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；

gRPC-c

# 1.概述

**//TODO， complete them**

RPC

IPC

gRPC

Support languages

Other RPCs

Protobuf

Tool

Language

grpc-c

# 安装

以下安装方式，需要能够联通外网，而且会直接跟github通信，去拿源码，要确保github没有被和谐。如果被和谐了......那你就翻墙吧。

## 2.1.下载源码：

1. 方式1，linux环境下，直接从github上down源码，命令如下：“git clone <https://github.com/Juniper/grpc-c.git> grpc-c”；（注：标准版本的gRPC，其github地址为：“git clone https://github.com/grpc/grpc.git grpc”）；
2. Windows下，登录gRPC的github地址：<https://github.com/Juniper/grpc-c;> （标准版本的grpc：https://github.com/grpc/grpc），下载相对应的压缩包；

## 2.2.基础环境准备

Step1中下载得到的目录是grpc-c，查看其中的README.md文件（vi或者windows下的markdown阅读器），得到其基础环境需要按照如下步骤准备：

sudo apt-get install autoconf automake libtool curl make g++ unzip

cd grpc-c //grpc-c is the directory of grpc we download in step1

autoreconf --install

git submodule update --init //更新init子模块，依赖于网络和github的可访问

./builddeps.sh //这个脚本，会完成grpc-c的依赖工作，具体参见2.2.1.

mkdir build;

cd build; //build目录是我们用来执行老三步(configure make make install的目录)

../configure;

make;

make install;

安装到此结束，如果需要使用到demo程序，那么需要继续执行如下命令：

cd examples; //build目录下的examples目录

make gencode; make; //make后将生成产出物;

产出物得到后，我们可以通过如下程序，测试我们的grpc-c是否正常：

./foo\_server test //server先启动，等待接收client发送过来的消息；

./foo\_client test //client主动发起请求，并收到hello world的返回值；

### 2.2.1.Builddeps.sh

该脚本着重完成依赖性的工作，我们着重看一下：

wujl@wujl:~/MO/gRPC/grpc-c$ cat builddeps.sh

#

# Copyright (c) 2018, Juniper Networks, Inc.

# All rights reserved.

#

#!/bin/sh

pfix=/usr/local //指定产出物的存放目录，这里默认放在了/usr/local下，这个目录一般的程序都可以直接引用到，我也没有改变他，直接使用了

if [ $# -gt 0 ]

then

pfix=$1

fi

echo "Building gRPC"

cd third\_party/grpc

git submodule update --init //第一个重要工具：grpc，更新

make && sudo make prefix=${pfix} install //编译，安装

cd ../../

这里安装的时候出现了error，主要是提示缺少基础环境中需要的c-ares组件，编译给出的建议是先执行git submodule update --init，于是再手动执行一遍，但是报错了：fatal: unable to access 'https://boringssl.googlesource.com/boringssl/'；猜测这个报错是由于谷歌被和谐了导致的，跟c-ares关系不大。

于是网上找了一下c-ares，是一个开源库，主要是用作异步的DNS解析的，libcurl这些库都使用了它，那这个就是一个很通用的底层库了。按照常理来说，应该就在系统里了，只是grpc没有找到而已。于是查看了一下相关资料，下载了一个新版的c-ares，之后执行老三步：./configure;make;make install;这时候去查看一下，/usr/local/lib/下已经多出了我们需要的c-ares的相关库了，再看一下/usr/local/lib/pkgxonfig下已经有了libcares.pc文件了。

于是再次make，成功通过。

手动执行install命令：make prefix=/usr/local install；成功完成。

echo "Installing Protobuf"

cd third\_party/protobuf //第二个重要工具：protobuf

./autogen.sh && ./configure --prefix=${pfx} && make && sudo make prefix=${pfix} install

sudo ldconfig

cd ../../

echo "Building protobuf-c"

cd third\_party/protobuf-c //第三个重要工具：protobuf-c

./autogen.sh && ./configure --prefix=${pfix} && make && sudo make install

cd ../../

# 3.Hello world源码解析

我们在第2章中，在build/examples目录下，通过make;make install;两条命令，得到了我们的测试代码：foo\_server和foo\_client；这两个程序可以完成一个简单的hello world的测试程序，现在我们对这个测试程序进行源码层次的分析，后续可以参照该demo进行一些学习。

