Sqlite开发

# 环境

配置日期：2015年6月1日

Sqlite: sqlite-amalgamation32k-201408060029.zip (源文件)

# An Introduction To The SQLite C/C++ Interface

SQLite C/C++接口简介

## Executive Summary

概要

The following two objects and eight methods comprise the essential elements of the SQLite interface:

以下2个对象和8个函数组成了SQLite接口的基本元素。

### sqlite3

The database connection object. Created by sqlite3\_open() and destroyed by sqlite3\_close().

数据库连接对象。由sqlite3\_open()创建，sqlite3\_close()销毁。

### sqlite3\_stmt

The prepared statement object. Created by sqlite3\_prepare() and destroyed by sqlite3\_finalize().

预处理语句对象。由sqlite3\_prepare()创建，sqlite3\_finalize()销毁。

### sqlite3\_open()

Open a connection to a new or existing SQLite database. The constructor for sqlite3.

打开一个新的或已存在的SQLite数据库连接。sqlite3对象的构造函数。

### sqlite3\_prepare()

Compile SQL text into byte-code that will do the work of querying or updating the database. The constructor for sqlite3\_stmt.

将SQL语句编译为字节码，用以对数据库进行查询或修改。sqlite3\_stmt对象的构造函数。

Compiling An SQL Statement

编译一个SQL语句

sqlite3\_prepare

sqlite3\_prepare\_v2

sqlite3\_prepare16

sqlite3\_prepare16\_v2

函数声明：

int sqlite3\_prepare(

sqlite3 \*db, /\* Database handle \*/

const char \*zSql, /\* SQL statement, UTF-8 encoded \*/

int nByte, /\* Maximum length of zSql in bytes. \*/

sqlite3\_stmt \*\*ppStmt, /\* OUT: Statement handle \*/

const char \*\*pzTail /\* OUT: Pointer to unused portion of zSql \*/

);

To execute an SQL query, it must first be compiled into a byte-code program using one of these routines.

执行一个SQL查询，需要首先将其通过上述调用之一编译为字节码。

The first argument, "db", is a database connection obtained from a prior successful call to sqlite3\_open(), sqlite3\_open\_v2() or sqlite3\_open16(). The database connection must not have been closed.

第一个参数，”db”，是通过之前成功的sqlite3\_open系列调用获取到的数据库连接。期间数据库连接不能够被关闭。

The second argument, "zSql", is the statement to be compiled, encoded as either UTF-8 or UTF-16. The sqlite3\_prepare() and sqlite3\_prepare\_v2() interfaces use UTF-8, and sqlite3\_prepare16() and sqlite3\_prepare16\_v2() use UTF-16.

第二个参数，"zSql"是将要编译的语句，sqlite3\_prepare()和sqlite3\_prepare\_v2()使用UTF-8编码，sqlite3\_prepare16()和sqlite3\_prepare16\_v2()使用UTF-16编码。

If the nByte argument is less than zero, then zSql is read up to the first zero terminator. If nByte is non-negative, then it is the maximum number of bytes read from zSql. When nByte is non-negative, the zSql string ends at either the first '\000' or '\u0000' character or the nByte-th byte, whichever comes first. If the caller knows that the supplied string is nul-terminated, then there is a small performance advantage to be gained by passing an nByte parameter that is equal to the number of bytes in the input string including the nul-terminator bytes as this saves SQLite from having to make a copy of the input string.

如果nByte参数小于0，则zSql将被读取到第一个’\0’。如果nByte非负，则zSql读取到第一个’\0’或者第nByte个字节（首先遇到的为准）。如果调用者确定zSql是’\0’结尾，则nByte设置为包括结束符’\0’的字节数时会有一点性能提升，因为此时SQLite不会复制输入字串。

If pzTail is not NULL then \*pzTail is made to point to the first byte past the end of the first SQL statement in zSql. These routines only compile the first statement in zSql, so \*pzTail is left pointing to what remains uncompiled.

如果pzTail非NULL，则pzTail指向zSql第一个SQL语句结束后的第一个字节。该调用仅编译zSql的第一个语句，因此\*pzTail是未编译的余下的部分。

\*ppStmt is left pointing to a compiled prepared statement that can be executed using sqlite3\_step(). If there is an error, \*ppStmt is set to NULL. If the input text contains no SQL (if the input is an empty string or a comment) then \*ppStmt is set to NULL. The calling procedure is responsible for deleting the compiled SQL statement using sqlite3\_finalize() after it has finished with it. ppStmt may not be NULL.

