

数据库原理与设计 上机实验指导

北京理工大学计算机学院
2015 年 11 月

目录

第 1 章 SQL SERVER 基本操作	0
1.1. 本书的实验环境	0
1.2. SQL SERVER 组成	0
1.3. SQL SERVER 安装	1
1.4. SQL SERVER 数据库引擎实例	2
1.5. SQL SERVER 服务	3
1.6. SQL SERVER MANAGEMENT STUDIO 工具	4
1.7. 设置 SQL SERVER 身份验证模式	7
1.8. 登录管理	8
1.9. 修改 SA 登录的口令及状态	10
1.10. 使用 SQL SERVER 验证模式连接数据库服务器	12
1.11. 小结	14
第 2 章 安全管理	15
2.1. 权限管理的基本方法	15
2.2. 使用服务器角色给登录授予权限	15
2.3. 使用数据库角色授权	18
2.4. 指定特定对象的权限	19
2.5. 实验目的、内容与要求	25
第 3 章 表	27
3.1. 表的概念	27
3.2. 数据完整性	28
3.3. 创建和修改表	30
3.4. 查询表	31
3.5. 自动编号列和标识符列	31
3.6. 基本数据定义与查询	36
3.7. 复杂查询	37
3.8. 示例数据库	38
第 4 章 TRANSACT-SQL 编程	42
4.1. 程序中处理错误	42
4.1.1. RAISERROR 语法	42
4.1.2. RAISERROR 示例	46
4.2. 游标	49

4.2.1. 游标的基本概念与操作	49
4.2.2. 处理游标中的行	51
4.3. 存储过程	53
4.3.1. 什么是存储过程	53
4.3.2. 存储过程的类型	54
4.4. 触发器	55
4.4.1. DML 触发器的类型	56
4.4.2. AFTER 触发器与 INSTEAD OF 触发器的比较	56
4.4.3. AFTER 触发器示例	57
4.4.4. 允许与禁止触发器	59
4.4.5. 删除的表 deleted 和插入的表 inserted	60
4.4.6. INSTEAD OF 触发器示例一	60
4.4.7. INSTEAD OF 触发器示例二	65
4.5. 实验	67
第 5 章 数据库备份与恢复	69
5.1. 实验的目的、内容与要求	69
5.2. 备份与恢复概述	69
5.3. 简单恢复模式下的备份与恢复	74
5.4. 完整恢复模式下的备份与恢复	76
第 6 章 实验内容	82
6.1. 实验 1 创建和修改数据库、数据表	82
6.1.1. 实验目的	82
6.1.2. 实验要求	82
6.1.3. 实验内容	82
6.2. 实验 2 数据的基本查询、高级查询和视图操作	83
6.2.1. 实验目的	83
6.2.2. 实验要求	84
6.2.3. 实验内容	84
6.3. 实验 3 数据库编程	85
6.3.1. 实验目的	85
6.3.2. 实验要求	85
6.3.3. 实验内容	85
6.4. 实验 4 数据库备份、恢复及安全管理	86
6.4.1. 实验目的	86

6.4.2. 实验要求	86
6.4.3. 实验内容	86

第1章 SQL Server 基本操作

1.1. 本书的实验环境

本书的示例与 Windows 和 SQL Server 环境有关，具体的环境如下：

Windows 环境：

Windows 7 旗舰版 32 位

主机名：GUO-PC

用户：guo，属于 Administrators 组

SQL Server 环境：

SQL Server 2012 Express（SQL Server 11.0.2100）

1.2. SQL Server 组成

SQL Server 提供包括数据库引擎、Analysis Services、Reporting Services、Integration Services 和 Master Data Service 服务等，这些服务由表 1-1 中的服务器组件提供。其中，数据库引擎是 SQL Server 的数据库服务器，Analysis Services 是 OLAP 分析服务器，Reporting Services 是报表服务器，Integration Services 是数据集成服务器。本书的内容只涉及数据库引擎服务器，它可以完成数据库的管理、安全管理（用户与权限）、数据库基本对象（表、索引、视图）和 T-SQL 对象的管理等。

表 1-1 SQL Server 主要服务器组件

服务器组件	说明
SQL Server 数据库引擎	SQL Server 数据库引擎包括数据库引擎（用于存储、处理和保护数据的核心服务）、复制、全文搜索、用于管理关系数据和 XML 数据的工具以及 Data Quality Services (DQS) 服务器。
Analysis Services	Analysis Services 包括用于创建和管理联机分析处理 (OLAP) 以及数据挖掘应用程序的工具。
Reporting Services	Reporting Services 包括用于创建、管理和部署表格报表、矩阵报表、图形报表以及自由格式报表的服务器和客户端组件。 Reporting Services 还是一个可用于开发报表应用程序的可扩展平台。
Integration Services	Integration Services 是一组图形工具和可编程对象，用于移动、复制和转换数据。 它还包括 Integration Services 的 Data Quality Services (DQS) 组件。
Master Data Services	Master Data Services (MDS) 是针对主数据管理的 SQL Server 解决方案。 可以配置 MDS 来管理任何领域（产品、客户、帐户）；MDS 中可包括层次结构、各种级别的安全性、事务、数据版本控制和业务规则，以及可用于管理数据的用于 Excel 的外接程序。

表 1-2 SQL Server 主要管理工具

管理工具	说明
SQL Server Management Studio	SQL Server Management Studio 是用于访问、配置、管理和开发 SQL Server 组件的集成环境。Management Studio 使各种技术水平的开发人员和管理员都能使用 SQL Server。Management Studio 的安装需要 Internet Explorer 6 SP1 或更高版本。
SQL Server 配置管理器	SQL Server 配置管理器为 SQL Server 服务、服务器协议、客户端协议和客户端别名提供基本配置管理。
SQL Server 事件探查器	SQL Server 事件探查器 提供了一个图形用户界面，用于监视数据库引擎实例或 Analysis Services 实例。
数据库引擎优化顾问	数据库引擎优化顾问可以协助创建索引、索引视图和分区的最佳组合。
数据质量客户端	提供了一个非常简单和直观的图形用户界面，用于连接到 DQS 数据库并执行数据清理操作。它还允许您集中监视在数据清理操作过程中执行的各项活动。数据质量客户端的安装需要 Internet Explorer 6 SP1 或更高版本。
SQL Server 数据工具	SQL Server 数据工具 (SSDT) 提供 IDE 以便为以下商业智能组件生成解决方案：Analysis Services、Reporting Services 和 Integration Services。（以前称作 Business Intelligence Development Studio）。SSDT 还包含“数据库项目”，为数据库开发人员提供集成环境，以便在 Visual Studio 内为任何 SQL Server 平台（无论是内部还是外部）执行其所有数据库设计工作。数据库开发人员可以使用 Visual Studio 中功能增强的服务器资源管理器，轻松创建或编辑数据库对象和数据或执行查询。SQL Server 数据工具安装需要 Internet Explorer 6 SP1 或更高版本。
连接组件	安装用于客户端和服务器之间通信的组件，以及用于 DB-Library、ODBC 和 OLE DB 的网络库。

1.3. SQL Server 安装

下载 SQL Server Express 2012

SQL Server 2012 包括 Enterprise、Business Intelligence、Standard 三个主要版本，另外，微软还提供 Web（Professional）、Developer 和 Express 版本。SQL Server Express 2012 是微软提供的一个免费版本。其下载的页面上提供了多种下载选项，其中，下载文件 SQEXPRWT_x86_CHS.exe 包括数据库引擎和工具 SQL Server Management Studio Express。

注解：如果安装的不是 Express 版本，则需要选择安装的组件。本书的内容中需要安装数据库引擎服务器组件和客户端工具组件。因为有些组件需要额外的安装条件，这会明显降低安装的速度，也可能导致安装失败。

SQL Server Express 2012 下载页面：

<http://www.microsoft.com/zh-cn/download/details.aspx?id=29062>

SQL Server Express 2012 SQLEXPRT_x86_CHS.exe 下载地址:

http://download.microsoft.com/download/3/6/E/36E9CA26-CC2C-4600-8D25-A152F9498FA1/CHS/x86/SQLEXPRT_x86_CHS.exe

安装示例数据库 AdventureWorks

后面内容中所提供的示例均使用 SQL Server 示例数据库 AdventureWorks。SQL Server 2012 的示例数据库 AdventureWorks 需要单独安装，其安装文件 AdventureWorks2012_Data.mdf 是 AdventureWorks2012 数据库的数据文件，其下载地址:

<http://download-codeplex.sec.s-mft.com/Download/Release?ProjectName=msftdbprodsamples&DownloadId=165399&FileTime=129762331847030000&Build=20717>

可以使用下面两种方法之一安装示例数据库 AdventureWorks。

方法一:

- (1) 在 SQL Server Management Studio 中从数据库的弹出菜单中选择“附加”数据库。
- (2) 在附加数据库对话框中添加数据库，选择文件 AdventureWorks2012_Data.mdf，注意要删除自动添加的日志文件，即完成 AdventureWorks2012 的安装。
- (3) 重命名 AdventureWorks2012 为 AdventureWorks

方法二:

- (1) 在 SQL Server Management Studio 的查询分析器中执行下面的命令:

```
exec sp_attach_db @dbname=N'AdventureWorks2012',  
@filename=N'D:\SQL Server 2000 Sample Databases\AdventureWorks2012_Data.mdf',  
GO  
USE master  
GO  
ALTER DATABASE AdventureWorks2012 MODIFY name=AdventureWorks  
GO
```

1.4. SQL Server 数据库引擎实例

数据库引擎的实例（简称“实例”或“数据库实例”，你也可以把它称做“SQL Server 实例”）是作为操作系统服务运行的 sqlservr.exe 进程。每个实例管理多个系统数据库（master、model、msdb 和 tempdb）以及零个或多个用户数据库（例如 SQL Server 示例数据库 AdventureWorks）。

每台计算机可以运行数据库引擎的多个实例。如果是第一次安装 SQL Server，可以安装一个默认实例，一台计算机上只能安装一个默认实例。默认实例没有名称，连接默认实例只需要提供计算机名，而连接有名实例则需要提供计算机名和实例名，如 GUO-PC\SQLEXPRESS，其中，GUO-PC 是运行实例的计算机名，SQLEXPRESS 是实例

名。

应用程序必须连接到某一个实例，只有通过实例才能访问 SQL Server 数据库。

数据库实例与其它的计算机进程没有本质上的区别，都是可执行程序的运行副本，所不同的是访问的数据不同。例如，运行 Word 的可执行程序文件 WINWORD.EXE 便会产生一个该程序的 WINWORD.EXE 进程，它可以称为 Word 实例，重复运行就会产生多个 Word 实例，每个 Word 文档都必须通过一个 Word 实例才能够访问。Word 文档就相当于 Word 数据库。

注解：严格意义上讲，不通过 Word 实例也可以访问 Word 文档。但从安全性的角度考虑，数据库软件厂商不会公开其数据库的内部结构（除了像 Access 个人数据库外），也不会提供访问数据库的其它方式，只能通过实例访问数据库。

实例处理所有应用程序请求的数据库服务操作。在完成某一连接后，应用程序通过该连接将 Transact-SQL 语句发送给该实例。该实例将这些 Transact-SQL 语句解析为针对数据库中的数据和对象的操作并将操作的结果或错误消息返回给应用程序。

注解：与 SQL Server 不同，Oracle 的实例定义为由后台进程和系统全局区 SGA（用于访问数据库的内存）两部分组成，Oracle 实例并不负责解析 SQL 语句，也不负责管理连接和传送数据，这些任务由 Oracle 服务器进程完成。

1.5. SQL Server 服务

SQL Server 服务是 SQL Server 引擎的 Windows 进程。在操作数据库前，必须要保证该服务是处于运行状态。可能通过 SQL Server 配置管理器（在 SQL Server 配置管理菜单项中）来启动、关闭及配置这个服务。（如图 1-1 所示）

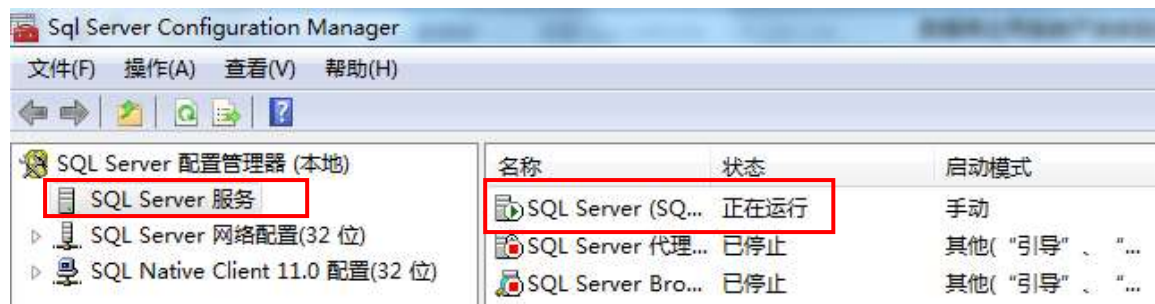


图 1-1 SQL Server 配置管理器

也可以通过 Windows 的服务管理器来启动和关闭该服务（图 1-2）。



图 1-2 Windows 服务管理器中的 SQL Server 服务

1.6. SQL Server Management Studio 工具

1.6.1. 连接 SQL Server 数据库引擎

运行 SQL Server Management Studio (简称 SSMS)。第一个出现的界面是连接 SQL Server 服务器的对话框，如图 1-3。



图 1-3 SSMS 连接数据库服务器

在图 1-3 中，在服务器类型中选择“数据库引擎”

(注：如果只安装

了数据库引擎组件，则服务器类型列表框不能选择，它自动选择“数据库引擎”)。

服务器名称“GUO-PC\SQLEXPRESS”中的 GUO-PC 是所连接的服务器的主机名，而 SQLEXPRESS 则是数据库实例名。如果连接默认实例，则只有主机名。

身份验证列表框列出了“Windows 身份验证”和“SQL Server 身份验证”两种方式。身份验证在后面有专门的实验进行练习，因此在这里选择“Windows 身份验证”。

当成功连接到 SQL 数据库服务器后，则进入 SSMS 的基本操作界面，如图 1-4 所示。

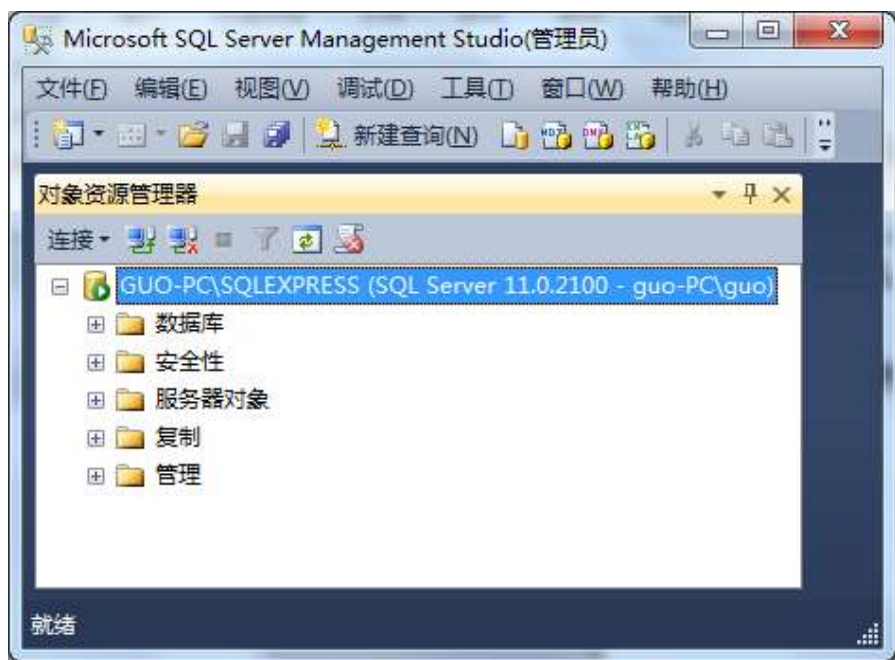


图 1-4 SSMS 的基本操作界面

检查 SQL Server 服务器的基本信息

对象资源管理器(图 1-4)的根节点是一个连接的数据库引擎节点,显示了 SQL Server 数据库引擎的基本信息 GUO-PC\SQLEXPRESS(SQL Server 11.0.2100-guo-PC/guo),包括运行数据库引擎服务器的主机名 GUO-PC、数据库实例名 SQLEXPRESS (如果是默认实例则没有实例名)、SQL Server 数据库引擎的版本 11.0.2100、连接数据库引擎的登录名 GUO-PC\guo。

上面的示例是数据库引擎在本地主机运行,所以,服务器名称和登录名中的主机都是 GUO-PC。如果使用 GUO-PC 上的用户连接的服务器运行的主机名是 Server1、实例名为 EEXPRESS,则显示的信息应为: Server1\SQLEXPRESS(SQL Server 11.0.2100-guo-PC/guo)。



图 1-5 对象资源按理器的按钮

浏览对象资源管理器工具栏

其中, SSMS 左边的窗口是对象资源管理器,右边的窗口是摘要窗口。在对象资源管理器窗口的工具栏中有五个按钮,分别是连接、断开连接、停止、刷新和筛选。如果需要使用新的用户建立与数据库服务器的连接,则可以使用工具栏的“连接”按钮并选择“数据库引擎”,这样就会出现与图 1-3 完全相同的界面。

使用查询编辑器

可以使用 SSMS 的“新建查询”或者“数据库引擎查询”打开一个查询编辑器(如

图 1-6 所示)。查询编辑器中可以执行 T-SQL 语句，它还可以将其中的 T-SQL 语句保存到相应的文件中。需要注意的是，查询编辑器标题栏中的连接信息，它决定了执行 T-SQL 语句的环境与权限，图 1-6 中的 guo-PC\guo 表示使用登录 guo-PC\guo 连接了数据库引擎。

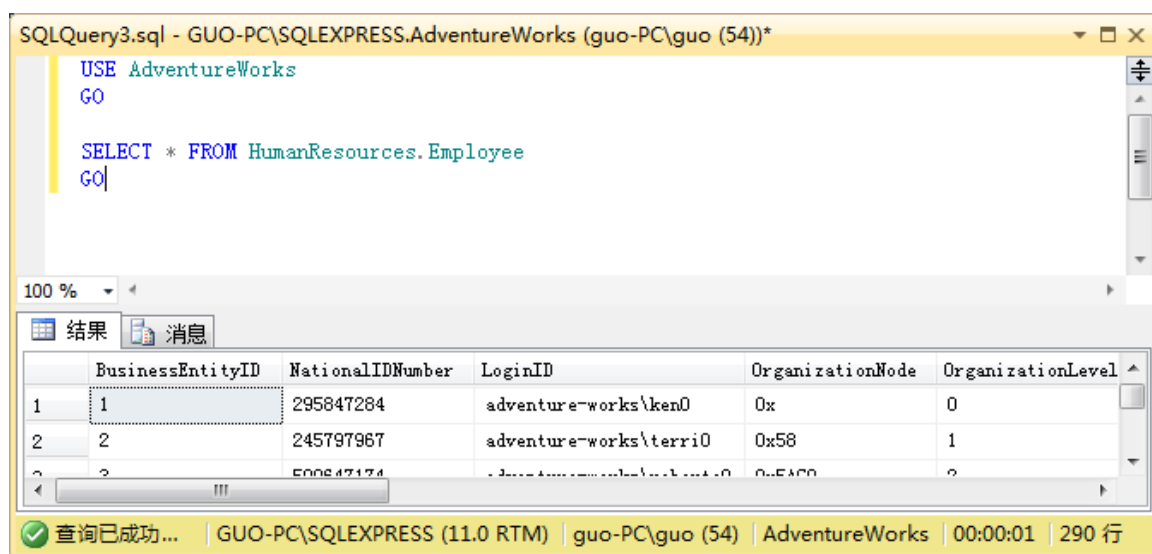


图 1-6 SSMS 查询编辑器

检查数据库

展开 SQL Server 服务器下面的节点“数据库”后会显示该数据库服务器上所有的数据库。其中，有系统数据库和示例数据库“AdventureWorks”。进一步展开“系统数据库”节点后会显示所有的系统数据库，这些数据库的具体信息会在后续的内容中进一步学习。

可以选择其中的一个数据库节点如 master，展开后显示该数据库的更详细的信息，如图 1-7。



图 1-7 master 系统数据库

检查数据库服务器的安全性

展开对象资源管理器的“安全性”节点，可以查看数据库服务器的安全性，包括登录名、服务器角色和凭据。

注意登录名中的“guo-PC\guo”，它的存在使得以 Windows 的用户 guo 登录到 Windows 后可以通过 Windows 身份验证模式直接连接到数据库服务器。具体的机制在后面的内容中会作进一步的解释。

1.7. 设置 SQL Server 身份验证模式

SQL Server 提供了两种身份验证模式，一是 Windows 身份验证模式，二是 SQL Server 和 Windows 身份验证模式（简称混合验证模式）。混合验证模式可以使用 Windows 身份验证模式，也可以使用 SQL Server 身份验证模式。Windows 身份验证模式是假设登录到 Windows 的用户是安全的，在连接到 SQL Server 时不需要提供用户密码。SQL Server 身份验证模式在连接数据库服务器时必须提供 SQL Server 的登录名和相应的密码。两种身份验证模式都使用登录连接数据库服务器。

两种验证模式中，Windows 身份验证模式更为常用。特别是通过应用程序访问数据库时，由于不需要在应用程序中硬编码登录密码，所以提高了数据库服务器的安全性。而有些应用，比如用户通过 Internet 直接访问数据库时，由于用户本身无法登录到数据库服务器或其所在的域，因而就必需使用 SQL Server 身份验证模式。另外，有一些应用程序本身只支持 SQL Server 身份验证，例如 ERwin Data Modeler 4.0。所以本实验的目的是通过操作掌握数据库服务器的身份验证模式的设置，以根据不同的应用设置不同的验证模式。

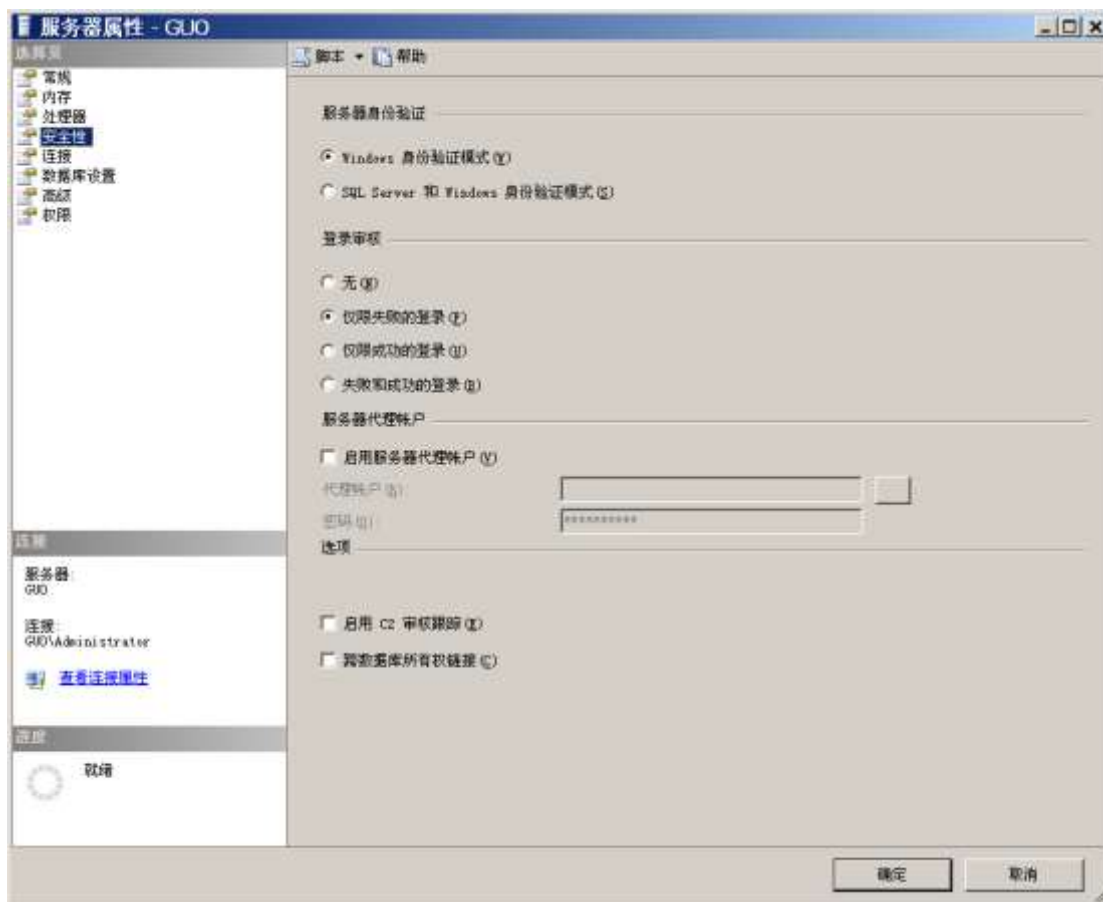


图 1-8 修改 SQL Server 的验证模式

修改 SQL Server 身份验证的操作很简单,选择 SSMS 中对象资源管理器的 SQL Server 服务器节点,通过弹出菜单的“属性”项进入到修改属性对话框。选择对话框左侧列表中

