LoadRunner 函数大全之中文解释

```
// button_press 函数激活指定的推按钮。
int button_press ( LPCSTR button );

// button_set 函数将按钮状态设置为 ON 或 OFF。
int button_set ( LPCSTR button, int state );

// close_session 函数关闭所有打开的窗口并结束
// 当前的 Baan 会话。在 Baan 模板中创建的此函数
// 出现在脚本的 vuser_end 部分中。
int close_session();

// edit_get_text 函数返回在指定 edit 对象中
// 找到的所有文本。若要从特定块中读取文本，
// 请使用 edit_get_block。
int edit_get_text ( LPCSTR edit, char *out_string );

// edit_set 函数使用指定的字符串设置 edit 对象的
// 内容。该字符串将替换任何现有字符串。
int edit_set ( LPCSTR edit, LPCSTR text );

// edit_set_insert_pos 函数将光标放置
// 在 edit 对象内的指定位置。
int edit_set_insert_pos (LPCSTR edit, int row, int column );

// edit_set_selection 函数突出显示指定文本。
int edit_set_selection ( LPCSTR edit, int start_row, int start_column, int end_row,  int
end_column );

// edit_type 函数将文本字符串输入到 edit
// 对象中。该文本字符串不会替换现有字符串；
// 它替换的是位于当前光标位置的指定文本。
int edit_type ( LPCSTR edit, LPCSTR text );

// init_session 函数通过指定登录数据和配置
// 信息打开 Baan 连接。此函数向 Baan 服务器
// 呈现包含在 Baan Configuration 部分中
// 的信息。
int init_session ( char * host, char * user, char *password, char *BSE,  char *Bshell_name, char
* settings );

// list_activate_item 函数双击列表中的项目。
```

```
// 项目可由其逻辑名称指定。
int list_activate_item ( LPCSTR list, LPCSTR item );

// list_collapse_item 函数隐藏展开的 TreeView
// 列表中的子项，例如文件夹中的各个文件。
int list_collapse_item (LPCSTR list, LPCSTR item );

// list_expand_item 函数显示展开的
// TreeView 列表中所隐藏的子项，例如
// 文件夹中的各个文件。
int list_expand_item (LPCSTR list, LPCSTR item );

// list_get_selected 函数返回列表中的选定
// 项目。它既查找标准列表，也查找多选项列表。
int list_get_selected (LPCSTR list, LPCSTR out_item, LPCSTR out_num );

// list_select_item 函数从列表中选择项目
// （在项目上执行一次鼠标单击）。项目可由
// 其名称或数字索引指定。索引被指定为一个
// 字符串，并前置有字符 #。列表中的第一个
// 项目编号为 0。例如，列表中的第三个项目
// 将表示为 "#2"。
int list_select_item ( LPCSTR list, LPCSTR item );

// menu_select_item 函数根据菜单
// 的逻辑名称和项目名称从菜单中选择
// 项目。注意，菜单和项目表示为单个
// 字符串，并使用分号分隔。
int menu_select_item ( LPCSTR menu_item );

// obj_get_info 函数检索指定属性的值，
// 并将其存储在 out_value 中。
int obj_get_info ( LPCSTR object, LPCSTR property, char *out_value );

// obj_get_text 函数从指定的对象
// 或对象区域中读取文本。
int obj_get_text (LPCSTR object, LPCSTR out_text );

// obj_mouse_click 函数在对象内的
// 指定坐标处单击鼠标。
int obj_mouse_click ( LPCSTR object, int x, int y, [mouse_button] );

// obj_mouse_dbl_click 函数在对象内的
// 指定坐标处双击鼠标。
```

```
int obj_mouse_dbl_click ( LPCSTR object, int x, int y, [mouse_button] );

// obj_mouse_drag 函数在 GUI
// 对象中执行鼠标拖动操作。指定的
// 坐标是相对于 GUI 对象（而非
// 屏幕）的左上角。
int obj_mouse_drag (LPCSTR object, int start_x, int start_y, int end_x, int end_y, [ ButtonT
mouse_button] );

// obj_type 函数指定将 keyboard_input
// 发送到的目标对象。
int obj_type ( LPCSTR object, unsigned char keyboard_input, [unsigned char modifier ] );

// obj_wait_info 函数等待对象
// 属性达到指定值，然后继续
// 测试运行。如果未达到指定
// 值，则函数将一直等到时间
// 到期，然后再继续测试。
int obj_wait_info (LPCSTR object, LPCSTR property, LPCSTR value, UINT time );

// scroll_drag_from_min 函数将滚动屏
// 移动到与最小位置相距指定距离的位置。
int scroll_drag_from_min ( LPCSTR object, [int orientation], int position );

// scroll_line 函数滚动指定行数。
// 此函数可用于滚动栏和滑块对象。
int scroll_line ( LPCSTR scroll, [ScrollT orientation], int lines );

// scroll_page 函数将滚动屏移动指定页数。
int scroll_page ( LPCSTR scroll, [ScrollT orientation], int pages );

// set_default_timeout 函数设置回放
// 期间 Baan Vuser 函数的超时期间段。
// 例如，当脚本执行 set_window 函数
// 时，如果窗口在指定超时时间段内没有
// 出现，则会生成错误。
void set_default_timeout ( long time );

// set_exception 函数指定在发生异常时
// 应执行的操作。应指定要调用以处理异常
// 窗口的函数。
void set_exception ( LPCSTR title, long function );

// set_think_time 函数指定脚本执行
```

```
// 期间要使用的思考时间范围。运行脚本
// 时，LoadRunner 使用指定范围内的
// 随机思考时间，并在每个操作完成后
// 暂停该思考时间长度。
void set_think_time ( USHORT start_range, USHORT end_range );

// set_window 函数将输入定向到
// 当前应用程序窗口并在 GUI 图中
// 设置对象标识范围。
int set_window ( LPCSTR window [, int    timeout ] );

// start_session 函数在 Baan
// 服务器上开始指定的会话。
int start_session ( LPCSTR session );

// static_get_text 函数返回在指定
// 静态 text 对象中找到的所有文本。
int static_get_text ( LPCSTR static_obj, LPCSTR out_string );

// tab_select_item 函数选择一个选项卡项目。
int tab_select_item ( LPCSTR tab, LPCSTR item );

// tbl_activate_cell 函数在指定表单元格中
// 按 Enter 键。如果指定了列名，LoadRunner
// 将直接从数据库中获取该名称。
int tbl_activate_cell (LPCSTR table, LPCSTR row, LPCSTR column );

// tbl_get_cell_data 函数根据
// 单元格包含的数据类型获取表中
// 指定单元格的内容。如果指定了
// 列名，将从数据库自身（而非应用
// 程序）中获取该名称。
int tbl_get_cell_data (LPCSTR table, LPCSTR row, LPCSTR column, LPCSTR out_text );

// tbl_get_selected_cell 函数
// 检索焦点所在的表单元格的行号和
// 列名。注意，列名取自数据库自身，
// 而非应用程序。
int tbl_get_selected_cell (LPCSTR table, char *out_row, char *out_column );

// tbl_press_zoom_button 函数
// 激活指定表单元格的缩放窗口。
int tbl_press_zoom_button ( LPCSTR table, LPCSTR row, LPCSTR column );
```

```
// tbl_set_cell_data 函数根据单元格
// 包含的数据类型使用指定数据设置单元格
// 的值。如果指定了列名，LoadRunner
// 将直接从数据库中获取该名称。
int tbl_set_cell_data (LPCSTR table, LPCSTR row, LPCSTR column, LPCSTR data );

// tbl_set_selected_cell 函数将焦点
// 设置到表中的指定单元格上。指定列名时，
// LoadRunner 将直接从数据库中获取该名称。
int tbl_set_selected_cell (LPCSTR table, LPCSTR row, LPCSTR column );

// tbl_set_selected_row 函数选择表中的指定行。
int tbl_set_selected_row (LPCSTR table, LPCSTR row );

// tbl_set_selected_rows 函数选择指定行范围。
int tbl_set_selected_rows(LPCSTR table, LPCSTR from_row , LPCSTR to_row );

// tbl_wait_selected_cell 函数等待
// 表单元格显示后，再继续脚本执行。
int tbl_wait_selected_cell (LPCSTR table, char *row, char *column, UINT time );

// toolbar_button_press 函数激活工具栏中的按钮。
int toolbar_button_press (LPCSTR toolbar, LPCSTR button );

// type 函数描述发送给用于测试
// 的应用程序的键盘输入。
int type (LPCSTR keyboard_input );

// win_activate 函数通过向指定窗口
// 授予焦点并将其升到显示器最上端，
// 使其成为活动窗口（等价于单击窗口
// 标题栏）。所有后续输入都将提交给
// 此窗口。
int win_activate (LPCSTR window );

// win_close 函数关闭指定窗口。
int win_close ( LPCSTR window );

// win_get_info 函数检索指定属性的值
// 并将其存储在 out_value 中。
int win_get_info ( LPCSTR window, LPCSTR property, char *out_value );

// win_get_text 函数从指定窗口或
// 窗口区域读取文本。
```

```
int win_get_text ( LPCSTR window, LPCSTR out_text );

// win_max 函数将指定窗口
// 最大化以充满整个屏幕。
int win_max (LPCSTR window );

// win_min 函数将指定窗口最小化为图标。
int win_min (LPCSTR window );

// win_mouse_click 函数在选中窗口
// 的指定坐标处执行鼠标单击操作。
int win_mouse_click (LPCSTR window, int x, int y, ButtonT button );

// win_mouse_dbl_click 函数在选中窗口
// 的指定坐标处执行鼠标双击操作。
int win_mouse_dbl_click (LPCSTR window, int x, int y, ButtonT button );

// win_mouse_drag 函数在窗口内执行
// 鼠标拖动操作。注意，指定的坐标是
// 相对于窗口（而非屏幕）的左上角。
int win_mouse_drag (LPCSTR window, int start_x, int start_y, int end_x, int end_y, ButtonT
button );

// win_move 函数将窗口移动到新的绝对位置。
int win_move ( LPCSTR window, int x, int y );

// win_resize 函数更改窗口的位置。
int win_resize ( LPCSTR window, int width, int height );

// win_restore 函数将窗口从图标化
// 或最大化状态还原为其原始大小。
int win_restore (LPCSTR window );

// win_wait_info 函数等待窗口
// 属性达到指定值，然后继续测试
// 运行。如果未达到指定值，则函数
// 将一直等到时间到期，然后再继
// 续测试。
int win_wait_info (LPCSTR window, LPCSTR property, LPCSTR value, UINT time );

// win_type 函数指定 keyboard_input
// 将发送到的目标窗口。
int win_type (LPCSTR window, LPCSTR keyboard_input );
```

```
// ctrx_<obj>_get_info 函数系列将属性
// 的值分配给值缓冲区。ctrx_obj_get_info
// 是一般函数，它可以适用于任何由录制器
// 所标识为对象的对象。
int ctrx_obj_get_info( const char * window_name, long xpos, long ypos, eObjAttribute attribute,
char *value, CTRX_LAST );

// ctrx_<obj>_get_info 函数系列将属性的值
// 分配给值缓冲区。ctrx_button_get_info
// 获取命令按钮的信息。
int ctrx_button_get_info( const char * window_name, long xpos, long ypos, eObjAttribute
attribute, char *value, CTRX_LAST );

// ctrx_<obj>_get_info 函数系列将属性的值
// 分配给值缓冲区。ctrx_edit_get_info
// 获取文本框的信息。
int ctrx_edit_get_info( const char * window_name, long xpos, long ypos, eObjAttribute attribute,
char *value, CTRX_LAST );

// ctrx_<obj>_get_info 函数系列将属性的值
// 分配给值缓冲区。ctrx_list_get_info
// 获取列表框的信息。
int ctrx_list_get_info( const char * window_name, long xpos, long ypos, eObjAttribute attribute,
char *value, CTRX_LAST );

// ctrx_connect_server 将 Citrix 客户端连接到 Citrix 服务器。
int ctrx_connect_server (char * server_name, char * user_name, char * password, char * domain);

// ctrx_disconnect_server 断开客户端与 Citrix 服务器的连接。
int ctrx_disconnect_server (char * server_name);

// ctrx_nfuse_connect 使用 NFUSE 应用
// 程序门户建立与 Citrix 服务器的连接。在
// 定义 NFUSE 门户的个性化规范的 ICA 文件
// 中找到的规范将从服务器上下载，在此之后
// 建立连接。
int ctrx_nfuse_connect(char * url);

// 使用 ctrx_get_bitmap_value
// 检索位图的哈希字符串值以用于您
// 的自定义同步函数中。位图坐标由
// 前四个参数指定。
int ctrx_get_bitmap_value (long x_start, long y_start, long width, long height, char * buffer);
```

```
// ctrx_get_text 将矩形中的文本分配到 text_buffer
// 中。随后，文本可被用于关联。
int ctrx_get_text( char *window_name, long xpos, long ypos, long width, long height, char *
filename, char * text_buffer, CTRX_LAST );

// ctrx_get_text_location 在 xpos、
// ypos、width 和 height 指定区域中
// 搜索指定文本。如果找到字符串，当函数
// 返回后，xpos 和 ypos 即为找到文本的
// 位置。如果未找到字符串，xpos 和 ypos
// 则为零。
int ctrx_get_text_location( LPCSTR window_name, long *xpos, long *ypos, long *width, long
*height, LPSTR text, long bMatchWholeWordOnly, LPCSTR filename, CTRX_LAST );

// ctrx_get_waiting_time 从运行时设置中获取当前等待
// 时间，或者通过 ctrx_set_waiting_time 设置的值。
int ctrx_get_waiting_time ( long * time );

// 使用 ctrx_get_window_name 检索
// 当前获得焦点的窗口的名称。
int ctrx_get_window_name (LPSTR buffer);

// 使用 ctrx_get_window_position
// 检索名为 title 变量值的窗口的位置。
// 如果 title 为 NULL，则函数将检索
// 当前拥有焦点的窗口的位置。
int ctrx_get_window_position (LPSTR title, long *xpos, long *ypos, long *width, long *height);

// ctrx_list_select_item 函数从列表中选择项目。
// 它支持 ListBox 或 ComboBox 类的列表。
int ctrx_list_select_item(char * window_name, long xpos, long ypos, char * item, CTRX_LAST );

// ctrx_menu_select_item 突出显示
// 菜单中的项目，但不激活它。
int ctrx_menu_select_item ( char *    window_name, char *    menu_path, CTRX_LAST) ;

// ctrx_mouse_click 等待窗口 window_name
// 出现，然后执行鼠标单击操作。
int ctrx_mouse_click (long x_pos, long y_pos, long mouse_button, long key_modifier, char *
window_name);

// ctrx_obj_mouse_click 等待窗口 window_name
// 出现，然后执行鼠标单击操作。
int ctrx_obj_mouse_click (const char * obj_desc, long x_pos, long y_pos, long mouse_button,
```

long key_modifier, char * window_name);

// ctrx_mouse_double_click 等待窗口 window_name
// 出现，然后执行鼠标双击操作。
int ctrx_mouse_double_click (long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_obj_mouse_double_click 等待窗口 window_name
// 出现，然后执行鼠标双击操作。
int ctrx_obj_mouse_double_click (const char * obj_desc, long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_mouse_down 等待窗口 window_name
// 出现，然后执行按下鼠标按键操作。
int ctrx_mouse_down(long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_obj_mouse_down 等待窗口 window_name
// 出现，然后执行按下鼠标按键操作。
int ctrx_obj_mouse_down(const char * obj_desc, long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_mouse_up 等待窗口 window_name
// 出现，然后在指定位置执行释放鼠标按键操作。
int ctrx_mouse_up(long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_obj_mouse_up 等待窗口 window_name
// 出现，然后在指定位置执行释放鼠标按键操作。
int ctrx_obj_mouse_up(const char * obj_desc, long x_pos, long y_pos, long mouse_button, long key_modifier, char * window_name);

// ctrx_set_window 是同步函数，它等待
// 窗口出现，然后 Vuser 才在该窗口中模拟
// 任何键盘或鼠标活动。
int ctrx_set_window (char * window_name);

// ctrx_set_window_ex 是同步函数，它至多
// 等待 time 秒，若窗口出现，Vuser 将在该
// 窗口中模拟任何键盘或鼠标活动。
int ctrx_set_window_ex (char * window_name, long time);

// ctrx_key 模拟用户在 Citrix 客户端中
// 按下非字母数字键。

int ctrx_key (char * key, long int key_modifier);

// 函数 ctrx_type 模拟用户键入字母数字字符。
int ctrx_type (char * data);

// ctrx_save_bitmap 将位图保存为文件。
// 该文件将保存在 Vuser 结果日志目录中。
int ctrx_save_bitmap(long x_start, long y_start, long width, long height, const char * file_name);

// ctrx_set_connect_opt 在建立 Citrix 客户端
// 与 Citrix 服务器的连接之前设置连接选项，然后
// 执行与服务器的连接。
int ctrx_set_connect_opt (eConnectionOption option, char * value);

// ctrx_set_exception 定义当不规则
// 事件发生时要执行的操作。此事件必须
// 与名为 window_title 的窗口（通常
// 为弹出式对话框）的外观相关联。当窗口
// 出现时，将调用 handler 函数。
void ctrx_set_exception (char * window_title, long handler, [void *context]);

// ctrx_set_waiting_time 更改
// 同步函数默认 60 秒的等待时间。
int ctrx_set_waiting_time (long time);

// ctrx_sync_on_bitmap 是同步函数，
// 它等待指定位图出现，然后再继续执行。
int ctrx_sync_on_bitmap (long x_start, long y_start, long width, long height, char * hash);

// ctrx_sync_on_bitmap_change 是同步
// 函数，它等待指定位图改变，然后再继续执行。
// 该函数通常用在窗口改变而窗口名称保持
// 不变的情况下。如果窗口名称改变，
// 则 ctrx_set_window 将被自动
// 生成。
int ctrx_sync_on_bitmap_change (long x_start, long y_start, long width, long height, <extra_args>, CTRX_LAST);

// ctrx_sync_on_obj_info 被调用时，执行将暂停，
// 直到指定对象的属性具有指定的值。
int ctrx_sync_on_obj_info (char * window_name, long xpos, long ypos, eObjAttribute attribute, char * value, <CTRX_LAST>);

