优炫数据库C接口使用手册 2.1

UXSINO
优 炫 软 件

优炫数据库C接口使用手册 2.1

法律声明

优炫数据库管理系统(简称：UXDB) 是由北京优炫软件股份有限公司开发并发布的一款商业性数据库管理系统。

注意

由于产品版本安装或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

北京优炫软件股份有限公司（总部）

- 地址：北京市海淀区学院南路62号中关村资本大厦11层（邮编：100081）
- 网址：http://www.uxsino.com
- 邮箱：<uxdb_support@uxsino.com>
- 电话：010-82886998
- 传真：010-82886338
- 服务热线：400-650-7837

目录

表格清单

范例清单

前言

1. 文档目的

本文档介绍优炫数据库的C接口libuxsql库函数。

2. 文档对象

- 技术支持工程师
- 维护工程师
- 开发工程师

3. 修改记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

表 1. 文档更新记录

数据库版本	发布日期	修改说明
2.1.1.5C	2022-09-21	第一次正式发布。

第 1 章 概述

libuxsql是应用程序员使用UXDB的C接口。libuxsql是一个库函数的集合，它们允许客户端程序传递查询给UXDB后端服务器并且接收这些查询的结果。

libuxsql也是很多其他UXDB应用接口的底层引擎，包括为 C++、Perl、Python、Tcl 和 ECUX 编写的接口。如果使用那些包，某些libuxsql行为很重要。特别是，第 15 章 环境变量、第 16 章 口令文件和第 19 章 SSL 支持描述了任何使用libuxsql的应用的用户可见行为。

本文档末尾（第 22 章 示例程序）包含了一些短程序来展示如何编写使用libuxsql的应用。

使用libuxsql的客户端程序必须包括头文件libuxsql-fe.h并必须与libuxsql库链接在一起。

第 2 章 数据库连接控制函数

下列函数可以建立到UXDB后端服务器的连接。一个应用程序可以同时打开多个后端连接（原因之一就是为了访问多个数据库）。每个连接用一个UXconn对象表示，它从函数UXSQLconnectdb、UXSQLconnectdbParams或UXSQLsetdbLogin得到。注意这些函数将返回一个非空的对象指针，除非正好没有内存来分配UXconn对象。在通过该连接对象发送查询之前，应该调用UXSQLstatus函数来检查返回值以确定是否成功连接。

> **警告**
>
> 如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，开始每个会话时应该从search_path中移除公共可写的方案。可以把参数关键词uxoptions设置为-csearch_path=。也可以在连接后发出UXSQLexec(*conn*, "SELECT ux_catalog.set_config('search_path', '', false)")。这种考虑并非专门针对libuxsql，它适用于每一种用来执行任意SQL命令的接口。

> **警告**
>
> 在 Unix 上，复制一个拥有打开 libuxsql 连接的进程可能导致不可预料的结果，因为父进程和子进程会共享相同的套接字和操作系统资源。出于这个原因，不推荐这样的用法，尽管从子进程执行exec来载入新的可执行代码是安全的。

> **注意**
>
> 在 Windows 上，如果一个单一数据库连接被反复地开启并且关闭，可以提升性能。在内部，libuxsql 为开启和关闭分别调用WSAStartup()和WSACleanup()。WSAStartup()会增加一个内部 Windows 库引用计数而WSACleanup()则会减少之。当引用计数正好为一时，调用WSACleanup()会释放所有资源并且所有 DLL 会被卸载。为了避免这种情况，一个应用可以手动调用WSAStartup()，这样当最后的数据库连接被关闭时资源不会被释放。

UXSQLconnectdbParams

开启一个到数据库服务器的新连接。

```
UXconn *UXSQLconnectdbParams(const char * const *keywords,
        const char * const *values,
        int expand_dbname);
```

这个函数使用从两个以NULL结尾的数组中取得的参数打开一个新的数据库连接。第一个数组keywords被定义为一个字符串数组，每一个元素是一个关键词。第二个数组values给出了每个关键词的值。和下面的UXSQLsetdbLogin不同，可以在不改变函数签名的情况下扩展参数集合，因此使用这个函数（或者与之相似的非阻塞的UXSQLconnectStartParams和UXSQLconnectPoll）对于新应用编程更好。

当前能被识别的参数关键词被列举在第 2.2 节 “参数关键词”中。

当expand_dbname为非零时，*dbname*关键词的值被允许识别为一个连接字符串。只有*dbname*的第一次出现会按这种方式扩展，任何后续*dbname*值会被当做一个普通数据库名处理。有关可能的连接字符串格式的详情请参见第 2.1 节 “连接字符串”。

被传递的数组可以为空，这样就会使用所有默认参数。也可以只包含一个或几个参数设置。两个参数数组应该在长度上匹配。对于参数数组的处理将会停止于keywords数组中第一个非-NULL元素。

如果任何一个参数是NULL或者一个空字符串，那么将会检查相应的环境变量（请参见第 15 章 环境变量。如果该环境变量也没有被设置，那么将使用默认值。

通常，关键词的处理是从这些数组的头部开始并且以索引顺序进行。这样做的效果是，当关键词有重复时，只会保留最后一个被处理的值。因此，通过放置关键词*dbname*，可以决定什么会被*conninfo*字符串所重载，以及什么不会被重载。

UXSQLconnectdb

开启一个到数据库服务器的新连接。

```
UXconn *UXSQLconnectdb(const char *conninfo);
```

这个函数使用从字符串conninfo中得到的参数开启一个新的数据库连接。

被传递的字符串可以为空，这样将会使用所有的默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个URI。请参见第 2.1 节 “连接字符串”。

UXSQLsetdbLogin

开启一个到数据库服务器的新连接。

```
UXconn *UXSQLsetdbLogin(const char *uxhost,
                        const char *uxport,
                        const char *uxoptions,
                        const char *uxtty,
                        const char *dbName,
                        const char *login,
                        const char *pwd);
```

这是UXSQLconnectdb带有固定参数集合的整合。它具有相同的功能，不过其中缺失的参数总是采用默认值。对任意一个固定参数写NULL或一个空字符串将会使它采用默认值。

如果*dbName*包含一个=符号或者具有一个合法的连接URI前缀，它会被当作一个*conninfo*字符串，就像它已经被传递给了UXSQLconnectdb，然后按照为UXSQLconnectdbParams指定的方式应用其余参数。

UXSQLsetdb

开启一个到数据库服务器的新连接。

```
UXconn *UXSQLsetdb(char *uxhost,
    char *uxport,
            char *uxoptions,
```

```
                char *uxtty,
                char *dbName);
```

这是一个调用UXSQLsetdbLogin的宏，其中为*login*和*pwd*参数使用空指针。提供它是为了向后兼容非常早的程序。

UXSQLconnectStartParams
UXSQLconnectStart
UXSQLconnectPoll

以非阻塞的方式建立一个到数据库服务器的连接。

```
UXconn *UXSQLconnectStartParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);

UXconn *UXSQLconnectStart(const char *conninfo);

UxdbPollingStatusType UXSQLconnectPoll(UXconn *conn);
```

这三个函数被用来开启一个到数据库服务器的连接，这样应用执行线程不会因为远程I/O被阻塞。这种方法的要点在于等待 I/O 完成可能在应用的主循环中发生，而不是在UXSQLconnectdbParams或UXSQLconnectdb中，并且因此应用能够把这种操作和其他动作并行处理。

在UXSQLconnectStartParams中，数据库连接使用从keywords和values数组中取得的参数创建，并且被expand_dbname控制，这和之前描述的UXSQLconnectdbParams相同。

在UXSQLconnectStart中，数据库连接使用从字符串conninfo中取得的参数创建，这和之前描述的UXSQLconnectdb相同。

只要满足一些限制，UXSQLconnectStartParams或UXSQLconnectStart或UXSQLconnectPoll都不会阻塞。

- hostaddr和host参数必须被合适地使用，以防止做DNS查询。请参见第 2.2 节 “参数关键词”中这些参数的文档。
- 如果调用UXSQLtrace，确保追踪的该流对象不会阻塞。
- 如后文所述，必须要确保在调用UXSQLconnectPoll之前，套接字处于合适状态。

要开始无阻塞的连接请求，可调用UXSQLconnectStart或者UXSQLconnectStartParams。如果结果为空，则libuxsql无法分配一个新的UXconn结构。否则，一个有效的UXconn指针会被返回（不过还没有表示一个到数据库的有效连接）。接下来调用UXSQLstatus(conn)。如果结果是CONNECTION_BAD，则连接尝试已经失败，通常是因为有无效的连接参数。

如果UXSQLconnectStart成功，下一个阶段是轮询libuxsql，这样它能够继续进行连接序列。使用UXSQLsocket(conn)来获得该数据库连接底层的套接字描述符（警告：不要假定在UXSQLconnectPoll调用之间套接字会保持相同）。如果UXSQLconnectPoll(conn)上一次返回UXRES_POLLING_READING，等到该套接字准备好读取（按照select()、poll()或类似的系统函数所指示的）。则再次调用UXSQLconnectPoll(conn)。反之，如果UXSQLconnectPoll(conn)上一次返回UXRES_POLLING_WRITING，等到该套接字准备好写入，则再次调用UXSQLconnectPoll(conn)。在第一次迭代时，即如果你还没有调

用UXSQLconnectPoll，行为就像是它上次返回了UXRES_POLLING_WRITING。持续这个循环直到UXSQLconnectPoll(conn)返回UXRES_POLLING_FAILED指示连接过程已经失败，或者返回UXRES_POLLING_OK指示连接已经被成功地建立。

在连接期间的任意时刻，该连接的状态可以通过调用UXSQLstatus来检查。如果这个调用返回CONNECTION_BAD，那么连接过程已经失败。如果该调用返回CONNECTION_OK，则该连接已经准备好。如前所述，这些状态同样都可以从UXSQLconnectPoll的返回值检测。在一个异步连接过程中（也只有在这个过程中）也可能出现其他状态。如下状态指示该连接过程的当前阶段，并且可能有助于为用户提供反馈。

CONNECTION_STARTED

等待连接被建立。

CONNECTION_MADE

连接 OK，等待发送。

CONNECTION_AWAITING_RESPONSE

等待来自服务器的回应。

CONNECTION_AUTH_OK

收到认证，等待后端启动结束。

CONNECTION_SSL_STARTUP

协商 SSL 加密。

CONNECTION_SETENV

协商环境驱动的参数设置。

CONNECTION_CHECK_WRITABLE

检查连接是否能够处理写事务。

CONNECTION_CONSUME

在连接上消耗所有剩余的响应消息。
注意，尽管这些常数将被保留（为了维护兼容性），一个应用永远不应该依赖这些状态按照特定顺序出现，或者根本就不依赖它们，或者不依赖状态总是这些文档中所说的值。一个应用可能做如下事情。

```
switch(UXSQLstatus(conn))
{
      case CONNECTION_STARTED:
         feedback = "Connecting...";
         break;

      case CONNECTION_MADE:
         feedback = "Connected to server...";
         break;
.
```

```
.
.
      default:
        feedback = "Connecting...";
}
```

在使用UXSQLconnectPoll时，连接参数connect_timeout会被忽略，判断是否超时是应用的责任。UXSQLconnectStart后面跟着UXSQLconnectPoll循环等效于UXSQLconnectdb。

注意当UXSQLconnectStart或UXSQLconnectstartParams返回一个非空的指针时，必须在用完之后调用UXSQLfinish来处理那些结构和任何相关的内存块。即使连接尝试失败或被放弃时也必须完成这些工作。

UXSQLconndefaults

返回默认连接选项。

```
UXSQLconninfoOption *UXSQLconndefaults(void);

typedef struct
{
  char   *keyword;   /* 该选项的关键词 */
  char   *envvar;    /* 依赖的环境变量名 */
  char   *compiled;  /* 依赖的内建默认值 */
  char   *val;       /* 选项的当前值，或者 NULL */
  char   *label;     /* 连接对话框中域的标签 */
  char   *dispchar;  /* 指示如何在一个连接对话框中显示这个域。值：
                        ""       显示输入的值
                        "*"      口令域 - 隐藏值
                        "D"      调试选项 - 默认不显示 */
  int    dispsize;  /* 用于对话框的以字符计的域尺寸 */
} UXSQLconninfoOption;
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的UXSQLconnectdb选项和它们的当前缺省值。返回值指向一个UXSQLconninfoOption结构的数组，该数组以一个包含空*keyword*指针的条目结束。如果无法分配内存，则返回该空指针。注意当前缺省值（*val*域）将依赖于环境变量和其他上下文。一个缺失或者无效的服务文件将会被无声地忽略掉。调用者必须把连接选项当作只读对待。

在处理完选项数组后，把它交给UXSQLconninfoFree释放。如果没有这么做，每次调用UXSQLconndefaults都会导致一小部分内存泄漏。

UXSQLconninfo

返回被一个活动连接使用的连接选项。

```
UXSQLconninfoOption *UXSQLconninfo(UXconn *conn);
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的UXSQLconnectdb选项和它们的当前缺省值。返回值指向一个UXSQLconninfoOption结构的数组，该数组以一个包含空*keyword*指针的条目结束。上述所有对于UXSQLconndefaults的注解也适用于UXSQLconninfo的结果。

UXSQLconninfoParse

返回从提供的连接字符串中解析到的连接选项。

UXSQLconninfoOption *UXSQLconninfoParse(const char *conninfo, char **errmsg);

解析一个连接字符串并且将结果选项作为一个数组返回，或者在连接字符串有问题时返回NULL。这个函数可以用来抽取所提供的连接字符串中的UXSQLconnectdb选项。返回值指向一个UXSQLconninfoOption结构的数组，该数组以一个包含空*keyword*指针的条目结束。

所有合法选项将出现在结果数组中，但是任何在连接字符串中没有出现的选项的UXSQLconninfoOption的val会被设置为NULL，默认值不会被插入。

如果errmsg不是NULL，那么成功时*errmsg会被设置为NULL， 否则设置为被malloc过的错误字符串以说明该问题（也可以将*errmsg设置为NULL并且函数返回NULL，这表示一种内存耗尽的情况）。

在处理完选项数组后，把它交给UXSQLconninfoFree释放。如果没有这么做，每次调用UXSQLconninfoParse都会导致一小部分内存泄漏。反过来，如果发生一个错误并且errmsg不是NULL，确保使用UXSQLfreemem释放错误字符串。

UXSQLfinish

关闭与服务器的连接。同时释放UXconn对象使用的内存。

void UXSQLfinish(UXconn *conn);

注意，即使与服务器的连接尝试失败（由UXSQLstatus指示），应用也应当调用UXSQLfinish来释放UXconn对象使用的内存。不能在调用UXSQLfinish之后再使用UXconn指针。

UXSQLreset

重置与服务器的通讯通道。

void UXSQLreset(UXconn *conn);

此函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。

UXSQLresetStart
UXSQLresetPoll

以非阻塞方式重置与服务器的通讯通道。

int UXSQLresetStart(UXconn *conn);

UxdbPollingStatusType UXSQLresetPoll(UXconn *conn);

这些函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。它们和上面的UXSQLreset的不

同在于它们工作在非阻塞方式。这些函数受
到UXSQLconnectStartParams、UXSQLconnectStart和UXSQLconnectPoll相同的限制。

要发起一次连接重置，调用UXSQLresetStart。如果它返回 0，那么重置失败。如果返回 1，用与使用UXSQLresetPoll建立连接的相同方法使用UXSQLconnectPoll重置连接。

UXSQLpingParams

UXSQLpingParams报告服务器的状态。它接受与UXSQLconnectdbParams相同的连接参数。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
UXPing UXSQLpingParams(const char * const *keywords,
                       const char * const *values,
                       int expand_dbname);
```

该函数返回下列值之一。

UXSQLPING_OK

服务器正在运行，且可以接受连接。

UXSQLPING_REJECT

服务器正在运行，但是处于一种不允许连接的状态（启动、关闭或崩溃恢复）。

UXSQLPING_NO_RESPONSE

无法连接到服务器。这可能表示服务器没有运行，或者给定的连接参数中有些错误（例如，错误的端口号），或者有网络连接问题（例如，一个防火墙阻断了连接请求）。

UXSQLPING_NO_ATTEMPT

没有尝试连接服务器，因为提供的参数显然不正确，或者有一些客户端问题（例如，内存用完）。

UXSQLping

UXSQLping报告服务器的状态。它接受与UXSQLconnectdb相同的连接参数。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
UXPing UXSQLping(const char *conninfo);
```

返回值和UXSQLpingParams相同。

2.1. 连接字符串

几个libuxsql函数会解析一个用户指定的字符串来获得连接参数。这些字符串有两种被接受的格式：纯关键词 = 值字符串以及URI。URI通常遵循RFC 3986，除非像下文进一步描述的那样允许多主机连接字符串。

2.1.1. 关键词/值连接字符串

在第一种格式中，每一个参数设置的形式都是关键词 = 值。等号两边的空白是可选的。要写一个空值或一个包含空白的值，将它用单引号包围，例如关键词 = 'a value'。值里面的单引号和反斜线必须用一个反斜线转义，即\'和\\。

示例。

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

能被识别的参数关键词在第 2.2 节 “参数关键词”中列出。

2.1.2. 连接 URI

连接URI的一般形式如下。

```
uxsino://[user[:password]@][netloc][:port][,...][/dbname][?param1=value1&...]
```

URI模式标志符可以是uxsino://或uxdb://。每一个URI部分都是可选的。下列示例展示了合法的URI语法。

```
uxsino://
uxsino://localhost
uxsino://localhost:5433
uxsino://localhost/mydb
uxsino://user@localhost
uxsino://user:secret@localhost
uxsino://other@localhost/otherdb?connect_timeout=10&application_name=myapp
uxsino://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

URI的层次部分的组件可以作为参数给出。示例如下。

```
uxsino:///mydb?host=localhost&port=5433
```

在任意URI部分中可以使用百分号编码来包括有特殊含义的符号，例如用%3D替换=。

任何不对应第 2.2 节 “参数关键词”中列出的关键词的连接参数将被忽略并且关于它们的警告消息会被发送到stderr。

为了提高JDBC 连接URI的兼容性，参数ssl=true的实例会被翻译成sslmode=require。

主机部分可能是主机名或一个 IP 地址。要指定一个 IPv6 主机地址，将它封闭在方括号中。

