1. 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

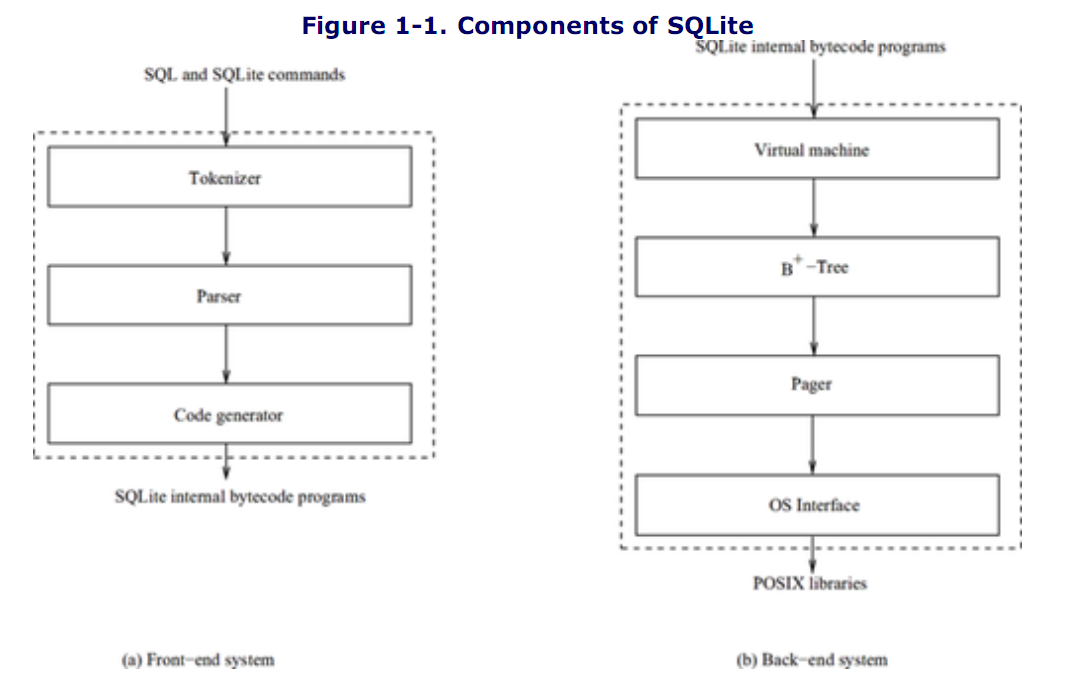
SQLite允许向单个sqlite3\_exec函数传入多个SQL语句，例如：begin; insert into Students values(2000); insert into Courses values('SQLite Database', 2000); commit.

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。



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264-1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230bytes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。

1. 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（231-1）。（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

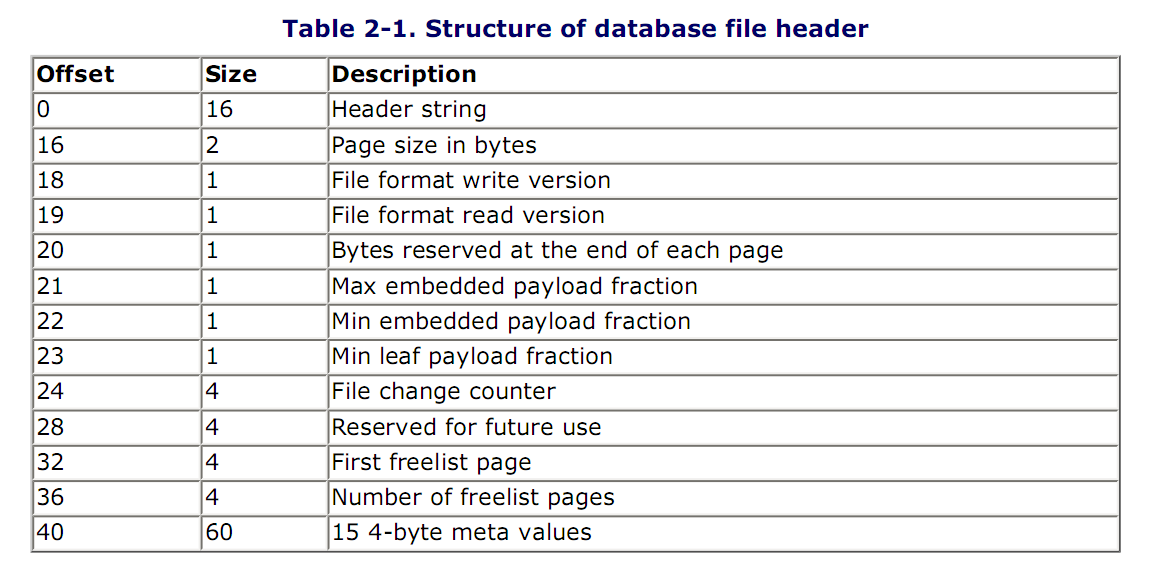
每当一个数据库文件被创建，SQLite使用默认页大小（在编译时指定的大小），在创建数据库的第1个表之前，可以使用预处理命令修改页的大小。

