[打开数据库 2](#_Toc465672631)

[关闭数据库 2](#_Toc465672632)

[读写数据库记录 3](#_Toc465672633)

[存入记录 3](#_Toc465672634)

[获取记录 5](#_Toc465672635)

[删除记录 6](#_Toc465672636)

[使用C结构体 6](#_Toc465672637)

[使用指针的C结构体 7](#_Toc465672638)

[打开和关闭游标 10](#_Toc465672639)

[使用游标获取记录 11](#_Toc465672640)

[搜索记录 12](#_Toc465672641)

[操作冗余记录集 15](#_Toc465672642)

[使用游标存储数据 17](#_Toc465672643)

[使用游标删除记录 20](#_Toc465672644)

[使用游标覆盖记录 20](#_Toc465672645)

Berkeley DB是一个开源的文件数据库，介于关系数据库与内存数据库之间，使用方式与内存数据库类似，它提供的是一系列直接访问数据库的函数，而不是像关系数据库那样需要网络通讯、SQL解析等步骤。

## 打开数据库

要打开一个数据库，首先需要执行db\_create()函数来初始化一个DB句柄。一旦完成了句柄的初始化，你能用open()方法打开数据库。默认情况下，DB不会在数据库不存在的时候自动创建。如果希望DB能够自动创建一个数据库，可以为open（）函数指定DB\_CREATE标志。

## 关闭数据库

一旦使用了数据库，你必须关闭它。可以使用DB->close()方法来关闭（上面的例子已经描述了）

关闭一个数据库将导致它不能再被访问，直到你再次打开它。在关闭数据库之前，请先关闭该数据库的游标。在关闭数据库期间，如果有活动的游标存在，将会导致出现数据冲突错误。

需要注意当你关闭数据库的时候，默认的情况会把缓冲的内容写入到硬盘上。你可以手工调用DB->sync()方法。

## 读写数据库记录

在进行数据库记录读写的时候，需要注意对于重复记录（duplicate records）和非重复记录（即普通记录），在操作的时候是略有不同的。 多个数据库记录如果共享相同的Key，彼此之间存在主次关系，即一个记录有对应的一个或多个副本。共享同一个Key的一组记录被称为duplicates set（冗余记录集）（**这里的所谓冗余，实际上是指主键非唯一的情况，也就是说两条或多条记录，数据不同，而Key的值是一样的，对非冗余记录集的情况，发生的情况是新的覆盖旧的。但是如果是冗余记录集的话，就会出现不覆盖，而是存在两条记录的情况，这就是所谓duplicates set**）对于DB来说，一个冗余记录集的Key只保存一份。

默认情况下，DB数据库不支持冗余记录集。当启动了冗余记录集支持后，典型的做法是使用游标（后面介绍）访问一个冗余记录集中所有的记录。

DB提供了两个基本的机制，用来获取和存储Key/Data数据

1，使用DBT->put()和DBT->get()对所有没有启动冗余记录集机制的记录进行访问。具体使用方法后面介绍。

2，使用游标。游标提供了多种方式用来存储和获取数据库记录。有关游标的使用方法会在后面的章节中详细介绍。

### 存入记录

记录会按照你选定的存取方式有组织的存储在数据库中。在某些案例中（例如使用BTree），你也许需要记录按照你预订的顺序进行存储。（可以参考Setting Comparison Comparison Function获取更多信息）

无论如何，存储和获取数据记录的机制都不没有机会改变你已经选定的数据库访问方式，排序例程（如果有的话），和已经打开的数据库。从代码的角度看，一个简单的数据库读写操作与你选定的数据库访问方式无关。

你可以使用DB->put()方法来写数据库记录。这个方法要求你提供一个记录的key和data的结构体。你可以提供一个或多个标志用来控制DB的写行为。

在这些有效地标志中，DB\_NOOVERWRITE也许是你最感兴趣的。这个标志禁止覆盖数据库中已有的记录信息。当key已经存在时，put()方法返回DB\_KEYEXIST的错误，即使你的数据库支持冗余记录集，也会返回这个错误。例如：

#include <db.h>

#include <string.h>

char \*description = "Grocery bill.";

DBT key,data;

DB \*my\_database;

int ret;

float money;

/\* 略过数据库打开部分\*/

money=123.22;

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=&money;

key.size=sizeof(float);

data.data=description;

data.size=strlen(description)+1;

ret = my\_database->put(my\_database,NULL,&key,&data,DB\_NOOVERWRITE);

if(ret==DB\_KEYEXIST){

my\_database->err(my\_database,ret,"Put failed because key %f already exists",money);