\*ppStmt指向可被sqlite3\_step()执行的编译好的语句。如果出现错误，\*ppStmt将被置为NULL。如果输入语句不包含SQL（如空字串或注释）则\*ppStmt被置为NULL。使用完成后，由调用者负责使用sqlite3\_finalize()销毁编译后的SQL语句。ppStmt不能为空。

On success, the sqlite3\_prepare() family of routines return SQLITE\_OK; otherwise an error code is returned.

调用成功后，sqlite3\_prepare系列函数返回SQLITE\_OK；否则返回错误代码。

The sqlite3\_prepare\_v2() and sqlite3\_prepare16\_v2() interfaces are recommended for all new programs. The two older interfaces are retained for backwards compatibility, but their use is discouraged. In the "v2" interfaces, the prepared statement that is returned (the sqlite3\_stmt object) contains a copy of the original SQL text. This causes the sqlite3\_step() interface to behave differently in three ways:

新的程序建议使用sqlite3\_prepare\_v2和sqlite3\_prepare16\_v2。两个老接口出于向前兼容的目的而保留，但是不建议使用。在”v2”接口中，返回的预处理语句（sqlite3\_stmt对象）包含原始SQL文本的拷贝。这导致sqlite3\_step()接口在三种情况下的不同行为：

1.If the database schema changes, instead of returning SQLITE\_SCHEMA as it always used to do, sqlite3\_step() will automatically recompile the SQL statement and try to run it again. As many as SQLITE\_MAX\_SCHEMA\_RETRY retries will occur before sqlite3\_step() gives up and returns an error.

1. 如果数据库结构改变，sqlite3\_step()会自动重新编译SQL语句并尝试再次运行，而非像过去一样返回SQLITE\_SCHEMA。当重试次数达到SQLITE\_MAX\_SCHEMA\_RETRY后会返回错误。

2.When an error occurs, sqlite3\_step() will return one of the detailed error codes or extended error codes. The legacy behavior was that sqlite3\_step() would only return a generic SQLITE\_ERROR result code and the application would have to make a second call to sqlite3\_reset() in order to find the underlying cause of the problem. With the "v2" prepare interfaces, the underlying reason for the error is returned immediately.

2. 当错误发生时，sqlite3\_step()会返回详细错误代码或扩展错误代码。老的模式是sqlite3\_step()仅返回SQLITE\_ERROR，程序需要再次调用sqlite3\_reset()获取导致错误的原因。使用”v2”预处理接口，可以立即获取可能导致错误的原因。

3.If the specific value bound to host parameter in the WHERE clause might influence the choice of query plan for a statement, then the statement will be automatically recompiled, as if there had been a schema change, on the first sqlite3\_step() call following any change to the bindings of that parameter. The specific value of WHERE-clause parameter might influence the choice of query plan if the parameter is the left-hand side of a LIKE or GLOB operator or if the parameter is compared to an indexed column and the SQLITE\_ENABLE\_STAT3 compile-time option is enabled.

3. 如果WHERE字句中绑定到宿主参数的值可能影响查询计划的选择，则语句将会被自动重新编译，和发生结构改变时类似，第一次调用sqlite3\_step()会跟踪绑定的参数变化。【…】

### sqlite3\_bind()

Store application data into parameters of the original SQL.

将程序数据存储到原始SQL中。

### sqlite3\_step()

Advance an sqlite3\_stmt to the next result row or to completion.

移动sqlite3\_stmt对象到下一个结果行或结束位置。

### sqlite3\_column()

Column values in the current result row for an sqlite3\_stmt.

sqlite3\_stmt对象当结果行的列值。

### sqlite3\_finalize()

Destructor for sqlite3\_stmt.

sqlite3\_stmt对象的析构函数。

### sqlite3\_close()

Destructor for sqlite3.

sqlite3对象的析构函数

### sqlite3\_exec()

A wrapper function that does sqlite3\_prepare(), sqlite3\_step(), sqlite3\_column(), and sqlite3\_finalize() for a string of one or more SQL statements.

对一个或多个SQL语句字符串的包装函数，依次调用sqlite3\_prepare(), sqlite3\_step(), sqlite3\_column(), 和sqlite3\_finalize()。

## Introduction

SQLite currently has over 200 distinct APIs. This can be overwhelming to a new programmer. Fortunately, most of the interfaces are very specialized and need not be considered by beginners. The core API is small, simple, and easy to learn. This article summarizes the core API.

A separate document, [The SQLite C/C++ Interface](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\intro.html), provides detailed specifications for all C/C++ APIs for SQLite. Once the reader understands the basic principles of operation for SQLite, [that document](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\intro.html) should be used as a reference guide. This article is intended as introduction only and is neither a complete nor authoritative reference for the SQLite API.