本部分同样包含一些基本的gRPC的使用方式。

## 3.1.Protoc

对于gRPC来说，开发流程应该是：

1. 确定客户端和服务器端通信的数据组成；
2. 编写.proto文件；
3. 使用protoc工具将.proto文件，生成为我们需要的.h和.cpp文件；

至此，通信层的工作完成，我们可以通过这些源文件和头文件，完成应用层的调用；

参照这个流程，我们针对foo\_server和foo\_client进行一些印证。

### 3.1.1.寻找proto源文件

我们查看build/examples目录下的makefile文件，它是最直接的来源。我们首先执行了一次：make gencode命令，之后才执行了make，那make gencode命令是做什么用的呢，我们看一下源码：

gencode:

@(for protofile in ${PROTOS} ; do \

${PROTOC} -I ${srcdir} --grpc-c\_out=. \

--plugin=protoc-gen-grpc-c=../compiler/protoc-gen-grpc-c \

$$protofile; \

done)

这段代码告诉我们：所有的proto文件，都是来自于${PTOTOS}，变量同样定义在makefile中：PROTOS = `ls -1 ${srcdir}/\*.proto`，也就是说，我们需要的proto文件，在${srcdir}下，后缀名为proto。

同样的，我们在makefile中找到${srcdir}的定义：srcdir = ../../examples；

这样就明了了，我们需要的proto文件，存在于：grpc-c/examples/下。

### 3.1.2.分析proto源文件

root@wujl:/home/wujl/MO/gRPC/grpc-c/examples# cat foo.proto

syntax = "proto3"; //使用的是protobuf版本是最新的proto3

package foo; //包，类似于C++中的命名空间

// The greeting service definition.

service Greeter { //服务，我们主要使用的类

// Sends a greeting

rpc SayHello (HelloRequest) returns (HelloReply) {} //类的方法，这个也是我们的hello world实现的方法原型

}

// The request message containing the user's name.

message HelloRequest { //请求类型定义

string name = 1; //包含一个字符串，其数字标号为1

}

// The response message containing the greetings

message HelloReply { //返回类型定义

string message = 1; //包含一个字符串，其数字标号为1

}

### 3.1.3.标准的proto格式阐述

Proto文件，作为gRPC的数据通信格式，有自己的一套标准，简要记录一下。

交互式的数据格式，有很多种，当前比较通用的有xml、json等，protobuf是由google提出并定义的一种数据交互格式。相比于json、xml，他的优点是使用接口更简单、序列化和解析速度更快、数据size更小传输压力降低。同时google作为世界级大厂，对标准的定义保证了其向后兼容、跨平台、丰富的语言接口等几个特性都做的极为完善。当然，二进制格式自然也有其缺点，最明显的就是不像json、xml等的可读性和自解释性强，网络通信的时候调试难度会大一些。

由于实现使用的是C++，proto语法设计会让C++用户倍感亲切~~

首先看一个例子：

message Person {  
 required string name = 1; // Your name  
 required int32 id = 2; // Your ID  
 optional string email = 3;  
  
 enum PhoneType {  
 option allow\_alias = true;  
 MOBILE = 0;  
 CELLPHONE = 0;  
 HOME = 1;  
 WORK = 2;  
 }  
  
 message PhoneNumber {  
 required string number = 1;  
 optional PhoneType type = 2 [default = HOME];  
 }  
  
 repeated PhoneNumber phone = 4;  
  
