China-pub.com
下载

China-pub.com
下载

第16章　数据库函数库

16.1　引言

80年代早期，UNIX环境被认为不适合运行多用户数据库系统（见 Stonebraker〔1981〕和 Weinberger〔1982〕）。早期的系统，如V7，因为没有提供任何形式的IPC机制（除了半双工管道），也没有提供任何形式的记录锁机制，所以确实不适合运行多用户数据库。新一些的系统，象SVR4和4.3+BSD，则为运行可靠的、多用户的数据库系统提供了一个适合的环境。很多商业公司在许多年前就已提供这种系统。

本章将设计一个简单的、多用户数据库的函数库。通过此函数库提供的 C语言函数，其他程序可以访问数据库中的记录。这个C函数库只是一个完整的数据库的很小的一部分，并不包括其他很多部分，如查询语言等，关于其他部分可以参阅专门介绍数据库的书。我们感兴趣的是一个数据库函数库与UNIX的接口，以及这些接口与前面各章节的关系（如12.3节的记录锁）。

16.2　历史

dbm（3）是一个在UNIX系统中很流行的数据库函数库，它由Ken Thompson开发，使用了动态散列结构。最初，它与 V7一起提供，并出现在所有伯克利的版本中，也包含在伯克利的SVR4兼容函数库中。Seltzer和Yigit〔1991〕中有关于dbm函数库使用的动态散列算法历史的详细介绍，以及这个库的其他实现方法。但是，这些实现的一个致命缺点是它们都不支持多个进程对数据库的并发修改。它们都没有提供并发控制（如记录锁）。

4.3+BSD提供了一个新的库db(3)，这个库支持三种不同的访问方式：面向记录、散列和 B树。同样， db也没有提供并发控制（这一点在 db手册的BUGS栏中说得很清楚）。Seltzer和Olson〔1992〕中说以后的版本将提供像大部分商业数据库系统一样的并发控制功能。

绝大部分商业的数据库函数库提供多进程并发修改一个数据库所需要的并发控制。这些系统一般都使用12.3节中介绍的建议记录锁，并且用B+树来实现他们的数据库。

16.3　函数库

本节将定义数据库函数库的C语言接口，下一节再讨论其实现。

当打开一个数据库时，通过返回值得到一个 DB结构的指针。这一点很像通过 fopen得到一个FILE结构的指针（见5.2节），以及通过opendir得到一个DIR结构的指针（见4.21节）。我们将用此指针作为参数来调用以后的数据库函数。

```
#include "db.h"

DB     *db_open(const char *pathname, int oflag, int mode);
                                        返回：若成功则为DB结构的指针，若失败则为NULL

void   db_close(DB *db);
```

China-pub.com
下载

如果db_open成功返回，则将建立两个文件：*pathname.idx* 和*pathname.dat*，*pathname.idx*是索引文件，*pathname.dat*是数据文件。*oflag*被用作第二个参数传递给open（见3.3节），表明这些文件的打开模式（只读、读写或如果文件不存在则建立等）。如果需要建立新的数据库，*mode*将作为第三个参数传递给open（文件访问权限）。

当不再使用数据库时，调用db_close来关闭数据库。db_close将关闭索引文件和数据文件，并释放数据库使用过程中分配的所有用于内部缓冲的存储空间。

当向数据库中加入一条新的记录时，必须提供一个此记录的关键字，以及与此关键字相联系的数据。如果此数据库存储的是人事信息，关键字可以是雇员号，数据可以是此雇员的姓名、地址、电话号码、受聘日等等。我们的实现要求关键字必须是唯一的（比方说，不会有两个雇员记录有同样的雇员号）。

```
#include "db.h"

int db_store(DB *db, const char key, const char data, int flag);
```

返回：若成功则为0，若错误则为非0(见下)

*key*和*data*是由NULL结束的字符串。它们可以包含除了NULL外的任何字符，如换行符。

*flag*只能是DB_INSERT（加一条新记录）或DB_REPLACE（替换一条已有的记录）。这两个常数定义在db.h头文件中。如果使用DB_REPLACE，而记录不存在，则返回值为－1。如果使用DB_INSERT，而记录已经存在，则返回值为1。

通过提供关键字*key*可以从数据库中取出一条记录。

```
#include "db.h"

char *db_fetch(DB *db, const char key,);
```

返回：若成功则为指向数据的指针，若记录没有找到则为NULL

如果记录找到了，返回的指针指向与关键字联系在一起的数据。

通过提供关键字*key*，也可以从数据库中删除一条记录。

```
#include "db.h"

int db_delete(DB *db, const char key);
```

返回：若成功则为0，若记录没有找到则为-1

除了通过关键字访问数据库外，也可以一条一条记录地访问数据库。因此，首先调用db_rewind回到数据库的第一条记录，再调用db_nextrec顺序地读每个记录。

```
#include "db.h"

void db_rewind(DB *db)

char *db_nextrec(DB *db, char *key);
```

返回：若成功则返回指向数据的指针，若到达数据库的尾端则为NULL

如果*key*是非NULL的指针，db_nextrec将当前记录的关键字存入*key*中。

db_nextrec不保证记录访问的次序，只保证每一条记录被访问恰好一次。如果顺序存储三条关键字分别为A、B、C的记录，则无法确定db_nextrec将按什么顺序返回这三条记录。它可

China-pub.com 下载

能按B、A、C的顺序返回，也可能按其他顺序。实际的顺序由数据库的实现决定。

这七个函数提供了数据库函数库的接口。接下来介绍实现。

16.4 实现概述

大多数数据库访问的函数库使用两个文件来存储信息：一个索引文件和一个数据文件。索引文件包括索引值（关键字）和一个指向数据文件中对应数据记录的指针。有许多技术可用来组织索引文件以提高按关键字查询的速度和效率，散列表和 B+树是两种常用的技术。我们采用固定大小的散列表来组织索引文件结构，并采用链表法解决散列冲突。在介绍 db_open时，曾提到将建立两个文件：一个以.idx为后缀的索引文件和一个以.dat为后缀的数据文件。

我们将关键字和索引以NULL结尾的字符串形式存储——它们不能包含任一的二进制数据。有些数据库系统用二进制的形式存储数值数据（如用 1、2或4个字节存储一个整数）以节省空间，这样一来使函数复杂化，也使数据库文件在不同的平台间移植比较困难。比方说，网络上有两个系统使用不同的二进制格式存储整数，如果想要这两个系统都能够访问数据库就必须解决这个问题（今天不同体系结构的系统共享文件已经很常见了）。按照字符串形式存储所有的记录，包括关键字和数据，能使这一切变得简单。这确实会需要更多的磁盘空间，但随着磁盘技术的发展，这渐渐不再构成问题。

db_store要求对每个关键字，最多只有一个对应的记录。有些数据库系统允许多条记录使用同样的关键字，并提供方法访问与一个关键字相关的所有记录。另外，我们只有一个索引文件，这意味着每个数据记录只能有一个关键字。有些数据库允许一条记录拥有多个关键字，并且对每一个关键字使用一个索引文件。当加入或删除一条记录时，要对所有的索引文件进行相应的修改。（一个有多个索引的例子是雇员库文件，可以将雇员号作为关键字，也可以将雇员的社会保险号作为关键字。由于一般雇员的名字并不保证唯一，所以名字不能作为关键字。）

图16-1是数据库实现的基本结构。索引文件由三部分组成：空闲链表指针、散列表和索引记录。图16-1 *ptr*字段中实际存储的是以ASCII字符串形式记录的文件中的位移量。

