SQLite 数据库文件格式全面分析

0 前言

性急的兄弟可以跳过前言直接看第 1 章，特别性急的兄弟可以跳过前面各章，直接看鸣谢。SQLite 数据库包括多方面的知识，比如 VDBE 什么的。据说那些东西会经常变。确实，我用的是 3.6.18 版，我看跟其它文档中描述的 3.3.6 的 VDBE 已经很不一样了。所以决定先写文件格式，只要是 3.?.?的版本，文件格式应该不会有太大变化吧。

网上介绍 SQLite 文件格式的文章并不少，但一般都是针对小文件：一个表，几条记录，两个页。本文准备一直分析到比较大的文件，至少 B-tree 和 B+tree 中得有内结点(就是说不能只有一个既是根又是叶的结点，就是说表中得多点记录，得建索引)，还要争取对 SQLite 的各类页都做出分析。

在分析的过程中，争取把 SQLite 数据库关于文件格式的基本规定也都介绍一下。这样，本文既是一个综合性的技术文档，又带有实例说明，兄弟们参考时岂不是就很方便了吗？

既然是技术文档，要想读懂总得先掌握点 SQLite 数据库的基本知识吧。所以，先介绍参考文献。

0.1 参考文献

1-The Definitive Guide to SQLite . Michael Owens：比较经典的 SQLite 著作。我边看边翻译了其中的部分内容，但翻得不好，大家还是看原文吧。

2-SQLite 源代码：有关 SQLite 的最原始说明都在源代码中了。先浏览一下代码还是很有收获的，特别是几个主要的.h 文件。有关文件格式的说明主要在 btreeInt.h 中。

3-SQLite 入门与分析 ：网上 Arrowcat 的系列文章。Arrowcat 应该是一个很博学的人，看他的文章收获很大，在此也算是鸣谢吧。

4-SQLite . Chris Newman：我没看，因为也是网上能够下载到的重要资源，所以列出。看目录内容应该比参考文献 1 简单一些，但出版日期也更早了一些。

5-NULL：在网上搜了半天，国内为什么就没有关于 SQLite 的好书呢？

6- http://www.sqlite.org/fileformat.html：如果这篇文章看懂了，其实我这篇东西根本就不用再看了。这是介绍 SQLite 文件格式的权威文档，列在最后，是因为我也是写完这篇东西后才看到的。该文档由 SQLite 官方网站提供，当初没看，一是因为上网少，还没仔细浏览人家的网站就开始干了(太激动)，其实归根结蒂还是因为英语不好。看到此文档这后还敢把我的东西发出来，有两个原因：一、为其他英语比我强不了多少的兄弟提供一点方便，二、我这里有例子，看起来更形象一些吧。

0.2 术语

本文涉及的绝大多数术语都是在出现时再进行简单解释，但还是有个别概念需要先说明清楚，比如：

(1) Btree、B-tree 和 B+tree：

Btree 是为磁盘存储而优化了的一种树结构，其一般性说明可参考各类《数据结构》教材。根据实现方法的不同，Btree 又分为很多类型。在 SQLite 中，存储表数据用的是 B+tree，存储表索引用的是 B-tree。由于历史原因，SQLite 在 3.0 版以前只使用 B-tree，从 3.0 版开始，才对表数据使用了 B+tree。因此，在 SQLite 的官方文档中，有时 B-tree 表示存储表索引的 B-tree，有时又是两种 Btree 的统称。本文中将两种 Btree 的概念加以了区分，而将 Btree 作为两种树的统称，这是与 SQLite 官方文档及当前大多数 SQLite 介绍性文档相区别的地方。

(2) auto-vacuum 数据库：

一般情况下，当一个事务从数据库中删除了数据并提交后，数据库文件的大小保持不变。即使整页的数据都被删除，该页也会变成“空闲页”等待再次被使用，而不会实际地被从数据库文件中删除。执行 vacuum 操作，可以通过重建数据库文件来清除数据库内所有的未用空间，使数据库文件变小。但是，如果一个数据库在创建时被指定为 auto_vacuum 数据库，当删除事务提交时，数据库文件会自动缩小。使用 auto_vacuum 数据库可以节省空间，但却会增加数据库操作的时间，有利有弊。Auto_vacuum 数据库需要使用附加的格式，如指针图页（本文第 6 章有介绍），本文重点讨论非 auto_vacuum 数据库。

(3) 数据库映像、数据库文件和日志文件：

“数据库映像”是 SQLite 数据库的磁盘映像。SQLite 数据库存储在单一的“数据库文件”中。一般情况下，数据库映像和数据库文件是一致的，可以理解为数据库映像就是数据库文件的内容，但有例外。如果事务对数据库进行了修改，这些修改会暂存在“日志文件”中，此时可以认为数据库映像分布在数据库文件和日志文件两个文件中。日志文件有自己的格式，本文第 7 章专门介绍。

(4) SQLite 的当前版本：

我开始写这篇东西时，SQLite 的当前版本为 3.6.18。现在已经变成 3.6.20 了，文件格式没变。

1 小文件的分析

1.1 准备数据库

执行 SQLite 的命令行工具，创建一个新的数据库 food_test.db。

```
D:\SQLite\CLP>sqlite3 foods_test.db
```

创建一个新表。

```
CREATE TABLE foods(
   id integer primary key,
   type_id integer,
   name text );
```

插入 2 条记录。

```
INSERT INTO "foods" VALUES(1, 1, 'Bagels');
INSERT INTO "foods" VALUES(2, 1, 'Bagels, raisin');
```

退出命令行工具。

当前目录下多了一个文件 foods_test.db，大小为 2K。

现在，就可以用 UltraEdit 或 WinHex 之类的软件对其进行分析了。

1.2 SQLite 文件格式

SQLite 有 3 类数据库。除内存数据库外，SQLite 把每个数据库(main 或 temp)都存储到一个单独的文件中。

SQLite 数据库文件由固定大小的“页(page)”组成。页的大小可以在 512~32768 之间(包含这两个值，必须是 2 的指数)，默认大小为 1024 个字节(1KB)。页大小可以在数据库刚刚创建时设置，一旦创建了数据库对象之后，这个值就不能再改变了。

数据库中所有的页从 1 开始顺序编号。在具体的实现中，页号用 4 字节来表示，并限制最大页号不得超过 2^31(参 pager.c)。文件的第 1 个页被称为 page 1，第 2 个页被称为 page 2，依此类推。编号为 0 的页表示“无此页”。

注：关于一个数据库文件中可以有多少个页，SQLite 是这样实现的：为了限制数据库文件的大小（不要太大），SQLite 在程序中对文件页数进行了限制，文件页数的最大值默认为 1073741823，在 sqliteLimit.h 中定义，可以在运行时改变。

页的类型可以是：Btree 页、空闲(free)页或溢出(overflow)页。Btree 又可以是 B-tree 或 B+tree，每一种树的结点又区分为内部页和叶子页。一个数据库文件中可能没有空闲页或溢出页，但必然有 Btree 页。关于 Btree 页的格式规定是 SQLite 数据库的核心内容，本文的前半部分都是在介绍 Btree 页。

注：其实 SQLite 还有两种页。一种称为“锁页(locking page)”。只有 1 页，位于数据库文件偏移为 1G 开始的地方，如果文件不足 1G，就没有此页。该页是用于文件加锁的区域，不能存储数据(参源代码 io.h)。好在，如果只是读数据，即使文件大于 1G，也不会有指针指向此页，因此下面我们就不再提它了。另一种称为 “指针位图页(pointer-map page)”，这类页用于在 auto-vacuum 的数据库中存储元数据。本文不涉及 auto-vacuum 数据库，也就不讨论这种页了。

从逻辑上来说，一个 SQLite 数据库文件由多个多重 Btree 构成。每个 Btree 存储一个表的数据或一个表的索引，索引采用 B-tree，而表数据采用 B+tree，每个 Btree 占用至少一个完整的页，每个页是 Btree 的一个结点。每个表或索引的第 1 个页称为根页，所有表或索引的根页编号都存储在系统表 sqlite_master 中，表 sqlite_master 的根页为 page 1。

注：sqlite_master 是一个系统表，保存了数据库的 schema 信息，详参“关于 sqlite_master 表”一节。

数据库中第一个页(page 1)永远是 Btree 页。Page 1 的前 100 个字节是一个对数据库文件进行描述的“文件头”。它包括数据库的版本、格式的版本、页大小、编码等所有创建数据库时设置的永久性参数。关于这个特殊文件头的文档在 btreeInt.h 中，具体格式如下：

偏移量	大小	说明
0	16	头字符串，如果不改源程序，此字符串永远是"SQLite format 3"。
16	2	页大小(以字节为单位)。
18	1	文件格式版本(写)。对于 SQLite 的当前版本，此值为 1。如果该值大于 1，表示文件为只读。SQLite 将来版本对此域的规定可能改变。
19	1	文件格式版本(读)。对于 SQLite 的当前版本，此值为 1。如果该值大于 1，SQLite 认为文件格式错，拒绝打开此文件。SQLite 将来版本对此域的规定可能改变。
20	1	每页尾部保留空间的大小。(留作它用，默认为 0。)

21	1	Btree 内部页中一个单元最多能够使用的空间。 255 意味着 100%，默认值为 0x40，即 64(25%)，这保证了一个结点(页)至少有 4 个单元。
22	1	Btree 内部页中一个单元使用空间的最小值。默认值为 0x20，即 32(12.5%)。
23	1	Btree 叶子页中一个单元使用空间的最小值。默认值为 0x20，即 32(12.5%)。 注：SQLite 的当前版本规定 21~23 的 3 个字节值只能是 0X402020。原来这 3 个字节值是可变的，从 3.6.0 版开始被固定下来了。
24	4	文件修改计数，通常被事务使用，由事务增加其值。SQLite 用此域的值验证内存缓冲区中数据的有效性。
28	4	未使用。
32	4	空闲页链表首指针。参“空闲页”一节。
36	4	文件内空闲页的数量。
40	60	15 个 4 字节的元数据变量。

从偏移 40 开始的 15 个 4 字节元数据变量在 btreeInt.h 中的定义如下：

40　　4　　Schema 版本：每次 schema 改变(创建或删除表、索引、视图或触发器等对象，造成 sqlite_master 表被修改)时，此值+1。

44　　4　　File format of schema layer：当前允许值为 1~4，超过此范围，将被认为是文件格式错。

48　　4　　Size of page cache。

52　　4　　Largest root-page (auto/incr_vacuum)：对于 auto-vacuum 数据库，此域为数据库中根页编号的最大值，非 0。对于非 auto-vacuum 数据库，此域值为 0。

