DBMaster

数据库管理员手册

SYSCOM

SYSCOM Computer Engineering Co./Corporate Headquarters

B1, 2-7F No. 115 Emei Street, Wanhua District,
Taipei City 108, Taiwan (R.O.C.)

www.dbmaker.com

www.dbmaker.com.tw/service

Document No.645049-237361/DBM54CN-M03312017-DBAG

发行日期: 2017-03-31

对于本手册中没有体现的关于产品最新功能的描述，请在安装完成SYSCOM DBMaster 软件后阅读 README.TXT 文件。

注册商标
SYSCOM，SYSCOM 图标和 DBMaster 是SYSCOM 公司的注册商标。
Microsoft, MS-DOS, Windows 和 Windows NT 是 Microsoft 公司的注册商标。
UNIX 是 The Open Group 的注册商标。
ANSI 是美国国家标准化组织的注册商标。

手册中提到的其他产品名称或许是它们各自持有者的注册商标，仅仅是为提供此信息。
SQL 是行业语言，并不为任何公司或任何组织所有。

注意事项
本手册中有关软件的描述，均以该软件所提供的使用许可为基础。

对于授权许可的详细信息，请与您的经销商联系。关于计算机产品的特殊用途的市场性与适用性，经销商不会给予任何说明和保证。因外界因素如地震、过热、过冷和潮湿而引起产品的任何损坏以及由于使用不正确的电压和不兼容的软硬件而引起的损失和损坏，经销商概不负责。

虽然该手册的内容已经过仔细核对，但错误再所难免。若手册再有改动，不另行通知。还请见谅。

目录

1 简介

欢迎使用DBMaster数据库管理员手册，本手册将提供给您作为DBMaster数据库管理员所必须了解的知识。DBMaster是一个功能强大且使用灵活的SQL数据库管理系统（DBMS），它支持交互式的结构化查询语言（SQL），Microsoft开放式数据库连接（ODBC）标准接口，以及嵌入式的ESQL/C语言。DBMaster也支持Java工具的接口，针对COBOL语言的DCI接口。由于DBMaster完全遵循开放式的架构，以及标准ODBC接口，您可以轻松地利用前端开发工具开发应用程序，或利用市面上的ODBC、JDBC或DCI兼容应用软件来查询DBMaster数据库的内容。

DBMaster可以很容易地从个人使用的“单用户数据库”升级到企业级的“分布式数据库”。无论您使用的是单一使用者数据库还是商用数据库，DBMaster先进的安全性、完整性和可靠性都能保证用户重要数据的安全。另外，DBMaster的跨平台支持特性则在您需要作硬件升级时，提供给您最佳的调整弹性。

DBMaster提供了卓越的多媒体处理功能，可以让您储存、查询、恢复和操纵各种类型的多媒体数据。利用DBMaster提供的二进制大型对象（BLOBs），可以让您的多媒体数据完全享有DBMaster先进的安全性和崩溃恢复功能。而利用文件对象（File Object）的数据类别，也可以让您的DBMaster数据库有能力管理外部编辑的文件。

此手册是针对那些既不熟悉DBMaster DBMS的概念和原则，也不熟悉DBMaster SQL查询语言语法的数据库管理员们而设计的。然而，您对计算机操作应具备一定的基础，并且对运行DBMaster的操作系统能够熟练

操作。不过本书并不会说明操作系统的操作方法，当您有这方面的问题时，您必须参考相关的操作系统说明。

基本上，本手册会介绍作为一个DBMaster系统管理员所应该具备的管理观念和原则，也会介绍如何使用DBMaster SQL命令建立和维护一个DBMaster数据库，以及如何调整DBMaster的执行效率。为了让您能够容易的了解这些内容，我们会以例子和图解来辅助说明。

一个数据库架构的好坏对系统的执行效率有很大的影响，其中必须考虑的因素很多，包括数据的存放位置，如何访问数据，是否需要建立索引，以及如何保护数据等。本手册也会针对这些相关的因素加以说明，使您能有更多的知识来判断如何让DBMaster数据库以最佳的效率工作。由于在说明的过程中本手册大多以SQL命令来解说，因此，如果您能熟悉SQL语言，这将有助于本手册的阅读。

本手册中的大多概念，命令和例子都可以使用DBMaster提供的命令行工具dmSQL来实现。然而有些管理功能只能通过其它的DBMaster工具来操作，本书则会以其它工具来说明。有关如何使用DBMaster工具操作的信息，请参考1.1章节*其它相关文件*。

1.1 其它相关文件

除了本书之外，我们还为您提供了其它用户手册和参考文献，帮助您更深地了解DBMaster数据库管理系统。

您可以根据自己的需要，参考相关手册：

- 有关DBMaster SQL语言的语法和使用的相关信息，请参考*SQL命令与函数参考手册*。
- 有关嵌入式的ESQL/C语言的语法和使用，请参考*ESQL/C程序员参考手册*。
- 有关dmSQL命令行工具的使用方法，请参考*dmSQL使用手册*。
- 有关DBMaster的错误信息和警告信息，请参考*错误信息参考手册*。
- 有关使用DBMaster工具来配置和管理数据库的信息，请参考*数据库管理工具用户手册*、*服务器管理工具用户手册*和*配置管理工具用户手册*。
- 有关SQL存储过程的使用方法，请参考*SQL存储过程用户手册*。
- 有关本地ODBC API和JDBC API的详细信息，请参考*ODBC程序员参考手册*和*JDBC程序员参考手册*。

1.2 技术支持

在软件试用期间，Syscom Computer Engineering Co.(Syscom)将为您提供30天的免费email支持和电话支持。当软件注册后，我们还会再为您提供30天的免费技术支持。如此一来，您就可以获得60天的免费支持。不仅如此，在您购买软件后，Syscom对任何问题都会以email的方式为您提供技术支持。

您除了可以获得免费的技术支持外，还可以以20%的零售价购买其它产品。要想获得更多的详细资料和价格信息，请与sales@dbmaker.com保持联系。

您可以通过任何一种方式（普通信件、电话或email）与Syscom技术支持保持联系，请登录至：www.casemaker.com/support 以获取详细信息。在与Syscom技术支持联系之前，请先查询当前数据库的常见问题解答。

无论您以何种方式与Syscom的技术支持联系时，请务必写上以下有效信息：

- 产品名称和版本号
- 注册号
- 注册的用户名和地址
- 供应商/发行者的地址
- 操作平台和计算机系统配置
- 错误发生前执行的动作
- 如果可以，请提供错误信息和编号
- 其它一些相关信息

1.3 文档协定

为方便用户的阅读和使用，本手册使用了一种标准的排版约定，注释、程序、示例和命令行都用缩进排版的方式进行了特别的设置。

协定	说明
斜体字	斜体字表示必须输入的信息占位符，例如用户名和表名。此字符可用实际的名称来替换。有时，文档也会使用斜体字来介绍新的关键字，强调着重点。
黑体字	黑体字表示文件名、数据库名、表名称、字段名、用户名和其它数据库对象。它也用于强调程序执行步骤中的菜单命令。
关键字	文字段落中， SQL语言使用的关键字都是以大写字母出现的。
小符号	文档中出现的小写字符表示键盘上的按键，两个键名之间的加号（+）表示在按住第一个键不放的同时，再按第二个键。两个键名之间的逗号（，）表示释放第一个键以后，再按第二个键。
注意	包含一些重要的信息。
➲ **程序**	表示后面跟随的是程序的执行步骤或连续的项目。很多任务都是通过这种方式描述，给用户提供一个逻辑顺序步骤得以效仿。
➲ **示例**	例子用来阐明描述，通常包括屏幕上出现的文本，用户也可以将这些例子输入到计算机中，通过屏幕看到运行结果。当然，示例还包括一些原型和语法。
命令行	包括文本，这些命令都可以输入计算机中，显示在屏幕上。通常用于显示 SQL 命令的输入输出或 dmconfig.ini 中的内容。

表 1-1 文档协定

2 概述

组成数据库的数据文件的物理结构非常复杂。如DBMaster数据库管理系统，将数据库的执行和整个数据分离开。该数据库被视为二维表所组成的集合，而这些二维表包含了赋有数据值的行和字段。这些表十分形象化，为数据建模提供了灵活性。

DBMaster提供了一系列从表中获得数据的方法：交互式的dmSQL命令行工具为日常的事务处理或ad-hoc查询提供了方便，并且DBMaster的应用程序接口（API）为应用程序快速方便的开发提供了基础。DBMaster也提供了很多跨平台的简单易用的图形化工具。

2.1 DBMaster的特征

作为一个关系型数据库管理系统，DBMaster在继承所有传统数据库管理系统特征的同时，还增加了很多强大而高级的特征。这些特征不仅能提高DBMaster的操作性能，还能提供给用户很多传统数据库所无法比拟的特征，尤其是支持多媒体技术。

支持多媒体

强大的多媒体处理能力允许数据库有效的存储和管理多种多媒体数据，包括文本、图表、音频、视频和动画。这些多媒体处理能力也为用户提供了很大的灵活性，并允许用户以不同的方式来存储这些多媒体数据。

多媒体特征包括：

- 二进制大型对象（BLOBs）和文件对象（FOs）
- 表中的BLOB和FO字段
- 可以使用现有的多媒体工具来编辑文件对象
- 内建全文搜索引擎

多媒体数据可以作为二进制大型对象（BLOBs）直接存储在数据库中。此数据类型和传统数据类型相比，具有同样的安全性、可靠性和完整性。另外，多媒体数据也可以作为文件对象来存储，并且允许在数据库的控制下被第三方多媒体工具完全访问。

支持64位

DBMaster为windows x64和Linux x64操作系统提供64位端口的支持。用户必须为64位windows或Linux操作系统环境下，x86-64结构的CPU安装相适合的64位DBMaster版本。

64位版本有以下限制：

- 64/32位数据库服务器创建的数据库无法被32/64位数据库服务器启动，并且存储过程和用户自定义函数在这两个版本上不兼容。
- 如果用户希望在不同的操作系统中实现，则必须移植他们的数据库。然而，32/64位的用户可以连接至64/32的数据库服务器。
- 用户必须使用64位的C编译器去编译、创建用户自定义函数、ESQL/C以及存储过程。对.NET程序，用户必须使用VS2005或更高版本来编译和链接64位应用程序。对JDBC或java存储过程，用户必须使用64位JVM来编译java程序。
- DBMaster在64位机器上的共享内存大小为2^{31}页（2^{31}×页大小字节），而在32位环境上的共享内存大小为2GB字节。

支持JDBC

DBMaster不仅支持JDBC 3.0功能，同时也支持Java Transaction API（JTA）功能。JDBC JTA允许连接现行的Java AP服务器，例如BEA WebLogic™。

要想了解JDBC和JDBA工具，请参考产品用户手册。有关JDBC的详细信息，请登录至：http://java.sun.com/products/jdbc/

有关JTA的详细信息，请登录至：http://java.sun.com/products/jta/

支持Microsoft处理服务器（MTS）

MTS（Microsoft事务处理服务器）是Windows NT的主要组成部分。在Windows2000系统中，MTS是操作系统默认安装的一部分。在Windows NT平台上，MTS作为事务处理服务器，提供其它平台上类似软件的功能，例如CICS、Tuxedo等，这是专为创造稳定的数据源环境而设计的。

DBMaster支持MTS，因此用户可以执行事务操作。

使用DBMaster和MTS时需要注意下列事项：

- DBMaster需要Microsoft Data Access Components （MDAC）2.6或者更高版本与MTS协同作业。最新的MDAC版本请从下列地址下载：http://www.microsoft.com/data
- 如果使用MDAC 2.5，请在**dmconfig.ini**配置文件的DM_COMMON_OPTION节中添加DM_DifEn = 0选项。

支持开放式平台

使用DBMaster内置的ODBC 3.0兼容性平台和支持ANSI SQL-92的特性，您就可以利用当前的多种流行工具（Visual C++、Visual Basic、Delphi和AcuBench）来快速地创建应用程序。DBMaster允许开发者和管理员使用他们自己熟悉的工具来开发应用程序，并且无需约束开发环境。

开放式平台特征包括：

- 符合ANSI-99标准
- 支持ODBC 3.0
- ESQL/C预处理程序
- 支持JDBC 2.0

内置的ESQL/C预处理程序和传统的C开发环境相比，大大简化了程序的开发过程。数据库应用程序可以使用高性能的嵌入式SQL查询语言来编写，并且可以被DBMaster预处理程序自动地转换成ODBC的函数调用。

数据完整性

DBMaster也支持传统的数据完整特性。主键、外键和参照行为确保了数据的完整性。用户定义的数据类型、域、字段和表约束确保了在指定域中只能输入有效数据。

数据完整性特征包括：

- 主键和外键的完整性检查

- 完全支持参照行为
- 表和字段约束
- 用户自定义数据类型
- 默认字段值

数据可靠性

由DBMaster提供的高性能数据保护特征，例如自动崩溃恢复，数据库的一致性检验和自动备份功能，确保了数据的安全性。这些特征能够保证在操作系统发生损坏或出现磁盘故障时，数据的一致性和安全性。

数据可靠性特征包括：

- 在线事务处理
- 在线完整、差异和增量备份
- 自动崩溃恢复
- 自动增量备份
- 自动更新统计
- 数据一致性检验
- 支持多日志文件
- 可以进行BLOB备份

