SYBASE®
An SAP® Company

用户定义的函数

Sybase IQ 15.4

文档 ID： DC01138-01-1540-01
最后修订日期： 2011 年 11 月

目录

读者

用户定义的函数指南适用于SQL分析人员、C开发人员、C++开发人员、以及想要扩展Sybase® IQ功能的Java开发人员。

作为开发人员，请使用任务、概念以及API参考资料编写外部非SQL用户定义函数。

作为SQL分析人员，请使用本指南开发参考了外部非SQL用户定义函数的SQL查询。

了解用户定义函数

了解如何在 Sybase IQ 中使用用户定义函数。

Sybase IQ 允许用户定义函数 (UDF)，该函数在数据库容器中执行。UDF 执行功能作为可选组件可用于 Sybase IQ 或者 RAP – Trading Edition Enterprise® 的 RAPStore 组件。

您必须获得专门许可方能使用这些外部 C/C++ UDF 接口。

这些外部 C/C++ UDF 不同于交互式 SQL UDF（用于早期版本的 Sybase IQ）。交互式 SQL UDF 未作改变，且无需特别许可。

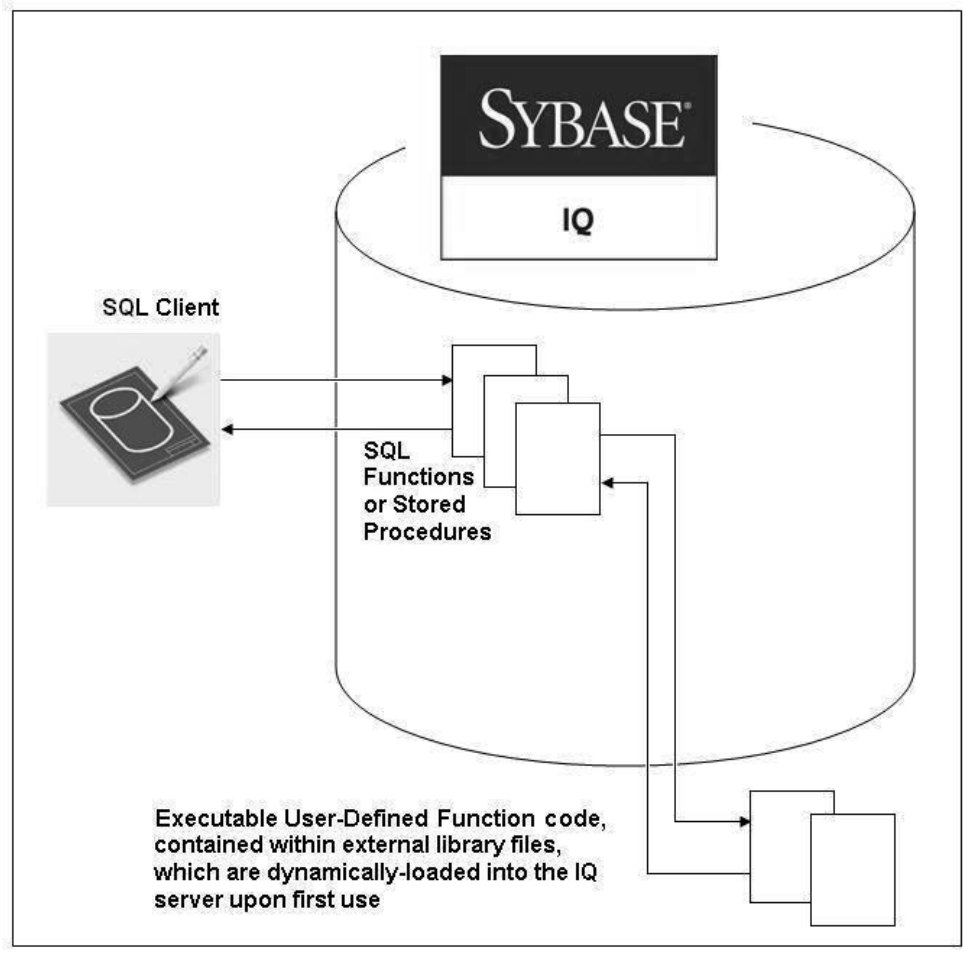
在 Sybase IQ 中执行的 UDF 充分利用了服务器的卓越性能，同时还使用户可灵活分析数据并为其提供灵活的编程解决方案。 用户定义函数由两部分组成：

- UDF 声明，以及
- UDF 可执行代码

UDF 是在 SQL 环境中通过可说明参数并提供对外部库的引用的 SQL 函数或存储过程进行声明的。

UDF 的实际可执行部分包含于外部（共享对象或动态负载）库文件中，当 UDF 声明函数或与该库相关的存储过程被首次调用时，服务器将自动装载该库文件。装载后，该库仍驻留于服务器中，以便通过后续调用引用该库的 SQL 函数或存储过程来进行快速访问。

Sybase IQ 用户定义函数的体系结构如下图所示。



Sybase IQ 支持高性能同进程的外部 C/C++ 用户定义函数。此种 UDF 样式支持使用 C/C++ 代码（遵循本指南中的接口描述）编写函数。

首先将 UDF 的 C/C++ 源代码编译为一个或多个外部库，而后在需要时将这些库装载入服务器的进程空间。通过 SQL 函数向服务器定义 UDF 的调用机制。当从 SQL 查询调用 SQL 函数时，服务器会装载相应的库（如果尚未装载）。

为了便于管理 UDF 的安装过程，Sybase 建议 UDF 开发人员将众多 UDF 函数打包在单个库中。

为便于构建 UDF，Sybase IQ 包括一个基于 C 语言的 API。此 API 由一组预定义的 UDF 入口点、一个明确定义的上下文数据结构、以及一系列 SQL 回调函数（提供从 UDF 至服务器的通信机制）组成。Sybase IQ UDF API 允许软件供应商和专业的最终用户开发、打包和销售自己的 UDF。

学习路线图：UDF 的类型

Sybase IQ 中可用的用户定义函数 (UDF) 类型

UDF 类型	描述	所需的许可证	请参见
UDF (SQL)	使用 SQL 编写的用户定义函数。	无	《系统管理指南，卷 2》>使用过程和批处理 > 用户定义函数简介
使用 C 或 C++ 编写的标量 UDF	操作单值的外部 V3 C/C++ 过程。	IQ_UDF	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
使用 C 或 C++ 编写的标量 UDF	操作单值的外部 V4 C/C++ 过程。	IQ_IDA	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
使用 C 或 C++ 编写的集合 UDF	操作多值的外部 V3 C/C++ 过程。集合 UDF 有时也称为 UDA 或者 UDAF。编写集合 UDF 的上下文结构与编写标量 UDF 的上下文结构稍有不同。	IQ_UDF	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
使用 C 或 C++ 编写的集合 UDF	操作多值的外部 V4 C/C++ 过程。集合 UDF 有时也称为 UDA 或者 UDAF。编写集合 UDF 的上下文结构与编写标量 UDF 的上下文结构稍有不同。	IQ_IDA	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
表 UDF	能够生成一组行的外部 C/C++ 过程，可用作 SQL 语句的 **FROM** 子句的表表达式。	IQ_IDA	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
参数化表函数	表 UDF 除了可以接受标量参数外，还可以接受表（非标量）参数，并且可在行集分区上并行执行。也称为参数化表用户定义函数。	IQ_IDA	学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
Java 标量 UDF	以 Java 代码实现的进程外（外部环境）标量用户定义函数。	无	Java 外部环境 （第 322 页）
Java 表 UDF	以 Java 代码实现的进程外（外部环境）表 UDF。	无	Java 外部环境 （第 322 页）

学习路线图：外部 C 和 C++ UDF 类型

是通过 IQ_IDA 许可证提供的高性能进程中外部 C 和 C++ 用户定义函数。

必须有 IQ_UDF 或 IQ_IDA 许可证，才能使用第 3 版 API。必须有 IQ_IDA 许可证才能使用第 4 版 API。

UDF 类型	输入参数	返回	API	请参见：
标量 UDF	标量	一个标量值	第3版，第4版	声明和定义用户定义的标量函数（第 32 页）
集合 UDF	标量	一个标量值	第3版，第4版	声明和定义集合 UDF （第 46 页）
表 UDF	标量	表	第 4 版	表 UDF 和 TPF （第 93 页）
表参数化函数 (TPF)	标量和表	表	第 4 版	表参数化函数 （第 128 页）

这些 UDF 可以是确定性 UDF，也可以是非确定性 UDF。函数的计算结果可以取决于输入参数和数据（确定性函数），也可以取决于一些随机行为（非确定性函数）。对于非确定性 UDF 的参数，通常需要用随机种子作为其中的一个输入参数。

符合 Sybase IQ 数据库的用户定义函数

开发能够与 Sybase IQ 数据库一起使用的用户定义函数。

无缝执行

UDF 必须能够在数据库容器中无缝运行。虽然 Sybase IQ 是一个由众多文件所组成的复杂产品，但是它同用户之间的交互主要是通过一个服务器进程 (iqsrv15) 来完成的，其中使用了业界标准的结构化查询语言 (SQL)。应该完全使用 SQL 命令来执行 UDF；用户在使用 UDF 时无需了解其底层实现方法。

使用 `EXTFN_V3_API` 和 `EXTFN_V4_API` 所提供的回调函数， UDF 能够对消息文件 (`.iqmsg`) 执行写入操作。

UDF 应该对由 `EXTFN_V3_API` 和 `EXTFN_V4_API` 所定义的内存和临时结果进行管理。

Sybase IQ 是一个多用户应用程序。多名用户可以同时执行同一个 UDF。某些 OLAP 查询会导致在同一个查询中多次执行某个 UDF，有时为并行执行。有关设置 UDF 并行执行的详细信息，请参见 集合 UDF 调用模式 （第 82 页）。

国际化

Sybase IQ 已被国际化，供全球用户使用。错误消息被记录于外部文件，这使您无需大量改动代码便可将错误消息本地化为新的语言。

若要支持多种语言，UDF 也应进行国际化处理。通常，大多数 UDF 的操作对象为数值型数据。在某些情况下，UDF 可能会接受字符串关键字，并将其作为一个或多个参数。除了 UDF 所用到的异常文本和日志消息外，请将这些关键字也至于外部文件中。

Sybase IQ 也被本地化处理为一些非英语语言。若要支持将其本地化为 Sybase IQ 所支持的语言，Sybase 建议您对 UDF 进行国际化处理，以便其他独立组织以后对其进行本地化处理。

有关 Sybase IQ 中的国际语言支持的详细信息，请参见《系统管理指南：第一卷》>“国际语言和字符集”。

也请参见 于 www.Sybase.com。与输入关键字、异常信息和日志条目相对，该文件讨论了如何使用多字节数据进行调试。

平台差异

开发能够运行于 Sybase IQ 所支持的不同平台的 UDF。Sybase IQ 15.x 服务器运行于 64 位体系结构之上，并且为运行 MS Windows（64 位）系列操作系统的多个平台所支持。Sybase IQ 也为多个版本的 UNIX（64 位）所支持，包括 Solaris、HP-UX、AIX 和 Linux。

要避免的做法

了解用于创建用户定义的函数的好的做法。

- 请不要在 SQL 注册脚本中对库路径进行硬编码。这种做法将会降低灵活性，不便于用户将 UDF 安装至与 Sybase IQ 相同的目录中。
- 请不要编写含义模糊的代码，或者创建会导致意外死循环、且未向用户提供取消 UDF 调用的机制的代码（参见 **'get_is_cancelled()'** 函数）。
- 不要执行每次调用时都重复的复杂或使用大量内存的操作。在对包含成千上万行的表调用 UDF 时，高效执行变得至关重要。Sybase 建议您每次为一千至数千行分配内存块，而非逐行进行。
- 不要从 UDF 内打开数据库连接或执行数据库操作。UDF 执行所需的所有参数和数据必须作为参数传递给 UDF。
- 在命名 UDF 时，不要使用保留字。

用户定义函数的命名约定

UDF 名称必须与 Sybase IQ 中的其它标识符遵循相同的限制。

Sybase IQ 标识符的最大长度是 128 个字节。为方便使用，UDF 的名称应该以字母字符开头。Sybase IQ 所定义的字母字符包括字母表中的字母、下划线字符 (_)、at 符号

(@)、数字或井号 (#) 和美元符号 ($)。UDF 的名称应该完全由这些字母字符和数字（数字 0 至 9）组成。UDF 的名称不应与 SQL 的保留字相冲突。SQL 保留字列表如 Sybase IQ《参考：构件块、表和过程》>“SQL 语言元素”>“保留字”中所示。

虽然 UDF 名称（与其它标识符一样）也可以包含保留字、空格、上面列出的那些以外的字符，并可以以非字母字符开头，但不建议这样做。如果 UDF 名称具有其中任何特征，则您必须用引号或方括号括住它们，这会使它们的使用变得更加困难。

UDF 与其它 SQL 函数和存储过程驻留在相同的名称空间中。为避免与现有存储过程和函数相冲突，应在 UDF 前面加上唯一的简短（2 至 5 个字母）首字母缩略词和下划线。选择与本地环境中已定义的其它 SQL 函数或存储过程不冲突的 UDF 名称。

下面是一些已经使用的前缀：

- **debugger_tutorial** – 在本机安装 Sybase IQ 时所交付的存储过程。
- **ManageContacts** – 随 Sybase IQ 的演示数据库而交付的存储过程。
- **Show** – 用于显示来自 Sybase IQ 演示数据库的数据的存储过程。
- **sp_Detect_MPX_DDL_conflicts** – 在本机安装 Sybase IQ 时所交付的存储过程。
- **sp_iqevbegintxn** – 在本机安装 Sybase IQ 时所交付的存储过程。
- **sp_iqmpx** – Sybase IQ 所提供的协助 multiplex 管理的函数和存储过程。
- **ts_** – 可选的财务时序和预测函数。

SQL 数据类型

UDF 声明仅支持某些 SQL 数据类型。

可以在 UDF 声明中作为 UDF 参数的数据类型或返回值数据类型使用以下 SQL 数据类型：

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
UNSIGNED BIGINT	DT_UNSBIGINT	a_sql_uint64	无符号的 64 位整数，需要 8 个字节的存储空间。
BIGINT	DT_BIGINT	a_sql_int64	带符号的 64 位 整数，需要 8 个字节的存储空间。
UNSIGNED INT	DT_UNSINT	a_sql_uint32	无符号的 32 位整数，需要 4 个字节的存储空间。
INT	DT_INT	a_sql_int32	带符号的 32 位 整数，需要 4 个字节的存储空间。
SMALLINT	DT_SMALLINT	short	带符号的 16 位整数，需要 2 个字节的存储空间。

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
TINYINT	DT_TINYINT	unsigned char	无符号的 8 位整数，需要 1 个字节的存储空间。
DOUBLE	DT_DOUBLE	double	带符号的 64 位 双精度浮点数，需要 8 个字节的存储空间。
REAL	DT_FLOAT	float	带符号的 32 位 浮点数，需要 4 个字节的存储空间。
FLOAT	DT_FLOAT	float	在 SQL 中， FLOAT 是带符号的 32 位浮点 数（需要 4 个字节的存储空间），或者带符号的 64 位双精度 浮点数 (需要 8 个字节的存储空间），具体情况取决于相关联的精度。 仅当未提供可选的 FLOAT 数据类型精度时，才能在 UDF 声明中使用 SQL 数据类型 FLOAT。没有精度时， FLOAT 是 REAL 的同义词。
CHAR(<n>)	DT_FIXCHAR	char	在数据库缺省 字符集中填补空白的定长字符串 。长度上限“<n>”是 32767。 数据不以空字节结尾。
VARCHAR(<n>)	DT_VARCHAR	char	数据库缺省 字符集中的变长字符串 。长度上限“<n>”是 32767。数据不以空字节结尾。对于 UDF 输入参数， 如果值不是空值，则实际长度必须从 an_extfn_value 结构中的 *total_len* 字段中检索。 同样，对于这种类型的 UDF 结果，实际长度也必须在 *total_len* 字段中设置。

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
LONG VARCHAR(<n>) 或 CLOB	DT_VARCHAR	char	数据库缺省 字符集中的变长字符串 。LONG VARCHAR 数据类型仅可用作输入参数，不能用作返回值数据类型。对于第 3 版 UDF，长度上限“<n>”是 4GB（千兆字节）。数据不以空字节结尾。LONG VARCHAR 可对数据类型指定 WD 索引或 TEXT 索引。对于 UDF 输入参数， 如果值不是空值，则实际长度必须从 an_extfn_value 结构中的 *total_len* 字段中检索。 如果现有标量或集合 UDF 包含通过 `get_value()` 和 `get_piece()` 方法读取值片段的循环，则无需重建或重新编译现有标量或集合 UDF，即可使用 LOB 数据类型作为输入参数。对于第 3 版 UDF， *remain_len* > 0 或达到 4GB 后 循环才会停止（第 4 版中没有 4GB 的限制）。 表 UDF 和 TPF 不用 `get_piece()` 方法处理和检索数据。表 UDF 和 TPF 必须改用 `Blob (a_v4_extfn_blob)` API。可用 `blob_length` 决定输入参数长度。 大对象数据支持需要使用单独许可的 Sybase IQ 选项。
BINARY(<n>)	DT_BINARY	unsigned char	填补空字节的 的定长二进制值，长度上限“<n>”是 32767。数据不以空字节结尾。

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
VARBINARY(<n>)	DT_BINARY	unsigned char	变长 二进制值，长度上限“<n>”是 32767。数据不以空字节结尾。对于 UDF 输入参数， 如果值不是空值，则实际长度必须从 an_extfn_value 结构中的 *total_len* 字段中检索。同样，对于这种 UDF 结果，实际长度也必须在 *total_len* 字段中设置。数据不以空字节结尾。
LONG BINARY(<n>) 或 BLOB	DT_BINARY	unsigned char	填补空字节的 的定长二进制值，对于第 3 版 UDF，长度上限“<n>”是 4GB（千兆字节）。LONG BINARY 数据类型仅可用作输入参数，不能用作返回值数据类型。 如果现有标量或集合 UDF 包含通过 get_value() 和 get_piece() 方法读取值片段的循环，则无需重建或重新编译现有标量或集合 UDF，即可使用 LOB 数据类型作为输入参数。对于第 3 版 UDF， *remain_len* > 0 或达到 4GB 后 循环才会停止（第 4 版中没有 4GB 的限制）。 表 UDF 和 TPF 不用 get_piece() 方法处理和检索数据。表 UDF 和 TPF 必须改用 Blob (a_v4_extfn_blob) API。可用 blob_length 决定输入参数长度。 大对象数据支持需要使用单独许可的 Sybase IQ 选项。

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
DATE	DT_TIMESTAMP_STRUCT	unsigned integer	日历日期值，以无符号整数的形式传入或传出 UDF。赋予 UDF 的值保证能在比较和排序运算过程中使用。值越大表明日期越晚。如果需要实际日期组成部分，则 UDF 必须调用 convert_value 函数，才能转换为 DT_TIMESTAMP_STRUCT 型数据。这种数据类型通过以下结构表示日期和时间： `typedef struct sqldatetime { unsigned short year; /* e.g. 1992*/ unsigned char month; /* 0-11 */ unsigned char day_of_week; /* 0-6 0=Sunday, 1=Monday, ... */ unsigned short day_of_year; /* 0-365 */ unsigned char day; /* 1-31 */ unsigned char hour; /* 0-23 */ unsigned char minute; /* 0-59 */ unsigned char second; /* 0-59 */ a_sql_uint32 microsecond; /* 0-999999 */ } SQLDATETIME;`

SQL 数据类型	C 或 C++ 数据类型标识符	C 或 C++ Typedef	描述
	DT_TIMESTAMP_STRUCT	unsigned bigint	用于准确描述给定日期中某个时刻的值 。赋予 UDF 的值保证能在比较和排序运算过程中使用。值越大表明时间越晚。如果需要实际时间组成部分，则 UDF 必须调用 `convert_value` 函数，才能转换为 `DT_TIMESTAMP_STRUCT` 型数据。
`DATETIME`、`SMALLDATETIME` 或 `TIMESTAMP`	DT_TIMESTAMP_STRUCT	unsigned bigint	日历日期和时间值。赋予 UDF 的值保证能在比较和排序运算过程中使用。值越大表明日期时间越晚。如果需要实际时间组成部分，则 UDF 必须调用 `convert_value` 函数，才能转换为 `DT_TIMESTAMP_STRUCT` 型数据。
`TABLE`	DT_EXTFN_TABLE	a_v4_extfn_table	用于表示输入表参数结果集。只能将这种数据类型用于 TPF。

Unsupported Data Types

无法在 UDF 声明中作为 UDF 参数的数据类型或返回值数据类型使用以下 SQL 数据类型：

- `BIT` – 通常应该在 UDF 声明中作为 `TINYINT` 数据类型来处理，然后从 `BIT` 进行的隐式数据类型转换将自动处理数值转换。
- `DECIMAL`（<精度>、<标度>）或 `NUMERIC`（<精度>、<标度>） – 取决于使用情况，`DECIMAL` 通常作为 `DOUBLE` 数据类型来处理，但是可以强加各种约定以便能够使用 `INT` 或 `BIGINT` 数据类型。
- `LONG VARCHAR (CLOB)` — 只能用作输入参数，不能用作返回值数据类型。直通 TPF 是一个例外，此时 `LONG VARCHAR` 用作返回值数据类型。
- `LONG BINARY (BLOB)` — 只能用作输入参数，不能用作返回值数据类型。直通 TPF 是一个例外，此时 `LONG BINARY` 用作返回值数据类型。
- `TEXT` – 目前不支持。

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）
- convert_value （第 280 页）

- 表 (a_v4_extfn_table) （第 290 页）

构建 UDF

设计、构建和测试 UDF。

用户定义函数的设计基础

当开发 UDF 时，请牢记一些基本的注意事项。

本文档假定 UDF 开发人员熟悉软件开发基础知识，包括程序设计与开发以及独立测试的技能。

除了标准的软件开发习惯之外，使用非 Java 语言的 UDF 开发人员应该记住，其所开发的代码将运行于 Sybase IQ 数据库容器之中，因此有必要了解数据库容器所带来的限制。

集合 UDF 的开发人员还应该熟悉 OLAP 查询以及如何将其转换为 UDF 调用模式。

因为 UDF 可能同时被多个线程调用，所以必须将 UDF 构造为线程安全的。

示例代码

UDF 源代码示例随产品一起提供。最新版本的示例代码始终随 Sybase IQ 最新版一起提供。

在 UNIX 平台之上，示例 UDF 代码位于 `$SYBASE/IQ-15_4/samples/udf` 之中（$SYBASE 为安装根目录）。

在 Windows 平台上，UDF 代码示例位于以下路径：`C:\Documents and Settings\All Users\SybaseIQ\samples\udf`。

用户定义函数指南 中所述的示例 UDF 代码可能并非最新版本（随 Sybase IQ 产品交付）。 示例 UDF 源代码的最新修订记录于针对您的操作系统平台的 Sybase IQ 发行公告。

设置动态库接口

指定要在动态可链接库中使用的接口样式。

每个动态装载的库只能包含此定义的一个副本：

```
extern "C" a_sql_uint32 extfn_use_new_api(void )
{
     return EXTFN_V4_API;
}
```

此定义用于告知服务器使用的是哪种接口 样式，从而告知如何访问此动态链接库中所定义的 UDF 。对于高性能 UDF，只支持新接口样式 `EXTFN_V3_API` 和 `EXTFN_V4_API`。

向第 4 版 API 升级

Sybase 建议升级到随 15.4 附带的第 4 版 API。

前提条件

安装 15.4 版 Sybase IQ 服务器。

过程

如果已为 15.1 版、15.2 版或 15.3 版 Sybase IQ 服务器开发标量或集合 UDF，则这些 UDF 会使用第 3 版 API 接口样式，还会引用 `extfnapiv3.h` 头文件。修改旧 C 或 C++ 外部库文件，即可引用 `extfnapiv4.h` 头文件。

原有第 3 版标量和集合函数都会继续按设计发挥作用。但是，要在 PlexQ 中利用标量和集合分配，必须将头文件和库版本升级到第 4 版。无需更改标量或集合函数类型定义名称。

1. 打开 C 或 C++ 外部库文件后，定义标量 或集合用户定义函数。
2. 找到所有 `#include 'extfnapiv3.h'` 实例，然后改为 `#include 'extfnapiv4.h'`。
3. 将动态库接口设置为 `EXTFN_V4_API`。
4. 重新生成。

下一

Sybase 合作伙伴必须确保库可以将 `extfn_get_license_info` 导出为条目。

另请参见

- 外部函数原型 （第 91 页）
- 许可证信息 (a_v4_extfn_license_info) （第 285 页）
- 定义集合 UDF （第 52 页）
- Defining a Scalar UDF （第 37 页）
- 开发表 UDF （第 98 页）
- 开发 TPF （第 129 页）

库版本（extfn_get_library_version）

使用 `extfn_get_library_version` 方法，从当前 multiplex 节点上提取库的版本。仅当安装的库与其他节点兼容时，服务器才会考虑对跨多个 multiplex 节点的查询进行拆分。

实现
v4 library 可以定义可选的入口点：

```
size_t extfn_get_library_version( uint8 *buff, size_t len );
```

描述
提取库版本的方法执行于库的级别，其名称不加 a_v4 前缀。

如果 v4 库定义了可选的入口点，则服务器允许对其他节点分发查询。入口点使用库版本字符串填充提供的缓冲区（包含 ASCII 字符的 C 样式字符串，以 **\0** 结尾），并且返回所填充版本字符串的实际大小（该值限定于 256 字节以内）。

如果未定义入口点，则服务器不对 multiplex 中的其他节点分发 UDF 。

另请参见
- 库版本兼容性 (extfn_check_version_compatibility) （第 17 页）
- 设置动态库接口 （第 15 页）

库版本兼容性 (extfn_check_version_compatibility)

请使用 `extfn_check_version_compatibility` 方法，为 multiplex 中各个节点的库版本定义兼容性标准。

实现
v4 library 可以定义可选的入口点：

```
a_bool extfn_check_version_compatibility( uint8 *buff, size_t
len );
```

描述
提取库版本的方法执行于库的级别，其名称不加 a_v4 前缀。

该可选入口点接受包含版本字符串和版本字符串长度的缓冲区。其返回值将指出目标节点的库版本与版本字符串参数是否彼此兼容。库的开发人员对兼容性标准作了定义。

同 `extfn_get_library_version` 进行交互
在检查版本兼容型之前，领导节点将调用 `extfn_get_library_version`。如果 `extfn_get_library_version` 未在领导节点上予以实现，则不进行分发。如果

extfn_get_library_version 未在领导节点上予以实现，则 UDF 或 TPF 可进行分发。适合于分发并不能保证一定会进行分布式查询处理。

语句 extfn_get_library_version 方法可以返回一个长度为 0 的字符串；然而，这并不意味着 extfn_get_library_version 未被实现。

注意： 如果 extfn_get_library_version 返回一个长度为 0 的字符串，则 TPF 或 UDF 仍然可进行分发。

如果 extfn_get_library_version 返回一个长度为 0 的字符串，则工作节点是否接受分布式工作取决于 extfn_check_version_compatibility 在工作节点上的实现。工作节点需要一个与其兼容的库，以便处理分布式工作。

另请参见

- 库版本（extfn_get_library_version） （第 17 页）
- 设置动态库接口 （第 15 页）

许可证信息 (extfn_get_license_info)

如果您为 Sybase 的设计合作伙伴，请实现库级函数 extfn_get_license_info，以使服务器能够通过 v4 UDF 获取许可证信息。

数据类型
an_extfn_license_info

实现

```
(_entry an_extfn_get_license_info) ( an_extfn_license_info
**license_info );
```

参数
license_info 是一个输出参数，返回从库中接收到的许可证信息。 您可以在 a_v4_extfn_license_info 结构中 定义许可证信息。

描述
Sybase 合作伙伴必须在 a_v4_extfn_license_info 结构中指定由 Sybase 提供的许可证密钥，同时必须确保库将 extfn_get_license_info 导出为一个入口点。

添加 extfn_get_license_info 方法

如果您为 Sybase 的设计合作伙伴，请填充 a_v4_extfn_license_info 中的字符串，并将 extfn_get_license_info 定义为一个 v4 入口点。

1. 在 a_v4_extfn_license_info 结构中，请指定您的公司名称。最大长度为 255 个字符。

2. 在 a_v4_extfn_license_info 结构中，请指定其他库信息，例如库的版本和内部版本号。最大长度为 255 个字符。
3. 在 a_v4_extfn_license_info 结构中，请输入由 Sybase 所提供的许可证密钥。
4. 请确保库将 extfn_get_license_info 导出为一个入口点。

```
a_v4_extfn_license_info my_info = {
    1,
    "Company Name",
    "Library Info String",
    (void *)"KEY_STRING"
};

void SQL_CALLBACK extfn_get_license_info( an_extfn_license_info
**license_info )
/
************************************************************************
************************/
{
    *license_info = (an_extfn_license_info *)& my_info;
}
```

编译并链接源代码以构建动态链接库

在为任何用户定义的函数构建动态可链接库时，使用编译和链接开关。

1. UDF 动态可链接库必须包含函数 **extfn_use_new_api()** 的实现。此函数的源代码在“设置动态链接库接口”（第 15 页）中。此函数将库中所有函数需遵循的 API 样式通知服务器。示例源文件 my_main.cxx 包含此函数，并且无需修改即可使用。
2. UDF 动态可链接库还必须包含至少一个 UDF 函数的对象代码。UDF 动态可链接库可选择性地包含多个 UDF。
3. 将各个 UDF 的对象代码以及 **extfn_use_new_api()** 链接在一起，以形成单个库。例如，构建“libudfex:”库
   - 编译每个源文件以生成对象文件：

     ```
     my_main.cxx
     my_bit_or.cxx
     my_bit_xor.cxx
     my_interpolate.cxx
     my_plus.cxx
     my_plus_counter.cxx
     my_sum.cxx
     my_byte_length.cxx
     my_md5.cxx
     my_toupper.cxx
     tpf_agg.cxx
     tpf_blob.cxx
     tpf_dt.cxx
     ```

```
tpf_filt.cxx
tpf_oby.cxx
tpf_pby.cxx
tpf_rg_1.cxx
tpf_rg_2.cxx
udf_blob.cxx
udf_main.cxx
udf_rg_1.cxx
udf_rg_2.cxx
udf_rg_3.cxx
udf_utils.cxx
```

- 将生成的每个对象链接至单个库。

在完成动态链接库的编译和链接之后，请执行如下任务：

- （建议）更新 **CREATE FUNCTION ... EXTERNAL NAME** 或 **CREATE PROCEDURE ... EXTERNAL NAME**，为 UDF 库包含 一个显式路径名。
- 将 UDF 库文件置于 保存所有 Sybase IQ 库的目录之中。
- 以库 装载路径启动服务器，该路径包括 UDF 库的存储位置。

  在 UNIX 平台上，请修改 `start_iq startup` 脚本中的 LD_LIBRARY_PATH。尽管 LD_LIBRARY_PATH 对于所有的 UNIX 变量是通用的，但是在 HP 平台上首选为 SHLIB_PATH，在 AIX 平台上首选为 LIB_PATH。

  在 UNIX 平台上，外部名称格式可以包含一个 完全限定的名称，此时，将不会使用 LD_LIBRARY_PATH。在 Windows 平台上，将不会使用完全限定的 名称，并且使用 PATH 环境变量来定义库搜索路径。

4. 在 Windows 平台上运行 `iqdir15/samples/udf/build.bat`。在 UNIX 平台上运行 `iqdir15/samples/udf/build.sh`。

编译并链接适用于 Windows 的示例 UDF

运行 `build.bat` 脚本，以编译并链接 `samples\udf` 目录中的示例标量和集合 UDF、表 UDF 以及 TPF。

1. 导航至 `%ALLUSERSPROFILE%\samples\udf`。
2. 运行 `build.bat`：

参数	描述
-clean	删除对象和构建目录
-v3	构建使用 v3 API 的示例标量和集合 UDF
-v4	（默认）构建使用 v4 API 的示例表 UDF 和 TPF

编译并链接适用于 UNIX 的示例 UDF

运行 `build.sh` 脚本，以编译并链接 `samples/udf` 目录中的示例标量和集合 UDF、表 UDF 以及 TPF。

1. 导航至 `$IQDIR15/samples/udf`。
2. 运行 `build.sh`：

参数	描述
-clean	删除对象和构建目录
-v3	构建使用 v3 API 的示例标量和集合 UDF
-v4	（默认）构建使用 v4 API 的示例表 UDF 和 TPF

AIX Switches

在 AIX 上构建共享库时使用下列编译和链接开关。

xlC `10.0` 在 PowerPC 上

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

注意： 如果要在 AIX 6.1 系统上进行编译，则最低级别的 xlC 编译器是 `10.0`。

编译开关

```
-q64 -qarch=ppc64 -qtbtable=full -qsrcmsg -qalign=natural -
qnoansialias
-qmaxmem=-1 -qenum=int -qhalt=e -qflag=w -qthreaded -
qxflags=NLOOPING
-qtmplinst=none -qthreaded
```

链接开关

```
-brtl -G -lg  -lpthreads_compat -lpthreads -lm_r -ldl -bnolibpath -
v
```

HP-UX Switches

在 HP-UX 上构建共享库时使用下列编译和链接开关。

aCC `6.24` 在 Itanium 上

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译开关

```
+noeh -ext +W740,749,829 +W1031 +DD64 +DSblended +FPD -Aa +ub
-U_HP_INSTANTIATE_T_IN_LIB -Wc,-ansi_for_scope,on -mt -z
```

链接开关

```
 -b -Wl,+s
```

Linux Switches

在 Linux 上构建共享库时使用下列编译和链接开关。

x86 上的 g++ 4.1.1

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译开关

```
-fPIC -fsigned-char -fno-exceptions -pthread -fno-omit-frame-
pointer
-Wno-deprecated -Wno-ctor-dtor-privacy -O2 -Wall
```

注意： 在编译用于在 Linux 上构建共享库的 C++ 应用程序时，将 **-O2** 和 **-Wall** 开关添加到编译 UDF 开关列表中可以缩短计算时间。

链接开关

```
-ldl -lnsl -lm -lpthread -shared -Wl,-Bsymbolic -Wl,-shared
```

注意： 也可在 Linux 上使用 gcc。用 **gcc** 链接时，可通过将 `-lstdc++` 添加到链接开关，在 C++ 运行时库中进行链接。

示例

- *示例 1*

  ```
  g++ -c my_interpolate.cxx -fPIC -fsigned-char -fno-exceptions -
  pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *示例 2*

  ```
  g++ -c my_main.cxx -fPIC -fsigned-char -fno-exceptions -pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *示例 3*

  ```
  ld -G my_main.o my_interpolate.o -ldl -lnsl -lm -lpthread -shared
        -o my_udf_library.so
  ```

xlC 10.0 在 PowerPC 上

编译开关

```
 -q64 -qarch=ppc64 -qcheck=nullptr -qinfo=gen -qtbtable=full -
qsrcmsg
-qnoansialias -qminimaltoc -qmaxmem=-1 -qenum=int -qhalt=e -qflag=w
-qthreaded
-qxflags=NLOOPING -qtmplinst=none
```

链接开关

```
-qmkshrobj -ldl -lg  -qthreaded -lnsl -lm
```

Solaris Switches

在 Solaris 上构建共享库时使用下列编译和链接开关。

SPARC 上的 Sun Studio 12

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译参数

```
-mt -noex +w -KPIC -i -instances=explicit -V -xtarget=ultra3cu -m64
-xlibmopt
-xlibmil -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype
```

链接开关

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat
```

x86 上的 Sun Studio 12

编译开关

```
+w2 -m64 -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype,doubunder -errtags -
mt  -noex
-KPIC -instances=explicit -xlibmopt -xlibmil
```

链接开关

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat -m64
```

Windows Switches

在 Windows 上构建共享库时使用下列编译和链接开关。

x86 上的 Visual Studio 2008

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译和链接开关

本示例用于包含 my_plus 函数的 DLL。必须为 DLL 中所含的每个 UDF 的描述符函数包含一个 EXPORT 开关。

```
cl /Zi /LD   /I includefilepath     my_main.cxx my_plus.cxx  /link /
map
/INCREMENTAL:NO -EXPORT:extfn_use_new_api -EXPORT:my_plus  /
out:libiqudfex.dll
```

示例

环境设置

```
set VCBASE=c:\dev\vc9
      set MSSDK=C:\dev\mssdk6.0a
      set IQINSTALLDIR=C:\Sybase\IQ
      set OBJ_DIR=%IQINSTALLDIR%\IQ-15_4\samples\udf\objs
      set SRC_DIR=%IQINSTALLDIR%\IQ-15_4\samples\udf\src
      call %VCBASE%\VC\bin\vcvars32.bat
```

- *示例 1*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_4\sdk
\include"
     -Fo"%OBJ_DIR%\my_interpolate.o" %SRC_DIR%\my_interpolate.cxx
```

- *示例 2*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_4\sdk
\include"
      -Fo"%OBJ_DIR%\my_main.o" %SRC_DIR%\my_main.cxx
```

- *示例 3*

```
%VCBASE%\VC\bin\amd64\link /LIBPATH:%VCBASE%\VC\lib\amd64
      /LIBPATH:%MSSDK%\lib\bin64 kernel32.lib -manifest -DLL -
nologo
      -MAP:"%OBJ_DIR%\libudfex.map_deco" /OUT:"%OBJ_DIR%
\libudfex.dll"
      "%OBJ_DIR%\my_interpolate.o" "%OBJ_DIR%\my_main.o" /DLL
      -EXPORT:extfn_use_new_api -EXPORT:my_interpolate
```

- *示例 4*

```
%MSSDK%\bin\mt -nologo -manifest "%OBJ_DIR%
\libudfex.dll.manifest"
      -outputresource:"%OBJ_DIR%\libudfex.dll;2"
```

Testing User-Defined Functions

在对 UDF 外部代码进行了编码、编译和链接且定义了相应的 SQL 函数和存储过程后，便可对 UDF 进行测试。

数据库所要求的可靠性极高。在数据库环境中运行 UDF 必须保持这种高可靠性。随着 UDF API 的首次实现，UDF 在 Sybase IQ 服务器中开始运行。如果 UDF 提前中止或异常中止，则 Sybase IQ 服务器可能会终止运行。请对开发环境或测试环境进行全面测试，以确保在任何情况下 UDF 都不会发生提前中止或异常中止。

启用和禁用用户定义的函数

使用 `inmemory_external_procedure` 安全性功能启用或禁用服务器利用 高性能进程中 UDF 的能力。

数据库的数据完整性应该得到保持。在任何情况下，数据均不得丢失、修改、增补或损坏。由于 UDF 在 Sybase IQ 服务器中执行，存在损坏数据的风险，因此，管理内存及使用任何其他指针时要小心。Sybase 强烈建议在只读 Multiplex 节点中安装和执行 UDF。为提供额外保护，请对每个服务器使用受保护功能 **(-sf)** 启动选项，以启用或禁用 UDF 的执行。

注意： 缺省情况下已禁止在 Multiplex 写入程序节点和协调节点中执行 UDF。所有其他节点在缺省情况下启用。

管理员可以通过 在服务器启动命令或配置文件中指定以下代码，为任何服务器启用第 3 版和第 4 版 UDF：

```
-sf -inmemory_external_procedure
```

管理员可以通过 在服务器启动命令或配置文件中指定以下代码，为任何服务器禁用第 3 版和第 4 版 UDF：

```
-sf inmemory_external_procedure
```

中提供了有关 **-sf** 标志的其它信息。 列在 SQL Anywhere 文档中的值不适用于 Sybase IQ，不应使用。

首次执行用户定义函数

为了确保尽可能最安全的环境，Sybase 强烈建议您在采用 multiplex 安装的只读服务器节点安装并调用 UDF 。

Sybase IQ 服务器将不会装载包含 UDF 代码的库，直到首次调用 UDF。由于驻留 UDF 代码的库尚未被装载，UDF 的首次执行速度可能非常慢。装载库之后，相同 UDF 的后续调用或者同一个库中的其他 UDF 调用将实现预期的执行速度。

- **使用存储过程 SA_EXTERNAL_LIBRARY_UNLOAD 的库** – 当 Sybase IQ 服务器停止和重启时，不重新装载这些库。

在空闲时段执行维护操作、且需要关闭和重启 IQ 服务器时，请于服务器重启后运行一些测试性查询。这样可以确保内存中装载有正确的库，以便在工作期间优化查询性能。

Managing External Libraries

首次调用 UDF 时将会装载外部库。在服务器正常运行期间，所装载的库将始终保持装载状态。当调用 **CREATE FUNCTION** 或 **CREATE PROCEDURE** 时，不会对库进行卸载；同样，当调用 **DROP FUNCTION** 或 **DROP PROCEDURE** 时，也不会对库进行自动卸载。

如果必须更新库版本，则 **dbo.sa_external_library_unload** 过程将强制对库进行重载（不重启服务器）。仅当讨论中的库当前未被使用，对外部库进行卸载的调用才会成功执行。该过程采用了一个可选参数 (long varchar)，以指定即将卸载的库的名称。如果未指定任何参数，则将卸载所有未使用的外部库。

注意： 在替换 动态链接库之前，请从运行中的 Sybase IQ 服务器上卸载现有的库。如果您未卸载该库，则服务器可能会发生运行故障。在替换 动态 链接库之前，请关闭 Sybase IQ 服务器，或者使用 **sa_external_library_unload** 函数 重载该库。

在 Windows 平台上，使用以下工具卸载外部函数库：

```
call sa_external_library_unload('library.dll')
```

对于 UNIX，使用以下命令卸载外部函数库：

```
call sa_external_library_unload('library.so')
```

如果某个注册函数使用了完整路径，例如 `/abc/def/library`，则首先注销该函数。

在 Windows 平台上，使用

```
call sa_external_library_unload('\abc\def\library.dll')
```

在 UNIX 平台上，使用

```
call sa_external_library_unload('/abc/def/library.so')
```

注意： 仅当库尚未位于库装载路径中的目录中时，SQL 函数声明中才需要库路径。

控制错误检查和调用跟踪

计算涉及第 3 版和第 4 版外部用户定义函数的语句时， **external_UDF_execution_mode** 选项 控制着错误检查和调用跟踪的执行次数。

可以在 UDF 开发期间使用 **external_UDF_execution_mode** 来帮助开发 UDF 时进行的调试。

允许值
0, 1, 2

缺省值
0

范围
可设置为公用、临时或用户。

说明
设为 0（缺省值）时，外部 UDF 求值的方式可以优化使用 UDF 的语句的性能。

如果设置为 1，则会计算外部 UDF，以验证传出或传入 每个 UDF 的信息。此设置适用于标量和集合 UDF。

如果设置为 2，则会计算外部 UDF，以便验证 传出或传入 UDF 的信息，同时验证 `iqmsg` 文件中的日志信息、对 UDF 所提供函数的每一次调用以及 这些函数返回服务器的每次回调。此设置适用于所有 C 或 C++ 外部 UDF。对于表 UDF 和 TPF，会启用内存跟踪。

查看 Sybase IQ 日志文件

Sybase IQ 提供了完善的日志和跟踪功能。如果 UDF 代码遇到问题，UDF 应提供相同或更详细的日志功能。

数据库的日志文件通常会同数据库文件和配置文件放在一起。在 UNIX 平台之上，有两个文件以数据库实例命名，其中一个以 `.stderr` 为扩展名，另一个以 `.stdout` 为扩展名。在 Windows 平台之上，缺省情况下不生成 `stderr` 文件。

为了捕获 Windows 系统中的 `stderr` 消息以及 `stdout` 消息，请重定向 `stdout` 和 `stderr`：

```
iqsrv15.exe @iqdemo.cfg iqdemo.db 2>&1 > iqdemo.stdout
```

Windows 的输出消息与 UNIX 平台上所生成的输出消息稍有不同。

对用户定义函数使用 Microsoft Visual Studio 调试工具

Microsoft Visual Studio 2008 开发人员使用 Microsoft Visual Studio Debugger 对用户定义函数代码做单步执行。

1. 将调试程序附加到运行的服务器：
   ```
   devenv /debugexe "%IQDIR15%\Bin64\iqsrv15.exe"
   ```
2. 转到“调试”(Debug) | “附加到进程”(Attach to Process)
3. 若要同时启动服务器和调试程序，请执行下列操作：
   ```
   devenv /debugexe "%IQDIR15%\bin32\iqsrv15.exe" [commandline
   options for your server]
   ```
   每个平台都有一个调试程序，每个调试程序都有自己的命令行语法。不需要 Sybase IQ 源代码。当执行用户定义函数且在设置的 断点发生中断时，msvs 调试程序将

能够予以识别 。当控制权由用户定义函数返回至服务器时，您将只会看到机器代码。

运行时修改 UDF

许多 Sybase IQ 安装于任务关键型环境之中，在这些环境中，消耗程序对可用性的需求极高。系统管理员必须能够在不对 Sybase IQ 服务器产生影响（或影响很小）的前提下安装和升级 UDF。

在移动、覆盖或删除关联的库文件时，应用程序一定不要尝试访问外部库。因为库会在调用相关的 SQL 函数时自动装载，所以在对现有 UDF 库执行任何类型的维护时，一定要完全按照下列步骤的顺序操作：

1. 确保调用 UDF 的所有用户没有任何正在进行的挂起查询
2. 撤消用户的执行权限，并删除引用外部 UDF 代码模块的 SQL 函数和存储过程
3. 使用 **call sa_external_library_unload** 命令从 Sybase IQ 服务器卸载库（关闭 IQ 服务器也将自动卸载库）。
4. 对外部库文件执行所需的维护（复制、移动、更新、删除）。
5. 如果移动了库，请在注册脚本中编辑 SQL 函数和存储过程定义以反映外部库位置。
6. 授予用户执行权限，并运行注册脚本以重新创建引用外部 UDF 代码模块的 SQL 函数和存储过程。
7. 调用 SQL 函数或存储过程（参考外部 UDF 代码）以确保 Sybase IQ 服务器可以动态装载外部库。

授予和撤消权限

缺省情况下，用户定义的函数由创建者拥有， 尽管创建者可在创建时指定另一所有者。

所有者可用 **GRANT EXECUTE** 命令将权限授予其他用户。例如，函数的创建者 *fullname* 可以通过发出以下命令 允许 *another_user* 使用 *fullname*：

```
GRANT EXECUTE ON fullname TO another_user
```

或者可通过发出以下命令来撤消权限：

```
REVOKE EXECUTE ON fullname FROM another_user
```

请参见《系统管理指南：第一卷》>“管理用户 ID 和权限”>“管理单个用户 ID 和权限”>“在 Interactive SQL 中授予针对过程的权限”。

删除用户定义的函数

用户定义的函数一创建就会一直保留在数据库中， 直到有人特意将其删除为止。只有函数或过程的 所有者或具有 DBA 权限的用户，才可以将函数或过程从数据库中删除。

例如，要从数据库中删除标量或集合函数 *fullname* ，请输入：

```
DROP FUNCTION fullname
```

要从数据库中删除名为 *fullname* 的表 UDF 或 TPF， 请输入：

```
DROP PROCEDURE fullname
```

标量 UDF 和集合 UDF

标量和集合用户定义函数返回单值至调用环境。

注意： 标量和集合 UDF 为可许可选件，需要 IQ_UDF 或 IQ_IDA 许可证。安装许可证以启用用户定义函数。

您可使用多种配置安装 Sybase IQ。UDF 必须安装于此环境之中，而且必须能够运行于所有支持的配置环境。Sybase IQ 安装程序提供了默认的安装目录，也可允许用户选择不同的安装目录。当安装 UDF 库及相关的 SQL 函数定义脚本时，UDF 开发人员应该考虑提供同样的灵活性。

标量和集合 UDF 限制

外部的 C/C++ 标量和集合用户定义函数有一些限制。

- 所有 UDF 的写入方式允许不同用户在接收不同上下文函数的同时调用这些 UDF。
- UDF 访问全局或共享数据结构时，UDF 定义必须实现其对该数据的访问的相应锁定，包括所有普通代码路径和所有错误处理情况下的该锁定的释放。
- C++ 中实现的 UDF 可为其类提供过载的“新”运算符，但绝不能过载全局的“新”运算符。在某些平台上，这样做的影响不限于该特定库中定义的代码。
- 所有集合 UDF 以及所有确定型标量 UDF 的写入方式应当使相同输入值的接收始终能产生相同的输出值。情形并非如此的任何标量函数必须声明为 NONDETERMINISTIC，以避免潜在的不正确回应。
- 用户无需 DBA 授权便可创建标准 SQL 函数，但他们无法在没有 DBA 权限的情况下创建将调用外部库的函数。尝试执行此操作会产生一条错误消息“`You do not have permission to use the create function statement`”（您无权使用 create function 语句）。

创建标量或集合 UDF

了解如何创建并配置外部 C/C++ 标量与集合用户定义函数。

有关使用 Interactive SQL 创建 UDF 的说明，请参见《系统管理指南：第二卷》>“使用过程和批处理”。

1. 使用 **CREATE FUNCTION** 或 **CREATE AGGREGATE FUNCTION** 语句向服务器声明 UDF。将这些语句作为命令写入并执行，或者通过 Sybase Central 新建函数向导使用合适的 CREATE 语句。

CREATE FUNCTION 语句的外部 C/C++ 形式需要 DBA 或 RESOURCE 权限，因此标准用户无权声明任何此类型的 UDF。

2. 编写 UDF 库标识函数。 （第 15 页）
3. 将 UDF 定义为一组 C 或 C++ 函数。请参见“定义标量 UDF” （第 37 页）或“定义集合 UDF” （第 52 页）。
4. 用 C/C++ 实现函数入口点。
5. 编译 UDF 函数和库标识函数。 （第 19 页）
6. 将编译的文件链接到动态可链接库。

对 SQL 语句中的 UDF 的任何引用首先将链接动态可链接库（如果需要）。然后将调用“调用模式” （第 80 页）。

由于这些高性能的外部 C/C++ 用户定义函数涉及将非服务器库代码装载到服务器的进程空间中，因此，编写错误或恶意编写的函数会导致数据完整性、数据安全性和服务器的稳定性存在潜在风险。为管理此类风险，每台 Sybase IQ 服务器可以明确规定启用或禁用此功能 （第 25 页）。

用 SQL Anywhere 术语在 Sybase Central 中创建 UDF

Watcom-SQL 和 Transact-SQL 是 SQL Anywhere 支持的 SQL 方言，在创建用户定义的函数时可以使用。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. 选择**视图 > 文件夹**。
3. 在左侧窗格中，右键单击**过程和 函数**，然后选择**新建 > 函数**。
4. 为函数输入名称并选择将拥有该函数的用户。
5. 选择函数的 SQL 术语或语言。单击“下一步”。
6. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。
7. 键入返回值的名称，然后单击“下一步”。
8. 添加说明新函数用途的注释。单击**“完成”**。
9. 在右窗格中，单击“SQL”选项卡以填写过程代码。

声明和定义用户定义的标量函数

Sybase IQ 支持可在任何可以使用 SQRT 函数的位置使用的简单的用户定义的标量函数 (UDF)。

这些标量 UDF 可以是确定型函数（这意味着对于一组给定的参数值，函数始终返回相同的结果值），也可以是非确定型标量函数（这意味着相同的参数可以返回不同结果）。

注意： 本章中引用的标量 UDF 示例随 IQ 服务器一起安装，这些示例可作为 .cxx 文件存在于 `$IQDIR15/samples/udf` 下。您还可以在 `$IQDIR15/lib64/libudfex` 动态可链接库中找到这些示例。

声明标量 UDF

只有 DBA 或具有 DBA 权限的用户才可以声明进程中外部 UDF。还有一个服务器启动选项，允许管理员启用或禁用此样式的用户定义的函数。

编写并编译了 UDF 代码后，创建一个从相应库文件中调用 UDF 并将输入数据发送到 UDF 的 SQL 函数。

注意： 您还可以“在 Sybase Central 中创建用户定义的函数声明” （第 37 页）。

缺省情况下，所有用户定义的函数使用 UDF 的所有者的访问权限。

注意： 用户需要具有 DBA 授权才能声明 UDF 函数。

标量 UDF 创建语法是：

```
scalar-udf-declaration:
CREATE FUNCTION [ owner.]function-name
     ( [ parameter , ... ] )
RETURNS data-type
     [ routine-characteristics ... ]
EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

routine-characteristics:
     [NOT] DETERMINISTIC
  | { IGNORE | RESPECT } NULL VALUES
     | SQL SECURITY { INVOKER | DEFINER }
```

上述语法中的特性的缺省值为：

```
DETERMINISTIC
RESPECT NULL VALUES
SQL SECURITY DEFINER
```

为了最大限度地消除安全隐患，Sybase 建议您对 EXTERNAL NAME 子句的库名称部分的安全目录使用完全限定路径名。

SQL 安全性

定义是作为 INVOKER（调用函数的用户）还是作为 DEFINER（拥有函数的用户）执行函数。缺省为 DEFINER。

SQL SECURITY INVOKER 使用更多内存，这是因为调用过程的每个用户都需要注释。此外，将同时对用户名和 INVOKER 执行名称解析。使用所有对象名（表、过程等）的相应所有者限定这些名称。

外部名称

使用 **EXTERNAL NAME** 子句的函数是包含对外部库函数调用的包装。使用 **EXTERNAL NAME** 的函数在 **RETURNS** 子句后可以不使用其它子句。库名可包含文件扩展名，在

Windows 中通常为 `.dll`，在 UNIX 中通常为 `.so`。在没有扩展名的情况下，该软件附加平台特定的缺省库文件扩展名。

临时函数 不支持 **EXTERNAL NAME** 子句。请参见 SQL Anywhere Server - 编程 > SQL Anywhere 外部调用接口。

注意： 此参考指向 SQL Anywhere 文档。

启动服务器时可带有 库装载路径（其中有 UDF 库的位置）。在 UNIX 变体中，请修改 `start_iq startup` 脚本中的 LD_LIBRARY_PATH。对于所有 UNIX 变体，虽然 LD_LIBRARY_PATH 是通用路径，但最好将 SHLIB_PATH 用在 HP 中，将 LIB_PATH 用在 AIX 中。

在 UNIX 平台中，指定的外部名称可以含有 完全限定名，这种情况下不会使用 LD_LIBRARY_PATH。在 Windows 平台中，不能使用完全限定名， 库搜索路径由 PATH 环境变量指定。

注意： 可更新游标中不支持标量用户定义函数和用户定义集合函数。

另请参见

- Defining a Scalar UDF （第 37 页）

UDF Example: my_plus Declaration

“my_plus”示例是返回将函数的两个整数参数值相加所得结果的简单标量函数。

my_plus 声明

如果 my_plus 驻留在动态可链接库 my_shared_lib 中，此示例的声明将类似于：

```
CREATE FUNCTION my_plus (IN arg1 INT, IN arg2 INT)
     RETURNS INT
     DETERMINISTIC
     IGNORE NULL VALUES
     EXTERNAL NAME 'my_plus@libudfex'
```

此声明指出 my_plus 是一个简单标量 UDF，该函数驻留在具有名为 describe_my_plus 的描述符例程的 my_shared_lib 中。由于 UDF 的行为可能需要多个实际 C/C++ 入口点进行实现，因此，此组入口点未直接包含在 CREATE FUNCTION 语法中。其实，CREATE FUNCTION 语句的 EXTERNAL NAME 子句标识了此 UDF 的描述符函数。描述符函数在调用时将返回描述符结构，下一节中对此结构进行了详细定义。该描述符结构包含了体现此 UDF 实现的必要和可选函数指针。

此声明指出 my_plus 接受两个 INT 参数并返回 INT 结果值。如果使用不是 INT 的参数调用此函数，且该参数可以隐式转换为 INT，则在调用此函数之前将进行转换。如果使用无法隐式转换为 INT 的参数调用此函数，将生成转换错误。

而且，该声明还指出此函数是确定型函数。确定型函数在提供了相同的输入值的情况下始终返回相同的结果值。这意味着结果不依赖于除提供的参数值之外的任何外部信息，也不受以前调用所产生的任何副作用的影响。缺省情况下，假定函数是确定型函数，因此，如果忽略 CREATE 语句中的此特性，结果将相同。

上述声明的最后一部分是 IGNORE NULL VALUES 特性。如果任何输入参数为空值，则几乎所有内置标量函数都将返回空结果值。IGNORE NULL VALUES 指出 my_plus 函数遵守该约定，因此，如果其任一输入值为空值，则实际不会调用此 UDF 例程。由于 RESPECT NULL VALUES 是函数的缺省值，因此必须在此 UDF 的声明中指定此特性才能获取性能优势。在给定空输入值时可能返回非空结果的所有函数，必须使用缺省的 RESPECT NULL VALUES 特性。

在以下查询示例中，my_plus 显示在 SELECT 列表以及等效算术表达式中：

```
SELECT my_plus(t.x, t.y) AS x_plus_y_one, (t.x + t.y)AS x_plus_y_two
FROM t
WHERE t.z = 2
```

在以下示例中，以不同方式在同一查询的多个不同位置使用 my_plus：

```
SELECT my_plus(t.x, t.y), count(*)
FROM t
WHERE t.z = 2
AND my_plus(t.x, 5) > 10
AND my_plus(t.y, 5) > 10
GROUP BY my_plus(t.x, t.y)
```

UDF Example: my_plus_counter Declaration

示例 `my_plus_counter` 列举的是 一个简单的非确定性标量 UDF，可接收一个整数参数，并返回该参数值与 内部整数使用计数器相加的结果。如果输入参数值是 空值，则结果是使用计数器的当前值。

my_plus_counter 声明

假设 my_plus_counter 也位于动态链接库 my_shared_lib 内，此示例的声明如下：

```
CREATE FUNCTION my_plus_counter (IN arg1 INT DEFAULT 0)
     RETURNS INT
     NOT DETERMINISTIC
     RESPECT NULL VALUES
     EXTERNAL NAME 'describe_my_plus_counter@my_shared_lib'
```

RESPECT NULL VALUES 特性表示，即使输入参数的值为 NULL 也会调用该函数。这确实有必要，因为 my_plus_counter 的语义中包含如下含义：

- 内部保留一个使用计数，即使参数为 NULL 时该计数也会递增。
- 传递空值参数时也会生成非空值。

因为 RESPECT NULL VALUES 是缺省值，所以即使声明中省略该子句，结果也完全相同。

Sybase IQ 限制所有非确定性函数的使用。 只允许将其用在顶层查询块的 SELECT 列表， 或 UPDATE 语句的 SET 子句中。不能将其用在 子查询中，也不能将其用在 WHERE、ON、GROUP BY 或 HAVING 子句中。这种限制适用于非确定性 UDF 及 GETUID 和 NUMBER 之类的非确定性内置函数。

如上声明中需要说明的最后一点是输入参数上的 DEFAULT 限定符。该限定符向服务器指明，调用该函数时可以不带参数，届时服务器会针对缺少的参数自动以零作为其参数值。如果指定缺省值，则它必须可以隐式转换为该参数的数据类型。

在下面的示例中，第一个 SELECT 列表项将运行的计数器添加到各行的 t.x 的值上。第二个和第三个 SELECT 列表项返回的各行的值均与 NUMBER 函数的返回值相同。

```
SELECT my_plus_counter(t.x),
     my_plus_counter(0),
     my_plus_counter(),
     NUMBER()
FROM t
```

UDF 示例：my_byte_length 声明

my_byte_length 是一个简单的标量用户定义函数，它以字节为单位返回列的大小。

my_byte_length 声明

如果 **my_byte_length** 位于动态可链接库 my_shared_lib 中，则本示例的声明为：

```
CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//             RETURNS UNSIGNED INT
//             DETERMINISTIC
//           IGNORE NULL VALUES
//             EXTERNAL NAME 'my_byte_length@libudfex'
```

此声明表示 **my_byte_length** 是一个驻留于 my_shared_lib 中的简单的标量 UDF，它有一个名为 describe_my_byte_length 的描述符例程。由于 UDF 的行为实现 可能需要多个实际的 C/C++ 入口点 ，所以入口点组并非 **CREATE FUNCTION** 语法的直接组成部分。而是使用 **CREATE FUNCTION** 语句的 **EXTERNAL NAME** 子句来确定 UDF 的描述符函数 。当调用描述符函数时，将返回描述符 结构。该描述符 结构包含所需的及可选的函数指针 （体现该 UDF 的实现）。

本声明还提到说：**my_byte_length** 接受一个 `LONG BINARY` 参数并返回 `UNSIGNED INT` 结果值。

注意： 大对象数据支持需要使用单独许可的 Sybase IQ 选项。

声明规定此函数具有确定性，当提供的输入值相同时 ，其所返回的结果值也始终相同。这意味着，其结果值不 取决于所提供的参数值以外的任何外部信息， 也不受此前调用的影响。缺省情况下， 假定函数具有确定性，因此，如果在 **CREATE** 语句中省略此特性，结果集将是 相同的。

本声明的最后一部分是 IGNORE NULL VALUES 特性。一旦有任何输入参数为空，几乎所有的内置标量函数都返回 NULL 结果值。IGNORE NULL VALUES 描述如下：**my_byte_length** 函数遵循该约定，因此当其中任意一个输入值为空时，此 UDF 例程实际上不会被调用。由于 RESPECT NULL VALUES 为函数的默认值，因此该特性必须在此 UDF 声明中指定，以便获得性能优势。在输入值为空的情况下可能返回非空结果的所有函数必须使用默认的 RESPECT NULL VALUES 特性。

my_byte_length 在 **SELECT** 列表中的本示例查询会为 `exTable` 中的每一行返回包含一行一列的表格，其中 `INT` 表示二进制文件的大小：

```
SELECT my_byte_length(exLOBColumn)
FROM exTable
```

在 Sybase Central 中声明标量用户定义函数

Sybase IQ 支持简单标量 UDF，后者在可使用 SQRT 函数的任何地方都可以使用。这些标量 UDF 可以是确定性 UDF，这意味着对于一组给定的参数值，函数 始终返回相同的结果值。Sybase IQ 也支持非确定性 标量函数，这意味着同样的参数会返回 不同的结果。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. 在左窗格中，右键单击“过程和函数”，然后选择“新建”>“函数”。
3. 在“欢迎”(Welcome) 对话框中，键入函数名称并选择将成为该函数所有者的用户。
4. 若要创建用户定义的函数，请选择“外部 C/C++”。单击“下一步”。
5. 在“外部函数属性”对话框中，选择“标量”。
6. 键入动态可链接库文件的名称（省略 .so 或 .dll 扩展名）。
7. 键入描述符函数的名称。单击“下一步”。
8. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。单击“下一步”。
9. 选择函数是否为确定型函数。
10. 指定函数是使用空值还是忽略空值。
11. 选择用于运行函数的特权是来自定义用户（定义者）还是调用用户（调用者）。
12. 添加说明新函数用途的注释。单击“完成”。
13. 在右窗格中，单击“SQL”选项卡以填写过程代码。

Defining a Scalar UDF

用于定义标量用户定义的函数的 C/C++ 代码包含 4 个必需项。

- **extfnapiv3.h** – 包含 UDF 接口定义头文件。
- **_evaluate_extfn** — 求值函数。所有求值函数都有两个参数：
  - 标量 UDF 上下文结构的一个实例，实例在每次使用 UDF 时都是唯一的，其中包含一组回调函数指针以及一个 UDF 可用于存储 UDF 专用数据的指针。
  - 指向数据结构的指针，允许访问参数值以及通过所提供回调生成的结果值。
- **a_v3_extfn_scalar** – 标量 UDF 描述符结构的一个实例，其中包含指向求值函数的一个指针。
- **Descriptor function** – 返回指向标量 UDF 描述符结构的一个指针。