```
uxsino://[2001:db8::1234]/database
```

主机组件会被按照参数host对应的描述来解释。如果主机部分是空或开始于一个斜线，将使用一个 Unix 域套接字连接，否则将启动一个 TCP/IP 连接。不过要注意，斜线是 URI 层次部分中的一个保留字符。因此，要指定一个非标准的 Unix 域套接字目录，要么忽略 URI 中的主机说明并且指定该主机为一个参数，要么在 URI 的主机部分用百分号编码路径。

uxsino:///dbname?host=/var/lib/uxsino
uxsino://%2Fvar%2Flib%2Fuxsino/dbname

可以在一个URI中指定多个主机，每一个都有一个可选的端口。形式为uxsino://host1:port1,host2:port2,host3:port3/的URI等效于host=host1,host2,host3 port=port1,port2,port3形式的连接字符串。每一个主机都将被尝试，直到成功地建立连接。

2.1.3. 指定多个主机

可以指定多个要连接的主机，这样它们会按给定的顺序被尝试。在键/值格式中，host、hostaddr和port选项都接受逗号分隔的值列表。在指定的每一个选项中都必须给出相同数量的元素，这样第一个hostaddr对应于第一个主机名，第二个hostaddr对应于第二个主机名，以此类推。不过，如果仅指定一个port，它将被应用于所有的主机。

在连接URI格式中，在URI的host部分可以列出多个由逗号分隔的host:port对。

不管是哪一种格式，单一的主机名可以被翻译成多个网络地址。常见的示例是一个主机同时具有IPv4和IPv6地址。

当多个主机被指定时或者单个主机名被翻译成多个地址时，所有的主机和地址都将按照顺序被尝试，直至遇到一个成功的。如果没有主机可以到达，则连接失败。如果成功地建立一个连接但是认证失败，也不会尝试列表中剩下的主机。

如果使用了口令文件，可以为不同的主机使用不同的口令。所有其他连接选项对列表中的每一台主机都是相同的，例如不能为不同的主机指定不同的用户名。

2.2. 参数关键词

host

要连接的主机名。如果一个主机名以斜线开始，则表示一个 Unix 域通信而不是 TCP/IP 通信，其值是存储套接字文件的目录名。当host没有指定或者为空时，默认连接到/tmp中的一个 Unix 域套接字。在没有 Unix 域套接字的机器上，默认连接到localhost。

也接受由逗号分隔的主机名列表，这种情况下将依次尝试列表中的主机名，列表中的空项会选择上述的默认行为。详情请参见第 2.1.3 节 “指定多个主机”。

hostaddr

要连接的主机的数字 IP 地址。它应该是标准的 IPv4 地址格式，例如172.28.40.9。如果机器支持 IPv6，也可以使用那些地址。当为这个参数指定一个非空字符串时，总是会使用 TCP/IP 通信。

使用hostaddr代替host允许应用避免一次主机名查找，这对于具有时间约束的应用可能非常重要。不过，GSSAPI 或 SSPI 认证方法以及verify-full SSL 证书验证还是需要主机名。使用规则如下。

- 如果指定了host但没有hostaddr，则会发生主机名查找（在使用UXSQLconnectPoll时，UXSQLconnectPoll第一次考虑这个主机名时会发生查找，可能会导致UXSQLconnectPoll阻塞一段时间）。
- 如果hostaddr被指定且没有host，hostaddr的值给出了服务器的网络地址。如果认证方法要求主机名则连接尝试将会失败。

- 如果host和hostaddr都被指定，hostaddr的值给出服务器的网络地址。host的值将被忽略，除非认证方法要求它，在那种情况下它将被用作主机名。

注意如果host不是网络地址hostaddr上的服务器名，认证很可能会失败。此外，当host和hostaddr都被指定时，host被用来在口令文件中标识该连接（请参见第 16 章 口令文件）。

逗号分隔的hostaddr值列表也被接受，这种情况下将依次尝试列表中的主机。列表中的空项会导致对应的主机名被使用，或者如果对应的主机名也为空时使用默认主机名。详情请参见第 2.1.3 节 “指定多个主机”。

如果没有一个主机名或主机地址，libuxsql将尝试使用一个本地 Unix 域套接字连接，或者在没有 Unix 域套接字的机器上尝试连接到localhost。

port

在服务器主机上要连接的端口号，或者用于 Unix 域连接的套接字文件名扩展。如果在host或hostaddr参数中给出了多个主机，这个参数可以指定一个逗号分隔的端口列表，列表长度与主机列表相同，或者这个参数可以指定一个单一的端口号用于所有的主机。空字符串或者逗号分隔列表中的空项指定的是UXDB编译时建立的默认端口号。

dbname

数据库名。默认和用户名相同。在一般的环境下，会为扩展格式检查该值，请参见第 2.1 节 “连接字符串”。

user

UXDB 要作为哪个用户连接。默认与运行着该应用的用户的操作系统名相同。

password

服务器要求口令认证时要使用的口令。

passfile

指定用于存放口令的文件名（请参见第 16 章 口令文件。默认是~/.uxpass，或者是Microsoft Windows上的%APPDATA%\uxsino\uxpass.conf（如果该文件不存在也不会报错）。

connect_timeout

连接的最大等待时间，以秒为单位（写作一个十进制整数，例如10）。零、负值或者不指定表示无限等待。允许的最小超时是2秒，因此值1会被解释为2。这个超时单独应用于每个主机名或者IP地址。例如，如果指定两个主机并且connect_timeout为5，每个主机都会在5秒内没有建立起连接的情况下超时，因此花费在等待连接上的总时间可能高达10秒。

client_encoding

为连接设置client_encoding配置参数。除了被相应服务器选项所接受的值，还能使用auto从客户端的当前区域（Unix 系统上的LC_CTYPE环境变量）决定正确的编码。

uxoptions

指定在连接开始时发送给服务器的命令行选项。例如，设置这个参数为-c geqo=off会把会话的geqo参数值设置为off。这个字符串中的空格被认为是命令行参数的分隔符，除非用一个反斜线（\）对它转义，用\\可以表示一个字面意义上的反斜线。

application_name

为application_name配置参数指定一个值。

fallback_application_name

为application_name配置参数指定一个候补值。如果通过一个连接参数或UXAPPNAME环境变量没有为application_name给定一个值，将使用这个值。在希望设置一个默认应用名但不希望它被用户覆盖的一般工具程序中指定一个候补值很有用。

keepalives

控制是否使用客户端的 TCP 保持存活机制。默认值是 1，表示打开。但是如果不想要保持存活机制，可以改成 0 表示关闭。对于通过一个 Unix 域套接字建立的连接会忽略这个参数。

keepalives_idle

控制非活跃状态多少秒之后 TCP 向服务器发送一个存活消息。零表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果存活机制被禁用也将忽略这个参数。该参数在TCP_KEEPIDLE、等效的套接字选项可用的系统及 Windows支持，在其他系统没有效果。

keepalives_interval

控制一个 TCP 存活消息多少秒内没有被服务器认可时应该被重传。零表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。该参数在TCP_KEEPIDLE、等效的套接字选项可用的系统及 Windows支持，在其他系统没有效果。

keepalives_count

控制该客户端到服务器的连接被认为断开之前可以丢失的 TCP 存活消息数量。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立的连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。该参数在TCP_KEEPIDLE、等效的套接字选项可用的系统及 Windows支持，在其他系统没有效果。

tcp_user_timeout

在强制关闭连接之前，控制传输数据可能保持未确认的毫秒数。零值表示使用系统默认值。对于通过 Unix 域套接字进行的连接，将忽略此参数。该参数在TCP_USER_TIMEOUT可用的系统上支持，在其他系统没有效果。

tty

被忽略（之前指定向哪里发送服务器调试输出）。

replication

这个选项决定该连接是否应该使用复制协议而不是普通协议。这是UXDB的复制连接以及ux_basebackup类工具在内部使用的协议，但也可以被第三方应用使用。

支持下列值，大小写无关。

true、on、yes、1

连接进入到物理复制模式。

database

连接进入到逻辑复制模式，连接到dbname参数中指定的数据库。

false、off、no、0

常规连接，默认情况。

在物理或者逻辑复制模式中，仅能使用简单查询协议。

gssencmode

此选项决定是否或以何种优先级与服务器协商安全GSS TCP/IP 连接。有如下三种模式。

disable

仅尝试非GSSAPI加密的连接。

prefer（default）

如果存在GSSAPI凭据（即凭据缓存中），先尝试GSSAPI加密连接； 如果失败或没有凭据，则尝试非GSSAPI加密连接。

require

仅尝试GSSAPI加密连接。

对于Unix 域套接字通信，gssencmode被忽略。

sslmode

此选项决定是否或以何种优先级与服务器协商SSL TCP/IP 连接。有如下六种模式。

disable

只尝试非SSL连接。

allow

首先尝试非SSL连接，如果失败再尝试SSL连接。

prefer（默认）

首先尝试SSL连接，如果失败再尝试非SSL连接。

require

只尝试SSL连接。如果存在一个根 CA 文件，以verify-ca被指定的相同方式验证该证书。

verify-ca

只尝试SSL连接，并且验证服务器证书是由一个可信的证书机构颁发的（CA）。

verify-full

只尝试SSL连接，验证服务器证书是由一个可信的CA颁发并且请求的服务器主机名匹配证书中的主机名。

这些选项如何工作的详细描述请参见第 19 章 SSL 支持

对于 Unix 域套接字通信，sslmode会被忽略。

sslcompression

如果设置为1，通过SSL连接发送的数据将被压缩。如果设置为0，压缩将被禁用。默认值为0。如果建立的连接没有使用SSL，则这个参数会被忽略。

现如今SSL压缩被认为是不安全的，因此已经不再推荐使用。OpenSSL 1.1.0默认禁用压缩，并且很多操作系统发行版在前面的版本中也将其禁用，因此如果服务器不接受压缩，将这个参数设置为on不会有任何效果。另一方面，1.0.0之前的OpenSSL不支持禁用压缩，因此对那些版本来说会忽略这个参数，而是否使用压缩取决于服务器。

如果安全性不是主要考虑的问题，在网络是瓶颈的情况下，压缩能够改进吞吐量。如果CPU性能是受限的因素，禁用压缩能提高响应时间和吞吐量。

sslcert

这个参数指定客户端 SSL 证书的文件名，它替换默认的~/.uxsino/uxsino.crt。如果没有建立 SSL 连接，这个参数会被忽略。

sslkey

这个参数指定客户端证书的密钥位置，替代默认的~/.uxsino/uxsino.key的文件名，或者它能够指定一个从外部“引擎”（引擎是OpenSSL的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立 SSL 连接，这个参数会被忽略。

sslrootcert

这个参数指定一个包含 SSL 证书机构（CA）证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。默认值是~/.uxsino/root.crt。

sslcrl

这个参数指定 SSL 证书撤销列表（CRL）的文件名。列在这个文件中的证书如果存在，在尝试认证该服务器证书时会被拒绝。默认值是~/.uxsino/root.crl。

requirepeer

这个参数指定服务器的操作系统用户，例如requirepeer=uxdb。当建立一个 Unix 域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是否运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与 TCP/IP 连接上 SSL 证书相似的服务器认证（注意，如果 Unix 域套接字在/tmp或另一个公共可写的位置，任何用户能开始一个在那里监听的服务器。使用这个参数来保证连接的是一个由可信用户运行的服务器）。这个选项只在实现了peer认证方法的平台支持。

krbsrvname

当用 GSSAPI 认证时，指定使用的 Kerberos 服务名。为了让 Kerberos 认证成功，必须匹配在服务器配置中指定的服务名。

gsslib

用于GSSAPI身份验证的GSS库。目前，除了在包含GSSAPI和SSPI支持的Windows构建中，这一点被忽略了。 在这种情况下，将此设置为gssapi使 libuxsql 使用 GSSAPI 库进行身份验证，而不是默认 SSPI。

service

用于附加参数的服务名。它指定保持附加连接参数的ux_service.conf中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。请参见第 17 章 连接服务文件

target_session_attrs

如果这个参数被设置为read-write，只有默认接受读写事务的连接才是可接受的。在任何成功的连接上将发出查询SHOW transaction_read_only，如果它返回on则连接将被关闭。如果在连接字符串中指定了多个主机，只要连接尝试失败就会尝试剩余的服务器。这个参数的默认值any认为所有连接都可接受。

第 3 章 连接状态函数

这些函数可以被用来询问一个已有数据库连接对象的状态。

下列函数返回一个连接所建立的参数值。这些值在连接的生命期中是固定的。使用多主机连接字符串，且使用同一个UXconn对象建立新连接，UXSQLhost、UXSQLport以及UXSQLpass可能会改变。其他值在UXconn对象中都是固定的。

UXSQLdb

返回该连接的数据库名。

```
char *UXSQLdb(const UXconn *conn);
```

UXSQLuser

返回该连接的用户名。

```
char *UXSQLuser(const UXconn *conn);
```

UXSQLpass

返回该连接的口令。

```
char *UXSQLpass(const UXconn *conn);
```

UXSQLpass将返回连接参数中指定的口令，如果连接参数中没有口令并且能从口令文件中得到口令，则返回得到的口令。在后一种情况中，如果连接参数中指定了多个主机，在连接被建立之前都不能依赖UXSQLpass的结果。连接的状态可以用函数UXSQLstatus检查。

UXSQLhost

返回活跃连接的服务器主机名。可能是主机名、IP 地址或者一个目录路径（如果通过 Unix 套接字连接，路径的情况很容易区分，因为路径总是一个绝对路径，以/开始）。

```
char *UXSQLhost(const UXconn *conn);
```

如果连接参数同时指定了host和hostaddr，则UXSQLhost将返回host信息。如果仅指定了hostaddr，则返回hostaddr。如果在连接参数中指定了多个主机，UXSQLhost返回实际连接到的主机。

如果*conn*参数是NULL，则UXSQLhost返回NULL。否则，如果有一个错误产生主机信息（或许是连接没有被完全建立或其他错误），它会返回一个空字符串。

如果在连接参数中指定了多个主机，则在连接建立之前都不能依赖于UXSQLhost的结果。连接的状态可以用函数UXSQLstatus检查。

UXSQLhostaddr

返回活跃连接的服务器 IP 地址。可以是主机名解析出的地址，也可以是通过hostaddr参数提供的 IP 地址。

char *UXSQLhostaddr(const UXconn *conn);

如果*conn* 参数为 NULL则 UXSQLhostaddr 返回 NULL 。 否则，如果生成主机信息时出现错误（如果连接尚未完全建立或出现错误），则返回一个空字符串。

UXSQLport

返回活跃连接的端口。

char *UXSQLport(const UXconn *conn);

如果在连接参数中指定了多个端口，UXSQLport返回实际连接到的端口。

如果*conn*参数是NULL，则UXSQLport返回NULL。如果有一个错误产生端口信息（或许是连接没有被完全建立或者有什么错误），它会返回一个空字符串。

如果在连接参数中指定了多个端口，则在连接建立之前都不能依赖于UXSQLport的结果。连接的状态可以用函数UXSQLstatus检查。

UXSQLoptions

返回被传递给连接请求的命令行选项。

char *UXSQLoptions(const UXconn *conn);

下列函数返回随着在UXconn对象上执行操作而改变的状态数据。

UXSQLstatus

返回该连接的状态。

ConnStatusType UXSQLstatus(const UXconn *conn);

该状态可以是一系列值之一。不过，其中只有两个在一个异步连接过程之外可见：CONNECTION_OK和CONNECTION_BAD。一个到数据库的完好连接的状态为CONNECTION_OK。一个失败的连接尝试则由状态CONNECTION_BAD表示。通常，一个 OK 状态将一直保持到UXSQLfinish，但是一次通信失败可能导致该状态过早地改变为CONNECTION_BAD。在那种情况下，该应用可以通过调用UXSQLreset尝试恢复。

关于其他可能会被返回的状态代码，请见UXSQLconnectStartParams、UXSQLconnectStart和UXSQLconnectPoll的条目。

UXSQLtransactionStatus

返回服务器的当前事务内状态。

UXTransactionStatusType UXSQLtransactionStatus(const UXconn *conn);

该状态可能是UXSQLTRANS_IDLE（当前空闲）、UXSQLTRANS_ACTIVE（一个命令运行中）、UXSQLTRANS_INTRANS（空闲，处于一个合法的事务块中）或者UXSQLTRANS_INERROR（空闲，处于一个失败的事务块中）。如果该连接损坏，将会

报告UXSQLTRANS_UNKNOWN。只有当一个查询已经被发送给服务器并且还没有完成时，才会报告UXSQLTRANS_ACTIVE。

UXSQLparameterStatus

查找服务器的一个当前参数设置。

const char *UXSQLparameterStatus(const UXconn *conn, const char *paramName);

某一参数值会被服务器在连接开始或值改变时自动报告。UXSQLparameterStatus可以被用来询问这些设置。它为已知的参数返回当前值，为未知的参数返回NULL。

会被报告的参数包括 server_version、 server_encoding、 client_encoding、 application_name、 is_superuser、 session_authorization、 DateStyle、 IntervalStyle、 TimeZone、 integer_datetimes以及 standard_conforming_strings。注意 server_version、 server_encoding以及 integer_datetimes在启动之后无法改变。

3.0 协议之前的服务器不报告参数设置，但是libuxsql包括获得server_version以及client_encoding值的逻辑。推荐使用UXSQLparameterStatus而不是ad hoc代码来确定这些值（不过注意在一个 3.0 之前的连接上，连接开始后通过SET改变client_encoding不会被UXSQLparameterStatus反映）。对于server_version（另见UXSQLserverVersion），它以一种数字形式返回信息，这样更容易进行比较。

如果没有为standard_conforming_strings报告值，应用假设它是off，也就是说反斜线会被视为字符串中的转义。这个参数的存在可以被作为转义字符串语法（E'...'）被接受的指示。

尽管返回的指针被声明成const，事实上它指向与UXconn结构相关的可变存储。指针并非在所有查询中都是有效的。

UXSQLprotocolVersion

询问所使用的 前端/后端协议。

int UXSQLprotocolVersion(const UXconn *conn);

应用可能希望用这个函数来确定某些特性是否被支持。当前，可能值是 2（2.0 协议）、3（3.0 协议）或零（连接损坏）。协议版本在连接启动完成后将不会改变，但是理论上在连接重置期间是可以改变的。

UXSQLserverVersion

返回一个表示服务器版本的整数。

int UXSQLserverVersion(const UXconn *conn);

应用使用这个函数来判断它们连接到的数据库服务器的版本。结果通过将服务器的主版本号乘以10000再加上次版本号形成。

UXSQLerrorMessage

返回由连接上的操作最近产生的错误消息。

char *UXSQLerrorMessage(const UXconn *conn);

几乎所有的libuxsql在失败时都会为UXSQLerrorMessage设置一个消息。注意按照libuxsql习惯，一个非空UXSQLerrorMessage结果由多行构成，并且将包括一个尾部新行。调用者不应该直接释放结果。当相关的UXconn句柄被传递给UXSQLfinish时，它将被释放。
在UXconn结构上的多个操作之间，结果字符串不会保持不变。

UXSQLsocket

获得到服务器连接套接字的文件描述符号。一个合法的描述符将会大于等于零。结果为 -1 表示当前没有打开服务器连接（在普通操作期间不会改变，但是在连接设置或重置期间可能改变）。

int UXSQLsocket(const UXconn *conn);

UXSQLbackendPID

返回处理这个连接的后端进程的进程ID（PID）。

int UXSQLbackendPID(const UXconn *conn);

后端PID有助于调试目的并且可用于与NOTIFY消息（它包括发出提示的后端进程的PID）进行比较。注意PID属于一个在数据库服务器主机上执行的进程，而不是本地主机进程！

UXSQLconnectionNeedsPassword

如果连接认证方法要求一个口令但没有可用的口令，返回真（1）。否则返回假（0）。

int UXSQLconnectionNeedsPassword(const UXconn *conn);

这个函数可以在连接尝试失败后被应用于决定是否向用户提示要求一个口令。

UXSQLconnectionUsedPassword

如果连接认证方法使用一个口令，返回真（1）。否则返回假（0）。

int UXSQLconnectionUsedPassword(const UXconn *conn);

这个函数能在一次连接尝试失败或成功后用于检测该服务器是否要求一个口令。

下面的函数返回与 SSL 有关的信息。这些信息在连接建立后通常不会改变。

UXSQLsslInUse

如果该连接使用了 SSL 则返回真（1），否则返回假（0）。

int UXSQLsslInUse(const UXconn *conn);

UXSQLsslAttribute

返回关于该连接有关 SSL 的信息。

const char *UXSQLsslAttribute(const UXconn *conn, const char *attribute_name);

可用的属性列表取决于使用的 SSL 库以及连接类型。如果一个属性不可用， 会返回 NULL。

下列属性通常是可用的。

library

正在使用的 SSL 实现的名称（当前只实现了 "OpenSSL"）。

protocol

正在使用的 SSL/TLS 版本。常见的值有"TLSv1"、 "TLSv1.1"以及"TLSv1.2"，但是如果一种实现使用了其他协议，可能返回其他字符串。

key_bits

加密算法所使用的密钥位数。

cipher

所使用的加密套件的简短名称，例如 "DHE-RSA-DES-CBC3-SHA"。这些名称与每一种 SSL 实现相关。

compression

如果正在使用 SSL 压缩，则返回压缩算法的名称。如果使用了压缩 但是算法未知则返回 ″on″ 。如果没有使用压缩，则返回 ″off″。

UXSQLsslAttributeNames

返回一个可用的 SSL 名称的数组。该数组以一个 NULL 指针终结。

const char * const * UXSQLsslAttributeNames(const UXconn *conn);

UXSQLsslStruct

返回一个描述该连接的 SSL 实现相关对象的指针。

void *UXSQLsslStruct(const UXconn *conn, const char *struct_name);

可用的结构，这些结构取决于使用的 SSL 实现。对于 OpenSSL，有一个结构可用， 其名称为 ″OpenSSL″。它返回一个指向该 OpenSSL SSL 结构 的指针。要使用这个函数，可以用下面的代码。

```
#include <libuxsql-fe.h>
#include <openssl/ssl.h>

...