存储管理

DBMaster的存储管理提供了灵活的数据存储功能。它对表中的行数和数据库中的表数量都无实际限制。一张表甚至可以覆盖几个磁盘！ 同时，DBMaster也支持在线更改表结构，所以用户可动态的开发应用程序以适应所需。

存储管理特征包括：

- 固定表空间和自动扩展表空间

- 在UNIX平台上支持裸设备
- 数据库最大限制256 PB （petabytes： 1PB = 1,024 TB）
- 数据库中的表数无实际限制
- 表中的记录数无实际限制
- 动态重新定义表结构

DBMaster也可以动态扩展存储空间，但受到有效磁盘空间的限制。存储空间的大小也可以被固定或手动调整。在UNIX平台，DBMaster支持裸设备，允许数据绕过文件系统而直接写入其中。

安全性管理

DBMS的集中式特性和多用户特性，要求数据库必须具备一些安全控制机制，这样不仅可以防止用户的越权存取，也可以限制对特许用户的访问。用户级别和组级别可以对数据库的访问权限进行限制。表和字段的权限管理可以控制他们的访问对象。

安全性管理特征包括：

- 用户级别和组级别的安全保护
- 嵌套的组
- 表和字段的权限管理
- 存储命令和存储过程的权限管理

高级语言特征

高级语言特征弥补了传统数据库的不足。用户可以使用存储命令、存储过程、触发器和用户自定义函数对DBMaster的功能进行扩展和定制。逻辑事务可以直接写入数据库引擎中以集中控制，从而简化了数据库的管理和维护。

高级语言特征包括：

- 内置函数
- 用户自定义函数
- 存储命令
- 存储过程

2.2 数据库模式

数据库管理员可以以不同的模式来启动数据库。每种模式都为连接和访问数据库提供了不同的选项，它可以将数据库的单用户系统转换成大型的多用户分布式系统。

数据库模式要依据数据库服务器的运行平台来选择。DBMaster支持三种数据库模式：单用户模式、多连接模式和客户机/服务器模式。

单用户模式

单用户模式只能用于UNIX/Linux平台。对于非共享的数据库而言，此模式是DBMaster的一个简化版本。由于单用户数据库无需考虑锁、安全性和网络支持，所以此模式的优势在于小巧的应用程序和优越的执行速度。

由于在此模式下只能同时建立一个连接，所以数据库无法运行别的服务器和后台程序，如备份服务器、复制服务器、全局事务处理服务器。再者，由于此模式下的数据库无法通过网络互联，所以用户只能通过主机来访问数据库。

多连接模式

多连接模式只能用于Windows平台。此模式的优势在于利用DBMaster的安全性和可靠性，数据库可以同时开启多个连接。但由于没有网络支持，所有连接都必须通过主机来访问数据库。

此模式的局限性在于数据库不支持其它服务器和后台程序，如备份服务器、复制服务器或全局事务处理服务器。

客户机/服务器模式

客户机/服务器模式支持所有平台。此模式允许将多个同时连接到一个数据库的连接，通过TCP/IP网络从任何一个已连接的计算机连接到主机

上，并且支持DBMaster的安全性、可靠性和并发控制等特征。除此之外，还可以通过数据加密来提高网络传输的安全性。此模式支持所有额外的服务器和后台处理程序，如备份服务器、复制服务器和全局事务处理服务器。

注意 *在同一时间，一个连接不支持多个语句操作。*

2.3 DBMaster的操作平台和工具

DBMaster通过一个应用程序接口和几种工具就能轻松地管理一个数据库。这些工具包括基于交互式SQL查询语言的命令行工具和管理多服务器的图形化工具。初学者可以运用跨平台一致性的图形化工具来理解DBMaster的特性。

应用程序接口

应用程序接口（API）是一个直接操作在数据库引擎上的低级程序库。API可用于一般的程序设计语言如C++、Visual Basic来创建应用程序。DBMaster支持ODBC 3.0接口，在支持当前所有核心功能的同时，也提供了很多额外的功能。

dmSQL交互式查询工具

dmSQL是一个基于字符的交互式工具，可直接使用DBMaster的各种功能。使用dmSQL来操作数据库，执行特殊查询也可以立即查看到运行结果。很多时候，dmSQL是唯一不需使用常规编程语言来开发程序就可以达到完全使用数据库功能的工具。

数据库管理工具（JDBA）

JDBA工具是一种跨平台的图形化用户工具，该工具将有助于用户方便地管理DBMaster数据库对象。对于使用高效、灵活的SQL数据库管理系统DBMaster而言，JDBA工具隐藏了DBMS和操作语言的复杂性，并且提供了易于理解、使用方便的图形界面。它允许用户无需学习查询语言就能访问数据库，也允许用户无需考虑SQL语言的命令输入格式，就能快速地管理和操作数据库。JDBA工具也向用户提供了监视数据库用的统计信息。

服务器管理工具（JServer Manager）

JServer Manager是一个可以在多种系统平台上运行的图形化工具，主要用于创建和管理数据库。功能包括：创建、启动、关闭、删除、诊断、备份、恢复数据库。

配置管理工具（Jconfiguration）

JConfiguration工具是管理所有数据库配置参数的图形化工具。它为更改DBMaster配置文件中的关键字提供了一个简单直接的方式。每一个配置参数都在用户界面内定义清楚，它可以减少查阅参考文档和记忆关键字的需要。

ESQL/C语言预处理器

ESQL/C语言预处理器是一个图形化、交互式工具，用于编辑和预处理ESQL/C程序。它为管理多个ESQL/C程序，并执行编辑/预处理过程提供了一个容易使用的平台。在预处理过程中，针对那些警告/错误信息，您只需点击输出窗口中的每一个出错命令行就可以对它们进行检验。

2.4 语法图说明

本手册列举的语法图用于指示所有SQL命令的语法。这些语法图为我们在命令行工具中建造SQL语句提供了帮助。下图2-1就是一个语法图的示例。

为了使用语法图，我们可以用一个由起点到终点的线路来表示。用户不能省略命令中必须的元素，但可以根据需求来选择可选的命令选项。

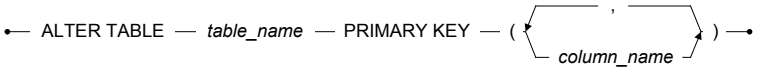


图2-1 ALTER TABLE语句语法图

斜体部分表示数据库中名称的占位符，在实际输入过程中，我们可以用真实的名称来替代这些占位符。在上例中，我们可以用数据库的表名来替代<table_name>。例如：用**Customers**来替代<table_name>，可在**Customers**表上执行命令。

同时也请您注意语法图中的箭头方向，有些时候它会形成一个循环，代表循环中的项目是可以重复的，通常可重复的项目会以逗号分隔，如上例中的字段名。

3 系统架构

本章我们将讨论DBMaster的两种不同架构。首先我们将介绍DBMaster的进程和数据库通信与控制区域（DCCA），它为每个已启动的数据库存储了必要的信息，然后再介绍DBMaster的两种架构。

3.1 DBMaster进程

DBMaster进程主要是根据用户指令和数据库功能来存储和管理数据。 一个DBMaster进程由数个层级组成。如图3-1所示，用户的应用程序通过应用程序接口（API）和DBMaster进行通信。

API将传递用户指令或传递函数调用给SQL引擎，并将它们解释成一连串的数据库引擎功能调用。同时，SQL引擎也将解析SQL指令并把这些指令转化成数据库引擎能接受的一组函数序列，数据库引擎将依据这组函数序列来处理表中的数据。

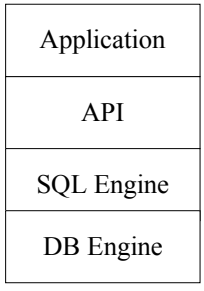


图 3-1DBMaster进程

SQL引擎和数据库引擎所扮演的角色是截然不同的。基本上，SQL引擎处理SQL的分析和查询优化，数据库引擎则负责管理存储空间及内存缓冲区、同步控制、崩溃恢复及其它。所有的模块相互合作来维持整个数据库内数据的一致性。大多数性能调优都与底层的数据库引擎相关。

API层和SQL引擎在单用户模式和客户机\服务器架构模式是相同的。 然而，数据库引擎在单用户模式和客户机\服务器架构模式则有所不同。单用户模式仅能服务一位用户，而客户机\服务器架构模式则能同时服务多位用户。

在客户机\服务器架构模式下，应用程序和API层是结合在一起的，并且执行于客户端机器上；而SQL引擎和数据库引擎结合在一起，并且执行于服务器端。在这种架构之下，API层是通过网络协议来和SQL引擎通信的。

3.2 数据库通信与控制区域（DCCA）

在启动数据库时，DBMaster会首先分配一大块内存区域来存储数据库的相关信息，如缓冲区和不同类型的控制信息。这一大块内存称之为数据库通信与控制区域（DCCA）。它包含三块区域：页缓冲区、日志缓冲区和系统控制区域（SCA）。

数据库通信与控制区域（DCCA）对于DBMaster来说非常重要，特别是在客户机\服务器架构模式下。在Microsoft Windows和UNIX单用户模式下，DCCA都是从私有内存中分配而来的。而在UNIX系统的客户机\服务器架构模式中，DCCA必须被所有同一数据库的DBMaster进程共享，所以它不能从私有内存中配置，故可利用UNIX中标准的共享内存技术来配置数据库通信与控制区域。所有运行在C/S模式下的DBMaster进程都是通过DCCA来互相通信的。

我们可以很容易地调整数据库通信与控制区域的大小和其使用方法，而这种调整将对DBMaster的性能产生很大的影响。有关DCCA的详细描述请参考第18章*性能调优*。

3.3 单用户模式架构

DBMaster的单用户模式是只能服务一位使用者的关系式数据库，因单用户模式不需处理并发控制，所以它的程序较小且执行速度较快。假如您的数据库仅为一人使用，单用户模式的DBMaster将是您最好的选择。下图3-2显示了DBMaster的单用户模式架构。

因为仅有一位使用者连接到单用户模式的DBMaster数据库，数据库通信与控制区域将在私有内存中配置，而非在共享内存中配置。因此此模式不具有加锁机制和功能。基于性能的考虑，在DBMaster引擎执行时会将所有数据在内存中进行处理，而后适时地将修改过的数据写入磁盘文件中（包括数据文件和日志文件），**dmconfig.ini**配置文件是一个纯文本文件，用来储存DBMaster的配置参数。

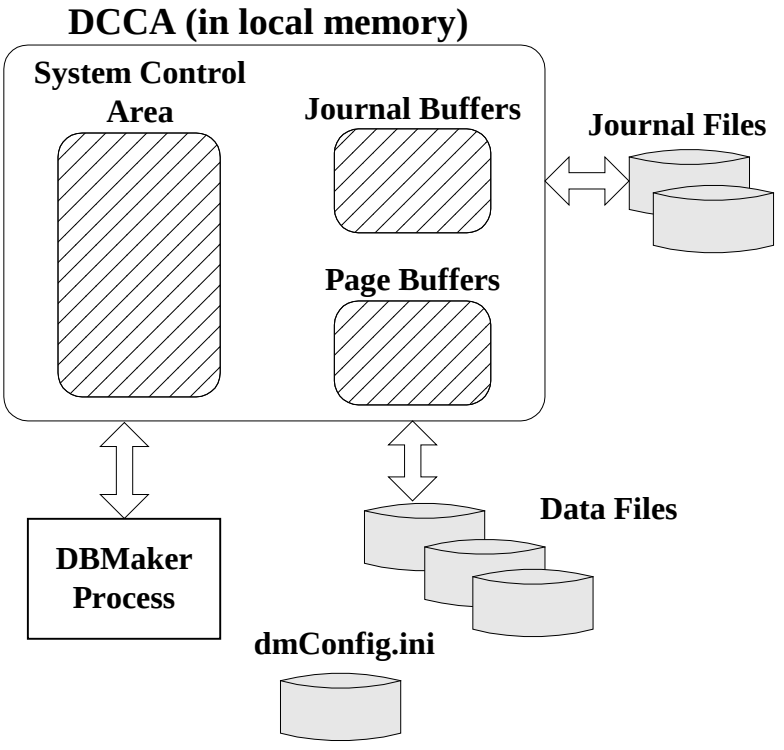


图3-2 DBMaster单用户模式架构

3.4 客户机/服务器模式

DBMaster的应用程序不仅能执行于单用户模式下，而且也能执行于客户机\服务器架构模式下。在客户机\服务器架构模式下，会有两个进程参与运作：客户端进程和服务器端进程。一般来说，客户端进程执行于前端的个人计算机或工作站，且使用DBMaster所提供的应用程序接口，通过网络来和服务端进程通信。请注意，在客户机\服务器架构模式下，所有的机器都可以使用不同的硬件平台。

在DBMaster的客户机\服务器架构模式下，网络管理模块必须存在于客户端和服务器端。网络管理模块负责在客户端和服务器端之间传送数据。在客户机\服务器架构模式下，网络通信协议非常重要。目前DBMaster仅支持一种网络协议——TCP/IP。假如在DBMaster的客户机\服务器架构模式下，您的系统并不支持TCP/IP，那么您必须在执行DBMaster程序前安装TCP/IP协议。举例来说，假如您使用UNIX或者Windows作为您的客户端平台，那么您就不需要额外安装TCP/IP协议。因为上述操作系统均支持内建的TCP/IP。在Windows平台下，TCP/IP协议仅仅是安装在系统内的特定网络协议之一。图3-3显示了DBMaster客户机/服务器模式的系统架构图。