 reserved 2, 15, 9 to 11; // reserved保留  
 reserved "foo", "bar";  
 }

* 在protobuf3之前的版本，每一个变量之前都必须使用required、optional、repeated中的一个进行声明，代表这个字段：出现1次、出现0次或1次、出现0次或多次。但是在protobuf3之后的版本中，不再强制，默认都是optional；
* 支持的数据类型有多种，例如：float, double, [s|u]int[32|64], bool, string，同时还支持自定义类型的嵌套。每一个字段后，紧跟一个数字标号，这个标号要是有效范围内唯一的！
* 支持C/C++的注释风格：//；
* 如果后续某些字段不想要了，为了兼容性起见，不应该仅仅注释或者删除掉，而是应该用reserverd关键字对字段、尤其是对数字标号进行保留，防止其他开发者再次使用，造成老的程序版本对这些字段进行解析时出现异常；
* 对于optional的字段，可以使用default关键字提供默认值，这样当message解析时，如果有default值就会使用该值，如果没有就会使用默认的初值；
* Enum中有选项allow\_alias，一旦打开了这个选项，允许同一个枚举值有多个枚举的名字，例如这里的MOBILE和CELLPHONE；

再扩展一些proto的语言特性：

#### 导入其他proto文件

使用import关键字可以将其他的proto文件导入到当前文件，默认的import行为只能使用导入文件中的直接定义，如果需要嵌套使用导入文件中的内容，达到类似递归的效果，可以使用import public，例如：

import public “new.proto”;

import “other.proto”

默认情况下，import在当前目录下寻找对应的proto文件完成导入的动作；但是如果要导入的文件不在当前目录，可以使用--proto\_path来指定目录，例如：protoc --proto\_path=../ ........;

#### 嵌套类型

Message中可以使用嵌套类型，并且不限制嵌套的层次深度，例如：

message A {

message B {

string url = 1;

string title = 2;

}

result result = 1;

}

message C {

1. B result = 1;

}

#### 版本更新须知

对于一个大型系统逐渐演化来说，向后兼容的特性尤为重要，如果要修改一个proto文件，需要遵守一些约定：

* 一个已经存在的字段，不修改其数字标号；
* 不再使用的字段，可以通过OBSOLETE\_前缀表示该字段已经废弃，但一定不能重复使用数字标号，用reserved将这些数字标号保护起来；
* Int32, uint32, int64, uint64, bool都是兼容的，应用可以进行强制类型转换；
* 修改default默认值是允许的，因为default值并不会真正的传输，只跟程序使用的.proto文件有关；

#### Package

我们可以在proto文件的开头声明package，起到类似于命名空间的效果，例如：

package foo; //namespace foo

### 3.1.4.产出对应的h和c文件

我们通过Hello world中的makefile中指定的protoc命令，查看一下通用的产出命令：

protoc -I ../../examples --grpc-c\_out=. \

--plugin=protoc-gen-grpc-c=../compiler/protoc-gen-grpc-c \

$$protofile;

我们逐个画一下重点：

1. 首先指定引用其他.proto文件的目录：“-I ../../examples”，说明需要去改目录找proto文件；这里的-I，其实就是--proto\_path；
2. 指定输出文件的目录：--grpc-c\_out=.；这里指定要产出的产出物输出到当前目录，标准的protoc程序支持--cpp\_out，--csharp\_out, 等多种语言的out目录指定，但并不支持C语言的，猜测这里应该是grpc-c自行添加的；
3. 指定要用来编译出代码头文件和源文件的编译工具：--plugin=......;这里可以看到，用到的编译工具，是build目录下的compiler目录下的一个二进制工具，这个工具是我们自己编译出来的，并不是protoc默认支持的那些；
4. 最后明确本次要编译哪些proto文件；

## 3.2.头文件和源文件

经过3.1.的不懈努力，我们弄清楚了头文件和源文件是怎么来的，ok，接下来我们看一看得到的这两个文件，跟proto文件做一下对应，之后我们就要看怎么用他们了。

### 3.2.1.头文件

头文件的名字是foo.grpc-c.h，可以看到这个名字中的foo来自于我们的proto文件，grpc-c来自于我们的grpc-c protoc工具中指定的plugin名字；

观察头文件的构成，主要有如下几个部分：proto文件中指定的结构体；使用gRPC进行数据传输时，结构体和protobuf的编码解码；对上层应用(client; server;)提供的应用层调用接口；