  SSL *ssl;

  dbconn = UXSQLconnectdb(...);
  ...
```

```
ssl = UXSQLsslStruct(dbconn, "OpenSSL");
if (ssl)
{
  /* 使用 OpenSSL 函数来访问 ssl */
}
```

这个结构可以被用来验证加密级别、检查服务器证书等等。有关OpenSSL的相关文档可以参照其官网信息。

UXSQLgetssl

返回连接中使用的 SSL 结构，如果没有使用 SSL 则返回空。

```
void *UXSQLgetssl(const UXconn *conn);
```

这个函数等效于UXSQLsslStruct(conn, "OpenSSL")。在新的应用中不应该使用它，因为被返回的结构是 OpenSSL 专用的，如果使用了另一种 SSL 实现就不再可用。要检查一个连接是否使用 SSL，应该调用UXSQLsslInUse，而要得到有关连接的更多细节应该使用UXSQLsslAttribute。

第 4 章 命令执行函数

一旦到一个数据库服务器的连接被成功建立，这里描述的函数可以被用来执行 SQL 查询和命令。

4.1. 主要函数

UXSQLexec

提交一个命令给服务器并且等待结果。

```
UXresult *UXSQLexec(UXconn *conn, const char *command);
```

返回一个UXresult指针或者可能是一个空指针。除了内存不足的情况或者由于严重错误无法将命令发送给服务器之外，一般都会返回一个非空指针。UXSQLresultStatus函数应当被调用来检查返回值是否代表错误（包括空指针的值，它会返回UXRES_FATAL_ERROR）。用UXSQLerrorMessage可得到关于那些错误的详细信息。

命令字符串可以包括多个 SQL 命令（用分号分隔）。在一次UXSQLexec调用中被发送的多个查询会在一个事务中处理，除非其中有显式的BEGIN/COMMIT命令将该查询字符串划分成多个事务。但是注意，返回的UXresult结构只描述该字符串中被执行的最后一个命令的结果。如果一个命令失败，该字符串的处理会在它那里停止并且返回的UXresult会描述错误情况。

UXSQLexecParams

提交一个命令给服务器并且等待结果，它可以在 SQL 命令文本之外独立地传递参数。

```
UXresult *UXSQLexecParams(UXconn *conn,
                const char *command,
                int nParams,
                const Oid *paramTypes,
                const char * const *paramValues,
                const int *paramLengths,
                const int *paramFormats,
                int resultFormat);
```

UXSQLexecParams与UXSQLexec相似，但是提供了额外的功能：参数值可以与命令字符串分开指定，并且可以以文本或二进制格式请求查询结果。UXSQLexecParams只在 3.0 协议及其后的连接中被支持，当使用 2.0 协议时它会失败。

该函数的参数如下。

conn

要在其中发送命令的连接对象。

command

要执行的 SQL 命令字符串。如果使用了参数，它们在该命令字符串中被引用为$1、$2等。

nParams

提供的参数数量。它是数组*paramTypes[]*、*paramValues[]*、*paramLengths[]*和*paramFormats[]*的长度（当*nParams*为零时，数组指针可以是NULL）。

paramTypes[]

通过 OID 指定要赋予给参数符号的数据类型。如果*paramTypes*为NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。

paramValues[]

指定参数的实际值。这个数组中的一个空指针表示对应的参数为空，否则该指针指向一个以零终止的文本字符串（用于文本格式）或者以服务器所期待格式的二进制数据（用于二进制格式）。

paramLengths[]

指定二进制格式参数的实际数据长度。它对空参数和文本格式参数被忽略。当没有二进制参数时，该数组指针可以为空。

paramFormats[]

指定参数是否为文本（在参数相应的数组项中放一个零）或二进制（在参数相应的数组项中放一个一）。如果该数组指针为空，那么所有参数都会被假定为文本串。

以二进制格式传递的值要求后端所期待的内部表示形式的知识。例如，整数必须以网络字节序被传递。传递numeric值要求关于服务器存储格式的知识，正如src/backend/utils/adt/numeric.c::numeric_send()以及src/backend/utils/adt/numeric.c::numeric_recv()中所实现的。

resultFormat

指定零来得到文本格式的结果，或者指定一来得到二进制格式的结果（目前没有规定要求以不同格式得到不同的结果列，尽管在底层协议中这是可以实现的）。

UXSQLexecParams相对于UXSQLexec的主要优点是参数值可以从命令串中分离，因此避免了冗长的书写、容易发生错误的引用以及转义。

和UXSQLexec不同，UXSQLexecParams至多允许在给定串中出现一个 SQL 命令（其中可以有分号，但是不能有超过一个非空命令）。这是底层协议的一个限制，但是有助于抵抗 SQL 注入攻击。

提示

通过 OID 指定参数类型很繁琐，特别是如果不愿意将特定的 OID 值硬编码到程序中时。不过，即使服务器本身也无法确定参数的类型，可以避免这样做，或者选择一种与想要的不同的类型。在 SQL 命令文本中，附加一个显式造型给参数符号来表示发送什么样的数据类型。示例如下。

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

> 这强制参数$1被当作bigint，而默认情况下它将被赋予与x相同的类型。当以二进制格式发送参数值时，强烈推荐以这种方式或通过指定一个数字类型的 OID 来强制参数类型决定。因为二进制格式比文本格式具有更少的冗余，并且因此服务器将不会有更多机会为检测一个类型匹配错误。

UXSQLprepare

提交一个请求用给定参数创建一个预备语句并且等待完成。

```
UXresult *UXSQLprepare(UXconn *conn,
                const char *stmtName,
                const char *query,
                int nParams,
                const Oid *paramTypes);
```

UXSQLprepare创建一个后面会由UXSQLexecPrepared执行的预备语句。这个特性允许命令被反复执行而无需每次都进行解析和规划。UXSQLprepare只在协议 3.0 及之后的连接中被支持，当使用协议 2.0 时它将失败。

该函数从*query*串创建一个名为*stmtName*的预备语句，该串必须包含一个单一 SQL 命令。*stmtName*可以是""来创建一个未命名语句，在这种情况下任何已存在未命名语句将被自动替换。如果语句名称已经在当前会话中被定义，则是一种错误。如果使用了任何参数，它们在查询中以$1、$2等引用。*nParams*是参数的个数，其类型在数组*paramTypes[]*中被预先指定（当*nParams*为零时，该数组指针可以是NULL）。*paramTypes[]*通过 OID 指定要赋予给参数符号的数据类型。如果*paramTypes*是NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。还有，查询能够使用编号高于*nParams*的参数符号，它们的数据类型也会被自动推测（找出推测出的数据类型的方法见UXSQLdescribePrepared）。

正如UXSQLexec一样，结果通常是一个UXresult对象，其内容代表服务器端成功或失败。一个空结果表示内存不足或者根本无法发送命令。关于错误的更多信息请见UXSQLerrorMessage。

用于UXSQLexecPrepared的预备语句也能通过执行 SQL PREPARE语句来创建。还有，尽管没有libuxsql函数来删除一个预备语句，SQL DEALLOCATE语句可被用于此目的。

UXSQLexecPrepared

发送一个请求来用给定参数执行一个预备语句，并且等待结果。

```
UXresult *UXSQLexecPrepared(UXconn *conn,
                const char *stmtName,
                int nParams,
                const char * const *paramValues,
                const int *paramLengths,
                const int *paramFormats,
                int resultFormat);
```

UXSQLexecPrepared像UXSQLexecParams，但是要被执行的命令是用之前准备的语句的名字指定，而不是指定一个查询串。这个特性允许将被重复使用的命令只被解析和规划一次，而不是在每次被执行时都被解析和规划。这个语句必须之前在当前会话中已经被准备好。UXSQLexecPrepared仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

参数和UXSQLexecParams相同，除了给定的是一个预备语句的名称而不是一个查询语句，以及不存在*paramTypes[]*参数（因为预备语句的参数类型已经在它被创建时决定好了）。

UXSQLdescribePrepared

提交一个请求来获得有关指定预备语句的信息，并且等待完成。

UXresult *UXSQLdescribePrepared(UXconn *conn, const char *stmtName);

UXSQLdescribePrepared允许一个应用获得有关一个之前预备好的语句的信息。UXSQLdescribePrepared仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

*stmtName*可以用""或者NULL来引用未命名语句，否则它必须是一个现有预备语句的名字。如果成功，一个UXresult以及状态UXRES_COMMAND_OK会被返回。函数UXSQLnparams和UXSQLparamtype可以被应用到这个UXresult来得到关于该预备语句参数的额外信息，而函数UXSQLnfields、UXSQLfname、UXSQLftype等提供该语句结果列（如果有）的信息。

UXSQLdescribePortal

提交一个请求来得到有关指定入口的信息，并且等待完成。

UXresult *UXSQLdescribePortal(UXconn *conn, const char *portalName);

UXSQLdescribePortal允许一个应用获得有关一个之前被创建的入口的信息（libuxsql不提供对入口任何直接的访问，但是可以使用这个函数来观察一个通过**DECLARE CURSOR** SQL 命令创建的游标的属性）。UXSQLdescribePortal仅被协议 3.0 及之后的连接支持，当使用协议 2.0 时它会失败。

*portalName*可以用""或者NULL来引用未命名入口，否则它必须是一个现有入口的名字。如果成功，一个UXresult和状态UXRES_COMMAND_OK会被返回。函数UXSQLnfields、UXSQLfname、UXSQLftype等可以被应用到UXresult来获得有关该入口结果列（如果有）的信息。

UXresult结构封装了由服务器返回的结果。libuxsql应用程序员应该小心地维护UXresult的抽象。使用下面的存储器函数来得到UXresult的内容。避免直接引用UXresult结构的域，因为它们可能在未来更改。

UXSQLresultStatus

返回该命令的结果状态。

ExecStatusType UXSQLresultStatus(const UXresult *res);

UXSQLresultStatus能返回下列值之一。

UXRES_EMPTY_QUERY

发送给服务器的字符串为空。

UXRES_COMMAND_OK

一个不返回数据的命令成功完成。

UXRES_TUPLES_OK

一个返回数据的命令（例如SELECT或者SHOW）成功完成。

UXRES_COPY_OUT

从服务器复制出数据的传输开始。

UXRES_COPY_IN

复制数据到服务器的传输开始。

UXRES_BAD_RESPONSE

无法理解服务器的响应。

UXRES_NONFATAL_ERROR

发生了一次非致命错误（一个提示或警告）。

UXRES_FATAL_ERROR

发生了一次致命错误。

UXRES_COPY_BOTH

向服务器复制数据/从服务器复制数据的传输开始。这个特性当前只被用于流复制，因此这个状态应该不会在普通应用中出现。

UXRES_SINGLE_TUPLE

UXresult包含来自于当前命令的一个单一结果元组。这个状态只在查询选择了单一行模式时发生（请参见第 6 章 单行模式检索查询结果。

如果结果状态是UXRES_TUPLES_OK或者UXRES_SINGLE_TUPLE，那么下面所描述的函数能被用来检索该查询所返回的行。注意，一个恰好检索零行的SELECT命令仍然会显示UXRES_TUPLES_OK。UXRES_COMMAND_OK用于从不返回行的命令（不带RETURNING子句的INSERT或者UPDATE等）。一个UXRES_EMPTY_QUERY可能表示客户端软件中的一个缺陷。

一个状态为UXRES_NONFATAL_ERROR的结果将不会被UXSQLexec或者其他查询执行函数直接返回，这类结果将被传递给提示处理器（请参见第 13 章 通知处理。

UXSQLresStatus

将UXSQLresultStatus返回的枚举转换成描述状态编码的字符串常量。调用者不应该释放结果。

```
char *UXSQLresStatus(ExecStatusType status);
```

UXSQLresultErrorMessage

返回与该命令相关的错误消息，如果有错误则会返回一个空字符串。

char *UXSQLresultErrorMessage(const UXresult *res);

如果有一个错误，被返回的字符串将包含一个收尾的新行。调用者不应该直接释放结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

紧跟着一个UXSQLexec或UXSQLgetResult调用，UXSQLerrorMessage（在连接上）将返回与UXSQLresultErrorMessage相同的字符串（在结果上）。不过，一个UXresult将保持它的错误消息直到被销毁，而连接的错误消息将在后续操作被执行时被更改。当想要知道与一个特定UXresult相关的状态，使用UXSQLresultErrorMessage。而当想要知道连接上最后一个操作的状态，使用UXSQLerrorMessage。

UXSQLresultVerboseErrorMessage

返回与UXresult对象相关的错误消息的重新格式化的版本。

char *UXSQLresultVerboseErrorMessage(const UXresult *res,
UXVerbosity verbosity,
UXContextVisibility show_context);

在有些情况下，客户端可能希望得到之前报告过的错误的更加详尽的版本。如果在产生给定UXresult的连接上 verbosity 设置有效，UXSQLresultVerboseErrorMessage会通过计算已经被UXSQLresultErrorMessage产生过的消息来满足这种需求。如果UXresult不是一个错误结果，则会报告“UXresult is not an error result”。返回的字符串包括一个新行作为结尾。

和大部分从UXresult中提取数据的其他函数不同，这个函数的结果是一个全新分配的字符串。调用者在不需要这个字符串以后，必须使用UXSQLfreemem()释放它。

如果内存不足，可能会返回 NULL。

UXSQLresultErrorField

返回一个错误报告的一个域。

char *UXSQLresultErrorField(const UXresult *res, int fieldcode);

*fieldcode*是一个错误域标识符，见下列符号。如果UXresult不是一个错误或者警告结果或者不包括指定域，会返回NULL。域通常不包括一个收尾的新行。调用者不应该直接释放结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

下列域代码可用。

UX_DIAG_SEVERITY

严重性。域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、 DEBUG、INFO或LOG（在一个提示消息中）。或者是其中之一的一个本地化翻译。总是存在。

UX_DIAG_SEVERITY_NONLOCALIZED

域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、 DEBUG、INFO或LOG（在一个提示消息中）。这和UX_DIAG_SEVERITY域相同，不过内容不会被本地化。

UX_DIAG_SQLSTATE

用于错误的 SQLSTATE 代码。SQLSTATE 代码标识了已经发生的错误的类型，它可以被前端应用用来执行特定操作（例如错误处理）来响应一个特定数据库错误。这个域无法被本地化，并且总是存在。

UX_DIAG_MESSAGE_PRIMARY

主要的人类可读的错误消息（通常是一行）。总是存在。

UX_DIAG_MESSAGE_DETAIL

细节：一个可选的次级错误消息，它携带了关于问题的等多细节。可能有多行。

UX_DIAG_MESSAGE_HINT

提示：一个关于如何处理该问题的可选建议。它与细节的区别在于它提供了建议（可能不合适）而不是铁的事实。可能有多行。

UX_DIAG_STATEMENT_POSITION

包含一个十进制整数的字符串，它表示一个错误游标位置，该位置是原始语句字符串的索引。第一个字符的索引是 1，位置以字符计算而不是以及字节计算。

UX_DIAG_INTERNAL_POSITION

这被定义为与UX_DIAG_STATEMENT_POSITION域相同，但是它被用在游标位置引用一个内部产生的命令而不是客户端提交的命令时。当这个域出现时，UX_DIAG_INTERNAL_QUERY域将总是出现。

UX_DIAG_INTERNAL_QUERY

一个失败的内部产生的命令的文本。例如，这可能是由一个 PL/uxSQL 函数发出的 SQL 查询。

UX_DIAG_CONTEXT

指示错误发生的环境。当前这包括活动过程语言函数的调用栈追踪以及内部生成的查询。追踪是每行一项，最近的排在最前面。

UX_DIAG_SCHEMA_NAME

如果错误与某个特定的数据库对象相关，这里是包含该对象的模式名（如果有）。

UX_DIAG_TABLE_NAME

如果错误与某个特定表相关，这里是该表的名字（该表的模式参考模式名域）。

UX_DIAG_COLUMN_NAME

如果错误与一个特定表列相关，这里是该表列的名字（参考模式和表名域来标识该表）。

UX_DIAG_DATATYPE_NAME

如果错误与一个特定数据类型相关，这里是该数据了行的名字（该数据类型的模式名参考模式名域）。

UX_DIAG_CONSTRAINT_NAME

如果错误与一个特定约束相关，这里是该约束的名字。相关的表或域参考上面列出的域（为了这个目的，索引也被视作约束，即使它们不是用约束语法创建的）。

UX_DIAG_SOURCE_FILE

报告错误的源代码所在的文件名。

UX_DIAG_SOURCE_LINE

报告错误的源代码行号。

UX_DIAG_SOURCE_FUNCTION

报告错误的源代码函数的名字。

> **注意**
>
> 用于模式名、表名、列名、数据类型名和约束名的域只提供给有限的错误类型。不要假定任何这些域的存在保证另一个域的存在。核心错误源会遵守上面提到的内在联系，但是用户定义的函数可能以其他方式使用这些域。同样地，不要假定这些域代表当前数据库中同类的对象。

客户端负责格式化显示信息来迎合它的需要，特别是根据需要打断长的行。出现在错误消息域中的新行字符应该被当作分段而不是换行。

libuxsql内部产生的错误将有严重和主要消息，但是通常没有其他域。3.0 协议之前的服务器返回的错误将包括严重和主要消息，并且有时候还有细节消息，但是没有其他域。

注意错误与只从UXresult对象中有效，对UXconn对象无效。没有UXSQLerrorField函数。

UXSQLclear

Frees the storage associated with a 释放与一个UXresult相关的存储。每一个命令结果不再需要时应该用UXSQLclear释放。

void UXSQLclear(UXresult *res);

可以按照需要保留UXresult对象，当发出一个新命令时它也不会消失，甚至关闭连接时也不会消失。要去掉它，必须调用UXSQLclear。没有这样做将会导致在应用中的内存泄露。

4.2. 检索查询结果信息

这些函数被用来从一个代表成功查询结果（也就是状态为UXRES_TUPLES_OK或者UXRES_SINGLE_TUPLE）的UXresult对象中抽取信息。它们也可以被用来从一个成功的 Describe 操作中抽取信息：一个 Describe 的结果具有和该查询被实际执行所提供的完全相同的列信息，但是它没有行。对于其他状态值的对象，这些函数会认为结果具有零行和零列。

UXSQLntuples

返回查询结果中的行（元组）数（注意，UXresult对象被限制为不超过INT_MAX行，因此一个int结果就足够了）。

int UXSQLntuples(const UXresult *res);

UXSQLnfields

返回查询结果中每一行的列（域）数。

int UXSQLnfields(const UXresult *res);

UXSQLfname

返回与给定列号相关联的列名。列号从 0 开始。调用者不应该直接释放该结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

char *UXSQLfname(const UXresult *res,
int column_number);

如果列号超出范围，将返回NULL。

UXSQLfnumber

返回与给定列名相关联的列号。

int UXSQLfnumber(const UXresult *res,
const char *column_name);

如果给定的名字不匹配任何列，将返回 -1。

给定的名称被视作一个 SQL 命令中的一个标识符，也就是说，除非被双引号引用，它是小写形式的。例如，给定一个 SQL 命令。

SELECT 1 AS FOO, 2 AS "BAR";

将得到结果。

UXSQLfname(res, 0) *foo*
UXSQLfname(res, 1) *BAR*
UXSQLfnumber(res, "FOO") *0*
UXSQLfnumber(res, "foo") *0*
UXSQLfnumber(res, "BAR") *-1*
UXSQLfnumber(res, "\"BAR\"") *1*

UXSQLftable

返回给定列从中取出的表的 OID。列号从 0 开始。

Oid UXSQLftable(const UXresult *res,
int column_number);

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回InvalidOid。可以查询系统表ux_class来确定究竟是哪个表被引用。

当包括libuxsql头文件，类型oid以及常数InvalidOid将被定义。它们将都是某种整数类型。

UXSQLftablecol

返回构成指定查询结果列的列（在其表中）的列号。查询结果列号从 0 开始，但是表列具有非零编号。