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。



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%)。

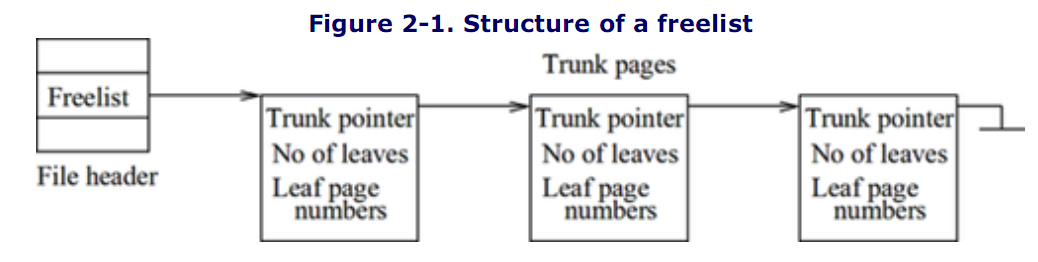
叶子页最小负载（偏移量23）和最小embedded payload相似，但是是针对B+树的叶子页来说的。（默认值是32,12.5%）。叶子页的最大负载百分比永远是100%，文件头中并没有保存该值的变量。

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。



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。

1. 页缓存管理（Page Cache Management）

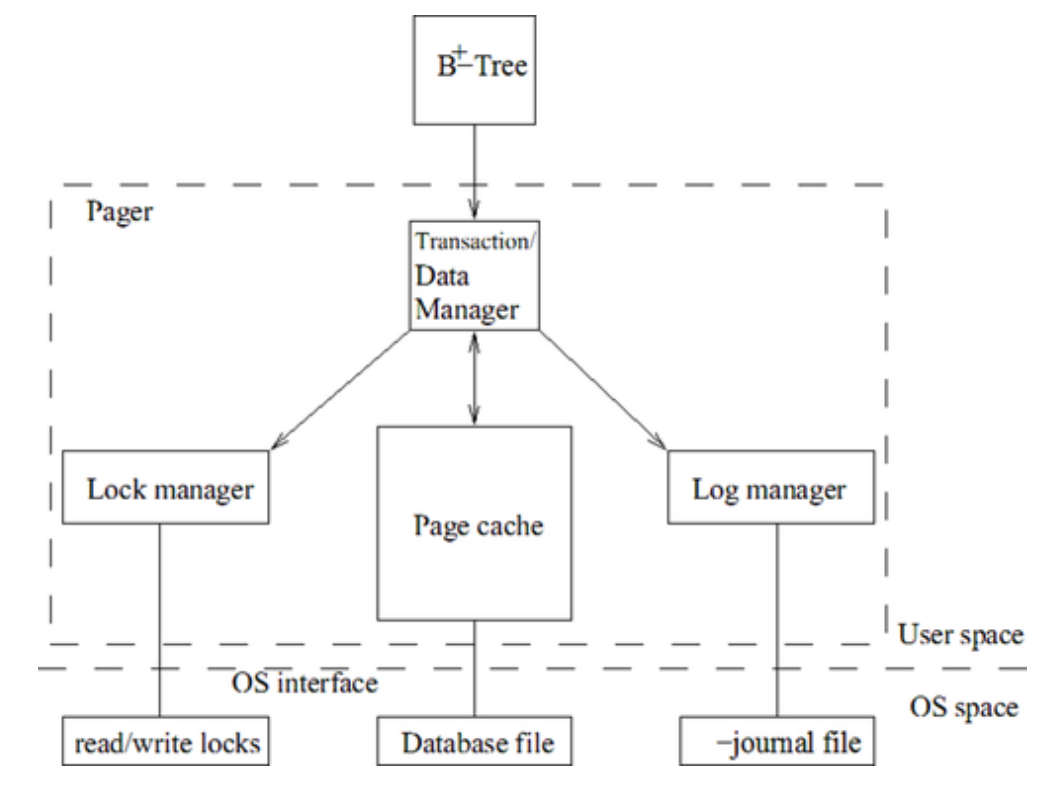
Pager是进入数据库文件和日志文件的唯一模块（通过操作系统的本地IO API）。但是它并不翻译数据库的内容，也不独立修改数据。（pager可能修改一些文件头信息，比如文件修改次数）。Pager基于随机存取/字节导向的文件系统操作，然后将其抽象为随机存取/页导向的文件系统操作。它定义了易用的、文件系统无关的进入数据库文件页的接口。B+树模块总是使用页接口来进入数据库，而不直接进入任何数据库文件或者日志文件。它将数据库文件视为统一大小的页组成的数组。

