，键值是该行索引的所有列的组合，数据是该行的行号。为了进入索引表中的一行，SQLite 首先搜索索引表来找到相关的整数值，然后使用这个值来在 B+树中查找完整的数据。

Figure 6-2. Format of index record



SQLite 把索引视为一种表，并把索引存储在 B 树中。它把搜索键映射为一个行号，排序函数可以对索引进行排序。每个索引记录包含索引列值（索引到的行号后面的列）的拷贝。索引记录的格式如上图所示。整个记录作为 B 树的键值，没有数据部分。索引记录的编码格式和表的编码格式相同，除了行号是后添加的，行号的类型不会出现在记录头中，因为该类型总是有符号整型并且经过哈弗曼编码。（其他数据的值和存储格式都是从表中拷贝出来的。）列 x 的一个索引如下表所示：

Table 6-4. An index on column x

x	rowid
NULL	54321
456	2
abc	-5
abc	1
hello	100

SQLite 也支持多列索引。下表表示了包含 y 和 x 的一个索引。索引中的内容按照他们第一列的值排序。

Table 6-5. An index on columns y and x

y	x	rowid
987	NULL	54321
12345	abc	1
def	456	2
xyz	abc	-5
world	hello	100

索引主要用于加快数据库的搜索速度。例如，对于查询请求：SELECT y FROM t1 WHERE x=456，SQLite 对 t1(x)进行一次索引查找，然后找到所有的 x=456 的行号，在表 t1 中搜索所有的行号来获取相应的 y 值。

6.3 数据类型管理

SQLite 数据处理发生在虚拟机模块中。虚拟机是唯一对数据进行操作的模块，具体操作由它执行的字节码程序来控制，字节码程序决定把数据存储到哪里，或者从哪里读取数据。为数据分配合适的存储类型，做一些必要的类型转换，这两者是虚拟机的主要任务。有三处数据交换的地方可能发生类型转换：从应用程序到虚拟机引擎，从引擎到应用程序，从引擎到引擎。对于前两种情况，虚拟机为用户数据指定类型。虚拟机会尝试把用户数据转化为 SQL 类型，或者反过来，把 SQL 类型转换为用户数据类型。对于最后一种情况，数据转换要求进行表达式求值。接下来的小节中会详细讨论这三种数据转换问题。

6.3.1 为用户数据指定类型

6.2 节已经介绍了表和索引的存储格式，每个数据都会有一个存储类型。任何输入给 SQLite 的值，无论是 SQL 命令中的字面量，或者是绑定在预处理命令上的值，都会在命令执行之前被分配一个存储类型。该类型被用于对数据进行合适的编码。虚拟机通过 3 步来决定给定列的输入值的存储格式：首先决定输入值的存储类型，然后决定列的 SQL 类型，最后，如果需要的话，进行类型转换。对于接下来描述的情况，SQLite 可能在数字存储类型（整型和浮点型）和文本存储类型之间转换。

6.3.1.1 输入数据类型

SQL 命令的部分字面量被分配如下的存储格式：

- TEXT，如果值是被单引号或者双引号括起来的
- INTEGER，如果值未被括起来且没有小数点或者指数符号
- REAL，如果未被括起来且有小数点或者指数符号
- BLOB，如果该值使用了 X’ABCD’说明

否则，输入值被拒绝，查询请求失败。使用 sqlite3_bind_*绑定的 SQL 参数值会被分配一个和绑定类型最符合的存储类型（例如：sqlite3_bind_blob 绑定的参数会被分配为 BLOB 类型）。

一个值的存储类型依赖于表达式的最终运算符。用户定义的函数可能返回任何存储类型的值。一般来说，在 SQL 命令预处理的时候是无法决定表达式结果的类型的。虚拟机在运行时为获得的值指定类型。

6.3.1.2 列之间的联系

尽管列（除了整型主键列）可以存储任意类型的值，该值可能和所在列的 SQL 类型有一定关联。为了最大化 SQLite 和其他静态类型的数据库之间的兼容性，SQLite 支持类型关联的概念。一个列的关联类型就是该列的推荐存储类型，这只是一种推荐，而不是强制要求，该列仍然可以存储任何类型。也就是说存储的时候会优先考虑关联类型，然后才是其他类型。

NOTE

SQLite 是弱类型的，例如：没有域限制。它允许在任何列中存储任何类型的数据，无论该列声明的是什么类型。（行号列是例外，该列只存储整型数，拒绝存储其他数。）这可以让你在使用 create table 命令的时候省略 SQL 类型声明。例如：create table T1(a, b, c)是一个合法的命令。

列的优先声明类型称为关联类型。列的类型是 5 种类型之一：TEXT, NUMERIC, INTEGER, REAL, 和 NONE。（"text," "integer," 和 "real"同时也是存储类型，这可能导致命名冲突，可以通过上下文决定类型，这样就能解决了）列的关联类型依赖于 CREATE TABLE 命令为该列声明的类型，关联类型的推断依据如下原则：

1.

