LibCurl基本使用

# 1.概述

#include <curl/curl.h>

curl\_global\_init() ：初始化libcurl；

curl\_global\_cleanup()：释放资源占用；

curl\_easy\_init() ：得到easy interface指针；

curl\_easy\_setopt() ：设置传输选项；

curl\_easy\_perform() ：完成传输任务；

curl\_easy\_cleanup() ：释放内存；

# 2.各函数描述

## 2.1.curl\_global\_init函数

函数原型：CURLcode curl\_global\_init(long flags);

本函数只能使用一次，如需再次使用，需要使用curl\_easy\_cleanup函数后再次使用；

如果在调用curl\_easy\_init前还没有调用该函数，libcurl库将自动执行一遍该调用，但需要注意，该函数不是线程安全的！

所以多线程中，应当主动在主线程中调用该函数执行初始化，防止出现多线程不安全的情况。

Flags取值：

* CURL\_GLOBAL\_ALL：初始化所有可能的调用；
* CURL\_GLOBAL\_SSL：初始化支持安全套接字层；
* CURL\_GLOBAL\_WIN32：初始化win32套接字库；
* CURL\_GLOBAL\_NOTHING：没有额外的初始化；

## 2.2.curl\_global\_cleanup函数

函数原型：void curl\_global\_cleanup(void);

结束libcurl的使用时，调用该函数执行curl\_global\_init操作的工作清理。

## 2.3.curl\_easy\_init()函数

函数原型：CURL \*curl\_easy\_init(void);

该函数用来初始化一个CURL的指针(类似于FILE指针)，一般而言，该函数意味着一个会话的开始，返回一个easy\_handler，用在其他的easy系列的函数中。

调用结束后，使用curl\_easy\_cleanup函数可以反初始化该函数。

## 2.4.curl\_easy\_cleanup()函数

函数原型：void curl\_easy\_cleanup(CURL \*handle);

配合curl\_easy\_init函数使用，清空其资源占用；

## 2.5.curl\_easy\_setopt()函数

函数原型：CURLcode curl\_easy\_setopt(CURL \*handle, CURLoption option, parameter);

此函数告知curl库接下来要执行何种操作，比如要查看一个网页的html源代码等。

## 2.6.curl\_easy\_perform()函数

函数原型：CURLcode curl\_easy\_perform(CURL \*handle);

在执行完curl\_easy\_setopt函数后调用该函数，可以执行设置的需求。

## 2.7.curl\_easy\_getinfo()函数

函数原型：CURLcode curl\_easy\_getinfo(CURL \*handle, CURLINFO info, …);

该函数可以获取到curl会话中的附加信息。需要注意，只有在curl函数返回CURLE\_OK的时候，该函数才会有意义。该函数一般会在perform函数(例如curl\_easy\_perform)执行完成后被调用。

第二个参数的含义及其后紧跟的第三个参数的限制，如下所示：

### 2.7.1.CURLINFO\_SIZE\_DOWNLOAD

使用该选项作为第二个参数时，需要第三个参数时一个double类型的指针，这样会在一次传输结束后，将本次传输所下载的字节数赋值到指针所指向的double变量中。

这个字节数，只能反映最近一次的下载。

### 2.7.2.CURLINFO\_SPEED\_DOWNLOAD

使用该选项作为第二个参数时，传入double型指针作为第三个参数。这里的速度，指的是下载的平均速度，单位是**字节每秒**。

### 2.7.3.CURLINFO\_TOTAL\_TIME

使用该选项作为第二个参数时，需要传入double型指针作为第三个参数。这个参数记录了传输的总耗时(单位是**秒**)。

这里的总耗时，包括了域名解析、TCP连接建立等所有时间。

### 2.7.4.CURLINFO\_CONTENT\_TYPE

获得从http服务器中收到的header中的content-type信息。传入的第三个参数，是一个char \*\*类型的指针，例如：

char \*contype;

curl\_easy\_getinfo(pCurl, CURLINFO\_CONTENT\_TYPE, &contype);

### 2.7.5.CURLINFO\_RESPONSE\_CODE

获得http服务器返回的返回码，第三个参数应为long类型的指针。

一般而言，http服务器返回码是200的情况下，代表了操作正常。

# 3.curl\_easy\_setopt函数选项

## 3.1.CURLOPT\_URL

设置要访问的URL。

## 3.2.CURLOPT\_WRITEFUNCTION,CURLOPT\_WRITEDATA

回调函数原型：size\_t callback\_function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

回调函数在libcurl接收到数据后将被调用，因此函数大多用作数据保存的功能，例如处理下载文件。

CURLOPT\_WRITEDATA用于表明callback\_function中stream指针的来源。

如果没有通过CURLOPT\_WRITEFUNCTION设置回调函数，libcurl将提供一个默认的回调函数，这个回调函数简单的将数据打印在标准输出中。

如果通过CURLOPT\_WRITEDATA属性给默认回调函数传递一个已经打开的文件指针，数据将输出到该文件中。

## 3.3.CURLOPT\_HEADERFUNCTION,CURLOPT\_HEADERDATA

回调函数原型：size\_t function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

Libcurl一旦接收到http包头后将调用该回调函数。

CURLOPT\_HEADERDATA类似CURLOPT\_WRITEDATA。

## 3.4.CURLOPT\_READFUNCTION,CURLOPT\_READDATA

回调函数原型是：size\_t function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

Libcurl需要读取数据传递给远程主机时调用该函数。

CURLOPT\_READERDATA类似CURLOPT\_WRITEDATA。

## 3.5.CURLOPT\_NOPROGRESS,CURLOPT\_PROGRESSFUNCTION,CURLOPT\_PROGRESSDATA

与数据传输进度相关的参数。

CURLOPT\_PROGRESSFUNCTION指定的回调函数，正常情况下每秒被libcurl调用一次，为了使其被调用，CURLOPT\_NOPROGRESS必须设置成false；

CURLOPT\_PROGRESSDATA指定的参数将作为CURLOPT\_PROGRESSFUNCTION函数的第一个参数。

typedef int (\*curl\_progress\_callback)(void \*clientp,

double dltotal,

double dlnow,

double ultotal,

double ulnow);

## 3.6.CURLOPT\_TIMEOUT,CURLOPT\_CONNECTIONTIMEOUT

CURLOPT\_TIMEOUT设置文件传输的超时时间；

CURLOPT\_CONNECTIONTIMEOUT设置连接的等待时间；

## 3.7.CURLOPT\_FOLLOWLOCATION

设置重定位url。

## 3.8.CURLOPT\_RANGE,CURLOPT\_RESUME\_FROM

断点续传相关功能。

CURLOPT\_RANGE指定char \*参数传递给libcurl，用于指明http域的RANGE头域，例如：

* bytes=0-499：头500个字节；
* bytes=500-999：第二个500字节；
* bytes=-500：最后500个字节；
* bytes=500-：500字节以后的范围；
* bytes=0-0,-1：第一个和最后一个字节；
* bytes=500-600,601-999：同时指定多个范围；

CURLOPT\_RESUME\_FROM传递一个long参数给libcurl，指定希望开始传递的偏移量。

## 3.9.CURLOPT\_INTERFACE

指定网卡名字，或网卡ip值。

## 3.10.CURLOPT\_NOBODY

设置是否同步传输http的body，将该值设置为非零值，将不传输body，这个可以用作HEAD方式的传输实现。

# 4.curl\_easy\_perform函数说明

## 4.1.返回值

* CURLE\_OK：curl任务正确完成；
* CURLE\_UNSUPPORTED\_PROTOCOL：不支持的协议，由URL的头部指定；
* CURLE\_COULDNT\_CONNECT：不能连接到remote主机或者代理；
* CURLE\_REMOTE\_ACCESS\_DENED：访问被拒绝；
* CURLE\_HTTP\_RETURNED\_ERROR：http返回错误；
* CURLE\_READ\_ERROR：读取本地文件错误；

需要更加详细的错误描述字符串，可以通过const char \*curl\_easy\_strerror(CURLcode errornum)函数获取；

# 5.HTTP消息头

当使用libcurl发送http请求时，它会自动添加一些http包头，可以通过CURLOPT\_HTTPHEADER属性手动替换、添加或删除一些http包头。

* Host：http1.1版本都要求客户端请求提供这个信息头；
* Pragma=“no-cache”：表示无需缓冲数据；
* Accept=“\*/\*”：允许接收任何类型的数据；
* Expect：以POST的方式向http服务器提交请求时，libcurl会设置该消息头为“100-continue”，它要求服务器在正式处理该请求之前，返回一个“OK”消息。如果POST的数据很小，libcurl可能不会设置该消息头。

## 5.1.自定义选项

越来越多的协议都构建在http协议之上（如soap），libcurl对这些做了支持。

http支持get，head，post提交请求，可以设置CURLOPT\_CUSTOMREQUEST来设置自定义的请求方式(libcurl默认以get方式提交)：

curl\_easy\_setopt(easy\_handler, CURLOPT\_CUSTOMREQUEST, “MyOwnRequest”);

## 5.2.修改消息头

http协议提供消息头：请求消息头用于告诉服务器如何处理请求，响应消息头告诉浏览器如何处理接收到的数据。

struct curl\_slist \*headers = NULL;

headers = curl\_slist\_append(headers, “Server : who are you?”);

headers = curl\_slist\_append(headers, “X-silly-content : yes”);

curl\_easy\_setopt(easyhandler, CURLOPT\_HTTPHEADER, headers);

curl\_easy\_perform(easyhander);

curl\_slist\_free\_all(headers);

## 5.3.重设消息头

对于已经存在的消息头，可以重设其值：

headers = curl\_slist\_append(header, “Accept: Agent-007”);

headers = curl\_slist\_append(header, “Host: munged.host.line”);

## 5.4.删除消息头

对于已经存在的消息头，设置它的内容为空就可以删除该消息头：

headers = curl\_slist\_append(header, “Host:”);

# 6.获取http应答头信息

发送http请求后，服务器会返回应答头信息和数据，如果只是打印应答头的信息，可以通过：curl\_easy\_setopt(curl, CURLOPT\_HEADERFUNCTION,callback\_function);的方式来完成。

如果需要获取应答头中的信息如应答码、cookies列表等，需要如下函数：CURLcode curl\_easy\_getinfo(CURL \*curl, CURLINFO info, …);实现，其中info就是要获取的内容，罗列几个参数值：

* CURLINFO\_RESPONSE\_CODE：返回的应答码；
* CURLINFO\_HEADER\_SIZE：返回的头大小；
* CURLINFO\_COOKIELIST：cookie列表；

