iAnywhere SOLUTIONS™
A SYBASE COMPANY

Adaptive Server® Anywhere
SQL 用户指南

部件号：DC32071-01-0902-01
上次修改时间：2005 年 1 月

版权所有

版权所有 © 1989-2005 Sybase, Inc. 部分版权所有 © 2001-2004 iAnywhere Solutions, Inc. 保留所有权利。

未经 iAnywhere Solutions, Inc. 的事先书面许可，本书的任何部分不能以任何形式、任何手段 （电子的、机械的、手工的、光学的或其它手段）进行复制、传播或翻译。 iAnywhere Solutions, Inc. 是 Sybase, Inc. 的子公司。

Sybase、 SYBASE （徽标）、 AccelaTrade、 ADA Workbench、 Adaptable Windowing Environment、 Adaptive Component Architecture、 Adaptive Server、 Adaptive Server Anywhere、 Adaptive Server Enterprise、 Adaptive Server Enterprise Monitor、 Adaptive Server Enterprise Replication、 Adaptive Server Everywhere、 Adaptive Server IQ、 Adaptive Warehouse、 AnswerBase、 Anywhere Studio、 Application Manager、 AppModeler、 APT Workbench、 APT-Build、 APT-Edit、 APT-Execute、 APT-Library、 APT-Translator、 ASEP、 AvantGo、 AvantGo Application Alerts、 AvantGo Mobile Delivery、 AvantGo Mobile Document Viewer、 AvantGo Mobile Inspection、 AvantGo Mobile Marketing Channel、 AvantGo Mobile Pharma、 AvantGo Mobile Sales、 AvantGo Pylon、 AvantGo Pylon Application Server、 AvantGo Pylon Conduit、 AvantGo Pylon PIM Server、 AvantGo Pylon Pro、 Backup Server、 BayCam、 Bit-Wise、 BizTracker、 Certified PowerBuilder Developer、 Certified SYBASE Professional、 Certified SYBASE Professional 徽标、 ClearConnect、 Client Services、 Client-Library、 CodeBank、 Column Design、 ComponentPack、 Connection Manager、 Convoy/DM、 Copernicus、 CSP、 Data Pipeline、 Data Workbench、 DataArchitect、 Database Analyzer、 DataExpress、 DataServer、 DataWindow、 DB-Library、 dbQueue、 Developers Workbench、 Direct Connect Anywhere、 DirectConnect、 Distribution Director、 Dynamic Mobility Model、 Dynamo、 e-ADK、 E-Anywhere、 e-Biz Integrator、 EC Gateway、 ECMAP、 ECRTP、 eFulfillment Accelerator、 Electronic Case Management、 Embedded SQL、 EMS、 Enterprise Application Studio、 Enterprise Client/Server、 Enterprise Connect、 Enterprise Data Studio、 Enterprise Manager、 Enterprise Portal （徽标）、 Enterprise SQL Server Manager、 Enterprise Work Architecture、 Enterprise Work Designer、 Enterprise Work Modeler、 eProcurement Accelerator、 eremote、 Everything Works Better When Everything Works Together、 EWA、 E-Whatever、 Financial Fusion、 Financial Fusion （及设计）、 Financial Fusion Server、 Formula One、 Fusion Powered e-Finance、 Fusion Powered Financial Destinations、 Fusion Powered STP、 Gateway Manager、 GeoPoint、 GlobalFIX、 iAnywhere、 iAnywhere Solutions、 ImpactNow、 Industry Warehouse Studio、 InfoMaker、 Information Anywhere、 Information Everywhere、 InformationConnect、 InstaHelp、 Intelligent Self-Care、 InternetBuilder、 iremote、 iScript、 Jaguar CTS、 jConnect for JDBC、 KnowledgeBase、 Logical Memory Manager、 Mail Anywhere Studio、 MainframeConnect、 Maintenance Express、 Manage Anywhere Studio、 MAP、 M-Business Channel、 M-Business Network、 M-Business Server、 MDI Access Server、 MDI Database Gateway、 media.splash、 Message Anywhere Server、 MetaWorks、 MethodSet、 ML Query、 MobiCATS、 My AvantGo、 My AvantGo Media Channel、 My AvantGo Mobile Marketing、 MySupport、 Net-Gateway、 Net-Library、 New Era of Networks、 Next Generation Learning、 Next Generation Learning Studio、 O DEVICE、 OASiS、 OASiS 徽标、 ObjectConnect、 ObjectCycle、 OmniConnect、 OmniSQL Access Module、 OmniSQL Toolkit、 Open Biz、 Open Business Interchange、 Open Client、 Open Client/Server、 Open Client/Server Interfaces、 Open ClientConnect、 Open Gateway、 Open Server、 Open ServerConnect、 Open Solutions、 Optima++、 Orchestration Studio、 Partnerships that Work、 PB-Gen、 PC APT Execute、 PC DB-Net、 PC Net Library、 PhysicalArchitect、 Pocket PowerBuilder、 PocketBuilder、 Power Through Knowledge、 power.stop、 Power++、 PowerAMC、 PowerBuilder、 PowerBuilder Foundation Class Library、 PowerDesigner、 PowerDimensions、 PowerDynamo、 Powering the New Economy、 PowerJ、 PowerScript、 PowerSite、 PowerSocket、 Powersoft、 Powersoft Portfolio、 Powersoft Professional、 PowerStage、 PowerStudio、 PowerTips、 PowerWare Desktop、 PowerWare Enterprise、 ProcessAnalyst、 QAnywhere、 Rapport、 Relational Beans、 RepConnector、 Replication Agent、 Replication Driver、 Replication Server、 Replication Server Manager、 Replication Toolkit、 Report Workbench、 Report-Execute、 Resource Manager、 RW-DisplayLib、 RW-Library、 S.W.I.F.T. Message Format Libraries、 SAFE、 SAFE/PRO、 SDF、 Secure SQL Server、 Secure SQL Toolset、 Security Guardian、 SKILS、 smart.partners、 smart.parts、 smart.script、 SQL Advantage、 SQL Anywhere、 SQL Anywhere Studio、 SQL Code Checker、 SQL Debug、 SQL Edit、 SQL Edit/TPU、 SQL Everywhere、 SQL Modeler、 SQL Remote、 SQL Server、 SQL Server Manager、 SQL Server SNMP SubAgent、 SQL Server/CFT、 SQL Server/DBM、 SQL SMART、 SQL Station、 SQL Toolset、 SQLJ、 Stage III Engineering、 Startup.Com、 STEP、 SupportNow、 Sybase Central、 Sybase Client/Server Interfaces、 Sybase Development Framework、 Sybase Financial Server、 Sybase Gateways、 Sybase Learning Connection、 Sybase MPP、 Sybase SQL Desktop、 Sybase SQL Lifecycle、 Sybase SQL Workgroup、 Sybase Synergy Program、 Sybase User Workbench、 Sybase Virtual Server Architecture、 SybaseWare、 Syber Financial、 SyberAssist、 SybMD、 SyBooks、 System 10、 System 11、 System XI （徽标）、 SystemTools、 Tabular Data Stream、 The Enterprise Client/Server Company、 The Extensible Software Platform、 The Future Is Wide Open、 The Learning Connection、 The Model For Client/Server Solutions、 The Online Information Center、 The Power of One、 TotalFix、 TradeForce、 Transact-SQL、 Translation Toolkit、 Turning Imagination Into Reality、 UltraLite、 UltraLite.NET、 UNIBOM、 Unilib、 Uninull、 Unisep、 Unistring、 URK Runtime Kit for UniCode、 Versacore、 Viewer、 VisualWriter、 VQL、 Warehouse Control Center、 Warehouse Studio、 Warehouse WORKS、 WarehouseArchitect、 Watcom、 Watcom SQL、 Watcom SQL Server、 Web Deployment Kit、 Web.PB、 Web.SQL、 WebSights、 WebViewer、 WorkGroup SQL Server、 XA-Library、 XA-Server 和 XP Server 是 Sybase, Inc. 或其子公司的商标。

所有其它商标均归各自的所有者所有。

目录

关于本手册 ix
SQL Anywhere Studio 文档 x
文档约定 xiii
Adaptive Server Anywhere 示例数据库 xvi
查找详细信息并提供反馈 xvii

1 设计数据库 3
简介 4
数据库设计概念 5
设计过程 12
设计数据库表属性 29

2 使用数据库对象 31
简介 32
使用数据库 33
使用表 44
使用视图 59
使用索引 68
使用临时表 82
在 Sybase Central 中复制数据库对象 83

3 确保数据完整性 85
数据完整性概述 86
使用列缺省值 90
使用表和列约束 96
使用域 101
实施实体完整性和参照完整性 105
系统表中的完整性规则 110

4 使用事务和隔离级别 111
事务简介 112
隔离级别和一致性 116
事务阻塞和死锁 124
选择隔离级别 127
隔离级别教程 131
锁定的工作方式 146
特殊并发问题 159
复制和并发 162
小结 164

5 监控和提高性能 167
性能分析工具 168
最佳性能提示 175
使用高速缓存提高性能 191
使用索引改进性能 197
使用键提高查询性能 198
查询结果排序 200
在查询处理中使用工作表 201
监控数据库性能 203
碎片 209
分析数据库过程 213

6 查询：从表中选择数据 225
查询概述 226
SELECT 列表：指定列 229
FROM 子句：指定表 237
WHERE 子句：指定行 238

7 对查询结果进行汇总、分组和排序 251
使用集合函数汇总查询结果 252
GROUP BY 子句：将查询结果划分为组 258
了解 GROUP BY 260
HAVING 子句：选择数据组 264
ORDER BY 子句：对查询结果进行排序 266
使用 UNION、 INTERSECT 和 EXCEPT 对查询结果执行集合运算 270
标准和兼容性 278

8 连接：从多个表检索数据 281
示例数据库模式 282
连接的工作原理 283
连接概述 284
显式连接条件 （ON 短语） 290
交叉连接 294
内连接和外连接 296
专用连接 304
自然连接 312
键连接 316

9 公用表表达式 331
关于公用表表达式 332
公用表表达式的典型应用 336
递归公用表表达式 340
部件激增问题 343
递归公用表表达式中的数据类型声明 347
最短距离问题 349
使用多个递归公用表表达式 353

10 使用 OLAP 355
了解 OLAP 356
了解小计 357
NULL 值和小计行 360
使用 ROLLUP 362
使用 CUBE 368
使用 GROUPING SETS 371
使用 OLAP 函数 374

11 使用子查询 383
子查询简介 384
在 WHERE 子句中使用子查询 385
HAVING 子句中的子查询 386
子查询比较测试 389
使用 ANY 和 ALL 的定量比较测试 391
用 IN 条件测试集合成员资格 395
存在测试 397
外部引用 399

子查询和连接 400
嵌套的子查询 403
子查询的工作方式 405

12 添加、更改和删除数据 415
数据修改语句 416
使用 INSERT 添加数据 417
使用 UPDATE 更改数据 422
使用 INSERT 更改数据 424
使用 DELETE 删除数据 425

13 查询优化与执行 427
优化程序的作用 428
优化程序的工作原理 429
查询执行算法 439
物理数据组织和访问 455
索引 458
语义查询转换 466
子查询和函数高速缓存 480
读取访问计划 482

14 Transact-SQL 兼容性 503
Transact-SQL 支持概述 504
Adaptive Server 体系结构 507
为实现 Transact-SQL 兼容性配置数据库 514
编写兼容的 SQL 语句 523
Transact-SQL 过程语言概述 529
存储过程的自动转换 532
从 Transact-SQL 过程中返回结果集 534
Transact-SQL 过程中的变量 535
Transact-SQL 过程中的错误处理 536

15 与其它版本 SQL 语言的区别 539
Adaptive Server Anywhere SQL 功能 540

16 在数据库中使用 XML 545
什么是 XML？ 546

在关系数据库中存储 XML 文档 547
将关系数据导出为 XML 548
导入 XML 文档作为关系数据 550
以 XML 格式获取查询结果 558
使用 SQL/XML 以 XML 的形式获取查询结果 582

17 导入和导出数据 593
将数据传入与传出数据库 594
批量操作的性能问题 595
批量操作的数据恢复问题 596
将数据导入数据库 597
将数据从数据库中导出 608
重建数据库 620
抽取数据 630
将数据库迁移到 Adaptive Server Anywhere 631
使用 SQL 命令文件 637
Adaptive Server Enterprise 兼容性 640

18 访问远程数据 641
简介 642
访问远程数据的基本概念 644
使用远程服务器 646
使用外部登录 653
使用代理表 655
连接远程表 660
连接多个本地数据库中的表 662
将本机语句发送到远程服务器 663
使用远程过程调用 (RPC) 664
事务管理和远程数据 667
内部操作 669
远程数据访问疑难解答 673

19 用于进行远程数据访问的服务器类 677
概述 678
基于 JDBC 的服务器类 679
基于 ODBC 的服务器类 683

20 使用过程、触发器和批处理 699
过程和触发器概述 700
过程和触发器的优点 701
过程简介 702
用户定义的函数简介 710
触发器简介 714
批处理简介 722
控制语句 724
过程和触发器的结构 728
从过程返回结果 732
在过程和触发器中使用游标 738
过程和触发器中的错误和警告 742
在过程中使用 EXECUTE IMMEDIATE 语句 752
过程和触发器中的事务和保存点 754
编写过程的提示 755
批处理中允许的语句 757
从过程调用外部库 758

21 调试数据库中的逻辑 767
调试数据库简介 768
教程：调试程序入门 770
使用断点 780
使用变量 783
使用连接 785

索引 787

关于本手册

主题

这本手册介绍如何设计和创建数据库；如何导入、导出和修改数据；如何检索数据以及如何建立存储过程和触发器。

读者

本手册供 Adaptive Server Anywhere 的所有用户使用。

预备知识

本手册假定您了解有关数据库管理系统 （特别是 Adaptive Server Anywhere）的基础知识。如果您对此还不熟悉，应考虑先阅读 *SQL Anywhere Studio 介绍*，然后再阅读本手册。

SQL Anywhere Studio 文档

本手册是 SQL Anywhere 文档集的一部分。本节介绍文档集中包含的手册及其使用方法。

SQL Anywhere Studio 文档

我们以多种形式提供了 SQL Anywhere Studio 文档：联机文档，将所有手册都合并到一个大帮助文件中； PDF 文件，每本手册有一个单独的 PDF 文件；以及可以购买到的印刷手册。这些文档包括以下手册：

- **SQL Anywhere Studio 介绍** 这本手册概述了 SQL Anywhere Studio 数据库管理和同步技术。手册中提供了一些教程，分别介绍 SQL Anywhere Studio 的各个组成部分。
- **SQL Anywhere Studio 的新功能** 这本手册面向的读者是该软件旧版本的用户。该手册列出了本产品此版本和以前版本的新功能，并介绍了升级步骤。
- **Adaptive Server Anywhere 数据库管理指南** 这本手册介绍有关运行、管理和配置数据库和数据库服务器等方面的内容。
- **Adaptive Server Anywhere SQL 用户指南** 这本手册介绍如何设计和创建数据库；如何导入、导出和修改数据；如何检索数据以及如何建立存储过程和触发器。
- **Adaptive Server Anywhere SQL 参考手册** 这本手册为 Adaptive Server Anywhere 使用的 SQL 语言提供了一套完整的参考资料。还介绍了 Adaptive Server Anywhere 系统表和过程。
- **Adaptive Server Anywhere 编程指南** 这本手册介绍如何使用 C、C++ 和 Java 编程语言建立和部署数据库应用程序。使用 Visual Basic 和 PowerBuilder 等工具的用户可以使用这些工具提供的编程接口。它还介绍了 Adaptive Server Anywhere ADO.NET 数据提供程序。
- **Adaptive Server Anywhere SNMP Extension Agent 用户指南** 这本手册介绍如何配置 Adaptive Server Anywhere SNMP Extension Agent，以使用 SNMP 管理应用程序来管理 Adaptive Server Anywhere 数据库。

- **Adaptive Server Anywhere 错误消息** 这本手册提供了 Adaptive Server Anywhere 错误消息及其诊断信息的完整列表。

- **SQL Anywhere Studio 安全指南** 这本手册提供有关 Adaptive Server Anywhere 数据库中的安全功能的信息。 Adaptive Server Anywhere 7.0 荣获美国政府授予的 TCSEC （可信计算机系统评估标准） C2 安全等级。对于那些希望以相当于 C2 认证环境的方式运行当前版本的 Adaptive Server Anywhere 的用户，他们可能会对这本手册很感兴趣。

- **MobiLink 管理指南** 这本手册介绍如何使用用于移动计算的 MobiLink 数据同步系统，该系统支持在单个 Oracle、 Sybase、 Microsoft 或 IBM 数据库与多个 Adaptive Server Anywhere 或 UltraLite 数据库之间共享数据。

- **MobiLink 客户端** 这本手册介绍如何设置和同步 Adaptive Server Anywhere 及 UltraLite 远程数据库。

- **MobiLink 服务器启动同步用户指南** 这本手册介绍 MobiLink 服务器启动的同步，利用这项 MobiLink 功能，可从统一数据库启动同步。

- **MobiLink 教程** 这本手册提供多个教程来指导您如何设置和运行 MobiLink 应用程序。

- **QAnywhere 用户指南** 这本手册介绍 MobiLink QAnywhere。该产品是一个消息传送平台，支持开发和部署用于移动和无线客户端，以及传统桌面和便携式客户端的消息传送应用程序。

- **iAnywhere Solutions ODBC 驱动程序** 这本手册介绍如何设置 ODBC 驱动程序以从 MobiLink 同步服务器和 Adaptive Server Anywhere 远程数据访问功能访问除 Adaptive Server Anywhere 之外的统一数据库。

- **SQL Remote 用户指南** 这本手册介绍用于移动计算的 SQL Remote 数据复制系统的各个方面，此系统支持在单个 Adaptive Server Anywhere 或 Adaptive Server Enterprise 数据库与多个 Adaptive Server Anywhere 数据库之间共享数据 （使用电子邮件或文件传输等间接链接）。

- **SQL Anywhere Studio 帮助** 这本手册包含 Sybase Central、Interactive SQL 和其它图形工具的上下文相关帮助。印刷文档集中没有这本手册。
- **UltraLite 数据库用户指南** 这本手册面向的读者为所有 UltraLite 开发人员。它介绍了 UltraLite 数据库系统并提供了所有 UltraLite 编程接口的公共信息。
- **UltraLite 接口指南** 对于各个 UltraLite 编程接口都单独提供了一本手册。这些接口中有一些是作为 UltraLite 组件提供的，以便于快速开发应用程序；另外一些是作为静态接口提供的，以便于 C、C++ 和 Java 开发。

除本文档集外， PowerDesigner 和 InfoMaker 还有自己的联机文档。

文档格式

SQL Anywhere Studio 按下列形式提供文档：

- **联机文档** 这些联机文档包含完整的 SQL Anywhere Studio 文档，其中包括手册和 SQL Anywhere 工具的上下文相关帮助。每次发布产品的维护版本时都会对联机文档进行更新，因此联机文档是最全和最新的文档来源。

  若要在 Windows 操作系统中阅读联机文档，请选择 [开始] → [程序] → [SQL Anywhere 9] → [联机手册]。可以使用左窗格中的 HTML 帮助目录、索引和搜索功能，以及右窗格中的链接和菜单来浏览联机文档。

  若要在 UNIX 操作系统上阅读联机文档，请查看 SQL Anywhere 安装目录下的 HTML 文档。

- **PDF 手册** 以一组 PDF 文件的形式提供的 SQL Anywhere 手册，可以使用 Adobe Acrobat Reader 进行查看。

  可通过联机手册或 Windows [开始] 菜单来查看 PDF 手册。

- **印刷手册** 可从 Sybase 销售部门或 eShop 购买整套文档集，Sybase 网上商店的网址为 http://eshop.sybase.com/eshop/documentation。

文档约定

本节列出了文档中使用的印刷和图形约定。

语法约定

在 SQL 语法说明中，使用了以下约定：

- **关键字** 所有 SQL 关键字都以大写字母显示，就像下例中的 ALTER TABLE：

  ALTER TABLE [*owner*.]*table-name*

- **占位符** 对于必须替换为相应的标识符或表达式的项，其显示方式就像下例中的 *owner* 和 *table-name*：

  ALTER TABLE [*owner*.]*table-name*

- **重复项** 对于重复项列表，只显示列表中的一个元素，后跟省略号 （三个句点），如以下示例中的 *column-constraint*：

  ADD *column-definition* [*column-constraint, …*]

  允许指定一个或多个列表元素。在此示例中，如果指定了多个元素，则必须用逗号将它们隔开。

- **可选部分** 语句的可选部分放在方括号内。

  RELEASE SAVEPOINT [*savepoint-name*]

  上面的方括号表示 *savepoint-name* 是可选的。不应键入方括号。

- **选项** 如果不必选择项目列表的项目或者只能选中一个，则用垂直条分隔这些项目，并将列表放在方括号内。

  [**ASC** | **DESC**]

  例如，可以从 ASC、 DESC 中任选一个，或者一个也不选。不应键入方括号。

- **二选一选项** 如果必须明确选择一个选项，则会将替换选项放在大括号内，并用一条竖线分隔选项。

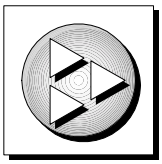
  [**QUOTES** { **ON** | **OFF** }]

如果使用了 QUOTES 选项，则必须提供 ON 和 OFF 两者之一。不应键入方括号和大括号。

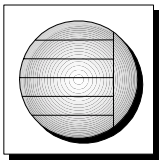
图标

本文档中使用了下列图标。

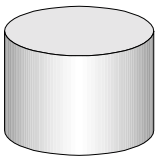
- 客户端应用程序。



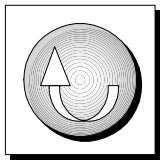
- 数据库服务器，例如 Sybase Adaptive Server Anywhere。



- 数据库。在某些高水平的图中，可以使用此图标表示数据库和管理数据库的数据库服务器。



- 复制或同步中间件。用于帮助在数据库之间共享数据。例如 MobiLink 同步服务器和 SQL Remote 消息代理。



- 编程接口。



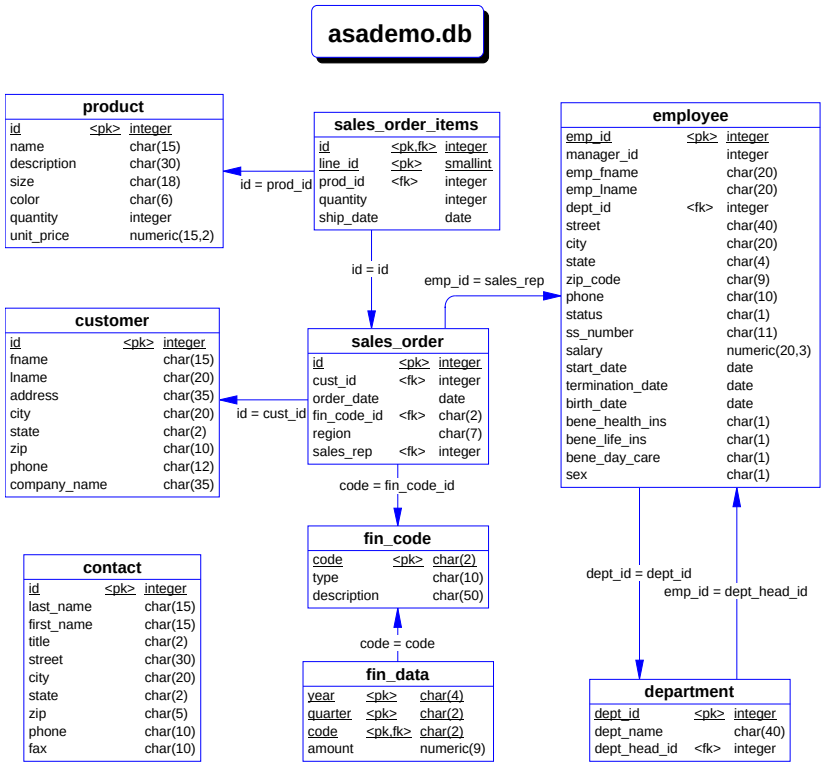
Adaptive Server Anywhere 示例数据库

文档中的许多示例都使用 Adaptive Server Anywhere 示例数据库。

该示例数据库包含在名为 *asademo.db* 的文件中，并位于 SQL Anywhere 目录中。

该示例数据库是一家小公司的数据库。其中包含公司的内部信息 （雇员、部门和财务信息）、产品信息和销售信息 （销售订单、客户和联系方式）。数据库中的所有信息都是虚构的。

下图显示该示例数据库中的表以及这些表之间的相互关系。



查找详细信息并提供反馈

查找详细信息

附加信息和资源，包括交换代码，可从 iAnywhere Developer Network 获得，网址是 http://www.ianywhere.com/developer/。

如果您有问题或是需要帮助，可将邮件发布到下面所列的 iAnywhere Solutions 新闻组。

当您向这些新闻组发布邮件时，请提供问题的详细信息，包括 SQL Anywhere Studio 版本的编译版本号。您可以在命令提示符下键入 **dbeng9 -v** 来得到这些信息。

新闻组位于 *forums.sybase.com* 新闻服务器上。这些新闻组包括：

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

新闻组免责声明
iAnywhere Solutions 没有义务为其新闻组提供解决方案、信息或建议；除提供系统操作员监控服务和确保新闻组的运行和可用性外，iAnywhere Solutions 没有义务提供任何其它服务。

如果时间允许， iAnywhere Solutions 技术顾问和其他员工也会提供新闻组服务。他们是在自愿的基础上提供帮助的，所以可能无法定期提供解决方案和信息。他们可以提供多少帮助取决于他们的工作量。

反馈

我们欢迎您就本文档提出意见、建议或其它反馈信息。

您可以将意见和建议通过电子邮件发送到 SQL Anywhere 文档小组，地址为 iasdoc@ianywhere.com。虽然我们不回复该地址收到的电子邮件，但我们会认真阅读所有建议。

此外，您还可以通过上面所列的新闻组就文档和软件提供反馈信息。

第 1 部分　设计和创建数据库

这一部分介绍了设计和建立数据库的主要概念和策略。它讲述了数据库设计问题以及使用表、视图和索引的方法，还提供了有关参照完整性和事务方面的内容。

第 1 章

设计数据库

关于本章

本章介绍关系数据库设计的基本概念，并为您提供循序渐进的指导，帮助您设计自己的数据库。本章使用称作概念数据建模的方法进行讲解，集中讨论实体以及实体间的关系。

简介

就中小型数据库而言，数据库设计虽然并不困难，但却非常重要。数据库设计的不好，会导致效率低下，并且可能造成数据库系统不可靠。客户端应用程序是针对数据库的特定部分来建立的，并且依赖于数据库设计，因此不好的设计可能很难在以后进行修订。

有关详细信息，请参见一些介绍性读物，例如，由 C. J. Date 编著的 *A Database Primer*。如果您对学习数据库理论很感兴趣， C. J. Date 编著的 *An Introduction to Database Systems* 是一本有关这方面很好的教科书。

数据库设计概念

在设计数据库时，需要计划要存储有关哪些事物的信息，以及要保存有关各个事物的哪些信息。您还需要确定这些事物的相互关系。如果使用数据库设计中的术语，在这一步创建的数据库原型就称作**概念数据库模型**。

实体和关系

要存储其相关信息的可识别对象或事物称作**实体**。它们之间的关联称作**关系**。在数据库描述语言中，可以将实体看做名词，将关系看做动词。

由于概念模型对实体和关系进行了明确的区分，因此这种模型非常有用。这种模型将在任何特定数据库管理系统中实施设计所涉及的细节隐藏起来，从而使设计者可以集中考虑基础数据库结构。因此，这种模型也成为了一种用于讨论数据库设计的通用语言。

实体关系图

概念数据库模型主要由一个显示实体和关系的示意图构成。这个示意图通常称作**实体关系图**。因此，许多人也使用实体关系建模这个词来指创建概念数据库模型的任务。

概念数据库设计是一个由上至下的设计方法。现在有许多功能完备的工具可以帮助您按照这种方法或其他方法进行设计，例如， Sybase PowerDesigner。虽然本章的目的只是进行介绍，但也提供了足够的信息可以帮助您设计简单的数据库。

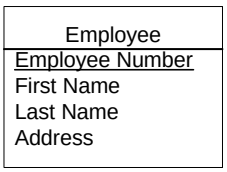
实体

在数据库中，一个实体对应于一个名词。可识别的对象，例如，雇员、订单项、部门和产品，都是实体的示例。在数据库中用表代表各个实体。置入数据库的实体都源于要使用数据库执行的活动，例如，跟踪销售电话和维护雇员信息，等等。

属性

每个实体都包含一些**属性**。属性是指要为事物存储的特定特性。例如，在雇员实体中，需要存储雇员 ID 号、姓氏和名字、地址，以及与一个特定雇员相关的其他信息。属性也称作特性。

实体用一个矩形框表示。在矩形框内部，列出与该实体相关联的属性。



标识符是指所有其他属性都依赖的一个或多个属性。它在实体中唯一地标识一个项目。在要组成标识符的属性名下面加上下划线。

在上面的 Employee 实体中， Employee Number 唯一地标识一个雇员。所有其他属性都存储只与那个雇员相关的信息。例如，一个雇员编号唯一地确定一个雇员的名字和地址。两个雇员可能具有相同的名字或相同的地址，但可以确保他们的雇员编号不同。 Employee Number 下面带有下划线，表示它是标识符。

为每个实体都创建一个标识符是一个良好的习惯。这些标识符在表中将成为主键，下文中将对此进行说明。主键值必须唯一，并且不能为空或未定义。主键唯一地标识表中的每一行，并且提高数据库服务器的性能。

关系

在数据库中，实体之间的一个关系对应于一个动词。一个雇员属于一个部门，或者一个办事处位于一座城市。数据库中的关系可能表现为表间的外键关系，也可能自身就成为独立的表。您将在本章中看到这两种情况的示例。

数据库中的关系就是控制实体中数据的规则或惯例的编码。如果每个部门有一个部门经理，可以在部门和雇员之间建立一对一的关系来标识该部门经理。

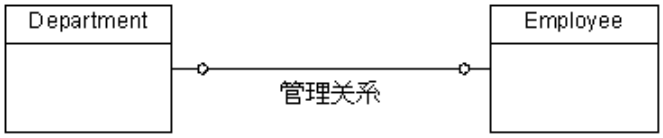
当关系置入数据库结构后，就不可能再出现例外。没有地方可以输入另一个部门经理。复制部门条目将复制部门 ID，而它是标识符。标识符不允许有重复。

提示
严格的数据库结构有很大好处，因为它可以消除不一致的问题，例如一个部门有两个经理。另一方面，作为设计者，您应该使设计具有足够的灵活性以便于进行扩展，以满足某些未预见到的需要。对设计合理的数据库进行扩展通常并不很困难，但修改现有表结构可能会致使整个数据库及其客户端应用程序无法使用。

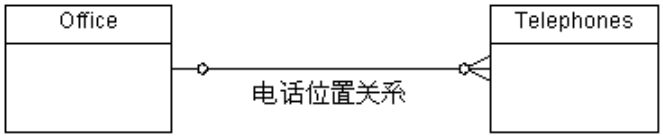
关系的基数

表之间有三种关系。这三种关系对应于关系中所涉及的实体的**基数**（数量）。

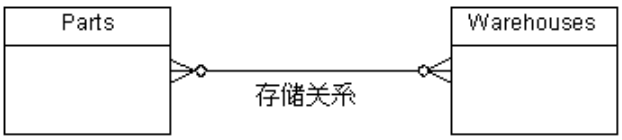
- **一对一关系**　关系通过在两个实体间画一条连线表示。连线上可以有其他标记，例如，下图中所示的两个圆圈。这些标记的用途将在下文中进行说明。在下图中，一个部门由一个雇员管理。



- **一对多关系**　如果 [实体 1] 中包含的一项可以与 [实体 2] 中的多项相关联，这样一种关系用多条连线连接到 [实体 2] 来表示。在下图中，一个办事处可以有多部电话。



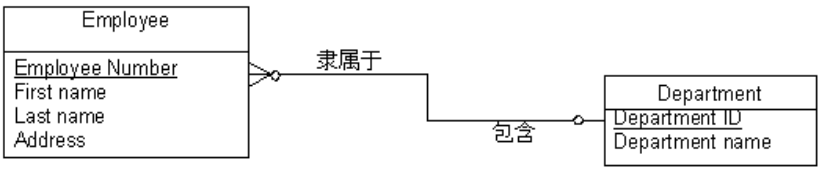
- **多对多关系**　在这种情况下，两个实体的连接处都要画多条连线。这表示一个仓库可以存放许多不同的部件，而同一类部件也可以存放在许多仓库中。



角色

您可以用两个**角色**来描述每种关系。角色是用于从每个观察点描述关系的动词或短语。例如，雇员和部门之间的关系可以用以下两个角色来描述：

1. 雇员*属于*部门。
2. 而部门*包含*雇员。



角色非常重要，因为它们为您提供了一种方便且有效的方法来验证您的工作。

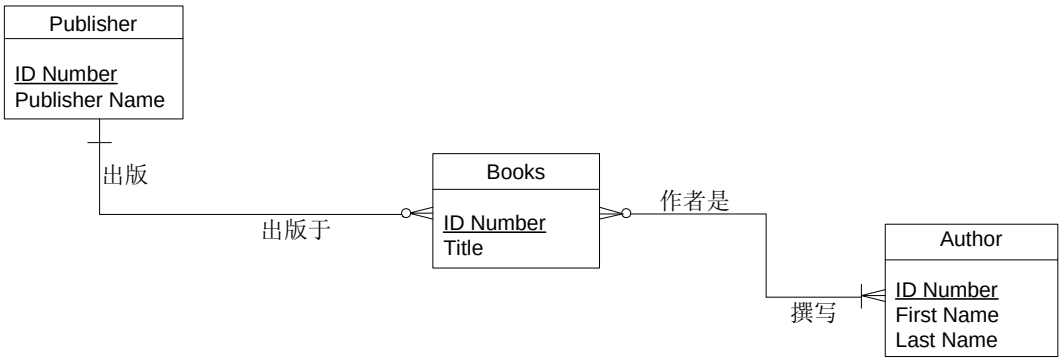
提示

不管是从左到右读取还是从右到左读取，下面的规则都会使读取这些图示变得容易：读取 (1) 第一个实体的名称， (2) *第一个实体*旁边的角色， (3) 到*第二个实体*的连接的基数， (4) 第二个实体的名称。

强制元素

表示关系的连线末端的小圆圈具有非常重要的作用。圆圈表示存在于该实体内的元素在另一个实体内不必有对应的元素。

如果出现的是一段交叉线而不是圆圈，则表示另一个实体中的每个元素在该实体中*至少*应有一个对应元素。下面举例说明这些标记的含义。



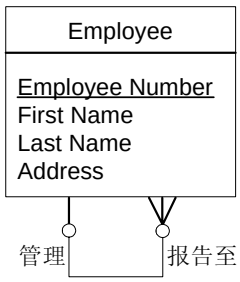
此图具有以下四个含义：

1. 一家出版社出版了*零或多*本书。
2. 一本书由*恰好*一家出版社出版。
3. 一本书由*一或多*位作者撰写。
4. 一位作者撰写了*零或多*本书。

提示
可以把小圆圈看做数字 0，把交叉线看做数字 1。圆圈表示*至少零*。交叉线表示*至少一*。

反身关系

有时，同一个实体内的条目之间也存在关系。这种关系称作**反身**关系。关系的两端都连接到同一个实体。



此图具有以下两个含义。

1. 一个雇员最多只向另外一个雇员报告。

2. 一个雇员管理零个或多个雇员。

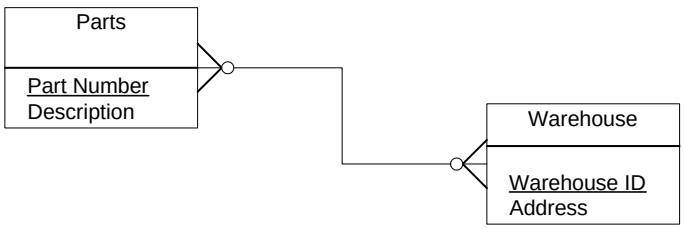
请注意，在这个关系中，关系在两个方向上都应该是可选的。某些雇员不是经理。同样，至少应该有一个雇员是公司的总经理，因此不向任何人报告。

自然，还应指定一个雇员不能是他自己的经理。这个限制是一种*业务规则*。业务规则将在下文中作为 「设计过程」第 12 页的一部分进行讨论。

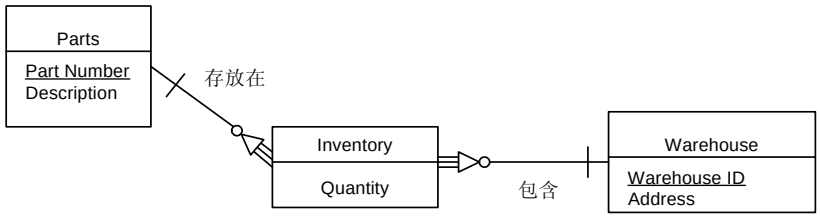
将多对多关系更改为实体

如果有属性与*关系*相关联，而不是与实体相关联，则可以将该关系更改为实体。有时，多对多关系可能会出现这种情况。有些属性特定于关系，因此将其添加到任何一个实体中都不合理。

假设部件存放在多个不同的仓库中。而您画的设计图如下所示。



但您希望记录各个部件在各个地点的存货数量。该属性只能与关系相关联。每个数量都依赖于所涉及的部件和仓库。要表示这样一种情况，可以按以下方式重画设计图：



请注意以下细节的变化：

1. 两个新关系将关系实体分别与原有的两个实体连接起来。这两个关系的名称继承自原有关系的两个角色：分别为*存放在*和*包含*

2. Inventory 实体中的每个条目要求 Parts 实体中有一个强制条目，Warehouse 实体中有一个强制条目。这些关系都是强制的，因为仓储关系只在与一个特定部件和一个特定仓库相关联时才有意义。

3. 新实体既依赖于 Parts 实体，也依赖于 Warehouse 实体，表示新实体由这两个实体的标识符共同标识。在这个新设计图中， Parts 实体中的一个标识符和 Warehouse 实体中的一个标识符唯一地标识 Inventory 实体中的一个条目。圆圈和多线条之间的三角形将两个新关系连接到新的 Inventory 实体，并表示依赖性。

不要在 Inventory 实体中添加 Part Number 或 Warehouse ID 特性。Inventory 实体中的每个条目都依赖于一个特定部件和一个特定仓库，但这些三角形可以更清晰的表示这种依赖性。

设计过程

设计过程包含五个主要步骤。

- 「第 1 步：确定实体和关系」第 12 页 .
- 「第 2 步：确定所需数据」第 15 页 .
- 「第 3 步：规范化数据」第 17 页 .
- 「第 4 步：解析关系」第 23 页 .
- 「第 5 步：验证设计」第 26 页 .

有关实现数据库设计的详细信息，请参见 「使用数据库对象」第 31 页。

第 1 步：确定实体和关系

❖ **确定设计中的实体及实体之间的关系：**

1 **确定高级别的活动** 确定要使用该数据库执行的一般活动。例如，可能要用它来跟踪有关雇员的信息。

2 **确定实体** 对于这些活动，确定需要维护有关哪些类对象的信息。这些对象将成为实体。例如，聘用*雇员*，将雇员分配到某个*部门*，并确定其*技能*级别。

3 **确定关系** 对这些活动进行分析，然后确定实体间会有哪些关系。例如，部件和仓库之间有关系。定义两个角色来描述每个关系。

4 **分解活动** 开始时确定了高级别的活动。现在，需要进一步分析这些活动，确定是否可以将其中一些分解为较低级别的活动。例如，象*维护雇员信息*这样一个高级别活动可以分解为：

- 添加新雇员。
- 更改现有雇员信息。
- 删除已离职的雇员。

5 **确定业务规则**　对业务说明进行分析，确定应遵守哪些规则。例如，[一个部门有且仅有一个经理] 就可以作为一个业务规则。这些规则将被置入数据库的结构中。

实体和关系示例

示例

ACME Corporation 是一家小公司，它在五个地点设有办事处。目前，ACME 有 75 名雇员。该公司正准备迅速发展，并且已经确定了九个部门，每个部门都有自己的部门经理。

为帮助公司招聘新雇员，人事部门确定了 68 项技能，并且认为公司将来需要具有这些技能的雇员。聘用了一个雇员后，将对该雇员在每项技能上的专业级别进行评定。

确定高级别的活动

ACME Corporation 需要考虑的高级别活动有：

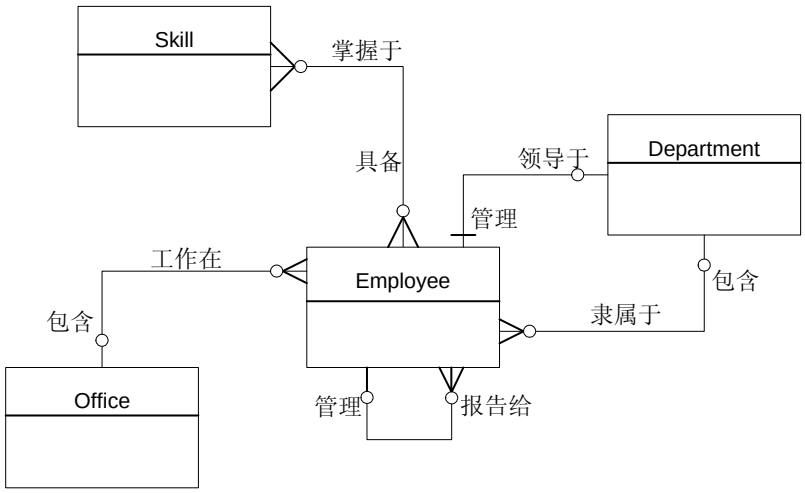
- 聘用雇员。
- 解聘雇员。
- 维护雇员的个人信息。
- 维护有关公司所需技能的信息。
- 维护有关哪个雇员具有哪项技能的信息。
- 维护有关部门的信息。
- 维护有关办事处的信息。

确定实体和关系

确定实体 （对象）和连接实体的关系 （角色）。根据以上说明和高级别的活动创建一个设计图。

用矩形框表示实体，用连线表示关系。用两个角色标记每个关系。还应使用适当的标注表示那些一对多、一对一和多对多关系。

下面是一个粗略的实体关系图。在本章后面的部分将逐渐对其进行改进。



分解高级别的活动

根据上述高级别的活动可以确定以下较低级别的活动：

- 添加或删除雇员。
- 添加或删除办事处。
- 列出某个部门的雇员。
- 在技能列表中添加技能。
- 确定某个雇员的技能。
- 确定某个雇员在各项技能上的技能级别。
- 确定在某项技能上具有相同技能级别的所有雇员。
- 更改雇员的技能级别。

使用这些较低级别的活动可以确定是否需要新表或新关系。

确定业务规则

业务规则通常可以表示为一对多、一对一和多对多关系。

可能相关的业务规则有以下几个：

- 现在有五个办事处；扩展计划允许增加到最多十个。

- 雇员可以更换部门或办事处。
- 每个部门有一个部门经理。
- 每个办事处最多有三个电话号码。
- 每个电话号码有一个或多个分机。
- 聘用了一个雇员后，将对该雇员在各项技能上的专业级别进行评定。
- 每个雇员具有三到二十项技能。
- 可以将雇员分配到一个办事处，也可以不分配。

第 2 步：确定所需数据

❖ **确定所需数据：**

1 确定支持数据。

2 列出所有需要跟踪的数据。

3 为每个实体设置数据。

4 列出每个实体的可用数据。描述实体 （对象）的数据可以回答涉及何人、何事、何处、何时以及何故的问题。

5 列出每个关系 （动词）需要的所有数据。

6 列出适用于每个关系的数据 （如果有）。

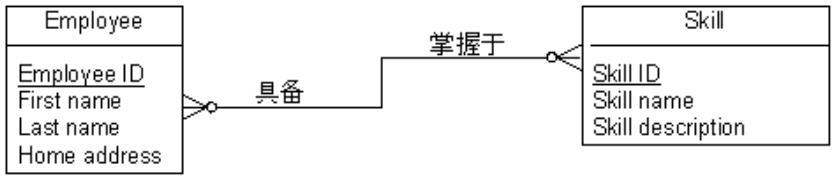
确定支持数据

您所确定的支持数据将成为实体中属性的名称。例如，以下数据可能适用于 Employee 实体、 Skill 实体，和 Expert In 实体。

雇员	技能	擅长于
雇员 ID	技能 ID	技能级别
雇员的名字	技能名称	掌握该项技能的日期

雇员	技能	擅长于
雇员的姓氏	技能说明	
雇员的部门		
雇员所在的办事处		
雇员的地址		

根据这些数据创建的设计图将如下图所示：



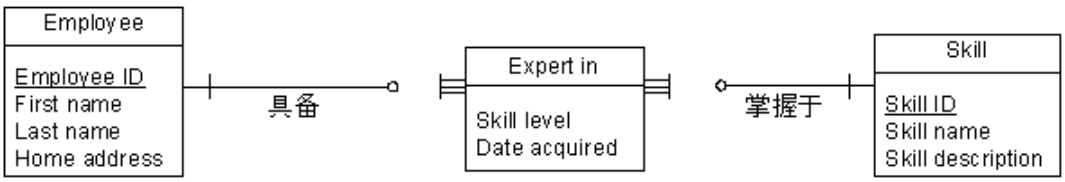
请注意，列出的所有属性并未都在这张设计图中出现。未出现的属性可归为以下两类：

1. 有一些属性隐式包含在其他关系中；例如，雇员的部门和雇员所在的办事处分别用连接 Department 和 Office 实体的关系表示。

2. 其他一些属性未出现的原因是它们与这些实体并不相关联，而是与这些实体间的关系相关联。上图并不完整。

在绘制完整的实体关系图时，第一类属性会自然出现。

可以通过将多对多关系转换为实体来添加第二类属性。



新实体同时依赖于 "Employee" 实体和 "Skill" 实体。它将借用这些实体中的标识符，因为它同时依赖于这两个实体。

注意

- 确定支持数据时，一定要参考前面确定的活动以了解将如何访问这些数据。

  例如，在某些情况下可能需要按雇员的名字列出雇员，而在另一些情况下可能需要按姓氏列出。要满足这两种需要，应创建一个 First Name 属性和一个 Last Name 属性，而不应创建一个既包含名字又包含姓氏的属性。将姓氏和名字分开后，以后可以创建两个索引，分别适用于这两项任务。

- 请选择一致的名称。使用一致的名称可以使数据库便于维护，并且便于阅读报告和输出窗口。

  例如，如果一个属性使用了缩略名称，如 Emp_status，则另一个属性不应使用完整名称，如 Employee_ID。应使名称保持一致，如 Emp_status 和 Emp_ID。

- 在这个阶段，数据是否与正确的实体相关联并不十分重要。您可以根据自己的判断进行设计。在下一节中，将对设计进行测试，检查您的判断是否正确。

第 3 步：规范化数据

规范化是指一系列测试，通过这些测试可以消除冗余的数据，并确保数据与正确的实体或关系相关联。共有五项测试。本节介绍其中前三项测试。这三项测试最重要，因此也最常使用。

为什么要进行规范化？
规范化的目的是消除冗余并提高一致性。例如，如果在多个位置都存储了客户的地址，在这些客户搬家后将很难正确更新所有副本。

有关规范化测试的更多信息，请参见有关数据库设计的书籍。

范式

数据规范化包括几项测试。数据在通过了第一项测试后，我们认为它满足第一范式；通过了第二项测试后，它满足第二范式；通过了第三项测试后，则满足第三范式。

❖ **规范化数据库中的数据：**

1 列出这些数据。

- 为每个实体至少确定一个键。每个实体必须有一个标识符。
- 确定关系的键。关系的键是该关系所连接的两个实体的键。
- 检查支持数据列表中是否有计算数据。计算数据通常不存储在关系数据库中。

2 第一范式测试。

- 如果一个属性在同一个条目上可以有几个不同的值，则删除这些重复的值。
- 用删除的数据创建一个或多个实体或关系。

3 第二范式测试。

- 找出带有多个键的实体和关系。
- 删除只依赖于键的一部分的数据。
- 用删除的数据创建一个或多个实体或关系。

4 第三范式测试。

- 删除依赖于实体或关系中的其他数据，而不依赖于键的数据。
- 用删除的数据创建一个或多个实体或关系。

数据和标识符

开始进行规范化 （对设计进行测试）之前，简单地列出数据，并为每个表确定一个唯一的标识符。标识符可以由一个数据 （属性）或几个数据 （复合标识符）组成。

标识符是唯一地标识实体中各行的一组属性。例如， Employee 实体的标识符是 Employee ID。 Works In 关系的标识符由 Office Code 和 Employee ID 属性组成。

数据库中每个关系的标识符可由该关系所连接的各个实体的标识符组成。在下表中，带有星号的属性为该实体或关系的标识符。

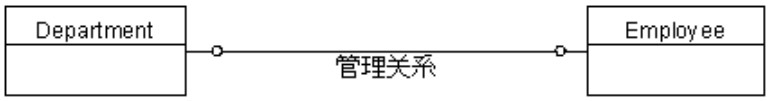
实体或*关系*	**属性**
办事处	* 办事处代码 办事处地址 电话号码
工作地点为	* 办事处代码 * 雇员 ID
部门	* 部门 ID 部门名称
经理为	* 部门 ID * 雇员 ID
属于	* 部门 ID * 雇员 ID
技能	* 技能 ID 技能名称 技能说明
擅长于	* 技能 ID * 雇员 ID 技能级别 掌握日期

实体或关系	**属性**
雇员	* 雇员 ID 姓氏 名字 社保号码 地址 电话号码 出生日期

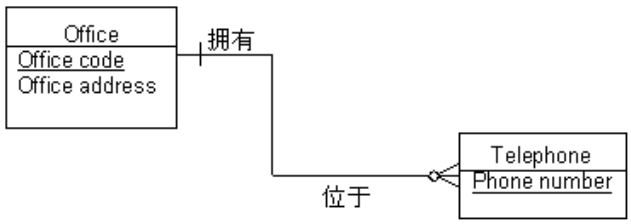
第一范式测试

- 进行第一范式测试时，应找出可能有重复值的属性。
- 如果有多个值都适用于某一项，则删除这些属性。将这些重复属性移到一个新实体中。

在下面的实体中，Phone number 可能会重复 - 一个办事处可以有多个电话号码。



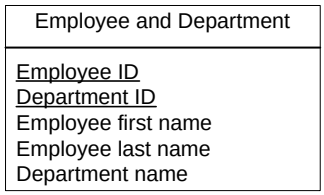
删除重复的属性，并创建一个名为 Telephone 的新实体。在 Office 与 Telephone 之间建立一个关系。



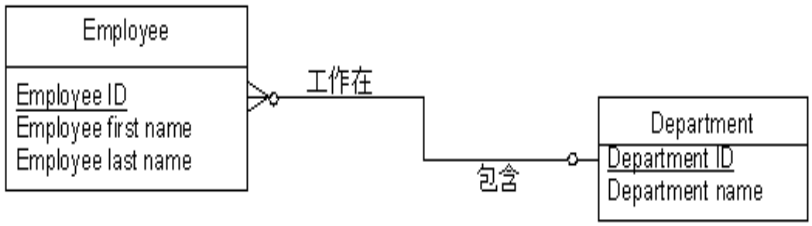
第二范式测试

- 删除不依赖于整个键的值。
- 只需要检查标识符由多个属性组成的实体和关系。进行第二范式测试时，应删除所有不依赖于整个标识符的数据。每个属性都应依赖于组成标识符的所有属性。

在这个示例中， Employee and Department 实体的标识符由两个属性组成。有些数据与这两个标识符属性并不都具有依赖关系，例如，部门名称只依赖于其中一个属性， Department ID，而雇员的名字只依赖于 Employee ID。



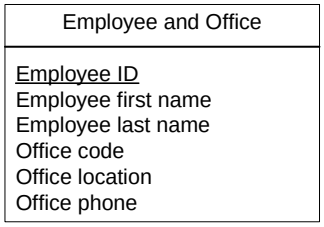
将其他雇员数据不依赖的标识符 Department ID 移到其名为 Department 的实体中。还应移动所有依赖于它的属性。在 Employee 和 Department 之间创建一个关系。



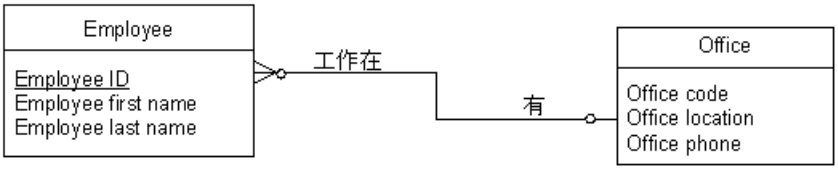
第三范式测试

- 删除不直接依赖于键的数据。
- 进行第三范式测试时，应删除所有依赖于其他属性而不直接依赖于标识符的属性。

在这个示例中， Employee and Office 实体中有一些属性依赖于其标识符 Employee ID。但是， Office location 和 Office phone 等属性却依赖于另一个属性， Office code。这些属性不依赖于标识符 Employee ID。



删除 Office code 和那些依赖于它的属性。创建另一个名为 Office 的实体。然后，创建一个连接 Employee 和 Office 实体的关系。



第 4 步：解析关系

执行完规范化过程后，设计几乎就完成了。唯一还需要做的事情就是生成与概念数据模型相对应的**物理数据模型**。这个过程也称作解析关系，因为其中涉及的大量工作就是将概念模型中的关系转换为相应的表和外键关系。

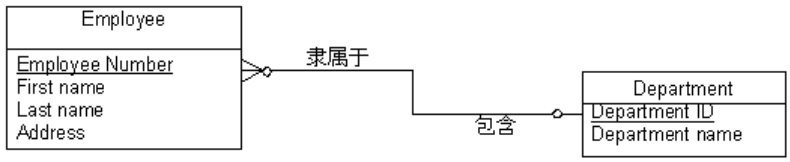
虽然概念模型在很大程度上独立于实施细节，但物理数据模型与特定数据库应用程序中的表结构和可用选项紧密相关。在这里，我们所指的应用程序就是 Adaptive Server Anywhere。

解析不带有数据的关系

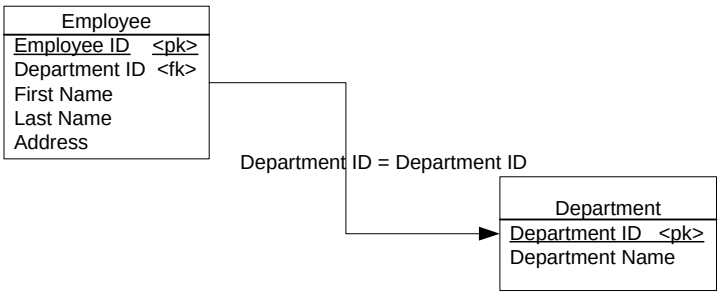
要实施不带有数据的关系，需要定义外键。**外键**是指包含另一个表中的主键值的一列或几列。利用外键一次可以访问多个表中的数据。

使用象 Sybase PowerDesigner 的 DataArchitect 组件的数据库设计工具可以自动生成物理数据模型。但如果要自己生成物理模型，有一些基本规则可以帮助您确定将哪些列设置为键。

- **一对多**　一个一对多关系总是会变为一个实体和一个外键关系。



请注意，实体将变为表。实体中的标识符 （至少一部分）将变为表中的主键。属性将变为列。在一对多关系中，一方的实体中的标识符成为*多*方的表中新加的外键列。

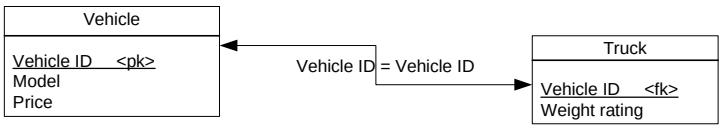


在这个示例中， Employee *实体*变为 Employee *表*。同样，Department 实体变为 Department 表。 Employee 表中将出现一个名为 Department ID 的外键。

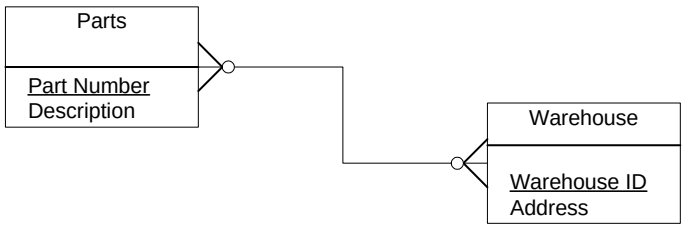
- **一对一**　对于一对一关系，可在其中任意一个表中设置外键。如果关系在一侧是强制的，但在另一侧是可选的，应在可选的一侧设置外键。在这个示例中，在 Truck 表中设置外键 (Vehicle ID) 是因为车辆不一定都是卡车。



因此，上面的实体关系模型将解析为下面的数据库基本结构。

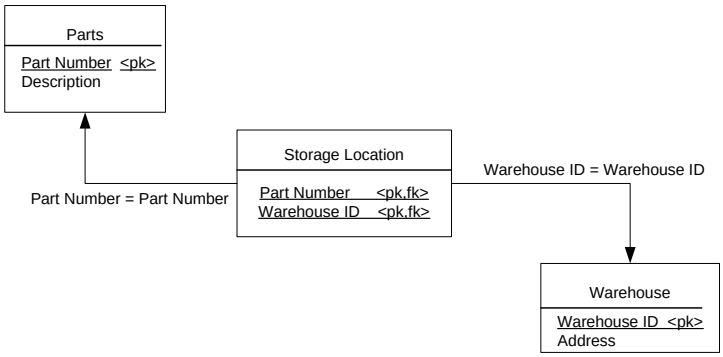


- **多对多**　在多对多关系中，创建的新表中有两个外键。必须这样安排才会使数据库更高效。

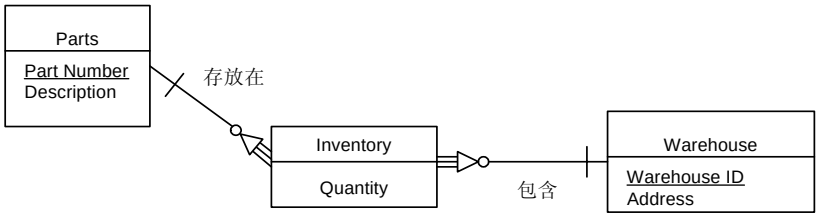


新的 Storage Location 表将 Parts 和 Warehouse 表关联起来。

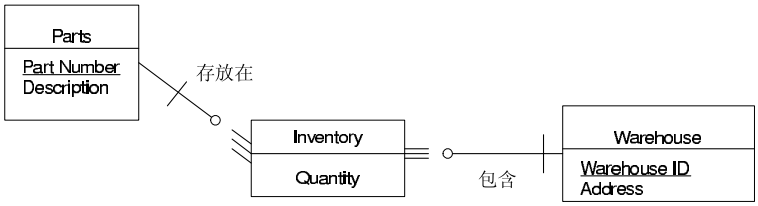
解析带有数据的关系



有些关系可能会带有数据。这种情况常常会在多对多关系中出现。



在这种情况下，每个实体都将解析为一个表。每个角色将变为指向另一个表的外键。



Inventory 实体借用 Parts 和 Warehouse 表中的标识符，因为它与这两个表都有依赖关系。经过解析后，这些借用的标识符将成为 Inventory 表的主键。

提示
概念数据模型可以简化设计过程，因为它将大量细节隐藏起来。例如，多对多关系总会生成一个额外的表和两个外键引用。在概念数据模型中，通常可以用一个连接来标识这类结构。

第 5 步：验证设计

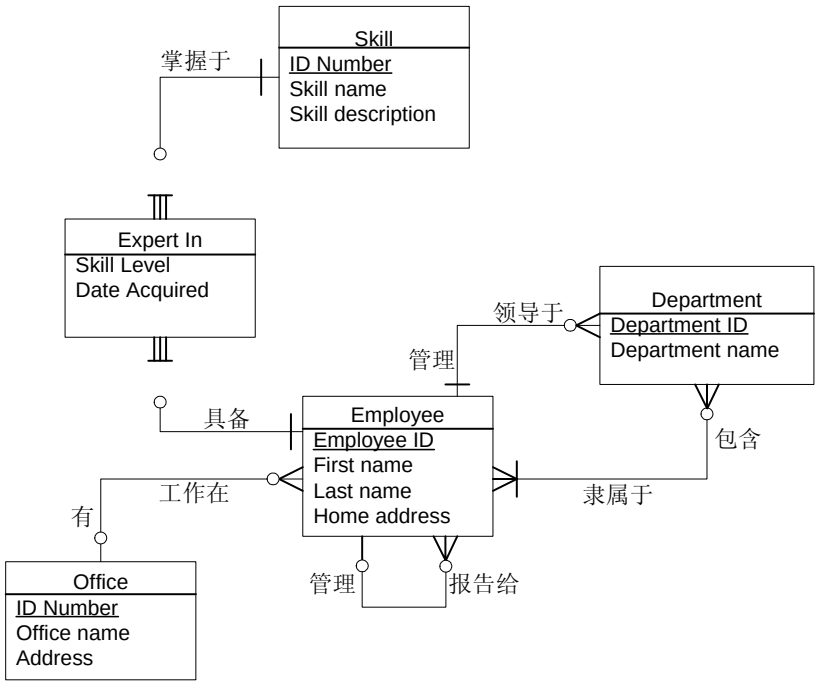
实施设计之前，需要确保该设计能够满足您的需要。检查在设计过程之初确定的活动，确保可以访问这些活动所需要的数据。

- 能否找到一条获取所需信息的途径？
- 设计是否满足您的需求？
- 能否获得所需要的全部数据？

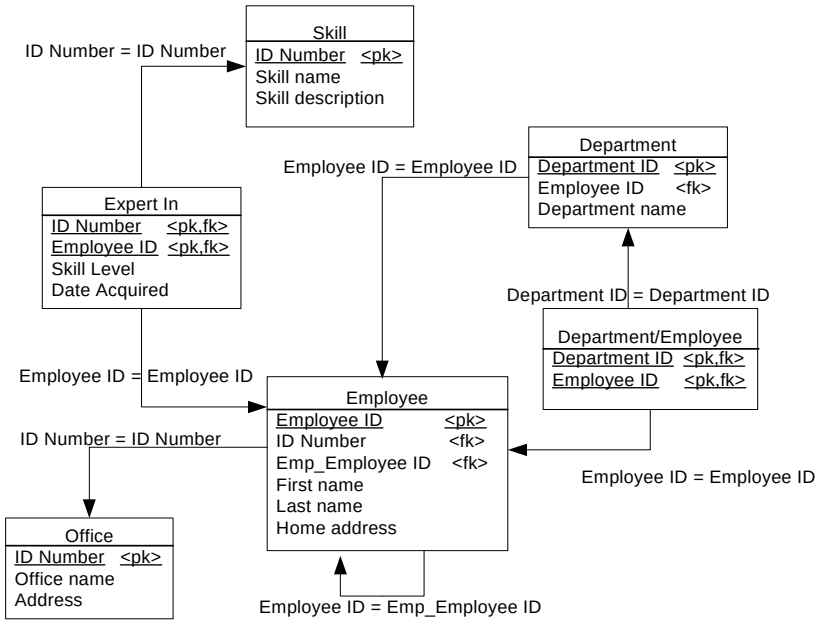
如果您对上述所有问题的回答都是肯定的，那么就可以实现您的设计了。

最终设计

按照第 1 步到第 3 步为前面提到的小公司设计数据库，将得到以下实体关系图。这个数据库现在处于第三范式。



相应的物理数据模型如下所示。


Skill
ID Number <pk>
Skill name
Skill description
ID Number = ID Number
Department
Department ID <pk>
Employee ID <fk>
Department name
Employee ID = Employee ID
Expert In
ID Number <pk,fk>
Employee ID <pk,fk>
Skill Level
Date Acquired
Department ID = Department ID
Department/Employee
Department ID <pk,fk>
Employee ID <pk,fk>
Employee ID = Employee ID
Employee
Employee ID <pk>
ID Number <fk>
Emp_Employee ID <fk>
First name
Last name
Home address
ID Number = ID Number
Employee ID = Employee ID
Office
ID Number <pk>
Office name
Address
Employee ID = Emp_Employee ID

设计数据库表属性

数据库设计将确定需要哪些表，以及每个表中包含哪些列。本节介绍如何指定每个列的属性。

对于每一列，必须确定列名、数据类型和大小、是否允许空值，以及是否希望数据库限制该列中允许包含的值。

选择列名

列名中可以是任何字母、数字或符号的组合。但是，如果列名包含字母、数字或下划线之外的字符，不以字母开头，或者与关键字相同，则必须将列名用双引号引起来。

选择列的数据类型

Adaptive Server Anywhere 中的可用数据类型有：

- 整数数据类型
- 小数数据类型
- 浮点数据类型
- 字符数据类型
- 二进制数据类型
- 日期 / 时间数据类型
- 域 （用户定义的数据类型）

有关数据类型的详细信息，请参见 『ASA SQL 参考』> 「SQL 数据类型」。

长二进制数据类型可用于在数据库中存储诸如图像 （例如，存储为位图）或字处理文档等信息。这些类型的信息通常称为二进制大对象，或 BLOBS。

有关每种数据类型的详细信息，请参见 『ASA SQL 参考』> 「SQL 数据类型」。

NULL 和 NOT NULL

如果记录中要求某列必须有值，应将该列定义为 NOT NULL。否则，该列可以包含 NULL 值，表示没有值。 SQL 中的缺省设置是允许 NULL 值，但如果没有充分的理由来允许 NULL 值，则应将列显式声明为 NOT NULL。

有关 NULL 值的详细信息，请参见 『ASA SQL 参考』> 「NULL 值」。有关在比较中使用 NULL 的信息，请参见 『ASA SQL 参考』> 「搜索条件」。

选择约束

虽然列的数据类型可以限制该列中允许包含的值 （例如，只允许包含数字或只允许包含日期），但可能需要进一步限制允许的值。

可以通过指定 CHECK 约束来限制任何列的值。可以使用可能出现在 WHERE 子句中的任何有效条件来限制所允许的值。大多数 CHECK 约束都使用 BETWEEN 或 IN 条件。

有关有效条件的详细信息，请参见 『ASA SQL 参考』> 「搜索条件」。有关为表和列指派约束的详细信息，请参见 「确保数据完整性」第 85 页。

示例

示例数据库有一个名为 Department 的表，该表具有名为 dept_id、dept_name 和 dept_head_id 的列。它的定义如下所示：

列	数据类型	大小	NULL/NOT NULL	约束
dept_id	integer	-	NOT NULL	无
dept_name	char	40	NOT NULL	无
dept_head_id	integer	-	NULL	无

如果指定 NOT NULL，则必须为该表中的每一行提供值。

第 2 章

使用数据库对象

关于本章

本章将介绍创建、变更和删除数据库对象 （例如表、视图和索引）的方法。

简介

借助于 Adaptive Server Anywhere 工具，您可以创建数据库文件用来保存数据。创建了这种文件之后，您便可以开始管理数据库了。例如，您可以添加数据库对象 （例如表或用户），还可以设置数据库总体属性。

本章将介绍如何创建数据库以及数据库中的对象。其中包括了使用 Sybase Central、 Interactive SQL 和命令行实用程序执行这些任务的过程。如果要在开始之前了解更多概念上的信息，请参见以下章节：

- 「设计数据库」第 3 页
- 「确保数据完整性」第 85 页
- 『SQL Anywhere Studio 介绍』> 「关于 Sybase Central」
- 『SQL Anywhere Studio 介绍』> 「使用 Interactive SQL」

用于执行本章中各项任务的 SQL 语句被称为**数据定义语言** (DDL)。数据库对象的定义构成了数据库模式：您可以将模式看做一个空数据库。

过程和触发器也是数据库对象，但是放在 「使用过程、触发器和批处理」第 699 页中讨论。

本章内容

本章包含以下资料：

- 有关使用数据库对象的简介 （本节）
- 如何创建和使用数据库本身的说明
- 如何创建和变更表、视图和索引的说明

使用数据库

本节将介绍如何创建和使用数据库。在阅读本节时，请记住以下几个简单的概念：

- 您可以创建的数据库 （称为关系数据库）是通过主键和外键相互关联的表的集合。这些表在数据库中保存信息，表和键一起定义数据库的结构。数据库可能存储在一个或多个设备上的一个或多个数据库文件中。
- 数据库文件还包含系统表，随着数据库的建立，系统表保存模式定义。

创建数据库

Adaptive Server Anywhere 提供多种创建数据库的方法：在 Sybase Central 中创建、在 Interactive SQL 中创建和在命令行中创建。创建数据库也被称为初始化数据库。创建完数据库之后，您可以连接到该数据库并在数据库中建立所需的表和其它对象。

事务日志

在创建数据库时，您必须指定放置事务日志的位置。此日志按照进行更改的时间顺序存储对数据库所做的所有更改。如果数据库文件上出现介质故障，则事务日志对进行数据库恢复至关重要。它还可以使您的工作更加有效。缺省情况下，事务日志与数据库文件放在相同的目录中，但对于生产用途不建议这样做。

有关放置事务日志的详细信息，请参见 『ASA 数据库管理指南』> 「针对数据保护来配置数据库」。

数据库文件兼容性

Adaptive Server Anywhere 数据库是一个操作系统文件。就像复制任何其它文件一样，可以将数据库复制到其它位置。

除了有些操作系统存在一些文件系统文件大小限制或 Adaptive Server Anywhere 对大文件的支持限制外，数据库文件可以在所有操作系统间兼容。从任何操作系统创建的数据库都可以从另一操作系统使用，方法是复制数据库文件。同样，通过个人服务器创建的数据库可与网络服务器一起使用。 Adaptive Server Anywhere 服务器可以管理使用该软件的早期版本创建的数据库，但是较旧的服务器不能管理较新的数据库。

有关限制的详细信息，请参见 『ASA 数据库管理指南』> 「大小和数量限制」。

使用其它应用程序创建数据库

某些应用程序设计系统 （例如 Sybase PowerBuilder）包含一些用于创建数据库对象的工具。这些工具通常通过其 ODBC 接口构建可以提交到服务器的 SQL 语句。如果您在使用这些工具中的某一个工具，则不需要构建 SQL 语句就可以完成诸如创建表、指派权限等操作。

本章将介绍用于定义数据库对象的 SQL 语句。如果您使用某种交互式 SQL 工具 （例如 Interactive SQL）来构建数据库，可直接使用这些语句。即使您要使用应用程序设计工具，如果该设计工具不支持某些功能，那么您可能需要使用 SQL 语句将这些功能添加到数据库。

对于更高级的应用，诸如 Sybase PowerDesigner 之类的数据库设计工具可以为开发设计合理的数据库提供更彻底、更可靠的方法。

有关数据库设计的详细信息，请参见 「设计数据库」第 3 页。

创建数据库 (Sybase Central)

您可以在 Sybase Central 中使用 [创建数据库] 实用程序创建数据库。创建完数据库之后，该数据库显示在 Sybase Central 左窗格中其服务器的下面。

有关详细信息，请参见 「创建数据库 (SQL)」第 35 页 和 「创建数据库（命令行）」第 36 页。

❖ 创建新数据库 (Sybase Central)：

1. 选择 [工具] → [Adaptive Server Anywhere 9] → [创建数据库]。
2. 请按照向导中的说明进行操作。

❖ 基于当前连接创建新数据库：

1. 连接到数据库。
2. 在左窗格中，选择 Adaptive Server Anywhere 插件。
3. 在右窗格中，单击 [实用程序] 选项卡。

4　在右窗格中双击 [创建数据库]。

5　请按照向导中的说明进行操作。

为 Windows CE 创建数据库

您可以通过将 Adaptive Server Anywhere 数据库文件复制到 Windows CE 设备来为该设备创建数据库。

Sybase Central 具有某些便于创建 Windows CE 数据库的功能。如果您的 Windows 桌面型计算机上安装有 Windows CE 服务，则当您在 Sybase Central 中创建数据库时，您可以选择创建 Windows CE 数据库。Sybase Central 会去掉 Windows CE 数据库的某些功能，并可以选择将所得到的数据库文件复制到您的 Windows CE 设备。

创建数据库 (SQL)

在 Interactive SQL 中，您可以使用 CREATE DATABASE 语句创建数据库。在使用此语句之前，您需要连接到一个现有的数据库。

❖ 创建新数据库 (SQL)：

1　连接到一个现有的数据库。

2　执行 CREATE DATABASE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE DATABASE 语句」。

示例

在 *c:\temp* 目录中创建一个数据库名为 *temp.db* 的数据库文件。

```
CREATE DATABASE 'c:\\temp\\temp.db'
```

该目录路径是相对于数据库服务器的路径。您可以在服务器命令行上使用 -gu 命令行选项设置执行此语句所需的权限。缺省设置需要 DBA 权限。

反斜线在 SQL 中是转义字符，必须成对使用。有关详细信息，请参见 『ASA SQL 参考』> 「字符串」。

创建数据库 （命令行）

您可以从命令行使用 *dbinit* 实用程序创建数据库。通过此实用程序，您可以使用命令行参数来为数据库指定不同的选项。

❖ **创建新数据库 （命令行）：**

1 打开命令提示符。

2 运行 *dbinit* 实用程序。并添加任何所需的参数。

例如，若要创建 4 Kb 页面大小的名为 *company.db* 的数据库，请键入：

```
dbinit -p 4096 company.db
```

有关详细信息，请参见 『ASA 数据库管理指南』> 「初始化实用程序」。

消除数据库

消除数据库之后，所有表和数据都将从磁盘中删除，其中包括记录数据库变更的事务日志。所有数据库文件都是只读的，以防止数据库文件被意外修改或删除。

在 Sybase Central 中，您可以使用 [消除数据库] 实用程序消除数据库。您需要连接到一个数据库以实现此功能，但 [消除数据库] 向导允许您指定任何要消除的数据库。为了消除未运行的数据库，数据库服务器必须处于运行状态。

在 Interactive SQL 中，您可以使用 DROP DATABASE 语句消除数据库。可以使用数据库服务器 `-gu` 命令行选项设置所需的权限。缺省设置为要求具有 DBA 权限。

您还可以从命令行使用 *dberase* 实用程序消除数据库。使用此实用程序时，要清除的数据库一定不能正在运行。

❖ **消除数据库 (Sybase Central)：**

1 在左窗格中，选择 Adaptive Server Anywhere 插件。

2　在右窗格中，单击 [实用程序] 选项卡。

3　在右窗格中双击 [消除数据库]。

4　请按照向导中的说明进行操作。

❖ **消除数据库 (SQL)：**

- 执行 DROP DATABASE 语句。

  例如，

  ```
  DROP DATABASE 'c:\\temp\\temp.db'
  ```

❖ **消除数据库 （命令行）：**

- 从命令行运行 *dberase* 实用程序。

  例如，

  ```
  dberase company.db
  ```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP DATABASE 语句」和 『ASA 数据库管理指南』> 「清除实用程序」。

从数据库断开连接

当使用完数据库之后，您可以从数据库断开连接。 Adaptive Server Anywhere 还允许您断开其他用户与给定数据库的连接；有关在 Sybase Central 中执行此操作的详细信息，请参见 『ASA 数据库管理指南』> 「管理连接的用户」。

数据库服务器启动时，会给每个与该服务器建立的新连接指派一个唯一的连接 ID。您可以通过使用 **connection_property** 函数请求连接号来获得用户的 *connection-id*。下面的语句返回当前连接的连接 ID：

```
SELECT connection_property( 'number' )
```

❖ **从数据库断开连接 (Sybase Central)：**

1 选择一个数据库。

2 选择 [工具] → [断开连接]。

❖ **从数据库断开连接 (SQL)：**

- 执行 DISCONNECT 语句。

示例 1 下面的语句显示如何从 Interactive SQL 中使用 DISCONNECT 断开所有连接：

```
DISCONNECT ALL
```

示例 2 下面的语句显示如何在嵌入式 SQL 中使用 DISCONNECT：

```
EXEC SQL DISCONNECT :conn_name
```

❖ **断开其他用户与数据库的连接 (SQL)：**

1 使用 DBA 权限连接到一个现有的数据库。

2 执行 DROP CONNECTION 语句。

示例 下面的语句删除 ID 号为 4 的连接。

```
DROP CONNECTION 4
```

有关详细信息，请参见 『ASA SQL 参考』> 「DISCONNECT 语句 [ESQL] [Interactive SQL]」和 『ASA SQL 参考』> 「DROP CONNECTION 语句」。

设置数据库对象的属性

大多数数据库对象 （包括数据库本身）都具有您可以查看或设置的属性。某些属性是不可配置的，它们反映您在创建数据库或对象时选择的设置。其它属性是可配置的。

查看和设置属性的最佳方式是使用 Sybase Central 中的属性表。

如果您未使用 Sybase Central，则可以在使用 CREATE 语句创建对象时指定属性。如果该对象已存在，则可以使用 SET OPTION 语句修改选项。

❖ 查看和编辑数据库对象的属性 (Sybase Central)：

1. 在 Sybase Central 的左窗格中，打开对象所在的文件夹。
2. 选择该对象。该对象的属性会出现在 Sybase Central 的右窗格中。
3. 在右窗格中，单击相应的选项卡来编辑所需的属性。

   您也可以在对象的属性表上查看和编辑属性。要查看属性表，请右击对象，然后从弹出式菜单中选择 [属性]。

设置数据库选项

数据库选项是可配置的设置，这些设置可以改变数据库运转或执行的方式。在 Sybase Central 中，所有这些选项都集中在 [数据库选项] 对话框中。对于 Interactive SQL，您可以在 SET OPTION 语句中指定选项。

❖ 设置数据库的选项 (Sybase Central)：

1. 打开所需的服务器。
2. 右击所需的数据库，然后从弹出式菜单中选择 [选项]。
3. 编辑所需的值。

❖ 设置数据库的选项 (SQL)：

- 在 SET OPTION 语句中指定所需属性。

提示

您还可以使用 [数据库选项] 对话框为特定的用户和组设置数据库选项（当您为用户或组打开此对话框时，该对话框分别叫做 [用户选项] 对话框或 [组选项] 对话框。）

当您为数据库本身设置选项时，您实际是在为该数据库中的 PUBLIC 组设置选项，这是因为所有用户和组都从 PUBLIC 继承选项设置。

指定统一数据库

在 Sybase Central 中，您可以为 [SQL Remote 复制] 指定统一数据库。统一数据库是在复制系统中充当 [主] 数据库的那个数据库。统一数据库包含所有要复制的数据，而其远程数据库可能只包含它们自己的数据子集。一旦出现冲突或不一致，统一数据库将被认为具有所有数据的主副本。

有关详细信息，请参见 『SQL Anywhere Studio 介绍』> 「统一数据库和远程数据库」。

❖ 设置统一数据库 (Sybase Central)：

1. 右击所需的数据库，然后从弹出式菜单中选择 [属性]。
2. 单击 [SQL Remote] 选项卡。
3. 选择 [此远程数据库具有对应的统一数据库] 选项。
4. 配置所需的设置。

显示数据库中的系统对象

在数据库中，表、视图、存储过程或域都是系统对象。系统表存储有关数据库本身的信息，而系统视图、过程和域在很大程度上可以支持 Sybase Transact-SQL 兼容性。

在 Interactive SQL 中，您不能显示所有系统对象的列表，但可以浏览系统表的内容；有关详细信息，请参见 「显示系统表」第 58 页。

❖ 显示数据库中的系统对象 (Sybase Central)：

1. 打开所需的服务器。

2　右击所需的已连接数据库，然后选择 [按所有者过滤对象]。

3　选择 **SYS** 和 **dbo**，然后单击 [确定]。

系统表、系统视图、系统过程和系统域出现在它们各自的文件夹中。例如，系统表与普通表一起出现在 [表] 文件夹中。

有关详细信息，请参见 『ASA SQL 参考』> 「系统表」。

在您使用数据库时记录 SQL 语句

当您在 Sybase Central 中使用数据库时，应用程序将根据您的操作自动生成 SQL 语句。您可以在一个单独的名为 [设计详细信息] 的窗格中跟踪这些语句，或者将信息保存到文件中。每个数据库和数据库服务器在 [设计详细信息] 窗格中都有对应的选项卡。数据库服务器的选项卡与 [服务器消息] 窗口包含的信息是相同的。

您在使用 Interactive SQL 时，还可以记录您执行的语句。

有关在 Interactive SQL 中记录语句的信息，请参见 『SQL Anywhere Studio 介绍』> 「记录命令」。

❖ 记录 Sybase Central 生成的 SQL 语句：

1　从 [视图] 菜单中选择 [设计详细信息]。

在 Sybase Central 窗口的底部会出现 [设计详细信息] 窗格。

2　右击 [设计详细信息] 窗格，然后从弹出式菜单中选择 [选项]。

即会出现 [选项] 对话框。

3　在随之出现的对话框中，指定所需的设置。如果您要将记录的信息保存到文件，请单击 [保存] 按钮。

启动未连接的数据库

使用 Sybase Central 和 Interactive SQL，您都可以在不连接数据库的情况下启动该数据库。

❖ **在不进行连接的情况下启动服务器上的数据库 (Sybase Central)：**

1 右击所需的服务器，然后从弹出式菜单中选择 [启动数据库]。

2 在 [启动数据库] 对话框中，键入所需的值。

该数据库作为断开连接的数据库出现在服务器下面。

❖ **在不进行连接的情况下启动服务器上的数据库 (SQL)：**

1 启动 Interactive SQL。

2 执行 START DATABASE 语句。

示例 在服务器 sample 上，启动数据库文件 *c:\asa9\sample_2.db* 作为 sam2。

```
START DATABASE 'c:\asa9\sample_2.db'
 AS sam2
 ON sample
```

有关详细信息，请参见 『ASA SQL 参考』> 「START DATABASE 语句 [Interactive SQL]」。

将 jConnect 元数据支持安装到现有的数据库

如果创建数据库时没有安装 jConnect 元数据支持，则您可以以后使用 Sybase Central 安装该支持。

❖ **将 jConnect 元数据支持添加到现有的数据库 (Sybase Central)：**

1 连接到要升级的数据库。

2　在左窗格中，选择 Adaptive Server Anywhere 插件。

3　在右窗格中，单击 [实用程序] 选项卡。

4　在右窗格中，双击 [升级数据库]。

5　单击向导简介页上的 [下一步]。

6　从列表中选择要升级的数据库。单击 [下一步]。

7　如果需要，可选择创建数据库的备份。单击 [下一步]。

8　选择 [安装 jConnect 元信息支持] 选项。

9　请按照向导中的其余说明进行操作。

使用表

首次创建数据库时，数据库中仅有的表就是系统表。系统表中保存了数据库模式。

本节将介绍如何创建、变更和删除表。您可以在 Interactive SQL 中执行示例，但 SQL 语句不依赖于您使用的管理工具。

有关详细信息，请参见『SQL Anywhere Studio 介绍』>「在 Interactive SQL 中编辑表值」。

为了在必要时更加方便地重新创建数据库模式，请创建命令文件以定义数据库中的表。命令文件应该包含 CREATE TABLE 和 ALTER TABLE 语句。

有关组、表和作为另一个用户连接的详细信息，请参见『ASA 数据库管理指南』>「引用组拥有的表」和『ASA 数据库管理指南』>「数据库对象名和前缀」。

创建表

首次创建数据库时，数据库中仅有的表就是系统表，系统表中保存了数据库模式。随后，您可以在 Interactive SQL 中使用 SQL 语句或通过 Sybase Central 创建新表以保存实际数据。

您可以创建两种类型的表：

- **基表** 保存持久性数据的表。在您显式删除数据或删除该表之前，基表及其数据都一直存在。被称之为基表是为了和临时表及视图区分开来。
- **临时表** 临时表中的数据只为一个连接而保存。全局临时表定义（但不包括数据）一直保存在数据库中直到被删除。局部临时表定义和数据的存在时间仅为一个连接的持续时间。

  有关临时表的详细信息，请参见 「使用临时表」第 82 页。

表由行和列组成。每一列保存的是一类特定的信息 （例如电话号码或名称），而每一行指定的是特定的条目。

❖ 创建表 (Sybase Central)：

1　连接到数据库。

2　打开 [表] 文件夹。

3　从 [文件] 菜单中，选择 [新建] → [表]。

即会出现 [创建表] 向导。

4　在 [创建表] 向导中：

- 为新表键入一个名称。
- 为该表选择一个所有者。
- 为该表创建主键。
- 配置所需的其它选项。

5　单击 [完成]。

6　在右窗格的 [列] 选项卡上，您可以给该表添加列。

7　在完成时选择 [文件] → [保存]。

❖ 创建表 (SQL)：

1　使用 DBA 权限连接到数据库。

2　执行 CREATE TABLE 语句。

示例

以下语句可创建一个新表来描述公司中雇员的资格。表中的列分别保存标识号、技能名和每种技能的类型 （技术或管理）。

```
CREATE TABLE skill (
     skill_id INTEGER NOT NULL,
     skill_name CHAR( 20 ) NOT NULL,
     skill_type CHAR( 20 ) NOT NULL
 )
```

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

变更表

本节将介绍如何更改表的结构或列定义。例如，您可以添加列、更改不同的列属性或彻底删除列。

在 Sybase Central 中，您可以在 Sybase Central 右窗格的 SQL 选项卡上执行这些任务。在 Interactive SQL 中，您可以使用 ALTER TABLE 语句执行这些任务。

如果使用 Sybase Central，则当您在 [列] 文件夹中选中某个列时，您还可以使用菜单命令来管理列 （在主键中添加或删除列、更改它们的属性或删除列）。

有关变更数据库对象属性的信息，请参见 「设置数据库对象的属性」第 38 页。

有关授予和撤消表权限的信息，请参见 『ASA 数据库管理指南』> 「授予针对表的权限」和 『ASA 数据库管理指南』> 「撤消用户权限」。

变更表 (Sybase Central)

您可以在 Sybase Central 右窗格的 [列] 选项卡上变更表。例如，您可以添加或删除列、更改列定义或更改表或列属性。

❖ 变更现有的表 (Sybase Central)：

1. 连接到数据库。
2. 打开 [表] 文件夹。
3. 在左窗格中，选择要变更的表。
4. 在右窗格中，单击 [列] 选项卡，然后进行所需的更改。

提示
您可以添加列。方法是：选择表的 [列] 选项卡，然后选择 [文件] → [新建列]。

您可以删除列。方法是：在 [列] 选项卡上选中列，然后选择 [编辑] → [删除]。

您可以将列复制到表。方法是：在右窗格的 [列] 选项卡上选中列，然后单击 [复制] 按钮。选择所需的表，单击右窗格的 [列] 选项卡，然后单击 [粘贴] 按钮。

您必须单击 [保存表] 按钮或选择 [文件] → [保存表]。只有在执行此操作后，所做的更改才会在表中生效。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」。

变更表 (SQL)

您可以在 Interactive SQL 中使用 ALTER TABLE 语句变更表。

❖ 变更现有的表 (SQL)：

1　使用 DBA 权限连接到数据库。

2　执行 ALTER TABLE 语句。

示例

以下命令将一列添加到 skill 表中，以便用于输入技能的可选说明：

```
ALTER TABLE skill
 ADD skill_description CHAR( 254 )
```

此语句添加一个名为 skill_description 的列，该列最多可以保存几个句子用于说明该技能。

您还可以使用 ALTER TABLE 语句修改列属性。以下语句将示例数据库的 skill_description 列从最多 254 个字符缩短为最多 80 个字符：

```
ALTER TABLE skill
 MODIFY skill_description CHAR( 80 )
```

任何超过 80 个字符的当前输入都将被剪裁以符合 80 个字符的限制，并显示一条警告。

以下语句将 skill_type 列的名称更改为 classification：

```
ALTER TABLE skill
 RENAME skill_type TO classification
```

以下语句删除 classification 列。

```
ALTER TABLE skill
 DROP classification
```

以下语句更改整个表的名称：

```
ALTER TABLE skill
 RENAME qualification
```

这些示例显示了如何更改数据库的结构。 ALTER TABLE 语句几乎可以更改有关表的所有内容 — 您可以使用它添加或删除外键、将列从一种类型更改为另一种类型等。在所有这些情况下，只要进行了更改，则存储过程、视图和引用该表的任何其它项都将无法工作。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」和 「确保数据完整性」第 85 页。

删除表

本节将介绍如何从数据库中删除表。您可以使用 Sybase Central 或 Interactive SQL 执行此任务。在 Interactive SQL 中，删除 (Delete) 表也称作将其删除 (Drop)。

您不能删除正被用作 SQL Remote 发布中项目的表。如果在 Sybase Central 中试图执行此操作，则将出现错误。

❖ 删除表 (Sybase Central)：

1. 连接到数据库。
2. 打开该数据库的 [表] 文件夹。
3. 右击表，然后从弹出式菜单中选择 [删除]。

❖ **删除表 (SQL)：**

1　使用 DBA 权限连接到数据库。

2　执行 DROP TABLE 语句。

示例

以下 DROP TABLE 命令删除 skill 表中的所有记录，然后从数据库中删除 skill 表的定义

```
DROP TABLE skill
```

与 CREATE 语句类似， DROP 语句在删除表前后自动执行 COMMIT 语句。这就会使上次执行 COMMIT 或 ROLLBACK 之后对数据库的所有更改成为永久更改。 DROP 语句还删除表上的所有索引。

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

浏览表中的信息

您可以使用 Sybase Central 或 Interactive SQL 浏览保存在数据库的表中的数据。

如果您在使用 Sybase Central，则可以通过在右窗格中单击 [数据] 选项卡来查看表中的数据。

如果您在使用 Interactive SQL，则执行以下语句：

```
SELECT * FROM table-name
```

您可以通过 Interactive SQL [结果] 选项卡或通过 Sybase Central 编辑表中的数据。

管理主键

主键是由一列或多列组合组成的唯一标识符，主键的值在行中数据的生存期内不发生更改。因为唯一性对良好的数据库设计至关重要，所以在定义表时最好指定主键。

本节将介绍如何在数据库中创建和编辑主键。您可以使用 Sybase Central 或 Interactive SQL 来执行这些任务。

多列主键中的列顺序
主键列顺序是由创建表的过程中列的顺序决定的。它不依赖于主键声明中指定的列的顺序。

管理主键 (Sybase Central)

在 Sybase Central 中，表的主键在多处显示：

- 在表属性表的 [列] 选项卡上。
- 在右窗格的 [列] 选项卡上 （前提是已在左窗格中选中了表）。

表属性表和 [列] 选项卡上的列表都会显示主键列 （同时还有非主键列）。主键列的顺序由创建表期间的列顺序决定。这可能和主键中的列的实际排序有所不同。

❖ 创建和编辑主键 (Sybase Central)

1. 打开 [表] 文件夹。
2. 在左窗格中，选择所需的表。
3. 在右窗格中，单击 [列] 选项卡。
4. 按需要更改主键。

   要让某列成为主键列，请在该列名称旁边的复选框中放置复选标记。清除复选框可从主键中删除该列。

管理主键 (SQL)

您可以在 Interactive SQL 中使用 CREATE TABLE 和 ALTER TABLE 语句创建和修改主键。这些语句可用于设置多个表属性，包括列约束和检查。

主键中的列不能包含空值。您必须在主键中的列上指定 NOT NULL。

❖ **修改现有表的主键 (SQL)：**

1　使用 DBA 权限连接到数据库。

2　执行 ALTER TABLE 语句。

示例 1

以下语句创建与以前相同的 skill 表，唯一不同之处是它添加了一个主键：

```
CREATE TABLE skill (
   skill_id INTEGER NOT NULL,
   skill_name CHAR( 20 ) NOT NULL,
   skill_type CHAR( 20 ) NOT NULL,
   primary key( skill_id )
 )
```

该表中每一行的主键值必须是唯一的，这就意味着您不能给多行指定同一个 skill_id。表中的每一行由它的主键值唯一地标识。

示例 2

以下语句将列 skill_id 和 skill_type 添加到 skill 表的主键中：

```
ALTER TABLE skill
   ADD PRIMARY KEY ( "skill_id", "skill_type" )
```

如果在 ALTER TABLE 语句中指定了 PRIMARY KEY 子句，则该表不能已具有由 CREATE TABLE 语句或另一个 ALTER TABLE 语句创建的主键。

示例 3

以下语句从 skill 表的主键中删除所有列。在删除主键之前，一定要搞清楚该操作将对数据库带来什么后果。

```
ALTER TABLE skill
   DELETE PRIMARY KEY
```

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」和 「管理主键 (Sybase Central)」第 50 页。

管理外键

本节将介绍如何在数据库中创建和编辑外键。您可以使用 Sybase Central 或 Interactive SQL 来执行这些任务。

外键的作用是将子表 （即外表）中的值与父表 （即主表）中的值关联。一个表可能具有引用多个父表的多个外键，从而链接多种类型的信息。

管理外键 (Sybase Central)

在 Sybase Central 中，表的外键显示在 [外键] 选项卡上 （在选中表时，该选项卡位于右窗格中）。

如果外表列的值无法匹配主表主键中的值，则您不能在该表中创建外键。

在创建了外键之后，您可以在每个表位于右窗格的 [引用表] 中跟踪外键；此文件夹显示引用当前所选表的所有外表。

❖ 在给定表中创建新外键 (Sybase Central)：

1 选择要为其创建外键的表。

2 单击右窗格中的 [外键] 选项卡。

3 从 [文件] 菜单中，选择 [新建] > [外键]。

即会出现 [创建外键] 向导。

4 请按照向导中的说明进行操作。

❖ 删除外键 (Sybase Central)：

1 选择要为其删除外键的表。

2 单击右窗格中的 [外键] 选项卡。

3 右击您要删除的外键，然后从弹出式菜单中选择 [删除]。

❖ 显示具有引用了给定表的外键的所有表 (Sybase Central)：

1 在左窗格中，选择所需的表。

2 在右窗格中，单击 [引用表] 选项卡。

提示
在使用向导创建外键时，您可以设置该外键的属性。要在创建外键之后查看属性，请在 [外键] 选项卡上右击外键，然后从弹出式菜单中选择 [属性]。

您可以查看引用表的属性，方法是在 [引用表] 选项卡上右击该表，然后从弹出式菜单中选择 [属性]。

管理外键 (SQL)

您可以在 Interactive SQL 中使用 CREATE TABLE 和 ALTER TABLE 语句创建和修改外键。这些语句可用于设置多个表属性，包括列约束和检查。

一个表只能定义一个主键，但它可以按需要具有多个外键。

❖ 修改现有表的外键 (SQL)：

1　使用 DBA 权限连接到数据库。

2　执行 ALTER TABLE 语句。

示例 1

您可以创建名为 emp_skill 的表，该表保存每个雇员的每项合格技能的技能等级说明，如下所示：

```
CREATE TABLE emp_skill(
     emp_id INTEGER NOT NULL,
     skill_id INTEGER NOT NULL,
     "skill level" INTEGER NOT NULL,
     PRIMARY KEY( emp_id, skill_id ),
     FOREIGN KEY REFERENCES employee,
     FOREIGN KEY REFERENCES skill
 )
```

emp_skill 表定义具有一个由两列 （emp_id 列和 skill_id 列）组成的主键。一个雇员可能有多项技能，因此会显示在多个行上，同时，多个雇员也可能都具有某一给定技能，因此 skill_id 列也可能出现多次。但是，对于某一给定雇员和技能的组合则只有一个条目。

emp_skill 表还具有两个外键。外键条目指示 emp_id 列必须包含 employee 表中有效的雇员号， skill_id 列必须包含 skill 表中的有效条目。

示例 2

您可以将名为 foreignkey 的外键添加到现有的 skill 表，并让此外键引用 contact 表中的主键，如下所示：

```
ALTER TABLE skill
 ADD FOREIGN KEY "foreignkey" ("skill_id")
 REFERENCES "DBA"."contact" ("id")
```

此示例在表 skill （外表）的 skill_id 列和表 contact （主表）的 id 列之间建立关系。 [DBA] 表示 contact 表的所有者。

示例 3

您可以在创建外键时指定外键的属性。例如，以下语句创建的外键与示例 2 中创建的相同，但它将外键定义为 NOT NULL，同时定义更新或删除时的限制。

```
ALTER TABLE skill
 ADD NOT NULL FOREIGN KEY "foreignkey" ("skill_id")
 REFERENCES "DBA"."contact" ("id")
 ON UPDATE RESTRICT
 ON DELETE RESTRICT
```

在 Sybase Central 中，您还可以在 [创建外键] 向导或外键的属性表中指定属性。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」和 「管理外键 (Sybase Central)」第 52 页。

使用计算列

计算列是一个以表达式作为值的列，该表达式可以引用同一行中其它列（被称为相关列）的值。如果您要为一个复杂表达式创建索引，该表达式可能包含一个或多个相关列的值，计算列就特别有用。在查询优化过程中， Adaptive Server Anywhere 优化程序会自动尝试将涉及复杂表达式的谓语转换为只是引用计算列的定义的谓语。例如，假定您要查询包含有关产品发货摘要信息的表：

```
CREATE TABLE SHIPMENTS(
    SHIPMENT_ID   INTEGER NOT NULL PRIMARY KEY,
    SHIPMENT_DATE TIMESTAMP,
    PRODUCT_CODE  CHAR(20) NOT NULL,
    QUANTITY      INTEGER NOT NULL,
    TOTAL_PRICE   DECIMAL(10,2) NOT NULL )
```

特别是，该查询将返回平均成本在两美元到四美元之间的发货批次。您可以如下编写这一查询

```
SELECT *
 FROM SHIPMENTS
 WHERE ( TOTAL_PRICE / QUANTITY ) BETWEEN 2.00 AND 4.00
```

但是， WHERE 子句中的谓语不是可优化的，因为它不引用单个基列。如果 SHIPMENTS 表相对较大，可能更适合用索引检索而不是顺序扫描。为此，请为 SHIPMENTS 表创建计算列 AVERAGE_COST，如下所示：

```
ALTER TABLE shipments
 ADD average_cost DECIMAL(30,22)
 COMPUTE (total_price / quantity)

 CREATE INDEX idx_avg_cost ON shipments( average_cost
ASC )
```

选择计算列的类型是很重要的；如果查询中的表达式数据类型与计算列的数据类型精确匹配， Adaptive Server Anywhere 优化程序只将复杂的表达式替换为计算列。要确定表达式的类型，您可以使用内置函数 EXPRTYPE()，它会用现成的 SQL 术语返回表达式的类型：

```
SELECT EXPRTYPE( 'SELECT (TOTAL_PRICE / QUANTITY ) AS X
FROM
 SHIPMENTS', 1 )
 FROM DUMMY
```

对于 SHIPMENTS 表，上述查询会返回 **NUMERIC(30,22)**。在优化过程中， Adaptive Server Anywhere 优化程序会将上述查询重写为

```
SELECT *
 FROM SHIPMENTS
 WHERE AVERAGE_COST BETWEEN 2.00 AND 4.00
```

并且， WHERE 子句中的谓语现在是可优化的，这样，优化程序就能使用新的 idx_avg_cost 索引为该查询的访问计划选择带索引的扫描。在插入和更新行时，由数据库服务器自动维护计算列的值。大多数应用程序不需要直接更新或插入计算列的值；不过，由于计算列与其它列一样也是基列，因此，必要时可以在谓语和表达式中直接引用计算列。

虽然您可以使用 INSERT、 UPDATE 或 LOAD TABLE 语句在计算列中插入值，但这既不是该功能的推荐用法，也不是该功能的预期用法。 LOAD TABLE 语句允许对计算列进行 *可选* 计算，这可以在一系列复杂的卸载 / 重装过程中协助 DBA，或者，如果在 COMPUTE 表达式引用不确定的值 （如 CURRENT TIMESTAMP）时必须让计算列的值保持不变，也可以使用 LOAD TABLE 语句。

修改计算列表达式

您可以使用 ALTER TABLE 语句更改计算列中使用的表达式。下面的语句更改计算列所基于的表达式。

```
ALTER TABLE table_name
 ALTER column-name SET COMPUTE ( new-expression )
```

在执行上述语句时，将重新计算该列。如果新表达式无效，则 ALTER TABLE 语句失败。

下面的语句使某个列不再成为计算列。

```
ALTER TABLE table_name
 ALTER column-name DROP COMPUTE
```

执行此语句后，该列中现有的值不会更改，但它们不再自动更新。

插入和更新计算列

计算列限制允许的 INSERT 和 UPDATE 语句。

- **不要直接插入或更新**　不能直接将值插入到计算列中。如果名为 Computed 的列是计算列，下面的语句则会出现重复的插入列错误：

  ```
  -- Incorrect statement
   INSERT INTO T1 (id, computed)
   VALUES( 3006, 'bad insert statement' )
  ```

  同样，您不能使用 UPDATE 语句直接更改计算列的值。

- **列出列名**　您必须始终在 INSERT 语句中显式指定具有计算列的表的列名。

- **触发器**　如果对计算列定义触发器，任何影响该列的 INSERT 或 UPDATE 语句都将触发触发器。

何时对计算列重新进行计算

计算列在出现以下情况时会重新计算：

- 删除、添加或重命名了列。
- 重命名了表。
- 修改了任何列的数据类型或 COMPUTE 子句。
- 插入了行。
- 更新了行。

查询计算列时*不会*对它进行重新计算。如果您使用与时间相关的表达式，或者使用一个在其它某方面依赖数据库状态的表达式，则计算列可能无法给出正确的结果。

在数据库中 / 之间复制表或列

使用 Sybase Central，您可以复制现有的表或列并将它们插入同一个数据库中的另一个位置或完全不同的数据库中。

如果未使用 Sybase Central，请参见以下内容之一：

- 要将 SELECT 语句结果插入给定的位置，请参见 『ASA SQL 参考』> 「SELECT 语句」。
- 要将一行或选择的多行从数据库的其它位置插入表中，请参见 『ASA SQL 参考』> 「INSERT 语句」。

显示系统表

在数据库中，表、视图、存储过程或域都是系统对象。**系统表**存储数据库的模式，即有关数据库本身的信息。系统视图、过程和域在很大程度上可以 Sybase Transact-SQL 兼容性。

有关数据库中表的所有信息都显示在系统表中。这些信息分布在多个表中。

有关详细信息，请参见 『ASA SQL 参考』> 「系统表」。

❖ 显示系统表 (Sybase Central)：

1. 右击所需的已连接数据库，然后从弹出式菜单中选择 [按所有者过滤对象]。
2. 选择 [SYS] 旁边的复选框，然后单击 [确定]。

   系统表、系统视图、系统过程和系统域出现在它们各自的文件夹中。例如，系统表与普通表一起出现在 [表] 文件夹中。

❖ 浏览系统表 (SQL)：

1. 连接到数据库。
2. 执行 SELECT 语句，指定您要浏览的系统表。 SYS 用户 ID 拥有系统表。

示例

在 [结果] 窗格的 [结果] 选项卡上显示表 sys.systable 的内容。

```
SELECT *
 FROM SYS.SYSTABLE
```

使用视图

视图是计算出的表。您可以使用视图以一种可以控制的格式向数据库用户显示所要提供的信息。

视图和基表之间的相似之处

视图和数据库的永久表 （永久表也称为基表）在多个方面具有相似之处：

- 您可以为视图指派访问权限，就像为基表指派访问权限一样。
- 您可以在视图上执行 SELECT 查询。
- 您可以在某些视图上执行 UPDATE、 INSERT 和 DELETE 操作。
- 您可以基于某些视图创建另一些视图。

视图和永久表之间的差异

视图和永久表之间存在一些差异：

- 不能创建视图索引。
- 在所有视图上您都不能执行 UPDATE、 INSERT 和 DELETE 操作。
- 您不能将完整性约束和键指派给视图。
- 视图引用基表中的信息，但不保留这些信息的副本。您每次调用视图时都重新计算视图。

定制访问的好处

视图允许您定制对数据库中数据的访问。定制访问可以有以下几个好处：

- **提高安全性**　因为可以只允许访问相关的信息。
- **提高可用性**　因为可以以一种比基表中的数据更易于理解的格式向用户和应用程序开发人员提供数据。
- **提高一致性**　因为可以在数据库中集中一般查询的定义。

创建视图

在浏览数据时， SELECT 语句对一个或多个表进行操作并产生一个结果集，该结果集也是一个表。与基表类似， SELECT 查询的结果集具有列和行。视图为特定的查询指定一个名称，并在数据库系统表中保存定义。

假设您需要经常列出每个部门的雇员数。您可以使用以下语句得到该列表：

```
SELECT dept_ID, count(*)
 FROM employee
 GROUP BY dept_ID
```

您可以使用 Sybase Central 或 Interactive SQL 创建包含此语句结果的视图。

❖ 创建新视图 (Sybase Central)：

1 连接到数据库。

2 打开该数据库的 [视图] 文件夹。

3 从 [文件] 菜单中，选择 [新建] → [视图]。

即会出现 [创建视图] 向导。

4 请按照向导中的说明进行操作。当该向导退出时，您可以在右窗格的 SQL 选项卡上编辑代码。

5 通过输入您要使用的表和列完成代码。对于上面的示例，请键入 **employee** 和 **dept_ID**。

6 从 [文件] 菜单中选择 [保存视图]。

新视图即会显示在 [视图] 文件夹中。

❖ 创建新视图 (SQL)：

1 连接到数据库。

2 执行 CREATE VIEW 语句。

示例

创建一个名为 DepartmentSize 的视图，该视图包含本节开始处给出的 SELECT 语句的结果：

```
CREATE VIEW DepartmentSize AS
 SELECT dept_ID, count(*)
 FROM employee
 GROUP BY dept_ID
```

由于视图中的信息不单独存储在数据库中，因此引用视图将执行关联的 SELECT 语句以检索相应的数据。

一方面，如果某个用户修改了 employee 表，则 DepartmentSize 视图中的信息会自动更新，这会有一定的好处。另一方面，您每次使用视图时，复杂的 SELECT 语句可能会增加 SQL 查找正确的信息所需的时间。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE VIEW 语句」。

使用视图

有关 SELECT 语句的限制

可以用作视图的 SELECT 语句有一些限制。特别是您不能在 SELECT 查询中使用 ORDER BY 子句。关系表的一个特性是行或列的顺序没有特殊意义，而使用 ORDER BY 子句将在视图的行上强加顺序。您可以在视图定义中使用 GROUP BY 子句、子查询和连接。

若要完善视图，您可以调整 SELECT 查询本身直到它完全以所需的格式提供您需要的结果。完全调整好 SELECT 查询之后，便可以在查询前面添加一个短语来创建视图：

```
CREATE VIEW view-name AS
```

更新视图

某些视图上允许使用 UPDATE、 INSERT 和 DELETE 语句，但另一些视图上却不允许使用，这取决于其关联的 SELECT 语句。

您不能更新包含集合函数 （例如 COUNT(*)）的视图。您也不能更新 SELECT 语句中包含 GROUP BY 子句的视图和包含 UNION 操作的视图。在所有这些情况下，都无法将 UPDATE 转换为对基表的一个操作。

复制视图

在 Sybase Central 中，您可以在数据库之间复制视图。若要执行此操作，请在 Sybase Central 的右窗格中选择视图，然后将其拖到另一个已连接数据库的 [视图] 文件夹中。于是将创建一个新视图，并将原始视图的代码复制到其上。

请注意，只将视图代码复制到新视图。不复制其它视图属性，例如权限。

使用 WITH CHECK OPTION 子句

即使在允许对视图执行 INSERT 和 UPDATE 语句时，在基表中插入或更新的行也有可能不符合视图本身的要求。例如，即使 INSERT 或 UPDATE 修改了基表，视图中也会没有新行。

使用 WITH CHECK OPTION 子句的示例

以下示例说明了 WITH CHECK OPTION 子句的用途。此可选子句是 CREATE VIEW 语句中的最终子句。

❖ **创建一个显示销售部的雇员的视图 (SQL)：**

1 键入以下语句：

```
CREATE VIEW sales_employee
 AS SELECT emp_id,
           emp_fname,
           emp_lname,
           dept_id
 FROM employee
 WHERE dept_id = 200
```

该视图的内容如下所示：

```
SELECT *
 FROM sales_employee
```

它们在 Interactive SQL 中的形式如下所示：

emp_id	emp_fname	emp_lname	dept_id
129	Philip	Chin	200
195	Marc	Dill	200

emp_id	emp_fname	emp_lname	dept_id
299	Rollin	Overbey	200
467	James	Klobucher	200
…	…	…	…

2　将 Philip Chin 调到市场部　此视图更新导致该条目从视图中消失，这是因为它不再符合视图选择标准。

```
UPDATE sales_employee
 SET dept_id = 400
 WHERE emp_id = 129
```

3　列出销售部的所有雇员　检查视图。

```
SELECT *
 FROM sales_employee
```

Emp_id	Emp_fname	Emp_lname	Dept_id
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
…	…	…	…

4　列出销售部的所有雇员　检查视图。

```
SELECT *
 FROM sales_employee
```

Emp_id	emp_fname	emp_lname	dept_id
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200

Emp_id	emp_fname	emp_lname	dept_id
641	Thomas	Powell	200
…	…	…	…

在使用 WITH CHECK OPTION 创建视图时，将对视图上的所有 UPDATE 或 INSERT 语句进行检查以确保新行符合视图条件。如果不符合，则该操作将引发错误并被拒绝。

以下修改的 sales_employee 视图拒绝前面的 UPDATE 语句，同时生成以下错误消息：

```
在基表 'employee' 中插入 / 更新时 WITH CHECK OPTION 违规
```

- **创建一个显示销售部的雇员的视图 （第二次尝试）** 这一次使用 WITH CHECK OPTION。

```
CREATE VIEW sales_employee
 AS SELECT emp_id, emp_fname, emp_lname, dept_id
     FROM employee
     WHERE dept_id = 200
 WITH CHECK OPTION
```

检查选项可被继承

如果在 sales_employee 视图上定义了视图 （例如 V2），那么在 V2 上所做的任何更新或插入，只要导致 sales_employee 上的 WITH CHECK OPTION 条件失败，都将被拒绝，即使在定义 V2 时未使用检查选项也是如此。

修改视图

您可以使用 Sybase Central 和 Interactive SQL 修改视图。在执行此操作时，您不能直接重命名现有视图，而必须创建一个具有新名称的新视图，将以前的代码复制到新视图，然后删除旧视图。

在 Sybase Central 中，您可以在右窗格中对象的 SQL 选项卡上编辑视图、过程和函数的代码。您可以在单独的窗口中编辑视图。方法是：右击视图，然后从弹出式菜单中选择 [在 ‘ 新建 ’ 窗口中编辑]。在 Interactive SQL 中，您可以使用 ALTER VIEW 语句修改视图。 ALTER VIEW 语句用新定义替换视图定义，但它会保留视图上的权限。

有关变更数据库对象属性的详细信息，请参见 「设置数据库对象的属性」第 38 页。

有关设置权限的详细信息，请参见 『ASA 数据库管理指南』> 「授予针对表的权限」和 『ASA 数据库管理指南』> 「授予针对视图的权限」。有关撤消权限的信息，请参见 『ASA 数据库管理指南』> 「撤消用户权限」。

❖ **编辑视图定义 (Sybase Central)：**

1. 打开 [视图] 文件夹。
2. 选择所需的视图。
3. 在右窗格中，单击 SQL 选项卡，然后编辑视图的代码。

提示
如果您要编辑多个视图，可能会需要为每个视图单独打开一个窗口，而不是在右窗格的 SQL 选项卡上编辑每个视图。打开单独窗口的方法是：右击视图，然后从弹出式菜单中选择 [在 ' 新建 ' 窗口中编辑]。

❖ **编辑视图定义 (SQL)：**

1. 使用 DBA 权限或以视图所有者的身份连接到数据库。
2. 执行 ALTER VIEW 语句。

示例

重命名 DepartmentSize 视图的列名 （在 「创建视图」第 60 页一节中已作了介绍），以便它们的名称可以提供详细信息。

```
ALTER VIEW DepartmentSize
   (Dept_ID, NumEmployees)
 AS
   SELECT dept_ID, count(*)
   FROM Employee
   GROUP BY dept_ID
```

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER VIEW 语句」。

删除视图

您可以在 Sybase Central 和 Interactive SQL 中删除视图。

❖ **删除视图 (Sybase Central)：**

1 打开 [视图] 文件夹。

2 右击所需的视图，然后从弹出式菜单中选择 [删除]。

❖ **删除视图 (SQL)：**

1 使用 DBA 权限或以视图所有者的身份连接到数据库。

2 执行 DROP VIEW 语句。

示例 删除名为 DepartmentSize 的视图。

```
DROP VIEW DepartmentSize
```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

浏览视图中的信息

若要浏览视图中保存的数据，您可以使用 Interactive SQL 实用程序。该实用程序允许您执行查询以标识要查看的数据。有关使用这些查询的详细信息，请参见 「查询：从表中选择数据」第 225 页。

如果您在使用 Sybase Central，则可以右击您对其具有权限的视图，然后从弹出式菜单中选择 [在 Interactive SQL 中查看数据]。此命令将打开 Interactive SQL，视图的内容会显示在 [结果] 窗格的 [结果] 选项卡上。要浏览视图，可在 Interactive SQL 中执行 `select * from owner.view` 语句。

系统表中的视图

有关数据库中的视图的所有信息都保存在系统表 SYS.SYSTABLE 中。在系统视图 SYS.SYSVIEWS 中，这些信息的显示格式更便于阅读。有关这些内容的详细信息，请参见 『ASA SQL 参考』> 「SYSTABLE 系统表」和 『ASA SQL 参考』> 「SYSVIEWS 系统视图」。

您可以使用 Interactive SQL 浏览这些表中的信息。在 [SQL 语句] 窗格中键入以下语句即可查看到 SYS.SYSVIEWS 视图中的所有列：

```
SELECT *
 FROM SYS.SYSVIEWS
```

若要抽取包含特定视图定义的文本文件，请使用如下语句：

```
SELECT viewtext FROM SYS.SYSVIEWS
     WHERE viewname = 'DepartmentSize';
 OUTPUT TO viewtext.sql
 FORMAT ASCII
```

使用索引

在设计和创建数据库时，性能是一个需要考虑的重要因素。索引可以显著提高用于搜索特定行或特定行子集的语句的性能。另一方面，索引会占用额外的磁盘空间，还可能减慢插入、更新和删除的速度。

选择一组索引

为数据库选择一组适当的索引是优化性能的重要组成部分。确定一组适当的索引也是一个要求很高的问题。有些索引可能会极大地提高性能，但索引也会有相关的成本问题，存储空间和修改数据时的开销这两个方面都需要增加成本。

索引顾问是帮助您正确选择索引的工具。它会分析一个查询或一组操作，然后建议您给数据库添加哪些索引。有未使用的索引，它也会告诉您。

有关索引顾问的详细信息，请参见 「启动索引顾问」第 76 页。

何时使用索引

索引对列中的各行或表中的各列进行排序。索引像一个电话号码簿，电话号码簿先按照姓进行排序，然后按照名对相同的姓进行排序。这种排序加快了对特定的姓搜索其电话号码的速度，但它对查找位于特定地址的电话号码未提供任何帮助。同样，数据库索引只对搜索特定的列有帮助。

随着表的大小不断增大，索引的用途也越来越大。随着电话号码簿内容的增多，查找某个位于给定地址的电话号码所需的平均时间也将会增加。但是，与在一个较小的电话号码簿中查找某个人（比方说 K. Kaminski）的电话号码相比，在一个较大的电话号码簿中查找所花费的时间不会长得太多。

当存在合适的索引并且使用该索引将提高性能时，数据库服务器查询优化程序将自动使用索引。

创建索引有一些副作用。特别是当修改列中的数据时，所有索引都必须与表本身保持一致，从而导致索引可能会影响插入、更新和删除的性能。因此，应该删除不必要的索引。请使用 [索引顾问] 标识不必要的索引。

有关索引顾问的详细信息，请参见 「启动索引顾问」第 76 页。

对经常被搜索的列使用索引

索引需要额外的空间，并可能稍微降低某些会修改表中的数据的语句（例如 INSERT、 UPDATE 和 DELETE 语句）的性能。但是，索引可以显著提高搜索性能，因此强烈建议您，只要经常搜索数据，就应使用索引。

有关性能的详细信息，请参见 「使用索引」第 177 页。

Adaptive Server Anywhere 自动为主键和外键列建立索引。因此，没有必要在键列上手工创建索引，通常也不建议这样做。如果列只是键的一部分，则创建索引可能会有帮助。

Adaptive Server Anywhere 在任何可能的时候都会自动使用索引提高任何数据库语句的性能。一旦创建索引，就无需显式引用索引了。另外，当删除、更新或插入行后，将自动对索引进行更新。

索引提示

Adaptive Server Anywhere 允许您提供索引提示，它通过强制使用特定索引来替换优化程序所选的查询访问计划。此功能只应由高级用户和数据库管理员使用。它可能导致性能降低。

要提供索引提示，请在查询中添加 **FORCE INDEX** 或 **WITH (INDEX ())** 子句。有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。

有关变更数据库对象属性的信息，请参见 「设置数据库对象的属性」第 38 页。

使用聚簇索引

虽然标准索引可以大大提高语句搜索特定的行或特定行子集时的性能，但在索引中相继出现的两行在数据库中却不一定出现在同一页上。

通过创建聚簇索引，您可以进一步提高索引的性能。 Adaptive Server Anywhere 中的聚簇索引存储表行的顺序大致与表行在对应索引中的顺序相同。

使用聚簇索引特性可以提高临近两行出现在同一数据库中页上的可能性。通过进一步减少每一页需要读取到内存中的次数，这有助于提高性能。

例如，如果您选择在聚簇索引中相继出现的两行，则很可能您在检索在同一页上相继出现的两行，这样便可将读取到内存中的页数减少一半。

Adaptive Server Anywhere 中的聚簇是只是大致聚簇。虽然服务器尝试保留键顺序，但不保证能够实现总体聚簇。此外，随着时间推移减弱，由于越来越多的行插入到数据库中，聚簇功能会减弱。

您可以使用下面的语句，对每个表实现一个聚簇索引：

- CREATE TABLE 语句
- ALTER TABLE 语句
- CREATE INDEX 语句
- DECLARE LOCAL TEMPORARY TABLE 语句

可以联合使用若干语句来维护和恢复聚簇效果：

- UNLOAD TABLE 语句可用于按照索引键的顺序卸载表。
- LOAD TABLE 语句可用于按照索引键的顺序在表中插入行。
- INSERT 语句按照每个主键顺序尝试在包含相邻行的页上放置新行。
- REORGANIZE 表语句可以通过按照聚簇索引重新排列行来恢复聚簇。没有指定聚簇的表是按照主键排序的。

优化程序假定表行是按照键顺序存储的，因而会相应地进行索引扫描。

创建索引

索引是在指定表的一列或多列上创建的。您可以在基表或临时表上创建索引，但不能在视图上创建索引。要创建单独的索引，请使用 Sybase Central 或 Interactive SQL。您可以让 [索引顾问] 指导您为数据库选择适当的索引。

❖ **为给定表创建新索引 (Sybase Central)：**

1. 打开 [索引] 文件夹。
2. 从 [文件] 菜单中，选择 [新建] → [索引]。

   即会出现 [创建索引] 向导。
3. 为索引命名，并从列表中选择表。单击 [下一步]。
4. 请按照向导中的说明进行操作。

   新索引即会显示在 [索引] 文件夹中。

❖ **为给定表创建新索引 (SQL)：**

1. 使用 DBA 权限或以表 （在该表上创建索引）所有者的身份连接到数据库。
2. 执行 CREATE INDEX 语句。

除了在表的一列或多列上创建索引外，您还可以使用计算列在内置函数上创建索引。有关详细信息，请参见 『ASA SQL 参考』> 「CREATE INDEX 语句」。

示例

为加快对示例数据库中雇员的姓的搜索速度，您可以使用以下语句创建一个名为 EmpNames 的索引：

```
CREATE INDEX EmpNames
 ON employee (emp_lname, emp_fname)
```

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE INDEX 语句」和 「监控和提高性能」第 167 页。

校验索引

您可以对索引进行校验以确保索引中引用的每一行都实际存在于表中。对于外键索引，校验检查过程还可确保对应的行存在于主表中，并且它们的散列值相匹配。这种检查补充了由 VALIDATE TABLE 语句执行的有效性检查。

应在没有任何连接对数据库进行更改时对表或整个数据库进行校验；否则，可能会报告虚假错误，指出某种形式的数据库损坏，而实际上并没有任何损坏。

❖ **校验索引 (Sybase Central)**

1 使用 DBA 权限或以表 （在该表上创建索引）所有者的身份连接到数据库。

2 在左窗格中，打开 [索引] 文件夹。

3 右击所需的索引，然后从弹出式菜单中选择 [校验]。

❖ **校验索引 (SQL)：**

1 使用 DBA 权限或以表 （在该表上创建索引）所有者的身份连接到数据库。

2 执行 VALIDATE INDEX 语句。

❖ **校验索引 （命令行）：**

1 打开命令提示符。

2 运行 dbvalid 实用程序。

示例

校验名为 EmployeeIndex 的索引。如果您提供的是表名而不是索引名，则将校验主键索引。

```
VALIDATE INDEX EmployeeIndex
```

校验名为 EmployeeIndex 的索引。 **-i** 命令行选项指定每个给定的对象名称都是索引。

```
dbvalid -i EmployeeIndex
```

有关详细信息，请参见 『ASA SQL 参考』> 「VALIDATE INDEX 语句」和 『ASA 数据库管理指南』> 「校验实用程序」。

删除索引

如果不再需要某个索引，则可以在 Sybase Central 或 Interactive SQL 中将其从数据库中删除。

❖ 删除索引 (Sybase Central)

1. 使用 DBA 权限或以表 （在该表上创建索引）所有者的身份连接到数据库。
2. 在左窗格中，打开 [索引] 文件夹。
3. 右击所需的索引，然后从弹出式菜单中选择 [删除]。

❖ 删除索引 (SQL)

1. 使用 DBA 权限或以表 （该表与索引关联）所有者的身份连接到数据库。
2. 执行 DROP INDEX 语句。

示例

以下语句从数据库中删除索引：

```
DROP INDEX EmpNames
```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

索引顾问概述

选择一组适当的索引可能会给数据库性能带来极大的变化。为了帮助您选择这样一组索引， Adaptive Server Anywhere 提供了 [索引顾问]。[索引顾问] 指导您完成为单个查询或一组数据库请求（称为**负载**）选择索引的过程。它创建许多不同的虚拟索引组。对于每一组，索引顾问优化查询和其它请求，就像这些虚拟索引存在一样。然后，索引顾问将这些探查的结果汇集成一组建议。

[索引顾问] 的工作分为几个阶段。了解这些阶段有助于您充分利用该工具。

1. 启动 [索引顾问]

您可以从 Interactive SQL 或 Sybase Central 运行 [索引顾问]。如果要分析单个查询的索引，请从 Interactive SQL 访问 [索引顾问]；如果要分析负载的索引，请从 Sybase Central 访问 [索引顾问]。

有关如何启动 [索引顾问] 的信息，请参见 「启动索引顾问」第 76 页。

2. 捕获负载。

   负载是一组查询或其它数据操作语句，[索引顾问] 尝试通过这些查询或语句来优化性能。根据您目标的不同，您可能想让负载成为具有代表性的一组操作，也可能想标识一组关键的瓶颈操作。[索引顾问] 可以捕获一个或多个负载，然后将它们存储起来供以后分析使用。

   有关详细信息，请参见 「了解负载」第 77 页。

3. 分析负载。

   [索引顾问] 会生成候选索引并研究它们对性能的影响，从而分析负载或一个查询。为了研究不同候选索引的影响，[索引顾问] 会在各组不同的索引下反复重新优化负载中的查询。但它不执行查询。

   [索引顾问] 还可以使用不同设置存储对任何负载的多个分析结果。

   在分析负载时，[索引顾问] 显示一组选项：

   - **推荐使用的聚簇索引** 如果选择此选项，[索引顾问] 将分析聚簇索引和非聚簇索引的效果。

     对于某些负载而言，选择得当的聚簇索引与非聚簇索引相比，可以在更大程度上提高性能，但您必须重组表（使用 REORGANIZE TABLE 语句），这样才能使索引生效。此外，如果考虑聚簇索引的效果，分析将占用更长的时间。

     有关聚簇索引的详细信息，请参见 「使用聚簇索引」第 69 页。

- **保留现有的次级索引** [索引顾问] 可以通过在数据库中维护现有的一组次级索引或忽略现有的次级索引来执行分析。次级索引是这样一个索引：它不是唯一约束，也不是主键或外键。在选择访问计划时，总是会考虑为强制参照完整性约束而存在的索引。

分析包括以下步骤：

- **生成候选索引** 对于每个负载，[索引顾问] 都会生成一组候选索引。在大表上创建实际索引是一个耗时的操作，因此 [索引顾问] 创建它的候选索引作为虚拟索引。虚拟索引不能用来实际执行查询，但查询优化程序可以使用虚拟索引来预计执行计划的成本，就像实际存在此类索引一样。虚拟索引允许 [索引顾问] 执行假定推测分析，这样就不会产生创建和管理实际索引的开销。虚拟索引的列数限制为四列。

- **测试候选索引的收益和成本** [索引顾问] 会要求 Adaptive Server Anywhere 查询优化程序预计在使用和不使用候选索引的不同组合的情况下在负载中执行查询的成本。

- **生成建议** [索引顾问] 汇编查询成本的结果，并按索引提供的总收益对索引排序。它提供一个 SQL 脚本，您可以通过运行该脚本来执行建议，也可以保存该脚本供自己检查和分析。

有关详细信息，请参见 「理解分析」第 78 页。

4. 实现建议

   [索引顾问] 提供一个 SQL 脚本，您可以通过运行该脚本来实现它的建议。在实现之前，您可能想根据自己对数据库的了解来评估这些建议。

结合 Interactive SQL 使用索引顾问

[索引顾问] 一次只可以分析一条来自 Interactive SQL 的语句。

有关详细信息，请参见 「评估建议」第 80 页和 「实现建议」第 81 页。

启动索引顾问

索引顾问指导您正确地选择索引。您可以通过下列方式使用 [索引顾问]：

- **Interactive SQL** 您可以从 Interactive SQL 中使用 [索引顾问] 来分析索引对于某个查询的益处。
- **Sybase Central** 您可以从 Sybase Central 中使用 [索引顾问] 来分析索引对于负载或一组数据库请求的益处。

❖ 启动索引顾问 (Sybase Central)

1. 在 Sybase Central 的左窗格中，右击 [索引] 文件夹。
2. 从弹出式菜单中选择 [索引顾问]。

❖ 启动索引顾问 (Interactive SQL)

1. 在 Interactive SQL 的 [SQL 语句] 窗格中，输入要分析的查询。
2. 选择 [工具] → [索引顾问]。

停止索引顾问

[索引顾问] 在捕获用于分析的负载时，确实会影响数据库性能。因此，在 [索引顾问] 捕获负载时，数据库服务器窗口将显示信息性消息。

通常从 Sybase Central 用户界面停止 [索引顾问]。某些情况下，您可能需要从其它计算机停止 [索引顾问]：例如，因疏忽而运行 [索引顾问] 时。

要暂停或停止捕获负载，请分别调用 sa_pause_workload_capture 过程或 sa_stop_workload_capture 过程：

```
call sa_pause_workload_capture;
 call sa_stop_workload_capture;
```

要停止 [索引顾问]，请调用 sa_stop_index_consultant 系统存储过程：

```
call sa_stop_index_consultant
```

了解索引顾问

为数据库选择最佳的一组索引，是个没有明确答案的问题。[索引顾问] 是在广泛搜索各种可选索引模式的结果并细致分析与索引相关的成本和收益的基础上提出建议的。不过，问题的本质在于，并不能保证使用 [索引顾问] 的建议始终都能得到最佳的一组索引。

本节对 [索引顾问] 如何工作的一些方面进行了介绍。本节的目标是为您提供信息，这些信息能帮助您更好地判断 [索引顾问] 提出的建议。

了解负载

[索引顾问] 工作的第一步是捕获数据操作和查询操作的负载。为此，它在一个或一组应用程序对数据库执行操作时捕获信息。

负载包括捕获步骤中执行的所有 SELECT、 INSERT、 UPDATE 和 DELETE 语句。这些操作的上下文不会捕获到负载中。例如，虽然作为存储过程、触发器或事件处理程序的一部分而执行的查询会被捕获为负载的一部分，但周围的代码不会被捕获。这样就会导致下面的结果：

- 不捕获参数。

  参数化的 SQL 语句的存储方式是将相应的实际值替换为语句定义。例如，如果某个查询的 WHERE 子句带有参数，那么在该查询执行 15 次之后 （每一次都采用不同的参数值），[索引顾问] 将捕获 15 个不同的查询 （每个查询都带有单独的常量），而不是捕获一个带参数的查询。

- 不能分析对临时表所做的查询。

  虽然针对临时表的查询会被收集起来作为负载的一部分，但不会捕获临时表中存在的数据。因此，无法在分析步骤中分析查询。[索引顾问] 不能就临时表上的适用索引提供建议。

- 不完全捕获连接状态。

连接状态包含在执行查询时有效的数据库选项集。OPTIMIZATION_GOAL 选项通常对优化程序选择的执行计划有着重大影响，它通常与查询存储在一起。不过，不存储其它选项。

- 不捕获服务器状态。

  服务器状态包含：数据是否位于高速缓存中，或者是否必须从磁盘读取数据。它还包含并发操作的影响。

注意
不要在负载捕获步骤中更改数据库模式。不要在捕获步骤与分析步骤之间更改数据库模式。此类更改会使 [索引顾问] 的建议无效。

理解分析

分析步骤是通过反复执行下面一组操作来执行的：

1. 创建一组候选虚拟索引。

   虚拟索引不包含实际数据，不能用于实际的查询执行。它可以由优化程序在为查询选择适当的执行计划时使用。 [索引顾问] 生成许多组不同的虚拟索引。

2. 为这一组候选虚拟索引优化负载操作。

   [索引顾问] 会为负载中的每条语句检索 Adaptive Server Anywhere 查询优化程序选择的计划。优化程序在候选的各组虚拟索引中考虑适用于每条语句的虚拟索引，但不执行这些语句。 [索引顾问] 不修改用户数据。

   对于每个查询，查询优化程序将比较多个不同执行计划的执行成本。它会根据内部成本模型来预计执行成本。影响每个执行计划成本的一个重要选择是使用索引访问哪些表，以及不使用索引访问哪些表。每组虚拟索引都会打开新的可选执行计划，并关闭其它计划。

   成本模型取决于数据库服务器的状态。具体而言，虽然 [索引顾问] 本身不从磁盘读取数据或对数据库执行操作，但成本模型取决于高速缓存中有哪些数据以及必须从磁盘访问哪些数据。因此，在每次您针对特定负载运行分析时，运行 [索引顾问] 不一定生成相同的建议。例如，在刚启动完 （此时高速缓存中没有数

据）的数据库服务器上运行 [索引顾问]，与在已经运行了一段时间的数据库服务器上运行 [索引顾问] 相比，所得到的建议可能是不同的。

有关成本模型的详细信息，请参见 「优化程序预计值和直方图」第 429 页。

了解建议

[索引顾问] 提供一组选项卡存放给定分析的结果。您可以保存分析结果供以后检查使用。

[摘要] 选项卡

[摘要] 选项卡上概述了负载和分析，包括负载中的查询数量、推荐索引的数量、推荐索引所需的页数，以及推荐索引应该产生的收益等信息。收益数量用内部成本单位计量。

[推荐索引] 选项卡

[推荐索引] 选项卡包含有关每个推荐索引的数据。它提供如下信息：

- **聚簇** 每个表最多可以有一个聚簇索引。某些情况下，聚簇索引的收益比非聚簇索引的收益大得多。

  有关聚簇索引的详细信息，请参见 「使用聚簇索引」第 69 页。

- **页数** 选择创建索引时，预计保存索引所需的数据库页数量。

  有关数据库页大小的详细信息，请参见 『ASA 数据库管理指南』> 「初始化实用程序」。

- **相对收益** 一个 1 至 10 之间的数字，表明预计的创建指定索引的总体收益。数字越大，说明预计收益越大。

  相对收益是使用独立于 [总成本收益] 列的内部算法计算的。预计相对收益时，有几因素是总成本收益中没有的。例如，一个索引的存在可以显著影响与另一索引相关的收益。在此情况下，相对收益将尝试估计每个索引各自的影响。

  有关详细信息，请参见 「评估建议」第 80 页。

- **总收益** 与索引相关的成本缩减，针对负载中的所有操作总计得出，用内部成本单位计量。

有关成本模型的详细信息，请参见 「优化程序预计值和直方图」第 429 页。

- **更新成本** 添加索引将增加成本，一方面需要额外的存储空间，另一方面在修改数据时也需要额外的工作。 [更新成本] 列是与索引相关的额外维护成本的预计值。它用内部成本单位计量。
- **总成本收益** 总收益减去与索引相关的更新成本。

[请求] 选项卡

[请求] 选项卡为负载内的各个请求细分建议的影响。该信息包含在应用推荐索引之前和之后的预计成本，以及由查询使用的虚拟索引。点击一个按钮即可查看为请求找到的最佳执行计划。

[更新] 选项卡

[更新] 选项卡细分建议的影响。

[未使用的索引] 选项卡

[未使用的索引] 选项卡列出已存在于数据库中，但未在负载内的任何请求执行中使用的索引。只有次级索引会被列出，也就是说，既不会列出主键和外键上的索引，也不会列出唯一约束。

[日志] 选项卡

[日志] 选项卡列出已为该分析完成的活动。

评估建议

虽然 [索引顾问] 的建议基于对可用选项的广泛搜索，但在生产数据库中实现这些建议之前先对它们进行评估仍是一个好做法。例如，您可能想执行以下检查：

- **建议的索引是否与您自己的期望相符？** 如果您非常了解数据库中的数据，并且了解目前对数据库运行的查询，那么您可能想凭借自己的认识来检查建议的索引是否有用。或许建议的索引只影响很少运行的单个查询，或许建议的索引位于小表上，总体影响较小。或许 [索引顾问] 建议应删除的索引用于其它一些没有包含在您的负载中的任务。
- **建议的各个索引所产生的效果之间是否存在很强的相关性？** 索引建议会尝试单独评估各个索引的相对收益。不过，情况可能会是这样：两个索引只有都存在时才会被使用 （如果它们存在，查询可能会使用两个，如果缺少其中一个，则哪个不用）。您可能想深入了解 [请求] 选项卡并检查查询计划，以查看建议的索引的使用方式。

- **您是否能在创建聚簇索引时重组表？** 为了充分利用聚簇索引的优点，您应该使用 REORGANIZE TABLE 语句重组创建聚簇索引所在的表。如果 [索引顾问] 推荐许多聚簇索引，您可能必须在卸载和重装数据库之后才能获得全部益处。卸载和重装表是一个非常耗时的操作，并且需要大量的磁盘空间资源。您可能需要确认自己具有实现建议所需的时间和资源。
- **分析期间的服务器和连接的状态是否反映了生产运营中的实际状态？** 分析结果取决于服务器的状态，包括位于高速缓存中的数据。它们也取决于连接的状态，包括某些数据库选项设置。由于分析只创建虚拟索引而不执行请求，因此，服务器的状态在分析期间实质是静态的 （其它连接引入的更改除外）。如果该状态不能代表数据库的典型操作，您可能想在其它不同条件下重新运行分析。

实现建议

[索引顾问] 以 SQL 脚本的形式提供建议。您可以从 [索引顾问] 实现建议，也可以保存该脚本供以后使用。

建议索引的名称是根据分析的名称生成的。您在数据库中创建索引之前，可能会需要先将这些名称重命名。

有关从 Interactive SQL 运行命令文件的信息，请参见 「使用 SQL 命令文件」第 637 页。

系统表中的索引

有关数据库中的索引的所有信息都保存在系统表 SYS.SYSINDEX 和 SYS.SYSIXCOL 中。在系统视图 SYS.SYSINDEXES 中，这些信息的显示格式更便于阅读。您可以使用 Sybase Central 或 Interactive SQL 浏览这些表中的信息。

使用临时表

不管是局部临时表还是全局临时表，其目的是相同的：临时存储数据。但是，局部临时表和全局临时表之间的差异以及各自的优势则在于每种表存在的持续时间的长短。

局部临时表的存在时间仅为一个连接存在的持续时间，或者如果该表是在复合语句中定义的，则其存在时间是复合语句存在的持续时间。

有关详细信息，请参见 『ASA SQL 参考』> 「DECLARE LOCAL TEMPORARY TABLE 语句」。

全局临时表的定义永久地保留在数据库中，但行仅存在于给定的连接中。在关闭数据库连接后，全局临时表中的数据就会消失。但是，表定义仍保留在数据库中，以便您下次打开数据库时可以访问表定义。

临时表存储在临时文件中。与任何其它 dbspace 类似，可以对临时文件中的页进行高速缓存。对临时表的操作决不会写入事务日志。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

在 Sybase Central 中复制数据库对象

在 Sybase Central 中，您可以复制现有的数据库对象，然后将它们插入同一数据库中的其它位置，或插入一个完全不同的数据库中。

要复制对象，请在 Sybase Central 的左窗格中选中对象，然后将它拖到相应的文件夹或容器中；或者，复制对象，然后将它粘贴到相应的文件夹或容器中。此时将创建新对象，而且原始对象的代码会被复制到新对象中。在同一个数据库中复制对象时，必须将新对象重命名。

您还可以将对象粘贴到数据库中的其它对象上。例如，如果您将一个表粘贴到某个用户，就会授予该用户访问该表的权限。

复制数据库对象的代码供其它应用程序使用

在 Sybase Central 中，当您从以下列表复制任何对象时，该对象的 SQL 将被复制到剪贴板上，这样，您就可以将它粘贴到其它应用程序（如 Interactive SQL 或文本编辑器）中。例如，如果在 Sybase Central 中复制一个索引，然后将它粘贴到文本编辑器中，则出现该索引的 CREATE INDEX 语句。

- 项目
- 检查约束
- 列（仅限 [复制] 按钮）
- 域
- 事件
- 外部登录
- 外键
- 索引
- 消息类型
- 过程和函数
- 发布
- 远程服务器

- 系统触发器
- 表
- 触发器
- UltraLite 项目
- UltraLite 语句
- 唯一约束
- 用户
- 视图
- Web 服务

第 3 章

确保数据完整性

关于本章

在数据库中构建完整性约束是确保数据完整有效的最可靠的方法。本章讨论 Adaptive Server Anywhere 中用于确保数据库中的数据有效并且可靠的功能。

可以实施几类完整性约束。例如，您可以通过对表和列施加约束和 CHECK 约束来确保各个条目正确。选择适当的数据类型来设置列属性或设置特殊缺省值对此任务有所帮助。

本章中使用了 CREATE TABLE 和 ALTER TABLE 语句，这些语句的基本形式在 「使用数据库对象」第 31 页中进行了介绍。

数据完整性概述

如果数据具有完整性，则表示数据有效 （正确并且准确），而且数据库的关系结构是完整的。参照完整性约束可以加强数据库的关系结构。这些规则使数据在各个表之间保持一致。

Adaptive Server Anywhere 支持存储过程，利用存储过程可以对数据进入数据库的方式进行具体的控制。您还可以创建触发器，或系统在发生特定操作 （例如，更新特定的列）时会自动调用的自定义存储过程。

