Базы данных

Лекция 03 – Berkeley Db. Начало

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.02.09

Оглавление

Базы данных

В Berkeley DB база данных представляет собой набор записей.
Записи, в свою очередь, состоят из пар ключ/данные.
Концептуально базу данных можно представить как содержащую таблицу с двумя столбцами, где столбец 1 содержит ключ, а столбец 2 содержит данные.
И ключ, и данные управляются с помощью структур **DBT** (db.h).

Использование базы данных БД включает в себя:
- добавление;
- получение;
- удаление

записей базы данных.

Открытие базы данных

Чтобы открыть базу данных, сначала необходимо использовать функцию **db_create()**, которая создает и инициализирует дескриптор БД.
Для открытия базы данных после получения дескриптора нужно использовать его метод **open()**.
По умолчанию, если базы данных еще не существуют, они не создаются.
Это поведение можно переопределить, указав флаг **DB_CREATE** в методе[1] **open()**.

```
int db_create(DB **,      // указатель на место в памяти, где будет дескриптор
              DB_ENV *,   // указатель на окружение
              u_int32_t); // флаги открытия базы данных
```

1) >120 методов

Пример:

```
#include <db.h>

...

DB        *dbp;   // DB structure handle
u_int32_t  flags; // флаги открытия базы данных
int        ret;   // function return value

// Инициализация структуры базы данных.
// База данных не открывается в окружении, поэтому указатель на окружение NULL.
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
  // Обработка ошибок
}

// Флаги открытия базы данных
flags = DB_CREATE;          // Если база данных не существет, она будет создана

ret = dbp->open(dbp,        // указатель на структуру базы данных
                NULL,       // указатель на транзакцию (transaction pointer)
                "foo.db",   // фпайл на диске, в котором размещается база данных
                NULL,       // необязательное логическое имя базы данных
                DB_BTREE,   // метод доступа к базе данных
                flags,      // флаги открытия базы данных
                0);         // режим доступа к файлу (используем режим по умолчанию)
if (ret != 0) {
  // Обработка ошибок
}
```

Закрытие баз данных

Для закрытия используется метод **DB->close()**.

Прежде чем закрыть базу данных, всегда необходимо убедиться, что все обращения к базе данных завершены.

Закрытие базы данных делает ее непригодной для использования до тех пор, пока она не будет открыта снова. Перед закрытием базы данных рекомендуется закрыть все открытые *курсоры*.

Курсоры — это механизм, с помощью которого можно перебирать записи в базе данных.

С их использованием можно получать, добавлять и удалять записи базы данных.

Если база данных допускает дублирование записей, то курсоры — это самый простой способ получить доступ ко всем записям с данным ключом, кроме первой.

Активные курсоры во время закрытия базы данных могут привести к неожиданным результатам, особенно если какой-либо из этих курсоров выполняет запись в базу данных.

Когда закрывается последний открытый дескриптор базы данных, по умолчанию ее кэш сбрасывается на диск. Это означает, что любая информация, которая была в кэше изменена, при закрытии последнего дескриптора гарантированно будет записана на диск.

Эту операцию можно выполнить вручную с помощью метода **DB->sync()**, но для обычных операций завершения работы в этом нет необходимости.

Пример:

```
#include <db.h>
...
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

Флаги открытия базы данных

Флаги приведены здесь не все, а только используемые в однопоточных приложениях.
Чтобы при вызове **DB->open()** указать более одного флага , нужно использовать побитовое ИЛИ:

```
u_int32_t open_flags = DB_CREATE | DB_EXCL;
```

DB_CREATE

Если база данных в настоящее время не существует, она будет создана.
По умолчанию попытка открытия несуществующей базы данных завершается ошибкой.

DB_EXCL

Создание базы данных для использования в эксклюзивном режиме.
Если база данных уже существует, произойдет сбой.
Этот флаг имеет смысл только при использовании вместе с **DB_CREATE**, чтобы гарантированно создать новую базу данных.

DB_RDONLY

База данных открывается только для операций чтения.
Любая последующая операция записи в базу данных вызывает сбой.

DB_TRUNCATE

Файл на диске, содержащий базу данных, будет физически обрезан (очищен).
Будут удалены все базы данных, физически содержащиеся в файле.

Функции сообщения об ошибках

БД предлагает несколько полезных методов.