除了获取应答信息外，这个函数还可以curl的一些内部信息，如请求时间、连接时间等。

# 7.Curl无法正常工作

传输失败总是有可能的。

可以将CURLOPT\_VERBOSE属性设置为1，libcurl会自动输出通信过程中的一些细节，如果使用的是http协议，请求头/响应头也会被输出；将CURLOPT\_HEADER设为1，这些头信息将出现在消息的内容中。

# 8.密码

客户端向服务器发送请求时，许多协议都需要提供用户名和密码，libcurl提供了多种方式来设置：

## 8.1.CURLOPT\_USERPWD

参数形式：user:password；

函数：curl\_easy\_setopt(easy\_handler, CURLOPT\_USERPWD, “username:password”);

## 8.2.CURLOPT\_PROXYUSERPWD

访问代理服务器时，可能时时需要提供用户名和密码进行用户身份验证，这种情况下libcurl提供了另一个属性：CURLOPT\_PROXYUSERPWD实现：

curl\_easy\_setopt(easy\_handler, CURLOPT\_PROXYUSERPWD, “username:password”);

# 9.返回值

## 9.1.CURLE\_OK

Value = 0；

Desc：正确返回

## 9.2.CURLE\_OPERATION\_TIMEDOUT

Value = 28;

desc：超时

## 9.3.CURLE\_PARTIAL\_FILE

Value = 18;

desc：文件传输短于或大于预期部分文件，只有部分文件被传输；

## 9.4.CURLE\_TFTP\_NOTFOUND

Value = 68;

desc：tftp server上不存在需要下载的文件；

## 9.5.CURLE\_COULDNT\_CONNECT

Value = 7；

Desc ：http服务器无法连接

sqlite3

# 1.概述

## 1.1.linux安装

sudo apt-get install sqlite3

sudo apt-get install libsqlite3-dev

## 1.2.基本使用

### 1.2.1.创建数据库

sqlite3 test.db

### 1.2.2.创建表

sqlite> create table testTable(id integer primary key, value text);

该表包含两个字段：id，value。其中id是主键字段。

需要注意，在1.2.1中创建了数据库之后，最少要为该数据库创建一个表或视图，这样数据库才会保存到磁盘，否则数据库不会被创建。

如果需要判断表是否存在，可以使用：

sqlite> create table if not exists testTable(id integer primary key, value text);

那么只有在数据库不存在的情况下，才会创建表，否则不予创建；

### 1.2.3.写入数据

sqlite> insert into testTable(id, value) values(1, “Michale”);

sqlite> insert into testTable(value) values( “Pony”);

### 1.2.4.查询数据

sqlite> select \* from test;

可以设置格式化查询结果：

sqlite> .mode column;

sqlite> .header on;

sqlite> select \* from test;

其结果可能如下所示：

id value

--------- -------------

1. Michale
2. Pony

其中，.mode column设置为列显示模式，header将显示列名。

### 1.2.5.修改表结构

#### 增加列

sqlite> alter table testTable add column email text not null ‘’ collate nocase;;

#### 创建视图

sqlite> create view nameview as select \* from testTable;

#### 创建索引

create index test\_idx on testTable(value);

### 1.2.6.一些有用的sqlite命令

#### 显示表结构

sqlite> .schema [table]

#### 获取所有表和视图

sqlite> .tables

#### 获取指定表的索引列表

sqlite> .indices [table]

#### 导出数据库到sql文件

sqlite> .output [filename]

sqlite> .dump

sqlite> .output stdout

#### 从sql文件导入数据库

sqlite> .read [filename]

#### 格式化输出到csv格式

sqlite> .output [filename.csv]

sqlite> .separator ,

sqlite> select \* from test;

sqlite> .output stdout

#### 从csv文件导入数据库

sqlite> create table newtable(id integer primary key, value text);

sqlite> .import [filename.csv] newtable

#### 备份数据库

sqlite3 mytable.db .dump > backup.sql

#### 恢复数据库

sqlite3 mytable.db < backup.sql

# 2.C/C++使用sqlite

使用sqlite，需包含头文件：#include "sqlite3.h"

## 2.1.sqlite3\_open

函数原型：sqlite3\_open(const char \*filename, sqlite3 \*\*ppDb);

该函数打开一个指向sqlite数据库文件的链接，返回一个用于其他sqlite程序的数据库连接对象。

如果filename是NULL或“.memory”，那么该函数将会在RAM中创建一个内存数据库，这只会在session的有效时间内持续。

如果filename不是NULL，那么该函数将使用这个参数值尝试打开数据库文件。如果该文件并不存在，函数将创建一个新的命名为该名称的数据库文件并打开。

## 2.2.sqlite3\_exec

函数原型：sqlite3\_exec(sqlite3 \*pDb, const char \*sql, sqlite\_callback , void \*data, char \*\*errmsg);

该函数执行一条sqlite命令，sql命令由参数提供，可以将多个命令一起执行。

参数说明：

* 第一个参数是打开的sqlite3数据库对象；
* sqlite\_callback是一个回调函数，data作为其第一个参数；
* errmsg可以获取程序执行的错误信息；

该函数解析并执行sql参数所给的每条命令，直到字符串结束或执行出现错误为止。

### 2.2.1.回调函数

int sqlite3\_callback(

void \*data, /\*Data provided in the 4th argument of sqlite3\_exec()\*/

int n, /\*The number of columns in row\*/

char \*\*cont, /\*An array of strings representing fields in the row\*/

char \*\*colName);

如下所示：

static int selectCallback(void \*data, int nColumn, char \*\*columnValue, char \*\*columnName){

printf("data = [%s]\n", (**const** **char** \*)data);

**int** i = 0;

**for**(i = 0; i < argc; i++)

{

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : NULL);

}

printf("\n");

**return** 0;

}

针对该函数的参数，进行如下说明：

1. data：在sqlite3\_exec中传入的void\*参数，通过该参数，可以传入一些特殊的指针(类、结构体)，然后在这里强制类型转换到对应的类型，再对该数据进行操作；
2. nColumn：代表了这一条记录有多少个字段(也就是有多少列)；
3. columnValue：保存了所有查询得到的数据，实际上，这是个一维数组，而不是二维数组，每个元素都是一个char\*的类型，相当于一个字符串，代表了一个value；
4. columnName：与columnValue相对应，表示这个字段的字段名称；

针对函数的返回值，作如下说明：

1. 返回0，sqlite3\_exec()函数将继续执行查询操作；
2. 返回非零值，sqlite3\_exec()函数将立即返回，并且返回SQLITE\_ABORT；

这里的执行逻辑是，假设需要有N条记录被select到，那么每一条记录被读取到后，都会先调用该回调函数，如果调用返回0，那么将继续执行查询操作，否则停止查询动作。

通常来说，只有在查询操作时，回调函数才是有意义的，insert等操作没有必要使用回调。

除了可以通过回调函数得到当前select的结果外，还可以直接调用函数sqlite3\_get\_table来得到当前表的内容，示例如下：

char \*errmsg = NULL;

**char** \*\*pDbTableResult = NULL;

**int** nRow = 0, nColumn = 0;

int ret=sqlite3\_get\_table(pDb, strSQL, &pDbTableResult, &nRow, &nColumn, &errmsg );

**if**(SQLITE\_OK != ret)

{

printf("sqlite3\_get\_table with strSQL [%s] failed, ret = %d, Msg is [%s], errmsg returned is [%s]\n", strSQL, ret, sqlite3\_errmsg(pDb), errmsg);

sqlite3\_free\_table( pDbTableResult );

sqlite3\_free(errmsg);

sqlite3\_close(pDb);

**return** ;

}

//Show this infomation

**int** i = 0, j = 0;

**for**(i = 0; i <= nRow; i++)

{

**for**(j = 0; j < nColumn; j++)

{

printf("%s\t", pDbTableResult[i \* nColumn + j]);

}

printf("\n");

}

sqlite3\_free\_table( pDbTableResult );

sqlite3\_free(errmsg);

sqlite3\_close(pDb);

## 2.3.sqlite3\_get\_table

可以通过回调函数获取表信息，也可以通过函数sqlite3\_get\_table函数查询：

int sqlite3\_get\_table(sqlite3 \*db, const char \*sqlCmd, char \*\*queryResult, int \*nRow, int \*nColumn, char \*\*errmsg);

第三个参数是一个一位数组，其内存布局是：字段名称，后面紧跟每个字段的值；

第四个参数是查询出多少行记录；注意，这里不包含字段名那一行！

第五个参数是查询出多少列(也就是多少字段)；

所以，queryResult时机上包含的字符串数量，应该是：(\*nRow + 1) \* (\*nColumn)，因为第一行的字段名没有计入到nRow里面去。

此外，执行结束后，需要调用：sqlite3\_free\_table(char \*\*queryResult);来释放资源。

## 2.4.sqlite3\_close

函数原型：sqlite3\_close(sqlite3 \*pDb);

该函数关闭之前调用sqlite3\_open函数打开的数据库对象。所有与连接相关的语句都应在关闭之前完成。

如果有正在执行的查询操作，sqlite3\_close()将返回SQLITE\_BUSY禁止关闭的错误消息。

## 2.5.错误码

执行sqlite函数的返回值有其特定含义，于sqlite.h中定义，如下：

|  |  |  |
| --- | --- | --- |
| 错误码 | 值 | 含义 |
| SQLITE\_OK | 0 | Successful result |
| SQLITE\_ERROR | 1 | SQL error or missing database  如：执行的sql语句有错误，重复创建表等； |
| SQLITE\_INTERNAL | 2 | Internal logic error in SQLite |
| SQLITE\_PERM | 3 | Access permission denied |
| SQLITE\_ABORT | 4 | Callback routine requested an abort  回调函数返回结果不为零； |
| SQLITE\_BUSY | 5 | The database file is locked |
| SQLITE\_LOCKED | 6 | A table in the database is locked |
| SQLITE\_NOMEM | 7 | A malloc() failed |
| SQLITE\_READONLY | 8 | Attempt to write a readonly database |
| SQLITE\_INTERRUPT | 9 | Operation terminated by sqlite3\_interrupt() |
| SQLITE\_IOERR | 10 | Some kind of disk I/O error occurred |
| SQLITE\_CORRUPT | 11 | The database disk image is malformed |
| SQLITE\_NOTFOUND | 12 | Unknown opcode in sqlite3\_file\_control() |
| SQLITE\_FULL | 13 | Insertion failed because database is full |
| SQLITE\_CANTOPEN | 14 | Unable to open the database file  例如：指定的数据库文件处于只读目录下； |
| SQLITE\_PROTOCOL | 15 | Database lock protocol error |
| SQLITE\_EMPTY | 16 | Database is empty |
| SQLITE\_SCHEMA | 17 | The database schema changed |
| SQLITE\_TOOBIG | 18 | String or BLOB exceeds size limit |
| SQLITE\_CONSTRAINT | 19 | Abort due to constraint violation  例如：指定了一个属性是primary key，重复添加相同的值给该属性； |
| SQLITE\_MISMATCH | 20 | Data type mismatch |
| SQLITE\_MISUSE | 21 | Library used incorrectly |
| SQLITE\_NOLFS | 22 | Uses OS features not supported on host |
| SQLITE\_AUTH | 23 | Authorization denied |
| SQLITE\_FORMAT | 24 | Auxiliary database format error |
| SQLITE\_RANGE | 25 | 2nd parameter to sqlite3\_bind out of range |
| SQLITE\_NOTADB | 26 | File opened that is not a database file |
| SQLITE\_NOTICE | 27 | Notifications from sqlite3\_log() |
| SQLITE\_WARNING | 28 | Warnings from sqlite3\_log() |
| SQLITE\_ROW | 100 | sqlite3\_step() has another row ready |
| SQLITE\_DONE | 101 | sqlite3\_step() has finished executing |
|  |  |  |