有关过程和触发器的详细信息，请参见 「使用过程、触发器和批处理」第 699 页。

哪些情况会导致数据无效

如果不进行适当的检查，下列几种情况可能会导致数据库中的数据无效。可以使用本章中介绍的功能防止这些情况发生。

信息不正确

- 操作员键入的交易日期不正确。
- 由于操作员在输入数据时丢失了一位数字，导致雇员的工资变为实际工资的十分之一。

数据重复

- 两名不同人员在公司数据库的部门表中添加相同的新部门（**dept_id** 为 200）。

外键关系失效

- 由 **dept_id** 300 标识的部门会关闭，并且由于疏忽而导致一条雇员记录未分配到新部门。

完整性约束属于数据库

为确保数据库中数据的有效性，需要创建检查语句以定义有效数据和无效数据，并设计数据必须遵守的规则 （也称作业务规则）。检查语句和规则一起构成**约束**。

在数据库中构建完整性约束

构建数据库本身的约束比构建客户端应用程序的约束或将约束形成文字说明提供给数据库用户更为可靠。构建数据库的约束将成为数据库本身定义的一部分，并且数据库将在所有应用程序中一致地实施这些约束。只需在数据库中设置约束一次，对以后与数据库进行的所有交互都将实施该约束。

相反，构建客户端应用程序的约束在每次软件更改时都会受到影响，而且可能需要在多个应用程序，或一个客户端应用程序的多个位置中实施。

数据库内容如何更改

从客户端应用程序提交 SQL 语句后，数据库表中的信息会发生更改。实际上，只有几个 SQL 语句会修改数据库中的信息。您可以：

- 使用 UPDATE 语句可以更新表中某一行的信息。
- 使用 DELETE 语句可以删除现有行。
- 使用 INSERT 语句可以在表中插入新行。

数据完整性工具

要保持数据完整性，您可以使用缺省值、数据约束和保持数据库参照结构的约束。

缺省值

可以为列指派缺省值以确保某些类型的数据条目更可靠。例如：

- 可以使用当前日期作为某一列的缺省值，来记录任何用户或客户端应用程序操作的事务日期。
- 使用其他类型的缺省值还可以使列值在输入一行新数据时自动递增，无需执行任何其他特定用户操作。利用此功能，可以确保项目（例如，订购单）具有唯一的序列编号。

有关这些列缺省值和其它列缺省值的详细信息，请参见「使用列缺省值」第 90 页。

约束

可以对各列或各表中的数据应用几种类型的约束。例如：

- NOT NULL 约束限制列不能包含空值条目。
- 为列分配 CHECK 约束可以确保该列中的每一项都满足特定的条件。例如，可以确保 [工资] 列的条目在指定的范围内，从而避免在键入新值时可能出现的用户错误。
- 可以针对不同列中的相对值设置 CHECK 约束。例如，您可以确保图书馆数据库中的 date_returned 条目晚于 date_borrowed 条目。
- 利用触发器可以实施更复杂的 CHECK 条件。有关触发器的详细信息，请参见 「使用过程、触发器和批处理」第 699 页。

还可以从域继承列约束。有关这些表约束和列约束以及其它表约束和列约束的详细信息，请参见 「使用表和列约束」第 96 页。

实体和参照完整性

由主键和外键定义的关系将关系数据库表中的信息联系起来。必须将这些关系构建数据库设计。以下完整性规则用于保持数据库的结构：

- **实体完整性** 跟踪主键，确保给定表中的每一行都由一个非空主键唯一地标识。
- **参照完整性** 跟踪定义表间关系的外键，确保所有外键值与相应主键中的值匹配，或包含 NULL 值 （如果定义为允许 NULL 值）。

有关实施参照完整性的详细信息，请参见 「实施实体完整性和参照完整性」第 105 页。有关设计适当的主键和外键关系的详细信息，请参见 「设计数据库」第 3 页。

使用触发器实施高级完整性规则

也可以使用触发器来保持数据完整性。**触发器**是指一个存储在数据库中的过程，只要指定的表中的信息发生更改，就会自动执行该过程。触发器是一个强大的机制，数据库管理员和开发人员可以用它来确保数据的可靠性。

有关触发器的详细信息，请参见 「使用过程、触发器和批处理」第 699 页。

用于实施完整性约束的 SQL 语句

以下 SQL 语句用于实施完整性约束：

- **CREATE TABLE 语句**　使用这个语句可在创建数据库时实施完整性约束。
- **ALTER TABLE 语句**　使用这个语句可在现有数据库中添加完整性约束，或修改现有数据库的约束。
- **CREATE TRIGGER 语句**　使用这个语句可以创建触发器，实施更复杂的业务规则。
- **CREATE DOMAIN 语句**　此语句创建用户定义的数据类型。数据类型的定义可以包括约束。

有关这些语句的语法的详细信息，请参见 『ASA SQL 参考』> 「SQL 语句」。

使用列缺省值

如果设置了列缺省值，每次有人在数据库表中输入一行新数据时都会自动为特定列指派一个指定的值。指派的缺省值不需要客户端应用程序执行任何操作，但是如果客户端应用程序确实为列指定了值，新值将替换列的缺省值。

使用列缺省值可以自动在列中快速填入信息，例如插入某一行的日期或时间或键入信息的人的用户 ID。使用列缺省值有助于保持数据完整性，但不能实施数据完整性。客户端应用程序总是可以替换缺省值。

支持的缺省值

SQL 支持以下缺省值：

- CREATE TABLE 语句或 ALTER TABLE 语句中指定的字符串
- CREATE TABLE 语句或 ALTER TABLE 语句中指定的数字
- 自动递增的数字：每增加一行，新行中该列的值将在原有最大值的基础上加一
- 使用 NEWID 函数生成的通用唯一标识符 (UUID)。
- 当前日期、时间或时间戳
- 当前数据库用户的用户 ID
- NULL 值
- 不参照任何数据库对象的常数表达式

创建列缺省值

可以使用 CREATE TABLE 语句在创建表时创建列缺省值，或使用 ALTER TABLE 语句在创建了表之后添加列缺省值。

示例

以下语句为 **sales_order** 表中名为 **id** 的现有列添加一个条件，以便它自动递增 （除非客户端应用程序指定一个值）：

```
ALTER TABLE sales_order
 MODIFY id DEFAULT AUTOINCREMENT
```

其他各种缺省值也使用相似的方式指定。有关详细信息，请参见『ASA SQL 参考』> 「ALTER TABLE 语句」和 『ASA SQL 参考』> 「CREATE TABLE 语句」。

修改和删除列缺省值

可以使用 ALTER TABLE 语句更改或删除列缺省值，方式与用该语句创建缺省值时相同。以下语句将名为 **order_date** 的列的缺省值从其当前设置更改为 CURRENT DATE：

```
ALTER TABLE sales_order
 MODIFY order_date DEFAULT CURRENT DATE
```

要删除列缺省值，可将其修改为 NULL。以下语句删除 **order_date** 列中的缺省值：

```
ALTER TABLE sales_order
 MODIFY order_date DEFAULT NULL
```

在 Sybase Central 中管理列缺省值

在 Sybase Central 中，可以使用列属性表的 [值] 选项卡添加、变更和删除列缺省值。

❖ **显示列的属性表：**

1. 连接到数据库。
2. 打开该数据库的 [表] 文件夹。
3. 双击包含要更改的列的表。
4. 在右窗格中，单击 [列] 选项卡。
5. 选择所需的列。
6. 选择 [文件] → [属性]。

   该列的属性表即会出现。

当前日期和时间缺省值

对于数据类型为 DATE、 TIME 或 TIMESTAMP 的列，可以使用当前日期、当前时间或当前时间戳作为缺省值。您选择的缺省值必须与该列的数据类型兼容。

当前日期缺省值的有用示例

使用当前日期作为缺省值可以：

- 在联系人数据库中记录通话日期
- 在销售记录数据库中记录订单日期
- 在图书馆数据库中记录借阅人的借书日期

当前时间戳

当前时间戳与当前日期缺省值相似，但精度更高。例如，联系管理应用程序的用户在一天中可能要与同一个客户联系多次：使用当前时间戳缺省值可以区分这些联系。

由于时间戳记录的日期和时间可以精确到百万分之一秒，因此，当事件的顺序在数据库中非常重要时，可能需要使用当前时间戳。

有关时间戳、时间和日期的详细信息，请参见 『ASA SQL 参考』> 「SQL 数据类型」。

用户 ID 缺省值

将 DEFAULT USER 指派给列可以标识在数据库中输入信息的人。这种方法既简单又可靠。例如，销售人员在管理佣金时可能需要该信息。

为表的主键设置用户 ID 缺省值对于偶尔连接的用户非常有用，并且有助于避免在更新信息时发生冲突。这些用户可以将与他们的工作相关的表复制到便携式计算机上，在不连接到多用户数据库的情况下进行更改，等到他们回来后，再将事务日志应用到服务器上。

自动增量缺省值

自动增量缺省值适用于那些数字值本身并没有含义的数字数据字段。每新增一行，该列中新行的值将在原有最大值的基础上加一。可以使用自动增量列记录订购单编号，标识客户服务呼叫，或其他需要标识号的条目。

自动增量列通常是主键列，或限制为只能包含唯一值的列 （请参见 「实施实体完整性」第 105 页）。例如，自动增量缺省值在该列是索引的第一列时会起很大作用，因为服务器使用索引或键定义来查找最大值。

虽然在其他情况下也可以使用自动增量缺省值，但这样做可能反而会导致数据库性能降低。例如，在各列的下一个值存储为整数 （4 个字节）的情况下，使用大于 2^{31} - 1 的值或较大的双精度值或数字值可能会导致返转为负值。

您可以使用 @@identity 全局变量来检索插入到自动增量列中的最新值。有关详细信息，请参见 『ASA SQL 参考』> 「@@identity 全局变量」。

自动增量和负数

自动增量设计用于正整数。

创建表时，自动增量的初始值为 0。执行完在该列中显式插入负值的插入操作后，初始值仍保持为分配给该列的最大值。如果插入操作未提供任何值，自动增量列将生成 1，强制生成的所有其他值都为正数。

在 UltraLite 应用程序中，创建表时自动增量值不会设置为 0。如果该列使用带符号的数据类型，自动增量列将生成负值。

应将自动增量列定义为不带符号的数据类型，以防止使用负值。

自动增量和 IDENTITY 列

具有自动增量缺省值的列在 Transact-SQL 应用程序中称为 IDENTITY 列。有关 IDENTITY 列的信息，请参见 「特殊 IDENTITY 列」第 521 页。

NEWID 缺省值

UUID （通用唯一标识符）也称为 GUID （全局唯一标识符）可用于唯一标识表中的行。在一台计算机上所生成的值与在其它计算机上生成的值不相同。因此，可将它们用作复制和同步环境中的键。

相比较而言，将 UUID 值和 GLOBAL AUTOINCREMENT 值用作主键各有优缺点，例如，

- UUID 的设置比 GLOBAL AUTOINCREMENT 更容易，因为没有必要为每个远程数据库都分配一个唯一的数据库 ID。也没有必要考虑系统中数据库的数目或单个表中的行数。可以使用抽取实用

程序 [dbxtract] 来进行数据库 ID 的指派。如果使用 BIGINT 数据类型，使用 GLOBAL AUTOINCREMENT 时通常不用考虑这个问题，但使用较小的数据类型时需要考虑这一点。

- UUID 值比 GLOBAL AUTOINCREMENT 所需的值大得多，要求主表和外表有更多空间。使用 UUID 时，这些列中的索引的效率较低。也就是说， GLOBAL AUTOINCREMENT 性能更好些。
- UUID 没有隐性顺序。例如，如果 A 和 B 是 UUID 值， A > B 并不意味着 A 在 B 之后生成，即使 A 和 B 都是在同一台计算机上生成的。如果您需要此行为，可能需要其它列和索引。

有关详细信息，请参见 『ASA SQL 参考』> 「NEWID 函数 [Miscellaneous]」或 『ASA SQL 参考』> 「UNIQUEIDENTIFIER 数据类型 [Binary]」。

NULL 缺省值

对于允许使用 NULL 值的列，指定 NULL 缺省值与不指定任何缺省值完全相同。如果插入行的客户端未显式指派值，该行会自动获得一个 NULL 值。

如果某些列的信息是可选的或者并不总有信息，而且并不要求数据库中的数据都是正确的，则可以使用 NULL 缺省值。

有关 NULL 值的详细信息，请参见 『ASA SQL 参考』> 「NULL 值」。

字符串和数字缺省值

只要某一列的数据类型为字符串或数字，就可以指定一个特定字符串或数字作为该列的缺省值。必须确保指定的缺省值可以转换为该列的数据类型。

缺省字符串和数字适用于具有典型条目的列。例如，如果公司有两个办事处，总部位于 **city_1**，而较小的办事处位于 **city_2**，则可能需要将地点列的缺省条目设置为 **city_1**，以便更容易地输入数据。

常数表达式缺省值

可以使用不参照任何数据库对象的常数表达式作为缺省值。使用常数表达式可以设置象*自今天起第十五天的日期*这样的列缺省值。要设置上述缺省值，应输入

```
... DEFAULT ( dateadd( day, 15, getdate() ) )
```

使用表和列约束

除基本表结构 （列的数量、名称和数据类型，该表的名称和位置）外，CREATE TABLE 语句和 ALTER TABLE 语句还可以指定许多不同的表属性，这些属性可用于控制数据完整性。约束允许您对可显示在某个列中的值或不同列中的值之间的关系施加限制。约束可以是表范围内的约束，也可以应用于各个列。

注意
变更表可能会影响到数据库的其他用户。虽然可以在其他连接处于活动状态时执行 ALTER TABLE 语句，但如果任何其他连接正在使用要变更的表，则不能执行 ALTER TABLE 语句。对于大型表， ALTER TABLE 操作可能需要很长时间。在处理该语句的过程中，所有其他引用正被变更的表的请求都将被禁止。

本节介绍如何使用约束来帮助确保表中数据的正确性。

对列使用 CHECK 约束

使用 CHECK 条件可以确保一列中的值满足某个明确的条件或规则。例如，可能只有使用这些规则或条件才能确保数据的合理性，或者可能需要使用更严格的规则来反映公司的政策和程序。

如果某列的有效值限制在一定范围内，则可以单独在该列上使用 CHECK 条件。

示例 1

- 可以实施特定的格式要求。例如，如果表中有一列用于存储电话号码，您可能希望确保用户以相同的格式输入电话号码。对于北美洲电话号码，可以使用类似以下约束：

```
ALTER TABLE customer
 MODIFY phone
 CHECK ( phone LIKE '(___) ___-____' )
```

示例 2

- 可以确保输入的值与一组有限数量的值中的一个相匹配。例如，要确保 **city** 列只包含几个允许城市 （例如，公司设有办事处的那些城市）中的一个，可以使用类似以下的约束：

```
ALTER TABLE office
 MODIFY city
 CHECK ( city IN ( 'city_1', 'city_2', 'city_3' )
)
```

- 缺省情况下，比较字符串时不区分大小写，除非创建数据库时显式指定需要区分大小写。

示例 3

- 可以确保日期或数字在特定的范围内。例如，您可能需要使用以下约束确保雇员表中的 **start_date** 列必须在公司成立日期和当前日期之间：

```
ALTER TABLE employee
 MODIFY start_date
 CHECK ( start_date BETWEEN '1983/06/27'
                    AND CURRENT DATE )
```

- 有几种日期格式可以使用。不管当前选项设置是什么，数据库都可以识别本例中使用的 YYYY/MM/DD 格式。

列 CHECK 测试只有在条件返回 FALSE 值时才会失败。如果条件返回的值为 UNKNOWN，则允许进行更改。

对表使用 CHECK 约束

作为约束应用于表上的 CHECK 条件通常确保正在添加或修改的行中的两个值之间具有正确的关系。

为约束指定名称时，约束单独保存在系统表中，您可以单独替换或删除它们。由于这种情况比较灵活，因此建议您尽可能命名 CHECK 约束或者使用单个的列约束。

例如，在图书馆数据库中， **date_borrowed** 必须在 **date_returned** 之前。

```
ALTER TABLE loan
 ADD CONSTRAINT valid_date CHECK(date_returned >=
date_borrowed)
```

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」。

从域继承列 CHECK 约束

您可以在域中附加 CHECK 约束。使用这些数据类型定义的列将继承 CHECK 约束。为列显式指定的 CHECK 约束将替换域中的 CHECK 约束。

任何使用 **posint** 数据类型定义的列都只接受正整数，除非为该列本身显式指定了 CHECK 约束。在下例中，该域只接受正整数。由于任何前缀为 @ 符号的变量都会在计算 CHECK 约束时被列名替换，因此，可以使用任何前缀为 @ 的变量名称代替 **@col**。

```
CREATE DATATYPE posint INT
 CHECK ( @col > 0 )
```

带有 DELETE CHECK 子句的 ALTER TABLE 语句将从表定义中删除所有 CHECK 约束，包括从域继承的那些 CHECK 约束。

在对域定义了列后，在该域定义中对约束进行的任何更改都*不会*应用到该列。创建列时，它会从该域获取约束，但是两者之间没有更深的关联。

有关域的详细信息，请参见 『ASA SQL 参考』> 「域」。

在 Sybase Central 中管理表和列约束

在 Sybase Central 中，可以在表或列属性表的 [约束] 选项卡上添加、变更和删除列约束。

❖ 管理约束

1. 打开 [表] 文件夹。
2. 在右窗格中，双击要更改的表。
3. 右窗格中具有代表唯一约束和检查约束的单独选项卡。
4. 对要修改的约束进行适当的更改。例如，要添加表或列约束，请单击 [检查约束] 选项卡，然后选择 [文件] → [新建] → [表检查约束] 或者 [文件] → [新建] → [列检查约束]。

修改和删除 CHECK 约束

有几种方法可以改变表中现有的 CHECK 约束集。

- 您可以给表或某个列添加新 CHECK 约束。
- 通过将列的 CHECK 约束设置为 NULL，可以删除该约束。例如，以下语句将删除 **customer** 表中 **phone** 列的 CHECK 约束：

```
ALTER TABLE customer
 MODIFY phone CHECK NULL
```

- 您可以用与添加 CHECK 约束相同的方法来替换列的 CHECK 约束。例如，以下语句在 **customer** 表中的 **phone** 列中添加或替换 CHECK 约束：

```
ALTER TABLE customer
 MODIFY phone
 CHECK ( phone LIKE '___-___-____' )
```

- 您可以修改在表中定义的 CHECK 约束：
  - 您可以使用带有 ADD *表约束*子句的 ALTER TABLE 添加新的 CHECK 约束。
  - 如果已经定义了约束名，您可以修改各个约束。
  - 如果尚未定义约束名，您可以使用 ALTER TABLE DELETE CHECK 删除所有现有的 CHECK 约束 （包括从域继承的列 CHECK 约束和其他 CHECK 约束），然后添加新的 CHECK 约束。

    使用带有 DELETE CHECK 子句的 ALTER TABLE 语句：

    ```
    ALTER TABLE table_name
     DELETE CHECK
    ```

使用 Sybase Central，您可以添加、修改和删除表 CHECK 约束和列 CHECK 约束。有关详细信息，请参见 「在 Sybase Central 中管理表和列约束」第 98 页。

从表中删除列并不会删除表约束中保存的与该列相关联的 CHECK 约束。如果不删除这些约束，尝试在该表中插入数据时 （甚至在只查询数据时）会出现以下错误消息：有一列未找到。

只有在返回的值为 FALSE 时，表 CHECK 约束才会失败。如果返回的值为 UNKNOWN 则可以进行更改。

使用域

域是指一个用户定义的数据类型，与其他属性一起使用时，它可以限制可接受值的范围或提供缺省值。域继承自内部数据类型之一。可允许值的范围通常由一个检查约束限制。此外，域还可以指定缺省值，而且可以允许空值，也可以不允许空值。

您可以出于不同的原因定义自己的域。

- 如果不允许输入不适当的值，则可以避免许多常见错误。在域中设置的约束可以确保所有用于保存所需范围内的值或以所需格式保存值的列和变量只能保存所需的值。例如，使用一种数据类型可以确保只能在数据库中键入位数正确的信用卡号。
- 利用域可以使应用程序和数据库结构更便于理解。
- 域使用起来更方便。例如，您可能希望所有表标识符都是正整数，并且在缺省情况下自动递增。您可以通过在每次定义新表时输入适当的约束和缺省值来实施这个限制，但如果定义一个新域，然后简单地声明标识符只能使用属于指定域的值，则更为简便。

有关域的详细信息，请参见 『ASA SQL 参考』> 「SQL 数据类型」。

创建域 (Sybase Central)

可以使用 Sybase Central 创建域，或将域指派到一列。

❖ **创建新域 (Sybase Central)：**

1. 在左窗格中，选择 [域] 文件夹。
2. 选择 [文件] → [新建] → [域]。

   [创建域] 向导即会出现。
3. 请按照向导中的说明进行操作。

所有域都显示在 Sybase Central 的 [域] 文件夹中。

❖ **将域指派到列 (Sybase Central)：**

1 对于所需的表，请在右窗格中单击 [列] 选项卡。

2 在所需列的数据类型列中，执行以下任一操作：

- 从下拉列表中选择域，或者
- 单击下拉列表旁边的按钮，并在属性表中选择域。

创建域 (SQL)

可以使用 CREATE DOMAIN 语句创建和定义域。

❖ **创建新域 (SQL)：**

1 连接到数据库。

2 执行 CREATE DOMAIN 语句。

示例 1：简单域

如果数据库中有些列用于存储人名，其他列用于存储地址，则可以键入以下语句来定义域。

```
CREATE DOMAIN persons_name CHAR(30)
 CREATE DOMAIN street_address CHAR(35)
```

定义了这些域后，则可以按照类似使用内部数据类型的方式使用它们。例如，可以执行以下语句，用这些定义来定义表。

```
CREATE TABLE customer (
    id INT  DEFAULT AUTOINCREMENT  PRIMARY KEY
    name persons_name
    address street_address
 )
```

示例 2：缺省值、检查约束和标识符

在上例中，该表的主键指定为整数类型。实际上，许多表都需要相似的标识符。但创建一个标识符域供这些应用程序使用，要比将主键指定为整数更为方便。

创建域时，可以指定一个缺省值，并提供检查约束，以确保不能将不适当的值键入任何属于这种类型的列。

通常使用整数值作为表标识符。正整数非常适合于用作唯一标识符。由于许多表中都可能会使用这样的标识符，因此可以定义以下域。

```
CREATE DOMAIN identifier INT
 DEFAULT AUTOINCREMENT
 CHECK ( @col > 0 )
```

这个检查约束使用变量 @col。您可以使用这个定义来重写 Customer 表，如下所示。

```
CREATE TABLE customer (
    id identifier PRIMARY KEY
    name persons_name
    address street_address
 )
```

示例 3：内部域

Adaptive Server Anywhere 附带一些预定义的域。可以按照使用自建域的方式使用这些预定义的域。例如，已经为您创建了以下货币域。

```
CREATE DOMAIN MONEY NUMERIC(19,4)
 NULL
```

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE DOMAIN 语句」。

删除域

可以使用 Sybase Central 或 DROP DOMAIN 语句来删除域。

只有 DBA 用户或创建域的用户可以删除该域。此外，如果数据库中的任何变量或列是某个域的实例，则不能删除该域，因此，首先需要删除所有属于该类型的列或变量，才能删除该域。

❖ 删除域 (Sybase Central)：

1　打开 [域] 文件夹。

2　右击要删除的域，然后从弹出式菜单中选择 [删除]。

❖ **删除域 (SQL)：**

1 连接到数据库。

2 执行 DROP DOMAIN 语句。

示例 以下语句将删除 customer_name 域。

```
DROP DOMAIN customer_name
```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

实施实体完整性和参照完整性

个人服务器通过数据库的关系结构识别数据库中的信息，并确保各表中的所有行都能保持表之间的关系 （在数据库结构中描述）。

实施实体完整性

用户插入或更新行后，数据库服务器将确保该表的主键仍有效：该表中的每一行都由主键唯一地标识。

示例 1

示例数据库中的 **employee** 表使用雇员 ID 作为主键。在该表中添加了一个新雇员后，数据库服务器将检查新雇员 ID 值是否唯一，以及是否非空。

示例 2

示例数据库中的 **sales_order_items** 表使用两个列来定义主键。

该表保存有关订购项目的信息。一列包含指定订单的 **id**，但每个订单可能涉及几个项目，因此该列本身不能作为主键。另一列 **line_id** 标识哪一行对应于该项目。将 **id** 列和 **line_id** 列结合起来可以唯一地指定一个项目，并形成主键。

如果客户端应用程序破坏了实体完整性

实体完整性要求主键的每个值在该表中都是唯一的，并且不能有空值。如果客户端应用程序尝试插入或更新一个主键值，但提供的值不唯一，则会破坏实体完整性。如果实体完整性被破坏，将导致不会将新信息添加到数据库中，而会向客户端应用程序发送一条错误消息。

应用程序编程人员应确定如何将该信息传达给用户，以便于用户采取适当的措施。适当的措施通常很简单，只需要求用户为主键提供另外一个唯一的值。

主键实施实体完整性

为每个表指定了主键后，客户端应用程序开发人员或数据库管理员不需要采取任何其他措施即可保持数据完整性。

表的所有者在创建表时为其定义了主键。如果他们以后要修改表的结构，也可以重新定义主键。

使用某些应用程序开发系统和数据库设计工具可以创建或变更数据库表。如果使用这样的系统，可能不必显式输入 CREATE TABLE 或 ALTER TABLE 语句：该应用程序可以自动使用您提供的信息生成这些语句。

有关创建主键的详细信息，请参见 「管理主键」第 49 页。有关 CREATE TABLE 语句的详细语法，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。有关更改表结构的信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」。

实施参照完整性

外键 （由一个特定列或列组合构成）将一个表 （**外**表）中的信息与另一个表 （**参照**表或**主**表）中的信息关联起来。要使外键关系有效，外键中的条目必须与参照表中一行的主键值相对应。有时，可能会参照其他某个唯一的列组合，而不参照主键。

示例 1

示例数据库包含一个雇员表和一个部门表。雇员表的主键是雇员 ID，而部门表中的主键是部门 ID。在雇员表中，部门 ID 称作部门表的**外键**，因为雇员表中的每个部门 ID 都严格对应于部门表中的部门 ID。

外键关系是一个多对一关系。雇员表中的几个条目都具有相同的部门 ID 条目，但部门 ID 是部门表的主键，因此在部门表中是唯一的。如果外键参照部门表中包含重复条目或空值条目的列，则无法获知部门表中的哪一行是适当的参照。这是一个强制外键。

示例 2

假设该数据库中还包含一个办事处表，列出了办事处的地点。雇员表中可能有一个办事处表的外键，显示雇员工作的办事处位于哪座城市。该数据库的设计人员可以选择在聘用雇员时不指派办事处地点，例如，或者因为尚未将新雇员分配到办事处，或者因为他们不在外面的办事处工作。在这种情况下，该外键可以允许空值，因此是可选外键。

外键实施参照完整性

与主键一样，可以使用 CREATE TABLE 或 ALTER TABLE 语句创建外键。一旦创建了外键，该键中的列就只能包含通过该外键关联的表中的主键值。

有关创建外键的详细信息，请参见 「管理主键」第 49 页。

丢失参照完整性

如果有人执行了以下操作，数据库可能会丢失参照完整性：

- 更新或删除主键值。所有参照该主键的外键都将变为无效。
- 在外表中添加一个新行，并为外键输入一个没有相应主键值的值。该数据库将不满足完整性要求。

Adaptive Server Anywhere 对这两类完整性丢失都提供保护。

如果客户端应用程序破坏了参照完整性

如果客户端应用程序在一个表中更新或删除了一个主键值，而数据库中其他的地方有一个外键参照该主键值，则会有破坏参照完整性的危险。

示例

如果服务器允许更新或删除该主键，并且未对参照该主键的外键进行任何变更，该外键将变为无效。任何使用该外键参照的尝试（例如，在 SELECT 语句中使用 KEY JOIN 子句）都将失败，因为参照表中没有相应的值。

由于 Adaptive Server Anywhere 防止破坏实体完整性的措施比较直接，只是简单地拒绝输入数据并返回一条错误消息，因此对参照完整性的潜在破坏变得更为复杂。有几种方式（称作参照完整性动作）可以帮助您保持参照完整性。

参照完整性动作

在更新或删除被参照的主键时保持参照完整性的操作非常简单，只需拒绝进行更新或删除。但常常也可以对每个外键执行具体的操作来保持参照完整性。数据库管理员和表的所有者可以使用 CREATE TABLE 和 ALTER TABLE 语句指定，在完整性被破坏时，对参照被修改的主键的外键执行哪些操作。

可以为更新和删除主键的情况分别指定各个可用参照完整性动作：

- **RESTRICT** 生成一条错误消息，并在尝试修改被参照的主键时，拒绝进行修改。这是缺省的参照完整性动作。
- **SET NULL** 将所有参照被修改的主键的外键设置为 NULL。
- **SET DEFAULT** 将所有参照被修改的主键的外键设置为该列的缺省值 （在表定义中指定的缺省值）。
- **CASCADE** 与 ON UPDATE 一起使用时，此动作将把所有参照被更新的主键的外键更新为新值。与 ON DELETE 一起使用时，此动作将删除所有包含参照被删除的主键的外键的行。

参照完整性动作由系统触发器执行。该触发器在主表上定义，但使用*辅助*表的所有者权限执行。这一行为意味着可以在所有者不同的表之间进行级联操作，而无需获得额外权限。

参照完整性检查

对于外键定义为 RESTRICT 的操作将破坏参照完整性时，则将在执行语句时进行缺省检查动作。如果指定了 CHECK ON COMMIT 子句，则只会在提交事务时进行该检查动作。

使用数据库选项控制检查时间

对于外键定义为 RESTRICT 的操作将破坏参照完整性时，则设置 WAIT_FOR_COMMIT 数据库选项可以控制该行为。使用 CHECK ON COMMIT 子句可以替换此选项。

如果 WAIT_FOR_COMMIT 设置为 OFF （缺省设置），将不允许执行会导致数据库不一致的操作。例如，尝试删除仍具有雇员的部门将会失败。以下语句将产生此错误消息：表 `'department'` 中行的主键在另一表中被引用：

```
DELETE FROM department
 WHERE dept_id = 200
```

把 WAIT_FOR_COMMIT 设置为 ON 将导致执行提交之前不对参照完整性进行检查。如果数据库处于不一致状态，将不允许执行提交操作，并报告一个错误。在此模式下，虽然数据库用户可以删除仍具有雇员的部门，但在执行以下操作之前该用户不能将更改提交到数据库：

- 删除或重新分配属于该部门的雇员。
- 重新按搜索条件执行 SELECT 语句，以选择破坏参照完整性的行。
- 将 **dept_id** 行插回到 **department** 表中。
- 回退事务，撤消 DELETE 操作。

系统表中的完整性规则

有关数据库完整性检查和规则的所有信息都存储在以下系统表中：

系统表	说明
SYS.SYSTABLE	SYS.SYSTABLE 的 **view_def** 列包含 CHECK 约束。对于视图， **view_def** 包含创建视图的 CREATE VIEW 命令。您可以通过查看 **table_type** 列 （是 BASE 或 VIEW）来检查一个特定表是基表还是视图。
SYS.SYSTRIGGER	SYS.SYSTRIGGER 包含参照完整性动作。 **referential_action** 列包含一个字符，该字符指出动作是级联 (C)、删除 (D)、设置为空值 (N)，还是限制 (R)。 **event** 列包含一个字符，该字符指定导致发生以下动作的事件：插入、删除 (A)、插入、更新 (B)、更新 (C)、删除 (D)、删除、更新 (E)、插入 (I)，更新 (U)、插入、删除、更新 (M)。 **trigger_time** 列说明是在发生触发事件之后 (A) 还是之前 (B) 执行该动作。
SYS.SYSFOREIGNKEYS	此视图以更便于阅读的格式显示 SYS.SYSFOREIGNKEY 和 SYS.SYSFKCOL 两个表中的外键信息。
SYS.SYSCOLUMNS	此视图以更便于阅读的格式显示 SYS.SYSCOLUMN 表中的信息，包括缺省设置和列的主键信息。

有关各个系统表的内容的详细信息，请参见 『ASA SQL 参考』> 「系统表」。可以使用 Sybase Central 或 Interactive SQL 浏览这些表和视图。

第 4 章

使用事务和隔离级别

关于本章

可将 SQL 语句组合成事务。事务具有以下属性：事务中的所有语句或者都执行，或者都不执行。每个事务的设计宗旨都是执行一项将数据库从一个一致状态更改为另一个一致状态的任务。

本章将介绍事务，以及如何在应用程序中使用事务。本章还将介绍如何在 Adaptive Server Anywhere 中设置隔离级别，以限制并发事务之间的干扰。

事务简介

为确保数据完整性，必须能确定数据库中的信息在哪些状态下是**一致的**。下面的示例可以很好地说明一致性的概念：

一致性示例

假设您使用数据库来管理财务帐户，并且要将资金从一位客户的帐户转到另一位客户的帐户上。转帐前后数据库都处于一致状态；但在将资金从一个帐户中划出尚未存入另一个帐户之前，数据库处于不一致状态。转帐过程中，当客户帐户上的总金额与转帐前的总金额相同时，数据库就处于一致状态。但在转帐过程进行到一半时，数据库处于不一致状态。取款和存款操作必须都执行，或都不执行。

事务是一个逻辑工作单元

事务是一个逻辑工作单元。每个事务都是一系列在逻辑上相关的命令，用于完成一项任务，并将数据库从一个一致状态转换到另一个一致状态。什么样的状态是一致状态取决于您的数据库。

事务内的语句被视为不可分的单元：或者所有语句都执行，或者都不执行。在每个事务的末尾，需要**提交**所做更改，将这些更改永久应用到数据库中。如果由于某种原因事务中的一些语句未正确处理，则任何中间更改都会被撤消，或**回退**。这种特性有另外一个说法，即事务是**原子的**。

将语句组合成事务对于保护数据的一致性 （即使在出现介质故障或系统故障时）和管理并发数据库操作都非常关键。可以安全地交错执行事务，并且每个事务的完成都标志着一个时间点，在该时间点数据库中的信息处于一致状态。

在正常运行期间，如果出现系统故障或数据库崩溃， Adaptive Server Anywhere 将在下次启动数据库时自动恢复数据。自动恢复过程将恢复所有已完成的事务，并回退在发生故障时任何尚未提交的事务。事务的原子特性可以确保将数据库恢复到一个一致状态。

有关数据库备份和数据恢复的详细信息，请参见 『ASA 数据库管理指南』> 「备份和数据恢复」。

有关并发数据库使用的详细信息，请参见 「并发简介」第 114 页。

使用事务

Adaptive Server Anywhere 期望您会将命令组合到事务中。如果您知道哪些命令或动作意味着事务的开始或结束，便可以充分地利用这项功能。

开始事务

事务起始于以下事件之一：

- 在连接到数据库后执行的第一个语句
- 在事务结束后执行的第一个语句

完成事务

以下事件之一将完成事务：

- 执行 COMMIT 语句使对数据库进行的更改成为永久更改。
- 执行 ROLLBACK 语句撤消由事务执行的所有更改。
- 执行具有自动提交副作用的语句：数据定义命令 （例如 ALTER、CREATE、 COMMENT 和 DROP）都具有自动提交副作用。
- 与数据库断开连接将执行隐式回退。
- ODBC 和 JDBC 具有一个自动提交设置，可以在每个语句后强制执行 COMMIT 命令。缺省情况下， ODBC 和 JDBC 要求将自动提交设置为 ON，并且要求每个语句都是一个事务。如果要利用事务设计的各种可能性，则应关闭自动提交设置。

  有关自动提交的详细信息，请参见 『ASA 编程指南』> 「设置自动提交或手动提交模式」。
- 将数据库选项 CHAINED 设置为 OFF 与在每个语句后强制自动提交类似。缺省情况下，那些使用 jConnect 或 Open Client 应用程序的连接已将 CHAINED 设置为 OFF。

  有关详细信息，请参见 『ASA 编程指南』> 「设置自动提交或手动提交模式」和 『ASA 数据库管理指南』> 「CHAINED 选项 [compatibility]」。

Interactive SQL 中的选项

使用 Interactive SQL 可以控制您的应用程序中的事务何时终止，以及如何终止：

- 如果您将 AUTO_COMMIT 选项设置为 ON，则 Interactive SQL 将自动在成功执行每一语句后提交您的结果，并且自动在每一失败的语句后执行 ROLLBACK。

- COMMIT_ON_EXIT 选项的设置控制退出 Interactive SQL 时对未提交的更改执行的操作。如果该选项设置为 ON（缺省设置），则 Interactive SQL 执行 COMMIT 语句；否则，它会用 ROLLBACK 语句撤消未提交的更改。

Adaptive Server Anywhere 也支持 Transact-SQL 命令 （例如 BEGIN TRANSACTION），以便和 Sybase Adaptive Server Enterprise 兼容。有关详细信息，请参见 「Transact-SQL 兼容性」第 503 页。

并发简介

并发是指数据库服务器同时处理多个事务的能力。如果数据库服务器中没有特殊的机制，并发事务可能会相互干扰，导致信息不一致或不正确。

示例

百货商店中的数据库系统必须允许多个店员同时更新顾客帐户。各个店员在协助每位顾客购物时必须能及时更新帐户的状态：他们不能等到没有其他人使用该数据库时再进行更新。

谁需要了解并发

并发是所有数据库管理员和开发人员都应考虑的问题。即使对于单用户数据库，如果要处理来自多个应用程序 （甚至一个应用程序的多个连接）的请求，也必须考虑并发问题。这些应用程序和连接之间也会发生干扰，其方式与网络环境下的多个用户完全相同。

事务大小影响并发

将 SQL 语句组合成事务的方式对数据完整性和系统性能有很大影响。如果事务太短，不包含完整的逻辑工作单元，则可能会导致数据库中出现不一致。如果事务太长，包含几个不相关的动作，则在执行 ROLLBACK 时很可能会不必要地撤消本可以安全提交到数据库中的工作。

如果事务很长，则会降低并发性能，因为这些事务会导致无法并发处理其他事务。

决定事务的长度的因素有许多，具体取决于应用程序的类型和环境。

事务内的保存点

使用**保存点**将各组相关语句分开可以在事务内标识重要的状态，并且可以返回到这些状态。

SAVEPOINT 语句定义事务执行过程中的中间点。使用 ROLLBACK TO SAVEPOINT 语句可以撤消该点后的所有更改。执行了 RELEASE SAVEPOINT 语句或事务结束后，则不能再使用该保存点。请注意，保存点对提交没有影响。当执行 COMMIT 时，该事务内的所有更改都在数据库中永久生效。

RELEASE SAVEPOINT 或 ROLLBACK TO SAVEPOINT 命令不会释放任何锁：只有在事务结束后才会释放锁。

命名和嵌套保存点

如果对保存点进行适当命名和嵌套，一个事务内就可以有多个活动保存点。 SAVEPOINT 和 RELEASE SAVEPOINT 之间的更改可以通过回退到前一个保存点或回退事务本身来取消。在提交事务之前，事务内的更改还没有永久地应用到数据库中。事务结束后，所有保存点都将被释放。

不能在批量操作模式下使用保存点。使用保存点的附加开销很少。

隔离级别和一致性

有四个隔离级别

使用 Adaptive Server Anywhere 可以控制一个事务中的操作对其他并发事务中的操作的可见程度，方法是设置名为**隔离级别**的数据库选项。

Adaptive Server Anywhere 提供了以下四种不同的隔离级别：

此隔离级别 …	具有这些特征 …
0- 未提交的读操作	◆ 允许读取有写锁定或无写锁定的行 ◆ 未应用读锁定 ◆ 无法确保并发事务将不会修改行或回退对行所做的更改
1- 已提交的读操作	◆ 只允许读取没有写锁定的行 ◆ 仅为读取当前行获取并保持读锁定，但当游标离开该行时释放读锁定 ◆ 无法确保数据在事务执行过程中不发生更改
2- 可重复的读操作	◆ 只允许读取没有写锁定的行 ◆ 读取结果集中的每一行时获取读锁定，并一直保持到事务结束为止
3- 可序列化	◆ 只允许读取结果中没有写锁定的行 ◆ 打开游标时获取读锁定，并一直保持到事务结束为止

这些隔离级别能阻止部分或全部干扰。级别 3 是最高隔离级别。较低的级别允许的不一致也较多，但通常性能会更高。级别 0 （未提交的读操作）为缺省设置。

注意

所有隔离级别都确保每个事务或者完全执行，或者根本不执行，并且不会丢失任何更新。

隔离仅在事务之间执行：同一事务中的多个游标可能互相干扰。

典型的不一致类型

有三种典型的不一致类型在执行并发事务的过程中可能会出现。这个列表并不完整，还可能会出现其他类型的不一致。 ISO SQL/92 标准中提到了这三种类型，而且，由于较低隔离级别的行为是依据这些类型定义的，因此它们非常重要。

- **脏读**　事务 A 修改了一行，但未提交或回退所做更改。事务 B 读取了被修改的行。之后，事务 A 在执行 COMMIT 之前又对该行进行了更改，或回退了所做更改。这两种情况下，事务 B 读取到该行时，该行都处于从未被提交的状态。
  - 有关脏读的详细信息，请参见 「脏读教程」第 131 页。
- **非可重复的读取**　事务 A 读取了一行。之后，事务 B 修改或删除了该行，并执行了 COMMIT 命令。如果事务 A 之后再次尝试读取同一行，该行将已被更改或删除。
  - 有关非可重复的读取的详细信息，请参见 「非可重复的读取教程」第 134 页。
- **幻像行**　事务 A 读取了一组满足某个条件的行。之后，事务 B 执行 INSERT 命令插入了一行，或对以前不满足 A 的条件的一行执行 UPDATE 命令。事务 B 提交了这些更改。这些新提交的行现在满足该条件。事务 A 之后重新执行最初的读取操作，并获得了不同的行集。
  - 有关幻像行的详细信息，请参见 「幻像行教程」第 138 页。

还可能有其他类型的不一致。 ISO SQL/92 标准选择了这三种不一致，因为它们是典型问题，而且从这三方面来描述事务之间的锁定量非常方便。

隔离级别与脏读、非可重复的读取，和幻像行

对于不同的隔离级别， Adaptive Server Anywhere 所允许的不一致行为也有所不同。 x 表示防止该行为发生，而 √ 表示该行为可以发生。

隔离级别	脏读	非可重复的读取	幻像行
0- 未提交的读操作	√	√	√

隔离级别	脏读	非可重复的读取	幻像行
1- 已提交的读操作	x	√	√
2- 可重复的读操作	x	x	√
3- 可序列化	x	x	x

此表说明了以下两点：

- 每个隔离级别都消除了这三种典型的不一致类型中的一种。
- 每个级别都消除了所有较低级别所消除的不一致类型。

这四个隔离级别在 ODBC 下具有不同的名称。这些名称是基于它们所防止的不一致的名称而命名的。有关说明，请参见 「ValuePtr 参数」第 121 页。

游标不稳定性

另一种重要的不一致是**游标不稳定性**。存在这种不一致时，一个事务可以修改正由另一事务的游标所引用的行。游标稳定性可以确保使用游标的应用程序不会导致数据库中的数据出现不一致。

示例

事务 A 使用游标读取了一行。事务 B 修改并提交了该行。在不知道该行已被修改的情况下，事务 A 对其进行了修改。

消除游标不稳定性

Adaptive Server Anywhere 在隔离级别 1、 2 和 3 上实现了**游标稳定性**。游标稳定性可以确保任何其它事务都不能对游标的当前行中包含的信息进行修改。游标的一行中的信息可能是一个特定表中所包含的信息的副本，也可能是多个表的不同行中的数据的组合。只要在 SELECT 语句中使用了连接或子选择，就可能会涉及多个表。

有关 SQL 编程过程与游标的信息，请参见 「使用过程、触发器和批处理」第 699 页。

只有在通过另一个应用程序使用 Adaptive Server Anywhere 时，才会使用游标。有关详细信息，请参见 『ASA 编程指南』> 「在应用程序中使用 SQL」。

对于使用游标的应用程序，需要考虑的一个相关但又独特的问题是，应用程序是否可以看到对基础数据的更改。通过指定游标的敏感性，可以控制应用程序能看到哪些更改。

有关游标敏感性的详细信息，请参见 『ASA 编程指南』> 「Adaptive Server Anywhere 游标」。

设置隔离级别

数据库的每个连接都有其自己的隔离级别。此外，数据库也可以为每个用户或组存储一个缺省隔离级别。使用 PUBLIC 设置可以为整个数据库的组设置一个缺省隔离级别。

您也可以用表提示来设置隔离层，但这是一个高级功能，只应在需要时使用它。有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」中的 WITH 表提示一节。

隔离级别是一个数据库选项。可以使用 SET OPTION 语句更改数据库选项。例如，以下命令将把当前用户的隔离级别设置为 3，该级别是最高级别。

```
SET OPTION ISOLATION_LEVEL = 3
```

使用 SET OPTION 命令可以更改您的连接的隔离级别，以及与您的用户 ID 相关联的缺省级别。如果有权限，还可以为其他用户或组更改隔离级别。

❖ 为当前用户 ID 设置隔离级别：

- 执行 SET OPTION 语句。例如，以下语句将把当前用户的隔离级别设置为 3：

  ```
  SET OPTION ISOLATION_LEVEL = 3
  ```

❖ 为用户或组设置隔离级别：

1　以具有 DBA 权限的用户身份连接到数据库。

2 在 ISOLATION_LEVEL 前加上该组的名称和一个圆点，然后执行 SET OPTION 语句。例如，以下命令将把特殊组 PUBLIC 的缺省隔离级别设置为 3。

```
SET OPTION PUBLIC.ISOLATION_LEVEL = 3
```

❖ **仅为当前会话设置隔离级别：**

- 使用 TEMPORARY 关键字执行 SET OPTION 语句。例如，以下语句将在您的连接期间把隔离级别设置为 3：

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
```

一旦您断开连接后，隔离级别将回复到其原有值。

缺省隔离级别

连接到数据库时，数据库服务器将按照以下方式确定您的初始隔离级别：

1. 可以为每个用户和组设置缺省隔离级别。如果您的用户 ID 在数据库中有一个存储的级别，数据库服务器将使用该级别。
2. 如果没有，数据库服务器将检查您所属的组，直到找到一个级别。所有用户都是特殊组 PUBLIC 的成员。如果之前未找到任何其他设置， Adaptive Server Anywhere 将使用指派给该组的级别。

有关用户和组的详细信息，请参见 『ASA 数据库管理指南』> 「管理用户 ID 和权限」。

有关 SET OPTION 语句语法的详细信息，请参见 『ASA SQL 参考』> 「SET OPTION 语句」。

有些情况下，可能需要在事务中间更改隔离级别，例如，在只有一个表或一组表需要序列化访问时。有关在事务内更改隔离级别的信息，请参见 「在事务内更改隔离级别」第 122 页。

从支持 ODBC 的应用程序设置隔离级别

ODBC 应用程序调用 **SQLSetConnectAttr**，调用时将 **Attribute** 设置为 SQL_ATTR_TXN_ISOLATION 并根据相应隔离级别设置 **ValuePtr**：

ValuePtr 参数

ValuePtr	隔离级别
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXT_SERIALIZABLE	3

通过 ODBC 更改隔离级别

您可以使用 *ODBC32.dll* 库中的 **SQLSetConnectOption** 函数通过 ODBC 更改连接的隔离级别。

SQLSetConnectOption 函数采用三个参数：ODBC 连接句柄的值，您希望设置隔离级别这一事实，以及与该隔离级别相对应的值。下表中显示了这些值。

字符串	值
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXT_SERIALIZABLE	8

请不要使用 SET OPTION 语句从 ODBC 应用程序中更改隔离级别。由于 ODBC 驱动程序不分析语句，所以，在 ODBC 中执行任何语句都不会被 ODBC 驱动程序识别。这可能会导致意外的锁定行为。

示例

以下函数调用会将连接 MyConnection 的隔离级别设置为隔离级别 2：

```
SQLSetConnectOption( MyConnection.hDbc,
                     SQL_TXN_ISOLATION,
                     SQL_TXN_REPEATABLE_READ )
```

ODBC 使用隔离功能支持各种数据库锁选项。例如，在 PowerBuilder 中，可以使用事务对象的 Lock 属性在连接到数据库时设置隔离级别。Lock 属性是一个字符串，按以下方式设置：

```
SQLCA.lock = "RU"
```

Lock 选项仅在 CONNECT 发生时起作用。在 CONNECT 之后再更改 Lock 属性不会对连接产生任何影响。

在事务内更改隔离级别

有时，您可能会发现不同的隔离级别适用于一个事务内的不同部分。Adaptive Server Anywhere 支持在事务中间更改数据库的隔离级别。

注意
你也可以使用表提示设置隔离级别，但是这是一个高级功能，只应在需要时使用它。有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」中的 WITH 表提示一节。

如果在事务中间更改了 ISOLATION_LEVEL 选项，新设置只会影响以下对象：

- 任何在更改后打开的游标
- 任何在更改后执行的语句

您可能希望在事务处理过程中更改隔离级别，因为这样做可以对事务放置的锁数量进行控制。有的事务可能需要读取一个大型表，但却仅对其中少数几行执行具体的操作。如果不一致对该事务不会产生严重影响，则可以在扫描该大型表时设置较低的隔离级别，以避免延迟其他事务的操作。

在其他一些情况下，您也可能希望在事务中间更改隔离级别，例如，在只有一个表或一组表需要序列化访问时。

有关在事务中间更改隔离级别的示例，请参见 「幻像行教程」第 138 页。

查看隔离级别

使用 CONNECTION_PROPERTY 函数可以检查当前连接的隔离级别。

❖ **查看当前连接的隔离级别：**

- 执行以下语句：

```
SELECT CONNECTION_PROPERTY('ISOLATION_LEVEL')
```

事务阻塞和死锁

执行事务时，数据库服务器将在行上放置锁，以防止其他事务干扰被更改的行。**锁**用于控制可允许的干扰程度和类型。

Adaptive Server Anywhere 利用**事务阻塞**机制使事务能够并发执行而不会发生干扰，或只有很少的干扰。一个事务能获取一个锁，以防止其他并发事务修改（甚至访问）特定的行。这种事务阻塞方案总能避免某些类型的干扰。例如，当一个事务更新表中的某一特定行时，它将锁定该行以确保任何其他事务都不能同时更新或删除同一行。

事务阻塞

当一个事务尝试执行某个操作，但却因另一个事务持有的锁而被禁止时，将出现冲突，并且该事务尝试执行其操作的进程将被阻碍。

「两阶段锁定」第 155 页介绍了死锁，当两个或多个事务相互阻塞以致所有事务都不能继续执行时，将出现这种死锁情况。

有时，一组事务会进入一种特殊状态，在该状态下这些事务都不能继续执行。有关详细信息，请参见 「死锁」第 125 页。

BLOCKING 选项

如果两个事务在同一行上分别获得了读锁定，当其中一个事务尝试修改该行时，所发生的情况将取决于数据库的 BLOCKING 设置。要修改这一行，该事务必须阻塞另一个事务，但由于另一个事务阻塞它，因此无法实现。

- 如果 BLOCKING 设置为 ON （缺省设置），则尝试写入的事务将等到另一个事务释放其读锁定。那时，即可进行写入。
- 如果 BLOCKING 被设置为 OFF，则尝试写入的语句将收到一个错误。

当 BLOCKING 被设置为 OFF 时，该语句将终止而不会等待，而它所做的任何部分更改都将被回退。在这种情况下，请稍后再尝试执行该事务。

隔离级别越高，越容易出现阻塞，因为执行的锁定和检查更多。通常，隔离级别越高，并发性越低。并发性降低的程度取决于各个并发事务的特性。

有关 BLOCKING 选项的详细信息，请参见 『ASA 数据库管理指南』> 「BLOCKING 选项 [database]」。

死锁

事务阻塞可能会导致**死锁**。出现死锁情况时，一组事务将进入一种特殊状态，在该状态下这些事务都不能继续执行。

死锁的原因

有两个原因会导致死锁：

- **循环阻塞冲突**　事务 A 被事务 B 阻塞，而事务 B 又被事务 A 阻塞。显然，等待并不能解决这个问题。必须取消其中一个事务，使另一个事务能够继续执行。如果多个 （大于二）事务出现循环阻塞，也会导致这种情况。
- **所有活动数据库线程均被阻塞**　事务被阻塞时，不会释放其数据库线程。如果数据库配置了三个线程，而事务 A、 B 和 C 在当前未执行请求的事务 D 上受到阻塞，由于没有可用线程，将出现死锁情况。

Adaptive Server Anywhere 将自动回退最后一个被阻塞的语句 （消除死锁情况）并向该事务返回一个错误，指出发生哪种死锁。

服务器使用的数据库线程数取决于各个数据库的设置。

有关设置数据库线程的数量的信息，请参见 『ASA 数据库管理指南』> 「控制线程行为」。

确定被阻塞的连接

您可以使用 sa_conn_info 系统过程来确定哪些连接在其他哪些连接上受到阻塞。该过程将返回一个结果集，每个连接对应于该结果集中的一行。该结果集中有一列显示连接是否受到阻塞，如果是，则指出它在其他哪个连接上受到阻塞。

有关详细信息，请参见 『ASA SQL 参考』> 「sa_conn_info 系统过程」。

数据库服务器使用 LOG_DEADLOCKS 选项和 sa_report_deadlocks 系统过程提供更详细的死锁报告。当打开 LOG_DEADLOCKS 选项时，数据库服务器会在内部缓冲区中记录有关阻塞的连接的信息。

❖ 使用死锁报告：

1 打开 LOG_DEADLOCKS 选项。

```
SET OPTION PUBLIC.LOG_DEADLOCKS='ON'
```

数据库服务器会在内部缓冲区中记录有关死锁的信息。

有关详细信息，请参见 『ASA 数据库管理指南』> 「LOG_DEADLOCKS 选项 [database]」。

2 用 sa_report_deadlocks 检索死锁信息。

```
CALL sa_report_deadlocks()
```

有关详细信息，请参见 『ASA SQL 参考』> 「sa_report_deadlocks 系统过程」。

选择隔离级别

选择哪个隔离级别取决于应用程序要完成哪类任务。本节提供了一些有关选择隔离级别的指导。

选择隔离级别时，必须在对一致性和准确性的需求与不受阻碍地执行并发事务的需求之间找到平衡点。如果一个事务只涉及一个表中的一两个特定值，而另一个事务搜索许多大型表并且可能需要锁定许多行或整个表，并且可能需要很长时间才能完成，那么前者对其他进程的干扰通常要比后者少。

例如，如果您的事务涉及在银行帐户间转帐，或者甚至需要检查帐户余额，则必须尽可能地确保返回的信息正确。相反，如果只需要粗略地估计不活动帐户所占的比例，则可能不会介意该事务是否需要等待其他事务，而且甚至愿意牺牲一些准确性，以避免干扰该数据库的其他用户。

此外，转帐可能只会影响包含两个帐户余额的那两行，而要计算不活动帐户的比例则必须读取所有帐户。因此，转帐延迟其他事务的可能性更小。

Adaptive Server Anywhere 提供了四个隔离级别：级别 0、 1、 2 和 3。级别 3 提供完全隔离，并确保以可序列化调度的方式交错执行事务。

可序列化调度

要并发处理事务，数据库服务器必须执行一个事务中的部分语句，然后执行其他事务的一些语句，然后再继续处理第一个事务中后面的操作。交错执行各个事务中部分操作的顺序称作**调度**。

以这种方式并发地应用多个事务可能会产生多种结果，包括上文中介绍的三种特殊的不一致情况。有时，也可能达到数据库的最终状态，就象依次执行各个事务时那样，意思是说，始终在一个事务全部处理完成之后再开始处理下一个事务。只要在按某个顺序依次执行事务时，数据库达到的状态与实际的调度相同，那么这个调度就叫做**可序列化**调度。

有关 Adaptive Server Anywhere 如何处理序列化的详细信息，请参见「两阶段锁定」第 155 页。

可序列化性是判断正确性的公认标准。我们认为可序列化调度是正确的，因为数据库没有因并发执行事务而受到影响。

隔离级别会影响事务的可序列化性。在隔离级别 3 上，所有调度都是可序列化的。缺省设置是 0。

可序列化意味着并发不会产生附加影响

即使在依次执行事务时，数据库的最终状态也取决于执行这些事务的顺序。例如，如果一个事务将某个特定单元设置为值 5，而另一个事务将其设置为数字 6，那么该单元的最终值将由最后执行哪个事务确定。

知道调度可序列化并不能确定按哪个顺序执行事务最好，而只能说明并发不会产生附加影响。按某种顺序依次执行一组事务得到的所有结果都被认为是正确的。

不可序列化的调度将会造成不一致

如果调度不可序列化，将出现多种不一致问题。「典型的不一致类型」第 117 页中介绍的不一致就是其中的典型问题。在各种情况下，出现不一致的原因都在于，交错执行语句的方式产生了在依次执行所有事务时不可能产生的结果。例如，只有在这种情况下才会发生脏读：一个事务可以选择某些行而同时另一个事务正在同一行中插入数据或更新数据时。

各个隔离级别的典型事务

各个隔离级别分别适用于特定类型的任务。以下信息可以帮助您确定各个特定操作适合使用哪个级别。

级别 0 的典型事务

涉及浏览或输入数据的事务可能需要几分钟时间才能完成，并且需要读取大量行。如果使用隔离级别 2 或 3，可能会降低并发性。对于这类事务，通常使用隔离级别 0 或 1。

例如，对于需要从数据库中读取大量信息以生成统计摘要的决策支持应用程序，即使读取到的几行数据以后被修改，可能也不会对其产生很大影响。如果要求这样的应用程序使用较高的隔离级别，它可能会在大量数据上获取读锁定，从而导致其他应用程序无法写入数据。

级别 1 的典型事务

隔离级别 1 特别适合与游标一起使用，因为这种组合既可以确保游标稳定性，又不会大幅提高锁定需求。 Adaptive Server Anywhere 通过较早释放在游标的当前行上获取的读锁定来实现这一优点。在级别 2 和级别 3 上，这些锁定必须持续到事务结束以确保可重复的读取。

例如，通过游标更新存货量的事务特别适合使用这个级别，因为随着进货或卖出对存货量进行的每次调整都不会丢失，而这些频繁调整对其他事务的影响又可以降低到最小程度。

级别 2 的典型事务

处于隔离级别 2 时，其他事务不能更改符合您的条件的行。因此，如果必须多次读取行并且需要第一个结果集中包含的行不会更改，则可以使用这个级别。

由于需要相对较大数量的读锁定，使用这个隔离级别时应谨慎。与级别 3 的事务一样，改进数据库和索引的设计可以减少获取的锁数目，从而大大提高数据库的性能。

级别 3 的典型事务

隔离级别 3 适合于将安全性放在首位的事务。由于消除了幻像行，因此可以对行集放心地执行多步操作，而不必担心在操作过程中因出现新行导致结果损坏。

虽然隔离级别 3 可以提供很高的完整性，但在需要支持大量并发事务的大型系统中应谨慎使用。 Adaptive Server Anywhere 在这一级别上放置的锁数目比任何其他级别都多，因此也增大了一个事务阻碍对其他许多事务进行处理的可能性。

提高隔离级别 2 和 3 的并发性

隔离级别 2 和 3 使用大量锁定，因此，对于经常使用这些隔离级别的数据库而言，优良的设计非常重要。如果必须使用可序列化事务，设计数据库时，特别是在设计索引时，应牢记项目的业务规则，这一点非常重要。还可以将大事务分割成几个较小的事务，从而缩短锁定行的时间，这样便可提高性能。

虽然可序列化事务最有可能阻塞其他事务，但其效率并不一定低。处理这些事务时， Adaptive Server Anywhere 可以执行某些优化。虽然增加了锁数目，但这些优化可能仍会使性能得到提高。例如，由于无论是否符合搜索条件，所有读取的行都必须被锁定，因此数据库服务器可以自由地合并读取行的操作和放置锁的操作。

减小锁定的影响

应尽量避免以隔离级别 3 运行事务。这些事务可能会放置大量的锁，因此将影响其他并发事务的执行效率。

如果某个操作的性质要求它必须以隔离级别 3 运行，则可以通过将查询设计为读取尽量少的行和索引条目来减少该操作对并发性的影响。这些步骤有助于提高级别 3 事务的运行速度，而且，还有助于减少放置的锁数目，这一点可能更为重要。

特别是，您可能会发现添加索引会大大提高事务的运行速度，特别是在其中至少有一个事务必须以隔离级别 3 运行时。索引有两个优点：

- 使用索引可以更有效地对行进行定位
- 利用索引的搜索操作需要的锁数目可能比较少。

有关 Adaptive Server Anywhere 使用的锁定方法的详细信息，请参见「锁定的工作方式」第 146 页。

有关性能和 Adaptive Server Anywhere 如何计划其信息访问以执行命令的详细信息，请参见「监控和提高性能」第 167 页。

隔离级别教程

不同隔离级别的行为有很大不同，应使用哪个级别取决于您的数据库和要执行的操作。以下一组教程可以帮助您确定不同的任务适合使用哪个隔离级别。

脏读教程

以下教程举例说明并发执行多个事务时可能会出现的一种不一致情况。一家小商品销售公司的两位雇员同时访问公司的数据库。第一个人是公司的销售经理。第二个人是会计。

销售经理想要将公司销售的 T 恤衫的价格提高 $0.95，但在 SQL 语言的语法方面有点儿小问题。同时，会计正要计算当前存货的零售额，以便将它包含到他自愿在下次召开管理会议时提交的报告中。但销售经理却不知道这个情况。

提示：
对数据库进行以下变更之前，为谨慎起见，建议使用 SELECT 代替 UPDATE 对更改进行测试。

在本示例中，您将扮演这两个人的角色，而且两个人都并发使用演示数据库。

1. 启动 Interactive SQL。
2. 以销售经理的身份连接到示例数据库：
   - 在 [连接] 对话框中，选择 ODBC 数据源 ASA 9.0 Sample。
   - 在 [高级] 选项卡上的 [连接名字段] 中键入以下字符串，以便于找到此窗口：

     ConnectionName=Sales Manager

   单击 [确定] 进行连接。
3. 再启动一个 Interactive SQL 的实例。
4. 以会计的身份连接到示例数据库：

- 在 [连接] 对话框中，选择 ODBC 数据源 ASA 9.0 Sample。
- 在 [高级] 选项卡上的 [连接名字段] 中键入以下字符串，以便于找到此窗口：

  ConnectionName=Accountant

- 单击 [确定] 进行连接。

5. 作为销售经理，将所有 T 恤衫的价格提高 $0.95：

   - 在标题为 Sales Manager 的窗口中，执行以下命令：

     ```
     SELECT id, name, unit_price
      FROM product;
      UPDATE PRODUCT
      SET unit_price = unit_price + 95
      WHERE NAME = 'Tee Shirt'
     ```

   结果为：

id	name	unit_price
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
…	…	…

   您很快就发现应输入 0.95 而不是 95，但在还未能改正错误之前，会计在另一个办公室访问了数据库。

6. 公司的会计担心存货会占用太多资金。作为会计，执行以下命令计算所有库存商品的零售额：

   ```
   SELECT SUM( quantity * unit_price )
      AS inventory
    FROM product
   ```

   结果为：

inventory
21453.00

可惜的是，计算出的结果并不准确。由于销售经理不小心将 T 恤衫的价格提高了 $95，而结果是用错误的价格计算得到的。这个错误演示了一种被称作**脏读**的典型不一致类型。作为会计，您访问了销售经理已经输入但尚未提交的数据。

您可以清除脏读和 「隔离级别和一致性」第 116 页中介绍的其它不一致。

7. 作为销售经理，回退第一次所做的更改，然后输入正确的 UPDATE 命令，即可改正错误。检查新值是否正确。

```
ROLLBACK;
 UPDATE product
 SET unit_price = unit_price + 0.95
 WHERE NAME = 'Tee Shirt';
```

id	name	unit_price
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
…	…	…

8. 会计并不知道他计算出的金额不正确。您可以在他的窗口中再次执行他的 SELECT 语句查看正确的值。

```
SELECT SUM( quantity * unit_price )
   AS inventory
 FROM product;
```

inventory
6687.15

9. 在销售经理的窗口中完成事务。她应该输入 COMMIT 语句以将所做更改永久应用到数据库中，但您应输入 ROLLBACK，以避免更改您计算机上的演示数据库的副本。

```
ROLLBACK;
```

由于数据库服务器并发处理销售经理的工作和会计的工作，因此会计在不知情的情况下从数据库中得到了错误的信息。

非可重复的读取教程

「脏读教程」第 131 页中的示例演示了第一类不一致 （即脏读）。在该示例中，会计对零售额进行了计算，而销售经理正在更新一个价格。会计的计算使用了销售经理已经输入但正在进行改正的错误信息。

以下示例演示了另一类不一致：非可重复的读取。在本示例中，您还将扮演这两个人的角色，而且两个人都并发使用演示数据库。销售经理想要给塑料太阳帽定一个新的销售价格。会计想要核实某些出现在最近一份订单上的商品的价格。

本例开始时两个连接都处于隔离级别 1，而不是隔离级别 0 （随 Adaptive Server Anywhere 提供的演示数据库的缺省隔离级别）。将隔离级别设置为 1 可以消除上一教程中演示的不一致类型 （即脏读）。

1. 启动 Interactive SQL。
2. 以销售经理的身份连接到示例数据库：
   - 在 [连接] 对话框中，选择 ODBC 数据源 ASA 9.0 Sample。
   - 在 [高级] 选项卡上，输入以下字符串，以便于找到此窗口：

     ConnectionName=Sales Manager

   - 单击 [确定] 进行连接。
3. 再启动一个 Interactive SQL 的实例。

4. 以会计的身份连接到示例数据库：
   - 在 [连接] 对话框中，选择 ODBC 数据源 ASA 9.0 Sample。
   - 在 [高级] 选项卡上，输入以下字符串，以便于找到此窗口：

     ConnectionName=Accountant

   - 单击 [确定] 进行连接。
5. 执行以下命令，将会计的连接的隔离级别设置为 1。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1
```

6. 在销售经理的窗口中执行以下命令，将隔离级别设置为 1：

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1
```

7. 会计决定列出太阳帽的价格。作为会计，执行以下命令。

```
SELECT id, name, unit_price FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
…	…	…

8. 销售经理决定为塑料太阳帽定一个新的销售价格。作为销售经理，执行以下命令：

```
SELECT id, name, unit_price FROM product
 WHERE name = 'Visor';
 UPDATE product
 SET unit_price = 5.95 WHERE id = 501;
 COMMIT;
```

id	name	unit_price
500	Visor	7.00
501	Visor	5.95

9. 比较 [Sales Manager] 窗口中太阳帽的价格和 [Accountant] 窗口中相同太阳帽的价格。虽然销售经理已经输入了新价格并提交了更改，但 [Accountant] 窗口中显示的仍然是旧价格。

   这种不一致被称作**非可重复的读取**，因为如果会计在*同一事务*中再次执行相同的 SELECT 命令，则不会得到相同的结果。您自己可以试试看。作为会计，再次执行该 SELECT 命令。注意，现在销售经理输入的销售价格将显示出来。

```
SELECT id, name, price
 FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
…	…	…

   当然，如果会计已经完成了他的事务，例如，在再次使用 SELECT 之前发出了 COMMIT 或 ROLLBACK 命令，情况则有所不同。该数据库可供多个用户同时使用，而且完全允许某人在会计执行事务之前或之后更改值。结果中的变化仅仅因为发生在执行其事务的过程中才不一致。这种情况将导致调度不可序列化。

10. 会计注意到了这种情况，并决定从现在开始不希望看到价格发生更改。隔离级别 2 消除了可重复的读取。作为会计，执行以下命令：

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
 SELECT id, name, unit_price
 FROM product
```

11. 销售经理决定最好将塑料太阳帽的销售推迟到下个星期，这样，她就不必为预计明天会收到的一个大订单报那个较低的价格。在她的窗口中，尝试执行以下语句。该命令将开始执行，然后她的窗口将看起来似乎被冻结一样。

```
UPDATE product
 SET unit_price = 7.00
 WHERE id = 501
```

数据库服务器在隔离级别 2 上必须确保可重复的读取。为此，它在会计读取的产品表的每一行上都放置了一个读锁定。当销售经理尝试将价格更改回原来的值时，她的事务必须在产品表中包含塑料太阳帽的那一行上获取一个写锁定。由于写锁定是排它的，因此她的事务必须等到会计的事务释放其读锁定后才能继续执行。

12. 会计查看完价格后，由于他不想无意中更改了数据库，因此使用 ROLLBACK 语句完成他的事务。

```
ROLLBACK
```

注意，在数据库执行完这条语句后，销售经理的事务就算完成。

id	name	unit_price
500	Visor	7.00
501	Visor	7.00

13. 销售经理现在就可以完成她的工作了。她希望提交她所做的更改，恢复到原来的价格。

```
COMMIT
```

锁的类型和不同的隔离级别

将会计的隔离级别从级别 1 升级到级别 2 后，数据库服务器将在以前未获取任何锁定的位置上使用读锁定。一般来说，各个隔离级别所需的锁类型和处理其他事务持有的锁的方式各不相同。

处于隔离级别 0 时，数据库服务器只需要写锁定。它使用这些锁来确保任何两个事务都无法完成相互冲突的修改。例如，级别 0 的事务在更新或删除一行之前在该行上获取一个写锁定，而插入新行时，这些新行上已经具有写锁定。

级别 0 的事务对它们读取的行不进行任何检查。例如，当一个级别 0 的事务读取一行时，它不会检查其他事务是否已在该行上获取了何种锁。由于不需要进行任何检查，因此级别 0 的事务特别快。速度快是以牺牲一致性为代价的。如果读取的行已被其他事务执行了写锁定，就有返回脏数据的危险。

使用级别 1 的事务在读取行之前将检查该行是否已被写锁定。虽然需要多执行一项操作，但这些事务可以确保读取的所有数据都是已提交的数据。请尝试重复第一个教程，将隔离级别设置为 1（而不是 0），然后您会发现，在销售经理更新 T 恤衫价格的事务保持未完成状态时，无法进行会计计算。

当会计将其隔离级别提高到级别 2 后，数据库服务器开始使用读锁定。从这时开始，数据库服务器将在符合会计的选择条件的每一行上为他的事务获取一个读锁定。

事务阻塞

在上面的教程中，[Sales Manager] 窗口在执行她的 UPDATE 命令时被冻结。数据库服务器开始执行她的命令，之后发现会计的事务已在销售经理需要更改的行上获取了一个读锁定。这时，数据库服务器只是暂停该 UPDATE 命令的执行。在会计使用 ROLLBACK 完成了他的事务之后，数据库服务器自动释放了他的锁定。发现不再有任何阻碍后，数据库服务器将继续执行销售经理的 UPDATE 命令。

一般来说，当一个事务试图在另一个事务持有锁的行上获取排它锁时，或当一个事务试图在另一个事务持有排它锁的行上获取共享锁时，将出现锁定冲突。这个事务必须等待另一个事务完成才能继续。我们说，那个必须等待的事务被另一个事务**阻塞**。

当数据库服务器发现导致某个事务无法立即继续执行的锁定冲突时，它可以或者暂停执行该事务，或者终止该事务，回退所有更改，并返回一个错误。可以通过设置 BLOCKING 选项来控制其行为。当 BLOCKING 被设置为 ON 时，第二个事务将等待，如上面的教程所述。

有关阻塞选项的详细信息，请参见 「BLOCKING 选项」第 124 页。

幻像行教程

以下教程继续使用相同的背景情况。在本例中，会计查看部门表时，销售经理创建了一个新部门。您会发现将出现一个幻像行。

如果您尚未执行上一教程（「非可重复的读取教程」第 134 页）中的第 1 步到第 4 步，请执行这些步骤，打开两个 Interactive SQL 实例。

1. 在 [Sales Manager] 窗口中执行以下命令，将隔离级别设置为 2。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

2. 在 [Accountant] 窗口中执行以下命令，将隔离级别设置为 2。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

3. 在 [Accountant] 窗口中输入以下命令，列出所有部门。

```
SELECT * FROM department
 ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
…	…	…

4. 销售经理决定设立一个新部门，专门管理国外市场。 Philip Chin（emp_id 为 129）将作为这个新部门的经理。

```
INSERT INTO department
    (dept_id, dept_name, dept_head_id)
    VALUES(600, 'Foreign Sales', 129);
 COMMIT;
```

   最后一个命令为这个新部门创建了一个新条目。这个条目作为新的一行出现在销售经理窗口中的表的底部。

5. 但是，会计并不知道这个新部门。处于隔离级别 2 时，数据库服务器通过放置锁来确保所有行都不会发生更改，但却不会通过放置锁来防止其他事务插入新行。

会计只有再次执行 SELECT 命令才能发现新行。在会计的窗口中，再次执行 SELECT 语句。您将看到新行已附加到表中。

```
SELECT *
 FROM department
 ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
500	Shipping	703
…	…	…

出现的新行被称作**幻像行**，这是因为，从会计的角度来看，该行就象幻影那样，不知从何而来。会计的连接处于隔离级别 2。在该级别上，数据库服务器只在会计使用的行上获取锁，而对其他行则不进行任何控制，因此，没有任何限制阻止销售经理插入新行。

6. 会计希望在将来避免这种奇异的事情发生，因此他将其当前事务的隔离级别提高到级别 3。为会计输入以下命令。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
 SELECT *
 FROM department
 ORDER BY dept_id
```

7. 而销售经理想要再增加一个部门来处理针对大型企业合作伙伴的销售业务。在销售经理的窗口中执行以下命令：

```
INSERT INTO department
    (dept_id, dept_name, dept_head_id)
 VALUES(700, 'Major Account Sales', 902)
```

销售经理的窗口在执行该命令的过程中将会暂停，因为会计放置的锁阻止了该命令。单击工具栏上的 [中断 SQL 语句] 按钮，（或从 [SQL] 菜单中选择 [停止]），中断这次输入。

8. 为避免更改随 Adaptive Server Anywhere 附带的演示数据库，应回退旨于增加新部门的插入操作。在销售经理的窗口中执行以下命令：

```
ROLLBACK
```

在会计将他的隔离级别提高到级别 3 并再次选择了部门表中的所有行之后，数据库服务器在该表中的每一行上都放置了防插入锁定，并且放置了一个额外的幻像锁以防止在该表的末尾插入新行。当销售经理尝试在该表的末尾插入新行时，就是这最后一个锁阻塞了她的命令。

注意，即使销售经理的连接仍处于隔离级别 2，但销售经理的命令也会被阻塞。数据库服务器按照每个事务的隔离级别和语句的要求放置防插入锁定 （与读锁定类似）。一旦放置这些锁定，所有其他并发事务必须都遵从。

有关锁定的详细信息，请参见 「锁定的工作方式」第 146 页。

锁定的实际含义教程

下面继续使用相同的背景情况。在本教程中，会计和销售经理都有涉及销售订单表 (sales order) 和销售订单商品表 (sales order items) 的任务。会计需要核实根据销售雇员在 2001 年 4 月的销售情况付给他们的佣金支票的金额。销售经理发现有几个订单没有添加到数据库中，想要添加这些订单。

他们的工作演示了幻像锁定。**幻像锁定**是一个共享锁，放置在带有索引的扫描位置上以防止幻像行。当使用隔离级别 3 的事务选择了符合给定条件的行时，数据库服务器将放置防插入锁定，以防止其他事务插入也符合给定条件的行。替您放置的锁的数量取决于搜索条件和数据库的设计。

如果尚未执行上一教程中的第 1 步到第 3 步，请执行这些步骤。这些步骤介绍了如何启动两个 Interactive SQL 实例。

1. 在 [Sales Manager] 窗口和 [Accountant] 窗口中执行以下语句，将其隔离级别都设置为 2。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2
```

2. 每个月都会向销售代表支付佣金。所付佣金是他们在该月的销售额乘以一定的百分。会计正在准备 2001 年 4 月的佣金支票。他的第一项任务是计算各个销售代表在该月的总销售额。

在会计的窗口中输入以下命令。价格、销售订单信息和雇员数据存储在单独的表中。使用外键关系连接这些表，以将所需信息合并到一起。

```
SELECT emp_id, emp_fname, emp_lname,
    SUM(sales_order_items.quantity * unit_price)
       AS "April sales"
 FROM employee
    KEY JOIN sales_order
    KEY JOIN sales_order_items
    KEY JOIN product
 WHERE '2001-04-01' <= order_date
    AND order_date < '2001-05-01'
 GROUP BY  emp_id, emp_fname, emp_lname
```

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

3. 销售经理发现由 Philip Chin 拿到的一个大订单没有输入数据库。Philip 希望能立即拿到佣金，因此销售经理输入了这个遗漏的订单。该订单是在 4 月 25 日拿到的。

   在销售经理的窗口中，输入以下命令。将销售订单和各项商品输入到单独的表中，因为一个订单可能包含多项商品。应先为销售订单创建条目，然后再将各项商品添加到该订单中。为保持参照完整性，只有在订单已经存在的情况下，数据库服务器才允许将各项商品添加到该订单中。

```
INSERT into sales_order
 VALUES ( 2653, 174, '2001-04-22', 'r1',
       'Central', 129);
 INSERT into sales_order_items
 VALUES ( 2653, 1, 601, 100, '2001-04-25' );
 COMMIT;
```

4. 会计不可能知道销售经理刚添加了一个新订单。如果早些将这个新订单添加到数据库中，计算 Philip Chin 四月份的销售额时将包括这个订单。

   在会计的窗口中再次计算四月份的总销售额。使用相同的命令进行计算，您将发现 Philip Chin 在四月份的销售额变为 $4560.00。

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

   假设会计现在对四月份拿到的所有订单作出标记，以表示已支付了佣金。虽然销售经理刚输入的订单没有计入 Philip 四月份的总销售额中，但在第二次搜索时可能会找到该订单并将其标记为佣金已付！

5. 处于隔离级别 3 时，数据库服务器放置防插入锁定以确保任何其他事务都不能添加符合搜索或选择条件的行。

   首先，回退插入 Philip 的那份遗漏的订单的操作：在销售经理的窗口中，执行以下语句：

```
ROLLBACK
```

6. 在会计的窗口中，执行以下两个语句。

```
ROLLBACK;
 SET TEMPORARY OPTION ISOLATION_LEVEL = 3;
```

7. 在销售经理的窗口中，执行以下语句，删除那个新订单。

```
DELETE
 FROM sales_order_items
 WHERE id = 2653;
 DELETE
 FROM sales_order
 WHERE id = 2653;
 COMMIT;
```

8. 在会计的窗口中，执行与以前相同的查询。

```
SELECT emp_id, emp_fname, emp_lname,
    SUM(sales_order_items.quantity * unit_price)
       AS "April sales"
 FROM employee
    KEY JOIN sales_order
    KEY JOIN sales_order_items
    KEY JOIN product
 WHERE '2001-04-01' <= order_date
    AND order_date < '2001-05-01'
 GROUP BY  emp_id, emp_fname, emp_lname
```

由于已将隔离级别设置为 3，因此数据库服务器将自动放置防插入锁定，以确保销售经理在会计完成其事务之前不能插入四月份的订单商品项。

9. 返回到销售经理的窗口。再次尝试输入 Philip Chin 的那份遗漏的订单。

```
INSERT INTO sales_order
 VALUES ( 2653, 174, '2001-04-22',
          'r1','Central', 129)
```

销售经理的窗口将会挂起；该操作无法完成。单击工具栏上的 [中断 SQL 语句] 按钮，（或从 [SQL] 菜单中选择 [停止]），中断这次输入。

10. 虽然销售经理不能输入四月份的订单，但您可能会认为她还可以输入五月份的订单。

将该命令的日期更改为 5 月 5 日，然后重试。

```
INSERT INTO sales_order
 VALUES ( 2653, 174, '2001-05-05', 'r1',
       'Central', 129)
```

销售经理的窗口将再次挂起。单击工具栏上的 [中断 SQL 语句] 按钮，（或从 [SQL] 菜单中选择 [停止]），中断这次输入。虽然除防止插入所必需的锁以外数据库服务器没有放置多余的锁，但这些锁也可能会干扰大量其他事务。

数据库服务器在表索引中放置锁。例如，如果它在一个索引中放置了幻像锁定，那么就不能在该索引的前一位置插入新行。但是，如果没有合适的索引，它必须锁定表中的每一行。

有些情况下，防插入锁定可能会阻塞某些在表中插入行的操作，但允许其他一些插入操作。

11. 销售经理想要在编号为 2651 的订单中再添加一项商品。可使用以下命令。

```
INSERT INTO sales_order_items
 VALUES ( 2651, 2, 302, 4, '2001-05-22' )
```

一切都顺利完成，因此销售经理决定也在编号为 2652 的订单中添加以下商品。

```
INSERT INTO sales_order_items
 VALUES ( 2652, 2, 600, 12, '2001-05-25' )
```

销售经理的窗口将挂起。单击工具栏上的 [中断 SQL 语句] 按钮，（或从 [SQL] 菜单中选择 [停止]），中断这次输入。

12. 在本教程的最后，请撤消所有更改，以避免更改演示数据库。在销售经理的窗口中输入以下命令。

```
ROLLBACK
```

在会计的窗口中输入相同的命令。

```
ROLLBACK
```

现在可以关闭这两个窗口了。

锁定的工作方式

在数据库服务器处理事务时，可以锁定表中的一行或多行。锁可以通过防止其他事务并发访问来保持数据库中存储的信息的可靠性。锁还可以通过标识正在进行更新的信息来提高查询结果的准确性。

数据库服务器会自动放置这些锁，而无需显式指定。它将一直保留事务获取的所有锁，直到该事务完成（例如，通过使用 COMMIT 或 ROLLBACK 语句），但有一个例外。这个例外在 「较早释放读锁定」第 156 页中进行了说明。

可以访问该行的事务被称为持有锁。其他事务也许可以对被锁定的行进行有限的访问，也许根本不能访问，这取决于锁的类型。

获取有关表上的锁定的信息

您可以使用 sa_locks 系统过程列出有关数据库中保留的锁的信息。有关详细信息，请参见 『ASA SQL 参考』> 「sa_locks 系统过程」。

也可以在 Sybase Central 中查看锁。请在左窗格中选择数据库，然后右窗格中会出现名为 [表锁] 的选项卡。这个选项卡会显示每个锁的连接 ID、用户 ID、表名、锁类型和锁名。

可以锁定的对象

Adaptive Server Anywhere 会在以下对象上放置锁。

- **表中的行** 事务可以锁定特定的一行，以防止其他事务更改该行。如果一个事务要修改某行，必须在该行上放置写锁定。
- **行之间的插入点** 事务通常使用由索引确定的顺序对行进行扫描，或按顺序对行进行扫描。在这两种情况下，都可以将锁放置在扫描位置上。例如，在索引中放置锁可以防止其他事务插入具有特定值或特定值范围的行。
- **表模式** 事务可以锁定表的模式，以防止其他事务修改该表的结构。

在这些对象中，行是最直观的。读取、更新、删除或插入行的事务应限制其他事务对该行的同时访问，这是可以理解的。同样，更改表结构（例如，插入新的一列）的事务也会对其他事务产生很大影响。在这种情况下，必须限制其他事务进行访问以防止出现错误。

行排序

可以使用索引根据构建该索引时建立的特定条件对行进行排序。

如果没有索引， Adaptive Server Anywhere 将按照行在磁盘上的物理位置对其进行排序。在按顺序进行扫描时，具体顺序由数据库服务器的内部机制确定。在按顺序进行的扫描中不应依靠行的顺序。然而，从对行进行扫描的角度来说，虽然 Adaptive Server Anywhere 使用自己选择的顺序，但它处理请求的方式与处理带索引的扫描的方式相似。在扫描过程中，它可以在各个位置放置锁，就象使用索引时那样。

通过锁定扫描位置，一个事务可以防止其他事务按照行的顺序执行某些与特定值范围相关的操作。在扫描位置上总会放置插入锁和防插入锁。

例如，一个事务可能会删除一行，因此就会删除一个特定的主键值。在该事务提交更改或回退更改之前，必须确保它能够执行这两项操作。如果删除了一行，该事务必须确保任何其他事务都不能使用相同的主键值插入一行，从而可以确保能够执行回退。锁定该行占据的扫描位置可以保留这个权利，同时对其他事务的影响也最小。

锁的类型

Adaptive Server Anywhere 使用四种不同类型的锁在事务之间实施其锁定方案并确保事务间适当的隔离级别：

- **读锁定**　（共享）
- **幻像锁定或防插入锁定**　（共享）
- **写锁定**　（排它）
- **防幻像锁定或插入锁定**　（共享）

请注意，数据库服务器将自动放置这些锁，而无需显式指定。

这些锁中的每一种都有单独的用途，而且全部一起工作。每种锁都能防止一类特定的不一致。如果没有这些锁，就可能会出现这些不一致。根据您选择的隔离级别，数据库服务器将使用这些锁中的一些或全部来保持您所要求的一致性等级。

上述锁类型具有以下用途：

- 事务在插入、更新或删除行时将获取**写锁定**。设置了写锁定后，任何其他事务都不能在同一行上获取读锁定或写锁定。写锁定是排它的。

- 事务在读取一行时可以获取**读锁定**。几个事务可以在同一行上获取读锁定 （读锁定是共享锁，也被称作非排它锁）。在某一行被放置了读锁定后，任何其他事务都不能在该行上获取写锁定。因此，事务可以通过获取读锁定来确保任何其他事务都不能修改或删除一行。

- **防插入锁定** （也被称作**幻像锁定**）放置在扫描位置上以防止幻像行。它可以防止其他事务在表中被放置了防插入锁定的行与前一行之间插入新行。对于使用索引的搜索，防插入锁定要求在读取的每一行上放置一个读锁定，并放置一个额外的读锁定以防止在结果集末尾的索引处插入行。对于不使用索引的搜索，幻像锁定要求在表中的所有行上放置读锁定，以防止插入操作变更结果集，因此可能会对并发性产生不良影响。

- **插入锁定** （也被称作**防幻像锁定**）放置在扫描位置上以保留插入行的权利。一个事务在某一行上获取了插入锁定后，任何其他事务都不能在同一行上获取防插入锁定。获取插入锁定的同时，总会在相应的行上获取读锁定，以确保任何其他进程都不能更新或销毁该行，从而无法回避该插入锁定。

Adaptive Server Anywhere 根据需要使用这四类锁定来确保您所要求的一致性级别。您不需要显式请求使用某个特定的锁定，而应按照下一节中的说明控制一致性级别。了解各种锁类型有助于您选择隔离级别和理解各个级别对性能的影响。

排它锁与共享锁

这四类锁每种都可归为以下两个类别之一：

- **排它锁** 一次只能有一个事务在表中的某一行上持有排它锁。只要一个事务在一行上持有某种类型的锁定，任何其他事务都不能在同一行上获取排它锁。一个事务获取了排它锁后，其他事务要求锁定该行的请求都将被拒绝。

  写锁定是排它的。

- **共享锁** 任意数量的事务都可以同时在任意一行上获取共享锁。共享锁有时也被称作非排它锁。

  读锁定、插入锁定和防插入锁定都是可以共享的。

一次只能有一个事务更改任意一行。否则，两个同时执行的事务可能会尝试将一个值更改为两个不同的新值。因此，写锁定应该是排它的。

相反，如果多个事务想要读取一行，则不会出现任何问题。因为所有事务都不会更改该行，所以也不会发生冲突。因此，读锁定可以是共享的。

可以对防插入锁定和插入锁定进行相似的推理。多个事务可以通过各自获取一个防插入锁定来防止在特定的扫描位置插入行。同样的逻辑也适用于插入锁定。当某个特定事务需要进行排它访问时，只需在同一行上同时获取一个防插入锁定和一个插入锁定即可实现排它访问。这些锁定由同一个事务持有时，它们之间不会发生冲突。

具体哪些锁定会发生冲突？

下表显示了哪两个锁定之间会发生冲突。

	读	写	防插入	插入
读		冲突		
写	冲突	冲突		
防插入				冲突
插入			冲突	

只有在锁定是由不同的事务持有的时，才会出现这些冲突。例如，一个事务可以在一个扫描位置上同时获取防插入锁定和插入锁定，以获取对该位置的排它访问。

查询过程中的锁定

Adaptive Server Anywhere 在用户输入 SELECT 语句时使用的锁定取决于该事务的隔离级别。

以隔离级别 0 执行的 SELECT 语句

以隔离级别 0 执行 SELECT 语句时，不需要任何锁定操作。数据库服务器不对各个事务进行保护以防止其他事务更改数据。程序员或数据库用户在解读这些查询的结果时应考虑到这个限制。

以隔离级别 1 执行的 SELECT 语句

如果告诉您 Adaptive Server Anywhere 在以隔离级别 1 运行事务时使用的锁定几乎不比以隔离级别 0 运行时多，您可能会很吃惊。实际上，数据库服务器只以两种方式修改其操作。

操作中的第一个区别与获取锁定无关，而与是否遵从这些锁定有关。处于隔离级别 0 时，事务可以读取任意一行，无论其他事务是否在该行上获取了写锁定。相反，处于隔离级别 1 的事务在读取每一行之前必须检查该行是否已被放置了写锁定。它不能读取任何被放置了写锁定的行，因为这样做可能会导致读取脏数据。

操作中的第二个区别使游标具有稳定性。游标稳定性是通过在游标的当前行上获取一个读锁定来实现的。移动游标时，这个读锁定将被释放。如果游标中包含连接的结果，则可能会影响多行。在这种情况下，数据库服务器将在所有为游标的当前行提供信息的行上获取读锁定，并在选择了游标的另一行作为当前行后释放这些锁定。

以隔离级别 2 执行的 SELECT 语句

处于隔离级别 2 时， Adaptive Server Anywhere 修改其过程以确保可重复的读取。如果您的 SELECT 命令从表中的每一行返回值，那么，数据库服务器将在读取表中的每一行时在该行上获取一个读锁定。如果您的 SELECT 语句包含 WHERE 子句，或其他会对要选择的行加以限制的条件，那么，数据库服务器将读取每一行，测试该行中的值是否符合条件，如果符合，则在该行上获取一个读锁定。

与所有隔离级别一样，在级别 2 获取的锁定包括那些在级别 1 和级别 0 设置的所有锁定。因此，级别 2 也确保了游标稳定性，并且不允许脏读。

以隔离级别 3 执行的 SELECT 语句

以隔离级别 3 工作时， Adaptive Server Anywhere 必须确保所有调度均可序列化。特别是，除达到各个较低级别的要求外，还必须消除幻像行。

为达到这个要求，数据库服务器使用了读锁定和防插入锁定。当您进行选择时，数据库服务器在为结果集提供信息的每一行上获取一个读锁定。通过这样做，可以确保任何其他事务在您使用完那些信息之前都不能对其进行修改。

这个要求与数据库服务器在以隔离级别 2 运行时使用的过程相似，但区别在于必须在读取的每一行上获取锁定，*无论该行是否符合任何附带的条件*。例如，如果您要选择销售部门中所有雇员的名字，无论以隔离级别 2 还是隔离级别 3 执行该事务，服务器都必须锁定所有包含销售人员信息的行。但是，以隔离级别 3 执行时，服务器还必须在每个*不*属于销售部门的雇员的相应行上获取读锁定。否则，访问该数据库的其他某个人可以在您仍在使用您的结果时将另一个雇员转到销售部门。

必须在每一行上获取一个读锁定而不管该行是否符合您的条件，这一要求有两个重要含义。

- 数据库服务器需要放置的锁数目可能要比隔离级别 2 所需的锁数目要多得多。
- 数据库服务器的运转效率可能会稍微提高一些：它可以在读取每一行后立即在该行上获取读锁定，因为无论该行中的信息是否被接受都必须放置锁定。

服务器放置的防插入锁的数量可能会有很大不同，这取决于您的条件和该表中的可用索引。假设您要选择雇员 ID 为 123 的雇员的相关信息。如果雇员 ID 是雇员表的主键，那么数据库服务器就可以提高其操作效率。它可以使用索引（它是自动为主键构建的）更高效地对行进行定位。此外，其他事务也不可能将另一个雇员的 ID 更改为 123，因为主键值必须是唯一的。服务器只需在包含有关该雇员的信息的那一行上获取一个读锁定，即可确保不会将该 ID 编号分配给其他雇员。

相反，如果您要选择销售部门中的所有雇员，数据库服务器则需要获取更多锁定。由于可以在该部门中添加任意多个雇员，因此数据库可能必须读取雇员表中的每一行，并测试每个人是否属于销售部门。如果属于这种情况，则必须在每一行上同时获取读锁定和防插入锁定。

插入过程中的锁定

INSERT 操作将创建新行。 Adaptive Server Anywhere 使用以下过程来确保数据完整性。

有关在插入过程中如何使用锁定的详细信息，请参见 「防插入锁定」第 154 页。

1. 在内存中指定一个位置用于存储新行。最初，数据库的其余部分不知道该位置，因此目前还无需考虑其他事务可能访问该位置的问题。
2. 将所提供的任何值填入新行。
3. 在新行上放置写锁定。
4. 在要插入该行的表中放置一个插入锁定。上文中已经讲过插入锁定与防插入锁定是矛盾的，因此，获取了该插入锁定后，任何其他事务都不能通过获取防插入锁定来阻塞这个插入操作。
5. 将该行插入表中。现在，其他事务第一次可以看到这一新行存在。它们不能修改或删除该行，因为早先已经获取了写锁定。

6. 更新所有受影响的索引，并校验参照完整性和唯一性 （如果适当）。校验参照完整性的意思是指，应确保任何外键所指向的主键都应该存在。主键值必须是唯一的。也可以将其他列定义为只包含唯一的值。如果这样的列存在，则需要校验唯一性。

7. 在事务日志文件中记录该操作并释放所有锁定，只要不违反参照完整性，就可以提交该事务。

8. 根据要求插入其他行，并将触发器触发 （如果选择了层叠选项）。

唯一性

可以确保特定的一列或一组列中的所有值都是唯一的。数据库始终通过为唯一列构建索引 （即使您没有显式创建）来执行此任务。

特别是，所有主键值都必须是唯一的。数据库服务器自动为每个表的主键构建索引。因此，不应请求数据库服务器在主键上创建索引，因为这样做是多余的。

遗孤和参照完整性

通常，一个外键会参照另一个表中的一个主键。如果该主键不存在，孤立的外键被称作**遗孤**。 Adaptive Server Anywhere 会自动确保您的数据库中不包含任何遗孤。这个过程被称作**校验参照完整性**。数据库服务器通过计算遗孤数来校验参照完整性。

WAIT FOR COMMIT

可以请求数据库服务器将参照完整性校验延迟到事务结束时。利用这种方式，可以插入包含外键的一行，然后再插入包含丢失的主键的另一行。必须在同一事务中执行这两个操作。否则，数据库服务器将不允许执行您的操作。

要请求数据库服务器将参照完整性检查延迟到提交时，请将 WAIT_FOR_COMMIT 选项的值设置为 ON。缺省情况下，此选项被设置为 OFF。要将该选项打开，请发出以下命令：

```
SET OPTION WAIT_FOR_COMMIT = ON;
```

提交事务之前，数据库服务器将通过检查您的事务产生的遗孤数来校验是否保持了参照完整性。在每个事务结束时，遗孤数必须为零。

即使在插入该行时所需的主键存在，数据库服务器也必须确保在提交结果时它还存在。方法是，在目标行上放置一个读锁定。放置了这个读锁定后，任何其他事务仍可以读取该行，但都不能删除或变更该行。

更新过程中的锁定

数据库服务器使用以下过程修改一条特定记录中包含的信息。

1. 在受影响的行上放置写锁定。
2. 如果索引中包含更改的条目，则请删除与旧值相对应的各个索引条目。记录此操作所产生的任何遗孤。
3. 更新受影响的每个值。
4. 如果索引值被更改，添加新索引条目。校验唯一性 （如果适当）并校验参照完整性 （如果更改了主键或外键）。
5. 在事务日志文件中记录该操作，包括该行中所有条目的原有值，并释放所有锁定，只要不违反参照完整性，就可以提交该事务。
6. 如果选择了层叠选项并且主键或外键受到了影响，则执行层叠插入或删除操作。

您可能会惊讶地发现，象在表中更改值这样简单的操作实际可能需要执行大量操作才能完成。如果您要更改的值不属于主键或外键，数据库服务器需要完成的工作量会少得多。如果更改的值不包含在索引中 （此索引或显式创建，或因为您已将该属性声明为唯一的而隐式创建），需要完成的工作量还会更少。

UPDATE 操作过程中的参照完整性校验操作并不比 INSERT 过程中的校验操作简单。实际上，更改主键的值时，就可能会产生遗孤。插入替换值时，数据库服务器必须再次检查是否产生了遗孤。

删除过程中的锁定

DELETE 操作的步骤与 INSERT 操作的步骤几乎相同，只是顺序相反。

1. 在受影响的行上放置写锁定。
2. 删除与该行中的值相对应的各个索引条目。在马上就要删除各个索引条目之前，根据需要获取一个或多个防插入锁定，以防止其他事务在这个删除操作提交之前插入类似的条目。为校验参照完整性，数据库服务器还会跟踪该删除操作产生的遗孤。
3. 从表中删除该行，这样其他事务就无法再看到该行。在提交该事务之前不能销毁该行，因为这样做之后将无法再回退该事务。

4. 在事务日志文件中记录该操作，包括该行中所有条目的值，释放所有锁定，并销毁该行，只要不违反参照完整性，就可以提交该事务。

5. 如果选择了层叠选项并修改了主键或外键，则请层叠该删除操作。

防插入锁定

数据库服务器必须确保可以回退 DELETE 操作。为达到这一目的而采取的一项措施就是获取防插入锁定。这些锁定不是排它的，但它们可以使其他事务不能插入将导致无法回退 DELETE 操作的行。例如，删除的行可能包含主键值，或其他唯一值。如果允许其他事务插入具有相同值的行，则无法在不违反唯一性属性的前提下撤消 DELETE 操作。

Adaptive Server Anywhere 通过索引实施唯一性约束。对于具有单属性主键的简单表，使用一个幻像锁定就足够了。其他情况可能会导致所需锁数目迅速增加。例如，该表可能没有主键或其他与表中任何属性相关联的索引。由于表中的行没有任何基础排序标准，防止插入操作的唯一方法就是在整个表上放置防插入锁定。

删除一行可能意味着需要获取大量锁定。可以通过多种方法将它对数据库中并发性的影响减至最小。如上文中所述，索引和主键可以减少需要的锁数目，因为它们会对表中的行强制进行排序。数据库服务器会自动利用这些排序。它可以直接锁定*下*一行，而不必在表中的每一行上获取锁定。如果没有索引，行就没有顺序，因此下一行这个概念也就没有意义了。

数据库服务器在被删除的行的下一行上获取防插入锁定。如果删除了表的最后一行，数据库服务器则直接在不可见的末尾行上放置一个防插入锁定。实际上，如果表不包含索引，所需的防插入锁定数目就是表中的行数加一。

防插入锁定和读锁定

虽然一个或多个防插入锁定会排除插入锁定，而一个或多个读锁定会排除写锁定，但是防插入锁定 / 插入锁定和读锁定 / 写锁定之间不存在交互。例如，虽然不能在包含读锁定的行上获取写锁定，但可以在只具有防插入锁定的行上获取写锁定。这种灵活的安排为数据库服务器提供了更多选择，但这也意味着服务器必须经常注意在获取防插入锁定的同时也应获取读锁定。否则，其他事务将可以删除该行。

两阶段锁定

通常，上文中提供的有关锁定的一般信息应该足够满足您的需要。但有时，了解更多有关数据库服务器内部机制的知识对您执行基本类型的操作也会有所帮助。从这些基础知识出发，您可以更好地理解和预测数据库用户可能会遇到的潜在问题。

两阶段锁定对于确保调度可序列化非常重要。**两阶段锁定协议**指定了各个事务都应遵循的过程。

该协议之所以重要的原因在于，如果所有事务都遵循该协议，它可以确保可序列化调度，因此也是正确的调度。它还可以帮助您理解为什么有些锁定方法允许某些类型的不一致。

两阶段锁定协议

1. 在对任何行进行操作之前，事务必须在该行上获取锁定。
2. 释放锁定之后，事务必须不再获取任何锁定。

实际上，事务通常一直持有锁定，直到以 COMMIT 或 ROLLBACK 语句结束事务为止。如果在事务结束之前释放锁定将导致无法执行回退更改的操作，只要回退更改，就必须对行进行操作，将其返回到以前的状态。

两阶段锁定协议包含以下重要法则：

两阶段锁定法则
如果所有事务都遵守两阶段锁定协议，那么所有可能的交错执行调度都是可序列化的。

换言之，如果所有事务都遵循两阶段锁定协议，那么上述所有不一致都不会出现。

该协议定义了确保数据完全一致性所需要执行的操作，但您可以决定在对数据库执行某些操作时允许出现一些类型的不一致。如果消除所有不一致，通常也会导致数据库的效率降低。

无论使用哪个隔离级别，在修改、插入和删除的行上都要放置写锁定。这些锁定在提交或回退之前一直存在。

不同隔离级别的读锁定

隔离级别	读锁定
0	无
1	放置在结果集中出现的行上，只有在有游标位于某一行时才持有这些锁定
2	放置在结果集中出现的行上，在用户执行 COMMIT 或 ROLLBACK 命令之前，这些锁定一直存在。
3	在计算结果集时放置在读取的所有行和所有插入点的交叉位置

有关详细信息，请参见 「可序列化调度」第 127 页

锁定的细节可以划分为两部分：执行 INSERT、 UPDATE、 DELETE 或 SELECT 时会出现什么情况；以及各个隔离级别对读锁定、防插入锁定和插入锁定的放置有哪些影响。

虽然可以通过设置隔离级别来控制数据库服务器内发生的锁定量，但所有级别上都会发生许多锁定，即使级别 0 也一样。这些锁定操作是必要的。例如，一个事务更新了一行后，在该事务完成之前，任何其他事务都不能修改同一行。如果没有这种预防措施，则无法回退第一个事务。

数据库服务器在隔离级别 0 上执行的锁定操作应该首先学习，因为这些操作是基础。其他级别增加了锁定功能，但没有去掉任何较低级别提供的功能。因此，转到较高的隔离级别将增加较低的级别中没有的操作。

较早释放读锁定

处于隔离级别 3 时，事务在它读取的每一行上获取一个读锁定。通常，事务从不在其结束之前释放锁定。实际上，如果要使调度可序列化，事务就不能过早释放锁定。

Adaptive Server Anywhere 总是将写锁定一直保留到事务完成时。如果它过早释放了锁定，其他事务则可能会修改该行，从而导致无法回退第一个事务。

只有在一种特殊情况下才会释放读锁定。处于隔离级别 1 时，事务仅在某一行成为游标的当前行时，才在该行上获取读锁定。但是，处于隔离级别 1 时，当该行不再是当前行后，锁定将被释放。这一行为是可以接受的，因为数据库服务器在隔离级别 1 上不需要确保可重复的读取。

有关隔离级别的详细信息，请参见 「选择隔离级别」第 127 页。

特殊优化

以上几节介绍了当所有事务以某个给定隔离级别运行时获取的锁定。例如，当所有事务都以隔离级别 2 运行时，执行的锁定如上面相应章节中所述。

实际上，数据库可能需要以不同的级别来处理多个事务。有些事务（例如，在帐户间转帐）必须是可序列化的，因此以隔离级别 3 运行。而对于其他操作 （例如，更新地址或计算日平均销售额），使用较低的隔离级别就足够了。

尽管数据库服务器并不都以级别 3 处理所有事务，但是它会对某些操作进行优化以提高性能。特别是，许多额外的防插入锁定和插入锁定通常对支持级别 3 事务是必要的。在某些情况下，如果没有级别 3 的事务，数据库服务器可以避免放置或检查某些类型的锁定。

例如，数据库服务器使用防插入锁定防止以下两类不同的情况：

1. 确保在具有唯一属性的表中执行的删除操作可以回退。

2. 在级别 3 的事务中消除幻像行。

如果没有级别 3 的事务使用某个特定的表，那么数据库服务器就不需要在不包含唯一属性的表的索引中放置防插入锁定。但是，即使只有一个级别 3 的事务，所有事务 （即使是处于级别 0 的事务）也都必须放置防插入锁定，只有这样，级别 3 的事务才能识别它们的操作。

当然，数据库服务器在尝试进行上述优化时总会在表中附加说明。假如一个级别 3 的事务突然开始执行，您也可以确信会为其放置好必要的锁定。

您几乎无法控制同时混用隔离级别的情况，因为有太多因素取决于数据库的各个用户希望执行的特定操作。但是，应尽量减少执行级别 3 的操作的次数，因为这些操作可能会导致数据库运行的速度大幅下降。其影响是相当大的，因为数据库服务器被强制对较低级别的操作也执行额外的操作。

特殊并发问题

本节将讨论以下特殊并发问题：

- 「主键生成」第 159 页
- 「数据定义语句和并发」第 160 页

主键生成

您可能会遇到数据库应自动生成唯一编号的情况。例如，如果您正在构建一个用于存储销售发票的表，您可能希望数据库自动指派唯一的发票编号，而不希望让销售人员随意指定。

有许多方法可以生成这种编号。

示例

例如，可以通过在前一个发票编号上加 1 来获取发票编号。如果有多人同时在数据库中添加发票，则不能使用这种方法。两个人可能会决定使用相同的发票编号。

这个问题有多种解决方法：

- 为每个添加新发票的人指派一个发票编号范围。

  您可以通过创建一个具有用户名列和发票编号列的表来实施这种方案。每个添加发票的用户在该表中都将有一行。用户每次添加发票时，该表中的编号将递增，并用于新添加的发票。为处理数据库中的所有表，该表应具有三列：表名、用户名和最后的键值。应定期检查每个人是否还有足够的编号可用。

- 创建一个具有以下两列的表：表名和最后的键值。

  此表中有一行包含使用的最后一个发票号。每次有人添加发票时，将建立一个新连接，递增该表中的编号，然后立即提交更改。递增后的编号可以用于新添加的发票。其他用户仍可以取得发票编号，因为您使用了一个单独的事务来更新该行，而且这个事务很快就已完成。

- 使用带有 NEWID 的缺省值及 UNIQUEIDENTIFIER 二进制数据类型的列可生成全局唯一标识符。

UUID 和 GUID 值可用于唯一标识表中的行。在一台计算机上所生成的值与在其它计算机上生成的值不相同。因此，可将它们用作复制和同步环境中的键。

有关生成唯一标识符的详细信息，请参见 「NEWID 缺省值」第 93 页。

- 使用缺省值为 AUTOINCREMENT 的列。

  例如，

```
CREATE TABLE orders (
    order_id INTEGER NOT NULL DEFAULT
AUTOINCREMENT,
    order_date DATE,
    primary key( order_id )
 )
```

  每次在表中插入行时，如果没有为自动增量列指定值，将生成一个唯一的值。如果指定了一个值，将使用指定的值。如果指定的值大于该列的当前最大值，将使用指定的值作为以后插入行时自动递增的起点。自动增量列中最近插入的行的值可以用全局变量 **@@identity** 表示。

复制数据库中的唯一值
如果您复制了数据库并且有多人可以添加以后必须合并的条目，则需要使用不同的方法。有关详细信息，请参见 「复制和并发」第 162 页。

数据定义语句和并发

有些数据定义语句（例如， CREATE INDEX、 ALTER TABLE 和 TRUNCATE TABLE）可以更改整个表，只要该表当前正被另一连接使用，就无法完成这些语句。这些数据定义语句可能需要很长时间，而且数据库服务器在处理这些命令的过程中，将不处理引用相同表的请求。

CREATE TABLE 语句不会造成任何并发冲突。

GRANT 语句、 REVOKE 语句 和 SET OPTION 语句也不会造成并发冲突。这些命令会影响任何发送到数据库服务器的新 SQL 语句，但不会影响现有的未完成的语句。

不能对已连接到数据库的用户执行 GRANT 和 REVOKE 命令。

数据定义语句和同步的数据库

在使用同步的数据库中使用数据定义语句时应特别谨慎。有关详细信息，请参见另外两本手册 『MobiLink 管理指南』> 「MobiLink 管理指南」和 『SQL Remote 用户指南』> 「SQL Remote 用户指南」。

复制和并发

您的网络上有些计算机可能是便携式计算机。人们可能会将这些便携式计算机从办公室中拿走，或者，这些计算机可能偶尔才连接到该网络上。这些计算机的用户可能希望在未连接到网络的时候也能使用某些数据库应用程序。

数据库复制是这个问题的最理想的解决方案。使用 SQL Remote 或 MobiLink 同步技术，您可以将统一数据库 （或称作主数据库）中的信息发布到任意数量的其他计算机上。您可以精确地控制复制到任一特定计算机上的信息。任何人都可以接收到特定的表，甚至表中的部分行和列。通过对每个人接收到的信息进行自定义，可以确保他们的数据库副本只包含他们所需的信息，而不会没有必要地增大。

另外两本名为 *SQL Remote 用户指南*和 *MobiLink 同步用户指南*的手册中提供了有关 SQL Remote 复制 > 和 MobiLink 同步的详细信息。因此，本节并未提供完整的信息，而仅仅介绍了一些与锁定和并发问题直接相关的概念。

使用 SQL Remote 和 MobiLink 可以从中央 （统一）数据库更新被复制的数据库，也可以通过在远程计算机上处理事务来更新这个相同的中央数据库。由于更新在两个方向上都可以进行，因此这项功能称作**双向复制**。

由于无论在中央计算机上处理事务，还是在远程计算机上处理事务，其结果都可能会影响统一数据库，即具有处理并发事务的效果。

事务可能同时在不同的计算机上发生，甚至可能涉及相同的数据。但在这个例子中，这些计算机可能并没有物理地连接在一起。远程计算机无法与统一数据库进行联系以设置任何形式的锁定，或标识哪些行已被更改。因此，锁定并不能象所有事务都由一台服务器处理时那样防止不一致。

而且，由于任何给定的远程计算机上都没有数据库的完整副本，这使得情况更为复杂。假设有一个直接在主 （统一）数据库上执行的事务。该事务可能会影响两个或多个表中的行。可能不会在远程数据库上执行相同的事务，因为无法确保该计算机上复制了一个受影响的表，还是两个表都复制了。即使有相同的表，这些表中包含的信息也可能不完全一样，这取决于最近一次同步这两个数据库中的信息的时间。

要克服以上约束，应基于操作而不是事务进行复制。**操作**是指对表中一行的更改。这个更改可能是执行 UPDATE、 INSERT 或 DELETE 语句的结果。 UPDATE 或 DELETE 语句产生的操作标识各列的初始值，而 INSERT 或 UPDATE 语句产生的事务记录最终值。

一个事务可能会产生零个、一个或多个操作。一个操作不可能从两个或多个事务产生。如果有两个事务修改了表，那么将产生两个或多个相应的操作。

如果一个操作是从远程计算机上处理的事务产生的，那么必须将它传递到统一数据库，从而可以合并信息。另一方面，如果一个操作是从统一计算机上的事务产生的，那么可能需要将该操作发送到*某些*远程站点，但不需要发送到其他站点。由于每个远程站点都可能包含完整数据库的一部分的复本，因此只有在该操作影响到数据库的那一部分时，SQL Remote 才知道将该操作传递到远程站点。

基于事务日志的复制

SQL Remote 使用的是**基于事务日志**的复制机制。在一台计算机上激活 SQL Remote 时，它将扫描事务日志，找出它必须传送的操作，并准备一条或多条消息。

SQL Remote 可以使用多种方法在计算机间传递这些消息。它可以创建包含这些消息的文件，并将它们存储到指定的目录中。 SQL Remote 也可以使用任何一种最常用的消息传送协议来传递消息。您或许可以使用现有的电子邮件系统。

将来自远程站点的操作合并到统一数据库时，可能会出现冲突。例如，分别位于不同远程站点的两个人可能更改了同一个表中的同一个值。虽然 Adaptive Server Anywhere 的内置锁定功能可以消除由同一台服务器处理的并发事务之间的冲突，但对于两个都具有更改同一个值的权限的用户，则不可能自动消除他们之间的所有冲突。

作为数据库管理员，您可以对数据库进行适当的设计，或编写用于解决冲突的算法，从而避免这个潜在问题。例如，您可以决定只让一个人负责在特定的表中更新特定范围的值。如果这种采取限制的做法不切实际，则可以使用 SQL Remote 的冲突解决功能实施触发器和过程，以适合于所涉及的数据的方式来解决冲突。

SQL Remote 提供了充分利用数据库复制所需要的工具和编程功能。有关详细信息，请参见 *SQL Remote 用户指南*和 *MobiLink 同步用户指南*。

小结

在数据库中，事务和锁定是重要性仅次于表之间关系的两个方面。使用锁定时一定要谨慎，并且在构建事务时也一定要仔细，这一点对任何数据库的完整性和性能都有好处。如果要创建的数据库必须并发执行大量命令，这两者都非常重要。

事务可以将 SQL 语句组合成逻辑工作单元。事务有两种结束方式，一种是回退所有已做更改，另一种是提交这些更改并将其永久应用到数据库中。

出现系统故障时，事务对数据恢复非常重要。它们在交错执行并发事务的语句中也扮演着关键角色。

要提高性能，必须并发执行多个事务。每个事务都由 SQL 语句组成。如果要并发执行两个或多个事务，数据库服务器必须调度各个语句的执行。并发事务可能会导致出现新的，不一致的结果，但如果依次执行相同的事务则不会出现这种结果。

可能会出现多种不一致，但有四种典型类型特别重要，因为 ISO SQL/92 标准中提到了这些类型，而且隔离级别也是依据它们定义的。

- **脏读** 一个事务读取了由另一个事务修改但尚未提交的数据。
- **非可重复的读取** 一个事务读取相同的行两次，但得到的值却不同。
- **幻像行** 一个事务使用特定的条件执行了两次选择行的操作，但在第二次的结果集中发现了新行。
- **更新丢失** 由于允许一个事务基于以前的数据保存更新，导致另一个事务对一行的更改完全丢失。

如果按调度执行事务时数据库达到的状态与依次执行各个事务时相同，那么这个调度就叫做可序列化调度。我们说，可序列化调度是**正确的**。可序列化调度不会造成以上任何一种不一致。

通过锁定，可以控制允许的干扰程度和类型。Adaptive Server Anywhere 提供了四个锁定级别：隔离级别 0、 1、 2 和 3。使用最高级别 （级别 3）时， Adaptive Server Anywhere 可确保调度是可序列化的，意思是说，执行所有事务的结果与依次运行这些事务相同。

可惜的是，一个事务获取的锁定可能会阻碍其他事务的进程。由于这个问题，只要较低隔离级别可能会引起的不一致是可以忍受的，就应使用较低的级别。通过较高的隔离级别来提高数据一致性常常意味着并发性会降低，也就是说，数据库处理并发事务的效率会降低。常常需要在一致性与性能之间找到平衡点，从而确定适合各个操作的隔离级别。

如果不同事务的锁定要求之间发生冲突，可能会导致阻塞或死锁。Adaptive Server Anywhere 具有用于处理这两种情况的机制，并为您提供了用于控制它们的选项。

但是，使用较高隔离级别的事务也不*总*会影响并发性。只有在其他事务需要访问被锁定的行时才会受到阻碍。可以通过仔细设计数据库和事务来提高并发性。例如，可以通过把一个事务分割成两个较短的事务来缩短持有锁定的时间，或者，您可能会发现添加索引可以让事务以较高的隔离级别运行，而放置的锁定反而会减少。

由于便携式计算机越来越普及，因此可能需要对数据库进行复制。复制是 Adaptive Server Anywhere 提供的一项便利功能，但伴随它也产生了一些与并发有关的新问题。我们将在另一本手册中对这些主题进行讨论。

第 5 章

监控和提高性能

关于本章 本章介绍如何监控和提高数据库的性能。

性能分析工具

虽然改进数据库性能不一定会很费力，但是最好总是先拟订一个计划。在进行任何更改之前，要评估数据库的当前性能，并且要考虑好各种选择。通过使用 Adaptive Server Anywhere 的性能功能和性能分析工具重新评估数据库模式，您可以诊断并解决性能问题，并使数据库保持以最佳状态运行。

从根本上讲，数据库性能的优劣在很大程度上取决于它的结构设计。因此，改进性能的最基本的方法之一是使用好的模式设计。数据库模式是数据库的结构，它包括了对许多内容的定义，例如，表、视图、触发器以及它们之间的关系。您要重新评估数据库模式，并且要注意以下方面，在这些方面进行一些小的更改，可能会得到显著的改进。

有多种工具可以帮助您分析和监控 Adaptive Server Anywhere 数据库的当前性能。这些工具包括请求记录、过程分析、图形式计划、性能监控器和计时实用程序。

请求记录

在不清楚是服务器还是客户端有故障时，要对特定应用程序进行性能分析，最好先从请求记录开始。在确定问题的根源是否是向服务器发出特定请求时，它也很有用。

请求记录会记录从应用程序接收的各个请求以及发送到应用程序的响应。要确定应用程序要求服务器执行什么操作，请求级记录最有用。

例如，记录下来的信息有：时间戳、连接 ID 和请求类型。可使用 -zr 服务器选项。使用 -zo 服务器选项，您可以将输出重定向到一个请求日志文件，以便进一步分析。

有关详细信息，请参见 『ASA 数据库管理指南』> 「-zr 服务器选项」和 『ASA 数据库管理指南』> 「-zo 服务器选项」。

sa_get_request_times ([request_log_filename [, connection_id]]) 存储过程会读取请求日志，并用日志中的语句和它们的执行时间来填充一个全局临时表 **satmp_request_time**。对于 INSERT/UPDATE/DELETE 语句，记录的时间是直接了当的。对于查询，记录的时间是从 PREPARE 到 DROP （描述 / 打开 / 读取 / 关闭）所经过的总时间。这意味着您需要知道任何打开的游标。

分析候选对象的 **satmp_request_time**。成本低但经常执行的语句可能会出现性能问题

您可以使用 sa_get_request_profile([request_log_filename [, connection_id]]) 调用 **sa_get_request_times()** 并将结果 **satmp_request_time** 汇总到另一全局临时表 **satmp_request_profile** 中。此过程还可将语句组合在一起，并提供调用次数、执行时间，等等。

过滤请求日志

可以对输出到请求日志的内容进行过滤，只包括来自特定连接或来自特定数据库的请求。在监控具有多个活动连接或多个数据库的服务器时，这有助于减小日志的大小。

❖ 根据连接过滤

- 使用以下语法：

  ```
  call
  sa_server_option('requests_for_connection',connecti
  on-id)
  ```

  其中， *connection-id* 可以从 **sa_conn_info()** 中获取。

❖ 根据数据库过滤

- 使用以下语法：

  ```
  call
  sa_server_option('requests_for_database',database-
  id)
  ```

  在连接到该数据库时，从 **connection_property('DBNumber')** 中可以获取 database-id。在显式重置过滤或关闭服务器之前，过滤将一直有效。

❖ 重置过滤

- 使用下面两个语句中的任何一个，按连接或按数据库重置：

  ```
  call sa_server_option('requests_for_connection',-1)
  ```

```
call sa_server_option('requests_for_database',-1)
```

将主机变量输出到请求日志

主机变量值可以输出到请求日志。

❖ 包括主机变量值

- 要在请求日志中包括主机变量值：
  - 使用值为 **sql+hostvars** 的 -zr 服务器命令行选项
  - 执行以下语句：

```
call
sa_server_option('RequestLogging','sql+host')
```

请求日志分析过程 **sa_get_request_times** 可识别日志中的主机变量并将它们添加到全局临时表 **satmp_request_hostvar** 中。对于 9.0.0 （该版本中不存在此临时表）以前的数据库版本，主机变量值被忽略。

索引顾问

[索引顾问] 会提供索引建议，这些索引有助于改进大负载或复杂查询的性能。它以某个单一的查询或某个负载作为输入，并就要添加到数据库的索引和要删除的未使用的索引提供建议。

[索引顾问] 会在数据库中创建一组虚拟索引。它使用这些索引来预计查询执行成本，以便了解哪些索引会导致执行计划得到改进。 [索引顾问] 会评估多列索引及单列索引，还会调查聚簇索引或非聚簇索引的影响。

有关索引顾问的详细信息，请参见 「索引顾问概述」第 73 页。

过程分析

过程分析向您显示存储过程、函数、事件和触发器的执行时间。您也可以查看过程中每一行的执行时间。利用数据库分析信息，您可以确定微调哪些过程就可以提高数据库的性能。

过程分析可以帮助您通过请求记录来分析您发现成本较高的特定数据库过程（包括存储过程、函数、事件和触发器）。它还可以帮助您发现高成本的隐藏过程，例如，触发器、事件和嵌套的存储过程调用。此外，它还可以帮助在过程主体中定位有潜在问题的区域。

您可以使用存储过程来查看服务器收集的过程分析信息。**sa_procedure_profile_summary** 存储过程提供有关数据库中所有过程的信息。您可以使用该过程在同一结果集中查看存储过程、函数、事件和触发器的分析数据。但是，检查此信息的更好方法是使用 Sybase Central。

分析可以动态地启用 / 禁用，而且它生成的数据是瞬时性的，由服务器存储在内存中。您可以使用 Sybase Central 中的 [分析] 选项卡查看它。分析功能一旦启用，数据库就会收集分析信息，直到您禁用分析功能或服务器关机为止。分析信息是累计的，精确到 1 毫秒。

图形式计划

Interactive SQL 中的图形式计划功能显示查询的执行计划。它对于诊断特定查询的性能问题很有用。例如，计划中的信息可以帮助您确定将索引添加到数据库的什么位置。

图形式计划提供的信息远远多于简单计划或详细计划提供的信息。您可以选择查看带或不带统计信息的图形式计划。通过这两种计划，您都可以快速地查看出计划的哪些部分被预计会占用最多的资源。含统计信息的图形式计划虽然在查看时会占用较多的资源，但也会提供在查询执行过程中服务器监测到的实际查询执行统计信息，并且允许直接比较查询优化程序在构建访问计划时使用的预计值和在执行过程中监测到的实际统计信息。但是，请注意，优化程序通常无法精确地预计查询的成本，因此应该会存在差异。图形式计划是访问计划的缺省格式。

您可以通过单击图示中的节点来获取有关计划中的节点的详细信息。含统计信息的图形式计划不仅可以向您显示随图形式计划提供的所有预计值，而且还可显示执行语句的实际运行时成本。为此，必须实际执行该语句。这意味着在访问占用大量资源的查询的计划时存在延迟。还意味着任何查询部分（如删除或更新）都要实际执行，虽然您可以执行回退来撤消这些更改。

当您遇到性能问题并且预计行计数或运行时间与您的预期值不同时，就可以使用含统计信息的图形式计划。含统计信息的图形式计划会提供预计值和实际统计信息供您比较。如果实际值和预计值之间存在较大的差异，则表明优化程序可能没有足够的信息来准备正确的预计值。

下面是您可以在含统计信息的图形式计划中检查的一些关键统计信息以及一些可能的补救方法：

- 行计数表示的是结果集中的行数。如果预计行计数和实际行计数之间存在很大的差异，则基础谓语的选择性可能不正确。
- 准确的选择性预计值对于查询优化程序的正确运行是非常关键的。例如，如果优化程序错误地将某一谓语的选择性预计得很高（例如 5% 的选择性），但实际该谓语的选择性要低很多（例如 50%），那么性能就可能会受到损害。一般而言，预计值可能会不精确。但是，非常严重的错误确实表明可能存在问题。如果该谓语位于没有直方图的基列之上，则通过执行 CREATE STATISTICS 语句创建直方图就可以更正问题。如果选择性错误仍然是问题，作为最后一招，您可以考虑随查询文本中的谓语一起指定选择性的用户预计值。
- 运行时间表示的是查询的执行时间。如果运行时间对于表扫描或索引扫描不正确，则可以通过执行 REORGANIZE TABLE 语句来提高性能。您可以使用 sa_table_fragmentation 和 sa_index_density 函数来确定表或索引是否已碎片化。
- 当预计值的来源是 [推测] 时，优化程序将没有可用的信息，这可能表示存在问题。如果预计值来源是 [索引] 并且选择性预计值不正确，则问题可能是索引分布不均匀：使用 REORGANIZE INDEX 语句对索引进行碎片整理可能会有好处。
- 如果高速缓存读取次数和高速缓存命中次数完全相同，则说明整个数据库都在高速缓存中 - 这就太好了。当读取次数大于命中次数时，则意味着服务器在尝试从高速缓存读取时失败，它必须从磁盘中读取。在某些情况下（如散列连接），这是您所需要的。而在另外一些情况下（如嵌套循环连接），较低的高速缓存命中率可能说明存在性能问题，而您可以通过增加高速缓存大小来提高性能。

性能监控器

[性能监控器] 可用于跟踪与数据库服务器操作 （包括磁盘和内存访问）相关的详细信息。

利用 Sybase Central 性能监控器，您可以将能通过 Sybase Central 连接到的任何 Adaptive Server Anywhere 数据库服务器的各种统计信息制成图表。 Sybase Central 中的所有统计信息都在 [统计] 文件夹中显示。

该性能监控器的功能包括：

- 实时更新 （以可调整的间隔）
- 带颜色代码且大小可调整的图例
- 可配置的外观属性

当使用 Sybase Central 性能监控器时，请注意它会对服务器使用实际查询来收集其统计信息，因此该监控器本身会影响某些统计信息 （如 [高速缓存读取 / 秒]）。另外一种更为精确的方法是使用 Windows 性能监控器来绘制服务器统计信息的图形。

Windows 性能监控器有两项优点：

- 它提供更多的性能统计信息 （主要是与网络通信相关的统计信息）。
- 与 Sybase Central 监控器不同， Windows 监控器是非侵入性监控器。它使用共享内存方案，而不是对服务器执行查询，因此不会对统计信息本身造成影响。

如果您同时运行 Adaptive Server Anywhere 的多个版本，也可以同时运行性能监控器的多个版本。

计时实用程序

<installation-dir>\samples\asa 目录中提供了一些性能测试实用程序，包括 fetchtst、 instest 和 trantest。在实用程序所在文件夹中的 *Readme.txt* 文件中，可以找到完整的文档。这些工具可比带统计信息的图形式计划提供更准确的计时。这些实用程序可以指出特定的服务器和数据库配置可达到的最佳性能 （例如吞吐量）。

Fetchtst 测量任意查询的读取速率。 Instest 确定将行插入表中所需的时间。 Trantest 测量可由给定服务器配置 （假定一种数据库设计和一组事务）处理的负载。

并发

在数据库服务器处理事务时，可以锁定表中的一行或多行。锁可以通过防止其他事务并发访问来保持数据库中存储的信息的可靠性。锁还可以通过标识正在进行更新的信息来提高查询结果的准确性。

数据库服务器会自动放置这些锁，而无需显式指定。它会保存事务获取的所有锁，直到完成该事务。可以访问该行的事务被称为持有锁。其他事务也许可以对被锁定的行进行有限的访问，也许根本不能访问，这取决于锁的类型。

如果经常有多名用户同时访问行，则可能会损害性能。如果您怀疑有锁定问题，可以考虑使用 **sa_locks** 过程来获取有关数据库中锁的信息。如果发现了锁定问题，可使用 AppInfo 连接属性来找到有关所涉及的连接过程的信息。

最佳性能提示

Adaptive Server Anywhere 会自动提供出色的性能。然而，以下提示将帮助您获得本软件的最佳性能。

始终使用事务日志

您可能会认为如果不使用事务日志， Adaptive Server Anywhere 将运行得更快，因为这样只需要在磁盘上维护较少的信息。但是，情况恰恰相反。事务日志不仅提供大量的保护，而且能够显著地提高性能。

当在不使用事务日志的情况下运行时， Adaptive Server Anywhere 必须在每个事务的结束时执行一次检查点操作。写入这些更改将耗费大量的资源。

但是，如果使用事务日志， Adaptive Server Anywhere 只需要在更改发生时写入详细介绍这些更改的注释。它可以选择在最有效的时间一次写入所有新数据库页。**检查点**将确保信息进入数据库文件并确保信息是一致的和最新的。

提示
请始终使用事务日志。这将有助于保护您的数据*并*大大提高性能。

如果您可以将事务日志存储在包含主数据库文件的设备之外的其他物理设备上，则可以进一步提高性能。额外的驱动器头通常不必进行查找便可到达事务日志的末尾。

增加高速缓存大小

Adaptive Server Anywhere 会将最近使用的页存储在高速缓存中。如果某一请求需要多次访问该页或者另一个连接使用同一页，它们会发现该页已经在内存中，这样就可以避免从磁盘中读取信息。这对于加密数据库尤其重要，因为加密数据库比未加密的数据库需要更大的高速缓存。

如果您的高速缓存太小， Adaptive Server Anywhere 将无法在内存中将页保存足够长的时间，从而无法实现上述优点。

在 UNIX、 Windows NT/2000/XP 和 Windows 95/98/Me 上，数据库服务器会根据需要动态地更改高速缓存大小。但是，高速缓存仍然受到可用的物理内存量以及其他应用程序占用的内存量的限制。

在 Windows CE 和 Novell NetWare 上，高速缓存的大小在数据库服务器启动时进行设置。考虑到并发运行的其他应用程序和进程的要求，务必要给数据库高速缓存分配尽可能多的内存。特别是，使用 Java 对象的数据库会大大受益于较大的高速缓存大小。如果您在数据库中使用 Java，则应使用至少 8 MB 的高速缓存。

提示
由于从内存中检索信息比从磁盘中读取信息快许多倍，因此增加高速缓存大小通常会显著地提高性能。您可能会发现购买更多的内存是值得的，因为这样可以获取更大的高速缓存。

有关详细信息，请参见 「使用高速缓存提高性能」第 191 页。

规范化表结构

通常，表中每个列的信息应该仅依赖于主键的值。如果情况并非如此，那么一个表可能会包含相同信息的多个副本，而该表就需要规范化。

规范化将减少关系数据库中的重复项。例如，假设贵公司的工作人员分散在众多的办事处工作。若要将数据库规范化，应该将有关办事处的信息 （如办事处的地址和主要电话号码）放入单独的表中，而不是为每个员工复制所有这些信息。

但是，您也不能过分地延伸适用于一般情况的规范化概念。如果重复信息量不大，则最好复制该信息，并使用触发器或其他约束维护信息的完整性。

有效使用索引

当执行查询时， Adaptive Server Anywhere 会选择如何访问每个表。索引将大大加快访问的速度。当数据库服务器无法找到合适的索引时，它会转为按顺序扫描表 - 此过程可能需要较长的时间。

例如，假设您需要在一个大型数据库中搜索特定的人员，但只知道他们的名字或姓氏 （而不是都知道）。如果没有索引， Adaptive Server Anywhere 会扫描整个表。但是，如果您创建了两个索引 （一个首先包含姓氏，另一个首先包含名字）， Adaptive Server Anywhere 将首先扫描这两个索引，并且通常可以更快地向您返回信息。

使用索引

虽然索引使 Adaptive Server Anywhere 能够非常有效地定位信息，但在添加索引时应多加小心。每当您插入、删除或更新行时，由于 Adaptive Server Anywhere 还必须更新所有受影响的索引，因此每个索引都会带来额外的工作量。

当索引能够使 Adaptive Server Anywhere 更为有效地访问数据时，应考虑添加索引。当索引可以避免不必要地按顺序访问大型表时，尤其应添加索引。但是，如果您在表中添加行时需要更好的性能并且不用考虑快速查找信息方面的问题，则应尽可能少使用索引。

您可能要使用 [索引顾问] 指导您完成为数据库选择一组有效的索引的操作。有关详细信息，请参见 「索引顾问概述」第 73 页。

聚簇索引

可使用聚簇索引存储表中的行，其顺序与索引中行出现的顺序几乎相同。

有关详细信息，请参见 「索引」第 458 页和 「使用聚簇索引」第 69 页。

将不同的文件放置在不同的设备上

磁盘驱动器的运行速度远远低于现代的处理器或内存。通常，等待磁盘读写页就是数据库服务器速度慢的原因。

当您将不同的物理数据库文件放置在不同的物理设备上时，几乎总能提高数据库的性能。例如，当一个磁盘驱动器忙于与高速缓存交换数据库页时，另一个设备可能会正在写入日志文件。

请注意，若要实现这些优点，设备必须是独立的。分区成多个小逻辑驱动器的单个磁盘不大可能提供这些优点。

Adaptive Server Anywhere 使用四种类型的文件：

1. 数据库 (*.db*)

2. 事务日志 (*.log*)

3. 事务日志镜像 (*.mlg*)

4. 临时文件 (*.tmp*)

数据库文件包含数据库的全部内容。一个文件可以包含一个数据库。或者，您可以添加最多 12 个 dbspace，它们是包含相同数据库的附加文件。您可以选择符合需要的数据库文件位置。

事务日志文件是在出现故障时恢复数据库信息所必需的。为了提供额外的保护，您可以在第三种文件 （称作**事务日志镜像文件**）中维护一个副本。 Adaptive Server Anywhere 会在相同的时间将相同的信息写入这两个文件中的每一个。

提示
通过将事务日志镜像文件 （如果您使用了事务日志镜像文件）放置在单独的物理设备上，您可以对磁盘故障进行更好地保护，而 Adaptive Server Anywhere 的运行速度也会更快，因为它可以有效地写入日志文件和日志镜像文件。要指定事务日志文件和事务日志镜像文件的位置，请使用 dblog 命令行实用程序或 Sybase Central 中的 [更改日志文件设置] 向导。

对于将联合排序和形成联合等操作， Adaptive Server Anywhere 所需的空间可能大于高速缓存中的可用空间。当需要特定空间时， Adaptive Server Anywhere 通常会集中使用该空间。数据库的总体性能越来越多地依赖于包含第四种文件 （即**临时文件**）的设备的速度。

提示
如果临时文件位于快速设备上，并且该设备在物理空间上独立于包含数据库文件的设备，则 Adaptive Server Anywhere 将运行得更快。这是因为许多必须使用临时文件的操作也需要从数据库中检索大量的信息。通过将信息放置在两个单独的磁盘上，这些操作就可以同时进行。

认真选择临时文件的位置。 Adaptive Server Anywhere 会按显示顺序检查以下环境变量，以确定将临时文件放置在哪个目录中。

1. ASTMP

2. TMP

3. TMPDIR

4. TEMP

如果以上环境变量均未定义， Adaptive Server Anywhere 就会将其临时文件放置在当前目录中 - 这并不是获取最佳性能的适当位置。

如果没有定义环境变量， Adaptive Server Anywhere 会将它的临时文件放在 */tmp* 目录 （对于 UNIX）和当前目录 （对于 Windows）中。

如果您的计算机配备有足够数量的快速设备，则可以将这些文件中的每一个放置在一个单独的设备上，从而获取更高的性能。您甚至可以将数据库分为多个位于单独设备上的 dbspace。在这种情况下，请将单独 dbspace 中的表分组，使常见的连接操作从不同的文件中读取信息。

一种类似的策略是将临时文件和数据库文件放置在 RAID 设备或 Windows NT 带区集 (stripe set) 上。虽然这些设备充当逻辑驱动器，但它们会在多个物理设备上分配文件并使用多个驱动器头来访问信息，这样将显著地提高性能。

您可以在启动数据库服务器时指定 -fc 选项，以在数据库服务器遇到文件系统已满时执行回调函数。

有关详细信息，请参见 『ASA 数据库管理指南』> 「-fc 服务器选项」。

有关工作表的详细信息，请参见 「在查询处理中使用工作表」第 201 页。

有关数据恢复的信息，请参见 『ASA 数据库管理指南』> 「备份和数据恢复」。

有关事务日志和 *dbcc* 实用程序的信息，请参见 『ASA 数据库管理指南』> 「事务日志实用程序选项」。

关闭自动提交模式

如果应用程序以**自动提交**模式运行， Adaptive Server Anywhere 就会将每个语句当作单独的事务。实际上，这等效于将 COMMIT 语句附加到每个命令的末尾。

不要以自动提交模式运行，而应该将命令分组，使每个命令组执行一个逻辑任务。如果您禁用了自动提交，则必须在每个逻辑命令组后执行一个显式提交命令。另外，请注意如果逻辑事务太大，则会发生阻塞和死锁。

如果您没有使用事务日志文件，则使用自动提交模式将导致极高的成本。每个语句会强制执行一个检查点操作 - 该操作可能需要将多页信息写入磁盘。

每个应用程序接口各自有不同的自动提交行为设置方式。对于 Open Client、 ODBC 和 JDBC 接口，自动提交是缺省行为。

有关自动提交的详细信息，请参见 『ASA 编程指南』> 「设置自动提交或手动提交模式」。

使用批量操作方法

如果您需要将海量的信息装载到数据库中，则可以借助于为这些任务提供的特殊工具。

如果您将要装载大文件，一种较为有效的方法是在装载数据后在表上创建索引。

有关提高批量操作性能的信息，请参见 「批量操作的性能问题」第 595 页。

在校验表时使用 WITH EXPRESS CHECK 选项

如果您发现校验具有较小高速缓存的大型数据库需要很长的时间，则可以使用以下两个选项之一来减少所用的时间。通过使用 VALIDATE TABLE 语句的 WITH EXPRESS CHECK 选项或者校验实用程序的 `-fx` 选项，可以大大增加表的校验速度。

有关在校验数据库时提高性能的信息，请参见 『ASA 数据库管理指南』> 「在校验数据库时改善性能」。

尝试使用 Adaptive Server Anywhere 的压缩功能

通过对一个连接或所有连接启用压缩并调整包压缩的最小大小限值，在某些情况下可以大大提高 Adaptive Server Anywhere 的性能。

要确定在特定情况下启用压缩是否能提供帮助，建议您在生产性环境中使用通信压缩之前，应在您的特定网络上使用特定的应用程序进行性能分析。

启用通信压缩可增强存储在数据包中的信息的质量，从而减少传输一定数据的所需数据包的数量。通过减少包数量，可以更快地传输数据。

通过指定压缩阈值，可以选择您希望数据包压缩的最小大小。最佳的压缩阈值可能会受到多种因素的影响，其中包括所用网络的类型和速度。

有关使用压缩提高性能的信息，请参见 『ASA 数据库管理指南』> 「调整通信压缩设置以改善性能」。

有关压缩设置的详细信息，请参见 『ASA 数据库管理指南』> 「Compress 连接参数 [COMP]」和 『ASA 数据库管理指南』> 「CompressionThreshold 连接参数 [COMPTH]」。

减少客户端和服务器之间的请求数

如果您处于下列情况：

- 网络的等待时间较长
- 应用程序发送过多的打开和关闭游标请求

您可以使用 LazyClose 和 PrefetchOnOpen 网络连接参数来减少客户端和服务器之间的请求数，从而提高性能。

有关这些参数的信息，请参见 『ASA 数据库管理指南』> 「LazyClose 连接参数 [LCLOSE]」和 『ASA 数据库管理指南』> 「PreFetchOnOpen 连接参数」。