## Core Objects And Interfaces

The principal task of an SQL database engine is to evaluate SQL statements. of SQL. To accomplish this, the developer needs two objects:

* The [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) object: sqlite3
* The [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) object: sqlite3\_stmt

Strictly speaking, the [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) object is not required since the convenience wrapper interfaces, [sqlite3\_exec](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\exec.html) or [sqlite3\_get\_table](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\free_table.html), can be used and these convenience wrappers encapsulate and hide the [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) object. Nevertheless, an understanding of [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) is needed to make full use of SQLite.

The [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) and [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) objects are controlled by a small set of C/C++ interface routine listed below.

* [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html)
* [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html)
* [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html)
* [sqlite3\_column()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html)
* [sqlite3\_finalize()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\finalize.html)
* [sqlite3\_close()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\close.html)

Note that the list of routines above is conceptual rather than actual. Many of these routines come in multiple versions. For example, the list above shows a single routine named [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) when in fact there are three separate routines that accomplish the same thing in slightly different ways: [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html), [sqlite3\_open16()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) and [sqlite3\_open\_v2()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html). The list mentions [sqlite3\_column()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) when in fact no such routine exists. The "sqlite3\_column()" shown in the list is place holders for an entire family of routines to be used for extracting column data in various datatypes.

Here is a summary of what the core interfaces do:

|  |  |
| --- | --- |
| [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) | This routine opens a connection to an SQLite database file and returns a [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) object. This is often the first SQLite API call that an application makes and is a prerequisite for most other SQLite APIs. Many SQLite interfaces require a pointer to the [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) object as their first parameter and can be thought of as methods on the [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) object. This routine is the constructor for the [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) object. |
| [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) | This routine converts SQL text into a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) object and returns a pointer to that object. This interface requires a [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) pointer created by a prior call to [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) and a text string containing the SQL statement to be prepared. This API does not actually evaluate the SQL statement. It merely prepares the SQL statement for evaluation.  Think of each SQL statement as a small computer program. The purpose of [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) is to compile that program into object code. The [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) is the object code. The [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) interface then runs the object code to get a result.  New applications should always invoke [sqlite3\_prepare\_v2()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) instead of [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html). The older [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) is retained for backwards compatibility. But [sqlite3\_prepare\_v2()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) provides a much better interface. |
| [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) | This routine is used to evaluate a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) that has been previously created by the [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) interface. The statement is evaluated up to the point where the first row of results are available. To advance to the second row of results, invoke [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) again. Continue invoking [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) until the statement is complete. Statements that do not return results (ex: INSERT, UPDATE, or DELETE statements) run to completion on a single call to [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html). |
| [sqlite3\_column()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) | This routine returns a single column from the current row of a result set for a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) that is being evaluated by [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html). Each time [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) stops with a new result set row, this routine can be called multiple times to find the values of all columns in that row.  As noted above, there really is no such thing as a "sqlite3\_column()" function in the SQLite API. Instead, what we here call "sqlite3\_column()" is a place-holder for an entire family of functions that return a value from the result set in various data types. There are also routines in this family that return the size of the result (if it is a string or BLOB) and the number of columns in the result set.   * [sqlite3\_column\_blob()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_bytes()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_bytes16()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_count()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_count.html) * [sqlite3\_column\_double()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_int()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_int64()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_text()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_text16()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_type()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) * [sqlite3\_column\_value()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) |
| [sqlite3\_finalize()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\finalize.html) | This routine destroys a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) created by a prior call to [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html). Every prepared statement must be destroyed using a call to this routine in order to avoid memory leaks. |
| [sqlite3\_close()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\close.html) | This routine closes a [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) previously opened by a call to [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html). All [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) associated with the connection should be [finalized](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\finalize.html) prior to closing the connection. |

### Typical Usage Of Core Routines And Objects

An application will typically use [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) to create a single [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) during initialization. Note that [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) can be used to either open existing database files or to create and open new database files. While many applications use only a single [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html), there is no reason why an application cannot call [sqlite3\_open()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\open.html) multiple times in order to open multiple [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) - either to the same database or to different databases. Sometimes a multi-threaded application will create separate [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) for each threads. Note that a single [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) can access two or more databases using the [ATTACH](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\lang_attach.html) SQL command, so it is not necessary to have a separate database connection for each database file.

Many applications destroy their [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) using calls to [sqlite3\_close()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\close.html) at shutdown. Or, for example, an application that uses SQLite as its [application file format](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\appfileformat.html) might open [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) in response to a File/Open menu action and then destroy the corresponding [database connection](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) in response to the File/Close menu.

To run an SQL statement, the application follows these steps:

1. Create a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) using [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html).
2. Evaluate the [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) by calling [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) one or more times.
3. For queries, extract results by calling [sqlite3\_column()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\column_blob.html) in between two calls to [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html).
4. Destroy the [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) using [sqlite3\_finalize()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\finalize.html).