发现sqlite相关函数执行出错时，可以通过sqlite3\_errmsg(sqlite3 \*pDb)函数将当前的错误原因打印出来；

# 3.并发

## 3.1.多线程

sqlite支持三种线程模式：

* **单线程：**这种模式下，没有执行任何互斥，多线程使用将不安全。该模式下，sqlite会禁用所有的mutex锁，一旦并发使用就会出错。
* **多线程：**这种模式下， 只要一个数据库连接不被多个线程同时使用就是安全的。在sqlite源码中，是启用bCoreMutex，禁用bFullMutex。实际上就是禁用数据库连接和prepared statement上的锁，因此不能在多个线程中并发使用同一个数据库连接和prepared statement。
* **串行：**这种模式下，sqlite是线程安全的。源码中将启用所有的锁，包括bCoreMutex和bFullMutex，因为数据库连接和prepared statement都已经加锁，所以对这些的操作也都变成串行了。

这三种模式的设置方式：

|  |  |  |  |
| --- | --- | --- | --- |
| **模式** | **编译时设置**  **SQLITE\_THREADSAFE** | **初始化前设置**  **Sqlite3\_config()函数** | **备注** |
| 单线程 | 0 | SQLITE\_CONFIG\_SINGLETHREAD | Sqlite3\_config函数需要在初始化前进行设置，这里所说的初始化，指的是调用sqlite3\_initialize函数，而这个函数在sqlite3\_open时自动调用 |
| 多线程 | 2 | SQLITE\_CONFIG\_MULTITHREAD |
| 串行 | 1 | SQLITE\_CONFIG\_SERIALIZED |
|  |  |  |

因此，如果需要达到线程安全，需要将sqlite3设置为串行模式，一般而言，默认的编译产物就是串行模式。

在串行模式下，互斥可以达到**写写互斥、读写互斥；读读可以同时进行**。一旦资源正在被占用，那么执行sqlite3\_exec函数将返回错误值SQLITE\_BUSY，以此表示资源无法被使用，具体而言描述如下：

* 有写操作时，其他读操作被驳回；
* 有写操作时，其他写操作被驳回；
* 有读操作时，其他写操作被驳回；
* 有读操作时，其他读操作可以并发执行；
* 开启事务时，提交事务之前，其他写操作被驳回；
* 开启事务时，提交事务前，其他事务请求被驳回；

## 3.2.多进程

针对多进程，sqlite同样提供了互斥操作，其互斥级别同样是写写互斥、读写互斥，读读OK。但由于sqlite自身并不支持存储过程(procedure)，因此各个进程之间同时对同一个数据表进行写入时，极有可能造成数据写入混乱的情况，需针对性处理该情况。

针对这种情况，学术上可以通过其延时执行的功能来解决。通过微改sqlite内部的延时执行功能，数据库操作一个临时结果，等待一个周期检查对应的执行结果和预期结果是否一致。如果一直就把临时结果赋值为正式结果，否则就认为操作执行失败。最后设计出来的程序可能并不能保证一定是按照预期的顺序执行的，但最少可以保证结果是正确的。

## 3.3.超时处理

在多线程、多进程中，一旦资源正在被占用，另外的申请者再次请求资源可能会出现资源被锁住无法使用的情况，这时候的返回值将是SQLITE\_BUSY。本部分所述即为针对这种情况的解决方案，通用的解决方案有两个：

1. 使用进程或线程间的同步机制以避免该情况的出现：如使用信号量(semphore)、互斥锁(mutex)等，本部分属于在应用层自行处理资源占用逻辑，避免调用sqlite时出现SQLITE\_BUSY的情况；
2. 使用sqlite自己提供的两个busy handler函数，但需要注意的是，这两个函数会互相影响，设置一个的同时会清除另外一个！

这里只对第二种方法进行描述。

### 3.3.1.sqlite3\_busy\_handler

函数原型：int sqlite3\_busy\_handler(sqlite3 \*pDb, int(\*)(void \*, int), void \*)

不注册此函数时默认的回调函数是NULL，申请不到资源将直接返回；

函数可以自己定义一个回调函数，当出现资源申请失败的时候，执行该回调函数进行延时并返回非零重试本次操作，回调函数的第二个参数代表了因为本次BUSY而需要回调该函数的最大次数，如果达到了最大次数(可以由此得到最大超时时间)还没有得到资源，才真正返回SQLITE\_BUSY；

最后传入的参数将作为回调函数第一个参数被使用；

回调函数返回非零，数据库会重试当前的操作；返回0则当前操作返回SQLITE\_BUSY；

### 3.3.2.sqlite3\_busy\_timeout

函数原型：int sqlite3\_busy\_timeout(sqlite3 \*pDb, int timeout);

不调用该函数时，默认超时等待时间为0，申请资源失败将直接返回SQLITE\_BUSY。

注册该函数，定义一个timeout(**单位是毫秒**)，那么在timeout后会重试一次资源申请，此次申请如果仍旧失败，才会返回SQLITE\_BUSY。

但是针对该函数的使用需要明确如下：因为sqlite的源码中注册了一个默认的sqlite3\_busy\_handler( sqliteDefaultBusyCallback )，而这个回调函数在编译时可能使得传入的timeout必须要大于1000并且是其整数倍才有意义，因此该timeout的设置可能会超时时间较大，且依赖性较强。

因此使用3.3.1中所述回调函数更为优越。

# 4.事务

事务(transaction)是一个对数据库的执行工作单元。它是以逻辑顺序完成的工作单元或序列，可以由用户手动执行，也可以由数据库程序自动完成。

事务具有以下四个标准属性，简称为ACID：

* 原子性(Atomicity)：确保工作单位内的所有操作都能够完成，如果未能完成，事务会在出现故障时终止，之前的操作也会回滚到执行该事务之前的操作；
* 一致性(Consistency)：确保数据库在成功提交的事务上正确的改变状态；
* 隔离性(Isolation)：事务操作相互之间独立、透明；
* 持久性(Durability)：确保已经提交事务的结果或效果在系统发生故障的情况下依然存在；

## 4.1.命令行控制事务

使用下面的命令控制事务：

* BEGIN TRANSACTION：开始事务处理；
* COMMIT：保存更改，或者可以使用END TRANSACTIOIN命令；
* ROLLBACK：回滚所作的修改；

事务控制命令只能与DML命令(INSERT UPDATE DELETE)一起使用，不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

### 4.1.1.BEGIN TRANSACTION

事务可以使用“BEGIN TRANSACTION”或者“BEGIN”来启动。事务会持续执行下去，直到遇到下一个COMMIT或ROLLBACK命令。不过在数据库关闭或者发生错误时，事务也会回滚。

### 4.1.2.COMMIT

COMMIT命令用于把事务调用的更改保存到数据库中，它会把从上次COMMIT或ROLLBACK命令以来的所有事务保存到数据库。

COMMIT命令与“END TRANSACTION”作用完全相同。

### 4.1.3.ROLLBACK

ROLLBACK命令用于撤销尚未保存到数据库中的事务，只能用于撤销 自上次发出COMMIT或ROLLBACK命令 以后的事务。

## 4.2.C/C++接口控制事务

通常来说一次sqlite3\_exec就是一个事务，如果需要删除一万条数据，sqlite就做了一万次如下操作：开始新事务🡪删除一条数据🡪提交事务🡪开始事务……。这个操作对系统资源的占用是巨大的，因为时间都花在了开始事务、提交事务上。

这种情况下，可以将这些操作做成一个事务，这样一来可以减少开始事务、提交事务的时间消耗，而来一旦发生了错误还可以执行回滚操作。

事务的操作并没有特别的接口函数，只是一个普通的sql语句，如下实例：

int ret = sqlite3\_exec(pDb, “begin transaction”, 0, 0, &errmsg); //开始事务

int ret = sqlite3\_exec(pDb, “commit transaction”, 0, 0, &errmsg); //提交事务

int ret = sqlite3\_exec(pDb, “rollback transaction”, 0, 0, &errmsg); //回滚事务

# 5.数据同步

## 5.1.设置方式

PRAGMA synchronous=FULL; //其值是2

PRAGMA synchronous=NORMAL; //值是1

PRAGMA synchronous=OFF; //值是0

## 5.2.参数含义

FULL(2)，sqlite数据库引擎会在紧急时刻暂停，以确认数据已经写入到磁盘(sync函数)。这样在系统断电或者电源出问题时就能够确保数据库重启后数据不出问题，但这种模式下，很慢。

NORMAL(1)，大部分紧急时刻会暂停，但不像FULL模式下频繁。这种模式下又很小的几率是回出现由于电源等问题导致数据库损坏的情况的，但一般来说这种情况下硬盘都已经发生了故障。

OFF(0)，sqlite在传递数据给系统以后直接继续而不暂停。如果运行sqlite的应用程序崩溃，数据不会损伤，但在系统崩溃或写入时意外断电，可能导致数据库损坏。但是这种模式下，一些操作可能速度会有极大提升。

在sqlite2中，默认设置为NORMAL；在sqlite3中，默认设置为FULL。

**//TODO，这里设置为OFF的操作执行成功了，但在事务中并未发现明显的效率提高，需要以后明确该部分的用例实现**

Mysql

# 1.概述

## 1.1.安装

sudo apt-get install mysql-server;

sudo apt-get install mysql-client;