```
set_errcall( )
```

Определяет функцию, которая вызывается, когда БД выдает сообщение об ошибке.
Префикс ошибки и сообщение передаются этому обратному вызову.
Приложение должно самостоятельно отображать эту информацию.

```
set_errfile( )
```

Устанавливает **`FILE *`** (стандартная C-библиотека) для отображения сообщений об ошибках, выдаваемых библиотекой DB.

```
set_errpfx( )
```

Задает префикс, используемый для любых сообщений об ошибках, выдаваемых библиотекой БД.

```
err( )
```

Выдает сообщение об ошибке. Сообщение об ошибке отправляется функции обратного вызова, определенной вызовом **`set_errcall( )`**. Если данный вызов не использовался, то сообщение об ошибке отправляется в файл, определенный функцией **`set_errfile( )`**. Если ни один из этих методов не использовался, то сообщение об ошибке отправляется в **`stderr`**.

Сообщение об ошибке состоит из строки префикса, определенного вызовом **`set_errpfx( )`**, необязательного сообщения, отформатированного в стиле **`printf( )`**, сообщения об ошибке и символа новой строки.

```
errx( )
```

Ведет себя идентично **err( )**, за исключением того, что текст сообщения, связанный со значением ошибки, к строке ошибки не добавляется.

Помимо этих для информационных сообщений существуют и другие вызовы.

```
set_msgcall( )
set_msgfile( )
set_msgpfx( )
msg( )
```

Также можно использовать функцию **db_strerror( )** для прямого возврата строки ошибки, соответствующей конкретному номеру ошибки.

Чтобы отправить все сообщения об ошибках данного дескриптора базы данных в функцию обратного вызова для специфической обработки, сначала следует создать функцию обратного вызова:

```
void error_handler(const DB_ENV *dbenv,
                   const char   *error_prefix,
                   const char   *msg) {
    ... // code to handle the error prefix and error message.
}
```

Затем ее регистрируем:

```
#include <db.h>
#include <stdio.h>

DB *dbp; // указатель на структуру-дескриптор базы данных
int ret;

ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
        fprintf(stderr, "%s: %s\n", "my_program", db_strerror(ret));
        return(ret);
}

dbp->set_errcall(dbp, error_handler);     // регистрация ф-ции обратного вызова
dbp->set_errpfx(dbp, "example_program");  // префикс
```

Выдаем сообщение об ошибке открытия :

```
ret = dbp->open(dbp,       // DB structure pointer
                NULL,      // Transaction pointer
                "mydb.db", // On-disk file that holds the database
                NULL,      // Optional logical database name
                DB_BTREE,  // Database access method
                DB_CREATE, // Open flags
                0);        // File mode (using defaults)
if (ret != 0) {
    dbp->err(dbp, ret, "Database open failed: %s", "mydb.db");
    return(ret);
}
```

Управление базами данных в окружениях

Окружения часто используются для широкого класса приложений БД.

Окружение — это инкапсуляция одной или нескольких баз данных.

Окружения предлагают очень много функций, которые не может предложить автономная база данных БД:

- файлы с несколькими базами данных;
- поддержка многопоточности и многопроцессности;
- транзакционная обработка;
- поддержка высокой доступности (репликация);
- подсистема протоколирования (регистрации).

Чтобы использовать окружение, необходимо сначала создать дескриптор окружения, а затем его открыть.

При открытии необходимо определить каталог, в котором находится окружение.

Этот каталог должен существовать до открытия.

Во время открытия можно определить некоторые свойства окружения, например, можно ли создать окружение, если оно еще не существует.

Также при открытии окружения необходимо инициализировать в памяти кэш.

Пример