```
int UXSQLftablecol(const UXresult *res,
           int column_number);
```

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回零。

UXSQLfformat

返回指示给定列格式的格式编码。列号从 0 开始。

```
int UXSQLfformat(const UXresult *res,
          int column_number);
```

格式代码零指示文本数据表示，而格式代码一表示二进制表示（其他代码被保留用于未来的定义）。

UXSQLftype

返回与给定列号相关联的数据类型。被返回的整数是该类型的内部 OID 号。列号从 0 开始。

```
Oid UXSQLftype(const UXresult *res,
        int column_number);
```

可以查询系统表ux_type来得到多个数据类型的名字和属性。内建数据类型的OID被定义在源代码树中的文件src/include/catalog/ux_type_d.h中。

UXSQLfmod

返回与给定列号相关联的列的修饰符类型。列号从 0 开始。

```
int UXSQLfmod(const UXresult *res,
       int column_number);
```

修饰符值的解释是与类型相关的，它们通常指示精度或尺寸限制。值 -1 被用来指示“没有信息可用”。大部分的数据类型不适用修饰符，在那种情况中值总是 -1。

UXSQLfsize

返回与给定列号相关的列的尺寸（以字节计）。列号从 0 开始。

```
int UXSQLfsize(const UXresult *res,
       int column_number);
```

UXSQLfsize返回在一个数据库行中为这个列分配的空间，换句话说是服务器对该数据类型的内部表示的尺寸（因此，它对客户端并不是真地非常有用）。一个负值指示该数据类型是变长的。

UXSQLbinaryTuples

如果UXresult包含二进制数据，返回 1。如果包含的是文本数据，返回 0。

```
int UXSQLbinaryTuples(const UXresult *res);
```

这个函数已经被废弃（除了与COPY一起使用），因为一个单一UXresult可以在某些列中包含文本数据而且在另一些列中包含二进制数据。UXSQLfformat要更好。只有结果的所有列是二进制（格式 1）时UXSQLbinaryTuples才返回 1。

UXSQLgetvalue

返回一个UXresult的一行的单一域值。行和列号从 0 开始。调用者不应该直接释放该结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

```
char *UXSQLgetvalue(const UXresult *res,
                    int row_number,
                    int column_number);
```

对于文本格式的数据，UXSQLgetvalue返回的值是该域值的一种空值结束的字符串表示。对于二进制格式的数据，该值是由该数据类型的typsend和typreceive函数决定的二进制表示（在这种情况下该值实际上也跟随着一个零字节，但是这通常没有用处，因为该值很可能包含嵌入的空）。

如果该域值为空，则返回一个空串。关于区分空值和空字符串值请见UXSQLgetisnull。

UXSQLgetvalue返回的指针指向作为UXresult结构一部分的存储。不应该修改它指向的数据，并且如果要在超过UXresult结构本身的生命期之外使用它，必须显式地把该数据拷贝到其他存储中。

UXSQLgetisnull

测试一个域是否为空值。行号和列号从 0 开始。

```
int UXSQLgetisnull(const UXresult *res,
                   int row_number,
                   int column_number);
```

如果该域是空，这个函数返回 1。如果它包含一个非空值，则返回 0（注意UXSQLgetvalue将为一个空域返回一个空串，不是一个空指针）。

UXSQLgetlength

返回一个域值的真实长度，以字节计。行号和列号从 0 开始。

```
int UXSQLgetlength(const UXresult *res,
                   int row_number,
                   int column_number);
```

这是特定数据值的真实数据长度，也就是UXSQLgetvalue指向的对象的尺寸。对于文本数据格式，这和strlen()相同。对于二进制格式这是基本信息。注意不应该依赖于UXSQLfsize来得到真值的数据长度。

UXSQLnparams

返回一个预备语句的参数数量。

```
int UXSQLnparams(const UXresult *res);
```

只有在查看UXSQLdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

UXSQLparamtype

返回所指示的语句参数的数据类型。参数号从 0 开始。

```
Oid UXSQLparamtype(const UXresult *res, int param_number);
```

只有在查看UXSQLdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

UXSQLprint

将所有的行打印到指定的输出流，以及有选择地将列名打印到指定的输出流。

```
void UXSQLprint(FILE *fout,     /* 输出流 */
          const UXresult *res,
          const UXSQLprintOpt *po);
typedef struct
{
   uxsqlbool  header;     /* 打印输出域标题和行计数 */
   uxsqlbool  align;      /* 填充对齐域 */
   uxsqlbool  standard;   /* 旧的格式 */
   uxsqlbool  html3;      /* 输出 HTML 表格 */
   uxsqlbool  expanded;   /* 扩展表格 */
   uxsqlbool  pager;      /* 如果必要为输出使用页 */
   char    *fieldSep;   /* 域分隔符 */
   char    *tableOpt;   /* 用于 HTML 表格元素的属性 */
   char    *caption;    /* HTML 表格标题 */
   char    **fieldName; /* 替换域名称的空终止数组 */
} UXSQLprintOpt;
```

这个函数以前被uxsql用来打印查询结果，但是现在不是这样了。注意它假定所有的数据都是文本格式。

4.3. 检索其他结果信息

这些函数被用来从UXresult对象中抽取其他信息。

UXSQLcmdStatus

返回来自于产生UXresult的 SQL 命令的命令状态标签。

char *UXSQLcmdStatus(UXresult *res);

通常这就是该命令的名称，但是它可能包括额外数据，例如已被处理的行数。调用者不应该直接释放该结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

UXSQLcmdTuples

返回受该 SQL 命令影响的行数。

char *UXSQLcmdTuples(UXresult *res);

这个函数返回一个字符串，其中包含着产生UXresult的SQL语句影响的行数。这个只能被用于下列情况：执行一个SELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH或COPY语句，或者一个包含INSERT、UPDATE或DELETE语句的预备查询的EXECUTE。如果产生UXresult的命令是其他什么东西，UXSQLcmdTuples会返回一个空串。调用者不应该直接释放该结果。它将在相关的UXresult句柄被传递给UXSQLclear之后被释放。

UXSQLoidValue

如果该SQL命令是一个正好将一行插入到具有 OID 的表的INSERT，或者是一个包含合适INSERT语句的预备查询的EXECUTE，这个函数返回被插入行的 OID。否则，这个函数返回InvalidOid。如果被INSERT语句影响的表不包含 OID，这个函数也将返回InvalidOid。

Oid UXSQLoidValue(const UXresult *res);

4.4. 用于包含在 SQL 命令中的转义串

UXSQLescapeLiteral

char *UXSQLescapeLiteral(UXconn *conn, const char *str, size_t length);

为了让一个串能用在 SQL 命令中，UXSQLescapeLiteral可对它进行转义。当在 SQL 命令中插入一个数据值作为文字常量时，这个函数很有用。一些字符（例如引号和反斜线）必须被转义以防止它们被 SQL 解析器解释成特殊的意思。UXSQLescapeLiteral执行这种操作。

UXSQLescapeLiteral返回一个*str*参数的已被转义版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用UXSQLfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入*length*（如果在*length*字节被处理之前找到一个终止字节，UXSQLescapeLiteral会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被UXDB字符串解析器正确地处理。还会加上一个终止零字节。包括在结果串中的UXDB字符串必须用单引号包围。

发生错误时，UXSQLescapeLiteral返回NULL并且一个合适的消息会被存储在*conn*对象中。

> **提示**
>
> 在处理从一个非可信源接收到的串时，做正确的转义特别重要。否则就会有安全性风险：容易受到“SQL 注入”攻击，其中可能会有预期之外的 SQL 语句会被传输到数据库。

注意，当一个数据值被作为UXSQLexecParams或其兄弟例程中的一个独立参数传递时，没有必要做转义而且做转义也不正确。

UXSQLescapeIdentifier

```
char *UXSQLescapeIdentifier(UXconn *conn, const char *str, size_t length);
```

UXSQLescapeIdentifier转义一个要用作 SQL 标识符的字符串，例如表名、列名或函数名。当一个用户提供的标识符可能包含被 SQL 解析器解释为标识符一部分的特殊字符时，或者当该标识符可能包含大小写形式应该被保留的大写字符时，这个函数很有用。

UXSQLescapeIdentifier返回一个*str*参数的已被转义为 SQL 标识符的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用UXSQLfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入*length*（如果在*length*字节被处理之前找到一个终止字节，UXSQLescapeIdentifier会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被作为一个 SQL 标识符正确地处理。还会加上一个终止零字节。返回串也将被双引号包围。

发生错误时，UXSQLescapeIdentifier返回NULL并且一个合适的消息会被存储在*conn*对象中。

> **提示**
>
> 与字符串一样，要阻止 SQL 注入攻击，当从一个不可信的来源接收到 SQL 标识符时，它们必须被转义。

UXSQLescapeStringConn

```
size_t UXSQLescapeStringConn(UXconn *conn,
                             char *to, const char *from, size_t length,
                             int *error);
```

UXSQLescapeStringConn转义字符串，它很像UXSQLescapeLiteral。与UXSQLescapeLiteral不一样的是，调用者负责提供一个合适尺寸的缓冲区。此外，UXSQLescapeStringConn不产生必须包围UXDB字符串的单引号。它们应该在结果要插入的 SQL 命令中提供。参数*from*指向要被转义的串的第一个字符，并且*length*参数给出了这个串中的字节数。一个终止的零字节不是必须的，并且不应该被计入*length*（如果在*length*字节被处理之前找到一个终止字节，UXSQLescapeStringConn会停止在零，该行为更像strncpy）。 *to*应当指向一个缓冲区，它能够保持至少比*length*值的两倍还要多至少一个字节，否则该行为是未被定义的。如果*to*和*from*串重叠，行为也是未被定义的。

如果*error*参数不是NULL，那么成功时*error被设置为零，错误时设置为非零。当前唯一可能的错误情况涉及源串中非法的多字节编码。错误时仍然会产生输出串，但是可以预期服务器将认为它是畸形的并且拒绝它。在发生错误时，一个合适的消息被存储在*conn*对象中，不管*error*是不是NULL。

UXSQLescapeStringConn返回写到*to*的字节数，不包括终止的零字节。

UXSQLescapeString

UXSQLescapeString是一个更老的被废弃的UXSQLescapeStringConn版本。

size_t UXSQLescapeString (char *to, const char *from, size_t length);

UXSQLescapeStringConn和UXSQLescapeString之间的唯一区别是不需要UXconn或*error*参数。正因为如此，它不能基于连接属性（例如字符编码）调整它的行为并且因此它可能给出错误的结果。还有，它没有方法报告错误情况。

UXSQLescapeString可以在一次只使用一个UXDB连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。在其他环境中它是一个安全性灾难并且应该用UXSQLescapeStringConn来避免。

UXSQLescapeByteaConn

把要用于一个 SQL 命令的二进制数据用类型bytea转义。和UXSQLescapeStringConn一样，只有在将数据直接插入到一个 SQL 命令串时才使用它。

unsigned char *UXSQLescapeByteaConn(UXconn *conn,
const unsigned char *from,
size_t from_length,
size_t *to_length);

当某些字节值被用作一个SQL语句中的bytea文字的一部分时，它们必须被转义。UXSQLescapeByteaConn转义使用十六进制编码或反斜线转义的字节。

*from*参数指向要被转义的串的第一个字节，并且*from_length*参数给出这个二进制串中的字节数（一个终止的零字节是不需要的也是不被计算的）。*to_length*参数指向一个将保持生成的已转义串长度的变量。这个结果串长度包括结果的终止零字节。

UXSQLescapeByteaConn返回一个*from*参数的已被转义为二进制串的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用UXSQLfreemem()释放。返回串中的所有特殊字符都被替换掉，这样它们能被UXDB的字符串解析器以及bytea输入函数正确地处理。还会加上一个终止零字节。不是结果串一部分的UXDB字符串必须被单引号包围。

在发生错误时，将返回一个空指针，并且一个合适的错误消息被存储在*conn*对象中。当前，唯一可能的错误是没有足够的内存用于结果串。

UXSQLescapeBytea

UXSQLescapeBytea是一个更老的被废弃的UXSQLescapeByteaConn版本。

unsigned char *UXSQLescapeBytea(const unsigned char *from,
size_t from_length,
size_t *to_length);

与UXSQLescapeByteaConn的唯一区别是UXSQLescapeBytea不用一个UXconn参数。正因为这样，UXSQLescapeBytea只能在一次只使用一个UXDB连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。如果在有多个数据库连接的程序中使用，它可能给出错误的结果（在那种情况下使用UXSQLescapeByteaConn）。

UXSQLunescapeBytea

将二进制数据的一个字符串表示转换成二进制数据—它是UXSQLescapeBytea的逆向函数。当检索文本格式的bytea数据时，需要这个函数，但检索二进制个事时则不需要它。

unsigned char *UXSQLunescapeBytea(const unsigned char *from, size_t *to_length);

*from*参数指向一个字符串，例如UXSQLgetvalue被应用到一个bytea列上所返回的。UXSQLunescapeBytea把这个串表示转换成它的二进制表示。它返回一个指向用malloc()分配的缓冲区的指针，在错误时返回NULL，并且把缓冲区的尺寸放在*to_length*中。当结果不再需要时，它必须使用UXSQLfreemem释放。

这种转换并不完全是UXSQLescapeBytea的逆函数，因为当从UXSQLgetvalue接收到字符串时，并不能期待它被“转义”。不需要考虑字符串引用，并且因此也不需要一个参数。

第 5 章 异步命令函数

UXSQLexec函数对于在普通的同步应用中提交命令是足够的。不过，它的一些缺点可能对某些用户很重要。

- UXSQLexec会等待命令完成。该应用可能有其他的工作要做（例如维护用户界面），这时它将不希望阻塞等待回应。
- 因为客户端应用的执行在它等待结果时会被挂起，对于应用来说很难决定要不要尝试取消正在进行的命令（除了在信号处理器中完成，没有其他方法）。
- UXSQLexec只能返回一个UXresult结构。如果提交的命令串包含多个SQL命令， 除了最后一个UXresult之外都会被UXSQLexec丢弃。
- UXSQLexec总是收集命令的整个结果，把它缓存在一个单一的UXresult中。虽然这简化了应用的错误处理逻辑，它对于包含很多行的结果并不现实。

不想受到这些限制的应用可以改用构建UXSQLexec的底层函数：UXSQLsendQuery以及UXSQLgetResult。还有 UXSQLsendQueryParams、 UXSQLsendPrepare、UXSQLsendQueryPrepared、 UXSQLsendDescribePrepared以及 UXSQLsendDescribePortal，它们可以与UXSQLgetResult一起使用来分别复制UXSQLexecParams、 UXSQLprepare、UXSQLexecPrepared、 UXSQLdescribePrepared和 UXSQLdescribePortal的功能。

UXSQLsendQuery

向服务器提交一个命令而不等待结果。如果该命令被成功发送则返回 1，否则返回 0（此时，可以用UXSQLerrorMessage获取关于失败的信息）。

```
int UXSQLsendQuery(UXconn *conn, const char *command);
```

在成功调用UXSQLsendQuery后，一次或者多次调用UXSQLgetResult来获取结果。
在UXSQLgetResult返回一个空指针之前，都不能再次调用UXSQLsendQuery，返回的空指针指示该命令已经完成。

UXSQLsendQueryParams

向服务器提交一个命令和单独的参数，而不等待结果。

```
int UXSQLsendQueryParams(UXconn *conn,
                         const char *command,
                         int nParams,
                         const Oid *paramTypes,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

这个函数等效于UXSQLsendQuery，不过查询参数可以独立于查询字符串分开指定。该函数的参数处理和UXSQLexecParams一样。和UXSQLexecParams类似，它不能在 2.0 协议的连接上工作，并且它只允许在查询字符串中有一条命令。

UXSQLsendPrepare

发送一个请求，用给定参数创建一个预备语句，而不等待完成。

```
int UXSQLsendPrepare(UXconn *conn,
                     const char *stmtName,
                     const char *query,
                     int nParams,
                     const Oid *paramTypes);
```

这个函数是UXSQLprepare的异步版本：如果它能发送这个请求，则返回 1；如果不能，则返回 0。在成功调用之后，调用UXSQLgetResult判断服务器是否成功创建了预备语句。这个函数的参数的处理和UXSQLprepare一样。和UXSQLprepare类似，它不能在 2.0 协议的连接上工作。

UXSQLsendQueryPrepared

发送一个请求，用给定参数执行一个预备语句，而不等待结果。

```
int UXSQLsendQueryPrepared(UXconn *conn,
                           const char *stmtName,
                           int nParams,
                           const char * const *paramValues,
                           const int *paramLengths,
                           const int *paramFormats,
                           int resultFormat);
```

这个函数与UXSQLsendQueryParams类似，但是要执行的命令是通过一个之前已经命名的预备语句指定，　　而不是一个给出的查询字符串。该函数的参数处理和UXSQLexecPrepared一样。和UXSQLexecPrepared类似，它不能在 2.0 协议的连接上工作。

UXSQLsendDescribePrepared

发送一个请求，获得指定的预备语句的信息，但不等待完成。

```
int UXSQLsendDescribePrepared(UXconn *conn, const char *stmtName);
```

这个函数是UXSQLdescribePrepared的一个异步版本：如果它能够发送请求，则返回　　1；否则，返回　　0。在一次成功的调用后，调用UXSQLgetResult来得到结果。该函数的参数处理和UXSQLdescribePrepared一样。和UXSQLdescribePrepared类似，它不能在 2.0 协议的连接上工作。

UXSQLsendDescribePortal

提交一个请求，来获得关于指定入口的信息，但不等待完成。

```
int UXSQLsendDescribePortal(UXconn *conn, const char *portalName);
```

这个函数是UXSQLdescribePortal的一个异步版本：如果它能够发送请求，则返回　　1；否则，返回　　0。在一次成功的调用后，调用UXSQLgetResult来得到结果。该函数的参数处理和UXSQLdescribePortal一样。和UXSQLdescribePortal类似，它不能在 2.0 协议的连接上工作。