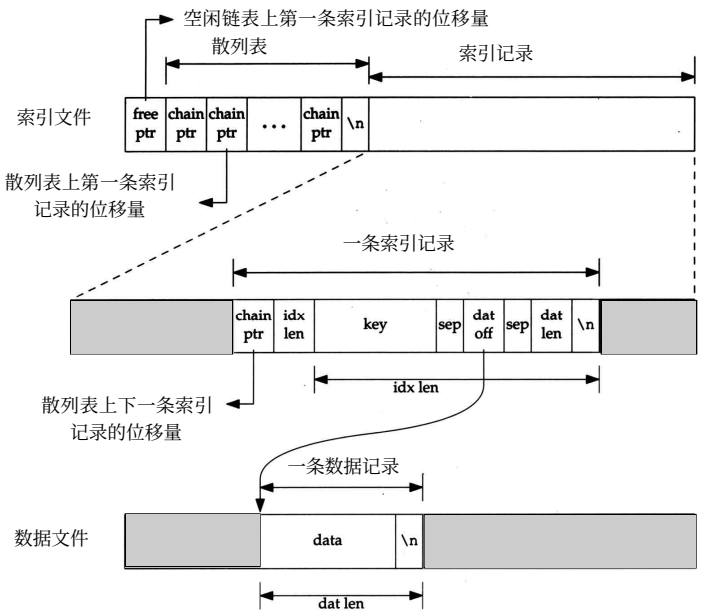

图16-1 索引文件和数据文件结构

China-pub.com
下载

当给定一个关键字要在数据库中寻找一条记录时， db_fetch根据关键字计算散列值，由此散列值可确定一条散列链（*chain ptr*(链表指针)字段可以为0，表示一条空的散列链）。沿着这条散列链，我们可以找到所有有同样散列值的索引记录。当遇到一个索引记录的 *chain ptr*字段为0时，表示到达了此散列链的末尾。

下面来看一个实际的数据库文件。程序16-1建立了一个新的数据库，并且加入了三条记录。由于所有的字段都以ASCII字符串的形式存储在数据库中，所以可以用任何标准的 UNIX工具来查看索引文件和数据文件。

```
$ ls -l db4.*
-rw-r--r--  1  stevens28 Oct 30 06:42 db4.dat
-rw-r--r--  1  stevens28 Oct 30 06:42 db4.idx
$ cat db4.idx
   0  53  35  0
   0  10Alpha:0:6
   0  10beta:6:14
   17 11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

程序16-1 建立一个数据库并向其写三条记录

```
#include    "db.h"

int
main(void)
{
    DB      *db;

    if ( (db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
                                            FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

为了使这个例子简单，将每个 *ptr*字段的大小定为4个ASCII字符，将散列表的大小（散列链的条数）定为3。由于每一个*ptr*记录的是一个文件位移量，所以4个ASCII字符限制了一个索引文件或数据文件的大小最多只能为10 000字节。16.8节做性能方面的测试时，将*ptr*字段的大小设为6（这样文件大小可以达到1 000 000字节），将散列表的大小设为100。

索引文件的第一行为

```
0  53  35  0
```

分别为空闲链表指针（0表示空闲链表为空），和三个散列链表的指针：53、35和0。下一行

```
0  10Alpha:0:6
```

显示了一条索引记录的结构。第一个4字符的字段（0）表示这一条记录是此散列链的最后一条。

China-pub.com 下载

下一个4字符的字段（10）为*idx len*，（索引记录长度）表示此索引记录剩下部分的长度。通过两个read操作来读取一条索引记录：第一个read读取这两个固定长度的字段（*chain ptr*和*idx len*），然后再根据*idx len*来读取后面的不定长部分。剩下的三个字段为：*key*（关键字）、*dat off*（数据记录的位移量）和*dat len*（数据记录的长度）。这三个字段用sep（分隔符）隔开，在这里使用的是分号。由于此三个字段都是不定长的，所以需要一个专门的分隔符，而且这个分隔符不能出现在关键字中。最后用一个\n（回车符）结束这一条索引记录。由于在*idx len*中已经有了记录的长度，所以这个回车符并不是必须的，加上回车符是为了把各条索引记录分开，这样就可以用标准的UNIX工具如cat和more来查看索引文件。key是将记录加入数据库时选择的关键字。数据记录在数据文件中的位移为0，长度为6。从数据文件中可看到数据记录确实从0开始，长度为6个字节。（与索引文件一样，这里自动在每条数据记录的后面加上一个回车符，以便于使用UNIX工具。在调用db_fetch时，此回车符不作为数据返回。）

如果在这个例子中跟踪三个散列链，可以看到第一条散列链上的第一条记录的位移量是 53（gamma）。这条链上下一条记录的位移量为17（Alpha），并且是这条链上的最后一条记录。第二条散列链上的第一条记录的位移量是 35（beta），且是此链上最后一条记录。第三条散列链为空。

请注意索引文件中索引记录的顺序和数据文件中对应数据记录的顺序与程序 16-1中调用db_store的顺序一样。由于在调用db_open时使用了O_TRUNC标志，索引文件和数据文件都被截断，整个数据库相当于从新初始化。在这种情形下， db_store将新的索引记录和数据记录添加到对应的文件末尾。后面将看到db_store也可以重复使用这两个文件中因删除记录而生成的空间。

在这里使用固定大小的散列表作为索引是一个妥协。当每个散列链均不太长时，这个方法能保证快速地查找。我们的目的是能够快速地查找任一的关键字，同时又不使用太复杂的数据结构，如B树或动态可扩充散列表。动态可扩充散列表的优点是能保证仅用两次磁盘操作就能找到数据记录（详见Selter和Yigit〔1991〕）。B树能够用关键字的顺序来遍历数据库（采用散列表的db_nextrec函数就做不到这一点）。

16.5 集中式或非集中式

当有多个进程访问数据库时，有两种方法可实现库函数：

(1) 集中式 由一个进程作为数据库管理者，所有的数据库访问工作由此进程完成。库函数通过IPC机制与此中心进程进行联系。

(2) 非集中式 每个库函数独立申请并发控制（加锁），然后调用它自己的I/O函数。

使用这两种技术的数据库都有。UNIX系统中的潮流是使用非集中式方法。如果有适当的加锁函数，因为避免使用了IPC，那么非集中式方法一般要快一些。图16-2描绘了集中式方法的操作。

图中特意表示出IPC像绝大多数UNIX的消息传送一样需要经过操作系统内核（14.9节的共享存储不需要这种经过内核的拷贝）。我们看到，在集中方式下，中心控制进程将记录读出，然后通过IPC机制将数据送给请求进程。注意到中心控制进程是唯一的通过I/O操作存取数据库文件的进程。

集中式的优点是能够根据需要来对操作模式进行控制。例如，可以通过中心进程给不同的进程赋予不同的优先级，而用非集中式方法则很难做到。在这种情况下只能依赖于操作内核的磁盘I/O调度策略和加锁策略（如当三个进程同时等待一个锁开锁时，哪个进程下一个得到锁）。

China-pub.com
下载

图16-3描绘了非集中式方法，本章的实现就是采用这种方法。使用库函数访问数据库的用户进程是平等的，它们通过使用记录锁机制来实现并发控制。

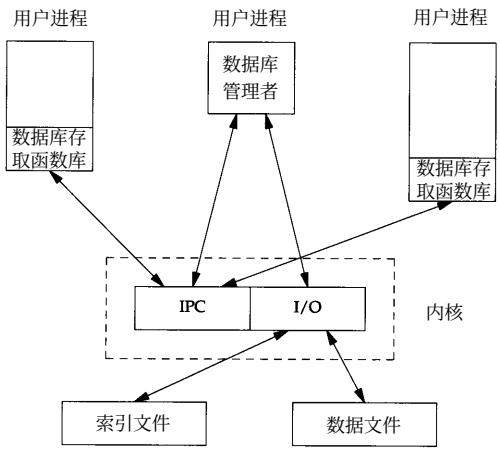

图16-2 集中式数据库访问

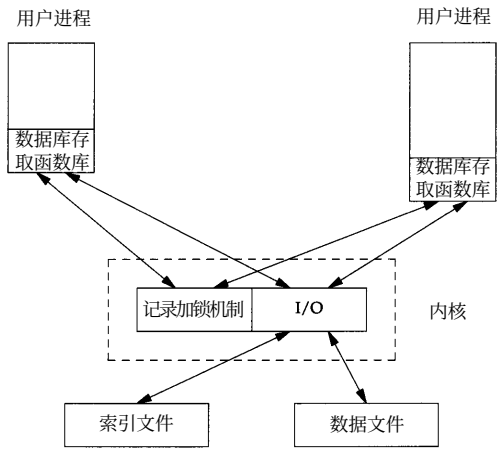

图16-3 非集中式数据库访问

16.6 并发

由于很多系统的实现都采用两个文件（一个索引文件和一个数据文件）的方法，所以我们也使用这种方法，这要求我们能够控制对两个文件的加锁。有很多方法可用来对两个文件进行加锁。