在Unix系统中，当一个客户端程序连接到数据库服务器时，DBMaster网络服务器会将其转给其它服务器进程来处理并发查询，最初的网络服务器将继续等待其他用户的连接。

在Windows NT下的情形则稍有不同，因为NT是一个多线程（multithread）系统，所以在NT下的网络服务进程（**dmserver.exe**）也是一个多线程的程序。所以当客户端进程连接NT上的网络服务进程时，DBMaster的网络服务进程会创建一个线程（thread）来处理接踵而来的查询。在这种情况下，数据库通信与控制区域就会从进程拥有的内存中配置，而非从共享内存中配置。所以在Windows NT系统中，一个数据库仅有一个 DBMaster服务进程。因为越来越多的操作系统可支持多线程，当前研究也指出多线程程序比多进程程序效率更高，所以DBMaster将尽可能使用多线程的技术来代替多进程。

客户机\服务器架构模式中的DBMaster有三个组件：服务端程序、客户端程序和客户端程序库。每一个组件的作用如下文所述。

服务器程序

DBMaster的服务端程序名为**DmServer**。此程序包括了网络管理模块和处理数据存取的数据库引擎。在客户端程序连接到数据库服务端程序之前，必须先启动服务端程序，这样客户端程序才能连接到数据库服务器上。

客户端程序

DBMaster的客户端程序名为**dmsqlc**。用户可以使用此程序来连接数据库，并且下达SQL指令来处理数据。

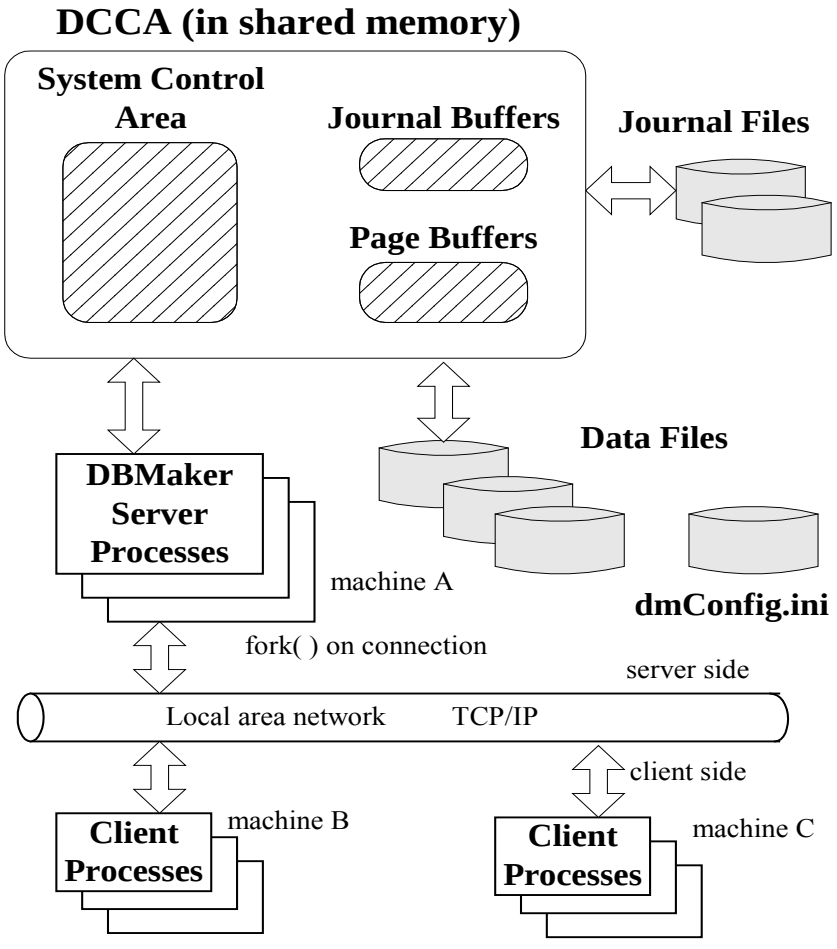


图3-3 DBMaster客户机/服务器模式的系统架构

客户端程序库

DBMaster的客户端程序库在UNIX系统上为 **libdmapic.a**，在微软的Windows系统下为**dmapi<version number>.lib**。使用者如果想要开发

自己的客户端程序，必须和这个程序库结合。举例来说，使用者可以使用各种开发工具来开发他们的前端应用程序。当建立前端应用程序时，使用者必须连接这个程序库。唯有如此，使用者自行开发的应用程序才能和服务端程序相连。

4 基本的数据库管理

本章我们将介绍基本的数据库管理，包括如何利用交互式的SQL工具（dmSQL）来创建数据库、启动数据库、连接数据库和关闭数据库。为了演示我们的操作，数据库管理员可以通过**dmSQL**命令行工具和编辑**dmconfig.ini**配置文件的方式来执行上述操作。当然，数据库管理员也可以使用JConfiguration工具和JServer Manager工具来执行。

以下章节我们将描述基本数据库管理所必需具备的配置参数和命令。第一章主要描述了配置文件的角色和格式，其它章节描述了一些功能的特殊设置以及这些设置在创建、启动和连接时，如何影响数据库的执行。

4.1 配置文件 - dmconfig.ini

当DBMaster运行时，它需要读取很多的配置参数。DBMaster引擎将利用这些配置参数来设定数据库的运行状态。数据库中的文件存储单元、内存运行时间和网络连接都需要通过配置参数来设置。这些参数都存储在**dmconfig.ini**配置文件中。配置变量可以设置成一个关键字等于某个数值的形式（请参考后面的格式部分）。用户可以通过设定配置参数和使用 JConfiguration工具的方式来定制数据库。JConfiguration工具通过使用图形用户界面来简化对配置参数的管理。要想获得更多有关JConfiguration工具的信息请查询*配置管理工具用户手册*。某些参数（关键字）必须在数据库创建前设置，有些必须在数据库启动前设置。此外，有些参数在数据库创建后不能被更改。以下章节将描述如何通过直接更改配置文件的方式来管理这些设置。要想获得**dmconfig.ini**的所有参数，请查阅第20章*dmconfig.ini中的关键字*。

dmconfig.ini文件的存放位置

在UNIX平台上，DBMaster可从以下三个地方来查找**dmconfig.ini**配置文件：

- 当前目录
- DBMaster环境变量指定的目录
- DBMaster安装目录（*~DBMaster\version*）

DBMaster 将顺序的搜索**dmconfig.ini**的每一个存放位置。如果数据库节没有在**dmconfig.ini**配置文件中的某一位置，那么DBMaster将顺序的搜索下一个位置。

然而在Microsoft的Windows系统下，**dmconfig.ini**文件的存放位置与UNIX有所不同。DBMaster将仅从以下两个位置来搜索**dmconfig.ini**配置文件：

- 由DBMaster环境变量指定的目录

- DBMaster的安装目录，典型的Windows安装目录通常是 ***C:\DBMaster\Version***

当DBMaster需要查询某一个数据库中的参数时，DBMaster会扫描上述三个目录（若是Microsoft Windows系统则为**安装**目录）来寻找dmconfig.ini配置文件中的对应值。使用者可以使用任何一种文本编辑器来添加或修改**dmconfig.ini**配置文件中的参数值。

当数据库管理员创建一个新的数据库，且**dmconfig.ini**的各节中没有相同数据库名时，DBMaster会于最先找到的配置文件中或在当前目录下新建的**dmconfig.ini**文件中（若是微软Windows系统则为**安装**目录），新增一段以数据库名为节名的新节。

因此，当数据库管理员启动数据库时，其对应的节必须存在于配置文件中。否则，DBMaster会回传错误信息。 虽然您可以在上述三个目录中分别建立配置文件，并在不同的配置文件中写入各种节，但实际上我们并不建议您如此操作。使用一个大家共享的**dmconfig.ini**配置文件，能让您更容易地管理数据库系统。

JConfiguration会显示所有配置文件中列出的数据库节。在UNIX系统上，JConfiguration工具将显示以上列举的所有目录中配置文件的值。

dmconfig.ini配置文件格式

dmconfig.ini配置文件分为几个不同的节。第一节是一些常用关键字的定义，接下来是标题名，也就是数据库的名称。节中的关键字定义了此数据库的设定值，所有分号后的字符串均视为批注。

➲ **示例**

dmconfig.ini配置文件的一般格式：

```
[Section_name1]
<key_word1> = <value1>
<key_word2> = <value2> <value3>    ; this is a comment;
...

[Section_name2]
<key_word3> = <value4> <value5>
```

```
<key_word4> = <value6>
...
```

文件名称和大小

数据库由操作系统文件组成，这些文件都是在**dmconfig.ini**配置文件中用关键字来定义的。此参数**<filename>**可由一个参数值<value>来替换。<filename> 参数可以是一个简单的文件名，如**firstdb.sdb**；一个相对路径，如 ***mydb/firstdb.sdb***；或一个完整路径，如 ***/disk1/mydb/firstdb.sdb*** （“/”是UNIX系统的分隔符，“\”是Windows系统的分隔符）。

其中<np>参数代表页数。页是磁盘空间中文件的存储单位，页的大小由关键字**DB_PgSiz**来设定，默认的数据页大小是8K。

除了使用页为单位，用户还可以指定M或G为单位。在没有指定M 或G的情况下，默认单位是页。如果指定M或G作为单位，那么实际大小要比指定值少一页。例如，如果数据页的大小是16K，而文件大小设置为8M，则文件的实际大小将是8,176K而不是8,192K。

➲ **示例**

指示文件名和文件大小的一般格式：

```
[Section_name1]
<key_word1> = <filename>
<key_word2> = <filename> <filename>
<key_word1> = <np>
```

文件的存放位置

假如不同的使用者要从不同的工作目录中来存取数据库，那么在**dmconfig.ini**中的文件名就必须为全路径名，只有这样，DBMaster才能正确地找到文件。

另外一种方式为设定关键字**DB_DbDir**，此关键字指示了数据库的主目录（数据库目录）。

➲ **示例1**

以下将数据库的目录名设置为db，而非目录标题名DB1。而且数据库文件也可以分别放在不同的磁盘上。

```
[DB1]
DB_DbDir = /disk1/db
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

实际的文件名为：

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
DB_BbFil -- /disk1/db/DB1.SBB (using default file name)
```

➲ **示例2**

使用**DB_DbFil**关键字：

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

实际的文件名为：

```
DB_DbFil -- mydb2 (in current directory)
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.SBB (in current directory)
```

注意 *此规则也适用于用户定义文件。*

一些重要的关键字

以下列举了一些重要关键字的简短描述。有关创建和启动数据库的关键字将在本章随后章节详细介绍。第20章为DBMaster关键字的详细清单。下面我们将列举几个在**dmconfig.ini**文件中出现的有效关键字：

- **DB_DbDir = <***文件名***>** — 数据库文件的存放路径
- **DB_DbFil = <***文件名***>** — 系统数据库文件的文件名为**<*filename*>**
- **DB_PgSiz = <4, 8, 16, 32>** — 数据页大小（4KB、8KB、16KB或32KB）

- **DB_JnFil** = <*文件名*> — 系统日志文件的文件名为<*filename*>
- **DB_JnlSz** = <*np*> — 系统日志文件大小为<np>个页
- **<*logical_file*>** = <*文件名*> <*np*> — 指定用户定义的文件名 **<*logical_file*>** 对应到<*文件名*>，而其大小为<*np*>个页。换句话说，<*filename*>是<*logical_file*>的实际文件名，而<*logical_file*>是数据库系统内部所使用名称。
- **DB_NBufs** = <*np*> — 指定缓冲区的大小为<*np*>个页
- **DB_SvAdr** = <*IP地址*> 或 <*主机名*> — 数据库服务器的IP地址或主机名。在客户机/服务器模式下，此选项必须在客户机端设置。
- **DB_PtNum** = <*端口号*> — 数据库服务器和客户端的TCP/IP通信端口号。
- **DB_MaxCo** = <*number*> — 数据库可以操作的最大连接数。

注意 *除了<logical_file>区分大小写 外，其余所有符号均不区分大小写。*

默认值

一些参数有默认值。因此，如果在**dmconfig.ini**中没有对该参数进行设定，那么将使用它的默认值。有关关键字的详细描述以及默认值的设定，请查看第20章*dmconfig.ini中的关键字*。

支持环境变量

对于存储在CD-ROM上的只读数据库而言，用户很难在**dmconfig.ini**文件中指定路径。如果DBMaster支持默认的环境变量**$APP_HOME**和**$APP_DRIVE**将会使其变得简单。

- **$APP_HOME**: DBMaster的主安装目录，从注册表中获取DBMaster的HOME信息。例如DBMaster安装在***D:\dbmaster\5.4***目录下，当读取**dmconfig.ini**文件时，DBMaster会自动将环境变量**$APP_HOME**替换为“***D:\dbmaster\5.4***”。