如果 SQL 类型包含子字符串 INT，那么该列的关联类型是整型

2.

如果 SQL 类型包含子字符串 CHAR，CLOB, 或者 TEXT，该列的关联类型就是 TEXT。（SQL 类型 VARCHAR 包含字符串 CHAR，所以它的关联类型是 TEXT）

3.

如果 SQL 类型包含子字符串 BLOB，或者未指明任何类型，那么该列没有关联类型。

4.

如果 SQL 类型包含子字符串 REAL, FLOA, 或者 DOUB，那么该列的关联类型是 REAL。

5.

否则，关联类型是 NUMERIC。

虚拟机对上面的规则依序进行评估。模式匹配是大小写不敏感的。例如：如果某列声明的 SQL 类型是 BLOBINT，关联类型就是 INTERGER，而不是 NONE。如果一个 SQL 表是使用 create table table1 as select...命令创建的，那么所有的列都没有 SQL 类型，它们的关联变量是 NONE。隐式行号总是整型的。

6.3.1.3 类型转换

存储类型和关联类型之间存在一种关系。如果一个用户输入的值不满足该关系，该值要么被拒绝要么被转换为合适的格式。当一个值被插入列中的时候，虚拟机首先指定最适合的存储类型（见 6.3.1.1），然后尝试把数据类型转换为该列的关联类型，转换规则如下所示：

1.

一个关联类型是 TEXT 的列存储 NULL，TEXT，或者 BLOB 存储类型的数据。如果一个数字值（整型或者浮点型）被插入到该列，那么数据就会转换为文本格式，最终的存储类型也变为 TEXT。

2.

一个关联类型是 NUMERIC 的列可能包含所有的 5 中存储类型。当一个文本格式的值被插入到 NUMERIC 列的时候，首先尝试把该值转换为整型或者浮点型。如果转换成功，就使用 INTEGER 或者 REAL 存储类型来存储这个值。如果转换失败，该值按照 TEXT 存储类型存储。不会去尝试把数据转换为 NULL 或者 BLOB。

3.

一个关联类型是 INTEGER 的列和关联类型是 NUMERIC 的列一样，唯一例外的地方是：如果插入一个没有小数点的浮点型数据（或者一个被转换成没有小数点的浮点型数据的文本数据），该值被转换为整型，并且使用 INTEGER 存储类型存储。

4.

一个关联类型是 REAL 的列和关联类型是 NUMERIC 的列相似，唯一例外的地方是：它会强制把整型数据转换为浮点型。（SQLite 对此做了优化，整型数以整型的形式存储在磁盘上，这样节省了空间，只有被读取的时候才会转换为浮点数）

5.

关联类型是 NONE 的列没有优先转换类型，虚拟机不会进行任何转换尝试。

NOTE

所有的关系型数据库都有类型转换，它们拒绝接收不能转换为期望类型的数据。SQLite 与之不同，即使格式转换是不可能的，也会存储这个数据。例如：如果有一个列的类型是 INTEGER，想要在其中插入一个 string，虚拟机将检查 string 是否看起来像一个数字。如果是的话，该 string 就被转换为一个数字，并以 REAL 或者 INTEGER 的类型存储。但是如果数据不是数字的话（例如“abc”），那么就使用 TEXT 存储类型来存储这个 string。如果列的关联类型是 TEXT，那么在存储数据之前会先尝试把数字转换成 ASCII 码文本的形式。但是 BLOB 不能以 BLOB 的形式存储在 TEXT 列，因为 SQLite 不能把 BLOB 转换为 TEXT。SQLite 允许把 string 插入到整型列中，这是一种特性，而不是一个 bug。

6.3.1.4 一个简单的例子

```
CREATE TABLE T1(a, b, c);
INSERT INTO T1 VALUES(177, NULL, 'hello');
```