The foregoing is all one really needs to know in order to use SQLite effectively. All the rest is optimization and detail.

## Convenience Wrappers Around Core Routines

The [sqlite3\_exec()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\exec.html) interface is a convenience wrapper that carries out all four of the above steps with a single function call. A callback function passed into [sqlite3\_exec()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\exec.html) is used to process each row of the result set. The [sqlite3\_get\_table()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\free_table.html) is another convenience wrapper that does all four of the above steps. The [sqlite3\_get\_table()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\free_table.html) interface differs from [sqlite3\_exec()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\exec.html) in that it stores the results of queries in heap memory rather than invoking a callback.

It is important to realize that neither [sqlite3\_exec()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\exec.html) nor [sqlite3\_get\_table()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\free_table.html) do anything that cannot be accomplished using the core routines. In fact, these wrappers are implemented purely in terms of the core routines.

## Binding Parameters and Reusing Prepared Statements

In prior discussion, it was assumed that each SQL statement is prepared once, evaluated, then destroyed. However, SQLite allows the same [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) to be evaluated multiple times. This is accomplished using the following routines:

* [sqlite3\_reset()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\reset.html)
* [sqlite3\_bind()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\bind_blob.html)

After a [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) has been evaluated by one or more calls to [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html), it can be reset in order to be evaluated again by a call to [sqlite3\_reset()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\reset.html). Think of [sqlite3\_reset()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\reset.html) as rewinding the [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) program back to the beginning. Using [sqlite3\_reset()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\reset.html) on an existing [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) rather than creating a new [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) avoids unnecessary calls to [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html). For many SQL statements, the time needed to run [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) equals or exceeds the time needed by [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html). So avoiding calls to [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) can give a significant performance improvement.

It is not commonly useful to evaluate the exact same SQL statement more than once. More often, one wants to evaluate similar statements. For example, you might want to evaluate an INSERT statement multiple times with different values. Or you might want to evaluate the same query multiple times using a different key in the WHERE clause. To accommodate this, SQLite allows SQL statements to contain [parameters](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\lang_expr.html#varparam) which are "bound" to values prior to being evaluated. These values can later be changed and the same [prepared statement](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) can be evaluated a second time using the new values.

SQLite allows a [parameter](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\lang_expr.html#varparam) wherever a string literal, numeric constant, or NULL is allowed. (Parameters may not be used for column or table names.) A [parameter](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\lang_expr.html#varparam) takes one of the following forms:

* **?**
* **?***NNN*
* **:***AAA*
* **$***AAA*
* **@***AAA*

In the examples above, *NNN* is an integer value and *AAA* is an identifier. A parameter initially has a value of NULL. Prior to calling [sqlite3\_step()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\step.html) for the first time or immediately after [sqlite3\_reset()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\reset.html), the application can invoke the [sqlite3\_bind()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\bind_blob.html) interfaces to attach values to the parameters. Each call to [sqlite3\_bind()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\bind_blob.html) overrides prior bindings on the same parameter.

An application is allowed to prepare multiple SQL statements in advance and evaluate them as needed. There is no arbitrary limit to the number of outstanding [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html). Some applications call [sqlite3\_prepare()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\prepare.html) multiple times at start-up to create all of the [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) they will ever need. Other applications keep a cache of the most recently used [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) and then reuse [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) out of the cache when available. Another approach is to only reuse [prepared statements](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\stmt.html) when they are inside of a loop.

## Configuring SQLite

The default configuration for SQLite works great for most applications. But sometimes developers want to tweak the setup to try to squeeze out a little more performance, or take advantage of some obscure feature.

The [sqlite3\_config()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\config.html) interface is used to make global, process-wide configuration changes for SQLite. The [sqlite3\_config()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\config.html) interface must be called before any [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) are created. The [sqlite3\_config()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\config.html) interface allows the programmer to do things like:

* Adjust how SQLite does [memory allocation](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\malloc.html), including setting up alternative memory allocators appropriate for safety-critical real-time embedded systems and application-defined memory allocators.
* Set up a process-wide [error log](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\errlog.html).
* Specify an application-defined page cache.
* Adjust the use of mutexes so that they are appropriate for various [threading models](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\threadsafe.html), or substitute an application-defined mutex system.

After process-wide configuration is complete and [database connections](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\sqlite3.html) have been created, individual database connections can be configured using calls to [sqlite3\_limit()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\limit.html) and [sqlite3\_db\_config()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\db_config.html).