}

### 获取记录

你可以使用DB->get()函数来获取数据库的记录。注意如果你的数据库支持冗余记录集，默认情况下这个方法只能返回冗余记录集中的第一条记录。由于这个原因，如果你的数据库启动了冗余记录集，那么你应该用游标来获取数据结果，而不是DB->get()函数。

（你也可以使用bulk get的方式获取一个冗余结果集。当需要使用的时候，需要为get()方法指定DB\_MULTIPLE标志）

默认情况下DB->get()函数返回的是找到的第一条匹配的记录。如果是在支持冗余结果集的DB中执行时，你可以通过设置DB\_GET\_BOTH标志，让get方法返回与提供的Key和Data完全匹配的记录（***默认情况下是提供一个Key，查找匹配的记录，当Key都一样的时候，返回第一条符合的记录。但是Key一样不代表Data也一样，这个DB\_GET\_BOTH就是在对比的时候，让DB不仅对比Key，而且同时对比Data***）。如果特定记录找不到，方法返回DB\_NOTFOUND。

#include <db.h>

#include <string.h>

#define DESCRIPTION\_SIZE 199

DBT key,data;

DB \*my\_database;

float money;

char description[DESCRIPTION\_SIZE+1];

/\*省略数据库打开的部分\*/

money=122.33;

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=&money;

key.size=sizeof(float);

data.data=description;

data.ulen=DESCRIPTION\_SIZE+1;

data.flags = DB\_DBT\_USERMEM;

my\_database->get(my\_database,NULL,&key,&data,0);

注意本例中，data.size字段，随后会自动被获取回来的数据的实际大小所覆盖。

### 删除记录

你可以使用DB->del()方法来删除数据库中的记录。如果数据库支持冗余记录集，那么所有与给定的Key对应的记录都将被删除。如果不希望全部删除，而是删除冗余记录集中的某一条记录的话，需要使用游标（cursor）。更为彻底的删除是使DB->truncate()方法（这个函数被翻译为截断），直接把整个数据库的所有记录清除干净。举例来讲：

#include <db.h>

#include <string.h>

DBT key;

DB \*mydb;

float money = 123.22;

memset(&key,0,sizeof(DBT));

key.data=\*money;

key.size=sizeof(float);

mydb->del(mydb,NULL,&key,0);

## 使用C结构体

使用结构体来存储多种数据类型是非常容易的事情，可以把DB的数据库看作是一个具有两个列的表，一个列用来存储Key。另一个列用来存储这个结构体。采用结构体存储的方式，可以把一个包含有n列的表变成一个Key和一个包含n-1列的结构体。

当一个C结构体里不包含任何指针型的字段时，你可以安全的使用原始数据类型来存储和获取这些数据。下面是一个例子：

#include <db.h>

#include <string.h>

typedef struct my\_struct {

int id ; char famil[MAXLINE];char surname[MAXLINE];

}MY\_STRUCT;

DBT key,data;

DB \*mydb;

MY\_STRUCT user;

char \*fname="David";

char \*sname="Rider";

/\*省略数据库打开部分\*/

user.id=1;

strncpy(user.famil,fname,strlen(fname)+1);

strncpy(user.surname,sname,strlen(sname)+1);

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=&(user.id);

key.size=sizeof(int);

data.data=&user;

data.size=sizeof(MY\_STRUCT);

mydb->put(mydb,NULL,&key,&data,DB\_NOOVERWRITE);

### 使用指针的C结构体

我们经常需要在C结构体中，使用指针来动态的分配内存。这个方式可以按照实际数据量来分配空间，避免预分配耗尽可用资源。

当使用C结构体进行存储操作的时候，必须保证结构体中所有指针指向的内存是单一连续的内存块。DB将使用一个特定的位置来标识数据快的起始位置，并用一个size来表示这个数据块的大小。但如果你的结构体中包含指针，往往每个指针指向的都是动态分配的内存，那么这些内存可以在不同的位置分配，不符合必须是连续内存的要求。这是一个问题。这个问题的最简单的解决方案是在一个单独的内存位置打包你的data，然后把数据存储到其他的位置。（这个过程有时被称为资料整合。）例如：

#include <db.h>

#include <string.h>

#include <stdlib.h>

typedef struct my\_struct {

int id ;

char \*familiar\_name; //这里改成指针了。

char \*surname;

