DB2 IBM Informix

IBM

版本 10.0/8.5

IBM Informix 数据库设计和实现指南

G151-0116-00

DB2 IBM Informix

IBM

版本 10.0/8.5

IBM Informix 数据库设计和实现指南

G151-0116-00

注意！

在使用本资料及其支持的产品之前，请阅读第 B-1 页的『声明』中的信息。

第一版（2004 年 12 月）

目录

第 1 部分 数据库设计和实现基础

第 2 部分 管理数据库

第 3 部分 对象关系数据库

第 4 部分 维数据库

第 5 部分 附录

介绍

本介绍内容

本介绍提供本手册中的信息的概述并描述本手册使用的约定。

有关本手册

本手册提供可帮助您设计、实现和管理 Informix 数据库的信息。本手册提供了用于说明进行数据库设计的不同方法的数据模型并阐述了如何使用结构化查询语言（SQL）来实现和管理数据库。

本手册是讨论 SQL 的 Informix 实现的若干本手册中的一本。*《IBM Informix: SQL 教程指南》*阐述如何使用基本的和高级的 SQL 以及存储过程语言（SPL）例程来存取和处理数据库中的数据。*《IBM Informix: SQL 指南: 语法》*包含 SQL 和 SPL 的所有语法描述。*《IBM Informix: SQL 参考指南》*提供 SQL 的除语言语句外的各个方面的参考信息。

用户类型

本手册是为以下用户编写的:

- 数据库管理员
- 数据库服务器管理员
- 数据库应用程序程序员

本手册假定您具有以下背景:

- 对于计算机、操作系统和操作系统提供的实用程序的应用知识
- 使用关系数据库经验或熟悉数据库概念
- 一些计算机编程经验

如果您对关系数据库、SQL 或操作系统的经验有限，请参阅数据库服务器的*《IBM Informix: 入门指南》*以获取补充标题的列表。

软件相关性

编写本手册时，我们假定您使用下列其中一个数据库服务器:

- IBM Informix Dynamic Server V10.0
- IBM Informix Extended Parallel Server V8.50

关于语言环境的假定

IBM Informix 产品可以支持许多语言、文化和代码集。与字符集、整理和数字数据、货币、日期和时间的表示法相关的所有信息都集中在单一环境中，称为 Global Language Support（GLS）语言环境。

编写本手册中的示例时假定您正在使用缺省语言环境 **en_us.8859-1**。此语言环境支持日期、时间和货币的美国英语格式约定。另外，此语言环境还支持 ISO 8859-1 代码集，该代码集包括 ASCII 代码集和许多 8 位字符，如 é、è 和 ñ。

如果计划在数据或 SQL 标识符中使用非缺省字符，或者要遵照字符数据的非缺省整理规则，则需要指定适当的非缺省语言环境。

有关如何指定非缺省语言环境、附加语法和与 GLS 语言环境相关的其它注意事项的指示信息，请参阅*《IBM Informix: GLS 用户指南》*。

演示数据库

随数据库服务器产品一起提供的 DB–Access 实用程序包括以下一个或多个演示数据库:

- 对于所有 IBM Informix 数据库，**stores_demo** 数据库利用一家虚构的体育用品批发商的有关信息举例说明了关系模式。IBM Informix 手册中的许多示例基于 **stores_demo** 数据库。
- 对于 Extended Parallel Server，**sales_demo** 数据库举例说明了数据仓储应用程序的维模式。有关维数据建模的概念性信息，请参阅本*《IBM Informix: 数据库设计和实现指南》*的第 4 页。
- 对于 Dynamic Server，**superstores_demo** 数据库举例说明了对象 – 关系模式。**superstores_demo** 数据库包含扩展数据类型、类型和表继承以及用户定义的例程的示例。

有关如何创建和填充演示数据库的信息，请参阅*《IBM Informix: DB–Access 用户指南》*。有关这些数据库及其内容的描述，请参阅*《IBM Informix: SQL 参考指南》*。

用来安装演示数据库的脚本在 UNIX 平台上驻留在 **$INFORMIXDIR/bin** 目录中，在 Windows 环境中驻留在 **%INFORMIXDIR%\bin** 目录中。

Extended Parallel Server V8.50 中的新增功能

有关 IBM Informix Extended Parallel Server V8.50 中新增功能的描述，请参阅*《IBM Informix: 入门指南》*（V8.50）。

Dynamic Server V10.0 中的新增功能

有关 IBM Informix Dynamic Server V10.0 中的新增功能的描述，请参阅*《IBM Informix: 入门指南》*（V10.0）。

文档约定

本节描述此手册使用的约定。这些约定使得从文档集的本卷和其它卷中收集信息更容易。

讨论了以下约定:

- 印刷约定
- 其它约定
- 语法图

- 命令行约定
- 示例代码约定

印刷约定

本手册使用以下约定来介绍新术语、演示屏幕显示和描述命令语法等。

约定	含义
KEYWORD	编程语言语句中的所有主要元素（关键字）都以 serif 字体的大写字母显示。
斜体 ***斜体*** *斜体*	在文本中，新术语和强调的字以斜体显示。在语法和代码示例中，您要指定的变量值以斜体显示。
粗体字 **粗体字**	程序实体（如类、事件和表）、环境变量、文件和路径名以及界面元素（如图标、菜单项和按钮）的名称以粗体字显示。
等宽字体 *等宽字体*	产品显示的信息和您输入的信息以等宽字体显示。
KEYSTROKE	您要按的键以 sans serif 字体的大写字母显示。
>	此符号表示一个菜单项。例如: “选择**工具 > 选项**”意味着从**工具**菜单中选择**选项**项。

技巧: 当系统指示您“输入”字符或“执行”命令时，在输入后立即按 Enter 键。当系统指示您“输入”文本或“按”其它键时，则不需要按 Enter 键。

功能部件、产品和平台

功能部件、产品和平台标记标识包含特定于功能部件、产品或平台的信息的段落。此标记的某些示例如下:

Dynamic Server

标识特定于 IBM Informix Dynamic Server 的信息

Dynamic Server 结束

Extended Parallel Server

标识特定于 IBM Informix Extended Parallel Server 的信息

Extended Parallel Server 结束

仅适用于 UNIX

标识特定于 UNIX 平台的信息

仅适用于 UNIX 结束

仅适用于 Windows

标识特定于 Windows 环境的信息

仅适用于 Windows 结束

此标记可以应用于一节中的一个或多个段落。当整节应用于特定产品或平台时，此标记注释为标题文本的一部分，例如:

表排序（仅 Linux）

语法图

本指南使用由以下组件构建的语法图来描述语句的语法，以及除系统级别命令以外的所有命令的语法。

注: 从 2004 年开始，已重新规定语法图的格式以符合 IBM 标准。

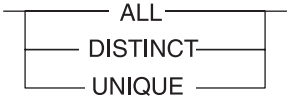
已对描绘 SQL 和命令行语句的语法图进行以下方面的更改:

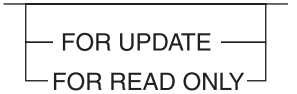
- 语句开头和结尾的符号现在为双箭头而不是位于末尾的垂直线。
- 语法段图开头和结尾的符号现在是垂直线而不是箭头。

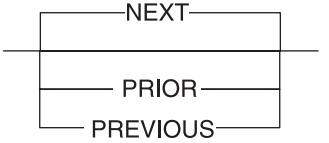
- 循环的可重复次数现在以图脚注的形式说明，而不是用门符号中的数字说明。
- 长于一行的语法语句现在在另一行继续而不是通过连续行向下循环。
- 产品或特定于条件的路径现在以图脚注而不是图标的形式说明。

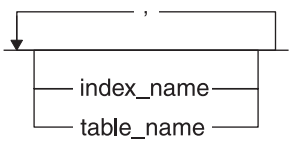
下表描述了语法图组件。

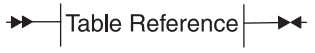
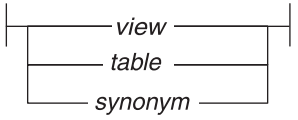
PDF 中描绘的组件	HTML 中描绘的组件	含义
	`>>----------`	语句开始。
	`---------->`	语句在下一行上继续。
	`>----------`	语句从先前行继续。
	`----------><`	语句结束。
	`--------SELECT----------`	必需项。
	`--+-----------------+---` `  '------LOCAL------'`	可选项。

	`---+-----ALL-------+---`
	`  +--DISTINCT-----+`
	`  '---UNIQUE------'`
		带选项的必需项。必须显示一项且仅显示一项。

	`---+------------------+---`
	`  +--FOR UPDATE-----+`
	`  '--FOR READ ONLY--'`
		带有选项的可选项显示在主行之下，你可以指定一个主行。

	`  .---NEXT---------.`
	`----+----------------+---`
	`  +---PRIOR--------+`
	`  '---PREVIOUS-----'`
		主行之下的值为可选，您可以指定一个值。如果您不指定一个项，则行之上的值将用作缺省值。

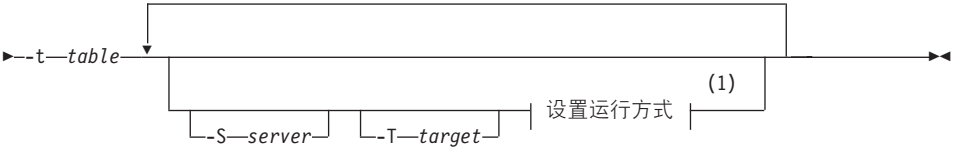
	`  .-------,-----------.`
	`  V                   |`
	`---+-----------------+---`
	`  +---index_name---+`
	`  '---table_name---'`
		可选项。允许几个项；每次重复之前必须有一个逗号。

PDF 中描绘的组件	HTML 中描绘的组件	含义
	>>-| Table Reference |-><	语法段的引用。
Table Reference 	表参考 |--+-----view--------+--| +------table------+ '----synonym------'	语法段。

如何阅读命令行语法图

以下命令行语法图使用一些在先前部分的表中列出的元素。

创建没有约定的作业

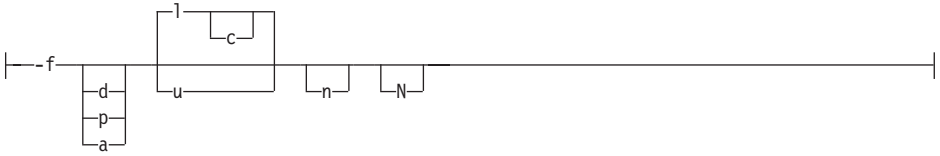


注:

1 请参阅第 17-4 页

此图中的第二行具有名为“设置运行方式”的段，根据图脚注，该段位于第 17-4 页上。该段显示在以下段图中（该图使用段开始和结束组件）。

设置运行方式:



要正确地构造命令，从左上方开始该命令。沿该图向右进行，包括想要的元素。该图中的元素区分大小写。

“创建没有约定的作业”图阐述了以下步骤:

1. 输入 **onpladm create job**，然后输入作业的名称。
2. （可选）输入 **-p**，然后输入项目的名称。
3. 输入以下必需元素:
   - **-n**
   - **-d** 和设备的名称
   - **-D** 和数据库的名称
   - **-t** 和表的名称
4. （可选）您可以选择一个或多个以下元素并重复它们任意次数:
   - **-S** 和服务器名称
   - **-T** 和目标服务器名称
   - 运行方式。要设置运行方式，请遵循“设置运行方式”段图，输入 **-f**，（可选）输入 **d**、**p** 或 **a**，然后（可选）输入 **l** 或 **u**。
5. 请遵循该图直至结束符。

图已完成。

关键字和标点符号

关键字是为语句和除系统级别命令以外的所有命令所保留的字。当关键字在语法图中出现时，它以大写字母显示。当在命令中使用关键字时，可以用大写或小写字母写关键字，但该关键字的拼写必须与在语法图中的拼写完全一样。

还必须在语句和命令中完全按照语法图中所显示的那样来使用任何标点符号。

标识符和名称

变量在语法图和示例中用作标识符和名称的占位符。可以根据上下文使用任意的名称、标识符或文字来替换变量。变量还可用来表示在附加语法图中扩充的复杂语法元素。当变量出现在语法图、示例或文本中时，它以*小写斜体字*的形式显示。

以下语法图使用变量来说明简单的 SELECT 语句的一般格式。

当写此格式的 SELECT 语句时，应使用特定列和表的名称来替换变量 *column_name* 和 *table_name*。

示例代码约定

本手册中出现了 SQL 代码的示例。除非另有说明，否则该代码并不特定于任何单个 IBM Informix 应用程序开发工具。

如果示例中仅列示了 SQL 语句，则未用分号定界它们。例如：您可能看到以下示例中的代码：

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

要对特定产品使用此 SQL 代码，必须应用该产品的语法规则。例如：如果正在使用 DB–Access，则必须用分号定界多个语句。如果正在使用 SQL API，则必须在每个语句的开始处使用 EXEC SQL，并在语句的结束处使用分号（或其它适当的定界符）。

技巧：代码示例中的省略号指示将在整个应用程序中添加更多的代码，但不必显示它也能描述正在讨论的概念。

有关对特定应用程序开发工具或 SQL API 使用 SQL 语句的详细指导，参见产品的手册。

附加文档

有关其它信息，请参阅以下类型的文档：

- 安装指南
- 联机说明
- Informix 错误消息
- 手册
- 联机帮助

安装指南

安装指南位于产品 CD 的 **/doc** 目录中，或位于产品压缩文件的 **/doc** 目录中（如果您是从 IBM Web 站点下载的产品压缩文件）。此外，您还可以从地址为 http://www.ibm.com/software/data/informix/pubs/library/ 的 IBM Informix 联机文档站点获取安装指南。

联机说明

以下部分描述了补充此手册中的信息的联机文件。在您开始使用 IBM Informix 产品之前，请检查这些文件。它们包含关于应用程序和性能问题的重要信息。

联机文件	描述	格式
TOC 说明	TOC（目录）注释文件提供到发行说明、已修订和已知缺陷文件以及单独手册标题的所有文档说明文件的超链接的综合目录。	HTML
文档说明	每个手册的文档说明文件包含一些重要信息和更正，这些重要信息补充手册中的信息或自出版以来已修改的信息。	HTML，文本
发行说明	发行说明文件描述了与 IBM Informix 产品的早期版本的功能差异，以及这些差异是如何影响当前产品的。对于某些产品，此文件还包含关于任何已知问题及其变通方法的信息。	HTML，文本
机器说明	（仅非 Windows 平台）机器说明文件描述了您在计算机上配置和使用 IBM Informix产品所必须采取的特定于平台的操作。	文本
已修订和已知缺陷文件	此文本文件列出了当前版本已识别的问题。它还列出了当前版本和以前版本中已修订的计算机报告的缺陷。	文本

查找联机说明

联机说明可从地址为 http://www.ibm.com/software/data/informix/pubs/library/ 的 IBM Informix 联机文档站点获得。此外，您还可以如下在安装之前或之后查找这些文件。

安装前

所有的联机说明都位于产品 CD 的 **/doc** 目录中。访问文档说明、发行说明和已修订及已知缺陷文件的最简便方法是通过从 TOC 说明文件的超链接访问。

机器说明文件以及已修订和已知缺陷文件仅以文本格式提供。

安装后

在缺省语言环境的 UNIX 平台上，文档说明、发行说明和机器说明文件位于 **$INFORMIXDIR/release/en_us/0333** 目录下。

Dynamic Server

在 Windows 上，文档和发行说明文件位于 **Informix** 文件夹中。要显示此文件夹，请从任务栏选择**开始 > 程序 > IBM Informix Dynamic Server 版本 > 文档说明** 或**发行说明**。

机器说明不适用于 Windows 平台。

Dynamic Server 结束

联机说明文件名

联机说明有以下文件格式:

联机文件	文件格式	示例
TOC 说明	*prod_os*_tocnotes_*version*.html	**ids_win_tocnotes_10.0.html**
文档说明	*prod_bookname*_docnotes_*version*.html/txt	**ids_hpl_docnotes_10.0.html**
发行说明	*prod_os*_relnotes_*version*.html/txt	**ids_unix_relnotes_10.0.txt**
机器说明	*prod*_machine_notes_*version*.txt	**ids_machine_notes_10.0.txt**
已修订和已知缺陷文件	*prod*_defects_*version*.txt	**ids_defects_10.0.txt** **client_defects_2.90.txt**
	ids_win_fixed_and_known_defects_*version*.txt	**ids_win_fixed_and_known_defects_10.0.txt**

Informix 错误消息

此文件是 Informix 产品和版本号的错误消息及其更正操作的综合索引。

在 UNIX 平台上，使用 **finderr** 命令来读取错误消息及其更正操作。

Dynamic Server

在 Windows 上，使用 Informix 错误消息实用程序来读取错误消息及其更正操作。要显示此实用程序，请从任务栏选择**开始 > 程序 > IBM Informix Dynamic Server 版本 > Informix 错误消息**。

Dynamic Server 结束

您还可以从地址为 http://www.ibm.com/software/data/informix/pubs/library/ 的 IBM Informix 联机文档站点访问这些文件。

手册

联机手册

随 IBM Informix 产品提供了包含电子格式的手册的 CD。您可安装文档或直接从 CD 访问此文档。要获取关于如何安装、读取和打印联机手册的信息，请参阅 CD

附带的安装插入。您还可以从地址为 http://www.ibm.com/software/data/informix/pubs/library/ 的 IBM Informix 联机文档站点获取相同的联机手册。

打印的手册

要订购硬拷贝手册，请联系销售代表或访问地址为 http://www.ibm.com/software/howtobuy/data.html 的 IBM 出版物中心 Web 站点。

联机帮助

随每个图形用户界面（GUI）提供的 IBM Informix 联机帮助显示了这些界面及其执行的功能的信息。使用每个 GUI 提供的帮助工具可显示联机帮助。

辅助选项

IBM 致力于使身有残疾的人可以访问我们的文档。这些书可以 HTML 格式获得，以便可以通过辅助技术（如屏幕阅读器软件）来访问它们。手册中的语法图可以点分十进制格式获得，该格式是仅在您正使用屏幕阅读器时才可用的访问格式。要获取关于点分十进制格式的更多信息，请参阅“辅助选项”附录。

IBM Informix Dynamic Server V10.0 和 CSDK V2.90 文档集

以下表列出了作为 IBM Informix Dynamic Server V10.0 和 CSDK V2.90 文档集一部分的手册。这些手册的 PDF 和 HTML 版本可在 http://www.ibm.com/software/data/informix/pubs/library/ 处获得。您可以从地址为 http://www.ibm.com/software/howtobuy/data.html 的 IBM 出版物中心订购这些手册的硬拷贝版本。

表 1. 数据库服务器手册

手册	主题
管理员指南	理解、配置和管理数据库服务器。
管理员参考大全	Informix Dynamic Server 的参考资料，如数据库服务器实用程序 **onmode** 和 **onstat** 的语法，以及配置参数、**sysmasters** 表以及逻辑日志记录的描述。
备份和恢复指南	当您使用 **ON-Bar** 和 **ontape** 实用程序来备份和恢复数据时，您需要理解的概念和方法。
DB-Access 用户指南	使用 **DB-Access** 实用程序来从 Informix 数据库访问、修改和检索数据。
DataBlade API Function Reference	DataBlade API 函数和 DataBlade API 支持的 ESQL/C 函数的子集。您可以使用 DataBlade API 来开发访问 Informix 数据库中的数据的客户机 LIBMI 应用程序和 C 用户定义的例程。

表 1. 数据库服务器手册 (续)

手册	主题
DataBlade API Programmer's Guide	DataBlade API, 是随 Dynamic Server 提供的 C 语言应用程序编程界面。您可以使用 DataBlade API 来开发访问存储在 Informix 数据库中的数据的客户机和服务器应用程序。
数据库设计和实现指南	设计、实现和管理 Informix 数据库。
Enterprise Replication 指南	如何设计、实现和管理 Enterprise Replication 系统, 以在多个数据库服务器之间复制数据。
错误消息文件	当使用 IBM Informix 产品时可能会收到的编号错误消息的原因和解决方案。
入门指南	描述与 IBM Informix Dynamic Server 捆绑的产品以及与其它 IBM 产品的互操作性。总结 Dynamic Server 的重要功能以及每个版本的新功能。
SQL 参考指南	关于 Informix 数据库、数据类型、系统目录表、环境变量和 stores_demo 演示数据库的信息。
SQL 指南: 语法	所有 Informix SQL 和 SPL 语句的语法的详细描述。
SQL 教程指南	SQL 教程（由 Informix 产品实现）, 描述使用关系数据库时会使用的基本理念和术语。
High-Performance Loader 用户指南	访问和使用 High-Performance Loader（HPL）以将大量数据装入到 Informix 数据库或从 Informix 数据库卸装大量数据。
Microsoft Windows 安装指南	在 Windows 上安装 IBM Informix Dynamic Server 的指示信息。
Unix and Linux 安装指南	在 UNIX 和 Linux 上安装 IBM Informix Dynamic Server 的指示信息。
J/Foundation Developer's Guide	为带有 J/Foundation 的 Informix Dynamic Server 以 Java 编程语言写用户定义的例程（UDR）。
Large Object Locator DataBlade Module User's Guide	使用 Large Object Locator, 可由其它创建或存储大对象数据的模块使用的基础 DataBlade 模块。Large Object Locator 使您能够对大对象创建单个一致界面, 并扩展大对象的概念以包含存储在数据库之外的数据。
迁移指南	转换为 Informix 数据库服务器的最新版本或从 Informix 数据库服务器的最新版本复原。在不同的 Informix 数据库服务器之间迁移。
Optical Subsystem Guide	Optical Subsystem, 一种支持在光盘上存储 BYTE 和 TEXT 数据的实用程序。
性能指南	配置和操作 IBM Informix Dynamic Server 以达到最佳性能。
R-Tree Index User's Guide	对合适的数据类型创建 R 树索引, 创建使用 R 树索引存取方法的新运算符类, 管理使用 R 树索引辅助存取方法的数据库。
SNMP Subagent Guide	IBM Informix 子代理程序, 允许简单网络管理协议（SNMP）网络管理器监视 Informix 服务器的状态。

表 1. 数据库服务器手册 (续)

手册	主题
Storage Manager 管理员指南	Informix Storage Manager（ISM），它管理 Informix 数据库服务器的存储设备和介质。
Trusted Facility Guide	Dynamic Server 的安全审计功能，包括审计日志的创建和维护。
用户定义的例程和数据类型开发者指南	如何定义新数据类型和启用用户定义的例程（UDR）以扩展 IBM Informix Dynamic Server。
Virtual-Index Interface Programmer's Guide	通过虚拟索引界面（VII）创建辅助存取方法（索引）以扩展 IBM Informix Dynamic Server 的内置索引模式。通常与 DataBlade 模块一起使用。
Virtual-Table Interface Programmer's Guide	通过虚拟表界面（VTI）创建主存取方法以便用户对不符合 Informix Dynamic Server 存储模式的 Informix 表和数据具有单个 SQL 界面。

表 2. 客户机／连接性手册

手册	主题
Client Products Installation Guide	在使用 UNIX、Linux 和 Windows 的计算机上安装 IBM Informix 客户机软件开发者工具箱（客户机 SDK）和 IBM Informix 连接。
Embedded SQLJ User's Guide	使用 IBM Informix Embedded SQLJ 在 Java 程序中嵌入 SQL 语句。
ESQL/C Programmer's Manual	嵌入式 SQL for C 的 IBM Informix 实现。
GLS 用户指南	Global Language Support（GLS）功能，它允许 IBM Informix API 和数据库服务器处理不同语言、文化习俗和代码集。
JDBC Driver Programmer's Guide	安装和使用 Informix JDBC 驱动程序，以从 Java 应用程序或 applet 内与 Informix 数据库连接。
.NET Provider Reference Guide	使用 Informix .NET Provider，使 .NET 客户机应用程序能够在 Informix 数据库中访问和操纵数据。
ODBC Driver Programmer's Manual	使用 Informix ODBC 驱动程序 API，访问 Informix 数据库并与 Informix 数据库服务器进行交互。
OLE DB Provider Programmer's Guide	安装和配置 Informix OLE DB Provider 以启用客户机应用程序，如 ActiveX Data Object（ADO）应用程序和 Web 页面，以访问 Informix 服务器上的数据。
Object Interface for C++ Programmer's Guide	C++ 对象界面的体系结构和完整类参考。

表 3. DataBlade 开发者工具箱指南

手册	主题
DataBlade Developer's Kit User's Guide	使用 BladeSmith 和 BladePack 开发和封装 DataBlade 模块。

表 3. DataBlade 开发者工具箱指南 (续)

手册	主题
DataBlade Module Development Overview	开发 DataBlade 模块的基本指导。包含阐述 DataBlade 模块的开发的示例。
DataBlade Module Installation and Registration Guide	安装 DataBlade 模块并使用 BladeManager 来在 Informix 数据库中管理 DataBlade 模块。

符合业界标准

美国国家标准协会（ANSI）和国际标准化组织（ISO）已联合为结构化查询语言（SQL）建立了一套业界标准。基于 IBM Informix SQL 的产品完全符合 SQL-92 入门级标准（发布为 ANSI X3.135-1992），该标准与 ISO 9075:1992 相同。此外，IBM Informix 数据库服务器的许多功能部件符合 SQL-92 中级和完全级标准以及 X/Open SQL 公共应用程序环境（CAE）标准。

IBM 欢迎您提出宝贵意见

我们希望了解您在我们的手册中发现有用的任何更正或说明，它将帮助我们改进未来版本。包括以下信息:

- 您正在使用的手册的名称和版本
- 段和页号
- 您对本手册的建议

请按以下电子邮件地址给我们发送评论:

ctscrcf@cn.ibm.com

此电子邮件地址专用于报告我们的文档中的错误和遗漏。要立即获取对技术问题的帮助，请联系 IBM 技术支持。

我们衷心感谢您的建议。

第 1 部分 数据库设计和实现基础

第 1 章 规划数据库

在本章中

本章描述了若干问题，数据库管理员（DBA）必须了解这些问题才能有效地规划数据库。本章讨论选择数据库的数据模型、使用符合 ANSI 的数据库以及对数据库使用定制语言环境。

选择数据库的数据模型

在使用 IBM Informix 产品创建数据库之前，必须决定要使用哪种类型的数据模型来设计数据库。本手册描述了下列数据库模型:

- 关系数据模型

  此数据模型代表用于联机事务处理（OLTP）的数据库设计。OLTP 的用途是在不丢失任何事务的前提下处理大量的小型事务。OLTP 数据库设计成能够满足企业的日常需求并为那些需求调整数据库性能。本手册的第 1 部分，『数据库设计和实现基础』描述了如何为 OLTP 构建和实现关系数据模型。第 2 部分，『管理数据库』讨论如何管理数据库。

- 对象关系数据模型

  Dynamic Server 支持对象关系数据库，这些对象关系数据库利用基本的关系设计原理，但还包括了诸如扩展数据类型、用户定义的例程、用户定义的数据类型

转换和用户定义的聚集之类的功能来扩展关系数据库的功能。本手册的第 3 部分，『对象关系数据库』讨论如何使用 Dynamic Server 的可扩展功能来扩展可在数据库中存储的数据种类以及在组织和存取数据方面提供更大的灵活性。

- 维数据模型

  此数据模型通常用于构建数据集市，数据集市是一种数据仓库。在数据仓储环境中，对数据库进行优化以便进行数据检索和分析。此类信息性处理称为联机分析处理（OLAP）或决策支持处理。本手册的第 4 部分，『维数据库』描述了如何为 OLAP 构建和实现维数据模型。

除了选择用来设计数据库的数据模型之外，还必须作出下列决定，这些决定确定了哪些功能可供使用数据库的应用程序使用:

- 应使用哪个数据库服务器？
  - Dynamic Server
  - Extended Parallel Server
- 数据库是否需要符合 ANSI？
- 数据库在它的表中是否将使用非英语语言字符？

本章的其余部分描述这些决定的含义并对您所作的决定将对数据库产生何种影响进行总结。

使用符合 ANSI 的数据库

当在 CREATE DATABASE 语句中使用 MODE ANSI 关键字时就会创建符合 ANSI 的数据库。然而，创建符合 ANSI 的数据库并不能确保此数据库*保持*符合 ANSI。如果对 ANSI 数据库执行非 ANSI 操作（如 CREATE INDEX），您就会接收到警告，但应用程序不会禁止该操作。

您可能会由于下列原因而想要创建符合 ANSI 的数据库:

- 对对象的特权和访问权

  ANSI 规则控制着对对象（如表和同义词）的特权和访问权。

- 名称隔离

  ANSI 表命名模式允许不同的用户在数据库中创建表而不会发生名称冲突。

- 事务隔离
- 数据恢复

  符合 ANSI 的数据库为 Dynamic Server 强制执行非缓冲型日志记录和隐式事务。

可以对符合 ANSI 的数据库和不符合 ANSI 的数据库使用相同的 SQL 语句。

符合 ANSI 的数据库和不符合 ANSI 的数据库之间的差别

您指定为符合 ANSI 的数据库与不符合 ANSI 的数据库在下列方面有所不同:

- 事务
- 事务日志记录
- 所有者命名
- 对对象的特权
- 缺省隔离级别
- 字符数据类型
- 十进制数据类型
- 转义字符
- 游标行为
- SQLCA 的 SQLCODE
- 同义词行为

事务

*事务*是可以作为单一工作单元对待的 SQL 语句集合。在符合 ANSI 的数据库中发出的所有 SQL 语句都将自动包含在事务中。对于不符合 ANSI 的数据库，事务处理是一个选项。

在不符合 ANSI 的数据库中，用 BEGIN WORK 语句和 COMMIT WORK 或 ROLLBACK WORK 语句将事务括起来。然而，在符合 ANSI 的数据库中，由于所有语句都自动地包含在事务中，所以不需要 BEGIN WORK 语句。只需要使用 COMMIT WORK 或 ROLLBACK WORK 语句指示事务结束。

有关事务的更多信息，请参阅第 4-1 页的第 4 章，『实现关系数据模型』和*《IBM Informix: SQL 教程指南》*。

事务日志记录

符合 ANSI 的数据库在运行时进行非缓冲型事务日志记录。在符合 ANSI 的数据库中，不能将日志记录方式更改为缓冲型日志记录，也不能关闭日志记录。

Dynamic Server 不符合 ANSI 的数据库在运行时可以进行缓冲型日志记录或非缓冲型日志记录。非缓冲型日志记录能够提供更全面的数据恢复，但缓冲型日志记录能够提供更好的性能。

Extended Parallel Server 不符合 ANSI 的数据库在运行时只能进行非缓冲型日志记录。非缓冲型日志记录能够提供更全面的数据恢复。

有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中对 CREATE DATABASE 语句的描述。

所有者命名

在符合 ANSI 的数据库中，强制执行所有者命名。当在 SQL 语句中提供对象名时，除非您就是该对象的所有者，否则 ANSI 标准要求该名称包括前缀 *owner*。*owner* 与*名称*的组合在数据库中必须是唯一的。如果您是该对象的所有者，则数据库服务器提供您的用户名来作为缺省值。

不符合 ANSI 的数据库不强制执行所有者命名。有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的『所有者名』段。

对对象的特权

关于在数据库中创建表时，缺省情况下将表级别特权授予哪些用户，符合 ANSI 的数据库和不符合 ANSI 的数据库会有所不同。ANSI 标准指定数据库服务器将任何表级别特权只授予表所有者（以及 DBA，如果他们不是同一用户的话）。然而，在不符合 ANSI 的数据库中，会将特权授予 PUBLIC。另外，数据库服务器提供了两个表级别特权“改变”和“索引”，这两个特权没有包括在 ANSI 标准中。

要运行用户定义的例程，您必须具有该例程的“执行”特权。在为符合 ANSI 的数据库创建具有所有者特权的过程时，只有用户定义的例程的所有者具有“执行”特权。当在不符合 ANSI 的数据库中创建具有所有者特权的例程时，缺省情况下数据库服务器会将“执行”特权授予 PUBLIC。

将 **NODEFDAC** 环境变量设置为“yes”时，会使不符合 ANSI 的数据库去模拟符合 ANSI 的数据库的行为: 在用户创建表或具有所有者特权的例程时，不自动向 PUBLIC 授予特权。有关特权的更多信息，请参阅第 6-1 页的第 6 章，『授予和限制对数据库的存取权』以及《*IBM Informix: SQL 指南: 语法*》中对 GRANT 语句的描述。

缺省隔离级别

数据库隔离级别指定程序与其它程序的并行操作相隔离的程度。所有符合 ANSI 的数据库的缺省隔离级别都是“可重复读”。支持事务日志记录的不符合 ANSI 的数据库的缺省隔离级别是“已提交读取”。不使用事务日志记录的不符合 ANSI 的数据库的缺省隔离级别是“未落实的读”。有关隔离级别的信息，请参阅《*IBM Informix: SQL 教程指南*》以及《*IBM Informix: SQL 指南: 语法*》中对 SET TRANSACTION 和 SET ISOLATION 语句的描述。

字符数据类型

如果数据库不符合 ANSI，在字符字段（CHAR、CHARACTER、LVARCHAR、NCHAR、NVARCHAR、VARCHAR 和 CHARACTER VARYING）接收到超出字段指定长度的字符串时不会出错。数据库服务器会截断多余的字符而不产生错误消息。因此，对于 CHAR(*n*) 列或变量，当插入或更新的值的长度超过 *n* 个字节时，不强制执行数据的语义完整性。

在符合 ANSI 的数据库中，如果任何字符字段（CHAR、CHARACTER、LVARCHAR、NCHAR、NVARCHAR 和 VARCHAR、CHARACTER VARYING）接收到超出字段指定宽度的字符串则会出错。

DECIMAL 数据类型

如果数据库不符合 ANSI，则声明时使用了精度却无小数位的 DECIMAL 数据类型可以存储指定精度的浮点值。如果既没有指定精度也没有指定小数位，则缺省精度为 16。

在符合 ANSI 的数据库中，所有 DECIMAL 值都为定点并且必须使用显式精度声明。如果没有为 DECIMAL 数据类型指定小数位，则小数位等于 0，并且只能存储整数值。

转义字符

在符合 ANSI 的数据库中，转义字符只能对百分号（%）和下划线（_）字符进行特殊转义。还可以使用转义字符对它本身进行转义。有关转义字符的更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的『条件』段。

游标行为

如果数据库不符合 ANSI，则声明 SELECT 语句的更新游标时需要使用 FOR UPDATE 关键字。SELECT 语句还必须符合下列条件:

- 它从单个表中进行选择。
- 它不包含任何聚集函数。
- 它不包含 DISTINCT、GROUP BY、INTO TEMP、ORDER BY、UNION 或 UNIQUE 子句和关键字。

在符合 ANSI 的数据库中，声明游标时 FOR UPDATE 关键字是隐含的，且*所有*满足以上列表所述限制的游标都是潜在的更新游标。可以在 DECLARE 语句中用 FOR READ ONLY 关键字指定游标是只读的。

有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中对 DECLARE 语句的描述。

SQL 通信区域的 SQLCODE 字段

如果没有任何行满足 DELETE、INSERT INTO *tablename* SELECT、SELECT...INTO TEMP 或 UPDATE 语句的搜索条件，则数据库服务器将 SQLCODE 设置为 100（如果数据库符合 ANSI 的话）或 0（如果数据库不符合 ANSI 的话）。

有关更多信息，请参阅《*IBM Informix: SQL 教程指南*》中对 SQLCODE 的描述。

同义词行为

同义词在符合 ANSI 的数据库中始终是专用的。如果您尝试创建公用同义词或使用 PRIVATE 关键字来在符合 ANSI 的数据库中指定专用同义词，则您会接收到错误。

有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中对 CREATE SYNONYM 语句的描述。

确定现有数据库是否符合 ANSI

以下列表描述用来确定数据库是否符合 ANSI 的两种方法:

- 从 **sysmaster** 数据库中，可以执行以下语句:

  ```
  SELECT name,is_ansi FROM sysmaster:sysdatabases
  ```

  对于数据库服务器中的每个数据库，此查询对符合 ANSI 的数据库返回值 1，对不符合 ANSI 的数据库返回值 0。

- 如果您正在使用 SQL API（如 IBM Informix ESQL/C），则可以测试 “SQL 通信区域”（SQLCA）。明确地说，在使用 DATABASE 或 CONNECT 语句打开符合 ANSI 的数据库后，SQLCAWARN 结构中的第三个元素立即包含 W。有关 SQLCA 的信息，请参阅《*IBM Informix: SQL 教程指南*》或 SQL API 手册。

对数据库使用定制语言环境（GLS）

Global Language Support（GLS）允许您使用不同的语言环境。GLS 语言环境是为特定语言或文化定义了约定的环境。

缺省情况下，IBM Informix 产品使用美国英语 ASCII 代码集并在具有 ASCII 整理顺序的美国英语环境中工作。如果您打算使用任何下列功能，则将环境设置为适应非缺省语言环境:

- 数据包含非 ASCII 字符
- 用户指定的对象名包含非 ASCII 字符
- 与非缺省代码集的排序和整理顺序一致

- 特定于文化的整理和排序顺序，如在字典或电话号码簿中使用的那些整理和排序顺序

有关 GLS 环境变量的描述以及如何实现非缺省语言环境的详细信息，请参阅《*IBM Informix: GLS 用户指南*》。

第 2 章 构建关系数据模型

在本章中

创建关系数据库的第一步是构造数据模型：要存储的数据的精确的完整定义。本章提供了一种数据建模方法的概述。有关定义数据模型的特定于列的属性的信息，请参阅第 3-1 页的第 3 章，『选择数据类型』。要了解如何实现本章描述的数据模型，请参阅第 4-1 页的第 4 章，『实现关系数据模型』。

要理解本章中的材料，您需要对 SQL 及关系数据库理论有基本的了解。

构建数据模型

您对数据库中的数据的类型以及需要如何组织该数据已经有了一些概念。本信息是数据模型的开始。构建具有形式符号表示法的数据模型有下列优点:

- 您可周密地考虑数据模型。

  脑海中的模型通常包含未经检查的假定；当设计正式出台时，您会证实这些假定。
- 可以使设计更容易地让其他人理解。

  正式的语句使模型成为显式的，因此别人能够以同一格式返回意见和建议。

实体关系数据模型概述

存在多种正式的数据建模方法。大多数方法迫使您全面而精确地工作。如果您了解某种方法，则务必使用该方法。

本章提供了实体关系（E-R）数据模型的摘要。 E-R 数据建模方法遵循这些步骤:

1. 标识和定义主体数据对象（实体、关系和属性）。
2. 使用 E-R 方法来对数据对象制图。
3. 将 E-R 数据对象转换成关系构造。
4. 解析逻辑数据模型。
5. 将逻辑数据模型规范化。

本章讨论了步骤 1 至 5。第 4 章讨论了将逻辑数据模型转换为物理模式的最终步骤。

数据建模的最终成果是类似于第 2-25 页的图 2-21（它显示了个人电话号码簿的表的最终集合）中编码的完全定义的数据库设计。个人电话号码簿是本章开发的一

个示例。由于该示例相当小，可在一个章节中完全地开发出来，但也足够大，能显示整个方法，所以我们使用此示例而不是使用演示数据库。

标识和定义主体数据对象

要创建数据模型，您首先标识和定义主体数据对象：实体、关系和属性。

发现实体

*实体*是用户特别感兴趣的主体数据对象。它通常是要在数据库中记录的人员、地点、事物或事件。如果数据模型是语言，则实体将是名词。软件附带提供的演示数据库包含下列实体：*customer*、*orders*、*items*、*stock*、*catalog*、*cust_calls*、*call_type*、*manufact* 和 *state*。

选择可能的实体

您大概可以立即列出数据库的若干实体。请列出您可以标识的所有实体的初步列表。然后，与数据库的潜在用户会谈，了解他们对必须在数据库中记录的内容的意见。确定每个实体的基本特征，如“必须至少有一个地址与名称相关联”。您所作的关于实体的所有决定都将成为*商业规则*。第 2-5 页的电话薄示例提供了本章中的示例的一些商业规则。

以后，在将数据模型*规范化*时，一些实体可以扩充或者变为其它数据对象。有关更多信息，请参阅第 2-24 页的『将数据模型规范化』。

实体列表

当实体列表似乎已完整时，请检查列表以确保每个实体都具有下列性质：

- 它是重要的。

  只列示对数据库用户重要的实体以及值得花费精力和成本进行计算机制表的实体。

- 它是非特殊的。

  只列示事物的类型而不是个别的实例。例如：*交响乐*可能是一个实体，但*贝多芬第五交响乐*将是实体实例或实体出现。

- 它是基本的。

  只列示独立存在并且不需要别的事物对它们进行说明的实体。称作特性、特征或描述的事物都不是实体。例如：*部件号*是称为*部件*的基本实体的特征。并且，不要列示可以从其它实体派生的事物；例如：避免任何可以在 SELECT 表达式中计算得出的和、平均数或其它数量。

- 它是一元的。

确保您命名的每个实体都表示单个类。不能将它划分成子类别（每个子类别都具有自己的特征）。在第 2-5 页的图 2-1 中的电话号码簿示例中，电话号码（这是一个表面上很简单的实体）实际上由三个类别组成，每个类别都具有不同的特征。

这些选择既不简单也不是自动的。要发现实体的最佳选择，您必须仔细考虑要存储的数据的性质。当然，那就是正式数据模型的关键所在。下一节详细描述电话号码簿示例。

电话号码簿示例

假定您为个人电话号码簿创建数据库。数据库模型必须记录用户需要的人员和组织的名称、地址和电话号码。

首先定义实体。仔细查看电话号码簿中的某一页以标识它包含的实体。第 2-5 页的图 2-1 显示了电话号码簿中的样本页。

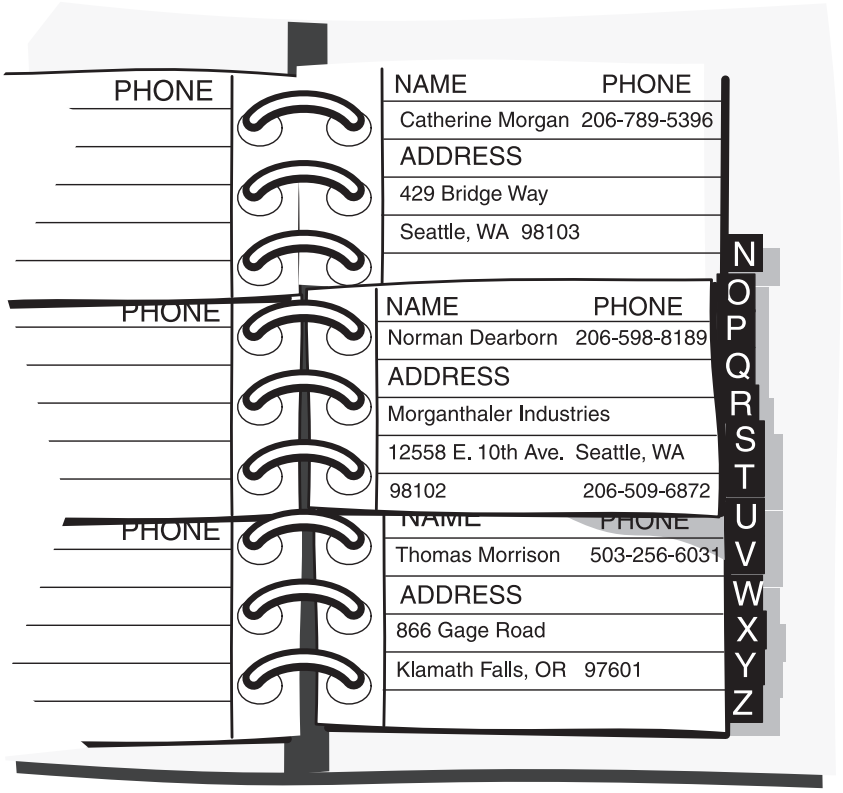


图 2-1. 电话号码簿的部分页

现有数据的物理形式可能会把您引入岐途。注意不要让电话号码簿中的页和条目布局误导您尝试指定表示电话号码簿中的条目（带有名称、电话号码和地址字段的按字母顺序排列的记录）的实体。您要做的是对数据而不是对介质进行建模。

非特殊且重要的实体: 首先我们会看到电话号码簿中记录的实体包含下列各项:

- （人员和组织的）名称
- 地址
- 电话号码

这些实体符合前面的条件吗？很明显，它们对模型很重要并且是非特殊的。

基本实体: 一项很好的测试是询问实体的数目是否可以独立于任何其它实体而变化。电话号码簿有时会列示没有电话号码或当前地址的人员（搬了家或换了工作的人员），并且也可以同时列示多个人员使用的地址和电话号码。这三个实体的数目全都可以独立地变化；这一事实明显表明它们是基本的，而不是从属的。

一元实体: 名称可以分为人员姓名和企业名。在此模型中，您决定所有名称都应具有相同的特征；即，您没有计划对公司和人员分别记录不同的信息。同样，您决定只存在一种类型的地址；您不需要将家庭地址视为与商业地址不同。

然而，您也认识到存在多种类型的电话号码。*语音*号码由人员接听，*传真*号码连接至传真机，而*调制解调器*号码连接至计算机。您决定要记录关于每类号码的不同信息，因此这三种类型是不同的实体。

对于个人电话号码簿示例，您决定要跟踪下列实体:

- 名称
- 地址（邮寄）
- 电话号码（语音）
- 电话号码（传真）
- 电话号码（调制解调器）

对实体制图

在本章的后面，您可以学习到如何使用 E-R 图。现在，为电话号码簿示例中的每个实体创建一个独立的矩形框，如图 2-2 所示。第 2-14 页的『对数据对象制图』显示了如何通过关系来使实体成为一体。

图 2-2. 个人电话号码簿示例中的实体

定义关系

在选择数据库实体之后，您需要考虑它们之间的关系。关系并不总是明显的，但必须找到所有值得记录的关系。确保找到所有关系的唯一方法是详尽地列示所有的可能关系。考虑每一对实体 *A* 和 *B* 并询问 “*A* 与 *B* 之间存在什么关系？”

关系是两个实体之间的关联。通常，连接两个实体的动词或介词意味着存在关系。实体之间的关系是从*连接性*、*存在相关性*和*基数*等方面描述的。

连接性

连接性指的是实体实例的数目。实体实例是实体的特定出现。图 2-3 显示三种类型的连接性分别是一对一（写作 1:1）、一对多（写作 1:n）和多对多（写作 m:n）。

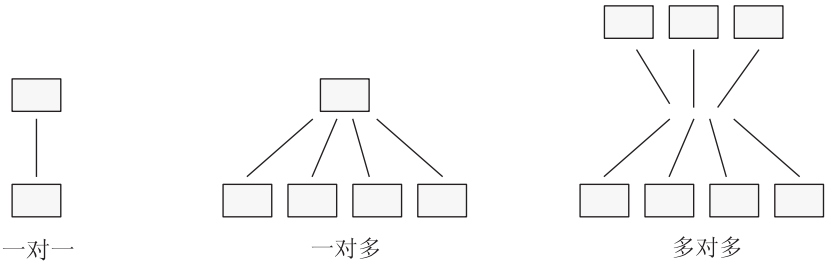


图 2-3. 关系中的连接性

例如: 在电话号码簿示例中，地址可以与多个名称相关联。所以，地址与名称实体之间的关系的连接性就是一对多（1:n）。

存在相关性

存在相关性描述关系中的实体是可选的还是必需的。请分析商业规则以标识实体是否必须存在于关系中。例如: 商业规则可能会指定地址必须与名称相关联。这样的关联指示了名称与地址实体之间的关系是必需存在相关性。可选存在相关性的示例可以是指示某个人可能有也可能没有子女的商业规则。

基数

基数对实体在关系中可以出现的次数加以约束。1:1 关系的基数始终为 1。但 1:n 关系的基数是开放的; *n* 可以是任何数字。如果需要对 *n* 设置上限，请对关系指定基数。例如: 在商店销售示例中，可以限制客户每次可以购买的商品项数。您通常使用应用程序或存储过程语言（SPL）来设置基数约束。

发现关系

一种很方便的发现关系的方法是准备一个矩阵，该矩阵先在每一行上列出所有实体的名称，然后在每一列上再次这样做。图 2-4 中的矩阵反映了个人电话号码簿的实体。

	name	address	number (voice)	number (fax)	number (modem)
name					
address					
number (voice)					
number (fax)					
number (modem)					

图 2-4. 反映个人电话号码簿的实体的矩阵

您可以忽略此矩阵的阴影部分。您必须考虑对角单元；即，您必须询问 “*A* 与另一个 *A* 之间存在什么关系？” 这个问题。在此模型中，答案始终是 “无”。在 name 与 name 或者 address 与另一个 address 之间不存在关系，至少您需要在此模型中记录 “无”。当 *A* 与另一个 *A* 之间存在关系时，表示找到了*递归*关系。（请参阅第 2-23 页的『解析其它特殊关系』。）

对于答案明显为 “无” 的所有单元格，在矩阵中写下 “无”。图 2-5 显示了当前矩阵。

	name	address	number (voice)	number (fax)	number (modem)
name	none				
address		none			
number (voice)			none		
number (fax)				none	
number (modem)					none

图 2-5. 包含初始关系的矩阵

尽管在此模型中没有实体与它们本身相关，但在其它模型中情况并非总是如此。典型的示例是：一个雇员是另一个雇员的经理。另一个示例出现在制造业：一个部件实体是另一个部件的组件。

在其余单元格中，写下行上的实体与列中的实体之间存在的连接性关系。下列类型的关系是有可能的:

- *一对一*（1:1），在此关系中，对于一个实体 *B*，不会存在多个实体 *A*，并且对于一个 *A*，也不会有多个 *B*。
- *一对多*（1:n），在此关系中，绝不会存在多个实体 *A*，但可以有若干个实体 *B* 与 *A* 相关（反之亦然）。
- *多对多*（m:n），在此关系中，可以有若干个实体 *A* 与一个 *B* 相关，并且可以有若干个实体 *B* 与一个 *A* 相关。

一对多关系最为常见。电话号码簿模型显示了一对多和多对多关系。

如第 2-9 页的图 2-5 所示，第一个未填充的单元格表示名称与地址之间的关系。这些实体之间存在什么样的连接性？您可能会问自己“可以有多少个名称与一个地址相关联？”。您决定一个名称可以有零个或一个地址，但不能有多个。您在 **name** 对面及 **address** 下面写下 `0-1`，如图 2-6 所示。

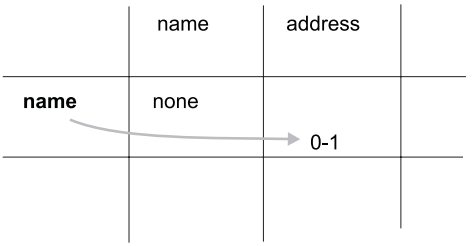


图 2-6. Name 与 Address 之间的关系

问问您自己：有多少个地址可以与一个名称相关联。您决定一个地址可以与多个名称相关联。例如：您可以了解到多个人在一间公司工作或两个以上的人的住所在同一地址。

地址可以与零个名称相关联吗？即，可能存在没有任何名称使用的地址吗？您的决定是“是的，可以存在”。在 **address** 下面及 **name** 对面，您写下 `0-n`，如图 2-7 所示。

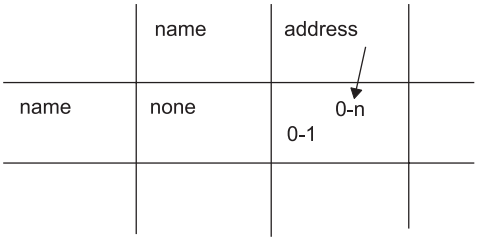


图 2-7. Address 与 Name 之间的关系

如果您决定除非地址与至少一个名称相关联否则不能存在，则写下 `1-n` 而不是 `0-n`。

当在任何一端将关系的基数限制为 1 时，就是 1:n 关系。在这种情况下，名称与地址之间的关系是 1:n 关系。

现在考虑第 2-9 页的图 2-5 中的下一个单元格：名称与语音电话号码之间的关系。名称可以与多少个语音电话号码相关联，是一个还是多个？当您查看电话号码簿时，您会看到通常为某个人记录多个电话号码。一个繁忙的推销员会有家庭电话号码、办公室电话号码、寻呼机号码和车载电话号码。但也可能有不带相关联电话号码的名称。您在 **name** 对面及 **number (voice)** 下面写下 `0-n`，如图 2-8 所示。

	name	address	number (voice)	
name	none	0-n 0-1	0-n	

图 2-8. Name 与 Number 之间的关系

此关系的另一端是什么？有多少个名称可以与一个语音电话号码相关联？您决定只有一个名称可以与一个语音电话号码相关联。电话号码可以与零个名称相关联吗？您决定除非某人使用某个电话号码，否则不需要记录该电话号码。您在 **number (voice)** 下面及 **name** 对面写下 1，如图 2-9 所示。

	name	address	number (voice)	
name	none	0-n 0-1	1 0-n	

图 2-9. Number 与 Name 之间的关系

要以同一方式填写矩阵的其余部分，请考虑下列因素:

- 一个名称可以与多个传真号码相关联；例如: 一间公司可以有若干台传真机。反过来，一个传真号码可以与多个名称相关联；例如: 若干个人员可以使用同一个传真号码。
- 调制解调器号码必须刚好与一个名称相关联。（这是为了使示例复杂化而作的任意规定；请将其作为一项设计需求予以接受。）然而，一个名称可以有多个相关联的调制解调器号码；例如: 一台公司计算机可以带有若干拨号线路。
- 尽管现实世界中的语音电话号码与地址之间、调制解调器号码与地址之间以及传真号码与地址之间存在着一些关系，但在此模型中需要记录“无”。通过 *name*，已存在间接的关系。

图 2-10 显示了完成后的矩阵。

	name	address	number (voice)	number (fax)	number (modem)
name	none	0-n 0-1	1 0-n	1-n 0-n	1 0-n
address		none	none	none	none
number (voice)			none	none	none
number (fax)				none	none
number (modem)					none

图 2-10. 电话号码簿的完成后矩阵

矩阵显示的其它决定包括传真号码与调制解调器号码之间、语音电话号码与传真号码之间或语音电话号码与调制解调器号码之间不存在任何关系。

您可能会不赞同其中某些决定（例如: 不支持语音电话号码与调制解调器号码之间的关系）。但是，为了实现本示例，这些就是我们的商业规则。

对关系制图

现在，保存您在本节创建的矩阵。您将在第 2-14 页的『对数据对象制图』学习如何创建 E-R 图。

标识属性

实体包含*属性*，属性是特征或修饰符、性质、数量或特色。属性是关于实体的事实或不可分解的信息部分。稍后，当将实体表示为表时，将把它的属性作为新列添加到模型中。

在可以标识数据库属性之前，必须标识实体。在确定实体之后，问问自己“我需要了解关于每个实体的哪些特征？”。例如: 在 *address* 实体中，您可能需要关于 *street*、*city* 和 *zip code* 的信息。*address* 实体的这些特征中的每一个都将成为属性。

为实体选择属性

要选择属性，请选择具有下列性质的属性:

- 它们是重要的。

只包括对数据库用户有用的属性。

- 它们是直接的，而不是派生的。

  可以从现有属性派生（例如: 通过 SELECT 语句中的表达式）的属性不应该作为模型的一部分。派生数据会使数据库的维护复杂化。

  在设计的后阶段，您可以考虑添加派生属性以改进性能，但在此阶段，请将它们排除在外。有关如何改进数据库服务器的性能的信息，请参阅《*IBM Informix: 性能指南*》。

- 它们是不可分解的。

  属性只可以包含单个的值，而不能包含列表或重复组。必须将组合值分到个别的属性中。

- 它们包含同一类型的数据。

  例如: 在生日属性中，您只应输入日期值，而不应输入名称或电话号码。

属性的定义规则与列的那些定义规则相同。有关如何定义列的信息，请参阅第 2-18 页的『对列设置约束』。

将下列属性添加至电话号码簿示例以生成第 2-16 页的图 2-15 所示的图:

- 将街道、城市、州和邮政编码添加至 *address* 实体。
- 将生日、电子邮件地址、周年纪念日和子女的名字添加至 *name* 实体。
- 将类型添加至 *voice* 实体以区分车载电话、家庭电话和办公室电话。语音电话号码只能与一种语音类型相关联。
- 将照管传真机的时间添加至 *fax* 实体。
- 将调制解调器是支持 9600 波特率、14400 波特率还是 28800 波特率添加至 *modem* 实体。

列示属性

现在，列示电话号码簿示例的属性以及您认为它们所属的实体。列表看起来应该象图 2-11。

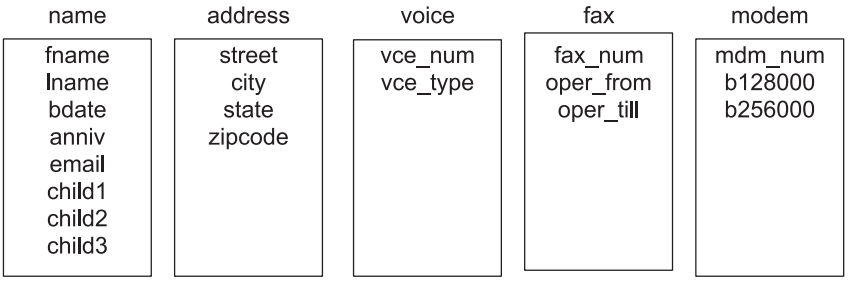


图 2-11. 电话号码簿示例的属性

关于实体出现

实体出现是一个附加的数据对象。表中的每一行都表示实体的单一特定出现。例如: 如果 *customer* 是实体，则 **customer** 表表示客户的想法；在这个表中，每一行都表示一个特定的客户，如 Sue Smith。记住，实体成为表，属性成为列，实体出现成为行。

对数据对象制图

现在，您了解并理解了数据库中的实体和关系，这是关系数据库设计过程中最为重要的部分。在确定实体和关系之后，用于显示您在数据库设计期间的思考过程的方法可能很有用。

大多数数据建模方法提供了一些途径来以图形方式显示实体和关系。IBM Informix 文档使用 E-R 图方法，此方法最初是由 C. R. Bachman 开发的。E-R 图用于以下用途。这些图能够:

- 对组织的信息性需求建模
- 标识实体及它们的关系
- 提供数据定义的起始点（数据流图）
- 为应用程序开发者以及数据库和系统管理员提供优良的文档源
- 创建数据库的逻辑设计，此逻辑设计可转换为物理模式

存在若干种不同样式的 E-R 图。如果您已经有了喜欢的样式，请使用它。图 2-12 显示了样本 E-R 图。

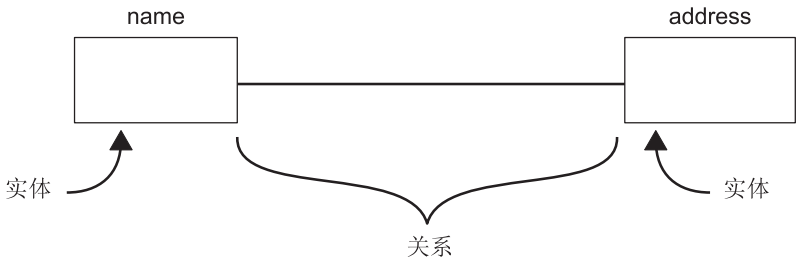


图 2-12. 实体关系图的符号

在 E-R 图中，一个框表示一个实体。线表示连接实体的关系。另外，图 2-13 显示了如何使用图形项来显示关系的下列特征:

- 关系链接上的圆指示关系中的*可选性*（可以出现零个实例）。
- 关系链接上的小竖线指示该实体*刚好有一个*实例与另一个实体相关联（将该竖线看作 1）。
- 末端分叉表示关系中的*多个*。

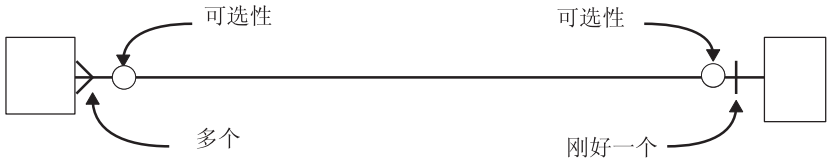


图 2-13. 实体关系图中的关系的部件

阅读 E-R 图

您首先从左到右读图，然后从右到左读图。对于图 2-14 中的 *name-address* 关系，您阅读关系的方式如下：名称可以与零个或刚好一个地址相关联；地址可以与零个、一个或许多个名称相关联。

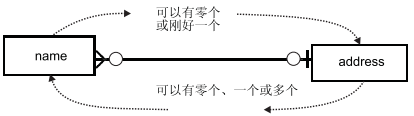


图 2-14. 阅读实体关系图

电话号码簿示例

图 2-15 显示了电话号码簿示例并包括实体、关系和属性。此图包括您使用矩阵建立的关系。在研究图符号之后，请将图 2-15 中的 E-R 图与第 2-12 页的图 2-10 中的矩阵作比较。请您自行验证两幅图中的关系是否相同。

在最初设计模型时，矩阵（如第 2-12 页的图 2-10）是非常有用的工具，这是因为当您填写该矩阵时，就会迫使您思考每一种可能的关系。然而，这些关系也出现在诸如图 2-15 之类的图中，当您复查现有的模型时，这种类型的图可能更易于阅读。

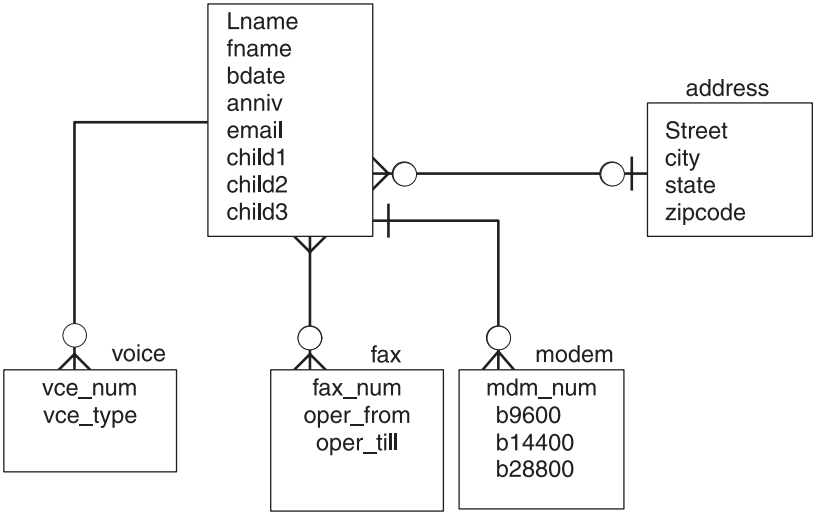


图 2-15. 电话号码簿示例的初步实体关系图

在图完成之后

本章的其余部分描述如何执行下列任务:

- 将实体、关系和属性转换为关系构造。
- 解析 E-R 数据模型。
- 将 E-R 数据模型规范化。

第 4 章说明了如何根据 E-R 数据模型来创建数据库。

将 E-R 数据对象转换为关系构造

您到目前为止所了解的所有数据对象（实体、关系、属性和实体出现）都转换为 SQL 表，也就是表、列和行之间的连接。数据库的表、列和行必须满足第 2-17 页的『定义表、行和列』中的规则。

在将数据对象规范化之前，请检查它们是否满足这些规则。要将数据对象规范化，请分析实体、关系和属性之间的相关性。第 2-24 页的『将数据模型规范化』对规范化作了讨论。

在将数据模型规范化之后，可使用 SQL 语句来创建基于数据模型的数据库。第 4 章描述了如何创建数据库并提供了电话号码簿示例的数据库模式。

您选择的每个实体都作为模型中的一个表来表示。表代表的是抽象概念的实体，每一行都表示该实体的个别特定*出现*。另外，实体的每个属性都由表中的一列表示。

下列思想是大多数关系数据模型方法（包括 E-R 数据模型）的基础。在设计数据模型时，请遵循下列规则以便将模型规范化时能够节省时间和减轻工作量。

定义表、行和列

您已经熟悉了由*行*和*列*组成的*表*的思想。但是，在定义正式数据模型的表时，必须遵守下列规则:

- 行必须是独立的。

  表的每一行都是独立的，并且不依赖于*同*一个表的任何其它行。因此，表中的行的顺序在模型中不重要。即使将表的所有行打乱成具有随机顺序，模型也应该依旧正确。

  在实现数据库之后，可以告知数据库服务器按特定顺序存储行以提高效率，但该顺序并不影响模型。

- 行必须是唯一的。

在每一行中，某些列必须包含唯一的值。如果没有任何一列具有此属性，则作为一个整体的某一组列的值在每一行中必须不同。

- 列必须是独立的。

  表中的列的顺序在模型中是没有意义的。即使将各列重新排列，模型也应该依旧正确。

  在实现数据库之后，那些使用星号来表示*所有列*的程序和存储查询依赖于列的最终顺序，但该顺序不影响模型。

- 列值必须是一元的。

  列只可以包含单个的值，而不能包含列表或重复组。必须将组合值分离到单独的列中。例如：如果您决定将人员的名和姓看作独立的值，就象本章中的示例显示的那样，则姓名必须位于独立的列中，而不能位于单个 **name** 列中。

- 每一列都必须具有唯一的名称。

  同一个表中的两列不能共享同一个名称。然而，可以有多个包含相似信息的列。例如：电话号码簿示例中的 **name** 表包含子女姓名的列。可以将每个列命名为 *child1* 和 *child2*，等等。

- 每一列都必须包含单一类型的数据。

  列必须包含具有相同数据类型的信息。例如：标识为整数的列必须只包含数字信息，而不能包含名称中的字符。

如果您以前只使用过作为数组或顺序文件组织的数据，则这些规则似乎不太自然。然而，关系数据库理论指出仅利用遵循这些规则的表、行和列就可以表示所有类型的数据。当您有了少许经验之后，您就会下意识地遵循这些规则了。

对列设置约束

当使用 CREATE TABLE 语句来定义表和列时，您对每一列进行约束。这些约束指定是要让该列包含字符还是数字、要让日期使用的格式以及其它约束。当*域*标识属性可以具有的有效值的集合时，它就描述了约束。

域特征

创建表时，您定义列的域特征。列可以包含下列域特征：

- 数据类型（INTEGER、CHAR 和 DATE 等等）
- 格式（例如：yy/mm/dd）
- 范围（例如：1000 至 5400）
- 含义（例如：序列号）
- 可允许的值（例如：仅等级 S 或 U）

- 唯一性
- 空支持
- 缺省值
- 引用约束

有关如何定义域的信息，请参阅第 3 章。有关如何创建表和数据库的信息，请参阅第 4 章。

确定表的键

表的列或者是*键*列或者是*描述符*列。键列是能够唯一地标识表中的特定行的列。例如：每个雇员的社会保险号是唯一的。描述符列指定表中特定行的非唯一特征。例如：两个雇员可以拥有相同的名字 Sue。名字 Sue 是雇员的非唯一特征。在表中，主要类型的键是主键和外键。

创建表时，指定主键和外键。主键和外键用来在物理上使表相关。下一个任务是指定每个表的主键。也就是说，必须标识表的某个可量化特征，此特征将每一行与所有其它各行区别开。

主键

表的*主键*是一个列，它的值在每一行中都不同。由于它们不相同，所以它们使每一行都是唯一的。如果不存在任何一个这样的列，则主键是两个或更多个列的*组合*，当这些列的值放到一起时，它们在每一行中都不同。

模型中的每个表都必须具有主键。此规则自动从规定所有行都必须唯一的规则得出。如果有必要的话，主键由所有的列组合而成。

为了提高效率，主键应该是数字数据类型（INT 或 SMALLINT）、SERIAL 或 SERIAL8 数据类型或者是短字符串（如用作代码的字符串）。建议您不要使用长字符串来作为主键。

主键列中绝不会允许空值。空值是不可比较的；也就是说，不能称它们相同或不相同。因此，它们不能使某一行对于其它各行来说是唯一的。如果某一列允许空值，则它不能作为主键的一部分。

某些实体具有现成的主键，如目录代码或标识号，它们是在模型外部定义的。有时可将多个列或一组列用作主键。有资格成为主键的所有列或组都称为*候选键*。所有候选键都值得您记录下来，这是因为它们的唯一性属性使它们在 SELECT 操作中是可预测的。

组合键: 一些实体缺少具有可靠唯一性的特征。不同的人可以拥有完全相同的姓名；不同的书籍可以具有完全相同的标题。通常，您可以找到作为主键工作的属

性组合。例如: 人们很少会有完全相同的姓名和完全相同的地址，不同的书籍也很少具有完全相同的标题、作者和出版日期。

系统指定的键: 系统指定的主键通常优于组合键。系统指定的键是当某个实体最初进入数据库时与该实体的每个实例相连接的数字或代码。最容易实现的系统指定的键是序列号，这是因为数据库服务器可以自动生成它们。Informix 数据库服务器为序列号提供了 SERIAL 和 SERIAL8 数据类型。然而，使用数据库的人可能不喜欢简单的数字代码。其它代码可以基于实际的数据；例如: 雇员标识代码可以基于人员的姓名首字母与雇佣他们的日期的各个位的组合。在电话号码簿示例中，对 **name** 表使用系统指定的主键。

外键（连接列）

*外键*是一个表中的一列或一组列，它包含与另一个表中的*主键*相匹配的值。外键用来连接表。图 2-16 显示了演示数据库中的 **customer** 和 **order** 表的主键和外键。

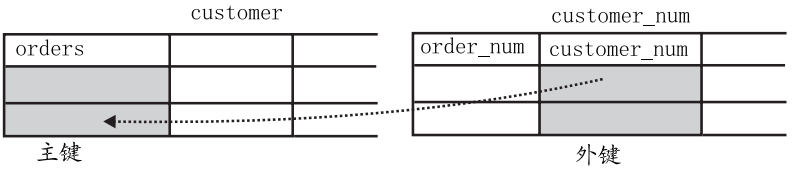


图 2-16. Customer-Order 关系中的主键和外键

技巧: 为了便于维护和使用表，重要的是为主键和外键选择名称以使关系变得明显。在图 2-16 中，主键和外键列具有相同的名称 **customer_num**。另外，可以将图 2-16 中的列命名为 **customer_custnum** 和 **orders_custnum**，以使每个列具有相异的名称。

模型中所有出现外键的位置都会加以标注，这是因为它们的存在会对从表中删除行的能力加以限制。在可以安全地删除行之前，必须删除所有通过外键引用该行的行，或者必须用特殊的语法来定义关系，以允许使用单个删除命令来从主键和外键列中删除行。数据库服务器不允许违反引用完整性的删除操作。

为了保持引用完整性，在删除外键行引用的主键之前，应删除所有那些外键行。如果对数据库施加引用约束，则数据库服务器不允许删除带有匹配外键的主键。数据库服务器也不允许添加未引用现有主键值的外键值。有关参照完整性的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

将键添加至电话号码簿图

图 2-17 显示了对主键和外键的初始选择。此图反映了一些重要的决定。

对于 **name** 表，选择了主键 **rec_num**。**rec_num** 的数据类型是 SERIAL。**rec_num** 的值由系统生成。如果查看 **name** 表中的其它列（或属性），您会看到与列相关联的数据类型大部分基于字符。在这些列中，没有任何一个能单独作为主键的合适候选键。如果将表的元素组合成组合键，则会创建不方便的键。SERIAL 数据类型提供了一个键，这个键还可用来将其它表连接到 **name** 表。

voice、**fax**、**modem** 和 **address** 表每个都通过 **rec_num** 键连接到 **name**。

对于 **voice**、**fax** 和 **modem** 表，电话号码都用作主键。**address** 表包含特殊列（**id_num**），该列除了作为主键没有其它用途。这样做是因为如果 **id_num** 不存在，则所有其它列将必须一起作为一个组合键使用，以确保不会存在重复的主键。将所有列用作主键将是非常低效和混乱的。

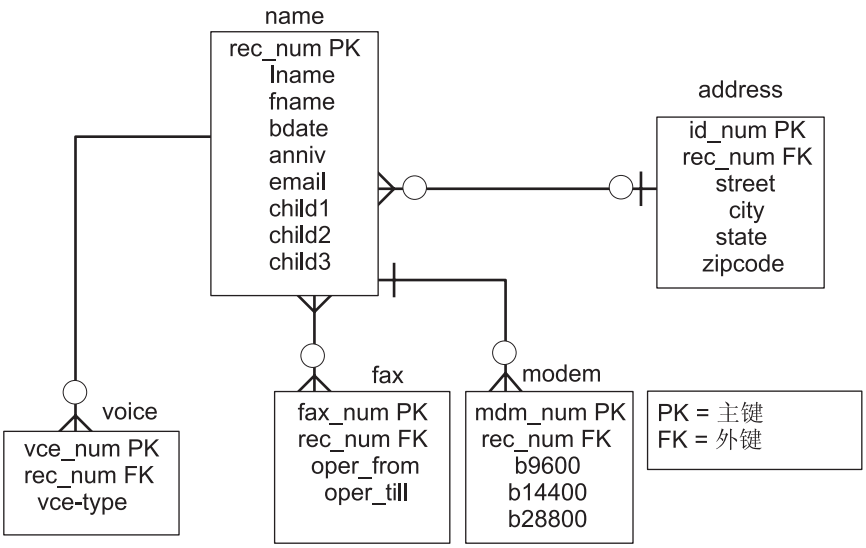


图 2-17. 添加了主键和外键的电话号码簿图

解析关系

良好数据模型的目标是创建数据库服务器能够快速访问的结构。要进一步改进电话号码簿数据模型，您可以解析关系并将数据模型规范化。本节阐述如何以及为何要解析数据库关系。第 2-24 页的『将数据模型规范化』讨论了将数据模型规范化。

解析 m:n 关系

多对多（m:n）关系使模型和应用程序开发过程更加复杂和容易令人混淆。解析 m:n 关系的关键是将两个实体分开并用第三个*相交*实体来在它们之间创建两个一对多（1:n）关系。相交实体通常包含来自两个相连接的实体的属性。

要解析 m:n 关系，请再次分析商业规则。您是否已经准确地对关系制图？如第 2-21 页的图 2-17 所示，电话号码簿示例在 *name* 与 *fax* 实体之间具有 m:n 关系。商业规则指出 “一个人可以有零个、一个或许多个传真号码； *一个传真号码可以用于若干个人*”。根据我们先前为 *voice* 实体选择的作为主键的内容，存在 m:n 关系。

fax 实体中存在一个问题，其原因在于电话号码（已将其指定为主键）可以在 *fax* 实体中出现多次；这违反了主键的条件。记住，主键必须是唯一的。

要解析这种 m:n 关系，可以在 *name* 与 *fax* 实体之间添加相交实体，如图 2-18 所示。新的相交实体 *faxname* 包含两个属性: **fax_num** 和 **rec_num**。此实体的主键是这两个属性的组合。从个别来看，每个属性都是一个外键，它引用作为其来源的表。由于一个名称可以与许多个传真号码相关联，并且在另一个方向上每个 **faxname** 组合可以与一个 **rec_num** 相关联，所以 **name** 与 **faxname** 表之间的关系是 1:n 关系。由于每个号码可以与许多个 **faxname** 组合相关联，所以 **fax** 与 **faxname** 表之间的关系是 1:n 关系。

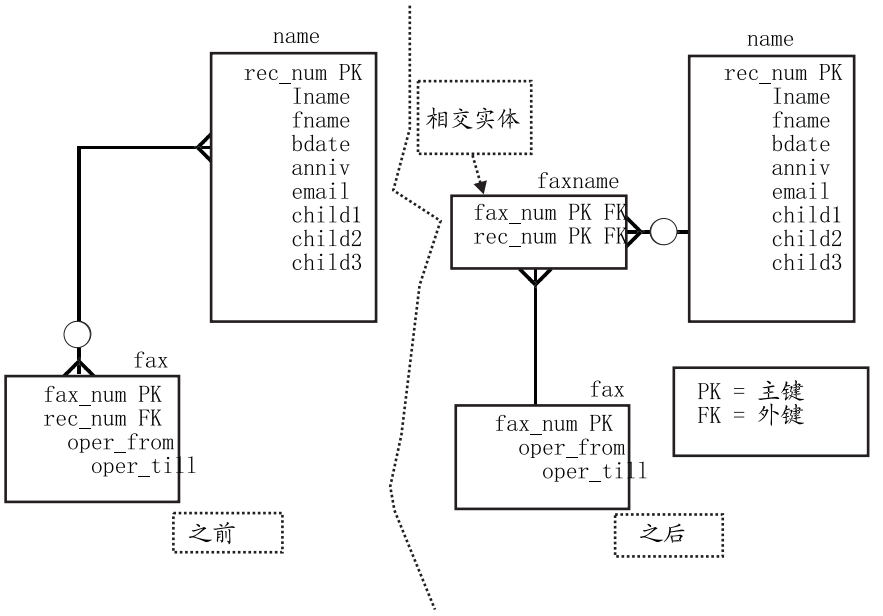


图 2-18. 解析多对多（m:n）关系

解析其它特殊关系

您可能会遇到其它可能会导致数据库无法平稳运行的特殊关系。以下列表显示了这些关系:

- 复杂关系
- 递归关系
- 冗余关系

*复杂*关系是三个或更多个实体之间的关联。要使该关系存在，所有实体都必须存在。为了降低这种复杂性，请将所有复杂关系重新分类为通过二元关系与每个原始实体相关的实体。

*递归*关系是同一实体类型的多个出现之间的关联。这些类型的关系不会经常出现。递归关系的示例包括物料单（部件由子部件组成）和组织结构（雇员管理其他雇员）。您可以选择不解析递归关系。关于递归关系的扩展示例，请参阅*《IBM Informix: SQL 教程指南》*。

当两个或更多个关系表示同一概念时，存在*冗余*关系。冗余关系提高了数据模型的复杂程度，并会导致开发者不正确地在模型中放置属性。冗余关系可能会作为 E-R 图中的重复条目出现。例如: 可能有两个包含相同属性的实体。要解析冗余关

系，请复查数据模型。有没有多个包含相同属性的实体？可能需要向模型添加一个实体来解决冗余性。*《IBM Informix: 性能指南》* 讨论了与数据模型中的冗余性相关的附加主题。

将数据模型规范化

本章中的电话号码簿示例似乎是一个不错的模型。此刻，可以在数据库中实现这个模型，但此示例以后可能会导致应用程序开发和数据处理操作出现问题。*规范化*是一种正式的应用一组规则以使属性与实体相关联的方法。

将数据模型规范化时，可以实现下列目标。您可以:

- 使设计具有更大的灵活性。
- 确保将属性置于正确的表中。
- 降低数据冗余度。
- 提高程序员的效率。
- 降低应用程序维护成本。
- 使数据结构的稳定性达到最高程度。

规范化由若干个步骤组成，用于将实体简化为更合意的物理属性。这些步骤称为规范化规则，也称为*范式*。存在若干个范式；本章讨论前三个范式。每个范式都对数据比上一个范式更具约束性。因此，在可以获得第二范式之前必须先获得第一范式，并且，在可以获得第三范式之前必须先获得第二范式。

第一范式

如果实体不包含重复组，则它满足第一范式。就关系而言，如果表不包含重复列，则它满足第一范式。重复列会降低数据的灵活性、浪费磁盘空间和导致更难以搜索数据。在图 2-19 中的电话号码簿示例中，看上去 **name** 表包含重复列 *child1*、*child2* 和 *child3*。

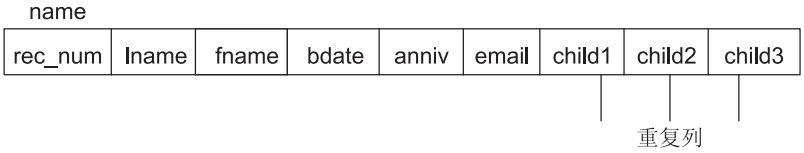


图 2-19. 规范化之前的 name 实体

您可以在当前表中发现一些问题。这个表总是在磁盘上为三个子女记录保留空间，而无论该人是否有子女。可以记录的最大子女数是 3，但您的一些熟人可能有四个或更多的子女。要查找特定的子女，就必须在每一行中搜索全部三个列。

要消除重复列并使该表满足第一范式，请将该表分为两个表，如图 2-20 所示。将重复列放到其中一个表中。两个表之间的关联是通过主键与外键的组合建立的。由于 **name** 表中不存在关联就不能存在子女，所以可使用外键 **rec_num** 来引用 **name** 表。

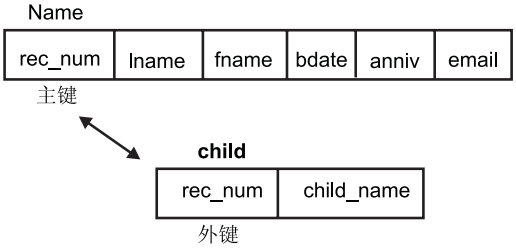


图 2-20. Name 实体达到第一范式

现在，检查第 2-21 页的图 2-17，了解有没有不满足第一范式的组。由于认为 **b9600**、**b14400** 和 **b28800** 列是重复列，所以 *name-modem* 关系不满足第一范式。向 **modem** 表添加名为 **b_type** 的新属性以包含 **b9600**、**b14400** 和 **b28800** 的出现。图 2-21 显示了通过第一范式规范化的数据模型。

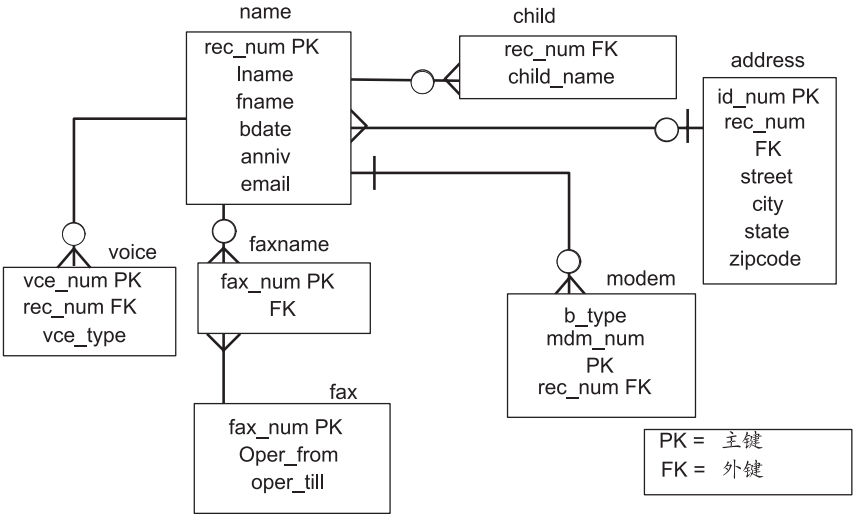


图 2-21. 个人电话号码簿的数据模型

第二范式

如果实体的所有属性都依赖于整个（主）键，则该实体满足第二范式。就关系而言，一个表的每个列都必须*在功能上依赖于*该表的整个主键。功能相关性指示两个不同的列中的值之间存在链接。

如果某个属性的值*依赖于*某列，则当该列中的值更改时，该属性的值必须更改。该属性是该列的函数。下列说明使这一点更为明确:

- 如果该表具有一个单列主键，则该属性必须依赖于该主键。
- 如果该表具有组合主键，则该属性必须依赖于该主键的全部各列中的作为一个整体的值，而不是依赖于那些列的其中一列或其中某些列。
- 如果该属性还依赖于其它列，则它们必须是候选键的列；也就是在每一行中都唯一的列。

如果不将模型转换为第二范式，则有数据冗余和难以更改数据的风险。要将第一范式表转换为第二范式表，请除去不依赖于主键的列。

第三范式

如果实体满足第二范式并且它的所有属性都不以传递方式依赖于主键，则该实体满足第三范式。*传递相关性*表示描述符键属性不仅依赖于整个主键，还依赖于其它描述符键属性，那些描述符键属性又依赖于主键。就 SQL 而言，第三范式表示表中没有任何列依赖于某个描述符列，该描述符列又依赖于主键。

要转换为第三范式，请除去那些依赖于其它描述符键属性的属性。

规范化规则总结

本节讨论了下列范式:

- 第一范式: 如果表不包含重复列，则它满足第一范式。
- 第二范式: 如果表满足第一范式并且只包含依赖于整个（主）键的列，则该表满足第二范式。
- 第三范式: 如果表满足第二范式并且只包含以非传递方式依赖于主键的列，则该表满足第三范式。

按照 E. F. Codd（关系数据库的发明者）的规定，当遵循这些规则时，模型的表满足第三范式。当表不满足第三范式时，模型中会存在冗余数据或在尝试更新表时会发生问题。

如果在模型中找不到放置遵循这些规则的属性的位置，则可能是发生了下列其中一个错误:

- 该属性不具有良好的定义。

- 该属性是派生的，不是直接的。
- 该属性实际上是实体或关系。
- 模型中缺少某些实体或关系。

第 3 章 选择数据类型

在本章中

准备数据模型之后，必须将它作为数据库和表来实现。要实现数据模型，首先为每个列定义域或一组数据值。本章讨论为了定义列数据类型和约束而必须作出的决定。

第二个步骤使用 CREATE DATABASE 和 CREATE TABLE 语句来实现模型并用数据填充表，具体讨论见第 4 章。

定义域

要完成第 2 章描述的数据模型，必须为每个列定义域。列的域描述约束并标识属性（列）可以具有的有效值的集合。

域的用途是保护模型中的数据的*语义完整性*；也就是说，确保它切合实际地反映事实。如果能够用名称来替换电话号码或者可以在只有整数才是有效值的位置中输入小数，则数据模型的完整性是有风险的。

要定义域，请指定在数据值可以作为域的一部分之前必须满足的*约束*。要指定列域，请使用下列约束:

- 数据类型
- 缺省值
- 检查约束
- 引用约束

数据类型

对任何列的第一个约束都是该列的数据类型中隐式的约束。选择数据类型时，您就对列进行了约束，以使其只能包含该数据类型能够表示的值。

每种数据类型都表示特定类型的信息，而不表示其它类型的信息。列的正确数据类型是这样的: 它能够表示所有对该列正确的数据值，但是表示尽可能少的对该列不正确的值。

本章描述内置数据类型。

Dynamic Server

有关 Dynamic Server 支持的扩展数据类型的信息，请参阅第 8-1 页的第 8 章，『在 Dynamic Server 中创建和使用扩展数据类型』。

Dynamic Server 结束

选择数据类型

表中的每一列都必须具有数据类型。由于下列原因，数据类型的选择至关重要:

- 它建立该列可以存储的有效数据项的集合。
- 它确定可以对数据执行的操作的类型。

  例如: 不能对使用字符数据类型定义的列应用聚集函数，如 SUM。

- 它确定每个数据项在磁盘上占用多少空间。

  容纳数据项所需的空间对于小型表的重要程度与对于带有数十万行的表的重要程度是不相同的。当表的行数有如此之多时，4 字节与 8 字节数据类型之间的差别可能极为重要。

第 3-4 页的图 3-1 显示了决策树，此决策树概述了在内置数据类型之间所作的选择。下列各节对这些选择作了说明。

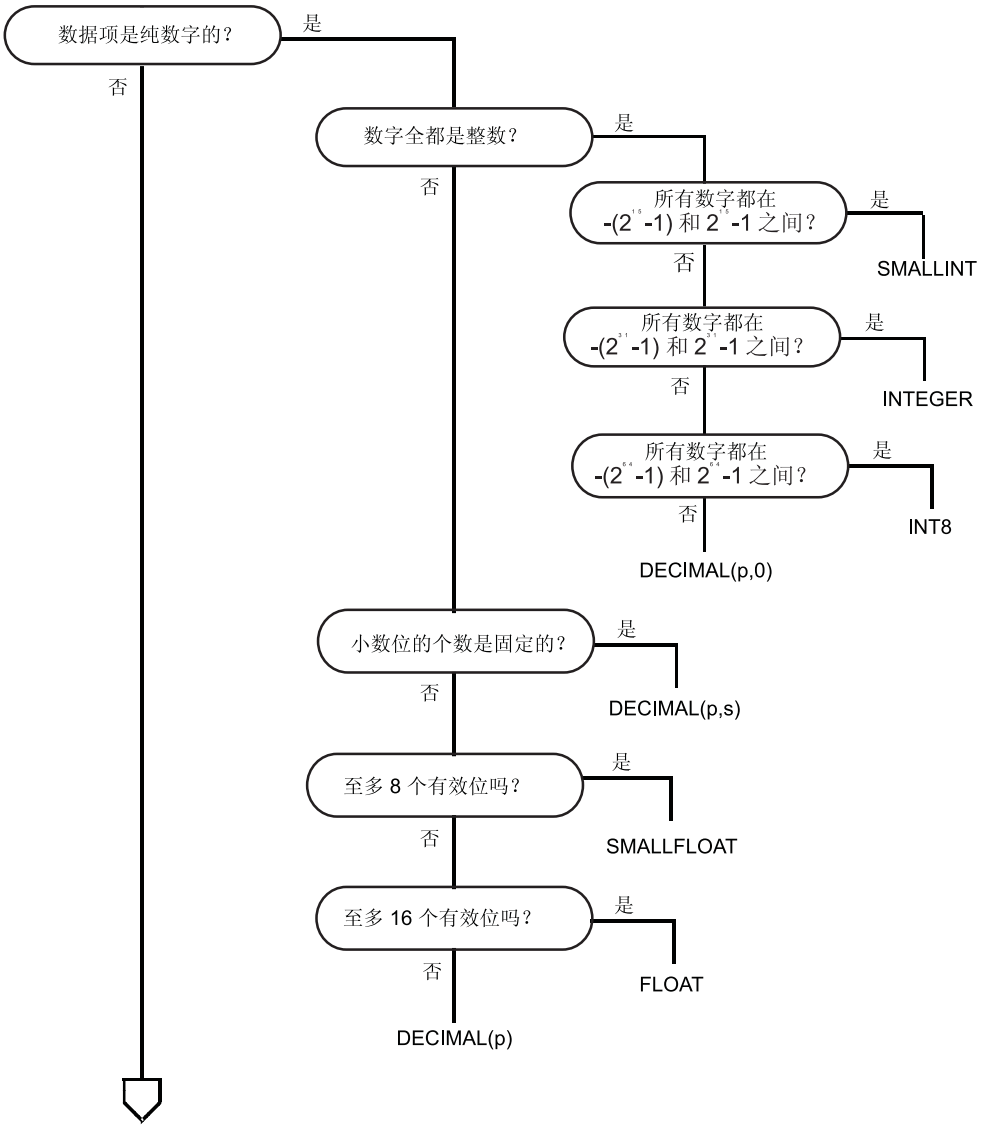


图 3-1. 选择数据类型 (1/2)

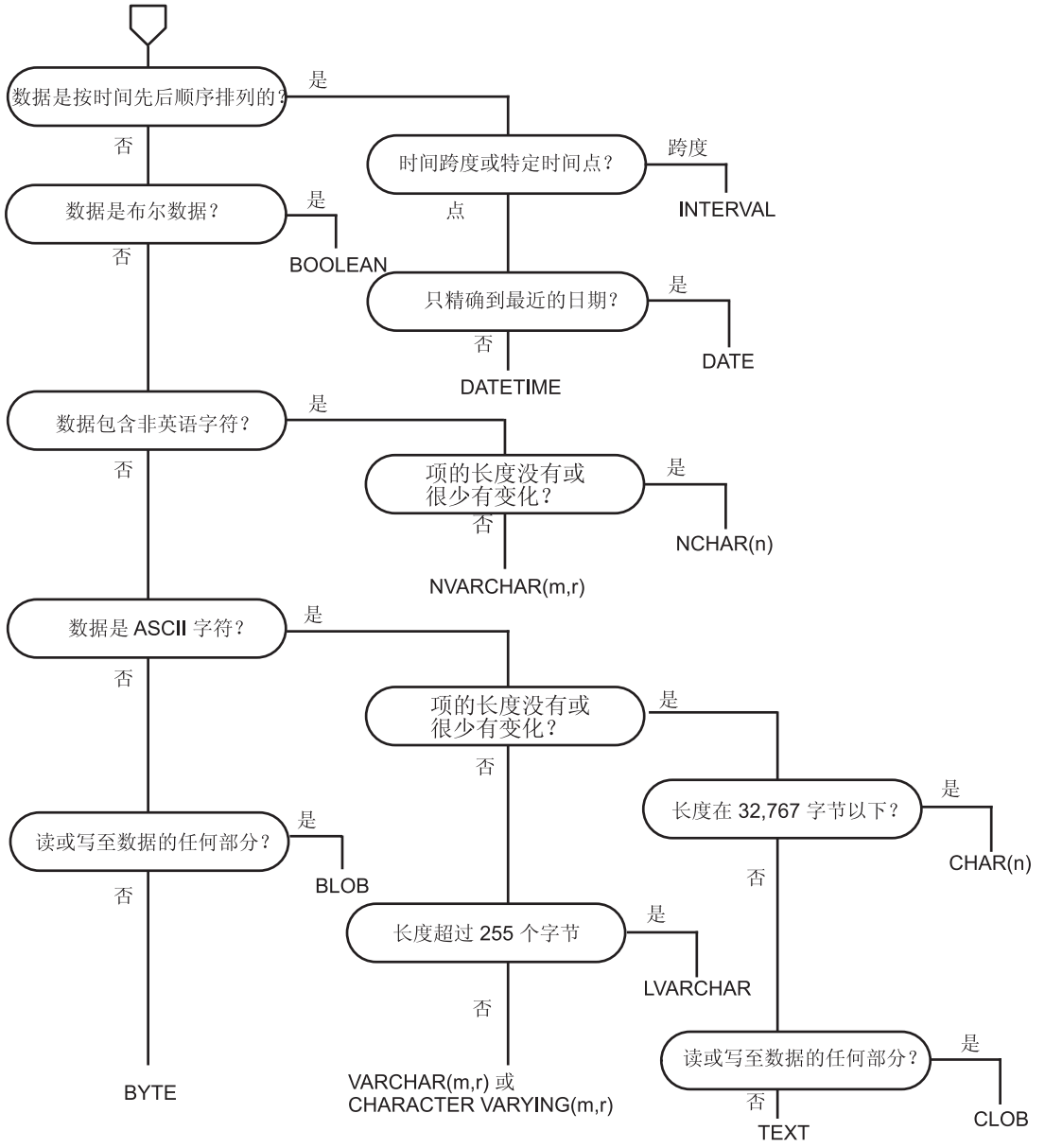


图 3-1. 选择数据类型 (2/2)

数字类型

一些数字数据类型最适合用于计数器和代码，一些最适合用于工程数量，一些最适合用于货币。

计数器和代码：INTEGER、SMALLINT 和 INT8

INTEGER 和 SMALLINT 数据类型存放小整数。当您事先知道要存储的最大值和最小值时，这两种数据类型适合用于包含计数、序号、数字标识代码或任何范围的整数的列。

这两种数据类型都是作为带符号二进制整数存储的。 INTEGER 值有 32 位，可以表示从 $-2^{31}-1$ 到 $2^{31}-1$ 的整数。

SMALLINT 值只有 16 位。它们可以表示从 –32767 到 32767 的整数。

INT 和 SMALLINT 数据类型具有下列优点:

- 它们占用很少的空间（对于 SMALLINT，每个值占用 2 个字节，对于 INTEGER，每个值占用 4 个字节）。
- 可以对它们执行算术表达式（如 SUM 和 MAX）和排序比较。

使用 INTEGER 和 SMALLINT 的缺点是它们可以存储的值的范围有限。数据库服务器不能存储超出整数容量的值。当然，当您知道要存储的最大值和最小值时，这样的超限不是问题。

如果需要存储将填满 INTEGER 的更大范围的值，则可以使用 INT8。

INT8 数据类型具有下列优点:

- 可以容纳大范围的值。（范围从 $-(2^{63}-1)$ 到 $2^{63}-1$ 的整数。）
- 可以对它们执行算术表达式（如 SUM 和 MAX）和排序比较。

使用 INT8 数据类型的缺点在于其使用的磁盘空间大于 INTEGER。IBM Informix Extended Parallel Server（XPS）使用 8 个字节的磁盘空间来存储 INT8，IBM Informix Dynamic Server（IDS）则使用 10 个字节。

自动序列：SERIAL 和 SERIAL8

SERIAL 数据类型具有带有特殊功能的非零正 INTEGER 范围。同样，SERIAL8 数据类型具有带有特殊功能的非零正 INT8 范围。每当将新行插入表中时，数据库服务器都将自动生成 SERIAL 或 SERIAL8 列的新值。

一个表不能有多个 SERIAL 和多个 SERIAL8 列。因为数据库服务器将生成值，所以新行中的序列值总是不相同的，即使多个用户同时添加行亦如此。由于普通程序很难在那些情况下生成唯一的数字代码，所以此服务很有用。（然而 Dynamic

Server 也支持顺序对象，顺序对象也可通过 CURRVAL 和 NEXTVAL 运算符支持该功能。）有关顺序对象的更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE SEQUENCE。）

SERIAL 数据类型最多可以生成 2^{31}–1 个正整数。因此，数据库服务器将使用所有的正数序列号，直到它在表中插入 2^{31}–1 行为止。然而，对于大多数用户来说，并不会发生用完正数序列号的问题，这是因为单个应用程序需要在 68 年内每秒插入一行或者 68 个应用程序需要在一年内每秒插入一行才能用完正数序列号。但是，如果已使用了所有的正数序列号，则数据库服务器将回绕并开始生成以 1 开始的整数值。

SERIAL8 数据类型最多可以生成 2^{63}–1 个正整数。利用合理的起始值，实际上不可能导致 SERIAL8 值在插入期间回绕。

对于 SERIAL 和 SERIAL8 数据类型，所生成的数字的序列总是递增的。从表中删除行之后，它们的序列号将不重复使用。根据 SERIAL 或 SERIAL8 列进行排序的行是按它们的创建顺序返回的。

可以在 CREATE TABLE 语句中指定 SERIAL 或 SERIAL8 列的初始值。这使得在不同的表中生成系统指定的键的不同子序列成为可能。**stores_demo** 数据库使用了此技术。在 **stores_demo** 中，客户号从 101 开始，而订单号从 1001 开始。只要这间小公司注册的客户数不超过 899 个，所有客户号就都是三位的，并且订单号都是四位的。

SERIAL 或 SERIAL8 列不会自动成为唯一列。如果要完全确保不出现重复的序列号，则必须应用唯一约束（请参阅第 4-4 页的『使用 CREATE TABLE』）。如果使用 DB–Access 中的交互式模式编辑器来定义表，则将自动对任何 SERIAL 或 SERIAL8 列应用唯一约束。

SERIAL 和 SERIAL8 数据类型具有下列优点:

- 它们提供了一种很方便的方法来生成系统指定的键。
- 它们生成唯一的数字代码，即使多个用户更新表时亦如此。
- 不同的表可以使用不同范围的数字。

SERIAL 和 SERIAL8 数据类型具有下列缺点:

- 在一个表中，只允许一个 SERIAL 或 SERIAL8 列。
- 它们只能生成任意数。

改变下一个 SERIAL 或 SERIAL8 数: 数据库服务器在创建 SERIAL 或 SERIAL8 列时设置该列的起始值（请参阅第 4-4 页的『使用 CREATE TABLE』）。以后，可以使用 ALTER TABLE 语句来复位下一个值，即用于所插入的下一行的值。

可以将下一个值设置为任何大于当前最大值的值。这样做会在序列中创建间隔。

如果尝试将下一个值设置为小于列中当前的最大值的值，则虽然不会有错误但是值并没有得到设置。允许下一个值设置为小于列中的部分值，会在某些情况下导致重复值，因此这是不允许的。

近似数: FLOAT 和 SMALLFLOAT

在科学、工程和统计应用程序中，通常知道数字只具有少数几位的准确度，并且数字的量值与它的精确位数一样重要。

浮点数据类型就是为这些类型的应用程序设计的。它们可以表示任何具有大范围量值（从无限大到极微小）的数字数量，无论是小数还是整数。它们可以很容易地表示从地球到太阳的平均距离（1.5×10^{11} 米）或普朗克常量（6.626×10^{-34} 焦耳秒）。例如:

```
CREATE TABLE t1 (f FLOAT);
INSERT INTO t1 VALUES (0.00000000000000000000000000000000000001);
INSERT INTO t1 VALUES (1.5e11);
INSERT INTO t1 VALUES (6.626196e-34);
```

存在两种大小的浮点数据类型。FLOAT 类型是双精度二进制浮点数，就象计算机上 C 语言中所实现的那样。FLOAT 数据类型值通常占用 8 个字节。SMALLFLOAT（也称为 REAL）数据类型是单精度二进制浮点数，通常占用 4 个字节。这两种数据类型之间的主要区别是它们的精度。

浮点数具有下列优点:

- 它们存储非常大和非常小的数字，包括小数。
- 它们使用 4 个或 8 个字节简洁地表示数字。
- 算术函数（如 AVG 和 MIN）和排序比较对这些数据类型有效。

浮点数的主要缺点是它们的精度范围之外的各个位都被视为零。

可调整精度浮点: DECIMAL(p)

在不符合 ANSI 的数据库中，DECIMAL(*p*) 数据类型是与 FLOAT 和 SMALLFLOAT 类似的浮点数据类型。重要区别是指定它保留多少个有效位。写作 *p* 的精度的范围为 1 到 32，也就是从比 SMALLFLOAT 低的精度到 FLOAT 的

精度的两倍。DECIMAL(*p*) 数的量值的范围为 10^{-130} 到 10^{124}。DECIMAL(*p*) 数使用的存储空间取决于它们的精度；它们占用 1 + *p*/2 个字节（如果有必要的话，上舍入到整数）。

然而在符合 ANSI 的数据库中，DECIMAL(*p*) 是小数位为零的定点数据类型，所以如果数据类型有 *p* 或更多有效数字，DECIMAL(*p*) 始终存储精度为 *p* 的整数值。任何小数部分都将截断。

不要将 DECIMAL(*p*) 数据类型与下一节讨论的 DECIMAL(*p,s*) 数据类型混淆。DECIMAL(*p*) 数据类型只指定了精度。

与 FLOAT 相比，DECIMAL(*p*) 数据类型具有下列优点:

- 可以设置精度以适合于应用程序（从近似到精确）。
- 可以精确地表示 32 位的数字。
- 使用的存储器与数字的精度成比例。
- 无论使用什么主机操作系统，每个 Informix 数据库服务器都支持相同的精度和量值范围。

DECIMAL(*p*) 数据类型具有下列缺点:

- 对 DECIMAL(*p*) 值执行的算术运算和排序的性能在一定程度上不如对 FLOAT 值执行的算术运算和排序。
- 许多编程语言不能以它们支持 FLOAT 和 INTEGER 的方式来支持 DECIMAL(*p*) 数据格式。当程序从数据库中抽取 DECIMAL(*p*) 值时，它可能必须要将该值转换为另一种格式才能进行处理。
- DECIMAL(*p*) 数据类型的格式和值取决于数据库是否符合 ANSI。

固定精度数: DECIMAL 和 MONEY

大多数商业应用程序需要存储在小数点的左右两边具有固定位数的数。例如: 美国货币金额写成小数点右边有两位。正常情况下，根据所记录的交易的类型，您还知道左边所需的位数: 个人预算可能是 5 位，小企业可能是 7 位，国家预算可能是 12 或 13 位。

由于小数点固定在特定的位置（而无论数值是多少），所以这些数是*定点数*。DECIMAL(*p,s*) 数据类型设计为存放十进制数。当指定具有此类型的列时，您将其*精度*（*p*）写作它可以存储的总位数（从 1 到 32）。您将其*小数位*（*s*）写作位于小数点右边的那些位的数目。（图 3-2 显示了精度与小数位之间的关系。）小数位可以是零，表示只存储整数。当只存储整数时，DECIMAL(*p,s*) 提供了存储高达 32 位的整数的方法。

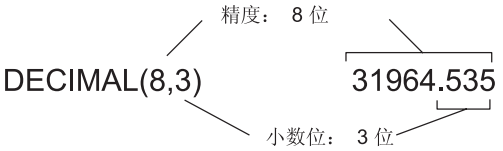


图 3-2. 定点数中的精度与小数位之间的关系

与 DECIMAL(*p*) 数据类型相似，DECIMAL(*p*,*s*) 占用的空间与它的精度成比例。一个值占用 (*p* +3)/2 个字节（如果小数位是偶数）或 (*p* + 4)/2 个字节（如果小数位是奇数），并且上舍入至整数个字节。

除了具有一项额外的特征外，MONEY 类型与 DECIMAL(*p*,*s*) 完全相同。每当数据库服务器将 MONEY 值转换为字符以进行显示时，它将自动包括货币符号。

与 INTEGER 和 FLOAT 相比，DECIMAL(*p*,*s*) 的优点是精度可以高得多（最高 32 位，INTEGER 是 10 位，FLOAT 是 16 位），并且精度和所需的存储器量都可以调整，以适合于应用程序。

DECIMAL(*p*,*s*) 的缺点是算术运算的效率较低，并且许多编程语言不支持此格式的数字。因此，当程序抽取数字时，它通常必须将该数字转换为另一数字格式才能进行处理。

选择货币格式:

Global Language Support

每个国家或地区都有自己的显示货币值的方式。当 Informix 数据库服务器显示 MONEY 值时，它将参考用户指定的货币格式。缺省语言环境指定美国英语货币格式，它具有以下格式:

```
$7,822.45
```

对于非英语语言环境，可使用语言环境文件的 MONETARY 类别来更改当前格式。有关如何使用语言环境的更多信息，请参阅《*IBM Informix: GLS 用户指南*》。

Global Language Support 结束

要定制此货币格式，请适当地选择语言环境或设置 **DBMONEY** 环境变量。有关更多信息，请参阅《*IBM Informix: SQL 参考指南*》。

按时间先后顺序排列的数据类型

按时间先后顺序排列的数据类型记录时间。DATE 数据类型存储日历日期。DATETIME 采用任何程度的精度（从年到秒的小数）记录时间点。INTERVAL 数据类型存储时间跨度；即持续时间。

日历日期: DATE

DATE 数据类型存储日历日期。DATE 值实际上是带符号整数，它的内容是作为从 1899 年 12 月 31 日午夜开始的每一个整天的计数解释的。

DATE 格式有足够的精度来将日期带入遥远的将来（58000 个世纪）。负数 DATE 值是作为纪元日之前的天数计数来解释的；即 DATE 值 -1 表示 1899 年 12 月 30 日这一天。