这些部分是可选的：

- **_start_extfn** – 一个初始化函数，每次使用 SQL 时通常调用一次。如果提供此项，您还必须在标量 UDF 描述符结构中包含一个指向该函数的指针。所有初始化函数均使用一个参数，即一个指向标量 UDF 上下文结构的指针，每次使用 UDF 时该指针都是唯一的。所传递的上下文结构与向求值例程中传递的上下文结构相同。
- **_finish_extfn** – 一个关闭函数，每次使用 SQL 时通常调用一次。如果提供此项，您还必须在标量 UDF 描述符结构中包含一个指向该函数的指针。所有关闭函数均使用一个参数，即一个指向标量 UDF 上下文结构的指针，每次使用 UDF 时该指针都是唯一的。所传递的上下文结构与向求值例程中传递的上下文结构相同。

另请参见

- 声明标量 UDF （第 33 页）

Scalar UDF Descriptor Structure

标量 UDF 描述符结构 a_v3_extfn_scalar 定义如下：

```
typedef struct a_v3_extfn_scalar {    //
    // Metadata descriptor for a scalar UDF
    // supplied by the UDF library to the server
    // An optional pointer to an initialize function
    void (*_start_extfn)(a_v3_extfn_scalar_context * cntxt);
    //
    // An optional pointer to a shutdown function
    void (*_finish_extfn)(a_v3_extfn_scalar_context *    cntxt);
    //
    // A required pointer to a function that will be
    // called for each invocation of the UDF on a
    // new set of argument values
    void (*_evaluate_extfn)(a_v3_extfn_scalar_context * cntxt, void
*args_handle);
    // RESERVED FIELDS MUST BE INITIALIZED TO NULL
 void  *reserved1_must_be_null;
 void  *reserved2_must_be_null;
 void  *reserved3_must_be_null;
 void  *reserved4_must_be_null;
 void  *reserved5_must_be_null;
                    …

} a_v3_extfn_scalar;
```

对于每个定义的标量 UDF，始终应该有一个 **a_v3_extfn_scalar** 实例。如果没有提供可选初始化函数，则描述符结构中的对应值应该为空指针。同理，如果没有提供关闭函数，则描述符结构中的对应值应为空指针。

在调用任何求值例程之前至少要调用一次初始化函数，而在最后一次调用求值函数之后至少要调用一次关闭函数。初始化函数和关闭函数通常在每次使用时只调用一次。

Scalar UDF Context Structure

传递至标量 UDF 描述符结构内指定的每个函数的标量 UDF 上下文结构 **a_v3_extfn_scalar_context** 定义如下：

```
typedef struct a_v3_extfn_scalar_context {
//-------- Callbacks available via the context --------
```

```
//
  short (SQL_CALLBACK *get_value)(
    void           *arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value
              );
 short (SQL_CALLBACK *get_piece)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value,
              a_sql_uint32    offset
              );
 short (SQL_CALLBACK *get_value_is_constant)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              a_sql_uint32 *  value_is_constant
              );
 short (SQL_CALLBACK *set_value)(
              void *          arg_handle,
              an_extfn_value *value,
              short              append
              );
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)(
              a_v3_extfn_scalar_context * cntxt
              );
 short (SQL_CALLBACK *set_error)(
              a_v3_extfn_scalar_context * cntxt,
              a_sql_uint32    error_number,
              const char *    error_desc_string
              );
 void (SQL_CALLBACK *log_message)(
              const char *msg,
              short msg_length
                 );
 short (SQL_CALLBACK *convert_value)(
                 an_extfn_value   *input,
                 an_extfn_value   *output
 //---------- Data available from the context ----------
 void * _user_data;   // read-write field
 //---------- For Server Internal Use Only -------------
 void  * _for_server_internal_use;
} a_v3_extfn_scalar_context;
```

注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

标量 UDF 上下文结构中的 _user_data 字段可用 UDF 所需的数据填充。通常会填入由 _start_extfn 函数 分配的堆结构，然后由 _finish_extfn 函数释放。

标量 UDF 上下文结构的其余部分将被填入引擎提供一组回调函数，以便在每个用户的 UDF 函数内使用。这些回调函数多数会通过一个短结果值返回成功状态；返回 true 则表示成功。编写良好的 UDF 实现不会导致故障状态，但在开发期间（也可能在给定

UDF 库的所有内部调试中），Sybase 建议您检查回调返回的状态值。故障可能源自 UDF 实现内的编码错误，例如要求提供的参数数量超过 UDF 定义使用的参数数量。

多数回调使用的一般参数集包括：

- **arg_handle** — 可由所有形式的求值方法接收的一个指针，通过该指针提供传递给 UDF 的输入参数的值，还可以通过该指针设置 UDF 结果值。
- **arg_num** — 表示将被访问的输入参数的一个整数。输入参数将从 1 开始，按从左到右的升序顺序编号。
- **cntxt** — 指向服务器传递给所有 UDF 入口点的上下文结构的指针。
- **value** — 指向 an_extfn_value 结构的一个实例的指针，用于从服务器获得输入参数值，或用于设置函数的结果值。an_extfn_value 结构的格式如下：

```
typedef struct an_extfn_value {
void * data;
a_SQL_uint32 piece_len;
union {
        a_SQL_uint32 total_len;
        a_SQL_uint32 remain_len;
} len;
a_SQL_data_type type;
} an_extfn_value;
```

表 1. 外部标量函数上下文：a_v3_extfn_scalar_context

a_v3_extfn_scalar_context 方法的结构	描述
void set_cannot_be_distributed(a_v3_extfn_scalar_context * cntxt)	即使已达到库级别的分配标准， 也可在 UDF 级别禁止分配。缺省情况下，假定库可分配时 UDF 也可分配。 UDF 负责作出 禁止对服务器进行分配的决策。

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

示例：my_plus 定义

my_plus 标量 UDF 的定义示例。

my_plus 定义

由于此 UDF 不需要初始化函数或关闭函数，因此描述符结构内的对应值将被设置为 0。描述符函数名称与声明中使用的 EXTERNAL NAME 相匹配。求值方法不检查参数的数据类型，因为它们被声明为 INT 类型。

```
#include "extfnapiv3.h"
#include <stdlib.h>


//  A simple deterministic scalar UDF that just adds
//  two integer arguments and then returns the result.
```

```
//
//  Corresponding SQL declaration:
//
//    CREATE FUNCTION my_plus(IN arg1 INT, IN arg2 INT)
//                                            RETURNS INT
//                                            DETERMINISTIC
//                                    IGNORE NULL VALUES
//                                            EXTERNAL NAME
'my_plus@libudfex'
//
#if defined __cplusplus
extern "C" {
#endif

static void my_plus_evaluate(a_v3_extfn_scalar_context *cntxt,
                      void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, arg2, result;

  //  Get first argument
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg1 = *((a_sql_int32 *)arg.data);

  //  Get second argument
  (void) cntxt->get_value( arg_handle, 2, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg2 = *((a_sql_int32 *)arg.data);

  //  Set the result value
  outval.type = DT_INT;
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + arg2;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_descriptor = {
    0,
    0,
    &my_plus_evaluate,
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
        NULL                    // _for_server_internal_use
```

```
};


a_v3_extfn_scalar *my_plus()
{
  return &my_plus_descriptor;
}

#if defined __cplusplus
}
#endif
```

示例：my_plus _counter 定义

此示例标量 UDF 可检查参数值指针数据，以确定 输入参数值是否为空。该 UDF 也有初始化函数和 关闭函数，二者均容许多次调用。

my_plus _counter 定义

```
#include "extfnapiv3.h"
#include <stdlib.h>

//  A simple non-deterministic scalar UDF that adds
//  an internal integer usage counter to its integer
//  argument and then returns the resulting integer.
//
//  Here, the start function creates a little structure for
//  the counter, and then the finish function deallocates it.
//
//  Corresponding SQL declaration:
//
//          CREATE FUNCTION plus_counter(IN arg1 INT)
//                      RETURNS INT
//                      NOT DETERMINISTIC
//                      RESPECT NULL VALUES
//                      EXTERNAL NAME 'my_plus_counter@libudfex'


typedef struct my_counter {
  a_sql_int32 _counter;
} my_counter;

#if defined __cplusplus
extern "C" {
#endif

static void my_plus_counter_start(a_v3_extfn_scalar_context *cntxt)
{
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  //  If we have not already allocated the
  //  counter structure, then do so now
  if (!cptr) {
    cptr = (my_counter *)malloc(sizeof(my_counter));
    cntxt->_user_data = cptr;
  }
  cptr->_counter = 0;
}
```

```
static void my_plus_counter_finish(a_v3_extfn_scalar_context *cntxt)
{
  //  If we still have an allocated the
  //  counter structure, then free it now
  if (cntxt->_user_data) {
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_plus_counter_evaluate(a_v3_extfn_scalar_context
*cntxt,
                                     void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, result;

  //  Increment the usage counter
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  cptr->_counter += 1;

  //  Get the one argument
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (!arg.data) {
    // argument value was NULL;
    arg1 = 0;
  } else {
    arg1 = *((a_sql_int32 *)arg.data);
  }

  outval.type = DT_INT;
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + cptr->_counter;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_counter_descriptor =
    {   &my_plus_counter_start,
        &my_plus_counter_finish,
        &my_plus_counter_evaluate,
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // _for_server_internal_use
};


a_v3_extfn_scalar *my_plus_counter()
```

```
{
  return &my_plus_counter_descriptor;
}

#if defined __cplusplus
}
#endif
```

示例：my_byte_length Definition

标量 UDF 示例 **my_byte_length** 通过逐段流式传输数据来测量列的大小，然后以字节为单位返回列的大小。

my_byte_length definition

注意： 大对象数据支持需要使用单独许可的 Sybase IQ 选项。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>


//  A simple function that returns the size of a cell value in bytes
//
//    CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//            RETURNS UNSIGNED INT
//            DETERMINISTIC
//          IGNORE NULL VALUES
//            EXTERNAL NAME 'my_byte_length@libudfex'

#if defined __cplusplus
extern "C" {
#endif

static void my_byte_length_evaluate(a_v3_extfn_scalar_context
*cntxt,
                      void *arg_handle)
{

    if (cntxt == NULL || arg_handle == NULL)
    {
        return;
    }

    an_extfn_value  arg;
    an_extfn_value  outval;

    a_sql_uint64 total_len;

    //  Get first argument
    a_sql_uint32 fetchedLength = 0;
    (void) cntxt->get_value( arg_handle, 1, &arg );
    if (arg.data == NULL)
    {
```

```
        return;
    }

    fetchedLength += arg.piece_len;

    // saving total length as it loses scope inside get_piece
    total_len = arg.len.total_len;

    while (fetchedLength < total_len)
    {
       (void) cntxt->get_piece( arg_handle, 1, &arg, fetchedLength );
        fetchedLength += arg.piece_len;
    }

   //if this fails, the function did not get the full data from the
cell
    assert(fetchedLength == total_len);

    outval.type = DT_UNSINT;
    outval.piece_len = 4;
    outval.data = &fetchedLength;
    cntxt->set_value(arg_handle, &outval, 0);
}

static a_v3_extfn_scalar my_byte_length_descriptor = {
    0,
    0,
    &my_byte_length_evaluate,
    0,          // Reserved - initialize to NULL
    0,          // Reserved - initialize to NULL
    0,          // Reserved - initialize to NULL
    0,          // Reserved - initialize to NULL
    0,          // Reserved - initialize to NULL
        NULL                    // _for_server_internal_use
};

a_v3_extfn_scalar *my_byte_length()
{
  return &my_byte_length_descriptor;
}

#if defined __cplusplus
}
#endif
```

另请参见

- 示例：my_byte_length Definition （第 44 页）

声明和定义集合 UDF

Sybase IQ 支持集合 UDF。例如，SUM 函数就是一个内置集合函数。 简单的集合函数可从一组参数值生成 单一结果值。可以编写集合 UDF， 在可使用 SUM 集合的任何地方都可以使用。

注意： 此处引用的集合 UDF 示例 都已随服务器一同安装，可在 `$IQDIR15/samples/udf` 中以 .cxx 文件的形式找到。 也可在 `$IQDIR15/lib64/libudfex` 动态链接 库中查找这些示例。

集合函数可生成单一结果，也可生成一组结果。输出结果集中数据点的数量，可能未必与输入集中数据点的数量相符。多输出集合 UDF 必须用临时输出文件存储结果。

声明集合 UDF

与标量 UDF 相比，集合 UDF 的功能更强大，创建也更复杂。

编写并编译了 UDF 代码后，创建一个从相应库文件中调用 UDF 并将输入数据发送到 UDF 的 SQL 函数。

注意： 您还可以“在 Sybase Central 中创建用户定义的函数声明” （第 37 页）。

如果要实现集合 UDF，则必须决定：

- 它是否将仅在整个数据集或分区上用作联机分析处理 (OLAP) 样式集合，例如 RANK。
- 它是否将用作简单集合或 OLAP 样式集合，例如 SUM。
- 它是否仅用作整个组上的简单集合。

集合 UDF 的声明和定义可反映出这些使用决策。

用户定义的集合函数的创建语法是：

```
aggregate-udf-declaration:
    CREATE AGGREGATE FUNCTION [ owner.]function-name
          ( [ parameter , ... ] )
    RETURNS data-type
        [ aggregate-routine-characteristics ... ]
    EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

aggregate-routine-characteristics:
      DUPLICATE { SENSITIVE | INSENSITIVE }
        -- is the server allowed to eliminate DISTINCT
    | SQL SECURITY {INVOKER | DEFINER}
    | OVER restrict
    | ORDER order-restrict
        | WINDOW FRAME
        { { ALLOWED | REQUIRED }
            [ window-frame-constraints ... ]
          | NOT ALLOWED }
```

```
    | ON EMPTY INPUT RETURNS { NULL | VALUE }
  -- Call or skip function on NULL inputs

window-frame-constraints:
      VALUES { [ NOT ] ALLOWED }
    | CURRENT ROW { REQUIRED | ALLOWED }
    | [ UNBOUNDED ] { PRECEDING | FOLLOWING } restrict

restrict:   { [ NOT ] ALLOWED } | REQUIRED

order-restrict:
{ NOT ALLOWED | SENSITIVE | INSENSITIVE | REQUIRED
```

返回数据类型、参数、数据类型和缺省值的处理都与标量 UDF 定义中的处理相同。

如果可将集合 UDF 用作简单集合，则可能能够将其与 DISTINCT 限定符一同使用。集合 UDF 声明中的 DUPLICATE 子句 决定：

- 因为结果对于重复值敏感而考虑消除重复值， 然后再调用集合 UDF （例如对于内置“COUNT(DISTINCT T.A)”），还是
- 结果是否对重复条目不敏感（例如对于“MAX(DISTINCT T.A)”）。

优化程序可通过 DUPLICATE INSENSITIVE 选项考虑在不影响结果的情况下消除 重复值，从而使优化程序可以 选择如何执行查询。写入集合 UDF 预计会出现重复值。如果必须消除重复值，则服务器会先执行操作，然后再开始调用 _next_value_extfn 集。

通过大多不属于标量 UDF 语法的其余子句，均可指定这种函数的使用方法。缺省情况下， 会假定既能将集合 UDF 用作简单集合， 也能将其用作 OLAP 样式集合，窗口构架种类不限。

对于要专门用作简单集合函数的集合 UDF，请用以下代码对其进行声明：

```
OVER NOT ALLOWED
```

任何随后将该集合用作 OLAP 样式集合的尝试都将生成错误。

对于允许或需要使用 OVER 子句的集合 UDF，UDF 定义者 可通过先后指定“ORDER”和限制条件类型， 对 ORDER BY 子句是否可出现在 OVER 子句中指定限制条件。窗口排序限制条件类型：

- **REQUIRED** — 必须指定 ORDER BY，不可将其删除。
- **SENSITIVE** — 指定或不指定 ORDER BY 均可，但一旦指定则不可将其删除。
- **INSENSITIVE** — 指定或不指定 ORDER BY 均可，但服务器可删除排序以提高效率。
- **NOT ALLOWED** — 不可指定 ORDER BY。

只有以针对整个集的 OLAP 样式集合的形式， 或已排序的分区的形式，如 内置 RANK，才能通过以下代码声明有意义的集合 UDF：

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME NOT ALLOWED
```

只有以采用缺省窗口构架 UNBOUNDED PRECEDING 到 CURRENT ROW 的 OLAP 样式集合的形式，才能通过以下代码声明有意义的集合 UDF:

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME ALLOWED
    RANGE NOT ALLOWED
    UNBOUNDED PRECEDING REQUIRED
    CURRENT ROW REQUIRED
    FOLLOWING NOT ALLOWED
```

所有各种选项和限制设置的缺省值如下:

```
DUPLICATE SENSITIVE
SQL SECURITY DEFINER
OVER ALLOWED
ORDER SENSITIVE
WINDOW FRAME ALLOWED
CURRENT ROW ALLOWED
PRECEDING ALLOWED
UNBOUNDED PRECEDING ALLOWED
FOLLOWING ALLOWED
UNBOUNDED FOLLOWING ALLOWED
```

SQL 安全性

定义是作为 INVOKER（调用函数的用户）还是作为 DEFINER（拥有函数的用户）执行函数。缺省为 DEFINER。

如果指定了 **SQL SECURITY INVOKER**，将使用更多内存，这是因为调用过程的每个用户都需要注释。此外，如果指定了 **SQL SECURITY INVOKER**，将同时对用户名和 INVOKER 执行名称解析。使用所有对象名（表、过程等）的相应所有者限定这些名称。

外部名称

使用 **EXTERNAL NAME** 子句的函数是包含对外部库函数调用的包装。使用 **EXTERNAL NAME** 的函数在 **RETURNS** 子句后可以不使用其它子句。库名可包含文件扩展名，在 Windows 中通常为 `.dll`，在 UNIX 中通常为 `.so`。在没有扩展名的情况下，该软件附加平台特定的缺省库文件扩展名。

临时函数 不支持 **EXTERNAL NAME** 子句。请参见 SQL Anywhere Server - 编程 > SQL Anywhere 外部调用接口。

注意： 此参考指向 SQL Anywhere 文档。

启动服务器时可带有 库装载路径（其中有 UDF 库的位置）。在 UNIX 变体中，通过修改 start_iq 启动脚本中的 LD_LIBRARY_PATH，可添加库装载路径。对于所有 UNIX 变体，虽然 LD_LIBRARY_PATH 是通用路径，但最好将 SHLIB_PATH 用在 HP 中，将 LIB_PATH 用在 AIX 中。

在 UNIX 平台中，指定的外部名称可以含有 完全限定名，这种情况下不会使用 LD_LIBRARY_PATH。在 Windows 平台中，不能使用完全限定名， 库搜索路径由 PATH 环境变量指定。

注意： 可更新游标中不支持标量用户定义函数和用户定义集合函数。

另请参见

- 定义集合 UDF （第 52 页）
- 集合用户定义函数的上下文存储 （第 79 页）

示例：my_sum 声明

示例“my_sum”类似于 内置 SUM，除了仅可对整数执行运算。

my_sum 声明

既然 my_sum 像 SUM 一样可以在任意上下文内使用，它的声明相对比较简单：

```
CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
    RETURNS BIGINT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_integer_sum@my_shared_lib'
```

各种使用限制均缺省为 ALLOWED，指定该函数可在 SQL 语句内允许使用集合函数的任意位置上使用。

没有任何使用限制时，my_sum 可用作整个行集上的简单集合，如下所示：

```
SELECT MIN(t.x), COUNT (*), my_sum(t.y)
FROM t
```

没有任何使用限制时，my_sum 也可用作针对 GROUP BY 子句所指定的每个组进行计算的简单集合：

```
SELECT t.x, COUNT(*), my_sum(t.y)
FROM t
GROUP BY t.x
```

由于没有使用限制，my_sum 可用作带有 OVER 子句的 OLAP 样式集合，如下累计求和示例所示：

```
SELECT t.x,
    my_sum(t.x)
        OVER (ORDER BY t.x ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
        AS cumulative_x,
    COUNT(*)
FROM t
GROUP BY t.x
ORDER BY t.x
```

示例：my_bit_xor 声明

“my_bit_xor”示例类似于 SQL Anywhere (SA) 的内置 BIT_XOR，只是它仅仅执行无符号整数运算。

my_bit_xor 声明

形成的声明为：

```
CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED    INT)
 RETURNS UNSIGNED INT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_bit_xor@my_shared_lib'
```

像 my_sum 示例一样，my_bit_xor 没有关联任何使用限制，因此可用作简单集合或具有任意类型窗口的 OLAP 样式集合。

示例：my_bit_or 声明

示例“my_bit_or” 类似于 SA 内置 BIT_OR，但仅可对无符号整数执行运算， 并且仅可用作简单集合。

my_bit_or 声明

形成的声明类似如下：

```
CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
     RETURNS UNSIGNED INT
     ON EMPTY INPUT RETURNS NULL
     OVER NOT ALLOWED
     EXTERNAL NAME 'describe_my_bit_or@ my_shared_lib'
```

与 my_bit_xor 示例不同，声明中的 OVER NOT ALLOWED 短语限制该函数用于简单集合。由于该使用限制，my_bit_or 只能用作整个行集上的简单集合，或用作针对 GROUP BY 子句所指定的每个组进行计算的简单集合，如下示例所示：

```
SELECT t.x, COUNT(*), my_bit_or(t.y)
FROM t
GROUP BY t.x
```

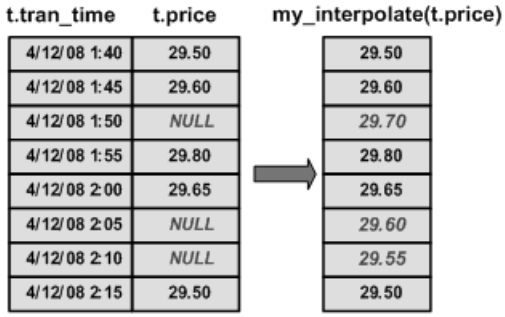
示例：my_interpolate 声明

“my_interpolate”示例是 OLAP 样式的 UDAF，它跨任意一组与两个方向上最近非空值相邻的空值执行线性内插操作，尝试在序列中缺少值的位置中填入值（其中由空值表示缺少值）。

my_interpolate 声明

如果给定行上的输入不是 NULL，则该行的结果与输入值相同。

图 1： my_interpolate 结果



为合理控制运算成本，运行 my_interpolate 时必须使用固定宽度行式窗口，但用户可以根据他/她期望看到的相邻 NULL 值的最大数目设置窗口宽度。该函数输入一组双精度浮点值，通过运算生成一组双精度浮点值。

形成的 UDAF 声明类似如下：

```
CREATE AGGREGATE FUNCTION my_interpolate (IN arg1 DOUBLE)
RETURNS DOUBLE
    OVER REQUIRED
    WINDOW FRAME REQUIRED
        RANGE NOT ALLOWED
        PRECEDING REQUIRED
        UNBOUNDED PRECEDING NOT ALLOWED
        FOLLOWING REQUIRED
        UNBOUNDED FOLLOWING NOT ALLOWED
    EXTERNAL NAME 'describe_my_interpolate@my_shared_lib'
```

OVER REQUIRED 表示该函数不能用作简单集合（即使使用 ON EMPTY INPUT 也没有意义）。

WINDOW FRAME 详细信息指定使用此函数时必须使用固定宽度行式窗口，该窗口可从当前行向前和向后两个方向延伸。由于这些使用限制，my_interpolate 用作带有 OVER 子句的 OLAP 样式集合，类似于：

```
SELECT t.x,
    my_interpolate(t.x)
    OVER (ORDER BY t.x ROWS BETWEEN 5 PRECEDING AND 5 FOLLOWING)
        AS x_with_gaps_filled,
    COUNT(*)
FROM t
GROUP BY t.x
ORDER BY t.x
```

在 my_interpolate 的 OVER 子句中，前面的行和后面的行的精确数值可以变化，您可以选择使用 PARTITION BY 子句；否则，这些行一定会与在声明中给定使用限制的以上示例类似。

在 Sybase Central 中声明集合 UDF

Sybase IQ 支持集合 UDF。 例如，SUM 函数就是一个内置集合函数。 简单的集合函数可采用一组参数值，然后从这组输入生成 单一结果值。可以编写用户定义的集合函数， 在可使用 SUM 集合的任何地方 都可以使用。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. 在左窗格中，右键单击“过程和函数”，然后选择“新建”>“函数”。
3. 在“欢迎”(Welcome) 对话框中，键入函数名称并选择将成为该函数所有者的用户。
4. 若要创建用户定义的函数，请选择“外部 C/C++”。单击“下一步”。
5. 选择“集合”。
6. 键入动态可链接库文件的名称（省略 .so 或 .dll 扩展名）。
7. 键入描述符函数的名称。单击“下一步”。
8. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。单击“下一步”。
9. 选择用于运行函数的特权是来自定义用户（定义者）还是调用用户（调用者）。
10. 指定是允许在 **OVER** 子句中使用函数，需要在该子句中使用函数，还是不允许在该子句中使用函数。单击“下一步”。

    如果不允许在 **OVER** 子句中使用函数，请继续执行步骤 14。
11. 指定当函数用于定义窗口时，是否需要使用 **ORDER BY** 子句。单击“下一步”。
12. 指定是允许在 **WINDOW FRAME** 子句中使用函数，需要在 **WINDOW FRAME** 子句中使用函数，还是不允许在 **WINDOW FRAME** 子句中使用函数。单击“下一步”。

    如果不允许在 **WINDOW FRAME** 子句中使用函数，请跳到步骤 14。
13. 标识对 **WINDOW FRAME** 子句的约束。单击“下一步”。
14. 指定是否需要在调用此函数前通过数据库服务器将重复输入值滤出。
15. 标识在不带任何数据调用函数时函数返回值是 **NULL**，还是固定值。单击“下一步”。
16. 添加说明新函数用途的注释。单击“完成”。
17. 在右窗格中，单击“SQL”选项卡以填写过程代码。

    新函数将显示在“过程和函数”中。

定义集合 UDF

用于定义用户定义集合函数的 C/C++ 代码包含 8 个必需项。

- **extfnapiv3.h** — 是 UDF 接口定义头文件。对于第 4 版 API，该文件是 `extfnapiv4.h`。
- **_start_extfn** — 是每使用一次 SQL 都要调用一次的初始化函数。所有初始化 函数都接收一个参数：指向集合 UDF 上下文结构的指针 （每次使用集合 UDF 指针都

不相同）。所传递的上下文结构，与向所有为该次使用而提供的函数传递的上下文结构相同。

- **_finish_extfn** — 是每使用一次 SQL 都要调用一次的关闭函数。所有关闭 函数都接收一个参数：指向集合 UDF 上下文结构的指针 （每次使用集合 UDF 指针都不相同）。
- **_reset_extfn** — 是一个重置函数，每当创建新组、新分区时都会调用一次， 如有必要，也会在每当开始移动窗口时调用。所有重置 函数都接收一个参数：指向集合 UDF 上下文结构的指针 （每次使用集合 UDF 指针都不相同）。
- **_next_value_extfn** – 针对每组新输入参数调用的函数。_next_value_extfn 使用两个参数：
  - 指向集合 UDF 上下文的指针，以及
  - 一个 args_handle。

  与在标量 UDF 中一样，arg_handle 可与所提供的回调函数指针一起使用以访问实际参数值。
- **_evaluate_extfn** – 与标量 UDF 求值函数类似的一个求值函数。所有求值函数都有两个参数：
  - 指向集合 UDF 上下文结构的指针，以及
  - 一个 args_handle。
- **a_v3_extfn_aggregate** – 集合 UDF 描述符结构的一个实例，包含指向该 UDF 所有已提供函数的指针。
- **Descriptor function** – 一个描述符函数，返回指向该集合 UDF 描述符结构的一个指针。

除必需项外，还有若干可选项，支持针对特定使用情况实现更加优化的访问：

- **_drop_value_extfn** – 针对超出移动窗口构架的每个输入参数值集调用的可选函数指针。该函数不应设置集合结果。使用 get_value callback 函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数。

  在下列情况下，可将函数指针设置为空指针：
  - 集合不能与窗口构架一起使用，
  - 集合在某种程度上不可逆，或
  - 用户对最佳性能没有兴趣。

  如果未提供 _drop_value_extfn，且用户已经指定移动窗口，则可在每次窗口构架移动时调用重置函数，并通过调用 next_value 函数包含该窗口内的各行，最后调用求值函数。

  如果已提供 _drop_value_extfn，则在每次窗口构架移动时，针对超出窗口构架的各行调用该删除值函数，然后针对刚刚添加到窗口构架中的各行调用 next_value 函数，最后调用求值函数以生成集合结果。
- **_evaluate_cumulative_extfn** – 针对每组新输入参数值调用的可选函数指针。如果已提供该函数，且在跨越从 UNBOUNDED PRECEDING 到 CURRENT ROW 的基于行的窗口构架内使用，则可调用此函数，而无需调用“下一个值”函数以及紧邻其后调用的求值函数。

_evaluate_cumulative_extfn 必须通过 set_value 回调设置集合结果。可通过常用的 get_value 回调函数访问其输入参数值集合。在下列情况下，该函数指针应设置为空指针：

- 绝不会以此方式使用该集合，或
- 用户不担心最佳性能。

- **_next_subaggregate_extfn** – 可与 _evaluate_superaggregate_extfn 一起使用的可选回调函数指针，支持通过并行运行来优化该集合的某些用法。

  某些集合用作简单集合（换言之即带有 OVER 子句的非 OLAP 样式集合）时，可以通过首先生成一组中间集合结果来分区，其中每个中间结果均为来自一个不连接的输入行子集的计算结果。

  此类可分区集合的示例包括：

  - SUM，其中，可通过针对每个不连接的输入行子集执行 SUM，然后在 sub-SUM 上执行 SUM，由此计算得出最终 SUM；以及
  - COUNT(*)，其中，可通过针对每个不连接的输入行子集执行 COUNT，然后在每个分区的 COUNT 上执行 SUM，由此计算得出最终 COUNT。

  当集合满足上述条件时，服务器可以选择并行执行该集合的计算。对于集合 UDF，仅当已提供 _next_subaggregate_extfn 函数指针和 _evaluate_superaggregate_extfn 指针时，才能应用该并行优化。

  _reset_extfn 函数不设置集合最终结果，而且按照定义，它将使用与集合 UDF 的定义返回值完全相同的数据类型的唯一输入参数值。

  通过正常的 get_value 回调函数访问子集合输入值。子集合与超级集合之间不可进行直接通信；所有此类通信将由服务器处理。子集合和超级集合不共用上下文结构。而是将各个子集合完全视为非分区集合。独立超级集合将看到类似如下的调用模式：

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

  或类似如下：

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

  如果既未提供 _evaluate_superaggregate_extfn，也未提供 _next_subaggregate_extfn，则集合 UDF 受限，不许在含有 GROUP BY CUBE 或 GROUP BY ROLLUP 的查询块中用作简单集合。

- **_evaluate_superaggregate_extfn** – 可与 _next_subaggregate_extfn 一起使用的可选回调函数指针，支持通过并行来优化用作简单集合时的某些用法。调用 _evaluate_superaggregate_extfn 返回分区集合的结果。通过使用 a_v3_extfn_aggregate_context 结构中的正常 set_value 回调函数将结果值发送至服务器。

另请参见

- 声明集合 UDF （第 46 页）
- 集合用户定义函数的上下文存储 （第 79 页）
- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

集合 UDF 描述符结构

集合 UDF 描述符结构由多个部分组成。

- **typedef struct a_v3_extfn_aggregate** – 库提供的集合 UDF 函数的元数据描述符。
- **_start_extfn** – 初始化函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。通常用于分配某些结构并将其地址存储在 a_v3_extfn_aggregate_context 内的 _user_data 字段中。每 a_v3_extfn_aggregate_context 只能调用一次 _start_extfn。

  ```
  void (*_start_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_finish_extfn** – 关闭函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。通常用于释放地址存储在 a_v3_extfn_aggregate_context 中的 _user_data 字段内的某些结构。每 a_v3_extfn_aggregate_context 只能调用一次 _finish_extfn。

  ```
  void (*_finish_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_reset_extfn** – “启动新组”函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。通常用于重置其地址存放在 a_v3_extfn_aggregate_context 中的 _user_data 字段内的结构中的某些值。_reset_extfn 可重复调用。

  ```
  void (*_reset_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_next_value_extfn** – 针对每组新输入参数值调用的必需函数指针。该函数不设置集合结果。可通过 get_value 回调函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

  ```
  void (*_next_value_extfn)(a_v3_extfn_aggregate_context *cntxt,
  void *args_handle);
  ```

  注意: get_piece 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。
- **_evaluate_extfn** – 必需函数指针，调用它可返回生成的集合结果值。_evaluate_extfn 将通过 set_value 回调函数发送至服务器。

```
void (*_evaluate_extfn)(a_v3_extfn_aggregate_context *cntxt, void
*args_handle);
```

- **_drop_value_extfn** – 针对超出移动窗口构架的每个输入参数值集调用的可选函数指针。不要使用此函数设置集合结果。可通过 get_value 回调函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需访问输入参数值。在下列情况下，请将 _drop_value_extfn 设置为空指针：
  - 集合不能与窗口构架一起使用。
  - 集合在某种程度上不可逆。
  - 用户对最佳性能没有兴趣。

注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

如果未提供该函数，且用户已经指定移动窗口，则可在每次窗口构架移动时调用重置函数，并通过调用 next_value 函数包含该窗口内的现有各行。最后调用求值函数。

但是，如果已提供该函数，则每次窗口构架移动时，均将针对超出窗口架构的各行调用 drop_value 函数，然后针对刚刚添加到窗口构架中的各行调用 next_value 函数。最后，调用求值函数以生成集合结果。

```
void (*_drop_value_extfn)(a_v3_extfn_aggregate_context    *cntxt,
void *args_handle);
```

- **_evaluate_cumulative_extfn** – 针对每组新输入参数值调用的可选函数指针。如果已提供该函数，且在跨越从 UNBOUNDED PRECEDING 到 CURRENT ROW 的行式窗口构架内使用，则可调用此函数，而无需调用 next_value 函数以及紧邻其后调用的求值函数。_evaluate_cumulative_extfn 必须通过 set_value 回调设置集合结果。可通过 get_value 回调函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

```
void (*_evaluate_cumulative_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

- **_next_subaggregate_extfn** – 可选回调函数指针，通过 _evaluate_superaggregate_extfn 函数（在某些用法中还使用 _drop_subaggregate_extfn 函数），支持通过并行和部分结果集合来优化集合的某些用法。

  某些集合用作简单集合（换言之即带有 OVER 子句的非 OLAP 样式集合）时，可以通过首先生成一组中间集合结果来分区，其中每个中间结果均为来自一个不连接的输入行子集的计算结果。此类可分区集合的示例包括：
  - SUM，其中，可通过针对每个不连接的输入行子集执行 SUM，然后在 sub-SUM 上执行 SUM，由此计算得出最终 SUM；以及

- COUNT(*)，其中，可通过针对每个不连接的输入行子集执行 COUNT，然后在每个分区的 COUNT 上执行 SUM，由此计算得出最终 COUNT。

当集合满足上述条件时，服务器可以选择并行执行该集合的计算。对于集合 UDF，仅当已提供 _next_subaggregate_extfn 回调和 _evaluate_superaggregate_extfn 回调时，才能应用该优化。此用法模式不需要使用 _drop_subaggregate_extfn。

同样，如果能一同使用集合和基于 RANGE 的 OVER 子句，则 _next_subaggregate_extfn、_drop_subaggregate_extfn 和 _evaluate_superaggregate_extfn) 函数 均由集合 UDF 的实现提供时可以进行优化。

_next_subaggregate_extfn 不设置集合最终结果，而且按照定义，它将使用与集合 UDF 的返回值完全相同的数据类型的唯一输入参数值。可通过 get_value 回调函数访问子集输入值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

子集合与超级集合之间不可进行直接通信；所有此类通信将由服务器处理。子集合和超级集合不共用上下文结构。而将各个子集合完全视为非分区集合。独立超级集合将看到类似如下的调用模式：

```
       _start_extfn
 _reset_extfn
       _next_subaggregate_extfn (repeated 0 to N times)
       _evaluate_superaggregate_extfn
       _finish_extfn
```

```
void (*_next_subaggregate_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

- **_drop_subaggregate_extfn** – 可选回调函数指针，与 _next_subaggregate_extfn 和 _evaluate_superaggregate_extfn 一起使用，支持通过部分集合来优化使用基于 RANGE 的 OVER 子句时的某些用法。每当共享通用排序键值的一组行全部超出移动窗口时即调用 _drop_subaggregate_extfn。仅当所有三个函数均由 UDF 提供时才能应用此优化。

```
void (*_drop_subaggregate_extfn)(a_v3_extfn_aggregate_context
*cntxt,    void *args_handle);
```

- **_evaluate_superaggregate_extfn** – 可选回调函数指针，与 _next_subaggregate_extfn（有时也可与 _drop_subaggregate_extfn）一起使用时，支持通过并行运行来优化某些用法。

  将要返回分区集合结果时，可按如上所述调用 _evaluate_superaggregate_extfn。通过使用 a_v3_extfn_aggregate_context 结构中的 set_value 回调函数将结果值发送至服务器：

```
void (*_evaluate_superaggregate_extfn)
(a_v3_extfn_aggregate_context *cntxt, void *args_handle);
```

- **NULL fields** – 将以下字段初始化为 NULL：

```
void *    reserved1_must_be_null;
void *    reserved2_must_be_null;
void *    reserved3_must_be_null;
```

```
void *    reserved4_must_be_null;
void *    reserved5_must_be_null;
```

- **Status indicator bit field** – 包含允许引擎优化用于处理集合的算法的指示符的位字段。

```
a_sql_uint32  indicators;
```

- **_calculation_context_size** – 服务器为每个 UDF 计算上下文分配的字节数。服务器可以在查询处理过程中分配多个计算上下文。

  a_v3_extfn_aggregate_context_user_calculation_context 中可使用当前活动组的上下文。

```
short _calculation_context_size;
```

- **_calculation_context_alignment** – 指定用户计算上下文的对齐要求。有效值包括 1、2、4 或 8。

```
short _calculation_context_alignment;
```

- **External memory requirments** – 下列字段允许优化程序考虑外部分配的内存的开销。有了这些值，优化程序就能考虑可以执行多个并发计算的程度。这些计数器应当基于典型的行或组来估计，并且不应为最大值。如果没有 UDF 分配的内存，则将这些字段设为零。
  - external_bytes_per_group – 分配到每个集合开头的组的内存量。通常为所有在 reset() 调用期间分配的内存。
  - external_bytes_per_row — UDF 为组中每一行分配的内存量。通常为 next_value() 期间所分配的内存量。

```
double            external_bytes_per_group;
double            external_bytes_per_row;
```

- **Reserved fields for future use** – 初始化下列字段：

```
a_sql_uint64      reserved6_must_be_null;
a_sql_uint64      reserved7_must_be_null;
a_sql_uint64      reserved8_must_be_null;
a_sql_uint64      reserved9_must_be_null;
a_sql_uint64      reserved10_must_be_null;
```

- **Closing syntax** — 用下列语法完成描述符：

```
//---------- For Server Internal Use Only ----------
void  * _for_server_internal_use;
} a_extfn_aggregate;
```

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

计算上下文

_user_calculation_context 字段允许服务器对多组数据同时执行计算。

集合 UDF 必须在其处理各行时保留中间计数器， 以便进行计算。这些计数器的简单管理模型， 是由起始 API 函数分配内存，将指针存入其 集合上下文 _user_data 字段，然后由集合的终止 API 释放内存。服务器可通过 基于 _user_calculation_context 字段的备选方法对多组数据同时执行计算。

_user_calculation_context 字段是服务器分配的内存指针，由服务器为每个并发处理的组创建。服务器确保 _user_calculation_context 始终为当前正在处理的那一组行指向正确的计算上下文。在 UDF API 调用之间，服务器可能会根据数据分配新的 _user_calculation_context 值。服务器可能会在处理查询期间将计算上下文区域保存并恢复到磁盘。

UDF 将所有中间计算值存储在此字段中。以下说明一个典型的用法：

```
struct my_average_context
{
        int    sum;
        int    count;
};

reset(a_v3_aggregate_context *context)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count = 0;
    mycontext->sum = 0;
}

    next_value(a_v3_aggregate_context *context, void *args_handle)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count++;
    ..
}
```

在此模型中，_user_data 字段仍然可用，但其中不能存储任何有关中间结果计算的值。_user_calculation_context 在开始和完成入口点处均为 NULL。

要用 _user_calculation_context 启用并发处理功能，必须通过 UDF 对其计算上下文指定大小和对齐要求，并且定义 用于存储其值的结构，并用该结构的 sizeof() 设置 a_v3_extfn_aggregate 和 _calculation_context_size。

UDF 还必须通过 _calculation_context_alignment 指定 _user_calculation_context 的数据对齐要求。如果 user_calculation_context 内存只包含一个字符字节数组，则无需特别的对齐，并且可以指定 1 字节对齐。同样，双精度浮点数值可能需要 8 字节对齐。对齐要求因平台和数据类型的不同而异。指定比所需的大的对齐始终都是可行的；但若使用最小的对齐，内存使用效率会更高。

集合 UDF 上下文结构

集合 UDF 上下文结构 a_v3_extfn_aggregate_context 具有和标量 UDF 上下文结构完全相同的回调函数指针集。

此外，与标量 UDF 上下文类似，它具有读/写 _user_data 指针，还具有一组描述当前使用情况和位置的只读数据字段。一个语句中的每个 UDF 唯一实例都具有一个集合 UDF 上下文实例，它在调用时传递到集合 UDF 描述符结构中指定的每一个函数。集合上下文结构定义为：

- **typedef struct a_v3_extfn_aggregate_context** — 为在查询中引用的外部函数的每一个实例创建一个上下文。如果用在查询内并行的子树中，则并行子树会具有单独的上下文。
- **Callbacks available via the context** — 回调例程的常见参数包括：
  - **arg_handle** — 由服务器提供的函数实例和参数的句柄。
  - **arg_num** — 参数数目。返回值为 0 到 N。
  - **data** — 参数数据的指针。

  上下文必须在 get_piece 之前调用 get_value，但仅在 piece_len 小于 total_len 时才需要调用 get_piece。

```
short (SQL_CALLBACK *get_value)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value
        );
short (SQL_CALLBACK *get_piece)(
        void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value,
        a_sql_uint32    offset
        );
```

- **Determining whether an argument is a constant** – UDF 可以询问给定的参数是否为常量。这非常有用。例如，允许工作在第一次调用 _next_value 函数时执行一次，而不是在每次调用 _next_value 函数时执行。

```
short (SQL_CALLBACK *get_value_is_constant)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        a_sql_uint32 *  value_is_constant
        );
```

- **Returning a null value** — 要返回空值，可将 an_extfn_value 中的“data”设为 NULL。调用 set_value 时会忽略 total_len 字段。如果 append 为 FALSE，则提供的数据会变为参数的值；否则，该数据会附加到参数的当前值中。预期的情形为，在使用 append=TRUE 为一个参数调用 set_value 前，先使用 append=FALSE 为该参数进行调用。对于固定长度的数据类型（换句话说，所有数字数据类型），会忽略 append 字段。

```
short (SQL_CALLBACK *set_value)(
  void *          arg_handle,
        an_extfn_value *value,
        short              append
        );
```

- **Determining whether the statement was interrupted** – 如果 UDF 入口点执行工作的时间较长（许多秒），则可能的话，它应当每秒或每两秒调用一次 get_is_cancelled 回调，以查看用户是否中断了当前的语句。如果语句已被中断，则返回非零值，并且 UDF 入口点应立即执行。最后，调用 _finish_extfn 函数执行任何必要的清理，但随后不再调用任何其它的 UDF 入口点。

```
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)
(a_v3_extfn_aggregate_context * cntxt);
```

- **Sending error messages** – 如果 UDF 入口点遇到某个错误，该错误可导致向用户发回错误消息并且关闭当前语句，则应调用 set_error 回调例程。set_error 导致当前语句回退；用户会看到 `Error from external UDF: <error_desc_string>`，并且 SQLCODE 是 <error_number> 的负值形式。调用 set_error 之后，UDF 入口点会立即执行返回。最后，调用 _finish_extfn 函数执行任何必要的清理，但随后不再调用任何其它的 UDF 入口点。

```
void (SQL_CALLBACK *set_error)(
  a_v3_extfn_aggregate_context * cntxt,
        a_sql_uint32    error_number,
        // use error_number values >17000 & <100000
        const char *    error_desc_string
        );
```

- **Writing messages to the message log** — 长度超过 255 字节的消息会被截断。

```
void (SQL_CALLBACK *log_message)(
  const char *msg,
        short msg_length
  );
```

- **Converting one data type to another** – 对于输入：
  - **an_extfn_value.data** – 输入数据指针。
  - **an_extfn_value.total_len** – 输入数据的长度。
  - **an_extfn_value.type** — 输入的 DT_ 数据类型。

  对于输出：
  - **an_extfn_value.data** — UDF 提供的输出数据指针。
  - **an_extfn_value.piece_len** – 输出数据的最大长度。
  - **an_extfn_value.total_len** – 服务器设置的转换输出长度。
  - **an_extfn_value.type** — 所需输出的 DT_ 数据类型。

```
short (SQL_CALLBACK *convert_value)(
  an_extfn_value   *input,
        an_extfn_value   *output
        );
```

- **Fields reserved for future use** — 这些留待将来使用：

```
void  *   reserved1;
void  *   reserved2;
void  *   reserved3;
void  *   reserved4;
void  *   reserved5;
```

- **Data available from the context** — 此数据指针可通过任何用法使用外部例程所需的任何上下文数据填充。UDF 会分配和释放此内存。每个语句都有处于活动状态的单个 _user_data 实例。不要将此内存用于中间结果值。

```
void  * _user_data;
```

- **Currently active calculation context** — UDF 应使用此内存位置存储计算集合的中间值。此内存由服务器根据 a_v3_extfn_aggregate 中请求的大小进行分配。中间计算必须存储在此内存中，因为引擎可能会对一个以上的组执行并发计算。在每个 UDF 入口点之前，服务器会确保正确的上下文数据处于活动状态。

```
void  * _user_calculation_context;
```

- **Other available aggregate information** – 在包括 start_extfn 在内的所有外部函数入口点上均可用。零表示未知或不适用的值。每个分区或每组中估计的平均行数。
  - **a_sql_uint64 _max_rows_in_frame;** — 计算窗口构架中定义的最大行数。对于基于范围的窗口，这表示唯一值。零表示未知或不适用的值。
  - **a_sql_uint64 _estimated_rows_per_partition;** – 显示每个分区或每组中估计的平均行数。0 表示未知或不适用的值。
  - **a_sql_uint32 _is_used_as_a_superaggregate;** – 标识此实例为普通集合还是超级集合。如果实例为普通集合，则返回的结果为 0。
- **Determining window specifications** — 查询上存在窗口时的窗口规范：
  - **a_sql_uint32 _is_window_used;** — 确定语句是否为窗口化的。
  - **a_sql_uint32 _window_has_unbounded_preceding;** – 返回值为 0，表示窗口没有未绑定的之前行。
  - **a_sql_uint32 _window_contains_current_row;** – 返回值为 0，表示窗口不包含当前行。
  - **a_sql_uint32 _window_is_range_based;** – 如果返回码为 1，则窗口基于范围。如果返回码为 0，则窗口基于行。
- **Available at reset_extfn() calls** – 返回当前分区中的实际行数；或者，若为非窗口集合，则返回 0。

  ```
  a_sql_uint64 _num_rows_in_partition;
  ```
- **Available only at evaluate_extfn() calls for windowed aggregates** — 分区中当前求值的行号（从 1 开始）。这在未受限制窗口的求值阶段很有用。

  ```
  a_sql_uint64 _result_row_from_start_of_partition;
  ```
- **Closing syntax** — 用下列语法完成上下文：

  ```
  //---------- For Server Internal Use Only ----------
    void  * _for_server_internal_use;
  } a_v3_extfn_aggregate_context;
  ```

外部集合函数上下文：**a_v3_extfn_aggregate_context**：

a_v3_extfn_aggregate_context 方法的结构	描述
void set_cannot_be_distributed(a_v3_extfn_aggregate_context * cntxt)	即使已达到库级别的分配标准，也可在 UDF 级别禁止分配。缺省情况下，假定库可分配时 UDF 也可分配。UDF 负责作出 禁止对服务器进行分配的决策。

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

示例：my_sum 定义
集合 UDF **my_sum** 示例仅可对整数执行运算。

my_sum 定义

由于 **my_sum** 可用在任何上下文中（如同 SUM），因此已提供所有优化过的可选条目。在此例中，也可将普通 _evaluate_extfn 函数 用作 _evaluate_superaggregate_extfn 函数。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>


//  Simple aggregate UDF that adds up a set of
//  integer arguments, and whenever asked returns
//  the resulting big integer total.  For int
//  arguments, the only difference between this
//  UDF and the SUM built-in aggregate is that this
//  UDF will return NULL if there are no input rows.
//
//  The start function creates a little structure for
//  the running total, and the finish function then
//  deallocates it.
//
//  Since there are no aggregate usage restrictions
//  for this aggregate UDF, the corresponding SQL declaration
//  will look like:
//
//              CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
//                      RETURNS BIGINT
//                      ON EMPTY INPUT RETURNS NULL
//                      EXTERNAL NAME 'my_integer_sum@libudfex'


typedef struct my_total {
  a_sql_int64   _total;
  a_sql_uint64  _num_nonnulls_seen;
} my_total;


extern "C"
void my_integer_sum_start(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
void my_integer_sum_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
```

```
void my_integer_sum_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;
  cptr->_total = 0;
  cptr->_num_nonnulls_seen = 0;
}


extern "C"
void my_integer_sum_next_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}


extern "C"
void my_integer_sum_drop_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total -= arg1;
    cptr->_num_nonnulls_seen--;
  }
}


extern "C"
void my_integer_sum_evaluate(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  outval;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
```

```
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


extern "C"
void my_integer_sum_cum_evaluate(
                       a_v3_extfn_aggregate_context *cntxt,
                       void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  int arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it into the
total.
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


extern "C"
void my_integer_sum_next_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
```

```
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}


extern "C"
void my_integer_sum_drop_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total -= arg1;
    cptr->_num_nonnulls_seen--;
  }
}



a_v3_extfn_aggregate my_integer_sum_descriptor =
{
        &my_integer_sum_start,
        &my_integer_sum_finish,
        &my_integer_sum_reset,
        &my_integer_sum_next_value,
        &my_integer_sum_evaluate,
        &my_integer_sum_drop_value,
        &my_integer_sum_cum_evaluate,
        &my_integer_sum_next_subagg_value,
        &my_integer_sum_drop_subagg_value,
        &my_integer_sum_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0, // indicators
        ( short )sizeof( my_total ), // context size
        8, // context alignment
        0.0, //external_bytes_per_group
        0.0, // external bytes per row
        0, // reserved6_must_be_null
        0, // reserved7_must_be_null
        0, // reserved8_must_be_null
        0, // reserved9_must_be_null
        0, // reserved10_must_be_null
```

```
        NULL // _for_server_internal_use
};

extern "C"
a_v3_extfn_aggregate *my_integer_sum()
{
  return &my_integer_sum_descriptor;
}
```

示例：my_bit_xor 定义

集合 UDF **my_bit_xor** 示例类似于 SA 内置 BIT_XOR，但 **my_bit_xor** 仅可对无符号整数执行运算。

my_bit_xor 定义

由于输入和输出数据类型是相同的，可使用普通的 _next_value_extfn 和 _evaluate_extfn 函数来累积子集合值并生成超级集合结果。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>


//  Generic aggregate UDF that exclusive-ORs a set of
//  unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  Since there are no aggregate usage restrictions
//  for this aggregate UDF, the corresponding SQL declaration
//  will look like:
//
//             CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED
INT)
//                       RETURNS UNSIGNED INT
//                       ON EMPTY INPUT RETURNS NULL
//                       EXTERNAL NAME 'my_bit_xor@libudfex'


typedef struct my_xor_result {
  a_sql_uint64 _num_nonnulls_seen;
  a_sql_uint32 _xor_result;
} my_xor_result;

#if defined __cplusplus
extern "C" {
#endif

static void my_xor_start(a_v3_extfn_aggregate_context *cntxt)
{
}
```

```
static void my_xor_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;
  cptr->_xor_result = 0;
  cptr->_num_nonnulls_seen = 0;
}


static void my_xor_next_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }
}


static void my_xor_drop_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and remove it from the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen--;
  }
}


static void my_xor_evaluate(a_v3_extfn_aggregate_context *cntxt,
                            void *arg_handle)
{
  an_extfn_value  outval;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;
```

```
  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static void my_xor_cum_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and include it in the result,
  //  unless that input value is null.
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value
  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_xor_descriptor =
    {
        &my_xor_start,
        &my_xor_finish,
        &my_xor_reset,
        &my_xor_next_value,
        &my_xor_evaluate,
        &my_xor_drop_value,
        &my_xor_cum_evaluate,
        &my_xor_next_value,
        &my_xor_drop_value,
        &my_xor_evaluate,
        NULL, // reserved1_must_be_null
```

```
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0,    // indicators
        ( short )sizeof( my_xor_result ), // context size
        8,    // context alignment
        0.0,  // external_bytes_per_group
        0.0,  // external bytes per row
        0,    // reserved6_must_be_null
        0,    // reserved7_must_be_null
        0,    // reserved8_must_be_null
        0,    // reserved9_must_be_null
        0,    // reserved10_must_be_null
        NULL  // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_bit_xor()
{
  return &my_xor_descriptor;
}

#if defined __cplusplus
}
#endif
```

示例：my_bit_or 定义

集合 UDF **my_bit_or** 示例类似于 SA 内置 BIT_OR，但 **my_bit_or** 仅可对无符号整数执行运算，并且仅可用作简单集合。

my_bit_or 定义

my_bit_or 定义比 **my_bit_xor** 示例简单一些。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>


//  A simple (non-OLAP) aggregate UDF that ORs a set
//  of unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  The aggregate usage restrictions for this aggregate UDF
//  only allow its use as a simple aggregate, so the
//  corresponding SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
//            RETURNS UNSIGNED INT
//                      ON EMPTY INPUT RETURNS NULL
//            OVER NOT ALLOWED
```

```
//            EXTERNAL NAME 'my_bit_or@libudfex'


typedef struct my_or_result {
  a_sql_uint32 _or_result;
  a_sql_uint32 _non_null_seen;
} my_or_result;



#if defined __cplusplus
extern "C" {
#endif

static void my_or_start(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_finish(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;
  cptr->_or_result = 0;
  cptr->_non_null_seen = 0;
}


static void my_or_next_value(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data)
  {
      arg1 = *((a_sql_uint32 *)arg.data);
      cptr->_or_result |= arg1;
      cptr->_non_null_seen = 1;
  }
}


static void my_or_evaluate(a_v3_extfn_aggregate_context *cntxt,
                           void *arg_handle)
{
  an_extfn_value  outval;
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;
```

```
  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_non_null_seen)
  {
      outval.data = &cptr->_or_result;
  }
  else
  {
      // Return null if no values seen
      outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}



static a_v3_extfn_aggregate my_or_descriptor =
{
    &my_or_start,
    &my_or_finish,
    &my_or_reset,
    &my_or_next_value,
    &my_or_evaluate,
    NULL, // drop_val_extfn
    NULL, // cume_eval,
    NULL, // next_subaggregate_extfn
    NULL, // drop_subaggregate_extfn
    NULL, // evaluate_superaggregate_extfn
    NULL, // reserved1_must_be_null
    NULL, // reserved2_must_be_null
    NULL, // reserved3_must_be_null
    NULL, // reserved4_must_be_null
    NULL, // reserved5_must_be_null
    0, // indicators
    ( short )sizeof( my_or_result ), // context size
    8, // context alignment
    0.0, //external_bytes_per_group
    0.0, // external bytes per row
    0, // reserved6_must_be_null
    0, // reserved7_must_be_null
    0, // reserved8_must_be_null
    0, // reserved9_must_be_null
    0, // reserved10_must_be_null
    NULL // _for_server_internal_use
};


extern "C"
a_v3_extfn_aggregate *my_bit_or()
{
  return &my_or_descriptor;
}


#if defined __cplusplus
```

```
}
#endif
```

示例：my_interpolate 定义

集合 UDF **my_interpolate** 示例是 OLAP 样式集合 UDF， 会尝试在每个方向上执行任何相邻空值集到最近 非空值的线性插值操作，从而将空值填入序列。

my_interpolate 定义

要以合理的开销运行，**my_interpolate** 必须通过 固定宽度、基于行的窗口运行， 但用户能根据预计的相邻空值数上限设置窗口宽度。 如果输入给定行的值不是空值， 则该行的结果与输入值相同。这种函数可接收一组双精度浮点值， 然后生成一组最终双精度值。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>

//  MY_INTERPOLATE
//
//  OLAP-style aggregate UDF that accepts a double precision
//  floating point argument.  If the current argument value is
//  not NULL, then the result value is the same as the
//  argument value.  On the other hand, if the current row's
//  argument value is NULL, then the result, where possible,
//  will be the arithmetic interpolation across the nearest
//  preceding and nearest following values that are not NULL.
//  In all cases the result is also a double precision value.
//
//  The start function creates a structure for maintaining the
//  argument values within the window including their NULLness.
//  The finish function then deallocates this structure.
//
//  Since there are some strict aggregate usage restrictions
//  for this aggregate (must be used with a row-based window
//  frame that includes the current row), the corresponding
//  SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_interpolate(IN arg1 DOUBLE)
//                       RETURNS DOUBLE
//                       OVER REQUIRED
//                       WINDOW FRAME REQUIRED
//                              RANGE NOT ALLOWED
//                              PRECEDING REQUIRED
//                              UNBOUNDED PRECEDING NOT ALLOWED
//                              FOLLOWING REQUIRED
//                              UNBOUNDED FOLLOWING NOT ALLOWED
//                       EXTERNAL NAME  'my_interpolate@libudfex'


typedef struct my_window {
  int      _allocated_elem;
  int      _first_used;
  int      _next_insert_loc;
```

```
  int     *_is_null;
  double  *_dbl_val;
  int      _num_rows_in_frame;
} my_window;



#if defined __cplusplus
extern "C" {
#endif


static void my_interpolate_reset(a_v3_extfn_aggregate_context
*cntxt)
{
  assert(cntxt->_user_data);
  my_window *cptr = (my_window *)cntxt->_user_data;

  cptr->_first_used = 0;
  cptr->_next_insert_loc = 0;
  cptr->_num_rows_in_frame = 0;
  for (int i=0; i<cptr->_allocated_elem; i++) {
    cptr->_is_null[i] = 1;
  }
}


static void my_interpolate_start(a_v3_extfn_aggregate_context
*cntxt)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  // Make sure function was defined correctly
  if (!cntxt->_is_window_used)
  {
      cntxt->set_error(cntxt, 20001, "Function requires window");
      return;
  }
  if (cntxt->_window_has_unbounded_preceding ||
      cntxt->_window_has_unbounded_following)
  {
      cntxt->set_error(cntxt, 20002, "Window cannot be unbounded");
      return;
  }
  if (cntxt->_window_is_range_based)
  {
      cntxt->set_error(cntxt, 20003, "Window must be row based");
      return;
  }

  if (!cptr) {
    //
    cptr = (my_window *)malloc(sizeof(my_window));
    if (cptr) {
      cptr->_is_null = 0;
      cptr->_dbl_val = 0;
```

```
      cptr->_num_rows_in_frame = 0;
      cptr->_allocated_elem = ( int )cntxt->_max_rows_in_frame;
      cptr->_is_null = (int *)malloc(cptr->_allocated_elem
                                     * sizeof(int));
      cptr->_dbl_val = (double *)malloc(cptr->_allocated_elem
                                        * sizeof(double));
      cntxt->_user_data = cptr;
    }
  }
  if (!cptr || !cptr->_is_null || !cptr->_dbl_val) {
      // Terminate this query
      cntxt->set_error(cntxt, 20000, "Unable to allocate memory");
      return;
  }
  my_interpolate_reset(cntxt);
}


static void my_interpolate_finish(a_v3_extfn_aggregate_context
*cntxt)
{
  if (cntxt->_user_data) {
    my_window *cptr = (my_window *)cntxt->_user_data;
    if (cptr->_is_null) {
      free(cptr->_is_null);
      cptr->_is_null = 0;
    }
    if (cptr->_dbl_val) {
      free(cptr->_dbl_val);
      cptr->_dbl_val = 0;
    }
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_interpolate_next_value(a_v3_extfn_aggregate_context
*cntxt,
                                      void *arg_handle)
{
  an_extfn_value  arg;
  double arg1;
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Get the one argument, and stash its value
  //  within the rotating window arrays
  //
  int curr_cell_num = cptr->_next_insert_loc % cptr-
>_allocated_elem;
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data != NULL ) {
    arg1 = *((double *)arg.data);
    cptr->_dbl_val[curr_cell_num] = arg1;
    cptr->_is_null[curr_cell_num] = 0;
  } else {
    cptr->_is_null[curr_cell_num] = 1;
```

```
  }

  //  Then increment the insertion location and number of rows in
frame
  cptr->_next_insert_loc = ((cptr->_next_insert_loc + 1)
                            % cptr->_allocated_elem);
  cptr->_num_rows_in_frame++;
}


static void my_interpolate_drop_value(a_v3_extfn_aggregate_context
*cntxt,
                                      void * /*arg_handle*/)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Drop one value from the window by incrementing past it and
  // decrement the number of rows in the frame
  cptr->_first_used = ((cptr->_first_used + 1) % cptr-
>_allocated_elem);
  cptr->_num_rows_in_frame--;
}


static void my_interpolate_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                    void *arg_handle)
{

  an_extfn_value  outval;
  my_window *cptr = (my_window *)cntxt->_user_data;
  double  result;
  int     result_is_null = 1;
  double  preceding_value;
  int     preceding_value_is_null = 1;
  double  preceding_distance = 0;
  double  following_value;
  int     following_value_is_null = 1;
  double  following_distance = 0;
  int j;

  //  Determine which cell is the current cell
  int curr_cell_num =
      ((int)(cntxt->_result_row_from_start_of_partition-1))%cptr-
>_allocated_elem;
  int tmp_cell_num;

  int result_row_offset_from_start_of_frame = cptr->_first_used <=
curr_cell_num ?
          ( curr_cell_num - cptr->_first_used ) :
          ( curr_cell_num + cptr->_allocated_elem - cptr-
>_first_used );
```

```
  //  Compute the result value
  if (cptr->_is_null[curr_cell_num] == 0) {
    //
    //  If the current rows input value is not NULL, then there is
    //  no need to interpolate, just use that input value.
    //
    result = cptr->_dbl_val[curr_cell_num];
    result_is_null = 0;
    //
  } else {
    //
    //  If the current rows input value is NULL, then we do
    //  need to interpolate to find the correct result value.
    //  First, find the nearest following non-NULL argument
    //  value after the current row.
    //
    int rows_following = cptr->_num_rows_in_frame -
                result_row_offset_from_start_of_frame - 1;
    for (j=0; j<rows_following; j++) {
      tmp_cell_num = ((curr_cell_num + j + 1) % cptr-
>_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          following_value = cptr->_dbl_val[tmp_cell_num];
          following_value_is_null = 0;
          following_distance = j + 1;
          break;
      }
    }
    //  Second, find the nearest preceding non-NULL
    //  argument value before the current row.
    //
    int rows_before = result_row_offset_from_start_of_frame;
    for (j=0; j<rows_before; j++) {
     tmp_cell_num = ((curr_cell_num + cptr->_allocated_elem - j - 1)
                     % cptr->_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          preceding_value = cptr->_dbl_val[tmp_cell_num];
          preceding_value_is_null = 0;
          preceding_distance = j + 1;
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceding_value_is_null && !following_value_is_null) {
      //
     //  No choice but to mirror the nearest following non-NULL value
      //  Example:
      //
      //    Inputs:  NULL    Result of my_interpolate:  40.0
      //             NULL                               40.0
      //             40.0                               40.0
      //
      result = following_value;
      result_is_null = 0;
      //
```

```
   } else if (!preceding_value_is_null && following_value_is_null) {
     //
     //  No choice but to mirror the nearest preceding non-NULL value
     //  Example:
     //
     //    Inputs:  10.0    Result of my_interpolate:  10.0
     //             NULL                               10.0
     //
     result = preceding_value;
     result_is_null = 0;
     //
   } else if (!preceding_value_is_null && !following_value_is_null)
{
     //
     //  Here we get to do real interpolation based on the
     //  nearest preceding non-NULL value, the nearest following
     //  non-NULL value, and the relative distances to each.
     //  Examples:
     //
     //    Inputs:  10.0    Result of my_interpolate:  10.0
     //             NULL                               20.0
     //             NULL                               30.0
     //             40.0                               40.0
     //
     //    Inputs:  10.0    Result of my_interpolate:  10.0
     //             NULL                               25.0
     //             40.0                               40.0
     //
     result = (  preceding_value
                 + (  (following_value - preceding_value)
                      * (  preceding_distance
                           / (preceding_distance +
following_distance))));
     result_is_null = 0;
   }
  }

  //  And last, pass the result value out
  outval.type = DT_DOUBLE;
  outval.piece_len = sizeof(double);
  if (result_is_null) {
    outval.data = 0;
  } else {
    outval.data = &result;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_interpolate_descriptor =
    {
        &my_interpolate_start,
        &my_interpolate_finish,
        &my_interpolate_reset,
        &my_interpolate_next_value, //( timeseries_expression )
        &my_interpolate_evaluate,
```

```
        &my_interpolate_drop_value,
       NULL, // cume_eval,
       NULL, // next_subaggregate_extfn
       NULL, // drop_subaggregate_extfn
       NULL, // evaluate_superaggregate_extfn
       NULL, // reserved1_must_be_null
       NULL, // reserved2_must_be_null
       NULL, // reserved3_must_be_null
       NULL, // reserved4_must_be_null
       NULL, // reserved5_must_be_null
       0, // indicators
       0, // context size
       0, // context alignment
       0.0, //external_bytes_per_group
       ( double )sizeof( double ), // external bytes per row
       0, // reserved6_must_be_null
       0, // reserved7_must_be_null
       0, // reserved8_must_be_null
       0, // reserved9_must_be_null
       0, // reserved10_must_be_null
       NULL // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_interpolate()
  { return &my_interpolate_descriptor; }

#if defined __cplusplus
  }
#endif
```

集合用户定义函数的上下文存储

上下文区域用于在相同查询（特别是 OLAP 样式的查询）内的多个 UDF 调用之间传递数据。

上下文变量决定是否由 UDF 自身（强制 Sybase IQ 服务器以串行方式运行 UDF）对集合函数的中间结果进行管理，又或内存是否由 Sybase IQ 服务器进行管理。

如果将 _calculation_context_size 设置为 0，则 UDF 需要管理内存中的所有中间结果，并且强制 Sybase IQ 服务器依序对数据调用 UDF（而不能在 OLAP 查询期间以并行方式调用 UDF 的多个实例）。

如果将 _calculation_context_size 设置为非零值，则 Sybase IQ 服务器将为每个 UDF 调用管理一个单独的上下文区域，并且允许以并行方式调用 UDF 的多个实例。为使内存使用效率最佳，可以考虑将 _calculation_context_alignment 设置为小于默认值（取决于所需的上下文存储空间的大小）。

有关上下文存储的详细信息，请参见集合 UDF 描述符结构 （第 55 页）“集合 UDF 描述符结构”一节中的 _calculation_context_size 和 _calculation_context_alignment 的说明。这些变量位于描述符结构的末尾附近。

有关上下文存储的使用的详细讨论，请参见计算上下文 （第 58 页）。

重要： 若要将内存中的中间结果存储在集合 UDF 中，请使用 _start_extfn 函数初始化内存，然后使用 _finish_extfn 函数清除并释放任何内存。

另请参见

- 声明集合 UDF （第 46 页）
- 定义集合 UDF （第 52 页）

调用标量和集合 UDF

在您使用内置的非集合函数的任何位置，都可以根据权限使用用户定义的函数。

以下 Interactive SQL 语句从包含名字和姓氏的两列返回全名：

```
SELECT fullname (GivenName, LastName)
FROM Employees;
```

fullname (Employees.GivenName,Employees.SurName)
Fran Whitney
Matthew Cobb
Philip Chin
...