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。

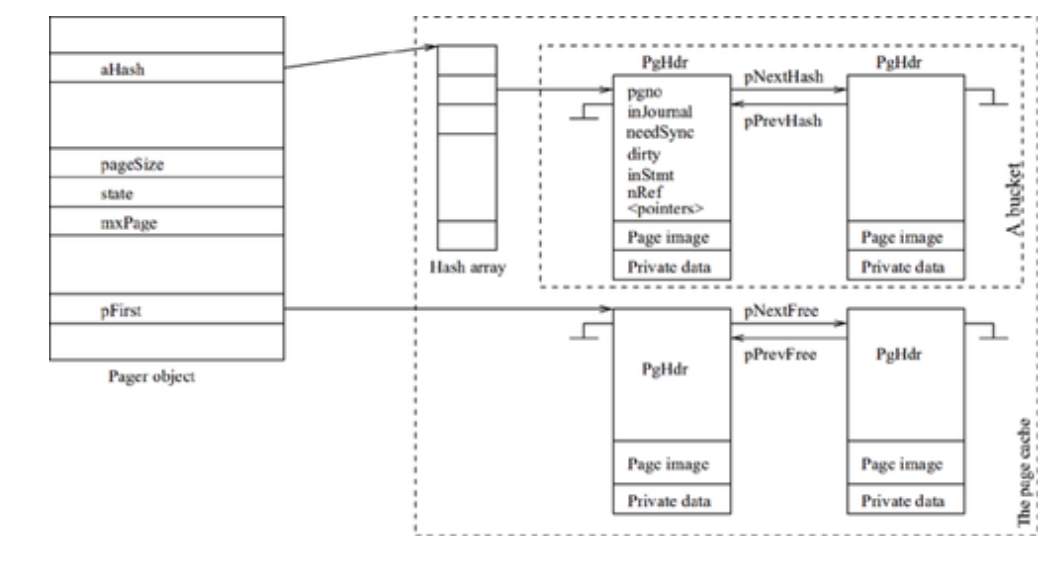
NOTE

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。

1. 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。

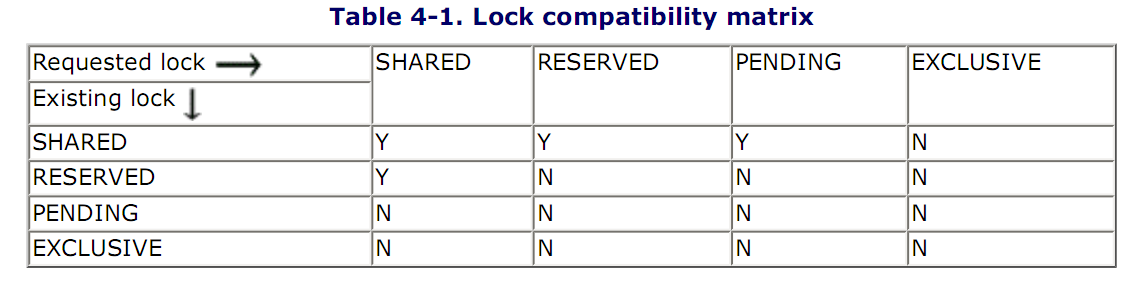
PENDING

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。

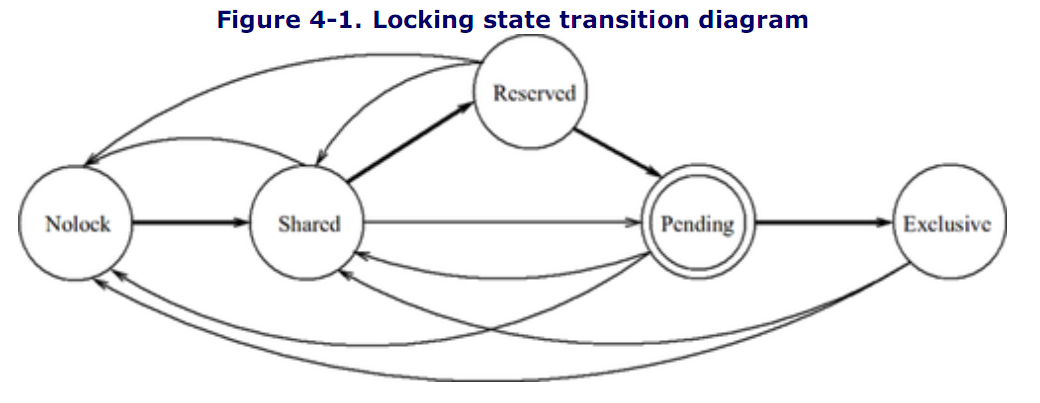


NOTE

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。



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。

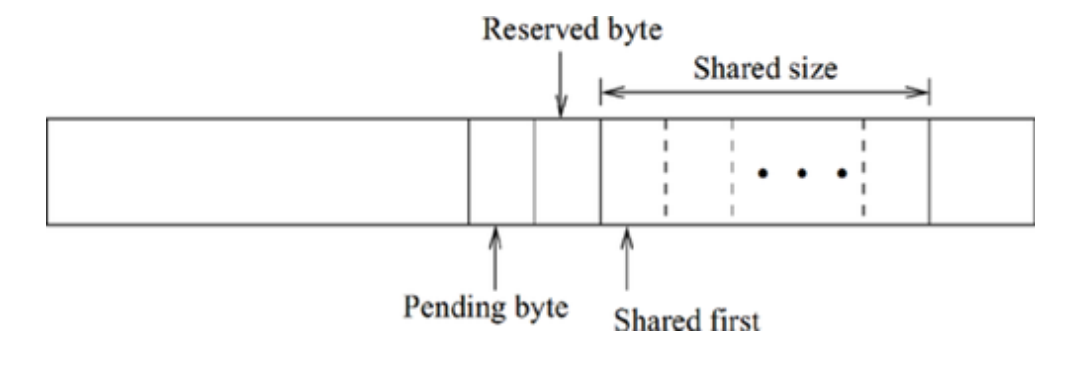
死锁是一个恼人的问题。解决死锁的方法有两种：1、预防；2、检测并去除。SQLite采取预防的方法。如果事务获取锁失败，该事务将进行有限次的尝试（尝试次数可以在主程序运行时设定，默认次数是1）。如果所有的尝试都失败了，SQLite返回SQLITE\_BUSY。主程序可以在稍候重试或者放弃事务。最终，系统中将不会出现死锁。无论如何，理论上，系统中会发生饥饿现象，即一个事务一直申请一个锁但是永远也获得不了这个锁。但是SQLite并不是一个企业级别的数据库，所以饥饿现象不是一个大问题。