- **$APP_DRIVE**: 此变量将在Linux操作系统返回一个空字符串，在Windows操作系统上返回一个主安装目录所在的驱动器盘符。例如在***D:\dbmaster\5.4***目录下安装DBMaster，DBMaster将返回驱动盘符“***D:***”，并且当读取**dmconfig.ini**文件时，自动将目录“***D:\dbmaster\5.4***”替换为“***D:***”。

DBMaster同样也支持系统环境变量，例如在操作系统中定义的***$TEMP*** = "***C:\TEMP"***。当读取**dmconfig.ini**文件时，DBMaster会自动将**$TEMP**替换为“***C:\TEMP***”。

如果系统环境变量中已经定义了默认环境变量（**$APP_HOME**、**$APP_DRIVE**），当读取**dmconfig.ini**文件时，DBMaster将不会找到此系统环境变量值，而会优先使用默认环境变量值。

➲ 示例

如果用户有一个CD-ROM，用户可以按照以下设置将DBMaster软件和数据库放置到CD-ROM中：

```
dmconfig.ini
[DBSAMPLE5]
DB_DbDir=$APP_DRIVE\database
DB_FoDir=$APP_DRIVE\database\fo
DB_TpFil=$TEMP\DBSAMPLE5.tmp
DB_SMode=6
```

dmconfig.ini配置文件示例

下例在**dmconfig.ini**中定义了两部分内容：一个是**Personnel**数据库，另一个是**LIBRARY**数据库。

➲ 示例

dmconfig.ini配置文件的典型示例：

```
[Personnel]
DB_DbFil = /disk1/bin/PERSONNEL.DB
DB_JnFil = /disk1/bin/PERSONNEL.JNL
f1.os = /disk1/bin/PERSONNEL.OS 100
f1.blob = /disk1/bin/PERSONNEL.BLOB 1000
```

```
DB_NBufs = 0          ; auto-configure number of data buffers
DB_NJnlB = 100        ; number of journal buffers
DB_MaxCo = 100        ; maximum number of connections
DB_JnlSz = 2000             ; size of journal file (pages)
DB_RTime = 0          ; restoration target time
DB_SvAdr = 192.72.116.130   ; server's IP address
DB_PtNum = 21000        ; and port number

[LIBRARY]
DB_DbFil=/disk3/usr/lib/library.db
DB_JnFil=/disk3/usr/lib/library.jnl
DB_SvAdr = 192.72.116.137
DB_PtNum = 26999
DB_JnlSz = 2000
```

4.2 创建数据库

创建数据库需要事先制定一些计划。创建数据库前必须先考虑好一些配置参数的设置，因为当数据库创建成功后，有些配置参数就不能被更改了。以下参数表示数据库创建成功后不能再更改的关键字：

- 数据库名称
- 数据库安全性（不管数据库是否拥有不同的用户权限）
- 事件敏感度（用于指定是否区分对象名称中的大小写字母）
- BLOB帧大小（为每个BLOB帧分配的磁盘空间大小）
- 语言设置（用于指定字符类型：ASCII、Big5……）
- 语言代码命令（此模式用于对字符类型的数据进行排序）

其它参数在数据库创建好后还可以被更改。但是在数据库创建之前，也要对他们进行认真考虑，这些参数包括：

- 表空间名称、表空间的存放位置、大小和表空间的可扩展性
- 日志文件数
- 日志文件名、日志文件大小和日志文件的存放位置
- 系统数据和BLOB文件名、大小和存放位置
- 默认的用户数据和BLOB文件名、大小、存放位置
- 系统临时文件名和存放位置
- 用户定义的文件名、大小和存放位置
- DBMaster的日志路径
- 备份目录
- 表复制的日志路径
- 用户文件对象

- 允许使用裸盘（只针对UNIX平台）
- 授予客户机/服务器模式
- 数据库的IP地址和端口号（针对客户机\服务器架构模式的数据库）
- 默认的用户ID和密码
- 存储单元的分配

DBMaster为创建数据库提供了一个容易使用的JServer Manager向导工具。数据库管理员可以通过编辑配置文件**dmconfig.ini**和使用dmSQL的方式来创建数据库。以下章节粗略地介绍了创建数据库的过程，这些章节也大致遵循使用JServer Manager创建数据库的步骤。

数据库的命名

在命名数据库之前，请注意数据库的命名规则：

- 数据库名称最多为128个字符
- 数据库名称可以包括任意的字母字符、数字字符和下划线
- 字符可以出现在任一位置
- 数据库名称不区分大小写
- 数据库名称必须唯一

数据库可以通过JServer Manager的创建数据库向导或使用dmSQL来命名。

➲ **示例**

使用dmSQL创建数据库：

```
dmSQL> CREATE DB <database name>;

dmSQL> TERMINATE DB;

dmSQL> QUIT;
```

设置对象名称的大小写敏感度

您可以指定对象名称的敏感度。在不敏感模式下，所有标识符都是以大写字母来定义的。一旦数据库创建好后，此设置就不能被更改。将关键字设为0表示数据库区分大小写。此关键字的默认值为1，所以数据库创建时会自动设置为不敏感模式。在**dmconfig.ini**文件中可使用如下参数来指定数据库的敏感度：

DB_IDCap = <*值*> （默认值 **= *1***）

存储参数的设置

一个数据库有10种不同类型的操作系统文件，它们分别是：系统数据文件和系统BLOB文件、默认的用户数据文件和用户BLOB文件、系统日志文件、系统临时文件、用户定义文件、DBMaster日志文件、备份文件和表复制日志文件。当首次创建数据库时，用户可以为每个文件指定文件名称和路径，DBMaster也可以为它们分配默认值。在创建数据库之前，您最好对这些文件在数据库中扮演的角色有一个总的认识。

本章讨论的大多数参数都可以在JConfiguration Tool的存储页签中直接更改。要想获得如何使用JConfiguration Tool来更改数据库参数的信息，请查看*配置管理工具用户手册*。 有关更多管理文件的信息，请查看第5.2章*文件类型*。

创建数据库时，DBMaster 会依据**dmconfig.ini**中的参数来创建系统数据库文件、日志文件和系统BLOB文件。如果您没有设定**DB_DbFil**、**DB_JnFil**和**DB_BbFil**参数，那么它们将使用默认值。

以上三个参数的默认值为：

DB_DbFil -- 数据库名称 + '.SDB'

DB_JnFil -- 数据库名称 + '.JNL'

DB_BbFil – 数据库名称 + '.SBB'

指定数据库的目录

数据库目录是指数据库创建和存储时，相关文件的默认存放位置。如果定义的文件被赋予一个完整路径名称，DBMaster将使用这个路径名来访问它；如果文件名缺少完整路径，DBMaster将首先搜索数据库目录。如果没有找到，DBMaster将使用此文件名并且假定它位于当前目录中。

在Windows环境下创建数据库时，DBMaster会为数据库分配一个默认的目录（DBMaster安装目录/bin）来存放数据库文件，为了安全起见，您最好为数据库创建一个新目录，因为多个数据库不应该创建在同一个数据库目录下。在**dmconfig.ini**文件中可使用如下参数来指定数据库目录：

DB_DbDir = <*路径名*> （默认值：<*DBMaster安装目录*>*bin*\<*数据库名称*>）

➲ **示例**

设置数据库目录为：***/disk1/db***

```
[DB1]
DB_DbDir = /disk1/db
```

创建系统表空间

一个DBMaster数据库都可以被分割成数个较小的逻辑空间，我们称这种逻辑空间为表空间。也就是说，表空间把数据库分割成几个方便管理的存取空间。在逻辑视图中， 我们可以看到表空间包含一个或多个表和索引；在物理视图中，表空间是由一个或数个文件组成的物理存储空间。每当您新建了一个数据库时，DBMaster都会自动产生一个系统表空间和一个默认用户表空间。

系统表空间由系统数据文件和系统BLOB文件组成。系统表空间用于记录整个数据库的系统目录，数据库管理员可以指定系统表空间中，系统数据和BLOB文件的初始位置。

系统表空间不可以被删除，但是其它用户表空间是可以被删除的。系统数据库文件的初始大小为200页（200 × **DB_PgSiz** KB）。在**dmconfig.ini**文件中可使用如下参数来定义系统表空间类型：

系统数据文件：**DB_DbFil = <***文件名***>**（默认值：**<***数据库名称***>.*SDB***）

系统BLOB文件：**DB_BbFil = <***文件名***>**（默认值：**<***数据库名称***>.*SBB***）

<filename>参数可以是一个简单的文件名，如**firstdb.sdb**，或者是一个相对路径，如***mydb/firstdb.sdb***，也可以是一个完整路径，如***/disk1/mydb/firstdb.sdb***（“/”是UNIX的分隔符，“\”是Microsoft Windows的分隔符）。

➲ **示例**

在**dmconfig.ini**配置文件中输入如下命令，将系统表空间文件存储于***/disk1/mydb/***目录下。

```
DB_DbFil = /disk1/mydb/firstdb.sdb
DB_BbFil = /disk1/mydb/firstdb.sbb
```

创建用户默认表空间

默认的用户表空间初始包含一个数据文件和一个BLOB文件，用户数据都存储在这些文件中。数据库管理员可以指定用户默认表空间中，数据文件和BLOB文件的初始大小和存放位置。存放数据文件的单位为页（1页可以为 4KB、8KB、16KB 或 32KB），而存放BLOB文件的单位为帧。用户可以自由定义帧大小，有关如何定义帧大小的内容，我们会在稍后章节*如何设定BLOB帧大小*中进行介绍。默认状态下，用户默认表空间是自动扩展的。也就是说，如果表空间的数据已满，DBMaster会自动扩展文件，随之表空间的大小也得以扩展。但是通过创建其它表空间来存储用户表会使数据库系统更加灵活和高效。

JDBA工具可帮助使用者创建和管理表空间。在向表空间中添加数据文件和BLOB文件时，如果没有指定它们的完整路径，那么文件会被创建在数据库目录中。虽然用户表空间可以被删除，但是系统表空间是不能被删除的。您可以在**dmconfig.ini**文件中使用如下参数来指定默认用户数据文件和用户BLOB文件：

用户数据文件：**DB_UsrDb = <***文件名***>**（默认值：**<***数据库名称***>.*DB***）

用户BLOB文件：**DB_UsrBb** = <*文件名*>（默认值：<*数据库名称*>.*BB*）

<filename>参数可以是一个简单的文件名，如**firstdb.sdb**，或者是一个相对路径，如***mydb/firstdb.sdb***，也可以是一个完整路径，如***/disk1/mydb/firstdb.sdb***（“/”是UNIX的分隔符，“\”是Microsoft Windows的分隔符）。

创建日志文件

日志文件对数据库的更改提供了一个实时的历史记录。我们最多可以创建8个日志文件，并且每一个日志文件的大小都是固定的。当所有的日志文件都已填满时（例如事务没有提交，且它们拥有的日志块无法释放），由于没有额外的空间可用，所以当前的事务将被终止，我们把这种状况称作日志满。要确信最长的事务不会将日志文件中的所有记录占用完。如果日志文件没有指定完整路径，那么它们将被创建在数据库目录中。在数据库启动后，日志文件就不能被更改了。如果我们需要对日志文件作增减或更改日志文件的大小，那我们需要以一个新的日志模式来启动数据库。有关新日志模式的信息，请参考4.3章节*启动数据库*。同时，您也可以参考5.2章节*文件类型*以获取更多的日志文件信息。在**dmconfig.ini**文件中可使用如下参数来指定日志文件名、文件位置和大小：

日志文件名称：**DB_JnFil** = <*文件名*>（默认值：<*数据库名称*>.*JNL*）

日志文件大小（页）：**DB_JnlSz** = <*size*> 其中*size* = 100pages ~ 8G

➲ **示例**

在dmconfig.ini配置文件中输入如下命令，DBMaster将在两个不同的磁盘下的***/mydb***目录中创建两个大小为500页的日志文件：

```
DB_JnFil = /disk1/mydb/firstdb1.jnl/disk2/mydb/firstdb2.jnl
DB_JnlSz = 500
```

创建系统临时文件

DBMaster的系统临时文件用于存储数据库运行时的信息，如排序结果，这些文件会在需要的时候自动产生并在关闭数据库时自动被删除。如果

没有指定临时文件的完整路径，那么它们将被创建在数据库目录中。您最多可以指定8个系统临时文件，每一个临时文件最大为2G字节。为了提高磁盘的I/O性能，临时文件可以存放在不同的磁盘上。用户应该为整个临时文件（单个文件的大小为2GB）保留足够的磁盘空间，否则将返回错误信息。在启动数据库之前，您可以使用JConfiguration工具或通过编辑**dmconfig.ini**的方式在启动数据库前来指定一个系统临时文件。在**dmconfig.ini**文件中可使用如下参数来指定系统临时文件的名称和存储位置：

DB_TpFil = <*文件名*>[<*filename*>…]（默认值：<*数据库名称*>*.TMP*）

指定BLOB帧大小

帧是BLOB文件的最小存储单位，它用于存储大的文件对象数据，如LONG VARCHAR或LONG VARBINARY。数据库创建后，BLOB的帧大小就不能改变了。帧大小的取值范围为8 KB—256 KB，一个BLOB文件的帧大小是由磁盘空间的利用程度和操作性能之间的权衡来决定的。如果用户要经常检索BLOB，建议您最好将帧大小调整到能包含整个BLOB文件，这样在操作的过程中，您每次只需访问一次磁盘，从而提高了操作性能。如果我们将帧以最大的 BLOB 数据来划分，那么在其它包含较小 BLOB数据的帧中会存在一些未被使用的磁盘空间，这样就浪费了磁盘存储空间；如果我们将帧以最小的BLOB数据来划分，那么在访问大的BLOB 数据时，会降低它的操作性能。您可以在dmconfig.ini文件中使用如下参数来指定BLOB帧大小：