56　　4　　1=UTF-8、2=UTF16le、3=UTF16be。

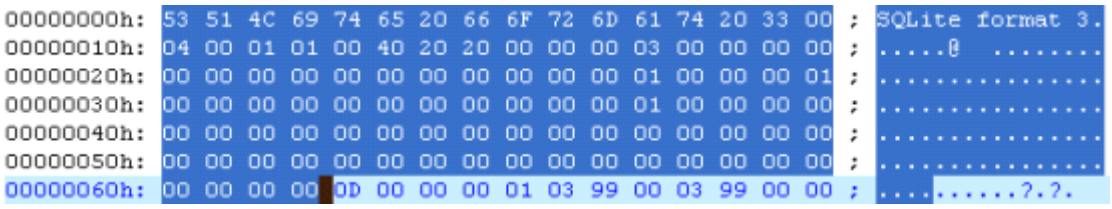
60　　4　　User version。此域值供用户应用程序自由存取，其含义也由用户定义。

64　　4　　Incremental vacuum mode：对于 auto-vacuum 数据库，如果是 Incremental vacuum 模式，此域值为 1。否则，此域值为 0。

68　　4　　未使用。

72　　4　　未使用。

用 UltraEdit 打开文件 foods_test.db，page 1 在 0X0000~0X03FF。其中文件头内容如下(深蓝色部分)：



前 16 个字节为头字符串，程序中固定设为"SQLite format 3"。

0X0400：页大小，0X0400=1024 字节。

0X01：文件格式版本(写)，值为 1。

0X01：文件格式版本(读)，值为 1。

0X40：Btree 内部页中一个单元最多能够使用的空间。0X40=64，即 25%。

0X20：Btree 内部页中一个单元使用空间的最小值。0X20=32，即 12.5%。

0X20：Btree 叶子页中一个单元使用空间的最小值。0X20=32，即 12.5%。

0X00000003：文件修改计数，现在已经修改了 3 次，分别是 1 次创建和两次插入。

从 0X20 开始的 4 个字节：空闲页链表首指针。当前值为 0，表示该链表为空。

从 0X24 开始的 4 个字节：文件内空闲页的数量。当前值为 0。
从 0X28 开始的 4 个字节：Schema version。当前值为 0X00000001。以后，每次 sqlite_master 表被修改时，此值+1。
从 0X38 开始的 4 个字节：采用的字符编码。此处为 0X00000001，表示采用的是 UTF-8 编码。
注意：在 SQLite 文件中，所有的整数都采用大端格式，即高位字节在前。

1.3 Btree 页格式介绍

1.3.1 Btree 页的分区

页的类型有 Btree 页、空闲页和溢出页，本文前 3 章介绍的都是 Btree 页，其他类型的页在第 4、5 章介绍。
每个 Btree 页由四个部分构成：

1．页头
2．单元指针数组
3．未分配空间
4．单元内容区

首先介绍“单元”的概念：Btree 页内部以单元(cell)为单位来组织数据，一个单元包含一个(或部分，当使用溢出页时)payload(也称为 Btree 记录)。由于各类数据大小各不相同，每个单元的大小也就是可变的，所以 Btree 页内部的空间需要进行动态分配(程序内部动态分配，不是动态申请空间)，单元是 Btree 页内部进行空间分配和回收的基本单位。
页内所有单元的内容集中在页的底部，称为“单元内容区”，由下向上增长。由于单元的大小可变，因此需要对每个单元在页内的起始位置(称为单元指针)进行记录。单元指针保存在单元指针数组中，位于页头之后。单元指针数组包含 0 个或多个指针，由上向下增长。
单元指针数组和单元内容区相向增长，中间部分为未分配空间。系统尽量保证未分配空间位于最后的指针之后，这样，就很容易增加新的单元，而不需要整理碎片。
单元不需要是相邻和有序的，但单元指针是相邻和有序的。每个指针占 2 个字节，表示该单元在单元内容区中距页开始处的偏移。页中单元的数量保存在页头中。

1.3.2 页头格式

页头包含用来管理页的信息，它通常位于页的开始处。对于数据库文件的 page 1，页头始于第 100 个字节处，因为前 100 个字节是文件头(file header)。
页头的格式如下：

偏移量	大小	说明
0	1	页类型标志。1: intkey, 2: zerodata, 4: leafdata, 8: leaf。

1	2	第 1 个自由块的偏移量。
3	2	本页的单元数。
5	2	单元内容区的起始地址。
7	1	碎片的字节数。
8	4	最右儿子的页号(the Ptr(n) value)。仅内部页有此域。

下面对页头各域分别进行介绍。

页类型标志：

如果 leaf 位被设置，则该页是一个叶子页，没有儿子；

如果 zerodata 位被设置，则该页只有关键字，而没有数据；

如果 intkey 位被设置，则关键字是整型；

如果 leafdata 位设置，则 tree 只存储数据在叶子页。

注：

以上内容见于大多数 SQLite 介绍性文档，btreeInt.h 中也这么说。但通过分析程序代码，并从参考文献 6 中得到确认，结论如下：

上述描述与实际实现是矛盾的。可以这样理解：就不用管各标志位的含义了，如果是 B+tree 的叶子页，该字节值为 0X0D，如果是 B+tree 的内部页，该字节值为 0X05，如果是 B-tree 的叶子页，该字节值为 0X0A，如果是 B-tree 的内部页，该字节值为 0X02。由此可见：intkey 标志倒是可以作为判断 B+tree 树和 B-tree 的标志（置 1 为 B+tree 树），程序中实际也是这样应用的。

第 1 个自由块的偏移量：

由于随机地插入和删除单元，将会导致一个页上单元和空闲区域互相交错。单元内容区域中没有使用的空间收集起来形成一个空闲块链表，这些空闲块按照它们地址的升序排列。页头偏移为 1 的 2 个字节指向空闲块链表的头。每个空闲块至少 4 个字节，因为一个空闲块的开始 4 个字节存储控制信息：前 2 个字节指向下一个空闲块(0 意味着没有下一个空闲块了)，后 2 个字节为该空闲块的大小。

碎片的字节数：

由于空闲块大小至少为 4 个字节，所以单元内容区中的 3 个字节或更小的自由空间(称为碎片，fragment)不能存在于空闲块列表中。所有碎片的总的字节数将记录在页头偏移为 7 的位置(碎片最多为 255 个字节，在它达到最大值之前，页会被整理)。

单元内容区的起始地址：

单元内容区的起始地址记录在页头偏移为 5 的地方。这个值为单元内容区域和未使用区域的分界线。

最右儿子的页号：

如果本 Btree 页是叶子页，则无此域，页头长为 8 个字节。如果本 Btree 页为内部页，则有此域，页头长为 12 个字节。页头偏移为 8 的 4 个字节包含指向最右儿子的指针，该指针的含义将在第 2 章介绍。

有关 Btree 页格式的其它规定，将在下一节中用到时再介绍。

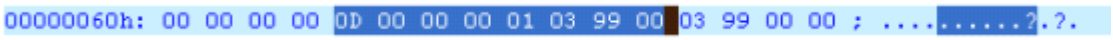
1.4 Page 1 格式分析

Btree 的基本原理这里就不详细介绍了。Btree 有多种实现方法，各类《数据结构》教材中的介绍也各不相同，但原理大同小异，随便找一本参考一下吧。最简单的 Btree 只有一个结点，既是根页，也是叶子页。

当前 foods_test.db(大小为 2K)只有两个页，都是 Btree 页。每个页都是一个 Btree(B+tree，因为存储的是表数据)，都是上述的单结点 Btree。其中 page 1 为系统表 sqlite_master 的根页，下面我们对该页进行详细分析。

1.4.1 页头分析

该页的页头从 0X64=100 处开始(前面 100 个字节是文件头)，8 个字节(因为是叶子页)。如下图中深蓝色部分所示：



说明：
0X0D：说明该页为 B+tree 的叶子结点。
0X0000：第 1 个自由块的偏移量。值为 0，说明当前自由块链表为空。
0X0001：本页的单元数。当前 sqlite_master 表中只有一条记录，所以本页当前只有 1 个单元。
0X0399：单元内容区的起始地址。
0X00：碎片的字节数。当前值为 0。

1.4.2 单元指针数组

单元指针数组在页头之后，当前只有一个指针，为 0X0399。

1.4.3 关于可变长整数

可变长整数是 SQLite 的特色之一，使用它既可以处理大整数，又可以节省存储空间。由于单元中大量使用可变长整数，故在此先加以介绍。
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。
下面是一些可变长整数的例子：

0x00	转换为	0x00000000
0x7f	转换为	0x0000007f
0x81 0x00	转换为	0x00000080
0x82 0x00	转换为	0x00000100
0x80 0x7f	转换为	0x0000007f
0x8a 0x91 0xd1 0xac 0x78	转换为	0x12345678
0x81 0x81 0x81 0x81 0x01	转换为	0x10204081

可变长整数可用于存储 rowid、字段的字节数或 Btree 单元中的数据。

1.4.4 关于 sqlite_master 表

sqlite_master 是一个系统表，保存了数据库的 schema 信息。在逻辑上 sqlite_master 包含 5 个

字段，如下表所示：

编号	字段	说明
1	type	值为"table"、 "index"、 "trigger"或"view"之一。
2	name	对象名称，值为字符串。
3	tbl_name	如果是表或视图对象，此字段值与字段 2 相同。如果是索引或触发器对象，此字段值为与其相关的表名。
4	rootpage	对触发器或视图对象，此字段值为 0。对表或索引对象，此字段值为其根页的编号。
5	SQL	字符串，创建此对象时所使用的 SQL 语句。

1.4.5 B+tree 叶子页的单元格式

单元是变长的字节串。一个单元包含一个(或部分，当使用溢出页时)payload。B+tree 叶子页单元的结构如下：

大小	说明
var(1–9)	Payload 大小，以字节为单位。
var(1–9)	数据库记录的 Rowid 值。
*	Payload 内容，存储数据库中某个表一条记录的数据。
4	溢出页链表中第 1 个溢出页的页号。如果没有溢出页，无此域。

结合实例来说明吧。

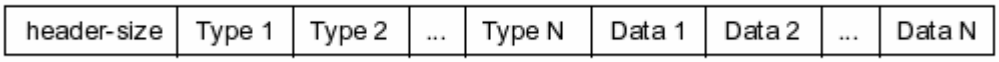
当前的单元内容区中只有一个单元，从 0X0399 开始，内容如下图所示：

```
00000390h: 00 00 00 00 00 00 00 00 00 65 01 07 17 17 17 01 ; .........e......
000003a0h: 81 29 74 61 62 6C 65 66 6F 6F 64 73 66 6F 6F 64 ; ?tablefoodsfood
000003b0h: 73 02 43 52 45 41 54 45 20 54 41 42 4C 45 20 66 ; s.CREATE TABLE f
000003c0h: 6F 6F 64 73 28 0A 20 20 69 64 20 69 6E 74 65 67 ; oods(.  id integ
000003d0h: 65 72 20 70 72 69 6D 61 72 79 20 6B 65 79 2C 0A ; er primary key,.
000003e0h: 20 20 74 79 70 65 5F 69 64 20 69 6E 74 65 67 65 ;   type_id intege
000003f0h: 72 2C 0A 20 20 6E 61 6D 65 20 74 65 78 74 20 29 ; r,.  name text )
```