// ctrx_sync_on_window 是同步函数，

```
// 它等待窗口被创建或变为活动的。
int ctrx_sync_on_window (char * window_name, eWindowEvent event, long x_start, long y_start,
long width, long height, char * filename, <CTRX_LAST>);

// ctrx_unset_window 是同步函数，它等待
// 窗口被关闭，然后脚本才继续执行。
int ctrx_unset_window (char * window_name);

// ctrx_wait_for_event 是同步函数，
// 它等待事件发生。
int ctrx_wait_for_event (char * event);

// 如果窗口存在，ctrx_win_exist 返回 E_OK（零）。
// 在 window_name 中可以使用通配符 (*)。
int ctrx_win_exist (char * window_name, long waiting_time);

// 有关 memchr 的详细信息，请参考 C 语言文档。
void *memchr ( const void *s, int c, size_t n );

// 有关 memcmp 的详细信息，请参考 C 语言文档。
int memcmp ( const void *s1, const void *s2, size_t n );

// memcpy 函数从 src 缓冲区中将 n 个
// 字符复制到 dest 缓冲区。
void *memcpy ( void *dest, const void *src, size_t n );

// 函数 memmove（以及所有不返回 integer 类型
// 的函数）必须明确在 Vugen 脚本中声明。
void *memmove ( void *dest, const void *src, size_t n );

// 有关 memset 的详细信息，请参考 C 语言文档。
void *memset ( void *buffer, int c, size_t n );

// 有关 getenv 的详细信息，请参考 C 语言文档。
char *getenv ( const char *varname );

// 有关 putenv 的详细信息，请参考 C 语言文档。
int putenv ( const char *envstring );

// 有关 system 的详细信息，请参考 C 语言文档。
int system ( const char *string );

// 有关 calloc 的详细信息，请参考 C 语言文档。
void *calloc ( size_t num elems, size_t elem_size );
```

```
// 有关 free 的详细信息，请参考 C 语言文档。
void free ( void *mem_address );

// 有关 malloc 的详细信息，请参考 C 语言文档。
void *malloc ( size_t num_bytes );

// 有关 realloc 的详细信息，请参考 C 语言文档。
void *realloc ( void *mem_address, size_t size );

// 有关 abs 的详细信息，请参考 C 语言文档。
int abs ( int n );

// 限制：cos 函数在 AIX 平台中无法使用。
double cos ( double x );

// 函数 floor（以及所有不返回 int 值的函数）
// 必须明确在 Vugen 脚本中声明。限制：此
// 函数在 AIX 平台中无法使用。
double floor ( double x );

// 在调用 rand 前，请调用 srand 以
// 播种伪随机数生成器
int rand ( void );

// 限制：sin 函数在 AIX 平台中无法使用。
double sin ( double x );

// 函数 sqrt（以及所有不返回 int 值的
// 函数）必须明确在 Vugen 脚本中声明。
// 限制：sqrt 函数在 AIX 平台中
// 无法使用。
double sqrt ( double x );

// 在调用 rand 前，请调用 srand 以
// 播种伪随机数生成器
int srand ( time );

// 有关 fclose 的详细信息，请参考 C 语言文档。
int fclose ( FILE *file_pointer );

// 请不要在脚本中包括操作系统头文件（例如，stdio.h）。
// 但是，这样将导致某些类型（包括 feof 使用的 FILE
// 类型）未经定义。这时，请使用 long 替代 FILE
```

```
// 类型。
int feof ( FILE *file_pointer );

// 有关 ferror 的详细信息，请参考 C 语言文档。
int ferror ( FILE *file_pointer );

// 请不要在脚本中包括操作系统头文件（例如，
// stdio.h）。但是，这样会导致某些类型
// （包括 fgetc 使用的 FILE 类型）未经定义。
// 这时，请使用 long 替代 FILE 类型。
int fgetc ( FILE *file_pointer );

// 请不要在脚本中包括操作系统头文件（例如，
// stdio.h）。但是，这样会导致某些类型
// （包括 fgets 使用的 FILE 类型）未经定义。
// 这时，请使用 long 替代 FILE 类型。
char *fgets ( char *string, int maxchar, FILE *file_pointer );

// 通过将 t 或 b 字符添加到 fopen 的 access_mode
// 参数，此访问模式字符串还将用于指定打开文件的方式
// （文本还是二进制）。
FILE *fopen ( const char *filename, const char *access_mode );

// 有关 fprintf 的详细信息，请参考 C 语言文档。
int fprintf ( FILE *file_pointer, const char *format_string [, args ] );

// 请不要在脚本中包括操作系统头文件（例如，
// stdio.h）。但是，这样会导致某些类型
// （包括 fputc 使用的 FILE 类型）未经定义。
// 这时，请使用 long 替代 FILE 类型。
int fputc ( int c, FILE *file_pointer );

// 请不要在脚本中包括操作系统头文件（例如，
// stdio.h）。但是，这样会导致某些类型
// （包括 fread 使用的 FILE 类型）未经定义。
// 这时，请使用 long 替代 FILE 类型。
size_t fread ( void *buffer, size_t size, size_t count, FILE *file_pointer );

// 有关 fscanf 的详细信息，请参考 C 语言文档。
int fscanf ( FILE *file_pointer, const char *format string [, args] );

// 有关 fseek 的详细信息，请参考 C 语言文档。
int fseek ( FILE *file_pointer, long offset, int origin );
```

```
// 请不要在脚本中包括操作系统头文件（例如，
// stdio.h）。但是，这样会导致某些类型
// （包括 fwrite 使用的 FILE 类型）未经定义。
// 这时，请使用 long 替代 FILE 类型。
size_t fwrite ( const void *buffer, size_t size, size_t count, FILE *file_pointer );

// 有关 rewind 的详细信息，请参考 C 语言文档。
void rewind ( FILE *file_pointer );

// 有关 sprintf 的详细信息，请参考 C 语言文档。
int sprintf ( char *string, const char *format_string[, args] );

// 有关 sscanf 的详细信息，请参考 C 语言文档。
int sscanf ( const char *buffer, const char *format_string, args );

// 有关 chdir 的详细信息，请参考 C 语言文档。
int chdir ( const char *path );

// chdrive 将当前工作驱动器更改为 drive（表示新驱动器
// 的 integer 类型值）。例如，1 = A、2 = B，依此类推。
int chdrive ( int drive );

// 有关 getcwd 的详细信息，请参考 C 语言文档。
char *getcwd ( char *path, int numchars );

// getdrive 函数返回表示驱动器字母
// 的 integer 类型值：1 = A、2 = B，依此类推。
int getdrive ( void );

// 有关 mkdir 的详细信息，请参考 C 语言文档。
int mkdir ( const char *path );

// 有关 remove 的详细信息，请参考 C 语言文档。
int remove ( const char *path );

// 有关 rmdir 的详细信息，请参考 C 语言文档。
int rmdir ( const char *path );

// 根据系统时钟，time 函数返回从世界标准
// 时间 1970 年 1 月 1 日子夜 (00:00:00)
// 作为开始所经过的秒数。返回值存储在 timeptr
// 所给出的位置。如果 timeptr 为 NULL，则
// 该值不会被存储。
time_t time ( time_t *timeptr );
```

```
// 在 Unix 下，ctime 不是线程级安全函数。所以，请使用 ctime_r。
// 有关详细信息，请参阅平台相关文档。
char *ctime ( const time_t *time );

// 有关 ftime 的详细信息，请参考 C 语言文档。
void ftime ( struct _timeb *time1 );

// 在 Unix 下，localtime 不是线程级安全函数。
// 所以，请使用 localtime_r。有关详细信息，请
// 参阅平台相关文档。
struct tm *localtime ( const time_t *time );

// 在 Unix 下，gmtime 不是线程级安全函数。所以，请使用 gmtime_r。
// 有关详细信息，请参阅平台相关文档。
struct tm *gmtime ( const time_t *time );

// 在 Unix 下，asctime 不是线程级安全函数。所以，请使用 asctime_r。
// 有关详细信息，请参阅平台相关文档。
char *asctime ( const struct tm *time );

// 通过停止在第一个非数字字符上，atof 只
// 读取字符串的初始位置。函数 atof（以及
// 所有不返回 integer 类型值的函数）
// 必须明确在 Vugen 脚本中声明。
double atof ( const char *string );

// 通过停止在第一个非数字字符上，atoi 只
// 读取字符串的初始位置。
int atoi ( const char *string );

// 通过停止在第一个非数字字符上，atol 只
// 读取字符串的初始位置。函数 atol（以及
// 所有不返回 integer 类型值的函数）必须
// 明确在 Vugen 脚本中声明。
long atol ( const char *string );

// itoa 将 value 的数字转换为以 radix 作为
// 基数的字符串 str。通常，radix 是 10。
int itoa ( int value, char *str, int radix );

// 通过停止在第一个非数字字符上，strtol
// 只扫描字符串的初始位置。所有前置空格
// 都将被去除。endptr 指向停止扫描的字符。
```

```
// 函数 strtol（以及所有不返回 integer
// 类型值的函数）必须明确在 Vugen 脚本中
// 声明。
long strtol ( const char *string, char **endptr, int radix );

// 有关 tolower 的详细信息，请参考 C 语言文档。
int tolower (int c);

// 有关 toupper 的详细信息，请参考 C 语言文档。
int toupper ( int c );

// 有关 isdigit 的详细信息，请参考 C 语言文档。
int isdigit ( int c );

// 函数 isalpha 检查 c 的值是否
// 处于 A - Z 或 a - z 的范围之内。
int isalpha ( int c );

// strcat 连接两个字符串。
char *strcat ( char *to, const char *from );

// strchr 返回指向字符串中
// 第一个匹配字符的指针。
char *strchr ( const char *string, int c );

// strcmp 比较 string1 和 string2 以确定字母排序的次序。
int strcmp ( const char *string1, const char *string2 );

// strcpy 将一个字符串复制给另一个。
char *strcpy ( char *dest, const char *source );

// strdup 复制字符串的副本。
char *strdup ( const char *string );

// stricmp 对两个字符串进行不区分大小写的比较。
int stricmp ( const char *string1, const char *string2 );

// strlen 返回字符串的长度（以字节为单位）。
size_t strlen ( const char *string );

// strlwr 将字符串转换为小写。
char *strlwr ( char *string );

// strncat 将一个字符串的 n 个字符连接到另一字符串。
```

```
char *strncat ( char *to_string, const char *from_string, size_t n );

// strncmp 比较两个字符串的前 n 个字符。
int strncmp ( const char *string1, const char *string2, size_t n );

// strncpy 将一个字符串的前 n 个字符复制到另一字符串。
char *strncpy ( char *dest, const char *source, size_t n );

// strnicmp 对两个字符串的 n 个
// 字符进行不区分大小写的比较，以
// 确定其字母排序的次序。
int strnicmp ( const char *string1, const char *string2, size_t num);

// strrchr 查找一个字符串中的最后一个匹配字符。
char *strrchr ( const char *string, int c );

// strset 使用指定字符填充字符串。
char *strset( char *string1, int character );

// strspn 返回指定字符串中包含另一
// 字符串起始字符的长度。
size_t *strspn ( const char *string, const char *skipset );

// strstr 返回一个字符串在另一字符串中第一次发生匹配的指针。
char *strstr ( const char *string1, const char *string2 );

// strtok 从由指定的字符分隔的字符串中返回标记。
// 注意，在 Vugen 文件中，需要明确声明不返回
// integer 类型值的 C 函数。
char *strtok ( char *string, const char *delimiters );

// strupr 将字符串转换为大写。
char *strupr ( char *string );

// lrc_CoCreateInstance 函数在本地系统或为特定
// 对象创建的默认主机中创建该对象的单个未初始化实例
// 并返回未知接口，该接口可用于获取其他接口。创建
// 该实例后，VuGen 调用 lrc_CoGetClassObject
// 以检索接口。如果 COM 对象位于远程计算机中，
// 将使用 lrc_CreateInstanceEx 取代
//  lrc_CoCreateInstance。
HRESULT lrc_CoCreateInstance(GUID * pClsid, IUnknown * pUnkOuter, unsigned long
dwClsContext, GUID * riid, LPVOID * ppv, BOOL __CheckResult);
```

```
// lrc_CreateInstanceEx 函数在指定的
// 远程计算机上创建未初始化的对象，并且
// 可以返回任意数量的请求接口。
HRESULT lrc_CreateInstanceEx(char *clsidStr, Iunknown *pUnk, DWORD dwClsCtx, ...);

// lrc_CoGetClassObject 函数提取
// 指定类的类工厂。
void lrc_CoGetClassObject( REFCLSID rclsid, Long dwClsContext, COSERVERINFO *
pServerInfo, REFIID riid, LPVOID * ppv);

// lrc_GUID 函数返回命名对象（例如
// COM 接口）的 GUID。VuGen 使用它
// 检索标识符，该标识符用于检索接口
// 标识符和用于 COM 通信的 COM 对象
// 的 PROGID。
GUID lrc_GUID(const char *str);

// lrc_GUID_by_ref 函数返回指向命名对象
// （例如 COM 接口）的 GUID 的指针。VuGen
// 使用它检索标识符，该标识符用于检索接口
// 标识符和用于 COM 通信的 COM 对象的
// PROGID。
GUID* lrc_GUID_by_ref(const char *str);

// lrc_DispMethod 函数使用 IDispatch:Invoke
// 方法调用接口的方法。lrc_DispMethod 调用
// 将 wflags 设置为 DISPATCH_METHOD。
VARIANT lrc_DispMethod(IDispatch * pDispatch, char *idName, unsigned long locale, ...);

// lrc_DispMethod1 使用 IDispatch 接口调用
// （或获取）同名的方法（或属性）。lrc_DispMethod1
// 调用将 wflags 设置为 DISPATCH_METHOD 和
// DISPATCH_PROPERTYGET。它可以用在方法与属性
// 具有同一名称的情况下。
VARIANT lrc_DispMethod1(IDispatch * pDispatch, char *idName, unsigned long locale, ...);

// lrc_DispPropertyGet 调用使用 IDispatch 接口
// 获取属性并将 wflags 设置为 DISPATCH_PROPERTYGET。
VARIANT lrc_DispPropertyGet(IDispatch * pDispatch, char *idName, unsigned long locale, ...);

// lrc_DispPropertyPut 使用 IDispatch 接口设置属性。
// 该调用将设置 DISPATCH_PROPERTYPUT 标志。
void lrc_DispPropertyPut(IDispatch * pDispatch, char *idName, unsigned long locale, ...);
```

```
// lrc_DispPropertyPutRef 使用 IDispatch 接口根据
// 引用设置属性，并设置 DISPATCH_PROPERTYPUTREF 标志。
void lrc_DispPropertyPutRef(IDispatch * pDispatch, char *idName, unsigned long locale, ...);

// lrc_CreateVBCollection 函数创建填充安全
// 数组值的 Visual Basic (VB) Collection 对象，
// 并将集合接口指针返回到 pCollection 中。
// VB 集合是由 COM 实现为接口的变量
// SafeArray。
HRESULT lrc_CreateVBCollection(SAFEARRAY *items, _Collection** pCollection);

// lrc_CoObject_from_variant 函数从变量中
// 提取 IUnknown 接口类型指针。
IUnknown* lrc_CoObject_from_variant(VARIANT var);

// lrc_DispObject_from_variant 函数从变量中
// 提取 IDispatch 接口类型指针。
IDispatch* lrc_DispObject_from_variant(VARIANT var);

// lrc_CoObject_by_ref_from_variant 函数从指向
// 变量的指针中提取 IUnknown 接口类型指针。
IUnknown* lrc_CoObject_by_ref_from_variant(VARIANT var);

// lrc_DispObject_by_ref_from_variant 函数从指向
// 变量的指针中提取 IDispatch 接口类型指针。
IDispatch* lrc_DispObject_by_ref_from_variant(VARIANT var);

// 输入表示整数的字符串时，lrc_int 函数返回
// integer 类型值。此参数可以是文字字符串、
// 变量或参数。
int lrc_int(const char* str);

// 输入表示整数的字符串时，lrc_int_by_ref
// 函数返回指向 integer 类型值的指针。此参数
// 可以是文字字符串、变量或
// 参数。
int* lrc_int_by_ref(const char* str);

// lrc_save_int 函数将 integer 值
// 保存在指定变量 name 下的字符串中，以便
// 您将其用于参数化。VuGen 将此函数生成
// 为注释掉的调用。如果要将此值用作参数，
// 可以更改 name 参数并取消调用的
// 注释。
```

```
int lrc_save_int(const char* name, int val);

// lrc_save_int_by_ref 函数将 integer 值
// 保存在字符串中，并将 val 设置为指向该字符串。
// VuGen 将此函数生成为注释掉的调用。如果要
// 将此值用作参数，可以更改 name 并取消调用
// 的注释。
int lrc_save_int_by_ref(const char* name, int *val);

// lrc_get_bstr_length 返回 BSTR 类型字符
// 串的长度。BSTR 字符串可以包括 null。
int   lrc_get_bstr_length(BSTR str);

// lrc_get_bstr_sub 从输入字符串 str 的开始处
// 返回 size 个字符的子集。
BSTR lrc_get_bstr_sub(BSTR str, int size);

// lrc_print_bstr 将 BSTR 字符串输出到
// 用于调试目的的标准输出上。
int lrc_print_bstr(BSTR str);

// lrc_BSTR1 创建长度为 len 的 BSTR 字符串，它可以包括 null。
BSTR lrc_BSTR1 (const char* str, long len);

// lrc_save_BSTR1 函数将 BSTR str 保
// 存到字符串 name 中。
int lrc_save_BSTR1 (const char* name, BSTR str);

// 输入表示无符号整数的字符串时，lrc_uint 函数
// 返回无符号 integer 类型值。
unsigned int lrc_uint(const char* str);

// 输入表示无符号整数的字符串时，lrc_uint_by_ref
// 函数返回指向无符号 integer 类型值的
// 指针。
unsigned int* lrc_uint_by_ref(const char* str);

// lrc_save_uint 函数将无符号 integer 值
// 保存在指定变量 name 下的字符串中，以便
// 您将其用于参数化。VuGen 将此函数生
// 成为注释掉的调用。如果要将此值用作
// 参数，可以更改 name 参数并取消调用的
// 注释。
int lrc_save_uint(const char* name, unsigned int val);
```

```
// lrc_save_uint_by_ref 函数将
// 无符号 integer 类型值保存在字符串中，
// 并将 val 设置为指向该字符串。VuGen 将此
// 函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的注释。
int lrc_save_uint_by_ref(const char* name, unsigned int *val);

// 输入表示 long 类型的字符串时，lrc_long 函数
// 返回长整型值。
long lrc_long(const char* str);

// 输入表示 long 类型的字符串时，lrc_long_by_ref 函数
// 返回指向长整型值的指针。
long* lrc_long_by_ref(const char* str);

// lrc_save_long 函数将长整型值
// 保存在指定变量 name 下的字符串中，
// 以便您可以将其用于参数化。VuGen
// 将此函数生成为注释掉的调用。如果
// 要将此值用作参数，可以更改 name
// 并取消调用的注释。
int lrc_save_long(const char* name, long val);

// lrc_save_long_by_ref 函数将长整型值保存在字符串中，
// 并将 val 设置为指向该字符串。
int lrc_save_long_by_ref(const char* name, long *val);

// 输入表示无符号 long 类型值的字符串时，
// lrc_ulong 函数返回无符号长整型值。
unsigned long lrc_ulong(const char* str);

// 输入表示无符号 long 类型值的字符串时，
// lrc_ulong_by_ref 函数返回指向无符号长整型值
// 的指针。
unsigned long* lrc_ulong_by_ref(const char* str);

// lrc_save_ulong 函数将无符号
// 长整型值保存在指定变量 name 下
// 的字符串中，以便您可以将其用于参数化。
// VuGen 将此函数生成为注释掉的调用。
// 如果要将此值用作参数，可以更改 name
// 并取消调用的注释。
int lrc_save_ulong(const char* name, unsigned long val);
```

```
// lrc_save_ulong_by_ref 函数将无符号长整型值保存为字符串，
// 并将 val 设置为指向该字符串。
int lrc_save_ulong_by_ref(const char* name, unsigned long *val);

// 输入表示短整型值的字符串时，lrc_short 函数
// 返回短整型值。
short lrc_short(const char* str);

// 输入表示短整型值的字符串时，lrc_short_by_ref 函数
// 返回指向短整型值的
// 指针。
short* lrc_short_by_ref(const char* str);

// lrc_save_short 函数将短整型值保存
// 在指定变量 name 下的字符串中，以便
// 您可以将其用于参数化。VuGen 将此
// 函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_short(const char* name, short val);

// lrc_save_short_by_ref 函数将短整型值保存在字符串中，
// 并将 val 设置为指向该字符串。
int lrc_save_short_by_ref(const char* name, short *val);

// 输入表示货币值的字符串时，lrc_currency 函
// 数返回货币值。
CY lrc_currency(const char* str);

// 输入表示货币值的字符串时，
// lrc_currency_by_ref 函数返回指向
// 货币结构的指针。
CY* lrc_currency_by_ref(const char* str);

// lrc_save_currency 函数将货币 (CY) 值
// 保存在指定变量 name 下的字符串中，以便您
// 可以将其用于参数化。VuGen 将此函数
// 生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_currency(const char* name, CY val);

// lrc_save_currency_by_ref 函数将由“val”
```

```
// 指针引用的货币值保存到字符串参数中。
int lrc_save_currency_by_ref(const char* name, CY *val);

// 输入表示 date 类型值的字符串时，
// lrc_date 函数返回 DATE 类型值。
DATE lrc_date(const char* str);

// 输入表示 date 类型值的字符串时，lrc_date_by_ref 函数
// 返回指向 DATE 的指针。
DATE* lrc_date_by_ref(const char* str);

// lrc_save_date 函数将 date 类型值保存
// 为字符串。VuGen 将此函数生成为注释掉的调用。
// 如果要将此值用作参数，可以更改 name 并取消
// 调用的注释。
int lrc_save_date(const char* name, DATE val);

// lrc_save_date_by_ref 函数将 date 类型值保存为字符串。
int lrc_save_date_by_ref(const char* name, DATE *val);

// 输入包含“true”或“false”的字符串时，
// lrc_bool 函数返回 Boolean 类型值。
VARIANT_BOOL lrc_bool(const char* str);

// 输入包含“true”或“false”的字符串时，
// lrc_bool_by_ref 函数返回指向 Boolean
// 类型值的指针。
VARIANT_BOOL* lrc_bool_by_ref(const char* str);

// lrc_save_bool 函数将 Boolean 类型值
// 保存为字符串参数。VuGen 将此函数生成为
// 注释掉的调用。如果要将此值用作参数，可以
// 更改 name 并取消调用的注释。
int lrc_save_bool(const char* name, VARIANT_BOOL val);

// lrc_save_bool_by_ref 函数将 Boolean 类型值
// 保存到字符串参数中。
int lrc_save_bool_by_ref(const char* name, VARIANT_BOOL *val);

// 输入表示无符号短整型值的字符串时，
// lrc_ushort 函数返回无符号
// 短整型值。
unsigned short lrc_ushort(const char* str);
```

```
// 输入表示无符号短整型值的字符串时，
// lrc_ushort_by_ref 函数返回指向无符号短整型值
// 的指针。
unsigned short* lrc_ushort_by_ref(const char* str);

// lrc_save_ushort 函数将无符号
// 短整型值保存在参数中。
int lrc_save_ushort(const char* name, unsigned short val);

// lrc_save_ushort_by_ref 函数将无符号短整型值
// 保存到参数中。
int lrc_save_ushort_by_ref(const char* name, unsigned short *val);

// 输入包含浮点数的字符串时，
// lrc_float 函数返回浮点数。
float lrc_float(const char* str);

// 输入包含浮点数的字符串时，
// lrc_float_by_ref 函数返回指向浮点数
// 的指针。
float* lrc_float_by_ref(const char* str);

// lrc_save_float 函数将浮点类型
// 浮点值保存在字符串参数中。VuGen 将此
// 函数生成为注释掉的调用。如果要使用
// 该参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_float(const char* name, float val);

// lrc_save_float_by_ref 函数将浮点
// 类型浮点值保存在字符串参数中。
int lrc_save_float_by_ref(const char* name, float *val);

// 输入包含 double 类型值的字符串时，
// lrc_double 函数返回 double 类型值。
double lrc_double(const char* str);

// 输入包含 double 类型值的字符串时，lrc_double_by_ref
// 函数返回指向 double 类型值的指针。
double* lrc_double_by_ref(const char* str);

// lrc_save_double 函数将双精度浮点
// 类型值保存在字符串参数中。VuGen 将
// 此函数生成为注释掉的调用。如果要将
```

```
// 此值用作参数，可以更改 name 并取消
// 调用的注释。
int lrc_save_double(const char* name, double val);

// lrc_save_double_by_ref 函数将双精度浮点
// 类型值保存在字符串参数中。
int lrc_save_double_by_ref(const char* name, double *val);

// 输入包含 dword 类型值的字符串时，lrc_dword 函数
// 返回双字类型值。
DWORD lrc_dword(const char* str);

// lrc_save_dword 函数将双字类型值
// 保存在字符串参数中。VuGen 将此函数
// 生成为注释掉的调用。如果要将此值用作
// 参数，可以更改 name 并取消调用的注释。
int lrc_save_dword(const char* name, DWORD val);

// lrc_BSTR 函数将任何字符串转换为 BSTR。
BSTR lrc_BSTR(const char* str);

// lrc_save_BSTR 函数将 BSTR 值保存
// 为字符串参数。VuGen 将此函数生成为
// 注释掉的调用。如果要将此值用作参数，
// 可以更改 name 并取消调用的注释。
int lrc_save_BSTR(const char* name, BSTR val);

// lrc_ascii_BSTR 函数将字符串
// 转换为 ascii_BSTR。
BSTR lrc_ascii_BSTR(const char* str);

// lrc_ascii_BSTR_by_ref 函数将字符串转换
// 为 ascii_BSTR，并返回指向该 BSTR 的指针。
BSTR* lrc_ascii_BSTR_by_ref(const char* str);

// 当不再使用 COM 对象时，
// lrc_Release_Object 函数释放该对象。
// 释放对象后，对象的引用计数将减 1
// （例如，IUnknown_1 到 IUnknown_0）。
//
int lrc_Release_Object(const char* interface);

// lrc_save_ascii_BSTR 函数将 ascii BSTR 保存
// 到字符串参数中。VuGen 将此函数生成为注释掉的调用。
```

```
// 如果要将此值用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_ascii_BSTR(const char* name, BSTR val);

// lrc_save_ascii_BSTR_by_ref 函数将 ascii BSTR 保
// 存到字符串参数中。
int lrc_save_ascii_BSTR_by_ref(const char* name, BSTR *val);

// lrc_save_VARIANT 函数将任何数据类型的值
// 保存到字符串参数中。
int lrc_save_VARIANT(const char* name, VARIANT val);

// lrc_save_variant_<Type-Name> 函数系列
// 由 VuGen 生成，用于将指定的 <Type-Name>
// 变量保存为字符串参数。VuGen 将这些代码行
// 生成为注释掉的调用。如果要将此值用作参数，
// 可以更改 name 并取消调用的
// 注释。
int lrc_save_variant_<Type-Name>(const char* name, VARIANT val);

// lrc_save_variant_short 将 short 变量类型值保存到字符串参数中。
int lrc_save_variant_short(const char* name, VARIANT val);

// lrc_save_variant_ushort 将 short 变量类型值保存到字符串参数中。
int lrc_save_variant_ushort(const char* name, VARIANT val);

// lrc_save_variant_char 将 short 变量类型值保存到字符串参数中。
int lrc_save_variant_char(const char* name, VARIANT val);

// lrc_save_variant_int 将 int 变量类型值保存到字符串参数中。
int lrc_save_variant_int(const char* name, VARIANT val);

// lrc_save_variant_uint 将无符号 integer 变量
// 类型值到字符串参数中。
int lrc_save_variant_uint(const char* name, VARIANT val);

// lrc_save_variant_ulong 将无符号 long 变量
// 类型值保存到字符串参数中。
int lrc_save_variant_ulong(const char* name, VARIANT val);

// lrc_save_variant_BYTE 将 BYTE 变量类型值保存到字符串参数中。
int lrc_save_variant_BYTE(const char* name, VARIANT val);

// lrc_save_variant_long 将 long 变量类型值保存到字符串参数中。
```

```
int lrc_save_variant_long(const char* name, VARIANT val);

// lrc_save_variant_float 将 float 变量类型值保存到字符串参数中。
int lrc_save_variant_float(const char* name, VARIANT val);

// lrc_save_variant_double 将 double 变量
// 类型值保存到字符串参数中。
int lrc_save_variant_double(const char* name, VARIANT val);

// lrc_save_variant_bool 将 boolean 变量类型值保存到字符串参数中。
int lrc_save_variant_bool(const char* name, VARIANT val);

// lrc_save_variant_scode 将 scode 变量类型值保存到字符串参数中。
int lrc_save_variant_scode(const char* name, VARIANT val);

// lrc_save_variant_currency 将 currency 变量
// 类型值保存到字符串参数中。
int lrc_save_variant_currency(const char* name, VARIANT val);

// lrc_save_variant_date 将 DATE 变量类型值保存到字符串参数中。
int lrc_save_variant_date(const char* name, VARIANT val);

// lrc_save_variant_BSTR 将 BSTR 变量类型值保存到字符串参数中。
int lrc_save_variant_BSTR(const char* name, VARIANT val);

// lrc_save_variant_<Type-Name>_by_ref 函数
// 系列由 VuGen 生成，以便将通过变量中的引用方式
// 存储的、指定了 <Type-Name> 的变量保存为字符串
// 参数。VuGen 将这些代码行生成为注释掉的调用。
// 如果要将此值用作参数，可以更改 name 并取消调用的
// 注释。
//
int lrc_save_variant_<Type-Name>_by_ref(const char* name, VARIANT val);

// lrc_save_variant_short_by_ref 将通过变量中
// 的引用方式存储的值保存为参数。
int lrc_save_variant_short_by_ref(const char* name, VARIANT val);

// lrc_save_variant_ushort_by_ref 将
// 通过变量中的引用方式存储的值保存为参数。
int lrc_save_variant_ushort_by_ref(const char* name, VARIANT val);

// lrc_save_variant_char_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
```

```
int lrc_save_variant_char_by_ref(const char* name, VARIANT val);

// lrc_save_variant_int_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_int_by_ref(const char* name, VARIANT val);

// lrc_save_variant_uint_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_uint_by_ref(const char* name, VARIANT val);

// lrc_save_variant_ulong_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_ulong_by_ref(const char* name, VARIANT val);

// lrc_save_variant_BYTE_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_BYTE_by_ref(const char* name, VARIANT val);

// lrc_save_variant_long_by_ref 将
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_long_by_ref(const char* name, VARIANT val);

// lrc_save_variant_float_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_float_by_ref(const char* name, VARIANT val);

// lrc_save_variant_double_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_double_by_ref(const char* name, VARIANT val);

// lrc_save_variant_bool_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_bool_by_ref(const char* name, VARIANT val);

// lrc_save_variant_scode_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_scode_by_ref(const char* name, VARIANT val);

// lrc_save_variant_currency_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_currency_by_ref(const char* name, VARIANT val);

// lrc_save_variant_date_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
```

```
int lrc_save_variant_date_by_ref(const char* name, VARIANT val);

// lrc_save_variant_BSTR_by_ref 将通过
// 变量中的引用方式存储的值保存为参数。
int lrc_save_variant_BSTR_by_ref(const char* name, VARIANT val);

// lrc_BYTE 函数将字符串转换
// 为无符号字符（字节）数值。
BYTE lrc_BYTE(const char* str);

// lrc_BYTE_by_ref 函数将字符串转换
// 为无符号字符（字节）数值，并返回指向该字节
// 的指针。
char * lrc_BYTE_by_ref(const char* str);

// lrc_save_BYTE 函数将 byte 类型值保存为参数。
// VuGen 将此函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 param_name 并取消调用的
// 注释。
int lrc_save_BYTE(const char* param_name, BYTE val);

// lrc_save_BYTE_by_ref 函数将 byte
// 类型值保存为参数。VuGen 将此函数生成为
// 注释掉的调用。如果要将此值用作参数，可以
// 更改 param_name 并取消调用的注释。
int lrc_save_BYTE_by_ref(const char* param_name, BYTE *val);

// 输入表示超级整型值的字符串时，lrc_hyper 函
// 数返回超级（64 位）整型值。
hyper lrc_hyper(const char* str);

// 输入表示超级整型值的字符串时，
// lrc_hyper_by_ref 函数返回指向
// 超级（64 位）整型值的指针。
hyper* lrc_hyper_by_ref(const char* str);

// lrc_save_hyper 函数将 64 位超级
// 整型值保存在字符串参数中。VuGen 将
// 此函数生成为注释掉的调用。如果要将
// 此值用作参数，可以更改 name 并取消
// 调用的注释。
int lrc_save_hyper(const char* name, hyper val);

// lrc_save_hyper_by_ref 函数将 64 位
```

```
// 超级整型值保存到字符串参数中。
int lrc_save_hyper_by_ref(const char* name, hyper *val);

// 输入表示无符号超级整型值的字符串时，
// lrc_uhyper 函数返回无符号超级
// （64 位）整型值。
uhyper lrc_uhyper(const char* str);

// 输入表示无符号超级整型值的字符串时，
// lrc_uhyper_by_ref 函数返回指向
// 无符号超级（64 位）整型值。
uhyper* lrc_uhyper_by_ref(const char* str);

// lrc_save_uhyper 函数将无符号 64 位
// 超级整型值保存在字符串参数中。VuGen 将
// 此函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_uhyper(const char* name, uhyper val);

// lrc_save_uhyper_by_ref 函数将无符号
// 64 位超级整型值保存到字符串参数中。VuGen
// 将此函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_uhyper_by_ref(const char* name, uhyper *val);

// lrc_char 函数将包含 char 类型数值
// 的字符串转换为 char 变量。
char lrc_char(const char* str);

// lrc_char_by_ref 函数将包含 char 类型数值的
// 字符串转换为 char 变量。
char* lrc_char_by_ref(const char* str);

// lrc_save_char 函数将 char 类型
// (0 - 127) 保存到字符串参数中。
// VuGen 将此函数生成为注释掉的调用。
// 如果要将此值用作参数，可以更改 name
// 并取消调用的注释。
int lrc_save_char(const char* name, char val);

// lrc_save_char_by_ref 函数将 char 类型
// (0 - 127) 保存到字符串参数中。VuGen 将
```

```
// 此函数生成为注释掉的调用。如果要将此值
// 用作参数，可以更改 name 并取消调用的
// 注释。
int lrc_save_char_by_ref(const char* name, char *val);

// lrc_variant_empty 函数返回空变量。
VARIANT lrc_variant_empty();

// lrc_variant_empty_by_variant 返回
// 包含对空变量引用的变量。
VARIANT lrc_variant_empty_by_variant();

// lrc_variant_null 函数返回 null 变量。
VARIANT lrc_variant_null();

// lrc_variant_null_by_variant 返回
// 包含对 null 变量引用的变量。
VARIANT lrc_variant_null_by_variant();

// lrc_variant_short 函数将字符串转换为
// 短整型值，并在变量中将其返回。
VARIANT lrc_variant_short(const char* str);

// lrc_variant_short_by_variant 函数将字符串
// 转换为短整型值，并返回包含对某变量（其中包含
// 该短整型值）引用的变量。
VARIANT lrc_variant_short_by_variant(const char* str);

// lrc_variant_ushort 函数将字符串转换为
// 无符号短整型值，并在变量中将其返回。
VARIANT lrc_variant_ushort(const char* str);

// lrc_variant_char 函数将字符串转换为
// char 类型值，并在变量中将其返回。
VARIANT lrc_variant_char(const char* str);

// lrc_variant_int 函数将字符串转换为整型
// 值，并在变量中将其返回。
VARIANT lrc_variant_int(const char* str);

// lrc_variant_uint 函数将字符串转换为
// 无符号整型值，并将其存储在变量中返回。
VARIANT lrc_variant_uint(const char* str);
```

```
// lrc_variant_ulong 函数将字符串转换为
// 无符号长整型值，并将其存储在变量中返回。
VARIANT lrc_variant_ulong(const char* str);

// lrc_variant_BYTE 函数将字符串转换为
// 无符号 char (byte) 类型值并存储于变量中。
VARIANT lrc_variant_BYTE(const char* str);

// lrc_variant_BYTE_by_variant 函数将字符串
// 转换为无符号 char (byte) 类型值，并返回包含
// 对某变量（包含该 byte）引用的变量。
VARIANT lrc_variant_BYTE_by_variant(const char* str);

// lrc_variant_long 函数将字符串
// 转换为存储于变量中的长整型值。
VARIANT lrc_variant_long(const char* str);

// lrc_variant_long_by_variant 函数将
// 字符串转换为存储于变量中的长整型值，并且
// 返回包含对某变量（其中包含该长整型值）
// 引用的变量。
VARIANT lrc_variant_long_by_variant(const char* str);

// lrc_variant_float 函数将字符串转换为
// 存储于变量中的浮点类型值。
VARIANT lrc_variant_float(const char* str);

// lrc_variant_float_by_variant 函数将
// 字符串转换为浮点类型值，并且返回包含对某
// 变量（其中包含该值）引用的变量。
VARIANT lrc_variant_float_by_variant(const char* str);

// lrc_variant_double 函数将字符串转换为
// 存储于变量中的双精度型浮点值。
VARIANT lrc_variant_double(const char* str);

// lrc_variant_double_by_variant 函数将
// 字符串转换为双精度型值，并且返回包含对某变量
// （其中包含该值）引用的变量。
VARIANT lrc_variant_double_by_variant(const char* str);

// lrc_variant_bool 函数将包含“true”或“false”
// 的字符串转换为存储于变量中的 Boolean 类型值。
VARIANT lrc_variant_bool(const char* str);
```

```
// lrc_variant_bool_by_variant 函数将
// 包含“true”或“false”的字符串转换为
// Boolean 值，并且返回包含对某变量（其中
// 包含该值）引用的变量。
VARIANT lrc_variant_bool_by_variant(const char* str);

// lrc_variant_scode 函数将包含系统错误代码值的。
// 字符串转换为存储于变量中的错误代码
VARIANT lrc_variant_scode(const char* errcode);

// lrc_variant_scode_by_variant 函数将
// 包含系统错误代码值的字符串转换为错误代码，
// 并且返回包含对某变量（其中包含该值）引用
// 的变量。
VARIANT lrc_variant_scode_by_variant(const char* errcode);

// lrc_variant_currency 函数将包含货币值
// 的字符串转换为存储于变量中的货币值。
VARIANT lrc_variant_currency(const char* str);

// lrc_variant_currency_by_variant 函数
// 将包含货币值的字符串转换为货币值，并且
// 返回包含对某变量（其中包含该值）引用的
// 变量。
VARIANT lrc_variant_currency_by_variant(const char* str);

// lrc_variant_date 函数将包含 date 类型值的
// 字符串转换为存储于变量中的 date 类型值。
VARIANT lrc_variant_date(const char* str);

// lrc_variant_date_by_variant 函数将
// 包含 date 类型值的字符串转换为 date 类型值，
// 并且返回包含对某变量（其中包含该 date 类型值）
// 引用的变量。
VARIANT lrc_variant_date_by_variant(const char* str);

// lrc_variant_BSTR 函数将字符串转换为
// 存储于变量中的 BSTR 类型值。
VARIANT lrc_variant_BSTR(const char* str);

// lrc_variant_BSTR_by_variant 函数将字符串
// 转换为 BSTR 值，并且返回包含对某变量（其中
// 包含该值）引用的变量。
```

```
VARIANT lrc_variant_BSTR_by_variant(const char* str);

// lrc_variant_ascii_BSTR 函数将字符串分配给
// 存储于变量中的 ASCII BSTR 值
VARIANT lrc_variant_ascii_BSTR(const char* str);

// lrc_variant_CoObject 函数将 IUnknown 接口
// 指针分配给变量。
VARIANT lrc_variant_CoObject(IUnknown* pUnknown);

// lrc_variant_CoObject_by_variant 函数将
//  IUnknown 接口指针分配给变量，并且返回
// 包含对某变量（其中包含 IUnknown 引用）
// 引用的变量。
VARIANT lrc_variant_CoObject_by_variant(IUnknown* pUnknown);

// lrc_variant_DispObject 函数将
//  IDispatch 接口指针分配给变量。
VARIANT lrc_variant_DispObject(IDispatch* pDispatch);

// lrc_variant_DispObject_by_variant 函数
// 将 IDispatch 接口指针分配给变量，并且返回包含
// 对某变量（其中包含 IDispatch 引用）引用的
// 变量。
VARIANT lrc_variant_DispObject_by_variant(IDispatch* pDispatch);

// lrc_variant_short_by_ref 函数将字符串
// 分配给通过变量中的引用方式存储的短整型值。
VARIANT lrc_variant_short_by_ref(const char* str);

// lrc_variant_ushort_by_ref 函数将字符串
// 分配给通过变量中的引用方式存储的无符号短整型值。
VARIANT lrc_variant_ushort_by_ref(const char* str);

// lrc_variant_char_by_ref 函数将字符串分配
// 给通过变量中的引用方式存储的 char 类型值。
VARIANT lrc_variant_char_by_ref(const char* str);

// lrc_variant_int_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的 integer 类型值。
VARIANT lrc_variant_int_by_ref(const char* str);

// lrc_variant_uint_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的无符号 integer 类型值。
```

```
VARIANT lrc_variant_uint_by_ref(const char* str);

// lrc_variant_ulong_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的长整型值。
VARIANT lrc_variant_ulong_by_ref(const char* str);

// lrc_variant_BYTE_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的 char (byte) 类型值。
VARIANT lrc_variant_BYTE_by_ref(const char* str);

// lrc_variant_long_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的长整型值。
VARIANT lrc_variant_long_by_ref(const char* str);

// lrc_variant_float_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的浮点型浮点值。
VARIANT lrc_variant_float_by_ref(const char* str);

// lrc_variant_double_by_ref 函数将字符串分配给
// 通过变量中的引用方式存储的双精度型值。
VARIANT lrc_variant_double_by_ref(const char* str);

// lrc_variant_bool_by_ref 函数将包含“true”
// 或“false”的字符串分配给通过变量中的
// 引用方式存储的 Boolean 类型值。
VARIANT lrc_variant_bool_by_ref(const char* str);

// lrc_variant_scode_by_ref 函数将包含
// 系统错误代码值的字符串分配给通过变量中的
// 引用方式存储的错误代码中。
VARIANT lrc_variant_scode_by_ref(const char* str);

// lrc_variant_currency_by_ref 函数将字符串
// 分配给通过变量中的引用方式存储的货币类型值。
VARIANT lrc_variant_currency_by_ref(const char* str);

// lrc_variant_date_by_ref 函数将字符串
// 分配给通过变量中的引用方式存储的 date 类型值。
VARIANT lrc_variant_date_by_ref(const char* str);

// lrc_variant_BSTR_by_ref 函数将字符串分配给的
// 通过变量中引用方式存储的 BSTR 值。
VARIANT lrc_variant_BSTR_by_ref(const char* str);
```

```
// lrc_variant_ascii_BSTR_by_ref 函数将字符串
// 分配给通过变量中的引用方式存储的 ascii BSTR 值。
VARIANT lrc_variant_ascii_BSTR_by_ref(const char* str);

// lrc_variant_CoObject_by_ref 函数将指针
// 分配给变量中的 IUnknown 接口。
VARIANT lrc_variant_CoObject_by_ref(IUnknown* pUnknown);

// lrc_variant_DispObject_by_ref 函数将指针
// 转换为变量中的 IDispatch 接口。
VARIANT lrc_variant_DispObject_by_ref(IDispatch* pDispatch);

// lrc_variant_variant_by_ref 函数创建包含现有
// 变量的新变量。
VARIANT lrc_variant_variant_by_ref(VARIANT * pVar);

// lrc_variant_from_variant_by_ref 函数从
// 另一个变量引用中获取变量。
VARIANT lrc_variant_from_variant_by_ref(VARIANT var);

// Create<n>D<Type-Name>Array 函数系列
// 创建由 Type-name 指定的类型的 n 维数组。
// 对于每一维，该调用必须指定 lower_bound
// 和 upper_bound。
<Type-Name> Array Create<n>D<Type-Name>Array(int lower_bound,int upper_bound, int
lower_bound, int upper_bound...);

// Destroy<Type-Name>Array 函数系列销毁
// 由 Type-name 指定的类型的数组。对于在
// 脚本中（而非由 VuGen）创建的数组，可以使用
// 它们恢复内存。
void Destroy<Type-Name>Array(<Type-Name>Array Array);

// DestroyBoolArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyBoolArray(BoolArray Array);

// DestroyBstrArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyBstrArray(BstrArray Array);

// DestroyByteArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyByteArray(ByteArray Array);
```

```
// DestroyCharArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyCharArray(CharArray Array);

// DestroyCoObjectArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyCoObjectArray(CoObjectArray Array);

// DestroyCurrencyArray 释放由数组占用的内存。于
// 该函数用恢复由脚本为数组分配的内存。
void DestroyCurrencyArray(CurrencyArray Array);

// DestroyDateArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyDateArray(CurrencyArray Array);

// DestroyDispObjectArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyDispObjectArray(DispObjectArray Array);

// DestroyDoubleArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyDoubleArray(DoubleArray Array);

// DestroyErrorArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyErrorArray(ErrorArray Array);

// DestroyFloatArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyFloatArray(FloatArray Array);

// DestroyIntArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyIntArray(IntArray Array);

// DestroyLongArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyLongArray(LongArray Array);

// DestroyShortArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyShortArray(ShortArray Array);
```

```
// DestroyUIntArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyUIntArray(UIntArray Array);

// DestroyULongArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyULongArray(ULongArray Array);

// DestroyUShortArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyUShortArray(UShortArray Array);

// DestroyVariantArray 释放由数组占用的内存。
// 该函数用于恢复由脚本为数组分配的内存。
void DestroyVariantArray(VariantArray Array);

// GetBufferFrom<n>DByteArray 函数系列从 n 维
// byte 类型数组的最后一维中提取缓冲区。此函数
// 的返回值是一个指向 byte 类型的缓冲区的指针。
// 缓冲区的大小返回到 *plinesize 所指定的
// 地址中。
char * GetBufferFrom<n>DByteArray(ByteArray Array, int ind1...indn-1, size_t *pLineSize);

// Fill<n>DByteArray 函数系列将 n 维 byte 数组
// 的最后一维中填充缓冲区的内容。
char * Fill<n>DByteArray(ByteArray Array, int ind1...indn-1, const char* buffer, size_t
buff_size);

// lrc_variant_<Type-Name>Array 函数系列将在
// Type-Name 中指定的类型的数组分配给变量。
VARIANT lrc_variant_<Type-Name>Array(<Type-Name>Array array);

// lrc_variant_BoolArray 将数组分配给变量。
VARIANT lrc_variant_BoolArray(BoolArray array);

// lrc_variant_BstrArray 将数组分配给变量。
VARIANT lrc_variant_BstrArray(BstrArray array);

// lrc_variant_ByteArray 将数组分配给变量。
VARIANT lrc_variant_ByteArray(ByteArray array);

// lrc_variant_CharArray 将数组分配给变量。
VARIANT lrc_variant_CharArray(CharArray array);
```

```
// lrc_variant_CoObjectArray 将数组分配给变量。
VARIANT lrc_variant_CoObjectArray(CoObjectArray array);

// lrc_variant_CurrencyArray 将数组分配给变量。
VARIANT lrc_variant_CurrencyArray(CurrencyArray array);

// lrc_variant_DateArray 将数组分配给变量。
VARIANT lrc_variant_DateArray(DateArray array);

// lrc_variant_DispObjectArray 将数组分配给变量。
VARIANT lrc_variant_DispObjectArray(DispObjectArray array);

// lrc_variant_DoubleArray 将数组分配给变量。
VARIANT lrc_variant_DoubleArray(DoubleArray array);

// lrc_variant_ErrorArray 将数组分配给变量。
VARIANT lrc_variant_ErrorArray(ErrorArray array);

// lrc_variant_FloatArray 将数组分配给变量。
VARIANT lrc_variant_FloatArray(FloatArray array);

// lrc_variant_IntArray 将数组分配给变量。
VARIANT lrc_variant_IntArray(IntArray array);

// lrc_variant_LongArray 将数组分配给变量。
VARIANT lrc_variant_LongArray(LongArray array);

// lrc_variant_ShortArray 将数组分配给变量。
VARIANT lrc_variant_ShortArray(ShortArray array);

// lrc_variant_UintArray 将数组分配给变量。
VARIANT lrc_variant_UintArray(UintArray array);

// lrc_variant_UlongArray 将数组分配给变量。
VARIANT lrc_variant_UlongArray(UlongArray array);

// lrc_variant_UshortArray 将数组分配给变量。
VARIANT lrc_variant_UshortArray(UshortArray array);

// lrc_variant_VariantArray 将数组分配给变量。
VARIANT lrc_variant_VariantArray(VariantArray array);

// lrc_variant_<Type-Name>Array_by_ref 函数系列
```

```
// 返回对由 Type-name 指定的类型的数组的引用。返回类型
// 为变量。
VARIANT lrc_variant_<Type-Name>Array_by_ref( <Type-Name>Array array );

// lrc_variant_BoolArray_by_ref 返回对数组的引用
VARIANT lrc_variant_BoolArray_by_ref(BoolArray array);

// lrc_variant_BstrArray_by_ref 返回对数组的引用
VARIANT lrc_variant_BstrArray_by_ref(BstrArray array);

// lrc_variant_ByteArray_by_ref 返回对数组的引用
VARIANT lrc_variant_ByteArray_by_ref(ByteArray array);

// lrc_variant_CharArray_by_ref 返回对数组的引用
VARIANT lrc_variant_CharArray_by_ref(CharArray array);

// lrc_variant_CoObjectArray_by_ref 返回对数组的引用
VARIANT lrc_variant_CoObjectArray_by_ref(CoObjectArray array);

// lrc_variant_CurrencyArray_by_ref 返回对数组的引用
VARIANT lrc_variant_CurrencyArray_by_ref(CurrencyArray array);

// lrc_variant_DateArray_by_ref 返回对数组的引用
VARIANT lrc_variant_DateArray_by_ref(DateArray array);

// lrc_variant_DispObjectArray_by_ref 返回对数组的引用
VARIANT lrc_variant_DispObjectArray_by_ref(DispObjectArray array);

// lrc_variant_DoubleArray_by_ref 返回对数组的引用
VARIANT lrc_variant_DoubleArray_by_ref(DoubleArray array);

// lrc_variant_ErrorArray_by_ref 返回对数组的引用
VARIANT lrc_variant_ErrorArray_by_ref(ErrorArray array);

// lrc_variant_FloatArray_by_ref 返回对数组的引用
VARIANT lrc_variant_FloatArray_by_ref(FloatArray array);

// lrc_variant_IntArray_by_ref 返回对数组的引用
VARIANT lrc_variant_IntArray_by_ref(IntArray array);

// lrc_variant_LongArray_by_ref 返回对数组的引用
VARIANT lrc_variant_LongArray_by_ref(LongArray array);

// lrc_variant_ShortArray_by_ref 返回对数组的引用
```

```
VARIANT lrc_variant_ShortArray_by_ref(ShortArray array);

// lrc_variant_UintArray_by_ref 返回对数组的引用
VARIANT lrc_variant_UintArray_by_ref(UintArray array);

// lrc_variant_UlongArray_by_ref 返回对数组的引用
VARIANT lrc_variant_UlongArray_by_ref(UlongArray array);

// lrc_variant_UshortArray_by_ref 返回对数组的引用
VARIANT lrc_variant_UshortArray_by_ref(UshortArray array);

// lrc_variant_VariantArray_by_ref 返回对数组的引用
VARIANT lrc_variant_VariantArray_by_ref(VariantArray array);

// lrc_Get<Type-Name>ArrayFromVariant 函数系列从变量中
// 提取由 Type-name 指定的类型的数组。
<Type-Name> lrc_Get<Type-Name>ArrayFromVariant const VARIANT* var);

// lrc_GetBoolArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetBoolArrayFromVariant(const VARIANT* var);

// lrc_GetBstrArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetBstrArrayFromVariant(const VARIANT* var);

// lrc_GetByteArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetByteArrayFromVariant(const VARIANT* var);

// lrc_GetCharArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetCharArrayFromVariant(const VARIANT* var);

// lrc_GetCoObjectArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetCoObjectArrayFromVariant(const VARIANT* var);

// lrc_GetCoInstanceArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetCoInstanceArrayFromVariant(const VARIANT* var);

// lrc_GetCurrencyArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetCurrencyArrayFromVariant(const VARIANT* var);

// lrc_GetDateArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetDateArrayFromVariant(const VARIANT* var);

// lrc_GetDispObjectArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetDispObjectArrayFromVariant(const VARIANT* var);
```

```
// lrc_GetDoubleArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetDoubleArrayFromVariant(const VARIANT* var);

// lrc_GetErrorArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetErrorArrayFromVariant(const VARIANT* var);

// lrc_GetFloatArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetFloatArrayFromVariant(const VARIANT* var);

// lrc_GetIntArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetIntArrayFromVariant(const VARIANT* var);

// lrc_GetLongArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetLongArrayFromVariant(const VARIANT* var);

// lrc_GetShortArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetShortArrayFromVariant(const VARIANT* var);

// lrc_GetUintArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetUintArrayFromVariant(const VARIANT* var);

// lrc_GetUlongArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetUlongArrayFromVariant(const VARIANT* var);

// lrc_GetUshortArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetUshortArrayFromVariant(const VARIANT* var);

// lrc_GetVariantArrayFromVariant 从变量中提取数组。
VARIANT lrc_GetVariantArrayFromVariant(const VARIANT* var);

// lrc_Get<Type-Name>Array_by_refFromVariant
// 函数系列从变量中的指针引用提取在 Type-Name 中
// 指定的类型的数组。
<Type-Name> lrc_Get<Type-Name>Array_by_refFromVariant const VARIANT* var);

// lrc_GetShortArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetShortArray_by_refFromVariant(const VARIANT* var);

// lrc_GetUshortArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetUshortArray_by_refFromVariant(const VARIANT* var);
```

```
// lrc_GetCharArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetCharArray_by_refFromVariant(const VARIANT* var);

// lrc_GetIntArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetIntArray_by_refFromVariant(const VARIANT* var);

// lrc_GetUintArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetUintArray_by_refFromVariant(const VARIANT* var);

// lrc_GetUlongArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetUlongArray_by_refFromVariant(const VARIANT* var);

// lrc_GetByteArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetByteArray_by_refFromVariant(const VARIANT* var);

// lrc_GetLongArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetLongArray_by_refFromVariant(const VARIANT* var);

// lrc_GetFloatArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetFloatArray_by_refFromVariant(const VARIANT* var);

// lrc_GetDoubleArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetDoubleArray_by_refFromVariant(const VARIANT* var);

// lrc_GetBoolArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetBoolArray_by_refFromVariant(const VARIANT* var);

// lrc_GetScodeArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetScodeArray_by_refFromVariant(const VARIANT* var);

// lrc_GetCurrencyArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetCurrencyArray_by_refFromVariant(const VARIANT* var);
```

```
// lrc_GetDateArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetDateArray_by_refFromVariant(const VARIANT* var);

// lrc_GetBstrArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetBstrArray_by_refFromVariant(const VARIANT* var);

// lrc_GetCoObjectArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetCoObjectArray_by_refFromVariant(const VARIANT* var);

// lrc_GetDispObjectArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetDispObjectArray_by_refFromVariant(DispObjectArray array);

// lrc_GetVariantArray_by_refFromVariant 从变量中的
// 指针引用提取数组。
VARIANT lrc_GetVariantArray_by_refFromVariant(DispObjectArray array);

// GetElementFrom<n>D<Type-Name>Array
// 函数系列从 SafeArray 中检索指定类型的元素。
// 返回变量的类型是实际语言类型（如无符号长整型），
// 而非 lrc 类型（例如，ulong 类型）。
// VuGen 将这些调用生成为注释掉的代码，
// 您可以使用它们存储参数化变量。若要使用
// 这些调用，请取消它们的注释并将函数
// 返回值分配给适当类型的变量。
<Actual Type-Name> GetElementFrom<n>D<Type-Name>Array(<Type-Name>Array Array, int
index1,..., indexn);

// PutElementIn<n>D<Type-Name>Array 函数
// 系列将给定类型的值存储于在 Type-Name 中
// 给出的 lrc 类型的数组中。
void PutElementIn<n>D<Type-Name>Array(<Type-Name>Array array, int index1,...int indexn,
<Actual Type-Name> Value);

// lrc_FetchRecordset 函数将指针在记录集中
// 向前或向后移动到当前记录。
void lrc_FetchRecordset(recordset15* recordset,          int steps);

// lrc_print_recordset 函数输出由 NumberOfRows
// 指示的行数。如果 NumberOfRows 等于 -1，则
// 函数将输出所有行。
```

```
void lrc_print_recordset (Recordset15* recordset, long NumberOfRows);

// lrc_print_variant 函数输出变量。
// 如果变量包含数组或记录，则只输出
// 内容的类型。
void lrc_print_variant (Variant var);

// lrc_save_rs_param 函数将指定字段
// 的值保存到 ADO 记录集的记录中。
void  lrc_save_rs_param(Recordset15* recordset, int row, short col, void* reserved,
char*param_name);

// lrc_RecordsetWrite 函数更新 ADO
// 记录集中的字段。首先，它将指针在记录
// 集中移动指定步数。然后，它将值写入当前
// 记录中。此函数可以表示用户对网格中项目
// 的更改。
void  lrc_RecordsetWrite(Recordset15* recordset, VARIANT vIndex, VARIANT vValue, long
steps);

// lrc_<type>_by_ref 函数将指针的内存分配给
// 类型 <type> 的变量，并使用 str 中包含的值
// 对其进行初始化。
int * lrc_<type>_by_ref(const char* str);

// lrc_FetchRecordsetUntilEOF 函数将指针
// 在记录中移动，直到移动到记录集的末尾。
void  lrc_FetchRecordsetUntilEOF(Recordset15* recordset);

// lrc_RecordsetAddColumn 函数将新列添加
// 到记录集中。它类似于 Append ADO 方法。
void  lrc_RecordsetAddColumn(Recordset15* recordset, BSTR name, int type,        long
size, int attribute);

// lrc_RecordsetDeleteColumn 函数从记录集中删除列。
// 它类似于 Delete ADO 方法。
void  lrc_RecordsetDeleteColumn(Recordset15* recordset, VARIANT index);

// lrd_alloc_connection 函数分配 LRD_CONNECTION
// 结构。在 lrd_free_connection 释放该结构之前，它是
// 有效的。
LRDRET lrd_alloc_connection ( LRD_CONNECTION **mpptConnection, unsigned int
muiDBType, LRD_CONTEXT *mptContext, long mliOption, int miDBErrorSeverity );
```

```
// lrd_assign 函数将 null 终止字符串
// （文字或存储区域）通过其描述符分配给变量。
// 必要时，字符串将被转换为变量的
// 数据类型和长度。若要在字符串中包含
// null 字符 ('\0')，或者如果字符串不是
// null 终止的，请使用 lrd_assign_ext 或
// lrd_assign_literal。只能通过
// 指定 NULL 或 0（不带引号），才能为 lrd_assign
// 分配 NULL 值。
LRDRET lrd_assign ( LRD_VAR_DESC *mptVarDesc, char *mpcValStr, char *mpszValStrFmt,
unsigned long muliIndex, long mliOffset );

// lrd_assign_ext 将带有显式长度的
// 存储区域值通过其描述符分配给变量。
// 必要时，该值将被转换为变量的数据类型
// 和长度。该值可以包含嵌入的 null 字符
// ('\0')。也可以通过使用 NULL 或 0（不带引号）
// 指定 NULL 值。
LRDRET lrd_assign_ext ( LRD_VAR_DESC *mptVarDesc, char *mpcValStr, long mliValStrLen,
char *mpszValStrFmt, unsigned long muliIndex, long mliOffset );

// lrd_assign_literal 函数将由引号
// 括起的文字字符串值通过其描述符分配给变量
// （标量或数组）。必要时，该值将被转换为
// 变量的数据类型和长度。该值可以
// 包含嵌入的 null 字符 ('\0')。也可以
// 通过使用 NULL 或 0（不带引号）指定
// NULL 值。字符串的长度由 (sizeof(string) -1)
// 确定。
LRDRET lrd_assign_literal ( LRD_VAR_DESC *mptVarDesc, char *mpcValStr, char
*mpszValStrFmt, unsigned long muliIndex, long mliOffset );

// lrd_assign_bind 函数将 null 终止
// 字符串值分配给标量变量并将变量绑定到
// 占位符。如果字符串不是 null 终止的，请使用
// lrd_assign_bind_ext 或 lrd_assign_bind_literal。
// 此函数将执行 lrd_assign 和 lrd_bind_placeholder
// 函数，但它被限定为标量型
// 变量。
LRDRET lrd_assign_bind ( LRD_CURSOR *mptCursor, char *mpszPlaceholder, char
*mpcValStr, LRD_VAR_DESC *mptVarDesc, char *mpszValStrFmt, unsigned long muliOption,
int miDBErrorSeverity );

// lrd_assign_bind_ext 函数将带有
```