的“安全性”，右侧的窗口中出现服务器安全属性窗口。在图 1-8 中选择 SQL Server 和 Windows 身份验证模式。当前是 Windows 身份验证模式，它是安装时设定的，安装时也可以设定为混合验证模式。

1.8. 登录管理

为了测试 SQL Server 登录的管理，首先需要新建一个 Windows 用户。下面的实验中新建的 Windows 用户的名称为 david。

为了测试如何操作才能使一个 Windows 的用户以 Windows 的身份验证模式登录到 SQL Server，我们在实验的过程中分成多个步骤，这样的步骤被分成创建 Windows 用户、在 SQL Server 中创建相应的登录和给 SQL Server 的登录分配相应的权限三个步骤。每个步骤完成后进行测试以检查不正确的配置会发生什么样的问题，这样以加深对整个配置过程的理解。

具体的操作步骤如下：

- (1) 使用管理员用户 guo 创建 Windows 用户 david。
- (2) 使用新的用户 david 登录 Windows 并连接 SQL Server。如果成功则进行第 5 步，否则，进行第 3 步。
- (3) 使用用户 guo 为 david 用户创建 SQL Server 登录 GUO-PC\david。
- (4) 第二次使用新的用户 david 登录 Windows 并连接 SQL Server。
- (5) 使用管理员用户 guo 连接数据库并设置 SQL Server 登录 GUO-PC\david 的默认数据库和权限。

1.8.1. 第一次连接 SQL Server

使用新创建的 Windows 用户重新登录 Windows，注意检查 SSMS 连接对话框中的连接信息是否是如图 1-9 所示内容，特别是用户名是否是本地主机的新用户 david。然后使用 SSMS 连接 SQL Server，连接成功。



图 1-9 使用 david 用户连接数据库服务器

注解：在有些环境下，可能连接失败。Windows 用户使用 Windows 身份验证连接数据库服务器必须在数据库服务器中有对应的登录。对应的登录有两种：一是为 Windows 用户创建的登录，例如，Windows 主机 GUO-PC 上的用户 david 创建的数据库登录名为 GUO-PC\david，david 用户使用该登录连接数据库服务器；二是为 Windows 用户组创建的登录，例如，在数据库服务器中有一个名为 BUILTIN\Users 的登录，它是一个内置的登录，与所有属于 Users 组的 Windows 用户对应，每个属于 Windows 用户组 Users 的用户都使用该登录连接数据库（在没有对应用户登录的情况下）。如果在数据库服务器中没有对应的登录，则连接失败。

1.8.2. 创建 SQL Server 登录

新建登录需要使用具有相应权限的 Windows 用户重新登录 Windows 并连接 SQL Server。在 SQL Server 对象资源管理器中选择“安全性”，右键并选择“新建登录名...”进入到 SQL Server 新建登录对话框（如图 1-10 所示）。在“登录名”文本框中输入所要

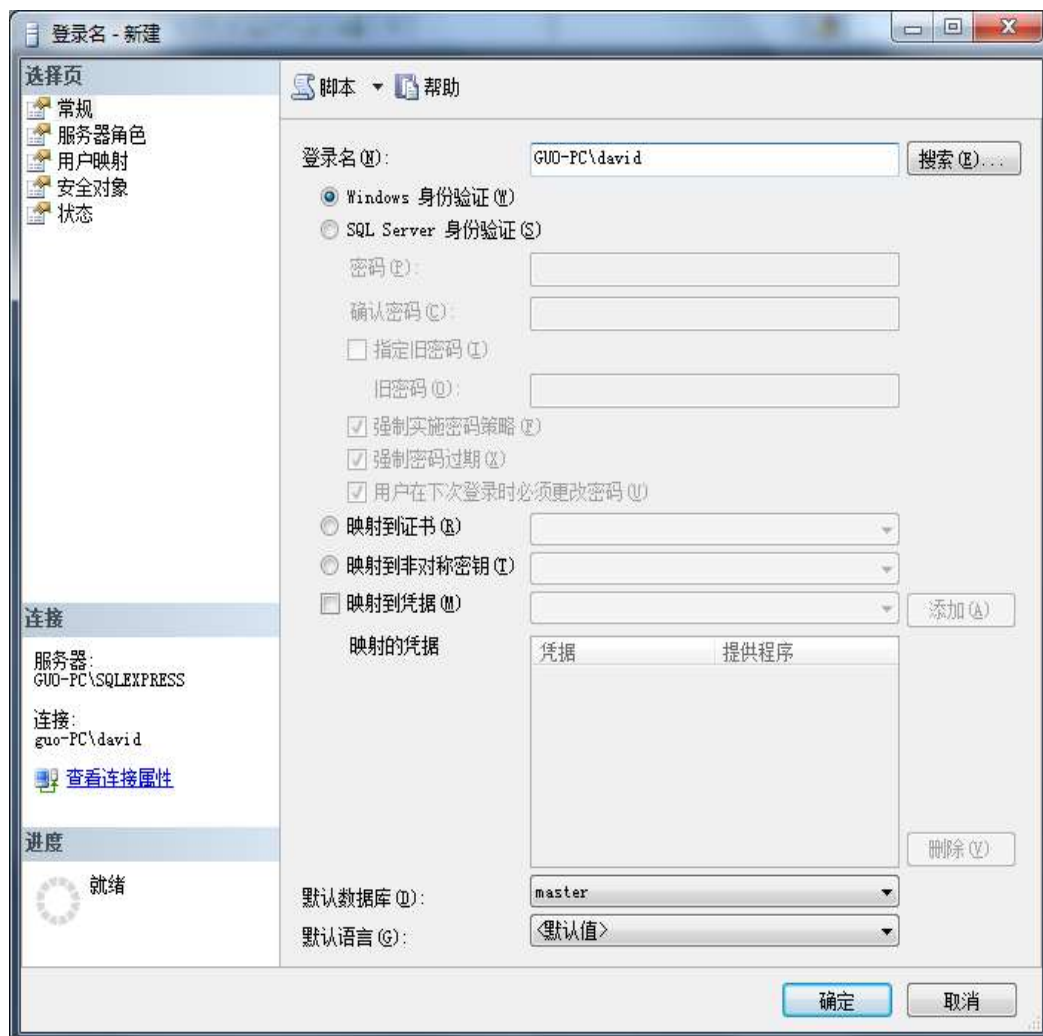


图 1-10 创建登录 GUO-PC\david

创建的登录名 GUO-PC\david，其中，GUO-PC 表示主机名，david 表示主机 GUO-PC 上的 Windows 用户名。

注解：可以使用图 1-10 中右边窗口中的“脚本”将要执行的操作转换为 T-SQL 语句并显示在一个查询编辑器窗口中，或者保存到一个文件中。

完成新建登录后，可以通过 SSMS 查看新建的登录是否确实存在。展开 SSMS 的“对象资源管理器”的“安全性”节点，检查前面操作所创建的 SQL Server 登录 GUO-PC\david 是否存在。

1.9. 修改 sa 登录的口令及状态

sa 登录是 SQL Server 的一个特殊登录，使用它连接数据库服务器后可以完成对服务

器所有的管理操作，因此，需要对 sa 登录进行重点保护。

sa 登录的口令在安装过程中可以设定，但如果在安装时选择的是 Windows 身份验证模式则无法设定它的密码，且该登录的状态也被设定为“禁用”。

修改 sa 登录的密码与状态与修改其它 SQL Server 登录的密码与状态具有相同的操作过程。从 SSMS 的“安全性”节点选择相应的登录（这里为 sa），进入到登录属性修改对话框。

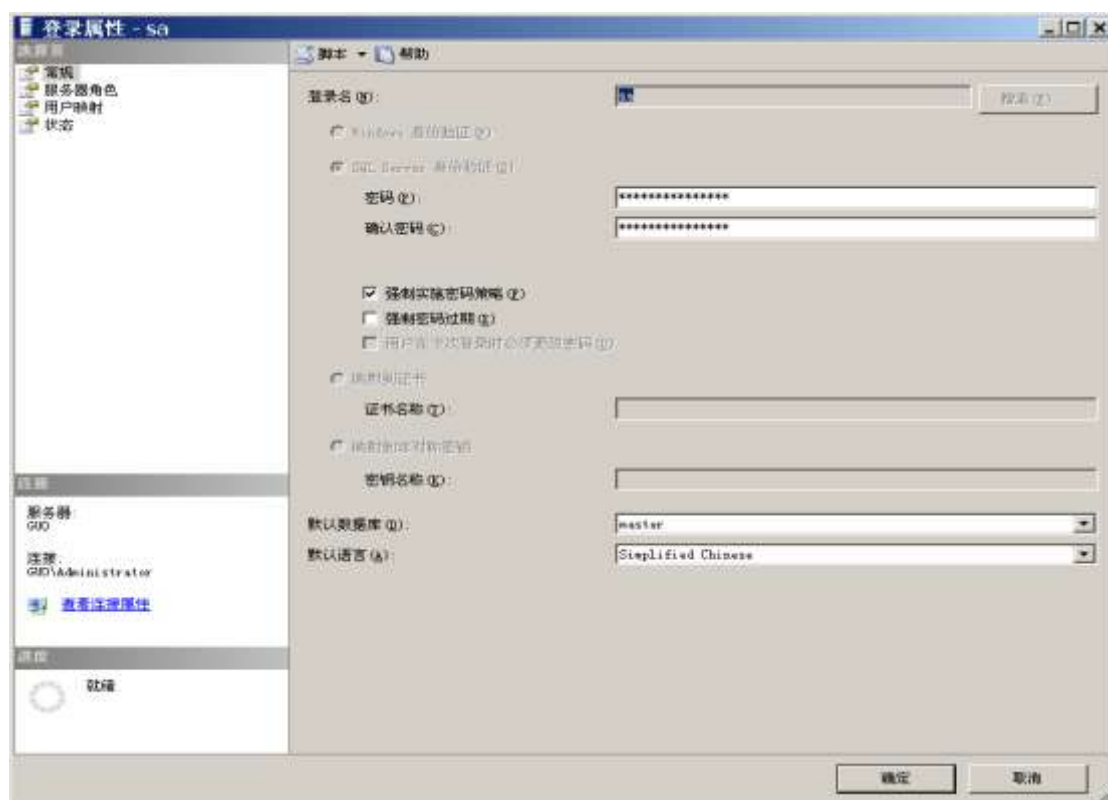


图 1-11 sa 登录属性

输入新的密码，然后选择左边列表中的“状态”，进入到状态属性修改页面，如图 1-12 所示。注意，sa 初始的状态为“禁用”，把登录状态修改为“启用”并点击确定。

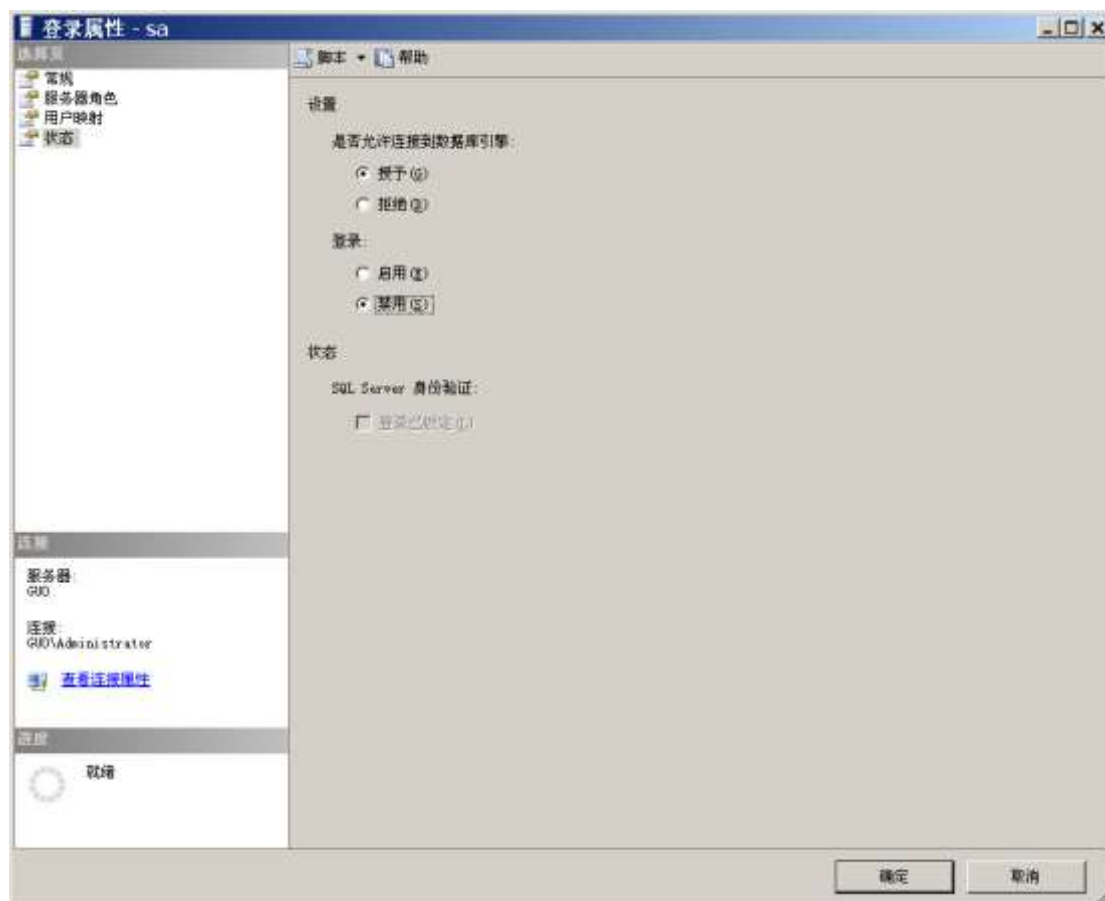


图 1-12 sa 登录的状态

1.10. 使用 SQL Server 验证模式连接数据库服务器

在完成对数据库服务器的登录验证模式的修改后，可以使用 Windows 身份验证也可以使用 SQL Server 身份验证模式登录数据库服务器。在



图 1-13 使用 sa 登录 SQL Server

SQL Server 登录 sa 属性的修改后，现在可以使用 SQL Server 登录连接数据库服务器了。

使用 SSMS 断开与服务器的连接，重新连接时使用 SQL Server 登录模式并使用 sa 用户登录连接数据库服务器，如图

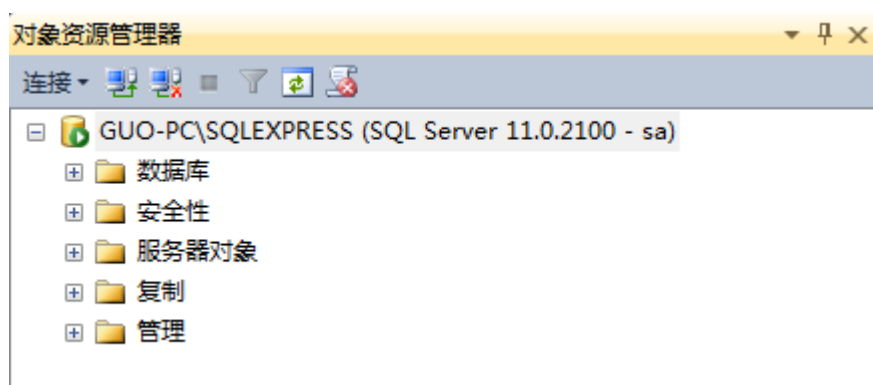


图 1-14 使用 sa 登录 SQL Server 的 SSMS

1-14 所示。连接后注意 SSMS 中显示的信息的变化。

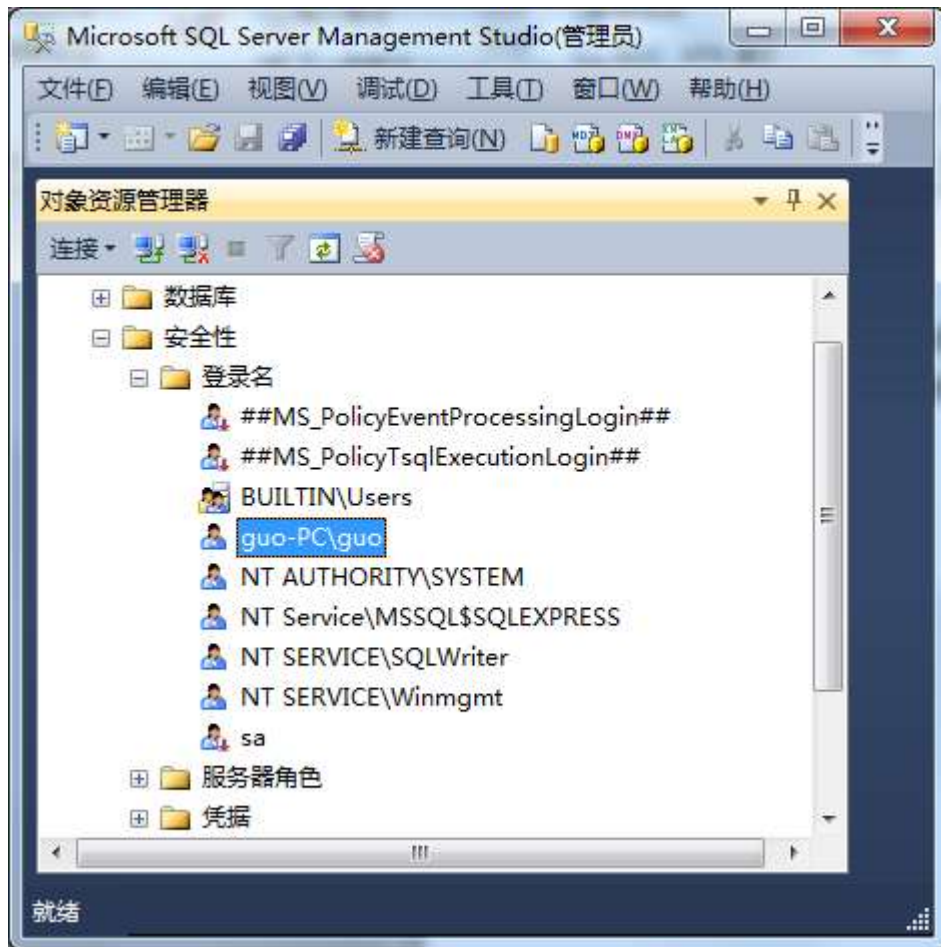


图 1-15 对象资源管理器的安全性

1.11. 小结

通过本实验完成了对 SSMS 的初步认识, SSMS 其它的功能操作分别在以后的各实验进行完成。如果需要系统的了解 SSMS 的功能, 可以参考 SQL Server 的联机文档。

第2章 安全管理

2.1. 权限管理的基本方法

可以使用下面几种方式授予登录操作数据库的权限：

- (1) 将登录加入到服务器角色中。例如，将登录加入到 `sysadmin` 服务器角色中，登录就具有对所有数据库的全部操作权限。
- (2) 将登录映射到某一个数据库的某一个用户，并将映射的用户加入到具有相应操作权限的数据库角色中。
- (3) 将登录映射到某一个数据库的某一个用户，授予该用户对某些特定的数据库对象的操作权限。

其中，方法 1 和方法 2 都是一种隐含的授权操作，有时这些方法会将过多的权限授予登录。方法 3 是一种更为细化的授权操作。

2.2. 使用服务器角色给登录授予权限

2.2.1. 数据库服务器角色

SQL Server 2012 提供了 9 个服务器角色（“角色”类似于 Windows 中的“组”），这些角色从服务器一级定义了对数据库服务器操作的权限。但这些角色的管理只限于给这些角色添加或删除成员，并不能创建新的服务器角色或删除已有的服务器角色，也不能修改它们中的权限，所以它们也被你为固定服务器角色。

除固定服务器角色外，SQL Server 2012 及以后的版本中，可以创建删除用户定义的服务器角色，也可以修改用户定义的服务器角色的权限。

需要注意的是服务器角色与数据库角色的区别，后者是在数据库一级定义的操作权限，它是针对某一数据库而不是数据库服务器的。

服务器角色的成员是数据库服务器的登录或者其它服务器角色，既可以是 Windows 登录也可以是 SQL Server 登录。通过将某一登录添加到某一个服务器角色中，该登录就继承服务器角色的权限，拥有对整个数据库服务器相应的操作权限。例如，将前面创建的 Windows 登录 `GUO\david` 加入到 `sysadmin` 服务器角色中，该登录就拥有对整个数据库服务器的所有操作权限（`sysadmin` 拥有对数据库服务器操作的所有权限）。

如果不需要对整个服务器进行管理，则一般不需要将登录添加到某一个服务器角色中，而是使用数据库角色来完成安全权限的配置。

2.2.2. 权限不足的操作

创建一个使用 SQL Server 验证模式的 SQL Server 的登录 `student`，首先使用具有相应操作权限的登录（如 `sa`）连接到 SQL Server 数据库服务器，在“安全性”节点中点击右

键并选择“创建新登录”进入到创建新登录的对话框，在对话框中输入新创建的登录的名称 student，选择 SQL Server 验证模式，如果不需要设置登录的密码则不能选择“强制实施密码策略”，否则必须设置由字母和数字混合组成的密码。

现在，使用新创建的登录 student 连接到 SQL Server 数据库服务器，并创建另一个新登录的操作或其它的操作，则会出现如下的错误信息：



图 2-1 不具有权限的错误信息

2.2.3. 修改登录的服务器角色集

使用 sa 或 Windows 的 administrators 用户组的用户连接到数据库服务器，选择“安全性/登录”节点中的 student 登录，修改属性将 sysadmin 服务器角色加入到 student 登录的服务器角色集中。

注解：修改登录的服务器角色集类似于 Windows 中修改用户的组属性，即修改 Windows 用户隶属于组的集合。

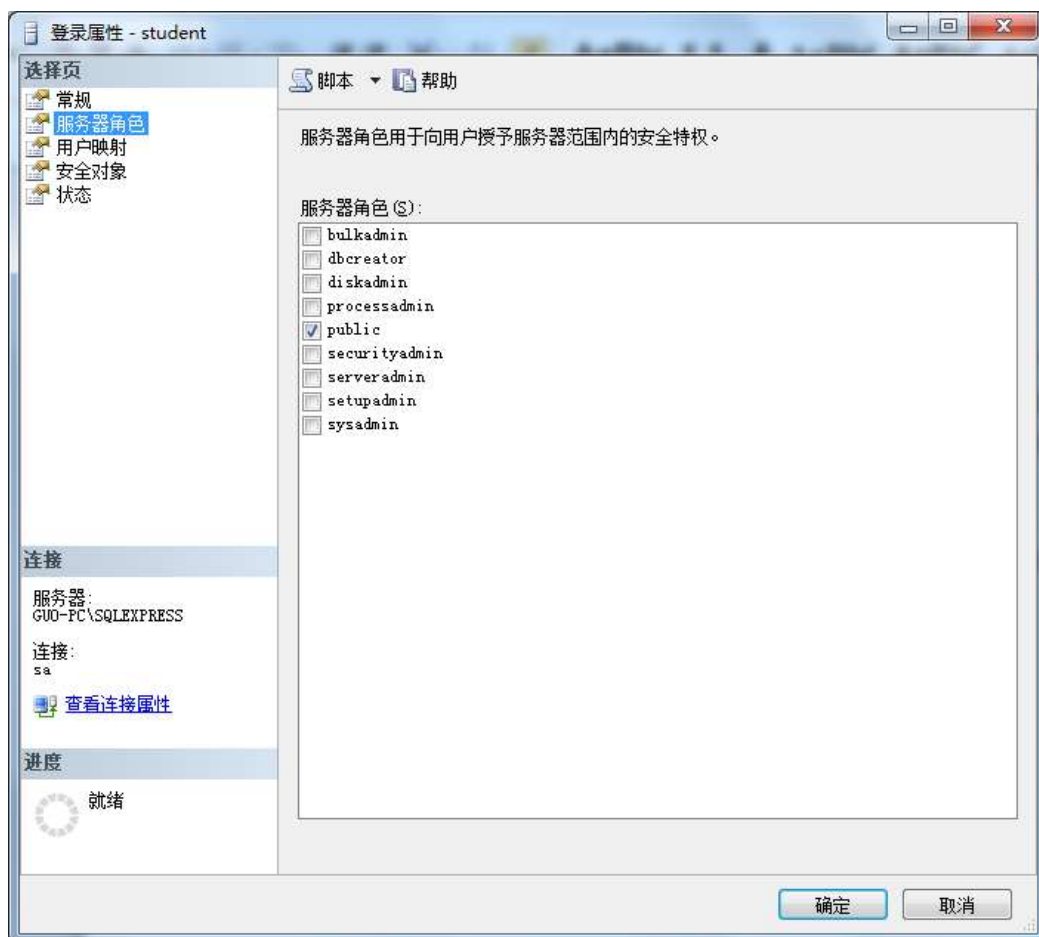


图 2-2 修改登录的服务器角色集

也可以使用如下的 T-SQL 命令将登录添加到服务器成员列表中：

```
--将登录student添加到服务器角色sysadmin中
ALTER SERVER ROLE [sysadmin] ADD MEMBER [student]
GO
```