以下语句从提供的名字和姓氏返回全名：

```
SELECT fullname ('Jane', 'Smith');
```

fullname ('Jane','Smith')
Jane Smith

已被授予函数的“执行”权限的任何用户都可以使用 *fullname* 函数。

标量和集合 UDF 调用模式

调用模式为函数在收集结果时所执行的步骤。

标量和集合 UDF 回调函数

这组回调函数由引擎通过 `a_v3_extfn_scalar_context` 结构提供，并用在用户的 UDF 函数中。

- **get_value** – 在计算方法中用于检索每个输入参数值的函数。 对于窄型参数数据（小于 256 字节）， 调用 get_value 就足以检索 整个参数值。对于宽型参数数据，

如果传递到此回调的 an_extfn_value 结构中的 piece_len 字段返回的值小于 total_len 字段中的值，则用 get_piece 回调检索其余输入值。

- **get_piece** – 用于检索长参数输入值的后续片段的函数。

  注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

- **get_is_constant** – 用于判断指定的输入参数值是否为常量的函数。 可用于优化 UDF，例如， 可在首次调用 _evaluate_extfn 函数期间运行一次， 而不是每当执行计算调用时都运行。
- **set_value** – 在计算函数中用于将此次调用的 UDF 结果值告知服务器的函数。 如果结果是窄型数据，则调用一次 set_value 足矣。但是，如果结果数据值是宽型数据， 则必须多次调用 set_value， 才能传递全部值，而且对于除最后片段以外的每个片段， 回调的 append 参数应为 true。要返回 空的结果，UDF 应该用空的指针设置结果值 an_extfn_value 结构 中的数据字段。
- **get_is_cancelled** – 用于判断是否已取消语句的函数。如果 UDF 条目的运行时间加长（达到数秒），则应该（如有可能）每秒或每两秒都调用一次 get_is_cancelled 回调函数，以确定用户是否已打断当前语句的执行。如果未打断当前语句的执行，则返回值是 0。

  Sybase IQ 能处理极大的数据集，某些查询也可长时间运行。查询的执行时间偶尔会非常长。如果完成查询所耗的时间过长，则用户可通过 SQL 客户端取消查询。本地函数跟踪用户取消查询的时间。UDF 还必须以这样的方式写入，即可以跟踪用户是否已取消查询。换句话说，UDF 应该支持用户取消调用 UDF 的运行时间很长的查询的功能。
- **set_error** – 可用于将错误告知服务器并最终告知用户的函数。 UDF 条目遇到错误时可调用此回调例程， 使错误消息送回用户。 调用时 set_error 会从当前语句退回，用户会收到外部 `UDF 错误: error_desc_string`， SQLCODE 是所提供的 error_number 的求反形式。 要防止与现有错误发生冲突，用于 UDF 的 error_number 值应该介于 17000 和 99999 之间。“error_desc_string”的长度上限为 140 个字符。
- **log_message** – 用于向服务器消息日志发送消息的函数。消息字符串必须是可显示的文本字符串，不长于 255 字节。
- **convert_value** – 可通过该函数在各类数据之间转换数据。主要用于在 DT_DATE、DT_TIME、DT_TIMESTAMP 和 DT_TIMESTAMP_STRUCT 之间转换数据。输入和输出 an_extfn_value 传递到该函数。

另请参见

- 标量 UDF 调用模式 （第 82 页）
- 集合 UDF 调用模式 （第 82 页）
- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

标量 UDF 调用模式

标量 UDF 调用模式下随附函数指针的期望调用模式。

```
_start_extfn(if supplied)
_evaluate_extfn (repeated 0 to numerous times)
_finish_extfn(if supplied)
```

另请参见

- 标量和集合 UDF 回调函数 （第 80 页）
- 集合 UDF 调用模式 （第 82 页）

集合 UDF 调用模式

与标量调用模式相比，用户提供的集合 UDF 函数的调用模式更为复杂， 也更加多变。

使用下列表定义的示例：

```
create table t (a int, b int, c int)
insert into t values (1, 1, 1)
insert into t values (2, 1, 1)
insert into t values (3, 1, 1)
insert into t values (4, 2, 1)
insert into t values (5, 2, 1)
insert into t values (6, 2, 1)
```

使用下列缩写形式：

RR = a_v3_extfn_aggregate_context. _result_row_offset_from_start_of_partition — 此值表示在其中计算值的当前分区中当前的行号。该值在窗口集合期间设置，主要用于未受限制的窗口的求值步骤；它在所有求值调用上均可用。

Sybase IQ 是一种多用户应用程序。许多用户都能同时执行同一 UDF。某些 OLAP 查询会在同一次查询过程中多次执行 UDF，有时会同时执行。

另请参见

- 标量和集合 UDF 回调函数 （第 80 页）
- 标量 UDF 调用模式 （第 82 页）

简单拆组集合

简单拆组集合调用模式对所有行的输入值进行总计，并产生一个结果。

查询

```
select my_sum(a) from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
```

```
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 21
_finish_extfn(cntxt)
```

结果

```
my_sum(a)
21
```

简单分组集合

简单分组集合调用模式对组中所有行的输入值进行总计，并产生一个结果。**_reset_extfn** 标识组的开头。

查询

```
select b, my_sum(a) from t group by b order by b
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns  6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

结果

```
b,   my_sum(a)
1,   6
2,   15
```

通过未受限制的窗口实现 OLAP 样式集合调用模式

对“b”分区会创建与对“b”分组相同的分区。未受限制的窗口会导致为分区的每行求“a”的值。由于这是未受限制的查询，在求值循环前会首先将所有值填充到 UDF。_window_has_unbounded_preceding 和 _window_has_unbounded_following 的上下文指示符设置为 1

查询

```
select b, my_sum(a) over (partition by b rows between
unbounded preceding and
```

```
unbounded following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=1
_next_value_extfn(cntxt, args)    input a=2
_next_value_extfn(cntxt, args)    input a=3
_evaluate_extfn(cntxt, args)         rr=1  returns 6
_evaluate_extfn(cntxt, args)         rr=2  returns 6
_evaluate_extfn(cntxt, args)      rr=3  returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=4
_next_value_extfn(cntxt, args)    input a=5
_next_value_extfn(cntxt, args)    input a=6
_evaluate_extfn(cntxt, args)      rr=1  returns 15
_evaluate_extfn(cntxt, args)      rr=2  returns 15
_evaluate_extfn(cntxt, args)      rr=3  returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  6
1,  6
1,  6
2,  15
2,  15
2,  15
```

OLAP 样式的未优化累积窗口集合

如果未提供 _evaluate_cumulative_extfn，则会通过这种调用模式计算此累计总和，而这样的效率低于使用 _evaluate_cumulative_extfn。

查询

```
select b, my_sum(a) over (partition by b
rows between unbounded preceding and current row)
from t
order by b
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)     -- input a=1
_evaluate_extfn(cntxt, args)    -- returns 1
_next_value_extfn(cntxt, args)  -- input a=2
_evaluate_extfn(cntxt, args)    -- returns 3
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=4
```

```
_evaluate_extfn(cntxt, args)    -- returns 4
_next_value_extfn(cntxt, args)  -- input a=5
_evaluate_extfn(cntxt, args)    -- returns 9
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP 样式的已优化累积窗口集合

如果提供了 _evaluate_cumulative_extfn，此累计总和会在如下位置求值：next_value/求值序列组合到每个分区中每一行的单个 _evaluate_cumulative_extfn 调用中。

查询

```
select b, my_sum(a) over (partition by b rows between unbounded
preceding and current row)
from t
order by b
```

调用模式

```
_start_extnfn(cntxt)
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=1 returns 1
_evaluate_cumulative_extfn(cntxt, args) -- input a=2 returns 3
_evaluate_cumulative_extfn(cntxt, args) -- input a=3 returns 6
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=4 returns 4
_evaluate_cumulative_extfn(cntxt, args) -- input a=5 returns 9
_evaluate_cumulative_extfn(cntxt, args) -- input a=6 returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP 样式的未优化移动窗口集合

如果未提供 _drop_value_extfn 函数，则会借此计算此移动窗口总和，但效率要比使用 _drop_value_extfn 低得多。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_evaluate_extfn(cntxt, args)           returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)         input a=2
_evaluate_extfn(cntxt, args)           returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args )        input a=2
_next_value_extfn(cntxt, args )        input a=3
_evaluate_extfn(cntxt, args)           returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
_evaluate_extfn(cntxt, args)              returns 4
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
_next_value_extfn(cntxt, args)         input a=5
_evaluate_extfn(cntxt, args)              returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=5
_next_value_extfn(cntxt, args)         input a=6
_evaluate_extfn(cntxt, args)              returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP 样式的已优化移动窗口集合

如果已提供 _drop_value_extfn 函数， 则会用这种调用模式计算此移动窗口总和， 而这样的效率高于使用 _drop_value_extfn。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_aggregate_extfn(cntxt, args)              -- returns  1
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  3
_drop_value_extfn(cntxt)                            -- input a=1
_next_value_extfn(cntxt, args)      -- input a=3
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      -- input a=4
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  4
_next_value_extfn(cntxt, args)      -- input a=5
_evaluate_aggregate_extfn(cntxt, args)                 -- returns 9
_drop_value_extfn(cntxt)                         -- input a=4
_next_value_extfn(cntxt, args)      -- input a=6
_evaluate_aggregate_extfn(cntxt, args)              -- returns  11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP 样式的未优化移动窗口跟随集合

如果未提供 _drop_value_extfn 函数，此移动窗口总和通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但被求值的行不是由下一值函数所给的最后一行。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
```

```
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)       input a=2
_evaluate_extfn(cntxt, args)         returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=1
_next_value_extfn(cntxt, args)       input a=2
_next_value_extfn(cntxt, args)       input a=3
_evaluate_extfn(cntxt, args)         returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=2
_next_value_extfn(cntxt, args)       input a=3
_evaluate_extfn(cntxt, args)         returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=4
_next_value_extfn(cntxt, args)       input a=5
_evaluate_extfn(cntxt, args)         returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=4
_next_value_extfn(cntxt, args)       input a=5
_next_value_extfn(cntxt, args)       input a=6
_evaluate_extfn(cntxt, args)         returns 15
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=5
_next_value_extfn(cntxt, args)       input a=6
_evaluate_extfn(cntxt, args)         returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  3
1,  6
1,  5
2,  9
2,  15
2,  11
```

OLAP 样式的已优化移动窗口跟随集合

如果提供了 _drop_value_extfn 函数，此移动窗口总和通过如下调用模式进行求值。同样，此情形与之前的移动窗口示例相似，但被求值的行不是由下一值函数所给的最后一行。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)       input a=2
_evaluate_extfn(cntxt, args)         returns 3
```

```
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)        returns 6
_dropvalue_extfn(cntxt)                                 input a=1
_evaluate_extfn(cntxt, args)        returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)        returns 9
_next_value_extfn(cntxt, args)      input a=6
_evaluate_extfn(cntxt, args)        returns 15
_dropvalue_extfn(cntxt)                                    input a=4
_evaluate_extfn(cntxt, args)        returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  3
1,  6
1,  5
2,  9
2,  15
2,  11
```

OLAP 样式的未优化移动窗口（不包括当前行）

假设 UDF my_sum 的工作方式与内置 SUM 类似。如果未提供 _drop_value_extfn 函数，此移动窗口计数通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但是当前行不是窗口构架的一个部分。

查询

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
```

```
_evaluate_extfn(cntxt, args)          returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

结果

```
b           my_sum(a)
----------- -------------
          1          NULL
          1             1
          1             3
          2             6
          2             9
          2            12
```

OLAP 样式的已优化移动窗口（不包括当前行）

如果提供了 _drop_value_extfn 函数，此移动窗口计数通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但是当前行不是窗口构架的一个部分。

查询

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_dropvalue_extfn(cntxt)                 input a=1
_next_value_extfn(cntxt, args)      input a=4
_evaluate_extfn(cntxt, args)          returns 9
_dropvalue_extfn(cntxt)                 input a=2
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

结果

```
b           my_sum(a)
---------- -------------
       1          NULL
       1             1
       1             3
       2             6
```

```
        2             9
        2            12
```

外部函数原型

可通过 Sybase IQ 安装目录子目录中名为 `extfnapiv3.h`（对于第 4 版 API，是 `extfnapiv4.h`）的头文件定义 API。此头文件用于处理 外部函数原型的与平台相关的功能。

要通知数据库服务器该库不是使用旧 API 写入的，请按如下方式提供一个函数：

```
uint32 extfn_use_new_api( )
```

此函数返回一个不带符号的 32 位整数。如果返回值为非零，则数据库服务器假定您用的是新 API。

如果无法通过 DLL 导出此函数，则数据库服务器 就会假定使用的是旧 API。如果使用的是新 API， 则返回值必须是 `extfnapi.v4h` 中所指定的 API 版本号。

每个库应该按如下所示实现并导出此函数：

```
unsigned int extfn_use_new_api(void)
    {
        return EXTFN_V4_API;
    }
```

此函数的存在及其返回的 `EXTFN_V4_API`，可向 Sybase IQ 引擎 表明库中含有写入本书所记述的新 API 的 UDF。

函数原型

函数的名称必须与 **CREATE PROCEDURE** 或 **CREATE FUNCTION** 语句中引用的相匹配。将函数声明为：

```
void function-name ( an_extfn_api *api, void *argument-handle )
```

函数必须返回 void，并且必须将用于传递参数的结构和由 SQL 过程提供的参数的句柄用作参数。

`an_extfn_api` 结构的形式如下：

```
typedef struct an_extfn_api {
short (SQL_CALLBACK *get_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    );
 short (SQL_CALLBACK *get_piece)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value,
                    a_sql_uint32    offset
                    );
 short (SQL_CALLBACK *set_value)(
                    void *          arg_handle,
```

```
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    short          append
                    );
void (SQL_CALLBACK *set_cancel)(
                    void *    arg_handle,
                    void *    cancel_handle
                    );
} an_extfn_api;
```

注意： **get_piece** 回调在第 3 版和第 4 版标量和集合 UDF 中都有效。对于第 4 版表 UDF 和 TPF，请改用 Blob (`a_v4_extfn_blob`) 和 Blob 输入流 (`a_v4_extfn_blob_istream`) 结构。

an_extfn_value 结构的格式如下：

```
typedef struct an_extfn_value {
void *                data;
  a_sql_uint32         piece_len;
  union {
  a_sql_uint32    total_len;
  a_sql_uint32    remain_len;
} len;
  a_sql_data_type        type;
} an_extfn_value;
```

注意
对 OUT 参数调用 `get_value` 会返回 参数的数据类型，还会以空值的形式返回数据。

任何给定参数的 `get_piece` 函数 只能紧接着相同参数的 get_value 函数调用。

要返回空值，请在 `an_extfn_value` 中将数据设置为 NULL。

`set_value` 的 append 字段决定着所提供的数据是 替换 (false) 现有数据，还是附加到 (true) 现有数据之后。 对于同一参数，必须先通过 append=FALSE 调用 `set_value`，然后再用 append=TRUE 对其进行调用。 对于定长数据类型，忽略 append 字段。

头文件本身包含额外的注释。

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）

表 UDF 和 TPF

表 UDF 是外部用户定义的 C、C++ 或 Java 表函数。表 UDF 不同于标量和集合 UDF，会生成多个行作为输出。SQL 查询以表表达式的形式消耗表 UDF 的输出。

标量和集合 UDF 能使用第 3 版或第 4 版 API，但表 UDF 只能使用第 4 版 API。

可用 **CREATE PROCEDURE** 语句声明表 UDF SQL 函数。标量和集合 UDF 采用 **CREATE FUNCTION** 语句。

表参数化函数 (TPF) 是增强型表 UDF，可用标量值或行组作为输入。

另请参见

- 表参数化函数 （第 128 页）
- 声明和定义用户定义的标量函数 （第 32 页）
- 声明和定义集合 UDF （第 46 页）
- 学习路线图：外部 C 和 C++ UDF 类型 （第 6 页）
- 创建 Java 表 UDF （第 334 页）

用户角色

有两类用户可以使用表 UDF：UDF 开发人员和 SQL 分析师。

- **UDF 开发人员** – 用 C 或 C++ 开发表 UDF。
- **SQL 分析师** – 开发并分析在 **FROM** 子句中引用表表达式的 SQL 查询。表表达式是表 UDF 生成的一组行。

另请参见

- 表 UDF 开发人员的学习路线图 （第 93 页）
- SQL 分析师学习路线图 （第 94 页）

表 UDF 开发人员的学习路线图

用有注释的示例了解如何开发 C 或 C++ 表 UDF。开发工作完成后，SQL 分析师就能在 SQL 查询中引用开发出来的 UDF 了。

此路线图的前提是：

- 计算机中有 C 或 C++ 开发环境。
- 熟悉标准编程方法。

任务	请参见
熟悉表 UDF 和 TPF 限制。	表 UDF 限制 （第 95 页）
创建表 UDF。	开发表 UDF （第 98 页）
（可选）定义库版本验证器，以便进行分布式查询处理 (DQP)。	库版本（extfn_get_library_version） （第 17 页） 库版本兼容性 (extfn_check_version_compatibility) （第 17 页）
编译并链接源代码。	编译并链接源代码以构建动态链接库 （第 19 页）
通过 **CREATE PROCEDURE** 语句对服务器声明 UDF。以命令的形式写入并执行这些语句，或通过 Sybase Central 或 Sybase Control Center 发出 **CREATE** 语句。	SQL 分析师学习路线图 （第 94 页）

SQL 分析师学习路线图

在 SQL 查询中引用 C 或 C++ 表 UDF。

任务	请参见：
向 UDF 开发人员索取 `.dll` 或 `.so` 文件（如 `myudf.dll`）。 将 `.dll` 文件放入 `bin64` 目录，将 `.so` 文件放入 `lib64` 或 `LD_LIBRARY_PATH` 目录。	不适用。

定义 **CREATE PROCEDURE** 语句，从而引用 `.dll` 文件和回调函数。
例如：
`CREATE PROCEDURE my_udf( IN num_row INT )`
`RESULT( id INT )`
`EXTERNAL NAME 'udf_rg_proc@myudf.dll'`
	CREATE PROCEDURE 语句（表 UDF） （第 160 页）

通过 UDF 选择行。 例如： `SELECT * FROM my_udf(5)`	SELECT 语句 （第 175 页） FROM 子句 （第 169 页）

另请参见

- 针对表 UDF 和 TPF 查询的 SQL 参考 （第 158 页）

表 UDF 限制

表 UDF 和 TPF 有一些限制。

- IQ_IDA 许可证用户仅可对 Multiplex 读取程序节点运行表 UDF 和 TPF。
- 不允许对任何外部过程使用 **TEMPORARY PROCEDURE** 子句。创建时尝试创建临时外部过程会产生错误。
- 不允许使用 **NO RESULT SET** 子句。表 UDF 和 TPF 必须特意声明其结果的内容。
- 如果已指定可选的 **DYNAMIC RESULT SETS integer-expression** 子句，则必须将该值设置为 1。表 UDF 和 TPF 不返回多个结果集。
- 表 UDF 或 TPF 不能在 **CALL** SQL 语句或已嵌入 **EXEC** 的 SQL 语句中引用。表 UDF 或 TPF 只能在 SQL 语句的 **FROM** 子句中引用。
- 不能将 **LANGUAGE** 子句用于 UDF 或 TPF。如果有 **LANGUAGE** 子句，则执行时会报告语法错误。
- 仅可将关键字 **IN** 用于 **parameter** 子句；对于表 UDF 或 TPF，关键字 **INOUT** 和 **OUT** 不受支持。
- **EXTERNAL NAME** 子句与标量和集合 UDF 的语法相同。

开始使用

请先熟悉示例文件、概念和限制，然后再开发表 UDF 和 TPF。

示例文件

示例表 UDF 文件安装于服务器。当定义您自己的表 UDF 时，请使用示例作为模型。

示例文件位于：

- `%ALLUSERSPROFILE%\samples\udf` (Windows)
- `$IQDIR15/samples/udf` (UNIX)

文件	描述
`apache_log_reader.cxx`	实现一个表 UDF，该 UDF 读取 Apache 日志文件，并以表的格式显示文件中的各行。此 UDF 举实例描述了如何通过 UDF 使 SQL 查询写入程序能够实时使用计算机生成的数据。
`build.sh / build.bat`	一个脚本，用于编译并链接 `samples/udf` 目录中的示例标量、UDF 集合、表 UDF 和 TPF。
`my_md5.cxx`	一个简单的确定性标量 UDF，用于计算输入文件（LOB 二进制参数）的 MD5 散列值。

文件	描述
tpf_agg.cxx	使用输入表中的行，对输入数据进行聚合，并将各行数据返回给服务器。
tpf_blob.cxx	实现一个 TPF。如果指定字符或数字的数量为偶数，则该 TPF 从输入表中读取 LOB 数据，并将数据传至结果集。此 TPF 描述了如何读取 LOB 数据，以及用户如何将 LOB 数据类型传至结果集。
tpf_dt.cxx	
tpf_filt.cxx	描述了如何使用 TPF 对行进行过滤。本例使用调用程序提供的行块，并将其传至输入表参数。输入表模式必须与该函数的输出结果集相匹配。
tpf_oby.cxx	描述 TPF 如何 生成有序输出 ，并将其传递。
tpf_pby.cxx	描述 TPF 如何 生成分区输出 ，并将其传递。
tpf_rg_1.cxx	类似于表 UDF 示例 udf_rg_2.cxx.。它根据输入参数生成数据行。
tpf_rg_2.cxx	基于 tpf_rg_1.cxx 中的示例而构建，但是使用 fetch_into 而非 fetch_block 从输入表中提取行。
udf_main.cxx	此文件与所有示例相链接，并且包括一组通用的 v4 API 所需的入口点。这样您就可以重用代码，而不用在每个示例中都包含它。
udf_rg_1.cxx	一个简单的表 UDF，可以生成整数数据行。
udf_rg_2.cxx	一个简单的表 UDF，可以生成整数数据行。该整数数据行使用 describes 以确保在 SQL 中定义的模式与 UDF 的实现相互匹配。它还描述了一些优化程序的属性。
udf_rg_3.cxx	一个简单的表 UDF，使用 _fetch_block 提取方法生成 100 个整数数据块。
udf_utils.cxx	一组实用程序函数及宏，对于 UDF/TPF 开发者十分有用。这些示例依赖于本文件中的项目。
udf_utils.h	一组实用程序函数及宏，对于 UDF/TPF 开发者十分有用。这些示例依赖于本文件中的项目。

了解生产者和消耗程序

服务器和 UDF 交换数据行时会建立生产者和消耗程序关系。

生产和消耗是指表行数据。生产者可以生成表行；消耗程序可以消耗表行。

对于每个相符的查询行，服务器都会执行一次标量和集合 UDF。这些 UDF 消耗标量输入参数，并且生成并返回一个标量参数。这种数据交换通过 get_value() 和 set_value() API 在 evaluate 方法的执行过程中进行。

但是，如果 UDF 必须生成或消耗表，则标量生成和消耗是一种低效数据交换方式。生成表的表 UDF 和消耗表的 TPF，都采用第 4 版 API 的 `row block` 数据结构。通过行块可进行批量行和列数据交换。 行块由生产者填充，由消耗程序读取。

下例中的表 UDF `my_table_udf()` 是数据生产者。服务器 Sybase IQ 是数据消耗程序：

```
SELECT * FROM my_table_udf()
```

一般而言，表 UDF 始终都是数据生产者。但服务器可能不总是消耗程序：

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table_udf() ) )
```

外部 TPF `my_tpf()` 是通过 **SELECT * from my_table_udf()** 指定的表输入参数的消耗程序。 Sybase IQ 是 `my_tpf()` TPF 生成的表的消耗程序。因此，TPF 既可以是 消耗程序，也可以是生产者。

TPF 不必消耗表 UDF 产生的数据。下例中的 TPF 消耗由内部查询生成的表数据，而内部查询则由 Sybase IQ 服务器生成：

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table where my_table.c1
< 10 ) )
```

因此，Sybase IQ 在 TPF 中既可以是表数据消耗程序，也可以是表数据生产者。

在第 4 版 API 中，行块用于定义对其生成数据和消耗其中数据的内存区域。一般而言，行块的布局 在概念上与表的行和列格式一致；行块由一些行组成，而每行则由一些列组成。 生产者或消耗程序必须分配行块，到时还必须释放行块。

行和列都有各自的仅适用于其本身的具体属性。 例如，行有状态标志，指示行是否存在。TPF 可通过这种标志更改行状态， 不必移动列数据。列有空值掩码，指示数据值是否为空。行块还有一些其它属性， 如最大行数、当前行数等。如果 UDF 要为处理大量的行而 创建行块，但按需生成较少的行，则可使用这些行块属性。

行的消耗过程由下列某个获取 API 执行：

- `fetch_into`
- `fetch_block`

消耗程序分配行块，以及向生产者传递行块时，都会调用 `fetch_into`。然后会请求生产者 尽可能多填充一些行，最多可达最大行数。消耗程序要让生产者分配行块时会调用 `fetch_block`。`Fetch_block` 效率很高（如果要开发可以过滤多行数据的 TPF）。服务器（消耗程序）通过 `fetch_into` API 分配行块并由 TPF 获取数据。然后 TPF 就能通过 `fetch_block` API 向 输入参数传递同一行块。

另请参见

- 行块数据交换 （第 122 页）

开发表 UDF

开发表 UDF 一般步骤包括：确定输入和输出、声明 v4 库、定义 a_v4_extfn_proc 描述符、定义库入口点函数、定义服务器获取行信息的方法、实现具有 a_v4_extfn_proc 结构的函数、以及实现具有 a_v4_extfn_table_func 结构的函数。

1. 确定表 UDF 的输入和输出。

   输入由过程所接受的参数定义，而输出则由过程的 **RESULT** 子句的声明方法定义。在 SQL 中表 UDF 的声明与其实现相互分离。这意味着，表 UDF 的特定实现可能与具体声明相绑定。当开发表 UDF 时，请确保其实现与声明彼此匹配。

2. 声明库为 v4 库。

   若要使 Sybase IQ 能够将库识别为 v4 库，库必须包含 extfnapiv4.h 头文件，它位于您的 Sybase IQ 的安装目录下的子目录中。

   该头文件定义了 v4 API 的功能和函数，是 v3 API 的超集；extfnapiv4.h 包含 extfnapiv3.h。

   若要创建表 UDF 或 TPF，库必须提供 extfn_use_new_api() 入口点。对于 v4 库，extfn_use_new_api() 必须返回 EXTFN_V4_API。

3. 定义 a_v4_extfn_proc 描述符。

   当开发 v4 表 UDF 或者 TPF 时，库必须声明服务器可以调用的函数。

   创建 a_v4_extfn_proc 类型的变量，将该结构的每个成员设置为表 UDF（实现函数）中的函数的地址。服务器可以通过库的入口指针使用该变量中的信息。并非 a_v4_extfn_proc 的所有成员都需要用到，其中有两个保留字段必须设置为空。

   当您开发自己的函数时，请使用此描述符函数作为模型。

```
static a_v4_extfn_proc udf_proc_descriptor =
{
    udf_proc_start,       // optional
    udf_proc_finish,      // optional
    udf_proc_evaluate,    // required
    udf_proc_describe,    // required
    udf_proc_enter_state,// optional
    udf_proc_leave_state,// optional
    NULL,                 // Reserved: must be NULL
    NULL                  // Reserved: must be NULL
};
```

4. 定义库入口点函数。

   表 UDF 库必须提供能够返回 a_v4_extfn_proc 描述符指针的函数入口点。其与第三步中所述之描述符相同。

此回调函数是库所需的主要入口点。

当您开发自己的库入口点时，请使用此函数作为模型：

```
extern "C"
a_v4_extfn_proc * SQL_CALLBACK udf_rg_proc()
/*****************************************/
{
    return &udf_proc_descriptor;
}
```

5. 定义服务器从表 UDF 中获取行信息的方法。

   当开发 v4 表 UDF 或者 TPF 时，库必须声明如何将行信息传至服务器。

   创建 `a_v4_extfn_table_func` 类型的变量，将该结构的每个成员设置为表 UDF（实现函数）中的函数的地址。运行期间，变量中的信息对服务器可用。

   并非 `a_v4_extfn_table_func` 的所有成员都需要用到，其中有两个保留字段必须设置为空。

   当您开发自己的表 UDF 时，请使用此描述符作为模型。

```
{
    udf_table_func_open,        // required
    udf_table_func_fetch_into,  // one of fetch_into or
fetch_block required
    udf_table_func_fetch_block, // one of fetch_into or
fetch_block required
    udf_table_func_rewind,      // optional
    udf_table_func_close,       // required
    NULL,                       // Reserved: must be NULL
    NULL                        // Reserved: must be NULL
};
```

   开始执行时，服务器调用 `a_v4_extfn_proc` 的 `_evaluate_extfn` 函数，从而令表 UDF 得以将其正予实现的表函数告知服务器。若要实现于此，表 UDF 必须为给予服务器的 `a_v4_extfn_table` 创建一个实例。此结构包含一个指向 `a_v4_extfn_table_func` 描述符的指针，以及结果集中的列的数量。

   当您开发自己的表 UDF 时，请使用此描述符作为模型。

```
static a_v4_extfn_table_udf_rg_table = {
    &udf_table_funcs, // Table function descriptor
    1                 // number_of_columns
};
```

6. 实现具有 `a_v4_extfn_proc` 结构的函数。

   表 UDF 必须对每个 `a_v4_extfn_proc` 函数（声明于步骤 3 中的 `a_v4_extfn_proc` 描述符之中）予以实现。

7. 实现具有 `a_v4_extfn_table_func` 结构的函数。

   表 UDF 必须对每个 `a_v4_extfn_table_func` 函数（声明于步骤 5 中的 `a_v4_extfn_table_func` 描述符之中）予以实现。

另请参见

- 标量和集合 UDF 调用模式 （第 80 页）
- udf_rg_2 （第 105 页）
- udf_rg_3 （第 109 页）
- 实现示例表 UDF udf_rg_1 （第 100 页）
- 表 UDF 实现示例 （第 100 页）
- 外部函数 (a_v4_extfn_proc) （第 270 页）
- 表函数 (a_v4_extfn_table_func) （第 298 页）
- _evaluate_extfn （第 271 页）

表 UDF 实现示例

实现示例从简单的表 UDF 开始，随后其复杂度不断增加。

表 UDF 实现示例包含于相同目录中。这些示例从简单的表 UDF 开始，其后随着示例的改进，其复杂度不断增加、功能不断完善。

在名为 `libv4apiex` 的预编译动态库中该示例可用。（库的扩展名与平台相关。）此库与定义于 `udf_main.cxx` 中的函数相链接，其中包括例如 **extfn_use_new_api** 的库级函数。请将 `libv4apiex` 置于服务器可以读取的目录之中。

另请参见

- 运行 udf_rg_1.cxx 中的示例表 UDF （第 105 页）
- 运行 udf_rg_ 2 .cxx 中的示例表 UDF （第 108 页）
- 运行 udf_rg_3.cxx 中的示例表 UDF （第 112 页）

实现示例表 UDF udf_rg_1

名为 **udf_rg_1** 的示例表 UDF 描述了 v4 表 UDF 是如何生成 n 行数据的。表 UDF 的实现在 `udf_rg_1.cxx` 中的示例目录之中。

1. 确定表 UDF 的输入和输出。

   此示例根据输入参数的值生成 *n* 行数据。输入为单个整数参数，输出为由 `integer` 类型的单列所组成的多个行。

   定义此过程所需的 **CREATE PROCEDURE** 语句为：

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_1( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME ‘udf_rg_1@libv4apiex’
   ```

2. 声明库为 v4 库。

   在此示例中，`udf_rg_1.cxx` 包括头文件 `extfnapiv4.h` ：

   ```
   #include “extfnapiv4.h”
   ```

   函数导出定义于 `udf_main.cxx`，以通知服务器此库包含有 v4 表 UDF：

```
a_sql_uint32 SQL_CALLBACK extfn_use_new_api( void )
/************************************************/
{
    return EXTFN_V4_API;
}
```

3. 定义 a_v4_extfn_proc 描述符。

   在 udf_rg_1.cxx 中声明了必要的描述符：

```
static a_v4_extfn_proc udf_rg_descriptor =
{
    NULL,             // _start_extfn
    NULL,             // _finish_extfn
    udf_rg_evaluate, // _evaluate_extfn
    udf_rg_describe, // _describe_extfn
    NULL,             // _leave_state_extfn
    NULL,             // _enter_state_extfn
    NULL,             // Reserved: must be NULL
    NULL              // Reserved: must be NULL
};
```

4. 定义库入口点函数。

   此回调函数声明了主要的入口点函数。它只是将指针返回于 a_v4_proc_descriptor 的变量 *udf_rg_descriptor*。

```
extern "C"
a_v4_extfn_proc * SQL_CALLBACK udf_rg_1_proc()
/********************************************/
{
    return &udf_rg_descriptor;
}
```

5. 定义服务器从表 UDF 中获取行信息的方法。

   对 a_v4_extfn_table_func 描述符作了声明，该描述符用于告知服务器从表 UDF 中检索行数据的方法。

```
static a_v4_extfn_table_func udf_rg_table_funcs =
{
    udf_rg_open,       // _open_extfn
    udf_rg_fetch_into, // _fetch_into_extfn
    NULL,              // _fetch_block_extfn
    NULL,              // _rewind_extfn
    udf_rg_close,      // _close_extfn
    NULL,              // Reserved: must be NULL
    NULL               // Reserved: must be NULL
};
```

   在此示例中，_fetch_into_extfn 函数将行数据传至服务器。这是一种最容易理解和实现的数据传输方法。本文档将数据传输方法引用为*行块数据交换*。有以下两种行块数据交换函数：_fetch_into_extfn 和 _fetch_block_extfn。

   运行期间，当调用 _evaluate_extfn 函数时，UDF 将通过设置结果集参数来发布表函数描述符。若要实现于此，UDF 必须为 a_v4_extfn_table 创建实例：

```
static a_v4_extfn_table udf_rg_table = {
    &udf_rg_table_funcs,    // Table function descriptor
    1                     // number_of_columns
};
```

此结构包含一个指向 udf_rg_table_funcs 结构的指针，以及结果集中的列的数量。此表 UDF 在其结果集中生成一个单列。

6. 实现具有 a_v4_extfn_proc 结构的函数。

   在此示例中，所需的函数 _describe_extfn 不执行任何操作。其他示例描述了表 UDF 使用 describe 函数的方法：

```
static void UDF_CALLBACK udf_rg_describe(
       a_v4_extfn_proc_context *ctx )
/
*****************************************************************
/
{
   // This required function is not needed in this simple example.
}
```

   _evaluate_extfn 方法将获取 UDF 结果集的相关信息发送至服务器。通过调用具有 a_v4_extfn_proc_context 结构（参数 0）的 set_value 方法来完成此操作。参数 0 代表返回值，对于表 UDF 而言，该返回值为 DT_EXTFN_TABLE 类型的数据。此方法构建了 an_extfn_value 结构，将数据类型设置为 DT_EXTFN_TABLE，并将其值指针指向第 5 步中所创建的 a_v4_extfn_table 对象。对于表 UDF，该类型必须始终为 DT_EXTFN_TABLE。

```
static void UDF_CALLBACK udf_rg_evaluate(
         a_v4_extfn_proc_context *ctx,
        void *args_handle )
/*********************************/
{
    an_extfn_value    result_table = { &udf_rg_table,
                      sizeof( udf_rg_table ),
                      sizeof( udf_rg_table ),
                      DT_EXTFN_TABLE };

    // Tell the server what functions table functions are being
    // implemented and how many columns are in our result set.
    ctx->set_value( args_handle, 0, &result_table );
}
```

7. 实现具有 a_v4_extfn_table_func 结构的函数。

   在此示例中，表 UDF 需要读取传入的参数（其中包含生成行的数量），并对随后将要使用的信息进行缓存。因为对于参数具有的每个新值都将调用 _open_extfn 方法，所以这是一个获取该信息的合适位置。

   除了生成行的总数以外，表 UDF 还必须记住要生成的下一行。当服务器开始从表 UDF 提取行的时候，可能需要重复调用 _fetch_into_extfn 方法。这意味着表 UDF 必须记住已生成的最后一行。

以下结构创建于 `udf_rg_1.cxx`，以包含调用之间的状态信息：

```
struct udf_rg_state {
    a_sql_int32    next_row;  // The next row to produce
    a_sql_int32    max_row;   // The number of rows to generate.
};
```

打开方法首先使用具有 `a_v4_proc_context` 结构的 `get_value` 方法来读取参数 1 的值。使用具有 `a_v4_proc_context` 结构的 `alloc` 函数来配置 `udf_rg_state` 实例。表 UDF 应该使用内存管理函数（`alloc` 和 `free`）对具有 `a_v4_proc_context` 结构的数据进行管理，以便尽可能地管理其内存空间。然后将状态对象保存至 `a_v4_proc_context` 结构的 *user_data* 字段。执行完毕后，该字段中存储的内容可供表 UDF 使用。

```
static short UDF_CALLBACK udf_rg_open(
          a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value     value;
    udf_rg_state *    state = NULL;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

    // Send an error to the client if we could not get the value.
    tctx->proc_context->set_error(
        tctx->proc_context,
        17001,
        "Error: Could not get the value of parameter 1" );

    return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (udf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                   sizeof( udf_rg_state ) );

    // Start generating at row zero.
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = *(a_sql_int32 *)value.data;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

`_fetch_info_extfn` 方法返回行数据至服务器。该方法将被重复调用，直至其返回 false。对于此示例，表 UDF 从 `a_v4_extfn_proc_context` 对象的 *user_data* 字段中检索状态信息，以确定要生成的下一行以及要生成的总行数。该方法可生成的行数至多不超过所传入的行块结构中指定的最大行数。

对于此示例，表 UDF 生成 INT 类型的单列。它将状态信息中所存储的 *next_row* 数据复制到第一列的数据指针中。每次循环时，表 UDF 把一个新值复制到数据指针中，当生成的行数达到最大值、或者行块已满时，则退出循环。

```
static short UDF_CALLBACK udf_rg_fetch_into(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block *rb)
/*************************************/
{
    udf_rg_state    *state = (udf_rg_state *)tctx->user_data;

    // Because we are implementing fetch_into, the server has provided
    // us with a row block. We need to inform the server how many rows
    // this call to _fetch_into has produced.
    rb->num_rows = 0;

    // The server provided row block structure contains a max_rows
    // field.  This field is the maximum number of rows that this row
    // block can handle.  We can not exceed this number.  We will also
    // stop producing rows when we have produced the number of rows
    // required as per the max_row in the state.
    while( rb->num_rows < rb->max_rows && state->next_row < state->max_row ) {

           // Get the current row from the row block data.
           a_v4_extfn_row      &row = rb->row_data[ rb->num_rows ];

           // Get the column data for the current row.
           a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

           // Copy the integer value for the next row to generate
           // into the column data for the current row.
           memcpy( col0.data, &state->next_row, col0.max_piece_len );

           state->next_row++;
           rb->num_rows++;
    }

    // If we produced any rows, return true.
    return( rb->num_rows > 0 );
}
```

当所有行提取完毕后，表 UDF 为参数的每个新值调用一次 _close_extfn 方法。也就是说，每次调用 _open_extfn 之后，都将调用 _close_extfn。在此示例中，当调用 _open_extfn 时，表·UDF 必须释放内存，通过从 a_v4_extfn_proc_context 对象的 *user_data* 字段中检索状态信息，而后调用 free 方法予以实现。

```
static short UDF_CALLBACK udf_rg_close(
          a_v4_extfn_table_context *tctx)
/***********************************/
{
    udf_rg_state *    state = NULL;

    // Retrieve the state that was saved in user_data
    state = (udf_rg_state *)tctx->user_data;

    // Free the memory for the state using the
a_v4_extfn_proc_context
```

```
    // function free.
    tctx->proc_context->free( tctx->proc_context, state );
    tctx->user_data = NULL;

    return 1;
}
```

另请参见

- udf_rg_2 （第 105 页）
- udf_rg_3 （第 109 页）
- 行块数据交换 （第 122 页）
- Describe API （第 193 页）
- _evaluate_extfn （第 271 页）
- fetch_into （第 292 页）
- 表 (a_v4_extfn_table) （第 290 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- _open_extfn （第 300 页）
- _close_extfn （第 302 页）

运行 udf_rg_1.cxx 中的示例表 UDF

`udf_rg_1` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `udf_rg_1.cxx` 中的 `samples` 目录中。

1. 请将 `libv4apiex` 库至于服务器能够读取的目录之中。
2. 若要向服务器声明表 UDF，请发出以下命令：

```
CREATE PROCEDURE udf_rg_1( IN num INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_1@libv4apiex'
```

3. 从表 UDF 中选择行：

```
SELECT * FROM udf_rg_1( 5 );
```

udf_rg_2

示例表 UDF `udf_rg_2` 基于 `udf_rg_1.cxx` 中的示例而构建，且具有相同的行为。该过程名为 **udf_rg_2**，它的实现在 `udf_rg_2.cxx` 中的示例目录中。

表 UDF **udf_rg_2** 在 `a_v4_extfn_proc` 描述符中提供了 `_describe_extfn` 的替代实现方法。

```
static void UDF_CALLBACK udf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this table udf.
```

```
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

    a_sql_data_type        type     = DT_NOTYPE;
    a_sql_uint32        num_cols    = 0;
    a_sql_uint32        num_parms    = 0;


    // Inform the server that we support a single input
    // parameter.
    num_parms = 1;
    desc_rc = ctx->describe_udf_set
        ( ctx,
          EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
          &num_parms,
          sizeof( num_parms ) );

    // Checks the return code and sets an error if the
    // describe was unsuccessful for any reason.
    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of parameter 1 is int.
    type = DT_INT;
    desc_rc = ctx->describe_parameter_set
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_TYPE,
          &type,
          sizeof( type ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the number of columns in our
    // result set is 1.
    num_cols = 1;
    desc_rc = ctx->describe_parameter_set
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
          &num_cols,
          sizeof( num_cols ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of column 1 in our
    // result set is int.
    type = DT_INT;
    desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_TYPE,
          &type,
          sizeof( type ) );
```

```
    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    // The following describes will inform the server of various
    // optimizer related characteristics.
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    an_extfn_value        p1_value;
    a_v4_extfn_estimate    num_rows;

    // If the value of parameter 1 was constant, then we can
    // inform the server how many distinct values will be.
    desc_rc = ctx->describe_parameter_get
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
          &p1_value,
          sizeof( p1_value ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    if( desc_rc != EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE ) {

        // Inform the server that this UDF will produce n rows.
        num_rows.value = *(a_sql_int32 *)p1_value.data;
        num_rows.confidence = 1;
        desc_rc = ctx->describe_parameter_set
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that this UDF will produce n distinct
        // values for column 1 of its result set.
        desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    }
}
```

该 `describe` 方法有两种主要功能：

- 通知服务器其所支持的模式。
- 通知服务器某些已知的优化属性。

在若干状态中将调用 `describe` 函数。然而，并非在每个状态中 `describe` 属性皆可用。`describe` 方法确定了某种状态，在此状态中，通过检查 `a_v4_extfn_proc` 结构中的 *current_state* 变量对该方法予以执行。

当处于标注状态时，表 UDF **udf_rg_2** 将通知服务器，它具有一个 `INTEGER` 类型的参数，并且其结果集包含一个 `INTEGER` 类型的单列。通过设置如下属性予以实现：

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS
- EXTFNAPIV4_DESCRIBE_PARM_TYPE
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS
- EXTFNAPIV4_DESCRIBE_COL_TYPE

如果 `describe` 方法中所设置的信息同 **CREATE PROCEDURE** 语句的过程定义不匹配，则 `describe_parameter_set` 和 `describe_column_set` 方法将返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE`。然后，describe 方法将设置一个错误参数以表明此处的客户端不匹配。

此示例使用定义于 `udf_utils.h` 中的 `UDF_CHECK_DESCRIBE` 宏以检查 describe 的返回值，并当执行失败时，设置一个错误参数。

在优化期间，表 UDF **udf_rg_2** 将通知服务器其所返回的行的数量与参数 1 中所指示的行数相同。因为生成的行在递增，所以该值也是唯一的。在优化期间，只能使用具有常量值的参数。使用 `describe` 属性 `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE` 以获取常量参数的值。当表 UDF 确定了属性值可用时，**udf_rg_2** 将会把 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 和 `EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` 设置为获取的值。

另请参见

- udf_rg_3 （第 109 页）
- 实现示例表 UDF udf_rg_1 （第 100 页）

运行 udf_rg_2.cxx 中的示例表 UDF

`udf_rg_2` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `udf_rg_2.cxx` 中的 `samples` 目录中。

1. 若要向服务器声明表 UDF，请发出以下命令：

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

2. 从表 UDF 中选择行：

```
SELECT * FROM udf_rg_2( 5 );
```

3. 若要查看 `describe` 的影响行为，请执行 **CREATE PROCEDURE** 语句，该语句所具有的模式不同于表 UDF 所发布的模式。例如：

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT, IN extra INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

4. 从表 UDF 中选择行：

```
SELECT * FROM udf_rg_2( 5 );
```

IQ 返回错误。

udf_rg_3

示例表 UDF **udf_rg_3** 基于 **udf_rg_2** 而构建，并且具有类似的行为。该过程名为 **udf_rg_3**，它的实现在 `udf_rg_3.cxx` 中的 `samples` 目录中。

表 UDF **udf_rg_3** 和 **udf_rg_2** 之间的行为差异在于 **udf_rg_3** 只能生成从 0 到 99 的 100 个唯一值，而后根据需要重复此序列。此表 UDF 提供了 `_start_extfn` 和 `_finish_extfn` 方法，并且包含 `_describe_extfn` 的修订版本以解释函数中的不同语义。

请使用 `fetch_block` 而非 `fetch_into` 方法，以使表 UDF 拥有行块结构并使用自己的数据布局。为描述此问题，在数组中对生成的数字作预分配。当执行读取操作时，表 UDF 将行块数据指针直接指向包含数据的内存地址，而不是将数据复制到服务器所提供的行块中，因而避免了额外的复制操作。

如下的辅助结构可以存储数字数组。此结构也可保存一个指向所分配行块的指针，以供释放行块之用。

```
#define MAX_ROWS 100
struct RowData {

    a_sql_int32             numbers[MAX_ROWS];
    a_sql_uint32            piece_len;
    a_v4_extfn_row_block * rows;

    void Init()
    {
           rows = NULL;
           piece_len = sizeof( a_sql_int32 );
           for( int i = 0; i < MAX_ROWS; i++ ) {
               numbers[i] = i;
           }
    }
};
```

当表 UDF 开始执行（执行完毕）时，将使用 `a_v4_extfn_proc_context` 中的 `_start_extfn`（`_finish_extfn`）方法分配此结构（释放此结构）。

```
static void UDF_CALLBACK udf_rg_start(
          a_v4_extfn_proc_context *ctx )
/***********************************/
{
```

```
    // The start_extfn method is a good place to allocate our row
    // data.  This method is called only once at the beginning of
    // execution.
    RowData *row_data = (RowData *)
          ctx->alloc( ctx, sizeof( RowData ) );
    row_data->Init();
    ctx->_user_data = row_data;
}
```

finish 方法执行两个功能:

- 释放 RowData 结构。
- 如果表 UDF 在执行读取操作时遇到错误且无法销毁行块，则销毁该行块。

```
static void UDF_CALLBACK udf_rg_finish(
          a_v4_extfn_proc_context *ctx )
/*************************************/
{
    if( ctx->_user_data != NULL ) {

           RowData *row_data = (RowData *)ctx->_user_data;


           // If rows is non-null here, it means an error occurred and
           // fetch_block did not complete.
           if( row_data->rows != NULL ) {
               DestroyRowBlock( ctx, row_data->rows, 0, false );
           }

           ctx->free( ctx, ctx->_user_data );
           ctx->_user_data = NULL;
    }
}
```

`fetch_block` 方法是:

```
static short UDF_CALLBACK udf_rg_fetch_block(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block **rows )
/*****************************************/
{
    udf_rg_state *    state         = (udf_rg_state*)tctx->user_data;
    RowData *        row_data    = (RowData *)tctx->proc_context->_user_data;

    // First call, we need to build the row block
    if( *rows == NULL ) {

          // This function will build a row block structure that holds
          // MAX_ROWS rows of data.  See udf_utils.cxx for details.
          *rows = BuildRowBlock( tctx->proc_context, 0, MAX_ROWS, false );

          // This pointer gets saved here because in some circumstances
          // when an error occurs, its possible we may have allocated
          // the rowblock structure but then never called back into
          // fetch_block to deallocate it.  In this case, when the finish
          // method is called, we will end up deallocating it there.
          row_data->rows = *rows;
    }
```

```
    (*rows)->num_rows = 0;

    // The row block we allocated contains a max_rows member that was
    // set to the macro MAX_ROWS (100 in this case). This field is the
    // maximum number of rows that this row block can handle.  We can
    // not exceed this number.  We will also stop producing rows when
    // we have produced the number of rows required as per the max_row
    // in the state.
    while( (*rows)->num_rows < (*rows)->max_rows &&
             state->next_row < state->max_row ) {

          a_v4_extfn_row      &row = (*rows)->row_data[ (*rows)->num_rows ];
          a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

          // Row generation here is a matter of pointing the data
          // pointer in the rowblock to our pre-allocated array of
          // integers that was stored in the proc_context.
          col0.data = &row_data->numbers[(*rows)->num_rows % MAX_ROWS];
          col0.max_piece_len = sizeof( a_sql_int32 );
          col0.piece_len = &row_data->piece_len;
          state->next_row++;
          (*rows)->num_rows++;
    }
    if( (*rows)->num_rows > 0 ) {
          return 1;
    } else {
         // When we are finished generating data, we can destroy the
         // row block structure.
         DestroyRowBlock( tctx->proc_context, *rows, 0, false );
         row_data->rows = NULL;
         return 0;
    }
}
```

首次调用此方法时，将使用 `udf_utils.cxx` 中的帮助函数 **BuildRowBlock** 分配一个行块。将指向该行块的指针保存于 `RowData` 结构中，以备后用。

通过将列数据的数据指针设置为序列（此前分配的数字数组）中下一个数字的地址，从而完成行的生成。列数据的 `piece_len` 指针也必须进行初始化，将其设置为 `piece_len`（`RowData` 的成员）的地址。因为行具有固定的数据长度，所以对于所有行此数字相同。

当最后一次调用读取操作、且不再生成新的数据时，将使用 `udf_utils.cxx` 中的帮助函数 **DestroyRowBlock** 销毁行块结构。

为了容纳此表 UDF（仅生成 100 个唯一值），请将 `EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` 设置为 100。源自 describe 方法的下述代码片段对此作了描述：

```
static void UDF_CALLBACK udf_rg_describe(
          a_v4_extfn_proc_context *ctx )
/*****************************************************************/
{
...
...
...

    a_v4_extfn_estimate distinct = {
```

```
            MAX_ROWS, 1.0
    };

    // Inform the server that this UDF will produce MAX_ROWS
    // distinct values for column 1 of its result set.
    desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
            &distinct,
            sizeof( distinct ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );
...
...
...