// 显式长度的存储区域通过其描述符分配给
// 标量变量，并将变量绑定到占位符。
// 必要时，该值将被转换为变量的类型和
// 长度。该值可以包含嵌入的 null 字符 ('\0')。
// 也可以通过使用 NULL 或 0（不带引号）
// 指定 NULL 值。此函数执行 lrd_assign_ext
// 和 lrd_bind_placeholder
// 函数，但它被限定为标量型
// 变量。
LRDRET lrd_assign_bind_ext (LRD_CURSOR *mptCursor, char *mpszPlaceholder, char *mpcValStr, long mliValStrLen, LRD_VAR_DESC *mptVarDesc, char *mpszValStrFmt, unsigned long muliOption, int miDBErrorSeverity);

// lrd_assign_bind_literal 函数将
// 由引号括起的文字字符串值通过其描述符
// 分配给标量变量，然后将变量绑定到 SQL 语句
// 文本中的占位符。必要时，该值将被转换
// 为变量的类型和长度。该值可以包含嵌入
// 的 null 字符 ('\0')。也可以通过
// 使用 NULL 或 0（不带
// 引号）指定 NULL
// 值。
LRDRET lrd_assign_bind_literal (LRD_CURSOR *mptCursor, char *mpszPlaceholder, char *mpcValStr, LRD_VAR_DESC *mptVarDesc, char *mpszValStrFmt, unsigned long muliOption, int miDBErrorSeverity);

// lrd_attr_set 函数设置 LRDDBI 句柄
// 的属性。在使用 lrd_handle_alloc 调用此
// 函数前必须显式分配句柄。只有 Oracle 8.0 和
// 更高版本才支持此函数。
LRDRET lrd_attr_set (void *mpvLRDDBIHandleTarget, long mliAttrType, void *mpvValue, long mliValueLen, int miDBErrorSeverity);

// lrd_attr_set_from_handle 函数设
// 置 LRDDBI 句柄的属性。在使用 lrd_handle_alloc
// 调用此函数前必须显式分配句柄。
// 只有 Oracle 8.0 和更高版本才支持
// 此函数。
LRDRET lrd_attr_set_from_handle (void *mpvLRDDBIHandleTarget, long mliAttrType, void *mpvValue, long mliValueLen, int miDBErrorSeverity);

// lrd_attr_set_literal 函数设置
// LRDDBI 句柄的属性。此函数使用文字
// 字符串指定属性，从而允许其包含 NULL 字符。

```
LRDRET lrd_attr_set_literal ( void *mpvLRDDBIHandleTarget, long mliAttrType, void
**mppvValue, int miDBErrorSeverity );

// lrd_bind_col 函数将主机变量绑定到输出列。
// VuGen 为结果集中的所有列生成 lrd_bind_col
// 语句。
LRDRET lrd_bind_col ( LRD_CURSOR *mptCursor, int muiColNum, LRD_VAR_DESC
*mptVarDesc, long mliOption, int miDBErrorSeverity );

// lrd_bind_cols 函数将主机变量绑定
// 到输出列。VuGen 为结果集中的所有列
// 生成 lrd_bind_col 语句。
LRDRET lrd_bind_cols ( LRD_CURSOR *mptCursor, LRD_BIND_COL_INFO
*mptBindColInfo, int miDBErrorSeverity );

// lrd_bind_cursor 将光标绑定到
// SQL 语句文本中的占位符。此函数
// 要与包含 PL/SQL 命令的语句配合使用。
LRDRET lrd_bind_cursor( LRD_CURSOR *mptCursor, char *mpszPlaceholder, unsigned long
muliOption, LRD_CURSOR **mpptBoundCursor, int miDBErrorSeverity );

// lrd_bind_placeholder 函数将
// 主机变量或数组绑定到占位符。此函数
// 要与 lrd_assign 配合使用。
LRDRET lrd_bind_placeholder ( LRD_CURSOR *mptCursor, char *mpszPlaceholder,
LRD_VAR_DESC *mptVarDesc, unsigned long muliOption, int miDBErrorSeverity );

// lrd_cancel 函数取消提交给数据库
// 服务器的前一语句或信息。在 Oracle
// 中，lrd_cancel 通知服务器取消查询并
// 释放与光标相关的所有资源。如果只需要
// 多行查询的第一行，此函数是非常有用的。
// 可以在 lrd_fetch 之后调用 lrd_cancel
// 以通知服务器不需要执行其他提取
// 操作。
LRDRET lrd_cancel ( LRD_CONNECTION *mptConnection, LRD_CURSOR *mptCursor, long
mliOption, int miDBErrorSeverity );

// lrd_close_all_cursors 函数关闭
// 用于指定连接的所有打开的光标。
LRDRET lrd_close_all_cursors ( LRD_CONNECTION *mptConnection );

// lrd_close_connection 函数
// 关闭与数据库的指定连接。
```

LRDRET lrd_close_connection (LRD_CONNECTION **mpptConnection, long mliOption, int miDBErrorSeverity);

// lrd_close_context 函数关闭 ctlib 函数的上下文。
LRDRET lrd_close_context (LRD_CONTEXT **mpptContext, long mliOption, int miDBErrorSeverity);

// lrd_close_cursor 函数关闭指定
// 数据库光标。关闭光标后，则无法再使用
// 该光标执行 SQL 语句。
LRDRET lrd_close_cursor (LRD_CURSOR **mpptCursor, int miDBErrorSeverity);

// lrd_col_data 函数设置指向
// 特定列的数据的指针。
// 此函数访问数据，但不执行绑定操作。
// 注意，若要引用数据（通过指针），必须将其
// 转换为相应类型。lrd_col_data 只出现
// 在 Vuser 脚本目录的 print.inl 文件中。
LRDRET lrd_col_data (LRD_CURSOR *mptCursor, unsigned int muiColNum, long mliOffset, LRD_VAR_DESC *mptVarDesc, unsigned long *mpuliFetchedLen, int miDBErrorSeverity);

// lrd_commit 函数提交当
// 前数据库事务。提交事务后，
// 不能执行回滚。
LRDRET lrd_commit (LRD_CONTEXT *mptContext, LRD_CONNECTION *mptConnection, int miDBErrorSeverity);

// lrd_ctlib_cursor 函数指定要初始化
// 的 CtLib 光标命令。此函数指定命令
// 类型及其适用选项。
LRDRET lrd_ctlib_cursor (LRD_CURSOR *mptCursor, char *mpcName, long mliNameLen, char *mpcText, long mliTextLen, LRDOS_INT4 mjCmdType, LRDOS_INT4 mjCmdOption, int miDBErrorSeverity);

// lrd_data_info 函数基于请求
// 规范获取 I/O 描述符信息。
// 请求规范字符串是 null 终止的字符串，
// 其中包含由分号分隔的 <keyword>=<value>
// 规范对。
LRDRET lrd_data_info (LRD_CURSOR *mptCursor, unsigned int muiColNum, unsigned long muliIODescNum, char *mpszReqSpec, int miDBErrorSeverity);

// lrd_db_option 函数为数据库光标、
// 上下文或连接设置选项。

```
LRDRET lrd_db_option ( void *mpvTargetObject, unsigned long muliOption, void
*mpvOptionValue, int miDBErrorSeverity );

// lrd_dynamic 函数指定要处理的动态 SQL 语句。
LRDRET lrd_dynamic ( LRD_CURSOR *mptCursor, char *mpcID, long mliIDLen, char
*mpcText, long mliTextLen, LRDOS_INT4 mjCmdType, int miDBErrorSeverity );

// lrd_end 函数关闭 VuGen 环境
// 并清除所有打开的结构和指针。此函数
// 在 Vuser 脚本中只被调用一次。
LRDRET lrd_end ( void *mpvEndParams );

// lrd_env_init 函数分配并
// 初始化 LRDBI 环境句柄。
// 如果被指向的指针是 NULL，则将分配
// 新的 LRDDBI 环境句柄。如果该值不是 NULL，
// 且被指向的 LRDDBI 句柄具有要求
// 的属性，则不会分配新的 LRDDBI 句柄，
// 并且函数将成功返回。否则将
// 返回 LRDRET_E_EXISTING_HANDLE_ERROR。
LRDRET lrd_env_init ( unsigned int muiDBType, void **mppvLRDDBIHandleEnv, unsigned
long muliMode, int miDBErrorSeverity );

// lrd_exec 函数执行 lrd_stmt 设置的 SQL 语句。
LRDRET lrd_exec ( LRD_CURSOR *mptCursor, unsigned long muliTotalRows, unsigned long
muliSkipRows, unsigned long *mpuliRowsProcessed, long mliOption, int miDBErrorSeverity );

// lrd_fetch 函数从结果集中提取后续若干行。
// 在录制的脚本中，结果集中的行数是参数
// mliRequestedRows 的绝对值。例如，
// 如果 lrd_fetch 的第二个参数是 -14，
// 则检索了十四行数据。
//
LRDRET lrd_fetch ( LRD_CURSOR *mptCursor, long mliRequestedRows, long
mliRowsBatchSize, unsigned long *mpuliFetchedRows, LRD_PRINT_ROW_TYPEDEF
mpfjPrintRow, int miDBErrorSeverity );

// lrd_fetchx 函数使用扩展
// 提取 (SQLExtendedFtech) 从结果集
// 中提取后续若干行。在录制的脚本中，
// 结果集中的行数是参数 mliRequestedRows
// 的绝对值。例如，如果 lrd_fetchx 的第二个
// 参数是 -14，则检索了十四行数据。
LRDRET lrd_fetchx ( LRD_CURSOR *mptCursor, long mliRequestedRows, long
```

mliRowsBatchSize, unsigned long *mpuliFetchedRows, LRD_PRINT_ROW_TYPEDEF mpfjPrintRow, long mliOption, int miDBErrorSeverity);

// lrd_fetch_adv 使用 ODBC 扩展
// 提取 (SQLExtendedFtech) 从结果集
// 中提取多行。在录制的脚本中，
// 结果集中的行数是参数 mliRequestedRows 的
// 绝对值。
LRDRET lrd_fetch_adv(LRD_CURSOR *mptCursor, long mliRequestedRows, long mliRowsBatchSize, unsigned long *mpuliFetchedRows, LRD_PRINT_ROW_TYPEDEF mpfjPrintRow, long mliOption, int miDBErrorSeverity, long mlilFetchOffset, unsigned short musOrientation);

// lrd_free_connection 释放 lrd_alloc_connection
// 创建的 LRD_CONNECTION 结构。
LRDRET lrd_free_connection (LRD_CONNECTION **mpptConnection, long mliOption, int miDBErrorSeverity);

// lrd_handle_alloc 函数分配并初始化
// LRDBI 句柄。注意，在调用此函数前必须
// 显式地（使用 lrd_handle_alloc）或
// 隐式地（使用 lrd_logon）分配父句柄。
LRDRET lrd_handle_alloc (unsigned int muiDBType, void *mpvLRDDBIHandleParent, long mliDBHandleType, void **mppvLRDDBIHandle, int miDBErrorSeverity);

// lrd_handle_free 函数显式地释放
// LRDDDBI 句柄。注意，句柄必须已被
// 显式地（使用 lrd_handle_alloc）
// 或隐式地（使用 lrd_logon）分配。只有
// Oracle 8.0 和更高版本才支持此函数。
LRDRET lrd_handle_free (void **mppvLRDDBIHandle, int miDBErrorSeverity);

// lrd_init 函数设置包含 LRD 环境
// 初始化信息的 LRD_INIT_INFO 结构。
// 此函数在脚本中只被调用一次，并且要
// 置于所有其他 LRD 函数调用之前。
LRDRET lrd_init (LRD_INIT_INFO *mptInitInfo, LRD_DEFAULT_DB_VERSION *mpatDefaultDBVersion);

// lrd_initialize_db 函数初始化数据库
// 进程环境。注意，内存回调函数是
// 不受支持的。
LRDRET lrd_initialize_db(unsigned int muiDBType, long mliMode, int miDBErrorSeverity);

```
// lrd_logoff 函数终止由 lrd_logon 或
//  lrd_logon_ext 启动的简单数据库会话。
LRDRET lrd_logoff ( void *mppvLRDDBIHandleSvcCtx, int miDBErrorSeverity );

// lrd_logon 函数创建简单登录会话，
// 其中的用户名、密码和数据库服务器
// 标识字符串都是 null 终止字符串。
// 只有 Oracle 8.0 和更高版本才支持此函数。
LRDRET lrd_logon (void *mpvLRDDBIHandleEnv, void **mppvLRDDBIHandleSvcCtx, char
*mpcUserName, char *mpcPassword, char *mpcServerID, int miDBErrorSeverity);

// lrd_logon_ext 函数通过显式指定
// 长度的用户名、密码和数据库
// 服务器标识字符串创建简单数据库会话。
// 只有 Oracle 8.0 和更高版本
// 才支持此函数。
LRDRET lrd_logon_ext (void const *mpvLRDDBIHandleEnv, void
**mppvLRDDBIHandleSvcCtx, char *mpcUserName, long mliUserNameLen, char
*mpcPassword, long mliPasswordLen, char *mpcServerID, long mliServerIDLen, int
miDBErrorSeverity );

// lrd_msg 函数允许您向日志文件
// output.txt 发送数据库消息。只有
// 当指定消息的类匹配 Log 运行时设置中的
// 设置时，消息才发送给日志文件。例如，
// 如果在运行时设置中选择“Brief Log”，
// 则只有分配给 brief 类的调式消息才会
// 出现。
LRDRET lrd_msg ( int miMsgClass, char *mpszMsg );

// 函数 lrd_oci8_to_oci7 将现有 Oracle OCI 8
// 连接 mpptConnectionconverts 转换为 Oracle
// OCI 7 连接。脚本将使用 OCI 7 连接通过执行
// OCI 7 和 OCI 8 操作来继续运行。此函数修改现有
// 连接 mpptConnection 并且不创建任何与数据库的
// 新的连接。
//
LRDRET lrd_oci8_to_oci7 ( void* mpvLRDDBIHandleSvcCtx, LRD_CONNECTION**
mpptConnection, int miDBErrorSeverity );

// lrd_open_connection 函数设置 LRD_CONNECTION
// 结构。在使用 lrd_close_connection 释放该结构之前，
// 它是有效的。
LRDRET lrd_open_connection ( LRD_CONNECTION **mpptConnection, unsigned int
```

muiDBType, char *mpszUser, char *mpszPassword, char *mpszServer, char *mpszExtConnectStr, LRD_CONTEXT *mptContext, long mliOption, int miDBErrorSeverity);

// lrd_open_context 函数打开 ctlib 函数的上下文。
LRDRET lrd_open_context (LRD_CONTEXT **mpptContext, unsigned int muiDBType, long mliLibVersionBehavior, unsigned int muiOption, int miDBErrorSeverity);

// lrd_open_cursor 函数通过设置 LRD_CURSOR
// 结构打开光标。您可以使用单个光标执行后续 SQL
// 语句。在使用 lrd_close_cursor 释放该结构之前，
// 它是有效的。
LRDRET lrd_open_cursor (LRD_CURSOR **mpptCursor, LRD_CONNECTION *mptConnection, int miDBErrorSeverity);

// lrd_option 函数将值分配给 LRD
// 选项 (LRD_OPTION_) 之一。
LRDRET lrd_option (unsigned long muliOption, void *mpvOptionValue);

// lrd_ora8_attr_set 函数设置 LRDDBI 句柄
// 的属性。在使用 lrd_handle_alloc 调用此
// 函数前必须显式分配句柄。只有 Oracle 8.0 和
// 更高版本才支持此函数。
LRDRET lrd_ora8_attr_set (void *mpvLRDDBIHandleTarget, long mliAttrType, void *mpvValue, long mliValueLen, int miDBErrorSeverity);

// lrd_ora8_attr_set_from_handle 函数设置
// LRDDBI 句柄的属性。在使用 lrd_handle_alloc
// 调用此函数之前，必须显式地分配句柄。
// 只有 Oracle 8.0 和更高版本才支持此函数。
//
LRDRET lrd_ora8_attr_set_from_handle (void *mpvLRDDBIHandleTarget, long mliAttrType, void *mpvValue, long mliValueLen, int miDBErrorSeverity);

// lrd_ora8_attr_set_literal 函数
// 设置 LRDDBI 句柄的属性。此函数使用
// 文本字符串指定属性，从而允许其包含 NULL
// 字符。
LRDRET lrd_ora8_attr_set_literal (void *mpvLRDDBIHandleTarget, long mliAttrType, void *mpvValue, int miDBErrorSeverity);

// lrd_ora8_bind_col 函数将主机变量
// 绑定到输出列。VuGen 为结果集中的所有列生成
// lrd_ora8_bind_col 语句。
LRDRET lrd_ora8_bind_col (void *mpvLRDDBIHandleStmt, void

**mppvLRDDBIHandleDefine, int muiColNum, LRD_VAR_DESC *mptVarDesc, long mliMode, int miDBErrorSeverity);

// lrd_ora8_bind_placeholder 函数将
// 主机变量或数组绑定到占位符。此函数要
// 与 lrd_assign 配合使用。
LRDRET lrd_ora8_bind_placeholder (void *mpvLRDDBIHandleStmt, void **mppvLRDDBIHandleBind, char *mpszPlaceholder, LRD_VAR_DESC *mptVarDesc, unsigned long muliOption, long mliMode, int miDBErrorSeverity);

// lrd_ora8_commit 函数从 Oracle 8.x 客户端
// 的上一保存点提交当前数据库事务。
LRDRET lrd_ora8_commit (void *ServiceContextHandle, long Mode, int ErrorSeverity);

// lrd_ora8_exec 函数执行 lrd_stmt 设置
// 的 SQL 语句。此函数重置由 lrd_ora8_stmt 设置
// 的 SQL 语句内容；在每个 lrd_ora8_exec 函数
// 调用之后，您必须提供新的语句（并可选择
// 绑定参数）。
LRDRET lrd_ora8_exec (void *mpvLRDDBIHandleSvcCtx, char *mpvLRDDBIHandleStmt, unsigned long muliTotalRows, unsigned long muliSkipRows, unsigned long *mpuliRowsProcessed, LRD_ORA8_PRINT_ROW_TYPEDEF mpfjPrintRow, void *Reserved1, void *Reserved2, long mliMode, int miDBErrorSeverity);

// lrd_ora8_fetch 函数从结果集中
// 提取后续若干行。在录制的脚本中，结果集
// 中的行数是参数 mliRequestedRows 的
// 绝对值。例如，如果 lrd_ora8_fetch 的
// 第二个参数是 -14，则检索了十四
// 行数据。
LRDRET lrd_ora8_fetch (void *mpvLRDDBIHandleStmt, long mliRequestedRows, long mliRowsBatchSize, unsigned long *mpuliFetchedRows, LRD_PRINT_ROW_TYPEDEF mpfjPrintRow, long mliOrientation, long mliMode, int miDBErrorSeverity);

// lrd_ora8_handle_alloc 函数分配
// 并初始化 LRDBI 句柄。句柄类型只由
// 限定符指定。限定符是 LRD_HTYPE_ORACLE_ 字符串
// 的唯一后缀。例如，在 LRD_HTYPE_ORACLE_SESSION 中，
// 限定符是“SESSION”。
LRDRET lrd_ora8_handle_alloc (void const * mpvParentHandle, long const mliDBHandleType, void * * const mppvLRDDBIHandle, int const miDBErrorSeverity);

// lrd_ora8_lob_read 函数在大对象
// (LOB) 描述符已从数据库中提取后，从该

// 描述符中读取字符。注意，lrd_ora8_lob_read
// 读取的字符无法被访问。此函数用于增加
// Oracle 服务器的负载并模拟现实的服务器
// 活动。
LRDRET lrd_ora8_lob_read (void * const mpvLRDDBIHandleSvcCtx, void * const mpvLRDDBIHandleLob, unsigned long const mjAmtToRead, unsigned long * const mpjAmtRead, unsigned long const mjOffset, unsigned long const mjBufl, unsigned short const mjCsid, unsigned char const mjCsfrm, int const miDBErrorSeverity);

// lrd_ora8_rollback 函数将当前数据库事务回
// 滚到 Oracle 8.x 客户端的上一保存点。
LRDRET lrd_ora8_rollback (void *const ServiceContextHandle, long const Mode, int const ErrorSeverity);

// lrd_ora8_save_col 函数保存指定
// 表单元格的动态值并将其分配给参数。
// 在 Vuser 脚本中，此函数用于关联
// 语句和其他 SQL 语句。不使用在查询
// 期间提取的实际结果，而是使用参数
// 替换常量值。然后，此参数可由同一
// 脚本中的其他数据库语句使用。注意，
// lrd_ora8_save_col 总是置于
// lrd_ora8_fetch 语句之前。
LRDRET lrd_ora8_save_col (void *mpvLRDDBIHandleStmt, unsigned int muiColNum, long mliRowNum, unsigned long muliOption, char *mpszParamName);

// lrd_ora8_stmt 函数准备用于执行的 SQL
// 或 PL/SQL 语句。语句的文本是 null 终止
// 字符串，在该字符串中没有任何 null 字符。
// （对于包含 null 字符的字符串，请使用
// lrd_ora8_stmt_literal 或 lrd_ora8_stmt_ext。）
LRDRET lrd_ora8_stmt (void *mpvLRDDBIHandleStmt, char *mpcStmtText, long mliLanguage, long mliMode, int miDBErrorSeverity);

// lrd_ora8_stmt_ext 函数准备
// 用于执行的 SQL 或 PL/SQL 语句。
// 语句文本是由地址和显式长度指定的。
// 文本可以包含 null 字符，并且应该
// 由 null 字符终止。lrd_ora8_stmt_ext
// 可用于可能包含嵌入 null 字符的动态
// 构造存储区域。
LRDRET lrd_ora8_stmt_ext (void *mpvLRDDBIHandleStmt, char *mpcStmtText, long mliStmtTextLen, long mliLanguage, long mliMode, int miDBErrorSeverity);

// lrd_ora8_stmt_literal 函数准备
// 用于执行的文字 SQL 或 PL/SQL 语句。
// lrd_ora8_stmt_ext 可用于包含
// 嵌入 null 字符的字符串常量。语句的
// 长度由公式 sizeof()-1 确定。
LRDRET lrd_ora8_stmt_literal (void *mpvLRDDBIHandleStmt, char *mpcStmtText, long mliLanguage, long mliMode, int miDBErrorSeverity);

// 在自动提取中，数据由 exec 命令提取，
// 所以组合提取与输出的 lrd_fetch 是
// 不适用的。lrd_print 使用指向
// bympfjPrintRow 的函数输出行。
LRDRET lrd_print (LRD_CURSOR *mptCursor mptCursor, LRD_PRINT_ROW_TYPEDEF mpfjPrintRow);

// lrd_reset_rows 函数为使用 ODBC SqlSetPos 函数
// 的 UPDATE 操作准备可供提取的行。
LRDRET lrd_reset_rows (LRD_CURSOR *mptCursor, long mliRowIndex);

// lrd_result_set 函数准备用于
// 通过光标输出字符串（通常为 SQL 语句）
// 的下一结果集。对于 CtLib，它发出 ct_result
// 命令，并且在 ODBC 中它运行用于当前数据库
// 语句的 SqlMoreResults。
LRDRET lrd_result_set (LRD_CURSOR *mptCursor, long mliOpt1, long mliOpt2, int miDBErrorSeverity);

// lrd_result_set_ext 函数准备
// 用于通过光标输出字符串（通常为 SQL 语句）
// 的下一结果集。它发出用于当前数据库
// 语句的 ct_result。
LRDRET lrd_result_set_ext (LRD_CURSOR *mptCursor, long mliOpt1, long mliOpt2, long *mpliReturnCode, long *mpliResultType, int miDBErrorSeverity);

// lrd_rollback 函数将当前数据库
// 事务回滚到上一保存点。
LRDRET lrd_rollback (LRD_CONTEXT *mptContext, LRD_CONNECTION *mptConnection, int miDBErrorSeverity);

// lrd_row_count 函数返回
// 受到 UPDATE、DELETE 或 INSERT 语句影响的行数。
// 只有 ODBC 和 DB2 Vuser 才支持它。
LRDRET lrd_row_count (LRD_CURSOR * mptCursor, long * mpliRowCount, int miDBErrorSeverity);

// lrd_save_col 函数保存指定
// 表单元格的动态值并将其分配给某
// 参数。在 Vuser 脚本中，此函数
// 用于关联语句和其他 SQL 语句。不
// 使用在查询期间提取的实际结果，而是
// 使用参数替换常量值。然后，此参数
// 可由同一脚本中的其他数据库
// 语句使用。注意，lrd_save_col 总是
// 置于 lrd_fetch 语句之前。
LRDRET lrd_save_col (LRD_CURSOR *mptCursor, unsigned int muiColNum, long mliRowNum, unsigned long muliOption, char *mpszParamName);

// lrd_save_last_rowid 函数将当前结果集
// 最后一行的 rowid 保存到参数中。最后的 rowid 值
// 可用在该脚本的稍后位置。
LRDRET lrd_save_last_rowid (LRD_CURSOR *mptCursor, char *mpszParamName);

// lrd_save_ret_param 函数将存储
// 过程的返回参数的值保存到参数中。
// 如果函数无法检索返回参数，该
// 参数将标记为未初始化。此函数用于关联
// 数据库语句。
LRDRET lrd_save_ret_param (LRD_CURSOR *mptCursor, unsigned int muiRetParamNum, unsigned long muliOption, char *mpszParamName);

// lrd_save_value 函数保存占位符
// 描述符的动态值。此函数
// 用于设置输出占位符（例如，Oracle 中
// 的某些存储过程）的关联数据库语句。
// 在同一脚本中，此动态保存的值
// 可由其他数据库语句使用。注意，lrd_save_value 总是
// 跟随在 lrd_exec 语句之后。
LRDRET lrd_save_value (LRD_VAR_DESC *mptVarDesc, unsigned long muIiIndex, unsigned long muliOption, char *mpszParamName);

// lrd_send_data 函数将某数据块
// 发送给服务器。数据的位置由
// 请求规范字符串 mpszReqSpec 指定。
// 此 null 终止字符串包含关键字及其值的列表。
LRDRET lrd_send_data (LRD_CURSOR *mptCursor, unsigned int muiUnused, unsigned long muliIODescNum, char *mpszReqSpec, LRD_VAR_DESC *mptVarDesc, int miDBErrorSeverity);

// lrd_server_detach 函数删除
// 用于数据库操作的数据源的访问路径。只有 Oracle 8.0 和
// 更高版本才支持此函数。
LRDRET lrd_server_detach (void const * mpvLRDDBIHandleServer, long const mliMode, int const miDBErrorSeverity);

// lrd_session_begin 函数启动
// 服务器的用户会话。只有 Oracle 8.0 和
// 更高版本才支持此函数。
LRDRET lrd_session_begin (void *mpvLRDDBIHandleSvcCtx, void *mpvLRDDBIHandleSession, unsigned long muliCredentials, long mliMode, int miDBErrorSeverity);