注解：在联机文档中提供的命令示例 `ALTER SERVER ROLE diskadmin ADD [Domain\Juan]` 有误。

在将新的登录 `student` 添加到 `sysadmin` 角色中后，登录 `student` 就具有了对数据库服务所有的操作权限。

再次进行前面的创建一个新登录的操作，这次发现 `student` 登录可以完成这一操作。实际上，`student` 可以完成所有对数据库服务器的操作。

完成后使用下面的命令将登录 `student` 从 `sysadmin` 中删除。

```
--从服务器角色sysadmin中删除角色成员student
ALTER SERVER ROLE [sysadmin] DROP MEMBER [student]
GO
```

2.3. 使用数据库角色授权

使用数据库角色授权是将对某一数据库的所有对象某些操作权限授予某一登录。但这种授权不是直接将操作权限授予登录，而是首先将登录映射到某一个数据库的某一用户，再将用户添加到数据库角色成员列表中。例如，图 2-3 中的登录 student 映射到数据库 AdventureWorks 的 stu_user 用户，然后将 stu_user 添加到 db_owner 数据库角色成员列表中，这样 stu_user 就继承了 db_owner 的所有权限，而 student 登录又继承了 stu_user 对数据库 AdventureWorks 的操作权限，因为 db_owner 包含了对数据库的所有操作权限，所以，student 登录就获得了对数据库 AdventureWorks 的所有操作权限。

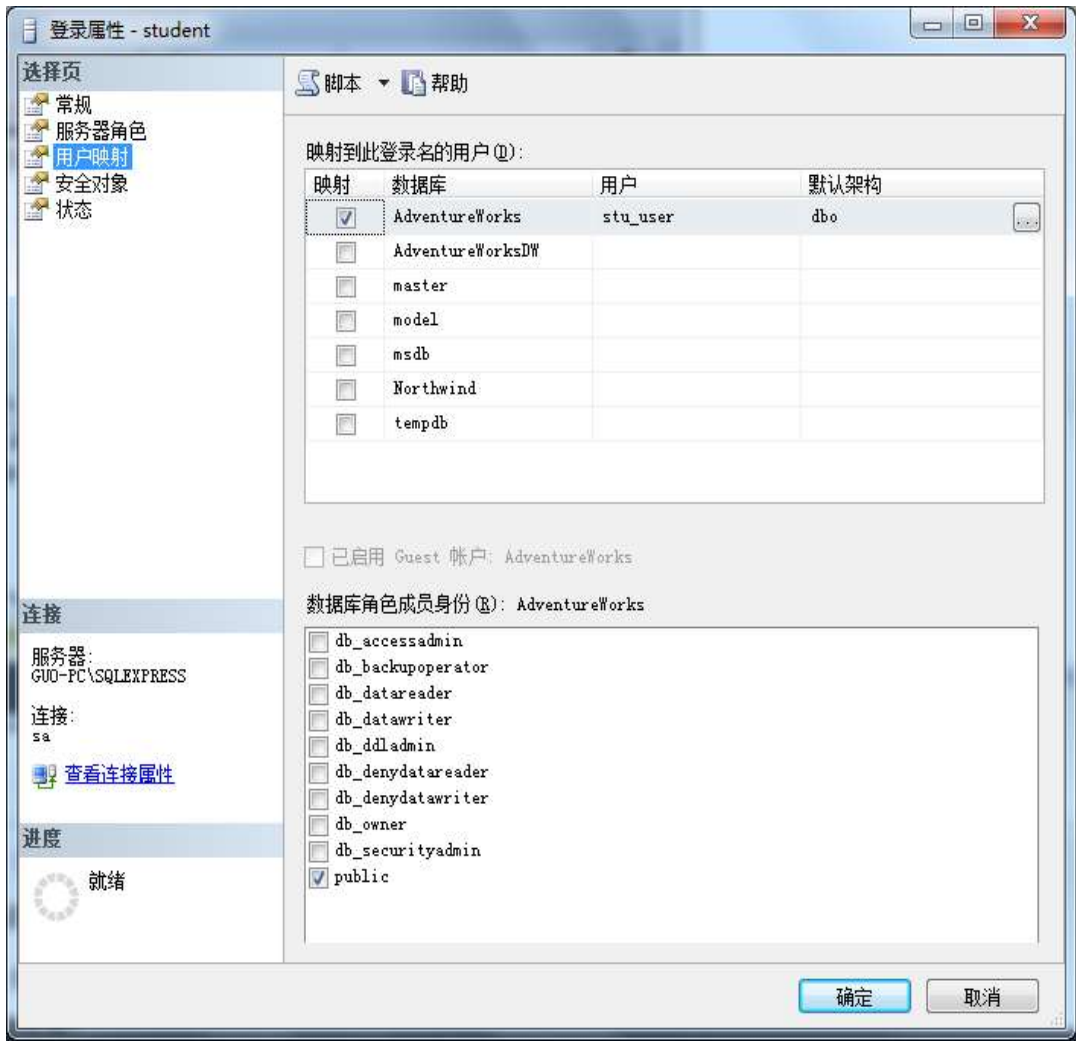


图 2-3 登录的数据库用户映射

操作步骤如下：

- （1）使用系统管理员登录（在 Windows Administrators 组中的 Windows 用户或 SQL Server 的 sa 登录）连接到数据库。
- （2）创建一个登录（也可以使用没有经过授权的已有的登录）student。

```
CREATE LOGIN student
WITH PASSWORD=' ', DEFAULT_DATABASE=master
```

(3) 设置登录在数据库 AdventureWorks 的用户映射。

```
USE AdventureWorks
GO
CREATE USER stu_user FOR LOGIN student
GO
```

(4) 将用户 stu_user 添加到数据库角色 db_owner（数据库拥有者）中。

```
USE [AdventureWorks]
GO
ALTER ROLE [db_owner] ADD MEMBER [stu_user]
GO
```

经过上面的操作（图 2-3），登录 student 就成为数据库 AdventureWorks 的拥有者，可以对该数据库进行相应的操作，如创建、修改、删除数据库对象等。

2.4. 指定特定对象的权限

使用数据库服务器角色是对所有数据库中的所有对象授权，而使用数据库角色则是对某一个数据库中的所有对象进行授权。当需要对特定的数据库对象进行授权时，则必须创建应用程序角色或对特定对象进行授权。下面的操作过程演示了对数据库 AdventureWorks 中的表 Person.Address（其中，Person 是架构名，Address 是表名）进行查询和更新的授权操作。

- (1) 创建一个 SQL Server 验证的登录 application，设置它的默认数据库为 AdventureWorks 并设置在该数据库中的用户映射。
- (2) 以该登录连接数据库服务器，查看数据库 AdventureWorks 下的数据库对象，结果只能看到数据库的系统对象，数据库的其它对象则看不到。
- (3) 选择数据库 AdventureWorks 的用户 application 修改其属性。
- (4) 在属性对话框中选择“安全对象”页，如图 2-4 所示：

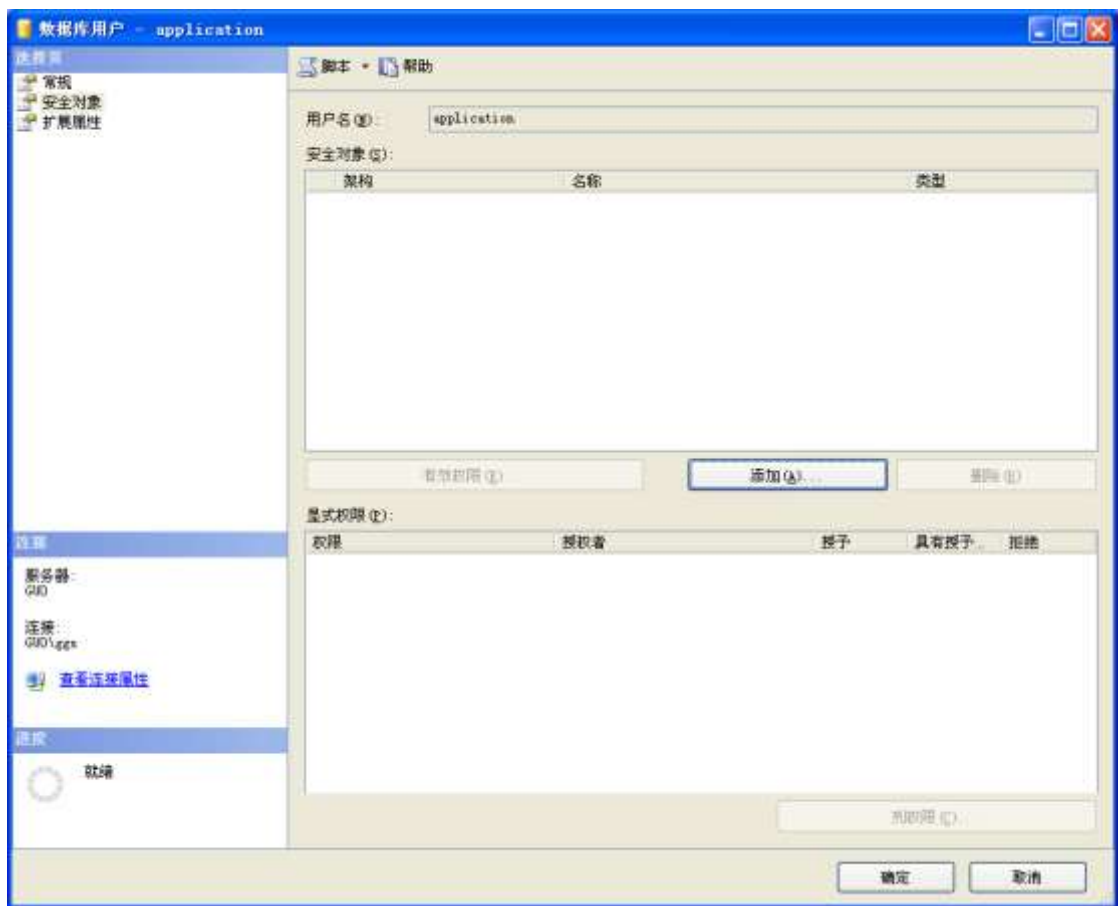


图 2-4 修改数据库用户的安全对象属性

- (5) 在图 2-4 中，选择右边窗口中的“添加”按钮，添加对特定数据库对象的操作权限，进入到图 2-5 所示的操作。

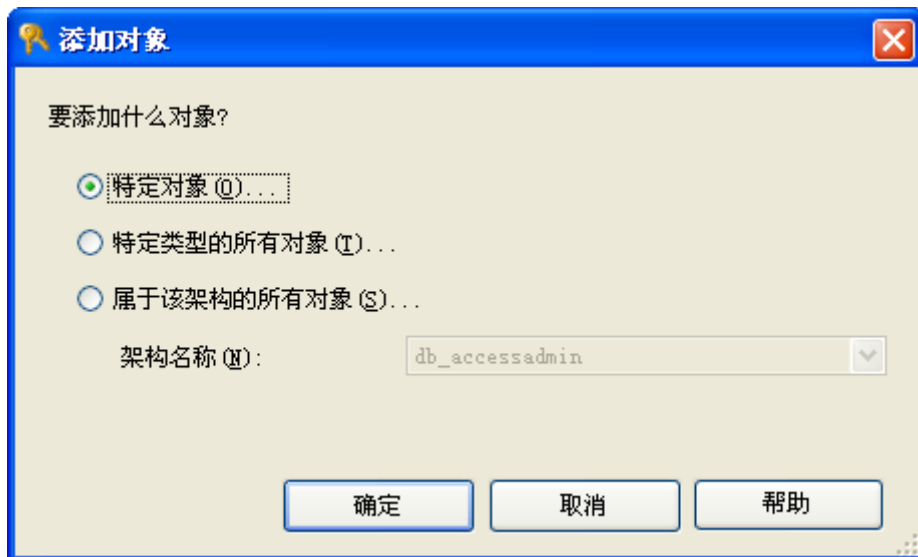


图 2-5 用户授权中选择对象类型

选择相应的对象，如图 2-5 中只选择了表 Person.Address，可以选择多个对象。

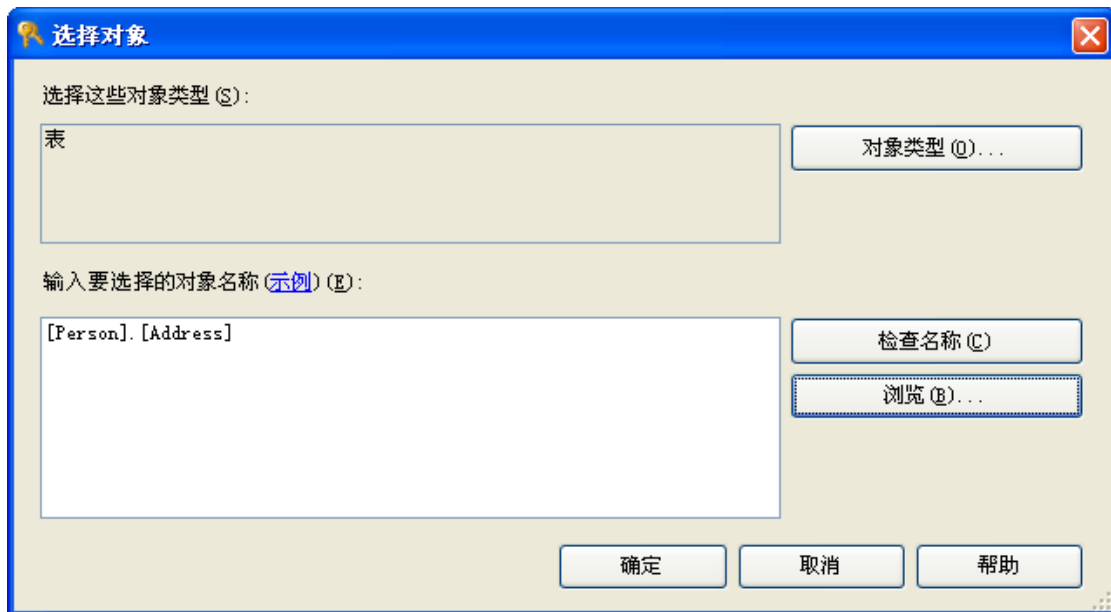


图 2-6 选择授权的数据库对象

- (6) 点击“确定”按钮后回到修改“安全对象”属性对话框，如图 2-7 所示。在该对话框中选择 Select 的授权。它也可以使用如下的命令完成：

```
use AdventureWorks
GO
grant select on Person.Address to application
GO
```

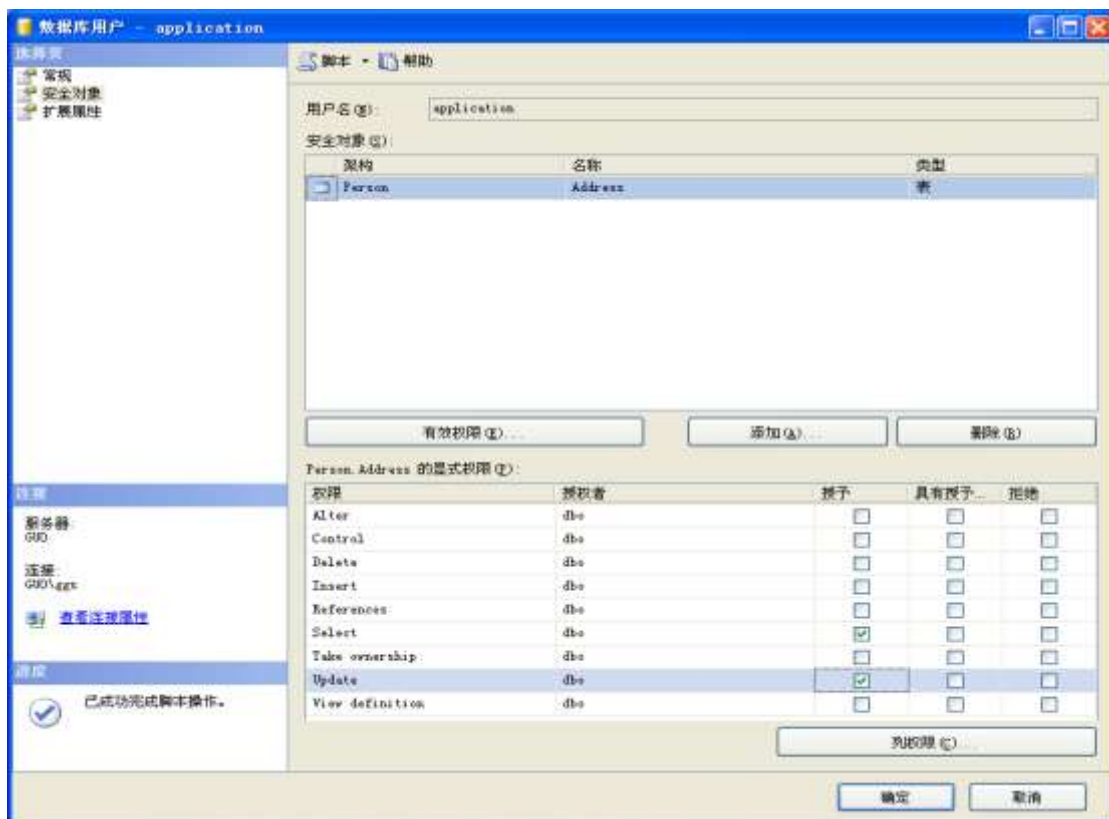


图 2-7 授予用户操作数据库对象 Person.Address 的权限

- (7) 选择 Update 权限，点击“列权限”按钮将更新某些列的权限授予用户，如图 2-8。在此，我们授予更新列 AddressLine1、AddressLine2、City 和 PostalCode 的权限。

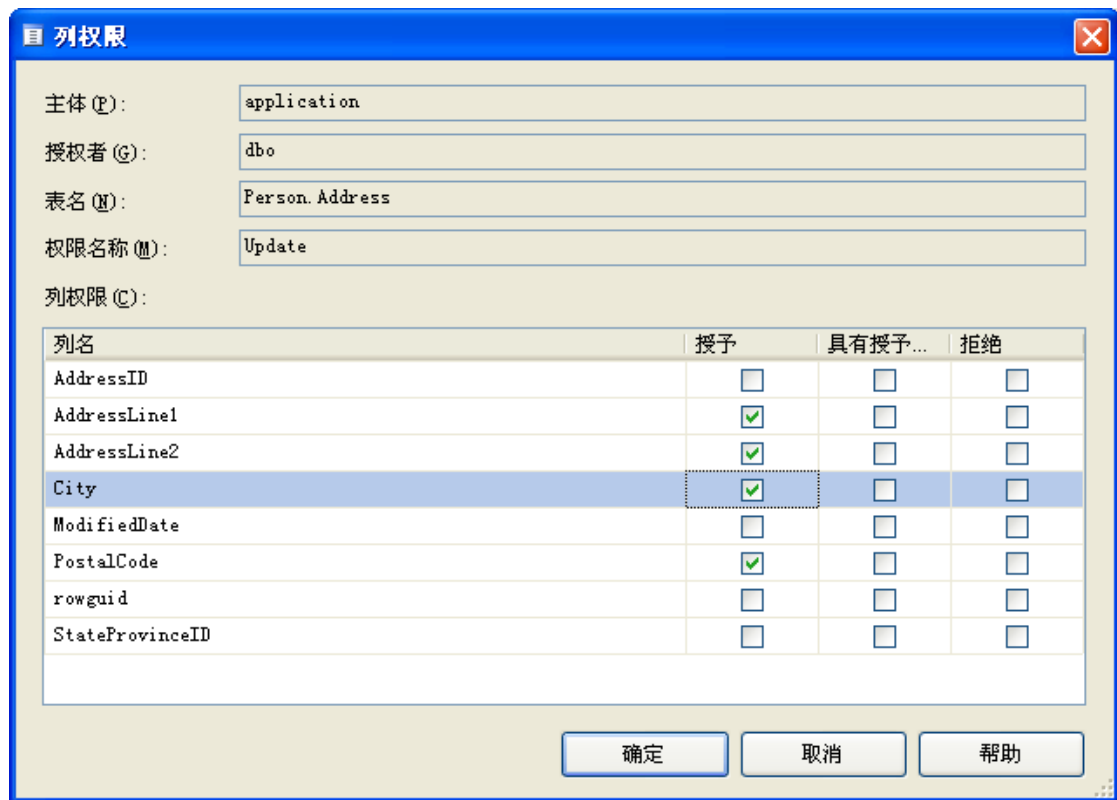


图 2-8 授予操作某些列的权限

更新这些列的授权操作也可以使用下面的命令完成：

```
use AdventureWorks
GO
GRANT UPDATE ON Person.Address (AddressLine1,AddressLine2,City,PostalCode)
TO application AS dbo
GO
```

- (8) 点击图 2-8 中的“确定”完成授权的操作。

通过上面的授权操作，使用登录 application 连接数据库后可以对数据库 AdventureWorks 中的 Person.Address 进行查询所有列和更新表中的 AddressLine1、AddressLine2、City 和 PostalCode 列的操作。下面演示经过授权的操作和没有经过授权的操作的结果。