如上安装后，通过：mysql -u root -p 可以正常登录，说明安装成功；

有一些版本，mysql-server安装时，会要求输入密码，有一些不要求输入；如果输入了密码，执行上述命令时就输入该密码就可以；否则不需要输入。

sudo apt-get install libmysqlclient-dev;

执行该语句，安装C和C++开发时需要的动态库和头文件。

## 1.2.使用

查看当前所有的数据库：show databases;

新建数据库：create database dbName;

切换到某一个数据库：use dbName;

建表格：create table tblName(...);

删除数据库：drop database dbName;

删除表格：drop table tblName;

清空表格内容：delete from tblName;

插入一条记录：insert into tblName values(...);

更新表中数据：update tblName set key=value, key=value where key=value and key=value;

# 2.C/C++调用mysql

## 2.1.初始化和反初始化流程

初始化和反初始化，要配对出现，否则容易出现资源泄露：

mysql\_library\_init() mysql\_library\_end()这两个函数，为mysql库提供恰当的初始化和结束处理。对于与客户端库连接的应用程序，他们提供了改进的内存管理功能。

Mysql\_library\_init()的调用可以省略，因为必要的时候，mysql\_init()函数会自动调用它。

最重要的三个函数：

* Mysql\_init()：初始化函数，传入一个MYSQL类型的变量的指针，该函数为指针赋值，后续所有mysql操作都需要这个变量；
* Mysql\_real\_connect()：打开某一个数据库，需要指定mysql的用户名、密码，指定打开哪个数据库，等；只有该函数执行成功，才能对数据库进行增删改查；
* Mysql\_close()：反初始化函数；传入init时初始化的变量指针，完成资源释放；

**Mysql\_real\_connect()**

函数原型：MYSQL \* mysql\_real\_connect(MYSQL\* pDb, const char \* host, const char \* use, const char \* passwd, const char \* db, unsigned int port, const char \* unix\_socket, unsigned long client\_flag);

该函数尝试与运行在主机上的mysql数据库引擎建立连接。在所有其他mysql语句执行之前，该函数必须执行成功。

pDb：通过mysql\_init函数初始化的数据库句柄；

Host：主机名或IP地址。如果该值为NULL或“localhost”，连接将于本地主机相连。如果操作系统支持套接字(unix)或命名管道(windows)，将使用它们而不是TCP/IP连接到服务器。

User：用户名；

Passwd：密码；

Db：要访问的数据库名字；

Port：如果不是0，将作为tcp/ip连接的端口号。注意，host指定了连接的类型；

Unix\_socket如果不是NULL，就描述了应使用的套接字或者命名管道。注意，Host决定了连接的类型！

Client\_flag值通常设置为0.

## 2.2.执行sql语句

### 2.2.1.Mysql\_query()

函数原型：int mysql\_query(MYSQL \*pDb, const char \* cmdStr);

cmdStr是需要被执行的sql语句；

如果sql语句执行成功，返回0；否则返回非0值；

### 2.2.2.获取select的结果

有两种方式获取select语句的查询结果：一种是一次性的将所有结果都读取到内存中，自行查询；另一种是一次一条的向mysql查询结果。

两种方式各有优缺点：

方式1的优点是数据预存到内存中，访问方便快捷，无需再次与数据库交互；缺点是查询条数过多的时候，内存空间可能不足以存放这么多的数据量；

方式2的优点是无需考虑内存限制；缺点是每次都要与数据库进行交互。

#### 方式1

MYSQL\_RES \* pRes = mysql\_store\_result();

Int rowNum = mysql\_num\_rows(pRes);

MYSQL\_ROW row ;

While(NULL != (row = mysql\_fetch\_row(pRes)))

{

//a= row[0], b=row[1], and so on.

}

Mysql\_free\_result(pRes);

#### 方式2

MYSQL\_RES \* pRes = mysql\_use\_result();

MYSQL\_ROW row;

While(NULL != (row = mysql\_fetch\_row(pRes)))

{

//a= row[0], b=row[1], and so on.

}

Mysql\_free\_result(pRes);

## 2.3.获取错误信息

所有mysql函数返回错误时，可以通过

* mysql\_errno(MYSQL \* pDb)得到错误编号；
* 通过mysql\_error(MYSQL \* pDb)获得错误描述；

# 3.事务

//TODO

syslog

# 1.概述

在2001年定义的RFC3164中， 描述了BSD syslog协议，不过这个协议的很多内容都不是强制性的，大多是“建议”或“约定”，也由于这个规范出的比较晚，很多设备并不遵守或不完全遵守这个约定。

约定发送syslog的设备为Device，转发syslog的设备是Relay，接收syslog的设备是Collector。但这个划分并不绝对。例如，一个设备作为Relay接收了Device发送的syslog，但并没有转发，而是直接使用了，那他就同样是一个Collector。

syslog消息发送到collector的UDP的514端口，不需要接收方应答，RFC3164建议Device也是用514作为源端口。

syslog消息使用UDP协议发送，UDP报文不能超过1024个字节，并且全部由可打印的字符组成。完整的syslog消息，由三部分组成：PRI，HEADER，MSG。大部分syslog都包含PRI和MSG部分，而HEADER可能没有。

# 2.格式

如下为一个syslog消息：

<30>Oct 9 22:33:20 hlfedora auditd[1787]: The audit daemon is exiing;

这里的“30”是PRI部分，“Oct 9 22:33:20 hlfedora”是HEADER部分，其余是message部分。

## 2.1.PRI部分

PRI部分由一对尖括号包含的一个数字组成，这个数字包含了程序模块(facility)、严重性(severity)两个信息，其规则是：facility\*8 + severity = PRI。

facility定义如下：

|  |  |
| --- | --- |
| Numerical | Facility |
| 0 | Kernel message |
| 1 | User-level message |
| 2—15 | System using |
| 16—23 | Local use |
|  |  |

由此可见，user(1)和16—23都可以给自有的程序使用。

severity定义如下：

|  |  |
| --- | --- |
| Numerical | Severity |
| 0 | Emergency：system is unusable |
| 1 | Alert: action must be taken immediately |
| 2 | Critical: critical conditions |
| 3 | Error: error conditions |
| 4 | Warning: warning conditions |
| 5 | Notice: normal but significant condition |
| 6 | Infomational: informational message |
| 7 | Debug: debug message |

PRI不允许以0开头，也就是说，除非PRI的值就是0，否则像001和01和00都是不允许的情况，应该是1或者0.

## 2.2.HEADER部分

HEADER部分包括两个字段：时间和主机名(或IP)。

### 2.2.1.时间

时间紧跟在PRI后面，中间没有空格，格式必须是“Mmm dd hh:mm:ss”，不包括年份。

* “日”的数字如果是0-9之间，那么前面将补一个空格，也就是月份后面将出现两个空格；
* 而“时分秒”如果不足两位，则前面补零；
* 月份取值包括：Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec

### 2.2.2.主机名

时间后面跟一个空格，然后是主机名(或IP)，主机名不得包括域名。

XML

# 1.libxml

Libxml2是一个C语言实现的xml程序库，可以简单方便的提供对xml文档的各种操作，支持XPATH查询，并部分支持SXLT转换等功能。

## 1.1.安装方式

sudo apt-get install libxml2

sudo apt-get install libxml2-dev

或可以在官网(http://xmlsoft.org)下载源码后，自行编译并安装；

## 1.2.基本数据类型

### 1.2.1.内部字符类型xmlChar

typedef unsigned char xmlChar; //xmlstring.h

使用unsigned char作为内部字符格式，是因为它能很好的适应UTF-8编码格式，而UTF-8编码正是libxml2的内部编码，其他格式的编码都要经过转换才能再libxml2中使用。

因为总是会用到xmlChar和char之间的类型转换，所以定义了一个宏BAD\_CAST：

#define BAD\_CAST (xmlChar \*) //xmlstring.h

这里只是进行了一个强制类型转换。

### 1.2.2.文件类型及指针

文件类型：xmlDoc

文件类型指针：xmlDocPtr

xmlDoc是个结构体，保存了一个xml文件的相关信息，如文件名、文件类型、子节点等；

xmlDocPtr是一个指针，等同于xmlDoc \*；

### 1.2.3.节点类型及指针

节点类型：xmlNode

节点类型指针：xmlNodePtr

typedef struct \_xmlNode xmlNode;

typedef xmlNode \*xmlNodePtr;

节点之间以链表和树两种形式同时组织起来的，结构体中有next和prev两个指针，组成双向链表；有parent和children两种指针，组织成树；其中还包含如下重要属性：

* content：节点中的文字内容；
* doc：节点所属的文件；
* name：节点的名字；
* ns：节点的namespace；
* properties：节点的属性列表；

### 1.2.4.节点集合类型及指针

节点集合类型：xmlNodeSet

节点集合类型指针：xmlNodeSetPtr

节点集合代表一个由节点组成的变量，节点集合只会作为xpath的查询结果出现，因此被定义在xpath.h中。

typedef struct \_xmlNodeSet xmlNodeSet;

typedef xmlNodeSet \*xmlNodeSetPtr;

struct \_xmlNodeSet

{

int nodeNr; //number of nodes in this set

int nodeMax; //size of array as allocated;

xmlNodePtr \*nodeTab; //array of nodes in no particular order

};

可以看出，节点集合共有三个成员，分别是节点集合的节点数、最大能容纳的节点数、节点数组的头指针。

## 1.3.基本函数

#include <libxml/parser.h>

#include <libxml/tree.h>

#include <libxml/xpath.h>

### 1.3.1.空白字符处理

int xmlKeepBlanksDefault(int val);

该函数可以控制如下场景：解析xml时，是否去除空白字符。如果设置为不去除空白字符，那么这些字符也会被当做一个node来处理。

val == 0：去除空白字符；

val == 1：不去除；

返回1表示设置成功，0表示设置失败。

### 1.3.2.文件载入和保存

xmlDocPtr xmlParseFile(const char \*filename);

将xml文件从硬盘上载入到内存中，并生成DOM树。使用完毕后，需要使用xmlFreeDoc函数释放资源；

如果函数执行成功，返回这个文档的根节点；否则返回NULL；

int xmlSaveFormatFileEnc(const char \*filename, xmlDocPtr cur, const char \*encoding, int format);