// lrd_session_end 函数结束服务器的用户会话。
// 只有 Oracle 8.0 和更高版本才支持此函数。
LRDRET lrd_session_end (void const *mpvLRDDBIHandleSvcCtx, void const *mpvLRDDBIHandleSession, long const mliMode, int const miDBErrorSeverity);

// lrd_siebel_incr 函数按指定
// 值递增字符串（由数值表示）。在该函数
// 递增字符串后，它将字符串写回到原
// 内存位置。表示字符串的数值
// 以 36 为基。Siebel 使用以 36 为基的
// 算法，使用从 0 到 9 的数字或
// 从 A 到 Z 的字母（对应于 10 到 35）表示
// 数值。
LRDRET lrd_siebel_incr (char *string, int increment);

// lrd_siebel_str2num 用于自动的 Siebel 关联。
LRDRET lrd_siebel_str2num (const char *szString);

// {0>The lrd_stmt function associates a character string<}72{>?lrd_stmt 函数将字符串（通常<0）
// 为 SQL 语句}与光标相关联。如果一个 lrd_stmt 函数
// 之后跟随有另一个（两者之间没有 lrd_exec），
// 则这些字符串将被连接。
// 注意，单个光标可以支持多个连续
// 的 SQL 语句。虽然可以将 SQL 语句
// 文本拆分为若干个段，但无法拆分
// 占位符标识符。
LRDRET lrd_stmt (LRD_CURSOR *mptCursor, char *mpcText, long mliTextLen, LRDOS_INT4 mjOpt1, LRDOS_INT4 mjOpt2, int miDBErrorSeverity);

// lrd_to_printable 函数将变量

```
// 或数组元素转换为可打印字符串。
// 在录制期间，将在脚本目录的 print.inl 文件
// 中生成此函数。它提供了有关如何在 VuGen 的
// 网格中显示结果的信息。
LRDRET lrd_to_printable ( LRD_VAR_DESC *mptVarDesc, unsigned long muliIndex, char
*mpszOutputStr, unsigned long muliOutputStrSize, char *mpszOutputStrFmt );

// SiebelPreSave_x 函数指示，
// 在对 2 层 Siebel 调用 lrd_fetch 或 lrd_ora8_fetch 之前，
// 需要为自动关联保存哪些 Siebel 参数。
int SiebelPreSave_x(void);

// SiebelPostSave_x 函数在对 2 层
// Siebel 调用 lrd_fetch 或 lrd_ora8_fetch 之后
// 保存以后的 Siebel 参数的值。
int SiebelPostSave_x(void);

// ms_dns_query 函数使用
// 域名服务 (DNS) 解析指定主机名的 IP 地址。
// 此函数不会自动将域添加到主机名（需要您显式
// 包含该域）。
char * ms_dns_query( char *transaction,    char *dnsserver, [char *query_host,] [char
*local_address,] LAST );

// ms_dns_nextresult 函数在 IP 地址列表
// （由 ms_dns_query 返回）中查询下一个 IP 地址。
char* ms_dns_nextresult (char *ipaddresslist);

// ftp_delete 函数从 FTP 服务器中删除文件。
int ftp_delete (char *transaction, char *item_list, LAST);

// ftp_delete_ex 函数针对指定会话
// 从 FTP 服务器中删除文件。
int ftp_delete_ex (FTP *ppftp, char *transaction, char *item_list, LAST);

// ftp_dir 函数在 FTP 服务器上运行 dir 命令。
int ftp_dir (char *transaction, char *item_list, LAST);

// ftp_dir_ex 函数针对指定会话在 FTP
// 服务器上运行 dir 命令。
int ftp_dir_ex (FTP *ppftp, char *transaction, <List of Attributes>, LAST);

// ftp_get 函数在 FTP 服务器上设置工作目录。
int ftp_get (char *transaction, char *item_list, LAST);
```

```
// ftp_get_last_download_details 获取
// 上次下载的统计信息。它以毫秒
// 为单位将总计下载持续时间分配给 pDuration 指向
// 的长整型变量。此持续时间包括实际传输时间和
// 连接开销。
int ftp_get_last_download_details ( unsigned long * pByteCount, unsigned long *pDuration);

// ftp_get_last_download_details_ex 获取
// 会话中上次下载的统计信息。
// 它以毫秒为单位将总计下载持续时间
// 分配给 pDuration 指向的长整型变量。
// 此持续时间包括实际传输时间和连接开销。
int ftp_get_last_download_details_ex (FTP *ppFtp, unsigned long * pByteCount, unsigned long *
pDuration);

// ftp_get_ex 函数针对指定会话在 FTP 服务器
// 上设置工作目录。
int ftp_get_ex (FTP *ppftp, char *transaction, char *item_list, LAST);

// ftp_get_last_error 函数返回 FTP 会话
// 期间发生的最后一次错误。
char * ftp_get_last_error (FTP *ppftp);

// ftp_get_last_error_id 函数
// 返回在指定 FTP 会话期间发生的
// 最后一次错误的 ID 编号。
int ftp_get_last_error_id (FTP *ppftp);

// ftp_get_last_handshake_duration 以毫秒为单位返回最后一次
// 连接创建的持续时间。
double ftp_get_last_handshake_duration ();

// ftp_get_last_handshake_duration_ex 以毫秒为单位返回会话中
// 最后一次连接创建的持续时间。
double ftp_get_last_handshake_duration_ex (FTP *ppFtp);

// ftp_get_last_transfer_duration 以毫秒
// 为单位返回最后一次 get 命令的净传输时间。
// 净传输持续时间不包括连接开销。
double ftp_get_last_transfer_duration ();

// ftp_get_last_transfer_duration_ex 以
// 毫秒为单位返回此会话最后一次 get 命令
```

```
// 的净传输时间。净传输持续时间不包括
// 连接开销。
double ftp_get_last_transfer_duration_ex (FTP *ppftp);

// ftp_logon 函数执行登录到 FTP 服务器的操作。
int ftp_logon ( char *transaction, char *url, LAST);

// ftp_logon_ex 函数针对特定会话登录到 FTP 服务器。
int ftp_logon_ex (FTP *ppftp, char *transaction, char *url, LAST);

// ftp_logon 函数执行从 FTP 服务器注销的操作。
int ftp_logout ( );

// ftp_logout_ex 函数针对特定会话
// 从 FTP 服务器注销。
int ftp_logout_ex (FTP *ppftp);

// ftp_mkdir 函数在 FTP 服务器上设置工作目录。
int ftp_mkdir(char *transaction, char * path);

// ftp_mkdir_ex 函数针对指定会话
// 在 FTP 服务器上设置工作目录。
int ftp_mkdir_ex(FTP *ppftp, char *transaction, char * path);

// ftp_put 函数在 FTP 服务器上设置工作目录。
int ftp_put ( char *transaction, char *item_list, LAST);

// ftp_put_ex 函数针对指定会话
// 在 FTP 服务器上设置工作目录。
int ftp_put_ex (FTP *ppftp, char *transaction, char *item_list, LAST);

// ftp_rendir 函数重命名 FTP 服务器上的文件或目录。
int ftp_rendir (char *transaction, <item list>, LAST);

// 在下列示例中，ftp_rendir_ex 函数针对指定会话
// 重命名 FTP 服务器上的文件或目录。
int ftp_rendir_ex (FTP *ppftp, char *transaction, ITEM LIST, LAST);

// ftp_rmdir 函数重命名 FTP 服务器上的目录。
int ftp_rmdir (const char *transaction, const char * path);

// ftp_rmdir_ex 函数针对指定会话
// 删除 FTP 服务器上的目录。
int ftp_rmdir_ex (FTP *ppftp, const char *transaction, const char *path);
```

```
// imap_logon 函数通过身份验证信息（使用纯文本用户名和密码）
// 登录到 IMAP 服务器。
int imap_logon(char *transaction, char *url, char *certificate, char *key, char *password, LAST);

// imap_logon_ex 函数针对特定会话
// 登录到 MS Exchange 服务器。它使用
// 简单 IMAP（MS Exchange 协议）执行登录。
int imap_logon_ex(IMAP *ppimap, char *transaction, char *url, char *certificate, char *key, char
*password, LAST);

// imap_logout 函数从 IMAP 服务器注销。
int imap_logout( );

// imap_logout_ex 函数针对特定会话
// 从 IMAP 服务器注销。
int imap_logout_ex(IMAP *ppimap);

// imap_free_ex 函数释放 IMAP 会话描述符。
// 在从 IMAP 服务器注销后调用此函数。
int imap_free_ex(IMAP *ppimap);

// imap_get_result 函数获取 IMAP 服
// 务器返回代码。使用此函数可确定
// 由 IMAP 服务器发出的有关以前的 IMAP 函数的精确错误
// 代码。
int imap_get_result( );

// imap_get_result_ex 函数获取
// 指定服务器会话的 IMAP 服务器返回代码。
// 使用此函数可确定由 IMAP 服务器发出的有关以前的 IMAP 函数
// 的错误代码。
int imap_get_result_ex(IMAP *ppimap);

// imap_select 函数选择
// 邮箱以访问其邮件。调用此函数
// 将修改邮箱状态（未读邮件数、邮件
// 总数，等等）。若要检索邮箱属性，请调用
// imap_get_attribute_int。
int imap_select(char *transaction, char *mailbox, LAST);

// imap_select_ex 函数选择邮箱
// 以访问其邮件。调用此函数将修改
// 邮箱状态（未读邮件数、邮件总数，
```

```
// 等等)。若要检索邮箱属性，
// 请调用 imap_get_attribute_int_ex。
int imap_select_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_examine 函数选择邮箱
// 以查看其邮件。调用此函数不会
// 修改邮箱状态（未读邮件数、邮件总数，
// 等等)。
int imap_examine (char *transaction, char *mailbox, LAST);

// imap_examine_ex 函数选择
// 邮箱以查看其邮件。调用
// 此函数不会修改邮箱状态
// （未读邮件数、邮件总数，等等)。
int imap_examine_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_status 函数请求指定
// 邮箱的状态。它既不更改当前
// 选中邮箱，也不影响查询邮箱中任何
// 邮件的状态。请求这些状态
// 后，请使用 imap_get_attribute_int 获取
// 状态值。
int imap_status (char *transaction, char *mailbox, char *item, LAST);

// imap_status_ex 函数请求
// 指定邮箱的状态。它既不
// 更改当前选中邮箱，也不影响查询
// 邮箱中任何邮件的状态。请求
// 这些状态后，可使用 imap_get_attribute_int_ex 读取
// 状态值。
int imap_status_ex (IMAP *ppimap, char *transaction, char *mailbox, char *item, LAST);

// imap_list_mailboxes 函数列出
// 客户端可使用的邮箱。回复中包含名称属性、
// 层次结构分隔符和邮箱名称。
int imap_list_mailboxes(char *transaction, char *reference, char *name, LAST);

// imap_list_mailboxes_ex 函数列出
// 客户端可使用的邮箱。回复中包含名称属性、
// 层次结构分隔符和邮箱名称。
int imap_list_mailboxes_ex (IMAP *ppimap, char *transaction, char *reference, char *name,
LAST);

// imap_list_subscriptions 函数列出
```

```
// 对客户端声明为订阅或活动的邮箱。
// 回复中包含名称属性、层次结构分隔符和
// 邮箱名称。
int imap_list_subscriptions (char *transaction, char *reference, char *name, LAST);

// imap_list_subscriptions_ex 函数
// 列出对客户端声明为订阅或活动的邮箱。
// 回复中包含名称属性、层次结构分隔符和
// 邮箱名称。
int imap_list_subscriptions_ex (IMAP *ppimap, char *transaction, char *reference, char *name,
LAST);

// imap_subscribe 函数将指定
// 邮箱设置为订阅或活动的。若要检索具有订阅
// 的邮箱列表，请调用 imap_list_subscriptions。
int imap_subscribe(char *transaction, char *mailbox, LAST);

// imap_subscribe_ex 函数将指定
// 邮箱设置为订阅或活动的。若要
// 检索具有订阅的邮箱列表，请调用
// imap_list_subscriptions_ex。
int imap_subscribe_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_unsubscribe 函数取消
// 对指定邮箱的订阅。取消订阅
// 命令将从服务器的“活动”或“订阅”
// 邮箱集中删除指定邮箱名称。
int imap_unsubscribe(char *transaction, char *mailbox, LAST);

// imap_unsubscribe_ex 函数取消
// 对指定邮箱的订阅。取消订阅
// 命令将从服务器的“活动”或“订阅”邮箱
// 集中删除指定邮箱名称。
int imap_unsubscribe_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_expunge 函数使用指定
// 函数删除 IMAP 服务器上的邮件。
int imap_expunge(char *transaction, char *method, [char *message,] LAST);

// imap_expunge_ex 函数针对特定会话
// 从 IMAP 服务器上永久删除邮件。
int imap_expunge_ex(IMAP *ppimap, char *transaction, char *method, [char *message,] LAST);

// imap_store 函数更改与当前
```

```
// 邮箱中的指定邮件相关的标志。
// 可以更改所有标志、在现有标志上添加
// 标志或删除标志。
int imap_store(char *transaction, char *method, char *message, char *action, ENDITEM, LAST);

// imap_store_ex 函数更改
// 与当前邮箱中的指定邮件相关的标志。
// 可以更改所有标志、在现有标志上
// 添加标志或删除标志。
int imap_store_ex (IMAP *ppimap, char *transaction, char *method, char *message, char *action,
ENDITEM, LAST);

// imap_copy 函数将消息
// 从当前邮箱复制到另一邮箱中。
// 可以指定复制单个邮件或某个范围的邮件。
// 如果目标邮箱不存在，服务器将返回错误。
int imap_copy(char *transaction, char *message, char *mailbox, ENDITEM, LAST);

// imap_copy_ex 函数将消息从
// 当前邮箱复制到另一邮箱中。可以指定
// 复制单个邮件或某个范围的邮件。如果目标
// 邮箱不存在，服务器将返回错误。
int imap_copy_ex (IMAP *ppimap, char *transaction, char *message, char *mailbox, ENDITEM,
LAST);

// imap_fetch 函数针对特定会话
// 检索与邮箱邮件相关联的数据。
// 可以请求任何邮件数据值，
// 包括标志、标题和邮件 ID。
int imap_fetch (char *transaction, char *mode, char *message, ENDITEM, LAST);

// imap_fetch_ex 函数针对特定会话
// 检索与邮箱邮件相关联的数据。
// 可以请求任何邮件数据值，包括标志、
// 标题和邮件 ID。
int imap_fetch_ex (IMAP *ppimap, char *transaction, char *mode, char *message, char *saveto,
ENDITEM, LAST);

// imap_search 函数搜索
// 邮箱以查找与搜索标准匹配的邮件。
// 搜索标准可以由一个或多个搜索项组成。
int imap_search (char *transaction, char *key, char *param, char * saveto, ENDITEM, LAST);

// imap_search_ex 函数搜索邮箱
```

```
// 以查找与搜索标准匹配的邮件。搜索标准
// 可以由一个或多个搜索项组成。
int imap_search_ex (IMAP *ppimap, char *transaction, char *key, char *param, ENDITEM,
LAST);

// 使用 imap_set_max_param_len 设置
// 用于将邮件作为参数存储的缓冲区的
// 最大大小。该函数要在将邮件保存到参数的函数
// （imap_fetch 和 imap_search）之前被调用。
int imap_set_max_param_len ( char *size );

// 使用 imap_set_max_param_len_ex 设置用于
// 将邮件作为参数存储的缓冲区的最大大小。
// 该函数要在将邮件保存到参数
// 的函数（imap_fetch 和 imap_search）之前被调用。
int imap_set_max_param_len_ex ( IMAP *ppimap, char *size );

// imap_append 函数将文本表达式
// 作为新邮件追加到指定邮箱的末尾。
int imap_append (char *transaction, char *message, char *flag, char *mailbox, ENDITEM,
LAST);

// imap_append_ex 函数将文本表达式
// 作为新邮件追加到指定邮箱的末尾。
int imap_append_ex (IMAP *ppimap, char *transaction, char *message, char *flag, char *mailbox,
ENDITEM, LAST);

// imap_create 函数创建新的邮箱。
int imap_create(char *transaction, char *mailbox, LAST);

// imap_create_ex 函数针对特定会话创建新邮箱。
int imap_create_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_delete 函数删除现有邮箱。
int imap_delete(char *transaction, char *mailbox, LAST);

// imap_delete_ex 函数针对特定会话
// 删除现有邮箱。
int imap_delete_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_noop 函数在邮箱上执行 noop 操作。
// 这对于测试网络非常有用。
int imap_noop(char *transaction);
```

```
// imap_noop_ex 函数在指定 IMAP 会话
// 的邮箱上执行 noop 操作。
// 这对于测试网络非常有用。
int imap_noop_ex (IMAP *ppimap, char *transaction);

// imap_check 函数在当前
// 邮箱中请求检查点。检查点涉及任何依赖于
// 实现并适用于邮箱内部的内务管理。
int imap_check (char *transaction);

// imap_check_ex 函数在指定 IMAP 会话
// 的当前邮箱中请求检查点。检查点
// 涉及任何依赖于实现并适用于邮箱内部的
// 内务管理。
int imap_check_ex (IMAP *ppimap, char *transaction);

// imap_close 函数将设置
// 有 \Deleted 标记的所有邮件从
// 当前邮箱中永远删除。它将邮件从
// 选中状态返回到身份验证状态。
// 当前选中邮箱将关闭。
int imap_close(char *transaction, char *mailbox, LAST);

// imap_close_ex 函数针对特定会话将
// 设置有 \Deleted 标记的所有邮件
// 从当前邮箱中永久删除。
// 它将邮件从选中状态返回到身份验证状态。
// 当前选中的邮箱将被关闭。
int imap_close_ex (IMAP *ppimap, char *transaction, char *mailbox, LAST);

// imap_custom_request 函数执行
// 自定义的 IMAP 请求。此函数对于
// 处理未包含在 IMAP4rev1 协议规范中、特定
// 于 IMAP 服务器的扩展非常有用。
int imap_custom_request (char *transaction, char *operation, char *arguments );

// imap_custom_request_ex 函数针对特定会话
// 执行自定义的 IMAP 请求。此函数对于
// 处理未包含在 IMAP4rev1 协议规范中、特定
// 于 IMAP 服务器的扩展非常有用。
int imap_custom_request_ex (IMAP *ppimap, char *transaction, char *operation, char
*arguments);

// imap_get_attribute_int 函数将 IMAP 会话
```

```
// 属性的值作为 integer 类型值返回。
int imap_get_attribute_int(char *property);

// imap_get_attribute_int_ex 函数针对特定会话将 IMAP 属性的
// 值作为 integer 类型值返回。
int imap_get_attribute_int_ex(IMAP *ppimap, char *attribute);

// imap_get_attribute_sz 函数将 IMAP 会话
// 属性的值作为字符串返回。
char *imap_get_attribute_sz(char *property);

// imap_get_attribute_sz_ex 函数针对特定会话
// 将 IMAP 属性的值作为字符串返回。
char *imap_get_attribute_sz_ex(IMAP *ppimap, char *attribute);

// mldap_add 函数向 LDAP 目录中添加条目。
int mldap_add (char *transaction, char *dn, <List of Attributes>, LAST);

// mldap_add_ex 函数针对特定会话
// 向 LDAP 目录中添加条目。
int mldap_add_ex (MLDAP *pldap, char *transaction, char *dn, <List of Attributes>, LAST);

// mldap_delete 函数删除条目或条目的属性。
// 如果指定条目名称，该条目将被删除。
// 如果指定属性名称，则只有该属性
// 被删除。
int mldap_delete (char *transaction, char *dn, LAST);

// mldap_delete_ex 函数针对指定会话
// 从 LDAP 服务器删除文件。
int mldap_delete_ex (MLDAP *pldap, char *transaction, char *dn, LAST);

// mldap_get_attrib_name 函数检索
// 指定属性索引的属性名。可以选择
// 将此名称保存为参数。
char * mldap_get_attrib_name (char *transaction, char *index, [char *param,] LAST);

// mldap_get_attrib_name_ex 函数检索
// 指定会话的指定索引的属性名。
// 可以选择将此名称保存为参数。
char * mldap_get_attrib_name_ex (MLDAP *pldap, char *transaction, char *index,  [char
*param,] LAST);

// mldap_get_attrib_value 函数检索
```

```
// 当前条目的属性值。既可以
// 指定属性的索引，也可以指定
// 属性的名称。默认情况下，
// 如果不指定任何值，函数将
// 返回第一个属性的值。可选的 offset 参数允
// 许您获取第一个属性之外的其他属性的值。
// 例如，如果属性具有多个值，offset 为 0 将返回第一个值，
// offset 为 1 将返回第二个值，依此类推。
char * mldap_get_attrib_value (char *transaction, char *name, char *index, [char *offset,] [char
*param,] LAST);

// mldap_get_attrib_value_ex 函数检索
// 指定会话指定属性的值。既可以指定
// 属性的索引，也可以指定属性的名称。
// 默认情况下，如果不指定任何值，函数将返回
// 第一个属性的值。可选的 offset 参数
// 允许您获取第一个属性之外的其他属性
// 的值。例如，如果属性具有多个值，
// offset 为 0 将返回第一个值，offset 为 1 将返回第二个值，
// 依此类推。
// 1 would return the second, etc.
char * mldap_get_attrib_value_ex (MLDAP *pldap, char *transaction, char *name, char *index,
[char *offset,] [char *param,] LAST);

// mldap_get_next_entry 函数显示作为
// 搜索结果的下一条目。对于异步搜索，
// 此函数检索并显示条目；对于同步搜索，
// 它只显示下一条目。函数返回 DN 条目的名称，
// 但您也可以命令该函数将其
// 保存为参数。如果没有其他条目，或者如果在检索
// 项目时发生错误，函数将
// 返回 NULL。
char * mldap_get_next_entry (char *transaction, [char *timeout,] [char *param,] LAST);

// mldap_get_next_entry_ex 函数显示
// 作为指定会话的搜索结果的下一
// 条目。对于异步搜索，此函数检索
// 并显示条目；对于同步搜索，它只显示
// 下一条目。函数返回 DN 条目的名称，
// 但您也可以命令该函数将其保存为参数。
// 如果没有其他条目，或者如果
// 在检索项目时发生错误，函数将
// 返回 NULL。
char * mldap_get_next_entry_ex (MLDAP *pldap, char *transaction, [char *timeout,] [char
```

*param,] LAST);

// mldap_logon 函数执行登录到 LDAP 服务器的操作。
int mldap_logon (char *transaction, char *url, LAST);

// mldap_logon_ex 函数针对特定会话登录
// 到 LDAP 服务器。
int mldap_logon_ex (MLDAP *pldap, char *transaction, char *url, LAST);

// mldap_logoff 函数执行从 LDAP 服务器注销的操作。
int mldap_logoff ( );

// mldap_logoff_ex 函数针对特定会话
// 从 LDAP 服务器注销。
int mldap_logoff_ex (MLDAP *pldap);

// mldap_modify 函数更改 LDAP 条目
// 属性的值。每个函数只能修改一个条目。
// 若要修改另一个条目的属性，请使用
// 另一个 mldap_modify 函数。
int mldap_modify (char *transaction, char *dn, <List of Attributes>, LAST);

// mldap_modify_ex 函数针对特定会话
// 向 LDAP 目录中添加条目。
int mldap_modify_ex (MLDAP *pldap, char *transaction, char *dn, <List of Attributes>, LAST);

// mldap_rename 函数替换 LDAP 服务器上的 DN
// 项。您可以替换一个或多个属性类型。例如，
// 如果某雇员更改了名称，您可以更改
// 其 CN 属性。如果某雇员换到了其他部门，
// 您可以更改其 OU 属性。
int mldap_rename (char *transaction, char *dn, char *NewDn, LAST);

// mldap_rename_ex 函数针对特定会话替换 LDAP 服务器上
// 的 DN 项。您可以替换
// 一个或多个属性类型。例如，某雇员
// 更改了名称，您可以更改其 CN 属性。
// 如果某雇员换到了其他部门，您可以
// 更改其 OU 属性。
int mldap_rename_ex (MLDAP *pldap, char *transaction, char *dn, char *NewDn, LAST);