减小表宽

具有大量列的表称为宽表。当表中的列数导致各个行的大小超过数据库页大小时，则会在两个或更多个数据库页之间拆分每个行。行占用的页数越多，读取每一行所需的时间越长。如果您发现性能很低，并且您知道您的表具有很多列，请考虑使表规范化以减少列数。如果这无法实现，较大的数据库页大小可能会有所帮助，尤其是在多数表都很宽的情况下。

减小主键宽度

宽主键由两个或更多个列组成。主键中包含的列越多，对服务器的需求越多。减小主键中的列数可以改进性能。

声明约束

当不同表中的列值之间存在隐含关系时，表之间存在未声明的主键 - 外键关系。不声明该关系可以节省进行索引维护的时间，这很正确，但是，声明该关系可以在进行连接时提高查询的性能，因为该成本模型可以更好得进行预计。

使用适当的数据类型

数据类型存储有关特定数据集的信息，包括值的范围、可以对那些值执行的操作以及值在内存中的存储方式。为数据使用适当的数据类型可以提高性能。例如，不要为只含数值数据的值指派字符或字符串数据类型。并且，只要有可能，就要选择成本经济的数值和字符串数据类型。

使用 AUTOINCREMENT 创建主键

主键值必须是唯一的。虽然为主键创建唯一值的方法有多种，但最有效的方法是将缺省列值设置为 AUTOINCREMENT ON。对于要保持唯一值的任何列，您都可以使用此缺省值。使用 AUTOINCREMENT 功能生成主键值比其它方法速度快，原因是该值由服务器生成。

替换成本高的触发器

评估触发器的用途以查看一些触发器是否可以由服务器中提供的功能替换。例如，使用最新的更新时间和用户信息更新列的触发器可由服务器中相应的特殊值替换。此外，使用现有触发器上的缺省设置也可以提高性能。

最大程度地减少级联参照动作

就性能而言，级联参照动作的代价会很大，因为它们会导致更新每个事务的多个表。例如，如果从 *employee* 到 *department* 都使用 ON UPDATE CASCADE 定义了外键，那么，在更新部门 ID 时， employee 表就会自动更新。虽然级联参照动作很方便，但有时在应用程序逻辑中实现它们可能会更有效。

关注列的顺序

对行中的列的访问是按其创建顺序连续进行的。例如，为了访问行末尾的列， Adaptive Server Anywhere 就必须跳过行中先出现的列。主键列始终存储在行的开头。因此，创建表时让较小的和 / 或经常被访问的列放在表中很少被访问的列的前面，是很重要的。

升级以利用新功能

Adaptive Server Anywhere 经常接受性能功能和可用性方面的评估并在这些方面得到增强。使用每个后续版本，您可以利用新功能和行为更改，它们会帮助您优化性能。

虽然您可以在较新版本的 Adaptive Server Anywhere 上直接运行用较旧版本的 Adaptive Server Anywhere 创建的数据库，但许多新功能却只有在升级数据库后才可用。运行升级实用程序可添加和修改系统表、系统过程和数据库选项以将数据库从较旧的 Adaptive Server Anywhere 版本升级到较新的版本。

重建数据库

重建数据库是卸载并重装整个数据库的过程。它有时称为升级数据库文件格式。

重建会删除所有信息 （包括数据和模式），并以统一方式将其全部放回到原位置，从而填充空间并提高性能，这与整理磁盘驱动器碎片非常相似。重建还可以让您有机会更改某些特定设置。相比之下，使用升级实用程序比重建要快，而且它不影响文件的存储。

升级数据库时，重建还会为您提供对最新版本的软件中的所有新功能和性能增强的访问权。

检查数据库和服务器配置

检查数据库 / 服务器配置，确保硬件和文件空间适合您的使用。

使用适当的页大小

选择的页大小会影响数据库的性能。 Adaptive Server Anywhere 支持以下字节的页大小：1024、 2048、 4096、 8192、 16384 或 32768，其中 2048 是缺省值。选择任何页大小都有优点和缺点。

虽然较小的页包含的信息较少并且可能会强制降低空间使用效率 （尤其当您插入的行略大于半页大小时更是如此），但是较小的页大小也使得 Adaptive Server Anywhere 可以使用较少的资源运行，因为它在相同大小的高速缓存中可以存储更多的页。如果您的数据库必须在内存有限的小型计算机上运行，较小的页将特别有用。当您的数据库主要用来从任意位置检索小段信息时，较小的页也非常有用。

相比而言，较大的页大小有助于 Adaptive Server Anywhere 更有效地读取数据库。较大的页大小往往还对大型数据库和执行顺序表扫描的查询有好处。通常，磁盘的物理设计会使磁盘检索较少的大块比检索较多的小块更有效率。较大的页大小还有其他优点，其中包括提高索引的涵盖范围，进而减少索引级别数，而且，表可以包括更多的列。

请记住，较大的页大小对内存的需求也更多。而且，由于页上最多能存储 255 行，所以行数较少的表不会填满每一页，这样，空间使用效率就比较低。此外，除非您能确保总是有较大的数据库服务器高速缓存，否

则，对于大多数据应用程序，都不建议使用极其大的页大小 （16 kb 或 32 kb）。在使用 16 kb 或 32 kb 页大小前，先要调查内存和磁盘空间增加对性能特性的影响。

服务器的内存使用量与装载的数据库数和数据库的页大小成比例。强烈建议您在选择页大小时进行性能测试 （和一般测试）。然后，选择可提供满意结果的最小的页大小 (>= 4K)。如果要在同一服务器上启动大量数据库，选择正确的 （且合理的）页大小相当重要。

您不能更改现有数据库的页大小，而必须新建数据库并使用 *dbinit* 的 `-p` 选项来指定页大小。例如，以下命令将创建一个页大小为 4K 的数据库。

```
dbinit -p 4096 new.db
```

有关较大页大小的详细信息，请参见 『ASA 数据库管理指南』> 「设置最大页面大小」。

分散读取

如果您使用的是 Windows NT Service Pack 2 或更高版本的系统，或 Windows 2000/XP 系统，如果页大小至少为 4K，则数据库服务器可以将磁盘上数据库页的连续部分直接读取到高速缓存中适当的位置，不使用 64K 的缓冲。此特性可显著提高系统性能。

检查文件、表和索引碎片

更改数据库时，自然会出现碎片。如果文件、表或索引出现了过多的碎片，则可能会破坏性能。当数据库的大小增加时，这一问题就变得越为重要。 Adaptive Server Anywhere 包含一些存储过程，它们会生成有关文件、表和索引碎片的信息。

如果性能显著降低，请考虑

- 重建数据库以减少表和 / 或索引碎片
- 将数据库单独放置在磁盘分区上以减少文件碎片
- 定期运行可用的 Windows 实用程序之一以减少文件碎片
- 对表进行重组以减少数据库碎片
- 如果对多个表执行了大量的删除 / 更新 / 插入活动，请重建数据库

- 使用 REORGANIZE TABLE 语句整理表中行的碎片，或者压缩可能由于执行 DELETE 而变得分散的索引。对表进行重组可以减少用于存储表及其索引的总页数，而且还可以减少索引树中的级别数。

有关检测和修复文件、表和索引碎片的详细信息，请参见 「碎片」第 209 页。

消除操作系统文件碎片

要消除操作系统文件碎片问题，请定期运行可用的磁盘碎片整理实用程序之一。文件碎片可能会对性能产生有害影响。

数据库服务器会在您在 Windows NT/2000/XP 上启动数据库时确定数据库文件中文件碎片的数量，并且会在碎片数大于一时在服务器消息窗口中显示以下信息：

```
Database file "mydatabase.db" consists of nnn fragments
```

您也可以使用 DBFileFragments 数据库属性来获取数据库文件的碎片数。

最大程度地减少索引碎片

索引用于加快对特定列的搜索速度，但如果对索引表执行了太多的 DELETE 操作，索引就会出现过多的碎片。如果碎片被频繁访问，并且高速缓存容量不足，不能保存所有索引，则会导致性能降低。

sa_index_density 存储过程提供有关数据库索引中的碎片程度的信息。要运行此过程，必须具有 DBA 权限。以下语句调用 **sa_index_density** 存储过程：

```
CALL sa_index_density (['table_name'[,'owner_name']])
```

如果索引的碎片太多，则可以运行 REORGANIZE TABLE。您也可以删除并重新创建该索引。但是，如果该索引是主键，您还将需要删除并重新创建外键索引。

此外，您可以通过在表上创建聚簇索引来提高性能。聚簇索引可导致存储表中的行，顺序与这些行在索引中的出现顺序大致相同。

最大程度地减少表碎片

当没有连续存储行时，或者当行在多个页之间被拆分时，会出现表碎片。性能之所以会降低，是因为这些行需要更多地访问页面。

Adaptive Server Anywhere 在每一页上保留了额外的空间，以允许行略微地增大。当对行的更新导致行超过当前页保留的可用空间时，该行会被拆分，并且初始行位置将包含一个指针，该指针指向存储连续行的另一

页。例如，如果用 UPDATE 语句填充空行或将新列插入表，可能会导致严重的行拆分。当更多的行存储在单独的页上时，需要更多的时间来访问附加页。

您可以使用 **sa_table_fragmentation** 存储过程来获取有关数据库表的碎片程度的信息。以下语句调用 **sa_table_fragmentation** 存储过程：

```
CALL sa_table_fragmentation (['table_name' [,'owner_name']])
```

最大程度地减少表碎片的方法有三种：

- 您可以指定表页中应该为未来的更新保留的空间百分比。这一 PCTFREE 规格可以用 CREATE TABLE、 ALTER TABLE、 DECLARE LOCAL TEMPORARY TABLE 或 LOAD TABLE 来设置。
- 您可以重组特定表。要对特定表或表中某些部分进行碎片整理，您可以运行 REORGANIZE TABLE。重组表的操作不会中断数据库访问。
- 您可以重建整个数据库。重建更为全面，原因是它要对所有表（包括系统表）进行碎片整理。此外，它会重组表行以便它们按聚簇索引和主键指定的顺序出现。

获取适当的硬件

在 PC 上运行时，确保服务器满足 Adaptive Server Anywhere 的最低 CPU、内存和磁盘要求：

- Adaptive Server Anywhere 最少只需 4 Mb 的内存就可以运行。如果您在数据库中使用 Java，则 Adaptive Server Anywhere 需要 8 Mb 的内存。如果您使用管理工具， Adaptive Server Anywhere 要求至少 32 Mb 的 RAM。除操作系统要求的内存外，计算机还必须添加以上要求的内存。
- 有足够的空间用于存放数据库和日志文件。
- 请记住，这些是最低要求。如果您只满足最低硬件要求，并发现性能不佳，请考虑升级某些或全部硬件。一般而言，请评估硬件配置以查看它是否适用于放置在服务器上的负载种类。

您可以在启动数据库服务器时指定 -fc 选项，以在数据库服务器遇到文件系统已满时执行回调函数。

有关详细信息，请参见 『ASA 数据库管理指南』> 「-fc 服务器选项」。

选择优化程序的优先级

OPTIMIZATION_GOAL 选项控制 Adaptive Server Anywhere 是为响应时间 （第一行）还是为总资源消耗 （所有行）而优化 SQL 语句。简而言之，您可以选择优化查询处理的目标是为了快速返回第一行还是尽可能降低返回完整结果集的成本。

如果将选项设置为 first-row， Adaptive Server Anywhere 会选择试图减少查询结果的首行提取时间的访问计划，而且很可能以总检索时间为代价。特别是， Adaptive Server Anywhere 优化程序通常会尽量避免需要实现结果以便减少返回首行的时间的访问计划。例如，使用此设置，优化程序会倾向于利用索引来满足查询的 ORDER BY 子句的访问计划，而不是需要显式排序操作的计划。

您可以使用查询的 FROM 子句中的 FASTFIRSTROW 表提示将特定查询的优化目标设置为第一行，而不必更改 OPTIMIZATION_GOAL 设置。

如果此选项设置为所有行 （缺省值），则 Adaptive Server Anywhere 将优化查询以选择预计总检索时间最少的访问计划。将 OPTIMIZATION_GOAL 设置为所有行可能适用于要处理整个结果集的应用程序，例如 PowerBuilder DataWindow 应用程序。

指定正确的游标类型

指定正确的游标类型可以提高性能。例如，如果游标是只读的，则将它声明为 [只读] 可以加速优化和执行，因为要生成的资料较少 （没有检查约束，等等）。如果游标是可更新的，则可以跳过某些重写。此外，如果查询是可更新的，则根据优化程序选择的执行计划，执行引擎必须使用键集驱动方法。请记住，键集游标成本更高。

考虑收集小表上的统计信息

Adaptive Server Anywhere 利用统计信息确定执行各语句的最有效策略。Adaptive Server Anywhere 自动收集和更新这些统计信息，并将它们永久地存储在数据库中。处理一个语句时所收集的统计信息对于搜索后续语句的有效执行方式很有用。

缺省情况下， Adaptive Server Anywhere 为含有五行或更多行的所有表创建统计信息。如果需要为含有少于五行的表创建统计信息，则可以使用 CREATE STATISTICS 语句做到这一点。此语句为所有表创建统计信息，与表的行数无关。一旦创建了统计信息， Adaptive Server Anywhere 就会自动对其进行维护。

谨慎提供显式选择性预计值

有时，统计信息可能会变得不准确。如果在添加、更新或删除了大量数据后只执行了很少的查询，则最有可能发生这种情况。统计信息不准确或不可用会降低性能。如果 Adaptive Server Anywhere 更新统计信息花费的时间太长，请尝试执行 CREATE STATISTICS 或 DROP STATISTICS 以刷新它们。此外，在较新的服务器版本 （9.0 版或更高版本）中， Adaptive Server Anywhere 不仅尝试在 LOAD TABLE 上和查询执行期间更新统计信息，还会尝试在更新 DML 语句上更新统计信息。

然而，在极少数情况下，这些措施可能无效。如果您知道某条件的成功率与优化程序估计的不一样，您可以在搜索条件中显式地提供用户的估计。

虽然用户定义的预计有时可以提高性能，但是要避免在不断使用的语句中提供显式的用户定义预计。如果数据改变，显式估计可能会变得不准确并强制优化程序选择低效率的计划。

对于由于软件选择的访问计划不佳而引发的性能问题，如果您使用了不准确的选择性预计值作为变通解决办法，则可以将 USER_ESTIMATES 设置为 OFF 以忽略这些值。

有关详细信息，请参见 『ASA SQL 参考』> 「显式选择性估计」。

减少成本高的用户定义函数

在必须多次执行成本很高的用户定义函数的查询中，减少这些函数可以提高性能。

使用高速缓存提高性能

数据库高速缓存是内存中的特定区域，它被数据库服务器用来存储数据库页以供重复的快速访问。高速缓存中可访问的页数越多，数据库服务器从磁盘中读取数据所需的次数就越少。由于从磁盘中读取数据是相当慢的操作，因此可用的高速缓存量通常是决定性能的关键因素之一。

当数据库启动时，您可以在数据库服务器命令行上控制数据库高速缓存的大小。

动态调整高速缓存大小

Adaptive Server Anywhere 提供了自动调整数据库高速缓存大小的功能。在不同的操作系统上，这些功能将有所不同。在 Windows NT/2000/XP、 Windows 95/98/Me 和 UNIX 操作系统上，高速缓存会增大和缩小。详细说明将在下面几节中提供。

完全的**动态调整高速缓存大小**有助于确保数据库服务器的性能不会因为分配的内存不足而受到影响。当数据库服务器可以有效地使用更多内存（只要内存可用）时，高速缓存将增大，当其它应用程序需要高速缓存内存时，高速缓存将变小，这样，数据库服务器就不会对系统上的其它应用程序造成不适当的影响。当然，系统上可用的物理内存会限制动态调整高速缓存大小的有效性。

通常情况下，动态高速缓存大小调整以大约每分钟一次的速率评估高速缓存要求。但是，启动新数据库后或者当文件显著增大时，会对统计信息取样并且高速缓存的大小可能会每五到三十秒调整一次。在最初的三十秒期间，取样速率会下降到每分钟一次。文件显著增大是指，自数据库启动以后或者自上次发生触发取样速率提高的增长后出现了 1/8 的增长。此变化可在数据库动态启动时和插入大量数据时更快速地适应高速缓存大小，从而可以进一步提高性能。

动态调整高速缓存大小使您在许多情况下都无需显式配置数据库高速缓存，从而使 Adaptive Server Anywhere 使用起来更加方便。

Windows CE 或 Novell NetWare 没有动态调整高速缓存大小的功能。当使用 Address Windowing Extensions (AWE) 高速缓存时，动态调整高速缓存大小的功能将被禁用。

有关 AWE 高速缓存的详细信息，请参见 『ASA 数据库管理指南』> 「-cw 服务器选项」。

限制高速缓存使用的内存

高速缓存的初始、最小和最大大小都可以通过数据库服务器命令行进行控制。

- **初始高速缓存大小** 您可以通过指定数据库服务器 -c 命令行选项更改初始高速缓存的大小。缺省值如下：
  - **Windows CE** 相应的公式如下：

    max(600K, min(dbsize , physical-memory))

    其中 *dbsize* 是所启动的数据库文件的大小总和， *physical-memory* 是计算机上物理内存的 25%。
  - **Windows NT/2000/XP、 Windows 95/98/Me、 NetWare** 相应的公式如下：

    max(2M, min(dbsize , physical-memory))

    其中 *dbsize* 是所启动的数据库文件的大小总和， *physical-memory* 是计算机上物理内存的 25%。

    如果在 Windows 2000、Windows XP 或 Windows Server 2003 上使用了 AWE 高速缓存，则公式如下：

    min(100% of available memory-128MB, dbsize)

    如果该值小于 2 Mb，则没有使用 AWE 高速缓存。

    有关 AWE 高速缓存的信息，请参见 『ASA 数据库管理指南』> 「-cw 服务器选项」。
  - **UNIX** 至少 8 Mb。

    有关 UNIX 初始高速缓存大小的信息，请参见 「动态调整高速缓存大小 (UNIX)」第 194 页。
- **最大高速缓存大小** 您可以通过指定数据库服务器 -ch 命令行选项来控制最大高速缓存大小。其缺省值基于由计算机上的物理内存决定的启发式算法。在非 UNIX 计算机上，这通常大约是总物理内存的 90%，但是不会超过 256 Mb。在 UNIX 计算机上，这是以下值中的较大者：

```
(25% total physical-memory,
    25%(available-physical-memory + available-
swap-space),
    sum-of-file-sizes-on-the-command-line)
```

但是不会大于 256 Mb，也不会小于 24 Mb。

- **最小高速缓存大小**　您可以通过指定数据库服务器 `-cl` 命令行选项来控制最小高速缓存大小。缺省情况下，最小高速缓存大小与初始高速缓存大小相同。

您也可以使用 `-ca` 命令行选项来禁用动态调整高速缓存的功能。

有关命令行选项的详细信息，请参见 『ASA 数据库管理指南』> 「数据库服务器」。

动态调整高速缓存大小 （Windows NT/2000/XP、 Windows 95/98/Me）

在 Windows NT/2000/XP 和 Windows 95/98/Me 上，数据库服务器将每分钟计算一次高速缓存和运行统计信息并计算出最佳的高速缓存大小。服务器将计算出目标高速缓存大小，该大小使用当前未使用的所有物理内存 （保留下来供系统使用的 5 Mb 内存除外）。目标高速缓存大小决不会小于指定或隐式的最小高速缓存大小。目标高速缓存大小决不会超过指定或隐式的最大高速缓存大小 （即所有打开的数据库和临时文件的大小与主堆大小的总和）。

为了避免高速缓存大小波动，数据库服务器会以递增的方式增加高速缓存大小。它不是立即将高速缓存大小调整到目标值，而是按当前高速缓存大小和目标高速缓存大小之间差值的 75% 逐次调整高速缓存大小。

Windows 2000、 Windows XP 和 Windows Server 2003 可以使用 Address Windowing Extensions (AWE) 支持较大的高速缓存大小，方法是在启动数据库服务器时指定 `-cw` 命令行选项。 AWE 高速缓存不支持动态高速缓存调整。

有关详细信息，请参见 『ASA 数据库管理指南』> 「-cw 服务器选项」。

动态调整高速缓存大小 (UNIX)

在 UNIX 上，数据库服务器使用交换空间和内存来管理高速缓存大小。交换空间是大多数 （但不是全部） UNIX 操作系统上的系统范围资源。在本节，内存和交换空间的总和称作**系统资源**。有关详细信息，请参见相应的操作系统文档。

在启动时，数据库会从系统资源中分配指定的最大高速缓存大小。它会将最大高速缓存大小的一部分装载到内存 （初始高速缓存大小）中，并保留其余部分作为交换空间。

在数据库服务器关闭之前，数据库服务器所用的系统资源总量是恒定的，但在内存中装入的比例会出现变化。数据库服务器会每分钟计算一次高速缓存和运行统计信息。如果数据库服务器处于繁忙状态并且需要很多内存，它可能会将高速缓存页从交换空间移动到内存中。如果系统中的其它进程需要内存，则数据库服务器可能会将高速缓存页从内存中移出，移至交换空间中。

初始高速缓存大小

缺省情况下，将使用基于可用系统资源的启发式算法来指定初始高速缓存大小。初始高速缓存大小始终小于数据库总大小的 1.1 倍。

如果初始高速缓存大小大于可用系统资源的 3/4，数据库服务器将退出并报告内存不足错误

您可以使用 -c 选项来更改初始高速缓存的大小。

最大高速缓存大小

最大高速缓存必须小于计算机上的可用系统资源。缺省情况下，将使用基于计算机上可用系统资源和总物理内存的启发式算法来指定最大高速缓存大小。高速缓存大小决不会超过指定或隐式的最大高速缓存大小（即所有打开的数据库和临时文件的大小与主堆大小的总和）。

如果您指定的最大高速缓存大小大于可用系统资源，服务器将退出并报告内存不足错误。如果您指定的最大高速缓存大小大于可用内存，服务器将发出性能降低警告，但不会退出。

数据库服务器将从系统资源中分配所有*最大*高速缓存大小，并且在服务器退出之前不会放弃该大小。您应该确保所选择的最大高速缓存大小提供了较优的 Adaptive Server Anywhere 性能，并且为其他应用程序保留了足够的空间。缺省最大高速缓存大小的公式就是要尽量达到这一平衡的启发式算法。只有当缺省值在您的系统上显得不合适时，才需要调整缺省值。

如果您指定的最大高速缓存大小小于 8 Mb，则将无法运行 Java 应用程序。较小的最大高速缓存大小将影响性能。

可以使用 -ch 服务器选项设置最大高速缓存大小，并限制自动高速缓存的增加。有关详细信息，请参见 『ASA 数据库管理指南』> 「-ch 服务器选项」。

最小高速缓存大小

如果指定了 -c 选项，则最小高速缓存大小与初始高速缓存大小相同。如果没有指定 -c 选项，则 UNIX 上的最小高速缓存大小是 8 Mb。

您可以使用 -cl 服务器选项来调整最小高速缓存大小。

有关详细信息，请参见 『ASA 数据库管理指南』> 「-cl 服务器选项」。

监控高速缓存大小

以下统计信息已经添加到 Windows 性能监控器和数据库的 property 函数。

- **CurrentCacheSize**　当前高速缓存大小 (KB)
- **MinCacheSize**　允许的最小高速缓存大小 (KB)
- **MaxCacheSize**　允许的最大高速缓存大小 (KB)
- **PeakCacheSize**　高速缓存大小峰值 (KB)

注意：
Windows 性能监控器在 Windows NT、 Windows 2000 和 Windows XP 中提供。

有关这些属性的详细信息，请参见 『ASA 数据库管理指南』> 「服务器级属性」。

有关监控性能的信息，请参见 「监控数据库性能」第 203 页。

使用高速缓存预热

高速缓存预热旨在帮助减少对数据库执行的初始查询的执行时间。通过在数据库服务器的高速缓存中预装载上次启动数据库时引用的数据库页，可以实现这一点。如果在每次启动数据库时对其执行相同的查询或相似的查询，则预热高速缓存可以提高性能。

您要控制数据库服务器命令行上的高速缓存预热设置。当数据库启动并且打开了高速缓存预热时，可能会发生两个活动：数据库页的收集和高速缓存重装 （预热）。

对所引用的数据库页的收集由 -cc 数据库服务器选项控制，缺省情况下处于打开状态。当数据库页收集打开时，数据库服务器会保持跟踪数据库启动中请求的每个数据库页，直到发生以下情况之一：收集的页数达到最大值 （该值基于高速缓存大小和数据库大小），收集速率降至最低阈值以下，或者数据库关闭。请注意，服务器控制页的最大数量和收集阈值。完成收集后，会将引用的页记录在数据库中，这样，便可以在下次启动数据库时使用它们预热高速缓存。

高速缓存预热 （重装）在缺省情况下处于打开状态，并且由 -cr 数据库服务器选项控制。为了预热高速缓存，服务器会检查数据库是否包含以前记录的页集合。如果包括，服务器会将相应的页装载到高速缓存中。在高速缓存装载页时，数据库仍旧可以处理请求，但是，如果在数据库中检测到大量 I/O 活动，预热可能会停止。高速缓存预热在这种情况下之所以停止，是为了避免对未包含在重装到高速缓存中的页集中的页进行访问的查询性能降低。如果您希望有关高速缓存预热的消息显示在数据库服务器窗口中，可以指定 -cv 选项。

有关用于高速缓存预热的数据库服务器选项的详细信息，请参见『ASA 数据库管理指南』> 「-cc 服务器选项」、『ASA 数据库管理指南』> 「-cr 服务器选项」和 『ASA 数据库管理指南』> 「-cv 服务器选项」。

使用索引改进性能

正确选择索引可以大大提高性能。在「使用索引」第 68 页中介绍了创建和管理索引。

使用键提高查询性能

主键和外键虽然主要用于校验，但也可以提高数据库性能。

示例

以下示例说明主键如何加快查询的执行速度。

```
SELECT *
 FROM employee
 WHERE emp_id = 390
```

服务器要执行该查询，最简单的方法是查看 **employee** 表中的所有 75 个行，并检查每行中的雇员 ID 号是否为 390。由于只有 75 名雇员，因此这种方法不会花费很长的时间，但对于包含数千个条目的表，顺序搜索可能要花费很长的时间。

每个主键或外键所体现的参照完整性约束是由 Adaptive Server Anywhere 借助于索引（用每个主键或外键声明隐式创建）来强制的。 emp_id 列是 employee 表的主键。通过对应的主键索引，可以快速地检索到雇员编号 390。无论 employee 表中有 100 行还是有 1,000,000 行，这种快速搜索方法都将使用大致相同的时间。

使用主键提高查询性能

主键提高以下语句的性能：

```
SELECT *
 FROM employee
 WHERE emp_id = 390
```

有关 [计划] 选项卡的信息

[结果] 窗格中的 [计划] 选项卡包含以下信息：

```
employee <employee>
```

只要 [计划] 选项卡的 [计划] 说明中带括号的名称与表的名称相同，则表示已使用表的主键来提高性能。

使用外键提高查询性能

以下查询列出客户 ID 为 113 的客户的订单：

```
SELECT *
 FROM sales_order
 WHERE cust_id = 113
```

有关 [计划] 选项卡的信息

[结果] 窗格中的 [计划] 选项卡包含以下信息：

```
sales_order <ky_so_customer>
```

其中 ky_so_customer 指 sales_order 表具有的用于 customer 表的外键。

单独的主键和外键索引

系统将自动为主键和外键创建单独的索引。这样， Adaptive Server Anywhere 就可以更为有效地执行许多操作。此功能在版本 7.0 中引入。

查询结果排序

许多查询都包含 ORDER BY 子句，该子句确保各行按可预知的顺序排列。索引可以快速地将信息排序。例如，下面的查询可以使用 customer 表的 lname 列上的索引按姓氏字母顺序访问 customer 表中的行：

```
SELECT *
 FROM customer
 ORDER BY customer.lname
```

包含 WHERE 和 ORDER BY 子句的查询

当一个查询既包含 WHERE 子句又包含 ORDER BY 子句时，潜在的问题就会出现。

```
SELECT *
 FROM customer
 WHERE id > 300
 ORDER BY company_name
```

Adaptive Server Anywhere 必须在两种策略之间做出选择：

1. 按照公司名称的顺序检查整个客户表，查看每行的客户 ID 是否大于 300。

2. 使用 id 列上的键只读取 id 大于 300 的公司。然后，需要按照公司名称对结果进行排序。

如果只有很少的 id 值大于 300，则第二种策略较好，因为只需要扫描很少的行并且可以迅速地将行排序。如果大多数 id 值都大于 300，则第一种策略要好得多，因为它不需要排序。

有关排序的详细信息，请参见 「ORDER BY 子句：对查询结果进行排序」第 266 页或 「GROUP BY 子句：将查询结果划分为组」第 258 页。

在查询处理中使用工作表

工作表是已实现的临时结果集，这些临时结果集在执行查询的过程中创建。当 Adaptive Server Anywhere 确定使用工作表所需的成本少于其他策略时，就会使用工作表。通常，读取前几行所需的时间在使用工作表时要多一些，但是在某些情况下，如果可以使用工作表，检索所有行所需的成本则可能会大大降低。由于存在这种差异， Adaptive Server Anywhere 会根据 OPTIMIZATION_GOAL 设置选择不同的策略。缺省值是前几行 (first-row)。如果该值设置为前几行， Adaptive Server Anywhere 将尝试避免使用工作表。如果该值设置为所有行 (all-rows)，Adaptive Server Anywhere 将在工作表会减少查询的总执行成本时使用工作表。

在以下情况中将使用工作表：

何时使用工作表

- 当查询包含 ORDER BY、 GROUP BY 或 DISTINCT 子句，并且 Adaptive Server Anywhere 不使用索引来进行行排序时。如果存在适当的索引而且 OPTIMIZATION_GOAL 设置是前几行，Adaptive Server Anywhere 将避免使用工作表。但是，当 OPTIMIZATION_GOAL 设置为所有行时，使用索引读取查询的所有行可能会比建立工作表并将行排序更为耗费成本。如果优化目标设置为所有行， Adaptive Server Anywhere 会选择成本较低的策略。对于 GROUP BY 和 DISTINCT，基于散列的算法将使用工作表，但通常在读取查询中的所有行时更为有效。
- 当选择散列连接算法时，将使用多个工作表来存储中间结果 （如果内存不够存放）并使用一个工作表来存储连接的结果。
- 当用值敏感打开游标时，将创建一个工作表来保存基表的行标识符和主键。当向前从查询中读取行时，将填充该工作表。但是，如果从游标中读取最后一行，则将填充整个表。
- 当用不敏感的语义打开游标时，将在查询打开时用查询结果填充工作表。
- 当多行 UPDATE 操作正在执行，并且所更新的列出现在 Update 命令的 WHERE 子句或用于 Update 操作的索引中时。
- 当多行 UPDATE 或 DELETE 操作在 WHERE 子句中有一个引用所修改的表的子查询时。

- 当从 SELECT 语句中执行 INSERT，并且该 SELECT 语句引用插入表时。
- 当执行多行 INSERT、 UPDATE 或 DELETE 操作，并且定义相关触发器的表可能在操作过程中触发时。

在这些情况下，操作所影响的记录会进入工作表。在某些情况 （如键集驱动的游标）下，将在工作表上建立临时索引。在查询结果出现之前，将所需记录提取到工作表中的操作可能会需要大量的时间。通过创建可用来在上述第一种情况下执行排序的索引，可以减少检索前几行所需的时间。但是，如果使用工作表，则可以减少读取所有行所需的时间，因为它支持基于散列和合并排序的查询算法。这些算法采用顺序 I/O，这比用于索引扫描的随机 I/O 要快一些。

数据库服务器中的查询优化程序将分析每个查询，以确定工作表是否将提供最佳性能。 Adaptive Server Anywhere 的新版本中对优化程序的改进可能会改善查询的访问计划。用户无需执行任何操作即可利用这些优化。

注意

上述的 INSERT、 UPDATE 和 DELETE 情况通常不会有性能问题，因为它们通常是一次性操作。但是，如果出现问题，您可能还能够改写该命令以避免冲突并避免建立工作表。这并不总是可能的。

监控数据库性能

Adaptive Server Anywhere 提供了一组统计信息，可用来监控数据库性能。可从 Sybase Central 中访问这些统计信息，而客户端应用程序则可以通过函数形式访问这些统计信息。此外，服务器会向 Windows 性能监控器提供这些统计信息。

本节介绍如何通过客户端应用程序访问性能及相关的统计信息，如何使用 Sybase Central 监控数据库性能，如何使用 Windows 性能监控器监控数据库性能，以及如何检测文件、表和索引碎片。

通过客户端应用程序获取数据库统计信息

Adaptive Server Anywhere 提供了一组系统函数，它们可以访问每个连接、每个数据库或者整个服务器范围的信息。可用信息的类型有多种，其中包括静态信息 （如服务器名称）和详细的性能相关统计信息 （如磁盘和内存使用情况）。

检索系统信息的函数　以下函数检索系统信息：

- **property 函数**　提供给定属性在整个引擎范围的值。
- **connection_property 函数**　提供给定连接 （在缺省情况下为当前连接）的给定属性的值。
- **db_property 函数**　提供给定数据库 （在缺省情况下为当前数据库）的给定属性的值。

仅提供您要检索的属性的名称作为参数。这些函数将返回当前服务器、连接或数据库的值。

有关详细信息，请参见 『ASA SQL 参考』> 「PROPERTY 函数 [System]」、『ASA SQL 参考』> 「CONNECTION_PROPERTY 函数 [System]」和 『ASA SQL 参考』> 「DB_PROPERTY 函数 [System]」。

有关系统函数提供的属性的完整列表，请参见 『ASA SQL 参考』> 「系统函数」。

示例　以下语句将名为 server_name 的变量设置为当前服务器的名称：

```
SET server_name = property( 'name' )
```

以下查询返回当前连接的用户 ID：

```
SELECT connection_property( 'userid' )
```

以下查询返回当前数据库的根文件的文件名：

```
SELECT db_property( 'file' )
```

提高查询效率

为了提高性能，监控数据库活动的客户端应用程序应该使用 property_number 函数来标识指定的属性，然后使用该编号重复检索统计信息。下面的一组语句说明通过 Interactive SQL 执行的过程：

```
CREATE VARIABLE propnum INT ;
 CREATE VARIABLE propval INT ;
 SET propnum = property_number( 'cacheread' );
 SET propval = property( propnum )
```

通过使用属性名称的方法可用于获取多种不同的数据库统计信息，这包括从已执行的事务日志页写操作的次数和检查点操作的次数，到从内存高速缓存中读取索引叶页的次数。

您可以通过 Sybase Central 数据库管理工具以图形方式查看其中的多种统计信息。

通过 Sybase Central 监控数据库统计信息

利用 Sybase Central 性能监控器，您可以绘制能通过 Sybase Central 连接到的任何 Adaptive Server Anywhere 数据库服务器的统计信息的图形。Sybase Central 中的所有统计信息都在 [统计] 文件夹中显示。

该性能监控器的功能包括：

- 实时更新 （以可调整的间隔）
- 带颜色代码且大小可调整的图例
- 可配置的外观属性

当使用 Sybase Central 性能监控器时，请注意它会对服务器使用实际查询来收集其统计信息，因此该监控器本身会影响某些统计信息 （如 [高速缓存读取 / 秒]）。另外一种更为精确的方法是使用 Windows 性能监控器来绘制服务器统计信息的图形。

有关设置属性的信息，请参见 「设置数据库对象的属性」第 38 页。

打开 Sybase Central 性能监控器

当 [统计] 文件夹打开时，您可以在 Sybase Central 的右窗格中显示性能监控器。

❖ 打开性能监控器：

1. 在左窗格中，选择所需的服务器。
2. 在右窗格中单击 [性能监控器] 选项卡。

注意
性能监控器只绘制您已经提前添加的统计信息的图形。

另请参见

- 「添加和删除统计信息」第 205 页
- 「配置 Sybase Central 性能监控器」第 206 页
- 「通过 Windows 性能监控器监控数据库统计信息」第 207 页

添加和删除统计信息

❖ 将统计信息添加到 Sybase Central 性能监控器：

1. 在左窗格中，选择所需的服务器。
2. 在右窗格中，单击 [统计] 选项卡。
3. 右击当前未被制成图表的统计信息，然后从弹出式菜单中选择 [添加到性能监控器]。

❖ **从 Sybase Central 性能监控器中删除统计信息：**

1 执行下列操作之一：

- 如果您在使用右窗格中的 [统计] 选项卡，请右击当前已被制成图表的统计信息。
- 如果您在使用右窗格中的 [性能监控器] 选项卡，请在图例中右击所需的统计信息。

2 从弹出式菜单中选择 [从性能监控器中删除]。

提示
也可以在统计信息的属性表上将统计信息添加到性能监控器或从性能监控器中删除统计信息。

另请参见

- 「打开 Sybase Central 性能监控器」第 205 页
- 「配置 Sybase Central 性能监控器」第 206 页
- 「通过 Windows 性能监控器监控数据库统计信息」第 207 页

配置 Sybase Central 性能监控器

Sybase Central 性能监控器是可配置的；您可以选择它所使用的图形类型以及图形更新之间的时间间隔。

❖ **选择图形类型：**

1 选择 [工具] → [选项]。

2 在 [选项] 对话框中单击 [图表] 选项卡。

3 选择一种图形类型。

❖ **设置更新间隔：**

1. 选择 [工具] → [选项]。
2. 在 [选项] 对话框中单击 [图表] 选项卡。
3. 移动滑块以反映新的时间值 （或者直接在所提供的文本框中键入时间值）。

另请参见

- 「打开 Sybase Central 性能监控器」第 205 页
- 「添加和删除统计信息」第 205 页
- 「通过 Windows 性能监控器监控数据库统计信息」第 207 页

通过 Windows 性能监控器监控数据库统计信息

除了使用 Sybase Central 性能监控器之外，您也可以使用 Windows 性能监控器。

Windows 性能监控器有两项优点：

- 它提供更多的性能统计信息 （主要是与网络通信相关的统计信息）。
- 与 Sybase Central 监控器不同， Windows 监控器是非侵入性监控器。它使用共享内存方案，而不是对服务器执行查询，因此不会对统计信息本身造成影响。

有关可监控的性能统计信息的完整列表，请参见 『ASA 数据库管理指南』> 「性能监控器统计」。

注意：
Windows 性能监控器在 Windows NT、 Windows 2000 和 Windows XP 中提供。如果您同时运行 Adaptive Server Anywhere 的多个版本，也可以同时运行性能监控器的多个版本。

❖ **使用 Windows NT 中的 Windows 性能监控器：**

1 当 Adaptive Server Anywhere 引擎或数据库正在运行时，启动性能监控器：

- 选择 [开始] → [程序] → [管理工具（公用）] → [性能监控器]。

2 选择 [编辑] → [添加到图表] 或者单击工具栏上的加号按钮。随即出现 [添加到图表] 对话框。

3 从 [对象] 列表中选择以下对象之一：

- **Adaptive Server Anywhere 连接** 监控单个连接的性能。从所显示的列表中选择要监控的连接。
- **Adaptive Server Anywhere 数据库** 监控单个数据库的性能。从所显示的列表中选择要监控的数据库。
- **Adaptive Server Anywhere 引擎** 监控整个服务器范围的性能。

[计数器] 框显示您可以查看的统计信息的列表。

4 从 [计数器] 列表中，单击要查看的统计信息。若要选择多项统计信息，请在单击的同时按住 CTRL 或 SHIFT 键。

5 如果您已经选择 [Adaptive Server Anywhere 连接] 或 [Adaptive Server Anywhere 数据库]，请从 [实例] 框中选择一个实例。

6 若要添加对选定计数器的说明，请单击 [说明]。

7 若要显示该计数器，请单击 [添加]。

8 当选择完要显示的所有计数器后，单击 [完成]。

有关 Windows 性能监控器的更多信息，请参见该程序的联机帮助。

碎片

当您对数据库做出更改时，数据库文件、表和索引可能会出现碎片。碎片会降低性能。 Adaptive Server Anywhere 提供了相关的信息，可用于评估文件、表和索引的碎片程度。

本节介绍如何检测文件、表和索引中的碎片以及如何整理碎片。

文件碎片

如果数据库文件出现过多的碎片，则会破坏性能。这一问题属于磁盘碎片问题，当数据库的大小增加时，它就会变得越为重要。

当您在 Windows NT/2000/XP 上启动数据库时，数据库服务器会确定每个 dbspace 中的文件碎片数。如果碎片数大于一，服务器就会在服务器消息窗口中显示以下信息：

数据库文件 "mydatabase.db" 由 nnn 个片断组成

您也可以使用 DBFileFragments 数据库属性来获取数据库文件的碎片数。

有关详细信息，请参见 『ASA 数据库管理指南』> 「数据库级属性」。

❖ 解决文件碎片问题：

1. 将数据库单独放置在一个磁盘分区中。
2. 定期运行一个可用的 Windows 磁盘碎片整理实用程序。

表碎片

当行没有被连续存储或者行被拆分到多个页上时，由于这些行需要附加的页访问，因此会降低性能。表碎片不同于文件碎片。

Adaptive Server Anywhere 在每一页上保留了额外的空间，以允许行略微地增大。当对行的更新使行大小超过为它分配的初始空间时，该行将被拆分，而初始行位置将包含一个指针，它指向存储整个行的另一页。例

如，如果用 UPDATE 语句填充空行或将新列插入表，可能会导致严重的行拆分。当更多的行存储在单独的页上时，需要更多的时间来访问附加页。

通过指定应该为将来的更新保留的空间在表页中所占的百分比，可以减少表中的碎片数。这一 PCTFREE 规格可以用 CREATE TABLE、ALTER TABLE、 DECLARE LOCAL TEMPORARY TABLE 或 LOAD TABLE 来设置。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」、『ASA SQL 参考』> 「ALTER TABLE 语句」、『ASA SQL 参考』> 「DECLARE LOCAL TEMPORARY TABLE 语句」和 『ASA SQL 参考』> 「LOAD TABLE 语句」。

您可以使用 sa_table_fragmentation 存储过程来获取有关数据库表的碎片程度的信息。要运行此过程，必须具有 DBA 权限。以下语句调用 sa_table_fragmentation 存储过程：

```
CALL sa_table_fragmentation (['table_name'
[,'owner_name']])
```

有关详细信息，请参见 『ASA SQL 参考』> 「sa_table_fragmentation 系统过程」。

整理表碎片

当您检测到性能因为表碎片过多而降低时，以下过程将非常有用。卸载和重装数据库是较为全面的过程，因为它会整理所有表 （包括系统表）的碎片。若要整理特定表或表中某些部分的碎片，请运行 REORGANIZE TABLE。重组表的操作不会中断数据库访问。

❖ **整理数据库中所有表的碎片：**

1. 卸载数据库。
2. 重装数据库以回收磁盘空间并提高性能。

有关卸载数据库的详细信息，请参见 『ASA 数据库管理指南』> 「使用 dbunload 命令行实用程序卸载数据库」。

有关重建数据库的详细信息，请参见 『ASA 数据库管理指南』> 「重建实用程序」。

❖ **整理单个表的碎片：**

- 执行 REORGANIZE TABLE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「REORGANIZE TABLE 语句」。

索引碎片

索引用于加快对特定列的搜索速度，但如果对索引表执行了太多的 DELETE 操作，索引就会出现过多的碎片。如果碎片被频繁访问，并且高速缓存容量不足，不能保存所有索引，则会导致性能降低。

sa_index_density 存储过程提供有关数据库索引中的碎片程度的信息。要运行此过程，必须具有 DBA 权限。以下语句调用 sa_index_density 存储过程：

```
CALL sa_index_density (['table_name'[,'owner_name']])
```

如果索引的碎片太多，则可以运行 REORGANIZE TABLE。您也可以删除并重新创建该索引。但是，如果该索引是主键，您还将需要删除并重新创建外键索引。

有关详细信息，请参见 『ASA SQL 参考』> 「REORGANIZE TABLE 语句」。

有关删除索引的详细信息，请参见 「删除索引」第 73 页。

监控查询性能

Adaptive Server Anywhere 包括大量用于测试查询性能的工具。有关每种工具的完整文档可以在 *Readme.txt* 文件中找到，该文件与工具位于相同的文件夹中。

fetchtst

功能： 确定检索某一结果集所需的时间。

位置： *SQL Anywhere 9\Samples\Asa\PerformanceFetch*

odbcfet

功能： 确定检索某一结果集所需的时间。此函数与 fetchtst 相似，但是功能较少。

位置： *SQL Anywhere 9\Samples\Asa\PerformanceFetch*

instest

功能： 确定将行插入表中所需的时间。

位置： *SQL Anywhere 9\Samples\Asa\PerformanceInsert*

trantest

功能： 测量给定服务器配置 （已给定数据库设计和一组事务）可以处理的负载。

位置： *SQL Anywhere 9\Samples\Asa\PerformanceTransaction*

有关用于测量查询执行时间的系统过程的信息，请参见 『ASA SQL 参考』> 「sa_get_request_profile 系统过程」和 『ASA SQL 参考』> 「sa_get_request_times 系统过程」。

分析数据库过程

过程分析显示执行存储过程、函数、事件、系统触发器和触发器需要多长时间。您也可以查看过程中每一行的执行时间。利用数据库分析信息，您可以确定调整哪些过程就可以提高数据库的性能。

当启用分析时， Adaptive Server Anywhere 会监控使用了哪些存储过程、函数、事件、系统触发器和触发器，并保持跟踪执行它们需多长时间以及调用它们的次数。

分析信息由服务器存储在内存中，可以通过 Sybase Central 中的 [分析] 选项卡或通过 Interactive SQL 来进行查看。分析功能一旦启用，数据库就会收集分析信息，直到您禁用分析功能或服务器关机为止。

有关在 Interactive SQL 中获取分析信息的详细信息，请参见 「在 Interactive SQL 中查看过程分析信息」第 220 页。

启用过程分析

过程分析跟踪所有连接对过程和触发器的使用。您可以在 Sybase Central 或 Interactive SQL 中启用分析功能。要启用并使用过程分析，必须具有 DBA 权限。

❖ 启用分析 (Sybase Central)：

1. 作为具有 DBA 权限的用户连接到数据库。
2. 在左窗格中选择数据库。
3. 从 [文件] 菜单中选择 [属性]。

   随即出现 [数据库] 属性表。
4. 在 [分析] 选项卡上选择 [在此数据库上开始收集分析信息]。
5. 单击 [确定] 以关闭该属性表。

注意

您也可以通过在 Sybase Central 中右击数据库来启用分析。从弹出式菜单中，选择 [分析] → [开始分析]。

❖ **启用分析 (SQL)：**

1 作为具有 DBA 权限的用户连接到数据库。

2 使用 ON 设置调用 sa_server_option 存储过程。

例如，输入以下语句：

```
CALL sa_server_option ( 'procedure_profiling',
'ON')
```

如有必要，您能够看到特定用户正在使用什么过程，并且不会阻止其它连接使用数据库。当连接已存在或有多个用户使用同一用户 ID 进行连接时，这是非常有用的。

❖ **过滤用户的过程分析：**

1 以具有 DBA 权限的用户身份连接到数据库。

2 调用以下过程：

```
CALL sa_server_option ('ProfileFilterUser','userid')
```

userid 的值是正被监控的用户的名称。

重置过程分析

当您重置分析后，数据库会清除旧信息，并立即开始收集有关过程、函数、事件和触发器的新信息。

下面几节假定您已经作为具有 DBA 权限的用户连接到数据库并且过程分析已启用。

❖ **重置分析 (Sybase Central)：**

1 在左窗格中选择数据库。

2 从 [文件] 菜单中选择 [属性]。

随即出现 [数据库] 属性表。

3　在 [分析] 选项卡上单击 [立即重置]。

4　单击 [确定] 以关闭该属性表。

注意　您也可以通过在 Sybase Central 中右击数据库来重置分析。从弹出式菜单中，单击 [分析] → [重置分析信息]。

❖ **重置分析 (SQL)：**

- 使用 RESET 设置调用 sa_server_option 存储过程。

  例如，输入以下语句：

  ```
  CALL sa_server_option ('procedure_profiling',
  'RESET')
  ```

禁用过程分析

当收集完分析信息后，可以禁用分析或清除分析。如果禁用分析，数据库将停止收集分析信息，而在此刻之前收集的信息会留在 Sybase Central 中的 [分析] 选项卡上。如果清除分析，数据库将关闭分析功能，并且删除 Sybase Central 中 [分析] 选项卡上的所有分析数据。

❖ **禁用分析 (Sybase Central)：**

1　在左窗格中选择数据库。

2　从 [文件] 菜单中选择 [属性]。

随即出现 [数据库] 属性表。

3　在 [分析] 选项卡上，清除 [在此数据库上开始分析信息] 选项。

4　单击 [确定] 以关闭该属性表。

注意　您也可以通过在 Sybase Central 中右击数据库来禁用分析。从弹出式菜单中，选择 [分析] → [停止分析]。

❖ **禁用分析 (SQL)：**

- 使用 OFF 设置调用 sa_server_option 存储过程。

  例如，输入以下语句：

  ```
  CALL sa_server_option ('procedure_profiling',
  'OFF')
  ```

❖ **清除分析 (Sybase Central)：**

1. 在左窗格中选择数据库。
2. 从 [文件] 菜单中选择 [属性]。

   随即出现 [数据库] 属性表。
3. 在 [分析] 选项卡上，单击 [立即清除]。

   只有在分析已启用的情况下，才能清除分析。
4. 单击 [确定] 以关闭该属性表。

注意 您也可以通过在 Sybase Central 中右击数据库来清除分析。从弹出式菜单中，选择 [分析] → [清除分析信息]。

❖ **清除分析 (SQL)：**

- 使用 CLEAR 设置调用 sa_server_option 存储过程。

  例如，输入以下语句：

  ```
  CALL sa_server_option ('procedure_profiling',
  'CLEAR')
  ```

在 Sybase Central 中查看过程分析信息

根据您是选择查看有关整个数据库、特定类型的对象的信息还是选择查看有关特定过程的信息，过程分析会为您提供不同的信息。该信息可以通过以下方式显示：

- 数据库中所有分析过的对象的详细信息
- 所有存储过程和函数的详细信息
- 所有事件的详细信息
- 所有触发器的详细信息
- 所有系统触发器的详细信息
- 单个分析过的对象的详细信息

要查看分析信息，必须连接到数据库并启用分析。

当您查看有关整个数据库的分析信息时，以下列将出现：

- **Name**　列出对象的名称。
- **Owner**　列出对象的所有者。
- **Table**　列出触发器属于哪个表 （该列仅出现在数据库 [分析] 选项卡上）。
- **事件**　显示系统触发器的触发器类型，可以是 **Update** 或 **Delete**。
- **类型**　列出对象的类型，例如，过程。
- **# Exes.**　列出每个对象被调用的次数。
- **#msecs.**　列出每个对象的总执行时间。

这些列提供已经在数据库中执行的所有过程的分析信息的摘要。一个过程可以调用其他过程，因此列出的项目可能会多于用户专门调用的项目。

❖ 查看存储过程和函数的摘要分析信息：

1　在左窗格中选择 [过程和函数] 文件夹。

2　单击右窗格中的 [分析] 选项卡。

有关数据库中所有存储过程和函数的分析信息会出现在 [分析] 选项卡上。

❖ **查看事件的摘要分析信息：**

1 在左窗格中打开 [事件] 文件夹。

数据库中所有事件的列表将出现在右窗格中的 [事件] 选项卡上。

2 单击右窗格中的 [分析] 选项卡。

有关数据库中所有事件的分析信息随即出现在 [分析] 选项卡上。

❖ **查看触发器的摘要分析信息：**

1 在左窗格中打开 [触发器] 文件夹。

数据库中所有触发器的列表会出现在 [触发器] 选项卡上。

2 单击右窗格中的 [分析] 选项卡。

有关数据库中所有触发器的分析信息会出现在 [分析] 选项卡上。

❖ **查看系统触发器的摘要分析信息**

1 在左窗格中，打开 [系统触发器] 文件夹。

数据库中的所有触发器的列表会出现在 [系统触发器] 选项卡上。

2 单击右窗格中的 [分析] 选项卡。

有关数据库中所有系统触发器的分析信息会出现在 [分析] 选项卡上。

查看特定过程的分析信息

Adaptive Server Anywhere 提供有关单个存储过程、函数、事件和触发器的过程分析信息。 Sybase Central 所显示的有关单个过程的信息不同于它所显示的有关数据库中所有存储过程、函数、事件或触发器的信息。

当您查看有关特定过程的分析信息时，以下列将出现：

- **Calls**　列出对象被调用的次数。
- **Milliseconds**　列出每个对象的总执行时间。
- **Line**　在过程中每一行的旁边列出行号。
- **Source**　逐行显示 SQL 过程。

过程会分行显示，您可以查看哪些行具有较长的执行时间并因此可以通过更改来提高过程的性能。要访问过程分析信息，您必须连接到数据库，启用分析并具有 DBA 权限。

❖ 查看存储过程或函数的分析信息：

1　在左窗格中展开数据库。

2　在左窗格中选择 [过程和函数] 文件夹。

数据库中所有存储过程和函数的列表会出现在右窗格中的 [过程和函数] 选项卡上。

3　在左窗格中单击要分析的存储过程或函数。

4　单击右窗格中的 [分析] 选项卡。

有关特定存储过程或函数的分析信息随即出现在右窗格中的 [分析] 选项卡上。

❖ 查看事件的分析信息：

1　在左窗格中展开数据库。

2　在左窗格中选择 [事件] 文件夹。

数据库中所有事件的列表会出现在右窗格中的 [事件] 选项卡上。

3　在左窗格中单击要分析的事件。

4　单击右窗格中的 [分析] 选项卡。

有关特定事件的分析信息随即出现在右窗格中的 [分析] 选项卡上。

❖ 查看触发器的分析信息：

1. 在左窗格中展开数据库。
2. 在左窗格中打开 [触发器] 文件夹。

   所有触发器的列表会出现在右窗格中的 [触发器] 选项卡上。
3. 在右窗格中选择您要分析的触发器。
4. 单击右窗格中的 [分析] 选项卡。

   有关特定触发器的分析信息随即出现在右窗格中的 [分析] 选项卡上。

❖ 查看系统触发器的分析信息

1. 在左窗格中展开数据库。
2. 在左窗格中，打开 [系统触发器] 文件夹。

   所有系统触发器的列表会出现在右窗格中的 [系统触发器] 选项卡上。
3. 在右窗格中选择您要分析的系统触发器。
4. 单击右窗格中的 [分析] 选项卡。

   有关特定系统触发器的分析信息会出现在右窗格中的 [分析] 选项卡上。

在 Interactive SQL 中查看过程分析信息

您可以使用存储过程来查看过程分析信息。无论您是在 Sybase Central 中还是在 Interactive SQL 中查看分析信息，该信息都是相同的。

sa_procedure_profile_summary 存储过程提供有关数据库中所有过程的信息。您可以使用该过程在同一结果集中查看存储过程、函数、事件、系统触发器和触发器的分析数据。以下参数将限制该过程返回的行。

- **p_object_name**　指定要分析的对象的名称。
- **p_owner_name**　指定要分析的对象的所有者。
- **p_table_name**　指定要分析其触发器的表。
- **p_object_type**　指定要分析的对象的类型。您可以从以下五个选项中进行选择。通过选择其中一个值，可以将结果集限制为指定类型的对象。
  - **P**　存储过程
  - **F**　函数
  - **T**　触发器
  - **E**　事件
  - **S**　系统触发器
- **p_ordering**　指定结果集的排序顺序。

请注意，由于一个过程可以调用其他过程，因此所列出的项目可能会多于用户专门调用的项目。

下面几节假定您已经作为具有 DBA 权限的用户连接到数据库并且过程分析已启用。

❖ 查看所有过程的摘要分析信息：

1. 执行 sa_procedure_profile_summary 存储过程。

   例如，输入以下语句：

   ```
   CALL sa_procedure_profile_summary
   ```

2. 从 [SQL] 菜单中选择 [执行]。

   [结果] 窗格中会出现一个结果集，该结果集包含有关数据库中所有过程的信息。

有关 sa_procedure_profile_summary 存储过程的详细信息，请参见『ASA SQL 参考』> 「sa_procedure_profile_summary 系统过程」。

在 Interactive SQL 中查看特定过程的分析信息

sa_procedure_profile 存储过程提供有关特定过程中各个行的信息。结果集包括行号、执行时间以及占过程中所有行的总执行时间的百分比。您可以使用以下参数来限制该过程返回的行：

- **p_object_name** 指定要分析的对象的名称。
- **p_owner_name** 指定要分析的对象的所有者。
- **p_table_name** 指定要分析其触发器的表。

如果查询中没有包含任何参数，则该过程将返回已调用的所有过程的分析信息。

❖ 查看过程中特定行的分析信息：

1. 执行 sa_procedure_profile 存储过程。

   例如，输入以下语句：

   ```
   CALL sa_procedure_profile
   ```

2. 从 [SQL] 菜单中选择 [执行]。

   [结果] 窗格中会显示一个结果集，该结果集包含各个过程行的分析信息。

有关 sa_procedure_profile 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_procedure_profile 系统过程」。

第 2 部分　查询和修改数据

这一部分介绍了如何查询和修改数据。它包括几个有关查询（从简单到复杂）的章节，以及有关插入、删除和更新数据方面的内容。

第 6 章

查询：从表中选择数据

关于本章

SELECT 语句可以从数据库中检索数据。您可以使用该语句在一个或多个表中检索行的子集以及在一个或多个表中检索列的子集。

本章集中介绍单表 SELECT 语句的基本概念。 SELECT 语句的高级用法在本手册的稍后部分进行介绍。

查询概述

查询请求数据库中的数据并接收结果。此过程也称为数据检索。所有 SQL 查询都是使用 SELECT 语句表达的。

查询是由子句组成的

您可以通过子句构造 SELECT 语句。在下面的 SELECT 语法中，每个新行都是一个单独的子句。这里只列出最常见的子句。

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

SELECT 语句中的子句如下：

- SELECT 子句指定您要检索的列。它是 SELECT 语句中唯一必需的子句。
- FROM 子句指定从中请求列的表。在所有从表中检索数据的查询中都需要该子句。没有 FROM 子句的 SELECT 语句具有不同的含义，本章不讨论它们。

  虽然大多数查询都是对表执行操作，但是查询也可以从具有列和行的其它对象 （包括视图、其它查询 （派生表）和存储过程结果集）中检索数据。有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。
- ON 子句指定 FROM 子句中的表将如何进行连接。该子句只用于多表查询，本章中不进行介绍。

  有关多表查询的信息，请参见 「连接：从多个表检索数据」第 281 页。
- WHERE 子句指定表中您要查看的行。
- GROUP BY 子句用于集合数据。

- HAVING 子句指定要在其上收集集合数据的行。
- ORDER BY 子句对结果集中的行进行排序。（缺省情况下，从关系数据库返回行的顺序没有任何意义。

  有关 GROUP BY、 HAVING 和 ORDER BY 子句的信息，请参见「对查询结果进行汇总、分组和排序」第 251 页。

大多数子句是可选的，但是如果包括这些子句，它们就必须以正确的顺序出现。

有关 SELECT 语句语法的详细信息，请参见 『ASA SQL 参考』> 「SELECT 语句」。

SQL 查询

在本手册中，显示 SELECT 语句和其它 SQL 语句时，每个子句都在单独一行上， SQL 关键字以大写字母显示。这并不是必需的。您可以使用大写或小写字母键入 SQL 关键字，也可以在任何位置换行。

关键字和换行符

例如，以下 SELECT 语句从 Contact 表中查找居住在 California 的联系人的名和姓。

```
SELECT first_name, last_name
 FROM Contact
 WHERE state = 'CA'
```

如果按以下方式输入该语句，虽然不便于阅读，但同样有效：

```
SELECT first_name,
 last_name from contact
 wHere state
  = 'CA'
```

字符串和标识符的区分大小写

在 Adaptive Server Anywhere 数据库中，标识符 （即表名、列名等）是不区分大小写的。

缺省情况下，字符串是不区分大小写的，因此 'CA'、 'ca'、 'cA' 和 'Ca' 都是等同的，但是如果您将数据库创建为区分大小写的数据库，则字符串的大小写就有意义了。示例数据库是不区分大小写的。

限定标识符

如果引用的对象存在不明确性，则可以限定数据库标识符的名称。例如，示例数据库中多个表都具有名为 city 的列，因此您可能必须用表的名称限定对 city 的引用。在一个较大的数据库中，您可能还需要使用表的所有者的名称来标识该表。

```
SELECT DBA.contact.city
 FROM contact
 WHERE state = 'CA'
```

由于本章中的示例涉及单表查询，通常不用表的名称或表所有者的名称对语法模型和示例中的列名进行限定。

省去这些元素是为了便于阅读；包括这些限定符也没有任何错误。

本章其余几节更详细地分析 SELECT 语句的语法。

SELECT 列表：指定列

选择列表

选择列表通常由一系列使用逗号分隔的列名组成，或将星号用作速记代表所有列。

一般地说，选择列表包括一个或多个表达式，由逗号分隔。选择列表的一般语法如下：

SELECT *expression* [, *expression*]…

如果列表中的任何表名或列名不符合有效标识符的规则，则您必须将该标识符括在双引号中。

选择列表表达式可以包括 *（所有列）、一组列名、字符串、列标题和包括算术运算符的表达式。您还可以包括集合函数，相关内容在 「对查询结果进行汇总、分组和排序」第 251 页中进行了介绍。

有关表达式可由哪些元素组成的详细信息，请参见『ASA SQL 参考』>「表达式」。

以下几节提供可在选择列表中使用的多种表达式的示例。

从表中选择所有列

在 SELECT 语句中，星号 (*) 具有特殊含义。它代表 FROM 子句指定的所有表中的所有列名。如果您要查看某个表中的所有列，则可以使用星号节省键入时间并避免键入错误。

当使用 SELECT * 时，列返回的顺序是创建表时定义这些列的顺序。

选择表中所有列的语法是：

```
SELECT *
 FROM table-expression
```

SELECT * 查找当前在表中的所有列，因此表结构中的更改 （例如添加、删除或重命名列）将自动修改 SELECT * 的结果。逐个列出列可以使您更精确地控制结果。

示例

以下语句检索 department 表中的所有列。未包括 WHERE 子句；因此该语句检索表中的每一行：

```
SELECT *
 FROM department
```

结果如下：

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
…	…	…

在 SELECT 关键字后面按顺序列出表中所有列名将得到完全相同的结果：

```
SELECT dept_id, dept_name, dept_head_id
 FROM department
```

像列名一样，可以使用表名对 "*" 进行限定，如在下面的查询中：

```
SELECT department.*
 FROM department
```

从表中选择特定的列

若要只选择表中特定的列，请使用此语法：

SELECT *column_name* [, *column_name*]…
FROM *table-name*

您必须使用逗号将每个列名与它后面的列名分隔开。例如：

```
SELECT emp_lname, emp_fname
 FROM employee
```

重新排列列的顺序

您列出列名的顺序决定列显示的顺序。以下两个示例显示如何指定显示的列顺序。两个示例都查找并显示 department 表所有五行中的部门名称和标识号，但是显示顺序不同。

```
SELECT dept_id, dept_name
 FROM department
```

dept_id	dept_name
100	R & D
200	Sales
300	Finance
400	Marketing
…	…

```
SELECT dept_name, dept_id
 FROM department
```

dept_name	dept_id
R & D	100
Sales	200
Finance	300
Marketing	400
…	…

重命名查询结果中的列

查询结果由一组列组成。缺省情况下，每个列的标题都是选择列表中提供的表达式。

当显示查询结果时，每个列的缺省标题都是创建该列时赋予它的名称。您可以通过以下方式之一指定不同的列标题 （即**别名**）：

SELECT *column-name* **AS** *alias*

SELECT *column-name alias*

SELECT *alias* = *column-name*

提供别名可以产生更加便于阅读的结果。例如，您可以在部门列表中将 dept_name 更改为 Department，如下所示：

```
SELECT dept_name AS Department,
     dept_id AS "Identifying Number"
 FROM department
```

部门	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
…	…

在别名中使用空格和关键字

dept_id 的 Identifying Number 别名括在双引号中，因为它是一个标识符。如果要在别名中使用关键字，您也可以使用双引号。例如，如果没有引号，则以下查询无效：

```
SELECT dept_name AS Department,
     dept_id AS "integer"
 FROM department
```

如果您要确保与 Adaptive Server Enterprise 兼容，应使用 30 个字节 （或更少）的带引号的别名。

查询结果中的字符串

到目前为止，您看到的 SELECT 语句产生的结果只包含 FROM 子句指定的表中的数据。通过将字符串括在单引号中并在选择列表中用逗号将它们与其它元素分开，还可以在查询结果中显示字符串。

字符串中若要包括引号，您需要在它前面再放一个引号。

例如：

```
SELECT 'The department''s name is' AS "Prefix",
     Department = dept_name
 FROM department
```

前缀	部门
部门的名称为	R & D
部门的名称为	Sales
部门的名称为	Finance
部门的名称为	Marketing
部门的名称为	Shipping

计算 SELECT 列表中的值

选择列表中的表达式可能比单纯的列名或字符串更加复杂。例如，您可以使用选择列表中数字列中的数据执行计算。

算术运算

为了说明您可以在选择列表中执行的数字运算，我们从示例数据库中产品的名称、库存数量和单价列表开始。

```
SELECT name, quantity, unit_price
 FROM product
```

name	quantity	unit_price
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
…	…	…

假设惯例是在当某种产品的库存为 10 件时补充产品库存。以下查询列出在再次订购前每种产品必须卖出的数量：

```
SELECT name, quantity - 10
     AS "Sell before reorder"
 FROM product
```

name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
…	…

您还可以结合多个列中的值。以下查询列出库存中每种产品的总价值：

```
SELECT name,
     quantity * unit_price AS "Inventory value"
 FROM product
```

name	Inventory value
Tee Shirt	252
Tee Shirt	756
Tee Shirt	1050
Baseball Cap	1008
…	…

算术运算符优先级

当表达式中有多个算术运算符时，先计算乘法、除法和模，然后计算减法和加法。当表达式中的所有算术运算符优先级级别相同时，执行顺序为从左到右。括号中的表达式优先级高于所有其它运算符。

例如，以下 SELECT 语句计算库存中每种产品的总价值，然后从该值中减去 5 美元。

```
SELECT name, quantity * unit_price - 5
 FROM product
```

为了避免误解，建议您使用括号。以下查询和上一个查询具有相同的含义并给出相同的结果，但可能会发现它更易于理解：

```
SELECT name, ( quantity * unit_price ) - 5
 FROM product
```

有关运算符优先级的详细信息，请参见 『ASA SQL 参考』> 「运算符优先级」。

字符串运算

您可以使用字符串连接运算符来连接字符串。您可以使用 || （与 SQL/92 兼容）或 + （受 Adaptive Server Enterprise 支持）作为连接运算符。

以下示例说明选择列表中字符串连接运算符的用法：

```
SELECT emp_id, emp_fname || ' ' || emp_lname AS Name
 FROM employee
```

emp_id	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
…	…

日期和时间运算

虽然您可以在日期和时间列上使用运算符，但此操作通常涉及函数的使用。有关 SQL 函数的信息，请参见 『ASA SQL 参考』> 「SQL 函数」。

消除重复查询结果

可选的 DISTINCT 关键字可以消除 SELECT 语句结果中重复的行。

如果不指定 DISTINCT，您将得到所有行，其中包括重复行。或者，您可以在选择列表前指定 ALL 以得到所有行。为了和 SQL 的其它实现兼容， Adaptive Server 语法允许使用 ALL 显式请求所有行。 ALL 是缺省设置。

例如，如果您在不使用 DISTINCT 的情况下搜索 contact 表中的所有城市，您会得到 60 行：

```
SELECT city
 FROM contact
```

您可以使用 DISTINCT 消除重复条目。以下查询只返回 16 行：

```
SELECT DISTINCT city
 FROM contact
```

NULL 值是重复的

DISTINCT 关键字将 NULL 值视为是相互重复的。换句话说，当 SELECT 语句中包括 DISTINCT 时，不管遇到多少个 NULL 值，结果中只返回一个 NULL。

FROM 子句：指定表

每个涉及表、视图或存储过程中数据的 SELECT 语句都需要有 FROM 子句。

FROM 子句可以包括链接两个或多个表的 JOIN 条件，也可以包括与其它查询 （派生表）的连接。有关这些功能的信息，请参见 「连接：从多个表检索数据」第 281 页。

限定表名

在 FROM 子句中，始终允许使用表和视图的全称语法，例如：

```
SELECT select-list
 FROM owner.table_name
```

如果某个用户 ID 与当前连接的用户 ID 不同，或者不是当前连接的用户 ID 所属组的名称，只有在这样的用户 ID 拥有对象时，才有必要限定表、视图和过程名称。

使用相关名

您可以赋予表名一个相关名以减少需要键入的字符。您可以在 FROM 子句中指派相关名，方法是在表名之后键入该相关名，如下所示：

```
SELECT d.dept_id, d.dept_name
 FROM Department d
```

对 Department 表的所有其它引用 （例如，在 WHERE 子句中）都*必须*使用该相关名。相关名必须符合有效标识符的规则。

有关 FROM 子句的详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。

从对象而不是表中查询

FROM 子句中最常用的元素是表名。也有可能会从其它数据库对象（这些数据库对象具有与表类似的结构，即，有明确定义的行集和列集）中查询行。除了从表和视图进行查询以外，您还可以使用返回结果集的派生表 （它们是 SELECT 语句）或存储过程。

例如，以下查询会对存储过程的结果集执行操作。

```
SELECT *
 FROM sp_customer_products( 149 )
```

有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。

WHERE 子句：指定行

SELECT 语句中的 WHERE 子句指定具体检索哪些行的搜索条件。一般格式为：

SELECT *select_list*
    FROM *table_list*
    WHERE *search-condition*

WHERE 子句中的搜索条件 （也称为限定或谓语）包括以下几种：

- **比较运算符** (=, <,> 等）例如，您可以列出收入超过 $50,000 的所有雇员：

```
SELECT emp_lname
 FROM employee
 WHERE salary > 50000
```

- **范围** （BETWEEN 和 NOT BETWEEN）例如，您可以列出收入在 $40,000 到 $60,000 的所有雇员：

```
SELECT emp_lname
 FROM employee
 WHERE salary BETWEEN 40000 AND 60000
```

- **列表** （IN、 NOT IN）例如，您可以列出 Ontario、 Quebec 或 Manitoba 的所有客户：

```
SELECT company_name , state
 FROM customer
 WHERE state IN( 'ON', 'PQ', 'MB')
```

- **字符匹配** （LIKE 和 NOT LIKE）例如，您可以列出电话号码以 415 开头的所有客户。（电话号码在数据库中作为字符串存储）：

```
SELECT company_name , phone
 FROM customer
 WHERE phone LIKE '415%'
```

- **未知值** （IS NULL 和 IS NOT NULL）例如，您可以列出有经理的所有部门：

```
SELECT dept_name
 FROM Department
 WHERE dept_head_id IS NOT NULL
```

- **组合** （AND、 OR）例如，您可以列出收入超过 $50,000 并且名字以字母 A 开头的所有雇员。

  ```
  SELECT emp_fname, emp_lname
   FROM employee
   WHERE salary > 50000
   AND emp_fname like 'A%'
  ```

另外， WHERE 关键字可以引入以下内容：

- **Transact-SQL 连接条件** 连接在 「连接：从多个表检索数据」第 281 页中有所介绍。

有关搜索条件的详细信息，请参见 『ASA SQL 参考』> 「搜索条件」。

以下几节介绍如何使用 WHERE 子句。

在 WHERE 子句中使用比较运算符

您可以在 WHERE 子句中使用比较运算符。运算符遵循以下语法：

WHERE *expression comparison-operator expression*

有关比较运算符的详细信息，请参见 『ASA SQL 参考』> 「比较运算符」。有关表达式可由哪些元素组成的说明，请参见 『ASA SQL 参考』> 「表达式」。

有关比较的说明

- **排序顺序** 比较字符数据时， "<" 表示排序顺序中稍前的， ">" 表示排序顺序中稍后的。排序顺序是由创建数据库时选择的归类决定的。您可以通过对数据库运行 *dbinfo* 命令行实用程序查找归类：

  ```
  dbinfo -c "uid=DBA;pwd=SQL"
  ```

  您还可以通过 Sybase Central 查找归类。它在数据库属性表的 [扩展信息] 选项卡上。

- **尾随空白** 当创建数据库时，您可以指示是否忽略尾随空白或不忽略尾随空白以进行比较。

缺省情况下，创建数据库时不忽略尾随空白。例如， 'Dirk' 和 'Dirk ' 不同。您可以创建具有空格填充的数据库，以便忽略尾随空白。在 Adaptive Server Enterprise 数据库中，缺省情况下忽略尾随空白。

- **比较日期** 比较日期时， "<" 表示日期较早， ">" 表示日期较晚。
- **区分大小写** 当创建数据库时，您可以指示字符串比较是否区分大小写。

  缺省情况下，创建数据库时不区分大小写。例如， 'Dirk' 和 'DIRK' 相同。您可以创建区分大小写的数据库，区分大小写是 Adaptive Server Enterprise 数据库的缺省行为。

这里是一些使用比较运算符的 SELECT 语句：

```
SELECT *
 FROM product
 WHERE quantity < 20
 SELECT E.emp_lname, E.emp_fname
 FROM employee E
 WHERE emp_lname > 'McBadden'
 SELECT id, phone
 FROM contact
 WHERE state  != 'CA'
```

NOT 运算符

NOT 运算符对表达式进行否定。以下两个查询中的任何一个都将查找所有价格为 $10 （或不到）的 T 恤衫和棒球帽。但是，请注意否定逻辑运算符 (NOT) 和否定比较运算符 (!>) 之间的差别。

```
SELECT id, name, quantity
 FROM product
 WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
 AND NOT unit_price > 10
 SELECT id, name, quantity
 FROM product
 WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
 AND unit_price !> 10
```

在 WHERE 子句中使用范围 （between 和 not between）

BETWEEN 关键字指定包含的范围，将搜索该范围的上限值、下限值以及介于这两个值之间的值。

❖ 列出价格在 $10 到 $15 （含）之间的所有产品：

- 键入以下查询：

```
SELECT name, unit_price
 FROM product
 WHERE unit_price BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

您可以使用 NOT BETWEEN 查找不在该范围中的所有行。

❖ 列出价格低于 $10 或高于 $15 的所有产品：

- 执行以下查询：

```
SELECT name, unit_price
 FROM product
 WHERE unit_price NOT BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7

name	unit_price
…	…

在 WHERE 子句中使用列表

IN 关键字可用于选择与一组值中任何一个值匹配的值。该表达式可以是一个常量或一个列名，而列表可以是一组常量，通常为一个子查询。

例如，在不使用 IN 的情况下，如果您需要一个所有居住在 Ontario、Manitoba 或 Quebec 的联系人姓名和州的列表：

```
SELECT company_name , state
 FROM customer
 WHERE state = 'ON' OR state = 'MB' OR state = 'PQ'
```

但是，如果使用 IN，您将得到相同的结果。 IN 关键字后面的各项必须使用逗号分隔开并括在括号中。在字符、日期或时间值的两边使用单引号。例如：

```
SELECT company_name , state
 FROM customer
 WHERE state IN( 'ON', 'MB', 'PQ')
```

或许 IN 关键字最重要的用法是用在嵌套查询 （也称作子查询）中。

WHERE 子句中的匹配字符串

LIKE 关键字指示其后的字符串是一个匹配模式。 LIKE 用于字符、二进制或日期和时间。

LIKE 的语法为：

{ **WHERE** | **HAVING** } *expression* [**NOT**] **LIKE** *match-expression*

将要进行匹配的表达式与匹配表达式进行比较，匹配表达式可以包含以下特殊符号：

符号	含义
%	匹配具有任意数目字符 （没有字符或多个字符）的任意字符串
_	匹配任意一个字符
[分类符]	方括号中的分类符可以采用以下形式： • **范围** 范围的格式为 *rangespec1-rangespec2*，其中 *rangespec1* 表示字符范围的开始，连字符表示范围，而 *rangespec2* 表示字符范围的结束 • **集合** 集合可由任何一组离散的值以任何顺序组成。例如， [a2bR]。 请注意范围 [a-f] 以及集合 [abcdef] 和 [fcbdae] 都返回相同一组值。
[^ 分类符]	分类符前面的脱字符 (^) 表示不包含。 [^a-f] 表示不在范围 a-f 中； [^a2bR] 表示不是 a、 2、 b 和 R。

您可以将列数据与包含该表中所显示的通配符的常量、变量或其它列匹配。如果使用常量，您应该将匹配子串和字符串括在单引号中。

示例

以下所有示例都将 LIKE 用于 Contact 表的 last_name 列。查询格式如下：

```
SELECT last_name
 FROM contact
 WHERE last_name LIKE match-expression
```

第一个示例输入以下内容

```
SELECT last_name
 FROM contact
 WHERE last_name LIKE 'Mc%'
```

匹配表达式	说明	返回值
'Mc%'	搜索所有以字母 **Mc** 开头的名称	McEvoy

匹配表达式	说明	返回值
'%er'	搜索所有以字母 **er** 结尾的名称	Brier, Miller, Weaver, Rayner
'%en%'	搜索所有包含字母 **en** 的名称。	Pettengill, Lencki, Cohen
'_ish'	搜索所有以字母 **ish** 结尾的四个字母的名称。	Fish
'Br[iy][ae]r'	搜索 Brier、 Bryer、 Briar 或 Bryar。	Brier
'[M-Z]owell'	搜索所有以 **owell** 结尾、以从 M 到 Z 范围内的一个字母开头的名称。	Powell
'M[^c]%'	搜索所有以 M 开头、第二个字母不是 c 的名称	Moore, Mulley, Miller, Masalsky

通配符需要使用 LIKE

使用通配符时如果没有 LIKE，则会将该通配符视为**文字**，而不是模式：它们只代表它们自己的值。以下查询试图查找只由四个字符 415% 组成的所有电话号码。它不查找以 415 开头的电话号码。

```
SELECT phone
 FROM Contact
 WHERE phone = '415%'
```

将 LIKE 用于日期和时间值

不仅可以将 LIKE 用于字符数据，您还可以将 LIKE 用在日期和时间字段上。当您将 LIKE 用于日期和时间值时，日期将被转换为标准的 DATETIME 格式，然后转换为 VARCHAR。

这便是搜索 DATETIME 值时使用 LIKE 的一个功能。由于日期和时间条目可能包含多种日期格式，所以必须小心编写等同性测试才能成功。

例如，如果您将值 9:20 和当前日期插入到一个名为 arrival_time 的列中，则子句：

```
WHERE arrival_time = '9:20'
```

无法找到该值，原因是该条目中既有日期又有时间。但是，下面的子句将找到 9:20 这个值：

```
WHERE arrival_time LIKE '%09:20%'
```

使用 NOT LIKE

对于 NOT LIKE，您可以使用的通配符与可用于 LIKE 的通配符相同。若要查找 Contact 表中地区号不是 415 的所有电话号码，您可以使用以下这两个查询中的任何一个：

```
SELECT phone
 FROM Contact
 WHERE phone NOT LIKE '415%'
 SELECT phone
 FROM Contact
 WHERE NOT phone LIKE '415%'
```

字符串和引号

当输入或搜索字符和日期数据时，您必须将其括在单引号中，如以下示例所示。

```
SELECT first_name, last_name
 FROM contact
 WHERE first_name = 'John'
```

如果 quoted_identifier 数据库选项设置为 OFF （缺省设置为 ON），则您还可以在字符或日期数据两边使用双引号。

在 Interactive SQL 会话期间， Interactive SQL 会自动将 quoted_identifier 设置为 ON。

❖ 为当前用户 ID 将 quoted_identifier 选项设置为 OFF：

- 键入以下命令：

  ```
  SET OPTION quoted_identifier = 'OFF'
  ```

提供 quoted_identifier 选项是为了与 Adaptive Server Enterprise 兼容。缺省情况下， Adaptive Server Enterprise 选项将 quoted_identifier 设为 OFF， Adaptive Server Anywhere 选项将 quoted_identifier 设为 ON。

字符串中的引号

在字符条目中指定文字引号有两种方法。第一种方法是使用两个连续的引号。例如，如果您以一个单引号开始字符条目并想包括一个单引号作为条目的一部分，则请使用两个单引号：

```
'I don''t understand.'
```

使用双引号 （将 quoted_identifier 设置为 OFF）：

```
"He said, ""It is not really confusing."""
```

第二种方法只有在 quoted_identifier 设置为 OFF 时才适用，该方法将一种引号括在另一种引号中。换句话说，将包含双引号的条目括在单引号中，反之亦然。这里是一些示例：

```
'George said, "There must be a better way."'
 "Isn't there a better way?"
 'George asked, "Isn''t there a better way?"'
```

未知值：NULL

列中的 NULL 意味着用户或应用程序未在该列中进行输入。该列的数据值未知或不可用。

NULL 不同于零 （数字值）或空白 （字符值）。相反， NULL 值允许您将数字列有意的零输入或字符列有意的空白输入和未输入区别开来，未输入对数字列和字符列来说都是 NULL。

输入 NULL

可以在允许 NULL 值的列中 （根据 CREATE TABLE 语句中所指定的）以两种方法输入 NULL：

- **缺省值** 如果未输入任何数据并且列没有任何其它缺省设置，则输入 NULL。
- **显式输入** 您可以通过键入单词 NULL （不带引号）显式输入值 NULL。

  如果在字符列中键入的单词 NULL 带有引号，则将其作为数据（而不是空值）对待。

例如， department 表的 dept_head_id 列允许空值。您可以为没有经理的部门输入两行，如下所示：

```
INSERT INTO department (dept_id, dept_name)
 VALUES (201, 'Eastern Sales')
 INSERT INTO department
 VALUES (202, 'Western Sales', null)
```

当检索 NULL 时

当检索 NULL 时， Interactive SQL 中的查询结果在适当的位置显示 "(NULL)"：

```
SELECT *
 FROM department
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(NULL)
202	Western Sales	(NULL)

基于 NULL 对列进行测试

您可以在搜索条件中使用 IS NULL 将列值和 NULL 进行比较，并根据比较的结果选择它们或执行特定的操作。只有返回值 TRUE 的列才被选中或引发指定的操作；那些返回 FALSE 或 UNKNOWN 的列不会被选中，也不引发指定的操作。

以下示例只选择 unit_price 小于 $15 或为 NULL 的行：

```
SELECT quantity , unit_price
 FROM product
 WHERE unit_price < 15
 OR unit_price IS NULL
```

将任何值与 NULL 比较的结果都是 UNKNOWN，这是因为无法确定 NULL 是否等于 （或不等于）某个给定的值或另一个 NULL。

有些条件从不返回 TRUE，因此使用这些条件的查询不返回结果集。例如，永远无法确定以下比较为 TRUE，这是因为 NULL 表示具有未知值：

```
WHERE column1 > NULL
```

此逻辑还适用于在 WHERE 子句中使用两个列名 （即连接两个表）的情况。包含条件

```
WHERE column1 = column2
```

的子句不会返回列中包含 NULL 的行。

您还可以使用此模式查找 NULL 或非 NULL：

```
WHERE column_name IS [NOT] NULL
```

例如：

```
WHERE advance < $5000
 OR advance IS NULL
```

有关详细信息，请参见 『ASA SQL 参考』> 「NULL 值」。

NULL 的属性

下面详述 NULL 的属性。

- **FALSE 和 UNKNOWN 之间的差异** 虽然 FALSE 和 UNKNOWN 都不返回值，但是因为与 FALSE ("NOT FALSE") 相对的是 TRUE，所以 FALSE 和 UNKNOWN 之间存在重要的逻辑差异。例如，

  ```
  1 = 2
  ```

  计算得到 FALSE，而其相反命题

  ```
  1 != 2
  ```

  计算得到 TRUE。但是 "not unknown" 仍是未知的。如果比较中包括空值，则您不能对表达式进行否定以得到相对的一组行或相反的真假值。

- **用某个值替代 NULL** 使用 ISNULL 内置函数用某个特定的值替代空值。替代的目的只是为了进行显示，未影响实际列值。语法是：

ISNULL(*expression*, *value*)

例如，使用以下语句从 test 中选择所有行，然后将列 t1 中的所有空值显示为值 "unknown"。

```
SELECT ISNULL(t1, 'unknown')
 FROM test
```

- **计算得到 NULL 的表达式**　如果任何操作数为空值，则使用算术或位运算符的表达式计算得到 NULL。例如：

  ```
  1 + column1
  ```

  如果 column1 为 NULL，则计算得到 NULL。

- **连接字符串和 NULL**　如果您将字符串和 NULL 连接，则表达式计算得到字符串。例如：

  ```
  SELECT 'abc' || NULL || 'def'
  ```

  返回字符串 **abcdef**。

使用逻辑运算符连接条件

逻辑运算符 AND、 OR 和 NOT 可用来在 WHERE 子句中连接搜索条件。

使用 AND

AND 运算符连接两个或多个条件并且只有当所有条件都为真时才返回结果。例如，以下查询只查找联系人的姓是 Purcell 并且联系人的名是 Beth 的那些行。它不查找 Beth Glassmann 所在的行。

```
SELECT *
 FROM contact
 WHERE first_name = 'Beth'
     AND last_name = 'Purcell'
```

使用 OR

OR 运算符也连接两个或多个条件，但只要*任何*一个条件为真，它就返回结果。以下查询搜索 first_name 列中包含 Elizabeth 各种变形形式的那些行。

```
SELECT *
 FROM contact
 WHERE first_name = 'Beth'
     OR first_name = 'Liz'
```

使用 NOT

NOT 运算符对它后面的表达式进行否定。以下查询列出所有不居住在 California 的联系人：

```
SELECT *
 FROM contact
 WHERE NOT state = 'CA'
```

当语句中使用多个逻辑运算符时，通常先计算 AND 运算符，然后再计算 OR 运算符。您可以使用括号更改执行的顺序。例如：

```
SELECT *
 FROM contact
 WHERE ( city = 'Lexington'
           OR city = 'Burlington' )
     AND state = 'MA'
```

第 7 章

对查询结果进行汇总、分组和排序

关于本章

集合函数可以显示指定列中值的汇总。通过集合函数，您还可以使用 GROUP BY 子句、 HAVING 子句和 ORDER BY 子句来对查询结果进行分组和排序，并可使用 UNION 运算符来合并查询结果。

本章将介绍如何对查询结果进行分组和排序。

使用集合函数汇总查询结果

您可以将集合函数应用于表中的所有行、应用于 WHERE 子句所指定的表的子集或者应用于表中一组行或多组行。 Adaptive Server Anywhere 可以从应用了集合函数的每一组行生成一个值。

下面是可用集合函数中的一部分：

- **avg （表达式）** 对于返回的行，计算所提供的表达式的平均值。
- **count （表达式）** 在表达式不是 NULL 的情况下计算所提供的组中的行数。
- **count(*)** 计算每一组中行的数目。
- **list （字符串表达式）** 一个包含逗号分隔的列表的字符串，列表由每一组行中 *string-expr* 的所有值组成。
- **max （表达式）** 对于返回的行，计算表达式的最大值。
- **min （表达式）** 对于返回的行，计算表达式的最小值。
- **stddev （表达式）** 对于返回的行，计算表达式的标准偏差。
- **sum （表达式）** 对于返回的行，计算表达式的总和。
- **variance （表达式）** 对于返回的行，计算表达式的方差。

有关集合函数的完整列表，请参见 『ASA SQL 参考』> 「集合函数」。

您可以将可选关键字 DISTINCT 与 AVG、 SUM、 LIST 和 COUNT 结合使用，以在应用集合函数前消除重复值。

语法语句引用的表达式通常是列名称。引用的表达式也可以是较为一般性的表达式。

例如，通过以下语句，您可以求出在将 1 美元与每一个价格相加之后所有产品的平均价格：

```
SELECT AVG (unit_price + 1)
 FROM product
```

示例

以下查询将根据雇员表中的年薪来计算工资单总额：

```
SELECT SUM(salary)
 FROM employee
```

若要使用集合函数，您必须提供函数名称，后随一个表达式，然后将对表达式的值进行运算。表达式（在此示例中是 salary 列）是函数的参数，并且必须在括号内指定。

在哪些地方可以使用集合函数

可以在选择列表中使用集合函数（如前面各例中所示），也可以在包括 GROUP BY 子句的选择语句的 HAVING 子句中使用集合函数。

有关 HAVING 子句的详细信息，请参见 「HAVING 子句：选择数据组」第 264 页。

您不能在 WHERE 子句或 JOIN 条件中使用集合函数。但是，在其选择列表中具有集合函数的 SELECT 语句通常包括 WHERE 子句，该子句限制可以对其应用集合函数的行。

如果 SELECT 语句包括 WHERE 子句，而不包括 GROUP BY 子句，则集合函数可以为 WHERE 子句指定的一部分行只生成一个值。

只要在不包括 GROUP BY 子句的 SELECT 语句中使用集合函数，集合函数就只生成一个值。无论集合函数是用于表中所有行，还是用于 WHERE 子句定义的一部分行，上述原则都适用。

您可以在同一选择列表中使用多个集合函数，并且可以在一个 SELECT 语句中生成多个标量集合。

集合函数和外部引用

Adaptive Server Anywhere 第 8 版及更高版本遵守 SQL/99 标准，该标准阐明了当集合函数出现于子查询中时如何使用集合函数。这些更改将影响为该软件的以前版本编写的语句的行为：以前正确的查询现在可能产生错误消息，结果集也可能发生更改。

当集合函数出现于子查询中并且集合函数引用的列是外部引用时，整个集合函数本身现在被视为外部引用。这意味着集合函数现在是在外部块中计算的，而不在子查询中计算，并且变成了子查询内的常量。

在子查询中使用外部引用集合函数时受到以下限制：

- 外部引用集合函数只能出现在位于 SELECT 列表或 HAVING 子句中的子查询中，并且这些子句必须位于紧接的外部块中。
- 外部引用集合函数只能包含一个外部列引用。
- 本地列引用和外部列引用不能在同一集合函数中混用。

通过重写集合函数以使它只包括本地引用，可以解决与新标准相关的某些问题。例如，子查询 (SELECT MAX(S.y + R.y) FROM S) 同时包含本地列引用 (S.y) 和外部列引用 (R.y)，现在这是非法的。可以将该子查询改写为 (SELECT MAX(S.y) + R.y FROM S)。在改写的内容中，集合函数只具有本地列引用。如果外部引用集合函数出现在非 SELECT 或 HAVING 的子句中，则可以采用同一方法进行改写。

示例

以下查询在 Adaptive Server Anywhere 第 7 版中生成以下结果。

```
SELECT name, (SELECT SUM(p.quantity)
                FROM sales_order_items)
 FROM product p
```

name	sum(p.quantity)
Tee shirt	30,716
Tee shirt	59,238

在更高版本中，同一查询会生成错误消息 ASA 错误 -149: 对 'name' 的函数或列引用还必须出现在 GROUP BY 中。该语句不再有效的原因在于：外部引用集合函数 sum(p.quantity) 现在是在外部块中计算的。在更高版本中，该查询在语义上等效于以下语句 （只是 Z 不作为结果集的一部分显示）：

```
SELECT name,
        SUM(p.quantity) as Z,
        (SELECT Z FROM sales_order_items)
 FROM product p
```

由于现在是由外部块来计算集合函数，因此，该外部块被视作分组查询，并且列名必须出现在 GROUP BY 子句中，才能出现在 SELECT 列表中。

集合函数和数据类型

一些集合函数仅对于某些类型的数据有意义。例如，您只能将 SUM 和 AVG 与数字列一起使用。

不过，您可以使用 MIN 求出字符类型的列中的最小值，即求出按字母表的顺序最靠前的字母。

```
SELECT MIN(last_lname)
 FROM  contact
```

使用 COUNT(*)

COUNT(*) 函数不要求使用表达式作为参数，因为按照定义它不使用与任何特定列有关的信息。 COUNT(*) 函数用于求出表中的总行数。以下语句将求出雇员的总数：

```
SELECT COUNT(*)
 FROM employee
```

COUNT(*) 返回指定表中的行数，重复行也计算在内。该函数将每一行都计算在内，并且包括包含 NULL 的那些行。

与其它集合函数一样，您也可以将 count(*) 与选择列表中的其它集合函数、 WHERE 子句等合并起来：

```
SELECT count(*), AVG(unit_price)
 FROM product
 WHERE  unit_price > 10
```

count(*)	AVG(product.unit_price)
5	18.2

将集合函数与 DISTINCT 一起使用

DISTINCT 关键字可以和 SUM、 AVG 和 COUNT 一起使用。在您使用 DISTINCT 时，在计算总和、平均值或计数前消除重复值。

例如，为了求出具有联系人的不同城市的数目，键入：

```
SELECT count(DISTINCT city)
 FROM contact
```

count(distinct contact.city)
16

您可以在查询中将多个集合函数与 DISTINCT 一起使用。每个 DISTINCT 都是独立计算的。例如：

```
SELECT count( DISTINCT first_name ) "first names",
        count( DISTINCT last_name ) "last names"
 FROM contact
```

first names	last names
48	60

集合函数和 NULL

对于除 COUNT(*) 之外的所有函数，执行了集合函数的列中的任何 NULL 都将被忽略。如果一个列中的所有值均为 NULL，则 COUNT(column_name) 返回 0。

如果没有一行满足 WHERE 子句中指定的条件，则 COUNT 返回 0。其它函数都返回 NULL。以下是一些示例：

```
SELECT COUNT (DISTINCT name)
 FROM product
 WHERE unit_price > 50
```

count(DISTINCT name)
0

```
SELECT AVG(unit_price)
 FROM product
 WHERE unit_price > 50
```

AVG(product.unit_price)
(NULL)

GROUP BY 子句：将查询结果划分为组

GROUP BY 子句可以将表的输出划分为若干个组。您可以使用 GROUP BY 子句按一个或多个列名称进行分组，或者通过在表达式中使用数值数据类型，按计算出的列的结果进行分组。

将 GROUP BY 与集合函数一起使用

GROUP BY 子句几乎总是出现在包括集合函数的语句中，在此情况下，集合函数会为每一个组生成一个值。这些值被称作**矢量集合**。（请记住，标量集合是由不具有 GROUP BY 子句的集合函数生成的单个值。

示例

以下查询列出了每一种产品的平均价格：

```
SELECT name, AVG(unit_price) AS Price
 FROM product
 GROUP BY name
```

name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
…	…

由具有集合函数和 GROUP BY 的 SELECT 语句生成的摘要值 （矢量集合）在每一行结果中以列的形式出现。而由具有集合函数但没有 GROUP BY 的查询生成的摘要值 （标量集合）也以列的形式出现，但仅具有一行。例如：

```
SELECT AVG(unit_price)
 FROM product
```

AVG(product.unit_price)
13.3

了解 GROUP BY

当查询涉及 GROUP BY 子句时，弄清哪些查询有效以及哪些查询无效可能十分困难。本节将介绍如何使用具有 GROUP BY 的查询，以便您可以更好地理解查询的结果和有效性。

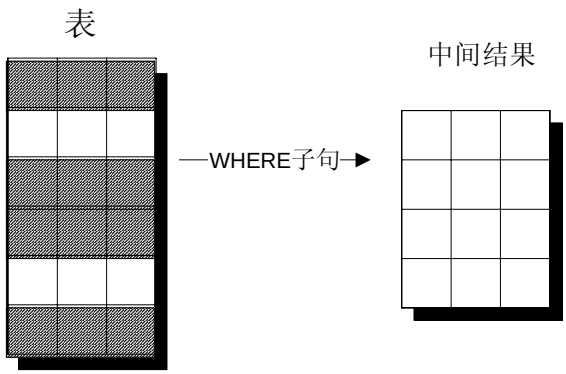
具有 GROUP BY 的查询是如何执行的

假设有一个以下形式的单表查询：

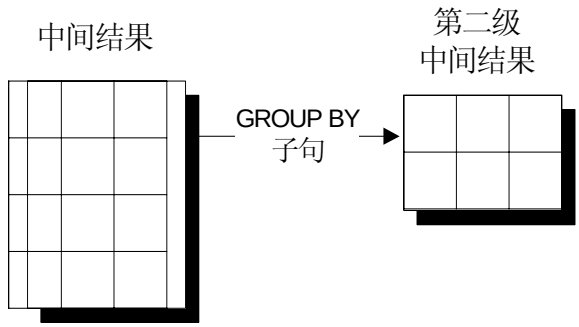
SELECT *select-list*
    FROM *table*
    WHERE *where-search-condition*
    GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
    HAVING *having-search-condition*

该查询可以被视为是按以下方式执行的：

1. **应用 WHERE 子句** 这就生成只包含该表的某些行的中间结果。



2. **将结果划分为若干个组** 按照 GROUP BY 子句的命令，此操作将生成一个中间结果，每一组具有一行。对于每一组，生成的每一行都包含 *group-by-expression*，并且在 *select-list* 和 *having-search-condition* 中包含计算出的集合函数。

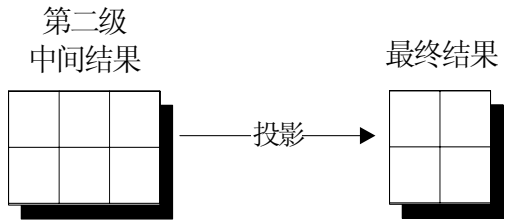


3. **应用任何 ROLLUP 操作**　将作为 ROLLUP 操作的一部分计算的分类汇总行添加到结果集中。

   有关详细信息，请参见 「使用 ROLLUP」第 362 页。

4. **应用 HAVING 子句**　此时，此第二个中间结果中的不满足 HAVING 子句条件的所有行都将被删除。

5. **将结果投影到显示屏中**　此操作只从第 3 步中获取需要在查询的结果集中显示的那些列，即，它只获取与来自 *select-list* 的表达式相对应的那些列。



此进程对具有 GROUP BY 子句的查询提出如下要求：

- 首先计算 WHERE 子句。因此，只对满足 WHERE 子句的那些行计算任何集合函数。

- 最终结果集是根据第二个中间结果 （该结果保留已分区的行）建立的。第二个中间结果保留与 *group-by-expression* 相对应的那些行。因此，如果一个非集合函数的表达式出现在 *select-list* 中，则该表达式也必须出现在 *group-by-expression* 中。在投影步骤中不可以执行任何函数计算。
- 一个表达式包括在 *group-by-expression* 中，但不包括在 *select-list* 中。它被投影到结果之外。

具有多个列的 GROUP BY

您可以在 GROUP BY 子句中列出多个表达式，以便使多个组嵌套；即，可以通过表达式的任意组合对表进行分组。

以下查询列出了产品的平均价格，首先按名称分组，然后按尺寸分组：

```
SELECT name, size, AVG(unit_price)
 FROM product
 GROUP BY name, size
```

name	size	AVG(product.unit_price)
Tee Shirt	Small	9
Tee Shirt	Medium	14
Tee Shirt	One size fits all	14
Baseball Cap	One size fits all	9.5
…	…	…

GROUP BY 中的不位于选择列表中的列

Sybase 对 SQL/92 标准的一个扩展是允许对不位于选择列表中的表达式执行 GROUP BY 子句， Adaptive Server Enterprise 和 Adaptive Server Anywhere 均支持这一扩展。例如，以下查询列出了每一城市中联系人的数目：

```
SELECT state, count(id)
 FROM contact
 GROUP BY state, city
```

WHERE 子句和 GROUP BY

您可以在语句中将 WHERE 子句与 GROUP BY 一起使用。在 GROUP BY 子句前对 WHERE 子句求值。在执行任何分组前消除不满足 WHERE 子句中的条件的行。下面是一个示例：

```
SELECT  name, AVG(unit_price)
 FROM product
 WHERE id > 400
 GROUP BY name
```

只有其 id 值超过 400 的那些行才包括在用于生成查询结果的组中。

示例

以下查询说明了如何在一个查询中使用 WHERE、 GROUP BY 和 HAVING 子句：

```
SELECT name, SUM(quantity)
 FROM product
 WHERE name LIKE '%shirt%'
 GROUP BY name
 HAVING SUM(quantity) > 100
```

name	SUM(product.quantity)
Tee Shirt	157

在此示例中：

- WHERE 子句只包括其名称具有 *shirt* 一词 （Tee Shirt、Sweatshirt）的那些行。
- GROUP BY 子句收集具有普通名的行。
- SUM 集合函数将计算每一组中的产品总数。
- HAVING 子句从最终结果中排除其库存总计不超过 100 的那些组。

HAVING 子句：选择数据组

HAVING 子句用于限制查询返回的行。它也为 GROUP BY 子句设置条件，方法与 WHERE 为 SELECT 子句设置条件的方法相类似。

HAVING 子句搜索条件基本上等同于 WHERE 搜索条件，只是 WHERE 搜索条件不能包括集合函数，而 HAVING 搜索条件经常包括集合函数。下面的示例是合法的：

```
HAVING AVG(unit_price) > 20
```

但下面的示例不合法：

```
WHERE AVG(unit_price) > 20
```

将 HAVING 与集合函数一起使用

以下语句是将 HAVING 子句与集合函数一起使用的简单示例。

要列出那些有多种尺寸或颜色的产品，您需要一个查询来按名称给 product 表中的行分组，但要去除只包含一种独特产品的组：

```
SELECT name
 FROM product
 GROUP BY name
 HAVING COUNT(*) > 1
```

name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

有关何时可在 HAVING 子句中使用集合函数的信息，请参见 「在哪些地方可以使用集合函数」第 253 页。

不与集合函数一起使用 HAVING

HAVING 子句也可以不与集合函数一起使用。

以下查询对产品进行分组，然后将结果集限制为只包含名称以 B 开头的那些组。

```
SELECT name
 FROM product
 GROUP BY name
 HAVING name LIKE 'B%'
```

name
Baseball Cap

在 HAVING 中有多个条件

在 HAVING 子句中可以包括多个条件。可以用 AND、 OR 或 NOT 运算符将这些条件组合起来，如下面的示例所示。

要列出那些有多种尺寸或颜色而且一个版本的价格超过 10 美元的产品，您需要一个查询来按名称将 product 表中的行分组，但要去除只包含一种独特产品的组，而且要去除那些最高单价低于 10 美元的组。

```
SELECT name
 FROM product
 GROUP BY name
 HAVING COUNT(*) > 1
 AND MAX(unit_price) > 10
```

name
Tee Shirt
Sweatshirt

ORDER BY 子句：对查询结果进行排序

ORDER BY 子句允许按一列或多列对查询结果进行排序。每一次排序都可以是升序 (ASC) 或降序 (DESC)。如果没有指定任何排序顺序，则按升序 (ASC) 进行排序。

简单示例

以下查询将返回按名称排序的结果：

```
SELECT id, name
 FROM product
 ORDER BY name
```

id	name
400	Baseball Cap
401	Baseball Cap
700	Shorts
600	Sweatshirt
…	…

按多列排序

如果您在 ORDER BY 子句中指定了多列，则排序是嵌套的。

以下语句首先以升序按名称对 product 表中的衬衫进行排序，然后在每一名称内按数量 （降序）进行排序：

```
SELECT id, name, quantity
 FROM product
 WHERE name like '%shirt%'
 ORDER BY name, quantity DESC
```

id	name	quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75
301	Tee Shirt	54

id	name	quantity
…	…	…

使用列位置

您可以在选择列表中使用列的位置号，而不使用列名称。列名称和选择列表编号可以混合使用。以下两段语句都与前面的一段语句生成相同的结果。

```
SELECT id, name, quantity
 FROM product
 WHERE name like '%shirt%'
 ORDER BY 2, 3 DESC
 SELECT id, name, quantity
 FROM product
 WHERE name like '%shirt%'
 ORDER BY 2, quantity DESC
```

大多数 SQL 版本都要求 ORDER BY 项出现在选择列表中，但 Adaptive Server Anywhere 没有这样的限制。以下查询是按数量对结果进行排序的，尽管该列不出现在选择列表中：

```
SELECT id, name
 FROM product
 WHERE name like '%shirt%'
 ORDER BY 2, quantity DESC
```

ORDER BY 和 NULL

对于 ORDER BY，以升序排列时 NULL 排在所有其他值之前。

ORDER BY 和大小写

ORDER BY 子句对混合大小写数据的影响取决于在创建数据库时指定的数据库归类和大小写。

显式限制查询返回的行数

您可以使用 FIRST 或 TOP 关键字限制查询的结果集中包括的行数。这些关键字用于包括 ORDER BY 子句的查询。

示例

以下查询返回在按姓氏对雇员进行排序时首先出现的雇员的信息：

```
SELECT FIRST *
 FROM employee
 ORDER BY emp_lname
```

以下查询返回按姓氏排序时出现的前五个雇员：

```
SELECT TOP 5 *
 FROM employee
 ORDER BY emp_lname
```

在使用 TOP 时，您也可以使用 START AT 来提供偏移。以下语句列出按姓氏以降序进行排序时出现的第五个和第六个雇员：

```
SELECT TOP 2 START AT 5 *
 FROM employee
 ORDER BY emp_lname DESC
```

FIRST 和 TOP 只能与 ORDER BY 子句联合使用，以确保获得一致的结果。如果使用 FIRST 或 TOP 时没有 ORDER BY，则会触发语法警告，并且很可能产生无法预知的结果。

注意
'start at' 值必须大于 0。当 'top' 为常量时，其值必须大于 0 ；当 'top' 为变量时，其值必须大于或等于 0。

ORDER BY 和 GROUP BY

您可以使用 ORDER BY 子句以一种特定的方式对 GROUP BY 的结果排序。

示例

以下查询将求出每一产品的平均价格并按平均价格对结果排序：

```
SELECT name, AVG(unit_price)
 FROM product
 GROUP BY name
 ORDER BY AVG(unit_price)
```

name	AVG(product.unit_price)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333

name	AVG(product.unit_price)
Shorts	15
…	…

使用 UNION、 INTERSECT 和 EXCEPT 对查询结果执行集合运算

本节中介绍的运算符对两个或多个查询的结果执行集合运算。尽管使用 WHERE 子句或 HAVING 子句中的操作也可以执行许多运算，但是有一些运算使用除这些基于集合的运算符之外的任何方法都很难执行。例如：

- 当以未规范化方式保存数据时，您可能希望将表面上根本不相同的信息集合到单个结果集中，即使表是不相关的。
- 集合运算符处理 NULL 的方式与在 WHERE 子句或 HAVING 子句中是不同的。在 WHERE 子句或 HAVING 子句中，具有完全相同的非空条目的两个包含空值行不被视为完全相同，因为这两个 NULL 值没有定义为完全相同。集合运算符将两个这样的行视为相同。

  有关 NULL 和集合运算的详细信息，请参见 「集合运算符和 NULL」第 274 页。

有关详细信息，请参见 『ASA SQL 参考』> 「EXCEPT 操作」、『ASA SQL 参考』> 「INTERSECT 操作」和 『ASA SQL 参考』> 「UNION 操作」。

使用 UNION 运算将集合组合在一起

UNION 运算符可以将两个或多个查询的结果组合到一个结果集中。

缺省情况下， UNION 运算符可以从结果集中删除重复行。如果使用 ALL 选项，则不删除重复行。结果集中的列与引用的第一个表中的列具有相同的名称。可以使用任何数目的 UNION 运算符。例如：

```
x UNION y UNION z
```

缺省情况下，按从左到右的顺序对包含多个 UNION 运算符的语句求值。可以使用括号来指定求值的顺序。

例如，以下两个表达式不是等效的，因为它们从结果集中删除重复行的方式不同：

```
x UNION ALL (y UNION z)
 (x UNION ALL y) UNION z
```

在第一个表达式中，在 y 和 z 之间的 UNION 中消除重复行，在该集和 x 间的 UNION 中没有消除重复行。在第二个表达式中，重复行包括在 x 和 y 之间的 UNION 中，但随后在与 z 的 UNION 中被消除。

使用 EXCEPT 和 INTERSECT

EXCEPT 运算列出两个结果集之间的差异。下面的结构列出出现在 query-1 的结果集中但没有出现在 query-2 的结果集中的所有那些行。

query-1
EXCEPT
query-2

INTERSECT 运算列出同时出现在这两个结果集中的行。下面的结构列出同时出现在 query-1 和 query-2 的结果集中的所有那些行。

query-1
INTERSECT
query-2

与 UNION 运算一样， EXCEPT 和 INTERSECT 都采用 ALL 修饰符，该修饰符可防止从结果集中消除重复行。

有关详细信息，请参见 『ASA SQL 参考』> 「EXCEPT 操作」和 『ASA SQL 参考』> 「INTERSECT 操作」。

集合运算的规则

以下规则适用于 UNION、 EXCEPT 和 INTERSECT 运算：

- **选择列表中项的数目相同**　查询中的所有选择列表中的表达式（例如列名称、算术表达式和集合函数）数目必须相同。以下语句是无效的，因为第一个选择列表比第二个选择列表长：

```
-- This is an example of an invalid statement
 SELECT store_id, city, state
 FROM stores
 UNION
 SELECT store_id, city
 FROM stores_east
```

- **数据类型必须匹配** SELECT 列表中的相应表达式必须属于同一数据类型，或者在两种数据类型之间必须能够进行隐式数据转换，或者应提供显式转换。

  例如，在 CHAR 数据类型的列和 INT 数据类型的列之间是不可能进行 UNION、 INTERSECT 或 EXCEPT 运算的，除非提供了显式转换。但是，在 MONEY 数据类型的列和 INT 数据类型的列之间是可以进行集合运算的。

- **列排序** 您必须以相同的顺序将相应的表达式放置于集合运算的各个查询中，因为集合运算将按 SELECT 子句的各个查询中给定的顺序一一比较这些表达式。

- **多个集合运算** 您可以将若干个集合运算排列在一起，如以下示例所示：

  ```
  SELECT city AS Cities
   FROM contact
   UNION
   SELECT city
   FROM customer
   UNION
   SELECT city
   FROM employee
  ```

  对于 UNION 运算，查询的顺序是不重要的。对于 INTERSECT，当存在两个或多个查询时顺序是很重要的。对于 EXCEPT，顺序始终都很重要。

- **列标题** 执行 UNION 运算后生成的表中的列名称出自该语句中的第一个单独的查询。如果您要为结果集定义新的列标题，则可以在第一个查询的选择列表中这样做，如以下示例所示：

```
SELECT city AS Cities
 FROM contact
 UNION
 SELECT city
 FROM customer
```

在下面的查询中，列标题仍为 city，因为在 UNION 语句的第一个查询中定义了该列标题。

```
SELECT city
 FROM contact
 UNION
 SELECT city AS Cities
 FROM customer
```

或者，您也可以使用 WITH 子句定义列名。例如：

```
WITH V( Cities )
 AS ( SELECT city
      FROM contact
      UNION
      SELECT city
      FROM customer )
 SELECT * FROM V
```

- **对结果进行排序**　您可以使用 SELECT 语句的 WITH 子句对选择列表中的列名进行排序。例如：

```
WITH V( cityname )
 AS ( SELECT Cities = city
      FROM contact
      UNION
      SELECT city
      FROM customer )
 SELECT * FROM V
 ORDER BY cityname
```

或者，您也可以在查询列表的结尾使用单个 ORDER BY 子句，但必须使用整数而不是列名称，如以下示例所示：

```
SELECT Cities = city
 FROM contact
 UNION
 SELECT city
 FROM customer
 ORDER BY 1
```

集合运算符和 NULL

集合运算符 UNION、 EXCEPT 和 INTERSECT 处理 NULL 的方式与 NULL 在搜索条件中时处理的方式不同。此差异是使用集合运算符的主要原因之一。

在比较行时，集合运算符将 NULL 值视为彼此相等。相反，在搜索条件中将 NULL 与 NULL 进行比较时，结果是未知的 （不为真）。

此差异的一个特别有用的结果是：`query-1 EXCEPT ALL query-2` 的结果集中的行数*始终*是各个查询的结果集中的行数之差。

以两个表 T1 和 T2 为例，其中每个表都具有以下列：

```
col1 INT,
 col2 CHAR(1)
```

表中的数据如下：

- 表 T1。

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

- 表 T2

col1	col2
1	a
2	x

col1	col2
3	(NULL)

一个询问同时出现在 T1 和 T2 中的行的查询如下：

```
SELECT T1.col1, T1.col2
 FROM T1 JOIN T2
 ON T1.col1 = T2.col2
 AND T1.col2 = T2.col2
```

T1.col1	T1.col2
1	a

行 (3, NULL) 不出现在结果集中，因为 NULL 和 NULL 之间的比较不为真。相比之下，使用 INTERSECT 运算符处理该问题，则会包括一个有 NULL 的行：

```
SELECT col1, col2
 FROM T1
 INTERSECT
 SELECT col1, col2
 FROM T2
```

col1	col2
1	a
3	(NULL)

以下查询使用搜索条件列出 T1 中那些不出现在 T2 中的行：

```
SELECT col1, col2
 FROM T1
 WHERE col1 NOT IN (
     SELECT col1
     FROM t2
     WHERE t1.col2 = t2.col2 )
```

```
OR col2 NOT IN (
    SELECT col2
    FROM t2
    WHERE t1.col1 = t2.col1 )
```

col1	col2
2	b
3	(NULL)
4	(NULL)
3	(NULL)
4	(NULL)

比较没有排除 T1 中包含 NULL 的行。相比之下，使用 EXCEPT ALL 处理该问题会排除在这两个表中都出现的包含 NULL 的行。在此情况下，会认为 T2 中的 (3, NULL) 行与 T1 中的 (3, NULL) 行相同。

```
SELECT col1, col2
 FROM T1
 EXCEPT ALL
 SELECT col1, col2
 FROM T2
```

col1	**col2**
2	b
3	(NULL)
4	(NULL)
4	(NULL)

EXCEPT 运算符仍然有更多的限制。它从 T1 中将两个 (3, NULL) 行都去除，并将一个 (4, NULL) 行作为重复项排除。

```
SELECT col1, col2
 FROM T1
 EXCEPT
 SELECT col1, col2
 FROM T2
```

col1	col2
2	b
4	(NULL)

标准和兼容性

本节将介绍有关 Adaptive Server Anywhere GROUP BY 子句的标准和兼容性方面的内容。

GROUP BY 和 SQL/92 标准

对于 GROUP BY 的 SQL/92 标准的要求如下：

- 在 SELECT 子句的表达式中使用的列必须位于 GROUP BY 子句中。否则，使用该列的表达式应该是集合函数。
- GROUP BY 表达式只能包含来自选择列表的列名称，而不能包含只用作矢量集合的参数的那些列名称。

具有矢量集合函数的标准 GROUP BY 的结果将生成一行，行中的每一个组都具有一个值。

Adaptive Server Anywhere 和 Adaptive Server Enterprise 支持对 HAVING 的扩展，允许集合函数不位于选择列表和 GROUP BY 子句中。

与 Adaptive Server Enterprise 的兼容性

Adaptive Server Enterprise 支持若干个对 GROUP BY 子句的扩展，而这些扩展在 Adaptive Server Anywhere 中是不受支持的。它们包括以下各项：

- **选择列表中未分组的列** Adaptive Server Enterprise 允许选择列表中的列名称不出现在 GROUP BY 子句中。例如，在 Adaptive Server Enterprise 中，以下语句是有效的：

  ```
  SELECT name, unit_price
   FROM product
   GROUP BY name
  ```

  而在 Adaptive Server Anywhere 中则不支持此语法。

- **嵌套的集合函数**　以下查询将矢量集合嵌套在标量集合内，这种用法在 Adaptive Server Enterprise 中是有效的，但在 Adaptive Server Anywhere 中则无效：

```
SELECT MAX(AVG(unit_price))
 FROM product
 GROUP BY name
```

- **GROUP BY 和 ALL**　Adaptive Server Anywhere 不支持在 GROUP BY 子句中使用 ALL。

- **不带 GROUP BY 的 HAVING**　Adaptive Server Anywhere 不支持没有 GROUP BY 子句的 HAVING，除非 SELECT 和 HAVING 子句中的所有表达式都是集合函数。例如，以下查询在 Adaptive Server Enterprise 中是有效的，但在 Adaptive Server Anywhere 中不支持该查询：

```
--unsupported syntax
 SELECT unit_price
    FROM product
    HAVING COUNT(*) > 8
```

  不过，以下语句在 Adaptive Server Anywhere 中是有效的，因为函数 MAX 和 COUNT 都是集合函数：

```
SELECT MAX(unit_price)
 FROM product
 HAVING COUNT(*) > 8
```

- **HAVING 条件**　Adaptive Server Enterprise 支持对 HAVING 的扩展，允许非集合函数不位于选择列表和 GROUP BY 子句中。在 Adaptive Server Anywhere 中只允许此类型的集合函数。

- **具有 ORDER BY 或 GROUP BY 的 DISTINCT**　Adaptive Server Enterprise 允许 ORDER BY 或 GROUP BY 子句中使用的列不出现在选择列表中，甚至在 SELECT DISTINCT 查询中也是如此。这可能导致在 SELECT DISTINCT 结果集中出现重复值。 Adaptive Server Anywhere 不支持此用法。

- **UNION 中的列名称** Adaptive Server Enterprise 允许在联合查询的 ORDER BY 子句使用列。在 Adaptive Server Anywhere 中，ORDER BY 子句必须使用整数，以标记用来对结果进行排序的列。

第 8 章

连接：从多个表检索数据

关于本章

在创建数据库时，您通过将特定于不同对象的信息放置在多个不同的表中，而不是放置在具有许多冗余条目的一个大表中来规范化数据。

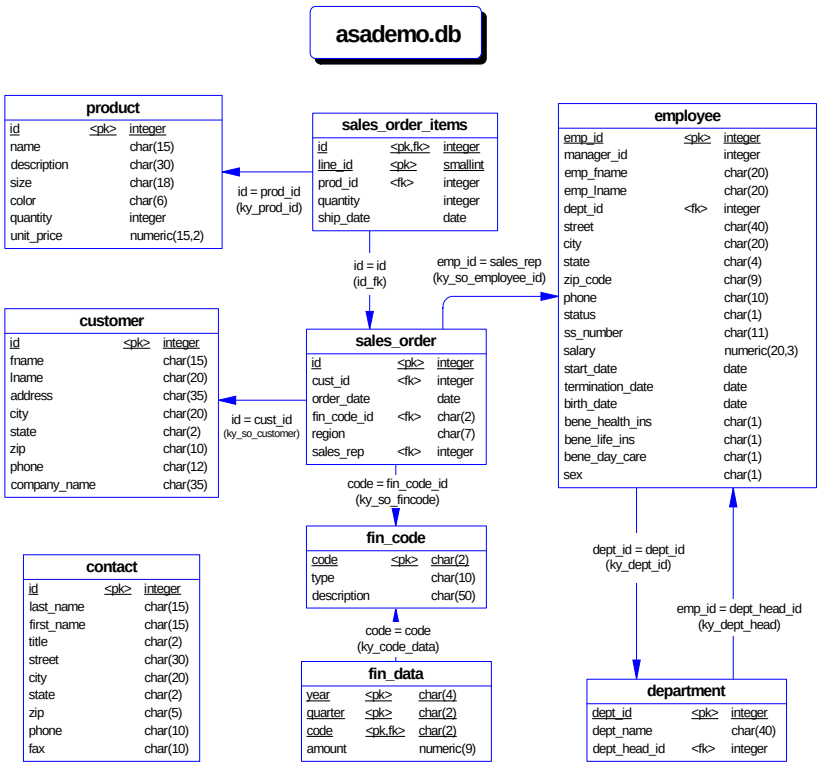
连接操作通过使用来自两个或多个表（或多个视图）的信息，重新创建更大的表。使用不同的连接，您可以构建多种此类虚拟表，每一个表都适合于执行某种特定任务。

在开始前

本章假设您了解一些查询知识以及选择语句的语法。在「查询：从表中选择数据」第 225 页中提供了与查询有关的信息。

示例数据库模式

本章将频繁引用示例数据库。下面的图中给出该示例数据库，其中包含与这些表关联的外键的名称。示例数据库保留在名为 *asademo.db* 的文件中，并且位于您的安装目录中。



连接的工作原理

关系数据库将有关不同类型的对象的信息存储在不同的表中。例如，与雇员有关的信息在一个表中出现，而与部门有关的信息在另一个表中出现。雇员表包含诸如雇员姓名和地址等有关信息。部门表包含与一个部门有关的信息，例如部门名称以及部门的主管。

大多数问题只能通过使用来自不同表的信息的组合来回答。例如，您可能要回答 [谁管理销售部门？] 这样的问题。若要找到此人的姓名，您必须使用来自部门表的信息确定相关人员，然后在雇员表中查找此人的姓名。

连接是一种通过建立来自多个表的信息的新虚拟表来回答此类问题的方法。例如，您可以通过将雇员表中包含的信息与部门表中包含的信息进行组合来创建部门领导的列表。您使用 FROM 子句指定哪些表包含所需信息。

若要使连接有效，您必须组合每个表的正确的列。为了列出部门领导，组合后的表中的每一行都应包含部门的名称以及管理该部门的雇员的姓名。通过指定连接操作的具体类型或使用 ON 短语，您可以控制在组合表中匹配列的方式。

连接概述

连接操作通过比较指定的列中的值来组合多个表中的行。本节概要介绍 Adaptive Server Anywhere 连接语法。本节中的所有概念都会在以后各节中更详细地探讨。

FROM 子句

使用 FROM 子句指定要连接哪些基表、临时表、视图或派生表。 FROM 子句可用于 SELECT 或 UPDATE 语句中。 FROM 子句的简写语法如下：

FROM *table_expression*, …

其中：

table_expression:
table | *view* | *derived table* | *joined table* | **(** *table_expression*, … **)**

table or view:
[*userid***.**] *table-or-view-name* [[**AS**] *correlation-name*]

derived table:
(*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, … **)**]

joined table:
table_expression join_operator table_expression [**ON** *join_condition*]

join_operator:
[**KEY** | **NATURAL**] [*join_type*] **JOIN** | **CROSS JOIN**

join_type:
INNER | **FULL** [**OUTER**] | **LEFT** [**OUTER**] | **RIGHT** [**OUTER**]

注意

您不能将 ON 短语与 CROSS JOIN 一起使用。

有关完整语法，请参见 『ASA SQL 参考』> 「FROM 子句」。

有关 ON 短语的语法，请参见 『ASA SQL 参考』> 「搜索条件」。

连接条件

表可以使用**连接条件**进行连接。连接条件只不过是搜索条件。它基于列中各值间的关系从连接的各表中选择行的子集。例如，以下查询从 product 和 sales_order_items 表中检索数据。

```
SELECT *
 FROM product JOIN sales_order_items
    ON product.id = sales_order_items.prod_id
```

此查询中的连接条件是

```
product.id = sales_order_items.prod_id
```

此连接条件意味着，只有各行在两个表中具有同一产品 ID 后，在结果集中才可以组合这些行。

连接条件可以是显式连接条件或生成的连接条件。**显式连接条件**是放置于 ON 短语或 WHERE 子句中的连接条件。以下查询使用 ON 短语。该查询生成两个表的矢量积 （各行的所有组合），但不包括 ID 号不匹配的那些行。结果是一个客户列表，列表中具有这些客户的订单的详细信息。

```
SELECT *
 FROM customer JOIN sales_order
   ON sales_order.cust_id = customer.id
```

生成的连接条件是您指定 KEY JOIN 或 NATURAL JOIN 时自动创建的连接条件。在键连接的情况下，生成的连接条件基于各表间的外键关系。在自然连接的情况下，生成的连接条件基于具有相同名称的列。

提示：
键连接语法和自然连接语法都是快捷方式：您可以通过使用关键字 JOIN （*而不*使用 KEY 或 NATURAL），然后在 ON 短语中显式指示同一连接条件，获得相同的结果。

在您将 ON 短语与键连接或自然连接一起使用时，所使用的连接条件是显式指定的连接条件与生成的连接条件的**联合**。这意味着，这些连接条件用关键字 AND 组合在一起。

连接的表

Adaptive Server Anywhere 支持以下类别的连接的表。

- **交叉连接** 两个表的交叉连接生成来自这两个表的各行的所有可能组合。结果集的规模是第一个表中的行数乘以第二个表中的行数。交叉连接也称作矢量积或笛卡儿积。您不能将 ON 短语与交叉连接一起使用。
- **键连接（缺省）** 连接条件是基于已内置于数据库中的外键关系自动生成的。如果在使用 JOIN 关键字时没有指定连接类型并且没有 ON 短语，则缺省是键连接。
- **自然连接** 连接条件是基于具有相同名称的列自动生成的。
- **使用 ON 短语连接** 您指定显式连接条件。在与键连接或自然连接一起使用时，该连接条件既包含生成的连接条件，也包含显式连接条件。在与 JOIN 关键字一起使用（而不是与 KEY 或 NATURAL 关键字一起使用）时，没有生成的连接条件。您不能将 ON 子句与交叉连接一起使用。

内连接和外连接

可以通过指定 INNER、 LEFT OUTER、 RIGHT OUTER 或 FULL OUTER 对键连接、自然连接和具有 ON 子句的连接加以限定。缺省值是 INNER。在使用关键字 LEFT、 RIGHT 或 FULL 时，关键字 OUTER 是可选的。

在内连接中，结果中的每一行都满足连接条件。

在左外连接或右外连接中，为一个表保留所有行，另一个表不满足连接条件的行返回空值。例如，在右外连接中，保留右侧的表，左侧的用空值补充。

在完全外连接中，保留这两个表的所有行；并且为不满足连接条件的行提供空值。

连接两个表

为了理解简单内连接的计算方式，请参见以下查询。它回答的问题是：哪种产品尺码的订购数量与库存数量相同？

```
SELECT DISTINCT name, size,
     sales_order_items.quantity
 FROM product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
   AND product.quantity = sales_order_items.quantity
```

name	size	quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

该查询解释如下。请注意，这是对如何处理此查询的概念说明，用来阐释涉及连接的查询的语义。它不表示 Adaptive Server Anywhere 实际计算结果集的方式。

- 创建 product 表和 sales_order_items 表的矢量积。矢量积包含来自两个表的各行的每一组合。
- 排除产品 ID 不相同的所有行 （由于连接条件 `product.id = sales_order_items.prod_id`）。
- 排除数量不相同的所有行 （由于连接条件 `product.quantity = sales_order_items.quantity`）。
- 创建具有以下三列的结果表：product.name、 product.size 和 sales_order_items.quantity。
- 排除所有重复行 （由于 DISTINCT 关键字）。

有关如何计算外连接的说明，请参见 「外连接」第 297 页。

连接两个以上的表

使用 Adaptive Server Anywhere，对您可以连接的表的数目没有固定限制。

在连接两个以上的表时，可以选择使用括号。如果您没有使用括号，Adaptive Server Anywhere 会按从左到右的顺序计算语句的值。因此，`A JOIN B JOIN C` 等效于 `(A JOIN B) JOIN C`。其他示例：

```
SELECT *
 FROM A JOIN B JOIN C JOIN D
```

等效于

```
SELECT *
 FROM ((A JOIN B) JOIN C) JOIN D
```

只要连接两个以上的表，连接就会涉及表的表达式。在 `A JOIN B JOIN C` 示例中，表的表达式 `A JOIN B` 被连接到 C。这意味着，从理论上讲，首先连接 A 和 B，然后结果被连接到 C。

如果表的表达式包含外连接，则连接的顺序是十分重要的。例如，`A JOIN B LEFT OUTER JOIN C` 被解释为 `(A JOIN B) LEFT OUTER JOIN C`。这意味着表的表达式 `A JOIN B` 被连接到 C。表的表达式 `A JOIN B` 是保留的，表 C 是提供空值的。

有关外连接的详细信息，请参见 「外连接」第 297 页。

有关 Adaptive Server Anywhere 如何执行表的表达式的键连接的详细信息，请参见 「表的表达式的键连接」第 320 页。

有关 Adaptive Server Anywhere 如何执行表的表达式自然连接的详细信息，请参见 「表的表达式的自然连接」第 313 页。

连接兼容数据类型

在连接两个表时，您所比较的列必须具有相同的或兼容的数据类型。

有关连接中数据类型转换的详细信息，请参见 『ASA SQL 参考』> 「使用比较运算符的转换」。

在 delete、 update 和 insert 语句中使用连接

您可以在 DELETE、 UPDATE 和 INSERT 语句以及 SELECT 语句中使用连接。如果将 ANSI_UPDATE_CONSTRAINTS 选项设置为 OFF，则可以更新包含连接的某些游标。对于在 Adaptive Server Anywhere 版本 7 之前的版本中创建的数据库， OFF 是缺省值。对于在版本 7 或更高版本中创建的数据库，缺省值为 ON。

有关详细信息，请参见 『ASA 数据库管理指南』> 「ANSI_UPDATE_CONSTRAINTS 选项 [compatibility]」。

非 ANSI 连接

Adaptive Server Anywhere 支持用于连接的 ISO/ANSI 标准。它还支持以下非标准连接：

- 「Transact-SQL 外连接 （*= 或 =*）」第 301 页
- 「连接中的重复相关名 （星形连接）」第 306 页
- 「键连接」第 316 页
- 「自然连接」第 312 页

您可以使用 REWRITE 函数查看非 ANSI 连接的等效 ANSI 连接。

有关详细信息，请参见 『ASA SQL 参考』> 「REWRITE 函数 [Miscellaneous]」。

显式连接条件 （ON 短语）

您可以单独使用显式连接条件而不用键连接或自然连接，也可以与键连接或自然连接一起使用。通过在紧随该连接之后插入 ON 短语来指定连接条件。该连接条件始终针对它前面的连接。

示例

在以下查询中，第一个 ON 短语用于将 sales_order 连接到 customer。第二个 ON 短语用于将表的表达式 (sales_order JOIN customer) 连接到基表 sales_order_item。

```
SELECT *
 FROM sales_order JOIN customer
     ON sales_order.cust_id = customer.id
   JOIN sales_order_items
     ON sales_order_items.id = sales_order.id
```

可被引用的表

在 ON 短语中引用的表必须是 ON 短语修饰的连接的一部分。例如，以下语句无效：

```
FROM (A KEY JOIN B) JOIN (C JOIN D ON A.x = C.x)
```

造成语句无效的原因在于，连接条件 `A.x = C.x` 引用表 A，而该表不是该连接条件修饰的连接的一部分 （在此例子中为 `C JOIN D`）。

但是，截至 ANSI/ISO 标准 SQL99 和 Adaptive Server Anywhere 7.0，对于该规则有一个例外：如果您在表的表达式之间使用逗号，则连接的 ON 条件可以引用 FROM 子句中在语法上位于该条件前面的表。因此，以下语句是有效的：

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

有关逗号的详细信息，请参见 「逗号」第 294 页。

生成的连接和 ON 短语

如果使用 JOIN 关键字并且没有指定任何连接类型，则键连接是缺省设置，除非您使用 ON 短语。如果您将 ON 短语与未指定类型的 JOIN 一起使用，则缺省不是键连接，也没有其它生成的连接条件。

例如，以下是键连接，因为在使用 JOIN 关键字并且没有 ON 短语时键连接是缺省设置：

```
SELECT *
 FROM A JOIN B
```

以下是表 A 和表 B 之间的连接，并且具有连接条件 `A.x = B.y`。它*不是 键连接*。

```
SELECT *
 FROM A JOIN B ON A.x = B.y
```

如果您指定 KEY JOIN 或 NATURAL JOIN *并且*使用 ON 短语，则最终的连接条件是生成的连接条件与显式连接条件的联合。例如，以下语句具有两个连接条件：一个因键连接生成的连接条件和一个在 ON 短语中显式指定的连接条件。

```
SELECT *
 FROM A KEY JOIN B ON A.x = B.y
```

如果键连接生成的连接条件是 `A.w = B.z`，则以下语句是等效的：

```
SELECT *
 FROM A JOIN B
   ON A.x = B.y
   AND A.w = B.z
```

有关键连接的详细信息，请参见 「键连接」第 316 页。

显式连接条件的类型

大多数连接条件都是基于等同性的，因此称作**等值连接**。例如，

```
SELECT *
 FROM department JOIN employee
     ON department.dept_id = employee.dept_id
```

但是，您并非一定要在连接条件中使用等号 (=)。您可以使用任何搜索条件，例如包含 LIKE、 SOUNDEX、 BETWEEN、 >（大于）和 != （不等于）的搜索条件。

示例

以下示例回答问题：某人订购的哪些产品超出了库存数量？

```
SELECT DISTINCT product.name
 FROM product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
     AND product.quantity > sales_order_items.quantity
```

有关搜索条件的详细信息，请参见 『ASA SQL 参考』> 「搜索条件」。

使用 WHERE 子句用于连接条件

除了使用外连接的情况下，您都可以在 WHERE 子句中指定连接条件来代替 ON 短语。但是，您应该知道，如果查询包含外连接，则 WHERE 子句和 ON 短语两者之间可能存在语义差别。

ON 短语是 FROM 子句的一部分，因此在 WHERE 子句前进行处理。除了外连接，结果通常不会不同。在外连接中，使用 WHERE 子句可能会将连接转换为内连接。

在决定是否将连接条件放置于 ON 短语或 WHERE 子句中时，请牢记以下规则：

- 在指定外连接时，将连接条件放置到 WHERE 子句中可能会将外连接转换为内连接。

  有关 WHERE 子句和外连接的详细信息，请参见 「外连接和连接条件」第 298 页。

- ON 短语中的条件只能引用位于关联的 JOIN 连接表的表达式中的表。而 WHERE 子句中的条件可以引用任何表，即使这些表不是连接的一部分。

- 您不能将 ON 短语与关键字 CROSS JOIN 一起使用，但您始终可以使用 WHERE 子句。

- 在连接条件位于 ON 短语中时，键连接不是缺省设置。但是，如果连接条件被放置于 WHERE 子句中，则键连接可以是缺省设置。

  有关在哪些条件下键连接是缺省设置的详细信息，请参见 「在键连接是缺省值时」第 316 页。

在本文档的一些示例中，连接条件被放置于 ON 短语中。在使用外连接的示例中，这是必需的。在其他一些例子中，这样做是为了更明显地表明它们是连接条件，而不是一般的搜索条件。

交叉连接

两个表的交叉连接生成来自这两个表的各行的所有可能组合。交叉连接也称作矢量积或笛卡儿积。

第一个表的每一行针对第二个表的每一行都出现一次。因此，结果集中的行数是第一个表中的行数与第二个表中的行数相乘、然后减去由于 WHERE 子句中的限制而省略的所有行数而得出的。

您不能将 ON 短语与交叉连接一起使用。但是，您可以将限制放置于 WHERE 子句中。

内部修饰符和外部修饰符不适用于交叉连接

除了 WHERE 子句中存在的额外限制，两个表中的所有行都始终出现在交叉连接的结果集中。因此， INNER、 LEFT OUTER 和 RIGHT OUTER 关键字不适用于交叉连接。

例如，以下语句连接两个表。

```
SELECT *
 FROM A CROSS JOIN B
```

来自此查询的结果集包括 A 中的所有列和 B 中的所有列。 A 的任一行和 B 的任一行的每一组合在结果集中均有一行。如果 A 具有 *n* 行， B 具有 *m* 行，则查询返回 *n* x *m* 行。

逗号

逗号的使用方式类似于连接运算符，但并不完全相同。逗号所创建的矢量积与关键字 CROSS JOIN 所创建的矢量积完全相同。但是，连接关键字创建表的表达式，而逗号则创建表的表达式列表。

在以下两个表的简单内连接中，逗号和关键字 CROSS JOIN 是等效的：

```
Select *
 FROM A CROSS JOIN B CROSS JOIN C
 WHERE A.x = B.y
```

和

```
Select *
 FROM A, B, C
 WHERE A.x = B.y
```

通常，您可以使用逗号来代替关键字 CROSS JOIN。逗号语法大体上等效于交叉连接语法，只在表的表达式中使用逗号的生成的连接条件情况下除外。

有关逗号如何用于生成的连接条件的信息，请参见 「表的表达式的键连接」第 320 页。

在星形连接的语法中，逗号有特殊用途。有关详细信息，请参见 「连接中的重复相关名 （星形连接）」第 306 页。

内连接和外连接

关键字 INNER、 LEFT OUTER、 RIGHT OUTER 和 FULL OUTER 可用于修饰键连接、自然连接和具有 ON 短语的连接。缺省值是 INNER。关键字 OUTER 是可选的。这些修饰符不适用于交叉连接。

内连接

缺省情况下，连接是**内连接**。这意味着，只有在行满足连接条件的情况下，这些行才包括在结果集中。

示例

例如，以下查询的结果集的每一行都包含来自一个 customer 行和一个 sales_order 行的信息，满足键连接条件。如果特定客户没有下任何订单，则没有满足该条件，并且结果集不包含对应于该客户的行。

```
SELECT fname, lname, order_date
 FROM customer KEY INNER JOIN sales_order
 ORDER BY order_date
```

fname	lname	order_date
Hardy	Mums	1/2/00
Aram	Najarian	1/3/00
Tommie	Wooten	1/3/00
Alfredo	Margolis	1/6/00
…	…	…

因为内连接和键连接是缺省值，所以，您可以使用以下 FROM 子句获得同样的结果。

```
FROM customer JOIN sales_order
```

外连接

两个表的左和右**外连接**在一个表中保留所有行，另一个表不满足连接条件时为空值。**左外连接**保留左侧表中的每一行，**右外连接**保留右侧表中的每一行。在**完全外连接**中，保留两个表的所有行。

左外连接或右外连接左右两边表的表达式称作**保留的**和**提供空值的**。在左外连接中，左侧表的表达式是保留的，右侧表的表达式是提供空值的。

有关使用 Transact-SQL 语法创建外连接的信息，请参见 「Transact-SQL 外连接 （*= 或 =*）」第 301 页。

示例

例如，以下语句包括所有客户，无论这些客户是否下了订单。如果某个特定的客户未下任何订单，则结果中对应于订单信息的每一列都包含空值。

```
SELECT lname, order_date, city
 FROM customer LEFT OUTER JOIN sales_order
      ON customer.id = sales_order.cust_id
 WHERE customer.state = 'NY'
 ORDER BY order_date
```

lname	order_date	city
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
…	…	…

您可以按如下说明解释此语句中的外连接。请注意，这只是概念上的说明，并不表示 Adaptive Server Anywhere 实际是如何计算结果集的。

- 为客户下的每一销售订单返回一行。由于为每一个销售订单都返回一行，因此，如果客户下了两个或两个以上的销售订单，就会返回多行。此结果与内连接的结果相同。使用 ON 条件来匹配客户和销售订单行。 WHERE 子句不用于此步骤。

- 为没有下任何销售订单的每一客户包含一行。这确保包括客户表中的每一行。对于所有这些行，都会用空值填充 sales_order 中的列。因为使用了关键字 OUTER，所以加上这些行，如果是内连接就不会有这些行。 ON 条件和 WHERE 子句均不用于此步骤。
- 使用 WHERE 子句，排除客户不居住在纽约的每一行。

外连接和连接条件

外连接的常见错误通常出在连接条件的放置上。在大多数情况下，如果您在 WHERE 子句中对提供空值的表施加限制，则连接等效于内连接。

原因在于：在搜索条件的任何输入都为空时，大多数搜索条件无法计算为 TRUE （真）。对提供空值的表的 WHERE 子句的限制将值与空值进行比较，导致从结果集中删除该行。保留的表中的行不再保留，因此连接是内连接。

例外的是有些比较在输入都为空时的值为真。这些包括 IS NULL、 IS UNKNOWN、 IS FALSE、 IS NOT TRUE 和涉及 ISNULL 或 COALESCE 的表达式。

示例

例如，以下语句计算左外连接。