- (9) 使用 application 登录连接查询分析器，在查询分析器中输入查询表

Person.Address 的命令并执行，得到的结果如所示。

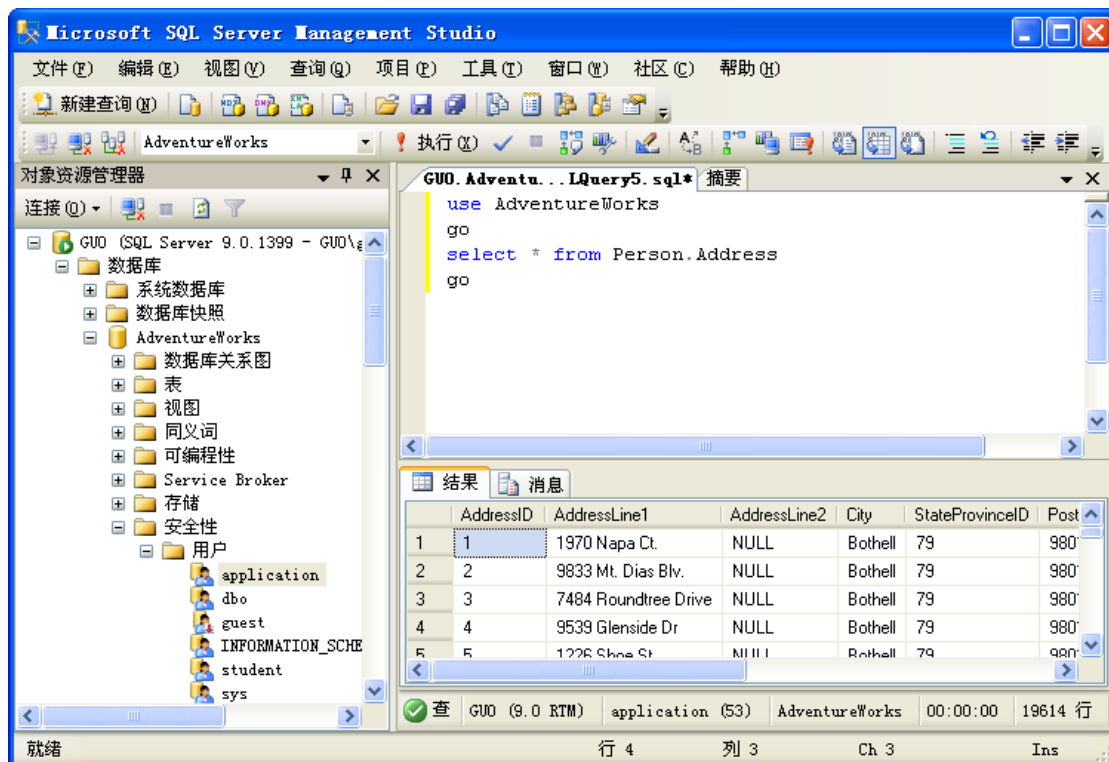


图 2-9 查询数据库的表

- (10) 输入相应的命令查询表 Person.AddressType，得到的结果如所示，这是因为该登录没有将查询该表的权限。

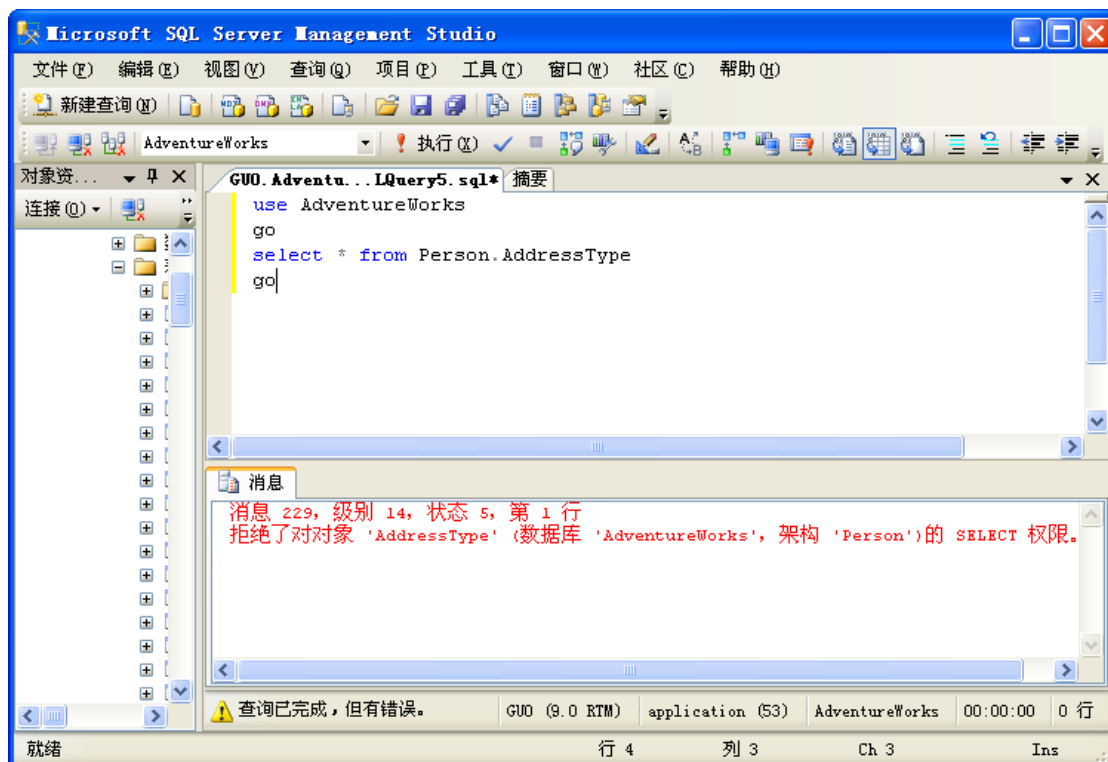


图 2-10 没有授权的查询操作

- (11) 更新表中的列，在查询分析器中使用命令更新表 Person.Address 的列

AddressLine1, 如图 2-11 所示。

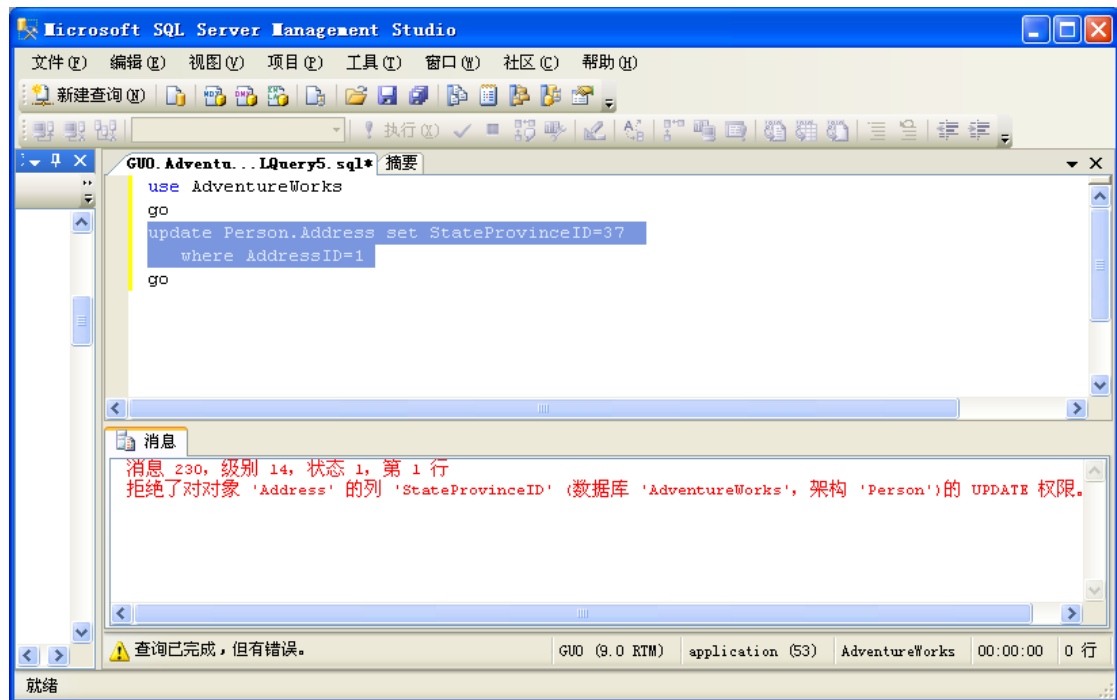


图 2-11 更新表中经过授权的列

- (12) 更新表 Person.Address 中的列 StateProvinceID, 这是一个没有操作权限的列, 结果如图 2-12 所示。

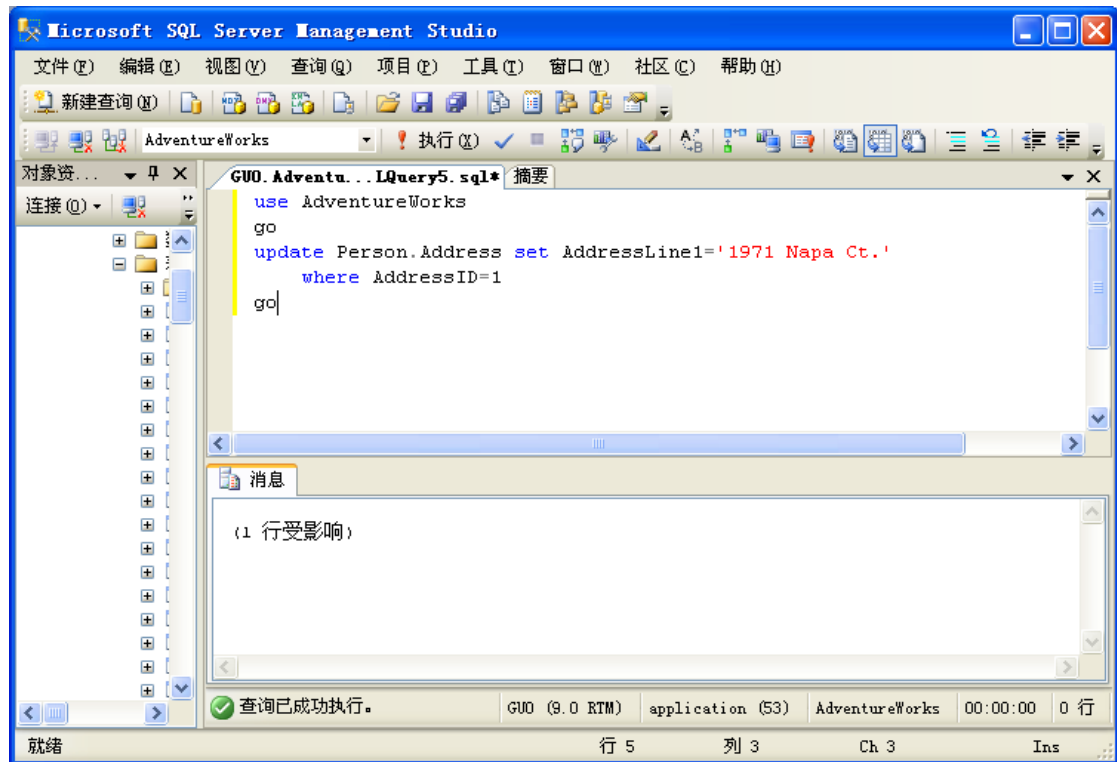


图 2-12 更新表中没有授权的列

2.5. 实验目的、内容与要求

2.5.1. 实验目的

通过本实验，掌握 SQL Server 安全管理中的登录、数据库服务器角色、用户、数据库角色、特定对象权限等基本概念与安全机制，掌握数据库服务器角色授权、数据库角色授权和特定对象授权的方法与各种方法的差异。

2.5.2. 实验内容

实验一 SQL Server 的基本安全管理

使用管理员登录连接到数据库服务器创建一个 SQL Server 登录并使用它完成如下的操作：

创建一个数据库，在数据库中创建一个表，创建表可以使用如下的命令。

```
CREATE TABLE StudentInfo (id char(10) primary key,  
name varchar(10) not null);
```

具体的实验内容：

- (1) 创建一个登录并将创建的数据库设置为该登录的默认数据库。
- (2) 设置登录在默认数据库中的用户映射。
- (3) 尝试将登录加入到两个不同的数据库服务器角色中对默认数据库进行操作（如创建表 **Score**，查询表 **StudentInfo** 等）。注意每次只使用一个数据库服务器角色来验证授权。
- (4) 尝试使用两个不同的数据库角色进行相应的授权操作。
- (5) 只使用特定对象授权，完成相应的操作以验证授权的成功和没有授权时发生的错误。

2.5.3. 实验要求

实验报告中要求给出实验内容中（1）—（5）的每一个的描述。其中，

- (1) 使用 **T-SQL 命令而非图形界面描述具体的操作过程**。
- (2) 内容（3）要求使用不同的服务器角色进行实验，要求至少使用 2 个以上的服务器角色进行测试，简单描述如何将登录加入到服务器角色的操作过程，给出加入不同服务器角色的测试方法与测试结果。注意，加入服务器角色是分别加入，是即加入一个服务器角色进行测试后，将其从服务器角色中删除然后再加入另一个服务器角色。验证授权前后及不同服务器角色授权之间的差异
- (3) 内容（4）与内容（3）类似，只是所使用的是数据库角色而不是服务器角色。
- (4) 内容（5）要求选择一个或多个具体的数据库对象（如表）设置查询、更新或插入的权限进行相应的测试。其中，有一部分内容必须是仅能够对部分列具有操作权限而对其它列没有操作权限。对于有操作权限与没有操作权限的都要进

行相应的测试，给出测试的方法、结果与结果分析。要求结合在数据库原理中所学习的 SQL 语言的知识使用命令完成而不是使用图形界面。

每个小的实验都必须给出具体的实验方法、实验结果与结果分析。在对每个实验结果的分析中要描述产生正确结果或错误的原因，有些可以写出相应的解决方法。例如，操作了对某个表的查询是因为登录 XXX 在数据库 YYY 中映射到用户 UUU，而用户又属于数据库角色 DB_ROLEX，数据库的角色 DB_ROLEX 具有对数据库的...操作权限，所以能够查询该表。对于不能完成的操作也必须做出相应的分析。

第3章 表

3.1. 表的概念

表是包含数据库中所有数据的数据库对象。表定义是一个列集合。数据在表中的组织方式与在电子表格中相似，都是按行和列的格式组织的。每一行代表一条唯一的记录，每一列代表记录中的一个字段。例如，在包含公司雇员数据的表中，每一行代表一名雇员，各列分别代表该雇员的信息，如雇员编号、姓名、地址、职位以及家庭电话号码等。

表包括下列主要组件：

◆ 列

每一列代表由表建模的对象的某个属性，例如，一个部件表有 ID 列、颜色列和重量列。

◆ 行

每一行代表由表建模的对象的一个单独的实例。例如，公司运送的每个部件在部件表中均占一行。

下图显示了 AdventureWorks 示例数据库中的 HumanResources.Department 表。

表 - HumanR. . . Department 摘要				
	DepartmentID	Name	GroupName	ModifiedDate
▶	1	Engin	Research and D...	2007-10-23 21:...
	2	Tool Design	Research and D...	1998-6-1 0:00:00
	3	Sales	Sales and Marke...	1998-6-1 0:00:00
	4	Marketing	Sales and Marke...	1998-6-1 0:00:00
	5	Purchasing	Inventory Mana...	1998-6-1 0:00:00
	6	Research and D...	Research and D...	1998-6-1 0:00:00
	7	Production	Manufacturing	1998-6-1 0:00:00
	8	Production Control	Manufacturing	1998-6-1 0:00:00
	9	Human Resources	Executive Gener...	1998-6-1 0:00:00
	10	Finance	Executive Gener...	1998-6-1 0:00:00
	11	Information Ser...	Executive Gener...	1998-6-1 0:00:00
	12	Document Control	Quality Assurance	1998-6-1 0:00:00
	13	Quality Assurance	Quality Assurance	1998-6-1 0:00:00
	14	Facilities and Mai...	Executive Gener...	1998-6-1 0:00:00
	15	Shipping and Re...	Inventory Mana...	1998-6-1 0:00:00
	16	Executive	Executive Gener...	1998-6-1 0:00:00
*	NULL	NULL	NULL	NULL

3.2. 数据完整性

指定表域的第一步是确定列数据类型。域是列中允许的值的集合。域不仅包括强制数据类型的概念，还包括列中允许的值。例如，`Production.Product` 表中的 `Color` 列的域包括 `nvarchar` 数据类型和大小限制（15 个字符）。该域还可指定列中所允许的字符串，如 `Red`、`Blue`、`Green`、`Yellow`、`Brown`、`Black`、`White`、`Teal`、`Gray` 和 `Silver`。

空值

列可以接受空值，也可以拒绝空值。在数据库中，`NULL` 是一个特殊值，表示未知值的概念。`NULL` 不同于空字符或 `0`。实际上，空字符是一个有效的字符，`0` 是一个有效的数字。`NULL` 只是表示此值未知这一概念。`NULL` 也不同于零长度字符串。如果列定义中包含 `NOT NULL` 子句，则不能为该行插入值为 `NULL` 的行。如果列定义中仅包含 `NULL` 关键字，则接受 `NULL` 值。

列中允许 `NULL` 值可以增加使用该列的所有逻辑比较的复杂程度。`SQL-92` 标准规定：对 `NULL` 值的任何比较都不能取值为 `TRUE` 或 `FALSE`，而是取值为 `UNKNOWN`。此规定在比较运算符中引入了三值逻辑，而要正确运用该逻辑很困难。

约束、规则、默认值和触发器

表列中除了具有数据类型和大小属性之外，还有其他属性。其他属性是保证数据库中数据完整性和表的引用完整性的重要部分。

- ◆ 数据完整性是指列中每个事件都有正确的数据值。数据值的数据类型必须正确，并且数据值必须位于正确的域中。
- ◆ 引用完整性指示表之间的关系得到正确维护。一个表中的数据只应指向另一个表中的现有行，不应指向不存在的行。

约束

通过约束可以定义 `SQL Server 2005` 数据库引擎 自动强制实施数据库完整性的方式。约束定义关于列中允许值的规则，是强制实施完整性的标准机制。使用约束优先于使用 `DML` 触发器、规则和默认值。另外，查询优化器也使用约束定义生成高性能的查询执行计划。

`SQL Server 2005` 支持下列约束类：

- ◆ `NOT NULL` 指定列不接受 `NULL` 值。
- ◆ `CHECK` 约束通过限制可放入列中的值来强制实施域完整性。`CHECK` 约束指定应用于为列输入的所有值的布尔值（计算结果为 `TRUE`、`FALSE` 或未知）搜索条件。所有计算结果为 `FALSE` 的值均被拒绝。可以为每列指定多个 `CHECK` 约束。
- ◆ `UNIQUE` 约束强制实施列集中值的唯一性。根据 `UNIQUE` 约束，表中的任何两行都不能有相同的列值。另外，主键也强制实施唯一性，但主键不允许 `NULL` 作为一个唯一值。

- ◆ **PRIMARY KEY** 约束标识具有唯一标识表中行的值的列或列集。在一个表中，不能有两行具有相同的主键值。不能为主键中的任何列输入 **NULL** 值。建议使用一个小的整数列作为主键。每个表都应有一个主键。限定为主键值的列或列组合称为候选键。



3.2.1. PRIMARY KEY 约束

表通常具有包含唯一标识表中每一行的值的一列或一组列。这样的一列或多列称为表的主键 (**PK**)，用于强制表的实体完整性。在创建或修改表时，您可以通过定义 **PRIMARY KEY** 约束来创建主键。

一个表只能有一个 **PRIMARY KEY** 约束，并且 **PRIMARY KEY** 约束中的列不能接受空值。由于 **PRIMARY KEY** 约束可保证数据的唯一性，因此经常对标识列定义这种约束。

如果为表指定了 **PRIMARY KEY** 约束，则 **SQL Server 2005** 数据库引擎 将通过为主键列创建唯一索引来强制数据的唯一性。当在查询中使用主键时，此索引还可用来对数据进行快速访问。因此，所选的主键必须遵守创建唯一索引的规则。

如果对多列定义了 **PRIMARY KEY** 约束，则一列中的值可能会重复，但来自 **PRIMARY KEY** 约束定义中所有列的任何值组合必须唯一。

可以在创建表时创建单个 **PRIMARY KEY** 约束作为表定义的一部分。如果表已存在，且没有 **PRIMARY KEY** 约束，则可以添加 **PRIMARY KEY** 约束。一个表只能有一个 **PRIMARY KEY** 约束。

如果已存在 **PRIMARY KEY** 约束，则可以修改或删除它。例如，可以让表的 **PRIMARY KEY** 约束引用其他列，更改列的顺序、索引名、聚集选项或 **PRIMARY KEY** 约束的填充因子。但是，不能更改使用 **PRIMARY KEY** 约束定义的列长度。

- ◆ **FOREIGN KEY** 约束标识并强制实施表之间的关系。有关详细信息，请参阅 **FOREIGN KEY** 约束。

一个表的外键指向另一个表的候选键。在下面的示例中，学生选课表 **SC** 建立了引用以前定义的学生基本信息表 **Student** 的外键。

如果一个外键值没有候选键，则不能向行中插入该值(**NULL** 除外)。**ON DELETE** 子句控制尝试删除现有外键指向的行时采取的操作。**ON DELETE** 子句包含以下几个选项：

- **NO ACTION** 指定删除失败并提示错误。
- **CASCADE** 指定还将删除包含指向已删除行的外键的所有行。
- **SET NULL** 指定将包含指向已删除行的外键的所有行都设置为 **NULL**。
- **SET DEFAULT** 指定将包含指向已删除行的外键的所有行都设置为它们的默认值。有关详细信息，请参阅默认值。

ON UPDATE 子句定义尝试更新现有外键指向的候选键值时采取的操作。另外，此子句也支持 NO ACTION、CASCADE、SET NULL 和 SET DEFAULT 选项。

列约束和表约束

约束可以是列约束，也可以是表约束。列约束指定为列定义的一部分，并且只应用于该列。前面的示例中的约束是列约束。表约束的声明与列定义无关，可以应用于表中多个列。当一个约束中必须包含多个列时，必须使用表约束。

例如，如果一个表的主键内有两个或两个以上的列，则必须使用表约束将这些列包含在主键内。假设有一个记录学生成绩的表，其中包括学号（StudentID）、所选课程编号（CourseID）和该门课程的成绩（Grade），则该表的主键为（学号，课程编号）。可以通过在表的两列主键中同时包含 StudentID 列和 CourseID 列强制实施这一点，如以下示例中所示：

```
CREATE TABLE SC (
    StudentID int FOREIGN KEY REFERENCES Student (StudentID),
    CourseID int FOREIGN KEY REFERENCES Course (CourseID),
    Grade numeric,
    CONSTRAINT PK PRIMARY KEY (StudentID, CourseID)
)
```

3.3. 创建和修改表

下面的命令创建一个学生基本信息表 Student，表中包括的列有：学号、姓名、性别、年龄和班级。

```
CREATE TABLE Student (
    StudentID int PRIMARY KEY,
    Name nvarchar(32) not null,
    Sex char(1) CHECK (Sex in ('F', 'f', 'M', 'm')),
    Age int,
    Class char(8)
)
```

说明：在计算机的表示的方法中，括号“()”常用于表示集合，而集合中的元素使用的分隔符则为逗号“，”，集合中的元素是无序的，因此，列定义集合中的列的位置是没有先后顺序的。

上面的命令创建一个名称为 Student 的表，它有三个列 StudentID、Name、Sex、Age 和 Class，分别表示学号、姓名、性别、年龄和班级。StudentID 的属性有数据类型 int 和主键约束 PRIMARY KEY。

SQL Server 联机丛书有关于 CREATE TABLE 命令详细详细说明，但大多数情况下你可以把 CREATE TABLE 命令看作下面的一种简化形式：

CREATE TABLE 表名 列的定义

其中，列的定义是一个集合，包括每一个列的定义，形式如（列 1 的定义，列 2 的定义...，列 n 的定义），而每个列的定义又包括三部分：列名、数据类型和约束。列定义中的列名和数据类型是在创建表时必须指定的，即每个列都必须有列名和数据类型，而约束是可选的，有些列没有约束，而有约束的列可以在创建表时指定，也可以在创建表后添加。

实际上，SQL 中的创建表的命令可以进一步抽象为形式如下的创建对象的命令，

命令 对象 对象的定义

比如创建索引、视图等都符合上面的抽象形式。在学习 SQL 的命令时，先分析命令的结构有助于加强对命令的理解和易于记忆。

每个表至多可定义 1024 列。表和列的名称必须遵守标识符的规定，在特定表中必须是唯一的，但同一数据库的不同表中可使用相同的列名。必须为每列指定数据类型。

3.4. 查询表

3.5. 自动编号列和标识符列

对于每个表，均可创建一个包含系统生成的序号值的标识符列，该序号值以唯一方式标识表中的每一行。例如，当在表中插入行时，标识符列可自动为应用程序生成唯一的编号。标识符列在其所定义的表中包含的值通常是唯一的。

每个表中均可创建一个全局唯一标识符列，该列中包含在全球联网的所有计算机中不重复的值。当必须合并来自多个数据库系统的相似数据时（例如，在一个客户帐单系统中，其数据位于世界各地的分公司），通常需要保证列包含全局唯一值。当数据被汇集到中心以进行合并和制作报表时，使用全局唯一值可防止不同国家/地区的客户具有相同的帐单号或客户 ID。

3.5.1. IDENTITY 属性

通过使用 IDENTITY 属性可以实现标识符列。这使得开发人员可以为表中所插入的第一行指定一个标识号（Identity Seed 属性），并确定要添加到种子上的增量（Identity Increment 属性）以确定后面的标识号。将值插入到有标识符列的表中之后，SQL Server 2005 数据库引擎会通过向种子添加增量来自动生成下一个标识值。当向现有表中添加标识符列时，还会将标识号添加到现有表行中，并按照最初添加这些行的顺序开始应用种子值和增量值。同时还为所有新添加的行生成标识号。不能修改现有表列来添加 IDENTITY 属性。

在用 IDENTITY 属性定义标识符列时，注意下列几点：

- (1) 一个表只能有一个使用 IDENTITY 属性定义的列，且必须通过使用 decimal、

int、numeric、smallint、bigint 或 tinyint 数据类型来定义该列。

- (2) 可指定种子和增量。二者的默认值均为 1。
- (3) 标识符列不能允许为空值，也不能包含 DEFAULT 定义或对象。
- (4) 在设置 IDENTITY 属性后，可以使用 \$IDENTITY 关键字在选择列表中引用该列。还可以通过名称引用该列。
- (5) OBJECTPROPERTY 函数可用于确定一个表是否具有 IDENTITY 列，COLUMNPROPERTY 函数可用于确定 IDENTITY 列的名称。
- (6) 通过使值能够显式插入，SET IDENTITY_INSERT 可用于禁用列的 IDENTITY 属性。

语法

IDENTITY [(seed , increment)]

以下示例将使用 IDENTITY 属性，为自动递增标识号创建一个新表。

参数

Seed: 装载到表中的第一个行使用的值。

increment : 与前一个加载的行的标识值相加的增量值。

必须同时指定种子和增量，或者二者都不指定。如果二者都未指定，则取默认值 (1,1)。

备注

如果在经常进行删除操作的表中存在着标识列，那么在标识值之间可能会有间隔。如果这是要考虑的问题，那么请不要使用 IDENTITY 属性。但是，为了确保未产生间隔，或者填补现有的间隔，在用 SET IDENTITY_INSERT ON 显式输入标识值之前，请先对现有的标识值进行计算。