UXSQLgetResult

等待来自于一个之前的 UXSQLsendQuery、 UXSQLsendQueryParams、 UXSQLsendPrepare、 UXSQLsendQueryPrepared、 UXSQLsendDescribePrepared或UXSQLsendDescribePortal调用的结果，并且返回它。当该命令完成并且没有更多结果时，将返回一个空指针。

UXresult *UXSQLgetResult(UXconn *conn);

UXSQLgetResult必须被反复调用直到它返回一个空指针，空指针表示该命令完成（如果在没有命令活动时被调用，UXSQLgetResult将立即返回一个空指针）。UXSQLgetResult的每个非空结果应该使用之前描述的同一个UXresult访问器处理。不要忘记在处理完之后使用UXSQLclear释放每一个结果对象。注意，当一个命令是活动状态并且UXSQLconsumeInput还没有读取必要的响应数据时， UXSQLgetResult将会阻塞。

> **注意**
>
> 即使当UXSQLresultStatus指出一个致命错误时，UXSQLgetResult也应当被调用直到它返回一个空指针，以允许libuxsql完全处理该错误信息。

使用UXSQLsendQuery和UXSQLgetResult解决了UXSQLexec的一个问题：如果一个命令字符串包含多个SQL命令，可以获得这些命令的个别结果（这样就允许一种简单的重叠处理形式， 客户端可以处理一个命令的结果，而同时服务器可以继续处理同一命令字符串中后面的查询）。

UXSQLsendQuery和UXSQLgetResult还有一个特性是一次从大型结果中检索一行，请参见第 6 章单行模式检索查询结果。

如果只调用UXSQLgetResult（不调用UXSQLsendQuery等）将仍会导致客户端阻塞直到服务器完成下一个SQL命令。下面两个函数的正确使用可以避免这种情况。

UXSQLconsumeInput

用于有可用的来自服务器的输入。

int UXSQLconsumeInput(UXconn *conn);

UXSQLconsumeInput返回 1 表明“没有错误”，而返回 0 表明有某种错误发生（此时可以用UXSQLerrorMessage）。注意该结果并不表明是否真正收集了任何输入数据。在调用UXSQLconsumeInput之后，应用可以检查UXSQLisBusy或UXSQLnotifies查看它们的状态是否改变。

即使应用还不准备处理一个结果或通知，UXSQLconsumeInput也可以被调用。这个函数将读取可用的数 据并且把它保存在一个缓冲区中，从而导致select()的读准备指示消失。因此应用可以使用UXSQLconsumeInput立即清除select()条件，并且在空闲时再检查结果。

UXSQLisBusy

如果一个命令繁忙则返回 1，也就是说UXSQLgetResult会阻塞等待输入。返回 0 表示可以调用UXSQLgetResult而不用担心阻塞。

int UXSQLisBusy(UXconn *conn);

UXSQLisBusy本身不会尝试从服务器读取数据，因此必须先调用UXSQLconsumeInput，否则繁忙状态永远不会结束。

一个使用这些函数的典型应用有一个主循环，在主循环中会使用select()或poll()等待所有它必须响应的情况。其中之一是来自服务器的输入可用，对select()来说意味着UXSQLsocket标识的文件描述符上有可读的数据。当主循环检测到输入准备好时，它将调用UXSQLconsumeInput读取输入。然后它可以调用UXSQLisBusy，如果UXSQLisBusy返回假（0）则接着调用UXSQLgetResult。它还可以调用UXSQLnotifies检测NOTIFY消息（请参见第 9 章 异步通知。

一个使用UXSQLsendQuery/UXSQLgetResult的客户端也可以尝试取消一个正在被服务器处理的命令，请参见第 7 章 取消正在处理的查询但是，不管UXcancel的返回值是什么，应用都必须继续使用UXSQLgetResult进行正常的结果读取序列。一次成功的取消只会导致命令比不取消时更快终止。

通过使用上述函数，可以避免在等待来自数据库服务器的输入时被阻塞。不过，在应用发送输出给服务器时还是可能出现阻塞。这种情况比较少见，但是如果发送非常长的 SQL 命令或者数据值时确实可能发生（在应用通过COPY IN发送数据时最有可能发生）。为了避免这种可能性并且实现完全地非阻塞数据库操作，可以使用下列附加函数。

UXSQLsetnonblocking

把连接的状态设置为非阻塞。

int UXSQLsetnonblocking(UXconn *conn, int arg);

如果*arg*为 1，把连接状态设置为非阻塞；如果*arg*为 0，把连接状态设置为阻塞。成功返回 0，发生错误返回 -1。

在非阻塞状态，调用 UXSQLsendQuery、UXSQLputline、UXSQLputnbytes、UXSQLputCopyData和 UXSQLendcopy将不会阻塞， 但是如果它们需要被再次调用则会返回一个错误。

注意UXSQLexec不会遵循任何非阻塞模式；如果调用UXSQLexec，那么它的行为总是阻塞的。

UXSQLisnonblocking

返回数据库连接的阻塞状态。

int UXSQLisnonblocking(const UXconn *conn);

如果连接被设置为非阻塞状态，返回 1，如果是阻塞状态返回 0。

UXSQLflush

尝试把任何排队的输出数据刷新到服务器。如果成功（或者发送队列为空）返回 0， 如果因某种原因失败则返回 -1，或者如果如果尚未发送队列中的所有数据，则返回 1（这种情况只在连接为非阻塞时候才会发生）。

int UXSQLflush(UXconn *conn);

在一个非阻塞连接上发送任何命令或者数据之后，要调用UXSQLflush。如果它返回 1，就要等待套接字变成可读或可写。如果它变为可写，应再次调用UXSQLflush。如果它变为可读，则应

先调用UXSQLconsumeInput，然后再调用UXSQLflush。一直重复直到UXSQLflush返回 0（有必要检查是否可读并且用UXSQLconsumeInput耗尽输入，因为服务器可能阻塞发送数据的尝试，例如 NOTICE 消息，并且发出读数据命令之前它不会读数据）。一旦UXSQLflush返回 0，应等待套接字变成可读并且接着按照上文所述读取响应。

第 6 章 单行模式检索查询结果

通常，libuxsql会收集一个 SQL 命令的整个结果并且把它作为单个UXresult返回给应用。这对于返回大量行的命令是行不通的。对于这类情况，应用可以使用UXSQLsendQuery和UXSQLgetResult的单行模式。在这种模式中，结果行以一次一行的方式被返回给应用。

要进入到单行模式，在一次成功的UXSQLsendQuery（或者其他兄弟函数）调用后立即调用UXSQLsetSingleRowMode。这种模式选择只对当前正在执行的查询有效。然后反复调用UXSQLgetResult，直到它返回空，如第 5 章 异步命令函数所示。如果该查询返回行，它们会作为单个的UXresult对象返回，它们看起来都像普通的查询结果，只不过其状态代码是UXRES_SINGLE_TUPLE而非UXRES_TUPLES_OK。在最后一行之后或者紧接着该查询返回零行之后，一个状态为UXRES_TUPLES_OK的零行对象会被返回，这就是代表不会有更多行的信号（但是注意仍然有必要继续调用UXSQLgetResult直到它返回空）。所有这些UXresult对象将包含相同的行描述数据（列名、类型等等），这些数据和查询的普通UXresult对象的相同。每一个对象都应该按常规用UXSQLclear释放。

UXSQLsetSingleRowMode

为当前正在执行的查询选择单行模式。

int UXSQLsetSingleRowMode(UXconn *conn);

这个函数只能在调用UXSQLsendQuery或一个其兄弟函数之后立刻调用，并且要在任何连接上的其他操作之前调用，例如UXSQLconsumeInput或UXSQLgetResult。如果在正确的时间被调用，该函数会为当前查询激活单行模式并且返回 1。否则模式会保持不变并且该函数返回0。在任何情况下，当前查询结束之后模式都会恢复到正常。

> **注意**
>
> 在处理一个查询时，服务器可能返回一些行并且接着遇到一个错误导致查询被中断。通常，libuxsql会丢弃掉这样的行并且只报告错误。但是在单行模式中，那些行（错误之前返回的行）已经被返回给应用。因此，应用将看到一些UXRES_SINGLE_TUPLE UXresult对象并且看到一个UXRES_FATAL_ERROR对象。为了得到正确的事务行为，如果查询最终失败，应用必须丢弃或者撤销使用之前处理的行完成的事情。

第 7 章 取消正在处理的查询

客户端应用可以使用本节描述的函数请求取消一个正在被服务器处理的命令。

UXSQLgetCancel

创建一个数据结构，这个数据结构包含取消一个通过特定数据库连接发出的命令所需要的信息。

UXcancel *UXSQLgetCancel(UXconn *conn);

给定一个UXconn连接对象，UXSQLgetCancel创建一个UXcancel对象。如果给定的*conn*为NULL或者一个不合法的连接，它将返回NULL。UXcancel对象是一个透明的结构，它不能直接被应用访问。它只能被传递给UXSQLcancel或UXSQLfreeCancel。

UXSQLfreeCancel

释放一个由UXSQLgetCancel创建的数据结构。

void UXSQLfreeCancel(UXcancel *cancel);

UXSQLfreeCancel释放一个之前由UXSQLgetCancel创建的数据对象。

UXSQLcancel

要求服务器放弃当前命令的处理。

int UXSQLcancel(UXcancel *cancel, char *errbuf, int errbufsize);

如果取消请求成功发送，则返回值为 1，否则为 0。如果不成功，则*errbuf*会被填写一个解释为何不成功的错误消息。*errbuf*必须是一个大小为*errbufsize*的字符数组（推荐大小为 256 字节）。

不过，成功的发送并不保证请求会有任何效果。如果取消有效，那么当前的命令将提前终止并且返回一个错误结果。如果取消失败（也就是说， 因为服务器已经完成命令的处理），那么就根本不会有可见的结果。

如果*errbuf*是信号处理器中的一个局部变量，UXSQLcancel可以从一个信号处理器中安全地调用。在UXSQLcancel有关的范围内，UXcancel都是只读的，因此也可以在一个从操纵UXconn对象的线程中独立出来的线程中调用它。

第 8 章 快速路径接口

UXDB提供一种快速路径接口来向服务器发送简单的函数调用。

> **提示**
>
> 还可以通过创建一个定义该函数调用的预备语句来达到类似或者更强大的功能。然后，用参数和结果的二进制传输执行该语句，从而取代快速函数调用。

函数UXSQLfn请求通过快速路径接口执行服务器函数。

```
UXresult *UXSQLfn(UXconn *conn,
            int fnid,
            int *result_buf,
            int *result_len,
            int result_is_int,
            const UXSQLArgBlock *args,
            int nargs);

typedef struct
{
   int len;
   int isint;
   union
   {
      int *ptr;
      int integer;
   } u;
} UXSQLArgBlock;
```

*fnid*参数是要被执行的函数的 OID。*args*和*nargs*定义了要传递给函数的参数；它们必须匹配已声明的函数参数列表。当一个参数结构的*isint*域为真时，*u.integer*值被以指定长度（必须是 1、2 或者 4 字节）整数的形式发送给服务器；这时候会发生恰当的字节交换。当*isint*为假时，**u.ptr*中指定数量的字节将不做任何处理被发送出去；这些数据必须是服务器 预期的用于该函数参数数据类型的二进制传输的格式（由于历史原因*u.ptr*被声明为类型int *，其实把它考虑成void *会更好）。*result_buf*是放置该函数返回值的缓冲区。调用者必须已经分配了足够的空间来存储返回值（这里没有检查！）。实际的结果长度将被放在*result_len*指向的整数中返回。如果预期结果是 2 或 4 字节整数，把*result_is_int*设为 1；否则设为 0。把*result_is_int*设为 1 导致libuxsql在必要时对值进行交换字节，这样它就作为对客户端机器正确的int值被传输，注意对任一种允许的结果大小都会传递一个 4 字节到**result_buf*。当*result_is_int*是 0 时，服务器发送的二进制格式字节将不做修改直接返回（在这种情况下，把*result_buf*考虑为类型void *更好）。

UXSQLfn总是返回一个有效的UXresult指针。在使用结果之前应该检查结果状态。当结果不再使用后，调用者有义务使用UXSQLclear释放UXresult。

注意我们没办法处理空参数、空结果，也没办法在使用这个接口时处理集值结果。

第 9 章 异步通知

UXDB通过LISTEN和NOTIFY命令提供了异步通知。客户端会话可以用LISTEN命令在特定的通知频道中监听通知（也可以用UNLISTEN命令停止监听）。当任何会话执行带有特定频道名的NOTIFY命令时，所有正在监听该频道的会话会被异步通知。可以传递一个“载荷”字符串来与监听者沟通附加的数据。

libuxsql应用把LISTEN、UNLISTEN和NOTIFY命令作为常规 SQL 命令提交。 随后通过调用UXSQLnotifies来检测NOTIFY消息的到达。

函数UXSQLnotifies从来自服务器的未处理通知消息列表中返回下一个通知。如果没有待处理的信息则返回一个空指针。一旦UXSQLnotifies返回一个通知，该通知会被认为已处理并且将被从通知列表中删除。

```
UXnotify *UXSQLnotifies(UXconn *conn);

typedef struct uxNotify
{
   char *relname;           /* notification channel name */
   int  be_pid;            /* process ID of notifying server process */
   char *extra;            /* notification payload string */
} UXnotify;
```

在处理完UXSQLnotifies返回的UXnotify对象后，别忘了用UXSQLfreemem把它释放。释放UXnotify指针就足够了；*relname*和*extra*域并不代表独立分配的内存（这些域的名称是历史性的，尤其是频道名称与关系名称没有联系）。

例 22.2 “libuxsql示例程序 2”给出了一个程序示例展示异步通知的使用。

UXSQLnotifies实际上并不从服务器读取数据；它只是返回被另一个libuxsql函数之前读取的消息。

没有可用命令提交时，更好的检查NOTIFY消息的方法是调用UXSQLconsumeInput，然后检查UXSQLnotifies。可以使用select()来等待服务器数据到达，这样在空闲时可以不浪费CPU能力（参考UXSQLsocket来获得用于select()的文件描述符）。注意不管是用UXSQLsendQuery/UXSQLgetResult提交命令还是简单地使用UXSQLexec，这种方法都能正常工作。不过，注意在每次UXSQLgetResult或UXSQLexec之后检查UXSQLnotifies，查看在命令的处理过程中是否有通知到达。

第 10 章 COPY命令相关函数

UXDB的COPY命令有用于libuxsql的对网络连接读出或者写入的选项。这一节描述的函数允许应用通过提供或复制的数据来充分利用这个功能。

处理过程如下。 首先，应用通过UXSQLexec或者一个等效的函数发出 SQL COPY命令。对这个命令的响应（如果命令无误）是一个状态代码，是UXRES_COPY_OUT或者UXRES_COPY_IN（取决于指定的拷贝方向）的UXresult对象。 接着，应用使用本节介绍的函数接收或传送数据行。数据传输结束之后，另外一个UXresult对象被返回以表明传输的成功或者失败。它的状态将是：UXRES_COMMAND_OK表示成功，UXRES_FATAL_ERROR表示发生了一些问题。此时通过UXSQLexec发出进一步的 SQL 命令（在COPY操作的处理过程中，不能用同一个连接执行其它 SQL 命令）。

如果一个COPY命令是通过UXSQLexec在一个可能包含额外命令的字符串中发出的，那么应用在完成COPY序列之后必须继续用UXSQLgetResult取得结果。只有在UXSQLgetResult返回NULL时，才能确信UXSQLexec的命令字符串已经处理完毕， 并且可以安全地发出更多命令。

这一节的函数应该只在从UXSQLexec或UXSQLgetResult获得了UXRES_COPY_OUT或UXRES_COPY_IN结果状态的后执行。

一个承载了这些状态值之一的UXresult对象携带了正在开始的COPY操作的一些额外数据。这些额外的数据可以用于那些与带查询结果的连接一起使用的函数。

UXSQLnfields

返回要拷贝的列（域）的个数。

UXSQLbinaryTuples

0 表示整体拷贝格式都是文本（行用新行分隔，列用分隔字符分隔等等）。1 表示整体拷贝格式都是二进制。

UXSQLfformat

返回与拷贝操的每列相关的格式代码（0 是文本，1 是二进制）。当整体拷贝格式是文本时，那么每列的格式代码将总是零，但是二进制格式可以同时支持文本和二进制列（不过，就目前的COPY实现而言，二进制拷贝中只会出现二进制列；所以目前每列的格式总是匹配总体格式）。

> **注意**
>
> 这些额外的数据值只在使用协议 3.0 时可用。在使用协议 2.0 时，所有这些函数都返回 0。

10.1. 发送COPY数据的函数

这些函数用于在COPY FROM STDIN期间发送数据。连接不是COPY_IN状态时调用它们将会失败。

UXSQLputCopyData

在COPY_IN状态中向服务器发送数据。

```
int UXSQLputCopyData(UXconn *conn,
        const char *buffer,
        int nbytes);
```

指定*buffer*中长度为*nbytes*的COPY数据传输到服务器。如果数据被放在队列中，结果是 1；如果因为缓冲区满而无法被放在队列中（只可能发生在连接是非阻塞模式时），那么结果是0；如果发生错误，结果为 -1（如果返回值为 -1，那么使用UXSQLerrorMessage检索细节。如果值是0，那么等待可写然后重试）。

应用可以把COPY数据流划分成任意方便大小放到缓冲区中。在发送时，缓冲区负载边界没有语义意义。数据流的内容必须匹配COPY命令预期的数据格式。

UXSQLputCopyEnd

在COPY_IN状态中向服务器发送数据结束的指示。

```
int UXSQLputCopyEnd(UXconn *conn,
        const char *errormsg);
```

如果*errormsg*是NULL，则成功结束COPY_IN操作。如果*errormsg*不是NULL，则COPY被强制失败，*errormsg*指向的字符串是错误消息（不过，服务器不一定会传回准确的错误信息，因为服务器可能已经因为其自身原因导致COPY失败。还要注意的是在使用 3.0 协议之前的连接时，强制失败的选项是不能用的）。

如果终止消息被发送，则结果为 1；在非阻塞模式中，结果为 1 也可能只表示终止消息被成功地放在了发送队列中（在非阻塞模式中，要确认数据确实被发送出去，应该等待可写并且调用UXSQLflush，重复这些直到返回0）。0表示该函数由于缓冲区满而无法将该终止消息放在队列中，这只会发生在非阻塞模式中（在这种情况下，等待可写并且再次尝试UXSQLputCopyEnd调用）。如果发生系统错误，则返回 -1，可以使用UXSQLerrorMessage检索详情。

在成功调用UXSQLputCopyEnd之后，调用UXSQLgetResult获取COPY命令的最终结果状态。可以用常规方法等待可用结果。然后返回到正常的操作。

10.2. 接收COPY数据的函数

这些函数用于在COPY TO STDOUT的过程中接收数据。连接不在COPY_OUT状态时调用它们将会失败。

UXSQLgetCopyData

在COPY_OUT状态下从服务器接收数据。

```
int UXSQLgetCopyData(UXconn *conn,
        char **buffer,
        int async);
```

在COPY期间尝试从服务器获取另外一行数据。数据总是以每次一个数据行的方式被返回；如果只有一个部分行可用，那么它不会被返回。成功返回一个数据行涉及到分配一块内存来保存该数据。*buffer*参数必须为非NULL。**buffer*被设置为指向分配到的内存的指针，或者是在没有返回缓冲区的情况下指向NULL。一个非NULL的结果缓冲区在不需要时必须用UXSQLfreemem释放。