0X65：Payload 数据的字节数。可以看出 Payload 数据是从 07 17 ~20 29。

0X01：foods(table 对象)在 sqlite_master 表中对应记录的 rowid，值为 0X01。

Payload 的格式如下图所示：



每个 payload 由两部分组成。第 1 部分是记录头，由 N+1 个可变长整数组成，N 为记录中的字段数。第 1 个可变长整数(header-size)的值为记录头的字节数。跟着的 N 个可变长整数与记录的各字段一一对应，表示各字段的数据类型和宽度。用可变长整数表示各字段类型和宽度的规定如下表所示：

类型值	含义	数据宽度(字节数)
0	NULL	0
N in 1..4	有符号整数	N
5	有符号整数	6
6	有符号整数	8
7	IEEE 符点数	8
8-11	未使用	N/A
N>12 的偶数	BLOB	(N-12)/2

N>13 的奇数	TEXT	(N-13)/2

header-size 的值包括 header-size 本身的字节和 Type1~TypeN 的字节。
Data1~DataN 为各字段数据，与 Type1~TypeN 一一对应，类型和宽度由 Type1~TypeN 指定。
本例的 payload 数据为：
0X07：记录头包括 7 个字节。
0X17：字段 1。TEXT，长度为：(23-13)/2=5。值为：table。
0X17：字段 2。TEXT，长度为：(23-13)/2=5。值为：foods。
0X17：字段 3。TEXT，长度为：(23-13)/2=5。值为：foods。
0X01：字段 4。整数，长度为 1。值为：0X02。表示本表 B+tree 的根页编号为 2。
0X8129：字段 5。TEXT。0X8129 为可变长整数，转换为定长为 0XA9=169。可知字段长度为：(169-13)/2=78=0X4E。对应数据为下图中蓝色部分。

```
00000390h: 45 20 4E 4F 43 41 53 45 29 65 01 07 17 17 17 01 ; E NOCASE)e.....
000003a0h: 81 29 74 61 62 6C 65 66 6F 6F 64 73 66 6F 6F 64 ; ?tablefoodsfood
000003b0h: 73 02 43 52 45 41 54 45 20 54 41 42 4C 45 20 66 ; s.CREATE TABLE f
000003c0h: 6F 6F 64 73 28 0A 20 20 69 64 20 69 6E 74 65 67 ; oods(.  id integ
000003d0h: 65 72 20 70 72 69 6D 61 72 79 20 6B 65 79 2C 0A ; er primary key,.
000003e0h: 20 20 74 79 70 65 5F 69 64 20 69 6E 74 65 67 65 ;   type_id intege
000003f0h: 72 2C 0A 20 20 6E 61 6D 65 20 74 65 78 74 20 29 ; r,.  name text )
```

1.5 Page 2 格式分析

Page 2 为表 foods 的根页。foods 的 Btree 只有一个结点，既是根页，也是叶子页。
有了 page 1 的分析经验，page 2 就好分析了。作为对上一节内容的巩固，再简单分析一下吧。
用 UltraEdit 打开文件 foods_test.db，page 2 在 0X0400~0X07FF。本页不再是文件首页(没有文件头)，所以页头起始于本页的开始处，其内容如下(深蓝色部分)：

```
00000400h: 0D 00 00 00 02 03 DE 00 03 F3 03 DE 00 00 00 00 ; ......?.??...
```

因为是 Btree 的叶子页，所以页头只有 8 个字节。说明：
0X0D：说明该页为 B+tree 的叶子结点。
0X0000：第 1 个自由块的偏移量。0，说明当前自由块链表为空。
0X0002：本页的单元数。当前 foods 表中只有 2 条记录。
0X03DE：单元内容区的起始地址。
0X00：碎片的字节数。当前为 0。

两个单元的指针分别为 0X03F3 和 0X03DE。由于单元指针按次序排列，所以指针 0X03F3 指向本表的第 1 条记录，我们选择它进行分析。
0X03F3 指向的单元内容如下图所示(深蓝色部分)：

```
000007d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 13 02 ; ................
000007e0h: 04 00 01 29 01 42 61 67 65 6C 73 2C 20 72 61 69 ; ...).Bagels, rai
000007f0h: 73 69 6E 0B 01 04 00 01 19 01 42 61 67 65 6C 73 ; sin.......Bagels
```

说明：
0X0B：Payload 数据的字节数。可以看出 Payload 数据是从 04 00 ~6C 73，共 11 个字节。
0X01：本记录 rowid 的值为 1。
Payload 数据：
0X04：记录头包括 4 个字节。
0X00：字段 1。类型为 NULL(详细说明参“关于自增长的主键字段”一小节)。
0X01：字段 2。1 字节整数。值为 01，表示本记录 type_id 字段的值为 01。

0X19：字段 3。TEXT，长度为：(25-13)/2=6。值为：Bagels。
按此方法，第 2 条记录所对应的单元应该很容易分析了吧。

1.5.1 关于自增长的主键字段

SQLite 规定：SQLite 会对所有整型主键字段应用自动增长属性。
个人分析，还没找到权威说明：如果一个表具有整型主键字段，如表 foods 具有主键字段 id，则该表的 id 字段值即为其 rowid 值。
对上述字段 1 的进一步说明是：
由于 rowid 值已经在单元头部保存了，所以将字段 1 的类型设为 NULL，这样既节省空间，又容易保持数据一致性。
以上内容可以用以下语句创建一个新的数据库来验证。但请等看完下一章后再来验证这些内容吧，因为现在有关索引的内容还没介绍呢。

```
CREATE TABLE foods(
   id integer primary key,
   type_id integer,
   name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
INSERT INTO "foods" VALUES(1, 1, 'Bagels');
INSERT INTO "foods" VALUES(2, 1, 'Bagels, raisin');
INSERT INTO "foods" VALUES(40, 1, 'Bavarian Cream Pie');
INSERT INTO "foods" VALUES(30, 1, 'Bear Claws');
INSERT INTO "foods" (type_id,name) VALUES(1, 'Black and White cookies');
```

当然，如果主键字段为非整数，则 rowid 与主键字段值分别存储，则不存在上述问题，可用以下语句创建一个新的数据库来验证。

```
CREATE TABLE test(
   id text primary key,
   name text );
INSERT INTO test VALUES('aaa', 'Bagels');
INSERT INTO test VALUES('bbb', 'Bagels, raisin');
CREATE INDEX test_name_idx  on test (name COLLATE NOCASE);
```

2 “大”文件的分析

前面分析的数据库文件只有 2 个页，每个页自成一个 Btree，每个 Btree 只有一个结点，既是根页，也是叶子页。所以，至此，我们还没见过 Btree 的内部页呢。

2.1 准备数据库

向表 foods 继续插入记录。

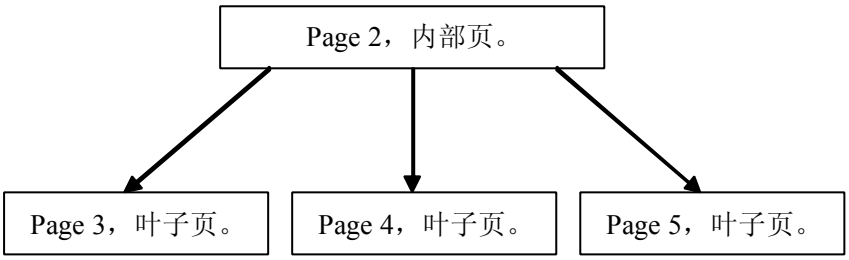
```
INSERT INTO "foods" VALUES(3, 1, 'Bavarian Cream Pie');
```

```
INSERT INTO "foods" VALUES(4, 1, 'Bear Claws');
INSERT INTO "foods" VALUES(5, 1, 'Black and White cookies');
INSERT INTO "foods" VALUES(6, 1, 'Bread (with nuts)');
INSERT INTO "foods" VALUES(7, 1, 'Butterfingers');
INSERT INTO "foods" VALUES(8, 1, 'Carrot Cake');
INSERT INTO "foods" VALUES(9, 1, 'Chips Ahoy Cookies');
INSERT INTO "foods" VALUES(10, 1, 'Chocolate Bobka');
INSERT INTO "foods" VALUES(11, 1, 'Chocolate Eclairs');
INSERT INTO "foods" VALUES(12, 1, 'Chocolate Cream Pie');
INSERT INTO "foods" VALUES(13, 1, 'Cinnamon Bobka');
INSERT INTO "foods" VALUES(14, 1, 'Cinnamon Swirls');
INSERT INTO "foods" VALUES(15, 1, 'Cookie');
INSERT INTO "foods" VALUES(16, 1, 'Crackers');
INSERT INTO "foods" VALUES(17, 1, 'Cupcake');
INSERT INTO "foods" VALUES(18, 1, 'Cupcakes');
INSERT INTO "foods" VALUES(19, 1, 'Devils Food Cake');
INSERT INTO "foods" VALUES(20, 1, 'Dinky Donuts');
INSERT INTO "foods" VALUES(21, 1, 'Dog biscuits');
INSERT INTO "foods" VALUES(22, 1, 'Donuts');
INSERT INTO "foods" VALUES(23, 1, 'Drakes Coffee Cakes');
INSERT INTO "foods" VALUES(24, 1, 'Entenmann''s Cake');
INSERT INTO "foods" VALUES(25, 1, 'Kaiser Rolls');
INSERT INTO "foods" VALUES(26, 1, 'Marble Ryes');
INSERT INTO "foods" VALUES(27, 1, 'Mini Ritz');
INSERT INTO "foods" VALUES(28, 1, 'Muffin');
INSERT INTO "foods" VALUES(29, 1, 'Muffin Tops');
INSERT INTO "foods" VALUES(30, 1, 'Muffin Stumps');
INSERT INTO "foods" VALUES(31, 1, 'Nut Bread');
INSERT INTO "foods" VALUES(32, 1, 'Pastries (of the Gods)');
INSERT INTO "foods" VALUES(33, 1, 'Peach Muffin');
INSERT INTO "foods" VALUES(34, 1, 'Peppridge Farms Cookies (Milanos)');
INSERT INTO "foods" VALUES(35, 1, 'Pizza Bagels');
INSERT INTO "foods" VALUES(36, 1, 'Pie');
INSERT INTO "foods" VALUES(37, 1, 'Pie (blueberry)');
INSERT INTO "foods" VALUES(38, 1, 'Pie (Blackberry) Pie');
INSERT INTO "foods" VALUES(39, 1, 'Pie (Boysenberry)');
INSERT INTO "foods" VALUES(40, 1, 'Pie (Huckleberry)');
INSERT INTO "foods" VALUES(41, 1, 'Pie (Raspberry)');
INSERT INTO "foods" VALUES(42, 1, 'Pie (Strawberry)');
INSERT INTO "foods" VALUES(43, 1, 'Pie (Cranberry)');
INSERT INTO "foods" VALUES(44, 1, 'Pie (Peach)');
INSERT INTO "foods" VALUES(45, 1, 'Poppy Seed Muffins');
INSERT INTO "foods" VALUES(46, 1, 'Triscuits');
INSERT INTO "foods" VALUES(47, 1, 'Wedding Cake (Royal)');
```