```
#include <db.h>
...
DB_ENV    *myEnv;          // структура-дескриптор окружения
DB        *dbp;            // структура-дескриптор базы данных
u_int32_t  db_flags;       // флаги открытия базы данных
u_int32_t  env_flags;      // флаги открытия окружения
int        ret;            // статус выполнения функции

// Создаем дескриптор объекта окружения и инициализируем его.
ret = db_env_create(&myEnv, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating env handle: %s\n", db_strerror(ret));
    return -1;
}

// Флаги окружения
env_flags = DB_CREATE |    // Если окружение не существует, создаем его
            DB_INIT_MPOOL; // Инициализация кэша в памяти.
// Открываем окружение
ret = myEnv->open(myEnv,                    // DB_ENV ptr – указатель на окружение
                  "/export1/testEnv",       // домашний каталог окружения
                  env_flags,                // флаги открытия
                  0);                       // режим доступа к файлу (по умолчанию)
if (ret != 0) {
    fprintf(stderr, "Environment open failed: %s", db_strerror(ret));
    return -1;
}
```

Когда окружение открыто, можно открывать в нем базы данных.

По умолчанию базы данных хранятся в домашнем каталоге окружения или, если указан какой-либо путь в имени файла базы данных, относительно этого каталога:

```
// Инициализируем DB структуру.
// Передаем указатель на окружение, где будет открываться эта DB.
ret = db_create(&dbp, myEnv, 0);
if (ret != 0) {
  // Обработка ошибок
}

// Database open flags
db_flags = DB_CREATE;       /* If the database does not exist, create it.*/

// open the database
ret = dbp->open(dbp,        /* DB structure pointer */
                NULL,       /* Transaction pointer */
                "my_db.db", /* On-disk file that holds the database. */
                NULL,       /* Optional logical database name */
                DB_BTREE,   /* Database access method */
                db_flags,   /* Open flags */
                0);         /* File mode (using defaults) */
if (ret != 0) {
  // Обработка ошибок
}
```

После завершения работы с окружением, его необходимо закрыть.
Перед закрытием рекомендуется закрыть все открытые базы данных.

```
if (dbp != NULL) {
    dbp->close(dbp, 0);
}

if (myEnv != NULL) {
    myEnv->close(myEnv, 0);
}
```

Пример базы данных

Создается несколько приложений, загружающих и извлекающих данные инвентаризации из баз данных BDB.

Пример 2.1 Структура stock_db

Сначала создается структура, которая будет использоваться для хранения всех указателей и имен баз данных:

```
/* File: gettingstarted_common.h */
#include <db.h>

typedef struct stock_dbs {
    DB   *inventory_dbp;     // база данных, содержащая инвентарную информацию
    DB   *vendor_dbp;        // база данных, содержащая информацию о поставщиках

    char *db_home_dir;       // каталог с файлами баз данных
    char *inventory_db_name; // имя базы данных с инвентарной информацией
    char *vendor_db_name;    // имя базы данных поставщиков
} STOCK_DBS;

// Прототипы функций приложения
int databases_setup(STOCK_DBS *, const char *, FILE *);
int databases_close(STOCK_DBS *);
void initialize_stockdbs(STOCK_DBS *);
int open_database(DB **, const char *, const char *, FILE *);
void set_db_filenames(STOCK_DBS *my_stock);
```

Пример 2.2. Вспомогательные функции stock_db

Нужны некоторые служебные функции, которые используются, чтобы убедиться, что структура **stock_db** находится в нормальном состоянии перед ее использованием.

Одна из функций представляет собой простую функцию, которая инициализирует все указатели структуры полезным значением по умолчанию.

Вторая используется для указания общего пути ко всем именам баз данных, чтобы можно было явно указать, где должны находиться все файлы базы данных.