在成功返回一行之后，返回的值就是该数据行里数据的字节数（大于0）。被返回的字符串总是空终止的，虽然这可能只是对文本COPY有用。 结果为0表示该COPY仍然在处理中，但是还没有可用的行（只在*async*为真时才可能）。结果为-1表示COPY已经完成。结果为-2表示发生了错误（参考UXSQLerrorMessage获取原因）。

当*async*为真时（非零），UXSQLgetCopyData将不会阻塞等待输入；如果COPY仍在处理过程中并且没有可用的完整行，那么它将返回 0 （在这种情况下等待可读，然后在再次调用UXSQLgetCopyData之前，调用UXSQLconsumeInput）。当*async*为假（零）时，UXSQLgetCopyData将阻塞，直到数据可用或者操作完成。

在UXSQLgetCopyData返回 -1 之后，调用UXSQLgetResult获取COPY命令的最后结果状态。可以用常规方法等待可用结果。然后返回到正常的操作。

第 11 章 控制函数

这些函数控制libuxsql行为的各种细节。

UXSQLclientEncoding

返回客户端编码。

int UXSQLclientEncoding(const UXconn *_conn_);

请注意，它返回的是编码ID，而不是一个符号串字符串，如EUC_JP。如果不成功，它会返回 -1。要把一个编码ID转换为一个编码名称，命令如下。

char *ux_encoding_to_char(int _encoding_id_);

UXSQLsetClientEncoding

设置客户端编码。

int UXSQLsetClientEncoding(UXconn *_conn_, const char *_encoding_);

_conn_是一个到服务器的连接，而_encoding_是想设置的编码。如果函数成功地设置编码，则返回 0，否则返回 -1。这个连接的当前编码可以使用UXSQLclientEncoding确定。

UXSQLsetErrorVerbosity

决定UXSQLerrorMessage和UXSQLresultErrorMessage返回消息的详细程度。

```
typedef enum
{
  UXSQLERRORS_TERSE,
  UXSQLERRORS_DEFAULT,
  UXSQLERRORS_VERBOSE,
  UXSQLERRORS_SQLSTATE
} UXVerbosity;
```

UXVerbosity UXSQLsetErrorVerbosity(UXconn *conn, UXVerbosity verbosity);

UXSQLsetErrorVerbosity设置细节模式，并返回该连接上之前的设置。

- TERSE模式返回的消息只包括严重性、主要文本以及位置；这些内容通常放在一个单一行中。
- DEFAULT模式生成的消息包括上面的信息加上任何细节、提示或者上下文域（这些字段可能跨越多行）。
- VERBOSE模式包括所有可用的字段。
- SQLSTATE模式仅包括错误严重性和SQLSTATE错误代码，如果其中之一是可用的（如果没有，输出类似于TERSE模式）。

更改详细程度设置不会影响已存在的UXresult对象的可用的消息，只会影响随后创建的对象。（如果想要用不同的详细程度打印之前的错误，请见UXSQLresultVerboseErrorMessage）

UXSQLsetErrorContextVisibility

决定如何处理UXSQLerrorMessage和UXSQLresultErrorMessage返回的消息中的CONTEXT域。

```
typedef enum
{
  UXSQLSHOW_CONTEXT_NEVER,
  UXSQLSHOW_CONTEXT_ERRORS,
  UXSQLSHOW_CONTEXT_ALWAYS
} UXContextVisibility;

UXContextVisibility UXSQLsetErrorContextVisibility(UXconn *conn, UXContextVisibility
 show_context);
```

UXSQLsetErrorContextVisibility设置上下文显示模式，返回该连接上之前的设置。NEVER模式不会包括CONTEXT；ALWAYS则尽可能地包括这个域；ERRORS模式（默认）只在错误消息中包括CONTEXT域，在通知和警告消息中不会包括。（但是，如果详细程度设置为 TERSE或SQLSTATE，则无论上下文显示模式如何，都会省略CONTEXT。）

更改这个模式不会影响从已经存在的UXresult对象项中得到的消息，只会影响后续创建的UXresult对象（如果想要用不同的显示模式打印之前的错误，请见UXSQLresultVerboseErrorMessage）。

UXSQLtrace

启用对客户端/服务器通讯的跟踪，把跟踪信息输出到一个调试文件流中。

```
void UXSQLtrace(UXconn *conn, FILE *stream);
```

> **注意**
>
> 在 Windows上，如果libuxsql库和应用使用了不同的标志编译，那么这个函数调用会导致应用崩溃，因为FILE指针的内部表达是不一样的。特别是多线程/单线程、发布/调试 以及静态/动态标志，库和所有使用库的应用都应一致。

UXSQLuntrace

禁用UXSQLtrace打开的跟踪。

```
void UXSQLuntrace(UXconn *conn);
```

第 12 章 杂项函数

UXSQLfreemem

释放libuxsql分配的内存。

void UXSQLfreemem(void *ptr);

释放libuxsql分配的内存，尤其是UXSQLescapeByteaConn、UXSQLescapeBytea、UXSQLunescapeBytea和UXSQLnotifies分配的内存。特别重要的是，在微软 Windows 上要使用这个函数，而不是free()。这是因为只有 DLL 和应用的多线程/单线程、发布/调试以及静态/动态标志相同时，才能在一个 DLL 中分配内存并且在应用中释放它。在非微软 Windows 平台上，这个函数与标准库函数free()相同。

UXSQLconninfoFree

释放UXSQLconndefaults或UXSQLconninfoParse分配的数据结构。

void UXSQLconninfoFree(UXSQLconninfoOption *connOptions);

不能用UXSQLfreemem替代UXSQLconninfoFree，因为数组包含对子字符串的引用。

UXSQLencryptPasswordConn

准备一个UXDB口令的加密形式。

char *UXSQLencryptPasswordConn(UXconn *conn, const char *passwd, const char *user, const char *algorithm);

这个函数旨在用于那些希望发送类似于ALTER USER joe PASSWORD 'pwd'命令的客户端应用。最好不要在这样一个命令中发送原始的明文密码，因为它可能被暴露在命令日志、活动显示等等中。相反，在发送之前使用这个函数可以将口令转换为加密的形式。

*passwd*和*user*参数是明文口令以及用户的SQL名称。*algorithm*指定用来加密口令的加密算法。当前支持的算法是md5和scram-sha-256（on和off也被接受作为md5的别名，用于与较老的服务器版本兼容）。如果*algorithm*是NULL，这个函数将向服务器查询password_encryption设置的当前值。这种行为可能会阻塞当前事务，并且当前事务被中止或者连接正忙于执行另一个查询时会失败。如果希望为服务器使用默认的算法但避免阻塞，应在调用UXSQLencryptPasswordConn之前查询password_encryption，并且将该值作为*algorithm*传入。

返回值是一个由malloc分配的字符串。调用者可以假设该字符串不含有需要转义的任何特殊字符。在处理完它之后，用UXSQLfreemem释放结果。发生错误时，返回的是NULL，并且在连接对象中存储一条适当的消息。

UXSQLmakeEmptyUXresult

用给定的状态，构造一个空UXresult对象。

UXresult *UXSQLmakeEmptyUXresult(UXconn *conn, ExecStatusType status);

这是libuxsql内部函数，用于分配并初始化一个空的UXresult对象。如果不能分配内存，那么这个函数返回NULL。它也是可以对外使用的，因为一些应用发现生成结果对象（特别是带有错误状态的对象）本身也很有用。如果*conn*非空，并且*status*表示一个错误，那么指定连接的当前错误消息会被复制到UXresult中。另外，如果*conn*非空，那么连接中已注册的任何事件过程也会被复制到UXresult中（它们不会获得UXEVT_RESULTCREATE调用，但会看到UXSQLfireResultCreateEvents）。注意在该对象上最终应该调用UXSQLclear，正如对libuxsql本身返回的UXresult对象一样。

UXSQLfireResultCreateEvents

为每一个在UXresult对象中注册的事件过程触发一个UXEVT_RESULTCREATE事件（请参见第 14 章 事件系统。成功时返回非 0，如果任何事件过程失败则返回 0。

int UXSQLfireResultCreateEvents(UXconn *conn, UXresult *res);

conn参数被传送给事件过程，但不会被直接使用。如果事件过程不使用它，则会返回NULL。

已经接收到这个对象的UXEVT_RESULTCREATE或UXEVT_RESULTCOPY事件的事件过程不会被再次触发。

这个函数独立于UXSQLmakeEmptyUXresult的主要原因是在调用事件过程之前，适合创建一个UXresult并且填充它。

UXSQLcopyResult

为一个UXresult对象创建一个备份。这个备份不会以任何方式链接到源结果，并且当该备份不再需要时，必须调用UXSQLclear进行清理。如果函数失败，返回NULL。

UXresult *UXSQLcopyResult(const UXresult *src, int flags);

这个函数的意图并非是制作一个准确的备份。返回的结果总是会被放入UXRES_TUPLES_OK状态，并且不会拷贝来源中的任何错误消息（不过它确实会拷贝命令状态字符串）。*flags*参数决定还要拷贝的内容。它通常是几个标志的位与结果。UX_COPYRES_ATTRS指定复制源结果的属性（列定义）。UX_COPYRES_TUPLES指定复制源结果的元组（这也意味着复制属性）。UX_COPYRES_NOTICEHOOKS指定复制源结果的通知钩子。UX_COPYRES_EVENTS指定复制源结果的事件（但是不会复制与源结果相关的实例数据）。

UXSQLsetResultAttrs

设置UXresult对象的属性。

int UXSQLsetResultAttrs(UXresult *res, int numAttributes, UXresAttDesc *attDescs);

提供的*attDescs*被复制到结果中。如果*attDescs*指针为NULL或*numAttributes*小于1，那么请求将被忽略并且函数成功。如果*res*已经包含属性，那么函数会失败。如果函数失败，返回值是0。如果函数成功，返回值是非 0。

UXSQLsetvalue

设置一个UXresult对象的一个元组域值。

int UXSQLsetvalue(UXresult *res, int tup_num, int field_num, char *value, int len);

这个函数将自动按需增加结果的内置元组数组。但是，*tup_num*参数必须小于等于UXSQLntuples，意味着这个函数对元组数组一次只能增加一个元组。但已存在的任意元组中的任意域可以以任意顺序进行调整。如果*field_num*的值已经存在，它会被覆盖。如果*len*是 -1，或*value*是NULL， 该域值会被设置为一个 SQL 空值。*value*会被复制到结果的私有存储中，因此函数返回后就不再需要了。如果函数失败，返回值是 0。如果函数成功，返回值会是非 0。

UXSQLresultAlloc

为一个UXresult对象分配附属存储。

void *UXSQLresultAlloc(UXresult *res, size_t nBytes);

当*res*被清除时，这个函数分配的内存也会被释放掉。如果函数失败，返回值是NULL。结果被保证为按照数据的任意类型充分地对齐，类似malloc。

UXSQLresultMemorySize

检索为UXresult对象分配的字节数。

size_t UXSQLresultMemorySize(const UXresult *res);

此值是与UXresult对象关联的所有malloc请求的总和，就是说，UXSQLclear将释放的所有空间。此信息可用于管理内存消耗。

UXSQLlibVersion

返回所使用的libuxsql版本。

int UXSQLlibVersion(void);

在运行时，这个函数的结果可以被用来决定在当前已载入的 libuxsql 版本中特定的功能是否可用。例如，这个函数可以被用来决定哪些选项可以被用于UXSQLconnectdb。

结果通过将库的主版本号乘以10000再加上次版本号形成。

第 13 章 通知处理

服务器产生的通知和警告消息不会被查询执行函数返回，因为它们不代表查询失败。它们可以被传递给一个通知处理函数，并且在处理者返回后执行会继续正常进行。默认的处理函数会把消息打印在stderr上，但是应用可以通过提供它自己的处理函数来重载这种行为。

由于历史原因，通知处理有两个级别，称为通知接收器和通知处理器。通知接收器的默认行为是格式化通知并且将一个字符串传递给通知处理器来打印。不过，如果一个应用选择提供自己的通知接收器，它通常会忽略通知处理器层并且在通知接收器中完成所有工作。

函数UXSQLsetNoticeReceiver 为一个连接对象设置或者检查当前的通知接收器。相似地，UXSQLsetNoticeProcessor 设置或检查当前的通知处理器。

```
typedef void (*UXSQLnoticeReceiver) (void *arg, const UXresult *res);

UXSQLnoticeReceiver
UXSQLsetNoticeReceiver(UXconn *conn,
                       UXSQLnoticeReceiver proc,
                       void *arg);

typedef void (*UXSQLnoticeProcessor) (void *arg, const char *message);

UXSQLnoticeProcessor
UXSQLsetNoticeProcessor(UXconn *conn,
                        UXSQLnoticeProcessor proc,
                        void *arg);
```

这些函数中的每一个会返回之前的通知接收器或处理器函数指针，并且设置新值。如果提供了一个空函数指针，将不会采取任何动作，只会返回当前指针。

当接收到一个服务器产生的或者libuxsql内部产生的通知或警告消息，通知接收器函数会被调用。它会以一种UXRES_NONFATAL_ERROR UXresult的形式传递该消息（这允许接收器使用UXSQLresultErrorField抽取个别的域，或者使用UXSQLresultErrorMessage或者UXSQLresultVerboseErrorMessage得到一个完整的预格式化的消息）。被传递给UXSQLsetNoticeReceiver的同一个空指针也被传递（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知接收器会简单地抽取消息（使用UXSQLresultErrorMessage）并且将它传递给通知处理器。

通知处理器负责处理一个以文本形式给出的通知或警告消息。该消息的字符串文本（包括一个收尾的新行）被传递给通知处理器，外加一个同时被传递给UXSQLsetNoticeProcessor的空指针（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知处理器很简单。

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
   fprintf(stderr, "%s", message);
}
```

一旦设定了一个通知接收器或处理器，只要UXconn对象或者从它构造出的UXresult对象存在，该函数就应该能被调用。在一个UXresult创建时，UXconn的当前通知处理指针被复制到UXresult中，以备类似UXSQLgetvalue的函数使用。

第 14 章 事件系统

libuxsql的事件系统用于通知已注册的事件处理器关注的libuxsql事件，例如UXconn以及UXresult对象的创建和销毁。一种主要的使用情况是应用将自己的数据与一个UXconn或者UXresult关联在一起，并且确保那些数据在适当的时候被释放。

每一个已注册的事件处理器与两部分数据相关，对于libuxsql它们只是透明的void *指针。当事件处理器被注册到一个UXconn时，会有一个应用提供的转移指针。该转移指针在UXconn及其产生的所有UXresult的生命期内都不会改变。因此，如果使用它，它必须指向长期存在的数据。此外，还有一个instance data指针，它在每一个UXconn和UXresult中都开始于NULL。这个指针可以使用 UXSQLinstanceData、 UXSQLsetInstanceData、 UXSQLresultInstanceData和UXSQLsetResultInstanceData函数操纵。注意和转移指针不同，一个UXconn的实例数据不会被从它创建的UXresult自动继承。libuxsql不知道转移和实例数据指针指向的是什么（如果有），并且不会尝试释放它们—这属于事件处理器的责任。

14.1. 事件类型

枚举UXEventId命名了事件系统处理的事件类型。它的所有值的名称都以UXEVT开始。对于每一种事件类型，都有一个相应的事件信息结构用来承载传递给事件处理器的参数。事件类型如下。

UXEVT_REGISTER

当UXSQLregisterEventProc被调用时，注册事件会发生。这是初始化每个事件过程都可能需要的instanceData的最佳时机。每个连接的每个事件处理器只会触发一个注册事件。如果该事件过程失败，注册会被中止。

```
typedef struct
{
  UXconn *conn;
} UXEventRegister;
```

当收到一个UXEVT_REGISTER事件时，*evtInfo*指针应强制转换为UXEventRegister *。这个结构包含一个状态应该为CONNECTION_OK 的 UXconn，保证在得到一个良好的UXconn之后能立即调用UXSQLregisterEventProc。当返回一个失败代码时，所有的清理都必须被执行而不会发送UXEVT_CONNDESTROY事件。

UXEVT_CONNRESET

连接重置事件在UXSQLreset或UXSQLresetPoll完成时被触发。在两种情况中，只有重置成功才会触发该事件。如果事件过程失败，整个连接重置将失败，UXconn会被置为CONNECTION_BAD状态并且UXSQLresetPoll将返回UXRES_POLLING_FAILED。

```
typedef struct
{
  UXconn *conn;
} UXEventConnReset;
```

当收到一个UXEVT_CONNRESET事件时，*evtInfo*指针应强制转换为UXEventConnReset *。尽管所包含的UXconn刚被重置，所有的事件数据还是保持不变。这个事件应该被用来重置/重载/重新查询任何相关的instanceData。注意即使事件过程无法处

理UXEVT_CONNRESET，它仍将在连接被关闭时接收到一个UXEVT_CONNDESTROY事件。

UXEVT_CONNDESTROY

为了响应UXSQLfinish，连接销毁事件会被触发。由于 libuxsql 没有能力管理事件数据，事件过程有责任正确地清理它的事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
  UXconn *conn;
} UXEventConnDestroy;
```

当接收到一个UXEVT_CONNDESTROY事件时，*evtInfo*指针应强制转换为 UXEventConnDestroy *。这个事件在UXSQLfinish执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示一个来自UXSQLfinish的失败。还有，一个事件过程失败不该中断对不需要的内存的清理。

UXEVT_RESULTCREATE

为了响应任何生成一个结果的查询执行函数，结果创建事件会被触发。这些函数包括UXSQLgetResult。这个事件只有在结果被成功地创建之后才会被触发。

```
typedef struct
{
  UXconn *conn;
  UXresult *result;
} UXEventResultCreate;
```

当接收到一个UXEVT_RESULTCREATE事件时，*evtInfo*指针应强制转换为 UXEventResultCreate *。*conn*是用来生成结果的连接。这是初始化任何需要与结果关联的instanceData的理想位置。如果该事件过程失败，结果将被清除并且失败将会被传播。该事件过程不能尝试为自己的结果对象UXSQLclear操作。当返回一个失败代码时，所有清理必须被执行并且不会发送UXEVT_RESULTDESTROY事件。

UXEVT_RESULTCOPY

为了响应UXSQLcopyResult，结果复制事件会被触发。这个事件只会在复制完成后才被触发。只有成功地处理了UXEVT_RESULTCREATE和UXEVT_RESULTCOPY事件的事件过程才将会收到UXEVT_RESULTCOPY事件。

```
typedef struct
{
  const UXresult *src;
  UXresult *dest;
} UXEventResultCopy;
```

当收到一个UXEVT_RESULTCOPY事件时，*evtInfo*指针应强制转换为UXEventResultCopy *。*src*结果是被复制的内容，而*dest*结果则是复制的目标。这个事件可以被用来提供instanceData的一份深度副本，因为UXSQLcopyResult无法做到这一点。如果该事件过程失败，整个复制操作将失败并且*dest*结果将被清除。当返回失败代码时，所有清理必须被执行而不会为目标结果发送UXEVT_RESULTDESTROY事件。

UXEVT_RESULTDESTROY

为了响应UXSQLclear，结果销毁事件会被触发。由于 libuxsql 无法管理内存，事件过程的职责是正确清理其事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
  UXresult *result;
} UXEventResultDestroy;
```

当接收到一个UXEVT_RESULTDESTROY事件时，*evtInfo*指针应强制转换为 UXEventResultDestroy *。这个事件在UXSQLclear执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示来自UXSQLclear的失败。还有，一个事件过程失败不该中断不需要的内存的清理过程。

14.2. 事件回调函数

UXEventProc

UXEventProc是到一个事件过程的指针的 typedef，也就是从 libuxsql 接收事件的用户回调函数。一个事件过程的原型必须如下。