#### 结构体

我们的proto文件中，定义了两个message，分别是：HelloRequest和HelloReply。逻辑上来说，我们知道他们一个代表了client发给server的request，一个代表了server返回client的response。

这两个message都极为简单，HelloRequest中包含了一个字符串name，HelloResponse中包含了一个字符串message，由此我们对应到foo.grpc-c.h中可以看到两个结构体的定义：

struct \_Foo\_HelloRequest

{

ProtoBufCMessage base;

char \*name;

}

struct \_Foo\_HelloReply

{

ProtoBufCMessage base;

char \* message;

}

每一个结构体中，包含了我们定义的字符串，分别是name和message，除此之外，还存在一个变量base；

这个base变量，我的理解，其实就是类似于C++中的纯虚基类的概念，他的存在，主要的作用应该是为了从C代码向C++过度，因为gRPC本身只支持C++，并不支持C语言。当然到底是不是这个作用，还有待后续继续查看源代码来看，当前也只是猜测。

此外，我们看到这两个结构体的定义，都以“\_”开始，这个C标准一般不建议应用层对外直接使用，所以头文件中做了如下操作：

typedef struct \_Foo\_HelloRequest Foo\_HelloRequest;

typedef struct \_Foo\_HelloReply Foo\_HelloReply;

对于调用者来说，使用Foo\_HelloRequest和Foo\_HelloReply更为合适，更符合C标准的建议。

#### 编码、解码

我们知道，gRPC通信时使用的是protobuf的数据组织形式，这个形式以二进制形式体现，编解码遵循google定义的规则，效率更高。

具体到头文件中，我们可以看到一些编解码函数，完成的就是这个功能。

|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **函数** | **简述** | **其他** |
| 编码 | foo\_hello\_request\_get\_packed\_size | 获取protobufsize | foo\_hello\_request\_packer，依靠这几个函数编码 |
| foo\_hello\_request\_pack | 编码成protobuf |
| foo\_hello\_request\_pack\_to\_buffer | 编码到buf中 |
| foo\_hello\_request\_unpack | 解码到一个Foo\_HelloRequest | Foo\_hello\_request\_unpacker |
| foo\_hello\_request\_free\_unpacked | 释放解码资源 | Foo\_hello\_request\_free |
| 解码 | foo\_hello\_reply\_get\_packed\_size | 获取protobufsize | Foo\_hellp\_reply\_packer依赖这几个函数编码 |
| foo\_hello\_reply\_pack | 编码成protobuf |
| foo\_hello\_reply\_pack\_to\_buffer | 编码到buf中 |
| foo\_hello\_reply\_unpack | 解码到foo\_HelloReply | Foo\_hello\_reply\_unpacker |
| foo\_hello\_reply\_free\_unpacked | 释放解码资源 | Foo\_hello\_reply\_free |

这里所说的编码、解码，指的是我们的可视化的数据格式(结构体)转化为gRPC通信需要的protobuf数据格式；

Client和server都需要编码，同样也都需要解码，因此request和reply同样拥有类似的几个函数。

Free函数是一定要被调用的！！否则内存泄漏是一定会有的。

好了，底层的编解码完成了，我们接下来看到的这几个函数，就是封装了这些底层的编解码函数之后的一层了，主要包含如下几个函数：

* foo\_\_hello\_request\_packer
* foo\_\_hello\_request\_unpacker
* foo\_\_hello\_request\_free
* foo\_\_hello\_reply\_packer
* foo\_\_hello\_reply\_unpacker
* foo\_\_hello\_reply\_free