// mldap_search 函数在 LDAP 中搜索
// 特定属性。您指明 VuGen 用作搜索
// 基础的 DN。搜索范围可以是

```
// 基础自身 (base)、基础下一级 (Onelevel) 或
// 基础下的所有级别 (subtree)。您可以
// 指定属性及其值，还可以指定通配符。
// 如果使用通配符，VuGen 将返回指定属性
// 的所有值。
int mldap_search (char *transaction, char *base, char *scope, char *filter, [char *timeout,] [char
*mode,] [const char *SaveAsParam,][const char *Attrs,] LAST);

// mldap_search_ex 函数在 LDAP 中搜索
// 特定属性。您指明 VuGen 用作搜索
// 基础的 DN。搜索范围可以是
// 基础自身 (base)、基础下一级 (Onelevel) 或
// 基础下的所有级别 (subtree)。您可以
// 指定属性及其值，还可以指定通配符。
// 如果使用通配符，VuGen 将返回指定属性
// 的所有值。
int mldap_search_ex (MLDAP *pldap, const char *transaction, const char *base, const char
*scope, const char *filter, [const char *timeout,] [const char *mode,][const char
*SaveAsParam,][const char *Attrs,] LAST);

// mapi_delete_mail 函数删除 MS Exchange 服务器上
// 的当前或选定的电子邮件项。
int mapi_delete_mail(char *transaction, [char *ID,] LAST);

// mapi_delete_mail_ex 函数针对特定会话删除
// MS Exchange 服务器上的邮件。
int mapi_delete_mail_ex(MAPI *ppmapi, char *transaction, [char *ID,] LAST);

// mapi_get_property_sz 函数针对特定会话
// 为 MAPI 属性的值分配缓冲区，并
// 返回指向缓冲区的值。
char* mapi_get_property_sz(char *property);

// mapi_get_property_sz_ex 函数针对特定会话
// 为 MAPI 属性的值分配缓冲区，并
// 返回指向缓冲区的值。
char *mapi_get_property_sz_ex(MAPI *ppmapi, char *property);

// mapi_logon 函数使用简单 MAPI（一种
// MS Exchange 协议)登录到 MS Exchange 服务器。
int mapi_logon(char *transaction, char *profilename, char *profilepass, NULL);

// mapi_logon_ex 针对特定会话登录到
// MS Exchange 服务器。它使用简单 MAPI（一种 MS Exchange
```

```
// 协议）执行登录。
int mapi_logon_ex (MAPI *ppmapi, char *transaction, char *profilename,    char *profilepass,
NULL);

// mapi_logout 函数从 MS Exchange 服务器注销。
int mapi_logout( );

// mapi_logout_ex 函数针对特定函数
// 从 MAPI 服务器注销。
int mapi_logout_ex(MAPI *ppmapi);

// mapi_read_next_mail 函数使用 MAPI 服务器读取
// 邮件。使用 Peek 选项，可以控制
// 是否将邮件标记为“已读”或“未读”。
int mapi_read_next_mail(char *transaction, char *Show, [char *options,] LAST);

// mapi_read_next_mail_ex 函数针对
// 指定会话读取邮箱中的下一封邮件
// 消息。使用 Peek 选项，可以控制是否
// 将邮件标记为“已读”或“未读”。
int mapi_read_next_mail_ex( MAPI *ppmapi, char *transaction, char *Show, [char *Peek,] [char
*Type,] [char *Save,] [char *MessageId,] LAST );

// mapi_send_mail 函数使用 MAPI 服务器发送邮件消息。
int mapi_send_mail(char *transaction, char *To, [char *CC,] [char *BCC,] [char *Subject,] [char
*Type,] [char *Body,] [ATTACHMENTS,] LAST);

// mapi_send_mail_ex 函数针对指定会话
// 使用 MAPI 服务器发送邮件消息。
int mapi_send_mail_ex(MAPI *ppmapi, char *transaction, char *To, [char *CC,] [char *BCC,]
[char *Subject,] [char *Type,] [char *Body,] [ATTACHMENTS,] LAST);

// mapi_set_property_sz 函数设置 MAPI 会话属性值。
void mapi_set_property_sz(char *property, char *value );

// mapi_set_property_sz_ex 函数针对特定会话
// 设置 MAPI 会话属性值。
void mapi_set_property_sz_ex(MAPI *ppmapi, char *property, char *value);

// mms_close 函数关闭现有媒体播放器
// 会话。此函数仅对活动的播放器会话
// 有效。只有使用持续时间标记，而非 －1（无穷）
// 调用 mms_play 时（流传输尚未完成），播放器会话
// 才会处于活动状态。
```

```
int mms_close( );

// mms_close_ex 函数关闭指定的媒体
// 播放器会话。此函数仅对活动播放器
// 会话有效。只有使用持续时间标记，
// 而非 －1（无穷）调用 mms_play 时（流传输尚未完成），
// 播放器会话才会处于活动
// 状态。
int mms_close_ex(MMS *ppmms);

// mms_get_property 函数检索当前
// 媒体剪辑的属性。
double mms_get_property(int property);

// mms_get_property_ex 函数检索当前
// 媒体剪辑的属性。用于特定会话。
double mms_get_property_ex (MMS *ppmms, int property);

// mms_set_property 函数设置媒体播放器剪辑属性。
int mms_set_property(int property, char *value);

// mms_set_property_ex 函数针对特定会话
// 设置媒体剪辑属性。
int mms_set_property_ex(MMS *ppmms, int property, char *value);

// mms_isactive 函数检查 Media Player 是否处于
// 活动状态。它验证 Media Player 是否已打开
// 以及是否正在传入或传出数据。
int mms_isactive( );

// mms_isactive_ex 函数检查 Media Player
// 是否处于活动状态。它针对指定会话验证
// 是否打开了 Media Player，以及
// 是否正在传入或传出数据。
int mms_isactive_ex(MMS *ppmms);

// mms_pause 函数暂停媒体播放器剪辑。只有播放持续时间不为负值时，此函数
// 才起作用。
int mms_pause( );

// mms_pause_ex 函数针对指定会话
// 暂停媒体播放器剪辑。只有播放持续时间不为负值时，此函数
// 才起作用。
int mms_pause_ex(MMS *ppmms);
```

```
// mms_play 函数播放媒体播放器剪辑。如果只
// 希望连接到剪辑并手动控制它，请指定持续时间为
// 0，后面再跟所需的函数。
int mms_play(char *transaction, char *URL, [char *duration,] [char *starttime,] LAST);

// mms_play_ex 函数针对指定会话
// 播放媒体播放器剪辑。
int mms_play_ex(MMS *ppmms, char *transaction, char *URL, [char *duration,] [char
*starttime,] LAST);

// 在使用 mms_pause 暂停播放媒体剪辑后，
// mms_resume 函数继续播放。只有播放持续时间
// 不为负值时，此函数才起作用。
int mms_resume( DWORD resumetime, DWORD duration);

// 在指定会话中，在使用 mms_pause 暂停播放
// 媒体剪辑后，mms_resume_ex 函数继续
// 播放。只有播放持续时间不为负值时，此函数
// 才起作用。
int mms_resume_ex( MMS *ppmms, DWORD resumetime, DWORD duration);

// mms_sampling 函数通过收集指定
// 持续时间内的统计信息来采样播放媒体剪辑。
// 只有播放持续时间为无限（mms_play 中的
// 持续时间值设为 0）时，此函数才起作用。
int   mms_sampling(DWORD duration);

// mms_sampling_ex 函数通过收集
// 指定持续时间内的统计信息来获取
// Media Player 会话采样。只有播放持续时间为无限
// （mms_play_ex 中的持续时间值设为 0）时，
// 此函数才起作用。
int mms_sampling_ex(MMS *ppmms, DWORD duration);

// mms_set_timeout 函数设置用于
// 打开或关闭剪辑的 Media Player 超时值。
int mms_set_timeout (int type, int value);

// mms_set_timeout_ex 函数针对特定会话、设置用于
// 打开或关闭剪辑的 Media Player 超时值。
int mms_set_timeout_ex (MMS *ppmms, int type, int value);

// mms_stop 函数停止播放 Media Player
```

```
// 剪辑。只有播放持续时间为无限（mms_play 中的
// 持续时间值设为 0）时，此函数才起作用。
int   mms_stop( );

// mms_stop_ex 函数针对特定会话
// 停止播放媒体播放器剪辑。只有播放持续时间为无限
// （mms_play_ex 中的持续时间值设为 0）时，
// 此函数才起作用。
int mms_stop_ex(MMS *ppmms);

// nca_button_double_press 函数两次按指定的
// 推按钮。
int nca_button_double_press (LPCSTR name);

// nca_button_press 函数激活指定的推按钮。
int nca_button_press ( LPCSTR button );

// nca_button_set 函数将按钮状态设置为 ON
// 或 OFF。TOGGLE 选项反转当前状态。
int nca_button_set ( LPCSTR button, int istate );

// nca_combo_select_item 函数选择组合框中的项目。
int nca_combo_select_item (LPCSTR name, LPCSTR item_name);

// nca_combo_set_item 函数将 item_name 写入组合框名称。
int nca_combo_set_item ( LPCSTR name, LPCSTR item_name );

// nca_connect_server 函数使用指定的主机、端口号和模块
// 连接到 Oracle NCA 数据库服务器。
int nca_connect_server (LPCSTR host, LPCSTR port, LPCSTR command_line);

// nca_console_get_text 函数检索
// Oracle NCA 控制台消息。
int nca_console_get_text (char *text);

// nca_edit_box_press 函数在编辑框消息上按下。
int nca_edit_box_press (LPCSTR name);

// nca_edit_click 函数在指定编辑对象内单击，
// 以便将光标放置在框中。一旦
// 光标位于框中，用户就可以键入值
// 或从值列表中选择一个值。
int nca_edit_click ( LPCSTR edit );
```

```
// nca_edit_get_text 函数返回在
// 指定的编辑对象中找到的所有文本。
int nca_edit_get_text ( LPCSTR edit, char *out_string );

// nca_edit_press 函数激活编辑字段中的
// “浏览”按钮。这将打开可用值列表。
int nca_edit_press ( LPCSTR edit );

// nca_edit_set 函数将编辑对象的内容设置为
// 指定的字符串。它将替换现有字符串。
int nca_edit_set ( LPCSTR edit, LPCSTR text );

// nca_flex_click_cell 函数在 Flexfield 窗口中的
// 指定表单元格中单击。
int nca_flex_click_cell ( LPCSTR name, LPCSTR row, LPCSTR column);

// nca_flex_get_cell_data 函数获取 Flexfield 中指定单元格
// 的内容，并将其存储在变量中。
int nca_flex_get_cell_data(LPCSTR name, LPCSTR row, LPCSTR column, LPSTR data);

// nca_flex_get_column_name 函数获取 Flexfield 窗口中
// 某列的名称。此函数将列名写入
// 输出参数 column_name。
int nca_flex_get_column_name ( LPCSTR window_name, int column, LPCSTR column_name );

// nca_flex_get_row_name 函数获取 Flexfield 窗口中
// 某行的名称。此函数将行名写入
// 输出参数 row_name。
int nca_flex_get_row_name ( LPCSTR window_name, int row, LPCSTR row_name );

// nca_flex_press_clear 函数按
// 指定 Flexfield 窗口中的“清除”按钮。
int nca_flex_press_clear (LPCSTR name );

// nca_flex_press_find 函数按
// 指定 Flexfield 窗口中的“查找”按钮。
int nca_flex_press_find ( LPCSTR name );

// nca_flex_press_help 函数按指定
// Flexfield 窗口中的“帮助”按钮（问号）。
int nca_flex_press_help ( LPCSTR name );

// nca_flex_press_lov 函数单击
// 指定 Flexfield 窗口中的“值列表”按钮，
```

```
// 以便显示活动字段的值列表。
int nca_flex_press_lov( LPCSTR name, LPCSTR row, LPCSTR column    );

// nca_flex_press_ok 函数按指定
// Flexfield 窗口中的“确定”按钮。
int nca_flex_press_ok ( LPCSTR name );

// nca_flex_set_cell_data 函数设置指定
// Flexfield 窗口中的单元格数据。
int nca_flex_set_cell_data ( LPCSTR name, LPCSTR row, LPCSTR column, LPCSTR data );

// nca_flex_set_cell_data_press_ok 函数在手动
// 向 Flexfield 中输入（而不是从值列表中
// 选择）数据之后按 Flexfield 窗口中的
// “确定”按钮。
int nca_flex_set_cell_data_press_ok ( LPCSTR name, LPCSTR row, LPCSTR column, LPCSTR
data );

// exit_oracle_application 函数断开与
// Oracle NCA 数据库服务器的连接，并退出应用程序。
int exit_oracle_application( );

// nca_get_top_window 函数将顶部窗口的名称分配给
// 由 winName 指向的用户分配的缓冲区。
int nca_get_top_window ( char *winName);

// nca_java_action 函数使用指定参数
// 在 Java 对象上执行事件。
int nca_java_action(LPCSTR name, LPCSTR event, LPCSTR arglist);

// 使用 nca_java_delete_name 可以删除用于存储属性的内存
// （这些属性是录制 Java 对象时由 Vugen
// 保存的）。只有在 nca_java_set_option 中启用了
// JAVA_SAVE_PROP 选项，Vugen 才会保存 Java 对象。。
int nca_java_delete_name(LPCSTR name, LPCSTR property_name);

// nca_java_get_value 函数检索指定
// Java 对象的值。
int nca_java_get_value(LPCSTR name, char *value );

// nca_java_get_value_ex 检索 Java 对象内指定
// 属性 property_name 的值。
int nca_java_get_value_ex (LPCSTR name, LPCSTR property_name, LPSTR property_value);
```

```
// nca_java_set_option 函数在录制 Java 对象时设置选项。
int nca_java_set_option(int option, < option value > );

// nca_java_set_reply_property 函数设置
// 指定的 Java 回复属性。
void nca_java_set_reply_property (void * ReplyPropList);

// nca_list_activate_item 函数双击列表中的
// 项目。项目通过其逻辑名称指定。
int nca_list_activate_item ( LPCSTR list, LPCSTR item );

// nca_list_get_text 从列表中检索选定项目并放到 value 中。
int nca_list_get_text( LPCSTR name, char *value );

// nca_list_select_item 函数从列表中
// 选择项目（在项目上执行一次鼠标单击）。
// 项目通过其名称指定。
int nca_list_select_item ( LPCSTR list, LPCSTR item );

// nca_list_select_index_item 函数从列表中
// 选择项目（在项目上执行一次鼠标单击）。该项目
// 通过其数字索引指明。该索引被指定为
// 字符串，起始值为 0。
int nca_list_select_index_item ( LPCSTR list, int index );

// nca_logon_connect 函数执行到 Oracle NCA
// 数据库的登录。它使用指定的用户名和密码
// 连接到数据库。
int nca_logon_connect (LPCSTR connection_name, LPCSTR username, LPCSTR password,
LPCSTR database);

// nca_logon_cancel 函数取消与 Oracle NCA
// 数据库的连接。连接名称通过
// nca_logon_connect 的 connection_name
// 参数设置。但是，此函数并不断开与服务器
// 的连接。
int nca_logon_cancel (LPCSTR name);

// nca_lov_auto_select 函数输入一个字母
// 来指明要从值列表中选择的项目的第一个
// 字符。以指定字母开头的项目
// 被选中。如果存在多个以该字母
// 开头的项目，那么所有匹配的值都将显示在列表中，
// 您需要从新列表中选择一个
```

```
// 项目。
int nca_lov_auto_select ( LPCSTR name, char selection );

// nca_lov_find_value 函数查找对象的
// 值列表。当您单击“值列表”窗口中的“查找”
// 按钮时，将录制此函数。
int nca_lov_find_value ( LPCSTR name, LPCSTR value );

// nca_lov_get_item_name 函数在值列表中检索
// 某项目的名称，并将其写入该函数
// 的 value 参数。调用该函数之前，必须为值分配足够的
// 内存空间。
int nca_lov_get_item_name ( LPCSTR name, int item_index, char *value);

// nca_lov_retrieve_items 函数基于指定的
// 范围，从值列表中检索项目。该范围
// 通过函数的参数 first_item 和 last_item
// 指定，其中“1”表示第一个项目。
int nca_lov_retrieve_items ( LPCSTR name, int first_item, int last_item );

// nca_lov_select_index_item 函数使用项目的索引编号
// 从值列表中选择项目。
int nca_lov_select_index_item ( LPCSTR name, int index );

// nca_lov_select_item 函数从值列表中选择项目。
int nca_lov_select_item ( LPCSTR name, LPCSTR item );

// nca_lov_select_random_item 函数从值列表中
// 选择随机项目。第二个参数是
// 输出参数，用于指明随机选择时
// 选择的是哪个值。
int nca_lov_select_random_item ( LPCSTR name, char *item );

// nca_menu_select_item 函数根据
// 菜单的逻辑名称和项目的名称从菜单中选择
// 项目。注意，菜单和项目表示分别表示为
// 单个字符串，并使用分号分隔。
int nca_menu_select_item ( LPCSTR window, LPCSTR menu;item );

// nca_message_box_press 函数按
// 消息窗口中的指定按钮。该按钮通过其索引指定，
// 通常是从左到右
// 的按钮顺序。例如，如果消息框包含
// 三个按钮：“是”、“否”和“取消”，那么相应的
```

```
// 索引可能为 1、2 和 3。
int nca_message_box_press ( LPCSTR name, int button );

// nca_obj_get_info 函数检索指定属性的值，
// 并存储在 out_value 中。
int nca_obj_get_info ( LPCSTR object, LPCSTR property, char *out_value );

// nca_obj_mouse_click 函数在
// 对象内的指定坐标处单击鼠标。
int nca_obj_mouse_click ( LPCSTR object, int x, int y, unsigned char modifier);

// nca_obj_mouse_dbl_click 函数在
// 对象内的指定坐标处双击鼠标。
int nca_obj_mouse_dbl_click ( LPCSTR object, int x, int y, unsigned char modifier);

// nca_obj_status 函数返回指定对象的状态。
int nca_obj_status ( LPCSTR name );

// nca_obj_type 函数指定 keyboard_input 将
// 发送到的对象。此函数录制
// 特殊字符，如 Tab 键、功能键以及
// 快捷键组合。
int nca_obj_type ( LPCSTR object, unsigned char keyboard_input, unsigned char modifier);

// nca_popup_message_press 函数按
// 消息窗口中的指定按钮。
int nca_popup_message_press ( LPCSTR name, LPCSTR button );

// nca_response_get_cell_data 函数从“响应”框
// 中的单元格中检索数据。应指定“响应”对象
// 的名称和字段的名称。
int nca_response_get_cell_data ( const char *name, const char * rowname, char * data);

// nca_response_press_lov 函数单击“响应”框
// 中的下拉箭头。
int nca_response_press_lov(LPCSTR name, LPCSTR field);

// nca_response_press_ok 函数按指定响应框
// 中的“确定”
int nca_response_press_ok (LPCSTR name);

// nca_response_set_cell_data 函数向“响应”框中
// 的单元格中插入数据。应指定“响应”对象的名称、
// 单元格的名称以及数据。
```

```
int nca_response_set_cell_data (LPCSTR name, LPCSTR cell, LPCSTR data);

// nca_response_set_data 函数向“响应”框中
// 插入数据。应指定“响应”对象的名称
// 和数据。
int nca_response_set_data (LPCSTR name, LPCSTR data);

// nca_scroll_drag_from_min 函数从最小位置
// 滚动到指定距离。
int nca_scroll_drag_from_min ( LPCSTR object, int position );

// nca_scroll_line 函数滚动指定的
// 行数。此函数可用于
// 滚动栏和滑块对象。
int nca_scroll_line ( LPCSTR scroll, int lines );

// nca_scroll_page 函数滚动指定的
// 页数。此函数可用于
// 滚动栏和滑块对象。
int nca_scroll_page ( LPCSTR scroll, int pages );

// Vugen 将 nca_set_connect_opt 插入到 nca_connect_server 的前面，
// 这样就能识别为服务器连接
// 录制的值将不同于默认值。
// 保存录制的值将确保与服务器
// 的连接与录制脚本期间的服务器连接完全
// 相同。
int nca_set_connect_opt (eConnectionOption option, ...);

// nca_set_custom_dbtrace 设置在应用程序中
// 启用 DB 跟踪的自定义函数。如果使用
// 内置机制无法启用 DB 跟踪，可能需要
// nca_set_custom_dbtrace 和 nca_set_dbtrace_file_index。
// 当自定义应用程序包含非标准 UI 时，可能
// 会出现这种情况。nca_set_custom_dbtrace
// 设置在应用程序中启用 DB 跟踪的
// 自定义函数。
void nca_set_custom_dbtrace(long function);

// nca_set_dbtrace_file_index 标识跟踪文件，以便
// 以后供控制器和分析使用。如果使用
// 内置机制无法启用 DB 跟踪，可能需要
// nca_set_dbtrace_file_index 和 nca_set_custom_dbtrace。当
// 自定义应用程序包含非标准 UI 时，可能会出现
```

```
// 这种情况。
void nca_set_dbtrace_file_index(LPCSTR fileindex);

// nca_set_exception 函数指定在出现
// 异常时应执行的操作。应指定要调用以处理
// 异常窗口的函数。
void nca_set_exception (LPCSTR title, long function, [void *this_context]);

// 当显示了“启用诊断”窗口而且其中的密码
// 不同于默认值“apps”时，使用
// nca_set_diagnostics_password。必须在
// nca_connect_server 的后面添加此函数调用。
void nca_set_diagnostics_password( LPCSTR password );

// nca_set_iteration_offset 函数设置对象
// ID 编号的偏移值。在 Oracle NCA 脚本
// 常规录制中，VuGen 会录制每个
// 对象的名称。（在起始页中设置 record=names）。
// 如果您的版本不支持 record=names
// 标记，VuGen 将为每个对象生成一个新 ID 编号。
// 运行脚本的多次循环时，对象在
// 每次打开都将生成新的 ID 编号。
// 因此，当您回放脚本时，特殊对象
// 的 ID 编号将与首次循环之后
// 同一对象的 ID 不匹配，从而导致测试
// 失败。
void nca_set_iteration_offset (int offset);

// nca_set_server_response_time 函数为 Oracle NCA
// 服务器指定响应超时值。这是用户
// 向服务器发送请求之后继续停留在
// 侦听状态的时间。
void nca_set_server_response_time(int time);

// nca_set_think_time 函数指定脚本
// 执行期间使用的思考时间范围。测试使用指定时间
// 范围内的随机思考时间，并在每次操作之后暂停该
// 思考时间长度。
void nca_set_think_time ( DWORD start_range, DWORD end_range );

// nca_set_window 函数指明活动窗口的名称。
int nca_set_window ( LPCSTR window);

// nca_tab_select_item 函数选择选项卡项目。
```

```
int nca_tab_select_item ( LPCSTR tab, LPCSTR item );

// nca_tree_activate_item 函数激活
// Oracle NCA 树中的指定项目。
int nca_tree_activate_item (LPCSTR name, LPCSTR item);

// nca_tree_select_item 函数选择
// Oracle NCA 树中的指定项目。
int nca_tree_select_item (LPCSTR name, LPCSTR item);

// nca_tree_expand_item 函数展开 Oracle NCA 树中的节点。
int nca_tree_expand_item (LPCSTR name, LPCSTR item);

// nca_tree_collapse_item 函数折叠
// Oracle NCA 树中的节点。
int nca_tree_collapse_item (LPCSTR name, LPCSTR item);

// nca_win_close 函数关闭指定窗口。
int nca_win_close ( LPCSTR window );

// nca_win_get_info 函数检索指定属性的值，
// 并存储在 out_value 中。
int nca_win_get_info ( LPCSTR window, LPCSTR property, char *out_value );

// nca_win_move 将某窗口移到一个新的绝对位置。
int nca_win_move ( LPCSTR window, int x, int y );

// nca_win_resize 函数更改窗口的位置。
int nca_win_resize ( LPCSTR window, int width, int height );

// pop3_command 函数向 POP3 服务器发送
// 命令。服务器返回命令结果。
// 例如，如果发送命令“UIDL 1”，服务器将返回
// 第一封邮件的唯一 ID。
long pop3_command(char *transaction, char *command, [char *command,] [char
*save_to_param,] LAST);

// pop3_command_ex 函数针对特定会话
// 向 POP3 服务器发送命令。服务器
// 返回命令结果。例如，如果
// 发送命令“UIDL 1”，服务器将返回
// 第一封邮件的唯一 ID。
long pop3_command_ex (POP3 *pppop3, char *transaction, char *command,  [char *command,]
LAST);
```

```
// pop3_delete 函数删除 POP3 服务器上的邮件。
long pop3_delete(char *transaction, char *deleteList, [char *save_to_param,] LAST);

// pop3_delete_ex 函数针对特定会话
// 删除 POP3 服务器上的邮件。
long pop3_delete_ex(POP3 *pppop3, char *transaction, char *deleteList, LAST);

// pop3_logon 函数登录到 POP3 服务器。
// 它使用 FTP 协议使用的格式。
int pop3_logon (char *transaction, char *url, LAST);

// pop3_logon_ex 函数针对特定会话登录
// 到 POP3 服务器。它使用 FTP 协议使用的格式。
int pop3_logon_ex (POP3 *pppop3, char *transaction, char *url, LAST);

// pop3_logoff 函数从 POP3 服务器注销。
long pop3_logoff( );

// pop3_logoff_ex 函数针对特定会话
// 从 POP3 服务器注销。
long pop3_logoff_ex(POP3 *pppop3);

// pop3_free 函数释放 POP3 服务器并取消
// 所有挂起的命令。
void pop3_free( );

// pop3_free_ex 函数针对特定会话释放 POP3
// 服务器，并取消所有挂起的命令。
void pop3_free_ex(POP3 *pppop3);

// pop3_list 函数列出 POP3
// 服务器上的邮件。它返回该服务器上
// 存在的邮件总数。
long pop3_list(char *transaction, [char *save_to_param,] LAST);

// pop3_list_ex 函数针对特定会话
// 列出 POP3 服务器上的邮件。它返回该服务器上
// 存在的邮件总数。
long pop3_list_ex (POP3 *pppop3, char *transaction, LAST);

// pop3_retrieve 函数从 POP3 服务器
// 检索邮件。您可以指定邮件范围或
// 所有邮件。默认情况下，它在检索邮件之后
```

```
// 将其从服务器中删除。
long pop3_retrieve(char *transaction, char *retrieveList, < Options, > LAST);

// pop3_retrieve_ex 函数从 POP3 服务器
// 检索邮件。您可以指定邮件范围或
// 所有邮件。默认情况下，它在检索邮件之后
// 将其从服务器中删除。
long pop3_retrieve_ex(POP3 *pppop3, char *transaction, char *retrieveList,   char * deleteFlag,
[<Options>,] LAST);

// lreal_clip_size 函数返回当前与
// 播放器关联的剪辑的大小，单位为毫秒。
unsigned long lreal_clip_size(int miPlayerID);

// lreal_close_player 函数关闭
// RealPlayer 的指定示例。
int lreal_close_player ( int miplayerID );

// lreal_current_time 用于查明剪辑已运行
// 多长时间。返回的时间以毫秒为单位。
unsigned long lreal_current_time( int miplayerID );

// lreal_get_property 获取播放器的属性。
int lreal_get_property ( int miPlayerID, unsigned int miProperty );

// lreal_open_player 函数创建新的 RealPlayer 实例。
int lreal_open_player ( int miplayerID );

// lreal_open_url 函数将 URL 与
// RealPlayer 实例相关联。使用 lreal_play 向
// 实例发出命令让其播放时，将显示此处
// 指定的 szURL。
int lreal_open_url ( int miplayerID, LPSTR szURL );

// lreal_pause 函数将 RealPlayer 实例
// 暂停一段指定的时间（单位为毫秒）。此函数
// 模拟 RealPlayer 的“播放”菜单中的“暂停”命令。
int lreal_pause ( int miplayerID, unsigned long mulPauseTime );

// lreal_play 函数播放 RealPlayer 剪辑
// 一段指定的时间（单位为毫秒）。此函数
// 模拟 RealPlayer 的“播放”菜单中的“播放”命令。
int lreal_play ( int miplayerID, long mulTimeToPlay );
```

```
// lreal_seek 函数搜寻当前剪辑中的
// 指定位置。此函数模拟 RealPlayer 的
// “播放”菜单中的“搜寻至位置”命令。注意，
// 您必须输入以毫秒为单位的时间。
int lreal_seek ( int miplayerID, unsigned long mulTimeToSeek );

// lreal_stop 函数停止播放 RealPlayer 实例。此
// 函数模拟 RealPlayer 的“播放”菜单中的“停止”命令。
int lreal_stop ( int miplayerID);

// TE_connect 函数在您录制与主机
// 的连接时由 VuGen 生成。使用 com_string 的
// 内容连接到主机。
int TE_connect ( const char *com_string, unsigned int timeout );

// TE_find_text 搜索与通过 col1,
// row1, col2, row2 定义的矩形中的模式匹配的
// 文本。与模式匹配的文本将返回给
// match，实际的行与列位置将返回给
// retcol 和 retrow。搜索从矩形
// 左上角开始。
int TE_find_text ( const char *pattern, int col1, int row1, int col2, int row2, int *retcol, int *retrow,
char *match );

// TE_get_cursor_pos 返回当前鼠标在终端仿真器
// 屏幕上的位置的坐标。
int TE_get_cursor_pos ( int *col, int *row );

// TE_get_line_attribute 检查终端屏幕中
// 一行文本的格式。行中的第一个字符
// 由 col, row 定义。行中最后一个
// 字符的列坐标由 Width
// 指定。该函数将每个字符的字符格式存储在
// 缓冲区 buf 中。
char * TE_get_line_attribute ( int col, int row, int width, char *buf );

// TE_get_text_line 将一行文本从终端屏幕复制到
// 缓冲区。行中的第一个字符
// 由 col, row 定义。行中最后一个
// 字符的列坐标由 Width
// 指定。如果该行包含制表符或空格，将返回相同
// 数目的空格。
char * TE_get_text_line ( int col, int row, int width, char *text );
```

```
// TE_getvar 函数返回 RTE 系统变量的值。
int TE_getvar ( int var );

// TE_set_cursor_pos 将鼠标位置设置为 col, row。
int TE_set_cursor_pos( int col, int row );

// TE_setvar 函数设置 RTE 系统变量。
int TE_setvar ( int var, int val );

// TE_perror 将 TE_errno 的当前值转换为
// 相应的错误字符串，设置字符串格式，并将其发送到
// Topaz 代理日志或 LoadRunner 输出窗口。
void TE_perror ( char *prefix );

// TE_sperror 将 TE_errno 的当前值转换为
// 相应的错误字符串。
char *TE_sperror ();

// TE_type 函数描述发送到
// 终端仿真器的键盘输入。
int TE_type ( const char *string );

// TE_unlock_keyboard 用于在因为
// 出现错误消息而导致大型机终端的键盘
// 被锁定之后解除锁定。TE_unlock_keyboard 等价于
// 按 F3 键。
int TE_unlock_keyboard ( void );

// TE_typing_style 函数确定键入的
// 字符串如何提交给在终端仿真器上运行的
// 客户端应用程序。如果选择 FAST，
// 将把字符作为单个字符串发送。
// 此输入方式不需要参数。
int TE_typing_style ( const char *style );

// 回放期间，TE_wait_cursor 等待鼠标出现在
// 终端窗口中的指定位置。
int TE_wait_cursor ( int col, int row, int stable, int timeout );

// TE_wait_silent 函数等待客户端
// 应用程序静默指定的时间。当终端仿真器
// 未接到任何字符时，认为客户端处于
// 静默状态。如果客户端应用程序
// 由于过了超时时间（以秒为单位）
```

```
// 而未静默认指定的时间，该函数将
// 返回错误。
int TE_wait_silent ( int sec, int milli, int timeout );

// 执行期间，TE_wait_sync 函数暂停
// 脚本执行，并等待“X SYSTEM”消息
// 从屏幕上消失之后再继续。出现
// “X SYSTEM”消息表示系统处于“内部输入”
// 模式。
int TE_wait_sync (void);

// 您可以指示 VuGen 录制每次
// 进入 X SYSTEM 模式时系统停留在 X SYSTEM
// 模式的时间。要这样做，VuGen 在每个 TE_wait_sync 函数之后插入
// TE_wait_sync_transaction 函数。
int TE_wait_sync_transaction (char *transaction_name );

// TE_wait_text 函数等待与通过 col1,
// row1, col2, row2 定义的矩形中
// 的模式匹配的文本。与模式
// 匹配的文本将返回给 match，实际的
// 行和列位置返回给 retcol 和
// retrow。如果模式超时时间已过而未
// 显示模式，该函数将返回错误
// 代码。如果模式已显示在屏幕上，
// 该函数将立即返回。
int TE_wait_text ( const char *pattern, int timeout [, int col1, int row1, int col2, int row2, int
*retcol, int *retrow, char *match ] );

// TE_run_script_command 执行 PSL 命令。
int TE_run_script_command ( const char *command );

// TE_run_script_file 运行 PSL 脚本文件。
int TE_run_script_file ( const char *filename );

// sapgui_active_object_from_parent_method 函数
// 使用 ID 编号 control_id 从大的
// 父级对象中选择对象。嵌入的
// 对象由方法 method_name 返回。
int sapgui_active_object_from_parent_method ( const char *control_id, const char *method_name,
char *arg1, ..., char *argn, [optionArguments,] LAST );

// sapgui_active_object_from_parent_property 函数
// 使用 ID 编号 control_id 从大的
```

```
// 父级对象中选择对象。嵌入的
// 对象由属性 property_name 返回。
int sapgui_active_object_from_parent_property (const char *control_id, const char
*property_name, [args,]  LAST);

// sapgui_calendar_focus_date 将焦点置于
// 日历中的日期上。从日历中选择
// 日期时，将自动录制此函数。
// 但是，实际返回日期的函数是
// sapgui_calendar_select_interval。
int sapgui_calendar_focus_date(const char *description, const char *calendarID, const char *date,
[args,]  LAST ) ;

// sapgui_calendar_scroll_to_date 模拟使用
// 滚动栏使日期可见的操作。它不是通过
// 将焦点置于日期上来选择日期。
int sapgui_calendar_scroll_to_date(const char *description, const char *calendarID, const char
*date, [args,]  LAST ) ;

// sapgui_calendar_select_interval 从日历中
// 将日期间隔返回给调用日历
// 弹出框时焦点所在的控件。它等价于
// 在不显示日历的情况下，将控件文本
// 设置为日期字符串。
int sapgui_calendar_select_interval(const char *description, const char *calendarID, const char
*interval, [args,]  LAST ) ;

// sapgui_call_method 函数使用 SAP 标识符
// control_id 来标识 SAP 对象，并调用
// 对象的方法 method_name。它向方法传递
// 实际参数 arg1...argn。
int sapgui_call_method ( const char *control_id, const char *method_name, void *arg1, ..., void
*argn, [optionalArguments]  LAST );

// sapgui_call_method_of_active_object 函数调用
// 通过 sapgui_active_object_from_parent_method
// 或 sapgui_active_object_from_parent_property
// 选择的当前活动对象的方法 method_name。
int sapgui_call_method_of_active_object ( const char *method_name, char *arg1, [args,]  char
*argn, [optionalArguments,]  LAST );

// sapgui_create_new_session 创建一个新会话。它等价于
// 从“系统”菜单中选择“创建会话”。
int sapgui_create_new_session([optionalArgs,] LAST );
```

```
// sapgui_get_active_window_title 数据检索函数
// 获取当前 SAP 会话中活动窗口的名称，
// 并保存到 output_param_name 中。
int sapgui_get_active_window_title (const char *output_param_name, [args,]  LAST);

// sapgui_get_ok_code 数据检索函数获取
// “命令”字段的文本。“命令”字段是主窗口中
// 第一个工具栏左边的框。
int sapgui_get_ok_code( const char *outParamName, [args,]  LAST);

// sapgui_get_property 数据检索函数获取
// ID 编号为 control_id 的 SAP 对象中
// 指定属性 property_name 的值。
// 该值保存在 output_param_name 中。
int sapgui_get_property ( const char *control_id, const char *property_name, char
*output_param_name, [args,]  LAST);

// sapgui_get_property_of_active_object 数据检索
// 函数从当前活动对象中检索
// 指定属性 property_name 的值。
// 该值保存在 output_param_name 中。
int sapgui_get_property_of_active_object ( const char *property_name, const char
*output_param_name, [args,]  LAST );

// sapgui_get_text 数据检索函数
// 获取任意可视屏幕对象的文本属性，
// 并保存到参数 outParamName 中。
int sapgui_get_text(const char *description, const char *controlID, const char *outParamName,
[args,]  LAST);

// sapgui_grid_fill_data 在网格中输入表格数据
// 参数。当您在网格中输入数据并按 Enter 时，
// 录制此函数。表格参数 paramName 是
// 自动创建的。录制之后，可以在 VuGen 中编辑此表格参数，
// 以便更改数据。
int sapgui_grid_fill_data( const char *description, const char *gridID, const char *paramName,
[args,]  LAST );

// sapgui_grid_clear_selection 取消选择已在网格控件中
// 选定的所有单元格、行和列。
int sapgui_grid_clear_selection(const char *description, const char *gridID, [args,]  LAST );

// sapgui_grid_click 模拟用户在通过“row”或
```

```
// “column”指定的网格单元格中单击。
int sapgui_grid_click(const char *description, const char *gridID, const char *row, const char
*column, [args,]   LAST );

// sapgui_grid_click_current_cell 模拟用户
// 在当前选定的单元格中单击。
int sapgui_grid_click_current_cell(const char *description, const char *gridID, [args,]   LAST );

// sapgui_grid_deselect_column 取消选择某列。
int sapgui_grid_deselect_column(const char *description, const char *gridID, const char* column,
[args,]   LAST );

// sapgui_grid_double_click 模拟用户在
// 网格中的单元格中双击。
int sapgui_grid_double_click(const char *description, const char *gridID, const char *row, const
char *column, [args,]   LAST );

// sapgui_grid_double_click_current_cell 模拟用户在
// 当前选定的单元格中双击。
int sapgui_grid_double_click_current_cell(const char *description, const char *gridID, [args,]
LAST );

// sapgui_grid_get_cell_data 数据检索函数
// 获取网格单元格中的数据，并保存到 outParamName 中。
int sapgui_grid_get_cell_data(const char *description, const char *gridID, const char *row, const
char *column, const char *outParamName, [args,]   LAST );

// sapgui_grid_get_columns_count 数据检索
// 函数获取网格中的列数。
int sapgui_grid_get_columns_count(const char *description, const char *gridID, const char
*outParamName, [args,]   LAST );

// sapgui_grid_get_current_cell_column 数据
// 检索函数获取当前活动单元格的列标识符，
// 并保存到 outparamName 中。
int sapgui_grid_get_current_cell_column(const char *description, const char *gridID, const char
*outParamName, [args,]   LAST );

// sapgui_grid_get_current_cell_row 数据
// 检索函数获取当前活动单元格的行号，
// 并保存到 outparamName 中。
int sapgui_grid_get_current_cell_row(const char *description, const char *gridID, const char
*outParamName, [args,]   LAST );
```

```
// sapgui_grid_get_rows_count 数据检索
// 函数获取网格中的行数。
int sapgui_grid_get_rows_count(const char *description, const char *gridID, const char
*outParamName, [args,]  LAST );

// sapgui_grid_is_checkbox_selected 验证
// 函数在复选框被选中时返回 true，
// 在复选框未被选中时返回 false。
int sapgui_grid_is_checkbox_selected(const char *description, const char *gridID, const char
*row, const char *column, [args,]  LAST );

// sapgui_grid_open_context_menu 模拟用户
// 在网格中右键单击以便打开上下文菜单。
int sapgui_grid_open_context_menu(const char *description, const char *gridID, [args,]
LAST );

// sapgui_grid_press_button 函数单击网格单元格中的按钮。
int sapgui_grid_press_button(const char *description, const char *gridID, const char *row, const
char *column, [args,]  LAST );

// sapgui_grid_press_button_current_cell 模拟用户
// 单击当前活动的网格单元格中的按钮。
int sapgui_grid_press_button_current_cell(const char *description, const char *gridID, [args,]
LAST );

// sapgui_grid_press_column_header 模拟用户
// 单击网格控件中的列标题。
int sapgui_grid_press_column_header(const char *description, const char *gridID, const char
*column,  [args,]  LAST );

// sapgui_grid_press_ENTER 模拟用户
// 在网格处于活动状态时按 Enter 键。
int sapgui_grid_press_ENTER (const char *description, const char *gridID, [args,]  LAST );

// sapgui_grid_press_F1 模拟用户
// 在网格处于活动状态时按 F1 键。其结果是
// 显示上下文相关帮助。
int sapgui_grid_press_F1 (const char *description, const char *gridID, [args,]  LAST );

// sapgui_grid_press_F4 模拟用户在网格处于
// 活动状态时按 F4 键。其结果通常是
// 显示活动字段可能的选项。
int sapgui_grid_press_F4 (const char *description, const char *gridID, [args,]  LAST );
```

```
// sapgui_grid_press_toolbar_button 模拟
// 用户单击网格工具栏按钮。
int sapgui_grid_press_toolbar_button(const char *description, const char *gridID, const char
*buttonID, [args,]  LAST );

// sapgui_grid_press_toolbar_context_button 模拟用户
// 通过单击上下文按钮打开选项列表。
int sapgui_grid_press_toolbar_context_button(const char *description, const char *gridID, const
char *buttonID, [args,]  LAST );

// sapgui_grid_press_total_row 函数按
// 网格单元格中的总计行按钮。如果总计
// 行被精简了，该函数将展开它。如果总计行
// 已展开，该函数将精简它。
int sapgui_grid_press_total_row(const char *description, const char *gridID, const char *row,
const char *column, [args,]  LAST );

// sapgui_grid_press_total_row_current_cell 函数按
// 当前活动网格单元格中的总计行按钮。
// 如果总计行被精简了，该函数将展开它。
// 如果总计行已展开，该函数将精简它。
int sapgui_grid_press_total_row_current_cell(const char *description, const char *gridID, [args,]
LAST );

// sapgui_grid_scroll_to_row 滚动到网格中的指定行，
// 从而使其可见。它不会选择行。
int sapgui_grid_scroll_to_row(const char *description, const char *gridID, const char *row, [args,]
LAST ) ;

// sapgui_grid_select_all 选择网格控件中的所有单元格。
int sapgui_grid_select_all(const char *description, const char *gridID, [args,]  LAST );

// sapgui_grid_select_cell 选择网格控件中的单个单元格。
int sapgui_grid_select_cell(const char *description, const char *gridID, const char *row, const
char *column, [args,]  LAST );

// sapgui_grid_select_cell_column 选择与当前选择
// 位于同一行但位于不同列的单元格。
// 它在同一行中向左或向右移动选择，以便移动到
// 由参数 column 指定的列号。
int sapgui_grid_select_cell_column(const char *description, const char *gridID, const char
*column, [args,]  LAST );

// sapgui_grid_select_cell_row 选择与当前选择
```

```
// 位于同一列但位于不同行的单元格。
// 它在同一列中向上或向下移动选择，以便移动到
// 由参数 row 指定的行号。
int sapgui_grid_select_cell_row(const char *description, const char *gridID, const char *row,
[args,]   LAST );

// sapgui_grid_select_cells 函数选择
// 网格中的单元格。单元格列表以
// “LAST”或“BEGIN_OPTIONAL”结束。
int sapgui_grid_select_cells(const char *description, const char *gridID, const char *cell1 ,...,
celln, [args,]   LAST );

// sapgui_grid_select_column 选择网格控件中的一列。
int sapgui_grid_select_column ( const char *description, const char *gridID, const char* column,
[optionalArgs,] LAST );

// sapgui_grid_select_columns 选择网格控件中的
// 一列或多列。这些列不必相邻，
// 也不必按照网格中的显示顺序传递给
// 函数。
int sapgui_grid_select_columns(const char *description, const char *gridID, char *arg1, ..., char
*argn, [optionalArgs,] LAST);

// sapgui_grid_select_context_menu 模拟用户
// 从上下文菜单中选择选项。
int sapgui_grid_select_context_menu(const char *description, const char *gridID, const char
*functionCode, [args,]   LAST );

// sapgui_grid_select_rows 选择网格单元格中的一行或多行。
int sapgui_grid_select_rows(const char *description, const char *gridID, const char *rows, [args,]
LAST );

// sapgui_grid_select_toolbar_menu 模拟用户
// 从网格工具栏菜单中选择选项。
int sapgui_grid_select_toolbar_menu(const char *description, const char *gridID, const char
*functionCode, [args,]   LAST );

// sapgui_grid_selection_changed 验证函数
// 返回选择是否已发生更改。
int sapgui_grid_selection_changed(const char *description, const char *gridID, [args,]   LAST );

// sapgui_grid_set_cell_data 在网格单元格中插入数据。字符串
// newValue 被写入位于 row 和 column 的单元格中。
int sapgui_grid_set_cell_data(const char *description, const char *gridID, const char *row, const
```

```
char *column, const char *newValue, [args,]    LAST );

// sapgui_grid_set_checkbox  函数选择或清除网格复选框。
int sapgui_grid_set_checkbox(const char *description, const char *gridID, const char *row, const
char *column, const char *newValue, [args,]    LAST );

// sapgui_grid_set_column_order  模拟用户
//  将列拖动到网格中的新位置。
int sapgui_grid_set_column_order(const char *description, const char *gridID, const char *row,
const char *columns, [args,]    LAST );

// sapgui_grid_set_column_width  模拟用户
//  拖动列边框以便设置新宽度。
int sapgui_grid_set_column_width(const char *description, const char *gridID, const char*
column, const char* width, [args,] LAST);

// sapgui_htmlviewer_send_event  发送  HTML  事件。
int sapgui_htmlviewer_send_event(const char *htmlViewerID, const char *frame, const char
*data, const char *url, [args,]    LAST ) ;

// sapgui_is_checkbox_selected  验证函数返回
//  复选框的状态。如果复选框被选中，该函数返回  True。
//  如果复选框未被选中，该函数返回  False。
int sapgui_is_checkbox_selected(const char *description, const char *checkBoxID, [args,]
LAST );

// sapgui_is_object_available  验证函数检查
//  指定对象是否可用在函数中。
int sapgui_is_object_available (const char *object_name, [args,]    LAST);

// sapgui_is_object_changeable  验证函数返回
//  组件是否可被修改。如果组件既未被禁用，也不是只读，
//  则可被修改。
int sapgui_is_object_changeable(const char *controlID, [args,]    LAST);

// sapgui_is_radio_button_selected  验证函数
//  返回单选按钮是否已被选中。
int sapgui_is_radio_button_selected(const char *description, const char *buttonID,[args,]
LAST );

// sapgui_is_tab_selected  验证函数返回一个布尔值，
//  指明当前是否选择了选项卡  tabID。
int sapgui_is_tab_selected(const char *description, const char *tabID, [args,]    LAST);
```

```
// sapgui_logon 登录到 SAP 服务器。
int sapgui_logon ( const char *user_name, const char *password, const char *client_num, const
char *language, [args,]  LAST );

// sapgui_open_connection 函数打开
// 由 connection_name 定义的连接。如果在
// 现有的 SAP 客户端定义中未找到
// connection_name，该函数将试图连接到
// 使用该名称的服务器。
int sapgui_open_connection (const char *connection_name, const char *connection_id, [args,]
LAST );

// sapgui_open_connection_ex 函数打开
// 与由 connection_string 定义的服务器的连接。此函数
// 设置输出参数 connection_id。此参数用在
// sapgui_select_active_connection 中。
int sapgui_open_connection_ex (const char *connection_string, const char *connection_name,
const char *connection_id, [args,]  LAST );

// sapgui_press_button 模拟用户单击按钮 buttonID。
int sapgui_press_button( const char *description, const char *buttonID, [args,]  LAST );

// sapgui_select_active_connection 指定一个打开的服务器连接
// 作为工作连接。
int sapgui_select_active_connection(const char *connectionID);

// sapgui_select_active_session 从打开的会话集合中
// 选择当前工作会话。
int sapgui_select_active_session(const char *sessionID);

// 录制时，只要您在不属于前一活动
// 窗口的控件上执行操作，就会自动
// 生成 sapgui_select_active_window
// 语句。随后的所有操作都将在此
// 窗口上执行，直到调用下一个 sapgui_select_active_window
// 语句。
int sapgui_select_active_window ( const char *window_id );

// sapgui_select_combobox_entry 从组合框中选择项目 entryKey。
int sapgui_select_combobox_entry(const char *description, const char *objectID, const char
*entryKey, [args,]  LAST );

// sapgui_select_menu 模拟用户打开
// 菜单并选择菜单项目。
```

```
int sapgui_select_menu(const char *description, const char *menuID, [args,]   LAST );

// sapgui_select_radio_button 从组中选择一个单选按钮，
// 并清除该组中的所有其他按钮。
int sapgui_select_radio_button(const char *description, const char *buttonID, [args,]   LAST );

// sapgui_select_tab 激活一个选项卡页。它
// 模拟用户单击选项卡 tabID。
int sapgui_select_tab(const char *description, const char *tabID, [args,]   LAST );

// sapgui_send_vkey 函数通过发送
// 虚拟键来模拟键盘。
int sapgui_send_vkey(const char *key, [args,]   LAST );

// 如果 isSelected 为“True”，那么 sapgui_set_checkbox 将选中
// 复选框。如果 isSelected 为“False”，将清除该复选框。
int sapgui_set_checkbox(const char *description, const char *isSelected, const char *checkBoxID,
[args,]   LAST );

// sapgui_set_collection_property 函数使用组成
// GuiCollection 对象的字符串列表项目 arg1...argn
// 设置指定的属性 property_name。
int sapgui_set_collection_property ( const char *control_id, const char *property_name, char
*arg1, ..., char *argn, [optionalArgs,] LAST );

// sapgui_set_focus 将焦点置于对象 controlID 上。
int sapgui_set_focus(const char *controlID, [args,]   LAST );

// sapgui_set_ok_code 在“命令”字段中输入文本。
// “命令”字段是主窗口中第一个工具栏
// 左边的框。文本必须是 SAP 命令。
int sapgui_set_ok_code(const char *text, [args,]   LAST );

// sapgui_set_password 在“密码”文本框中输入
// 密码。录制函数时，密码文本被
// 隐藏。要回放脚本，请编辑函数并输入
// 密码。
int sapgui_set_password(const char *description, const char *password, const char *controlID,
[args,]   LAST );

// sapgui_set_property 函数使用 ID 编号为
// control_id 的 SAP 对象中的值 new_value 设置
// 指定属性 property_name。
int sapgui_set_property ( const char *control_id, const char *property_name, const char
```

```
*new_value, [args,]   LAST );

// sapgui_set_property_of_active_object 函数使用
// 值 new_value 设置当前活动对象的指定
// 属性 property_name。当前对象通过
// sapgui_active_object_from_parent_method 或
// sapgui_active_object_from_parent_property 选择。
int sapgui_set_property_of_active_object ( const char *property_name, const char *new_value,
[args,]   LAST );

// sapgui_set_text 将 text 参数的值写入
// 控件。如果 text 为文字，请将它放置在引号中：
// “The text to enter”。如果 text 是文件，
// 语法为 “file=filename.ext”。引号是语法的
// 组成部分。文件必须位于
// script 文件夹中。不能使用路径。
int sapgui_set_text(const char *description, const char *text, const char *controlID, [args,]
LAST );

// sapgui_status_bar_get_param 数据检索函数
// 从状态栏获取位置 paramIndex 中的参数，
// 并将其存储在参数 outputParamName 中。
int sapgui_status_bar_get_param(const char *paramIndex, const char *outputParamName, [args,]
LAST );

// sapgui_status_bar_get_text 数据检索函数
// 从状态栏获取文本，并将其存储在参数
// outputParamName 中。将检索用户看到的整个文本，
// 包括固定文本和参数。
int sapgui_status_bar_get_text(const char *outputParamName, [args,]   LAST );

// 成功调用 sapgui_status_bar_get_type 后，outputParamName 的值
// 为以下文字字符串之一
// ：“Success”、“Warning”或“Error”。该函数
// 用于测试上一操作的结果。
int sapgui_status_bar_get_type(const char *outputParamName, [args,]   LAST );

// 当您在表中输入数据并按 Enter 键时录制 sapgui_table_fill_data
// 。将自动创建表参数 paramName
// 。录制后可以在 VuGen 中编辑表参数
// 以便更改数据。
int sapgui_table_fill_data(const char *description, const char *tableID, const char *paramName,
[args,]   LAST );
```

```
// sapgui_table_get_column_width 数据检索函数
// 将列的宽度放到参数 paramName 中。
int sapgui_table_get_column_width(const char *description, const char *tableID, const char
*column, const char *paramName, [args,]  LAST );

// sapgui_table_get_text 数据检索函数将由
// row 和 column 指定的单元格中的文本放到参数 paramName 中。
int sapgui_table_get_text(const char *description, const char *tableID, const char *row, const char
*column, const char *paramName, [args,]  LAST );

// 如果选中列，则 sapgui_table_is_column_selected 返回 True。
int sapgui_table_is_column_selected ( char * tableID, char * column, [args,]  LAST );

// sapgui_table_is_checkbox_selected 验证函数
// 适用于表控件中的复选框。
// 它返回是选中还是清除了该复选框。
int sapgui_table_is_checkbox_selected(const char *description, const char *tableID, const char
*row, const char *column, [args,]  LAST );

// 如果选中表中的单选按钮，
// 则 sapgui_table_is_radio_button_selected 验证函数返回 True。
int sapgui_table_is_radio_button_selected(const char *description, const char *tableID, const char
*row, const char *column, [args,]  LAST );

// sapgui_table_is_row_selected 验证函数
// 适用于表控件中的行。
// 它返回行是否被选中。
int sapgui_table_is_row_selected(const char *tableID, const char *row, [args,]  LAST );

// sapgui_table_press_button 适用于表单元格中的
// 按钮控件。它模拟用户单击由 row 和 column 指定的
// 单元格中的按钮。
int sapgui_table_press_button(const char *description, const char *tableID, const char *row, const
char *column, [args,]  LAST );

// sapgui_table_reorder 对表中的列进行重新排序。
// order 参数是新顺序中
// 列的原有位置。使用空格来分隔列编号
// 。例如，“0 1 3 2”会将第 4 列
// （列 3）移动到第 3 个位置。
int sapgui_table_reorder ( const char *description, const char *tableID, const char *order, [args,]
LAST );

// sapgui_table_select_combobox_entry 从组合框中
```

```
// 选择项目 entryKey，并在由 row 和 column 指定的
// 表单元格中输入该值。
int sapgui_table_select_combobox_entry (const char *description, const char *tableID, const char
*row, const char *column, const char *entryKey, [args,]  LAST );

// sapgui_table_select_radio_button 适用于表单元格中的
// 单选按钮控件。它模拟用户选中由 row 和 column 指定
// 的单元格中的按钮。
int sapgui_table_select_radio_button(const char *description, const char *tableID, const char *row,
const char *column, [args,]  LAST );

// sapgui_table_set_checkbox 设置表中复选框
// 的状态。如果 newValue 为“True”，则该复选框处于选中状态。
// 如果 newValue 为“False”，则该复选框处于清除状态。
int sapgui_table_set_checkbox(const char *description, const char *tableID, const char *row,
const char *column, const char *newValue, [args,]  LAST );

// sapgui_table_set_column_selected 模拟用户
// 单击表中的列标题。
int sapgui_table_set_column_selected(const char *description, const char *tableID, const char
*column, const char *isSelected, [args,]  LAST );

// sapgui_table_set_column_width 模拟用户
// 拖动表中列标题的边缘。
int sapgui_table_set_column_width ( const char *description, const char *tableID, const char
*column, const char *width, [args,]  LAST );

// sapgui_table_set_focus 模拟用户在由
// row 和 column 指定的表单元格中单击。
int sapgui_table_set_focus (const char *description, const char *tableID, const char *row, const
char *column, const char *newValue, [args,]  LAST );

// sapgui_table_set_password 在表单元格密码字段中
// 设置密码。
int sapgui_table_set_password(const char *description, const char *tableID, const char *row,
const char *column, const char *password, [args,]  LAST );

// sapgui_table_set_row_selected 设置表中行的
// 选中状态。如果 isSelected 为“True”，
// 则行号为 row 的行处于选中状态。如果 isSelected
// 为“False”，则该行处于未选中状态。
int sapgui_table_set_row_selected(const char *tableID, const char *row, const char *isSelected,
[args,]  LAST );
```

```
// sapgui_table_set_text 在由 row 和 column 指定的
// 单元格中输入字符串文本。
int sapgui_table_set_text(const char *description, const char *tableID, const char *row, const char *column, const char *text, [args,]  LAST );

// sapgui_text_edit_scroll_to_line 模拟用户滚动，
// 直到指定行可见。但不选中该行。
int sapgui_text_edit_scroll_to_line( const char *description, const char *textEditID, const char *lineNumber, [args,]  LAST );

// sapgui_text_edit_set_selection_indexes 设置
// 选择的可视文本范围。参数 start 和 end 都是基于
// 零的字符索引。参数 start 对应
// 所选内容的开始位置，而参数 end 是
// 所选内容之后第一个字符的位置。
int sapgui_text_edit_set_selection_indexes( const char *description, const char *textEditID, const char *startNumber, const char *endNumber, [args,]  LAST );

// sapgui_text_edit_set_unprotected_text_part
// 通过基于零的索引将 text 的内容分配给未受保护的
// 文本部分，即 part。
int sapgui_text_edit_set_unprotected_text_part( const char *description, const char *textEditID, const char *part, const char *text, [args,]  LAST );

// sapgui_text_edit_get_first_visible_line 将在控件
// 顶部边框可见的第一行的编号
// 分配给 outParamName。行从 1 开始编号。
int sapgui_text_edit_get_first_visible_line( const char *description, const char *textEditID, const char *outParamName, [args,]  LAST );

// sapgui_text_edit_get_selection_index_start 将
// 选定范围开始位置的基于零的字符索引
// 分配给参数 outParamName。
int sapgui_text_edit_get_selection_index_start( const char *description, const char *textEditID, const char *outParamName, [args,]  LAST );

// sapgui_text_edit_get_selection_index_end 将
// 选定范围结束位置的基于零的字符索引
// 分配给参数 outParamName。这是所选内容之后第一个字符的
// 位置。
int sapgui_text_edit_get_selection_index_end( const char *description, const char *textEditID, const char *outParamName, [args,]  LAST );

// sapgui_text_edit_get_number_of_unprotected_text_parts 将
```

```
// 未受保护文本部分的编号分配给 outParamName。
int sapgui_text_edit_get_number_of_unprotected_text_parts( const char *description, const char
*textEditID, const char *outParamName, [args,]  LAST );

// sapgui_text_edit_double_click 模拟鼠标双击。
// 要设置所选内容，请在 sapgui_text_edit_double_click 之前
// 调用 sapgui_text_edit_set_selection_indexes。
int sapgui_text_edit_double_click( const char *description, const char *textEditID, [args,]
LAST );

// sapgui_text_edit_single_file_dropped 模拟
// 将 fileName 拖放到文本编辑控件中。
int sapgui_text_edit_single_file_dropped( const char *description, const char *textEditID, const
char * fileName, [args,]  LAST );

// sapgui_text_edit_multiple_files_dropped 模拟
// 将 listOfFiles 中的文件
// 拖放到文本编辑控件中。
int sapgui_text_edit_multiple_files_dropped( const char *description, const char *textEditID,
listOfFiles, [args,]  LAST );

// sapgui_text_edit_press_F1 调用
// 在应用程序中针对文本编辑控件定义的
// 上下文相关帮助。如果未定义帮助，
// 则 sapgui_text_edit_press_F1 不起作用。
int sapgui_text_edit_press_F1( const char *description, const char *textEditID, [args,]  LAST );

// sapgui_text_edit_press_F4 调用
// 应用程序中针对文本编辑控件定义的选项列表。如果未
// 定义选项，则 sapgui_text_edit_press_F4 不起作用。
int sapgui_text_edit_press_F4( const char *description, const char *textEditID, [args,]  LAST );

// sapgui_text_edit_open_context_menu 打开
// 应用程序中针对文本编辑控件定义的
// 快捷菜单。如果未定义菜单，则 sapgui_text_edit_open_context_menu
// 不起作用。
int sapgui_text_edit_open_context_menu( const char *description, const char *textEditID, [args,]
LAST );

// sapgui_text_edit_select_context_menu 选中
// 由 menuId 指定的快捷菜单项。
// menuId 是项目的函数代码。
int sapgui_text_edit_select_context_menu( const char *description, const char *textEditID, const
char *menuId, [args,]  LAST );
```

```
// sapgui_text_edit_modified_status_changed 设置
// 文本编辑控件的已修改状态。
// 其值可以是“True”，也可以是“False”。
int sapgui_text_edit_modified_status_changed( const char *description, const char *textEditID,
const char *value, [args,]  LAST );

// sapgui_toolbar_press_button 模拟在工具栏按钮上进行单击操作。
int sapgui_toolbar_press_button( const char * description, const char * toolbarID, const char *
buttonID, [args,]  LAST );

// sapgui_toolbar_press_context_button 模拟
// 用户按工具栏上下文按钮。
int sapgui_toolbar_press_context_button(const char * description, const char * toolbarID, const
char * buttonID, [args,]  LAST );

// sapgui_toolbar_select_menu_item 模拟
// 用户选择工具栏菜单项。
int sapgui_toolbar_select_menu_item( const char * description, const char * toolbarID, const char
* menuID, [args,]  LAST );

// sapgui_toolbar_select_menu_item_by_text 模拟
// 用户选择工具栏菜单项。
int sapgui_toolbar_select_menu_item_by_text( const char * description, const char * toolbarID,
const char * menuItem, [args,]  LAST );

// sapgui_toolbar_select_context_menu_item 模拟
// 用户选择上下文菜单项。
int sapgui_toolbar_select_context_menu_item( const char * description, const char * toolbarID,
const char * menuID, [args,]  LAST );

// sapgui_toolbar_select_context_menu_item_by_text
// 模拟用户选择上下文菜单项。
int sapgui_toolbar_select_context_menu_item_by_text( const char * description, const char *
toolbarID, const char * menuItem, [args,]  LAST );

// sapgui_tree_click_link 模拟用户单击树中的链接。
int sapgui_tree_click_link(const char *description, const char *treeID, const char *nodeKey, const
char *itemName, [args,]  LAST ) ;

// sapgui_tree_collapse_node 模拟用户单击
// “-”符号以折叠节点。调用 sapgui_tree_collapse_node 之后，
// “-”符号将替换为“+”符号，
// 而且子节点不可见。
```