如果要重新使用已删除的标识值，则可使用示例 B 中的示例代码来查找下一个可用的标识值。使用表名称、标识列数据类型和（该数据类型）的最大允许值数值 -1 来替代 tablename、column_type 和 MAX(column_type) - 1。

使用 DBCC CHECKIDENT 检查当前的标识值，并将其与标识列中的最大值进行比较。

如果发布了包含标识列的表进行复制，则必须使用与所用复制方式相应的方式来管理标识列。有关详细信息，请参阅复制标识列。

```

USE AdventureWorks
IF OBJECT_ID ('dbo.new_employees', 'U') IS NOT NULL
    DROP TABLE new_employees
GO
CREATE TABLE new_employees
(
    id_num int IDENTITY(1,1),
    fname varchar (20),
    minit char(1),
    lname varchar(30)
)

INSERT new_employees
    (fname, minit, lname)
VALUES
    ('Karin', 'F', 'Josephs')

INSERT new_employees
    (fname, minit, lname)
VALUES
    ('Pirkko', 'O', 'Koskitalo')

```

3.5.2. SET IDENTITY_INSERT

语法

```
SET IDENTITY_INSERT [ database_name . [ schema_name ] . ] table { ON | OFF }
```

参数

database_name: 指定的表所在的数据库的名称。

schema_name : 表所属的架构的名称。

table : 包含标识列的表的名称。

备注

任何时候，一个会话中只有一个表的 **IDENTITY_INSERT** 属性可以设置为 **ON**。如果某个表已将此属性设置为 **ON**，则对另一个表发出 **SET IDENTITY_INSERT ON** 语句时，SQL Server 2005 将返回一个错误信息，指出 **SET IDENTITY_INSERT** 已设置为 **ON**，并报告已将其属性设置为 **ON** 的表。

如果插入值大于表的当前标识值，则 SQL Server 自动将新插入值作为当前标识值使用。

SET IDENTITY_INSERT 的设置是在执行或运行时设置的，而不是在分析时设置的。

权限

用户必须是对象的所有者，或者是 **sysadmin** 固定服务器角色的成员，或者是 **db_owner** 或 **db_ddladmin** 固定数据库角色的成员。

以下示例将创建一个包含标识列的表，并显示如何使用 `SET IDENTITY_INSERT` 设置来填充由 `DELETE` 语句导致的标识值中的空隙。

```
-- 生成一个有间隔的标识列。  
DELETE dbo.Tool  
WHERE Name = 'Saw'  
GO  
  
SELECT *  
FROM dbo.Tool  
GO  
  
-- 试图插入标识列的值为3;  
-- 将会返回一个警告信息。  
INSERT INTO dbo.Tool (ID, Name) VALUES (3, 'Garden shovel')  
GO  
  
-- SET IDENTITY_INSERT 设置成 ON。  
SET IDENTITY_INSERT dbo.Tool ON  
GO  
  
-- 试图插入ID的值为3。  
INSERT INTO dbo.Tool (ID, Name) VALUES (3, 'Garden shovel')  
GO  
  
SELECT *  
FROM dbo.Tool  
GO  
  
-- 删除生成的表。  
DROP TABLE dbo.Tool  
GO
```

说明：插入标识列时必须显式给出标识列的名称，如果使用下面的语句则即使 `IDENTITY_INSERT` 设置为 `ON` 时也会出现同样的警告信息。

```
INSERT INTO dbo.Tool (ID, Name) VALUES (3, 'Garden shovel')
```

3.5.3. 使用常规语法查找标识值之间的间隔

以下示例显示了删除了数据时，用于在标识值中查找间隔的常规语法。

```
-- 这是一个在表中的标识列中查找间隔值的示例。
-- 表的名称为img，它有两个列：
-- id_num，是一个增长的标识列，
-- company_name，表示公司的名称

-- 创建img 表。
-- 如果img 表已经存在则删除它，然后创建该表。
IF OBJECT_ID ('dbo.img', 'U') IS NOT NULL
    DROP TABLE img
GO
CREATE TABLE img (id_num int IDENTITY(1,1), company_name sysname)
INSERT img(company_name) VALUES ('New Moon Books')
INSERT img(company_name) VALUES ('Lucerne Publishing')
-- SET IDENTITY_INSERT ON 使得可以插入标识列的值。
SET IDENTITY_INSERT img ON
select * from img
insert into img(id_num,company_name) values(1,'IBM')
DECLARE @minidentval smallint
DECLARE @nextidentval smallint
SELECT @minidentval = MIN($IDENTITY) FROM img
IF @minidentval = IDENT_SEED('img')
    SELECT @nextidentval = MIN($IDENTITY) + IDENT_INCR('img')
    FROM img t1
    WHERE $IDENTITY BETWEEN IDENT_SEED('img') AND 32766 AND
        NOT EXISTS (SELECT * FROM img t2
            WHERE t2.$IDENTITY = t1.$IDENTITY + IDENT_INCR('img'))
ELSE
    SELECT @nextidentval = IDENT_SEED('img')
SET IDENTITY_INSERT img OFF
```

3.5.4. 全局唯一标识符

尽管 **IDENTITY** 属性在一个表内自动进行行编号，但具有各自标识符列的各个表可以生成相同的值。这是因为 **IDENTITY** 属性仅在使用它的表上保证是唯一的。如果应用程序必须生成在整个数据库或世界各地所有网络计算机的所有数据库中均为唯一的标识符列，请使用 **uniqueidentifier** 数据类型和 **NEWID** (Transact-SQL) 或 **NEWSEQUENTIALID()** 函数。数据库引擎不会自动为该列生成值。若要插入全局唯一值，请为该列创建 **DEFAULT** 定义来使用 **NEWID** 或 **NEWSEQUENTIALID** 函数生成全局唯一值。

在设置 **ROWGUIDCOL** 属性后，通过使用 **\$ROWGUID** 关键字可以在选择列表中引用该列。这与通过使用 **\$IDENTITY** 关键字可以引用 **IDENTITY** 列的方法类似。一个表只能有一个 **ROWGUIDCOL** 列，且必须通过使用 **uniqueidentifier** 数据类型定义该列。

OBJECTPROPERTY (Transact-SQL) 函数可用于确定一个表是否具有 **ROWGUIDCOL** 列，**COLUMNPROPERTY** (Transact-SQL) 函数可用于确定 **ROWGUIDCOL** 列的名称。

以下示例创建 `uniqueidentifier` 列作为主键的表。此示例在 `DEFAULT` 约束中使用 `NEWSEQUENTIALID()` 函数为 newRow 提供值。将 `ROWGUIDCOL` 属性应用到 `uniqueidentifier` 列，以便可以使用 `$ROWGUID` 关键字对其进行引用。

```
CREATE TABLE Globally_Unique_Data
(guid uniqueidentifier CONSTRAINT Guid_Default
    DEFAULT NEWSEQUENTIALID() ROWGUIDCOL,
Employee_Name varchar(60)
CONSTRAINT Guid_PK PRIMARY KEY (Guid) );
```

3.6. 基本数据定义与查询

实验目的：掌握对数据库表的更新操作，掌握对视图的操作，理解对基本表和视图操作的异同。

内容与要求：使用 SQL Server 的查询分析器完成下列数据定义和查询(注：不要使用企业管理器对表进行管理，应使用命令创建和修改表)。

按表 2-1 和表 2-2 提供的内容建立客户表和合同表

表 2-1 客户基本信息表 CUSTOMER

编号	字段	数据类型	约束	说明
1	ID	CHAR (10)	主码	客户编号
2	NAME	VARCHAR (10)	非空	客户姓名
4	DEPARTMENT	VARCHAR (32)	非空	客户单位名称
5	PHONE	VARCHAR (12)		客户电话
6	MOBILE	CHAR (11)		手机
7	EMAIL	VARCHAR (32)		电子邮件地址
8	CITY	VARCHAR (20)		所在城市
9	ADDRESS	VARCHAR(128)		通信地址
10	ZIP	CHAR (6)		邮政编码

表 2-2 合同基本信息表 CONTRACT

编号	字段	数据类型	约束	说明
1	ID	Char(12)	主码	合同编号
2	CUSTOMER_ID	CHAR (12)	外码， 非空	引用客户表 CUSTOMER
3	SIGN_DATE	DATETIME		合同签订日期
4	AMOUNT	MONEY	非空	合同总额
5	CONTENTS	VARCHAR (512)		合同内容

给客户表添加一个表示性别的字段 **SEX**，数据类型为 **CHAR (1)**，取值范围为 (F, M, f, m) 的其中一个（练习表结构的修改和用户定义的完整性约束）

在客户表名称字段上建立的索引（按升序排列）

在客户表中插入相应的不少于 5 个的记录，在合同表中插入不少于 3 个的合同记录。

查询客户的姓名、单位、电话、手机和 **EMAIL** 地址。查询合同表的信息，输出合同编号、客户单位名称、签订日期和合同总额（注：需要与客户表进行联系查询）

3.7. 复杂查询

实验目的：掌握对的数据库表的更新操作，掌握对视图的操作，理解对基本表和视图操作的异同。

实验内容与要求

在原来的客户与合同信息表的基础上添加一个新的合同应收帐目明细表，如表 2-3 所示：

表 2-3 应收帐目信息表 COST

编号	字段	数据类型	约束	说明
1	ID	CHAR(10)	主码	应收帐目编号
2	CONTRAC_ID	CHAR(12)	外码	引用合同编号
3	AMOUNT	MONEY	非空	费用金额
4	TYPE	INT，值为 0 或 1	非空	费用类别(预付款、尾款)
5	RECEIVE_DATA	DATETIME	非空	应到日期

在合同中添或修改相应的记录，每个合同中每个客户有两个以上的合同，然后在应收账目信息表中为每一项合同添加预付款和尾款各一项记录

统计合同总额（所有合同的总额之和）

根据应收账目信息表，输出统计每个单项合同的总额（预付款+尾款，使用分组查询统计），输出内容如下所示：

合同编号	总额
合同编号 1	10,000,000.00
合同编号 2	12,000,000.00

在客户、合同和合同应收账目三个表的基础上创建一个视图，包括的内容有合同编号、客户单位名称、合同总额、签订日期、应收帐目费用、费用类别和应到日期。

使用该视图修改合同总额。

3.8. 示例数据库

SQL Server 包括 AdventureWorks (OLTP)、AdventureWorksDW (数据仓库) 和 AdventureWorksAS (Analysis Services) 等示例数据库。示例数据基于一个虚拟的公司 Adventure Works Cycles，这是一个大型的跨国制造公司，它生产金属和复合材料自行车，产品远销北美、欧洲和亚洲市场。

3.8.1. Adventure Works Cycles 业务方案

Adventure Works Cycles 是 AdventureWorks 示例数据库所基于的虚构公司，是一家大型跨国生产公司。公司生产金属和复合材料的自行车，产品远销北美、欧洲和亚洲市场。公司总部设在华盛顿州的伯瑟尔市，拥有 299 名雇员，而且拥有多个活跃在世界各地的地区性销售团队。

在 2000 年，Adventure Works Cycles 购买了位于墨西哥的小型生产厂 Importadores Neptuno。Importadores Neptuno 为 Adventure Works Cycles 产品生产多种关键子组件。这些子组件将被运送到伯瑟尔市进行最后的产品装配。2001 年，Importadores Neptuno 转型成为专注于旅行登山车系列产品的制造商和销售商。

实现一个成功的财务年度之后，Adventure Works Cycles 希望通过以下方法扩大市场份额：专注于向高端客户提供产品、通过外部网站扩展其产品的销售渠道、通过降低生产成本来削减其销售成本。

表 2-4 SalesOrderHeader 表定义

列	数据类型	非空	说明
SalesOrderID	int	非空	主键
RevisionNumber	tinyint	非空	随着时间的推移跟踪销售订单更改的递增编号
OrderDate	datetime	非空	创建销售订单的日期
DueDate	datetime	非空	客户订单到期的日期
ShipDate	datetime	非空	订单发送给客户的日期
Status	tinyint	非空	订单的当前状态。1=处理中，2=已批准，3=预定，4=已拒绝，5=已发货，6=已取消
OnlineOrderFlag	Flag (bit)	非空	0=销售人员下的订单，1=客户在线下的订单。
SalesOrderNumber	nvarchar(25)	非空	唯一的销售订单标识号。
PurchaseOrderNumber	OrderNumber (nvarchar(25))	非空	客户采购订单号引用
AccountNumber	AccountNumber (nvarchar(15))	非空	财务账号引用
CustomerID	int	非空	客户标识号，指向 Customer.CustomerID 的外键
ContactID	int	非空	客户联系人标识号，指向 Contact.ContactID 的外键
SalesPersonID	int	非空	创建销售订单的销售人员，指向 SalesPerson.SalesPersonID 的外键
TerritoryID	int	非空	进行销售的地区，指向 SalesTerritory.SalesTerritoryID 的外键
BillToAddressID	int	非空	客户开票地址，指向 Address.AddressID 的外键
ShipToAddressID	int	非空	客户收货地址，指向 Address.AddressID 的外键
ShipMethodID	int	非空	发货方法，指向 ShipMethod.ShipMethodID 的外键

CreditCardID	int	非空	信用卡标识号，指向 CreditCard.CreditCardID 的外键
CreditCardApprovalCode	varchar(15)	非空	信用卡公司提供的批准代码
CurrencyRateID	int	非空	所使用的外币兑换率，指向 CurrencyRate.CurrencyRateID 的外键
SubTotal	money	非空	销售小计，SalesOrderID 的销售小计的计算方式为 SUM(SalesOrderDetail.LineTotal)
TaxAmt	money	非空	税额
Freight	money	非空	运费
TotalDue		非空	客房的应付款总计，计算方式为 SubTotal+TaxAmt+Freight
Comment	nvarchar(128)	非空	销售代表添加的注释。
Rowguid	uniqueidentifier ROWGUIDCOL	非空	唯一标识行的 ROWGUIDCOL 号，用于支持合并复制。
ModifiedDate	datetime	非空	行的上次更新日期和时间。

表 2-5 SalesOrderDetail 表的定义

列	数据类型	非空	说明
SalesOrderID	int	非空	主键，指向 SalesOrderHeader.SalesOrderID 的外键
SalesOrderDetailID	Tinyint	非空	主键，用于确保数据唯一性的连续编号。
CarrierTrackingNumber	Nvarchar(25)	非空	发货人提供的发货跟踪号。
OrderQty	Smallint	非空	每个产品的订购数量。
ProductID	Int	非空	销售给客户的产品，指向 SpecialOfferProduct.ProductID 的外键。
SpecialOfferID	Int	非空	促销代码，指向 SpecialOfferProduct.SpecialOfferID。
UnitPrice	money	非空	单件产品的销售价格。
UnitPriceDiscount	money	非空	折扣金额。
LineTotal		非空	每件产品的小计，计算方式为 OrderQty*UnitPrice
Rowguid	uniqueidentifier ROWGUIDCOL	非空	唯一标识行的 ROWGUIDCOL 号，用于支持合并复制。
ModifiedDate	datetime	非空	行的上一次修改的日期和时间。

第4章 Transact-SQL 编程

4.1. 程序中处理错误

4.1.1. RAISERROR 语法

生成错误消息并启动会话的错误处理。RAISERROR 可以引用 sys.messages 目录视图中存储的用户定义消息，也可以动态建立消息。该消息作为服务器错误消息返回到调用应用程序，或返回到 TRY...CATCH 构造的关联 CATCH 块。

语法

```
RAISERROR ( { msg_id | msg_str | @local_variable }  
           { ,severity ,state }  
           [ ,argument [ ,...n ] ] )  
           [ WITH option [ ,...n ] ]
```

◆ msg_id

使用 sp_addmessage 存储在 sys.messages 目录视图中的用户定义错误消息号。用户定义错误消息的错误号应当大于 50000。如果未指定 msg_id，则 RAISERROR 引发一个错误号为 50000 的错误消息。

◆ msg_str

用户定义消息，格式与 C 标准库中的 printf 函数类似。该错误消息最长可以有 2,047 个字符。如果该消息包含的字符数等于或超过 2,048 个，则只能显示前 2,044 个并添加一个省略号以表示该消息已被截断。请注意，由于内部存储行为的缘故，代替参数使用的字符数比输出所显示的字符数要多。例如，赋值为 2 的代替参数 %d 实际在消息字符串中生成一个字符，但是还会在内部占用另外三个存储字符串。此存储要求减少了可用于消息输出的字符数。

当指定 msg_str 时，RAISERROR 将引发一个错误号为 5000 的错误消息。

msg_str 是一个字符串，具有可选的嵌入转换规格。每个转换规格都会定义参数列表中的值如何格式化并将其置于 msg_str 中转换规格位置上的字段中。转换规格的格式如下：

% [[flag] [width] [. precision] [{h | l}]] type

可在 msg_str 中使用的参数包括：

◆ flag

用于确定被替换值的间距和对齐的代码。具体的含义如下表所示，

表 4-1 RAISERROR 函数的 msg_str 参数中 flag 的含义

代码	前缀或对齐	说明
----	-------	----

- (减号)	左对齐	在给定字段宽度内左对齐参数值。
+ (加号)	符号前缀	如果参数值为有符号类型, 则在参数值的前面加上加号 (+) 或减号 (-)。
0 (零)	零填充	在达到最小宽度之前在输出前面加上零。如果出现 0 和减号 (-), 将忽略 0。
# (数字)	对 x 或 X 的十六进制类型使用 0x 前缀	当使用 o、x 或 X 格式时, 数字符号 (#) 标志在任何非零值的前面分别加上 0、0x 或 0X。当 d、i 或 u 的前面有数字符号 (#) 标志时, 将忽略该标志。
' ' (空格)	空格填充	如果输出值有符号且为正, 则在该值前加空格。如果包含在加号 (+) 标志中, 则忽略该标志。

◆ width

定义放置参数值的字段的最小宽度的整数。如果参数值的长度等于或大于 width, 则打印该值, 无需进行填充。如果该值小于 width, 则将该值填充到 width 中指定的长度。

星号 (*) 表示宽度由参数列表中的相关参数指定, 该宽度必须为整数值。

◆ precision

从字符串值的参数值中得到的最大字符数。例如, 如果一个字符串具有五个字符并且精度为 3, 则只使用字符串值的前三个字符。

对于整数值, precision 是指打印的最小位数。

星号 (*) 表示精度由参数列表中的相关参数指定, 该精度必须为整数值。

{h|l} type

与字符类型 d、i、o、x、X 或 u 一起使用, 用于创建 shortint (h) 值或 longint (l) 值。

类型规范	表示
d 或 i	有符号整数
o	无符号八进制数
s	字符串
u	无符号整数
x 或 X	无符号十六进制数

◆ @local_variable

是一个可以为任何有效字符数据类型的变量, 其中包含的字符串的格式化方式与 msg_str 相同。@local_variable 必须为 char 或 varchar, 或者能够隐式转换为这些数据类型。

◆ severity

用户定义的与该消息关联的严重级别。当使用 msg_id 引发使用 sp_addmessage 创

建的用户定义消息时，RAISERROR 上指定的严重性将覆盖 sp_addmessage 中指定的严重性。

任何用户都可以指定 0 到 18 之间的严重级别。只有 sysadmin 固定服务器角色成员或具有 ALTER TRACE 权限的用户才能指定 19 到 25 之间的严重级别。若要使用 19 到 25 之间的严重级别，必须选择 WITH LOG 选项。

20 到 25 之间的严重级别被认为是致命的。如果遇到致命的严重级别，客户端连接将在收到消息后终止，并将错误记录到错误日志和应用程序日志。

小于 0 的严重级别被解释为级别为 0。大于 25 的严重级别被解释为级别为 25。

◆ state

介于 1 至 127 之间的任意整数。state 的负值默认为 1。值为 0 或大于 127 会生成错误。

如果在多个位置引发相同的用户定义错误，则针对每个位置使用唯一的状态号有助于找到引发错误的代码段。

◆ argument

用于代替 msg_str 或对应于 msg_id 的消息中的定义的变量的参数。可以有 0 个或多个代替参数，但是代替参数的总数不能超过 20 个。每个代替参数都可以是局部变量或具有下列任一数据类型：tinyint、smallint、int、char、varchar、nchar、nvarchar、binary 或 varbinary。不支持其他数据类型。

◆ option

错误的自定义选项，可以是下表中的任一值。

表 4-2 RAISERROR 函数的参数 option 的值

值	说明
LOG	在 Microsoft SQL Server 数据库引擎实例的错误日志和应用程序日志中记录错误。记录到错误日志的错误目前被限定为最多 440 字节。只有 sysadmin 固定服务器角色成员或具有 ALTER TRACE 权限的用户才能指定 WITH LOG。
NOWAIT	将消息立即发送给客户端。
SETERROR	将 @@ERROR 值和 ERROR_NUMBER 值设置为 msg_id 或 50000，不用考虑严重级别。

RAISERROR 生成的错误与数据库引擎代码生成的错误的运行方式相同。RAISERROR 指定的值由 ERROR_LINE、ERROR_MESSAGE、ERROR_NUMBER、ERROR_PROCEDURE、ERROR_SEVERITY、ERROR_STATE 以及 @@ERROR 等系统函数来报告。当 RAISERROR 在严重级别为 11 或更高的情况下在 TRY 块中运行，它

便会将控制传输至关联的 CATCH 块。如果 RAISERROR 在下列情况下运行，便会将错误返回到调用方：

- 在任何 TRY 块的作用域之外运行。
- 在严重级别为 10 或更低的情况下在 TRY 块中运行。
- 在严重级别为 20 或更高的情况下终止数据库连接。

CATCH 块可以使用 RAISERROR 来再次引发调用 CATCH 块的错误，方法是使用 ERROR_NUMBER 和 ERROR_MESSAGE 之类的系统函数检索原始错误消息。对于严重级别为 1 到 10 的消息，@@ERROR 默认值为 0。

当 msg_id 指定 sys.messages 目录视图中可用的用户定义消息时，RAISERROR 按照与应用到使用 msg_str 指定的用户定义消息文本的规则相同的规则处理文本列中的消息。用户定义消息文本可以包含转换规格，并且 RAISERROR 将参数值映射到转换规格。使用 sp_addmessage 添加用户定义错误消息，而使用 sp_dropmessage 删除用户定义错误消息。

RAISERROR 可以代替 PRINT 将消息返回到调用应用程序。RAISERROR 支持类似于 C 标准库中 printf 函数功能的字符代替，而 Transact-SQL PRINT 语句则不支持。PRINT 语句不受 TRY 块的影响，而在严重级别为 11 到 19 的情况下在 TRY 块中运行的 RAISERROR 会将控制传输至关联的 CATCH 块。指定严重级别为 10 或更低以使用 RAISERROR 返回 TRY 块中的消息，而不必调用 CATCH 块。

通常，连续的参数替换连续的转换规格：第一个参数替换第一个转换规格，第二个参数替换第二个转换规格，以此类推。例如，在以下 RAISERROR 语句中，第一个参数 N'number' 替换第一个转换规格 %s，第二个参数 5 替换第二个转换规格 %d。

```
RAISERROR (N'This is message %s %d.', -- 信息,前缀N表示Unicode.
          10, -- 错误级别,
          1, -- 状态,
          N'number', -- 第个参数.
          5); -- 第个参数.
-- 返回的信息为: This is message number 5.
GO
```

如果为转换规格的宽度或精度指定了星号 (*), 则要用于宽度或精度的值被指定为整数参数值。在这种情况下，一个转换规格最多可以使用三个参数，分别用作宽度、精度和代替值。

例如，下列两个 RAISERROR 语句都返回相同的字符串。一个指定参数列表中的宽度值和精度值；另一个指定转换规格中的宽度值和精度值。

```
RAISERROR (N'<<%.3s>>', -- 信息.  
          10, -- 错误级别,  
          1, -- 状态,  
          7, -- 第个参数用于宽度.  
          3, -- 用于精度的参数.  
          N'abcde'); -- 第个参数用于字符串.  
-- 返回的信息为: <<   abc>>.  
GO  
RAISERROR (N'<<%.7.3s>>', -- 信息.  
          10, -- 错误级别,  
          1, -- 状态,  
          N'abcde'); -- 提供字符串的参数.  
-- 返回的信息为: <<   abc>>.  
--
```