16.6.1 粗锁

最简单的加锁方法是将这两个文件中的一个作为锁，并要求调用者在对数据库进行操作前必须获得这个锁。我们将这称为粗锁（coarse locking）。比方说，可以认为一个进程对索引文件的0字节加了读锁后，才能获得读整个数据库的权力。一个进程对索引文件的 0字节加了写锁后，才获得修改整个数据库的权力。可以使用 UNIX的记录锁机制来控制每次可以有多个读进程，而只能由一个写进程（见表 12-2）。db_fetch和db_nextrec函数将获得读锁，而 db_delete、db_store以及db_open则获得写锁。（db_open获得写锁的原因是如果要创建新文件的话，要在索引文件前端建立空闲区链表以及散列链表。）

粗锁的问题是它限制了最大程度的并发。用粗锁时，当一个进程向一条散列链中加一条记录时，其他进程无法访问另一条散列链上的记录。

16.6.2 细锁

下面用称为细锁（fine locking）的方法来加强粗锁以提高并发度。我们要求一个读进程或写进程在操作一条记录前必须先获得此记录所在散列链的读锁或写锁。允许对一条散列链同时可以有多个读进程，而只能有一个写进程。另外，一个写进程在操作空闲区链表（如 db_delete或db_store）前，必须获得空闲区链表的写锁。最后，当 db_store向索引文件或数据文件加一条新记录时，必须获得对应文件相应区域的写锁。

我们期望细锁能比粗锁提供更高的并发度。 16.8节将给出一些实际的比较测试结果。 16.7

China-pub.com 下载

节给出了采用细锁的实现，并详细讨论了锁的实现（粗锁是实现的简化）。

在源文件中，直接调用了read，readv，write和writev。没有使用标准I/O函数库。虽然使用标准I/O函数库也可以使用记录锁，但是需要非常复杂的缓存管理。我们不希望例如 fgets返回的数据是10分钟之前读入标准I/O缓存的，而被另一个进程在5分钟之前修改了。

我们对并发讨论依据的是数据库函数库的简单的需求。商业系统一般有更多的需要。关于并发更多的细节可以参见Data[1982]的第3章。

16.7 源码

我们从程序16-2中的头文件db.h开始。所有函数以及调用此函数库的用户进程都包含这一头文件。

程序16-2 db.h头文件

```
#include    <sys/types.h>
#include    <sys/stat.h>    /* open() & db_open() mode */
#include    <fcntl.h>       /* open() & db_open() flags */
#include    <stddef.h>      /* NULL */
#include    "ourhdr.h"

        /* flags for db_store() */
#define DB_INSERT   1       /* insert new record only */
#define DB_REPLACE  2       /* replace existing record */

        /* magic numbers */
#define IDXLEN_SZ       4   /* #ascii chars for length of index record */
#define IDXLEN_MIN      6   /* key, sep, start, sep, length, newline */
#define IDXLEN_MAX   1024   /* arbitrary */
#define SEP           ':'   /* separator character in index record */
#define DATLEN_MIN      2   /* data byte, newline */
#define DATLEN_MAX   1024   /* arbitrary */

        /* following definitions are for hash chains and free list chain
           in index file */
#define PTR_SZ          6   /* size of ptr field in hash chain */
#define PTR_MAX    999999   /* max offset (file size) = 10**PTR_SZ - 1 */
#define NHASH_DEF     137   /* default hash table size */
#define FREE_OFF        0   /* offset of ptr to free list in index file */
#define HASH_OFF   PTR_SZ   /* offset of hash table in index file */

typedef struct {    /* our internal structure */
  int   idxfd;  /* fd for index file */
  int   datfd;  /* fd for data file */
  int   oflag;  /* flags for open()/db_open(): O_xxx */

 char  *idxbuf;/* malloc'ed buffer for index record */
 char  *datbuf;/* malloc'ed buffer for data record*/
 char  *name;  /* name db was opened under */
 off_t idxoff; /* offset in index file of index record */
               /* actual key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
 size_t idxlen;/* length of index record */
               /* excludes IDXLEN_SZ bytes at front of index record */
               /* includes newline at end of index record */
 off_t datoff; /* offset in data file of data record */
 size_t datlen;/* length of data record */
               /* includes newline at end */
 off_t ptrval; /* contents of chain ptr in index record */
 off_t ptroff; /* offset of chain ptr that points to this index record */
 off_t chainoff;/* offset of hash chain for this index record */
 off_t hashoff;/* offset in index file of hash table */
```

China-pub.com
下载


```
  int   nhash;  /* current hash table size */
  long  cnt_delok;  /* delete OK */
  long  cnt_delerr; /* delete error */
  long  cnt_fetchok;/* fetch OK */
  long  cnt_fetcherr;/* fetch error */
  long  cnt_nextrec;/* nextrec */
  long  cnt_stor1;  /* store: DB_INSERT, no empty, appended */
  long  cnt_stor2;  /* store: DB_INSERT, found empty, reused */
  long  cnt_stor3;  /* store: DB_REPLACE, diff data len, appended */
  long  cnt_stor4;  /* store: DB_REPLACE, same data len, overwrote */
  long  cnt_storerr;/* store error */
} DB;

typedef unsigned long   hash_t; /* hash values */

            /* user-callable functions */
DB      *db_open(const char *, int, int);
void     db_close(DB *);
char    *db_fetch(DB *, const char *);
int      db_store(DB *, const char *, const char *, int);
int      db_delete(DB *, const char *);
void     db_rewind(DB *);
char    *db_nextrec(DB *, char *);
void     db_stats(DB *);

            /* internal functions */
DB      *_db_alloc(int);
int      _db_checkfree(DB *);
int      _db_dodelete(DB *);
int      _db_emptykey(char *);
int      _db_find(DB *, const char *, int);
int      _db_findfree(DB *, int, int);
int      _db_free(DB *);
hash_t   _db_hash(DB *, const char *);
char    *_db_nextkey(DB *);
char    *_db_readdat(DB *);
off_t    _db_readidx(DB *, off_t);

off_t    _db_readptr(DB *, off_t);
void     _db_writedat(DB *, const char *, off_t, int);
void     _db_writeidx(DB *, const char *, off_t, int, off_t);
void     _db_writeptr(DB *, off_t, off_t);
```

该程序定义了实现的基本限制。如果要支持更大的数据库的话，这些限制也可以修改。其中一些定义为常数的值也可定义为变量，只是会使实现复杂一些。例如，设定散列表的大小为137，一个也许更好的方法是让db_open的调用者根据数据库的大小通过参数来设定这个值，然后将这个值存在索引文件的最前面。

在DB结构中记录一个打开的数据库的所有信息。 db_open函数返回一个DB结构的指针，这个指针被用于其他所有函数。

选择用db_开头来命名用户可调用的库函数，用_db_开头来命名内部函数。

程序16-3中定义了函数db_open。它打开索引文件和数据文件，必要的话初始化索引文件。通过调用_db_alloc来为DB结构分配空间，并初始化此结构。

程序16-3 db_open函数

```
#include    "db.h"

/* Open or create a database.  Same arguments as open(). */

DB *
```

China-pub.com 下载

```
db_open(const char *pathname, int oflag, int mode)
{
    DB          *db;
    int         i, len;
    char        asciiptr[PTR_SZ + 1],
                hash[(NHASH_DEF + 1) * PTR_SZ + 2];
                    /* +2 for newline and null */
    struct stat statbuff;

        /* Allocate a DB structure, and the buffers it needs */
    len = strlen(pathname);
    if ( (db = _db_alloc(len)) == NULL)
        err_dump("_db_alloc error for DB");

    db->oflag = oflag;      /* save a copy of the open flags */

        /* Open index file */
    strcpy(db->name, pathname);
    strcat(db->name, ".idx");
    if ( (db->idxfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }
        /* Open data file */
    strcpy(db->name + len, ".dat");
    if ( (db->datfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }

        /* If the database was created, we have to initialize it */
    if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
            /* Write lock the entire file so that we can stat
               the file, check its size, and initialize it,
               as an atomic operation. */
        if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

        if (fstat(db->idxfd, &statbuff) < 0)
            err_sys("fstat error");
        if (statbuff.st_size == 0) {
                /* We have to build a list of (NHASH_DEF + 1) chain
                   ptrs with a value of 0.  The +1 is for the free
                   list pointer that precedes the hash table. */
            sprintf(asciiptr, "%*d", PTR_SZ, 0);
            hash[0] = 0;
            for (i = 0; i < (NHASH_DEF + 1); i++)
                strcat(hash, asciiptr);
            strcat(hash, "\n");

            i = strlen(hash);
            if (write(db->idxfd, hash, i) != i)
                err_dump("write error initializing index file");
        }
        if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
    }
    db->nhash    = NHASH_DEF;/* hash table size */
    db->hashoff = HASH_OFF; /* offset in index file of hash table */
                            /* free list ptr always at FREE_OFF */
    db_rewind(db);

    return(db);
}
```