}MY\_STRUCT;

DBT key,data;

DB \*my\_database;

MY\_STRUCT user;

int buffsize , bufflen;

char fname[]="Pete";

char sname[10];

char \*databuff;

strncpy(sname,"Oar",strlen("Oar")+1);

/\*省略数据库打开部分\*/

user.id=1;

user.familiar\_name=fname;

user.surname=sname;

/\*

有些数据存储在堆栈上，有些是闲散分布的，要把这个结构体的数据存储起来，首先我们需要进行整理，把他们打包到一个单独的内存空间

\*/

/\*获取内存缓冲区\*/

buffsize=sizeof(int)+(strlen(user.familiar\_name)+strlen(user.surname)+2);

databuff=malloc(buffsize);

memcpy(databuff,&(user.id),sizeof(int));

bufflen=sizeof(int);

memcpy(databuff+bufflen,user.familiar\_name,strlen(user.familiar\_name)+1);

bufflen+=strlen(user.familiar\_name)+1;

memcpy(databuff+bufflen,user.surname,strlen(user.surname)+1);

bufflen += strlen(user.surname)+1;

/\*现在存储它们\*/

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=&(user.id);

key.size=sizeof(int);

data.data=databuff;

data.size=bufflen;

my\_database->put(my\_database,NULL,&key,&data,DB\_NOOVERWRITE);

free(sname);

free(databuff);

要从数据库获取一个包含指针的结构体，如下：

#include <db.h>

#include <string.h>

#inclcude <stdlib.h>

typedef struct my\_struct {

int id;

char \*familiar\_name;

char \*surname;

}MY\_STRUCT;

int id;

DBT key,data;

DB \*my\_database;

MY\_STRUCT user;

char \*buffer;

/\*省略数据库打开的部分\*/

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

id=1;

key.data=&id;

key.size=sizeof(int);

my\_database->get(my\_database,NULL,&key,&data,0);

/\*

某些编译器不支持指向void的指针，因此用char\*替代之

\*/

buffer= data.data;

user.id = \*((int \*)data.data); //data.data的开头是id

user.familiar\_name=buffer+sizeof(int);

user.surname=buffer+sizeof(int)+strlen(user.familiar\_name)+1;

## 打开和关闭游标

游标是基于DBC结构的，要使用一个游标，必须使用DB->cursor()方法打开它。例如：

#include <db.h>

DB \*my\_database;

DBC \*cursorp;

/\*省略数据库打开的操作。\*/

/\*获取一个游标\*/

my\_database->cursor(my\_database,NULL,&cursorp,0);

用完了以后应该关闭游标。DBC->close()方法用来关闭一个打开的游标。当游标正在打开的时候，尤其是做数据库写入操作的游标还在打开的时候，不能关闭数据库，否则会引起不可预料的错误。要求在使用完一个游标以后，立即关闭它，释放资源（例如页面锁）

#include <db.h>

DB \*my\_database;

DBC \*cursorp;

/\*省略了数据库打开和游标的操作\*/

if(cursorp!=NULL)

cursorp->close(cursorp);

if(my\_database !=NULL)

my\_database->close(my\_database,0);

## 使用游标获取记录

使用游标对数据库记录进行遍历，只需要打开游标，然后使用DBC->get()方法。需要指定DB\_NEXT标志来进行从头到尾（first to last) 这个操作。例如：

#include <db.h>

#include <string.h>

DB \*my\_database;

DBC \*cursorp;

DBT key,data;

int ret;

/\*省略数据库打开的操作\*/

my\_database->cursor(my\_database,NULL,&cursorp,0);

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

while((ret=cursorp->get(cursorp,&key,&data,DB\_NEXT))==0){

/\*进行有用的处理部分\*/

}

if(ret != DB\_NOTFOUND){

/\*错误处理部分\*/

}

/\*必须关闭游标\*/

if(cursorp!=NULL) {

cursorp->close(cursorp);

}

if(my\_database !=NULL){

my\_database->close(my\_database,0);

}

如果是希望倒过来，从最后一条到第一条，则使用标志DB\_PREV取代DB\_NEXT标志：

#include <db.h>

#include <string.h>

DB \*my\_database;

DBC \*cursorp;

DBT key,data;

int ret;

/\*省略数据库打开的操作\*/

my\_database->cursor(my\_database,NULL,&cursorp,0);