将内存中的DOM树，保存在硬盘上，生成一个带格式的xml文件；

cur：需要保存的xml文档；

encoding：需要保存的编码类型，可以为NULL；

format：是否执行格式化。0表示不执行格式化，1表示格式化。只有在xmlIndentTreeOutput设置为1、或者xmlKeepBlanksDefault(0)时，format设置为1才能够生效。

该函数将返回成功写入的字节数。

### 1.3.3.内存载入和输出

xmlDocPtr xmlParseMemory(const char \*buffer, int size);

将一块内存中的数据，以xml格式进行解析，并生成一个DOM树。使用完毕后，需要使用xmlFreeDoc函数释放申请的资源。

buffer：存放xml格式数据的内存区；

size：内存中XML格式数据的长度；

返回值：如果执行成功，将返回根节点，否则返回NULL；

void xmlDocDumpFormatMemoryEnc(xmlDocPtr out\_doc, xmlChar \*\* doc\_txt\_ptr, int \* doc\_txt\_len, const char \*txt\_encoding, int format);

将DOM树导入到内存中，形成一个xml格式的数据。

out\_doc：需要输出成xml格式的xml文档节点；

doc\_txt\_ptr：输出文档的内存区。由本函数在内部申请。使用完成后，需要调用xmlFree()函数释放资源。

doc\_txt\_len：输出文档内存区的长度；

txt\_encoding：输出文档的编码类型；

format：是否执行格式化；同1.3.2中format。

### 1.3.4.创建和释放xml文档

xmlDocPtr xmlNewDoc(const xmlChar \*version);

在内存中创建一个新的xml文档，需使用xmlFreeDoc函数释放资源。

version：xml标准的版本，目前只能设置为“1.0”；

void xmlFreeDoc(xmlDocPtr cur);

释放内存中的xml文档。

### 1.3.5.节点操作

#### 获得根节点

xmlNodePtr xmlDocGetRootElement(xmlDocPtr doc);

获得指定xml文档的根节点；

* 使用得到的根节点，可以遍历整个xml文档；
* xmlNodePtr的next字段，指向下一个同级的xml节点；
* properties字段为xmlAttr类型，指向该xml节点的属性；
* children字段为xmlNodePtr类型，指向该xml节点的子节点；

#### 设置根节点

xmlNodePtr xmlDocSetRootElement(xmlDocPtr doc, xmlNodePtr root);

设置xml文档的根节点。

如果该文档原来有根节点，则返回根节点；否则返回NULL；

#### 获取节点的内容

xmlChar \* xmlNodeGetContent(xmlNodePtr cur);

cur：节点指针；

返回该节点的节点内容。如果该节点没有内容，就返回NULL；如果返回值不是NULL，那么需要调用xmlFree函数释放资源。

#### 设置节点的内容、长度

void xmlNodeSetContentLen(xmlNodePtr cur, const xmlChar \* content, int len);

cur：节点指针；

content：节点的新文本内容；

len：新文本长度；

#### 在节点内容后增加新内容

void xmlNodeAddContentLen(xmlNodePtr cur, const xmlChar \*content, int len);

在节点的原有内容后面增加新的内容；

#### 获取节点的属性

xmlChar \* xmlGetProp(xmlNodePtr node, const xmlChar\* name);

如果指定节点存在name为输入值的属性，那么返回该属性；否则返回NULL；

如果返回值不是NULL，需要使用xmlFree函数释放资源。

#### 设置节点的属性

xmlAttrPtr xmlSetProp(xmlNodePtr node, const xmlChar \*name, const xmlChar \*value);

设置节点的属性，如果已经存在这个属性了，那么使用输入值替换原有的属性值。

返回该属性节点的指针；

## 1.4.XPATH函数

### 1.4.1.生成xpath的上下文句柄

xmlXPathContextPtr xmlXPathNewContext(xmlDocPtr doc);

返回值：该文档的xpath上下文句柄或NULL；

如果返回值不是NULL，那么需要调用xmlXPathFreeContext()函数执行资源的释放。

### 1.4.2.释放xpath的上下文句柄

void xmlXPathFreeContext(xmlXPathContextPtr ctxt);

### 1.4.3.执行xpath表达式

xmlXPathObjectPtr xmlXPathEvalExpression(const xmlChar \* str, xmlPathContextPtr ctxt);

执行xpath的表达式，返回结果是节点集合。

str：xpath表达式；

ctxt：xpath的上下文句柄；

返回值：节点集合或NULL。如果返回值不是NULL，需要调用xmlXPathFreeObject()函数来释放资源。

### 1.4.4.释放xpath表达式结果集合

void xmlXPathFreeObject(xmlXPathObjectPtr obj);

## 1.5.XPATH基础

XPath使用路径表达式，来选取XML文档中的节点或节点集合，节点是通过路径(path)或者步(step)来选取的。

### 1.5.1.路径表达式

|  |  |
| --- | --- |
| 表达式 | 含义 |
| Nodename | 选取此节点的所有子节点 |
| / | 从根节点开始，选取指定节点 |
| // | 不考虑位置，选择所有符合条件的节点 |
| . | 选取当前节点 |
| .. | 选取当前节点的父节点 |
| @ | 选取属性 |

以如下xml文件为示例：

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>

<title lang="eng">Harry Porter</title>

<price>29.99</price>

</book>

<book>

<title lang="eng">Learning everyday</title>

<price>29.98</price>

</book>

</bookstore>

|  |  |
| --- | --- |
| 表达式 | 结果 |
| bookstore | 选取bookstore节点的所有子节点 |
| /bookstore | 从根节点开始，选取路径为bookstore的节点；  如果表达式起始于“/”，那么这个path就代表了一个绝对路径； |
| bookstore/book | 选取所有bookstore的子节点中的book节点 |
| //book | 选取所有book节点，不论其在文档中的位置 |
| bookstore//book | 选取所有bookstore节点的子节点中的book节点，不论其在文档中的位置 |
| //@lang | 选取所有名字是lang的属性，不论其位置 |

### 1.5.2.谓语

谓语用来查找某个特定的节点，或者查找包含某个特定的值的节点。

谓语被嵌在“[]”中。

|  |  |
| --- | --- |
| 表达式 | 结果 |
| /bookstore/book[1] | 选取bookstore节点的第一个book子元素 |
| /bookstore/book[last()] | 选取bookstore节点的最后一个book子元素 |
| /bookstore/book[last()-1] | 选取bookstore节点的倒数第二个book子元素 |
| /bookstore/book[position()<3] | 选取bookstore节点的所有子元素中的前两个 |
| //title[@lang] | 选取所有拥有名字是lang的属性的titile节点 |
| //title[@lang=”eng”] | 选取所有title元素，并且这些元素都有名字是lang的属性，且这些属性的值都是“eng”；  还支持：> >= < <= !=等比较运算符 |
| /bookstore/book[price>35.00] | 选取所有price元素的值大于35的book元素 |
| /bookstore/book[price>35.00]/title | 更进一步，选取所有title元素 |
|  |  |

### 1.5.3.通配符

|  |  |
| --- | --- |
| 通配符 | 描述 |
| \* | 匹配任何元素节点 |
| @\* | 匹配任何属性节点 |
| node() | 匹配任何类型的节点 |
| | | 逻辑或 |

比如：

/bookstore/\*：选取所有bookstore的子节点；

//\*：选取文档中的所有元素；

//title[@\*]：选取所有有属性的title元素；

//book/title | //book/price：选取book元素的所有title和price元素；

## 1.6.ICONV基础

在Linux上进行编码转化时，既可以利用iconv函数族编程实现，也可以利用iconv命令实现。前者在程序中被调用，后者针对文件，将文件从一种编码格式转换为另一种。

### 1.6.1.iconv函数族

#include <iconv.h>

iconv\_t iconv\_open(const char \*tocode, const char \*fromcode);

该函数指定要进行哪两种格式的字符集转换。该函数返回的是一个句柄，供以下两个函数使用。

size\_t iconv(iconv\_t cd, char \*\*inbuf, size\_t \*inbytesleft, char \*\*outbuf, size\_t \*outbytesleft);

此函数执行具体的转换工作。

从inbuf中读取字符，转换后输出到outbuf中。

inbytesleft用来记录还没有转换的字节数，outbytesleft用来记录输出缓冲的剩余空间。

int iconv\_close(iconv\_t cd);

关闭句柄，释放资源。

### 1.6.2.iconv命令

用法：iconv [选项…] [文件…]

#### 选项

-f：也可以“--from-code=”，用来指定原始的文本编码；

-t：也可以“--to-code=”，用来指定目标的文本编码格式；

-l：也可以“--list”，列举所有已知的字符集；

-c：从输出中，忽略无效的字符；

-o：也可以“--output=FILE”，指定输出文件；

--verbose：打印进度信息；

-V：也可以“--version”，打印版本信息；

#### 示例

iconv -f utf-8 -t gb2313 aaa.txt >bbb.txt

将aaa.txt文件，从utf-8类型转换为gb2312类型，并将结果重定向到bbb.txt中。

Opencv

# 1.安装

## 1.1.Linux

首先使用了系统默认的安装方式：sudo apt-get install libopencv-dev

安装完成后，使用：pkg-config --libs opencv 查看opencv的库安装在哪个目录，我们Makefile中需要用到；同样，使用pkg-config --cflags opencv可以看到头文件安装在哪个目录；

但这时候使用 pkg-config --modversion opencv 查看了一下默认安装的版本，是2.4.9，版本太老了，接口用的还是1.0的接口，完全满足不了我的开发需要，因此决定还是卸载了现在的版本后，下载最新的源码，自己编译安装。

卸载：sudo apt-get --purge remove libopencv-dev

sudo apt-get autoremove libopencv-dev

接下来使用源码安装的方式来进行安装。

Opencv的管网：<https://opencv.org>

在其中找到最新版本的程序，windows、android、ios都可以直接下载，但是linux的需要去github上找到源码，clone或者download，下载的3.4.1版本的github地址是：<https://github.com/opencv/opencv/tree/3.4.1>

先安装一些依赖项：

sudo apt-get install build-essential

sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev

sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev

再新建一个目录，用来存放cmake生成的产物：

cd opencvDir; //进入到下载得到的opencv的源码目录

mkdir release;

cd release;

cmake -D CMAKE\_BUILD\_TYPE=RELEASE -D CMAKE\_INSTALL\_PREFIX=/usr/local ..