China-pub.com
下载

如果数据库正被建立，则必须加锁。考虑两个进程试图同时建立同一个数据库的情况。第一个进程运行到调用fstat，并且在fstat返回后被内核切换。这时第二个进程调用db_open，发现索引文件的长度为0，并初始化空闲链表和散列链表。第二个进程继续运行，向数据库中添加了一条记录。这时第二个进程被阻塞，第一个进程继续运行，并发现索引文件的大小为 0（因为第一个进程是在 fstat返回后才被切换），所以第一个进程重新初始化空闲链表和散列链表，第二个进程写入的记录就被抹去了。要避免发生这种情况的方法是进行加锁，可以使用 12.3节中的readw_lock，writew_lock和un_lock这三个函数。

db_open调用程序16-4中定义的函数_db_alloc来为DB结构分配空间，包括一个索引缓存和一个数据缓存。

程序16-4 _db_alloc函数

```
#include    "db.h"

/* Allocate & initialize a DB structure, and all the buffers it needs  */

DB *
_db_alloc(int namelen)
{
    DB      *db;

            /* Use calloc, to init structure to zero */
    if ( (db = calloc(1, sizeof(DB))) == NULL)
        err_dump("calloc error for DB");

    db->idxfd = db->datfd = -1;             /* descriptors */

        /* Allocate room for the name.
           +5 for ".idx" or ".dat" plus null at end. */

    if ( (db->name = malloc(namelen + 5)) == NULL)
        err_dump("malloc error for name");

        /* Allocate an index buffer and a data buffer.
           +2 for newline and null at end. */

    if ( (db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
        err_dump("malloc error for index buffer");
    if ( (db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
        err_dump("malloc error for data buffer");

    return(db);
}
```

索引缓存和数据缓存的大小在db.h中定义。数据库函数库可以通过让这些缓存按需要动态扩张来得到加强。其方法可以是记录这两个缓存的大小，然后在需要更大的缓存时调用realloc。

在函数_db_free（见程序16-5）中，这些缓存将被释放，同时打开的文件被关闭。 db_open在打开索引文件和数据文件时如果遇到错误，则调用_db_free释放资源。db_close（见程序16-6）也调用_db_free。

函数db_fetch（见程序16-7）根据给定的关键字来读取一条记录。它调用 _db_find在数据库中查找一条索引记录，如果找到，再调用 _db_readdat来读取对应的数据记录。

程序16-5 _db_free函数

```
#include    "db.h"

/* Free up a DB structure, and all the malloc'ed buffers it
```

China-pub.com 下载

```
 * may point to.  Also close the file descriptors if still open. */
int
_db_free(DB *db)
{
    if (db->idxfd >= 0 && close(db->idxfd) < 0)
        err_dump("index close error");
    if (db->datfd >= 0 && close(db->datfd) < 0)
        err_dump("data close error");
    db->idxfd = db->datfd = -1;

    if (db->idxbuf != NULL)
        free(db->idxbuf);
    if (db->datbuf != NULL)
        free(db->datbuf);
    if (db->name != NULL)
        free(db->name);
    free(db);
    return(0);
}
```

程序16-6　db_close函数

```
#include    "db.h"

void
db_close(DB *db)
{
    _db_free(db);    /* closes fds, free buffers & struct */
}
```

程序16-7　db_fetch函数

```
#include    "db.h"

/* Fetch a specified record.
 * We return a pointer to the null-terminated data. */

char *
db_fetch(DB *db, const char *key)
{
    char    *ptr;

    if (_db_find(db, key, 0) < 0) {
        ptr = NULL;             /* error, record not found */
        db->cnt_fetcherr++;
    } else {
        ptr = _db_readdat(db);  /* return pointer to data */
        db->cnt_fetchok++;
    }
            /* Unlock the hash chain that _db_find() locked */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(ptr);
}
```

函数_db_find（见程序16-8）通过遍历散列链来查找记录。db_fetch，db_delete和db_store这几个需要根据关键字查找记录的函数都调用它。

China-pub.com
下载

程序16-8 _db_find函数

```
#include    "db.h"

/* Find the specified record.
 * Called by db_delete(), db_fetch(), and db_store(). */

int
_db_find(DB *db, const char *key, int writelock)
{
    off_t   offset, nextoffset;

        /* Calculate hash value for this key, then calculate byte
           offset of corresponding chain ptr in hash table.
           This is where our search starts. */

            /* calc offset in hash table for this key */
    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
    db->ptroff = db->chainoff;

            /* Here's where we lock this hash chain.  It's the
               caller's responsibility to unlock it when done.
               Note we lock and unlock only the first byte. */
    if (writelock) {
        if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("writew_lock error");
    } else {
        if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("readw_lock error");
    }
            /* Get the offset in the index file of first record
               on the hash chain (can be 0) */
    offset = _db_readptr(db, db->ptroff);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strcmp(db->idxbuf, key) == 0)
            break;      /* found a match */

        db->ptroff = offset;    /* offset of this (unequal) record */
        offset = nextoffset;    /* next one to compare */
    }
    if (offset == 0)
        return(-1);     /* error, record not found */

        /* We have a match.  We're guaranteed that db->ptroff contains
           the offset of the chain ptr that points to this matching
           index record.  _db_dodelete() uses this fact.  (The chain
           ptr that points to this matching record could be in an
           index record or in the hash table.) */
    return(0);
}
```

_db_find的最后一个参数指明需要加什么样的锁，0表示读锁，1表示写锁。我们知道，db_fetch需要加读锁，而db_delete和db_store需要加写锁。_db_find在获得需要的锁之前将等待。

_db_find中的while循环遍历散列链中的索引记录，并比较关键字。函数 _db_readidx用于读取每条索引记录。

请注意阅读_db_find中的最后一条注释。当沿着散列链进行遍历时，必须始终跟踪当前索引记录的前一条索引记录，其中有一个指针指向当前记录。这一点在删除一条记录时很有用，因为必须修改当前索引记录的前一条记录的链指针。

China-pub.com 下载

先来看看_db_find调用的一些比较简单的函数。_db_hash（见程序16-9）根据给定的关键字计算散列值。它将关键字中的每一个ASCII字符乘以这个字符在字符串中以1开始的索引号，将这些结果加起来，除以散列表的大小，将余数作为这个关键字的散列值。

程序16-9 _db_hash函数

```
#include    "db.h"

/* Calculate the hash value for a key. */

hash_t
_db_hash(DB *db, const char *key)
{
    hash_t      hval;
    const char  *ptr;
    char        c;
    int         i;

    hval = 0;
    for (ptr = key, i = 1; c = *ptr++; i++)
        hval += c * i;      /* ascii char times its 1-based index */

    return(hval % db->nhash);
}
```

_db_find调用的下一个函数是_db_readptr（见程序16-10）。这个函数能够读取以下三种不同的链表指针中的任意一种：(1) 索引文件最开始的空闲链表指针，(2) 散列表中指向散列链的第一条索引记录的指针，(3) 每条索引记录头的指向下一条记录的指针（这里的索引记录既可以处于一条散列链表中，也可以处于空闲链表中）。这个函数的调用者应做好必要的加锁，此函数不进行任何加锁。

程序16-10 _db_readptr函数

```
#include    "db.h"

/* Read a chain ptr field from anywhere in the index file:
 * the free list pointer, a hash table chain ptr, or an
 * index record chain ptr.  */

off_t
_db_readptr(DB *db, off_t offset)
{
    char    asciiptr[PTR_SZ + 1];

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("read error of ptr field");

    asciiptr[PTR_SZ] = 0;       /* null terminate */
    return(atol(asciiptr));
}
```

_db_find中的while循环通过调用_db_readidx来读取各条索引记录。_db_readidx（见程序16-11）是一个较长的函数，这个函数读取索引记录，并将索引记录的信息存储到适当的字段中。