由于 DATE 值是整数，所以值可以在算术表达式中使用。例如: 可以获取 DATE 列的平均值，也可以向 DATE 列加上 7 或 365。另外，提供了一组丰富的专门用于处理 DATE 值的函数。有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

DATE 数据类型是压缩的，每个项 4 个字节。可以对 DATE 列快速地执行算术函数和比较。

选择日期格式（GLS）: 您可以采用许多种方式来对日期组件加标点和排序。当应用程序显示 DATE 值时，它将参考用户指定的日期格式。缺省语言环境指定具有以下格式的美国英语日期格式:

```
10/25/2001
```

要定制此日期格式，请适当地选择语言环境或设置 **DBDATE** 环境变量。有关更多信息，请参阅《*IBM Informix: SQL 参考指南*》。

对于非缺省语言环境，可以使用 **GL_DATE** 环境变量指定日期格式。有关如何使用语言环境的更多信息，请参阅《*IBM Informix: GLS 用户指南*》。

精确时间点: DATETIME

DATETIME 数据类型存储从公元 1 年开始的时代中的任何时刻。事实上，DATETIME 实际上是 28 种数据类型的一个系列，其中，每种数据类型都具有不同的精度。在定义 DATETIME 列时，指定它的精度。列可以包含*年*、*月*、*日*、*小时*、*分钟*、*秒*和*小数*列表中的任何序列。因此，可以定义只存储年、只存储月和日或者精确到小时甚至精确到毫秒的日期和时间的 DATETIME 列。第 3-12 页的表 3-1 显示 DATETIME 值的大小视精度的不同而具有 2 到 11 个字节。

DATETIME 的优点是它可以存储特定的日期和时间值。与 DATE 列相比，DATETIME 列通常需要更多的存储器空间（这取决于 DATETIME 限定符）。Datetime 还具有不灵活的显示格式。有关如何避开显示格式的信息，请参阅第 3-13 页的『强制 DATETIME 或 INTERVAL 值的格式』。

表 3-1. DATETIME 数据类型的精度

精度	大小*	精度	大小*
年到年	3	天到小时	3
年到月	4	天到分钟	4
年到天	5	天到秒	5
年到小时	6	天到小数（*f*）	5 + *f*/2
年到分钟	7	小时到小时	2
年到秒	8	小时到分钟	3
年到小数（*f*）	8 + *f*/2	小时到秒	4
月到月	2	小时到小数（*f*）	4 + *f*/2
月到天	3	分钟到分钟	2
月到月	4	分钟到秒	3
月到分钟	5	分钟到小数（*f*）	3 + *f*/2
月到秒	6	秒到秒	2
月到小数（*f*）	6 + *f*/2	秒到小数（*f*）	2 + *f*/2
天到天	2	小数到小数（*f*）	1 + *f*/2
* 当 f 是奇数时，将大小取整到下一个整字节			

使用 INTERVAL 的持续时间: INTERVAL 数据类型存储持续时间，即时间长度。两个 DATETIME 值之间的差就是 INTERVAL，它表示那两个值之间的时间跨度。下列示例可能有助于您弄清楚这些差:

- 一个雇员在 1997 年 1 月 21 日开始工作（或者是 DATE 或者是 DATETIME）。
- 她已工作了 254 天（一个 INTERVAL 值，也就是 TODAY 函数与起始 DATE 或 DATETIME 值之间的差）。
- 她每天从 0900 点（一个 DATETIME 值）开始工作。
- 她工作 8 小时（一个 INTERVAL 值），其间 45 分钟用于午餐（另一个 INTERVAL 值）。
- 她的下班时间是 1745 点（她开始工作时的 DATETIME 与两个 INTERVAL 之和）。

与 DATETIME 相似，INTERVAL 是具有不同精度的数据类型的系列。INTERVAL 值可以表示年和月的计数；它也可以表示天、小时、分钟、秒或秒的小数的计数；可能的精度有 18 种。INTERVAL 值的大小的范围是 2 到 12 个字节，这取决于表 3-2 显示的公式。

表 3-2. INTERVAL 数据类型的精度

精度	大小*	精度	大小*
年（*p*）到年	1 + *p*/2	小时（*p*）到分钟	2 + *p*/2
年（*p*）到月	2 + *p*/2	小时（*p*）到秒	3 + *p*/2
月（*p*）到月	1 + *p*/2	小时（*p*）到小数（*f*）	4 + (*p* + *f*)/2
天（*p*）到天	1 + *p*/2	分钟（*p*）到分钟	1 + *p*/2
天（*p*）到小时	2 + *p*/2	分钟（*p*）到秒	2 + *p*/2
天（*p*）到分钟	3 + *p*/2	分钟（*p*）到小数（*f*）	3 + (*p* + *f*)/2
天（*p*）到秒	4 + *p*/2	秒（*p*）到秒	1 + *p*/2
天（*p*）到小数（*f*）	5 + (*p* + *f*)/2	秒（*p*）到小数（*f*）	2 + (*p* + *f*)/2
小时（*p*）到小时	1 + *p*/2	小数到小数（*f*）	1 + *f*/2
* 将小数大小取整到下一个整字节。			

INTERVAL 值可以是负数也可以是正数。可以对它们进行加减运算，也可以通过乘以或除以某个数来将它们缩放。对于 DATE 或 DATETIME，情况并非如此。您可以合理地问道“到 4 月 23 日为止的天数的一半是多少？”，但“4 月 23 日的一半是什么？”却不合理。

强制 DATETIME 或 INTERVAL 值的格式: 数据库服务器总是按 *year-month-day hour:minute:second.fraction* 的顺序显示 INTERVAL 或 DATETIME 值的组件。数据库服务器不参考对操作系统定义的日期格式，当它格式化 DATE 值时它却那样做。

您可以编写以系统定义的格式显示 DATETIME 值的日期部分的 SELECT 语句。其诀窍是使用 **EXTEND** 函数隔离组件字段并通过 **MDY()** 函数（此函数将它们转换为 DATE）传递它们。下列代码显示了一个不完整的示例:

```
SELECT ... MDY (
   EXTEND (DATE_RECEIVED, MONTH TO MONTH),
   EXTEND (DATE_RECEIVED, DAY TO DAY),
   EXTEND (DATE_RECEIVED, YEAR TO YEAR) )
   FROM RECEIPTS ...
```

选择 DATETIME 格式（GLS）

当应用程序显示 DATETIME 值时，它将参考用户指定的 DATETIME 格式。缺省语言环境指定具有以下格式的美国英语 DATETIME 格式:

```
2001-10-25 18:02:13
```

对于非缺省语言环境，可以使用 **GL_DATETIME** 环境变量指定 DATETIME 格式。有关如何使用语言环境的更多信息，请参阅《*IBM Informix: GLS 用户指南*》。

要定制此 DATETIME 格式，请适当地选择语言环境或设置 **GL_DATETIME** 或 **DBTIME** 环境变量。有关这些环境变量的更多信息，请参阅《*IBM Informix: GLS 用户指南*》。

BOOLEAN 数据类型（IDS）

BOOLEAN 数据类型是单字节的数据类型。合法布尔值包括 true（'t'）、false（'f'）或 NULL。值不区分大小写。

可以将 BOOLEAN 列与另一个 BOOLEAN 列或与布尔值作比较。例如: 可使用下列 SELECT 语句:

```
SELECT * FROM sometable WHERE bool_col = 't';
SELECT * FROM sometable WHERE bool_col IS NULL;
```

字符数据类型（GLS）

Informix 数据库服务器支持数种字符数据类型，包括 CHAR、NCHAR 和 NVARCHAR（特殊用途的字符数据类型）。

字符数据: CHAR(n) 和 NCHAR(n)

CHAR(*n*) 数据类型包含 *n* 个字节的序列。这些字符可以是英语与非英语字符的混合，并且可以是单字节字符或多字节字符（亚洲语言字符）。长度 *n* 的范围是 1 到 32767。

每当数据库服务器检索或存储 CHAR(*n*) 值时，它都刚好传送 *n* 个字节。如果插入的值的长度小于 *n*，则数据库服务器用单字节 ASCII 空格字符扩展该值以构成 *n* 个字节。如果插入的值的长度超过 *n* 个字节，则数据库服务器将截断额外的字符，而不返回错误消息。因此，对于 CHAR(*n*) 列或变量，当插入或更新的值的长度超过 *n* 个字节时，不强制执行数据的语义完整性。

CHAR 列中的数据是按代码集顺序排序的。例如: 在 ASCII 代码集中，字符 *a* 的代码集值为 97，*b* 的值为 98，依此类推。数据库服务器以此顺序将 CHAR(*n*) 数据排序。

NCHAR(*n*) 数据类型也包含 *n* 个字节的序列。这些字符可以是英语与非英语字符的混合，并且可以是单字节字符或多字节字符（亚洲语言字符）。长度 *n* 的限制与 CHAR(*n*) 数据类型的限制相同。每当检索或存储 NCHAR(*n*) 值时，刚好传送 *n* 个

字节。如果数据包含多字节字符，则传送的字符数可以小于字节数。如果插入的值的长度小于 *n*，则数据库服务器用空格字符扩展该值以构成 *n* 个字节。

数据库服务器根据语言环境指定的顺序对 NCHAR(*n*) 列中的数据进行排序。例如: 法语语言环境指定将字符 *ê* 排序在 *e* 值之后但在 *f* 值之前。换而言之，法语语言环境规定的排序顺序是 *e*，*ê*，*f*，等等。有关如何使用语言环境的更多信息，请参阅《*IBM Informix: GLS 用户指南*》。

技巧: CHAR(n) 与 NCHAR(n) 数据之间的唯一区别是数据的排序和比较方式不相同。可以在 CHAR(n) 列中存储非英语字符。然而，由于数据库服务器使用代码集顺序来对 CHAR(n) 列执行任何排序或比较，所以可能无法获得具有您所期望的顺序的结果。

CHAR(*n*) 或 NCHAR(*n*) 值可包含制表符和空格，但通常不包含其它不可打印字符。当用 INSERT 或 UPDATE 插入行时，或者在使用实用程序装入行时，输入不可打印字符是没有意义的。然而，当使用嵌入式 SQL 的程序创建行时，程序可插入除空（二进制零）字符以外的任何字符。由于标准程序和实用程序不期望遇到不可打印字符，所以在字符列中存储不可打印字符不是一个好的想法。

CHAR(*n*) 或 NCHAR(*n*) 数据类型的优点是它在所有数据库服务器上都可用。CHAR(*n*) 或 NCHAR(*n*) 的唯一缺点是它具有固定的长度。当行与行的数据值长度变化很大时，将浪费空间。

变长字符串: CHARACTER VARYING(m,r)、VARCHAR(m,r)、NVARCHAR(m,r) 和 LVARCHAR

通常，字符列中的项的长度是不相同的；也就是说，许多项具有平均长度，只有少数项具有最大长度。对于下列每一种数据类型，*m* 表示最大字节数，*r* 表示最小字节数。下列数据类型设计成当您存储此类数据时可节省磁盘空间:

- **CHARACTER VARYING(*m,r*)。** CHARACTER VARYING (*m,r*) 数据类型最多包含 *m* 个字节的序列，至少包含 *r* 个字节的序列。此数据类型是变长字符数据的符合 ANSI 的格式。CHARACTER VARYING (*m,r*) 支持代码集顺序，以便比较它的字符数据。
- **VARCHAR(*m,r*)。** VARCHAR (*m,r*) 是特定于 Informix 的数据类型，用于存储变长字符数据。它在功能上与 CHARACTER VARYING(*m,r*) 相同。
- **NVARCHAR(*m,r*)。** NVARCHAR(*m,r*) 也是特定于 Informix 的数据类型，用于存储变长字符数据。它按语言环境指定的顺序比较字符数据。

Dynamic Server

- **LVARCHAR**。 LVARCHAR 是特定于 Informix 的数据类型，用于存储 1 到 32739 的变长字符数据。 LVARCHAR 支持代码集顺序用于数据比较。

Dynamic Server 结束

技巧: NVARCHAR(*m,r)* 数据与 CHARACTER VARYING(*m,r*) 或 VARCHAR(*m,r*) 数据的数据比较方式有所不同。有关语言环境如何确定代码集和排序顺序的更多信息，请参阅第 3-14 页的『字符数据: CHAR(n) 和 NCHAR(n)』。

在定义这些数据类型的列时，指定 *m* 作为最大字节数。即使插入的值所包含的字节数小于 *m*，数据库服务器也不会（象对 CHAR(*n*) 和 NCHAR(n) 值所做的那样）用单字节空格扩展值。相反，数据库服务器在磁盘上用 1 个字节长度的字段来只存储实际内容。对于已建立索引的列，*m* 的限制是 254 字节，对于未建立索引的列，限制是 255 字节。

第二个参数 *r* 是可选的保留长度，它对字节数设置的限制比存储在磁盘上的值所需要的低。即使值需要的字节数小于 *r*，也仍然分配 *r* 个字节来存放该值。其目的是在更新行时节省时间。（请参阅第 3-16 页的『变长执行时间』。）

与 CHAR(*n*) 数据类型相比，CHARACTER VARYING(*m,r*) 或 VARCHAR(*m,r*) 数据类型的优点如下:

- 当数据项所需的字节数变化非常大，或者当只有少数几个项需要超过平均数的字节数时，能够节省磁盘空间。
- 对压缩程度更高的表执行的查询可以具有更快的执行速度。

与 NCHAR(*n*) 数据类型相比，这些优点也适用于 NVARCHAR(*m,r*) 数据类型。

以下列表描述了使用变长数据类型的缺点:

- 它们不允许长度超过 255 字节。
- 在某些情况下，表的更新速度可能会比较慢。
- 它们并非对所有 Informix 数据库服务器都可用。

变长执行时间

当使用任何 CHARACTER VARYING(*m,r*)、VARCHAR(*m,r*) 或 NVARCHAR(*m,r*) 数据类型时，各个表行具有不断变化的字节数，而不是具有固定字节数。当表行具有不断变化的字节数时，数据库操作的速度会受到影响。

由于可以将更多的行放到一个磁盘页中，所以，比各行具有固定字节数相比，数据库服务器可以使用少一些的磁盘操作来搜索表。因此，可以更快速地执行查询。基于同一原因，插入和删除操作的速度也会稍快。

在更新行时，数据库服务器必须执行的工作量取决于新行中的字节数与旧行中的字节数的比较。如果新行使用相同的字节数或更少，则与定长行相比，执行时间没有显著的不同。然而，如果新行与旧行相比需要多得多的字节数，则数据库服务器可能必须执行数倍的磁盘操作。因此，更新使用 CHARACTER VARYING(*m*,*r*)、VARCHAR(*m*,*r*) 或 NVARCHAR(*m*,*r*) 数据的表有时会比更新定长字段慢。

要减轻这种效果，请将 *r* 指定为能够适合于相当大部分的数据项的字节数。于是，大多数行使用保留字节数，填充操作将只浪费少量空间。仅当将使用保留字节数的值替换为使用超出保留字节数的值时，更新速度才会较慢。

大字符对象: TEXT

TEXT 数据类型存储文本块。此数据类型用于存储独立的文档: 商业表单、程序源、数据文件或备忘录。尽管可以在 TEXT 项中存储任何数据，但 Informix 工具期望 TEXT 项是可打印的，因此请将此数据类型限制为可打印的 ASCII 文本。

Extended Parallel Server

Extended Parallel Server 支持列中的 TEXT 数据类型，但不允许在 blob 空间中存储 TEXT 列或在 SPL 例程中使用 TEXT 值。

Extended Parallel Server 结束

TEXT 值不与包含它们的行存储在一起。它们分配在完整的磁盘页中（通常是在与行分开的区域中）。有关更多信息，请参阅 *《IBM Informix: 管理员指南》*。

与 CHAR(*n*) 和 VARCHAR(*m*,*r*) 相比，TEXT 数据类型的优点是 TEXT 数据项的大小没有限制（存放它的磁盘存储器的容量除外）。TEXT 数据类型的缺点如下:

- 在完整的磁盘页中分配它，因此短项浪费空间。
- 对可以如何在 SQL 语句中使用 TEXT 列有所限制。（有关该限制的更多信息，请参阅第 3-18 页的『使用 TEXT 和 BYTE 数据类型』。）
- 它并非对所有 Informix 数据库服务器都可用。

二进制对象: BYTE

BYTE 数据类型用来存放程序可以生成的任何数据: 图形图像、程序对象文件以及由任何字处理器保存的文档或电子表格。数据库服务器允许在 BYTE 列中存放任

何类型的具有任何长度的数据。

Extended Parallel Server

Extended Parallel Server 支持列中的 BYTE 数据类型，但不允许在 blob 空间中存储 BYTE 列或在 SPL 例程中使用 BYTE 值。

Extended Parallel Server 结束

就象 TEXT 一样，BYTE 数据项通常存储在与普通行数据分开的磁盘区域中的完整磁盘页中。

与 TEXT 或 CHAR(*n*) 相反，BYTE 数据类型的优点是它接受任何数据。它的缺点与 TEXT 数据类型的那些缺点相同。

使用 TEXT 和 BYTE 数据类型

数据库服务器存储和检索 TEXT 和 BYTE 列。要访存和存储 TEXT 或 BYTE 值，您通常使用采用支持嵌入式 SQL 的语言（如 IBM Informix ESQL/C）编写的程序。在这样的程序中，可以按照类似于读写顺序文件的方式来访存、插入或更新 TEXT 或 BYTE 值。

在任何 SQL 语句中（无论是交互式的还是编程的），都不能以下列方式使用 TEXT 或 BYTE 列:

- 在算术或布尔表达式中
- 在 GROUP BY 或 ORDER BY 子句中
- 在 UNIQUE 测试中
- 用于建立索引（无论是独自作为索引还是作为组合索引的一部分）

在以交互方式输入的或位于表单或报告中的 SELECT 语句中，可以对 TEXT 或 BYTE 值执行下列操作:

- 选择列名（可选择通过下标来抽取它的某部分）。
- 使用 LENGTH(*column_name*) 来返回列的长度。
- 使用 IS [NOT] NULL 谓词来测试列。

在交互式 INSERT 语句中，可使用 VALUES 子句来插入 TEXT 或 BYTE 值，但唯一可以对该列指定的值是空。然而，可使用 SELECT 格式的 INSERT 语句来从另一个表复制 TEXT 或 BYTE 值。

在交互式 UPDATE 语句中，可将 TEXT 或 BYTE 列更新为空或更新为返回 TEXT 或 BYTE 列的子查询。

更改数据类型

在构建表之后，可使用 ALTER TABLE 语句来更改对列指定的数据类型。尽管这样的改变有时是必需的，但您应该避免进行这样的更改，原因如下:

- 要更改数据类型，数据库服务器必须复制并重新构建表。对于大型的表，复制和重新构建可能需要大量的时间和磁盘空间。
- 一些数据类型更改会导致丢失信息。例如: 当将列由较长的字符类型更改为较短的字符类型时，长值将被截断；当更改为精度较低的数字类型时，将截断低位数位。
- 可能还必须更改现有的程序、表单、报告和存储查询。

空值

在大多数情况下，表中的列可以包含空值。空值表示该列的值可能是未知的或不适用的。例如: 在第 2 章 中的电话号码簿示例中，**name** 表的 **anniv** 列可以包含空值；如果您不知道某个人的周年纪念日，可以不指定。请不要将空值与零值或空白值混淆。例如: 以下语句将一行插入到 **stores_demo** 数据库的 **manufact** 表中，并指定 **lead_time** 列的值为空:

```
INSERT INTO manufact VALUES ('DRM', 'Drumm', NULL)
```

Dynamic Server

集合列不能包含空元素。第 8 章描述了集合数据类型。

Dynamic Server 结束

缺省值

缺省值是当 INSERT 语句中没有指定显式的值时将插入到列中的值。缺省值可以是您定义的文字字符串或下列其中一个 SQL 常量表达式:

- USER
- CURRENT
- TODAY
- DBSERVERNAME

并非所有列都需要缺省值，但是，当您使用数据模型时，您可能会发现使用缺省值能够缩短数据输入时间或预防数据输入错误的情况。例如: 电话号码簿模型有一个 **state** 列。查看此列的数据时，您发现超过 50% 的地址将 California 列示为州。为了节省时间，请指定字符串 CA 作为 **state** 列的缺省值。

检查约束

检查约束指定执行 INSERT 或 UPDATE 语句期间在可以将数据赋给列之前数据值必须满足的条件或要求。在插入或更新期间，如果某行对表上定义的任何检查约束求值为 *false*，则数据库服务器返回错误。然而，当检查约束求值为 NULL 时，数据库服务器不报告错误或拒绝记录。因此，创建表时，您可能想同时使用检查约束和 NOT NULL 约束。

要定义约束，请使用 CREATE TABLE 或 ALTER TABLE 语句。例如：以下要求将整数域的值约束为位于特定范围之内：

```
Customer_Number >= 50000 AND Customer_Number <= 99999
```

要表达对基于字符的域的约束，请使用 MATCHES 谓词和它支持的正则表达式语法。例如：以下约束将电话域限制为具有美国本地电话号码格式：

```
vce_num MATCHES '[2-9][2-9][0-9]-[0-9][0-9][0-9][0-9]'
```

有关检查约束的其它信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE TABLE 和 ALTER TABLE 语句。

引用约束

可以在每个表中标识主键和外键以对列设置引用约束。第 2-1 页的第 2 章，『构建关系数据模型』讨论了如何标识这些键。

当您尝试为主键和外键挑选列时，几乎所有数据类型组合都必须匹配。例如：如果将主键定义为具有 CHAR 数据类型，则还必须将外键定义为具有 CHAR 数据类型。然而，当您对一个表中的主键指定 SERIAL 数据类型时，您可以对该关系的外键指定 INTEGER。同样，当您对一个表中的主键指定 SERIAL8 数据类型时，您可以对该关系的外键指定 INT8。可以在关系中混合使用的数据类型组合只有：

- SERIAL 和 INTEGER
- SERIAL8 和 INT8

有关如何创建具有引用约束的表的信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE TABLE 和 ALTER TABLE 语句。

第 4 章 实现关系数据模型

在本章中

本章阐述如何使用 SQL 语法来实现第 2 章描述的数据模型。换而言之，本章阐述如何创建数据库和表以及如何使用数据来填充表。本章还讨论了数据库记录选项、表同义词和命令脚本。

创建数据库

现在您已准备好将数据模型创建为数据库中的表了。可以使用 CREATE DATABASE、CREATE TABLE 和 CREATE INDEX 语句来执行此操作。*《IBM Informix: SQL 指南: 语法》*描述了这些语句的语法。本节讨论如何使用 CREATE DATABASE 和 CREATE TABLE 语句来实现数据模型。

记住，电话号码簿数据模型只是用于进行举例说明。出于进行举例说明的目的，已将其转换为 SQL 语句。

您可能必须多次创建同一个数据库模型。可以将创建模型的语句存储下来并在以后再次执行那些语句。有关更多信息，请参阅第 4-8 页的『使用命令脚本』。

当表已存在时，必须使用数据行对它们进行填充。可以通过实用程序或定制编程手工完成此操作。

使用 CREATE DATABASE

数据库是存放数据模型的所有部件的容器。这些部件不仅包括表，还包括与数据库相关联的视图、索引、同义词和其它对象。必须先创建数据库，然后才可以创建任何其它内容。

当数据库服务器创建服务器时，它存储从它的系统目录中的 **DB_LOCALE** 环境变量派生的数据库语言环境。此语言环境确定数据库服务器如何解释存储在数据库中的字符数据。缺省情况下，数据库语言环境是使用 ISO8859-1 代码集的美国英语语言环境。有关如何使用备用语言环境的信息，请参阅《*IBM Informix: GLS 用户指南*》。

当数据库服务器创建数据库时，它将建立显示数据库的存在情况及其日志记录方式的记录。因为数据库服务器直接管理磁盘空间，所以这些记录对操作系统命令不可视。

避免名称冲突

通常，只有一个数据库服务器副本在计算机上运行，数据库服务器管理属于该计算机的所有用户的数据库。数据库服务器只保留一份数据库名的列表。数据库的名称必须与数据库服务器管理的任何其它数据库的名称不同。(运行数据库服务器的多个副本是有可能的。例如: 可以创建数据库服务器的多个副本，以创建一个安全的环境来在与运营数据分开的情况下进行测试。在这种情况下，确保创建数据库时使用正确的数据库服务器，并在以后存取数据库时也使用正确的数据库服务器。)

选择数据库空间

数据库服务器允许您在特定的*数据库空间*中创建数据库。数据库空间是磁盘存储器的命名区域。请向数据库服务器管理员询问是否应使用特定的数据库空间。可以将数据库放在独立的数据库空间中以将它与其它数据库隔离开，也可将它放在特定的磁盘设备上。有关数据库空间以及它们与磁盘设备的关系的信息，请参阅《*IBM Informix: 管理员指南*》。有关如何跨多个数据库空间将数据库的表分段的信息，请参阅第 11-1 页的第 11 章，『构建维数据模型』。

一些数据库空间是*镜像的*(在两个磁盘设备上重复以获得高可靠性)。如果数据库的内容非常重要，则可以将它放在镜像的数据库空间中。

选择记录类型

要指定日志记录数据库或非日志记录数据库，请使用 CREATE DATABASE 语句。数据库服务器为事务日志记录提供了下列选项:

- **完全不进行记录** 不建议使用此选项。如果由于硬件故障而丢失数据库，则将丢失自上次备份之后的所有数据改变。

  ```
  CREATE DATABASE db_with_no_log
  ```

  当不选择记录时，数据库中不允许 BEGIN WORK 以及其它与事务处理相关的 SQL 语句。这种情况将影响到使用数据库的程序的逻辑。

  Extended Parallel Server 不支持非日志记录数据库。但是，数据库服务器却支持非日志记录表。有关更多信息，请参阅第 7-3 页的『使用分布式查询配制数据库服务器』。

- **常规（非缓冲型）记录** 此选项对于大多数数据库而言都最为适合。发生故障时，将只丢失未提交的事务。

  ```
  CREATE DATABASE a_logged_db WITH LOG
  ```

- **缓冲型日志记录**即使丢失数据库，也只丢失少量最新改变，也可能不丢失任何改变。这种小风险的回报是改变操作期间的性能略有改进。

  ```
  CREATE DATABASE buf_log_db WITH BUFFERED LOG
  ```

  缓冲型日志记录最适合于频繁更新（因而更新速度至关重要）的数据库，但发生故障时可以根据其它数据重新创建更新。使用 SET LOG 语句来在缓冲型日志记录与常规记录之间进行切换。

- **符合 ANSI 的记录** 此记录与常规记录相同，但还强制执行用于事务处理的 ANSI 规则。有关更多信息，参阅第 1-2 页的『使用符合 ANSI 的数据库』。

  ```
  CREATE DATABASE std_rules_db WITH LOG MODE ANSI
  ```

  ANSI SQL 的设计禁止使用缓冲型日志记录。在创建符合 ANSI 的数据库时，不能关闭事务日志记录。

对于不符合 ANSI 的 Dynamic Server 数据库，数据库服务器管理员（DBA）可以打开和关闭事务日志记录，或由缓冲型日志记录更改为非缓冲型日志记录。例如: 可以在插入大量新行之前关闭记录。

可以使用 IBM Informix Server Administrator（ISA）或 **ondblog** 和 **ontape** 实用程序来更改记录状态或缓冲方式。有关这些工具的信息，请参阅《*IBM Informix: Dynamic Server 管理员指南*》。也可以使用 SET LOG 语句来在缓冲型日志记录与非缓冲型日志记录之间进行更改。有关 SET LOG 的信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

使用 CREATE TABLE

使用 CREATE TABLE 语句来创建您在数据模型中设计的每个表。此语句具有复杂的格式，但它基本上是表列的列表。对于每个列，提供下列信息:

- 列的名称
- 数据类型（来自您创建的域列表）

该语句还可包含下列其中一个或多个约束:

- 主键约束
- 外键约束
- NOT NULL 约束
- 唯一约束
- 缺省约束
- 检查约束

简而言之，CREATE TABLE 语句是您在第 2-25 页的图 2-21 的数据模型图中绘制的表的映像。以下示例显示了电话号码簿数据模型的语句:

```
CREATE TABLE name
   (
   rec_num  SERIAL PRIMARY KEY,
   lname    CHAR(20),
   fname    CHAR(20),
   bdate    DATE,
   anniv    DATE,
   email    VARCHAR(25)
   );

CREATE TABLE child
   (
   child    CHAR(20),
   rec_num    INT,
   FOREIGN KEY (rec_num) REFERENCES NAME (rec_num)
   );

CREATE TABLE address
   (
   id_num    SERIAL PRIMARY KEY,
   rec_num    INT,
   street    VARCHAR (50,20),
   city      VARCHAR (40,10),
   state     CHAR(5) DEFAULT 'CA',
   zipcode   CHAR(10),
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );

CREATE TABLE voice
   (
   vce_num    CHAR(13) PRIMARY KEY,
```

```
   vce_type   CHAR(10),
   rec_num    INT,
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );

CREATE TABLE fax
   (
   fax_num    CHAR(13),
   oper_from  DATETIME HOUR TO MINUTE,
   oper_till  DATETIME HOUR TO MINUTE,
   PRIMARY KEY (fax_num)
   );

CREATE TABLE faxname
   (
   fax_num    CHAR(13),
   rec_num    INT,
   PRIMARY KEY (fax_num, rec_num),
   FOREIGN KEY (fax_num) REFERENCES fax (fax_num),
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );

CREATE TABLE modem
   (
   mdm_num    CHAR(13) PRIMARY KEY,
   rec_num    INT,
   b_type     CHAR(5),
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );
```

在上述每个示例中，由于 CREATE TABLE 语句没有指定存储器选项，所以表数据存储在您对数据库指定的那个数据库空间中。可以对表指定与数据库的存储位置不同的数据库空间，也可以将表分段存放到多个数据库空间中。有关 Informix 数据库服务器支持的不同存储器选项的信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE TABLE 语句。下一节阐述一种将表分段存放到多个数据库空间中的方法。

创建分段表

要在表级别控制数据的存储位置，可以在创建表时使用 FRAGMENT BY 子句。以下语句创建根据循环法分布方案存储数据的分段表。在此示例中，各数据行将跨分段 **dbspace1**、**dbspace2** 和 **dbspace3** 进行程度或高或低的均匀分布。

```
CREATE TABLE name
   (
   rec_num  SERIAL PRIMARY KEY,
   lname    CHAR(20),
   fname    CHAR(20),
   bdate    DATE,
   anniv    DATE,
   email    VARCHAR(25)
   ) FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

有关可以用来创建分段表的不同分布方案的更多信息，请参阅第 5 章。

删除或修改表

使用 DROP TABLE 语句来除去表及其相关联的索引和数据。例如：要通过添加检查约束来更改表的定义，请使用 ALTER TABLE 语句。使用 TRUNCATE 语句来除去表中的所有行和所有的相应索引数据，同时保留表的定义。有关这些语句的信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

使用 CREATE INDEX

使用 CREATE INDEX 语句来对表中的一个或多个列创建索引，并可选择按索引的顺序对物理表进行群集。本节描述创建索引时可用的一些选项。有关 CREATE INDEX 语句的更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

假定您创建表 **customer**:

```
CREATE TABLE customer
(
   cust_num   SERIAL(101) UNIQUE
   fname      CHAR(15),
   lname      CHAR(15),
   company    CHAR(20),
   address1   CHAR(20),
   address2   CHAR(20),
   city       CHAR(15),
   state   CHAR(2),
   zipcode    CHAR(5),
   phone      CHAR(18)
);
```

以下语句显示如何对 **customer** 表的 **lname** 列创建索引:

```
CREATE INDEX lname_index ON customer (lname);
```

组合索引

可以创建包含多列的索引。例如：可以创建以下索引:

```
CREATE INDEX c_temp2 ON customer (cust_num, zipcode);
```

索引的双向遍历

ASC 和 DESC 关键字指定数据库服务器维护索引的顺序。在对列创建索引时，如果省略了这些关键字或指定了 ASC 关键字，则数据库服务器按升序存储键值。如果指定 DESC 关键字，则数据库服务器按降序顺序存储键值。

升序顺序表示键值按从最小键到最大键的顺序进行存储。例如：如果对 **customer** 表的 **lname** 列创建升序索引，则姓氏按以下顺序存储在索引中：`Albertson, Beatty, Currie`。

降序顺序表示键值按从最大键到最小键的顺序进行存储。例如: 如果对 **customer** 表的 **lname** 列创建降序索引，则姓氏按以下顺序存储在索引中: Currie，Beatty，Albertson。

数据库服务器的双向遍历功能允许您只对列创建一个索引并对指定按排序列的升序或降序顺序对结果进行排序的查询使用该索引。

对表名使用同义词

*同义词*是可以用来替换另一个 SQL 标识符的名称。使用 CREATE SYNONYM 语句来声明表、视图或（Dynamic Server）顺序对象的备用名称。

通常，使用同义词来引用不在当前数据库中的表。例如: 可以执行下列语句来创建 **customer** 和 **orders** 表名的同义词:

```
CREATE SYNONYM mcust FOR masterdb@central:customer;
CREATE SYNONYM bords FOR sales@boston:orders;
```

创建同义词之后，可以在原始表名有效的许多上下文中使用该同义词，如以下示例所示:

```
SELECT bords.order_num, mcust.fname, mcust.lname
   FROM mcust, bords
   WHERE mcust.customer_num = bords.Customer_num
   INTO TEMP mycopy;
```

CREATE SYNONYM 语句在当前数据库中的系统目录表 **syssyntable** 中存储同义词名称。该同义词可用于在该数据库中进行的任何查询。（然而如果设置了 **USETABLENAME** 环境变量，则 SQL 的某些 DDL 语句不支持用同义词代替表名。）

简短的同义词可以简化查询的编写工作，但同义词也可以扮演另一角色。它们允许将表移至另一个数据库中，甚至移至另一台计算机，同时保持查询不变。

假设有几个引用表 **customer** 和 **orders** 的查询。这些查询嵌入在程序、表单和报告中。这些表是演示数据库的一部分，演示数据库存放在数据库服务器 **avignon** 上。

现在，您决定将这些程序、表单和报告提供给网络中的另一台计算机（数据库服务器 **nantes**）的用户使用。那些用户有包含名为 **orders** 的表（这个表包含他们所在位置的订单）的数据库，但他们需要存取 **avignon** 上的表 **customer**。

对于那些用户，**customer** 表是外部的。这表示必须准备特殊版本（即必须用数据库服务器名对 **customer** 表加以限定的版本）的程序和报告吗？更好的解决方案是在用户的数据库中创建同义词，如以下示例所示:

```
DATABASE stores_demo@nantes;
CREATE SYNONYM customer FOR stores_demo@avignon:customer;
```

当在您的数据库中执行存储查询时，名称 **customer** 指的是实际的表。当在其它数据库中执行那些查询时，该名称会通过同义词解析为对存在于数据库服务器 **avignon** 上的表的引用。（在不符合 ANSI 的数据库中，同义词必须在数据库中的同义词、表、视图和顺序对象的名称中具有唯一性。在符合 ANSI 的数据库中，*owner.synonym* 组合必须在使用 **tabid** 值在数据库中注册的对象名称空间中具有唯一性。）

使用同义词链

为了继续前面的示例，假设将一台新的计算机添加到网络中。它的名称是 **db_crunch**。将 **customer** 表和其它表移至该计算机上以降低 **avignon** 上的负载。尽管可以非常容易地在新的数据库服务器上重新生成该表，但如何能够将所有存取重定向到它呢？一种方法是安装同义词以替换旧表，如以下示例所示:

```
DATABASE stores_demo@avignon EXCLUSIVE;
RENAME TABLE customer TO old_cust;
CREATE SYNONYM customer FOR stores_demo@db_crunch:customer;
CLOSE DATABASE;
```

当在 **stores_demo@avignon** 中执行查询时，对表 **customer** 的引用将找到该同义词并重定向至新计算机上的版本。在前一示例中，对于从数据库服务器 **nantes** 执行的查询，也会发生此类重定向。数据库 **stores_demo@nantes** 中的同义词仍将对 **customer** 的引用重定向至数据库 **stores_demo@avignon**；在该数据库中，新的同义词将查询发送至数据库 **stores_demo@db_crunch**。

当您想要在一个操作中将对某个表的所有存取重定向时（就象在本示例中一样），同义词链非常有用。然而，您应该尽快更新所有用户的数据库，以使它们的同义词直接指向该表。如果不这样做，数据库服务器处理额外的同义词时就会产生额外的开销，并且如果链中的任何计算机已关闭，就会找不到该表。

可以对本地数据库运行某个应用程序并且以后对另一台计算机上的数据库运行同一个应用程序。该程序在这两种情况下可以运行得一样好（尽管运行速度在网络数据库上可能会慢得多）。只要数据模型相同，程序就无法指出两个数据库之间的差别。

使用命令脚本

您可以交互地输入 SQL 语句以创建数据库和表。在某些情况下，可能必须将数据库和表创建两次或更多次。例如: 在测试版本令人满意后，可能必须再次创建数

据库以生成生产版本，或者必须在数台计算机上实现同一数据模型。为了节省时间和降低出错机会，可以将所有用于创建数据库的语句放在一个文件中并在以后再次执行那些语句。

捕获模式

dbschema 实用程序是一个这样的程序：它检查数据库的内容并生成重新创建该数据库所需的所有 SQL 语句。您可以构建数据库的第一个版本，进行更改，直到它完全令您满意为止。然后，可以使用 **dbschema** 来生成复制该数据库所必需的 SQL 语句。有关 **dbschema** 实用程序的信息，请参阅《*IBM Informix: 迁移指南*》。

执行文件

可以从命令文件运行用来以交互方式输入 SQL 语句的程序（如 DB–Access）。可以启动 DB–Access 来读取并执行您或 **dbschema** 准备的命令文件。有关更多信息，请参阅《*IBM Informix: DB–Access 用户指南*》。

一个示例

大部分 IBM Informix 数据库服务器产品附带提供了演示数据库（本书中的大多数示例使用的数据库）。演示数据库是作为一个操作系统命令脚本交付的，此命令脚本调用 IBM Informix 产品以构建数据库。可以复制此命令脚本并使用它来作为将您自己的数据模型自动化的基础。

填充数据库

对于初始测试，填充数据库的最简单方法是在 DB–Access 中输入 INSERT 语句。例如：要将一行插入到演示数据库的 **manufact** 表中，请在 DB–Access 中输入以下命令：

```
INSERT INTO manufact VALUES ('MKL', 'Martin', 15);
```

如果您正在准备应用程序，如用 C 编写的应用程序，则可以使用该应用程序来将行输入到数据库表中。

下表列示可以用来将信息输入到数据库中的 IBM Informix 工具。“参考”列中的首字母缩写词在表下面有解释。

工具	用途	参考
dbaccessdemo dbaccessdemo_ud	准备并填充样本数据库。	DB-A SQLR
DB–Access	通过输入显式命令来编辑数据库。	DB-A SQLS

工具	用途	参考
onunload 和 onload	从磁带或磁盘上的文件中复制整个数据库或所选数据库表，或将整个数据库或所选数据库表复制到磁带或磁盘上的文件中。	MG AR
dbload	将数据从一个或多个文本文件装入到一个或多个现有的表中。	MG
High-Performance Loader	复制整个数据库、选择的表或所选表的所选列。	HPL
LOAD 和 UNLOAD	从文本文件中装入数据或将数据装入文本文件。	SQLS
dbexport 和 dbimport	使用文本文件来复制整个数据库。	MG
Enterprise Replication	每次更新指定的表时，更新所选数据库。	ER
onxfer	将数据从 IBM Informix Dynamic Server 复制至 Extended Parallel Server。	MG
C 应用程序	使用嵌入在 C 程序中的 SQL 命令来更新数据库。	ESQLC DAPI DBDK
Java 应用程序	使用嵌入在 Java 程序中的 SQL 命令来更新数据库。	Java DBDK
网关应用程序	存取非 Informix 数据库中的数据。	GM GU

助记符	“参考”列的说明
SQLR	*《IBM Informix: SQL 参考指南》*
SQLS	*《IBM Informix: SQL 指南: 语法》*
MG	*《IBM Informix: 迁移指南》*
AR	*《IBM Informix: 管理员参考大全》*
GM	*IBM Informix: Enterprise Gateway Manager User Manual*
GU	*IBM Informix: Enterprise Gateway User Manual*
DBDK	*IBM Informix: DataBlade Developer's Kit User's Guide*
ESQL/C	*IBM Informix: ESQL/C Programmer's Manual*
Java	*IBM Informix: J/Foundation Developer's Guide*
HPL	*《IBM Informix: High-Performance Loader 用户指南》*
DB-A	*《IBM Informix: DB–Access 用户指南》*
ER	*《IBM Informix: Dynamic Server Enterprise Replication 指南》*

从其它 Informix 数据库移动数据

通常，可以从存储在另一个 Informix 数据库中的表中的数据或存储在操作系统文件中的数据派生出表的初始行。下列实用程序允许移动大量数据:

- **onunload/onload** 实用程序
- **dbexport/dbimport** 实用程序
- **dbload** 实用程序
- SQL LOAD 语句
- High Performance Loader（HPL）

也可以从另一数据库服务器上的别的数据库中选择所需的数据来作为数据库中的 INSERT 语句的一部分。如以下示例所示，可以从演示数据库中的 **items** 表中选择信息以插入到新表中:

```
INSERT INTO newtable
   SELECT item_num, order_num, quantity, stock_num,
      manu_code, total_price
   FROM stores_demo@otherserver:items;
```

将源数据装入表中

当数据源不是 Informix 数据库时，必须找到一种方法来将其转换为平面 ASCII 文件；即可打印数据的文件，其中每一行都表示一个表行的内容。

在将数据存放到 ASCII 文件中后，就可以使用 **dbload** 实用程序将该数据装入表中。有关 **dbload** 的更多信息，请参阅《*IBM Informix: 迁移指南*》。DB–Access 中的 LOAD 语句也可以从平面 ASCII 文件装入行。有关 LOAD 和 UNLOAD 语句的信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

Extended Parallel Server

在将数据存放到文件中后，就可以使用*外部表*将该数据装入表中。有关外部表的更多信息，请参阅 《*IBM Informix: 管理员指南*》。

Extended Parallel Server 结束

执行批量装入操作

如果关闭事务日志记录，则可以提高插入数百或数千行时的速度。由于发生故障时可以很容易地重新创建丢失的工作，所以记录这些插入操作是没有意义的。以下列表包含大批量装入操作的步骤:

- 如果其他用户有可能正在使用数据库，则使用 DATABASE EXCLUSIVE 语句排除他们。
- 请求管理员关闭数据库的记录。

  可以使用现有的日志来将数据库恢复到目前状态，并且可以再次运行批量插入操作来恢复那些行（如果丢失了那些行的话）。

Extended Parallel Server

不能对使用 Extended Parallel Server 的数据库关闭记录。但是，可以在数据库中创建非日志记录表（原始持久或静态持久）。

Extended Parallel Server 结束

- 执行用于将数据装入表的语句或运行用于将数据装入表的实用程序。
- 备份新装入的数据库。

  请求管理员执行完全备份或增量备份，或请求管理员使用 **onunload** 实用程序来只创建数据库的二进制副本。
- 复原事务日志记录并释放对数据库的互斥锁定。

第 2 部分 管理数据库

第 5 章 表分段存储策略

在本章中

本章描述数据库服务器支持的分段存储策略并提供不同分段存储策略的示例。本章讨论分段存储、表分段存储的分布方案、创建与修改分段表以及提供分段表的特权。

有关如何制定分段存储策略以减少数据争用和改进查询性能的信息，请参阅《*IBM Informix: 性能指南*》。

什么是分段存储?

分段存储是一项数据库服务器功能，它允许您在表级别控制数据的存储位置。分段存储使您能够根据某种算法或方案来在表中定义行或索引键组。可以将每个组或*分段*（也称为*分区*）存储在与特定物理磁盘相关联的独立数据库空间中。使用 SQL 语句来创建分段并将它们指定给数据库空间。

用来将行或索引键分组为分段的方案称为*分布方案*。分布方案以及在其中定位分段的数据库空间的集合共同构成*分段存储策略*。《*IBM Informix: 性能指南*》讨论了制定分段存储策略时必须作出的决定。

在决定是否将表行和／或索引键分段以及决定应该如何在各分段间分布行或键之后，您决定用来实现此分布的方案。有关 Informix 数据库服务器支持的分布方案的描述，请参阅第 5-4 页的『表分段存储的分布方案』。

创建分段表和索引时，数据库服务器将每个表和索引分段的位置以及其它相关信息存储在名为 sysfragments 的系统目录表中。可使用这个表来访问关于分段表和索引的信息。如果使用用户定义的例程来作为分段存储表达式的一部分，则该信息记录在 **sysfragexprudrdep** 中。有关这些系统目录表包含的信息的描述，请参阅《*IBM Informix: SQL 参考指南*》。

从最终用户或客户机应用程序的角度看来，分段表与非分段表完全相同。不要求客户机应用程序作任何修改就可以允许它们存取分段表中的数据。

对于某些分布方案，数据库服务器具有有关哪些分段包含哪些数据的信息，因此它可以将客户机数据请求路由至适当的分段，而无需访问不相关的分段。（对于循环法分布方案和某些基于表达式的分布方案，数据库服务器无法将客户机数据请求路由至适当的分段。）有关更多信息，请参阅第 5-4 页的『表分段存储的分布方案』。

为何使用分段存储?

如果您的目标是要改进下列各项中的至少一项，请考虑将表分段:

- 单用户响应时间
- 并行性
- 可用性
- 备份和复原特性
- 数据装入

对于您最终实现的分段存储策略，上面每个目标都有其特有的含义。主分段存储目标确定（至少是影响）实现分段存储策略的方式。当您决定是否使用分段存储来实现前述任何目标时，请记住，分段存储要求进行一些附加的管理和监视活动。

有关前述目标以及如何规划分段存储策略的更多信息，请参阅 *《IBM Informix: 性能指南》*。

分段存储是谁的职责?

数据库服务器管理员在分段存储方面的职责与数据库管理员（DBA）在那些方面的职责之间存在着一些重叠。DBA 创建数据库模式，这可以包括表分段存储。另一方面，数据库服务器管理员负责分配将在其中驻留分段表的磁盘空间。由于这些职责都不能以相互隔离的方式执行，所以，实现分段存储要求 DBA 与数据库服务器管理员进行合作。本手册只描述 DBA 为了实现分段存储策略而执行的那些任务。有关数据库服务器管理员为了实现分段存储策略而执行的任务的信息，请参阅 *《IBM Informix: 管理员指南》* 和 *《IBM Informix: 性能指南》*。

增强的分段存储（XPS）

Extended Parallel Server 可以跨属于不同协同服务器（coserver）的磁盘来将表和索引进行分段。每个表分段都可以驻留在与属于不同协同服务器（coserver）的物理磁盘相关联的独立数据库空间中。*dbslice* 提供了用来跨多个协同服务器（coserver）管理许多数据库空间的机制。在创建 dbslice 和数据库空间之后，可以创建跨多个协同服务器（coserver）进行分段的表和索引。

有关跨协同服务器（coserver）对表进行分段的优点的信息，请参阅 *IBM Informix: Extended Parallel Server Performance Guide*。有关如何创建 dbslice 和数据库空间的信息，请参阅 *IBM Informix: Extended Parallel Server Administrator's Guide*。

分段存储和记录

Dynamic Server

对于 Dynamic Server，分段表可以属于日志记录数据库或非日志记录数据库。与非分段表相同，如果某个分段表是非日志记录数据库的一部分，则发生故障时有可能会导致数据不一致。

Extended Parallel Server

对于 Extended Parallel Server，分段表始终属于日志记录数据库。然而，Extended Parallel Server 确实支持若干种日志记录表和非日志记录表类型。有关更多信息，请参阅第 7-3 页的『使用分布式查询配制数据库服务器』。

Extended Parallel Server 结束

表分段存储的分布方案

*分布方案*是数据库服务器用来将行或索引条目分布到分段的方法。Informix 数据库服务器支持下列分布方案:

- **基于表达式** 此分布方案将包含所指定的值的行放在同一个分段中。指定*分段表达式*，它定义用于对每个分段指定一组行的条件（或者作为范围规则或者作为某个任意规则）。可以指定*余项分段*，它存放所有与任何其它分段的条件都不匹配的行（尽管余项分段会降低基于表达式的分布方案的效率）。
- **循环法** 此分布方案将行一个接一个地放在分段中，并在分段系列中旋转以便均匀地分布行。数据库服务器以内部方式定义规则。

  对于 INSERT 语句，数据库服务器对随机数使用散列函数来确定在其中放置行的分段。对于 INSERT 游标，数据库服务器将第一行放在一个随机分段中，将第二行放在下一个顺序分段中，依此类推。如果其中一个分段已满，则将其跳过。

Extended Parallel Server

- **范围分布** 此分布方案确保行跨数据库空间均匀地分段。在范围分布中，数据库服务器根据用户指定的最小和最大整数值来确定行在各分段之间的分布。当数据分布密集且统一时，请使用范围分布方案。
- **系统定义的散列** 此分布方案使用系统定义的内部规则，在分布行时，该规则的目标是在每个分段中存放相同数目的行。
- **混合** 此分布方案将两种分布方案组合到一起。*主*分布方案选择 dbslice。*辅助*分布方案将行放在 dbslice 中的特定数据库空间中。数据库空间通常驻留在不同的协同服务器（coserver）上。

Extended Parallel Server 结束

有关用来指定分布方案的 SQL 语法的完整描述，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE TABLE 和 CREATE INDEX 语句。有关分段存储性能方面的讨论，参阅 《*IBM Informix: 性能指南*》。

基于表达式的分布方案

要指定基于表达式的分布方案，请使用 CREATE TABLE 或 CREATE INDEX 语句的 FRAGMENT BY EXPRESSION 子句。以下示例包括 FRAGMENT BY EXPRESSION 子句，它使用基于表达式的分布方案创建分段表:

```
CREATE TABLE accounts (id_num INT, name char(15))
   FRAGMENT BY EXPRESSION
id_num <= 100 IN dbspace_1,
id_num <100 AND id_num <= 200 IN dbspace_2,
id_num > 200 IN dbspace_3
```

当使用 CREATE TABLE 语句的 FRAGMENT BY EXPRESSION 子句来创建分段表时，必须为正在创建的表的每个分段提供一个条件。

您可以定义*范围规则*或*任意规则*，这些规则向数据库服务器指示将要如何将行分布至分段。下列各节描述不同类型的基于表达式的分布方案。

范围规则

范围规则使用 SQL 关系和逻辑运算符来定义表中每个分段的边界。范围规则可包含运算符的以下有限集合:

- 关系运算符 >、<、>= 和 <=
- 逻辑运算符 AND 和 OR
- 包含内置函数的代数表达式

如以下示例所示，范围规则可以基于简单的代数表达式。在该示例中，表达式是对某一列的简单引用。

```
   FRAGMENT BY EXPRESSION
id_num > 0  AND id_num <= 20 IN dbsp1,
id_num > 20 AND id_num <= 40 IN dbsp2,
id_num > 40 IN dbsp3
```

按照范围规则的表达式可以是多个代数表达式的逻辑乘或逻辑和。下一个示例显示了用来定义两组范围的两个代数表达式。第一组范围基于代数表达式: “YEAR(Died) - YEAR(Born)”；第二组范围基于 “MONTH(Born)”。

```
   FRAGMENT BY EXPRESSION
YEAR(Died) - YEAR(Born) < 21 AND MONTH(Born) >= 1  AND MONTH(Born) < 4 IN dbsp1,
YEAR(Died) - YEAR(Born) < 40 AND MONTH(Born) >= 4  AND MONTH(Born) <  7 IN dbsp2,
```

仲裁规则

任意规则使用 SQL 关系和逻辑运算符。与范围规则不同，任意规则允许使用任何关系运算符和任何逻辑运算符来定义规则。另外，可以在规则中引用任意数目个表列。如以下示例所示，任意规则通常包括使用 OR 逻辑运算符来将数据分组:

```
   FRAGMENT BY EXPRESSION
zip_num = 95228 OR zip_num = 95443 IN dbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbsp4,
REMAINDER IN dbsp5
```

使用 MOD 函数（IDS）

可以在 FRAGMENT BY EXPRESSION 子句中使用 MOD 函数来将表中的每一行映射至一组整数（散列值）。数据库服务器使用这些值来确定将把给定的行存储在哪个分段中。以下示例显示可以如何在基于表达式的分布方案中使用 MOD 函数:

```
   FRAGMENT BY EXPRESSION
MOD(id_num, 3) = 0 IN dbsp1,
MOD(id_num, 3) = 1 IN dbsp2,
MOD(id_num, 3) = 2 IN dbsp3
```

插入和更新行

插入或更新行时，数据库服务器按指定的顺序对分段表达式求值，以了解该行是否属于任何分段。如果是的话，数据库服务器在其中一个分段中插入或更新该行。如果该行不属于任何分段，则将该行放到余项子句指定的分段中。如果分布方案没有包括余项子句，并且该行与任何现有分段表达式的条件都不匹配，则数据库服务器返回错误。

循环法分布方案

要指定循环法分布方案，请使用 CREATE TABLE 语句的 FRAGMENT BY ROUND ROBIN 子句。以下语句举例说明具有循环法分布方案的分段表:

```
CREATE TABLE account_2
   ...
   ...
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

当数据库服务器接收到将一定数目的行插入使用循环法分布的表的请求时，数据库服务器以这样的方式分布行：每个分段中的行数保持大致相同。由于在各分段之间均匀地分布信息，所以循环法分布也称为*均匀分布*。用于将行分布至使用循环法分布的表的规则是数据库服务器的内部规则。

要点: 只能对表分段存储使用循环法分布方案。不能使用此分布方案来对索引进行分段。

范围分布方案（XPS）

当数据分布密集且统一并且分段存储列不包含重复项时，可使用范围分布方案来跨数据库空间均匀地分布行。范围分布使用用户指定的 MIN 和 MAX 值来确定行在各分段之间的分布。

以下语句包含 FRAGMENT BY RANGE 子句来指定范围分布方案:

```
CREATE TABLE cust_account (cust_id INT)
   ...
   ...
FRAGMENT BY RANGE (cust_id MIN 1000 MAX 5000)
   IN dbsp_1, dbsp_2, dbsp_3, dbsp_4)
```

MIN 和 MAX 值指定列中期望的值的总范围。必须在 FRAGMENT BY RANGE 子句中指定 MAX 值。如果省略 MIN 值，则缺省 MIN 值是 0。在前面的示例中，数据库服务器使用 **cust_id** 值来跨四个数据库空间分布表行。数据库服务器按如下方式将行分段。

存储器空间	用于具有列值的行
dbsp_1	1000 <= cust_id < 2000
dbsp_2	2000 <= cust_id < 3000
dbsp_3	3000 <= cust_id < 4000
dbsp_4	4000 <= cust_id < 5000

可以对单个列使用范围分段存储，在混合分布方案中，对于每个 FRAGMENT BY RANGE 子句，也可以对不同的列指定范围方案。有关如何在混合分布方案中使用范围分段存储的信息，请参阅第 5-8 页的『混合分布方案（XPS）』。

系统定义的散列分布方案（XPS）

数据库服务器使用系统定义的散列算法来通过将指定的键散列来均匀地分布数据。除均匀数据分布以外，系统定义的散列分段还允许为使用散列键的查询自动消除分段。可以对若干个表使用散列分段存储，以便当查询中连接了表时提供分段消除或者在本地协同服务器（coserver）上执行更多的处理。

除下列情况以外，系统定义的散列分布方案是用于跨分段均匀分布数据的首选方法:

- 使用了范围查询。

  范围分布方案可以使分段消除效果更好，从而提高查询性能。
- 指定的列包含数目非常不均匀的重复值，或者不同的值的数目非常少。

这两种情况都会导致数据歪斜，在这种情况下，一些分段变得比其它分段大。数据歪斜可能会导致性能不平均，其原因在于数据库服务器在一些分段中需要处理的数据量大于在其它分段中需要处理的数据量。

要指定系统定义的散列分布方案，请在 CREATE TABLE 语句中使用 FRAGMENT BY HASH 子句，如下所示:

```
CREATE TABLE new_tab (id INT, name CHAR(30))
   FRAGMENT BY HASH (id) IN dbspace1, dbspace2, dbspace3;
```

在系统定义的散列分布方案中，至少指定两个要在其中放置分段的数据库空间或指定一个 dbslice。

还可以对系统定义的散列分布方案指定组合键。

混合分布方案（XPS）

*混合*分布方案对同一个表组合使用基本策略和辅助级策略。基本策略可以是基于表达式的策略或范围分段存储策略。可以使用混合分布方案来对一列或两列应用不同的分段存储策略。

在定义混合分布方案时，可指定单个 dbslice、单个数据库空间或多个数据库空间来作为分段存储表达式的存储域。

以下语句定义基于表的两个列的混合方案:

```
CREATE TABLE hybrid_tab (col_1 INT, col_2 DATE, col_3 CHAR(4))
   FRAGMENT BY HYBRID (col_1) EXPRESSION
      col_1 >= 0 AND col_1 < 20 IN dbspace_1,
      col_1 >= 20 AND col_1 < 40 IN dbspace_2,
      col_1 >= 40 IN dbspace_3;
```

创建分段表

本节说明如何使用 SQL 语句来创建和管理分段表。可以在创建表时将该表分段，也可以对现有的非分段表进行分段。下列各节提供了这两种备用方法的概述。有关可用来创建分段表的 SQL 语句的完整语法，请参阅*《IBM Informix: SQL 指南: 语法》*。

在创建分段表之前，必须决定适当的分段存储策略。有关如何制定分段存储策略的信息，请参阅 *《IBM Informix: 性能指南》*。

创建新的分段表

要创建分段表，请使用 CREATE TABLE 语句的 FRAGMENT BY 子句。假定您想要创建与 **stores_demo** 数据库的 orders 表相类似的分段表。您决定使用具有

三个分段的循环法分布方案，并向数据库服务器管理员咨询以设置三个数据库空间（为每个分段设置一个）：dbspace1、dbspace2 和 dbspace3。以下 SQL 语句创建分段表:

```
CREATE TABLE my_orders (
   order_num      SERIAL(1001),
order_date    DATE,
   customer_num   INT,
   ship_instruct  CHAR(40),
   backlog        CHAR(1),
   po_num       CHAR(10),
   ship_date    DATE,
   ship_weight  DECIMAL(8,2),
   ship_charge  MONEY(6),
   paid_date    DATE,
   PRIMARY KEY (order_num),
   FOREIGN KEY (customer_num) REFERENCES customer(customer_num))
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

Dynamic Server

如果 **my_orders** 表驻留在 Dynamic Server 数据库中，则您可能决定改为使用基于表达式的分段存储创建表。假定 **my_orders** 表有 30000 行，并且您想跨三个存储在 dbspace1、dbspace2 和 dbspace3 中的分段均匀地分布行。以下语句显示了如何使用 order_num 列来定义基于表达式的分段存储策略:

```
CREATE TABLE my_orders (order_num SERIAL, ...)
   FRAGMENT BY EXPRESSION
      order_num < 10000 IN dbspace1,
      order_num >= 10000 and order_num < 20000 IN dbspace2,
      order_num >= 20000 IN dbspace3
```

Dynamic Server 结束

Extended Parallel Server

如果 **my_orders** 表驻留在 Extended Parallel Server 数据库中，则可使用系统定义的散列分布方案创建表以便获得跨分段的均匀分布。假定 **my_orders** 表有 120000 行，并且您想跨六个存储在不同数据库空间中的分段均匀地分布行。您决定使用 SERIAL 列 order_num 来定义分段。

以下示例显示如何使用 order_num 列来定义系统定义的散列分段存储策略:

```
CREATE TABLE my_orders (order_num SERIAL, ...)
   FRAGMENT BY HASH (order_num) IN dbspace1, dbspace2,
      dbspace3, dbspace4, dbspace5, dbspace6;
```

您可能会注意到分段表中的 SERIAL 列值与非分段表中的该列值有差异。Extended Parallel Server 在分段中按顺序指定 SERIAL 值，但分段可能会包含来自不连续的范围的值。不能指定这些范围。Extended Parallel Server 控制着这些范围并且只保证它们不重叠。

技巧: 可以将表分段存储在 Extended Parallel Server 上的数据库空间或 dbslice 中。

Extended Parallel Server 结束

从非分段表创建分段表

在下列情况下，可能需要将非分段表转换为分段表:

- 您有应用程序实现的表分段存储版本。

  您可能想要将若干个小表转换为一个大的分段表。下一节讲述在这种情况下如何继续。请遵循第 5-10 页的『使用多个非分段表』一节中的指示信息。

- 要对现有的大表进行分段。

  请遵循第 5-11 页的『使用单个非分段表』一节中的指示信息。

记住，在执行转换之前，必须设置适当数目的数据库空间以包含新创建的分段表。

使用多个非分段表

可以将两个或更多个非分段表组合到单个分段表中。这些非分段表必须具有完全相同的表结构，并且必须存储在独立的数据库空间中。要将非分段表组合到一起，请使用 ALTER FRAGMENT 语句的 ATTACH 子句。

例如: 假定有三个非分段表 account1、account2 和 account3，并且将这三个表分别存储在数据库空间 dbspace1、dbspace2 和 dbspace3 中。所有这三个表具有完全相同的结构，并且要将这三个表组合到一个表中，该表是通过公共列 acc_num 上的表达式进行分段的。

您要将 acc_num 小于或等于 1120 的行存储在 dbspace1 中。acc_num 大于 1120 但小于或等于 2000 的行存储在 dbspace2 中。最后，acc_num 大于 2000 的行将存储在 dbspace3 中。

要使用此分段存储策略将表分段，请执行以下 SQL 语句:

```
ALTER FRAGMENT ON TABLE tab1 ATTACH
   tab1 AS acc_num <= 1120,
   tab2 AS acc_num >  1120 and acc_num <= 2000,
   tab3 AS acc_num > 2000;
```

结果是单个表 tab1。另外两个表 tab2 和 tab3 已被消耗掉，它们不再存在。

有关如何使用 ALTER FRAGMENT 语句的 ATTACH 和 DETACH 子句来改进性能的信息，请参阅 *《IBM Informix: 性能指南》*。

使用单个非分段表

要从非分段表创建分段表，请使用 ALTER FRAGMENT 语句的 INIT 子句。例如：假定要将 orders 表转换为通过循环法进行分段的表。以下 SQL 语句执行此转换：

```
ALTER FRAGMENT ON TABLE orders INIT
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

该非分段表上的任何现有索引都变为采用与该表相同的分段存储策略进行分段。

分段表中的行标识

术语*行标识*指的是一个整数，它定义行的物理位置。在非分段表中，行的行标识是唯一并且恒定的值。与之相对照，分段表中的行不指定行标识。

要点：使用主键而不是行标识来作为应用程序中的存取方法。由于 SQL 的 ANSI 规范定义了主键，所以使用主键来存取数据可以提高应用程序的可移植性。

Extended Parallel Server

对于分段表，数据库服务器不支持行标识。

Extended Parallel Server 结束

Dynamic Server

为了适应必须引用分段表的行标识的应用程序，Dynamic Server 允许为分段表显式地创建行标识列。然而，对于类型表，Dynamic Server 不支持 WITH ROWIDS 子句。

要创建行标识列，请使用以下 SQL 语法：

- CREATE TABLE 语句的 WITH ROWIDS 子句
- ALTER TABLE 语句的 ADD ROWIDS 子句
- ALTER FRAGMENT 语句的 INIT 子句

创建行标识列时，数据库服务器执行下列操作：

- 对表中的每一行添加 4 个字节的唯一值

- 创建内部索引，数据库服务器使用该索引来通过行标识存取表中的数据
- 在 **sysfragments** 系统目录表中为该内部索引插入一行

Dynamic Server 结束

将智能大对象分段（IDS）

可以在 CREATE TABLE 语句的 PUT 子句中指定多个 sbspace 以实现对某个列中的智能大对象的循环法分段存储。如果对 CLOB 或 BLOB 列指定多个 sbspace，则数据库服务器将该列的智能大对象以循环法方式分布至指定的 sbspace。给定以下 CREATE TABLE 语句，数据库服务器可以将 **cat_photo** 列中的大对象以循环法方式分布至 **sbcat1**、**sbcat2** 和 **sbcat3**。

```
CREATE TABLE catalog (
   catalog_num  SERIAL,
   stock_num    SMALLINT,
   manu_code    CHAR(3),
   cat_descr    LVARCHAR,
   cat_photo    BLOB)
PUT cat_photo in (sbcat1, sbcat2, sbcat3;
```

修改分段存储策略

可以对分段表进行两种一般类型的修改。第一类由可以对非分段表进行的修改组成。这样的修改包括添加列、删除列以及更改列数据类型等。对于这些修改，请使用通常对非分段表使用的 ALTER TABLE 语句。第二类修改由对分段存储策略所作的更改组成。本节说明如何使用 SQL 语句来修改分段存储策略。

有时，在实现分段存储之后，您可能需要改变分段存储策略。最常见的情况是，在将分段存储与查询内并行化或查询间并行化配合使用时，将需要修改分段存储策略。在这些情况下，修改分段存储策略是可以用来改进数据库服务器系统性能的若干方法中的一种方法。

重新初始化分段存储策略

可使用带有 INIT 子句的 ALTER FRAGMENT 语句来对非分段表定义并初始化新的分段存储策略或对分段表转换现有分段存储策略。还可以使用 INIT 子句来更改分段表达式的求值顺序。

以下示例显示可以如何使用 INIT 子句来彻底地重新初始化分段存储策略。

假定最初创建了以下分段表:

```
CREATE TABLE account (acc_num INTEGER, ...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 in dbspace1,
      acc_num > 1120 and acc_num < 2000 in dbspace2,
      REMAINDER IN dbspace3;
```

假定使用此分布方案运作了数个月后您发现 dbspace2 中包含的分段中的行数是另外两个分段所包含的行数的两倍。这种不平衡导致包含 dbspace2 的磁盘成为 I/O 瓶颈。

要校正这种情况，您决定修改分布，以使每个分段中的行数大致均匀。应修改分布方案以使其包含四个分段而不是三个分段。新的数据库空间 dbspace2a, 用于包含新分段，该新分段存储先前包含在 dbspace2 中的前半部分的行。dbspace2 中的分段包含它先前存储的后半部分的行。

要实现新的分布方案，首先要创建数据库空间 dbspace2a，然后执行以下语句:

```
ALTER FRAGMENT ON TABLE account INIT
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 in dbspace1,
      acc_num >  1120 and acc_num <= 1500 in dbspace2a,
      acc_num >  1500 and acc_num < 2000 in dbspace2,
      REMAINDER IN dbspace3;
```

执行此语句时，数据库服务器将立即废弃旧的分段存储策略，并且根据新的分段存储策略将表包含的行重新分布。

还可以使用 ALTER FRAGMENT 的 INIT 子句来执行下列操作:

- 将单个非分段表转换为分段表
- 将分段表转换为非分段表
- 将通过任何策略进行分段的表转换为采用任何其它分段存储策略

有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 ALTER FRAGMENT 语句。

修改 Dynamic Server 的分段存储策略

Dynamic Server 允许您使用 ADD、DROP 和 MODIFY 子句来更改对表或索引的分段存储策略。有关这些选项的语法信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 ALTER FRAGMENT 语句。

使用 ADD 子句

在定义分段存储策略时，您可能需要添加一个或多个分段。可使用 ALTER FRAGMENT 语句的 ADD 子句来向表添加新分段。假定要将一个分段添加到使用以下语句创建的表中:

```
CREATE TABLE sales (acc_num INT, ...)
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

要向表 **sales** 添加新分段 dbspace4，请执行以下语句:

```
ALTER FRAGMENT ON TABLE sales ADD dbspace4;
```

如果分段存储策略基于表达式，则 ALTER FRAGMENT 的 ADD 子句包含用于在现有数据库空间之前或之后添加数据库空间的选项。

使用 DROP 子句

在定义分段存储策略时，可能需要删除一个或多个分段。对于 Dynamic Server，可使用 ALTER FRAGMENT ON TABLE 语句的 DROP 子句来从表中删除分段。假定要从使用以下语句创建的表中删除分段:

```
CREATE TABLE sales (col_a INT), ...)
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

以下 ALTER FRAGMENT 语句使用 DROP 子句来从 **sales** 表中删除第三个分段 **dbspace3**:

```
ALTER FRAGMENT ON TABLE sales DROP dbspace3;
```

发出此语句时，将把 dbspace3 中所有的行移至剩余的数据库空间 dbspace1 和 dbspace2。

使用 MODIFY 子句

使用带有 MODIFY 子句的 ALTER FRAGMENT 语句来修改现有分段存储策略中的一个或多个表达式。

假定最初创建了以下分段表:

```
CREATE TABLE account (acc_num INT, ...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 IN dbspace1,
      acc_num > 1120 AND acc_num < 2000 IN dbspace2,
      REMAINDER IN dbspace3;
```

执行以下 ALTER FRAGMENT 语句时，确保没有值小于或等于零的帐号存储在 dbspace1 包含的分段中:

```
ALTER FRAGMENT ON TABLE account
   MODIFY dbspace1 TO acc_num > 0 AND acc_num <=1120;
```

不能使用 MODIFY 子句来改变分布方案包含的分段数。请改为使用 ALTER FRAGMENT 的 INIT 或 ADD 子句。

修改 XPS 的分段存储策略

Extended Parallel Server 支持 ALTER FRAGMENT ON TABLE 语句的下列选项:

- ATTACH 子句
- DETACH 子句
- INIT 子句

使用 HASH 分段存储的表只支持 INIT 选项。

Extended Parallel Server 不支持 DROP 或 MODIFY 选项、ALTER FRAGMENT ON INDEX 语句或显式的 rowid 列。要处理删除或修改操作，可使用受支持的选项来替代 DROP 和 MODIFY。

使用 INIT 子句

如果对分段存储策略所作的更改要求移动数据，则可随 ALTER FRAGMENT ON TABLE 语句一起指定 INIT 子句。当使用 INIT 子句时，数据库服务器将使用新的分段存储方案来创建表的副本，并将原始表中的行插入到新表中。

假定创建了下面这个 **prod_info** 表，由于查询通常对 **id** 列使用相等搜索，所以通过基于 **id** 列的散列来分布分段:

```
CREATE TABLE prod_info
   (id      INT,
   color    INT,
   details  CHAR(100))
FRAGMENT BY HASH(id) IN dbsl;
```

假定您在某个时刻意识到需要执行其它重要的查询，那些查询指定 **color** 列值，但没有指定 **id** 值。要处理此类情况，可以修改 **prod_info** 表的数据布局，以便能够更好地进行分段消除。以下 ALTER FRAGMENT 语句显示了可以如何使用 INIT 子句来从散列分布方案更改为混合分布方案:

```
ALTER FRAGMENT ON TABLE prod_info INIT
   FRAGMENT BY HYBRID(id)
    EXPRESSION color = 1 IN dbsl, color = 2 IN dbsl2, ...
    REMAINDER IN dbsl8;
```

使用 ATTACH 和 DETACH 子句

如果需要移动数据，则可使用带有 INIT 子句的 ALTER FRAGMENT 语句。否则，可使用带有下列选项的 ALTER FRAGMENT 来修改现有分段的表达式:

- 使用 DETACH 子句来除去要修改其表达式的分段。
- 使用 ATTACH 子句来将该分段与新表达式重新连接。

假定最初创建了以下分段表:

```
CREATE TABLE account (acc_num INT, ...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 IN dbspace1,
      acc_num > 1120 AND acc_num < 2000 IN dbspace2,
      REMAINDER IN dbspace3;
```

下列语句修改 dbspace1 包含的分段，以确保不在该分段中存储值小于或等于零的帐号:

```
ALTER FRAGMENT ON TABLE account DETACH dbspace1 det_tab;
CREATE TABLE new_tab (acc_num INT, ...)
   FRAGMENT BY EXPRESSION
      acc_num > 0 AND acc_num <=1120 IN dbspace1;
ALTER FRAGMENT ON TABLE account ATTACH account, new_tab;
INSERT INTO account SELECT * FROM det_tab;
DROP TABLE det_tab;
```

要点: 当表采用散列分段存储时，不能使用带有 ATTACH 子句或 DETACH 子句的 ALTER TABLE 语句。然而，可以对采用散列分段存储的表使用带有 INIT 子句的 ALTER TABLE 语句。

使用 ATTACH 子句来添加分段: 可使用 ALTER FRAGMENT ON TABLE 语句的 ATTACH 子句来将分段添加到表中。假定要将一个分段添加到使用以下语句创建的表中:

```
CREATE TABLE sales (acc_num INT, ...)
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

要将新分段 dbspace4 添加到 **sales** 表，首先要创建一个与 **sales** 具有相同结构的新表，该新表指定了新分段。然后，将 ATTACH 子句与 ALTER FRAGMENT 语句配合使用来将新分段添加到该表。下列语句将新分段添加到 **sales** 表:

```
CREATE TABLE new_tab (acc_num INT, ...) IN dbspace4;
ALTER FRAGMENT ON TABLE sales ATTACH sales, new_tab;
```

在执行 ATTACH 子句之后，数据库服务器将 **sales** 表分段存储到四个数据库空间中: **sales** 的三个数据库空间和 **new_tab** 的数据库空间。**new_tab** 表已被消耗掉。

使用 DETACH 子句来删除分段: 可使用 ALTER FRAGMENT ON TABLE 语句的 DETACH 子句来从表中删除分段。假定要从使用以下语句创建的表中删除分段:

```
CREATE TABLE sales (acc_num INT)...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 IN dbspace1,
      acc_num > 1120 AND acc_num <= 2000 IN dbspace2,
      acc_num > 2000 AND acc_num < 3000 IN dbspace3,
      REMAINDER IN dbspace4;
```

要在不丢失任何数据的情况下从 **sales** 表中删除第三个分段 dbspace3，请执行以下语句:

```
ALTER FRAGMENT ON TABLE sales DETACH dbspace3 det_tab;
INSERT INTO sales SELECT * FROM det_tab;
DROP TABLE det_tab;
```

ALTER FRAGMENT 语句将 **dbspace3** 与 **sales** 表的分布方案拆离并将行放到新表 **det_tab** 中。INSERT 语句将先前位于 **dbspace3** 中的行重新插入到新的 **sales** 表中，该表现在具有三个分段: **dbspace1**、**dbspace2** 和 **dbspace4**。由于不再需要 **det_tab** 表，所以 DROP TABLE 语句将其删除。

授予和撤销对分段的特权（IDS）

如果要授予有用的分段特权，则需要一种策略来控制数据分布。一种有效的策略是通过表达式将数据记录分段。另一方面，循环法数据记录分布策略不是有用的策略，其原因在于此策略将把每个新数据记录添加至下一个分段。循环法分布使任何跟踪数据分布的清除方法无效，因此取消了对分段权限的任何实际使用。由于基于表达式的分布与循环法分布之间存在这种差异，所以 GRANT FRAGMENT 和 REVOKE FRAGMENT 语句只适用于采用基于表达式的分段存储的表。

创建分段表时，不存在缺省分段权限。使用 GRANT FRAGMENT 语句来授予对一个或多个分段的插入、更新或删除权限。如果要同时授予全部三项特权，请使用 GRANT FRAGMENT 语句的 ALL 关键字。然而，不能仅仅通过命名包含分段的表来授予分段特权。必须命名特定的分段。

当要取消插入、更新或删除特权时，请使用 REVOKE FRAGMENT 语句。此语句取消一个或多个用户对分段表的一个或多个分段的特权。如果要取消当前对某个表存在的所有特权，可使用 ALL 关键字。如果不在此命令中指定任何分段，则所取消的许可权适用于表中当前具有许可权的所有分段。

有关更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 GRANT FRAGMENT、REVOKE FRAGMENT 和 SET 语句。

第 6 章 授予和限制对数据库的存取权

在本章中

本章描述可以如何控制对数据库的存取权。在某些数据库中，每个用户都可以存取所有数据。在其它数据库中，拒绝某些用户存取某些或全部数据。

使用 SQL 来限制对数据的存取

可以在下列级别限制对数据的存取:

- 可使用 GRANT 和 REVOKE 语句来授予或拒绝对数据库或对特定表的存取权，并可以控制对数据库的使用的种类。
- 可使用 CREATE PROCEDURE 或 CREATE FUNCTION 语句来编写和编译用户定义的例程，此例程控制并监视可以读取、修改或创建数据库表的用户。
- 可使用 CREATE VIEW 语句来准备数据的受限制或经过修改的视图。此限制可以是垂直的（不包括特定的列）和／或水平的（不包括特定的行）。
- 可以将 GRANT 和 CREATE VIEW 语句组合使用，以便精确地控制用户可以修改表的哪些部分以及使用哪些数据。
- 使用 Dynamic Server 时可以用 SET ENCRYPTION PASSWORD 语句和 SQL 的内置加密和解密功能实现对敏感数据的列级别加密。未授权的用户即使能够查看加密的字符、BLOB 或 CLOB 列值，但如果没有 DES 或三重 DES 加密密钥（该密钥不存储于数据库中）的话也无法复原数据的明文。

控制对数据库的存取权

正常数据库特权机制基于 GRANT 和 REVOKE 语句，第 6-3 页的『授予特权』对这些语句作了讨论。然而，有时可使用操作系统的工具来作为对数据库访问的附加控制方法。

无论操作系统提供了哪些访问控制，当整个数据库的内容高度敏感时，您可能不想将其留在固定于计算机中的公用磁盘上。当数据必须安全时，您可以绕过正常软件控制。

当您或另一位经过授权的人员不使用数据库时，不必将数据库联机。可以通过下列其中一个方法来使其变为不可访问，这些方法具有不同程度的不方便性:

- 将物理介质从计算机上拆离并将其带走。如果磁盘本身不可卸下，则磁盘驱动器可能是可卸下的。
- 将数据库目录复制至磁带并占有该磁带。
- 使用加密实用程序来复制数据库文件。只保留加密版本。

要点: 在后两种情况下，在创建副本之后，必须记住使用以 NULL 数据覆盖被擦除文件的程序来擦除原始数据库文件。

与除去整个数据库目录相反，可以复制并接着擦除代表个别表的文件。不要忽视索引文件包含建立了索引的列中的数据的副本这个事实。请除去并擦除索引文件以及表文件。

授予特权

使用数据库的权限称为*权限特权*。例如: 使用数据库的权限称为“连接”特权; 将行插入表中的权限称为“插入”特权。使用 GRANT 语句可授予对数据库、表、视图或过程的特权，或将某个角色授予用户或另一个角色。使用 REVOKE 语句可撤销对数据库或数据库对象的特权，或撤销用户或另一个角色的角色。

*角色*是 DBA 分配的一类存取特权，如**工资单**。使用 CREATE ROLE 语句创建角色后，DBA 可以使用 GRANT 语句将存取特权分配给角色，然后将角色分配给个别用户（或分配给其它角色），从而使有相似工作任务的用户可以拥有其工作任务所需要的一组存取特权。通过将特权分配给角色，然后将角色分配给用户，可以简化对特权的管理。有关角色中管理存取特权的角色的其它信息，另见第 6-11 页的『外部例程』和第 6-13 页的『使用角色』。

下列特权组控制着用户可以对数据和数据库对象执行的操作:

- 数据库级别特权
- 所有权特权
- 表级别特权
- 列级别特权

Dynamic Server

- 类型级别特权
- 例程级别特权
- 语言级别特权

Dynamic Server 结束

- 自动化特权

有关 GRANT 和 REVOKE 语句的语法，请参阅《*IBM Informix: SQL 指南: 语法*》。

数据库级别特权

三个级别的数据库特权提供了全面的方法来控制哪些人可以存取数据库。只有个别用户（不是角色）可以拥有数据库级别的特权。

连接特权

最低的特权级别是“连接”，它向用户提供查询和修改表的基本能力。具有“连接”特权的用户可以执行下列功能:

- 执行 SELECT、INSERT、UPDATE 和 DELETE 语句（如果具有必需的表级别特权的话）。
- 执行 SPL 例程（如果具有必需的表级别特权的话）。
- 创建视图（如果允许查询视图所基于的表的话）。
- 创建临时表并对临时表创建索引。

在用户可以存取数据库之前，他们必须具有“连接”特权。通常，在不包含高度敏感或专用数据的数据库中，您在创建数据库后没多久就提供 GRANT CONNECT TO PUBLIC 特权。

如果不将“连接”特权授予 PUBLIC，则可以通过数据库服务器存取数据库的用户只包括那些明确对其授予“连接”特权的用户。如果有限的用户应该具有存取权，则此特权使您能够向他们提供此特权并对所有其他用户拒绝此特权。

用户和 public: 特权通过名称授予单个用户或以名称 PUBLIC 授予所有用户。授予 PUBLIC 的任何特权都作为缺省特权。

在执行语句之前，数据库服务器确定用户是否具有必需的特权。此信息位于系统目录中。有关更多信息，请参阅第 6-6 页的『系统目录表中的特权』。

数据库服务器首先查找以明确方式授予发出请求的用户的特权。如果找到这样的授权，则使用该信息。然后，数据库服务器查看是否已将限制性更弱的特权授予 PUBLIC。如果有的话，数据库服务器就使用限制性更弱的特权。如果没有对该用户进行任何授权，则数据库服务器查找已授予 PUBLIC 的特权。如果找到相关的特权，则使用该特权。

因此，要为所有用户设置最低级别的特权，应将特权授予 PUBLIC。在特定的情况下可覆盖该特权，方法是将更高的个别特权授予用户。

资源特权

“资源”特权具有与“连接”特权相同的权限。另外，具有“资源”特权的用户可以创建新的永久表、索引和 SPL 例程，从而永久地分配磁盘空间。

数据库管理员特权

最高级别的数据库特权是数据库管理员，即 DBA。创建数据库时，您自动成为 DBA。DBA 特权的拥有者可以执行下列功能:

- 执行 DROP DATABASE、START DATABASE 和 ROLLFORWARD DATABASE 语句。
- 删除或改变任何对象，而不管该对象的所有者是谁。
- 创建将由其他用户拥有的表、视图和索引。
- 将数据库特权（包括 DBA 特权）授予另一个用户。

在 V10.0 之前的 Dynamic Server 发行版中，DBA 可以执行直接修改系统目录表行的 DML 和 DDL 语句。然而在本发行版中，只有用户 **informix** 可以直接修改系统目录表。但如果您是用户 **informix**，则 IBM 强烈建议尽量不要修改任何系统目录表的内容或模式，因为此类操作可能会破坏数据库的完整性。

所有权权限

数据库以及其中的每个表、视图、索引、过程和同义词都具有所有者。尽管具有 DBA 特权的用户可以创建将要由其他用户拥有的对象，但是对象的所有者通常是创建该对象的人员。

数据库对象的所有者对该对象具有所有权限，并可以在不具有附加特权的情况下改变或删除该对象。

对于 Extended Parallel Server 的“通用键”（GK）索引，所有权权限的处理方式与其它对象的所有权权限的处理方式略有不同。在删除 GK 索引之前，不能删除任何出现在该 GK 索引的 FROM 子句中的表，即使该 GK 索引是由除表的创建者以外的人员创建的。有关更多信息，参阅第 12-12 页的『在数据仓储环境中使用 GK 索引』。

表级别特权

可以逐个表地应用七项特权以允许非所有者具有所有者的特权。其中的四项特权（“选择”、“插入”、“删除”和“更新”特权）控制对表中数据的 DML 存取。“索引”特权控制索引的创建。“改变”特权给予更改表定义的权限。“引用”特权给予对表指定引用约束的权限。

在符合 ANSI 的数据库中，只有表所有者才具有任何特权。在其它数据库中，作为表创建操作的一部分，数据库服务器会自动将除“改变”和“引用”以外的所有表特权授予 PUBLIC，除非为了限制 PUBLIC 的所有表特权而已将 **NODEFDAC** 环境变量设置为“yes”。当允许数据库服务器自动将所有表特权授予 PUBLIC 时，任何具有“连接”特权的用户都可以存取新创建的表。如果这不是您所期望的（如果存在具有“连接”特权的用户，但他们本不应该能够存取此表），则创建表之后必须撤销 PUBLIC 对该表的所有特权。

存取特权

四项特权控制用户可以如何存取表。作为表的所有者，您可以独立地授予或撤销下列特权:

- “选择”允许进行选择，包括选择到临时表中。
- “插入”允许用户添加新行。
- “更新”允许用户修改现有的行。
- “删除”允许用户删除行。

用户必须具有“选择”特权才能检索表的内容。然而，“选择”特权不是其它特权的前提条件。用户不必具有“选择”特权就可以具有“插入”或“更新”特权。

例如: 应用程序可能有一个使用表。每次启动特定的程序时，它都将一行插入到使用表中，以记载使用了该程序。在程序终止之前，它更新该行以显示运行该程序的时间长度，并可能记录它的用户执行的工作的计数。

如果要让程序的任何用户能够插入和更新此使用表中的行，则将对于该表的“插入”和“更新”特权授予 PUBLIC。然而，您可以将“选择”特权仅授予少数用户。

系统目录表中的特权: 特权记录在系统目录表中。任何具有“连接”特权的用户都可以查询系统目录表以确定授予哪些特权以及授予哪些用户。

数据库特权记录在 **sysusers** 系统目录表中，在这个表中，主键是用户的标识，唯一的另一个列包含单个字符 C、R 或 D，表示特权级别。对关键字 PUBLIC 的授权是作为用户名 PUBLIC（小写）来反映的。

表级别特权记录在 **systabauth** 中，它使用包含表号、授权者和被授权者的组合主键。在 **tabauth** 列中，特权的编码如以下列表所示。

代码	含义
s	无条件的选择
u	更新

- 未授权特权
i 插入
d 删除
x 索引
a 改变
r 引用

连字符表示未授予的特权，因此授予所有特权显示为 su-idxar，-u------ 表示只授予“更新”特权。代码字母通常是小写的，但当 GRANT 语句中使用了 WITH GRANT OPTION 关键字时，代码字母是大写的。

当第三个位置中出现星号（*）时，表示对该表和被授权者存在一些列级别的特权。特定的特权记录在 **syscolauth** 中。它的主键是表号、授权者、被授权者和列号的组合。唯一的属性是一个包含三个字母的列表，它显示特权的类型：s、u 或 r。

索引、改变和引用特权

“索引”特权允许它的拥有者对表创建和改变索引。与“选择”、“插入”、“更新”和“删除”特权相似，当创建表时，将把“索引”特权自动授予 PUBLIC。

可以将“索引”特权授予任何人，但为了行使该特权，用户还必须拥有“资源”数据库特权。因此，尽管“索引”特权是自动授予的（在符合 ANSI 的数据库中除外），对数据库只具有“连接”特权的用户仍然无法行使他们的“索引”特权。由于索引会占用大量的磁盘空间，所以这样的限制是合理的。

“改变”特权允许它的拥有者对表使用 ALTER TABLE 语句，包括添加和删除列以及复位 SERIAL 列的起始点等权力。只应该将“改变”特权授予对数据模型具有很好的理解的用户以及您信任他们会谨慎地行使其权力的用户。

“引用”特权允许对表施加引用约束。与“改变”特权相同，只应该将“引用”特权授予对数据模型具有很好的理解的用户。

类型表的隶属于特权（IDS）

可以授予或取消“隶属于”特权以控制用户是否可以使用类型表来作为继承层次结构中的超表。创建表时，将自动地把“隶属于”特权授予 PUBLIC（在符合 ANSI 的数据库中除外）。在符合 ANSI 的数据库中，将把对表的“隶属于”特权授予表的所有者。要限制哪些用户可以将某个表定义为继承层次结构中的超表，首先必须撤销 PUBLIC 的“隶属于”特权，然后指定要对其授予“隶属于”特权的用户。例如：要指定只有有限的一组用户可使用 **employee** 表来作为继承层次结构中的超表，可执行下列语句:

```
REVOKE UNDER ON employee
   FROM PUBLIC;

GRANT UNDER ON employee
   TO johns, cmiles, paulz
```

有关如何使用 UNDER 子句来在继承层次结构中创建表的信息，请参阅第 9-6 页的『表继承』。

表分段的特权（IDS）

使用 GRANT FRAGMENT 语句来授予对分段表的个别分段的插入、更新和删除特权。GRANT FRAGMENT 语句仅对采用基于表达式的分布方案进行分段的表有效。

假定创建一个 **customer** 表，这个表通过表达式分为三个分段，这些分段驻留在数据库空间 **dbsp1**、**dbsp2** 和 **dbsp3** 中。以下语句显示如何只将对前两个分段（**dbsp1** 和 **dbsp2**）的插入特权授予用户 **jones**、**reed** 和 **mathews**。

```
GRANT FRAGMENT INSERT ON customer (dbsp1, dbsp2)
   TO jones, reed, mathews
```

要授予对表的所有分段的特权，请使用 GRANT 语句或 GRANT FRAGMENT 语句。

有关 GRANT FRAGMENT 和 REVOKE FRAGMENT 语句的信息，请参阅《*IBM Informix: SQL 指南: 语法*》。

列级别特权

可使用特定列的名称来对“选择”、“更新”和“引用”特权进行限定。通过命名特定的列，就可以授予对表的特定存取权。可以允许用户只查看特定的列、只更新特定的列或只对特定的列施加引用约束。

可使用 GRANT 和 REVOKE 语句来授予或限制对表数据的存取权。此功能解决了只有特定用户才应该了解某个雇员的薪水、工作表现评价或其它敏感属性这一问题。假定雇员数据表是按以下示例所示的方式定义的:

```
CREATE TABLE hr_data
   (
   emp_key INTEGER,
   emp_name CHAR(40),
   hire_date DATE,
   dept_num SMALLINT,
   user-id CHAR(18),
   salary DECIMAL(8,2)
   performance_level CHAR(1),
   performance_notes TEXT
   )
```

由于这个表包含敏感数据，所以在创建它之后立即执行以下语句:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

对于“人力资源”部门中的所选人员以及对于所有经理，执行以下语句:

```
GRANT SELECT ON hr_data TO harold_r
```

这样，您就允许特定用户查看所有的列。(本章的最后一节讨论了一种限制经理只能查看他们的雇员的方法。)对于执行工作表现评价的第一线经理，可执行如下的语句:

```
GRANT UPDATE (performance_level, performance_notes)
   ON hr_data TO wallace_s, margot_t
```

此语句允许经理输入他们对他们的雇员的评价。您将只对“人力资源”部门的经理或被委托改变薪水等级的人员执行如下语句:

```
GRANT UPDATE (salary) ON hr_data to willard_b
```

对于“人力资源”部门中的雇员，可执行如下语句:

```
GRANT UPDATE (emp_key, emp_name, hire_date, dept_num)
   ON hr_data TO marvin_t
```

此语句使特定用户有能力维护不敏感的列，但拒绝他们更改工作表现等级或薪水的权限。MIS 部门中负责指定计算机用户标识的人员是如下语句的受益者:

```
GRANT UPDATE (user_id) ON hr_data TO eudora_b
```

对于允许连接至数据库但未授权查看薪水或工作表现评价的用户，执行如下的语句以允许他们查看不敏感的数据:

```
GRANT SELECT (emp_key, emp_name, hire_date, dept_num, user-id)
   ON hr_data TO george_b, john_s
```

这些用户可以执行如下的查询:

```
SELECT COUNT(*) FROM hr_data WHERE dept_num IN (32,33,34)
```

然而，任何执行如下查询的尝试都将生成错误消息并且不返回数据:

```
SELECT performance_level FROM hr_data
   WHERE emp_name LIKE '*Smythe'
```

类型级别特权

Dynamic Server 支持用户定义的数据类型（UDT）。创建用户定义的数据类型时，只有 DBA 或该数据类型的所有者才可以授权或撤销类型级别特权（这些特权控制哪些用户可以使用该 UDT）。Dynamic Server 支持下列类型级别特权:

- “使用”特权，这是使用用户定义的数据类型的权限

- “隶属于”特权，这是将命名行类型定义为继承层次结构中的超类型的权限

用户定义的类型的使用特权

要控制哪些用户可使用不透明类型、单值类型或命名行类型，请指定对该数据类型的“使用”特权。“使用”特权允许 DBA 或类型的所有者限制用户对列或程序变量（或者表或视图，对于命名行类型而言）指定数据类型或者对该数据类型指定数据类型转换的能力。创建数据类型时将把“使用”特权自动授予 PUBLIC（在符合 ANSI 的数据库中除外）。在符合 ANSI 的数据库中，将把对数据类型的“使用”特权授予该数据类型的所有者。

要限制哪些用户可以使用不透明、单值或命名行类型，首先必须撤销 PUBLIC 的“使用”特权，然后指定要对其授予“使用”特权的用户的名称。例如：要限制只有一组用户可使用名为 **circle** 的数据类型，可执行下列语句：

```
REVOKE USAGE ON circle
   FROM PUBLIC;

GRANT USAGE ON circle
   TO dawns, stevep, terryk, camber;
```

命名行类型的隶属于特权

对于命名行类型，可授予或取消“隶属于”特权，此特权控制用户是否可以将命名行类型指定为继承层次结构中的另一个命名行类型的超类型。创建命名行类型时将把“隶属于”特权自动授予 PUBLIC（在符合 ANSI 的数据库中除外）。在符合 ANSI 的数据库中，将把对命名行类型的“隶属于”特权授予该类型的所有者。

要限制特定用户将命名行类型定义为继承层次结构中的超类型的能力，首先必须撤销 PUBLIC 的“隶属于”特权，然后指定要对其授予“隶属于”特权的用户的名称。例如：要指定只有有限的一组用户可使用命名行类型 **person_t** 来作为继承层次结构中的超类型，可执行下列语句：

```
REVOKE UNDER ON person_t
   FROM PUBLIC;

GRANT UNDER ON person_t
   TO howie, jhana, alison
```

有关如何使用 UNDER 子句来在继承层次结构中创建命名行类型的信息，请参阅第 9-2 页的『类型继承』。

例程级别特权

可将“执行”特权应用到用户定义的例程（UDR）以授权非所有者执行 UDR。如果在不符合 ANSI 的数据库中创建 UDR，则缺省例程级别特权是 PUBLIC；除非首先取消“执行”特权，否则不需要将此特权授予特定用户。如果在符合 ANSI 的

数据库中创建例程，则缺省情况下没有其他用户具有“执行”特权；必须将“执行”特权授予特定用户。以下示例将“执行”特权授予用户 **orion**，以便 **orion** 可使用名为 **read_address** 的 UDR:

```
GRANT EXECUTE ON read_address TO orion;
```

sysprocauth 系统目录表记录例程级别特权。**sysprocauth** 系统目录表使用由例程号、授权者和被授权者组成的主键。在 **procauth** 列中，执行特权由小写的 *e* 指示。如果执行特权是使用 WITH GRANT 选项授予的，则该特权由大写的 *E* 表示。

有关例程级别特权的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

语言级别特权

Dynamic Server 支持用内建存储过程语言（SPL） 编写的 UDR，也支持用 C 语言和 Java 语言编写的 UDR（称为*外部例程*）。要创建任何 UDR，用户在该数据库中必须具有 RESOURCE 特权。另外，要使用 SPL 语言创建 UDR，用户必须还拥有对 SPL 语言的“使用”特权。

SPL 例程

缺省情况下，SPL 的语言使用特权会授予用户 **informix** 和具有 DBA 特权的用户。然而，只有用户 **informix** 才可以将语言使用特权授予其他用户。具有 DBA 特权的用户拥有语言使用特权，但不能将这些特权授予其他用户。缺省情况下，创建 SPL 例程的使用特权会授予 PUBLIC。

以下语句显示用户 **informix** 如何撤销 PUBLIC 使用 SPL 创建 UDR 的许可权，但却将该许可权授予 **mays**、**jones** 和 **freeman**:

```
REVOKE USAGE ON LANGUAGE SPL FROM PUBLIC
GRANT USAGE ON LANGUAGE SPL TO mays, jones, freeman
```

假定已取消了 PUBLIC 的对 SPL 例程的缺省“使用”特权。以下语句显示具有 DBA 特权的用户如何将注册 SPL 例程的“使用”特权授予用户 **franklin**、**reeves** 和 **wilson**:

```
GRANT USAGE ON LANGUAGE SPL TO franklin, reeves, wilson
```

外部例程

Dynamic Server 的本发行版不支持对用 C 语言或 Java 语言编写的外部例程的语言级别特权。然而当 IFX_EXTEND_ROLE 配置参数为 ON 时，会通过内建的 EXTEND 角色提供同样的功能，任何用户注册、删除或替换用 C 语言或 Java 语言编写的 UDR 或 DataBlade 模块时都需要该角色。

只有数据库服务器管理员（DBSA）（缺省情况下为用户 **informix**）可以进行 EXTEND 角色的授权。与其它角色不同，EXTEND 角色不需要使用 SET ROLE 语句激活，也不需要使用 GRANT 语句将特权分配给 EXTEND 角色。然而启用该功能时，只有拥有该角色的用户才可以创建或删除外部 UDR 或 DataBlade 模块。

DBSA 也可以选择将 IFX_EXTEND_ROLE 配置参数设置为 OFF（或将该参数复位）来禁用该限制。在这种情况下，任何对数据库拥有 RESOURCE 特权的用户都可以创建用 C 语言或 Java 语言 C 编写的 UDR。

自动化特权

这种设计似乎迫使您在最初设置数据库时执行大量单调乏味的 GRANT 语句。此外，当人们改变工作时，需要不断地维护特权。例如：如果“人力资源”部门的某个雇员被解雇，则您可能想尽快取消“更新”特权，否则这个满心不愉快的雇员可能会执行如下的语句:

```
UPDATE hr_data
   SET (emp_name, hire_date, dept_num) = (NULL, NULL, 0)
```

不具有那么戏剧性变化但具有同等必要性的情况是，在任何包含敏感数据的模型中，每天甚至每小时都需要更改特权。如果您预测到这种需要，则可准备一些自动化工具来帮助维护特权。

第一步应该是指定基于用户的工作（而不是基于表的结构）的特权级。例如：第一线经理需要下列特权:

- 对虚构的 **hr_data** 表的“选择”和有限“更新”特权
- 对这个以及其它数据库的“连接”特权
- 对那些数据库中的数个表的一定程度的特权

当将某位经理提升到本部职位或派到现场办公室时，必须取消所有那些特权并授予新的一组特权。

定义所支持的特权级，并对每一级指定必须给定存取权的数据库、表和列。然后，为每一级设计两个自动化例程，一个用于将该特权级授予用户，另一个用于取消该特权级。

通过命令脚本自动化

操作系统可能支持自动执行命令脚本。在大多数操作环境中，交互式 SQL 工具（如 DB–Access）接受从命令行执行命令和 SQL 语句。可以将这两项功能组合起来以自动进行特权维护。

详细信息取决于操作系统以及您正在使用的交互式 SQL 工具的版本。必须创建执行下列功能的命令脚本:

- 获取将要作为其参数而更改其特权的用户标识
- 准备 GRANT 或 REVOKE 语句的文件，将这些语句定制为包含该用户标识
- 调用交互式 SQL 工具（如 DB–Access），并指定参数以通知该工具选择数据库并执行已准备好的 GRANT 或 REVOKE 语句文件

这样，就可以将用户的特权级的更改减少到一两个命令。

使用角色

另一种避免逐个实例地更改用户特权这种困难的方法是使用角色。数据库环境中的角色概念与操作系统中的组概念类似。角色是一项数据库功能，它允许 DBA 通过将许多用户视为某个类的成员来将他们的特权标准化和进行更改。

例如: 可创建名为 *news_mes* 的角色，它授予用于处理公司新闻和消息的数据库的连接、插入和删除特权。当有新雇员来到时，只需要将那个人添加至 *news_mes* 角色。这个新雇员将获得 *news_mes* 角色的特权。此过程在相反方向上也能起作用。要更改 *news_mes* 的所有成员的特权，请更改该角色的特权。

创建角色: 要开始角色创建过程，请确定角色的名称以及要授予拥有该角色的用户的连接和特权。尽管连接和特权的确在您的域中，但您在声明新角色名称时仍需要考虑某些因素。请勿将下列任何 SQL 关键字用作角色名称:

alter	delete	insert	references
connect	execute	none	resource
DBA	extend	null	select
default	index	public	update

角色名一定不能与数据库中的现有角色名相同。角色名也一定不能与操作系统已知的用户名（包括服务器计算机已知的网络用户）相同。要确保角色名是唯一的，请检查共享内存结构中当前正在使用数据库以及下列系统目录表的用户的名称:

- **sysusers**
- **systabauth**
- **syscolauth**
- **sysfragauth**
- **sysprocauth**
- **sysfragauth**
- **sysroleauth**
- **sysxtdtypeauth**

在相反情况下，在将用户添加至数据库时，请检查用户名是否与任何现有角色名不相同。

在认可角色名之后，使用 CREATE ROLE 语句来创建新角色。创建角色后，缺省情况下将把所有用于角色管理的特权授予 DBA。

要点: 角色的作用域仅是当前数据库，因此，执行 SET ROLE 语句时，只在当前数据库中设置角色。

处理用户特权和将角色特权授予其它角色: 作为 DBA，您可以使用 GRANT 语句来将角色特权授予用户。还可以向用户提供将特权授予其他用户的选项。使用 GRANT 语句的 WITH GRANT OPTION 子句来执行此操作。将特权授予角色时，也可以如本例所示使用 WITH GRANT OPTION 子句:

```
GRANT rol1 TO usr1 WITH GRANT OPTION;
```

在授予角色特权时，可使用角色名来替代 GRANT 语句中的用户名。可以将角色特权授予另一个角色。例如: 假定已将角色 A 的特权授予角色 B。当用户启用角色 B 时，用户将同时获得角色 A 和角色 B 的特权。

然而，角色授权循环不能是传递的。如果将角色 A 的特权授予角色 B，并将角色 B 的特权授予角色 C，则将角色 C 的特权授予 A 将返回错误。

如果需要更改特权，则使用 REVOKE 语句来删除现有特权，然后使用 GRANT 语句来添加新特权。

启用缺省角色和非缺省角色: DBA 授予特权并将用户添加至角色之后，有两种可启用角色的方式。

- DBSA 可以使用 GRANT DEFAULT ROLE 语句为 PUBLIC 或个别用户指定*缺省角色*。用户连接到数据库时，该角色会自动激活为初始角色设置。
- 用户拥有的任何角色也都可以激活，方法是用户在 SET ROLE 语句中指定该角色。

启用角色时，已授予角色的所有特权以及显式授予您或 PUBLIC 的所有特权都成为可用的。

先将特权分配给角色，然后将该角色授权为一些指定用户的缺省角色，如果这样做的话，则对于那些用户从中运行应用程序（需要一组特定存取特权）的会话来讲是非常方便的。如果对应用程序进行重新编译（使其包含将必要的存取特权具体分配给用户的 GRANT 和 SET ROLE 语句）是不现实的，则请使用缺省角色。

确认角色中的成员资格和删除角色: 您自己会发现，在某些情况下不确定哪个用户包括在某个角色中。可能是您没有创建该角色，也可能是该角色的创建人员不可用。对 **sysroleauth** 和 **sysusers** 系统目录表发出查询以了解谁对哪个表拥有权限以及存在多少个角色。

在确定哪些用户拥有哪些角色之后，您可能会发现一些角色不再有用。要除去角色，请使用 DROP ROLE 语句。在除去角色之前，必须符合下列条件:

- 只能破坏 **sysusers** 系统目录表中列示为角色的那些角色，但是不能删除内建角色（例如 NONE 或 EXTEND）。
- 您必须具有 DBA 特权，否则必须向您提供角色中的可授予选项才能删除角色。

确定运行时的当前角色

如果对已经授予适当特权的角色遇到意外的特权错误，请确保在运行时已启用了该角色。要在连接到数据库时获取该信息，可以使用 **onstat -g sql** 或 **onstat -g ses** 命令，或者可以调用 SQL 的 **CURRENT_ROLE**( ) 函数或 **DEFAULT_ROLE**( ) 函数。

使用 SPL 例程来控制对数据的存取权

可以使用 SPL 例程来控制对数据库中的个别表和列的存取权。使用例程来进行各种程度的访问控制。SPL 的一项强大功能是有能力将 SPL 例程指定成具有 DBA 特权的例程。如果编写具有 DBA 特权的例程，则可以允许具有很少或不具有表特权的用户在执行该例程时具有 DBA 特权。在例程中，用户可以使用他们的临时 DBA 特权来执行特定任务。具有 DBA 特权的例程允许您完成下列任务:

- 可以限制个别用户可以从表中读取的信息量。
- 可以限制对数据库所作的所有更改并确保不会意外地将整个表清空或更改。
- 可以监视对表所作的整类更改，如删除或插入。
- 可以将所有对象创建（数据定义）操作限制为只在 SPL 例程中发生，以便对表、索引和视图的构建方式进行全面控制。

有关使用 SPL 编写的例程的信息，请参阅*《IBM Informix: SQL 教程指南》*。

限制数据读取

以下示例中的例程对用户隐藏 SQL 语法，但它要求用户对 **customer** 表具有“选择”特权。如果要限制用户可以选择的内容，则将例程编写为能够在以下环境中工作:

- 您是数据库的 DBA。
- 用户对数据库具有“连接”特权。他们对表不具有“选择”特权。

- 使用 DBA 关键字来创建 SPL 例程（或一组 SPL 例程）。
- SPL 例程（或一组 SPL 例程）为用户读取表。

如果要让用户只读取客户的名称、地址和电话号码，则可将此过程修改为如以下示例所示:

```
CREATE DBA PROCEDURE read_customer(cnum INT)
RETURNING CHAR(15), CHAR(15), CHAR(18);

DEFINE p_lname,p_fname CHAR(15);
DEFINE p_phone CHAR(18);

SELECT fname, lname, phone
   INTO p_fname, p_lname, p_phone
   FROM customer
   WHERE customer_num = cnum;

RETURN p_fname, p_lname, p_phone;

END PROCEDURE;
```

限制对数据的更改

使用 SPL 例程时，可以限制对表所作的更改。通过 SPL 例程来引导所有更改。SPL 例程进行更改，而不是由用户直接进行更改。如果要将用户限制为每次只能删除一行以确保他们不会意外除去表中所有的行，则使用下列特权来设置数据库:

- 您是数据库的 DBA。
- 所有用户对数据库都具有“连接”特权。他们可以具有“资源”特权。对于本示例，他们对正在受保护的表不具有“删除”特权。
- 使用 DBA 关键字来创建 SPL 例程。
- SPL 例程执行删除操作。

编写类似于以下的 SPL 过程，此过程使用带有用户提供的 **customer_num** 的 WHERE 子句来从 **customer** 表中删除行:

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DELETE FROM customer
   WHERE customer_num = cnum;

END PROCEDURE;
```