4.1.2. RAISERROR 示例

RAISERROR 示例：从 CATCH 块返回错误消息

以下代码示例显示如何在 TRY 块中使用 RAISERROR 使执行跳至关联的 CATCH 块中。它还显示如何使用 RAISERROR 返回有关调用 CATCH 块的错误的信息。

```

BEGIN TRY
    -- 错误级别小于10时执行下面的语句且打印系统错误信息;
    -- 错误级别等于10, 则仅执行下面的语句;
    -- 错误级别大于10则会导致执行跳转到CATCH 块.
    RAISERROR ('Error raised in TRY block.', -- 信息.
              16, -- 错误级别.
              1 -- 状态.
              );
END TRY
BEGIN CATCH
    DECLARE @ErrorMessage NVARCHAR(4000);
    DECLARE @ErrorSeverity INT;
    DECLARE @ErrorState INT;

    SELECT
        @ErrorMessage = ERROR_MESSAGE(),
        @ErrorSeverity = ERROR_SEVERITY(),
        @ErrorState = ERROR_STATE();

    -- 在CATCH块内使用RAISERROR返回关于导致执行
    -- 跳转到CATCH块的原始错误的错误信息
    RAISERROR (@ErrorMessage, -- 信息.
              @ErrorSeverity, -- 错误级别.
              @ErrorState -- 状态.
              );

```

RAISERROR 示例：在 sys.messages 中创建即席消息

以下示例显示如何引发 sys.messages 目录视图中存储的消息。该消息通过 sp_addmessage 系统存储过程，以消息号 50005 添加到 sys.messages 目录视图中。

```

sp_addmessage @msgnum = 50005,
              @severity = 10,
              @msgtext = N'<<%7.3s>>';
GO
RAISERROR (50005, -- 住处标识ID.
          10, -- 错误级别,
          1, -- 状态,
          N'abcde'); -- 提供字符串的参数.
-- 返回的信息为: <<   abc>>.
GO
sp_dropmessage @msgnum = 50005;
GO

```

RAISERROR 示例：使用局部变量提供消息文本

以下代码示例显示如何使用局部变量为 RAISERROR 语句提供消息文本。

```
DECLARE @StringVariable NVARCHAR(50);
SET @StringVariable = N'<<%7.3s>>';

RAISERROR (@StringVariable, -- 信息.
          10, -- 错误级别,
          1, -- 状态,
          N'abcde'); -- 提供字符串的参数.
-- 返回的信息为: <<   abc>>.
GO
```

4.2. 游标

4.2.1. 游标的基本概念与操作

下面我们通过图 4-1 中的程序介绍游标的基本概念与操作。

```
USE AdventureWorks
GO

-- (1) 声明游标 contact_cursor
DECLARE contact_cursor CURSOR FOR
SELECT LastName FROM Person.Contact
WHERE LastName LIKE 'B%'
ORDER BY LastName

-- (2) 打开游标
OPEN contact_cursor

-- (3) 执行第一次提取.
FETCH NEXT FROM contact_cursor

-- (4) 检查 @@FETCH_STATUS 以查看是否还有更多的行可以提取.
-- 如果有则提取下一行
WHILE @@FETCH_STATUS = 0
BEGIN
    -- This is executed as long as the previous fetch succeeds.
    FETCH NEXT FROM contact_cursor
END

-- (5) 关闭并删除游标
CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

图 4-1 游标的基本操作

该示例是一个简单的游标操作示例，在示例中声明了一个游标 `contact_cusor`，然后打开游标并从其中提取记录，最后关闭并删除游标。程序中并没有对从游标中提取的记录进行处理，其执行结果如图 4-2 所示（只显示部分结果）。

LastName

Bacalzo
(1 行受影响)
LastName

Bacon
(1 行受影响)
LastName

Bacon
(1 行受影响)

图 4-2 游标基本操作的执行结果

游标的基本操作有四个,包括:声明、打开、提取与关闭(具体的语法请参考 SQL Server 联机文档中的 Transact-SQL 语言参考)。其中,

每个游标的第一个操作总是声明。声明游标就像其它编程语言中声明变量一样,声明的变量必须有数据类型,游标的数据类型就是游标,但不同的是声明游标还需要有一个 **SELECT** 查询语句,查询语句中不能够使用变量。

游标的第二个操作是打开(程序中的第(2)步)。打开游标实际上是在服务器上执行游标声明中的 **SELECT** 语句,游标打开后游标的值就是查询语句的结果。实际上,游标是 T-SQL 中的一种特殊的数据类型,这种数据类型可以存储一个行集或记录集。在执行打开操作后,游标的行集会保存在临时数据库中,就像一个临时表一样,但只能通过游标操作该记录集。

游标的打开操作就像是变量的初始化。当然,变量在初始化后还可以改变其值,但没有相应的操作能够直接修改游标的行集。但这并不是说游标的内容在打开后就不发生变化了,如果声明的游标不是静态游标,则其内容会随着 **SELECT** 语句中表的记录的变化而变化。

游标打开后的操作通常是 **FETCH** 提取操作(程序中的第(3)、(4)步)。提取操作每次从游标中提取一行。提取出来的一行可以放到变量中以进行相应的处理,但该程序中并没有对提取的行进行任何操作。

游标有一个当前行,它有一个指针指向当前行,该指针就像一个“游标”一样,这

也是术语“游标”的由来。打开后当前行是第一行，程序中第一个 **FETCH NEXT** 语句即是提取当前行并把指针移到下一行。可以有其它的方式提取。

提取操作并不一定成功，如果记录集中没有记录或者游标当前行是最后一行的后面，则提取操作不成功。可以使用系统变量 `@@FETCH_STATUS` 检查前面的 **FETCH** 操作是否成功，如果成功则该系统变量的值为 0。程序中使用一个 **WHILE** 循环来提取其它所有的行，每次提取后都通过判断系统变量 `@@FETCH_STATUS` 的值是否为 0 以确定是否进行下一个行的提取。通常，在程序中我们都会采用这种结构来提取游标中所有的行。

游标使用完毕需要关闭（程序中的 **CLOSE** `contact_cursor` 语句）。游标一旦关闭便不能对其进行提取操作，必须再次打开后才可以进行提取。游标的关闭就像 C 或 C++ 语言中对一个分配内存的指针进行 **delete** 操作一样。

游标关闭后并不是说游标不存在了，游标在程序中仍然存在，不能声明一个相同名称的游标。只有在执行游标的删除操作（程序中的 **DEALLOCATE** `contact_cursor` 语句）才删除了游标，与该游标有关的资源也同时被释放。应注意，不能对一个打开的游标进行删除操作。

另外，许多程序中没有删除操作甚至没有关闭操作，但程序也不会出现错误。这与 C 或 C++ 程序中给指针分配的内存但没有释放是一样的道理，如果程序执行结束，服务器会自动释放相应的资源。

4.2.2. 处理游标中的行

通常，从游标中提取的行需要作进一步的处理，如使用其中的值进行判断、计算或更新数据库。因此，简单的 **FETCH** 操作并不能完成这一功能。下面的示例演示了上一节中的游标示例，所不同的是每次提取的行的值被存储到两个变量 `@LastName` 和 `@FirstName`，利用这两个变量生成一个包含姓名的字符串。

程序中第（1）步是声明两个变量@LastName 和@FirstName。程序中声明的变量必须以@符号开头。这两个变量用于保存从游标中提取的行的列值。

第（2）步是声明一个游标 contact_cursor，这与上一节所介绍的相同。

第（3）步是打开游标。

```
USE AdventureWorks
GO

-- （1）声明变量以存储由 FETCH 提取的值.
DECLARE @LastName varchar(50), @FirstName varchar(50)

-- （2）声明游标 contact_cursor
DECLARE contact_cursor CURSOR FOR
    SELECT LastName, FirstName FROM Person.Contact
        WHERE LastName LIKE 'B%'
        ORDER BY LastName, FirstName

-- （3）打开游标
OPEN contact_cursor

-- （4）执行第一次提取并将值存储在变量中.
-- 注：变量的顺序必须与游标 SELECT 语句中列的顺序相同
FETCH NEXT FROM contact_cursor
INTO @LastName, @FirstName

-- （5）检查 @@FETCH_STATUS 以查看是否有更多的行可以提取.
WHILE @@FETCH_STATUS = 0
BEGIN

    -- （6）连接并显示变量中当前值.
    PRINT 'Contact Name: ' + @FirstName + ' ' + @LastName

    -- （7）当前一个提取成功时执行下一个提取.
    FETCH NEXT FROM contact_cursor
    INTO @LastName, @FirstName
END

-- （8）关闭并删除游标
CLOSE contact_cursor
DEALLOCATE contact_cursor
```

图 4-3 提取游标的行并进行处理示例

第（4）步是第一次提取游标，语句中的 INTO @LastName, @FirstName 子句表明将提取的行的列值分别存储在变量@LastName 与@FirstName 中。需要注意的是，INTO 子句中的变量的个数与数据类型必须与游标的 SELECT 语句的输出结果的列数和数据类

型相一致。数据类型可以不完全相同，但必须保证变量的数据类型可以存储结果中相应的列值。


第（5）步检查系统变量 @@FETCH_STATUS 的值以判断前面的提取是否成功，如果成功则进行第（6）步。

第（6）步使用 PRINT 函数输出连接的字符串。这里的处理虽然简单，但与复杂处理的基本原理是一样的。

第（7）步是进行下一个提取。

第（8）步是关闭并删除游标。

执行结果如图 4-4 所示（由于结果中的行太多，只显示部分结果）：



```
Contact Name: Phillip Bacalzo
Contact Name: Dan Bacon
Contact Name: Dan Bacon
Contact Name: Abigail Bailey
Contact Name: Adrian Bailey
Contact Name: Alex Bailey
Contact Name: Alexa Bailey
Contact Name: Alexandra Bailey
Contact Name: Alexandria Bailey
```

图 4-4 提取游标的行并进行处理示例的结果

4.3. 存储过程

4.3.1. 什么是存储过程

首先，因为存储过程是一个段程序代码，所以称为过程（区别于非过程的标准 SQL），其次，因为它存储在数据库中，所以在过程的前面加上了“存储”称为存储过程。与存储过程不同的是可以将过程放在批处理或应用程序中，每次执行时需要将代码发送到数据库服务器去执行。存储过程的特性大都是由“存储”这一机制产生的。

（1） 存储过程的作用

SQL Server 中的存储过程与其他编程语言中的过程类似，存储过程可以：

- ① 接受输入参数并以输出参数的格式向调用过程或批处理返回多个值。
- ② 包含用于在数据库中执行操作（包括调用其他过程）的编程语句。
- ③ 向调用过程或批处理返回状态值，以指明成功或失败（以及失败的原因）。

可以使用 Transact-SQL EXECUTE 语句执行存储过程。存储过程与函数不同，因为存储过程不返回取代其名称的值，也不能直接在表达式中使用。

（2） 存储过程的优点

在 SQL Server 中使用存储过程而不使用存储在客户端计算机本地的 Transact-SQL 程序的好处包括：

- ① 存储过程已在服务器注册。由于在执行前已经存储在数据库中，因此，使用存储过程会比非存储过程获得更好的性能。
- ② 存储过程具有安全特性（例如权限）和所有权链接，以及可以附加到它们的证书。存储过程也是一种可授权的数据库对象，可以使用数据库的安全机制限制对存储过程的访问。例如，可以授予某一用户只能执行而不能修改某一存储过程或某一数据库中的所有存储过程。
- ③ 用户可以被授予权限来执行存储过程而不必直接对存储过程中引用的对象具有权限。例如，只能通过执行存储过程对一个表完成操作而对表没有直接的操作权限。同时，由于存储过程的执行逻辑是事先确定的，因此，执行哪些操作是可以预见的。
- ④ 存储过程可以加强应用程序的安全性。参数化存储过程有助于保护应用程序不受 SQL Injection 攻击（SQL 注入是一种攻击方式，在这种攻击方式中，恶意代码被插入到字符串中，然后将该字符串传递到 SQL Server 的实例以进行分析和执行。）
- ⑤ 存储过程允许模块化程序设计。存储过程一旦创建，以后即可在程序中调用任意多次。这可以改进应用程序的可维护性，并允许应用程序统一访问数据库。这一特性与其它高级程序语言中的子程序或函数相似。
- ⑥ 存储过程是命名代码，允许延迟绑定。这提供了一个用于简单代码演变的间接级别。
- ⑦ 存储过程可以减少网络通信流量。一个需要数百行 Transact-SQL 代码的操作可以通过一条执行过程代码的语句来执行，而不需要在网络中发送数百行代码。

建议在应用程序中尽量使用存储过程完成对数据库的操作。例外的情况是应用程序需要支持多种数据库，使用存储过程可能会给应用程序的兼容性带来更多的麻烦。

4.3.2. 存储过程的类型

Transact-SQL 存储过程的类型有用户定义的存储过程、扩展存储过程和系统存储过程三种。Microsoft 建议使用 CLR 来而不是扩展存储过程，因此不再介绍扩展存储过程。

1 用户定义的存储过程

存储过程是指封装了可重用代码的模块或例程。存储过程可以接受输入参数、向客户端返回行集或标量结果和消息、调用数据定义语言 (DDL) 和数据操作语言 (DML) 语句，然后返回输出参数。在 SQL Server 中，存储过程有两种类型：Transact-SQL 或 CLR。

（1） Transact-SQL

Transact-SQL 存储过程是指保存的 Transact-SQL 语句集合，可以接受和返回用户提供

的参数。例如，存储过程中可能包含根据客户端应用程序提供的信息在一个或多个表中插入新行所需的语句。

(2) CLR

CLR 存储过程是指对 Microsoft .NET Framework 公共语言运行时 CLR(Common Language Runtime) 方法的引用，可以接受和返回用户提供的参数。它们在 .NET Framework 程序集中是作为类的公共静态方法实现的。CLR 存储过程是一种使用 .NET 语言如 Visual Basic .NET 和 C# 编写的存储过程，与 Transact-SQL 存储过程不同的只是所使用的语言不同，本质上没有区别。

2 系统存储过程

SQL Server 中的许多管理活动都是通过一种特殊的存储过程执行的，这种存储过程被称为系统存储过程。例如，`sys.sp_changedbowner` 就是一个系统存储过程。从物理意义上讲，系统存储过程存储在 Resource 数据库(Resource 数据库是只读数据库，它包含了 SQL Server 2005 中的所有系统对象，SQL Server 2000 没有 Resource 数据库。)中，并且带有 `sp_` 前缀。从逻辑意义上讲，系统存储过程出现在每个系统定义数据库和用户定义数据库的 `sys` 构架中。例如，下例执行系统存储过程 `sp_tables` 显示 AdventureWorks 数据库中的所有表对象的列表。

```
USE AdventureWorks
GO

exec sp_tables
```

在 SQL Server 中，可将 GRANT（授权）、DENY（禁止）和 REVOKE（撤消权限）权限应用于系统存储过程以控制用户对系统存储过程的访问。

SQL Server 支持在 SQL Server 和外部程序之间提供一个接口以实现各种维护活动的系统存储过程。这些扩展存储程序使用 `xp_` 前缀。

4.4. 触发器

触发器是数据库服务器中发生事件时自动执行的特种存储过程。SQL Server 支持 DML、DDL 与登录触发器，我们这里只演示 DML 触发器。如果用户要通过数据操作语言 (DML) 事件编辑数据，则执行 DML 触发器。DML 事件是针对表或视图的 INSERT、UPDATE 或 DELETE 语句。

DML 触发器在以下方面非常有用：

- (1) DML 触发器可通过数据库中的相关表实现级联更改。不过，通过级联引用完整性约束可以更有效地进行这些更改。
- (2) DML 触发器可以防止恶意或错误的 INSERT、UPDATE 以及 DELETE 操作，并强制执行比 CHECK 约束定义的限制更为复杂的其他限制。

- (3) 与 CHECK 约束不同, DML 触发器可以引用其他表中的列。例如, 触发器可以使用另一个表中的 SELECT 比较插入或更新的数据, 以及执行其他操作, 如修改数据或显示用户定义错误信息。
- (4) DML 触发器可以评估数据修改前后表的状态, 并根据该差异采取措施。
- (5) 一个表中的多个同类 DML 触发器 (INSERT、UPDATE 或 DELETE) 允许采取多个不同的操作来响应同一个修改语句。

4.4.1. DML 触发器的类型

(1) AFTER 触发器

在执行了 INSERT、UPDATE 或 DELETE 语句操作之后执行 AFTER 触发器。指定 AFTER 与指定 FOR 相同, 它是 Microsoft SQL Server 早期版本中唯一可用的选项。AFTER 触发器只能在表上指定。

(2) INSTEAD OF 触发器

执行 INSTEAD OF 触发器代替通常的触发动作。还可为带有一个或多个基表的视图定义 INSTEAD OF 触发器, 而这些触发器能够扩展视图可支持的更新类型。

(3) CLR 触发器

CLR 触发器可以是 AFTER 触发器或 INSTEAD OF 触发器。CLR 触发器还可以是 DDL 触发器。CLR 触发器将执行在托管代码 (在 .NET Framework 中创建并在 SQL Server 中上载的程序集的成员) 中编写的方法, 而不用执行 Transact-SQL 存储过程。有关详细信息, 请参阅编程 CLR 触发器。

4.4.2. AFTER 触发器与 INSTEAD OF 触发器的比较

- (1) 执行 INSTEAD OF 触发器代替通常的触发操作。还可以对带有一个或多个基表的视图定义 INSTEAD OF 触发器, 这些触发器可以扩展视图可支持的更新类型。
- (2) 在执行 INSERT、UPDATE 或 DELETE 语句操作之后执行 AFTER 触发器。指定 AFTER 与指定 FOR 相同。AFTER 触发器只能在表上指定。

下表对 AFTER 触发器和 INSTEAD OF 触发器的功能进行了比较。

需要注意的是 AFTER 触发器晚于约束执行, 而 INSTEAD OF 触发器是先于约束执行。比如当插入一个记录到某一子表中时, 对于 AFTER 触发器, 会先检查插入的记录是否违反了引用约束条件, 即插入的外码列的值是否是父表中主码列的值, 而对于 INSTEAD OF 触发器则会先执行触发器。

表 4-3 AFTER 与 INSTEAD OF 触发器的比较

函数	AFTER 触发器	INSTEAD OF 触发器
适用范围	表	表和视图
每个表或视图包含触发器的数量	每个触发操作 (UPDATE、DELETE 和 INSERT) 包含多个	每个触发操作 (UPDATE、DELETE 和 INSERT) 包含一个

	触发器	触发器
级联引用	无任何限制条件	不允许在作为级联引用完整性约束目标的表上使用 INSTEAD OF UPDATE 和 DELETE 触发器。
执行	晚于： <ul style="list-style-type: none"> • 约束处理 • 声明性引用操作 • 创建插入的和删除的表 • 触发操作 	早于： <ul style="list-style-type: none"> • 约束处理 替代： <ul style="list-style-type: none"> • 触发操作 晚于： <ul style="list-style-type: none"> • 创建插入的和删除的表

4.4.3. AFTER 触发器示例

创建触发器使用 **CREATE TRIGGER** 命令。图 4-5 中的触发器示例创建了一个名为 **LowCredit** 的触发器，该触发器的触发事件为表 **Purchasing.PurchaseOrderHeader** 的 **INSERT** 操作，即每当插入一条记录到表中后就会自动执行触发器的程序。

```

USE AdventureWorks
GO
-- 如果触发器存在则删除它
IF OBJECT_ID ('Purchasing.LowCredit','TR') IS NOT NULL
    DROP TRIGGER Purchasing.LowCredit
GO

-- 在表 Purchasing.PurchaseOrderHeader 的 insert 操作
-- 上创建触发器 LowCredit
CREATE TRIGGER LowCredit ON Purchasing.PurchaseOrderHeader
AFTER INSERT
AS
DECLARE @creditrating tinyint,
        @vendorid int
SELECT @creditrating = v.CreditRating, @vendorid = p.VendorID
FROM Purchasing.PurchaseOrderHeader p INNER JOIN inserted i
    ON p.PurchaseOrderID = i.PurchaseOrderID
    JOIN Purchasing.Vendor v on v.VendorID = i.VendorID
IF @creditrating = 5
BEGIN
    RAISEERROR ('订单中厂商的信用等级太低不能接收新订单', 16, 1)
    ROLLBACK TRANSACTION
END

```

图 4-5 触发器示例

示例的基本功能是当插入订单记录到订单基本信息表 **Purchasing.PurchaseOrderHeader** 中时，根据插入订单的厂商编号从厂商基本信息表 **Purchasing.Vendor** 中查询厂商的信用等级（**Vender.CreditRating** 列）并将得到的信用等级的值存储到变量 **@creditrating** 中。如果厂商的信用等级等于 5，则程序使用系统函数 **RAISEERROR** 抛出一个错误信息，错误的严重级别为 16，**RAISEERROR** 函数中的第 3 个参数 1 表示状态，可以使用它定位程序出错的位置。如果信用等级太低，程序使用 **ROLLBACK TRANSACTION** 语句撤消当前事务。

测试触发器前先查询厂商基本信息表，使用如下的命令：

```
select VendorID,CreditRating from Purchasing.Vendor
Order by CreditRating desc
```

图 4-6 查询厂商基本信息表的命令

查询结果如下（只显示一部分）

图 4-7 厂商基本信息表的查询结果

VendorID	CreditRating
22	5
63	5
84	4
36	4
37	3
10	3
75	3
71	3
8	3
30	3
31	3
33	2
28	2
77	2
86	2
38	2
93	2

在下面的测试中我们将使用厂商编号为 22（信用等级为 5）、厂商编号为 93（信用等级为 2）的两个记录。

使用下面的命令检查订单基本信息表中的订单个数。

```
select count(*) from Purchasing.PurchaseOrderHeader
```

图 4-8 查询订单数量的命令

查询的结果如下：

```
-----
4000

(1 行受影响)
```

图 4-9 执行插入新订单前的订单数量

我们使用下面的命令插入一个新订单，其中，订单的编号由系统自动生成，员工的

```
insert into Purchasing.PurchaseOrderHeader
(EmployeeID, VendorID, OrderDate, ShipMethodID)
values (198, 22, getdate(), 5);
```

编号为 198，厂商的编号为 22，订单的日期使用系统当前的日期（由系统函数 `get_date` 返回），订单的发货类型编号为 5。由于这里只是测试触发器，所以我们并不关心插入数据的合理性，只要符合订单基本信息表的约束条件即可。

图 4-10 插入厂商编号为 22 的新订单

在成功创建触发器后，执行上面的插入操作命令，显示如下的结果：

```
消息50000, 级别16, 状态1, 过程LowCredit, 第15 行
订单中厂商的信用等级太低不能接收新订单
消息3609, 级别16, 状态1, 第1 行
事务在触发器中结束。批处理已中止。
```

图 4-11 插入厂商编号为 22 的新订单的执行结果

为了验证新订单是否插入到表中，我们仍然使用图 4-8 中的命令查询表中订单的数量，验证的结果与图 4-9 中的所显示的数量相同，说明新订单没有插入到订单基本信息表中。

接下来我们使用下面的命令插入一个另一个新订单，该订单中厂商的编号为 93，从图 4-7 所示的厂商基本信息中可知该厂商的信用等级为 2，符合触发器规定的厂商信用等级要求。

```
insert into Purchasing.PurchaseOrderHeader
(EmployeeID, VendorID, OrderDate, ShipMethodID)
values (198, 93, getdate(), 5);
```

图 4-12 插入厂商编号为 93 的新订单

执行结果如下：

```
(1 行受影响)
```

图 4-13 插入厂商编号为 93 的新订单的结果

说明新订单插入成功。我们仍然可以使用图 4-8 中所示的命令查询表中订单的数量，查询的结果如所示：

```
-----
4001
(1 行受影响)
```

图 4-14 插入厂商编号为 93 的新订单后的订单数量

4.4.4. 允许与禁止触发器

可以在需要时使用下面的命令禁止某一个触发器，

```
ALTER TABLE Purchasing.PurchaseOrderHeader
DISABLE TRIGGER LowCredit
```

图 4-15 禁止表 Purchasing.PurchaseOrderHeader 的触发器 LowCredit

执行该命令则禁止表前面创建的订单基本信息表的触发器 `LowCredit`，我们仍然使用图 4-10 中的命令插入新的订单，执行的结果显示新的订单插入成功。此时，说明并没有