这几个函数的作用显而易见，不再赘述。

#### 对外接口

理论上说，所有出现在头文件中的内容，外部都可以调用到；但是观察了代码的定位之后，我认为真正对外的接口、或者说是应用层使用的接口，其实只有如下几个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数** | **调用者** | **作用** | **其他** |
| foo\_\_hello\_request\_init | client | 初始化 |  |
| foo\_\_greeter\_\_service\_init | server | 初始化 |  |
| foo\_\_greeter\_\_say\_hello\_cb | Server | 接收到客户端请求后的处理函数 |  |
| foo\_\_greeter\_\_say\_hello | client | 客户端发请求，收结果 | Foo使用了该接口 |
| foo\_\_greeter\_\_say\_hello\_sync | client | 客户端发请求，收结果 | Foo没使用 |
| foo\_\_greeter\_\_say\_hello\_async | client | 客户端发请求，收结果 | Foo没使用 |

我们可以看到，这样整个对外接口就清晰多了，整个调用方式也更加明确了。OK，后续我们会在3.3.中查看具体的调用方式，但在此之前，还是先去看看codes。

### 3.2.2.源文件

我们得到了两个源文件：foo.grpc-c.server.c和foo.grpc-cc;

#### 首先看server；

源文件中只包含了一个函数：int foo\_\_greeter\_\_service\_init(grpc\_c\_server\_t \* server);该函数包含了2个函数调用：

1. grpc\_c\_methods\_alloc(),该函数用来对传入参数server进行一些必要的初始化动作；
2. 最关键的函数调用：grpc\_c\_register\_method();本函数将3.2.1.中描述的几个函数，包括packer、unpacker等作为参数传递；当然，真正处理请求的foo\_\_greeter\_\_say\_hello\_cb函数也要传递进去。经此之后，相当于我们将所有的函数注册到了server上，server会使用这些函数进行后续的操作；

我们再看一看foo\_\_greeter\_\_say\_hello\_cb函数的实现关键点，这个是最关键的服务器的代码了：

Context->gcc\_stream->read(context, (void \*\*)&h, 0, -1); //读request

Foo\_\_hello\_reply\_init(&r); //初始化foo\_\_HelloReply作为返回值

Context->gcc\_stream\_write(context, &r, 0, -1); //返回值

Context->gcc\_stream->finish(context, &status, 0); //这时候就将返回值，包装成http的形式，发送给client了。

#### 再来看client；

对于client，同样针对较为上层的接口，具体来说，是针对两个接口：foo\_\_hello\_request\_\_init和foo\_\_greeter\_\_say\_hello；

foo\_\_hello\_request\_\_init函数用来进行初始化：该函数首先定义了静态局部变量init\_value,再将其赋值给到传入参数message，之后就结束了。但是静态全局变量init\_value并不那么容易看懂，我们将关键步骤罗列如下：

static Foo\_\_HelloRequest init\_value = FOO\_\_HELLO\_REQUEST\_\_INIT;

#define FOO\_\_HELLO\_REQUEST\_\_INIT \

{ PROTOBUF\_C\_MESSAGE\_INIT (&foo\_\_hello\_request\_\_descriptor) \

, NULL }

#define PROTOBUF\_C\_MESSAGE\_INIT(descriptor) { descriptor, 0, NULL }

const ProtobufCMessageDescriptor foo\_\_hello\_request\_\_descriptor =

{

PROTOBUF\_C\_\_MESSAGE\_DESCRIPTOR\_MAGIC,

"foo.HelloRequest",

"HelloRequest",

"Foo\_\_HelloRequest",

"foo",

sizeof(Foo\_\_HelloRequest),

1,

foo\_\_hello\_request\_\_field\_descriptors,

foo\_\_hello\_request\_\_field\_indices\_by\_name,

1, foo\_\_hello\_request\_\_number\_ranges,

(ProtobufCMessageInit) foo\_\_hello\_request\_\_init,

NULL,NULL,NULL /\* reserved[123] \*/

};

我理解，这些操作，其实是在为我们foo\_HelloRequst结构体中的base变量做初始化。

接下来我们看一下client发送请求并获取返回值的函数：foo\_\_greeter\_\_say\_hello；

这个函数同样简单，直接通过一个调用：grpc\_c\_client\_request\_unary();就完成了，我暂时不进入到这个函数去看实现，这个实现是对grpc的封装层，协议相关性太大，后续有时间了再看，着重说一下这个函数的参数中，包含了我们之前定义好的packer和unpacker，这几个函数这里就被作为参数传递进去之后，将被正常使用。