监视对数据的更改

使用 SPL 例程时，可以创建对数据库所作的更改的记录。可以记录由特定用户所作的更改，也可以在每次进行更改时作记录。

可以监视单个用户对数据库所作的所有更改。通过用于跟踪每个用户所作的更改的 SPL 例程来引导所有更改。如果要在用户 **acctclrk** 每次修改数据库时作记录，请使用下列特权来设置数据库:

- 您是数据库的 DBA。
- 所有其他用户对数据库都具有“连接”特权。他们可以具有“资源”特权。对于本示例，他们对正在受保护的表不具有“删除”特权。
- 使用 DBA 关键字来创建 SPL 例程。
- SPL 例程执行删除操作并记录特定用户作了更改。

编写类似于以下示例的 SPL 例程（针对 UNIX 平台），此例程使用用户提供的客户号来更新表。如果用户刚好是 **acctclrk**，则将在文件 **updates** 中放置删除记录。

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
   WHERE customer_num = cnum;

IF username = 'acctclrk' THEN
   SYSTEM 'echo Delete from customer by acctclrk >>
/mis/records/updates' ;
END IF
END PROCEDURE;
```

要监视通过此过程进行的所有删除操作，请除去 IF 语句并使 SYSTEM 语句更具一般性。以下过程将先前例程更改为记录所有删除操作:

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);
LET username = USER ;
DELETE FROM tbname WHERE customer_num = cnum;

SYSTEM
 'echo Deletion made from customer table, by '||username
||'>>/hr/records/deletes';

END PROCEDURE;
```

限制对象创建操作

要对构建哪些对象以及如何构建它们加以约束，请在下列设置中使用 SPL 例程:

- 您是数据库的 DBA。
- 所有其他用户对数据库都具有“连接”特权。他们不具有“资源”特权。
- 使用 DBA 关键字来创建 SPL 例程（或一组 SPL 例程）。

- SPL 例程（或一组 SPL 例程）以您定义表、索引和视图的方式来创建它们。可使用这样的例程来设置培训数据库环境。

SPL 例程可以包括创建一个或多个表和相关联的索引，如以下示例所示:

```
CREATE DBA PROCEDURE all_objects()

CREATE TABLE learn1 (intone SERIAL, inttwo INT NOT NULL,
   charcol CHAR(10) );
CREATE INDEX learn_ix ON learn1 (inttwo);
CREATE TABLE toys (name CHAR(15) NOT NULL UNIQUE,
   description CHAR(30), on_hand INT);
END PROCEDURE;
```

要使用 **all_objects** 过程来控制将列添加至表的操作，请取消所有用户对数据库的“资源”特权。当用户尝试在此过程外部通过 SQL 语句来创建表、索引或视图时，他们无法完成该操作。当用户执行该过程时，他们具有临时的 DBA 特权，因此 CREATE TABLE 语句（这只是一个示例）会成功，并且可以保证对添加的每个列都加以约束。另外，用户创建的对象由那些用户拥有。对于 **all_objects** 过程，任何执行该过程的人都拥有两个表和索引。

使用视图

*视图*是合成的表。可以对其进行查询（就象它是一个表一样），在某些情况下，可以更新它（就象它是一个表一样）。然而，它并不是表。它是存在于真实的表以及其它视图中的数据的合成。

视图的基础是 SELECT 语句。创建视图时，定义一个 SELECT 语句，该语句在您访问视图时生成视图的内容。用户还使用 SELECT 语句来查询视图。在某些情况下，数据库服务器将用户的 SELECT 语句和对视图定义的 SELECT 语句合并，然后实际地执行经过组合的语句。有关视图性能的信息，请参阅 *《IBM Informix: 性能指南》*。

由于您编写确定视图内容的 SELECT 语句，所以可以使用视图来实现下列任何目的:

- 限制用户只能存取表的特定列

  您在视图的选择列表中只命名所允许的列。
- 限制用户只能存取表的特定行

  指定只返回所允许的行的 WHERE 子句。
- 将插入的和更新的值约束为具有特定范围

  可使用 WITH CHECK OPTION（在第 6-24 页讨论）来强制执行约束。

- 在不必在数据库中存储冗余数据的情况下提供对派生数据的存取权。

  编写表达式，这些表达式派生数据到视图中的选择列表中。每次查询视图时都重新派生数据。派生数据总是最新的，然而不会在数据模型中引入冗余。
- 隐藏复杂的 SELECT 语句的详细信息

  将多表连接的复杂性隐藏在视图中，以使用户和应用程序程序员都不需要重复它们。

创建视图

以下示例创建一个基于 **stores_demo** 数据库中的表的视图:

```
CREATE VIEW name_only AS
   SELECT customer_num, fname, lname FROM customer
```

此视图只显示该表的三个列。由于它不包含 WHERE 子句，所以视图不对可以出现的行加以限制。

以下示例基于两个表的连接:

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname,
   zipcode, customer_num
   FROM customer, state
   WHERE customer.state = state.code
```

州名表降低了数据库的冗余度；它允许只将完整的州名存储一次，这对于 Minnesota 之类的较长州名可能非常有用。这个 **full_addr** 视图允许用户检索地址，就象每一行都存储了完整的州名一样。下列两个查询是等价的:

```
SELECT * FROM full_addr WHERE customer_num = 105

SELECT address1, address2, city, state.sname,
   zipcode, customer_num
   FROM customer, state
   WHERE customer.state = state.code AND customer_num = 105
```

然而，定义基于连接的视图时务必小心谨慎。这样的视图不是*可修改的*；即，不能对它们使用 UPDATE、DELETE 或 INSERT 语句。有关如何修改视图的讨论，请参阅第 6-22 页。

以下示例对可以在视图中查看的行加以限制:

```
CREATE VIEW no_cal_cust AS
   SELECT * FROM customer WHERE NOT state = 'CA'
```

此视图显示 **customer** 表的所有列，但只显示特定的行。以下示例是一个视图，它限制用户只能查看与他们相关的行:

```
CREATE VIEW my_calls AS
   SELECT * FROM cust_calls WHERE user_id = USER
```

cust_calls 表的所有列都可用，但只显示那些包含可执行查询的用户的用户标识的行中的列。

带类型视图（IDS）

当要对两个显示同一数据类型的数据的视图加以区分时，可创建带类型视图。例如：假定要对下面这个表创建两个视图：

```
CREATE TABLE emp
(  name    VARCHAR(30),
   age     INTEGER,
   salary  INTEGER);
```

下列语句对 **emp** 表创建两个带类型视图 **name_age** 和 **name_salary**：

```
CREATE ROW TYPE name_age_t
(  name    VARCHAR(20),
   age    INTEGER);

CREATE VIEW name_age OF TYPE name_age_t AS
   SELECT name, age FROM emp;

CREATE ROW TYPE name_salary_t
(  name    VARCHAR(20),
   salary  INTEGER);

CREATE VIEW name_salary OF TYPE name_salary_t AS
   SELECT name, salary FROM emp
```

创建带类型视图时，该视图显示的数据具有命名行类型。例如：**name_age** 和 **name_salary** 视图包含 VARCHAR 和 INTEGER 数据。由于视图具有类型，所以对 **name_age** 视图执行的查询返回类型为 **name_age** 的列视图，而对 **name_salary** 视图执行的查询返回类型为 **name_salary** 的列视图。因此，数据库服务器能够对 **name_age** 和 **name_salary** 视图返回的行加以区分。

在某些情况下，带类型视图具有无类型视图所不具有的优点。例如：假定按如下方式重载 **myfunc()** 函数：

```
CREATE FUNCTION myfunc(aa name_age_t) ......;
CREATE FUNCTION myfunc(aa name_salary_t) .....;
```

由于 **name_age** 和 **name_salary** 视图是带类型视图，所以下列语句将解析为适当的 **myfunc()** 函数：

```
SELECT myfunc(name_age) FROM name_age;
SELECT myfunc(name_salary) FROM name_salary;
```

还可以使用表名的别名来编写上述 SELECT 语句：

```
SELECT myfunc(p) FROM name_age p;
SELECT myfunc(p) FROM name_salary p;
```

如果不将两个包含相同数据类型的视图创建为带类型视图，则数据库服务器无法对这两个视图显示的行加以区分。有关函数重载的更多信息，请参阅*《IBM Informix: 用户定义的例程与数据类型开发者指南》*。

视图中的重复行

即使基础表只包含唯一的行，视图也可能会生成重复的行。如果视图的 SELECT 语句可以返回重复的行，则视图本身可以显示为包含重复的行。

可通过两种方法防止此问题。一种方法是在视图中的投影列表中指定 DISTINCT。但是，当指定 DISTINCT 时，不可能修改视图。备用方法是始终选择已约束为必须唯一的列或一组列。（如果选择主键或候选键的列，则可以确保只返回唯一的行。第 2 章讨论了主键和候选键。）

对视图的限制

由于视图实际上不是表，所以不能对其建立索引，并且它也不能是 ALTER TABLE 和 RENAME TABLE 之类的语句的目标。不能使用 RENAME COLUMN 来将视图的列重命名。要更改任何关于视图的定义的内容，必须删除该视图并重新创建它。

由于必须与用户的查询合并，所以视图所基于的 SELECT 语句不能包含下列子句或关键字:

INTO TEMP　用户的查询可包含 INTO TEMP; 如果视图也包含它，则无法确定数据的目的地。

ORDER BY　用户的查询可包含 ORDER BY。如果视图也包含它，则对列或排序方向的选择可能会有冲突。

视图所基于的 SELECT 语句可以包含 UNION 关键字。在此类情况下，数据库服务器将视图存储在隐式临时表中，在该处，根据需要对联合进行求值。用户的查询将此临时表用作基本表。

当基础更改时

可以通过数种方法来更改视图所基于的表和视图。视图将自动反映大部分的更改。

当删除表或视图时，将自动删除同一数据库中依赖于该表或视图的任何视图。

唯一一种改变视图定义的方法是删除并重新创建该视图。因此，如果更改其它视图所依赖于的视图的定义，则还必须重新创建那些视图（这是因为它们将被全部删除）。

当将表重命名时，将把同一数据库中的任何依赖于该表的视图修改为使用新名称。当将列重命名时，将把同一数据库中的依赖于该表的视图更新为选择适当的列。然而，视图本身中的列的名称不更改。有关示例，请再调用基于 **customer** 表的以下视图:

```
CREATE VIEW name_only AS
   SELECT customer_num, fname, lname FROM customer
```

现在，假定按以下方式更改了 **customer** 表:

```
RENAME COLUMN customer.lname TO surname
```

要直接选择客户的姓，现在必须选择新列名。然而，从视图中看来，列的名称没有更改。下列两个查询是等价的:

```
SELECT fname, surname FROM customer

SELECT fname, lname FROM name_only
```

当删除列以改变表时，不会修改视图。如果使用那些视图，则会发生错误 -217（在查询的任何表中都找不到列）。不修改视图的原因是可以通过删除列并接着添加同名的新列来更改表中的列顺序。如果这样做，则基于该表的视图能够继续工作。它们将保留它们的原始列顺序。

数据库服务器允许视图基于外部数据库中的表和视图。对其它数据库中的表和视图所作的更改不会反映在视图中。在某人查询视图并且由于外部表已更改而出错之前，这样的更改可能不明显。

修改视图

您可以修改视图，就象它们是表一样。某些视图可以修改，而另一些则不能修改，这取决于它们的 SELECT 语句。限制是不相同的，这取决于您使用的是 DELETE、UPDATE 还是 INSERT 语句。

如果用于定义视图的 SELECT 语句没有包含下列任何一项，则该视图是*可修改的*:

- 两个或更多个表的连接
- 聚集函数或 GROUP BY 子句
- DISTINCT 关键字或它的同义词 UNIQUE
- UNION 关键字

当视图避免了所有这些受限特征时，视图的每一行都刚好与一个表的一行相对应。

使用视图进行删除

可以对可修改视图使用 DELETE 语句，就象它是表一样。数据库服务器将删除基础表的适当行。

更新视图

可以对可修改视图使用 UPDATE 语句。然而，数据库服务器不支持更新任何派生列。*派生列*是由 CREATE VIEW 语句的选择列表中的表达式（例如 `order_date + 30`）生成的列。

以下示例显示了一个可修改视图，它包含一个派生列和对该派生列可以接受的 UPDATE 语句:

```
CREATE    VIEW   response(user_id, received, resolved, duration) AS
   SELECT user_id, call_dtime, res_dtime, res_dtime - call_dtime
      FROM cust_calls
      WHERE user_id = USER;

UPDATE response SET resolved = TODAY
   WHERE resolved IS NULL;
```

由于视图的 duration 列表示一个表达式，所以不能更新该列（数据库服务器无法决定如何在表达式命名的两个列之间分布更新值，即使在理论上也无法这样做）。但只要没有在 SET 子句中命名派生列，就可以执行更新，就象视图是表一样。

即使基础表的行是唯一的，视图也可能会返回重复的行。无法对各重复行加以区分。如果更新一组重复行中的其中一行（例如: 如果使用游标来更新 WHERE CURRENT），则无法确定基础表中的哪一行接收到更新。

插入到视图中

仅当视图为可修改*并且*不包含派生列时才可将行插入到视图中。第二项限制的原因是插入的行必须为所有的列提供值，但数据库服务器无法指出如何通过表达式来对插入的值进行分布。如前一个示例所示，尝试插入到 **response** 视图中将失败。

当可修改的视图不包含派生列时，可对其进行插入，就象它是表一样。然而，对于视图未显示的任何列，数据库服务器使用 NULL 作为那些列的值。如果这样的列不允许 NULL 值，则会出错，并且插入失败。

在 Dynamic Server 视图（包括复杂视图）中插入列（或执行 UPDATE 或 DELETE 操作）的另一种机制是创建 INSTEAD OF 触发器，如《*IBM Informix SQL 指南: 语法*》所述。

使用 WITH CHECK OPTION 关键字

可以将不满足视图条件的行（也就是无法通过视图进行查看的行）插入到视图中。还可以更新视图的行，以使其不再满足视图的条件。

要避免将视图的行更新为使其不再满足视图的条件，请在创建视图时添加 WITH CHECK OPTION 关键字。这个子句要求数据库服务器对插入或更新的每一行进行测试，以确保它符合视图的 WHERE 子句设置的条件。如果不符合条件，则数据库服务器将拒绝该操作，并显示错误。

要点: 当视图定义中包含 UNION 运算符时，不能包括 WITH CHECK OPTION 关键字。

在前一个示例中，名为 **response** 的视图是按以下示例所示的方式定义的:

```
CREATE   VIEW   response (user_id, received,  resolved,  duration) AS
   SELECT user_id,call_dtime,res_dtime,res_dtime  - call_dtime
      FROM cust_calls
      WHERE user_id = USER
```

可以更新视图的 **user_id** 列，如以下示例所示:

```
UPDATE response SET user_id = 'lenora'
   WHERE received BETWEEN TODAY AND TODAY - 7
```

视图需要 **user_id** 等于 USER 的行。如果用户 **tony** 执行此更新，则更新后的行将从视图中消失。然而，可以如下例所示创建视图:

```
CREATE   VIEW    response  (user_id, received, resolved,duration) AS
   SELECT user_id, call_dtime, res_dtime, res_dtime - call_dtime
      FROM cust_calls
      WHERE user_id = USER
WITH CHECK OPTION
```

用户 **tony** 所作的上述 UPDATE 操作被当作错误而受到拒绝。

可使用 WITH CHECK OPTION 功能来强制执行任何类型的可作为布尔表达式陈述的数据约束。在以下示例中，您可以创建表的一个视图，对于此视图，将对数据的所有逻辑约束都作为 WHERE 子句的条件进行表达。然后，可以要求对该表所作的所有修改都通过该视图进行。

```
CREATE VIEW order_insert AS
   SELECT * FROM orders O
      WHERE order_date = TODAY -- no back-dated entries
         AND EXISTS -- ensure valid foreign key
            (SELECT * FROM customer C
               WHERE O.customer_num = C.customer_num)
         AND ship_weight < 1000 -- reasonableness checks
         AND ship_charge < 1000
WITH CHECK OPTION
```

由于需要执行 EXISTS 和其它测试（当数据库服务器检索现有的行时，期望这些测试成功），此视图显示 **orders** 中的数据的效率并不高。然而，如果对 **orders** 执行的插入操作只通过此视图进行（并且还没有使用完整性约束来约束数据），则用户无法插入延期订单、无效的客户号或过高的装运重量和运费。

当视图定义更改时重新执行已准备好的语句

数据库服务器使用您在对某个视图准备 SELECT 语句时已存在的视图定义。如果在对某个视图准备 SELECT 语句后该视图的定义发生更改，则由于所准备的语句没有反映新的视图定义，所以执行它将生成不正确的结果。不会生成 SQL 错误。

特权和视图

*创建*视图时，数据库服务器将测试您对基础表和视图具有的特权。*使用*视图时，只测试您具有的与该视图相关的特权。

创建视图时的特权

数据库服务器将进行测试，以确保您具有执行视图定义中的 SELECT 语句所需的所有特权。如果您不具有那些特权，则数据库服务器不创建视图。

此项测试确保用户无法通过对某个表创建视图并查询该视图来获取对该表的未经授权的存取权。

创建视图后，数据库服务器至少授予您（您是该视图的创建者和所有者）对它的“选择”特权。不会对 PUBLIC 进行自动授权，这与新创建的表的情况相同。

数据库服务器将测试视图定义以了解视图是否是可修改的。如果是的话，数据库服务器将授予您对该视图的“插入”、“删除”和“更新”特权（如果您对基础表或视图也具有那些特权的话）。换而言之，如果新视图是可修改的，则数据库服务器将从基础表或视图复制您的“插入”、“删除”和“更新”特权并对新视图授予那些特权。如果您对基础表只具有“插入”特权，则您将只接收到对视图的“插入”特权。

此项测试确保用户无法使用视图来获取对他们尚未具有的任何特权的存取权。

由于无法改变视图或对视图建立索引，所以永远不会授予对视图的“改变”和“索引”特权。

使用视图时的特权

当您尝试使用视图时，数据库服务器只测试已授予您的对该视图的特权。它不测试您是否具有存取基础表的权限。

如果视图由您创建，则您具有前一节记载的特权。如果您不是创建者，则您具有创建者（或某个具有 WITH GRANT OPTION 特权的人）授予您的特权。

因此，可以创建一个表并撤销对它的 PUBLIC 存取权；然后，可通过视图对该表授予有限存取特权。假定要授予对下面这个表的存取特权:

```
CREATE TABLE hr_data
   (
   emp_key INTEGER,
   emp_name CHAR(40),
   hire_date DATE,
   dept_num SMALLINT,
   user-id CHAR(18),
   salary DECIMAL(8,2),
   performance_level CHAR(1),
   performance_notes TEXT
   )
```

第 6-8 页的『列级别特权』一节显示了如何直接授予对 **hr_data** 表的存取特权。下列示例采用另一种方法。假定创建该表时执行了以下语句:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

（在符合 ANSI 的数据库中，这样的语句不是必需的。）现在，为不同的用户类创建一系列视图。对于应该对非敏感列具有只读存取权的用户，创建以下视图:

```
CREATE VIEW hr_public AS
   SELECT emp_key, emp_name, hire_date, dept_num, user_id
      FROM hr_data
```

对此视图拥有“选择”特权的用户可以查看不敏感的数据，但不能进行更新。对于必须输入新行的“人力资源”人员，创建另一个视图，如以下示例所示:

```
CREATE VIEW hr_enter AS
   SELECT emp_key, emp_name, hire_date, dept_num
      FROM hr_data
```

将对此视图的“选择”和“插入”特权授予这些用户。由于您（您同时是表和视图的创建者）对表和视图具有“插入”特权，所以您可以将对该视图的“插入”特权授予其他对该表不具有特权的人。

对于 MIS 部门中输入或更新新用户标识的人员，还创建另一个视图，如以下示例所示:

```
CREATE VIEW hr_MIS AS
   SELECT emp_key, emp_name, user_id
      FROM hr_data
```

此视图与前一视图的不同点在于，它不显示部门号和聘用日期。

最后，经理需要存取所有的列，并且他们需要能够只更新他们自己的雇员的工作表现评价数据。可通过创建 **hr_data** 表来满足这些需求，该表包含每个雇员的部门号和计算机用户标识。让经理作为他们所管理的部门的成员作为一条规则。于是，以下视图将限制经理只能存取只反映他们的雇员的行:

```
CREATE VIEW hr_mgr_data AS
   SELECT * FROM hr_data
      WHERE dept_num =
         (SELECT dept_num FROM hr_data
            WHERE user_id = USER)
      AND NOT user_id = USER
```

最后一个条件是必需的，这样经理才不具有对表中他们自己的行的更新存取权。因此，可以安全地将此视图的“更新”特权授予经理，但只授予对所选列的“更新”特权，如以下语句所示:

```
GRANT SELECT, UPDATE (performance_level, performance_notes)
   ON hr_mgr_data TO peter_m
```

第 7 章 使用分布式查询

在本章中

本章提供对分布式查询的概述。分布式查询允许在 IBM Informix 数据库服务器网络内跨多个数据库对数据进行共享式存取。不同的数据库服务器可以管理多个数据库，可以在单独的分布式查询中引用这些数据库。

分布式查询概述

IBM Informix 数据库服务器允许查询同一个数据库服务器或跨多个数据库服务器的多个数据库。这类查询称为*分布式查询*。数据库服务器可以驻留在单独的主机上、同一个网络中的不同计算机上或网关上。(通常，本章描述的分布式查询的大多数功能和限制都适用于函数调用，也适用于在多个数据库中引用对象或数据的分布式 INSERT、DELETE 或 UPDATE 操作。

注: IBM Informix Extended Parallel Server V8.40 仅支持参与者功能。该版本无法进行分布式查询。IBM Informix Extended Parallel Server V8.50 既支持参与者功能也支持协调者功能。各限制仍适用。

跨单个 Dynamic Server 实例的多个数据库进行分布式查询

跨单个 IBM Informix Dynamic Server 实例的多个数据库进行分布式操作应该遵循以下对返回数据类型的限制:

- 查询、DML 操作或函数调用可以返回任何内置数据类型，包括 BLOB、BOOLEAN、CLOB 和 LVARCHAR 内置不透明类型。
- 查询、DML 操作或函数调用不能返回 DISTINCT 或 OPAQUE 数据类型，除非是对内置数据类型的显式数据类型转换，并且所有 DISTINCT 和 OPAQUE 数据类型及所有显式数据类型转换都在存储或接受这些数据类型的每个参与数据库中进行了定义。

分布式查询中的协调者和参与者

为了支持跨多个数据库服务器进行分布式操作，IBM Informix 服务器保持着由一个协调者和一个或多个参与者构成的层次结构关系。对协调者和参与者的定义如下:

- 协调者对查询的解决方案提供指导。它还确定是否应该提交或放弃查询。
- 参与者指导一条分支上分布式查询的执行。该分支是分布式查询的一部分，只涉及该参与者数据库服务器。

以下示例涉及一个多服务器环境，其中 db 是本地数据库，db2 是驻留在同一服务器上的外部数据库，master_db 是远程服务器 new_york 上的外部数据库。

以下示例显示了一个查询，该查询可以用来使用数据库 db 作为协调者在另一台服务器上存取数据。

```
database db; select col1, col2 from db2:tab1, master_db@newyork:tab2;
```

一个会话将只有一个本地数据库，但是可以打开多个外部数据库。分布式查询必须始终对协调者生成。

使用分布式查询配制数据库服务器

要将多个 IBM Informix 服务器用于分布式查询，必须确保所有涉及的数据库服务器都配置为支持通过网络进行服务器到服务器通信。可能需要编辑以下配置文件以允许进行分布式查询:

- sqlhosts 文件
- onconfig 文件
- /etc/hosts.equiv 或 .rhosts
- /etc/services
- /etc/hosts

sqlhosts 文件包含每台数据库服务器的连接信息。要将几台数据库服务器设置为使用分布式查询，请使用以下的一种方法存储所有数据库的 sqlhosts 信息:

- 存储在一个 sqlhosts 文件中，由 INFORMIXSQLHOSTS 指向该文件
- 存储在多个 sqlhosts 文件中，每个文件位于每个数据库服务器目录下

注: IBM Informix Extended Parallel Server V8.40 仅支持参与者功能；该版本不能进行分布式查询。 IBM Informix Extended Parallel Server V8.50 既支持参与者功能也支持协调者功能。各限制仍适用。

有关配置 sqlhosts 文件的更多信息，请参阅《*管理员指南*》。

分布式查询的语法

本节描述了如何在分布式查询中指定远程服务器、数据库和数据库对象。

注: 设计分布式查询时，请注意某些 SQL 语法并不对所有服务器版本都有效。在 Dynamic Server 上有效但 Extended Parallel Server 上无效的语法，在 Extended Parallel Server 上是不受支持的，反之亦然。

由于这些潜在的语法不兼容性问题，SQL 语句有可能会通过协调者上的检查阶段，但是却在语句传递到参与者的时候返回错误。

访问远程服务器和数据库

分布式查询中任何语句的核心元素都是数据库段。使用这些段的语法可以指定远程数据库服务器、数据库或数据库对象。

数据库名称

“数据库名称”段用来指定数据库的名称。以下示例显示了指定远程数据库的不同方法:

```
empinfo@personnel '//personnel/empinfo'
```

数据库对象名称

“数据库对象名称”段用来指定数据库对象的名称，包括约束、索引、触发器和任何同义词。以下示例显示了如何存取远程对象:

```
empinfo@personnel:markg.emp_names empinfo@personnel:emp_names
```

指定协同服务器标识（XPS）

如果在运行的分布式查询中 Extended Parallel Server 既是协调者又是参与者，则可以将协同服务器指定为数据库和数据库对象段的一部分。以下示例显示了如何指定协同服务器标识:

```
orders@stores.2 empinfo@personnel.3:emp_names
```

注: 在任何给定的会话中，对远程服务器的第一次引用确定了必须如何指定对该服务器上对象的后续引用。一旦协同服务器标识用来限定服务器上的对象，则对同一服务器的后续引用（甚至对任何其它对象而言）都必须指定相同的协同服务器标识。用于不同远程服务器的协同服务器标识是独立的。

存取远程对象的有效语句

以下语句支持远程对象作为“数据库”和“数据库对象”段的一部分并且可以在分布式查询中使用:

- INSERT
- SELECT
- UPDATE
- DELETE
- CREATE VIEW
- CREATE SYNONYM
- CREATE DATABASE
- DATABASE
- LOAD
- UNLOAD
- LOCK
- UNLOCK
- INFO

对于 Extended Parallel Server（XPS）8.50，您不能在更改或添加数据的语句中涉及远程对象。对于实例而言，在远程对象上运行 INSERT、UPDATE 和 DELETE

语句是不受支持的。有关 Extended Parallel Server 不支持分布式查询的更多方面，请参阅第 7-7 页上的『跨服务器兼容性问题（XPS）』。

存取远程表

远程表是当前服务器以外的数据库服务器上的表。在另一服务器上存取表的通用语法是:

```
database@server:[owner.]table
```

此处 table 可以是表名称、视图名称或同义词。您可以选择指定表所有者。有关完整的语法选项，请参阅《*IBM Informix SQL 指南: 语法*》中关于“数据库”和“数据库对象”段的文档。

以下示例显示了存取远程表的查询:

```
DATABASE locdb; SELECT l.name, r.assignment FROM rdb@rsys:rtab r,
loctab l WHERE l.empid = r.empid;
```

该查询从本地表 loctab 中存取 name 和 empid 列，从远程表 rtab 中存取 assignment 和 empid 列。将 empid 用作连接列来连接数据。

以下示例显示了在远程表上存取数据并将数据插入本地表的查询:

```
DATABASE locdb; INSERT INTO loctab SELECT * FROM rdb@rsys:rtab;
```

该查询从远程表 rtab 中选择了所有数据，然后将数据插入本地表 loctab。

以下示例使用远程数据库 rdb 的 empid 和 priority 列在本地数据库中创建了一个视图。

```
DATABASE locdb; CREATE VIEW myview (empid, empprty)
AS SELECT empid, priority FROM rdb@rsys:rtab;
```

表许可权

存取其它数据库中的表以及远程表的许可权是在表所在位置进行控制的。存取远程服务器时会使用执行该查询的用户的登录名和密码来进行连接。要存取远程数据，用户必须具有远程表的适当许可权。

处理分布式查询时，数据库服务器会在存取远程对象时忽略当前本地数据库的活动角色。在远程服务器上会使用应用于每个远程数据库的缺省角色。如果没有定义缺省角色，用户的特权会对每个远程数据库中的对象定义存取许可权。

限定对表的引用

可以使用当前数据库和服务器名称来限定对表的引用。如果没有指定限定，则默认当前的数据库和服务器上下文。例如: 如果当前数据库是 locdb，当前服务器是 currsys，则对 loctab 的以下引用效果是相同的:

```
locdb@currsys:loctab
locdb:loctab
loctab
```

其它远程操作

除了查询和更新数据之外，您还可以使用分布式查询框架执行其它远程操作。

打开远程数据库

通过在 DATABASE 语句中指定远程对象，可以打开远程数据库，如下例所示:

```
DATABASE dbname@servername;
DATABASE "//servernam/database";
```

创建远程数据库

在使用 CREATE DATABASE 语句时利用服务器名称来限定数据库名称，可以创建远程数据库。

```
CREATE DATABASE remfoo@rsys;
```

创建远程同义词

可以在 CREATE SYNONYM 语句中使用限定名在另一个数据库或远程表中创建一个表的远程同义词。例如: 以下语句创建了 **rdb@srsys:rtab:** 的同义词。

```
CREATE SYNONYM myrtab FOR rdb@rsys:rtab;
```

同义词可以同时存在于本地和远程服务器上。在以上示例中，rtab 本身可以是 rdb2@rsys2:rtab2 的同义词。检索目录信息时会沿着同义词链查询，直到找到表所驻留的物理数据库和服务器。如果同义词最终指回其自身，则会返回一个错误。

监视分布式查询

使用 **onstat -x** 实用程序，可显示来自于分布式查询协调者的事务信息。

在位置 5 处的以下标志代码用于分布式查询:

C	分布式查询协调者
S	分布式查询参与者
B	分布式查询协调者和参与者
R	与远程对象引用的事务（XPS）

有关使用 onstat -x 的更多信息，请参阅《*管理员参考大全*》。

服务器环境和分布式查询

本节列出了影响分布式查询行为的配置参数和环境变量。

PDQPRIORITY 环境变量

建立连接时，会话的 PDQPRIORITY 的有效值会发送到远程站点。协调者中该参数的后续变化不会反映在远程站点上。然而，该环境变量的确切行为取决于分布式查询中数据库服务器的角色（协调者或参与者）。

PDQPRIORITY 对于不同的服务器版本具有不同的语法和语义。有关设置 PDQPRIORITY 的信息，请参阅服务器的《*性能指南*》。

DEADLOCK_TIMEOUT

该配置参数用来指定事务等待锁的时间。如果强制分布式事务等待的时间超过了指定的妙数，则拥有该事务的线程会假定存在多个服务器死锁。会返回以下错误消息:

```
-143 ISAM error: deadlock detected.
```

有关使用该配置参数的更多信息，请参阅《*管理员指南*》。

数据库访问限制

要在 Informix 数据库服务器环境中执行分布式查询，所有数据库都必须有符合 ANSI 的方式和日志记录。

分布式查询可以在不符合 ANSI 或符合 ANSI 的数据库中执行，但是每个参与数据库必须有相同的 ANSI 方式。换言之，分布式查询中涉及的所有数据库必须全部符合 ANSI，或者全部不符合 ANSI。只要它们是一致的，则分布式查询就可以在具有缓冲型日志记录或未缓冲型日志记录的数据库上执行。

Extended Parallel Server 不支持没有日志记录的数据库。因此，在 Extended Parallel Server 作为协调者或参与者的分布式查询中，所有数据库都必须进行日志记录。

事务处理

本节描述了在事务处理环境中使用分布式查询时涉及到的一些注意事项。

隔离级别

在远程站点的事务开始时，事务的隔离级别会发送到远程服务器。如果在事务过程中隔离级别发生了变化，则会将新值发送到远程站点。

DEADLOCK_TIMEOUT 和 SET LOCK MODE

使用分布式查询时，可以结合 DEADLOCK_TIMEOUT 配置参数使用 SET LOCK MODE 语句帮助防止服务器死锁。

请求 SET LOCK MODE 的 WAIT 选项时，数据库服务器会针对可能的死锁进行保护。然而，如果数据库发现可能出现死锁，则会终止操作并返回错误。

DEADLOCK_TIMEOUT 配置参数指定了数据库服务器线程等待获取锁的最大秒数。该值是 SET LOCK MODE WAIT 语句使用的缺省值。仅当在同一事务内获取当前数据库服务器和远程数据库服务器上的锁时该值才适用。

有关 SET LOCK MODE 语句的更多信息，请参阅《*IBM Informix SQL 指南: 语法*》。有关 DEADLOCK_TIMEOUT 配置参数的更多信息，请参阅第 7-7 页的『DEADLOCK_TIMEOUT』和《*IBM Informix Dynamic Server 管理员指南*》中关于多阶段落实协议的章节。

两阶段落实和恢复

两阶段落实协议用来确保分布式查询在多个数据库服务器之间统一提交或回滚。数据库服务器对任何在多个数据库服务器上修改数据的事务自动使用两阶段落实协议。

因为 Extended Parallel Server 不支持远程更新，所以不能进行单个事务内的多站点更新。因此，两阶段落实协议不适用于来自 Extended Parallel Server 的查询。在这种情况下，分布式事务被当作本地事务同样对待并且根据发生故障的点进行回滚和提交。需要使用两阶段落实协议的任何语句都会被放弃并且返回错误消息。

有关更多信息，请参阅《*IBM Informix Dynamic Server 管理员指南*》中关于多阶段落实协议的章节。

跨服务器兼容性问题（XPS）

本节列出了 Extended Parallel Server 不支持的分布式查询元素。

BYTE 和 TEXT 数据类型

存取远程表数据时不支持 BYTE 或 TEXT 数据类型。仅支持 Extended Parallel Server 的内置数据类型。基于用户定义的类型的 Dynamic Server 内置数据类型不受支持。

其它限制

Extended Parallel Server 使用分布式查询时有以下限制:

- 不支持远程存储过程。
- 触发器不能在触发器定义中引用远程对象。
- 不支持 IBM Informix 网关产品。

第 3 部分 对象关系数据库

第 8 章 在 Dynamic Server 中创建和使用扩展数据类型

在本章中

本章描述可以用来构建对象关系数据库的扩展数据类型。术语*对象关系*与数据库设计的特定方法或模型不相关，相反，它指的是任何使用 Dynamic Server 功能来扩展数据库功能的数据库。

对象关系数据库不是与关系数据库相对立的，相反，它是对关系数据库中已存在的功能的扩展。通常，使用 Dynamic Server 中的功能的某种组合来扩展数据库可以存储和处理的数据种类。这些功能包括扩展数据类型、智能大对象、类型和表继承、用户定义的数据类型转换和用户定义的例程（UDR）。手册这一部分中的各章描述了许多这些功能。有关 UDR 的信息，请参阅*《IBM Informix: 用户定义的例程与数据类型开发者指南》*和*《IBM Informix: SQL 教程指南》*。

要获取对象关系数据库的示例，可以创建 **superstores_demo** 数据库，该数据库包含 Dynamic Server 提供的一些功能的示例。有关如何创建 **superstores_demo** 数据库的信息，请参阅*《IBM Informix: DB–Access 用户指南》*。

Informix 数据类型

第 3-1 页的第 3 章，『选择数据类型』中的图 3-1 提供了根据将要存储的数据的类型来为表列选择适当数据类型的图。第 8-3 页的图 8-1 显示了数据类型的层次结构，此层次结构反映了数据库服务器管理数据类型的方式。

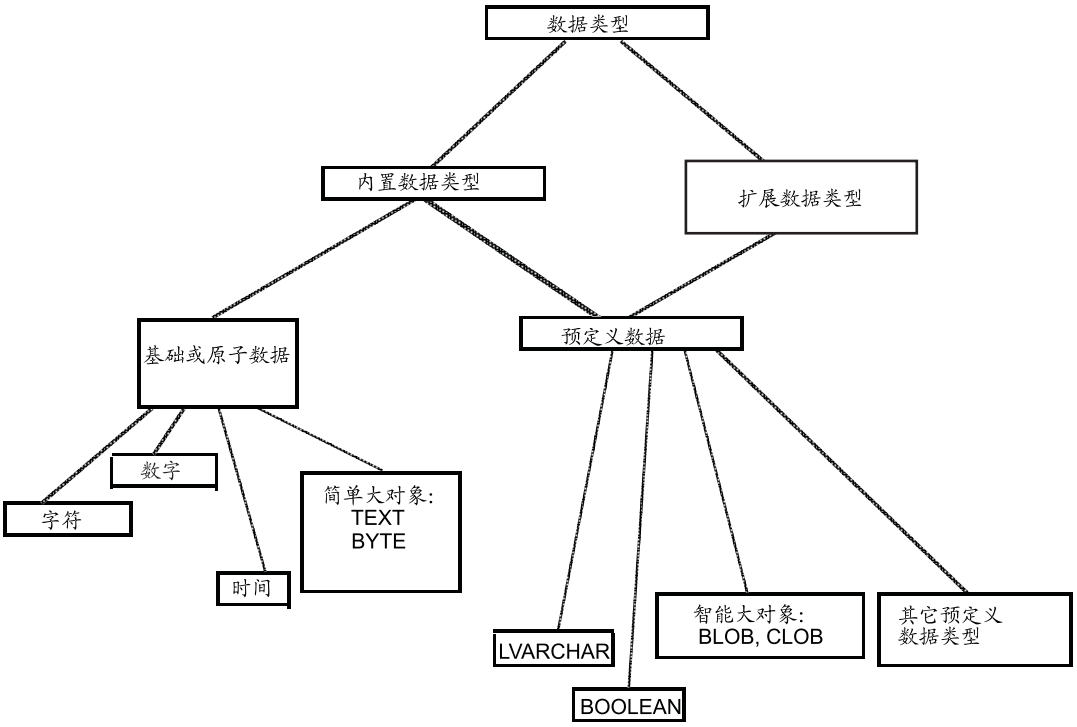


图 8-1. Informix 数据类型

基础或原子数据类型

所有 Informix 数据库服务器都支持*基础*或*原子*数据类型。因为这些类型是可以在 SELECT 语句中指定的最小单位，所以它们是基础类型。只有 Dynamic Server 才支持扩展数据类型和预定义数据类型。因为预定义数据类型与扩展数据类型共享某些特征，但它们是由数据库服务器提供的，所以它们位于独立的类别中。

有关基础数据类型的讨论，参阅第 3-1 页的第 3 章，『选择数据类型』。

预定义数据类型

就象提供了基础数据类型一样，数据库服务器还提供了预定义数据类型。但是，预定义数据类型与扩展数据类型具有某些公共特征。

BOOLEAN 和 LVARCHAR 数据类型

BOOLEAN 和 LVARCHAR 数据类型的行为与内置数据类型相似，但系统目录表将它们定义为扩展数据类型。

有关更多信息，请参阅第 3-1 页的第 3 章，『选择数据类型』以及《*IBM Informix: SQL 参考指南*》中的系统目录表。

BLOB 和 CLOB 数据类型

由于可以随机存取 BLOB 或 CLOB 中的数据，所以 BLOB 和 CLOB 数据类型不是基础数据类型。可以创建带有 BLOB 和 CLOB 列的表，但不能直接将数据插入该列中。必须使用函数才能插入和处理数据。

有关更多信息，请参阅第 8-6 页的『智能大对象』。

其它预定义数据类型

除 BLOB、BOOLEAN、CLOB 和 LVARCHAR 以外，预定义数据类型通常不作为表列的数据类型出现。相反，预定义数据类型是和这样的函数配合使用的：这些函数与复杂数据类型、用户定义的数据类型以及用户定义的例程相关联。下表列示了其余预定义数据类型。

clientbinval	indexkeyarray	sendrecv
ifx_lo_spec	lolist	stat
ifx_lo_stat	pointer	stream
impexp	rtnparamtypes	
impexpbin	selfuncargs	

有关这些预定义数据类型的更多信息，请参阅《*IBM Informix: 用户定义的例程与数据类型开发者指南*》。

扩展数据类型

扩展数据类型允许创建数据类型，来描述那些不太容易使用内置数据类型表示的数据的特征。但是，不能在分布式事务中使用扩展数据类型。图 8-2 显示了扩展数据类型。

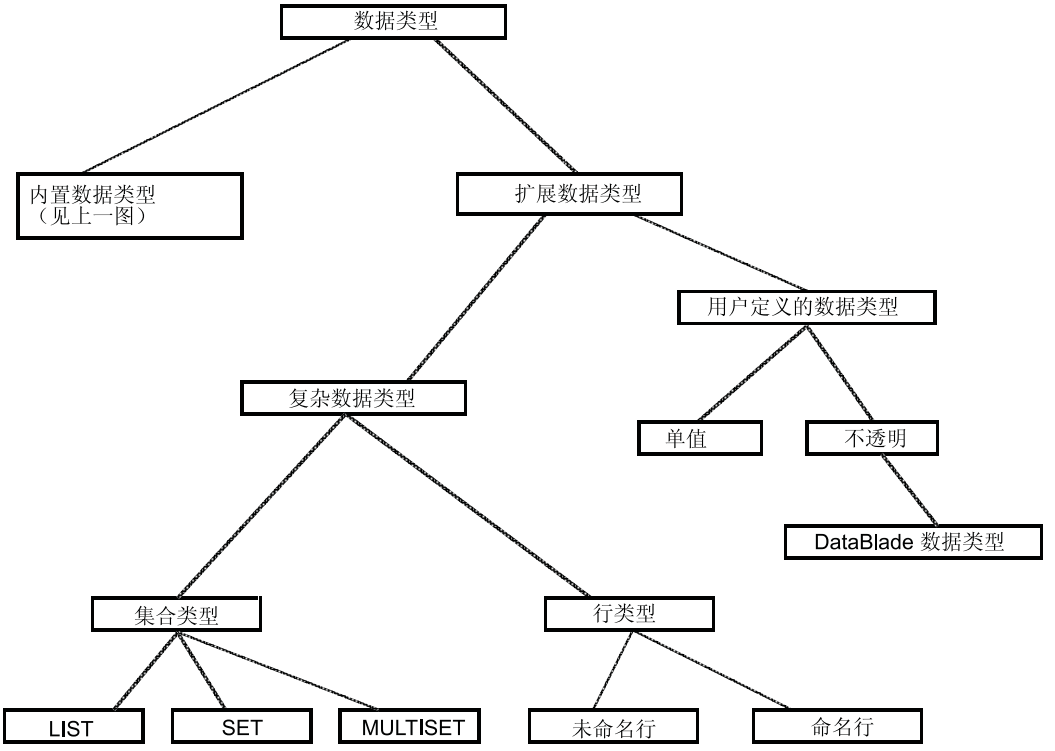


图 8-2. 扩展数据类型

复杂数据类型

复杂数据类型描述数据对象（这些数据对象全都是一种类型）的集合（LIST、SET 和 MULTISET）或不同类型的对象组（命名行和未命名行）。

用户定义的数据类型

用户定义的数据类型是并非由数据库服务器提供的数据类型。必须提供数据库服务器管理不透明数据类型或单值数据类型所需的所有信息。

单值数据类型

单值数据类型是使用 CREATE DISTINCT TYPE 语句创建的封装数据类型。单值数据类型与它所基于的数据类型具有相同的表示法，但却不同于该数据类型。可以根据内置类型、不透明类型、命名行类型或其它单值类型创建单值数据类型。不能根据下列任何数据类型创建单值数据类型:

- SERIAL
- SERIAL8

- 集合类型
- 未命名行类型

因为单值数据类型继承它的源数据类型的结构，所以创建单值数据类型时，将隐式地定义数据类型的结构。还可以定义对单值数据类型操作的函数、运算符和聚集。

有关单值数据类型的信息，请参阅第 10-10 页的『对单值数据类型进行数据类型转换』、*《IBM Informix: SQL 指南: 语法》*和*《IBM Informix: SQL 参考指南》*。

不透明数据类型

不透明数据类型是使用 CREATE OPAQUE TYPE 语句创建的封装数据类型。创建不透明数据类型时，必须显式地定义数据类型的结构以及对该不透明数据类型操作的函数、运算符和聚集。可以象使用内置类型那样使用不透明数据类型来定义列和程序变量。

有关创建不透明数据类型的信息，请参阅*《IBM Informix: 用户定义的例程与数据类型开发者指南》*和*《IBM Informix: SQL 指南: 语法》*。

DataBlade 数据类型

虽然 DataBlade 数据类型实际上不是数据类型，但第 8-5 页的图 8-2 中的图还是包括了它们。DataBlade 是一组用户定义的数据类型和用户定义的例程，它为专用应用程序提供工具。例如: 不同的 DataBlade 提供了用于管理图像、时序和天文观测资料的工具。此类应用程序经常需要不透明数据类型以及其它用户定义的数据类型。有关开发 DataBlade 的信息，请参阅 *IBM Informix: DataBlade API Programmer's Guide* 和 DataBlade Developers Kit。有关 IBM 提供的 DataBlade 的信息，请向客户代表咨询。

智能大对象

*智能大对象*是对 BLOB 或 CLOB 数据类型定义的对象。智能大对象允许应用程序随机存取列数据，这表示可以按任意顺序读或写 BLOB 或 CLOB 列的任何部分。可以创建 BLOB 或 CLOB 列来存储二进制数据或字符数据。

BLOB 数据类型

可以使用 BLOB 数据类型来存储程序可以生成的任何数据: 图形图像、卫星图像、视频剪辑、音频剪辑或者由任何字处理器或电子表格保存的格式化文档。在 BLOB 列中，数据库服务器允许任何类型的具有任何长度的数据。

与 CLOB 对象相似，BLOB 对象存储在与普通行数据分开的独立磁盘区域中的完整磁盘页中。

与 CLOB 相反，BLOB 数据类型的优点是它接受任何数据。在其它方面，BLOB 数据类型的优点和缺点与 CLOB 数据类型的优点和缺点相同。

CLOB 数据类型

可以使用 CLOB 数据类型来存储文本块。此数据类型用来存储 ASCII 文本数据，包括诸如 HTML 或 PostScript 之类的格式化文本。虽然可以在 CLOB 对象中存储任何数据，但是 IBM Informix 工具期望可打印的 CLOB 对象，所以将此数据类型限制为可打印的 ASCII 文本。

CLOB 值不与包含它们的行存储在一起。在完整的磁盘页（通常是与行分开的区域）中分配它们。（有关更多信息，请参阅 *《IBM Informix: 管理员指南》*。）

CLOB 数据类型与 TEXT 数据类型类似，但 CLOB 数据类型具有下列优点:

- 应用程序可以读写 CLOB 对象的任何部分。
- 由于应用程序可以访问 CLOB 对象的任何部分，所以访问速度可以显著提高。
- 缺省特征相对易于覆盖。数据库管理员能够在列级别覆盖 sbspace 的缺省特征。应用程序员可以在创建 CLOB 对象时覆盖列的某些缺省特征。
- 可以使用等于运算符（=）来测试两个 CLOB 值是否相等。
- 发生系统故障时，CLOB 对象是可恢复的，并且，如果 DBA 或应用程序员指定了事务隔离方式，则服从该隔离方式。（CLOB 对象的恢复要求数据库系统有必需的资源以提供足够大的缓冲区来处理 CLOB 对象。）
- 可以使用 CLOB 数据类型来为用户定义的数据类型提供大存储器。
- DataBlade 开发者可以对 CLOB 数据类型创建索引。

CLOB 数据类型的缺点如下:

- 在完整的磁盘页中分配它，因此短项浪费空间。
- 对可以如何在 SQL 语句中使用 CLOB 列有所限制。（请参阅第 8-7 页的『使用智能大对象』。）
- 它并非对所有 Informix 数据库服务器都可用。

使用智能大对象

要存储具有 BLOB 或 CLOB 数据类型的列，必须分配 sbspace。*sbspace* 是用于以尽可能高效率的方式存储 BLOB 和 CLOB 数据的逻辑存储器单元。可以编写允许用户访存和存储 BLOB 或 CLOB 数据的 IBM Informix ESQL/C 程序。要直接访问和处理智能大对象的应用程序员可以查阅 *IBM Informix: ESQL/C Programmer's Manual*。

在任何 SQL 语句（无论是交互式的还是编程的）中，*不能*以下列方式使用 BLOB 或 CLOB 列:

- 在算术或布尔表达式中
- 在 GROUP BY 或 ORDER BY 子句中
- 在 UNIQUE 测试中
- 用于建立索引，作为 Informix B 树索引的一部分

  但是，DataBlade 开发者具有对 CLOB 列创建索引的能力。

在以交互方式输入的 SELECT 语句中，BLOB 或 CLOB 列可以:

- 在创建表时通过 DEFAULT NULL 子句指定 NULL 值来作为缺省值
- 在创建表时使用 NOT NULL 约束不接受 NULL 值
- 使用 IS [NOT] NULL 谓词对其进行测试

在 ESQL/C 程序中，可以使用 **ifx_lo_stat()** 函数来确定 BLOB 或 CLOB 数据的长度。

复制智能大对象

Dynamic Server 提供了可以从 SQL 语句中调用的函数来导入和导出智能大对象。表 8-1 显示了智能大对象函数。

表 8-1. 智能大对象的 SQL 函数

函数名	用途
FILETOBLOB()	将文件复制到 **BLOB** 列中
FILETOCLOB()	将文件复制到 **CLOB** 列中
LOCOPY()	将 BLOB 或 CLOB 数据复制到另一个 **BLOB** 或 **CLOB** 列中
LOTOFILE()	将 BLOB 或 CLOB 数据复制到文件中

有关智能大对象函数的详细信息和语法，请参阅《*IBM Informix: SQL 指南: 语法*》中的『表达式』段。

要点: 不允许在 BLOB 和 CLOB 数据类型之间进行数据类型转换。

复杂数据类型

复杂数据类型通常是其它现有数据类型的组合。例如: 可以创建其组件包括内置类型、不透明类型、单值类型或其它复杂类型的复杂数据类型。与用户定义的类型相比，复杂数据类型的一项重要优点是用户可以访问和处理复杂数据类型的个别组件。

相反，内置类型和用户定义的类型是独立（封装）数据类型。因此，访问不透明数据类型的组件值的唯一方法是通过对不透明类型定义的函数进行。

图 8-3 显示了 Dynamic Server 支持的复杂数据类型以及可以用来创建复杂数据类型的语法。

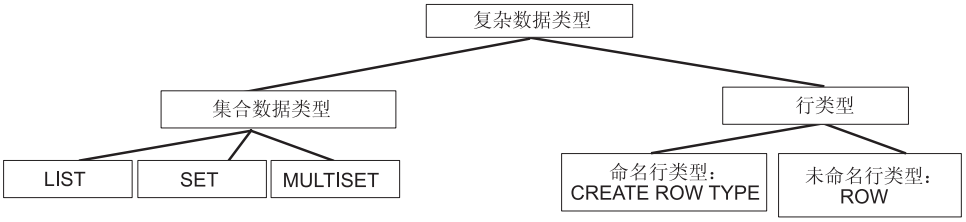


图 8-3. 复杂数据类型

图 8-3 说明的复杂数据类型提供了下列扩展数据类型支持:

- **集合类型** 每当需要在表单元格中存储和处理数据集合时，可以使用集合类型。可以对列指定集合类型。
- **行类型** 行类型通常包含多个字段。当要在列或变量中存储多种类型的数据时，可以创建行类型。行类型具有两类: 命名行类型和未命名行类型。可以对列和变量指定未命名行类型。可以对列、变量、表或视图指定命名行类型。当对表指定命名行类型时，表是*类型表*。类型表的主要优点是它们可用来定义继承层次结构。

有关如何对本章描述的复杂数据类型执行 SELECT、INSERT、UPDATE 和 DELETE 操作的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

集合数据类型

集合数据类型使您能够在表的单个行内存储和处理数据集合。集合数据类型有两个组件: *类型构造函数*（它确定集合类型是 SET、MULTISET 还是 LIST）和*元素类型*（它指定集合可包含的数据的类型）。（下列各节详细描述了 SET、MULTISET 和 LIST 集合类型。）

集合的元素几乎可以具有任何数据类型。（要获取例外情况的列表，请参阅第 8-13 页的『对集合的限制』。）集合的*元素*就是集合包含的值。在包含下列值的集合中: `{'blue', 'green', 'yellow', and 'red'}`, `'blue'` 表示集合中的单个元素。集合中的每个元素都必须具有相同的类型。例如: 元素类型为 INTEGER 的集合只能包含整数值。

集合的元素类型可以表示单个数据类型（列），也可以表示多个数据类型（行）。在以下示例中，**col_1** 列表示整数的 SET:

```
col_1 SET(INTEGER NOT NULL)
```

要定义包含多个数据类型的集合数据类型，可使用命名行类型或未命名行类型。在以下示例中，**col_2** 列 表示包含 **name** 和 **salary** 字段的行 SET:

```
col_2 SET(ROW(name VARCHAR(20), salary INTEGER) NOT NULL)
```

要点: 定义集合数据类型时，必须包括 NOT NULL 约束作为类型定义的一部分。不允许对集合数据类型施加任何其它列约束。

在将列定义为具有集合数据类型之后，可以对该集合执行下列操作:

- 选择和修改集合的个别元素（仅从 ESQL/C 程序中）。
- 对集合包含的元素计数。
- 确定集合是否包含特定值。

有关用来创建集合数据类型的语法的信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的『数据类型』段。有关如何将一种集合类型的值转换为具有另一集合类型的值的信息，请参阅《*IBM Informix: SQL 教程指南*》。

集合中的空值

集合不能包含 NULL 元素。然而，当集合是行类型时，可以对集合包含的行类型的任何或所有字段插入 NULL 值。假定创建下面这个带有集合列的表:

```
CREATE TABLE tab1 (col1 INT,
       col2 SET(ROW(a INT, b INT) NOT NULL));
```

由于只有行类型的组成字段指定了 NULL 值，所以允许下列语句:

```
INSERT INTO tab1 VALUES ( 25,"SET{ROW(NULL, NULL)}");

INSERT INTO tab1 VALUES ( 35,"SET{ROW(4, NULL)}");

INSERT INTO tab1 VALUES ( 45,"SET{ROW(14, NULL), ROW(NULL,5)}");

UPDATE tab1 SET col2 = "SET{ROW(NULL, NULL)}" WHERE col1 = 45;
```

然而，由于集合元素指定了 NULL 值，所以下列每个语句都返回一条错误消息:

```
INSERT INTO tab1 VALUES ( 45, "SET{NULL)}");

UPDATE tab1 SET col2 = "SET{NULL}" WHERE col1 = 55;
```

使用 SET 集合类型

SET 是元素的无序集合，在这个集合中，每个元素都是唯一的。当要存储其元素具有下列特征的集合时，将列定义为具有 SET 集合类型:

- 元素不包含重复值。
- 元素不具有与它们相关联的特定顺序。

为了说明可以如何使用 SET，想象人力资源部门需要关于公司中每个雇员的家属的信息。可使用集合类型来在 **employee** 表中定义一列，它存储雇员家属的姓名。以下语句创建一个表，在该表中，将 **dependents** 列定义为 SET:

```
CREATE TABLE employee
(
   name         CHAR(30),
   address   CHAR (40),
   salary   INTEGER,
   dependents  SET(VARCHAR(30) NOT NULL)
);
```

对任何给定行的 **dependents** 列执行的查询都将返回雇员的所有家属的姓名。在本例中，由于每个雇员的家属集合不应包含任何重复值，所以 SET 是适当的集合类型。定义为 SET 的列确保集合中的每个元素都是唯一的。

为了说明如何定义元素具有行类型的集合类型，假定要让 **dependents** 列包括雇员家属的姓名和生日。在以下示例中，将 **dependents** 列定义为 SET，其元素类型是行类型:

```
CREATE TABLE employee
(
   name         CHAR(30),
   address   CHAR (40),
   salary   INTEGER,
   dependents  SET(ROW(name VARCHAR(30), bdate DATE) NOT NULL)
);
```

dependents 列中的集合的每个元素都包含 **name** 值和 **bdate** 值。**employee** 表的每一行都包含关于雇员的信息以及带有该雇员的家属的姓名和生日的集合。例如: 如果某个雇员没有家属，则 **dependents** 列的集合是空的。如果某个雇员有 10 个家属，则集合应包含 10 个元素。

使用 MULTISET 集合类型

MULTISET 是元素的集合，在这个集合中，元素可以具有重复的值。例如: 整数的 MULTISET 可包含集合 {1,3,4,3,3}，此集合带有重复的元素。当要存储元素具有下列特征的集合时，将列定义为具有 MULTISET 集合类型:

- 元素可能不是唯一的。
- 元素不具有与它们相关联的特定顺序。

为了说明可以如何使用 MULTISET，假定人力资源部门想要跟踪奖励给公司中的雇员的奖金。要跟踪每个雇员随着时间的推移而得到的奖金，可使用 MULTISET 来在表中定义一个列，这个列记录每个雇员得到的所有奖金。在以下示例中，**bonus** 列是 MULTISET:

```
CREATE TABLE employee
(
   name         CHAR(30),
   address   CHAR (40),
   salary   INTEGER,
   bonus     MULTISET(MONEY NOT NULL)
);
```

可使用此语句中的 **bonus** 列来存储和访问每个雇员的奖金集合。对任何给定行的 **bonus** 列执行的查询都将返回雇员已收到的每笔奖金的美元金额。由于雇员可能会收到多笔相同金额的奖金（导致集合包含的元素并不全是唯一的），所以将 **bonus** 列定义为 MULTISET，它允许重复值。

使用 LIST 集合类型

LIST 是元素的有序集合，它允许重复值。LIST 与 MULTISET 的不同之处在于 LIST 中的每个元素在集合中都具有序数位置。列表中的元素的顺序与将值插入 LIST 的顺序相对应。当要存储元素具有下列特征的集合时，将列定义为具有 LIST 集合类型:

- 元素具有与它们相关联的特定顺序。
- 元素可能不是唯一的。

为了说明可以如何使用 LIST，假定销售部门要保留每个销售人员的销售总计的每月记录。可使用 LIST 来在表中定义一个列，该列包含每个销售人员的每月销售总计。以下示例创建一个表，在这个表中，**month_sales** 列是 LIST。 LIST 中的第一个条目（元素）具有序数位置 1，它可以与一月相对应，序数位置为 2 的第二个元素与二月相对应，依此类推:

```
CREATE TABLE sales_person
(
   name         CHAR(30),
   month_sales  LIST(MONEY NOT NULL)
);
```

可使用此语句中的 **month_sales** 列来存储和访问每位销售人员的每月销售总计。更确切地说，可对 **month_sales** 列执行查询来了解:

- 某个销售人员在指定月份内的总销售量
- 每个销售人员在指定月份内的总销售量

嵌套集合类型

*嵌套的集合*是包含另一集合类型的集合类型。可以将任何集合类型嵌套在另一集合类型中。对集合类型可以具有的嵌套深度没有实际的限制。然而，对已经嵌套了一两层以上的集合执行插入或更新可能比较困难。以下示例显示了数种创建对嵌套集合类型定义的列的方法:

```
col_1 SET(MULTISET(VARCHAR(20) NOT NULL) NOT NULL);

col_2 MULTISET(ROW(x CHAR(5), y SET(INTEGER NOT NULL))
NOT NULL);

col_3 LIST(MULTISET(ROW(a CHAR(2), b INTEGER) NOT NULL)
NOT NULL);
```

有关如何访问嵌套集合的信息，请参阅《*IBM Informix: SQL 教程指南*》。

将集合类型添加至现有的表

可以使用 ALTER TABLE 语句来添加或删除具有集合类型（或任何其它数据类型）的列。例如：以下语句将已定义为 SET 的 **flowers** 列添加至 **nursery** 表：

```
ALTER TABLE nursery ADD flower SET(VARCHAR(30) NOT NULL)
```

不能修改具有集合类型的现有列，也不能将不具有集合类型的列转换为具有集合类型。

有关添加和删除具有集合类型的列的更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 ALTER TABLE 语句。

对集合的限制

不能使用下列任何数据类型来作为集合的元素类型：

- TEXT
- BYTE
- SERIAL
- SERIAL8

不能使用 CREATE INDEX 语句来对集合创建索引，也不能创建集合列的函数索引。

命名行类型

*命名行类型*是在单个名称下定义的一组字段。*字段*指的是行类型的组件，不应将其与列混淆，列只与表相关联。命名行类型的字段与 C 语言结构的字段或面向对象编程中的类的成员相似。在创建命名行类型之后，对行类型指定的名称表示数据库中的唯一类型。要创建命名行类型，请指定行类型的名称并指定其组成字段的名称和数据类型。以下示例显示可以如何创建名为 **person_t** 的命名行类型：

```
CREATE ROW TYPE person_t
(
   name     VARCHAR(30) NOT NULL,
   address  VARCHAR(20),
   city    VARCHAR(20),
```

```
   state   CHAR(2),
   zip      VARCHAR(9),
   bdate    DATE
);
```

person_t 行类型包含 6 个字段: **name**、**address**、**city**、**state**、**zip** 和 **bdate**。创建命名行类型后，可以象使用任何其它数据类型那样使用它。**person_t** 可以出现在任何可以使用任何其它数据类型的位置。以下 CREATE TABLE 语句使用 **person_t** 数据类型:

```
CREATE TABLE sport_club
(
   sport     CHAR(20),
   sportnum  INT,
   member    person_t,
   since     DATE,
   paidup    BOOLEAN
)
```

可以使用大多数数据类型来定义行类型的字段。有关在行类型中不受支持的数据类型的信息，请参阅第 8-15 页的『对命名行类型的限制』。

有关用来创建命名行类型的语法，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE ROW TYPE 语句。有关如何对行类型值进行数据类型转换的信息，请参阅第 10-1 页的第 10 章，『在 Dynamic Server 中创建和使用用户定义的数据类型转换』。

何时使用命名行类型

命名行类型是一种在 Dynamic Server 中创建新数据类型的方法。创建命名行类型时，您是在定义数据库服务器已知的数据类型的字段模板。因此，行类型的字段定义与表的列定义类似: 它们都是根据数据库服务器已知的数据类型构造的。

当需要充当用户需要访问的组件值的容器的类型时，可以创建命名行类型。例如: 由于用户需要直接访问地址的个别组件值（如街道、城市、州和邮政编码），所以可以创建命名行类型来支持地址值。在将地址类型作为命名行类型创建后，用户总是可以直接访问每个字段。

相反，如果创建不透明数据类型来处理地址值，则使用 C 语言数据结构来存储所有的地址信息。由于不透明类型的组件值是封装的，所以必须定义函数来抽取街道、城市、州和邮政编码的组件值。因此，不透明数据类型是定义和使用都更为复杂的类型。

在定义数据类型之前，确定该类型是否仅仅是用户可以直接访问的一组值的容器。如果该类型符合此描述，则使用命名行类型。

选择命名行类型的名称

可以对命名行类型指定您喜欢的任何名称，只要该名称不违反为 SQL 标识符建立的约定。*《IBM Informix: SQL 指南: 语法》*中的『标识符』段描述了 SQL 标识符的约定。为了避免类型和表名冲突，本手册中的示例在行类型名末尾使用 **_t** 字符来指定命名行类型。

您必须具有“资源”特权才能创建命名行类型。由于所有数据类型共享同一个名称空间，所以，对命名行类型指定的名称不应该与数据库中存在的任何其它数据类型的名称相同。在符合 ANSI 的数据库中，*owner.type* 组合在数据库中必须是唯一的。在不符合 ANSI 的数据库中，名称在数据库中必须是唯一的。

要点: 在其他用户可以使用命名行类型之前，必须授予对其的 USAGE 特权。有关授予和取消对命名行类型的特权的信息，请参阅第 12-1 页的第 12 章，『实现维数据库（XPS）』。

对命名行类型的限制

本节描述使用命名行类型时适用的限制。

对数据类型的限制: 创建包含大对象列的类型表时，建议使用 BLOB 或 CLOB 数据类型而不是使用 TEXT 或 BYTE 数据类型。为了向后兼容，可以创建包含 TEXT 或 BYTE 字段的命名行类型并使用该类型来将现有的（无类型）表重新创建为类型表。然而，尽管可使用包含 TEXT 或 BYTE 字段的命名行类型来创建类型表，但不能使用这样的行类型作为列。可以对类型表或列指定包含 BLOB 或 CLOB 字段的命名行类型。

对约束的限制: 在 CREATE ROW TYPE 语句中，只能对命名行类型的字段指定 NOT NULL 约束。必须在 CREATE TABLE 语句中定义所有其它约束。有关更多信息，请参阅*《IBM Informix: SQL 指南: 语法》*中的 CREATE TABLE 语句。

对索引的限制: 不能使用 CREATE INDEX 语句来对命名行类型列创建索引。然而，可使用用户定义的例程来为行类型列创建函数索引。

对 SERIAL 数据类型的限制: 不能将包含 SERIAL 或 SERIAL8 数据类型的命名行类型用作表中的列类型。当数据库服务器尝试创建表时，下列语句将返回错误:

```
CREATE ROW TYPE row_t (s_col SERIAL)

CREATE TABLE bad_tab (col1 row_t)
```

然而，可使用包含 SERIAL 或 SERIAL8 数据类型的命名行类型来创建类型表。

有关 SERIAL 和 SERIAL8 类型在表层次结构中的使用和行为的信息，请参阅第 9-11 页的『表层次结构中的 SERIAL 类型』。

使用命名行类型来创建类型表

可以创建类型表或无类型表。*类型表*是对其指定了命名行类型的表。*无类型表*是没有对其指定命名行类型的表。CREATE ROW TYPE 语句创建命名行类型，但不为该行类型的实例分配存储器。要为命名行类型的实例分配存储器，必须对某个表指定该行类型。以下示例显示如何创建类型表:

```
CREATE ROW TYPE person_t
(
   name     VARCHAR(30),
   address  VARCHAR(20),
   city    VARCHAR(20),
   state   CHAR(2),
   zip      INTEGER,
   bdate    DATE
);

CREATE TABLE person OF TYPE person_t;
```

第一个语句创建 **person_t** 类型。第二个语句创建 **person** 表，它包含 **person_t** 类型的实例。更明确地说，类型表中的每一行都包含对该表指定的命名行类型的一个实例。在上述示例中，**person_t** 类型的字段定义 **person** 表的列。

要点: 由于在可以使用命名行类型来定义类型表之前该命名行类型必须存在，所以创建命名行类型的次序十分重要。

将数据插入类型表与将数据插入无类型表没有什么不同。在将数据插入类型表时，操作将创建行类型的实例并将其插入到表中。以下示例显示如何将行插入到 **person** 表中:

```
INSERT INTO person
VALUES ('Brown, James', '13 First St.', 'San Carlos', 'CA', 94070,
'01/04/1940')
```

INSERT 语句创建 **person_t** 类型的实例并将其插入到表中。有关如何插入、更新和删除对命名行类型定义的列的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

可使用单个命名行类型来创建多个类型表。在此情况下，每个表都具有唯一的名称，但所有表共享同一类型。

要点: 不能创建作为临时表的类型表。

有关在实现数据模型时使用类型表的优点的信息，请参阅第 9-2 页的『类型继承』。

更改表的类型

与无类型表相比，类型表的主要优点是类型表可以在继承层次结构中使用。通常，继承允许表获取另一个表的表示法和行为。有关更多信息，请参阅第 9-1 页的『什么是继承？』。

ALTER TABLE 语句的 DROP 和 ADD 子句允许在类型表和无类型表之间进行更改。ADD 和 DROP 操作都不影响表中存储的数据。

将无类型表转换为类型表: 如果要将现有的无类型表转换为类型表，可使用 ALTER TABLE 语句。例如: 考虑以下无类型表:

```
CREATE TABLE manager
(
   name     VARCHAR(30),
   department  VARCHAR(20),
   salary   INTEGER
);
```

要将无类型表转换为类型表，命名行类型的字段名和字段类型都必须与现有表的列名和列类型相匹配。例如: 要让 **manager** 表成为类型表，首先必须创建与表的列定义相匹配的命名行类型。以下语句创建 **manager_t** 类型，该类型包含与 **manager** 表的列相匹配的字段名和字段类型:

```
CREATE ROW TYPE manager_t
(
   name     VARCHAR(30),
   department  VARCHAR(20),
   salary   INTEGER
);
```

在创建要对现有无类型表指定的命名行类型之后，使用 ALTER TABLE 语句来对表指定该类型。以下语句改变 **manager** 表并使其成为具有 **manager_t** 类型的类型表:

```
ALTER TABLE manager ADD TYPE manager_t
```

新的 **manager** 表与旧表包含相同的列和数据类型，但现在提供了类型表的优点。

将类型表转换为无类型表: 还可以使用 ALTER TABLE 语句来将类型表更改为无类型表:

```
ALTER TABLE manager DROP TYPE
```

技巧: 将列添加至类型表需要三个 ALTER TABLE 语句来删除类型、添加列和将类型添加至表。

使用命名行类型来创建列

类型表和无类型表都可以包含对命名行类型定义的列。对命名行类型定义的列无论是出现在类型表中还是出现在无类型表中都具有相同的行为。在以下示例中，第一个语句创建命名行类型 **address_t**；第二个语句对 **employee** 表中的 **address** 列指定 **address_t** 类型:

```
CREATE ROW TYPE address_t
(
   street  VARCHAR(20),
   city    VARCHAR(20),
   state   CHAR(2),
   zip     VARCHAR(9)
);

CREATE TABLE employee
(
   name     VARCHAR(30),
   address  address_t,
   salary   INTEGER
);
```

在上述 CREATE TABLE 语句中，**address** 列具有 **address_t** 类型的 **street**、**city**、**state** 和 **zip** 字段。因此，**employee** 表（它只有 3 列）包含 **name**、**street**、**city**、**state**、**zip** 和 **salary** 的值。使用点符号表示法来访问对行类型定义的列的个别字段。有关使用点符号表示法来访问列字段的信息，请参阅 *《IBM Informix: SQL 教程指南》*。

在将数据插入到对其指定了行类型的列中时，需要使用 ROW 构造函数来指定行类型的行文字值。以下示例显示如何使用 INSERT 语句来将行插入到 **employee** 表中:

```
INSERT INTO employee
VALUES ('John Bryant',
  ROW('10 Bay Street', 'Madera', 'CA', 95400)::address_t, 55000);
```

对于对命名行类型执行的插入或更新操作，不强制执行强类型化。要确保行值具有命名行类型，必须显式地将数据类型转换为命名行类型以生成具有命名行类型的值，如上一示例所示。INSERT 语句插入三个值，其中一个是包含四个值的行类型值。更明确地说，此操作对 **name** 和 **salary** 列插入一元值，但它创建 **address_t** 类型的实例并将其插入到 **address** 列中。

有关如何插入、更新和删除对行类型定义的列的更多信息，请参阅 *《IBM Informix: SQL 教程指南》*。

在另一个行类型中使用命名行类型

可以使用命名行类型作为另一个行类型中的字段的数据类型。*嵌套行类型*是包含另一个行类型的行类型。可以将任何行类型嵌套在任何其它行类型中。对行类型可以具有的嵌套深度不存在实际的限制。然而，对深度嵌套的行类型执行插入或更新要求仔细地使用语法。

对于命名行类型，由于在可以使用命名行类型来在另一个行类型中定义列或字段之前该命名行类型必须存在，所以创建行类型的次序十分重要。在以下示例中，第一个语句创建 **address_t** 类型，在第二个语句中，该类型用来定义 **employee_t** 类型的 **address** 字段的类型:

```
CREATE ROW TYPE address_t
(
   street  VARCHAR (20),
   city    VARCHAR(20),
   state   CHAR(2),
   zip     VARCHAR(9)
);

CREATE ROW TYPE employee_t
(
   name     VARCHAR(30) NOT NULL,
   address  address_t,
   salary   INTEGER
);
```

要点: 不能递归地使用行类型。如果 **type_t** 是行类型，则不能使用 **type_t** 来作为包含在 **type_t** 中的字段的数据类型。

删除命名行类型

要删除命名行类型，请使用 DROP ROW TYPE 语句。仅当类型没有相关性时才能将其删除。如果下列任何条件成立，则不能删除命名行类型:

- 当前已对表指定该类型。
- 当前已对表中的列指定该类型。
- 当前已对另一个行类型中的字段指定该类型。

以下示例显示如何删除 **person_t** 类型:

```
DROP ROW TYPE person_t restrict;
```

有关如何从类型层次结构中删除命名行类型的信息，请参阅第 9-6 页的『从类型层次结构中删除命名行类型』。

未命名行类型

*未命名行类型*是使用 ROW 构造函数创建的一组类型字段。命名行类型与未命名行类型之间的一项重要区别是不能对表指定未命名行类型。只能使用未命名行类型定义列或字段的类型。另外，未命名行类型只是由它的结构标识，而命名行类型由它的名称标识。行类型的结构由它的字段的数目和数据类型组成。

以下语句对 **student** 表的列指定两个未命名行类型:

```
CREATE TABLE student
(
   s_name  ROW(f_name VARCHAR(20), m_init CHAR(1),
               l_name VARCHAR(20) NOT NULL),
   s_address  ROW(street VARCHAR(20), city VARCHAR(20),
                  state CHAR(2), zip VARCHAR(9))
   );
```

student 表的 **s_name** 和 **s_address** 列都包含多个字段。未命名行类型的每个字段都可以具有不同的数据类型。虽然 **student** 表只有两列，但未命名行类型总共定义了 7 个字段: **f_name**、**m_init**、**l_name**、**street**、**city**、**state** 和 **zip**。

以下示例显示如何使用 INSERT 语句来将数据插入到 **student** 表中:

```
INSERT INTO student
VALUES (ROW('Jim', 'K', 'Johnson'), ROW('10 Grove St.',
'Eldorado', 'CA', 94108))
```

有关如何修改对行类型定义的列的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

数据库服务器不对两个包含相同数目的字段并且带有相同类型的对应字段的未命名行类型加以区别。在未命名行类型的类型检查中，字段名是不相关的。例如: 数据库服务器不对下列未命名行类型加以区别:

```
ROW(a INTEGER, b CHAR(4));
ROW(x INTEGER, y CHAR(4));
```

有关未命名行类型的语法，请参阅《*IBM Informix: SQL 指南: 语法*》。有关如何对行类型值进行数据类型转换的信息，请参阅第 10-1 页的第 10 章，『在 Dynamic Server 中创建和使用用户定义的数据类型转换』。

以下数据类型不能是未命名行类型中的字段类型:

- SERIAL
- SERIAL8
- BYTE
- TEXT

当在未命名行类型的字段定义中指定了任何前述类型时，数据库服务器将返回错误。

第 9 章 理解 Dynamic Server 中的类型和表继承

在本章中

本章描述类型和表继承并阐述如何创建类型和表层次结构以修改各自层次结构中的类型和表。

什么是继承?

*继承*是允许类型或表获取另一个类型或表的属性的过程。继承属性的类型或表称为*子类型*或*子表*。其属性被继承的类型或表称为*超类型*或*超表*。继承使您能够进行增量修改，因此类型或表可以继承一组一般的属性并添加特定于它们自己的属性。可使用继承来限制修改程度，以使修改操作不会改变所继承的超类型或超表。

Dynamic Server 只对命名行类型和类型表支持继承。Dynamic Server 只支持单一继承。对于*单一继承*，每个子类型或子表只有一个超类型或超表。

类型继承

类型继承仅适用于命名行类型。可使用继承来将命名行类型集中成*类型层次结构*，在此层次结构中，每个子类型都继承在其下定义它的超类型的表示法（数据字段）和行为（UDR、聚集和运算符）。类型层次结构具有下列优点:

- 它鼓励利用模块来实现数据模型。
- 它确保一致地重复使用模式组件。
- 它确保不会意外地遗漏数据字段。
- 它允许类型继承对另一数据类型定义的 UDR。

定义类型层次结构

第 9-2 页的图 9-1 提供了一个简单类型层次结构的示例，该层次结构包含三个命名行类型。

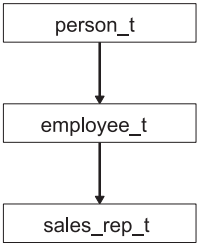


图 9-1. 类型层次结构的示例

类型层次结构顶部的超类型包含一组字段，所有下层子类型都继承这些字段。在可以创建超类型的子类型之前，该超类型必须已存在。以下示例创建第 9-2 页的图 9-1 显示的类型层次结构的 **person_t** 超类型:

```
CREATE ROW TYPE person_t
(
   name     VARCHAR(30) NOT NULL,
   address  VARCHAR(20),
   city    VARCHAR(20),
   state   CHAR(2),
   zip      INTEGER,
   bdate    DATE
);
```

要创建子类型，请指定 UNDER 关键字以及其属性由子类型继承的超类型的名称。以下示例说明可以如何将 **employee_t** 定义为继承 **person_t** 的所有字段的子类型。此示例添加在 **person_t** 类型中不存在的 **salary** 和 **manager** 字段。

```
CREATE ROW TYPE employee_t
(
   salary   INTEGER,
   manager  VARCHAR(30)
)
UNDER person_t;
```

要点：在可以创建继承超类型的属性的子类型之前，您必须对该超类型具有 UNDER 特权。有关 UNDER 特权的信息，请参阅第 12-1 页的第 12 章，『实现维数据库（XPS）』。

在第 9-2 页的图 9-1 中的类型层次结构中，**sales_rep_t** 是 **employee_t** 的子类型，而 **employee_t** 是 **sales_rep_t** 的超类型，就象 **person_t** 是 **employee_t** 的超类型一样。以下示例创建 **sales_rep_t**，它继承 **person_t** 和 **employee_t** 的所有字段并添加四个新字段。由于对子类型所作的修改不会影响它的超类型，所以 **employee_t** 不具有对 **sales_rep_t** 添加的那四个字段。

```
CREATE ROW TYPE sales_rep_t
(
rep_num      INT8,
region_num   INTEGER,
commission   DECIMAL,
home_office  BOOLEAN
)
UNDER employee_t;
```

sales_rep_t 类型包含 12 个字段：**name**、**address**、**city**、**state**、**zip**、**bdate**、**salary**、**manager**、**rep_num**、**region_num**、**commission** 和 **home_office**。

employee_t 和 **sales_rep_t** 类型的实例都继承对 **person_t** 类型定义的所有 UDR。对 **employee_t** 定义的任何附加 UDR 都自动应用于 **employee_t** 类型的实例并应用于它的子类型 **sales_rep_t** 的实例，但不应用于 **person_t** 的实例。

在前述类型层次结构中，由于每个子类型从单一超类型继承，所以此类型层次结构是单一继承的一个示例。图 9-2 说明了可以如何在单个超类型下面定义多个子类型。尽管单一继承要求每个子类型都从一个且只从一个超类型继承，但是对定义的类型层次结构的深度或宽度不存在实际限制。

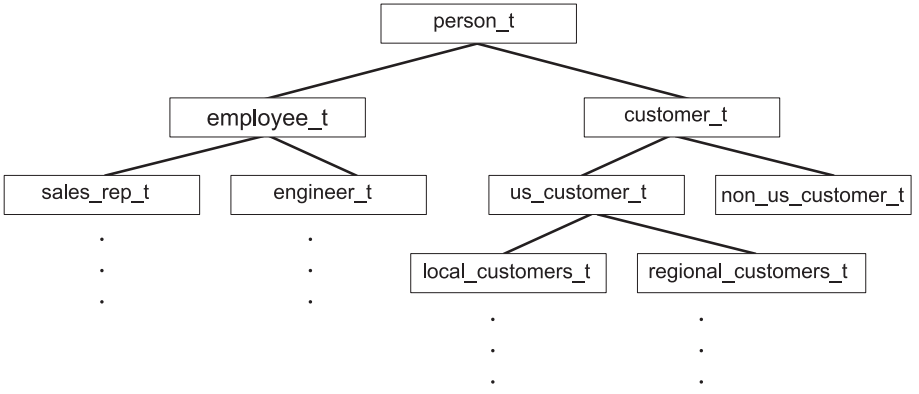


图 9-2. 具有树结构的类型层次结构的示例

任何层次结构的最顶部的类型都称为*根超类型*。在图 9-2 中，**person_t** 是层次结构的根超类型。除根超类型以外，层次结构中的任何类型都有可能同时既作为超类型又作为子类型。例如：**customer_t** 是 **person_t** 的子类型和 **us_customer_t** 的超类型。在层次结构中位于较低层的子类型包含根超类型的属性，但不直接从根超类型继承属性。例如：**us_customer_t** 只有一个超类型 **customer_t**，但由于 **customer_t** 本身是 **person_t** 的子类型，所以，**customer_t** 从 **person_t** 继承的字段和例程也由 **us_customer_t** 继承。

重载类型层次结构中的类型的例程

*例程重载*指的是这样的能力：将一个名称指定给多个例程并指定不同类型的自变量来供例程对它们进行操作。在类型层次结构中，子类型自动继承对其超类型定义的例程。然而，可以对子类型定义新的例程以覆盖所继承的同名例程。例如：假定对类型 **person_t** 创建了返回类型 **person_t** 的实例的姓和生日的 **getinfo()** 例程。可以对类型 **employee_t** 注册另一个 **getinfo()** 例程，它返回 **employee_t** 的实例的姓和薪水。这样，您就可以重载例程，以便对类型层次结构中的每个类型都有定制的例程，如图 9-3 所示。

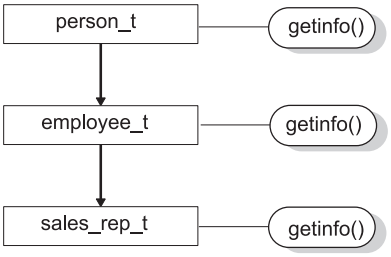


图 9-3. 类型层次结构中的例程重载的示例

当您重载例程以将例程定义为对类型层次结构中的不同类型具有相同的名称但具有不同的自变量时，指定的自变量确定执行哪个例程。例如：如果使用类型为 **employee_t** 的自变量来调用 **getinfo()**，则对类型 **employee_t** 定义的 **getinfo()** 例程将覆盖所继承的同名例程。同样，如果对类型 **sales_rep_t** 定义另一个 **getinfo()**，则使用类型为 **sales_rep_t** 的自变量来调用 **getinfo()** 将覆盖 **sales_rep_t** 从 **employee_t** 继承的例程。

有关如何创建和注册用户定义的例程（UDR）的信息，请参阅《*IBM Informix: 用户定义的例程与数据类型开发者指南*》。

继承和类型可替代性

在类型层次结构中，子类型自动继承对其超类型定义的所有例程。因此，如果调用带有子类型的自变量的例程，并且没有对该子类型定义例程，则数据库服务器可调用对超类型定义的例程。*类型可替代性*指的是这样的能力：在期望超类型的实例时使用子类型的实例。例如：假定创建例程 **p_info()**，它接受类型为 **person_t** 的自变量并返回类型为 **person_t** 的实例的姓和生日。如果没有注册其它 **p_info()** 例程，并且使用类型 **employee_t** 的自变量来调用 **p_info()**，则该例程将返回类型 **employee_t** 的实例的姓名和生日字段（从 **person_t** 继承）。由于 **employee_t** 继承它的超类型 **person_t** 的函数，所以可以这样做。

通常，当数据库服务器尝试对例程进行求值时，数据库服务器将搜索与您调用例程时指定的例程名和自变量相匹配的特征符。如果找到这样的例程，则数据库服务器将使用此例程。如果找不到精确匹配，则数据库服务器尝试查找以下例程：具有相同的名称，并且其自变量类型是调用例程时指定的自变量类型的超类型。第 9-5 页的图 9-4 显示了当使用具有子类型 **sales_rep_t** 的自变量调用 **get()** 例程时数据库服务器如何搜索可以使用的例程。虽然没有对 **sales_rep_t** 类型定义 **get()** 例程，但数据库服务器将搜索例程，直到找到已对层次结构中的超类型定义的 **get()** 例程为止。在本例中，没有对 **sales_rep_t** 及其超类型 **employee_t** 定义 **get()** 例程。然而，由于为 **person_t** 定义了例程，所以将调用此例程来对 **sales_rep_t** 的实例执行操作。

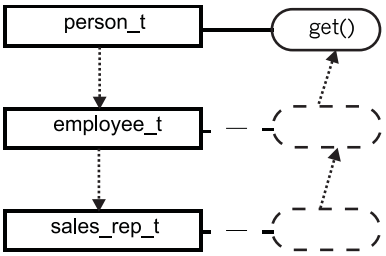


图 9-4. 数据库服务器如何在类型层次结构中搜索例程的示例

数据库服务器搜索它可以使用的例程的过程称为*例程解析*。有关例程解析的更多信息，请参阅《*IBM Informix: 用户定义的例程与数据类型开发者指南*》。

从类型层次结构中删除命名行类型

要从类型层次结构中删除命名行类型，请使用 DROP ROW TYPE 语句。然而，仅当类型不具有相关性时才可以将其删除。如果下列任何一个条件成立，则不能删除命名行类型:

- 当前已对表指定该类型。
- 该类型是另一类型的超类型。

以下示例显示如何删除 **sales_rep_t** 类型:

```
DROP ROW TYPE sales_rep_t RESTRICT;
```

要删除超类型，首先必须删除每个从该超类型继承属性的子类型。应该按照类型创建顺序的逆向顺序来删除类型层次结构中的类型。例如: 要删除图 9-4 显示的 **person_t** 类型，首先必须按以下顺序删除它的子类型:

```
DROP ROW TYPE sale_rep_t RESTRICT;
DROP ROW TYPE employee_t RESTRICT;
DROP ROW TYPE person_t RESTRICT;
```

要点: 要删除类型，您必须是数据库管理员或该类型的所有者。

表继承

只有对命名行类型定义的表才支持表继承。*表继承*是一项属性，它允许表从表层次结构中位于该表之上的超表继承行为（约束、存储选项和触发器）。*表层次结构*是可以在各个表之间定义的关系，在此关系中，子表继承超表的行为。表继承具有下列优点:

- 它鼓励利用模块来实现数据模型。
- 它确保一致地重复使用模式组件。
- 它允许您构造这样的查询: 其作用域可以是表层次结构中的某些表或全部表。

在表层次结构中，子表自动从其超表继承下列属性:

- 所有约束定义（主键、唯一约束和引用约束）
- 存储选项
- 所有触发器
- 索引
- 存取方法

类型层次结构与表层次结构之间的关系

必须对表层次结构中的每个表指定相应类型层次结构中的命名行类型。图 9-5 显示了在类型层次结构与表层次结构之间可以存在的关系的示例。

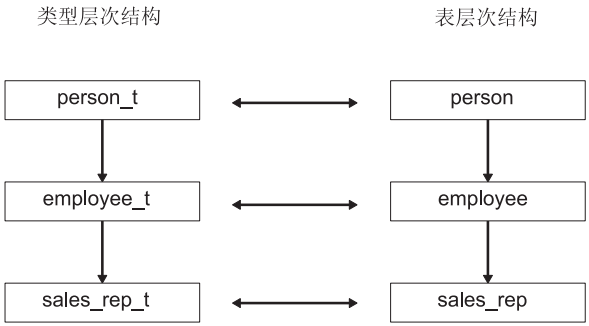


图 9-5. 类型层次结构与表层次结构之间的关系的示例

然而，还可以定义其中的命名行类型不必与表层次结构中的表一一对应的类型层次结构。图 9-6 显示了可以如何创建只将其某些命名行类型指定给表的类型层次结构。

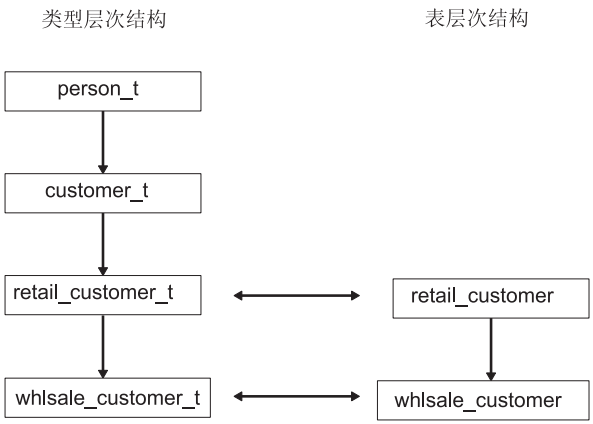


图 9-6. 只将其中某些类型指定给表的继承层次结构的示例

定义表层次结构

在可以创建表之前，用来定义表的类型必须已存在。同样，在定义相应的表层次结构之前定义类型层次结构。要在表层次结构中的特定子表和超表之间建立关

系，请使用 UNDER 关键字。下列 CREATE TABLE 语句定义第 9-7 页的图 9-5 显示的简单表层次结构。本节中的示例假定 **person_t**、**employee_t** 和 **sales_rep_t** 类型已存在。

```
CREATE TABLE person OF TYPE person_t;

CREATE TABLE employee OF TYPE employee_t UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t UNDER employee;
```

person、**employee** 和 **sales_rep** 表分别是对 **person_t**、**employee_t** 和 **sales_rep_t** 类型定义的。因此，对于类型层次结构中的每个类型，在表层次结构中都存在相应的表。另外，表层次结构的表之间的关系必须与类型层次结构的类型之间的关系相匹配。例如: **employee** 表从 **person** 表进行继承的方式与 **employee_t** 类型从 **person_t** 类型进行继承的方式相同，并且，**sales_rep** 表从 **employee** 表进行继承的方式与 **sales_rep_t** 类型从 **employee_t** 类型进行继承的方式相同。

子表自动继承所有已添加至超表的可继承属性。因此，可以随时添加或改变超表的属性，子表将自动继承更改。有关更多信息，请参阅第 9-10 页的『在表层次结构中修改表行为』。

要点: 在可以创建继承超表的属性的子表之前，您必须对该超表具有 UNDER 特权。有关更多信息，请参阅第 6-7 页的『类型表的隶属于特权（IDS）』。

表层次结构中的表行为的继承

当在超表下面创建子表时，子表将继承它的超表的所有属性，包括下列属性:

- 超表的所有列
- 约束定义
- 存储选项
- 索引
- 引用完整性
- 触发器
- 存取方法

另外，如果表 **c** 从表 **b** 继承，并且表 **b** 从表 **a** 继承，则表 **c** 自动继承表 **b** 的独特行为以及表 **b** 从表 **a** 继承的行为。因此，实际定义行为的超表与继承行为的子表之间可以相距若干层。例如: 考虑以下表层次结构:

```
CREATE TABLE person OF TYPE person_t
(PRIMARY KEY (name))
   FRAGMENT BY EXPRESSION
name < 'n' IN dbspace1,
```

```
name >= 'n' IN dbspace2;

CREATE TABLE employee OF TYPE employee_t
(CHECK(salary > 34000))
UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t
LOCK MODE ROW
UNDER employee;
```

在这个表层次结构中，**employee** 和 **sales_rep** 表继承 **person** 表的主键名和分段存储策略。**sales_rep** 表继承 **employee** 表的检查约束并添加 LOCK MODE。下表显示了层次结构中的每个表的行为。

表	表行为
person	PRIMARY KEY 和 FRAGMENT BY EXPRESSION
employee	PRIMARY KEY、FRAGMENT BY EXPRESSION 和 CHECK 约束
sales_rep	PRIMARY KEY、FRAGMENT BY EXPRESSION、CHECK 约束和 LOCK MODE ROW

表层次结构还可以包含这样的子表：在这些子表中，对子表定义的行为可以覆盖（以别的方式）从其超表继承的行为。考虑以下表层次结构，除了 **employee** 表添加了新的存储选项之外，这个表层次结构与先前示例完全相同：

```
CREATE TABLE person OF TYPE person_t
(PRIMARY KEY (name))
   FRAGMENT BY EXPRESSION
name < 'n' IN person1,
name >= 'n' IN person2;

CREATE TABLE employee OF TYPE employee_t
(CHECK(salary > 34000))
   FRAGMENT BY EXPRESSION
name < 'n' IN employ1,
name >= 'n' IN employ2
UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t
LOCK MODE ROW
UNDER employee;
```

同样，**employee** 和 **sales_rep** 表继承 **person** 表的主键名。然而，**employee** 表的分段存储策略覆盖 **person** 表的分段存储策略。因此，**employee** 和 **sales_rep** 表都在数据库空间 **employ1** 和 **employ2** 中存储数据，而 **person** 表在数据库空间 **person1** 和 **person2** 中存储数据。

在表层次结构中修改表行为

定义表层次结构之后，就不能修改现有表的结构（列）。然而，可以在层次结构中修改表的行为。第 9-10 页的表 9-1 显示了可以在表层次结构中修改的表行为以及用来进行修改的语法。

表 9-1. 可以在表层次结构中修改的表行为

表行为	语法	注意事项
约束定义	ALTER TABLE	要添加或删除约束，请使用 ADD CONSTRAINT 或 DROP CONSTRAINT 子句。有关更多信息，请参阅第 9-10 页的『对表层次结构中的表的约束』。
索引	CREATE INDEX, ALTER INDEX	有关更多信息，请参阅第 9-10 页的『将索引添加至表层次结构中的表』以及《*IBM Informix: SQL 指南: 语法*》中的 CREATE INDEX 和 ALTER INDEX 语句。
触发器	CREATE/DROP TRIGGER	不能删除继承的触发器。然而，可以从超表中删除触发器或将触发器添加至子表以覆盖继承的触发器。有关如何对超表和子表修改触发器的信息，请参阅第 9-11 页的『表层次结构中的表上的触发器』。有关如何创建触发器的信息，请参阅《*IBM Informix: SQL 教程指南*》。

当您修改层次结构中的超表时，任何现有的子表都将自动继承新的表行为。

要点: 当使用 ALTER TABLE 语句来修改表层次结构中的表时，只能使用 ADD CONSTRAINT、DROP CONSTRAINT、MODIFY NEXT SIZE 和 LOCK MODE 子句。

对表层次结构中的表的约束

只能在对其定义约束的表中改变或删除该约束。当约束是继承的时，不能从子表中删除或改变约束。但是，子表可添加附加的约束。任何从定义约束的表进行继承的子表都将继承您对该表定义的任何附加约束。由于约束具有可加性，所以，所有继承的约束以及当前（添加的）约束都适用。

将索引添加至表层次结构中的表

当对层次结构中的超表定义索引时，您在该超表下面定义的任何子表也将继承该索引。假定一个表层次结构包含表 **tab_a**、**tab_b** 和 **tab_c**，其中，**tab_a** 是 **tab_b** 的超表，而 **tab_b** 是 **tab_c** 的超表。如果对 **tab_b** 的某个列创建索引，则该索引在 **tab_b** 和 **tab_c** 中的该列上都存在。如果对 **tab_a** 的列创建索引，则该索引的范围将包括 **tab_a**、**tab_b** 和 **tab_c**。

要点: 不能删除或修改子表从超表继承的索引。然而，可以将索引添加至子表。

索引、唯一约束和主键全都紧密相关。当指定唯一约束或主键时，数据库服务器自动对该列创建唯一索引。因此，对超表定义的主键或唯一约束将应用于所有子表。例如: 假定有两个表（一个超表和一个子表），这两个表都包含 **emp_id** 列。如果超表指定 **emp_id** 具有唯一约束，则子表必须包含在子表和超表中都唯一的 **emp_id** 值。

要点: 即使层次结构中的某些表不继承主键，也不能在表层次结构中定义多个主键。

表层次结构中的表上的触发器

不能删除继承的触发器。然而，可以对子表创建触发器以覆盖该子表从超表继承的触发器。与约束不同，触发器不具有可加性；只有位于层次结构中的超表上的最接近的触发器才适用。

如果要禁用子表从它的超表继承的触发器，可以对子表创建空触发器以覆盖来自超表的触发器。由于触发器不具有可加性，所以将对该子表以及该子表下的任何子表执行这个空触发器，那些子表不会进行进一步的覆盖。

表层次结构中的 SERIAL 类型

表层次结构可以包含具有 SERIAL 和 SERIAL8 类型的列。然而，在表层次结构中只允许一个 SERIAL 和一个 SERIAL8 列。假定您创建以下类型和表层次结构:

```
CREATE ROW TYPE parent_t (a INT);
CREATE ROW TYPE child1_t (s_col SERIAL) UNDER parent_t;
CREATE ROW TYPE child2_t (s8_col SERIAL8) UNDER child1_t;
CREATE ROW TYPE child3_t (d FLOAT) UNDER child2_t;

CREATE TABLE parent_tab of type parent_t;
CREATE TABLE child1_tab of type child1_t UNDER parent_tab;
CREATE TABLE child2_tab of type child2_t UNDER child1_tab;
CREATE TABLE child3_tab of type child3_t UNDER child2_tab;
```

parent_tab 表不包含 SERIAL 类型。**child1_tab** 引入 SERIAL 计数器到层次结构中。**child2_tab** 从 **child1_tab** 继承 SERIAL 列并添加 SERIAL8 列。**child3_tab** 继承了 SERIAL 和 SERIAL8 列。

插入到层次结构中的任何表的 **s_col** 或 **s8_col** 列中的 0 值将插入单调递增值，而不管此插入操作由哪个表执行。

不能在 CREATE ROW TYPE 语句中设置 SERIAL 或 SERIAL8 类型的起始计数器值。要设置表层次结构中的 SERIAL 或 SERIAL8 列的起始值，可使用

ALTER TABLE 语句。以下语句说明如何改变表以修改将要在表层次结构中的任何位置插入的下一个 SERIAL 和 SERIAL8 值:

```
ALTER TABLE child3_tab
MODIFY (s_col SERIAL(100), s8_col SERIAL8 (200))
```

除先前描述的行为以外，所有适用于无类型表中的 SERIAL 和 SERIAL8 类型的列的规则也都适用于表层次结构中的 SERIAL 和 SERIAL8 类型的列。有关更多信息，请参阅第 3-1 页的第 3 章，『选择数据类型』和《*IBM Informix: SQL 参考指南*》。

将新表添加到表层次结构中

定义表层次结构之后，不能使用 ALTER TABLE 语句来对层次结构中的表添加、删除或修改列。然而，倘若新的子类型和子表不会干扰现有的继承关系，则可将新的子类型和子表添加到现有层次结构中。图 9-7 说明了一种可用来将类型和相应的表添加至现有层次结构的方法。虚线指示添加的子类型和子表。

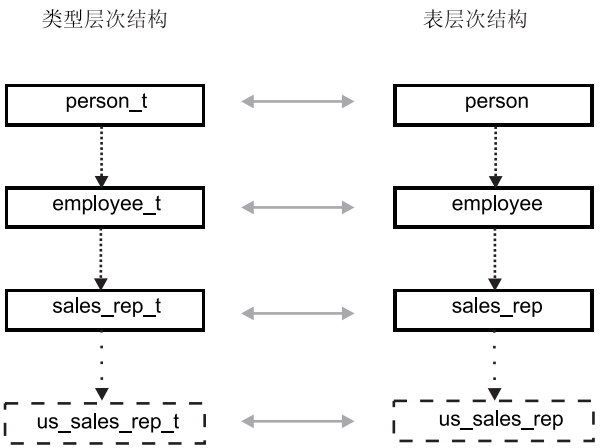


图 9-7. 可以如何将子类型和子表添加至现有继承层次结构的示例

下列语句说明可以如何将类型和表添加至图 9-7 显示的继承层次结构:

```
CREATE ROW TYPE us_sales_rep_t (domestic_sales DECIMAL(15,2))
UNDER employee_t;

CREATE TABLE us_sales_rep OF TYPE us_sales_rep_t
UNDER sales_rep;
```

也可以添加从现有超类型及其并行超表分出来的子类型和子表。图 9-8 显示了可以如何将 **customer_t** 类型和 **customer** 表添加至现有层次结构。在此示例中，**customer** 表和 **employee** 表都从 **person** 表继承属性。

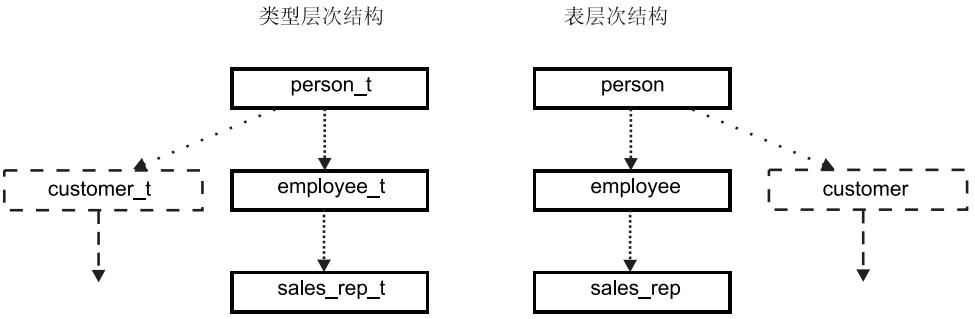


图 9-8. 在现有超类型和超表下面添加类型和表的示例

下列语句分别在 **person_t** 类型和 **person** 表下面创建 **customer_t** 类型和 **customer** 表:

```
CREATE ROW TYPE customer_t (cust_num INTEGER) UNDER person_t;

CREATE TABLE customer OF TYPE customer_t UNDER person;
```

删除表层次结构中的表

如果某个表及其相应命名行类型不具有相关性（它们不是超表和超类型），则可删除该表及其类型。必须先删除表，然后才能删除类型。关于删除表的一般信息，请参阅《*IBM Informix: SQL 指南: 语法*》中的 DROP TABLE 语句。有关如何删除命名行类型的信息，请参阅第 8-19 页的『删除命名行类型』。

改变表层次结构中的表的结构

不能使用 ALTER TABLE 语句来对表层次结构中的表添加、删除或修改列。*可以*使用 ALTER TABLE 语句来添加、删除或修改约束。

对表层次结构中的表添加、删除或修改列（也称改变表的结构）的过程是一项相当耗时的任务。

改变表层次结构中的表的结构:

1. 从所有要修改的子表和超表下载数据。
2. 删除子表和子类型。
3. 修改卸装的数据文件。
4. 修改超表。
5. 重新创建子类型和子表。
6. 上载数据。

查询表层次结构中的表

表层次结构允许在单个 SQL 命令中构造 SELECT、UPDATE 或 DELETE 语句，这些语句的作用域是超表及其子表。例如：对表层次结构中的任何超表执行的查询将返回该超表的所有列的数据以及子表从该超表继承的列的数据。要将查询结果限制为只返回表层次结构中的一个表，必须在查询中包括 ONLY 关键字。有关如何查询和修改表层次结构中的表的数据的更多信息，请参阅《*IBM Informix: SQL 教程指南*》。

对表层次结构中的表创建视图

可以创建基于表层次结构中的任何表的视图。例如：以下语句对 **person** 表创建视图，该表是第 9-7 页的图 9-5 显示的表层次结构的根超表：

```
CREATE VIEW name_view AS SELECT name FROM person
```

由于 **person** 表是一个超表，所以视图 **name_view** 显示 **person**、**employee** 和 **sales_rep** 表的 **name** 列中的数据。要创建只显示 **person** 表中的数据的视图，请使用 ONLY 关键字，如以下示例所示：

```
CREATE VIEW name_view AS SELECT name FROM ONLY(person)
```

要点：不能对定义在超表上的视图执行插入或更新操作，这是因为数据库服务器无法知道要将新行放到表层次结构中的什么位置。

有关如何创建带类型视图的信息，请参阅第 6-20 页的『带类型视图（IDS）』。

第 10 章 在 Dynamic Server 中创建和使用用户定义的数据类型转换

在本章中

本章描述用户定义的数据类型转换并阐述如何使用运行时数据类型转换来对扩展数据类型执行数据转换。

什么是数据类型转换?

*数据类型转换*是用于将值从一种数据类型转换为另一种数据类型的机制。数据类型转换允许在具有不同数据类型的值之间作比较或用一种数据类型的值替代另一数据类型的值。Dynamic Server 在下列类型的表达式中支持数据类型转换:

- 列表达式
- 常量表达式
- 函数表达式
- SPL 变量
- 主变量（**ESQL**）
- 语句局部变量（SLV）表达式

要将一种数据类型的值转换为具有另一数据类型，数据库或数据库服务器中必须存在数据类型转换。Dynamic Server 支持下列类型的数据类型转换:

- **内置数据类型转换** *内置数据类型转换*是构建到数据库服务器中的数据类型转换。内置数据类型转换执行不同内置数据类型之间的自动转换。
- **用户定义的数据类型转换** 用户定义的数据类型转换通常需要*数据类型转换函数*来处理从一种数据类型到另一数据类型的转换。要注册和使用用户定义的数据类型转换，必须使用 CREATE CAST 语句。

如果使用 CREATE CAST 语句创建数据类型转换时包括了 EXPLICIT 关键字，则用户定义的数据类型转换是*显式的*。（缺省选项是显式的。）永远不会自动调用显式数据类型转换。要调用显式数据类型转换，必须使用 CAST...AS 关键字或双冒号（::）数据类型转换运算符。

如果使用 CREATE CAST 语句创建数据类型转换时包括了 IMPLICIT 关键字，则用户定义的数据类型转换是*隐式的*。在运行时，数据库服务器自动调用隐式数据类型转换来执行数据转换。

所有数据类型转换都包括在 **syscasts** 系统目录表中。有关 **syscasts** 的更多信息，请参阅《*IBM Informix: SQL 参考指南*》。

创建用户定义的数据类型转换

当数据库服务器没有提供用于执行两种数据类型之间的转换的内置数据类型转换时，可以创建用户定义的数据类型转换来处理数据类型转换。通常，用户定义的数据类型转换用来为下列扩展数据类型提供数据类型转换:

- **不透明数据类型** 不透明数据类型的开发者必须定义数据类型转换来处理不透明数据类型的内部／外部表示法之间的转换。有关如何为不透明数据类型创建和注册数据类型转换的信息，请参阅《*IBM Informix: 用户定义的例程与数据类型开发者指南*》。
- **单值数据类型** 不能直接将单值数据类型与其源类型作比较。然而，Dynamic Server 自动注册从单值类型到源类型（反之亦然）的显式数据类型转换。单值类型不继承对其源类型定义的数据类型转换。另外，对单值类型定义的用户定义的数据类型转换不可用于其源类型。有关阐述如何对单值类型创建和使用数据类型转换的更多信息及示例，请参阅第 10-13 页的『为用户定义的数据类型转换创建数据类型转换函数』。
- **命名行类型** 在大多数情况下，可以显式地将命名行类型数据类型转换为另一行类型值，而不必创建数据类型转换。然而，要在命名行类型的值与其它一些数据类型的值之间进行转换，必须首先创建数据类型转换来处理该转换。

有关如何创建和使用用户定义的数据类型转换的示例，请参阅第 10-14 页的『单值数据类型之间的数据类型转换的示例』。有关 CREATE CAST 语句的语法，请参阅《*IBM Informix: SQL 指南: 语法*》。

调用数据类型转换

对于内置数据类型转换和用户定义的隐式数据类型转换，数据库服务器自动（隐式地）调用数据类型转换来处理数据转换。例如：可以将 INT 类型的值与 SMALLINT、FLOAT 或 CHAR 值作比较，而无需显式地对表达式进行数据类型转换，这是因为数据库服务器提供了系统定义的数据类型转换来透明地处理这些内置数据类型之间的转换。

当定义用户定义的显式数据类型转换来处理两种数据类型之间的转换时，必须显式地使用 CAST...AS 关键字或双冒号数据类型转换运算符（::）来调用数据类型转换。下列不完整的示例显示了两种调用显式数据类型转换的方法:

```
... WHERE new_col = CAST(old_col AS newtype)

... WHERE new_col = old_col::newtype
```

对用户定义的数据类型转换的限制

不能在两种内置数据类型之间创建用户定义的数据类型转换。也不能创建包括下列任何数据类型的用户定义的数据类型转换:

- 集合数据类型：LIST、MULTISET 或 SET
- 未命名行类型
- 智能大对象数据类型：CLOB 或 BLOB
- 简单大对象数据类型：TEXT 或 BYTE

通常，两种数据类型之间的数据类型转换要求每种数据类型都表示相同数目的组件值。例如: 如果行类型中的每个字段在不透明数据类型中有对应的字段，则可以在行类型与不透明数据类型之间进行数据类型转换。当您想要在两种具有相同存储结构的数据类型之间执行转换时，可使用不带数据类型转换函数的 CREATE CAST 语句。否则，必须创建以后可以使用 CREATE CAST 语句进行注册的数据类型转换函数。有关如何使用数据类型转换函数来创建用户定义的数据类型转换的示例，请参阅第 10-13 页的『为用户定义的数据类型转换创建数据类型转换函数』。

对行类型进行数据类型转换

仅当两种行类型具有相同数目的字段并且下列其中一个条件也成立时，才可以在任何两种行类型（命名行类型或未命名行类型）的值之间进行比较或替代:

- 两种行类型的所有对应字段都具有相同的数据类型。

  当两种行类型具有相同数目的字段并且对应字段的数据类型相同时，将它们视为*在结构上等价*。
- 在比较两种命名行类型时，存在用于执行转换的用户定义的数据类型转换。
- 存在用于对不具有相同数据类型的对应字段值执行必需转换的系统定义的数据类型转换或用户定义的数据类型转换。

  当对应的字段不具有相同的数据类型时，可使用系统定义的数据类型转换或用户定义的数据类型转换来对这些字段处理数据转换。

如果存在用于对个别字段处理数据转换的内置数据类型转换，则可显式地将一种行类型的值数据类型转换为具有其它行类型（除非行类型都是未命名行类型，在这种情况下，显式数据类型转换不是必需的）。

如果不存在用于处理字段转换的内置数据类型转换，则可创建用户定义的数据类型转换来处理字段转换。数据类型转换可以是隐式的，也可以是显式的。

通常，当将行类型数据类型转换为另一行类型时，可使用显式或隐式数据类型转换来处理个别字段转换。因为数据库服务器不对已进行显式数据类型转换的值应用任何附加的隐式数据类型转换，所以当对应字段之间的转换要求进行显式数据类型转换时，进行数据类型转换的字段的值必须与对应字段的值完全匹配。

命名行类型与未命名行类型之间的数据类型转换

要将命名行类型的值与未命名行类型的值作比较，可使用显式数据类型转换。假定创建下列命名行类型和表:

```
CREATE ROW TYPE info_t (x CHAR(1), y CHAR(20))
CREATE TABLE customer (cust_info info_t)
CREATE TABLE retailer (ret_info ROW (a CHAR(1), b CHAR(20)))
INSERT INTO customer2 VALUES(ROW('t','philips')::info_t2)
```

下列 INSERT 语句显示如何创建 **customer** 和 **retailer** 表的行类型值:

```
INSERT INTO customer VALUES(ROW('t','philips')::info_t)
INSERT INTO retailer VALUES(ROW('f','johns'))
```

要将 **customer** 表中的数据与 **retailer** 表中的数据作比较或进行替代，必须使用显式数据类型转换来将一种行类型的值转换为具有另一行类型。在以下查询中，将 **ret_info** 列（未命名行类型）显式地数据类型转换为 **info_t**（命名行类型）:

```
SELECT cust_info
FROM customer, retailer
WHERE cust_info = ret_info::info_t
```

通常，要在命名行类型与未命名行类型之间执行转换，必须显式地将一种行类型数据类型转换为另一行类型。可以在任何一个方向上执行显式数据类型转换：可以将命名行类型数据类型转换为未命名行类型，也可以将未命名行类型数据类型转换为命名行类型。以下语句返回与前一示例相同的结果。但是，本示例中的命名行类型显式地数据类型转换为未命名行类型:

```
SELECT cust_info
FROM customer, retailer
WHERE cust_info::ROW(a CHAR(1), b CHAR(20)) = ret_info
```

未命名行类型之间的数据类型转换

可以在不使用显式数据类型转换的情况下将结构上等价的两种未命名行类型作比较。还可以将一种未命名行类型与另一种未命名行类型作比较，条件是这两种行类型具有相同数目的字段，并且存在用于转换不具有相同数据类型的对应字段的值的数据类型转换。换而言之，如果所有用于处理字段转换的数据类型转换都是系统定义的或者都是隐式数据类型转换，则从一种未命名行类型到另一种未命名行类型的数据类型转换是隐式的。否则，必须显式地对未命名行类型进行数据类型转换才能将其与另一种行类型作比较。

假定创建以下 **prices** 表:

```
CREATE TABLE prices
(col1  ROW(a SMALLINT, b FLOAT)
 col2  ROW(x INT, y REAL) )
```

当存在用于在对应字段之间执行转换的内置数据类型转换时，可以对两种未命名行类型的值作比较（无需使用显式数据类型转换）。因此，以下查询不需要显式数据类型转换就可以将 **col1** 与 **col2** 值作比较:

```
SELECT * FROM prices WHERE col1 = col2
```

在本示例中，数据库服务器隐式地调用内置数据类型转换来将 SMALLINT 的字段值转换为 INT 并将 REAL 转换为 FLOAT。

如果两种行类型的对应字段不能隐式地相互进行数据类型转换，则可显式地在类型之间进行数据类型转换（如果存在用于处理这两种类型之间的转换的用户定义的数据类型转换的话）。

命名行类型之间的数据类型转换

命名行类型是强类型化的，这表示即使两种命名行类型在结构上等价，数据库服务器也将这两种行类型识别成两种独立的类型。因此，在可以在两种命名行类型之间执行比较之前，必须创建并注册用户定义的数据类型转换。有关如何创建和使用数据类型转换来处理两种命名行类型之间的转换的示例，请参阅第 10-13 页的『命名行类型之间的数据类型转换的示例』。

对字段使用显式数据类型转换

必须先存在一种用于处理对应字段数据类型之间的转换的数据类型转换（由系统定义或用户定义），才能在字段中包含不同数据类型的两种行类型（命名行类型或未命名行类型）之间进行显式数据类型转换。

当在两种行类型之间进行显式数据类型转换时，数据库服务器自动调用处理字段数据类型之间的转换所必需的任何显式数据类型转换。换而言之，当对行类型值执行显式数据类型转换时，除非必须进行多个级别的数据类型转换来对行类型的个别字段处理数据类型转换，否则不必显式地对该字段进行数据类型转换。

在整个本节中，使用以下示例中的行类型和表来阐述对命名和未命名行类型进行的显式数据类型转换的行为:

```
CREATE DISTINCT TYPE d_float AS FLOAT;
CREATE ROW TYPE row_t (a INT, b d_float);

CREATE TABLE tab1 (col1 ROW (a INT, b d_float));
CREATE TABLE tab2(col2 ROW (a INT, b FLOAT));
CREATE TABLE tab3 (col3 row_t);
```

对未命名行类型的字段进行的显式数据类型转换

当两种行类型之间的转换涉及显式数据类型转换以在特定字段值之间进行转换时，可以显式地对行类型值进行数据类型转换，但不需要显式地对个别字段进行数据类型转换。

以下语句显示如何将值插入到 **tab1** 表中:

```
INSERT INTO tab1 VALUES (ROW( 3, 5.66::FLOAT::d_float))
```

要将 **tab1** 的 **col1** 中的值插入到 **tab2** 的 **col2** 中，由于数据库服务器不会自动处理 **tab1** 的 **d_float** 单值类型到 **tab2** 表的 FLOAT 类型的转换，所以必须显式地对行值进行数据类型转换:

```
INSERT INTO tab2 SELECT col1::ROW(a INT, b FLOAT) FROM tab1
```

在本示例中，由于从 **d_float** 到 FLOAT 的转换要求进行显式数据类型转换（将单值类型转换为它的源类型要求进行显式数据类型转换），所以用来转换 **b** 字段的数据类型转换是显式的。

通常，要在两个未命名行类型之间进行数据类型转换，并且一个或多个字段使用显式数据类型转换，则必须在行类型级别而不是字段级别进行显式数据类型转换。

对命名行类型的字段进行的显式数据类型转换

当显式地将值数据类型转换为命名行类型时，数据库服务器自动调用任何用来将字段值转换为目标数据类型的隐式或显式数据类型转换。在以下语句中，**col1** 到类型 **row_t** 的显式数据类型转换将自动调用用于将 FLOAT 类型的字段值转换为 **d_float** 的显式数据类型转换:

```
INSERT INTO tab3 SELECT col2::row_t FROM tab2
```

以下 INSERT 语句包括针对 **row_t** 类型的显式数据类型转换。针对行类型的显式数据类型转换还调用显式数据类型转换来将 **row_t** 类型的 **b** 字段从 FLOAT 转换为 **d_float**。通常，针对行类型的显式数据类型转换还对行类型包含的个别字段（深度为一层）调用任何显式数据类型转换以处理转换。

```
INSERT INTO tab3 VALUES (ROW(5, 6.55::FLOAT)::row_t)
```

以下语句也有效，它与前面这个语句返回相同的结果。然而，此语句显示了为了将 **row_t** 值插入到 **tab3** 表中而执行的所有显式数据类型转换。

```
INSERT INTO tab3 VALUES (ROW(5, 6.55::float::d_float)::row_t)
```

在前面的示例中，行类型的 **b** 字段之间的转换要求进行两个级别的数据类型转换。数据库服务器将任何包含小数点的值作为 DECIMAL 类型的值来处理。另外，在 DECIMAL 与 **d_float** 数据类型之间不存在隐式数据类型转换，因此需要进行两个级别的数据类型转换: 从 DECIMAL 到 FLOAT 的数据类型转换以及从 FLOAT 到 **d_float** 的第二个数据类型转换。

对行类型的个别字段进行数据类型转换

如果对行类型的某个字段执行的操作要求进行显式数据类型转换，则可显式地对个别字段值进行数据类型转换，而不必考虑与该字段相关联的行类型。以下语句使用对字段值执行的显式数据类型转换来处理转换:

```
SELECT col1 from tab1, tab2 WHERE col1.b = col2.b::FLOAT::d_float
```

如果对行类型的某个字段执行的操作要求进行隐式数据类型转换，则可以简单地指定适当的字段值，数据库服务器将自动处理转换。在以下语句中（此语句对不同数据类型的字段值作比较），内置数据类型转换自动地在 INT 与 FLOAT 值之间进行转换:

```
SELECT col1 from tab1, tab2 WHERE col1.a = col2.b
```

对集合数据类型进行数据类型转换

在某些情况下，可使用显式数据类型转换来在两个具有不同元素类型的集合之间执行转换。要在任何两种集合类型的值之间进行比较或替代，两个集合都必须具有 SET、MULTISET 或 LIST 类型。

- 当所有组件类型相同时，两种元素类型是等价的。例如: 如果一个集合的元素类型是行类型，则另一集合类型也是行类型，并且具有相同数目的字段和相同的字段数据类型。
- 数据库中存在用于在元素类型的不具有相同数据类型的任何以及所有组件之间执行转换的数据类型转换。

  如果对应的元素类型不具有相同的数据类型，则 Dynamic Server 可使用内置数据类型转换或用户定义的数据类型转换来对这些元素类型处理数据转换。

当数据库服务器对集合数据类型的值进行插入、更新或比较时，在元素数据类型级别进行类型检查。因此，在两种集合类型之间的数据类型转换中，由于存储在集合中的实际数据具有特定的元素类型，所以数据转换在元素类型的级别发生。

在本节中的集合数据类型转换示例中，使用下列类型和表:

```
CREATE DISTINCT TYPE my_int AS INT;

CREATE TABLE set_tab1 (col1 SET(my_int NOT NULL));
CREATE TABLE set_tab2 (col2 SET(INT NOT NULL));
CREATE TABLE set_tab3 (col3 SET(FLOAT NOT NULL));
CREATE TABLE list_tab (col4 LIST(INT NOT NULL));
CREATE TABLE m_set_tab(col5 MULTISET(INT NOT NULL));
```

对集合类型转换的限制

由于每种集合数据类型（SET、MULTISET 和 LIST）具有不同的特征，所以不允许在具有不同集合类型的集合之间进行转换。例如: 存储在 LIST 集合中的元素有特定的顺序与它们相关联。如果插入到 LIST 集合中的元素可以插入到 MULTISET 集合中，则此顺序将丢失。因此，即使两个集合共享同一元素类型，也不能使用

来自具有另一集合类型的集合的元素来对一个集合插入或更新元素。以下 INSERT 语句将返回错误，这是因为对其执行插入操作的列是 MULTISET 集合，并且正在插入的值是 LIST 集合:

```
INSERT INTO m_set_tab SELECT col4 FROM list_tab -- returns error
```

具有不同元素类型的集合

对两个具有相同集合类型但具有不同元素类型的集合之间的转换的处理方式取决于每个集合的元素类型以及当元素类型不相同时数据库服务器用来将一种元素类型转换为另一元素类型的数据类型转换的类型，如下所示:

- 如果存在用于处理两种元素类型之间的转换的内置数据类型转换或隐式用户定义的数据类型转换，则不需要显式地在集合类型之间进行数据类型转换。
- 如果存在用于处理元素类型之间的转换的显式数据类型转换，则可对一个集合执行显式数据类型转换。

在元素类型之间使用隐式数据类型转换

当数据库中存在用于在两个集合的不同元素类型之间进行转换的隐式数据类型转换时，不需要使用显式数据类型转换即可从一个集合对另一集合插入或更新元素。以下 INSERT 语句从 **set_tab2** 表中检索元素并将元素插入到 **set_tab3** 表中。尽管 **set_tab2** 中的集合列具有 INT 元素类型并且 **set_tab3** 中的集合列具有 FLOAT 元素类型，但一个内置数据类型转换隐式地处理 INT 与 FLOAT 值之间的转换。在此情况下，没有必要进行显式数据类型转换。

```
INSERT INTO set_tab3 SELECT col2 FROM set_tab2
```

在元素类型之间使用显式数据类型转换

当两个集合的不同元素类型之间的转换是使用显式数据类型转换执行时，必须显式地将一个集合数据类型转换为另一个集合类型。在以下示例中，元素类型（INT 和 **my_int**）之间的转换要求进行显式数据类型转换。（单值类型与其源类型之间的数据类型转换总是显式的。）

以下 INSERT 语句从 **set_tab2** 表中检索元素并将元素插入到 **set_tab1** 表中。**set_tab2** 中的集合列具有 INT 元素类型，**set_tab1** 中的集合列具有 **my_int** 元素类型。由于元素类型（INT 和 **my_int**）之间的转换要求进行显式数据类型转换，所以必须显式地对集合类型进行数据类型转换。

```
INSERT INTO set_tab1 SELECT col2::SET(my_int NOT NULL)
   FROM set_tab2
```

要对集合类型执行显式数据类型转换，必须包括构造函数（SET、MULTISET 或 LIST）、元素类型和 NOT NULL 关键字。

将关系数据转换为 MULTISET 集合

当数据来自关系表时，可使用集合子查询来将行值数据类型转换为 MULTISET 集合。假定创建下列表:

```
CREATE TABLE tab_a ( a_col INTEGER);
CREATE TABLE tab_b (ms_col MULTISET(ROW(a INT) NOT NULL) );
```

以下示例显示可以如何使用集合子查询来将 **tab_a** 表中的 INT 值行转换为 MULTISET 集合。将把 **tab_a** 中的所有各行都转换为 MULTISET 集合并插入到 **tab_b** 表中。

```
INSERT INTO tab_b VALUES (
   (MULTISET (SELECT a_col FROM tab_a)))
```

对单值数据类型进行数据类型转换

单值类型不继承可以由某个单值类型用作其源类型的内置类型的任何内置数据类型转换。因此，存在的用于隐式地将内置数据类型转换为其它数据类型的内置数据类型转换不可用于使用该内置类型作为其源类型的单值类型。然而，当创建基于内置类型的单值类型时，数据库服务器提供了两个显式数据类型转换来处理从单值类型到内置类型以及从内置类型到单值类型的转换。

对单值类型使用显式数据类型转换

要在单值类型与其源类型的值之间进行比较或替代，必须显式地将一种类型数据类型转换为另一类型。例如: 要使用源类型的值来对单值类型的列进行插入或更新，必须显式地将值数据类型转换为单值类型。

假定创建基于 INTEGER 数据类型的单值类型 **int_type** 以及带有类型为 **int_type** 的列的表，如下所示:

```
CREATE DISTINCT TYPE int_type AS INTEGER;
CREATE TABLE tab_z(col1 int_type);
```

要将值插入到 **tab_z** 表中，必须显式地将 **col1** 列的值数据类型转换为 **int_type**，如下所示:

```
INSERT INTO tab_z VALUES (35::int_type)
```

假定创建基于 NUMERIC 数据类型的单值类型 **num_type** 以及带有类型为 **num_type** 的列的表，如下所示:

```
CREATE DISTINCT TYPE num_type AS NUMERIC;
CREATE TABLE tab_x (col1 num_type);
```

单值 **num_type** 不继承对 NUMERIC 数据类型存在的系统定义的数据类型转换。因此，以下插入操作要求进行两个级别的数据类型转换。第一个数据类型转换将值 35 从 INT 转换为 NUMERIC，第二个数据类型转换从 NUMERIC 转换为 num_type:

```
INSERT INTO tab_x VALUES (35::NUMERIC::num_type)
```

由于不存在用于直接从 INT 类型转换为 **num_type** 的数据类型转换，所以，对 **tab_x** 表执行的以下 INSERT 语句将返回错误:

```
INSERT INTO tab_x VALUES (70::num_type) -- returns error
```

在单值类型及其源类型之间进行数据类型转换

虽然具有单值类型的数据与该单值类型的源类型具有相同的表示法，但单值类型不能直接与其源类型进行比较。因此，创建单值数据类型时，Dynamic Server 自动注册下列显式数据类型转换:

- 从单值类型到其源类型的数据类型转换
- 从源类型到单值类型的数据类型转换

假定创建两个单值类型：一个用于处理电影字幕，另一个用于处理音乐录音。可以创建下列基于 VARCHAR 数据类型的单值类型:

```
CREATE DISTINCT TYPE movie_type AS VARCHAR(30);
CREATE DISTINCT TYPE music_type AS VARCHAR(30);
```

然后，可以创建 **entertainment** 表，它包含类型为 **movie_type**、**music_type** 和 VARCHAR 的列。

```
CREATE TABLE entertainment
(
video         movie_type,
compact_disc  music_type,
laser_disv    VARCHAR(30)
);
```

要将单值类型与其源类型作比较（反之亦然），必须执行从一种数据类型到另一数据类型的显式数据类型转换。例如: 假定您想要查找同时提供了录影带和激光影碟的电影。以下语句要求在 WHERE 子句中指定显式数据类型转换，以将具有单值类型（**music_type**）的值与具有其源类型（VARCHAR）的值作比较。在本示例中，将源类型显式地数据类型转换为单值类型。

```
SELECT video FROM entertainment
   WHERE video = laser_disc::movie_type
```

然而，还可以显式地将单值类型数据类型转换为源类型，如以下语句所示:

```
SELECT video FROM entertainment
   WHERE video::VARCHAR(30) = laser_disc
```

要在对同一源类型定义的两个单值类型之间执行转换，必须先执行中间数据类型转换以转换回源类型，然后再将数据类型转换为目标单值类型。以下语句将 **music_type** 的值与 **movie_type** 的值作比较:

```
SELECT video FROM entertainment
   WHERE video = compact_disc::VARCHAR(30)::movie_type
```

对单值类型添加和删除数据类型转换

为了对单值类型强制执行强类型化，数据库服务器提供了显式数据类型转换来处理单值类型与其源类型之间的转换。然而，单值类型的创建者可以删除现有的显式数据类型转换并创建隐式数据类型转换，以使单值类型与其源类型之间的转换不要求进行显式数据类型转换。

要点: 当删除单值类型与其源类型之间的由数据库服务器提供的显式数据类型转换，然后改为创建隐式数据类型转换来处理这些数据类型之间的转换时，就无法充分发挥单值类型的独特性。

以下 DROP CAST 语句删除两个自动对 **movie_type** 定义的显式数据类型转换:

```
DROP CAST(movie_type AS VARCHAR(30));
DROP CAST(VARCHAR(30) AS movie_type);
```

在删除现有数据类型转换之后，可以创建两个隐式数据类型转换来处理 **movie_type** 与 VARCHAR 之间的转换。下列 CREATE CAST 语句创建两个隐式数据类型转换:

```
CREATE IMPLICIT CAST (movie_type AS VARCHAR(30));
CREATE IMPLICIT CAST (VARCHAR(30) AS movie_type);
```

如果数据库中已存在用于在两种数据类型之间进行转换的数据类型转换，则不能创建这样的数据类型转换。

如果创建隐式数据类型转换来在单值类型与其源类型之间进行转换，则可以在不进行显式数据类型转换的情况下对两种类型作比较。在以下语句中，**video** 列与 **laser_disc** 列之间的比较要求进行转换。由于已创建隐式数据类型转换，所以 VARCHAR 与 **movie_type** 之间的转换是隐式的。

```
SELECT video FROM entertainment
   WHERE video = laser_disc
```

数据类型转换为智能大对象

数据库服务器提供了数据类型转换以允许将 TEXT 和 BYTE 对象转换为 BLOB 和 CLOB 数据类型。此功能允许用户将旧数据库中的 BYTE 和 TEXT 数据迁移到 BLOB 和 CLOB 列中。

以下示例显示如何使用显式数据类型转换来将 **stores_demo** 数据库中的 **catalog** 表中的 BYTE 列值转换为 BLOB 列值并更新 **superstores_demo** 数据库中的 **catalog** 表:

```
UPDATE catalog SET advert = ROW (
(SELECT cat_photo::BLOB FROM stores_demo:catalog
   WHERE catalog_num = 10027),
   advert.caption)
      WHERE catalog_num = 10027
```

数据库服务器没有提供用于将 BLOB 转换为 BYTE 值或将 CLOB 转换为 TEXT 值的数据类型转换。

为用户定义的数据类型转换创建数据类型转换函数

如果数据库包含不透明数据类型、单值数据类型或命名行类型，则您可能想创建允许在不同数据类型之间进行转换的用户定义的数据类型转换。当您想要在两种具有相同存储结构的数据类型之间执行转换时，可使用不带数据类型转换函数的 CREATE CAST 语句。然而，在某些情况下，必须创建以后可以注册为数据类型转换的数据类型转换函数。在下列情况下，需要创建数据类型转换函数:

- 转换操作是在两种具有不同存储结构的数据类型之间进行的
- 转换操作涉及对值进行处理以确保数据转换有意义

下列各节阐述如何创建和使用需要数据类型转换函数的用户定义的数据类型转换。

命名行类型之间的数据类型转换的示例

假定创建下一示例中显示的命名行类型和表。尽管命名行类型在结构上相当，但是 **writer_t** 和 **editor_t** 均是唯一的数据类型。

```
CREATE ROW TYPE writer_t (name VARCHAR(30), depart CHAR(3));
CREATE ROW TYPE editor_t (name VARCHAR(30), depart CHAR(3));

CREATE TABLE projects
(
   book_title  VARCHAR(20),
   writer      writer_t,
   editor      editor_t
);
```

要处理两种命名行类型之间的转换，必须首先创建用户定义的数据类型转换。以下示例创建一个数据类型转换函数并将其注册为数据类型转换，以处理从类型 **writer_t** 到 **editor_t** 的转换:

```
CREATE FUNCTION cast_rt (w writer_t)
   RETURNS editor_t
   RETURN (ROW(w.name, w.depart)::editor_t);
```

```
END FUNCTION;

CREATE CAST (writer_t as editor_t WITH cast_rt);
```

一旦创建并注册数据类型转换之后，可以显式地将类型为 **writer_t** 的值数据类型转换为具有 **editor_t** 类型。以下查询在 WHERE 子句中使用显式数据类型转换来将类型为 **writer_t** 的值转换为具有 **editor_t** 类型:

```
SELECT book_title FROM projects
   WHERE CAST(writer AS editor_t) = editor;
```

如果您愿意的话，可使用 :: 数据类型转换运算符来执行同一数据类型转换，如以下示例所示:

```
SELECT book_title FROM projects
   WHERE writer::editor_t = editor;
```

单值数据类型之间的数据类型转换的示例

假定您想要定义单值类型来表示 **dollar**、**yen** 和 **sterling** 货币。两种货币之间的任何比较都必须考虑汇率。因此，您需要创建的数据类型转换函数不仅需要处理从一种数据类型到另一数据类型的数据类型转换，还需要计算要比较的值的汇率。

以下示例显示如何对同一源类型 DOUBLE PRECISION 定义三种单值类型:

```
CREATE DISTINCT TYPE dollar AS DOUBLE PRECISION;
CREATE DISTINCT TYPE yen AS DOUBLE PRECISION;
CREATE DISTINCT TYPE sterling AS DOUBLE PRECISION;
```

定义单值类型后，您可以创建一个表，这个表提供制造商对可比较产品开出的价格。以下示例创建 **manufact_price** 表，这个表对 **dollar**、**yen** 和 **sterling** 单值类型各包含一列:

```
CREATE TABLE manufact_price
(
product_desc  VARCHAR(20),
us_price      dollar,
japan_price   yen,
uk_price      sterling
);
```

在将值插入 **manufact_price** 表时，可以将数据类型转换为 **dollar**、**yen** 和 **sterling** 值的适当单值类型，如下所示:

```
INSERT INTO manufact_price
   VALUES ('baseball', 5.00::DOUBLE PRECISION::dollar,
   510.00::DOUBLE PRECISION::yen,
   3.50::DOUBLE PRECISION::sterling);
```

由于单值类型不继承任何可用于其源类型的内置数据类型转换，所以前面每个 INSERT 语句都要求进行两次数据类型转换。对于每个 INSERT 语句，内部数据类型转换从 DECIMAL 转换为 DOUBLE PRECISION，外部数据类型转换从 DOUBLE PRECISION 转换为适当的单值类型（**dollar**、**yen** 或 **sterling**）。

在可以对 **dollar**、**yen** 和 **sterling** 数据类型作比较之前，必须创建数据类型转换函数并将它们注册为数据类型转换。以下示例创建可用来对 **dollar**、**yen** 和 **sterling** 值作比较的 SPL 函数。每个函数都将输入值乘以一个反映汇率的值。

```
CREATE FUNCTION
dollar_to_yen(d dollar)
   RETURN (d::DOUBLE PRECISION * 106)::CHAR(20)::yen;
END FUNCTION;

CREATE FUNCTION sterling_to_dollar(s sterling)
   RETURNS dollar
   RETURN (s::DOUBLE PRECISION * 1.59)::CHAR(20)::dollar;
END FUNCTION;
```

在编写数据类型转换函数之后，必须使用 CREATE CAST 语句来将函数注册为数据类型转换。下列语句将 **dollar_to_yen()** 和 **sterling_to_dollar()** 函数注册为显式数据类型转换:

```
CREATE CAST(dollar AS yen WITH dollar_to_yen);
CREATE CAST(sterling AS dollar WITH sterling_to_dollar);
```

在将函数注册为数据类型转换之后，将其用于需要在数据类型之间进行转换的运算。有关用来创建数据类型转换函数并将其注册为数据类型转换的语法，请参阅《*IBM Informix: SQL 指南: 语法*》中的 CREATE FUNCTION 和 CREATE CAST 语句。

在以下查询中，WHERE 子句包含一个显式数据类型转换，该数据类型转换调用 **dollar_to_yen()** 函数来将 **dollar** 与 **yen** 值作比较:

```
SELECT * FROM manufact_price
   WHERE CAST(us_price AS yen) < japan_price;
```

以下查询使用数据类型转换运算符来执行上述查询中显示的同一转换:

```
SELECT * FROM manufact_price
   WHERE us_price::yen < japan_price;
```

也可以使用显式数据类型转换来转换查询返回的值。以下查询使用数据类型转换来返回与 **dollar** 值相等的 **yen**。查询的 WHERE 子句还使用显式数据类型转换来将 **dollar** 与 **yen** 值作比较。

```
SELECT us_price::yen, japan_price FROM manufact_price
   WHERE us_price::yen < japan_price;
```

多级别数据类型转换

*多级别数据类型转换*是指这样的操作：它要求在表达式中进行两个或更多个级别的数据类型转换，以将一种数据类型的值转换为目标数据类型。由于 **yen** 与 **sterling** 值之间不存在数据类型转换，所以将两种数据类型作比较的查询要求进行多次数据类型转换。第一次（内部）数据类型转换将 **sterling** 值转换为 **dollar** 值；第二次（外部）数据类型转换将 **dollar** 值转换为 **yen** 值。

```
SELECT * FROM manufact_price
   WHERE japan_price < uk_price::dollar::yen
```

可以添加另一个数据类型转换函数以直接处理 **yen** 到 **sterling** 的转换。以下示例创建函数 **yen_to_sterling()** 并将其注册为数据类型转换。为了考虑汇率，此函数将 **yen** 值乘以 .01 以派生出等价的 **sterling** 值：

```
CREATE FUNCTION yen_to_sterling(y yen)
   RETURNS sterling
   RETURN (y::DOUBLE PRECISION * .01)::CHAR(20)::sterling;
END FUNCTION;

CREATE CAST (yen AS sterling WITH yen_to_sterling);
```

添加了 **yen** 到 **sterling** 的数据类型转换之后，可使用单级别数据类型转换来对 **yen** 和 **sterling** 值作比较，如以下查询所示：

```
SELECT japan_price::sterling, uk_price FROM manufact_price
   WHERE japan_price::sterling) < uk_price;
```

在 SELECT 语句中，显式数据类型转换返回 **yen** 值作为它们的 **sterling** 等价值。在 WHERE 子句中，数据类型转换允许对 **yen** 和 **sterling** 值作比较。

第 4 部分 维数据库

第 11 章 构建维数据模型

在本章中

本章描述维数据建模的概念和技术，并阐述如何构建一个简单的维数据模型。第 12-1 页的第 12 章，『实现维数据库（XPS）』阐述如何使用 SQL 来实现此维数据模型。

与维护关系数据模型相比，维护非常大型的数据仓库的维数据模型要困难得多。因此，数据仓库通常基于关系数据模型。然而，维数据模型特别适合于构建数据集市（数据仓库的子集）。

本章讨论的维数据建模的一般原理适用于使用 Dynamic Server 或 Extended Parallel Server 创建的数据库。尽管没有任何因素可以确定您应该使用哪个数据库服务器来构建维数据库，但假定的情况是使用 Extended Parallel Server 来构建大型的可伸缩仓库，并使用 Dynamic Server 来构建比较小的仓库、OLTP 系统和可操作系统。

为了理解维数据建模的概念，您应该掌握 SQL 和关系数据库理论的基本知识。本章只提供数据仓储概念的概述并描述一个简单的维数据模型。

数据仓储概述

从该术语的广义上来讲，*数据仓库*已用来表示存储了非常大量的历史数据的数据库。数据是作为一系列快照存储的，在这些快照中，每个记录都表示特定时间的数据。此数据快照允许用户重新构造历史并对不同的时间周期作准确的比较。数据仓库在将检索到的数据装入仓库之前先对该数据进行集成和转换。数据仓库的主要优点是提供了对所存储的大量信息的简易存取和分析。

术语“数据仓库”对不同的人可能意味着不同的含义。本手册使用统一性术语*数据仓储*和*数据仓储环境*来统称可用来存储数据的下列任何形式:

- 数据仓库

  为进行数据检索而作了优化的数据库。数据不是在事务级别存储的；一些级别的数据已作了归纳。与传统 OLTP 数据库（此类数据库自动执行日常操作）不同，数据仓库提供决策支持环境，您可以在此环境中对*整个企业*在一段时间内的业绩作出评估。通常，使用关系数据模型来构建数据仓库。

- 数据集市

  数据仓库的一个子集，它存储在比较小的数据库中并且面向特定的用途或数据主题，而不是用于整个企业的战略规划。数据集市可以包含运营数据、总结数据、维数据或元数据。通常，使用维数据模型来构建数据集市。

- 运营数据存储

  面向主题的系统，为了每次查找一两个记录以进行决策，对此系统作了优化。运营数据存储是数据仓库的混合形式，它包含定时信息、当前信息和经过集成的信息。数据通常具有比事务更高级别的粒度。可将运营数据存储用于日常的文书决策。此数据可用作数据仓库的公共数据源。

- 资源库

  资源库将多个数据源组合到一个规范化的数据库中。资源库中的记录频繁更新。数据是运营数据，而不是历史数据。根据特定系统需求的不同，可将资源库用于特定的决策支持查询。资源库能够满足需要具有企业范围的集成数据源来进行操作性处理的企业的需求。

为何构建维数据库?

通常，为进行联机事务处理（OLTP）而对关系数据库进行优化。 OLTP 系统被设计为能够满足企业的日常运作需要，并且针对那些运作需要对数据库性能进行了

调整。因此，数据库可以快速地检索少量记录，但是，如果您需要匆忙地检索大量记录并汇总数据，则速度可能会比较慢。OLTP 系统的一些潜在缺点如下:

- 在工商企业之间，数据可能不一致。
- 对数据的存取可能比较复杂。

相反，维数据库是为了支持分析商业趋势及预测而设计和调整的。此类信息性处理称为联机分析处理（OLAP）或决策支持处理。OLAP 也是数据库设计者用来描述进行信息性处理的维方法的术语。

维数据库为数据检索和分析作了优化。装入到数据库中的任何新数据通常都是以批处理方式更新的，并且通常是从多个源获取的。OLTP 系统倾向于围绕特定过程（如订单输入）组织数据，而维数据库倾向于面向主题并且以回答“哪些产品比较好卖”、“产品在一年中的哪些时候卖得最好”以及“在哪些地区滞销”之类的问题作为目标。

下表总结了 OLTP 与 OLAP 数据库之间的关键区别。

关系数据库（OLTP）	**维数据库（OLAP）**
数据是原子化的	数据经过汇总
数据是当前的	数据是历史的
每次处理一个记录	每次处理多个记录
面向过程	面向主题
是为高度结构化的重复处理设计的	是为高度非结构化的分析处理设计的

企业尝试通过关系技术解决的许多问题在本质上是多维的。例如: 对于创建按地区分类的产品销售汇总以及按产品分类的地区销售量汇总（等等）的 SQL 查询，在传统的关系数据库上可能需要进行数个小时的处理。然而，维数据库可以在一小段时间内处理同样的查询。

除了本章讨论的 OLTP 和 OLAP 数据库之间的特征模式设计差异，通常还应该针对这两种类型的任务对查询优化器进行不同的调整。例如: 在 OLTP 操作中，OPTCOMPIND 设置（由环境变量或 OPTCOMPIND 配置参数指定）通常应该设置为零以支持嵌套循环连接。而 OLAP 操作相反，当 OPTCOMPIND 设置为 2（支持基于成本的查询计划）时则更为有效。有关 **OPTCOMPIND** 环境变量和 OPTCOMPIND 配置参数的更多信息，请分别参阅《*IBM Informix: SQL 参考指南*》和 《*IBM Informix: 管理员参考大全*》。请参阅 《*IBM Informix: 性能指南*》，获得有关 OPTCOMPIND、连接方法和查询优化器的更多信息。

（Dynamic Server 也支持 SET ENVIRONMENT OPTCOMPIND 语句，从而在同时需要 OLTP 和 OLAP 操作的会话过程中动态更改 OPTCOMPIND 设置。有关 SQL 的 SET ENVIRONMENT 语句的更多信息，请参阅《*IBM Informix: SQL 指南: 语法*》。）

什么是维数据?

传统的关系数据库是围绕记录列表组织的。每个记录都包含组织成属性（字段）的相关信息。**stores_demo** 演示数据库的 **customer** 表（它包含姓名、公司、地址以及电话等字段）就是一个典型的示例。虽然这个表带有若干信息字段，但表中的每一行只与一个客户相关。如果要创建带有客户名和任何其它字段（例如: 电话号码）的两维矩阵，您就会意识到只存在一一对应关系。表 11-1 显示了包含只具有一一对应关系的字段的表。

表 11-1. 在字段之间只存在一一对应关系的表

Customer	**Phone number --->**		
Ludwig Pauli	408-789-8075	----------	----------
Carole Sadler	----------	415-822-1289	----------
Philip Currie	----------	----------	414-328-4543

可以将前面 **customer** 表中的字段的任意组合放到此矩阵中，但最终总会得到一一对应关系，这表示这个表不是多维的，并且不是很适合用于维数据库。

然而，请考虑在表的字段之间不仅包含一一对应关系的关系表。假定您创建一个表，该表包含在国家或地区的每个地区销售的产品的销售数据。为了简单起见，假定公司在三个地区销售三种产品。表 11-2 显示了可以如何在关系表中存储此数据。

表 11-2. 一个简单的关系表

Product	**Region**	**Unit Sales**
Football	East	2300
Football	West	4000
Football	Central	5600
Tennis racket	East	5500
Tennis racket	West	8000
Tennis racket	Central	2300
Baseball	East	10000
Baseball	West	22000
Baseball	Central	34000

对于第 11-4 页的表 11-2 中的表，由于每个地区有多种产品，并且每种产品也有多个地区，所以这个表将其自身导向多维表示法。表 11-3 显示了一个两维矩阵，这个矩阵能够更好地表示产品与地区数据之间的多对多关系。

表 11-3. 一个简单的两维示例

	Region	Central	East	West
Product	Football	5600	2300	4000
	Tennis Racket	2300	5500	8000
	Baseball	34000	10000	22000

虽然可以将此数据强制放到表 11-2 的包含三个字段的关系表中，但数据可以更加自然地放到表 11-3 的两维矩阵中。

与传统的关系表相比，*维*表的性能优点可能相当显著。维方法简化了对要进行汇总或比较的数据的存取。例如: 如果使用维表来查询产品在西部的销售量，则数据库服务器将查找 **West** 列并计算该列中的所有行值的总和。要对关系表执行同一查询，数据库服务器必须搜索并检索其 **Region** 列等于 west 的每一行然后再聚集数据。在此类查询中，维表对 **West** 列的所有值进行总计所需的时间，只是关系表查找所有 West 记录所需时间的一小部分。

维数据建模的概念

要构建维数据库，从维数据模型入手。维数据模型提供了一种简化数据库并使其易于理解的方法。您可以将维数据库想象成一个三维或四维的数据库*立方体*，在这个立方体中，用户可存取数据库的任何一个维上的内容。要创建维数据库，您需要允许将数据可视化的模型。

假定您的企业在不同的市场销售产品并对一段时间内的业绩作评估。很容易将此业务流程想象成数据立方体，这个立方体包含时间、产品和市场的维。图 11-1 显示了这个维模型。立方体线上的各个交叉将包含商业*量度*。这些量度与产品、市场和时间数据的特定组合相对应。

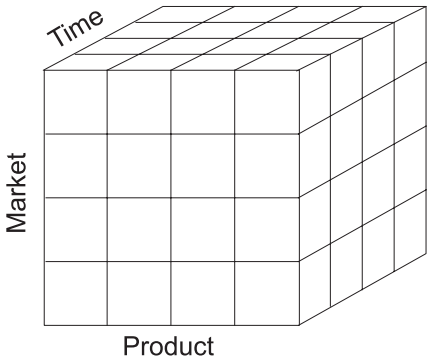


图 11-1. 企业中带有 Time 维、Product 维和 Market 维的维模型

维模型的另一个名称是*星形连接模式*。由于此模型的图看起来象是一颗星（一个中央表，周围显示一组其它的表），所以数据库设计者使用此名称。中央表是模式中唯一一个通过多个连接来连接至所有其它表的表。这个中央表称为*事实表*，其它表称为*维表*。维表全都只具有单一连接，此连接将这些维表连接至事实表，并且与查询无关。图 11-2 显示了一个企业的简单维模型，该企业在不同的市场销售产品并对一段时间内的业绩作评估。

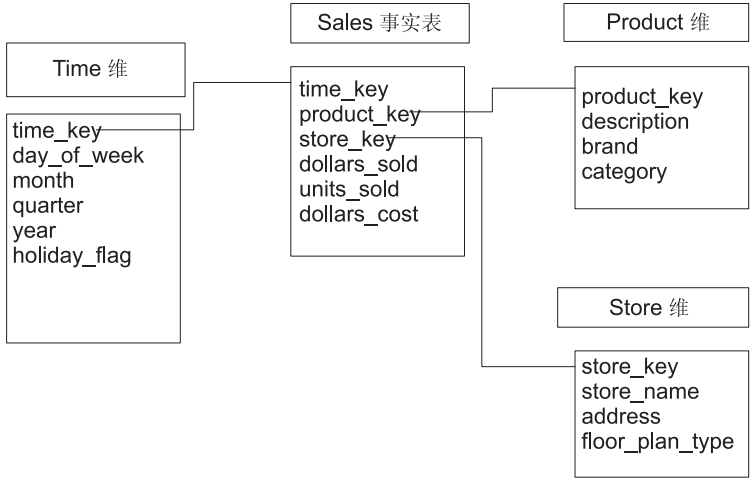


图 11-2. 典型的维模型

事实表

事实表存储企业的量度并指向每个维表的最低级别键值。*量度*是关于主题的定量或真实数据。量度通常是数字，并且与问题的*几个*或*多少*方面相对应。量度的示例包括价格、产品销量、产品库存以及收入等。量度可以基于表中的某列，它也可以是计算得到的。

表 11-4 显示了一个事实表，它的量度是售出件数之和、收入以及该日该帐户的该产品销售利润。

表 11-4. 带有样本记录的事实表

Product Code	Account Code	Day Code	Units Sold	Revenue	Profit
1	5	32104	1	82.12	27.12
3	17	33111	2	171.12	66.00
1	13	32567	1	82.12	27.12

在设计事实表之前，必须确定事实表的*粒度*。粒度与如何定义该事实表中的个别低级别记录相对应。粒度可以是个别的事务、每日快照或每月快照。表 11-4 中的事实表对每天售出给每个帐户的每种产品包含一行。因此，此事实表的粒度表示为 *product by account by day*。

数据模型的维

*维*表示真实世界中的一组对象或事件。您对数据模型标识的每个维都是作为维表实现的。维是限定符，由于它们回答一个问题的内容、时间以及地点这几方面，所以它们使事实表的量度具有意义。例如：考虑下列商业问题（维以斜体表示）：

- 在上一*年*，哪些*帐户*带来的收入最高？
- 各*供应商*分别给我们带来多少利润？
- 每种*产品*售出多少件？

在前一组问题中，收入、利润和售出件数是量度（*不是*维），这是因为它们每一个都表示定量或真实数据。

维元素

维可以定义多个*维元素*来获取不同级别的合计。例如：所有与销售组织的结构相关的元素都可以包含一个维。图 11-3 显示了 accounts 维定义的维元素。

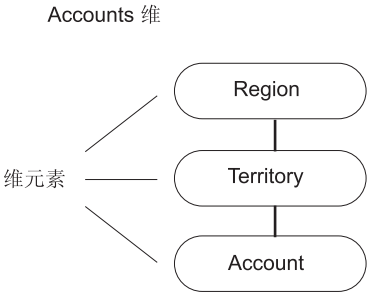


图 11-3. Accounts 维中的维元素

维由相关元素的层次结构构成。由于维具有层次结构，所以用户能够构造用来存取比先前详细信息层更高层（上卷）或更低层（下寻）的数据的查询。图 11-3 显示了维元素的分层关系：帐户上卷至地域，地域上卷至地区。根据用户要检索的数据的不同，他们可以在维的不同层进行查询。例如：用户可以对所有地区执行查询，然后下寻至地域或帐户层以获取详细信息。

维元素通常作为数字代码或短字符串存储在数据库中，以便于连接至其它表。

就象维可以定义多个维元素一样，每个维元素都可以定义多个维属性。

维属性

维属性是维表中的一列。每个属性都描述维层次结构中的一个总结层。维元素定义维表内的分层关系；属性通过用户熟悉的术语描述维元素。图 11-4 显示了 account 维的维元素和相应属性。

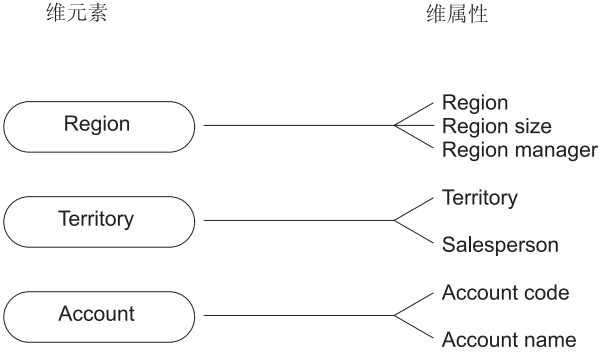


图 11-4. 与维元素对应的属性

由于维属性描述的是维中的项，所以当它们是文本时最有用。

技巧: 在设计过程中，有时会弄不清楚某个来自生产数据源的数字数据字段是量度的事实还是属性。通常，如果数字数据字段是我们每次采样时都更改的量度值，则它是事实。如果它是或多或少恒定的事物的具有离散值的描述，则它是维属性。

维表

*维表*是一个表，它存储企业的维的文本描述。对于层次结构中的每一层，维表都包含一个元素和属性（如果合适的话）。数据分析所需的最低级别的详细信息确定了层次结构中的最低层。高于这个基本层的层用来存储冗余数据。这个特殊的表减少了查询所需的连接数，并使用户可以更容易地在较高层执行查询，接着下寻至较低详细信息层。术语*下寻*表示将维表中的行头添加至查询。表 11-5 显示了基于 account 维的维表的示例。

表 11-5. 维表的示例

Acct Code	Account Name	Territory	Salesman	Region	Region Size	Region Manager
1	Jane's Mfg.	101	B. Adams	Midwest	Over 50	T. Sent
2	TBD Sales	101	B. Adams	Midwest	Over 50	T. Sent
3	Molly's Wares	101	B. Adams	Midwest	Over 50	T. Sent
4	The Golf Co.	201	T. Scott	Midwest	Over 50	T. Sent

构建维数据模型

要构建维数据模型，您需要一套可以概括完成数据库设计所需进行的决策的方法。由于此方法首先标识您所在的组织中的主要数据收集过程，所以此方法使用的是自顶向下的办法。数据库设计者的一项重要任务是从您所在的组织所使用的现有数据源入手。确定这些流程之后，根据每个业务流程来构建一个或多个事实表。下列步骤描述了用来构建数据模型的方法。

要构建维数据库:

1. 选择用来分析要对其进行建模的主题区域的业务流程。
2. 确定事实表的粒度。
3. 标识每个事实表的维和层次结构。
4. 标识事实表的量度。
5. 确定每个维表的属性。
6. 请用户验证数据模型。

尽管维数据库可以基于多个业务流程并且可以包含许多事实表，但本节描述的数据模型基于单一业务流程并且只有一个事实表。

选择业务流程

*业务流程*是您所在的组织中受一些旧系统支持的重要操作。您从这个系统收集要在维数据库中使用的数据。业务流程标识最终用户如何处理他们的数据、数据来自何处以及如何对该数据进行转换以使其有意义。信息可以有许多来源，这包括财务、销售分析、市场分析以及客户概要文件。以下列表显示了可以用来确定要在维数据库中包括哪些数据的不同业务流程:

- 销售
- 交付
- 库存
- 订单
- 发票

业务流程摘要

假定您所在的组织想要按产品系列和地区分析客户购买趋势以便可以发展更有效的营销策略。在本方案中，数据模型的主题区域是销售。

在对销售业务流程进行许多次的协商和彻底分析之后，假定您所在的组织收集到下列信息:

- 客户基础信息已更改。

  以前，销售区域是按城市划分的。现在，客户基础与两个地区相对应: “地区 1”表示加利福尼亚， “地区 2”表示所有其它州。
- 下列报告对于市场营销而言最为关键:
  - 按每个供应商的产品系列分类的每月收入、成本和净利润
  - 按产品、按地区和按月份分类的收入和售出件数
  - 每月的客户收入
  - 每个供应商的每季度收入
- 大多数销售分析都基于每月的结果，但您（在以后）可以选择按星期或记帐周期进行销售分析。
- 关系数据库中存在数据输入系统。

  要开发能够发挥作用的数据模型，您可以假定销售信息的关系数据库具有下列属性:
  - **stores_demo** 数据库提供了市场营销部门使用的许多收入数据。

- 分析员使用的产品代码作为目录号存储在 **catalog** 表中。
- 产品系列代码作为库存号存储在 **stock** 表中。产品系列名作为描述存储。
- 产品层次结构有点复杂。每个产品系列都有许多产品，每个制造商都有许多产品。

- 每种产品的所有成本数据都存储在不同采购系统上的名为 **costs.lst** 的平面文件中。
- 客户数据存储在 **stores_demo** 数据库中。

尚未将地区信息添加到数据库中。

维模型的一个重要特征是它使用最终用户熟悉的商业标号，而不是使用内部表或列名。完成业务流程后，您就应该有了创建维数据模型的量度、维和关系所需的所有信息。这个维数据模型用来实现第 12-1 页的第 12 章，『实现维数据库（XPS）』描述的 **sales_demo** 数据库。

stores_demo 演示数据库是本章开发的维数据模型的主数据源。有关用来填充 **sales_demo** 数据库的表的数据源的详细信息，请参阅第 12-4 页的『将数据从数据源映射到数据库』。

确定事实表的粒度

在收集关于主题区域的所有相关信息之后，设计过程的下一步是确定事实表的粒度。为此，您必须决定事实表中的个别低级别记录应包含什么内容。构成事实表的粒度的组件直接与数据模型的维相对应。因此，在定义事实表的粒度时，您标识数据模型的维。

粒度对数据库的大小有何影响

事实表的粒度还确定数据库需要多大的存储器空间。例如: 考虑事实表的以下可能粒度:

- Product by day by region
- Product by month by region

粒度为 *product by day by region* 的数据库要比粒度为 *product by month by region* 的数据库大得多，这是因为数据库包含每天所作的每项交易的记录而不是交易的每月合计。由于太精细的粒度可能会导致数据库极为庞大，所以您必须谨慎地确定事实表的粒度。相反，太粗略的粒度可能意味着数据不够详细，导致用户无法对数据库执行有意义的查询。

使用业务流程来确定粒度

仔细地复查从业务流程收集到的信息应该能够为您提供确定事实表的粒度所需的一切。总之，您所在的组织希望按产品线和区域分析客户购买趋势以便可以制订更有效的市场策略。

Customer by Product: 事实表的粒度始终表示对应的每个维的最低级别。当您复查从业务流程获得的信息时，事实表的客户和产品维的粒度十分明显。无法合理地对客户和产品作任何进一步的简化：它们已经表示事实表的个别记录的最低级别。（由于产品可以由多个组件组成，所以在某些情况下可以将产品进一步简化到产品组件级别。）

Customer by Product by District: 由于您所在的组织要分析的客户购买趋势包括地理组件，所以您仍需要决定地区信息的最低级别。业务流程指出销售区域过去是按城市划分的，但现在您所在的组织在客户基础的两个地区之间加以区分：“地区 1”表示加利福尼亚，“地区 2”表示所有其它州。尽管如此，在最低的级别，您所在的组织仍包括了销售区域数据，因此区域表示的是地理信息的最低级别并提供了第三个组件来进一步定义事实表的粒度。

Customer by Product by District by Day: 客户购买趋势始终是随时间而出现的，因此事实表的粒度必须包含时间成份。假定您的组织决定按星期、记帐周期、月、季度或年创建报告。在最低的级别，您可能想要选择基本粒度“天”。此粒度允许企业将星期二的销售情况与星期五的销售情况作比较，对每月第一天的销售情况作比较等。事实表的粒度现已完整。

选择“天”粒度的决定表示 **time** 维表中的每个记录都表示一天。就存储器需求而言，即使 10 年的每日数据也只有大约 3650 个记录，这是一个相对较小的维表。

标识维和层次结构

由于每个用于定义粒度的组件都与一个维相对应，所以，在确定事实表的粒度之后，可以很容易地标识数据模型的主维。图 11-5 显示了事实表的粒度与数据模型的维之间的关系。

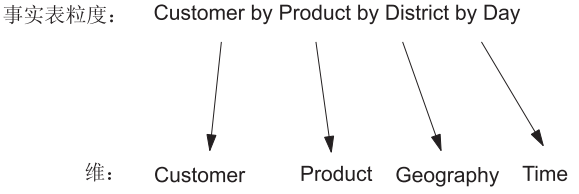


图 11-5. 与数据模型的维相对应的事实表粒度

只要确定了数据模型的维（customer、product、geography 和 time），模式图就开始成形了。

技巧：此时，可以将附加的维添加到事实表的主粒度，其中，新维在主维的每种组合下只使用单个值。如果您发现由于附加的维导致生成附加的记录而违反粒度，则必须修正事实表的粒度以适应附加的维。对于本数据模型，不需要添加附加的维。

现在，您可以绘制出每个维的维元素和层次结构。图 11-6 显示了各个维、维元素和继承层次结构之间的关系。

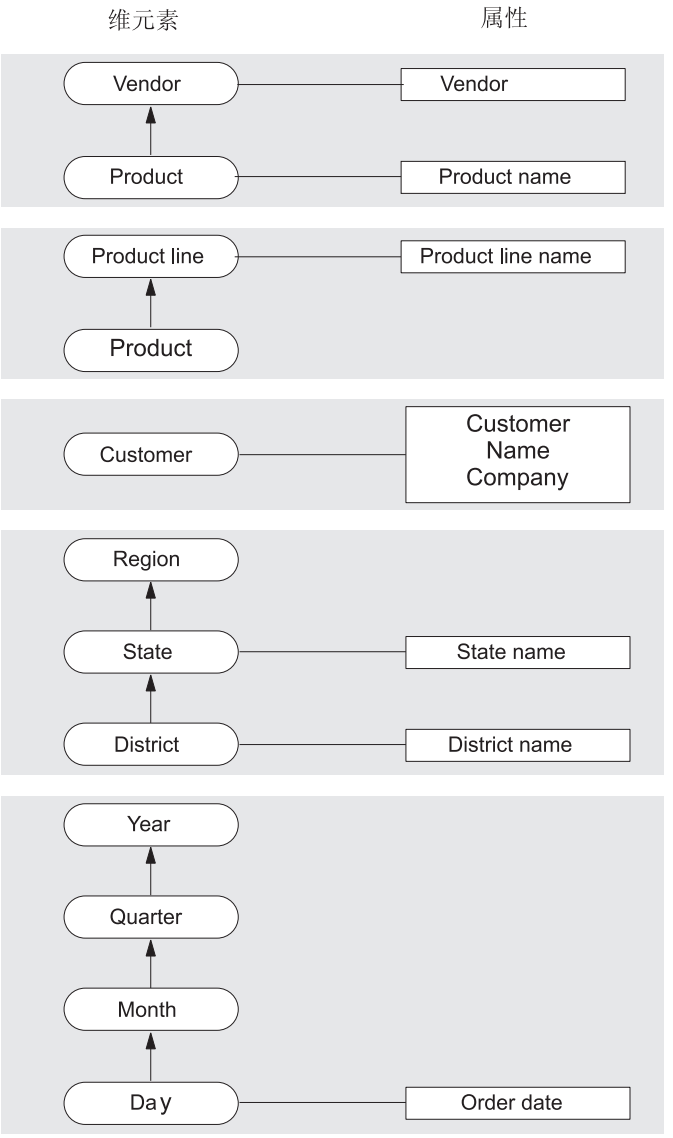


图 11-6. 维、维元素和继承层次结构之间的关系

在大多数情况下，维元素需要表达每个维的最小可能粒度，这不是因为查询需要存取个别的低级别记录，而是因为查询需要以精确的方式通过整个数据库。换而言之，尽管数据仓储环境提出的问题通常很广泛，但这些问题仍依赖于最低级别的产品详细信息。

选择事实表的量度

数据模型的量度不仅包括数据本身，还包括您根据现有数据计算而得的新值。在检查量度时，您可能会发现需要对事实表的粒度或维数作调整。

在设计数据模型时必须作出的另一重要决定是：是将计算得到的结果存储在事实表中还是在运行时得到这些值。

要回答的问题是“使用哪些量度来分析业务？”。记住，量度是指示*几个*或*多少*的定量或真实数据。通过分析销售业务流程收集到的信息将生成以下量度列表:

- 收入
- 成本
- 售出件数
- 净利润

使用这些量度来完成图 11-7 中的事实表。

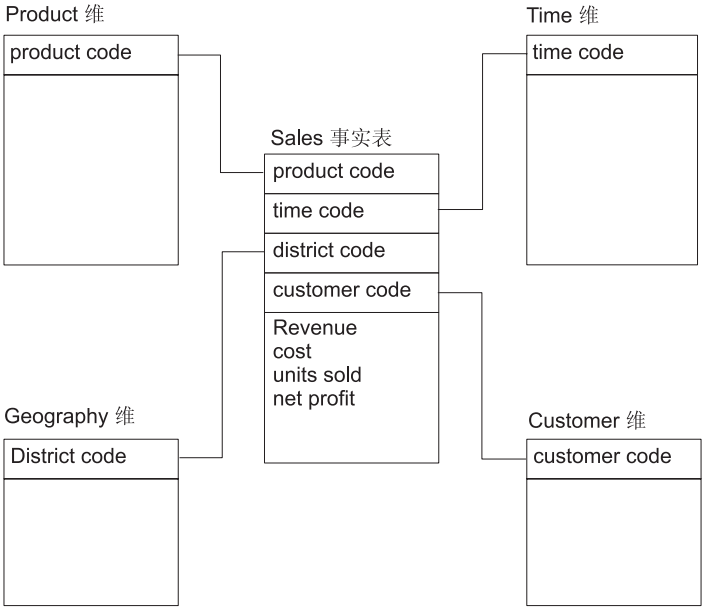


图 11-7. 引用每个维表的 Sales 事实表

使用键来将事实表与维表相连接

目前，假定第 11-15 页的图 11-7 的模式同时显示了数据库的逻辑和物理设计。数据库包含下列 5 个表:

- **Sales** 事实表
- **Product** 维表
- **Time** 维表
- **Customer** 维表
- **Geography** 维表

每个维表都包含一个主键（product、time_code、customer 和 district_code），事实表中的对应列是外键。事实表还有一个（组合）主键，它是这四个外键的组合。作为一项规则，事实表的每个外键在维表中都必须要有它的对应项。此外，维数据库中的任何带有组合键的表都必须是事实表，这表示维数据库中的每个表达多对多关系的表都是事实表。

技巧: 主键应该是短数字数据类型（INT、SMALLINT 或 SERIAL），或者是短字符串（如用作代码的短字符串）。不要使用长字符串来作为主键。

抗规范化

如果事实表的四个外键是严格管理的连续整数，则可以只为事实表的全部四个键保留 16 个字节（为 time、product、customer 和 geography 各保留 4 个字节）。如果事实表中的四个量度全都是 4 字节的整数列，则只需要保留另外 16 个字节。因此，事实表的每个记录都将只是 32 个字节。即使是包含上十亿行的事实表都将只需要大约 32 吉字节的主数据空间。

通过压缩键和数据，这样的节省存储器的事实表通常用于维数据库。维模型中的事实表在本质上是高度规范化的。由于在四个维表之间不存在相关性，所以不能将事实表中的四个键之间的极为复杂的多对多关系进一步规范化；事实上，每天都将每种产品销售给每个地区的所有客户。

事实表是维数据库中的最大的表。由于维表通常比事实表小得多，所以计算数据库的磁盘空间时可以将维表忽略。仅仅为了节省磁盘空间而将维数据库中的任何表规范化的努力是无意义的。此外，规范化的维表削弱了用户探究单个维表以设置约束和选择有用的行头的能力。

选择维表的属性

在完成事实表之后，您可以决定每个维表的维属性。为了举例说明如何选择属性，请考虑时间维。销售业务流程的数据模型定义了与时间维相对应的“天”粒度，因此 **time** 维表中的每个记录都表示一天。记住，表的每个字段都是由记录所表示的特定一天定义的。

对销售业务流程的分析还指出市场营销部门需要每月、每季度和年度报告，因此 time 维包括下列元素：day、month、quarter 和 year。对每个元素都指定一个属性，

该属性描述该元素和代码属性（以避免列值包含长字符串）。表 11-6 显示了 **time** 维表的属性以及表的每个字段的样本值。

表 11-6. Time 维的属性

time code	order date	month code	month	quarter code	quarter	year
35276	07/31/1999	7	july	3	third q	1999
35277	08/01/1999	8	aug	3	third q	1999
35278	08/02/1999	8	aug	3	third q	1999

第 11-17 页的表 11-6 显示您指定的属性名应该是令人感到熟悉的商业术语，以便于最终用户对数据库构造查询。图 11-8 显示了销售业务流程的完整数据模型以及对每个维表定义的所有属性。

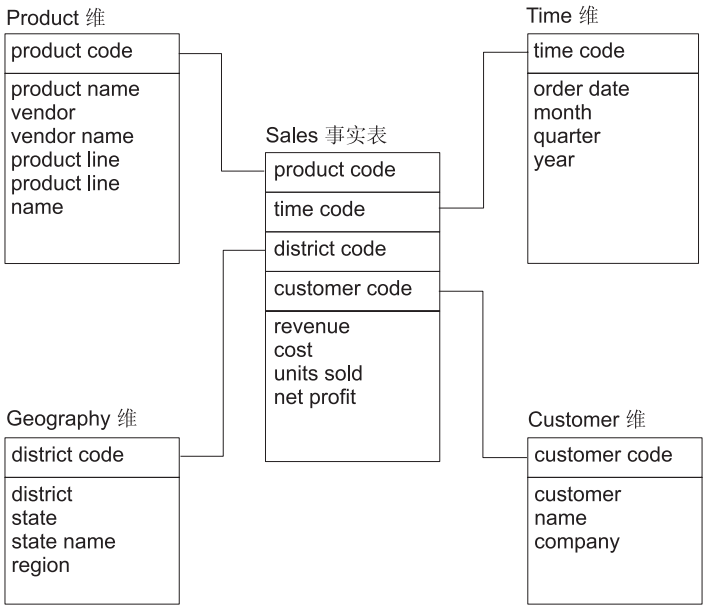


图 11-8. 销售业务流程的完整维数据模型

处理常见的维数据建模问题

前面各节描述的维模型只说明了维数据建模的最基本的概念和技术。为了满足企业的商业需求而构建的数据模型通常会牵涉到其它问题和困难，您必须解决这些问题和困难以使数据库能够获得最佳的潜在查询性能。本节描述可用来解决在构建维数据模型时发生的一些最常见问题的各种方法。

将维表中的属性数目最小化

包含客户或产品信息的维表很容易具有 50 到 100 个属性并包含数百万行。然而，具有太多属性的维表可能会导致行过宽和性能不佳。因此，您可能想将特定组的属性从维表中分出来并将它们放到称为*微型维*表的独立表中。微型维表由一小组从更大维表中分出来的属性组成。您可以选择为具有下列特征的属性创建微型维表:

- 字段很少用作查询中的约束。
- 字段频繁地一起比较。

图 11-9 显示了从 **customer** 表中分出来的人口统计信息的微型维表。

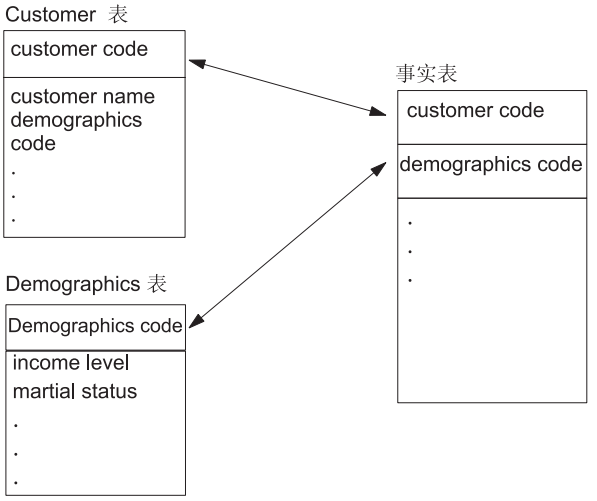


图 11-9. 人口统计信息的微型维表

在 **demographics** 表中，可以将人口统计键同时存储为事实表和 **customer** 表中的外键，这允许您将 demographics 表直接连接到事实表。还可以直接将人口统计键与 **customer** 表配合使用以浏览人口统计属性。

处理偶尔更改的维

在更新不频繁的维数据库（与 OLTP 系统相反）中，大多数维随着时间的推移是*相对*恒定的，这是因为销售区域或地区或者公司名和地址的更改不会频繁发生。然而，为了进行历史比较，当这些更改发生时必须对它们进行处理。图 11-10 显示了一个已更改的维的示例。

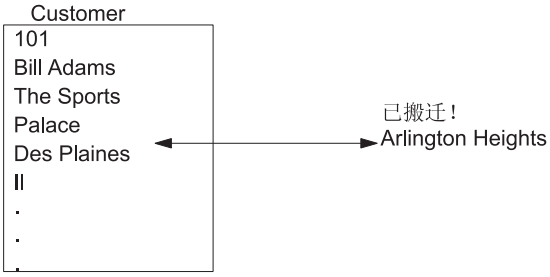


图 11-10. 发生更改的维

可以使用三种方法来处理维中发生的更改:

- 更改存储在维列中的值。

  在图 11-10 中，**customer** 维表中 `Bill Adams` 的记录已更新为显示新地址 `Arlington Heights`。这位客户的所有先前销售历史记录现在都与 Arlington Heights 区域而不是 Des Plaines 相关联。

- 使用新值和通用键来创建第二个维记录。

  这种方法能够有效地将历史记录分区。**customer** 维表现在将包含 Bill Adams 的两个记录。键为 101 的旧记录保持不变，并且事实表中的记录仍与其相关联。还为 Bill Adams 将一个新记录添加至 **customer** 维表，并且新键可能由旧键加上一些版本位组成（例如 101.01）。为 Bill Adams 添加至事实表的所有后续记录都与这个新键相关联。

- 在 **customer** 维表中为受影响的属性添加新字段并将旧属性重命名。

  除非需要依据新值跟踪旧历史记录（反之亦然），否则很少使用此方法。**customer** 维表获得名为 **current address** 的新属性，将旧属性重命名为 **original address**。包含关于 Bill Adams 的信息的记录同时包含原始地址值和当前地址值。

使用雪花模式

雪花模式是星型模式的变种，在此模式中，将非常大的维表规范化成多个表。在使用事实表的聚集时，如果想要避免连接至大型维表，则可以将具有层次结构的

维分解成雪花结构。例如: 如果要将商标信息从 **product** 维表中分出来，则可创建商标雪花，它对每个商标都包含一行，并且包含的行数要比 **product** 维表少得多。图 11-11 显示了商标和产品系列元素的雪花结构以及 **brand_agg** 聚集表。

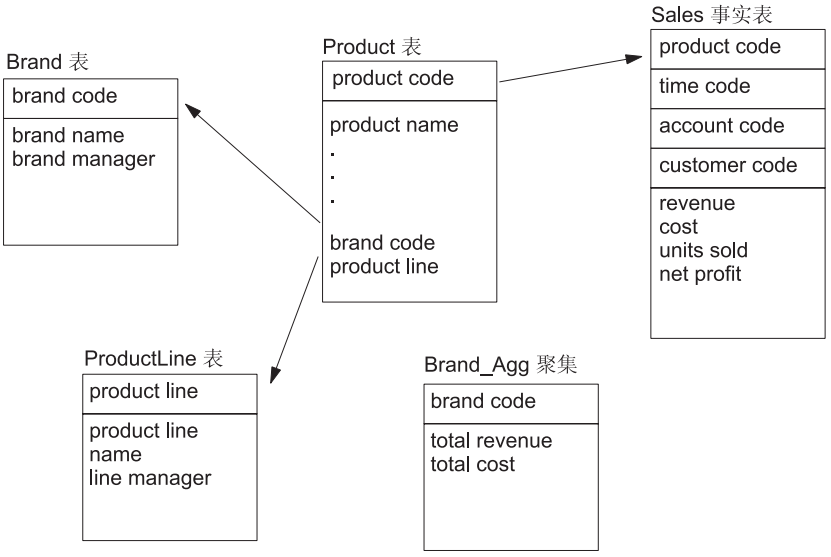


图 11-11. 雪花模式的示例

如果创建 **brand_agg** 聚集，它由商标代码和每个商标的总收入组成，则可使用雪花模式来避免连接至大得多的 **sales** 表，如对 **brand** 表和 **brand_agg** 表执行的以下查询所示:

```
SELECT brand.brand_name, brand_agg.total_revenue
FROM brand, brand_agg
   WHERE brand.brand_code = brand_agg.brand_code
   AND brand.brand_name = 'Anza'
```

如果没有*雪花*维表，就需要对整个 **product** 表（这可能是非常大的包含所有商标和产品系列属性的维表）使用 SELECT UNIQUE 或 SELECT DISTINCT 语句来消除重复行。

尽管当维表相对较小时雪花模式不是必需的，但使用包含数百万行的客户或产品维表的零售或邮购销售企业可使用雪花模式来显著改进性能。

如以下查询所示，如果聚集表不可用，则对已通过雪花模式规范化的维元素的任何连接现在都必须是三向连接。三向连接减少了维数据库的一些性能优点。

```
SELECT brand.brand_name, SUM(sales.revenue)
FROM product, brand, sales
   WHERE product.brand_code = brand.brand_code
   AND brand.brand_name = 'Alltemp'
GROUP BY brand_name
```

第 12 章 实现维数据库（XPS）

在本章中

本章阐述如何使用 SQL 来实现第 11-1 页的第 11 章，『构建维数据模型』描述的维数据模型。记住，此数据库仅用作数据仓储环境的说明性示例。为了实现本示例的目的，已将其转换为 SQL 语句。

本章描述 Extended Parallel Server 提供的 **sales_demo** 数据库。本章还描述 Extended Parallel Server 提供的特殊表类型和索引，适合于数据仓储和其它非常大型数据库应用程序的需要。

实现 sales_demo 维数据库

本节阐述可用来根据第 11 章中的数据模型创建维数据库的 SQL 语句。可使用交互式 SQL 来编写个别用于创建数据库的语句，也可以运行用于自动执行实现数据库所需的所有语句的脚本。CREATE DATABASE 和 CREATE TABLE 语句将数据模型作为数据库中的表来创建。创建数据库之后，可使用 LOAD 和 INSERT 语句来填充表。

使用 CREATE DATABASE

在可以创建数据库包含的任何表或其它对象之前，必须先创建该数据库。

当 Informix 数据库服务器创建数据库时，它设置用于显示数据库的存在情况及其日志记录方式的记录。数据库服务器直接管理磁盘空间，因此这些记录对操作系统命令不可视。

使用 Extended Parallel Server 创建数据库时，始终打开记录。然而，可以在数据库中创建非日志记录表。有关更多信息，请参阅第 7-3 页的『使用分布式查询配制数据库服务器』。

以下语句显示用来创建名为 **sales_demo** 的数据库的语法:

```
CREATE DATABASE sales_demo
```

对维和事实表使用 CREATE TABLE

本节包括用来创建 **sales_demo** 维数据库的表的 CREATE TABLE 语句。

当然，引用完整性是维数据库的一项重要需求。但是，**sales_demo** 数据库的以下模式没有定义在事实表与其维表之间存在的主键和外键关系。由于数据库服务器不强制执行约束检查时可以显著改进数据装入性能，所以此模式没有定义这些主键和外键关系。假定数据仓储环境通常要求在指定时间内装入数十或数百吉字节的数据，则当您决定如何在数据仓储环境中实现数据库时，数据装入性能应该是一个因素。假定如果 **sales_demo** 数据库是作为实际数据集市实现的，则使用一些数据抽取工具（而不是数据库服务器）来在事实表与维表之间强制执行引用完整性。

技巧: 在创建并装入表之后，可使用 ALTER TABLE 语句来向表添加主键和外键约束以强制执行引用完整性。只有快速装入方式才要求使用此方法。如果约束和索引是必需的，并且在进行装入之前删除它们的成本很高，则高级装入方式是最佳选项。

下列语句创建 **time**、**geography**、**product** 和 **customer** 表。这些表是 **sales** 事实表的维。一个 SERIAL 字段用作 **geography** 表的 **district_code** 列的主键。

```
CREATE TABLE time
(
time_code     INT,
order_date    DATE,
month_code    SMALLINT,
month_name    CHAR(10),
quarter_code  SMALLINT,
quarter_name  CHAR(10),
year INTEGER
);
```

```
CREATE TABLE geography
(
district_code  SERIAL,
district_name  CHAR(15),
state_code     CHAR(2),
state_name     CHAR(18),
region         SMALLINT
);

CREATE TABLE product (
product_code   INTEGER,
product_name   CHAR(31),
vendor_code    CHAR(3),
vendor_name    CHAR(15),
product_line_code  SMALLINT,
product_line_name  CHAR(15)
);

CREATE TABLE customer (
customer_code  INTEGER,
customer_name  CHAR(31),
company_name   CHAR(20)
);
```

sales 事实表带有指向每个维表的指针。例如: **customer_code** 引用 customer 表，**district_code** 引用 geography 表，等等。**sales** 表还包含售出件数、收入、成本和净利润的量度。

```
CREATE TABLE sales
(
customer_code  INTEGER,
district_code  SMALLINT,
time_code      INTEGER,
product_code   INTEGER,
units_sold     SMALLINT,
revenue        MONEY(8,2),
cost           MONEY(8,2),
net_profit     MONEY(8,2)
);
```

技巧: 最有用的量度（事实）是数字的，并且具有可加性。由于数据仓储环境中的数据库十分巨大，所以，对事实表执行的每个查询事实上都可能需要数千或数百万个记录来构造答案集。压缩这些记录的唯一有用方法是将它们聚集起来。在 **sales** 表中，每个量度列都是在数字数据类型上定义的，因此可以很容易地从 **units_sold**、**revenue**、**cost** 和 **net_profit** 列构建答案集。

为了便于您使用它们，名为 **createdw.sql** 的文件包含前面所有 CREATE TABLE 语句。

将数据从数据源映射到数据库

stores_demo 演示数据库是 **stores_demo** 数据库的主数据源。

第 12-4 页的表 12-1 显示了数据仓储商业术语与数据源之间的关系。它还显示了 **sales_demo** 数据库的每个列和表的数据源。

表 12-1. 数据仓储商业术语与数据源之间的关系

商业术语	数据源	表.列名
Sales 事实表:		
product code		sales.product_code
customer code		sales.customer_code
district code		sales.district_code
time code		sales.time_code
revenue	stores_demo:items.total_price	sales.revenue
units sold	stores_demo:items.quantity	sales.units_sold
cost	costs.lst（每一件）	sales.cost
net profit	计算值: revenue - cost	sales.net_profit
Product 维表:		
product	stores_demo:catalog.catalog_num	product.product_code
product name	stores_demo:stock.manu_code 和 stores_demo:stock.description	product.product_name
product line	stores_demo:orders.stock_num	product.product_line_code
product line name	stores_demo:stock.description	product.product_line_name
vendor	stores_demo:orders.manu_code	product.vendor_code
vendor name	stores_demo:manufact.manu_name	product.vendor_name
Customer 维表:		
customer	stores_demo:orders.customer_num	customer.customer_code
customer name	stores_demo:customer.fname 加 stores_demo:customer.lname	customer.customer_name
company	stores_demo:customer.company	customer.company_name
Geography 维表:		
district code	生成的	geography.district_code
district	stores_demo:customer.city	geography.district_name
state	stores_demo:customer.state	geography.state_code
state name	stores_demo.state.sname	geography.state_name
region	派生的: If state = ″CA″ THEN region = 1, ELSE region = 2	geography.region
Time 维表:		
time code	生成的	time.time_code
order date	stores_demo:orders.order_date	time.order_date
month	从生成的订单日期派生	time.month_name time.month.code

表 12-1. 数据仓储商业术语与数据源之间的关系 (续)

商业术语	数据源	表.列名
quarter	从生成的订单日期派生	time.quarter_name time.quarter_code
year	从订单日期派生	time.year

若干个带有 **.unl** 后缀的文件包含装入到 **sales_demo** 数据库中的数据。包含用于创建和装入数据库的 SQL 语句的文件具有 **.sql** 后缀。

仅适用于 UNIX

当数据库服务器在 UNIX 上运行时，可以从 **$INFORMIXDIR/demo/dbaccess** 目录中访问 ***.sql** 和 ***.unl** 文件。

仅适用于 UNIX 结束

仅适用于 Windows

当数据库服务器在 Windows 上运行时，可以从 **%INFORMIXDIR%\demo\dbaccess** 目录中访问 ***.sql** 和 ***.unl** 文件。

仅适用于 Windows 结束

将数据装入到维数据库中

实现维数据库时的一个重要步骤是开发和归档装入策略。本节阐述可用来填充 **sales_demo** 数据库的表的 LOAD 和 INSERT 语句。

技巧: 在实际数据仓储环境中，通常不使用 LOAD 或 INSERT 语句来将大量数据装入 Informix 数据库或从 Informix 数据库装入那些数据。

Informix 数据库服务器提供了不同的功能来实现数据的高性能装入和卸装。

使用 Extended Parallel Server 创建数据库时，可使用*外部表*来执行高性能装入和卸装。

有关高性能装入的信息，请参阅 *《IBM Informix: 管理员指南》* 或 *《IBM Informix: High-Performance Loader 用户指南》*。

以下语句首先将数据装入 **time** 表，以便您可以使用它来确定装入到 **sales** 表中的每一行的时间代码:

```
LOAD FROM 'time.unl' INSERT INTO time
```

以下语句装入 **geography** 表。一旦装入 **geography** 表之后，可使用区域代码数据来装入 **sales** 表。

```
INSERT INTO geography(district_name, state_code, state_name)
SELECT DISTINCT c.city, s.code, s.sname
   FROM stores_demo:customer c, stores_demo:state s
      WHERE c.state = s.code
```

下列语句将地区代码添加至 **geography** 表:

```
UPDATE geography
   SET region = 1
   WHERE state_code = 'CA'

UPDATE geography
   SET region = 2
   WHERE state_code <> 'CA'
```

以下语句装入 **customer** 表:

```
INSERT INTO customer (customer_code, customer_name, company_name)
SELECT c.customer_num, trim(c.fname) ||' '|| c.lname, c.company
FROM stores_demo:customer c
```

以下语句装入 **product** 表:

```
INSERT INTO product (product_code, product_name, vendor_code,
   vendor_name,product_line_code, product_line_name)
SELECT a.catalog_num,
   trim(m.manu_name)||' '||s.description,
   m.manu_code, m.manu_name,
   s.stock_num, s.description
FROM stores_demo:catalog a, stores_demo:manufact m,
   stores_demo:stock s
   WHERE a.stock_num = s.stock_num
      AND a.manu_code = s.manu_code
      AND s.manu_code = m.manu_code;
```

以下语句装入 **sales** 事实表，对每种产品、每个客户、每天和每个区域有一行。使用 **cost** 表中的成本来计算总成本（cost * quantity）。

```
INSERT INTO sales (customer_code, district_code, time_code,
   product_code, units_sold, cost, revenue, net_profit)
SELECT
   c.customer_num, g.district_code, t.time_code,
   p.product_code, SUM(i.quantity),
   SUM(i.quantity * x.cost), SUM(i.total_price),
   SUM(i.total_price) - SUM(i.quantity * x.cost)
FROM stores_demo:customer c, geography g, time t,
   product p,stores_demo:items i,
   stores_demo:orders o, cost x
WHERE c.customer_num = o.customer_num
   AND o.order_num = i.order_num
```

```
   AND p.product_line_code = i.stock_num
   AND p.vendor_code = i.manu_code
   AND t.order_date = o.order_date
   AND p.product_code = x.product_code
   AND c.city = g.district_name
GROUP BY 1,2,3,4;
```

创建 sales_demo 数据库

sales_demo 维数据库使用 **stores_demo** 数据库中的数据，因此必须创建这两个数据库才能实现 **sales_demo** 数据库。

有关如何使用 **dbaccessdemo** 脚本来实现 **sales_demo** 数据库的信息，请参阅《*IBM Informix: DB–Access 用户指南*》。

测试维数据库

可以创建 SQL 查询来检索业务流程摘要（请参阅第 11-10 页的『业务流程摘要』）中列示的标准报告所必需的数据。使用下列特别查询来测试是否已正确地实现了维数据库。

以下语句按每个供应商的产品系列分类来返回每月收入、成本和净利润:

```
SELECT vendor_name, product_line_name, month_name,
   SUM(revenue) total_revenue, SUM(cost) total_cost,
   SUM(net_profit) total_profit
FROM product, time, sales
WHERE product.product_code = sales.product_code
   AND time.time_code = sales.time_code
GROUP BY vendor_name, product_line_name, month_name
ORDER BY vendor_name, product_line_name;
```

以下语句按产品、按地区并按月份分类来返回收入和售出件数:

```
SELECT product_name, region, month_name,
   SUM(revenue), SUM(units_sold)
FROM product, geography, time, sales
WHERE product.product_code = sales.product_code
   AND geography.district_code = sales.district_code
   AND time.time_code = sales.time_code
GROUP BY product_name, region, month_name
ORDER BY product_name, region;
```

以下语句返回每月客户收入:

```
SELECT customer_name, company_name, month_name,
   SUM(revenue)
FROM customer, time, sales
WHERE customer.customer_code = sales.customer_code
   AND time.time_code = sales.time_code
GROUP BY customer_name, company_name, month_name
ORDER BY customer_name;
```

以下语句返回每个供应商的每季度收入:

```
SELECT vendor_name, year, quarter_name, SUM(revenue)
FROM product, time, sales
WHERE product.product_code = sales.product_code
   AND time.time_code = sales.time_code
GROUP BY vendor_name, year, quarter_name
ORDER BY vendor_name, year
```

Extended Parallel Server 中的日志记录表和非日志记录表

本节描述在数据仓储环境中可能特别有用的不同表类型。缺省情况下，Extended Parallel Server 采用 Dynamic Server 记录表的方式来对表进行记录。但是，数据仓储环境和其它涉及大量数据（以及很少或没有插入、更新或删除操作）的应用程序通常需要在同一数据库中组合使用日志记录表和非日志记录表。在许多情况下，由于临时表在数据库会话结束后不会继续存在，所以它们并不足够。为了同时满足日志记录表和非日志记录表的需要，Extended Parallel Server 支持下列类型的永久表和临时表:

- 原始永久表（非日志记录）
- 静态永久表（非日志记录）
- 操作永久表（日志记录）
- 标准永久表（日志记录）
- 暂存临时表（非日志记录）
- Temp 临时表（日志记录）

如果发出 CREATE TABLE 语句并且没有指定表类型，则将创建标准永久表。要在表类型之间进行更改，请使用 ALTER TABLE 语句。有关语法的信息，参阅《*IBM Informix: SQL 指南: 语法*》。

要点: 协同服务器（coserver）只能将它自己的数据库空间用作临时空间并进行访问。虽然临时表可以象永久表一样显式地跨数据库空间分段，但是协同服务器（coserver）只将数据插入到它管理的分段中。

选择表类型

数据仓储环境中的个别表通常具有不同的需求。为了帮助确定要用于表的适当表类型，回答下列问题:

- 表需要索引吗？
- 表需要定义哪些约束？
- 表的刷新和更新周期是什么？
- 表是只读表吗？

- 需要对表进行记录吗？

表 12-2 列示了 Extended Parallel Server 支持的 6 种类型的表的属性并显示了可以如何使用外部表来装入这些类型的表。请使用此信息来选择表类型以与表的特定需求相匹配。

表 12-2. Extended Parallel Server 的表类型的特征

类型	永久	记录	索引	使用少量追加功能	可用回滚	可恢复	可从归档复原	外部表装入方式
SCRATCH	否	否	否	是	否	否	否	快速或高级装入方式
TEMP	否	是	是	是	是	否	否	快速或高级装入方式
RAW	是	否	否	是	否	否	否	快速或高级装入方式
STATIC	是	否	是	否	否	否	否	无
OPERATIONAL	是	是	是	是	是	是	否	快速或高级装入方式
STANDARD	是	是	是	否	是	是	是	高级装入方式

暂存和 Temp 临时表

暂存表是非日志记录临时表，它不支持索引、约束或回滚。

尽管 Temp 表也支持批量操作（如少量追加功能），但它们是记录临时表。（快速方式装入使用绕过缓冲区高速缓存的*少量追加功能*。少量追加功能消除了与缓冲区管理相关联的开销，但不对数据进行记录。）Temp 表支持索引、约束和回滚。

技巧: 与普通插入相似，SELECT...INTO TEMP 和 SELECT...INTO SCRATCH 语句在协同服务器（coserver）之间是并行的。当跨节点的分段临时表是使用 SELECT...INTO TEMP 和 SELECT...INTO SCRATCH 显式创建的时，Extended Parallel Server 自动支持那些表。

Extended Parallel Server 根据以下条件创建显式临时表:

- 如果用来填充 Temp 或暂存表的查询没有生成行，则数据库服务器创建空的未分段表。
- 如果查询生成的行不超过 8 千字节，则临时表只驻留在一个数据库空间中。
- 如果行超过 8 千字节，则 Extended Parallel Server 创建多个分段并使用循环法分段存储方案来填充它们。

原始永久表

原始表是使用少量追加功能的非日志记录永久表。快速方式装入使用绕过缓冲区高速缓存的*少量追加功能*。可使用快速方式来装入原始表。有关快速方式装入的信息，请参阅 *《IBM Informix: 管理员参考大全》*。

原始表支持更新、插入和删除，但不对它们进行记录。原始表不支持索引或引用约束、回滚、可恢复性或从归档复原。

使用原始表来进行初始数据装入和净化。完成这些步骤后，将表改变为具有较高级别。例如: 如果装入原始表时出错或失败，则结果数据就是失败时位于磁盘上的那些数据。

在数据仓储环境中，当下列两个条件都成立时，可以选择将事实表创建成原始表:

- 事实表不需要指定约束和索引，另一些机制强制执行约束和索引。
- 创建和装入事实表并不是成本很高的作业。事实表对于决策支持而言可能有用但并不关键，即使丢失数据也可以很容易地重新装入表。

静态永久表

静态表是非日志记录的只读永久表，它们不支持插入、更新和删除操作。当预期不会对表执行插入、更新或删除操作时，可选择将表创建成静态表。对于静态表，可以创建和删除非群集索引和引用约束，这是因为它们不影响数据。

静态表不支持回滚、可恢复性或从归档复原。它们的优点是数据库服务器可使用快速扫描并避免执行查询时由于静态表是只读的而导致锁定。

技巧: 由于静态表是唯一支持 GK 索引的表类型，所以当您想要创建使用 GK 索引的表时，静态表非常重要。

操作永久表

操作表是记录永久表，它使用少量追加功能并且不执行逐个记录的记录。它们允许进行快速的更新操作。

对于操作表，可以回滚操作或在失败后进行恢复，但由于不对装入的批量插入记录进行记录，所以不能可靠地从日志的归档复原它们。在从另一个源派生数据（因此可复原性不是一个问题）但不需要回滚和可恢复性的情况下，使用操作表。

由于数据是被定期刷新的，所以可以将事实表创建成操作表。操作表支持快速装入方式（不存在索引和约束），数据是可恢复的。

标准永久表

Extended Parallel Server 中的标准表与使用 Dynamic Server 创建的日志记录数据库中的表相同。逐个记录地对所有操作进行记录，因此可以从归档复原标准表。标准表支持可恢复性和回滚。

如果表的更新和刷新周期不频繁，则可选择创建标准表类型，这是因为不需要在刷新周期内删除约束或索引。构建索引不仅耗时而且成本高昂，但却是必需的。

技巧: 标准表不使用轻附加功能，因此使用外部表来执行装入时不能使用快速装入方式。

在表类型之间切换

使用 ALTER TABLE 命令来在永久表的类型之间进行切换。如果表不符合新类型的限制，则改变操作失败，并且生成说明性的错误消息。下列限制适用于表改变操作:

- 在将表改变为具有 RAW 类型前，必须删除索引和引用约束。
- 在将表改变为具有 STANDARD 类型之前，必须执行第 0 级归档，以便表符合完全可恢复性限制。
- 不能改变 temp 或暂存临时表。

数据仓储环境的索引

除了传统（B 树）索引以外，Extended Parallel Server 提供了下列可用来改进数据仓储环境中的特别查询性能的索引:

- 位图索引

  位图索引是 B 树索引的特殊化变体。可使用位图索引来对可以包含少数几个值中的其中一个值（如婚姻状态或性别）的列建立索引。对于每个高度重复的值，位图索引存储该列可包含的每个值的压缩位图。使用位图索引，由于包含同一个键的行之间的距离缩短了，所以提高了存储效率。

  当下列条件都成立时，可以使用位图索引:

  – 索引中的键值包含许多重复项。
  – 表中的多个列具有优化器可用来改进表扫描性能的索引。

- 通用键（GK）索引

  GK 索引允许将表达式的结果、对数据集的选择或来自连接表的数据集的交集作为键存储在 B 树或位图索引中，这在对一个或多个大型表执行的特定查询中可能很有用。

要创建 GK 索引，涉及的所有表都应该是静态表。

为了改进建立索引的效率，Extended Parallel Server 还支持下列功能:

- 自动组合索引，以便在同一个表存取中使用。

  可将多列索引与单列索引组合在一起。

- 通过称为“跳过扫描”的存取方法来读取表。

  在表中扫描行时，数据库服务器只读取索引指示的行，并按照那些行在数据库中的出现顺序读取它们。*跳过扫描*存取方法保证不将任何一页读取两次。按顺序而不是随机地读取页，这降低了 I/O 资源需求。由于不必对索引列进行过滤，所以跳过扫描还降低了 CPU 需求。

- 使用散列半连接来减少处理特定多表连接的工作。

  对于代表对星型模式（在星型模式中，一个大表（事实表）与许多个小表（维表）连接在一起）的查询的连接而言，散列半连接特别有用。散列半连接可以有效地在连接开始之前尽可能减少行集。

分析预期要对数据库运行的查询的类型可以帮助您决定要创建的索引的类型。有关可用来改进查询性能的索引以及索引建立方法的信息，请参阅 *《IBM Informix: 性能指南》*。

在数据仓储环境中使用 GK 索引

当预期要对表频繁使用特定类型的查询时，可以创建 GK 索引。下列示例说明可以如何为一个或多个大表上的查询创建和使用 GK 索引。这些示例基于 **sales_demo** 数据库的表。

对选择定义 GK 索引

假定对 **sales** 事实表执行的典型查询返回 state = "CA" 的值。要改进此类查询的性能，可以创建允许将 SELECT 语句的结果作为键存储在索引中的 GK 索引。以下语句创建 **state_idx** 索引，此索引可以改进通过地理数据限制搜索的查询的性能:

```
CREATE GK INDEX state_idx ON geography
   (SELECT district_code FROM geography
      WHERE state_code = "CA");
```

数据库服务器可对以下类型的查询使用 **state_idx** 索引，此查询返回 state = "CA" 的按产品、按地区并按月份分类的收入和售出件数。数据库服务器使用 **state_idx** 索引来从 **geography** 表中检索 state = "CA" 的行以改进整体查询性能。

```
SELECT product_name, region, month_name, SUM(revenue),
SUM(units_sold)
FROM product, geography, time, sales
WHERE product.product_code = sales.product_code
   AND geography.district_code = sales.district_code
   AND state_code = "CA" AND time.time_code = sales.time_code
GROUP BY product_name, region, month_name
ORDER BY product_name, region;
```

对表达式定义 GK 索引

可以创建允许将表达式的结果作为键存储在索引中的 GK 索引。以下语句创建 **cost_idx** 索引，此索引可改进对 **sales** 表执行的包括售出产品的成本的查询的性能:

```
CREATE GK INDEX cost_idx ON sales
   (SELECT units_sold * cost FROM sales);
```

数据库服务器可以对以下类型的查询使用 **cost_idx** 索引，此查询返回在产品上花费了超过 $10,000.00 的客户的名称:

```
SELECT customer_name
FROM sales, customer
WHERE sales.customer_code = customer.customer_code
   AND units_sold * cost > 10000.00;
```

对连接表定义 GK 索引

可以创建允许将来自连接表的数据集的相交结果作为键存储在索引中的 GK 索引。假定要为 **sales** 表中的每个条目对 **time** 维表中的年数据创建 GK 索引。以下语句创建 **time_idx** 索引:

```
CREATE GK INDEX time_idx ON sales
(SELECT year FROM sales, time
   WHERE sales.time_code = time.time_code);
```

要点: 要创建前述 GK 索引，**sales** 表的 **time_code** 列必须是引用 **time** 表中的 **time_code** 列（主键）的外键。

数据库服务器可以对以下类型的查询使用 **time_idx** 索引，此查询返回在 1996 年后购买了产品的客户的名称:

```
SELECT customer_name
FROM   sales, customer, time
WHERE  sales.time_code = time.time_code AND year > 1996
    AND sale.customer_code = customer.customer_code;
```

第 5 部分 附录

附录. 辅助选项

本手册 HTML 版本中提供了点分十进制语法格式的语法图，该格式是一种可访问格式，它仅在您正使用屏幕阅读器时才可用。

点分十进制语法图

当为点分十进制格式时，每个语法元素写在单独一行上。如果两个或更多语法元素总是一起存在（或总是一起不存在），则这些元素可出现在同一行上，因为可以将它们当做单个组合语法元素。

每行以点分十进制数字开始；例如: 3 或 3.1 或 3.1.1。要正确听取这些数字，请确保将屏幕阅读器设置为读取标点符号。所有具有相同点分十进制数字的语法元素（例如: 所有具有数字 3.1 的语法元素）为互斥选项。如果您听取行 3.1 USERID 和 3.1 SYSTEMID，则语法可以包含 USERID 或 SYSTEMID 中的任何一个，但是不能够同时包含它们。

点分十进制编号级别表示嵌套的级别。例如: 如果带有点分十进制数字 3 的语法元素后跟一系列带有点分十进制数字 3.1 的语法元素，则所有编号 3.1 的语法元素都为编号 3 的语法元素的下级。

在点分十进制数字的旁边使用特定词语和符号，以添加关于语法元素的信息。有时，这些字和符号可能会出现在元素本身的开头处。为了易于识别，如果字或符号是语法元素的一部分，则在字或符号前放置反斜杠（\）符号。可在点分十进制数字旁使用 * 符号以表示语法元素重复。例如: 带有点分十进制数字 3 的语法元素 *FILE读做 3 * FILE。格式 3* FILE 表示语法元素 FILE 重复。格式 3* * FILE 表示语法元素 * FILE重复。

用来隔开一串语法元素的字符（如，逗号），在语法中显示在所隔开的项之前，紧挨着这些项。这些字符可作为单独项显示在同一行上，或作为相关项并带有相同点分十进制数字显示在不同行上。行还显示提供关于语法元素的信息的另一个符号。例如: 行 5.1*、5.1 LASTRUN 和 5.1 DELETE 表示如果您使用多个 LASTRUN 和 DELETE语法元素，则这些元素必须用逗号隔开。如果未提供分隔符，则假定您使用空格来隔开每个语法元素。

如果在语法元素之前放置 % 符号，则表示这是在其它地方定义的参考。跟随 % 符号的字符串是语法分段的名称而不是字面值。例如: 行 2.1 %OP1 表示您应该参考不同的语法分段 OP1。

以下词语和符号用于点分十进制数字旁:

? 指定可选的语法元素。后跟 ?符号的点分十进制数字表示所有带有相应点分十进制数字的语法元素和所有下级语法元素为可选。如果仅有一个带有点分十进制数字的语法元素，则 ? 符号将和该语法元素显示在同一行上（例如: 5? NOTIFY）。如果有多个带有点分十进制数字的语法元素，则 ? 符号单独显示在一行上，后跟可选的语法元素。例如: 如果您听取行 5 ?、5 NOTIFY 和 5 UPDATE，则您将知道语法元素 NOTIFY 和 UPDATE 为可选; 也就是说，您可以选择它们中的一个或一个都不选。 ? 符号等同于轨道图中的旁路行。

! 指定缺省语法元素。后跟 !符号的点分十进制数字和语法元素表示语法元素是共享相同点分十进制数字的所有语法元素的缺省选项。共享相同点分十进制数字的各语法元素中只有一个是可以指定 ! 符号的。例如: 如果您听取行 2? FILE、2.1! (KEEP) 和 2.1 (DELETE)，则您将知道 (KEEP) 是 FILE 关键字的缺省选项。在此示例中，如果您包含 FILE 关键字而不指定一个选项，则将应用缺省选项 KEEP。缺省选项还适用于下一个更高的点分十进制数字。在此示例中，如果省略 FILE 关键字，则将使用缺省 FILE(KEEP)。但是，如果您听取行 2? FILE、2.1、2.1.1! (KEEP) 和 2.1.1 (DELETE)，则缺省选项 KEEP 将只适用于下一个更高的点分十进制数字和 2.1（没有关联关键字），并且不适用于 2? FILE。如果省略关键字 FILE，则将不使用任何选项。

***** 指定可不重复或重复多次的语法元素。后跟 * 符号的点分十进制数字表示此语法元素可不使用或使用多次; 也就是说，它是可选的并且可以重复。例如: 如果您听取行 5.1* data-area，则您将知道您可以包含多个数据区或可以不包含任何数据区。如果您听取行 3*、3 HOST 和 3 STATE，则您将知道您可以同时包含 HOST 和 STATE，或一个都不包含。

注意:

1. 如果点分十进制数字旁有星号（*），并且仅有一个项带有该点分十进制数字，则您可以多次重复该同一个项。
2. 如果点分十进制数字旁有星号，并且多个项带有该点分十进制数字，则您可以从列表中使用多个项，但是每一个项您都无法使用多于一次。在前面的示例中，您可以写 HOST STATE，但是您无法写 HOST HOST。
3. * 符号等同于轨道语法图中的回环行。

+ 指定必须一次或多次包含的语法元素。后跟 + 符号的点分十进制数字表示必须一次或多次包含此语法元素。例如: 如果您听取行 6.1+ data-area，则您必须包含至少一个数据区。如果您听取行 2+、2 HOST 和 2 STATE，则您将知道您必须包含 HOST 和 STATE 中的一个或全部。对于 * 号，则

只有一个特定项是唯一带有该点分十进制数字的项时才能重复该项。+ 号和 * 号一样等同于轨道语法图中的回环行。

声明

IBM 可能在其他国家或地区不提供本文档中讨论的产品、服务或功能特性。有关您当前所在区域的产品和服务的信息，请向您当地的 IBM 代表咨询。任何对 IBM 产品、程序或服务的引用并非意在明示或暗示只能使用 IBM 的产品、程序或服务。只要不侵犯 IBM 的知识产权，任何同等功能的产品、程序或服务，都可以代替 IBM 产品、程序或服务。但是，评估和验证任何非 IBM 产品、程序或服务，则由用户自行负责。

IBM 公司可能已拥有或正在申请与本文档所述内容有关的各项专利。提供本文档并未授予用户使用这些专利的任何许可。您可以用书面方式将许可查询寄往:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

有关双字节（DBCS）信息的许可查询，请与您的国家或地区的 IBM 知识产权部门联系，或用书面方式将查询寄往:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

本条款不适用英国或任何这样的条款与当地法律不一致的任何国家或地区:
INTERNATIONAL BUSINESS MACHINES CORPORATION "按现状" 提供本出版物，不附有任何形式的（无论是明示的还是暗含的）保证，包括但不限于暗含的有关非侵权、适销和适用于某种特定用途的保证。某些国家或地区在某些交易中不允许免除明示或暗含的保证。因此本条款可能不适用于您。

本信息中可能包含技术方面不够准确的地方或印刷错误。此处的信息将定期更改；这些更改将编入本出版物的新版本中。IBM 可以随时对本出版物中描述的产品和／或程序进行改进和／或更改，而不另行通知。

本信息中对非 IBM Web 站点的任何引用都只是为了方便起见才提供的，不以任何方式充当对那些 Web 站点的保证。那些 Web 站点中的资料不是 IBM 产品资料的一部分，使用那些 Web 站点带来的风险将由您自行承担。

IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

本程序的被许可方如果要了解有关程序的信息以达到如下目的: （i）允许在独立创建的程序和其他程序（包括本程序）之间进行信息交换，以及（ii）允许对已经交换的信息进行相互使用，请与下列地址联系:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

只要遵守适当的条件和条款，包括某些情形下的一定数量的付费，都可获得这方面的信息。

本文档中描述的许可程序及其所有可用的许可资料均由 IBM 依据 IBM 客户协议、IBM 国际软件许可协议或任何同等协议中的条款提供。

此处包含的任何性能数据都是在受控环境中测得的。因此，在其他操作环境中获得的数据可能会有明显的不同。有些测量可能是在开发级的系统上进行的，因此不保证与一般可用系统上进行的测量结果相同。此外，有些测量是通过推算而估计的，实际结果可能会有差异。本文档的用户应当验证其特定环境的适用数据。

涉及非 IBM 产品的信息可从这些产品的供应商、其出版说明或其他可公开获得的资料中获取。IBM 没有对这些产品进行测试，也无法确认其性能的精确性、兼容性或任何其他关于非 IBM 产品的声明。有关非 IBM 产品性能的问题应当向这些产品的供应商提出。

所有关于 IBM 未来方向或意向的声明都可随时更改或收回，而不另行通知，它们仅仅表示了目标和意愿而已。

所有 IBM 的价格均是 IBM 当前的建议零售价，可随时更改而不另行通知。经销商的价格可与此不同。

本资料中包含用于日常业务运作的数据和报表的示例。为了尽可能完整地举例说明问题，这些示例包括了个人、公司、品牌和产品的名称。所有这些名称都是虚构的，如与实际商业企业所使用的名称和地址有雷同，纯属巧合。

版权许可:

本信息包括源语言形式的样本应用程序，这些样本说明不同操作平台上的编程方法。如果是为按照在编写样本程序的操作平台上的应用程序编程接口（API）进

行应用程序的开发、使用、经销或分发为目的，您可以任何形式对这些样本程序进行复制、修改、分发，而无须向 IBM 付费。这些示例并未在所有条件下作全面测试。因此，IBM 不能担保或暗示这些程序的可靠性、可维护性或功能。用户如果是为了按照 IBM 应用程序编程接口开发、使用、经销或分发应用程序，则可以任何形式复制、修改和分发这些样本程序，而无须向 IBM 付费。

凡这些样本程序的每份拷贝或其任何部分或任何衍生产品，都必须包括如下版权声明:

> ©（贵公司的名称）（年）。此部分代码是根据 IBM 公司的样本程序衍生出来的。© Copyright IBM Corp. （输入年份）。All rights reserved.

如果您正在查看此信息的软拷贝，则图片和彩色图例可能无法显示。

商标

AIX、DB2、DB2 Universal Database、Distributed Relational Database Architecture、NUMA-Q、OS/2、OS/390、OS/400、IBM Informix®、C-ISAM®、Foundation.2000™、IBM Informix ® 4GL、IBM Informix®DataBlade®Module、Client SDK™、Cloudscape™、Cloudsync™、IBM Informix®Connect、IBM Informix®Driver for JDBC、Dynamic Connect™、IBM Informix®Dynamic Scalable Architecture™(DSA)、IBM Informix®Dynamic Server™、IBM Informix®Enterprise Gateway Manager (Enterprise Gateway Manager)、IBM Informix®Extended Parallel Server™、i.Financial Services™、J/Foundation™、MaxConnect™、Object Translator™、Red Brick™、IBM Informix® SE、IBM Informix® SQL、InformiXML™、RedBack®、SystemBuilder™、U2™、UniData®、UniVerse® 和 wintegrate® 是 International Business Machines Corporation 的商标或注册商标。

Java 和所有基于 Java 的商标和徽标是 Sun Microsystems, Inc. 在美国和其他国家或地区的商标或注册商标。

Windows、Windows NT 和 Excel 是 Microsoft Corporation 在美国和／或其他国家或地区的注册商标或商标。

UNIX 是经 X/Open Company Limited 唯一许可的在美国和其他国家或地区的注册商标。

本出版物中使用的其他公司、产品和服务名称可能是其他公司的商标或服务标记。

索引

[A]

[B]

[C]

[D]

[F]

[G]

[H]

[J]

[K]

[L]

[M]

[P]

[Q]

[R]

[S]

[T]

[W]

[X]

[Y]

[Z]

[数字]

A

T

U

V

W

X

[特别字符]

中国印刷

G151-0116-00

Spine information:

IBM DB2 IBM Informix 版本 10.0/8.5

IBM Informix 数据库设计和实现指南