DB_BFrSz = <*nk*>。此<*nk*>参数表示帧大小，单位是千字节。系统BLOB 文件的大小为（Page size + （帧数– 1） × nk）。要想获得更多详细信息，请参考第7章*大型对象管理*。

➲ **示例**

将BLOB帧大小设为10KB：

```
DB_BFrSz = 10
```

设置表空间自动扩展的页数

当自动扩展表空间的数据文件或BLOB文件的页数已满时，DBMaster会自动扩展文件中的页数或帧数以适应表空间的增长。当有写入操作时，该参数可为已经写满的文件设置增加的页数或帧数。如果数据库管理员期望数据库能够快速的成长，那么您应该选择一个较大的表空间扩展数，从而来减少文件的追加次数。在启动数据库之前，您可以通过使用JConfiguration工具或编辑**dmconfig.ini**配置文件的方式来更改此设置。您可以在**dmconfig.ini**文件中使用如下参数来指定自动扩展表空间中页/帧的数量：

DB_ExtNp = <*np*>。此<*np*>关键字指扩展的页数 （默认值：*20 页/帧*）。

启动用户文件对象

FILE类型的数据可以作为用户文件对象或系统文件对象来存储。 用户文件对象是外部文件，可以通过数据库来访问。换句话说，用户文件对象只是一个外部文件的链接。启动用户文件对象将允许FILE类型的字段链接一个外部文件，它可以通过数据库服务器来访问。当然您也可以根据自己的需要来启动或禁止文件对象。即使设置被关闭，您也可以访问到已经插入的用户文件对象。在启动数据库之前，您可以通过JConfiguration工具的存储页签来设置允许使用用户文件对象，同时此参数是可以在启动数据库之前被更改的。将关键字设为 0，可以禁止插入用户文件对象。将关键字设为1，表示允许插入用户文件对象。您可以在**dmconfig.ini**文件中使用如下参数来指定是否允许插入用户文件对象：

DB_UsrFo = <*值*> （默认值：*0/禁止*）

创建系统文件对象目录

系统文件对象可以通过DBMaster来创建、删除和管理。所有系统文件对象都存放于系统文件对象的子目录中，更改系统文件对象的目录不会影响先前插入的系统文件对象的存放位置。在启动数据库之前，您可以通过JConfiguration工具来设置系统文件对象的名称和存放位置，或者在数据库的运行期间，通过JServer Manager或JDBA Run Time来设置。您可

以在**dmconfig.ini**文件中使用如下参数来设置系统文件对象的存放位置：

DB_FoDir = <*路径名***>** （默认值：\<*数据库目录*>***fo***）

创建用户自定义函数的DLL文件目录

数据库管理员可以指定用户自定义函数（UDF）的动态链接库（DLL）目录。 UDFs是编译后存储于动态链接库中的函数（.DLL用于Windows操作系统，.SO用于UNIX操作系统），这些DLL存储于为用户自定义函数而指定的文件目录中，可以使用SQL语句或ODBC应用程序来访问DBMaster支持的这些DLL。数据库在启动的时候就载入了UDFs，以下关键字用于指定UDF DLL文件的目录：

DB_LbDir = <*文件名***>** （默认值：*当前工作目录*）

开启日志系统

DBMaster提供了日志系统功能，该功能用来记录连接信息、用户信息、执行时间和SQL命令。还可以记录其它数据库信息，用来解决动态运行环境中产生的错误。

日志文件的格式为一种文本***CSV***格式，当用户把***.log***文件重命名为***.csv***文件时，可以使用excel工具来查看该文件，下表列出了log文件中的所有字段，并对每个字段进行了简要的描述。

字段名	描述
LOG_TIME	写日志的时间
BEG_TIME	命令执行的开始时间
STATE	共有四种状态：_、O、X、S，系统会依据“_、O、X、S”来判断是否为未知、正确、错误或减慢，检查序列会先检查rc，然后再检查执行时间
RETCODE	返回代码：0或错误代码
EXE_TIME	执行时间
SV_FUNC	当前执行的服务器状态

CONNECT_ID	连接ID
USERNAME	用户名称
LOGIN_TIME	登录时间
LOGIN_ADDR	登录IP地址
STMT_ID	声明ID
NUM_STMT	语句数量
ERROR_ARG	错误参数
OTHER_INFO	如果打开其它LGXXX设置（如：LgPln），这些信息将被记录在LOGNAME.TXT文件中，用户可以标记[INFO_XXX]为.TXT文件来检查
SQL_CMD	最近一次执行的SQL命令

要启动该功能，您可以在数据库启动前设置配置文件**dmconfig.ini**中的关键字**DB_LgSvr**，也可以在数据库运行中调用存储过程**SETSYSTEMOPTION()**。

日志系统记录的日志信息是分级别的。每一个级别中记录什么操作信息以及何时记录都是不同的。当日志系统开启后，DBMaster通过设置的选项来记录服务器端的操作，并将记录的信息存放到由关键字**DB_LgDir**指定的路径中。数据库服务器根据数据库名称和日志索引数来指定日志文件名称。

由于日志服务器在日志文件名中包含当前日期，因此日志文件名不能重复而且不能被重写。用户可以指定日志文件保存的天数，过期的日志文件将被后台程序删除。该设置可由**dmconfig.ini**文件中的**DB_LgDay**关键字来指定。但是日志文件的数量会增多，打包或压缩早期已经关闭的日志文件将可以节省一些存储空间。该项设置由**dmconfig.ini**文件中的关键字**DB_LgZip**来确定。第一次启动日志系统后，一些系统信息将被记录在文件DBNAME.LOG中，而日志信息将被记录在文件DBNAME_currentdate_1.LOG中。当日志文件的大小达到默认的100MB或由关键字**DB_LgFSz**指定的值时，日志信息将被记录到下一个日志文件DBNAME_currentdate_2.LOG中，依此类推，直到记录到第n个文

件。如，DBNAME_20080706_2.LOG、DBNAME_20080708_3.LOG、……、DBNAME_currentdate _n.LOG；其中n的值，如果没有由关键字**DB_LgDay**和**DB_LgFNo**特别指定的话，则默认为20。

如果用户开启了**DB_LgPln**、**DB_LgPar**或**DB_LgLck**、**DB_LgDay**、**DB_LgZip**功能，或者文件DMERROR.LOG中包含信息时，系统将产生附加信息，这些附加信息将被记录在文本文件DBNAME_ currentdate_n.TXT中。关于文件的默认大小以及滚动记录的特性，文本文件DBNAME_ currentdate_n.TXT同上文描述的日志文件DBNAME_ currentdate_n.LOG一样。这些附加信息除了记录在文本文件DBNAME_ currentdate_n.TXT中，还会以INFO_connection_id_number 的形式而被记录在日志文件DBNAME_ currentdate_n.LOG的OTHER_INFO字段中。这样，用户就可以通过connection_id_number来追踪这些附加信息的来源。请注意，文件 DBNAME_ currentdate_n.LOG 和 DBNAME_ currentdate_n.TXT的文件名是保持同步的，所以信息总是被记录到具有相等n值的各自文件中。这是通过对文件DBNAME_ currentdate_n.LOG 和 DBNAME_n.TXT的大小进行求和，当和的大小达到所设定的日志文件最大值时，就认为两个文件都已满，接下来的信息就立刻被记录到下一个文件DBNAME_ currentdate_n + 1.LOG和DBNAME_ currentdate_n + 1.TXT中。

如果开启**DB_LgSys**，系统信息也将被记录。

可以通过下面的关键字来对日志系统进行设置：

DB_LgDir = <*字符串*> （默认：DBDIR/lgdir）

DB_LgSvr = <*值*> （默认值 = 0/ 禁止）

DB_LgErr = <*值*> （默认值 = 3）

DB_LgSTm = <*值*> （默认值 = 5秒）

DB_LgFSz = <*值*> （默认值 =100MB）

DB_LgFNo = <*值*> （默认值 = 20）

DB_LgPln = <*值*> （默认值 = 0 / 禁止）

DB_LgSys = <*值*> （默认值 = 0）

DB_LgSql = <*值*> （默认值 = 0 / 禁止）

DB_LgPar = <*值*> （默认值 = 0 / 禁止）

DB_LgLck = <*值*> （默认值 = 0 / 禁止）

DB_LgDay=<*值*> （默认值 = 30）

DB_LgZip= <*值*> （默认值 = 1 / 允许）

➲ **示例**

记录一个查询时间 > 10 秒的慢操作以及代号 > 10000的错误，并且保留5天的日志文件并打包已经关闭的日志文件。我们可以在数据库启动前如下配置**dmconfig.ini**文件：

```
[DBNAME]
.........
DB_LgSvr=3;
DB_LgErr=2;
DB_LgSTm=10;
DB_LgDay=5;
DB_LgZip=1;
```

或者通过调用存储过程SETSYSTEMOPTION来达到同样的目的：

```
dmSQL> CALL SETSYSTEMOPTION('LGSVR', '3');
dmSQL> CALL SETSYSTEMOPTION('LGERR', '2');
dmSQL> CALL SETSYSTEMOPTION('LGSTM', '10');
dmSQL> CALL SETSYSTEMOPTION('LGDAY', '5');
dmSQL> CALL SETSYSTEMOPTION('LGZIP', '1');
```

日志系统用于服务器端，所以客户端和网络上的错误不会被记录。开启日志系统将会影响服务器的性能，尤其当日志级别比较高时，所以用户必须确保有足够的硬盘空间来存储服务器日志，否则当硬盘空间满后，日志信息将会丢失。

裸设备

DBMaster的物理存储系统非常灵活，DBMaster允许使用者在UNIX系统上利用一般文件、裸设备文件或两者结合来创建数据库。在**dmconfig.ini**配置文件中，如果文件名以***/dev/***开头，那么此文件被视为裸设备。

因为UNIX系统文件的I/O操作比裸设备慢，所以我们鼓励数据库管理员使用裸设备来建立数据库文件。要想使用裸设备来建立数据库文件，系统管理员必须在创建数据库之前先创建裸设备。请参考UNIX系统手册来创建裸设备。

您无需对裸设备进行分区就可以将多个文件存放在同一个裸设备上，为了实现这项功能，您必须遵守以下约定：

- 当您在单个裸设备上创建多个文件时，文件不能设置成自动扩展类型。
- 在裸设备上创建多个文件后，文件的大小不能被更改。
- 单个裸设备上最多能存放8TB的文件。
- 如果在裸设备上存放着自动扩展文件，那么此设备上将不能再存放其它文件。除了你设置为自动扩展的文件，**DB_DbFil**、**DB_BbFil**、**DB_UsrDb**、**DB_UsrBb**和**DB_TpFil**文件都是自动扩展文件。
- 如果将**DB_DbFil**、**DB_BbFil**、**DB_UsrDb**、**DB_UsrBb**和**DB_TpFil**创建在裸设备上，那么它们只能附有一个参数，即页数。您无法为它们设置偏移量。因为这些文件是自动扩展文件，只能独占一个裸设备，不能同其它文件共享一个裸设备，所以不需要设置偏移量。例如：该语句是有效的，它将创建一个500页的文件。

```
DB_DbFil = /dev/sda 500; Creating a file with 500 pages
```

该语句也是有效的，但是它创建的是一个30页的文件，参数500将被忽略。

```
DB_BbFil = /dev/sdb 30 500;
```

注意　*Microsoft Windows不支持裸设备。*

➲ **示例1**

```
[MYDB]
f1 = /dev/sda 0 500
f2 = /dev/sda 500 200
f3 = /dev/sdb 300
```

创建一个固定表空间**ts_raw**，其中包含以上裸设备文件：

```
dmSQL>CREATE TABLESPACE ts_ras DATAFILE f1, f2, f3
TYPE=DATA;
```

然后在裸设备中创建三个文件，假如选择数据页大小为4K，第一个文件的大小为500 × 4K = 2000K，在设备*/dev/sda*下的起始位置为0；第二个文件的大小为200 × 4K = 800K，在设备*/dev/sda*下的起始位置为500 × 4K = 2000K；第三个文件的大小为300 × 4K =1200K，在设备*/dev/sdb*下的起始位置为0。

➲ **示例2**

```
[MYDB2]
DB_JnlSz = 1000
DB_JnFil = dev/sda/j1.jnl 1000 /dev/sda/j2.inl 2000
```

同样假如您设定数据页为4K，您也可以创建两个普通的日志文件J1.jnl、J2.jnl 和两个裸设备日志文件，一个裸设备文件的起始位置为设备*/dev/sda*下的4,000K，另一个为设备*/dev/sda*下的8,000K。