}
```

另请参见

- udf_rg_2 （第 105 页）
- 实现示例表 UDF udf_rg_1 （第 100 页）

运行 udf_rg_3.cxx 中的示例表 UDF

`udf_rg_3` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `udf_rg_3.cxx` 中的 `samples` 目录中。

1. 若要向服务器声明表 UDF，请发出以下命令：

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_3( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_3@libv4apiex'
   ```

2. 从表 UDF 中选择行：

   ```
   SELECT * FROM udf_rg_3( 200 );
   ```

   此查询为 `c1` 生成从 0 到 99、再从 0 到 99 的值。

apache_log_reader

示例表 UDF `apache_log_reader` 将 Apache 访问日志或者 Apache 错误日志的内容读取至表数据之中。它的实现在 `samples` 目录中的 `apache_log_reader.cxx` 文件中。

示例访问日志 (`apache_access.log`) 和示例错误日志 (`apache_error.log`) 位于 `samples` 目录中。

`apache_log_reader` 示例使用 `_open_extfn` 方法打开日志文件。然后，它使用 `_fetch_into_extfn` 方法读取数据，并将其解析为过程支持的模式。最后，它使用 `_close_extfn` 方法关闭日志文件。

另请参见

- _open_extfn （第 300 页）
- _fetch_into_extfn （第 300 页）
- _close_extfn （第 302 页）

运行 apache_log_reader.cxx 中的示例表 UDF

apache_log_reader 示例包含于名为 libv4apiex（扩展名因平台而异）的预编译动态库。它的实现在 apache_log_reader.cxx 中的 samples 目录中。

1. 若要向服务器声明表 UDF，请发出以下命令：

```
create procedure apache_log_reader
(
    in file_name  varchar(4000),
    in log_format varchar(32),
    in ip_padding varchar(1)
)
result
(
    ip_address  varchar(15),
    log_name    varchar(4000),
    user_name   varchar(4000),
    access_time datetime,
    time_zone   int,
    request     varchar(4000),
    response    int,
    bytes_sent  int,
    referer     varchar(4000),
    browser     varchar(4000),
    error_type  varchar(4000),
    error_msg   varchar(4000)
)
external name 'apache_log_reader@libv4apiex'
```

2. 从表 UDF 中选择行。当执行 SQL 查询时，请使用完整路径访问日志文件。

```
SELECT * FROM apache_log_reader( 'apache_access.log', 'access',
null );
```

udf_blob

示例表 UDF udf_blob 描述了表 UDF 或 TPF 使用 blob API 读取 LOB 输入参数的方法。

udf_blob 计算某个字母在第一个输入参数中的出现次数。参数 **1** 的数据类型可以为 LONG VARCHAR 或 VARCHAR(64)。如果其类型为 LONG VARCHAR，则表 UDF 使用 blob API 读取该值。如果其类型为 VARCHAR(64)，则使用 get_value 读取整个值。

此代码片段来自于 _open_extfn 方法，描述了使用 blob API 读取参数 **1** 的方法。

```
static short UDF_CALLBACK udf_blob_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
```

```
…
…
    a_v4_extfn_blob     *blob         = NULL;

    ret = tctx->proc_context->get_value( tctx->args_handle, 2,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
             "get_value for argument 2 failed",
             ret == 1,
             0 );

    letter_to_find = *(char *)value.data;

    ret = tctx->proc_context->get_value( tctx->args_handle, 1,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
             "get_value for argument 1 failed",
             ret == 1,
             0 );

    if( EXTFN_IS_NULL(value) || EXTFN_IS_EMPTY(value) ) {
    state->return_value = 0;
    return 1;
    }

    if( EXTFN_IS_INCOMPLETE(value) ) {
    // If the value is incomplete, then that means we
// are dealing with a blob.
    tctx->proc_context->get_blob( tctx->args_handle, 1, &blob );
    return_value = ProcessBlob( tctx->proc_context,
 blob,
 letter_to_find );
    blob->release( blob );
    } else {
    // The entire value was put into the value pointer.
    return_value = CountNum( (char *)value.data,
value.piece_len,
letter_to_find );
    }
…
…
}
```

使用 `get_value` 对参数 **1** 进行检索。如果值为空或 NULL，则无需进一步处理。如果该值被确定为 `blob`（使用宏 `EXTFN_IS_INCOMPLETE`），则表 UDF 使用具有 `a_v4_extfn_proc_context` 结构的 `get_blob` 方法以获取 `a_v4_extfn_blob` 实例。`ProcessBlob` 方法读取 `blob` 以确定指定字母的出现数量。

另请参见

- Blob (a_v4_extfn_blob) （第 185 页）
- _open_extfn （第 300 页）

- get_blob （第 284 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）

运行示例表 UDF udf_blob.cxx

`udf_blob` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `udf_blob.cxx` 中的 `samples` 目录中。

1. 若要向服务器声明表 UDF，请发出以下命令：

```
CREATE PROCEDURE udf_blob( IN data long varchar, letter char(1) )
RESULT ( c1 BIGINT )
EXTERNAL NAME 'udf_blob@libv4apiex'
```

2. 从表 UDF 中选择行：

```
set temporary option Enable_LOB_Variables = 'On';
create variable testblob long varchar;
set testblob = 'aaaaaaaaaabbbbbbbbbbb';
select * from udf_blob(testblob, 'a');
```

   提供的字符串包含 10 个字母“a”。

查询处理状态

引用 UDF 的 SQL 语句会在 Sybase IQ 服务器中遍历各种查询处理状态。在其中的每种状态下，服务器都会用第 4 版 API 与 UDF 进行通信和协商。

另请参见

- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）

初始状态

服务器的初始状态。初始状态期间调用的唯一 UDF 方法为 `_start_extfn`。

服务器为创建的 UDF 的每个实例调用启动方法。如果查询由单个服务器线程所执行，则仅调用一次启动方法。如果查询由多个线程所处理，或者查询分布于多个节点，则服务器将创建不同的 UDF 实例，因此启动方法将被调用多次。

UDF 可以在 `a_v4_extfn_proc_context` 结构的 `_user_data` 字段中设置函数实例的级别数据，该数据是启动方法的参数。

标注状态

在标注状态下，服务器会用进行高效正确的查询优化所需的元数据更新分析树。

会调用 `[_enter_state]`、`_describe_extfn` 和 `[_leave_state]` 方法。`_enter_state` 和 `_leave_state` 方法是可选方法，如果 UDF 提供就会调用这些方法。

标注状态在第 4 版 API 中通过 a_v4_extfn_state 枚举由 EXTFNAPIV4_STATE_ANNOTATION 表示：

```
typedef enum a_v4_extfn_state {
    ⋯ EXTFNAPIV4_STATE_ANNOTATION, ⋯
} a_v4_extfn_state;
```

UDF 开发人员可在此阶段中进行一些初步模式协商。通过 UDF 向服务器描述模式支持什么，或通过 UDF 询问服务器模式声明方式，均可进行模式协商。

UDF 向服务器进行自我描述时，服务器会检测失配情况，还会向客户端报告 SQL 错误。例如，如果 UDF 作出其需要四个参数的描述，但 SQL 写入程序仅用两个参数声明了 UDF，则服务器会检测到这种情况，向客户端报告 SQL 错误。

UDF 通过询问服务器其声明方式自行验证时，会相应调整其运行时执行方式：比较声明方式，或通过 set_error 第 4 版 API 返回错误。例如，假定构建最多可返回五个输入标量整数的 UDF。则运行时 UDF 会判断提供了多少输入参数，然后相应调整其内部逻辑。然后 SQL 分析师就能按以下方式创建过程：

```
CREATE PROCEDURE my_sum_2( IN a INT, IN b INT ) EXTERNAL
"my_sum@my_lib"
CREATE PROCEDURE my_sum_3( IN a INT, IN b INT, IN c INT ) EXTERNAL
"my_sum@my_lib"
```

这两个函数采用相同的底层 my_sum 实现方式。UDF 认识到只有两个 my_sum_2 参数，然后尝试对参数 1 和参数 2 求和。对于 my_sum_3，UDF 会对参数 1、参数 2 和参数 3 求和。

UDF 开发人员仅可在标注状态下获取文字常量参数值。进入执行状态后才能获取其他值。要在标注状态下获取参数值，请对 PARM_CONSTANT_VALUE 和 PARM_IS_CONSTANT 属性使用 describe_parameter_get 方法。

在标注状态下，UDF 有权访问模式 describe 属性：

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS
- EXTFNAPIV4_DESCRIBE_PARM_NAME
- EXTFNAPIV4_DESCRIBE_PARM_TYPE
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH
- EXTFNAPIV4_DESCRIBE_PARM_SCALE
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS
- EXTFNAPIV4_DESCRIBE_COL_NAME
- EXTFNAPIV4_DESCRIBE_COL_TYPE
- EXTFNAPIV4_DESCRIBE_COL_WIDTH
- EXTFNAPIV4_DESCRIBE_COL_SCALE
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE

在标注阶段，UDF 可设置上列值以便向服务器定义其模式。如果服务器检测到 UDF 描述和 SQL 过程声明之间有失配之处，则会返回错误。这种技术称为*自我描述*。

UDF 可运用备用技术模式验证。这样 UDF 就需要获取模式描述类型值，并且设置检测到失配情况时报告的错误。通过这种方法可将验证留给 UDF 执行，但 UDF 可决定通过一个实现方案（例如，用于支持给定参数的多个数据类型的功能，或者支持数量不固定的参数的功能）支持多个模式。

另请参见

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（获取） （第 261 页）
- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（设置） （第 263 页）
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（获取） （第 227 页）
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（设置） （第 246 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute（获取） （第 228 页）
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性（设置） （第 247 页）
- EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute（获取） （第 230 页）
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性（设置） （第 248 页）
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute（获取） （第 235 页）
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性（设置） （第 250 页）
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute（获取） （第 236 页）
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性（设置） （第 251 页）
- EXTFNAPIV4_DESCRIBE_COL_NAME (Get) （第 195 页）
- EXTFNAPIV4_DESCRIBE_COL_NAME（设置） （第 211 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- EXTFNAPIV4_DESCRIBE_COL_SCALE (Get) （第 197 页）
- EXTFNAPIV4_DESCRIBE_COL_SCALE（设置） （第 214 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get) （第 202 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT（设置） （第 218 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get) （第 203 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE（设置） （第 218 页）

查询优化状态

在优化状态下，服务器处于初始查询计划构建过程。服务器会收集模式信息和一些初步统计信息。

会调用 [_enter_state]、_describe_extfn 和 [_leave_state] 方法。_enter_state 和 _leave_state 方法是可选方法，如果 UDF 提供就会调用这些方法。

查询优化状态在第 4 版 API 中通过 a_v4_extfn_state 枚举由 EXTFNAPIV4_STATE_OPTIMIZATION 表示：

```
typedef enum a_v4_extfn_state {
    ⋯ EXTFNAPIV4_STATE_OPTIMIZATION, ⋯
} a_v4_extfn_state;
```

在查询优化状态下执行的协商操作包括：

- 服务器和 UDF 确定已对输入表指定分区/排序/集群信息。
- 服务器和 UDF 确定输入表所需的分区/排序信息。
- UDF 对结果表声明物理属性（如排序属性）。
- UDF 描述所有属性和统计信息（如估算的开销），可将其用在查询优化过程中。
  - 估计的表范围包括：
    - **行数** – 在执行状态下出现在 UDF 中的行的总数。输入表参数和返回的表都有此值。
    - **行宽** – 是每行平均字节数的估计值。
  - 列范围估计值包括：
    - **离散值个数** – 表中某一列在所有行上的离散值数量。输入表参数和返回的表都有此值。

在优化状态下，UDF 有权访问 describe 属性：

- EXTFNAPIV4_DESCRIBE_PARM_NAME
- EXTFNAPIV4_DESCRIBE_PARM_TYPE
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH
- EXTFNAPIV4_DESCRIBE_PARM_SCALE
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND
- EXTFNAPIV4_DESCRIBE_COL_NAME

- EXTFNAPIV4_DESCRIBE_COL_TYPE
- EXTFNAPIV4_DESCRIBE_COL_WIDTH
- EXTFNAPIV4_DESCRIBE_COL_SCALE
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT

另请参见

- DEFAULT_TABLE_UDF_ROW_COUNT Option （第 168 页）
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（获取） （第 227 页）
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（设置） （第 246 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute（获取） （第 228 页）
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性（设置） （第 247 页）
- EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute（获取） （第 230 页）
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性（设置） （第 248 页）
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute（获取） （第 235 页）
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性（设置） （第 250 页）
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute（获取） （第 236 页）
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性（设置） （第 251 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute（获取） (第 237 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性（设置） （第 252 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute（获取） （第 238 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性（设置） （第 253 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute（获取） （第 239 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性（设置） （第 254 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get) （第 240 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set) （第 255 页）

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）（第 241 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）（第 242 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）（第 258 页）
- EXTFNAPIV4_DESCRIBE_COL_NAME (Get)（第 195 页）
- EXTFNAPIV4_DESCRIBE_COL_NAME（设置）（第 211 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)（第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置）（第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)（第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置）（第 213 页）
- EXTFNAPIV4_DESCRIBE_COL_SCALE (Get)（第 197 页）
- EXTFNAPIV4_DESCRIBE_COL_SCALE（设置）（第 214 页）
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Get)（第 198 页）
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL（设置）（第 215 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get)（第 202 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT（设置）（第 218 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get)（第 203 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE（设置）（第 218 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Get)（第 204 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER（设置）（第 219 页）
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get)（第 209 页）
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set)（第 224 页）

计划构建状态

在计划构建状态下，服务器会基于查询优化状态下找到的最佳计划构建查询执行计划。

会调用 `[_enter_state]`、`_describe_extfn` 和 `[_leave_state]` 方法。`_enter_state` 和 `_leave_state` 方法是可选方法，如果 UDF 提供就会调用这些方法。

计划构建状态在第 4 版 API 中通过 `a_v4_extfn_state` 枚举由 `EXTFNAPIV4_STATE_PLAN_BUILDING` 表示：

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_PLAN_BUILDING, …
} a_v4_extfn_state;
```

在查询处理过程中的这一时刻，服务器会判断需要通过 UDF 获取的列，还会请求提供有关需要通过表参数获取的列的信息。

如果 UDF 支持并行处理，并且服务器同意查询能进行并行处理，则服务器会创建多个 UDF 实例，以便进行分布式查询处理。

在计划构建状态下，UDF 有权访问所有描述属性。

例如，以下代码段可查询服务器以确定使用了哪些列：

```
    a_sql_int32     rc;
    rg_udf         *rgUdf = (rg_udf *)ctx->_user_data;
    rg_table       *rgTable = rgUdf->rgTable;
    a_sql_uint32    buffer_size = 0;

    buffer_size = sizeof(a_v4_extfn_column_list)   ( rgTable-
>number_of_columns - 1 ) * sizeof(a_sql_uint32);
    a_v4_extfn_column_list *ulist = (a_v4_extfn_column_list *)ctx-
>alloc(
                                    ctx,
                                    buffer_size );
    memset(ulist, 0, buffer_size);

    rc = ctx->describe_parameter_get( ctx,
                0,
                EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,
                ulist,
                buffer_size );

    if( rc != buffer_size ) {
        ctx->free( ctx, ulist );
       UDF_SQLERROR( PC(ctx), "Describe parameter type get failure.",
rc == buffer_size );
    } else {
        rgTable->unused_col_list = ulist;
    }
```

假定以上代码段摘自可生成 4 个结果集列的表 UDF，并且 SQL 语句是

```
SELECT c1, c2 FROM my_table_proc();
```

则 `describe` API 仅会返回 c1 和 c2。UDF 借此可优化结果集值的生成。

另请参见

- Describe API （第 193 页）

执行状态

在执行状态下，服务器会对 UDF 进行执行调用。

创建在计划构建状态下的执行计划，在执行状态下用于计算 SQL 查询结果集。

可调用以下方法：[_enter_state]_describe_extfn、evaluate_extfn、_open_extfn、_fetch_into_extfn、_fetch_block_extfn、_close_extfn、[_leave_state] 和 _finish_extfn。

执行状态在 a_v4_extfn_state API 中通过以下枚举方式表示：

```
typedef enum a_v4_extfn_state {
    ⋯ EXTFNAPIV4_STATE_EXECUTING, ⋯
} a_v4_extfn_state;
```

在执行状态下：

- 能使用输入表参数行和非常量标量数参数值。
- UDF 能打开有关输入表参数的结果集，还能获取行。

执行分区状态
如果有输入表参数，并且 SQL 查询中有 **PARTITION BY** 子句，则服务器会对每个可用分区调用 UDF 一次。

行块数据交换

行块是生产者和消耗程序之间的数据传输区域。

表 UDF 仅可生成行。表 UDF 可使用现有行块，也可自行构建行块。

TPF 可生成和消耗行。TPF 和表 UDF 的行生成方式相同，还可以使用现有行块或自行构建行块。TPF 可消耗输入表中的行，也可向生产者提供行块，或请求生产者自行创建行块。

另请参见

- 行块 (a_v4_extfn_row_block) （第 289 页）
- 表 (a_v4_extfn_table) （第 290 页）
- 表函数 (a_v4_extfn_table_func) （第 298 页）
- _open_extfn （第 300 页）
- _fetch_into_extfn （第 300 页）
- _fetch_block_extfn （第 301 页）
- _rewind_extfn （第 301 页）
- _close_extfn （第 302 页）

行块的提取方法

行块的提取方法是 _fetch_into_extfn 和 _fetch_block_extfn。这些方法是 a_v4_extfn_table_func 结构的一部分。

生成数据时，如果表 UDF 或 TPF 构建自己的行块，则 UDF 必须提供 fetch_block API 方法。如果 UDF 不构建自己的行块，则 UDF 必须提供 fetch_into API 方法。

消耗数据时，如果 TPF 构建自己的行块，则 UDF 调用生产者的 `fetch_into` 方法。如果 TPF 不构建自己的行块，则 TPF 必须调用生产者的 `fetch_block` 方法。

UDF 可以选择用于数据生成和数据消耗的提取方法。通常，这些准则适用：

- **fetch_into** – 当消耗程序拥有数据传输区内存并且请求生产者使用该区时，请使用此 API。在此情况下，消耗程序关注的是如何建立数据传输区，以及生产者如何将必要的数据复制到该区。
- **fetch_block** – 当消耗程序并不关心数据传输区的格式时，请使用此 API。`fetch_block` 请求生产者创建数据传输区，并且提供指向该区的指针。消耗程序拥有数据传输区内存，并且负责从该区中复制数据。

另请参见

- 表参数化函数 （第 128 页）
- fetch_into （第 292 页）
- fetch_block （第 294 页）

fetch_block 方法

将 `fetch_block` 方法用于基础数据存储。

当消耗程序不需要特殊格式的数据时，使用 `fetch_block` 方法作为入口点。`fetch_block` 请求生产者创建数据传输区，并提供指向该区的指针。消耗程序拥有数据传输区内存，并且负责从该区中复制数据。

如果消耗程序不需要特殊布局，则 `fetch_block` 方法比 `fetch_into` 方法更有效。`fetch_block` 调用提供了一个可填充行块，并且将该行块传至下一次 `fetch_block` 调用。此方法是 `a_v4_extfn_table_context` 结构的一部分。

如果基础数据存储无法轻松映射至行块结构，则 UDF 只需将行块指向其内存中的地址即可。这样可避免不必要的数据复制，从而满足其他的内存布局方案。

API 所使用的数据传输区由 `a_v4_extfn_row_block` 结构所定义，该结构定义为一组行，其中每一行又定义为一组列。行块的创建者可以为单行或者一组行分配足够的存储空间。生产者可以对各行进行填充，但是不能超出分配给行块的最大行数。如果生产者拥有额外的行，则其将通过 `fetch` 方法返回数值 1 以通知消耗程序。

提取针对表对象执行，该表对象或者为表 UDF 所生成的结果集，或者为输入表参数消耗的结果集。

另请参见

- 使用行块生成数据 （第 124 页）
- fetch_block （第 294 页）

fetch_block 方法

当消耗程序拥有数据传输区内存并且请求生产者使用该区时，请使用 `fetch_into` API。

如果生产者不知道应该如何在内存中安排数据，则 `fetch_into` 方法将十分有用。当消耗程序拥有特定格式的传输区时，此方法将被用作入口点。`fetch_into()` 函数将提取的行写入所提供的行块。此方法是 `a_v4_extfn_table_context` 结构的一部分。

API 所使用的数据传输区由 `a_v4_extfn_row_block` 结构所定义，该结构定义为一组行，其中每一行又定义为一组列。行块的创建者可以为单行或者一组行分配足够的存储空间。生产者可以对各行进行填充，但是不能超出分配给行块的最大行数。如果生产者拥有额外的行，则其将通过提取方法返回数值 1 以通知消耗程序。

此 API 使消耗程序可以有选择地构造行块，以便数据指针指向其自己的数据结构。这样生产者就可以直接填充消耗程序的内存。如果需要先进行数据整理或者验证检查，则消耗程序可能不希望这样做。

提取针对表对象执行，该表对象或者为表 UDF 所生成的结果集，或者为输入表参数消耗的结果集。

另请参见

- 使用行块生成数据 （第 124 页）
- fetch_into （第 292 页）

使用行块生成数据

表 UDF 或 TPF 可以使用行块结构生成数据。

`a_v4_extfn_row_block` 行块包含三个字段:

- **max_rows** – 行块可以在一块内存中存储的表行的数量。
- **num_rows** – 实际生成或可供消耗的行的数量。其大小不能超过 `max_rows`。
- **row_data** – 生成的或可供消耗的行数组。每行皆为 `a_v4_extfn_row` 结构。

另请参见

- 表 UDF 实现示例 （第 100 页）
- fetch_into （第 292 页）
- fetch_block （第 294 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）
- Row (a_v4_extfn_row) （第 288 页）
- 列数据 (a_v4_extfn_column_data) （第 190 页）

使用 fetch_into 生成数据

请使用 `fetch_into` API 方法以生成数据。

1. 根据读取调用中所生成的行的数量，设置 `num_rows` 的值。
2. 对于所生成的每一行，请将 `a_v4_extfn_row` 的 `row_status` 标志设置为 1（可用）或者 0（不可用）。缺省值为 1。
3. 对于行集中的每一列 (`a_v4_extfn_column_data`) :

选项	描述
is_null	如果返回值是空值，则将其设置为“真”。缺省值为 false。
数据	必须将所返回的数据复制到此指针中
piece_len	所返回数据的实际长度。对于固定长度的数据类型，该值不能超过 *max_piece_len*。对于固定长度数据类型，缺省值为 *max_piece_len*。

4. 对于每一列，返回 1 表示有生成行， 返回 0 则表示未生成行。

使用 fetch_block 生成数据

请使用 `fetch_block` API 方法以生成数据。

1. 将 `max_rows` 设置为生产者所分配的行块结构能够容纳的行数。
2. 当首次执行读取调用时，将分配一个能够容纳 `max_rows` 的行块结构。
3. 根据读取调用中所生成的行的数量，设置 `num_rows` 的值。
4. 对于所生成的每一行，请将 `a_v4_extfn_row` 的 `row_status` 标志设置为 1（可用）或者 0（不可用）。缺省值为 1。
5. 对于行集中的每一列 (`a_v4_extfn_column_data`) :

null_value	表示该值用于指示 NULL
null_mask	识别表示空值的位。
is_null	如果该值是空值，则设置 is_null 的值，使 `(*(cd->is_null) & cd->null_mask) == cd->null_value )`。
数据	将此指针设置为指向生产者内存中的区域（包含返回数据）。
piece_len	所返回数据的实际长度。对于固定长度的数据类型，该值不能超过 *max_piece_len*。对于固定长度数据类型，缺省值为 *max_piece_len*。

6. 从 `fetch_into` 返回 1 表示有生成行， 返回 0 则表示未生成行。当执行最后一次读取调用时，将释放分配给行块结构的内存空间。

行块分配

当生产者使用 `fetch_block` 方法生成数据，或当消耗程序使用 `fetch_into` 方法检索数据时，需要进行行块分配。

`udf_utils.cxx` 包含有示例代码，其中描述了分配和释放行块的方法。

当分配行块时，使用 `extfnapiv4.h` 头文件中的相关数据结构：

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;

typedef struct a_v4_extfn_row {
    a_sql_uint32                *row_status;
    a_v4_extfn_column_data      *column_data;
} a_v4_extfn_row;

typedef struct a_v4_extfn_row_block {
    a_sql_uint32                max_rows;
    a_sql_uint32                num_rows;
    a_v4_extfn_row              *row_data;
} a_v4_extfn_row_block;
```

当配行块时，开发人员必需确定行块能够容纳的行数、每行具有的列数、以及每列所需的字节数。

对于一个大小为 *m* 的行块（其中每行有 *n* 列），开发人员必需分配一个具有 `m` `a_v4_extfn_row` 结构的数组。开发人员必需为该数组中的每行分配 *n*`a_v4_extfn_column_data` 结构。

这些表概述了行块结构中每个成员的分配需求。

表 2. a_v4_extfn_row_block 结构

字段	要求
max_rows	设置此行块能够容纳的行数
num_rows	初始化为 0。使用过程中，将其设为行块包含的实际行数。
*row_data	分配一个数组，其中包含具有 `a_v4_extfn_row` 结构的 max_rows。

表 3. a_v4_extfn_row 结构

字段	要求
*row_status	分配足以容纳 **a_sql_uint32** 的内存空间
*column_data	分配一个包含结果集的列数的数组，该结果集具有 `a_v4_extfn_column_data` 结构

表 4. a_v4_extfn_column_data 结构

字段	要求
*is_null	分配足以容纳 **a_sql_byte** 的内存空间
null_mask	设置值以使公式 (*is_null & null_mask) == null_value，表示列值为空
null_value	设置值以使公式 (*is_null & null_mask) == null_value，表示列值为 NULL
*data	分配一个字节数组，其中足以容纳表示列数据类型的数据
*piece_len	分配足以容纳 **a_sql_uint32** 的内存空间
max_piece_len	设置为列的最大宽度
*blob_handle	始终由服务器所有． 初始化为空。

另请参见

- SQL 数据类型 （第 8 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）

表 UDF 查询计划对象

表 UDF 值和 TPF 值在查询计划中可见。

- **获取的区块** – 显示用于传输 UDF 生成的所有数据的区块数量。此值等于服务器调用 UDF 获取方法的次数。
- **最大行数/ `_fetch_into_extfn`** – （仅当 UDF 使用 `_fetch_into_extfn` 时可见。） 显示 UDF 每次调用 `_fetch_into_extfn` 时可生成的最大行数（取决于服务器）。
- **输出列的最小值/最大值** – 如果 UDF 已通过 `extfnapiv4_describe_col_maximum_value` 设置最小值/最大值，则显示每列的最小值/最大值。最小值/最大值仅对算术类数据列显示。
- **ORDER BY 节点（仅针对 TPF）** – 对于 TPF，会在查询计划中显示 ORDER BY 节点（显示为 TPF SubQuery 节点的下级节点）。ORDER BY 节点表明数据传入表参数时已经过排序。

- **输出行宽** – （仅当 UDF 使用 `_fetch_into_extfn` 时可见。） 显示输出列的宽度（以字节为单位）。此值用于最大行数的计算。
- **TableUDF 节点** – 代表着查询中的一个表 UDF 实例。TableUDF 节点是叶节点。
- **TPF 节点（仅针对 TPF）** – 与 TableUDF 节点相同，但可对 TPF 节点使用输入表参数。TPF 节点是内部节点（最多有一个下级节点），不同于 TableUDF 节点（叶节点）。
- **TPF SubQuery 节点（仅针对 TPF）** – 是 TPF 节点的下级节点。代表着输入表参数的子查询。
- **UDF 库** – UDF 库文件名。显示从中装载用来实现 UDF 的动态库在磁盘中的完整路径。
- **输出列的唯一性** – 用于通过 `extfnapiv4_describe_col_is_unique` 反映值集。
- **TABLE_UDF_ROW_BLOCK_SIZE_KB** – 如果指定的值不是 128 KB，则会在查询计划统计信息中显示选项值。

启用内存跟踪

启用内存跟踪，以定位您 UDF 中的内存泄露，并且释放已泄露的内存部分。内存跟踪将导致性能下降。

启用内存跟踪，以跟踪 `a_v4_extfn_proc_context alloc` 和 `a_v4_extfn_proc_context free` 的所有调用。未匹配释放方法的内存分配将被记录于 `iqmsg` 文件。

1. 请确保将 **external_UDF_execution_mode** 设置为 1 或 2（校验模式或跟踪模式）。
2. 使用具有 `a_v4_extfn_proc_context` 结构的 `alloc` 的 `free` 方法。

另请参见

- 分配 （第 282 页）
- 释放 （第 282 页）

表参数化函数

表参数化函数 (TPF) 由表 UDF 扩展而成，除标量输入参数外，还可以接收表输入参数。

可对 TPF 配置用户指定的分区方式。UDF 开发人员可声明这样的分区模式，即可将数据集细分为较小的查询处理部分（对多个节点分配）。这样就能在分布式服务器环境中，对行组分区同时执行 TPF。查询引擎支持大规模 TPF 并行化处理。

注意： Multiplex 需要单独的许可。请参见《使用 Sybase IQ Multiplex》。

TPF 开发人员的学习路线图

开发 C 或者 C++ TPF。

此路线图假定：

- 您的计算机具有 C 或者 C++ 开发环境。
- 对于可选的数据分区功能，则假定您的计算机具有 multiplex 环境。请参见《使用 Sybase IQ Multiplex》指南。

任务	请参见
熟悉表 UDF 开发。	表 UDF 开发人员的学习路线图 （第 93 页）
请遵循推荐过程以创建 TPF。	开发 TPF （第 129 页） TPF 的实现示例 （第 147 页）
为输入表建立表上下文，并在该环境中消耗表行。	消耗表参数 （第 130 页）
（可选）对传入数据排序。	对输入表的数据排序 （第 132 页）
（可选）对传入数据进行分区，在您的 multiplex 环境中实现 TPF 的并行处理。	输入数据分区 （第 133 页）

开发 TPF

查看开发 TPF 所需的主要步骤。

1. 执行开发表 UDF 所需的相同步骤。
2. 使用输入参数。
3. （可选）对输入表数据排序。
4. （可选）对输入表数据分区。
5. （可选）启用并行 TPF 处理。

另请参见

- 消耗表参数 （第 130 页）
- 对输入表的数据排序 （第 132 页）
- 输入数据分区 （第 133 页）
- _open_extfn （第 300 页）
- _fetch_into_extfn （第 300 页）
- _fetch_block_extfn （第 301 页）
- _rewind_extfn （第 301 页）
- _evaluate_extfn （第 271 页）
- 开发表 UDF （第 98 页）

消耗表参数

表参数是一种非常量参数。这意味着 TPF 必须处于执行状态才可以检索表参数。

TPF 可以通过以下方法检索表参数：

- `_open_extfn`
- `_fetch_into_extfn`
- `_fetch_block_extfn`
- `_rewind_extfn`
- `_evaluate_extfn`

若要消耗表参数，TPF 必须：

获取表对象

TPF 使用具有 `a_v4_extfn_proc_context` 结构的 `get_value` 方法为表参数获取表对象。

表对象 (`a_v4_extfn_table`) 可以从输入表启动行检索操作。以下代码片段描述了 `get_value` 方法是如何为 **1** 参数获取表对象的。为简单起见，此代码假定参数 **1** 为表。

```
a_v4_extfn_value        value;
a_v4_extfn_table *      table;

ctx->get_value( args_handle,
                        1,
                        &value );

table = (a_v4_extfn_table *)value.data;
```

另请参见

- get_value （第 275 页）

打开结果集

当使用 `get_value` 获取到表对象之后，TPF 必须在提取行之前使用具有 `a_v4_extfn_proc_context` 结构的 `open_result_set` 方法打开表对象的结果集。

调用 `open_result_set` 将返回 `a_v4_extfn_table_context` 的实例，TPF 可使用该实例处理表数据。它也会将表对象保存于 `a_v4_extfn_table_context` 对象的表成员中。

以下代码片段描述了 `open_result_set` 是如何得到 `a_v4_extfn_table_context` 实例以提取行的：

```
a_v4_extfn_table_context *  rs  = NULL;
```

```
ctx->open_result_set( ctx,
                      (a_v4_extfn_table *)value.data,
                       &rs ) );
```

另请参见

- open_result_set （第 283 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）

从结果集提取

TPF 使用打开的结果集从输入表中提取表数据。

通过对返回自 `open_result_set` 的 `a_v4_extfn_table_context` 对象调用 `fetch_block` 或 `fetch_into` 方法，以完成提取操作。TPF 可以选择所用的提取方法。如果使用 `fetch_block` 方法，则服务器负责分配行块。如果使用 `fetch_into` 方法，则 TPF 负责分配行块。

每次调用提取方法时，或者不返回任何内容（以返回 false 表示），或者返回填充过的行块结构。然后，该行块结构可用于消耗表数据。

另请参见

- fetch_into （第 292 页）
- fetch_block （第 294 页）
- 行块数据交换 （第 122 页）

使用行块消耗表数据

TPF 使用 `fetch_into` 或 `fetch_block` 行块结构以消耗表数据。

每次成功调用 `fetch_into` 或 `fetch_block` 时，将填充 `a_v4_extfn_row_block` 结构。

`a_v4_extfn_row_block` 的组成为：

- **max_rows** – 行块可在一块内存中存储的表行的数量。
- **num_rows** – 实际生成的、或者可用于消耗的行的数量。其大小不能超过 `max_rows`。
- **row_data** – 实际生成的、或者可用于消耗的行数组。每行皆为 `a_v4_extfn_row` 结构。

`row_data` 中的每行表数据的组成为：

- **row_stats** – 指示该行的值是否存在。为 1 表示该值存在； 为 0 则表示该值不存在。
- **column_data** – 同此行相关的列数据。

`column_data` 的组成为：

组成成员	描述
null_value	表示空值
null_mask	用于表示空值的一位或多位数据
data	指向列数据的指针。根据提取机制的类型，或者指向消耗程序中的地址，或者指向数据在 UDF 中的存储地址。
piece_len	可变长度数据类型的实际数据长度
blob	非空值表示必须 使用 blob API 读取列数据

另请参见

- 列数据 (a_v4_extfn_column_data) （第 190 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）
- Row (a_v4_extfn_row) （第 288 页）
- get_blob （第 297 页）

关闭结果集

当 TPF 处理完表数据之后，它将使用 a_v4_extfn_proc_context 的 close_result_set 方法关闭已打开的结果集。

此代码片段对 close_result_set 方法（用于关闭结果集）进行了描述。

```
ctx->close_result_set( ctx,
                       rs ) );
```

对输入表的数据排序

SQL 分析人员或 UDF 开发人员可以对输入数据进行排序。

SQL 分析人员通过在 **SELECT** 语句中写入 **ORDER BY** 子句以实现排序控制。

UDF 开发人员通过使用 **DESCRIBE_PARM_TABLE_ORDERBY** 属性实现排序控制。

两种方法都将导致服务器对输入数据进行排序，其结果可在排序节点的查询计划中看到。

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute（获取） （第 239 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性（设置） （第 254 页）

输入数据分区

可在并行 TPF 中用 **PARTITION BY** 子句表达和声明调用分区方式。

通过在 SQL 查询中利用并行服务器查询，以及能通过 **PARTITION BY** 子句使用的分配功能，SQL 分析师能高效地利用系统资源。服务器可将数据划分为各不相同的，基于值的行组，也可按行的范围划分行组，具体情况取决于所指定的子句。

- **基于值的分区** – 取决于表达式中的键值。计算取决于是否能看到集合的所有值相同的行时，这些分区用于提供值。
- **基于行的分区** – 简单高效的计算工作分担方法。必须执行并行查询时使用这种方法。

可通过对 TPF 的 **TABLE** 参数执行 **PARTITION BY <expr>** 子句表达分区设计。UDF 开发人员可用表参数元数据属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY**，以编程方式声明 UDF 需要先进行分区，然后才能继续调用。UDF 可查询分区以便强行分区，也可进行动态分区调适。

另请参见

- 使用 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 的并行 TPF PARTITION BY 示例 （第 135 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get) （第 240 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set) （第 255 页）
- 第 4 版 API describe_parameter 和 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY （第 133 页）

第 4 版 API describe_parameter 和 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY

对于所需的列，可用 `describe_parameter_set` 和 `describe_parameter_get` 对输入表参数进行分区。

声明

`describe_parameter` API 有两种声明方式。

describe_parameter_set 声明

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context        *cntxt,
            a_sql_uint32                   arg_num,
            a_v4_extfn_describe_parm_type  describe_type,
            void                           *describe_buffer,
            size_t                         describe_buffer_
            )
```

describe_parameter_get 声明

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context           *cntxt,
```

```
            a_sql_uint32                    arg_num,
            a_v4_extfn_describe_parm_type   describe_type,
            const void                      *describe_buffer,
            size_t                             describe_buffer_
            )
```

用法

arg_num 必须引用 TABLE 参数，**describe_buffer** 必须引用内存块 `a_v4_extfn_column_list` 结构类型，才能使用这些 API。

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

结构字段 **number_of_columns** 的值必须是下列某个值：

- 正整数 N，其中的 N 用于表示出现在分区依据列表中的列的数量。
- 0，用于表示 **PARTITION BY ANY**。
- -1，用于表示 **NO PARTITION BY**。

这种枚举类型在 `extfnapiv4.h` 头文件中定义：

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

可用 `v4_extfn_partitionby_col_num` 枚举类型构建列列表结构，以及执行 `describe_parameter_set` 和 `describe_parameter_get` API，以便将其要求告知服务器，以及判断已对哪些输入列分区。`describe_parameter_set` 和 `describe_parameter_get` API 的执行可能有以下情形：

执行 describe_parameter_set 的情形

列索引情形	描述
{ 1, 1 }	已按 UDF 的请求对输入表的第 1 列分区。
{ 2, 3, 1 }	已按 UDF 的请求对输入表的第 3 列和第 1 列分区。
{ 0 }	UDF 可按 UDF 的请求支持任何形式的输入表分区操作。

执行 describe_parameter_get 的情形

列索引情形	描述
{ 1, 2 }	已对输入表的第 2 列分区。

列索引情形	描述
{ 2, 1, 2 }	已对输入表的第 1 列和第 2 列分区。
{ 0 }	已通过不是基于列的模式对输入表分区。
NULL	未提供运行时分区。

注意： 对于输入查询，出现在选择列表中的 **PARTITION BY** 表达式不得是 **PARTITION BY ANY** 或 **PARTITION BY NONE**。

另请参见

- Describe API （第 193 页）
- 通过列号分区 (a_v4_extfn_partitionby_col_num) （第 287 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute（获取）（第 237 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性（设置） （第 252 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）

使用 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 的并行 TPF PARTITION BY 示例

可通过对 TPF 函数的 **TABLE** 参数使用 **PARTITION BY <expr>** 子句开发分区方法。UDF 开发人员可用表参数元数据属性 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY`，以编程方式声明 UDF 需要先进行分区，然后才能对其进行调用。

这些示例说明的是：

- UDF 向服务器描述分区要求时的各种 SQL 写入程序情形
- 各种情形下的有效查询和无效查询（SQL 异常）
- 服务器如何检测失配情况
- 因为使用 **PARTITION BY** SQL 子句和 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` UDF 属性而产生的各种潜在组合

另请参见

- 输入数据分区 （第 133 页）

示例过程定义
是一个支持 TPF **PARTITION BY** 子句示例的示例过程定义。

本节中的所有 TPF **PARTITION BY** 子句示例的前提都是首次执行此过程定义：

```
CREATE PROCEDURE my_tpf( arg1 TABLE( c1 INT, c2 INT ) )
  RESULTS ( r1 INT, r2 INT, r3 INT )
  EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );
```

另请参见

- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

describe_parameter_set 示例 1：对第 1 列执行的一列分区
此示例 UDF 可告知服务器对第 1 列 (c1) 执行分区。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              1 };    // column index 1 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

另请参见

- 示例过程定义 （第 136 页）

- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

对第 1 列执行一列分区的 SQL 写入程序语义
对于针对第 1 列 (c1) 执行的一列分区，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.x) 分区，SQL 写入程序也特意请求对同一列执行分区。如果两列相符，则通过以下协商查询可继续执行上面的查询，而且不会出错：

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.x ) )
V4 describe_parameter_get API returns: { 1, 1 }
```

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.x) 分区，SQL 写入程序只想让查询引擎对分区执行 UDF。服务器会使用 UDF 的分区首选项，因此会执行示例 1 中那样的有效查询。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ) )
```

此示例显示 SQL 写入程序不将 **PARTITION BY** 子句或 **PARTITION BY DEFAULT** 子句纳入指定的输入表查询。这种情况下，UDF 请求的指定输入表查询适用，目的是对列 T.x 执行分区。

对第 1 列执行一列分区时的 SQL 异常
对于针对第 1 列 (c1) 执行的一列分区， 这些示例查询无效。每个示例都会引发 SQL 异常。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.x) 分区，SQL 写入程序也特意请求对另一列 (T.y) 执行分区，这与 UDF 的请求有冲突，因此服务器会返回 SQL 错误。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

此示例与 UDF 发出的请求有冲突，因为 SQL 写入程序不想对输入表分区，因此服务器会返回 SQL 错误。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y ))
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.x) 分区，SQL 写入程序请求对多个列（T.x 和 T.y）执行分区，这与 UDF 的请求有冲突，因此服务器会返回 SQL 错误。

describe_parameter_set 示例 2：两列分区
此示例 UDF 可告知服务器对第 1 列 (c1) 和第 2 列 (c2) 执行分区。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );
```

```
    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

另请参见

- 示例过程定义 （第 136 页）
- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

两列分区的 SQL 写入程序语义

对于针对第 1 列 (c1) 和第 2 列 (c2) 执行的两列分区，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

在此示例中，UDF 向服务器作的描述是数据按列 T.y 和 T.x 分区。SQL 写入程序也请求对同一列执行分区。如果两列相符，则通过以下协商查询可继续执行上面的查询，而且不会出错：

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

在此示例中，SQL 写入程序不为进行分区指定具体的列。SQL 写入程序转而对输入表进行分区。UDF 请求对列 T.y 和 T.x 分区，因此，服务器会对列 T.y 和 T.x 中的输入数据分区。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
```

```
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ))
```

此示例显示 SQL 写入程序不包括 **PARTITION BY** 子句或 **PARTITION BY DEFAULT** 子句。服务器会使用 UDF 请求的分区，但由于 UDF 作的描述是其要求对列 T.y 和 T.x 分区，因此服务器会通过对列 T.y 和 T.x 进行分区执行查询。

示例 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x,T.y))
```

此示例在语义上与示例 1 相同。两列的顺序不相同，但在给定的分区中，列 T.x 和 T.y 的值一直相同。通过列 (T.x, T.y) 和列 (T.y, T.x) 得出的数据逻辑分区效果相同。

两列分区的 SQL 异常
对于针对第 1 列 (c1) 和第 2 列 (c2) 执行的两列分区， 这些示例查询无效。每个示例都会引发 SQL 异常。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ) )
```

此示例与 UDF 发出的请求有冲突，因为 SQL 写入程序不想对输入表分区。因此服务器会返回 SQL 错误。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ) )
```

在此示例中，UDF 向服务器作的描述是，数据按列 T.y 和 T.x 分区，而 SQL 写入程序请求对列 T.y 或 Tx 执行分区，这与 UDF 的请求有冲突。因此服务器会返回 SQL 错误。

describe_parameter_set 示例 3：对任何一列分区
此示例 UDF 可告知服务器其可对任何一列执行分区。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
```

```
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

另请参见

- 示例过程定义 （第 136 页）
- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

对任何一列分区的 SQL 写入程序语义

对于对任何一列分区，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.x) 分区，SQL 写入程序也特意请求对同一列执行分区。如果两列相符，则通过以下协商查询可继续执行上面的查询，而且不会出错：

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

在此示例中，SQL 写入程序和 UDF 都不为进行分区指定具体的列。SQL 写入程序转而对输入表进行分区，因此，服务器会在不是基于值的模式中安排分区操作，还会对一些范围内的行的数据进行分区。

describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持
此示例 UDF 可告知服务器不能对任何列执行分区，因为 UDF 不支持 **PARTITION BY ANY** 子句。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    // No describe calls
}
```

另请参见

- 示例过程定义 （第 136 页）
- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

PARTITION BY ANY 子句不受支持的 SQL 写入程序语义
UDF 不支持 **PARTITION BY ANY** 子句时，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ))
```

此示例显示 SQL 写入程序不包括 **PARTITION BY** 子句。服务器会使用 UDF 请求的分区，但由于 UDF 不支持任何分区要求，因此服务器会在未执行任何分区操作的情况下执行查询。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

在此示例中，SQL 写入程序在指定的输入表查询中请求 **NO PARTITION BY** 子句。因此，服务器会在不进行运行时分区的情况下执行查询。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x))
```

在此示例中，UDF 不描述任何分区要求。但是，SQL 写入程序请求按列 T.x 分区，因此服务器会通过对列 T.x 执行分区执行查询。

示例 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y))
```

在此示例中，UDF 不描述任何分区要求。但是，SQL 写入程序请求按列 T.y 分区。因此，服务器会通过对列 T.y 执行分区执行查询。

示例 5

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x))
```

在此示例中，UDF 不描述任何分区要求。但是，SQL 写入程序请求按列 T.y 和 T.x 分区。因此，服务器会通过对列 T.y 和 T.x 执行分区执行查询。

示例 6

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

在此示例中，SQL 写入程序请求进行 **PARTITION BY ANY** 分区。但是，UDF 不支持任何分区要求。因此，服务器会通过执行行范围分区执行查询。

describe_parameter_set 示例 5：分区不受支持

此示例 UDF 可告知服务器不支持任何分区操作。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_NONE };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
```

```
            "Runtime error, unable set partitioning requirements for
column."  );
        }
    }
}
```

另请参见

- 示例过程定义 （第 136 页）
- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 6：对第 2 列执行的一列分区 （第 145 页）

分区不受支持的 SQL 写入程序语义

UDF 不支持任何分区操作时，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

在此示例中，SQL 写入程序请求进行 **PARTITION BY ANY** 分区。但是，UDF 不支持任何分区操作，因此，服务器会在不进行运行时分区的情况下执行查询。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

此示例显示 SQL 写入程序不包括 **PARTITION BY** 子句或 **PARTITION BY DEFAULT** 子句。服务器会使用 UDF 请求的分区，但由于 UDF 不支持任何分区操作，因此服务器会在未执行任何分区操作的情况下执行查询。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

在此示例中，SQL 写入程序不请求进行分区，因此，服务器会在不进行运行时分区的情况下执行查询。

分区不受支持的 SQL 异常
示例查询无效，因为 UDF 不支持任何分区操作。每个示例都会引发 SQL 异常。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ))
```

此示例会产生 SQL 错误，因为 SQL 写入程序已请求对列 T.x 执行分区，但 UDF 不支持对任何列执行的任何分区操作。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

此示例会产生 SQL 错误，因为 SQL 写入程序已请求对列 T.y 执行分区，但 UDF 不支持对任何列执行的任何分区操作。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

此示例会产生 SQL 错误，因为 SQL 写入程序已请求对列 T.y 和 T.x 执行分区，但 UDF 不支持对任何列执行的任何分区操作。

describe_parameter_set 示例 6：对第 2 列执行的一列分区
此示例 UDF 可告知服务器对第 2 列 (c2) 执行分区。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              2 };    // column index 2 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
             “Runtime error, unable set partitioning requirements for
```

```
column.”  );
      }
   }
}
```

另请参见

- 示例过程定义 （第 136 页）
- describe_parameter_set 示例 1：对第 1 列执行的一列分区 （第 136 页）
- describe_parameter_set 示例 2：两列分区 （第 138 页）
- describe_parameter_set 示例 3：对任何一列分区 （第 140 页）
- describe_parameter_set 示例 4：PARTITION BY ANY 子句不受支持 （第 142 页）
- describe_parameter_set 示例 5：分区不受支持 （第 143 页）

对第 2 列执行一列分区的 SQL 写入程序语义

对于针对第 2 列 (c2) 执行的一列分区，这些示例查询有效。

示例 1

```
SELECT * FROM my_tpf(
TABLE( SELECT T.x, T.y FROM T )
OVER( PARTITION BY T.y )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.y) 分区，SQL 写入程序也特意请求对同一列执行分区。如果两列相符，则通过以下协商查询可继续执行上面的查询，而且不会出错：

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y ) )
V4 describe_parameter_get API returns: { 1, 2 }
```

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

在此示例中，SQL 写入程序不为进行分区指定具体的列。SQL 写入程序转而对输入表进行分区。UDF 请求对列 T.y 分区，因此，服务器会对列 T.y 中的输入数据分区。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

此示例显示 SQL 写入程序不将 **PARTITION BY** 子句或 **PARTITION BY DEFAULT** 子句纳入指定的输入表查询。这种情况下，UDF 请求的指定输入表查询适用，目的是对列 T.y 执行分区。

对第 2 列执行一列分区时的 SQL 异常
对于针对第 2 列 (c2) 执行的一列分区， 这些示例查询无效。每个示例都会引发 SQL 异常。

示例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.y) 分区，SQL 写入程序也特意请求对另一列 (T.x) 执行分区，这与 UDF 的请求有冲突。因此服务器会返回 SQL 错误。

示例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

此示例与 UDF 发出的请求有冲突，因为 SQL 写入程序不想对输入表分区。因此服务器会返回 SQL 错误。

示例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y )
```

在此示例中，UDF 向服务器作的描述是，数据按第一列 (T.y) 分区，SQL 写入程序请求对多个列（T.x 和 T.y）执行分区，这与 UDF 的请求有冲突。因此服务器会返回 SQL 错误。

TPF 的实现示例

实现示例从简单的 TPF 开始，其后随着示例的改进，其复杂度不断增加、功能不断完善。

TPF 实现示例位于 `samples` 目录。

在名为 `libv4apiex` 的预编译动态库中该示例可用。库的扩展名与平台相关。此库包括定义于 `udf_main.cxx` 中的函数，其中包括例如 `extfn_use_new_api` 的库级函数。请将 `libv4apiex` 置于服务器可以读取的目录之中。

tpf_rg_1

TPF 示例 `tpf_rg_1.cxx` 类似于表 UDF 示例 `udf_rg_2.cxx`。它根据输入参数生成数据行。

所生成行的数量为单个输入表中行值的总和。输出与 `udf_rg_2.cxx` 相同。

此示例的大多数代码同 `udf_rg_2.cxx` 相同。主要区别在于:

- 实现函数的名称具有 `tpf_rg` 前缀，而非 `udf_rg`。有关详细信息，请参见 `tpf_rg_1.cxx`。
- `_describe_extfn` 的实现对该示例的模式进行验证，但是不对生成行的数量进行估计。
- `_open_extfn` 的实现从输入表中读取各行，以便确定生成行的数量。

`_describe_extfn` 方法可满足 `udf_rg_2.cxx` 和此示例间的模式差异。特别是，参数 1 是一个表，该表有一个整数列。此代码片段对 `_describe_extfn` 作了描述:

```
static void UDF_CALLBACK tpf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this TPF
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

        …
…

        // Inform the server that the type of parameter 1 is a TABLE
        type = DT_EXTFN_TABLE;
        desc_rc = ctx->describe_parameter_set
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table should have a single
        // column.
        num_cols = 1;
        desc_rc = ctx->describe_parameter_set
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
              &num_cols,
                  sizeof( num_cols ) );
```

```
        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table column is an integer
        type = DT_INT;
        desc_rc = ctx->describe_column_set
            ( ctx,
              1,
                  1,
              EXTFNAPIV4_DESCRIBE_COL_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

…
…
    }
}
```

在 udf_rg_2.cxx 中，UDF 所生成的行的数量如果为常量，则在描述阶段即可得到该值。表参数不能为常量，所以直到处于执行状态时才可得到其值。因此，任何优化程序都无法在描述阶段即对所生成的行的数量做出估计。

在此示例中，仅当处于标注状态时调用 describe 才可起到作用。当处于其他状态时，类似调用将不执行任何操作。

_open_extfn 方法从输入表中读取行数据，并计算其和值。与在 udf_rg_2.cxx 示例中所采用的方法相同，使用 get_value 检索第一个输入参数的值。此处的不同之处在于参数为 a_v4_extfn_table 指针类型。此代码片段对 _open_extfn 作了描述：

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/************************************/
{
    an_extfn_value             value;
    tpf_rg_state *            state             = NULL;
    a_v4_extfn_table_context *    rs             = NULL;
    a_sql_uint32                num_to_generate    = 0;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

        // Send an error to the client if we could not get the value.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not get the value of parameter 1" );

        return 0;
    }
```

```
    // Open a result set for the input table.
    if( !tctx->proc_context->open_result_set( tctx->proc_context,
                          ( a_v4_extfn_table * )value.data,
                          &rs ) ) {
        // Send an error to the client if we could not open the result
        // set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not open result set on input table." );

        return 0;
    }

    a_v4_extfn_row_block *          rbfb            = NULL;
    a_v4_extfn_row *                rfb            = NULL;
    a_v4_extfn_column_data *        cdfb            = NULL;
    // When using fetch block to read rows from an input table, the
    // server will manage the row block allocation.
    while( rs->fetch_block( rs, &rbfb ) ) {

        // Each successful call to fetch will fill rows in the server
        // allocated row block.  The number of rows retrieved is
        // indicated by the num_rows member.
        for( unsigned int i = 0; i < rbfb->num_rows; i++ ) {
            rfb = &(rbfb->row_data[i]);
            cdfb = &(rfb->column_data[0]);

                // Only consider non-null values.  To determine null we
                // have to use the following logic.
             if( (*cdfb->is_null & cdfb->null_mask) != cdfb->null_value )
{
            num_to_generate += *(a_sql_int32 *)cdfb->data;
                }
}
    }

   if( !tctx->proc_context->close_result_set( tctx->proc_context, rs ) )
{
        // Send an error to the client if we could not close the
        // result set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not close result set on input table." );

        return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (tpf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                   sizeof( tpf_rg_state ) );
    // Start generating at row zero.
```

```
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = num_to_generate;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

当您使用 `get_value` 检索表对象时，请调用 `open_result_set` 以便从表中读取数据行。

若要从输入表中读取各行，UDF 可以使用 `fetch_into` 或 `fetch_block`。当 UDF 从输入表中读取行时，它成为数据消耗程序。如果消耗程序（在本例中为 UDF）想负责行块结构的管理，则其必须分配属于自己的行块结构，并使用 `fetch_into` 对数据进行检索。或者，如果消耗程序希望生产者（在本例中为服务器）负责行块结构的管理，则使用 `fetch_block`。`tpf_rg_1` 对后者作了描述。

使用打开的结果集时，`tpf_rg_1` 通过反复调用 `fetch_block` 以检索来自服务器的数据行。每当成功调用 `fetch_block` 时，将使用至多为 `num_rows` 的行填充服务器分配的行块结构。在 `tpf_rg_1` 中，对各行中第一列的值计算和值。与 `udf_rg_2.cxx` 示例相同，该和值被存储于 `a_v4_extfn_proc_context` 状态以备后用。

另请参见

- Describe API （第 193 页）
- _open_extfn （第 300 页）
- 表 (a_v4_extfn_table) （第 290 页）
- _fetch_block_extfn （第 301 页）

运行 tpf_rg_1 中的示例 TPF

`tpf_rg_1` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `tpf_rg_1.cxx` 中的 `samples` 目录中。

1. 向服务器声明 TPF。

   ```
   CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT ) )
   RESULT( c1 INT )
   EXTERNAL NAME 'tpf_rg_1@libv4apiex';
   ```

2. 声明用作 TPF 输入的表。

   ```
   CREATE TABLE test_table( val int );
   ```

3. 将行插入表中：

   ```
   INSERT INTO test_table values(1);
   INSERT INTO test_table values(2);
   ```

```
INSERT INTO test_table values(3);
COMMIT;
```

4. 从 TPF 中选择行。
   表 test_table 包含三行，其值为 1、2、3。这些值的和为 6。该示例生成 6 行。

```
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

   a) 若要查看 `describe` 对行为的影响，请执行 **CREATE PROCEDURE** 语句，该语句所具有的模式不同于 TPF 在 `describe` 中所发布的模式：

```
CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT,
num2 INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_1@libv4apiex';
```

   b) 从 TPF 中选择行：

```
// This will return an error that the number of columns in
select list
does not match input table param schema
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

tpf_rg_2

TPF 示例 `tpf_rg_2.cxx` 基于 `tpf_rg_1.cxx` 中的示例而构建，并具有类似的行为。它根据输入参数生成数据行。

此示例提供了 `_open_extfn` 在 `a_v4_extfn_func` 描述符中的替代实现方法。其行为与 `tpf_rg_1` 相同，不过 TPF 使用 `fetch_into` 而非 `fetch_block` 从输入表中读取各行。

此代码片段来自于 `_open_extfn` 方法，对 `fetch_into`（用于从输入表中读取各行）进行了描述：

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
…
…
…
   // This block of code will create a statically allocated row block
    // that can contain at most 1 row of data.

    a_sql_uint32            c1_data;
    a_sql_byte            c1_null     = 0x0;
    a_sql_uint32            c1_len     = 0;
    a_sql_byte            null_mask    = 0x1;
    a_sql_byte            null_value     = 0x1;
    a_v4_extfn_column_data    cd[1] =
    {
    { &c1_null,        // is_null
      null_mask,        // null_mask
      null_value,        // null_value
      &c1_data,        // data
      &c1_len,        // piece_len
      sizeof(c1_data),    // max_piece_len
```

```
      NULL            // blob
    }
    };

    a_sql_uint32        r_status;

    a_v4_extfn_row        row =
    {
    &r_status, &cd[0]
    };

    a_v4_extfn_row_block    rb =
    {
    1, 0, &row
    };

    // We are providing a row block structure that was statically
    // allocated to have a single row.  This means that each call to
    // fetch_into will return at most 1 row.
    while( rs->fetch_into( rs, &rb ) ) {

// Only consider non-null rows.  They way the column data has
// been defined allows us to treat c1_null as a boolean.
if( !c1_null ) {
    num_to_generate += c1_data;
}

    }

    …
    …
}
```

当使用 `fetch_into` 从输入表中检索行时，由 TPF 管理行块结构。在此示例中，创建了一个静态行块结构，它可以每次检索一行数据。或者，您可以分配一个动态行块结构，以便同时支持任意数量的行。

在代码片段中，所定义的行块结构将输入表中的列值存储于变量 *c1_data* 之中。如果遇到空行，则将变量 *c1_null* 设置为 1 以作描述。

另请参见

- _open_extfn （第 300 页）
- _fetch_into_extfn （第 300 页）

运行 tpf_rg_2 中的示例 TPF

`tpf_rg_2` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `tpf_rg_2.cxx` 中的 `samples` 目录中。

1. 发出 **CREATE PROCEDURE** 语句，向服务器声明 TPF。

```
CREATE OR REPLACE PROCEDURE tpf_rg_2( IN tab TABLE( num INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_2@libv4apiex';
```

2. 发出 **CREATE TABLE** 语句，声明用作 TPF 输入的表。

```
CREATE TABLE test_table( val INT );
```

3. 将行插入表中。

```
INSERT INTO test_table VALUES(1);
INSERT INTO test_table VALUES(2);
INSERT INTO test_table VALUES(3);
COMMIT;
```

4. 从 TPF 中选择行。

```
SELECT * FROM tpf_rg_2( TABLE( SELECT val FROM test_table ) );
```

表 test_table 包含三行，其值为 1、2、3。这些值的和为 6。该示例生成 6 行。

tpf_blob 中的传递 TPF

TPF 示例 `tpf_blob.cxx` 对高级 UDF LOB 和 CLOB 处理进行了描述。`samples` 目录中提供了此示例。`tpf_blob` 描述了一些在简单示例 `tpf_rg_1` 和 `tpf_rg_2` 中尚未涉及的概念；仅对相关部分进行讨论。

表 UDF 或 TPF 无法生成 LOB 或 CLOB 类型的数据。然而，使用名为*传递*的概念，可将 LOB 或 CLOB 数据从输入表传递至输出表。事实上，可以将任何数据类型从输入表传递至结果集。这样， TPF 就可以对行进行*过滤*，这意味着输出成为输入表行的子集。

`tpf_blob` 所支持的 **CREATE PROCEDURE** 语句为：

```
CREATE PROCEDURE tpf_blob( IN tab TABLE( num INT, s [LONG] <VARCHAR |
BINARY >,
                           IN pattern char(1) )
RESULT SET ( num INT, s [LONG] <VARCHAR | BINARY > )
EXTERNAL NAME ‘tpf_blob@libv4apiex’
```

该过程支持多种模式。在结果集和输入表中，**s** 列的数据类型可以为 `VARCHAR`、`BINARY`,`LONG VARCHAR`、或者 `LONG BINARY`。

动态模式支持

`tpf_blob` 过程的模式是动态的。

在结果集和输入表中，**s** 列的数据类型可以为 `VARCHAR`、`BINARY`,`LONG VARCHAR`、或者 `LONG BINARY`。您可以使用具有 `a_v4_extfn_proc_context` 结构的 `describe_column_get` 方法予以完成，以便获得输入表列的数据类型。根据实际定义的模式，对 TPF 的实现予以调整。根据 **s** 列的数据类型，对过程的 *pattern* 参数的解释有所不同。对于字符数据类型，该参数被解释为字母；对于二进制数据类型，该参数被解释为数字。

另请参见

- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- *describe_column_get （第 194 页）

处理输入表中的 LOB 和 CLOB 列

`tpf_blob` 示例对输入表中每个数据行内某种模式的出现次数进行计算。

在定义的过程中，如果 **s** 列包含有 `LONG VARCHAR` 或 `LONG BINARY` 类型的数据，则必须使用 `blob` API 处理数据。此代码片段来自以 `fetch_into_extfn` 方法，描述了 TPF 如何使用 `blob` API 处理源自输入表的 LOB 和 CLOB 数据。

```
if( EXTFN_COL_IS_BLOB(cd, 1) ) {
    ret = state->rs->get_blob( state->rs, &cd[1], &blob );

    UDF_SQLERROR_RT( tctx->proc_context,
              "Failed to get blob",
              (ret == 1 && blob != NULL),
              0 );

    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {

       num = ProcessBlob( tctx->proc_context, blob, state->pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = ProcessBlob( tctx->proc_context, blob, i );
    }
    ret = blob->release( blob );
    UDF_SQLERROR_RT( tctx->proc_context,
              "Failed to release blob",
              (ret == 1),
              0 );
    } else {
    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {
        num = CountNum( (char *)cd[1].data,
*(cd[1].piece_len),
state>pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = CountNum( (char *)cd[1].data, *(cd[1].piece_len), i );
    }
    }
```

对于输入表中的每一行，TPF 将查看其是否为使用 `EXTFN_COL_IS_BLOB` 宏的 BLOB。如果它是 BLOB，则 TPF 将使用具有 `a_v4_extfn_table_context` 结构的 `get_blob` 方法为指定列创建一个 BLOB 类型的对象。成功时，`get_blob` 方法将为 TPF 提供一个 `a_v4_extfn_blob` 实例，该实例允许 TPF 读取 BLOB 数据。当 TPF 完成 BLOB 数据的读取后，它将会对其调用 `release`。

`ProcessBlob` 方法描述了 BLOB 对象对数据的处理方法：

```
static a_sql_uint64 ProcessBlob(
    a_v4_extfn_proc_context *ctx,
    a_v4_extfn_blob *blob,
    char pattern)
/*****************************/
{
    char buffer[BLOB_ISTREAM_BUFFER_LEN];
    size_t len = 0;
    short ret = 0;

    a_sql_uint64 num = 0;

    a_v4_extfn_blob_istream *is = NULL;

    ret = blob->open_istream( blob, &is );
    UDF_SQLERROR_RT( ctx,
             "Failed to open blob istream",
             (ret == 1 && is != NULL),
             0 );

    for(;;) {
    len = is->get( is, buffer, BLOB_ISTREAM_BUFFER_LEN );
    if( len == 0 ) {
        break;
    }
    num += CountNum( buffer, len, pattern );
    }

    ret = blob->close_istream( blob, is );
    UDF_SQLERROR_RT( ctx,
             "Failed to close blob istream",
             (ret == 1),
             0 );
    return num;
}
```

针对 BLOB 对象的 `open_istream` 方法首先创建一个 `a_v4_extfn_blob_istream` 实例，然后使用该实例将指定数量的 BLOB 数据读取至缓存中（使用 `get` 方法）。

另请参见

- Blob 输入流 (a_v4_extfn_blob_istream) （第 188 页）
- Blob (a_v4_extfn_blob) （第 185 页）
- get_blob （第 297 页）
- fetch_into （第 292 页）

将输入表列传递至结果集

`tpf_blob` 描述了 TPF 是如何将输入表中各行传递至结果集的，以及如何使用 `row_status` 标记指示某一行的存在。

这样， TPF 就可以过滤掉不需要的行。

1. 在描述阶段，请确保 TPF 使用具有 `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` 属性的 `describe_column_set` 方法通知服务器：特定的结果集行是输入表行的子集。此代码片段来自于 `describe_extfn` 方法，对过滤作了描述：

```
else if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    // The output columns of this TPF are the same as the first
    // argument's input table columns.  The following describe
    // informs the consumer of this fact.
    a_v4_extfn_col_subset_of_input colMap;

    for( short i = 1; i <= 2; i++ ) {
        colMap.source_table_parameter_arg_num = 1;
        colMap.source_column_number = i;

        desc_rc = ctx->describe_column_set( ctx,
            0, i,
            EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
            &colMap, sizeof(a_v4_extfn_col_subset_of_input) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );
    }
}
```

2. 请将与传至 `fetch_into_extfn` 方法相同的行块结构传至针对输入表的 `fetch_into` 调用。这确保了结果集的行块结构同输入表一致。

另请参见

- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set) （第 224 页）
- fetch_into （第 292 页）
- _fetch_into_extfn （第 300 页）

运行 tpf_blob.cxx 中的示例 TPF

`tpf_blob` 示例包含于名为 `libv4apiex`（扩展名因平台而异）的预编译动态库。它的实现在 `tpf_blob.cxx` 中的 `samples` 目录中。

1. 向服务器声明 TPF：

```
CREATE OR REPLACE PROCEDURE tpf_blob( IN tab TABLE( num INT,
                                                    s long
varchar ),
                                                IN pattern char(1) )
RESULT( num INT, s long varchar )
EXTERNAL NAME 'tpf_blob@libv4apiex';
```

2. 声明用作 TPF 输入的表。

```
CREATE TABLE test_table( val INT, str LONG VARCHAR );
```

3. 将行插入表中：

```
INSERT INTO test_table VALUES(1, 'aaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(2, 'aaaaaaaaaaabbbbbbbbbb');
```

```
INSERT INTO test_table VALUES(3, 'aaaaaaaaaaaabbbbbbbbbbb');
INSERT INTO test_table VALUES(4, 'aaaaaaaaaaaaabbbbbbbbbbbb');
INSERT INTO test_table VALUES(5, 'aaaaaaaaaaaaaabbbbbbbbbbbbb');
COMMIT;
```

4. 从 TPF 中选择行：

```
SELECT * FROM tpf_blob( TABLE( SELECT val,str FROM test_table ),
'a' );
```

表 test_table 有三行，其中每行包含偶数个 **a**s。第一行有 10 个，第三行有 12 个，第五行有 14 个。

针对表 UDF 和 TPF 查询的 SQL 参考

针对表 UDF 和 TPF 查询的 SQL 语句参考。

ALTER PROCEDURE 语句

用修改后的版本替换现有过程。在 **ALTER PROCEDURE** 语句中包括修改后的整个过程，并对该过程重新分配用户权限。

语法

语法 1

```
ALTER PROCEDURE [ owner.]procedure-name procedure-definition
```

语法 2

```
ALTER PROCEDURE [ owner.]procedure-name
REPLICATE { ON | OFF }
```

语法 3

```
ALTER PROCEDURE [ owner.]procedure-name
SET HIDDEN
```

语法 4

```
ALTER PROCEDURE [ owner.]procedure-name
RECOMPILE
```

语法 5

```
ALTER PROCEDURE
[ owner.]procedure-name ( [ parameter, ⋯] )
[ RESULT (result-column, ...)]
EXTERNAL NAME 'external-call' [ LANGUAGE environment-name ] }
```

参数

- **procedure-definition** – **CREATE PROCEDURE** 语法（名称之后）
- **environment-name** – **JAVA [DISALLOW | ALLOW SERVER SIDE REQUESTS]**

- **external-call** – [*column-name*:]*function-name@library*; ...

用法

ALTER PROCEDURE 语句必须包括 整个新过程。可以将 **PROC** 用作 **PROCEDURE** 的同义词。

ALTER PROCEDURE 语句在语法上与 **CREATE PROCEDURE** 语句相同。

- **语法 1** – 除第一个单词不同以外，**ALTER PROCEDURE** 语句 的语法与 **CREATE PROCEDURE** 语句的语法相同。 Watcom 和 Transact-SQL 术语 过程都能通过使用 **ALTER PROCEDURE** 更改。

  对于 **ALTER PROCEDURE,**，有关过程的现有权限 不会受到更改。如果先后执行 **DROP PROCEDURE** 和 **CREATE PROCEDURE**，则会重新指定执行 权限。
- **语法 2** – 如果需要用 Sybase 复制服务器将过程复制到其他站点，请对过程设置 REPLICATE ON。
- **语法 3** – 使用 **SET HIDDEN** 对关联 过程的定义进行模糊处理，使之不可读。 可将过程卸载并重新装载到其他数据库中。

  注意： 这种设置不可逆。建议在数据库之外保留 原来的过程定义。

  如果使用 **SET HIDDEN**，则使用 调试程序进行调试将不会显示过程定义， 也无法通过过程概况分析获得过程定义。

不能并用语法 2 和语法 1。

- **语法 4** – 使用 **RECOMPILE** 语法 重新编译存储的过程。当重新编译一个过程时，会重新分析存储在目录中的定义，也会验证其语法。 对于可生成结果集但不包含 **RESULT** 子句的过程，数据库服务器会尝试确定 过程结果集的特点，并将信息存储在目录中。 如果自过程创建以来， 过程所引用的表发生变更，从而添加、删除或重命名了列， 这些信息会很有用。

  重新编译不会更改过程的定义。可以 重新编译使用 **SET HIDDEN** 子句隐藏其定义的过程，但其定义仍是隐藏的。
- **语法 5** – 可将这种语法用于 Java UDF。

  iq-environment-name: JAVA [DISALLOW | ALLOW SERVER SIDE REQUESTS]:

  DISALLOW 为缺省值。

  ALLOW 表示允许建立服务器端连接。

  注意： 除非有必要，否则不要指定 **ALLOW**。设置 **ALLOW** 会减慢某些类型的 Sybase IQ 表连接。

  不要在同一查询中一同使用 UDF 与 **ALLOW SERVER SIDE REQUESTS** 和 **DISALLOW SERVER SIDE REQUESTS**。

如果要将 **ALTER PROCEDURE** 语句 用于表 UDF，则所要受到的限制与使用 **CREATE PROCEDURE** 语句（外部过程）时所要受到的限制相同。

标准

- SQL — ISO/ANSI SQL 语法的供应商扩展。
- Sybase — 不受 Adaptive Server Enterprise 支持。

权限

必须是过程的所有者或者是 DBA。自动提交。

另请参见

- 表 UDF 限制 （第 95 页）
- CREATE PROCEDURE 语句（表 UDF） （第 160 页）

CREATE PROCEDURE 语句（表 UDF）

创建外部表用户定义函数的接口（表 UDF）。用户必须经过专门授权才能使用表 UDF。

使用 `a_v4_extfn` API 定义表 UDF。针对不使用 `a_v3_extfn` 或者 `a_v4_extfn` API 的外部过程，请于相关专题中获取 **CREATE PROCEDURE** 语句的参考信息。针对 Java UDF，请于相关专题中获取 **CREATE PROCEDURE** 语句的参考信息。

语法

```
CREATE [ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter[, ...]] )
| RESULT  result-column [, ...] )
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME  'external-call'
```

参数

- **参数：** – [**IN**] *parameter-namedata-type* [**DEFAULT** *expression*]

  | [**IN**] *parameter-nametable-type*
- **表类型：** – TABLE(*column-namedata-type* [, ...])
- **外部调用：** – [*column-name*:]*function-name@library*; ...