int sapgui_tree_collapse_node(const char *description, const char *treeID, const char *nodeKey, [args,] LAST) ;

// sapgui_tree_double_click_item 模拟用户
// 双击树中的某一项。如果该项是一个链接，
// 则打开其目标；如果是命令，则执行该命令。
int sapgui_tree_double_click_item(const char *description, const char *treeID, const char *nodeKey, const char *itemName, [args,] LAST) ;

// sapgui_tree_double_click_node 模拟
// 用户双击树中的某个节点。
int sapgui_tree_double_click_node(const char *description, const char *treeID, const char *nodeKey, [args,] LAST);

// sapgui_tree_expand_node 模拟用户通过单击“+”符号
// 展开一个节点。调用 sapgui_tree_expand_node 之后，
// “+”符号将替换为“-”符号，
// 而且子节点可见。
int sapgui_tree_expand_node(const char *description, const char *treeID, const char *nodeKey, [args,] LAST) ;

// sapgui_tree_get_item_text 数据检索函数
// 将树中某一项的文本属性放到 outParamName 中。
int sapgui_tree_get_item_text(const char *description, const char *treeID, const char *nodeKey, const char *itemName, const char *outParamName, [args,] LAST) ;

// sapgui_tree_get_node_text 数据检索函数
// 将节点文本放到 outParamName 中。
int sapgui_tree_get_node_text(const char *description, const char *treeID, const char *nodeKey, const char *outParamName, [args,] LAST) ;

// sapgui_tree_is_checkbox_selected 验证函数
// 适用于树控件中的复选框。如果复选框处于选中状态，
// 则返回 True，如果复选框处于清除状态，则返回 False。
int sapgui_tree_is_checkbox_selected(const char *description, const char *treeID, const char *nodeKey, const char *itemName, [args,] LAST) ;

// sapgui_tree_open_default_context_menu 打开
// 树的默认上下文相关菜单。
int sapgui_tree_open_default_context_menu(const char *description, const char *treeID, [args,] LAST) ;

// sapgui_tree_open_header_context_menu 模拟用户
// 右键单击树标题以打开上下文相关菜单。

int sapgui_tree_open_header_context_menu(const char *description, const char *treeID, const char *headerName, [args,] LAST) ;

// sapgui_tree_open_item_context_menu 模拟用户
// 右键单击树项以打开上下文相关菜单。
int sapgui_tree_open_item_context_menu(const char *description, const char *treeID, const char *nodeKey, const char *itemName, [args,] LAST) ;

// sapgui_tree_open_node_context_menu 模拟用户
// 右键单击树节点以打开上下文相关菜单。
int sapgui_tree_open_node_context_menu(const char *description, const char *treeID, const char *nodeKey, [args,] LAST) ;

// sapgui_tree_press_button 模拟用户单击树中的按钮。
int sapgui_tree_press_button(const char *description, const char *treeID, const char *nodeKey, const char *itemName, [args,] LAST);

// sapgui_tree_press_header 模拟用户
// 单击树中的列标题。
int sapgui_tree_press_header(const char *description, const char *treeID, const char *headerName, [args,] LAST) ;

// sapgui_tree_press_key 模拟用户在
// 树具有焦点时按键盘。
int sapgui_tree_press_key(const char *description, const char *treeID, const char *key, [args,] LAST);

// sapgui_tree_scroll_to_item 与
// sapgui_tree_select_item 自动录制为一对。它模拟
// 使用滚动条，以便 itemName 可见。
int sapgui_tree_scroll_to_item(const char *description, const char *treeID, const char *nodeKey, const char *itemName, [args,] LAST) ;

// sapgui_tree_scroll_to_node 滚动树，以便
// topNode 成为窗格顶端
// 的第一个可见节点。但不选中该节点。
int sapgui_tree_scroll_to_node(const char *description, const char *treeID, const char *topNode, [args,] LAST);

// sapgui_tree_select_column 模拟用户
// 使用鼠标选择树列。
int sapgui_tree_select_column(const char *description, const char *treeID, const char *columnName, [args,] LAST) ;

```
// sapgui_tree_select_context_menu 从树控件的
// 快捷菜单中选择一项。
int sapgui_tree_select_context_menu ( const char *description, const char *treeID, const char
*value, [args,]  LAST );

// sapgui_tree_select_item 适用于树控件中
// 任何可选项。它模拟用户单击
// itemName 项以便选中该项。如果在调用 sapgui_tree_select_item 时
// 该项在树窗格中不可见，
// 则滚动树以使该项可见。
int sapgui_tree_select_item(const char *description, const char *treeID, const char *nodeKey,
const char *itemName, [args,]  LAST ) ;

// sapgui_tree_select_node 模拟用户单击
// MultipleNodeSelection 树中的节点。调用
// sapgui_tree_select_node 会将节点 nodeKey
// 添加到选定节点的集合。
int sapgui_tree_select_node(const char *description, const char *treeID, const char *nodeKey,
[args,]  LAST ) ;

// sapgui_tree_set_checkbox 适用于树控件中的
// 复选框。如果该值为“True”，则选中复选框；
// 如果该值为“False”，则清除复选框。
int sapgui_tree_set_checkbox(const char *description, const char *treeID, const char *nodeKey,
const char *itemName, const char *value, [args,]  LAST );

// sapgui_tree_set_column_order 设置树中
// 列的顺序。新顺序由列在列列表中
// 的位置来确定。
int sapgui_tree_set_column_order(const char *description, const char *treeID, const char
*columns, [args,]  LAST ) ;

// sapgui_tree_set_column_width 将 columnName 的宽度设置为 width。
int sapgui_tree_set_column_width(const char *description, const char *treeID, const char
*columnName, const char *width, [args,]  LAST ) ;

// sapgui_tree_set_hierarchy_header_width 将
// 树视图中标题的宽度更改为 width。
int sapgui_tree_set_hierarchy_header_width(const char *description, const char *treeID, const
char *width, [args,]  LAST );

// sapgui_tree_set_selected_node 模拟用户单击
// SingleNodeSelection 树中的节点。调用
// sapgui_tree_set_selected_node 将取消选中
```

```
// 先前选中的节点，并选中节点 nodeKey。
int sapgui_tree_set_selected_node(const char *description, const char *treeID, const char
*nodeKey, [args,]  LAST );

// sapgui_tree_unselect_all 取消选中树中
// 所有选中的项，而不选中其他项。
int sapgui_tree_unselect_all(const char *description, const char *treeID, [args,]  LAST) ;

// sapgui_tree_unselect_column 取消选中树列。
int sapgui_tree_unselect_column(const char *description, const char *treeID, const char
*columnName, [args,]  LAST ) ;

// sapgui_tree_unselect_node 取消选中树节点。
int sapgui_tree_unselect_node(const char *description, const char *treeID, const char *nodeKey,
[args,]  LAST ) ;

// sapgui_window_close 关闭 SAP GUI 应用程序。
int sapgui_window_close( LAST);

// sapgui_window_maximize 将窗口设置为全屏大小。
int sapgui_window_maximize( LAST);

// sapgui_window_resize 将活动窗口调整为 width 和 height 大小。
int sapgui_window_resize( const char *width, const char *height, [args,]  LAST );

// sapgui_window_restore 将窗口还原为非最大化状态。
int sapgui_window_restore( LAST);

// sapgui_window_scroll_to_row 滚动到某一窗口行，但
// 不在该行上设置焦点，也不针对该行执行任何操作
int sapgui_window_scroll_to_row( const char * newPosition, [args,]  LAST );

// sapgui_apogrid_clear_selection 取消选中
// APO 网格中当前选中的区域。
int sapgui_apogrid_clear_selection ( const char *description, const char *gridID, [args,] LAST ) ;

// sapgui_apogrid_deselect_cell 取消选中 APO 网格中
// 由“row”和“column”指定的单元格。
int sapgui_apogrid_deselect_cell ( const char *description, const char *gridID, const char *row,
const char *column, [args,]  LAST ) ;

// sapgui_apogrid_deselect_column 取消选中 APO 网格中的
// 指定列。
int sapgui_apogrid_deselect_column ( const char *description, const char *gridID, const char
```

*column, [args], LAST) ;

// sapgui_apogrid_deselect_row 取消选中 APO 网格中的指定行。
int sapgui_apogrid_deselect_row (const char *description, const char *gridID, const char *row, [args,] LAST) ;

// sapgui_apogrid_double_click 模拟用户
// 双击 APO 网格中的单元格。
int sapgui_apogrid_double_click (const char *description, const char *gridID, const char *row, const char *column, [args,] LAST) ;

// sapgui_apogrid_get_cell_data 将 APO 网格中
// 某单元格的数据分配给参数 outParamName。
// 数据始终以字符串形式保存。
int sapgui_apogrid_get_cell_data (const char *description, const char *gridID, const char *row, const char *column, const char *outParamName, [args,] LAST) ;

// sapgui_apogrid_get_cell_format 将 APO 网格中
// 某单元格的格式描述分配给参数 outParamName。
int sapgui_apogrid_get_cell_format (const char *description, const char *gridID, const char *row, const char *column, const char *outParamName, [args,] LAST) ;

// sapgui_apogrid_get_cell_tooltip 将 APO 网格中
// 某单元格的工具提示分配给参数 outParamName。
int sapgui_apogrid_get_cell_tooltip (const char *description, const char *gridID, const char *row, const char *column, const char *outParamName, [args,] LAST) ;

// sapgui_apogrid_is_cell_changeable 返回一个值，指明
// 用户是否可以修改 APO 网格中某单元格的值。
int sapgui_apogrid_is_cell_changeable (const char *description, const char *gridID, const char *row, const char *column, [args,] LAST) ;

// sapgui_apogrid_open_cell_context_menu 打开 APO 网格中
// 某单元格的上下文菜单。
int sapgui_apogrid_open_cell_context_menu (const char *description, const char *gridID, const char *row, const char *column, [args,] LAST) ;

// sapgui_apogrid_press_ENTER 模拟用户在 APO 网格中按 Enter 键。
int sapgui_apogrid_press_ENTER (const char *description, const char *gridID, [args,] LAST) ;

// sapgui_apogrid_scroll_to_column 模拟
// 用户滚动到 APO 网格中的某列。
// 该函数不选中该列。
int sapgui_apogrid_scroll_to_column (const char *description, const char *gridID, const char

*column, [args,] LAST) ;

// sapgui_apogrid_scroll_to_row 模拟用户
// 滚动到 APO 网格中的某行。该函数不选中该行。
int sapgui_apogrid_scroll_to_row (const char *description, const char *gridID, const char *row, [args,] LAST) ;

// sapgui_apogrid_select_all 全选 APO 网格中的单元格。
int sapgui_apogrid_select_all (const char *description, const char *gridID, [args,] LAST) ;

// sapgui_apogrid_select_cell 选中 APO 网格中的一个单元格。
int sapgui_apogrid_select_cell (const char *description, const char *gridID, const char *row, const char *column, [args,] LAST) ;

// sapgui_apogrid_select_column 选中 APO 网格中的一列。
int sapgui_apogrid_select_column (const char *description, const char *gridID, const char *column, [args,] LAST) ;

// sapgui_apogrid_select_context_menu_item 选中 APO 网格的
// 上下文菜单中的一项。
int sapgui_apogrid_select_context_menu_item (const char *description, const char *gridID, const char *value, [args,] LAST) ;

// sapgui_apogrid_select_row 选中 APO 网格中的一行。
int sapgui_apogrid_select_row (const char *description, const char *gridID, const char *row, [args,] LAST) ;

// sapgui_apogrid_set_cell_data 将 newValue
// 分配给 APO 网格中某单元格的内容。
int sapgui_apogrid_set_cell_data (const char *description, const char *gridID, const char *row, const char *column, const char *newValue, [args,] LAST) ;

// smtp_abort_mail 取消当前的邮件事务。
// 废弃所有已存储的发件人、收件人和邮件数据。
// 清除所有缓冲区和状态表。
int smtp_abort_mail( );

// smtp_abort_mail_ex 针对特定会话取消
// 当前邮件事务。废弃所有
// 已存储的发件人、收件人和邮件数据。
// 清除所有缓冲区和状态表。
int smtp_abort_mail_ex(SMTP *ppsmtp);