程序16-11 _db_readidx函数

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */
```

China-pub.com
下载


```
/* Read the next index record.  We start at the specified offset in
 * the index file.  We read the index record into db->idxbuf and
 * replace the separators with null bytes.  If all is OK we set
 * db->datoff and db->datlen to the offset and length of the
 * corresponding data record in the data file.  */

off_t
_db_readidx(DB *db, off_t offset)
{
    int             i;
    char            *ptr1, *ptr2;
    char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
    struct iovec    iov[2];

        /* Position index file and record the offset.  db_nextrec()
           calls us with offset==0, meaning read from current offset.
           We still need to call lseek() to record the current offset. */
if ( (db->idxoff = lseek(db->idxfd, offset,
                         offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
    err_dump("lseek error");

    /* Read the ascii chain ptr and the ascii length at
       the front of the index record.  This tells us the
       remaining size of the index record. */
iov[0].iov_base = asciiptr;
iov[0].iov_len  = PTR_SZ;
iov[1].iov_base = asciilen;
iov[1].iov_len  = IDXLEN_SZ;
if ( (i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
    if (i == 0 && offset == 0)
        return(-1);     /* EOF for db_nextrec() */
    err_dump("readv error of index record");
}

asciiptr[PTR_SZ] = 0;           /* null terminate */
db->ptrval = atol(asciiptr);    /* offset of next key in chain */
                    /* this is our return value; always >= 0 */
asciilen[IDXLEN_SZ] = 0;        /* null terminate */
if ( (db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
                       db->idxlen > IDXLEN_MAX)
    err_dump("invalid length");

    /* Now read the actual index record.  We read it into the key
       buffer that we malloced when we opened the database. */
if ( (i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
    err_dump("read error of index record");
if (db->idxbuf[db->idxlen-1] != '\n')
    err_dump("missing newline");    /* sanity checks */
db->idxbuf[db->idxlen-1] = 0;       /* replace newline with null */

    /* Find the separators in the index record */
if ( (ptr1 = strchr(db->idxbuf, SEP)) == NULL)
    err_dump("missing first separator");
*ptr1++ = 0;                /* replace SEP with null */

if ( (ptr2 = strchr(ptr1, SEP)) == NULL)
    err_dump("missing second separator");
*ptr2++ = 0;                /* replace SEP with null */

if (strchr(ptr2, SEP) != NULL)
    err_dump("too many separators");

    /* Get the starting offset and length of the data record */
if ( (db->datoff = atol(ptr1)) < 0)
    err_dump("starting offset < 0");
if ( (db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
    err_dump("invalid length");
```

China-pub.com 下载

```
    return(db->ptrval);     /* return offset of next key in chain */
}
```

我们调用readv来读取索引记录开始处的两个固定长度的字段：指向下一条索引记录的链表指针和索引记录剩下的不定长部分的长度。然后，索引记录剩下的部分被读入：关键字、数据记录的位移量和数据记录的长度。我们并不读数据记录，这由调用者自己完成。例如，在db_fetch中，在_db_find按关键字找到索引记录前是不去读取数据记录的。

下面回到db_fetch。如果_db_find找到了索引记录，则调用 _db_readdat来读取对应的数据记录。这是一个很简单的函数（见程序16-12）。

程序16-12 _de_readdat函数

```
#include    "db.h"

/* Read the current data record into the data buffer.
 * Return a pointer to the null-terminated data buffer. */

char *
_db_readdat(DB *db)
{
    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
        err_dump("lseek error");

    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
        err_dump("read error");
    if (db->datbuf[db->datlen - 1] != '\n')     /* sanity check */
        err_dump("missing newline");
    db->datbuf[db->datlen - 1] = 0;     /* replace newline with null */

    return(db->datbuf);     /* return pointer to data record */
}
```

我们从db_fetch开始，现在已经读取了索引记录和对应的数据记录。注意到，只在_db_find中加了一个读锁。由于对这条散列链加了读锁，所以其他进程在这段时间内不可能对这条散列链进行修改。

下面来看函数db_delete（见程序16-13）。它开始时和db_fetch一样，调用_db_find来查找记录，只是这里传递给_db_find的最后一个参数为1，表示要对这一条散列链加写锁。

db_delete调用_db_dodelete（见程序16-14）来完成剩下的工作（在后面将看到 db_store也调用_db_dodelete）。_db_dodelete的大部分工作是修正空闲链表以及与关键字对应的散列链。

当一条记录被删除后，将其关键字和数据记录设为空。本章后面将提到的函数 db_nextrec要用到这一点。

_db_dodelete对空闲链表加写锁，这样能防止两个进程同时删除不同链表上的记录时产生相互影响，因为我们将被删除的记录移到空闲链表上，这将改变空闲链表，而一次只能有一个进程能这样做。

程序16-13 db_delete函数

```
#include    "db.h"

/* Delete the specified record */

int
db_delete(DB *db, const char *key)
```

China-pub.com
下载


```
{
    int     rc;

    if (_db_find(db, key, 1) == 0) {
        rc = _db_dodelete(db);  /* record found */
        db->cnt_delok++;
    } else {
        rc = -1;                /* not found */
        db->cnt_delerr++;
    }

    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

程序16-14 _db_dodelete函数

```
#include    "db.h"

/* Delete the current record specified by the DB structure.
 * This function is called by db_delete() and db_store(),
 * after the record has been located by _db_find(). */

int
_db_dodelete(DB *db)
{
    int     i;
    char    *ptr;
    off_t   freeptr, saveptr;

        /* Set data buffer to all blanks */
    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
        *ptr++ = ' ';
    *ptr = 0;   /* null terminate for _db_writedat() */

        /* Set key to blanks */
    ptr = db->idxbuf;
    while (*ptr)
        *ptr++ = ' ';

        /* We have to lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");

    /* Write the data record with all blanks */
_db_writedat(db, db->datbuf, db->datoff, SEEK_SET);

    /* Read the free list pointer.  Its value becomes the
       chain ptr field of the deleted index record.  This means
       the deleted record becomes the head of the free list. */
freeptr = _db_readptr(db, FREE_OFF);

    /* Save the contents of index record chain ptr,
       before it's rewritten by _db_writeidx(). */
saveptr = db->ptrval;

    /* Rewrite the index record.  This also rewrites the length
       of the index record, the data offset, and the data length,
       none of which has changed, but that's OK. */
_db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

    /* Write the new free list pointer */
_db_writeptr(db, FREE_OFF, db->idxoff);

    /* Rewrite the chain ptr that pointed to this record
```

China-pub.com 下载

```
            being deleted.  Recall that _db_find() sets db->ptroff
            to point to this chain ptr.  We set this chain ptr
            to the contents of the deleted record's chain ptr,
            saveptr, which can be either zero or nonzero. */
        _db_writeptr(db, db->ptroff, saveptr);

        if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
            err_dump("un_lock error");

        return(0);
    }
```

_db_dodelete通过调用函数_db_writedat（见程序16-15）清空数据记录。这时_db_writedat并不对数据文件加写锁，这是因为db_delete对这条记录索引的散列链加了写锁，这已经保证不会有其他进程能够读写这条记录。本章后面讲述 db_store时，将看到在另一种情况下，_db_writedat将数据添加到文件的末尾，并加锁。

_db_writedat调用writev来写数据记录及回车符。不能够假设调用者传递进来的字符缓存有空间让我们在其后面再添加一个回车符。回忆12.7节，在那里曾讨论过一个writev要比两个write快。

接着_db_dodelete修改索引记录。让这条记录的链表指针指向空闲链表的第一条记录（如果空闲链表为空，则这个链表指针置为0），空闲链表指针也被修改，指向当前删除的这条记录，这样就将这条删除的记录移到了空闲链表首。空闲链表实际上很象一个先进先出的堆栈。

程序16-15 _db_writedat函数

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */

/* Write a data record.  Called by _db_dodelete() (to write
   the record with blanks) and db_store(). */

void
_db_writedat(DB *db, const char *data, off_t offset, int whence)
{
    struct iovec    iov[2];
    static char     newline = '\n';