```
ALTER TABLE Purchasing.PurchaseOrderHeader
ENABLE TRIGGER LowCredit
```


执行触发器 LowCredit。

可以使用下面的命令允许触发器：

图 4-16 允许表 Purchasing.PurchaseOrderHeader 的触发器 LowCredit

4.4.5. 删除的表 deleted 和插入的表 inserted

DML 触发器语句使用两种特殊的表：删除的表 `deleted` 和插入的表 `inserted`。SQL Server 会自动创建和管理这两种表。可以使用这两种驻留内存的临时表来测试特定数据修改的影响以及设置 DML 触发器操作条件。但不能直接修改表中的数据或对表执行数据定义语言 (DDL) 操作。

在 DML 触发器中，`inserted` 和 `deleted` 表主要用于执行以下操作：

- (1) 扩展表之间的引用完整性。
- (2) 在以视图为基础的基表中插入或更新数据。
- (3) 检查错误并采取相应的措施。
- (4) 找出数据修改前后表的状态差异并基于该差异采取相应的措施。
- (5) 删除的表用于存储 `DELETE` 和 `UPDATE` 语句所影响的行的副本。在执行 `DELETE` 或 `UPDATE` 语句的过程中，行从触发器表中删除，并传输到删除的表中。删除的表和触发器表通常没有相同的行。
- (6) 插入的表用于存储 `INSERT` 和 `UPDATE` 语句所影响的行的副本。在执行插入或更新事务过程中，新行会同时添加到 `inserted` 表和触发器表中。插入的表中的行是触发器表中的新行的副本。
- (7) 更新事务类似于在删除操作之后执行插入操作；首先，旧行被复制到删除的表中，然后，新行被复制到触发器表和插入的表中。
- (8) 在设置触发器条件时，应使用激发触发器的操作相应的插入的和删除的表。尽管在测试 `INSERT` 时引用删除的表或在测试 `DELETE` 时引用插入的表不会导致任何错误，但在这些情况下，这些触发器测试表将不包含任何行。

图 4-5 触发器示例中的插入记录即存储在插入表 `inserted` 中，查询时首先从 `inserted` 表中得到插入记录（插入的新订单）的厂商的编号 `inserted.VendorID`，然后使用插入记录中的厂商编号到厂商基本信息表 `Purchasing.Vendor` 中查询得到该厂商的信用等级 `Vendor.CreditRating`。

4.4.6. INSTEAD OF 触发器示例一

Transact-SQL 语句创建个人信息表 `Person` 和员工信息表 `EmployeeTable`、一个视图 `Employee`、一个记录错误表 `PersonDuplicates` 和视图上的 `INSTEAD OF` 触发器 `IO_Trig_INS_Employee`。

具体的步骤如下：

- (1) 首先创建一个数据库 `Student` 用于存储所创建的表、视图和触发器。设置数据库

```
USE Student
GO
```

Student 为当前数据库:

图 4-17 设置 Student 为当前数据库

(2) 使用下面的命令创建个人信息表 Person:

```
--创建个人信息表
CREATE TABLE Person
(
    SSN          char(11) PRIMARY KEY,
    Name         nvarchar(10),
    Address      nvarchar(32),
    Birthdate    datetime
)
```

图 4-18 创建个人信息表 Person

表 Person 中的列 SSN 是个人的社会安全号码,它是该表的主码。

(3) 创建员工信息表 EmployeeTable, 命令如下:

```
--创建员工信息表
CREATE TABLE EmployeeTable
(
    EmployeeID    int PRIMARY KEY,
    SSN           char(11) UNIQUE,
    Department    nvarchar(10),
    Salary        money,
    CONSTRAINT FKEmpPer FOREIGN KEY (SSN)
    REFERENCES Person (SSN)
)
```

图 4-19 创建员工信息表 EmployeeTable

员工信息表 EmployeeTable 是个人信息表 Person 的子表,它的 SSN 列引用了 Person 表的 SSN 列。

(4) 创建视图 Employee, 命令如下:

```
--下面的视图使用某人的两个表中的所有相关数据建立报表:

CREATE VIEW Employee AS
SELECT P.SSN as SSN, Name, Address,
       Birthdate, EmployeeID, Department, Salary
FROM Person P, EmployeeTable E
WHERE P.SSN = E.SSN
```

图 4-20 创建视图 Employee

(5) 创建错误记录表 PersonDuplicates, 命令如下:

```
--可记录对插入具有重复的社会安全号的行的尝试。
--PersonDuplicates 表记录插入的值、尝试插入操作的用户的用户名
--和插入的时间:

CREATE TABLE PersonDuplicates
(
    SSN          char(11),
    Name         nvarchar(10),
    Address      nvarchar(32),
    Birthdate    datetime,
    InsertSNAME  nchar(12),
    WhenInserted datetime
)
```

图 4-21 创建错误记录表 PersonDuplicates

该表中的 SSN 列记录社会安全号码。

(6) 创建触发器，命令如下：

```
CREATE TRIGGER IO_Trig_INS_Employee ON Employee
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON
-- 检查重复输入的个人信息. 如果没有重复的, 则插入.
IF (NOT EXISTS (SELECT P.SSN
                FROM Person P, inserted I
                WHERE P.SSN = I.SSN))
    INSERT INTO Person
        SELECT SSN, Name, Address, Birthdate
        FROM inserted
--如果有重复的,
ELSE
-- 将重复的个人信息记录插入到PersonDuplicates 表.
    INSERT INTO PersonDuplicates
        SELECT SSN, Name, Address, Birthdate, SUSER_SNAME(), GETDATE()
        FROM inserted
-- 检查重复的员工信息. 如果没有重复的则插入.
IF (NOT EXISTS (SELECT E.SSN
                FROM EmployeeTable E, inserted
                WHERE E.SSN = inserted.SSN))
    INSERT INTO EmployeeTable
        SELECT EmployeeID, SSN, Department, Salary
        FROM inserted
ELSE
--如果有重复的员工信息, 修改员工信息表的已经存在的
--be a duplicate key violation error.
    UPDATE EmployeeTable
        SET EmployeeID = I.EmployeeID,
            Department = I.Department,
            Salary = I.Salary
        FROM EmployeeTable E, inserted I
        WHERE E.SSN = I.SSN
END
```

图 4-22 创建 INSTEAD OF 触发器 IO_Trig_INS_Employee

(7) 插入正确记录并查询结果。下面插入的 2 条记录均符合要求，

```
INSERT INTO Employee VALUES
('1011082007', 'Peter', 'Beijing,Haidian', '1980-12-10', 1, '技术部', 120000.00)
INSERT INTO Employee VALUES
('1011082008', 'Peter', 'Beijing,Haidian', '1979-1-10', 2, '技术部', 160000.00)
```

图 4-23 插入正确的记录

查询表 Person，结果如下，

SSN	Name	Address	Birthdate
1011082007	Peter	Beijing,Haidian	1980-12-10 00:00:00.000
1011082008	Peter	Beijing,Haidian	1979-01-10 00:00:00.000

(2 行受影响)

图 4-24 插入正确记录后 Person 表的记录

查询表 EmployeeTable，结果如下，

EmployeeID	SSN	Department	Salary
1	1011082007	技术部	120000.00
2	1011082008	技术部	160000.00

(2 行受影响)

图 4-25 插入正确记录后的 EmployeeTable 表的记录

查询表 PersonDuplicates，结果如下，

SSN	Name	InsertSNAME	WhenInserted

(0 行受影响)

图 4-26 插入正确记录后表 PersonDuplicates

(8) 插入错误记录。使用下面的命令插入一条记录

```
INSERT INTO Employee VALUES
('1011082007', 'Peter', 'Beijing,Haidian', '1980-12-10', 1, '技术部', 120000.00)
```

图 4-27 插入有相同 SSN 的记录

查询表 Person 与 EmployeeTable 结果与执行插入前相同，说明记录没有插入到这两个表中，但 PersonDuplicates 表的内容如下：

SSN	Name	InsertSNAME	WhenInserted
1011082007	Peter	GUO\david	2007-10-28 10:04:26.793
(1 行受影响)			

图 4-28 插入相同 SSN 后的 PersonDuplicates 表

该触发器还有需要验证的内容，请你自行验证。

4.4.7. INSTEAD OF 触发器示例二

下面是示例数据库 AdventureWorks 中的 Purchasing.Vendor 表中的一个 INSTEAD OF 触发器示例，它禁止删除 Vender 表的记录。

```
CREATE TRIGGER [Purchasing].[dVendor] ON [Purchasing].[Vendor]
INSTEAD OF DELETE NOT FOR REPLICATION AS
BEGIN
    DECLARE @Count int;
    SET @Count = @@ROWCOUNT;
    IF @Count = 0
        RETURN;

    SET NOCOUNT ON;

    BEGIN TRY
        DECLARE @DeleteCount int;

        SELECT @DeleteCount = COUNT(*) FROM deleted;
        IF @DeleteCount > 0
            BEGIN
                RAISERROR
                (N'Vendors cannot be deleted. They can only be marked as
not active.', -- Message
                11, -- Severity.
                1); -- State.

                -- Rollback any active or uncommittable transactions
                IF @@TRANCOUNT > 0
                BEGIN
                    ROLLBACK TRANSACTION;
                END
            END;
        END TRY
        BEGIN CATCH
            EXECUTE [dbo].[uspPrintError];

            -- Rollback any active or uncommittable transactions before
            -- inserting information in the ErrorLog
            IF @@TRANCOUNT > 0
            BEGIN
                ROLLBACK TRANSACTION;
            END

            EXECUTE [dbo].[uspLogError];
        END CATCH;
    END;
```

图 4-29 禁止删除触发器示例

下面是触发器有关的说明。创建触发器的命令选项如 NOT FOR REPLICATION 可以在联机文档的 CREATE TRIGGER 命令中找到，@@系统函数可以搜索联机文档的“系统函数”。

◆ NOT FOR REPLICATION

指示当复制代理修改涉及到触发器的表时，不应执行触发器。

◆ @@ROWCOUNT

系统函数@@ROWCOUNT 返回受上一语句影响的行数。

◆ @@TRANCOUNT

返回在当前连接上执行的 BEGIN TRANSACTION 语句的数目。

◆ SET NOCOUNT ON

阻止在结果集中返回可显示受 Transact-SQL 语句或存储过程影响的行计数的消息。如果选项则 ON 则不显示类似“5 行受影响”的消息，只显示“命令已成功完成”，关闭时（即设置为 ON，默认为 OFF）可提高性能。

◆ dbo.uspPrintError 存储过程

该存储过程打印错误信息，你可以在 AdventureWorks 示例数据库的“可编程性/存储过程”节点找到其源代码，如下：

```
-- uspPrintError prints error information about the error that caused
-- execution to jump to the CATCH block of a TRY...CATCH construct.
-- Should be executed from within the scope of a CATCH block otherwise
-- it will return without printing any error information.
CREATE PROCEDURE [dbo].[uspPrintError]
AS
BEGIN
    SET NOCOUNT ON;

    -- Print error information.
    PRINT 'Error ' + CONVERT(varchar(50), ERROR_NUMBER()) +
        ', Severity ' + CONVERT(varchar(5), ERROR_SEVERITY()) +
        ', State ' + CONVERT(varchar(5), ERROR_STATE()) +
        ', Procedure ' + ISNULL(ERROR_PROCEDURE(), '-') +
        ', Line ' + CONVERT(varchar(5), ERROR_LINE());
    PRINT ERROR_MESSAGE();
END;
```

图 4-30 dbo.uspPrintError 存储过程

◆ dbo.uspLogError 存储过程

dbo.uspLogError 存储过程将错误信息写到 ErrorLog 表中，你可以在示例数据库的“可编程性/存储过程”节点找到其源代码。

◆ 测试

测试的命令：

```
USE AdventureWorks
GO
DELETE Purchasing.Vendor WHERE BusinessEntityID<1500;
```

图 4-32 禁止删除触发器测试命令

测试的结果：

Vendors cannot be deleted. They can only be marked as not active.

消息 3609，级别 16，状态 1，第 1 行

事务在触发器中结束。批处理已中止。

图 4-33 禁止删除触发器测试结果

注释：*RAISEERROR* 抛出的错误级别不同会导致不同的测试结果。只有错误级别大于 10 时才会导致执行跳转到 *CATCH* 块。

思考题：*INSTEAD OF* 触发器需要撤消触发事件的操作吗？上面触发器中的撤消事务的操作 *ROLLBACK TRANSACTION* 是撤消 *delete* 操作吗？如果不是，是撤消什么样的操作？举例说明。

4.5. 实验

通过该实验加深存储过程和触发器的基本概念的理解，掌握基本存储过程、触发器的管理和执行，能够使用游标编写较复杂的存储过程，为后面的系统开发打好基础。

4.5.1. 实验一：存储过程创建与使用

创建一个存储过程，执行该存储过程完成如下功能：

查询示例数据库 AdventureWorks 中的订单基本信息表 Sales.SalesOrderHeader、客房基本信息表 Sales.Customer 和订单明细表 Sales.SalesOrderDetail，输出订单编号（SalesOrderID）、订单日期（OrderDate）、发货日期（ShipDate）、客户名称（CustomerID 为客户编号）和订单的总价，并将输出结果写到一个新建的表中。

其中，订单的总价是由订单明细表中的各项产品的单价（UnitPrice）相加得到的，即每个订单的总价=SUM（产品数量（OderQty）*产品单价（UnitPrice））。

存储过程需要创建输出结果的表，如果输出结果表已经存在则需要在创建之前将其删除。

4.5.2. 实验二：触发器的创建与测试

功能：

检查订单明细表 Sales.SalesOrderDetail 中的信息，如果修改记录中的产品单价 UnitPrice 大于产品公开报价（Production.Product.ListPrice），则不能进行修改并抛出错误信息，否则，进行修改并将修改的有关信息写到 Production.ProuctUpdateLog 表中。

要求：

1. 使用 **RAISEERROR** 抛出错误信息。
2. 修改信息记录表 **Production.ProductUpdateLog** 的内容：记录编号、订单编号、订单明细编号、产品编号、产品的公开报价、修改前产品的单价、修改后产品的单价、修改者的登录名。使用存储过程完成该功能，并在存储过程中调用该存储过程。
3. 给出触发器和存储过程的源代码和简要的说明（可以在代码中使用注释进行说明）。
4. 设计触发器测试方案并给出测试的命令和结果，必要时可对测试结果进行分析。

第5章 数据库备份与恢复

5.1. 实验的目的、内容与要求

5.1.1. 实验目的

- (1) 综合运用学习的数据库安全管理、备份与恢复、数据库管理工具的使用的相关知识完成数据库和备份与恢复。
- (2) 通过实验进一步掌握数据库简单恢复模式与完整恢复模式下的数据库备份与恢复方法中的基本概念与操作方法。
- (3) 理解数据库备份的重要性以及不同备份与恢复方法的差异。

SQL Server 数据库的备份可以使用 Tansact-SQL（简称 T-SQL，是 SQL Server 的一种可编程 SQL 语言）也可以使用 SSMS 来完成，这两种方法无论是在简单恢复模式还是在完整恢复模式下的备份都是可以用的。这里，我们主要以 T-SQL 的方法演示备份与恢复的操作，要求在实验中掌握这种备份与恢复方法，而对于使用 SSMS 进行备份与恢复则只需要了解即可。

5.1.2. 实验内容

(1) 实验一 简单恢复模式下数据库的备份与恢复

参考实验指导书中的例子，设计一个简单恢复模式下的数据库备份与恢复的实验方案，方案中要完成一个完整备份和多个差异备份，进行数据库的多次恢复实验，每次将数据库恢复到不同的数据库备份。在备份过程中设计相应的操作以验证多次不同恢复的差异。

(2) 实验二 完整恢复模式下数据库的备份与恢复

参考实验指导书中的示例，设计一个完整恢复模式下数据库备份与恢复的方案，方案要完成完整恢复（恢复到故障点）和时间点恢复（部分恢复）并比较两者的差异。方案中还需要设计相应的操作以便验证完整恢复和时间点恢复的差异。

5.1.3. 实验报告

注意在实验的过程中观察每次实验的结果，包括结果是否正确，备份的内容、处理的数据量、备份的时间，如果是恢复则还需要注意在恢复后的状态变化。这些相应的内容均需要写到实验报告中。

5.2. 备份与恢复概述

所有的恢复模式都允许备份完整或部分的 SQL Server 数据库或数据库的单个文件或文件组。不能创建表级备份。

5.2.1. 备份类型

数据的备份（“数据备份”）的范围可以是完整的数据库、部分数据库或者一组文件或文件组。对于这些范围，SQL Server 均支持完整和差异备份：

（1）完整备份

“完整备份”包括特定数据库（或者一组特定的文件组或文件）中的所有数据，以及可以恢复这些数据的足够的日志。

（2）差异备份

“差异备份”基于数据的最新完整备份，完整备份称为差异的“基准”或者差异基准。差异备份仅包括自建立差异基准后发生更改的数据。通常，建立基准备份之后很短时间执行的差异备份比完整备份更小，创建速度也更快。因此，使用差异备份可以加快进行频繁备份的速度，从而降低数据丢失的风险。通常，一个差异基准会由若干个相继的差异备份使用。还原时，首先还原完整备份，然后再还原最新的差异备份。

经过一段时间后，随着数据库的更新，包含在差异备份中的数据量会增加。这使得创建和还原备份的速度变慢。因此，必须重新创建一个完整备份，为另一个系列的差异备份提供新的差异基准。

（3）日志备份

第一次数据备份之后，在完整恢复模式或大容量日志恢复模式下，需要定期进行“事务日志备份”（或“日志备份”）。每个日志备份都包括创建备份时处于活动状态的部分事务日志，以及先前日志备份中未备份的所有日志记录。

5.2.2. 恢复模式

SQL Server 有简单恢复模式与完整恢复模式，两种不同的恢复模式下备份与恢复的方法都不相同，在进行备份前首先要确定是使用哪一种恢复模式。简单恢复模式备份与恢复都比较简单，但有可能会丢失最近对数据库所做的修改，而完整恢复模式相对复杂，丢失数据的风险更小，且可以恢复到某一时间点。

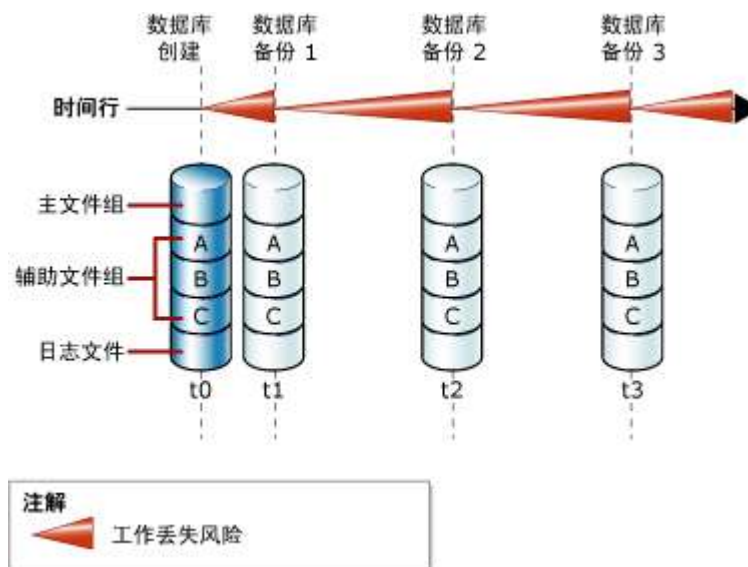
（1）简单恢复模式

简单恢复模式提供简单的备份与还原形式。由于不会备份事务日志，所以备份易于管理。不过，也正是由于这个原因，只能将数据还原到数据最近一次备份的结尾。如果发生故障，则数据库最近一次备份之后所做的修改将全部丢失。

下图显示简单恢复模式下最简单的备份和还原策略。其中有 5 个数据库备份，但只有在时间 t5 进行的备份才需要还原（根据需要也可以还原 t5 以前的备份，但只能还原一个备份）。还原这个备份会将数据库恢复到它 t5 这个时间点，所有后面的修改（以 t6 方块表示）都会丢失。

图 5-1 简单恢复模式下的备份策略

在简单恢复模式下，工作损失风险会随时间经过而增加，直到进行下一个完整备份或差异备份为止。可以通过提高备份的频率以减少丢失数据的风险，但过高的备份频率会影响应用的性能，同时也会使备份变得难以管理。



影响应用的性能，同时也会使备份变得难以管理。

下图显示只使用数据库完整备份的备份计划的工作损失风险，这个策略只适合可频繁备份的小型数据库。

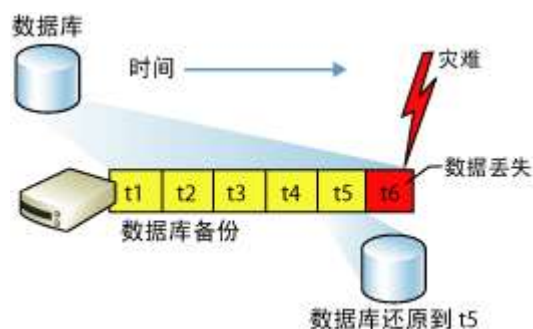


图 5-2 仅使用完整备份的备份策略

图 5-3 中使用数据库完整备份与差异备份，在时间 t1 完成一个数据库完整备份，之后在 t2、t3 和 t4 分别完成 3 个差异备份。在 t1 的差异备份比较小，但在 t4 的差异备份已经与完整备份相差无几，因此在 t5 时又开始一个新的完整备份。在这个备份策略下，如果在 t1 到 t5 之间发生故障，比如 t4，则先恢复 t1 的完整备份，然后恢复 t3 的完整备份。

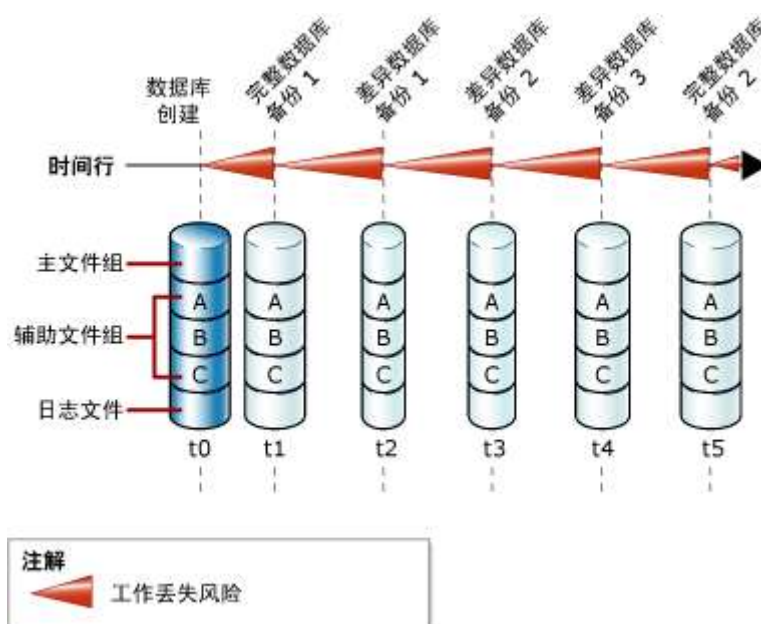


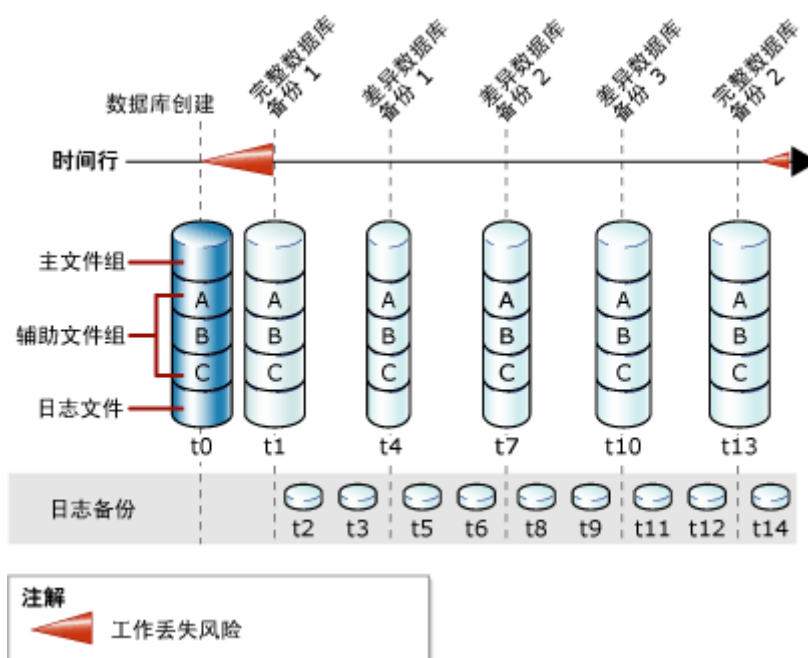
图 5-3 使用完整备份与差异备份的备份策略

(2) 完整恢复模式

完整恢复模式使用日志备份在最大范围内防止出现故障时丢失数据，这种模式需要备份和还原事务日志（“日志备份”）。使用日志备份的优点是允许将数据库还原到日志备份内包含的任何时点（“时间点恢复”）。假定可以在发生严重故障后备份活动日志，则可将数据库一直还原到没有发生数据丢失的故障点处。使用日志备份的缺点是它们会增加还原时间和复杂性。但在大多数的应用环境中还是使用日志备份。

下图显示了在完整恢复模式下的最简单的备份策略。在此图中，已完成了数据库备份 Db_1 以及两个例行日志备份 Log_1 和 Log_2。在 Log_2 日志备份后的某个时间，数据库出现数据丢失。在还原这三个备份前，数据库管理员必须备份活动日志（日志尾部）。然后还原 Db_1、Log_1 和 Log_2，而不恢复数据库。接着数据库管理员还原并恢复尾部日志备份（Tail）。这将把数据库恢复到故障点，从而恢复所有数据。

图 5-4 完整恢复模式下数据库和日志备份

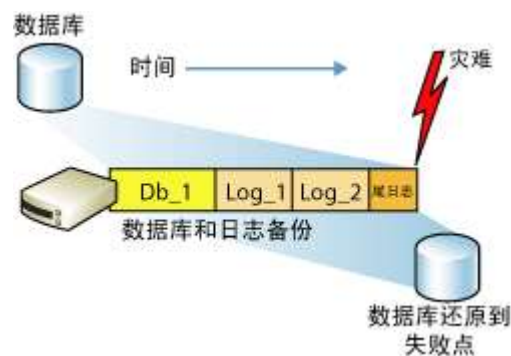


在第一个完整数据库备份完成并且常规日志备份开始之后，潜在的工作丢失风险的存在时间仅为数据库损坏时以及执行最新的常规日志备份时。因此，建议经常执行日志备份，以将工作丢失的风险限定在业务要求所允许的范围内。

出现故障后，可以尝试备份“日志尾部”（尚未备份的日志）。如果尾日志备份成功，则可以通过将数据库还原到故障点来避免任何工作丢失。

可以使用一系列日志备份将数据库前滚到其中一个日志备份中的任意时间点。若要最大程度地降低风险，建议安排例行日志备份。请注意，为了最大程度地缩短还原时间，可以对相同数据进行一系列差异备份以补充每个完整备份。

5-5 显示的备份策略使用差异数据库备份及一系列例行日志备份来补充完整数据库备份。使用事务日志备份可缩短潜在的工作丢失风险的存在时间，使该风险仅在最新日志备份之后存在。在第一个数据库备份完成后，会接着进行三个差异数据库备份。第三个差异备份很大，因此下一次数据库备份时（t13）开始一个新的数据库完整备份。该数据库备份将成为新的差异基准。



5-5 使用完整备份、差异备份和日志备份

在此图中的第一个数据库备份创建之前，数据库存在潜在的工作丢失风险（从时间 t_0 到时间 t_1 ）。该备份建立之后，例行日志备份将工作丢失的风险降为丢失自最近日志备份之后所做的更改（在此图中，最近备份的时间为 t_{14} ）。如果发生故障，则数据库管理员应该立即尝试备份活动日志（日志尾部）。如果此“尾日志备份”成功，则数据库可以还原到故障点。

5.2.3. 日志备份

在完整恢复模式和大容量日志恢复模式下，执行例行事务日志备份（“日志备份”）对于恢复数据十分必要。使用日志备份，可以将数据库恢复到故障点或特定的时间点。建议经常进行日志备份，其频率应足够支持业务需求，尤其是对损坏的日志驱动器可能导致的数据丢失的容忍程度降低时。适当的日志备份频率取决于对工作丢失风险的容忍程度与所能存储、管理和潜在还原的日志备份数量之间的平衡。每 15 到 30 分钟进行一次日志备份可能就已足够。但是如果业务要求将工作丢失的风险最小化，请考虑进行更频繁的日

志备份。频繁的日志备份还有增加日志截断频率的优点，其结果是日志文件更小。

在创建第一个日志备份之前，必须先创建完整备份（如数据库备份）。此后，必须定期备份事务日志。这不仅能最小化工作丢失风险，还有助于事务日志的截断。通常，事务日志在每次常规日志备份之后截断。但是，日志截断也可能会延迟。

（1）日志链

连续的日志备份序列称为“日志链”。日志链从数据库的完整备份开始。通常，仅当第一次备份数据库时，或者将恢复模式从简单恢复模式切换到完整恢复模式之后，才会开始一个新的日志链。

若要将数据库还原到故障点，必须保证日志链是完整的。也就是说，事务日志备份的连续序列必须能够延续到故障点。对于数据库备份，日志备份序列必须从数据库备份的结尾处开始延续。

（2）使用日志备份恢复数据库

还原日志备份将前滚事务日志中记录的更改，使数据库恢复到开始执行日志备份操作时的状态。还原数据库时，必须还原在所还原完整数据库备份之后创建的日志备份。通常情况下，在还原最新数据或差异备份后，必须还原一系列日志备份直到到达恢复点。然后恢复数据库。这将回滚所有在恢复开始时未完成的事务并使数据库在线。恢复数据库后，不得再还原任何备份。

5.3. 简单恢复模式下的备份与恢复

图 5-6 中给出了一个简单恢复模式下备份与恢复数据库 AdventureWorks 的示例。在示例中，第 1 步使用 `ALTER DATABASE` 命令将该数据库设置为简单恢复模式。虽然数据库的默认恢复模式为简单恢复模式，但建议在所有的情况下仍然使用该命令以确保数据库在备份之前是工作在简单恢复模式下，特别是在使用定制备份在情况下更是如此。

第 2 步使用 `BACKUP DATABASE` 进行数据库的完整备份。备份中的引号每个数据库的备份都必须有至少一个完整备份，在恢复时首先要恢复完整备份。

内容是备份文件集的文件名，文件名中的路径必须是已经存在的路径。一每次备份会产生一个备份集，而一个备份集文件可以保存多个备份集。

备份集在备份集文件中有一个唯一的编号，如果不加指定，文件中的第一个备份集的编号是 1，第 2 个备份集的编号是 2，以此类推。备份集的编号在管理备份时和恢复数据库时使用。第 1 步中的备份集在该文件中的编号为 1。

`WITH FORMAT` 表示格式化备份集文件。如果该文件已经存在，则会清空其中的内容，如果文件不存，则创建一个新的文件。

第 3 步是执行一个差异数据库备份，同样使用命令 **BACKUP DATABASE**。但不同的是在命令中使用 **WITH DIFFERENTIAL** 指定进行的备份是一个差异备份，如果是完整备份则不需要指定。注意该步骤中使用了与第 2 步相同的备份集文件，这样该备份集的编号为 2。如果是使用一个新的文件则备份集的编号仍然是 1。

建议在进行第三步之前对数据库进行一些修改操作，如创建一个表或插入一些记录到已有的表中等。这些操作的目的是用于验证在数据库恢复后是否将相应的修改操作结果也恢复回来了。

差异备份根据需要可以进行多次，在实际应用中往往是进行若干个差异备份，这里为了简单只进行一次差异备份。如果在实验的过程中进行多个差异备份，建议在每次差异备份之前都进行对数据库的修改操作，以便在恢复时验证修改结果的恢复情况。