## Extending SQLite

SQLite includes interfaces that can be used to extend its functionality. Such routines include:

* [sqlite3\_create\_collation()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_collation.html)
* [sqlite3\_create\_function()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_function.html)
* [sqlite3\_create\_module()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_module.html)
* [sqlite3\_vfs\_register()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\vfs_find.html)

The [sqlite3\_create\_collation()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_collation.html) interface is used to create new [collating sequences](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\datatype3.html#collation) for sorting text. The [sqlite3\_create\_module()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_module.html) interface is used to register new [virtual table](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\vtab.html) implementations. The [sqlite3\_vfs\_register()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\vfs_find.html) interface creates new [VFSes](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\vfs.html).

The [sqlite3\_create\_function()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\create_function.html) interface creates new SQL functions - either scalar or aggregate. The new function implementation typically makes use of the following additional interfaces:

* [sqlite3\_aggregate\_context()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\aggregate_context.html)
* [sqlite3\_result()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\result_blob.html)
* [sqlite3\_user\_data()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\user_data.html)
* [sqlite3\_value()](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\value_blob.html)

All of the built-in SQL functions of SQLite are created using exactly these same interfaces. Refer to the SQLite source code, and in particular the [date.c](http://www.sqlite.org/src/doc/trunk/src/date.c) and [func.c](http://www.sqlite.org/src/doc/trunk/src/func.c) source files for examples.

Shared libraries or DLLs can be used as [loadable extensions](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\loadext.html) to SQLite.

## Other Interfaces

This article only mentions the foundational SQLite interfaces. The SQLite library includes many other APIs implementing useful features that are not described here. A [complete list of functions](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\funclist.html) that form the SQLite application programming interface is found at the [C/C++ Interface Specification](file:///D:\devdata\sqlite\SQLite%203.8.10.2\sqlite-doc-3081002\c3ref\intro.html). Refer to that document for complete and authoritative information about all SQLite interfaces.

# NOTE

## VS2008下的调试

VS2008不支持超过32768行源代码的调试（由于pdb文件行号长度限制），无法调试单一sqlite.c文件。之前sqlite提供了分割的文件如sqlite-amalgamation32k-201408060029.zip，将单一.c文件分割为多个.c文件，避免该问题。似乎当前版本（3.10.2）没有提供分割的版本。如需调试，可升级至新版VS。

## 数据存储及转换

### 数据库编码

SQLite支持UTF-8和UTF-16两种编码格式。使用UTF-16时，由本地字节序决定是 UTF-16BE 还是UTF-16LE。编码转换由相应函数内部完成，使用sqlite\_open打开的文件，一样可以调用sqlite3\_column\_text16获取UTF-16编码的数据。

### 数据类型

Most SQL database engines (every SQL database engine other than SQLite, as far as we know) uses static, rigid typing. With static typing, the datatype of a value is determined by its container - the particular column in which the value is stored.

大多数SQL数据库引擎（目前除SQLite之外的所有数据库）使用静态，强数据类型。静态类型的数据类型由它的容器(列)决定。

SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container. The dynamic type system of SQLite is backwards compatible with the more common static type systems of other database engines in the sense that SQL statements that work on statically typed databases should work the same way in SQLite. However, the dynamic typing in SQLite allows it to do things which are not possible in traditional rigidly typed databases.

SQLite使用更加通用的动态数据类型系统。SQLite中值的数据类型和值本身相关联，而非其容器。

简单来说，SQLite的数据存储和数据类型是分离的，获取值时根据需要可将存储的值解释为不同数据类型。

存储类型包括：

* NULL

The value is a NULL value.

NULL值

* INTEGER

The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.

有符号整形，根据具体值保存为1,2,3,4,6,8个字节中。

* REAL

The value is a floating point value, stored as an 8-byte IEEE floating point number.

浮点数，保存为8字节IEEEE浮点类型

* TEXT

The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).

字符串，使用数据库编码保存。

* BLOB

The value is a blob of data, stored exactly as it was input.

二进制数据。

### sqlite3\_column代码解析

* 结论sqlite3\_column根据类型获取值会有较高的效率

REAL类型调用sqlite3\_column\_text会首先将浮点型格式化为字符串

TEXT类型调用sqlite3\_column\_double会尝试将字符串转换为浮点（失败返回0）

* 调用sqlite3\_step时准备相应行数据放入pStmt ->pResultSet中
* 调用columnMem(pStmt,i)获取到相应字段的Mem指针pOut = pStmt ->pResultSet[i]

/\*

\*\* Internally, the vdbe manipulates nearly all SQL values as Mem

\*\* structures. Each Mem struct may cache multiple representations (string,

\*\* integer etc.) of the same value.