用法

语句 **CREATE PROCEDURE** 语句在数据库中 创建过程。具有 DBA 权限的用户可以通过指定所有者 为其他用户创建过程。

如果存储过程返回一个结果集，则该函数也无法 设置输出参数或返回返回值。

当从多个过程中引用临时表时， 如果临时表定义不一致并且引用该表的语句被高速缓存， 则可能会发生潜在问题。在过程中引用临时表时应小心谨慎。

您可以使用 **CREATE PROCEDURE** 语句，以创建使用不同于 SQL 的编程语言 实现的外部表 UDF。然而，在创建外部 UDF 之前 请注意表 UDF 的限制。

适用于标量参数、结果列和 TABLE 参数列 的数据类型必须为有效的 SQL 数据类型。

参数名必须符合其它数据库 标识符（如列名）的规则。它们必须是有效的 SQL 数据类型。

TPF 支持混合标量参数以及单个表参数。 TABLE 参数必须为输入行组定义模式 ，输入行组则由 UDF 进行处理。TABLE 参数的定义 包括列名和列数据类型。

```
TABLE(c1 INT, c2 CHAR(20))
```

上述示例定义了一个包含两列的模式，分别是 INT 类型的 c1 列 和 CHAR(20) 类型的 c2 列。UDF 所处理的每一行 都必须为包含两个值的元组。与标量参数不同， 无法为表参数分配默认值。

- **IN 关键字** – 参数可冠以关键字 IN 作为前缀：
  - IN—该参数 对象为标量参数提供值， 或者为 UDF 的 TABLE 参数提供一组值。

  注意： 不能将 TABLE 参数声明为 INOUT 或者 OUT。只能有一个表参数（其位置并不重要）。
- **OR REPLACE** – 指定 **OR REPLACE** (**CREATE OR REPLACE PROCEDURE**) 将创建一个新过程或替换 同名的现有过程。该子句更改 过程的定义，但保留现有权限。如果您尝试替换已在使用中的过程， 则会返回错误。
- **RESULT** – 为外部 UDF 声明 其结果集中的列的名称和数据类型。 各列的数据类型必须为有效的 SQL 数据类型 （例如，结果集中的列不能为 TABLE 数据类型）。结果中的数据集隐含着 TABLE 数据类型。外部 UDF 只能有一个 TABLE 类型的结果集。

  注意： TABLE 不是输出值。表 UDF 的结果集中不能包含有 LONG VARBINARY 或 LONG VARCHAR 数据类型，但是参数化表函数 (TPF) 的结果集中可以包含有大对象 (LOB) 数据类型。

  TPF 不能生成 LOB 数据，但是其结果集中的列可以为 LOB 数据类型。然而，在输出中得到 LOB 数据的唯一方法是将列从输入表传递至输出表。如示例文件 `tpf_blob.cxx` 所示，描述属性 `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` 允许执行该操作。

  关于过程返回的结果集 的详细信息，请参见《系统管理指南》：第二卷＞使用过程和批处理。
- **SQL SECURITY** – 对于是以 INVOKER（调用过程的用户）身份， 还是以 DEFINER（拥有过程的用户） 身份执行过程进行定义。缺省时以 DEFINER 身份。

  如果指定了 SQL SECURITY INVOKER，则将会占用更多的内存， 因为必须为调用过程的每名用户进行标注。 此外，如果指定了 SQL SECURITY INVOKER，也将以调用者身份 执行名称解析。因此，应注意利用 所有对象的相应所有者 限定这些对象的名称（表、过程等）。例如，假设 user1 创建以下过程：

```
CREATE PROCEDURE user1.myProcedure()
    RESULT( columnA INT )
    SQL SECURITY INVOKER
    BEGIN
        SELECT columnA FROM table1;
    END;
```

如果 user2 试图运行此过程，而表 user2.table1 不存在，则会产生表查寻错误。此外，如果 user2.table1 存在，就会使用该表，而不使用预期的 user1.table1。为防止出现这种情况，请在语句中 限定表引用（user1.table1，而不是仅 table1）。

- **EXTERNAL NAME** – EXTERNAL NAME *'external-call'* : *external-call*: [*operating-system*:]*function-name@library*; ....

  外部 UDF 必须含有 **EXTERNAL NAME** 子句，该子句使用编程语言（例如 C 语言）定义了函数接口。数据库服务器则将函数装载入 其地址空间。

  库名可包含文件扩展名，在 Windows 中 通常为 .dll，在 UNIX 中通常为 .so。在 没有扩展名的情况下，该软件附加平台特定 的缺省库文件扩展名。以下是规范示例。

```
CREATE PROCEDURE mystring( IN instr CHAR(255),
    IN input_table TABLE(A INT) )
    RESULT (CHAR(255))
EXTERNAL NAME
'mystring@mylib.dll;Unix:mystring@mylib.so'
```

  更简单的编写前一个 EXTERNAL NAME 子句的方法 （使用特定于平台的缺省设置）如下：

```
CREATE PROCEDURE mystring( IN instr CHAR(255),
    IN input_table TABLE(A INT) )
    RESULT (CHAR(255))
EXTERNAL NAME 'mystring@mylib'
```

标准

- SQL — 符合 ISO/ANSI SQL 标准。
- Sybase—Transact-SQL **CREATE PROCEDURE** 语句与此不同。
- SQLJ — Java 结果集与建议采用的 标准的语法扩展相同。

权限

必须具有 RESOURCE 权限，否则只能创建临时 过程。具有 DBA 权限的用户可以通过指定所有者 为其他用户创建 UDF。用户必须具有 DBA 权限，才能创建 外部 UDF，或者为其他用户创建外部 UDF。

另请参见

- 示例文件 （第 95 页）

CREATE FUNCTION 语句

在数据库中创建一个新函数。

语法

语法 1

```
CREATE [ OR REPLACE ] [ TEMPORARY ] FUNCTION [ owner.]function-name
( [ parameter, ··· ] )
 RETURNS data-type routine-characteristics
 [ SQL SECURITY { INVOKER | DEFINER } ]
 { compound-statement
  | AS tsql-compound-statement
  | external-name }
```

语法 2

```
CREATE FUNCTION [ owner.]function-name ( [ parameter, ··· ] )
 RETURNS data-type
 URL url-string
 [ HEADER header-string ]
 [ SOAPHEADER soap-header-string ]
 [ TYPE { 'HTTP[:{ GET | POST } ] ' | 'SOAP[:{ RPC | DOC } ]' } ]
 [ NAMESPACE namespace-string ]
 [ CERTIFICATE certificate-string ]
 [ CLIENTPORT clientport-string ]
 [ PROXY proxy-string ]
```

参数

- **url-string：** – ' { **HTTP** | **HTTPS** | **HTTPS_FIPS** }://[*user:password@*]*hostname*[:*port*][/*path*] '
- **parameter：** – **IN***parameter-namedata-type* [**DEFAULT***expression*]
- **routine-characteristics：** – **ONEXCEPTIONRESUME** | [**NOT**] **DETERMINISTIC**
- **tsql-compound-statement：** – *sql-statement sql-statement* ···
- **external-name：** – **EXTERNALNAME***library-call* | **EXTERNALNAME***java-call***LANGUAGEJAVA**
- **library-call：** – '[*operating-system*:]*function-name@library*; ···'
- **operating-system：** – **UNIX**
- **java-call：** – '[*package-name*.]*class-name.method-namemethod-signature*'
- **method-signature：** – ([*field-descriptor*, ···.]) *return-descriptor*
- **field-descriptor 和 return-descriptor：** – **Z** | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[** *descriptor* | **L** *class-name***;**

示例

- **示例 1** – 将 `firstname` 字符串与 `lastname` 字符串并置起来：

```
CREATE FUNCTION fullname (
  firstname CHAR(30),
  lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
  DECLARE name CHAR(61);
  SET name = firstname || ' ' || lastname;
  RETURN (name);
END
```

下面的示例说明了 **fullname** 函数的使用。

- 从两个提供的字符串中返回完整的名称：

  ```
  SELECT fullname ('joe','smith')
  ```

fullname('joe', 'smith')
joe smith

- 列出所有雇员的姓名：

  ```
  SELECT fullname (givenname, surname)
  FROM Employees
  ```

fullname (givenname, surname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
...

- **示例 2** – 使用 Transact-SQL 语法：

  ```
  CREATE FUNCTION DoubleIt ( @Input INT )
  RETURNS INT
  AS
  DECLARE @Result INT
  SELECT @Result = @Input * 2
  RETURN @Result
  ```

  语句 `SELECT DoubleIt( 5 )` 返回值 `10`。
- **示例 3** – 创建一个用 Java 编写的外部函数：

  ```
  CREATE FUNCTION dba.encrypt( IN name char(254) )
  RETURNS VARCHAR
  EXTERNAL NAME
  'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
  LANGUAGE JAVA
  ```

用法

CREATE FUNCTION 语句将在数据库中创建用户定义的函数。通过指定所有者名称，可以为其他用户创建函数。根据权限，可以用与其它非集合函数的使用方法完全相同的方法使用用户定义的函数。

CREATE FUNCTION — 参数名称必须遵守 数据库标识符规则。 它们必须是有效的 SQL 数据类型， 而且必须以关键字 **IN** 作为前缀， 以表明参数是为函数 提供值的表达式。

指定 **CREATE OR REPLACE FUNCTION** 将创建一个新函数或替换同名的现有函数。 替换某个函数后，该函数的定义将更改，但现有的权限将保留。

不能将 **OR REPLACE** 子句与临时函数一起使用。

执行函数时，不必指定所有参数。如果在 **CREATE FUNCTION** 语句中提供了缺省值，则系统会为缺少的参数指派缺省值。如果调用者未提供参数且未设置缺省值，则发生错误。

如果指定 **TEMPORARY** (**CREATETEMPORARYFUNCTION**)，则将意味着该函数仅对创建它的连接可见，并在删除该连接时随之自动删除。也可以显式删除临时函数。无法在临时函数上执行 **ALTER**、**GRANT** 或 **REVOKE** 操作，而且与其它函数不同，临时函数不会在目录或事务日志中予以记录。

具有临时函数创建者（当前用户）权限才能执行临时函数，并且临时函数只能由其创建者所有。因此，创建临时函数时不要指定所有者。

如果已连接到只读数据库，则可以创建和删除临时函数。

SQL SECURITY—定义是作为 INVOKER（调用函数的用户）还是作为 DEFINER（拥有函数的用户）执行函数。缺省为 DEFINER。

如果指定了 **SQL SECURITY INVOKER**，系统将使用更多内存，因为必须为调用过程的每个用户进行注释。此外，如果指定了 **SQL SECURITY INVOKER**，则系统也将以调用者身份执行名称解析。因此，请注意利用所有对象的相应所有者限定这些对象的名称（表、过程等）。

compound-statement——组用 **BEGIN** 和 **END** 括起来的 SQL 语句，中间用分号分隔。请参见“BEGIN……END 语句”。

tsql-compound-statement——批 Transact-SQL 语句。请参见《参考：构件块、表和过程》>“与其它 Sybase 数据库的兼容性”>“Transact-SQL 过程语言概述”>“Transact-SQL 批处理概述”和“CREATE PROCEDURE 语句 [T-SQL]”。

EXTERNAL NAME—使用 **EXTERNAL NAME** 子句的函数是包含对外部库函数调用的包装。使用 **EXTERNAL NAME** 的函数在 **RETURNS** 子句之后可不含任何其它子句。库名可包含文件扩展名，在 Windows 中通常为 `.dll`，在 UNIX 中通常为 `.so`。在没有扩展名的情况下，该软件附加平台特定的缺省库文件扩展名。

临时函数 不支持 **EXTERNAL NAME** 子句。 请参见“SQL Anywhere Server – 编程”>“SQL Anywhere 外部调用接口”。

注意： 此参考指向 SQL Anywhere 文档。

EXTERNAL NAME LANGUAGE JAVA—将 **EXTERNAL NAME** 与 **LANGUAGE JAVA** 子句一起使用的函数是包含 Java 方法的包装。有关调用 Java 过程的信息，请参见“CREATE PROCEDURE 语句”。

ON EXCEPTION RESUME—使用 Transact-SQL-like 错误处理。请参见“CREATE PROCEDURE 语句”。

NOT DETERMINISTIC—指定为 **NOT DETERMINISTIC** 的函数每次在查询中调用时都将重新求值。不是以这种方式指定的函数的结果可以存入高速缓存以便提高性能，并且每次在查询求值过程中使用相同的参数调用函数时，都会重用高速缓存的结果。

对于具有副作用（如修改基础数据）的函数，应将其声明为 **NOT DETERMINISTIC**。例如，一个生成主键值并用在 **INSERT … SELECT** 语句中的函数应声明为 **NOT DETERMINISTIC**：

```
CREATE FUNCTION keygen( increment INTEGER )
RETURNS INTEGER
NOT DETERMINISTIC
BEGIN
  DECLARE keyval INTEGER;
  UPDATE counter SET x = x + increment;
  SELECT counter.x INTO keyval FROM counter;
  RETURN keyval
END
INSERT INTO new_table
SELECT keygen(1), ...
FROM old_table
```

如果函数对给定的输入参数总是返回相同的值，则可以将其声明为 **DETERMINISTIC**。

除非所有用户定义的函数都声明为 **NOT DETERMINISTIC**，否则它们将被视为确定型函数。确定型函数为相同参数返回一致的结果并且没有副作用。也就是说，数据库服务器假设对具有相同参数的同一函数的两次连续调用将返回相同的结果，同时不会对查询语义产生有害副作用。

注意： CLR、ESQL 和 ODBC、Perl 和 PHP 外部 环境中的用户定义的函数由 Catalog 存储 (SQL Anywhere) 处理。 它们不能 利用 Sybase IQ 的性能特性。 Java 中的用户定义的函数由 Sybase IQ 来处理， 并可提高性能。

在某些情况下，如果在用户定义的函数 中发出查询，SQL Anywhere 和 Sybase IQ 之间的语义差异 可导致不同的查询结果。 例如，Sybase IQ 将 CHAR 和 VARCHAR 数据类型视为 各不相同的数据类型，而 SQL Anywhere 按 VARCHAR 的处理方式 来处理 CHAR 数据。

要修改用户定义的函数或通过加密函数定义来隐藏函数的内容， 请使用 **ALTER FUNCTION** 语句。

URL—仅用于定义 **HTTP** 或 **SOAP** Web 服务客户端函数。指定 Web 服务的 URL。其中的用户名和口令参数是可选的，它们提供了一种用于提供 **HTTP** 基本验证所需的证

书的方法。**HTTP** 基本鉴定以 64 为基数对用户和口令信息进行编码并在 **HTTP** 请求的“Authentication”标头中传递编码后的信息。

对于 Web 服务客户端函数来说，返回类型为 **SOAP** 和 **HTTP** 的函数 必须为字符数据类型中的一种，如 VARCHAR。 返回的值为 **HTTP** 响应的正文。 不含任何 **HTTP** 标头信息。 如果需要详细信息 （如状态信息），请使用过程而不是 函数。

参数值作为请求的一部分进行传递。所用的语法取决于请求的类型。对于 **HTTP**:**GET**，参数将作为 URL 的一部分进行传递；对于 **HTTP**:**POST** 请求，值将位于请求的主体中。**SOAP** 请求的参数总是被绑定在请求主体中。

HEADER—创建 **HTTP** Web 服务客户端函数时，此子句用于添加或修改 **HTTP** 请求标头条目。仅可为 **HTTP** 标头指定可打印 ASCII 字符，且这些字符不区分大小写。有关如何使用此子句的详细信息，请参见“CREATE PROCEDURE 语句”的 **HEADER** 子句。

有关使用 HTTP 标头的详细信息，请参见“SQL Anywhere Server – 编程”>“HTTP Web 服务” »“使用 Web 客户端访问 Web 服务” »“开发 Web 客户端应用程序” »“Web 客户端函数和过程要求与推荐”>“HTTP 请求标头管理”。

注意： 此参考指向 SQL Anywhere 文档。

SOAPDHEADER—当将 **SOAP** Web 服务声明为函数时，此子句用于指定一个或多个 **SOAP** 请求 标头条目。 **SOAP** 标头可声明为 静态常量，也可以使用参数 替代机制动态设置（为 hd1、hd2 等声明 **IN**、**OUT** 或 **INOUT** 参数）。 Web 服务函数可以定义一个或多个 **IN** 模式 替代参数，但无法定义 **INOUT** 或 **OUT** 替代参数。 有关如何使用该子句的详细信息，请参见“SQL Anywhere Server – SQL 参考”>“SQL 语句”>“CREATE PROCEDURE 语句（Web 客户端）”中的 **SOAPHEADER** 子句。

注意： 此参考指向 SQL Anywhere 文档。

TYPE—指定提出 Web 服务请求时使用的格式。如果指定 **SOAP** 或未包括类型子句，则使用缺省类型 **SOAP**:**RPC** 。**HTTP** 意味着 **HTTP**:**POST**。**SOAP** 请求始终作为 XML 文档发送。**HTTP**:**POST** 始终用于发送 **SOAP** 请求。

NAMESPACE—仅适用于 **SOAP** 客户端函数并且标识出 **SOAP**:**RPC** 和 **SOAP**:**DOC** 请求通常都需要的方法命名空间。处理请求的 **SOAP** 服务器使用此命名空间来解释 **SOAP** 请求消息主体中的实体名称。可以从 Web 服务服务器中提供的 **SOAP** 服务的 WSDL 说明中获取该命名空间。缺省值为过程的 URL，最多为可选路径组件（但不包括）。

CERTIFICATE—要发出安全 (HTTPS) 请求，客户端必须具有对 HTTPS 服务器所用证书的访问权限。在分号分隔的关键字/值对的字符串中指定所需信息。证书可置于文件中且假设该文件的名称使用文件关键字，或者整个证书可置于字符串中，但不能同时出现这两种情况。它提供以下键：

键	缩写	描述
文件		证书的文件名

键	缩写	描述
certificate	cert	证书
company	co	证书中指定的公司
unit		证书中指定的公司单元
名称		证书中指定的常见名称

只有定向到 HTTPS 服务器或可以从非安全服务器重定向至安全服务器的请求才需要证书。

CLIENTPORT—标识 HTTP 客户端过程使用 TCP/IP 在其上通信的端口号。假定该端口用于并且建议只用于跨防火墙的连接，因为防火墙根据 TCP/UDP 端口进行过滤。可以指定一个端口号、端口号范围或两者的组合，例如 CLIENTPORT '85,90-97'。

请参见《系统管理指南第一卷》>“连接和通信参数”>“网络通信参数”>“ClientPort 通信参数 [CPort]”。

PROXY—指定代理服务器的 URI。在客户端必须通过代理服务器访问网络时使用。表示该过程将连接到代理服务器并通过它向 Web 服务发送请求。

副作用

- 自动提交

标准

- SQL - 符合 ISO/ANSI SQL 标准。
- Sybase — Adaptive Server Enterprise 不支持。

权限

必须具有 RESOURCE 权限。

外部函数（包括 Java 函数）必须有 DBA 权限。

DEFAULT_TABLE_UDF_ROW_COUNT Option

使您得以替换表 UDF（C、C++ 或 Java 表 UDF）所返回的默认行数估计。

允许值
0 至 4294967295

缺省值
200000

范围
设置此选项需要具有 DBA 权限。只能为 PUBLIC 组临时设置。

描述
表 UDF 可以使用 **DEFAULT_TABLE_UDF_ROW_COUNT** 选项为查询处理器提供估计行数（表 UDF 将返回该值）。这是 Java 表 UDF 传递此信息的唯一方法。然而，对于使用 C/C++ 编写的表 UDF，UDF 开发人员应该考虑在 `describe` 阶段发布此信息，即使用 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 描述参数发布它所期望返回的行数。`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 的值始终会覆盖 **DEFAULT_PROXY_TABLE_UDF_ROW_COUNT** 选项的值。

另请参见
- 查询处理状态 （第 115 页）

TABLE_UDF_ROW_BLOCK_SIZE_KB Option

控制服务器分配的行块大小（以千字节为单位）。表 UDF 和 TPF 使用行块。

允许值
0 至 4294967295

缺省值
128

范围
设置此选项需要具有 DBA 权限。只能为 PUBLIC 组临时设置。

描述
指定从服务器所读取的行块大小（以千字节为单位）。

当您使用 `fetch_into` 从表 UDF 读取行时，或者当您使用 `fetch_block` 从 TPF 输入表读取行时，服务器将会分配行块。

指定的行块大小所能容纳的行数即为行块所包含的行数。如果指定的行块尺寸小于容纳单行所需的行块尺寸，则服务器将至少分配单行大小的行块。

FROM 子句

指定 **SELECT** 语句中涉及的数据库表或视图。

语法

```
... FROM table-expression [, …]
```

```
table-expression :
    table-name
    | view-name
    | procedure-name
    | common-table-expression
    | (subquery) [[  AS]  derived-table-name [column_name, ...)]]
    | derived-table
```

```
    | join-expression
    | ( table-expression, ... )
    | openstring-expression
    | apply-expression
    | contains-expression
    | dml-derived-table

table-name :
    [ userid.]table-name ]
    [ [ AS ] correlation-name ]
    [ FORCE INDEX ( index-name ) ]

view-name :
    [ userid.]view-name [ [ AS ] correlation-name ]

procedure-name :
    [  owner, ] procedure-name ([  parameter, ...])
    [  WITH( column-name datatype,)]
    [ [   AS] correlation-name ]

parameter :
scalar-expression | table-parameter

table-parameter :
TABLE(select-statement) [ OVER (table-parameter-over)]

table-parameter-over :
[ PARTITION BY {ANY| NONE| table-expression } ]
[ ORDER BY { expression | integer } [ ASC | DESC ] [, ⋯] ]

derived-table :
    ( select-statement )
    [ AS ] correlation-name [ ( column-name, ... ) ]

join-expression :
    table-expression join-operator table-expression
    [ ON join-condition ]

join-operator :
    [ KEY | NATURAL ] [ join-type ] JOIN
    | CROSS JOIN

join-type :
    INNER
    | LEFT [ OUTER ]
    | RIGHT [ OUTER ]
    | FULL [ OUTER ]

openstring-expression :
    OPENSTRING ( { FILE | VALUE } string-expression )
    WITH ( rowset-schema )
    [ OPTION ( scan-option ...  ) ]
    [ AS ] correlation-name

apply-expression :
    table-expression { CROSS | OUTER } APPLY table-expression
```

```
contains-expression :
    { table-name  | view-name } CONTAINS ( column-name [,...], contains-
query ) [ [ AS ] score-correlation-name ]
```

```
rowset-schema :
    column-schema-list
    | TABLE [owner.]table-name [ ( column-list ) ]
```

```
column-schema-list :
    { column-name user-or-base-type |  filler( ) } [ , ... ]
```

```
column-list :
    { column-name | filler( ) } [ , ... ]
```

```
scan-option :
    BYTE ORDER MARK { ON | OFF }
    | COMMENTS INTRODUCED BY comment-prefix
    | DELIMITED BY string
    | ENCODING encoding
    | ESCAPE CHARACTER character
    | ESCAPES { ON | OFF }
    | FORMAT { TEXT  | BCP  }
    | HEXADECIMAL { ON | OFF }
    | QUOTE string
    | QUOTES { ON | OFF }
    | ROW DELIMITED BY string
    | SKIP integer
    | STRIP { ON | OFF | LTRIM | RTRIM | BOTH }
```

```
contains-query :
    string
```

```
dml-derived-table :
    ( dml-statement  ) REFERENCING ( [ table-version-names  | NONE ] )
```

```
dml-statement :
    insert-statement
    delete-statement
    update-statement
    merge-statement
```

```
table-version-names :
    OLD [ AS ] correlation-name [ FINAL [ AS ] correlation-name ]
    | FINAL [ AS ] correlation-name
```

示例

- **示例 1** – 以下是有效的 **FROM** 子句:

```
...
FROM Employees
...
...
FROM Employees NATURAL JOIN Departments
...
...
FROM Customers
```

```
KEY JOIN SalesOrders
KEY JOIN SalesOrderItems
KEY JOIN Products
...
```

- **示例 2** – 下面的查询阐释如何在查询中使用 派生表:

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
     ( SELECT CustomerID, count(*)
       FROM SalesOrders
        GROUP BY CustomerID )
     AS sales_order_counts ( CustomerID,
                             number_of_orders )
ON ( Customers.ID = sales_order_counts.cust_id )
WHERE number_of_orders > 3
```

用法

SELECT 语句需要用一个表列表来指定 该语句要使用的表。

注意: 虽然这里的说明针对的是表, 但除非另有说明, 同样适用于视图。

FROM 表列表创建由所有指定表中的所有列 组成的结果集。 构成的表中行的 所有组合最初都在结果集中, 但使用连接条件和/或 **WHERE** 条件 通常会减少组合数。

- **SCALAR** – *scalar-parameter* 是 有效的 SQL 数据类型的任何对象。
- **TABLE** – 可以使用 表、视图或常见的 *table-expression* 名称 指定 **TABLE** 参数, 如果在 **TABLE** 参数外也使用该对象, 则将它们视为该对象的新实例。

  此查询阐释了有效的 **FROM** 子句, 其中对同一表 T 的两个引用将视为 同一表 T 的两个不同的实例。

```
SELECT * FROM T, my_proc(TABLE(SELECT T.Z, T.X FROM T)
OVER(PARTITION BY T.Z));
```

  表参数化函数 (TPF) 示例 - 此查询阐释有效的 **FROM** 子句。

```
SELECT * FROM R, SELECT * FROM my_udf(1);
SELECT * FROM my_tpf(1, TABLE(SELECT c1, c2 FROM t))
   (my_proc(R.X, TABLE T OVER PARTITION BY T.X)) AS XX;
```

  如果使用子查询来定义 **TABLE** 参数, 则必须具有以下限制:

  - **TABLE** 参数 必须是 **IN** 类型。
  - **PARTITION BY** 或 **ORDER BY** 子句必须引用派生表 和外部引用的列。 *expression-list* 中的表达式可以是 整数 *K*, 它引用 **TABLE** 输入参数的第 K 列。

  注意: 表 UDF 只能在 SQL 语句的 **FROM** 子句 中引用。

- **PARTITION BY** – **PARTITION BY** 子句从逻辑上指定执行引擎 如何执行函数调用。 执行引擎 必须为每个分区调用函数, 并且该函数 必须在每次调用时对整个分区进行处理。

  PARTITION BY 还指定必须如何对输入 数据进行分区, 以便每次调用函数时 只处理一个分区的数据。 函数调用 的次数必须等于分区数。 对于 TPF, 通过运行时

服务器和 UDF 之间 的动态协商来建立并行特性。 如果 可针对 *N*输入 分区并行执行 TPF， 则可以使用 *M* <=*N* 对函数进行 *M* 次实例化。 每次函数 实例化可以多次调用，每次调用只占用 一个分区。

注意： 执行引擎可以按任意分区顺序调用 函数，并且无论分区顺序如何，函数将假定返回 相同的结果集。 分区 不能拆分成两次函数调用。

只能为 **PARTITION BY** *expression-list* 或 **PARTITION BY ANY** 子句指定一个 **TABLE** 输入参数。 对于必须指定的所有其它 **TABLE** 输入参数， 请显式或隐式指定 **PARTITION BY NONE** 子句。

- **ORDER BY** – **ORDER BY** 子句指定执行引擎 将通过 *expression-list* 对每个分区中的输入数据进行排序。 UDF 需要每个分区都具有此 物理属性。 如果只有一个分区，则所有 输入数据将按 **ORDER BY** 指定进行排序。可以使用 **PARTITION BY NONE** 子句（或不使用 **PARTITION BY**）子句为 **ORDER BY** 子句指定任何 **TABLE** 输入 参数。

  有关 **FROM** 子句中用于全文本 搜索的 *contains-expression* 的信息， 请参见《Sybase IQ 中的非结构化数据分析》。
- **连接** – *join-type* 关键字包括：

表 5. FROM 子句 join-type 关键字

join-type 关键字	描述
CROSS JOIN	返回两个源表的笛卡儿乘积 （矢量积）
NATURAL JOIN	比较两个表中所有对应同名列是否等同 （特殊等值连接；列的长度和数据类型都相同）
KEY JOIN	将第一个表的外键值 限制为与第二个表的主键值相等
INNER JOIN	丢弃结果表中所有未在两个表中 都具备对应行的行
LEFT OUTER JOIN	保留左表的不匹配行， 但丢弃右表中的不匹配行
RIGHT OUTER JOIN	保留右表的不匹配行， 但丢弃左表中的不匹配行
FULL OUTER JOIN	保留左表和右表中的 所有不匹配行

不要混淆 **FROM** 子句中的逗号方式连接和关键字方式连接。 同一查询 可以使用两种方式编写， 即分别使用这些连接方式中的 *一种*。 请优先使用 ANSI 语法关键字方式连接。

下面的查询使用逗号方式连接：

```
SELECT *
  FROM Products pr, SalesOrders so, SalesOrderItems si
  WHERE pr.ProductID = so.ProductID
    AND pr.ProductID = si.ProductID;
```

同一查询使用关键字方式连接（更为可取）：

```
SELECT *
  FROM Products pr INNER JOIN SalesOrders so
```

```
    ON (pr.ProductID = so.ProductID)
  INNER JOIN SalesOrderItems si
    ON (pr.ProductID = si.ProductID);
```

ON 子句用于过滤内部连接、 左连接、右连接和完全连接的数据。 交叉连接没有 **ON** 子句。 在内部连接中，**ON** 子句 等同于 **WHERE** 子句。 不过，在外部连接中，**ON** 和 **WHERE** 子句 还是有所差别。 在外部连接中，**ON** 子句用于过滤矢量积的行， 然后将通过空值扩展的不匹配行 纳入结果中。 **WHERE** 子句则可 对外部连接生成的匹配行和不匹配行中的行 都予以消除。 务必确保所需的不匹配行 不会被 **WHERE** 子句中的谓词消除。

不能在外部连接 **ON** 子句内使用子查询。

有关如何编写 Transact-SQL 兼容连接的信息， 请参见《参考： 构件块、表和过程》>“与其它 Sybase 数据库的兼容性”。

可通过指定 *userid* 对不同用户 拥有的表进行限定。 当前用户所属组拥有的表 在缺省情况下无需指定用户 ID 便可找到。

相关名用于为表赋予一个仅供 SQL 语句 使用的临时名称。 在引用必须由表名限定的列， 但表名很长不方便键入时， 相关名颇为有用。 在同一查询中多次引用同一表时，也有必要使用相关名来 区别各个表的实例。 如果没有指定 相关名，则表名将在当前语句中 用作相关名。

如果表表达式中的同一个表 使用了两次相同的相关名，该表按仅列出 一次处理。 例如，在下面的语句中：

```
SELECT *
FROM SalesOrders
KEY JOIN SalesOrderItems,
SalesOrders
KEY JOIN Employees
```

`SalesOrders` 表的两个实例按一个实例处理， 因此等效于：

```
SELECT *
FROM SalesOrderItems
KEY JOIN SalesOrders
KEY JOIN Employees
```

与之相反的是， 在下面语句中 `Person` 表因具有两个不同的相关名 HUSBAND 和 WIFE， 被视为两个实例处理：

```
SELECT *
FROM Person HUSBAND, Person WIFE
```

无需在 **FROM** 子句中使用一个或多个表或视图， 您便可通过 **SELECT** 语句在不创建视图的 情况下在组中使用组， 或者与组连接。 **SELECT** 语句的这种用法 称为派生表。

连接列 需要相似的数据类型来获得最佳性能。

- **性能注意事项** – 在优化程序 启用的情况下，Sybase IQ 允许在 **FROM** 子句中使用 16 到 64 个表，具体随查询而定；不过，如果在非常复杂的查询的 **FROM** 子句中使用 16 到 18 个以上的表， 则性能可能会受到影响。

  注意： 如果 省略 **FROM** 子句，或者查询中的 所有表都在 SYSTEM dbspace 中， 则查询将由 SQL Anywhere 处理而非 Sybase IQ 处理且行为可能不同， 特别是关于语法和语义限制和 选项设置的影响方面。

  如果您的查询 不需要 **FROM** 子句， 则可以通过添加 **FROM iq_dummy** 子句强制 Sybase IQ 处理查询，在这种情况下，`iq_dummy` 表示在数据库中创建的 包含一行和一列的表。

标准

- SQL—符合 ISO/ANSI SQL 标准。
- Sybase — 某些版本的 Adaptive Server Enterprise 中不 支持 **JOIN** 子句。 这样就必须改用 **WHERE** 子句构建 连接。

权限

必须连接到数据库。

SELECT 语句

从数据库检索信息。

语法

```
SELECT [ ALL | DISTINCT ] [ FIRST | TOP number-of-rows ] select-list
… [ INTO { host-variable-list | variable-list | table-name } ]
… [ INTO LOCAL TEMPORARY TABLE { table-name } ]
… [ FROM table-list ]
… [ WHERE search-condition ]
… [ GROUP BY [ expression [, …]
      | ROLLUP ( expression [, …] )
      | CUBE ( expression [, …] ) ] ]
… [ HAVING search-condition ]
… [ ORDER BY { expression | integer } [ ASC | DESC ] [, …] ]
… [ row-limitation-option ]
```

参数

- **select-list：** –

  ```
  { column-name
  | expression [ [ AS ] alias-name ]
  | * }
  ```

- ***row-limitation-option:*** –

```
LIMIT { [ offset-expression, ] limit-expression
| limit-expression OFFSET offset-expression }
```

- ***limit-expression:*** – *simple-expression*
- ***offset-expression:*** – *simple-expression*
- ***simple-expression:*** –

```
integer
| variable
| ( simple-expression )
| ( simple-expression { + | - | * } simple-expression )
```

示例

- **示例 1** – 列出系统目录中的所有表和视图:

```
SELECT tname
FROM SYS.SYSCATALOG
WHERE tname LIKE 'SYS%' ;
```

- **示例 2** – 列出所有客户及其订单总值:

```
SELECT CompanyName,
  CAST( sum(SalesOrderItems.Quantity *
  Products.UnitPrice) AS INTEGER) VALUE
FROM Customers
  LEFT OUTER JOIN SalesOrders
  LEFT OUTER JOIN SalesOrderItems
  LEFT OUTER JOIN Products
GROUP BY CompanyName
ORDER BY VALUE DESC
```

- **示例 3** – 列出雇员人数:

```
SELECT count(*)
FROM Employees;
```

- **示例 4** – 嵌入式 SQL SELECT 语句:

```
SELECT count(*) INTO :size FROM Employees;
```

- **示例 5** – 按年份、模型和颜色列出总销售额:

```
SELECT year, model, color, sum(sales)
FROM sales_tab
GROUP BY ROLLUP (year, model, color);
```

- **示例 6** – 选择所有具有一定折扣的项目，放入临时表:

```
SELECT * INTO #TableTemp FROM lineitem
WHERE l_discount < 0.5
```

- **示例 7** – 返回在按姓氏对雇员进行排序时首先出现的雇员的信息:

```
SELECT FIRST *
FROM Employees
ORDER BY Surname;
```

- **示例 8** – 返回在按姓氏对员工姓名排序时出现的前五名雇员:

```
SELECT TOP 5 *
FROM Employees
ORDER BY Surname;
```

```
SELECT *
FROM Employees
ORDER BY Surname
LIMIT 5;
```

- **示例 9** – 列出按姓氏以降序进行排序时出现的第五个和第六个雇员：

```
SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT 4,2;
```

- **示例 10** – 返回在按姓氏对员工姓名排序时出现的前五名雇员：

```
CREATE OR REPLACE VARIABLE atop INT = 10;
```

```
SELECT TOP (atop -5) *
FROM Employees
ORDER BY Surname;
```

```
SELECT *
FROM Employees
ORDER BY Surname
LIMIT (atop-5);
```

- **示例 11** – 列出按姓氏以降序进行排序时出现的第五个和第六个雇员：

```
CREATE OR REPLACE VARIABLE atop INT = 10;
```

```
SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT (atop - 8) OFFSET (atop -2 -3 -1);
```

用法

可在 Interactive SQL 中使用 **SELECT** 语句 浏览数据库中的数据，或者将数据从数据库导出到外部文件。

也可以在过程或嵌入式 SQL 中使用 **SELECT** 语句。带 **INTO** 子句的 **SELECT** 语句用于在 **SELECT** 语句只返回一行时从数据库中检索结果。（用 **SELECT INTO** 创建的表不继承 IDENTITY/AUTOINCREMENT 表。）对于多行查询，必须使用游标。如果选择多列且没有使用 *#table*，则 **SELECT INTO** 会创建一个永久基表。无论列数多少，**SELECT INTO***#table* 始终会创建临时表。**SELECT INTO** 单列表会选择到宿主变量中。

注意： 在编写对 **SELECT INTO** 临时表执行的脚本和存储过程时，在 **CAST** 表达式中对不是基列的选择列表项进行封装。这样可保证临时表的列数据类型是所需的数据类型。

如果表的名称相同但所有者不同，则需要提供别名。没有别名的查询将返回错误结果：

```
SELECT * FROM user1.t1
WHERE NOT EXISTS
(SELECT *
FROM user2.t1
WHERE user2.t1.col1 = user1.t.col1);
```

要返回正确结果，请为每个表使用一个别名：

```
SELECT * FROM user1.t1 U1
WHERE NOT EXISTS
(SELECT *
FROM user2.t1 U2
WHERE U2.col1 = U1.col1);
```

具有 *variable-list* 的 **INTO** 子句仅用在过程中。

在 **SELECT** 语句中， 存储过程调用可出现在基表或视图 允许的任意位置。 请注意，需要考虑 CIS 功能补偿性能注意事项。 例如，**SELECT** 语句也可以 返回来自过程的结果集。 有关语法和示例， 请参见“SQL Anywhere Server – SQL 参考”>“SQL 语句”>“FROM 子句”。

注意： 此参考指向 SQL Anywhere 文档。

有关在存储过程内影响从临时表进行选择的限制， 请参见《系统管理指南第二卷》>“使用过程和批处理”>“过程简介”>“在结果集内返回过程结果”。

ALL 或 DISTINCT—如果二者都未指定，则将检索满足 **SELECT** 语句的子句的所有行。如果指定 **DISTINCT**，则会消除重复的输出行。这叫做语句结果的投影。在许多情况下，当指定 **DISTINCT** 时，很多语句的执行时间会显著延长，因此请仅在必要时才使用 **DISTINCT**。

如果使用 **DISTINCT**，则该语句不能包含使用 **DISTINCT** 参数的集合函数。

FIRST 或 TOP number-of-rows—指定从查询返回的行数。**FIRST** 返回从查询选择的第一行。**TOP** 从查询返回指定的行数，其中 *number-of-rows* 的范围为 1 – 2147483647，可以是整数常量或整数变量。

FIRST 和 **TOP** 主要与 **ORDER BY** 子句一起使用。如果没有与 **ORDER BY** 子句一起使用这些关键字，则同一查询每次运行的结果可能会不同，因为优化程序可能会选择不同的查询计划。

FIRST 和 **TOP** 只允许在查询的顶级 **SELECT** 中使用，因此它们不能用于派生表或视图定义。在视图定义中使用 **FIRST** 或 **TOP** 可能会导致在视图上运行查询时忽略该关键字。

使用 **FIRST** 等同于将 ROW_COUNT 数据库选项设置为 1。使用 **TOP** 等同于将 ROW_COUNT 选项设置为相同的行数。如果 **TOP** 和 ROW_COUNT 都进行了设置，则 **TOP** 的值优先。

在涉及全局变量、系统函数或代理表的查询中使用时，ROW_COUNT 选项可能会产生不一致的结果。有关详细信息，请参见“ROW_COUNT 选项”。

select-list—*select-list* 是一个由逗号分隔的表达式列表，用于指定从数据库检索的内容。如果指定一个星号 (*)，将选中 **FROM** 子句（table-name 指定表的所有列）中

的所有表的所有列。*select-list* 中允许使用集合函数和解析函数。请参见《参考：构件块、表和过程》>“SQL 函数”。

注意： 在 Sybase IQ 中，顶级 **SELECT** 的选择列表中允许标量子查询（嵌套的 select），这与在 SQL Anywhere 和 Adaptive Server Enterprise 中一样。 子查询不能 用在条件值表达式中（例如 **CASE** 语句中）。

子查询还可以在 **WHERE** 或 **HAVING** 子句谓词（支持的谓词类型之一）中使用。 但在 **WHERE** 或 **HAVING** 子句中， 子查询不能用在值表达式中，也不能用在 **CONTAINS** 或 **LIKE** 谓词中。 外部链接的 **ON** 子句 或 **GROUP BY** 子句中不允许子查询。

有关子查询的用法的详细信息，请参见《参考：构件块、表和过程》>“SQL 语言元素”>“表达式”>“表达式中的子查询”以及《参考：构件块、表和过程》>“SQL 语言元素”>“搜索条件”>“搜索条件中的子查询”。

---

alias-names 在整个查询中 都可以用来表示带别名的表达式。 Interactive SQL 还在 **SELECT** 语句的 每个输出列的顶部显示 别名。 如果表达式后面 没有指定可选的 *alias-name*，Interactive SQL 将显示该表达式。 如果为列别名使用与列名一样的名称或表达式， 该名称将处理为带别名的列， 而不是表列名。

INTO host-variable-list—仅用于嵌入式 SQL。它指定 **SELECT** 语句结果的位置。*select-list* 中的每一项都必须有一个 *host-variable*。选择列表中的项依次放入宿主变量中。每个 *host-variable* 还可以有一个指示符宿主变量，以便程序可以判定选择列表项是否为 NULL。

INTO variable-list—仅用在过程中。它指定 **SELECT** 语句结果的位置。选择列表中的每一项都必须有一个变量。选择列表中的项依次放入变量中。

INTO table-name—用于创建表并用数据填充表。

如果表名以 **#** 开头，则该表创建为临时表。否则，该表创建为永久基表。对于所要创建的永久表，查询必须满足以下条件：

- *select-list* 中包含多个项目，**INTO** 目标是单个 *table-name* 标示符，或
- select-list 包含一个 *****，**INTO** 目标指定为 *owner.table*。

若要创建有一列的永久表，表名必须指定为 *owner.table*。可忽略临时表的所有者说明。

作为创建表的副作用，此语句会在执行前导致 **COMMIT**。执行此语句需要具有 RESOURCE 权限。新表未被授予任何权限：该语句是后接 **INSERT... SELECT** 的 **CREATE TABLE** 简写形式。

不允许在存储过程或函数中执行 **SELECT INTO**，因为 **SELECT INTO** 为原子语句，无法在原子语句中执行 **COMMIT**、**ROLLBACK** 或某些 **ROLLBACK TO SAVEPOINT** 语句。有关详细信息，请参见《系统管理指南第二卷》>“使用过程和批处理”>“控制语句”>“原子复合语句”和《系统管理指南第二卷》>“使用过程和批处理”>“过程中的事务和保存点”。

用此语句创建的表没有定义主键。可以使用 **ALTER TABLE** 添加主键。在对表应用任何 UPDATE 或 DELETE 之前应添加主键；否则，这些操作会使受影响的行的所有列值记录在事务日志中。

该子句仅限于在有效的 SQL Anywhere 查询中使用。 不支持 Sybase IQ 扩展。

INTO LOCAL TEMPORARY TABLE—创建本地临时表并用查询结果对其进行填充。使用此子句时，临时表名不必以 # 开头。

FROM table-list—行是从 *table-list* 指定的表和视图中检索的。连接可使用连接运算符进行指定。有关详细信息，请参见“FROM 子句”。不带 **FROM** 子句的 **SELECT** 语句可用于显示不是从表中派生的表达式的值。例如：

```
SELECT @@version
```

显示全局变量 @@version 的值。这等效于：

```
SELECT @@version
FROM DUMMY
```

注意： 如果 省略 **FROM** 子句，或者查询中的 所有表都在 SYSTEM 数据库空间中，则查询将由 SQL Anywhere 而非 Sybase IQ 处理 且行为可能不同，特别是关于 语法和语义限制和选项设置的 影响方面。 有关可能应用于 处理的规则，请参见 SQL Anywhere 文档。

如果您的查询不需要 **FROM** 子句， 则可以通过添加“`FROM iq_dummy`”子句强制 Sybase IQ 处理查询， 在这种情况下，`iq_dummy` 表示 在数据库中创建的包含一行和一列的表。

WHERE search-condition—指定从 **FROM** 子句命名的表选择哪些行。还用于在多个表之间进行连接。这是通过在 **WHERE** 子句中放置一个条件来完成的，该条件将一个表中的一列或一组列与另一个表中的一列或一组列相关。两个表都必须在 **FROM** 子句中列出。

一组列的 **SELECT** 和 **WHERE** 子句中不允许使用相同的 **CASE** 语句。请参见《参考：构件块、表和过程》>“SQL 语言元素”>“搜索条件”。

Sybase IQ 还支持对子查询 谓词执行析取。 每个子查询可以 与其它谓词一起显示在 **WHERE** 或 **HAVING** 子句内， 并可使用 AND 或 OR 运算符进行组合。 请参见《参考：构件块、表和过程》>“SQL 语言元素”>“搜索条件”>“搜索条件中的子查询”>“分离子查询谓词”。

GROUP BY—可以按列、 别名或函数进行分组。**GROUP BY** 表达式也必须出现 在选择列表中。 指定列、别名或函数的 每个不同的值集在查询结果均可以找到 与之相对应的一行。 结果行通常称为组， 因为对于表列表中的每组行， 结果中均有一行。 如果是 **GROUP BY**，所有空值都视为 完全相同。 然后可将集合函数应用于这些组 以获得有意义的结果。

GROUP BY 必须含有一个以上的常量。不需向 **GROUP BY** 子句添加常量即可选择分组查询中的常量。如果 **GROUP BY** 表达式只含有一个常量，则会返回错误，查询也会被拒绝。

如果使用 **GROUP BY**，选择列表、**HAVING** 子句和 **ORDER BY** 子句则无法引用除 **GROUP BY** 子句中之外的任何标示符。但有以下例外：*select-list* 和 **HAVING** 子句可以包含集合函数。

ROLLUP 运算符—**ROLLUP** 子句中的 **GROUP BY** 运算符可用不同的详尽程度来分析小计。它可以创建从详细级别一直累计到总计的小计。

ROLLUP 运算符要求以参数的方式提供分组表达式的有序列表。**ROLLUP** 首先计算 **GROUP BY** 中指定的标准集合值。然后，**ROLLUP** 在整个分组列的列表中从右侧移到左侧，并以累积方式创建更高级别的小计。在结尾处创建总计。如果 *n* 代表分组列数，则 **ROLLUP** 会创建 *n+1* 个级别的小计。

对 **ROLLUP** 运算符的限制如下：

- **ROLLUP** 支持可用于 **GROUP BY** 子句的所有集合函数，但 **ROLLUP** 目前支持 **COUNT DISTINCT** 和 **SUM DISTINCT**。
- **ROLLUP** 仅可用于 **SELECT** 语句；在 **SELECT** 子查询中不可使用 **ROLLUP**。
- 当前不支持将多个 **ROLLUP**、**CUBE** 和 **GROUP BY** 列组合在同一个 **GROUP BY** 子句中的分组规范。
- 不支持以常量表达式作为 **GROUP BY** 键。

GROUPING 可与 **ROLLUP** 运算符配合使用来区分存储空值和 **ROLLUP** 创建的查询结果中的空值。

ROLLUP 语法：

```
                SELECT ... [ GROUPING ( column-name ) ...] ... GROUP BY
[ expression [, ...] | ROLLUP ( expression [, ...] ) ]
```

有关运算符表达式的格式，请参见《参考：构件块、表和过程》>“SQL 语言元素”>“表达式”。

GROUPING 采用列名作为参数并返回布尔值：

表 6. 使用 ROLLUP 运算符时 GROUPING 返回的值

如果结果值是	GROUPING 返回
由 **ROLLUP** 运算创建的空值	1 (TRUE)
指示该行是小计所在行的空值	1 (TRUE)
并非由 **ROLLUP** 运算创建	0 (FALSE)
存储的 NULL	0 (FALSE)

有关 **ROLLUP** 示例，请参见《系统管理指南第二卷》>“使用 OLAP”。

CUBE 运算符—Theoperator in the **CUBE** 子句中的 **GROUP BY** 运算符可将数据分布到具有多个维度的分组中，并以此来分析数据。**CUBE** 需要分组表达式（维度）的有序列表作为参数，并让 **SELECT** 语句计算所有可能维度组的组合的小计。

对 **CUBE** 运算符的限制如下：

- **CUBE** 支持可用于 **GROUP BY** 子句的所有集合函数，但 **CUBE** 目前不支持 **COUNT DISTINCT** 或 **SUM DISTINCT**。
- **CUBE** 目前不支持逆分布解析函数 **PERCENTILE_CONT** 和 **PERCENTILE_DISC**。
- **CUBE** 仅可用于 **SELECT** 语句；在 **SELECT** 子查询中不可使用 **CUBE**。
- 当前不支持将多个 **ROLLUP**、**CUBE** 和 **GROUP BY** 列组合在同一个 **GROUP BY** 子句中的 **GROUPING** 规范。
- 不支持以常量表达式作为 **GROUP BY** 键。

GROUPING 可与 **CUBE** 运算符配合使用来区分存储空值和 **CUBE** 创建的查询结果中的空值。

CUBE 语法：

```
SELECT ⋯ [ GROUPING ( column-name ) ⋯] ⋯ GROUP BY
[ expression [, ⋯] | CUBE ( expression [, ⋯] ) ]
```

GROUPING 采用列名作为参数并返回布尔值：

表 7. 使用 CUBE 运算符时 GROUPING 返回的值

如果结果值是	GROUPING 返回
由 **CUBE** 运算创建的空值	1 (TRUE)
指示该行是小计所在行的空值	1 (TRUE)
并非由 **CUBE** 运算创建	0 (FALSE)
存储的 NULL	0 (FALSE)

生成查询计划时，Sybase IQ 优化程序 会估计通过 **GROUP BY CUBE** 散列运算 生成的组的总数。 MAX_CUBE_RESULTS 数据库选项 对优化程序视为可以运行的散列算法的估计行数 设置一个上限。 如果实际行数 超过 MAX_CUBE_RESULT 选项值， 优化程序将停止处理查询，并返回错误消息“估计数目： nnn 超过 GROUP BY CUBE 或 ROLLUP 的 DEFAULT_MAX_CUBE_RESULT”，其中 *nnn* 是 IQ 优化程序估计的数值。 有关设置 MAX_CUBE_RESULT 选项的信息， 请参见“MAX_CUBE_RESULT 选项”。

有关 **CUBE** 示例，请参见《系统管理指南第二卷》>“使用 OLAP”。

HAVING search-condition—根据组值而不是各行值。只有当此语句有 **GROUP BY** 子句或选择列表只包括集合函数时，才能使用 **HAVING** 子句。**HAVING** 子句中引用的任何列名都必须位于 **GROUP BY** 子句中或用作 **HAVING** 子句中的集合函数的参数。

ORDER BY—排序查询结果。**ORDER BY** 列表中的每一项均可标记为 ASC 以按升序排序，或者标记为 DESC 以按降序排序。如果两者都未指定，则假定为升序。如果表达式是整数 **n**，则查询结果按选择列表中的第 **n** 项排序。

在嵌入式 SQL 中，**SELECT** 语句用于从数据库中检索结果，并通过 **INTO** 子句将值放入宿主变量。**SELECT** 语句必须只返回一行。对于多行查询，必须使用游标。

不能在 **SELECT** 列表中包含 Java 类，但可以创建一个充当 Java 类的包装的函数或变量，然后选择它。

row-limitation 子句 - 行限制子句允许您只返回满足 **WHERE** 子句的行的子集。 一次只能指定一个 *row-limitation* 子句。 如果指定该子句，以一种有意义的方式对这些行进行排序需要使用 **ORDER BY** 子句。

LIMIT 和 **OFFSET** 参数可以是针对宿主变量、整数常量或整数变量的简单算术表达式。**LIMIT** 参数的结果值必须是大于或等于 0 的值。**OFFSET** 参数的结果值必须是大于或等于 0 的值。如果未指定 *offset-expression*，缺省值是 0。

行限制子句 **LIMIT***offset-expression*，*limit-expression* 等同于 **LIMIT***limit-expression***OFFSET***offset-expression*。

缺省情况下禁用 **LIMIT** 选项。 使用 RESERVED_KEYWORDS 选项以启用 **LIMIT** 关键字。

标准

- SQL—符合 ISO/ANSI SQL 标准。
- Sybase - 受 Adaptive Server Enterprise 支持， 有一些语法差异。

权限

必须有指定表和视图的 SELECT 权限。

a_v4_extfn 的 API 参考

针对 a_v4_extfn 函数、方法和属性的参考信息。

Blob (a_v4_extfn_blob)

请使用 a_v4_extfn_blob 结构以表示一个独立的 BLOB 对象。

实现

```
typedef struct a_v4_extfn_blob {
    a_sql_uint64 (SQL_CALLBACK *blob_length)(a_v4_extfn_blob *blob);
    void (SQL_CALLBACK *open_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream **is);
    void (SQL_CALLBACK *close_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream *is);
    void (SQL_CALLBACK *release)(a_v4_extfn_blob *blob);
} a_v4_extfn_blob;
```

方法总结

方法名称	数据类型	描述
blob_length	a_sql_uint64	以字节为单位返回指定 BLOB 的长度。
open_istream	无类型	打开一个可用于自指定 BLOB 开始读取数据的输入流。
close_istream	无类型	关闭针对于指定 BLOB 的输入流。
release	无类型	指示调用方已完成对此 blob 的调用，并且 blob 所有者可以释放资源。**release()** 调用完后引用 blob 会产生错误。调用 **release()** 时所有者通常会删除内存。

描述
如下情况时请使用 a_v4_extfn_blob 对象：

- 表 UDF 需要从标量输入值读取 LOB 或 CLOB 数据
- TPF 需要从输入表中的列读取 LOB 或 CLOB 数据

约束和限制
无。

blob_length

使用 `blob_length` v4 API 方法返回指定 BLOB 的长度（以字节为单位）。

声明

```
a_sql_uint64 blob_length(
    a_v4_extfn_blob *
)
```

用法
以字节为单位返回指定 BLOB 的长度。

参数

参数	描述
blob	需要获取长度值的 BLOB。

返回
指定的 BLOB 的长度。

另请参见

- open_istream （第 186 页）
- close_istream （第 187 页）
- release （第 188 页）

open_istream

请使用 `open_istream` v4 方法打开输入流，以便从 BLOB 读取数据。

声明

```
void open_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream **is
    )
```

用法
打开一个可用于自指定 BLOB 开始读取数据的输入流。

参数

参数	描述
blob	需要打开输入流的 BLOB。
is	输出参数，用于标识所返回的已打开输入流。

返回
无。

另请参见
- blob_length （第 186 页）
- close_istream （第 187 页）
- release （第 188 页）

close_istream

请使用 `close_istream` v4 API 方法关闭指定 BLOB 的输入流。

声明

```
void close_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream *is
    )
```

用法
关闭以前使用 `open_istream` API 打开的输入流。

参数

参数	描述
blob	需要关闭输入流的 BLOB。
is	标识所要关闭的输入流的参数。

返回
无。

另请参见
- blob_length （第 186 页）
- open_istream （第 186 页）
- release （第 188 页）

release

请使用 `release` v4 API 方法指示调用方已经使用完当前选定的 BLOB 。所有者得以释放内存。

声明

```
void release(
a_v4_extfn_blob *blob
)
```

用法

指示调用方已完成对此 blob 的调用，并且 blob 所有者可以释放资源。**release()** 调用完后引用 blob 会产生错误。调用 **release()** 时所有者通常会删除内存。

参数

参数	描述
blob	需要释放的 BLOB。

返回

无。

另请参见

- blob_length （第 186 页）
- open_istream （第 186 页）
- close_istream （第 187 页）

Blob 输入流 (a_v4_extfn_blob_istream)

请使用 `a_v4_extfn_blob_istream` 结构为 LOB 或 CLOB 标量输入列、或输入表中的 LOB 或 CLOB 列读取 BLOB 数据。

实现

```
typedef struct a_v4_extfn_blob_istream {
    size_t (SQL_CALLBACK *get)( a_v4_extfn_blob_istream *is, void
*buf, size_t len );
    a_v4_extfn_blob     *blob;
    a_sql_byte          *beg;
    a_sql_byte          *ptr;
    a_sql_byte          *lim;
} a_v4_extfn_blob_istream;
```

方法总结

方法名称	数据类型	描述
get	size_t	从 BLOB 输入流中获取指定量的数据。

数据成员及数据类型的摘要

数据成员	数据类型	描述
Blob	a_v4_extfn_blob	基本 BLOB 结构（输入流据此创建）。
Beg	a_sql_byte	指向当前数据块起始位置的指针。
Ptr	a_sql_byte	指向数据块中当前字节的指针。
Lim	a_sql_byte	指向当前数据块结尾位置的指针。

获取

请使用 `get` v4 API 方法从 BLOB 输入流获取指定量的数据。

声明

```
size_t get(
    a_v4_extfn_blob_istream *is,
    void *buf,
    size_t len
)
```

用法

从 BLOB 输入流中获取指定量的数据。

参数

参数	描述
is	需要从中检索数据的输入流。
buf	需要存储数据的缓存。
len	需要检索的数据量。

返回

接收的数据量。

列数据 (a_v4_extfn_column_data)

a_v4_extfn_column_data 结构代表一个单列的数据值。当生成结果集数据时，生产者将使用该结构，而当读取输入表列数据时，消耗程序将使用该结构。

实现

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
is_null	a_sql_byte *	指向 存储 NULL 信息的字节。
null_mask	a_sql_byte	用于表示空值的一位或多位数据
null_value	a_sql_byte	表示空值
data	void*	指向列数据的指针。根据提取机制的类型，或者指向消耗程序中的地址，或者指向数据在 UDF 中的存储地址。
piece_len	a_sql_uint32 *	可变长度数据类型的实际数据长度
max_piece_len	size_t	此列的最大允许数据长度。
blob_handle	void*	非空值表示必须 使用 blob API 读取列数据

描述

a_v4_extfn_column_data 结构代表数据值以及特定数据列的相关属性。当生成结果集数据时，生产者将使用此结构。数据生产者也希望为 *data*、*piece_len* 和 *is_null* 标志创建存储空间。

is_null、*null_mask* 和 *null_value* 数据成员指示列中有空值，并且对某些情况（ 8 列共用一个包含有 NULL 位编码的字节 ）或者其他情况（每列使用一个完整字节）进行处置。

此示例描述如何解释用于代表 NULL 的三个字段：*is_null*、*null_mask* 和 *null_value*。

```
is_value_null()
   return( (*is_null & null_mask) == null_value )

set_value_null()
   *is_null = ( *is_null & ~null_mask) | null_value

set_value_not_null()
   *is_null = *is_null & ~null_mask | (~null_value & null_mask)
```

另请参见

- get_blob （第 297 页）

列的列表 (a_v4_extfn_column_list)

对 **PARTITION BY** 或 **TABLE_UNUSED_COLUMNS** 进行描述时，请使用 `a_v4_extfn_column_list` 结构以提供列的列表。

实现

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];      // there are
number_of_columns entries
} a_v4_extfn_column_list;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
number_of_columns	a_sql_uint32	列表中的列数。
column_indexes	a_sql_uint32 *	大小为 number_of_columns 且带有列索引（从 1 开始）的连续数组。

描述

列列表中的内容，其含义的改变取决于该列表是否同 **TABLE_PARTITIONBY** 或 **TABLE_UNUSED_COLUMNS** 一起使用。

另请参见

- 第 4 版 API describe_parameter 和 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY （第 133 页）
- 使用 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 的并行 TPF PARTITION BY 示例 （第 135 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute（获取） （第 243 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set) （第 259 页）

列顺序 (a_v4_extfn_order_el)

请使用 a_v4_extfn_order_el 结构以描述列中的元素顺序。

实现

```
typedef struct a_v4_extfn_order_el {
    a_sql_uint32       column_index;    // Index of the column in the
table (1-based)
    a_sql_byte       ascending;              // Nonzero if the column
is ordered "ascending".
} a_v4_extfn_order_el;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
column_index	a_sql_uint32	表中列的索引（从 1 开始）。
ascending	a_sql_byte	如果列顺序为“升序”，则其值“非零”。

描述

a_v4_extfn_order_el 结构对列作了描述，并且表明该列是否应为升序或降序。a_v4_extfn_orderby_list 结构包含具有这些结构的一个数组。对于 **ORDERBY** 子句中的每一列，都有一个 a_v4_extfn_order_el 结构。

另请参见

- 按列表排序 (a_v4_extfn_orderby_list) （第 286 页）

列子集 (a_v4_extfn_col_subset_of_input)

使用 a_v4_extfn_col_subset_of_input 结构以声明，输出 列中的某个始终取自于 UDF 的某个特定输入列。

实现

```
typedef struct a_v4_extfn_col_subset_of_input {
    a_sql_uint32    source_table_parameter_arg_num;    // arg_num of
the source table parameter
    a_sql_uint32    source_column_number;        // source column of
the source table
} a_v4_extfn_col_subset_of_input;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
source_table_parameter_arg_num	a_sql_uint32 *	源表参数的 *arg_num*
source_column_number	a_sql_uint32 *	源表中的源列

描述
查询优化程序使用输入子集来推断输出列中的 值的逻辑属性。例如， 输入列中离散值数量为输出列中离散值数量的上限，并且 输入列上的任何本地谓词同样保持于输出列之上。

另请参见
- 描述列的类型 (a_v4_extfn_describe_col_type) （第 263 页）

Describe API

_describe_extfn 函数是 `a_v4_extfn_proc` 的组成部分。UDF 使用 `a_v4_extfn_proc_context` 对象中的 `describe_column`、`describe_parameter` 和 `describe_udf` 属性以获取和设置逻辑属性。

_describe_extfn 声明