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）

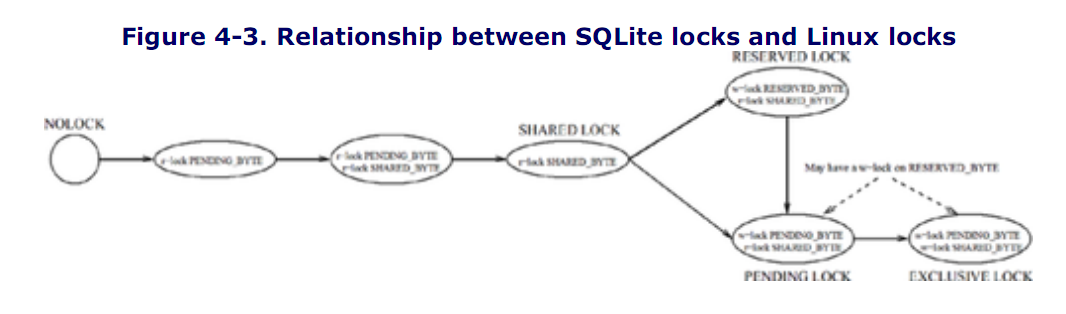
一个线程可能在获得SHARED LOCK之后仅仅想要获得一个PENDING LOCK。为了获得PENDING LOCK，需要在PENDING\_BYTE中获取一个写入锁。（这保证了没有新的SHARED LOCK可以被获取，但是已经存在的SHARED LOCK可以并存）注意线程并不释放自己的SHARED LOCK。（这保证了另一个线程不能获得EXCLUSIVE LOCK）

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。



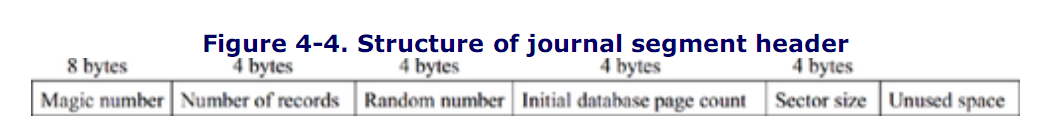
4.3 日志管理

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



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。

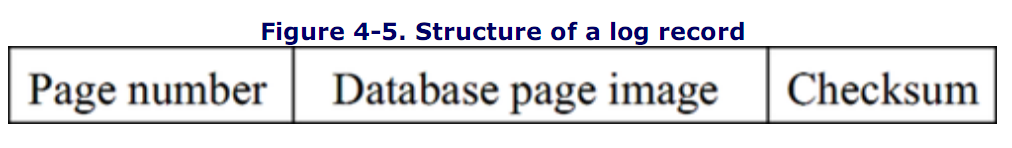
NOTE

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。

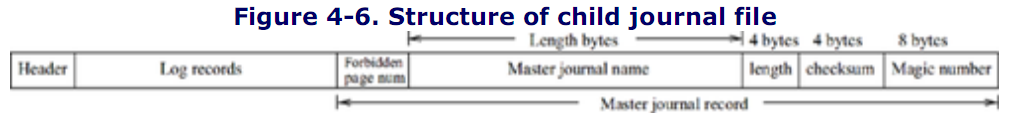
4.3.2 日志记录结构

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。



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。



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。

1. 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。

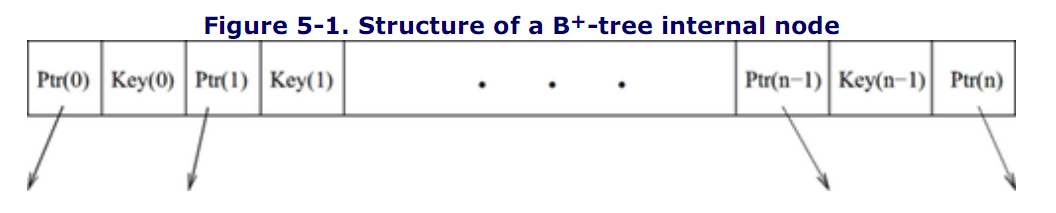
B+树是B树的一个变体，B+树把所有的数据项都存储在叶子节点中。数据项（键值和数据）通过键值排序，内部节点只保存搜索信息（键值）和子节点指针。中间节点中的键值按照顺序存储，这些键值用来指示搜索的方向。

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：

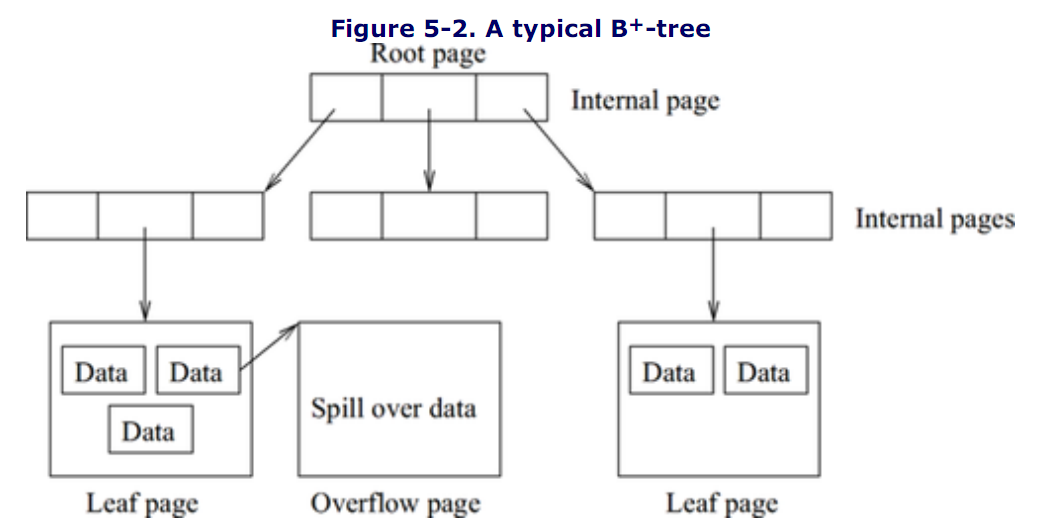
* Ptr(0)指向的子树的所有键值都小于或者等于key(0)
* Ptr(1)指向的子树的所有键值都小于或者等于key(1)且大于等于key(0)，以此类推
* Ptr(n)指向的子树的所有键值都大于或者等于key(n-1)