\*/

struct Mem {

sqlite3 \*db; /\* The associated database connection \*/

char \*z; /\* String or BLOB value \*/

double r; /\* Real value \*/

union {

i64 i; /\* Integer value used when MEM\_Int is set in flags \*/

int nZero; /\* Used when bit MEM\_Zero is set in flags \*/

FuncDef \*pDef; /\* Used only when flags==MEM\_Agg \*/

RowSet \*pRowSet; /\* Used only when flags==MEM\_RowSet \*/

VdbeFrame \*pFrame; /\* Used when flags==MEM\_Frame \*/

} u;

int n; /\* Number of characters in string value, excluding '\0' \*/

u16 flags; /\* Some combination of MEM\_Null, MEM\_Str, MEM\_Dyn, etc. \*/

u8 enc; /\* SQLITE\_UTF8, SQLITE\_UTF16BE, SQLITE\_UTF16LE \*/

#ifdef SQLITE\_DEBUG

Mem \*pScopyFrom; /\* This Mem is a shallow copy of pScopyFrom \*/

void \*pFiller; /\* So that sizeof(Mem) is a multiple of 8 \*/

#endif

void (\*xDel)(void \*); /\* If not null, call this function to delete Mem.z \*/

char \*zMalloc; /\* Dynamic buffer allocated by sqlite3\_malloc() \*/

};

* 调用相应sqlite3\_value函数获取数据

按位与flags，判断是否含有所需类型数据，如果没有，则调用相应函数进行转换。

/\* One or more of the following flags are set to indicate the validOK

\*\* representations of the value stored in the Mem struct.

\*\*

\*\* If the MEM\_Null flag is set, then the value is an SQL NULL value.

\*\* No other flags may be set in this case.

\*\*

\*\* If the MEM\_Str flag is set then Mem.z points at a string representation.

\*\* Usually this is encoded in the same unicode encoding as the main

\*\* database (see below for exceptions). If the MEM\_Term flag is also

\*\* set, then the string is nul terminated. The MEM\_Int and MEM\_Real

\*\* flags may coexist with the MEM\_Str flag.

\*/

#define MEM\_Null 0x0001 /\* Value is NULL \*/

#define MEM\_Str 0x0002 /\* Value is a string \*/

#define MEM\_Int 0x0004 /\* Value is an integer \*/

#define MEM\_Real 0x0008 /\* Value is a real number \*/

#define MEM\_Blob 0x0010 /\* Value is a BLOB \*/

#define MEM\_AffMask 0x001f /\* Mask of affinity bits \*/

#define MEM\_RowSet 0x0020 /\* Value is a RowSet object \*/

#define MEM\_Frame 0x0040 /\* Value is a VdbeFrame object \*/

#define MEM\_Undefined 0x0080 /\* Value is undefined \*/

#define MEM\_Cleared 0x0100 /\* NULL set by OP\_Null, not from data \*/

#define MEM\_TypeMask 0x01ff /\* Mask of type bits \*/

* 简单示例

字段类型为REAL，值为12.3

则flag为Mem\_Real，r=12.3，u为0，z为0，n为0

sqlite3\_column\_double：直接返回r

sqlite3\_column\_int：调用doubleToInt64转换为整形

sqlite3\_column\_text：调用sqlite3VdbeMemStringify转换为字串z，并写入长度n

sqlite3\_column\_blob：同上

sqlite3\_column\_bytes：如果已转换为text，返回n，否则先转为text

* 备注

BLOB值的flag为MEM\_Blob|MEM\_Str，所以对于非BLOB类型数据，实际是转换为TEXT。

## 列信息/列名获取

### 使用SQL查询语句

通过SQL查询语句 pragma table\_info(table\_name)查询table\_name的表结构，获取列名等信息。推荐使用该方法。获取到的信息包括：

|  |  |
| --- | --- |
| cid | 列ID |
| name | 列名称 |
| type | 列数据类型 |
| notnull | 是否允许非空 |
| dflt\_value | 默认值 |
| pk | 是否主键 |

### 使用内部函数sqlite3FindTable

不建议使用该方式。因为该函数声明为SQLITE\_PRIVATE（即static，仅在sqlite.c文件中能够使用），SQLITE\_PRIVATE函数的直接调用可能带来未知的问题。

* 直接调用（不建议）

可以在SQLITE.C中将其声明和定义更改为SQLITE\_API，并在调用前添加如下声明：

extern "C"

{

struct Table;

Table \*sqlite3FindTable(sqlite3\*,const char\*, const char\*);

};

* 间接调用

在SQLITE.C中新增一个SQLITE\_API类型的函数，完成查询并返回结果，外部调用该函数。

* 注意事项

打开数据库后，直接调用sqlite3FindTable返回结果为NULL，需要首先进行一次查询才可以获取到Table信息：

char\* szSqlSelTable = "select name from sqlite\_master where type = 'table'";

nRet = sqlite3\_exec(pdb, szSqlSelTable, 0, 0, 0);

if (nRet != SQLITE\_OK)