```
SELECT *
 FROM customer KEY LEFT OUTER JOIN sales_order
     ON sales_order.order_date < '2000-01-03'
```

作为比较，以下语句创建内连接。

```
SELECT lname, order_date
 FROM customer KEY LEFT OUTER JOIN sales_order
     WHERE sales_order.order_date < '2000-01-03'
```

这两个语句中的第一个语句可按如下解释：首先，将 customer 表左外连接到 sales_order 表。结果集包括客户表中的每一行。对于在 2000 年 1 月 3 日之前没有任何订单的那些客户，用空值填充销售订单字段。

在第二个语句中，首先左外连接 customer 和 sales_order。结果集包括客户表中的每一行。对于没有任何订单的那些客户，用空值填充销售订单字段。接下来，通过只选择在其中客户自 2000 年 1 月 3 日以后已下了

订单的那些行，应用 WHERE 条件。对于没有下任何订单的那些客户，这些值是空值。将任何值与空值进行比较将计算为 UNKNOWN （未知）。因此，这些行被删除，该语句简化为内连接。

有关搜索条件的详细信息，请参见 『ASA SQL 参考』> 「搜索条件」。

理解复杂外连接

在查询包括使用外连接表的表达式时，连接的顺序是十分重要的。例如， `A JOIN B LEFT OUTER JOIN C` 被解释为 `(A JOIN B) LEFT OUTER JOIN C`。这意味着表的表达式 `(A JOIN B)` 被连接到 C。表的表达式 `(A JOIN B)` 是保留的，表 C 是提供空值的。

考虑以下语句，在该语句中， A、 B 和 C 是表：

```
SELECT *
 FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C
```

为了理解此语句，首先要记住， Adaptive Server Anywhere 是按从左到右的顺序对语句求值的，并且按以下所示添加括号。这导致

```
SELECT *
 FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C
```

接下来，您可能想要将右外连接转换为左外连接，以使这两个连接的类同。要做到这一点，只需反转右外连接中各表的位置，得到：

```
SELECT *
 FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B)
```

A 是保留的表， B 是用于嵌套的外连接的提供空值的表。 C 是用于第一个外连接的保留的表。

您可按如下所示解释该连接：

- 将 A 连接到 B，保留 A 中的所有行。
- 接下来，将 C 与 A 和 B 的连接的结果相连接，保留 C 中的所有行。

该连接不具有 ON 短语，因此在缺省情况下是键连接。 Adaptive Server Anywhere 为此类型的连接生成连接条件的方法在 「不包含逗号的表的表达式的键连接」第 320 页中予以说明。

此外，外连接的连接条件必须只包括以前已在 FROM 子句中引用的表。此限制依据 ANSI/ISO 标准，并且强制实施以避免出现多义性。例如，以下两个语句在语义上是不正确的，因为在引用表本身之前在连接条件中引用了 C。

```
SELECT *
 FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C
```

和

```
SELECT *
 FROM A LEFT OUTER JOIN B ON A.x = C.x, C
```

视图和派生表的外连接

还可为视图和派生表指定外连接。

语句

```
SELECT *
 FROM V LEFT OUTER JOIN A ON (V.x = A.x)
```

可按如下所述解释：

- 计算视图 V。
- 通过保留来自 V 的所有行并且使用连接条件 `V.x = A.x`，将计算出的视图 V 中的所有行与 A 相连接。

示例

以下示例定义名为 V 的视图，该视图返回薪水超过 $60 000 的女雇员的雇员 ID 和部门名称。

```
CREATE VIEW V AS
 SELECT employee.emp_id, dept_name
   FROM employee JOIN department
     ON employee.dept_id = department.dept_id
   WHERE sex = 'F' and salary > 60000
```

接下来，使用此视图添加这些女士就职的部门以及销售区域的列表。视图 V 会保留下来， sales_order 提供空值。

```
SELECT DISTINCT V.emp_id, region, V.dept_name
   FROM V LEFT OUTER JOIN sales_order
     ON V.emp_id = sales_order.sales_rep
```

emp_id	region	dept_name
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
…	…	…

Transact-SQL 外连接 （*= 或 =*）

与 ANSI/ISO SQL 标准相符合， Adaptive Server Anywhere 支持 LEFT OUTER、 RIGHT OUTER 和 FULL OUTER 关键字。为与早于版本 12 的 Adaptive Server Enterprise 版本兼容， Adaptive Server Anywhere 还支持这些关键字的 Transact-SQL 副本：*= 和 =*。但是，有一些与 Transact-SQL 语义有关的限制和可能存在的问题。

有关 Transact-SQL 外连接的详细论述，请参见白皮书 *Transact-SQL 外连接的语义和兼容性*, 您可以在以下网址获得该白皮书 http://www.ianywhere.com/whitepapers/tsql.html。

警告：
在创建外连接时，*不要* 将 *= 语法与 ON 短语语法混合使用。这同样适用于在查询中引用的视图。

在 Transact-SQL 方言中，您通过在 FROM 子句中提供表的逗号分隔的列表，并且在 WHERE 子句中使用特殊运算符 *= 或 =* 来创建外连接。在早于版本 12 的 Adaptive Server Enterprise 中，连接条件必须出现在 WHERE 子句中 （不支持 ON 短语）。

示例

例如，以下左外连接列出所有客户并查找这些客户的订单日期 （如果有）：

```
SELECT fname, lname, order_date
 FROM customer, sales_order
 WHERE customer.id *= sales_order.cust_id
 ORDER BY order_date
```

该语句等效于以下语句，后者使用了 ANSI/ISO 语法：

```
SELECT fname, lname, order_date
 FROM customer LEFT OUTER JOIN sales_order
 ON customer.id = sales_order.cust_id
 ORDER BY order_date
```

Transact-SQL 外连接限制

对于 Transact-SQL 外连接有若干限制：

- 如果您指定了某一外连接并对来自该外连接的提供空值的表的某一列指定了限定，则结果可能是并非您预期的结果。查询中的限定不从结果集中排除行，而是影响结果集的这些行中出现的值。对于不满足该限定的行，在提供空值的表中显示空值。
- 您不能在单个查询中混合使用 ANSI/ISO SQL 语法和 Transact-SQL 外连接语法。如果视图是使用用于某一外连接的一个方言定义的，则必须使用同一方言来用于对该视图的任何外连接查询。
- 提供空值的表不能同时涉及 Transact-SQL 外连接和常规连接，或同时涉及两个外连接。例如，不允许以下 WHERE 子句，因为表 S 违反上述限制。

  ```
  WHERE R.x *= S.x
   AND S.y = T.y
  ```

  如果您无法重写查询以避免在外连接和常规连接子句中都使用一个表，则必须将语句划分为两个单独的查询，或者只使用 ANSI/ISO SQL 语法。
- 如果某一连接条件涉及外连接的提供空值的表，则您不能使用包含这样的连接条件的子查询。例如，以下 WHERE 子句是不允许使用的：

```
WHERE R.x *= S.y
 AND EXISTS ( SELECT *
              FROM T
              WHERE T.x = S.x )
```

将视图与 Transact-SQL 外连接一起使用

如果您用外连接定义一个视图，然后查询该视图并且对来自该外连接的提供空值的表的列加以限定，则结果可能是并非您预期的结果。该查询返回来自提供空值的表的所有行。不符合该限定的行在这些行的适当列中显示空值。

以下规则确定您通过包含外连接的视图对列可以执行哪些类型的更新：

- 对外连接视图不允许使用 INSERT 和 DELETE 语句。
- 对外连接视图允许使用 UPDATE 语句。如果该视图定义了 WITH CHECK 选项，如果任何受影响的列出现在 WHERE 子句的表达式中，且该表达式包括来自多个表的列，更新会失败。

空值是如何影响 Transact-SQL 连接的

表中或视图中所连接的空值永远不在 Transact-SQL 外连接中彼此匹配。如果将空值与其它任何空值进行比较，则结果为 FALSE （假）。

专用连接

本节描述如何创建某些专用连接，例如自连接、星形连接和使用派生表的连接。

自连接

在**自连接**中，表通过引用使用不同相关名的同一表，与其自身相连接。

示例 1

以下自连接生成雇员对列表。每一雇员姓名与各雇员姓名彼此结合在一起显示。

```
SELECT a.emp_fname, a.emp_lname,
        b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
```

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
…	…	…	…

因为 employee 表具有 75 行，所以此连接包含 75 x 75 = 5 625 行。它还包括将每一雇员与他们自己一起列出的行。例如，它包括以下行

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney

如果您想要排除包含同一姓名两次的行，则添加一个连接条件，指示雇员 ID 应互不相同。

```
SELECT a.emp_fname, a.emp_lname,
         b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
 WHERE a.emp_id != b.emp_id
```

没有了这些重复行，该连接包含 75 x 74 = 5 550 行。

该新连接包含将每一雇员与每一其他雇员配对的行，但因为每一姓名对可以按两个可能的顺序出现，所以每一对出现两次。例如，以上连接的结果包含以下两行。

emp_fname	emp_lname	emp_fname	emp_lname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

如果姓名的顺序不重要，则可以生成包含 (75 x 74)/2 = 2 775 个唯一对的列表。

```
SELECT a.emp_fname, a.emp_lname,
         b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
 WHERE a.emp_id < b.emp_id
```

此语句通过只选择其雇员 a 的 emp_id 小于雇员 b 的 emp_id 的那些行，消除了重复行。

示例 2

以下自连接使用相关名 report 和 manager 来区分 employee 表的两个实例，并且创建雇员及其经理的列表。

```
SELECT report.emp_fname, report.emp_lname,
     manager.emp_fname, manager.emp_lname
 FROM employee AS report JOIN employee AS manager
     ON (report.manager_id = manager.emp_id)
 ORDER BY report.emp_lname, report.emp_fname
```

此语句生成以下部分显示的结果。雇员姓名出现在左侧的两列中，其经理的姓名出现在右侧。

emp_fname	emp_lname	emp_fname	emp_lname
Alex	Ahmed	Scott	Evans

emp_fname	emp_lname	emp_fname	emp_lname
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
…	…	…	…

连接中的重复相关名 （星形连接）

使用重复表名称的原因是为了创建**星形连接**。在星形连接中，一个表或视图与若干其它表或视图相连接。

若要创建星形连接，应在 FROM 子句中多次使用同一表名称、视图名称或相关名。这是对 ANSI/ISO SQL 标准的扩展。能够使用重复名并不添加任何附加功能，但这可以使某些查询写起来更加容易。

为使语法有意义，重复名必须位于不同的连接中。如果在同一连接中使用两次表名称或视图名称，则忽略第二个实例。例如， `FROM A,A` 和 `FROM A CROSS JOIN A` 都被解释为 `FROM A`。

以下示例 （在该示例中， A、 B 和 C 都是表）在 Adaptive Server Anywhere 中是有效的。在此示例中，表 A 的同一实例同时连接到 B 和 C。请注意，在星形连接中需要使用逗号分隔这些连接。在星形连接中使用逗号有其特定的星形连接的语法。

```
SELECT *
 FROM A LEFT OUTER JOIN B ON A.x = B.x,
      A LEFT OUTER JOIN C ON A.y = C.y
```

下一个示例是等效的。

```
SELECT *
 FROM A LEFT OUTER JOIN B ON A.x = B.x,
      C RIGHT OUTER JOIN A ON A.y = C.y
```

这两个示例都等效于以下 ANSI/ISO 语法。（括号是可选的。

```
SELECT *
 FROM (A LEFT OUTER JOIN B ON A.x = B.x)
 LEFT OUTER JOIN C ON A.y = C.y
```

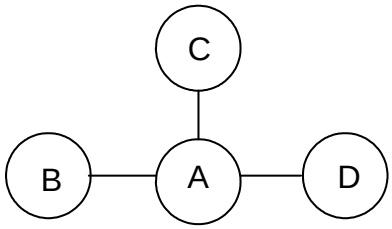
在下一示例中，表 A 被连接到三个表：B、 C 和 D。

```
SELECT *
 FROM A JOIN B ON A.x = B.x,
      A JOIN C ON A.y = C.y,
      A JOIN D ON A.w = D.w
```

这等效于以下标准 ANSI/ISO 语法。（括号是可选的。

```
SELECT *
 FROM ((A JOIN B ON A.x = B.x)
 JOIN C ON A.y = C.y)
 JOIN D ON A.w = D.w
```

使用复杂连接，可有助于绘制图示。前面的示例可以通过以下图示描述，该图示阐释通过表 A 连接的表 B、 C 和 D。



注意

只有将 EXTENDED_JOIN_SYNTAX 选项设置为 ON 后 （缺省设置），您才可以使用重复表名称。

有关详细信息，请参见 『ASA 数据库管理指南』> 「EXTENDED_JOIN_SYNTAX 选项 [database]」。

示例 1

创建给 Rollin Overbey 下订单的客户名称的列表。请注意 FROM 子句中的那些表，其中有一个表 （employee 表）中的任何列都不在结果中，所连接的任何列 - 例如 customer.id 或 employee.id- 都不会出现在结果中。不过，只有在 FROM 子句中使用 employee 表才能实现该连接。

```
SELECT customer.fname, customer.lname,
       sales_order.order_date
 FROM     sales_order KEY JOIN customer,
          sales_order KEY JOIN employee
 WHERE employee.emp_fname = 'Rollin'
   AND employee.emp_lname = 'Overbey'
 ORDER BY sales_order.order_date
```

fname	lname	order_date
Tommie	Wooten	1/3/00
Michael	Agliori	1/8/00
Salton	Pepper	1/17/00
Tommie	Wooten	1/23/00
…	…	…

以下是采用标准 ANSI/ISO 语法的等效语句：

```
SELECT customer.fname, customer.lname,
   sales_order.order_date
 FROM sales_order JOIN customer
   ON sales_order.cust_id = customer.id
 JOIN employee
   ON sales_order.sales_rep = employee.emp_id
 WHERE employee.emp_fname = 'Rollin'
   AND employee.emp_lname = 'Overbey'
 ORDER BY sales_order.order_date
```

示例 2

此示例回答如下问题：每一客户各产品订购多少，谁是获得该订单的销售人员的经理？

要回答上述问题，应从列出您需要检索的信息开始。在此例子中，需要检索的信息是产品、数量、客户名称和经理姓名。接下来，列出保存这些信息的表。它们是 product、 sales_order_items、 customer 和 employee。查看示例数据库的结构时 （请参见 「示例数据库模式」第 282 页），您将注意到，这些表全部通过 sales_order 表进行关联。您可以为 sales_order 表创建星形连接，以检索其它表中的信息。

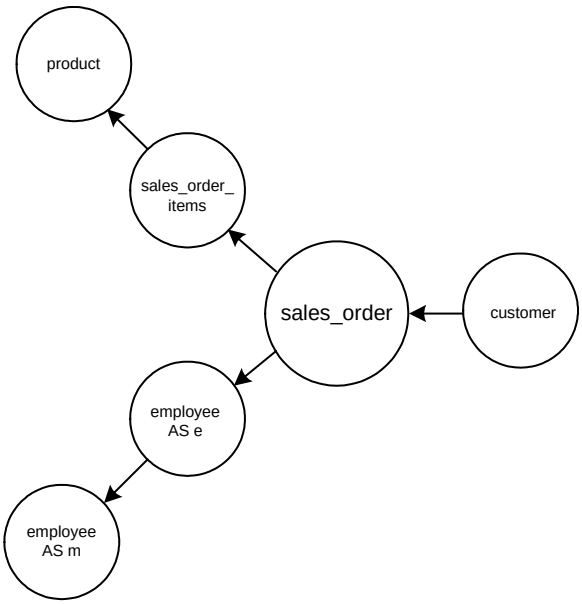
此外，您需要创建自连接，以获取经理的姓名，因为 employee 表包含经理的 ID 号及所有雇员的姓名，但没有给出只列有经理姓名的列。有关详细信息，请参见 「自连接」第 304 页。

以下语句围绕 sales_order 表创建星形连接。这些连接全都是外连接，因此结果集将包括所有客户。有些客户没有下订单，因此对应这些客户的其它值为空。结果集中的列是客户、产品、订购的数量以及销售人员的经理的姓名。

```
SELECT customer.fname, product.name,
   SUM(sales_order_items.quantity), m.emp_fname
 FROM    sales_order
             KEY RIGHT OUTER JOIN customer,
          sales_order
             KEY LEFT OUTER JOIN sales_order_items
             KEY LEFT OUTER JOIN product,
          sales_order
             KEY LEFT OUTER JOIN employee AS e
             LEFT OUTER JOIN employee AS m
             ON (e.manager_id = m.emp_id)
 WHERE customer.state = 'CA'
 GROUP BY customer.fname, product.name, m.emp_fname
 ORDER BY SUM(sales_order_items.quantity) DESC,
customer.fname
```

fname	name	SUM(sales_order_items.quantity)	emp_fname
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
…	…	…	…

以下是此星形连接中各表的图示。箭头指示这些外连接的方向（左或右）。正如您所看到的，整个客户列表是贯穿所有连接来维护的。



以下标准 ANSI/ISO 语法等效于示例 2 中的星形连接。

```
SELECT customer.fname, product.name,
   SUM(sales_order_items.quantity), m.emp_fname
 FROM sales_order LEFT OUTER JOIN sales_order_items
     ON sales_order.id = sales_order_items.id
   LEFT OUTER JOIN product
     ON sales_order_items.prod_id = product.id
   LEFT OUTER JOIN employee as e
     ON sales_order.sales_rep = e.emp_id
   LEFT OUTER JOIN employee as m
     ON e.manager_id = m.emp_id
   RIGHT OUTER JOIN customer
     ON sales_order.cust_id = customer.id
 WHERE customer.state = 'CA'
 GROUP BY customer.fname, product.name, m.emp_fname
 ORDER BY SUM(sales_order_items.quantity) DESC,
customer.fname
```

涉及派生表的连接

派生表使您能够在 FROM 子句中嵌套查询。使用派生表，您可以对组进行分组或构造与组之间的连接而无需创建视图。

在以下示例中，内部 SELECT 语句（用括号括起来）创建一个派生表，该派生表是按客户 ID 值进行分组的。外部 SELECT 语句为此表指派相关名 sales_order_counts，并使用连接条件将它与 customer 表相连接。

```
SELECT lname, fname, number_of_orders
 FROM customer JOIN
     (    SELECT cust_id, count(*)
         FROM sales_order
         GROUP BY cust_id    )
     AS sales_order_counts (cust_id, number_of_orders)
     ON (customer.id = sales_order_counts.cust_id)
 WHERE number_of_orders > 3
```

结果是下了三个以上订单的那些客户的名称的表，并且包括每一客户所下订单的数目。

有关派生表的键连接的说明，请参见 「视图和派生表的键连接」第 326 页。

有关派生表的自然连接的说明，请参见 「视图和派生表的自然连接」第 315 页。

有关派生表的外连接的说明，请参见 「视图和派生表的外连接」第 300 页。

自然连接

在您指定自然连接时， Adaptive Server Anywhere 基于具有相同名称的列生成连接条件。因此，要使用基表的自然连接，必须至少有一对具有相同名称的列，每个表一列。如果没有公用列名称，将发出错误消息。

如果表 A 和表 B 具有一个共同的列名称，并且该列称作 x，则

```
SELECT *
 FROM A NATURAL JOIN B
```

等效于以下语句：

```
SELECT *
 FROM A JOIN B
  ON A.x = B.x
```

如果表 A 和表 B 具有两个共同的列名称，并且它们分别称作 a 和 b，则 `A NATURAL JOIN B` 等效于以下语句：

```
A JOIN B
  ON A.a = B.a
  AND A.b = B.b
```

示例

例如，您可以使用自然连接来连接 employee 表和 department 表，因为这两个表具有一个共同的列名称：dept_id 列。

```
SELECT emp_fname, emp_lname, dept_name
 FROM employee NATURAL JOIN department
 ORDER BY dept_name, emp_lname, emp_fname
```

emp_fname	emp_lname	dept_name
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
…	…	…

以下语句是等效的。该语句显式指定已在前面的示例中生成的连接条件。

```
SELECT emp_fname, emp_lname, dept_name
 FROM employee JOIN department
   ON (employee.dept_id = department.dept_id)
 ORDER BY dept_name, emp_lname, emp_fname
```

具有 ON 短语的自然连接

在您指定了 NATURAL JOIN *并*将连接条件放置于 ON 短语后，结果是两个连接条件的联合。

例如，以下两个查询是等效的。在第一个查询中， Adaptive Server Anywhere 生成连接条件 `employee.dept_id = department.dept_id`。该查询还包含显式连接条件。

```
SELECT emp_fname, emp_lname, dept_name
 FROM employee NATURAL JOIN department
   ON employee.manager_id = department.dept_head_id
```

下一个查询是等效的。在该查询中，前一查询中生成的自然连接条件是在 ON 短语中指定的。

```
SELECT emp_fname, emp_lname, dept_name
 FROM employee JOIN department
   ON employee.manager_id = department.dept_head_id
    AND employee.dept_id = department.dept_id
```

表的表达式的自然连接

当自然连接至少有一侧具有多表的表达式时，Adaptive Server Anywhere 通过为连接运算符的每一侧比较列集合，并且查找具有相同名称的列，生成连接条件。

例如，在以下语句中

```
SELECT *
 FROM (A JOIN B) NATURAL JOIN (C JOIN D)
```

有两个表的表达式。表的表达式 A JOIN B 中的列名称与表的表达式 C JOIN D 中的列名称进行比较，并且为每一明确的匹配列名称对生成连接条件。*明确的匹配列对*意味着列名称在两个表的表达式中均出现，但不在同一表的表达式中出现两次。

如果有不明确的列名称对，将发出错误消息。但是，列名称可以在同一表的表达式中出现两次，只要它还不匹配另一个表的表达式中的列的名称。

列表的自然连接

在表的表达式列表位于自然连接的至少一侧时，为列表中的每一表的表达式生成单独的连接条件。

考虑以下表：

- 表 A 包含名为 a、 b 和 c 的列
- 表 B 包含名为 a 和 d 的列
- 表 C 包含名为 d 和 c 的列

在此例子中，连接 (A,B) NATURAL JOIN C 导致 Adaptive Server Anywhere 生成两个连接条件：

```
ON A.c = C.c
  AND B.d = C.d
```

如果对于 A-C 或 B-C 没有公用列名称，将发出错误消息。

如果表 C 包含列 a、 d 和 c，则连接 (A,B) NATURAL JOIN C 是无效的。原因在于：列 a 出现在所有三个表中，因此连接是不明确的。

示例

以下示例回答问题：对于每一销售，提供与所销售的产品以及销售它的人员有关的信息。

```
SELECT *
 FROM (employee KEY JOIN sales_order)
   NATURAL JOIN (sales_order_items KEY JOIN product)
```

这等效于

```
SELECT *
 FROM (employee KEY JOIN sales_order)
   JOIN (sales_order_items KEY JOIN product)
     ON sales_order.id = sales_order_items.id
```

视图和派生表的自然连接

对 ANSI/ISO SQL 标准的扩展是：您可以在自然连接的两边指定视图或派生表。在以下语句中，

```
SELECT *
 FROM View1 NATURAL JOIN View2
```

将 View1 中的列与 View2 中的列进行比较。例如，如果发现名为 emp_id 的列在两个视图中均出现，并且没有其它列具有相同的名称，则生成的连接条件是 `(View1.emp_id = View2.emp_id)`。

示例

以下示例说明在自然连接中使用的视图可以包括表达式，而不只是包括列，并且采用与自然连接相同的方式处理它们。首先，创建具有名为 x 的列的视图 V，如下所示：

```
CREATE VIEW V(x) AS
 SELECT R.y + 1
 FROM R
```

接下来，创建该视图与派生表的自然连接。该派生表具有相关名 T 及名为 x 的列。

```
SELECT *
 FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x)
```

此连接等效于以下语句：

```
SELECT *
 FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x)
```

键连接

在您指定键连接时， Adaptive Server Anywhere 基于数据库中的外键关系生成连接条件。若要使用键连接，表和表之间必须有外键关系，否则将发出错误消息。

键连接是 Sybase 对 ANSI/ISO SQL 标准的扩展。键连接不提供任何更强大的功能，但它可以使某些查询写起来更容易。

在键连接是缺省值时

在满足以下所有条件后，键连接在 Adaptive Server Anywhere 中是缺省值：

- 使用关键字 JOIN。
- *未*指定关键字 CROSS、 NATURAL 或 KEY。
- 没有 ON 短语。

示例

例如，以下查询是一个简单的键连接，它基于数据库中的外键关系连接表 product 和 sales_order_items。

```
SELECT *
 FROM product KEY JOIN sales_order_items
```

下一个查询是等效的。它省去了 KEY 一词，但缺省情况下，没有 ON 短语的 JOIN 是 KEY JOIN。

```
SELECT *
 FROM product JOIN sales_order_items
```

下一个查询也是等效的，因为在 ON 短语中指定的连接条件正好与 Adaptive Server Anywhere 在示例数据库中基于这些表的外键关系为这些表生成的连接条件相同。

```
SELECT *
 FROM product JOIN sales_order_items
 ON sales_order_items.prod_id = product.id
```

具有 ON 短语的键连接

在您指定 KEY JOIN *并*将连接条件放置于 ON 短语中时，结果是两个连接条件的联合。例如，

```
SELECT *
 FROM A KEY JOIN B
 ON A.x = B.y
```

如果由 A 和 B 的键连接生成的连接条件是 A.w = B.z，则该查询等效于

```
SELECT *
 FROM A JOIN B
 ON A.x = B.y AND A.w = B.z
```

在有多个外键关系时的键连接

在 Adaptive Server Anywhere 尝试基于外键关系生成连接条件时，它有时会找到多个关系。在这些情况下， Adaptive Server Anywhere 通过将外键的角色名与该外键引用的主键表的相关名相匹配，确定要使用的外键关系。

以下几节描述 Adaptive Server Anywhere 如何为键连接生成连接条件。这些信息是根据 「描述键连接的操作的规则」第 328 页中的内容汇总的。

相关名和角色名

相关名是在查询的 FROM 子句中使用的表或视图的名称 - 它的初始名称或者在 FROM 子句中定义的别名。

角色名是外键的名称。对于给定的外 （子）表，角色名必须是唯一的。

如果您没有为外键指定角色名，则按如下所示指派角色名：

- 如果没有任何外键的名称与主表名称相同，则将该主表名称指派为角色名。
- 如果主表名称已由其它外键使用，则角色名将由该主表名称后接用零填充的三位数字组合而成 （对于外表是唯一的）。

如果您不知道外键的角色名，则可以通过展开左窗格中的数据库容器，在 Sybase Central 中找到它。在左窗格中选择表，然后在右窗格中单击 [外键] 选项卡。该表的外键列表会显示在右窗格中。

有关包括示例数据库中所有外键的角色名的图示，请参见 「示例数据库模式」第 282 页。

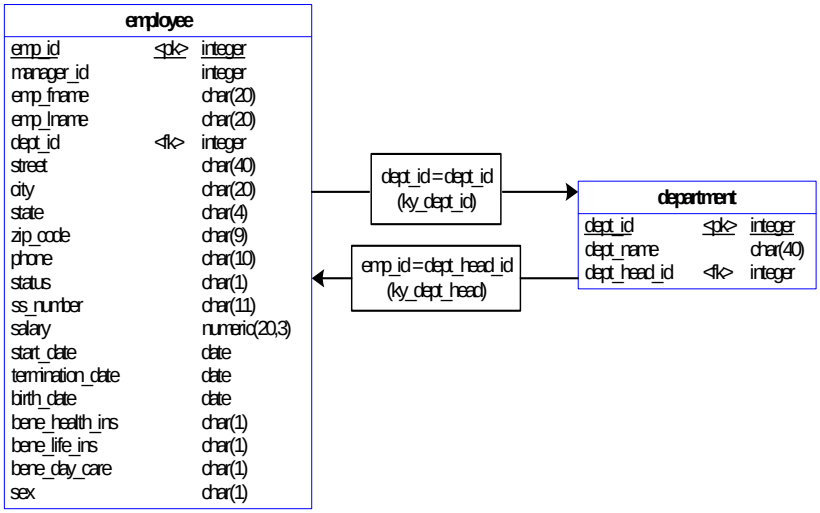
生成连接条件

Adaptive Server Anywhere 查找具有与主键表的相关名相同的角色名的外键。

- 如果恰好具有一个与连接中的表同名的外键，则 Adaptive Server Anywhere 使用它来生成连接条件。
- 如果具有多个与表同名的外键，则连接是不明确的，并且将发出错误消息。
- 如果没有与表同名的外键，则 Adaptive Server Anywhere 将查找任何外键关系，即使这些名称不匹配。如果有多个外键关系，则连接是不明确的，并且将发出错误消息。

示例 1

在示例数据库中，在 employee 表和 department 表之间定义了两种外键关系：employee 表中的外键 ky_dept_id 引用 department 表； department 表中的外键 ky_dept_head 引用 employee 表。



以下查询是不明确的，因为该查询具有两个外键关系并且这两个外键关系都不具有与主键表名称相同的角色名。因此，尝试此查询将导致语法错误 SQLE_AMBIGUOUS_JOIN (-147)。

```
SELECT employee.emp_lname, department.dept_name
 FROM employee KEY JOIN department
```

示例 2

此查询通过为 department 表指定相关名 ky_dept_id 修改示例 1 中的查询。现在，外键 ky_dept_id 与它引用的表具有相同的名称，所以使用它来定义连接条件。结果中包括所有雇员的姓氏以及他们就职的部门。

```
SELECT employee.emp_lname, ky_dept_id.dept_name
 FROM employee KEY JOIN department AS ky_dept_id
```

以下查询是等效的。在此示例中，不必为 department 表创建别名。在此查询的 ON 短语中指定与上述语句生成的相同的连接条件：

```
SELECT employee.emp_lname, department.dept_name
 FROM employee JOIN department
     ON department.dept_id = employee.dept_id
```

示例 3

如果想要列出职位是部门领导的所有雇员，则应使用外键 ky_dept_head，并且应按如下所示重写示例 1。此查询通过为主键表 employee 指定相关名 ky_dept_head，强制使用外键 ky_dept_head。

```
SELECT ky_dept_head.emp_lname, department.dept_name
 FROM employee AS ky_dept_head KEY JOIN department
```

以下查询是等效的。在此查询的 ON 短语中指定上述语句生成的连接条件：

```
SELECT employee.emp_lname, department.dept_name
 FROM employee JOIN department
     ON department.dept_head_id = employee.emp_id
```

示例 4

如果外键角色名等同于主键表名，则不需要相关名。例如，我们可以为 employee 表定义外键 department：

```
ALTER TABLE employee ADD FOREIGN KEY department
(dept_id) REFERENCES department (dept_id)
```

现在，此外键关系是在这两个表之间指定 KEY JOIN 时的缺省连接条件。如果定义了外键 department，则以下查询等效于示例 3。

```
SELECT employee.emp_lname, department.dept_name
 FROM employee KEY JOIN department
```

注意

如果您在 Interactive SQL 中尝试此示例，则应使用以下语句撤消对示例数据库的更改：

```
ALTER TABLE employee DROP FOREIGN KEY department
```

表的表达式的键连接

Adaptive Server Anywhere 通过在语句中检查每一对表的外键关系，为表的表达式的键连接生成连接条件。

以下示例连接四对表。

```
SELECT *
 FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D)
```

这四个表对是 A-C、 A-D、 B-C 和 B-D。 Adaptive Server Anywhere 考虑每一对中的关系，然后从整体上为该表的表达式创建生成的连接条件。 Adaptive Server Anywhere 进行此操作的方式取决于表的表达式是否使用逗号。因此，以下两个示例中的生成的连接条件是不同的。 `A JOIN B` 是*不包含逗号的表的表达式* `(A,B)` 是*表的表达式列表*

```
SELECT *
 FROM (A JOIN B) KEY JOIN C
```

在语义上不同于

```
SELECT *
 FROM (A,B) KEY JOIN C
```

在以下各节中说明连接行为的两种类型：

- 「不包含逗号的表的表达式的键连接」第 320 页
- 「表的表达式列表的键连接」第 322 页

不包含逗号的表的表达式的键连接

当所连接的两个表的表达式都不包含逗号时，Adaptive Server Anywhere 检查该语句中表对的外键关系，并且生成*单个*连接条件。

例如，以下连接具有两个表对：A-C 和 B-C。

```
(A NATURAL JOIN B) KEY JOIN C
```

Adaptive Server Anywhere 通过查看表对 A-C 和 B-C 中的外键关系生成一个用于将 C 与 (A NATURAL JOIN B) 连接的连接条件。它根据存在多个外键关系时确定键连接的规则为这两个对生成一个连接条件：

- 首先，它在 A-C 和 B-C 中查找一个外键，该单个外键所具有的角色名与其所引用的某一主键表的相关名相同。如果恰好有一个外键满足此条件，则连接条件使用此外键。如果有多个外键的角色名与表的相关名相同，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键的名称与表的相关名相同，则 Adaptive Server Anywhere 查找这些表间的任何外键关系。如果有一个外键关系，则连接条件使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键关系，则发出错误消息。

有关详细信息，请参见 「在有多个外键关系时的键连接」第 317 页。

示例

以下查询查找职位为销售代表的所有雇员及其所在部门。

```
SELECT employee.emp_lname, ky_dept_id.dept_name
 FROM (employee KEY JOIN department as ky_dept_id)
   KEY JOIN sales_order
```

您可以按如下所示解释该查询。

- Adaptive Server Anywhere 考虑使用表的表达式 (employee KEY JOIN department as ky_dept_id) 并基于外键 ky_dept_id 生成连接条件 employee.dept_id = ky_dept_id.dept_id。
- 然后，Adaptive Server Anywhere 考虑表对 employee-sales_order 和 ky_dept_id-sales_order。请注意，只有一个外键可以存在于表 sales_order 和 employee *以及* sales_order 和 ky_dept_id 之间，否则连接不明确。出现这种情况时，表 sales_order 和 employee (ky_so_employee_id) 之间正好有一种外键关系，而 sales_order 和 ky_dept_id 之间没有外键关系。因此，生成的连接条件是 sales_order.emp_id = employee.sales_rep。

以下查询因此等效于前一查询：

```
SELECT employee.emp_lname, department.dept_name
 FROM (employee JOIN department
     ON ( employee.dept_id = department.dept_id ) )
 JOIN sales_order
     ON (employee.emp_id = sales_order.sales_rep)
```

表的表达式列表的键连接

为了给两个表的表达式列表的键连接生成连接条件， Adaptive Server Anywhere 在语句中检查表对，并且为*每一对*生成连接条件。最终的连接条件是用于每一对的连接条件的联合。每一对之间必须是外键关系。

以下示例连接两个表对：A-C 和 B-C。

```
SELECT *
 FROM (A,B) KEY JOIN C
```

Adaptive Server Anywhere 通过为两个对 A-C 和 B-C 的每一对生成连接条件，生成用于将 C 与 (A,B) 连接的连接条件。它根据具有多个外键关系时的键连接规则生成连接条件：

- 对于每一对， Adaptive Server Anywhere 查找与主键表的相关名具有相同角色名的外键。如果恰好有一个外键满足此条件，则连接条件使用此外键。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每一对，如果没有与表的相关名同名的外键，则 Adaptive Server Anywhere 将查找这些表间的任何外键关系。如果有一个外键关系，则连接条件使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每一对，如果没有任何外键关系，则发出错误消息。
- 如果 Adaptive Server Anywhere 能够为每一对确定恰好一个连接条件，则它使用 AND 组合这些连接条件。

有关详细信息，请参见 「在有多个外键关系时的键连接」第 317 页。

示例

以下查询返回向特定区域售出至少一个订单的所有销售人员的姓名。

```
SELECT DISTINCT employee.emp_lname,
   ky_dept_id.dept_name, sales_order.region
 FROM (sales_order, department AS ky_dept_id)
    KEY JOIN employee
```

emp_lname	dept_name	region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
…	…	…

此查询处理两个表对：sales_order 和 employee ； department AS ky_dept_id 和 employee。

对于 sales_order 和 employee 对，没有任何外键与其中一表具有相同的角色名。但是，具有与这两个表相关的外键 (ky_so_employee_id)。该外键是与这两个表关联的唯一外键，因此使用它，并且导致生成的连接条件 (`employee.emp_id = sales_order.sales_rep`)。

对于 department AS ky_dept_id 和 employee 对，有一个外键与主键表具有相同的角色名。该外键是 ky_dept_id，并且它与在查询中为 department 表指定的相关名匹配。没有任何其它外键与主键表的相关名同名，因此，使用 ky_dept_id 为该表对形成连接条件。生成的连接条件是 (`employee.dept_id = ky_dept_id.dept_id`)。请注意，具有与这两个表关联的其它外键，但因为外键所具有的名称不同于这两个表中的任何一个表的名称，所以不考虑它。

最终连接条件将为每一表对生成的连接条件添加到一起。因此，以下查询是等效的：

```
SELECT DISTINCT employee.emp_lname,
department.dept_name, sales_order.region
 FROM ( sales_order, department )
    JOIN employee
    ON employee.emp_id = sales_order.sales_rep
    AND employee.dept_id = department.dept_id
```

表的表达式列表和不包含逗号的表的表达式的键连接

当通过具有不包含逗号的表的表达式和键连接来连接表的表达式列表时， Adaptive Server Anywhere 为表的表达式列表中的每一表生成连接条件。

例如，以下语句是具有不包含逗号的表的表达式和表的表达式列表的键连接。此示例为表 A 和表的表达式 `C NATURAL JOIN D` 生成连接条件，并且为表 B 和表的表达式 `C NATURAL JOIN D` 生成连接条件。

```
SELECT *
 FROM (A,B) KEY JOIN (C NATURAL JOIN D)
```

`(A,B)` 是表的表达式列表， `C NATURAL JOIN D` 是表的表达式。因此，Adaptive Server Anywhere 必须生成两个连接条件：它为 A-C 和 A-D 对生成一个连接条件，为 B-C 和 B-D 对生成第二个连接条件。它根据有多个外键关系时的键连接规则生成连接条件：

- 对于每组表对， Adaptive Server Anywhere 查找与某一主键表的相关名具有相同角色名的外键。如果恰好有一个外键满足此条件，则连接条件使用此外键。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每组表对，如果没有任何外键具有与表的相关名相同的名称，则 Adaptive Server Anywhere 将查找这些表间的任何外键关系。如果恰好有一个外键关系，则连接条件使用此关系。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每组对，如果没有任何外键关系，则发出错误消息。
- 如果 Adaptive Server Anywhere 能够为每一组表对确定恰好一个连接条件，则它将使用关键字 AND 来组合这些连接条件。

示例 1

考虑以下 5 个表的连接：

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

在此例中， Adaptive Server Anywhere 通过在 `(A,B)` 与 E 之间*或者* `C NATURAL JOIN D` 与 E 之间生成条件，为连接到 E 的键连接生成连接条件。「不包含逗号的表的表达式的键连接」第 320 页中说明了这一情况。

如果 Adaptive Server Anywhere 在 (A,B) 和 E 间生成连接条件，则它需要创建两个连接条件，一个用于 A-E，一个用于 B-E。它必须查找每一表对内的有效的外键关系。「表的表达式列表的键连接」第 322 页中说明了这一情况。

如果 Adaptive Server Anywhere 在 C NATURAL JOIN D 和 E 之间创建连接条件，则它将只创建一个连接条件，因此，必须只在 C-E 和 D-E 对中找到一个外键关系。「不包含逗号的表的表达式的键连接」第 320 页中说明了这一情况。

示例 2

以下是表的表达式和表的表达式列表的键连接的示例。该示例提供职位是销售代表并且还是经理的雇员的姓名和部门。

```
SELECT DISTINCT employee.emp_lname,
ky_dept_id.dept_name
 FROM (sales_order, department AS ky_dept_id)
   KEY JOIN (employee JOIN department AS d
     ON employee.emp_id = d.dept_head_id)
```

Adaptive Server Anywhere 生成两个连接条件：

- 表对 sales_order-employee 和 sales_order-d 之间必须正好有一种外键关系，即 sales_order.sales_rep = employee.emp_id。
- 表对 ky_dept_id-employee 和 ky_dept_id-d 之间必须正好有一种外键关系，即 ky_dept_id.dept_id = employee.dept_id。

该示例与以下语句是等效的。在以下版本中，不必创建相关名 department AS ky_dept_id，因为只有在说明应使用两个外键中的哪一个来连接 employee 和 department 时才需要相关名。

```
SELECT DISTINCT employee.emp_lname,
department.dept_name
 FROM (sales_order, department)
   JOIN (employee JOIN department AS d
       ON employee.emp_id = d.dept_head_id)
     ON sales_order.sales_rep = employee.emp_id
       AND department.dept_id = employee.dept_id
```

视图和派生表的键连接

当您在键连接中包括视图或派生表时， Adaptive Server Anywhere 遵守与表相同的基本过程，但还有以下差异：

- 对于每一键连接， Adaptive Server Anywhere 考虑在查询和视图的 FROM 子句中使用表对，并且为所有对的集合生成一个连接条件，而与视图中的 FROM 子句是否包含逗号或连接关键字无关。
- Adaptive Server Anywhere 基于其角色名与视图或派生表的相关名具有相同名称的外键连接表。
- 当您在键连接中包括视图或派生表时，该视图不能包含 UNION、ORDER BY、 DISTINCT、 GROUP BY 或集合函数。如果它包含任何这些项，则发出错误消息。

派生表的工作方式与视图大体相同。唯一差别就是：表的定义包括在语句中，而不是引用预定义的视图。

示例 1

例如，在以下语句中， View1 是一个视图。

```
SELECT *
 FROM View1 KEY JOIN B
```

View1 的定义可以是以下任何定义，并可导致生成与 B 相同的连接条件。（结果集将不同，但连接条件将是相同的。）

```
SELECT *
 FROM C CROSS JOIN D
```

或者

```
SELECT *
 FROM C,D
```

或者

```
SELECT *
 FROM C JOIN D ON (C.x = D.y)
```

在每一个例子中，为了为 View1 和 B 的键连接生成连接条件，Adaptive Server Anywhere 都会考虑使用表对 C-B 和 D-B，并且生成单个连接条件。它基于在 「表的表达式的键连接」第 320 页中描述的用于多个外键关系的规则生成连接条件，只不过它会查找与视图 （而不是在视图中引用的表）的相关名同名的外键。

使用上述任何视图定义，您可以按如下所示解释 `View1 KEY JOIN B` 的处理过程：

Adaptive Server Anywhere 通过考虑使用表对 C-B 和 D-B 来生成单个连接条件。它根据有多个外键关系时用于确定键连接的规则，生成连接条件：

- 首先，它为具有与视图的相关名相同的角色名的单个外键查找 C-B 和 D-B。如果恰好有一个外键满足此条件，则连接条件使用此外键。如果有多个外键具有与视图的相关名相同的角色名，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键与视图的相关名同名，则 Adaptive Server Anywhere 将查找这些表间的任何外键关系。如果有一个外键关系，则连接条件使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键关系，则发出错误消息。

假设此生成的连接条件是 `B.y = D.z`。我们现在可以扩展该原始连接。

```
SELECT *
 FROM View1 KEY JOIN B
```

等效于

```
SELECT *
 FROM View1 JOIN B ON B.y = View1.z
```

有关详细信息，请参见 「在有多个外键关系时的键连接」第 317 页。

示例 2

以下视图包含与每一部门的经理有关的所有雇员信息。

```
CREATE VIEW V AS
 SELECT department.dept_name, employee.*
 FROM employee JOIN department
   ON employee.emp_id = department.dept_head_id
```

以下查询将该视图连接到表的表达式。

```
SELECT *
 FROM V KEY JOIN (sales_order, department ky_dept_id)
```

这等效于

```
SELECT *
 FROM V JOIN (sales_order, department ky_dept_id)
 ON (V.emp_id = sales_order.sales_rep
 AND V.dept_id = ky_dept_id.dept_id)
```

描述键连接的操作的规则

以下规则总结前面提供的信息。

规则 1：两个表的键连接

此规则适用于 `A KEY JOIN B`，其中 A 和 B 是基表或临时表。

1. 从 A 中查找所有引用 B 的外键。

   如果存在其角色名是表 B 的相关名的外键，则将它标记为首选外键。

2. 从 B 中查找所有引用 A 的外键。

   如果存在其角色名是表 A 的相关名的外键，则将它标记为首选外键。

3. 如果有多个首选键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。

4. 如果有单个首选键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。

5. 如果没有首选键，则使用 A 和 B 间的其它外键：

   - 如果在 A 和 B 间有多个外键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
   - 如果有单个外键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
   - 如果没有任何外键，则该连接是无效的，将生成错误。

规则 2：不包含逗号的表的表达式的键连接

此规则适用于 `A KEY JOIN B`，其中 A 和 B 是不包含逗号的表的表达式。

1. 对于每对表 （来自表达式 A 的一个表和来自表达式 B 的一个表），列出所有外键，并且标记各表间的所有首选外键。用于确定首选外键的规则在前面的规则 1 中提供。
2. 如果有多个首选键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
3. 如果有单个首选键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
4. 如果没有首选键，则使用表对间的其它外键：
   - 如果有多个外键，则该连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
   - 如果有单个外键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
   - 如果没有任何外键，则该连接是无效的，将生成错误。

规则 3：表的表达式列表的键连接

此规则适用于 `(A1, A2, ...)KEY JOIN ( B1, B2, ...)`，其中的 A1、 B1 等等都是不包含逗号的表的表达式。

1. 对于每一对表的表达式 Ai 和 Bj，通过应用规则 1 或规则 2 为该表的表达式 `(Ai KEY JOIN Bj)` 找到唯一生成的连接条件。如果按规则 1 或规则 2 用于一对表的表达式的任何 KEY JOIN 是不明确的，则生成语法错误。
2. 用于此 KEY JOIN 表达式的生成的连接条件是在第 1 步中找到的连接条件的联合。

规则 4：表的表达式列表和不包含逗号的表的表达式的键连接

此规则适用于 `(A1, A2, ...)KEY JOIN ( B1, B2, ...)`，其中的 A1、 B1 等等都是可以包含逗号的表的表达式。

1. 对于每一对表的表达式 Ai 和 Bj，通过应用规则 1、规则 2 或规则 3 为表的表达式 `(Ai KEY JOIN Bj)` 找到唯一生成的连接条件。如果按规则 1、规则 2 或规则 3 用于一对表的表达式的任何 KEY JOIN 是不明确的，则生成一个语法错误。

2. 用于此 KEY JOIN 表达式的生成的连接条件是在第 1 步中找到的连接条件的联合。

第 9 章

公用表表达式

关于本章

使用 SELECT 语句的 WITH 前缀，您可以定义公用表表达式。您可以像使用临时视图一样在查询中使用这些公用表表达式。本章介绍了如何使用它们。

关于公用表表达式

公用表表达式是临时视图，它们只在某一条 SELECT 语句的范围内是已知的。使用它们，您编写查询会更容易，您还可以编写无法以其它方式表达的查询。

如果查询涉及多个集合函数，或者在存储过程中定义一个引用程序变量的视图，那么，公用表表达式就会很有用，或者可能是必需的。公用表表达式还提供了方便的方法来暂时存储值集。

使用递归公用表表达式，您可以查询呈现分层信息（例如，公司内的隶属关系）的表。使用它们，还可以解决部件激增问题和最短距离问题。

有关递归查询的信息，请参见 「递归公用表表达式」第 340 页。

例如，以确定哪个部门的雇员数量最多的问题为例。示例数据库中的雇员表列出了一家虚构公司内的所有雇员，并指定了每名雇员的工作部门。下面的查询列出了部门 ID 代码和每个部门中的雇员总数。

```
SELECT dept_id, count(*) AS n
 FROM employee
 GROUP BY dept_id
```

此查询可用于提取雇员最多的部门，如下所示：

```
SELECT dept_id, n
 FROM ( SELECT dept_id, count(*) AS n
        FROM employee GROUP BY dept_id ) AS a
 WHERE a.n =
   ( SELECT max(n)
     FROM ( SELECT dept_id, count(*) AS n
            FROM employee GROUP BY dept_id ) AS b )
```

虽然此语句提供了正确的结果，但它有一些缺点。第一个缺点是重复的子查询使得此语句非常笨拙。第二是此语句没有在子查询之间提供清楚的链接。

解决这些问题的一个方法是创建一个视图，然后使用该视图重新表达查询。这种方法可避免上述问题。

```
CREATE VIEW CountEmployees(dept_id, n) AS
     SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id;
```

```
SELECT dept_id, n
 FROM CountEmployees
 WHERE n = ( SELECT max(n)
             FROM CountEmployees );
```

这种方法的缺点是需要一些开销，因为引擎必须在创建视图时更新系统表。如果经常使用该视图，则使用这种方法是合理的。但是，如果只在特定 SELECT 语句中使用一次该视图，则首选方法是改为使用公用表表达式。

使用公用表表达式

公用表表达式是使用 WITH 子句定义的，该子句在 SELECT 语句中的 SELECT 关键字前面。子句的内容定义了之后可能会在语句中的其它位置引用的一个或多个临时视图。此子句的语法模拟 CREATE VIEW 语句的语法。您可以使用公用表表达式来表达上一个查询，如下所示。

```
WITH CountEmployees(dept_id, n) AS
   ( SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id )
 SELECT dept_id, n
 FROM CountEmployees
 WHERE n = ( SELECT max(n)
             FROM CountEmployees )
```

改为搜索雇员最少的部门说明这种查询可以返回多个行。

```
WITH CountEmployees(dept_id, n) AS
   ( SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id )
 SELECT dept_id, n
 FROM CountEmployees
 WHERE n = ( SELECT min(n)
             FROM CountEmployees )
```

在示例数据库中，两个部门共享着最少数量（即 9 名）的雇员。

多个相关名

就像在使用表时一样，您可以为公用表表达式的多个实例指定不同的相关名。这样做使您可以将公用表表达式与它本身连接。例如，下面的查询生成了具有相同数量雇员的部门对，虽然示例数据库中只有两个具有相同数量雇员的部门。

```
WITH CountEmployees(dept_id, n) AS
     ( SELECT dept_id, count(*) AS n
       FROM employee GROUP BY dept_id )
 SELECT a.dept_id, a.n, b.dept_id, b.n
 FROM CountEmployees AS a JOIN CountEmployees AS b
 ON a.n = b.n AND a.dept_id < b.dept_id
```

多个表的表达式

一个 WITH 子句可以定义多个公用表表达式。这些定义必须用逗号分隔开。下面的示例列出了工资额最少的部门和雇员数量最多的部门。

```
WITH
   CountEmployees(dept_id, n) AS
     ( SELECT dept_id, count(*) AS n
       FROM employee GROUP BY dept_id ),
   DeptPayroll( dept_id, amt ) AS
      ( SELECT dept_id, sum(salary) AS amt
        FROM employee GROUP BY dept_id )
 SELECT count.dept_id, count.n, pay.amt
 FROM CountEmployees AS count JOIN DeptPayroll AS pay
 ON count.dept_id = pay.dept_id
 WHERE count.n = ( SELECT max(n) FROM CountEmployees )
    OR pay.amt = ( SELECT min(amt) FROM DeptPayroll )
```

允许使用公用表表达式的位置

虽然可以在整个查询主体或任何子查询中引用公用表表达式定义，但只允许在三个位置使用它们。

- **顶级 SELECT 语句** 允许在顶级 SELECT 语句中使用公用表表达式，但不允许在子查询中使用。

  ```
  WITH DeptPayroll( dept_id, amt ) AS
     ( SELECT dept_id, sum(salary) AS amt
       FROM employee GROUP BY dept_id )
   SELECT dept_id, amt
   FROM DeptPayroll
   WHERE amt = ( SELECT max(amt)
                 FROM DeptPayroll )
  ```

- **视图定义中的顶级 SELECT 语句** 允许在定义视图的顶级 SELECT 语句中使用公用表表达式，但不允许在定义中的子查询中使用。

```
CREATE VIEW LargestDept (dept_id, size, pay) AS
    WITH
      CountEmployees(dept_id, n) AS
        ( SELECT dept_id, count(*) AS n
          FROM employee GROUP BY dept_id ),
      DeptPayroll( dept_id, amt ) AS
        ( SELECT dept_id, sum(salary) AS amt
          FROM employee GROUP BY dept_id )
   SELECT count.dept_id, count.n, pay.amt
   FROM CountEmployees count JOIN DeptPayroll pay
   ON count.dept_id = pay.dept_id
   WHERE count.n = ( SELECT max(n) FROM
CountEmployees )
      OR pay.amt = ( SELECT max(amt) FROM
DeptPayroll )
```

- **INSERT 语句中的顶级 SELECT 语句**　允许在 INSERT 语句中的顶级 SELECT 语句中使用公用表表达式，但不允许在 INSERT 语句中的子查询中使用。

```
INSERT INTO LargestPayrolls (dept_id, payroll,
date)
   WITH DeptPayroll( dept_id, amt ) AS
     ( SELECT dept_id, sum(salary) AS amt
       FROM employee
       GROUP BY dept_id )
   SELECT dept_id, amt, CURRENT TIMESTAMP
   FROM DeptPayroll
   WHERE amt = ( SELECT max(amt)
                 FROM DeptPayroll )
```

公用表表达式的典型应用

一般而言，只要表的表达式必须在一个查询中出现多次，公用表表达式就很有用。下面的典型情况适合于公用表表达式。

- 涉及多个集合函数的查询。
- 过程中必须包含对程序变量的引用的视图。
- 使用临时视图存储一组值的查询。

此列表并不完整。您可能会遇到许多其它要使用公用表表达式的情况。

多个集合函数

只要必须在一个查询中显示多个级别的集合，公用表表达式就很有用。这是上一部分使用的示例中的情况。任务是检索雇员数量最多的部门的部门 ID。为此，要使用 count 集合函数来计算每个部门的雇员数量，并使用 max 函数选择最大的部门。

在编写查询来确定哪个部门的工资额最多时，会出现类似的情况。 sum 集合函数用于计算每个部门的工资额，而 max 函数用于确定哪个部门最大。查询中同时出现这两个函数表明公用表表达式可能有用。

```
WITH DeptPayroll( dept_id, amt ) AS
     ( SELECT dept_id, sum(salary) AS amt
       FROM employee GROUP BY dept_id )
 SELECT dept_id, amt
 FROM DeptPayroll
 WHERE amt = ( SELECT max(amt)
               FROM DeptPayroll )
```

引用程序变量的视图

有时，公用表表达式对创建包含对程序变量的引用的视图可能会很方便。例如，您可以定义过程中标识特定客户的变量。您要查询该客户的购买历史记录，并且如果您要多次访问类似信息或者可能会使用多个集合函数，则您需要创建一个包含有关该特定客户的信息的视图。

您无法创建引用程序变量的视图，因为无法将视图范围限制为您的过程的范围。一旦创建了视图，就可以在其它环境中使用它。但是，您可以在您的过程中的查询内使用公用表表达式。因为公用表表达式的范围限制到语句，所以变量引用不会造成任何多义性，因此可以使用变量引用。

下面的语句在示例数据库中选择各位不同销售代表的销售总额。

```
SELECT emp_fname || ' ' || emp_lname AS sales_rep_name,
        sales_rep AS sales_rep_id,
        sum( p.unit_price * i.quantity ) AS total_sales
 FROM employee LEFT OUTER JOIN sales_order AS o
               INNER JOIN sales_order_items AS i
               INNER JOIN product AS p
 WHERE '2000-01-01' <= order_date
                    AND order_date < '2001-01-01'
 GROUP BY sales_rep, emp_fname, emp_lname
```

上面的查询是出现在下面过程中的公用表表达式的基础。销售代表的 ID 号和所讨论的年份是即将使用的参数。如此过程所示，可在 WITH 子句中引用过程参数和任何已声明的局部变量。

```
CREATE PROCEDURE sales_rep_total (
   IN rep  INTEGER,
   IN yyyy INTEGER )
 BEGIN
   DECLARE start_date DATE;
   DECLARE end_date   DATE;
   SET start_date = YMD( yyyy,  1,  1 );
   SET   end_date = YMD( yyyy, 12, 31 );
   WITH total_sales_by_rep ( sales_rep_name,
                             sales_rep_id,
                             month,
                             order_year,
                             total_sales ) AS
   ( SELECT emp_fname || ' ' || emp_lname AS
sales_rep_name,
            sales_rep AS sales_rep_id, month(
order_date),
            year(order_date),
            sum( p.unit_price * i.quantity ) AS
total_sales
     FROM employee LEFT OUTER JOIN sales_order o
                        INNER JOIN sales_order_items i
                        INNER JOIN product p
     WHERE start_date <= order_date AND
                         order_date <= end_date AND
           sales_rep = rep
     GROUP BY year(order_date), month(order_date),
              emp_fname, emp_lname, sales_rep )
   SELECT sales_rep_name,
            monthname( YMD(yyyy, month, 1) ) AS
month_name,
            order_year,
```

```
            total_sales
   FROM total_sales_by_rep
   WHERE total_sales =
    ( SELECT max( total_sales) FROM total_sales_by_rep
)
   ORDER BY order_year ASC, month ASC;
 END;
```

下面的语句说明了如何调用上面的过程。

```
CALL sales_rep_total(129, 2000);
```

存储值的视图

有时，公用表表达式在某个 SELECT 语句或某个过程中存储一组特定的值会很有用。例如，假定一家公司要按三分之一年度而不是按季度分析它的销售人员的结果。由于没有代表三分之一的内置日期部分（虽然有代表季度的内置日期部分），所以有必要将这些日期存储在过程中。

```
WITH thirds (q_name, q_start, q_end) AS
 ( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
   SELECT 'T2', '2000-05-01', '2000-08-31' UNION
   SELECT 'T3', '2000-09-01', '2000-12-31' )
 SELECT q_name,
        sales_rep,
        count(*) AS num_orders,
       sum( p.unit_price * i.quantity ) AS total_sales
 FROM thirds LEFT OUTER JOIN sales_order AS o
     ON q_start <= order_date AND order_date <= q_end
                  INNER JOIN sales_order_items AS i
                  INNER JOIN product AS p
  GROUP BY q_name, sales_rep
  ORDER BY q_name, sales_rep
```

使用此方法时应该小心，因为值可能需要定期维护。例如，如果要为任何其它年度使用上面的语句，则必须对它进行修改。

您还可以在过程中应用此技术。下面的示例声明了一个过程，该过程将所讨论的年份作为参数。

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
 BEGIN
   WITH thirds (q_name, q_start, q_end) AS
   ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30)
UNION
     SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31)
UNION
     SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31) )
```

```
   SELECT q_name,
          sales_rep,
          count(*) AS num_orders,
          sum(p.unit_price * i.quantity) AS total_sales
   FROM thirds JOIN sales_order AS o
     ON q_start <= order_date AND order_date <= q_end
             KEY JOIN sales_order_items AS i
             KEY JOIN product AS p
   GROUP BY q_name, sales_rep
   ORDER BY q_name, sales_rep;
 END;

CALL sales_by_third (2000);
```

递归公用表表达式

公用表表达式可以是递归的。当 RECURSIVE 关键字紧跟在 WITH 后面出现时，公用表表达式是递归的。一个 WITH 子句可以包含多个递归表达式，并且可以同时包含递归公用表表达式和非递归公用表表达式。

递归公用表表达式提供了一种方便的方法来编写将关系返回到任意深度的查询。例如，给定一个呈现公司内的隶属关系的表，您可以很容易地编写一个查询，让它返回所有隶属于某特定人员的雇员。

根据您编写查询的方式，您可以限制递归级别数，也可以不加限制。如果限制级别数，您可以只返回顶级管理人员（举例来说），但是，如果命令链比您的预期要长，则可能会排除一些雇员。如果不对级别数加以限制，则可确保不会排除任何雇员，但是，如果图形包含任何循环，则可能会引入无限递归。例如，如果某雇员直接或间接地向他自己报告。公司的管理层次结构中也可能会发生这种情况，例如，公司的某名雇员也是董事会成员。

递归提供了更加容易的方法来遍历呈现树状或类似树状的数据结构的表。在不使用递归表达式的情况下使用一个语句遍历这种结构的唯一方法是针对每个可能的级别让表与它自身连接一次。例如，如果报告层次最多包含七个级别，您必须将雇员表与它自身连接七次。如果公司进行重组并且引入了一个新的管理级别，您必须重写查询。

递归公用表表达式包含一个*初始子查询*（也就是种子）以及一个在每次迭代期间给结果集追加其它行的*递归子查询*。连接这两个部分只能使用运算符 UNION ALL。初始子查询是普通的非递归查询，首先得到处理。递归部分包含对上一次迭代期间添加的行的引用。只要迭代不生成新行，递归就会自动停止。对于在上一次迭代之前选择的行，无法引用。

递归子查询的选择列表必须在编号和数据类型方面都与初始子查询的选择列表匹配。如果无法执行数据类型的自动转换，可显式转换一个子查询的结果，以便它们与其它子查询的结果匹配。

有关公用表表达式的详细信息，请参见 「关于公用表表达式」第 332 页。

选择层次数据

下面的查询说明了如何按管理级别列出雇员。级别 0 表示没有经理的雇员。级别 1 表示直接向某一位级别 0 的经理报告的雇员，级别 2 表示直接向级别 1 的经理报告的雇员，依此类推。

```
WITH RECURSIVE
   manager ( emp_id, manager_id,
             emp_fname, emp_lname, mgmt_level ) AS
 ( ( SELECT emp_id, manager_id,        -- initial
subquery
            emp_fname, emp_lname, 0
     FROM employee AS e
     WHERE manager_id = emp_id )
   UNION ALL
   ( SELECT e.emp_id, e.manager_id,    -- recursive
subquery
            e.emp_fname, e.emp_lname, m.mgmt_level + 1
     FROM employee AS e JOIN manager AS m
      ON   e.manager_id =  m.emp_id
       AND e.manager_id <> e.emp_id
       AND m.mgmt_level < 20 ) )
 SELECT * FROM manager
 ORDER BY mgmt_level, emp_lname, emp_fname
```

递归查询中将管理级别限制为低于 20 的条件是一项重要的预防措施。它可以防止在表数据包含循环时出现无限递归。

MAX_RECURSIVE _ITERATIONS 选项

选项 MAX_RECURSIVE_ITERATIONS 旨在捕获失控的递归查询。此选项的缺省值是 100。超过此递归级别数的递归查询会终止，但是会导致出现错误。

虽然此选项似乎可以降低停止条件的重要性，但是通常并不是这样。在每次迭代期间选择的行数可能会以指数方式增加，在达到最大值之前就会严重影响数据库性能。递归查询中的停止条件提供了为每种情况设置适当限制的方法。

递归公用表表达式的限制

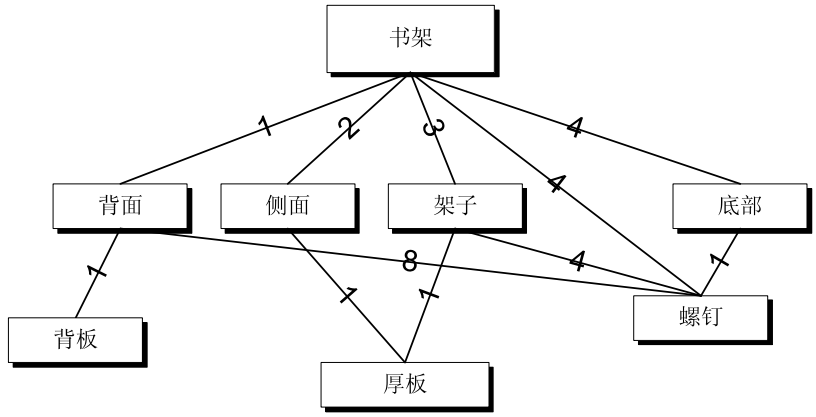
以下限制适用于递归公用表表达式。

- 递归公用表表达式的定义中不能出现对其它递归公用表表达式的引用。因此，递归公用表表达式不能相互递归。但是，非递归公用表表达式可以包含对递归公用表表达式的引用，递归公用表表达式可以包含对非递归公用表表达式的引用。
- 允许在初始子查询和递归子查询之间使用的唯一集合运算符是 UNION ALL。禁止使用其它集合运行符。
- 在递归子查询的定义中，对递归表表达式的自我引用只能出现在递归子查询的 FROM 子句中。
- 当自我引用出现在递归子查询的 FROM 子句中时，对递归表的引用不能出现在外连接的空值提供方。
- 递归子查询不能包含 DISTINCT、 GROUP BY 或 ORDER BY 子句。
- 递归子查询不能使用任何集合函数。
- 为了防止递归查询失控，如果递归级别数超过 MAX_RECURSIVE_ITERATIONS 选项的当前设置，则会生成错误。此选项的缺省值是 100。

部件激增问题

部件激增问题是递归的典型应用。在此问题中，组合特定对象所必需的组件由图形表示。目标是使用数据库表表示此图形，然后计算必需元素部件的总数。

例如，下图显示了一个简单书架的组件。书架由三层架板、一块背板和四条腿组成，四条腿用四个螺钉固定。每块架板用四个螺钉固定。背板用八个螺钉固定。



下表中的信息表示书架图形的边。第一列命名组件，第二列命名该组件的其中一个子组件，第三列指定需要多少个子组件。

组件	子组件	数量
书架	背面	1
书架	侧面	2
书架	架子	3
书架	底部	4
书架	螺钉	4

组件	子组件	数量
背面	背板	1
背面	螺钉	8
侧面	厚板	1
架子	厚板	1
架子	螺钉	4

下面的语句创建了书架表并插入了上表中显示的数据。

```
CREATE TABLE bookcase (
        component      VARCHAR(9),
        subcomponent   VARCHAR(9),
     quantity       integer,
      PRIMARY KEY (component, subcomponent)
 );

INSERT INTO bookcase
   SELECT 'bookcase', 'back',      1 UNION
   SELECT 'bookcase', 'side',      2 UNION
   SELECT 'bookcase', 'shelf',     3 UNION
   SELECT 'bookcase', 'foot',      4 UNION
   SELECT 'bookcase', 'screw',     4 UNION
   SELECT 'back',     'backboard', 1 UNION
   SELECT 'back',     'screw',     8 UNION
   SELECT 'side',     'plank',     1 UNION
   SELECT 'shelf',    'plank',     1 UNION
   SELECT 'shelf',    'screw',     4;
```

创建了书架表以后，您可以使用以下查询重新为它的部件创建表，如上所示。

```
SELECT * FROM bookcase
 ORDER BY component, subcomponent;
```

构建了此表后，可以生成一个由基本部件及构建书架所需的每种基本部件的数量组成的列表。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT component, subcomponent, quantity
   FROM bookcase WHERE component = 'bookcase'
     UNION ALL
   SELECT b.component, b.subcomponent, p.quantity *
b.quantity
   FROM parts p JOIN bookcase b ON p.subcomponent =
b.component )
 SELECT subcomponent, sum(quantity) AS quantity
 FROM parts
 WHERE subcomponent NOT IN ( SELECT component FROM
bookcase )
 GROUP BY subcomponent
 ORDER BY subcomponent;
```

下面显示了此查询的结果。

子组件	数量
背板	1
底部	4
厚板	5
螺钉	24

或者，您也可以重写此查询以执行其它级别的递归，从而消除在主 SELECT 语句中使用子查询的需要：

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT component, subcomponent, quantity
   FROM bookcase WHERE component = 'bookcase'
     UNION ALL
   SELECT p.subcomponent, b.subcomponent,
     IF b.quantity IS NULL
     THEN p.quantity
     ELSE p.quantity * b.quantity
     ENDIF
   FROM parts p LEFT OUTER JOIN bookcase b
   WHERE p.subcomponent = b.component AND
         p.subcomponent IS NOT NULL
```

```
 )
SELECT component, sum(quantity) FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```

此查询的结果与上一个查询的那些结果相同。

递归公用表表达式中的数据类型声明

临时视图中各列的数据类型由初始子查询中的数据类型定义。递归子查询中各列的数据类型必须匹配。数据库服务器会自动尝试转换由递归子查询返回的值，以便与初始查询的那些值匹配。如果这无法实现，或者如果转换中可能会丢失信息，则会生成错误。

一般而言，当初始子查询返回实际值或 NULL 值时，通常需要进行显式转换。当初始子查询从与递归子查询不同的列中选择值时，也可能需要进行显式转换。

如果初始子查询的列与递归子查询的列具有不同的域，则可能需要进行转换。对于初始子查询中的 NULL 值，必须总是进行转换。

例如，书架部件激增示例会正确运行，因为初始子查询返回书架表的行，并因此继承了所选列的数据类型。

有关详细信息，请参见 『ASA SQL 用户指南』> 「部件激增问题」。

如果按照如下所示重写此查询，则需要进行显式转换。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT NULL, 'bookcase', 1         -- ERROR! Wrong
domains!
     UNION ALL
   SELECT b.component, b.subcomponent,
          p.quantity * b.quantity
   FROM parts p JOIN bookcase b
     ON p.subcomponent = b.component )
 SELECT * FROM parts
 ORDER BY component, subcomponent
```

如果没有进行转换，就会由于以下原因而导致错误：

- 组件名的正确数据类型是 VARCHAR，但是第一列为 NULL。
- 假定数字 1 是短整数，但是数量列的数据类型是 INT。

不需要对第二列进行转换，因为初始查询的该列已经是字符串。

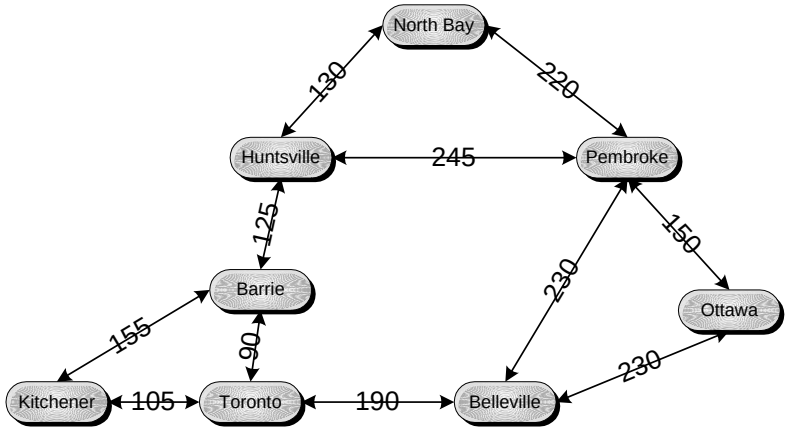
如果转换初始子查询中的数据类型，查询的行为可以像预期一样：

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT CAST(NULL AS VARCHAR),  -- CASTs must be used
          'bookcase',              -- to declare the
          CAST( 1   AS INT)       -- correct datatypes
       UNION ALL
   SELECT b.component, b.subcomponent,
          p.quantity * b.quantity
   FROM parts p JOIN bookcase b
     ON p.subcomponent = b.component )
 SELECT * FROM parts
 ORDER BY component, subcomponent
```

最短距离问题

您可以使用递归公用表表达式在定向图形上查找所需的路径。数据库表中的每一行表示一个定向边。每一行指定一个原点、一个终点，以及从原点到达终点的成本。根据问题的不同，成本可能表示距离、旅行时间或一些其它测量单位。使用递归，您可以通过此图形查看可能的路线。然后，您可以从可能的路线集中选择您感兴趣的路线。

例如，以查找 Kitchener 市和 Pembroke 市之间的可取驱车路线的问题为例。可能的路线有多条，每一条都会使您经过一组不同的中间城市。目标是找到最短的路线，以及将它们与合理的备用方案进行比较。



首先，定义一个表来表示此图形的边并为每个边插入一行。由于此图形的所有边都恰巧是双向的，所以也必须插入表示相反方向的边。通过选择初始行集可以实现这一点，但要将原点和终点互换。例如，一个行必须表示从 Kitchener 到 Toronto 的行程，另一行表示从 Toronto 返回 Kitchener 的行程。

```
CREATE TABLE travel (
     origin      AS VARCHAR(10),
     destination AS VARCHAR(10),
     distance    AS INT,
   PRIMARY KEY (origin, destination)
 );
```

```
INSERT INTO travel
   SELECT 'Kitchener',  'Toronto',    105 UNION
   SELECT 'Kitchener',  'Barrie',     155 UNION
   SELECT 'North Bay',  'Pembroke',   220 UNION
   SELECT 'Pembroke',   'Ottawa',     150 UNION
   SELECT 'Barrie',     'Toronto',     90 UNION
   SELECT 'Toronto',    'Belleville', 190 UNION
   SELECT 'Belleville', 'Ottawa',     230 UNION
   SELECT 'Belleville', 'Pembroke',   230 UNION
   SELECT 'Barrie',     'Huntsville', 125 UNION
   SELECT 'Huntsville', 'North Bay',  130 UNION
   SELECT 'Huntsville', 'Pembroke',   245;

INSERT INTO travel   -- Insert the return trips
   SELECT destination, origin, distance
   FROM travel;
```

下一个任务是编写递归公用表表达式。由于旅程将从 Kitchener 开始，因此初始子查询会先选择从 Kitchener 出发的所有可能路线以及每一条路线的距离。

递归子查询可延伸路线。对于每一条路线，它都会添加从前面路段的终点继续延伸的路段，这样就增加了新路段的长度，从而可以保持每条路线的总运行成本。为了提高效率，如果路线满足以下任一条件，就会终止：

- 路径返回到起始位置。
- 路径返回到前面的位置。
- 路径到达所需的终点。

在当前示例中，任何路径都不应返回到 Kitchener，并且，如果到达 Pembroke，所有路径都应终止。

保证递归查询在您使用它们查看循环图形时正确终止特别重要。在这种情况下，由于一条路线可以包括两个中间城市间的任意大的往返行程数，因此上面的条件是不够的。下面的递归查询通过将任意给定路线中的最大路段数限制为七段来保证终止。

由于示例查询的关键是选择可行路线，因此主查询只选择比最短路线长小于 50% 的那些路线。

```
WITH RECURSIVE
     trip (route, destination, previous, distance,
segments) AS
 ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
          destination, origin, distance, 1
   FROM travel
   WHERE origin = 'Kitchener'
     UNION ALL
   SELECT route || ', ' || v.destination,
          v.destination,            -- current endpoint
          v.origin,                -- previous endpoint
          t.distance + v.distance,  -- total distance
          segments + 1              -- total number of
segments
   FROM trip t JOIN travel v ON t.destination =
v.origin
   WHERE v.destination <> 'Kitchener'  -- Don't return
to start
     AND v.destination <> t.previous   -- Prevent
backtracking
     AND v.origin      <> 'Pembroke'   -- Stop at the
end
     AND segments                      -- TERMINATE
RECURSION!
          < ( SELECT count(*)/2 FROM travel ) )
 SELECT route, distance, segments FROM trip
 WHERE destination = 'Pembroke' AND
       distance < 1.5 * ( SELECT min(distance)
                          FROM trip
                        WHERE destination = 'Pembroke'
)
 ORDER BY distance, segments, route;
```

当针对上面的数据集运行此语句时，会生成下面的结果。

路线	距离	路段
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4

路线	距离	路段
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

使用多个递归公用表表达式

递归查询可以包括多个递归查询，只要这些查询不相交即可。它还可以混合包括递归公用表表达式和非递归公用表表达式。如果至少有一个公用表表达式是递归的，则必须有 RECURSIVE 关键字。

例如，下面的查询（它与上一个查询返回的结果相同）使用第二个非递归公用表表达式选择最短路线的长度。第二个公用表表达式的定义通过逗号与第一个公用表表达式的定义分隔开。

```
WITH RECURSIVE
   trip (route, destination, previous, distance,
segments) AS
     ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
              destination, origin, distance, 1
       FROM travel
       WHERE origin = 'Kitchener'
       UNION ALL
       SELECT route || ', ' || v.destination,
              v.destination,
              v.origin,
              t.distance + v.distance,
              segments + 1
       FROM trip t JOIN travel v ON t.destination =
v.origin
       WHERE v.destination <> 'Kitchener'
         AND v.destination <> t.previous
         AND v.origin      <> 'Pembroke'
         AND segments
               < ( SELECT count(*)/2 FROM travel ) ),
   shortest ( distance ) AS                  --
Additional,
     ( SELECT min(distance)                  -- non-
recursive
       FROM trip                             -- common
table
       WHERE destination = 'Pembroke' )      --
expression
 SELECT route, distance, segments FROM trip
 WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT distance FROM shortest
)
 ORDER BY distance, segments, route;
```

像非递归公用表表达式一样，递归表达式在用于存储过程中时可以包含对局部变量或过程参数的引用。例如，下面定义的 best_routes 过程可识别两个指定城市间的最短路线。

```
CREATE PROCEDURE best_routes (
    IN initial VARCHAR(10),
    IN final   VARCHAR(10)
 )
 BEGIN
   WITH RECURSIVE
       trip (route, destination, previous, distance,
segments) AS
   ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
            destination, origin, distance, 1
     FROM travel
     WHERE origin = initial
       UNION ALL
     SELECT route || ', ' || v.destination,
          v.destination,           -- current endpoint
          v.origin,               -- previous endpoint
            t.distance + v.distance,  -- total distance
            segments + 1              -- total number
of segments
     FROM trip t JOIN travel v ON t.destination =
v.origin
     WHERE v.destination <> initial     -- Don't return
to start
       AND v.destination <> t.previous  -- Prevent
backtracking
       AND v.origin      <> final       -- Stop at the
end
       AND segments                      -- TERMINATE
RECURSION!
            < ( SELECT count(*)/2 FROM travel ) )
   SELECT route, distance, segments FROM trip
   WHERE destination = final AND
         distance < 1.4 * ( SELECT min(distance)
                            FROM trip
                           WHERE destination = final )
   ORDER BY distance, segments, route;
 END;

CALL best_routes ( 'Pembroke', 'Kitchener' );
```

第 10 章

使用 OLAP

关于本章

本章介绍联机分析处理 (OLAP) 功能。 OLAP 是一种用于对存储在关系数据库中的信息进行数据分析的方法。使用 OLAP，您可以获得具有小计行的结果集并将数据组织成多维 CUBE。它允许您使用过滤器，以便能够深入分析数据，而且，它是为快速返回结果集而设计的。

了解 OLAP

使用联机分析处理 (OLAP)，您可以分析存储在关系数据库中的数据。使用 OLAP 获取的结果集可以具有小计行，并且可以组织成多维 CUBE。

使用 OLAP，您可以进行时间序列分析，成本分配、目标寻求、特别多维结构更改、非过程建模、异常警告和数据采集。 Adaptive Server Anywhere 提供了数据的多维概念视图，包括对层次和多个层次的完全支持。

使用 OLAP 功能，您可以完成以下任务：

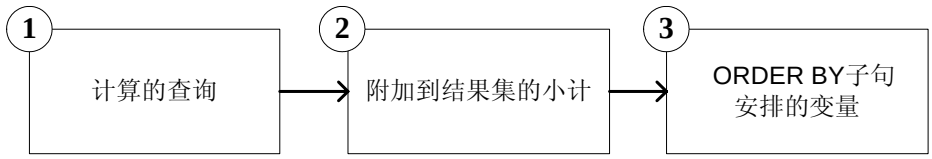
- 在各个维度间应用的计算
- 在整个层次中应用的计算
- 各时间序列的趋势分析
- 直达基础数据

有些数据库产品提供单独的 OLAP 模块，该模块要求先将数据从数据库移动到 OLAP 模块中，然后再对其进行分析。 Adaptive Server Anywhere 在数据库本身中内置了 OLAP 功能。这样就可以更加容易和更加顺畅地实现部署以及与其它数据库功能 （例如存储过程）的集成。

了解小计

小计行可以帮助您分析数据，尤其是在以下情况下：有大量数据、数据有不同的维数、数据包含在不同的表甚至完全不同的数据库中。例如，一名销售经理可能会发现按销售代表、区域和季度进行分类的有关销售数据的报告对于了解销售模式十分有用。数据小计可让销售经理从不同的角度了解总体销售情况。如果根据销售经理要比较的条件提供了摘要信息，分析此数据就会更加容易。

使用 OLAP，最终用户看不到分析和计算行和列小计的过程。以下图示从概念上说明了 Adaptive Server Anywhere 如何创建小计：



1. 这一步会生成一个中间结果集。根据在查询中设置的条件，查询被制成表格形式。
2. OLAP 创建小计并将它们附加到结果集中。
3. 行按照查询中的 ORDER BY 子句排列。

注意
对于最终用户而言，此序列是完全连续的。

关于 OLAP 操作

请像下面这样考虑 OLAP 操作 （ROLLUP、 CUBE 和 GROUPING SETS）：

- **前缀**列表是为查询而构建的。前缀是 *group-by-list* 中的项目子集。

前缀是通过从查询的*分组依据列表*中的各项中排除最右侧的一项或多项而构建的。排除后所剩余的列称为**前缀列**。

以下面的 ROLLUP 查询为例：

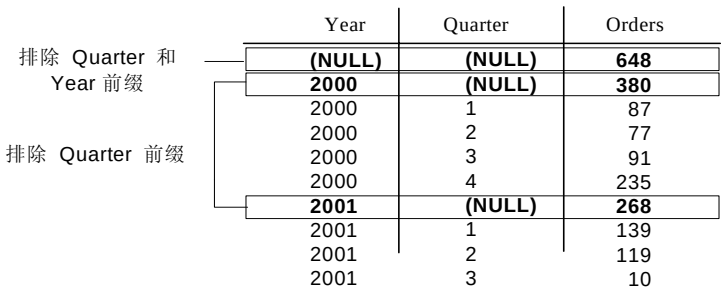
```
SELECT year (order_date) Year, quarter
(order_date) Quarter, count (*) Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

在上面的示例中，*分组依据列表*包括两个变量 （**Year** 和 **Quarter**）。

```
GROUP BY ROLLUP (Year, Quarter)
```

此查询有两个前缀：

- **排除 Quarter** 前缀列集包含单个列 Year。
- **排除 Quarter 和 Year** 没有前缀列。



Year	Quarter	Orders
(NULL)	**(NULL)**	**648**
2000	**(NULL)**	**380**
2000	1	87
2000	2	77
2000	3	91
2000	4	235
2001	**(NULL)**	**268**
2001	1	139
2001	2	119
2001	3	10

注意
前缀个数与*分组依据列表*中的项数相同。

- 对于每个前缀，小计行是对应于所有那些前缀列具有相同值的行而构建的。

例如，让我们再次检查一下下面的查询：

```
SELECT year (order_date) AS Year, quarter
(order_date) AS Quarter, COUNT (*) Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

在此查询中，包含 Year 列的前缀位于 Year=2000 的摘要行和 Year=2001 的摘要行前面。不包括列的前缀有一个摘要行，该行是中间结果集中所有行的小计。

小计行中每一列的值如下：

- **前缀中包括的列** 列的值。例如，在前面的查询中，在 Year=2000 的行的小计中， Year 列的值为 2000。
- **前缀中排除的列** NULL。例如， Quarter 列中具有由包含 Year 列的前缀生成的小计行的值，为 NULL。
- **集合函数** 对排除的列的值进行的合计。

  小计值是对基础数据中的行而不是对集合行进行计算而得出的。在许多情况下，例如，使用 SUM 或 COUNT 时，结果都是相同的，但是，在使用 AVG、 STDDEV 和 VARIANCE 这样的静态函数时，结果是不同的，因而这时差别就很重要。

NULL 值和小计行

当 GROUP BY 操作的输入中的行包含 NULL 时，可能会将由 ROLLUP、 CUBE 或 GROUPING SETS 操作添加的小计行与包含的 NULL 值属于原始输入数据的行混淆。

GROUPING 函数通过以下方法将小计行与其它行区分开：采用*分组依据列表*中的列作为它的参数，并在该列为 NULL 时 （因为行是小计行）返回 1，在该列不为 NULL 时返回 0。

示例

下面的示例在结果集中包括 GROUPING 列。输入数据包含 NULL 的结果行被突出显示，小计行则没有。 GROUPING 列会突出显示。查询是 **employee** 表和 **sales_order** 表之间的外连接。查询选择居住在德克萨斯、纽约或加利福尼亚的女雇员。 NULL 会显示在与不是销售代表（因而没有销售额）的那些女雇员相对应的列中。

```
SELECT employee.emp_id AS Employee,
        year(order_date) AS Year,
        COUNT(*) AS Orders,
        GROUPING ( Employee ) AS GE,
        GROUPING ( Year ) AS GY
 FROM employee LEFT OUTER JOIN sales_order
 ON employee.emp_id = sales_order.sales_rep
 WHERE employee.sex IN ('F')
 AND employee.state IN ('TX', 'CA', 'NY')
 GROUP BY ROLLUP (Year, Employee)
 ORDER BY Year, Employee
```

下面的表是查询的结果集。

这些雇员不是销售代表，所以没有要记录的销售额

Employee	Year	Orders	GY	GE
(NULL)	(NULL)	5	1	0
(NULL)	(NULL)	169	1	1
102	**(NULL)**	**1**	**0**	**0**
390	**(NULL)**	**1**	**0**	**0**
1062	**(NULL)**	**1**	**0**	**0**
1090	**(NULL)**	**1**	**0**	**0**
1507	**(NULL)**	**1**	**0**	**0**
(NULL)	2000	98	1	0
667	2000	34	0	0
949	2000	31	0	0
1142	2000	33	0	0
(NULL)	2001	66	1	0
667	2001	20	0	0
949	2001	22	0	0
1142	2001	24	0	0

使用 ROLLUP

ROLLUP 操作提供集合行的小计。它使用 GROUP BY 子句在查询的结果集中添加小计行。 ROLLUP 会生成一个结果集，显示所选列中的值的层次的集合。如果希望结果集显示总计和小计，请使用 ROLLUP 操作。

此 SQL 语法 ...	定义以下集合 ...
`GROUP BY ROLLUP (A, B, C);`	(A, B, C) (A, B) (A) ( )

有关 ROLLUP 查询返回的行的详细信息，请参见 「关于 OLAP 操作」第 357 页。有关在查询中使用 GROUP BY 子句的详细信息，请参见『ASA SQL 参考』> 「GROUP BY 子句」。

示例 1

在下面的示例中，查询返回按年和季度汇总销售订单数的数据。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

下面的表是查询的结果集。在结果集中突出显示小计行。每一小计行在计算小计的列中的值为 NULL。

	Year	Quarter	Orders
1	**(NULL)**	**(NULL)**	**648**
2	**2000**	**(NULL)**	**380**
3	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	235
2	**2001**	**(NULL)**	**268**
3	2001	1	139
	2001	2	119
	2001	3	10

标有 [1] 的行表示两年（2000、2001）和所有季度的订单总数。该行在 Year 列和 Quarter 列中的值均为 NULL，是所有列均未包括在前缀中的行。

注意
每个 ROLLUP 操作均返回的结果集中均具有这样一行：除集合列外，每个列中均显示 NULL。该行表示对执行集合函数的每一列的汇总。例如，如果 SUM 是正在讨论的集合函数，则该行将表示所有值的总数之和。

突出显示行 [2] 的行分别表示 2000 年和 2001 年的订单总数。这两个行的 Quarter 列中均为 NULL，因为该列中的值会累计，为 Year 提供小计。

注意
结果集中这种行的数目取决于 ROLLUP 查询中出现的变量数。

标有 [3] 的其余行通过提供两年内每个季度的订单总数提供了摘要信息。

示例 2

下面是 ROLLUP 操作的另一个示例，与第一个示例相比，它返回的结果集略微复杂一些。结果集按年、季度和区域汇总了销售订单数。在此示例中，只检查第一和第二季度和两个选定区域（加拿大和东部地区）。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, region, COUNT (*) AS Orders
 FROM sales_order
 WHERE region IN ('Canada', 'Eastern') AND quarter IN
('1', '2')
 GROUP BY ROLLUP (Year, Quarter, Region)
 ORDER BY Year, Quarter, Region
```

下面的表是查询的结果集。每一小计行在计算小计的列中的值为 NULL。

	Year	Quarter	Region	Orders
1	**(NULL)**	**(NULL)**	**(NULL)**	**183**
	2000	**(NULL)**	**(NULL)**	**68**
	2000	1	(NULL)	36
	2000	1	Canada	3
	2000	1	Eastern	33
2	2000	2	(NULL)	32
	2000	2	Canada	3
	2000	2	Eastern	29
	2001	**(NULL)**	**(NULL)**	**115**
	2001	1	(NULL)	57
	2001	1	Canada	11
	2001	1	Eastern	46
	2001	2	(NULL)	58
	2001	2	Canada	4
	2001	2	Eastern	54

第一行 [1] 是所有行的集合，在 Year、 Quarter 和 Region 列中为 NULL。该行的 Orders 列中的值表示 2000 年和 2001 年的第 1 和第 2 季度在加拿大和东部地区的订单总数。

标有 [2] 的行表示每年 （2000 年 和 2001 年）的第 1 和第 2 季度在加拿大和东部地区的销售订单总数。这些行 [2] 的值等于行 [1] 中呈现的总数。

标有 [3] 的行按区域提供有关给定年份和季度的订单总数的数据。

Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	**1**	**(NULL)**	**36**
2000	1	Canada	3
2000	1	Eastern	33
2000	**2**	**(NULL)**	**32**
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	**1**	**(NULL)**	**57**
2001	1	Canada	11
2001	1	Eastern	46
2001	**2**	**(NULL)**	**58**
2001	2	Canada	4
2001	2	Eastern	54

3

标有 [4] 的行在结果集中提供了有关每年、每季度和每个区域的订单总数的数据。



Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	1	(NULL)	36
2000	1	Canada	3
2000	1	Eastern	33
2000	2	(NULL)	32
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	1	(NULL)	57
2001	1	Canada	11
2001	1	Eastern	46
2001	2	(NULL)	58
2001	2	Canada	4
2001	2	Eastern	54

了解 ROLLUP 和小计行

ROLLUP 等效于一组 GROUP BY 查询的 UNION。以下查询的结果集是相同的。 GROUP BY A, B 的结果集包括所有那些 A 和 B 保持不变的行的小计。要实现联合，可为列 C 指派 NULL。

此 ROLLUP 查询 ...	等效于没有 ROLLUP 的此查询 ...
SELECT A, B, C, SUM(D) FROM T1 GROUP BY ROLLUP (A, B, C);	SELECT * FROM ((SELECT A, B, C, SUM(D) GROUP BY A, B, C) UNION ALL (SELECT A, B, NULL, SUM(D) GROUP BY A, B) UNION ALL (SELECT A, NULL, NULL, SUM(D) GROUP BY A) UNION ALL (SELECT NULL, NULL, NULL, SUM(D)))

使用 **CUBE**

CUBE 操作是一项 OLAP 功能。 OLAP 是一种组织数据的方法，它可以适应您分析和管理数据的方式，以便省时省力地生成您需要的结果集。您要预先计算报告的摘要值，这样可以加快报告计算的速度。如果要在报告中处理大量数据， CUBE 很有用。

像 ROLLUP 一样， CUBE 运算符为结果集的集合值提供小计。对变量执行 CUBE 操作时，结果集包括许多基于变量值组合的小计行。与 ROLLUP 不同的是， CUBE 可返回您在查询中指定的所有可能的变量组合。

CUBE 运算符会返回增加了数据维数信息的结果集。您可以进一步分析数据的列，这些列称为维。 CUBE 提供所有可能的维组合的交叉表格报告，并生成一个结果集，该结果集显示所选列中的值的所有组合的集合。

注意
当维不是同一层次的一部分时， CUBE 特别有用。

此 SQL 语法 ...	定义以下集合 ...
`GROUP BY CUBE (A, B, C);`	(A, B, C) (A, B) (A, C) (A) (B, C) (B) (C) ( )

有关 CUBE 查询返回的行的详细信息，请参见 「关于 OLAP 操作」第 357 页。有关在查询中使用 GROUP BY 子句的详细信息，请参见『ASA SQL 参考』> 「GROUP BY 子句」。

示例

在后面的示例中，查询会返回一个结果集，该结果集汇总了订单总数，然后按年和季度计算订单数的小计。

注意
随着要比较的变量数量的增加，查询返回的 CUBE 的复杂程度也会增加。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY CUBE (Year, Quarter)
 ORDER BY Year, Quarter
```

下面的表是查询的结果集。在结果集中突出显示小计行。每一小计行在计算小计的列中的值为 NULL。

	Year	Quarter	Orders
1	**(NULL)**	**(NULL)**	**648**
2	**(NULL)**	**1**	**226**
	(NULL)	**2**	**196**
	(NULL)	**3**	**101**
	(NULL)	**4**	**125**
3	**2000**	**(NULL)**	**380**
	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	125
3	**2001**	**(NULL)**	**268**
	2001	1	139
	2001	2	119
	2001	3	10

第一个突出显示的行 [1] 表示两年和所有季度内的订单总数。 Orders 列中的值是每个标有 [3] 的行中的值的总和。它也是标有 [2] 的行中的四个值的总和。

注意
所有 CUBE 操作返回的结果集均至少有一行在除集合列以外的每个列中显示 NULL。该行表示对执行集合函数的每一列的汇总。

下一组突出显示的行 [2] 按季度表示两年内的订单总数。后面的标有 [3] 的两行分别表示 2000 年和 2001 年中所有季度的订单总数。

了解 CUBE

CUBE 操作等效于包括所有可能的变量组合的 GROUPING SETS 查询。

此 CUBE 查询 ...	等效于没有 CUBE 的此查询 ...
`SELECT A, B, C, SUM( D ) FROM t GROUP BY CUBE( A, B, C)`	`SELECT A, B, C, SUM( D ) FROM t GROUP BY GROUPING SETS( ( A, B, C ), ( A, B ), ( A ), ( B, C ), ( B ), ( A, C ), ( C ), () )`

有关在前面的 SQL 查询中使用的 GROUPING SETS 语法的信息，请参见 「使用 GROUPING SETS」第 371 页。

使用 GROUPING SETS

使用 GROUPING SETS，您可以在同一查询中对几组不同分组列上的组进行计算。尽管 CUBE 和 ROLLUP 会在结果集中添加一组预定义的小计，但使用 GROUPING SETS 可以显式指定要添加哪些小计。这比使用 ROLLUP 或 CUBE 更加复杂，但是却使您可以更好地控制结果集。如果只想返回 CUBE 的多维结果集的一部分，此操作很有用。使用 GROUPING SETS，您可以选择要比较的变量而不必像 CUBE 一样返回所有变量的比较或者像 ROLLUP 一样返回层次子集。

此 SQL 语法 ...	定义以下集合 ...
`GROUP BY GROUPING SETS ((A, B), (A, C), (C));`	(A, B) (A, C) (C)

有关 GROUPING SETS 查询返回的行的详细信息，请参见 「关于 OLAP 操作」第 357 页。有关在查询中使用 GROUP BY 子句的详细信息，请参见 『ASA SQL 参考』> 「GROUP BY 子句」。

示例

下面的示例说明了可以如何使用 GROUPING SETS 定制从查询返回的结果。查询指定结果集返回每年的小计行，但不返回各个季度的小计行。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY GROUPING SETS ((Year, Quarter), (Year))
 ORDER BY Year, Quarter
```

下面的表是查询的结果集。小计行会突出显示。

Year	Quarter	Orders
2000	**(NULL)**	**380**
2000	1	87
2000	2	77
2000	3	91
2000	4	125
2001	**(NULL)**	**268**
2001	1	139
2001	2	119
2001	3	10

结果集只包括每年 （2000 和 2001）的小计行。请注意，它不提供两年和所有季度的订单数的总数之和。

如果使用 CUBE 运算符运行查询，您将获得所有订单、每个季度 （1、2、 3 和 4）和每年 （2000 和 2001）的附加小计行。

了解 GROUPING SETS

与 ROLLUP 相似， GROUPING SETS 操作等效于 GROUP BY 查询的 UNION ALL。

此 GROUPING SETS 查询 ...	等效于没有 GROUPING SETS 的此查询 ...
SELECT A, B, C, SUM(D) FROM t GROUP BY GROUPING SETS (A, B, C)	SELECT * FROM ((SELECT A, NULL, NULL, SUM(D) FROM t GROUP BY A) UNION ALL (SELECT NULL, B, NULL, SUM(D) FROM t GROUP BY B) UNION ALL (SELECT NULL, NULL, C, SUM(D) FROM t GROUP BY C))

使用连接的 GROUPING SETS

指定 GROUPING SETS 的另一种方法是使用连接的 GROUPING SETS。它们提供了一种简练的方法来生成较大的分组组合。使用连接的 GROUPING SETS 可根据每个 GROUPING SET 列表生成分组的向量积。您可以对使用 CUBE 和 ROLLUP 指定的分组执行连接。使用向量积运算，甚至可以使少量的连接分组生成大量最终组。

此 SQL 语法 ...	定义以下集合 ...
`GROUP BY GROUPING SETS (A, B)` `(Y, Z)`	(A, Y) (A, Z) (B, Y) (B, Z)

使用 OLAP 函数

使用 OLAP 函数，可以对数据执行分析任务，例如计算移动平均值、排序和累计值。您可以在 *select-list* 或 SELECT 语句的 ORDER BY 子句中包括 OLAP 函数。 OLAP 函数不能用作集合函数的参数。因此，您无法具有类似 SUM(RANK()) 的函数。

使用 OLAP 函数时，会指定一个**窗口**，该窗口定义应用该函数的行及其顺序。行集是作为相对于当前行而言在其之前和之后的一系列行或许多行定义的。例如，您可以计算前三个月期间的平均值。

以下是可用的 OLAP 函数：

- **秩函数** 让您按排序顺序编译数据集中的值列表。例如，按每个季度的总销售额的降序顺序列出销售代表。
- **报告函数** 让您比较非集合值与集合值。例如，列出花销小于平均值的所有季度。
- **窗口函数** 让您通过为数据定义移动窗口来分析数据。例如，计算含 30 天的时间段内的移动平均值，如 NASDAQ 数据。

秩函数

使用秩函数，您可以根据在查询中指定的顺序为结果集中的每一行计算排序值。例如，一名销售经理可能需要确定公司中的最领先或最落后的销售人员，业绩最好或最差的销售区域，或者销售情况最好或最差的产品。秩函数可以提供此信息。

以下统计函数被归到秩函数一类：

- **CUME_DIST 函数** 根据 ORDER BY 子句的定义，计算从查询返回的一行相对于该查询返回的其它行的位置。有关信息，请参见『ASA SQL 参考』> 「CUME_DIST 函数 [Aggregate]」。
- **RANK 函数** 根据 ORDER BY 子句的定义，计算从查询返回的每一行相对于该查询返回的其它行的排序。有关信息，请参见『ASA SQL 参考』> 「RANK 函数 [Aggregate]」。

- **DENSE_RANK 函数**　计算组中值的排序。 RANK 和 DENSE_RANK 的区别在于：当值之间有关联时， RANK 在排序顺序之间保留间隙。有关信息，请参见 『ASA SQL 参考』> 「DENSE_RANK 函数 [Aggregate]」。
- **PERCENT_RANK 函数**　计算排序的百分比而不是分数。此函数返回一个介于 0 和 1 之间的小数值。有关信息，请参见 『ASA SQL 参考』> 「PERCENT_RANK 函数 [Aggregate]」。
- **ROW_NUMBER 函数**　为每一行指派一个唯一编号，编号由 ORDER BY 子句确定。有关信息，请参见 『ASA SQL 参考』> 「ROW_NUMBER 函数 [Aggregate]」。

示例 1

下面的 SQL 查询查找犹他州的男女雇员并按薪水以降序顺序排列他们。

```
SELECT emp_lname, salary, sex,
      RANK () OVER (ORDER BY salary DESC) "Rank"
      FROM employee
 WHERE state IN ('UT')
```

下面的表是查询的结果集。

emp_lname	salary	sex	Rank
Shishov	72995.00	F	1
Wang	68400.00	M	2
Cobb	62000.00	M	3
Morris	61300.00	M	4
Diaz	54900.00	M	5
Driscoll	48023.00	M	6
Hildebrand	45829.00	F	7
Goggin	37900.00	M	8
Rebeiro	34576.00	M	9
Bigelow	31200.00	F	10
Lynch	24903.00	M	11

示例 2

您可以划分数据以提供不同的结果。使用示例 1 中的查询，您可以按性别划分数据，从而来更改数据。下面的示例按薪水以降序排列雇员并按性别划分雇员。

```
SELECT emp_lname, salary, sex,
      RANK () OVER (PARTITION BY sex
      ORDER BY salary DESC) "Rank"
      FROM employee
 WHERE state IN ('UT')
```

下面的表是查询的结果集。

这些雇员是按薪水以降序排列的，但会按照性别划分。

emp_lname	salary	sex	Rank
Wang	68400.00	M	1
Cobb	62000.00	M	2
Morris	61300.00	M	3
Diaz	54900.00	M	4
Driscoll	48023.00	M	5
Goggin	37900.00	M	6
Rebeiro	34576.00	M	7
Lynch	24903.00	M	8
Shishov	72995.00	F	1
Hildebrand	45829.00	F	2
Bigelow	31200.00	F	3

示例 3

下面的示例查找犹他州和亚利桑那州的一些女雇员并根据薪水以降序顺序排列她们。 PERCENT_RANK 函数用于以降序顺序提供累计总数。

```
SELECT emp_lname, salary,
      PERCENT_RANK () OVER (ORDER BY salary DESC)
"Rank"
      FROM employee
 WHERE state IN ('UT','AZ') AND sex IN ('F')
```

下面的表是查询的结果集。

emp_lname	salary	Rank
Shishov	72995.00	0
Jordan	51432.00	0.25
Hildebrand	45829.00	0.5
Bigelow	31200.00	0.75
Bertrand	29800.00	1

示例 4

您可以使用秩函数在数据集中查找最高或最低的百分点。在下面的示例中，查询会返回其薪水在数据集的最高的五个百分点之内的男雇员。

```
SELECT *
 FROM ( SELECT emp_lname, salary,
        PERCENT_RANK () OVER (ORDER BY salary DESC)
"Rank"
        FROM employee
        WHERE sex IN ('M')  )
        AS DT ( last_name, salary, percent )
 WHERE percent < 0.05
```

下面的表是查询的结果集。

last_name	salary	percent
Scott	96300.00	0
Sheffield	87900.00	0.025
Lull	87900.00	0.025

报告函数

使用报告函数，您可以比较非集合值与集合值。例如，有些客户所订购产品的数量超过了某产品在指定年份内的平均数量，销售人员可能需要编辑所有这些客户的列表，或者，经理可能需要将雇员的薪水与部门的平均薪水进行比较。

任何集合函数都可以与报告函数联合使用。有关可用集合函数的列表，请参见 『ASA SQL 参考』> 「集合函数」。

注意
您可以使用与窗口函数相同的统计函数 （LIST 除外）。

示例 1

下面的查询返回一个结果集，该结果集显示了销售数量高于平均销售数量的产品的列表。该结果集按年划分。

```
SELECT *
 FROM (SELECT year(order_date) AS Year, prod_id,
         SUM( quantity ) AS Q,
         AVG (SUM(quantity))
         OVER (PARTITION BY Year) AS Average
         FROM sales_order JOIN sales_order_items
         GROUP BY year(order_date), prod_id
         ORDER BY Year)
     AS derived_table
 WHERE Q > Average
```

下面的表是查询的结果集。

Year	prod_id	Q	Average
2000	700	2700	1787.000
2000	601	1932	1787.000
2000	600	2124	1787.000
2000	400	2030	1787.000
2001	700	1836	1048.000
2001	401	1057	1048.000
2001	400	1248	1048.000

对于 2000 年，平均订单数是 1,787。四种产品 （700、 601、 600 和 400）的销售数量高于该数量。在 2001 年，平均订单数是 1,048，有三种产品超过该数量。

示例 2

下面的查询返回一个结果集，该结果集显示了其薪水比其部门的平均薪水高一个标准偏差的雇员。标准偏差是数据与平均值的差异的测量单位。

```
SELECT *
 FROM (SELECT
         emp_lname   AS Employee,
         dept_id     AS Dept,
         CAST( salary as DECIMAL( 10, 2) ) AS Salary,
         CAST( AVG( salary )
             OVER (PARTITION BY dept_id) AS DECIMAL
(10,2) ) AS Average,
         CAST (STDDEV_POP (salary)
           OVER (PARTITION BY dept_id) AS DECIMAL (10,
2 ) ) AS Std_Dev
```

```
        FROM employee
        GROUP BY Dept, Employee, Salary)
    AS derived_table
WHERE Salary > Average + Std_Dev
ORDER BY Dept, Salary, Employee
```

下面的表是查询的结果集。每个部门至少有一名雇员的薪水远远偏离于平均值。

Employee	Dept	Salary	Average	Std_Dev
Lull	100	87900.00	58736.28	16829.60
Sheffield	100	87900.00	58736.28	16829.60
Scott	100	96300.00	58736.28	16829.60
Sterling	200	64900.00	48390.95	13869.60
Savarino	200	72300.00	48390.95	13869.60
Kelly	200	87500.00	48390.95	13869.60
Shea	300	138948.00	59500.00	30752.40
Blaikie	400	54900.00	43640.67	11194.02
Morris	400	61300.00	43640.67	11194.02
Evans	400	68940.00	43640.67	11194.02
Martinez	500	55500.80	33752.20	9084.50

雇员 Scott 的收入为 $96,300.00，而部门平均收入为 $58,736.28。该部门的标准偏差是 16,829.00，这表示低于 $75,565.88 (`58736.28 + 16829.60 = 75565.88`) 的薪水会在平均值的一个标准偏差范围内。雇员 Scott 的 $96,300.00 远远高于该数字。

窗口函数

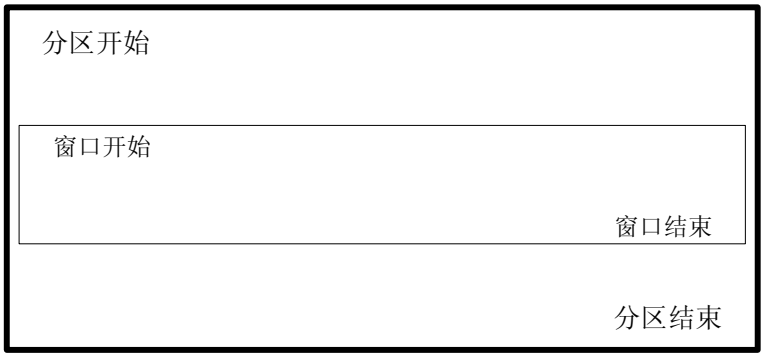
使用窗口函数，您可以通过计算每行周围窗口上的集合值来分析数据。结果集会返回表示一组行的摘要值。您可以使用窗口函数计算一家公司在一段指定时间内的销售数据的移动平均值。

除 LIST 以外的任何集合函数都可以与窗口函数联合使用。

有关可用集合函数的列表，请参见 『ASA SQL 参考』> 「集合函数」。

注意
您可以使用与报告函数相同的统计函数。

下图显示了分区及其包含的窗口视图。该窗口通过一定数量的行定义，并通过该分区向下分布。



示例

下面的示例显示了一个窗口函数。查询返回一个结果集，该结果集按部门划分数据，然后提供员工薪水的累计汇总 （从在公司的时间最长的员工开始）。结果集只包括居住在加利福尼亚、犹他州、纽约或亚利桑那州的那些雇员。 Sum Salary 列提供了雇员薪水的累计总额。

```
SELECT dept_id, emp_lname, start_date, salary,
 SUM(salary) OVER (PARTITION BY dept_id
 ORDER BY start_date
 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
"Sum_Salary"
 FROM employee
 WHERE state IN ('CA', 'UT', 'NY', 'AZ') AND dept_id IN
('100', '200')
 ORDER BY dept_id, start_date;
```

下面的表是查询的结果集。结果集按部门划分。

dept_id	emp_lname	start_date	salary	Sum_Salary
100	Whitney	1984-08-28	45700.00	45700.00
100	Cobb	1985-01-01	62000.00	107700.00
100	Shishov	1986-06-07	72995.00	180695.00
100	Driscoll	1986-07-01	48023.69	228718.69
100	Guevara	1986-10-14	42998.00	271716.69
100	Wang	1988-09-29	68400.00	340116.69
100	Soo	1990-07-31	39075.00	379191.69
100	Diaz	1990-08-19	54900.00	434091.69
200	Overbey	1987-02-19	39300.00	39300.00
200	Martel	1989-10-16	55700.00	95000.00
200	Savarino	1989-11-07	72300.00	167300.00
200	Clark	1990-07-21	45000.00	212300.00
200	Goggin	1990-08-05	37900.00	250200.00

对于部门 100，来自加利福尼亚、犹他州、纽约或亚利桑那州的雇员的薪水累计总额是 $434,091.69，而部门 200 的雇员的薪水累计总额是 $250,200.00。

第 11 章

使用子查询

关于本章

在创建查询时，可使用 WHERE 和 HAVING 子句来限制该查询所返回的行。

有时候，您所选择的行取决于在多个表中存储的信息。通过 WHERE 或 HAVING 子句中的子查询，您可以根据取自其它表的信息从一个表中选择行。在「连接：从多个表检索数据」第 281 页中提供了执行上述操作的其它方法。

事先准备

本章是有关子查询的，介绍了一些有关查询和 select 语句的语法方面的知识。在「查询：从表中选择数据」第 225 页中提供了与查询有关的信息。

子查询简介

关系数据库将有关不同类型的对象的信息存储在不同的表中。例如，您应该将与产品有关的信息存储在一个表中，将与销售订单有关的信息存储在另一个表中。产品表包含与各种产品有关的信息。销售订单项表包含与客户的订单有关的信息。

通常，仅使用一个表只能回答最简单的问题。例如，如果公司在产品库存量不足 50 的情况下重新订购这些产品，则可以使用下面的查询回答 [哪些产品接近库存不足？] 的问题。

```
SELECT id, name, description, quantity
 FROM product
 WHERE quantity < 50
```

但是，如果 [接近库存不足] 取决于典型客户所订购的每一类型的产品的数量，则必须用取自 sales_order_items 表的值来替代数字 "50"。

子查询的结构

子查询的结构类似于常规查询，并且出现在主查询的 SELECT、FROM、 WHERE 或 HAVING 子句中。继续看上一个例子，您可以使用子查询来选择客户订购的产品的平均数，然后在主查询中使用该平均数查找接近库存不足的产品。以下查询查找子查询中产品的名称和说明，这些产品的数目小于客户订购的每一类型的产品的平均数目的两倍。

```
SELECT name, description
 FROM product WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     )
```

在 WHERE 子句中，子查询帮助从 FROM 子句中列出的表中选择那些在查询结果中出现的行。在 HAVING 子句中，子查询帮助选择在查询结果中出现的行组，这些行组是由主查询的 GROUP BY 子句指定的。

在 WHERE 子句中使用子查询

WHERE 子句中的子查询的作用是作为行选择过程的一部分。当被用来选择行的条件取决于其它表的结果时，可以在 WHERE 子句中使用子查询。

示例

查找其库存量小于平均订购量的两倍的产品。

```
SELECT name, description
 FROM product WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

这是一个由两个步骤组成的查询：首先，查找每个订单请求的产品的平均数；然后查找库存数小于该平均数的两倍的产品。

由两个步骤组成的查询

sales_order_items 表的 quantity 列存储每个产品类型、客户和订单请求的产品*数量*。该子查询是

```
SELECT avg(quantity)
 FROM sales_order_items
```

它返回 sales_order_items 表中各产品的平均数量，该数量是 25.851413。

下一个查询返回其库存数量小于以前提取的值的两倍的产品的名称和说明。

```
SELECT name, description
 FROM product
 WHERE quantity < 2*25.851413
```

使用子查询将上述两个步骤合并成一个操作。

在 WHERE 子句中使用子查询的目的

WHERE 子句中的子查询是搜索条件的一部分。「查询：从表中选择数据」第 225 页一章介绍了可以在 WHERE 子句中使用的简单搜索条件。

HAVING 子句中的子查询

尽管您通常在 WHERE 子句中使用子查询作为搜索条件，但有时候也可以在一个查询的 HAVING 子句中使用子查询。当子查询出现在 HAVING 子句中时，像 HAVING 子句中的任何表达式一样，该子查询是作为行组选择的一部分来使用的。

下面的请求很自然地产生了一个在 HAVING 子句中使用子查询的查询：[库存数量中哪些产品的平均数量超过每个客户订购的每一产品的平均数量的两倍？]

示例

```
SELECT name, avg(quantity)
 FROM product
 GROUP BY name
 HAVING avg(quantity) > 2* (
     SELECT avg(quantity)
     FROM sales_order_items
     )
```

name	avg(product.quantity)
Tee Shirt	52.333333
Baseball Cap	62
Shorts	80

该查询的执行过程如下：

- 子查询计算 sales_order_items 表中各产品的平均数量。
- 主查询然后遍历 product 表，计算产品平均数量，并且按产品名分组。
- HAVING 子句然后检查每一平均数量是否超过子查询所找到的数量的两倍。如果超过两倍，则主查询返回该行组；否则，它不返回行组。
- SELECT 子句为每一组生成一个摘要行，并且显示每一产品的名称及其库存平均数量。

如下例所示，您也可以在 HAVING 子句中使用外部引用，但与前一个例子稍有不同。

示例

[查找其平均订购数量超过这些产品的库存数量的一半的那些产品的产品 ID 号和行 ID 号。]

```
SELECT prod_id, line_id
 FROM sales_order_items
 GROUP BY prod_id, line_id
 HAVING 2* avg(quantity) > (
     SELECT quantity
     FROM product
     WHERE product.id = sales_order_items.prod_id)
```

prod_id	line_id
401	2
401	1
401	4
501	3
…	…

在此例中，子查询必须生成与 HAVING 子句所测试的行组相对应的产品的库存数量。子查询使用外部引用 sales_order_items.prod_id 为该特定产品选择记录。

使用比较符号的子查询返回单个值

此查询使用比较符号 ">"，暗示该子查询必须恰好返回一个值。在此例中，它恰好返回了一个值。由于 product 表的 ID 字段是主键，因此，product 表中只有一条记录与任何特定产品 ID 相对应。

子查询测试

「查询：从表中选择数据」第 225 页一章介绍了可以在 HAVING 子句中使用的简单搜索条件。因为子查询只是在 WHERE 或 HAVING 子句中出现的表达式，所以，子查询上的搜索条件可能很相似。

这些子查询测试包括：

- **子查询比较测试** 将表达式的值与子查询为主查询的表中的每一记录生成的单个值进行比较。
- **定量比较测试** 将表达式的值与子查询生成的每一组值进行比较。
- **子查询集合成员资格测试** 检查表达式的值是否与子查询生成的某一组值匹配。
- **存在测试** 检查子查询是否生成一些行。

子查询比较测试

子查询比较测试（=、 <>、 <.<=、 >、 >=）是简单比较测试的修改后的版本。这两者的唯一区别在于：在子查询比较测试中，在运算符之后的表达式是子查询。此测试用于将来自主查询的某行中的值与子查询生成的*单个*值进行比较。

示例

下面的查询包含一个子查询比较测试的示例：

```
SELECT name, description, quantity
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
…	…	…

以下子查询从 sales_order_items 表检索单个值，即每个客户订单的每一类型的产品的平均数量。

```
SELECT avg(quantity)
 FROM sales_order_items
```

然后，主查询将每一库存产品的数量与该值进行比较。

比较测试中的子查询返回一个值

比较测试中的子查询必须恰好返回一个值。请看以下查询，其子查询从 sales_order_items 表中提取两列：

```
SELECT name, description, quantity
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity), max (quantity)
     FROM sales_order_items)
```

它返回错误 [子查询只允许一个选择列表项]。

使用 ANY 和 ALL 的定量比较测试

定量比较测试有两个类别：ALL 测试和 ANY 测试。

ANY 测试

ANY 测试可以和 SQL 比较运算符 （=、 <>、 <、 <=、 >、 >=）一起使用，将单个值与子查询生成的一列数据值进行比较。为了执行该测试， SQL 使用指定的比较运算符来将测试值与该列中的每一数据值进行比较。如果*任何*比较运算生成的结果都为 TRUE，则 ANY 测试将返回 TRUE。

与 ANY 一起使用的子查询必须返回单列。

示例

查找在订单 #2005 的第一批产品已发出之后所发的那些订单的订单 ID 和客户 ID。

```
SELECT id, cust_id
 FROM sales_order
 WHERE order_date > ANY (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2005)
```

id	cust_id
2006	105
2007	106
2008	107
2009	108
…	…

在执行此查询时，主查询根据订单 #2005 的*每一个*产品的发运日期测试每一订单的订单日期。如果订单日期大于订单 #2005 的*一批*货物的发运日期，则来自 sales_order 表的订单 ID 和客户 ID 是结果集的一部分。 ANY 测试因此与 OR 运算符相类似：可以将上述查询理解为 [此

销售订单是否为已发运订单 #2005 的第一批产品之后所下的销售订单，或者是否为已发运订单 #2005 的第二批产品之后所下的销售订单，或者 …]

理解 ANY 运算符

ANY 运算符可能会造成一点混淆。它很容易使人将该查询理解为 [返回在已发运订单 #2005 的任何产品之后所下的那些订单]。但这意味着该查询将返回订单 #2005 的*所有*产品都已发运后所下的订单的订单 ID 和客户 ID，这不是本查询所希望查找的那些产品。

而应将本查询理解为 [对于在 2005 号订单发运了*至少一件*产品后所下的订单，返回它们的订单 ID 和客户 ID]。使用关键字 SOME 可以更直观地表达出该查询。以下查询等效于前一查询。

```
SELECT id, cust_id
 FROM sales_order
 WHERE order_date > SOME (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2005)
```

关键字 SOME 等效于关键字 ANY。

与 ANY 运算符有关的说明

ANY 测试还有两个重要特性：

- **空子查询结果集**　如果子查询生成空结果集，则 ANY 测试返回 FALSE。这是有意义的，因为如果没有任何结果，则至少有一个结果满足比较测试就不成立了。
- **子查询结果集中的空值**　假设在子查询结果集中至少有一个空值。如果比较测试对于结果集中的所有非 NULL 数据值均为假，则 ANY 搜索返回 NULL。原因在于，在此情况下，您无法明确断定其比较测试为真的子查询是否具有值。可能有也可能没有值，具体情况取决于结果集中的空数据的 [正确] 值。

ALL 测试

与 ANY 测试类似， ALL 测试与 6 个 SQL 比较运算符 （=、 <>、 <、 <=、 >、 >=）中的一个结合使用，将单个值与子查询生成的数据值进行比较。为了执行该测试， SQL 使用指定的比较运算符将测试值与结果集中的每一数据值进行比较。如果*所有*比较运算生成的结果都为 TRUE，则 ALL 测试就返回 TRUE。

示例

以下是很自然使用 ALL 测试处理的请求：[查找在订单 #2001 的所有产品都已发运之后所下的那些订单的订单 ID 和客户 ID]。

```
SELECT id, cust_id
 FROM sales_order
 WHERE order_date > ALL (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2001)
```

id	cust_id
2002	102
2003	103
2004	104
2005	101
…	…

在执行此查询时，主查询根据订单 #2001 的*每一个*产品的发运日期测试每一订单的订单日期。如果订单日期大于订单 #2001 的*每批*货物的发运日期，则来自 sales_order 表的订单 ID 和客户 ID 是结果集的一部分。 ALL 测试因此与 AND 运算符相类似，可以将上述查询理解为 [此销售订单是否为在已发运订单 #2001 的第一个产品之后所下的销售订单，并且此销售订单是否为已发运订单 #2001 的第二个产品之后所下的销售订单，并且 …]

有关 ALL 运算符的说明

ALL 测试还有三个重要特性：

- **空子查询结果集**　如果子查询生成空结果集，则 ALL 测试返回 TRUE。这是有意义的，因为，如果没有结果，则对结果集中的每一个值的比较测试仍为真。
- **子查询结果集中的空值**　如果比较测试对于结果集中的任一值为假，则 ALL 搜索返回 FALSE。如果所有值均为真，则它返回 TRUE。否则，它返回 UNKNOWN （未知）。例如，如果在子查询结果集中有空值，但搜索条件对于所有非空值均为 TRUE，则可能会发生这种情况。
- **对 ALL 测试取非**　以下表达式*不是*等效的。

```
NOT a = ALL (subquery)
 a <> ALL (subquery)
```

有关此测试的详细信息，请参见 「定量比较测试」第 408 页。

用 IN 条件测试集合成员资格

您可以使用子查询集合成员资格测试将来自主查询的值与子查询中的多个值进行比较。

子查询集合成员资格测试将主查询中的每一行的单个数据值与该子查询生成的单一列中的数据值进行比较。如果来自主查询的数据值与该列中的*某*一数据值匹配，则该子查询返回 TRUE。

示例

选择领导运输部门或财务部门的雇员的姓名：

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

emp_fname	emp_lname
Jose	Martinez
Mary Anne	Shea

此示例中的子查询

```
SELECT dept_head_id
 FROM department
 WHERE (dept_name='Finance' OR dept_name = 'Shipping')
```

从 department 表提取对应于运输部门和财务部门领导的 ID 号。主查询然后返回其 ID 号与该子查询找到的两个 ID 号中的一个相匹配的雇员的姓名。

集合成员资格测试等效于 =ANY 测试

子查询集合成员资格测试等效于 =ANY 测试。以下查询等效于上一例子中的查询。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id =ANY (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

集合成员资格测试的取非

您还可以使用子查询集合成员资格测试来提取其列值不等于由子查询生成的任何值的那些行。为了对集合成员资格测试取非，应在关键字 IN 之前插入 NOT 一词。

示例

下面的查询中的子查询返回不是财务部门或运输部门领导的雇员的名和姓。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id NOT IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

存在测试

在子查询比较测试中使用的子查询和在集合成员资格测试中使用的子查询都从子查询表返回数据值。不过，有时候您可能更关心子查询是否返回*一些*结果，而不是返回*哪些*结果。存在测试 (EXISTS) 就是用来检查子查询是否生成一些查询结果行的。如果子查询生成一行或多行结果，则 EXISTS 测试就返回 TRUE。否则，它就返回 FALSE。

示例

下面是一个使用子查询表示的请求：[哪些客户在 2001 年 7 月 13 日以后下了订单？]

```
SELECT fname, lname
 FROM customer
 WHERE EXISTS (
     SELECT *
     FROM sales_order
     WHERE (order_date > '2001-07-13') AND
      (customer.id = sales_order.cust_id))
```

fname	lname
Almen	de Joie
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

有关存在测试的说明

在这里，对于 customer 表中的每一行，该子查询检查客户 ID 是否与在 2001 年 7 月 13 日之后下了订单的某位客户的 ID 相对应。如果相对应，则该查询将从主表提取该客户的名和姓。

EXISTS 测试不使用子查询的结果，它只检查该子查询是否生成一些行。因此，应用于以下两个子查询的存在测试返回相同的结果。它们是子查询并且无法自行处理，因为这两个子查询引用属于主查询 （但不属于子查询）的客户表。

有关详细信息，请参见 「相关子查询」第 405 页。

```
SELECT *
 FROM sales_order
 WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id)
 SELECT ship_date
 FROM sales_order
 WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id)
```

sales_order 表中的哪些列出现在 SELECT 语句中并无多大关系，不过，按照惯例，使用 "SELECT *" 表示法。

对存在测试取非

您可以采用 NOT EXISTS 的形式颠倒 EXISTS 测试的逻辑。在此情况下，如果子查询没有生成任何行，则该测试返回 TRUE，否则返回 FALSE。

相关子查询

您可能已经注意到，子查询中包含对 customer 表上的 id 列的引用。对主表中的列或表达式的引用被称作**外部引用**，并且这种子查询是**相关的**。在概念上， SQL 处理上述查询的方式是遍历 customer 表并为每一客户执行子查询。如果 sales_order 表中的订单日期晚于 2001 年 7 月 13 日，并且 customer 表和 sales_order 表中的客户 ID 相匹配，则 customer 表中的名和姓就会显示出来。因为子查询引用主查询，所以，本节中的子查询与前几节中的那些子查询不同，如果您尝试由它本身来运行，将会返回错误。

外部引用

在子查询的正文中，通常需要引用主查询的活动行中的列的值。请看以下查询：

```
SELECT name, description
 FROM product
 WHERE quantity < 2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     WHERE product.id = sales_order_items.prod_id)
```

此查询提取其库存数量少于该产品的平均订购数量两倍的产品 （具体而言，即由主查询中的 WHERE 子句测试的产品）的名称和说明。此子查询通过扫描 sales_order_items 表执行此任务。但在子查询的 WHERE 子句中， product.id 列引用*主*查询 （而不是子查询）的 FROM 子句中指定的表中的列。由于 SQL 遍历 product 表中的每一行，因此，它在对子查询的 WHERE 子句求值时使用当前行的 id 值。

有关外部引用的说明

此子查询中的 product.id 列是外部引用的一个例子。使用外部引用的子查询称为相关子查询。外部引用是一个列名称，该列名称不引用该子查询的 FROM 子句中的任何表的任何列，而是引用主查询的 FROM 子句指定的表的列。如上例所示，外部引用中的列的值来自主查询当前测试的行。

子查询和连接

子查询优化程序会自动将使用子查询的多个查询改写为连接。

示例

请看以下请求：[Clarke 女士和 Suresh 女士何时下的订单，并且是向哪些销售代表下的订单？] 可以用以下查询回答上述请求：

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id IN (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

Order_date	sales_rep
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
…	…

该子查询生成与其姓名在 WHERE 子句中列出的两个客户相对应的客户 ID 的列表，并且主查询查找与这两位女士的订单相对应的订单日期和销售代表。

用连接替代子查询

可以使用连接回答同样的问题。下面是该查询的使用了两表式连接的替代形式：

```
SELECT order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND
   (lname = 'Clarke' OR fname = 'Suresh')
```

此查询形式将 sales_order 表连接到 customer 表，以查找每一客户的订单，然后只返回 Suresh 和 Clarke 的那些记录。

某些连接无法被改写为子查询

这两个查询都会找到正确的订单日期和销售代表，并且都同样正确。许多人会认为子查询形式更为自然，因为该请求没有询问与客户 ID 有关的任何信息，并且因为将 sales_order 和 customer 表连接在一起来回答该问题看起来似乎有一些奇怪。

但是，如果将该请求更改为包括来自 customer 表的某些信息，则该子查询形式将不再有效。例如，如果请求是 [Clarke 女士和 Suresh 女士是何时下的订单、向哪些销售代表下的订单以及这两位女士的全名是什么？]，则需要在主 WHERE 子句中包括 customer 表：

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = 'Suresh')
```

fname	lname	order_date	sales_rep
Belinda	Clarke	1/5/01	1596
Belinda	Clarke	1/27/00	667
Belinda	Clarke	11/11/00	467
Belinda	Clarke	2/4/01	195
…	…	…	…

某些子查询无法被改写为连接

同样，在某些情况下，可以使用子查询，但无法使用连接。例如：

```
SELECT name, description, quantity
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
…	…	…

在此例中，内部查询是汇总查询，而外部查询不是汇总查询，因此，无法通过一个简单连接来合并这两个查询。

有关连接的详细信息，请参见 「连接：从多个表检索数据」第 281 页。

嵌套的子查询

正如我们所看到的，子查询始终出现在查询的 HAVING 子句或 WHERE 子句中。子查询自身也可以包含 WHERE 子句和 / 或 HAVING 子句，同样，子查询也可以出现在其它子查询中。位于其它子查询内的子查询被称为**嵌套的子查询**。

示例

如果在某一天有人在收费部门中订购了产品，并且同一天发运了一些订单，则请列出这些订单的订单 ID 和行 ID。

```
SELECT id, line_id
 FROM sales_order_items
 WHERE ship_date = ANY (
     SELECT order_date
     FROM sales_order
     WHERE fin_code_id IN (
         SELECT code
         FROM fin_code
         WHERE (description = 'Fees')))
```

id	line_id
2001	1
2001	2
2001	3
2002	1
…	…

有关嵌套子查询的说明

- 在此示例中，最里面的子查询生成一列其说明是 "Fees" 的财务代码。

```
SELECT code
 FROM fin_code
 WHERE (description = 'Fees')
```

- 下一个子查询查找其代码与最里面的子查询中选择的某一代码相匹配的产品的订单日期。

```
SELECT order_date
 FROM sales_order
 WHERE fin_code_id IN (subquery)
```

- 最后，最外面的查询查找在此子查询中找到的某一日期发运的订单的订单 ID 和行 ID。

```
SELECT id, line_id
 FROM sales_order_items
 WHERE ship_date = ANY (subquery)
```

嵌套的子查询还可以有三层以上的嵌套层次。尽管对嵌套的最大层次数没有限制，但与嵌套层次较少的查询相比，嵌套层次在三层或三层以上的查询的运行时间要长得多。

子查询的工作方式

当查询包含子查询时，要想弄清哪些查询有效、哪些查询无效可能是一件困难的事。同样，要想弄清楚多层查询的作用也可能非常棘手，不过，这样会有助于理解 Adaptive Server Anywhere 是如何处理子查询的。有关处理查询的一般信息，请参见 「对查询结果进行汇总、分组和排序」第 251 页。

相关子查询

在简单查询中，对于查询的每一行，数据库服务器计算并处理该查询的 WHERE 子句一次。不过，有时候子查询只返回一个结果，因此数据库服务器不必为获得整个结果集对其计算多次。

不相关子查询

请看以下查询：

```
SELECT name, description
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

在此示例中，子查询只计算一个值：sales_order_items 表的平均数量。在计算该查询时，数据库服务器计算该值一次，并且将 product 表的 quantity 字段中的每个值与该值进行比较，以确定是否选择相应的行。

相关子查询

当子查询包含外部引用时，您就不能使用此简便方式。例如，以下查询

```
SELECT name, description
 FROM product
 WHERE quantity < 2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     WHERE product.id=sales_order_items.prod_id)
```

根据 product 表中的活动行返回一个值。此类子查询被称作相关子查询。在这些例子中，子查询可为外部查询的每一行返回不同的值，因此数据库服务器需要执行多次计算。

将 WHERE 子句中的子查询转换为连接

Adaptive Server Anywhere 查询优化程序会对一些多层查询进行转换，以使用连接。执行这一转换不需要任何用户操作。本节将介绍哪些子查询可转换为连接，以便您可以了解数据库中的查询的性能。

示例

问题 [Clarke 女士和 Suresh 女士何时下的订单，并且是向哪些销售代表下的订单？] 可表示为两层查询：

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id IN (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

或者，可以采用同样正确的方法来表示为使用连接的查询：

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND
   (lname = 'Clarke' OR fname = 'Suresh')
```

要将多层查询改写为连接必须满足一定的条件，运算符的类型不同，这些条件也将有所不同。请回忆一下子查询出现在查询的 WHERE 子句中的情况，其形式如下

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery* **)**
| [**NOT**] *expression comparison-operator* { **ANY | SOME** } **(** *subquery* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery* **)**
| [**NOT**] *expression* **IN (** *subquery* **)**
| [**NOT**] **EXISTS (** *subquery* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

子查询是否可转换为连接取决于多种因素，例如运算符的类型、查询的结构和子查询的结构等。

比较运算符

如果要将跟随在比较运算符 （=、 <>、 <、 <=、 >、 >=）后面的子查询转换为连接，则这样的子查询必须满足某些条件。只有当跟随在比较运算符后面的子查询为主查询的每一行恰好返回一个值时，它们才是有效的。除了这一条件外，只有在子查询满足以下条件后才可以将该子查询转换为连接

- 不包含 GROUP BY 子句
- 不包含关键字 DISTINCT
- 不是 UNION 查询
- 不是汇总查询

示例

假设 [Suresh 的产品是何时订购的，并且是向哪一个销售代表下的订单？] 这一请求被表示为以下子查询

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id = (
     SELECT id
     FROM customer
     WHERE fname = 'Suresh')
```

该查询满足上述条件，因此可转换为使用连接的查询：

```
SELECT order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = 'Suresh')
```

但是， [查找其库存量小于平均订购量的两倍的产品] 这一请求不能被转换为连接，因为该子查询包含集合函数 avg：

```
SELECT name, description
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

定量比较测试

跟在关键字 ALL、 ANY 和 SOME 之后的子查询只有满足以下条件后，它们才可转换为连接。

- 主查询不包含 GROUP BY 子句并且不是汇总查询，或者子查询恰好返回一个值。
- 子查询不包含 GROUP BY 子句。
- 子查询不包含关键字 DISTINCT。
- 子查询不是 UNION 查询。
- 子查询不是汇总查询。
- 不可对 [表达式比较运算符 ANY/SOME （子查询）] 取非。
- 必须对 [表达式比较运算符 ALL （子查询）] 取非。

其中的前 4 个条件相对易于理解。

示例

[Clarke 女士和 Suresh 女士何时下的订单，并且是向哪些销售代表下的订单？] 这一请求可按以下子查询形式进行处理：

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id = ANY (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

或者，还可以用连接形式表示该查询

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = 'Suresh')
```

但是， [Clarke 女士和 Suresh 女士以及也是客户的任何雇员何时下的订单？] 应表示为联合查询，并因此无法转换为连接：

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id = ANY (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh'
     UNION
     SELECT emp_id
     FROM employee)
```

同样，[查找不是在订单 #2001 的所有产品都已发运后所下的那些订单的订单 ID 和客户 ID] 可以自然而然地用以下子查询来表示：

```
SELECT id, cust_id
 FROM sales_order
 WHERE NOT order_date > ALL (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2001)
```

该子查询可转换为以下连接：

```
SELECT sales_order.id, cust_id
 FROM sales_order, sales_order_items
 WHERE (sales_order_items.id=2001) and (order_date <=
ship_date)
```

但是，[查找在所有产品的第一个发运日期后未发运的那些订单的订单 ID 和客户 ID] 将被表示为汇总查询：

```
SELECT id, cust_id
 FROM sales_order
 WHERE NOT order_date > ALL (
     SELECT first (ship_date)
     FROM sales_order_items )
```

因此，它将不会被转换为连接。

对具有 ANY 和 ALL 运算符的子查询取非

第 5 个条件有点令人糊涂：以下形式的查询

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL(** *subquery* **)**
are converted to joins, *as are queries of the form*

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY(** *subquery* **)**

但查询

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL(** *subquery* **)**

和

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY(** *subquery* **)**

不可以转换为连接。

ANY 和 ALL 表达式的逻辑等效性

这是因为前两个查询实际上是等效的，后两个查询也是这样。请回忆一下， any 运算符类似于 OR 运算符，但参数数量是可变的；而同样，ALL 运算符也类似于 AND 运算符。正如表达式

```
NOT ((X > A) AND (X > B))
```

等效于以下表达式

```
(X <= A) OR (X <= B)
```

表达式

```
NOT order_date > ALL (
     SELECT first (ship_date)
     FROM sales_order_items )
```

等效于以下表达式

```
order_date <= ANY (
     SELECT first (ship_date)
     FROM sales_order_items )
```

对 ANY 和 ALL 表达式取非

一般而言，表达式

NOT *column-name operator* **ANY(** *subquery* **)**

等效于以下表达式

column-name inverse-operator **ALL(** *subquery* **)**

而表达式

NOT *column-name operator* **ALL(** *subquery* **)**

等效于以下表达式

column-name inverse-operator **ANY(** *subquery* **)**

其中 *inverse-operator* 是通过对 *operator* 取非而获得的，如下表所示：

运算符及其反向运算符表

下表列出了每一运算符的反向运算符。

运算符	反向运算符
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

集合成员资格测试

只有在满足以下条件后，其子查询跟随在关键字 IN 后的查询才可转换为连接：

- 主查询不包含 GROUP BY 子句并且不是汇总查询，或者子查询恰好返回一个值。
- 子查询不包含 GROUP BY 子句。
- 子查询不包含关键字 DISTINCT。
- 子查询不是 UNION 查询。
- 子查询不是汇总查询。

- 不得对 [表达式 IN （子查询）] 取非。

示例

因此，由以下查询表示的 [查找还是部门领导的雇员的姓名]：

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

将被转换为具有连接的查询，因为它满足上述条件。但是，如果用 UNION 查询表示 [查找既是部门领导又是客户的雇员的姓名]，则该请求将不会被转换为连接。

无法转换跟随在 IN 运算符之后的 UNION 查询

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping')
     UNION
     SELECT cust_id
     FROM sales_order)
```

同样， [查找不是部门领导的雇员的姓名] 用以下取非的子查询来表示

```
SELECT emp_fname, emp_lname
 FROM employee
   WHERE NOT emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

并且不会被转换为连接。

对于跟随在 IN 关键字和 ANY 关键字之后的子查询，将它们转换为连接所必须满足的条件是相同的。这并非巧合，而是因为表达式

具有 IN 运算符的查询可转换为具有 ANY 运算符的查询

WHERE *column-name* **IN(** *subquery* **)**

在逻辑上等效于以下表达式

WHERE *column-name* = **ANY(** *subquery* **)**

因此查询

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

等效于以下查询

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id = ANY (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

从概念上讲， Adaptive Server Anywhere 将具有 IN 运算符的查询转换为具有 ANY 运算符的查询，并据此决定是否将子查询转换为连接。

存在测试

只有在满足以下两个条件后，跟随在关键字 EXISTS 之后的子查询才可转换为连接：

- 主查询不包含 GROUP BY 子句并且不是汇总查询，或者子查询恰好返回一个值。
- 联合 [EXISTS （子查询）] 不被取非。
- 该子查询是相关的；即，它包含外部引用。

示例

[哪些客户在 2001 年 7 月 13 日之后下了订单] 的请求可使用此类查询（其未取非的子查询包含外部引用 **customer.id** = **sales_order.cust_id**）以公式的形式表示出来，因此，可转换为连接。

```
SELECT fname, lname
 FROM customer
 WHERE EXISTS (
     SELECT *
     FROM sales_order
     WHERE (order_date > '2001-07-13')
  AND (customer.id = sales_order.cust_id))
```

EXISTS 关键字主要用来告诉数据库服务器检查空结果集。在使用内连接时，数据库服务器自动只显示这样的行：这些行的数据来自 FROM 子句中的所有表。因此，该查询返回的行与具有以下子查询的查询所返回的行相同：

```
SELECT fname, lname
 FROM customer, sales_order
 WHERE (sales_order.order_date > '2001-07-13')
 AND (customer.id = sales_order.cust_id)
```

第 12 章

添加、更改和删除数据

关于本章

本章介绍如何修改数据库中的数据。

本章的大部分内容介绍 INSERT、 UPDATE 和 DELETE 语句，同时也对用于批量装载和批量卸载的语句进行了说明。

数据修改语句

用于添加、更改或删除数据的语句称为**数据修改**语句。最常见的数据修改语句包括：

- **Insert** 向表中添加新行
- **Update** 更改表中现有的行
- **Delete** 从表中删除特定的行

任何单个 INSERT、 UPDATE 或 DELETE 语句都只更改一个表或视图中的数据。

除了这些常见语句之外，如果要批量装载和删除数据，则 LOAD TABLE 和 TRUNCATE TABLE 语句将十分有用。

有时，这些数据修改语句被一并称为 SQL 的**数据修改语言** (DML) 部分。

数据修改的权限

只有在要修改的数据库表上具有适当的权限之后，您才可以执行数据修改语句。数据库管理员和数据库对象的所有者使用 GRANT 和 REVOKE 语句来决定哪些用户具有哪些数据修改功能的权限。