```
void (UDF_CALLBACK *_describe_extfn)(a_v4_extfn_proc_context
*cntxt );
)
```

用法
_describe_extfn 函数向服务器描述了过程评测。

每个 `describe_column`、`describe_parameter` 和 `describe_udf` 属性都有一个相关的获取和设置方法、一组属性类型以及与之其关联的数据类型。获取方法从服务器检索信息；而设置方法则向服务器描述 UDF 的逻辑属性（例如输出列的数量或者某一输出列的离散值数量）。

另请参见
- *describe_column_get （第 194 页）
- *describe_column_set （第 209 页）
- *describe_parameter_get （第 226 页）
- *describe_parameter_set （第 245 页）
- *describe_udf_get （第 260 页）
- *describe_udf_set （第 262 页）
- 外部函数 (a_v4_extfn_proc) （第 270 页）

*describe_column_get

`describe_column_get` 第 4 版 API 方法由表 UDF 用于检索有关表参数的一列的属性。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_column_get)(
    a_v4_extfn_proc_context       *cntxt,
    a_sql_uint32                     arg_num,
    a_sql_uint32                        column_num,
    a_v4_extfn_describe_parm_type describe_type,
    void                                *describe_buffer,
    size_t                              describe_buffer_len );
```

参数

参数	描述
cntxt	此 UDF 的过程上下文对象。
arg_num	表参数的序数（0 是指结果表，1 是指第一个输入参数）。
column_num	列的序数从 1 开始。
describe_type	指示要检索的属性的选择器。
describe_buffer	对于指定的要从服务器获取的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回

成功时会返回已写入 **describe_buffer** 的字节数。如果出错或未检索到属性，则此函数会返回某个通用 `describe_column` 错误。

另请参见

- *describe_column_set （第 209 页）

*describe_column_get 的属性

以下代码中有 `describe_column_get` 第 4 版 API 方法的属性。

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
```

```
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME (Get)
EXTFNAPIV4_DESCRIBE_COL_NAME 属性指示列的名称。用于 `describe_column_get` 情形。

数据类型
`char[]`

描述
列名。该属性仅对表参数有效。

用法
如果 UDF 获取该属性，则返回指定列的名称。

返回
成功时会返回列名长度。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果缓冲区长度不足或者为 0，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – 如果该参数不是 TABLE 参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_COL_NAME（设置） （第 211 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- 通用 describe_column 错误 （第 305 页）

- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)
EXTFNAPIV4_DESCRIBE_COL_TYPE 属性指示列的数据类型。用于 describe_column_get 情形。

数据类型
a_sql_data_type

描述
列的数据类型。该属性仅对表参数有效。

用法
如果 UDF 获取该属性，则返回指定列的数据类型。

返回
成功时会返回 sizeof(a_sql_data_type)。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果描述缓冲区大小不同于 a_sql_data_type，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 如果状态不晚于初始状态，则会返回获取错误。

查询处理状态

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)
EXTFNAPIV4_DESCRIBE_COL_WIDTH 属性指示列的宽度。用于 describe_column_get 情形。

数据类型
a_sql_uint32

描述
列宽。列宽是以字节为单位的存储空间量，用于存储关联数据类型的值。该属性仅对表参数有效。

用法
如果 UDF 获取该属性，则返回由 **CREATE PROCEDURE** 语句所定义的列的宽度。

返回
成功时会返回 `sizeof(a_sql_uint32)`。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_sql_uint32`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果查询处理状态不晚于初始状态，则会返回获取错误。

查询处理状态

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_SCALE (Get)
The **EXTFNAPIV4_DESCRIBE_COL_SCALE** 属性指示列的标度。用于 `describe_column_get` 情形。

数据类型
`a_sql_uint32`

描述
列的标度。对于算术型数据类型，参数标度是指数字的小数点右边的位数。该属性仅对表参数有效。

用法
如果 UDF 获取该属性，则返回由 **CREATE PROCEDURE** 语句所定义的列的标度。此属性仅对算术数据类型有效。

返回
成功时，如果返回值则返回 `sizeof(a_sql_uint32)`，或者

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 如果列的标度对于指定列的数据类型不可用，则会返回获取错误。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_sql_uint32`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果查询处理状态不晚于初始状态，则会返回获取错误。

查询处理状态

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- EXTFNAPIV4_DESCRIBE_COL_SCALE（设置） （第 214 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Get)
The **EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL** 属性指示列是否可以为空。用于 `describe_column_get` 情形。

数据类型
`a_sql_byte`

描述
如果列可以是空值，则该值为“真”。该属性仅对表参数有效。该属性仅对参数 0 有效。

用法
当某个 UDF 获取该属性时，如果列可以为 NULL 则返回 1，否则返回 0。

返回
成功时，如果具有该属性，则返回 `sizeof(a_sql_byte)`，如果未获取该属性，则返回

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE。如果查询不涉及该列则可能返回。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果描述缓冲区大小不同于 a_sql_byte，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 如果指定的参数是一个输入表，并且查询处理状态不晚于计划构建状态，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL（设置） （第 215 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Get)
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES 属性用于描述某一列的离散值。用于 describe_column_get 情形。

数据类型
a_v4_extfn_estimate

描述
某一列的估计离散值数量。该属性仅对表参数有效。

用法
如果 UDF 获得了该属性，则会返回某一列的估计离散值数量。

返回
成功时，如返回某一值，则返回 sizeof(a_v4_extfn_estimate)，或者：

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – 如果无法获取该属性，则返回该参数。如果查询不涉及该列则可能返回。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果描述缓冲区大小不同于 a_v4_extfn_estimate，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 如果指定的参数是一个输入表，并且查询处理状态晚于优化状态，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 计划构建状态
- 执行状态

示例
考虑 _describe_extfn API 函数中的过程定义及代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
RESULTS ( r1 INT, r2 INT, r3 INT )
EXTERNAL ‘my_tpf_proc@mylibrary’;
```

```
CREATE TABLE T( x INT, y INT, z INT );
```

```
select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

本示例显示了 TPF 如何获取输入表中某个列 的离散值数量。如果有益于选择合适的处理算法，则 TPF 可能需要得到 该值。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_PLAN_BUILDING ) {
            a_v4_extfn_estimate num_distinct;

            a_sql_int32 ret = 0;

          // Get the number of distinct values expected from the first
column
        // of the table input parameter 'col_table'
            ret = cntxt->describe_column_get( cntxt, 2, 1
                        EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
                        &num_distinct,
                        sizeof(a_v4_extfn_estimate) );

            // default algorithm is 1
            _algorithm = 1;

            if( ret > 0 ) {
                // choose the best algorithm for sample size.

                if ( num_distinct.value < 100 ) {
                    // use faster algorithm for small distinct values.
                        _algorithm = 2;
                }
            }
            else {
                if ( ret < 0 ) {
                        // Handle the error
                        //   or continue with default algorithm
                } else {
                        // Attribute was unavailable
                        // We will use the default algorithm.
```

```
                }
            }
    }
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES（设置） （第 216 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Get)

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE 属性指示某列在表中是唯一的。用于 `describe_column_get` 情形。

数据类型

`a_sql_byte`

描述

如果表中的列是惟一的，则该值为“真”。该属性仅对表参数有效。

用法

当 UDF 获取该属性时，如果列具有唯一性，则返回 1，否则返回 0。

返回

成功时会返回 `sizeof(a_sql_byte)` 或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 如果无法获取该属性。如果查询不涉及该列则可能返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_sql_byte`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果查询处理状态不晚于初始状态，则会返回获取错误。

查询处理状态

在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE（设置） （第 217 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get)
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT 属性指示某一列是否是常量。用于 `describe_column_get` 情形。

数据类型
`a_sql_byte`

描述
如果该列在语句的生存期内保持不变，则为“真”。该属性仅对输入的表参数有效。

用法
当 UDF 获取该属性时，如果列在语句的生存期内保持不变，则返回值为 1，否则返回 0。如果输入表的选择列表中的列为常量表达式或者 NULL，则输入表的列保持不变。

返回
成功时会返回 `sizeof(a_sql_byte)`，如果返回该值，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 无法获取该属性。如果查询不涉及该列，则返回该参数。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_sql_byte`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 如果指定的参数不是一个输入表，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT（设置） （第 218 页）
- 通用 describe_column 错误 （第 305 页）

- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get)
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE 属性表示列的常量值。用于 `describe_column_get` 情形。

数据类型
`an_extfn_value`

描述
如果列在语句的生存期内保持不变，则为列值。如果该列的 `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT` 返回为“真”，则该值可用。该属性仅对表参数有效。

用法
对于输入表中包含常量值的列，返回该值。如果该值不可用，则返回 NULL。

返回
成功时会返回 `sizeof(a_sql_byte)`，如果返回该值，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 无法获取该属性。如果查询不涉及该列，或者该值未被视为常量，则返回该参数。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_sql_byte` 则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 如果指定的参数不是一个输入表，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE（设置） （第 218 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Get)
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER 属性指示结果表中的某列是否为用户所使用。用于 `describe_column_get` 情形。

数据类型
`a_sql_byte`

描述
或者用于确定消耗程序是否用到结果表中的某个列，或者用于表明不需要输入表中的某个列。对于表参数有效。允许用户设置或检索关于单列的信息，而类似的属性 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` 则设置或检索关于一次调用中涉及的所有列的信息。

用法
UDF 查询该属性，以确定结果表中的某列是否为用户所需。这有助于 UDF 避免对未使用的列执行不必要的操作。

返回
成功时会返回 `sizeof(a_sql_byte)`，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 如果无法获取该属性。如果查询不涉及该列则可能返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果描述缓冲区大小不同于 `a_v4_extfn_estimate`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 如果指定的参数并非参数 0，则会返回获取错误。

查询处理状态
有效于：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

在 `_describe_extfn` API 函数中的过程定义及代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
```

```
      EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

当运行该 TPF 时，了解用户是否已经 选定了结果集中的 `r1` 列很有帮助。如果用户不需要 `r1`，则可能 不必计算 `r1`，我们也不必为服务器 生成该值。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 0, 1,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER（设置） （第 219 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Get)

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE 属性指示列的最小值。用于 `describe_column_get` 情形。

数据类型
`an_extfn_value`

描述
如果可用，则为列的最小值。仅对参数 0 和表参数有效。

用法
如果 UDF 获得 **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** 属性，则列数据的最小值通过 **describe_buffer** 予以返回。如果输入表为基表，则基于表中的所有列数据产生最小值，并且仅当表列具有索引时才可以访问该最大值。如果输入表为其他 UDF 的结果，则最小值为由该 UDF 所设置的 `EXTFNAPIV4_DESCRIBE_COL_TYPE`。

该属性的数据类型对于不同的列是不同的。UDF 可以使用 `EXTFNAPIV4_DESCRIBE_COL_TYPE` 确定列的数据类型。UDF 也可以根据

EXTFNAPIV4_DESCRIBE_COL_WIDTH 确定列的存储需求，以便提供相应大小的缓冲区保存该值。

describe_buffer_length 允许服务器确定缓冲区是否有效。

如果 **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** 属性不可用，则 describe_buffer 为空。

返回
成功时会返回 describe_buffer_length，或者：

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – 如果无法获取该属性。如果查询不涉及该列，或者对于所请求的列无法获得最小值，则可能发生这种情况。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果描述缓冲区的大小不足以保存最小值，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 如果状态不晚于初始状态，则会返回获取错误。

查询处理状态
在除初始状态外的任何状态中皆有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
_describe_extfn API 函数中的过程定义及代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL 'my_tpf_proc@mylibrary' ;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

本示例描述了 TPF 如何获取输入表中两列的最小值以供内部优化使用。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
      if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
           a_sql_int32 min_value = 0;
           a_sql_int32 ret = 0;

        // Get the minimum value of the second column of the
        // table input parameter 'col_table'
```

```
        ret = cntxt->describe_column_get( cntxt, 2, 2
                  EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
                  &min_value,
                  sizeof(a_sql_int32) );

     if( ret < 0 ) {
               // Handle the error.
        }

      }
}
```

另请参见

- 查询处理状态 （第 115 页）
- EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE（设置） （第 221 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- 通用 describe_column 错误 （第 305 页）

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Get)

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE 属性指示列的最大值。用于 `describe_column_get` 情形。

数据类型

`an_extfn_value`

描述

列的最大值。该属性仅对参数 0 和表参数有效。

用法

如果 UDF 获得 `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` 属性，则列数据的最大值将通过 **describe_buffer** 予以返回。如果输入表为基表，则基于表中的所有列数据产生最大值，并且仅当表列具有索引时才可以访问该最大值。如果输入表为其他 UDF 的结果，则最大值为由该 UDF 所设置的 `COL_MAXIMUM_VALUE`。

该属性的数据类型对于不同的列是不同的。UDF 可以使用 `EXTFNAPIV4_DESCRIBE_COL_TYPE` 确定列的数据类型。UDF 也可以根据 `EXTFNAPIV4_DESCRIBE_COL_WIDTH` 确定列的存储需求，以便提供相应大小的缓冲区保存该值。

describe_buffer_length 允许服务器确定缓冲区是否有效。

如果 `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` 不可用，则 `describe_buffer` 为空。

返回
成功时会返回 describe_buffer_length 或者：

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – 如果无法获取该属性。如果查询不涉及该列，或者对于所请求的列无法获得最大值，则可能发生这种情况。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果描述缓冲区的大小不足以保存最大值，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 如果状态不晚于初始状态，则会返回获取错误。

查询处理状态
在除初始状态外的任何状态中皆有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
_describe_extfn API 函数中的 **PROCEDURE** 定义和代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

本示例描述了 TPF 如何获取输入表中两列的最大值以供内部优化使用。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
         a_sql_int32 max_value = 0;
         a_sql_int32 ret = 0;

         // Get the maximum value of the second column of the
         // table input parameter  'col_table'
         ret = cntxt->describe_column_get( cntxt, 2, 2
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
            }
```

```
        }
}
```

另请参见

- 查询处理状态 （第 115 页）
- EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE（设置） （第 222 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- 通用 describe_column 错误 （第 305 页）

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get)

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT 属性为输入行中指定的值设置子集。在 `describe_column_get` 情形中使用该属性将返回错误。

数据类型

`a_v4_extfn_col_subset_of_input`

描述

列值是是输入列中所指定的值的子集。

用法

仅可对此属性进行设置。

返回

返回错误参数 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE`。

查询处理状态

在任何状态皆返回错误参数 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE`。

另请参见

- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set) （第 224 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

*describe_column_set

`describe_column_set` 第 4 版 API 方法可在服务器中设置 UDF 列级属性。

说明

列级属性用于描述结果集或 TPF 输入表中的列的各种特性。例如，UDF 可告知服务器其结果集中的一列仅会有十个非重复值。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_column_set)(
    a_v4_extfn_proc_context        *cntxt,
    a_sql_uint32        arg_num,
    a_sql_uint32        column_num,
    a_v4_extfn_describe_udf_type        describe_type,
    const void        *describe_buffer,
    size_t        describe_buffer_len );
```

参数

参数	描述
cntxt	此 UDF 的过程上下文对象。
arg_num	表参数的序数（0 是指结果表，1 是指第一个输入参数）。
column_num	列的序数从 1 开始。
describe_type	指示要设置的属性的选择器。
describe_buffer	对于指定的要在服务器中设置的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回

成功时会返回已写入 **describe_buffer** 的字节数。如果出错或未检索到属性，则此函数会返回某个通用 `describe_column` 错误。

另请参见

- *describe_column_get （第 194 页）

***describe_column_set 的属性**

以下代码中有 `describe_column_set` 属性。

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME（设置）
EXTFNAPIV4_DESCRIBE_COL_NAME 属性指示列名。用在 `describe_column_set` 情形中。

数据类型
`char[]`

描述
列名。该属性仅对表参数有效。

用法
对于参数 0，如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的列名。这种比较可以确保 **CREATE PROCEDURE** 语句的列名和 UDF 预期列名相同。

返回
成功时会返回列名长度。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不同于标注，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — 如果参数不是表参数，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE` — 如果输入列名长于 128 个字符，或者输入列名和存储在目录中的列名不一致，则会返回设置错误。

查询处理状态

- 标注状态

示例

```
short desc_rc = 0;
     char name[7] = ‘column1’;
     // Verify that the procedure was created with the second column
of the result table as an int
     if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
          desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_NAME,
                                                                name,
                                                      sizeof(name) );
          if( desc_rc < 0 ) {
          // handle the error.
          }
     }
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_NAME (Get) （第 195 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_TYPE（设置）

`EXTFNAPIV4_DESCRIBE_COL_TYPE` 属性指示列的数据类型。用在 `describe_column_set` 情形中。

数据类型

`a_sql_data_type`

描述

列的数据类型。该属性仅对表参数有效。

用法

对于参数零，如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的列数据类型。UDF 借此可确保 **CREATE PROCEDURE** 语句的数据类型与 UDF 预期数据类型相同。

返回

成功时会返回 `a_sql_data_type`。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_sql_data_type`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不同于标注，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE` — 如果输入数据类型和存储在目录中的数据类型不一致，则会返回设置错误。

查询处理状态

- 标注状态

示例

```
short desc_rc = 0;
a_sql_data_type type = DT_INT;

   // Verify that the procedure was created with the second column of
the result table as an int
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
```

```
EXTFNAPIV4_DESCRIBE_COL_TYPE,
        &type,
                sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

另请参见

- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置）

EXTFNAPIV4_DESCRIBE_COL_WIDTH 属性指示列的宽度。用在 `describe_column_set` 情形中。

数据类型

`a_sql_uint32`

描述

列宽。列宽是以字节为单位的存储空间量，用于存储关联数据类型的值。该属性仅对表参数有效。

用法

如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的列的宽度。UDF 借此可确保 **CREATE PROCEDURE** 语句的列宽与 UDF 预期列宽相同。

返回

成功时会返回 `sizeof(a_sql_uint32)`。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_sql_uint32`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果查询处理状态不同于标注，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` — 如果输入宽度和存储在目录中的宽度不一致，则会返回设置错误。

查询处理状态

在以下状态下有效：

- 标注状态

另请参见

- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_SCALE（设置）
EXTFNAPIV4_DESCRIBE_COL_SCALE 属性指示列的标度。用在 `describe_column_set` 情形中。

数据类型
`a_sql_uint32`

描述
列的标度。对于算术型数据类型，参数标度是指数字的小数点右边的位数。该属性仅对表参数有效。

用法
如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的列的标度。UDF 借此可确保 **CREATE PROCEDURE** 语句的列宽与 UDF 预期列宽相同。此属性仅对算术数据类型有效。

返回
成功时会返回 `sizeof(a_sql_uint32)`，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果没有指定的列的数据类型的标度，则会返回设置错误。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_sql_uint32`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果查询处理状态不同于标注，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` — 如果输入标度和存储在目录中的标度不一致，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 标注状态

示例

```
    short desc_rc = 0;
    a_sql_uint32 scale = 0;

            // Verify that the procedure has a scale of zero for the
```

```
second result table column.
        if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_SCALE,
            &scale,
            sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_SCALE (Get) （第 197 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL（设置）

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL 属性指示列是否可以为空。用在 `describe_column_set` 情形中。

数据类型

`a_sql_byte`

描述

如果列可以是空值，则该值为“真”。该属性仅对表参数有效。该属性仅对参数 0 有效。

用法

对于结果表列，如果可以是空列，则 UDF 可设置此属性。如果 UDF 不特意设置此属性，则会假定列可以是空列。服务器可在优化状态下使用这些信息。

返回

成功时会返回 `sizeof(a_sql_byte)`（如果已设置该属性），或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果该属性不能设置；如果查询不涉及该列则可能返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_sql_byte`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不是优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` — 如果 UDF 尝试对此属性设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
在以下状态下有效:

- 优化状态

另请参见
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Get) （第 198 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES（设置）
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES 属性用于描述列的非重复值。用在 `describe_column_set` 情形中。

数据类型
`a_v4_extfn_estimate`

描述
某一列的估计离散值数量。该属性仅对表参数有效。

用法
如果 UDF 知道其结果表中的一列能有多少非重复值，则 UDF 可设置此属性。服务器可在优化状态下使用这些信息。

返回
成功时会返回 `sizeof(a_v4_extfn_estimate)`（如果 UDF 设置该值），或者:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果该属性不能设置。如果查询不涉及该列则可能返回。

失败时会返回:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_v4_extfn_estimate`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不是优化状态，则会返回设置错误。

查询处理状态
在以下状态下有效:

- 优化状态

另请参见
- EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Get) （第 199 页）

- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE（设置）
EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE 属性指示列在表中是否为唯一的列。用在 `describe_column_set` 情形中。

数据类型
`a_sql_byte`

描述
如果表中的列是惟一的，则该值为“真”。该属性仅对表参数有效。

用法
如果 UDF 知道结果表列值是唯一的值，则 UDF 可设置此属性。服务器可在优化状态下使用这些信息。UDF 仅可对参数 0 设置此属性。

返回
成功时会返回 `sizeof(a_sql_byte)`，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果该属性不能设置。如果查询不涉及该列则可能返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区大小不同于 `a_sql_byte`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果查询处理状态不是优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 **arg_num** 不是零，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` — 如果 UDF 尝试对此属性设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

另请参见
- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Get) （第 201 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT（设置）
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT 属性指示列是否为常量。用在 `describe_column_set` 情形中。

数据类型
`a_sql_byte`

描述
如果该列在语句的生存期内保持不变，则为“真”。该属性仅对输入的表参数有效。

用法
这是一个只读属性。所有设置该属性的尝试都会返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE`。

返回

- `EXTFNAPIV4_DESCRIBE_INVALID ATTRIBUTE` — 这是一个只读属性；所有设置尝试都会返回此错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不是优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 **arg_num** 不是零，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` — 如果 UDF 尝试对此属性设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
不适用。

另请参见

- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get) （第 202 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE（设置）
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE 属性指示列的常量值。用在 `describe_column_set` 情形中。

数据类型
`an_extfn_value`

描述
如果列在语句的生存期内保持不变，则为列值。如果该列的 EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT 返回为“真”，则该值可用。该属性仅对表参数有效。

用法
此属性是只读属性。

返回

- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE — 这是一个只读属性；所有设置尝试都会返回此错误。

查询处理状态
不适用。

另请参见

- 通用 describe_column 错误 （第 305 页）
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get) （第 203 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER（设置）
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER 属性指示消耗程序是否使用结果表中的列。用在 describe_column_set 情形中。

数据类型
a_sql_byte

描述
或者用于确定消耗程序是否用到结果表中的某个列，或者用于表明不需要输入表中的某个列。对于表参数有效。允许用户设置或检索关于单列的信息，而类似的属性 EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 则设置或检索关于一次调用中涉及的所有列的信息。

用法
UDF 会对输入表中的列设置 **EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER**，以告知生产者 UDF 不需要该列的值。

返回
成功时会返回 sizeof(a_sql_byte)，或者：

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE — 如果该属性不能设置。如果查询不涉及该列则可能返回。

失败时会返回某个通用 describe_column 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果描述缓冲区大小不同于 a_v4_extfn_estimate，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果指定的参数是参数 0，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是优化状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 要设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

_describe_extfn API 函数中的 PROCEDURE 定义和代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
      EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

运行此 TPF 时有益于服务器了解此 TPF 是否使用 列 y。 如果该 TPF 不需要 y，则服务器可用所了解的这种情况 进行优化，不向 TPF 发送 这些列信息。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 2, 2,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Get) （第 204 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE（设置）

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE 属性用于对列指定最小值。用在 `describe_column_set` 情形中。

数据类型

`an_extfn_value`

描述

是列能有的最小值（如果有）。仅对参数 0 有效。

用法

如果 UDF 知道列的最小数据值是什么，则 UDF 可设置 `EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE`。服务器可在优化过程中使用这些信息。

UDF 可用 `EXTFNAPIV4_DESCRIBE_COL_TYPE` 确定列的数据类型，也可用 `EXTFNAPIV4_DESCRIBE_COL_WIDTH` 确定列的存储要求，以便提供大小调整得相当的缓冲区，以存储要设置的值。

返回

成功时会返回 `describe_buffer_length`，或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果该属性不能设置。如果查询不涉及该列，或者所请求的列未提供最小值，则返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区不够大，不能存储最小值，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不是优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 **arg_num** 不是 0，则会返回设置错误。

查询处理状态

在以下状态下有效：

- 优化状态

示例

用于实现 `_describe_extfn` 回调 API 函数的 **PROCEDURE** 定义和 UDF 代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

此例所示的是可从中将其用于服务器（或用于另一可接收此 TPF 的结果，将其用作输入的 TPF）的 TPF，以了解第一个结果集列的最小值。在此例中，第一列的最小输出值是 27。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_int32 min_value = 27;
    a_sql_int32 ret = 0;

// Tell the server what the minimum value of the first column
// of our result set will be.

    ret = cntxt->describe_column_set( cntxt, 0, 1
        EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
        &min_value,
        sizeof(a_sql_int32) );

    if( ret < 0 ) {
        // Handle the error.
    }
}
}
```

另请参见

- 查询处理状态 （第 115 页）
- EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Get) （第 205 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- 通用 describe_column 错误 （第 305 页）

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE（设置）

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE 属性用于对列指定最大值。用在 `describe_column_set` 情形中。

数据类型

`an_extfn_value`

描述
列的最大值。该属性仅对参数 0 和表参数有效。

用法
如果 UDF 知道列的最大数据值是什么，则 UDF 可设置 `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE`。服务器可在优化过程中使用这些信息。

UDF 可用 `EXTFNAPIV4_DESCRIBE_COL_TYPE` 确定列的数据类型，也可用 `EXTFNAPIV4_DESCRIBE_COL_WIDTH` 确定列的存储要求，以便提供大小调整得相当的缓冲区，以存储要设置的值。

`describe_buffer_length` 是此缓冲区的 `sizeof()`。

返回
成功时会返回 `describe_buffer_length`（如果已设置值），或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` — 如果该属性不能设置。如果查询不涉及该列，或者所请求的列未提供最大值，则返回。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果描述缓冲区不够大，不能存储最大值，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果查询处理状态不是优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 **arg_num** 不是 0，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

示例
用于实现 `_describe_extfn` 回调 API 函数的 PROCEDURE 定义和 UDF 代码段：

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL  'my_tpf_proc@mylibrary' ;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

此例所示的是可从中将其用于服务器（或用于另一可接收此 TPF 的结果，将其用作输入的 TPF）的 TPF，以了解第一个结果集列的最大值。在此例中，第一列的最大输出值是 500000。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
         a_sql_int32 max_value = 500000;
         a_sql_int32 ret = 0;

        // Tell the server what the maximum value of the first column
         // of our result set will be.

      ret = cntxt->describe_column_set( cntxt, 0, 1
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
         }

   }
}
```

另请参见

- 查询处理状态 （第 115 页）
- EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Get) （第 207 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_TYPE（设置） （第 212 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get) （第 196 页）
- EXTFNAPIV4_DESCRIBE_COL_WIDTH（设置） （第 213 页）
- 通用 describe_column 错误 （第 305 页）

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set)

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT 属性为输入行中指定的值设置子集。用于 `describe_column_set` 情形。

数据类型
`a_v4_extfn_col_subset_of_input`

描述
列值是是输入列中所指定的值的子集。

用法
设置此描述属性将通知查询优化程序，所指列值是输入列中指定值的子集。例如，考虑一个 TPF 过滤器，该 TPF 消耗表资源，并且基于函数对行进行过滤。在此种情况

下，返回表为输入表的子集。为 TPF 过滤器设置 **EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT** 属性以优化查询。

返回
成功时会返回 `sizeof(a_v4_extfn_col_subset_of_input)`。

失败时会返回某个通用 `describe_column` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果缓冲区长度小于 `sizeof (a_v4_extfn_col_subset_of_input)`，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – 如果源表的列索引超出了范围，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 如果进行设置的 `subset_of_input` 列不适用（例如，如果该列不在选择列表中），则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果查询处理并非处于优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 如果缓冲区长度小于 0，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 如果调用的是参数而非返回表，则设置返回的错误。

查询处理状态
在以下状态下有效：

- 优化状态

示例

```
a_v4_extfn_col_subset_of_input colMap;

    colMap.source_table_parameter_arg_num = 4;
    colMap.source_column_number = i;

    desc_rc = ctx->describe_column_set( ctx,
        0, i,
        EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
        &colMap, sizeof(a_v4_extfn_col_subset_of_input) );
```

另请参见

- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get) （第 209 页）
- 通用 describe_column 错误 （第 305 页）
- 查询处理状态 （第 115 页）

*describe_parameter_get

`describe_parameter_get` 第 4 版 API 方法可从服务器中获取 UDF 参数属性。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                             describe_buffer_len );
```

参数

参数	描述
cntxt	过程上下文对象。
arg_num	表参数的序数（0 是指结果表，1 是指第一个输入参数）
describe_type	指示要设置的属性的选择器。
describe_buffer	对于指定的要在服务器中设置的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回

成功时会返回 0 或已写入 **describe_buffer** 的字节数。值为 0 表明服务器不能获取属性，但没出错。如果出错或未检索到属性，则此函数会返回某个通用 **describe_parameter** 错误。

*describe_parameter_get 的属性

以下代码中有 `describe_parameter_get` 属性。

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
```

```
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

} a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（获取）
EXTFNAPIV4_DESCRIBE_PARM_NAME 属性指示参数名称。用在 describe_parameter_get 情形中。

数据类型
char[]

描述
UDF 的参数名称。

用法
用于获取 **CREATE PROCEDURE** 语句中所定义的参数名称。对于参数 0 无效。

返回
成功时会返回参数名称长度。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 不够大，不能存储参数名称，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE —如果状态不晚于初始状态，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果参数是结果表，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（设置） （第 246 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取）
EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性在 `describe_parameter_get` 情形下用于返回数据类型。

数据类型
`a_sql_data_type`

描述
UDF 的参数数据类型。

用法
用于获取 **CREATE PROCEDURE** 语句中所定义的参数数据类型。

返回
成功时会返回 `sizeof(a_sql_data_type)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 不是 `sizeof(a_sql_data_type)`，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于初始状态，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性指示参数的宽度。用在 `describe_parameter_get` 情形中。

数据类型
`a_sql_uint32`

描述
UDF 的参数宽度。EXTFNAPIV4_DESCRIBE_PARM_WIDTH 仅适用于标量参数。参数宽度是以字节为单位的存储空间量，用于存储关联数据类型的参数。

- **定长数据类型** – 存储数据所需的字节。
- **变长数据类型** – 长度上限。
- **LOB 数据类型** – 将句柄存入数据所需的存储空间量。
- **TIME 数据类型** – 存储经过编码的时间所需的存储空间量。

用法
用于获取 **CREATE PROCEDURE** 语句中所定义的参数宽度。

返回
成功时会返回 sizeof(a_sql_uint32)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果查询处理状态不晚于初始状态，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_uint32 的大小不相同，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果指定的参数是表参数，则会返回获取错误。这包括参数 0 或参数 *n*，其中的 *n* 是输入表。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
示例过程定义：

```
CREATE PROCEDURE my_udf(IN p1 INT, IN p2 char(100))
RESULT (x INT)
EXTERNAL NAME ‘my_udf@myudflib’;
```

示例 _describe_extfn API 函数代码段：

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_uint32 width = 0;
    a_sql_int32 ret = 0;

    // Get the width of parameter 1
```

```
    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    &width,
    sizeof(a_sql_uint32) );

    if( ret < 0 ) {
        // Handle the error.
    }

    //Allocate some storage based on parameter width
    a_sql_byte *p = (a_sql_byte *)cntxt->alloc( cntxt, width )

    // Get the width of parameter 2
    ret = cntxt->describe_parameter_get( cntxt, 2,
        EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
        &width,
        sizeof(a_sql_uint32) );
    if( ret <= 0 ) {
        // Handle the error.
    }

    // Allocate some storage based on parameter width
    char *c = (char *)cntxt->alloc( cntxt, width )

    …

}
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性（设置） （第 247 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute（获取）

EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性指示参数的标度。用在 `describe_parameter_get` 情形中。

数据类型

`a_sql_uint32`

描述

UDF 的参数标度。对于算术型数据类型，参数标度是指数字的小数点右边的位数。

此属性对于以下数据无效：

- 非算术数据类型
- 表参数

用法

用于获取 **CREATE PROCEDURE** 语句中所定义的参数标度。

返回
成功时会返回 (a_sql_uint32) 的大小。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_uint32 的大小不相同，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不晚于初始状态，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果指定的参数是表参数，则会返回获取错误。这包括参数 0 或参数 *n*，其中的 *n* 是输入表。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
用于获取参数 1 的标度的示例 _describe_extfn API 函数代码段：

```
if( cntxt->current_state > EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_uint32 scale = 0;
    a_sql_int32 ret = 0;

    ret = ctx->describe_parameter_get( ctx, 1,
EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    &scale, sizeof(a_sql_uint32) );

    if( ret <= 0 ) {
    // Handle the error.
    }
}
```

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性（设置） （第 248 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性指示参数是否为空。用在 describe_parameter_get 情形中。

数据类型
a_sql_byte

描述
如果执行时参数值可以是空值，则该值为“真”。对于表参数或者参数 0，该值为“假”。

用法
获取查询执行期间指定的参数是否可以为空。

返回
成功时会返回 `sizeof(a_sql_byte)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_sql_byte` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于计划构建状态，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 执行状态

示例：EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL (Get)
示例过程定义，`_describe_extfn` API 函数代码段，以及用于获取 **EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL** 值的 SQL 查询。

过程定义
本主题中示例查询所用的示例过程定义：

```
CREATE PROCEDURE my_udf(IN p INT)
RESULT (x INT)
EXTERNAL NAME ‘my_udf@myudflib’;
```

API 函数代码段
本主题中示例查询所用的示例 `_describe_extfn` API 函数代码段：

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{

    if( cntxt->current_state > EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_byte can_be_null = 0;
    a_sql_int32 ret = 0;

    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    &can_be_null,
    sizeof(a_sql_byte) );

    if( ret <= 0 ) {
```

```
        // Handle the error.
    }
}
}
```

示例 1：未使用 NOT NULL
本示例所创建的表包含单个整数列，且没有指定 **NOT NULL** 修饰符。相关子查询传入 `has_nulls` 表中的 *c1* 列。当在执行状态中调用过程 **my_udf_describe** 时，对 `describe_parameter_get` 的调用将为 **can_be_null** 赋值 1。

```
CREATE TABLE has_nulls ( c1 INT );
INSERT INTO has_nulls VALUES(1);
INSERT INTO has_nulls VALUES(NULL);
SELECT * from has_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(has_nulls.c1)) > 0;
```

示例 2：使用 NOT NULL
本示例所创建的表包含单个整数列，且没有指定 **NOT NULL** 修饰符。相关子查询传入 `no_nulls` 表中的 *c1* 列。当在执行状态中调用过程 **my_udf_describe** 时，对 `describe_parameter_get` 的调用将为 **can_be_null** 赋值 0。

```
CREATE TABLE no_nulls ( c1 INT NOT NULL);
INSERT INTO no_nulls VALUES(1);
INSERT INTO no_nulls VALUES(2);
SELECT * from no_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(no_nulls.c1)) > 0;
```

示例 3：使用常量
本示例使用常量调用过程 **my_udf**。当在执行状态中调用过程 **my_udf_describe** 时，对 `describe_parameter_get` 的调用将为 **can_be_null** 赋值 0。

```
SELECT * from my_udf(5);
```

示例 4：使用 NULL
本示例使用 NULL 调用过程 **my_udf**。当在执行状态中调用过程 **my_udf_describe** 时，对 `describe_parameter_get` 的调用将为 **can_be_null** 赋值 1。

```
SELECT * from my_udf(NULL);
```

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性用于返回非重复值数量。用在 `describe_parameter_get` 情形中。

数据类型
`a_v4_extfn_estimate`

描述
返回所有调用中的估计离散值数量。仅对标量参数有效。

用法
如果有这些信息，则 UDF 会返回估计的非重复值数量，可信度为 100%。如果没有这些信息，则 UDF 会返回估计的数量 0，可信度为 0%。

返回
成功时会返回 `sizeof(a_v4_extfn_estimate)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_v4_extfn_estimate` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果参数是表参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
示例 `_describe_extfn` API 函数代码段：

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_est.value = 0.0;
    desc_est.confidence = 0.0;

    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
        &desc_est, sizeof(a_v4_extfn_estimate) );
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性（设置） （第 250 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性用于返回参数是否为常量。用在 describe_parameter_get 情形中。

数据类型
a_sql_byte

描述
如果语句的参数为常量，则该值为“真”。仅对标量参数有效。

用法
如果指定的参数的值不是常量，则会返回 0；如果指定的参数的值是常量，则会返回 1。

返回
成功时会返回 sizeof(a_sql_byte)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_byte 的大小不相同，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不晚于初始状态，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果参数是表参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
示例 _describe_extfn API 函数代码段：

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
        &desc_byte, sizeof( a_sql_byte ) );
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性（设置） （第 250 页）

- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性指示参数值。用在 `describe_parameter_get` 情形中。

数据类型
`an_extfn_value`

描述
如果在描述时是已知的，则为参数值。仅对标量参数有效。

用法
用于返回参数值。

返回
成功时会返回 `sizeof(an_extfn_value)`（如果有值），或者：

- `EXTFNAPIV4_DESCRIBE_NOT_AVILABLE` — 如值不是常量则会返回值。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `an_extfn_value` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果参数是表参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

示例
示例 `_describe_extfn` API 函数代码段：

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_int32 desc_rc;
    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
```

```
        &arg,
        sizeof( an_extfn_value ) );
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性（设置） （第 250 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute（获取）
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS` 属性指示表中的列数。用在 `describe_parameter_get` 情形中。

数据类型
`a_sql_uint32`

描述
表中的列数。仅对参数 0 和表参数有效。

用法
返回指定的表参数中的列数。参数 0 用于返回结果表中的列数。

返回
成功时会返回 `sizeof(a_sql_uint32)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `size of a_sql_uint32` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — 如果参数不是表参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性（设置） （第 252 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute（获取）
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 属性指示表中的行数。用在 `describe_parameter_get` 情形中。

数据类型
`a_v4_extfn_estimate`

描述
表中的估计行数。仅对参数 0 和表参数有效。

用法
用于在指定的表参数或结果集中返回估计的行数，可信度为 100%。

返回
成功时会返回 `a_v4_extfn_estimate` 的大小。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_v4_extfn_estimate` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于初始状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — 如果参数不是表参数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性（设置） （第 253 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute（获取）
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY` 属性指示表中行的顺序。用在 `describe_parameter_get` 情形中。

数据类型
`a_v4_extfn_orderby_list`

描述
表中各行的次序。该属性仅对参数 0 和表参数有效。

用法
通过此属性可用 UDF 代码：

- 确定输入 **TABLE** 参数是否已经过排序
- 声明结果集已经过排序

如果参数数量是 0，则该属性是指外发结果集。如果该参数 > 0，并且参数类型是表参数，则该属性是指输入 **TABLE** 参数。

顺序由 `a_v4_extfn_orderby_list`（是一个支持列序数列表的结构）及与其相关联的升序或降序属性指定。如果 UDF 通过属性对外发结果集设置顺序，则服务器就能通过优化进行排序。例如，如果 UDF 已对第一个结果集列生成升序顺序，则服务器会通过对同一列发出请求消除冗余顺序。

如果 UDF 不对外发结果集设置 orderby 属性，则服务器会假定数据未经排序。

如果 UDF 对输入 **TABLE** 参数设置 orderby 属性，则服务器保证会对输入数据进行排序。在这种情况下，UDF 会向服务器作出输入数据必须经过排序的描述。如果服务器检测到运行时冲突，则会报告 SQL 异常。例如，如果 UDF 作出这样的描述，即输入 **TABLE** 参数第一列的顺序必须是升序，但 SQL 语句含有降序子句，则服务器会报告 SQL 异常。

如果 SQL 不含排序子句，则服务器会自动进行排序，以确保按要求对输入 **TABLE** 参数排序。

返回
如果成功则会返回从 `a_v4_extfn_orderby_list` 复制的字节数。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性（设置） （第 254 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 属性指示 UDF 需要分区。用于 describe_parameter_get 情形。

数据类型

a_v4_extfn_column_list

描述

UDF 开发者使用 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** 以编程方式声明，调用之前须先对 UDF 进行分区。

用法

UDF 可以查询分区以强制执行，或者动态调整分区。UDF 负责分配 **a_v4_extfn_column_list**，需要考虑输入表的总列数，并将其发送至服务器。

返回

成功时会返回 a_v4_extfn_column_list 的大小。该值等于：

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果缓冲区的长度小于预期大小，则会返回获取错误。

查询处理状态

在以下状态下有效：

- 查询优化状态
- 计划构建状态
- 执行状态

示例

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
        a_sql_uint32    col_count   = 0;
        a_sql_uint32    buffer_size = 0;
        a_v4_extfn_column_list    *clist = NULL;

        col_count = 3;    // Set to the max number of possible pby
```

```
columns

       buffer_size = sizeof( a_v4_extfn_column_list ) + (col_count -
1) * sizeof( a_sql_uint32 );

        clist = (a_v4_extfn_column_list *)ctx->alloc( ctx,
buffer_size );

        clist->number_of_columns = 0;
        clist->column_indexes[0] = 0;
        clist->column_indexes[1] = 0;
        clist->column_indexes[2] = 0;

    args->r_api_rc = ctx->describe_parameter_get( ctx,
        args->p3_arg_num,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    clist,
    buffer_size );
}
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set) （第 255 页）
- 通用 describe_parameter 错误 （第 306 页）
- 第 4 版 API describe_parameter 和 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY （第 133 页）
- 使用 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 的并行 TPF PARTITION BY 示例 （第 135 页）
- 查询处理状态 （第 115 页）
- 输入数据分区 （第 133 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND` 属性指示消耗程序请求回绕输入表。用在 `describe_parameter_get` 情形中。

数据类型
`a_sql_byte`

描述
表示消耗程序想要回绕输入表。仅对表输入参数有效。缺省情况下，该属性为“假”。

用法
UDF 会查询此属性以检索 true/false 值。

返回
成功时会返回 `sizeof(a_sql_byte)`。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_byte 的大小不相同，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是优化或计划构建状态，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果 UDF 尝试通过参数 0 获取此属性，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER — 如果 UDF 尝试通过非表参数获取此属性，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 优化状态
- 计划构建状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）（第 258 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）（第 242 页）
- _rewind_extfn （第 301 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）
EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性指示参数是否支持回绕。用在 describe_parameter_get 情形中。

数据类型
a_sql_byte

描述
表示生产者是否支持回绕。仅对表参数有效。

如果打算将 DESCRIBE_PARM_TABLE_HAS_REWIND 设置为 true，还必须提供回绕表回调实现方案 (**_rewind_extfn()**)。如果未提供回调方法，则服务器将不能执行 UDF。

用法
UDF 会询问表输入参数是否支持回绕。UDF 必须先通过 **DESCRIBE_PARM_TABLE_REQUEST_REWIND** 请求回绕（这是一个前提条件），然后才能使用此属性。

返回
成功时会返回 `sizeof(a_sql_byte)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_sql_byte` 的大小不相同，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不大于标注，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — 如果 UDF 尝试通过非表参数获取此属性，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 UDF 尝试通过结果表获取此属性，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）（第 241 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）（第 258 页）
- _rewind_extfn （第 301 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute（获取）
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` 属性用于列出未被消耗的列。用在 `describe_parameter_get` 情形中。

数据类型
`a_v4_extfn_column_list`

描述
服务器或者 UDF 将不会使用的输出表列的列表。

对于输出表参数，UDF 通常会对所有列生成数据，而服务器会消耗所有列。对于输入表参数，情况也是如此，服务器通常会对所有列生成数据，而 UDF 会消耗所有列。

但是，某些情况下服务器或 UDF 可能不会消耗所有列。这种情况下，最佳做法是让 UDF 对描述属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS** 执行输出表 **GET**。此操作可查询服务器，以列出不会被服务器消耗的输出表列。然后 UDF 就可以在填充输出表列数据时使用列出的输出表列，也就是说，UDF 不会尝试对未使用的列填充数据。

总之，对于输出表，UDF 会轮询未使用的列的列表。对于输出表，UDF 会推送未使用的列的列表。

用法
UDF 会询问服务器是否要使用所有输出表列。UDF 必须分配包括所有输出表列的 `a_v4_extfn_column_list`，然后还必须将其传递给服务器。服务器随后将所有估计不会用到的列的序数标为 1。生成数据时可使用服务器返回的列表。

返回
成功时会返回列列表大小：`sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不晚于计划构建状态，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 不够大，不能存储返回的列表，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果 UDF 尝试通过输入表获取此属性，则会返回获取错误。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — 如果 UDF 尝试通过非表参数获取此属性，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set) (第 259 页)

*describe_parameter_set

`describe_parameter_set` 第 4 版 API 方法可以设置有关一个 UDF 参数的属性。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                              describe_buffer_len );
```

参数

参数	描述
cntxt	过程上下文对象。
arg_num	表参数的序数（0 是指结果表，1 是指第一个输入参数）
describe_type	指示要设置的属性的选择器。
describe_buffer	对于指定的要在服务器中设置的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回

成功时会返回 0 或已写入 **describe_buffer** 的字节数。值为 0 表明服务器不能设置属性，但没出错。如果出错或未检索到属性，则此函数会返回某个通用 **describe_parameter** 错误。

*describe_parameter_set 的属性

以下代码中有 `describe_parameter_set` 属性。

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
```

```
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    } a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_NAME 属性指示参数名称。用在 describe_parameter_set 情形中。

数据类型
char[]

描述
UDF 的参数名称。

用法
如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的参数的名称。如果两个值不一致，则服务器会返回错误。UDF 借此可确保 **CREATE PROCEDURE** 语句的参数名称与 UDF 预期参数名称相同。

返回
成功时会返回参数名称长度。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是标注状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果参数是结果表，则会返回设置错误。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试重置参数名称，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 标注状态

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性（获取） （第 227 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性指示参数的数据类型。用在 describe_parameter_set 情形中。

数据类型
a_sql_data_type

描述
UDF 的参数数据类型。

用法
如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的参数类型。如果两个值不一致，则服务器会返回错误。这项检查可确保 **CREATE PROCEDURE** 语句的参数数据类型和 UDF 的预期参数数据类型相同。

返回
成功时会返回 sizeof(a_sql_data_type)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 不是 sizeof(a_sql_data_type)，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果查询处理状态不是标注状态，则会返回设置错误。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试对参数设置的数据类型不同于已对其指定的数据类型，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 标注状态

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性指示参数的宽度。用在 describe_parameter_set 情形中。

数据类型
a_sql_uint32

描述
UDF 的参数宽度。EXTFNAPIV4_DESCRIBE_PARM_WIDTH 仅适用于标量参数。参数宽度是以字节为单位的存储空间量，用于存储关联数据类型的参数。

- **定长数据类型** – 存储数据所需的字节。
- **变长数据类型** – 最大长度。
- **LOB 数据类型** – 将句柄存入数据所需的存储空间量。
- **TIME 数据类型** – 存储经过编码的时间所需的存储空间量。

用法
这是一个只读属性。其宽度由相关联的列数据类型派生而来。数据类型一设置完毕，就不能更改宽度。

返回
成功时会返回 sizeof(a_sql_uint32)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果查询处理状态不是标注状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_uint32 的大小不相同，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果指定的参数是表参数，则会返回设置错误。这包括参数 0 或参数 *n*，其中的 *n* 是输入表。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试重置参数宽度，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 标注状态

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute（获取） （第 228 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性指示参数的标度。用在 describe_parameter_set 情形中。

数据类型
a_sql_uint32

描述
UDF 的参数标度。对于算术型数据类型，参数标度是指数字的小数点右边的位数。

此属性对于以下数据无效：

- 非算数数据类型
- 表参数

用法
这是一个只读属性。其标度由相关联的列数据类型派生而来。数据类型一设置完毕，就不能更改标度。

返回
成功时会返回 `sizeof(a_sql_uint32)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_sql_uint32` 的大小不相同，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 如果状态不是标注状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果指定的参数是表参数，则会返回设置错误。这包括参数 0 或参数 *n*，其中的 *n* 是输入表。

查询处理状态
在以下状态下有效：

- 标注状态

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute（获取） （第 230 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性用于返回参数是否为空值。将此属性用在 `describe_parameter_set` 情形下会返回错误。

数据类型
`a_sql_byte`

描述
如果执行时参数值可以是空值，则该值为“真”。对于表参数或者参数 0，该值为“假”。

用法
这是一个只读属性。

返回
这是一个只读属性，因此所有设置结果的尝试都会返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` 错误。

查询处理状态
不适用。

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性用于返回非重复值数量。将此属性用在 `describe_parameter_set` 情形下会返回错误。

数据类型
`a_v4_extfn_estimate`

描述
返回所有调用中的估计离散值数量。仅对标量参数有效。

用法
这是一个只读属性。

返回
这是一个只读属性，所有设置结果的尝试都会返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` 错误。

查询处理状态
不适用。

另请参见
- EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES Attribute（获取）（第 233 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取）（第 228 页）
- 通用 describe_parameter 错误（第 306 页）
- 查询处理状态（第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性（设置）
`EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES` 属性用于返回参数是否为常量。将此属性用在 `describe_parameter_set` 情形下会返回错误。

数据类型
`a_sql_byte`

描述
如果语句的参数为常量，则该值为“真”。仅对标量参数有效。

用法
这是一个只读属性。

返回
这是一个只读属性，所有设置结果的尝试都会返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` 错误。

查询处理状态
不适用。

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute（获取） （第 235 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（设置） （第 247 页）
- 通用 describe_parameter 错误 （第 306 页）
- 查询处理状态 （第 115 页）
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute（获取） （第 236 页）
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性（获取） （第 228 页）

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性（设置）

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性指示参数值。用在 `describe_parameter_set` 情形中。

数据类型
`an_extfn_value`

描述
如果在描述时是已知的，则为参数值。仅对标量参数有效。

用法
这是一个只读属性。

返回
这是一个只读属性，所有设置结果的尝试都会返回 `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` 错误。

查询处理状态
不适用。

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性指示表中的列数。用在 describe_parameter_set 情形中。

数据类型
a_sql_uint32

描述
表中的列数。仅对参数 0 和表参数有效。

用法
如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的参数的名称。如果两个值不一致，则服务器会返回错误。UDF 借此可确保 **CREATE PROCEDURE** 语句的参数名称与 UDF 预期参数名称相同。

返回
成功时会返回 sizeof(a_sql_uint32)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 size of a_sql_uint32 的大小不相同，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是标注状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER — 如果参数不是表参数，则会返回设置错误。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试重置指定的表的列数，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 标注状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute（获取）（第 237 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性指示表中的行数。用在 describe_parameter_set 情形中。

数据类型
a_sql_a_v4_extfn_estimate

描述
表中的估计行数。仅对参数 0 和表参数有效。

用法
如果 UDF 估计结果集中的行数，则 UDF 会对参数 0 设置此属性。服务器会在优化过程中用估计的行数作出查询处理决策。不能对输入表设置此值。

如果不设置值，则服务器缺省情况下会使用由 **DEFAULT_TABLE_UDF_ROW_COUNT** 选项指定的行数。

返回
成功时会返回 a_v4_extfn_estimate。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_v4_extfn_estimate 的大小不相同，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是优化状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER — 如果参数不是表参数，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果表参数不是结果表，则会返回获取错误。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试重置指定的表的列数，则会返回获取错误。

查询处理状态
在以下状态下有效：

- 查询优化状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute（获取） （第 238 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性指示表中行的顺序。用在 describe_parameter_set 情形中。

数据类型
a_v4_extfn_orderby_list

描述
表中各行的次序。该属性仅对参数 0 和表参数有效。

用法
通过此属性可用 UDF 代码：

- 确定输入 **TABLE** 参数是否已经过排序
- 声明结果集已经过排序。

如果参数数量是 0，则该属性是指外发结果集。如果该参数 > 0，并且参数类型是表参数，则该属性是指输入 **TABLE** 参数。

顺序由 a_v4_extfn_orderby_list（是一个支持列序数列表的结构）及与其相关联的升序或降序属性指定。如果 UDF 通过属性对外发结果集设置顺序，则服务器就能通过优化进行排序。例如，如果 UDF 已对第一个结果集列生成升序顺序，则服务器会通过对同一列发出请求消除冗余顺序。

如果 UDF 不对外发结果集设置 orderby 属性，则服务器会假定数据未经排序。

如果 UDF 对输入 **TABLE** 参数设置 orderby 属性，则服务器保证会对输入数据进行排序。在这种情况下，UDF 会向服务器作出输入数据必须经过排序的描述。如果服务器检测到运行时冲突，则会报告 SQL 异常。例如，如果 UDF 作出这样的描述，即输入 **TABLE** 参数第一列的顺序必须是升序，但 SQL 语句含有降序子句，则服务器会报告 SQL 异常。

如果 SQL 不含排序子句，则服务器会自动进行排序，以确保按要求对输入 **TABLE** 参数排序。

返回
如果成功则会返回从 a_v4_extfn_orderby_list 复制的字节数。

查询处理状态
在以下状态下有效：

- 标注状态
- 查询优化状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute（获取）（第 239 页）
- 查询处理状态（第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 属性指示 UDF 需要分区。用于 describe_parameter_set 情形。

数据类型

a_v4_extfn_column_list

描述

UDF 开发者使用 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** 以编程方式声明，调用之前须先对 UDF 进行分区。

用法

UDF 可以查询分区以强制执行，或者动态调整分区。UDF 必须分配 **a_v4_extfn_column_list**，需要考虑输入表的总列数，并将其发送至服务器。

返回

成功时会返回 a_v4_extfn_column_list 的大小。该值等于：

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 如果缓冲区的长度小于预期大小，则会返回设置错误。

查询处理状态

在以下状态下有效：

- 标注状态
- 查询优化状态

示例

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32    rc = 0;
        a_v4_extfn_column_list pbcol =
        { 1,        // 1 column in the partition by list
        2 };    // column index 2 requires partitioning

        // Describe partitioning for argument 1 (the table)
```

```
        rc = ctx->describe_parameter_set(
        ctx, 1,
        EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
        &pbcol,
        sizeof(pbcol) );

        if( rc == 0 ) {
        ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
        }
    }
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get) （第 240 页）
- 通用 describe_parameter 错误 （第 306 页）
- 第 4 版 API describe_parameter 和 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY （第 133 页）
- 使用 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 的并行 TPF PARTITION BY 示例 （第 135 页）
- 查询处理状态 （第 115 页）
- 输入数据分区 （第 133 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND` 属性指示消耗程序请求回绕输入表。用在 `describe_parameter_set` 情形中。

数据类型
`a_sql_byte`

描述
表示消耗程序想要回绕输入表。仅对表输入参数有效。缺省情况下，该属性为“假”。

用法
如果 UDF 需要输入表回绕功能，则 UDF 必须在优化过程中设置此属性。

返回
成功时会返回 `sizeof(a_sql_byte)`。

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果 **describe_buffer** 和 `a_sql_byte` 的大小不相同，则会返回设置错误。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是优化状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果 UDF 尝试对参数 0 设置此属性，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER — 如果 UDF 尝试对非表参数设置此属性，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试对此属性设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

示例
在此示例中，如果在优化状态下调用函数 **my_udf_describe**，则会通过调用 describe_parameter_set 告知表输入参数 1 的生产程序：可能必须回绕。

示例过程定义：

```
CREATE PROCEDURE my_udf(IN t TABLE(c1 INT))
RESULT (x INT)
EXTERNAL NAME 'my_udf@myudflib';
```

示例 _describe_extfn API 函数代码段：

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
a_sql_byte rewind_required = 1;
a_sql_int32 ret = 0;

ret = cntxt->describe_parameter_set( cntxt, 1,
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
&rewind_required,
sizeof(a_sql_byte) );

if( ret <= 0 ) {
    // Handle the error.
}
}
}
```

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）（第 241 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）（第 258 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）（第 242 页）

- _rewind_extfn （第 301 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）
EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性指示参数是否支持回绕。用在 describe_parameter_set 情形中。

数据类型
a_sql_byte

描述
表示生产者是否支持回绕。仅对表参数有效。

如果打算将 DESCRIBE_PARM_TABLE_HAS_REWIND 设置为 true，还必须提供回绕表回调 **(_rewind_extfn())** 实现方案。如果不提供回调方法，则服务器不能执行 UDF。

用法
如果 UDF 可在没有开销的情况下对其结果表提供回绕功能，则 UDF 会在优化状态下设置此属性。如果 UDF 提供回绕功能的开销很大，请不要设置此属性，或将其设置为 0。如果设置为 0，则服务器会提供回绕支持。

返回
成功时会返回 sizeof(a_sql_byte)。

失败时会返回某个通用 describe_parameter 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果 **describe_buffer** 和 a_sql_byte 的大小不相同，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不是优化状态，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER —如果 UDF 尝试通过非表参数设置此属性，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果指定的参数不是结果表，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试对此属性设置的值不是 0 或 1，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）（第 241 页）

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）（第 242 页）
- _rewind_extfn （第 301 页）
- 查询处理状态 （第 115 页）

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` 属性列出未被使用的列。用于 `describe_parameter_set` 情形。

数据类型
`a_v4_extfn_column_list`

描述
服务器或者 UDF 将不会使用的输出表列的列表。

对于输出表参数，UDF 通常会对所有列生成数据，而服务器会消耗所有列。对于输入表参数，情况也是如此，服务器通常会对所有列生成数据，而 UDF 会消耗所有列。

但是，某些情况下服务器或 UDF 可能不会消耗所有列。此种情况下的最佳做法是，由 UDF 对输出表执行 **GET** 操作以获取描述属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS**。该操作对服务器进行查询，以获取输出表中将不会被服务器用到的列的列表。接下来，当为输出表生成列数据时，UDF 可以使用该列表；也就是说，UDF 跳过了对未使用列生成数据的操作。

总之，对于输出表，UDF 会轮询未使用的列的列表。对于输入表，UDF 将推入未使用列的列表。

用法
优化期间，如果将不会使用输入表参数的某些列，则 UDF 会设置该属性。UDF 必须分配包括所有输出表列的 `a_v4_extfn_column_list`，然后还必须将其传递给服务器。服务器则将所有未被映射的列的序号标记为 1。服务器将该列表复制到内部的数据结构中。

返回
成功时会返回列列表大小：`sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`.