这一部分，由于向上连接应用层，向下连接封装grpc的layer，所以很多代码看起来似懂非懂，所说也并不一定都正确，有时间的时候，我会继续向下看看，有错误的地方也会改正过来。

## 3.3.调用方式

分析完如上的内容，我们的hello world的demo，就很容易看了。

#### 首先看server：

Server的应用层源文件是：foo\_server.c；

grpc\_c\_init(); //初始化grpc-c的基础环境，这个是grpc-c的层面初始化

grpc\_c\_server\_create(); //初始化一个server指针，类似于一个句柄的概念，后续要使用它进行信息发送等，这个是封装rpc的layer一层的调用；

grpc\_c\_server\_add\_insecure\_http2\_port(); //指定ip和port到server

foo\_greeter\_\_service\_init(); //demo的初始化，我们的应用这一层

grpc\_c\_server\_start(); //启动server

grpc\_c\_server\_wait(); //等待client请求

#### 再来看client：

Client的应用层源文件是：foo\_client.c；

其中只包含一个main函数，查看一下他的几个关键函数调用：

grpc\_c\_init(); //初始化grpc-c的基础环境，这个是grpc-c的层面初始化

grpc\_c\_client\_init(); //初始化一个client指针，类似于一个句柄的概念，后续要使用它进行信息发送等，这个是封装rpc的layer一层的调用；

Foo\_\_hello\_request\_init(); //demo的初始化，这个是我们的应用这一层的了

Foo\_greeter\_say\_hello(); //实际发送hello命令，并接受请求

# 4.其他几个demo

在examples目录下，除了我刚才介绍的foo之外，还有几个proto，分别是：bidi\_streaming.proto, client\_streaming.proto, server\_streaming.proto;

观察一下这几个proto文件，其内容，除了package有些差异，其他并无不同，说明其不同应该体现在应用层了。

看了一下几个应用层的代码，server端并无太大差异，差异体现在client端的代码上，主要的差异，是从foo里的同步方式修改为了使用异步方式，按照字母顺序，先看了bidi的client应用层代码，以其为例说明一下，其他的请自行查阅源码。

## 4.1.Bidi-client-app-codes

源文件：bidi\_streaming\_client.c；

源码的结构比较清晰：main中初始化，之后通过异步方式发送请求，通过一个全局flag明确是否已经收到了server的结果，一个线程负责校验这个flag，等到了结果就退出线程，主线程随之做退出流程。

由于公司环境，不让用visio，因此不画图了。

着重看一下异步的实现：

bidi\_streaming\_\_greeter\_\_say\_hello\_\_async(client, NULL, 0, NULL, &cb, (void \*)1);

函数传入的client参数，可以理解成grpc-c初始化过的client的句柄；

主要的逻辑实现在cb函数中；

函数立即返回，不阻塞；因此如果需要有返回值，需要自行实现线程等待；

再来看一下cb函数具体做了什么（关键代码截取）：

bidi\_streaming\_\_hello\_request\_\_init(&h); //初始化request结构

context->gcc\_stream->write(context, &h, 0, -1);

context->gcc\_stream->write\_done(context, 0, -1); //写请求给server

context->gcc\_stream->read(context, (void \*\*)&r, 0, -1) //等待返回值

context->gcc\_stream->finish(context, NULL, 0); //结束请求，可以置flag了

# 5.自实现的架子

根据以上所述，我自实现了一个hello world，当然主要的代码部分还都是抄的人家的，但做了一些应用层的修改和变动，包括源码目录结构、源码利用方式等，这样修改后，我认为更大程度上可以屏蔽grpc相关的内容，把精力更多的放在应用层代码的编写上。