启动客户机/服务器数据库

任何数据库都可以以单用户模式或多用户模式来启动。在创建数据库之前，确定数据库的主要功能和服务模式是非常有必要的。如果数据库是多用户模式，您必须给数据库分配一个IP地址或DNS名，这样才能进行网络互联。同样，您也可以通过指定TCP/IP的端口号来启动数据库服务器，这时客户端工具会利用这些设置来连接服务器端的数据库。在启动数据库之前，您可以对这些参数进行更改，建议您最好在创建数据库之前来设置这些参数。客户机不会连接一个配置错误的服务器，如果这些设置都被禁止，那么数据库将以单用户模式来启动。这些参数可以从JConfiguration工具的连接页面来更改，也可以通过编辑**dmconfig.ini**文件中的如下参数来更改：

IP地址/服务器名：**DB_SvAdr = <*IP地址*>** 或 **<*主机名*>** （默认值：本地主机名或127.0.0.1）

端口号：**DB_PtNum = <*端口号*>** （默认值：2,300，取值范围为1,024~65,535）

默认的用户名和密码

默认的用户名和密码必须在数据库中事先定义。当启动数据库时，这两个参数无需校验，但在连接数据库时，系统会对他们进行审核。

➲ **示例**

在连接数据库时，用户可以指定默认的用户名和密码：

```
DB_UsrId = <user name>
DB_PasWd = <*****>
```

更改语言字符集

通过设定关键字**DB_WsorT**，DBMaster可以为数据提供多种文字排列指令。例如，关键字**DB_WsorT**可以设定排列指令的大小写敏感。默认的排列次序是二进制排序。

DBMaster支持不同的字符集（语言代码），例如英文US-ASCII、繁体中文BIG5、简体中文GBK 和日文JIS。**dmconfig.ini**配置文件中的关键字**DB_LCode**就是用来指定DBMaster字符集的。针对每一种语言版本，DBMaster都会有几种不同的排列次序。

我们以繁体中文为例，排序指令可以依据代码序列、笔画数或拼音来指定。DBMaster默认的排序命令使用二进制序列，当创建一个数据库时，关键字**DB_Order**可以更改排序指令的行为。语言代码参数也可以通过JConfiguration工具的创建数据库页面来设置。

设置排序指令

排序指令是一组规则，该规则用来指定DBMaster在响应数据库查询和包含GROUP BY、ORDER BY以及DISTINCT子句的语句时如何提取数

据。同时，排序指令也决定如何处理一些特定的查询，如包含WHERE和DISTINCT子句的语句。

您可以通过指定关键字**DB_WsorT**来设置文字排序指令。当指定了大小写不敏感排序指令，DBMaster便会认为大小写字母的值相同的（如，'John' = 'john'。使用大小写不敏感排序指令而使获得的查询结果中既有大写字母也有小写字母有时是非常必要的。

下面的**dmconfig.ini**变量用来指定排序指令的大小写敏感度：

DB_WsorT = <*值*>（默认： 0 / 二进制排序）

1 代表大小写不敏感排序指令

2 代表大小写敏感排序指令

下例说明了在创建数据库之前，如何设置本地语言和排序文件。

➲ **示例**

将语言类型设置为繁体中文BIG5：

```
[MY_DB]
……
DB_LCode =1                ; BIG5 for traditional Chinese
DB_Order = big5_stroke.ord ; order definition file
……
```

关键字**DB_Order**指示了用户自定义的排序文件名为**big5_stroke.ord**，它位于DBMaster的***shared/codeorder***子目录下。此文件是一个纯文本文件，它会影响DBMaster的排序结果。关键字**DB_Order**只有在数据库创建时才会使用， 此后它将一直存在于数据库中。如果缺少这个关键字，在用户创建数据库时，排序序列会默认成二进制序列。用户一旦指定了定义文件，它就必须一直存在，否则数据库将启动失败。

用户定义的代码页

自定义代码页是用户自定义的纯文本文件，代码页用于排列有效字符的序列。以下是一个代码页的例子：codename_ordertype.ord，其中codename是指语言代码的名称，ordertype是指排序的类型。 例如：big5_stroke.ord。

➲ **示例**

定义的字符集如下：

```
   Comment: Write information here.

[BEGIN]       // begin to arrange the character sequence

c             // ASCII 0x63
0x62          // Character 'b'
a             // ASCII 0x61

[SINGLE]      // Single-Byte Character Default Order

[DOUBLE]      // Double-Byte Character Default Order

0xA440        // one of Chinese characters

0xA441       // one of Chinese characters

0xA442       // one of Chinese characters
```

[BEGIN]关键字之前的语句都是注释，所有出现在“//”或“/*”之后的语句也都是注释。在**[BEGIN]**关键字后，每一行都代表一个字符。字符定义应该出现在行的首位，并且至少跟随一个空格或一个新行字符。排序定义文件中的字符都是由小到大排列的，从上例可知，字符**c**小于字符**b**，字符**b**小于字符**a**。

如果某些字符无法用文本编辑器来编辑，那么这些字符可能是十六进制的。例如字符**a**可以用**a**来表示也可以用代码值**0x61**来表示，这对一些隐藏字符是很有效的。

排序指令的创建者可以只规定某些字符的排序，而将其它字符设成默认状态，例如二进制。关键字**[SINGLE]**和**[DOUBLE]**可用于设置单字节和双字节，这两个关键字可以不在字符集中定义。如果没有**[SINGLE]**关键字，那么缺少的单字节将在字符集中的所有字符之前出现。如果没有**[DOUBLE]**关键字，那么缺少的双字节将在字符集中的所有字符之后出现。

DBMaster会忽略字符集中的错误，举例来说，如果**[BEGIN]**丢失，DBMaster会使用默认的排序指令。如果同一个字符出现多次，DBMaster会使用第一次出现的字符而将其它字符忽略。在创建一个数据库后，数据库管理员应该仔细检查排序指令的准确性。

在分布式数据库环境下，所有数据库都应该使用同一个排序指令。当移动或复制数据库到其它PC时，请不要忘记复制已定义的排序指令。

数据库通信与控制区域

数据库通信与控制区域（DCCA）是一块储存数据库控制信息和数据的内存。在多用户模式下，数据库通信与控制区域是从共享内存中配置而来，并且用于做进程之间的通信。当启动数据库时，数据库系统会配置数据库通信与控制区域来储存数据库的相关信息。数据库通信与控制区域可以分为三个部份：页缓冲区、日志缓冲区和系统控制区。

和数据库通信与控制区域（DCCA）相关的配置参数有：

- **DB_NBufs=** <*np*> — DBMaster页缓冲区页数，默认值为0（自动配置）。
- **DB_NJnlB=** <*np*> — DBMaster日志缓冲区页数，默认值为64。
- **DB_ScaSz=** <*np*> — 指定DBMaster系统控制区页数，默认值为200。
- **DB_MaxCo=** <*number*> — DBMaster当前可处理的最大事务数。此关键字也可用于创建数据库或当数据库以一个新日志模式启动时，对日志文件进行格式化处理。

数据库通信与控制区域的大小略等于页缓冲区、日志缓冲区和系统控制区三者的大小之和。但此估计值可能会比真正的需求值小，当数据库通信与控制区域的配置不够大时，DBMaster会自动配置足够大的内存而非上述的估计值。

在UNIX系统中，数据库通信与控制区域的大小不能超过多使用者环境中可配置共享内存的大小。因为在这种情形下，数据库通信与控制区域是从共享内存中分配而来的，使用者可以参考UNIX手册来加大系统的共享

内存。通常来说，增大系统共享内存需要重建系统的核心，拥有较大的缓冲区和控制区域会使DBMaster的执行更加顺畅。

有关DCCA页缓冲区、日志缓冲区和系统控制区之间关系得详细信息，请参考第18章*性能调优*。

DCCA 参数也可以从JConfiguration工具的缓冲区或控制页中来获得。要想获得更多信息，请参考*配置管理工具用户手册*。

4.3 启动数据库

DBMaster可以使用多种方式来启动一个数据库，启动数据库的目的在于从操作系统中配置所需要的资源，将之初始化并等待使用者的连接。在启动数据库之前，请检查配置文件中的参数是否设定。 这些参数包括：

- 数据库启动模式
- 启用客户机/服务器数据库
- 数据库的IP地址和端口号（针对客户机/服务器数据库）
- 默认的用户ID和密码
- 内存分配
- 错误报告的方式

用户可以使用dmSQL或JServer Manager来启动数据库。有关如何使用dmSQL来启动数据库的详细信息，请参考以下章节；有关JServer Manager 的信息，请参考*服务器管理工具用户手册*。

启动单用户模式的数据库

使用者在使用单用户模式的数据库时，必须先启动数据库，并在使用后将之关闭。

➲ **示例**

您可以使用dmSQL执行以下指令来启动单用户模式的数据库：

```
dmSQL> START DB <database name> <user name> <password>;

.

< do DML here >

.

dmSQL> TERMINATE DB;
```

注意 *只有拥有DBA权限的用户才能启动数据库，数据库管理员权限的相关信息请参考第9章安全性管理。如果数据库以单用户模式启动，仅有一位使用者能访问此数据库。*

启动客户机/服务器模式的数据库

运行在客户机/服务器模式下的数据库，数据库管理员必须在服务器端先启动数据库，然后不同机器上的客户端才能通过网络连接至数据库。在数据库启动之前，必须先在服务器端配置如下两个关键字。

和单用户模式的数据库相比，启动C/S架构数据库较为复杂。首先我们需要知道服务器的IP地址， 因为并非所有的客户端都和服务器均位于同一台机器，所以网络地址成为网络上区别机器的唯一识别信息。您可以在**dmconfig.ini**文件中设置**DB_SvAdr**关键字来指定服务器的IP地址。

其次要定义网络端口号，服务端程序将会绑定于这个给定的网络端口上（您可以在**dmconfig.ini**文件中设置**DB_PtNum**关键字来定义），并等待客户的连接。所有的客户端程序必须和该端口号连接以便和服务端程序通信。

➲ **示例1**

您可以通过以下指令来设定服务器的IP地址以及服务器和客户机的端口号：

```
DB_SvAdr = <server IP address> (on client side)

DB_PtNum = <port number> (on both server and client sides)
```

➲ **示例2**

您可以使用DmServer在服务器端启动客户机/服务器数据库：

```
UNIX> dmserver <database name>
```

➲ **示例3**

在您输入用户名和密码后，DmServer将启动数据库并等待客户的连接。

```
UNIX> dmserver [-f] [-t port_number] [-u username [-p password]]
```

```
database_name
```

Unix开关变量的描述如下：

- **f —** 在前台运行服务器程序，DmServer通常是运行在后台的。
- **t —** 指定使用的端口号，DBMaster会优先使用此端口号，而非**dmconfig.ini**中指定的端口号。
- **u —** 登录用户名。
- **p —** 指定用户密码。

如果您在命令行中没有指定用户名和密码，DmServer将在**dmconfig.ini**中查找**DB_UsrId**和**DB_PasWd** 这两个参数。如果没有找到，DmServer将会提示用户输入用户名和密码。

启动模式

在启动数据库之前，用户可以通过**DB_SMode**关键字来指定数据库的启动模式。 此**DB_SMode**关键字存在6种不同的值，对应于6种不同的启动模式：

- **1 —**正常启动。如果数据库在最后一次会话时发生崩溃，那么DBMaster会自动执行崩溃恢复，将数据库恢复至稳定状态。
- **2 —**新日志模式。如果在**dmconfig.ini**中设置了新日志的文件名或存放位置，那么数据库应以新日志模式来启动。新日志的文件名和位置也可以通过Jconfiguration工具的存储页面来设置。如果以前的日志文件名还保存着，那么所有旧的日志记录将被清除。如果用户想更改日志文件的大小、增加更多的日志文件或更改日志文件的名称，那么您必须选择此模式来启动数据库。建议您在选择此模式之前先执行一个备份。
- **3 —**备份还原模式。此模式是利用备份文件（包括日志文件）来启动数据库，DBMaster会利用增量备份文件将操作回滚到**DB_RTime**指定的时间点上。如果没有设定时间点或指定的时间晚于最后一次增量备份的时间，**DB_RTime**将恢复为默认值。请参考第15章*数据库恢复、备份和还原*。

- **4** —数据库复制模式。设置为数据库复制的源数据库，将此模式下启动的系统作为主（源）数据库。请参考第17章*数据复制*。
- **5** —数据库复制模式。设置为数据库复制的目标数据库，将此模式下启动的系统作为从数据库。请参考第17章*数据复制*。
- **6** —只读模式。表示将启动只读数据库，当组成数据库的文件只有读取权限，或者您不希望使用者修改数据库中的数据时，可以以只读模式来启动数据库。

启动模式也可以通过JConfiguration的启动数据库页面或使用JServer Manager工具的启动数据库高级设置窗口来设置。

强制启动模式

如果数据库因某些原因而损坏，在您启动数据库时会返回一些错误信息。这时只能运用DBMaster的**强制启动模式**来启动数据库。您可以将**DB_ForcS**关键字设为1来强制启动数据库。要想获取更多详细信息，请参考第15章*数据库恢复，备份和还原*。

Email出错报告系统

DBMaster会将所有错误信息都存储在dmerror.log文件中，除非数据库管理员经常检查这些dmerror.log文件，否则某些错误信息会被忽略掉。为了防止这种现象的发生，DBMaster提供了一种email出错报告系统来确保数据库管理员清楚的了解系统的出错状况。