{

fprintf(stderr, "szSqlSelTable error\n");

}

Table \*pTab;

pTab = sqlite3FindTable(pdb, "memberu", 0);

## Shell程序解析

### SQL语句的执行

* main()

rc = process\_input(&data, 0);

读取并解析输入

* process\_input()

rc = shell\_exec(p->db, zSql, shell\_callback, p, &zErrMsg);

SQL语句输入完整后，执行SQL语句

* shell\_callback()

SQL语句的回调，显示输出结果

# 示例

## 使用核心对象及接口

核心函数：

•sqlite3\_open()

•sqlite3\_prepare()

•sqlite3\_step()

•sqlite3\_column()

•sqlite3\_finalize()

•sqlite3\_close()

### 步骤

1. sqlite3\_open打开数据库，获取sqlite3\* pdb

2. sqlite3\_prepare预处理查询语句zSql，得到pStmt

3. sqlite3\_column\_count获取列数，循环调用sqlite3\_step() == SQLITE\_ROW，获取每一行结果

4. 对每一行，sqlite3\_column\_name获取列名，sqlite3\_column\_type获取列类型（枚举），sqlite3\_column\_xxx获取列值。注意：如果字段为NULL，sqlite3\_column\_text返回NULL。

5. sqlite3\_finalize销毁pStmt

6. sqlite3\_close关闭数据库（否则会导致泄漏）

备注：

1. 字段值为NULL时（值类型，和类型无关）sqlite3\_column返回值:

|  |  |
| --- | --- |
| 函数 | 返回值 |
| sqlite3\_column\_type | SQLITE\_NULL  #define SQLITE\_NULL 5 (sqlite3) |
| sqlite3\_column\_text | NULL |
| sqlite3\_column\_blob | NULL |
| sqlite3\_column\_bytes | 0 |
| sqlite3\_column\_double | 0 |
| sqlite3\_column\_int | 0 |

### 示例代码

#include "..\Sqlite3008006\sqlite3.h"

#include <iostream>

#include <sstream>

using namespace std;

/\*

test for sqlite c/c++ interface routine

sqlite3\_open()

sqlite3\_prepare()

sqlite3\_step()

sqlite3\_column()

sqlite3\_finalize()

sqlite3\_close()

\*/

inline const char\* Val(const unsigned char\* pval)

{

const char\* p = (const char\*)pval;

return p ? p : "null";

}

int main(void)

{

char\* szFileName = "D:\\data\\sqlite\\test1.db";

sqlite3\* pdb = NULL;

sqlite3\_stmt\* pStmt = NULL;

int nRet = 0;

nRet = sqlite3\_open(szFileName, &pdb);

if (nRet)

{

printf("open file %s error.\n", szFileName);

sqlite3\_close(pdb);

return -1;

}

try

{

char\* zsql = "select \* from book";

nRet = sqlite3\_prepare\_v2(pdb, zsql, strlen(zsql) + 1, &pStmt, NULL);

if (nRet != SQLITE\_OK || !pStmt)

{

stringstream ss;

ss << "sqlite3\_prepare\_v2 error: " << nRet << endl;

throw ss.str();

}

bool bHeaderOutput = true;

int ncol = sqlite3\_column\_count(pStmt);

if (ncol <= 0)

{

stringstream ss;

ss << "sqlite3\_column\_count returns no data: " << ncol << endl;

throw ss.str();

}

while ((nRet = sqlite3\_step(pStmt)) == SQLITE\_ROW)

{

if (bHeaderOutput)

{

stringstream ss;

ss << sqlite3\_column\_name(pStmt, 0);

for (int ii = 1; ii < ncol; ++ii)

{

ss << "|" << sqlite3\_column\_name(pStmt, ii);

}

ss << endl;

cout << ss.str();

bHeaderOutput = false;

}

stringstream ss2;

ss2 << Val(sqlite3\_column\_text(pStmt, 0));

for (int ii = 0; ii < ncol; ++ii)

{

ss2 << "|" << Val(sqlite3\_column\_text(pStmt, ii));

}

ss2 << endl;

cout << ss2.str();

}

if (nRet == SQLITE\_ERROR)

{

throw -2;

}

}

catch (string e)

{

stringstream ss;

ss << "error: " << e << endl;

cout << ss.str();

}

catch (int e)

{

stringstream ss;

ss << "error "<< e << " : " << sqlite3\_errmsg(pdb) << endl;

cout << ss.str();

}

if (pStmt)

{

nRet = sqlite3\_finalize(pStmt);

pStmt = NULL;

}

nRet = sqlite3\_close(pdb);

pdb = NULL;

return 0;