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

while((ret=cursorp->get(cursorp,&key,&data,DB\_PREV))==0){

/\*进行有用的处理部分\*/

}

if(ret != DB\_NOTFOUND){

/\*错误处理部分\*/

}

/\*必须关闭游标\*/

if(cursorp!=NULL) {

cursorp->close(cursorp);

}

if(my\_database !=NULL){

my\_database->close(my\_database,0);

}

### 搜索记录

你可以通过游标来获取数据库记录。可以使用Key作为检索条件，也可以使用Key和Data的组合。对于支持冗余数据集的数据库，你也可以执行部分匹配。在所有的案例中，作为关键词的Key和Data参数都是放在可能作为返回值的变量中。

如果搜索失败，游标的状态不会改变，同时会返回DB\_NOTFOUND信息。要使用游标进行搜索，使用方法DBT->get()，当你使用这些标志的时候，你可以提供以下的标志：

（注：在下面的游标标志列表中，包含有SET关键词的游标标志，例如：DB\_SET，在执行检查的时候，只匹配给定的记录的key部分，进行查询搜索。包含有GET关键词的游标标志，游标会同时匹配key和data两部分。不管怎样设置关键词，游标的key和data的DBT会被从数据库中获取回来的数据值所填充。）

DB\_SET 把游标移动到符合要求的第一条记录处。

DB\_SET\_RANGE 对于不使用BTree访问方式的数据库，这个标志的功能与DB\_SET一样，对于BTree的存取方式，游标将会移动到第一条大于等于你所提供的特定Key的记录处。这个对比依赖与提供对比的函数，如果没有提供对比函数，则系统采用数据字典默认提供的方法（***该标志类似于模糊查询，只不过模糊查询对比的时候，返回的是符合要求的的一组数据，而这里是定位到第一条***）下面举例说明：假设数据库的记录使用下面的三个字符串作为Key：

Alabama 1

Alaska 2

Arizona 3

当提供搜索关键词是Alaska时，游标移动到第二条key(Alaska)上。提供Al作为key时，游标移动到第一条（Alabama）上；提供搜索key=Alas时，移动到第二条（Alaska）上。提供key=Ar将会让游标移动到最后一条key（Arizona）上。

DB\_GET\_BOTH 对比的时候，同时考虑key和data，把游标移动到第一条同时符合key和data的记录上。

DB\_GET\_BOTH\_RANGE 同时对比key和data。对比的条件是key相等（注意，是Key相等），data是大于等于指定的data。把游标移动到第一条符合该条件的记录上。如果数据库支持冗余记录集，那么在相等key的记录上会有多个data。游标会移动到符合key记录的，大于等于给定data的最小的一条记录上。举例说明，给定key/data对如下所述：

Alabama/Athens 1

Alabama/Florence 2

Alaska/Anchorage 3

Alaska/Fairbanks 4

Arizona/Avondale 5

Arizona/Florence 6

|  |  |  |
| --- | --- | --- |
| 搜索Key | 搜索的data | 最终定位位置 |
| Alaska | Fa | Alaska/Fairbanks |
| Arizona | Fl | Arizona/Florence |
| Alaska | An | Alaska/Anchorage |

举例：

#include <db.h>

#include <string.h>

DBC \*cursorp;

DBT key,data;

DB \*dbp;

int ret;

char \*search\_data = "Fa";

char \*search\_key = "Alaska";

/\*省略数据库打开部分\*/

/\*获取一个游标\*/

dbp->cursor(dbp,NULL,&cursorp,0);

key.data=search\_key;

key.size = strlen(search\_key)+1;

data.data=search\_data;

data.size=strlen(search\_data)+1;

/\*定位到符合要求的游标位置\*/

ret=cursorp->get(cursorp,&key,&data,DB\_GET\_BOTH\_RANGE);

if(!ret) {

/\* 处理data的部分\*/

}else{

/\*错误句柄处理\*/

}

/\*关闭游标\*/

if(cursorp!=NULL)

cursorp->close(cursorp);

if(dbp!=NULL)

dbp->close(dbp,0);

### 操作冗余记录集

两条使用相同key的记录被称为冗余记录。对于冗余记录来讲，只有data部分是唯一的。冗余记录只有BTree和Hash访问方法支持。如果数据库支持了冗余记录集，那么他就可能包含有共享一个key的多个记录。默认情况下，通常数据库执行get()方法会在这个冗余记录集中，获取第一条记录作为返回。如果要访问其余的记录信息，可以使用游标来操作。后面的DBC->get()的标志，主要兴趣就在于如何操作数据库中的冗余记录集