出错报告系统可以通过JConfiguration中的两个配置参数来激活，也可以在数据库的启动过程中通过JServer Manager来激活。这两个关键字是**DB_ERMRv**和**DB_ERMSv**。您可以使用DB_ERMRv关键字来指定出错报告email的接收地址，使用**DB_ERMSv**来设置SMTP 服务器到路由email的地址。有关如何使用JConfiguration Tool和JServer Manager来设置email出错报告系统的信息，请分别参考*配置管理工具用户手册*和*服务器管理工具用户手册*。

4.4 连接数据库

本章将讨论如何连接一个正在运行的客户机\服务器模式数据库。用户在执行DML操作之前，必须先连接一个数据库。对客户机\服务器模式数据库而言，当用户断开数据库后，此数据库仍然在运行。在数据库关闭之前，用户仍能重新连接到该数据库。

用户可通过一些参数来连接一个客户机\服务器模式数据库，包括端口号、服务器地址、连接时间间隔和锁超时值。这些参数可以通过更改**dmconfig.ini** 中的关键字或JConfiguration工具来设置。

单用户数据库只允许连接单个用户，所以每次使用该数据库时，必须先启动它。 请参考*启动数据库*章节以获取更多信息。

客户机/服务器模式的数据库

DB_SvAdr和**DB_PtNum** 这两个关键字必须先在**dmconfig.ini**文件中设置。 如果**dmconfig.ini**文件同时也定义了**DB_Usrld**和**DB_PasWd**这两个关键字，那么在CONNECT命令中可以忽略***<username>***和***<password>***。

➲ **示例**

使用者可以使用dmsqlc来连接和断开客户机\服务器模式数据库：

```
dmSQL> CONNECT TO <database name> <username> <password>;

.

< do DML here >

.

dmSQL> DISCONNECT;

dmSQL> QUIT;
```

连接超时

在客户机\服务器模式数据库中，有时会因服务器未启动或服务器的IP地址错误等因素导致客户端无法连接至服务器。在这种情形下，用户会等待很长一段时间。使用者可以通过关键字**DB_CTimO**（以秒计）来设定最大的等待时间，其默认值为 5 秒。

锁超时

锁为同一数据库上的多事务处理提供了并发控制，有关更多事务和并发控制的信息，请参考第九章*并发控制*章节。您可以在**dmconfig.ini**文件中设定**DB_LTimO**关键字来指示用户将等待锁的时间（以秒计）。

举例来说，当**DB_LTimO** = 10 时，表示如果用户等待锁超过10秒时，DBMaster会传回一个 “锁超时”的错误信息；将**DB_LTimO**设为 0，表示用户根本不愿意等待锁被释放；将**DB_LTimO**设为 –1，意为关闭此功能，而用户将等待锁定直到被释放为止。每个用户都可以拥有自己的**DB_LTimO**值。

压缩数据

访问数据库中的内容（也就是数据）是引起网络传输的主要因素。在网络传输前将数据压缩可以大大减少实际传输的数据量，从而有效地提高性能。

连接数据库之前通过设置关键字**DB_NetZc**来开启网络压缩功能。该功能在服务器端将传输的数据进行压缩，并在客户端接收到该数据后将其解压缩。

➲ **示例**

连接数据库之前，在**dmconfig.ini**配置文件中设置如下关键字以开启网络压缩功能：

```
[DBNAME]
DB_NetZc = 1;
```

4.5 关闭数据库

当数据库操作完成时，必须关闭数据库。在关闭数据库时，DBMaster会将使用的资源全部释放并交还给操作系统，如 DCCA。如果此时仍有事务存在于数据库引擎中，DBMaster会将之中断。

然而，如果在关闭数据库时仍有事务连接于数据库引擎中，DBMaster仍会关闭数据库但不会删除这些进程。在这种情况下，数据库管理员应该手动删除这些进程。否则在下次启动数据库时，会产生“事务回滚时无法给文件加锁”的错误信息。

因此，数据库管理员（DBA用户）在关闭数据库之前，应该确认所有用户都从数据库中注销。要关闭一个数据库，DBA必须先连接数据库，然后再下达关闭数据库的指令。只有DBA才拥有关闭数据库的权限。

➲ **示例**

用户可以使用dmSQL来关闭单用户或客户机\服务器模式数据库：

```
dmSQL> CONNECT TO <database name> <DBA username> <password>;
dmSQL> TERMINATE DB;
dmSQL> QUIT;
```

5 储存架构

本章将介绍DBMaster的储存架构。DBMaster的储存架构包括逻辑层次和物理层次。

逻辑层次的储存架构是数据在数据库中的组织方式，也就是用户看到的数据库的储存架构。物理层次的储存架构由操作系统文件组成，这些操作系统文件对应于表空间中的信息，这些信息完全通过DBMaster来管理，一般用户是看不见的。

本章也介绍了如何通过表空间和文件来管理数据库的空间配置。

5.1 储存架构

一个DBMaster数据库由一个或数个逻辑分割组成，我们称此逻辑分割为表空间。表空间是DBMaster在逻辑上的主要储存结构。从逻辑上的观点来看，表空间包含一个或多个表和索引（请参看以下图解）；从物理上的观点来看，表空间是由一个或多个操作系统文件组成的数据储存架构（请参看下图5-1和5-2）。

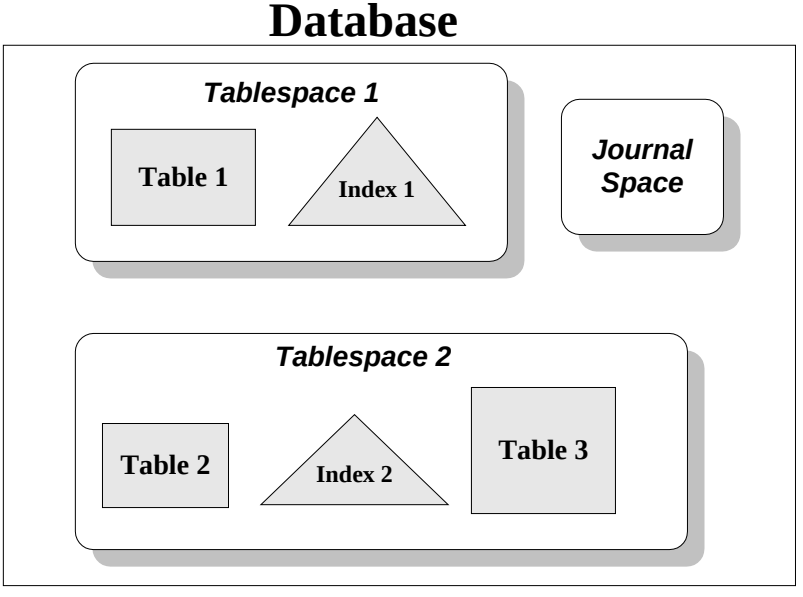


图 5-1 DBMaster储存架构的逻辑视图

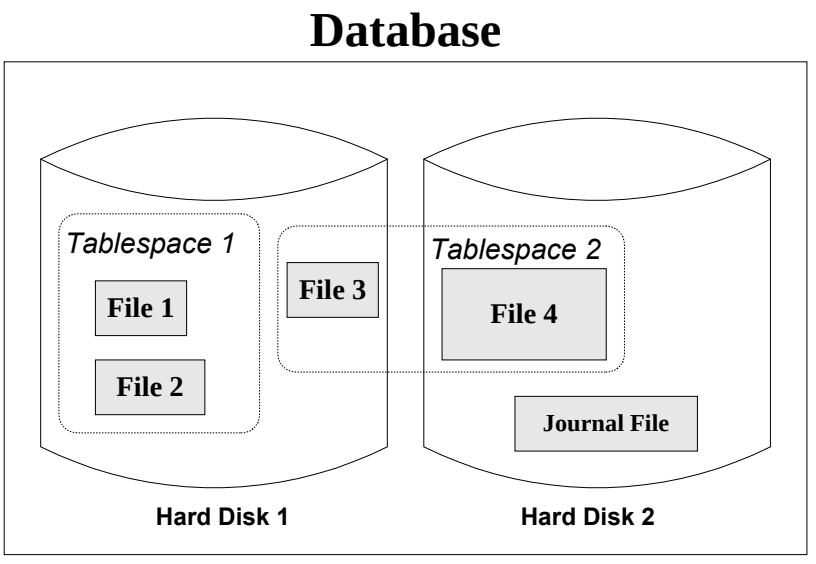


图 5-2 DBMaster储存架构的物理视图

5.2 文件类型

DBMaster中存在10种不同的操作系统文件，这些文件包括：系统数据文件和系统BLOB文件、用户数据文件和用户BLOB文件、系统日志文件、系统临时文件、用户自定义文件、DBMaster日志文件、备份文件和表复制日志文件。其中的系统数据文件、系统BLOB文件、用户数据文件和用户BLOB 文件主要涉及到了数据库的存储架构和表空间。日志文件在存储事务操作的记录上扮演了一个重要的角色，并且是数据库备份和恢复所不能缺少的一部分。

为了提高数据库的操作性能，DBMaster将数据存放于两种不同类型的文件中：数据文件和二进制大型对象文件（BLOB）。BLOB数据可用来存放超过一个数据页的大型数据对象，如影像、声音或是大型文本文件。DBMaster将BLOB数据存放于BLOB文件中，将一般的数据和索引键存放于数据文件中。为了达到高性能的目的，DBMaster在处理这两种类型的数据时采用了不同的做法。

用户数据文件

数据文件由数据页组成，而BLOB文件由帧组成。这两种文件的最大容量皆为8TB，然而数据页和帧有两个主要的区别：

- 一个数据页的大小可以是4KB、8KB、16KB或32KB，可以在创建数据库时通过关键字**DB_PgSiz**来设定。
- 一个数据页可以存放一个以上的数据项，但是一个帧只能包括一个BLOB数据项。

数据文件的最小存储单位是数据页，数据页在存储表或索引时所使用的格式基本相同，一个数据页包括四个部分：页头区、记录数据区、剩余空间和记录目录表。

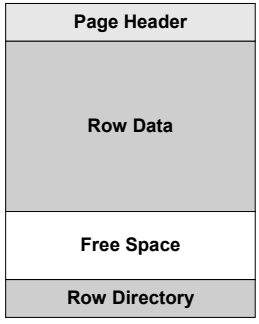


图 5-3 数据页格式

DBMaster系统中的数据页头区存放了一般的数据页信息；记录数据区则以行或字段的方式记录了实际的表或索引数据；记录目录表则记录了每一条记录在数据页里的位置；剩余空间指的是数据页里尚未存放数据的可利用空间。

用户BLOB文件

帧是BLOB文件的最小存储单位。帧大小只能在创建数据库之前设定，范围为8KB——256KB。一个BLOB帧包含三个部分：帧标头、BLOB数据区和剩余空间。要想获取更多BLOB文件的信息，请参考第7章*大型对象管理*。

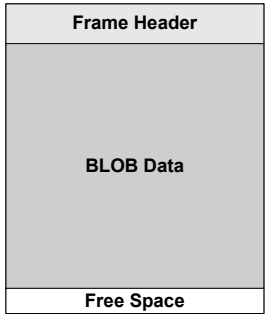


图 5-4 帧格式

正如数据页头区，帧标头也包含了DBMaster系统的一般帧信息；BLOB数据区则存放了BLOB数据，而且每一帧最多只能存放一个BLOB型态的数据。如果BLOB数据的大小大于一帧，那么BLOB数据则可以分放于几个帧中；剩余空间指的是帧中尚未存放BLOB数据的可利用空间。

日志文件

DBMaster的日志由一个或几个物理日志文件组成。逻辑上讲，日志文件由逻辑块组成，每一个逻辑块都是512字节，日志记录记录着数据库的每一个改变。日志记录是日志文件的逻辑元素，一个日志块可以包含几个日志记录，一个日志记录也可以横跨几个日志块。当日志记录被一个激活的事务拥有时，将无法被其它事务重复使用。

所有的日志文件形成了日志记录的循环，日志记录由首尾相接的连续日志块组成。如果数据库配置了多个日志文件，那么在当前日志满时，DBMaster会自动跳转到新日志文件中。否则，日志记录将重写日志文件开始的日志块。当所有日志文件都被活动事务填满后，当前的事务会因缺少可用的日志块而异常中止，我们称这种现象为日志满。

日志文件除了包含日志记录外，还包含一些日志状态块。这些日志状态块是当数据库遭到系统损坏或介质故障时，用于做恢复和还原的。恢复和还原将在稍后的章节描述。

DBMaster配置了日志块缓冲区来加速文件的访问，并且采用日志先行（WAL）原则。当日志缓冲区已满或事务终止时，会将日志缓冲区中的数据写入磁盘。

在DMCONFIG.INI中设置日志参数

用户可以通过一些日志文件参数来提高数据库的操作性能：

- **DB_JnFil —** 日志文件的名称，DBMaster允许您设定1至8个日志文件， 每个日志文件名之间用空白或逗号隔开。

➲ **示例**

这里有7个日志文件，您应该尽量把日志文件放在与数据文件不同的磁盘上以增加系统性能：

```
DB_JnFil = myDb.jn1, myDb.jn2, myDb.jn3, /disk1/usr/myDb.jn4,myDb.jn5,
/disk2/usr/myDb.jn6, myDb.jn7
```