        /* If we're appending, we have to lock before doing the lseek()
           and write() to make the two an atomic operation.  If we're
           overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END)      /* we're appending, lock entire file */
        if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

    if ( (db->datoff = lseek(db->datfd, offset, whence)) == -1)
        err_dump("lseek error");
    db->datlen = strlen(data) + 1;  /* datlen includes newline */

    iov[0].iov_base = (char *) data;
    iov[0].iov_len  = db->datlen - 1;
    iov[1].iov_base = &newline;
    iov[1].iov_len  = 1;
    if (writev(db->datfd, &iov[0], 2) != db->datlen)
        err_dump("writev error of data record");

    if (whence == SEEK_END)
        if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}
```

China-pub.com
下载

我们并没有为索引文件和数据文件的空闲记录设置各自的空闲链表。这是因为当一条记录被删除时，对应的索引记录被加入了空闲链表，而这条索引记录中有指向数据文件中数据记录的指针。有其他很多更好的处理记录删除的方法，只是它们会更复杂一些。

程序16-16列出了函数 _db_writeidx，_db_dodelete调用此函数来写一条索引记录。和_db_writedat一样，这一函数也只有在添加新索引记录时才需要加锁。 _db_dodelete调用此函数是为了重写一条索引记录，在这种情况下调用者已经在散列链上加了写锁，所以不再需要加另外的锁。

程序16-16 _db_writeidx函数

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */

/* Write an index record.
 * _db_writedat() is called before this function, to set the fields
 * datoff and datlen in the DB structure, which we need to write
 * the index record. */

void
_db_writeidx(DB *db, const char *key,
                     off_t offset, int whence, off_t ptrval)
{
    struct iovec    iov[2];
    char            asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
    int             len;

    if ( (db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);

    sprintf(db->idxbuf, "%s%c%d%c%d\n",
                key, SEP, db->datoff, SEP, db->datlen);
    if ( (len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
        err_dump("invalid length");
    sprintf(asciiptrlen, "%*d%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

        /* If we're appending, we have to lock before doing the lseek()
           and write() to make the two an atomic operation.  If we're
           overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END)      /* we're appending */
        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
                                                SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

        /* Position the index file and record the offset */
    if ( (db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
        err_dump("lseek error");

    iov[0].iov_base = asciiptrlen;
    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
    iov[1].iov_base = db->idxbuf;
    iov[1].iov_len  = len;
    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
        err_dump("writev error of index record");

    if (whence == SEEK_END)
        if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}
```

_db_dodelete调用的最后一个函数是 _db_writeptr（见程序16-17）。它被调用两次——一次

China-pub.com 下载

用来写空闲链表指针，一次用来写散列链表指针（指向被删除索引记录的指针）。

程序16-17 _db_writeptr函数

```
#include    "db.h"

/* Write a chain ptr field somewhere in the index file:
 * the free list, the hash table, or in an index record. */

void
_db_writeptr(DB *db, off_t offset, off_t ptrval)
{
    char    asciiptr[PTR_SZ + 1];

    if (ptrval < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);
    sprintf(asciiptr, "%*d", PTR_SZ, ptrval);

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("write error of ptr field");
}
```

程序16-18中列出了最长的数据库函数db_store。它从调用_db_find开始，以查看这个记录是否存在。如果记录存在且标志为DB_REPLACE，或记录不存在且标志为DB_INSERT，这些都是允许的。替换一条已存在的记录，指的是该记录的关键字一样，而数据很可能不一样。

注意到因为db_store可能会改变散列链，所以调用_db_find的最后一条参数说明要对散列链加写锁。

如果要加一条新记录到数据库中，则调用函数_db_findfree（见程序16-19）在空闲链表中搜索一个有同样关键字大小和同样数据大小的已删除的记录。

_db_findfree中的while循环遍历空闲链表以搜寻一个有对应关键字大小和数据大小的索引记录项。在这个简单的实现中，只有当一个已删除记录的关键字大小及数据大小与新加入的记录的关键字大小及数据大小一样时才重用已删除的记录的空间。其他更好的算法一般更复杂。

_db_findfree需要对空闲链表加写锁以避免与其他使用空闲链表的进程互相影响。当一条记录从空闲链表上取下来后，就可以打开这个写锁。_db_dodelete也要修改空闲链表。

程序16-18 db_store函数

```
#include    "db.h"

/* Store a record in the database.
 * Return 0 if OK, 1 if record exists and DB_INSERT specified,
 * -1 if record doesn't exist and DB_REPLACE specified. */

int
db_store(DB *db, const char *key, const char *data, int flag)
{
    int     rc, keylen, datlen;
    off_t   ptrval;

    keylen = strlen(key);
    datlen = strlen(data) + 1;      /* +1 for newline at end */
    if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
        err_dump("invalid data length");
```

China-pub.com
下载


```
        /* _db_find() calculates which hash table this new record
           goes into (db->chainoff), regardless whether it already
           exists or not.  The calls to _db_writeptr() below
           change the hash table entry for this chain to point to
           the new record.  This means the new record is added to
           the front of the hash chain. */

    if (_db_find(db, key, 1) < 0) {     /* record not found */
        if (flag & DB_REPLACE) {
            rc = -1;
            db->cnt_storerr++;
            goto doreturn;      /* error, record does not exist */
        }

            /* _db_find() locked the hash chain for us; read the
               chain ptr to the first index record on hash chain */
        ptrval = _db_readptr(db, db->chainoff);

        if (_db_findfree(db, keylen, datlen) < 0) {
                /* An empty record of the correct size was not found.
                   We have to append the new record to the ends of
                   the index and data files */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);
                /* db->idxoff was set by _db_writeidx().  The new
                   record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor1++;
        } else {
                /* We can reuse an empty record.
                   _db_findfree() removed the record from the free
                   list and set both db->datoff and db->idxoff. */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
                /* reused record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor2++;
        }

    } else {                        /* record found */
        if (flag & DB_INSERT) {
            rc = 1;
            db->cnt_storerr++;
            goto doreturn;      /* error, record already in db */
        }

            /* We are replacing an existing record.  We know the new
               key equals the existing key, but we need to check if
               the data records are the same size. */
        if (datlen != db->datlen) {
            _db_dodelete(db);   /* delete the existing record */

                /* Reread the chain ptr in the hash table
                   (it may change with the deletion). */
            ptrval = _db_readptr(db, db->chainoff);

                /* append new index and data records to end of files */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);
                /* new record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor3++;
```

China-pub.com 下载

```
        } else {
                /* same size data, just replace data record */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            db->cnt_stor4++;
        }
    }
    rc = 0;      /* OK */
doreturn:   /* unlock the hash chain that _db_find() locked */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

程序16-19 _db_findfree函数

```
#include    "db.h"

/* Try to find a free index record and accompanying data record
 * of the correct sizes.  We're only called by db_store(). */

int
_db_findfree(DB *db, int keylen, int datlen)
{
    int     rc;
    off_t   offset, nextoffset, saveoffset;

        /* Lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");

        /* Read the free list pointer */
    saveoffset = FREE_OFF;
    offset = _db_readptr(db, saveoffset);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
            break;      /* found a match */

        saveoffset = offset;
        offset = nextoffset;
    }

    if (offset == 0)
        rc = -1;    /* no match found */
    else {
            /* Found a free record with matching sizes.
               The index record was read in by _db_readidx() above,
               which sets db->ptrval.  Also, saveoffset points to
               the chain ptr that pointed to this empty record on
               the free list.  We set this chain ptr to db->ptrval,
               which removes the empty record from the free list. */

        _db_writeptr(db, saveoffset, db->ptrval);
        rc = 0;

            /* Notice also that _db_readidx() set both db->idxoff
               and db->datoff.  This is used by the caller, db_store(),
               to write the new index record and data record. */
    }
            /* Unlock the free list */
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

China-pub.com
下载

回到函数db_store，在调用_db_find后，有四种可能：

(1) 加入一条新的记录，而_db_findfree没有找到对应大小的空闲记录。这意味着要将这条新记录添加到索引文件和数据文件的末尾。通过调用 _db_writeptr将新记录加到对应的散列链的链首。

(2) 加入一条新记录，且 _db_findfree找到对应大小的空闲记录。这条空闲记录被_db_findfree从空闲链表上移下来，新的索引记录和数据记录被写入，然后通过调用_db_writeptr将新记录加到对应的散列链的链首。

(3) 要替换一条记录，而新数据记录的长度与已存在的记录的长度不一样。调用_db_dodelete将老记录删除，然后将新记录添加到索引文件和数据文件的末尾（也可以用其他方法，如可以再找一找是否有适当大小的已删除的记录项）。最后调用_db_writeptr将新记录加到对应的散列链的链首。