* DB\_NEXT,DB\_PREV ：显示数据库中下一条/前一条记录。这个标志并不关心当前的记录是否属于冗余结果记录集。这两个标志前面已经介绍过。
* DB\_GET\_BOTH\_RANGE 用来搜索游标中的特定记录，也是不管是否是冗余结果记录集，只负责搜索信息。
* DB\_NEXT\_NODUP, DB\_PREV\_NODUP 获取数据库中下一条/前一条非冗余记录。这个标志允许你跳过所有的冗余记录集中的记录。如果你在调用DBC->get()方法的时候，使用了DB\_PREV\_NODUP参数，这个参数可以接着上次找的结果继续往前找，比如我们有一个结果集，如下所示：

Alabama/Athens 1

Alabama/Florence 2

Alaska/Anchorage 3

Alaska/Fairbanks 4

Arizona/Avondale 5

Arizona/Florence 6

当前的游标位置是4，使用标志DB\_PREV\_NODUP调用get()方法，搜索Key是Ala，data是F开头的对象，那么就会定位到2上。相似的原理，如果使用DB\_NEXT\_NODUP调用get方法，那么就定位到6上去了。当没有符合要求的Key存在时，返回DB\_NOTFOUND标志，并且不会改变游标的内容。

* DB\_NEXT\_DUP 获取共享当前key的下一条记录。如果游标已经定位在最后一条符合要求的记录上时，再调用get方法，将会返回DB\_NOTFOUND，并且不会改变游标的内容。下面的代码演示了简明扼要的操作步骤

#include <db.h>

#include <string.h>

DB \*dbp;

DBC \*cursorp;

DBT key,data;

int ret;

char \*search\_key = "Al";

/\*省略数据库打开部分\*/

dbp->cursor(dbp,NULL,&cursorp,0);

key.data=search\_key;

key.size=strlen(search\_key)+1;

ret=cursorp->get(cursorp,&key,&data,DB\_SET);

while(ret != DB\_NOTFOUND){

printf("key:%s,data:%s\n",(char\*)key.data,(char\*)data.data);

ret = cursorp->get(cursorp,&key,&data,DB\_NEXT\_DUP);

}

if(cursorp!=NULL)

cursorp->close(cursorp);

if(dbp != NULL)

dbp->close(dbp,0);

## 使用游标存储数据

你可以使用游标把记录存储到数据库中去。Berkeley DB可以根据你设置的标志、数据库DB的访问方式以及数据库是否支持冗余数据及等，采用不同的存储数据记录的方式。当你使用游标向数据库添加记录时，你游标所处的位置正是你刚刚插入记录的位置。可以使用方法DBC->put()来向数据库写入记录。可以设置以下的标志位：

* DB\_NODUPDATA 如果数据库中已经存在相同key的记录，则不会插入，并且返回DB\_KEYEXIST。如果不存在，那么就会向数据库插入新的记录。新记录插入的顺序由数据库的插入操作顺序决定。如果数据库提供排序的函数，那么插入的记录会按照函数排序的结果执行。否则（比如BTree）由数据字典排序，短的字符串排序在长字符串之前。这个DB\_NODUPDATA标志只能对使用BTree和Hash访问方法的数据库使用。并且，只有数据库被配置为支持冗余数据成员（在数据库创建时启动了DB\_DUPSORT标志）。对于使用Queue和Recno访问方式的数据库，不能使用这个标志。更多信息本书最后会有描述。
* DB\_KEYFIRST 对于不支持冗余数据的数据库，设置这个标志跟默认情况下的操作一样。如果数据库支持冗余记录集，并且一个冗余排序函数已经被指定，那么新插入的记录将按照冗余排序函数的要求存储。但是如果key已经在数据库中存在，并且没有指定冗余排序的函数的时候，指定DB\_KEYFIRST将使新插入的数据作为该key的第一条数据（first data）存储，也就是说，使用get()函数返回这个数据集的第一条记录时，将会返回刚插入的这条记录。
* DB\_KEYLAST 与前面的DB\_KEYFIRST原理一样，只不过它会把新插入的记录作为最后一条记录。下面是一个例子：

#include <db.h>

#include <string.h>

DB \*dbp;

DBC \*cursorp;

DBT data1,data2,data3;