失败时会返回某个通用 `describe_parameter` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 如果并非处于优化状态，则会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 如果 UDF 试图获取输入表的该属性，则会返回设置错误。

- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – 如果 UDF 试图为非表参数设置该属性，则会返回设置错误。

查询处理状态
在以下状态下有效：

- 优化状态

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute（获取） （第 243 页）

*describe_udf_get

`describe_udf_get` 第 4 版 API 方法可从服务器中获取 UDF 属性。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_udf_get)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    void    *describe_buffer,
    size_t    describe_buffer_len );
```

参数

参数	描述
cntxt	此 UDF 的过程上下文对象。
describe_type	指示要检索的属性的选择器。
describe_buffer	对于指定的要在服务器中设置的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回
成功时会返回 0 或已写入 **describe_buffer** 的字节数。值为 0 表明服务器不能获取属性，但没出错。如果出错或未检索到属性，则此函数会返回某个通用 `describe_udf` 错误。

另请参见

- *describe_udf_set （第 262 页）
- 通用 describe_udf 错误 （第 306 页）

返回 *describe_udf_get 的属性

以下代码中有 describe_udf_get 属性。

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（获取）

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性指示参数数量。用在 describe_udf_get 情形中。

数据类型
a_sql_uint32

描述
提供给 UDF 的参数的数量。

用法
用于获取 **CREATE PROCEDURE** 语句中所定义的参数数量。

返回
成功时会返回 sizeof(a_sql_uint32)。

失败时会返回某个通用 describe_udf 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果描述缓冲区大小不同于 a_sql_uint32，则会返回获取错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不晚于初始状态，则会返回获取错误。

查询处理状态

- 标注状态
- 查询优化状态
- 计划构建状态
- 执行状态

另请参见

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（设置） （第 263 页）
- 通用 describe_udf 错误 （第 306 页）
- 查询处理状态 （第 115 页）

*describe_udf_set

`describe_udf_set` 第 4 版 API 方法可在服务器中设置 UDF 属性。

声明

```
a_sql_int32 (SQL_CALLBACK *describe_udf_set)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    const void    *describe_buffer,
    size_t    describe_buffer_len );
```

参数

参数	描述
cntxt	此 UDF 的过程上下文对象。
describe_type	指示要设置的属性的选择器。
describe_buffer	对于指定的要在服务器中设置的属性，是用于存储描述信息的结构。具体结构或数据类型由 **describe_type** 参数表示。
describe_buffer_length	**describe_buffer** 的字节长度。

返回

成功时会返回已写入 **describe_buffer** 的字节数。如果出错或未检索到属性，则此函数会返回某个通用 `describe_udf` 错误。

如果出错或未检索到属性，则此函数会返回某个通用 `describe_udf` 错误，或者：

- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` — 如果任一 **cntxt** 或 **describe_buffer** 参数是空值，或者 **describe_buffer_length** 是 0，都会返回设置错误。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — 如果所请求的属性的大小和所提供的 **describe_buffer_length** 不一致，则会返回设置错误。

另请参见

- *describe_udf_get （第 260 页）
- 通用 describe_udf 错误 （第 306 页）

*describe_udf_set 的属性

以下代码中有 `describe_udf_set` 属性。

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（设置）
EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性指示参数数量。用在 describe_udf_set 情形中。

数据类型
a_sql_uint32

描述
提供给 UDF 的参数的数量。

用法
如果 UDF 设置此属性，则服务器会比较该值与 **CREATE PROCEDURE** 语句中所提供的参数数量。如果两个值不一致，则服务器会返回 SQL 错误。UDF 借此可确保 **CREATE PROCEDURE** 语句的参数数量与 UDF 预期参数数量相同。

返回
成功时会返回 sizeof(a_sql_uint32)。

失败时会返回某个通用 describe_udf 错误，或者：

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — 如果描述缓冲区大小不同于 a_sql_uint32，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 如果状态不同于标注，则会返回设置错误。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE — 如果 UDF 尝试重置参数数据类型，则会返回设置错误。

查询处理状态

- 标注状态

另请参见
- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性（获取） （第 261 页）
- 通用 describe_udf 错误 （第 306 页）
- 查询处理状态 （第 115 页）

描述列的类型 (a_v4_extfn_describe_col_type)

枚举类型 a_v4_extfn_describe_col_type 选择由 UDF 检索或设置的列属性。

实现

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
```

```
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    EXTFNAPIV4_DESCRIBE_COL_LAST
} a_v4_extfn_describe_col_type;
```

成员摘要

组成成员	描述
EXTFNAPIV4_DESCRIBE_COL_NAME	列名称（有效标识符）。
EXTFNAPIV4_DESCRIBE_COL_TYPE	列数据类型。
EXTFNAPIV4_DESCRIBE_COL_WIDTH	字符串宽度（数值精度）。
EXTFNAPIV4_DESCRIBE_COL_SCALE	数值标量。
EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL	如果列可以为空，则为“真”。
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES	该列中的估计离散值数量。
EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE	如果表中的列是惟一的，则该值为“真”。
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT	如果列在语句的生存期内保持不变，则为“真”。
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE	如果在描述时是已知的，则为参数值。
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER	如果列为表的消耗程序所必需，则该值为“真”。
EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE	列的最小值（如果已知）。
EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE	列的最大值（如果已知）。
EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT	结果列值为输入表列的子集。

组成成员	描述
EXTFNAPIV4_DESCRIBE_COL_LAST	v4 API 的第一个非法值。带外值。

描述参数的类型 (a_v4_extfn_describe_parm_type)

枚举类型 a_v4_extfn_describe_parm_type 选择由 UDF 检索或设置的参数属性。

实现

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    EXTFNAPIV4_DESCRIBE_PARM_LAST
} a_v4_extfn_describe_parm_type;
```

成员摘要

组成成员	描述
EXTFNAPIV4_DESCRIBE_PARM_NAME	参数名称（有效标识符）。
EXTFNAPIV4_DESCRIBE_PARM_TYPE	数据类型。
EXTFNAPIV4_DESCRIBE_PARM_WIDTH	字符串宽度（数值精度）。
EXTFNAPIV4_DESCRIBE_PARM_SCALE	数值标量。
EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL	如果该值可以是空值，则为“真”。
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES	所有调用中的估计离散值数量。

组成成员	描述
EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT	如果语句的参数为常量，则该值为“真”。
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE	如果在描述时是已知的，则为参数值。
这些选择器可以检索或 设置表参数的属性。这些枚举值不能与 标量参数一起使用：	
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS	表中的列数。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS	表中的估计行数。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY	表中各行的次序。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY	分区；将 *number_of_columns*=0 用于 ANY。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND	如果消耗程序希望具备回绕输入表的功能，则为“真”。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND	如果生产者支持回绕功能，则返回“真”。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS	服务器或者 UDF 将不会使用的输出表列的列表。
EXTFNAPIV4_DESCRIBE_PARM_LAST	v4 API 的第一个非法值。带外值。

Describe Return (a_v4_extfn_describe_return)

当 `a_v4_extfn_proc_context.describe_xxx_get()` 或 `a_v4_extfn_proc_context.describe_xxx_set()` 不成功时，枚举类型 `a_v4_extfn_describe_return` 将提供一个返回值。

实现

```
typedef enum a_v4_extfn_describe_return {
   EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE          =  0,   // the specified operation has no
meaning either for this attribute or in

the current context.
   EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH       = -1,   // the provided buffer size
does not match the required length or the

length is insufficient.
   EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER       = -2,   // the provided parameter number
is invalid
```

```
   EXTFNAPIV4_DESCRIBE_INVALID_COLUMN           = -3,   // the column number is invalid
for this table parameter
   EXTFNAPIV4_DESCRIBE_INVALID_STATE          = -4,   // the describe method call is not
valid in the present state
   EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE       = -5,   // the attribute is known but not
appropriate for this object
    EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE        = -6,   // the identified attribute is
not known to this server version
   EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER        = -7,   // the specified parameter is
not a table parameter (for describe_col_get()

or set())
   EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE       = -8,   // the specified attribute
value is illegal
    EXTFNAPIV4_DESCRIBE_LAST                = -9
} a_v4_extfn_describe_return;
```

成员摘要

组成成员	返回值	描述
EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE	0	指定操作对于此属性或者 在当前上下文中没有意义。
EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH	-1	所提供的缓冲区大小同需要的长度不匹配，或者长度不够。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER	-2	提供的参数数量无效。
EXTFNAPIV4_DESCRIBE_INVALID_COLUMN	-3	列号对于此表参数无效。
EXTFNAPIV4_DESCRIBE_INVALID_STATE	-4	在当前状态中对 describe 方法的调用无效。
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE	-5	属性已知，但并不适合于此对象。
EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE	-6	该服务器版本不能识别所确定的属性。
EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER	-7	所指定的参数并非表参数（对于 `describe_col_get()` 或 `describe_col_set()`）。
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE	-8	所指定的属性无效。
EXTFNAPIV4_DESCRIBE_LAST	-9	v4 API 的第一个非法值。

描述

`a_v4_extfn_proc_context.describe_xxx_get()` 和 `a_v4_extfn_proc_context.describe_xxx_set()` 的返回值为带符号整数。如果结果为正整数，则操作成功，返回值为 复制的字节数量。如果返回值小于或等于零，则操作不成功，返回值为 `a_v4_extfn_describe_return` 值中的一个。

描述 UDF 的类型 (a_v4_extfn_describe_udf_type)

使用枚举类型 `a_v4_extfn_describe_udf_type` 对 UDF 检索或设置的逻辑属性进行选择。

实现

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extfn_describe_udf_type;
```

成员摘要

组成成员	描述
EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS	提供给 UDF 的参数的数量。
EXTFNAPIV4_DESCRIBE_UDF_LAST	带外值。

描述

UDF 使用 `a_v4_extfn_proc_context.describe_udf_get()` 方法以检索属性， 使用 `a_v4_extfn_proc_context.describe_udf_set()` 方法设置 关于 UDF 的整体属性。 枚举器 `a_v4_extfn_describe_udf_type` 对 UDF 检索或设置的逻辑属性进行选择。

另请参见

- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）

执行状态 (a_v4_extfn_state)

`a_v4_extfn_state` 枚举类型表示 UDF 的查询处理状态。

实现

```
typedef enum a_v4_extfn_state {
   EXTFNAPIV4_STATE_INITIAL,                // Server initial state,
not used by UDF
    EXTFNAPIV4_STATE_ANNOTATION,                // Annotating parse
```

```
tree with UDF reference
    EXTFNAPIV4_STATE_OPTIMIZATION,              // Optimizing
   EXTFNAPIV4_STATE_PLAN_BUILDING,             // Building execution
plan
    EXTFNAPIV4_STATE_EXECUTING,                 // Executing UDF and
fetching results from UDF
    EXTFNAPIV4_STATE_LAST
} a_v4_extfn_state;
```

成员摘要

组成成员	描述
EXTFNAPIV4_STATE_INITIAL	服务器初始状态。该状态期间所调用的唯一 UDF 方法为 `_start_extfn`。
EXTFNAPIV4_STATE_ANNOTATION	引用 UDF 的标注分析树。该状态期间未调用 UDF。
EXTFNAPIV4_STATE_OPTIMIZATION	优化。服务器先调用 UDF 的 `_start_extfn` 方法，而后调用 `_describe_extfn` 函数。
EXTFNAPIV4_STATE_PLAN_BUILDING	构建查询执行计划。服务器调用 UDF 的 `_describe_extfn` 函数。
EXTFNAPIV4_STATE_EXECUTING	执行 UDF 并从中提取结果值。服务器先调用 `_describe_extfn` 函数，再从 UDF 提取数据。然后，服务器再调用 `_evaluate_extfn` 以启动提取周期。在提取周期中，服务器将调用 `a_v4_extfn_table_func` 中所定义的函数。当提取完成时，服务器将调用 UDF 的 `_close_extfn` 函数。
EXTFNAPIV4_STATE_LAST	v4 API 的第一个非法值。带外值。

描述

`a_v4_extfn_state` 枚举类型指出服务器处于 UDF 的哪一个执行阶段。当进行状态转换时，服务器将调用 UDF 的 `_leave_state_extfn` 函数以通知 UDF 它正转出前一状态。服务器将调用 UDF 的 `_enter_state_extfn` 函数以通知 UDF 它正转入新的状态。

UDF 的查询处理状态将会限制其所能执行的操作。例如，当处于标注状态时，UDF 仅能对常量参数类型的数据进行检索。

另请参见

- 查询处理状态 （第 115 页）
- _start_extfn （第 270 页）

- _evaluate_extfn （第 271 页）
- _enter_state_extfn （第 272 页）
- _leave_state_extfn （第 272 页）
- 表函数 (a_v4_extfn_table_func) （第 298 页）

外部函数 (a_v4_extfn_proc)

服务器使用 a_v4_extfn_proc 结构调用 UDF 中的各入口点。服务器将 a_v4_extfn_proc_context 实例传给每个函数。

方法总结

方法	描述
_start_extfn	分配一个结构，并将其地址存储于 a_v4_extfn_proc_context 中的 **_user_data** 字段。
_finish_extfn	释放一个结构，其地址存储于 a_v4_extfn_proc_context 中的 **user_data** 字段。
_evaluate_extfn	基于新参数组的每次函数调用都将调用所需的函数指针。
_describe_extfn	请参见 Describe API （第 193 页）。
_enter_state_extfn	UDF 可以使用此函数来分配结构。
_leave_state_extfn	UDF 可以使用此函数来释放状态所需的内存或资源。

_start_extfn

使用 _start_extfn 第 4 版 API 方法作为指向初始化函数的可选指针，其唯一参数是指向 a_v4_extfn_proc_context 结构的指针。

声明

```
_start_extfn(
a_v4_extfn_proc_context *
)
```

用法

使用 _start_extfn 方法分配结构并将其地址存入 a_v4_extfn_proc_context 中的 _user_data 字段。如果无须进行初始化，则必须将此函数指针设置为空指针。

参数

参数	描述
cntxt	过程上下文对象。

_finish_extfn

将 `_finish_extfn` v4 API 方法用作指向一个关闭函数的可选指针，其唯一参数为指向 `a_v4_extfn_proc_context` 的指针。

声明

```
_finish_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

用法

`_finish_extfn` API 释放一个结构，其地址存储于 `a_v4_extfn_proc_context` 中的 `user_data` 字段。如果无需进行清理，则必须将此函数指针设置为空指针。

参数

参数	描述
cntxt	过程上下文对象。

_evaluate_extfn

将 `_evaluate_extfn` v4 API 方法作为所需的函数指针，并于每次基于新参数组调用函数时调用该指针。

声明

```
_evaluate_extfn(
     a_v4_extfn_proc_context *cntxt,
     void *args_handle
)
```

用法

`_evaluate_extfn` 函数必须向服务器描述如何通过填充 `a_v4_extfn_table` 结构的`a_v4_extfn_table_func` 部分来提取结果，并且在使用参数 0 的上下文中通过 `set_value` 方法将该信息发送至服务器。在基于参数 0 调用 `set_value` 方法之前，此函数还必须通过填充 `a_v4_extfn_table` 结构的 `a_v4_extfn_value_schema` 部分来通知服务器其输出模式。它可以使用回调函数 `get_value` 来访问其输入参数值。此时 UDF 可以使用常量和非常量参数。

参数

参数	描述
cntxt	过程上下文对象。
args_handle	服务器中的参数的句柄。

_describe_extfn

在每个状态开始时调用 `_describe_extfn`，以允许服务器获取或设置逻辑属性。UDF 可以使用 `a_v4_proc_context object` 中的六种 describe 方法（`describe_parameter_get`，`describe_parameter_set`，`describe_column_get`，`describe_column_set`，`describe_udf_get`，和 `describe_udf_set`）来完成此操作。

请参见 Describe API （第 193 页）。

_enter_state_extfn

UDF 可以将 `_enter_state_extfn` v4 API 方法作为可选入口点来执行，每当 UDF 进入新的状态时则给予通知。

声明

```
_enter_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

用法

UDF 可以使用此通知来分配结构。

参数

参数	描述
cntxt	过程上下文对象。

_leave_state_extfn

`_leave_state_extfn` v4 API 方法为可选入口点，当 UDF 转出查询处理状态时，可以使用该方法接收通知。

声明

```
_leave_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

用法
UDF 可用该通知释放状态所需的内存或资源。

参数

参数	描述
cntxt	过程上下文对象。

外部过程上下文 (a_v4_extfn_proc_context)

请使用 `a_v4_extfn_proc_context` 结构以保留来自于服务器和 UDF 的上下文信息。

实现

```
typedef struct a_v4_extfn_proc_context {
.
.
.
} a_v4_extfn_proc_context;
```

方法总结

返回类型	方法	描述
短整型	**get_value**	获取 UDF 的输入参数。
短整型	**get_value_is_constant**	允许 UDF 查询其所获取的参数是否为常量。
短整型	**set_value**	UDF 在 `_evaluate_extfn` 或 `_describe_extfn` 函数中使用该方法，向服务器描述其所输出的结果值的格式，并告知服务器如何从 UDF 中提取结果值。
a_sql_uint32	**get_is_cancelled**	每秒（或每两秒）调用一次 **get_is_cancelled** 方法，以查看用户是否已中断当前语句的执行。
短整型	**set_error**	对当前语句执行回滚操作，并且生成一个错误。
无类型	**log_message**	将消息写入到消息日志中。
短整型	**convert_value**	将一种数据类型转换为另一种数据类型。
短整型	**get_option**	获取可设置选项的值。
无类型	**alloc**	分配长度至少为 "len" 的内存块。

返回类型	方法	描述
无类型	**free**	释放由 **alloc()** 所分配的具有指定生命周期的内存块。
a_sql_uint32	**describe_column_get**	请参见*describe_column_get （第 194 页）。
a_sql_uint32	**describe_column_set**	请参见*describe_column_set （第 209 页）。
a_sql_uint32	**describe_parameter_get**	请参见*describe_parameter_get （第 226 页）。
a_sql_uint32	**describe_parameter_set**	请参见*describe_parameter_set （第 245 页）。
a_sql_uint32	**describe_udf_get**	请参见*describe_udf_get （第 260 页）。
a_sql_uint32	**describe_udf_set**	请参见*describe_udf_set （第 262 页）。
短整型	**open_result_set**	为表中的值打开结果集。
短整型	**close_result_set**	关闭已打开的结果集。
短整型	**get_blob**	检索为 BLOB 的输入参数。
短整型	**set_cannot_be_distributed**	禁用 UDF 级别的分发（即使库具有可分发性）。

数据成员及数据类型的摘要

数据成员	数据类型	描述
_user_data	void*	使用外部例程所需的任何环境数据皆可对该指针赋值。
_executionMode	a_sql_uint32	通过 **External_UDF_Execution_Mode** 选项指出请求的调试／跟踪级别。这是只读字段。
current_state	a_sql_uint32	*current_state* 属性反映了当前环境的执行模式。可以通过诸如 `_describe_extfn` 的函数进行查询，以确定应采取何种措施。

描述

除了保留来自服务器和 UDF 的上下文信息之外，`a_v4_extfn_proc_context` 结构允许 UDF 回调至服务器以执行某些操作。UDF 可以在 `_user_data` 成员的该结构中存储私有数据。由服务器将此结构的实例传至 `a_v4_extfn_proc` 方法中的函数。直至服务器达到标注状态之后，用户数据才予以保留。

get_value

使用 get_value 第 4 版 API 方法在 SQL 查询中获取向 UDF 发送的输入参数的值。

声明

```
short get_value(
       void *           arg_handle,
       a_sql_uint32     arg_num,
       an_extfn_value *value
       )
```

用法

get_value API 在计算方法中用于检索每个 UDF 输入参数的值。对于窄型参数数据 (>32K)，调用 get_value 就足以 检索整个参数值。

通过任何有权访问 arg_handle 指针的 API，均可调用 get_value API。这包括可接收 a_v4_table_context 并将其作为参数的 API 函数。a_v4_table_context 有可用于执行上述调用的 args_handle 成员变量。

对于所有长度固定的数据类型，均可从返回值中得到数据，无须执行其他调用即可获取所有数据。生产程序可决定最大长度，而后者会通过调用 get_value 方法全部返回。所有长度固定的数据类型都应该保证可以存入一个连续的缓冲区。对于变长数据，限制取决于生产程序。

对于长度不固定的数据类型，可能必须用 get_blob 方法创建 blob 才能获得数据，具体情况取决于数据的长度。可将宏 **EXTFN_IS_INCOMPLETE** 用于 get_value 返回的值，以判断是否需要 blob 对象。如果 **EXTFN_IS_INCOMPLETE** 的计算结果是 true，则需要 blob。

对于表输入参数，类型是 **AN_EXTFN_TABLE**。对于此类参数，必须通过 open_result_set 方法创建结果集，才能读取表中的值。

如果调用 _evaluate_extfn API 前 UDF 需要参数值，则 UDF 应该实现 _describe_extfn API。通过 _describe_extfn API，UDF 能用 describe_parameter_get 方法获取常量表达式的值。

参数

参数	描述
arg_handle	消耗程序提供的上下文指针。
arg_num	要获取其值的参数的索引。参数索引始于 1。
value	指定的参数的值。

返回

如果成功则返回 1，否则返回 0。

an_extfn_value 结构

an_extfn_value 结构代表着 `get_value` API 返回的输入参数的值。

这段代码显示 **an_extfn_value** 结构的声明方法：

```
short typedef struct an_extfn_value {
       void*           data;
       a_sql_uint32    piece_len,
       an_extfn_value *value {
                              a_sql_uint32    total_len;
                              a_sql_uint32    remain_len;
       } len;
       a_sql_data_type    type;
} an_extfn_value;
```

下表说明调用 `get_value` 方法后返回的 **an_extfn_value** 对象的值：

get_value API 返回的值	EXTFN_IS_INCOM-PLETE	total_len	piece_len	数据
空值	FALSE	0	0	空值
空字符串	FALSE	0	0	非空值
大小 < MAX_UINT32	FALSE	actual	actual	非空值
大小 < MAX_UINT32	TRUE	actual	0	非空值
大小 >= MAX_UINT32	TRUE	MAX_UINT32	0	非空值

an_extfn_value 的类型字段含有值的数据类型。对于以表为输入参数的 UDF，那种参数的数据类型是 **DT_EXTFN_TABLE**。对于第 4 版表 UDF，不会使用 `remain_len` 字段。

另请参见

- _evaluate_extfn （第 271 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- _describe_extfn （第 272 页）
- *describe_parameter_get （第 226 页）

get_value_is_constant

使用 `get_value_is_constant` 第 4 版 API 方法判断指定的输入参数 是否为常量。

声明

```
short get_value_is_constant(
       void *          arg_handle,
       a_sql_uint32    arg_num,
```

```
        an_extfn_value *value_is_constant
        )
```

用法
UDF 可询问给定的参数是否为常量。可借此优化 UDF，例如，可在首次调用 `_evaluate_extfn` 函数期间运行一次，而不是每当执行计算调用时 都运行。

参数

参数	描述
arg_handle	服务器中参数的句柄。
arg_num	要检索的输入参数的索引值。 索引值是 1..N 的值。
value_is_constant	用于进行存储的 out 参数是常量。

返回
如果成功则返回 1，否则返回 0。

另请参见

- _evaluate_extfn （第 271 页）

set_value

使用 `set_value` 第 4 版 API 方法向消耗程序描述结果集有多少列，以及应该如何读取数据。

声明

```
short set_value(
        void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value
        )
```

用法
此方法由 UDF 在 `_evaluate_extfn` API 中使用。UDF 必须调用 `set_value` 方法才能告知消耗程序结果集中有多少列，以及 UDF 支持什么 `a_v4_extfn_table_func` 函数。

对于 `set_value` API，UDF 会通过 `_evaluate_extfn` API 或通过 `a_v4_extfn_table_context` 结构的 **args_handle** 成员提供相应 **arg_handle** 指针。

对于第 4 版表 UDF，`set_value` 方法的 **value** 的参数必须是 *DT_EXTFN_TABLE* 类型的参数。

参数

参数	描述
arg_handle	消耗程序提供的上下文指针。
arg_num	要对其设置值的参数的索引。只有 0 是受支持的参数。
value	指定的参数的值。

返回
如果成功则返回 1，否则返回 0。

另请参见
- _evaluate_extfn （第 271 页）
- 表函数 (a_v4_extfn_table_func) （第 298 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）

get_is_cancelled

使用 `get_is_cancelled` 第 4 版 API 方法判断是否已取消 语句。

声明

```
short get_is_cancelled(
       a_v4_extfn_proc_context *          cntxt,

       )
```

用法
如果 UDF 条目的运行时间加长（达到数秒），则应该（如有可能）每秒或每两秒都调用一次 `get_is_cancelled` 回调函数，以确定用户是否已打断当前语句的执行。如果已打断语句的执行，则会返回非零值，然后应该立即返回 UDF 条目。调用 `_finish_extfn` 函数以进行必要清理。此后不得调用任何其他 UDF 条目。

参数

参数	描述
cntxt	过程上下文对象。

返回
如果已打断语句的执行，则会返回非零值。

另请参见
- 标量和集合 UDF 回调函数 （第 80 页）

set_error

使用 set_error 第 4 版 API 方法将错误反馈给服务器 并最终告知用户。

声明

```
void set_error(
       a_v4_extfn_proc_context *          cntxt,
       a_sql_uint32                       error_number,
       const char                         *error_desc_string
       )
```

用法

如果 UDF 条目遇到应该向用户发送错误消息， 并且关闭当前语句的错误，则调用 set_error API。调用后，set_error API 会回退当前语句，用户还会看到“Error raised by user-defined function: <error_desc_string>”。SQLCODE 是 所提供的 <error_number> 的求反形式。

为了防止与现有错误代码发生冲突，UDF 生成的错误编号应该介于 17000 和 99999 之间。如果提供的数值不在此范围内，仍会回退语句，但错误消息会是“Invalid error raised by user-defined function: (<error_number>) <error_desc_string>”，SQLCODE 为 -1577。 **error_desc_string** 的最大长度为 140 个字符。

对 set_error 的调用执行完毕后，UDF 条目会立即执行返回操作；最终会调用 _finish_extfn 函数，以执行必要清理。此后不得调用任何其他 UDF 条目。

参数

参数	描述
cntxt	过程上下文对象
error_number	要设置的错误编号
error_desc_string	要使用的消息字符串

另请参见

- 标量和集合 UDF 回调函数 （第 80 页）

log_message

使用 log_message 第 4 版 API 方法向服务器消息日志发送消息。

声明

```
short log_message(
       const char         *msg,
       short              msg_length
```

```
        )
```

用法

`log_message` 方法用于将消息写入消息日志。消息字符串必须 是可显示的文本字符串，不长于 255 字节；较长的消息可能会被截断。

参数

参数	描述
msg	要写入日志的消息字符串
msg_length	消息字符串的长度

另请参见

- 控制错误检查和调用跟踪 （第 26 页）

convert_value

使用 `convert_value` 第 4 版 API 方法转换数据类型。

声明

```
short convert_value(
       an_extfn_value *input,
       an_extfn_value *output
       )
```

用法

. `convert_value` API 主要用于在 `DT_DATE`、`DT_TIME`、`DT_TIMESTAMP` 和 `DT_TIMESTAMP_STRUCT` 之间进行转换。会向该函数传递输入和输出 `an_extfn_value`。

输入参数

参数	描述
an_extfn_value.data	输入数据指针
an_extfn_value.total_len	输入数据的长度
an_extfn_value.type	输入的 DT_ datatype

输出参数

参数	描述
an_extfn_value.data	UDF 提供的输出数据点
an_extfn_value.piece_len	输出数据的最大长度。
an_extfn_value.total_len	服务器设置的转换后的长度
an_extfn_value.type	所需输出的 DT_ datatype

返回
如果成功则返回 1，否则返回 0。

另请参见

- get_value （第 275 页）

get_option

`get_option` 第 4 版 API 方法用于获取可设置选项的值。

声明

```
short get_option(
a_v4_extfn_proc_context * cntxt,
char *option_name,
an_extfn_value *output
)
```

参数

参数	描述
cntxt	过程上下文对象
option_name	要获取的选项的名称
输出	• `an_extfn_value.data` — UDF 提供的输出数据指针 • `an_extfn_value.piece_len` — 输出数据的最大长度 • `an_extfn_value.total_len` — 转换数据后的服务器设置 长度 • `an_extfn_value.type` — 服务器设置的值数据类型

返回
如果成功则返回 1，否则返回 0。

另请参见

- 外部函数原型 （第 91 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）

分配

`alloc` v4 API 方法分配内存块。

声明

```
void*alloc(
     a_v4_extfn_proc_context *cntxt,
     size_t len
     )
```

用法

分配长度至少为 **len** 的内存块。返回的内存以 8 字节为单位进行调整。Sybase 推荐将 `alloc()` 作为内存分配的唯一方法，该方法允许服务器对外部例程的内存使用状况进行跟踪。服务器可以调整其他内存用户、跟踪泄露、并能提供改善的诊断和监控服务。

仅当 **external_UDF_execution_mode** 设置为 1 或 2 时（校验模式或跟踪模式），启用内存跟踪。

参数

参数	描述
cntxt	过程上下文对象
len	分配的内存长度（以字节为单位）

另请参见

- 释放 （第 282 页）
- 启用内存跟踪 （第 128 页）

释放

`free` v4 API 方法释放已分配的内存块。

声明

```
void free(
     a_v4_extfn_proc_context *cntxt,
     void *mem
)
```

用法

释放由 `alloc()` 分配的具有指定生命周期的内存块。

仅当 **external_UDF_execution_mode** 设置为 1 或 2 时（校验模式或跟踪模式），启用内存跟踪。

参数

参数	描述
cntxt	过程上下文对象
mem	指向由 `alloc` 方法分配的内存块的指针。

另请参见

- 分配 （第 282 页）
- 启用内存跟踪 （第 128 页）

open_result_set

`open_result_set` 第 4 版 API 方法用于打开表值结果集。

声明

```
short open_result_set(
a_v4_extfn_proc_context *cntxt,
a_v4_extfn_table *table,
a_v4_extfn_table_context **result_set
)
```

用法

`open_result_set` 用于打开表值结果集。UDF 可打开结果集，从 `DT_EXTFN_TABLE` 类输入参数中读取行。服务器（或另一 UDF）可打开结果集，通过 UDF 读取行。

参数

参数	描述
cntxt	过程上下文对象
table	要对其打开结果集的表对象
result_set	已设置为已打开的结果集的 输出参数

返回

如果成功则返回 1，否则返回 0。

有关 `open_result_set` 用法示例，请参见 `fetch_block` 和 `fetch_into` 第 4 版 API 方法说明。

另请参见

- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- fetch_into （第 292 页）
- fetch_block （第 294 页）

close_result_set

close_result_set 第 4 版 API 方法用于关闭已打开的结果集。

声明

```
short close_result_set(
    a_v4_extfn_proc_context *cntxt,
    a_v4_extfn_table_context *result_set
)
```

用法

仅可对每个结果集使用一次 close_result_set。

参数

参数	描述
cntxt	过程上下文对象
result_set	要关闭的结果集

返回

如果成功则返回 1，否则返回 0。

get_blob

使用 get_blob 第 4 版 API 方法检索输入 blob 参数。

声明

```
short get_blob(
       void            *arg_handle,
       a_sql_uint32    arg_num,
       a_v4_extfn_blob **blob
       )
```

用法

使用 get_blob 在调用 get_value() 后检索 blob 输入参数。如果 *piece_len* < *total_len*，则可用宏 EXTFN_IS_INCOMPLETE 判断是否必须有 blob 对象才能读取通过 get_value() 返回的值的数据。blob 对象会以输出参数的形式返回，并由调用方拥有。

get_blob 用于获取可用于读取 blob 内容的 blob 句柄。仅可对含有 blob 对象的列调用此方法。

参数

参数	描述
arg_handle	服务器中参数的句柄
arg_num	参数值是 1...N 的数值
blob	含有 blob 对象的输出参数

返回
如果成功则返回 1，否则返回 0。

另请参见
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- get_value （第 275 页）

set_cannot_be_distributed

即使已达到库级别的分配标准，`set_cannot_be_distributed` 第 4 版 API 方法也可在 UDF 级别禁止分配。

声明

```
void set_cannot_be_distributed( a_v4_extfn_proc_context *cntxt)
```

用法
缺省行为下，库可分配时 UDF 也可分配。可在 UDF 中用 `set_cannot_be_distributed` 作出禁止对服务器分配的决策。

许可证信息 (a_v4_extfn_license_info)

如果您是 Sybase 的设计合作伙伴，请使用 `a_v4_extfn_license_info` 结构为您的 UDF 定义库级许可证验证，其中包括您的公司名称、库版本信息以及 Sybase 所提供的许可证密钥。

实现

```
typedef struct an_extfn_license_info {
    short     version;
} an_extfn_license_info;

typedef struct a_v4_extfn_license_info {
    an_extfn_license_info version;

    const char        name[255];
    const char        info[255];
```

```
    void *        key;
} a_v4_extfn_license_info;
```

数据成员摘要

数据成员	描述
version	仅供内部使用。必须设置为 1。
名称	为 UDF 集赋值您的公司名称。
info	为 UDF 集赋值其他库信息，例如库版本和内部版本号。
key	（仅限 Sybase 合作伙伴）Sybase 所提供的许可证密钥。该密钥是一个 26 字节的数组。

优化程序估计 (a_v4_extfn_estimate)

请使用 `a_v4_extfn_estimate` 结构来描述估计，其中包括一个值 和一个置信度。

实现

```
typedef struct a_v4_extfn_estimate {
    double    value;
    double    confidence;
} a_v4_extfn_estimate;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
value	double	估计值。
confidence	double	与估计相关联的置信度。置信度的变化范围为 0.0 至 1.0，其中 0.0 表示估计无效，1.0 表示估计经确认为真。

按列表排序 (a_v4_extfn_orderby_list)

请使用 `a_v4_extfn_orderby_list` 结构以描述表的 ORDER BY 属性。

实现

```
typedef struct a_v4_extfn_orderby_list {
    a_sql_uint32        number_of_elements;
    a_v4_extfn_order_el order_elements[1];      // there are
number_of_elements entries
} a_v4_extfn_orderby_list;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
number_of_elements	a_sql_uint32	条目数
order_elements[1]	a_v4_extfn_order_el	元素的顺序

描述
存在一些 *number_of_elements* 条目，其中每个条目都含有一个表明 元素是升序还是降序的标志，以及一个 指示相关表中对应列的列索引。

另请参见
- 列顺序 (a_v4_extfn_order_el) （第 192 页）

通过列号分区 (a_v4_extfn_partitionby_col_num)

a_v4_extfn_partitionby_col_num 枚举类型表示列号，允许 UDF 表示对 **PARTITION BY** 的支持（类似于 SQL 所提供的支持）。

实现

```
typedef enum a_v4_extfn_partitionby_col_num {
    EXTFNAPIV4_PARTITION_BY_COLUMN_NONE = -1,        // NO PARTITION
BY
    EXTFNAPIV4_PARTITION_BY_COLUMN_ANY  = 0,         // PARTITION BY
ANY
                                  // + INTEGER representing a specific
column ordinal
} a_v4_extfn_partitionby_col_num;
```

成员摘要

枚举类型 a_v4_extfn_partitionby_col_num 的成员	值	描述
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE	-1	不进行分区
EXTFNAPIV4_PARTITION_BY_COLUMN_ANY	0	通过 任意正整数（代表特定的列顺序号）进行分区
Column Ordinal Number	N > 0	进行分区的表列号的序号

描述
此结构允许 UDF 以编程方式描述分区以及进行分区的列。

当填充 a_v4_extfn_column_list number_of_columns 字段时，使用此枚举类型。当向服务器描述对分区操作的支持情况时，UDF 将 number_of_columns 设置为一个枚举值，或将其设置为一个代表列序号的正整数。 例如，若要向服务器描述不支持分区操作，则需要创建如下结构：

```
a_v4_extfn_column_list nopby = {
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE,
0
};
```

EXTFNAPIV4_PARTITION_BY_COLUMN_ANY 成员通知服务器，UDF 支持任何形式的分区操作。

若要描述进行分区的一组序号，则需创建如下结构：

```
a_v4_extfn_column_list nopby = {
2,
3, 4
};
```

此结构表明将通过 2 列进行分区，其序号分别分别为 3 和 4。

注意： 此例仅用于描述目的，并非合法代码。调用方必须相应地分配容纳 3 个整数的结构。

Row (a_v4_extfn_row)

请使用 a_v4_extfn_row 结构表示单行中的数据。

实现

```
/* a_v4_extfn_row - */
typedef struct a_v4_extfn_row {
    a_sql_uint32           *row_status;
    a_v4_extfn_column_data    *column_data;
} a_v4_extfn_row;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
row_status	a_sql_uint32 *	行的状态。对于存在的行，将该值设置为 1，否则将该值设置为 0。
column_data	a_v4_extfn_column_data *	行的列数据数组。

描述

行结构包含特定行列的信息。此结构定义了一个单独行的状态，并且包括一个指向行内单独列的指针。 行的状态是一个表明该行是否存在的标志。可以使用嵌套的提取调用更改行的状态标志，而无需对行块结构进行处理。

将 *row_status* 标志设置为 1，则表明该行可用并且可被包括在结果集中。将 *row_status* 设置为 0，则表明应忽略该行。当使用 TPF 作为过滤器时，这一点非常有用。因为 TPF 可能会将输入表中的各行传递至结果集，但又想跳过其中的某些行，此时，它就可以通过将这些行的状态标志设置为 0 来予以实现。

另请参见

- 列数据 (a_v4_extfn_column_data) （第 190 页）

行块 (a_v4_extfn_row_block)

请使用 `a_v4_extfn_row_block` 结构表示行块中的数据。

实现

```
/* a_v4_extfn_row_block - */
typedef struct a_v4_extfn_row_block {
    a_sql_uint32        max_rows;
    a_sql_uint32        num_rows;
    a_v4_extfn_row  *row_data;
} a_v4_extfn_row_block;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
max_rows	a_sql_uint32	该行块能够处理的最大行数
num_rows	a_sql_uint32	必须小于或等于行块所包含的最大行数
row_data	a_v4_extfn_row *	行块矢量

描述

`fetch_into` 和 `fetch_block` 方法使用行块结构以实现数据生成及数据消耗。分配器设置最大行数。生产者错误地设置了行数。数据消耗程序不应尝试执行超出所生成行数的读取操作。

由 `row_block` 结构的所有者确定 *max_rows* 数据成员的值。例如，当表 UDF 执行 `fetch_into` 时，服务器会将 *max_rows* 的值设为可以装入 128K 内存的行的数量。然而，当表 UDF 执行 `fetch_block` 时，将会自行确定 *max_rows* 的值。

约束和限制

num_rows 和 *max_rows* 的值皆大于 0。*num_rows* 必须小于等于 *max_rows*。对于有效行块而言，row_data 字段不应为 NULL。

表 (a_v4_extfn_table)

请使用 a_v4_extfn_table 结构以表示数据在表中的存储方法，以及消耗程序提取数据的方法。

实现

```
typedef struct a_v4_extfn_table {
    a_v4_extfn_table_func       *func;
    a_sql_uint32                number_of_columns;
} a_v4_extfn_table;
```

数据成员及数据类型的摘要

数据成员	数据类型	描述
func	a_v4_extfn_table_func *	该成员包含一组函数指针，供消耗程序提供结果数据使用
number_of_columns	a_sql_uint32 *	表中的列数

表上下文 (a_v4_extfn_table_context)

a_v4_extfn_table_context 结构表示表上的一个打开的结果集。

实现

```
typedef struct a_v4_extfn_table_context {

//    size_t struct_size;

    /* fetch_into() - fetch into a specified row_block. This entry point
        is used when the consumer has a transfer area with a specific format.
       The fetch_into() function will write the fetched rows into the provided row block.
    */
    short (UDF_CALLBACK *fetch_into)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *);

    /* fetch_block() - fetch a block of rows. This entry point is used
        when the consumer does not need the data in a particular format. For example,
        if the consumer is reading a result set and formatting it as HTML, the consumer
       does not care how the transfer area is layed out. The fetch_block() entry point is
        more efficient if the consumer does not need a specific layout.

        The row_block parameter is in/out. The first call should point to a NULL row
block.
        The fetch_block() call sets row_block to a block that can be consumed, and this
block
        should be passed on the next fetch_block() call.
    */
    short (UDF_CALLBACK *fetch_block)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block);

    /*  rewind() - this is an optional entry point. If NULL, rewind is not supported.
Otherwise,
        the rewind() entry point restarts the result set at the beginning of the table.
```

```
    */
    short (UDF_CALLBACK *rewind)(a_v4_extfn_table_context *);

    /* get_blob() - If the specified column has a blob object, return it.  The blob
         is returned as an out parameter and is owned by the caller.  This method should
         only be called on a column that contains a blob.  The helper macro
EXTFN_COL_IS_BLOB can
         be used to determine whether a column contains a blob.
    */
    short (UDF_CALLBACK *get_blob)(a_v4_extfn_table_context *cntxt,
                    a_v4_extfn_column_data *col,
                    a_v4_extfn_blob **blob);

   /* The following fields are reserved for future use and must be initialized to NULL.
*/
    void  *reserved1_must_be_null;
    void  *reserved2_must_be_null;
    void  *reserved3_must_be_null;
    void  *reserved4_must_be_null;
    void  *reserved5_must_be_null;

    a_v4_extfn_proc_context     *proc_context;
    void                        *args_handle;           // use in
a_v4_extfn_proc_context::get_value() etc.
    a_v4_extfn_table            *table;
    void                        *user_data;
    void                        *server_internal_use;

   /* The following fields are reserved for future use and must be initialized to NULL.
*/
    void  *reserved6_must_be_null;
    void  *reserved7_must_be_null;
    void  *reserved8_must_be_null;
    void  *reserved9_must_be_null;
    void  *reserved10_must_be_null;

} a_v4_extfn_table_context;
```

方法总结

数据类型	方法	描述
短整型	**fetch_into**	提取至一个指定的 `row_block`
短整型	**fetch_block**	提取行块
短整型	**rewind**	在表的开头重启结果集
短整型	**get_blob**	如果指定的列包含 BLOB 对象，返回 BLOB 对象。

数据成员及数据类型的摘要

数据成员	数据类型	描述
proc_context	a_v4_extfn_ proc_context *	指向过程上下文对象的指针。UDF 可以使用该指针设置错误、记录信息、取消操作等等。
args_handle	void*	服务器所提供参数的句柄。
table	a_v4_extfn_table *	指向打开的结果集表。调用 `a_v4_extfn_proc_context open_result_set` 后对其赋值。

数据成员	数据类型	描述
user_data	void*	使用外部例程所需的任何环境数据皆可对该指针赋值。
server_internal_use	void*	仅供内部使用。

描述
当消耗程序和生产者需要彼此不同的格式时，将 `a_v4_extfn_table_context` 结构作为生产者和消耗程序之间的中间层以帮助管理数据。

UDF 可以使用 `a_v4_extfn_table_context` 从输入表参数读取行。服务器或其他 UDF 可以使用 `a_v4_extfn_table_context` 从 UDF 的结果表中读取行。

服务器可以执行 `a_v4_extfn_table_context` 方法，从而 解决阻抗失配问题。

另请参见
- fetch_into （第 292 页）
- fetch_block （第 294 页）
- rewind （第 296 页）

fetch_into

`fetch_into` v4 API 方法将数据提取至指定的行块中。

声明

```
short fetch_into(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *)
```

用法
如果生产者不知道应该如何在内存中安排数据，则 `fetch_into` 方法将十分有用。当消耗程序拥有特定格式的传输区时，此方法将被用作入口点。`fetch_into()` 函数将提取的行写入所提供的行块。此方法是 `a_v4_extfn_table_context` 结构的一部分。

当消耗程序拥有数据传输区内存并且请求生产者使用该区时，请使用 `fetch_into` 方法。当消耗程序关注于数据传输区的设置方法， 并且由生产者将必要的数据复制到该区时， 请您使用 `fetch_into` 方法。

参数

参数	描述
cntxt	从 `open_result_set` API 获取的表上下文对象

参数	描述
row_block	用于存储提取结果的行块对象

返回
如果成功则返回 1，否则返回 0。

如果 UDF 返回 1，则消耗程序知道还有未提取的行，并将再次调用 `fetch_into` 方法。而如果 UDF 返回 0 ，则表示所有行已提取完毕，不再需要调用 `fetch_into` 方法。

请考虑如下过程定义，这是一个 TPF 函数的示例，它消耗输入参数表并将其生成为结果表。两者分别为通过 `get_value` 和 `set_value` v4 API 方法获取和返回的 SQL 值的实例。

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

此过程定义包含两个表对象：

- 名为 b 的输入表参数
- 返回的结果集表

下面的示例描述了调用方（在本例中为服务器）将如何提取输出表。服务器可能会决定使用 `fetch_into` 方法。调用实体（在本例中为 TPF）提取输入表。TPF 决定将使用哪一个提取 API。

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

本示例描述了，在从输入表提取／消耗之前，必须先使用基于 `a_v4_extfn_proc` 结构的 `open_result_set` API 建立表上下文。`open_result_set` 需要用到表对象，可通过 `get_value` API 获取。

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table         *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

当创建完表上下文后，`rs` 结构将执行 `fetch_into` API 并对各行执行提取操作。

```
a_v4_extfn_row_block    *rb = // get a row block to hold a series of
INT values.
rs->fetch_into( rs, &rb )  // fetch the rows.
```

在将各行生成为结果表之前，必须先创建表对象，而后使用基于 `a_v4_extfn_proc_context` 结构的 `set_value` API 将其返回至调用方。

此示例描述表 UDF 必须创建 a_v4_extfn_table 结构的实例。对表 UDF 的每次调用都应该返回单独的 a_v4_extfn_table 结构的实例。该表包含状态字段，用于跟踪当前行和生成行的数量。可以将表的状态存储为实例的一个字段。

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

在下面的示例中，每生成一行，则递增**current_row**，直至达到所生成的行的数量，此时 fetch_into 将返回 false 以表明到达文件末尾。 消耗程序执行由表 UDF 所实现的 fetch_into API。作为调用 fetch_into 方法的一部分，消耗程序提供了表上下文以及用于存储提取结果的行块。

```
     rs->fetch_into( rs, &rb )

short UDF_CALLBACK my_table_func_fetch_into(
    a_v4_extfn_table_context *tctx,
    a_v4_extfn_row_block *rb)
/*****************************************/
{
    my_table *myTable = tctx->table;

    if( rgTable->current_row < rgTable->rows_to_generate ) {
            // Produce the row...
    rgTable->current_row++;
    return 1;
    }

    return 0;
}
```

另请参见

- fetch_block 方法 （第 124 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- get_value （第 275 页）
- set_value （第 277 页）
- 表 (a_v4_extfn_table) （第 290 页）

fetch_block

fetch_block v4 API 方法对行块执行提取操作。

声明

```
short fetch_block(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block)
```

用法
当消耗程序不需要特殊格式的数据时，使用 `fetch_block` 方法作为入口点。`fetch_block` 请求生产者创建数据传输区，并提供指向该区的指针。消耗程序拥有数据传输区内存，并且负责从该区中复制数据。

如果消耗程序不需要特定布局，则 `fetch_block` 方法更为有效。 `fetch_block` 调用为可消耗块设置了 `fetch_block`，并且将该块传递至下一次 `fetch_block` 调用。此方法是 `a_v4_extfn_table_context` 结构的一部分。

参数

参数	描述
cntxt	表上下文对象。
row_block	输入／输出参数。首次调用应该总是指向一个空 **row_block**。

当调用 `fetch_block` 并且 **row_block** 指向 NULL 时，UDF 必须分配一个 `a_v4_extfn_row_block` 结构。

返回
如果成功则返回 1，否则返回 0。

如果 UDF 返回 1，则消耗程序知道还有未提取的行，并将再次调用 `fetch_block` 方法。而如果 UDF 返回 0 ，则表示所有行已提取完毕，不再需要调用 `fetch_block` 方法。

请考虑如下过程定义，这是一个 TPF 函数的示例，它消耗输入参数表并将其生成为结果表。两者分别为通过 `get_value` 和 `set_value` v4 API 方法获取和返回的 SQL 值的实例。

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

此过程定义包含两个表对象：

- 名为 b 的输入表参数
- 返回的结果集表

下面的示例描述了调用方（在本例中为服务器）将如何提取输出表。服务器可能会决定使用 `fetch_block` 方法。调用实体（在本例中为 TPF）提取输入表，并决定将使用哪一个提取 API。

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

本示例描述了，在从输入表提取／消耗之前，必须先使用基于 `a_v4_extfn_proc` 结构的 `open_result_set` API 建立表上下文。`open_result_set` 需要用到表对象，可通过 `get_value` API 获取。

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table                *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

当创建完表上下文后，`rs` 结构将执行 `fetch_block` API 并对各行执行提取操作。

```
a_v4_extfn_row_block    *rb =  // get a row block to hold a series of
INT values.
rs->fetch_block( rs, &rb )  // fetch the rows.
```

在将各行生成为结果表之前，必须先创建表对象，而后使用基于 `a_v4_extfn_proc_context` 结构的 `set_value` API 将其返回至调用方。

此示例描述表 UDF 必须创建 `a_v4_extfn_table` 结构的实例。对表 UDF 的每次调用都应该返回单独的 `a_v4_extfn_table` 结构的实例。该表包含状态字段，用于跟踪当前行和生成行的数量。可以将表的状态存储为实例的一个字段。

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

另请参见

- fetch_block 方法 （第 123 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）
- 外部过程上下文 (a_v4_extfn_proc_context) （第 273 页）
- get_value （第 275 页）
- set_value （第 277 页）
- open_result_set （第 283 页）
- 表 (a_v4_extfn_table) （第 290 页）

rewind

使用 `rewind` 第 4 版 API 方法从表的开头重新启动结果集。

声明

```
short rewind(
       a_v4_extfn_table_context          *cntxt,
       )
```

用法
对已打开的结果集调用 `rewind` 方法，即可回绕到表的开头。如果 UDF 要回绕输入表，则必须在 **EXTFNAPIV4_STATE_OPTIMIZATION** 状态下用 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND** 参数通知生产程序。

`rewind()` 是可选条目。如果是空表，则不支持回绕。否则， `rewind()` 条目会在表的开头重新启动结果集。

参数

参数	描述
cntxt	表上下文对象

返回
如果成功则返回 1，否则返回 0。

另请参见
- 查询优化状态 （第 118 页）
- 执行状态 （第 121 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）

get_blob

请使用 `get_blob` v4 API 方法从指定列返回 BLOB 对象。

声明

```
 short get_blob(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_column_data *col,
a_v4_extfn_blob **blob
)
```

用法
BLOB 作为输出参数返回，并为调用方所有。仅对包含 BLOB 对象的列调用此方法。

请使用帮助程序宏 `EXTFN_COL_IS_BLOB` 来确定列是否包含 BLOB 对象。这是头文件 `extfnapiv4.h` 中的 `EXTFN_COL_IS_BLOB` 声明：

```
 #define EXTFN_COL_IS_BLOB(c, n)        (c[n].blob_handle != NULL)
```

参数

参数	描述
cntxt	表上下文对象
col	为其获取 BOLB 的列数据指针
blob	成功时，则包含与列相关的 BLOB 对象

返回
如果成功则返回 1，否则返回 0。

另请参见
- 表上下文 (a_v4_extfn_table_context) （第 290 页）

表函数 (a_v4_extfn_table_func)

消耗程序使用 `a_v4_extfn_table_func` 结构检索来自生产者的结果。

实现

```
typedef struct a_v4_extfn_table_func {
//    size_t struct_size;

    /*  Open a result set. The UDF can allocate any resources needed
for the result set.
    */
    short (UDF_CALLBACK *_open_extfn)(a_v4_extfn_table_context *);

   /*  Fetch rows into a provided row block. The UDF should implement
this method if it does
        not have a preferred layout for its transfer area.
    */
   short (UDF_CALLBACK *_fetch_into_extfn)(a_v4_extfn_table_context
*, a_v4_extfn_row_block

*row_block);

   /*  Fetch a block that is allocated and configured by the UDF. The
UDF should implement this
        method if it has a preferred layout of the transfer area.
    */
    short (UDF_CALLBACK *_fetch_block_extfn)
(a_v4_extfn_table_context *, a_v4_extfn_row_block

**row_block);

   /*  Restart a result set at the beginning of the table. This is an
optional entry point.
```

```
    */
    short (UDF_CALLBACK *_rewind_extfn)(a_v4_extfn_table_context *);

    /*  Close a result set. The UDF can release any resources
allocated for the result set.
    */
    short (UDF_CALLBACK *_close_extfn)(a_v4_extfn_table_context *);

    /* The following fields are reserved for future use and must be
initialized to NULL. */
    void *_reserved1_must_be_null;
    void *_reserved2_must_be_null;

} a_v4_extfn_table_func;
```

方法总结

方法	数据类型	描述
_open_extfn	无类型	服务器调用该方法，通过打开结果集来启动行提取操作。UDF 可以分配结果集所需的任何资源。
_fetch_into_extfn	短整型	将行提取至所提供的行块中。如果 UDF 的传输区未设置首选布局，则它将执行此方法。
_fetch_block_extfn	短整型	提取由 UDF 所分配和配置的块。如果 UDF 的传输区设置了首选布局，则它将执行此 方法。
_rewind_extfn	无类型	可选函数，服务器调用该函数从表的开头重启提取操作。
_close_extfn	无类型	服务器调用该方法，通过关闭结果集来终止行提取操作。UDF 可以释放已分配给结果集的任何资源。
_reserved1_must_be_null	无类型	留作将来使用。必须初始化为 NULL。
_reserved1_must_be_null	无类型	留作将来使用。必须初始化为 NULL。

描述

a_v4_extfn_table_func 结构定义了从表提取结果的方法。

另请参见

- 表 (a_v4_extfn_table) （第 290 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- _open_extfn （第 300 页）
- _fetch_into_extfn （第 300 页）

- _fetch_block_extfn （第 301 页）
- _rewind_extfn （第 301 页）
- _close_extfn （第 302 页）

_open_extfn

服务器会调用 `_open_extfn` 第 4 版 API 方法，以便开始提取行。

声明

```
void _open_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

用法

UDF 用此方法打开结果集并分配向服务器发送结果所需的所有资源（如流）。

参数

参数	描述
cntxt	过程上下文对象

另请参见

- 表上下文 (a_v4_extfn_table_context) （第 290 页）

_fetch_into_extfn

`_fetch_into_extfn` 第 4 版 API 方法用于将行提取到所提供的行块中。

声明

```
short _fetch_into_extfn(
    a_v4_extfn_table_context *cntxt,
    a_v4_extfn_row_block *row_block
    )
```

用法

如果 UDF 的传输区域没有首选布局， 则 UDF 应该实现此方法。

参数

参数	描述
cntxt	过程上下文对象
row_block	要提取到的行块对象。

返回

如果成功则返回 1，否则返回 0。

另请参见

- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）

_fetch_block_extfn

_fetch_block_extfn 第 4 版 API 方法用于提取由 UDF 分配和配置的块。

声明

```
short _fetch_block_extfn(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **
)
```

用法

如果 UDF 的传输区域有首选布局，则 UDF 应该实现此方法。

参数

参数	描述
cntxt	过程上下文对象
row_block	要提取到的行块对象

返回

如果成功则返回 1，否则返回 0。

另请参见

- 表上下文 (a_v4_extfn_table_context) （第 290 页）
- 行块 (a_v4_extfn_row_block) （第 289 页）

_rewind_extfn

_rewind_extfn 第 4 版 API 方法用于从表的开头重新启动结果集。

声明

```
void _rewind_extfn(
a_v4_extfn_table_context *cntxt,
)
```

用法

此函数是可选条目。回绕到结果表开头后，UDF 会实现 _rewind_extfn 方法。仅当 UDF 能以高效经济的方式提供回绕功能时，UDF 才会该考虑实现这种方法。

如果 UDF 决定实现 `_rewind_extfn` 方法，则应该在 **EXTFNAPIV4_STATE_OPTIMIZATION** 状态下通过设置参数 0 的 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND** 参数告知消耗程序。

UDF 可以决定不提供回绕功能，这种情况下服务器会进行补偿并提供回绕功能。

注意： 服务器可选择不调用 `_rewind_extfn` 方法来执行回绕。

参数

参数	描述
cntxt	过程上下文对象

返回
无返回值。

另请参见

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute（获取）（第 241 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性（设置）（第 256 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性（设置）（第 258 页）
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute（获取）（第 242 页）
- 查询处理状态 （第 115 页）
- 执行状态 (a_v4_extfn_state) （第 268 页）
- 表上下文 (a_v4_extfn_table_context) （第 290 页）

_close_extfn

服务器调用 `_close_extfn` v4 API 方法来终止对行的提取操作。

声明

```
void _close_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

用法
当提取操作执行完毕之后，UDF 使用此方法关闭结果集，并释放分配给该结果集的所有资源。

参数

参数	描述
cntxt	过程上下文对象

另请参见

- 表上下文 (a_v4_extfn_table_context) （第 290 页）

a_v4_extfn 的 API 故障排除

`describe_column`、`describe_parameter` 和 `describe_udf` 第 4 版 API 方法都能返回通用错误消息。执行服务器中没有的 UDF 会返回不能执行语句错误。

通用 describe_column 错误

因为执行 **describe_column** 获取和设置调用而返回的常见错误。

获取	设置
EXTFNAPIV4_DESCRIBE_ BUFFER_ SIZE_MISMATCH – 如果 **cntxt** 或 **describe_buffer** 是空值，或者 **describe_buffer_length** 为 0，都会返回获取错误。	EXTFNAPIV4_DESCRIBE_ BUFFER_ SIZE_MISMATCH – 如果 **cntxt** 或 **describe_buffer** 是空值，或者 **describe_buffer_length** 为 0，都会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_ STATE – 如果 **cntxt** 是无效的上下文参数，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_ STATE – 如果 **cntxt** 是无效的上下文参数，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_ PARAMETER – 如果提供的参数数量超出了过程的合法范围，则会返回获取错误：**< 0** 或 **>** 过程的参数数量。	EXTFNAPIV4_DESCRIBE_INVALID_ PARAMETER – 如果提供的参数数量超出了过程的合法范围，则会返回设置错误：**< 0** 或者 **>** 过程的参数数量。
EXTFNAPIV4_DESCRIBE_NON_TA- BLE_PARAMETER – 如果 **arg_num** 不是表参数，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_NON_TA- BLE_PARAMETER – 如果 **arg_num** 不是表参数，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_ COLUMN – 如果列号对于表参数无效，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_ COLUMN – 如果列号对于表参数无效，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_UNKNOWN_ ATTRIBUTE – 如果 **describe_type** 值不是 `a_v4_extfn_describe_parm_type` 中的有效描述类型，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_UNKNOWN_ ATTRIBUTE – 如果 **describe_type** 值不是 `a_v4_extfn_describe_parm_type` 中的有效描述类型，则会返回设置错误。

通用 describe_udf 错误

因为执行 describe_udf 获取和设置调用而返回的常见错误。

获取	设置
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果任一 **cntxt** 或 **describe_buffer** 参数是空值，或者 **describe_buffer_length** 是 0，都会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER — 如果任一 **cntxt** 或 **describe_buffer** 参数是空值，或者 **describe_buffer_length** 是 0，都会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 是无效的上下文参数，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 是无效的上下文参数，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE – 如果 **describe_type** 值不是 a_v4_extfn_describe_udf_type 描述类型之一，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE – 如果 **describe_type** 值不是 a_v4_extfn_describe_udf_type 描述类型之一，则会返回设置错误。

通用 describe_parameter 错误

因为执行 describe_parameter 获取和设置调用而返回的常见错误。

获取	设置
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 或者 **describe_buffer** 是空值，或者如果 **describe_buffer_length** 为 0，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 或者 **describe_buffer** 是空值，或者如果 **describe_buffer_length** 为 0，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 参数无效，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果 **cntxt** 参数无效，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果提供的参数数量超出了过程的合法范围（即，如果参数数量少于 0，或者多于过程的参数数量），则会返回获取错误。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 如果提供的参数数量超出了过程的合法范围（即，如果参数数量少于 0，或者多于过程的参数数量），则会返回设置错误。

获取	设置
EXTFNAPIV4_DESCRIBE_ UNKNOWN_ ATTRIBUTE – 如果 **describe_type** 是无效 a_v4_extfn_describe_parm_type 描述类型，则会返回获取错误。	EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – 如果 **describe_type** 是无效 a_v4_extfn_describe_parm_type 描述类型，则会返回设置错误。
EXTFNAPIV4_DESCRIBE_BUFFER_ SIZE_MISMATCH – 如果请求的属性大小和提供的 describe_buffer_length 之间存在差异，则会返回获取错误。对于固定长度的属性，例如 a_sql_byte 数据类型，其大小必须相互匹配。对于可变长度的属性数据类型，例如 char[]，所提供的缓冲区至少应能装下请求的属性值。	EXTFNAPIV4_DESCRIBE_BUFFER_ SIZE_MISMATCH – 如果请求的属性大小和提供的 **describe_buffer_length** 之间存在差异，则会返回设置错误。对于固定长度的属性，例如 a_sql_byte 数据类型，其大小必须相互匹配。

另请参见

- 查询处理状态 （第 115 页）

缺失 UDF 将返回错误

当尝试执行一个服务器上不存在的 UDF 时将返回错误。

如您尝试执行类似如下的查询：

```
select my_sum1(n_tabkey) from tabudf()
```

其中：

- tabudf() 是一个表 UDF，并且
- 服务器上不存在 my_sum1() UDF。

返回如下错误：

```
Could not execute statement.
External procedures or functions are not allowed across server types.
SQLCODE=-1579, ODBC 3 State="HY000"
Line 1, column 1
```

外部 UDF 环境

UDF 定义不当可能会引起内存违规，也可能会引发数据库服务器故障。所以，在数据库服务器的外部（即在外部环境中）运行 UDF，可消除服务器的这种风险。

如果外部环境中出现运行时异常，则服务器进程不会受影响。服务器会向 UDF 调用方报错，所有后续 UDF 调用都会使外部环境重新启动。

注意： 使用外部运行时环境无需得到 IQ_UDF 或 IQ_IDA 授权。使用外部运行时环境无需调用 `a_v3_extfn` 或 `a_v4_extfn` API。

对于 UDF，数据库服务器支持以下外部运行时环境：

- CLR（Microsoft .NET 公共语言运行库）
- ESQL 和 ODBC（嵌有 C/C++ 的 SQL 或 ODBC 服务器端请求）
- Java
- Perl
- PHP

环境各有各的一套 API，用于处理参数及向服务器返回值。例如，Java 外部环境采用 JDBC API。

系统表

系统表 SYSEXTERNENV 用来存储辨别和启动每个外部环境所需的信息。

系统表 SYSEXTERNENVOBJECT 用于存储非 Java 外部对象。

SQL 语句

可通过以下 SQL 语法设置或修改外部环境在 SYSEXTERNENV 表中的位置。

```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ LOCATION location-string ]
```

某个外部环境被设置为在数据库服务器上使用后，即可以在数据库中安装对象并在外部环境内创建使用这些对象的存储过程和函数。这些对象、存储过程和存储函数的安装、创建与使用，都类似于 Java 类安装及 Java 存储过程和函数的创建和使用方法。

要添加针对外部环境的注释，可以执行以下语句：

```
COMMENT ON EXTERNAL ENVIRONMENT environment-name
 IS comment-string
```

要通过文件或表达式将 Perl 或 PHP 外部对象（如 Perl 脚本）安装到数据库中，请执行类似以下代码的 **INSTALL EXTERNAL OBJECT** 语句：

```
INSTALL EXTERNAL OBJECT object-name-string
  [ update-mode ]
```

```
 FROM { FILE file-path | VALUE expression }
 ENVIRONMENT environment-name
```

对于安装好的 Perl 或 PHP 外部对象，要添加注释，可以执行以下语句：

```
COMMENT ON EXTERNAL [ENVIRONMENT] OBJECT object-name-string
 IS comment-string
```

要从数据库中删除已安装好的 Perl 或 PHP 外部对象，请使用 **REMOVE EXTERNAL OBJECT** 语句：

```
REMOVE EXTERNAL OBJECT object-name-string
```

外部对象安装在数据库中后，即可在外部存储过程和函数定义中使用（类似于当前的 Java 存储过程和函数创建机制）。

```
CREATE PROCEDURE procedure-name(...)
 EXTERNAL NAME '...'
 LANGUAGE environment-name

CREATE FUNCTION function-name(...)
 RETURNS ...
 EXTERNAL NAME '...'
 LANGUAGE environment-name
```

这些存储过程和函数创建完毕后，即可像数据库中任何其他存储过程或函数一样使用。遇到外部环境存储过程或函数时，数据库服务器会自动启动外部环境（如果尚未启动），并发送获取外部环境所需的信息，以便从数据库中获取并执行外部对象。根据需要，会返回执行后产生的任何结果集或返回值。

要按需启动或停止外部环境，请使用 **START EXTERNAL ENVIRONMENT** 和 **STOP EXTERNAL ENVIRONMENT** 语句：

```
 START EXTERNAL ENVIRONMENT environment-name
STOP EXTERNAL ENVIRONMENT environment-name
```

在外部环境中执行 UDF

在 CLR、ESQL 和 ODBC、Java、Perl 或 PHP 外部环境中执行 UDF。

前提条件

没有授权前提条件。使用外部运行时环境无需拥有 IQ_IDA 许可证。使用外部运行时环境无需调用 `a_v3_extfn` 或 `a_v4_extfn` API。

过程

1. 在数据库服务器中设置要使用的外部环境。

   ```
   ALTER EXTERNAL ENVIRONMENT environment-name
       [ LOCATION location-string ]
   ```

2. 将外部对象（CLR、ESQL 和 ODBC、Java、Perl 或 PHP）安装到数据库中。

3. 用 **CREATE PROCEDURE** 和 **CREATE FUNCTION** 语句创建使用这些外部环境中的对象的存储过程和函数。
4. 引用所创建的存储过程或函数。在查询的 **FROM** 子句中引用存储过程。

另请参见

- CLR 外部环境 （第 311 页）
- ESQL 和 ODBC 外部环境 （第 314 页）
- Java 外部环境 （第 322 页）
- PERL 外部环境 （第 344 页）
- PHP 外部环境 （第 348 页）
- CREATE PROCEDURE 语句 (Java UDF) （第 339 页）
- CREATE FUNCTION 语句 (Java UDF) （第 341 页）

外部环境限制

所有外部 UDF 环境都受限制。

- 不支持 **NO RESULT SET** 选项。
- 只支持 **IN** 参数：不支持 **INOUT/OUT**。
- 不许使用结果值为 `LONG VARCHAR` 或 `LONG BINARY` 的函数。

另请参见

- Java 外部环境限制 （第 329 页）

CLR 外部环境

数据库服务器支持 CLR 存储过程和函数。CLR 存储过程或函数与 SQL 存储过程或函数的的行为基本相同，只是过程或函数的代码用 C# 或 Visual Basic 之类的 .NET 语言编写，并且在数据库服务器外（即在单独的 .NET 可执行文件内）执行。

每个数据库只有一个此 .NET 可执行文件的实例。执行 CLR 函数和存储过程的所有连接都使用同一个 .NET 可执行实例，但每个连接的命名空间是独立的。静态在连接期间内会一直保持，但在连接之间不共享。仅支持 .NET 版本 2.0。

要调用外部 CLR 函数或过程，需要在定义相应存储过程或函数时指定 EXTERNAL NAME 字符串，确定装载哪个 DLL 以及调用程序集中的哪个函数。此外，在定义存储过程或函数时，还必须指定 LANGUAGE CLR。以下是一个声明示例：

```
CREATE PROCEDURE clr_stored_proc(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
EXTERNAL NAME 'MyCLRTest.dll::MyCLRTest.Run( int, ushort, out
```