}

## 核心函数的简易封装

The sqlite3\_exec() interface is a convenience wrapper that carries out all four of the above steps with a single function call. A callback function passed into sqlite3\_exec() is used to process each row of the result set. The sqlite3\_get\_table() is another convenience wrapper that does all four of the above steps. The sqlite3\_get\_table() interface differs from sqlite3\_exec() in that it stores the results of queries in heap memory rather than invoking a callback.

sqlite3\_exec()接口是在一次调用中完成上述四步的简易封装。sqlite3\_exec()的回调函数用来处理结果集的每一列。sqlite3\_get\_table()是另一个类似的封装函数，和sqlite3\_exec()的区别是不使用回调函数，而是将结果集存放在堆内存中。

It is important to realize that neither sqlite3\_exec() nor sqlite3\_get\_table() do anything that cannot be accomplished using the core routines. In fact, these wrappers are implemented purely in terms of the core routines.

值得注意的是，sqlite3\_exec()和sqlite3\_get\_table()仅能完成使用核心函数所完成的工作。实际上，这两个封装完全是用核心函数实现的。

### sqlite3\_exec()

#### 函数声明

执行函数

One-Step Query Execution Interface

int sqlite3\_exec(

sqlite3\* pdb, /\* An open database \*/

const char \*sql, /\* SQL to be evaluated \*/

int (\*callback)(void\*,int,char\*\*,char\*\*), /\* Callback function \*/

void \* parg, /\* 1st argument to callback \*/

char \*\*errmsg /\* Error msg written here if not NULL \*/

);

回调函数

int (\*callback)(void\* parg, // parg from sqlite3\_exec

int ncol, // column count

char\*\*azColValue, // column value array

char\*\*azColName // column name array

),

#### 示例代码

#include <stdio.h>

#include <sqlite3.h>

static int callback(void \*NotUsed, int argc, char \*\*argv, char \*\*azColName){

int i;

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

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

}

printf("\n");

return 0;

}

int main(int argc, char \*\*argv){

sqlite3 \*db;

char \*zErrMsg = 0;

int rc;

if( argc!=3 ){

fprintf(stderr, "Usage: %s DATABASE SQL-STATEMENT\n", argv[0]);

return(1);

}

rc = sqlite3\_open(argv[1], &db);

if( rc ){

fprintf(stderr, "Can't open database: %s\n", sqlite3\_errmsg(db));

sqlite3\_close(db);

return(1);

}

rc = sqlite3\_exec(db, argv[2], callback, 0, &zErrMsg);

if( rc!=SQLITE\_OK ){

fprintf(stderr, "SQL error: %s\n", zErrMsg);

sqlite3\_free(zErrMsg);

}

sqlite3\_close(db);

return 0;

}

### sqlite3\_get\_table()

#### 函数声明

查询函数

int sqlite3\_get\_table(

sqlite3 \*db, /\* An open database \*/

const char \*zSql, /\* SQL to be evaluated \*/

char \*\*\*pazResult, /\* Results of the query \*/

int \*pnRow, /\* Number of result rows written here \*/

int \*pnColumn, /\* Number of result columns written here \*/

char \*\*pzErrmsg /\* Error msg written here \*/

);

azResult是结果字串集，含列名。

例如：查询结果为：

Name | Age

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

Alice | 43

Bob | 28

Cindy | 21

则：

nRow = 3, nColumn = 2

azResult[0] = "Name";

azResult[1] = "Age";

azResult[2] = "Alice";

azResult[3] = "43";

azResult[4] = "Bob";

azResult[5] = "28";

azResult[6] = "Cindy";

azResult[7] = "21";

结果集下标：ncol为第i列列名, row \* ncol + i为第row行第i列的值（row从1开始计数）

释放结果

void sqlite3\_free\_table(char \*\*result);

#### 示例代码

下述代码仅主要部分，完整代码可参考4.1.2

const char\* zSql = "select \* from book";

char\*\* azResults = NULL;

int nRow = 0;

int nCol = 0;

char\* zErrmsg = NULL;

cout << endl << "using sqlite3\_get\_table" << endl;

int nRet = sqlite3\_get\_table(pdb, zSql, &azResults, &nRow, &nCol, &zErrmsg);

if (nRet != SQLITE\_OK)

{

stringstream ss;

ss << "select error: " << zErrmsg << endl;

}

else

{

// col title has one row. So ii is from 0 to nRow

for (int ii = 0; ii <= nRow; ++ii)

{

stringstream ss;

ss << Val(azResults[ii \* nCol]);

for (int jj = 0; jj < nCol; ++jj)

{

ss << "|" << Val(azResults[ii \* nCol + jj]);

}

ss << endl;

cout << ss.str();

}

}