## 5.1.HelloWorld

### 5.1.1.代码结构

代码的目录结构如下：

myTest/

├── HelloWorld

│   ├── apps

│   │   ├── client

│   │   └── server

│   ├── genFiles

│   └── protoFiles

├── libGrpccCommon

│   ├── inc

│   ├── lib

│   └── src

└── tools

第一级目录下包含：HelloWorld；libGrpccCommon；tools；三个子目录，作用如下：

* HelloWorld中包含了所有的应用层代码，换种通俗说法，就是所有需要我们写的代码；包括了：proto文件，proto生成的.h和.c文件，自实现的client和server的应用层代码源文件；
* LibGrpccCommon中包含了二次封装的grpc-c的一些通用代码，包括但不限于：重新实现的一些字符串函数(例如snprintf\_safe等)，线程池等；这部分代码原来是存在于grpc-c目录中的lib子目录下，且其中包含了子目录，我将其提炼出来，将子目录删除后作为一个统一的目录存在，能够更好的适应编译需求；
* Tools目录包含了我们使用到的工具，现在最主要的就是protoc了，该工具负责将proto文件转换为头文件和源文件；

#### libGrpccCommon的使用方式

目录下包含三个子目录：inc和lib为对外调用时使用的目录；src为源文件目录，其编译后得到的产出物是一个so，也就是lib中包含的so，该so需要被HelloWorld应用层代码调用；

Src目录进入后，可直接执行make命令，编译得到so；

#### HelloWorld应用层代码详细说明

protoFiles中放置proto源文件；

HelloWorld/apps目录下执行：make gencode，会自动寻找../../tools目录下的protoc工具，将本目录下的protoFiles中的proto文件转换为头文件和源文件，保存在genFiles目录中；

apps目录下执行make命令，会分别将client子目录和server子目录中的应用代码进行编译，得到两个产出物：hellocClient和helloServer，这两个产出物就可以被运行了。

### 5.1.2.部署

我们部署需要的文件如下：

.

├── helloClient

├── helloServer

├── libcares.so -> libcares.so.2.2.0

├── libcares.so.2 -> libcares.so.2.2.0

├── libcares.so.2.2.0

├── libgpr.so -> libgpr.so.3.0.0

├── libgpr.so.3 -> libgpr.so.3.0.0

├── libgpr.so.3.0.0

├── libGrpccCommon.so

├── libgrpc.so -> libgrpc.so.3.0.0

├── libgrpc.so.3 -> libgrpc.so.3.0.0

├── libgrpc.so.3.0.0

├── libprotobuf-c.so -> libprotobuf-c.so.1.0.0

├── libprotobuf-c.so.1 -> libprotobuf-c.so.1.0.0

└── libprotobuf-c.so.1.0.0

可以看到，除了我们的二进制文件：helloClient和helloServer外，我们需要的就是几个动态库：libgpr, libgrpc, libprotobuf-c, libcares, libGrpccCommon;

## 5.2.MultiRequests

实现一个更加复杂一些的demo，增加我们对grpc-c的一些认识和理解。

主要分两类功能：基本数据类型的使用；简单业务类型的处理；

### 5.2.1.功能说明

基本数据类型的使用，主要设计了几个case：

* getInt, setInt；
* getFloat, setFloat；
* getDouble, setDouble；
* getString, setString；
* getEnum, setEnum；
* getBasicStructure：结构体只有一层，包含了基本数据类型；
* getRecStructure：结构体内有嵌套其他结构体；