可以将权限授予个别用户、组或公共组。有关权限的详细信息，请参见『ASA 数据库管理指南』> 「管理用户 ID 和权限」。

事务和数据修改

修改数据时，事务日志存储每个数据修改语句影响的每行的旧状态和新状态的副本。这意味着如果您开始一个事务，意识到犯了一个错误，于是将事务回退，您还可以将数据库恢复到它以前的状态。

有关事务的详细信息，请参见 「使用事务和隔离级别」第 111 页。

使用 INSERT 添加数据

您可以使用 INSERT 语句向数据库中添加行。 INSERT 语句具有两种格式：您可以使用 VALUES 关键字或 SELECT 语句：

使用 VALUES 的 INSERT

VALUES 关键字指定新行中某些列或所有列的值。使用 VALUES 关键字的 INSERT 语句语法的简化版本如下：

INSERT [**INTO**] *table-name* [(*column-name*, …)]
    VALUES (*expression* , …)

如果您为表中的每列都提供一个值，并且所提供值的顺序是使用 SELECT * 执行查询时它们显示的顺序，则可以省略列名列表。

通过 SELECT 执行 INSERT

您可以在 INSERT 语句中使用 SELECT 语句从一个或多个表中选取值。如果您正向其中插入数据的表有大量的列，您还可以使用 WITH AUTO NAME 以简化语法。使用 WITH AUTO NAME 时，您只需在 SELECT 语句中而不必在 INSERT 和 SELECT 语句中指定列名。 SELECT 语句中的名称必须是列引用或带别名的表达式。

使用 SELECT 语句的 INSERT 语句语法的简化版本如下：

INSERT [**INTO**] *table-name*
    WITH AUTO NAME *select-statement*

有关 INSERT 语句的详细信息，请参见 『ASA SQL 参考』> 「INSERT 语句」。

将值插入行的所有列

以下 INSERT 语句将一个新行添加到 department 表中，为该行中的每列都提供一个值：

```
INSERT INTO department
 VALUES ( 702, 'Eastern Sales', 902 )
```

注意

- 键入值的顺序应该和原始 CREATE TABLE 语句中列名的顺序相同，即第一个值是 ID 编号，然后是名称，最后是部门主管 ID。
- 将值用括号括起来。

- 将所有字符数据括在单引号中。
- 对每个要添加的行使用单独的 INSERT 语句。

将值插入特定的列

您可以通过只指定某些列和它们的值将数据添加到行中的这些列。不包括在列列表中的所有其它列的定义必须允许使用空值或具有缺省值。如果您跳过某个具有缺省值的列，该列为缺省值。

例如，只在两列 （dept_id 和 dept_name）中添加数据需要如下语句：

```
INSERT INTO department (dept_id, dept_name)
 VALUES ( 703, 'Western Sales' )
```

dept_head_id 列没有缺省值，但它允许使用空值。将为该列指派一个空值。

您列出列名的顺序必须和列出值的顺序一致。以下示例和上一个示例产生相同的结果：

```
INSERT INTO department (dept_name, dept_id )
 VALUES ('Western Sales', 703)
```

未指定的列的值

如果只为行中的某些列指定值，则未指定值的列可能发生以下四种情况之一：

- **输入空值** 如果该列允许使用空值并且不存在缺省值，则将为空值。
- **输入缺省值** 如果该列存在缺省值，则为缺省值。
- **输入唯一的、具有顺序的值** 如果该列具有 AUTOINCREMENT 缺省值或 IDENTITY 属性，则将为唯一的序列值。
- **INSERT 被拒绝，并显示一条错误消息** 如果该列既不允许使用空值也不存在缺省值，则将显示一条错误消息。

缺省情况下，除非您在创建表时在列定义中显式规定 NOT NULL，否则列允许使用空值。您可以使用 ALLOW_NULLS_BY_DEFAULT 选项变更缺省设置。

使用约束限制列数据

您可以为列或域创建约束。约束控制您可以添加或不能添加的数据类型。

有关约束的详细信息，请参见 「使用表和列约束」第 96 页。

显式插入空值

您可以通过键入 NULL 将空值显式插入列中。不要将 NULL 括在引号中，否则它将被当做一个字符串。

例如，以下语句将 NULL 显式插入 dept_head_id 列中：

```
INSERT INTO department
 VALUES (703, 'Western Sales', NULL )
```

使用缺省值提供值

您可以对某个列进行定义，以便即使该列未接收到任何值，也会在插入行时自动填上缺省值。可以通过为该列提供缺省值来完成此任务。

有关缺省值的详细信息，请参见 「使用列缺省值」第 90 页。

使用 SELECT 添加新行

若要将值从一个或多个表中取出并添加到另一个表中，您可以在 INSERT 语句中使用 SELECT 子句。 SELECT 子句可以将值插入行中的某些或全部列中。

从现有的表中取值可以很方便地为某些列插入值。然后，您可以使用 UPDATE 为其它列添加值。

在为表中的某些列 （但不是全部列）插入值之前，确保存在缺省值或为不插入值的列指定空值。否则，将出现错误。

当将行从一个表插入到另一个表时，两个表必须具有兼容的结构，也就是说，匹配的列必须是相同的数据类型或者 Adaptive Server 可以自动转换的数据类型。

示例

如果列的顺序与 CREATE TABLE 语句中列的顺序相同，则您不需要在任何一个表中指定列名。假设您有一个名为 newproduct 的表，该表以与 product 表相同的格式包含一些产品信息行。若要将 newproduct 中所有行添加到 product 表：

```
INSERT product
 SELECT *
 FROM newproduct
```

您可以在 INSERT 语句中使用 SELECT 语句表达式。

将数据插入某些列中

就像使用 VALUES 子句一样，您可以使用 SELECT 语句将数据添加到行中的某些列（但不是所有列）。只要在 INSERT 子句中指定要向其中添加数据的列即可。

从相同的表中插入数据

您可以根据相同表中的其它数据将数据插入表中。实质上，这意味着复制全部或部分行。

例如，您可以根据现有产品将新产品插入 product 表中。以下语句将新的特大号 T 恤衫（短背心、 V 型领和水手领等样式）添加到 product 表中。标识号比现有大小的衬衫大 10：

```
INSERT INTO product
 SELECT id+ 10, name, description,
     'Extra large', color, 50, unit_price
 FROM product
 WHERE name = 'Tee Shirt'
```

插入文档和图像

如果要在数据库的 LONG BINARY 列中存储文档或图像，您可以编写一个应用程序，该应用程序将文件的内容读入一个变量中，然后提供该变量作为 INSERT 语句的值。

有关向应用程序添加 INSERT 语句的详细信息，请参见 『ASA 编程指南』> 「如何使用预准备语句」和 『ASA SQL 参考』> 「SET 语句」。

您还可以使用 xp_read_file 系统函数将文件内容插入表中。如果要从 Interactive SQL （或者不提供全部编程语言的某些其它环境）插入文件内容，该函数将非常有用。

使用此外部函数需要 DBA 权限。

示例

在本示例中，您创建一个表，然后将一个图像插入该表的某个列中。您可以从 Interactive SQL 执行这些步骤。

1. 创建一个表以保存某些图像。

```
CREATE TABLE pictures
 ( c1 INT DEFAULT AUTOINCREMENT PRIMARY KEY,
     filename VARCHAR(254),
     picture LONG BINARY )
```

2. 在数据库服务器的当前工作目录中，将 *portrait.gif* 的内容插入表中。

```
INSERT INTO pictures (filename, picture)
 VALUES (     'portrait.gif',
     xp_read_file( 'portrait.gif' ) )
```

有关详细信息，请参见 『ASA SQL 参考』> 「xp_read_file 系统过程」。

使用 UPDATE 更改数据

您可以使用 UPDATE 语句 （后接表或视图的名称）更改表中的单个行、一组行或所有行。像在所有数据修改语句中一样，您一次只能更改一个表或视图中的数据。

UPDATE 语句指定您要更改的行和新数据。新数据可以是一个常量、指定的表达式或从其它表取出的数据。

如果 UPDATE 语句违反了完整性约束，则不会进行更新并将显示一条错误消息。例如，如果正被添加的一个值是错误的数据类型，或者如果它违反为所涉及的某个列或数据类型定义的约束，则将不会进行更新。

UPDATE 语法

UPDATE 语法的简化版本如下：

UPDATE *table-name*
    SET *column_name = expression*
    WHERE *search-condition*

如果 Newton Ent. 公司 （在示例数据库的 customer 表中）被 Einstein, Inc. 接管，则您可以使用如下语句更新该公司的名称：

```
UPDATE customer
 SET company_name = 'Einstein, Inc.'
 WHERE company_name = 'Newton Ent.'
```

您可以在 WHERE 子句中使用任何表达式。如果不确定公司名称是如何拼写的，则您可以使用如下所示的语句尝试更新名为 Newton 的任何公司：

```
UPDATE customer
 SET company_name = 'Einstein, Inc.'
 WHERE company_name LIKE 'Newton%'
```

搜索条件不需要引用被更新的列。 Newton Entertainments 的公司 ID 是 109。由于 ID 值是表的主键，所以您可以确定使用以下语句更新正确的行：

```
UPDATE customer
 SET company_name = 'Einstein, Inc.'
 WHERE id = 109
```

SET 子句

SET 子句指定将被更新的列和它们的新值。 WHERE 子句确定将被更新的行。如果不具有 WHERE 子句，则将用 SET 子句中提供的值更新所有行中指定的列。

您可以在 SET 子句中提供数据类型正确的任何表达式。

WHERE 子句

WHERE 子句指定将被更新的行。例如，以下语句用 "Extra Large Tee Shirt" 替换 "One Size Fits All Tee Shirt"。

```
UPDATE product
 SET size  = 'Extra Large'
 WHERE name = 'Tee Shirt'
    AND size = 'One Size Fits All'
```

FROM 子句

您可以使用 FROM 子句从一个或多个表中取出数据并将它们放入正在更新的表中。

使用 INSERT 更改数据

您可以使用 INSERT 语句的 ON EXISTING 子句用新值更新表中现有的行 （根据主键查寻）。此子句仅可用于拥有主键的表。如果尝试对没有主键的表或代理表使用此子句，则将造成语法错误。

指定 ON EXISTING 子句会使服务器查寻每个输入行的主键。如果相应的行不存在，则会插入新行。对于表中已经存在的行，您可以选择：

- 为重复的键值生成错误。如果不指定 ON EXISTING 子句，此行为是缺省行为。
- 在没有任何提示的情况下忽略输入行，不生成任何错误。
- 用输入行中的值更新现有的行

有关详细信息，请参见 『ASA SQL 参考』> 「INSERT 语句」。

使用 DELETE 删除数据

简单的 DELETE 语句格式如下：

DELETE [**FROM**] *table-name*
　WHERE *column-name = expression*

您也可以使用更复杂的格式，如下所示

DELETE [**FROM**] *table-name*
　FROM *table-list*
　WHERE *search-condition*

WHERE 子句

使用 WHERE 子句指定要删除的行。如果未出现任何 WHERE 子句，则 DELETE 语句将删除表中所有的行。

FROM 子句

DELETE 语句第二个位置上的 FROM 子句是一个特殊的功能，允许您从一个或多个表中选择数据，然后从指定的第一个表中删除相应的数据。您在 FROM 子句中选择的行指定删除的条件。

示例

本示例使用示例数据库。为了执行示例中的语句，您应该将选项 WAIT_FOR_COMMIT 设置为 OFF。以下语句只为当前连接执行此操作：

```
SET TEMPORARY OPTION WAIT_FOR_COMMIT = 'OFF'
```

这允许您删除某些行，即使这些行包含被外键引用的主键，但是除非也删除相应的外键，否则不允许 COMMIT （提交）。

以下视图显示产品和已售出的产品的价值：

```
CREATE VIEW ProductPopularity as
 SELECT  product.id,
     SUM(product.unit_price *
sales_order_items.quantity) as "Value Sold"
 FROM  product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
 GROUP BY product.id
```

使用此视图，您可以从 product 表中删除销售额小于 $20,000 的那些产品。

```
DELETE
 FROM product
 FROM product NATURAL JOIN ProductPopularity
 WHERE "Value Sold" < 20000
```

完成了该示例后，您应该回退您的更改：

```
ROLLBACK
```

从表中删除所有行

您可以使用 TRUNCATE TABLE 语句作为快速删除表中所有行的方法。这样要比不带任何条件的 DELETE 语句更快，因为 DELETE 会记录每个更改，而事务日志不会逐个记录截断表操作。

除非您执行 DROP TABLE 语句，否则使用 TRUNCATE TABLE 语句清空的表的表定义仍将保留在数据库中，同时保留的还有其索引和其它关联的对象。

如果另一个表的行通过参照完整性约束引用该表，则您不能使用 TRUNCATE TABLE。请从外表中删除这些行，或截断外表然后再截断主表。

TRUNCATE TABLE 语法

TRUNCATE TABLE 的语法如下：

TRUNCATE TABLE *table-name*

例如，要删除 sales_order 表中的所有数据，请键入以下语句：

```
TRUNCATE TABLE sales_order
```

TRUNCATE TABLE 语句不触发在该表上定义的触发器。

第 13 章

查询优化与执行

关于本章

当对每个查询进行完语法分析之后，优化程序都将对该查询进行分析并决定一个访问计划，以便使用尽可能少的资源来计算结果。本章将介绍优化程序在优化查询时所经历的步骤。还将介绍作为优化程序设计基础的前提，并讨论优化过程中的选择性预计、成本预计以及其它步骤。

虽然也必须对 update、 insert 和 delete 语句进行优化，但本章将着重介绍 SELECT 查询的优化。对其它这些命令的优化遵循类似的原则。

优化程序的作用

优化程序的作用是设计一种有效的方式来执行 SQL 语句。优化程序以**访问计划**的形式来表示它所选择的方法。访问计划将描述要扫描哪些表、每个表使用哪些索引 （如果有）、将使用哪种连接策略以及按什么顺序读取表。通常，有大量的计划都能达到相同的目标。其它变量可能会进一步扩大可能的访问计划的数量。

基于成本的优化

优化程序会使用高效的算法 （而且在某些情况下还是专有的算法）来开始选择可用的做法。它将根据各个查询所需资源的预计值来做出决定。优化程序既会考虑磁盘访问操作的成本，也会考虑各项操作所需要的预计 CPU 成本。

独立于语法的优化

大多数命令都可以使用 SQL 语言以不同的形式来表示。这些表示形式在语义上等效 （因为它们完成相同的任务），但在语法上却具有明显的差异。 Adaptive Server Anywhere 优化程序仅根据每个语句的语义设计出合适的访问计划，几乎毫无例外。

语义差异虽然看起来可能很明显，但通常不会造成任何影响。例如，谓语、表和属性在查询语法中的顺序的差异并不会影响访问计划的选择。而优化程序也不会因为查询是否包含视图而受到影响。

合适的计划，并不一定是最佳的计划

优化程序的目标是找出合适的访问计划。在理想情况下，优化程序将尽可能地找出最为高效的访问计划，但这一目标通常是不切实际的。对于复杂的查询，可能会存在大量的可能性。

无论优化程序具有多高的效率，如果对每一种选择都进行分析，则会消耗一定的时间和资源。优化程序会将进一步优化所需的成本和执行到当前为止所找到的最佳计划所需的成本进行比较。如果优化程序已设计出了成本较低的计划，就会停止下来执行该计划。与执行已找到的访问计划相比，进一步的优化可能会消耗更多的资源。

如果查询非常复杂并且占用大量的资源，优化程序就可能要运行较长的时间。如果查询占用非常多的资源，则优化程序可能会运行很长的时间，从而导致出现明显的延迟。

优化程序的工作原理

Adaptive Server Anywhere 优化程序必须决定按什么顺序访问查询中的表以及是否为各个表使用索引。优化程序会尽量选取最佳的策略。

执行各个查询的最佳策略是，在最短的时间内以最低的成本取得结果。优化程序将通过预计所需的磁盘读写次数来确定各个策略的成本，然后选择成本最低的策略。

优化程序使用通用磁盘访问成本模型来区分对数据库文件的随机检索和顺序检索之间存在的相对性能差异。可以利用 ALTER DATABASE 语句为特定的硬件配置校准数据库。特定成本模型的细节可以用 sa_get_dtt() 存储过程来确定。

缺省情况下，对查询处理进行优化是为了返回完整的结果集。您可以使用 OPTIMIZATION_GOAL 选项对此进行更改，以最大程度地降低快速返回第一行所需的成本。

打开 [结果] 窗格中的 [计划] 选项卡，可以查看 Interactive SQL 中任何查询的访问计划。要更改所显示的细节级别，可更改 [选项] 对话框的 [计划] 选项卡的设置 （在 [工具] 菜单中）。

有关优化目标的详细信息，请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_GOAL 选项 [database]」。

有关读取访问规划的详细信息，请参见 「读取访问计划」第 482 页。

优化程序预计值和直方图

优化程序会根据**直方图** （存储在数据库中）和**试探法** （有根据的推测）来选择语句处理策略。

直方图

直方图也被称作列统计信息，存储有关值在列中的分布情况的信息。在 Adaptive Server Anywhere 中，直方图通过以下方式表示列的数据分布：将列的域划分为一组连续的值域 （也称为**桶**）；对于每个值域 （即桶），记下表中有多少行的列值在桶内。

Adaptive Server Anywhere 会特别注意在有大量行的表中的单一列值。重要的单值选择性保留在单个的直方图桶 （例如，包括列域中的单个值的桶）中。 Adaptive Server Anywhere 会尝试在每个直方图中保留最少

数量的单个桶，通常是在 10 与 100 之间，具体取决于表的大小。此外，所有选择性大于 1% 的单个值均以单个桶的形式保留。因此，给定列的直方图会记下该列的前 N 个单值选择性，而值 N 取决于表的大小和大于 1% 的单值选择性的数量。

如果满足值域的最低数量，则低选择性频率一旦出现就会被高选择性频率替换。直方图在看到了足够多的选择性大于 1% 的值后，将只拥有多于最低数量的单值域。

Adaptive Server Anywhere 更新作为查询执行的副产品的直方图。在执行查询时， Adaptive Server Anywhere 会比较直方图为给定谓语预计的行数和实际满足该谓语的行数，然后调整直方图中的值，以便减少可能在后续的优化中出现的错误。

Adaptive Server Anywhere 还会在执行 LOAD TABLE 和 UPDATE 语句以及执行 CREATE INDEX 语句的过程中创建或更新直方图。

试探法

对于潜在的执行计划中的每个表，优化程序会预计将形成部分结果的行数。该行数取决于表的大小以及查询的 WHERE 子句或 ON 子句中的限制。

给定某一列的直方图后， Adaptive Server Anywhere 会预计满足该列上的给定查询谓语的行数，其方法是累加与满足指定谓语的值相重叠的所有值域中的行数。对于部分包含在查询结果集中的直方图中的值域，Adaptive Server Anywhere 将使用该值域内的内插值。

在许多情况下，优化程序会使用更为高级的尝试法。例如，只有在没有更合适的统计信息时，优化程序才使用缺省预计值。另外，优化程序还会利用索引和键来改进它对行数的推测。下面是一些单列的示例：

- 使某列等于某个值：当该列具有唯一索引或者成为主键时预计一行。
- 索引列与常量的比较：探测索引以预计满足比较条件的行的百分比。
- 使某个外键等于某个主键 （键连接）：使用关系表大小确定预计值。例如，如果一个有 5000 行的表具有一个外键，该外键引用一个有 1000 行的表，则优化程序会推测：对于每个主键行存在五个外键行。

另请参见

有关直方图的详细信息，请参见 『ASA SQL 参考』> 「SYSCOLSTAT 系统表」。

有关获取谓语选择性和列值分布的信息，请参见以下内容：

- 『ASA SQL 参考』> 「sa_get_histogram 系统过程」
- 『ASA 数据库管理指南』> 「直方图实用程序」
- 『ASA SQL 参考』> 「ESTIMATE 函数 [Miscellaneous]」
- 『ASA SQL 参考』> 「ESTIMATE_SOURCE 函数 [杂类]」

更新列统计信息

列统计信息永久地存储在数据库的系统表 SYSCOLSTAT 中。如果使用 INSERT、 UPDATE 或 DELETE 语句更改了大量数据，统计信息会自动更新。

如果您怀疑性能不佳是由于统计信息没有准确反映当前列值所致，则可能需要执行 DROP STATISTICS 或 CREATE STATISTICS 语句。 CREATE STATISTICS 删除旧的统计信息并创建新的统计信息，而 DROP STATISTICS 只删除现有的统计信息。

当更为准确的统计信息可供使用时，优化程序可以计算出更适当的预计值，从而提高后继查询的性能。但是，只有当不正确的预计值导致优化不当的查询时，不正确的预计值才会造成问题。

当您执行 LOAD TABLE 时，系统将创建表的统计信息。 Adaptive Server Anywhere 自动为具有五行或者更多行的表创建直方图。但是，在执行 CREATE STATISTICS 语句时，无论表中有多少行，都将为指定的表创建一个直方图。

有关列统计信息的更多信息，请参见：

- 『ASA SQL 参考』> 「SYSCOLSTAT 系统表」
- 『ASA SQL 参考』> 「DROP STATISTICS 语句」
- 『ASA SQL 参考』> 「CREATE STATISTICS 语句」

自动性能调优

查询中最常见的约束之一是与列值相等。下面的示例测试 sex 列的等同性。

```
SELECT *
 FROM employee
 WHERE sex = 'f'
```

查询通常会在第二次执行时以不同的方式进行优化。对于上述类型的约束， Adaptive Server Anywhere 将积累经验，自动考虑其值分布不常见的列。数据库会永久地存储这一信息，直到您使用 DROP STATISTICS 命令将其显式删除为止。请注意，在该列上有谓语的后继查询可能会使服务器在该列上重新创建一个直方图。

基础假定

Adaptive Server Anywhere 查询优化程序的设计方向和理念建立在大量的假定之上。通过理解优化程序的决定，您可以提高您自己的应用程序的质量或性能。这些假定提供了将帮助您了解其余章节所含信息的上下文。

最少的管理工作

过去，要想使数据库服务器发挥出很高的性能，主要需要依赖于有知识渊博、尽心尽责的数据库管理员。为了取得很好的数据库性能，数据库管理员需要花费大量的时间来调整各种各样的数据存储和性能控制参数。随着数据库数据的变化，经常需要不断地调整这些控制参数。

随着数据库的发展和变化， Adaptive Server Anywhere 不断积累经验并做出了调整。每个查询都会更好地了解数据库中的数据分布。 Adaptive Server Anywhere 将自动存储此信息并利用它优化将来的查询。

每个查询既参与提供这种内部知识，又受益于这种内部知识。每个用户都可以受益于 Adaptive Server Anywhere 通过执行其他用户的查询而获取的知识。

因此，统计信息收集机制是数据库服务器中一个必不可少的部分，它不需要任何外部的机制。如果您发现它在某种情况下可能会有帮助，则可以为数据库服务器提供数据分布的预计值，以供它在优化过程中使用。如果您将这些预计值编为触发器或过程代码，则您将承担维护这些预计值并在适当的时候将其更新的责任。

为前几行或整个结果集进行优化

利用 OPTIMIZATION_GOAL 选项，您可以指定查询处理的优化目标是快速返回第一行还是尽量降低返回整个结果集所需的成本。缺省值为第一行。

有关详细信息，请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_GOAL 选项 [database]」。

优化混合或 OLAP 负载

使用 OPTIMIZATION_WORKLOAD 选项，可以指定是否应该针对通常将更新、删除或插入与查询同时执行 （混合负载）的数据库优化查询处理，或者数据库中的更新活动的主要形式是否是很少与查询同时执行的批处理式更新。

有关详细信息，请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_WORKLOAD 选项 [database]」。

统计信息存在并正确

优化程序是自行调优的，它在内部存储所需的全部信息。系统表 SYSCOLSTAT 是数据分布和谓语选择性预计值的持久存储库。在完成每个查询时， Adaptive Server Anywhere 将使用在查询执行过程中收集到的统计信息更新 SYSCOLSTAT。因此，所有后继的查询将得到更为准确的预计值。

优化程序非常依赖这些统计信息，因此它所生成的访问计划的质量在很大程度上取决于这些统计信息。如果您最近插入了大量的新行，这些统计信息可能会不再准确地描述相应的数据。您可能会发现后继查询的执行速度非常慢。

如果您对数据做出了较大的变更并且发现查询执行速度很慢，则可能需要执行 DROP STATISTICS 和 / 或 CREATE STATISTICS。

索引通常可以满足谓语

通常， Adaptive Server Anywhere 可以借助于索引来计算谓语。利用索引，优化程序可以加快数据访问速度并减少所读取的信息量。例如，如果 OPTIMIZATION_GOAL 设置为第一行 (first-row)， Adaptive Server Anywhere 优化程序将尝试使用索引来满足 ORDER BY 子句和 GROUP BY 子句。

当优化程序无法找到合适的索引时，它会采用顺序表扫描，这可能会占用大量的资源。索引可以在连接表时显著地提高性能。只要在表中添加索引或重写查询将有助于高效地处理常见的请求，就应该执行这样的操作。

虚拟内存是稀有资源

操作系统和大量的应用程序会频繁地共享标准计算机的内存。 Adaptive Server Anywhere 将内存当作稀有资源。由于 Adaptive Server Anywhere 非常经济地使用内存，因此可以在小型计算机上运行。如果您要在便携式计算机或旧式计算机上运行数据库，这种经济性就显得非常重要。

保留额外的内存 （例如用于保存游标的内容）可能会占用大量的资源。如果高速缓存已满，则可能需要将一个或多个页写入磁盘，以便为新页留出空间。为了完成后继的操作，可能需要重新读取某些页。

鉴于这种情况， Adaptive Server Anywhere 将较高的成本与需要附加高速缓存开销的执行计划相关联。这种成本会促使优化程序放弃选择使用工作表的计划。

另一方面，它在提高性能的情况下会非常小心地使用内存。例如，如果在处理查询的过程中需要重复使用子查询的结果，它就会高速缓存这些结果。

将子查询重写为 EXISTS 谓语

作为 Adaptive Server Anywhere 设计基础的假定要求它节约内存并且在缺省情况下尽可能快地返回游标的前几项结果。为了实现这些目标， Adaptive Server Anywhere 会将所有集合操作子查询 （如 IN、 ANY 或 SOME 谓语）重写为 EXISTS 谓语。这样， Adaptive Server Anywhere 就可以避免创建不必要的工作表，并且可以更方便地找到用于对表进行访问的合适索引。

非相关子查询是这样的子查询：它们不包含对查询中其它更高级别部分中的表的显式引用。

下面是一个包含无关子查询的普通查询。它选择有关在 2001 年 1 月 1 日没有下订单的所有客户的信息。

非相关子查询

```
SELECT *
 FROM customer c
 WHERE c.id NOT IN
     (    SELECT o.cust_id
         FROM sales_order o
         WHERE o.order_date = '2001-01-01' )
```

计算此查询的一种可能方式是先读取 sales_order 表并为所有在 2001 年 1 月 1 日下订单的客户创建工作表，然后读取 customer 表并为该工作表列出的每名客户抽取一行。

但是， Adaptive Server Anywhere 会避免将结果实现为工作表。它还将优先考虑最快返回前几行结果的计划。因此，优化程序会使用 EXISTS 谓语重写这些查询。在这种形式中，子查询会变得**相关**：子查询现在包含对 customer 表的 ID 列的显式引用。

相关子查询

```
SELECT *
 FROM customer c
 WHERE NOT EXISTS
     (    SELECT *
         FROM sales_order o
         WHERE o.order_date = '2000-01-01'
             AND o.cust_id = c.id )
```

```
c<seq> : o<key_so_customer>
```

该子查询在语义上等效于上面的那个子查询，但在以这种新语法表示时，该子查询表现出两处优点。

1. 优化程序可以选择使用 sales_order 表的 cust_id 属性或 order_date 属性上的索引。（但是，在示例数据库中，只有 id 和 cust_id 列带有索引。
2. 优化程序可以选择计算子查询，而不将中间结果实现为工作表。

Adaptive Server Anywhere 可以在处理过程中高速缓存此相关子查询的结果。这一策略使 Adaptive Server Anywhere 能够重新使用先前计算出的结果。在上面的那个查询中，由于客户标识号在 customer 表中是唯一的，因此高速缓存不会起什么作用。

有关子查询高速缓存的进一步信息，请参见 「子查询和函数高速缓存」第 480 页。

访问计划高速缓存

通常，优化程序会在每次执行查询时为查询选择访问计划。通过在执行时进行优化，优化程序可以根据当前的系统状态以及当前选择性预计值和基于主机变量值的预计值来选择计划。对于执行得非常频繁的查询，查询优化的成本可能会超过执行时优化所带来的好处。对于在存储过程、存储函数和触发器中执行的查询以及 INSERT、 UPDATE 和 DELETE 语句，优化程序将在执行查询的操作之间高速缓存执行计划。

当存储过程、存储函数或触发器中的语句由一个连接执行多次后，优化程序将为该语句建立可重用的计划。可重用的计划不会将主机变量值用于选择性预计或重写优化。因此，可重用的计划可能具有较高的成本。如果可重用计划的成本接近语句的最佳观测成本，优化程序会选择将可重用的计划添加到计划高速缓存中。否则，对每一次执行进行优化的优点将超过避免优化所节约的成本，而且执行计划不会被高速缓存。

计划高速缓存是逐连接高速缓存 , 它用于缓存执行访问计划的数据结构。要重用高速缓存的计划，其步骤包括在高速缓存中查找计划并将其重置为初始状态。这通常比优化语句要快得多。如果高速缓存的计划未被频繁使用，并且不会提高高速缓存使用率，它们就可能被存储到磁盘中。优化程序会定期重新优化查询，以检验高速缓存的计划是否仍具有较高的效率。

您可以使用数据库或连接属性 QueryCachePages 来确定用于高速缓存执行计划的页数。这些页占用临时文件中的空间，但不一定会驻留在内存中。

您可以使用 QueryCachedPlans 统计信息显示当前对多少个查询执行计划进行了高速缓存。可以使用 CONNECTION_PROPERTY 检索此属性以显示为给定连接高速缓存了多少个查询执行计划，或者可以使用 DB_PROPERTY 来计算所有连接间高速缓存的执行计划的数目。此属

性可以与 QueryCachePages、 QueryOptimized、 QueryBypassed 和 QueryReused 组合使用，以帮助确定 MAX_PLANS_CACHED 选项的最佳设置

要高速缓存的最大计划数量用选项设置 MAX_PLANS_CACHED 来指定。缺省值为 20。若要禁用计划高速缓存，请将此选项设置为 0。

有关详细信息，请参见 『ASA 数据库管理指南』> 「MAX_PLANS_CACHED 选项 [database]」。

优化步骤

Adaptive Server Anywhere 优化程序在生成合适的访问计划时会遵循以下步骤。

1. 分析器将 SQL 查询转换为内部表示形式。它可以重写查询，即，将查询转换为在语法上不同但在语义上等效的形式。例如，可以将子查询重写为连接。这些转换会使语句更便于分析。

2. 严格意义上的优化只在执行之前才开始。如果您正在应用程序中使用游标，优化将在游标打开时开始。与其它许多商业数据库系统不同， Adaptive Server Anywhere 只是在执行每个语句之前才优化该语句。

3. 优化程序会对语句执行语义优化。它会重写每个 SQL 语句 — 只要这样做有助于产生更合适、更有效的访问计划。

4. 优化程序对每个子查询执行连接枚举。

5. 优化程序优化访问顺序。

由于 Adaptive Server Anywhere 对每个语句执行实时优化，优化程序可以访问主机变量和存储过程变量的值。因此，优化程序会执行更有效的选择性分析，从而做出更为适当的选择。

Adaptive Server Anywhere 将优化您执行的每个查询，而不管您在这之前已经将该查询执行过多少次，但存储过程或用户定义函数中包含的查询则属于例外。对于存储过程或用户定义函数中包含的查询，优化程序可以高速缓存访问计划，使它们可以被重用。

有关详细信息，请参见 「访问计划高速缓存」第 436 页。

由于 Adaptive Server Anywhere 每次执行查询时都会保存统计信息，因此优化程序可以积累执行先前计划的经验，并且在适当的时候调整它的选择。

简单查询

如果优化程序将查询识别为简单查询，则将采用尝试法（而不是基于成本的优化）- 优化程序会决定是使用索引扫描还是使用顺序表扫描，然后立即建立并执行访问计划。第 4 步和第 5 步将被绕过。

简单查询是一种 DYNAMIC SCROLL 或 NO SCROLL 游标，它不包含任何类型的子查询、多个表、代理表、用户定义函数、 NUMBER(*)、UNION、集合、 DISTINCT、 GROUP BY 或单个列上的多个谓语。只有当 WHERE 子句包含每个主键列上的条件时，简单查询才可以包含 ORDER BY。

查询执行算法

下面将介绍 Adaptive Server Anywhere 计算查询时使用的算法。

表访问算法

对单个表进行访问的基本方式是索引扫描和顺序表扫描。

索引扫描

索引扫描是指使用索引来确定哪些行满足搜索条件。它仅读取那些满足条件的页。索引可以将行按排序的顺序返回。

索引扫描在简要计划和详细计划中显示为 *correlation_name*<*index_name*>，其中 *correlation_name* 是在 FROM 子句中指定的相关名 （如果未指定相关名，则为表名）； *index_name* 是索引的名称。

索引为读取大表中的少数行提供了高效的机制。但是，索引扫描从数据库中读取页的顺序是随机的，这种方法所占用的资源比按顺序读取所占用的资源多。如果页上有多行满足搜索条件，索引扫描还可以多次引用同一表页。如果索引扫描仅匹配少数几页，这些页很可能会留在高速缓存中，而多次访问不会导致额外的 I/O。但是，如果有多页匹配搜索条件，它们可能无法全部装入高速缓存。这会使索引扫描从磁盘中多次读取同一页。

如果 OPTIMIZATION_GOAL 设置为第一行，则优化程序往往会优先使用索引扫描 （与顺序表扫描相比）。这是因为索引扫描会比表扫描更快地返回查询的前几行。

索引也可用于满足在 ORDER BY 子句中显式定义或者为 GROUP BY 或 DISTINCT 子句隐式定义的排序要求。与基于散列的分组和非重复方法相比，带 GROUP BY 和 DISTINCT 的排序方法可以更快地返回初始行，但它们在返回整个结果集时的速度可能较慢。

如果搜索条件具有 sargable 特性，并且优化程序对搜索条件选择性的预计值较小，使索引扫描的成本低于顺序表扫描的成本，则优化程序将使用索引扫描来满足搜索条件。

在从索引读取行之后，索引扫描也可以计算非可优化搜索条件。在索引扫描中计算条件比在索引扫描之后在过滤器中进行计算要更有效。

有关 Adaptive Server Anywhere 何时可以利用索引的详细信息，请参见「谓语分析」第 467 页。

有关优化目标的详细信息，请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_GOAL 选项 [database]」。

顺序表扫描

顺序表扫描是指按照行在数据库中的存储顺序来读取表上所有页中的所有行。

顺序表扫描在简要计划和详细计划中显示为 *correlation_name*<*seq*>，其中 *correlation_name* 是在 FROM 子句中指定的相关名 （如果未指定相关名，则为表名）。

当大多数表页可能具有匹配查询搜索条件的行或者尚未定义合适的索引时，将使用这种扫描。

虽然顺序表扫描可能会比索引扫描读取更多的页，但由于这些页是以连续块的形式从磁盘中读取的，因此磁盘 I/O 的成本会非常低 （如果数据库文件在磁盘上没有碎片，这种性能改善将最为明显）。顺序 I/O 会尽量减少因磁盘头移动和旋转等待时间而导致的开销。对于大表，顺序表扫描还会一次读取包含多个页的组。这将进一步降低顺序表扫描相对于索引扫描的成本。

虽然顺序表扫描可能会比匹配许多行的索引扫描花费的时间少，但如果顺序表扫描被执行多次，它们将无法像索引扫描那样有效地利用高速缓存。由于索引扫描访问的表页很可能较少，因此这些页很可能会在高速缓存中，这将加快访问的速度。所以，对于重复的表访问 （如嵌套循环连接的右侧），最好采用索引扫描。

对于隔离级别 3， Adaptive Server Anywhere 会在所访问的每一行上获取锁 - 即使它不满足搜索条件。对于这一级别，顺序表扫描会在表中的所有行上获取锁，而索引扫描只在匹配搜索条件的行上获取锁。这意味着顺序表扫描在多用户环境中可能会显著地降低吞吐量。因此，优化程序在隔离级别 3 会优先使用索引访问，而不是顺序访问。顺序扫描可在扫描过程中有效地计算表列和常数之间的简单比较判定。其它只引用被扫描的表的搜索条件在这些简单比较之后进行计算，而此方法比顺序扫描后在过滤器中计算条件要更稍微高效一些。

散列表扫描

散列表扫描是一种运算符，使用它， Adaptive Server Anywhere 可以扫描散列连接的生成端，就像它是内存中的表一样。这样，您便可以将具有以下结构的计划：

```
table1<seq>*JH ( arbitrary dfos...( table2<seq> ) )
```

转换为以下结构：

```
table1<seq>*JF ( arbitrary dfos...( HTS JNB table2<idx> ) )
```

其中 idx 是索引，可用于探测存储在散列表中的连接键值。

当散列连接和扫描之间具有干预运算符时，它会减少需由其它运算符处理的所需行数。

当索引探测的选择性很高时 （例如，当生成端的行数与索引的基数相比较少时），此策略最有用。

注意
如果散列连接的生成端很大，则执行常规顺序扫描会更有效。

优化程序会计算域值生成大小，方式与它为其它连接执行计算阈值相似，只不过执行期间会将 (HTS JNB table<idx>) 视为顺序扫描 (table<seq>)。

注意
如果散列表的生成端不得不溢出到磁盘，则会使用顺序策略。

并行索引扫描

在具有多个磁盘转轴的卷上， Adaptive Server Anywhere 可通过并行方式使用索引扫描表。通过并行索引扫描，服务器可以利用多张磁盘间的表和索引页同时安排多个输入 / 输出操作。并行执行的每个扫描都会搜索与不同键匹配的行，并可确保在需要这些页时它们在内存中。

注意
并行索引扫描在压缩索引上不受支持，而且在隔离级别 3 中也不受支持。

IN 列表

当 IN 谓语可以使用索引来得到满足时，将使用 IN 列表算法。例如，在以下查询中，优化程序发现它可以使用 employee 表的主键索引来对该表进行访问。

```
SELECT *
 FROM employee
 WHERE emp_id in (102, 105, 129)
```

为了完成这一任务，用特殊的 IN 列表表在左侧建立了一个连接。将从 IN 列表中读取行，并将行用于探查 employee 表。多个 IN 列表可以使用同一索引来得到满足。如果优化程序选择不使用索引来满足 IN 谓语（可能是因为其它索引会提供更好的性能），那么， IN 列表就会作为过滤器中的谓语出现。

派生表

派生表是包含在查询的 FROM 子句中的 SELECT 语句。 SELECT 语句的结果集在逻辑上被当作表来处理。查询优化程序也可能在查询重写过程中生成派生表，例如在包含基于集合的运算 UNION、 INTERSECT、或 EXCEPT 的查询中。图形式计划显示派生表的名称和计算列的列表。

有关派生表的详细信息，请参见 「FROM 子句：指定表」第 237 页和『ASA SQL 参考』> 「FROM 子句」。

示例

下面的示例数据库查询在其图形式计划里有派生表：

```
SELECT emp_id FROM employee
 UNION ALL
 SELECT dept_id FROM (         SELECT TOP 5 dept_id
     FROM department
     ORDER BY dept_name DESC ) MyDerivedTable
```

递归表

递归表是作为查询中的 WITH 子句的结果构建的公用表表达式。 WITH 子句用于递归联合查询。公用表表达式是临时视图，它们只在某一条 SELECT 语句的范围内是已知的。

有关详细信息，请参见 「公用表表达式」第 331 页。

连接算法

当多个表出现在 FROM 子句中时，需要使用连接算法。您不能指定所使用的连接算法 - 这一选择是由优化程序做出的。

嵌套循环连接算法

嵌套循环连接是通过为左侧的每行完全读取右侧的内容来计算左右两侧的连接的。（由于优化程序将为请求中的每个块选择正确的连接顺序，因此表在查询中的语法顺序无关紧要。）

如果连接条件不包含等同性条件或者它正在为第一行进行优化，优化程序就可能会选择嵌套循环连接。

由于嵌套循环连接会多次读取右侧，因此它对右侧的成本非常敏感。如果右侧是索引扫描或小表，则可以使用先前迭代中的高速缓存页计算右侧。另一方面，如果右侧是顺序表扫描或匹配许多行的索引扫描，则需要从磁盘中多次读取右侧。通常，嵌套循环连接的效率比其它连接方法的效率低。但是，与必须在返回之前计算整个结果的连接方法相比，嵌套循环连接可以快速地返回第一个匹配行。

嵌套循环连接是唯一可以为包含连接的查询提供敏感语义的连接算法。这意味着连接上的敏感游标只能用嵌套循环连接来执行。

半连接只从右侧读取第一个匹配行。它是更为有效的嵌套循环连接版本，但只能在使用 EXISTS （有时是 DISTINCT）关键字时使用。

嵌套循环半连接算法

与上面所述的嵌套循环连接相似，嵌套循环半连接算法使用嵌套循环算法将左侧的每行与右侧连接起来。由于使用了嵌套循环连接，会将右侧读取多次，因此对于较大的输入，最好进行索引扫描。然而，嵌套循环

半连接与嵌套循环连接有两方面的不同。第一，半连接只从左侧输出值；右侧只用来限制左侧的哪些行将显示在结果中。第二，在遇到第一个匹配项后，嵌套循环半连接算法将立即停止右侧的每个搜索。当连接的输入包括来自已经被重写为连接的存在限定 （IN、 SOME、 ANY、 EXISTS）嵌套查询表的表达式，则可将嵌套循环半连接用作连接算法。

[嵌套块连接] 算法和 [排序块] 算法

嵌套块连接 （也被称作块嵌套循环连接）从左侧读取行块，并按照连接属性 （在连接条件中使用的列）将行排序。嵌套块连接的左侧子项被称作排序块节点。对于具有相同连接属性的每个行块，将扫描一次右侧。如果左侧有若干个行与右侧的每个行连接，则此算法将提高嵌套循环连接的性能。

如果左侧的许多行的连接属性具有相同值并且右侧具有满足搜索条件的索引，则优化程序将会选择嵌套块连接。

每个嵌套块连接具有作为排序块节点的左侧子项。为此节点显示的成本是从左侧输入中读取并将行排序所需的成本。

左侧输入以块的形式被读入到内存中。对左侧输入中表的更改可能不会在结果中出现。因此，嵌套块连接无法提供敏感语义。

在左侧输入中的行被复制到内存之前，嵌套块连接会锁定这些行。

当内存条件不足时禁用嵌套块连接算法。

[散列连接] 算法

散列连接算法会用两个输入中的较小输入建立内存中的散列表，然后读取较大的输入并探查内存中的散列表以查找匹配项，然后将找到的匹配项写入工作表中。如果内存无法容纳较小的输入，散列连接运算符就会将这两个输入分为较小的工作表。将以递归的方式对这些较小的工作表进行处理，直到较小的输入能够装入内存为止。

如果较小的输入能够装入内存，则散列连接算法的性能将达到最佳，这与较大输入的大小无关。通常，如果其中一个输入将明显小于另一个输入，优化程序就会选择散列连接。

如果散列连接算法的运行环境中没有足够的高速缓存来保存其连接属性为特定值的所有行，散列连接算法将无法完成。在这种情况下，散列连接方法会放弃中间结果，而改用基于索引的嵌套循环连接。较小表的所有行都将被读取并用来探查工作表以查找匹配项。这种基于索引的策略要比其它连接方法慢得多，如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用散列连接生成访问计划。当因为内存不足而需要嵌套循环策略时，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库或连接属性读取此监视器，或者在 Windows 性能监视器的 [查询：内存不足策略] 计数器中读取此监视器。

当内存条件不足时，禁止在 Windows CE 上使用散列连接算法。

注意： Windows CE、 95、 98 或 Me 上可能没有提供 Windows 性能监视器。

有关详细信息，请参见 『ASA 数据库管理指南』> 「连接级属性」中的 QueryLowMemoryStrategy。

在返回第一行之前，散列连接算法计算会计算所有结果行。

散列连接算法会使用一个工作表，该工作表提供不敏感的语义 （除非已请求对值敏感的游标）。

在输入中的行被复制到内存之前，散列连接会锁定这些行。

[散列半连接] 算法

散列半连接是散列连接算法的变体，它执行左侧和右侧的半连接。与上面所说的嵌套循环半连接相同，右侧只用来确定左侧的哪些行将显示在结果中。使用散列半连接时，将读取右侧，以形成内存中的散列表，然后，将从左侧的每行探查该表。一旦找到任何匹配项，左侧的行被输出到结果中，匹配过程将重新开始以查找左侧的下一行。必须至少有一个等同性连接条件，查询优化器才会考虑散列半连接。与嵌套循环半连接相同，当连接的输入包括来自已经被重写为连接的存在限定 （IN、 SOME、 ANY、 EXISTS）嵌套查询表的表达式，则可使用散列半连接。如果连接条件包括不等同性条件，或不存在合适的索引以对右侧进行带索引的检索，使用散列半连接将比使用嵌套循环半连接将获得较好的性能。

对于散列连接，如果没有足够的高速缓存来完成操作，散列半连接算法将还原为嵌套循环半连接策略。如果是这样，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库或连接属性读取此监视器，或者在 Windows 性能监视器的 [查询：内存不足策略] 计数器中读取此监视器。

*注意：*Windows CE、 95、 98 或 Me 上可能没有提供 Windows 性能监视器。

有关详细信息，请参见 『ASA 数据库管理指南』> 「连接级属性」中的 QueryLowMemoryStrategy。

[散列反半连接] 算法

散列反半连接是散列连接算法的变体，它在左侧和右侧之间执行反半连接。与上面所介绍的嵌套循环反半连接相同，右侧只用来确定左侧的哪些行将显示在结果中。使用散列反半连接时，将读取右侧，以形成内存中的散列表，然后，将从左侧的每行探测该表。仅当没有从右侧找到任何匹配的行时，才输出左侧的每行。与嵌套循环反半连接相同，当连接的输入包括来自已经被重写为反连接的广泛限定 （NOT IN、 ALL、 NOT EXISTS）嵌套查询表的表达式时，则可使用散列反半连接。如果连接条件包括不等同性条件，或者如果不存在用于对右侧进行成本足够低的索引检索的合适索引，则使用散列反半连接往往比使用嵌套循环反半连接效果更好。

与散列连接一样，如果没有足够的高速缓存内存来完成操作，散列反半连接算法可能会恢复为嵌套循环反半连接策略。如果是这样，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库或连接属性读取此监视器，或者在 Windows 性能监视器的 [查询：内存不足策略] 计数器中读取此监视器。

*注意：*Windows CE、 95、 98 或 Me 上可能没有提供 Windows 性能监视器。

有关详细信息，请参见 『ASA 数据库管理指南』> 「连接级属性」中的 QueryLowMemoryStrategy。

[合并连接] 算法

合并连接算法读取均已按照连接属性进行排序的两个输入。对于左侧输入中的每个行，该算法会通过按排序顺序访问行来读取右侧输入中的所有匹配行。

如果输入尚未按连接属性排序 （可能是因为先前的合并连接或因为曾使用索引来满足搜索条件），优化程序将添加一个排序，以生成正确的行顺序。这一排序会增加合并连接的成本。

与散列连接相比，合并连接的优点之一是排序成本可以在若干个连接上分摊 （只要这些合并连接涉及相同的属性）。如果输入的大小可能比较接近或者排序成本可以在若干个操作上分摊，优化程序将优先选择合并连接 （而不是散列连接）。

[递归散列连接] 算法

递归散列连接是在递归联合查询中使用的散列连接算法的变体。

有关详细信息，请参见 「[散列连接] 算法」第 444 页和 「递归公用表表达式」第 340 页。

[递归左外散列连接] 算法

递归左外散列连接是在某些特定递归联合查询中使用的散列连接算法的变体。

有关详细信息，请参见 「[散列连接] 算法」第 444 页和 「递归公用表表达式」第 340 页。

重复排除算法

重复排除运算符会产生不含重复行的输出。重复排除节点可以由优化程序引入 （例如在将嵌套查询转换为连接时）。

有关详细信息，请参见 「[非重复散列] 算法」第 448 页、「[非重复排序] 算法」第 448 页和 「[非重复索引] 算法」第 448 页。

[非重复散列] 算法

非重复散列算法读取输入并建立内存中的散列表。如果在散列表中找到某一输入行，将忽略该输入行；否则，该输入行将被写入工作表中。如果输入不能完全装入内存中的散列表，则会将其分为较小的工作表并以递归的方式对其进行处理。

如果非重复行能够装入内存中的表，非重复散列算法就会发挥优良的性能，这与输入中的总行数无关。

非重复散列将使用工作表，因此可以提供不敏感的语义或对值敏感的语义。

如果非重复散列算法的运行环境中的可用高速缓存极少，该算法将无法完成。在这种情况下，非重复散列方法会放弃中间结果，而改用非重复索引算法。如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用非重复散列算法生成访问计划。

只要一找到以前没有返回的行，非重复散列将返回该行。但是，必须完全实现非重复散列的结果后，才能从查询返回。如果必要，优化器将向执行计划添加一个工作表以确保该实现。

非重复散列会锁定其输入的行。

[非重复排序] 算法

如果输入按所有列进行排序，则可以使用非重复排序算法。该算法读取每一行并将之与前一行进行比较。如果相同，则忽略该行；否则，将输出该行。如果行已经排序 （可能是由于索引或合并连接），非重复排序算法将非常有效；如果输入未排序，优化程序将插入一个排序。非重复排序本身不使用任何工作表，但是所插入的任何排序都会使用工作表。

[非重复索引] 算法

非重复索引算法维护输入中的唯一行的工作表。当从输入中读取行时，将搜索工作表上的索引，以查找先前见到的输入行重复项。如果找到重复项，则忽略该输入行。否则，将该输入行插入工作表中。工作表索引将在 SELECT 列表中的所有列上创建；为了提高索引性能，将包含一个散列表达式作为第一个表达式。此散列表达式是计算值，它体现 SELECT 列中所有列的值。

当遇到非重复行时，非重复索引方法将返回这些行。这样，它就可以比其它重复排除方法更快地返回前几行。非重复索引算法一次仅在内存中存储两行，而且在内存严重不足的情况下也能发挥优良的性能。但是，如果非重复行的数目太大，非重复索引算法的执行成本通常高于非重复散列算法的执行成本。用于存储非重复行的工作表可能无法装入高速缓存，进而导致以随机访问模式多次重复读取工作表页。

由于非重复索引方法使用工作表，它不能提供完全敏感的语义；但是，它也不提供完全不敏感的语义，因此不敏感的游标需要另一个工作表。

非重复索引方法会锁定其输入的行。

分组算法

分组算法计算输入的汇总。只有当查询包含 GROUP BY 子句或集合函数 （如 `SELECT COUNT(*) FROM T`），分组算法才适用。

有关详细信息，请参见 「[散列分组依据] 算法」第 449 页、「[排序分组依据] 算法」第 450 页、「[索引分组依据] 算法」第 451 页和 「[单行分组依据] 算法」第 451 页。

[散列分组依据] 算法

散列分组依据算法会用组行创建内存中的散列表。当从输入中读取行时，将更新组行。如果内存无法容纳散列表，将把输入分为较小的表，并以递归的方式对这些表进行分区，直到能够装入内存为止。如果没有足够的内存可用来保存分区，优化程序就会放弃散列分组依据算法的中间结果，而改用索引分组依据算法。如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用散列分组依据算法生成访问计划。

如果组能够装入内存，散列分组依据算法就会发挥优良的性能，这与输入的大小无关。

在返回第一行之前，散列分组依据算法会计算所有结果行，该算法可用于满足完全敏感的游标或对值敏感的游标。必须完全实现散列分组依据的结果后，才能从查询返回。如果必要，优化器将向执行计划添加一个工作表以确保该实现。

[按照集进行散列分组] 算法

执行 GROUPING SETS 查询时使用的 [散列分组依据] 算法的变体。

有关详细信息，请参见 「[散列分组依据] 算法」第 449 页。

聚簇散列 Group By 算法

在有些情况下，输入表的分组列中的值会聚簇，这样，相似的值便可以靠近在一起显示。例如，如果表包含的列始终设置为当前日期，则具有一个日期的所有行在表中就会相对靠近。 [聚簇散列分组依据] 算法利用此聚簇。

当为显著大于可用内存的表分组时，优化程序可能会使用 [聚簇散列分组依据]。特别是当 HAVING 谓语只返回一小部分行时，它十分有效。

如果在数据更新与查询执行同时进行的环境中选择 [聚簇散列分组依据] 算法，则可能会导致出现大量不能充分利用优化程序执行的工作的现象。因此， [聚簇散列分组依据] 最适合于偶尔进行批处理式更新和基于读取的查询的 OLAP 负载。将 OPTIMIZATION_WORKLOAD 选项设置为 **OLAP** 以指示优化程序应该在它调查的可能性中包括 [聚簇散列分组依据] 算法。

有关详细信息，请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_WORKLOAD 选项 [database]」。

[排序分组依据] 算法

排序分组依据算法读取按分组列排序的输入。当读取每一行时，该算法会将该行与前一行进行比较。如果分组列匹配，则将更新当前组；否则，将输出当前组并启动新组。

[按照集进行排序分组] 算法

执行 GROUPING SETS 查询时使用的 [排序分组依据] 算法的变体。此算法要求索引可用。

有关详细信息，请参见 「[排序分组依据] 算法」第 450 页。

[索引分组依据] 算法

索引分组依据算法与非重复索引算法类似。此算法会建立一个工作表，其中每一组都有一行。当读取输入行时，将使用索引在工作表中查找关联组。它会更新集合函数并将组行重写到工作表中。如果没有找到组记录，则将初始化新的组记录并将其插入到工作表中。

当优化程序有理由确信输入大小非常小时，它就会选择索引分组依据算法。

索引分组依据算法会在返回第一行之前计算所有结果行，并且将完全实现其输入。该算法可用于满足 FULLY INSENSITIVE 的要求。

[单行分组依据] 算法

当没有指定 GROUP BY 时，就会产生单行集合。

[单行 XML 分组依据] 算法

处理基于 XML 的查询时使用的算法。

[按照集进行分组] 算法

处理包含 GROUPING SETS 的 OLAP 查询时使用的算法。

当由于缺少索引而无法使用 [按照集进行排序分组] 算法时，可以使用此算法。

排序和联合算法

排序算法适用于包含 order by 子句的查询，联合算法适用于联合查询。

有关详细信息，请参见 「合并排序」第 452 页和 「Union all」第 452 页。

合并排序

排序运算符将输入的信息读入到内存中，在内存中对输入的信息进行排序，然后输出经过排序的结果。如果输入的信息不能完全装入到内存中，则将按顺序分多个步骤运行，然后将它们合并在一起。排序不返回任何行，直到它读取了所有的输入行。排序会锁定其输入行。

如果合并排序算法的运行环境中的可用高速缓存极少，该算法可能会无法完成。在这种情况下，合并排序算法将使用基于索引的排序方法对输入的其余部分进行排序。它会读取输入行并将其插入到工作表中，而且会在工作表的排序列上建立索引。在这种情况下，将使用复杂的索引扫描从工作表中读取行。这种基于索引的策略比其它连接方法要慢得多。如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用合并排序算法生成访问计划。当由于内存不足而需要使用基于索引的策略时，性能计数器将递增；您可以使用 QueryLowMemoryStrategy 属性或者在 Windows 性能监视器的 [查询：内存不足策略] 中读取此监视器。

排序关键字的大小、行大小和输入信息的总大小都会影响排序性能。对于较大的行，使用 VALUES SENSITIVE 游标的成本可能会低一些。在这种情况下， SELECT 列表中的行不会被复制到排序所使用的工作表中。虽然排序不会将输出行写到工作表中，但是必须实现排序的结果后，才能将行返回到应用程序。如果必要，优化器将添加一个工作表以确保该实现。

Union all

union all 算法从其每一项输入中读取行并将这些行输出，而不管是否有重复项。该算法用于实现 UNION 和 UNION ALL 子句。对于 UNION 子句，需要使用重复排除算法来删除 union all 所生成的任何重复项。

递归联合

递归联合算法在执行递归联合查询期间应用。

有关详细信息，请参见 「递归公用表表达式」第 340 页。

从顶部 N 排序

从顶部 N 排序算法适用于包含 TOP N 子句和 ORDER BY 子句的查询。对于只为结果集中需要的那些行进行排序，这是一种有效的算法。

有关详细信息，请参见 『ASA SQL 参考』> 「SELECT 语句」。

其它算法

下面是可以在访问计划中使用的其它方法。

过滤器、预过滤器和散列过滤器算法

过滤器的用途是应用搜索条件。搜索条件出现在 WHERE 子句和 HAVING 子句的语句中，还出现在 FROM 子句中 JOINS 的 ON 条件中。

预过滤器与过滤器相同，只不过它不取决于输入。例如预过滤器适用于 `WHERE 1 = 2` 的情况。

如果执行计划读取一个表中的所有行，然后再读取与该表相连的另一个表的任何行，则可以使用散列过滤器。散列过滤器生成一组位，并且会为第一个表中与该表的搜索条件相匹配的每一行打开一个位。访问第二个表时，将拒绝不能与第一个表连接的行。

例如，计划为：

```
R<idx> *JH S<seq> JH* T<idx>
```

此处，我们将 R 连接到 S 和 T。我们将读取 R 的所有行，然后再从 T 读取行，并立即拒绝不能与 R 连接的 T 中的行。这将减少必须存储在第二个散列连接中的行数。

有关详细信息，请参见 「WHERE 子句：指定行」第 238 页。

锁算法

锁的含义是指，在特定隔离级别存在锁。例如，在隔离级别 1，一次仅为一行使用一个锁。如果您处于隔离级别 0，将不会获取锁，但该节点将仍被称作锁。在这种情况下，锁节点会检验该行是否仍然存在。

有关详细信息，请参见 「锁定的工作方式」第 146 页。

行限制算法

行限制是通过 SELECT 语句的 TOP n 或 FIRST 子句来设置的。

有关详细信息，请参见 『ASA SQL 参考』> 「SELECT 语句」。

Bloom 过滤器算法

Bloom 过滤器或散列映射是一种数据结构，表示列或列集中的值的分布。它可以在满足下列条件的查询中使用：

- 查询中的操作在向以后的操作返回行以前会读取它的整个输入。例如，某一列上的两个表的连接需要读取所有相关行以确定它们是否满足连接条件。
- 查询中以后的操作引用该操作的结果中的行。例如，同一列中的第二个连接只使用满足第一个连接的条件的那些行。

激增算法

激增算法在执行 EXCEPT 和 INTERSECT 等集合运算期间使用。它是以下这样的运算的功能：结果集中的行数与作为运算对象的两个集合中的行数显式相关。激增算法可确保结果集中的行数正确。

有关详细信息，请参见 「使用 UNION、 INTERSECT 和 EXCEPT 对查询结果执行集合运算」第 270 页。

窗口函数算法

此算法用于计算使用窗口函数的 OLAP 查询。

有关详细信息，请参见 「使用 OLAP 函数」第 374 页和 「窗口函数」第 379 页。

物理数据组织和访问

每个表或条目的存储分配都对查询的效率有着较大的影响。下面所述的几点都会影响查询的执行速度，所以它们特别重要。

对插入的行的磁盘分配

Adaptive Server Anywhere 会在可能的情况下连续存储行

每一个小于数据库文件页面大小的新行都始终会存储在一页上。如果现有页没有足够的空间可用于新行， Adaptive Server Anywhere 就会将该行写入新页。例如，如果新行需要 600 字节的空间，但一个被部分填充的页上只有 500 字节的空间， Adaptive Server Anywhere 就会将该行放在一个新页上。

为了使表页在磁盘上更为连续， Adaptive Server Anywhere 将以块的形式（每一块包含八页）来分配表页。例如，当它需要分配某个页时，它将先分配八页，将要分配的页插入块中，然后用后面的七页将块填满。此外，它会使用可用页位图来查找 dbspace 中的连续页块，并通过读取 64K 的组和使用位图查找相关页来执行顺序扫描。这样，顺序扫描的效率就会得到提高。

Adaptive Server Anywhere 可以按任意顺序存储行

Adaptive Server Anywhere 查找页上的空间并按照它接收行的顺序插入行。它会将每一行分配给某一页，但它在表上选择的位置可能与行的插入顺序不对应。例如，为了连续存储一个很长的行，引擎可能需要开始一个新页。如果下一行较短，它可能适合放在下一页上的空位置。

所有表的行都未进行排序。如果行的接收或处理顺序非常重要，请在 SELECT 语句中使用 ORDER BY 子句对结果进行排序。依赖于表中行顺序的应用程序可能会不发出警告就失败。

如果您经常需要让表的行具有特定的顺序，则应在查询的 ORDER BY 子句中所指定的那些列上创建索引。

没有为 NULL 列预留空间

缺省情况下，每当 Adaptive Server Anywhere 插入一行时，它只会预留必需的空间，用于存储那些内含在创建时它所包含的值的行。它不会为存储空值或容纳可能会扩大的字段（例如文本字符串）而预留空间。

您可以通过在创建表时使用 PCTFREE 选项来强制 Adaptive Server Anywhere 预留空间。有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

行标识符在插入后不可变

一旦在页上给行分配主位置，行就永远不会从该页上移开。如果更新操作更改该行中的某些值，使该行不能装入给它分配的页，则将在另一页上插入行拆分和额外的信息。

这一特性值得注意，因为在您插入行时， Adaptive Server Anywhere 不会允许额外的空间。例如，假设您在表中插入大量的空行，然后使用 UPDATE 语句以一次一列地填入值。结果将是单个行中的每个值几乎都将存储在单独的页上。若要检索一行中的所有值，引擎就可能需要读取若干个磁盘页。这一简单的操作会不必要地变得极为缓慢。

您应该考虑在插入时用数据填入新行。一旦插入，它们将具有足够的空间来保存您需要保存的数据。

数据库文件从不会缩小

当您从数据库中插入和删除行时， Adaptive Server Anywhere 会自动重用这些行占用的空间。因此， Adaptive Server Anywhere 可以将一行插入先前由另一行占用的空间。

Adaptive Server Anywhere 会记录每个页上的空白空间大小。当您要求它插入新行时，它首先会搜索有关现有页空间的记录。如果它发现现有的页上有足够的空间，就会将新行放在该页上，并在必要时重组该页的内容。否则，它会开始一个新页。

但是，如果您在一段时间内删除了大量的行，而且所插入的新行不够小，无法利用空白空间，则数据库中的信息就会变得非常分散。您可以重装该表，也可以使用 REORGANIZE TABLE 语句整理该表的碎片。

有关详细信息，请参见 『ASA SQL 参考』> 「REORGANIZE TABLE 语句」。

表大小和页面大小

您为数据库选择的页面大小会影响数据库的性能。通常，如果页面大小较小，可能有益于那些从任意位置检索较少行的操作。相比之下，较大的页往往有益于那些按顺序执行表扫描的查询 — 尤其是当行在页上的存储顺序与通过索引对行进行检索的顺序相同时。在这种情况下，如果将一页读入内存以获取一行的值，则可能会产生副作用：会将下几行的内容装入内存。通常，磁盘的物理设计会使磁盘检索较少的大块比检索较多的小块更有效率。

Adaptive Server Anywhere 将为数据库 （页面大小至少为 2K）中足够大的表创建位图。每个表的位图都反映每个表页在整个 dbspace 文件中的位置。对于页面大小为 2K、 4K 或 8K 的数据库，服务器将利用位图来读取较大的表页块 (64K)，而不是一次读取一页，这将减少磁盘 I/O 操作的总次数并因此而提高性能。用户不能控制服务器有关创建或使用位图的标准。

请注意，位图 （也被称作页位置图）只能用于在版本 8.0 和更高版本中创建的数据库。如果数据库是从较早版本升级而得到的，那么，即使数据库表符合服务器的标准，服务器也不会为数据库表创建位图。此外，服务器也不会为工作表或系统表创建位图。

如果您选择较大的页面大小 （如 4 KB），则可能需要扩大高速缓存的大小。相同的空间所容纳的大页较少。例如， 1 MB 的内存可以保存 1000 个大小各为 1 KB 的页，但只能保存 250 个大小为 4 KB 的页。到底使用多少页数才算合适，完全取决于数据库以及应用程序执行的查询的性质。您可以用各种高速缓存大小进行性能测试。如果高速缓存区不能保存足够的页，当 Adaptive Server Anywhere 开始与磁盘交换常用页时，性能就会受到损害。

页面大小也会影响索引。缺省情况下，索引页的散列大小为 10 字节：对于每个索引条目，它们大约存储前 10 个字节的数据。这样就支持大致 200 个页面大小为 4K 的条目，这意味着每个索引页保存 200 行 （或 40 000 个具有两级索引的行）。每个新的索引级别都支持大 200 倍的表。页面大小可能会大大地影响条目数，条目数又将影响表所需的索引深度。大型数据库的页面大小应该是 4K。

Adaptive Server Anywhere 会尽可能地将页添满。只有当新对象太大而无法装入现有页上的空白空间时，空白空间才会累积。因此，调整页面大小可能不会对数据库的总体大小造成很大的影响。

索引

索引可以大大地提高索引列上的搜索性能。但是，索引会占用数据库中的空间并降低插入、更新和删除操作的速度。本节将帮助您确定何时应该创建索引，并介绍如何利用索引使性能达到最佳。

在许多情况下，创建索引会提高数据库的性能。索引根据部分或所有列中的值对表的行进行排序。索引使 Adaptive Server Anywhere 可以快速地查找行。它可以通过限制所访问的数据库页数来支持较大的并发性。索引还使 Adaptive Server Anywhere 能够方便地对表的行强制唯一性约束。

索引顾问是一种工具，用于为数据库选择一组适当的索引。有关详细信息，请参见 「索引顾问概述」第 73 页。

何时创建索引

没有任何简单的公式可以确定是否应该为特定的列创建索引。您必须考虑权衡索引检索的优点与该索引的维护开销。以下因素可能会帮助您确定是否应该创建索引。

- **键和唯一列** Adaptive Server Anywhere 自动在主键、外键和唯一列上创建索引。您不应该在这些列创建附加的索引。组合键属于例外，它们有时可以通过附加的索引来得到改进。

  有关详细信息，请参见 「复合索引」第 460 页。

- **搜索频率** 如果某一特定列被频繁地搜索，则可以通过在该列上创建索引改善性能。在较少被搜索的列上创建索引可能是不值得的。

- **表大小** 在较大的表（包含许多行）上创建索引比在较小表上创建索引优越性更强。例如，对于只有 20 行的表，由于按顺序进行扫描所需的时间不会比进行索引查寻所需的时间长，因此在该表上创建索引不会有太大的好处。

- **更新次数** 每次在表中插入或删除行以及每次更新索引列时，都会更新索引。列上的索引会降低插入、更新和删除操作的速度。频繁更新的数据库的索引数应该比只读数据库的索引数少。

- **空间考虑**　索引会占用数据库中的空间。如果数据库大小是您关注的主要问题，则应该尽量少创建索引。
- **数据分布**　如果索引查寻返回的值太多，则索引查寻的成本会高于按顺序进行扫描的成本。当 Adaptive Server Anywhere 发现这一情况时，它就不会使用索引。例如， Adaptive Server Anywhere 不会使用只有两个值的列 （如示例数据库中的 employee.sex）上的索引。因此，在只有少数几个非重复值的列上不应该创建索引。

索引顾问是一种工具，用于为数据库选择一组适当的索引。有关详细信息，请参见 「索引顾问概述」第 73 页。

临时表

您可以在本地临时表和全局临时表上创建索引。如果临时表会很大，并且会按排序顺序或连接对其进行多次访问，则需要在临时表上创建索引。否则，查询的任何性能改善都可能会被创建和删除索引的成本所抵消。

有关详细信息，请参见 「使用索引」第 68 页。

提高索引性能

如果索引没有按预期的方式执行，则可以考虑采取以下措施。

- 重组复合索引。
- 增加页面大小。

这些措施旨在提高索引选择性和索引条目数，下面将对此进行介绍。

索引选择性

索引选择性指的是索引所具备的定位所需索引条目而无需读取额外数据的能力。

如果选择性较低，则必须从索引所引用的表页中检索额外的信息。这种检索被称作**完全比较**，它们会对索引性能产生负面影响。

FullCompare 属性函数保持跟踪已经发生的完全比较的次数。您也可以使用 Sybase Central 性能监视器或 Windows 性能监视器来监控这一统计信息。

注意： Windows CE、 95、 98 或 Me 上可能没有提供 Windows 性能监视器。

此外，完全比较的次数也会在含统计信息的图形式计划中提供。有关详细信息，请参见 「计划中使用的常见统计信息」第 485 页。

有关 FullCompare 函数的详细信息，请参见 『ASA 数据库管理指南』> 「数据库级属性」。

索引结构和索引条目数

索引是像树一样以多个级别来组织的。索引的第一页被称作根页，它在下一级分叉为一个或多页，而这些页又再次分叉，直至达到索引的最低级别。位于最低级别的索引页被称作叶页。若要定位特定行，具有 *n* 级的索引需要对索引页读取 *n* 次并对包含实际行的数据页读取一次。通常，由于频繁使用的索引页往往被存储在高速缓存中，所以需要从磁盘中读取的次数会少于 *n* 次。

索引条目数是一个页面上存储的索引条目的数量。与条目数较小的索引的级别数相比，条目数较大的索引的级别数可能会更少。因此，如果索引条目数较大，通常意味着索引性能更佳。

您可以使用 sa_index_levels 系统过程来查看索引中的级别数。

有关详细信息，请参见 『ASA SQL 参考』> 「sa_index_levels 系统过程」。

复合索引

索引可以包含一个、两个或更多个列。两个或更多个列上的索引被称作**复合索引**。例如，以下语句创建一个具有两列的复合索引：

```
CREATE INDEX name
 ON employee (emp_lname, emp_fname)
```

如果第一列不能单独提供较高的选择性，复合索引将会非常有用。例如，当许多雇员具有相同的姓氏时， emp_lname 和 emp_fname 上的复合索引非常有用。因为每个雇员都有唯一的 ID，所以 emp_id 和 emp_lname 上的复合索引可能没有用处，因此列 emp_lname 不会提供任何附加选择性。

利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引不同于使用两个单独的索引。复合索引的结构与电话簿类似，它首先按姓氏对雇员进行排序，然后按名字对所有姓氏相同的雇员进行排序。如果您知道姓氏，电话簿将非常有用，如果您知道名字和姓氏，电话簿则更为有用，但如果您只知道名字而不知道姓氏，电话簿将没有用处。

压缩的 B 树索引方法可显著提高复合索引的性能。

列顺序

在创建复合索引时，应该仔细考虑列的顺序。对索引中的所有列执行搜索或仅对前几列执行搜索时，复合索引非常有用；仅对后面的任意列执行搜索时，复合索引则没有用处。

如果您很可能仅对一个列多次执行搜索，则该列应该是复合索引中的第一列。如果您很可能对一个两列索引中的两个列执行单独的搜索，则应该创建另一个仅包含第二列的索引。

包含多个列的主键始终会自动以复合索引的形式创建索引，其列的顺序是它们在表定义中出现的顺序，而不是在主键定义中指定的顺序。您应该考虑将通过主键来执行的搜索，以确定哪一列应该排在最前面。在后面的任何被频繁搜索的主键列上，应该考虑添加额外的索引。

例如，假设您在两个列上创建一个复合索引。一个列包含雇员的名字，另一个列包含雇员的姓氏。您可以创建一个先包含名字后包含姓氏的索引。或者，您也可以创建一个先包含姓氏后包含名字的索引。虽然这两个索引以两个列组织信息，但它们具有不同的功能。

```
CREATE INDEX fname_lname
     ON employee emp_fname, emp_lname;
 CREATE INDEX lname_fname
     ON employee emp_lname, emp_fname;
```

假设您需要搜索名字 John。唯一有用的索引是在索引的第一列包含名字的索引。由于名字为 John 的雇员会出现在索引中的任意位置，因此先按姓氏再按名字组织的索引没有用处。

如果您认为很可能需要仅按名字或仅按姓氏查找雇员，则应该创建这两个索引。

或者，您也可以创建两个索引，一个索引仅包含一个列。但是，请注意， Adaptive Server Anywhere 在处理单个查询时只使用一个索引来对任何一个表进行访问。即使您知道名字和姓氏， Adaptive Server Anywhere 也可能需要读取额外的行，以查找包含正确姓氏的行。

当您使用 CREATE INDEX 命令创建索引时 （如上例所示），列会按命令中所示的顺序创建。

主键索引和列顺序

列在主键索引中的顺序被强制为与列在表定义中出现的顺序相同，这与 PRIMARY KEY 约束中指定的列顺序无关。此外， Adaptive Server Anywhere 还将强制一个附加约束：表的主键列必须位于每个行的开头。因此，如果主键被添加到现有表中，服务器就可以重写整个表以确保键列位于每个行的开头。

在多个列出现在主键中的情况下，您应该考虑所需的搜索类型。如果合适，应切换列在表定义中的顺序，使最常搜索的列排在最前面，或者根据需要为其它列创建单独的索引。

复合索引和 ORDER BY

缺省情况下，索引的列按升序排列，但您可以选择通过在 CREATE INDEX 语句中指定 DESC 来将这些列按降序排列。

只要 ORDER BY 子句仅包含索引中的列， Adaptive Server Anywhere 就可以选择使用索引来优化 ORDER BY 查询。此外，索引列的排序方式必须与 ORDER BY 子句完全相同或完全相反。对于单列索引，这种排序方式始终会使查询可以得到优化，但复合索引则需要稍微多考虑一些问题。下表显示了一个两列索引的可能性。

索引列	可优化的 ORDER BY 查询	不可优化的 ORDER BY 查询
ASC、 ASC	ASC、 ASC 或 DESC、 DESC	ASC、 DESC 或 DESC、 ASC
ASC、 DESC	ASC、 DESC 或 DESC、 ASC	ASC、 ASC 或 DESC、 DESC
DESC、 ASC	DESC、 ASC 或 ASC、 DESC	ASC、 ASC 或 DESC、 DESC
DESC、 DESC	DESC、 DESC 或 ASC、 ASC	ASC、 DESC 或 DESC、 ASC

含有两个以上的列的索引遵循与上述规则相同的一般规则。例如，假设您具有以下索引：

```
CREATE INDEX idx_example
 ON table1 (col1 ASC, col2 DESC, col3 ASC)
```

在这种情况下，以下查询可以得到优化：

```
SELECT col1, col2, col3 from table1
 ORDER BY col1 ASC, col2 DESC, col3 ASC
```

和

```
SELECT col1, col2, col3 from example
 ORDER BY col1 DESC, col2 ASC, col3 DESC
```

索引不用于优化在 ORDER BY 子句中具有 ASC 和 DESC 的其它任何模式的查询。例如：

```
SELECT col1, col2, col3 from table1
 ORDER BY col1 ASC, col2 ASC, col3 ASC
```

不会得到优化。

索引的其它用途

Adaptive Server Anywhere 使用索引来实现其它性能优点。通过索引，Adaptive Server Anywhere 可以强制列唯一性，减少必须锁定的行数和页数，以及更好地预计谓语的选择性。

- **强制列唯一性**　如果没有索引，每当插入值时， Adaptive Server Anywhere 都需要扫描整个表以确保该值是唯一的。因此，Adaptive Server Anywhere 会自动在每个具有唯一性约束的列上建立索引。
- **减少锁**　索引可以减少在插入、更新和删除过程中必须锁定的行数和页数。这种减少是索引在表上强制的排序所导致的。

  有关索引和锁定的详细信息，请参见 「锁定的工作方式」第 146 页。
- **预计选择性**　由于索引是经过排序的，优化程序可以通过扫描索引的上层级别来预计满足给定查询的值所占的百分比。这一操作被称作部分索引扫描。

索引的类型

Adaptive Server Anywhere 支持两种索引，并自动根据索引列的声明宽度在这两种索引之间进行选择。对于小于 10 字节的总列宽， Adaptive Server Anywhere 将使用 B 树索引，它包含顺序保留编码或散列值 （它表示带索引的数据）。当索引键的长度大于数据库页面大小的八分之一或 256 字节 （以较小的数字为准）时，也将使用散列 B 树索引。对于声明长度介于这两个界限之间的数据值， Adaptive Server Anywhere 会使用压缩的 B 树索引，它以压缩的形式存储每个键。

索引可以存储为聚簇形式，也可以存储为非聚簇形式。聚簇索引可能有助于提高性能，但在一个表上只能对一个索引进行聚簇。

散列 B 树索引

散列 B 树索引不存储表中的实际行值。散列 B 树索引所存储的是初始数据的顺序保留编码。每个索引条目中用于存储该散列值的字节数被称作散列大小，服务器会自动根据所有索引列的声明宽度选择散列大小。当服务器搜索索引以查找特定行时，将比较这些散列值。

散列值

Adaptive Server Anywhere 必须表示索引中的值以确定如何将这些值排序。例如，如果您为一列名称创建索引，它就必须知道 Amos 位于 Smith 之前。

对于索引中的每个值， Adaptive Server Anywhere 都会创建一个相应的散列值。它所存储的是索引中的散列值，而不是实际值。 Adaptive Server Anywhere 可以用散列值执行运算。例如，它可以确定两个值是否相等或者在两个值中哪一个值更大。

当您为较小的存储类型 （如整数）创建索引时， Adaptive Server Anywhere 创建的散列值将与初始值占用相同的空间。例如，整数的散列值大小为 4 字节，而存储整数需要相同的空间大小。由于散列值具有相同的大小， Adaptive Server Anywhere 可以按与实际值的一一对应关系来使用散列值。 Adaptive Server Anywhere 始终可以通过比较散列值来确定两个值是否相等 （或哪一个值更大）。但是，它只能通过读取相应表中的行来检索实际值。

当您为包含较大数据类型的列创建索引时，散列值的长度通常会小于该类型的大小。例如，如果您为包含字符串值的列创建索引，所用散列值的长度最多为 9 字节。因此， Adaptive Server Anywhere 不能始终仅使用散列值来比较两个字符串。如果散列值相等， Adaptive Server Anywhere 必须从表中的相应行检索两个实际值并对其进行比较。

例如，假设您为书名创建索引，而其中许多书名都类似。如果您要搜索特定的书名，索引可能会只识别一组可能的行。在这种情况下， Adaptive Server Anywhere 必须检索每个候选行并检查全名。

复合索引

列的排序序列也被称作复合索引。但是，这些索引中的每个索引键最多是 9 字节的散列值。因此，散列值不一定能够唯一地标识正确的行。当两个散列值相等时， Adaptive Server Anywhere 必须检索实际值并对其进行比较。

压缩 B 树索引

压缩 B 树索引以压缩形式存储索引内部节点中的每个索引值。为此，压缩 B 树索引会使用 Patricia trie 来存储值， Patricia trie 是一种优化的 trie 数据结构形式，它通过跳跃计数得到扩展，以压缩其表示形式。因此，当总体的数据长度足够大时，压缩 B 树索引将比散列索引提供更大的改进。更为重要的是，压缩算法会有效地处理相等 （或大致相等）的索引值，因此索引值中的通用子字符串对存储要求和性能的影响可以忽略不计。如果索引列的声明宽度之和介于 10 字节和数据库页面大小 （最大为 256 字节）的八分之一之间，则将自动选择压缩 B 树索引。

建议的页面大小

数据库的页面大小对索引条目数有很大的影响。当页面大小加倍时，索引条目数也会近似地加倍。

每一次索引查寻都需要对每个索引级别执行一次页读取并对表页执行一次页读取，而一个查询就可能需要数千次索引查寻。索引条目数较大，通常意味着需要的索引级别较少，这将大大地提高搜索性能。因此，为了提高索引性能，应该使用较大的页面大小 （如 4K）。当您要使用压缩 B 树索引为长字符串列创建索引时，也可能需要使用较大的页面大小，但对较小页面大小的大小限制会防止您创建这些索引。

语义查询转换

为了提高运行效率， Adaptive Server Anywhere 通常会将查询重写为新的形式（可能分为几个步骤）。它将确保以新的方式表示查询的新版本仍会计算出相同的结果。也就是说， Adaptive Server Anywhere 将查询重写为在语义上等效但在语法上不同的形式。

Adaptive Server Anywhere 可以执行大量不同的重写操作。如果您读取访问计划，则经常会发现访问计划并不对应于初始语句的文字解释。例如，优化程序会尽可能多地用连接重写子查询。优化程序可以自由地重写 SQL 语句这一事实以及它重写 SQL 语句的某些方式对您来说都很重要。

示例

与 SQL 语言定义不同，一些语言会对 AND 和 OR 运算强制严格的行为。一些语言可确保将首先对左侧条件求值。如果可以确定整个条件为真，编译器将确保将不会对右侧条件求值。

这样，您就可以将那些在其它情况下需要两个嵌套 IF 语句的条件组合成一个条件。例如，在 C 语言中，您可以按如下方式在使用指针之前测试该指针是否为 NULL。您可以将嵌套条件

```
if ( X != NULL ) {
     if ( X->var != 0 ) {
         ... statements ...
     }
 }
```

替换为更为精简的表达式

```
if ( X != NULL && X->var != 0 ) {
          ... statements ...
 }
```

与 C 语言不同， SQL 语言在执行顺序方面没有的这样规则。 Adaptive Server Anywhere 可以根据需要自由地重新排列这些条件的顺序。由于 SQL 语言规范不区分重新排序后的形式，这些形式在语义上是等效的。特别是，查询优化程序可以任意地将 WHERE、 HAVING 和 ON 子句中的谓语重新排序。

谓语分析

谓语是用逻辑运算符 AND 和 OR 组合一起，在 WHERE、 HAVING 或 ON 子句中组成条件组的条件表达式。在 SQL 中，取值为 UNKNOWN 的谓词被解释为 FALSE。

可以利用索引从表中检索行的谓语被称作**可优化搜索 (sargable)** 谓语。该名称来自于短语 *search argument-able* 包含列与常量、其它列或表达式的比较的谓语可能是 sargable 谓语。

以下语句中的谓语是 sargable 谓语。Adaptive Server Anywhere 可以使用 employee 表的主索引有效地计算该谓语。

```
SELECT *
 FROM employee
 WHERE employee.emp_id = 123
```

```
employee<employee>
```

对比之下，下面的谓语就*不*是可优化搜索谓语。虽然 emp_id 列被编入主索引，但由于结果中包含所有行 （或除一行之外的所有行），因此使用此索引并不会加快计算速度。

```
SELECT *
 FROM employee
 where employee.emp_id <> 123
```

```
employee<seq>
```

同样，没有索引可用于搜索其名字以字母 "k" *结尾*因此，计算此结果的唯一方法是分别检查每一行。

函数

通常，在列名上包含函数的谓语不是 sargable 谓语。例如，在以下查询上将不使用索引：

```
SELECT * from sales_order
 WHERE year(order_date)='2000'
```

您有时可以重写查询以避免使用函数，从而使其成为 sargable 查询。例如，您可以重写以上查询：

```
SELECT * from sales_order
 WHERE order_date > '1999-12-31'
 AND order_date < '2001-01-01'
```

如果您将函数值存储在计算列中并在该列上建立索引，使用函数的查询就会成为 sargable 查询。**计算列**是从表上其它列中获得值的列。例如，如果有一个名为 order_date 的列，它保存订单的日期，您可以创建一个名为 order_year 的计算列，该列保存从 order_date 列中抽取的年份值。

```
ALTER TABLE sales_order
 ADD order_year INTEGER
 COMPUTE year(order_date)
```

然后，可以按普通方式在列 order_year 上添加索引：

```
CREATE INDEX idx_year
 ON sales_order (order_year)
```

如果您执行以下语句

```
SELECT * from sales_order
 WHERE year(order_date) = '2000'
```

服务器会发现存在一个索引列，它包含这一信息并使用该索引来回答查询。

为了进行列替代，计算列的域必须与 COMPUTE 表达式的域相同。在上面的示例中，如果 `year(order_date)` 返回的是字符串而不是整数，优化程序将不会用计算列替代表达式，因此可能没有使用索引 idx_year 来检索所需的行。

有关计算列的详细信息，请参见 「使用计算列」第 54 页。

示例

在下面的每个示例中，属性 *x* 和 *y* 分别是一个表中的各个列。属性 *z* 包含在一个单独的表中。假定下面这些属性中的每一个都带有索引。

Sargable	非 Sargable
x = 10	*x* <> 10
x IS NULL	*x* IS NOT NULL
x > 25	*x* = 4 OR *y* = 5

Sargable	非 Sargable
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'
x = 20 - 2	*x* + 2 = 20

有时，谓语是否是 sargable 谓语可能不很明显。在这些情况下，您可能能够重写该谓语，使其成为 sargable 谓语。对于每一个示例，您都可以根据 "u" 在字母表中是 "t" 后面的下一个字母这一事实来重写谓语 *x* LIKE 'pat%'： *x* >= 'pat' 和 *x* < 'pau'。在这种形式下，属性 *x* 上的索引对于查找有限范围内的值是非常有帮助的。幸运的是， Adaptive Server Anywhere 将自动为您执行这一特定的转换。

用于在表上进行索引检索的 **matching** 谓语是匹配谓语。 where 子句可以包含大量的匹配谓语。到底哪一个匹配谓语最合适，这取决于连接策略。在考虑替代连接策略时，优化程序会重新计算它对匹配谓语做出的选择。

语义转换的类型

为了搜索更有效和更方便的查询表示形式，优化程序可以执行大量的转换。由于优化程序执行了这些转换，计划可能看起来与初始查询的文字解释相当不同。常见的操作包括：

- 排除不必要的 DISTINCT
- 取消子查询嵌套
- 将谓语推入包含 UNION 或 GROUP 的视图
- 排除连接
- MIN 或 MAX 函数优化
- 用 OR 连接， IN 列表优化
- LIKE 优化

- 将外部连接转换为内部连接
- 通过谓语推导发现可利用的条件
- 排除不必要的大小写转换

下面的几个小节将分别讨论上述各项运算。

排除不必要的 DISTINCT

有时 DISTINCT 条件是不必要的。例如，结果中的一个或多个列的属性可能包含 UNIQUE 条件，因为它实际上是主键 — 与这一原因显式或隐式地相关。

示例

由于 product 表包含主键 p.id（它是结果集的一部分），因此以下命令中的 DISTINCT 关键字是不必要的。

```
SELECT DISTINCT p.id, p.quantity
 FROM product p
```

```
p<seq>
```

数据库服务器实际上执行的是语义上等效的查询：

```
SELECT p.id, p.quantity
 FROM product p
```

同样，以下查询的结果包含两个表的主键，因此结果中的每个行肯定是不重复的。

```
SELECT DISTINCT *
 FROM sales_order o JOIN customer c
     ON o.cust_id = c.id
 WHERE c.state = 'NY'
```

```
c<seq> JNL o<ix_sales_cust>
```

取消子查询嵌套

利用 SQL 语言提供的方便语法，您可以将语句表达为嵌套查询。但是，由于 Adaptive Server Anywhere 可以更为有效地利用子查询的 WHERE 子句中具有较高选择性的条件，因此将嵌套查询重写为连接通常会提高执行和优化的效率。

示例

对于外部块中的每个行，以下示例中的子查询最多可以匹配一行。由于该子查询最多可以匹配一行， Adaptive Server Anywhere 将发现可以将该子查询转换为内部连接。

```
SELECT s.*
 FROM sales_order_items s
 WHERE EXISTS
      (    SELECT *
         FROM product p
         WHERE s.prod_id = p.id
             AND p.id = 300 AND p.quantity > 300)
```

在转换之后，相同的语句将使用连接语法在内部进行表达：

```
SELECT s.*
 FROM product p JOIN sales_order_items s
     ON p.id = s.prod_id
 WHERE p.id = 300 AND p.quantity > 20
```

```
p<seq> JNL s<ky_prod_id>
```

同样，以下查询在子查询中包含一个连接性的 EXISTS 谓语。该子查询可以匹配多个行。

```
SELECT p.*
 FROM product p
 WHERE EXISTS
      (    SELECT *
         FROM sales_order_items s
         WHERE s.prod_id = p.id
             AND s.id = 2001)
```

Adaptive Server Anywhere 在 SELECT 列表中使用 DISTINCT 将该查询转换为内部连接。

```
SELECT DISTINCT p.*
 FROM product p JOIN sales_order_items s
     ON p.id = s.prod_id
 WHERE s.id = 2001
```

```
DistI[ s<id_fk> JNL p<product> ]
```

对于外部块中的每一行，如果该子查询最多可以匹配一行，则 Adaptive Server Anywhere 也可以在比较中排除子查询。下面的查询中就存在这种情况。

```
SELECT *
 FROM product p
 WHERE p.id =
      (    SELECT s.prod_id
         FROM sales_order_items s
         WHERE s.id = 2001
             AND s.line_id = 1 )
```

Adaptive Server Anywhere 以如下方式重写该查询。

```
SELECT p.*
 FROM product p, sales_order_items s
 WHERE p.id = s.prod_id
     AND s.id = 2001
     AND s.line_id = 1
```

```
p<seq> JNL s<sales_order_items>
```

在执行取消子查询嵌套的重写优化时，将把 DUMMY 表当作一个特殊的表。即使子查询不相关，也总是会在形式为 `SELECT expression FROM DUMMY` 的子查询上执行子查询展平。

将谓语推入包含 GROUP 或 UNION 的视图

查询经常会限制视图的结果，使其只返回少数几个记录。在视图包含 GROUP BY 或 UNION 的情况下，最好让服务器只计算所需行的结果。

示例 假定我们将视图 product_summary 定义为

```
CREATE VIEW product_summary( product_id, num_orders,
total_qty) as
 SELECT prod_id, count(*), sum( quantity )
 FROM sales_order_items
 GROUP BY prod_id
```

对于所订购的每一件产品，将返回包含该产品的订单数以及所有订单订购的总数。现在假定此视图上有以下查询：

```
SELECT *
 FROM product_summary
 WHERE product_id = 300
```

它将输出限制为产品 ID 为 300 的产品。该查询和视图中的查询可以组合成一个语义上等效的 SELECT 语句，即：

```
SELECT prod_id, count(*), sum( quantity )
 FROM sales_order_items
 GROUP BY prod_id
 HAVING prod_id = 300.
```

此查询最初级的执行计划是计算每一件产品的集合，然后将结果限制为产品 ID 为 300 的那一行。但是，由于 product_id 列是分组列，可以将该列上的 HAVING 谓语推入查询的 WHERE 子句，从而生成

```
SELECT prod_id, count(*), sum( quantity )
 FROM sales_order_items
 WHERE prod_id = 300
 GROUP BY prod_id
```

它会大大地减少所需的计算。如果该谓语具有足够的选择性，则优化程序现在就可以使用 prod_id 上的索引快速地仅检索产品 300 的那些行，而不是按顺序扫描 sales_order_items 表。

相同的优化也用于包含 UNION 或 UNION ALL 的视图。

排除连接

连接排除重写优化会通过从查询中排除表来降低查询的连接度 （如果这样做足够安全）。通常，当查询包含主键 / 外键连接并且查询中只引用了主表中的主键列时，就会使用这种优化。

示例　例如，查询

```
SELECT s.id, s.line_id, p.id
 FROM sales_order_items s KEY JOIN product p
```

将被重写为

```
SELECT s.id, s.line_id, s.prod_id
 FROM sales_order_items s
 WHERE s.prod_id IS NOT NULL.
```

由于 sales_order_items 表中具有引用 product 表的 NULL 外键的所有行都不会出现在结果中，因此第二个查询在语义上与第一个查询等效。

连接排除优化也可以应用于外部连接中包括的表 （虽然使这种优化有效的条件要复杂得多）。在其它特定条件下，主键 / 主键连接中包含的表可能也会成为候选的排除对象。

用户应该注意到，在使用这种优化时， DESCRIBE 的结果可能因为列替代而不同于预期的结果。此外，如果 UPDATE 语句引用一个或多个已排除的基表， UPDATE 或 DELETE WHERE CURRENT 请求就可能会失败。若要回避这一问题，请确保已排除的表中的附加列存在于该查询的 SELECT 列表中 （以便从一开始就避免优化），或者使用单独的语句更新必要的行。

MIN 或 MAX 函数优化

MIN/MAX 重写优化用于利用现有索引有效地计算包括 MAX() 或 MIN() 集合函数的简单集合查询的结果。这种优化的目标是，能够通过使用索引的单行查找来计算出结果。若要成为这种优化的候选对象，查询必须：

- 不能包含 GROUP BY 子句
- 必须位于单个表上
- 不能在 WHERE 子句中包含除连接性等同条件之外的任何条件
- 只能在查询的 SELECT 列表中包含一个集合函数 （MAX 或 MIN）

示例

为了说明此优化，假定 sales_order_items 表中存在 (prod_id ASC, quantity ASC) 上的索引 prod_qty。那么以下查询

```
SELECT MIN( quantity )
 FROM sales_order_items
 Where prod_id = 300
```

在内部被重写为

```
SELECT MIN( quantity )
 FROM ( SELECT FIRST quantity
        FROM sales_order_items
        WHERE prod_id = 300 and quantity IS NOT NULL
        ORDER BY prod_id ASC, quantity ASC ) as
s(quantity)
```

在应用这种优化时，可能不会为集合查询生成 NULL_VALUE_ELIMINATED 警告。

重写后的查询的访问计划（简单形式）为：

```
GrByS[ RL[ sales_order_items<prod_qty> ] ]
```

IN 列表优化

为了利用索引列上的 IN 谓语， Adaptive Server Anywhere 优化程序支持一种特殊的优化。这种优化也同样适用于相同索引列上用 OR 连在一起的多个谓语，因为这两种查询在语义上是等效的。若要能够进行这种优化， IN 列表必须只包含常量。

当优化程序遇到满足条件的 IN 列表谓语，并且该 IN 列表谓语具有足够的选择性来考虑索引检索时，优化程序就会将该 IN 列表谓语转换为嵌套循环连接。以下示例将说明这种优化的工作原理。

假设我们具有以下查询

```
SELECT *
 FROM sales_order
 WHERE sales_rep = 142 or sales_rep = 1596
```

它列出了这两个销售代表的所有订单。该查询在语义上等效于

```
SELECT *
 FROM sales_order
 WHERE sales_rep IN (142, 1596)
```

优化程序预计到该 IN 列表谓语的组合选择性足够高，可以保证索引检索。因此，优化程序会将该 IN 列表当作虚拟表，并基于 sales_rep 属性将该虚拟表连接到 sales_order 表。虽然这种优化的实际效果是在访问计划中包含附加的 [连接]，但查询的连接度不会提高，因此优化时间不应该受到影响。

这种优化具有两项主要的优点。首先，可以将 IN 列表当作 sargable 谓语并利用它来进行索引检索。其次，优化程序可以将 IN 列表排序，以匹配索引的排序序列，从而使检索更为有效。

以上查询的访问计划的简单形式为

```
IN JNL sales_order<ky_so_employee_id>
```

LIKE 优化

常见的 LIKE 谓语通常包含作为文字常量或主机变量的模式。根据 LIKE 谓语包含的模式，优化程序可能会完全重写该 LIKE 谓语，或者用附加的条件 （它们可用来对相应表执行索引检索）扩大该 LIKE 谓语。

示例

在下面的每一个示例中，假定 LIKE 谓语中的模式是文字常量或主机变量， X 是基表中的列。

- `X LIKE '%'` 被重写为 `X IS NOT NULL`
- `X LIKE 'abc'` 被重写为 `X = 'abc'`
- `X LIKE 'abc%'` 用谓语 `X < 'abcZ'` 和 `X > = 'abc_'` 来扩大

其中 Z 和 _ 表示此数据库归类序列的相应高位值和低位值。如果此数据库被配置为存储填补空白的字符串，则第二个比较运算符为 >（而不是 >=），这将确保语义正确。

将外部连接转换为内部连接

在大多数情况下，优化程序将为它的访问计划生成左深 (left-deep) 处理树。这一规则的唯一例外是存在右深 (right-deep) 嵌套外部连接表达式的情况。查询执行引擎的用于计算 LEFT OUTER JOIN 或 RIGHT OUTER JOIN 的算法要求，保留的表必须在任何连接策略中先于提供空

值的表。因此，只要可能，优化程序就会找机会将 LEFT OUTER JOIN 或 RIGHT OUTER JOIN 转换为内部连接，这是因为内部连接是可交换的，它使优化程序在执行连接枚举时具有更大的自由度。

当提供空值的表上不允许空值的谓语出现在查询的 WHERE 子句中时，就可以将 LEFT OUTER JOIN 或 RIGHT OUTER JOIN 转换为内部连接。由于该谓语不允许空值，因此将从结果中排除外部连接生成的任何全为 NULL 的行，从而使该查询在语义上等效于内部连接。

示例

例如，假设有以下查询

```
SELECT *
 FROM product p KEY LEFT OUTER JOIN sales_order_items s
 WHERE s.quantity > 15
```

它旨在列出所有产品以及数量较大的订单； LEFT OUTER JOIN 旨在确保列出所有产品（包括没有订单的产品）。该查询的问题在于，由于谓语 `s.quantity > 15` 将在 `s.quantity` 为 NULL 的情况下被解释为 FALSE， WHERE 子句中的谓语将从结果中排除所有没有订单的产品。因此，该查询在语义上等效于

```
SELECT *
 FROM product p KEY JOIN sales_order_items s
 WHERE s.quantity > 15
```

而且服务器将优化的正是查询的这种重写形式。

在本示例中，查询的编写几乎毫无疑问地存在错误；它应该是

```
SELECT *
 FROM product p
     KEY LEFT OUTER JOIN sales_order_items s
           ON s.quantity > 15
```

这样，数量的测试就是外部连接条件的一部分。

虽然这种优化极少应用于简单的外部连接查询，但它通常适用于这样的情况：查询将引用一个或多个那些用外部连接编写的视图。该查询的 WHERE 子句可以包括限制视图输出的条件，以便从一个或多个表的表达式中排除所有提供空值的行，从而使这种优化能够适用。

发现可利用的条件

任何查询的有效访问策略几乎都依赖于 WHERE/ON/HAVING 子句中是否存在 sargable 条件。只有通过将 sargable 条件作为匹配谓语利用，才有可能进行索引检索。此外，只有当存在等值连接条件时，才可能使用散列、合并和块嵌套循环连接。由于上述原因， Adaptive Server Anywhere 会对初始查询文本中的搜索条件进行详细的分析，以发现可供优化程序利用的简化或隐含条件。

作为一个预处理步骤，视图扩展和合并一旦发生，就会对初始语句中的谓语进行一些简化。例如：

- 如果 X 可为空，`X = X` 会被重写为 `X IS NOT NULL`；否则将排除该谓语。
- `ISNULL(X,X)` 被重写为 `X`。
- 如果 X 是数字列， `X+0` 会被重写为 `X`。
- `AND 1=1` 被排除。
- `OR 1=0` 被排除。
- 由单个元素组成的 IN 列表谓语会被转换为简单的相等条件。

在这一预处理步骤之后， Adaptive Server Anywhere 会尝试将初始搜索条件规范化为连接性规范形式 (CNF)。若要使表达式成为 CNF 的形式，该表达式中的每一项必须用 AND 连起来。每一项都由单个原子条件或一组用 OR 连起来的条件组成。

如果将任意条件转换为 CNF，所生成的表达式可能具有相似的复杂性，但也会包含大得多的条件集。 Adaptive Server Anywhere 将发现这种情况，并避免不经分析就将条件转换为 CNF。实际上， Adaptive Server Anywhere 会分析初始表达式以查找初始搜索条件所暗示的可利用谓语，并且用 AND 将这些推导出的条件与查询连起来。如果完全的规范化需要复制占用大量资源的谓语 （例如定量的子查询谓语），则还将避免完全的规范化。但是，只要可行，该算法就会将 IN 列表谓语合并起来。

将搜索条件完全规范化或找到可利用的条件后，优化程序会执行传递性分析，以发现传递等同条件 （主要是传递连接条件和包含常量的条件）。这样，在基于成本的优化阶段中执行连接枚举时，优化程序将具有更大的自由度，因为这些传递条件允许附加的替代连接顺序。

示例

假定初始查询为

```
SELECT e.emp_lname, s.id, s.order_date
 FROM sales_order s, employee e
 WHERE (e.emp_id = s.sales_rep and
           (s.sales_rep = 142 or s.sales_rep = 1596)
       )
          OR
       ( e.emp_id = s.sales_rep and s.cust_id = 667)
```

该查询不包含连接性等值连接条件；因此，如果不进行详细的谓语分析，优化程序将无法发现有效的访问计划。幸运的是， Adaptive Server Anywhere 能够将整个表达式转换为 CNF，从而生成等效的查询

```
SELECT e.emp_lname, s.id, s.order_date
 FROM sales_order as s, employee as e
 WHERE e.emp_id = s.sales_rep AND
     (s.sales_rep = 142 or s.sales_rep = 1596 or
s.cust_id = 667)'
```

该查询现在可以作为内部连接查询进行有效的优化。

排除不必要的大小写转换

缺省情况下， Adaptive Server Anywhere 数据库支持不区分大小写的字符串比较。有时，优化程序可能会遇到这样的查询：用户使用 UPPER() 或 LOWER() 内置函数在这些查询中显式强制进行不必要的文本转换。如果数据库的归类序列允许， Adaptive Server Anywhere 将自动排除这种不必要的转换。排除大小写转换后，就可以使用比较谓语对相应表进行索引检索。

示例

在不区分大小写的数据库上，以下查询

```
SELECT *
 FROM customer
 WHERE UPPER(lname) = 'SMITH'
```

在内部被重写为

```
SELECT *
 FROM customer
 WHERE lname = 'SMITH'
```

优化程序现在可以考虑使用 customer.lname 上的索引。

子查询和函数高速缓存

Adaptive Server Anywhere 在处理子查询时会高速缓存结果。这种高速缓存是一个请求一个请求地执行的；并发请求或多个连接决不会共享已高速缓存的结果。如果 Adaptive Server Anywhere 需要为同一组相关值重新计算子查询，它就可以直接从高速缓存中检索结果。如此一来，Adaptive Server Anywhere 就可以避免许多重复和多余的计算。当请求完成 （查询的游标关闭）时， Adaptive Server Anywhere 就会释放被高速缓存的值。

在进行查询处理时， Adaptive Server Anywhere 会监视被高速缓存的子查询值的重用频率。如果相关变量的值极少重复， Adaptive Server Anywhere 就只需要将大多数的值计算一次。在这种情况下， Adaptive Server Anywhere 会发现重新计算偶尔出现的重复值比高速缓存大量只出现一次的条目更为有效。因此，服务器将在语句的剩余部分挂起该子查询的高速缓存，而为外部查询块中的每个行重新计算该子查询。

如果相关列的大小大于 255 字节， Adaptive Server Anywhere 也不会进行高速缓存。在这种情况下，您可能需要重写查询或者将另一列添加到表中，以提高这些操作的效率。

函数高速缓存

一些内置函数和用户定义函数的高速缓存方式与子查询结果的高速缓存方式相同。这样就可以大大改进那些在查询处理过程中用相同参数调用的且占用大量资源的函数。但是，这可能意味着函数的调用次数少于其它情况下的预期调用次数。

对于要高速缓存的函数，它必须满足两个条件：

- 对于一组给定的参数，它必须始终返回相同的结果。
- 它必须对基础数据没有副面影响。

满足这些条件的函数称为**确定型**或**等幂**函数。

除了少数几个例外，内置函数均被视为确定型函数。 RAND、 NEW_ID 和 GET_IDENTITY 函数被视为非确定型函数，它们的结果不进行高速缓存。

除非在创建用户定义函数时将其指定为 NOT DETERMINISTIC，否则会将视其为确定型函数。

有关用户定义的函数的详细信息，请参见 『ASA SQL 参考』>「CREATE FUNCTION 语句」。

读取访问计划

优化程序可以告诉您它为响应任何语句而选择的查询优化策略（计划）。

优化程序的工作是理解查询的语义并构建用于计算查询结果的计划。该计划可能与您使用的语法并不完全对应。优化程序可以自由地以语义上等效的任意形式重写查询。

有关 Adaptive Server Anywhere 在重写查询时所遵循的规则的详细信息，请参见「将子查询重写为 EXISTS 谓语」第 434 页和「语义查询转换」第 466 页。

有关优化程序用来实现查询的方法的信息，请参见「查询执行算法」第 439 页。

您可以在 Interactive SQL 中查看计划，也可以使用 SQL 函数查看计划。您可以选择检索以下几种不同格式的访问计划：

- 简单计划
- 详细计划
- 图形计划
- 含统计信息的图形计划
- UltraLite （简单、详细或图形）

也可以用特定游标类型获得 SQL 查询的计划。

有关如何访问计划的详细信息，请参见「对计划进行访问」第 499 页。有关如何读取计划的信息，请参见「文本计划」第 490 页和「图形式计划」第 493 页。

下面将介绍访问计划中显示的统计信息以及其它项。

计划中使用的缩写

下面是简单计划以及图形式计划的简称形式中使用的缩写：

名称	简单计划 / 简称
排除所有散列	EAH
排除散列	EH
散列分组依据	GrByH
散列表扫描	HTS
聚簇散列分组依据	GrByHClust
散列汇总分组依据	GrByHR
排序分组依据	GrByO
排序汇总分组依据	GrByOR
单行分组依据	GrByS
索引分组依据	GrByI
非重复散列	DistH
非重复索引	DistI
非重复排序	DistO
从顶部 N 排序	StrN
散列过滤	HF
交叉所有散列	IAH
交叉散列	IH
存在连接	JE
嵌套循环半连接	JNLS
散列存在	JHE
散列不存在	JHNE
散列连接	JH
排序块	SrtBl
左外散列连接	JHO
完全外散列连接	JHFO

名称	简单计划 / 简称
递归散列连接	JHR
左外递归散列连接	JHRO
嵌套块连接	JNB
左外部嵌套块连接	JNBO
不存在连接	JNE
嵌套循环连接	JNL
左外部嵌套循环连接	JNLO
完全外部嵌套循环连接	JNLFO
合并连接	JM
左外部合并连接	JMO
完全外部合并连接	JMFO
排除合并	EM
排除所有合并	EAM
交叉合并	IM
交叉所有合并	IAM
行限制	RL
行重复	RR
派生表	DT
递归表	RT. 在简单计划中是 rt<seq>
递归联合	RU
Union all	UA
表扫描	In short plan is *tablename*<seq>. 在图形式计划中就只是表名。
索引扫描	在简要计划中是 *tablename*<*indexname*>。在图形式计划中就只是表名。
IN 列表	IN

有关算法的说明，请参见 「查询执行算法」第 439 页。

计划中使用的常见统计信息

下面的统计信息是实际的测量值。

统计	解释
Invocations	从子树中请求行的次数。
RowsReturned	为当前节点返回的行数。
RunTime	执行子树所需的时间 （包括执行子项所需的时间）。
CacheHits	成功读取高速缓存的次数。
CacheRead	已经在高速缓存中查找的数据库页数。
CacheReadTable	已经从高速缓存中读取表页数。
CacheReadIndLeaf	已经从高速缓存中读取的索引叶页数。
CacheReadIndInt	已经从高速缓存中读取的索引内部节点页数。
DiskRead	已经从磁盘中读取的页数。
DiskReadTable	已经从磁盘中读取的表页数。
DiskReadIndLeaf	已经从磁盘中读取的索引叶页数。
DiskReadIndInt	已经从磁盘中读取的索引内部节点页数。
DiskWrite	已经写入磁盘的页数 （工作表页数或修改表页数）。
IndAdd	已经添加到索引中的条目数。
IndLookup	已经在索引中查找的条目数。
FullCompare	已经在索引中的散列值之上执行的比较次数。

计划中的常见预计值

统计	解释
EstRowCount	节点在每次被调用时将返回的预计行数。

统计	解释
AvgRowCount	每次调用时返回的平均行数。这不是预计值，而是按照 RowsReturned / Invocations 进行计算。如果该值与 EstRowCount 之间的差值很大，则说明选择性预计值可能会不正确。
EstRunTime	执行所需的预计时间 （EstDiskReadTime、EstDiskWriteTime 和 EstCpuTime 的总和）。
AvgRunTime	执行所需的平均时间 （测量值）。
EstDiskReads	从磁盘中读取的预计次数。
AvgDiskReads	从磁盘中读取的平均次数 （测量值）。
EstDiskWrites	写入磁盘的预计次数。
AvgDiskWrites	写入磁盘的平均次数 （测量值）。
EstDiskReadTime	从磁盘中读取行所需的预计时间。
EstDiskWriteTime	将行写入磁盘所需的预计时间。
EstCpuTime	执行所需的预计处理器时间。

计划中与 SELECT、INSERT、UPDATE 和 DELETE 相关的项

项	解释
优化目标	确定优化查询处理的意图：是迅速返回第一行，还是为最大程度地降低返回整个结果集的成本。 请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_GOAL 选项 [database]」。
优化负载	确定优化查询处理的目标是针对更新和读取混合进行的负载还是针对主要基于读取的负载。 请参见 『ASA 数据库管理指南』> 「OPTIMIZATION_WORKLOAD 选项 [database]」。

项	解释
ANSI 更新约束	控制所允许的更新范围 （选项包括 OFF、 CURSORS 和 STRICT）。 请参见 『ASA 数据库管理指南』> 「ANSI_UPDATE_CONSTRAINTS 选项 [compatibility]」
优化级别	留作将来使用。
选择列表	查询所选择的表达式的列表。

计划中与锁相关的项

项	解释
被锁定的表	所有被锁定的表及其关联的隔离级别的列表。

计划中与扫描相关的项

项	解释
表名	表的实际名称。
相关名	表的别名。
预计行数	表中的预计行数。
预计页数	表中的预计页数。
预计行大小	表的预计行大小。
页位置图	在使用页位置图来读取多页时为 [是]。

计划中与索引扫描相关的项

项	解释
索引名	索引的名称。

项	解释
键类型	可以是 PRIMARY KEY （主键）、FOREIGN KEY （外键）、CONSTRAINT （唯一约束）或 UNIQUE （唯一索引）之一。如果索引是非唯一的辅助索引，则不会显示键类型。
深度	索引的高度。 有关详细信息，请参见 「表大小和页面大小」第 456 页。
预计叶页数	预计的叶页数。
基数	索引的基数 （如果它不同于预计行数）。它仅适用于 Adaptive Server Anywhere 数据库版本 6.0 和更早版本。
选择性	匹配域范围的预计行数。
方向	FORWARD （向前）或 BACKWARD （向后）。
域范围	域范围显示为列表 (col_name=value) 或 col_name IN [low, high]。

计划中与连接、过滤器和预过滤器相关的项

项	解释
谓语	在此节点中求出的搜索条件以及选择性预计值和测量值。 有关详细信息，请参见 「计划中的选择性」第 498 页

计划中与散列过滤相关的项

项	解释
生成值	输入中的不重复值的预计数。
探测值	查看谓语时输入中的不重复值的预计数。
位	选择生成散列映射的位数。
页数	存储散列映射所需的页数。

计划中与联合相关的项

项	解释
联合列表	UNION 操作中涉及的列。

计划中与 GROUP BY 相关的项

项	解释
集合	所有集合函数。
GROUP BY 列表	GROUP BY 子句中的所有列。

计划中与 DISTINCT 相关的项

项	解释
DISTINCT 列表	DISTINCT 子句中的所有列。

计划中与 IN 列表相关的项

项	解释
IN 列表	指定集合中的所有表达式。
表达式 SQL	要与该列表进行比较的表达式。

计划中与 SORT 相关的项

项	解释
Order-by	列出要作为排序依据的所有表达式。

计划中与行限制相关的项

项	解释
行限制计数	按照 FIRST 或 TOP n 的指定所返回的最大行数。

文本计划

文本计划有两种类型：简要计划和详细计划。若要在 Interactive SQL 中选择计划类型，请从 [工具] 菜单中打开 [选项] 对话框，然后单击 [计划] 选项卡。要使用 SQL 函数访问计划，请参见 「使用 SQL 函数来对计划进行访问」第 500 页。

用冒号来分隔连接策略

下面的命令包含两个**查询块**：sales_order 和 sales_order_items 表中的外部 select 语句，及从 product 表中进行选择的子查询。

```
SELECT *
 FROM sales_order AS o
     KEY JOIN sales_order_items AS i
 WHERE EXISTS
     (    SELECT *
         FROM product p
         WHERE p.id = 300 )
```

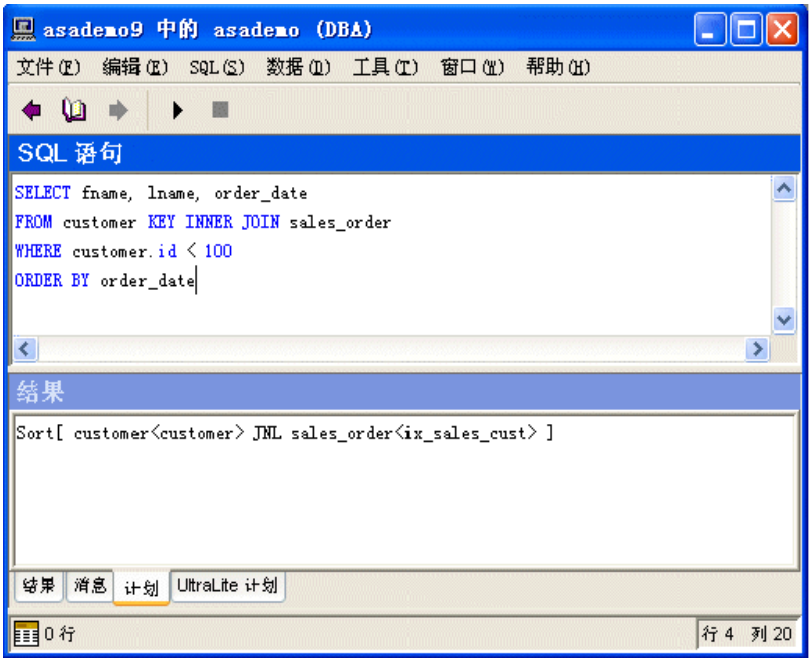
```
i<seq> JNL o<sales_order> : p<seq>
```

用冒号来分隔连接策略。计划总是会首先列出主块的连接策略，然后才列出其它查询块的连接策略。其它查询块的连接策略的顺序可能既不与语句中的顺序相对应，也不与它们的执行顺序相对应。

简单文本计划

当您要快速比较计划时，简单计划将非常有用。它所提供的信息量在所有访问计划格式中是最少的，但它仅在一行上提供这些信息。

在下面的示例中，由于 ORDER BY 子句使整个结果集进行排序，计划以 SORT 一词开头。对 customer 表的访问是通过它的主键索引 （也被称作 customer）进行的。因为列 customer.id 是主键，所以将使用索引扫描来满足搜索条件。缩写 JNL 表示优化程序正在使用嵌套循环连接来处理查询。最后，使用外键索引 ky_so_customer 对 sales_order 表进行访问，以查找 customer 表中的匹配行。



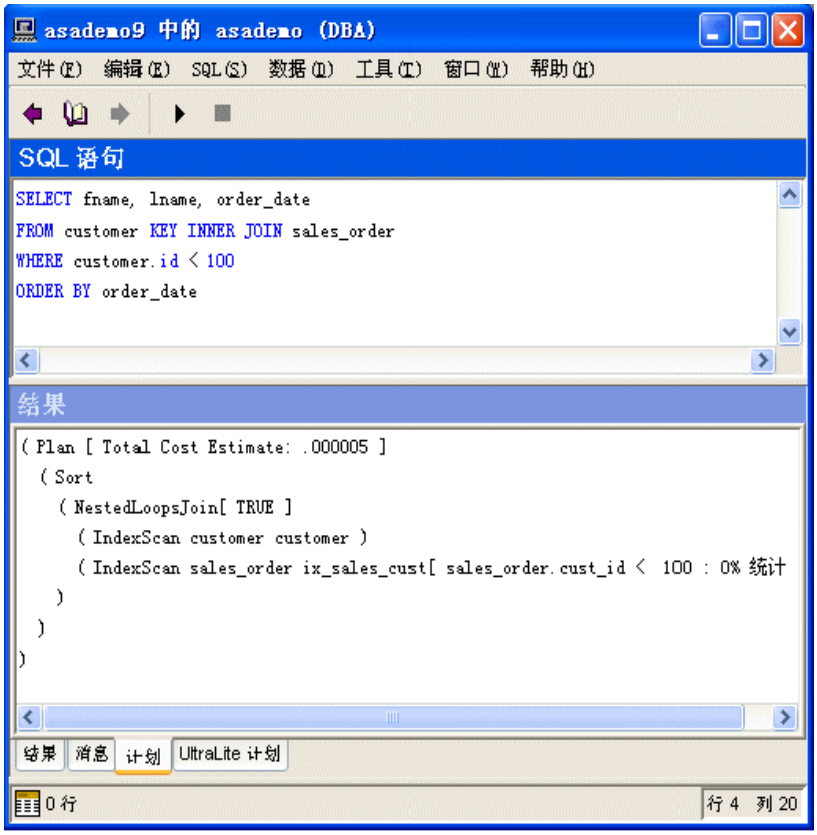
有关计划中使用的代码字的详细信息，请参见 「计划中使用的缩写」第 482 页。

详细文本计划

详细计划提供的信息略多于简单计划，而且这些信息以便于打印和查看（无需向下滚动）的形式提供。

在同一查询的详细计划输出中，第一行为 `Plan [ I/O Estimate: 1 ]`。 Plan （或 Sub-plan）一词指示查询块的开头 （在本例中只有一个查询块）。 I/O 将预计查询所需的 I/O 数 （在本例中为一）。同样，该计划会指出结果已排序并且嵌套循环连接是将要使用的连接算法。 TRUE 一词 （或搜索条件）和算法的选择性预计

值 （本例中没有）将与该算法出现在同一行上。 WHERE 条件在以 FILTER 开头的行上提供，后接搜索条件、搜索条件的选择性预计值以及选择性预计值的来源。



有关计划中使用的代码字的详细信息，请参见 「计划中使用的缩写」第 482 页。

图形式计划

图形式计划有两种类型：图形式计划和带有统计信息的图形式计划。若要在 Interactive SQL 中选择计划类型，请从 [工具] 菜单中打开 [选项] 对话框，然后单击 [计划] 选项卡。要使用 SQL 函数访问计划，请参见「使用 SQL 函数来对计划进行访问」第 500 页。

当图形式计划显示出来后，您可以通过右击左窗格并选择 [自定义] 来配置它的显示方式。

您可以将图形式计划打印出来以供以后参考。要打印计划，请右击某个节点，然后选择 [打印]。

若要获取有关图形式计划中每个节点的上下文相关帮助，请选择该节点，在其上右击，然后选择 [帮助]。例如，右击 [嵌套循环连接] 并选择 [帮助] 以获取有关该执行部分使用的资源的信息。另外，若将鼠标光标悬停在图形式计划中的每个元素之上，也可以获取弹出式信息。

图形式计划

图形式计划提供的信息远远多于简单计划或详细计划提供的信息。您可以选择查看图形式计划或含统计信息的图形式计划。通过这两种计划，您都可以快速地查看出计划的哪些部分被预计会占用最多的资源。含统计信息的图形式计划虽然在查看时会占用较多的资源，但也会提供在查询执行过程中服务器监测到的实际查询执行统计信息，并且允许直接比较查询优化程序在构建访问计划时使用的预计值和在执行过程中监测到的实际统计信息。但是，请注意，优化程序通常无法精确地预计查询的成本，因此应该会存在差异。图形式计划是访问计划的缺省格式。

图形式计划旨在以直观方式提供一些关键信息：

- 图形式计划中显示的每个操作都会显示在一个容器中。这个容器表明操作是否会实现数据、是否是索引扫描、是否是表扫描，或者是否为某种其它操作。
- 计划中一个操作传递到下一个操作的行数由连接操作的线条的粗细表示。这样，便为对查询中的多数数据执行的操作提供了一个直观指示器。
- 速度特别慢的操作的容器会带有红色边框。

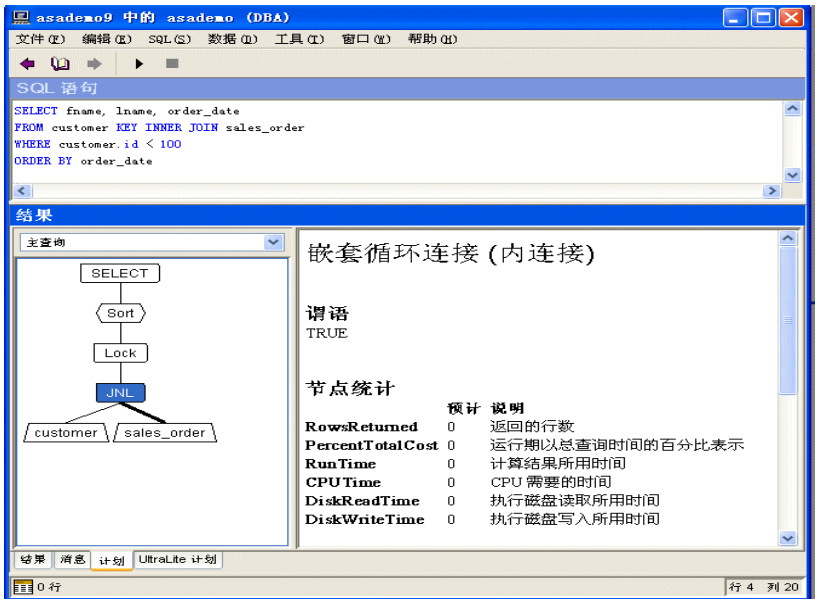
通常，执行粗线条上以及带红框的子树中的节点最有可能决定查询性能。

这些显示功能中的每一个都是可以自定义的。

如果一个查询被识别为简单查询，则会绕过某些优化步骤，并且 [查询优化程序] 部分和 [谓语] 部分都不会显示在图形式计划中。有关简单查询的详细信息，请参见 「优化程序的工作原理」第 429 页。

下面是曾用来描述简单和详细文本计划的同一查询，现在用图形式计划来表示该查询。以下图示采用了树的形式，表示每个节点从它下面的节点请求行。 [锁] 节点表示结果集已实现或者行已返回到应用程序。在本例中，排序要求将结果实现。在级别 0，不会实际锁定行：Adaptive Server Anywhere 只是确保从基表中读取行以后没有将行删除。在级别 1，行仅被锁定到访问下一行时为止。在级别 2 和级别 3，将应用读取锁定并一直保持到发出 COMMIT 命令为止。

您可以通过单击图示中的节点来获取有关计划中的节点的详细信息。在本例中选择了嵌套循环连接节点。右窗格中的信息仅与该节点相关。例如，谓语说明只有 TRUE，表示在查询执行的这一阶段，没有应用任何谓语 （查询中的唯一谓语 (customer.id < 100) 应用在 customer 表查询节点中）。



有关计划中使用的缩写的详细信息，请参见 「计划中使用的缩写」第 482 页。

含统计信息的图形式计划

含统计信息的图形式计划不仅可以向您显示随图形式计划提供的所有预计值，而且还可显示执行语句的实际运行时成本。为此，必须实际执行该语句。这意味着在访问占用大量资源的查询的计划时存在延迟。还意味着任何查询部分 （如删除或更新）都要实际执行，虽然您可以执行回退来撤消这些更改。

当您遇到性能问题并且预计行计数或运行时间与您的预期值不同时，可以使用带有统计信息的图形式计划。含统计信息的图形式计划会提供预计值和实际统计信息供您比较。如果实际值和预计值之间存在较大的差异，则表明要注意优化程序可能没有足够的信息来选择好的访问计划。

影响查询执行的数据库选项和其它全局设置只对根运算符显示。

下面是您可以在含统计信息的图形式计划中检查的一些关键统计信息以及一些可能的补救方法：

- **选择性统计信息** 谓语的选择性 （条件表达式）是满足条件的行的百分比。预计的谓语的选择性为优化程序预计其成本提供了依据信息。如果选择性预计值较低，查询优化程序生成的访问计划可能会较差。例如，如果优化程序错误地将某一谓语的选择性预计得很高 （例如 5% 的选择性），但实际的选择性低得多 （例如 50%），则它就可能会选择不适当的计划。选择性预计值不精确，但是重大的误差则表示确实有潜在问题。

  如果您确定查询主要部分的选择性信息不准确，则可以使用 CREATE STATISTICS 为所讨论的列生成一组新的统计信息。在极少数情况下，您可能希望提供显式选择性预计，尽管它们会带来潜在的维护问题。

  有关选择性的详细信息，请参见 「计划中的选择性」第 498 页。

  有关创建统计信息的详细信息，请参见 『ASA SQL 参考』> 「CREATE STATISTICS 语句」。

  有关用户预计值的详细信息，请参见 『ASA SQL 参考』> 「显式选择性估计」。

  如果查询被确定为是简单查询，则不会显示选择性统计信息，这样将绕过优化程序。有关简单查询的详细信息，请参见 「优化程序的工作原理」第 429 页。

  选择性低的指示符在以下位置出现：

- **RowsReturned 的实际值和预计值** RowsReturned 是结果集中的行数。 RowsReturned 统计信息显示在树顶部的根节点的表中。如果预计的返回行数与实际返回的行数存在重大差异，则表明需要注意优化程序正在处理选择性很低的信息。

- **谓语选择性的实际值和预计值** 查找**谓语**的小标题以查看谓语选择性。有关读取谓语信息的信息，请参见 「计划中的选择性」第 498 页。

  如果该谓语位于没有直方图的基列之上，则通过执行 CREATE STATISTICS 语句创建直方图就可以更正问题。如果选择性错误仍然是问题，作为最后一招，您可以考虑随查询文本中的谓语一起指定选择性的用户预计值。

- **预计值来源** 统计信息窗格中的 [谓语] 小标题下也列出了选择性预计值的来源。有关读取谓语信息的信息，请参见 「计划中的选择性」第 498 页。

  [推测] 的预计值来源表明优化程序没有选择性信息可供使用。如果预计值来源是 [索引] 并且选择性预计值不正确，则您的问题可能在于索引分布不均匀：使用 REORGANIZE TABLE 语句对索引进行碎片整理可能会有用。

  有关选择性预计值的可能来源的完整列表，请参见 『ASA SQL 参考』> 「ESTIMATE_SOURCE 函数 [杂类]」。

- **高速缓存读取次数和命中次数** 如果高速缓存读取次数和高速缓存命中次数完全相同，则执行查询所需的所有信息都在高速缓存中 - 这太好了。当读取次数大于命中次数时，则意味着服务器在尝试从高速缓存读取时失败，它必须从磁盘中读取。在某些情况下 （如散列连接），这是您所需要的。而在另外一些情况下 （如嵌套循环连接），较低的高速缓存命中率可能说明存在性能问题，而您可以通过增加高速缓存大小来提高性能。

  有关高速缓存管理的详细信息，请参见 「增加高速缓存大小」第 175 页。

- **缺少适当的索引** 在查询执行计划中，通常很难看出一个索引是否可以帮助提供更好的性能。 Adaptive Server Anywhere 中使用的一些基于扫描的算法可在不使用索引的情况下为许多查询提供极好的性能。

如果您猜测其它查询可以帮助为查询提供更好的性能，请使用 [索引顾问]。有关详细信息，请参见 「索引顾问概述」第 73 页。

有关索引和性能的详细信息，请参见 「最佳性能提示」第 175 页和 「使用索引」第 177 页。

- **数据碎片问题** 运行时间的实际值与预计值在根节点统计信息中提供。运行时间表示的是查询的执行时间。如果运行时间对于表扫描或索引扫描不正确，则可以通过执行 REORGANIZE TABLE 语句来提高性能。

  有关详细信息，请参见 『ASA SQL 参考』> 「REORGANIZE TABLE 语句」和 「检查文件、表和索引碎片」第 185 页。

下面是含统计信息的图形式计划的示例。同样，所选择的是嵌套循环连接节点。右窗格中的统计信息指出了该查询部分所使用的资源。



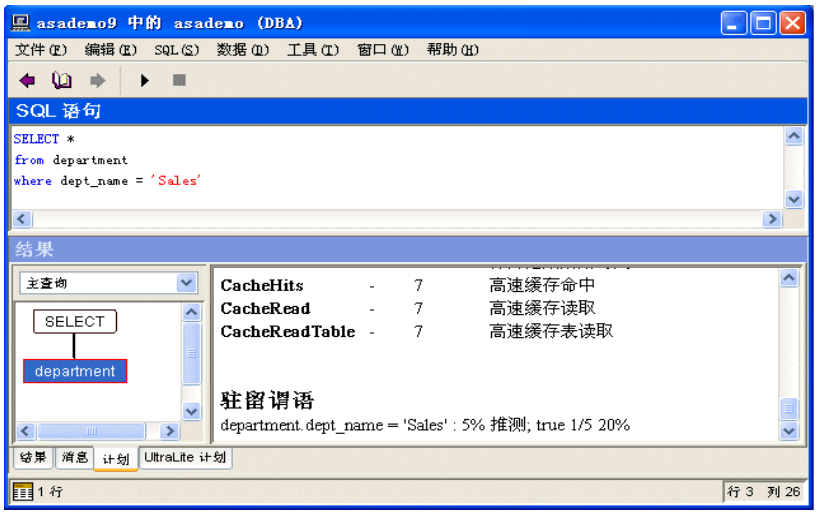
有关计划中使用的代码字的详细信息，请参见 「计划中使用的缩写」第 482 页。

计划中的选择性

下面是一个谓语的示例，该示例显示了搜索条件的选择性。在该示例中，所选择的节点代表 department 表的扫描，统计信息窗格将谓语显示为搜索条件和选择性统计信息。

不会为那些绕过了某些优化步骤的简单查询显示选择性信息。有关简单查询的详细信息，请参见 「优化程序的工作原理」第 429 页。

访问计划取决于数据库中可用的统计信息，而统计信息又取决于以前执行了什么查询。在下面显示的内容中，您可以看到不同的统计信息和计划。



该谓语说明为

```
department.dept_name = 'Sales' : 5% Guess; true 1/5 20%
```

它可以按如下方式来理解：

- `department.dept_name = 'Sales'` 是谓语。
- 5% 是优化程序的选择性预计值。也就是说，优化程序将它的查询访问选择基于该预计值 （5% 的行满足该谓语）。

  这一输出与 ESTIMATE 函数提供的输出相同。有关详细信息，请参见 『ASA SQL 参考』> 「ESTIMATE 函数 [Miscellaneous]」。

- `Guess` 是预计值来源。这一输出与 ESTIMATE_SOURCE 函数提供的输出相同。有关选择性预计值的可能来源的完整列表，请参见『ASA SQL 参考』> 「ESTIMATE_SOURCE 函数 [杂类]」。

- `true 1/5 20%` 是执行期间谓语的实际选择性。对谓语进行了五次预计，有一次预计选择性为 20%，是正确的。

  如果实际选择性与预计值有很大差异，并且如果对谓语进行了许多次预计，则不正确的预计值可能会导致查询性能出现重大问题。收集有关谓语的统计信息可为优化程序提供有关以何作为选择依据的更佳信息，从而可以提高性能。

注意：
如果您选择的是图形式计划而不是含统计信息的图形式计划，则最后两项统计信息不会显示出来。

对计划进行访问

您可以在 Interactive SQL 中或使用 SQL 函数来查看访问计划。从 [索引顾问] 中也可以看到该计划

在 Interactive SQL 中对计划进行访问

Interactive SQL 中提供了下列类型的计划：

- **简要计划**
- **详细计划**
- **图形式计划**
- **含统计信息的图形式计划**

若要选择您需要查看的计划类型，请单击 [工具] → [选项] 并选择 [计划] 选项卡。

若要查看计划，请执行一个查询，然后打开 [计划] 选项卡。 [计划] 选项卡位于 Interactive SQL 窗口的底部。

UltraLite 计划

您也可以查看您的查询的 UltraLite 计划。 UltraLite 计划不包括统计信息。

若要在 Interactive SQL 中查看 UltraLite 计划，请从 [工具] 菜单中打开 [选项] 对话框，然后选择 [显示 UltraLite 计划]。缺省情况下，该选项将处于选中状态。您可以通过选择上述类型之一（图形式计划、简单计划或详细计划）来控制 UltraLite 计划类型。如果您选择含统计信息的图形式计划，则您将看到图形式计划。

[UltraLite 计划] 选项卡将会出现在 Interactive SQL 窗口的底部。对于某些查询， UltraLite 执行计划可能会与为 Adaptive Server Anywhere 选择的计划有所不同。

有关详细信息，请参见 『ASA SQL 参考』> 「GRAPHICAL_ULPLAN 函数 [Miscellaneous]」。

使用 SQL 函数来对计划进行访问

您可以使用 SQL 函数来对计划进行访问并检索 XML 格式的输出。

- 要访问简单计划，请参见 『ASA SQL 参考』> 「EXPLANATION 函数 [Miscellaneous]」。
- 要访问详细计划，请参见 『ASA SQL 参考』> 「PLAN 函数 [Miscellaneous]」。
- 要访问图形式计划，请参见 『ASA SQL 参考』> 「GRAPHICAL_PLAN 函数 [Miscellaneous]」。
- 要访问 UltraLite 计划，请参见 『ASA SQL 参考』> 「GRAPHICAL_ULPLAN 函数 [Miscellaneous]」、『ASA SQL 参考』> 「SHORT_ULPLAN 函数 [Miscellaneous]」或 『ASA SQL 参考』> 「LONG_ULPLAN 函数 [Miscellaneous]」。

第 3 部分　SQL 方言和兼容性

这一部分介绍了 Transact-SQL 的兼容性以及其它 SQL 实现通常所不具有的 Adaptive Server Anywhere 的那些功能。

第 14 章

Transact-SQL 兼容性

关于本章

Transact-SQL 是 Sybase Adaptive Server Enterprise 所支持的 SQL 方言。

本章是有关创建与 Adaptive Server Anywhere 和 Adaptive Server Enterprise 均兼容的应用程序的指南。它将介绍 Adaptive Server Anywhere 对 Transact-SQL 语言元素和语句的支持以及对 Adaptive Server Enterprise 系统表、视图和过程的支持。

Transact-SQL 支持概述

Adaptive Server Anywhere 支持 **Transact-SQL** 中相当大的一部分，Transact-SQL 是 Sybase Adaptive Server Enterprise 所支持的 SQL 方言。本章将介绍 Adaptive Server Anywhere 和 Adaptive Server Enterprise 之间的 SQL 兼容性。

目标

Adaptive Server Anywhere 对 Transact-SQL 的支持可以达到以下目的：

- **应用程序可移植性** 所编写的许多应用程序、存储过程和批处理文件都可以与 Adaptive Server Enterprise 数据库和 Adaptive Server Anywhere 数据库一起使用。
- **数据可移植性** Adaptive Server Anywhere 和 Adaptive Server Enterprise 数据库之间可以轻松地相互进行数据交换和复制。

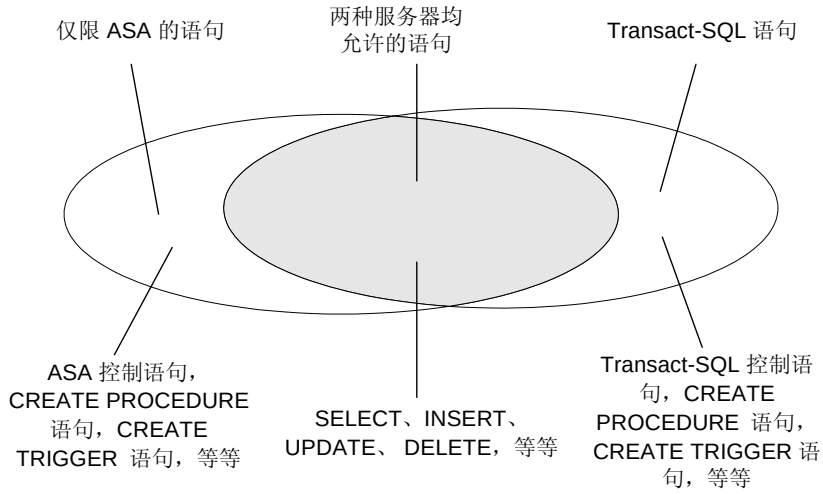
主要的目标是编写既可用于 Adaptive Server Enterprise 又可用于 Adaptive Server Anywhere 的应用程序。现有的 Adaptive Server Enterprise 应用程序通常需要做出一些更改才能在 Adaptive Server Anywhere 数据库上运行。

怎样才能支持 Transact-SQL

Adaptive Server Anywhere 通过以下形式支持 Transact-SQL：

- 许多 SQL 语句在 Adaptive Server Anywhere 和 Adaptive Server Enterprise 之间兼容。
- 对于某些语句（尤其是在过程、触发器和批处理语句中使用的过程语言），将支持单独的 Transact-SQL 语句以及 Adaptive Server Anywhere 的先前版本中所支持的语法。对于这些语句， Adaptive Server Anywhere 支持 SQL 的两种**方言**。本章将这两种方言称作 Transact-SQL 和 Watcom-SQL。
- 过程、触发器或批处理语句将以 Transact-SQL 或 Watcom-SQL 方言执行。在整个批处理语句或过程中，必须仅使用一种方言的控制语句。例如，每种方言都有不同的流控制语句。

下图说明了这两种方言的重叠情况。



相似之处和差异

在处理现有数据时， Adaptive Server Anywhere 支持大多数的 Transact-SQL 语言元素、函数和语句。例如， Adaptive Server Anywhere 支持所有数字函数、所有字符串函数 （一种函数除外）、所有集合函数以及所有日期和时间函数。又比如， Adaptive Server Anywhere 支持 Transact-SQL 外部连接 （使用 =* 和 *= 运算符），还支持使用连接的 DELETE 和 UPDATE 扩展语句。

另外， Adaptive Server Anywhere 还支持大多数的 Transact-SQL 存储过程语言 （CREATE PROCEDURE 和 CREATE TRIGGER 语法、控制语句等）以及 Transact-SQL 数据定义语言语句的许多 （但不是所有）方面。

每种产品所支持的体系结构和配置功能存在着设计上的差异。设备管理、用户管理和维护任务 （如备份）往往是因系统而异的。即使在这种情况下， Adaptive Server Anywhere 仍将 Transact-SQL 系统表作为视图来提供 （在视图中，在 Adaptive Server Anywhere 中没有任何意义的表将不显示任何行）。此外， Adaptive Server Anywhere 为某些常见的管理任务提供了一组系统过程。

在讨论这些方言的数据操作和数据定义语言等兼容性高的方面之前，本章将首先讨论一些差异最为明显的系统层面上的问题。

仅限 Transact-SQL

Adaptive Server Anywhere 所支持的某些 SQL 语句只在一种方言中使用，而不在另一种方言中使用。您不能在一个过程、触发器或批处理语句中混用两种方言。例如， Adaptive Server Anywhere 支持以下语句，但这些语句仅用于 Transact-SQL 方言：