DBT key1,key2;

char \*key1str="My First String";

char \*data1str="My first data";

char \*key2str="A Second string";

char \*data2str = "My second data";

char \*data3str = "My third data";

int ret;

key1.data=key1str;

key1.size=strlen(key1str)+1;

data1.data = data1str;

data1.size=strlen(data1str)+1;

key2.data=key2str;

key2.size=strlen(key2str)+1;

data2.data=data2str;

data2.size=strlen(data2str)+1;

data3.data=data3str;

data3.size=strlen(data3str)+1;

/\*省略数据库打开部分\*/

/\*获取游标\*/

dbp->cursor(dbp,NULL,&cursorp,0);

/\*假设数据库是空的，先使用游标把key1/data1的记录放进去\*/

ret = cursorp->put(cursorp,&key1,&data1,DB\_KEYFIRST);

/\*然后把第二条记录放进去，放入后的排序依赖于数据库的存取方式和当前的排序方式，

就目前情况看，放入第二条记录后，新放入的记录貌似排在第一位，“A Second string”以字母A开头

而第一次插入的“My First String”是以字母M开头，A应该排在M之前。

\*/

ret=cursorp->put(cursorp,&key2,&data2,DB\_KEYFIRST);

/\*

\* 下面就到了亮点了，如果你的数据库不支持冗余记录集，

\* 那么当前记录将被新放入的记录覆盖掉data部分。

\*/

/\*

\* 但如果数据库支持冗余记录集（duplicates ），那么新插入的记录

\* 将被放入冗余记录列表中，第三个字母是t，比s大，

\* 因此应该是放在冗余记录列表的第二位。

\* 但是由于使用了DB\_KEYFIRST标志，也就是说，强制要求放在冗余列表的第一位

\*/

ret = cursorp->put(cursorp,&key2,&data3,DB\_KEYFIRST);

dbp->sync(); //同步数据到硬盘

if(cursorp!= NULL)

cursorp->close(cursorp);

if(dbp != NULL)

dbp->close(dbp,0);

## 使用游标删除记录

使用游标删除数据，仅仅需要把游标定位到需要删除的记录的位置，然后调用DBC->del()方法即可。例如：

#include <db.h>

#include <string.h>

DB \*dbp;

DBC \*cursorp;

DBT key,data;

char \*key1str = "My first string";

int ret;

/\*省略数据库打开操作\*/

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=key1str;

key.size=sizeof(key1str)+1;

dbp->cursor(dbp,NULL,&cursorp,0);

while((ret=cursorp->get(cursorp,&key,&data,DB\_SET))==0){

cursorp->del(cursorp,0);

}

if(cursorp!=NULL)

cursorp->close(cursorp);

if(dbp!=NULL)

dbp->close(dbp,0);

## 使用游标覆盖记录

你可以使用带有DB\_CURRENT标志的DBC->put()方法，向数据库覆盖记录。

#include <db.h>

#include <string.h>

DB \*dbp;

DBC \*cursorp;

DBT key,data;

char \*key1str= "My first string";

char \*replacement\_data = "replace me";

int ret;

memset(&key,0,sizeof(DBT));

memset(&data,0,sizeof(DBT));

key.data=key1str;

key.size=strlen(key1str)+1;

/\*省略数据库打开部分\*/

/\*获取游标\*/

dbp->cursor(dbp,NULL,&cursorp,0);

/\*定位鼠标\*/

ret= cursorp->get(cursorp,&key,&data,DB\_SET);

if(ret==0){

data.data=replacement\_data;

data.size=strlen(replacement\_data)+1;

cursorp->put(cursorp,&key,&data,DB\_CURRENT);

}

/\*关闭游标\*/

if(cursorp!=NULL)

cursorp->close(cursorp);

if(dbp!=NULL)

dbp->close(dbp,0);

使用游标覆盖数据时请注意，只能覆盖data，对key的记录无效，即使指定了，也会被忽略掉。

另外对于经过排序的冗余记录集，对冗余记录集中的某一条记录执行覆盖data部分的数据时，只有新记录的排序区与被覆盖掉的旧数据完全相同时才能执行成功。也就是说如果你要对一个排序过的冗余记录集中的某一条记录执行replacing（覆盖），并且使用默认提供的排序操作的话，那么覆盖操作将会在执行到排序的时候失败。然而，如果你提供一个默认的自定义排序例程（例如基于data之外的少数几个字节的排序）。那么你有可能执行一个直接的覆盖操作，并且不会触发这里描述的限制条件。

在有加密的环境下，如果你希望覆盖一个冗余记录集中的某一条数据，并且没有自己定制的默认排序例程，那么应该删除并创建一个新的记录来达到预期的效果。