// smtp_free 函数释放 SMTP 服务器

```
// 并取消所有挂起命令。
void smtp_free( );

// smtp_free_ex 函数释放 SMTP 服务器
// 并取消所有挂起命令。
void smtp_free_ex (SMTP *ppsmtp);

// smtp_logon 函数登录到 SMTP 服务器。
// 它使用 FTP 协议所用的格式。
int smtp_logon (char *transaction, char *url, [ char *CommonName, char *LogonUser, char
*LogonPass,] LAST);

// smtp_logon_ex 函数针对特定会话
// 登录到 SMTP 服务器。
int smtp_logon_ex ( SMTP *ppsmtp, char *transaction, char *url, [ char *CommonName, char
*LogonUser, char *LogonPass,] LAST );

// smtp_logout 函数从 SMTP 服务器注销。
int smtp_logout( );

// smtp_logout_ex 函数针对特定会话
// 从 SMTP 服务器注销。
int smtp_logout_ex (SMTP *ppsmtp);

// smtp_send_mail 函数使用 SMTP 服务器发送一封邮件。
int smtp_send_mail (char *transaction, char *RecipientTo, [char *RecipientCC,] [char
*RecipientBCC,] char *Subject, [char *From,][char * ContentType, <char * charset,>] char
*MAILOPTIONS, char *MAILDATA, LAST);

// smtp_send_mail_ex 函数针对指定会话
// 使用 SMTP 服务器发送一封邮件。
int smtp_send_mail_ex (SMTP *ppsmtp, char *transaction, char *RecipientTo,  [char
*RecipientCC,] [char *RecipientBCC,] char *Subject,[char *From,] [char * ContentType, < char *
charset,>] char *MAILOPTIONS, char *MAILDATA, LAST);

// smtp_translate 函数为 SMTP 服务器
// 转换消息。应指定源文件和目标文件、
// 内容标题和编码类型。
int smtp_translate (char *filename, char *content_header, ENCODING_TYPE encoding, char
*output_filename);

// smtp_translate_ex 函数为 SMTP 服务器
// 转换消息。应指定源文件和目标文件、
// 内容标题和编码类型。
```

```
int smtp_translate_ex (SMTP *ppsmtp, char *filename, char *content_header,
ENCODING_TYPE encoding, char *output_filename);

// lrt_abort_on_error 函数
// 在上一个 Tuxedo 函数产生错误时
// 中止当前事务。这意味着事务异常
// 结束。将撤消事务执行过程中
// 对资源所做的所有更改。
void lrt_abort_on_error ( );

// lrt_ClarifyCorrelate 在使用 Clarify CRM 应用程序时
// 由 Vugen 自动生成，
// 并将返回的数据
// 保存在对数据库的 DB_OBJNUM 或 DB_OBJIDS 调用
// 的回复缓冲区中。该数据用于关联。
void lrt_ClarifyCorrelate (void *buf, int occurrence, char *param);

// lrt_display_buffer 函数将有关缓冲区的
// 信息存储到输出文件 section_name.out 中。
// 该文件包含每个
// 发送缓冲区和回复缓冲区的缓冲区描述。该信息
// 以以下格式显示：
int lrt_display_buffer (const char* bufferName, const char* bufferPtr, long actualLen,
long expectedLen );

// lrt_Fadd[32]_fld 函数将新字段
// 添加到 FML 缓冲区。必须将 FML 字段的名称或其字段 ID
// 指定为第二个参数。
// 必须在第三个参数中指定新
// FML fielFML 缓冲区的值。字段长度是可选的。
// 只需将 FML 字段的类型指定为
// CARRAY（二进制数据）。其他字段的长度
// 由字段类型和值来确定。
int lrt_Fadd[32]_fld ( FBFR[32] *fbfr, "name=fldname" | "id=idval", "value=fldvalue" [, "len=8"],
LRT_END_OF_PARAMS );

// lrt_Finitialize[32] 函数初始化
// 某个现有的 FML 缓冲区。该函数替代
// Tuxedo 函数 Finit 和 Fsizeof 的使用。
int lrt_Finitialize[32] ( FBFR[32] *fbfr );

// lrt_Fname[32] 函数提供从字段标识符
// 到其字段名的运行时转换。
char * lrt_Fname[32] ( FLDID[32] fieldid );
```

```
// lrt_Fstrerror[32] 函数检索
// 与 FML 错误代码对应的错误消息字符串。
char *lrt_Fstrerror[32] ( int err );

// lrt_getFerror[32] 函数检索上次失败的
// FML 操作的错误代码。在多任务
// 环境中，该函数用于为每个任务
// 提供一个独立的错误状态，而不
// 依赖于全局错误变量 (Ferrno)。
int lrt_getFerror[32] ( void );

// lrt_gettperrno 函数检索
// 上次失败事务的错误代码。在多任务
// 环境中，该函数用于为每个任务
// 提供一个独立的错误状态，而不
// 依赖于全局错误变量 (tperrno)。
int lrt_gettperrno ( void );

// lrt_gettpurcode 函数检索最后
// 一次调用 lrt_tpgetrply、lrt_tpcall、
// lrt_tprecv 或 lrt_tpdequeue 时设置的
// 全局应用程序返回代码变量 (tpurcode)。如果指定，
// tpurcode 还将包含使用 lrt_tpenqueue
// 发送的“user-return code”的值。
long lrt_gettpurcode ( void );

// lrt_InterateIDCals 在使用 Clarify CRM 应用程序时
// 由 Vugen 自动生成，并将返回的数据保存在
// 对数据库的 AS_EXESVC 调用
// 的回复缓冲区中。该数据用于关联。
void lrt_InterateIDCals (void *buf, int occurrence, char *param);

// lrt_memcpy 函数将指定的字节数
// 从源复制到目标。在复制到目标
// 之前，源保存在一个参数中
// 。该函数与
// C 函数 (memcpy) 功能相同。如果在用户计算机上
// 找不到 C 函数 (memcpy)，
// 则提供 lrt_memcpy 函数。由于我们使用 C 解释器，
// 因此不能假定在每台用户计算机上
// 都可以找到标准 C 库。
void lrt_memcpy ( void *dest, const void *source, unsigned long count );
```

```
// lrt_save_fld_val 函数将 FML 缓冲区的
// 当前值保存到 paramName 指定的参数中。
// 该函数用于关联脚本中的
// 查询。并不使用查询期间实际获取的
// 结果，而是用一个参数来代替
// 该常量值。之后，同一个脚本中的其他数据库语句
// 可以使用该参数。
int lrt_save_fld_val ( FBFR *fbfr, char *name, FLDOCC occ, char *paramName );

// lrt_save32_fld_val 函数将 FML32 缓冲区的
// 当前值保存到 paramName 指定的参数中。
// 该函数用于关联脚本中的
// 查询。并不使用查询期间实际获取的
// 结果，而是用一个参数来代替
// 该常量值。之后，同一个脚本中的其他数据库语句
// 可以使用该参数。
int lrt_save32_fld_val ( FBFR32 *fbfr, char *name, FLDOCC32 occ, char *paramName );

// lrt_save_parm 函数将字符数组的一部分
// 保存到 parm_name 指定的参数中。该函数
// 保存 parm_len 指定的字符数，
// 从 offset 指定的偏移量开始。
int lrt_save_parm ( char *buffer, int offset, int parm_len, char *parm_name );

// lrt_set_carray_error_list 函数设置
// 可用于 CARRAY 回复缓冲区的错误消息
// 列表。使用该函数之前，应在 CARRAY_ERROR_INFO 结构中
// 定义错误消息。
void lrt_set_carray_error_list (CARRAY_ERROR_INFO *newcarrayErrors);

// lrt_set_env_list 函数设置之后可以由 lrt_tuxputenv 设置
// 的变量列表。在调用 lrt_set_env_list
// 之后，可以设置 allowedEnv 列表
// 中的环境变量。
void lrt_set_env_list ( char **allowedEnv );

// lrt_strcpy 函数将指定的字符串从
// 源复制到目标。在复制到
// 目标之前，源保存在一个参数中。该
// 函数与 C 函数 (strcpy) 功能相同。
void lrt_strcpy ( char *destString, const char *sourceString );

// lrt_tpabort 函数中止当前的 Tuxedo
// 或 System/T 事务。这意味着事务的
```

```
// 异常结束。将撤消事务执行过程中
// 对资源所做的所有更改。lrt_tpabort
// 只能由事务的发起者调用。
int lrt_tpabort ( long flags );

//  lrt_tpacall 函数将请求消息发送给
// 指定的服务。这是可以以简要模式打印调试信息的
// 少数几个函数之一。
int lrt_tpacall ( char *svc, char *data, long len, long flags );

// lrt_tpalloc 函数分配新缓冲区，并
// 返回一个指向指定类型缓冲区的指针。由于
// 一些缓冲区类型在使用前
// 需要初始化，因此 lrt_tpalloc 会在分配之后，返回之前
// 初始化缓冲区。
char *lrt_tpalloc ( char *type, char *subtype, long size );

// lrt_tpbegin 函数开始一个 System/T 事务。
// System/T 中的事务是没有完全成功或根本没有生效的
// 工作的一个逻辑单元。
// 此类事务允许由多个进程执行工作，
// 而在不同的站点这些进程可能被视为
// 单一的工作单元。事务的发起者可以
// 可以使用 lrt_tpabort 或 lrt_tpcommit
// 结束事务。
int lrt_tpbegin(unsigned long timeout,   long flags);

// lrt_tpbroadcast 函数允许客户端或服务器
// 将未经请求的消息发送到系统中已注册的客户端。
int lrt_tpbroadcast ( char *lmid, char *username, char *cltname, char * data, long len, long flags );

// lrt_tpcall 函数发送服务请求并等待其回复。
int lrt_tpcall ( char *svc, char *idata, long ilen, char **odata, long *olen, long flags );

// lrt_tpcancel 函数取消调用描述符。
// 在 lrt_tpcancel 之后，调用描述符 cd 不再
// 有效，并将忽略对 cd 的任何回复。任何
// 取消与事务关联的调用描述符的尝试
// 都会产生错误。
int lrt_tpcancel ( int cd );

// lrt_tpchkauth 函数检查应用程序配置
// 是否要求身份验证。这通常由应用程序客户端
// 在调用 lrt_tpinitialize 之前使用，
```

```
// 以确定是否要求输入密码。
int lrt_tpchkauth ( );

// lrt_tpchkunsol 函数通过检查来确定
// Tuxedo 客户端是否接收到任何未经请求的消息。
int lrt_tpchkunsol ( void );

// lrt_tpcommit 函数提交当前的 System/T 事务。
int lrt_tpcommit ( long flags );

// lrt_tpconnect 函数建立一个半双工
// 会话式服务连接。
int lrt_tpconnect ( char *svc, char *data, long len, long flags );

// lrt_tpdequeue 函数获取用于处理的消息，
// 并将其从队列中删除。默认情况下，获取的是队列
// 顶端的消息。要请求一条特定的
// 消息，请在 ctl 中指定消息标识符。
int lrt_tpdequeue ( char *qspace, char *qname, TPQCTL *ctl, char **data, long *len, long flags );

// lrt_tpdiscon 函数断开会话式
// 服务连接。该函数只能由
// 会话的发起者调用。调用
// 该函数之后，您将再也无法在
// 该连接上进行发送或接收。
int lrt_tpdiscon ( int cd );

// lrt_tpenqueue 函数存储要在 qname 指定的队列上
// 处理的消息。
int lrt_tpenqueue ( char *qspace, char *qname, TPQCTL *ctl, char *data, long len, long flags );

// lrt_tpfree 函数释放先前由
// lrt_tpalloc 或 lrt_tprealloc 获取的缓冲区。
void lrt_tpfree ( char *ptr );

// lrt_tpgetlev 函数检查事务是否正在执行。
int lrt_tpgetlev ( ) ;

// lrt_tpgetrply 函数返回上次所发送请求的回复。
int lrt_tpgetrply ( int *cd, char **data, long *len, long flags );

// lrt_tpgprio 函数返回上次发送或接收
// 的请求的优先级。优先级范围
// 从 1 到 100，最高优先级为 100。
```

```
int lrt_tpgprio ( );

// lrt_tpinitialize 函数使客户端可以加入 System/T
// 应用程序。该函数将替换 TVG_tpinit 函数。
int lrt_tpinitialize (["usrname=value",] ["cltname=value", ] ["passwd=value", ] [grpname=value,]
[flags= value, ] [datalen=value, ] [data=value, ] LRT_END_OF_PARAMS );

// lrt_tprealloc 函数更改类型化缓冲区的大小。
char * lrt_tprealloc ( char *ptr, long size );

// lrt_tprecv 函数通过打开的连接接收
// 发自另一个程序的数据。它
// 与 lrt_tpsend 结合使用，
// 且只能由对该连接没有控制权的
// 程序发出。该函数可以以简要
// 模式打印调试信息。
int lrt_tprecv ( int cd, char **data, long *len, long flags, long *revent );

// lrt_tpresume 函数继续执行全局事务。它
// 与 lrt_tpsuspend 语句结合使用。
int lrt_tpresume ( TPTRANID *tranid, long flags );

// lrt_tpscmt 函数与 lrt_tpcommit 结合使用，
// 设置 lrt_tpcommit 应在何时返回。
int lrt_tpscmt ( long flags );

// lrt_tpsend 函数通过打开的连接
// 将消息发送到另一个程序。调用方
// 必须拥有连接控制权。它与
// lrt_tprecv 结合使用。该函数可以以
// 简要模式打印调试信息。
int lrt_tpsend ( int cd, char *data, long len, long flags, long *revent );

// lrt_tpsetunsol 函数
// 设置在接收到未经请求的消息时调用的回调过程。
// 回调过程必须在外部 DLL 中定义，
// 并具有以下原型：
void *lrt_tpsetunsol ( void *func );

// lrt_tpsprio 函数设置下一个
// 发送或转发请求的优先级。
int lrt_tpsprio ( int prio, long flags );

// lrt_tpstrerror 函数检索 System/T 错误
```

```
// 的错误消息字符串。
char *lrt_tpstrerror ( int err );

// lrt_tpsuspend 函数挂起全局事务。
// 它与 lrt_tpresume 语句结合使用。
int lrt_tpsuspend ( TPTRANID *tranid, long flags );

// lrt_tpterm 函数从 System/T 应用程序中
// 删除客户端。如果客户端处于事务
// 模式，则回滚该事务。
int lrt_tpterm ( );

// lrt_tptypes 函数确定有关类型化缓冲区的信息。
long lrt_tptypes ( char *ptr, char *type, char *subtype );

// lrt_tuxgetenv 函数在环境列表中搜索与环境名称
// 相对应的值
// 。在通常没有环境变量的平台上，
// 该函数对不同平台间环境值的
// 可移植性非常有用。
char *lrt_tuxgetenv ( char *name );

// lrt_tuxputenv 函数更改现有
// 环境变量的值，或者新建一个变量。
// 使用 lrt_set_env_list 确定
// 可以设置的变量。在通常没有环境变量的平台上，
// 该函数对不同平台间环境值的
// 可移植性非常有用。默认
// 情况下，lrt_tuxputenv 只允许更改 WSNADDR，
// 使用 lrt_set_env_list 时除外。
int lrt_tuxputenv (char *string);

// lrt_tuxreadenv 函数读取包含
// 环境变量的文件，并将其添加到环境中。
// 在通常没有环境变量的平台上，
// 该函数对不同平台间环境值的
// 可移植性非常有用。
int lrt_tuxreadenv ( char *file, char *label );

// lrt_tx_begin 函数开始一个全局事务。
int lrt_tx_begin ( );

// lrt_tx_close 函数关闭一组资源管理器。
// 该函数与 lrt_tx_open 结合使用。
```

```
int lrt_tx_close ( );

// lrt_tx_commit 函数提交一个全局事务。
int lrt_tx_commit ( );

// lrt_tx_info 函数返回全局事务信息。
int lrt_tx_info ( TXINFO *info );

// lrt_tx_open 函数打开一组资源管理器。
// 该函数与 lrt_tx_close 结合使用。
int lrt_tx_open ( );

// lrt_tx_rollback 函数回滚一个全局事务。
int lrt_tx_rollback ( );

// lrt_tx_set_commit_return 函数将 commit_return
// 特征设置为 when_return 中指定的值。
int lrt_tx_set_commit_return ( COMMIT_RETURN when_return );

// lrt_tx_set_transaction_control 函数将 transaction_control
// 特征设置为 control 中指定的值。
int lrt_tx_set_transaction_control ( TRANSACTION_CONTROL control );

// lrt_tx_set_transaction_timeout 函数将 transaction_timeout
// 特征设置为 timeout 中指定的值。
int lrt_tx_set_transaction_timeout ( TRANSACTION_TIMEOUT timeout );

// lr_advance_param 函数使脚本使用
// 参数的下一个可用值。如果要
// 运行多次循环，可以在
// 参数属性中指定自动前进到
// 每次循环的下一个值。在某次循环中使用该函数
// 可前进到下一个值。
int lr_advance_param ( const char * param);

// lr_abort 函数中止脚本
// 的执行。它停止 Actions 部分的执行，
// 执行 vuser_end 部分，然后
// 结束该执行。当因特定的错误情况
// 需要手动中止运行时，
// 该函数非常有用。使用该函数
// 结束运行时，状态为“停止”。
void lr_abort( );
```

```
// lr_continue_on_error 函数指定如何
// 处理错误。如果发生错误，可以选择
// 继续运行，或者中止运行执行。
void lr_continue_on_error ( int value );

// lr_convert_string_encoding 在下列编码
// 之间转换字符串编码：系统区域设置、Unicode 和 UTF-8。
// 该函数将结果字符串（包括其终止
// 结果字符串 NULL）保存在参数 paramName 中。
int lr_convert_string_encoding ( const char *sourceString, const char *fromEncoding, const char
*toEncoding, const char *paramName);

// lr_debug_message 函数在指定的消息级别
// 处于活动状态时发送一条调试消息。如果指定的
// 消息级别未处于活动状态，则不发出消息。
// 您可以从用户界面
// 或者使用 lr_set_debug_message，将处于活动状态的消息级别
// 设置为 MSG_CLASS_BRIEF_LOG 或 MSG_CLASS_EXTENDED_LOG。要确定当前级
别，
// 请使用 lr_get_debug_message。
int lr_debug_message (unsigned int message_level, const char * format, ... );

// lr_decrypt 函数对已编码的字符串进行解密。
// 该函数在录制过程中生成，以便
// 对密码进行编码。VuGen 录制实际的密码，
// 但在 lr_decrypt function 函数中
// 显示密码的编码版本。
char * lr_decrypt (const char *EncodedString);

// lr_disable_ip_spoofing 在脚本运行过程中禁用 IP 欺骗。
int lr_disable_ip_spoofing ( );

// lr_enable_ip_spoofing 在脚本运行过程中启用 IP 欺骗。
int lr_enable_ip_spoofing ( );

// lr_end_sub_transaction 函数标记
// 子事务的结束。要标记子事务的
// 开始，请使用 lr_start_sub_transaction
// 函数。应紧接子事务步骤前后
// 插入这些函数。
int lr_end_sub_transaction (const char * sub_transaction, int status );

// lr_end_transaction 函数标记事务的
// 结束，并录制执行事务
```

```
// 所用的时间量。要指明希望分析的事务，
// 请在事务之前放置 lr_start_transaction
// 函数，并在事务之后
// 放置 lr_end_transaction 函数。
int lr_end_transaction (const char * transaction_name, int status ) ;

// lr_end_transaction_instance 函数标记
// 事务实例的结束，并录制
// 执行事务所用的时间量。要指明
// 希望分析的事务实例，请在
// 事务之前放置 lr_start_transaction_instance 函数，
// 并在事务之后
// 放置 lr_end_transaction_instance 函数。
int lr_end_transaction_instance (long parent_handle, int status );

// lr_end_timer 停止在调用 lr_start_timer 时开始计时的
// 计时器。它以秒为单位返回已用时间。
// 分辨率取决于运行时环境。
// 最大分辨率为一微秒。
double lr_end_timer (merc_timer_handle_t timer);

// lr_eval_string 函数在评估任何嵌入的参数之后
// 返回输入字符串。如果字符串
// 实参 (argument) 只包含一个形参 (parameter)，该函数
// 返回形参的当前值。
char * lr_eval_string (const char * instring );

// lr_eval_string_ext 函数通过将参数替换为
// 字符串值来评估 in_str。
// 它创建包含扩展字符串的缓冲区，
// 并将 out_str 设置为指向该缓冲区。它还
// 将缓冲区的长度分配给 out_len。
int lr_eval_string_ext (const char * in_string, unsigned long const in_len, char ** const out_str,
unsigned long * const out_len, unsigned long const options, const char *file, long const line );

// lr_eval_string_ext_free 函数释放
// lr_eval_string_ext 分配的内存。
void lr_eval_string_ext_free (const char **param);

// lr_error_message 函数将错误消息发送到
// 输出窗口和 Vuser 日志文件。要发送
// 不是特定错误消息的特殊通知，
// 请使用 lr_output_message。
int lr_error_message (const char * format, exp1, exp2,...expn. );
```

```
// 使用 lr_exit 函数可在执行过程中
// 退出脚本运行。
void lr_exit (int continuation_option, int exit_status);

// lr_fail_trans_with_error 函数使用
// lr_end_transaction 语句中的 LR_AUTO，将所有
// 打开事务的默认退出状态设置为 LR_FAIL，
// 并发送错误消息。
int lr_fail_trans_with_error (const char * format, exp1, exp2,...expn.);

// lr_get_attrib_double 函数返回
// 命令行参数的值，其类型为
// 双精度浮点型。应将命令行参数的名称
// 放置在函数的实参 (argument) 字段，lr_get_attrib_double 将返回
// 该参数的值。
double lr_get_attrib_double (const char * parameter);

// lr_get_attrib_long 函数返回
// 命令行参数的值，其类型为长整型。
// 应将命令行参数名称放置在
// 函数的实参 (argument) 字段，lr_get_attrib_long
// 将返回该参数的值。
long lr_get_attrib_long (const char * parameter);

// lr_get_attrib_string 函数返回命令行
// 参数字符串。应将参数名称放置在函数的
// argument 字段，lr_get_attrib_string 将返回
// 与该参数关联的字符串值。
char * lr_get_attrib_string (const char * argument);

// lr_get_debug_message 函数返回当前的
// 日志运行时设置。该设置确定
// 发送到输出端的信息。日志设置是
// 使用运行时设置对话框或通过使用
// lr_set_debug_message 函数指定的。
unsigned int lr_get_debug_message ( );

// lr_get_host_name 函数返回
// 执行脚本的计算机的名称。
char * lr_get_host_name ( );

// lr_get_vuser_ip 函数返回 Vuser 的
// IP 地址。执行 IP 欺骗时，每个 Vuser 都
```

```
// 可以使用不同的地址。使用该函数可以确定
// 当前 Vuser 的 IP 地址。
char * lr_get_vuser_ip( );

// lr_get_master_host_name 函数返回运行
// 控制器或优化模块控制台的计算机的名称。
char * lr_get_master_host_name ( );

// lr_get_transaction_duration 函数返回
// 到该点为止指定事务的持续时间（秒）
// 。使用该函数可确定
// 事务结束前的总
// 事务时间。lr_get_transaction_duration 只针对打开事务返回
// 大于零的值。
double lr_get_transaction_duration (const char * transaction);

// lr_get_transaction_status 返回事务的
// 当前状态。不能在 lr_end_transaction 之后
// 调用 lr_get_transaction_status。由于 lr_get_transaction_status
// 只能返回打开事务的状态，因此
// 无法报告最终的事务状态。
int lr_get_transaction_status ( const char *transaction_name ) ;

// lr_get_trans_instance_status 返回事务实例的
// 当前状态。不能在 lr_end_transaction_instance 之后
// 调用 lr_get_trans_instance_status。它
// 无法报告最终的事务实例状态。
int lr_get_trans_instance_status ( long transaction_handle ) ;

// lr_get_trans_instance_duration 函数返回
// 到该点为止打开事务实例的持续
// 时间（秒）。使用该函数可确定事务结束前
// 的总事务时间。
double lr_get_trans_instance_duration (long trans_handle);

// lr_get_transaction_think_time 函数返回
// 到该点为止指定事务的
// 思考时间。它只针对打开事务返回
// 大于零的值。
double lr_get_transaction_think_time (const char * transaction);

// lr_get_trans_instance_think_time 函数返回
// 到该点为止指定事务的思考时间。
double lr_get_trans_instance_think_time (long trans_handle);
```

```
// lr_get_transaction_wasted_time 函数返回
// 到该点为止指定事务的浪费时间（秒）。
double lr_get_transaction_wasted_time (const char * transaction);

// lr_get_trans_instance_wasted_time 函数返回
// 到该点为止指定事务的浪费时间。
double lr_get_trans_instance_wasted_time (long trans_handle);

// lr_load_dll 函数加载 DLL (Windows)
// 或共享对象 (UNIX)，使您可以在
// 使用 C 解释器回放时调用外部函数。加载
// DLL 之后，您就可以调用 DLL 中定义的
// 任何函数，而不必声明。
int lr_load_dll (const char *library_name );

// lr_log_message 函数将消息发送到
// Vuser 或代理日志文件（取决于应用程序），
// 而不是发送到输出窗口。通过向日志文件发送错误消息或
// 其他信息性消息，可以将
// 该函数用于调试。
int lr_log_message (const char * format, exp1, exp2,...expn.);

// lr_message 函数将消息发送到日志文件和
// 输出窗口。在 VuGen 中运行时，输出文件为 output.txt。
int lr_message (const char * format, exp1, exp2,...expn.);

// lr_output_message 函数将带有脚本部分和行号的消息
// 发送到输出窗口和日志文件。
int lr_output_message (const char * format, exp1, exp2,...expn.);

// lr_next_row 函数获取指定文件中参数的
// 下一个可用行的值。如果
// 要运行多次循环，可以在参数
// 属性中指定前进到每次循环
// 的下一行。在特定循环中使用
// 该函数，可以前进到下一组值。
int lr_next_row ( const char * dat_file );

// lr_param_increment 函数检索
// source_param 的值，并以 1 为单位递增其值，
// 然后将递增后的值作为 null 终止字符串
// 存储在 destination_param 中。
int lr_param_increment (const char * destination_param, const char * source_param);
```

```
// lr_peek_events 函数用于在脚本执行期间
// 的特定时刻接收事件。应将该函数
// 插入到 Vuser 程序中希望 Vuser 暂停的
// 位置。如果脚本中不
// 包含 lr_peek_events 函数，您将无法
// 暂停 Vuser。
void lr_peek_events ( );

// lr_rendezvous_ex 函数在 Vuser 脚本中创建
// 一个集合点。执行该语句时，
// Vuser 程序会停止并等待 LoadRunner 赋予
// 权限以便继续。
int lr_rendezvous_ex (const char * rendezvous_name);

// lr_rendezvous 函数在 Vuser 脚本中
// 创建一个集合点。执行该语句时，
// Vuser 程序会停止并等待 LoadRunner 赋予
// 权限以便继续。
int lr_rendezvous (const char * rendezvous_name);

// lr_resume_transaction 函数继续执行
// 被 lr_stop_transaction 挂起的脚本中
// 事务数据的报告。调用 lr_stop_transaction 之后，
// “get”事务函数返回的统计信息
// 只反映该调用之前的数据，
// 直到调用此函数。
void lr_resume_transaction (const char * transaction_name);

// lr_resume_transaction_instance 函数可以继续报告由
// lr_stop_transaction_instance 挂起的脚本中的
// 事务数据。
// 调用 lr_stop_transaction_instance 后，
// “get”事务函数返回的统计信息
// 只反映该调用之前的数据，直到调用
// 此函数。
void lr_resume_transaction_instance ( long trans_handle );

// lr_save_datetime 函数将当前日期
// 和时间，或具有指定偏移的日期和时间
// 保存在参数中。如果达到 MAX_DATETIME_LEN 个字符，
// 结果字符串将截断。
void lr_save_datetime(const char *format, int offset, const char *name);
```

```
// lr_save_searched_string 函数在字符串
// 或字符数组缓冲区中搜索字符串 search_string，
// 并找到 search_string 第 n 次出现的位置，其中
// n 为 occurrence 加 1。要保存的子字符串
// 从 search_string 第 n 次出现位置的末尾偏移 offset 处开始，
// 长度为 string_len。
int lr_save_searched_string (const char *buffer, long buf_size, unsigned int occurrence, const char *search_string, int offset, unsigned int string_len, const char *parm_name );

// lr_save_string 函数将指定的以 null 终止的
// 字符串赋给参数。该函数可用于关联
// 查询。要确定参数值，请使用
// 函数 lr_eval_string。
int lr_save_string (const char *param_value, const char *param_name);

// lr_save_var function 函数将指定的变长
// 字符串赋给参数。该函数可用于
// 关联查询。要确定参数值，
// 请使用函数 lr_eval_string。
int lr_save_var (const char * param_value, unsigned long const value_len, unsigned long const options, const char * param_name);

// lr_set_debug_message 函数设置脚本
// 执行的调试消息级别 message_lvl。通过设置
// 消息级别，可以确定发送哪些
// 信息。启用设置的方法是将 LR_SWITCH_ON 作为 on_off 传递，
// 禁用设置的方法是传递 LR_SWITCH_OFF。
int lr_set_debug_message (unsigned int message_level, unsigned int on_off);

// lr_set_transaction 函数用于在一次调用中
// 创建事务、其持续时间及
// 状态。如果要在事务中捕获的
// 业务流程不由顺序步骤组成，
// 或者是否要创建事务视
// 只有在测试过程中才可知的条件而定，
// 请使用该函数。
int lr_set_transaction(const char *name, double duration, int status);

// lr_set_transaction_instance_status 函数使用
// 事务句柄为 trans_handle 设置打开事务的
// 状态。该句柄由
// lr_start_transaction_instance 返回。
int lr_set_transaction_instance_status (int status, long trans_handle);
```

```
// lr_set_transaction_status 函数设置
// 那些在其 lr_end_transaction 语句中包含 LR_AUTO 的
// 当前打开事务的状态。
int lr_set_transaction_status (int status);

// lr_set_transaction_status_by_name 函数使用
// 名称 trans_name 设置打开事务的
// 默认状态。该事务的 lr_end_transaction
// 语句必须使用自动状态分配，方法是
// 将 LR_AUTO 作为其 status 参数传递。
int lr_set_transaction_status_by_name (int status, const char *trans_name);

// lr_start_transaction 函数标记
// 事务的开始。要指明要分析的
// 事务，请使用函数 lr_start_transaction 和
// lr_end_transaction。应紧接事务前后
// 插入这些函数。
int lr_start_transaction ( const char * transaction_name );

// lr_start_transaction_instance 函数标记
// 事务实例的开始。事务
// 实例是名为 transaction_name 的事务的一次
// 发生。实例由它们的句柄
// 标识，句柄使它们区别于
// 同一事务的其他实例。
long lr_start_transaction_instance ( const char * transaction_name, long handle);

// lr_start_sub_transaction 函数标记
// 子事务的开始。要标记子事务
// 的结束，请使用 lr_end_sub_transaction。
// 请紧接子事务操作前后
// 插入这些函数。
int lr_start_sub_transaction (const char * sub_transaction, const char * parent_transaction);

// 调用 lr_stop_transaction 后，
// “get”事务函数返回的统计信息只反映
// 该调用之前的数据，直到调用 lr_resume_transaction
// 。指定的事务必须已
// 使用 lr_start_transaction 打开。
double lr_stop_transaction (const char * transaction_name);

// 调用 lr_stop_transaction_instance 后，
// “get”事务函数返回的统计信息
// 只反映该调用之前的数据，直到调用 lr_resume_transaction_instance
```

```
// 。指定的事务实例
// 必须已使用 lr_start_transaction_instance 打开。
double lr_stop_transaction_instance (long parent_handle);

// 通过 lr_think_time 可以在运行期间暂停测试
// 执行。这对于模拟思考时间非常有用，
// 思考时间是真实用户在操作之间停下来思考的时间。
void lr_think_time (double time);

// 通过函数 lr_user_data_point，可以记录
// 自己的数据以进行分析。每次要记录一个点
// 时，请使用该函数记录采样名称和
// 值。将自动记录采样的时间。
// 执行后，可以使用用户定义的数据
// 点图形来分析结果。
int lr_user_data_point (const char * sample_name, double value);

// 除附加参数 log_flag 之外，函数 lr_user_data_point_ex
// 与 lr_user_data_point 相同。
int lr_user_data_point_ex ( const char *sample_name, double value, int log_flag);

// 除参数 transaction_handle（通过该参数可以
// 将数据点与某个特定事务实例关联起来）之外，
// 函数 lr_user_data_point_instance
// 与 lr_user_data_point 相似。
long lr_user_data_point_instance (const char * sample_name, double value, long
transaction_handle);

// 除附加参数函数 log_flag 之外，函数
// lr_user_data_point_instance_ex 与
// lr_user_data_point_instance 相同。
long lr_user_data_point_instance_ex (const char*sample_name, double value, long
transaction_handle, int log_flag);

// lr_vuser_status_message 函数向控制器
// 或优化模块控制台的 Vuser 窗口的“状态”区域
// 发送字符串。它还将该字符串
// 发送到 Vuser 日志。从 VuGen 运行时，
// 消息被发送到 output.txt。
int lr_vuser_status_message (const char * format);

// 通过 lr_wasted_time 可以从所有打开事务中
// 减去在偶然或次要的操作上浪费的时间。
void lr_wasted_time (long time);
```

```
// lr_whoami 函数获取关于 Vuser 的信息。
void lr_whoami (int *vuser_id, char **sgroup, int *scid);

// radius_account 模拟 RAS 或 NAS 服务器，该服务器
// 向 RADIUS 服务器发送记帐信息。
// 请将其插入在 radius_authenticate 调用之后、
// WAP 会话之前或过程中。
int radius_account(const char *AccountName, const char *Action, <List of Arguments>, LAST);

// radius_authenticate 模拟 RAS 或 NAS 服务器，该服务器
// 在允许用户访问 WAP 网关之前，
// 向 RADIUS 服务器发送用户的身份验证信息。
// 请将其插入在调用 wap_connect 之前。
int radius_authenticate(<List of Arguments>, LAST);

// radius_get_last_error 返回最近的 RADIUS
// 函数调用的错误代码。
int radius_get_last_error ();

// radius_set_timeout 为函数 radius_authenticate
// 和 radius_account functions 设置连接超时值。它
// 会覆盖 RADIUS 运行时设置中的超时值
// 设置。参数 Timeout 是以秒为单位的超时值
// 的字符串表示。例如“5”。
int radius_set_timeout (const char *Timeout);

// wap_add_const_header 函数向常量头列表中
// 添加一个头。当建立 CO 类型的连接时，这些头
// 被传送至网关。
int wap_add_const_header ( const char *mpszHeader, const char *mpszValue );

// wap_bearer_down 可以断开与承载网络的连接。
// 载体是 WAP 网络的最底层服务。
// 有关详细信息，请参阅 wap_bearer_up。
int wap_bearer_down();

// wap_bearer_up 可以连接到承载网络。承
// 载体是 WAP 网络的最底层服务。
// 当测试只包括 SMS 消息时该函数非常有用，
// 使您不必使用 wap_connect 连接至
// WAP 网关。
int wap_bearer_up();
```

```
// wap_connect 函数可以连接至 WAP 网关，网关的
// IP 地址和端口在运行时设置中指定。
int wap_connect (    );

// wap_disconnect 函数可以断开与 WAP 网关的连接。
// 连接是通过调用 wap_connect 或 web_url 打开的。
int wap_disconnect (    );

// wap_format_si_msg 函数可以格式化 SI 类型的消息。
int    wap_format_si_msg (const char * mpszURL, const char * Slid,    const char * mpszCreated,
const char * mpszExpires, const char * mpszAction, const char * mpszUserMsg, char *
mpsfOutputMsg, unsigned int miOutputMsglen );

// wap_format_sl_msg 函数可以格式化 SL 类型的消息。
int    wap_format_sl_msg (const char * mpszURL, const char * mpszAction,    char *
mpsfOutputMsg, unsigned int miOutputMsglen );

// wap_mms_msg_decode 函数可以将字符串转换为
// MMS 消息对象。通常，该字符串
// 是接收到的 HTTP 消息的正文。
int wap_mms_msg_decode(void **MMSObject, const char *ParamName);

// wap_mms_msg_encode 函数可以将
// wap_mms_msg_create 创建的 MMS 消息对象 (MMSObject)
// 转换为字符串，该字符串随后可用于 Web 操作函数。
// 该方法不同于使用 wap_mms_msg_submit
// 或 as_wap.h 头文件中的其他宏，后者
// 直接接受 MMS 消息对象。
int wap_mms_msg_encode(void *MMSObject, const char *ParamName);

// wap_mms_msg_get_field 从 MMSObject 中
// 检索指定字段 FieldName 的值，
// 并将其存储在参数 ParamName 中。
int wap_mms_msg_get_field(void *MMSObject, const char *FieldName, const char
*ParamName);

// wap_mms_msg_get_multipart_entry 从 MMSObject
// 中的多部分项 EntryNumber 中
// 检索指定字段 FieldName 的值，
// 并将其存储在参数 ParamName 中。
int wap_mms_msg_get_multipart_entry(void *MMSObject, unsigned int EntryNumber, const
char *FieldName, const char *ParamName);

// wap_mms_msg_number_multipart_entries 可以检索
```

```
// MMSObject 中多部分项的数目。
int wap_mms_msg_number_multipart_entries(void *MMSObject, unsigned int
*NumberOfEntries);

// wap_pi_push_submit 函数可以向 PPG 提交一个“推”消息。
int   wap_pi_push_submit(const char * mpszName, const char * mpszURL, const char *
mpszPushID, const char * mpszClientAddress, const char * mpszDeliverBefore, const char *
mpszDeliverAfter, const char * mpszMsgSource, const char * mpszMsgType, const char *
mpszMsgContent);

// wap_pi_push_submit_ex 函数可以向 PPG 提交一个“推”消息。
int   wap_pi_push_submit_ex(const char * mpszName, const char * mpszURL, const char *
mpszPushID, const char * mpszClientAddress, const char * mpszDeliverBefore, const char *
mpszDeliverAfter, const char * mpszMsgSource, const char * mpszMsgType, const char *
mpszMsgContent [, CustomAttributes]);

// wap_pi_push_cancel 函数可以取消一个发送到 PPG 的消息。
int   wap_pi_push_cancel(const char * mpszName, const char * mpszURL, const char *
mpszPushID, const char * mpszClientAddress);

// wap_radius_connection 已过时。请使用
// radius_authenticate 和 radius_account。
int wap_radius_connection (const char * mpszAction, <List of Arguments>, LAST);

// wap_set_bearer 函数可以设置用于重放
// UDP 或 CIMD2 (SMS) 的载体类型。脚本使用该
// 载体类型，直到脚本结束或
// 再一次调用函数 wap_set_bearer 进行设置。
int wap_set_bearer ( const char *mpszBearer );

// wap_set_capability 函数可以设置网关连接的
// 客户端容量。建立网关连接时，
// 将对这些容量进行协商。
int wap_set_capability ( const char *mpszCap );

// wap_set_connection_mode 函数可以设置会话的连接
// 模式和安全级别。需要指明
// 连接的连接模式类型
// 和安全级别。该函数只对以后的
// 连接有效。如果连接已经建立，
// 则忽略该函数。
int wap_set_connection_mode( const char *   ConnectionMode,
const char * SecurityLevel);
```

```
// wap_set_connection_options 可以设置：IP 地址和端口
// （通过它们与网关通信）、连接
// 模式、安全级别和载体类型
int wap_set_connection_options( const char *argument_list, ..., LAST);

// wap_set_gateway 函数可以设置 IP 地址和
// 端口，通过它们可以与网关通信。
int   wap_set_gateway( const char * GateWayIP, const char * GateWayPort );

// wap_set_sms_user 可以设置 SMS 载体
// 的登录信息。可以在“运行时设置”的
// “网关”选项卡中设置载体的端口和
// 登录信息。该函数必须位于 wap_connect
// 或第一个 web_url 函数之前。
int wap_set_sms_user (const char *argument_list, ..., LAST);

// wap_wait_for_push 函数等待一个“推”消息
// 到达。如果超时前有消息到达，
// 则解析该消息，以确定其类型和
// 消息属性的值。如果解析成功，
// 客户端会发出一个“拉”消息，以检索
// 相关数据。通过配置运行时设置，可以禁用“拉”事件，
// 指明不检索消息
// 数据。在文件 default.cfg 中找到
// WAP 部分。将标志 PushRetrieveMsg 设置为
// 1 可以检索消息（默认），设置为 0 可以
// 禁止消息。
int wap_wait_for_push ( );

// wap_mms_msg_destroy 函数可以销毁
// 使用 wap_mms_msg_create 或 wap_mms_msg_decode 创建的消息。
// 请在已确认消息后将该函数放置在
// 脚本末尾。如果会话结束时
// 不销毁 MMS 消息，它将使用额外
// 资源并影响性能。
int wap_mms_msg_destroy (void * MMSObject );

// wap_mms_msg_create 函数可以创建消息。
// 请在建立与服务器的连接后
// 将该函数放置在脚本开头。
// 在会话结束时，请使用函数 wap_mms_msg_destroy
// 销毁该消息。
int wap_mms_msg_create (void **MMSObject );
```

```
// wap_mms_msg_add_field 函数可以向 MMS 消息添加消息字段，
// 添加的内容包括字段名和值。可以
// 在脚本中多次使用该函数。
int wap_mms_msg_add_field (void * MMSObject, const char * FieldName,
const char * FieldValue );

// wap_mms_msg_add_multipart_entry 函数可以向 MMS 消息
// 添加多部分项。项
// 可以是字符串（在最后一个参数中指定），
// 也可以是外部文件。可以在脚本中
// 多次使用该函数。
int wap_mms_msg_add_multipart_entry (void ** MMSObject,     const char *DataSource, const
char * ContentType,     const char *Headers, const char * Data );

// wap_mms_msg_retrieve 函数可以向 MMS 中心
// 发送请求，以获得 URL 处的消息。MMS
// 中心将消息写入 Message。它作为
// 文件 as_wap.h 中的一个宏实现。
void wap_mms_msg_retrieve (Name, URL, Message); /* MACRO (char*, char*, void**) */

// wap_mms_msg_retrieve_by_push 函数等待
// “推”消息到达。收到
// 该消息后，客户端将发出一个“拉”消息，
// 以检索相关数据。
void wap_mms_msg_retrieve_by_push (Message); /* MACRO (void**) */

// wap_mms_msg_submit 函数可以向 MMS 服务器
// 发送消息 MMSMessage。它作为
// 文件 as_wap.h 中的一个宏实现。
void wap_mms_msg_submit (Name, URL, Message); /* MACRO (char*, char*, void*) */

// wap_mms_msg_submit 函数使用指定的内容类型
// 向 MMS 服务器发送消息 MMSMessage。它作为
// 文件 as_wap.h 中的一个宏实现。
void wap_mms_msg_submit_enc (Name, URL, Message, EncodingType ); /* MACRO (char*,
char*, void*, char*) */

// wap_mms_msg_submit_and_receive_enc 函数使用指定的
// 内容类型向 MMS 服务器发送消息 Message，
// 并在返回前同步接收响应
// ReceivedMMS。它作为
// 文件 as_wap.h 中的一个宏实现。
void wap_mms_msg_submit_and_receive_enc (Name, URL, Message, ReceivedMMS,
EncodingType ); /* MACRO (char*, char*, void*, void**, char*) */
```

```
// wap_mms_msg_submit_and_receive 函数向 MMS 服务器
// 发送消息 Message，并在返回前
// 同步接收响应 ReceivedMMS。
// 它作为文件 as_wap.h 中的一个宏实现。
void wap_mms_msg_submit_and_receive(Name, URL, Message, ReceivedMMS); /* MACRO
(char*, char*, void*, void**) */

// wap_push_msg_get_field 从最近收到的
// “推”消息中检索字段 FieldName 的值。
// 检索到的值存储在参数 ParamName 中。
int wap_push_msg_get_field(const char *FieldName, const char *ParamName);

// wap_send_sms 函数向指定地址发送一个 SMS
// 类型的消息。
int wap_send_sms (const char * List of Arguments, LAST);

// web_add_auto_header 函数是一个服务函数，
// 它向所有后续 HTTP 请求添加头。
int web_add_auto_header (const char *Header, const char *Content );

// web_add_cookie 函数可以添加新的 Cookie。如果
// 名称和路径与现有 Cookie 匹配，则现有
// Cookie 被新 Cookie 覆盖。如果已过“过期”
// 日期，则删除该 Cookie。
int web_add_cookie (const char *Cookie);

// web_add_filter 指定下载下一个操作函数中的内容时
// 要使用的筛选器。
// 筛选器将包含或排除含有匹配条件的 URL，
// 视传递到该函数的 Action 属性
// 而定。默认
// 操作为“Action=Exclude”。
int web_add_filter ( [Action,]< List of Attributes >, LAST ) ;

// web_add_auto_filter 指定下载随后发生的操作函数的内容
// 时要使用的筛选器。
// 调用 web_remove_auto_filter 后，将禁用
// 该筛选器。筛选器将包含或排除含有匹配条件的 URL，
// 视传递到该函数的 Action 属性
// 而定。默认
// 操作为“Action=Exclude”。
int web_add_auto_filter ( [Action,]< List of Attributes >, LAST);
```

```
// web_remove_auto_filter 函数可以禁用
// 上次调用 web_add_auto_filter 时的筛选器设置。
int web_remove_auto_filter ( char *Id, LAST );

// web_add_header 函数是一个服务函数，
// 它向下一个 HTTP 请求添加用户定义的头。
int web_add_header (const char *Header, const char *Content );

// web_cache_cleanup 函数是一个服务函数，
// 它清除缓存模拟程序中的内容。
// 如果在“浏览器模拟”选项卡上启用运行时
// 设置选项“每次循环时模拟一个新用户”，
// 则该函数会在每次循环开始时
// 自动调用。
int web_cache_cleanup();

// web_cleanup_auto_headers 函数是一个
// 服务函数，它会禁止向后续 HTTP 请求
// 添加用户定义的头。
int web_cleanup_auto_headers ( );

// web_cleanup_cookies 函数删除脚本使用的
// 所有当前存储的 Cookie。
int web_cleanup_cookies ( );

// web_concurrent_end 函数可以标记
// 并发组的结束，并开始并发执行所有注册
// 为并发的函数（位于函数
// web_concurrent_start 和
// web_concurrent_end 之间的函数）。
// 单击“并发函数”，以查看并发组中可能
// 包含的函数的列表。
int web_concurrent_end ( reserved );

// web_concurrent_start 函数可以
// 标记并发组的开始。组中的所有函数
// 均并发执行。组的结束由函数
// web_concurrent_end 标记。在并发组
// 中可以包含操作函数和几个服务函数。
// 单击“并发函数”，以查看并发组中可能
// 包含的函数的列表。
//
int web_concurrent_start ( [char * ConcurrentGroupName,] NULL );
```

```
// web_convert_param 函数将 HTML 文本
// 转换为纯文本或 URL，或将纯文本转换为 URL。
int web_convert_param (const char *ParamName, [char *SourceString] char *SourceEncoding,
char *TargetEncoding, LAST);

// web_create_html_param 函数是一个服务
// 函数，用于在 Web 脚本中关联 HTML 语句。
// 函数 web_create_html_param
// 检索重放期间生成的动态信息，然后将
// 第一次出现的动态信息保存在某个
// 参数中。
int web_create_html_param (const char *ParamName, const char *LeftBoundary,  const char
*RightBoundary );

// web_create_html_param_ex 函数是一个服务
// 函数，用于在 Web 脚本中关联 HTML 语句。
// 函数 web_create_html_param_ex 检索重放
// 期间生成的动态信息，然后将该动态信息保存在
// 某个参数中。
int web_create_html_param_ex (const char *ParamName, const char *LeftBoundary,  const char
*RightBoundary, const char *Instance );

// web_custom_request 函数是一个操作函数，
// 通过它可以使用任意方法创建自定义 HTTP 请求
// 或创建正文。默认情况下，VuGen 只为无法
// 用其他 Web 函数解释的请求生成该函数。
//
int web_custom_request (const char *RequestName, <List of Attributes>, [EXTRARES, <List of
Resource Attributes>,] LAST );

// web_disable_keep_alive 函数是一个服务
// 函数，它禁用 Keep－Alive HTTP 连接。
// 默认情况下，KeepAlive 设置处于启用状态。
int web_disable_keep_alive ( );

// web_dump_cache 保存浏览器缓存。它与
// web_load_cache 一起使用，以实现 Vuser
// 持续缓存。脚本始终使用相同的初始缓存运行。
int web_dump_cache ( const char *Name, const char * fileName, [ const char * Replace], LAST );

// web_enable_keep_alive 函数是一个服务
// 函数，它启用 Keep－Alive HTTP 连接。
int web_enable_keep_alive ( );
```

```
// web_find 函数在 HTML 页中搜索指定的文本字符串。
int web_find (const char *StepName, <Attributes and Specifications list>, char *searchstring,
LAST );

// web_get_int_property 函数返回
// 关于上一个 HTTP 请求的指定信息。
int web_get_int_property (const int HttpInfoType);

// web_global_verification 函数注册
// 一个请求，以在所有后续操作函数返回的
// 网页中搜索指定的文本字符串。这与函数
// web_reg_find 不同，后者只为下一个
// 操作函数注册请求。可以搜索页面的正文、
// 头、HTML 代码或全部内容。
//
int web_global_verification (<List of Attributes>, LAST );

// 操作函数 web_image 模拟
// 鼠标单击由属性定义的图像。
// 该函数只能在上一个操作
// 的上下文中执行。
int web_image (const char *StepName, <List of Attributes>, [EXTRARES, <List of Resource
Attributes>,] LAST );

// web_image_check 函数可以验证 HTML 页中
// 包含指定图像。
int web_image_check(const char *CheckName, <List of Attributes>, <"Alt=alt"|| "Src=src">,
LAST );

// web_link 函数是一个操作函数，
// 它模拟鼠标单击由属性定义的
// 链接。web_link 只能在上一个
// 操作的上下文中执行。
int web_link (const char *StepName, <List of Attributes>, [EXTRARES, <List of Resource
Attributes>,] LAST );

// web_remove_cookie 函数从一个对 Vuser
// 可用的 Cookie 列表中删除一个 Cookie。该函数
// 指定要删除的 Cookie 的名称。
int web_remove_cookie (const char *Cookie);

// web_load_cache 可以从文件中还原浏览器
// 缓存。它与 web_dump_cache 一起
// 使用，以实现 Vuser 持续缓存。脚本
```

```
// 始终使用相同的初始缓存运行。
int web_load_cache ( const char *Name, const char * fileName, LAST );

// web_reg_add_cookie 函数注册一个搜索，
// 在下一个操作函数（如 web_url）检索到的网页上
// 搜索一个文本字符串。如果
// 找到了字符串，将添加 Cookie。
int web_reg_add_cookie(const char * cookie, const char * searchstring, LAST );

// web_reg_find 函数注册一个请求，以
// 在下一个操作函数（如 web_url）检索到
// 的网页上搜索一个文本字符串。
int web_reg_find (const char *attribute_list, LAST);

// web_reg_save_param 是一个注册类型
// 的函数。它注册一个请求，以在检索到的
// 网页中查找并保存一个文本字符串。只有
// 在执行了下一个操作函数（如 web_url）后
// 才会执行该操作。
int web_reg_save_param (const char *ParamName, <List of Attributes>, LAST);

// web_remove_auto_header 函数是一个服务
// 函数，它停止向后续 HTTP 请求添加特定的用户
// 定义头。该函数将取消由 web_add_auto_header
// 启动的指定头的自动头生成。
//
int web_remove_auto_header (const char *Header, char *Implicit, LAST);

// web_revert_auto_header 函数是一个服务
// 函数，它停止向后续 HTTP 请求添加特定的用户
// 定义头。该函数将取消由 web_add_auto_header
// 启动的指定头的自动头生成。它不断生成隐性头，
// 如同不曾调用过函数 web_add_auto_header
// 或 web_remove_auto_header 一样。
int web_revert_auto_header (char *Header);

// web_report_data_point 函数在脚本中
// 定义要包括在测试结果中的数据点。最常见的
// 数据点是步骤超时，它表明上一个步骤是否超时。
//
int web_report_data_point ( const char * EventType, const char * EventName , const char *
DataPointName , LAST );

// web_save_header 函数将所有随后
```

```
// 发生的操作函数的主 URL 的请求和
// 响应头保存在参数 param 中。每个头由
// “\r\n”（或仅“\n”）分隔。每个新的
// 请求头将替换参数的当前值。
int web_save_header (const char *type, const char *param );

// web_save_param_length 创建名为
// “<Param>_Length”的新参数（如果尚无该参数），
// 并将 Param 的长度保存在参数“<Param>_Length”
// 中。该长度采用十六进制格式。
int web_save_param_length( const char *Param, LAST );

// web_save_timestamp_param 保存当前
// 时间戳。在某些应用中，VuGen 用一个参数
// 替换脚本中的所有非空时间戳。为了保存
// 该参数的值，VuGen 自动生成对
// web_save_timestamp_param 的调用。
// 保存的值是自 1970 年 1 月 1 日午夜
// 开始的毫秒数。
int web_save_timestamp_param( const char * tmstampParam, LAST );

// web_set_certificate 指定一个证书在
// 证书列表中的编号。然后，只要某个安全的
// Web 服务器需要客户端提供证书，就使用
// 指定的证书。
int web_set_certificate (const char *CertificateNumber);

// web_set_certificate_ex 设置证书
// 和关键文件属性，如位置、类型和密码。
// 该信息用于需要证书的 HTTPS 请求。
// 所有参数都是以 null 终止的字符串。
// 关键字不区分大小写；但属于关键字
// 的值是区分大小写的。关键字
//
// 值的开头和末尾不允许出现
// 空格。注意，只有使用 Internet Explorer
// 时才录制该函数。
int web_set_certificate_ex (const char *option_list, LAST);

// web_set_connections_limit 函数是
// 一个服务函数，它设置脚本执行期间可以行
// 同时运的最大请求数。
// 在加载页面资源或加载框架集页面
// 中的框架等情形下，会发出
```

```
// 多个请求。任何 Vuser 的默认限制为
// 可以同时发出四个请求。
int web_set_connections_limit (const char *Limit );

// web_set_max_html_param_len 函数是
// 一个服务函数，用于关联 HTML 语句。
// 仅当启用“在录制过程中关联”时
// 才录制该函数（请参阅 VuGen 的录制选项）。
int web_set_max_html_param_len (const char *length );

// web_set_max_retries 函数设置操作的
// 最大重试次数。当发生错误时，对于 HTTP
// 响应消息 500-599 和网络 API 错误
// （HttpSendRequest 等），会尝试进行
// 重试。对于超时或函数参数错误，不尝试
// 进行重试。
int web_set_max_retries (const char *MaxRetries );

// web_set_option 函数设置 Web 选项。
// 它是一个服务函数，影响其后的所有函数，
// 直到指定了新值。vuser_init 部分
// 结束运行时，将保存当前选项值。
// 在每次循环开始之前，这些值将还原为
// 保存的值。
int web_set_option (const char *OptionID, const char * OptionValue, LAST );

// web_set_proxy 函数是一个服务函数，
// 它指定将所有后续 HTTP 请求定向
// 到指定的代理服务器。要直接向服务器
// 提交 HTTP 请求（即不使用代理
// 服务器），请使用函数 web_set_proxy
// 传递一个空字符串 ("") 作为参数。
int web_set_proxy (const char *proxy_host:port );

// web_set_proxy_bypass 函数是一个服务函数，
// 它指定要直接访问的 URL 的列表，该访问会避开
// 代理服务器。可以在避开的 URL 列表中包含 <local>，
// 以使所有本地主机（如 Intranet 服务器）
// 都避开代理服务器。
int web_set_proxy_bypass (const char *bypass1..n );

// web_set_proxy_bypass_local 函数是
// 一个服务函数，它指定代理服务器是否应
// 避开本地 (Intranet) 地址。该函数会覆盖
```

```
// 运行时设置代理选项“对本地 (intranet)
// 地址不使用代理服务器”。如果代理避开
// 字符串包含 <local>，则该函数
// 和 UI 复选框都将没有以下效果：将
// 始终避开本地地址。
int web_set_proxy_bypass_local ( const char *no_local );

// web_set_secure_proxy 函数是一个服务函数，
// 它指定将所有后续 HTTPS 请求
// 定向到指定的安全代理服务器。
int web_set_secure_proxy (const char *secure_proxy_host_port );

// web_set_sockets_option 函数配置
// 客户端上的套接字选项。对于启用或禁用
// 某项功能（如 TRACE_SSL_IO）的选项，
// 请指定“1”以启用，指定“0”以禁用。
// 以下列表显示了支持的选项：
int web_set_sockets_option( const char *option, const char * value );

// web_set_timeout 函数是一个服务函数，它
// 指定 CONNECT、RECEIVE 或 STEP 操作完成
// 之前等待的最长时间。
int web_set_timeout (const char *Action, const char *TimeOut );

// web_set_user 函数是一个服务函数，它
// 指定 Web 服务器或代理服务器的登录字符串
// 和密码。如果多个代理服务器需要身份验证，
// 可以多次调用该函数。web_set_user
// 会覆盖运行时代理身份验证的用户名和密码
// 设置。
int web_set_user (const char *username, const char *password, const char *host:port );

// web_sjis_to_euc_param 函数将一个 SJIS 编码的
// 以 null 终止的字符串转换为 EUC 编码的字符串，并将它
// 赋给参数。可以使用函数 lr_eval_string
// 确定参数的值。
int web_sjis_to_euc_param (LPCSTR param_name, LPCSTR param_value_in_SJIS);

// web_submit_data 函数是一个操作函数，
// 它执行“无条件的”或“无上下文的”表单
// 提交。通过它可以生成 GET 和 POST
// 请求，如同由 HTML 表单生成的请求。执行该请求
// 不需要有表单上下文。
int web_submit_data ( const char *StepName, <List of Attributes>, ITEMDATA, <List of data>,
```