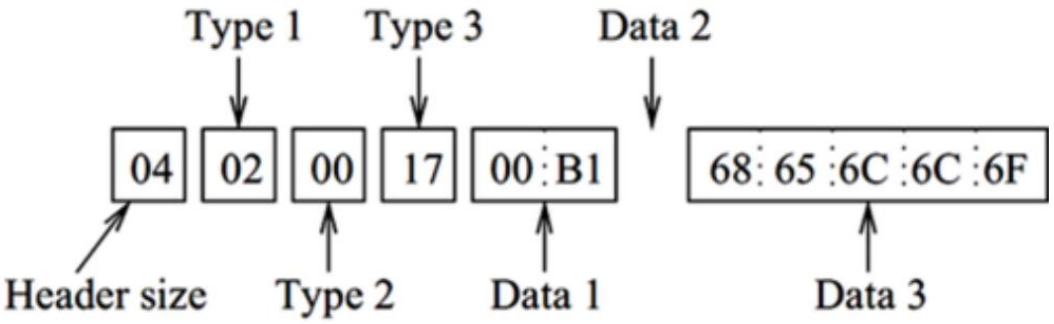


Table data record for the INSERT operation

让我们通过一个简单的例子来更好地说明。一个写好的行如上图所示。a,b,c 的初始类型是 integer,null,text。所有列的关联类型都是 NONE，虚拟机不会进行任何类型转换。在上面的例子中，该行(数据头加上数据本身)由 11 个 bytes 组成：（所有数字都是 16 进制的）

1.

数据头是 4bytes 大小，第一个 byte 表示数据头本身的大小，随后的 3 个 bytes 表示 3 个动态类型。

2.

type1 是 2（代表一个 2bytes 的有符号整型）,被编码为 0x02。

3.

type2 是 0，代表 NULL，被编码为 0x00。

4.

type3 是 23（代表文本格式，（23-13）/2=5bytes 大小），被编码为 0x17.

5.

data1 是一个 2bytes 的整型 00B1，转化为十进制是 177，注意不能用一个 byte 来编码 177，因为 B1 是-79，而不是 177.

6.

data2 是 NULL，不占用数据项中的任何 byte。

7.

data3 是一个 5bytes 的 strng：68 65 6C 6C 6F，终止符“/0”被省略了。

6.3.2 把来自引擎的数据转换为用户数据

sqlite3_column_*函数从 SQLite 引擎读取数据，并尝试把数据转换为合适的类型。例如，如果内部表示是 FLOAT，需要读取的是文本格式的结果（sqlite3_column_text），那么在返回数据之前，虚拟机使用 sprintf()库函数在内部进行转换。下表展示了数据转换的规则。

Table 6-6. Data conversion protocol

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is NULL pointer
NULL	BLOB	Result is NULL pointer
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as for INTEGER⟶TEXT
FLOAT	INTEGER	Convert from float to integer
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	Same as FLOAT⟶TEXT
TEXT	INTEGER	Use `atoi()` C library function
TEXT	FLOAT	Use `atof()` C library function
TEXT	BLOB	No change
BLOB	INTEGER	Convert to TEXT then use `atoi()`
BLOB	FLOAT	Convert to TEXT then use `atof()`
BLOB	TEXT	Add a `\000` terminator if needed

6.3.3 为表达式数据指定类型

在对内部数据进行比较或者表达式计算之前，虚拟机不能转换内部数据。它对内部数据使用如下的转换方法。

6.3.3.1 处理 NULL 型数据

NULL 型数据可以被存储在除了主键列之外的任何列中。NULL 的存储类型就是 NULL。SQL 标准并没有特别说明如何处理列中的 NULL 型数据。例如：我们应该如何比较 NULL 和其他的 NULL 或者其他的数据？SQLite 处理 NULL 的方法和其他许多数据库相同。对于 SELECT

DISTINCT 命令（UNION 和 GROUP BY 都是复合 SELECT 命令）来说，NULL 并不会被特殊处理。对于一个 UNIQUE 列，内建的 SUM 函数会按照 SQL 标准处理 NULL。对 NULL 进行运算会得到 NULL。

6.3.3.2 表达式的类型

SQLite 支持三种类型的表达式操作：

二进制比较操作符=，<,<=,>,>=,!=

集合成员操作符 IN

三元比较操作符 BETWEEN

比较的输出依赖于两个被比较的值的类型，根据如下的规则：

1.

如果左侧值的存储类型是 NULL，那么它就被认为比其他任何值都小（包括另外一个值是 NULL 的情况）。

2.

INTEGER 和 REAL 类型的值要比 TEXT 和 BLOB 类型的值小。当 INTEGER 或者 REAL 和 INTEGER 或者 REAL 进行比较的时候，按照正常的数字比较方式比较。

3.

TEXT 值比 BLOB 值小。当两个 TEXT 进行比较的时候，使用标准 c 语言库函数 memcmp 决定那个大。无论如何，这可以被用户自己的函数重写。

4.