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。



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。



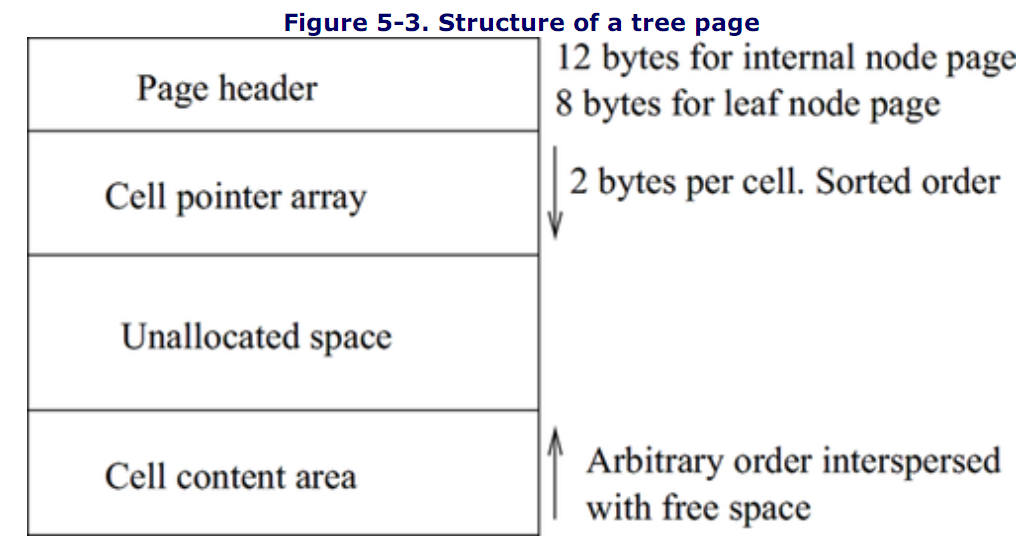
一个页中只能存储一个节点（内部节点或者叶子节点），这个页就被视为节点页（内部节点页或者叶子节点页）。对于每个节点，每个数据项的键值和数据合在一起作为负载（payload）。固定数量的预先设置好的负载直接存储在页中。如果负载大于规定的数量，超过的字节存储到溢出页（overflow page）：超过的负载按照顺序存储在溢出页链表中，内部节点也可以有溢出页。

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：



每个页被分为4个部分：

1.

页头

2.

Cell指针数组

3.

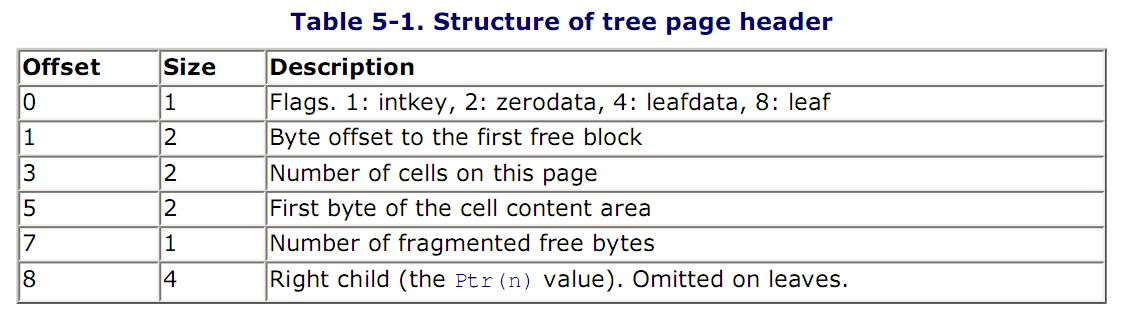
未分配空间

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。

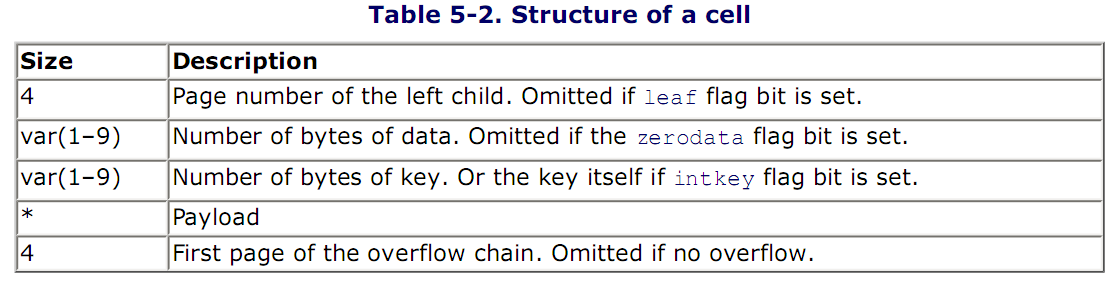


偏移量是0的flags定义了页的格式。如果leaf bit为真，这意味着该页是一个叶子节点并且没有子节点。如果zerodata为真，那么该页只包含键值而没有任何数据。如果intkey为真，那么键值是一个整型变量，并且该键值被保存在cell头中而不是复杂区域（见之后的章节）。如果leafdata为真，那么树只在叶子节点存储数据。对于内部节点页，页头也包含最右边的孩子节点指针（offset 8）。