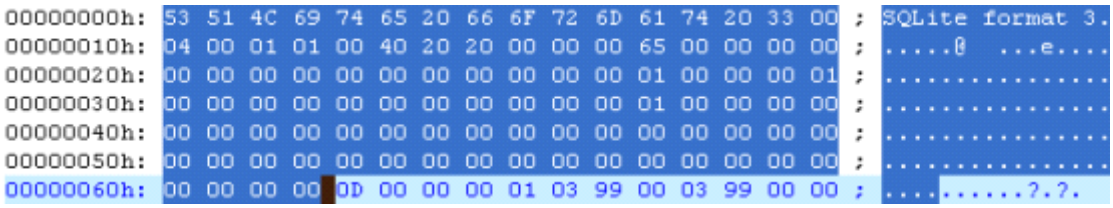
```
INSERT INTO "foods" VALUES(48, 2, 'Bran');
INSERT INTO "foods" VALUES(49, 2, 'Cheerios');
INSERT INTO "foods" VALUES(50, 2, 'Corn Flakes');
INSERT INTO "foods" VALUES(51, 2, 'Double Crunch');
INSERT INTO "foods" VALUES(52, 2, 'Fruit Loops');
INSERT INTO "foods" VALUES(53, 2, 'Grape Nuts');
INSERT INTO "foods" VALUES(54, 2, 'Honey Combs');
INSERT INTO "foods" VALUES(55, 2, 'Kasha');
INSERT INTO "foods" VALUES(56, 2, 'Kix');
INSERT INTO "foods" VALUES(57, 2, 'Life');
INSERT INTO "foods" VALUES(58, 2, 'Pancakes');
INSERT INTO "foods" VALUES(59, 2, 'Reese''s Peanut-Butter Puffs');
INSERT INTO "foods" VALUES(60, 2, 'Rice Krispies');
INSERT INTO "foods" VALUES(61, 2, 'Special K');
INSERT INTO "foods" VALUES(62, 2, 'Tightly Wrapped Magic Pan Crepes');
INSERT INTO "foods" VALUES(63, 3, 'Broiled Chicken');
INSERT INTO "foods" VALUES(64, 3, 'Casserole');
INSERT INTO "foods" VALUES(65, 3, 'Chicken');
INSERT INTO "foods" VALUES(66, 3, 'Chicken (for wedding)');
INSERT INTO "foods" VALUES(67, 3, 'Chicken Cashew (not ordered)');
INSERT INTO "foods" VALUES(68, 3, 'Chicken Kiev');
INSERT INTO "foods" VALUES(69, 3, 'Kung-Pao Chicken');
INSERT INTO "foods" VALUES(70, 3, 'Chicken Marsala');
INSERT INTO "foods" VALUES(71, 3, 'Chicken Piccata');
INSERT INTO "foods" VALUES(72, 3, 'Chicken with Poppy Seeds');
INSERT INTO "foods" VALUES(73, 3, 'Chicken, whole stuffed w/Gorganzolla and Ham');
INSERT INTO "foods" VALUES(74, 3, 'Chicken Skins');
INSERT INTO "foods" VALUES(75, 3, 'Chicken Wing (Shoulder Blades)');
INSERT INTO "foods" VALUES(76, 3, 'Chicken (Kenny Rogers) ');
INSERT INTO "foods" VALUES(77, 3, 'Chicken (Tyler) ');
INSERT INTO "foods" VALUES(78, 3, 'Colonel Chang Chicken');
INSERT INTO "foods" VALUES(79, 3, 'Cornish Game Hen');
INSERT INTO "foods" VALUES(80, 3, 'Duck');
INSERT INTO "foods" VALUES(81, 3, 'Duck, juicy breasts of');
INSERT INTO "foods" VALUES(82, 3, 'Turkey');
INSERT INTO "foods" VALUES(83, 3, 'Turkey, Kramer');
INSERT INTO "foods" VALUES(84, 3, 'Turkey Jerky');
INSERT INTO "foods" VALUES(85, 3, 'Turkey Chili');
INSERT INTO "foods" VALUES(86, 4, 'A1 Sauce');
INSERT INTO "foods" VALUES(87, 4, 'Barbeque Sauce');
INSERT INTO "foods" VALUES(88, 4, 'Dijon Mustard');
INSERT INTO "foods" VALUES(89, 4, 'Dill');
INSERT INTO "foods" VALUES(90, 4, 'Ginger');
INSERT INTO "foods" VALUES(91, 4, 'Gravy');
```

INSERT INTO "foods" VALUES(92, 4, 'Honey Mustard');
INSERT INTO "foods" VALUES(93, 4, 'Ketchup and Mustard together');
INSERT INTO "foods" VALUES(94, 4, 'Ketchup');
INSERT INTO "foods" VALUES(95, 4, 'Ketchup (secret)');
INSERT INTO "foods" VALUES(96, 4, 'Maple Syrup');
INSERT INTO "foods" VALUES(97, 4, 'Mustard (fancy)');
INSERT INTO "foods" VALUES(98, 4, 'Parsley');
INSERT INTO "foods" VALUES(99, 4, 'Pepper');
INSERT INTO "foods" VALUES(100, 4, 'Pesto');

列出上述命令，是为了兄弟们自行验证时方便。现在表内有 100 条记录，文件大小为 5K，说明 foods 表在文件中占 4 个页，其 B+tree 的逻辑结构如下图所示：



用 UltraEdit 打开文件 foods_test.db，观察文件头，其内容如下(深蓝色部分)：



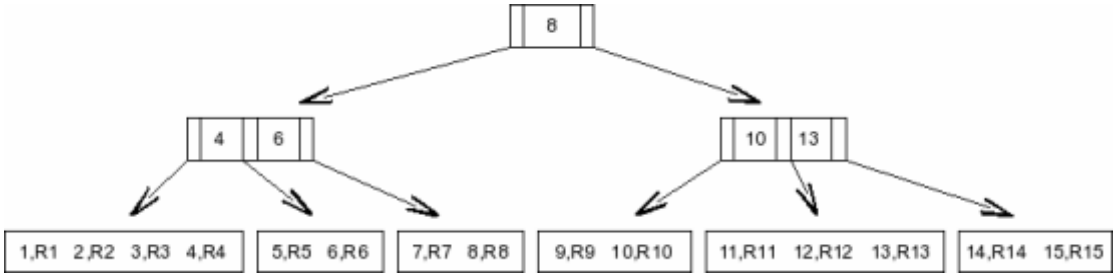
与前文比较，文件头内容基本无变化，只有偏移为 24 处的文件修改计数变成了 0X00000065，表示现在文件已经修改了 101 次，包括 1 次创建和 100 次插入。

Page 1 其他部分无变化。

2.2 B+tree 内部页格式介绍

Page 2 仍然是表 foods 的根页，但已经变成了内部页，格式有较大的变化。

对于数据库表，从 SQLite3 开始采用了 B+tree，在此，先对 B+tree 的结构做一个简单介绍。B+tree 与 B-tree 的主要区别在于，B-tree 的所有页上都包含数据，而 B+tree 的数据只存在于叶子页上，内部页只存储导航信息。B+tree 所有的叶子页都在同一层上，并按关键字排序，所有的关键字必须唯一，其逻辑结构举例如下图所示：



B+tree 中根页(root page)和内部页(internal pages)都是用来导航的，这些页的指针域都是指向下级页的指针，数据域仅仅包含关键字。所有的数据库记录都存储在叶子页(leaf pages)内。

在叶节点一级，页和页内的单元都是按照关键字的顺序排列的，所以 B+tree 可以沿水平方向遍历，时间复杂度为 O(1)。
我们将根页和内部页统称为内部页，它们的结构是相同的，其逻辑结构如下：

```
-------------------------------------------------------------
|  Ptr(0)  | Key(0)  | Ptr(1)  | Key(1)  | ...  | Key(N-1)  | Ptr(N)  |
-------------------------------------------------------------
```

内部页包含 N 个关键值(Key(0)~ Key(N-1))和 N+1 个子页指针(Ptr(0)~ Ptr(N))，其值为子页的页号。其中，Ptr(N)存储在页头中偏移为 8 的地方(4 字节整数，只有内部页的页头有此域，参“Btree 页格式介绍”一节)。其他的每对子页指针和关键值(Ptr(i) 和 Key(i))组成 1 个单元，共 N 个单元。Ptr(i) 所指向子树中关键字的最大值<=Key(i)，Ptr(N) 所指向子树中关键字的值都>Key(N-1)。

2.3 B+tree 内部页格式分析

现在对 foods B+tree 仅有的内部页进行分析。
文件第 2 页页头的内容如下：(图中深蓝色部分)

```
00000400h: 05 00 00 00 02 03 F6 00 00 00 00 05 03 FB 03 F6 ; ......?....?.??
00000410h: 03 96 03 7E 03 6A 03 58 03 3F 03 29 03 11 02 F7 ; .?~.j.X.?.)...?
00000420h: 02 E2 02 CC 02 BF 02 B0 02 A2 02 93 02 7C 02 69 ; .??????|.i
00000430h: 02 56 02 49 02 2F 02 18 02 05 01 F3 01 E3 01 D6 ; .V.I./.....???
00000440h: 01 C4 01 B0 01 A0 01 83 01 70 01 48 01 35 01 2B ; .????p.H.5.+
00000450h: 01 15 00 FA 00 E2 00 CA 00 B4 00 9D 00 87 00 75 ; ...??????u
00000460h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

前文对页头格式已经有比较详细的介绍，这里不再赘述。直接对内容进行说明：
0X05：说明该页为 B+tree 的内部页。
0X0000：第 1 个自由块的偏移量。此处为 0，表示本页没自由块。
0X0002：本页有 2 个单元。
0X03F6：单元内容区的起始位置。
0X00：碎片的字节数，此处为 0。
0X00000005：最右儿子的页号，即 Ptr(N)。由于本页有 2 个单元，所以此处即 Ptr(2)。其值为 0X05，即 Ptr(2)指向第 5 页。第 5 页是表数据的最后一页，也是当前文件的最后一页。

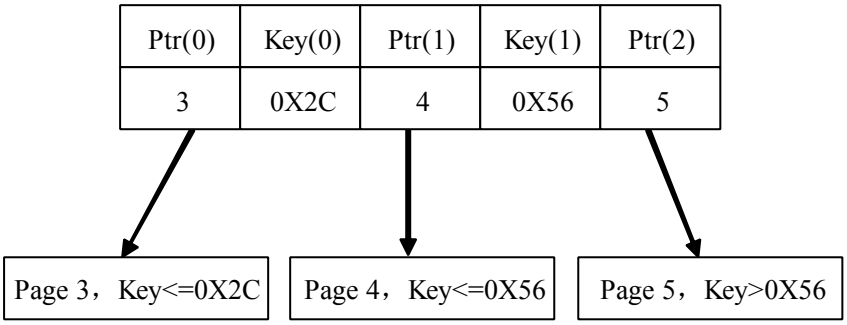
单元指针数组在页头之后，有 2 个指针，分别为 0X03FB 和 0X03F6。
注意：这两个指针后面还有一些乱七八糟内容，我也曾为此迷惑过。这些不是指针，而是属于“未分配空间”的一部分。因为此页在还没有成为内部页(还是叶子页)时，曾经插入过不少记录，有过不少指针。现在成为内部页了，只使用两个指针，但以前使用过的空间也没必要清零，再次使用时自然会覆盖。提示：此页尾部的内容区也存在这个情况，不再单独解释。

下面来看单元内容区的数据，内容如下：(图中深蓝色部分)