当两个 BLOB 进行比较的时候，总是使用 memcmp 进行比较。

在应用这些规则之前，虚拟机的首要任务是决定比较操作的操作数的存储类型。它首先决定操作数的初步存储类型，然后（如果需要的话），依据关联类型转换数据，最后，使用上述的 4 条规则转换数据。

如果表达式是列，或者通过别名持有的列的引用，那么表达式的关联类型就是该列的关联类型。否则，表达式没有 SQL 类型，并且它的关联类型是 NONE。在比较之前，SQLite 会尝试在 NUMERIC 存储类型（INTEGER and REAL）和 TEXT 存储类型之间进行转换。对于二进制比较，这种转换会在下一个枚举量上进行。表达式指的是任何 SQL 标量表达式或者字面量，而不是一个列值。

当两个列值比较的时候，如果任一列有 NUMERIC 的关联类型，那么两个值的优先关联类型就是 NUMERIC。这意味着在比较之前，虚拟机尝试把另一个值转换为 NUMERIC。

当列与表达式的结果进行比较的时候，在比较之前，列的关联类型会被应用到表达式的结果上。

当两个表达式进行比较的时候，不会进行任何转换。比较会按照上面提到的 4 个原则进行，数字永远小于字符串。

在 SQLite 中，表达式 a BETWEEN b AND c 等价于 a >= b AND a <=c，尽管这意味着对 a 应用了不同的比较关系，要求计算两个表达式。

使用上面枚举的三条关于二进制等于的规则处理 a IN (SELECT b ...)这样的表达式。例如：如果 b 是一个列值，a 是一个表达式，那么在进行比较之前，b 的关联类型被应用到 a 上。SQLite 把 a IN (x, y, z)这样的表达式等价于 a = z OR a = y OR a = z，尽管通过不同的相等表达式对 a 应用了不同的比较关系。

我们通过一些简单的例子来说明。假设有一个表 t1CREATE TABLE t1(a TEXT, b NUMERIC, c BLOB)创建的，通过 INSERT INTO t1 VALUES ('500', '500', '500')向其中插入一行。a,b,c 的最终存储格式是 TEXT, INTEGER 和 TEXT.

在比较之前，SELECT a < 60, a < 40 FROM t1 把 60 和 40 转换为"60"和"40"。因为 a 的关联类型是 TEXT。命令返回 1|0,因为作为字符串，“500”小于“60”，不小于“40”.

SELECT b < 60, b < 600 FROM t1 不会进行任何转换，按照正常比较数字的方法进行比较，返回值是 0|1。

SELECT c < 60, c < 600 from t1 不会转换 60 和 600，因为 c 的关联类型是 NONE。这两个值（存储类型是 NUMERIC）比“500”（存储类型是 TEXT）小，所以返回值是 0|0。

6.3.3.3 操作符类型

所有的算术操作符（除了异或操作符||）使用 NUMERIC 作为操作数的关联类型。如果有操作数不能转换为 NUMERIC，那么操作的结果是 NULL。对于异或操作符，两个操作数的关联类型是 TEXT。如果有一个操作数不能转换为 TEXT（因为它是 NULL 或者 BLOB），那么异或操作的结果是 NULL。

6.3.3.4 ORDER BY 中的类型

当数据通过 ORDER BY 进行排序的时候，排序之前不会发生任何存储类型转换。比较标准如下：先是存储类型为 NULL 的值，然后是 INTEGER 和 REAL 值（按照正常的数字排序方法排列），然后是 TEXT 值，一般是 memcmp()排序，最后是 BLOB 值，也是通过 memcmp()排序。通过重写 memcmp()函数，可以自定义文本排序。

6.3.3.5 GROUP BY 中的类型

当通过 GROUP BY 把数据集合起来的时候，集合操作之前不进行任何类型转换。不同的存储类型会被区别对待，除了 INTEGER 和 REAL 值，如果按照数字比较这两个值是相等的，那么就认为它们是相等的。

6.3.3.6 SELECT 复合指令中的类型

SELECT 复合操作（即 UNION, INTERSECT 和 EXCEPT）在值之间进行隐式比较。在比较之前，关联类型可能被应用到每个值上。同样的关联类型会应用到所有的值上。被应用的关联类型是最左边的 SELECT 复合操作（在该处有一个列值，该列值不是其他类型的表达式）返回的列的关联类型。如果对于一个给定的 SELECT 复合操作，没有返回列值，那么在比较之前，不会有关联变量应用到值上。

第七章 其他信息

在之前的章节中我们讨论了设计原则，工程权衡，SQLite 的实现问题，并且展示了一些简单的应用程序。由于篇幅所限，本文不会讨论前台的分析系统。我鼓励你访问 SQLite 的官网获取更多信息。你也可以向

sqlite-users-subscribe@sqlite.org 发送邮件来订阅。SQLite 的源代码已经被很好地注释过了，并且源代码内部也有很多说明信息，这可以使你更好地学习 SQLite。