- Transact-SQL 控制语句 IF 和 WHILE
- Transact-SQL 的 EXECUTE 语句
- Transact-SQL 的 CREATE PROCEDURE 和 CREATE TRIGGER 语句
- Transact-SQL 的 BEGIN TRANSACTION 语句
- Transact-SQL 过程或批处理语句中可以使用*未*以分号分隔的 SQL 语句

仅限 Adaptive Server Anywhere

Adaptive Server Enterprise 不支持以下语句：

- 控制语句 CASE、 LOOP 和 FOR
- IF 和 WHILE 的 Adaptive Server Anywhere 版本
- CALL 语句
- CREATE PROCEDURE、 CREATE FUNCTION 和 CREATE TRIGGER 语句的 Adaptive Server Anywhere 版本
- 由分号分隔的 SQL 语句

注意

不能在一个过程、触发器或批处理语句中混用两种方言。这意味着：

- 可以在一个批处理语句、过程或触发器中同时包含仅限 Transact-SQL 的语句以及属于两种方言的语句。
- 可以在一个批处理语句、过程或触发器中同时包含 Adaptive Server Enterprise 不支持的语句以及两种服务器都支持的语句。
- 不能在一个批处理语句、过程或触发器中同时包含仅限 Transact-SQL 的语句和仅限 Adaptive Server Anywhere 的语句。

Adaptive Server 体系结构

Adaptive Server Enterprise 和 Adaptive Server Anywhere 是互补的产品，其体系结构分别满足不同的需求。当工作组或部门服务器需要的管理比较少时， Adaptive Server Anywhere 非常适用，此时作为个人数据库使用。 Adaptive Server Enterprise 则适合用作大型数据库的企业级服务器。

本节将介绍 Adaptive Server Enterprise 和 Adaptive Server Anywhere 在体系结构上的差异。还将介绍 Adaptive Server Anywhere 为达到数据库管理的兼容性而包含的类似于 Adaptive Server Enterprise 的工具。

服务器和数据库

Adaptive Server Enterprise 中服务器和数据库之间的关系不同于 Adaptive Server Anywhere 中服务器和数据库之间的关系。

在 Adaptive Server Enterprise 中，每个数据库都存在于服务器中，而每个服务器也都可以包含多个数据库。用户可以具有服务器的登录权限并且可以连接到服务器。连接之后，用户可以使用该服务器上他们对其具有权限的每个数据库。全系统范围的系统表保存在 master 数据库中，该数据库包含服务器上所有数据库共有的信息。

Adaptive Server Anywhere 上没有 master 数据库

在 Adaptive Server Anywhere 中，不存在与 Adaptive Server Enterprise 的 master 数据库相对应的级别。相反，每个数据库都是一个独立的实体，它们包含所有系统表。用户具有数据库 （而不是服务器）的连接权限。当用户建立连接后，他们将连接到单独的数据库。没有在 master 数据库级别维护的全系统范围的系统表集。每个 Adaptive Server Anywhere 数据库服务器都可以动态地装载和卸载多个数据库，而用户可以在每个数据库上建立独立的连接。

Adaptive Server Anywhere 为支持 Transact-SQL 和 Open Server 提供了一些工具，以便按照与 Adaptive Server Enterprise 类似的方式执行某些任务。例如， Adaptive Server Anywhere 提供了 Adaptive Server Enterprise **sp_addlogin** 系统过程的实现，它执行几乎是等效的操作：将用户添加到数据库中。

有关 Open Server 支持的信息，请参见 『ASA 数据库管理指南』> 「将 Adaptive Server Anywhere 作为 Open Server」。

文件操作语句

Adaptive Server Anywhere 不支持 Transact-SQL 的下列两种语句：DUMP DATABASE 和 LOAD DATABASE。 Adaptive Server Anywhere 具有其自己的 CREATE DATABASE 和 DROP DATABASE 语句，但这些语句的语法不同。

设备管理

Adaptive Server Anywhere 和 Adaptive Server Enterprise 使用不同的模式来管理设备和磁盘空间，这反映了这两种产品的不同用途。 Adaptive Server Enterprise 使用大量的 Transact-SQL 语句制定出一套综合的资源管理方案，而 Adaptive Server Anywhere 则自动管理它自己的资源，其数据库是常规的操作系统文件。

Adaptive Server Anywhere 不支持 Transact-SQL 的 DISK 语句，如 DISK INIT、 DISK MIRROR、 DISK REFIT、 DISK REINIT、 DISK REMIRROR 和 DISK UNMIRROR。

有关磁盘管理的信息，请参见 『ASA 数据库管理指南』> 「使用数据库文件」。

缺省值和规则

Adaptive Server Anywhere 不支持 Transact-SQL 的 CREATE DEFAULT 语句或 CREATE RULE 语句。 CREATE DOMAIN 语句用于将缺省值和规则 （称作 CHECK 条件）合并为域的定义，因此提供了与 Transact-SQL 的 CREATE DEFAULT 和 CREATE RULE 语句相似的功能。

在 Adaptive Server Enterprise 中， CREATE DEFAULT 语句创建指定的**缺省值**。通过使用 **sp_bindefault** 系统过程，将此缺省值绑定到特定的列，此缺省值可以用作列的缺省；或者绑定到数据类型，此缺少值可以用作域中所有列的缺省值。

CREATE RULE 语句创建指定的**规则**，该规则可用于定义列的域 （通过将此规则绑定到特定的列）或者用作域中所有列的规则 （通过将此规则绑定到特定的数据类型）。规则是使用 **sp_bindrule** 系统过程来绑定到数据类型或列的。

在 Adaptive Server Anywhere 中，域可以有与之关联的缺省值和 CHECK 条件 （它们将应用于在该数据类型上定义的所有列）。可以使用 CREATE DOMAIN 语句来创建域。

您可以使用 CREATE TABLE 语句或 ALTER TABLE 语句为单独的列定义缺省值和规则 （或 CHECK 条件）。

有关这些语句的 Adaptive Server Anywhere 语法的说明，请参见 『ASA SQL 参考』> 「SQL 语句」。

系统表

除了自己的系统表之外， Adaptive Server Anywhere 还提供了一组系统视图，这些视图模拟 Adaptive Server Enterprise 系统表中的相关部分。『ASA SQL 参考』> 「Transact-SQL 兼容性视图」中提供了这些视图的列表和各个说明，还介绍了两种产品的系统目录。本节将简短地概述它们的差异。

Adaptive Server Anywhere 系统表完全位于每个数据库中，而 Adaptive Server Enterprise 系统表则部分位于每个数据库中，部分位于 master 数据库中。 Adaptive Server Anywhere 体系结构中不包含 master 数据库。

在 Adaptive Server Enterprise 中，数据库所有者 （用户 ID **dbo**）拥有系统表。而在 Adaptive Server Anywhere 中，系统所有者 （用户 ID **SYS**）拥有系统表。 **dbo** 用户 ID 拥有 Adaptive Server Anywhere 提供的与 Adaptive Server Enterprise 兼容的系统视图。

管理角色

与 Adaptive Server Anywhere 相比， Adaptive Server Enterprise 具有一组更为详细的管理角色。虽然可以向 Adaptive Server Enterprise 上的多个登录帐户授予任意角色，而且一个帐户可以拥有多个角色，但 Adaptive Server Enterprise 中有一组不应重复的角色。

Adaptive Server Enterprise 的角色

在 Adaptive Server Enterprise 中，不应重复的角色包括：

- **系统管理员** 负责与特定应用程序无关的常规管理任务；可以访问任何数据库对象。

- **系统安全员** 负责 Adaptive Server Enterprise 中与安全相关的任务，但对数据库对象不具有特殊的权限。
- **数据库所有者** 对自己拥有的数据库中的对象具有完全权限，可以将用户添加到数据库中并向其它用户授予在数据库中创建对象和执行命令的权限。
- **数据定义语句** 可以向用户授予执行特定的数据定义语句（CREATE TABLE 或 CREATE VIEW）的权限，使用户能够创建数据库对象。
- **对象所有者** 每个数据库对象都有一个所有者，该所有者可以向其它用户授予访问该对象的权限。对象的所有者自动具有该对象的各种权限。

在 Adaptive Server Anywhere 中，以下全数据库范围的权限具有管理角色：

- 与 Adaptive Server Enterprise 数据库所有者类似，数据库管理员（DBA 权限）对该数据库中的所有对象 （SYS 所拥有的对象除外）具有完全权限，并且可以向其它用户授予在该数据库中创建对象和执行命令的权限。缺省的数据库管理员是用户 ID **DBA**。
- RESOURCE 权限允许用户在数据库中创建任意类型的对象。它取代 Adaptive Server Enterprise 中授予单独的 CREATE 语句权限的方案。
- Adaptive Server Anywhere 的对象所有者与 Adaptive Server Enterprise 的对象所有者的权限相同。对象的所有者自动具有该对象上的任何权限 （包括授予权限的权利）。

为了顺利地访问 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中保存的数据，应该在数据库中创建具有适当权限 （在 Adaptive Server Anywhere 中为 RESOURCE 权限，或者在 Adaptive Server Enterprise 中为执行单个 CREATE 语句的权限）的用户 ID，并利用该用户 ID 创建对象。如果在每个环境中都使用相同的用户 ID，对象名和限定符可以在这两种数据库中完全相同，从而可确保能进行兼容的访问。

用户和组

Adaptive Server Enterprise 和 Adaptive Server Anywhere 在用户和组模式上存在着一些差异。

在 Adaptive Server Enterprise 中，用户将连接到某个服务器，而且每个用户都需要该服务器的登录 ID 和口令以及该服务器上他们要访问的每个数据库的用户 ID。数据库的每个用户只能是一个组的成员。

在 Adaptive Server Anywhere 中，用户可以直接连接到数据库，并且不需要单独的登录 ID 即可连接到数据库服务器。实际上，每个用户都会接收数据库上的用户 ID 和口令，这样他们便可以使用该数据库。用户可以是许多组的成员，并且组的层次性嵌套是允许存在的。

这两种服务器都支持用户组，所以您可以一次向多个用户授予权限。但是，这两种服务器中的组在某些细节上存在差异。例如， Adaptive Server Enterprise 只允许每个用户成为一个组的成员，而 Adaptive Server Anywhere 没有这种限制。有关具体信息，请比较这两种产品中有关用户和组的文档。

Adaptive Server Enterprise 和 Adaptive Server Anywhere 都具有公共组，该组用于定义缺省权限。每个用户都会自动成为公共组的成员。

Adaptive Server Anywhere 支持使用以下 Adaptive Server Enterprise 系统过程来管理用户和组。

有关每个过程的参数，请参见 『ASA SQL 参考』> 「Adaptive Server Enterprise 系统和分类过程」。

系统过程	说明
sp_addlogin	在 Adaptive Server Enterprise 中，将用户添加到服务器。在 Adaptive Server Anywhere，将用户添加到数据库。
sp_adduser	在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中，将用户添加到数据库。在 Adaptive Server Enterprise 中，它与 **sp_addlogin** 是截然不同的两个任务，但在 Adaptive Server Anywhere 中，它们是相同的。
sp_addgroup	将组添加到数据库中。

系统过程	说明
sp_changegroup	将用户添加到组中，或者将用户从一个组移到另一个组。
sp_droplogin	在 Adaptive Server Enterprise 中，从服务器中删除用户。在 Adaptive Server Anywhere 中，从数据库中删除用户。
sp_dropuser	从数据库中删除用户。
sp_dropgroup	从数据库中删除组。

在 Adaptive Server Enterprise 中，登录 ID 是全服务器范围的。在 Adaptive Server Anywhere 中，用户却属于单个的数据库。

数据库对象权限

Adaptive Server Enterprise 和 Adaptive Server Anywhere 在授予特定数据库对象的权限时所使用的 GRANT 和 REVOKE 语句非常相似。它们都允许在数据库表和视图上拥有 SELECT、 INSERT、 DELETE、 UPDATE 和 REFERENCES 权限以及在数据库表的选定列拥有 UPDATE 权限。它们都允许在存储过程上授予 EXECUTE 权限。

例如，以下语句在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中都有效：

```
GRANT INSERT, DELETE
 ON TITLES
 TO MARY, SALES
```

该语句向用户 **MARY** 和 **SALES** 组授予对 **TITLES** 表使用 INSERT 语句和 DELETE 语句的权限。

虽然 Adaptive Server Anywhere 不允许在 GRANT EXECUTE 语句上使用 WITH GRANT OPTION，但 Adaptive Server Anywhere 和 Adaptive Server Enterprise 都支持 WITH GRANT OPTION 子句，该子句允许权限的接收者向其它用户授予这种权限。

全数据库范围的权限

对于全数据库范围的用户权限， Adaptive Server Enterprise 和 Adaptive Server Anywhere 使用不同的模式。有关这些模式，将在 「用户和组」第 511 页中进行讨论。 Adaptive Server Anywhere 使用 DBA 权限来向用户授予对数据库的完全权限。 Adaptive Server Enterprise 中的系统管理员享有服务器上所有数据库的这一权限。但是， Adaptive Server

Anywhere 数据库上的 DBA 权限不同于 Adaptive Server Enterprise 数据库所有者的权限，后者必须使用 Adaptive Server Enterprise 的 **SETUSER** 语句获取对其它用户拥有的对象的权限。

Adaptive Server Anywhere 使用 RESOURCE 权限来授予用户在数据库中创建对象的权限。相应的 Adaptive Server Enterprise 权限为 GRANT ALL，它由数据库所有者来使用。

为实现 Transact-SQL 兼容性配置数据库

通过在创建数据库时 （或者如果正在处理现有的数据库，则是在重建数据库时）选择正确的选项，可以排除 Adaptive Server Anywhere 和 Adaptive Server Enterprise 之间的一些行为差异。其它差异则可以通过用 SET TEMPORARY OPTION 语句 （在 Adaptive Server Anywhere 中）或 SET 语句 （在 Adaptive Server Enterprise 中）设置连接级别的选项来进行控制。

创建与 Transact-SQL 兼容的数据库

本节将介绍您在创建或重建数据库时必须做出的选择。

快速入门

下面是创建与 Transact-SQL 兼容的数据库时所需要执行的步骤。本节的其余部分将介绍您需要设置的选项。

❖ **创建与 Transact-SQL 兼容的数据库 (Sybase Central)：**

1. 启动 Sybase Central。
2. 选择 [工具] → [Adaptive Server Anywhere 9] → [创建数据库]。
3. 请按照向导中的说明进行操作。
4. 当您看到一个名为 [模拟 Adaptive Server Enterprise] 的按钮时，单击该按钮，然后单击 [下一步]。
5. 请按照向导中的其余说明进行操作。

❖ **创建与 Transact-SQL 兼容的数据库 （命令行）：**

- 在命令提示符下输入以下命令：

  ```
  dbinit -b -c -k db-name.db
  ```

  有关这些选项的详细信息，请参见 『ASA 数据库管理指南』> 「初始化实用程序选项」。

❖ **创建与 Transact-SQL 兼容的数据库 (SQL)：**

1 连接到任意 Adaptive Server Anywhere 数据库。

2 在 Interactive SQL 中输入以下语句：

```
CREATE DATABASE 'db-name.db'
 ASE COMPATIBLE
```

在该语句中， ASE COMPATIBLE 表示与 Adaptive Server Enterprise 兼容。

使数据库区分大小写

缺省情况下， Adaptive Server Enterprise 数据库中的字符串比较是区分大小写的，而 Adaptive Server Anywhere 中的字符串比较却不区分大小写。

在使用 Adaptive Server Anywhere 建立与 Adaptive Server Enterprise 兼容的数据库时，请选中区分大小写选项。

- 如果您在使用 Sybase Central，则此选项位于 [创建数据库] 向导中。
- 如果您在使用 *dbinit* 命令行实用程序，请指定 `-c` 选项。

在比较中忽略尾随空白

在使用 Adaptive Server Anywhere 建立与 Adaptive Server Enterprise 兼容的数据库时，请选择在比较中忽略尾随空白的选项。

- 如果您在使用 Sybase Central，则此选项位于 [创建数据库] 向导中。
- 如果您在使用 *dbinit* 命令行实用程序，请指定 `-b` 命令行选项。

当您选择此选项后， Adaptive Server Enterprise 和 Adaptive Server Anywhere 会认为以下两个字符串是相同的：

```
'ignore the trailing blanks   '
 'ignore the trailing blanks'
```

如果您不选择此选项， Adaptive Server Anywhere 就会认为上面的两个字符串不相同。

选择此选项的副作用是当客户端应用程序读取字符串时，字符串将被填上空白。

删除历史系统视图

较早的 Adaptive Server Anywhere 版本使用的两种系统视图的名称与为实现兼容性而提供的 Adaptive Server Enterprise 系统视图的名称相冲突。这两种视图包括 SYSCOLUMNS 和 SYSINDEXES。如果您在使用 Open Client 或 JDBC 接口，请在创建数据库时排除这两种视图。您可以使用 *dbinit* -k 命令行选项达到这一目的。

如果您在创建数据库时没有使用这一选项，以下两个语句将返回不同的结果：

```
SELECT * FROM SYSCOLUMNS ;

 SELECT * FROM dbo.syscolumns ;
```

❖ **从现有的数据库中删除 Adaptive Server Anywhere 系统视图：**

1 以具有 DBA 权限的用户身份连接到数据库。

2 执行以下命令：

```
DROP VIEW SYS.SYSCOLUMNS ;
 DROP VIEW SYS.SYSINDEXES
```

注意
一定不要删除 dbo.syscolumns 或 dbo.sysindexes 系统视图。

为实现 Transact-SQL 兼容性设置选项

可以使用 SET OPTION 语句来设置 Adaptive Server Anywhere 数据库选项。有几项数据库选项设置与 Transact-SQL 的行为相关。

设置 allow_nulls_by_default 选项

缺省情况下，如果您没有显式指定新列允许空值， Adaptive Server Enterprise 就不允许新列中出现空值。 Adaptive Server Anywhere 在缺省情况下允许在新列中出现空值，这符合 SQL/92 ISO 标准。

若要使 Adaptive Server Enterprise 以符合 SQL/92 的方式运行，请使用 **sp_dboption** 系统过程将 **allow_nulls_by_default** 选项设置为真。

若要使 Adaptive Server Anywhere 以与 Transact-SQL 兼容的方式运行，请将 **allow_nulls_by_default** 选项设置为 OFF。使用如下 SET OPTION 语句就可以达到这一目的：

```
SET OPTION PUBLIC.allow_nulls_by_default = 'OFF'
```

设置 quoted_identifier 选项

缺省情况下， Adaptive Server Enterprise 处理标识符和字符串的方式与 Adaptive Server Anywhere 有所不同，后者符合 SQL/92 ISO 标准。

quoted_identifier 选项在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中均可用。为了以兼容的方式处理标识符和字符串，应确保在这两种数据库中将该选项设置为相同的值。

要想获得 SQL/92 行为，请在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中均将 **quoted_identifier** 选项设置为 ON。

要想获得 Transact-SQL 行为，请在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中均将 **quoted_identifier** 选项设置为 OFF。如果您选择此设置，则不能在双引号中使用与关键字相同的标识符。

有关 **quoted_identifier** 选项的详细信息，请参见 『ASA 数据库管理指南』> 「QUOTED_IDENTIFIER 选项 [compatibility]」。

将 automatic_timestamp 选项设置为 ON

Transact-SQL 使用特殊的属性定义 **timestamp** 列。当 **automatic_timestamp** 选项设置为 ON 时， Adaptive Server Anywhere 处理 **timestamp** 列的方式与 Adaptive Server Enterprise 行为类似。

在 Adaptive Server Anywhere 中将 **automatic_timestamp** 选项设置为 ON（缺省设置为 OFF）后，未定义显式缺省值的数据类型为 TIMESTAMP 的任何新列都将接收 **timestamp** 的缺省值。

有关 **timestamp** 列的信息，请参见 「特殊的 Transact-SQL timestamp 列和数据类型」第 519 页。

设置 string_rtruncation 选项

Adaptive Server Enterprise 和 Adaptive Server Anywhere 都支持 string_rtruncation 选项，该选项会影响在 INSERT 或 UPDATE 字符串被截断时所报告的错误消息。请确保在每种数据库中将该选项设置为相同的值。

有关 STRING_RTRUNCATION 选项的详细信息，请参见 『ASA 数据库管理指南』> 「STRING_RTRUNCATION 选项 [compatibility]」。

有关涉及 Transact-SQL 兼容性的数据库选项的详细信息，请参见『ASA 数据库管理指南』> 「兼容性选项」。

区分大小写

数据库中的是否区分大小写的特性指的是：

- **数据** 数据的区分大小写会反映在索引等内容中。
- **标识符** 标识符包括表名、列名等。
- **口令** 口令的区分大小写的处理方式与其它标识符有所不同。

数据的区分大小写特性

在创建数据库时，应确定 Adaptive Server Anywhere 数据在比较时是否区分大小写。缺省情况下，虽然数据始终保持输入时的大小写，但 Adaptive Server Anywhere 数据库在比较时不区分大小写。

Adaptive Server Enterprise 是否区分大小写取决于在 Adaptive Server Enterprise 系统上安装的排序顺序。通过重新配置 Adaptive Server Enterprise 排序顺序，可以更改单字节字符集的区分大小写特性。

标识符的区分大小写特性

Adaptive Server Anywhere 不支持区分大小写的标识符。在 Adaptive Server Enterprise 中，标识符的区分大小写特性遵循数据的区分大小写特性。用户 ID 和其它标识符一样，也区分大小写。数据库的缺省用户 ID 是大写的 **DBA**。

在 Adaptive Server Enterprise 中，域名是区分大小写的。在 Adaptive Server Anywhere 中，除 Java 数据类型之外，域名不区分大小写。

口令的区分大小写特性

在 Adaptive Server Anywhere 中，口令区分大小写的特性与数据的区分大小写的特性是分别指定的。缺省情况下，如果您没有指定口令的区分大小写特性，则口令会遵循数据库的区分大小写特性。不过，您可以在不区分大小写的数据库中指定口令是区分大小写的，而且您也可以在区分大小写的数据库中拥有不区分大小写的口令。口令中使用的扩展字符区分大小写，而与口令的区分大小写设置无关。数据库的缺省口令是大写的 **SQL**。

有关设置口令区分大小写特性的详细信息，请参见 『ASA SQL 参考』> 「CREATE DATABASE 语句」中的 PASSWORD CASE 子句。

在 Adaptive Server Enterprise 中，用户 ID 和口令的区分大小写特性遵循服务器的区分大小写特性。

确保兼容的对象名

每个数据库对象在特定的**命名空间**中都有唯一的名称。在此命名空间之外，允许出现重复的名称。某些数据库对象在 Adaptive Server Enterprise 和 Adaptive Server Anywhere 中占用不同的命名空间。

在 Adaptive Server Anywhere 中，索引和触发器由创建它们时它们所在的表的所有者拥有。对于给定的所有者，索引和触发器名称必须是唯一的。例如，虽然用户 user1 拥有的表 t1 和用户 user2 拥有的 t2 可能具有同名的索引，但一个用户拥有的两个表却不能有同名的索引。

对于索引名， Adaptive Server Enterprise 的命名空间限制比 Adaptive Server Anywhere 的少。索引名在给定表上必须是唯一的，但任何两个表都可以具有同名的索引。为了达到 SQL 兼容性，对于给定表所有者，应遵循 Adaptive Server Anywhere 的唯一索引名限制。

对于触发器名称， Adaptive Server Enterprise 的命名空间限制比 Adaptive Server Anywhere 的更严格。触发器名称在数据库中必须是唯一的。为了达到 SQL 兼容性，您应该遵守 Adaptive Server Enterprise 的限制并确保触发器名称在数据库中是唯一的。

特殊的 Transact-SQL timestamp 列和数据类型

Adaptive Server Anywhere 支持 Transact-SQL 的特殊 timestamp 列。timestamp 列与 tsequal 系统函数一起检查行是否已更新。

timestamp 的两种含义

Adaptive Server Anywhere 具有 TIMESTAMP 数据类型，它保存精确的日期和时间信息。它不同于特殊的 Transact-SQL TIMESTAMP 列和数据类型。

在 Adaptive Server Anywhere 中创建 Transact-SQL timestamp 列

若要创建 Transact-SQL timestamp 列，请创建一个 (Adaptive Server Anywhere) 数据类型为 TIMESTAMP 且缺省设置为 timestamp 的列。虽然名称 timestamp 比较常用，但该列可以具有任何名称。

例如，下面的 CREATE TABLE 语句包含一个 Transact-SQL timestamp 列：

```
CREATE TABLE table_name (
     column_1 INTEGER ,
     column_2 TIMESTAMP DEFAULT TIMESTAMP
 )
```

下面的 ALTER TABLE 语句将一个 Transact-SQL timestamp 列添加到 sales_order 表中：

```
ALTER TABLE sales_order
 ADD timestamp TIMESTAMP DEFAULT TIMESTAMP
```

在 Adaptive Server Enterprise 中，名称为 timestamp 且未指定数据类型的列会自动接收 TIMESTAMP 数据类型。在 Adaptive Server Anywhere 中，您必须显式指定数据类型。

如果将 AUTOMATIC_TIMESTAMP 数据库选项设置为 ON，则无需设置缺省值：任何用 TIMESTAMP 数据类型创建且不含显式缺省值的新列都将接收 timestamp 的缺省值。以下语句将 AUTOMATIC_TIMESTAMP 设置为 ON：

```
SET OPTION PUBLIC.AUTOMATIC_TIMESTAMP='ON'
```

timestamp 列的数据类型

Adaptive Server Enterprise 将 timestamp 列当作允许 NULL 的 VARBINARY(8) 域，而 Adaptive Server Anywhere 将 timestamp 列当作 TIMESTAMP 数据类型，它包括日期和时间 （秒的小数部分占六个小数位）。

在为随后的更新从表中读取时，读取时间戳值的变量应该与列说明相对应。

向现有的表添加时间戳

如果向现有表添加特殊的 timestamp 列，所有现有行在 timestamp 列中的值都将为 NULL。若要为现有行输入时间戳值 （当前时间戳），请更新表中的所有行，而不更改数据。例如，以下语句会更新 sales_order 表中的所有行，而不更改任何行中的值：

```
UPDATE sales_order
 SET region = region
```

在 Interactive SQL 中，您可能需要设置 TIMESTAMP_FORMAT 选项以查看行值的差异。以下语句将 TIMESTAMP_FORMAT 设置为显示秒钟小数部分的所有六位数字。

```
SET OPTION TIMESTAMP_FORMAT='YYYY-MM-DD
HH:NN:ss.SSSSSS'
```

如果所有六位都未显示，某些 timestamp 列值可能看起来相同，但实际上却不同。

将 tsequal 用于更新

利用 tsequal 系统函数，您可以确定 timestamp 列是否已得到更新。

例如，某一应用程序可能用 SELECT 将一个 timestamp 列添加到变量中。当某一选定行的 UPDATE 被提交时，它可能会使用 tsequal 函数检查该行是否已被修改。 tsequal 函数将比较表中的时间戳值和在 SELECT 中获取的时间戳值。时间戳相同意味着没有变化。如果时间戳不同，则意味着该行在执行 SELECT 后出现了变化。

使用 tsequal 函数的典型 UPDATE 语句看起来如下：

```
UPDATE publishers
 SET city = 'Springfield'
 WHERE pub_id = '0736'
 AND TSEQUAL(timestamp, '1995/10/25 11:08:34.173226')
```

tsequal 函数的第一个参数是特殊 timestamp 列的名称，第二个参数是在 SELECT 语句中检索的时间戳。在嵌入式 SQL 中，第二个参数可能是一个主机变量，包含该列上进行最近的 FETCH 操作的 TIMESTAMP 值。

特殊 IDENTITY 列

若要创建 IDENTITY 列，请使用以下 CREATE TABLE 语法：

```
CREATE TABLE table-name (
     ...
     column-name numeric(n,0) IDENTITY NOT NULL,
     ...
 )
```

其中 *n* 足够大，可以容纳可能在表中插入的最大行数的值。

IDENTITY 列存储顺序编号，如自动生成的发票编号和雇员编号。IDENTITY 列的值唯一地标识表中的每一行。

在 Adaptive Server Enterprise 中，数据库中的每个表只能有一个 IDENTITY 列。其数据类型必须是不带小数位的数值，而且 IDENTITY 列不应允许空值。

在 Adaptive Server Anywhere 中， IDENTITY 列是列缺省设置。您可以使用 INSERT 语句将不在序列中的值显式地插入列中。如果 identity_insert 选项不是 *on*， Adaptive Server Enterprise 将不允许使用 INSERT 将值插入 IDENTITY 列。在 Adaptive Server Anywhere 中，您需要自己设置 NOT NULL 属性并确保只有一个列是 IDENTITY 列。Adaptive Server Anywhere 允许任何数字数据类型成为 IDENTITY 列。

在 Adaptive Server Anywhere 中， identity 列和列的 AUTOINCREMENT 缺省设置是相同的。

使用 @@identity 检索 IDENTITY 列值

当您第一次将行插入表中时， IDENTITY 列将被分配值 1。对于每次后继的插入，该列的值都会递增 1。最近插入 identity 列的值可在 @@identity 全局变量中提供。

每当一个语句尝试在表中插入行时， @@identity 的值都会出现变化。

- 如果该语句影响不含 IDENTITY 列的表，@@identity 将被设置为 0。
- 如果该语句插入多行， @@identity 将反映被插入到 IDENTITY 列中的最后一个值。

这种更改是永久性的。如果该语句失败或者包含该语句的事务被回退，@@identity 将不会恢复它原先的值。

有关 @@identity 的行为的详细信息，请参见 『ASA SQL 参考』> 「@@identity 全局变量」。

编写兼容的 SQL 语句

本节提供了有关编写将在多个数据库管理系统上使用的 SQL 的一般原则，并讨论 Adaptive Server Enterprise 和 Adaptive Server Anywhere 在 SQL 语句级别的兼容性问题。

编写可移植的 SQL 的一般原则

在编写将在多个数据库管理系统上使用的 SQL 语句时，应该使您的 SQL 语句尽可能地清晰。即使多个服务器支持给定的 SQL 语句，每个系统上的缺省行为也可能不相同。有关编写兼容的 SQL 语句的一般原则包括：

- 详细地写出所有可用选项，而不是使用缺省行为。
- 使用括号来明确语句中的执行顺序，而不是对运算符采用相同的缺省优先级。
- 使用在变量名前加 @ 符号这一 Transact-SQL 约定，以实现 Adaptive Server Enterprise 可移植性。
- 在过程、触发器和批处理语句中声明变量和游标要紧跟在 BEGIN 语句之后。这是 Adaptive Server Anywhere 的要求，尽管 Adaptive Server Enterprise 允许在过程、触发器或批处理语句中的任意位置进行声明。
- 避免将 Adaptive Server Enterprise 或 Adaptive Server Anywhere 中的保留字用作数据库中的标识符。
- 采用较大的命名空间。例如，要确保每个索引都有唯一的名称。

创建兼容表

Adaptive Server Anywhere 支持那些允许在数据类型定义中封装约束和缺省定义的域。它还支持在 CREATE TABLE 语句中使用显式缺省值和 CHECK 条件。但是，它不支持命名的约束或命名的缺省值。

NULL

Adaptive Server Anywhere 和 Adaptive Server Enterprise 在处理 NULL 的某些方面存在差异。在 Adaptive Server Enterprise 中，有时处理 NULL 就好像它是个值。

例如， Adaptive Server Enterprise 中的唯一索引允许最多一个空值的行，不允许重复的行。在 Adaptive Server Anywhere 中则可以包含空行。

缺省情况下， Adaptive Server Enterprise 中的列缺省为 NOT NULL，而 Adaptive Server Anywhere 中的缺省设置为 NULL。您可以使用 **allow_nulls_by_default** 选项控制该设置。通过显式指定 NULL 或 NOT NULL，可以使您的数据定义语句成为可移植的语句。

有关此选项的信息，请参见 「为实现 Transact-SQL 兼容性设置选项」第 516 页。

临时表

您可以通过在 CREATE TABLE 语句前放置井号 (#) 来创建临时表。这些临时表是 Adaptive Server Anywhere 声明的临时表，并且只能在当前连接中才可用。有关 Adaptive Server Anywhere 中声明的临时表的信息，请参见 『ASA SQL 参考』> 「DECLARE LOCAL TEMPORARY TABLE 语句」。

在 Adaptive Server Enterprise 中表的实际放置方式不同于在 Adaptive Server Anywhere 中表的实际放置方式。 Adaptive Server Anywhere 支持 **ON** *segment-name* 子句，但 *segment-name* 会引用 Adaptive Server Anywhere dbspace。

有关 CREATE TABLE 语句的信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

编写兼容的查询

若要编写在 Adaptive Server Anywhere 数据库和 Adaptive Server Enterprise 数据库上都能运行的查询，应遵循下面两条标准：

- 查询中的数据类型、表达式和搜索条件必须兼容。
- SELECT 语句本身的语法必须兼容。

本节将介绍兼容的 SELECT 语句语法，并说明兼容的数据类型、表达式和搜索条件。以下示例假定 QUOTED_IDENTIFIER 的设置为 OFF，即 Adaptive Server Enterprise 的缺省设置，但不是 Adaptive Server Anywhere 的缺省设置。

Adaptive Server Anywhere 支持 Transact-SQL SELECT 语句的以下子集。

语法

SELECT [**ALL** | **DISTINCT**] *select-list*
…[**INTO** *#temporary-table-name*]
…[**FROM** *table-spec* [**HOLDLOCK** | **NOHOLDLOCK**],
…　*table-spec* [**HOLDLOCK** | **NOHOLDLOCK**], …]
…[**WHERE** *search-condition*]
…[**GROUP BY** *column-name*, …]
…[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }
[**ASC** | **DESC**], …]

参数

select-list:
*table-name.**
| ***
| *expression*
| *alias-name = expression*
| *expression as identifier*
| *expression as T_string*

table-spec:
[*owner* .]*table-name*
…[[**AS**] *correlation-name*]
…[**(INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**] **)**]

alias-name:*identifier* :
| '*string*' | "*string*"

有关 SELECT 语句的完整描述，请参见 『ASA SQL 参考』> 「SELECT 语句」。

Adaptive Server Anywhere 不支持 Transact-SQL SELECT 语句语法的以下关键字和子句：

- SHARED 关键字
- COMPUTE 子句
- FOR BROWSE 子句

- GROUP BY ALL 子句

注意

- 只有对于其表名以 # 开头的声明临时表，才支持 INTO table_name 子句，该子句将基于 SELECT 语句结果集创建新表。声明的临时表的存在时间仅为一个连接的存在时间。
- Adaptive Server Anywhere 不支持对 GROUP BY 子句的 Transact-SQL 扩展 （它允许引用未用于创建组的列和表达式）。在 Adaptive Server Enterprise 中，此扩展会生成摘要报告。
- 将会对 FOR READ ONLY 子句和 FOR UPDATE 子句进行语法分析，但不会产生任何影响。
- 将会对表的说明的性能参数部分进行语法分析，但不会产生任何影响。
- Adaptive Server Anywhere 支持 HOLDLOCK 关键字。该关键字会保留指定表或视图上的共享锁直至事务完成，从而使共享锁更为严格 （而不是无论事务是否已完成，都在不需要所需的数据页时立即释放共享锁）。对于已为其指定了 HOLDLOCK 的表，查询将在隔离级别 3 执行。
- HOLDLOCK 选项仅应用于指定该选项的表或视图，并且仅应用于该选项所在的语句所定义的事务期间。如果将隔离级别设置为 3，则会将 HOLDLOCK 应用于事务中的每一个 SELECT 语句。您不能在一个查询中同时指定 HOLDLOCK 和 NOHOLDLOCK 选项。
- Adaptive Server Anywhere 识别 NOHOLDLOCK 关键字，但该关键字不会产生任何影响。
- Transact-SQL 使用 SELECT 语句向局部变量赋值：

  ```
  SELECT @localvar = 42
  ```

  Adaptive Server Anywhere 中的相应语句为 SET 语句：

  ```
  SET localvar = 42
  ```

  可使用 SET 语句和 @ 符后跟名称的 Transact-SQL 约定设置变量名：

  ```
  SET @localvar = 42
  ```

- Adaptive Server Enterprise 不支持 SELECT 语句语法的以下子句：
  - INTO 主机变量列表
  - INTO 变量列表。
  - 带括号的查询。
- Adaptive Server Enterprise 在 WHERE 子句中使用连接运算符，而不使用 FROM 子句和 ON 条件来进行连接。

连接的兼容性

在 Transact-SQL 中， WHERE 子句中的连接的语法如下：

start of select, *update*, *insert*, *delete*, *or subquery*
FROM { *table-list* | *view-list* } **WHERE** [**NOT**]
[*table-name*.| *view name*.]*column-name*
join-operator
[*table-name*.| *view-name*.]*column_name*
[{ **AND** | **OR** } [**NOT**]
[*table-name*.| *view-name*.]*column_name*
join-operator
[*table-name*.| *view-name*.]*column-name*]…
end of select, *update*, *insert*, *delete*, *or subquery*

WHERE 子句中的 *join-operator* 可以是任意比较运算符，或者也可以是以下两个**外部连接运算符**之一：

- ***=** 左外部连接运算符
- **=*** 右外部连接运算符。

Adaptive Server Anywhere 支持 Transact-SQL 外部连接运算符作为本地 SQL/92 语法的替代语法。不能在一个查询中混用这些方言。这一规则也适用于查询使用的视图 - 视图上的外部连接查询必须遵循视图定义查询所使用的方言。

Adaptive Server Anywhere 还为除外部连接之外的连接提供了 SQL/92 语法，在这种语法中，连接放置在 FROM 子句中，而不是放置在 WHERE 子句中。

有关 Adaptive Server Anywhere 中的连接和 ANSI/ISO SQL 标准中的连接的信息，请参见 「连接：从多个表检索数据」第 281 页和 『ASA SQL 参考』> 「FROM 子句」。

有关连接的 Transact-SQL 兼容性的详细信息，请参见 「Transact-SQL 外连接 （*= 或 =*）」第 301 页。

Transact-SQL 过程语言概述

存储过程语言是用于存储过程、触发器和批处理语句的 SQL 语言的一部分。

除了基于 SQL/92 的 Watcom-SQL 方言之外， Adaptive Server Anywhere 还支持 Transact-SQL 存储过程语言中相当大的一部分。

Transact-SQL 存储过程概述

基于 ISO/ANSI 标准草案的 Adaptive Server Anywhere 存储过程语言在许多方面不同于 Transact-SQL 方言。它们的许多概念和功能都相似，但语法却不同。 Adaptive Server Anywhere 对 Transact-SQL 的支持是通过提供方言之间的自动转换来利用类似的概念的。但是，过程必须仅使用两种方言中的一种来编写，而不能混用这两种方言。

Adaptive Server Anywhere 对 Transact-SQL 存储过程的支持

Adaptive Server Anywhere 对 Transact-SQL 存储过程的支持包括多个方面，例如：

- 传递参数
- 返回结果集
- 返回状态信息
- 提供参数的缺省值
- 控制语句
- 错误处理
- 用户定义函数

Transact-SQL 触发器概述

触发器兼容性需要触发器功能和语法的兼容性。本节提供有关 Transact-SQL 和 Adaptive Server Anywhere 触发器的功能兼容性的概述。

Adaptive Server Enterprise 在触发语句完成后执行触发器：它们是**语句级之后**触发器。 Adaptive Server Anywhere 既支持**行级**触发器（在修改每一行之前或之后执行），又支持语句级触发器 （在整个语句后执行）。

行级触发器不属于 Transact-SQL 兼容性功能，在 「使用过程、触发器和批处理」第 699 页中会讨论行级触发器。

有关不支持或不同的 Transact-SQL 触发器的说明

Adaptive Server Anywhere 中不支持的或不同的 Transact-SQL 触发器功能包括：

- **触发器引发其它触发器** 假定触发器执行将引发其它触发器的操作 （在用户直接执行该操作的情况下）。对于这种情况，Adaptive Server Anywhere 和 Adaptive Server Enterprise 的响应方式略有不同。缺省情况下， Adaptive Server Enterprise 中的触发器最多能够引发可配置的嵌套级别 （缺省值为 16）的其它触发器。您可以使用 Adaptive Server Enterprise 的**嵌套触发器**选项来控制嵌套级别。在 Adaptive Server Anywhere 中，触发器可以毫无限制地引发其它触发器 （除非内存不足）。

- **触发器自行引发** 假定触发器执行将引发相同触发器的操作 （在用户直接执行该操作的情况下）。对于这种情况， Adaptive Server Anywhere 和 Adaptive Server Enterprise 的响应方式略有不同。在 Adaptive Server Anywhere 中，非 Transact-SQL 触发器以递归方式自行引发，而 Transact-SQL 方言触发器则不以递归方式自行引发。

  缺省情况下， Adaptive Server Enterprise 中的触发器不以递归方式调用其自身，但您可以打开 **self_recursion** 选项，以允许触发器以递归方式调用其自身。

- **触发器中的 ROLLBACK 语句** Adaptive Server Enterprise 允许在触发器中使用 ROLLBACK TRANSACTION 语句来回退触发器所属的整个事务。由于触发操作及其触发器一起形成一个原子语句，Adaptive Server Anywhere 不允许在触发器中使用 ROLLBACK （或 ROLLBACK TRANSACTION）语句。

  Adaptive Server Anywhere 提供了与 Adaptive Server Enterprise 兼容的 ROLLBACK TRIGGER 语句，以撤消触发器中的操作。请参见『ASA SQL 参考』> 「ROLLBACK TRIGGER 语句」。

Transact-SQL 批处理语句概述

在 Transact-SQL 中，**批处理语句**是一起提交并作为一个组依次执行的 SQL 语句组。批处理语句可以存储在命令文件中。 Adaptive Server Anywhere 中的 Interactive SQL 实用程序和 Adaptive Server Enterprise 中的 *isql* 实用程序为交互地执行批处理语句提供了类似的功能。

过程中使用的控制语句也可以在批处理语句中使用。 Adaptive Server Anywhere 支持在批处理语句中使用控制语句，并支持以类似于 Transact-SQL 方式使用以 GO 语句终止的非分隔语句组来表示批处理语句的结尾。

对于命令文件中存储的批处理语句， Adaptive Server Anywhere 支持在命令文件中使用参数。 Adaptive Server Enterprise 不支持参数。

有关参数的信息，请参见 『ASA SQL 参考』> 「PARAMETERS 语句 [Interactive SQL]」。

存储过程的自动转换

除了支持 Transact-SQL 替代语法之外， Adaptive Server Anywhere 还提供了用于在 Watcom-SQL 方言和 Transact-SQL 方言之间进行转换的辅助工具。返回有关 SQL 语句的信息并可实现 SQL 语句自动转换的函数包括：

- **SQLDialect(statement)** 返回 **Watcom-SQL** 或 **Transact-SQL**。
- **WatcomSQL(statement)** 返回语句的 Watcom-SQL 语法。
- **TransactSQL(statement)** 返回语句的 Transact-SQL 语法。

它们是函数，因此可以使用 Interactive SQL 中的 SELECT 语句来访问它们。例如：

```
select SqlDialect('select * from employee')
```

返回值 Watcom-SQL。

使用 Sybase Central 转换存储过程

Sybase Central 具有创建、查看和变更过程和触发器的功能。

❖ 使用 Sybase Central 转换存储过程：

1. 使用 Sybase Central 作为要更改的过程的所有者 （或者作为 DBA 用户）连接到数据库。
2. 打开 [过程和函数] 文件夹。
3. 右击要转换的过程，然后根据要使用的方言从弹出式菜单中选择其中一个 [转换为] 命令。

   该过程即会以选定方言出现在右窗格中。如果选定方言不是用来存储该过程的方言，服务器就会将其转换为该方言。任何未经转换的行都会显示为注释。
4. 根据需要重写任何未经转换的行。

5　完成后，选择 [文件] → [保存过程]，将转换后的版本保存到数据库中。您也可以将文本导出至文件，以便在 Sybase Central 之外进行编辑。

从 Transact-SQL 过程中返回结果集

Adaptive Server Anywhere 使用 RESULT 子句来指定所返回的结果集。在 Transact-SQL 过程中，第一个查询的列名或别名将返回到发出调用的环境中。

Transact-SQL 过程的示例

以下 Transact-SQL 过程说明了 Transact-SQL 存储过程如何返回结果集：

```
CREATE PROCEDURE showdept (@deptname varchar(30))
 AS
     SELECT employee.emp_lname, employee.emp_fname
     FROM department, employee
     WHERE department.dept_name = @deptname
     AND department.dept_id = employee.dept_id
```

Watcom-SQL 过程的示例

下面是相应的 Adaptive Server Anywhere 过程：

```
CREATE PROCEDURE showdept(in deptname varchar(30))
 RESULT ( lastname char(20), firstname char(20))
 BEGIN
     SELECT employee.emp_lname, employee.emp_fname
     FROM department, employee
     WHERE department.dept_name = deptname
     AND department.dept_id = employee.dept_id
 END
```

有关过程和结果的详细信息，请参见 「从过程返回结果」第 732 页

Transact-SQL 过程中的变量

Adaptive Server Anywhere 使用 SET 语句来向过程中的变量赋值。在 Transact-SQL 中，将使用含空表列表的 SELECT 语句或 SET 语句来赋值。下面的简单过程说明了 Transact-SQL 语法的工作原理：

```
CREATE PROCEDURE multiply
                    @mult1 int,
                    @mult2 int,
                    @result int output
 AS
 SELECT @result = @mult1 * @mult2
```

可以按如下方式来调用该过程：

```
CREATE VARIABLE @product int
 go
 EXECUTE multiply 5, 6, @product OUTPUT
 go
```

当该过程执行后，变量 **@product** 的值为 30。

有关使用 SELECT 语句给变量赋值的详细信息，请参见 「编写兼容的查询」第 524 页。有关使用 SET 语句给变量赋值的详细信息，请参见『ASA SQL 参考』> 「SET 语句」。

Transact-SQL 过程中的错误处理

Watcom-SQL 方言中的缺省过程错误处理方式不同于 Transact-SQL 方言中的缺省过程错误处理方式。缺省情况下， Watcom-SQL 方言过程在它们遇到错误时退出，并且将 SQLSTATE 值和 SQLCODE 值返回到发出调用的环境中。

显式的错误处理可以使用 EXCEPTION 语句来置入到 Watcom-SQL 存储过程中，或者，您可以使用 ON EXCEPTION RESUME 语句指示该过程在遇到错误时从下一个语句继续执行。

当 Transact-SQL 方言过程遇到错误时，将从下一个语句继续执行。全局变量 **@@error** 保存最近执行过的语句的错误状态。您可以在语句之后检查这一变量，以强制从过程中返回。例如，以下语句在出错时将导致退出。

```
IF @@error != 0 RETURN
```

当该过程执行完之后，返回值将表明该过程是成功还是失败。此返回状态是一个整数，并且可以按如下方式对其进行访问：

```
DECLARE @status INT
 EXECUTE @status = proc_sample
 IF @status = 0
     PRINT 'procedure succeeded'
 ELSE
     PRINT 'procedure failed'
```

下表介绍了内置的过程返回值以及这些值的含义：

值	含义
0	过程执行时没有发生错误
-1	缺少对象
-2	数据类型错误
-3	进程被选作死锁牺牲品
-4	权限错误
-5	语法错误

值	含义
-6	杂类用户错误
-7	资源错误，如空间不足
-8	非致命的内部问题
-9	达到系统限制
-10	致命的内部不一致性
-11	致命的内部不一致性
-12	表或索引损坏
-13	数据库损坏
-14	硬件错误

RETURN 语句可用于返回其它整数，由用户定义它们的含义。

在过程中使用 RAISERROR 语句

RAISERROR 语句是用于生成用户定义错误的 Transact-SQL 语句。其功能与 SIGNAL 语句的功能类似。

有关 RAISERROR 语句的说明，请参见 『ASA SQL 参考』> 「RAISERROR 语句 [T-SQL]」。

RAISERROR 语句本身并不会导致从过程中退出，但它可以同 RETURN 语句或 **@@error** 全局变量的测试组合在一起控制用户定义错误之后的执行过程。

如果 ON_TSQL_ERROR 数据库选项被设置为 CONTINUE，RAISERROR 语句将不再发出有关执行结束错误的信号。相反，该过程会完成并存储 RAISERROR 状态代码和消息，然后返回最近的 RAISERROR。如果导致 RAISERROR 的过程是从其它过程中调用的，RAISERROR 将在最外部的调用过程终止后返回。

当该过程终止后，您将丢失中间的 RAISERROR 状态和代码。如果错误在返回时随 RAISERROR 一起发生，则将返回错误信息并丢失 RAISERROR 信息。应用程序可以通过在不同的执行点检查 **@@error** 全局变量，查询中间 RAISERROR 状态。

Watcom-SQL 方言中与 Transact-SQL 类似的错误处理

您可以通过向 CREATE PROCEDURE 语句提供 ON EXCEPTION RESUME 子句来使 Watcom-SQL 方言过程以类似于 Transact-SQL 的方式处理错误：

```
CREATE PROCEDURE sample_proc()
 ON EXCEPTION RESUME
 BEGIN
     ...
 END
```

ON EXCEPTION RESUME 子句的存在将防止执行显式的异常处理代码，因此应该避免同时使用这两个子句。

第 15 章

与其它版本 SQL 语言的区别

关于本章

Adaptive Server Anywhere 完全符合基于 SQL-92 的第 127 号美国联邦信息处理标准发布 (FIPS PUB)。

Adaptive Server Anywhere 完全符合 ISO/ANSI SQL-92 标准，而且，除一些微小差异外，也符合 SQL-99 核心规范。

Adaptive Server Anywhere 各项功能的参考文档中提供了有关兼容性的完整、详细的信息。

本章介绍 Adaptive Server Anywhere 所特有的一些功能，其他 SQL 实现通常不具有这些功能。

Adaptive Server Anywhere SQL 功能

Adaptive Server Anywhere 所支持的以下 SQL 功能在其他许多 SQL 实现中都没有。

类型转换

支持完全类型转换。任何数据类型都可以与任何其他数据类型进行比较，或一起用于任何表达式。

日期

Adaptive Server Anywhere 具有日期、时间和时间戳类型，包括年月日、小时、分钟、秒和更小的时间单位。插入或更新日期字段时，或与日期字段进行比较时，支持使用自由格式日期。

此外，还可以对日期进行以下运算：

- **日期 + 整数** 在日期中加上指定的天数。
- **日期 - 整数** 从日期中减去指定的天数。
- **日期 - 日期** 计算两个日期之间的天数。
- **日期 + 时间** 用日期和时间创建一个时间戳。

另外，还提供了许多用于操纵日期和时间的函数。有关这些函数的说明，请参见 『ASA SQL 参考』> 「SQL 函数」。

完整性

Adaptive Server Anywhere 支持实体完整性和参照完整性。这是通过对 CREATE TABLE 和 ALTER TABLE 命令进行以下扩展实现的。

```
PRIMARY KEY ( column-name, ... )
 [NOT NULL] FOREIGN KEY [role-name]
                [(column-name, ...)]
            REFERENCES table-name [(column-name, ...)]
                [ CHECK ON COMMIT ]
```

PRIMARY KEY 子句声明关系的主键。之后， Adaptive Server Anywhere 将确保主键的唯一性，并确保主键中的任何一列都不包含空值。

FOREIGN KEY 子句定义此表与另一个表之间的关系。这个关系由此表中的一列 （或多列）表示，该列必须包含另一个表的主键中的值。之后，系统将确保这些列的完整性 - 只要修改了这些列，或在此表中插入

了一行，都将对这些列进行检查，以确保有一列或多列为空值，或这些值与另一个表的主键中某行的对应列相匹配。有关详细信息，请参见『ASA SQL 参考』> 「CREATE TABLE 语句」。

连接

Adaptive Server Anywhere 允许表之间的**自动连接**。除其他实现中支持的 NATURAL 和 OUTER 连接操作符之外， Adaptive Server Anywhere 还允许在表之间基于外键关系建立 KEY 连接。这降低了执行连接时 WHERE 子句的复杂程度。

更新

Adaptive Server Anywhere 允许 UPDATE 命令引用多个表。在多个表上定义的视图也可以更新。许多 SQL 实现不允许更新连接的表。

变更表

对 ALTER TABLE 命令进行了扩展。除了用于实体完整性和参照完整性的更改外，还支持以下变更：

```
ADD column data-type
 MODIFY column data-type
 DELETE column
 RENAME new-table-name
 RENAME old-column TO new-column
```

使用 MODIFY 可以更改字符列的最大长度，也可以从一种数据类型转换为另一种数据类型。有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TABLE 语句」。

支持子查询 WHERE 表达式

Adaptive Server Anywhere 允许在任何可以放置表达式的位置放置子查询。许多 SQL 实现只允许将子查询放置在比较运算符的右边。以下命令在 Adaptive Server Anywhere 中有效，但在大多数其他 SQL 实现中无效。

```
SELECT    emp_lname,
             birth_date,
             (    SELECT dept_name
                  FROM department
                  WHERE emp_id = employee.emp_ID
                  AND dept_id = 200 )
 FROM employee
```

附加函数

Adaptive Server Anywhere 支持几个 ANSI SQL 定义中没有的函数。有关可用函数的完整列表，请参见 『ASA SQL 参考』> 「SQL 函数」。

游标

使用 Embedded SQL 时，可以使用 FETCH 语句任意移动游标位置。可以相对于当前游标位置或相对于游标最前面或最后面给定数量的记录，前移或后移游标。

第 4 部分　数据库中的 XML

这一部分介绍了如何在数据库中使用 XML。

第 16 章

在数据库中使用 XML

关于本章

本章摘要介绍了 Adaptive Server Anywhere 中的 XML 支持，包括导入、导出、存储和查询 XML 数据。

什么是 XML？

[可扩展标记语言](XML) 以文本格式呈现结构化数据。 XML 是专门为解决大规模电子发布的难题而设计的。

XML 是一种简单的标记语言，就像 HTML，但它也很灵活，就像 SGML。 XML 是分层的，而且它的主要用途是为人和计算机软件都能进行创作和阅读而描述数据的结构。

XML 不提供描述各种不同形式数据的一组静态元素，而是让您来定义元素。这样，使用 XML 就可以描述很多种结构化数据。 XML 文档可以选择使用文档类型定义 (DTD) 或 XML 模式来定义 XML 文件中使用的结构、元素和属性。

有关 XML 的更多详细信息，请参见 http://www.w3.org/XML/。

XML 和 Adaptive Server Anywhere

XML 配合 Adaptive Server Anywhere 使用有多种方法：

- 在数据库中存储 XML 文档
- 将关系数据导出为 XML
- 将 XML 导入数据库
- 以 XML 的形式查询关系数据

在关系数据库中存储 XML 文档

Adaptive Server Anywhere 支持两种可用于在您的数据库中存储 XML 文档的数据类型：XML 数据类型和 LONG VARCHAR 数据类型。这两种数据类型都会在数据库中将 XML 文档存储为字符串。

使用 XML 数据类型的好处

您可以在 XML 数据类型与任何其它能够与字符串相互转换的数据类型之间进行转换。注意，在将字符串转换为 XML 时，不检查字符串格式是否正确。

在您从关系数据中生成元素时，任何在 XML 中无效的字符都会被转义，除非该数据是 XML 类型的。例如，假定您想用以下内容生成一个 <product> 元素：

```
<hat>bowler</hat>
```

以便该元素内容包含小于号和大于号。如果您像下面这样编写一个指定该元素内容是 XML 类型的查询：

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML ) AS
product )
```

那么，大于号和小于号就不会被加上引号，并且您会得到以下结果：

```
<product><hat>bowler</hat></product>
```

但是，如果该查询没有指定该元素内容是 XML 类型，例如：

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product )
```

那么，小于号和大于号就会像下面这样被实体引用替换：

```
<product>&lt;hat&gt;bowler&lt;/hat&gt;</product>
```

请注意，属性始终都会带有引号，无论它是什么数据类型。

有关如何将元素内容转义的详细信息，请参见 「无效的列名」第 561 页。

有关 XML 数据类型的详细信息，请参见 『ASA SQL 参考』> 「XML 数据类型 [Character]」。

将关系数据导出为 XML

Adaptive Server Anywhere 提供了两种以 XML 的形式导出关系数据的方法：Interactive SQL OUTPUT 语句和 ADO.NET DataSet 对象。

使用 FOR XML 子句和 SQL/XML 函数，您可以利用您数据库中的关系数据生成 XML 格式的结果集。然后，您可以使用 UNLOAD 语句或 xp_write_file 系统过程将生成的 XML 导出到一个文件。

从 Interactive SQL 中将关系数据导出为 XML

Interactive SQL OUTPUT 语句支持一种将查询结果输出到生成的 XML 文件的 XML 格式。

这个生成的 XML 文件以 UTF-8 格式编码，而且包含一个嵌入的 DTD。在这个 XML 文件中，二进制值编码为字符数据 (CDATA) 块，而二进制数据会显示为两位数的十六进制字符串。

有关使用 OUTPUT 语句导出 XML 的详细信息，请参见 『ASA SQL 参考』> 「OUTPUT 语句 [Interactive SQL]」。

INPUT 语句不接受 XML 作为文件格式。不过，可使用 OPENXML 函数或 ADO.NET DataSet 对象导入 XML。

有关导入 XML 的详细信息，请参见 「导入 XML 文档作为关系数据」第 550 页。

使用 DataSet 对象将关系数据导出为 XML

使用 ADO.NET DataSet 对象，您可以在 XML 文档中保存 DataSet 的内容。您在填充了 DataSet （例如，用对您的数据进行查询的结果）后，就可以在 XML 文件中保存 DataSet 中的模式或既保存其模式又保存其数据。 WriteXml 方法在 XML 文件中既保存模式又保存数据，而 WriteXmlSchema 方法在 XML 文件中只保存模式。您可以使用 Adaptive Server Anywhere ADO.NET 数据提供程序来填充 DataSet 对象。

有关使用 DataSet 将关系数据导出为 XML 的信息，请参见 『ASA 编程指南』> 「使用 AsaCommand 对象插入、更新和删除行」。

导入 XML 文档作为关系数据

Adaptive Server Anywhere 支持两种将 XML 导入数据库的不同方法：

- 使用 OPENXML 函数从 XML 文档中生成结果集
- 使用 ADO.NET DataSet 对象将 XML 文档中的数据和 / 或模式读入 DataSet

使用 OPENXML 导入 XML

OPENXML 在查询的 FROM 子句中使用，目的是从 XML 文档中生成结果集。 OPENXML 使用 XPath 查询语言的子集从 XML 文档中选择节点。

使用 XPath 表达式

在您使用 OPENXML 时，会对 XML 文档进行分析，而结果会采用树形式的模型。这个树由节点组成。 XPath 表达式用于选择树中的节点。下面的列表描述了一些常用的 XPath 表达式：

- / 指明 XML 文档的根节点
- **.（一个句点）**指明 XML 文档的当前节点
- // 指明当前节点的所有子代，包括当前节点
- **..** 指明当前节点的父节点
- **./@** ***属性名*** 指明当前节点的具有名称*属性名*的属性
- **./** ***子级名*** 指明当前节点的那些是具有名称*子级名*的元素的子级

请看下面的 XML 文档：

```
<inventory>
  <product id="301" size="Medium">Tee Shirt
   <quantity>54</quantity>
  </product>
  <product id="302" size="One size fits all">Tee Shirt
   <quantity>75</quantity>
  </product>
```

```
  <product id="400" size="One size fits all">Baseball
Cap
   <quantity>112</quantity>
  </product>
 </inventory>
```

<inventory> 元素不是根节点。您可以使用以下 XPath 表达式来引用它：

```
/inventory
```

假定当前节点是一个 <quantity> 元素。您可以使用以下 XPath 表达式来引用此节点：

```
.
```

要查找所有是 <inventory> 元素的子级的 <product> 元素，请使用以下 XPath 表达式：

```
/inventory/product
```

如果当前节点是 <product> 元素，而您需要引用 size 属性，请使用以下 XPath 表达式：

```
./@size
```

有关 OPENXML 支持的 XPath 语法的完整列表，请参见 『ASA SQL 参考』> 「OPENXML 函数 [String]」。

有关 XPath 查询语言的信息，请参见 http://www.w3.org/TR/xpath。

使用 OPENXML 生成结果集

OPENXML 的第一个 *xpath-query* 参数每匹配一次，就会在结果集中生成一行。 WITH 子句指定结果集的模式以及如何为结果集中的每一列找到值。例如，假定有以下查询：

```
SELECT * FROM OPENXML( '<inventory>
                          <product>Tee Shirt
                           <quantity>54</quantity>
                           <color>Orange</color>
                          </product>
                          <product>Baseball Cap
```

```
                                <quantity>112</quantity>
                                 <color>Black</color>
                                </product>
                               </inventory>',
                              '/inventory/product' )
    WITH ( name CHAR (25) './text()',
           quantity CHAR(3) 'quantity',
           color CHAR(20) 'color')
```

第一个 *xpath-query* 参数是 **/inventory/product**，并且 XML 中有两个 <product> 元素，所以，此查询会生成两行。

WITH 子句指定了有三列：name、 quantity 和 color。这些列的值是从 <product>、 <quantity> 和 <color> 元素中获得的。上面的查询会生成以下结果：

name	quantity	color
Tee Shirt	54	Orange
Baseball Cap	112	Black

有关详细信息，请参见 『ASA SQL 参考』> 「OPENXML 函数 [String]」。

使用 OPENXML 来生成边缘表

OPENXML 可以用来生成边缘表，在这种表中，XML 文档中的每一个元素都会有对应的一行。您可能会需要生成边缘表，以便能够使用 SQL 来查询结果集中的数据。

下面的 SQL 语句创建一个变量 x，该变量包含一个 XML 文档。该查询生成的 XML 有一个名为 <root> 的根元素，这个根元素是使用 XMLELEMENT 函数生成的，而且会使用指定了 ELEMENTS 修饰符的 FOR XML AUTO 为 employee、 sales_order 和 customer 等表中的每一列生成元素。

有关 XMLELEMENT 函数的信息，请参见 『ASA SQL 参考』> 「XMLELEMENT 函数 [String]」。

有关 FOR XML AUTO 的信息，请参见 「使用 FOR XML AUTO」第 565 页。

```
CREATE VARIABLE x XML;
 SET x=(SELECT XMLELEMENT( NAME root,
          (SELECT * FROM employee
          KEY JOIN sales_order
          KEY JOIN customer
          FOR XML AUTO, ELEMENTS)));
```

生成的 XML 看上去会像下面这样 （已对结果进行格式化处理以便更容易阅读 - 查询返回的结果是一个连续字符串）：

```
<root>
  <employee>
   <emp_id>299</emp_id>
   <manager_id>902</manager_id>
   <emp_fname>Rollin</emp_fname>
   <emp_lname>Overbey</emp_lname>
   <dept_id>200</dept_id>
   <street>191 Companion Ct.</street>

  <city>Kanata</city>
   <state>CA</state>
   <zip_code>94608</zip_code>
   <phone>5105557255</phone>
   <status>A</status>
   <ss_number>025487133</ss_number>
   <salary>39300.000</salary>
   <start_date>1987-02-19</start_date>

  <birth_date>1964-03-15</birth_date>
   <bene_health_ins>Y</bene_health_ins>
   <bene_life_ins>Y</bene_life_ins>
   <bene_day_care>N</bene_day_care>
   <sex>M</sex>
   <sales_order>
    <id>2001</id>
    <cust_id>101</cust_id>

   <order_date>2000-03-16</order_date>
    <fin_code_id>r1</fin_code_id>
    <region>Eastern</region>
    <sales_rep>299</sales_rep>
    <customer>
     <id>101</id>
     <fname>Michael</fname>
     <lname>Devlin</lname>
     <address>114 Pioneer Avenue</address>
```

```
    <city>Kingston</city>
     <state>NJ</state>
     <zip>07070</zip>
     <phone>2015558966</phone>
     <company_name>The Power Group</company_name>
    </customer>
   </sales_order>
</employee>
...
```

下面的查询使用 descendant-or-self (***//****) XPath 表达式来匹配上面 XML 文档中的每一个元素，而且，对于每一个元素都使用 id 元属性来为节点获取 id，而且将父 (***..***/) XPath 表达式与 id 元属性一起使用来获取父节点。 localname 元属性用于获取每个元素的名称。

```
SELECT * FROM OPENXML( x, '//*' )
  WITH (id INT '@mp:id',
        parent INT '../@mp:id',
        name CHAR(20) '@mp:localname',
        text LONG VARCHAR 'text()' )
 ORDER BY id
```

此查询生成的结果集会显示每个节点的 id、父节点的 id，以及 XML 文档中各元素的名称和内容。

id	parent	name	text
5	(NULL)	root	(NULL)
23	15	emp_id	299
47	15	manager_id	902
74	15	emp_fname	Rollin
...	...	...	...

查询列中的 XML

如果您有一个表，它的某一列包含 XML，您可以使用 OPENXML 一次查询出该列中所有的 XML 值。使用侧向派生表可以实现这一点。

下面的语句创建一个两列 （manager_id 和 reports）的表。 reports 列包含从 employee 表中生成的 XML 数据。

```
CREATE TABLE t (manager_id INT, reports XML);
 INSERT INTO t
 SELECT manager_id, XMLELEMENT( NAME reports,
                     XMLAGG(
                      XMLELEMENT( NAME e, emp_id)))
 FROM employee
 GROUP BY manager_id;
```

请执行下面的查询来查看 t 表中的数据：

```
SELECT * FROM t;
```

此查询会产生以下结果：

manager_id	reports
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
1576	<reports> <e>184</e> <e>207</e> <e>318</e> <e>409</e> ... </reports>
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>

manager_id	reports
703	<reports> <e>191</e> <e>750</e> <e>868</e> <e>921</e> ... </reports>
...	...

下面的查询使用侧向派生表来生成一个两列的结果集，一列列出各位经理的 id，另一列列出接受该经理领导的各位雇员的 id：

```
SELECT manager_id, eid
 FROM t, LATERAL( OPENXML( t.reports, '//e' )
 WITH (eid INT '.') ) dt
```

此查询会生成以下结果：

manager_id	eid
1293	148
1293	390
1293	586
1293	757
...	...

有关侧向派生表的详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。

使用 DataSet 对象导入 XML

使用 ADO.NET DataSet 对象，您可以将 XML 文档中的数据和 / 或模式读入 DataSet。

- ReadXml 方法用既包含模式又包含数据的 XML 文档中的内容填充 DataSet。
- ReadXmlSchema 方法只从 XML 文档中读取模式。在使用 XML 文档中的数据填充了 DataSet 后，您可以用 DataSet 中的更改来更新您数据库中的表。

使用 Adaptive Server Anywhere ADO.NET 数据提供程序，也可以操作 DataSet 对象。

有关利用 Adaptive Server Anywhere .NET 数据提供程序使用 DataSet 从 XML 文档中读取数据和 / 或模式的信息，请参见 『ASA 编程指南』> 「使用 AsaDataAdapter 对象获取数据」。

以 XML 格式获取查询结果

Adaptive Server Anywhere 支持两种以 XML 格式从您的关系数据中获取查询结果的不同方法：

- **FOR XML 子句** FOR XML 子句可以用在 SELECT 语句中，以生成 XML 文档。

  有关使用 FOR XML 子句的信息，请参见 「使用 FOR XML 子句以 XML 格式检索查询结果」第 559 页和 『ASA SQL 参考』> 「SELECT 语句」。

- **SQL/XML** Adaptive Server Anywhere 支持基于从关系数据中生成 XML 文档的 SQL/XML 标准草案的函数。

  有关在查询中使用这些函数中的一个或多个的信息，请参见「使用 SQL/XML 以 XML 的形式获取查询结果」第 582 页。

Adaptive Server Anywhere 支持的 FOR XML 子句和 SQL/XML 函数为您在从关系数据中生成 XML 时提供了两种选择。在大多数情况下，这两种方法生成的 XML 是相同的，您使用哪一种都可以。

例如，这个查询使用 FOR XML AUTO 生成 XML，

```
SELECT id, name
 FROM product
 WHERE color='black'
 FOR XML AUTO
```

而这个查询使用 XMLELEMENT 函数生成 XML：

```
SELECT XMLELEMENT(NAME product,
           XMLATTRIBUTES(id, name))
 FROM product
 WHERE color='black'
```

两种查询都会生成下面的 XML （已对结果进行格式化处理以便更容易阅读）：

```
<product id="302" name="Tee Shirt"/>
 <product id="400" name="Baseball Cap"/>
 <product id="501" name="Visor"/>
 <product id="700" name="Shorts"/>
```

提示

如果您要生成深度嵌套的文档， FOR XML EXPLICIT 查询可能会比 SQL/XML 查询效率高，因为 EXPLICIT 模式查询通常会使用 UNION 来生成嵌套，而 SQL/XML 使用子查询来生成所需嵌套。

使用 FOR XML 子句以 XML 格式检索查询结果

使用 Adaptive Server Anywhere 时，通过在您的 SELECT 语句中使用 FOR XML 子句，您可以对数据库执行 SQL 查询并以 XML 文档的形式返回结果。 XML 文档是 XML 类型的。

有关 XML 数据类型的信息，请参见 『ASA SQL 参考』> 「XML 数据类型 [Character]」。

FOR XML 子句可以用在任何 SELECT 语句中，包括子查询、使用 GROUP BY 子句或集合函数的查询，以及视图定义。

有关如何使用 FOR XML 子句的示例，请参见 「FOR XML 示例」第 562 页。

Adaptive Server Anywhere 不会为 FOR XML 子句生成的 XML 文档生成模式。

在 FOR XML 子句中，您会指定控制所生成的 XML 的格式的三种 XML 模式中的一种：

- **RAW** 表示每一个将查询匹配为 XML <row> 元素的行以及每一个将查询匹配为属性的列。

  有关详细信息，请参见 「使用 FOR XML RAW」第 563 页。

- **AUTO** 将查询结果返回为嵌套的 XML 元素。*选择列表*中引用的每一个表都会表示为 XML 中的一个元素。元素的嵌套顺序基于*选择列表*中表的顺序而定。

  有关详细信息，请参见 「使用 FOR XML AUTO」第 565 页。

- **EXPLICIT** 允许您编写包含有关正确嵌套的信息的查询，这样您就能够控制最终 XML 的形式。

有关详细信息，请参见 「使用 FOR XML EXPLICIT」第 569 页。

下面的部分介绍了 FOR XML 子句的所有三种模式：二进制数据、NULL 值和无效的 XML 名称等方面的行为，另外还提供了 FOR XML 子句的使用方法示例。

FOR XML 和二进制数据

您在 SELECT 语句中使用 FOR XML 子句时，无论使用哪种模式，BINARY、 LONG BINARY、 IMAGE 或 VARBINARY 列都会输出为自动以 base64 编码格式表示的属性或元素。

如果您要使用 OPENXML 从 XML 生成结果集， OPENXML 会假定 BINARY、 LONG BINARY、 IMAGE 和 VARBINARY 等类型以 base64 格式编码并自动对它们进行解码。

有关 OPENXML 的详细信息，请参见 『ASA SQL 参考』> 「OPENXML 函数 [String]」。

FOR XML 和 NULL 值

缺省情况下，结果中会忽略包含 NULL 值的元素和属性。这一点是由 FOR_XML_NULL_TREATMENT 选项控制的。

请看包含以下数据的表：

id	fname	lname	company_name
100	Robert	Michaels	NULL
101	Michael	Devlin	The Power Group

如果您在 FOR_XML_NULL_TREATMENT 选项设置为 OMIT （缺省值）的情况下执行下面的查询

```
SELECT id, fname, lname, company_name
 FROM customer
 FOR XML RAW
```

Robert Michaels 的结果中就会忽略 company_name 属性：

```
<row id="100" fname="Robert" lname="Michaels"/>
 <row id="101" fname="Michael" lname="Devlin"
  company_name="The Power Group"/>'
```

如果 FOR_XML_NULL_TREATMENT 选项设置为 EMPTY，那么结果中就会包括一个空属性：

```
<row id="100" fname="Robert" lname="Michaels"
  company_name=""/>
 <row id="101" fname="Michael" lname="Devlin"
  company_name="The Power Group"/>'
```

在这种情况下，会为 Robert Michaels 生成一个空 company_name 属性。

有关 FOR_XML_NULL_TREATMENT 选项的信息，请参见 『ASA 数据库管理指南』> 「FOR_XML_NULL_TREATMENT 选项 [database]」。

无效的列名

Adaptive Server Anywhere 在对不是合法 XML 名称的名称 （例如，包括空格的列名）进行编码时会使用以下规则：

- 不是有效 XML 名称字符的 Unicode 字符会编码为 _x*HHHH*_，其中 *HHHH* 为该字符的四位十六进制代码点。 Unicode 字符的代码点值如果无法在 16 位中指定，该字符就会被编码。

  例如，下面的查询包含带有空格的列名：

  ```
  SELECT emp_id AS "employee id"
   FROM employee
   FOR XML RAW
  ```

  并且会返回以下结果：

  ```
  <row employee_x0020_id="102"/>
   <row employee_x0020_id="105"/>
   <row employee_x0020_id="129"/>
   <row employee_x0020_id="148"/>
   ...
  ```

- 非 Unicode 字符会编码为 _x*HHHHHHHH*_，其中 *HHHHHHHH* 为该字符的八位十六进制代码点。
- 下划线 (_) 后如果跟有字符 x，就会被转义。
- 冒号 (:) 不会被转义，这样，使用 FOR XML 查询就可以生成命名空间声明和限定的元素名和属性名。

有关 FOR XML 子句语法的详细信息，请参见 『ASA SQL 参考』> 「SELECT 语句」。

提示

在 Interactive SQL 中执行包含 FOR XML 子句的查询时，您可能会需要通过设置 TRUNCATION_LENGTH 选项来增加列的长度。

有关设置截断长度的信息，请参见 『ASA 数据库管理指南』> 「TRUNCATION_LENGTH 选项 [Interactive SQL]」和 『SQL Anywhere Studio 帮助』> 「[选项] 对话框：[结果] 选项卡」。

FOR XML 示例

下面的示例说明了如何在 SELECT 语句中使用 FOR XML 子句。

- 下面的示例说明了如何在子查询中使用 FOR XML 子句：

```
SELECT XMLELEMENT(
    NAME root,
       (SELECT * FROM employee
        FOR XML RAW))
```

- 下面的示例说明了如何在使用了 GROUP BY 子句和集合函数的查询中使用 FOR XML 子句：

```
SELECT name, AVG(unit_price) AS Price
 FROM product
 GROUP BY name
 FOR XML RAW
```

- 下面的示例说明了如何在视图定义中使用 FOR XML 子句：

```
CREATE VIEW emp_dept
 AS SELECT emp_lname, emp_fname, dept_name
 FROM employee JOIN department
 ON employee.dept_id = department.dept_id
 FOR XML AUTO
```

使用 FOR XML RAW

当您在查询中指定 FOR XML RAW 时，每一行都会表示为一个 <row> 元素，而每一列都是 <row> 元素的属性。

语法　**FOR XML RAW**[, **ELEMENTS**]

参数　**ELEMENTS**　让 FOR XML RAW 为结果中的每一列生成 XML 元素，而不是属性。如果有 NULL 值，生成的 XML 文档中就会忽略该元素。下面的查询会生成 <emp_id> 元素和 <dept_name> 元素：

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 FOR XML RAW, ELEMENTS
```

并给出以下结果：

```
<row>
    <emp_id>102</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 <row>
    <emp_id>105</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>

<row>
    <emp_id>160</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 <row>
    <emp_id>243</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 ...
```

用法

在您执行包含 FOR XML RAW 的查询时， BINARY、 LONG BINARY、IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。

缺省情况下，结果中会忽略 NULL 值。这一点是由 FOR_XML_NULL_TREATMENT 选项控制的。

有关如何在包含 FOR XML 子句的查询中返回 NULL 值的信息，请参见「FOR XML 和 NULL 值」第 560 页。

FOR XML RAW 不会返回结构完好的 XML 文档，因为该文档没有一个单一的根节点。如果需要 <root> 元素，一种插入一个该元素的方法是：使用 XMLELEMENT 函数。例如，

```
SELECT XMLELEMENT( NAME root,
                   (SELECT emp_id AS id, emp_fname AS
name
                    FROM employee FOR XML RAW))
```

有关 XMLELEMENT 函数的详细信息，请参见 『ASA SQL 参考』> 「XMLELEMENT 函数 [String]」。

通过指定别名，可以更改 XML 文档中使用的属性名或元素名。下面的查询会将 id 属性重命名为 product_id：

```
SELECT id AS product_id
 FROM product
 WHERE color='black'
 FOR XML RAW
```

并给出以下结果：

```
<row product_id="302"/>
 <row product_id="400"/>
 <row product_id="501"/>
 <row product_id="700"/>
```

结果的顺序取决于优化程序选择的计划，除非您以其它方式请求。如果您希望结果按某种特定顺序显示，必须在查询中加入 ORDER BY 子句，例如，

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 FOR XML RAW
```

示例　假定像下面这样，您想检索有关某名雇员隶属于哪个部门的信息：

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 FOR XML RAW
```

就会返回下面的 XML 文档：

```
<row emp_id="102" dept_name="R &amp; D"/>
 <row emp_id="105" dept_name="R &amp; D"/>
 <row emp_id="160" dept_name="R &amp; D"/>
 <row emp_id="243" dept_name="R &amp; D"/>
 ...
```

使用 FOR XML AUTO

AUTO 模式会在 XML 文档中生成嵌套的元素。选择列表中引用的每一个表在生成的 XML 中都会表示为一个元素。嵌套的顺序基于选择列表中引用表的顺序而定。在您指定 AUTO 模式时，会为选择列表中的每个表创建一个元素，而该表中的每一列都是一个单独的属性。

语法　**FOR XML AUTO**[, **ELEMENTS**]

参数　**ELEMENTS**　让 FOR XML AUTO 为结果中的每一列生成 XML 元素，而不是属性。例如，

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 FOR XML AUTO, ELEMENTS
```

在这种情况下，结果集中的每一列都会返回为一个单独的元素，而不是 <employee> 元素的属性。如果有 NULL 值，生成的 XML 文档中就会忽略该元素。

```
<employee>
    <emp_id>102</emp_id>
    <department>
       <dept_name>R &amp; D</dept_name>
    </department>
 </employee>

<employee>
    <emp_id>105</emp_id>
    <department>
       <dept_name>R &amp; D</dept_name>
    </department>
 </employee>


<employee>
    <emp_id>160</emp_id>
    <department>
    <dept_name>R &amp; D</dept_name>
    </department>
 </employee>
 ...
```

用法

在您使用 FOR XML AUTO 执行查询时， BINARY、 LONG BINARY、IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。缺省情况下，结果中会忽略 NULL 值。通过将 FOR_XML_NULL_TREATMENT 选项设置为 EMPTY，您可以将 NULL 值返回为空属性。

有关设置 FOR_XML_NULL_TREATMENT 选项的信息，请参见『ASA 数据库管理指南』> 「FOR_XML_NULL_TREATMENT 选项 [database]」。

除非另有要求，否则数据库服务器返回表中行的顺序没有任何意义。如果您希望结果按某种特定顺序显示，或者，对于有多个子级的父元素，您必须在查询中加入 ORDER BY 子句，以便让所有子级都相邻。如果您不指定 ORDER BY 子句，结果的嵌套就会取决于优化程序选择的计划，而且您可能得不到需要的嵌套。

FOR XML AUTO 不会返回结构完好的 XML 文档，因为该文档没有一个单一的根节点。如果需要 <root> 元素，一种插入一个该元素的方法是：使用 XMLELEMENT 函数。例如，

```
SELECT XMLELEMENT( NAME root,
                   (SELECT emp_id AS id, emp_fname AS
name
                    FROM employee FOR XML AUTO ) )
```

有关 XMLELEMENT 函数的详细信息，请参见 『ASA SQL 参考』> 「XMLELEMENT 函数 [String]」。

通过指定别名，您可以更改 XML 文档中使用的属性名或元素名。下面的查询会将 id 属性重命名为 product_id：

```
SELECT id AS product_id
 FROM product
 WHERE color='black'
 FOR XML AUTO
```

生成的 XML 如下：

```
<product product_id="302"/>
 <product product_id="400"/>
 <product product_id="501"/>
 <product product_id="700"/>
```

您也可以用别名来重命名表。下面的查询将表重命名为 product_info：

```
SELECT id AS product_id
 FROM product AS product_info
 WHERE color='black'
 FOR XML AUTO
```

生成的 XML 如下：

```
<product_info product_id="302"/>
 <product_info product_id="400"/>
 <product_info product_id="501"/>
 <product_info product_id="700"/>
```

示例

下面的查询生成既包含 <employee> 元素又包含 <department> 元素的 XML，而 <employee> 元素 （在选择列表中列在第一位的表）是 <department> 元素的父级。

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 FOR XML AUTO
```

上面的查询会生成以下 XML：

```
<employee emp_id="102">
    <department dept_name="R &amp; D"/>
 </employee>
 <employee emp_id="105">
    <department dept_name="R &amp; D"/>
 </employee>


 <employee emp_id="160">
    <department dept_name="R &amp; D"/>
 </employee>
 <employee emp_id="243">
    <department dept_name="R &amp; D"/>
 </employee>
 ...
```

如果您按以下所示更改选择列表中列的顺序：

```
SELECT department.dept_name, employee.emp_id
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY 1, 2
 FOR XML AUTO
```

结果就会像下面这样嵌套：

```
<department dept_name="R &amp; D">
    <employee emp_id="102"/>
    <employee emp_id="105"/>
    <employee emp_id="160"/>
    ...
 </department>
```

```
<department dept_name="Sales">
    <employee emp_id="129"/>
    <employee emp_id="195"/>
    <employee emp_id="299"/>
    ...
 </department>
 ...
```

为该查询生成的 XML 还是既包含 <employee> 元素又包含 <department> 元素，但这一次， <department> 元素是 <employee> 元素的父级。

使用 FOR XML EXPLICIT

使用 FOR XML EXPLICIT，您可以控制查询返回的 XML 文档的结构。查询必须用特定的方法编写，让查询结果中指定有关您需要的嵌套的信息。使用 FOR XML EXPLICIT 支持的可选指令，您可以配置各列的处理方式。例如，您可以控制让列显示为元素或显示为属性内容，或者，列只用来给结构排序，而不出现在生成的 XML 中。

有关如何使用 FOR XML EXPLICIT 编写查询的示例，请参见 「编写 EXPLICIT 模式的查询」第 571 页。

参数

在 EXPLICIT 模式中，SELECT 语句中的前两列必须分别命名为 **Tag** 和 **Parent**。 Tag 和 Parent 是元数据列，它们的值用来确定查询返回的 XML 文档中元素的父子关系，也就是嵌套。

- **Tag 列** 这是在选择列表中指定的第一列。 Tag 列存储当前元素的标记值。标记号可以使用的值是 1 到 255。
- **Parent 列** 这一列存储当前元素的父级的标记号。如果这一列中的值是 NULL，该行就会被放置在 XML 层次结构的顶层。

例如，有这样一个查询，在没有指定 FOR XML EXPLICIT 时，它会返回下面的结果集。（**first_name!1** 和 **id!2** 数据列的用途会在下一节 「给查询添加数据列」第 570 页中介绍）。

Tag	Parent	first_name!1	id!2
1	NULL	'Beth'	NULL

Tag	Parent	first_name!1	id!2
2	NULL	NULL	'102'

在这个示例中， Tag 列中的值是结果集中各元素的标记号。两行的 Parent 列都包含 NULL 值。这就是说，两个元素都会在层次结构的顶层生成，在查询包括 FOR XML EXPLICIT 子句时，会给出以下结果：

```
<first_name>Beth</first_name>
 <id>102</id>
```

但是，如果第二行在 Parent 列中的值是 1，结果看起来就会像下面这样：

```
<first_name>Beth
    <id>102</id>
 </first_name>
```

有关如何使用 FOR XML EXPLICIT 编写查询的示例，请参见 「编写 EXPLICIT 模式的查询」第 571 页。

给查询添加数据列

除了 Tag 列和 Parent 列以外，查询还必须包含一个或多个数据列。这些数据列的名称控制着在标记过程中解释列的方式。每个列名都拆分为由惊叹号 (!) 分隔的字段。可以为数据列指定以下字段：

ElementName!*TagNumber*!*AttributeName*!*Directive*

ElementName 元素的名称。对于给定的行，会从标记号匹配的第一列的 *ElementName* 字段中提取为该行生成的元素的名称。如果有多个 *TagNumber* 相同的列，对于后面的 *TagNumber* 相同的列，会忽略 *ElementName*。在上面的示例中，第一行会生成一个名为 <first_name> 的元素。

TagNumber 元素的标记号。对于有给定标记值的行，所有在其 *TagNumber* 字段中具有相同值的列都会向对应该行的元素提供内容。

AttributeName 指定列值是 *ElementName* 元素的属性。例如，如果数据列具有名称 prod_id!1!color，那么 color 就会显示为 <prod_id> 元素的属性。

Directive 使用这个可选字段，您可以进一步控制 XML 文档的格式。您可以为 *Directive* 指定以下值中的任何一个：

- **hide** 指明此列会为生成结果而被忽略。这条指令可以用于加入只用来给表排序的列。属性名会被忽略，而且不出现在结果中。

  有关使用 **hide** 指令的示例，请参见 「使用 hide 指令」第 578 页。

- **element** 指明列值作为使用名称 *AttributeName* 的嵌套元素插入，而不是作为属性。

  有关使用 **element** 指令的示例，请参见 「使用 element 指令」第 576 页。

- **xml** 指明列值插入时不加引号。如果指定了 *AttributeName*，则值作为使用该名称的元素插入。否则，它在插入时没有包装元素。如果不使用此指令，标记字符就会被转义，除非列是 XML 类型。例如，值 **<a/>** 会插入为 **<a/>**。

  有关使用 **xml** 指令的示例，请参见 「使用 xml 指令」第 579 页。

- **cdata** 指明列值要作为 CDATA 节插入。 *AttributeName* 会被忽略。

  有关使用 **cdata** 指令的示例，请参见 「使用 cdata 指令」第 580 页。

用法

在您执行包含 FOR XML EXPLICIT 的查询时， BINARY、 LONG BINARY、 IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。缺省情况下，结果集中的任何 NULL 值都会被忽略。通过更改 FOR_XML_NULL_TREATMENT 选项的设置，您可以更改这种行为。

有关 FOR_XML_NULL_TREATMENT 选项的详细信息，请参见『ASA 数据库管理指南』> 「FOR_XML_NULL_TREATMENT 选项 [database]」和 「FOR XML 和 NULL 值」第 560 页。

编写 EXPLICIT 模式的查询

假定您要使用 FOR XML EXPLICIT 编写一个生成以下 XML 文档的查询：

```
<employee emp_id='129'>
    <customer cust_id='107' region='Eastern'/>
    <customer cust_id='119' region='Western'/>
    <customer cust_id='131' region='Eastern'/>
 </employee>


 <employee emp_id='195'>
    <customer cust_id='109' region='Eastern'/>
    <customer cust_id='121' region='Central'/>
 </employee>
```

实现这一点的方法是：编写一条按指定的准确顺序返回以下结果集的 SELECT 语句，然后将 FOR XML EXPLICIT 附加到该查询。

Tag	Parent	employee!1!emp_id	customer!2!cust_id	customer!2!region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern
2	1	129	119	Western
2	1	129	131	Central
1	NULL	195	NULL	NULL
2	1	195	109	Eastern
2	1	195	121	Central

在您编写查询时，对于指定的行，只有部分列会成为所生成的 XML 文档的组成部分。只有在 *TagNumber* 字段 （列名中的第二个字段）中的值与 Tag 列中的值匹配时， XML 文档中才会加入一列。

在该示例中，第三列用于两个在其 Tag 列中有值 1 的行。在第四和第五列中，值用于在其 Tag 列中有值 2 的行。元素名从列名中的第一个字段中提取。在这种情况下，会创建 <employee> 元素和 <customer> 元素。

属性名来自列名中的第三个字段，所以 emp_id 属性是为 <employee> 元素创建的，而 cust_id 属性和 region 属性是为 <customer> 元素生成的。

下面的步骤说明了如何使用示例数据库来构建一个 FOR XML EXPLICIT 查询，让该查询生成一个与本节开始时出现的 XML 文档类似的 XML 文档。

❖ 编写 FOR XML EXPLICIT 查询

1　编写 SELECT 语句以生成顶层元素。

在本示例中，查询中的第一个 SELECT 语句生成 <employee> 元素。查询中的前两个值必须是 Tag 和 Parent 列值。<employee> 元素位于层次结构的顶层，所以会给它分配一个 Tag 值 1 以及一个 Parent 值 NULL。

注意
如果您要编写一个使用 UNION 的 EXPLICIT 模式查询，则只有第一条 SELECT 语句中指定的列名才会被使用。不会用作元素名或属性名的列名必须在第一条 SELECT 语句中指定，因为在后面的 SELECT 语句中指定的列名会被忽略。

要为上面的表生成 <employee> 元素，您的第一条 SELECT 语句要像下面这样：

```
SELECT
       1       AS tag,
       NULL    AS parent,
       emp_id  AS [employee!1!emp_id],
       NULL    AS [customer!2!cust_id],
       NULL    AS [customer!2!region]
 FROM employee
```

2　编写 SELECT 语句来生成子元素。

第二个查询会生成 <customer> 元素。由于这是一个 EXPLICIT 模式查询，所以在所有 SELECT 语句中指定的前两个值都必须是 Tag 值和 Parent 值。 <customer> 元素会得到标记号 2，而由于它是 <employee> 元素的子级，所以它具有 Parent 值 1。第一条 SELECT 语句已经指定了 emp_id、 cust_id 和 region 都是属性。

```
SELECT
       2,
       1,
       emp_id,
       cust_id,
       region
 FROM employee KEY JOIN sales_order
```

3 给查询添加 UNION ALL 以便将两条 SELECT 语句合并到一起：

```
SELECT
       1       AS tag,
       NULL    AS parent,
       emp_id  AS [employee!1!emp_id],
       NULL    AS [customer!2!cust_id],
       NULL    AS [customer!2!region]
 FROM employee
 UNION ALL

SELECT
       2,
       1,
       emp_id,
       cust_id,
       region
 FROM employee KEY JOIN sales_order
```

4 添加 ORDER BY 子句来指定结果中行的顺序。行的顺序是最终生成的文档中使用的顺序。

```
SELECT
       1       AS tag,
       NULL    AS parent,
       emp_id  AS [employee!1!emp_id],
       NULL    AS [customer!2!cust_id],
       NULL    AS [customer!2!region]
 FROM employee
 UNION ALL
```

```
    SELECT
           2,
           1,
           emp_id,
           cust_id,
           region
     FROM employee KEY JOIN sales_order
     ORDER BY 3, 1
     FOR XML EXPLICIT
```

有关 EXPLICIT 模式的语法的信息，请参见 「参数」第 569 页。

FOR XML EXPLICIT 示例

下面的示例查询会检索有关雇员所下订单的信息。在本示例中，有三种元素：<emp>、 <order> 和 <dept>。 <emp> 元素具有 id 属性和 name 属性， <order> 元素具有 date 属性， <dept> 元素具有 name 属性。

```
SELECT
         1           tag,
         NULL        parent,
         emp_id      [emp!1!id],
         emp_fname   [emp!1!name],
         NULL        [order!2!date],
         NULL        [dept!3!name]
 FROM employee
 UNION ALL

SELECT
         2,
         1,
         emp_id,
         NULL,
         order_date,
         NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
         3,
         1,
         emp_id,
         NULL,
         NULL,
```

```
        dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

利用此查询可以得到以下结果：

```
<emp id="102" name="Fran">
    <dept name="R &amp;amp; D"/>
 </emp>
 <emp id="105" name="Matthew">
    <dept name="R &amp;amp; D"/>
 </emp>

<emp id="129" name="Philip">
    <order date="2000-07-24"/>
    <order date="2000-07-13"/>
    <order date="2000-06-24"/>
    <order date="2000-06-08"/>
    ...
    <dept name="Sales"/>
 </emp>

<emp id="148" name="Julie">
    <dept name="Finance"/>
 </emp>
 ...
```

使用 element 指令

如果您想生成子元素而不是属性，可以给查询添加 **element** 指令，就像下面这样：

```
SELECT
         1          tag,
         NULL       parent,
         emp_id     [emp!1!id!element],
         emp_fname  [emp!1!name!element],
         NULL       [order!2!date!element],
         NULL       [dept!3!name!element]
 FROM employee

UNION ALL
 SELECT
         2,
         1,
         emp_id,
```

```
            NULL,
            order_date,
            NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
            3,
            1,
            emp_id,
            NULL,
            NULL,
            dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

利用此查询可以得到以下结果：

```
<emp>
    <id>102</id>
    <name>Fran</name>
    <dept>
       <name>R &amp; D</name>
    </dept>
 </emp>

<emp>
    <id>105</id>
    <name>Matthew</name>
    <dept>
       <name>R &amp; D</name>
    </dept>
 </emp>

<emp>
    <id>129</id>
    <name>Philip</name>
    <order>
       <date>2000-07-24</date>
    </order>
    <order>
       <date>2000-07-13</date>
    </order>
    <order>
       <date>2000-06-24</date>
```

```
   </order>
   ...
   <dept>
      <name>Sales</name>
   </dept>
</emp>
...
```

使用 hide 指令

在下面的查询中，雇员 ID 用于给结果排序，但结果中不会出现雇员 ID，因为指定了 **hide** 指令：

```
SELECT
         1           tag,
         NULL        parent,
         emp_id      [emp!1!id!hide],
         emp_fname   [emp!1!name],
         NULL        [order!2!date],
         NULL        [dept!3!name]
 FROM employee
 UNION ALL

SELECT
         2,
         1,
         emp_id,
         NULL,
         order_date,
         NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
         3,
         1,
         emp_id,
         NULL,
         NULL,
         dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

此查询会返回以下结果：

```
<emp name="Fran">
    <dept name="R &amp; D"/>
 </emp>
 <emp name="Matthew">
    <dept name="R &amp; D"/>
 </emp>

<emp name="Philip">
    <order date="2000-07-24"/>
    <order date="2000-07-13"/>
    <order date="2000-06-24"/>
    <order date="2000-06-08"/>
    ...
    <dept name="Sales"/>
 </emp>

<emp name="Julie">
    <dept name="Finance"/>
 </emp>
 ...
```

使用 xml 指令

缺省情况下，当 FOR XML EXPLICIT 查询的结果包含不是有效 XML 字符的字符时，无效字符会被转义（有关信息请参见「无效的列名」第 561 页），除非该列是 XML 类型。例如，下面的查询会生成包含 "&" 符号的 XML：

```
SELECT
         1               AS tag,
         NULL            AS parent,
         id              AS [customer!1!id!element],
         company_name    AS [customer!1!company_name]
 FROM customer
 WHERE id = '115'
 FOR XML EXPLICIT
```

在此查询生成的结果中， "&" 符号会被转义，因为该列不是 XML 类型：

```
<customer company_name="Sterling &amp;amp; Co.">
    <id>115</id>
 </customer>
```

xml 指令表明列值在插入生成的 XML 中时不带引号。如果您用 **xml** 指令执行与上面的查询相同的查询：

```
SELECT
        1               AS tag,
        NULL            AS parent,
        id              AS [customer!1!id!element],
        company_name    AS
[customer!1!company_name!xml]
 FROM customer
 WHERE id = '115'
 FOR XML EXPLICIT
```

结果中的 "&" 号就不会带有引号：

```
<customer>
  <id>115</id>
  <company_name>Sterling & Co.</company_name>
 </customer>
```

请注意，此 XML 并非结构完好，因为它包含 "&" 号，而这在 XML 中是一个特殊字符。当查询生成了 XML 时，您有责任确保该 XML 结构完好并且有效，因为 Adaptive Server Anywhere 不会检查所生成的 XML 是否结构完好或有效。

当您指定 **xml** 指令时， *AttributeName* 字段会被忽略，并且会生成元素而不是属性。

使用 cdata 指令

下面的查询使用 **cdata** 指令在 CDATA 节中返回客户名称：

```
SELECT
        1               AS tag,
        NULL            AS parent,
        id              AS [product!1!id],
        description     AS [product!1!cdata]
 FROM product
 FOR XML EXPLICIT
```

此查询生成的结果会在 CDATA 节中列出对每种产品的说明。 CDATA 节中包含的数据不带引号：

```
<product id="300">
    <![CDATA[Tank Top]]>
 </product>
 <product id="301">
    <![CDATA[V-neck]]>
 </product>
```

```
<product id="302">
    <![CDATA[Crew Neck]]>
 </product>
 <product id="400">
    <![CDATA[Cotton Cap]]>
 </product>
 ...
```

使用 SQL/XML 以 XML 的形式获取查询结果

SQL/XML 是一个描述 XML 与 SQL 语言有效集成的标准草案：它描述了能够将 SQL 与 XML 一起使用的方法。使用受支持的函数，您可以编写查询来利用关系数据构建 XML 文档。

无效的名称和 SQL/XML

在 SQL/XML 中，不是合法 XML 名称的表达式 （例如，包含空格的表达式）会被转义，方式与处理 FOR XML 子句一样。 XML 类型的元素内容不会加上引号。

有关给无效表达式加引号的详细信息，请参见 「无效的列名」第 561 页。

有关使用 XML 数据类型的信息，请参见 「在关系数据库中存储 XML 文档」第 547 页。

使用 XMLAGG 函数

XMLAGG 函数用于从 XML 元素的集合中生成 XML 元素的林。XMLAGG 是一个集合函数，它会为查询中的所有行生成一个集合的 XML 结果。

在下面的查询中， XMLAGG 用于为每行生成一个 <name> 元素，而 <name> 元素会按雇员姓名排序。指定 ORDER BY 子句是为了给 XML 元素排序：

```
SELECT XMLELEMENT( NAME department,
                   XMLATTRIBUTES (dept_id ),
                   XMLAGG( XMLELEMENT( NAME name,
                             emp_lname )
                             ORDER BY emp_lname )
                  ) AS dept_list
 FROM employee
 GROUP BY dept_id
```

此查询会产生以下结果：

dept_list
<department dept_id="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </department>
<department dept_id="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </department>
<department dept_id="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </department>
...

有关 XMLAGG 函数的详细信息，请参见 『ASA SQL 参考』> 「XMLAGG 函数 [String]」。

使用 XMLCONCAT 函数

XMLCONCAT 函数通过连接所有传入的 XML 值来创建 XML 元素林。例如，下面的查询会将雇员表中的每一名雇员的 <first_name> 元素和 <last_name> 元素连接起来：

```
SELECT XMLCONCAT( XMLELEMENT( NAME first_name,
emp_fname ),
                  XMLELEMENT( NAME last_name, emp_lname
)
                   ) AS "Employee_Name"
 FROM employee
```

此查询会返回以下结果：

Employee_Name
<first_name>Fran</first_name> <last_name>Whitney</last_name>
<first_name>Matthew</first_name> <last_name>Cobb</last_name>
<first_name>Philip</first_name> <last_name>Chin</last_name>
<first_name>Julie</first_name> <last_name>Jordan</last_name>
...

有关详细信息，请参见 『ASA SQL 参考』> 「XMLCONCAT 函数 [String]」。

使用 XMLELEMENT 函数

XMLELEMENT 函数利用关系数据来构建 XML 元素。您可以指定所生成元素的内容，而且，如果需要，还可以为该元素指定属性和属性内容。

生成嵌套元素

下面的查询会生成嵌套的 XML，也就是为每个产品生成一个 <product_info> 元素，其中具有提供每种产品的名称、数量和说明的元素：

```
SELECT id,
        XMLELEMENT( NAME product_info,
        XMLELEMENT( NAME item_name, product.name ),
        XMLELEMENT( NAME quantity_left,
product.quantity ),
```

```
        XMLELEMENT( NAME description, product.size || '
' ||
                                    product.color || ' ' ||
                                product.name )
       ) AS results
FROM product
WHERE quantity > 30
```

此查询会产生以下结果：

id	results
301	<product_info> <item_name>Tee Shirt</item_name> <quantity_left>54</quantity_left> <description>Medium Orange Tee Shirt</description> </product_info>
302	<product_info> <item_name>Tee Shirt</item_name> <quantity_left>75</quantity_left> <description>One size fits all Black Tee Shirt</description> </product_info>
400	<product_info> <item_name>Baseball Cap</item_name> <quantity_left>112</quantity_left> <description>One size fits all Black Baseball Cap</description> </product_info>
…	…

指定元素内容

使用 XMLELEMENT 函数，您可以指定元素的内容。下面的语句会产生带有内容 **hat** 的 XML 元素。

```
SELECT id, XMLELEMENT( NAME product_type, 'hat' )
 FROM product
 WHERE name IN ( 'Baseball Cap', 'Visor' )
```

生成有属性的元素

通过在查询中加入 XMLATTRIBUTES 参数，可以为元素添加属性。此参数指定属性名称和内容。下面的语句会为每个项目的名称、颜色和单价生成一个属性。

```
SELECT id, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES( name,
                                      color,
                                      unit_price )
                     ) AS item_description_element
 FROM product
 WHERE id > 400
```

可以通过指定 AS 子句来命名属性：

```
SELECT id, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES ( unit_price AS
            price ),
                       product.name
                     ) AS products
 FROM product
 WHERE id > 400
```

有关详细信息，请参见 『ASA SQL 参考』> 「XMLELEMENT 函数 [String]」。

使用 XMLFOREST 函数

使用 XMLFOREST 可以构建 XML 元素林。对于每个 XMLFOREST 参数，都会生成一个元素。

下面的查询会生成一个 <item_description> 元素，它具有 <name>、<color> 和 <price> 元素：

```
SELECT id, XMLELEMENT( NAME item_description,
                        XMLFOREST( name,
                                   color,
                                   unit_price AS price )
                      ) AS product_info
 FROM product
 WHERE id > 400
```

此查询会生成以下结果：

id	product_info
401	<item_description> <name>Baseball Cap</name> <color>White</color> <price>10.00</price> </item_description>
500	<item_description> <name>Visor</name> <color>White</color> <price>7.00</price> </item_description>
501	<item_description> <name>Visor</name> <color>Black</color> <price>7.00</price> </item_description>
...	...

有关详细信息，请参见 『ASA SQL 参考』> 「XMLFOREST 函数 [String]」。

使用 XMLGEN 函数

XMLGEN 函数用于在 XQuery 构造函数的基础上生成 XML 值。

下面的查询生成的 XML 提供了有关示例数据库中客户订单的信息。它使用以下变量引用：

- **{$id}** 使用 sales_order 表中的 id 列中的值为 <id> 元素生成内容。
- **{$order_date}** 使用 sales_order 表中的 order_date 列中的值为 <date> 元素生成内容。
- **{$customer}** 使用 customer 表中的 company_name 列中的值为 <customer> 元素生成内容。

```
SELECT XMLGEN ( '<order>
                  <id>{$id}</id>
                  <date>{$order_date}</date>
                  <customer>{$customer}</customer>
                  </order>',
                 sales_order.id,
                 sales_order.order_date,
                 customer.company_name AS customer)
  FROM sales_order JOIN customer
  ON customer.id = sales_order.cust_id
```

此查询会生成以下结果：

order_info
<order> <id>2131</id> <date>2000-01-02</date> <customer>BoSox Club</customer> </order>
<order> <id>2126</id> <date>2000-01-03</date> <customer>Leisure Time</customer> </order>
<order> <id>2065</id> <date>2000-01-03</date> <customer>Bloomfield's</customer> </order>

order_info
<order>   <id>2127</id>   <date>2000-01-06</date>   <customer>Creative Customs Inc.</customer> </order>
...

生成属性

如果您希望订单 ID 号显示为 <order> 元素的属性，要按下面这样编写查询（请注意，变量引用包含在双引号中，因为它指定了一个属性值）：

```
SELECT XMLGEN ( '<order id="{$id}">
                  <date>{$order_date}</date>
                  <customer>{$customer}</customer>
                  </order>',
                 sales_order.id,
                 sales_order.order_date,
                 customer.company_name AS customer
               ) AS order_info

FROM sales_order JOIN customer
  ON customer.id = sales_order.cust_id
  ORDER BY sales_order.order_date
```

此查询会生成以下结果：

order_info
<order id="2131">   <date>2000-01-02</date>   <customer>BoSox Club</customer> </order>
<order id="2126">   <date>2000-01-03</date>   <customer>Leisure Time</customer> </order>
<order id="2065">   <date>2000-01-03</date>   <customer>Bloomfield's</customer> </order>

order_info
<order id="2127"> <date>2000-01-06</date> <customer>Creative Customs Inc.</customer> </order>
...

在两个结果集中，客户名 Bloomfield's 都加上了引号，变成了 **Bloomfield's**，因为撇号在 XML 中是一个特殊符号，而且从中生成 <customer> 元素的列不是 XML 类型。

有关给 XMLGEN 中的非法字符加引号的详细信息，请参见 「无效的名称和 SQL/XML」第 582 页。

指定 XML 文档的标头信息

Adaptive Server Anywhere 支持的 FOR XML 子句和 SQL/XML 函数不会在它们生成的 XML 文档中加入版本声明信息。您可以使用 XMLGEN 函数来生成标头信息。

```
SELECT XMLGEN( '<?xml version="1.0"
                encoding="ISO-8859-1" ?>
                <r>{$x}</r>',
               (SELECT fname, lname FROM customer FOR
XML RAW) AS x )
```

这样就会产生以下结果：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
 <r>
  <row fname="Michael" lname="Devlin"/>
  <row fname="Beth" lname="Reiser"/>
  <row fname="Erin" lname="Niedringhaus"/>
  <row fname="Meghan" lname="Mason"/>
  ...
 </r>
```

有关 XMLGEN 函数的详细信息，请参见 『ASA SQL 参考』> 「XMLGEN 函数 [String]」。

第 5 部分　远程数据和批量操作

这一部分介绍了如何装载和卸载数据库以及如何访问远程数据。

第 17 章

导入和导出数据

关于本章 本章介绍将批量数据传入与传出 Adaptive Server Anywhere 数据库的过程。

将数据传入与传出数据库

在许多情况下都可能需要将数据传入或传出数据库。例如：

- 将一组初始数据导入一个新数据库。
- 建立数据库的新副本，该副本可能具有已修改的结构。
- 将数据从您的数据库导出以用于其它应用程序，例如电子表格。
- 创建数据库的抽取用于复制或同步。
- 修复损坏的数据库。
- 重建数据库以提高其性能。
- 获取数据库软件的更新版本并完成软件升级。

这些任务经常作为**批量操作**组合在一起。它们由具有 **DBA 权限**的用户执行并且不属于典型的最终用户应用程序。批量操作可能对并发和事务日志有特定的要求，并且通常最好在无其他用户连接到数据库时执行。

批量操作的性能问题

批量操作的性能取决于该操作对数据库服务器来说是内部操作还是外部操作。

数据库服务器外部的：客户端

INPUT 和 OUTPUT 命令通过 Interactive SQL 客户端执行。如果 Interactive SQL 不在数据库服务器所在的计算机上运行，则正被读取或写入的文件的路径是相对于客户端，而不是相对于服务器的。 INPUT 对每个读取的行都作为单独的 INSERT 语句被记录在事务日志中。其结果是， INPUT 比 LOAD TABLE 要慢得多。这还意味着在 INPUT 过程中将触发 ON INSERT 触发器。如果缺少某些列的值，在可为空的列上插入空值，在不可为空的数字列上插入 0 （零），在不可为空的非数字列上插入空字符串。当需要考虑兼容性时， OUTPUT 语句十分有用，这是因为它能够以多种文件格式中的任何一种写出 SELECT 语句的结果集。

数据库服务器内部的：服务器端

LOAD TABLE、 UNLOAD TABLE 和 UNLOAD 语句对数据库服务器来说是内部语句。到正被写入或读取的文件的路径是相对于数据库服务器的。 LOAD TABLE 语句作为单个命令记录在事务日志中。数据文件包含的列数必须和将被装载的表相同。如果 DEFAULTS 选项设置为 OFF （缺省设置），则具有缺省值的列上所插入的缺少值将为空值、零或空字符串，或者如果 DEFAULTS 选项设置为 ON，则将插入缺省值。内部导入和导出只提供对文本和 BCP 格式的访问，但它却是一种较快的方法。

有关详细信息，请参见 「导入数据的性能提示」第 598 页。

批量操作的数据恢复问题

您应该在使用批量操作模式的前后对数据库进行备份，这是因为在此模式中您的数据库未受保护，可能由于介质故障而被损坏。如果以**批量操作**模式 （**-b** 选项）运行数据库服务器以导入大量数据，则数据库服务器*不会* 执行某些重要功能。具体来说：

当数据库服务器处于批量操作模式时，数据库服务器不会 …	这具有以下影响 …
保留回退日志	不能使用保存点，并且中止命令总会导致事务回退。
维护事务日志	不对更改进行记录。
锁定所有记录	无严重影响。
在数据定义命令前执行自动 COMMIT	如果没有自动 COMMIT，则 ROLLBACK 将撤消上次显式 COMMIT 之后的所有操作。

如果发生介质故障，则前面的条件会对数据库产生严重影响。因此，

如果您具有需要多次提交的数据，则使用 **-b** 选项运行可能会降低数据库的操作速度。每次执行 COMMIT 时，服务器都执行检查点操作，这会导致数据库服务器变慢。

有关信息，请参见 『ASA 数据库管理指南』> 「-b 服务器选项」和『ASA 数据库管理指南』> 「备份和数据恢复」。

将数据导入数据库

导入是一种管理任务，此任务以批量操作模式将数据读入数据库。Adaptive Server Anywhere 允许您：

- 从 ASCII 文件导入整个表或表的某些部分
- 从其它数据库格式 （如 DBASEII）导入整个表或表的某些部分
- 通过使用脚本自动执行导入过程来连续导入多个表
- 将数据插入或附加到表中
- 替换表中的数据
- 在导入前或导入过程中创建表
- 使用 BCP 格式子句在 Adaptive Server Anywhere 与 Adaptive Server Enterprise 之间导入文件

如果您正在尝试创建一个完整的新数据库，出于性能方面的原因，请考虑装载数据而不要导入数据。

有关卸载并重装整个数据库的详细信息，请参见 「重建数据库」第 620 页。

以下几节将介绍导入工具，随后是导入数据库、数据和表的说明。

导入注意事项

在开始将数据导入数据库前，请花些时间仔细考虑您拥有什么资源，并考虑清楚您要通过将数据导入数据库来完成什么工作。

考虑此因素 …	有关信息，请参见 …
可用的导入工具。	「导入工具」第 598 页

考虑此因素 …	有关信息，请参见 …
导入可能对数据库性能产生的影响。	「批量操作的性能问题」第 595 页 「导入数据的性能提示」第 598 页
数据与目标表之间的兼容性问题。	「用于导入的表结构」第 606 页

导入数据的性能提示

虽然将大量的数据导入到数据库可能要花费很多时间，但是您可以采用某些手段来节省时间：

- 将数据文件与数据库放在不同的物理磁盘驱动器上。这可以避免装载过程中过多的磁头移动。
- 按照 『ASA SQL 参考』> 「ALTER DBSPACE 语句」中的说明扩展数据库的大小。此命令允许在需要空间之前大规模扩展数据库，而不是平时需要空间时一次扩展 256 KB。在提高装载大量数据的性能的同时，将数据库连续保存在文件系统中也很有帮助。
- 使用临时表装载数据。当需要重复装载一组数据或需要合并具有不同结构的表时，本地或全局临时表将十分有用。
- 如果使用 LOAD TABLE 语句，则不要使用 **-b** 选项 （批量操作模式）启动数据库服务器，因为这是不必要的。
- 如果正在使用 INPUT 或 OUTPUT 命令，则在数据库服务器所在的计算机上运行 Interactive SQL 或客户端应用程序。通过网络装载数据将增加额外的通信开销。这可能意味着需要在下班时间装载新数据。

导入工具

可以使用多种工具将数据导入 Adaptive Server Anywhere 数据库。下面将讨论各种导入工具。

使用 Interactive SQL 导入向导

Interactive SQL 导入向导允许您选择要导入的文件、文件格式以及要在其中放置数据的目标表。您可以导入文本、 DBASEII、 DBASEIII、 Excel 2.1、 FOXPRO 和 Lotus 等格式的数据。可以选择将数据导入现有表中，或使用该向导创建并配置全新的表。

当处于以下情况时，请考虑使用 Interactive SQL 导入向导：

- 喜欢使用图形界面以非文本格式导入数据。
- 要在导入数据的同时创建表。

示例 1　此示例显示如何使用 Interactive SQL [数据] 菜单导入数据。

❖ 导入数据 （Interactive SQL 的 [数据] 菜单）

1. 在 [Interactive SQL] 窗口中，选择 [数据] → [导入]。

   即会出现 [打开] 对话框。

2. 定位到您要导入的文件，然后单击 [打开]。

   即会出现导入向导。

3. 指定数据库值在导入文件中的存储方式。

4. 选择 [使用现有的表] 选项，然后输入现有表的名称和位置。单击 [下一步]。

   您还可以单击 [浏览] 按钮并定位到要向其中导入数据的表。

5. 对于 ASCII 文件，您可以指定文件的字段分隔符、转义字符、是否应包括尾随空白，以及文件的编码。使用编码选项，您可以指定用于读取文件的代码页，以确保字符得到正确处理。如果您选择 **（缺省值）**，则会使用运行 Interactive SQL 的计算机的缺省编码。

   有关支持的代码页的信息，请参见 『ASA 数据库管理指南』> 「支持的代码页」。

6. 请按照向导中的其余说明进行操作。

导入会将新数据附加到现有的表中。如果导入成功，[消息] 窗格将显示导入数据花费的时间。如果导入不成功，则将显示一条消息指示导入不成功。

示例 2

下面的示例显示如何将数据添加到 ASA 9.0 示例数据库的 employee 表中。此示例添加一个名为 Julia Chan 的雇员的有关信息。

❖ 将数据导入示例数据库

1. 使用下面的值 （在同一行上输入）创建并保存一个名为 *import.txt* 的文本文件：

   ```
   100,500,'Julia','Chan',
    100, '300 Royal
   Drive','Springfield','OR','97015','6175553985',
    'A','017239033', 55700,'1984-09-29',,'1968-05-
   05','Y','Y','N','F'
   ```

2. 在 Interactive SQL 中，选择 [数据] → [导入]。

3. 找到 *import.txt* 文件并单击 [打开]。

   即会出现导入向导。

4. 指定数据库值在导入文件中的存储方式。

5. 选择 [使用现有的表] 选项，然后输入 **employee** 作为要向其中导入数据的表的名称。单击 [下一步]。

6. 请按照导入向导中的其余步骤进行操作。

使用 INPUT 语句导入数据

INPUT 语句允许您将多种文件格式的数据导入一个或多个表中。您可以选择缺省输入格式，也可以在每个 INPUT 语句上指定文件格式。Interactive SQL 可以执行包含多个 INPUT 语句的命令文件。

当处于以下情况时，请考虑使用 INPUT 语句导入数据：

- 要将数据导入到一个或多个表中。

- 要使用命令文件自动执行导入过程。
- 要以非文本格式导入数据。

有关信息，请参见 『ASA SQL 参考』> 「INPUT 语句 [Interactive SQL]」。

对数据库的影响

将更改记录在事务日志中。如果发生介质故障，则会记录更改。但是，由于 INPUT 语句将所有内容都写入事务日志，因此使用 INPUT 语句导入大量数据会对性能产生影响。

INPUT 语句的执行速度比 LOAD TABLE 语句更慢。如果数据文件的格式为 DBASE、 DBASEII、 DBASEIII、 FOXPRO 或 LOTUS 并且表不存在，则将创建表。

示例

此示例显示如何使用 Interactive SQL INPUT 语句导入数据。

❖ 导入数据 （Interactive SQL INPUT 语句）：

1. 确保要在其中放置数据的表存在。
2. 在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中输入 INPUT 语句。例如，

   ```
   INPUT INTO t1
    FROM file1
    FORMAT ASCII;
   ```

   其中 `t1` 是要在其中放置数据的表的名称， `file1` 是保存要导入的数据的文件的名称。
3. 执行该语句。

   如果导入成功， [消息] 窗格将显示导入数据花费的时间。如果导入不成功，则将显示一条消息指示导入不成功。

有关使用 INPUT 语句导入数据的信息，请参见 『ASA SQL 参考』> 「INPUT 语句 [Interactive SQL]」。

以下是一个 ASCII 文本文件中的 INPUT 语句的示例。

```
INPUT INTO employee
 FROM new_emp.inp
 FORMAT ASCII;
```

使用 LOAD TABLE 语句导入数据

LOAD TABLE 语句只允许以有效的方式将数据导入到表中，导入数据的格式为文本 /ASCII/FIXED。表必须存在并且表的列数必须和输入文件的字段数相同，而且所定义的数据类型必须兼容。使用 LOAD TABLE 语句导入时，文件中的每行在数据库表中占一行，各个值之间由分隔符隔开。

当处于以下情况时，请考虑使用 LOAD TABLE 语句：

- 要以文本格式导入数据。
- 要提高性能并且可以使用 LOAD TABLE 语句代替 INPUT 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「LOAD TABLE 语句」。

对数据库的影响

*不*将更改记录在事务日志中。如果需要恢复数据，则将不能使用此信息。数据在崩溃时丢失，从而导致此语句不能用于 SQL Remote 或 MobiLink 远程数据库。

LOAD TABLE 语句的执行速度比 INPUT 语句快得多。

LOAD TABLE 语句将文件的内容附加到表中现有的行；它不替换表中现有的行。

使用 INSERT 语句导入数据

INSERT 语句允许您将行添加到数据库中。由于您将要放在表中的数据直接包括在 INSERT 语句中，因此这被认为是交互式输入。您还可以使用具有远程数据访问功能的 INSERT 语句从另一个数据库 （而不是文件）导入数据。

当处于以下情况时，请考虑使用 INSERT 语句导入数据：

- 要将少量数据导入到单个表中。
- 文件格式不是什么问题。

- 要从外部数据库而不是从文件导入远程数据。

有关信息，请参见 『ASA SQL 参考』> 「INSERT 语句」。

对数据库的影响

将更改记录在事务日志中。这意味着如果发生介质故障，则可以从事务日志中恢复关于所做更改的信息。有关详细信息，请参见 『ASA 数据库管理指南』> 「事务日志」。

示例

此示例显示如何使用 INSERT 语句导入数据。使用此 INSERT 语句，将数据添加到示例数据库的 department 表中。

❖ 导入数据 （INSERT 语句）：

1. 确保要在其中放置数据的表存在。
2. 执行 INSERT 语句。例如，

```
INSERT INTO t1
 VALUES ( ... )
```

插入值会将新数据附加到现有的表上。

下面的示例将一个新行插入到示例数据库的 **department** 表中。

```
INSERT
 INTO department ( dept_id, dept_name, dept_head_id )
 VALUES ( 700, 'Training', 501)
```

使用代理表导入数据

代理表是一种本地表，它所包含的元数据可以像访问本地表一样访问远程数据库服务器上的表。这样您便可以直接导入数据。

当处于以下情况时，请考虑使用代理表导入数据：

- 有权访问远程数据。
- 您要直接从另一个数据库导入数据。

如何使用代理表

创建代理表，然后使用带 SELECT 子句的 INSERT 语句将数据从远程数据库插入到您的数据库的永久表中。

有关远程数据访问的信息，请参见 「访问远程数据」第 641 页。

有关 INSERT 语句的信息，请参见 『ASA SQL 参考』> 「INSERT 语句」。

装载工具

可以使用以下任一种工具来帮助您将数据装载到数据库中：

- 「使用 INSERT 语句导入数据」第 602 页
- 「使用 LOAD TABLE 语句导入数据」第 602 页

处理导入过程中的转换错误

当从外部源装载数据时，数据中可能有错误。例如，可能有某些日期不是有效的日期，某些数字不是有效的数字。 CONVERSION_ERROR 数据库选项允许您忽略这些转换错误，该选项会将它们转换为空值。

有关设置 Interactive SQL 数据库选项的详细信息，请参见 『ASA SQL 参考』> 「SET OPTION 语句」或 『ASA 数据库管理指南』> 「CONVERSION_ERROR 选项 [compatibility]」。

导入表

可以使用 Interactive SQL 的 [数据] 菜单或 [语句] 窗格将表导入到现有数据库中。表数据可以是文本、 DBASEII、 Excel 2.1、 FOXPRO 和 Lotus 等格式。

❖ 导入表 （Interactive SQL 的 [数据] 菜单）

1. 确保要在其中放置数据的表存在。
2. 在 [Interactive SQL] 窗口中，选择 [数据] → [导入]。

   即会出现 [打开] 对话框。
3. 定位到您要导入的文件，然后单击 [打开]。

即会出现导入向导。

4　选择 [使用以下名称新建表] 选项，然后在字段中输入新表的名称。

5　请按照向导中的其余说明进行操作。

如果导入成功，[消息] 窗格将显示导入数据花费的时间。如果导入不成功，则将显示一条消息指示导入不成功。[结果] 窗格中的 [计划] 选项卡显示所使用的执行计划。

❖ 导入表 (Interactive SQL)：

1　在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中，创建要向其中装载数据的表。

您可以使用 CREATE TABLE 语句创建该表。

2　执行 LOAD TABLE 语句。例如，

```
LOAD TABLE department
 FROM 'dept.txt'
```

3　要在值中保留尾随空白，请在 LOAD TABLE 语句中使用 STRIP OFF 子句。缺省设置 (STRIP ON) 在插入值之前从值中去除尾随空白。

LOAD TABLE 语句将文件的内容附加到表中现有的行；它不替换表中现有的行。您可以使用 TRUNCATE TABLE 语句从表中删除所有行。

TRUNCATE TABLE 语句和 LOAD TABLE 语句都不触发触发器，包括参照完整性操作 （例如级联删除）。

有关 LOAD TABLE 语句的语法的详细信息，请参见 『ASA SQL 参考』 > 「LOAD TABLE 语句」。

用于导入的表结构

要装载到表中的数据的结构不需要与目标表本身的结构相匹配。例如，列数据类型可能不同或顺序不同，或者导入数据中可能存在超出范围的值，这些值与目标表中的列不匹配。

重新排列表或数据

如果您知道要导入的数据的结构与目标表的结构不匹配，则有多种方法可供选择：

- 可以使用 LOAD TABLE 语句重新排列表中的列。
- 您可以使用 INSERT 语句的变形形式和一个全局临时表重新排列导入数据以适应表的结构。
- 可以使用 INPUT 语句指定列的特定集合或顺序。

允许列包含 NULL 值

如果正在导入的文件包含表中列的子集的数据，或者如果列的顺序不同，则您还可以使用 LOAD TABLE 语句 DEFAULTS 选项填充空白并合并非匹配表结构。

如果 DEFAULTS 为 OFF，则将任何未显示在列列表中且可为空的列置为空值。如果 DEFAULTS 为 OFF 并且列列表中省略了不可为空的列，则数据库服务器会试图将空字符串转换为该列的类型。如果 DEFAULTS 为 ON 并且列具有缺省值，则使用该值。

例如，如果要将两列装载到 employee 表中，并将其余列值设置为缺省值 （如果有的话），则 LOAD TABLE 语句应该如下：

```
LOAD TABLE employee (emp_lname, emp_fname)
 FROM 'new_employees.txt'
 DEFAULTS ON
```

合并不同的表结构

您可以使用 INSERT 语句的变形形式和一个全局临时表重新排列导入数据以适应表的结构。

❖ 使用全局临时表装载具有不同结构的数据：

1 在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中，创建一个全局临时表，该表的结构和输入文件的结构匹配。

您可以使用 CREATE TABLE 语句创建该全局临时表。

2　使用 LOAD TABLE 语句将数据装载到全局临时表中。

在关闭数据库连接后，全局临时表中的数据就会消失。但是，表定义仍然存在。您可以在下次连接到数据库时使用它。

3　使用带有 FROM SELECT 子句的 INSERT 语句从临时表中抽取数据并对其进行汇总，然后将这些数据放入一个或多个永久数据库表中。

将数据从数据库中导出

导出是一种将数据写出数据库的管理任务。如果需要共享数据库中的大部分或依据特定的标准抽取数据库的某些部分，您会发现导出是一个很有用的工具。 Adaptive Server Anywhere 允许您：

- 导出单独的表、查询结果或表模式
- 创建自动执行导出的脚本，以便可以连续导出多个表
- 导出到多种不同的文件格式
- 使用 BCP FORMAT 子句在 Adaptive Server Anywhere 与 Adaptive Server Enterprise 之间导出文件

如果您要导出整个数据库，出于性能方面的原因，请考虑卸载数据库而不要导出数据。

有关卸载并重装整个数据库的详细信息，请参见 「重建数据库」第 620 页。

以下是导出工具的概述，随后是导出查询结果、数据库和表的说明。

导出注意事项

在开始导出数据前，请花些时间仔细考虑您拥有什么资源，并考虑清楚您要将何种类型的信息导出数据库。

考虑此因素 …	有关信息，请参见 …
可用的导出工具。	「导出工具」第 609 页
导出对数据库性能的影响。	「批量操作的性能问题」第 595 页
如何处理 NULL 值。	「输出空值」第 617 页

导出工具

可以使用多种工具帮助您导出数据。

使用 Interactive SQL 导出数据

可以使用 Interactive SQL 导出向导将数据导出从数据库中获取的查询结果集。

当要将结果集保存到文件中时，请使用此导出工具。在 Interactive SQL 中，选择 [数据] → [导出]。允许您选择导出查询结果的文件格式。

示例　下面的示例显示如何将 SQL 查询的结果集导出到文件中。

❖ 使用 Interactive SQL 导出结果集数据

1. 使用 ASA 9.0 示例数据库执行下面的查询。

   ```
   SELECT * FROM employee
    WHERE state = 'GA'
   ```

   该结果集包含所有居住在 Georgia 的雇员的列表。

2. 在 Interactive SQL 中，选择 [数据] → [导出]。
3. 在 [导出] 对话框中，键入文件名，选择文件类型和目标。
4. 单击 [确定]。

   将结果集导出到指定位置的文件中。

使用 OUTPUT 语句导出数据

使用 OUTPUT 语句从数据库中导出查询结果、表或视图。

当需要考虑兼容性时， OUTPUT 语句十分有用，因为该语句能够以多种不同的文件格式写出 SELECT 语句的结果集。您可以指定缺省输出格式，也可以在每个 OUTPUT 语句上指定格式。 Interactive SQL 可以执行包含多个 OUTPUT 语句的命令文件。

当处于以下情况时，请考虑使用 Interactive SQL OUTPUT 语句：

- 要以非文本格式导出表或视图的全部或一部分。
- 要使用命令文件自动执行导出过程。

对数据库的影响

如果可以在使用 OUTPUT 语句、UNLOAD 语句和 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

使用 OUTPUT 语句导出大量的数据会对性能产生影响。如果可能，应该在服务器所在的计算机上使用 OUTPUT 语句，从而避免通过网络发送大量的数据。

有关详细信息，请参见 『ASA SQL 参考』> 「OUTPUT 语句 [Interactive SQL]」。

示例

下面的示例将数据从示例数据库中的 employee 表导出到名为 *employee.txt* 的 .TXT 文件中。

```
SELECT *
 FROM employee;
 OUTPUT TO employee.txt
 FORMAT ASCII
```

使用 UNLOAD TABLE 语句

UNLOAD TABLE 语句只允许以有效的方式导出数据，导出数据的格式为文本 /ASCII/FIXED。使用 UNLOAD TABLE 语句导出时，数据库表中的每行在文件中占一行，各个值之间由逗号分隔符隔开。按照主键值的顺序导出数据，以便使重装速度更快。

您可以在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中执行 UNLOAD TABLE 语句。

当处于以下情况时，请考虑使用 UNLOAD TABLE 语句：

- 要以文本格式导出整个表。
- 注重数据库性能。

对数据库的影响

当卸载表时， UNLOAD TABLE 语句在整个表上放置一个排它锁。

如果可以在使用 OUTPUT 语句、UNLOAD 语句和 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「UNLOAD TABLE 语句」。

示例

使用 ASA 9.0 示例数据库，可以通过执行下面的命令将 employee 表卸载到一个名为 *emp_test.txt* 的文本文件中：

```
UNLOAD TABLE employee TO 'c:\emp_test.txt'
```

使用 UNLOAD 语句

UNLOAD 语句类似于 OUTPUT 语句，因为两个语句都将查询结果导出到文件中。但是， UNLOAD 语句使您可以通过一种更有效的方式导出数据，导出数据的格式仅为文本 /ASCII/FIXED。使用 UNLOAD 语句导出时，数据库表中的每行在文件中占一行，各个值之间由逗号分隔符隔开。

当处于以下情况时，请考虑使用 UNLOAD 语句卸载数据：

- 要在需要考虑性能时导出查询结果。
- 要以文本格式输出。
- 要将导出命令嵌入到应用程序中。

对数据库的影响

如果可以在使用 OUTPUT 语句、UNLOAD 语句和 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

若要使用 UNLOAD 语句，用户必须在表上具有 ALTER 或 SELECT 权限。有关控制哪些用户可以使用 UNLOAD 语句的详细信息，请参见 『ASA 数据库管理指南』> 「-gl 服务器选项」。

UNLOAD 语句在当前隔离级别上执行。

当卸载并重装有代理表的数据库时，您必须创建一个外部登录来把本地用户映射到远程用户，即使该名用户在本地和远程数据库中的口令相同也是如此。如果您没有远程登录，那么重装有可能因为您无法连接到远程服务器而失败。

有关详细信息，请参见 『ASA SQL 参考』> 「UNLOAD 语句」。

使用 dbunload 实用程序导出数据

dbunload 实用程序允许您导出数据库中的一个、多个或所有表。可以导出表数据和表模式。如果要在数据库中重新排列表，您还可以使用 *dbunload* 实用程序创建必要的命令文件，然后根据需要对它们进行修改。卸载表时，可以只卸载结构、只卸载数据或既卸载结构又卸载数据。

您还可以抽取具有或不具有命令文件的一个或多个表。这些文件可用于在不同的数据库中创建相同的表。

注意
尽管 *dbunload* 实用程序在图形界面方面与 Sybase Central 有所不同，但在功能上是相同的。您可以使用其中的任何一个产生相同的结果。

当处于以下情况时，请考虑使用 *dbunload* 实用程序：

- 要以文本格式导出数据。
- 需要快速处理大量数据。
- 文件格式要求比较灵活。
- 需要重建或抽取数据库。

对数据库的影响

在尝试卸载数据库中的一部分表前建立连接。

有关 *dbunload* 实用程序的详细信息，请参见『ASA 数据库管理指南』> 「使用 dbunload 命令行实用程序卸载数据库」。

卸载工具

可以使用下列工具帮助您从数据库中卸载数据：

- 「使用 dbunload 实用程序导出数据」第 612 页
- 「使用 [卸载数据库] 向导」第 613 页

- 「使用 [卸载数据] 对话框」第 613 页

使用 [卸载数据库] 向导

您可以使用 [卸载数据库] 向导将现有数据库卸载到新数据库中。

您可以使用 Sybase Central [卸载数据库] 向导，以逗号分隔的 ASCII 格式卸载整个数据库并创建必要的 Interactive SQL 命令文件以完全重新创建数据库。对于创建 SQL Remote 抽取或建立具有相同或稍加修改结构的数据库新副本来说，这可能非常有用。 [卸载数据库] 向导可用于导出准备重新用在 Adaptive Server Anywhere 中的 Adaptive Server Anywhere 文件。

注意
Sybase Central [卸载数据库] 向导与 dbunload 实用程序在图形界面方面有所不同，但在功能上是等同的。您可以使用其中的任何一个产生相同的结果。

使用 [卸载数据] 对话框

可以使用 [卸载数据] 对话框卸载数据库中的一个或多个表。使用 [卸载数据库] 向导或卸载实用程序 (dbunload) 也可获得此功能，但是此对话框允许您一步卸载表而不用完成整个 [卸载数据库] 向导。

❖ 使用 [卸载数据] 对话框卸载表

1. 在 Sybase Central 中连接数据库。
2. 在 Sybase Central 的左窗格中，展开数据库，然后展开 [表] 文件夹。
3. 在 Sybase Central 的右窗格中，选择希望从其中导出数据的表。

   可以通过按住 SHIFT 键并单击表来选择多个表。
4. 选择 [文件] → [卸载数据]。

   出现 [卸载数据] 对话框。

5 选择所需选项，然后单击 [确定] 卸载数据。

导出查询结果

您可以在 Interactive SQL 中使用 [数据] 菜单或 OUTPUT 语句将查询（包括视图上的查询）导出到文件中。

您可以使用 BCP FORMAT 子句在 Adaptive Server Anywhere 和 Adaptive Server Enterprise 间导入和导出文件。

有关详细信息，请参见 「Adaptive Server Enterprise 兼容性」第 640 页。

❖ 导出查询结果 （Interactive SQL [数据] 菜单）：

1 在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中输入您的查询。

2 单击 [执行 SQL 语句] 显示结果集。

3 选择 [数据] → [导出]。

即会出现 [另存为] 对话框。

4 为导出的数据指定名称和位置。

5 指定文件格式，然后单击 [保存]。

如果导出成功， [消息] 窗格将显示导出查询结果集花费的时间、导出数据的文件名和路径以及写入的行数。

如果导出不成功，则将显示一条消息指示导出不成功。

❖ 导出查询结果 （Interactive SQL OUTPUT 语句）：

1 在 [Interactive SQL] 窗口的 [SQL 语句] 窗格中输入您的查询。

2 在查询的结尾处，键入 **OUTPUT TO** *'c:\ 文件名 '*。

例如，若要将整个 employee 表导出到文件 *employee.dbf*，请输入以下查询：

```
SELECT *
 FROM employee;
 OUTPUT TO 'c:\employee.dbf'
```

3　执行下列操作之一：

要实现这一目的 …	请使用此语句 …	示例 …
导出查询结果并将结果附加到另一个文件中 …	APPEND	`SELECT * FROM employee; OUTPUT TO 'c:\employee.dbf' APPEND`
导出查询结果并包括消息 …	VERBOSE	`SELECT * FROM employee; OUTPUT TO 'c:\employee.dbf' VERBOSE`
指定 ASCII（缺省文件格式）以外的文件格式 …	FORMAT	`SELECT * FROM employee; OUTPUT TO 'c:\employee.dbf' FORMAT dbaseiii;` 其中， *c:\employee.dbf* 是新文件的路径、名称和扩展名， *dbaseiii* 是此文件的文件格式。您可以将字符串括在单引号或双引号中，但只有当路径包含嵌入的空格时才需要使用单引号或双引号。 如果省略 FORMAT 选项，则文件类型缺省为 ASCII。

4　执行该语句。

如果导出成功， [消息] 窗格将显示导出查询结果集花费的时间、导出数据的文件名和路径以及写入的行数。如果导出不成功，则将显示一条消息指示导出不成功。

有关使用 OUTPUT 语句导出查询结果的详细信息，请参见『ASA SQL 参考』> 「OUTPUT 语句 [Interactive SQL]」。

提示
您可以结合使用 APPEND 和 VERBOSE 语句将结果和消息附加到现有的文件。例如，键入 `OUTPUT TO '文件名.sql' APPEND VERBOSE`。有关 APPEND 和 VERBOSE 的详细信息，请参见 『ASA SQL 参考』> 「OUTPUT 语句 [Interactive SQL]」。

OUTPUT TO、 APPEND 和 VERBOSE 语句相当于以前版本的 Interactive SQL 中的 >#、 >>#、 >& 和 >>& 运算符。您仍可以使用这些运算符重定向数据，但是新的 Interactive SQL 语句提供更精确的输出和更便于阅读的代码。

❖ 导出查询结果 （UNLOAD 语句）：

1 在 Interactive SQL 窗口的 [SQL 语句] 窗格中，输入 UNLOAD 语句。例如，

```
UNLOAD
 SELECT *
 FROM employee;
 TO 'c:\employee.dbf'
```

2 执行 UNLOAD 语句。

如果导出成功， [消息] 窗格将显示导出查询结果集花费的时间、导出数据的文件名和路径以及写入的行数。

如果导出不成功，则将显示一条消息指示导出不成功。

导出数据库

❖ 卸载整个或部分数据库 (Sybase Central)：

1 在左窗格中，选择 Adaptive Server Anywhere 插件。

2 在右窗格中，单击 [实用程序] 选项卡。

您可以对数据库执行的所有操作都会显示在右窗格中。

3　在右窗格中双击 [卸载数据库]。

即会出现 [卸载数据库] 向导。

您还可以通过以下两种方法打开 [卸载数据库] 向导：在左窗格中右击数据库，然后从弹出式菜单中选择 [卸载数据库]；或者选择 [工具] → [Adaptive Server Anywhere 9] → [卸载数据库] 命令。

4　请按照向导中的说明进行操作。

❖ **卸载整个或部分数据库 （命令行）：**

1　在命令提示符处，输入 *dbunload* 命令并使用 -c 选项指定连接参数。

例如，以下命令将整个数据库卸载到 c:\temp：

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" c:\temp
```

2　执行下列操作之一：

- 若要只导出数据，请使用 -d。例如：

  ```
  dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d
  c:\temp
  ```

- 若要只导出模式，请使用 -n。例如：

  ```
  dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -n
  c:\temp
  ```

有关详细信息，请参见 『ASA 数据库管理指南』> 「使用 dbunload 命令行实用程序卸载数据库」。

输出空值

若要抽取数据以用于其它软件产品中。由于其它软件包可能无法识别空值，因此提供两种方法指定如何输出空值：

- Interactive SQL NULLS 选项 - 允许您设置缺省行为，或更改特定会话的输出值

- IFNULL 函数 - 允许您将输出值应用到特定实例或查询

两种方法都允许您输出代替空值的特定值。指定如何输出空值提供了与其它软件包更高的兼容性。

❖ 指定空值输出 (Interactive SQL)：

1. 在 [Interactive SQL] 窗口中，选择 [工具] → [选项] 以显示 [选项] 对话框。
2. 单击 [结果] 选项卡。
3. 在 [将空值显示为] 字段中，键入要用来替换空值的值。
4. 执行下列操作之一：
   - 单击 [设为永久] - 使更改成为缺省设置
   - 单击 [确定] - 使更改仅对当前会话有效

有关设置 Interactive SQL 选项的详细信息，请参见 『ASA SQL 参考』 > 「SET OPTION 语句」。

导出表

除了在下面几节中介绍的方法之外，您还可以通过选择表中的所有数据并导出查询结果来导出表。

有关详细信息，请参见 「导出查询结果」第 614 页。

提示
您可以像导出表那样导出视图。

❖ 导出表 （命令行）：

- 在命令提示符处，输入以下 *dbunload* 命令，然后按 ENTER 键：

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t
employee c:\temp
```

其中， -c 指定数据库连接参数， -t 指定您要导出的表的名称。此 *dbunload* 命令将数据从示例数据库 （假设运行在缺省数据库服务器上并使用缺省的数据库名）卸载到 *c:\temp* 目录中的一组文件中。在当前目录中以缺省名称 *reload.sql* 创建命令文件，该命令文件将从数据文件重建数据库。

通过用逗号 (,) 分隔符分隔表名，您可以卸载多个表。

❖ 导出表 (SQL)：

- 执行 UNLOAD TABLE 语句。例如，