```
000007f0h: 73 69 6E 0B 01 04 00 00 00 04 56 00 00 00 03 2C ; sin.....V....,
```

由于单元内容区中各单元是反向增长的，所以两个单元的数据分别为：
0X00000003，0X2C
0X00000004，0X56
每个单元包括两部分内容：
一个 4 字节的页号，指向相应的儿子，即 Ptr(i)。此处分别指向第 3 页和第 4 页。
一个可变长整数，即 Key(i)。0X2C 表示最左儿子(文件第 3 页)中关键字值都<=0X2C。0X56

表示第 2 个儿子(文件第 4 页)中关键字都>0X2C，都<=0X56。注意：关键字值使用可变长整数，我们插入的记录少，在此都只有 1 个字节，所以看不出来。
前文刚介绍过，最右儿子的页号存储在页头中，值为 0X00000005，说明第 5 页中关键字值都>0X56。
重画前文 B+tree 的逻辑结构图如下所示：



2.4 叶子页格式分析

其实在上一章中分析 page 1 和 page 2 时，这两个页都是叶子页。这里再次对叶子页的格式进行分析，主要是为了验证前一节对内部页的分析结果，所以，咱就别嫌麻烦了。
文件第 4 页页头的内容如下：(图中深蓝色部分)

```
00000c00h: 0D 00 00 00 2A 00 6F 00 03 E7 03 D7 03 BC 03 B1 ; ....*.o..????
00000c10h: 03 A2 03 90 03 7C 03 6A 03 59 03 47 03 3B 03 31 ; .??|.j.Y.G.;.1
00000c20h: 03 26 03 17 02 F5 02 E1 02 D1 02 AA 02 94 02 84 ; .&...??????
00000c30h: 02 76 02 5A 02 37 02 24 02 0D 01 F7 01 E1 01 C2 ; .v.Z.7.$...???
00000c40h: 01 8F 01 7B 01 56 01 38 01 21 01 05 00 EE 00 E3 ; .?{.V.8.!...??
00000c50h: 00 C6 00 B9 00 A4 00 91 00 7E 00 6F 00 00 00 00 ; .????~.o....
```

之所以选择第 4 页，是因为该页为中间叶子，其记录的关键值应该在 Key(0)和 Key(1)之间。从上图可以看出，本页有 0X2A=42 个单元。第 1 个单元的入口地址为 0X03E7，最后一个单元的入口地址为 0X006F。
0X03E7 单元的内容为：

```
00000fe0h: 69 73 63 75 69 74 73 17 2D 04 00 01 31 01 50 6F ; iscuits.-...1.Po
00000ff0h: 70 70 79 20 53 65 65 64 20 4D 75 66 66 69 6E 73 ; ppy Seed Muffins
```

这是本页关键值最小的记录，可以看出其 rowid 值为 0X2D，恰大于 0X2C。
0X006F 单元的内容为：

```
00000c60h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D ; ................
00000c70h: 56 04 00 01 1D 04 41 31 20 53 61 75 63 65 11 55 ; V.....A1 Sauce.U
```

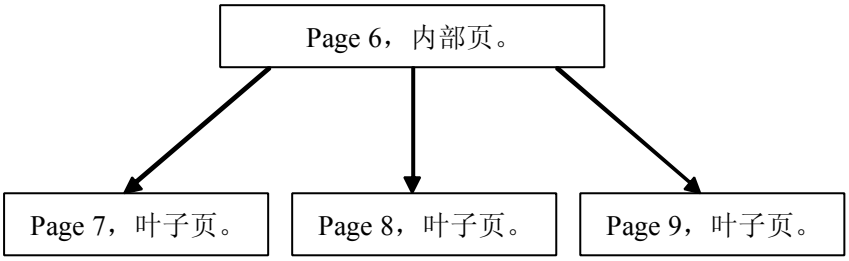
这是本页关键值最大的记录，可以看出其 rowid 值为 0X56=0X56。

3 索引格式分析

前面分析的都是表数据页的格式。表数据用 B+tree 来存储，而索引用 B-tree 来存储。两者的区别主要是：B-tree 中只存储关键字段的值和对应记录的 rowid 值；B-tree 树中的内部页也可以存储数据。

3.1 准备数据库

为表创建一个索引。
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
创建索引后文件大小为 9K，说明索引有 4 个页，其 B-tree 的逻辑结构如下图所示：



此处 4 个页是连续的，如果在创建表时同时创建索引，然后再插入记录，则用于存储数据的页和用于存储索引的页应该是交叉的。
此时观察 page 1 的内容，单元指针数组中多了一个单元指针，sqlite_master 表中多了索引对象记录，从其数据可以看出其根页为第 6 页。另外，Schema 版本的值变成了 2。

3.2 索引页的内部页

现在我们对索引 B-tree 唯一内部页(也就是根页，文件第 6 页)的格式进行分析。
文件第 6 页页头的内容如下：(图中深蓝色部分)

```
00001400h: 02 00 00 00 02 03 DC 00 00 00 00 09 03 F1 03 DC ; ......?....?ñ.Ü
```

说明：
0X02：说明该页为 B-tree 的内部页。
0X0000：第 1 个自由块的偏移量。0，说明当前自由块链表为空。
0X0002：本页有 2 个单元。
0X03DC：单元内容区的起始位置为 0X03DC。
0X00：碎片的字节数，当前为 0。
0X00000009：最右儿子的页号，即 Ptr(N)。由于本页有 2 个单元，所以此处即 Ptr(2)。其值为 0X09，即 Ptr(2)指向第 9 页。第 9 页是索引数据的最后一页，也是当前文件的最后一页。

单元指针数组在页头之后，有 2 个指针，分别为 0X03F1 和 0X03DC。

下面来看单元内容区的数据。
0X03F1 单元的内容如下：(图中深蓝色部分)

```
000017f0h: 21 00 00 00 07 0A 03 19 01 43 6F 6F 6B 69 65 0F ; !........Cookie.
```

0X00000007：为 Ptr(0)，指向第 7 页。
0X0A：Payload 数据的字节数。数据从 03~0F。
0X03：记录头包括 3 个字节。
0X19：字段 1。TEXT，长度为：(25-13)/2=6。字段值为：cookie。
0X01：字段 2。整数，长度为 1。字段值为：0X0F，表示索引值 cookie 所对应记录的关键值(即记录的 rowid 值)为 15。
注：如果是通过索引字段查找“cookie”，现在就可以按其 rowid 值=15 到数据 B+tree 树中去

检索记录的全部数据了。

0X03DC 单元的内容如下：(图中深蓝色部分)

```
000017d0h: 61 76 61 72 69 61 6E 20 43 72 65 61 00 00 00 08 ; avarian Crea....
000017e0h: 10 03 25 01 50 65 61 63 68 20 4D 75 66 66 69 6E ; ..%.Peach Muffin
000017f0h: 21 00 00 00 07 0A 03 19 01 43 6F 6F 6B 69 65 0F ; !.........Cookie.
```

0X00000008：为 Ptr(1)，指向第 8 页。

0X10：Payload 数据的字节数。数据从 03~21。

0X03：记录头包括 3 个字节。

0X25：字段 1。TEXT，长度为：(37-13)/2=12。字段值为：Peach Muffin。

0X01：字段 2。整数，长度为 1。字段值为：0X21，表示索引值 Peach Muffin 所对应记录的关键值为 33。

3.3 索引页的叶子页格式

对于 B-tree 来说，内部页与叶子页的格式差别其实不大。简单分析一下，为了验证上一节的结果。

文件第 8 页前半部分的内容如下：

```
00001c00h: 0A 00 00 00 29 01 6B 00 01 6B 01 7B 01 90 01 9D ; ....).k..k.{.??
00001c10h: 01 A9 01 B6 01 CB 01 DD 01 E6 01 F7 02 08 02 13 ; .??????...
00001c20h: 02 25 02 3D 02 46 02 61 02 76 02 86 02 91 02 A0 ; .%.=.F.a.v.???
00001c30h: 02 AA 02 BA 02 CC 02 DD 02 E7 02 F3 03 08 03 29 ; .??????..)
00001c40h: 03 31 03 46 03 4F 03 5F 03 6F 03 7D 03 88 03 9A ; .1.F.O._.o.}.??
00001c50h: 03 AA 03 BE 03 CC 03 D9 03 E5 03 54 03 66 03 74 ; .?????.T.f.t
```

页头的第 1 个字节值为 0X0A，说明该页为 B-tree 的叶子页。其它不再详述。

单元内容区的开始部分内容如下：

```
00001d60h: 03 45 01 4B 65 74 63 68 75 70 20 0F 03 23 01 43 ; .E.Ketchup ..#.C
00001d70h: 6F 72 6E 20 46 6C 61 6B 65 73 32 14 03 2D 01 43 ; orn Flakes2..-.C
00001d80h: 6F 72 6E 69 73 68 20 47 61 6D 65 20 48 65 6E 4F ; ornish Game HenO
00001d90h: 0C 03 1D 01 43 72 61 63 6B 65 72 73 10 0B 03 1B ; ....Crackers....
```

单元内容区的开始和结束部分内容如下：

```
00001fc0h: 1F 01 4E 75 74 20 42 72 65 61 64 1F 0C 03 1D 01 ; ..Nut Bread.....
00001fd0h: 50 61 6E 63 61 6B 65 73 3A 0B 03 1B 01 50 61 72 ; Pancakes:....Par
00001fe0h: 73 6C 65 79 62 1A 03 39 01 50 61 73 74 72 69 65 ; sleyb..9.Pastrie
00001ff0h: 73 20 28 6F 66 20 74 68 65 20 47 6F 64 73 29 20 ; s (of the Gods)
```

不再详细分析了，粗看可以看出：

此页记录的索引值确实都在 cookie 和 Peach Muffin 之间，不包括这两个值(这两个值已经在根页中了)。单元数据都按索引值大小排序。

4 碎片、自由块和空闲页

到目前为止，我们只对数据库表进行了插入操作，因此，文件格式还是很“整齐”的。如果对数据库进行删除和修改操作，就会产生碎片、自由块和空闲页。本文 1.2 节和 1.3 节对相应的概念有较详细介绍，本章就算是对前述内容的例证吧。

4.1 碎片

执行下面的 update 语句，每执行一句都将原记录的 name 字段长度减少 2 字节(可与前文的 insert 语句对照)。

update foods set name='Bavarian Cream P' where id=3;

执行完上面语句后，文件第 3 页的页头内容如下图所示：

```
00000800h: 0D 00 00 00 2C 00 75 02 03 F3 03 DE 03 C7 03 B4 ; ....,.u..????
```

可以看到，偏移为 7 的字节值变成了 2，说明当前页中有 2 字节的碎片。再执行下面语句：

update foods set name='Bear Cla' where id=4;

可以看到当前页中的碎片字节数已经变成了 4，如下图所示：