简单业务类型的处理，设计了几个case：

* getInfo, setInfo：获取/设置 一些信息；
* startWrite, stopWrite：启动/停止 服务器上一个写文件的业务

### 5.2.2.源码目录

参考章节5.1.中的HelloWorld代码结构，新建一个代码，目录如下：

myTest/

├── HelloWorld

│   ├── apps

│   │   ├── client

│   │   └── server

│   ├── genFiles

│   └── protoFiles

├── libGrpccCommon

│   ├── inc

│   ├── lib

│   └── src

├── MultiRequests

│   ├── apps

│   │   ├── client

│   │   └── server

│   ├── genFiles

│   └── protoFiles

└── tools

如上，其他目录没有变化，新增了一个MultiRequests目录，其中依然是：apps，genFiles，protoFiles三个目录，每个目录的作用不变；

### 5.2.3.实现Protofile

根据设计的case，实现了相关protoFiles，由于大部分功能重复性较强，因此只摘录几个有代表性的点进行描述，明细可自行查看代码。

#### Utils.proto

这里只定义了一个空类型，类似于我们函数调用中的void（int getValue(void)）的概念。

message TypeNull

{}

由于其并不传送任何有效参数，因此定义为了一个空的结构；

其实原本并不想定义这个结构，但是发现proto文件中如果不包含任何参数，无法生成头文件和源文件，因此构造了这个结构。

这里将其定义为了一个公共文件，其他需要使用到他的文件，直接使用import引用即可，我的multiReq.proto文件，也是这样引用的：import utils.proto;

#### 基本数据类型

以int为例作说明：

message BasicInt

{

int32 value = 1;

}

基本数据类型功能简单，仅包含：get，set两种，针对支持的各个基本数据类型，都做了对应的操作。

#### 包含结构体的结构体

复杂数据类型，设计了两个，一个是简单的结构体，另一个是包含了一个结构体的结构体，针对后者说明如下：

message BasicRStru

{

int32 valInt = 1;

message insideStru

{

int32 valInt = 1;

double valDouble = 2;

}

insideStru valInStru = 2;

string valString = 3;

}

这个结构体里包含了内部的一个结构体：insideStru；

除此之外，还包含了一个int和一个string；

### 5.2.4.生成相关头文件和源文件

cd app; make gencode;

在app目录下的makefile中，使用gencode关键字可以直接将proto文件转换为对应的头文件和源文件，转换后的文件是：

genFiles/

├── multiReq.grpc-c.c

├── multiReq.grpc-c.h

├── multiReq.grpc-c.service.c

├── utils.grpc-c.c

├── utils.grpc-c.h

└── utils.grpc-c.service.c

可见，我们定义了两个proto文件，就出现了2组头文件和源文件；

同HelloWrold一样，client中只需要4个文件(去除两个service文件)，server中这几个文件都需要。

### 5.2.5.应用层代码

应用层代码我只实现了一个基本数据类型int的get、set，和复杂数据类型中包含结构体的结构体的get、set，这两个作为使用样例，其他的参考这两个就可以了。

代码结构与HelloWorld并无二致，不再赘述，有一个点要着重说一下：

我不知道为何，这次生成的头文件和源文件中，出现了一个变量：has\_value，这个变量可以用来标识对应的变量是否存在，如果这个变量不设置为1，那么他对应的变量，将无法传输值！

### 5.2.6.部署运行

这次的部署，大致与HelloWorld相同，但是做了一些新增功能，先看文件目录：

grpccMultiReq

├── libcares.so -> libcares.so.2.2.0

├── libcares.so.2 -> libcares.so.2.2.0

├── libcares.so.2.2.0

├── libgpr.so -> libgpr.so.3.0.0

├── libgpr.so.3 -> libgpr.so.3.0.0

├── libgpr.so.3.0.0

├── libGrpccCommon.so

├── libgrpc.so -> libgrpc.so.3.0.0

├── libgrpc.so.3 -> libgrpc.so.3.0.0

├── libgrpc.so.3.0.0

├── libprotobuf-c.so -> libprotobuf-c.so.1.0.0

├── libprotobuf-c.so.1 -> libprotobuf-c.so.1.0.0

├── libprotobuf-c.so.1.0.0

├── multiReqClient

└── multiReqServer

文件目录并无改变；

multiReqClient和multiReqServer除了支持本机通信，还支持网络内的机器互相访问，本机通信，走的是本地unix通信，通过名字标识，例如：./multiReqServer test; ./multiReqClient test；这样两者会通过test这个名字来进行服务的查找、实现；

局域网内通信，走的是正常的socket通信(http2协议)，例如：./multiReqServer 192.168.11.69 3030; ./multiReqClient 192.168.11.69 3030;这样两者就可以通信了。

# 6.嵌入式平台的尝试

以36平台进行一些尝试。

## 6.1.编译

## 6.2.部署

## 6.3.测试程序