```
UNLOAD TABLE department
 TO 'dept.txt'
```

此语句将示例数据库中的 department 表卸载到服务器的当前工作目录中的文件 *dept.txt* 中。如果您正在对网络服务器运行，则该命令将数据卸载到服务器计算机 （而不是客户计算机）上的某个文件中。另外，将文件名作为一个字符串传递给服务器。如果文件名的目录以字母 n （\n 是换行符）或任何其它特殊字符开头，则在文件名中使用转义反斜线字符可以防止产生误解。

表的每一行都输出到输出文件的单个行上，不导出任何列名。各列由逗号分开 （即隔开）。可以使用 DELIMITED BY 子句更改分隔符。字段不是固定宽度的字段。只导出每个条目中的字符，而不是列的整个宽度。

有关 UNLOAD TABLE 语句的语法的详细信息，请参见 『ASA SQL 参考』> 「UNLOAD TABLE 语句」。

重建数据库

重建数据库是一个特定类型的导入和导出，涉及卸载和重装整个数据库。重建 （装载 / 卸载）和抽取过程用于重建数据库以及从部分旧数据库创建新数据库。重建数据库以一致的方法从数据库中取出所有信息然后再将它们放回，这样可以填充空间并提高性能，这在很大程度上类似于磁盘驱动器碎片整理。

如果要提高数据库性能、回收碎片化的磁盘空间或将数据库升级到更新版本的 Adaptive Server Anywhere，请考虑重建数据库。

您可以从 Sybase Central 或使用 *dbunload* 命令行实用程序执行重建数据库的操作。

注意
最好在重建前对数据库进行备份。有关信息，请参见 『ASA 数据库管理指南』> 「备份和恢复简介」。

从一个 Adaptive Server Anywhere 数据库到另一个

重建通常从 Adaptive Server Anywhere 数据库取出数据，然后将这些数据放回 Adaptive Server Anywhere 数据库。卸载和重装紧密地联系在一起，这是因为您通常都执行两个任务，而不是只执行其中的一个。

重建与导出

重建不同于导出，其不同之处在于除了导出并导入数据之外，重建还导出并导入表定义和模式。重建过程的卸载部分产生 ASCII 格式数据文件和一个包含表和其它定义的 *reload.sql* 文件。运行 *reload.sql* 脚本将重新创建表并将数据装载到这些表中。

如果正在使用 SQL Remote 或 MobiLink，则考虑抽取数据库 （从旧数据库创建新数据库）。有关抽取数据库的详细信息，请参见 「抽取数据」第 630 页。

重建复制数据库

重建数据库的过程取决于该数据库是否参与复制。如果数据库参与复制，则必须在整个操作过程中保留事务日志偏移，因为 Message Agent 和 Replication Agent 需要此信息。如果数据库不参与复制，则重建过程会简单一些。

重建数据库的原因

有几个原因需要考虑重建数据库。如果您想要实现以下目标，则可能要重建数据库：

- **升级数据库文件格式** 通过应用升级实用程序可以实现某些新功能，但是其它功能需要数据库文件格式升级，这是通过卸载和重装数据库实现的。[新功能] 文档说明若要获得特定的功能是否需要进行卸载和重装。

  可以使用新版本的 Adaptive Server Anywhere 数据库服务器而无需升级您的数据库。如果要使用新版本的某些功能（这些功能需要访问新系统表或数据库选项），则必须使用升级实用程序升级您的数据库。升级实用程序不卸载或重装任何数据。

  如果要使用依赖数据库文件格式更改的新版本功能，则必须卸载并重装您的数据库。您应该在重建数据库后对数据库进行备份。有关升级数据库的详细信息，请参见 『SQL Anywhere Studio 的新功能』> 「升级 Adaptive Server Anywhere」。

- **回收磁盘空间** 如果您删除数据，数据库不会收缩。而只是将所有空页标记为可用，以便可以重新使用这些页。除非重建数据库，否则不会从数据库中删除它们。如果您从数据库中删除了大量的数据并且预期不会添加更多的数据，则重建数据库可以回收磁盘空间。

- **提高数据库性能** 重建数据库可以提高性能，原因如下：

  - 如果数据库中页上的数据已经碎片化，则卸载和重装可以消除碎片现象。

注意

如果需要对数据库进行碎片整理，并且由于要连续访问数据库而不能进行完全重建，请考虑重组表而不是重建。

有关重组表的详细信息，请参见 『ASA SQL 参考』> 「REORGANIZE TABLE 语句」。

  - 由于可以按照主键的顺序卸载和重装数据，因此相关的行可能出现在相同或相邻的页上，从而使对相关信息的访问速度更快。

重建注意事项

重建 （装载和卸载）和抽取过程用于重建数据库以及从部分旧数据库创建新数据库。重建数据库前，应考虑如何进行该过程。

考虑此因素 …	有关信息，请参见 …
重建数据库涉及哪些内容	「重建数据库」第 620 页 「重建数据库的原因」第 621 页
可用于重建的工具	「重建和重载工具」第 626 页
重建对数据库性能造成的影响	「重建数据库的原因」第 621 页
重建对数据库用户造成的影响	「重建过程中尽量缩短停机时间」第 629 页
数据库是否参与复制	「重建参与复制的数据库」第 624 页 「重建未参与复制的数据库」第 622 页
数据库是否包含代理表	「重建含有代理表的数据库」第 626 页

重建未参与复制的数据库

只有当数据库未参与复制时，才可以使用以下过程。

有关从 Sybase Central 重建未参与复制的数据库的说明，请参见 『SQL Anywhere Studio 的新功能』> 「升级数据库文件格式」。

❖ **重建未参与复制的数据库 （命令行）：**

1. 在命令提示符处，使用以下选项之一执行 *dbunload* 命令行实用程序：

要实现这一目的 …	请使用此选项 …	示例
重建到新数据库 …	**-an**	`dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -an asacopy.db`
重装到一个现有的数据库 …	**-ac**	`dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -ac "uid=DBA;pwd=SQL;dbf=newdemo.db"`
替换现有数据库 …	**-ar**	`dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -ar "uid=DBA;pwd=SQL;dbf=newdemo.db"`

如果使用这些选项中的一个，则不在磁盘上创建数据的中间副本，因此不必在命令行上指定卸载目录。这为数据提供了更高的安全性。另外， `-ar` 和 `-an` 选项的执行速度应该比 Sybase Central 快，但 `-ac` 慢一些。

2　先关闭数据库并将事务日志存档，然后再使用重装的数据库。

注意

`-an` 和 `-ar` 选项只应用于到个人服务器的连接或通过共享内存到网络服务器的连接。

还有一些选项可用于 *dbunload* 实用程序以优化卸载，另外，一些连接参数选项可用于指定运行或非运行数据库参数。

重建参与复制的数据库

如果某个数据库参与复制，则重建该数据库需要特别注意。复制是基于事务日志中的偏移进行的。当重建数据库时，旧事务日志中的偏移不同于新事务日志中的偏移，从而使旧日志不可用。因此，在数据库参与复制时，采取良好的备份措施非常重要。

重建参与复制的数据库有两种方法。第一种方法使用 *dbunload* 实用程序 `-ar` 选项，以一种不影响复制的方式执行卸载和重装。第二种方法是一种完成相同任务的手工方法。

当进行导入和导出时，数据的目标无外于进数据库和出数据库。导入将数据读入数据库。导出将数据写出数据库。通常，信息来自或进入另一个非 Adaptive Server Anywhere 数据库。

但是，重建结合两项功能：装载和卸载。装载和卸载从 Adaptive Anywhere 数据库中取出数据和模式，然后将数据和模式放回 Adaptive Server Anywhere 数据库。卸载过程产生固定格式的数据文件和一个 *reload.sql* 文件，后者包含准确地重新创建表所需的表定义。运行 *reload.sql* 脚本重新创建表并将数据装载回这些表。

重建数据库可能是一项耗时的操作，并可能需要大量磁盘空间。同样，当卸载和重装数据库时，数据库无法使用。出于这些原因，除非您具有明确的目标，否则不建议在生产环境中重建数据库。

❖ 重建参与复制的数据库 （dbunload 实用程序）：

1. 关闭数据库。
2. 执行完全脱机备份：将数据库和事务日志文件复制到安全位置。
3. 在命令提示符处，运行 *dbunload* 以重建数据库：

   ```
   dbunload -c connection_string -ar directory
   ```

   其中， *connection_string* 是具有 DBA 权限的连接， *directory* 是复制环境中用于旧事务日志的目录，数据库没有其它连接。

   `-ar` 选项只应用于到个人服务器的连接或通过共享内存到网络服务器的连接。

有关详细信息，请参见 『ASA 数据库管理指南』> 「卸载实用程序选项」。

4 关闭新数据库。执行通常在恢复数据库之后执行的有效性检查。

5 使用任何需要的生产选项启动数据库。现在您可以允许用户访问重装的数据库。

注意

还有一些选项可用于 *dbunload* 实用程序以优化卸载，另外，一些连接参数选项可用于指定运行或非运行数据库参数。

如果上面的过程不能满足您的需要，您可以手工调整事务日志偏移。以下过程介绍如何执行该操作。

❖ 通过手工干预重建参与复制的数据库：

1 关闭数据库。

2 执行完全脱机备份：将数据库和事务日志文件复制到安全位置。

3 运行 *dbtran* 实用程序显示数据库当前事务日志文件的起始偏移和结束偏移。记录结束偏移以供稍后使用。

4 重命名当前事务日志文件，以便在卸载过程中该文件不被修改，然后将此文件放置在 dbremote 脱机日志目录中。

5 重建数据库。

有关此步骤的信息，请参见 「重建数据库」第 620 页。

6 关闭新数据库。

7 消除新数据库的当前事务日志文件。

8 在新数据库上使用 *dblog*，将第 3 步中记录的结束偏移用作 `-z` 参数，同时将相关偏移设置为零。

```
dblog -x 0 -z 137829 database-name.db
```

9 在运行消息代理时，在它的命令行上为它提供原始脱机目录的位置。

10 启动数据库。现在您可以允许用户访问重装的数据库。

重建含有代理表的数据库

当卸载并重装有代理表的数据库时，您必须创建一个外部登录来把本地用户映射到远程用户，即使该名用户在本地和远程数据库中的口令相同也是如此。如果您没有远程登录，那么重装有可能因为您无法连接到远程服务器而失败。

有关外部登录的详细信息，请参见 「使用外部登录」第 653 页。

重建和重载工具

有多种工具可帮助您重建数据库。还有一些工具可帮助您将数据重装入现有数据库中。

使用 dbunload 和 dbisql 实用程序重建数据库

dbunload 和 *dbisql* 实用程序允许您以逗号分隔的 ASCII 格式卸载整个数据库并创建必要的 Interactive SQL 命令文件以完全重新创建数据库。对于创建 SQL Remote 抽取或建立具有相同或稍加修改结构的数据库新副本来说，这可能非常有用。这些实用程序可用于导出准备重新用在 Adaptive Server Anywhere 中的 Adaptive Server Anywhere 文件。

注意
dbunload/dbisql 实用程序和 Sybase Central [卸载数据库] 向导在图形界面方面有所不同，但在功能上是等同的。您可以使用其中的任何一个产生相同的结果。

当处于以下情况时，请考虑使用 *dbunload* 或 *dbisql* 实用程序：

- 要重建数据库或从数据库抽取数据。
- 要以文本格式导出。
- 需要快速处理大量数据。
- 文件格式要求比较灵活。

有关详细信息，请参见 「重建未参与复制的数据库」第 622 页、「重建参与复制的数据库」第 624 页和 『ASA 数据库管理指南』> 「使用 [卸载数据库] 向导卸载数据库」。

使用 UNLOAD TABLE 语句重建数据库

UNLOAD TABLE 语句只允许以有效的方式导出数据，导出数据的格式为文本 /ASCII/FIXED。使用 UNLOAD TABLE 语句导出时，数据库表中的每行在文件中占一行，各个值之间由逗号分隔符隔开。为了使重装速度更快，按照主键值的顺序导出数据。

当处于以下情况时，请考虑使用 UNLOAD TABLE 语句重建数据库：

- 需要考虑性能。
- 要以文本格式导出数据。

对数据库的影响

UNLOAD TABLE 语句在整个表上放置一个排它锁。

若要使用 UNLOAD TABLE 语句，用户必须在表上具有 ALTER 或 SELECT 权限。

有关详细信息，请参见 『ASA SQL 参考』> 「UNLOAD TABLE 语句」。

导出表数据或表模式

❖ **导出表数据或表模式 （命令行）：**

1. 在命令提示符处，输入 *dbunload* 命令并使用 -c 选项指定连接参数。
2. 使用 -t 选项指定要导出数据或模式的表。

   例如，若要导出 employee 表的一部分，请输入

   ```
   dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t
   employee c:\temp
   ```

   通过用逗号分隔符分隔表名，您可以卸载多个表。

3 执行下列操作之一：

要实现这一目的 …	请添加此选项 …	示例 …
仅导出数据 …	**-d**	`dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d -t employee c:\temp`
仅导出模式 …	**-n**	`dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -n -t employee c:\temp`

4 按 *Enter* 键执行该命令。

这些示例中的 *dbunload* 命令将数据或模式从示例数据库表 （假设运行在缺省数据库服务器上并具有缺省数据库名）卸载到 *c:\temp* 目录中的一个文件中。在当前目录中以缺省名称 *reload.sql* 创建命令文件，该命令文件将从数据文件重建数据库。

重装数据库

重装包括创建一个空数据库文件，并使用 **reload.sql** 文件创建模式，然后将从另一个 Adaptive Server Anywhere 数据库卸载的所有数据插入到新创建的表中。从命令行重装数据库。

❖ 重装数据库 （命令行）：

1 打开命令提示符。

2 执行 *reload.sql* 脚本。

例如，以下命令在当前目录中装载 *reload.sql* 脚本。

```
dbisql -c "dbn=asademo;uid=DBA;pwd=SQL" reload.sql
```

重建过程中尽量缩短停机时间

以下步骤帮助您在重建数据库时尽量缩短停机时间。如果您的数据库全天运转，则这些步骤可能特别有用。

最好在开始实际备份之前，练习运行第 1 步到第 4 步以确定每步所需的时间。您可能还要在重建过程中的不同时间点保存文件的副本。

确保没有其它调度的备份重命名生产数据库的日志。如果出现错误，则您将需要按照正确的顺序将这些重命名日志中的事务应用到重建的数据库。

❖ 重建过程中尽量缩短停机时间：

1. 使用 DBBACKUP -r，创建数据库和日志的备份，然后重命名日志。
2. 在另一台计算机上重建备份的数据库。
3. 在生产服务器上再执行一次 DBBACKUP -r 以重命名日志。
4. 在第 3 步产生的日志上运行 **DBTRAN**，然后将这些事务应用到重建的服务器。

   现在的重建数据库包含直到第 3 步备份结束后的所有事务。
5. 关闭生产服务器并制作数据库和日志的副本。
6. 在生产服务器上复制重建的数据库。
7. 在第 5 步产生的日志上运行 **DBTRAN**。

   这应该是一个相对较小的文件。
8. 在重建的数据库上启动服务器，但不允许用户进行连接。
9. 应用执行第 8 步后产生的事务。
10. 允许用户进行连接。

抽取数据

抽取操作从统一 Adaptive Server Enterprise 或 Adaptive Server Anywhere 数据库中创建远程 Adaptive Server Anywhere 数据库。

您可以使用 Sybase Central [抽取数据库] 向导或抽取实用程序来抽取数据库。抽取实用程序是从统一数据库创建和同步远程数据库时推荐使用的方法。

有关如何执行数据库抽取的更多信息，请参见：

- 『SQL Remote 用户指南』> 「数据库抽取实用程序」
- 『SQL Remote 用户指南』> 「使用抽取实用程序」
- 『SQL Remote 用户指南』> 「抽取实用程序选项」
- 『SQL Remote 用户指南』> 「抽取组」
- 『MobiLink 客户端』> 「部署远程数据库」
- 『SQL Remote 用户指南』> 「在 Sybase Central 中抽取远程数据库」

将数据库迁移到 Adaptive Server Anywhere

您可以使用 sa_migrate 存储过程集或 [数据迁移] 向导将表从远程 Oracle、 DB2、 Microsoft SQL Server、 Sybase Adaptive Server Enterprise、Adaptive Server Anywhere 和 Microsoft Access 数据库导入到 Adaptive Server Anywhere 中。

使用 [数据迁移] 向导

您必须先创建**目标数据库**，然后才能使用 [数据迁移] 向导或存储过程的 sa_migrate 集来迁移数据。目标数据库是数据要迁移到的数据库。

有关创建数据库的信息，请参见 「创建数据库」第 33 页。

使用 [数据迁移] 向导，您可以创建远程服务器来连接远程数据库，还可以创建外部登录 （如果需要），以便让当前用户连接到远程数据库。

❖ **导入远程表 (Sybase Central)：**

1. 在 Sybase Central 中，连接到目标数据库。
2. 在左窗格中，选择 Adaptive Server Anywhere 插件。
3. 在右窗格中，单击 [实用程序] 选项卡。

   您可以对数据库执行的所有操作都会显示在右窗格中。
4. 双击 [迁移数据库]。

   即会出现 [迁移数据库] 向导。

   您还可以通过以下两种方法打开 [迁移数据库] 向导：在左窗格中右击目标数据库，然后从弹出式菜单中选择 [迁移数据库]；或者选择 [工具] → [Adaptive Server Anywhere 9] → [迁移数据库]。
5. 从列表中选择目标数据库，然后单击 [下一步]。

6 选择要用于连接到远程数据库 （要从该数据库迁移数据）的远程服务器，然后单击 [下一步]。

如果尚未创建远程服务器，则单击 [立即创建远程服务器] 以打开 [远程服务器创建] 向导。

有关创建远程服务器的详细信息，请参见 「创建远程服务器」第 646 页。

您还可以为远程服务器创建外部登录。缺省情况下， Adaptive Server Anywhere 在代表当前用户连接到远程服务器时使用该用户的用户 ID 和口令。但是，如果远程服务器没有定义与当前用户具有相同用户 ID 和口令的用户，则您必须创建一个外部登录。该外部登录为当前用户指派一个替代登录名和口令，以使该用户能够连接到远程服务器。

7 选择要迁移的表，然后单击 [下一步]。

您无法迁移系统表，所以此列表中不会出现任何系统表。

8 选择目标数据库上将拥有表的用户，然后单击 [下一步]。

如果尚未创建用户，则单击 [立即创建用户] 以打开 [用户创建] 向导。

9 选择是否要从远程表迁移数据和 / 或外键以及是否要保留为迁移进程创建的代理表，然后单击 [下一步]。

如果目标数据库是版本 8.0.0 或更早的版本，则不支持 [迁移外键] 选项。您必须将数据库升级为版本 8.0.1 或更高版本才能使用此选项。

有关升级的详细信息，请参见 『SQL Anywhere Studio 的新功能』> 「升级数据库」。

10 单击 [完成]。

使用 sa_migrate 存储过程

使用 sa_migrate 存储过程，您可以迁移远程数据。如果不想对表进行任何修改，则可以使用单步方法。或者，如果要删除表或外键映射，则可以使用扩展方法。

如果使用 sa_migrate 存储过程集，则必须完成以下步骤才能导入远程数据库：

- 创建目标数据库。

  有关创建数据库的信息，请参见 「创建数据库」第 33 页。

- 创建一个远程服务器以连接到远程数据库。

  有关创建远程服务器的信息，请参见 「创建远程服务器」第 646 页。

- 创建外部登录以连接到远程数据库。只有当用户在目标和远程数据库上具有不同的口令，或者当您在远程数据库上登录使用的用户 ID 与在目标数据库上登录使用的用户 ID 不同时，才需要外部登录。

  有关创建外部登录的信息，请参见 「创建外部登录」第 653 页。

- 创建本地用户，该用户将拥有目标数据库中迁移的表。

  有关创建用户的信息，请参见 『ASA 数据库管理指南』> 「创建新用户」。

下面的示例使用 sa_migrate 存储过程，单步导入属于远程数据库上的一名所有者的所有表。

使用 sa_migrate 完成数据库的一步迁移

为 *table_name* 和 *owner_name* 参数提供 NULL 值将迁移数据库中的所有表，包括系统表。远程数据库中属于不同用户的同名的表在目标数据库中都属于同一个所有者。出于这些原因，建议您一次性地迁移与一个所有者关联的表。

❖ **导入远程表 （单步方法）：**

1. 在 Interactive SQL 中，连接到目标数据库。

2 在 Interactive SQL 的 [语句] 窗格中，运行 sa_migrate 存储过程。例如，

```
CALL sa_migrate( 'local_a', 'ase', NULL, l_smith,
NULL, 1, 1, 1 )
```

此过程依次调用多个过程，并使用指定的条件迁移所有属于用户 l_smith 的远程表。

如果不想让目标数据库上的同一个用户拥有所有迁移的表，则必须为目标数据库上的每名所有者运行 sa_migrate 过程，同时指定 *local_table_owner* 和 *owner_name* 参数。建议您一次性地迁移与一个所有者关联的表。

有关详细信息，请参见 『ASA SQL 参考』> 「sa_migrate 系统过程」。

对于版本 8.0.0 或更早版本的目标数据库，将自动迁移外键。如果不想迁移外键，则您必须将数据库文件格式升级为版本 8.0.1 或更高版本。

有关升级的详细信息，请参见 『SQL Anywhere Studio 的新功能』> 「升级数据库」。

使用 sa_migrate 存储过程迁移各个表

❖ **导入远程表 （具有修改）：**

1 在 Interactive SQL 中，连接到目标数据库。

2 运行 sa_migrate_create_remote_table_list 存储过程。例如，

```
CALL sa_migrate_create_remote_table_list( 'ase',
      NULL, 'remote_a', 'mydb' )
```

您必须指定 Adaptive Server Enterprise 和 Microsoft SQL Server 数据库的数据库名。

这将用要迁移的远程表列表填充 dbo.migrate_remote_table_list 表。对于不想迁移的远程表，您可以从此表中删除相应的行。

不要为 *table_name* 和 *owner_name* 参数提供 NULL 值。这将迁移数据库中的所有表，包括系统表。远程数据库中属于不同用户的同名的表在目标数据库中都属于同一个所有者。建议您一次性地迁移与一个所有者关联的表。

有关 sa_migrate_create_remote_table_list 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_migrate_create_remote_table_list 系统过程」。

3 运行 sa_migrate_create_tables 存储过程。例如，

```
CALL sa_migrate_create_tables( 'local_a', )
```

此过程采用 dbo.migrate_remote_table_list 中的远程表列表，并为列出的每个远程表创建代理表和基表。此过程还为迁移的表创建所有主键索引。

有关 sa_migrate_create_tables 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_migrate_create_tables 系统过程」。

4 如果要将数据从远程表迁移到目标数据库上的基表中，请运行 sa_migrate_data 存储过程。例如，

输入以下存储过程：

```
CALL sa_migrate_data( 'local_a' )
```

此过程将数据从每个远程表迁移到由 sa_migrate_create_tables 过程创建的基表中。

有关 sa_migrate_data 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_migrate_data 系统过程」。

如果不想从远程数据库中迁移外键，则可以跳至第 7 步。

5 运行 sa_migrate_create_remote_fks_list 存储过程。例如，

```
CALL sa_migrate_create_remote_fks_list( 'ase' )
```

此过程使用与 dbo.migrate_remote_table_list 中列出的每个远程表关联的外键列表填充表 dbo.migrate_remote_fks_list。

您可以删除不想在本地基表上重新创建的任何外键映射。

有关 sa_migrate_create_remote_fks_list 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_migrate_create_remote_fks_list 系统过程」。

6 运行 sa_migrate_create_fks 存储过程。例如，

```
CALL sa_migrate_create_fks( 'local_a' )
```

此过程在基表上创建在 dbo.migrate_remote_fks_list 中定义的外键映射。

有关 sa_migrate_create_fks 存储过程的详细信息，请参见『ASA SQL 参考』> 「sa_migrate_create_fks 系统过程」。

7 如果您要删除为了进行迁移而创建的代理表，请运行 sa_drop_proxy_tables 存储过程。例如，

```
CALL sa_migrate_drop_proxy_tables( 'local_a' )
```

此过程删除为迁移创建的所有代理表，整个迁移过程到此完成。

有关 sa_migrate_drop_proxy_tables 存储过程的详细信息，请参见 『ASA SQL 参考』> 「sa_migrate_drop_proxy_tables 系统过程」。

使用 SQL 命令文件

本节介绍如何处理由一组命令组成的文件。**Command files** 是包含 SQL 语句的文本文件，在重复运行同样的 SQL 语句时十分有用。

创建命令文件

您可以使用任何喜欢的文本编辑器来创建命令文件。可以将注释行与要执行的 SQL 语句包括在一起。通常，命令文件还称为**脚本**。

在 Interactive SQL 中打开 SQL 命令文件

在 Windows 平台上，您可以让 Interactive SQL 作为 *.sql* 文件的缺省编辑器，这样，当您双击该文件时，它就会出现在 Interactive SQL 的 [SQL 语句] 窗格中。

有关让 Interactive SQL 作为 *.sql* 文件的缺省编辑器的详细信息，请参见『SQL Anywhere Studio 帮助』> 「[选项] 对话框：[常规] 选项卡」。

执行 SQL 命令文件

您可以通过以下任意方式执行命令文件：

- 您可以使用 Interactive SQL READ 命令执行命令文件。例如，以下语句执行文件 *temp.sql*：

  ```
  READ temp.sql
  ```

  有关详细信息，请参见 『ASA SQL 参考』> 「READ 语句 [Interactive SQL]」。

- 您可以将命令文件装载到 [SQL 语句] 窗格中并在那里直接执行该命令文件。

  通过选择 [文件] → [打开] 将命令文件装载回 [SQL 语句] 窗格。当提示输入文件名时，请输入 *temp.sql*。

- 您可以提供一个命令文件作为 Interactive SQL 的命令行参数。

运行 SQL 命令文件

运行 SQL 命令文件

可以交互地运行命令文件或以批处理模式运行命令文件。

❖ **直接运行命令文件：**

1 在 Interactive SQL 中，选择 [文件] → [运行脚本]。

2 在 [打开] 对话框中，查找并选择该文件。完成后单击 [打开]。

[运行脚本] 菜单项等同于 READ 语句。例如，在 [SQL 语句] 窗格中，您还可以通过键入以下内容运行命令文件：

```
READ 'c:\filename.sql'
```

其中 *c:\filename.sql* 是该文件的路径、名称和扩展名。只有当路径包含空格时才需要使用单引号 （如上所示）。

❖ **使用 [SQL 语句] 窗格运行命令文件：**

- 键入以下命令：

```
READ 'c:\filename.sql'
```

其中 *c:\filename.sql* 是该文件的路径、名称和扩展名。只有当路径包含空格时才需要使用单引号 （如上所示）。

❖ **以批处理模式运行命令文件：**

1 打开命令提示符。

2 提供命令文件作为 Interactive SQL 的命令行参数。

例如，以下命令对示例数据库运行命令文件 *myscript.sql*。

```
dbisql -c "dsn= ASA 9.0 Sample" myscript.sql
```

装载 SQL 命令文件

当开始新 Interactive SQL 会话时，您可以将命令文件的内容装载到 [SQL 语句] 窗格中，或者您可以直接运行内容。

❖ **将命令从文件装载到 [SQL 语句] 窗格中：**

1 选择 [文件] → [打开]。

2　在 [打开] 对话框中，查找并选择该文件。

3　完成后单击 [打开]。

将数据库输出写入文件

在 Interactive SQL 中，在执行下一个命令之前，每个命令的结果集数据一直保留在 [结果] 窗格中的 [结果] 选项卡上。若要记录您的数据，您可以将每个语句的输出保存到单独的文件中。如果 *statement1* 和 *statement2* 是两个 SELECT 语句，那么您可以分别将其输出到 *file1* 和 *file2* 中，如下所示：

statement1; OUTPUT TO *file1*;
    statement2; OUTPUT TO *file2*;

例如，下面的命令将查询结果保存到一个名为 *Employees.txt* 的文件中：

```
SELECT * FROM EMPLOYEE;
 OUTPUT TO 'C:\\My Documents\\Employees.txt'
```

有关详细信息，请参见 『ASA SQL 参考』> 「OUTPUT 语句 [Interactive SQL]」。

Adaptive Server Enterprise 兼容性

您可以使用 BCP FORMAT 子句在 Adaptive Server Anywhere 和 Adaptive Server Enterprise 间导入和导出文件。只要确保 BCP 输出是分隔的 ASCII 格式即可。如果您正在从 Adaptive Server Anywhere 导出 BLOB 数据以用于 Adaptive Server Enterprise，则将 BCP 格式子句用于 UNLOAD TABLE 语句。

有关 BCP 和 FORMAT 子句的详细信息，请参见 『ASA SQL 参考』> 「LOAD TABLE 语句」或 『ASA SQL 参考』> 「UNLOAD TABLE 语句」。

第 18 章

访问远程数据

关于本章

Adaptive Server Anywhere 可以访问位于不同服务器（Sybase 和非 Sybase）上的数据，就像这些数据存储在本地服务器上一样。

本章介绍如何配置 Adaptive Server Anywhere 来访问远程数据。

简介

通过 Adaptive Server Anywhere 远程数据访问，您可以访问其它数据源中的数据。您可以使用此功能将数据迁移到 Adaptive Server Anywhere 数据库中。您还可以使用此功能在数据库中查询数据，不过，这样的多数据库查询的性能要比所有数据都在一个 Adaptive Server Anywhere 数据库中时低得多。

利用远程数据访问，您可以：

- 通过 Adaptive Server Anywhere 使用 insert-select 将数据从一个位置移动到另一个位置。
- 访问关系数据库 （例如 Sybase、 Oracle 和 DB2）中的数据。
- 访问桌面数据，例如 Excel 电子表格、 MS-Access 数据库、FoxPro 和文本文件。
- 访问支持 ODBC 接口的任何其它数据源。
- 在本地和远程数据之间执行连接，不过，性能要比所有数据都位于一个 Adaptive Server Anywhere 数据库中时低得多。
- 在不同 Adaptive Server Anywhere 数据库的表之间执行连接操作。这样操作时的性能限制与使用其它远程数据源时是相同的。
- 在通常不具有 Adaptive Server Anywhere 功能的数据源上使用 Adaptive Server Anywhere 功能。例如，您可以对存储在 Oracle 中的数据使用 Java 功能，或在电子表格上执行子查询。 Adaptive Server Anywhere 将通过对检索后的数据执行操作来补偿远程数据源不支持的功能。
- 使用直通模式直接访问远程服务器。
- 对其它服务器执行远程过程调用。

Adaptive Server Anywhere 允许访问以下外部数据源：

- Adaptive Server Anywhere
- Adaptive Server Enterprise

- Oracle
- IBM DB2
- Microsoft SQL Server
- 其它 ODBC 数据源

有关平台可用性，请参见 『SQL Anywhere Studio 介绍』> 「Windows 和 NetWare 操作系统」。

从 PowerBuilder DataWindows 访问远程数据

通过在连接时将 DBParm Block 参数设置为 1，您可以使用 PowerBuilder DataWindow 访问远程数据。

- 在设计环境中，您可以通过访问 [数据库配置文件设置] 对话框中的 [事务] 选项卡并将 [Retrieve Blocking Factor]（检索块因子）设置为 1 来设置 Block 参数。
- 在连接字符串中，使用以下短语：

```
DBParm="Block=1"
```

访问远程数据的基本概念

本节介绍访问远程数据需要了解的基本概念。

远程表映射

Adaptive Server Anywhere 将表提供给客户端应用程序的方式就像表中的所有数据都存储在与该应用程序连接的数据库中一样。在内部，当执行涉及远程表的查询时，先确定其存储位置，然后访问远程位置以便能够检索数据。

若要让远程表对客户端显示为本地表，您可以创建映射到远程数据的本地代理表。

❖ **创建代理表：**

1. 定义远程数据所在的服务器。此操作指定服务器的类型和远程服务器的位置。

   有关详细信息，请参见 「使用远程服务器」第 646 页。

2. 如果两个服务器上的登录信息不同，则将本地用户登录信息映射为远程服务器用户登录信息。

   有关详细信息，请参见 「使用外部登录」第 653 页。

3. 创建代理表定义。这将指定本地代理表到远程表的映射。这其中包括远程表所在的服务器、数据库名、所有者名称、表名和远程表的列名。

有关详细信息，请参见 「使用代理表」第 655 页。

管理远程表映射

若要管理远程表映射和远程服务器定义，您可以使用 Sybase Central，或者可以使用像 Interactive SQL 这样的工具直接执行 SQL 语句。

服务器类

为每个远程服务器指派一个**服务器类**。服务器类指定用于与服务器进行交互的访问方法。不同类型的远程服务器要求使用不同的访问方法。服务器类提供 Adaptive Server Anywhere 详细服务器功能信息。 Adaptive Server Anywhere 根据这些功能对它与远程服务器的交互进行调整。

当前有两组服务器类。第一组基于 JDBC ；第二组基于 ODBC。

基于 JDBC 的服务器类包括：

- **asajdbc** ，用于 Adaptive Server Anywhere （版本 6 和更高版本）
- **asejdbc** ，用于 Adaptive Server Enterprise 和 SQL Server （版本 10 和更高版本）

基于 ODBC 的服务器类包括：

- **asaodbc** ，用于 Adaptive Server Anywhere （版本 5.5 和更高版本）
- **aseodbc** ，用于 Adaptive Server Enterprise 和 SQL Server （版本 10 和更高版本）
- **db2odbc** ，用于 IBM DB2
- **mssodbc** ，用于 Microsoft SQL Server
- **oraodbc** ，用于 Oracle 服务器 （版本 8.0 和更高版本）
- **odbc** ，用于所有其它 ODBC 数据源

有关远程服务器类的完整说明，请参见 「用于进行远程数据访问的服务器类」第 677 页。

使用远程服务器

在可以将远程对象映射到本地代理表之前，您必须定义远程对象所在的远程服务器。定义了远程服务器后，将在 SYSSERVERS 表中为该远程服务器添加一个条目。本节介绍如何创建、变更和删除远程服务器定义。

创建远程服务器

使用 CREATE SERVER 语句设置远程服务器定义。您可以直接执行语句，也可以使用 Sybase Central。

对于 ODBC 连接，每个远程服务器都对应于一个 ODBC 数据源。对于某些系统 （包括 Adaptive Server Anywhere），每个数据源都描述一个数据库，因此每个数据库都需要一个单独的远程服务器定义。

您必须具有 RESOURCE 权限才能创建服务器。

在 UNIX 平台上，您还需要参考 ODBC 驱动程序管理器。

有关 CREATE SERVER 语句的完整说明，请参见 『ASA SQL 参考』> 「CREATE SERVER 语句」。

示例 1

以下语句在 SYSSERVERS 表中为名为 ASEserver 的 Adaptive Server Enterprise 服务器创建一个条目：

```
CREATE SERVER ASEserver
 CLASS 'ASEJDBC'
 USING 'rimu:6666'
```

其中：

- **ASEserver** 是远程服务器的名称
- **ASEJDBC** 是一个关键字，指示该服务器是 Adaptive Server Enterprise 并且与服务器的连接基于 JDBC
- **rimu:6666** 是远程服务器所在的计算机名称和 TCP/IP 端口号

示例 2

以下语句在 SYSSERVERS 表中为名为 testasa 的基于 ODBC 的 Adaptive Server Anywhere 服务器创建一个条目：

```
CREATE SERVER testasa
 CLASS 'ASAODBC'
 USING 'test4'
```

其中：

- **testasa**　是远程服务器在此数据库中使用的名称。
- **ASAODBC**　是一个关键字，指示该服务器是 Adaptive Server Anywhere 并且与服务器的连接使用 ODBC。
- **test4**　是 ODBC 数据源名称。

示例 3

在 UNIX 平台中，以下语句在 SYSSERVERS 表中为名为 remasa 的基于 ODBC 的 Adaptive Server Anywhere 服务器创建一个条目：

```
CREATE SERVER remasa
 CLASS 'asaodbc'
 USING 'driver=/opt/sybase/SYBSsa9/lib/
dbodbc9_r.so;dsn=my_asa_dsn'
```

其中：

- **remasa**　是远程服务器在此数据库中使用的名称。
- **ASAODBC**　是一个关键字，指示该服务器是 Adaptive Server Anywhere 并且与服务器的连接使用 ODBC。
- **USING**　是对 ODBC 驱动程序管理器的引用。

示例 4

在 UNIX 平台中，以下语句在 sysservers 表中为名为 remase 的基于 ODBC 的 Adaptive Server Enterprise 服务器创建一个条目：

```
CREATE SERVER remase
 CLASS 'aseodbc'
 USING 'driver=/opt/sybase/SYBSsa9/drivers/lib/
libodbc.so;dsn=my_ase_dsn'
```

其中：

- **remase**　是远程服务器在此数据库中使用的名称
- **ASEODBC**　是一个关键字，指示该服务器是 Adaptive Server Enterprise 并且与服务器的连接使用 ODBC

- **USING** 是对 ODBC 驱动程序管理器的引用。

使用 Sybase Central 创建远程服务器

您可以使用 Sybase Central 中的向导创建远程服务器。有关详细信息，请参见 「创建远程服务器」第 646 页。

❖ 创建远程服务器 (Sybase Central)：

1. 从 Sybase Central 连接到主机数据库。
2. 打开该数据库的 [远程服务器] 文件夹。
3. 从 [文件] 菜单中，选择 [新建] → [远程服务器]。

   [创建远程服务器] 向导即会出现。
4. 在向导的第一页上，输入一个用于代表远程服务器的名称。此名称只是在本地数据库中标识该远程服务器；它不需要与服务器提供的名称对应。单击 [下一步]。
5. 选择适当的服务器类型，然后单击 [下一步]。
6. 选择数据访问方法 （ODBC 或 JDBC），并提供连接信息。单击 [下一步]。

   - 对于 ODBC，请提供数据源名称。
   - 对于 JDBC，以 *计算机名称*: *端口号*的形式提供 URL

   数据访问方法 （JDBC 或 ODBC）是 Adaptive Server Anywhere 用来访问远程数据库的方法。此方法与 Sybase Central 用来连接到您的数据库的方法无关。
7. 指定远程服务器是否是只读的。单击 [下一步]。
8. 为远程服务器创建一个外部登录。

缺省情况下， Adaptive Server Anywhere 在代表当前用户连接到远程服务器时使用该用户的用户 ID 和口令。但是，如果远程服务器没有定义与当前用户具有相同用户 ID 和口令的用户，则您必须创建一个外部登录。该外部登录为当前用户指派一个替代登录名和口令，以使该用户能够连接到远程服务器。

9 或者，您可以在创建远程服务器定义前，选择单击 [测试连接] 以测试是否可以连接到远程服务器。

单击 [完成] 创建远程服务器定义。

删除远程服务器

您可以使用 Sybase Central 或 DROP SERVER 语句从 Adaptive Server Anywhere 系统表中删除远程服务器。此操作若要成功，必须已经删除在该服务器上定义的所有远程表。

您必须具有 DBA 权限才能删除远程服务器。

❖ 删除远程服务器 (Sybase Central)：

1 从 Sybase Central 连接到主机数据库。

2 打开 [远程服务器] 文件夹。

3 右击您要删除的远程服务器，然后从弹出式菜单中选择 [删除]。

❖ 删除远程服务器 (SQL)：

1 从 Interactive SQL 连接到主机数据库。

2 执行 DROP SERVER 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「DROP SERVER 语句」。

示例　以下语句删除名为 testasa 的服务器：

```
DROP SERVER testasa
```

变更远程服务器

您可以使用 Sybase Central 或 ALTER SERVER 语句修改服务器的属性。这些更改直到下一次与远程服务器连接时才生效。

您必须具有 RESOURCE 权限才能变更服务器。

❖ 变更远程服务器的属性 (Sybase Central)：

1. 从 Sybase Central 连接到主机数据库。
2. 打开该数据库的 [远程服务器] 文件夹。
3. 右击远程服务器，然后从弹出式菜单中选择 [属性]。
4. 配置远程服务器的多个属性。

❖ 变更远程服务器的属性 (SQL)：

1. 从 Interactive SQL 连接到主机数据库。
2. 执行 ALTER SERVER 语句。

示例

以下语句将名为 ASEserver 的服务器的服务器类更改为 aseodbc。在此示例中，该服务器的 [数据源名] 是 ASEserver。

```
ALTER SERVER ASEserver
 CLASS 'aseodbc'
```

ALTER SERVER 语句还可用于启用或禁用服务器的已知功能。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER SERVER 语句」。

列出服务器上的远程表

当配置 Adaptive Server Anywhere 时，获取特定服务器上可用远程表的列表将很有帮助。 sp_remote_tables 过程返回服务器上表的列表。

```
sp_remote_tables servername
                 [,tablename]
                 [, owner ]
                 [, database]
```

如果给定了 *tablename*、 *owner* 或 *database*，则表的列表仅限于那些符合条件的表。

例如，要从名为 excel 的远程服务器获取所有可用 Microsoft Excel 工作表的列表：

```
sp_remote_tables  excel
```

或者，要获取名为 asetest 的 ASE 中生产数据库中所有归 'fred' 所有的表的列表：

```
sp_remote_tables  asetest, null, fred, production
```

有关详细信息，请参见 『ASA SQL 参考』> 「sp_remote_tables 系统过程」。

列出远程服务器功能

sp_servercaps 过程显示有关远程服务器功能的信息。 Adaptive Server Anywhere 使用此功能信息确定可以将 SQL 语句的哪些功能部分传递给远程服务器。

直到 Adaptive Server Anywhere 第一次连接到远程服务器后，才填充包含服务器容量的系统表。这些信息来自 SYSCAPABILITY 和 SYSCAPABILITYNAME 系统表。指定的服务器名必须与 CREATE SERVER 语句中使用的服务器名相同。

发布存储过程 sp_servercaps，如下所示：

```
sp_servercaps  servername
```

有关详细信息，请参见 『ASA SQL 参考』> 「sp_servercaps 系统过程」。

使用外部登录

缺省情况下， Adaptive Server Anywhere 每次代表其客户端连接到远程服务器时都会使用那些客户端的名称和口令。但是，您可以通过创建外部登录来替换这一缺省设置。外部登录是与远程服务器通信时使用的替代登录名和口令。

有关详细信息，请参见 『ASA 数据库管理指南』> 「使用集成登录」。

创建外部登录

您可以使用 Sybase Central 或 CREATE EXTERNLOGIN 语句创建外部登录。

只有登录名和 DBA 帐户才能添加或修改外部登录。

❖ 创建外部登录 (Sybase Central)：

1. 从 Sybase Central 连接到主机数据库。
2. 打开该数据库的 [远程服务器] 文件夹并选择远程服务器。
3. 从 [文件] 菜单中，选择 [新建] → [外部登录]。

   [创建外部登录] 向导即会出现。
4. 请按照向导中的说明进行操作。

❖ 创建外部登录 (SQL)：

1. 从 Interactive SQL 连接到主机数据库。
2. 执行 CREATE EXTERNLOGIN 语句。

示例

以下语句允许本地用户 **fred** 通过使用远程登录 **frederick** 和口令 **banana** 获得对服务器 **ASEserver** 的访问权限。

```
CREATE EXTERNLOGIN fred
 TO ASEserver
 REMOTE LOGIN frederick
 IDENTIFIED BY banana
```

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE EXTERNLOGIN 语句」。

删除外部登录

您可以使用 Sybase Central 或 DROP EXTERNLOGIN 语句从 Adaptive Server Anywhere 系统表中删除外部登录。

只有登录名和 DBA 帐户才可以删除外部登录。

❖ 删除外部登录 (Sybase Central)：

1. 从 Sybase Central 连接到主机数据库。
2. 打开 [远程服务器] 文件夹。
3. 在左窗格中，选择远程服务器，然后在右窗格中单击 [外部登录] 选项卡。
4. 右击外部登录，然后从弹出式菜单中选择 [删除]。

❖ 删除外部登录 (SQL)：

1. 从 Interactive SQL 连接到主机数据库。
2. 执行 DROP EXTERNLOGIN 语句。

示例

以下语句删除上面的示例中创建的本地用户 fred 的外部登录：

```
DROP EXTERNLOGIN fred TO ASEserver
```

另请参见

- 『ASA SQL 参考』> 「DROP EXTERNLOGIN 语句」

使用代理表

远程数据的位置透明性是通过创建映射到远程对象的本地**代理表**启用的。使用以下语句之一创建代理表：

- 如果该表已存在于远程存储位置，则使用 CREATE EXISTING TABLE 语句。此语句为远程服务器上现有的表定义代理表。
- 如果该表未存在于远程存储位置，则使用 CREATE TABLE 语句。此语句在远程服务器上创建新表，并且还为该表定义代理表。

指定代理表位置

CREATE TABLE 和 CREATE EXISTING TABLE 都可以使用 AT 关键字定义现有对象的位置。该位置字符串由四部分组成，每部分都由句点或分号隔开。分号分隔符允许在数据库和所有者字段中使用文件名和扩展名。

AT 子句的语法如下：

```
... AT 'server.database.owner.table-name'
```

- **Server**　这是服务器在当前数据库中使用的名称（在 CREATE SERVER 语句中指定的名称）。对于所有远程数据源，此字段是必需的。
- **Database**　数据库字段的含义取决于数据源。在某些情况下，此字段不适用，应该留空。但是，句点仍是必需的。

  在 Adaptive Server Enterprise 中，*database* 指定表所在的数据库。例如 master 或 pubs2。

  在 Adaptive Server Anywhere 中，此字段不适用，请将其留空。

  在 Excel、Lotus Notes 和 Access 中，您必须包括包含该表的文件的名称。如果文件名中包含句点，请使用分号分隔符。
- **Owner**　如果数据库支持所有权的概念，则此字段表示所有者名称。只有当多个所有者具有使用相同名称的表时才需要此字段。

- **Table-name** 这指定表的名称。如果是 Excel 电子表格，则这是工作簿中 [工作表] 的名称。如果将表名留空，则假定远程表的名称与本地代理表的名称相同。

示例： 以下示例说明位置字符串的用法：

- Adaptive Server Anywhere：

```
'testasa..DBA.employee'
```

- Adaptive Server Enterprise：

```
'ASEServer.pubs2.dbo.publishers'
```

- Excel：

```
'excel;d:\pcdb\quarter3.xls;;sheet1$'
```

- Access：

```
'access;\\server1\production\inventory.mdb;;parts'
```

创建代理表 (Sybase Central)

您可以使用 Sybase Central 或 CREATE EXISTING TABLE 语句创建代理表。您不能为系统表创建代理表。

CREATE EXISTING TABLE 语句创建映射到远程服务器上现有表的代理表。 Adaptive Server Anywhere 从位于远程位置的对象派生列属性和索引信息。

有关 CREATE EXISTING TABLE 语句的信息，请参见 『ASA SQL 参考』> 「CREATE EXISTING TABLE 语句」。

❖ 创建代理表 (Sybase Central)：

1 从 Sybase Central 连接到主机数据库。

2 在左窗格中，打开 [远程服务器] 文件夹。

3 在右窗格中，单击 [代理表] 选项卡。

4　从 [文件] 菜单中，选择 [新建] → [代理表]。

5　请按照向导中的说明进行操作。

提示
当代理表的远程服务器在左窗格中处于选中状态时，代理表会显示在 [代理表] 选项卡上的右窗格中。它们还显示在 [表] 文件夹中。它们通过其图标上的字母 P 与其它表进行区分。

使用 CREATE EXISTING TABLE 语句创建代理表

CREATE EXISTING TABLE 语句创建映射到远程服务器上现有表的代理表。 Adaptive Server Anywhere 从位于远程位置的对象派生列属性和索引信息。

❖ 使用 CREATE EXISTING TABLE 语句创建代理表 (SQL)：

1　连接到主机数据库。

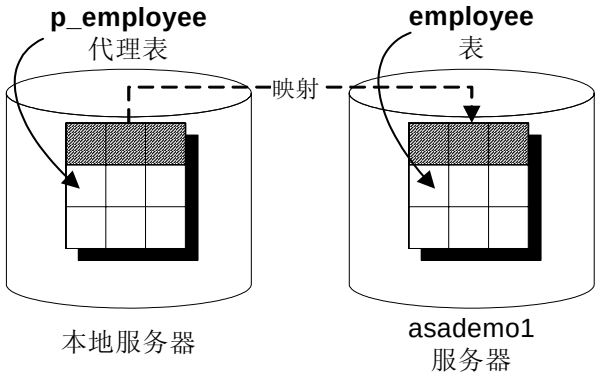
2　执行 CREATE EXISTING TABLE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE EXISTING TABLE 语句」。

示例 1

若要在当前服务器上为名为 asademo1 的服务器上名为 employee 的远程表创建名为 p_employee 的代理表，请使用以下语法：

```
CREATE EXISTING TABLE p_employee
 AT 'asademo1..DBA.employee'
```



示例 2

以下语句将代理表 a1 映射到 Microsoft Access 文件 mydbfile.mdb。在此示例中， AT 关键字使用分号 (;) 作为分隔符。为 Microsoft Access 定义的服务器名为 access。

```
CREATE EXISTING TABLE a1
 AT'access;d:\mydbfile.mdb;;a1'
```

使用 CREATE TABLE 语句创建代理表

CREATE TABLE 语句在远程服务器上创建新表，如果使用 AT 选项，则可以为该表定义代理表。您可以使用 Adaptive Server Anywhere 数据类型输入 CREATE TABLE 语句。 Adaptive Server Anywhere 自动将该数据转换为远程服务器的本机类型。

如果您使用 CREATE TABLE 语句创建本地和远程表，并随后使用 DROP TABLE 语句删除代理表，则远程表也将被删除。但是，如果您不想删除远程表，则可以使用 DROP TABLE 语句删除使用 CREATE EXISTING TABLE 语句创建的代理表。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

❖ 使用 CREATE EXISTING TABLE 语句创建代理表 (SQL)：

1 连接到主机数据库。

2　执行 CREATE TABLE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TABLE 语句」。

示例

以下语句在远程服务器 asademo1 上创建名为 employee 的表，并创建映射到该远程位置的名为 members 的代理表：

```
CREATE TABLE members
 ( membership_id INTEGER NOT NULL,
 member_name CHAR(30) NOT NULL,
 office_held CHAR( 20 ) NULL)
 AT 'asademo1..DBA.employee'
```

列出远程表上的列

如果您正在输入 CREATE EXISTING 语句或指定列列表，则获取远程表上可用列的列表将很有帮助。 sp_remote_columns 系统过程生成远程表上列的列表和这些数据类型的说明。

```
sp_remote_columns servername, tablename [, owner ]
 [, database]
```

如果给定了表名、所有者或数据库名，则列的列表仅限于那些符合条件的列。

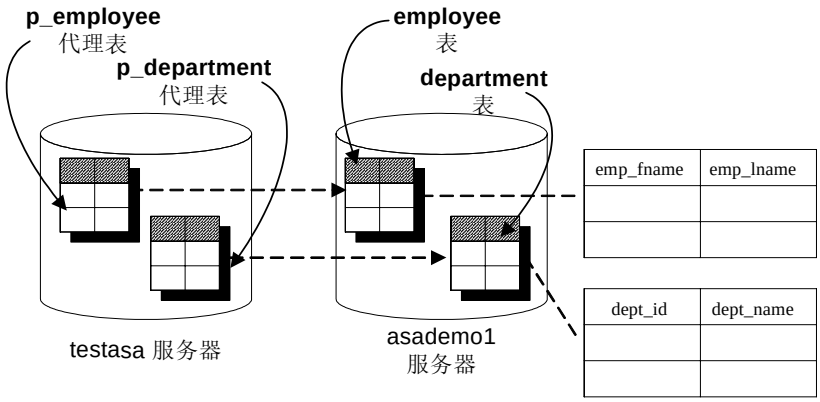
例如，以下语句返回名为 asetest 的 Adaptive Server Enterprise 服务器上 production 数据库中 sysobjects 表中列的列表：

```
sp_remote_columns  asetest, sysobjects, null,
production
```

有关详细信息，请参见 『ASA SQL 参考』> 「sp_remote_columns 系统过程」。

连接远程表

下图说明示例数据库中映射到名为 testasa 的本地服务器的远程 Adaptive Server Anywhere 表 employee 和 department。



在实际情况下，您可能会在不同 Adaptive Server Anywhere 数据库上的表之间使用连接。这里我们介绍一个简单的示例，该示例只使用一个数据库来说明这些原则。

❖ 在两个远程表之间进行连接 (SQL)：

1. 创建一个名为 *empty.db* 的新数据库。

   该数据库未保存任何数据。我们将只使用它来定义远程对象和通过它访问示例数据库。

2. 启动运行 empty.db 和示例数据库的数据库服务器。您可以使用以下命令行 （从安装目录执行）完成此任务：

   ```
   dbeng9 asademo empty
   ```

3. 使用用户 ID **DBA** 和口令 **SQL** 从 Interactive SQL 连接到 *empty.db*。

4. 在新数据库中，创建一个名为 testasa 的远程服务器。其服务器类为 asaodbc，连接信息是 **'ASA 9.0 Sample'**：

```
CREATE SERVER testasa
 CLASS 'asaodbc'
 USING 'ASA 9.0 Sample'
```

5　在本例中，我们在远程数据库和本地数据库上使用相同的用户 ID 和口令，因此不需要外部登录。

6　定义 employee 代理表：

```
CREATE EXISTING TABLE employee
 AT 'testasa..DBA.employee'
```

7　定义 department 代理表：

```
CREATE EXISTING TABLE department
 AT 'testasa..DBA.department'
```

8　在 SELECT 语句中使用代理表执行连接。

```
SELECT emp_fname, emp_lname, dept_name
 FROM employee JOIN department
 ON employee.dept_id = department.dept_id
 ORDER BY emp_lname
```

连接多个本地数据库中的表

Adaptive Server Anywhere 服务器上可能有多个本地数据库在同时运行。通过将其它本地 Adaptive Server Anywhere 数据库中的表定义为远程表，您可以执行跨数据库连接。

有关指定多个数据库的详细信息，请参见 「CREATE SERVER 语句中的 USING 参数值」第 680 页。

示例

例如，如果您正在使用数据库 db1 并想访问数据库 db2 的表中的数据，则需要设置指向数据库 db2 中表的代理表定义。例如，在名为 testasa 的 Adaptive Server Anywhere 上，您可能有三个数据库可用：db1、 db2 和 db3。

- 如果使用 ODBC，则为您将访问的每个数据库创建一个 ODBC 数据源名。
- 连接到您将执行连接的数据库之一。例如，连接到 db1。
- 为每个您将访问的其它本地数据库执行 CREATE SERVER 语句。这将设置到 Adaptive Server Anywhere 服务器的**回环**连接。

```
CREATE SERVER local_db2
 CLASS 'asaodbc'
 USING 'testasa_db2'
 CREATE SERVER local_db3
 CLASS 'asaodbc'
 USING     'testasa_db3'
```

或者，使用 JDBC：

```
CREATE SERVER local_db2
 CLASS 'asajdbc'
 USING 'mypc1:2638/db2'
 CREATE SERVER local_db3
 CLASS 'asajdbc'
 USING 'mypc1:2638/db3'
```

- 使用 CREATE EXISTING 创建其它要访问的数据库中表的代理表定义。

```
CREATE EXISTING TABLE employee
 AT 'local_db2...employee'
```

将本机语句发送到远程服务器

使用 FORWARD TO 语句将一个或多个语句以其本机语法发送到远程服务器。可以通过两种方式使用此语句：

- 将语句发送到远程服务器。
- 让 Adaptive Server Anywhere 进入直通模式以便向远程服务器发送一系列语句。

如果无法建立与指定服务器的连接，则将向用户发送一条消息解释原因。如果建立了连接，则会将任何结果转换为客户端程序可以识别的格式。

FORWARD TO 语句可用于校验是否正确配置了服务器。如果您将语句发送到远程服务器， Adaptive Server Anywhere 未返回错误消息，则远程服务器配置正确。

不能在过程或批处理中使用 FORWARD TO 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「FORWARD TO 语句」。

示例 1

以下语句通过选择版本字符串校验与名为 ASEserver 的服务器的连接：

```
FORWARD TO ASEserver {SELECT @@version}
```

示例 2

以下语句显示与名为 ASEserver 的服务器的直通会话：

```
FORWARD TO ASEserver
 select * from titles
 select * from authors
 FORWARD TO
```

使用远程过程调用 (RPC)

Adaptive Server Anywhere 用户可以向支持过程调用功能的远程服务器发布过程调用。

Sybase Adaptive Server Anywhere、 Sybase Adaptive Server Enterprise、Oracle 和 DB2 都支持此功能。发布远程过程调用与使用本地过程调用类似。

创建远程过程

您可以使用 Sybase Central 或 CREATE PROCEDURE 语句发布远程过程调用。

您必须具有 DBA 权限才能创建远程过程。

❖ 发布远程过程调用 (Sybase Central)：

1. 从 Sybase Central 连接到主机数据库。
2. 打开 [远程服务器] 文件夹。
3. 在左窗格中，选择要为其创建远程过程的远程服务器。
4. 在右窗格中，单击 [远程过程] 选项卡。
5. 从 [文件] 菜单中，选择 [新建] → [远程过程]。

   即会出现 [远程过程创建] 向导。
6. 请按照向导中的说明进行操作。

❖ 发布远程过程调用 (SQL)：

1. 首先将该过程定义到 Adaptive Server Anywhere。

   语法与本地过程定义相同，只是不使用 SQL 语句组成调用的主体，而是提供一个位置字符串定义该过程所驻留的位置。

```
CREATE PROCEDURE remotewho ()
 AT 'bostonase.master.dbo.sp_who'
```

2　执行该过程，如下所示：

```
call remotewho()
```

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE PROCEDURE 语句」。

示例

这里是一个具有参数的示例：

```
CREATE PROCEDURE remoteuser (IN uname char(30))
 AT 'bostonase.master.dbo.sp_helpuser'
 call remoteuser('joe')
```

远程过程的数据类型

RPC 参数可以使用以下数据类型。不允许使用其它数据类型：

- [UNSIGNED] SMALLINT
- [UNSIGNED] INT
- [UNSIGNED] BIGINT
- TINYINT
- REAL
- DOUBLE
- CHAR
- BIT

NUMERIC 和 DECIMAL 数据类型可以用于 IN 参数，但不能用于 OUT 或 INOUT 参数。

删除远程过程

您可以使用 Sybase Central 或 DROP PROCEDURE 语句删除远程过程。

您必须具有 DBA 权限才能删除远程过程。

❖ **删除远程过程 (Sybase Central)：**

1 打开 [远程服务器] 文件夹。

2 在左窗格中，选择远程服务器。

3 在右窗格中，单击 [远程过程] 选项卡。

4 在 [远程过程] 选项卡上，右击远程过程，然后从弹出式菜单上选择 [删除]。

❖ **删除远程过程 (SQL)：**

1 连接到数据库。

2 执行 DROP PROCEDURE 语句。

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

示例 删除名为 remoteproc 的远程过程。

```
DROP PROCEDURE remoteproc
```

事务管理和远程数据

事务提供一种对 SQL 语句进行分组的方式，这样可以将这些语句作为一个整体对待 - 要么将这些语句执行的所有工作都提交到数据库，要么一个也不提交。

远程表的事务管理在很大程度上与 Adaptive Server Anywhere 中本地表的事务管理相同，但也有一些差异。下面一节将对它们进行介绍。

有关事务的一般讨论，请参见 「使用事务和隔离级别」第 111 页。

远程事务管理概述

管理涉及远程服务器的事务的方法使用两阶段提交协议。 Adaptive Server Anywhere 执行的策略可以确保大多数情况的事务完整性。但是，当一个事务中调用了多个远程服务器时，仍然存在分布式工作单位处于不确定状态的可能性。即使使用了两阶段提交协议，也不包括恢复进程。

管理用户事务的一般逻辑如下：

1. Adaptive Server Anywhere 以 BEGIN TRANSACTION 通知向远程服务器引出工作。
2. 当准备好提交事务时， Adaptive Server Anywhere 向已成为事务一部分的每个远程服务器发送一个 PREPARE TRANSACTION 通知。这可以确保远程服务器准备好提交事务。
3. 如果 PREPARE TRANSACTION 请求失败，则将指示所有远程服务器回退当前事务。

   如果所有 PREPARE TRANSACTION 请求都成功，则服务器将向该事务涉及的每个远程服务器发送一个 COMMIT TRANSACTION 请求。

以 BEGIN TRANSACTION 开头的任何语句都可以开始一个事务。其它语句被发送到远程服务器作为单个、远程工作单位执行。

事务管理的限制

事务管理的限制如下：

- 不将保存点传播到远程服务器。
- 如果涉及远程服务器的事务中包含嵌套的 BEGIN TRANSACTION 和 COMMIT TRANSACTION 语句，则只处理最外层的一组语句。不将最内层的语句组 （包含 BEGIN TRANSACTION 和 COMMIT TRANSACTION 语句）传输到远程服务器。

内部操作

本节介绍 Adaptive Server Anywhere 代表客户端应用程序对远程服务器执行的基础操作。

查询分析

当从客户端接收到语句时，将对该语句进行分析。如果该语句不是有效的 Adaptive Server Anywhere SQL 语句，则将引发一个错误。

查询规范化

下一个步骤称为查询规范化。在此步骤中，将校验引用的对象并检查某些数据类型兼容性。

例如，假定有以下查询：

```
SELECT *
 FROM t1
 WHERE c1 = 10
```

查询规范化阶段校验系统表中是否存在具有列 c1 的表 t1。它还校验列 c1 的数据类型是否和值 10 兼容。例如，如果该列的数据类型为 datetime 型，则将拒绝此语句。

查询预处理

查询预处理对查询进行优化准备。它可能会更改语句的表示法，因此 Adaptive Server Anywhere 生成的用于传递到远程服务器的 SQL 语句在语句构成上将不同于原始语句。

预处理执行视图扩展，以便查询可以对视图引用的表执行操作。可以对表达式进行重新排序并对子查询进行转换以提高处理效率。例如，可以将某些子查询转换为连接。

服务器功能

前面的步骤是对所有查询 （本地和远程）执行的。

以下步骤取决于 SQL 语句的类型和所涉及的远程服务器的功能。

定义到 Adaptive Server Anywhere 的每个远程服务器都有一组与之关联的功能。这些功能存储在 syscapabilities 系统表中。在首次连接到远程服务器的过程中将对这些功能进行初始化。一般服务器类 odbc 严格按照从 ODBC 驱动程序返回的信息来确定这些功能。其它服务器类 （例如 db2odbc）具有远程服务器类型功能的更详细的信息，并使用该信息补充驱动程序返回的内容。

一旦为服务器初始化了 syscapabilities 之后，就只能从系统表检索功能信息了。这允许用户变更服务器的已知功能。

由于远程服务器可能并不支持给定 SQL 语句的所有功能，所以 Adaptive Server Anywhere 必须将语句分解成较简单的组成部分，达到可以将该查询提供给远程服务器的程度。未传递给远程服务器的 SQL 功能必须由 Adaptive Server Anywhere 本身进行处理。

例如，某个查询可能包含一个 ORDER BY 语句。如果远程服务器不能执行 ORDER BY，则将语句发送到远程服务器时将不包括 ORDER BY，当返回结果时， Adaptive Server Anywhere 执行 ORDER BY，然后再将该结果返回给用户。这样做的结果是用户可以使用 Adaptive Server Anywhere 支持的全部 SQL 功能，而不用担心特定后端的功能。

语句的完整直通

通常，处理语句最有效的方式是将尽可能多的原始语句发送到所涉及的远程服务器。 Adaptive Server Anywhere 将试图传递出尽可能多的语句。在大多数情况下，这将是最初提供给 Adaptive Server Anywhere 的完整语句。

在以下情况下， Adaptive Server Anywhere 将传递出完整的语句：

- 语句中的所以表都驻留在同一个远程服务器上。
- 远程服务器能够处理语句中的所有语法。

在少数情况下，让 Adaptive Server Anywhere 执行某些工作实际上可能比传递出这些工作更有效。例如， Adaptive Server Anywhere 可能具有更好的排序算法。在这种情况下，您可以考虑使用 ALTER SERVER 语句变更远程服务器的功能。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER SERVER 语句」。

语句的部分直通

如果某个语句包含对多个服务器的引用，或者使用远程服务器不支持的 SQL 功能，则会将查询分解为多个较简单的部分。

Select

通过移除不能传递的部分并让 Adaptive Server Anywhere 执行该功能，可以对 SELECT 语句进行分解。例如，假设远程服务器不能处理以下语句中的 atan2() 函数：

```
select a,b,c where atan2(b,10) > 3 and c = 10
```

发送到远程服务器的语句将被转换为：

```
select a,b,c where c = 10
```

Adaptive Server Anywhere 会在本地将 "where atan2(b,10) > 3" 应用到中间结果集。

连接

当连接两个表时，将选择一个表作为外表。根据应用到外表的 WHERE 条件对外表进行扫描。对于找到的每个符合条件的行，将对另一个表（称为内部表）进行扫描以查找符合连接条件的行。

当引用远程表时将使用相同的算法。因为搜索远程表通常要比搜索本地表耗费更多的资源 （由于网络 I/O），所以将采取一切办法让远程表成为连接中最外面的表。

UPDATE 和 DELETE

如果 Adaptive Server Anywhere 不能将 UPDATE 或 DELETE 语句完全传递给远程服务器，则它必须将该语句转换为一个表扫描，该表扫描包含原始 WHERE 子句尽量多的部分，随后在找到符合条件的行时，对 [游标当前所指的行] 执行定位的 UPDATE 或 DELETE。

例如，如果远程服务器不支持函数 **atan2**：

```
UPDATE t1
 SET a = atan2(b, 10)
 WHERE b > 5
```

将被转换为以下内容：

```
SELECT a,b
 FROM t1
 WHERE  b > 5
```

每次找到一行时， Adaptive Server Anywhere 都会计算 a 的新值并发布：

```
UPDATE t1
 SET a = 'new value'
 WHERE CURRENT OF CURSOR
```

如果 a 已具有等于 [新值] 的值，则没有必要执行定位的 UPDATE，也不会远程发送定位的 UPDATE。

为了处理需要表扫描的 UPDATE 或 DELETE，远程数据源必须支持 （[对游标当前所指的行]）执行定位的 UPDATE 或 DELETE 的功能。某些数据源不支持此功能。

不能更新临时表
在此版本的 Adaptive Server Anywhere 中，如果 Adaptive Server Anywhere 中需要中间临时表，则不能执行 UPDATE 或 DELETE。这种情况发生在具有 ORDER BY 的查询和某些具有子查询的查询中。

远程数据访问疑难解答

本节提供远程服务器疑难解答的一些提示。

不支持远程数据的功能

不支持对远程数据执行以下 Adaptive Server Anywhere 功能。试图使用这些功能将因此遇到问题：

- 对远程表执行 ALTER TABLE 语句
- 在代理表上定义的触发器将不会触发
- SQL Remote
- 忽略引用远程表的外键
- READTEXT、 WRITETEXT 和 TEXTPTR 函数。
- 定位的 UPDATE 和 DELETE
- 需要中间临时表的 UPDATE 和 DELETE。
- 向后滚动对远程数据打开的游标。 Fetch 语句必须是 NEXT 或 RELATIVE 1。
- 如果远程表上的某个列的名称是远程服务器上的关键字，则您不能访问该列中的数据。 Adaptive Server Anywhere 无法知道所有远程服务器保留字。您可以执行 CREATE EXISTING TABLE 语句并导入定义，但不能选择该列。

区分大小写

您的 Adaptive Server Anywhere 数据库的区分大小写设置应该和所访问的任何远程服务器使用的设置匹配。

缺省情况下，创建 Adaptive Server Anywhere 数据库时是不区分大小写的。如果使用此配置，当从区分大小写的数据库进行选择时将出现不可预知的结果。根据 ORDER BY 或字符串比较是被传递到远程服务器还是由本地 Adaptive Server Anywhere 执行，将出现不同的结果。

连接问题

执行以下步骤以确定您可以连接到远程服务器：

- 在配置 Adaptive Server Anywhere 之前使用客户端工具（例如 Interactive SQL）确定您可以连接到远程服务器。
- 执行到远程服务器的简单的直通语句以检查您的连接和远程登录配置。例如：

  ```
  FORWARD TO testasa {SELECT @@version}
  ```

- 打开远程跟踪以跟踪与远程服务器的交互。

  ```
  SET OPTION CIS_OPTION = 7
  ```

  一旦打开远程跟踪功能，跟踪信息就会出现在数据库服务器窗口中。通过在启动数据库服务器时指定 -o *filename* 服务器选项，您可以将此输出记录到文件中。

  有关 CIS_OPTION 选项的详细信息，请参见 『ASA 数据库管理指南』> 「CIS_OPTION 选项 [database]」。

  有关 -o 服务器选项的详细信息，请参见 『ASA 数据库管理指南』> 「-o 服务器选项」。

查询的一般问题

如果您遇到某些类型的问题涉及 Adaptive Server Anywhere 对远程表执行查询的方式，则了解 Adaptive Server Anywhere 如何执行该查询通常会很有帮助。您可以显示远程跟踪以及查询执行计划的说明：

```
SET OPTION CIS_OPTION = 7
```

有关在使用远程数据访问时使用 CIS_OPTION 选项调试查询的详细信息，请参见 『ASA 数据库管理指南』> 「CIS_OPTION 选项 [database]」。

一旦打开远程跟踪功能，跟踪信息就会出现在数据库服务器窗口中。通过在启动数据库服务器时指定 -o *filename* 服务器选项，您可以将此输出记录到文件中。

有关 -o 服务器选项的详细信息，请参见 『ASA 数据库管理指南』> 「-o 服务器选项」。

查询自身阻塞

如果您访问单个 Adaptive Server Anywhere 服务器上的多个数据库，则可能需要在 Windows 上使用 -gx 命令行选项增加该数据库服务器使用的线程的数目。

您必须具有足够的可用线程来支持由某个查询运行的各个任务。如果未能提供所需任务数目的线程，则可能导致查询自身阻塞。

管理远程数据访问连接

如果您通过 ODBC 访问远程数据库，到远程服务器的连接就会被指定一个名称。该名称可用于断开连接，作为取消远程请求的一种方法。

连接会被命名为 ASACIS_*conn-name*，其中 *conn-name* 是本地连接的连接 ID。该连接 ID 可以从 sa_conn_info 存储过程获得。

第 19 章

用于进行远程数据访问的服务器类

关于本章

本章将介绍 Adaptive Server Anywhere 如何与不同的服务器类进行连接。本章将介绍

- 每一个服务器类所支持的服务器类型
- 每一个服务器类的 CREATE SERVER 语句的 USING 子句值
- 特殊配置要求

概述

您在 CREATE SERVER 语句中所指定的服务器类决定了远程连接的行为。这些服务器类向 Adaptive Server Anywhere 提供详细的服务器功能信息。 Adaptive Server Anywhere 针对服务器的功能来设置 SQL 语句的格式。

服务器类包含两个类别：

- 基于 JDBC 的服务器类
- 基于 ODBC 的服务器类

每一个服务器类都具有一组独特的特性，数据库管理员和程序员需要知道这些特性，才能配置用于进行远程数据访问的服务器。

在学习本章时，请参考关于服务器类类别 （基于 JDBC 的类别或基于 ODBC 的类别）的章节以及关于各个服务器类的章节。

基于 JDBC 的服务器类

当 Adaptive Server Anywhere 在内部使用 Java 虚拟机和 jConnect 4.0 连接到远程服务器时，使用的便是基于 JDBC 的服务器类。基于 JDBC 的服务器类包括：

- **asajdbc**　Adaptive Server Anywhere （第 6 版和更高版本）。
- **asejdbc**　Adaptive Server Enterprise 和 SQL Server （第 10 版和更高版本）。

如果使用的是 NetWare，则只支持 asajdbc 类。

有关基于 JDBC 的类的配置说明

在访问用基于 JDBC 的类定义的远程服务器时，请考虑以下因素：

- 您的本地数据库必须能够支持 Java。

  有关详细信息，请参见 『ASA 编程指南』> 「为数据库启用 Java」。

- 在装入和运行 jConnect 时， Java 虚拟机需要的内存量超过缺省内存量。至少应将这些内存选项设置为以下值：

  ```
  SET OPTION PUBLIC.JAVA_NAMESPACE_SIZE = 3000000
  SET OPTION PUBLIC.JAVA_HEAP_SIZE = 1000000
  ```

- 因为 jConnect 4.0 是自动与 Adaptive Server Anywhere 一起安装的，所以不需要安装额外的驱动程序。
- 为获得最佳的性能， Sybase 建议采用基于 ODBC 的类 （asaodbc 或 aseodbc）。
- 必须安装可以使用 asejdbc 或 asajdbc 服务器类进行访问的远程服务器，以处理基于 jConnect 4.x 的客户端。 jConnect 安装脚本是 *SQL_anywhere.sql* （对于 Adaptive Server Anywhere）或 *sql_server.sql* （对于 Adaptive Server Enterprise）。请根据您使用的远程服务器类型运行上述脚本。

服务器类 asajdbc

具有服务器类 asajdbc 的服务器是 Adaptive Server Anywhere （第 6 版和更高版本）。对于 Adaptive Server Anywhere 数据源，没有任何特殊的配置要求。

CREATE SERVER 语句中的 USING 参数值

必须为要访问的每一个 Adaptive Server Anywhere 数据库执行单独的 CREATE SERVER 语句。例如，如果一个名为 testasa 的 Adaptive Server Anywhere 服务器在计算机 'banana' 上运行，并且它拥有三个数据库（db1、 db2、 db3），则配置本地 Adaptive Server Anywhere 的方式类似于：

```
CREATE SERVER testasadb1
 CLASS 'asajdbc'
 USING 'banana:2638/db1'
 CREATE SERVER testasadb2
 CLASS 'asajdbc'
 USING 'banana:2638/db2'
 CREATE SERVER testasadb2
 CLASS 'asajdbc'
 USING 'banana:2638/db3'
```

如果您不指定 */databasename* 值，则远程连接将使用远程 Adaptive Server Anywhere 缺省数据库。

有关 CREATE SERVER 语句的详细信息，请参见 『ASA SQL 参考』> 「CREATE SERVER 语句」。

服务器类 asejdbc

具有 asejdbc 服务器类的服务器是 Adaptive Server Enterprise, SQL Server （第 10 版和更高版本）。对于 Adaptive Server Enterprise 数据源，没有任何特殊的配置要求。

数据类型转换：JDBC 和 Adaptive Server Enterprise

当您发出 CREATE TABLE 语句后， Adaptive Server Anywhere 会自动将数据类型转换为相应的 Adaptive Server Enterprise 数据类型。下表列出了 Adaptive Server Anywhere 与 Adaptive Server Enterprise 之间的数据类型转换关系。

Adaptive Server Anywhere 数据类型	ASE 缺省数据类型
bit	bit
tinyint	tinyint
smallint	smallint
int	int
integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	不支持
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	不支持
float	real
real	real
double	float
smallmoney	numeric(10,4)
money	numeric(19,4)
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime

Adaptive Server Anywhere 数据类型	ASE 缺省数据类型
char(n)	varchar(n)
character(n)	varchar(n)
varchar(n)	varchar(n)
character varying(n)	varchar(n)
long varchar	text
text	text
binary(n)	binary(n)
long binary	image
image	image
bigint	numeric(20,0)

基于 ODBC 的服务器类

基于 ODBC 的服务器类包括：

- asaodbc
- aseodbc
- db2odbc
- mssodbc
- oraodbc
- odbc

定义 ODBC 外部服务器

定义基于 ODBC 的服务器的最常用方法是将其基于 ODBC 数据源。若要进行这样的定义，您必须在 [ODBC 管理器] 中创建数据源。

定义完数据源之后， CREATE SERVER 语句中的 USING 子句应该与 ODBC 数据源名匹配。

例如，若要配置名为 **mydb2** 的 DB2 服务器 （其数据源名也是 **mydb2**），请使用：

```
CREATE SERVER mydb2
 CLASS 'db2odbc'
 USING 'mydb2'
```

有关创建数据源的详细信息，请参见 『ASA 数据库管理指南』> 「创建 ODBC 数据源」。

使用连接字符串代替数据源

避免使用数据源的一个方法是，在 CREATE SERVER 语句的 USING 子句中提供连接字符串。若要提供连接字符串，必须知道您所使用的 ODBC 驱动程序的连接参数。例如，与 ASA 的连接可以是：

```
CREATE SERVER testasa
 CLASS 'asaodbc'
 USING 'driver=adaptive server anywhere
9.0;eng=testasa;dbn=sample;links=tcpip{}'
```

这就定义了与名为 **testasa** 的 Adaptive Server Anywhere 数据库服务器的连接，其中数据库是 **sample**，使用的协议是 TCP-IP。

另请参见

有关特定的 ODBC 服务器类的信息，请参见：

- 「服务器类 asaodbc」第 684 页
- 「服务器类 aseodbc」第 684 页
- 「服务器类 db2odbc」第 687 页
- 「服务器类 oraodbc」第 689 页
- 「服务器类 mssodbc」第 691 页
- 「服务器类 odbc」第 693 页

服务器类 asaodbc

具有 asaodbc 服务器类的服务器是 Adaptive Server Anywhere 5.5 版或更高版本。对于 Adaptive Server Anywhere 数据源，没有任何特殊的配置要求。

若要访问可以支持多个数据库的 Adaptive Server Anywhere 服务器，请创建 ODBC 数据源名，并定义与每一个数据库的连接。为所创建的每一个 ODBC 数据源名发出 CREATE SERVER 语句。

服务器类 aseodbc

具有 aseodbc 服务器类的服务器是 Adaptive Server Enterprise, SQL Server（第 10 版和更高版本）。 Adaptive Server Anywhere 要求必须安装 Adaptive Server Enterprise ODBC 驱动程序和 Open Client 连接库，才能连接到具有类 aseodbc 的远程 Adaptive Server。其性能要优于具有 asejdbc 类的 Adaptive Server。

注意

- Open Client 应为版本 11.1.1, EBF 7886 或更高版本。在安装 ODBC 并配置 Adaptive Server Anywhere 之前，请首先安装 Open Client 并验证与 Adaptive Server 的连接。 Sybase ODBC 驱动程序应是版本 11.1.1, EBF 7911 或更高版本。
- 在配置管理器中用以下属性配置用户数据源：
  - 在 [General] 选项卡中：

    为 [Data Source Name] 输入任意值。该值将在 CREATE SERVER 语句的 USING 子句中使用。

    服务器名应匹配 Sybase 接口文件中的服务器的名称。
  - 在 [Advanced] 选项卡中，选中 [Application Using Threads] 复选框，并选中 [Enable Quoted Identifiers] 复选框。
  - 在 [Connection] 选项卡中：

    设置字符集字段，以便与您的 Adaptive Server Anywhere 字符集匹配。

    将语言字段设置为您的用于显示错误消息的首选语言。
  - 在 [Performance] 选项卡中：

    将 [Prepare Method] 设置为 [2-Full]。

    将 [Fetch Array Size] 设置得尽可能大，以获得最佳性能。这会增加内存要求，因为该值是必须被高速缓存到内存中的行数。 Sybase 建议使用的值为 100。

    将 [Select Method] 设置为 [0-Cursor]。

    将 [Packet Size] 设置得尽可能大。 Sybase 建议使用的值是 -1。

    将 [Connection Cache] 设置为 1。

数据类型转换：ODBC 和 Adaptive Server Enterprise

当您发出 CREATE TABLE 语句后， Adaptive Server Anywhere 会自动将数据类型转换为相应的 Adaptive Server Enterprise 数据类型。下表列出了 Adaptive Server Anywhere 与 Adaptive Server Enterprise 之间的数据类型转换关系。

Adaptive Server Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
Bit	bit
Tinyint	tinyint
Smallint	smallint
Int	int
Integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	不支持
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	不支持
Float	real
Real	real
Double	float
Smallmoney	numeric(10,4)
Money	numeric(19,4)
Date	datetime
Time	datetime
Timestamp	datetime
Smalldatetime	datetime
Datetime	datetime

Adaptive Server Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
char(n)	varchar(n)
Character(n)	varchar(n)
varchar(n)	varchar(n)
Character varying(n)	varchar(n)
long varchar	text
Text	text
binary(n)	binary(n)
long binary	image
Image	image
Bigint	numeric(20,0)

服务器类 db2odbc

具有服务器类 db2odbc 的服务器是 IBM DB2

注意

- Sybase 认证 IBM 的 DB2 Connect 版本 5（并应用了补丁程序包 WR09044）能正常使用。根据该产品的说明进行配置，并测试您的 ODBC 配置。 Adaptive Server Anywhere 对 DB2 数据源的配置没有任何具体要求。
- 以下是 CREATE EXISTING TABLE 语句的示例，该语句适用于一个 DB2 服务器，其 ODBC 数据源名为 mydb2：

```
CREATE EXISTING TABLE ibmcol
 AT 'mydb2..sysibm.syscolumns'
```

数据类型转换：DB2

当您发出 CREATE TABLE 语句后， Adaptive Server Anywhere 会自动将数据类型转换为对应的 DB2 数据类型。下表列出了 Adaptive Server Anywhere 与 DB2 之间的数据类型转换关系。

Adaptive Server Anywhere 数据类型	DB2 缺省数据类型
Bit	smallint
Tinyint	smallint
Smallint	smallint
Int	int
Integer	int
Bigint	decimal(20,0)
char(1-254)	varchar(n)
char(255-4000)	varchar(n)
char(4001-32767)	long varchar
Character(1-254)	varchar(n)
Character(255-4000)	varchar(n)
Character(4001-32767)	long varchar
varchar(1-4000)	varchar(n)
varchar(4001-32767)	long varchar
Character varying(1-4000)	varchar(n)
Character varying(4001-32767)	long varchar
long varchar	long varchar
text	long varchar
binary(1-4000)	varchar for bit data

Adaptive Server Anywhere 数据类型	DB2 缺省数据类型
binary(4001-32767)	long varchar for bit data
long binary	long varchar for bit data
image	long varchar for bit data
decimal [defaults p=30, s=6]	decimal(30,6)
numeric [defaults p=30 s=6]	decimal(30,6)
decimal(128, 128)	不支持
numeric(128, 128)	不支持
real	real
float	float
double	float
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time
smalldatetime	timestamp
datetime	timestamp
timestamp	timestamp

服务器类 oraodbc

具有 **oraodbc** 服务器类的服务器是 Oracle 8.0 版或更高版本。

注意

- Sybase 认证 Oracle 的 ODBC 驱动程序 8.0.03 版本能正常使用。根据该产品的说明进行配置，并测试您的 ODBC 配置。

- 以下是名为 myora 的 Oracle 服务器的 CREATE EXISTING TABLE 语句的示例：

```
CREATE EXISTING TABLE employees
 AT 'myora.database.owner.employees'
```

- 由于 Oracle ODBC 驱动程序的限制，您不能对系统表发出 CREATE EXISTING TABLE 语句。如果您这样做，将会返回一则消息，说明无法找到表或列。

数据类型转换：Oracle

当您发出 CREATE TABLE 语句后， Adaptive Server Anywhere 会自动将数据类型转换为相应的 Oracle 数据类型。下表列出了 Adaptive Server Anywhere 与 Oracle 之间的数据类型转换关系。

Adaptive Server Anywhere 数据类型	Oracle 数据类型
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)
int	number(11,0)
bigint	number(20,0)
decimal(prec, scale)	number(prec, scale)
numeric(prec, scale)	number(prec, scale)
float	float
real	real
smallmoney	numeric(13,4)
money	number(19,4)
date	date
time	date

Adaptive Server Anywhere 数据类型	Oracle 数据类型
timestamp	date
smalldatetime	date
datetime	date
char(n)	if (n > 255) long else varchar(n)
varchar(n)	if (n > 2000) long else varchar(n)
long varchar	long 或 clob
binary(n)	if (n > 255) long raw else raw(n)
varbinary(n)	if (n > 255) long raw else raw(n)
long binary	long raw

服务器类 mssodbc

具有 **mssodbc** 服务器类的服务器是 Microsoft SQL Server 6.5 版，Service Pack 4。

注意

- Sybase 认证 Microsoft SQL Server 的 ODBC 驱动程序 3.60.0319 版（包括在 MDAC 2.0 版本中）能正常使用。根据该产品的说明进行配置，并测试您的 ODBC 配置。
- 以下是名为 mymssql 的 Microsoft SQL Server 的 CREATE EXISTING TABLE 语句的示例：

```
CREATE EXISTING TABLE accounts,
 AT 'mymssql.database.owner.accounts'
```

数据类型转换：Microsoft SQL Server

当您发出 CREATE TABLE 语句后， Adaptive Server Anywhere 会自动将数据类型转换为相应的 Microsoft SQL Server 数据类型。下表列出了 Adaptive Server Anywhere 与 Microsoft SQL Server 之间的数据类型转换关系。

Adaptive Server Anywhere 数据类型	Microsoft SQL Server 缺省数据类型
bit	bit
tinyint	tinyint
smallint	smallint
int	int
bigint	numeric(20,0)
decimal [defaults p=30, s=6]	decimal(prec, scale)
numeric [defaults p=30 s=6]	numeric(prec, scale)
float	if (prec) float(prec) else float
real	real
smallmoney	smallmoney
money	money
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	如果 (length > 255) 则为 text，否则 varchar(length)
character(n)	char(n)

Adaptive Server Anywhere 数据类型	Microsoft SQL Server 缺省数据类型
varchar(n)	如果 (length > 255) 则为 text，否则 varchar(length)
long varchar	text
binary(n)	if (length > 255) image else binary(length)
long binary	image
double	float
uniqueidentifierstr	uniqueidentifier

SQL Server uniqueidentifier 列会被映射到 Adaptive Server Anywhere 的 uniqueidentifierstr 列。结果字符串使用 STRTOUUID 函数可以转换为二进制 UUID 值。请参见 『ASA SQL 参考』> 「STRTOUUID 函数 [String]」。

服务器类 odbc

不具有自己的服务器类的 ODBC 数据源使用 **odbc** 服务器类。您可以使用符合 ODBC 2.0 版兼容性级别 1 或更高级别的任何 ODBC 驱动程序。Sybase 认证以下 ODBC 数据源可以正常使用：

- 「Microsoft Excel (Microsoft 3.51.171300)」第 694 页
- 「Microsoft Access (Microsoft 3.51.171300)」第 695 页
- 「Microsoft FoxPro (Microsoft 3.51.171300)」第 695 页
- 「Lotus Notes SQL 2.0」第 695 页

通过 Microsoft 数据访问组件 (MDAC) 发布 （位于 http://msdn.microsoft.com/data/downloads/default.aspx）可以获得 Microsoft ODBC 驱动程序的最新版本。下面列出的 Microsoft 驱动程序版本是 MDAC 2.0 的组成部分。

以下几节提供了有关访问这些数据源的说明。

Microsoft Excel (Microsoft 3.51.171300)

对于 Excel，每一个 Excel 工作簿在逻辑上都可以被视为包含若干个表的数据库。数据库中的表可以映射到 Excel 工作簿中的工作表。当您在 ODBC 驱动程序管理器中配置 ODBC 数据源名时，可以指定与该数据源相关联的缺省工作簿名称。不过，当您发出 CREATE TABLE 语句时，可以替换该缺省值并在位置字符串中指定工作簿名称。这样，您就可以只使用一个 ODBC DSN 来访问所有的 Excel 工作簿。

在这个示例中，创建了名为 **excel** 的 ODBC 数据源。若要创建具有名为 mywork 的工作表 （表）的名为 *work1.xls* 的工作簿，请执行以下语句：

```
CREATE TABLE mywork (a int, b char(20))
 AT 'excel;d:\work1.xls;;mywork'
```

若要创建第二个工作表 （表），请执行以下语句 （举例）：

```
CREATE TABLE mywork2 (x float, y int)
 AT 'excel;d:\work1.xls;;mywork2'
```

您可以使用 CREATE EXISTING 将现有的工作簿导入到 Adaptive Server Anywhere 中 （假定电子表格的第一行包含列名称）。

```
CREATE EXISTING TABLE mywork
 AT'excel;d:\work1;;mywork'
```

如果 Adaptive Server Anywhere 报告说该表未找到，则最好显式指出您希望映射到的列和行的范围。例如：

```
CREATE EXISTING TABLE mywork
 AT 'excel;d:\work1;;mywork$'
```

如果将 $ 添加到工作表名称中，则说明应选中整个工作表。

请注意，在 AT 指定的位置字符串中，应使用分号作为字段分隔符，而不是使用句点。这是因为句点出现在文件名中。 Excel 不支持所有者名称字段，因此不要填写该字段。

Excel 也不支持删除操作。此外，由于 Excel 驱动程序不支持定位更新，因此，可能无法进行某些更新。

Microsoft Access (Microsoft 3.51.171300)

Access 数据库存储于 *.mdb* 文件中。使用 ODBC 管理器创建一个 ODBC 数据源，并将该数据源映射到一个这类文件。通过 ODBC 管理器可以创建新的 *.mdb* 文件。如果您在通过 Adaptive Server Anywhere 创建表时您没有指定其它缺省文件，则上述数据库文件将成为缺省文件。

假定有一个名为 access 的 ODBC 数据源。

```
CREATE TABLE tab1 (a int, b char(10))
 AT 'access...tab1'
```

或者

```
CREATE TABLE tab1 (a int, b char(10))
 AT 'access;d:\pcdb\data.mdb;;tab1'
```

或者

```
CREATE EXISTING TABLE tab1
 AT 'access;d:\pcdb\data.mdb;;tab1'
```

Access 不支持所有者名称限定，将其保留为空。

Microsoft FoxPro (Microsoft 3.51.171300)

您可以将多个 FoxPro 表一起存储在一个 FoxPro 数据库文件 (.dbc) 内；或者，可以将每个表存储在各个单独的 .dbf 文件中。在使用 .dbf 文件时，请确保将文件名填充到位置字符串中；否则，将使用 Adaptive Server Anywhere 的启动目录。

```
CREATE TABLE fox1 (a int, b char(20))
 AT 'foxpro;d:\pcdb;;fox1'
```

如果您在 ODBC 驱动程序管理器中选择了 [自由表目录] 选项，则此语句将创建一个名为 *d:\pcdb\fox1.dbf* 的文件。

Lotus Notes SQL 2.0

您可以从 Lotus Web 站点 获得此驱动程序 （2.04.0203 版）。请阅读附随的文档，以便了解有关如何将 Notes 数据映射到关系表的说明。您可以很轻松地将 Adaptive Server Anywhere 表映射到 Notes 表单。

下面将介绍如何设置 Adaptive Server Anywhere 来访问 Address 示例文件。

- 使用 NotesSQL 驱动程序创建 ODBC 数据源。数据库将是示例名称文件 *c:\notes\data\names.nsf*。应选中 [映射特殊字符] 选项。对于此示例， [数据源名] 是 *my_notes_dsn*。
- 在 Adaptive Server Anywhere 中创建服务器：

```
CREATE SERVER names
 CLASS 'odbc'
 USING 'my_notes_dsn'
```

- 将 Person 表单映射到 Adaptive Server Anywhere 表中：

```
CREATE EXISTING TABLE Person
 AT 'names...Person'
```

- 查询该表

```
SELECT * FROM Person
```

避免出现要求输入口令的提示

Lotus Notes 不支持通过 ODBC API 发送用户名和口令。如果您尝试使用有口令保护的 ID 来访问 Lotus Notes，则在运行 Adaptive Server Anywhere 的机器上将出现一个窗口，提示您输入口令。在多用户服务器环境中应避免此行为。

若要以无人照管的方式访问 Lotus Notes，而不会收到要求输入口令的提示，则必须使用不带口令保护的 ID。可以通过清除口令保护 （[文件] → [工具] → [用户 ID] → [清除口令]）来取消对您的 ID 的口令保护，除非在创建 ID 时您的 Domino 管理员要求输入口令。在这一情况下，您将无法清除口令。

第 6 部分　存储过程和触发器

这一部分介绍了如何使用 SQL 存储过程和触发器在数据库中生成逻辑。在数据库中存储逻辑会使该逻辑可以自动供所有应用程序使用，从而带来一致性、性能和安全性等方面的益处。存储过程调试程序是用于调试各种类型逻辑的功能强大的工具。

第 20 章

使用过程、触发器和批处理

关于本章

过程和触发器在数据库中存储过程 SQL 语句，以供所有应用程序使用。它们增强了数据库的安全性，提高了效率并促进了标准化。用户定义的函数是一种类型的过程，它们将值返回到调用环境中，以供查询和其它 SQL 语句使用。批处理是作为一组提交到数据库服务器的 SQL 语句的集合。在过程和触发器中提供的许多功能 （例如控制语句）在批处理中也提供。

出于多种目的，服务器端 JDBC 提供比 SQL 存储过程更灵活的方法来将逻辑内置于数据库中。有关 JDBC 的信息，请参见 『ASA 编程指南』> 「JDBC 编程」。

过程和触发器概述

过程和触发器在数据库中存储过程 SQL 语句，以供所有应用程序使用。它们包括允许 SQL 语句的重复执行 （LOOP 语句）和条件执行 （IF 语句和 CASE 语句）的控制语句。

过程是通过 CALL 语句调用的，并且使用参数来接受值和将值返回到调用环境中。通过将过程名包括在 FROM 子句中， SELECT 语句也可以对过程结果集进行操作。

过程可以将结果集返回给调用者、调用其它过程或触发触发器。例如，用户定义的函数是一种将单个值返回到调用环境的存储过程。用户定义的函数不修改传递给它们的参数，而是拓展可用于查询和其它 SQL 语句的函数的范围。

触发器与特定数据库表相关联。只要有人插入、更新或删除关联表的行，触发器就会自动触发。触发器可以调用过程和触发其它触发器，但它们不具有任何参数并且无法由 CALL 语句调用。

过程调试程序

您可以使用数据库对象调试程序调试存储过程和触发器。有关详细信息，请参见 「调试数据库中的逻辑」第 767 页。

您可以在 Sybase Central 中分析存储过程，以便分析性能特性。有关详细信息，请参见 「分析数据库过程」第 213 页。

过程和触发器的优点

过程和触发器的定义在数据库中提供，与任何一个数据库应用程序相分离。这一分离具有许多优点。

标准化

过程和触发器使多个应用程序执行的操作标准化。通过对操作进行一次编码并将编码存储于数据库中以供将来使用，应用程序只需调用过程或触发触发器即可反复获得预期结果。此外，因为更改只需在一个地点进行，所以，如果操作的实现方式发生更改，则使用该操作的所有应用程序都可以自动获取新功能。

高效

在网络数据库服务器环境中使用的过程和触发器无需通过网络通信即可访问数据库中的数据。这意味着，与在某一客户机的应用程序中执行相比，过程和触发器在网络数据库服务器环境中执行的速度更快，并且对网络性能的影响较小。

当您创建过程或触发器时，系统自动检查过程或触发器的语法是否正确，然后将它们存储在系统表中。任何应用程序第一次调用过程或触发触发器时，将过程或触发器从系统表编译到服务器的虚拟内存中，然后从那里执行它们。由于过程或触发器的一个副本在第一次执行后将保留在内存中，因此，可以立即重复执行同一过程或触发器。此外，多个应用程序可以同时使用某一过程或触发器，或者一个应用程序可以循环使用它们。

如果过程包含简单查询并具有许多参数，则效率相对较低。对于复杂查询，过程的效率会更高。

安全性

通过授予用户对表中他们不能直接检查或修改的数据具有一定限制的访问权限，过程和触发器提供了安全性。

例如，触发器根据关联表的所有者的表权限执行，但具有插入、更新或删除该表中各行的权限的任何用户都可以触发这些触发器。同样，过程（包括用户定义的函数）根据过程所有者的权限执行，但授予权限的任何用户都可以调用这些过程。这意味着过程和触发器可以具有 （并通常具有）与调用它们的用户 ID 不同的权限。

过程简介

若要使用过程，您需要理解如何执行以下操作：

- 创建过程
- 从数据库应用程序调用过程
- 删除过程
- 控制谁具有使用过程的权限

本节讨论上述与使用过程有关的方面以及过程的一些不同应用。

创建过程

Adaptive Server Anywhere 提供大量工具，可用于创建新过程。

在 Sybase Central 中，您可以使用向导提供必要的信息。[创建过程] 向导还提供使用过程模板的选项。

此外，可以使用 CREATE PROCEDURE 语句创建过程。但是，您必须具有 RESOURCE 权限。您在哪里输入语句取决于所使用的工具。

❖ 创建新过程 (Sybase Central)：

1. 以具有 DBA 或资源权限的用户身份连接到数据库。
2. 打开数据库的 [过程和函数] 文件夹。
3. 从 [文件] 菜单中，选择 [新建] → [过程]。

   此时，就会出现 [创建过程] 向导。
4. 请按照向导中的说明进行操作。
5. 在完成此向导后，您可以在右窗格中的 [SQL] 选项卡上填写过程代码。

   新过程出现在 [过程和函数] 文件夹中。

❖ 创建新的远程过程 (Sybase Central)

1　以具有 DBA 权限的用户身份连接到数据库。

2　打开数据库的 [过程和函数] 文件夹。

3　从 [文件] 菜单中，选择 [新建] → [远程过程]。

即会出现 [远程过程创建] 向导。

4　请按照向导中的说明进行操作。

5　在完成此向导后，您可以在右窗格中的 [SQL] 选项卡上填写代码。

新的远程过程出现在 [过程和函数] 文件夹中。

❖ 使用不同工具创建过程：

- 遵循与您的工具有关的说明。在输入 CREATE PROCEDURE 语句前，您可能需要更改命令分隔符，使其不是分号。

有关连接的详细信息，请参见 『ASA 数据库管理指南』> 「连接到数据库」。

示例

以下简单示例创建过程 new_dept，该过程执行 INSERT 到示例数据库的 department 表的操作，从而创建新的部门。

```
CREATE PROCEDURE new_dept (
    IN id INT,
    IN name CHAR(35),
    IN head_id INT )
 BEGIN
    INSERT
    INTO DBA.department ( dept_id,
        dept_name, dept_head_id )
    VALUES ( id, name, head_id );
 END
```

过程的主体是复合语句。复合语句以 BEGIN 语句起始，以 END 语句结束。在 new_dept 的例子中，复合语句是包括在 BEGIN 和 END 语句之间的单个 INSERT。

过程的参数可以标记为 IN、 OUT 或 INOUT 之一。缺省情况下，参数是 INOUT 参数。 new_dept 过程的所有参数都是 IN 参数，因为该过程并不更改它们。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE PROCEDURE 语句」、『ASA SQL 参考』> 「ALTER PROCEDURE 语句」和 「使用复合语句」第 725 页。

变更过程

您可以使用 Sybase Central 或 Interactive SQL 修改现有过程。您必须具有 DBA 权限或是过程的所有者。

在 Sybase Central 中，您不能直接重命名现有过程，而是必须用新名称创建新过程，将以前的代码复制到这一新过程上，然后删除旧过程。

此外，可以使用 ALTER PROCEDURE 语句修改现有过程。您必须在此语句中包括新过程的全部内容 （采用与创建该过程的 CREATE PROCEDURE 语句相同的语法），还必须对该过程重新指派用户权限。

有关变更数据库对象属性的详细信息，请参见 「设置数据库对象的属性」第 38 页。

有关授予或撤消过程权限的详细信息，请参见 『ASA 数据库管理指南』> 「授予针对过程的权限」和 『ASA 数据库管理指南』> 「撤消用户权限」。

❖ 变更过程的代码 (Sybase Central)：

1. 打开 [过程和函数] 文件夹。
2. 选择所需的过程。然后，您可以执行以下操作之一：
   - 在右窗格中的 SQL 选项卡上直接编辑代码。
   - 在编辑代码前，将代码转换为 Watcom-SQL 或 Transact-SQL：
     - 右击所需的过程，然后从弹出式菜单中选择 [作为 Watcom-SQL 打开] 或 [作为 Transact-SQL 打开]。

- 在右窗格中的 SQL 选项卡上编辑代码。
- 通过在右窗格中右击该过程，在单独的窗口中编辑代码，然后从弹出式菜单中选择 [在 ‘ 新建 ’ 窗口中编辑]。

提示
如果您希望在过程之间复制代码，可为每个过程单独打开一个窗口。

❖ **变更过程的代码 (SQL)：**

1 连接到数据库。

2 执行 ALTER PROCEDURE 语句。在此语句中包括全新的过程。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER PROCEDURE 语句」、『ASA SQL 参考』> 「CREATE PROCEDURE 语句」和 「创建过程」第 702 页。

调用过程

CALL 语句调用过程。过程可由应用程序调用，或者由其它过程和触发器调用。

有关详细信息，请参见 『ASA SQL 参考』> 「CALL 语句」。

以下语句调用 new_dept 过程来插入 Eastern Sales 部门：

```
CALL new_dept( 210, 'Eastern Sales', 902 );
```

在此调用后，您最好检查 department 表，以查看是否已添加该新部门。

已被授予该过程的 EXECUTE 权限的所有用户都可以调用 new_dept 过程，即使他们对 department 表不具有任何权限。

有关 EXECUTE 权限的详细信息，请参见 『ASA SQL 参考』> 「EXECUTE 语句 [ESQL]」。

调用返回结果集的过程的另一种方法是在查询中调用它。您可以对过程的结果集执行查询，并应用 WHERE 子句和其它 SELECT 功能以限制结果集。

```
SELECT t.id, t.quantity_ordered AS q
 FROM sp_customer_products( 149 ) t
```

有关详细信息，请参见 『ASA SQL 参考』> 「FROM 子句」。

在 Sybase Central 中复制过程

在 Sybase Central 中，您可以在数据库间复制过程。若要进行这样的复制，请在 Sybase Central 的左窗格中选择过程，然后将它拖到另一连接的数据库的 [过程和函数] 文件夹中。这样就创建了一个新的过程，且原过程的代码复制给它。

请注意，只有过程代码被复制到该新过程中，不复制其它过程属性（权限等）。过程可被复制到同一数据库中，条件是为它提供了新名称。

删除过程

在创建了过程后，该过程将一直保留在数据库中，直到有人显式删除它。只有该过程的所有者或具有 DBA 权限的用户才可以从数据库中删除该过程。

❖ **删除过程 (Sybase Central)：**

1 作为具有 DBA 权限的用户或作为过程的所有者连接到数据库。

2 打开 [过程和函数] 文件夹。

3 右击所需过程并从弹出式菜单中选择 [删除]。

❖ **删除过程 (SQL)：**

1 作为具有 DBA 权限的用户或作为过程的所有者连接到数据库。

2　执行 DROP PROCEDURE 语句。

示例　以下语句从数据库中删除过程 new_dept：

```
DROP PROCEDURE new_dept
```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

在参数中返回过程结果

过程通过以下方式之一将结果返回到调用环境：

- 单独的值作为 OUT 或 INOUT 参数返回。
- 结果集可被返回。
- 可以使用 RETURN 语句返回单个结果。

本节描述如何从过程中将结果作为参数返回。

以下示例数据库上的过程将雇员的平均薪水作为 OUT 参数返回。

```
CREATE PROCEDURE AverageSalary( OUT avgsal
    NUMERIC (20,3) )
 BEGIN
    SELECT AVG( salary )
    INTO avgsal
    FROM employee;
 END
```

❖ 运行此过程并显示其输出 (SQL)：

1　用 **DBA** 用户 ID 和 **SQL** 口令从 Interactive SQL 连接到示例数据库。有关连接的详细信息，请参见 『ASA 数据库管理指南』> 「连接到数据库」。

2　在 [SQL 语句] 窗格中，键入以上过程代码。

3　创建保留过程输出的变量。在此例子中，输出变量是数值，具有三位小数，因此按如下所示创建变量：

```
CREATE VARIABLE Average NUMERIC(20,3)
```

4 使用所创建的用来保存结果的变量调用过程：

```
CALL AverageSalary(Average)
```

如果已创建过程并且该过程正确运行，则 Interactive SQL [消息] 窗格不显示任何错误。

5 执行 SELECT Average 语句以检查该变量的值。

查看输出变量 Average 的值。 [结果] 窗格的 [结果] 选项卡显示此变量的值 49988.623，这是雇员的平均薪水。

在结果集中返回过程结果

除了在单独参数中将结果返回到调用环境中之外，过程可以在结果集中返回信息。结果集通常是查询的结果。以下过程返回包含指定部门中每个雇员的薪水的结果集：

```
CREATE PROCEDURE SalaryList ( IN department_id INT)
 RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
 BEGIN
      SELECT emp_id, salary
      FROM employee
      WHERE employee.dept_id = department_id;
 END
```

如果 Interactive SQL 调用此过程，则将 RESULT 子句中的姓名与查询的结果进行匹配，并且 RESULT 子句中的姓名在显示的结果中用作列标题。

若要从 Interactive SQL 测试此过程，您可以 CALL （调用）该过程，指定公司的某一部门。在 Interactive SQL 中，结果出现在 [结果] 窗格的 [结果] 选项卡中。

示例

若要列出 R & D 部门 （部门 ID 100）中雇员的薪水，请键入以下内容：

```
CALL SalaryList (100)
```

Employee ID	Salary
102	45700

Employee ID	Salary
105	62000
160	57490
243	72995
…	…

如果您在 Interactive SQL 的 [选项] 对话框的 [结果] 选项卡上启用了返回多个结果集选项，才能返回多个结果集。每个结果集在 [结果] 窗格中用单独的选项卡显示。

有关详细信息，请参见 「从过程返回多个结果集」第 735 页。

用户定义的函数简介

用户定义的函数是一种过程，此类函数将单个值返回到调用环境中。Adaptive Server Anywhere 将所有用户定义的函数视为**等幂**，除非将它们声明为 NOT DETERMINISTIC。

等幂函数为相同的参数返回一致的结果，并且没有副作用。对具有相同参数的等幂函数的两个连续调用会返回相同的结果，并且对查询的语义不会导致任何不需要的副作用。

有关非确定性函数和确定性函数的详细信息，请参见 「函数高速缓存」第 480 页。

本节介绍如何创建、使用和删除用户定义的函数。

创建用户定义的函数

您使用 CREATE FUNCTION 语句创建用户定义的函数。但是，您必须具有 RESOURCE 权限。

以下简单示例创建一个函数，该函数连接两个字符串 （用空格连在一起）以根据名字和姓氏建立全名。

```
CREATE FUNCTION fullname (firstname CHAR(30),
     lastname CHAR(30))
 RETURNS CHAR(61)
 BEGIN
       DECLARE name CHAR(61);
       SET name = firstname || ' ' || lastname;
       RETURN ( name );
 END
```

❖ **使用 Interactive SQL 创建此示例：**

1. 用 **DBA** 用户 ID 和 **SQL** 口令从 Interactive SQL 连接到示例数据库。有关连接的详细信息，请参见 『ASA 数据库管理指南』> 「连接到数据库」。
2. 在 [SQL 语句] 窗格中，键入上述函数代码。

注意
如果您使用的工具并非 Interactive SQL 或 Sybase Central，则可能需要将命令分隔符更改为非分号的其它符号。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE FUNCTION 语句」。

CREATE FUNCTION 语法与 CREATE PROCEDURE 语句的语法稍有不同。存在以下方面的差异：

- 不需要 IN、 OUT 或 INOUT 关键字，因为所有参数都是 IN 参数。
- 需要 RETURNS 子句来指定所返回的数据类型。
- 需要 RETURN 语句来指定所返回的值。

调用用户定义的函数

在您要使用内置的非集合函数的任何地方，都可以根据权限使用用户定义的函数。

以下语句在 Interactive SQL 中从包含名字和姓氏的两列返回全名：

```
SELECT fullname (emp_fname, emp_lname)
 FROM employee;
```

fullname (emp_fname, emp_lname)
Fran Whitney
Matthew Cobb
Philip Chin
…

以下语句在 Interactive SQL 中从提供的名字和姓氏返回全名：

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

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

已被授予函数的 EXECUTE 权限的任何用户都可以使用 fullname 函数。

示例

以下用户定义的函数阐释变量的局部声明。

customer 表包含一些加拿大客户，这些客户散列在美国客户之中，但没有 country 列。美国邮区代码是数字，而加拿大邮政编码以字母开头，用户定义的 nationality 函数正是利用这一点来区分加拿大客户和美国客户。

```
CREATE FUNCTION nationality( cust_id INT )
 RETURNS CHAR( 20 )
 BEGIN
     DECLARE natl CHAR(20);
     IF cust_id IN ( SELECT id FROM customer
     WHERE LEFT(zip,1) > '9') THEN
              SET natl = 'CDN';
     ELSE
           SET natl = 'USA';
     END IF;
     RETURN ( natl );
 END
```

该示例声明变量 natl 来保存国籍字符串，使用 SET 语句来设置该变量的值，并且将 natl 字符串的值返回到调用环境。

以下查询列出 customer 表中的所有加拿大客户：

```
SELECT *
 FROM customer
 WHERE nationality(id) = 'CDN'
```

在以后几节中讨论游标和异常的声明。

不使用函数下述查询将可以更好地执行，尤其是在邮区代码上存在索引的情况下。例如，

```
Select *
 FROM customer
 WHERE zip > '99999'
```

注意　尽管此函数适合于概念阐述，但如果用于涉及多行的 SELECT 时，其执行效果会非常差。例如，如果您对包含 100 000 行的表使用该 SELECT 查询，返回其中的 10 000 行，则函数将被调用 10 000 次。如果您在同一查询的 WHERE 子句使用它，则该函数会被调用 100 000 次。

删除用户定义的函数

一旦您创建了用户定义的函数后，它就保留在数据库中，直到有人显式删除它。只有该函数的所有者或具有 DBA 权限的用户才可从数据库中删除函数。

以下语句从数据库中删除 fullname 函数：

```
DROP FUNCTION fullname
```

执行用户定义的函数的权限

用户定义的函数的所有权属于创建它的用户，因此该用户无需权限即可执行它。用户定义的函数的所有者可以使用 GRANT EXECUTE 命令向其它用户授予权限。

例如，函数 fullname 的创建者可以通过以下语句允许 another_user 使用 fullname：

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

以下语句撤消使用该函数的权限：

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

有关管理用户对函数的权限的详细信息，请参见 『ASA 数据库管理指南』> 「授予针对过程的权限」。

触发器简介

触发器是特殊形式的存储过程，在执行修改数据的查询时自动执行。只要参照完整性和其它声明性约束不足够，就可以使用触发器。

有关参照完整性的详细信息，请参见 「确保数据完整性」第 85 页和『ASA SQL 参考』> 「CREATE TABLE 语句」。

您可以实现更详细的检查及更复杂的参照完整性；或者可以对新数据实施检查，但允许旧数据将违反约束。触发器的另一用途是记录与数据库表有关的活动，而与使用数据库的应用程序无关。

触发器执行权限
触发器通过关联表的所有者的权限执行，而不是通过其操作导致触发器触发的用户 ID 的权限执行。触发器可以修改用户无法直接修改的表中各行。

触发器事件

可以对以下一种或多种触发事件定义触发器：

操作	说明
INSERT	只要有新行插入到与触发器关联的表中，就调用触发器。
DELETE	只要删除了关联表中的行，就调用触发器。
UPDATE	只要更新了关联表中的行，就调用触发器。
UPDATE OF column-list	只要更新了关联表中的行，且修改了 *column-list* 中的列，就调用触发器。

您可以为需要处理的每个事件分别编写触发器，或者，如果您有一些共享操作以及一些取决于事件的操作，您可以为所有事件创建触发器并使用 IF 语句区分所发生的操作。

有关详细信息，请参见 『ASA SQL 参考』> 「触发器操作条件」。

触发时间

触发器可以是**行级别**或**语句级别**：

- 行级别触发器为更改的每一行执行一次。行级别触发器在更改行之前或之后执行。

- 语句级别触发器在整个触发语句完成后执行。

对于在执行时依赖参照完整性操作 （例如在操作正执行或未执行的级联更新或删除）的触发器，其执行时间灵活的特点十分有用。

如果在触发器执行时发生错误，则触发该触发器的操作将失败。INSERT、 UPDATE 和 DELETE 是原子操作 （请参见 「原子复合语句」第 726 页）。在这些操作失败时，该语句的所有结果 （包括触发器的结果以及这些触发器调用的任何过程的结果）都恢复为其操作前状态。

有关触发器语法的完整说明，请参见 『ASA SQL 参考』> 「CREATE TRIGGER 语句」。

创建触发器

您可以使用 Sybase Central 或 Interactive SQL 创建触发器。在 Sybase Central 中，您可以使用向导提供必要的信息。在 Interactive SQL 中，可以使用 CREATE TRIGGER 语句。对于这两个工具，必须具有 DBA 或 RESOURCE 权限才能够创建触发器，并且必须对与触发器关联的表具有 ALTER 权限。

触发器的主体由复合语句组成：复合语句是用 BEGIN 和 END 语句括起来的分号分隔的 SQL 语句的集合。

不能在触发器内使用 COMMIT 和 ROLLBACK 以及某些 ROLLBACK TO SAVEPOINT 语句。

有关详细信息，请参见本节末尾的交叉引用列表。

❖ 为给定表创建触发器 (Sybase Central)：

1. 打开所需表的 [触发器] 文件夹。
2. 从 [文件] 菜单中，选择 [新建] → [触发器]。

   此时，就会出现 [创建触发器] 向导。
3. 请按照向导中的说明进行操作。

4 在完成向导后，您可以在右窗格中的 [SQL] 选项卡上填写触发器的代码。

❖ 为给定表创建触发器 (SQL)：

1 连接到数据库。

2 执行 CREATE TRIGGER 语句。

示例 1：行级别 INSERT 触发器

以下触发器是行级别 INSERT 触发器的示例。它检查为新雇员输入的出生日期是否合理：

```
CREATE TRIGGER check_birth_date
      AFTER INSERT ON Employee
 REFERENCING NEW AS new_employee
 FOR EACH ROW
 BEGIN
       DECLARE err_user_error EXCEPTION
       FOR SQLSTATE '99999';
       IF new_employee.birth_date > 'June 6, 2001' THEN
             SIGNAL err_user_error;
       END IF;
 END
```

此触发器在将任何行插入 employee 表之后触发。它检测并且不允许对应于出生日期晚于 2001 年 6 月 6 日的任何新行。

短语 REFERENCING NEW AS new_employee 使用别名 new_employee 以允许触发器代码中的语句引用新行中的数据。

发出错误将导致撤消触发语句以及该触发器之前执行的任何操作的结果。

对于将许多行添加到 employee 表的 INSERT 语句， check_birth_date 触发器为每一新行触发一次。如果该触发器对任何行触发失败，则 INSERT 语句的所有结果都回退。

您可以通过将示例的第一行更改为以下内容，指定触发器在插入行之前而非插入行之后触发：

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

REFERENCING NEW 子句引用该行的插入的值；它与触发器的计时（之前或之后）无关。

您会发现，在某些情况下，使用声明参照完整性或 CHECK 约束 （而不是触发器）实施约束会更容易。例如，使用列检查约束实施上例已证实效率更高并且更简明：

```
CHECK (@col <= 'June 6, 2001')
```

示例 2：行级别 DELETE 触发器示例

以下 CREATE TRIGGER 语句定义行级别 DELETE 触发器：

```
CREATE TRIGGER mytrigger BEFORE DELETE ON employee
 REFERENCING OLD AS oldtable
 FOR EACH ROW
 BEGIN
       ...
 END
```

REFERENCING OLD 子句使用别名 oldtable 使删除触发器代码能够引用所删除的行中的值。

您可以通过将示例的第一行更改为以下内容，指定触发器在删除行之后而非删除行之前触发：

```
CREATE TRIGGER check_birth_date AFTER DELETE ON
employee
```

REFERENCING OLD 子句与触发器的计时 （之前或之后）无关。

示例 3：语句级别 UPDATE 触发器示例

以下 CREATE TRIGGER 语句适合于语句级别 UPDATE 触发器：

```
CREATE TRIGGER mytrigger AFTER UPDATE ON employee
 REFERENCING NEW AS table_after_update
                 OLD AS table_before_update
 FOR EACH STATEMENT
 BEGIN
       ...
 END
```

REFERENCING NEW 和 REFERENCING OLD 子句允许 UPDATE 触发器代码引用所更新的行的旧值和新值。表别名 table_after_update 引用新行中的列，表别名 table_before_update 引用旧行中的列。

对于语句级别触发器和行级别触发器， REFERENCING NEW 和 REFERENCING OLD 子句在意义上稍有不同。对于语句级别触发器，REFERENCING OLD 或 NEW 别名是表别名，而在行级别触发器中，它们引用所变更的行。

有关详细信息，请参见 『ASA SQL 参考』> 「CREATE TRIGGER 语句」和 「使用复合语句」第 725 页。

执行触发器

只要对触发器中指定的表执行了 INSERT、 UPDATE 或 DELETE 操作，触发器就会自动执行。行级别触发器为受影响的每一行触发一次，而语句级别触发器则为整个语句触发一次。

在 INSERT、 UPDATE 或 DELETE 触发触发器时，操作的顺序如下所示：

1. 在触发器触发前。
2. 执行参照操作。
3. 执行操作本身。
4. 在触发器触发后。

如果其中任何步骤遇到在过程或触发器内未处理的错误，则前面的步骤被撤消，不执行随后的步骤，触发触发器的操作将失败。

变更触发器

您可以使用 Sybase Central 或 Interactive SQL 修改现有触发器。您必须是对其定义触发器的表的所有者或必须是 DBA，或者必须对该表具有 ALTER 权限并具有 RESOURCE 权限。

在 Sybase Central 中，不能直接重命名现有触发器，而是必须用新名称创建新的触发器，将以前的代码复制到这一新触发器上，然后删除旧触发器。

或者，您可以使用 ALTER TRIGGER 语句修改现有触发器。您必须在此语句中包括新触发器的全部内容 （采用与创建该触发器的 CREATE TRIGGER 语句的相同语法）。

有关变更数据库对象属性的详细信息，请参见 「设置数据库对象的属性」第 38 页。

❖ 变更触发器的代码 (Sybase Central)：

1. 打开 [触发器] 文件夹。
2. 选择所需的触发器。然后，您可以执行以下操作之一：
   - 在右窗格中的 SQL 选项卡上直接编辑代码。
   - 在编辑代码前，将代码转换为 Watcom-SQL 或 Transact-SQL：
     - 右击所需的触发器，然后从弹出式菜单中选择 [作为 Watcom-SQL 打开] 或 [作为 Transact-SQL 打开]。
     - 在右窗格中的 SQL 选项卡上编辑代码。
   - 通过在右窗格中右击该触发器，然后从弹出式菜单中选择 [在 ‘ 新建 ’ 窗口中编辑]，在单独的窗口中编辑代码。

提示
如果您希望在触发器之间复制代码，则可以为每个触发器打开单独的窗口。

❖ 变更触发器的代码 (SQL)：

1. 连接到数据库。
2. 执行 ALTER TRIGGER 语句。在此语句中包括全新的触发器。

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER TRIGGER 语句」。

删除触发器

一旦创建了触发器后，它将保留在数据库中，直到有人显式删除它。您必须对与触发器关联的表具有 ALTER 权限，才可以删除该触发器。

❖ **删除触发器 (Sybase Central)：**

1 打开 [触发器] 文件夹。

2 右击所需触发器并从弹出式菜单中选择 [删除]。

❖ **删除触发器 (SQL)：**

1 连接到数据库。

2 执行 DROP TRIGGER 语句。

示例

以下语句从数据库中删除 mytrigger 触发器：

```
DROP TRIGGER mytrigger
```

有关详细信息，请参见 『ASA SQL 参考』> 「DROP 语句」。

触发器执行权限

您不能授予执行触发器的权限，因为用户不能执行触发器：Adaptive Server Anywhere 触发它们以响应对数据库执行的操作。不过，触发器确实具有与它所执行的操作关联的权限，并且定义其权限以执行某些操作。

触发器使用定义了这些权限的表的所有者的权限 （而不是导致触发器触发的用户的权限，并且不是创建该触发器的用户的权限）执行。

在触发器引用表时，它使用表创建者的组成员资格找到没有指定的显式所有者名称的表。例如，如果 user_1.Table_A 上的触发器引用 Table_B，并且没有指定 Table_B 的所有者，那么， Table_B 就必须已经由 user_1 创建，或者， user_1 必须 （直接或间接地）是作为 Table_B 的所有者的某个组的成员。如果这两个条件都不具备，该触发器触发时将出现消息 [**没有找到表**]。

此外， user_1 必须具有执行该触发器中指定的操作的权限。

批处理简介

简单的批处理由一组 SQL 语句组成，这些语句用分号分隔，或是由单独的行分隔，行上只具有 **go** 一词。建议使用 **go**。例如，以下一组语句构成一个批处理，该批处理创建 Eastern Sales 部门并将所有销售代表从 Massachusetts 转移到该部门。

```
INSERT
 INTO department ( dept_id, dept_name )
 VALUES ( 220, 'Eastern Sales' )
 go
 UPDATE employee
 SET dept_id = 220
 WHERE dept_id = 200
 AND state = 'MA'
 go
 COMMIT
 go
```

您可以在应用程序中包括此组语句并一起执行它们。

Interactive SQL 和批处理

Interactive SQL 首先对一组分号分隔的语句 （例如上述语句）进行分析，然后将它们发送到服务器。在此情况下， Interactive SQL 将每一语句单独发送到服务器，而不是成批发送。如果在您的应用程序中没有此类分析代码，则应发送这些语句并它们视作批处理。用 BEGIN 和 END 将一组语句括起来可导致 Interactive SQL 将它们视作批处理。

过程和触发器中使用的许多语句也可以在批处理中使用。您可在批处理中使用控制语句 （CASE、 IF、 LOOP）等，包括复合语句 （BEGIN 和 END）。复合语句可以包括该复合语句内变量、异常、临时表或游标的声明。

只有在该名称的表已不存在时，以下批处理才创建表：

```
IF NOT EXISTS (
     SELECT * FROM SYSTABLE
     WHERE table_name = 't1' ) THEN
         CREATE TABLE t1 (
         firstcol INT PRIMARY KEY,
         secondcol CHAR( 30 )
```

```
        )
        go
ELSE
    MESSAGE 'Table t1 already exists' TO CLIENT;
END IF
```

如果从 Interactive SQL 运行此批处理两次，则在您第一次运行该批处理时它创建该表，并在下一次运行该批处理时在 Interactive SQL [消息] 窗格中显示消息。

控制语句

在过程或触发器的主体或在批处理中，有多种用于逻辑流和决策的控制语句。可用的控制语句包括：

控制语句	语法

复合语句
 	
有关详细信息，请参见『ASA SQL 参考』>「BEGIN 语句」。
	`BEGIN [ ATOMIC ]`
	`  Statement-list`
	`END`

条件执行：IF
 	
有关详细信息，请参见『ASA SQL 参考』>「IF 语句」。
	`IF condition THEN`
	`  Statement-list`
	`ELSEIF condition THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END IF`

条件执行：CASE
 	
有关详细信息，请参见『ASA SQL 参考』>「CASE 语句」。
	`CASE expression`
	`WHEN value THEN`
	`  Statement-list`
	`WHEN value THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END CASE`

重复：WHILE、 LOOP
 	
有关详细信息，请参见『ASA SQL 参考』>「LOOP 语句」。
	`WHILE condition LOOP`
	`  Statement-list`
	`END LOOP`

重复：FOR 游标循环
 	
有关详细信息，请参见『ASA SQL 参考』>「FOR 语句」。
	`FOR loop-name`
	`  AS cursor-name`
	`  CURSOR FOR select statement`
	`DO`
	`  Statement-list`
	`END FOR`

控制语句	语法
中断：LEAVE 有关详细信息，请参见『ASA SQL 参考』>「LEAVE 语句」。	`LEAVE label`
CALL 有关详细信息，请参见『ASA SQL 参考』>「CALL 语句」。	`CALL procname( arg, ... )`

有关每一语句的详细信息，请参见 『ASA SQL 参考』>「SQL 语句」中的各项。

使用复合语句

复合语句以关键字 BEGIN 起始并以关键字 END 结束。过程或触发器的主体是**复合语句**。复合语句还可以用于批处理。复合语句可被嵌套，并且可以与其它控制语句合并在一起，以定义过程和触发器中或批处理中的执行流。

复合语句允许一组 SQL 语句组合在一起并被视作一个单元。用分号在复合语句内分隔 SQL 语句。

有关复合语句的详细信息，请参见 『ASA SQL 参考』>「BEGIN 语句」。

复合语句中的声明

局部声明在复合语句中紧随在 BEGIN 关键字之后。这些局部声明只存在于复合语句内。在复合语句内，您可以声明：

- 变量
- 游标
- 临时表

- 异常 （错误标识符）

局部声明可由该复合语句中的任何语句引用，或者可由该复合语句内嵌套的任何复合语句中的任何语句引用。局部声明对于从复合语句调用的其它过程而言是不可见的。

原子复合语句

原子语句是完全执行或根本不执行的语句。例如，更新数千行的 UPDATE 语句在更新许多行后可能会遇到错误。如果该语句没有完成，则所有更改都恢复为其原始状态。 UPDATE 语句因此是原子语句。

所有非复合 SQL 语句都是原子语句。您可以通过在 BEGIN 关键字后添加 ATOMIC 关键字来使复合语句成为原子语句。

```
BEGIN ATOMIC
      UPDATE employee
      SET manager_ID = 501
      WHERE emp_ID = 467;
      UPDATE employee
      SET birth_date = 'bad_data';
 END
```

在此示例中，两个更新语句都属于原子复合语句。它们要么成功，要么失败，二者只能居其一。第一个更新语句将是成功的。第二个语句会导致数据转换错误，因为指派给 birth_date 列的值无法转换为日期。

该原子复合语句失败，并且两个 UPDATE 语句的结果被撤消。即使最终提交当前执行的事务，但原子复合语句中的这两个语句都不会生效。

如果原子复合语句成功，则在复合语句中进行的更改仅在提交当前执行的事务时才生效。如果原子复合语句成功而包含该语句的事务却回退，则原子复合语句也回退。在原子复合语句开始时建立一个保存点。语句中的任何错误都会导致回退到该保存点。

不能在原子复合语句中使用 COMMIT 和 ROLLBACK 以及某些 ROLLBACK TO SAVEPOINT 语句 （请参见 「过程和触发器中的事务和保存点」第 754 页）。

可能会有这样的情况：执行了原子复合语句中的某些语句，但不是全部语句。当复合语句中的异常处理程序处理错误时可能发生上述情况。

有关详细信息，请参见 「在过程和触发器中使用异常处理程序」第 747 页。

过程和触发器的结构

过程或触发器的主体由 「使用复合语句」第 725 页中讨论的复合语句组成。复合语句由 BEGIN 和 END 组成，它们将一组 SQL 语句括起来，用分号分隔每一语句。

在过程和触发器中允许的 SQL 语句

您可以在过程和触发器内使用几乎所有 SQL 语句，包括以下各种语句：

- SELECT、 UPDATE、 DELETE、 INSERT 和 SET VARIABLE。
- 用来执行其它过程的 CALL 语句。
- 控制语句 （请参见 「控制语句」第 724 页）。
- 游标语句 （请参见 「在过程和触发器中使用游标」第 738 页）。
- 异常处理语句 （请参见 「在过程和触发器中使用异常处理程序」第 747 页）。
- EXECUTE IMMEDIATE 语句。

某些您不能在过程和触发器中使用的 SQL 语句包括：

- CONNECT 语句
- DISCONNECT 语句。

可以在过程和触发器中使用 COMMIT、 ROLLBACK 和 SAVEPOINT 语句，但带有某些限制 （请参见 「过程和触发器中的事务和保存点」第 754 页）。

有关详细信息，请参见 『ASA SQL 参考』> 「SQL 语句」一章中每一 SQL 语句的用法。

为过程声明参数

过程参数在 CREATE PROCEDURE 语句中作为列表出现。参数名必须符合其它数据库标识符 （如列名）的规则。它们必须具有有效数据类型 （请参见 『ASA SQL 参考』> 「SQL 数据类型」），并且可以用关键字 IN、 OUT 或 INOUT 作为前缀。缺省情况下，参数是 INOUT 参数。这些关键字具有以下含义：

- **IN**　此参数是一个为过程提供值的表达式。
- **OUT**　此参数是一个可由过程为其赋值的变量。
- **INOUT**　此参数是一个为过程提供值的变量，并且可以由过程为其赋予新值。

可以在 CREATE PROCEDURE 语句中将缺省值指派给过程参数。缺省值必须是常量，可以是空值。例如，以下过程使用空值缺省值来用于 IN 参数，以避免执行没有任何意义的查询：

```
CREATE PROCEDURE
 CustomerProducts( IN customer_id
                        INTEGER DEFAULT NULL )
 RESULT ( product_id INTEGER,
             quantity_ordered INTEGER )
 BEGIN
      IF customer_id IS NULL THEN
           RETURN;
      ELSE
           SELECT     product.id,
                     sum( sales_order_items.quantity )
           FROM    product,
                      sales_order_items,
                      sales_order
           WHERE sales_order.cust_id = customer_id
           AND sales_order.id = sales_order_items.id
           AND sales_order_items.prod_id = product.id
           GROUP BY product.id;
      END IF;
 END
```

以下语句指派缺省的空值，并且过程返回而不是执行该查询。

```
CALL customerproducts();
```

将参数传递给过程

您可以通过 CALL 语句的两种形式之一利用存储过程参数的缺省值的优越性。

如果可选参数在 CREATE PROCEDURE 语句中位于参数列表的末尾，则可以从 CALL 语句忽略它们。作为示例，请考虑以下具有三个 INOUT 参数的过程：

```
CREATE PROCEDURE SampleProc( INOUT var1 INT
                                          DEFAULT 1,
                             INOUT var2 int DEFAULT 2,
                             INOUT var3 int DEFAULT 3 )
 ...
```

我们假设调用环境已设置了三个变量，用来保留传递给过程的值：

```
CREATE VARIABLE V1 INT;
 CREATE VARIABLE V2 INT;
 CREATE VARIABLE V3 INT;
```

可以调用过程 SampleProc，并且仅提供第一个参数，如下所示：

```
CALL SampleProc( V1 )
```

在这一情况中，缺省值用于 *var2* 和 *var3*

用可选参数调用过程的更灵活方法是按名称传递这些参数。可按如下所示调用 SampleProc 过程：

```
CALL SampleProc( var1 = V1, var3 = V3 )
```

或按如下所示调用该过程：

```
CALL SampleProc( var3 = V3, var1 = V1 )
```

将参数传递给函数

用户定义的函数不使用 CALL 语句调用，而是采用与内置函数同样的方式使用。例如，以下语句使用在 「创建用户定义的函数」第 710 页中定义的 fullname 函数来检索雇员的姓名：

❖ **列出所有雇员的姓名：**

- 键入以下语句：

```
SELECT fullname(emp_fname, emp_lname) AS Name
 FROM employee
```

Name
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
…

注意

- 在调用函数中可以使用缺省参数。但是，不能按名称将参数传递给函数。
- 参数是按值传递的，而不是按引用传递的。即使函数更改了参数的值，此更改也不会返回到调用环境。
- 不能在用户定义的函数中使用输出参数。
- 用户定义的函数不能返回结果集。

从过程返回结果

过程可以以单行数据或多行数据的形式返回结果。由单行数据组成的结果可作为参数传递回过程。由多行数据组成的结果作为结果集传递回过程。过程也可以返回在 RETURN 语句中给定的单个值。

有关如何从过程返回结果的简单示例，请参见 「过程简介」第 702 页。有关详细信息，请参见以下各节。

使用 RETURN 语句返回值

RETURN 语句将单个整数值返回到调用环境中，并且导致立即从过程退出。 RETURN 语句采用以下形式：

```
RETURN expression
```

提供的表达式的值被返回到调用环境中。为了在变量中保存该返回值，使用以下 CALL 语句的扩展：

```
CREATE VARIABLE returnval INTEGER ;
 returnval = CALL myproc() ;
```

作为过程参数返回结果

过程可以将结果作为作用于该过程的参数返回到调用环境中。

在过程内，可以采用以下方式为参数和变量指派值：

- SET 语句。
- 具有 INTO 子句的 SELECT 语句。

使用 SET 语句

以下过程 （看起来有些不够真实）使用 SET 语句将值返回到指派的 OUT 参数中：

```
CREATE PROCEDURE greater (    IN a INT,
                                  IN b INT,
                                  OUT c INT)
 BEGIN
     IF a > b THEN
```

```
        SET c = a;
    ELSE
        SET c = b;
    END IF ;
END
```

使用单行 SELECT 语句

单行查询从数据库中最多检索一行。此类型的查询将 SELECT 语句与 INTO 子句一起使用。 INTO 子句跟随在选择列表之后，位于 FROM 子句之前。它包含一组变量，用于为每一选择列表项检索值。变量的数目必须与选择列表项的数目相同。

在执行 SELECT 语句时，服务器检索 SELECT 语句的结果并将这些结果放置于变量中。如果查询结果包含多行，则服务器返回错误。对于返回多行的查询，必须使用游标。有关从过程返回多行的信息，请参见「从过程返回结果集」第 734 页。

如果查询导致未选择任何行，则显示 [**未 找到行**] 警告。

以下过程在过程参数中返回单行 SELECT 语句的结果。

❖ **返回给定客户下的订单的数目：**

- 键入以下语句：

```
CREATE PROCEDURE OrderCount (IN customer_ID INT,
                                        OUT Orders
INT)
 BEGIN
     SELECT COUNT(DBA.sales_order.id)
         INTO Orders
     FROM DBA.customer
         KEY LEFT OUTER JOIN "DBA".sales_order
     WHERE DBA.customer.id = customer_ID;
 END
```

您可以使用以下语句在 Interactive SQL 中测试此过程，它显示 ID 为 102 的客户下的订单的数目：

```
CREATE VARIABLE orders INT;
 CALL OrderCount ( 102, orders );
 SELECT orders;
```

注意

- *customer_ID* 参数被声明为 IN 参数。该参数保留传递到过程中的客户 ID。
- *Orders* 参数被声明为 OUT 参数。它保留返回到调用环境的订单变量的值。
- *Orders* 变量不需要 DECLARE 语句，因为它是在过程参数列表中声明的。
- SELECT 语句返回单行并将它放置于变量 *Orders* 中。

从过程返回结果集

结果集允许过程将多行结果返回到调用环境中。

以下过程返回已下订单的客户的列表，同时一起返回所下订单的总值。该过程不列出未下订单的客户。

```
CREATE PROCEDURE ListCustomerValue ()
 RESULT ("Company" CHAR(36), "Value" INT)
 BEGIN
     SELECT company_name,
         CAST( sum(    sales_order_items.quantity *
                         product.unit_price)
                         AS INTEGER ) AS value
     FROM customer
         INNER JOIN sales_order
         INNER JOIN sales_order_items
         INNER JOIN product
     GROUP BY company_name
     ORDER BY value DESC;
 END
```

- 键入以下语句：

```
CALL ListCustomerValue ()
```

Company	Value
Chadwicks	8076
Overland Army Navy	8064

Company	Value
Martins Landing	6888
Sterling & Co.	6804
Carmel Industries	6780
…	…

注意

- RESULT 列表中变量的数目必须与 SELECT 列表项的数目相匹配。如果数据类型不匹配，则如有可能，执行自动数据类型转换。
- RESULT 子句是 CREATE PROCEDURE 语句的一部分，并且不具有命令分隔符。
- SELECT 列表项的名称不必与 RESULT 列表中的那些名称相匹配。
- 在测试此过程时，缺省情况下 Interactive SQL 只显示第一个结果集。可以通过在 [选项] 对话框的 [结果] 选项卡上设置 [显示多个结果集] 选项，配置 Interactive SQL 显示多个结果集。
- 可以修改过程结果集，除非它们是从视图生成的。调用该过程的用户要求对基表具有适当的权限，以修改过程结果。这不同于通常执行过程的权限，它要求过程*所有者*必须对该表具有权限。

  有关在 Interactive SQL 中修改结果集的信息，请参见 『SQL Anywhere Studio 介绍』> 「在 Interactive SQL 中编辑表值」。
- 如果存储过程或用户定义的函数返回一个结果集，则不能同时设置输出参数或返回一个返回值。

从过程返回多个结果集

在 Interactive SQL 可以返回多个结果集之前，需要在 [选项] 对话框的 [结果] 选项卡上启用 [显示多个结果集] 选项。缺省情况下，此选项被禁用。如果更改该设置，它将在新创建的连接中 （例如新窗口）生效。

❖ **启用多个结果集功能：**

1 选择 [工具] → [选项]。

2 在出现的 [选项] 对话框中，单击 [结果] 选项卡。

3 选取 [显示多个结果集] 复选框。

在启用此选项后，过程可以将多个结果集返回到调用环境中。如果采用 RESULT 子句，则结果集必须是兼容的：它们必须在 SELECT 列表中具有相同数目的项，并且数据类型必须全都属于可以自动转换为 RESULT 列表中列出的数据类型的类型。

以下过程列出在数据库中列出的所有雇员、客户和联系人的姓名：

```
CREATE PROCEDURE ListPeople()
 RESULT ( lname CHAR(36), fname CHAR(36) )
 BEGIN
     SELECT emp_lname, emp_fname
     FROM employee;
     SELECT lname, fname
     FROM customer;
     SELECT last_name, first_name
     FROM contact;
 END
```

注意

- 若要在 Interactive SQL 中测试此过程，请在 [SQL 语句] 窗格中输入以下语句：

  ```
  CALL ListPeople ()
  ```

从过程返回可变结果集

RESULT 子句在过程中是可选的。通过省略 RESULT 子句，您可以编写返回不同结果集的过程，结果集中具有不同数目或类型的列，具体情况取决于执行它们的方式。

如果不使用可变结果集功能，则出于性能方面的原因，应使用 RESULT 子句。

例如，如果输入变量为 Y，以下过程返回两列，否则只返回一列。

```
CREATE PROCEDURE names( IN formal char(1))
 BEGIN
     IF formal = 'y' THEN
         SELECT emp_lname, emp_fname
         FROM employee
     ELSE
         SELECT emp_fname
         FROM employee
     END IF
 END
```

在过程中使用可变结果集受到某些限制的约束，具体限制取决于客户端应用程序使用的接口。

- **嵌入式 SQL**　您必须在打开用于结果集的游标后、但在返回任何行之前 DESCRIBE （描述）过程调用，从而获得正确形式的结果集。

  有关 DESCRIBE 语句的详细信息，请参见 『ASA SQL 参考』> 「DESCRIBE 语句 [ESQL]」。

- **ODBC**　可变结果集过程可由 ODBC 应用程序使用。 Adaptive Server Anywhere ODBC 驱动程序产生可变结果集的正确描述。

- **Open Client 应用程序**　Open Client 应用程序可以使用可变结果集过程。 Adaptive Server Anywhere 可以产生可变结果集的正确描述。

在过程和触发器中使用游标

游标在其结果集中从具有多行的查询或存储过程中一次检索一行。游标是查询或过程的句柄或标识符，也是结果集内的当前位置的句柄或标识符。

游标管理概述

管理游标类似于通过编程语言管理文件。以下步骤用于管理游标：

1. 使用 DECLARE 语句为特殊 SELECT 语句或过程声明游标。
2. 使用 OPEN 语句打开游标。
3. 使用 FETCH 语句从游标一次检索一行结果。
4. 警告 `Row Not Found` 指示结果集的结尾。
5. 使用 CLOSE 语句关闭游标。

缺省情况下，游标自动在事务的末尾关闭 （在 COMMIT 或 ROLLBACK 语句上）。使用 WITH HOLD 子句打开的游标将为随后的事务保持打开状态，直至有人显式关闭它们为止。

有关定位游标的详细信息，请参见 『ASA 编程指南』> 「游标定位」。

在过程中在 SELECT 语句上使用游标

以下过程在 SELECT 语句上使用游标。该过程基于 ListCustomerValue 过程 （在 「从过程返回结果集」第 734 页中描述了这一过程）中使用的同一查询，阐释了存储过程语言的若干功能。

```
CREATE PROCEDURE TopCustomerValue
     (    OUT TopCompany CHAR(36),
         OUT TopValue INT )
 BEGIN
     -- 1. Declare the "error not found" exception
     DECLARE err_notfound
         EXCEPTION FOR SQLSTATE '02000';
```

```
-- 2.    Declare variables to hold
    --        each company name and its value
    DECLARE ThisName CHAR(36);
    DECLARE ThisValue INT;

-- 3.    Declare the cursor ThisCompany
    --        for the query
    DECLARE ThisCompany CURSOR FOR
    SELECT company_name,
            CAST( sum( sales_order_items.quantity *
                    product.unit_price ) AS INTEGER )
            AS value
    FROM customer
        INNER JOIN sales_order
        INNER JOIN sales_order_items
        INNER JOIN product
    GROUP BY company_name;

-- 4. Initialize the values of TopValue
    SET TopValue = 0;
    -- 5. Open the cursor
    OPEN ThisCompany;

-- 6. Loop over the rows of the query
    CompanyLoop:
    LOOP
        FETCH NEXT ThisCompany
            INTO ThisName, ThisValue;
        IF SQLSTATE = err_notfound THEN
            LEAVE CompanyLoop;
        END IF;
        IF ThisValue > TopValue THEN
            SET TopCompany = ThisName;
            SET TopValue = ThisValue;
        END IF;
    END LOOP CompanyLoop;

-- 7. Close the cursor
    CLOSE ThisCompany;
END
```

注意　TopCustomerValue 过程具有以下显著的功能：

- 声明 "error not found" 异常。在完成了对查询结果的循环后，在过程的后期显示该异常。

有关异常的详细信息，请参见 「过程和触发器中的错误和警告」第 742 页。

- 声明两个局部变量 ThisName 和 ThisValue，以保存来自该查询的每一行的结果。
- 声明游标 ThisCompany。 SELECT 语句生成公司名称以及该公司所下订单的总值的列表。
- TopValue 的值被设置为初始值 0，以便以后在循环中使用。
- ThisCompany 游标打开。
- LOOP 语句对该查询的每一行进行循环，依次将每一公司名称放置于变量 ThisName 和 ThisValue 中。如果 ThisValue 大于当前的最高值，则将 TopCompany 和 TopValue 重置为 ThisName 和 ThisValue。
- 该游标在过程结束后关闭。
- 您还可以通过将 ORDER BY 值 DESC 子句添加到 SELECT 语句中，不借助循环编写此过程。然后，只需获取游标的第一行。

LOOP 构造在 TopCompanyValue 过程中是标准形式，在最后一行进行处理后退出。您可以使用 FOR 循环，以更紧凑的形式重新编写此过程。FOR 语句将上述过程的若干方面合并到单个语句中。

```
CREATE PROCEDURE TopCustomerValue2(
        OUT TopCompany CHAR(36),
        OUT TopValue INT )
 BEGIN
    -- Initialize the TopValue variable
    SET TopValue = 0;
    -- Do the For Loop
    FOR CompanyFor AS ThisCompany
        CURSOR FOR
        SELECT company_name AS ThisName ,
            CAST( sum( sales_order_items.quantity *
                    product.unit_price ) AS INTEGER )
            AS ThisValue
        FROM customer
            INNER JOIN sales_order
            INNER JOIN sales_order_items
            INNER JOIN product
```

```
        GROUP BY ThisName
    DO
        IF ThisValue > TopValue THEN
            SET TopCompany = ThisName;
            SET TopValue = ThisValue;
            END IF;
    END FOR;
END
```

过程和触发器中的错误和警告

在应用程序执行 SQL 语句后，它可以检查**状态码**。此状态码（即返回代码）指示执行的语句是成功还是失败，并给出失败的原因。您可以使用相同的机制指示对过程执行的 CALL 语句是成功还是失败。

错误报告使用 SQLCODE 或 SQLSTATE 状态说明。有关 SQLCODE 和 SQLSTATE 错误和警告值及其含义的完整说明，请参见 『ASA 错误消息』> 「ASA 错误消息」。只要执行 SQL 语句，就会在称作 SQLSTATE 和 SQLCODE 的特殊过程变量中出现值。该值指示在执行语句时是否遇到了任何意外情况。您可以在执行 SQL 语句后在 IF 语句中检查 SQLSTATE 或 SQLCODE 的值，并根据该语句成功与否采取措施。

例如， SQLSTATE 变量可用于指示是否成功获取某行。「在过程中在 SELECT 语句上使用游标」第 738 页一节中介绍的 TopCustomerValue 过程使用 SQLSTATE 测试来检测是否已处理了 SELECT 语句的所有行。

过程和触发器中缺省的错误处理

本节介绍如果您没有在过程中内置错误处理方式， Adaptive Server Anywhere 将如何处理执行过程期间发生的错误。

对于不同行为，可使用在 「在过程和触发器中使用异常处理程序」第 747 页中介绍的异常处理程序。处理警告的方式与处理错误的方式稍有不同：有关说明，请参见 「过程和触发器中警告的缺省处理」第 746 页。

有两种无需使用显式错误处理即可处理错误的方法：

- **缺省的错误处理** 过程或触发器失败，并且将错误代码返回到调用环境。
- **ON EXCEPTION RESUME** 如果 ON EXCEPTION RESUME 子句出现在 CREATE PROCEDURE 语句中，则过程将在出现错误后继续执行，在导致错误的语句之后的下一语句恢复。

使用 ON EXCEPTION RESUME 的过程的精确行为由 ON_TSQL_ERROR 选项设置指示。有关详细信息，请参见『ASA 数据库管理指南』> 「ON_TSQL_ERROR 选项 [compatibility]」。

缺省的错误处理

通常，通过对 SQLSTATE 值和 SQLCODE 值进行适当设置，如果过程或触发器中的 SQL 语句失败，则该过程或触发器将终止执行，并且相应的控制返回到应用程序。即使错误发生在从第一个过程或触发器直接或间接调用的过程或触发器中，上述情况也同样适用。对于触发器，导致该触发器触发的操作也将撤消，并且错误被返回到应用程序。

以下演示过程说明了在发生下面情况时的结果：应用程序调用过程 OuterProc， OuterProc 又调用过程 InnerProc，然后遇到错误。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT
 END
 CREATE PROCEDURE InnerProc()
     BEGIN
         DECLARE column_not_found
             EXCEPTION FOR SQLSTATE '52003';
         MESSAGE 'Hello from InnerProc.' TO CLIENT;
         SIGNAL column_not_found;
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE, ' in InnerProc.' TO CLIENT;
 END
```

注意

- InnerProc 中的 DECLARE 语句为与服务器已知的错误条件相关联的某一预定义的 SQLSTATE 值声明符号名称。
- MESSAGE 语句将消息发送到 Interactive SQL [消息] 窗格。
- SIGNAL 语句从 InnerProc 过程内生成错误条件。

以下语句执行 OuterProc 过程：

```
CALL OuterProc();
```

Interactive SQL [消息] 窗格显示以下内容：

```
Hello from OuterProc.

Hello from InnerProc.
```

InnerProc 中没有任何跟随在 SIGNAL 语句之后的语句执行：InnerProc 立即将控制传递回调用环境，在此例子中调用环境是 OuterProc。OuterProc 中没有任何跟随在 CALL 语句之后的语句执行。错误条件返回到调用环境中，并在那里进行处理。例如， Interactive SQL 通过显示描述该错误的消息窗口，然后处理该错误。

TRACEBACK 函数提供在发生错误时正执行的语句的列表。您可以通过键入以下语句，从 Interactive SQL 使用 TRACEBACK 函数：

```
SELECT TRACEBACK(*)
```

使用 ON EXCEPTION RESUME 处理错误

如果 ON EXCEPTION RESUME 子句出现在 CREATE PROCEDURE 语句中，则该过程在发生错误时检查随后的语句。如果该语句处理错误，则过程继续执行，在导致错误的语句之后的下一语句恢复。当发生错误时，它不将控制返回到调用环境。

可以通过 ON_TSQL_ERROR 选项设置修改使用 ON EXCEPTION RESUME 的过程的行为。有关详细信息，请参见 『ASA 数据库管理指南』> 「ON_TSQL_ERROR 选项 [compatibility]」。

错误处理语句包括以下这些：

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL

- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

以下示例阐释上述语句的工作方式。

删除过程

请记住，在继续本教程前，在 [SQL 语句] 窗格中输入以下命令来删除 InnerProc 和 OuterProc 过程：

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

以下演示过程说明了在发生下面情况时的结果：应用程序调用过程 OuterProc， OuterProc 又调用过程 InnerProc，然后遇到错误。这些演示过程基于在本节的前面部分中使用的那些过程：

```
CREATE PROCEDURE OuterProc()
 ON EXCEPTION RESUME
 BEGIN
     DECLARE res CHAR(5);
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     SELECT @res=SQLSTATE;
     IF res='52003' THEN
         MESSAGE 'SQLSTATE set to ',
             res, ' in OuterProc.' TO CLIENT;
     END IF
 END;

 CREATE PROCEDURE InnerProc()
 ON EXCEPTION RESUME
 BEGIN
     DECLARE column_not_found
         EXCEPTION FOR SQLSTATE '52003';
     MESSAGE 'Hello from InnerProc.' TO CLIENT;
     SIGNAL column_not_found;
     MESSAGE 'SQLSTATE set to ',
     SQLSTATE, ' in InnerProc.' TO CLIENT;
 END
```

以下语句执行 OuterProc 过程：

```
CALL OuterProc();
```

Interactive SQL [消息] 窗格然后显示以下内容：

```
Hello from OuterProc.

Hello from InnerProc.

SQLSTATE set to 52003 in OuterProc.
```

执行顺序如下所示：

1. OuterProc 执行并调用 InnerProc。
2. 在 InnerProc 中， SIGNAL 语句指示一个错误。
3. MESSAGE 语句不是错误处理语句，因此控制将被传递回 OuterProc 并且不显示该消息。
4. 在 OuterProc 中，跟随在错误后的语句将 SQLSTATE 值指派给名为 **res** 的变量。这是错误处理语句，因此将继续执行并显示 OuterProc 消息。

过程和触发器中警告的缺省处理

对错误和警告的处理方式不同。对错误的缺省操作是为 SQLSTATE 和 SQLCODE 变量设置值，并且在遇到错误时将控制返回到调用环境；对警告的缺省操作是设置 SQLSTATE 和 SQLCODE 值，并且继续执行过程。

删除过程

请记住，在继续本教程前，在 [SQL 语句] 窗格中输入以下命令来删除 InnerProc 和 OuterProc 过程：

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

以下演示过程阐释缺省情况下如何处理警告。这些演示过程基于在「过程和触发器中缺省的错误处理」第 742 页中使用的那些过程。在此例子中， SIGNAL 语句生成 `row not found` 条件，这是警告，而非错误。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT;
 END
 CREATE PROCEDURE InnerProc()
 BEGIN
     DECLARE row_not_found
         EXCEPTION FOR SQLSTATE '02000';
     MESSAGE 'Hello from InnerProc.' TO CLIENT;
     SIGNAL row_not_found;
     MESSAGE 'SQLSTATE set to ',
     SQLSTATE, ' in InnerProc.' TO CLIENT;
 END
```

以下语句执行 OuterProc 过程：

```
CALL OuterProc();
```

Interactive SQL [消息] 窗格然后显示以下内容：

```
Hello from OuterProc.Hello from InnerProc.SQLSTATE set to
02000 in InnerProc.SQLSTATE set to 00000 in OuterProc.
```

通过由警告 (02000) 设置的 SQLSTATE，这两个过程都在生成警告后继续执行。

在 InnerProc 中执行第二个 MESSAGE 语句重置该警告。成功执行的任何 SQL 语句都将 SQLSTATE 重置为 00000，将 SQLCODE 重置为 0。如果某一过程需要保存错误状态，则它必须在导致错误警告的语句执行后立即指派该值。

在过程和触发器中使用异常处理程序

经常需要截取某些类型的错误并在过程或触发器内处理它们，而不是将错误传递回调用环境。上述操作是通过使用**异常处理程序**执行的。

您可以使用复合语句的 EXCEPTION 部分定义异常处理程序（请参见「使用复合语句」第 725 页）。只要在复合语句中发生错误，异常处理程序就会执行。与错误不同，警告不会导致执行异常处理代码。如果错误出现在嵌套的复合语句中，或者出现在复合语句内的任何地方调用的过程或触发器中，异常处理代码也会执行。

删除过程

请记住，在继续本教程前，在 [SQL 语句] 窗格中输入以下命令来删除 InnerProc 和 OuterProc 过程：

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

用来阐释异常处理的演示过程基于在「过程和触发器中缺省的错误处理」第 742 页中使用的那些过程。在此情况下，附加代码处理 InnerProc 过程中的 `the column not found` 错误。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT
 END
 CREATE PROCEDURE InnerProc()
 BEGIN
      DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
      MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL column_not_found;
   MESSAGE 'Line following SIGNAL.' TO CLIENT;
      EXCEPTION
           WHEN column_not_found THEN
           MESSAGE 'Column not found handling.' TO
CLIENT;
           WHEN OTHERS THEN
        RESIGNAL ;
 END
```

EXCEPTION 语句声明异常处理程序本身。 EXCEPTION 语句后的各行不执行，除非发生了错误。每一 WHEN 子句都指定一个异常名称（用 DECLARE 语句声明）以及在遇到该异常时要执行的语句。 WHEN OTHERS THEN 子句指定当发生的异常没有出现在之前的 WHEN 子句时要执行的语句。

在此示例中， RESIGNAL 语句将该异常传递到更高级别的异常处理程序中。如果在异常处理程序中未指定 WHEN OTHERS THEN，则 RESIGNAL 是缺省操作。

以下语句执行 OuterProc 过程：

```
CALL OuterProc();
```

Interactive SQL [消息] 窗格然后显示以下内容：

```
Hello from OuterProc.Hello from InnerProc.Column not found
handling.SQLSTATE set to 00000 in OuterProc.
```

注意

- 在 InnerProc 中， EXCEPTION 语句执行，而不是 SIGNAL 语句之后的那些行执行。
- 因为遇到的错误是 `column not found` 错误，则包括的用来处理该错误的 MESSAGE 语句将执行，并且 SQLSTATE 重置为零 （指示没有发生任何错误）。
- 在异常处理代码执行后，控制被传递回 OuterProc，就像没有遇到任何错误一样继续执行。
- 不应将 ON EXCEPTION RESUME 与显式异常处理一起使用。如果包括 ON EXCEPTION RESUME，将不执行异常处理代码。
- 如果 [列未找到] 异常的错误处理代码只是一个 RESIGNAL 语句，控制就会传递回 SQLSTATE 仍然设置为值 52003 的 OuterProc 过程。这就像是 InnerProc 中没有错误处理代码的情况。由于在 OuterProc 中没有错误处理代码，因此该过程将失败。

异常处理和原子复合语句

在复合语句内处理异常时，该复合语句结束时不激活异常处理，并且出现异常前的更改也不被回退。即使对于原子复合语句，上述情况也同样适用。如果错误发生在原子复合语句内并被显式处理，则执行该原子复合语句中的某些语句，但不执行全部语句。

嵌套的复合语句和异常处理程序

只有 ON EXCEPTION RESUME 子句出现在过程定义中，跟随在导致出现错误的语句之后的代码才会执行。

您可以使用嵌套的复合语句来增强相关控制，以控制哪些语句在出现错误之后仍执行，哪些语句不执行。

删除过程

请记住，在继续本教程前，在 [SQL 语句] 窗格中输入以下命令来删除 InnerProc 和 OuterProc 过程：

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

以下演示过程阐释如何使用嵌套的复合语句来控制流。该过程基于在「过程和触发器中缺省的错误处理」第 742 页中用作示例的那一过程。

```
CREATE PROCEDURE InnerProc()
 BEGIN
     BEGIN
         DECLARE column_not_found
             EXCEPTION FOR SQLSTATE VALUE '52003';
             MESSAGE 'Hello from InnerProc' TO CLIENT;
             SIGNAL column_not_found;
                 MESSAGE 'Line following SIGNAL' TO
CLIENT
                 EXCEPTION
             WHEN column_not_found THEN
                MESSAGE 'Column not found handling' TO
                 CLIENT;
             WHEN OTHERS THEN
                 RESIGNAL;
     END;
         MESSAGE 'Outer compound statement' TO CLIENT;
 END
```

以下语句执行 InnerProc 过程：

```
CALL InnerProc();
```

Interactive SQL [消息] 窗格然后显示以下内容：

```
Hello from InnerProc Column not found handling Outer
compound statement
```

在遇到导致错误的 SIGNAL 语句时，控制传递到用于复合语句的异常处理程序中，并且输出 `Column not found handling` 消息。控制然后传递回外部复合语句中，并且输出 `Outer compound statement` 消息。

如果在内部复合语句中遇到了并非 `column not found` 的其它错误，则异常处理程序执行 RESIGNAL 语句。 RESIGNAL 语句将控制直接传递回调用环境，并且不执行其余的外部复合语句。

在过程中使用 EXECUTE IMMEDIATE 语句

EXECUTE IMMEDIATE 语句使用文字字符串 （在引号中）和变量的组合，允许在过程内构建语句。

例如，以下过程包含创建表的 EXECUTE IMMEDIATE 语句。

```
CREATE PROCEDURE CreateTableProc(
         IN tablename char(30) )
 BEGIN
     EXECUTE IMMEDIATE 'CREATE TABLE '
    || tablename
    || '(column1 INT PRIMARY KEY)'
 END
```

EXECUTE IMMEDIATE 语句可以与返回结果集的查询一起使用。例如：

```
CREATE PROCEDURE DynamicResult(
    IN Columns LONG VARCHAR,
    IN TableName CHAR(128),
    IN Restriction LONG VARCHAR DEFAULT NULL)
 BEGIN
     DECLARE Command LONG VARCHAR;
     SET Command = 'SELECT ' || Columns || ' FROM ' ||
TableName;
     IF ISNULL( Restriction,'') <> '' THEN
           SET Command = Command || ' WHERE ' ||
Restriction;
     END IF;
     EXECUTE IMMEDIATE WITH RESULT SET ON Command;
 END
```

以下语句调用此过程：

```
CALL DynamicResult(
    'table_id,table_name',
    'SYSTABLE',
    'table_id <= 10')
```

table_id	table_name
1	SYSTABLE

table_id	table_name
2	SYSCOLUMN
3	SYSINDEX
...	...