```
00000800h: 0D 00 00 00 2C 00 75 04 03 F3 03 DE 03 C7 03 B6 ; ....,.u..????
```

至于碎片字节数到多大才会整理，这得分析源程序了。

4.2 自由块

执行下面删除语句：

delete from foods where id=5;

文件第 3 页的页头内容如下：

```
00000800h: 0D 03 96 00 2B 00 75 04 03 F3 03 DE 03 C7 03 B6 ; ..?+.u..????
```

可以看到，第 1 个自由块的偏移量为 0X0396。

观察 0X0396 处的单元，数据如下图所示：

```
00000b90h: 20 6E 75 74 73 29 00 00 00 1E 01 3B 01 42 6C 61 ;  nuts).....;.Bla
00000ba0h: 63 6B 20 61 6E 64 20 57 68 69 74 65 20 63 6F 6F ; ck and White coo
00000bb0h: 6B 69 65 73 00 00 0D 04 04 00 01 1D 01 42 65 61 ; kies.........Bea
```

单元的前 2 个字节指向下一个空闲块，此处值为 0，表示没有下一个空闲块了。

0X001E 表示该空闲块的大小，从数值上来分析，应该是包含了这两个字节本身。

如果再删除 1 条记录，就可以看到自由块是如何串接的，我们就不做了。

4.3 空闲页

执行下面删除语句：

delete from foods where id>10;

文件大小仍为 9K，但此时文件中应该有了 6 个空闲页(索引和数据都只需要一个页就能放下了)，分别是页 3、4、5、7、8 和页 9。

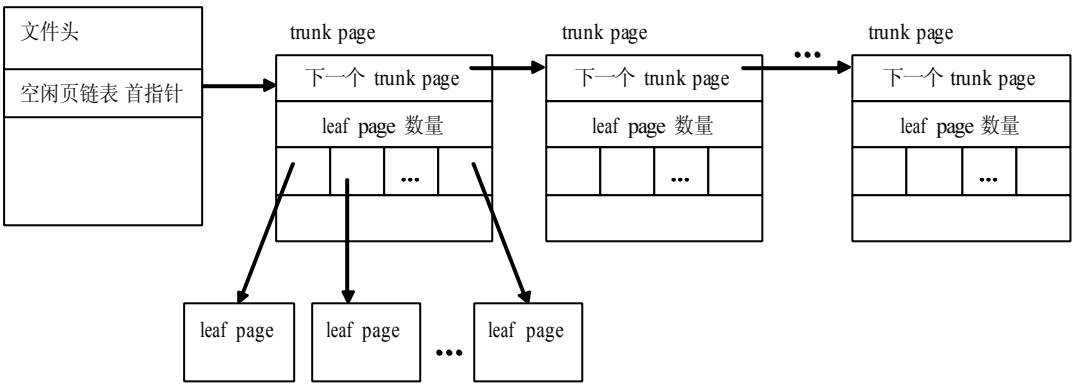
观察此时的文件头，内容为：

```
00000000h: 53 51 4C 69 74 65 20 66 6F 72 6D 61 74 20 33 00 ; SQLite format 3.
00000010h: 04 00 01 01 00 40 20 20 00 00 00 6A 00 00 00 00 ; .....@  ...j....
00000020h: 00 00 00 05 00 00 00 06 00 00 00 02 00 00 00 01 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 0D 00 00 00 02 03 3D 00 03 99 03 3D ; ..........=..?=
```

注意其中深蓝色部分，可以看到，偏移为 32 的 4 个字节中值为 0X00000005，表示空闲页链表首指针指向第 5 个页。对了，就应该是第 5 个页先空闲出来。

偏移为 36 的 4 个字节中值为 0X00000006，表示文件中空闲页的数量为 6。

现在得介绍一下空闲页链表的格式了。
空闲页有两种类型：trunk page(主干页)和 leaf page(叶子页)。文件头偏移为 32 处的指针指向空闲链表的第一个 trunk page，每个 trunk page 指向多个叶子页。偏移 36 处的 4 个字节为空闲页的总数量，包括所有的 trunk page 和 leaf page。空闲页链表的结构如下图所示：



其中，trunk page 的格式(从页的起始处开始)如下：
(1)4 个字节，指向下一个 trunk page 的页号，0 表示链表结束；
(2)4 个字节，该页 leaf page 的数量；
(3)0 个或多个指向 leaf page 的页号，每项 4 个字节。

文件第 6 页前部的内容如下：

```
00001000h: 00 00 00 00 00 00 00 05 00 00 00 09 00 00 00 08 ; ................
00001010h: 00 00 00 07 00 00 00 04 00 00 00 03 03 2F 03 21 ; ............./.!
00001020h: 03 14 03 08 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00001030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

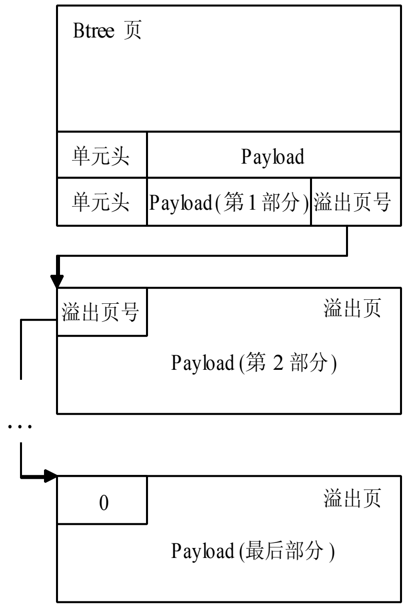
说明：
下一个 trunk page 的页号为 0X00000000，表示链表中只有此一个 trunk page，没有后继结点了。
该页 leaf page 的数量为 0X00000005，表示本页带有 5 个 leaf page，依次为 0X00000009，0X00000008，0X00000007、0X00000004 和 0X00000003。

SQLite 对 leaf page 的格式没有规定。

5 溢出页的格式

5.1 溢出页格式说明

如前所述，单元(cell)具有可变的大小，而页的大小是固定的，这就有可能一个单元比一个完整的页还大，这样的单元就会溢出到由溢出页组成的链表上，如下图所示：



在上图中，Btree 页的最后一个单元超大，需要使用溢出页。此时，单元的最后 4 个字节为溢出页链表中第 1 个溢出页的页号。对于每个溢出页，其头 4 个字节为下一溢出页的页号，该值为 0 时表示此页为溢出页链表的表尾页。

除最后一个溢出页外，每个溢出页全部填充数据(除了最开始的 4 个字节)。最后一个溢出页可能数据很少，甚至只有一个字节的数据，但一个溢出页不会存储来自两个单元的数据。

5.2 准备数据库

得换一个数据库了。我们用下面的方法来分析溢出页的格式，并对上节内容进行验证。

执行 SQLite 的命令行工具，创建一个新的数据库 food_text.db。

D:\SQLite\CLP>sqlite3 foods_text.db

创建一个新表。

```
CREATE TABLE foods(
   id integer primary key,
   type_id integer,
   name text );
```

插入 1 条记录，该记录包含一段较长的文本。

```
INSERT INTO "foods" VALUES(1, 1,
'00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
00000000010000000002000000000300000000040000000009
```

000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
000000000100000000020000000003000000000400000000 9
');
该记录的文本字段包括 21 行，每行 50 字节(49 字符+1 换行)，共 1050 个字节。

退出命令行工具。当前目录下多了一个文件 foods_text.db，大小为 3K，其中第 3 页应该为溢出页。

5.3 溢出页格式分析

用 UltraEdit 打开 foods_text.db 文件，对其进行分析。
文件第 2 页前部内容如下图所示：

```
00000400h: 0D 00 00 00 01 03 92 00 03 92 00 00 00 00 00 00 ; ......?.?.....
00000410h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

可以看出，该表唯一的记录单元始于 0X0392。0X0392 单元的内容如下图所示：

```
00000790h: 00 00 88 20 01 05 00 01 90 41 01 30 30 30 30 30 ; ..?....怉.00000
000007a0h: 30 30 30 30 31 30 30 30 30 30 30 30 30 30 32 30 ; 0000100000000020
000007b0h: 30 30 30 30 30 30 30 30 33 30 30 30 30 30 30 30 ; 0000000030000000
000007c0h: 30 30 34 30 30 30 30 30 30 30 30 30 39 0A 30 30 ; 0040000000009.00
000007d0h: 30 30 30 30 30 30 31 30 30 30 30 30 30 30 30 30 ; 0000001000000000
000007e0h: 32 30 30 30 30 30 30 30 30 30 30 33 30 30 30 30 ; 2000000000300000
000007f0h: 30 30 30 30 34 30 30 30 30 30 30 30 30 00 00 00 03; 000040000000....
```

0X8820 是可变长整数，其值：
0X8820=0B1000,1000,0010,0000 转换为整数=0B100,0010,0000=4*256+2*16=1056。说明数据区(从 0X0500 开始)长度为 1056，显然需要使用溢出页。
0X01：当前记录的 rowid，值为 0X01。
0X05：记录头包括 5 个字节。
0X00：字段 1。类型为 NULL(详细说明参“关于自增长的主键字段”一小节)。
0X01：字段 2。1 字节整数。对应的数据值为 01，表示本记录 type_id 字段的值为 01。
0X9041(可变长整数)：字段 3。TEXT，长度为：(2113-13)/2=1050，与插入文本值的长度相符。
本单元最后 4 个字节为 0X00000003，表示第 1 个溢出页的页号为 3。
文件第 3 页前部内容如图所示：