```
int eventproc(UXEventId evtId, void *evtInfo, void *passThrough)
```

*evtId*指示发生了哪一个UXEVT事件。*evtInfo*指针必须强制转换为合适的结构类型才能获得关于事件的进一步信息。当事件过程已被注册时，*passThrough*参数是提供给UXSQLregisterEventProc的指针。如果成功，该函数应该返回非零值，失败则返回零。

在任何一个UXconn中，一个特定事件过程只能被注册一次。这是因为该过程的地址被用作查找键来标识相关的实例数据。

> **注意**
>
> 在 Windows 上，函数能够有两个不同的地址：一个对 DLL 之外可见而另一个对 DLL 之内可见。应当特别注意只有其中之一会被用于libuxsql的事件过程函数，否则将会产生混淆。编写代码的最简单规则是将所有的事件过程声明为static。如果过程的地址必须对它自己的源代码文件之外可见，则提供一个单独的函数来返回该地址。

14.3. 事件支持函数

UXSQLregisterEventProc

为 libuxsql 注册一个事件回调过程。

```
int UXSQLregisterEventProc(UXconn *conn, UXEventProc proc,
                const char *name, void *passThrough);
```

在每一个想要接收事件的UXconn上必须注册一个事件过程。和内存不同，一个连接能注册多少个事件过程没有限制。如果该函数成功，它会返回一个非零值。如果它失败，则会返回零。

当一个 libuxsql 事件被触发时，*proc*参数将被调用。它的内存地址也被用来查找instanceData。*name*参数被用来在错误消息中引用该事件过程。这个值不能是NULL或一个零长度串。名称字符串被复制到UXconn中，因此传递的内容不需要长期存在。当一个事件发生时，*passThrough*指针被传递给*proc*。这个参数可以是NULL。

UXSQLsetInstanceData

设置连接*conn*的用于过程*proc*的instanceData为*data*。它在成功时返回非零值，失败时返回零（只有*proc*没有被正确地注册在*conn*中，才可能会失败）。

```
int UXSQLsetInstanceData(UXconn *conn, UXEventProc proc, void *data);
```

UXSQLinstanceData

返回连接*conn*的与过程*proc*相关的instanceData，如果没有则返回NULL。

```
void *UXSQLinstanceData(const UXconn *conn, UXEventProc proc);
```

UXSQLresultSetInstanceData

把结果的用于*proc*的instanceData设置为*data*。成功返回非零，失败返回零（只有*proc*没有被正确地注册在*conn*中，才可能会失败）。

```
int UXSQLresultSetInstanceData(UXresult *res, UXEventProc proc, void *data);
```

请注意，*data*表示的任何存储都不会由UXSQLresultMemorySize考虑，除非使用UXSQLresultAlloc分配。（推荐这种做法，因为它消除了在销毁结果时显式释放此类存储的需要。）

UXSQLresultInstanceData

返回与过程*proc*相关的结果instanceData，如果没有则返回NULL。

```
void *UXSQLresultInstanceData(const UXresult *res, UXEventProc proc);
```

14.4. 事件实例

下面是一个管理与 libuxsql 连接和结果相关的私有数据的框架示例。

```
/* 要求 libuxsql 事件的头文件（注意：包括 libuxsql-fe.h） */
#include <libuxsql-events.h>

/* instanceData */
typedef struct
{
   int n;
   char *str;
} mydata;
```

```
/* UXEventProc */
static int myEventProc(UXEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
   mydata *data;
   UXresult *res;
   UXconn *conn =
      UXSQLconnectdb("dbname=uxdb UXSQLoptions=-csearch_path=");

   if (UXSQLstatus(conn) != CONNECTION_OK)
   {
      fprintf(stderr, "Connection to database failed: %s",
            UXSQLerrorMessage(conn));
      UXSQLfinish(conn);
      return 1;
   }

   /* 在任何应该接收事件的连接上调用一次。
    * 发送一个 UXEVT_REGISTER 给 myEventProc。
    */
   if (!UXSQLregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
   {
      fprintf(stderr, "Cannot register UXEventProc\n");
      UXSQLfinish(conn);
      return 1;
   }

   /* conn 的 instanceData 可用 */
   data = UXSQLinstanceData(conn, myEventProc);

   /* 发送一个 UXEVT_RESULTCREATE 给 myEventProc */
   res = UXSQLexec(conn, "SELECT 1 + 1");

   /* 结果 instanceData 可用 */
   data = UXSQLresultInstanceData(res, myEventProc);

   /* 如果使用了 UX_COPYRES_EVENTS，则发送一个 UXEVT_RESULTCOPY 给 myEventProc */
   res_copy = UXSQLcopyResult(res, UX_COPYRES_TUPLES | UX_COPYRES_EVENTS);

   /* 如果在 UXSQLcopyResult 调用时使用了 UX_COPYRES_EVENTS，结果 instanceData 可用。*/
   data = UXSQLresultInstanceData(res_copy, myEventProc);

   /* 两个清除都发送一个 UXEVT_RESULTDESTROY 给 myEventProc */
   UXSQLclear(res);
   UXSQLclear(res_copy);

   /* 发送一个 UXEVT_CONNDESTROY 给 myEventProc */
   UXSQLfinish(conn);

   return 0;
}
```

```
static int
myEventProc(UXEventId evtId, void *evtInfo, void *passThrough)
{
  switch (evtId)
  {
    case UXEVT_REGISTER:
    {
      UXEventRegister *e = (UXEventRegister *)evtInfo;
      mydata *data = get_mydata(e->conn);

      /* 将应用相关的数据与连接关联起来 */
      UXSQLsetInstanceData(e->conn, myEventProc, data);
      break;
    }

    case UXEVT_CONNRESET:
    {
      UXEventConnReset *e = (UXEventConnReset *)evtInfo;
      mydata *data = UXSQLinstanceData(e->conn, myEventProc);

      if (data)
        memset(data, 0, sizeof(mydata));
      break;
    }

    case UXEVT_CONNDESTROY:
    {
      UXEventConnDestroy *e = (UXEventConnDestroy *)evtInfo;
      mydata *data = UXSQLinstanceData(e->conn, myEventProc);

      /* 因为连接正在被销毁，释放实例数据 */
      if (data)
        free_mydata(data);
      break;
    }

    case UXEVT_RESULTCREATE:
    {
      UXEventResultCreate *e = (UXEventResultCreate *)evtInfo;
      mydata *conn_data = UXSQLinstanceData(e->conn, myEventProc);
      mydata *res_data = dup_mydata(conn_data);

      /* 把应用相关的数据与结果（从 conn 复制过来）关联起来 */
      UXSQLsetResultInstanceData(e->result, myEventProc, res_data);
      break;
    }

    case UXEVT_RESULTCOPY:
    {
      UXEventResultCopy *e = (UXEventResultCopy *)evtInfo;
      mydata *src_data = UXSQLresultInstanceData(e->src, myEventProc);
      mydata *dest_data = dup_mydata(src_data);

      /* 把应用相关的数据与结果（从一个结果复制过来）关联起来 */
```

```
            UXSQLsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case UXEVT_RESULTDESTROY:
        {
            UXEventResultDestroy *e = (UXEventResultDestroy *)evtInfo;
            mydata *data = UXSQLresultInstanceData(e->result, myEventProc);

            /* 因为结果正在被销毁，释放实例数据 */
            if (data)
              free_mydata(data);
            break;
        }

        /* 未知事件 ID，只返回true。 */
        default:
            break;
    }

    return true; /* 事件处理成功 */
}
```

第 15 章 环境变量

下列环境变量能被用于选择默认的连接参数值，如果调用代码没有直接指定值，它们将被用于UXSQLconnectdb、UXSQLsetdbLogin和UXSQLsetdb。有助于防止数据库连接信息被硬编码到简单的客户端应用中。

- UXHOST的行为和host连接参数相同。
- UXHOSTADDR的行为和hostaddr连接参数相同。可以设置它替代或者作为UXHOST的补充，防止DNS查找负担。
- UXPORT的行为和port连接参数相同。
- UXDATABASE的行为和dbname连接参数相同。
- UXUSER的行为和user连接参数相同。
- UXPASSWORD的行为和password连接参数相同。出于安全原因，不推荐使用这个环境变量，因为某些操作系统允许非根用户通过ps看到进程的环境变量。可以考虑使用一个口令文件（见第 16 章 口令文件。
- UXPASSFILE的行为和passfile连接参数相同。
- UXSERVICE的行为和service连接参数相同。
- UXSERVICEFILE指定针对每个用户的连接服务文件名。如果没有设置，默认为~/.ux_service.conf（请参见第 17 章 连接服务文件。
- UXAPPNAME的行为和application_name连接参数相同。
- UXSSLMODE的行为和sslmode连接参数相同。
- UXSSLCOMPRESSION的行为和sslcompression连接参数相同。
- UXSSLCERT的行为和sslcert连接参数相同。
- UXSSLKEY的行为和sslkey连接参数相同。
- UXSSLROOTCERT的行为和sslrootcert连接参数相同。
- UXSSLCRL的行为和sslcrl连接参数相同。
- UXREQUIREPEER的行为和requirepeer连接参数相同。
- UXGSSENCMODE的行为和gssencmode连接参数相同。
- UXKRBSRVNAME的行为和krbsrvname连接参数相同。
- UXGSSLIB的行为和gsslib连接参数相同。
- UXCONNECT_TIMEOUT的行为和connect_timeout连接参数相同。
- UXCLIENTENCODING的行为和client_encoding连接参数相同。
- UXTARGETSESSIONATTRS的行为和target_session_attrs连接参数相同。

下列环境变量可用来为每一个UXDB会话指定默认行为（为每一个用户或每一个数据库设置默认行为的方法还可见ALTER ROLE和ALTER DATABASE命令）。

- UXDATESTYLE设置日期/时间表示的默认风格（等同于SET datestyle TO ...）。
- UXTZ设置默认的时区（等同于SET timezone TO ...）。
- UXGEQO为遗传查询优化器设置默认模式（等同于SET geqo TO ...）。

这些环境变量的正确值可参考SQL 命令SET。

下列环境变量决定libuxsql的内部行为，它们会覆盖编译在程序中的默认值。

- UXSYSCONFDIR设置包含ux_service.conf文件以及未来版本中可能出现的其他系统范围配置文件的目录。
- UXLOCALEDIR设置包含用于消息本地化的locale文件的目录。

第 16 章 口令文件

一个用户主目录中的.uxpass文件能够包含在连接需要时使用的口令（其他情况不会指定口令）。在微软的 Windows 上该文件被命名为%APPDATA%\uxsino\uxpass.conf（其中%APPDATA%指的是用户配置中的应用数据子目录）。另外，可以使用连接参数passfile或者环境变量UXPASSFILE指定一个口令文件。

这个文件应该包含下列格式的行。

hostname:*port*:*database*:*username*:*password*

（可以向该文件增加一个提醒：把上面的行复制到该文件并且在前面加上#）。前四个域的每一个都可以是文本值或者匹配任何内容的*。将使用与当前连接参数匹配的第一行的口令域。(因此，在使用通配符时，把更特殊的项放在前面。)如果一个条目需要包含:或者\，要用\对该字符转义。 如果指定了host连接参数，则主机名称字段与host连接参数匹配；如果指定了hostaddr参数，则与hostaddr参数匹配； 如果两者都没有给出，则搜索主机名localhost。 当连接是一个Unix域套接字连接并且host参数匹配libuxsql的默认套接字目录路径时，也会搜索主机名localhost。在一台后备服务器上，值为replication的数据库字段匹配连接到主服务器的流复制连接。否则数据库字段的用途有限，因为用户对同一个集群中的所有数据库都有相同的口令。

在 Unix 系统上，口令文件上的权限必须不允许所有人或组内访问，可以用chmod 0600 ~/.uxpass这样的命令实现。如果权限没有这么严格，该文件将被忽略。在微软 Windows 上，该文件被假定存储在一个安全的目录中，因此不会进行特别的权限检查。

第 17 章 连接服务文件

连接服务文件允许 libuxsql 连接参数与一个单一服务名称关联。此服务名称可以被一个 libuxsql 连接指定，与其相关的设置将被使用。允许在不重新编译 libuxsql 应用的前提下修改连接参数。可以使用UXSERVICE环境变量指定服务名称。

连接服务文件可以是每个用户各有一个，它位于~/.ux_service.conf或者环境变量UXSERVICEFILE指定的位置。

该文件使用“INI 文件”格式，其中小节名是服务名并且参数是连接参数。列表请参见第 2.2 节 “参数关键词”。示例如下。

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

share/ux_service.conf.sample中提供了一个示例文件。

第 18 章 连接参数的 LDAP 查找

libuxsql可以通过 LDAP 从一个中央服务器检索host或dbname之类的连接参数。这样做的好处是如果一个数据库的连接参数改变，不需要在所有的客户端机器上更新连接信息。

LDAP 连接参数查找使用连接服务文件ux_service.conf（请参见第 17 章 连接服务文件）。ux_service.conf中一个以ldap://开始的行将被识别为一个 LDAP URL 并且将执行一个 LDAP 查询。结果必须是一个keyword = value对列表，它将被用来设置连接选项。URL 必须遵循 RFC 1959 的形式。

ldap://[*hostname*[:*port*]]/*search_base*?*attribute*?*search_scope*?*filter*

其中*hostname*默认为localhost，*port*默认为 389。

一次成功的 LDAP 查找后，ux_service.conf的处理被终止。但是如果联系不上 LDAP 则会继续处理ux_service.conf。这是为了提供一个后备方案，可以加入更多指向不同 LDAP 服务器的 LDAP URL 行、keyword = value对或者默认连接选项。如果在这种情况下仍然希望得到一条错误消息，那么可以在 LDAP URL 之后添加一行语法上不正确的代码。

一个和 LDIF 文件一起创建的 LDAP 条目实例。

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn:mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

可以使用 LDAP URL 查询。

ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)

也可以将常规的服务文件条目和 LDAP 查找混合。ux_service.conf中一节的完整示例如下所示。

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

可以使用 LDAP URL 查询。

ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)

第 19 章 SSL 支持

UXDB本地支持使用SSL连接加密客户端/服务器通信以提高安全性。

libuxsql读取系统范围的OpenSSL配置文件。默认情况下，这个文件被命名为openssl.cnf并且位于openssl version -d所报告的目录中。可以通过设置环境变量OPENSSL_CONF把这个默认值覆盖为想要的配置文件的名称。

19.1. 服务器证书的客户端验证

默认情况下，UXDB不会执行服务器证书的任何验证。可以在不被客户端知晓的情况下伪造服务器身份（例如通过修改一个 DNS 记录或者接管服务器的 IP 地址）。为了防止欺骗，客户端必须能够通过一条信任链验证服务器的身份。

信任链建立方法：客户端验证服务器身份。在客户端上放置一份根（自签名的）证书机构（CA）的证书并且在服务器上放置由根证书签发的子证书；服务器验证客户端身份。在服务器上放置一份根（自签名的）证书机构（CA）的证书并且在客户端上放置由根证书签发的子证书；也可以使用一个或者更多个中间证书（通常与子证书一起存放），它由根证书签发并且可以签发子证书，它可以将子证书链接到根证书。

一旦信任链被建立起来，客户端有两种方法验证服务器发过来的子证书。如果参数sslmode被设置为verify-ca，libuxsql将通过检查该证书是否链接到存储在客户端上的根证书来验证服务器。如果sslmode被设置为verify-full，libuxsql还将验证服务器的主机名匹配存储在服务器证书中的名称。如果服务器证书无法被验证，则SSL连接将失败。在大部分对安全性很敏感的环境中，推荐使用verify-full。

在verify-full模式中，主机名被拿来与证书的主体别名属性 匹配，或者在不存在dNSName类型的主体别名时与通用名称属性匹配。 如果证书的名称属性以一个星号（*）开始，这个星号将被视作一个通配符，它将匹配所有除了句点（.） 之外的字符，即该证书不会匹配子域。如果连接使用 IP 地址而不是由主机名创建，那么该 IP 地址将被匹配（不做任何 DNS 查找）。

要允许服务器证书验证，必须将一个或者更多个根证书放置在用户主目录下的~/.uxsino/root.crt文件中（在Microsoft Windows上该文件名为%APPDATA%\uxsino\root.crt）。如果需要把服务器发来的证书链链接到存储在客户端的根证书，还应该将中间证书加到该文件中。

如果文件~/.uxsino/root.crl存在（微软 Windows 上的%APPDATA%\uxsino\root.crl），证书撤销列表（CRL）项也会被检查。

根证书文件和 CRL 的位置可以通过设置连接参数sslrootcert和sslcrl或环境变量UXSSLROOTCERT和UXSSLCRL改变。

19.2. 客户端证书

如果服务器尝试通过请求客户端的子证书来验证客户端的身份，libuxsql将发送用户主目录下文件~/.uxsino/uxsino.crt中存储的证书。该证书必须链接到该服务器信任的根证书。也必须存在一个匹配的私钥文件~/.uxsino/uxsino.key。该私钥文件不能允许全部用户或者组用户的任何访问，可以通过命令chmod 0600 ~/.uxsino/uxsino.key实现。在微软 Windows 上这些文件被命名为%APPDATA%\uxsino\uxsino.crt和%APPDATA%\uxsino\uxsino.key，不会有特别的权限检查，因为该目录已经被假定为安全。证书和密钥文件的位置可以使用连接参数sslcert和sslkey或者环境变量UXSSLCERT和UXSSLKEY覆盖。

uxsino.crt中的第一个证书必须是客户端的证书，因为它必须匹配客户端的私钥。可以选择将“中间”证书追加到该文件—这样做避免了在服务器上存放中间证书的要求。

19.3. 不同模式中提供的保护

sslmode参数的不同值提供了不同级别的保护。SSL 能够针对以下三类攻击提供保护。

窃听

如果第三方能够检查客户端和服务器之间的网络流量，它就能读取连接信息（包括用户名和口令）以及被传递的数据。SSL使用加密来阻止这种攻击。

中间人（MITM）

如果第三方能对客户端和服务器之间传送的数据进行修改，它就能假装是服务器并且因此能看到和修改数据，即使这些数据已被加密。然后第三方可以将连接信息和数据转送给原来的服务器，使得它不可能检测到攻击。这样做的通常途径包括 DNS 污染和地址劫持，客户端被重定向到一个不同的服务器。还有其他几种的攻击方式都能够完成这种攻击。SSL使用证书验证让客户端认证服务器，就可以阻止这种攻击。

模仿

如果第三方能伪装为授权的客户端，则能够简单地访问它本不能访问的数据。这是由于不安全的口令管理所致。SSL使用客户端证书来确保只有持有合法证书的客户端才能访问服务器，这样就能阻止这种攻击。

对于一个已知的SSL-secured连接，在连接被建立之前，SSL 使用必须在客户端和服务器同时配置。如果只在服务器上配置，客户端在知道服务器要求高安全性之前可能会结束发送敏感信息（例如口令）。在 libuxsql 中，要确保连接安全，可以设置sslmode参数为verify-full或verify-ca并且为系统提供一个根证书用来验证。这类似于使用https URL进行加密网页浏览。

当服务器被认证，客户端可以传递敏感数据。这意味着在此前之前，客户端都不需要知道是否使用证书进行身份认证，因此只在服务器配置中指定证书是安全的。

所有SSL选项都有加密和密钥交换的负荷，因此必须在性能和安全性之间做出平衡。表 19.1 “SSL 模式描述”总结了不同sslmode值所应对的风险，以及性能和安全性的声明。

表 19.1. SSL 模式描述

sslmode	窃听保护	MITM保护	声明
disable	否	否	不关心安全性，并且不为加密增加负荷。
allow	可能	否	不关心安全性，但如果服务器坚持，将承担加密带来的负荷。
prefer	可能	否	不关心安全性，但如果服务器支持，希望承担加密带来的负荷。
require	是	否	对数据加密，并且接受因此带来的负荷。信任该网络会保证总是连接到想要连接的服务器。

sslmode	窃听保护	MITM保护	声明
verify-ca	是	取决于 CA 策略	对数据加密，并且接受因此带来的负荷。确保连接到的是信任的服务器。
verify-full	是	是	对数据加密，并且接受因此带来的负荷。确保连接到的是指定并且信任的服务器。

verify-ca和verify-full之间的区别取决于根CA的策略。如果使用了一个公共CA，verify-ca允许连接到那些可能已经被其他注册到该CA的服务器。在这种情况下，应该使用verify-full。如果使用了一个本地CA或者自签名的证书，使用verify-ca就可以提供足够的保护。

sslmode的默认值是prefer。如表中所示，这在安全性的角度来说没有意义，并且它只承诺可能的性能负荷。提供它作为默认值只是为了向后兼容，不推荐在安全部署中使用它。

19.4. SSL 客户端文件使用

表 19.2 “libuxsql/客户端 SSL 文件用法”总结了与客户端 SSL 设置相关的文件。

表 19.2. libuxsql/客户端 SSL 文件用法

文件	内容	效果
~/.uxsino/uxsino.crt	客户端证书	由服务器要求
~/.uxsino/uxsino.key	客户端私钥	证明客户端证书是由拥有者发送；不代表证书拥有者可信
~/.uxsino/root.crt	可信的证书机构	检查服务器证书是由一个可信的证书机构签发
~/.uxsino/root.crl	被证书机构撤销的证书	服务器证书不能在这个列表上

19.5. SSL 库初始化

如果应用初始化libssl或libcrypto库以及带有SSL支持的libuxsql，应该调用UXSQLinitOpenSSL来告诉libuxsql：libssl或libcrypto库已被应用初始化，这样libuxsql将不会也去初始化那些库。

UXSQLinitOpenSSL

允许应用选择要初始化哪个安全性库。

void UXSQLinitOpenSSL(int do_ssl, int do_crypto);

当*do_ssl*是非零时，libuxsql将在第一次打开数据库连接前初始化OpenSSL库。当*do_crypto*是非零时，libcrypto库将被初始化。默认情况下（如果没有调用UXSQLinitOpenSSL），两个库都会被初始化。

如果应用使用并且初始化OpenSSL或者它的底层libcrypto库，必须在第一次打开数据库连接前以合适的非零参数调用这个函数。同时要确保在打开一个数据库连接前已经完成了初始化。

UXSQLinitSSL

允许应用选择要初始化哪个安全性库。

void UXSQLinitSSL(int do_ssl);

这个函数等效于UXSQLinitOpenSSL(do_ssl, do_ssl)。可用于要么初始化OpenSSL以及libcrypto要么都不初始化的应用。

第 20 章 在线程化程序中的行为

libuxsql默认是可重入和线程安全的。需要使用特殊的编译器命令行选项来编译应用代码。这个函数允许查询libuxsql的线程安全状态。

UXSQListhreadsafe

返回libuxsql库的线程安全状态。

int UXSQListhreadsafe();

如果libuxsql是线程安全的则返回 1，否则返回 0。

不允许两个线程同时尝试操纵同一个UXconn对象。不能从不同的线程通过同一个连接对象发出并发的命令（如果需要运行并发命令，可以使用多个连接）。

UXresult对象在创建后通常是只读的，并且因此可以在线程之间自由地被传递。但是，如果使用任何第 12 章 杂项函数或第 14 章 事件系统描述的UXresult修改函数，需要避免在同一个UXresult上的并发操作。

如果在应用中使用 Kerberos （除了在libuxsql中之外），需要对 Kerberos 调用加锁，因为Kerberos 函数不是线程安全的。参考libuxsql源代码中的UXSQLregisterThreadLock函数，可以了解在libuxsql和应用之间做合作锁定的方法。

如果在线程化应用中遇到问题，可以将该程序运行在src/tools/thread来查看是否平台有线程不安全的函数。这个程序会被configure运行，但是对于二进制发布，可能和用来编译二进制的库不匹配。

第 21 章 编译 libuxsql 程序

编译（即编译并且链接）一个使用libuxsql的程序，步骤如下。

- include libuxsql-fe.h 头文件。

  #include <libuxsql-fe.h>

  如果没有，那么编译器会发出如下错误消息。

  foo.c: In function `main':
  foo.c:34: `UXconn' undeclared (first use in this function)
  foo.c:35: `UXresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `UXRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `UXRES_TUPLES_OK' undeclared (first use in this function)

- 通过为编译器提供-I*directory*选项，向编译器指出UXDB头文件安装在哪里（在某些情况下编译器默认将查看该目录，因此可以忽略这个选项）。编译命令行示例如下。

  cc -c -I/home/uxdb/uxdbinstall/dbsql/include testprog.c

  如果使用 makefile，那么把该选项添加到CPPFLAGS变量中。

  CPPFLAGS += -I/home/uxdb/uxdbinstall/dbsql/include

  如果程序可能由其他用户编译，那么不应该硬编码目录位置。可以运行ux_config工具，在本地系统上找出头文件位置。

  $ ux_config --includedir
  /home/uxdb/uxdbinstall/dbsql/include

  如果安装了pkg-config，命令如下。

  $ pkg-config --cflags libuxsql
  -I/home/uxdb/uxdbinstall/dbsql/include

  如果没有为编译器指定正确的选项将导致错误消息。

  testlibuxsql.c:8:22: libuxsql-fe.h: No such file or directory

- 当链接最终的程序时，指定选项-luxsql，libuxsql库会被编译进去，也可以用选项-L*directory*向编译器指出libuxsql库所在的位置（再次，编译器将默认搜索某些目录）。为了最大的可移植性，将-L选项放在-luxsql选项前面。

  cc -o testprog testprog1.o testprog2.o -L/home/uxdb/uxdbinstall/dbsql/lib -luxsql

使用ux_config可以找出库目录。