(4) 要替换一条记录，而新数据记录的长度与已存在的记录的长度恰好一样。这是最容易的情况，只需要重写记录即可。

在向文件的末尾添加索引记录或数据记录时，需要加锁（回忆在程序 12-6中遇到的相对文件尾加锁问题）。在第(1)和第(3)种情况中，db_store调用_db_writeidx和_db_writedat时，第3个参数为0，第4个参数为SEEK_END。这里，第4个参数作为一个标志用来告诉这两个函数，新的记录将被添加到文件的末尾。 _db_writeidx用到的技术是对索引文件加写锁，加锁的范围从散列链的末尾到文件的末尾。这不会影响其他数据库的读用户和写用户（这些用户将对散列链加锁），但如果其他用户此时调用db_store来添加数据则会被锁住。_db_writedat使用的方法是对整个数据文件加写锁。同样这也不会影响其他数据库的读用户和写用户（它们甚至不对数据文件加锁），但如果其他用户此时调用db_store来向数据文件添加数据则会被锁住（见习题16.3）。

最后两个函数是db_nextrec和db_rewind，这两个函数用来读取数据库的所有记录。通常在下列形式的循环中的使用这两个函数：

```
db_rewind(db) ;
while( (ptr = db_nextrec(db, key)) != NULL) {
     /* process record */
}
```

前面曾警告过，记录的返回没有一定的次序——它们并不按关键字的顺序。

函数db_rewind（见程序16-20）将索引文件定位在第一条索引记录（紧跟在散列表后面）。

程序16-20 db_rewind函数

```
#include    "db.h"

/* Rewind the index file for db_nextrec().
 * Automatically called by db_open().
 * Must be called before first db_nextrec().
 */

void
db_rewind(DB *db)
{
    off_t   offset;

    offset = (db->nhash + 1) * PTR_SZ;      /* +1 for free list ptr */

        /* We're just setting the file offset for this process
           to the start of the index records; no need to lock.
```

China-pub.com 下载

```
            +1 below for newline at end of hash table. */

    if ( (db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
        err_dump("lseek error");
}
```

当db_rewind定位好索引文件后，db_nextrec一条一条按顺序读取索引记录。在程序 16-21 中可以看到，db_nextrec并不使用散列链表。由于db_nextrec除了读取散列链表中的记录外，还读取已删除的记录，所以必须检查记录是否是已被删除的（关键字为空），并忽略这些已删除的记录。

如果db_nextrec在循环中被调用时数据库正被修改，则db_nextrec返回的记录只是变化中的数据库在某一时刻的快照（snapshot）。db_nextrec总是返回一条“正确”的记录，也就是说它不会返回一条已删除的记录。但有可能一条记录刚被 db_nextrec返回后就被删除。类似的，如果db_nextrec刚跳过一条已删除的记录，这条记录的空间就被一条新记录重用，除非用db_rewind并重新遍历一遍，否则看不到这条新的记录。如果通过 db_nextrec获得一份数据库的准确的“冻结”的快照很重要，则应作到在这段时间内没有添加和删除。

下面来看db_nextrec使用的加锁。并不使用任何的散列链表，也不判断每条记录属于哪条散列链。所以有可能当 db_nextrec读取一条记录时，其索引记录正在被删除。为了防止这种情况，db_nextrec对空闲链表加读锁，这样就可避免与 _db_dodelete和_db_findfree相互影响。

程序16-21　db_nextrec函数

```
#include    "db.h"

/* Return the next sequential record.
 * We just step our way through the index file, ignoring deleted
 * records.  db_rewind() must be called before this function is
 * called the first time.
 */

char *
db_nextrec(DB *db, char *key)
{
    char    c, *ptr;

        /* We read lock the free list so that we don't read
           a record in the middle of its being deleted. */
    if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("readw_lock error");

    do {
            /* read next sequential index record */
        if (_db_readidx(db, 0) < 0) {
            ptr = NULL;     /* end of index file, EOF */
            goto doreturn;
        }
            /* check if key is all blank (empty record) */
        ptr = db->idxbuf;
        while ( (c = *ptr++) != 0  &&  c == ' ')
            ;   /* skip until null byte or nonblank */
    } while (c == 0);   /* loop until a nonblank key is found */

    if (key != NULL)
        strcpy(key, db->idxbuf);    /* return key */
    ptr = _db_readdat(db);  /* return pointer to data buffer */

    db->cnt_nextrec++;
doreturn:
```

China-pub.com
下载


```
        if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
            err_dump("un_lock error");
        return(ptr);
    }
```

16.8 性能

为了测试这个数据库函数库，也为了获得一些时间上的测试数据，我们编写了一个测试程序。这个程序接受两个命令行参数：要创建的子进程的个数和每个子进程向数据库写的数据记录的条数（*nrec*）。然后创建一个空的数据库（通过调用db_open），通过fork创建指定数目的子进程，等待所有子进程结束。每个子进程执行以下步骤：

(1) 向数据库写*nrec*条记录。

(2) 通过关键字读回*nrec*条记录。

(3) 执行下面的循环*nrec*×5次。

(a) 随机读一条记录。

(b) 每循环37次，随机删除一条记录。

(c) 每循环11次，随机添加一条记录并读取这条记录。

(d) 每循环17次，随机替换一条记录为新记录。间隔地一次用同样大小的记录替换，一次用比以前更长的记录替换。

(4) 将此进程写的所有记录删除。每删除一条记录，随机地寻找10条记录。

随着函数的调用次数增加，DB结构的cnt_xxx变量记录对一个数据库进行的操作数。每个子进程的操作数一般都会与其他子进程不一样，因为每个子进程用来选择记录的随机数生成器是每个子进程根据其进程号来初始化的。当*nrec*为500时，每个子进程的较典型的操作记数见表16-1。

表16-1 *nrec*为500时，每个子进程调用的操作的典型次数

操 作	计 数
db_store, DB_INSERT无空记录，添加到尾部	675
db_store, DB_INSERT,重用空记录空间	170
db_store, DB_REPLACE,长度不同，添加到尾部	100
db_store,DB_REPLACE，长度相同	100
db_store,记录没找到	20
db_fetch,记录找到	8300
db_fetch,记录没找到	750
db_delete,记录找到	840
db_delete,记录没找到	100

读取的次数大约是存储和删除的10倍，这可能是许多数据库应用程序的一般情况。

每一个子进程进行这些操作（读取，存储和删除）时，只对此子进程写的记录进行。由于所有的子进程对同一个数据库进行操作（虽然对不同的记录），所以所有的并发控制都会被使用到。数据库中的记录总条数随子进程数按比例增加。（当只有一个子进程时，一开始有*nrec*条记录写入数据库。当有两个子进程时，一开始有*nrec*×2条记录写入数据库，依此类推。）

我们通过运行测试程序的三个不同版本来比较加粗锁和加细锁提供的并发，并且比较三种不同的加锁方式（不加锁、建议锁和强制锁）。第一个版本加细锁，用16.7节中的程序。第二

China-pub.com 下载

个版本通过改变加锁的调用而使用粗锁，16.6节对此已介绍过。第三个版本将所有加锁函数均去掉，这样可以计算加锁的开销。通过改变数据库文件的许可标志位，还可以使第一和第二个版本（加细锁和加粗锁）使用建议锁或强制锁（本节所有的测试报告中，对加细锁只测试了采用强制锁的情况）。

本章所有的测试都在一台运行SVR4的80386系统上进行。

16.8.1 单进程的结果

表16-2显示了只有一个子进程运行的结果，*nrec*分别为500，1000，2000。

表16-2 单进程，不同的*nrec*和不同的加锁方法

	不加锁			建议锁						强制锁		
				粗锁			细锁			细锁		
nrec	用户	系统	时钟	用户	系统	时钟	用户	系统	时钟	用户	系统	时钟
500	15	68	84	16	78	94	15	79	94	16	92	109
1000	61	340	402	63	360	425	63	366	430	71	412	488
2000	157	906	1068	158	936	1096	158	934	1097	159	1081	1253

最后12列显示的是以秒为单位的时间。在所有的情况下，用户CPU时间加上系统CPU时间都基本上等于总的运行时间。这一组测试受CPU限制而不是受磁盘操作限制。

中间6列（建议锁）对加粗锁和加细锁的结果基本一样。这是可以理解的，因为对于单个进程来说加粗锁和加细锁并没有区别。

比较不加锁和加建议锁，可以看到加锁的调用对系统CPU时间增加了3%到15%。即使这些锁实际上并没有使用过（只有一个进程），fcntl系统调用仍会有一些时间的开销。用户CPU时间对四种不同的加锁基本上一样，这是因为用户代码基本上是一样的（除了调用fcntl的次数有些不同外）。