这时候，release目录下出现了一些产出物，这些产出物是下一步我们编译opencv的依赖项：

make ; make install;

安装完成！

同样可以使用上面的命令来查看opencv的相关信息：

Pkg-config --cflags opencv;

Pkg-config --libs opencv;

Pkg-config --modversion opencv;

要注意的是，opencv的头文件，有opencv和opencv2两个目录，代表了两个version的接口，现在更通用的是opencv2中的C++接口；

而且由于我们cmake的时候，指定了将产出物产出到/usr/local目录下，这个目录是系统默认会找的路径，因此makefile在编写的时候会轻松很多；

ONVIF

# 1.WSDL

WSDL, Web Services Description Language, 网络服务描述语言。

具体来说，WSDL是一种使用XML编写的文档，这种文档可以描述某个web service，包括但不限于：服务的位置，服务提供的操作/方法。

一个WSDL文档的根元素是definitions元素，其中包含7个重要的元素类型：types，import，message，portType，operations，binding和service；

## 1.1.声明

### 1.1.1.XML声明

WSDL是标准xml格式，要以xml声明开始，例如：<xml version=”1.0” encoding=”utf-8”>;

编码格式支持utf-8、utf-16；

### 1.1.2.Definition元素

所有wsdl文档的根元素都是definition元素，示例如下：

<definitions name="BookQuoteWS"  
 targetNamespace="BookQuote"  
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"  
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"  
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
 xmlns="http://schemas.xmlsoap.org/wsdl/">



说明如下：

* Name属性可以不设置，代表别名；
* targetNameSpace属性是默认的命名空间，定义之后整个wsdl文档的其他元素，都使用该命名空间；
* Xmlns是xml规范中的关键字，ns是namespace的缩写，可以不存在；这里要注意，xmlns:mh其实就是这个definition自己，这样做之后，wsdl文档中的其他元素，都可以通过mh::method来引用；

### 1.1.3.Types元素

Types元素用作一个容器，是自定义的特殊数据类型，例如:

<types>

<xsd:simpleType name="ISBN">  
 <xsd:restriction base="xsd:string">  
 <xsd:pattern value="[0-9]{9}[0-9Xx]" />  
 <xsd:restriction>  
 <xsd:simpleType>  
 <types>



自定义了一个type，名字为ISBN，类型为string，取值范围被限定；

### 1.1.4.Import元素

Import元素可以让当前文档使用其定义的其他wsdl文档，例如：

<definitions name="AllMhWebServices" xmlns="http://schemas.xmlsoap.org/wsdl/">  
 <import namespace="BookQuote"  
 location="http://www.Monson-Haefel.com/jwsbook/BookPrice.wsdl"/>  
 <definitions >



Import元素必须声明两个属性：namespace属性和location属性；

Namespace属性必须和要导入的wsdl文档中声明的targetNamesapce属性相同；location属性必须指定一个实际的wsdl文档。

## 1.2.抽象接口

抽象接口包含message、portType和operation；

portType相当于接口类名称；operation相当于接口函数；message相当于函数的参数信息；

### 1.2.1.Message元素

Message元素描述了web服务的有效负载，相当于函数调用中的参数和返回值类型等，例如：

<message name="GetBulkBookPriceRequest">  
 <part name="isbn" type="xsd:string"/>  
 <part name="quantity" type="xsd:int"/>  
 <message>



可以理解为这里定义了一个结构体:

Typedef struct

{

String isbn;

Int quantity;

}Get...Request;

每一个参数，必须使用part来定义，name和type是必须包含的属性；

### 1.2.2.Operation元素

Operation元素使用一个或多个messages消息来定义它的输入、输出以及错误，例如:

<operation name="getBulkBookPrice" parameterOrder="isbn quantity">  
 <input name="request" message="mh:GetBulkBookPriceRequest"/>  
 <output name="prices" message="mh:GetBulkBookPriceResponse"/>  
 <fault name="InvalidArgumentFault" message="mh:InvalidArgumentFault"/>  
 <operation>



Input表示传入参数；

Output表示返回值；

Fault可以不包括，也可以包含；

### 1.2.3.portType元素

portType元素定义了web服务的抽象接口，可以由一个或者多个operation元素组成，每个operation元素定义了一个RPC央视或者文档样式的web服务方法。

## 1.3.协议映射接口

Binding元素将一个抽象的portType映射到一组具体的协议(SOAP或http)、消息传递样式(RPC或document)以及编码样式(literal或SOAP encoding)。

## 1.4.实现接口

Service元素和port元素用来实现具体的业务。

Service中包含一个或多个port元素。每个port元素对应一个不同的web服务，port将一个URL赋予一个特定的binding(location指定)，继而由binding向下继续映射；

可以将两个或多个不同的port元素，将不同的url赋予相同的binding；

Service -- port -- binding -- portType -- message;

## 1.5.Demo-calc

### 1.5.1.文件目录

文件目录结构如下：

calc/

├── apps

│   ├── client

│   ├── Makefile

│   └── server

├── genFiles

│   ├── calc.h

│   ├── client

│   │   ├── calc.nsmap

│   │   ├── soapC.c

│   │   ├── soapClient.c

│   │   ├── soapH.h

│   │   └── soapStub.h

│   └── server

│   │   ├── calc.nsmap

│   ├── soapC.c

│   ├── soapH.h

│   ├── soapServer.c

│   └── soapStub.h

├── tools

│   ├── soapcpp2

│   └── wsdl2h

└── wsdlFiles

└── calc.wsdl

其中：

* Tools目录下的两个工具，是gsoap编译得到的，编译过程详见第2章；
* Wsdl文件，是直接down的标准的测试wsdl文件，位置：<http://www.cs.fsu.edu/~engelen/calc.wsdl；>
* genFiles目录下，存放的是通过wsdl文件得到的源文件和头文件，这是后续要被调用的，不是自行实现的应用层代码；
* Apps目录下存放的是应用层代码；

### 1.5.2.Gsoap生成源码

Gsoap工具生成源码，分为两步：wsdl2h和soapcpp2；

Wsdl2h -c -s -o calc.h calc.wsdl //这条命令根据wsdl文件生成头文件，-c是纯C语言代码，-s是不用stl相关代码；

Soapcpp2 -c -C -x -L calc.h //根据上一部生成的头文件，生成相关的源代码，-c指定纯C代码，-C指定生成client端，-x指定不生成xml文件，-L指定不生成soapClientLib.c文件；

Soapcpp2 -c -S -x -L calc.h //其他参数一致，-S指定生成服务器端的代码；

1.5.1.中已经得到了产出的源文件，client和server各有5个，出去soapClient.c和soapServer.c不同，其他几个都相同，大概描述其作用：

1. soapStub.h文件是存根文件，与calc.h的联系紧密；要注意，client和server都有该文件，但是**文件内容有不同之处**！
2. SoapC.c, soapH.h，soap的序列化和反序列化代码；已经包含了soapStub.h；
3. Calc.nsmap，命名空间定义文件；
4. soapClient.c soapServer.c，封装过一次的服务器端和客户端代码，应用层直接使用这一层的代码进行调用。

需要注意的是，我们应用层代码除了依赖以上生成的代码外，还依赖于源码中的stdsoap2.c文件(如果是C++则依赖于stdsoap2.cpp文件)；

### 1.5.3.应用层代码

分别对client和server进行关键代码概述。

#### Client

Soap\_init(&calcSoap); //struct soap calcSoap, 初始化

Soap\_set\_namespaces(&calcSoap, namespaces); //命名空间

Soap\_call\_ns2\_\_add(...); //调用add接口执行add操作

Soap\_end(&calcSoap); //结束

Soap\_done(&calcSoap); //程序结束

要注意，调用add接口函数的时候，要传入完整的server的url，例如：“127.0.0.1:3030”，该地址用来确认与哪个服务器进行通信。

#### Server

相比于client，main中还需要实现具体的函数实现，我们参照calc.h(wsdl2h工具生成)中的函数声明，得知要实现的函数分别是：ns2\_\_add, ns2\_\_sub, ns2\_\_mul, ns2\_\_div, ns2\_\_pow。

函数实现不再赘述，较为简单，要注意的是，每个函数都需要包含struct soap \* pSoap作为第一个参数。

Main函数中的函数调用过程是：

Soap\_init(&calcSoap); //struct soap calcSoap, 初始化

Soap\_set\_namespaces(&calcSoap, namespaces); //命名空间

Soap\_bind(...); //绑定端口，这里要注意，bind 的时候只绑定了端口，没有绑定ip，测试的时候就可以正常与client通信，如果绑定了ip就不可以正常通信，原因要后续查找一下

Soap\_accept(...); //接收client的请求

Soap\_serve(...); //提供服务

Soap\_end(&calcSoap); //结束

# 2.gSoap

## 2.1.编译

下载gsoap的地址：<http://www.cs.fsu.edu/~engelen/soap.html>

Linux解压缩后，得到源码目录，执行老三步：configure, make, make install, 做几点说明：

./configure --prefix=/home/wujl/test/gsoap-2.8/prefixDir --disable-ssl：prefix指定产出物的目录，不设定到系统默认目录下，而是修改到应用层目录下；由于我的应用暂时不适用https，所以禁止了ssl的编译；

make的时候遇到了几个错误，都是系统缺失某些依赖库导致的，这里将安装了几个库记录如下：apt-get install flex bison libglib2.0-dev openssl libssl-dev；

其中flex和bison，是ubuntu下编译时缺少yacc和lex需要安装的。

Make install之后，可以去prefix指定的目录下查看，应该已经生成了我们需要的二进制工具：wsdl2h和soapcpp2；库文件libgsoap.\*等；

## 2.2.使用

### 2.2.1.二进制工具

二进制工具有2个：wsdl2h和soapcpp2；

Wsdl2h主要用来将格式正确的wsdl文件转换为我们需要的header头文件；soapcpp2用来将wsdl转换得到的header头文件转换得到源文件，这些源文件将被应用层用来实现各种应用；

Wsdl2h工具的使用一般有如下几个参数：wsdl2h -c -s -o xxx.h xxx.wsdl

-c用来指定生成纯C代码；-s用来指定不使用STL库；-o指定目标文件名字；wsdl文件是原始的wsdl文件，可以包含多个；

有些情况下，还会通过-t ./typemap.dat指定一个dat文件以供使用；

Soapcpp2工具的使用一般有如下几个参数：soapcpp2 -c -C -x -L xxx.h; soapcpp2 -c -S -x -L xxx.h;

-c指定生成纯C代码；-C和-S明确是生成client还是server端的代码；-x指定不生成xml文件；-L不生成soapClientLib.c和soapServerLib.c文件；