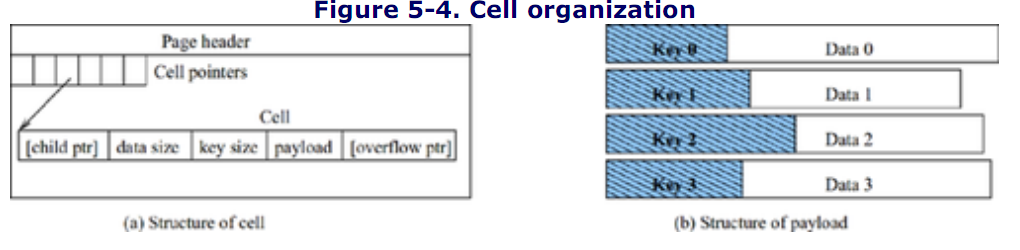
Cell被保存在页的最底部，向页开头的方向增长。Cell指针数组从页头之后的第一个byte开始，包含0个或者多个cell指针。每个cell指针是一个2byte整数，该整数指示了实际cell内容相对于页开头的偏移量。Cell指针按照相应的键值排序，尽管cell本身的存储不是按照顺序的。Cell指针数组的大小被存储在页头处偏移量为3的地方。

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。



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。



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。

1. 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）

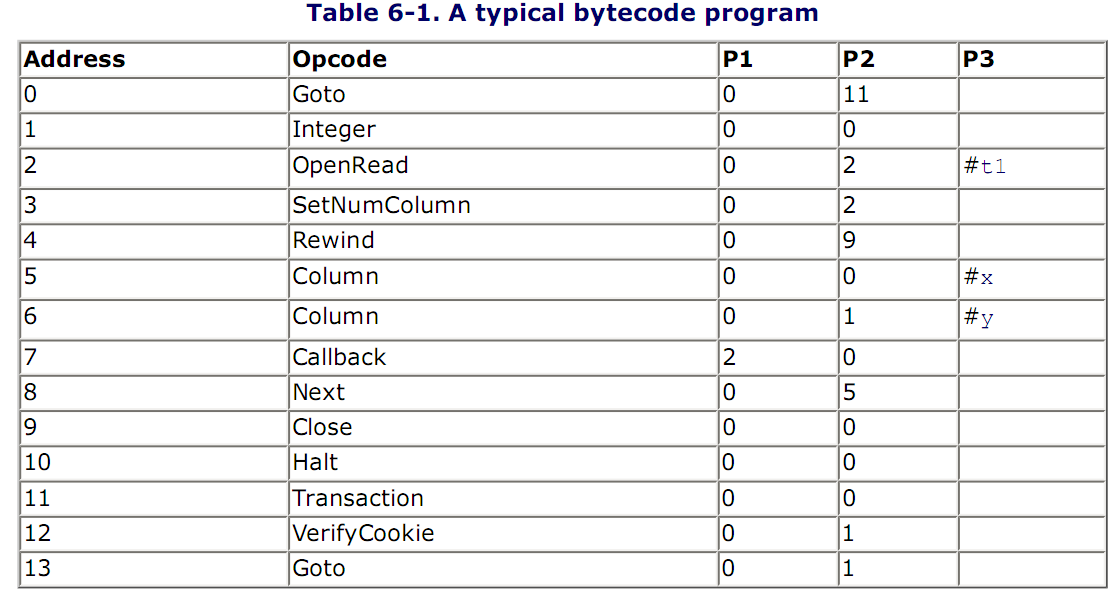
6.1 字节码编程语言

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。



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

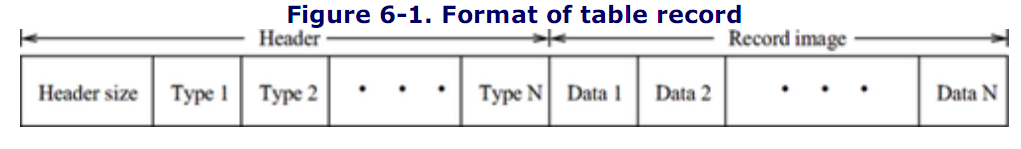
虚拟机把数据项存储在B/B+树中，每个数据项由键值和数据组成。只有虚拟机负责维护键值和数据的内部结构（尽管B+树模块可能把数据项分割到溢出页中，虚拟机把数据项视为逻辑上连续的字节字符串）。对于表和索引，SQLite使用两种相似但是不同的数据项格式。