```
string )'
LANGUAGE CLR;
```

在此例中，存储过程调用了 **clr_stored_proc**，执行时会装载 DLL `MyCLRTest.dll` 并调用函数 **MyCLRTest.Run**。**clr_stored_proc** 过程可接收三个 SQL 参数，即两个分别为 INT 类型和 UNSIGNED SMALLINT 类型的 IN 参数和一个 LONG VARCHAR 类型的 OUT 参数。在 .NET 端，这三个参数转换为 int 类型和 ushort 类型的输入参数和 string 类型的输出参数。除了 out 参数，CLR 函数还可以具有 ref 参数。如果相应的存储过程有一个 INOUT 参数，则用户必须声明一个 ref CLR 参数。

下表所列的是各种 CLR 参数类型，以及建议采用的相应 SQL 数据类型：

CLR 类型	建议采用的 SQL 数据类型
bool	bit
byte	tinyint
short	smallint
ushort	unsigned smallint
int	int
uint	unsigned int
long	bigint
ulong	unsigned bigint
decimal	numeric
float	real
double	double
DateTime	timestamp
string	long varchar
byte[]	long binary

DLL 的声明可以使用相对路径也可以使用绝对路径。如果指定的路径为相对路径，则外部 .NET 可执行文件会在该路径及其他位置搜索 DLL。可执行文件不会在全局程序集高速缓存 (GAC) 中搜索 DLL。

与现有 Java 存储过程和函数一样，CLR 存储过程和函数可以发出返回到数据库的服务器端请求，还可以返回结果集。而且，像 Java 一样，任何输出到 Console.Out 和 Console.Error 的信息都会自动重定向到数据库服务器消息窗口。

在数据库中使用 CLR，需确保数据库服务器能够找到并启动 CLR 可执行文件。通过执行以下语句可以验证数据库服务器是否能够找到并启动 CLR 可执行文件：

```
START EXTERNAL ENVIRONMENT CLR;
```

如果数据库服务器未能启动 CLR，则可能是数据库服务器无法找到 CLR 可执行文件。CLR 可执行文件是 dbextclr12.exe。

请注意，除了验证数据库服务器可以启动 CLR 可执行文件外，START EXTERNAL ENVIRONMENT CLR 语句并不是必需的。通常，进行 CLR 存储过程或函数的调用会自动启动 CLR。

类似地，停止 CLR 的实例时 STOP EXTERNAL ENVIRONMENT CLR 语句也不是必需的，因为连接终止时实例会自动消失。然而，如果要彻底离开 CLR 并且想要释放一些资源，STOP EXTERNAL ENVIRONMENT CLR 语句则可以为您的连接释放 CLR 实例。

与 Perl、PHP 和 Java 外部环境不同，CLR 环境不需要在数据库中安装任何内容。因此，使用 CLR 外部环境前，无需执行任何 INSTALL 语句。

下面是一个可以在外部环境中运行、以 C# 编写的函数的示例。

```
public class StaticTest
{
    private static int val = 0;

    public static int GetValue() {
        val += 1;
        return val;
    }
}
```

此函数编译到动态链接库后，即可从外部环境调用。数据库服务器会启动名为 `dbextclr12.exe` 的可执行文件映像，用以为您装载该动态链接库。

要用 Microsoft C# 编译器将此应用程序内置到动态链接库中，请使用类似以下命令的命令。上例的源代码假定位于名为 `StaticTest.cs` 的文件中。

```
csc /target:library /out:clrtest.dll StaticTest.cs
```

此命令可将编译过的代码放在名为 **clrtest.dll** 的 DLL 中。要调用编译后的 C# 函数 GetValue，应使用 Interactive SQL 按如下方式定义包装：

```
CREATE FUNCTION stc_get_value()
RETURNS INT
EXTERNAL NAME 'clrtest.dll::StaticTest.GetValue() int'
LANGUAGE CLR;
```

对于 CLR，EXTERNAL NAME 字符串以 SQL 的单独一行指定。为了能够找到 DLL，您可能需要在 EXTERNAL NAME 字符串中包含该 DLL 的路径。对于相关程序集（例如，myLib.dll 的代码调用的函数在 myOtherLib.dll 中或以某种方式与其相关），则由 .NET Framework 决定装载依赖关系。CLR 外部环境将对指定程序集的装载进行处理，但可能需要执行额外的步骤才能确保装载相关程序集。一种解决方案是，通过使用与 .NET Framework 一起安装的 Microsoft gacutil 实用程序，在全局程序集高速缓存 (GAC) 中注册所有依赖关系。对于自定义开发库，gacutil 要求先使用强命名密钥对库进行签名，然后才能在 GAC 中注册这些库。

要执行该编译后 C# 函数示例，请执行以下语句。

```
SELECT stc_get_value();
```

每次调用 C# 函数都会生成一个新的整数结果。返回值的顺序是 1、2、3 以此类推。

ESQL 和 ODBC 外部环境

要在外部环境而非数据库服务器中运行编译过的本地 C 函数，要用 EXTERNAL NAME 子句并在其后使用 LANGUAGE 属性定义存储过程或函数，从而指定 C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64 之一。

与 Perl、PHP 和 Java 外部环境不同的是，不需要在数据库中安装任何源代码或编译过的对象。因此，使用 ESQL 和 ODBC 外部环境前，无需执行任何 INSTALL 语句。

下面是一个可以在数据库服务器或外部环境中运行、以 C++ 编写的函数的示例。

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

// Note: extfn_use_new_api used only for
// execution in the database server

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intptr;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
```

```
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intptr = (int *) arg.data;
        k += *intptr * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

此函数在编译到动态链接库或共享对象中后，即可从外部环境调用。数据库服务器会启动名为 dbexternc12 的可执行文件映像，用以为您装载该动态链接库或共享对象。

请注意，32 位或 64 位版本的数据库服务器都可以使用，任何一个版本都可以启动 32 位或 64 位版本的 dbexternc12。这是使用外部环境的优点之一。请注意，dbexternc12 一旦由数据库服务器启动就会一直运行，直到连接终止或执行了 STOP EXTERNAL ENVIRONMENT 语句（包含正确的环境名）为止。每个用于执行外部环境调用的连接都将获得一个专用 dbexternc12 副本。

要调用编译过的本地函数 SimpleCFunction，应按如下方式定义包装：

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@c:\\c\\extdemo.dll'
LANGUAGE C_ODBC32;
```

这与在将编译后本地函数加载到数据库服务器地址空间时该函数的描述方式几乎完全相同。唯一的区别是使用了 LANGUAGE C_ODBC32 子句。该子句说明 SimpleCDemo 是一个在外部环境中运行的函数，它使用 32 位 ODBC 调用。C_ESQL32、C_ESQL64、C_ODBC32 或 C_ODBC64 的语言规范可向数据库服务器表明：发出服务器端请求时，外部 C 函数是发出 32 位还是 64 位调用，以及这些调用是 ODBC、ESQL 调用还是 `a_v4_extfn` API 调用。

如果本地函数不使用 ODBC、ESQL 或 SQL Anywhere C API 调用中的任何一个执行服务器端请求，则 C_ODBC32 或 C_ESQL32 可用于 32 位应用程序，C_ODBC64 或 C_ESQL64 可用于 64 位应用程序。这是以上所示的外部 C 函数中的情况。它未使用这些 API 中的任何一个。

要执行该编译后本地函数示例，请执行以下语句。

```
SELECT SimpleCDemo(1,2,3,4);
```

要使用服务器端 ODBC，C/C++ 代码必须使用缺省数据库连接。要获取数据库连接的句柄，请用 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会告知数据库服务器返回当前外部环境连接，而不是打开一个新连接。

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;

    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }

    HDBC dbc = (HDBC)arg.data;
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );
```

```
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

如果以上 ODBC 代码存储在文件 extodbc.cpp 中，则可用以下命令为 Windows 生成该文件。

```
cl extodbc.cpp /LD /Ic:\sa12\sdk\include odbc32.lib
```

下面的示例将创建一个表，定义用来调用编译后本地函数的存储过程包装，然后调用该本地函数来填充该表。

```
CREATE TABLE odbcTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;

SELECT ServerSideODBC();

// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

同样，要使用服务器端 ESQL，C/C++ 代码也必须使用缺省数据库连接。要获取数据库连接的句柄，请用 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会告知数据库服务器返回当前外部环境连接，而不是打开一个新连接。

```
#include <windows.h>
#include <stdio.h>

#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
```

```
    an_extfn_value      arg;
    an_extfn_value      retval;

    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;

    int ret = -1;

    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;

    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;

    api->set_value( arg_handle, 0, &retval, 0 );
}
```

如果以上嵌入式 SQL 语句存储在文件 extesql.sqc 中，则可用以下命令为 Windows 生成该文件。

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:\sa12\sdk\include c:\sa12\sdk\lib
\x86\dblibtm.lib
```

下面的示例将创建一个表，定义用来调用编译后本地函数的存储过程包装，然后调用该本地函数来填充该表。

```
CREATE TABLE esqlTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideESQL();
```

```
// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

如同前面的示例，要使用服务器端 SQL Anywhere C API 调用，C/C++ 代码必须使用缺省数据库连接。要获取数据库连接的句柄，请用 EXTFN_CONNECTION_HANDLE_ARG_NUM 参数调用 get_value。该参数会告知数据库服务器返回当前外部环境连接，而不是打开一个新连接。下例所示的框架用于获得连接句柄，初始化 C API 环境，以及将连接句柄转换成可与 SQL Anywhere C API 一起使用的连接对象 (a_sqlany_connection)。

```
include <windows.h>
#include "sacapidll.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    unsigned            offset;
    char                *cmd;

    SQLAnywhereInterface  capi;
    a_sqlany_connection * sqlany_conn;
    unsigned int          max_api_ver;

    result = extapi->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );

    if( result == 0 || arg.data == NULL )
    {
        return;
    }
    if( !sqlany_initialize_interface( &capi, NULL ) )
    {
        return;
    }
    if( !capi.sqlany_init( "MyApp",
            SQLANY_CURRENT_API_VERSION,
            &max_api_ver ) )
    {
        sqlany_finalize_interface( &capi );
        return;
```

```
    }
    sqlany_conn = sqlany_make_connection( arg.data );

    // processing code goes here

    capi.sqlany_fini();

    sqlany_finalize_interface( &capi );
    return;
}
```

如果以上 C 代码存储在文件 extcapi.c 中，则可用以下命令为 Windows 生成该文件。

```
cl /LD /Tp extcapi.c /Tp c:\sa12\SDK\C\sacapidll.c
      /Ic:\sa12\SDK\Include c:\sa12\SDK\Lib\X86\dbcapi.lib
```

下例用于定义用来调用编译过的本地函数的存储过程包装，然后调用该本地函数。

```
CREATE FUNCTION ServerSideC()
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extcapi.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideC();
```

上例中的 LANGUAGE 属性用于指定 C_ESQL32。对于 64 位应用程序要使用 C_ESQL64。必须使用嵌入式 SQL 语言属性，因为 SQL Anywhere C API 构建在与 ESQL 相同的层（库）上。

如前所述，每个用于执行外部环境调用的连接都将启动各自的 dbexternc12 副本。首次执行外部环境调用时，此可执行应用程序由服务器自动装载。但是，可以使用 START EXTERNAL ENVIRONMENT 语句预装载 dbexternc12。这在想要避免在首次执行外部环境调用时出现的略微延迟的情况下很有用。以下是该语句的示例。

```
START EXTERNAL ENVIRONMENT C_ESQL32
```

预装载 dbexternc12 在另一种情况也很有用，即想要调试外部函数时。可以使用调试程序连接到正在运行的 dbexternc12 过程，并在外部函数中设置断点。

更新动态链接库或共享对象时，STOP EXTERNAL ENVIRONMENT 语句很有用。该语句将为当前连接终止本地库装载程序 dbexternc12，从而释放对动态链接库或共享对象的访问。如果多个连接在使用同一个动态链接库或共享对象，则必须终止其每一个 dbexternc12 副本。必须在 STOP EXTERNAL ENVIRONMENT 语句中指定相应的外部环境名称。以下是该语句的示例。

```
STOP EXTERNAL ENVIRONMENT C_ESQL32
```

要从外部函数返回结果集，编译过的本地函数必须使用本地函数调用接口。

以下代码段所示的是如何设置结果集信息结构。它包含列计数、指向列信息结构数组的指针，以及指向列数据值结构数组的指针。本示例也使用 SQL Anywhere C API。

```
an_extfn_result_set_info    rs_info;

int columns = capi.sqlany_num_cols( sqlany_stmt );
```

```
an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );

an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );

rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

以下代码段所示的是如何描述结果集。它使用 SQL Anywhere C API 为先前由 C API 执行的 SQL 查询获取列信息。从 SQL Anywhere C API 获得的各列信息将被转换为用于描述结果集的列名称、类型、宽度、索引以及空值指示符。

```
a_sqlany_column_info        info;
for( int i = 0; i < columns; i++ )
{
    if( sqlany_get_column_info( sqlany_stmt, i, &info ) )
    {
        // set up a column  description
        col_info[i].column_name  = info.name;
        col_info[i].column_type  = info.native_type;
        switch( info.native_type )
        {
           case DT_DATE:      // DATE is converted to string by C API
           case DT_TIME:      // TIME is converted to string by C API
          case DT_TIMESTAMP:  // TIMESTAMP is converted to string by
C API
           case DT_DECIMAL:   // DECIMAL is converted to string by C
API
                col_info[i].column_type  = DT_FIXCHAR;
                break;
          case DT_FLOAT:     // FLOAT is converted to double by C API
                col_info[i].column_type  = DT_DOUBLE;
                break;
           case DT_BIT:       // BIT is converted to tinyint by C API
                col_info[i].column_type  = DT_TINYINT;
                break;
        }
        col_info[i].column_width = info.max_size;
      col_info[i].column_index = i + 1; // column indices are origin
1
        col_info[i].column_can_be_null = info.nullable;
    }
}
// send the result set description
if( extapi->set_value( arg_handle,
                    EXTFN_RESULT_SET_ARG_NUM,
                    (an_extfn_value *)&rs_info,
                    EXTFN_RESULT_SET_DESCRIBE ) == 0 )
{
    // failed
    free( col_info );
    free( col_data );
```

```
    return;
}
```

一旦描述了结果集，就可以返回结果集行。以下代码段所示的是如何返回结果集的行。它使用 SQL Anywhere C API 为先前由 C API 执行的 SQL 查询读取行。由 SQL Anywhere C API 返回的行会送回调用环境，一次发回一行。返回各行之前，必须先填充列数据值结构的数组。列数据值结构包括列索引、指向数据值的指针、数据长度和附加标志。

```
a_sqlany_data_value *value = (a_sqlany_data_value *)
    malloc( columns * sizeof(a_sqlany_data_value) );

while( capi.sqlany_fetch_next( sqlany_stmt ) )
{
    for( int i = 0; i < columns; i++ )
    {
        if( capi.sqlany_get_column( sqlany_stmt, i, &value[i] ) )
        {
            col_data[i].column_index = i + 1;
            col_data[i].column_data  = value[i].buffer;
            col_data[i].data_length  =
(a_sql_uint32)*(value[i].length);
            col_data[i].append  = 0;
            if( *(value[i].is_null) )
            {
                // Received a NULL value
                col_data[i].column_data = NULL;
            }
        }
    }
    if( extapi->set_value(  arg_handle,
                        EXTFN_RESULT_SET_ARG_NUM,
                        (an_extfn_value *)&rs_info,
                        EXTFN_RESULT_SET_NEW_ROW_FLUSH ) == 0 )
    {
        // failed
        free( value );
        free( col_data );
        free( col_data );
        extapi->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
}
```

Java 外部环境

数据库服务器支持 Java 存储过程和函数。Java 存储过程或函数与 SQL 存储过程或函数的行为基本相同，只是过程或函数的代码用 Java 编写并且在数据库服务器外（即在 Java VM 环境内）执行。

应该注意的是，每个数据库对应于一个 Java VM 实例，而不是每个连接对应于一个实例。Java 存储过程可以返回结果集。

在数据库支持中使用 Java 有几个前提条件：

1. 数据库服务器计算机中必须装有一个 Java 运行时环境副本。
2. 数据库服务器必须能够找到 Java 可执行文件 (Java VM)。

要在数据库中使用 Java，请确保数据库服务器能够找到并启动 Java 可执行文件。通过执行以下语句可验证这一点：

```
START EXTERNAL ENVIRONMENT JAVA;
```

如果数据库服务器未能启动 Java，问题的原因可能是数据库服务器不能找到 Java 可执行文件。这种情况下应执行 ALTER EXTERNAL ENVIRONMENT 语句，以特意设置 Java 可执行文件的位置。务必要包含可执行文件名。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'java-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:\\jdk1.6.0\\jre\\bin\\java.exe';
```

通过执行以下 SQL 查询，可查询将要用于数据库服务器的 Java VM 的位置：

```
SELECT db_property('JAVAVM');
```

请注意，除非用于验证数据库服务器能否启动 Java VM，否则不必使用 **START EXTERNAL ENVIRONMENT JAVA** 语句。一般而言，调用 Java 存储过程或函数会自动启动 Java VM。

同样，停止 Java 实例也不必使用 **STOP EXTERNAL ENVIRONMENT JAVA** 语句，因为数据库的所有连接终止时 Java 实例会自动消失。但是，如果要彻底停用 Java 并且想要释放一些资源，则 **STOP EXTERNAL ENVIRONMENT JAVA** 语句可以减少 Java VM 的使用次数。

验证了数据库服务器可以启动 Java VM 可执行文件后，接下来要做的事就是在数据库中安装所需的 Java 类代码。可用 **INSTALL JAVA** 语句执行这种操作。例如，可以执行以下语句来将 Java 类从文件安装到数据库中。

```
INSTALL JAVA
NEW
FROM FILE 'java-class-file';
```

也可以将 Java JAR 文件安装到数据库中。

```
INSTALL JAVA
NEW
JAR 'jar-name'
FROM FILE 'jar-file';
```

可以从变量安装 Java 类，如下所示：

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
NEW
```

```
FROM JavaClass;
```

要从数据库中删除 Java 类，请使用 **REMOVE JAVA** 语句，如下所示：

```
REMOVE JAVA CLASS java-class
```

要从数据库中删除 Java JAR，请使用 **REMOVE JAVA** 语句，如下所示：

```
REMOVE JAVA JAR 'jar-name'
```

要修改现有 Java 类，可以使用 **INSTALL JAVA** 语句的 **UPDATE** 子句，如下所示：

```
INSTALL JAVA
UPDATE
FROM FILE 'java-class-file'
```

也可以在数据库中更新现有 Java JAR 文件。

```
INSTALL JAVA
UPDATE
JAR 'jar-name'
FROM FILE 'jar-file';
```

可以从变量更新 Java 类，如下所示：

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
UPDATE
FROM JavaClass;
```

Java 类安装在数据库中后，接下来可以创建存储过程和函数，以与 Java 方法连接。**EXTERNAL NAME** 字符串含有调用 Java 方法以及返回 OUT 参数和返回值所需的信息。必须在 **EXTERNAL NAME** 子句的 LANGUAGE 属性中指定 **JAVA**。**EXTERNAL NAME** 子句的格式为：

EXTERNAL NAME '*java-call*' **LANGUAGE JAVA**

java-call：

```
[package-name.]class-name.method-name method-signature
```

method-signature：

```
( [ field-descriptor, ... ] ) return-descriptor
```

field-descriptor 和 return-descriptor：

- Z
- | B
- | S
- | I
- | J

- | F
- | D
- | C
- | V
- | [descriptor
- | Lclass-name;

Java 方法签名是参数类型和返回值类型的压缩字符表示形式。如果参数数量少于显示在方法签名中的数量，则差值必须等于 **DYNAMIC RESULT SETS** 中指定的数量，并且方法签名中超出过程参数列表中参数的每个参数的方法签名必须是 **[Ljava/SQL/ResultSet;**。

对于 Java UDF，无需设置 **DYNAMIC RESULT SETS**；暗示 **DYNAMIC RESULT SETS** 等于 1。

field-descriptor 和 *return-descriptor* 的含义如下：

字段类型	Java 数据类型
B	byte
C	char
D	double
F	float
I	int
J	long
L class-name;	是类 class-name 的实例。类名必须是完全限定的，而且名称中的任何点都必须替换为 /。例如 java/lang/String
S	short
V	无类型
Z	布尔型
[	数组的每个维度都使用一个

例如，

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
  java.sql.ResultSet[] rs ) {
}
```

可以有以下签名：

```
'(ZILjava/math/BigDecimal;[[B[Ljava/SQL/ResultSet;)D'
```

以下过程创建 Java 方法的接口。Java 方法不返回任何值 (V)。

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed()V'
LANGUAGE JAVA;
```

以下过程创建具有字符串 ([Ljava/lang/String;) 输入参数的 Java 方法的接口。Java 方法不返回任何值 (V)。

```
CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

以下过程创建 Java 方法 `Invoice.init` （可接收一个字符串参数 (Ljava/lang/String;)、一个双精度参数 (D)、另一个字符串参数 (Ljava/lang/String;) 和另一个双精度参数 (D)）的接口，并且不返回任何值 (V)。

```
 CREATE PROCEDURE init( IN arg1 CHAR(50),
                        IN arg2 DOUBLE,
                        IN arg3 CHAR(50),
                        IN arg4 DOUBLE)
EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/
String;D)V'
LANGUAGE JAVA
```

下面的示例 Java 含有函数 main，该函数可以接收一个字符串参数并将其写入数据库服务器消息窗口。其中还含有函数 **whare**，该函数可返回 Java 字符串。

```
import java.io.*;

public class Hello
{
    public static void main( String[] args )
    {
        System.out.print( "Hello" );
               for ( int i = 0; i  < args.length; i++ )
                   System.out.print( " " + args[i] );
        System.out.println();
    }
    public static String whare()
    {
        return( "I am SQL Anywhere." );
    }
}
```

以上 Java 代码位于 `Hello.java` 文件中，并使用 Java 编译器进行编译。所生成的类文件将装载到数据库中，如下所示。

```
INSTALL JAVA
NEW
FROM FILE 'Hello.class';
```

通过 Interactive SQL，用于在 `Hello` 类中连接方法 `main` 的存储过程将按如下方式创建：

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

请注意，main 的参数描述为 `java.lang.String` 数组。用 Interactive SQL 通过执行以下 SQL 语句来测试该接口。

```
CALL HelloDemo('SQL Anywhere');
```

如果检查数据库服务器消息窗口，将会找到此处写入的消息。所有到 `System.out` 的输出将重定向到服务器消息窗口。

通过 Interactive SQL，用于在 `Hello` 类中连接方法 `whare` 的函数将按如下方式创建：

```
CREATE FUNCTION Whare()
RETURNS LONG VARCHAR
EXTERNAL NAME 'Hello.whoAreYou(V)Ljava/lang/String;'
LANGUAGE JAVA;
```

请注意，函数 `whare` 以返回 `java.lang.String` 的形式进行描述。用 Interactive SQL 通过执行以下 SQL 语句来测试该接口。

```
SELECT Whare();
```

应该会在“Interactive SQL 结果”窗口中看到响应。

尝试排除 Java 外部环境未启动的故障时，也就是调用 Java 时应用程序遇到“主线程未找到”错误时，DBA 应该执行以下检查：

- 如果 Java VM 与数据库服务器的位数不相同，则确保数据库服务器计算机中装有位数与 VM 相同的客户端库。
- 确保共享对象 `sajdbc.jar` 和 `dbjdbc12/libdbjdbc12` 来自同一软件内部版本。
- 如果数据库服务器计算机中有多个 `sajdbc.jar`，则确保其都已与同一软件版本同步。
- 如果数据库服务器计算机很忙，则可能会因为超时而报错。

另请参见

- INSTALL JAVA 语句 （第 337 页）
- CREATE PROCEDURE 语句 (Java UDF) （第 339 页）
- CREATE FUNCTION 语句 (Java UDF) （第 341 页）
- REMOVE 语句 （第 342 页）
- START JAVA 语句 （第 343 页）
- STOP JAVA 语句 （第 344 页）

Multiplex 中的 Java 外部环境

要在 multiplex 配置中使用 Java 外部环境 UDF，必须先在需要使用 UDF 的 multiplex 的每个节点上安装 Java 类文件或者 JAR 文件。

使用 Sybase Control Center，Sybase Central，或者 Interactive SQL **INSTALL JAVA** 语句以安装 Java 类文件和 JAR。

另请参见

- INSTALL JAVA 语句 （第 337 页）

通过 Sybase Central 安装类

要使 Java 类能在数据库中使用，请通过 Sybase Central 将类安装到数据库中。必须知道要安装的 类的路径和文件名。

必须有 DBA 权限才能安装类。

安装类 (Sybase Central)：

1. 以 DBA 用户身份连接到数据库。
2. 打开**外部环境**文件夹。
3. 在此文件夹下，打开 **Java** 文件夹。
4. 右键单击右窗格，然后选择**新建 » Java 类**。
5. 按照向导中的说明进行操作。

使用 Interactive SQL 安装类

若要在数据库中使用您的 Java 类，请使用 Interactive SQL 中的 INSTALL JAVA 语句在数据库中安装类。您必须知道要安装的类的路径和文件名。

1. 以 DBA 用户身份连接到数据库。
2. 执行以下语句：

```
INSTALL JAVA NEW
FROM FILE 'path\\ClassName.class';
```

`path` 是类文件所处的目录，`ClassName.class` 是类文件的名称。

双反斜线可确保斜线不被视为转义字符。

例如，要安装名为 `Utility.class` 的文件中的类（保存在 `c:\source` 目录中），请执行以下语句：

```
INSTALL JAVA NEW
FROM FILE 'c:\\source\\Utility.class';
```

如果使用相对路径，它必须相对于数据库服务器的当前工作目录。

Java 外部环境限制

请先熟悉特定于 UDF 的 Java 外部环境的限制，然后再开发 Java UDF 和 Java 表 UDF。

- 不支持集合 Java 函数。
- 不能对涉及 Java UDF 的查询片段进行 DQP 或 SMP 处理。
- 不能在 Java 外部环境中 **DROP** 当前查询涉及的表。
- 不能在 Java 外部环境中 **ALTER** 当前查询涉及的表。
- 不支持 `UNSIGNED SMALLINT` 数据类型。
- 数值函数的精度限制为不高于 255。
- 对于 Java 表 UDF，只允许有一个结果集。

另请参见

- 外部环境限制 （第 311 页）

Java VM 内存选项

使用 **java_vm_options** 选项来指定启动 Java 虚拟机 (VM) 所需的任何附加命令行选项。

使用以下语法：

```
SET OPTION PUBLIC.java_vm_options='java-options';
```

在以下示例中，使用 **java_vm_options** 将 Java VM 的最大堆尺寸设置为 512 兆字节：

```
SET OPTION PUBLIC.java_vm_options='-Xmx512m';
```

在以下示例中，将 Java VM 的初始堆大小设置为 32 兆字节：

```
SET OPTION PUBLIC.java_vm_options='-Xms32m';
```

Java UDF 的 SQL 数据类型转换

SQL 到 Java 和 Java 到 SQL 数据类型转换是按照 JDBC 标准执行的。输入值支持 LOB 数据类型 `LONG VARCHAR` 和 `LONG BINARY`，但返回值不支持。

SQL 到 Java 的数据类型转换

用于 Java 标量 UDF 和 Java 表 UDF 输入值的数据类型转换方式。

SQL 类型	Java 类型
`BIGINT`	`long`
`BINARY`	`byte[ ]`
`BIT`	`boolean`

SQL 类型	Java 类型
CHAR	String
DATE	java.sql.Date
DECIMAL	java.math.BigDecimal
DOUBLE	double
IMAGE	byte[ ]
INTEGER	int
LONG BINARY	byte[ ] **注意**：大对象数据支持需要使用单独许可的 Sybase IQ 选项。
LONG VARCHAR	String **注意**：大对象数据支持需要使用单独许可的 Sybase IQ 选项。
MONEY	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
REAL	float
SMALLINT	short
SMALLMONEY	java.math.BigDecimal
TEXT	String
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	byte
UNSIGNED BIGINT	java.math.BigDecimal (with a precision of 20 and scale of 0)
UNSIGNED INT	java long
VARBINARY	byte[ ]
VARCHAR	String

Java 到 SQL 的数据类型转换

Java 标量 UDF 和 Java 表 UDF 的返回值数据类型。

Java 类型	SQL 类型
String	CHAR
String	VARCHAR
String	TEXT
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	MONEY
java.math.BigDecimal	SMALLMONEY
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[ ]	VARBINARY
byte[ ]	IMAGE
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	DATETIME/TIMESTAMP
java.lang.Double	DOUBLE
java.lang.Float	REAL
java.lang.Integer	INTEGER
java.lang.Long	BIGINT

创建 Java 标量 UDF

创建并编译 Java 类，将类文件安装至服务器，然后创建函数定义。

前提条件

- 您熟悉 Java 编程语言，并且能够编译 `.java` 文件。您了解生成的 `.class` 文件在文件系统中的位置。
- 您熟悉 Interactive SQL。您可以通过 Interactive SQL 连接至 `iqdemo` 数据库，并且可以通过 Interactive SQL 发出 **START EXTERNAL ENVIRONMENT JAVA** 指令。

过程

当创建您自己的 Java UDF 时，请使用该任务作为模板。

1. 请将此段 Java 代码放置于名为 `HelloJavaUDF.java` 的文件中：

```
public class HelloJavaUDF
{
    public static String helloJava( String name )
    {
        // Simply return Hello and the name passed in.
        return "Hello " + name;
    }
}
```

   该段代码使用静态方法 `helloJava` 创建 `HelloJavaUDF` Java 类。该方法采用单一字符串参数并返回字符串。

2. 编译 `HellowJavaUDF.java`：
   `javac <pathtojavafile>/HelloJavaUDF.java`
3. 在 Interactive SQL 中，连接至 `iqdemo` 数据库。
4. 在 Interactive SQL 中，将类文件安装至服务器：

使用绝对路径名	`INSTALL JAVA NEW FROM FILE '<absolutepathtofile>/HelloJavaUDF.class'`
	示例：
	`INSTALL JAVA NEW FROM FILE 'd:/mydirectory/HelloJavaUDF.class'`

使用相对路径名	`INSTALL JAVA NEW FROM FILE '<pathrelativetocwd>/HelloJavaUDF.class'`
	示例：
	`INSTALL JAVA NEW FROM FILE 'myreldir/HelloJavaUDF.class'`

5. 在 Interactive SQL 中，创建函数定义。
   请提供以下信息：
   - Java 软件包、类和方法的名称
   - 您函数参数的 Java 数据类型
   - 分配给 Java UDF 的 SQL 名称

```
CREATE FUNCTION my_helloJava(IN name VARCHAR(249) )
RETURNS VARCHAR(255)
EXTERNAL NAME 'example.HelloJavaUDF.helloJava(Ljava/lang/
String;)Ljava/lang/String;'
LANGUAGE JAVA
```

6. 在 Interactive SQL 中，在对 `iqdemo` 数据库的查询中使用 Java UDF 。

```
SELECT my_helloJava( GivenName ) FROM Customers WHERE ID <
110
```

另请参见

- SQL 到 Java 的数据类型转换 （第 329 页）
- Java 到 SQL 的数据类型转换 （第 331 页）

创建 SQL substr 函数的 Java 标量 UDF

创建 Java UDF 配置，其中 SQL 函数向 Java UDF 传递多个参数。

前提条件

- 您熟悉 Java 编程语言，并且能够编译 `.java` 文件。您了解生成的 `.class` 文件在文件系统中的位置。
- 您熟悉 Interactive SQL。您可以通过 Interactive SQL 连接至 `iqdemo` 数据库，并且可以通过 Interactive SQL 发出 **START EXTERNAL ENVIRONMENT JAVA** 指令。

过程

1. 请将此段 Java 代码放置于名为 `MyJavaSubstr` 的文件中：

```
public class MyJavaSubstr
{
   public static String my_java_substr( String in, int start, int
length )
    {
    String rc = null;

    if ( start < 1 )
    {
        start = 1;
    }

    // Convert the SQL start, length to Java start, end.
    start --;  // Java is 0 based, but SQL is one based.
    int endindex = start+length;
```

```
    try {
        if ( in != null )
        {
            rc = in.substring( start, endindex );
        }
    } catch ( IndexOutOfBoundsException ex )
    {
        System.out.println("ScalarTestFunctions:
                my_java_substr("+in+","+start+","+length+")
failed");
        System.out.println(ex);
    }
    return rc;
 }
```

2. 在 Interactive SQL 中，连接至 `iqdemo` 数据库。
3. 在 Interactive SQL 中，将类文件安装至服务器：
   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   MyJavaSubstr.class'
   ```
4. 在 Interactive SQL 中，创建函数定义：
   ```
   CREATE or REPLACE FUNCTION java_substr(IN a VARCHAR(255), IN b
   INT, IN c INT )
     RETURNS VARCHAR(255)
     EXTERNAL NAME
       'example.MyJavaSubstr.my_java_substr(Ljava/lang/
   String;II)Ljava/lang/String;'
     LANGUAGE JAVA
   ```
   注意：**Ljava/lang/String;II** 代码片段表示 `String, int, int` 参数类型。
5. 在 Interactive SQL 中，在对 `iqdemo` 数据库的查询中使用 Java UDF 。
   select GivenName, java_substr(Surname,1,1) from Customers where lcase(java_substr(Surname,1,1)) = 'a';

创建 Java 表 UDF

创建、编译、安装 Java 行生成器，并创建 Java 表 UDF 函数定义。

前提条件

- 您熟悉 Java 编程语言，并且能够编译 `.java` 文件。您了解生成的 `.class` 文件在文件系统中的位置。
- 您熟悉 Interactive SQL。您可以通过 Interactive SQL 连接至 `iqdemo` 数据库，并且可以通过 Interactive SQL 发出 **START EXTERNAL ENVIRONMENT JAVA** 指令。

过程

本示例执行 Java 行生成器 (RowGenerator) ，该行生成器采用单个整数输入并在结果集中返回行数。结果集包含两列：其一为 `INTEGER`，另一个为 `VARCHAR`。RowGenerator 依赖于两个实用类：

- example.ResultSetImpl
- example.ResultSetMetaDataImpl

这是 java.sql.ResultSet 接口和 java.sql.ResultSetMetaData 接口的简单实现方式。

1. 请将此段代码放置于名为 RowGenerator.java 的文件中：

```
package example;

import java.sql.*;

public class RowGenerator {

  public static void rowGenerator( int numRows, ResultSet rset[] ) {
    // Create the meta data needed for the result set
    ResultSetMetaDataImpl rsmd = new ResultSetMetaDataImpl(2);

    //The first column is the SQL type INTEGER.
    rsmd.setColumnType(1, Types.INTEGER);
    rsmd.setColumnName(1,"c1");
    rsmd.setColumnLabel(1,"c1");
    rsmd.setTableName(1,"MyTable");

    // The second column is the SQL type VARCHAR length 255
    rsmd.setColumnType(2, Types.VARCHAR);
    rsmd.setColumnName(2,"c2");
    rsmd.setColumnLabel(2,"c2");
    rsmd.setColumnDisplaySize(2, 255);
    rsmd.setTableName(2,"MyTable");

    // Create result set using the ResultSetMetaData
    ResultSetImpl rs = null;
    try {
      rs = new ResultSetImpl( (ResultSetMetaData)rsmd );
      rs.beforeFirst();  // Make sure we are at the beginning.
    } catch( Exception e ) {
      System.out.println( "Error: couldn't create result set." );
      System.out.println( e.toString() );
    }

    // Add the rows to the result set and populate them
    for( int i = 0; i < numRows; i++ ) {
      try {
        rs.insertRow();   // insert a new row.
       rs.updateInt( 1, i );   // put the integer value in the first column
       rs.updateString( 2, ("Str" + i) );  // put the VARCHAR/String value
in the second column
      } catch( Exception e ) {
        System.out.println( "Error: couldn't insert row/data on row " +
i );
        System.out.println( e.toString() );
      }
    }

    try {
```

```
      rs.beforeFirst();  // rewind the result set so that the server gets
it from the beginning.
    } catch( Exception e ) {
        System.out.println( e.toString() );
    }
    rset[0] = rs;  // assign the result set to the 1st of the passed in
array.
  }
}
```

2. 编译 `RowGenerator.java`、`ResultSetImpl.java` 和 `ResultSetMetaData.java`。Windows 目录 `%ALLUSERSPROFILE%\samples\java`（UNIX 上为 `$IQDIR15/samples/java`）包含 `ResultSetImpl.java` 和 `ResultSetMetaData.java`。

   ```
   javac <pathtojavafile>/ResultSetMetaDataImpl.java
   ```

   ```
   javac <pathtojavafile>/ResultSetImpl.java
   ```

   ```
   javac <pathtojavafile>/RowGenerator.java
   ```

3. 在 Interactive SQL 中，连接至 `iqdemo` 数据库。
4. 在 Interactive SQL 中，安装如下三个类文件：

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetMataDataImpl.class'
   ```

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetImpl.class'
   ```

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   RowGenerator.class'
   ```

5. 在 Interactive SQL 中，创建 Java 表函数定义。
   准备提供以下信息：
   - Java 软件包、类和方法的名称
   - 函数参数的 Java 数据类型
   - 分配给 Java UDF 的 SQL 名称

   ```
   CREATE or REPLACE PROCEDURE rowgenerator( IN numRows INTEGER )
       RESULT ( c1 INTEGER , c2 VARCHAR(255) )
       EXTERNAL NAME
           'example.RowGenerator.rowGenerator(I[Ljava/sql/
   ResultSet;)V'
       LANGUAGE JAVA
   ```

   注意： RESULT 集包含两列；其一为 `INTEGER`，另一个为 `VARCHAR(255)`。Java 原型有两个参数；其一为 INT (I)，另一个为 `java.sql.ResultSets ([Ljava/sql/ResultSet;)` 数组。Java 原型显示函数将返回 `Void(V)`。

6. 在 Interactive SQL 中，在对 `iqdemo` 数据库的查询中使用 Java 表 UDF 。

   ```
   SELECT * from rowGenerator(5);
   ```

该查询返回五行两列数据。

另请参见

- SQL 到 Java 的数据类型转换 （第 329 页）
- Java 到 SQL 的数据类型转换 （第 331 页）

Java 外部环境 SQL 语句参考

当开发 Java 存储过程和函数时，请使用这些 SQL 语句。

INSTALL JAVA 语句

使 Java 类可供在数据库中使用。

语法

```
INSTALL JAVA [ install-mode ] [ JAR jar-name ] FROM source
```

参数

- **install-mode：** – { **NEW** | **UPDATE** }
- **source：** – { **FILE** *filename* | **URL** *url-value* }

示例

- **示例 1** – 通过提供文件名和类的位置，安装用户创建的名为“Demo”的 Java 类：

  ```
  INSTALL JAVA NEW
  FROM FILE 'D:\JavaClass\Demo.class'
  ```

  安装后，使用类的名称引用它，不再使用它的原始文件路径位置。例如，下面的语句使用上一语句中安装的类：

  ```
  CREATE VARIABLE d Demo
  ```

  如果 Demo 类是软件包 sybase.work 的成员，必须使用该类的完全限定名：

  ```
  CREATE VARIABLE d sybase.work.Demo
  ```

- **示例 2** – 安装 zip 文件中包含的所有类，并将数据库中的这些类与 JAR 文件名关联：

  ```
  INSTALL JAVA
  JAR 'Widgets'
  FROM FILE 'C:\Jars\Widget.zip'
  ```

  不保留 zip 文件的位置，并且必须使用完全限定的类名（包名和类名）引用类。

用法

- **安装模式** – 如果指定 **NEW** 安装模式，则引用的 Java 类必须是新类，而不是目前已安装好的类的更新。如果数据库中存在同名的 类而且使用 **NEW** 安装模式，则会出现 错误。

  UPDATE 安装模式用于指定引用的 Java 类可以包含 已安装在给定数据库中的 Java 类的 替代类。

  要更新类或 JAR，您必须具有 DBA 权限，并且磁盘上的某个文件中要有较新版本的已编译的类文件或 JAR 文件。

  只有在安装类之后建立的新连接或者在安装类之后首次使用类的新连接才使用新定义。一旦 Java VM 装载了某个类定义，就会将其一直保留在内存中，直到连接关闭为止。

  如果已使用基于当前连接中的某个类的 Java 类或对象，则必须断开连接并重新连接，才能使用新的类定义。

  如果省略安装模式，则缺省安装模式为 **NEW**。
- **JAR** – JAR — 指定 *file-name* 或 *text-pointer* 必须 指定 JAR 文件或包含 JAR 的列。JAR 文件通常 带扩展名 `.jar` 或 `.zip`。

  可对已安装好的 JAR 和 zip 文件进行压缩或解压缩。但是，通过 Sun JDK **jar** 实用程序 生成的 JAR 文件不受支持。但支持由其他 zip 实用程序生成的文件。

  如果指定了 JAR 选项，则安装 了 JAR 包含的类后，会以 JAR 的形式保留 JAR。该 JAR 是与这些类的每一个关联的 JAR。用 **JAR** 选项安装在数据库中的 JAR 称为数据库的保留 JAR。

  保留下来的 JAR 会在 **INSTALL** 和 **REMOVE** 语句中引用。保留下来的 JAR 不影响其他 Java-SQL 类应用。SQL 系统用保留下来的 JAR 处理其他系统对 与给定数据相关联的类的请求。如果所请求的类 具有关联 JAR，则 SQL 系统可以直接提供该 JAR，而不是单个类。

  jar-name 是一个字符串值，最长 255 个字节。*jar-name* 用于在 后续 **INSTALL**、**UPDATE** 和 **REMOVE** 语句中标识保留下来的 JAR。
- **源代码** – 用于指定 Java 类的安装位置。

  对于 *file-name*，受支持的格式包括 完全限定文件名（如“`c:\libs \jarname.jar'`”和 “`/usr/u/libs/jarname.jar`”）和相对文件名 （相对于数据库服务器的当前工作目录）。

  filename 必须标识类文件 或者 JAR 文件。

  每个类的类定义都是在首次使用该类时由每个连接的 VM 载入的。当您 **INSTALL** 类时，将隐式重新启动连接上的 VM。因此，无论 **INSTALL** 的 *install-mode* 是 **NEW** 还是 **UPDATE**，都可以直接访问新类。

对于其他连接，VM 下次第一次访问新类时会装入新类。如果新类已由 VM 装入，则为该连接重新启动 VM（例如，用 **STOP JAVA** 和 **START JAVA** 重新启动）后，才能用该连接看到新类。

标准

- SQL - ISO/ANSI SQL 语法的供应商扩展。
- Sybase - 不受 Adaptive Server Enterprise 支持。

权限

- 执行 **INSTALL** 语句需要 DBA 权限。
- 任何用户可以任何方式引用所有已安装的类。

CREATE PROCEDURE 语句 (Java UDF)

创建外部 Java 表 UDF 的接口。

外部过程的 **CREATE PROCEDURE** 语句参考信息位于单独的主题中。表 UDF 的 **CREATE PROCEDURE** 语句参考信息位于单独的主题中。

如果您的查询引用 Sybase IQ 表，请注意相对于仅引用 Catalog 存储表的查询，该查询要应用不同的语法和参数。

仅在 **FROM** 子句中支持 Java 表 UDF。

语法

对于至少引用一个 Sybase IQ 表的查询：

```
CREATE [ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter, …] )
[ RESULT (result-column, ...)]
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'java-call' [ LANGUAGE Java ] }
```

对于仅引用 Catalog 存储表的查询：

```
CREATE [ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter, …] )
[ RESULT (result-column, ...)]
| NO RESULT SET
[ DYNAMIC RESULT SETS integer-expression ]
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'java-call' [ LANGUAGE Java ] }
```

参数

- **参数** – 对于至少引用一个 Sybase IQ 表的查询：

  [**IN***parameter_modeparameter-namedata-type* [**DEFAULT***expression*]

  对于仅引用 Catalog 存储表的查询：

[**IN** | **OUT** | **INOUT**] *parameter_mode**parameter-name**data-type* [**DEFAULT***expression*]

- **result-column** – column-name data-type
- **JAVA** – **[ALLOW | DISALLOW SERVER SIDE REQUESTS]**
- **java-call** – **'[package-name.]class-name.method-name method-signature'**

用法

对于 Java 表函数，只允许一个结果集。如果 Java 表函数使用 Sybase IQ 表连接或 Sybase IQ 表中某个列是 Java 表函数的 参数，则仅支持一个结果集。

如果 Java 表函数是 **FROM** 子句中唯一的项目， 则允许 *N* 个结果集。

JAVA: [ALLOW | DISALLOW SERVER SIDE REQUESTS]:

DISALLOW 为缺省值。

ALLOW 指示允许服务器端连接。

注意: 除非有必要，否则不要指定 **ALLOW**。 **ALLOW** 的设置会减慢特定类型的 Sybase IQ 表连接。 如果将过程定义从 **ALLOW** 改为 **DISALLOW**（反之亦然），则直到建立新的连接后，才会确认更改。

不要在同一查询中将 UDF 与 **ALLOW SERVER SIDE REQUESTS** 和 **DISALLOW SERVER SIDE REQUESTS** 同时使用。

标准

- SQL—符合 ISO/ANSI SQL 标准。
- Sybase—Transact-SQL **CREATE PROCEDURE** 语句与此不同。
- SQLJ—提议的 SQLJ1 标准中指定了 Java 结果集的语法扩展。

权限

必须具有 RESOURCE 权限， 否则只能创建临时过程。 具有 DBA 权限的用户可以通过指定 owner 为其他用户创建 UDF。 用户必须具有 DBA 权限才能创建 外部 UDF，或者为另一用户创建外部 UDF。

在过程中引用临时表

如果临时表定义不一致，则在过程之间共享该表会导致出现问题。

例如，假设有两个过程 `procA` 和 `procB`，这两个过程均定义临时表 `temp_table`，并调用名为 `sharedProc` 的另一过程。即未调用 `procA`，也未调用 `procB`，所以临时表还不存在。

现在假设 `temp_table` 在 `procA` 中的定义与在 `procB` 中的定义略有不同，而两个过程均使用相同的列名称和类型，但列顺序不同。

调用 `procA` 时，它返回预期结果。但调用 `procB` 时，它返回不同的结果。

这是因为，调用 procA 时，它创建 temp_table，然后调用 sharedProc。调用 sharedProc 时，会解析并验证其中的 **SELECT** 语句，然后缓存该语句解析后的表示形式，这样执行另一 **SELECT** 语句时就可以再次使用该语句。缓存的形式可反映 procA 中表定义的列顺序。

调用 procB 时将重新创建 temp_table，但列顺序不同。procB 调用 sharedProc 时，数据库服务器使用 **SELECT** 语句的缓存表示形式。因此，结果不同。

可通过执行以下某项操作避免出现这种问题：

- 确保以此方式使用的临时表定义一致
- 考虑改用全局临时表

CREATE FUNCTION 语句 (Java UDF)

用于在数据库中新建外部 Java 表 UDF 函数。

语法

```
CREATE [ OR REPLACE | TEMPORARY ] FUNCTION [ owner.]function-name
( [ parameter, ··· ] )
 RETURNS data-type routine-characteristics
 [ SQL SECURITY { INVOKER | DEFINER } ]
 { compound-statement
  | AS tsql-compound-statement
  | EXTERNAL NAME 'java-call' LANGUAGE JAVA }
```

参数

- **参数** – **IN***parameter-name**data-type* [**DEFAULT***expression*]
- **routine-characteristics** – **ONEXCEPTIONRESUME** | [**NOT**] **DETERMINISTIC**
- **tsql-compound-statement** – *sql-statement sql-statement* ···
- **environment-name** – **JAVA [ALLOW | DISALLOW SERVER SIDE REQUESTS]**
- **java-call** – '[*package-name*.]*class-name.method-name**method-signature*'
- **method-signature** – ([*field-descriptor*, ···.]) *return-descriptor*
- **field-descriptor 和 return-descriptor** – **Z** | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[** *descriptor* | **L** *class-name***;**

示例

- **示例 1** – 创建一个用 Java 编写的外部函数：

```
CREATE FUNCTION dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

用法

不许将 LONG BINARY 和 LONG VARCHAR 用作 return-value 数据类型。

EXTERNAL NAME LANGUAGE JAVA — 是将 **EXTERNAL NAME** 与 **LANGUAGE JAVA** 子句一起使用的函数，也是包含 Java 方法的包装。

iq-environment-name: JAVA [ALLOW/ *DISALLOW SERVER SIDE REQUESTS*]:

DISALLOW 为缺省值。

ALLOW 表示允许建立服务器端连接。

注意： 除非有必要，否则不要指定 **ALLOW**。设置 **ALLOW** 会减慢某些类型的 Sybase IQ 表的连接。

不要在同一查询中同使用 UDF 与 **ALLOW SERVER SIDE REQUESTS** 和 **DISALLOW SERVER SIDE REQUESTS**。

标准

- SQL — 符合 ISO/ANSI SQL 标准。
- Sybase — 不受 Adaptive Server Enterprise 支持。

权限

必须具有 RESOURCE 权限。

外部 Java 函数必须有 DBA 权限。

REMOVE 语句

从数据库删除类、软件包或 JAR 文件。删除的类不再可供用作变量类型。

语法

```
REMOVE JAVA classes_to_remove
```

参数

- **classes_to_remove：** – { **CLASS***java_class_name* [, *java_class_name*]… | **PACKAGE***java_package_name* [, *java_package_name*]… | **JAR***jar_name* [, *jar_name*]… [**RETAIN CLASSES**] }
- **jar_name：** – *character_string_expression*

示例

- **示例 1** – 从当前数据库中删除名为“Demo”的 Java 类：

```
REMOVE JAVA CLASS Demo
```

用法

任何要删除的类、软件包或 JAR 必须已安装。

java_class_name—要删除的一个或多个 Java 类的名称。这些类必须是当前数据库中已安装的类。

java_package_name—要删除的一个或多个 Java 软件包的名称。这些软件包的名称必须为当前数据库中的软件包的名称。

jar_name—最大长度为 255 的字符串值。

每个 *jar_name* 必须与当前数据库中保留的 JAR 的 *jar_name* 相等。*jar_name* 的等同性由 SQL 系统的字符串比较规则确定。

如果指定了 **JAR...RETAIN CLASSES**，则指定的 JAR 将不再保留在数据库中，并且保留的类没有与之关联的 JAR。如果指定了 **RETAIN CLASSES** ，则这是 **REMOVE** 语句的唯一操作。

标准

- SQL—ISO/ANSI SQL 语法的供应商扩展。
- Sybase—不受 Adaptive Server Enterprise 支持。通过使用嵌套事务，可以采用 Adaptive Server Enterprise 兼容的方式实现类似的功能。

权限

必须拥有 DBA 权限或必须拥有相应对象。

START JAVA 语句

启动 Java VM。

语法

```
START EXTERNAL ENVIRONMENT JAVA
```

示例

- **示例 1** – 启动 Java VM。

  ```
  START EXTERNAL ENVIRONMENT JAVA
  ```

用法

可在方便的时候使用 **START EXTERNAL ENVIRONMENT JAVA** 装载 Java VM，这样如果用户开始使用 Java 功能，则装载 Java VM 时一开始不会出现暂停。

标准

- SQL—ISO/ANSI SQL 语法的供应商扩展。
- Sybase—不适用。

权限

必须具有 DBA 权限。

STOP JAVA 语句

用于释放与 Java VM 相关联的资源。

语法

```
STOP EXTERNAL ENVIRONMENT JAVA
```

用法

STOP EXTERNAL ENVIRONMENT JAVA 的主要用途是减少 系统资源用量。

标准

- SQL—ISO/ANSI SQL 语法的供应商扩展。
- Sybase—不适用。

权限

DBA 授权

PERL 外部环境

Perl 存储过程或函数与 SQL 存储过程或函数的行为基本相同，只是过程或函数的代码用 Perl 编写，并且在数据库服务器外（即在 Perl 可执行文件实例内）执行。

值得注意的是，对于使用 Perl 存储过程和函数的每个连接，会有一个单独的 Perl 可执行文件实例。这一点不同于 Java 存储过程和函数。对于 Java，每个数据库对应于一个 Java VM 实例，而不是每个连接对应于一个实例。Perl 和 Java 间的另一个主要差异是，Perl 存储过程不返回结果集，而 Java 存储过程可以返回结果集。

在数据库支持中使用 Perl 有几个前提条件：

1. 数据库服务器计算机中必须装有 Perl，而且数据库服务器必须能够找到 Perl 可执行文件。
1. 数据库服务器计算机中必须装有 DBD::SQLAnywhere 驱动程序。

2. Windows 中也必须装有 Microsoft Visual Studio。它对于安装 DBD::SQLAnywhere 驱动程序是必要的，因此这是一个前提条件。

除了以上前提条件外，数据库管理员还必须安装 Perl External Environment 模块。

安装外部环境模块 (Windows)：

- 在 `SDK\PerlEnv` 子目录中执行以下命令：

```
perl Makefile.PL
nmake
nmake install
```

安装外部环境模块 (UNIX)：

- 在 `sdk/perlenv` 子目录中执行以下命令：

```
perl Makefile.PL
make
make install
```

构建和安装 Perl 外部环境模块后，数据库支持中的 Perl 即可使用。

要在数据库中使用 Perl，需确保数据库服务器能够找到并启动 Perl 可执行文件。通过执行以下语句可验证这一点：

```
START EXTERNAL ENVIRONMENT PERL;
```

如果数据库服务器未能启动 Perl，导致问题的原因可能是数据库服务器无法找到 Perl 可执行文件。这种情况下应执行 **ALTER EXTERNAL ENVIRONMENT** 语句，以特意设置 Perl 可执行文件的位置。务必要包含可执行文件名。

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:\\Perl\\bin\\perl.exe';
```

请注意，除非用于验证数据库服务器能否启动 Perl，否则不必使用 **START EXTERNAL ENVIRONMENT PERL** 语句。通常，进行 Perl 存储过程或函数的调用会自动启动 Perl。

同样，停止 Perl 也不必使用 **STOP EXTERNAL ENVIRONMENT PERL** 语句，因为连接终止时实例会自动消失。但是，如果要彻底停用 Perl 并且想要释放一些资源，则 **STOP EXTERNAL ENVIRONMENT PERL** 语句可以为您的连接释放该 Perl 实例。

验证数据库服务器可以启动 Perl 可执行文件后，要做的下一件事就是在数据库中安装所需的 Perl 代码。可用 **INSTALL** 语句执行这种操作。例如，可以执行以下语句来将 Perl 脚本从文件安装到数据库。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

也可以从表达式构建和安装 Perl 代码，如下所示：

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

还可以从变量构建和安装 Perl 代码，如下所示：

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

要从数据库中删除 Perl 代码，请使用 **REMOVE** 语句，如下所示：

```
REMOVE EXTERNAL OBJECT 'perl-script'
```

要修改现有 Perl 代码，可以使用 **INSTALL EXTERNAL OBJECT** 语句的 **UPDATE** 子句，如下所示：

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL
 INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL
 SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE PerlVariable
  ENVIRONMENT PERL
```

Perl 代码安装在数据库中后，接下来可以创建所需的 Perl 存储过程和函数。创建 Perl 存储过程和函数时，LANGUAGE 始终是 PERL，EXTERNAL NAME 字符串包含调用 Perl 子例程和返回 OUT 参数及返回值所需的信息。每次调用时 Perl 代码可使用以下全局变量：

- **$sa_perl_return** – 用于设置函数调用的返回值。
- **$sa_perl_argN** – 其中 N 是正整数 [0 .. n]。用于将 SQL 参数传递给 Perl 代码。例如，`$sa_perl_arg0` 是指参数 0，`$sa_perl_arg1` 是指参数 1，以此类推。
- **$sa_perl_default_connection** – 用于进行服务器端的 Perl 调用。
- **$sa_output_handle** – 用于将 Perl 代码的输出发送给数据库服务器消息窗口。

创建 Perl 存储过程时，其输入和输出参数以及返回值可以采用任何一组数据类型。但是，在进行 Perl 调用时，所有非二进制数据类型都会映射到字符串，而二进制数据会映射到数值数组。下面是一个简单的 Perl 示例：

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
```

```
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
            $_[3] );
  }'
  ENVIRONMENT PERL;

CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;

// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

下面的 Perl 示例使用一个字符串参数并将其写入数据库服务器消息窗口：

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle
$_[0]; }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;

// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

要使用服务器端 Perl，Perl 代码必须使用 *$sa_perl_default_connection* 变量。下面的示例将创建一个表，然后调用 Perl 存储过程来填充该表：

```
CREATE TABLE perlTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM
SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;
```

```
CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;

CALL PerlPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

PHP 外部环境

PHP 存储过程或函数与 SQL 存储过程或函数的行为基本相同，只是过程或函数的代码用 PHP 编写，并且在数据库服务器外（即在 PHP 可执行文件实例内）执行。

对于使用 PHP 存储过程和函数的每个连接，会有一个单独的 PHP 可执行文件实例。这一点与 Java 存储过程和函数有很大不同。对于 Java，每个数据库对应于一个 Java VM 实例，而不是每个连接对应于一个实例。PHP 和 Java 间的另一个主要差异是 PHP 存储过程不返回结果集，而 Java 存储过程可以返回结果集。PHP 仅返回 LONG VARCHAR 类型的对象，它是 PHP 脚本的输出。

在数据库支持中使用 PHP 有两个前提条件：

1. 数据库服务器计算机中必须装有 PHP 副本，而且数据库服务器必须能够找到 PHP 可执行文件。
2. 数据库服务器计算机中必须装有 PHP 扩展。

除了以上两个前提条件外，数据库管理员还必须安装 PHP 外部环境模块。包括多个 PHP 版本的预建模块。要安装预建模块，请将相应的驱动程序模块复制到 PHP 扩展目录中（可以在 `php.ini` 中找到）。在 UNIX 中，还可以使用符号链接。

安装外部环境模块 (Windows)：

1. 找到 PHP 安装目录中的 `php.ini` 文件，然后在文本编辑器中将其打开。找到用于指定 `extension_dir` 目录位置的行。如果未用任何特定目录设置 `extension_dir`，则最好将其设置为指向已隔离的目录，以提高系统安全性。
2. 将所需外部环境 PHP 模块从安装目录复制到 PHP 扩展目录中。请更改 x.y 以反映所选版本。

   ```
   copy "%SQLANY12%\Bin32\php-5.x.y_sqlanywhere_extenv12.dll"
     php-dir\ext
   ```
3. 将以下行添加到 `php.ini` 文件的动态扩展部分，以自动装载外部环境 PHP 模块。请更改 x.y 以反映所选版本。

   ```
   extension=php-5.x.y_sqlanywhere_extenv12.dll
   ```

   保存并关闭 `php.ini`。

4. 请确保您同时还将 PHP 驱动程序从安装目录安装到了 PHP 扩展目录中。此文件名遵循模式 `php-5.x.y_sqlanywhere.dll`，其中 x 和 y 为版本号。它们应该与在步骤 2 中所复制的文件的版本号相匹配。

安装外部环境模块 (UNIX)：

1. 找到 PHP 安装目录中的 `php.ini` 文件，然后在文本编辑器中将其打开。找到用于指定 `extension_dir` 目录位置的行。如果未用任何特定目录设置 `extension_dir`，则最好将其设置为指向已隔离的目录，以提高系统安全性。
2. 将所需的外部环境 PHP 模块从安装目录复制到 PHP 安装目录中。请更改 x.y 以反映所选版本。

   ```
   cp $SQLANY12/bin32/php-5.x.y_sqlanywhere_extenv12.so
     php-dir/ext
   ```
3. 将以下行添加到 `php.ini` 文件的动态扩展部分，以自动装载外部环境 PHP 模块。请更改 x.y 以反映所选版本。

   ```
   extension=php-5.x.y_sqlanywhere_extenv12.so
   ```

   保存并关闭 `php.ini`。
4. 请确保您同时还将 PHP 驱动程序从安装目录安装到了 PHP 扩展目录中。此文件名遵循模式 `php-5.x.y_sqlanywhere.so`，其中 x 和 y 为版本号。它们应该与在步骤 2 中所复制的文件的版本号相匹配。

要在数据库中使用 PHP，数据库服务器必须能够找到并启动 PHP 可执行文件。通过执行以下语句可以验证数据库服务器是否能够找到并启动 PHP 可执行文件：

```
START EXTERNAL ENVIRONMENT PHP;
```

如果显示一条消息，说明找不到“外部可执行文件”，则问题是数据库服务器不能找到 PHP 可执行文件。这种情况下应执行 **ALTER EXTERNAL ENVIRONMENT** 语句，以特意设置 PHP 可执行文件的位置（包括可执行文件名），或者确保含有 PHP 可执行文件的目录包含在 PATH 环境变量中。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

例如：

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:\\php\\php-5.2.6-win32\\php.exe';
```

要恢复缺省设置，请执行以下语句：

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php';
```

除非用于验证数据库服务器能否启动 PHP，否则不必使用 **START EXTERNAL ENVIRONMENT PHP** 语句。通常，进行 PHP 存储过程或函数的调用会自动启动 PHP。

同样，停止 PHP 也不必使用 **STOP EXTERNAL ENVIRONMENT PHP** 语句，因为连接终止时实例会自动消失。但是，如果要彻底停用 PHP 并且想要释放一些资源，则 **STOP EXTERNAL ENVIRONMENT PHP** 语句可以为您的连接释放该 PHP 实例。

验证数据库服务器可以启动 PHP 可执行文件后，要做的下一件事就是在数据库中安装所需的 PHP 代码。可用 **INSTALL** 语句执行这种操作。例如，可以执行以下语句来将特定 PHP 脚本安装到数据库。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

也可以从表达式构建和安装 PHP 代码，如下所示：

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

还可以从变量构建和安装 PHP 代码，如下所示：

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

要从数据库中删除 PHP 代码，请使用 **REMOVE** 语句，如下所示：

```
REMOVE EXTERNAL OBJECT 'php-script';
```

要修改现有 PHP 代码，可以使用 **INSTALL** 语句的 **UPDATE** 子句，如下所示：

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
 INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
 SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP 代码安装在数据库中后，接下来可以继续创建所需的 PHP 存储过程和函数。创建 PHP 存储过程和函数时，LANGUAGE 始终是 PHP，EXTERNAL NAME 字符串包含调用 PHP 子例程和返回 OUT 参数所需的信息。

参数通过 $argv 数组传递给 PHP 脚本，这与 PHP 从命令行获取参数的方式类似（即 $argv[1] 为第一个参数）。要设置输出参数，请将其赋值给相应的 $argv 元素。返回值始终是脚本的输出（LONG VARCHAR 数据类型）。

对于输入或输出参数，可用任何一组数据类型创建 PHP 存储过程。但为了在 PHP 脚本内部使用，这些参数会转换为（或者反向转换）布尔值、整数、双精度值或字符串。返回值始终是 LONG VARCHAR 类型的对象。以下是一个简单的 PHP 示例：

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<?php function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
  ENVIRONMENT PHP;

CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;

// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

对于 PHP，**EXTERNAL NAME** 字符串以 SQL 的单独一行指定。

要使用服务器端 PHP，PHP 代码可以使用缺省数据库连接。要获取数据库连接的句柄，请以空字符串参数（" 或 ""）调用 sasql_pconnect。空字符串参数会告知 PHP 驱动程序返回当前外部环境连接，而不是打开一个新连接。下面的示例将创建一个表，然后调用 PHP 存储过程来填充该表。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;

CALL PHPPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

对于 PHP，EXTERNAL NAME 字符串以 SQL 的单独一行指定。请注意，在上面的示例中，由于引号在 SQL 中的分析方式，单引号都是双写的。如果 PHP 源代码是在文件中，单引号则不用双写。

要将错误返回给数据库服务器，可抛出一个 PHP 异常。下面举例说明如何实现这一目的。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
  EXTERNAL NAME
    '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;

CALL PHPPopulateTable();
```

上例终止时会出现错误 SQLE_UNHANDLED_EXTENV_EXCEPTION，从而表明无法找到表 phpTabNoExist。

索引

符号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z