关于表16-2要注意的最后一点是强制锁比建议锁增加了大约15%的系统开销。由于对加强制细锁和加建议细锁的调用次数是一样的，故添加的系统开销来自read和write。

最后的运行测试是有多个子进程的不加锁的程序。与预想的一样，结果是随机的错误。一般情况包括：加入到数据库中的记录找不到，测试程序异常退出等。几乎每次运行测试程序，就有不同的错误发生。这是典型的竞态——多个进程在没有任何加锁的情况下修改同一个文件，错误情况不可预测。

16.8.2 多进程的结果

下一组测试主要目的是比较粗锁和细锁的不同。前面说过我们认为由于加细锁时数据库的各个部分被加锁的时间比加粗锁少，所以加细锁应能够提供更好的并发。表16-3显示了对*nrec*取500，子进程数目从1到12的测试结果。

所有的用户时间、系统时间和总时间的单位均为秒。所有这些时间均是父进程与所有子进程的总和。关于这些数据有许多需要考虑。

第8列是加建议粗锁与加建议细锁的运行总时间的时间差。从中可以看到使用细锁得到了多大的并发度。在运行测试的系统上，当使用不多于7个进程时，加粗锁要快。即使是在使用多于7个进程后，使用细锁的时间减少也不大（大约1%），这使我们怀疑使用那么多代码来实现细锁是否值得。

China-pub.com
下载

表16-3 nrec=500时不同加锁方法的比较

进程	建议锁							强制锁			
	粗锁			细锁			Δ	细锁			Δ
	用户	系统	时钟	用户	系统	时钟	时钟	用户	系统	时钟	百分比
1	16	79	96	16	83	99	3	16	96	112	16
2	42	230	273	43	237	281	8	43	271	315	14
3	79	454	536	81	464	547	11	78	545	626	18
4	128	753	884	132	757	892	8	123	888	1015	17
5	185	1123	1315	196	1173	1376	61	189	1366	1560	16
6	262	1601	1870	270	1611	1888	18	264	1931	2205	20
7	351	2164	2526	354	2174	2537	11	341	2527	2877	16
8	451	2801	3264	454	2766	3230	−34	438	3298	3750	19
9	565	3513	4092	569	3483	4067	−25	548	4148	4712	19
10	684	4293	5000	688	4215	4925	−75	658	5048	5732	20
11	812	5151	5987	811	5043	5876	−111	797	6198	7020	23
12	958	6075	7058	960	5992	6980	−78	937	7298	8265	22

我们认为从粗锁到细锁总时间会减少，但也认为对系统时间来说细锁仍将比粗锁高，对任何的进程数都应是这样。这样认为的原因是对细锁调用了更多次的 fcntl。如果将表16-1中的fcntl调用的次数加起来，平均对粗锁有22 110次，细锁25 680次（表16-1中的每个操作对于粗锁要调用两次fcntl，而对于细锁前三个db_store及记录删除（记录找到）需要调用四次 fcntl）。基于此，我们认为由于增加了16%的fcntl的调用会增加细锁的系统时间。所以测试中进程超过7个后，加细锁的系统时间反而下降使人疑惑。

最后一列是从加建议细锁到加强制细锁的系统时间的百分比增量。这与在表 16-2中看到的强制锁增加约15%~20%的系统开销是一致的。

由于所有这些测试的用户代码几乎一样（对加建议细锁和强制细锁增加了一些 fcntl调用），我们认为对每一行的用户CPU时间应基本一样。但在测试中，从加建议粗锁到加建议细锁总要增加约1%~3%的用户时间，而从加建议细锁到加强制细锁的用户时间总要减少 1%~3%。关于这一点，没有什么明显的解释。

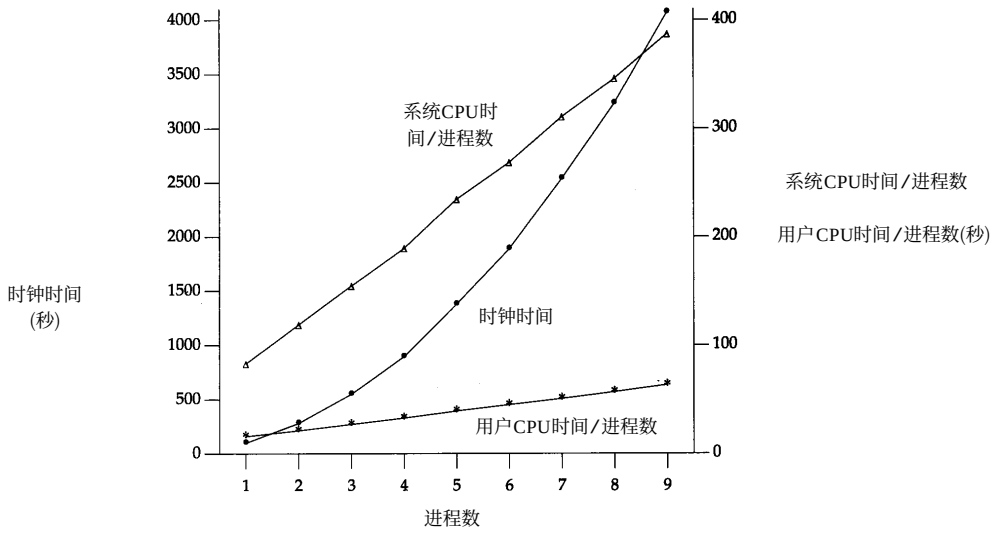

图16-4 表16-3中使用建议细锁的数据

China-pub.com 下载

表16-3的第一行与表16-2中的*nrec*取500的那一行很相似。这与预期相一致。

图16-4是表16-3中加建议细锁的数据图。我们绘制了进程数从1到9的总时间（没有绘制10，11和12，因为那要太多垂直坐标空间），也绘制了用户CPU时间除以进程数后的每进程用户CPU时间，另外还绘制了每进程系统CPU时间。

注意到这两个每进程CPU时间都是线性的，但总时间不是线性的。如果把表16-3某一行中的用户CPU时间与系统CPU时间加起来，并与总时间比较，两者的差距随进程数目增多而增大。可能的原因是当进程数增大时操作系统使用的CPU时间增多。操作系统的开销会是总时间增加，但不会影响单个进程的CPU时间。

用户CPU时间随进程数增加的原因可能是因为数据库中有了更多的记录。每一条散列链更长，所以_db_find函数平均要运行更长时间来找到一条记录。

16.9 小结

本章详细介绍了一个数据库函数库的设计与实现。出于介绍的目的，我们使这个函数库尽可能的小和简单，但也包括了多进程并发控制需要的对记录加锁的功能。

我们使用不同数目的进程，四种不同的加锁方法：不加锁，建议锁（细锁和粗锁），和强制锁，研究了这个函数库的性能。可以看到建议锁比不加锁在总时间上增加了约10%，强制锁比建议锁耗时再增加约10%。

习题

16.1 在_db_dodelete中使用的加锁是比较保守的。例如，如果等到真正要用空闲链表时再加锁，则可获得更大的并发度。如果将调用writew_lock移到调用_db_writedat和_db_readptr之间会发生什么呢？

16.2 如果db_nextrec不对空闲链表加读锁而被读的记录正在被删除，描述在怎样的情况下db_nextrec会返回一个正确的关键值但是数据记录却是空的（也就不正确了）。

16.3 在讨论了函数db_store后我们描述了_db_writeidx和_db_writedat的加锁。我们说过这种加锁不会干涉除了调用db_store外的其他的读进程和写进程。如果改为强制锁，这还成立吗？

16.4 怎样把fsync集成到这个数据库函数库中？

16.5 建立一个新的数据库并写入一些数据。写一个程序调用db_nextrec来读数据库中的记录，并调用_db_hash来计算每条记录的散列值。根据每条散列链上的记录树画出直方图。程序16-9中的函数是否足够？

16.6 修改数据库函数，使得索引文件中散列链的数目可以在数据库建立时指定。

16.7 如果你的系统支持网络文件系统，如Sun的网络文件系统（NFS）或AT&T的远程文件共享（RFS），比较两种情况的性能：（a）数据库与测试程序在同一台机器上，（b）数据库与测试程序在不同的机器上。这个数据库函数库提供的记录锁机制还能工作吗？