产出物目录罗列如下：

genFiles/

├── client

│   ├── RemoteDiscoveryBinding.nsmap

│   ├── soapC.c

│   ├── soapClient.c

│   ├── soapH.h

│   ├── soapStub.h

│   └── wsdd.nsmap

├── onvifDiscovery.h

└── server

├── RemoteDiscoveryBinding.nsmap

├── soapC.c

├── soapH.h

├── soapServer.c

├── soapStub.h

└── wsdd.nsmap

这里onvifDiscovery.h是通过wsdl2h工具产出的头文件，soapcpp2工具需要使用，但是应用层并不需要使用；

Client目录下包含了soapcpp2工具生成的所有源文件；server目录下同样包含了所有soapcpp2工具生成的源文件；

### 2.2.2.库

我们使用到的动态库比较简单，因为当前实现的功能比较少，罗列如下：

libs/

├── libgsoap.a

├── libgsoapck.a

# 3.Onvif初探--discovery

Onvif的内容比较多也比较繁杂，先从：onvifServer--discovery着手，进行了尝试。

## 3.1.代码结构

源码目录树如下：

apps/

├── client

│   ├── Makefile

│   └── onvifDiscoveryClient.c

├── libs

│   ├── libgsoap.a

│   ├── libgsoapck.a

│   └── libgsoapssl.a

├── Makefile

├── server

│   ├── Makefile

│   ├── onvifDiscoveryServer.c

│   └── soapClient.c

└── utils

├── stdsoap2.c

├── stdsoap2.h

├── threads.c

├── threads.h

├── wsaapi.c

├── wsaapi.h

├── wsddapi.c

└── wsddapi.h

Client目录为客户端代码，并未实现，无意义；

Libs目录下为用到的库文件，从2.1.章节编译后得到的目录中的lib目录下拷贝过来；

Server目录为实现的服务器端代码，用来搭建一个简单的onvifServer，只支持discovery功能，需要注意的是，我们要用到一些soapClient.c中(soapcpp2工具生成client代码时生成的一个文件)实现的函数，所以直接将该文件引入了源码目录下；

Utils目录是一些gsoap提供的实现一些公共方法的源文件，我们以源码的形式体现在了代码中；

## 3.2.代码流程

自行编写的应用层代码，其实只有一个：server -- onvifDiscoveryServer.c；

对其流程进行如下简单描述：

1. initSoapSocket():
   1. Soap\_init1(); //要注意，不要直接使用soap\_init，而要使用soap\_init1或者soap\_init2，明确指定使用UDP的方式init，用soap\_init不能传入协议方式，默认使用tcp，会造成后续功能不正常，因为onvif是基于组播的，组播是udp的；
   2. Soap\_set\_namespaces(); //命名空间
   3. Soap\_bind(); //端口绑定，绑定onvif协议规定的3702端口
   4. Setsockopt(); //一些socket参数这里设置一下，加入组播组也是这里加入的，组播组的地址是onvif协议规定的239.255.255.250；
2. Soap\_wsdd\_listen(); //循环体中执行该函数，循环接收请求并处理；

Gsoap实现了一套机制，可以自动映射到功能需要的函数中，这里我们要接收客户端的discovery请求，就需要实现wsdd\_event\_Probe函数，只需要实现了这个函数，在接受到客户端的discovery请求时，会自动调用该函数执行server端的功能。

其实除了wsdd\_event\_Probe函数，还有几个函数也需要实现，这些函数都存在于：utils -- wsddapi.h中；这些函数都是只在该头文件中做了声明，但是没有任何源文件中做了实现，所以如果没有实现这些函数，执行make命令编译应用层源代码的时候，会有undefined refrence的错误出现，也可以通过这个编译错误来找有哪些函数需要实现；

我们只做了discovery的处理，因此只实现了wsdd\_event\_Probe函数，其他几个函数，像wsdd\_event\_Hello等都是直接打了桩，没有实现；

wsdd\_event\_Probe函数主要实现步骤有三个：

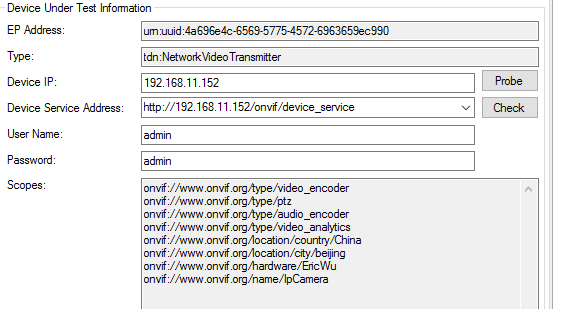
1. Soap\_wsdd\_init\_ProbeMatches:初始化参数
2. Soap\_wsa\_add\_RelatesTo:relatesTo参数赋值，这一步有两种做法：一种是这里显示的调用该函数赋值；第二种是soap初始化的时候，通过调用soap\_register\_plugin(&gOnvifServerSoap, soap\_wsa);也可以。两种方法二选一就可以；
3. Soap\_wsdd\_add\_ProbeMatch:将想要返回给client的server信息通过该函数添加到返回值结构体中；

还要着重说一下几个关键的返回值：

* UUID：每个onvif设备有一个uuid，这个uuid格式："urn:uuid:%8.8x-%4.4x-%4.4x-%4.4x-%4.4x%8.8x"，随机生成数字用来做初始化填充即可；
* DEVICE\_TYPE：这里见过两种写法了："tdn:NetworkVideoTransmitter"和"dn:NetworkVideoTransmitter"，都可以被发现。其他值不能发现，应该是协议定义了一些规则；
* DEVICE\_SCOPE：定义了支持哪些功能，和设备的一些信息，如硬件名字，国家，城市等；
* ServiceAddress：提供onvif协议的url地址，这里看到的都是一种格式："http://%s/onvif/device\_service"，由于没有实现其他onvif功能，所以并不知道是否可以修改；

## 3.3.功能测试

客户端使用了onvif test tool工具来测试，测试结果如下：



## 3.4.遗留问题

有几个问题还没有解决：

1. 打桩的几个函数中，是否还有其他需要实现的函数；
2. Uuid等几个参数的规则，是否在那个协议文档中描述，如何定义；

# 4.播放视频流

在设备可以被成功的发现之后，这一步的目标是可以成功的播放视频流。

## 4.1.编译流程

本以为顺着remoteDiscovery的流程去编译，会是很轻松的事情，但没想到遇到了很多问题，一一记录在此：

### 4.1.1.缺少onvif.xsd和common.xsd文件

在remotediscovery.wsdl之外，新增一个wsdl文件：media.wsdl；执行：../tools/wsdl2h -c -s -o ../genFiles/onvif.h ../wsdlFiles/media.wsdl ../wsdlFiles/remotediscovery.wsdl；这个过程中首先出现了连续两个错误：找不到onvif.xsd和common.xsd这两个文件，这里需要在onvif官网上下载这两个文件到wsdl文件目录下，之后修改media.wsdl文件，将：<xs:import namespace="http://www.onvif.org/ver10/schema" schemaLocation="../../../ver10/schema/onvif.xsd"/>修改为：<xs:import namespace="http://www.onvif.org/ver10/schema" schemaLocation="onvif.xsd"/>，意思是将onvif.xsd文件的寻址目录更换到当前目录；

这时候，再次执行：

../tools/wsdl2h -c -s -o ../genFiles/onvif.h ../wsdlFiles/media.wsdl ../wsdlFiles/remotediscovery.wsdl命令，可以正确执行，并且生成了onvif.h文件；

### 4.1.2.找不到duration.h和ns1.h

这时候执行：../tools/soapcpp2 -c -C -x -L ../genFiles/onvif.h；命令，发现出现如下错误信息：Critical error: #import: Cannot open file "custom/duration.h" for reading.

直观上，是没有找到这个头文件，而且有明确的提示信息：Hint: use option -I<path> (for example -Igsoap:gsoap/import:gsoap/custom:.)，于是直接执行：

../tools/soapcpp2 -c -C -x -L ../genFiles/onvif.h -I../../gsoap-2.8/output/share/gsoap/:../../gsoap-2.8/output/share/gsoap/import/:../../gsoap-2.8/output/share/gsoap/custom:

这次成功执行，得到了我们需要的文件；

### 4.1.3.找不到wsa5\_\_EndpointReference成员

有了以上的步骤，将第3章中的文件和目录结构整体拷贝过来，直接执行make命令，按照预期是会直接通过的，没想到出现了一堆的类似错误：

../utils/wsddapi.c:506:6: error: ‘struct wsdd\_\_HelloType’ has no member named ‘wsa5\_\_EndpointReference’

经查找，发现是由于生成的soapStub.h文件中，定义了一个宏：#define SOAP\_WSA\_2005，这个宏会产生这个错误。于是注释掉这个宏，继续make；

### 4.1.4.重复定义SOAP\_ENV\_\_Fault

编译中出现了：wsa5.h(288): \*\*ERROR\*\*: service operation name clash: struct/class 'SOAP\_ENV\_\_Fault' already declared at wsa.h:273，这个错误。

查阅资料后发现，这个错误是由于onvif.h中同时引用了两个头文件，两者中都有关于这个结构体的定义，因此很多资料都建议将这个结构体该个名字，但我直接选择了注释掉。

这里还有个很奇怪的事情，我使用了/\*\*/的方式注释掉，就可以正常编译了；但使用#if 0 的方式注释，就没有生效，百思不得其解。

### 4.1.5.未定义soap\_in\_xsd\_\_duration

之后，直接进行编译，可以预料到，因为增加了media.wsdl，一定会出现一堆的undefined refrence的错误，果然发现了一些。其中有一批的是关于duration的，这个不用我们自己实现，直接去gsoap源码目录，搜索duration.c文件，拷贝到编译目录下就可以。

### 4.1.6.multiple definition of `namespaces'

将所有的media相关的函数打桩后，编译，这时候出现了新的编译错误：obj/onvifMedia.o:(.data.rel.local+0x0): multiple definition of `namespaces'；

按照这个表述来说，是因为我引入了两个nsmp文件导致的。于是查看了一下生成的两个nsmp文件，发下内容完全一致，于是尝试在media.c中进行了注释，结果编译顺利通过。

## 4.2.代码结构

## 4.3.代码流程

## 4.4.遗留问题

Ffmpeg

# 1.编译安装

Ffmpeg对于h264的支持，依赖于libx264，所以先安装libx264；

下载libx264的源码后，执行configure，make，make install的老三步，configure的时候，我指定了一下参数：./configure --prefix=./output --enable-shared --disable-asm --host=arm-hisiv400-linux --cross-prefix=arm-hisiv400-linux-，这个是对海思平台指定的，一般的linux平台，指定自己的prefix就足够了；