```
00000800h: 00 00 00 00 30 39 0A 30 30 30 30 30 30 30 30 30 ; ....09.000000000
00000810h: 31 30 30 30 30 30 30 30 30 30 32 30 30 30 30 30 ; 1000000000200000
```

头 4 个字节值为 0，表示此页后面再无溢出页。
观察此页内容，大部分为文本字段的内容，这里不再详述了。

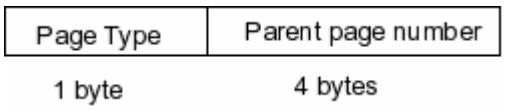
6 指针图页

只有 auto-vacuum 数据库才有指针图页(Pointer map page)。如果数据库文件头偏移为 52 字节的地方为一非零值，该数据库为 auto-vacuum 数据库。

数据库中所有的指针图页共同构成一个查找表，利用该表可以确定数据库中各页的类型及其父亲页的页号。查找表将页按下表分类：

页类型	字节值	说明
Btree 根页	0x01	Btree 结构的根页。没有父亲页，存储在指针图查找表中的值永远为 0。
空闲页	0x02	空闲页链表中的页。没有父亲页，指针图查找表中存储父亲页号的地方值为 0。
溢出页类型 1	0x03	溢出页链表的第 1 个页。父亲页是包含本页所属单元的 Btree 页。
溢出页类型 2	0x04	溢出页链表中非第 1 页的页。父亲页是它在链表中的前一个页。
Btree 页	0x05	Btree 结构的页，既不是根页也不是溢出页。父亲页就是 Btree 结构中的父结点。

指针图页本身不出现在指针图查找表中。Page 1 也不出现在指针图查找表中。指针图入口格式如下图所示：



每个指针图查找表入口使用 5 个字节。第 1 个字节为页类型，后 4 个字节为父亲页号(高位字节在前)。每个指针图页可以包含

num-entries := usable-size / 5

个入口，其中“可用大小”为页大小减页尾部保留空间的大小（参 1.2 节）。

如果数据库是 auto-vacuum 的，page 2 永远是指针图页。它保存了从第 3 页到第(2 + num-entries)页的指针图查找表入口。其中 page 2 的头 5 个字节保存的是第 3 页的指针图查找表入口，5~9 字节保存的是第 4 页的指针图查找表入口，依此类推。

数据库中下一个指针图页的页号是(3 + num-entries)，它保存了从第(4 + num-entries)页到第(3+2*num-entries)页的指针图查找表入口。一般而言，对于任何大于 0 的 n，页号为(2 + n * num-entries)的页为指针图页。非 auto-vacuum 数据库没有指针图页。

7 日志文件格式

SQLite 数据库在文件系统中表现为：

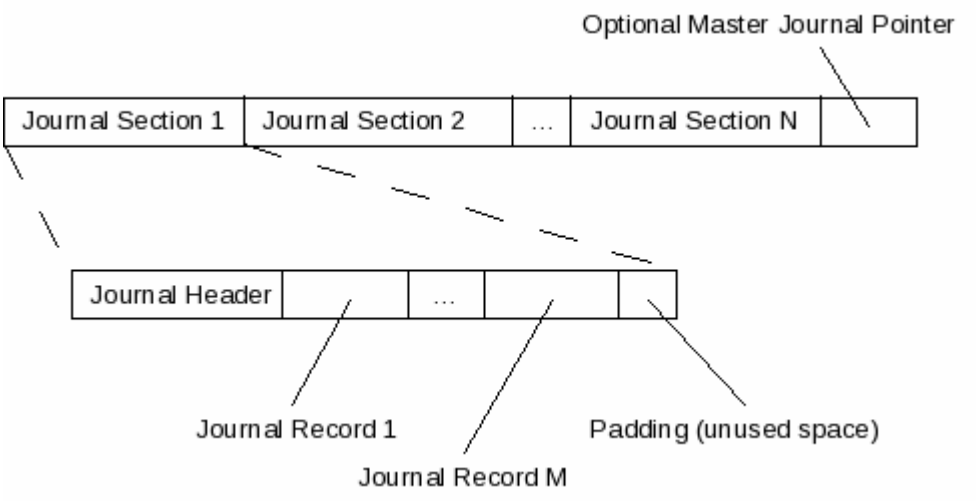
- 一个主数据库文件。该文件永远存在，可能只有 0 字节，但永远存在。
- 有时会有一个回卷日志文件，与数据库主文件同目录同名，带后缀"-journal"。

- 有时会有一个主日志文件，只有当回卷日志文件存在时，才有可能有主日志文件。主日志文件可能在文件系统的任何地方，可能是任何文件名。如果有主日志文件，其指针保存在回卷日志文件中。

使用热日志进行回卷的过程在其它文档中介绍，本文关注日志文件的格式。
7.1 介绍回卷日志文件格式，7.1 介绍主日志文件格式。

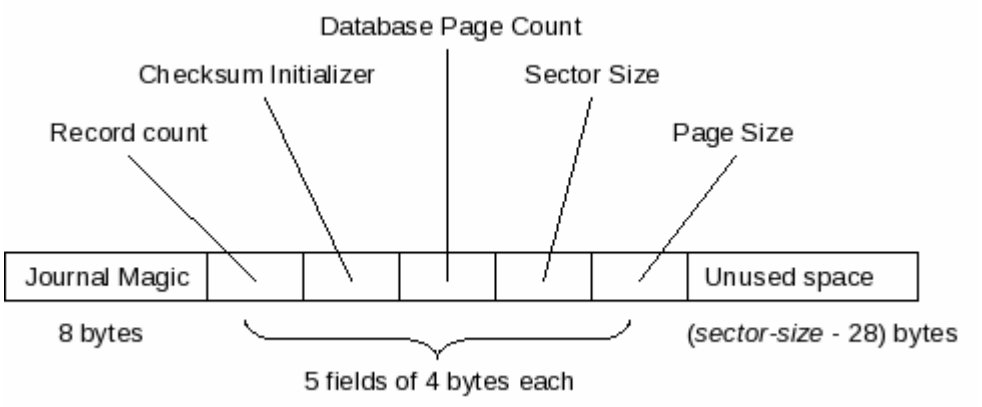
7.1.1 回卷日志文件细节

回卷日志文件由一个或多个日志扇区组成，可能会跟随一个主日志文件指针域。第 1 个日志扇区位于日志文件的开始处。文件中的扇区数没有限制。
每个日志扇区包含一个日志头，后面跟着 0 个或多个日志记录。日志头中有一个数值域，保存的是扇区大小(sector-size)。日志文件中每个扇区的大小必须是首扇区头中保存的扇区大小的整数倍(其它扇区中保存的扇区大小值没用)。如果一个扇区中头和所有记录大小之和不是规定扇区大小的整数倍，在最后一个扇区记录后面填充未使用空间，使其达到规定的大小。
下图图示了一个日志文件，其中包含 N 个扇区和一个主日志文件指针。第一个扇区包含 M 个日志记录。



7.1.1.1 日志头格式

日志头的大小为 sector-size，sector-size 值是一个 32-bit 大端格式无符号整数值，存储在日志文件中首个日志头偏移为 20 的位置。sector-size 必须是 2 的幂且大于等于 512。日志头只有前 28 个字节被使用，其它的空间为填充数据。每个日志头的前 28 个字节包含一个 8 字节的标识值(journal magic field)，后面跟着 5 个 32-bit 大端格式无符号整数域。日志头格式如下图所示：



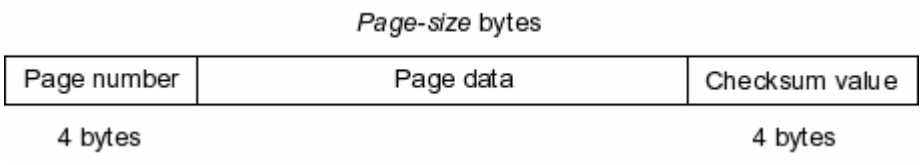
上图中各域的说明见下表。

偏移	大小	说明
0	8	journal magic：包含一个标识值，用来识别 SQLite 日志文件。该值是一个字节序列，为：0xd9 0xd5 0x05 0xf9 0x20 0xa1 0x63 0xd7。
8	4	record count：日志文件中跟在此日志头后面的记录数。
12	4	checksum initializer field：检查值初始化域，设为一个伪随机值，用来参与计算日志记录中的检查值。
16	4	database page count：在写事务执行任何修改之前，数据库文件中的页数。
20	4	sector size：被设为创建日志文件的设备的扇区大小。当读日志文件来决定每个日志头的大小时，这个值是有用的。
24	4	page size：相关数据库文件的页大小。

因为日志头总是出现在日志扇区的开始处，而日志扇区大小总是 sector-size 的整数倍，所以，日志头总是出现在偏移为 sector-size 整数倍的地方。

7.1.1.2 日志记录格式

每个日志记录包含一个单独的数据库页的数据，一个页号用来识别该页，还有一个检查值用来帮助判断日志文件数据是否已混乱。日志记录格式如下图所示：



一个日志记录包含 3 个域，在下表中分别说明。

偏移	大小	说明
0	4	与本记录相关的数据库文件页的页号。
4	page-size	此域包含了写事务开始前相关数据库页的原始数据。
4 + page-size	4	此域包含一个检查值。该值通过前一个域中的数据(数据库页中的原始数据)与存储在前一个日志头中检查值初始化域的值联合计算获得。

检查值初始化值为一个 32-bit 无符号整数。从页数据中每隔 200 个字节选取 1 个字节，所选第 1 个字节的偏移为 page-size % 200。将检查值初始化值与上述各字节值相加，保留一个 32-bit 无符号整数值(溢出部分清 0)。

例如，如果页大小为 1024 字节，数据页中需要与检查值初始化值相加的字节的偏移为 24、

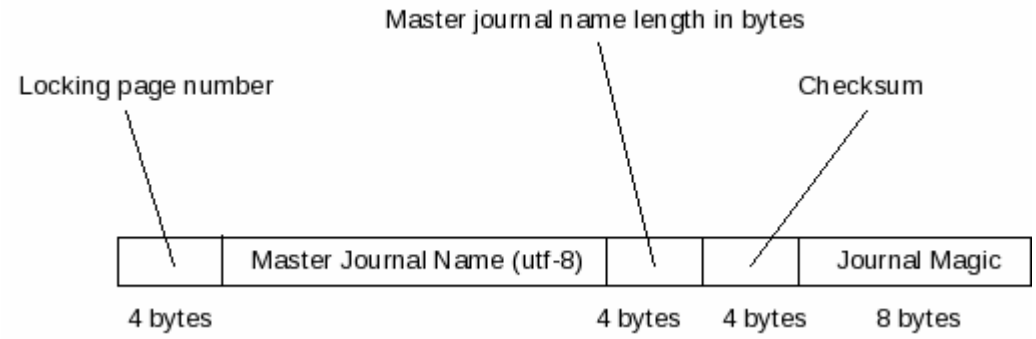
224、424、624 和 824(页的第 1 个字节偏移为 0，最后 1 个字节偏移为 1023)。如果上述偏移处的字节值为 0x23, 0x32, 0x9E, 0x62 and 0x1F，检查值初始化值为 0xFFFFFFE1，那么，存储在检查值域的值为 0x00000155。

```
    0xFFFFFFE1 +
    0x00000023 +
    0x00000032 +
    0x0000009E +
    0x00000062 +
    0x0000001F
   ------------
   0x100000155