```
/* File: gettingstarted_common.c */
#include "gettingstarted_common.h"

/* Initializes the STOCK_DBS struct.*/
void initialize_stockdbs(STOCK_DBS *my_stock) {

    my_stock->db_home_dir       = DEFAULT_HOMEDIR;
    my_stock->inventory_dbp     = NULL;
    my_stock->vendor_dbp        = NULL;

    my_stock->inventory_db_name = NULL;
    my_stock->vendor_db_name    = NULL;
}
```

```
/* Identify all the files that will hold our databases. */
void set_db_filenames(STOCK_DBS *my_stock) {

    size_t size;

    /* Create the Inventory DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(INVENTORYDB) + 1;
    my_stock->inventory_db_name = malloc(size);
    snprintf(my_stock->inventory_db_name,
             size, "%s%s",
             my_stock->db_home_dir,
             INVENTORYDB);

    /* Create the Vendor DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(VENDORDB) + 1;
    my_stock->vendor_db_name = malloc(size);
    snprintf(my_stock->vendor_db_name,
             size,
             "%s%s",
             my_stock->db_home_dir,
             VENDORDB);
}
```

Пример 2.3 Функция open_database()

Открывается несколько баз данных с одинаковыми флагами и настройками отчетов об ошибках. Для этого создается функция, которая выполняет эту операцию:

```
/* Opens a database */
int open_database(DB **dbpp,                // The DB handle that we are opening
                  const char *file_name,    // The file in which the db lives
                  const char *program_name, // Name of the program calling this func
                  FILE *error_file_pointer) // File where we want error messages sent {
    DB *dbp;    /* For convenience */
    u_int32_t open_flags;
    int       ret;

    /* Initialize the DB handle */
    ret = db_create(&dbp, NULL, 0);
    if (ret != 0) {
        fprintf(error_file_pointer, "%s: %s\n", program_name,
                db_strerror(ret));
        return(ret);
    }

    /* Point to the memory malloc'd by db_create() */
    *dbpp = dbp;

    /* Set up error handling for this database */
    dbp->set_errfile(dbp, error_file_pointer);
    dbp->set_errpfx(dbp, program_name);
```

```
    /* Set the open flags */
    open_flags = DB_CREATE;

    /* Now open the database */
    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    file_name,  /* File name */
                    NULL,       /* Logical db name (unneeded) */
                    DB_BTREE,   /* Database type (using btree) */
                    open_flags, /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database '%s' open failed.", file_name);
        return(ret);
    }

    return (0);
}
```

Example 2.4 The databases_setup() Function

С использованием **open_database( )** создается функция, которая откроет все базы данных.

```
/* opens all databases */
int databases_setup(STOCK_DBS *my_stock,
                    const char *program_name,
                    FILE *error_file_pointer) {
    int ret;

    /* Open the vendor database */
    ret = open_database(&(my_stock->vendor_dbp),
                        my_stock->vendor_db_name,
                        program_name, error_file_pointer);
    if (ret != 0) // обработка ошибок в open_database( ) => просто возврат кода ошибки
        return (ret);

    /* Open the inventory database */
    ret = open_database(&(my_stock->inventory_dbp),
                        my_stock->inventory_db_name,
                        program_name, error_file_pointer);
    if (ret != 0)
        return (ret);

    printf("databases opened successfully\n");
    return (0);
}
```

Функция databases_close()

Полезно иметь функцию, которая может закрыть для все базы данных:

```
/* Closes all the databases. */
int databases_close(STOCK_DBS *my_stock) {
    int ret;
     // Closing a database automatically flushes its cached data
     // to disk, so no sync is required here.

    if (my_stock->inventory_dbp != NULL) {
        ret = my_stock->inventory_dbp->close(my_stock->inventory_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Inventory DB close failed: %s\n", db_strerror(ret));
    }

    if (my_stock->vendor_dbp != NULL) {
        ret = my_stock->vendor_dbp->close(my_stock->vendor_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Vendor DB close failed: %s\n", db_strerror(ret));
    }

    printf("databases closed.\n");
    return (0);
}
```