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）

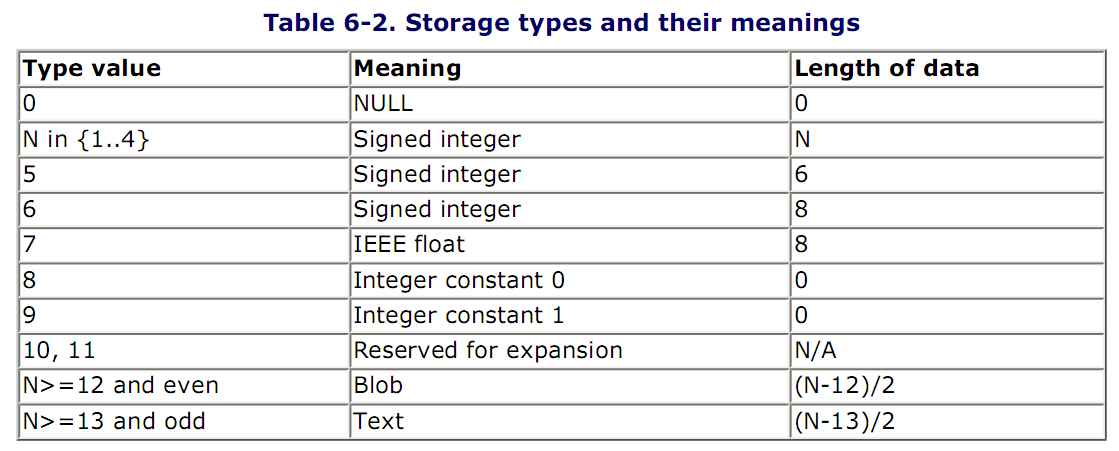
6.2.2 表数据项格式



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。



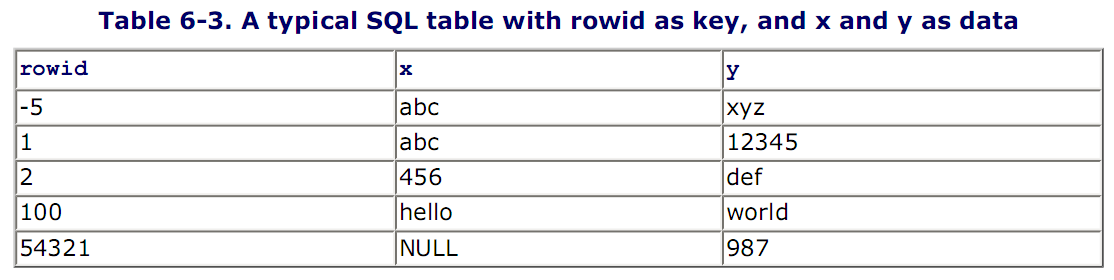
NULL类型就是SQL NULL。对于整型，数据值是有符号整型数，根据数据的大小分别存储在1，2，3，4，6，8bytes中。浮点型数据按照IEEE浮点数标准存储在8bytes中。类型编码8和9代表整型常数0和1.对于TEXT类型，使用默认的字符编码格式（UTF-8，UTF-16BE，UTF16-LE）存储在数据库文件中，对于UTF-16BE和UTF16-LE编码格式，分别是大端存储和小端存储的。（每个数据库文件只能存储同一种UTF格式）对于二进制大对象（BLOB），数据值是一个二进制块，按照输入原样存储。

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')。



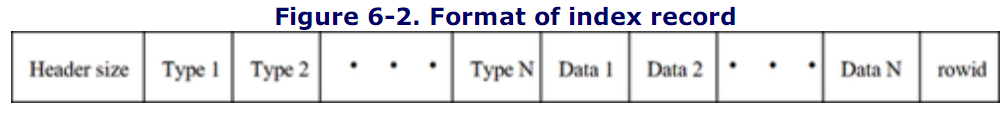
NOTE

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。

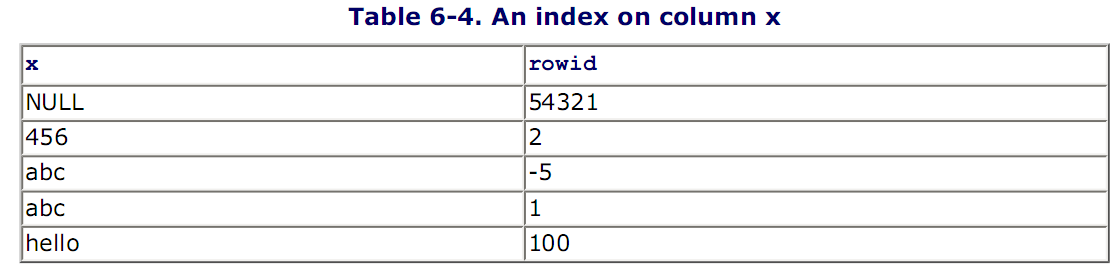
当行号作为数据的一部分被存储的时候，行号有一个内部整数类型。当作为一个键值存储的时候，行号是一个变长的哈夫曼编码。负数行号是允许的，但是需要用9bytes存储它们，所以并不推荐使用负数行号。SQLite添加的行号总是非负的，尽管你可以显式声明负数行号，上面的-5就是一个例子。