在原子复合语句中，不能使用导致 COMMIT （提交）的 EXECUTE IMMEDIATE 语句，因为在该上下文中不允许 COMMIT。

有关 EXECUTE IMMEDIATE 语句的详细信息，请参见 『ASA SQL 参考』> 「EXECUTE IMMEDIATE 语句 [SP]」。

过程和触发器中的事务和保存点

过程或触发器中的 SQL 语句是当前事务的一部分 （请参见 「使用事务和隔离级别」第 111 页）。您可以在一个事务内调用若干过程，或在一个过程中具有若干事务。

不允许将 COMMIT 和 ROLLBACK 用在任何原子语句中 （请参见 「原子复合语句」第 726 页）。请注意，由于触发触发器的 INSERT、UPDATE 或 DELETE 是原子语句，所以不允许将 COMMIT 和 ROLLBACK 用在触发器或触发器调用的任何过程中。

保存点 （请参见 「事务内的保存点」第 114 页）可在过程或触发器内使用，但 ROLLBACK TO SAVEPOINT 语句永远不可以引用在原子操作开始前的保存点。此外，在完成原子操作后，释放该原子操作内的所有保存点。

编写过程的提示

本节提供与开发过程有关的一些提示。

检查是否需要更改命令分隔符

在您编写过程时，不必在 Interactive SQL 或 Sybase Central 中更改命令分隔符。但是，如果使用某些其它浏览工具创建和测试过程和触发器，则最好将命令分隔符从分号更改为其它字符。

过程内的每一语句均以分号结束。对于要分析 CREATE PROCEDURE 语句本身的某些浏览应用程序而言，需要采用不是分号的其它命令分隔符。

如果您使用要求更改命令分隔符的应用程序，则最好将两个分号作为命令分隔符 (;;)；如果系统不允许使用多字符分隔符，则使用问号 (?) 作为命令分隔符。

记住在过程内分隔语句

应该在过程内用分号终止每一语句。尽管对于语句列表中的最后一个语句可以不使用分号，但最好还是在每一语句后都使用分号。

CREATE PROCEDURE 语句本身包含 RESULT 规范以及构成其主体的复合语句。在 BEGIN 或 END 关键字后，或在 RESULT 子句后，不需要任何分号。

在过程中为表使用全限定名

如果某一过程具有对该过程中包含的表的引用，则应始终以该表的所有者 （创建者）的名称作为表名称的开始部分。

当某一过程引用某一表时，它使用过程创建者的组成员资格用以确定没有显式指定所有者名称的表。例如，如果 user_1 创建的过程引用 Table_B，并且没有指定 Table_B 的所有者，那么， Table_B 就必须已经由 user_1 创建，或者， user_1 必须 （直接或间接地）是作为 Table_B 的所有者的组的成员。如果这两个条件都不具备，调用该过程时将出现消息 [没有找到表]。

通过在语句内使用相关名来提供用于该表的方便的名称，可以尽量减小使用长全限定名所带来的不便。在 『ASA SQL 参考』> 「FROM 子句」中介绍了相关名。

在过程中指定日期和时间

将日期和时间从过程发送到数据库时，它们将作为字符串发送。该字符串的日期部分根据 DATE_ORDER 数据库选项的当前设置进行解释。因为不同的连接可以将该选项设置为不同的值，因此，某些字符串可能会错误地转换为日期，或者数据库可能无法将字符串转换为日期。

在过程内使用数据字符串时，您应该使用明确的日期格式 *yyyy-mm-dd* 或 *yyyy/mm/dd*。服务器会将这些字符串明确解释为日期，与 DATE_ORDER 数据库选项设置无关。

有关日期和时间的详细信息，请参见 『ASA SQL 参考』> 「日期和时间数据类型」。

验证正确传递了过程输入参数

验证输入参数的一个方法是使用 MESSAGE 语句在 Interactive SQL [消息] 窗格中显示参数的值。例如，以下过程只显示输入参数 *var* 的值：

```
CREATE PROCEDURE message_test (IN var char(40))
 BEGIN
     MESSAGE var TO CLIENT;
 END
```

您还可以使用存储过程调试程序。

批处理中允许的语句

除了以下若干语句，所有 SQL 语句在批处理中都是可接受的 （包括数据定义语句，例如 CREATE TABLE、 ALTER TABLE 等）：

- CONNECT 或 DISCONNECT 语句。
- ALTER PROCEDURE 或 ALTER FUNCTION 语句。
- CREATE TRIGGER 语句。
- Interactive SQL 命令，例如 INPUT 或 OUTPUT。
- 不能在批处理中使用主机变量。

允许使用 CREATE PROCEDURE 语句，但必须是批处理中的最后一个语句。因此，批处理只能包含一个 CREATE PROCEDURE 语句。

在批处理中使用 SELECT 语句

可以在批处理中包括一个或多个 SELECT 语句。

以下是有效批处理：

```
IF EXISTS(    SELECT *
                 FROM SYSTABLE
                 WHERE table_name='employee' )
 THEN
     SELECT    emp_lname AS LastName,
                 emp_fname AS FirstName
     FROM employee;
     SELECT lname, fname
     FROM customer;
     SELECT last_name, first_name
     FROM contact;
 END IF
```

只有在第一个 SELECT 语句中结果集的别名才是必需的，因为服务器在批处理中使用第一个 SELECT 语句描述结果集。

在每一查询后需要使用 RESUME 语句，以检索下一个结果集。

从过程调用外部库

您可以从存储过程或用户定义的函数调用外部库中的函数。可以在 Windows 操作系统下调用 DLL 中的函数，在 NetWare 下调用 NLM 中的函数，并且在 UNIX 上的共享对象中调用函数。不能在 Windows CE 上调用外部函数。

本节描述如何在过程中使用外部库调用。外部存储过程示例以及建立包含它们的 DLL 所需的文件位于以下文件夹中：**%ASANY9%\Samples\Asa\ExternalProcedures**。

注意
从过程调用的外部库共享服务器的内存。如果您从过程调用外部库并且该外部库包含内存处理错误，则可能会造成该服务器崩溃或损坏数据库。在生产数据库上部署库之前，请确保对这些库进行彻底测试。

在本节中描述的 API 代替较旧的 API。仍支持写入较旧 API 的库 （在早于 7.0 版的版本中使用），但在进行新开发时，应使用新 API。

Adaptive Server Anywhere 包括一组可以利用此功能的系统过程，例如发送 MAPI 电子邮件消息。

有关系统过程的详细信息，请参见 『ASA SQL 参考』> 「系统过程和函数」。

创建具有外部调用的过程和函数

本节提供具有外部调用的过程和函数的某些示例。

需要 DBA 权限
要创建引用外部库的过程或函数，您必须具有 DBA 权限。此要求比用于创建其它过程或函数所需的 RESOURCE 权限更严格。

语法

您可按如下所示创建一个过程，该过程在 DLL *library.dll* 中调用函数 *function_name*：

```
CREATE PROCEDURE dll_proc ( parameter-list )
 EXTERNAL NAME 'function_name@library.dll'
```

如果您从某个过程调用外部 DLL，则该过程不能执行任何其它任务；它只以该 DLL 为基础构成一个参数包。

如下所示为类似的 CREATE FUNCTION 语句：

```
CREATE FUNCTION dll_func ( parameter-list )
 RETURNS data-type
 EXTERNAL NAME 'function_name@library.dll'
```

在这些语句中， *function_name* 是动态链接库中函数的导出的名称，*library.dll* 是该库的名称。 *parameter-list* 中的参数必须在类型和顺序上与库函数所预期的参数相符。库函数使用在 「外部函数原型」第 760 页中介绍的 API 访问过程参数。

外部函数返回的任何值又由该过程返回到调用环境中。

没有其它允许的语句

引用外部函数的过程可以不包括任何其它语句：其唯一目的在于获取用于函数的参数、调用函数，以及将任何值和返回的参数从函数返回到调用环境。您可以采用与其它过程相同的方式在过程调用中使用 IN、INOUT 或 OUT 参数：输入值传递到外部函数，并且该函数修改的任何参数都会在 OUT 或 INOUT 参数中返回到调用环境。

系统相关的调用

您可以指定操作系统相关的调用，以便在一个操作系统上运行时过程调用一个函数，并且在另一个系统上调用另一个函数（推测是相似的）。此类调用的语法涉及以操作系统名称作为函数名称的前缀。例如：

```
CREATE PROCEDURE dll_proc ( parameter-list )
 EXTERNAL NAME

'Windows95:95_fn@95_lib.dll;WindowsNT:nt_fn@nt_lib.dll
'
```

操作系统标识符必须是 **WindowsNT**、 **Windows95**、 **UNIX** 或 **NetWare** 之一。

如果函数列表不包含用于服务器在其上运行的操作系统的条目，但该列表确实包含未指定操作系统的条目，则数据库服务器调用该条目中的函数。

NetWare 调用在格式上与其它操作系统稍有不同。在 NetWare 上所有符号都是全局已知的，因此，导出的任何符号 （例如函数名称）在该系统上对于所有 NLM 必须是唯一的。因此，只要已经加载 NLM，该 NLM 的名称在调用中就不是必需的。

建议您始终使用库名，而不管是否已经加载 NLM。如果*尚未*加载 NLM，则*必须*提供一个库名。文件扩展名 *.nlm* 是可选的。

有关 CREATE PROCEDURE 语句的语法的详细信息，请参见 『ASA SQL 参考』> 「CREATE PROCEDURE 语句」。

有关 CREATE FUNCTION 语句的语法的详细信息，请参见 『ASA SQL 参考』> 「CREATE FUNCTION 语句」。

外部函数原型

本节描述外部库中用于函数的 API。

该 API 是通过名为 *extfnapi.h* 的头文件定义的，此头文件位于 SQL Anywhere Studio 安装目录的 *h* 子目录中。此头文件处理外部函数原型的平台相关的功能。该 API 取代外部库中函数的先前的 API。

声明 API 版本

若要通知数据库服务器该外部库不是使用旧 API 编写的，请按如下所示提供函数：

uint32 extfn_use_new_api( )

该函数返回 32 位无符号整数。如果返回值是非零值，则该数据库服务器就假定您没有使用旧 API。

如果该函数不是由 DLL 导出的，则数据库服务器就假定正在使用旧 API。在使用新 API 时，返回值必须是在 *extfnapi.h* 中定义的 API 版本号。

在 NetWare 上，要让数据库服务器知道外部过程是使用新 API 编写的，您的 NLM 就必须导出一个名为 extfn_use_new_api 的函数，或者是一个名为 *name*_use_new_api 的函数 （其中的 *name* 是 NLM 的名称）。例如，名为 *external.nlm* 的 NLM 将导出函数 external_use_new_api。

导出 *name*_use_new_api 可避免同时使用多个外部 NLM 时发生导出名称冲突。如果 NLM 导出名为 *name*_use_new_api 的函数，则 CREATE PROCEDURE 或 CREATE FUNCTION 语句必须包含 NLM 名称。

函数原型

函数的名称必须与在 CREATE PROCEDURE 或 CREATE FUNCTION 语句中引用的名称匹配。函数声明必须如下所示：

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

函数必须返回 void，并且必须使用以下参数：一个结构来传递参数，一个 SQL 过程提供的参数的句柄。

an_extfn_api 结构具有以下形式：

```
typedef struct an_extfn_api {
     short (SQL_CALLBACK *get_value)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value
                     );
     short (SQL_CALLBACK *get_piece)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value,
                     a_SQL_uint32    offset
                     );
     short (SQL_CALLBACK *set_value)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value
                     short          append
                     );
 void (SQL_CALLBACK *set_cancel)(
                     void *    arg_handle,
                     void *    cancel_handle
                     );
 } an_extfn_api;
```

an_extfn_value 结构具有以下形式：

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

注意

对 OUT 参数调用 **get_value** 将返回该参数的数据类型，并且将数据作为空值返回。

任何给定参数的 **get_piece** 函数只能在同一参数的 **get_value** 函数后立即调用。

若要返回空值，请在 **an_extfn_value** 中将 **data** 设置为空值。

set_value 的 **append** 字段确定提供的数据是替换 （假）还是附加到（真）现有数据。在用 `append=TRUE` 为同一参数调用 **set_value** 前，必须使用 `append=FALSE` 调用它。对于固定长度的数据类型，将忽略 **append** 字段。

头文件本身包含一些附加说明。

下表说明在 **an_extfn_api** 中定义的函数返回假所基于的条件：

函数	在以下条件为真时返回 0；否则返回 1
get_value()	- **arg_num** 是无效的；例如，**arg_num** 大于 **ext_fn** 中参数的数目 — 它是在正确初始化外部函数调用之前调用的
get_piece()	- **arg_num** 是无效的；例如，**arg_num** 不对应于最后的 **get_value**。 |- 该偏移超出用于 arg_num 参数的值的总长度。 - 在外部函数调用已正确初始化之前调用它。

函数	在以下条件为真时返回 0；否则返回 1
set_value()	- **arg_num** 是无效的；例如，**arg_num** 大于 **ext_fn** 中参数的数目。 - **arg_num** 参数是仅输入的。 - 提供的值的类型与 **arg_num** 参数的类型不匹配。 - 在外部函数调用已正确初始化之前调用它。

有关可以在 a_sql_data_type 字段中输入的值的详细信息，请参见『ASA 编程指南』> 「嵌入式 SQL 数据类型」。

有关将参数传递给外部函数的详细信息，请参见 「将参数传递给外部函数」第 763 页。

实施取消处理

应该取消的外部函数必须通过调用 **set_cancel** API 函数通知数据库服务器。您必须导出特殊函数，以使外部操作可被取消。该函数必须具有以下形式：

void an_extfn_cancel(void * *cancel_handle* **)**

如果该 DLL 不导出此函数，则数据库服务器在 DLL 中忽略任何用于函数的用户中断。在此函数中， *cancel_handle* 是一个指针，它是在以上 **an_extfn_api** 结构中列出的 *set_cancel* API 函数在每次调用该外部函数时，由指向数据库服务器的将要取消的函数提供的。

将参数传递给外部函数

数据类型

以下 SQL 数据类型可被传递给外部库：

SQL 数据类型	C 类型
CHAR	具有指定长度的字符数据
VARCHAR	具有指定长度的字符数据
LONG VARCHAR	具有指定长度的字符数据
BINARY	具有指定长度的二进制数据

SQL 数据类型	C 类型
LONG BINARY	具有指定长度的字符数据
TINYINT	1 字节整数
[UNSIGNED] SMALLINT	[无符号的] 2 字节整数
[UNSIGNED] INT	[无符号的] 4 字节整数
[UNSIGNED] BIGINT	[无符号的] 8 字节整数
VARBINARY	具有指定长度的二进制数据
REAL	单精度浮点数
DOUBLE	双精度浮点数

您不能使用日期或时间数据类型，并且不能使用精确数值数据类型。

若要为 INOUT 或 OUT 参数提供值，请使用 **set_value** API 函数。若要读取 IN 和 INOUT 参数，请使用 **get_value** API 函数。

传递空值

对于所有参数，您都可以将空值作为有效值传递。外部库中的函数可以将空值作为任何数据类型的返回类型提供。

外部函数返回类型

下表列出了支持的返回类型以及它们如何映射到 SQL 函数或过程的返回类型。

C 数据类型	SQL 数据类型
void	用于外部过程。
char *	返回 CHAR() 的函数。
long	返回 INTEGER 的函数。
float	返回 FLOAT 的函数。
double	返回 DOUBLE 的函数。

如果外部库中的函数返回空值，并且已声明 SQL 外部函数返回 CHAR()，则 SQL 扩展函数的返回值是空值。

隐藏过程、函数、触发器和视图的内容

在某些情况下，当您分发应用程序和数据库时，可能不希望透露过程、函数、触发器和视图中包含的逻辑。作为附加的安全手段，您可以使用 ALTER PROCEDURE、 ALTER FUNCTION、 ALTER TRIGGER 和 ALTER VIEW 语句的 SET HIDDEN 子句来隐藏这些对象的内容。

SET HIDDEN 子句可打乱所关联对象的内容，使这些内容变得不可读，而您仍然可以使用这些对象。您还可以卸载对象并将对象重装入其他数据库。

修改是不可逆的，对于使用 8.0 或更高版本创建的数据库，将会删除对象的原始文本。因此需要在数据库外保留对象的原始内容。

用存储过程调试程序进行调试将不会显示过程定义，对过程进行分析也不会显示源。

在已经隐藏的对象上运行上面任何一个语句不会产生任何结果。

要隐藏特定类型的所有对象的文本，您可以使用类似于下面的循环：

```
begin
     for hide_lp as hide_cr cursor for
         select proc_name,user_name
         from SYS.SYSPROCEDURE p, SYS.SYSUSERPERM u
         where p.creator = u.user_id
         and p.creator not in (0,1,3)
     do
         message 'altering ' || proc_name;
         execute immediate 'alter procedure "' ||
             user_name || '"."' || proc_name
             || '" set hidden'
     end for
 end
```

有关详细信息，请参见 『ASA SQL 参考』> 「ALTER FUNCTION 语句」、『ASA SQL 参考』> 「ALTER PROCEDURE 语句」、『ASA SQL 参考』> 「ALTER TRIGGER 语句」和 『ASA SQL 参考』> 「ALTER VIEW 语句」。

第 21 章

调试数据库中的逻辑

关于本章

本章介绍了如何使用 Sybase 调试程序协助开发 SQL 存储过程、触发器、事件处理程序及 Java 存储过程。

调试数据库简介

您可以在开发下列对象的过程中使用调试程序：

- SQL 存储过程、触发器、事件处理程序和用户定义的函数。
- 数据库中的 Java 存储过程

本章讲解如何设置和使用调试程序。

可单独授权的组件

数据库中的 Java 是单独授权的组件，在安装它之前必须进行订购。若要订购该组件，请参见 SQL Anywhere Studio 产品包中的卡片或者访问 http://www.sybase.com/detail?id=1015780。

调试程序的功能

使用调试程序可以执行许多任务，包括：

- **调试存储过程和触发器** 可以对 SQL 存储过程和触发器进行调试。
- **调试事件处理程序** 事件处理程序是 SQL 存储过程的扩展。本章中有关调试存储过程的内容也同样适用于调试事件处理程序。
- **浏览存储过程和类** 您可以浏览 SQL 过程的源代码，也可以浏览已安装的 Java 类的源代码（只要您的磁盘上有这些类的代码）。
- **调试 Java 类** 可以对存储在数据库中的 Java 类进行调试。
- **跟踪执行** 可以逐行单步执行在数据库中运行的存储过程或 Java 类代码。也可以监视被调用的函数堆栈。
- **设置断点** 可以让代码运行到断点处，并在该处停止运行。
- **设置中断条件** 断点包括代码行，但也可以指定中断执行代码的条件。例如，可以在第十次执行某行代码时停止执行，或仅在某个变量具有特定的值时停止执行。也可以在 Java 应用程序每次抛出某个特定例外时停止执行。
- **检查和修改局部变量** 在断点处停止执行后，可以检查和变更局部变量的值。

- **在表达式处中断执行及检查**　在断点处停止执行后，可以检查各种表达式的值。
- **检查和修改行变量**　行变量是行级触发器的 OLD 和 NEW 值。您可以检查和修改这些值。
- **执行查询**　在 SQL 过程中的断点处停止执行后，可以执行查询。这样，您就可以查看临时表中保存的中间结果，以及检查基表中的值和查看查询执行计划。

使用调试程序的要求

要使用调试程序，需要满足以下要求：

- **权限**　要使用调试程序，您必须具有 DBA 权限或被授予 SA_DEBUG 组中的权限。创建所有数据库时，都会将该组添加到数据库中。
- **Java 类的源代码**　必须为调试程序提供应用程序的源代码。对于 Java 类，源代码保存在硬盘上的目录中。对于存储过程，源代码保存在数据库中。
- **编译选项**　要调试 Java 类，编译这些类时必须使其包含调试信息。例如，如果使用 Sun Microsystems JDK 编译器 *javac.exe*，就必须使用 `-g` 命令行选项进行编译。

教程：调试程序入门

本教程讲解如何启动调试程序、连接到数据库、调试简单的存储过程，以及如何调试 Java 类。

第 1 课：连接到数据库并启动调试程序

本课讲解如何启动调试程序、连接到数据库，使用连接以进行调试。它使用 Adaptive Server Anywhere 示例数据库。

启动调试程序

❖ **启动调试程序：**

1 启动 Sybase Central：

选择 [开始] → [程序] → [SQL Anywhere 9] → [Sybase Central]。

2 连接到数据库。

在本教程中，您要连接到 Adaptive Server Anywhere 示例数据库。

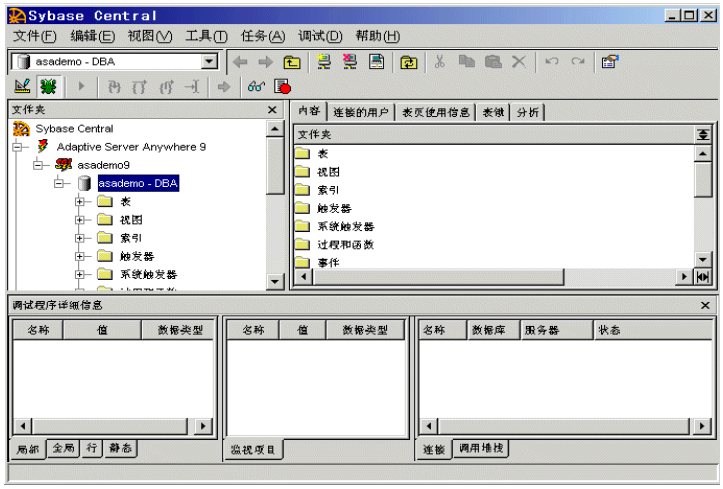
1. 在左窗格中，右击 [Adaptive Server Anywhere] 并从弹出式菜单中选择 [连接]。

[连接] 对话框即会出现。

3 选择 [调试] 模式。

使用 Sybase Central 时，可以在 [设计] 模式或 [调试] 模式中。在 [调试] 模式中运行时，调试程序断点是活动的。而且，Sybase Central 会显示调试菜单和 [调试程序详细信息] 窗格。

要选择 [调试] 模式，请单击 [任务] → [调试]。 Sybase Central 的底部会显示 [调试程序详细信息] 窗格， Adaptive Server Anywhere 工具栏会显示一组调试程序工具。



第 2 课：调试存储过程

本课通过一个示例会话讲解如何调试存储过程。本课上接 「第 1 课：连接到数据库并启动调试程序」第 770 页。

本教程说明了如何使用调试程序来标识存储过程中的错误。要设置该阶段，您可以故意在存储过程 debugger_tutorial （它是示例数据库的一部分）中引入一个错误。

debugger_tutorial 过程应该返回一个结果集，结果集中包含签订了最高价值订单的公司的名称及其订单价值。它通过循环遍历列出公司和订单的查询结果集来计算这些值。（通过使用 SELECT FIRST 查询，不在过程中添加逻辑就可以获得此结果。该过程用于创建方便的示例。）该过程中有一个有意安排的错误。在本教程中，您要诊断并修正该错误。

运行 debugger_tutorial 过程

debugger_tutorial 过程应该返回包含排在第一位的公司及其订购的产品价值的结果集。由于错误所致，它不会返回此结果集。在这一课，您要运行该存储过程。

❖ 运行 debugger_tutorial 存储过程

1 在 Sybase Central 左窗格中，打开 [过程和函数] 文件夹。

2 执行该过程。

右击 debugger_tutorial 过程，并从弹出式菜单中选择 [从 Interactive SQL 执行]。

一个 Interactive SQL 窗口会打开，并且会显示以下结果集：

top_company	top_value
(NULL)	(NULL)

这无疑是错误的结果。教程的其余部分将诊断生成此结果的错误。

3 关闭 Interactive SQL 窗口以清除工作区。

诊断错误

要诊断过程中的错误，请在过程中设置断点并单步执行代码，同时在执行过程时监视变量的值。

下面，您要在过程中的第一个可执行语句处设置断点。

❖ 诊断错误

1 将 Sybase Central 更改为 [调试] 模式。

从 [任务] 菜单中，选择 [调试]。 Sybase Central 会在主窗口的底部显示 [调试程序详细信息] 窗格。

2 在过程中的第一个可执行语句处设置断点。

该语句包含以下文本：

```
open cur_this_cust;
```

在垂直灰色条中这一行的左侧单击以设置断点。断点会显示为红色圆圈。另一种设置断点的方法是按 F9 键。

3　再次执行该过程。

1. 在左窗格中，右击 sp_customer_products 过程并从弹出式菜单中选择 [从 Interactive SQL 执行]。

2. 您会看到一个消息框，询问是否要从 Interactive SQL 调试连接。单击 [是]。

过程会在断点处停止执行。源代码窗口中的黄色箭头表示当前位置，当前位置在断点处。

4　检查变量。

[调试程序详细信息] 窗格中的 [局部变量] 窗口显示过程中的变量及其当前值和数据类型的列表。 top_company、top_value、 this_value 和 this_company 变量均未初始化，因此为 NULL。

5　单步执行代码。

多次按 F11 键以单步执行代码，直到到达下面的行：

```
if this_value > top_value then
```

单步执行存储过程的各行时，变量的值会改变。

位于 `if` 语句处时， this_value 设置为 3000，并且 top_value 仍旧为 NULL。

6　单步执行多个语句。

多次按 F11 键以查看执行采用了哪个分支。黄色箭头直接移回到循环开始处的标签语句中，该语句包含以下文本：

```
customer loop: loop
```

`if` 测试没有返回 true。该测试之所以会失败，是因为任何值与 NULL 的比较都返回 NULL。值为 NULL 会导致测试失败并且不执行 `if...end if` 语句中的代码。

此时，您可能会意识到问题在于 top_value 未初始化这一事实上。

确认诊断并修正错误

您可以测试 [问题在于调试程序中缺少对 top_value 的初始化] 这一假设，而不更改过程代码。

❖ 测试假设

1 为 top_value 设置值。

在 [局部] 窗口中，单击 top_value 变量的 [值] 字段，并输入值 3000。

2 再次单步执行循环。

按 F11 键以单步执行 `if` 语句的指令并检查 this_value 和 top_value 的值。在获取大于 3000 的 top_value 值以前，您可能必须单步执行多个循环。

3 禁用断点并执行过程。

1. 单击断点以便它变为灰色 （禁用）。
2. 按 F5 键以结束过程的执行。

[Interactive SQL] 窗口会再次出现。它会显示正确的结果。

top_company	top_value
Chadwicks	8076

假设已经得到确认。问题在于没有初始化 top_value。

❖ 修正错误

1 从 [任务] 菜单中，选择 [设计] 以离开 [调试] 模式。

2 紧跟在包含以下文本的行后面：

```
open cur_this_cust;
```

新建一个初始化 top_value 变量的行：

```
set top_value = 0;
```

3　按 CTRL+S 键以保存修改过的过程。

4　再次执行过程，并确认 Interactive SQL 显示了正确的结果。

您现在已经完成了本课。关闭 Sybase Central 和任何打开的 Interactive SQL 窗口。

第 3 课：调试 Java 类

本课讲解调试 Java 类的示例会话。

本课要求您的数据库组件中有 Java。

需要有可单独授权的选件
数据库中的 Java 是单独授权的组件，在安装它之前必须进行订购。

要订购此组件，请参见 『SQL Anywhere Studio 介绍』> 「单独授予许可的组件」。

在本课中，您将从 Interactive SQL 调用 JDBCExamples.Query()，在调试程序中中断执行，然后跟踪执行该方法的源代码。

JDBCExamples.Query() 方法对示例数据库执行以下查询：

```
SELECT id, unit_price
 FROM product
```

然后循环检查结果集中的所有行，并返回单价最高的那一行。

按调试要求编译 Java 类

您必须使用 javac `-g` 选项编译类才能对其进行调试。示例类已按调试要求编译。

准备数据库

要完成本教程，您必须启用示例数据库以使用 Java，并在示例数据库中安装 *JDBCExamples.class*。

有关说明，请参见 『ASA 编程指南』> 「设置 Java 示例」。

有关 JDBCExamples 类及其方法的信息，请参见 『ASA 编程指南』> 「JDBC 编程」。

在调试程序中显示 Java 源代码

调试程序在一组位置中查找源代码文件 （扩展名为 *.java*）。需要将安装目录的 *Samples\ASA\Java* 子目录添加到该位置列表中，这样才能在调试程序中显示数据库中当前执行的类的代码。

❖ 在调试程序中显示 Java 源代码：

1 启动 Sybase Central，并连接到示例数据库 （ASA 9.0 Sample ODBC 数据源）。

2 选择 Sybase Central 调试任务。

选择 [任务] → [调试]。

3 设置查找 Java 源代码的位置。

1. 从 [调试] 菜单中，选择 [设置 Java 源路径]。 [Java 源路径] 窗口即会出现。
2. 单击 [浏览文件夹]。浏览到存储 Java 源代码 （*.java* 文件）的文件夹。要查看 JDBCExamples 类的源代码，请浏览到安装 SQL Anywhere 的 *Samples\ASA\Java* 子目录。如果在缺省安装目录中安装了 SQL Anywhere，您要浏览到以下目录：

```
C:\Program Files\Sybase\SQL Anywhere 9\Samples
\ASA\Java
```

4 显示 JDBCExamples 类的源代码：

1. 在 Sybase Central 的左窗格中，打开 [Java 对象] 文件夹。

2. 在右窗格中，打开 [所有的 Java 类] 文件夹，并找到要调试的 JAR 文件或类。根据 Sybase Central 设置不同，您可能要单击 [创建者] 列。这样会按创建者为列表排序，以便 DBA 拥有的类显示在 SYS 拥有的类前面。
3. 双击 JDBCExamples 类。
4. 在右窗格中，单击 [源] 选项卡。类的源代码即会显示。

设置断点

❖ **在 Java 类中设置断点：**

1 在源代码窗口中，向下翻页，直到看到 **Query** 方法的开头。该方法接近该类的末尾，从以下行开始：

```
public static int Query() {
```

2 单击该行左侧的灰色列，直到它显示红色圆圈。

```
int max_price = 0;
```

重复单击该指示符将切换其状态。

运行该方法

❖ **从 Interactive SQL 中调用该方法：**

1 启动 Interactive SQL。使用 ID **DBA** 和口令 **SQL** 连接到示例数据库。

2 在 Interactive SQL 中输入以下命令，调用该方法：

```
SELECT JDBCExamples.Query()
```

该查询不会完成，而是在执行到断点处后在调试程序中停止。在 Interactive SQL 中，[中断 SQL 语句] 按钮处于活动状态。在调试程序的 [源代码] 窗口中，黄色箭头表示当前行。

您现在可以在调试程序中单步执行源代码并完成调试活动。

单步执行源代码

本节举例说明一些在调试程序中单步执行代码的方法。

上一节结束后，调试程序应在方法的第一条语句处停止执行 **JDBCExamples.Query**：

示例

您可以尝试执行以下示例步骤：

1. **单步执行到下一行** 选择 [调试] → [单步跳过]，或按 F10 键，单步执行到当前方法中的下一行。请尝试执行两三次。
2. **运行到游标** 使用鼠标选择以下行，然后选择 [调试] → [运行到游标]，或按 Ctrl+F10 以运行到该行并中断：

   ```
   max_price = price;
   ```

   黄色箭头将移动到该行。
3. **设置断点并执行到断点处** 将游标放置在以下行 （第 292 行）中，然后按 F9 键，在该行设置断点：

   ```
   return max_price;
   ```

   左栏中将出现一个红色停止符号，标记该断点。按 F5 键，执行到该断点。
4. **试验** 尝试使用不同的方法单步执行代码。最后按 F5 键以完成执行。

   [调试] 菜单中具有用于单步执行源代码的全套选项。

   执行完毕后， Interactive SQL [结果] 选项卡上的 [结果] 窗格中将显示值 24。

检查和修改变量

在本课中，您可以在调试程序中既可以修改局部变量 （在方法中声明的变量）的值，也可以修改类静态变量的值。

检查局部变量

在单步执行代码时，可以检查方法中局部变量的值，以更好的了解执行情况。您必须已使用 javac `-g` 选项编译了类，才可以这样做。

❖ **检查和修改变量的值：**

1 在 **JDBCExamples.Query** 方法的第一行设置断点。该行如下所示：

```
int max_price = 0
```

2 在 Interactive SQL 中，再次输入以下语句，执行该方法：

```
SELECT JDBCExamples.Query()
```

查询只执行到断点处。

3 按 F10 键以单步执行到下一行。现在已声明了 **max_price** 变量，并已将其初始化为零。

4 在 [局部] 选项卡列表中，双击 **max_price** 的 [值] 列条目，然后键入 45，将 **max_price** 的值更改为 45。

值 45 高于其他任何价格。现在，该查询返回的最高价格将不再是 24，而是 45。

5 重复按 F10 键以单步执行代码。这样做时，变量的值会显示在 [局部] 选项卡列表中。单步执行，直到 **stmt** 和 **result** 变量具有值。

6 单击 **result** 对象旁边的图标，将其展开，或者将游标放在该行上，然后按 Enter 键。这将显示该对象中的字段的值。

7 在体验过检查和修改变量后，按 F5 键以结束该查询的执行。本教程到此结束。

检查静态变量

除局部变量外，您还可以在调试程序的 [静态] 选项卡中显示类级变量（静态变量），并在 [监视] 选项卡中监视它们的值。有关详细信息，请参见调试程序的联机帮助。

使用断点

本节讲解如何使用断点控制调试程序中断执行源代码。

设置断点

断点可以控制调试程序在指定行中断执行。

❖ 设置断点

1 在 Sybase Central 运行 [调试] 任务时，显示要在其中设置断点的代码。

2 在窗口左侧的灰色列中单击，或者单击一行并按 F9 键来设置断点。红色圆圈表示行上设置了断点。

❖ 设置断点 （[调试] 菜单）

1 显示 [断点] 窗口。

在 Sybase Central 运行 [调试] 任务时，选择 [调试] → [断点]。

2 在 [断点] 窗口中，单击 [新建断点]。 [新建断点] 窗口即会出现。

3 从下拉列表中选择一个 [过程] 名称，并且可以选择输入条件和计数值。

条件是 SQL 或 Java 表达式必须计算为 true，断点才可中断执行。例如，您可以设置断点以应用于指定用户建立的连接，您可以输入以下条件：

```
CURRENT USER = 'user-name'
```

计数是断点停止执行前命中断点的次数。值为 0 意味着，断点始终停止执行。

4　单击 [确定] 以设置断点。断点会设置在过程中的第一个可执行语句上。

设置断点时，断点会应用于所有连接。要使断点只应用于特定连接，请设置断点条件。

❖ 设置断点条件：

1　右击断点标记，然后从弹出式菜单中选择 [编辑断点]。随即出现 [编辑断点] 对话框。

2　在 [条件] 字段中输入条件。例如，要设置断点使其仅应用于特定用户 ID 的连接，请输入下面的条件：

```
CURRENT USER='user-name'
```

其中 *user-name* 是将为其激活断点的用户 ID。

禁用和启用断点

您可以从 Sybase Central 右窗格或者从 [断点] 窗口更改断点的状态。

❖ 更改断点的状态

1　显示过程的源代码，该源代码中包含要更改其状态的断点。

2　单击要编辑的行左侧的断点指示符。行的状态会从活动断点切换为禁用的断点。

❖ 更改断点的状态 （[断点] 窗口）：

1　打开 [断点] 窗口。

2　编辑断点。

3　或者，您可以选择该断点，然后按 Delete 键，将其删除。

编辑断点条件

可以为断点添加条件，控制调试程序仅在满足特定条件或计数时在该断点处中断执行。

❖ 为断点添加条件或计数：

- 打开 [断点] 窗口。

  对于 Java 类，条件必须是 Java 布尔表达式。而对于过程和触发器，必须是 SQL 搜索条件。

示例

在过程 sp_contacts 中，您可以将断点

```
contact.id = contact.old_id
```

用于代码行：

```
DELETE FROM contact WHERE contact.id = contact.old_id
```

在 Java 方法 JDBCExamples.Query() 中，您可以将断点条件

```
(price < 10)
```

用于代码行：

```
if (max.price == price) or (price == 10)
```

使用变量

使用调试程序，您可以在单步执行代码时查看并编辑变量的行为。调试程序提供了 [调试程序详细信息] 窗口，以显示在存储过程和 Java 类中使用的各种变量。当 Sybase Central 运行 [调试] 任务时，[调试程序详细信息] 窗口会显示在 [Sybase Central] 窗口的底部。

局部变量

❖ **监视变量的值：**

1. 在要检查变量的过程中设置断点。

   有关设置断点的信息，请参见 「设置断点」第 780 页。

2. 在 [调试程序详细信息] 的 [变量] 窗格的 [局部] 选项卡上单击。

3. 运行该过程。变量和变量的值将显示在 [局部] 选项卡中。

其它变量

全局变量是由 Adaptive Server Anywhere 定义的，它包含有关当前连接、数据库和其它设置的信息。它们显示在 [变量] 窗口的 [全局] 选项卡上。

有关全局变量的列表，请参见 『ASA SQL 参考』> 「全局变量」。

行变量用于保留触发器中使用的值。它们显示在 [变量] 窗口的 [行] 选项卡上。

有关触发器的详细信息，请参见 「触发器简介」第 714 页。

静态变量用于 Java 类，它们显示在 [静态] 选项卡中。

调用堆栈

在调试嵌套过程或 Java 类时，检查已确定的调用顺序是很有用的。您可以在 [调用堆栈] 选项卡中查看过程的列表。

❖ **显示调用堆栈**

1. 在要检查变量的过程中设置断点。

2. 运行代码，直到断点。

过程的名称会显示在 [调用堆栈] 选项卡中。当前过程显示在该列表的最上面。调用它的过程紧接着显示在下面，依此类推。

使用连接

[连接] 窗口显示指向数据库的连接。任何时候都可以运行多个连接。一些连接可能会在断点处停止，而其它连接可能不会。

源代码窗口显示一个连接的状态。要切换连接，请双击 [连接] 窗口中的连接。

设置断点是一项很有用的技术，这样就可以针对某一个用户 ID 中断执行。设置断点的方法是设置以下形式的断点条件：

```
CURRENT USER = 'user-name'
```

SQL 特殊值 CURRENT USER 保存着连接的用户 ID。

有关详细信息，请参见 「编辑断点条件」第 782 页和 『ASA SQL 参考』> 「CURRENT USER 特殊值」。

索引

字母

*=
  Transact-SQL 外连接 301
* （星号）
  SELECT 语句 229
=*
  Transact-SQL 外连接 301
>
  比较运算符 239
@@identity 全局变量
  IDENTITY 列 522
Adaptive Server Anywhere
  与其它 SQL 方言的区别 540
Adaptive Server Anywhere SQL 功能 540
Adaptive Server Enterprise
  GROUP BY 兼容性 278
  兼容性 504
  迁移到 ASA 631
  数据导入 / 导出中的兼容性 640
Adaptive Server 体系结构 507
ALL
  关键字和 UNION 子句 270
  关键字和集合函数 252
  子查询测试 393
ALL 运算符
  关于 393
  注意 393
ALLOW_NULLS_BY_DEFAULT 选项
  为实现 Transact-SQL 兼容性设置 516
ALTER TABLE 语句
  CHECK 约束 96
  并发 160
  示例 47
  外键 53
  主键 50
AND
  使用逻辑运算符 249
ANSI
  SQL/92 标准和不一致 117
  非 ANSI 连接 289
ANSI 更新约束
  访问计划 485
ANSI 兼容 另请参见 SQL 标准 3
ANY 运算符
  关于 391
  问题 392
  子查询测试 391
AS 关键字
  别名 231
asademo.db 文件
  模式 282
asajdbc 服务器类 680
asaodbc 服务器类 684
asejdbc 服务器类 680
aseodbc 服务器类 684
ASTMP 环境变量
  临时文件位置 178
AT 子句
  CREATE EXISTING TABLE 语句 655
AUTO 模式
  使用 565
AUTOINCREMENT
  何时使用 160
AUTOMATIC_TIMESTAMP 选项
  为实现 Transact-SQL 兼容性设置 516
AvgDiskReads
  访问计划中的预计值 485

AvgDiskReadTime
访问计划中的预计值 485
AvgDiskWrites
estimate in access plans 485
AvgRowCount
访问计划中的预计值 485
AvgRunTime
访问计划中的预计值 485
B 树索引
关于 464
BCP 格式
使用 ASE 导入 / 导出 640
BEGIN TRANSACTION 语句
远程数据访问 667
BETWEEN 关键字
范围查询 241
BLOB
插入 420
关于 29
BLOCKING 选项
使用 124
Bloom 过滤器
查询执行算法 454
CacheHits
访问计划中的统计信息 485
CacheRead
访问计划中的统计信息 485
CacheReadIndLeaf
访问计划中的统计信息 485
CacheReadTable
访问计划中的统计信息 485
CALL 语句
参数 730
关于 700
示例 705
语法 724
CASCADE 动作
关于 108
CASE 语句
语法 724
cdata 指令
使用 580
CHECK 条件
Transact-SQL 508
CHECK 约束
表 97
列 96
删除 99
修改 99
域 98
CLOSE 语句
过程 738
COMMIT 语句
复合语句 726
过程和触发器 754
校验参照完整性 152
远程数据访问 667
COMPUTE 子句
CREATE TABLE 54
不支持 525
CONNECTION_PROPERTY 函数
关于 203
COUNT 函数
NULL 256
关于 255
CREATE DATABASE 语句
ASE 507
使用 35
CREATE DEFAULT 语句
不支持 508
CREATE DOMAIN 语句
Transact-SQL 兼容性 508
使用 101
CREATE EXISTING TABLE 语句
使用 657
CREATE FUNCTION 语句
关于 710

CREATE INDEX 语句
并发 160
CREATE PROCEDURE 语句
参数 729
示例 702
CREATE RULE 语句
不支持 508
CREATE SERVER 语句
JDBC 和 Adaptive Server Enterprise 681
ODBC 和 Adaptive Server Enterprise 686
远程服务器 680
CREATE SERVER 语句中的 USING 参数值 680
CREATE TABLE 语句
Transact-SQL 523
并发 160
代理表 658
关于 44
外键 53
主键 50
CREATE TRIGGER 语句
关于 715
CREATE VIEW 语句
WITH CHECK OPTION 子句 62
CUBE 操作
NULL 360
关于 368
示例 369
DataSet
用于导入 XML 556
用于以 XML 的形式导出关系数据 548
DataWindows
远程数据访问 643
DB_PROPERTY 函数
关于 203
DB2 远程数据访问 687
db2odbc 服务器类 687
dberase 实用程序
使用 37
dbinit 实用程序
使用 36
dbisql 实用程序
重建数据库 626
dbo 用户
ASE 509
dbspace
管理 508
dbunload 实用程序
导出数据 612
重建工具 626
DDL
并发 159, 160
关于 32
debugger_tutorial 过程
关于 771
DECLARE 语句
复合语句 725
过程 738, 743
DELETE 语句
删除过程中的锁定 153
使用 425
DISCONNECT 语句
使用 37
DISK 语句
不支持 508
DiskRead
访问计划中的统计信息 485
DiskReadIndInt
访问计划中的统计信息 485
DiskReadIndLeaf
访问计划中的统计信息 485
DiskReadTable
访问计划中的统计信息 485
DiskWrite
访问计划中的统计信息 485
DISTINCT 关键字
集合函数 252, 255
DISTINCT 列表

访问计划中的项 489
DISTINCT 排除
关于 470
DISTINCT 子句
SELECT 语句 235
排除不必要的 DISTINCT 470
DLL
从过程调用 758
DLLs
外部过程调用 759
DML
关于 416
DROP CONNECTION 语句
使用 37
DROP DATABASE 语句
ASE 507
使用 36
DROP TABLE 语句
示例 48
DROP TRIGGER 语句
关于 720
DROP VIEW 语句
示例 66
DROP 语句
并发 160
DUMP DATABASE 语句
不支持 507
DUMP TRANSACTION 语句
不支持 507
element 指令
使用 576
EstCpuTime
访问计划中的预计值 485
EstDiskReads
访问计划中的预计值 485
EstDiskReadTime
访问计划中的预计值 485
EstDiskWrites
访问计划中的预计值 485
EstRowCount
访问计划中的预计值 485
EstRunTime
访问计划中的预计值 485
Excel 和远程访问 694
EXCEPT 运算
NULL 274
规则 271
使用 271
组合查询 270
EXECUTE IMMEDIATE 语句
过程 752
EXISTS 谓语
将子查询重写为 434
EXISTS 运算符 397
EXPLICIT 模式
编写查询 571
使用 569
使用 cdata 指令 580
使用 element 指令 576
使用 hide 指令 578
使用 xml 指令 579
语法 569
FALSE 和 UNKNOWN 之间的差异 248
FALSE 条件
NULL 248
FETCH 语句
过程 738
fetchtst 211
FIPS 兼容 另请参见 SQL 标准 3
FIRST 子句
关于 267
FOR BROWSE 子句
不支持 525
FOR READ ONLY 子句
忽略 526
FOR UPDATE 子句
不支持 525
FOR XML AUTO

使用 565
FOR XML EXPLICIT
使用 569
使用 cdata 指令 580
使用 element 指令 576
使用 hide 指令 578
使用 xml 指令 579
语法 569
FOR XML RAW
使用 563
FOR XML 子句
BINARY 数据类型 560
EXPLICIT 模式语法 569
IMAGE 数据类型 560
LONG BINARY 数据类型 560
VARBINARY 数据类型 560
使用 AUTO 模式 565
使用 EXPLICIT 模式 569
使用 RAW 模式 563
限制 560
以 XML 格式获取查询结果 559
用法 560
FOR 语句
语法 724
FOR 子句
使用 FOR XML AUTO 565
使用 FOR XML EXPLICIT 569
使用 FOR XML RAW 563
以 XML 格式获取查询结果 559
FORWARD TO 语句
本机语句 663
FoxPro
远程数据访问 695
FROM 子句
存储过程 237
介绍 237
连接说明 284
派生表 237
full compares
访问计划中的统计信息 485
FullCompare
访问计划中的统计信息 485
go
批处理语句分隔符 722
GRANT 语句
Transact-SQL 511
并发 160
GROUP BY ALL 子句
不支持 525
GROUP BY 列表
访问计划中的项 489
GROUP BY 子句
Adaptive Server Enterprise 兼容性 278
order by 和 268
SQL 标准遵从性 278
SQL/92 标准 278
WHERE 子句 263
关于 258
集合函数 258
执行 260
GROUPING SETS
关于 371
连接 373
GROUPING 函数
NULL 360
ROLLUP 操作 360
GUID
缺省列值 93
生成 159
与全局自动增量比较 93
-gx 选项
线程 675
HAVING 子句
GROUP BY 和 264
具有和不具有集合 264

逻辑运算符 265
性能 467
选择数据组 264
子查询 386
HAVING 子句中的子查询 386
hide 指令
使用 578
HOLDLOCK 关键字
Transact-SQL 526
I/O
扫描位图 456
IBM DB2
对 DB2 的远程数据访问 687
迁移到 ASA 631
IDENTITY 列
检索值 522
特殊 IDENTITY 521
IF 语句
语法 724
IN 参数
定义 729
IN 关键字
匹配列表 242
IN 列表
访问计划中的项 489
算法 442
优化 475
IN 条件
子查询 395
IndAdd
访问计划中的统计信息 485
indexes
composite 460, 465
IndLookup
访问计划中的统计信息 485
INOUT 参数
定义 729
INPUT 语句
关于 600
使用 601
INSERT 语句
SELECT 417
插入过程中的锁定 151
关于 417
instest 211
Interactive SQL
.sql 文件的缺省编辑器 637
导出查询结果 614
将关系数据导出为 XML 548
命令分隔符 755
命令文件 637
批处理操作 637
批处理模式 637
运行脚本 637
重建数据库 626
装载命令 638
Interactive SQL 导入向导
关于 599
INTERSECT 运算
NULL 274
规则 271
使用 271
组合查询 270
INTO 子句
使用 733
invocations
访问计划中的统计信息 485
IS NULL 关键字 248
ISNULL 函数
关于 248
ISO SQL/92 标准
并发 117
典型不一致和 117
ISO 兼容 另请参见 SQL 标准 3
Java
调试 Java 类 775
关于调试 768

Java 调试程序
调试 Java 类 775
调试存储过程 771
教程 770
启动 770
显示源代码 776
要求 769
jConnect 元数据支持
安装 42
LEAVE 语句
语法 724
LIKE 条件
LIKE 优化 476
LIKE 优化 476
LIKE 运算符
通配符 243
LOAD DATABASE 语句
不支持 507
LOAD TABLE 语句
关于 602
使用 604
LOAD TRANSACTION 语句
不支持 507
LONG VARCHAR 数据类型
存储 XML 547
LOOP 语句
过程 738
语法 724
Lotus Notes
口令 696
远程数据访问 695
master 数据库
不支持 507
MAX 函数
重写优化 474
MESSAGE 语句
过程 743
Microsoft Access
迁移到 ASA 631
远程数据访问 695
Microsoft Excel
远程数据访问 694
Microsoft FoxPro
远程数据访问 695
Microsoft SQL Server
迁移到 ASA 631
远程访问 691
MIN 函数
重写优化 474
MIN 或 MAX 函数优化 474
msodbc 服务器类 691
NetWare
外部过程调用 759
外部函数 760
NEWID
何时使用 159
NEWID 函数
缺省列值 93
NLM
从过程调用 758
外部过程调用 759
外部函数 760
NOHOLDLOCK 关键字
忽略 526
NOT
使用逻辑运算符 249
NOT 关键字
搜索条件 240
Notes 和远程访问 695
NULL
CUBE 操作 360
DISTINCT 子句 236
EXCEPT 运算 274
INTERSECT 运算 274
ROLLUP 操作 360
Transact-SQL 兼容性 523
UNION 运算 274

比较 247
关于 246
集合函数 256
集合运算符 274
列定义 249
列缺省值 516
排序顺序 267
缺省参数 247
缺省值 94
属性 248
在列中允许 30
NULL 的属性 248
NULL 值
导入数据 606
忽略转换错误 604
示例 360
NULL 值和小计行 360
ODBC
外部服务器 683
应用程序 120
应用程序，和锁定 120
ODBC 服务器类 683, 693
odbcfet 211
OLAP
CUBE 操作 368
GROUPING SETS 操作 371
NULL 值 360
ROLLUP 操作 362
功能 356
关于 355
小计行 357, 366
OLAP 操作
关于 357
OLAP 函数
报告函数 377
窗口函数 379
关于 374
秩函数 374
OLAP 如何工作 357
ON EXCEPTION RESUME 子句
Transact-SQL 538
不用于异常处理 749
存储过程 742
关于 744
ON 短语
连接 290
ON 子句
连接 290
OPEN 语句
过程 738
OPENXML 函数
使用 550
OPTIMIZATION_WORKLOAD 选项
聚簇散列分组依据 450
OR
使用逻辑运算符 249
Oracle
迁移到 ASA 631
Oracle 和远程访问 689
oraodbc 服务器类 689
ORDER BY
访问计划中的项 489
ORDER BY 和 GROUP BY 268
ORDER BY 子句
GROUP BY 268
对查询结果进行排序 266
限制结果 267
性能 200
OUT 参数
定义 729
OUTPUT 语句 614
关于 609
用于将数据导出为 XML 548
PerformanceFetch 211
PerformanceInsert 211
PerformanceTraceTime 211
PerformanceTransaction 211

plans
abbreviations used in 482
reading 482
PowerBuilder
远程数据访问 643
PREPARE 语句
远程数据访问 667
PROPERTY 函数
关于 203
QUOTED_IDENTIFIER 选项
关于 245
为实现 Transact-SQL 兼容性设置 516
RAISERROR 语句
ON EXCEPTION RESUME 538
Transact-SQL 537
RAW 模式
使用 563
READ 语句
执行命令文件 637
reload.sql
导出表 618
导出表数据 627
重建复制数据库 624
重建数据库 620
重装数据库 628
RESIGNAL 语句
关于 748
RESTRICT 动作
关于 108
RETURN 语句
关于 732
REVOKE 语句
Transact-SQL 511
并发 160
ROLLBACK 语句
触发器 530
复合语句 726
过程和触发器 754
ROLLUP 操作
NULL 360
关于 362
了解 GROUP BY 261
示例 364
小计行 366
RowsReturned
访问计划中的统计信息 485
RunTime
访问计划中的统计信息 485
SA_DEBUG 组
调试程序 769
sa_migrate 系统过程
使用 633
sa_migrate_create_fks 系统过程
使用 634
sa_migrate_create_remote_fks_list 系统过程
使用 634
sa_migrate_create_remote_table_list 系统过程
使用 634
sa_migrate_create_tables 系统过程
使用 634
sa_migrate_data 系统过程
使用 634
sa_migrate_drop_proxy_tables 系统过程
使用 634
sa_pause_workload_capture 过程
暂停 [索引顾问] 76
sa_stop_index_consultant 过程
停止 [索引顾问] 76
sa_stop_workload_capture 过程
停止 [索引顾问] 76
SELECT 语句
INSERT 417
INTO 子句 733
Transact-SQL 524
变量 526
别名 231

关于 226
键和查询访问 198
列标题 231
列顺序 230
显示的字符串 232
游标 738
指定行 238
字符数据 245
SELF_RECURSION 选项
Adaptive Server Enterprise 530
SET DEFAULT 动作
关于 108
SET NULL 动作
关于 108
SET OPTION 语句
Transact-SQL 516
SET 子句
UPDATE 语句 422
SIGNAL 语句
Transact-SQL 537
过程 743
sp_addgroup 系统过程
Transact-SQL 511
sp_addlogin 系统过程
Transact-SQL 511
支持 507
sp_adduser 系统过程
Transact-SQL 511
sp_bindefault 过程
Transact-SQL 508
sp_bindrule 过程
Transact-SQL 508
sp_changegroup 系统过程
Transact-SQL 511
sp_dboption 系统过程
Transact-SQL 516
sp_dropgroup 系统过程
Transact-SQL 511
sp_droplogin 系统过程
Transact-SQL 511
sp_dropuser 系统过程
Transact-SQL 511
SQL
输入 227
与其它 SQL 方言的区别 540
SQL 3 兼容 另请参见 SQL 标准 3
SQL 92 兼容 另请参见 SQL 标准 3
SQL 99 兼容 另请参见 SQL 标准 3
SQL Anywhere Studio
文档 x
SQL Remote
复制和并发事务 162
远程数据访问 673
SQL Server 和远程访问 691
SQL 标准 3
GROUP BY 子句 278
非 ANSI 连接 289
兼容性 539
SQL 查询 227
SQL 命令文件
创建 637
关于 637
写入输出 639
运行 637
在 Interactive SQL 中打开 637
执行 637
SQL 语句
编写兼容的 SQL 语句 523
在 Sybase Central 中记录 41
SQL/3 兼容 另请参见 SQL 标准 3
SQL/92 兼容 另请参见 SQL 标准 3
SQL/99 兼容 另请参见 SQL 标准 3
SQL/XML
关于 558
SQL_TXN_ISOLATION 121
SQL_TXN_READ_COMMITTED
隔离级别 120

SQL_TXN_READ_UNCOMMITTED
隔离级别 120
SQL_TXN_REPEATABLE_READ
隔离级别 120
SQL_TXT_SERIALIZABLE
隔离级别 120
SQL-3 兼容 另请参见 SQL 标准 3
SQL3 兼容 另请参见 SQL 标准 3
SQL-92 兼容 另请参见 SQL 标准 3
SQL-99 兼容 另请参见 SQL 标准 3
SQLCA.lock
选择隔离级别 120
与隔离级别 117
SQLCODE 变量
简介 742
SQLSetConnectOption 121
SQLSTATE 变量
简介 742
SQLX 请参见 SQL/XML ix
Sybase Central
安装 jConnect 元数据支持 42
变更表 46
创建表 44
创建数据库 34
创建索引 70
复制表 57
复制数据库对象 83
管理外键 52
管理主键 50
记录 SQL 语句 41
列缺省值 91
列约束 98
启动未连接的数据库 42
删除表 48
删除视图 66
删除数据库 36
设置数据库选项 39
设置统一数据库 40
显示系统表 58
显示系统对象 40
消除数据库 36
校验索引 72
修改视图 65
重建数据库 626
转换过程 532
Sybase Central 性能监控器 204
SYSCOLSTAT
更新列统计信息 431
SYSCOLUMNS 视图
名称冲突 515
SYSINDEX 表
索引信息 81
SYSINDEXES 视图
名称冲突 515
索引信息 81
SYSIXCOL 表
索引信息 81
SYSSERVERS 系统表
远程服务器 646
SYSTABLE 系统表
视图信息 67
SYSVIEWS 视图
视图信息 67
TEMP 环境变量
临时文件位置 178
TIMESTAMP 数据类型
Transact-SQL 519
TMP 环境变量
临时文件位置 178
TMPDIR 环境变量
临时文件位置 178
TOP 子句
关于 267
TRACEBACK 函数 743
Transact-SQL
IDENTITY 列 521
NULL 523

timestamp 列 519
编写兼容的 SQL 语句 523
编写可移植的 SQL 523
变量 535
触发器 529
创建数据库 514
概述 504
关于 504
过程 529
结果集 534
连接 527
批处理语句 531
外连接 301
外连接限制 302
尾随空白 515
Transact-SQL 触发器概述 529
Transact-SQL 存储过程概述 529
Transact-SQL 过程语言概述 529
Transact-SQL 过程中的错误处理 536
Transact-SQL 兼容性 503
设置数据库选项 516
数据库 518
Transact-SQL 批处理语句概述 531
trantest 211
TRUNCATE TABLE 语句
关于 426
tsequal 函数 521
UNION ALL
查询执行算法 452
UNION 操作
组合查询 270
UNION 运算
NULL 274
规则 271
UNKNOWN
NULL 248
UNLOAD TABLE 语句
关于 610
UNLOAD 语句
关于 611
UPDATE 语句
更新过程中的锁定 153
使用 422
UUID
缺省列值 93
生成 159
与全局自动增量比较 93
ValuePtr 参数
关于 121
Watcom-SQL
编写兼容的 SQL 语句 523
关于 504
WHERE 子句
GROUP BY 子句 263
NULL 值 247
UPDATE 语句 423
关于 238
连接 292
性能 200, 467
与 HAVING 比较 264
子查询 385
字符串比较 242
WHERE 子句中的匹配字符串 242
WHILE 语句
语法 724
Windows CE
创建数据库 35
Windows 性能监控器
关于 207
WITH CHECK OPTION 子句
关于 62
WITH EXPRESS CHECK
性能 180
WITH 子句 331
优化程序算法 443
XML

从 Interactive SQL 中导出数据 548
导入为关系数据 550
定义 546
将关系数据导出为 548
使用 DataSet 对象导出数据 548
使用 DataSet 对象导入 556
使用 FOR XML AUTO 565
使用 FOR XML EXPLICIT 569
使用 FOR XML RAW 563
使用 OPENXML 导入 550
以 XML 格式从关系数据中获取查询结果 558
以 XML 格式获取查询结果 559
在 Adaptive Server Anywhere 数据库中使用 545
在关系数据库中存储 547
XML 和 Adaptive Server Anywhere 546
XML 数据类型
使用 547
xml 指令
使用 579
XMLAGG 函数
使用 582
XMLCONCAT 函数
使用 583
XMLELEMENT 函数
使用 584
XMLFOREST 函数
使用 586
XMLGEN 函数
使用 587
<
比较运算符 239

A

安全
过程 701
隐藏对象 765
安装
jConnect 元数据支持 42
按照集进行分组
查询执行算法 451
按照集进行排序分组
查询执行算法 450
按照集进行散列分组
查询执行算法 450

B

半连接
查询执行算法 443, 445
保存
结果集 639
保存点
过程和触发器 754
嵌套和命名 115
事务内 114
保存事务结果 113
保留表
在外连接中 297
保留字
远程服务器 673
报告函数
关于 377
示例 377, 378
被锁定的表
访问计划中的项 486
比较
NULL 值 247
尾随空白 239
比较测试
子查询 389
比较运算符
NULL 值 247
符号 239
子查询 407
编辑

表数据 49
数据库对象的属性 38
编辑断点条件 782
编写 EXPLICIT 模式的查询 571
编写过程的提示 755
编写兼容的 SQL 语句 523
编写兼容的查询 524
编写可移植的 SQL 的一般原则 523
便携式计算机
复制数据库 162
变更
表 46, 47
触发器 718
过程 704
列 46, 47
视图 64
变更远程服务器 650
变量
SELECT 语句 526
SET 语句 526
Transact-SQL 535
本地 526
调试 778
赋值 526
标量集合 253
标识符
区分大小写 518
唯一性 519
限定 227
在域中使用 102
标准 另请参见 SQL 标准 3
标准和兼容性 另请参见 SQL 标准 3
标准输出
重定向到文件 614
表
CHECK 约束 99
Transact-SQL 523
编辑数据 49
变更 46, 47
创建 44
从 Sybase Central 卸载 613
代理 655
导出 618
导出数据 627
导入 604
复制 57
复制行 420
工作表 201
管理表约束 98
管理外键 51, 52, 53
管理主键 49, 50
将键添加到 50, 52, 53
结构 46
列出远程 651
列名 29
删除 48
使用 44
使用 SQL 创建代理表 657, 658
属性 29
碎片 209
位图 456
系统表 58
显示来自其它表的引用 52
相关名 237
远程访问 644
约束 30
在 Sybase Central 中创建代理表 656
在 Sybase Central 中显示主键 50
在查询中命名 237
整理数据库中单个表的碎片 211
整理数据库中所有表的碎片 210
组读取 456
浏览数据 49
表达式
NULL 值 249
表达式 SQL

访问计划中的项 489
表大小
关于 456
表大小和页面大小 456
表的表达式
键连接 320
它们是如何连接的 287
语法 284
表名称
标识 227
过程和触发器 755
过程中全限定的 755
表扫描
磁盘分配和性能 455
关于 440
散列表扫描算法 441
表碎片
关于 209
别名
关于 231
相关名 237
并发 114
DDL 语句 160
ISO SQL/92 标准 117
不一致 117
复制 162
关于 117, 159
锁的类型 147
锁定的工作方式 146
性能 114
一致性 117
优点 114
主键 159
并发事务
阻塞 124, 138
并发性
使用索引提高 129
提高 129
提高和索引 157
并行索引扫描
扫描表 441
算法 441
捕获负载
索引顾问 74, 77
不大于
比较运算符 239
不等于
比较运算符 239
不可序列化的事务调度
影响 128
不相关子查询
关于 405
不小于
比较运算符 239
不一致
ISO SQL/92 标准 117
不可序列化调度的影响 128
非可重复的读取示例 135
幻像行 117, 138, 150, 156
使用锁定避免 146
锁定的实际含义 141
脏读 117, 149
脏读教程 131
不一致非可重复的读取 117, 134, 149
不支持远程数据的功能 673
部分索引扫描
关于 463
部件激增问题 343

C

参照完整性
被客户端应用程序破坏 107
丢失 107
动作 108
工具 87

关于 86
检查 108
介绍 88
列缺省值 90
实施 105, 106
外键 107
系统表中的信息 110
遗孤 152
约束 88, 96
在提交时校验 152
主键 540
参照完整性动作
由系统触发器实施 108
测试
数据库设计 17, 26
层次数据结构 341, 343
插入过程中的锁定 151
插入数据
BLOB 420
INPUT 语句 600
INSERT 语句 602
到所有列 417
列数据 INSERT 语句 418
缺省值 418
使用 INSERT 417
使用 SELECT 419
添加空值 418
约束 418
插入文档和图像 420
查看
表数据 49
查看数据 66
查看隔离级别 122
查看过程分析数据
Sybase Central 216
查询
Transact-SQL 524
从表中选择数据 225
导出 614
访问计划 482
公用表表达式 331
关于 226
集合运算 270
前缀 357
绕过优化程序 438
优化 428, 429
查询的一般问题
远程数据访问 674
查询的优化
读取访问计划 482
关于 428
假定 432
将子查询重写为 EXISTS 谓语 434
查询分析
远程数据访问 669
查询规范化
远程数据访问 669
查询过程中的锁定 149
查询结果
导出 614
查询结果排序 200
查询结果中的字符串 232
查询性能
读取访问计划 482
查询优化
步骤 437
绕过优化程序 438
查询优化与执行 427
查询预处理
远程数据访问 669
查询执行算法 439
Bloom 过滤器 454
IN 列表 442
UNION ALL 452
按照集进行分组 451
按照集进行排序分组 450

按照集进行散列分组 450
半连接 443, 445
并行索引扫描 441
窗口函数 454
单行 XML 分组依据 451
单行分组依据 451
反半连接 446
访问计划中使用的缩写 482
非重复排序 448
非重复散列 448
非重复索引 448
分组算法 449
过滤器和预过滤器 453
合并连接 447
合并排序 452
激增 454
聚簇散列分组依据 450
块嵌套循环连接 444
连接 443
排序分组依据 450
排序和联合 451
排序块连接 444
派生表 442
嵌套块连接 444
嵌套循环半连接 443
嵌套循环连接 443
散列半连接 445
散列表扫描 441
散列不存在 446
散列反半连接 446
散列分组依据 449
散列连接 444, 445, 446
顺序表扫描 440
索引分组依据 451
索引扫描 439
锁 453
行限制 454
重复排除 447
查询自身阻塞
远程数据访问 675
尝试使用 Adaptive Server Anywhere 的压缩功能 181
常数表达式缺省值 95
成本
索引顾问建议 79, 80
成本模型
[索引顾问] 使用 78
关于 429
程序变量
公用表表达式 336
冲突
事务阻塞 124, 138
锁定 124
锁定之间 149
循环阻塞 125
抽取
SQLRemote 的数据库 630
抽取数据库向导
关于 630
初始高速缓存大小 192
初始化
数据库 33
初始化实用程序 [dbinit]
使用 36
触发器
ROLLBACK 语句 530
Transact-SQL 519, 529
保存点 754
变更 718
查看分析数据 213
查看特定的分析数据 220
查看摘要分析数据 218
创建 715
错误处理 742

递归 530
概述 700
关于 699
结构 728
警告 746
命令分隔符 755
日期 756
删除 720
时间 756
使用 714
异常处理程序 747
优点 701
游标 738
语句级 529
允许的 SQL 语句 728
执行 718
执行权限 720
触发器创建向导
使用 715
传输数据
INPUT 语句 600
INSERT 语句 602
LOAD TABLE 语句 602
OUTPUT 语句 609
UNLOAD TABLE 语句 610
UNLOAD 语句 611
导入 597
导入 / 导出的情况 594
导入和导出 594
导入向导 599
关于 593
性能 595
性能提示 598
转换错误 604
窗口函数
查询执行算法 454
关于 379
示例 380
创建
Windows CE 数据库 35
表 44
触发器 715
从 SQL 创建数据库 35
从命令行创建数据库 36
过程 702
具有外部调用的过程和函数 758
列缺省值 90
视图 60
数据类型 101, 102
索引 70
用户定义的函数 710
域 101, 102
远程过程 702
创建兼容表 523
创建数据库 33
Sybase Central 34
创建数据库向导
创建与 Transact-SQL 兼容的数据库 514
使用 34
创建外部登录 653
创建与 Transact-SQL 兼容的数据库 514
创建远程服务器 646
创建远程过程 664
磁盘访问成本模型
关于 429
从 Interactive SQL 中将关系数据导出为 XML 548
从 PowerBuilder DataWindows 访问远程数据 643
从 Transact-SQL 过程中返回结果集 534
从表中删除所有行 426
从表中选择所有列 229
从表中选择特定的列 230
从过程调用外部库 758
从过程返回多个结果集 735
从过程返回结果 732

从过程返回结果集 734
从过程返回可变结果集 736
从支持 ODBC 的应用程序设置隔离级别 120
存储触发器
  查看分析数据 213
存储过程
  Transact-SQL 存储过程概述 529
  查看分析数据 213
  查看特定的分析信息 219
  查看摘要分析数据 217
  调试 771
  公用表表达式 336
  在 FROM 子句中使用 237
存储过程的自动转换 532
存储过程语言
  概述 529
存储值
  公用表表达式 338
存在测试
  关于 397
  取非 398
错误
  Transact-SQL 536, 538
  过程和触发器 742
  转换 604
错误处理
  ON EXCEPTION RESUME 744
  过程和触发器 742

D

打开 Sybase Central 性能监控器 205
大小写
  排序顺序 267
大于
  比较运算符 239
  范围说明 241
大于或等于
  比较运算符 239
代理表
  创建 644, 656, 657, 658
  导入数据 603
  关于 644, 655
  位置 655
代理表创建向导
  使用 656
单独的主键和外键索引 199
单行 XML 分组依据
  查询执行算法 451
单行分组依据
  查询执行算法 451
当前日期和时间缺省值 92
导出
  ASE 兼容性 640
  表 618
  查询结果 614
  关系数据以 XML 548
  介绍 608
  空值 618
  模式 627
导出表
  关于 618
  模式 627
导出工具 609
  dbunload 实用程序 612
  Interactive SQL 向导 609
  OUTPUT 语句 609
  UNLOAD TABLE 语句 610
  UNLOAD 语句 611
导出视图
  关于 618
导出数据
  dbunload 实用程序 612
  Interactive SQL 向导 609
  OUTPUT 语句 609
  UNLOAD TABLE 语句 610

UNLOAD 语句 611
备份数据库 596
工具 609
关于 594, 608
介绍 593
模式 627
注意事项 608
导出数据库
使用 616
导入
ASE 兼容性 640
XML 文档作为关系数据 550
非匹配表结构 606
工具 598
介绍 597
空值 606
使用 DataSet 对象将 XML 文档作为关系数据 556
使用 OPENXML 将 XML 文档作为关系数据 550
使用临时表 82
导入 XML 文档作为关系数据 550
导入工具
INPUT 语句 600
INSERT 语句 602
Interactive SQL 导入向导 599
LOAD TABLE 语句 602
代理表 603
关于 598
导入过程中的转换错误 604
导入和导出数据
关于 593
导入数据
DEFAULTS 选项 606
INPUT 语句 600
INSERT 语句 602
LOAD TABLE 语句 602, 604
包含 NULL 值 606
备份数据库 596
从其它数据库 603
代理表 603
导入 / 导入的情况 594
导入向导 599
到数据库 597
非匹配表结构 606
工具 598, 604
关于 594
介绍 593
临时表 606
性能 595
性能提示 598
注意事项 597
转换错误 604
导入数据的性能提示 598
导入向导
关于 598
使用 599
等待
访问已被锁定的行 138
校验参照完整性 152
等待访问被锁定的行
死锁 124
等号运算符
比较运算符 239
等幂函数
定义 480
等值连接
关于 291
笛卡儿积 294
递归查询
限制 341
递归子查询
部件激增问题 343
多个集合级别 336
关于 331
数据类型声明 347
最短距离问题 349

典型的不一致类型 117
调度
  不可序列化的影响 128
  可序列化 127
  可序列化性的影响 128
  可序列化与较早释放锁定 156
  两阶段锁定 155
调试
  Java 775
  编译类 775
  存储过程 771
  断点 778
  功能 768
  关于 767
  简介 768
  教程 771, 775
  局部变量 778
  权限 769
  要求 769
调试程序
  调试 Java 类 775
  调试存储过程 771
  功能 768
  关于 768
  检查变量 783
  教程 770
  连接 770
  启动 770
  入门 770
  使用断点 780
  使用连接 785
  显示源代码 776
  要求 769
调试数据库中的逻辑 767
调用堆栈
  调试程序 783
调用过程 705
调用用户定义的函数 711
定量比较测试 408
  子查询 391
定向图形 349
丢失参照完整性 107
动态调整高速缓存大小
  关于 191
  关于 UNIX 194
  关于 Windows 193
动作
  CASCADE 108
  RESTRICT 108
  SET DEFAULT 108
  SET NULL 108
逗号
  表的表达式列表 294
  星形连接 306
  在连接表的表达式时 320
独立于语法的优化 428
读取访问计划 482
  关键统计信息 495
读实体关系图 8
读锁定
  防插入锁定 154
  关于 147
断点
  单个连接 781
  单个用户 781
  调试 778
  关于 780
  计数 782
  禁用 781
  启用 781
  设置 780
  条件 782
  状态 781
断开连接
  从数据库 37

其他用户与数据库的连接 37
对表进行访问
索引扫描和顺序扫描 439
对表使用 CHECK 约束 97
对插入的行的磁盘分配 455
对查询结果进行汇总、分组和排序 251
对列使用 CHECK 约束 96
对象
可锁定的对象 146
隐藏 765
多对多关系
定义 7
解析 23, 25
多个事务
并发 114
多个数据库
连接 662

E

二进制大对象
插入 420
关于 29

F

发布
数据复制和并发 162
发现可利用的条件 478
法则
两阶段锁定 155
反半连接
查询执行算法 446
反馈
提供 xvii
文档 xvii
反身关系 9
返回类型
外部函数 764
返回值
过程 536
范式
第二范式 21
第三范式 22
第一范式 19
规范化数据库设计 17
范围查询 241
方向
访问计划中的项 487
防插入
锁定 147, 154
访问计划
Interactive SQL 499
SQL 函数 499
打印 493
读取 482
高速缓存 436
关于 428
简单文本计划 490
上下文相关帮助 493
缩写 482
统计信息的解释 485
图形式计划 493
详细文本计划 491
自定义 493
访问计划高速缓存 436
访问计划中使用的缩写 482
访问远程数据 641
基本概念 644
访问远程数据的基本概念 644
非 ANSI 连接 289
非可重复的读取
隔离级别 117, 156
关于 117
教程 134
示例 135

非排它锁 147
非脏读
  教程 131
非重复排序
  查询执行算法 448
非重复散列
  查询执行算法 448
非重复索引
  查询执行算法 448
分号
  命令分隔符 755
分散读取
  性能 185
分析
  了解 78
  索引顾问 78
分析负载
  索引顾问 74
分析数据库过程
  关于 213
分析信息
  触发器 220
  存储过程和函数 219
  事件 219
  系统触发器 220
分组算法
  查询执行算法 449
符号
  字符串比较 242
符合 SQL 标准 另请参见 SQL 标准 3
服务器
  启动未连接的数据库 42
  使用性能监控器绘图 204
服务器端装载 595
服务器功能
  远程数据访问 670
服务器和数据库
  兼容性 507
服务器类
  asajdbc 680
  asaodbc 684
  asejdbc 680
  aseodbc 684
  db2odbc 687
  msodbc 691
  ODBC 683, 693
  oraodbc 689
  定义 644
  关于 645
服务器状态
  索引顾问 81
复合索引
  关于 460
  列顺序的效果 461
  散列值 465
复合语句
  声明 725
  使用 725
  原子 726
复合语句中的声明 725
复杂的外连接 299
复制
  Sybase Central 中的数据库对象 83
  表 57
  并发 162
  并发问题 159
  过程 706
  使用 INSERT 复制数据 420
  视图 61
  重建数据库 624
复制和并发 162
复制数据库
  复制数据和并发 162
负载
  捕获 74, 77
  分析 74
  了解 77

索引顾问 74
已定义 73
负载捕获
停止 76
暂停 76

G

概念数据库模型
定义 5
概念数据模型
关于 3
高速缓存
UNIX 194
UNIX 上的 Java 应用程序 194
Windows 95/98 193
Windows NT 193
初始、最小和最大大小 192
动态调整大小 191
读取次数 / 命中次数比率 496
访问计划 436
关于 191
加密数据库需要更大的高速缓存 175
监控大小 195
性能 175
用户定义的函数 480
预热 196
子查询 480
高速缓存大小
UNIX 194
UNIX 上的 Java 应用程序 194
Windows 95/98 193
Windows NT 193
初始、最小和最大大小 192
监控 195
页面大小 456
高速缓存预热
关于 196
隔离级别
ODBC 120
查看 122
更改 119
关于 116
级别 0 的实施 149
级别 1 的实施 149
教程 131
每个级别的典型事务 128
设置 119
设置缺省值 119
提高级别 2 和 3 的并发性 129
选择 127
选择隔离级别的教程 137
与典型的不一致 117, 138, 141, 156
与典型事务 128
在级别 2 实施 150
在级别 3 实施 150
在事务内更改 122
隔离级别 0
SELECT 语句锁定 149
示例 131
隔离级别 1
SELECT 语句锁定 149
示例 134
隔离级别 2
SELECT 语句锁定 150
示例 138, 141
隔离级别 3
SELECT 语句锁定 150
示例 140
隔离级别和一致性 116
各个隔离级别的典型事务 128
更改隔离级别 119
更改数据
UPDATE 语句 422
关于 415

权限 416
使用多个表更新数据 423
更新过程中的锁定 153
更新列统计信息 431
更新选项卡
索引顾问建议 80
工具
导出数据 609
导入数据 598
卸载数据 612
重建数据库 626
重载数据 626
装载数据 604
工作表
查询处理 201
关于 201
性能提示 177
公用表表达式
部件激增问题 343
查看层次数据结构 341
常见应用 336
存储常量集 338
递归限制 341
递归中的数据类型 347
多个集合级别 336
关于 331
允许位置 334
最短距离问题 349
共享对象
从过程调用 758
共享锁与排它锁 148
故障排除
ANY 运算符 392
死锁 125
关闭自动提交模式 180
关键字
HOLDLOCK 526
NOHOLDLOCK 526
远程服务器 673
关系
定义 5
多对多 7
反身 9
更改为实体 10
关于 6
基数 7
角色 8
解析 23
强制与可选 8
选择 12
一对多 7
一对一 7
关系数据
导出为 XML 548
关于 OLAP 操作 357
管理角色
ASE 509
管理事务 668
管理外键 51
管理主键 49
规范化
关于 17
性能优点 176
规范化表结构 176
规则
Transact-SQL 508
归递表
算法 443
过程
EXECUTE IMMEDIATE 语句 752
Transact-SQL 532
Transact-SQL 概述 529
安全 701
保存点 754
编写提示 755
变更 704

表名称 755
参数 729, 730
创建 702
错误处理 536, 538, 742
调用 705
多个结果集 735
返回结果 707, 732
返回值 536
复制 706
概述 700
关于 699
结构 728
结果集 708, 734
结果集权限 734
警告 746
可变结果集 736
命令分隔符 755
缺省的错误处理 742
日期 756
删除 706
删除远程过程 665
时间 756
使用 702
使用游标 738
添加远程过程 664
外部函数 758
验证输入 756
异常处理程序 747
优点 701
游标 738
允许的 SQL 语句 728
在 FROM 子句中使用 237
转换 532
过程创建向导
关于 702
使用 702
过程分析
触发器 218, 220
存储过程和函数 217, 219
过程摘要 220
事件 217, 219
系统触发器 218, 220
用 SQL 禁用 215
用 SQL 启用 213
用 SQL 清除 215
用 SQL 重置 214
有关单个过程的信息 218, 222
在 Interactive SQL 中查看数据 220
在 Sybase Central 中查看数据 213, 216
在 Sybase Central 中禁用 215
在 Sybase Central 中启用 213
在 Sybase Central 中清除 215
在 Sybase Central 中重置 214
摘要数据 216
过程和触发器的结构 728
过程和触发器的优点 701
过程和触发器中的错误和警告 742
过程和触发器中的事务和保存点 754
过程和触发器中警告的缺省处理 746
过程和触发器中缺省的错误处理 742
过程语言
概述 529
过滤器
查询执行算法 453
过滤器和预过滤器 453

H

含统计信息的图形式计划
Interactive SQL 499
SQL 函数 499
查看 495
关于 493
函数
TRACEBACK 743

tsequal 521
查看分析数据 213
查看特定的分析数据 219
查看摘要分析数据 217
等幂型 480
高速缓存 480
确定型 480
外部 758
用户定义的 710
何时创建索引 458
何时使用索引 68
合并连接
查询执行算法 447
合并排序
查询执行算法 452
候选索引
关于 75
索引顾问 75
换行符
SQL 227
幻像
行 117, 150, 156
幻像锁 141
幻像行
教程 138
与隔离级别 117, 141, 156
恢复
导入 / 导出 596
回退日志
保存点 115
数据恢复 596
回退事务 113
汇总值
关于 252
绘图
使用性能监控器 204

J

基表 44
基础假定
优化程序 432
基数
访问计划中的项 487
关系和 7
基于 JDBC 的服务器类 679
基于 NULL 对列进行测试 247
基于成本的优化 428
激增
查询执行算法 454
集合
访问计划中的项 489
集合成员资格测试 411
=ANY 395
取非 396
集合函数
Adaptive Server Enterprise 兼容性 278
ALL 关键字 252
DISTINCT 关键字 252, 255
GROUP BY 子句 258
NULL 256
order by 和 group by 268
标量集合 253
多个级别 336
关于 252
矢量集合 258
数据类型 255
外部引用 253
集合运算
NULL 274
关于 270
规则 271
技术支持
新闻组 xvii
计划
Interactive SQL 499

SQL 函数 499
打印 493
高速缓存 436
简单文本计划 490
上下文相关帮助 493
详细文本计划 491
自定义 493
计划中的常见预计值 485
计划中使用的常见统计信息 485
计算 SELECT 列表中的值 233
计算列
INSERT 语句 56
UPDATE 语句 56
触发器 56
创建 54
使得使用函数的查询成为可优化搜索查询 468
索引 71
限制 57
重新计算 57
记录 SQL 语句 41
记住在过程内分隔语句 755
加号运算符
NULL 值 249
加密
高速缓存大小 175
隐藏对象 765
监控查询性能 211
监控高速缓存大小 195
监控和提高性能 167
监控器
打开 Sybase Central 性能监控器 205
配置 Sybase Central 性能监控器 206
性能监控器概述 204
监控数据库性能 203
监控性能
读取访问计划 482
访问计划中使用的缩写 482
用于测量查询的工具 211
兼容 SQL 标准 另请参见 SQL 标准 3
兼容性
ASE 504
GROUP BY 子句 278
非 ANSI 连接 289
服务器和数据库 507
区分大小写 518
使用 ASE 导入 / 导出 640
输出 NULL 617
为实现 Transact-SQL 兼容性设置选项 516
检查变量
调试程序 783
检查是否需要更改命令分隔符
过程 755
检查文件、表和索引碎片 185
检查约束
选择 30
在域中使用 102
检索一个查询中的前几行 267
简要计划
Interactive SQL 499
SQL 函数 499
关于 490
减少客户端和服务器之间的请求数 181
减小锁定的影响 129
键
性能 198
指派 18
键类型
访问计划中的项 487
键连接
ON 短语 290
表的表达式 320
表的表达式列表 322
不包含逗号的表的表达式 320
不包含逗号的列表和表的表达式 324
关于 316
规则 328

具有 ON 短语 317
如果有多个外键 317
视图和派生表 326
建议
索引顾问 79
建议的页面大小 465
将 group by 与集合函数一起使用 258
将 WHERE 子句中的子查询转换为连接 406
将本机语句发送到远程服务器 663
将不同的文件放置在不同的设备上 177
将参数传递给过程 730
将参数传递给函数 730
将参数传递给外部函数 763
将查询结果划分为组 258
将多对多关系更改为实体 10
将关系数据导出为 XML 548
将集合函数与 distinct 一起使用 255
将视图与 Transact-SQL 外连接一起使用 303
将输出写入文件 639
将外部连接转换为内部连接 476
将谓语推入包含 GROUP 或 UNION 的视图 472
将值插入特定的列 418
将值插入行的所有列 417
将子查询重写为 EXISTS 谓语 434
降序
ORDER BY 子句 266
交叉连接 294
交错执行事务 127
交换空间
数据库高速缓存 194
脚本
创建命令文件 637
关于命令文件 637
在 Interactive SQL 中运行 637
装载命令文件 638
角色
ASE 509
定义 8
角色名
关于 317
教程
非可重复的读取 134
隔离级别 131
幻像锁 141
幻像行 138
锁定的含义 141
脏读 131
较早释放锁定 155
例外 156
事务 128
节省时间的策略
导入数据 598
结果集
Transact-SQL 534
保存到文件 639
多个 735
过程 708, 734
可变 736
权限 734
限制行数 267
结束事务 113
解析
关系 23
禁用断点
关于 781
启用 781
禁用过程分析
SQL 215
Sybase Central 215
警告
过程和触发器 746
局部变量
调试程序 783
局部临时表
关于 82
聚簇散列分组依据
查询执行算法 450

聚簇索引
使用 69
索引顾问建议 79, 81
具有 group by 的查询是如何执行的 260
具有多个列的 group by 262
决策支持
隔离级别 128

K

可扩展标记语言 请参见 XML ix
可锁定的对象 146
可序列化调度
关于 127
两阶段锁定 155
影响 128
与较早释放锁定 156
可选关系 8
可选外键 106
可移植的 SQL 523
可以锁定的对象 146
可优化搜索谓语
关于 467
可重复的读取 117, 134, 149
客户端装载 595
空值
Transact-SQL 外连接 303
输出 617
空值是如何影响 Transact-SQL 外连接的 303
控制语句
列表 724
口令
Lotus Notes 696
区分大小写 518
块嵌套循环连接
查询执行算法 444
括号
UNION 运算符 270
在算术语句中 234

L

类
远程服务器 677
理解复杂外连接 299
联邦信息处理标准发布兼容 另请参见 SQL 标准 3
联合
查询执行算法 451
联机分析处理
CUBE 操作 368
GROUPING SETS 371
NULL 值 360
ROLLUP 操作 362
功能 356, 374
关于 355
它如何工作 357
小计行 357
联机分析处理 另请参见 OLAP 355
连接
delete、 update 和 insert 语句 288
FROM 子句 284
NATURAL 541
ON 短语 290
Transact-SQL 527
Transact-SQL 外 , 空值和 303
Transact-SQL 外 , 视图和 303
Transact-SQL 外 , 限制 302
WHERE 子句 292
保留的表 297
表的表达式 287
不同数据库中的表 662
查询执行算法 443
等值连接 291
笛卡儿积 294
调试 770
调试程序 785
逗号 294

非 ANSI 连接 289
更新游标 288
关于 284
键 541
键连接 316
将外部连接转换为内部连接 476
将子查询转换为 400
将子查询转换为连接 406
交叉连接 294
连接的表 286
连接排除重写优化 473
连接条件 285
两个表 286
两个以上的表 287
内 296
内连接和外连接 296
派生表 311
启动未连接的数据库 42
嵌套 287
缺省值是 KEY JOIN 286
如何计算内连接 286
数据类型转换 288
搜索条件 291
提供空值的表 297
外 297
星形连接 306
远程 667
远程表 660
重复相关名 306
自动 541
自连接 304
自然连接 312
连接的 GROUPING SETS
关于 373
连接的工作原理 283
连接的兼容性 527
连接多个本地数据库中的表 662
连接兼容数据类型 288
连接两个表 286
连接两个以上的表 287
连接排除重写优化 473
连接算法
关于 443
连接条件
类型 291
连接问题
远程数据访问 674
连接远程表 660
连接运算符
Transact-SQL 527
连接中的重复相关名 （星形连接）306
连接字符串
NULL 249
两阶段锁定
法则 155
关于 155
了解 CUBE 370
了解 group by 260
了解 OLAP 356
了解小计 357
列
CHECK 约束 99
GROUP BY 子句 258
IDENTITY 521
select 语句 230
timestamp 519
变更 46, 47
命名 29
缺省值 90
属性 29
数据类型 29
选择列表 230
约束 30
允许 NULL 值 30
指派数据类型和域 101
列出服务器上的远程表 651

列出远程表上的列 659
列出远程服务器功能 651
列属性
  AUTOINCREMENT 160
  NEWID 159
  生成缺省值 160
列统计信息
  更新 431
列统计信息 请参见 直方图 429
临时表 44
  Transact-SQL 524
  查询处理中的工作表 201
  导入数据 606
  关于 82
  局部和全局 82
  索引 459
临时文件
  工作表 177
逻辑运算符
  HAVING 子句 265
  连接条件 249

M

描述键连接的操作的规则 328
命令
  在 Interactive SQL 中装载 638
命令分隔符
  设置 755
命令文件
  SQL 语句窗格 637
  创建 637
  概述 637
  关于 637
  建立 637
  写入输出 639
  运行 637
  在 Interactive SQL 中打开 637
  执行 637
命名和嵌套保存点 115
命名空间
  触发器 519
  索引 519
模式
  导出 627
目录
  ASE 兼容性 509

N

哪些情况会导致数据无效 86
内部操作
  远程数据访问 669
内部装载 595
内连接
  关于 296
内连接和外连接 296

O

偶尔连接的用户
  复制数据和并发 162

P

排除不必要的 DISTINCT 470
排除不必要的大小写转换 479
排它锁 147
排它锁与共享锁 148
排序
  查询执行算法 451
  用索引 200
排序分组依据
  查询执行算法 450
排序和联合 451
排序块连接

查询执行算法 444
排序顺序
ORDER BY 子句 266
派生表
键连接 326
连接 311
算法 442
外连接 300
在 FROM 子句中使用 237
自然连接 315
配置
Sybase Central 性能监控器 206
配置 Sybase Central 性能监控器 206
批处理
关于 722
控制语句 722
使用 SELECT 语句 757
数据定义语句 722
允许的 SQL 语句 757
允许的语句 757
批处理操作
Interactive SQL 637
批处理模式
Interactive SQL 637
批处理语句
Transact-SQL 概述 531
编写 531
批处理中允许的语句 757
批量操作
数据恢复问题 596
性能 180
批量操作对性能的影响 595
批量装载
性能 595
撇号
字符串 245

Q

启动
索引顾问 76
启动未连接的数据库 42
启用断点
关于 781
启用过程分析
SQL 213
Sybase Central 213
迁移数据库
sa_migrate 存储过程 633
关于 631
数据迁移向导 631
前缀
ROLLUP 操作 366
小计行 366
前缀列
ROLLUP 操作 366
嵌套
连接 287
连接中的派生表 311
外连接 299
嵌套的复合语句和异常处理程序 749
嵌套的子查询
关于 403
嵌套和命名保存点 115
嵌套块连接
查询执行算法 444
嵌套块连接和排序块 444
嵌套循环半连接
查询执行算法 443
嵌套循环连接
查询执行算法 443
强制
外键 106
强制关系 8
强制列唯一性 463
清除过程分析
SQL 215

Sybase Central 215
清除实用程序 [dberase]
使用 37
请求
减少数量 181
请求级记录 请参见 请求记录 ix
请求记录
关于 168
请求日志
关于 168
请求选项卡
索引顾问建议 80
区分大小写
Transact-SQL 兼容性 518
标识符 227, 518
创建与 ASE 兼容的数据库 515
口令 518
数据 518
数据库 518
用户 ID 518
域 518
远程访问 673
取消
外部函数 763
取消请求
远程数据访问 675
取消子查询嵌套 471
权限
ASE 511
触发器 720
调试 769
调用外部函数的过程 758
过程结果集 734
数据修改 416
用户定义的函数 713
全局变量
调试程序 783
全局临时表
导入数据 606
关于 82
全局自动增量
与 GUID 和 UUID 比较 93
缺省值
INSERT 语句和 418
NEWID 93
NULL 94
Transact-SQL 508
常数表达式 95
创建 90
当前日期和时间 92
介绍 87
列 90
事务和锁定 160
用户 ID 92
在 Sybase Central 中创建 91
在域中使用 102
自动增量 92
字符串和数字 94
确保兼容的对象名 519
确保数据完整性 85
确定实体和关系 12
确定型函数
定义 480

R

日期
过程和触发器 756
输入规则 245
搜索 245
日志
回退日志 115
如果客户端应用程序破坏了参照完整性 107
如果客户端应用程序破坏了实体完整性 105

S

散列 B 树索引
  关于 464
散列半连接
  查询执行算法 445
散列表扫描
  算法 441
散列不存在
  查询执行算法 446
散列反半连接
  查询执行算法 446
散列分组依据
  查询执行算法 449
散列连接
  查询执行算法 444, 445, 446
散列映射
  查询执行算法 454
散列值
  索引 464
删除
  CHECK 约束 99
  表 48
  触发器 720
  过程 706
  列缺省值 91
  视图 66
  数据库文件 36
  数据类型 103
  索引 73
  用户定义数据类型 103
  域 103
删除过程中的锁定 153
删除连接
  远程数据访问 675
删除数据
  DELETE 语句 425
  TRUNCATE TABLE 语句 426
  关于 415
删除统计信息 205
删除外部登录 654
删除远程服务器 649
删除远程过程 665
设备
  管理 508
设计过程 12
设计数据库 3
  步骤 12
  概念 5
  关于 3
设计数据库表属性 29
设计详细信息窗格
  关于 41
设置断点
  调试程序 780
设置隔离级别 119
设置数据库对象的属性 38
设置数据库选项 39
深度
  访问计划中的项 487
生成
  唯一键 159
  物理数据模型 23
生成的连接和 ON 短语 290
生成的连接条件
  关于 285
升级
  数据库文件格式 621
升级数据库 621
升级数据库向导
  安装 jConnect 元数据支持 42
升序
  ORDER BY 子句 266
时间
  过程和触发器 756
实际值
  NULL 249
实施
  锁定 156

实施参照完整性 106
实施实体完整性和参照完整性 105
实施锁定的方式 156
实体
  定义 5
  关于 5
  实施完整性 105
  属性 5
  选择 12
实体关系图
  读 8
  关于 5
实体完整性
  被客户端应用程序破坏 105
  介绍 88
  主键 540
实现查询结果
  查询处理 201
实现建议
  索引顾问 81
矢量积 294
矢量集合 258
使用 count (*) 255
使用 CREATE TABLE 语句创建代理表 658
使用 CUBE 368
使用 DataSet 对象导入 XML 556
使用 DataSet 对象将关系数据导出为 XML 548
使用 FOR XML AUTO 565
使用 FOR XML EXPLICIT 569
使用 FOR XML RAW 563
使用 GROUPING SETS 371
使用 OLAP 355
使用 OLAP 函数 374
使用 OPENXML 导入 XML 550
使用 RETURN 语句返回值 732
使用 ROLLUP 362
使用 SELECT 添加新行 419
使用 SQL 创建代理表 657
使用 Sybase Central 转换存储过程 532
使用 WHERE 子句用于连接条件 292
使用 WITH CHECK OPTION 子句 62
使用表 44
使用表和列约束 96
使用代理表 655
使用调试程序的要求 769
使用断点 780
使用复合语句 725
使用高速缓存提高性能 191
使用过程、触发器和批处理 699
使用集合函数汇总查询结果 252
使用键提高查询性能 198
使用连接的 GROUPING SETS 373
使用列缺省值 90
使用逻辑运算符连接条件 249
使用批量操作方法 180
使用事务 112
使用事务和隔离级别 111
使用适当的页大小 184
使用视图 59, 61
使用数据库 33
使用数据库对象 31
使用索引 68
使用外部登录 653
使用外键提高查询性能 198
使用域 101
使用远程服务器 646
使用远程过程调用 664
使用主键提高查询性能 198
使用子查询 383
始终使用事务日志 175
示例
  非可重复的读取 134
  幻像锁 141
  幻像行 138
  锁定的含义 141
  脏读 131
示例数据库
  asademo.db 模式 282
事件
  查看分析数据 213
  查看特定的分析数据 219

查看摘要分析数据 217
事务
保存点 114
并发 114
并发复制 162
关于 112
管理 668
过程和触发器 754
开始 112
使用 112
数据修改 416
死锁 125
完成 113
一次多个 114
远程数据访问 667
之间的干扰 124, 138
子事务和保存点 114
阻塞 124, 125, 138
事务处理
调度 127
调度的影响 128
基于事务日志的复制 163
可序列化调度 127
两阶段锁定 155
性能 114
阻塞 124, 138
事务调度 127
影响 128
事务管理
和远程数据 667
事务管理的限制 668
事务和隔离级别 111
事务和数据修改 416
事务排序 127
事务日志
复制数据库 624
数据复制中的角色 163
数据恢复 596
性能 175
事务之间的干扰 124, 138
事务阻塞
关于 124
视图
FROM 子句 237
SYSCOLUMNS 515
SYSINDEXES 515
创建 60
导出 609
复制 61
更新 61
公用表表达式 331
检查选项 62
键连接 326
删除 66
使用 59, 61
外连接 300
修改 64
自然连接 315
浏览数据 66
视图创建向导
使用 60
收益
索引顾问建议 79
输出
NULL 617
导出表 618
导出查询结果 614
导出工具 609
导出数据 608
导出数据库 616
输出数据
保存到文件 639
导入和导出 594
到文件 611
输出重定向 614
输入

导入表 604
导入数据 597
输入数据
INPUT 语句 600
INSERT 语句 602
LOAD TABLE 语句 602
导入和导出 594
属性
SQLCA.lock 120
定义 5
设置所有数据库对象属性 38
选择 15
以 XML 格式获取查询结果 559
属性表 38
数据
查看 49
导出 608, 609
导出为 XML 548
导入 597
导入和导出 594
区分大小写 518
搜索 69
添加、更改和删除 415
完整性和正确性 127
无效 86
修改数据所需的权限 416
一致性 117
数据定义语句 请参见 DDL ix
数据定义语言 请参见 DDL ix
数据恢复
导入和导出 596
数据库
Transact-SQL 兼容性 514
安装 jConnect 元数据支持 42
查看和编辑属性 38
抽取 630
初始化 33
创建 33
从 SQL 初始化 35
从 SQL 创建 35
从 Sybase Central 初始化 34
从 Sybase Central 创建 34
从命令行创建 36
从命令行消除 37
从数据库断开连接 37
存储 XML 547
导出 616
导入 XML 550
规范化 17
启动未连接的数据库 42
迁移到 ASA 631
清除 36
区分大小写 515, 518
删除 36
设计 3
设计概念 5
设置统一数据库 40
设置选项 39
升级数据库文件格式 620
使用 33
使用对象 31
事务日志 33
为 Windows CE 创建 35
文件兼容性 33
显示系统表 58
显示系统对象 40
卸载 616
卸载和重装 622, 624, 629
验证设计 26
重建 622
重装 628
数据库对象
编辑属性 38

在 Sybase Central 中复制 83
数据库管理员
角色 510
数据库过程
查看分析数据 213
数据库内容如何更改 87
数据库设计概念 5
数据库统计信息
关于 207
数据库文件
碎片 185, 209
性能 177
数据库线程
被阻塞 125
数据库选项
索引顾问 81
为实现 Transact-SQL 兼容性设置 516
数据库页
索引顾问建议 79
数据类型
EXCEPT 运算 270
INTERSECT 运算 270
SQL 和 C 763
Transact-SQL timestamp 519
UNION 运算 270
创建 101, 102
递归子查询 347
集合函数 255
删除 103
选择 29
用户定义的 101
远程过程 665
指派给列 101
数据模型
关于 3
数据模型规范化 17
数据迁移向导
关于 631
使用 631
数据输入
隔离级别 128
数据完整性
不可序列化调度的影响 128
丢失 107
工具 87
关于 86
检查 108
列缺省值 90
列约束 30
确保 85
实施 105
系统表中的信息 110
约束 88, 96
数据修改的权限 416
数据修改语句
关于 416
数据一致性
ISO SQL/92 标准 117
幻像行 117, 138, 150, 156
可重复的读取 117, 134, 149
两阶段锁定 155
使用锁定确保 146
锁定的实际含义 141
脏读 117, 149
脏读教程 131
正确性 127
数据源
外部服务器 683
数据组织
物理 455
双向复制 162
双引号
字符串 245
顺序表扫描
磁盘分配和性能 455
关于 440

顺序扫描
磁盘分配和性能 455
关于 440
死锁
关于 124
事务阻塞 125
原因 125
诊断 125
死锁报告
关于 125
算法
查询执行 439
非重复索引 448
算术
表达式和运算符优先级 234
运算 252
碎片
表 209
关于 209
索引 211
文件 209
文件、表和索引 185
碎片整理
关于 209
数据库中的单个表 211
数据库中的所有表 210
硬盘 209
索引
B 树 464
HAVING 子句性能 467
Transact-SQL 519
WHERE 子句性能 467
帮助 170
成本 73
创建 70
关于 458
何时使用 68
候选 75
计算列 71
检查 81
建议的页面大小 465
结构 460
经常被搜索的列 69
聚簇 69
可优化搜索谓语 467
了解 [索引顾问] 77
列顺序的效果 461
临时表 459
散列 B 树 464
散列值 464
删除 73
生成的 458
使用 68
收益 73
碎片 211
索引的类型 464
索引顾问 68, 73, 76
索引提示 69
提高并发性 157
提高性能 459
条目数和页面大小 465
通常可以满足谓语 434
未使用 80
谓语分析 467
相关性 80
校验 71
性能 176
虚拟 75
压缩 B 树 465
叶页 460
优点和锁定 129
优化和 458
索引创建向导

使用 70
索引的类型 464
索引的其它用途 463
索引分组依据
查询执行算法 451
索引顾问
服务器状态 81
负载 74, 77
关于 73
简介 68, 170
建议 79
连接状态 81
了解 73, 77
评估建议 80
启动 76
实现建议 81
停止 76
索引扫描
并行索引扫描 441
并行索引扫描算法 441
关于 439
索引碎片 211
索引提示
使用 69
索引选择性
关于 459
锁
查询执行算法 453
冲突处理 124, 138
典型事务与隔离级别 128
隔离级别 116
类型 147
两阶段锁定 155
排它与共享 148
删除的过程 153
事务阻塞和死锁 124
死锁 125
通过索引减小影响 129
选择隔离级别的教程 137
在 Sybase Central 中查看 146
阻塞 124, 138
锁的类型 147
锁定
不一致与典型的隔离级别 117, 156
插入 147
插入的过程 151
冲突类型 149
读 147, 154
防插入 147, 154
非排它 147
更新的过程 153
关于 146
幻像行与隔离级别 138, 141
级别 0 的实施 149
级别 1 的实施 149
较早释放 128, 155
较早释放读锁定 156
可以锁定的对象 146
排它 147
通过索引减少 463
写 147
遗孤和参照完整性 152
用途 147
在级别 2 实施 150
在级别 3 实施 150
锁定的工作方式 146
锁定的实际含义教程 141
锁定的用途 147

T

特殊 IDENTITY 列 521
特殊并发问题 159
特殊的 Transact-SQL timestamp 列和数据类型 519
提高隔离级别 2 和 3 的并发性 129

提高索引性能 459
提高性能 175
提供空值的表
  在外连接中 297
提示
  性能 175
体系结构
  Adaptive Server 507
添加
  数据到数据库 593, 597
添加、更改和删除数据
  关于 415
添加和删除统计信息 205
添加数据
  BLOB 420
  到所有列 417
  关于 415
  列数据 INSERT 语句 418
  缺省值 418
  使用 INSERT 417
  使用 SELECT 419
  添加空值 418
  约束 418
添加外键向导
  使用 52
条件
  使用逻辑运算符连接 249
通过 Sybase Central 监控数据库统计信息 204
通过 Windows 性能监控器监控数据库统计信息 207
通过客户端应用程序获取数据库统计信息 203
通配符
  LIKE 运算符 243
  字符串比较 242
桶
  直方图 429
统计信息
  从性能监控器中删除 205
  访问计划 482, 485
  更新列统计信息 431
  监控 203
  可用 207
  添加到性能监控器 205
  显示 207
  性能 207
统计信息 另请参见 直方图 429
统计信息存在并正确 433
统一数据库
  设置 40
图标
  手册中使用的 xiv
图像
  插入 420
图形类型
  配置性能监控器 206
图形式计划
  Interactive SQL 499
  SQL 函数 499
  不是所有部分都出现在简单查询中 493
  打印 493
  关于 493
  上下文相关帮助 493
  说明 493
  谓语 497
  自定义 493

W

外部登录
  创建 653
  关于 653
  删除 654
外部登录创建向导
  使用 653
外部调用
  创建过程和函数 758
外部服务器

ODBC 683
外部函数
传递参数 763
返回类型 764
取消 763
原型 760
外部连接
Transact-SQL 301, 527
外部引用
HAVING 子句 386
关于 399
集合函数 253
外部装载 595
外键
参照完整性 107
创建 52, 53
管理 51
键连接 316
角色名 317
强制 / 可选 106
删除 52
生成的索引 458
完整性 540
性能 198
修改 53
在 Sybase Central 中显示 52
外连接
Transact-SQL, 视图和 303
Transact-SQL, 限制 302
复杂 299
关于 297
和连接条件 298
视图和派生表 300
限制 302
星形连接示例 308
完成事务 113
完全比较
关于 459
完全外连接
关于 297
完整性
丢失 107
工具 87
关于 86
检查 108
列缺省值 90
实施 105
使用触发器保持 88
系统表中的信息 110
约束 88, 96
完整性约束属于数据库 86
唯一标识符
表 49
唯一键
生成和并发 159
唯一结果
限制 235
唯一性
用索引强制 463
唯一约束
生成的索引 458
为过程声明参数 729
为前几行或整个结果集进行优化 433
为实现 Transact-SQL 兼容性配置数据库 514
为实现 Transact-SQL 兼容性设置选项 516
维护
性能 175
尾随空白
Transact-SQL 515
比较 239
创建数据库 515
未使用的索引
索引顾问建议 80
未知值
关于 246
位图 29

扫描 456
谓语
访问计划中的项 487
关于 467
性能 467
在访问计划中读取 497
谓语分析
关于 467
文本计划
读取访问计划 490
文档
SQL Anywhere Studio x
插入 420
约定 xii
文件
碎片 185
性能 177
文件碎片
关于 209
无效数据 86
物理数据模型
生成 23
物理数据组织 455

X

膝上型计算机
事务 162
系统安全员
ASE 509
系统表
ASE 509
Transact-SQL 名称冲突 515
视图 67
索引 81
所有者 509
显示 58
有关参照完整性的信息 110
系统表中的视图 67
系统表中的索引 81
系统表中的完整性规则 110
系统触发器
查看各项的分析信息 220
查看摘要分析数据 218
实施参照完整性动作 108
系统对象 40
显示 58
系统管理员
ASE 509
系统过程
tsequal 521
系统目录
ASE 509
系统视图
索引 81
有关参照完整性的信息 110
显式连接条件 285
显式连接条件 （ON 短语）290
显式连接条件的类型 291
显示数据库中的系统对象 40
显示系统表 58
限定
关于 238
限定名
数据库对象 227
限制
远程数据访问 673
限制高速缓存使用的内存 192
限制行数
FIRST 子句 267
TOP 子句 267
线程
在没有可用线程时出现死锁 125
相对收益
索引顾问建议 79
相关名
表名 237

关于 317
星形连接 306
在自连接中 304
相关子查询
关于 398, 405
外部引用 399
详细计划
Interactive SQL 499
SQL 函数 499
关于 491
向导
触发器创建 715
创建数据库 34
代理表创建 656
关于 598
过程创建 702
升级数据库 42
视图创建 60
数据迁移 631
索引创建 70
添加外键 52
消除数据库 36
域创建 101
远程服务器创建 648
远程过程创建 664, 702
消除
CHECK 约束 99
表 48
数据库 36
消除数据库向导
使用 36
消除重复查询结果 235
小计行
GROUPING SETS 371
NULL 值 360
ROLLUP 操作 366
定义 362, 368
构建 366
关于 357
连接的 GROUPING SETS 373
小于
比较运算符 239
范围说明 241
小于或等于
比较运算符 239
校验
XML 580
列约束 30
索引 71
校验表
WITH EXPRESS CHECK 180
效率
节省导入数据的时间 598
使用索引提高 157
提高和锁 129
协议
两阶段锁定 155
写锁定 147
卸载工具
关于 612
卸载数据对话框 613
卸载数据库向导 613
卸载和重装
数据库 622, 624, 629
卸载数据对话框
使用 613
卸载数据库
关于 620
使用 616
以逗号分隔的 ASCII 格式 626
卸载数据库向导
关于 613
使用 616
新闻组
技术支持 xvii

星号 (*)
SELECT 语句 229
星形连接 306
行
删除 456
使用 INSERT 复制 420
小计行 357
选择 238
行的连续存储 455
行限制
查询执行算法 454
行限制计数
访问计划中的项 490
性能
Windows 性能监控器中的统计信息 207
WITH EXPRESS CHECK 180
比较优化程序预计值和实际统计信息 495
表大小和页面大小 456
表碎片 209
测量查询速度 211
读取访问计划 482
分散读取 185
改善 69
高速缓存 175
高速缓存读取次数 / 命中次数比率 496
工作表 201
关于 175
监控 203
减少请求 181
键 198
建议的页面大小 465
批量操作 180
批量装载 595
事务日志 175
数据库设计 176
索引 68, 176
索引碎片 211
提高与锁 129
提示 175
谓语分析 467
文件管理 177
文件碎片 185, 209
选择性 496
压缩 181
页大小 184
预计值来源 496
运行时间的实际值和预计值 496
在 Windows 中监控 207
自动调优 432
自动提交 180
性能监控器
Sybase Central 204
Windows 207
概述 204
配置 206
启动 207
设置间隔时间 206
添加和删除统计信息 205
运行多个版本 207
在 Sybase Central 中打开 205
修改
CHECK 约束 99
列缺省值 91
视图 64
修改和删除 CHECK 约束 99
修改和删除列缺省值 91
虚拟内存
稀有资源 434
虚拟索引
[索引顾问] 分析步骤 78
关于 75
索引顾问 75
选项
BLOCKING 124

DEFAULTS 606
ISOLATION_LEVEL 119
设置数据库选项 39
选择隔离级别 127
选择列表
EXCEPT 语句 270
INTERSECT 语句 270
UNION 语句 270
UNION 运算 271
别名 231
访问计划 485
关于 229
计算列 233
选择列的数据类型 29
选择列名 29
选择性
读取访问计划 495
访问计划中的项 487
在访问计划中读取 497
选择性预计值
使用部分索引扫描 463
在访问计划中读取 497
选择约束 30
循环阻塞冲突 125

Y

压缩
性能 181
压缩 B 树索引
关于 465
延迟参照完整性检查 152
验证
数据库设计 26
验证正确传递了过程输入参数 756
页
对插入的行的磁盘分配 455
页大小
性能 184
页面大小
对插入的行的磁盘分配 455
关于 456
和索引 457, 465
页位置图
访问计划中的项 487
扫描 456
叶页 460
一次多个事务 114
一对多关系
解析 23
一对一关系
定义 7
解析 23
一致性
ISO SQL/92 标准 117
不可序列化调度的影响 128
非可重复的读取示例 135
隔离级别 116
关于 112
幻像行 117, 138, 150, 156
可重复的读取 117, 134, 149
两阶段锁定 155
使用锁定确保 146
锁定的实际含义 141
与典型事务 128
与隔离级别 117, 138, 141, 156
在事务过程中 117
脏读 117, 149
脏读教程 131
正确性和调度 127
遗孤和参照完整性 152
移动数据
导出 608
导入 597
导入和导出 594
关于 593

性能 595
疑难解答
调试类 775
性能 175
远程数据访问 673
以 XML 格式获取查询结果 558
异常
声明 743
异常处理程序
过程和触发器 747
嵌套的复合语句 749
引号
Adaptive Server Enterprise 245
字符串 245
引用
显示来自其它表的引用 52
影响
不可序列化的事务调度 128
事务调度 128
影响优化的假定 432
用 IN 条件测试集合成员资格 395
用户
偶尔连接 162
用户 ID
ASE 511
区分大小写 518
缺省值 92
用户定义的函数
参数 730
创建 710
调用 711
高速缓存 480
关于 710
删除 713
外部函数 758
执行权限 713
用户定义的数据类型
CHECK 约束 98
创建 101, 102
用户定义数据类型 101
删除 103
用冒号来分隔连接策略 490
用某个值替代 NULL 248
用于导入的表结构 606
用于进行远程数据访问的服务器类 677
用于实施完整性约束的 SQL 语句 88
优化
读取访问计划 482
基于成本 428
使用索引 157
优化步骤 437
优化程序
关于 428, 429
假定 432
绕过 438
谓语分析 467
语义转换 469
语义子查询转换 466
作用 428
优化程序的工作原理 429
优化程序的作用 428
优化程序估计值
关于 429
优化过程中的子查询转换 466
优化混合或 OLAP 负载 433
优化目标
访问计划 485
游标
LOOP 语句 738
SELECT 语句 738
不稳定性 118
过程 738
过程和触发器 738
稳定性 118
在连接中更新 288
游标不稳定性 118
游标管理

概述 738
游标稳定性 118
有关基于 JDBC 的类的配置说明 679
有效使用索引 176
右外连接
关于 297
与其它版本 SQL 语言的区别 539
语句
CALL 700, 705, 724, 730
CASE 724
CLOSE 738
COMMIT 726, 754
CREATE DATABASE 507
CREATE DEFAULT 508
CREATE DOMAIN 508
CREATE PROCEDURE 702, 729
CREATE RULE 508
CREATE TABLE 523
CREATE TRIGGER 715
DECLARE 725, 738, 743
DISK 508
DROP DATABASE 507
DUMP DATABASE 507
DUMP TRANSACTION 507
EXECUTE IMMEDIATE 752
FETCH 738
FOR 724
GRANT 511
IF 724
LEAVE 724
LOAD DATABASE 507
LOAD TRANSACTION 507
LOOP 724, 738
MESSAGE 743
OPEN 738
OUTPUT 614
RAISERROR 537, 538
RETURN 732
REVOKE 511
ROLLBACK 530, 754
SELECT 524, 526, 733
SIGNAL 537, 743
WHILE 724
复合 725
记录 41
优化 428
语句的部分直通
远程数据访问 671
语句的完整直通
远程数据访问 670
语句级触发器 529
语义查询转换 466
语义转换
关于 469
语义转换的类型 469
域
CHECK 约束 98
创建 101, 102
区分大小写 518
删除 103
使用 101
使用示例 101
指派给列 101
域创建向导
使用 101
域中的列 CHECK 约束 98
预订
数据复制和并发 162
预计行大小
访问计划中的项 487
预计行数
访问计划中的项 487
预计页数
访问计划中的项 487
预计叶页数

访问计划中的项 487
元数据支持
为 jConnect 安装 42
元素
从关系数据中生成 XML 548
以 XML 格式获取查询结果 559
在数据库中存储 XML 547
原型
外部函数 760
原子复合语句 726
原子事务 112
源代码窗口
设置断点 780
远程表
访问 641
关于 644
列出 651
列出列 659
远程表映射 644
远程服务器
变更 650
创建 646
发送本机语句 663
关于 646
类 677, 678
列出属性 651
删除 649
事务管理 667
外部登录 653
远程服务器创建向导
使用 648
远程过程
创建 702
调用 664
删除 665
数据类型 665
添加 664
远程过程创建向导
使用 664, 702
远程过程调用
关于 664
远程事务管理概述 667
远程数据
位置 655
远程数据访问
Lotus Notes SQL 2.0 695
Microsoft Access 695
Microsoft Excel 694
Microsoft FoxPro 695
PowerBuilder DataWindows 643
SQL Remote 不支持的 673
不支持的功能 673
关于 641
介绍 642
连接名 675
内部操作 669
区分大小写 673
性能限制 642
疑难解答 673
远程服务器 646
直通模式 663
远程数据库
复制 162
约定
文档 xii
约束 98
CHECK 约束 98
介绍 87
列和表 30
唯一约束 98
运算符
NOT 关键字 240
连接条件 249
散列表扫描 441
算术 234

优先级 234
运行
SQL 脚本 637
命令文件 637

Z

在 delete、update 和 insert 语句中使用连接 288
在 Interactive SQL 中查看过程分析信息 220
在 Interactive SQL 中查看特定过程的分析信息 222
在 Sybase Central 中创建代理表 656
在 Sybase Central 中复制数据库对象 83
在 Sybase Central 中管理表和列约束 98
在 Sybase Central 中管理列缺省值 91
在 WHERE 子句中使用比较运算符 239
在 WHERE 子句中使用范围 （between 和 not between）241
在 WHERE 子句中使用列表 242
在 WHERE 子句中使用子查询 385
在参数中返回过程结果 707
在查询处理中使用工作表 201
在关系数据库中存储 XML 文档 547
在过程和触发器中使用异常处理程序 747
在过程和触发器中使用游标 738
在过程和触发器中允许的 SQL 语句 728
在过程内分隔语句 755
在过程中使用 EXECUTE IMMEDIATE 语句 752
在过程中使用 RAISERROR 语句 537
在过程中为表使用全限定名 755
在过程中在 SELECT 语句上使用游标 738
在过程中指定日期和时间 756
在结果集中返回过程结果 708
在哪些地方可以使用集合函数 253
在批处理中使用 SELECT 语句 757
在事务内更改隔离级别 122
在数据库中使用 XML 545
在提交时检查参照完整性 152
在校验表时使用 WITH EXPRESS CHECK 选项 180
脏读
不一致 117
查询过程中的锁定 149
教程 131
与隔离级别 117
增加高速缓存大小 175
摘要分析数据
触发器 218
存储过程和函数 217
事件 217
摘要信息
CUBE 操作 368
ROLLUP 操作 362
小计行 357
摘要行
ROLLUP 操作 366
摘要值 258
支持
新闻组 xvii
直方图
更新 431
关于 429
执行触发器 718
指定代理表位置 655
指定统一数据库 40
指派
数据类型到列 101
域到列 101
秩函数
关于 374
示例 375, 376, 377
中断
运行菜单 780
中断条件
设置 780
重定向
输出到文件 614
重复结果
消除 235

重复排除
查询执行算法 447
重复行
使用 UNION 删除 270
重建
工具 626
尽量缩短停机时间 629
目的 620
数据库 620
重建工具 626
dbisql 实用程序 626
dbunload 实用程序 626
UNLOAD TABLE 语句 627
重建过程中尽量缩短停机时间 629
重建数据库
UNLOAD TABLE 语句 627
对用户的影响 624
非复制数据库 622
复制数据库 624
工具 626
关于 620
命令行 628
使用 dbunload 624
与导出比较 620
原因 621
注意事项 622
重命名查询结果中的列 231
重写优化
关于 469
重置过程分析
SQL 214
Sybase Central 214
重装数据库
关于 620
命令行 628
重装数据库 另请参见 重建数据库 ix
主键
并发 159
创建 50
管理 49
生成 159
生成的索引 458
实体完整性 105
使用 NEWID 创建 UUID 93
完整性 540
性能 198
修改 50
自动增量 92
主键实施实体完整性 105
专用连接 304
转换
重写优化 469
装载
Interactive SQL 中的命令 638
装载工具 604
装载数据
工具 604
转换错误 604
子查询
ALL 测试 393
ANY 测试 391
ANY 运算符 392
GROUP BY 386
HAVING 子句 386
IN 关键字 242
WHERE 子句 385, 406
比较测试 387, 389
比较运算符 407
存在测试 387, 397
定量比较测试 387, 391
改写为连接 400
高速缓存 480
关于 383, 384
集合成员资格测试 387, 395
嵌套的 403

外部引用 386
相关 399, 405
行选择 385
行组选择 386
优化程序内部 405
运算符类型 387
重写为 EXISTS 谓语 434
转换为连接 400, 406
子查询比较测试 389
子查询测试 387
子查询的工作方式 405
子查询和函数高速缓存 480
子查询和连接 400
子查询集合成员资格测试
关于 395
子句
COMPUTE 525
FOR BROWSE 525
FOR READ ONLY 526
FOR UPDATE 525
GROUP BY ALL 525
INTO 733
ON EXCEPTION RESUME 538, 744, 749
关于 226
子事务
保存点 114
过程和触发器 754
自定义图形式计划 493
自动化
生成唯一键 159
自动连接
外键 541
自动提交
事务 113
性能 180
自动性能调优 432
自动增量
IDENTITY 列 521
UltraLite 应用程序 93
带符号的数据类型 93
负数 93
缺省值 92
自连接 304
自然连接
表的表达式 313
关于 312
具有 ON 短语 313
视图和派生表的 315
字符串
关于 245
匹配 242
选择列表使用 233
引号 245
字符串和数字缺省值 94
字符串和引号 245
字符数据
搜索 245
总成本收益
索引顾问建议 80
总收益
索引顾问建议 79
阻塞 124, 138
故障排除 125
事务 124
死锁 125
组
ASE 511
组读取
表 456
组织
数据，物理 455
最大值
高速缓存大小 192
最短距离问题 349
最佳性能提示 175
最少的管理工作
优化程序 432

最小高速缓存大小 192
左外连接
  关于 297
作为过程参数返回结果 732
浏览
  表数据 49
  视图 66
浏览数据库
  隔离级别 128