下载ffmpeg的源码后，同样执行configure、make、make install的老三步，configure的时候我同样指定了参数：

./configure --prefix=./output --enable-gpl --enable-shared --disable-debug --disable-x86asm --enable-libx264 --target-os=linux --cc=gcc --extra-cflags=-I/home/wujl/test/x264-snapshot-20180520-2245/output/include --extra-ldflags=-L/home/wujl/test/x264-snapshot-20180520-2245/output/lib

Prefix指定的是产出物目录；

对于刚刚编译出来的libx264，需要用extra-cflags和ldflags来指定才能正常使用；

产出物目录结构如下：

output/

├── bin

├── include

│   ├── libavcodec

│   ├── libavdevice

│   ├── libavfilter

│   ├── libavformat

│   ├── libavutil

│   ├── libpostproc

│   ├── libswresample

│   └── libswscale

├── lib

│   └── pkgconfig

└── share

├── ffmpeg

│   └── examples

└── man

├── man1

└── man3

需要使用的头文件在include目录下，库文件在lib目录下，share目录下包含了一些有用的demo程序和manual；

# 2.Demo

做了几个demo：

* Jpg2yuv：将一个jpg图片转换为一个yuv文件；
* DecodeVideo2Pic：将一段裸流，解码后，将每帧的内容存放为一个图片；

## 2.1.Jpg2yuv

对一个标准的jpg文件进行解码，得到一个原始的yuv文件。

### 2.1.1.流程

实现流程如下：

1. av\_register\_all():向ffmpeg注册所有默认的编解码信息，如果有多种解码需求，其实一次调用就可以，多次调用也无妨；
2. Avformat\_open\_input():这里只是将要解码的jpg文件基本信息传入到变量中，并没有对文件进行任何解析；
   1. 本函数需要调用：avformat\_close\_input()函数进行资源的释放；
3. Avformat\_find\_stream\_info():本函数会对jpg文件的信息进行读取，是真正的处理format的函数，处理完之后，parameters会有效，后续codec的时候会用到这些参数；
4. Avcodec\_find\_decoder():寻找一个解码器，这里需要使用到step3中得到的相关信息；
5. Avcodec\_alloc\_context3():初始化codec的上下文信息；
   1. 本函数需要调用：avcodec\_free\_context()函数来进行资源的释放；
6. Avcodec\_parameters\_to\_context():这里要将step3中得到的相关parameter给到codec的上下文中，后续要使用上下文，而不是parameter了；
7. Avcodec\_open2():到此时，codec的上下文信息已经有了，调用该函数可以根据这些信息，打开编解码器了；
8. Av\_packet\_alloc():申请一个packet的资源；
   1. 本函数需要调用：av\_packet\_free()函数进行资源的释放；
9. Av\_read\_frame():本函数向packet中赋值，由于是一个jpg图片，类似于一个整帧的概念，因此一次读取就足够了；
10. Av\_frame\_alloc():申请一个frame的资源；
    1. 本函数需要调用：av\_frame\_free()函数进行资源的释放；
11. Avcodec\_send\_packet():
12. Avcodec\_receive\_frame():这两个函数，用来将packet的信息，解析到frame的内容中；这两个函数调用完之后，frame中已经包含了yuv的数据了；
13. 将yuv数据保存到文件中；
14. 释放所有资源；

### 2.1.2.后续扩展

现在只支持到了yuv420p格式的输出文件格式，有两个需要扩展的：

1. 支持直接解码成其他格式的yuv文件；这部分可以由ffmpeg完成；
2. 支持yuv文件的格式转换，例如yuv420p转换为420sp等，这个可以自写代码完成；

### 2.1.3.头文件调用须知

如果是c++代码来调用ffmpeg的头文件，需要：

extern "C" {

#include "libavformat/avformat.h"

#include "libavcodec/avcodec.h"

}

因为ffmpeg使用纯c实现，且头文件中并未包含C++调用需要的

## 2.2.DecodeVideo2Pic

对一个标准的裸流文件，每一帧都解析成一个图片，并保存；与jpg2yuv流程大致相同，唯一的不同在于，源是帧数据，而不是标准jpg文件；产出物是pgm文件；

### 2.2.1.流程

其调用ffmpeg的函数流程如下：

1. Av\_register\_all():
2. Avformat\_open\_input():
3. Avformat\_find\_stream\_info():
4. Avcodec\_find\_decoder():
5. Avcodec\_alloc\_context3():
6. Avcodec\_parameter\_to\_context():
7. Avcodec\_open2():
8. Av\_packet\_alloc():
9. Av\_frame\_alloc():至此函数，与jpg2yuv并无差别；但后续的处理方式有差别了，因为jpg只是一个有效图片，但视频流是多个帧数据，这里使用了parser模块；
10. Av\_parser\_init():初始化parser；
11. Looply start：按照block size读取裸流文件；
    1. Av\_parser\_parse2():对读取到的block进行parse；

要注意：这里传入的是一个block，并不是每传入一个block都会调用后续的几个函数进行处理，因为后续的几个函数处理的是一个整帧；这里要根据这个函数中的参数返回值，判断是不是得到了一个整帧，再来调用后续函数；

* 1. Avcodec\_send\_packet():
  2. Avcodec\_receive\_frame():
  3. saveAsPic():

1. 读取完所有帧之后，要发送一个特殊值NULL给到avcodec\_send\_packet()函数，告知其将内存中缓冲的所有数据，都转换为frame并执行后续应用流程，这一部很关键，不调用会导致丢数据；
2. 释放所有资源；

### 2.2.2.后续扩展

现在只支持了从h264文件，解析到pgm图片文件；后续要支持：

1. 解析h265文件；
2. 解析到其他图片格式，如png等；

Libjpeg

Libjpeg可以用来对jpg文件进行：解压缩，压缩，格式转换等功能；

该库缩写为：IJG，Independent libJpeG；

该库依赖性极低，可以单独使用；

# 1.安装

./configure --prefix=...... --enbale-shared=yes --enable-static=no

如果需要在嵌入式平台使用，需要：--host=...... --target=...... CC=cross-compiler.....

Make

Make install

# 2.解压缩

使用该库可以将一个jpg文件直接解压缩得到原始图片数据；也可以直接从内存读取jpg内容后解压缩；

## 2.1.解压缩jpg文件

### 2.1.1.流程

函数调用流程：

1. Jpeg\_std\_error():设置错误处理函数，这里有必要重新设置一下自己的处理函数，因为默认的情况下，libjpeg出现了错误，会直接exit出去，对于应用来说显然很难接受，可以通过setjmp等函数重新设置；
2. Jpeg\_create\_decompress():创建解压缩器；
3. Jpeg\_stdio\_src():从标准文件句柄(FILE \*fp = fopen(...);)得到jpg文件内容；
4. Jpeg\_read\_header():读取jpg文件的文件头；
5. 读取header后，解压缩的格式会使用默认的rgb格式，这里如果需要输出yuv等格式，就需要进行设置了：cinfo.out\_color\_space 中存放了产出物的格式；
6. Jpeg\_start\_decompress():解码；
7. Looply start:
   1. Jpeg\_read\_scanlines():按行读取解压缩后的内容；
   2. Fwrite();将解压缩数据写入文件，该文件就是我们需要的解压缩后的文件；
8. Jpeg\_finish\_decompress():解压缩结束，释放资源；
9. Jpeg\_destroy\_decompress():销毁解压缩器；

### 2.1.2.扩展

#### 解压缩文件的格式

解压缩文件的格式，定义在jpeglib.h中，是一个枚举值：

typedef enum {

JCS\_UNKNOWN, /\* error/unspecified \*/

JCS\_GRAYSCALE, /\* monochrome \*/

JCS\_RGB, /\* red/green/blue, standard RGB (sRGB) \*/

JCS\_YCbCr, /\* Y/Cb/Cr (also known as YUV), standard YCC \*/

JCS\_CMYK, /\* C/M/Y/K \*/

JCS\_YCCK, /\* Y/Cb/Cr/K \*/

JCS\_BG\_RGB, /\* big gamut red/green/blue, bg-sRGB \*/

JCS\_BG\_YCC /\* big gamut Y/Cb/Cr, bg-sYCC \*/

} J\_COLOR\_SPACE;

使用了几种格式：灰度图，RGB，YUV，也就是前三种有效格式；后几种格式的文件，并不知道具体代表何种含义。

要注意，解压缩得到的yuv，默认是yuv444packed格式的，一般的yuv444播放器，能播放的都是planar格式的，因此需要一次转换。

#### Yuv数据格式的转换

应有基本数据格式的转换库，这个自己实现即可；

## 2.2.解压缩内存数据

# 3.压缩

Live555

# 1.编译

## 1.1.ubuntu平台

tar -zxvf live555-latest.tar.gz

./genMakefiles linux-64bit

Make

产出物：

* BasicUsageEnvironment/libBasicUsageEnvironment.a;
* groupsock/libgroupsock.a;
* liveMedia/liblivemedia.a;
* UsageEnvironment/libUsageEnvironment.a;

## 1.2.树莓派

./genMakefiles linux

make

# 2.demo

demo程序源文件：testProgs/\*.cpp；

## 2.1.testOnDemandRTSPServer

本demo程序，顾名思义，是会建立一个rtsp server，并且在有命令请求的时候，进行响应。

程序的函数调用流程：

* TaskScheduler \* scheduler = BasicTaskScheduler::createNew(); //任务调度器
* UsageEnvironment \* env = BasicUsageEnvironment::createNew(\*scheduler); //基础运行环境
* RTSPServer \* rtspServer = RTSPServer::createNew(\*env, 8554, NULL); //rtspServer，8554是占用的port，可以手动修改；NULL是授权信息，如果需要限制用户名密码等，需要指定该值；
* ServerMediaSession \* sms = ServerMediaSession::createNew(\*env, “h264ESVideoTest”, “h264ESVideoTest”, “sessoin streamed by testOnDemandRTSPServer”); //创建一个session，明确指定session的名字，调用者需要知道这个名字，添加在url最末来访问，例如:rtsp://ip:port/h264ESVideoTest；
* sms->addSubSessioin(H264VideoFileServerMediaSubssesion::createNew(\*env, “test.h264”, false)); //session中增加h264的子类型，读取文件的形式完成h264码流原始信息的获取，读取的文件是test.h264；
* env->taskScheduler().doEventLoop(); //死循环，等待请求，播放数据

以上只是H264类型的文件完成rtspServer的调用流程，其他类型的函数调用都没有涉及，但基本都类似。