截短为 32-bits:
    0x00000155
```

7.1.1.3主日志文件指针

如果有主日志文件，主日志文件指针出现在日志文件的尾部。在最后一个日志扇区和主日志文件指针之前，可能有（也可能没有）未用空间。
主日志文件指针包含主日志文件的全路径和其它一些域，其格式如下图所示：



主日志文件指针包含 5 个域，在下表中分别说明。

偏移	大小	说明
0	4	locking page number：数据库文件锁页的页号(锁页的说明参 2.2.2 节)。
4	name-length	master journal name：主日志文件名，是一个采用 utf-8 编码的字符串，该字符串没有'\0'结束符。
4 + name-length	4	前一个域的字节数，4 字节大端格式无称号整数。
8 + name-length	4	Checksum：一个 4 字节大端格式无称号整数的检查值。该检查值是主日志文件名域中各字符的和(每个字符被看成一个 8-bit 的符号整数)。
12+name-length	8	与日志头中 journal magic 域相同的内容，包含一个标识值，用来识别 SQLite 日志文件。该值是一个字节序列，为：0xd9 0xd5 0x05 0xf9 0x20 0xa1 0x63 0xd7。

7.1.2 主日志文件细节

一个主日志文件包含两个或更多个回卷日志文件的全路径名，用 UTF-8 编码，以空字符(0x00)结束。多个路径名之间没有填充符，下一个路径名紧接着前一个路径名后的空字符开始。
了解主日志文件的内容并没有想像中的那么重要，读数据库映像时根本用不到这些知识。主日志文件只在清理时使用。

8 鸣谢

接触 SQLite 时间不长，但很喜欢。
分析源程序时，发现每个 SQLite 源文件的头部都有这样一段话：
The author disclaims copyright to this source code. In place of a legal notice, here is a blessing:
May you do good and not evil.
May you find forgiveness for yourself and forgive others.
May you share freely, never taking more than you give.
这几句话我很喜欢，翻译不好，就拿原文出来吧，与大家共勉。

接触 SQLite 时间不长，但收获很大。
所以，在此感谢所有为 SQLite 发展做出过贡献的人，都是好人啊！
“空转”是我的网名之一，网上网下知之者甚少，也就是一起骑车的几个人知道吧。如果本文对您能有一点点帮助，也算是他对 SQLite 做了一点贡献吧。

接触 SQLite 时间不长，所以本文难免会有很多错误，不是故意误导大家，是真不懂。如果有兄弟想对我提出指导，我的邮箱是：njgaoyi@yahoo.com.cn。如果我没有回信，不是因为不想回，是因为我很少上网，在此先行谢过。

感谢我的儿子，多好的孩子呀！他的自觉与乖巧使我能够不受打扰地做这些“不务正业”的工作。跟我比他算是聪明的了，希望他将来也能像我一样勤劳。

Ver 1.00: 2009-10-21
Ver 1.01: 2009-10-26
Ver 1.02: 2009-11-28
于南京

9 附录一：SQLite 使用的临时文件

原文：http://www.sqlite.org/tempfiles.html
本文讨论 SQLite 创建并使用的各种临时文件。主要讨论这些临时文件何时创建、何时删除、是干什么用的、它们为什么重要和如何在系统中避免使用它们。

9.1 七种临时文件

SQLite 当前使用 7 种不同类型的临时文件：

1. 回卷日志(Rollback Journals)文件
2. 主日志(Master Journal)文件
3. 语句日志(Statement Journal)文件
4. 临时数据库(TEMP Databases)
5. 物化的视图和子查询(Materializations Of Views And Subqueries)
6. 瞬时索引(Transient Indices)
7. VACUUM 时使用的瞬时数据库(Transient Database)

9.1.1 回卷日志(Rollback Journals)文件

空注：回卷日志文件是 SQLite 使用最多的临时文件，所以，在很多其它文档中，包括 SQLite 自己的一些文档中，回卷日志文件就称为日志文件。
回卷日志用来实现原子提交和回卷。回卷日志文件与数据库文件在相同的目录中，在数据库文件名后增加"-journal"后缀。回卷日志通常在事务开始时创建并在事务提交或回卷时删除。如果没有回卷日志，SQLite 就不能回卷一个未完成的事务，如果在事务执行当中发生断电等情况，整个数据库将可能发生混乱(go corrupt，空注：本来此处我翻译成“不一致”，后来研究了一下，可能不只是不一致，而是可能完全乱掉。)。
回卷日志通常在事务开始时创建并在事务提交或回卷时删除，但有例外。
如果在事务执行当中发生断电等情况，回卷日志可能会被保留在磁盘上。下次另一个应用程序试图打开数据库文件时，它会发现这个被遗弃的日志(称为“热日志”，"hot journal")，它会使用这个日志中的信息使数据库恢复到那个未完成事务开始之前的状态。这就是 SQLite 实现原子提交的方法。
如果一个应用程序在排它锁模式下使用 SQLite：
PRAGMA locking_mode=EXCLUSIVE;
SQLite 在排它锁模式(的连接)的第一个事务开始时创建回卷日志，但在事务结束时并不删除它。回卷日志可能被截短为 0 字节，或将其头部清 0(到底如何处理与 SQLite 的版本有关)，但不删除。回卷日志会一直保留到从排它存取模式中退出。
回卷日志何时创建和删除还受 journal_mode pragma 的影响。默认的日志模式是 DELETE，此模式下会在事务结束时删除日志。在 PERSIST 日志模式下，事务结束时不删除日志，而是将日志头的内容覆盖为 0，这也可以阻止其它进程用此日志做回卷操作，效果与删除它是一样的，但节省了从磁盘真正删除文件的代价。也就是说，PERSIST 模式的外在表现与排它锁模式是相同的。在 OFF 日志模式下不创建回卷日志。OFF 日志模式使 SQLite 的原子提

交和回卷功能不可用。在 OFF 模式下，ROLLBACK 不可用，如果发生断电等情况，数据库就可能会混乱。

9.1.2 主日志(Master Journal)文件

当一个单独的事务对附加到同一个连接中的多个数据库进行了修改时，主日志文件作为原子提交的一部分被用到。主日志文件与“主数据库”文件在相同的目录中，在数据库文件名后增加一个随机的后缀。主日志文件中包含了事务期间被改变了的所有附加数据库的名称。操作多个数据库的事务在主日志文件被删除后提交。参文档“Atomic Commit In SQLite”。
(此处略去若干字)

9.1.3 语句日志(Statement Journal)文件

语句日志文件用来在大事务中回卷特定的单个语句的执行结果。例如，UPDATE 语句试图修改 100 条记录，但在改了 50 条之后，遇到了违反约束的情况，这时整个语句都不能执行。语句日志用来取消对前 50 个记录的修改，从而使数据库恢复到语句开始之前的状态。
语句日志仅为 UPDATE 和 INSERT 这样可能改变大量记录的语句创建，这些语句执行时可能违反约束或抛出异常并需要取消特定的执行结果。如果在 BEGIN...COMMIT 之间不包含 UPDATE 或 INSERT 语句，并且当前连接中也没有其它活动的语句，则不会创建语句日志，而是用传统的回卷日志来替代。
(此处略去若干字)
语句日志被赋予随机的文件名，不需要跟主数据库在同一目录，在事务结束时被自动删除。语句日志的大小与 UPDATE 可 INSERT 命令影响的数据量有关。

9.1.4 临时数据库(TEMP Databases)

使用"CREATE TEMP TABLE"创建的临时表只在当前连接中可见。临时表和与其关联的索引、触发器、视图等集中起来存储在临时数据库文件中，临时数据库在第一个"CREATE TEMP TABLE"命令执行时被创建。这个独立的临时数据库文件也有它自己关联的回卷日志。临时数据库文件在当前连接关闭时被删除。
临时数据库文件很像用 ATTACH 命令附加的数据库，但有一些特殊属性，包括：临时数据库永远都是在连接关闭时自动删除；临时数据库永远使用 synchronous=OFF 和 journal_mode=PERSIST PRAGMA 设置；临时数据库不能被 DETACH。

9.1.5 物化的视图和子查询(Materializations Of Views And Subqueries)

如果查询中包含子查询，有时必须对子查询进行单独的处理并将结果保存在临时表中，然后再用此临时表的内容来求主查询的结果。这们称这种情况为物化(materializing)子查询。SQLite 的优化器会尽量避免物化，但有时很难做到。由物化而生成的每个临时表都存储在它们单独的临时文件中，当主查询完成后会自动删除。这种临时表的大小依赖于物化子查询

的数据量。比如使用 IN 操作符的子查询就经常需要物化，例如：
SELECT * FROM ex1 WHERE ex1.a IN (SELECT b FROM ex2);
为了避免创建临时表，上面的查询可以写成：
SELECT * FROM ex1 WHERE EXISTS(SELECT 1 FROM ex2 WHERE ex2.b=ex1.a);
SQLite 的较新版本(3.5.4 及以后版本)会自动进行上述重写。
如果 IN 操作符的右手是值列表，如下：
SELECT * FROM ex1 WHERE a IN (1,2,3);
列表值必须物化，因为会为这些值使用临时索引。也就是说上述语句相当于：
SELECT * FROM ex1 WHERE a IN (SELECT 1 UNION ALL
SELECT 2 UNION ALL
SELECT 3);
当子查询出现在 SELECT 语句的 FROM 子句中时，也需要物化。例如：
SELECT * FROM ex1 JOIN (SELECT b FROM ex2) AS t ON t.b=ex1.a;
SQLite 的优化器会将上述语句重写为：
SELECT ex1.*, ex2.b FROM ex1 JOIN ex2 ON ex2.b=ex1.a;
(此处略去若干字)

9.1.6 瞬时索引(Transient Indices)

SQLite 会使用瞬时索引来实现 SQL 语言的如下特色：

- ORDER BY 或 GROUP BY 子句
- 聚合查询中的 DISTINCT 保留字
- 被 UNION、EXCEPT 或 INTERSECT 结合起来的复合查询语句

每个瞬时索引存储在它们自己的临时文件中，当语句执行结束时被自动删除。
SQLite 尽量用已存在的索引实现 ORDER BY。但如果找不到合适的索引，SQLite 就会将查询结果存储在瞬时索引中，其索引关键字就是 ORDER BY 的条目。在结果形成后，SQLite 再回到瞬时索引的开始处，遍历并输出结果(已经是按需要的顺序保存了)。
SQLite 实现 GROUP BY 的方法是将输出的记录按 GROUP BY 条目排序。每行都与前面一行进行比较看是否开始了一个新的"group"。按 GROUP BY 条目排序的方法与 ORDER BY 完全相同。SQLite 尽量用已存在的索引实现，但如果没有合适的索引，就会创建瞬时索引。
(此处略去若干字)

9.1.7 VACUUM 时使用的瞬时数据库(Transient Database)

VACUUM 命令创建一个临时文件，并将数据库整个重建到该临时文件中。然后，临时文件的内容复制回原数据库文件中并将临时文件删除。
由 VACUUM 命令创建的临时文件仅在命令执行期间存在，大小不会超过原数据库。

9.2 编译期参数 SQLITE_TEMP_STORE 和 Pragma

回卷日志、主日志和语句日志文件总是写到磁盘。但其它种类的临时文件可能只存储在内存中而永远不会写磁盘。这由编译期参数 SQLITE_TEMP_STORE、temp_store pragma 和临时

文件的大小决定。
(此处略去若干字)

9.3 其它的临时文件优化

SQLite 使用页缓冲区保存刚才读或写的数据库页。其实页缓冲区不仅被主数据库文件使用，还可以用做存储瞬时索引和临时文件中的表。即使 SQLite 设置这些内容是应该存储在磁盘上，其实刚开始时它们也都是存储在缓冲区中，在缓冲区满之前是不会写盘的。
这意味着很多情况下临时表或索引很小(缓冲区足以容纳)，就没有临时文件创建，也就没有磁盘读写操作。
分配给临时表和索引的缓冲区空间大小由编译期参数
SQLITE_DEFAULT_TEMP_CACHE_SIZE 决定，默认值为 500 页。这个值在运行时不能修改。