6.2.4 索引值格式

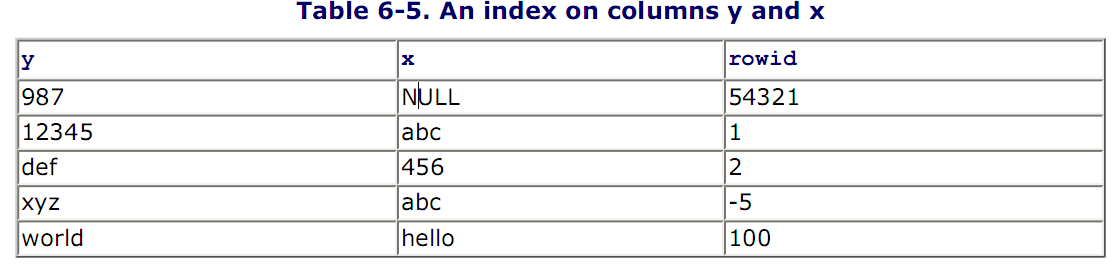
在前面的章节中，我们已经介绍了每个B+树的键值是一个整数，数据是表中的一行。索引刚好相反。对于一个索引，键值是该行索引的所有列的组合，数据是该行的行号。为了进入索引表中的一行，SQLite首先搜索索引表来找到相关的整数值，然后使用这个值来在B+树中查找完整的数据。



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：



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。



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）

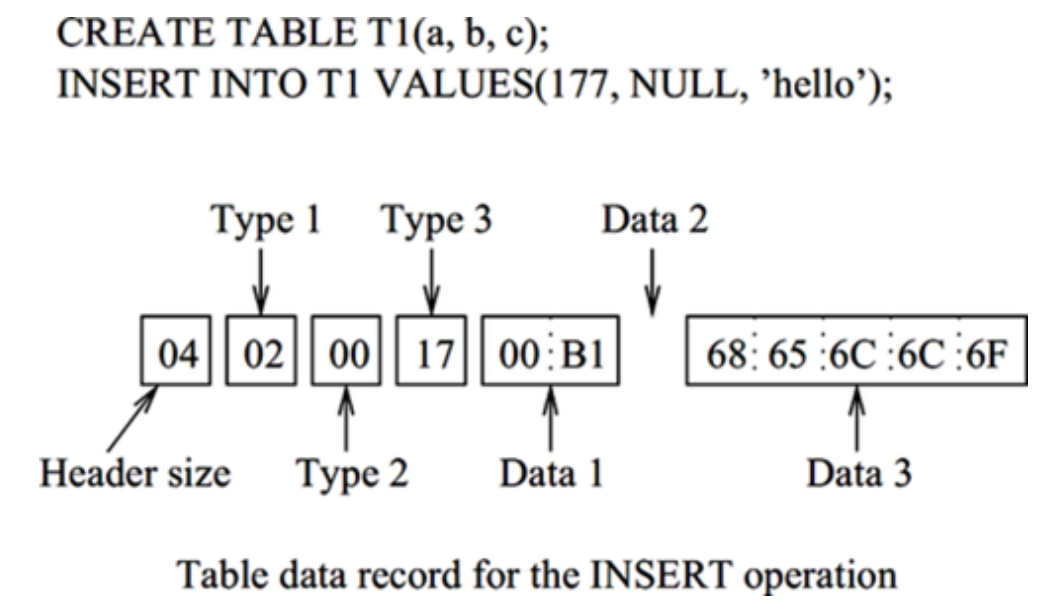
5.

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



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.

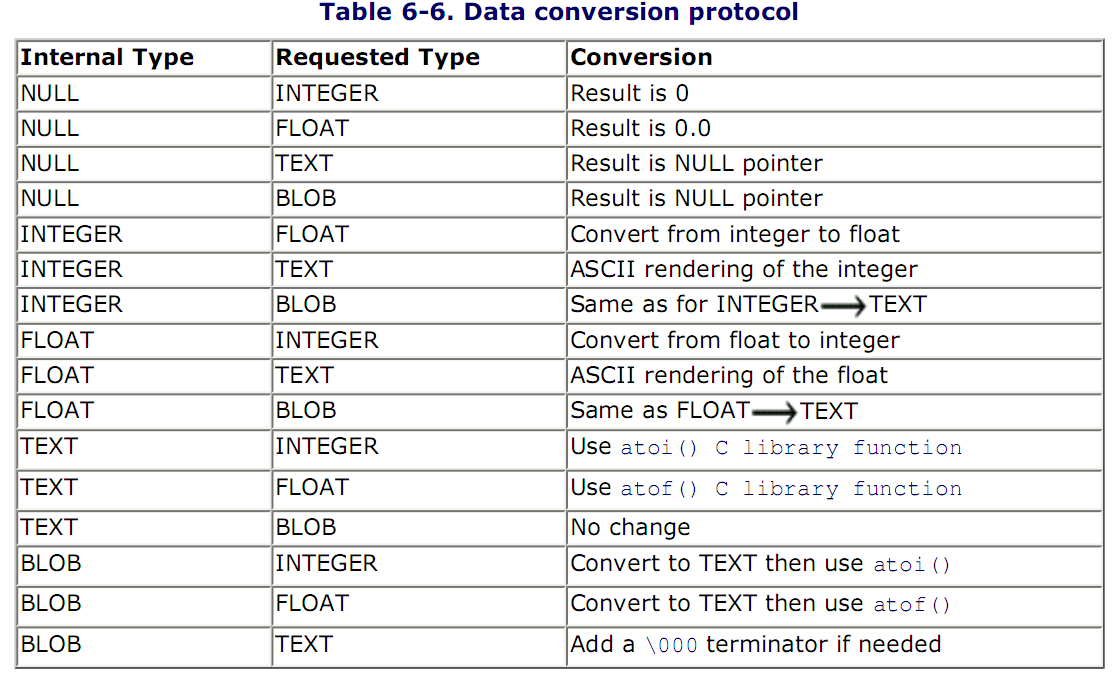
data2是NULL，不占用数据项中的任何byte。

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。



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。