```
USE master;
--1. 设置数据库为简单恢复模式
ALTER DATABASE AdventureWorks SET RECOVERY SIMPLE;
GO
-- 2.执行数据库的完整备份从份
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FORMAT;
GO
--3. 执行数据库的差异备份
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH DIFFERENTIAL;
GO
--4. 还原数据库完整备份 (自备份集 1)，但不恢复数据库。
RESTORE DATABASE AdventureWorks
    FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FILE=1, NORECOVERY;
--5. 还原数据库差异备份 (自备份集 2)，恢复数据库。
RESTORE DATABASE AdventureWorks
    FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FILE=2, RECOVERY;
GO
```

图 5-6 简单恢复模式的数据库备份与恢复示例

在操作第 4 步之前，应当将数据库删除。在完成第 4 步之后注意观察数据库的状态（应当是处于还原状态）。数据库是不会自动显示，需要刷新操作。

第 4 步是还原数据库的完整备份，使用命令 **RESTORE DATABASE.FROM DISK** 指

定备份集文件，该文件是第 1 和 2 步中备份集的文件。**WITH FILE=1, NORECOVERY** 表示从该备份集文件中的第 1 个备份集（它是一个完整备份）进行恢复（每次恢复的第一个备份必须是完整备份），**NORECOVERY** 表示只进行数据库的还原而不进行恢复。还原的操作只是将数据从备份集中拷贝到数据库的文件中，它不会进行未提交事务的撤消，也不会重做记录的日志文件中已经提交的事务。这时的数据库仍然处于还原状态，要把数据库变成正常状态还需要使用其它的命令（后面的示例有相应的内容）。在还原状态下的数据库可以继续还原后续的差异备份。如果没有 **NORECOVERY** 选项，则数据库在完成还原后会进行数据库的恢复操作，如果在还原与恢复的过程中执行正常，则数据库恢复到正常状态。也可以根据需要决定是在哪一个备份（完整或差异都可以）恢复数据库。

第 5 步是还原数据库的差异备份。**FILE=2** 表示从备份集文件中的第 2 个备份集进行还原。**RECOVERY** 表示还原后执行数据库的恢复，该选项不是必须的，因为默认的方式是 **RECOVERY**。

在完成第 5 步后，注意观察数据库已经变成正常的状态了。此时可以查询数据库的内容以验证差异备份前数据库的修改操作结果是否恢复。

5.4. 完整恢复模式下的备份与恢复

5.4.1. 恢复到故障点的备份与恢复示例

在该示例中，备份由一个数据库的完整备份和两个日志备份组成（**USE master;**

```
--1. 将数据库修改为完整恢复模式.
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
GO

--2. 执行数据库的完整备份.
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FORMAT;
GO

--3. 创建日志文件备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO

--4. 创建尾日志备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH NORECOVERY;
GO
```

图 5-7)。其中，数据库的备份可以由一个完整备份也可以由一个完整备份和若干个差异备份组成，我们把这些备份统称为数据库的备份以区别于日志备份。也就是说，完整恢复模式下的备份由数据库备份和日志备份组成。

```
USE master;
--1. 将数据库修改为完整恢复模式.
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
GO
```

```

--2. 执行数据库的完整备份.
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FORMAT;
GO
--3. 创建日志文件备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO
--4. 创建尾日志备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH NORECOVERY;
GO

```

图 5-7 完整恢复模式下的数据库备份示例一

在 USE master;

```

--1. 将数据库修改为完整恢复模式.
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
GO
--2. 执行数据库的完整备份.
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FORMAT;
GO
--3. 创建日志文件备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO
--4. 创建尾日志备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH NORECOVERY;
GO

```

图 5-7 中的日志备份中，只有一个常规日志备份和一个尾日志备份，在实际应用环境中，常规日志备份可能有多个（也可能没有）。

步骤 1 将数据库设置为完整恢复模式，缺省情况下，数据库的恢复模式为简单恢复模式。

步骤 2 执行数据库的完整备份，FORMAT 选项表示如果备份集文件存在则清空其内容，否则创建一个新的备份集文件。

步骤 3 使用 BACKUP LOG 创建一个常规日志备份，并且将备份写到与数据库完整备份相同的备份集文件中。这样，该日志备份的备份集编号为 2。

步骤 4 使用 BACKUP LOG 创建一个尾日志备份。尾日志备份使用“NORECOVERY”选项进行备份，尾日志备份完成后不能再对该数据库进行任何备份，这时候备份的数据库处于还原状态。尾日志备份通常在发生故障后进行备份，但任何一个使用 NORECOVERY 选项进行的日志备份都是尾日志备份。

如果最后一个日志备份不是尾日志备份，而在完成最后一个备份后又对数据库进行了修改，则在进行步骤 5 时会出现下面的错误：

消息3159, 级别16, 状态1, 第2 行

尚未备份数据库"AdventureWorks" 的日志尾部。如果该日志包含您不希望丢失的工作, 请使用BACKUP LOG WITH NORECOVERY 备份该日志。请使用RESTORE 语句的 WITH REPLACE 或WITH STOPAT 子句来只覆盖该日志的内容。

消息3013, 级别16, 状态1, 第2 行

RESTORE DATABASE 正在异常终止。

当故障发生后, 我们可以使用图 5-8 给出的操作步骤恢复备份的数据库。该恢复将数据库恢复到故障点。其中,

步骤 5 还原数据库的完整备份, FILE=1 指定从备份集的编号为 1 的备份集恢复, 该备份集是 USE master;

```
--1. 将数据库修改为完整恢复模式.
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
GO
--2. 执行数据库的完整备份.
BACKUP DATABASE AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH FORMAT;
GO
--3. 创建日志文件备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO
--4. 创建尾日志备份.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
    WITH NORECOVERY;
GO
```

图 5-7 中第 2 步创建的数据库完整备份。NORECOVERY 指定只还原数据库而不进行恢复。

步骤 6 还原第 1 个常规日志备份 (实际上只有一个)。

步骤 7 还原第 2 个日志备份 (实际上是尾日志备份)。

步骤 8 恢复数据库。在完成该步骤后数据库恢复到正常状态, 而前面的操作完成后, 数据库处于还原状态。

在操作该示例时, 建议在每次备份前对数据库进行一些相应的操作, 在恢复完成后验证这些修改操作的结果是否恢复。

```
--5.还原数据库完整备份 (自备份集 1).
RESTORE DATABASE AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=1,
NORECOVERY;
--6. 还原常规日志备份 (自备份集 2).
RESTORE LOG AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=2,
NORECOVERY;
--7. 还原尾日志备份 (自备份集 3).
RESTORE LOG AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=3,
NORECOVERY;
GO
--8. 恢复数据库:
RESTORE DATABASE AdventureWorks WITH RECOVERY;
GO
```

图 5-8 完整恢复模式下数据库的恢复示例一

5.4.2. 恢复到时间点的备份与恢复示例

在上一节中，数据库的恢复是恢复到故障点，如果缺少一个日志的备份，比如发生故障导致日志文件被破坏，或者应用要求恢复过去某一个时间，如恢复到错误删除一个表之前的状态，这时必须使用时间点恢复。这一节我们通过一个例子演示这种时间点恢复。

时间点恢复是部分恢复的一种，其它的方法还可以基于日志序列号、事务序列号等。

该示例由图 5-9、图 5-10 和图 5-11 组成，所有的操作步骤都在其中。

```
USE master;
--1. 确认数据库工作在完整恢复模式.
ALTER DATABASE AdventureWorks SET RECOVERY FULL;
GO

--2. 创建一个表，如果已经有该表则先删除.
drop table AdventureWorks.dbo.Table_1;
create table AdventureWorks.dbo.Table_1(id datetime);

--3. 备份数据库
BACKUP DATABASE AdventureWorks
TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FORMAT;
GO
```

图 5-9 完整恢复模式下和数据库备份示例二

步骤 1 将数据库 AdventureWorks 设置为完整恢复模式。

步骤 2 首先删除数据库中的表，如果该表不存在，则该命令会出现错误，不过它并不会影响后续的操作。之后，创建该表。在删除和创建表时使用了表的命名空间，AdventureWorks.dbo.Table_1 表示数据库 AdventureWorks 中的架构 dbo 中的表 Table_1。

步骤 3 创建一个数据库的完整备份，该备份所使用的备份集文件是 C:\SQLServerBackups\AdventureWorks.bak。在创建该备份之前，备份集文件中的路径必须已经存在。因为使用了 FORMAT 选项，所以所产生的备份集一定是该备份集文件中的第 1 个备份集。

在图 5-10 中的日志备份中，我们在每次备份前插入一条记录到前面所创建的表中，插入记录的值是当前系统的时间，它使用 T-SQL 函数 current_timestamp，命令 select current_timestamp 是查询当前的时间。

在执行完步骤 5 之后，不要立即执行步骤 6 和步骤 7，让第一个日志备份与第二个日志备份之间有些间隔以便于后面设定恢复的时间点，如果两者的时间间隔太短，则时间点设置就比较困难。

```
--4. 往表Table_1中插入第1条记录并查询当前时间
insert into AdventureWorks.dbo.table_1 values(current_timestamp);
select current_timestamp;

--5. 在插入第一条记录后备份日志文件.
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO

--6.插入第2条记录并显示当前时间.
insert into AdventureWorks.dbo.table_1 values(current_timestamp);
select current_timestamp;

--7. 在插入第2条记录后备份日志文件
BACKUP LOG AdventureWorks
    TO DISK = 'C:\SQLServerBackups\AdventureWorks.bak';
GO
```

图 5-10 完整恢复模式下的日志备份示例二

图 5-11 中的操作是恢复所备份的数据库，这里设置的时间点如果是插入第一条记录之前的时间，则在执行完步骤 9 之后数据库就会恢复成正常状态，所插入的两条记录均不会恢复；如果时间点设置在两条记录的时间之间，则在执行完步骤 10 数据库就会恢复到正常状态，则只有第 1 条记录会恢复；如果设置到插入第 2 条记录之后，则还需要执行步骤 11 数据库才会恢复成正常状态，2 条记录均会恢复。

```
--8. 还原完整数据库备份（在恢复之前先删除数据库）
RESTORE DATABASE AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=1,
NORECOVERY;

--9. 恢复日志文件（时间点为插入第1条记录后且最好是插入第2条记录前的时间
RESTORE LOG AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=2,RECOVERY, STOPAT = '2007-10-7 17:35:18';

--10. 恢复日志文件（注意时间与上面的时间相同）
RESTORE LOG AdventureWorks
FROM DISK = 'C:\SQLServerBackups\AdventureWorks.bak'
WITH FILE=3,RECOVERY, STOPAT = '2007-10-7 17:35:18';
GO

--11. 恢复数据库（这一步可能不再需要）
RESTORE DATABASE AdventureWorks WITH RECOVERY;
GO
```

图 5-11 数据库的时间点恢复示例

这个示例演示了数据库的时间点恢复，具体操作时间可以通过变化时间点以测试不同的恢复结果。

第6章 实验内容

6.1. 实验 1 创建和修改数据库、数据表

6.1.1. 实验目的

熟悉有关数据表的创建和修改等工作，并了解主键、外键以及约束的创建和应用，熟练掌握使用 SQL Server Management Studio 和 CREATE TABLE、ALTER TABLE 等 Transact-SQL 语句对数据表的操作方法。理解数据的完整性约束条件。

6.1.2. 实验要求

- 掌握创建数据库的技能
- 熟悉数据表的基本特点和类型
- 掌握用 SQL Server Management Studio 和 Transact-SQL 语言创建数据表的技能
- 掌握用 SQL Server Management Studio 和 Transact-SQL 语言修改数据表结构的技能
- 掌握用 SQL Server Management Studio 和 Transact-SQL 语言操作数据表中数据的相关技能
- 理解什么是数据的完整性
- 掌握约束的创建、查看及删除相关技能
- 掌握 Insert/delete/update 语句

6.1.3. 实验内容

分别使用 SQL Server Management Studio 和 Transact-SQL 语句，按下列要求创建和修改用户数据库。

1. 创建一个数据库，要求如下：
 - (1) 数据库名"testDB"。
 - (2) 数据库中包含一个数据文件，逻辑文件名为 testDB_data，磁盘文件名为 testDB_data.mdf，文件初始容量为 5MB，最大容量为 15MB，文件容量递增值为 1MB。
 - (3) 事务日志文件，逻辑文件名为 TestDB_log，磁盘文件名为

TestDB_log.ldf,文件 初始容量为 5MB, 最大容量为 10MB, 文件容量递增值为 1MB。

2. 对该数据库做如下修改:

(1) 添加一个数据文件, 逻辑文件名为 TestDB2_data, 实际文件为 TestDB2_data.ndf,文件初始容量为 1MB, 最大容量为 6MB, 文件容量递增值为 1MB。

(2) 将日志文件的最大容量增加为 15MB, 递增值改为 2MB。

(3) 收缩数据库的原因和方法

(4) 数据库的删除

3. 创建数据表。可以参考如下表结构示例, 也可以根据自己自拟数据表结构。

项目表 (Project)

字段名	数据类型	字段长度	注释
项目编码	char	10	主键
名称	varchar		
负责人编码	char	10	
客户	int		
开始日期	datetime		
结束日期	datetime		

员工数据表(Employee)

字段名	数据类型	字段长度	注释
员工编码	char	10	主键
姓名	varchar		
性别	varchar		
所属部门	varchar		
工资	money		

4. 修改表结构、添加或修改主键、外键以及各种约束的创建和应用;

5. 向创建的数据表中添加记录、修改记录和删除记录。

6.2. 实验 2 数据的基本查询、高级查询和视图操作

6.2.1. 实验目的

掌握 SQL 语句中基本查询语句的书写，包括条件查询、嵌套查询、连接查询；掌握外连接查询、统计查询、分组查询等查询语句的书写；掌握对视图的操作，理解对基本表和视图操作的异同。

6.2.2. 实验要求

- 熟悉 SELECT 语句的语法格式
- 掌握 WHERE、ORDER BY、GROUP BY、HAVING 子句的使用方法
- 掌握多表查询的概念
- 掌握内连接、外连接的使用方法
- 掌握相关和非相关子查询的使用方法
- 掌握组合查询的使用方法
- 理解视图机制，掌握用 Management Studio 工具和 T-SQL 语句创建视图、查看视图、修改视图和删除视图的方法
- 掌握用 Management Studio 工具和 T-SQL 语句创建和管理索引

6.2.3. 实验内容

1. 在查询分析器中书写 Transact-SQL 语句完成数据查询。可参考如下查询语句的要求
 - (1) 查询项目数据表中客户字段的唯一值，并查看查询结果。
 - (2) 查询工资高于 2000 的项目部的人员的姓名。
 - (3) 查询来自 CCH 公司的项目名称(以 CCH 开始)和负责人姓名
 - (4) 查询每个部门的平均工资，结果按照平均工资的多少排序
 - (5) 查询所有的员工姓名和负责人的项目名称
 - (6) 使用子查询输出所有负责 CCH 公司项目(以 CCH 开始)的员工姓名，以及没有负责 REALIDEA 公司(以 REALIDEA 开始)项目的员工姓名
 - (7) 将所有 REALIDEA 公司的项目的结束日期更改为 2011 年 1 月 8 日
 - (8) 录入部的张晓峰决定辞职，请将员工数据库中有关他的记录删除，并将他负责的项目移交给杨亭亭。书写语句对数据表做相应的更改
2. 掌握使用 T-SQL 语句创建视图的方法，包括视图的建立、删除、修改；了解如何应用视图有选择地查看所需数据，并熟悉通过视图更改数据表

中数据的方法。

- (1) 基于表"项目数据表"和"员工数据表"创建视图，要求为：
 - 视图名为"员工项目"。
 - 包含字段"编号"、"姓名"、"名称"和"开始日期"。
 - 字段别名分别是"员工编号"、"员工姓名"、"项目名称"、"项目开始日期"。
 - (2) 使用 **INSERT** 语句通过视图向员工数据表中添加一条记录，要求"姓名"字段值为"马中兴"。
 - (3) 建立适当的视图，将所有的表连接起来，观察数据，体会建立多个表的好处
 - (4) 系统存储过程语句 **sp_helptext** 查看视图
 - (5) 使用 Microsoft SQL Server Management Studio 和 T-SQL 语句修改视图
 - (6) 使用 Microsoft SQL Server Management Studio 和 T-SQL 语句删除视图
3. 给数据表创建唯一索引和聚簇索引。
- (1) 创建索引，
 - (2) 查看索引
 - (3) 使用系统视图查看索引信息
 - (4) 更名与删除索引

6.3. 实验 3 数据库编程

6.3.1. 实验目的

理解触发器的触发过程和类型，掌握创建触发器的方法。掌握存储过程的定义方法；理解游标的机制，能够使用游标编写较复杂的存储过程。定义和使用函数。

6.3.2. 实验要求

- 掌握存储过程的创建和执行方式
- 掌握存储过程的管理方式
- 掌握触发器的创建和管理方式
- 函数的定义和调用

6.3.3. 实验内容

实验内容可以参考本书 4.5 小节内容或者下面的内容。

1. 创建触发器，将插入员工的工资额限制在 5000 以内。

2. 创建触发器，将员工工资修改变动额限制在 2000 以内。
3. 删除员工数据表数据时，判断该员工是否有负责的项目，如果有，则不允许删除、并给出提示“将负责项目移交后再删除”的提示信息。
4. 在视图基础上创建 `instead of` 触发器，实现通过视图修改数据的功能，具体题目自拟。
5. 创建存储过程，要求返回某一特定部门所有员工的工资总和，其中特定部门的名称以存储过程的输入参数进行传递。
6. 创建函数，题目自拟。
7. 创建带有游标的存储过程，题目自拟。
8. 定义函数，题目自拟。

6.4. 实验 4 数据库备份、恢复及安全管理

6.4.1. 实验目的

理解数据库备份的过程和属性设置，掌握使用 SQL Server Management Studio 备份数据库的方法。理解和掌握 SQL Server 数据库安全性控制技术。

6.4.2. 实验要求

- 了解登录和用户的概念
- 掌握权限管理策略
- 掌握角色管理策略
- 掌握使用 SQL Server Management Studio 备份数据库的方法

6.4.3. 实验内容

1. 使用 SQL Server Management Studio 备份和恢复数据库。
2. 创建新用户、新用户授予权限、用户权限回收等，安全管理部分的实验内容可以参考本书 2.5 小节内容。