[EXTRARES, <List of Resource Attributes>,] LAST);

// web_submit_form 函数是一个操作函数，它
// 可以提交表单。函数 web_submit_form 只能在上一
// 操作的上下文中执行。
int web_submit_form (const char *StepName, <List of Attributes>, <List of Hidden Fields>, ITEMDATA, <List of Data Fields>, [EXTRARES, <List of Resource Attributes>,] LAST);

// 通过 web_switch_net_layer 函数，可以在
// 要重放的网络层间切换。通过它可以在同一
// 脚本中混用直接的 HTTP 和 WSP 调用。目前，
// 该函数只影响 WAP Vuser。
int web_switch_net_layer (const char *NetName);

// web_url 函数是一个操作函数，它可以加载
// 指定的网页（GET 请求）。函数 web_url
// 可以加载 URL 属性指定的 URL。函数 web_url
// 不需要上下文。
int web_url (const char *Name, const char * url, <List of Attributes>, [EXTRARES, <List of Resource Attributes>,] LAST);

// web_browser 操作函数直接在打开的
// 浏览器上执行操作。
int web_browser (const char *stepName, [const char *snapShot,] [DESCRIPTION, const char *browser,] ACTION, const char *userAction, LAST);

// web_button 操作函数模拟用户单击按钮。
int web_button (const char *stepName, [const char *snapShot,] DESCRIPTION, [const char *type,] [const char *tag,] [const char *arg1, … const char *argn,] LAST);

// web_check_box 操作函数选择或清除
// 复选框。如果 setOnOff 为“Set=ON”，则选中复选框。
// 如果 setOnOff 为“Set=OFF”，则清除复选框。
int web_check_box (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *arg1, … const char *argn, ACTION, const char *setOnOff, LAST);

// web_edit_field 函数在文本或密码输入元素
// 中输入 setValue。
int web_edit_field (const char *stepName, [const char *snapShot,] DESCRIPTION, [const char *type,] [const char *arg1, … const char *argn,] ACTION, const char *setValue, LAST);

// 永远不录制 web_eval_java_script。
// 可以在脚本中添加该函数，以处理没有
// 标准解决方案的情况。该函数有三种

```
// 用法，语法各不相同。请勿混淆不同语法
// 的参数。请勿在语法 1 或语法 2 中使用
// PrintProperties，勿在语法 2 或语法 3
// 中使用 script，以及勿在语法 1 或语法 3 中使用
// Expression 和 SaveExpressionResult。
int web_eval_java_script ( const char *stepName, const char *script, [DESCRIPTION, const char *arg1, …, const char *argn,] LAST );

// web_file 在形如 <INPUT TYPE=FILE NAME=”Name”>
// 的元素中输入文件名 SetPath。
int web_file ( const char *StepName, DESCRIPTION, [<List of attributes>], ACTION, const char * SetPath, LAST );

// web_image_link 操作函数模拟用户
// 单击图像（该图像是超文本链接）。
int web_image_link (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *arg1, … const char *argn, [ACTION, const char *clickCoordinates,] LAST );

// web_image_submit 操作函数模拟用户
// 单击类型为“image”的输入元素。
int web_image_submit (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *arg1, … const char *argn, ACTION, const char *clickCoordinates, LAST );

// web_list 函数模拟用户在列表框
// 中选择一项。选择项可以通过
// 选择项的文本或其位置识别。
// 第一个选项的位置为 1。
int web_list (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *type, [const char *arg1, … const char *argn,] ACTION, const char *selectAction, [const char *ordinalSelectAction,] LAST );

// web_map_area 函数可以激活客户端映射的一个
// 区域。无论该区域如何激活，都将录制该函数。通常，
// 用户单击一个区域会激活该区域。
int web_map_area (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *mapName, const char *arg1, … const char *argn, LAST );

// web_radio_group 操作函数模拟用户
// 在单选按钮组中选择一个按钮。
int web_radio_group (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *name, const char *arg1, … const char *argn, ACTION, const char *selection, LAST );

// web_reg_dialog 服务函数可以注册
// Java 脚本使用的信息。
```

int web_reg_dialog (DESCRIPTION, const char *type, [const char *message,][const char *browser,] ACTION, const char *userAction, LAST);

// web_static_image 模拟用户单击图像。
// 如果 IMG 元素被具有 HREF 属性的 <A> 元素
// 围起，则它是一个链接，该函数无效。在这种
// 情况下，请使用 web_image_link。
int web_static_image (const char *StepName, DESCRIPTION, <List of attributes>, LAST);

// web_text_area 操作函数在文本区域中输入 setText。
int web_text_area (const char *stepName, [const char *snapShot,] DESCRIPTION, const char *arg1, … const char *argn, ACTION, const char *setText, LAST);

// web_text_link 操作函数模拟
// 用户单击超文本链接。
int web_text_link (const char *stepName, [const char *snapShot,] DESCRIPTION, [const char *text,][const char *arg1, … const char *argn,] LAST);

// lrs_accept_connection 函数从
// old_socket 上的挂起连接队列中
// 取出第一个连接，使用相同属性创建
// 新的套接字。原来的套接字对于其他
// 连接仍然可用。
int lrs_accept_connection (char *old_socket, char *new_socket);

// lrs_ascii_to_ebcdic 函数可以将二进制
// 缓冲区数据从 ASCII 格式转换为 EBCDIC 格式。
// 转换后的缓冲区以 EBCDIC 格式存储在内部缓冲区
// （用户缓冲区）中。要检索缓冲区的内容，请使用
// lrs_get_user_buffer。要确定缓冲区大小，
// 请使用 lrs_get_user_buffer_size。
char *lrs_ascii_to_ebcdic (char *s_desc, char *buff, long length);

// lrs_cleanup 函数终止使用 Windows
// Sockets DLL（类似于 WSACleanup）
// 和系统资源，如释放文件描述符。请检查
// 返回代码，以验证清理成功。
int lrs_cleanup ( );

// lrs_close_socket 函数释放指定的
// 套接字描述符，以关闭套接字。再次引用
// 该套接字将会导致错误 WSAENOTSOCK。
// 如果这是对基础套接字的最后一次引用，
// 将丢弃相关的命名信息和队列数据。

```
// 注意，对于 TCP 套接字，函数
// lrs_close_socket 会将所有尚未发送
// 的数据发送出去。
int lrs_close_socket ( char *s_desc );

// lrs_create_socket 函数初始化一个套接字。
// 它执行 socket 命令，打开新的套接字。
// 如果提供了 LocalHost 参数，它会执行
// bind 命令，以命名该套接字。如果提供了 peer 参数，
// 它会执行 connect 命令，以建立与对等端的
// 连接。如果提供了 backlog 参数，它
// 会执行 listen 命令侦听该套接字。
int lrs_create_socket ( char *s_desc, char *type, [ char* LocalHost,] [char* peer,] [char *backlog,]
LrsLastArg );

// lrs_decimal_to_hex_string 函数可以将整数
// 转换为十六进制字符串。例如，如果字符串为
// 4，则该函数将它转换为 \x04。
char*   lrs_decimal_to_hex_string( char* s_desc, char* buf, long length);

// lrs_disable_socket 函数禁止套接字的一项
// 操作。可以禁止所有的发送和/或接收操作。
int lrs_disable_socket ( char *s_desc, int operation );

// lrs_ebcdic_to_ascii 函数将二进制缓冲区
// 数据从 EBCDIC 格式转换为 ASCII 格式。转换后
// 的缓冲区以 ASCII 格式存储在内部缓冲区
// （用户缓冲区）中。要检索缓冲区的内容，
// 请使用 lrs_get_user_buffer。要确定缓冲区
// 的大小，请使用 lrs_get_user_buffer_size。
char *lrs_ebcdic_to_ascii ( char *s_desc, char *buff, long length );

// lrs_exclude_socket 函数排除指定
// 套接字的所有操作。脚本中从该函数开始
// 向后的所有使用该指定套接字的函数都
// 将被忽略。建议将该函数放在脚本的
// vuser_init 部分。排除对脚本所有
// 部分中的套接字函数都有效。
//
int lrs_exclude_socket ( char *s_desc );

// lrs_free_buffer 函数释放为指定缓冲区分配的
// 内存。内存是在调用函数 lrs_get_buffer_by_name
// 或 lrs_get_last_received_buffer 时
```

```
// 分配的。
int lrs_free_buffer ( char *buffer );

// 对于指定的缓冲区描述符，函数
// lrs_get_buffer_by_name 获取
// 二进制缓冲区以及数据二进制表示的长度。
// 注意，数据缓冲区不是以 NULL 终止的。
int lrs_get_buffer_by_name ( char *buf_desc, char **data, int *size );

// lrs_get_last_received_buffer 函数获取
// 套接字上最后收到的缓冲区及其大小。注意，该
// 函数返回数据二进制表示的长度。数据缓冲区
// 不是以 NULL 终止的。
int lrs_get_last_received_buffer ( char *s_desc, char **data, int *size );

// lrs_get_last_received_buffer_size 函数
// 获取套接字 s_desc 上最后收到的缓冲区的
// 大小。注意，该函数返回数据二进制表示的
// 长度。
int lrs_get_last_received_buffer_size ( char *s_desc );

// lrs_get_received_buffer 函数检索
// 最后一次调用 lrs_receive、lrs_receive_ex
// 或 lrs_length_receive 收到的缓冲区的全部
// 或其一部分。请指定要检索的数据的
// 偏移量和长度。
char *lrs_get_received_buffer ( char *s_desc, int offset, int length, char *encoding );

// lrs_get_static_buffer 函数检索
// 来自数据文件的静态缓冲区或其一部分。
// 请指定缓冲区及要检索的数据的偏移量
// 和长度。该缓冲区在发生任何参数替换后
// 返回。
char *lrs_get_static_buffer ( char *s_desc, char *buffer, int offset, int length, char *encoding );

// lrs_get_socket_attrib 函数检索
// 指定的套接字属性。通过它可以获取有关
// 套接字的信息。只能从绑定或连接到某个
// 套接字的套接字检索属性。要将一个套接字
// 绑定到现有套接字，请使用 lrs_create_socket(…LocalPort=…)。
// 要连接到一个套接字，请使用 lrs_create_socket(…RemoteHost=…)。
char *lrs_get_socket_attrib ( char *s_desc , int attribute );

// lrs_get_socket_handler 函数检索指定
```

```
// 套接字的套接字句柄。检索到套接字句柄后，
// 可以在后续函数中使用。
int lrs_get_socket_handler ( char *s_desc );

// lrs_get_user_buffer 函数检索指定套接字的
// 用户数据缓冲区的内容。
char *lrs_get_user_buffer ( char *s_desc );

// lrs_get_user_buffer_size 函数检索指定套接字的
// 用户数据缓冲区的大小。
long lrs_get_user_buffer_size ( char *s_desc );

// lrs_hex_string_to_int 将一个
// 十六进制字符串转换为整数。请指定
// 指向包含要转换字符串的缓冲区的指针
// 及要转换字符串的长度。该函数将转换
// 字符串并赋值给整型引用。
int lrs_hex_string_to_int ( char* buff, long length, int* mpiOutput );

// lrs_length_receive 函数将指定长度的
// 数据从 sock_descriptor 读入缓冲区。
// 长度位于接收到的缓冲区自身内部。用户
// 必须知道该长度值位于缓冲区内何处
// （除非选择了 RecieveOption_None
// 选项），其位置通过参数 location_option
// 和 locators 指定。lrs_length_receive
// 首先获得长度值（下面称为 length），
// 再将 length 个字符从 socket_descriptor
// 读入缓冲区。
int lrs_length_receive(char *socket_descriptor, char *buffer, int location_option,  [char* locators],
[char* additional_params], LrsLastArg );

// lrs_length_send 函数将指定
// 长度的数据从 sock_descriptor 于
// 写入 buffer。它用发送参数化数据，
// 每次调用该函数时数据的长度可以
// 不同。使用 lrs_length_send 可以
// 不必每次计算长度，因为 Vugen
// 自动将正确的数据长度写入由参数
// locator 和 location_option
// 指定的字段。
int lrs_length_send(char *socket_descriptor, char *buffer, int location_option,  [char* locators],
[char* additional_params], LrsLastArg );
```

```
// lrs_receive 函数从数据报或流式套接字
// 读取输入数据。如果套接字上没有输入
// 数据，lrs_receive 将等待数据到达，
// 除非套接字是非分块套接字。
int lrs_receive ( char *s_desc, char *bufindex, [char *flags], LrsLastArg );

// lrs_receive_ex 函数从数据报或
// 流式套接字的输入数据中读取指定数目
// 的字节。除了可以指定要接收字节
// 数的功能外，它与函数 lrs_receive
// 等价。
int lrs_receive_ex ( char *s_desc, char *bufindex, [char *flags,] [char *size,] [char
*terminator,] [char *mismatch,] [char *RecordingSize,] LrsLastArg );

// lrs_save_param 函数将数据从缓冲区保存
// 到参数中。该函数用于关联或链接脚本中的
// 语句。
int lrs_save_param ( char *s_desc,char *buf_desc, char *param_name, int offset, int param_len);

// lrs_save_param_ex 函数将缓冲区或
// 缓冲区的一部分保存到参数中。参数 type
// 指定要保存的缓冲区数据的类型：用户缓冲区、
// 录制缓冲区或最后接收的缓冲区。该函数
// 会覆盖用户数据区。
int lrs_save_param_ex ( char *s_desc, char *type, char *buff, int offset, int length, char *encoding,
char *param );

// lrs_save_searched_string 函数将
// 缓冲区的一部分保存到参数中。该函数用于
// 关联或链接脚本中的语句。该函数扩展了
// lrs_save_param 的功能。
int lrs_save_searched_string (char* s_desc, char* buf_desc, char* param_name, char*
left_boundary, char* right_boundary, int ordinal, int offset, int param_len );

// lrs_send 函数将输出数据写到已
// 连接的数据报或流式套接字。如果不能
// 成功发送缓冲区中的所有数据，将重复
// 尝试，直到发送超时。如果无法为数据
// 找到可写的套接字，该函数将不断查找
// 套接字，直到发送超时。默认情况下，
// 发送超时值为 10 秒。可以使用
// lrs_set_send_timeout 修改超时值。
// 注意，lrs_send 成功完成并不表示
// 数据已成功传递。
```

```
int lrs_send ( char *s_desc, char *buf_desc, [char *target], [char *flags,] LrsLastArg );

// lrs_set_accept_timeout 函数设置
// 服务器在检测到可读套接字 (select)
// 并完成关联 (accept) 之前等待的时间。
// 如果套接字当前正在侦听，并且
// 收到了传入的连接请求，它将被标记为
// 可读。一旦套接字被标记为可读，
// 就可以保证完成 accept，不会阻塞。
// 可以为超时值指定秒数和毫秒
// 数。如果套接字在超时时间间隔内
// 未被接受，则脚本执行将终止。
// 默认的超时值为 10 秒。如果您发现
// 默认时间不够，请做相应
// 修改。
void lrs_set_accept_timeout ( long seconds, long u_sec );

// lrs_set_connect_timeout 函数设置连接
// 到套接字的超时值。请将这个可编程（非录制）
// 的函数插到连接命令 lrs_create_socket
// 之前。
void lrs_set_connect_timeout ( long seconds, long u_sec );

// lrs_set_receive_option 函数为
// lrs_receive 设置套接字接收选项。Option
// 指明在出现不匹配或检测到终止字符串时，
// 停止接收套接字信息。注意，该选项不会应用
// 于通过 lrs_receive_ex 进行的任何接收
// 操作。该函数设置的选项应用于该函数之后
// 出现的所有 lrs_receive，除非该选项被某个
// 后续 lrs_set_receive_option 调用重置。
int lrs_set_receive_option ( int option, int value, [char * terminator] );

// lrs_set_recv_timeout 函数设置 Vugen 从套接字
// 接收到期望数据之前等待的时间段。
void lrs_set_recv_timeout ( long sec, long u_sec );

// lrs_set_recv_timeout2 函数设置在套接字
// 上接收数据的超时限制。当 lrs_receive 接收
// 到数据缓冲区后，它将其大小与期望数据进行
// 比较。如果缓冲区大小不匹配，它将执行
// 更多的循环，重新读取输入套接字数据，
// 直到达到限制值 timeout2。可以在
// 完整日志跟踪中查看这些循环。
```

```
void lrs_set_recv_timeout2( long sec, long u_sec );

// lrs_set_send_buffer 函数指定下次调用
// lrs_send 时要发送的缓冲区。将发送
// lrs_set_send_buffer 中指定的缓冲区，
// 而不是函数 lrs_send 中指定的缓冲区。
int lrs_set_send_buffer ( char *s_desc, char *buffer, int size );

// lrs_set_send_timeout 函数设置向可写
// 套接字发送数据的超时值。可以为超时值指定
// 秒数和毫秒数。
void lrs_set_send_timeout ( long sec, long u_sec );

// lrs_set_socket_handler 函数为指定
// 套接字设置套接字句柄。
int lrs_set_socket_handler ( char *s_desc, int handler );

// lrs_set_socket_options 函数为指定
// 缓冲区设置一个选项。
int lrs_set_socket_options ( char *s_desc, int option, char *option_value );

// lrs_startup 函数初始化 Windows
// Sockets DLL。它指定可用于本应用的最高
// Windows Sockets 版本。该函数
// 必须在所有其他 LRS 函数之前执行。
// 通常，它出现于脚本的 vuser_init
// 部分。如果该函数失败，执行
// 将立即终止。
int lrs_startup ( int version );

// soap_request 函数执行一个 SOAP
// 请求。它向指定的 URL 发送 SOAP 包，
// 并接收服务器响应。
int soap_request (const char *StepName, URL, <XMLEnvelope>, LAST]);

// web_service_call 函数调用 Web 服务。
int web_service_call (const char *StepName, URL, <List of specifications>,
[BEGIN_ARGUMENTS, Arguments, END_ARGUMENTS,] [BEGIN_RESULT, Results,
END_RESULT] LAST );

// lr_xml_get_values 函数在 XML
// 输入字符串 XML 中查询匹配查询条件的值。
int lr_xml_get_values ( <List of specifications> [, <List of optional specifications> ] [, LAST]);
```

```
// lr_xml_set_values 函数在 XML
// 输入字符串 XML 中查询匹配查询条件的值，
// 并将 Value 或 ValueParam 设置为
// 查询匹配的元素的值。
int lr_xml_set_values ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// lr_xml_extract 函数查询 XML 输入字符串
// XML，并提取 XML 树中匹配查询条件
// 的片段。输出参数 XMLFragmentParam
// 包含提取的片段。
int lr_xml_extract ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// lr_xml_delete 函数在 XML 输入字符串
// XML 中进行查询，并删除与查询条件匹配
// 的 XML 树片段。可以通过在 XML 查询
// 中指定元素名称或其属性来删除
// 元素。输出参数 ResultParam 包含
// 删除后经修改的 XML 字符串（使用
// 源文档编码）。
int lr_xml_delete ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// lr_xml_replace 函数在 XML 输入字符串
// XML 中查询与查询条件匹配的值，并
// 用 XmlFragment 或 XmlFragmentParam
// 将它们替换为与查询匹配的元素的值。
// 可以通过在 XML 查询中指定元素名称
// 或属性来替换元素。结果字符串
// 放在 ResultParam 中（使用源文档
// 编码）。
int lr_xml_replace ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// lr_xml_insert 函数在 XML 输入字符串
// XML 中查询与查询条件匹配的值。然后，它在
// 查询返回的 XML 字符串中的一个或多个位置
// 处插入 XmlFragment 或 XmlFragmentParam。
int lr_xml_insert ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// lr_xml_find 函数在 XML 输入字符串 XML 中
// 查询与条件（Value 或 ValueParam）查询的值，
// 并返回出现次数。如果 SelectAll 为“no”，
// 则 lr_xml_find 返回 1 或 0。
int lr_xml_find ( <List of specifications> [, <List of optional specifications> ] [, LAST]);

// 函数 lr_xml_transform 使用样式表
```

```
// 中的可扩展样式表语言 (XSL) 规范转换
// XML 输入字符串 XML，并将作为结果产生
// 的格式化数据保存在 ResultParam 中
// （使用源文档编码）。
int lr_xml_transform ( <List of specifications> ]);
```