```
$ ux_config --libdir
/home/uxdb/uxdbinstall/dbsql/lib
```

或者使用pkg-config。

```
$ pkg-config --libs libuxsql
-L/home/uxdb/uxdbinstall/dbsql/lib -luxsql
```

这会打印出全部的选项，而不仅仅是路径。

指出这一部分问题的错误消息。

```
testlibuxsql.o: In function `main':
testlibuxsql.o(.text+0x60): undefined reference to `UXSQLsetdbLogin'
testlibuxsql.o(.text+0x71): undefined reference to `UXSQLstatus'
testlibuxsql.o(.text+0xa4): undefined reference to `UXSQLerrorMessage'
```

这表示缺少-l选项或者没有指定正确的目录。

```
/usr/bin/ld: cannot find -luxsql
```

第 22 章 示例程序

例 22.1. libuxsql 示例程序 1

```
/*
 *
 *
 * testlibuxsql.c
 *
 *     测试 libuxsql（UXDB 前端库） 的 C 版本。
 */
#include <stdio.h>
#include <stdlib.h>
#include "libuxsql-fe.h"

static void
exit_nicely(UXconn *conn)
{
 UXSQLfinish(conn);
 exit(1);
}

int
main(int argc, char **argv)
{
 const char *conninfo;
 UXconn    *conn;
 UXresult   *res;
 int   nFields;
 int   i,
   j;

 /*
    * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
    * 否则默认用设置 dbname=uxdb 并且为所有其他链接参数使用环境变量或默认值。
    */
 if (argc > 1)
  conninfo = argv[1];
 else
  conninfo = "dbname = uxdb";

 /* 建立到数据库的一个连接 */
 conn = UXSQLconnectdb(conninfo);

 /* 检查看后端连接是否成功建立 */
 if (UXSQLstatus(conn) != CONNECTION_OK)
 {
  fprintf(stderr, "Connection to database failed: %s",
    UXSQLerrorMessage(conn));
  exit_nicely(conn);
```

```
}

/* 设置安全搜索路径，使恶意用户无法取得控制。 */
res = UXSQLexec(conn,
   "SELECT ux_catalog.set_config('search_path', '', false)");
if (UXSQLresultStatus(res) != UXRES_TUPLES_OK)
{
 fprintf(stderr, "SET failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}

 /*
   * 任何时候不再需要 UXresult 时，应该 UXSQLclear 它来避免内存泄露
   */
UXSQLclear(res);

/*
   * 测试案例这里涉及使用一个游标，它必须用在一个事务块内。
   * 可以在一个单一的 "select * from ux_database" 的 UXSQLexec() 中操作，
   * 但是过于琐碎，不能作为一个好的示例。
   */

  /* 开始一个事务块 */
res = UXSQLexec(conn, "BEGIN");
if (UXSQLresultStatus(res) != UXRES_COMMAND_OK)
{
 fprintf(stderr, "BEGIN command failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}
UXSQLclear(res);

/*
   * 从 ux_database 取得行，它是数据库的系统目录
   */
res = UXSQLexec(conn, "DECLARE myportal CURSOR FOR select * from ux_database");
if (UXSQLresultStatus(res) != UXRES_COMMAND_OK)
{
 fprintf(stderr, "DECLARE CURSOR failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}
UXSQLclear(res);

res = UXSQLexec(conn, "FETCH ALL in myportal");
if (UXSQLresultStatus(res) != UXRES_TUPLES_OK)
{
 fprintf(stderr, "FETCH ALL failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}

/* 首先，打印出属性名 */
```

```
nFields = UXSQLnfields(res);
for (i = 0; i < nFields; i++)
 printf("%-15s", UXSQLfname(res, i));
printf("\n\n");

/* 接下来，打印出行 */
for (i = 0; i < UXSQLntuples(res); i++)
{
 for (j = 0; j < nFields; j++)
  printf("%-15s", UXSQLgetvalue(res, i, j));
 printf("\n");
}

UXSQLclear(res);

/* 关闭入口，不需要考虑检查错误 */
res = UXSQLexec(conn, "CLOSE myportal");
UXSQLclear(res);

/* 结束事务 */
res = UXSQLexec(conn, "END");
UXSQLclear(res);

/* 关闭到数据库的连接并且清理 */
UXSQLfinish(conn);

return 0;
}
```

例 22.2. libuxsql示例程序 2

```
/*
 *
 *
 * testlibuxsql2.c
 *     测试异步通知接口
 *
 * 开始这个程序，然后在另一个窗口的 uxsql 中
 *   NOTIFY TBL2;
 * 重复四次让这个程序退出。
 *
 * 或者，尝试用下列命令填充一个数据库
 *
 *  CREATE SCHEMA TESTLIBUXSQL2;
 *  SET search_path = TESTLIBUXSQL2;
 *  CREATE TABLE TBL1 (i int4);
 *  CREATE TABLE TBL2 (i int4);
 *  CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *    (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * Start this program, then from uxsql do this four times:
```

```
 *
 *  INSERT INTO TESTLIBUXSQL2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libuxsql-fe.h"

static void
exit_nicely(UXconn *conn)
{
 UXSQLfinish(conn);
 exit(1);
}

int
main(int argc, char **argv)
{
 const char *conninfo;
 UXconn    *conn;
 UXresult   *res;
 UXnotify   *notify;
 int   nnotifies;

 /*
    * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
    * 否则默认用设置 dbname=uxdb 并且为所有其他链接参数使用环境变量或默认值。
    */
 if (argc > 1)
  conninfo = argv[1];
 else
  conninfo = "dbname = uxdb";

 /* 建立一个到数据库的连接 */
 conn = UXSQLconnectdb(conninfo);

 /* 检查后端连接是否成功建立 */
 if (UXSQLstatus(conn) != CONNECTION_OK)
 {
  fprintf(stderr, "Connection to database failed: %s",
    UXSQLerrorMessage(conn));
  exit_nicely(conn);
 }
```

```
/* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
res = UXSQLexec(conn,
   "SELECT ux_catalog.set_config('search_path', '', false)");
if (UXSQLresultStatus(res) != UXRES_TUPLES_OK)
{
 fprintf(stderr, "SET failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}

 /*
   * 任何时候不再需要 UXresult 时，应该 UXSQLclear 来避免内存泄露
   */
UXSQLclear(res);

/*
   * 发出 LISTEN 命令启用规则的 NOTIFY 的通知。
   */
res = UXSQLexec(conn, "LISTEN TBL2");
if (UXSQLresultStatus(res) != UXRES_COMMAND_OK)
{
 fprintf(stderr, "LISTEN command failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}
UXSQLclear(res);

/* 在接收到四个通知后退出。 */
nnotifies = 0;
while (nnotifies < 4)
{
 /*
       * 在连接上发生某些情况前休眠状态。可以使用 select(2) 来等待输入，也可以使用 poll() 或
类似工具。
       */
 int   sock;
 fd_set  input_mask;

 sock = UXSQLsocket(conn);

 if (sock < 0)
  break;    /* 不应该发生 */

 FD_ZERO(&input_mask);
 FD_SET(sock, &input_mask);

 if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
 {
  fprintf(stderr, "select() failed: %s\n", strerror(errno));
  exit_nicely(conn);
 }

 /* 立即检查输入 */
```

```
 UXSQLconsumeInput(conn);
 while ((notify = UXSQLnotifies(conn)) != NULL)
 {
  fprintf(stderr,
    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
    notify->relname, notify->be_pid);
  UXSQLfreemem(notify);
  nnotifies++;
  UXSQLconsumeInput(conn);
 }
}

fprintf(stderr, "Done.\n");

/* 关闭到数据库的连接并且清理 */
UXSQLfinish(conn);

return 0;
}
```

例 22.3. libuxsql示例程序 3

```
/*
 *
 *
 * testlibuxsql3.c
 *     测试线外参数和二进制 I/0。
 *
 * 在运行之前，使用下列命令填充一个数据库
 *
 * CREATE SCHEMA testlibuxsql3;
 * SET search_path = testlibuxsql3;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe"s place', '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');
 *
 * 期望输出。
 *
 * tuple 0: got
 * i = (4 bytes) 1
 * t = (11 bytes) 'joe's place'
 * b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 * i = (4 bytes) 2
 * t = (8 bytes) 'ho there'
 * b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
```

```
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libuxsql-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(UXconn *conn)
{
 UXSQLfinish(conn);
 exit(1);
}

/*
 * 这个函数打印一个查询结果，该结果以二进制格式从上面注释中定义的表中取得。
 * 把它分离出来是因为 main() 函数需要使用它两次。
 */
static void
show_binary_results(UXresult *res)
{
 int   i,
   j;
 int   i_fnum,
   t_fnum,
   b_fnum;

 /* 使用 UXSQLfnumber 来避免假定结果中域的顺序 */
 i_fnum = UXSQLfnumber(res, "i");
 t_fnum = UXSQLfnumber(res, "t");
 b_fnum = UXSQLfnumber(res, "b");

 for (i = 0; i < UXSQLntuples(res); i++)
 {
  char    *iptr;
  char    *tptr;
  char    *bptr;
  int   blen;
  int   ival;

  /* 得到域值（忽略它们为空值的可能性！） */
  iptr = UXSQLgetvalue(res, i, i_fnum);
  tptr = UXSQLgetvalue(res, i, t_fnum);
  bptr = UXSQLgetvalue(res, i, b_fnum);

  /*
       * INT4 的二进制表示是按照网络字节序的，建议强制为本地字节序。
```

```
        */
 ival = ntohl(*((uint32_t *) iptr));

 /*
        * TEXT 的二进制表示是文本，并且因为 libuxsql 会为它追加一个零字节，它和 C 字符串效果相同。
        *
        * BYTEA 的二进制表示是一堆字节，其中可能包含嵌入的空值，因此必须注意域长度。
        */
 blen = UXSQLgetlength(res, i, b_fnum);

 printf("tuple %d: got\n", i);
 printf(" i = (%d bytes) %d\n",
    UXSQLgetlength(res, i, i_fnum), ival);
 printf(" t = (%d bytes) '%s'\n",
    UXSQLgetlength(res, i, t_fnum), tptr);
 printf(" b = (%d bytes) ", blen);
 for (j = 0; j < blen; j++)
  printf("\\%03o", bptr[j]);
 printf("\n\n");
 }
}

int
main(int argc, char **argv)
{
 const char *conninfo;
 UXconn    *conn;
 UXresult   *res;
 const char *paramValues[1];
 int   paramLengths[1];
 int   paramFormats[1];
 uint32_t binaryIntVal;

 /*
    * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
    * 否则默认用设置 dbname=uxdb 并且为所有其他链接参数使用环境变量或默认值。
    */
 if (argc > 1)
  conninfo = argv[1];
 else
  conninfo = "dbname = uxdb";

 /* 建立一个到数据库的连接 */
 conn = UXSQLconnectdb(conninfo);

 /* 检查看后端连接是否成功被建立 */
 if (UXSQLstatus(conn) != CONNECTION_OK)
 {
  fprintf(stderr, "Connection to database failed: %s",
    UXSQLerrorMessage(conn));
  exit_nicely(conn);
 }
```

```
/* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
res = UXSQLexec(conn, "SET search_path = testlibuxsql3");
if (UXSQLresultStatus(res) != UXRES_COMMAND_OK)
{
 fprintf(stderr, "SET failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}
UXSQLclear(res);

/*
   * 这个程序的要点在于用线外参数展示 UXSQLexecParams() 的使用，以及数据的二进制传输。
   *
   * 第一个示例将参数作为文本传输，但是以二进制格式接收结果。
   * 通过使用线外参数，能够避免使用繁杂的引用和转义，即便数据是文本。
   * 注意怎么才能对参数值中的引号不做任何操作。
   */

  /* 线外参数值 */
paramValues[0] = "joe's place";

res = UXSQLexecParams(conn,
      "SELECT * FROM test1 WHERE t = $1",
      1,  /* 一个参数 */
                NULL,    /* 让后端推导参数类型 */
                paramValues,
                NULL,    /* 因为文本不需要参数长度 */
                NULL,    /* 对所有文本参数的默认值 */
                1);      /* 要求二进制结果 */

if (UXSQLresultStatus(res) != UXRES_TUPLES_OK)
{
 fprintf(stderr, "SELECT failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}

show_binary_results(res);

UXSQLclear(res);

/*
   * 在第二个示例中，以二进制形式传输一个整数参数，并且再次以二进制形式接收结果。
   *
   * 尽管设置 UXSQLexecParams 让后端推导参数类型，但实际上通过在查询文本中强制转换参数符号来做出该决定。
   * 在发送二进制参数时，这是一种好的安全措施。
   */

  /* 将整数值 "2" 转换为网络字节序 */
binaryIntVal = htonl((uint32_t) 2);

/* 为 UXSQLexecParams 设置参数数组 */
```

```
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;  /* 二进制 */

res = UXSQLexecParams(conn,
      "SELECT * FROM test1 WHERE i = $1::int4",
      1,  /* 一个参数 */
                  NULL,    /* 让后端推导参数类型 */
                  paramValues,
                  paramLengths,
                  paramFormats,
                  1);      /* 要求二进制结果 */

if (UXSQLresultStatus(res) != UXRES_TUPLES_OK)
{
 fprintf(stderr, "SELECT failed: %s", UXSQLerrorMessage(conn));
 UXSQLclear(res);
 exit_nicely(conn);
}

show_binary_results(res);

UXSQLclear(res);

/* 关闭到数据库的连接并清理 */
UXSQLfinish(conn);

return 0;
}
```