- **DB_JnlSz —** 日志文件的大小，以M、G和页为单位，默认单位为页（页的大小由关键字**DB_PgSiz**来设定），所以 整个日志文件的大小为：

  (DB_JnlSz * DB_PgSiz)KB

在创建数据库时，数据库管理员应该事先估计所需日志空间的大小以便合理的设定日志文件的大小和日志文件的个数。如前所述，当日志空间不足时，会终止目前正在执行的事务。因此，若日志空间太小会导致长事务无法执行。如果数据库涉及到长事务，请选择更大或更多的日志文件。

- **DB_NJnlB —** 日志缓冲区的大小，以页为单位。（页的大小由关键字**DB_PgSiz**来设定）

扩大日志文件空间

在数据库运行期间，如果日志满的信息频繁出现，您可以采取扩大日志文件的方法来提高数据库的操作性能。在DBMaster 3.0版本，先前的备份是无法在调整日志文件大小后将数据库恢复到指定状态的。然而DBMaster3.0之后的版本却可以做到。为了保护数据库不受介质的损坏，您最好在恢复日志文件后立即执行一个完整备份。

➲ **DBA可以执行以下操作来调整日志文件的大小：**

1. 根据需要处理的最大事务来计算所需磁盘空间，从而来估计日志文件的大小和数量。
2. 关闭数据库。
3. 更新**dmconfig.ini**并且重新设定这两个参数：**DB_JnFil**、**DB_JnlSz**。

   注意 *这些选项也可以通过DBMaster的高级设置来更改–JServer Manager启动数据库向导的存储页面。*
4. 在**dmconfig.ini**中，将启动模式设置为新日志模式：**DB_SMode = 2**。

注意 *这些选项也可以通过DBMaster的高级设置来更改 – JServer Manager启动数据库向导的启动数据库页面。*

5. 重新启动数据库。
6. 在**dmconfig.ini**文件中，将启动模式重新设置成正常模式：**DB_SMode = 1**。

   注意 *此选项也可以通过JConfiguration的启动数据库页签来更改。*

7. 如果数据库处于BACKUP-DATA或BACKUP-DATA-AND-BLOB模式，请执行一个在线完整备份。

表空间

DBMaster数据库可以分割成数个较小的逻辑空间，我们称这种逻辑空间为表空间。表空间是存储的逻辑区域，它将数据库分割成几个方便管理的存取空间，每一个表空间都包含一个或数个操作系统文件。建议您在使用DBMaster表空间和文件时，请先熟悉以下几个术语。

表空间类型

表空间可以是固定大小也可以是自动扩展的。如果表空间的大小是固定的，那么这种表空间称为固定表空间（Regular Tablespaces）；如果表空间的大小可以自动扩展，那么这种表空间称为自动扩展表空间（Autoextend Tablespaces）。DBMaster还提供了一种特殊的表空间——系统表空间（System Tablespace）。

系统表空间

所有DBMaster数据库都至少包含两个表空间：一个系统表空间（SYSTABLESPACE）、一个默认表空间（DEFTABLESPACE）。当新建一个数据库时，DBMaster 产生的系统表空间用来存放系统目录表。系统目录表记录了整个数据库的信息。

默认表空间

当用户没有指定表空间时，用户新建的表会自动存储在默认表空间里。但是，创建另外的表空间来存储用户表可增加系统的性能和使用上的弹性。

临时表空间

临时表空间（TMPTABLESPACE）是可自动扩展的表空间，用于存储外部临时表（ETT）。临时表空间中包含两类文件：数据文件和BLOB文件，数据文件的逻辑名称为**DB_TMPDB**，物理名称为**DB_TMPDir**/DBNAME.TDB；BOLB文件的逻辑名称为**DB_TMPBB**，物理名称为**DB_TMPDir**/DBNAME.TBB。

当用户调用“create temporary table”或“select into”语句时，ETT将会自动生成并存储于临时表空间中。用户可以在临时表空间中创建临时表（当然系统将自动把ETT存储在临时表空间中），但不能在临时表空间中创建永久表。用户可以使用命令“ALTER TABLESPACE TMPTABLESPACE SET AUTOEXTEND OFF/ON”和 “ALTER DATAFILE *DB_TMPDB/ DB_TMPBB* ADD *n* PAGES”，但不能将文件添加到临时表空间中或从临时表空间中删除文件。创建数据库时，临时表空间将会自动创建，启动数据库时，临时表空间的大小将被重置为默认大小。

- 用户只能在临时表空间中创建临时表。
- 用户不能在临时表空间中创建永久表。
- 用户不能添加文件到临时表空间或从临时表空间中删除文件。
- 用户不能删除临时表空间。

固定表空间

固定表空间是一个固定大小的表空间，并且包含一个或多个数据文件。如果固定表空间里的文件太小以至于不够存放所有数据，您可以用手动的方式来加大表空间的大小。每个固定表空间里最多可以包含32,767个数据文件，数据库中所有文件的页总数不能超过8TB。

自动扩展表空间

自动扩展表空间是一个视需要可自动延伸的表空间，其中的文件也可以自动延伸，DBMaster 以插入文件的逆顺序来扩展它们。也就是说最后加入的数据文件将被第一个扩展。

您可以将自动扩展表空间更改为固定表空间来保证表空间不被扩展。反之亦然，如果固定表空间的空间已耗尽，您也可以将固定表空间更改为自动扩展表空间。换句话说，您可以用增加文件或扩大现有文件大小的方式来增加固定表空间的大小。 但是请注意：裸设备文件只能用于固定表空间，而不能用于自动扩展表空间。

在创建数据库时，DBMaster会自动创建一个可自动扩展的表空间，我们称之为系统表空间。用户创建表空间时，DBMaster的默认表空间为自动扩展表空间，为了防止默认表空间的无限制增长，您可以将它更改为固定表空间。

dmconfig.ini里定义了数据文件的页数，数据页数代表了自动扩展表空间的初始大小和固定表空间的实际大小。

5.3 表空间和文件的管理

在管理数据库的表空间和文件时需要考虑许多事项。例如：在新建数据库时决定数据库的大小和类型、将自动扩展表空间转换成固定表空间或将固定表空间转换成自动扩展表空间、将数据文件添加到表空间里、定义及变更表空间里的文件大小、 将不再需要的数据文件和表空间删除、将表移动到其它表空间。

您可以通过JDBA工具或结合dmSQL命令行工具和**dmconfig.ini**配置文件来对表空间进行管理。JDBA工具为表空间的管理程序提供了一个直观的用户界面，有关更多如何使用JDBA工具来管理表空间的信息，请参考*数据库管理工具用户手册*。

每一个DBMaster数据库至少包括一个表空间，称为系统表空间。新建一个数据库后，DBMaster会产生五个文件：系统数据文件、用户数据文件、系统BLOB文件、用户BLOB文件和日志文件。系统数据文件、系统BLOB文件和日志文件都存放于系统表空间中，这三个文件用于记录整个数据库的系统目录表；用户数据文件和用户BLOB文件都存放在默认用户表空间里。

如果您不创建其它表空间，那么用户表将储存在默认用户表空间里。建议您将用户表存储在其它的表空间里，这样可以增加系统的性能和灵活性。

系统表空间和文件的初始默认设置

在创建数据库时，DBMaster会产生一个系统表空间和三个系统文件，即系统数据文件、系统BLOB文件、系统日志文件。这些文件记录了数据库的结构和事务信息。DBMaster使用数据库名称加文件扩展名.SDB、.SBB和.JNL来命名系统数据文件、系统BLOB文件和系统日志文件。如果您不对它们的大小进行设置，它们会使用默认的文件大小，即200 × **DB_PgSiz** KB、20KB和4,000KB。如果想自定义文件名称，您

可以通过**dmconfig.ini**配置文件或使用JConfiguration的存储页面来设定它们的文件名。

➲ **示例**

下例说明了如何在**dmconfig.ini**文件中设置系统文件名：

```
[MY_DB]                                    ;database name
DB_DbDir = /disk1/usr                      ;database directory
DB_DbFil = datafile.sdb                    ;data file
DB_BbFil = blobfile.sbb                    ;BLOB file
DB_JnFil = jrnlfile.jnl                    ;journal file
```

利用以上**dmconfig.ini**文件中定义的值，在使用CREATE DB命令创建数据库时，DBMaster 会产生三个系统文件，不过这一次它不会使用文件的默认名称，而是使用**dmconfig.ini**里所定义的文件名。在这种情形下，系统数据文件名为 **datafile.sdb**、系统BLOB文件名为 **blobfile.sbb**、日志文件名为 **jrnlfile.jnl**。

系统表空间的默认状态是可自动扩展的，因此系统表空间的大小只是一个初始值，而非限制值。如果您想限制系统表空间所能使用的磁盘空间，您可以使用ALTER TABLESPACE命令将系统表空间更改成固定表空间。

当固定系统表空间的空间耗尽时，增大空间的唯一方法是在固定系统表空间里添加文件或对现有文件增加数据页，当然您也可以直接将固定表空间更改成自动扩展表空间。

默认的用户表空间和文件的初始设置

在创建数据库时，DBMaster会产生一个默认的用户表空间和两个存储用户数据的文件（用户数据文件和用户BLOB 文件）。这两个文件用来存储用户数据。DBMaster使用数据库名称加文件扩展名（.DB和.BB）来命名用户数据文件和用户BLOB文件。如果您不对它们的大小进行设置，它们会使用默认的文件大小，即200 × **DB_PgSiz** KB和20KB。您可以通过**dmconfig.ini**配置文件或使用Jconfiguration Tool的存储页面来设定它们的名称。

➲ 示例

下例说明了如何在**dmconfig.ini**里设置默认用户文件名：

```
[MY_DB]                                          ;database name
DB_UsrDb = /disk1/usr/f1.db  200                 ;data file
DB_UsrBb = /disk1/usr/f1.bb  20                  ;blob file
```

利用以上**dmconfig.ini**文件中定义的值创建数据库时，DBMaster会产生两个文件，不过这一次它不会使用文件的默认名而是使用dmconfig.ini里所定义的文件名。在这种情形下，默认的数据文件名为**f1.db**，大小为200页；默认的BLOB文件名为**f1.bb**，大小为20帧。

在默认状态下创建的表空间是自动扩展表空间，因此以上表空间的大小只是一个初始大小，而非限制值。

创建表空间

您可以通过dmSQL或JDBA工具来创建表空间，用它们来存放普通数据和BLOB文件。使用dmSQL创建表空间的详细信息，请参考*SQL命令与函数参考手册*；使用JDBA工具创建表空间的详细信息，请参考*数据库管理工具用户手册*。

一个表空间至少包含一个数据文件，但在向表空间中添加文件时，您可以添加数据文件也可以添加BLOB文件。默认状态下，向表空间中添加的文件为数据文件。如果您想添加一个BLOB类型的文件，那么必须在创建它时指定为BLOB类型。

新建一个表空间之前，您必须在**dmconfig.ini**文件中定义将出现在表空间中的文件名称和大小。

➲ 示例 1

下例表示在**dmconfig.ini**文件中必须定义的逻辑文件名**f1**、**f2**和**f3**，以及它们所对应的物理文件名和大小：

```
[MY_DB]                          ;database name
f1 = /disk1/usr/f1.dat 1000      ;a data file with 1000 pages
f2 = /disk2/usr/f2.dat 500       ;a data file with 500 pages
f3 = /disk1/usr/f3.blb 1000      ;a blob file with 1000 frames
```

在**dmconfig.ini**文件中配置好如上参数后，可利用以下命令创建一个固定表空间**ts_reg**，此表空间包含两个数据文件和一个BLOB文件，并且数据文件是分放在不同的磁盘上的：

```
dmSQL> CREATE TABLESPACE ts_reg DATAFILE f1, f2, f3 TYPE=BLOB;
```

➲ **示例 2**

下例可创建一个自动扩展表空间，此表空间包含一个数据文件和一个BLOB文件。数据文件的初始大小为500页，BLOB文件的初始大小为20帧。如果数据文件或BLOB文件已填满，它将会自动扩展：

```
[MY_DB]                          ;database name
f4 = /usr/f4.dat 500             ;a data file with initial 500 pages
f5 = /usr/f5.blb 20              ;a blob file with initial 20 frames
```

在**dmconfig.ini**文件中配置如上参数后，可利用以下命令创建一个自动扩展表空间：

```
dmSQL> CREATE AUTOEXTEND TABLESPACE ts_aut DATAFILE f4 TYPE=DATA, f5
TYPE=BLOB;
```

裸设备文件

在UNIX系统上，如果物理文件名的前缀为**/dev/**，DBMaster会将此文件视为裸设备文件，裸设备文件比普通文件的存取速度快。因此您可以使用裸设备文件来提高数据库的操作性能。您必须先创建一个裸设备文件，才能将它关联到表空间里。只有固定表空间才可以包含裸设备文件。

➲ **示例**

在**dmconfig.ini**文件中配置如下参数，可定义一个裸设备文件**f2**，对应的物理文件名为**/dev/rawf2**，大小为5000个数据页：

```
[MY_DB]                         ;database name
f2 = /dev/rawf2  5000           ;a raw device file with 5000 pages
```

以下SQL指令创建了一个包含以上裸设备文件的固定表空间**ts_raw**：

```
 dmSQL> CREATE TABLESPACE ts_raw DATAFILE f2;
```

扩展固定表空间

您可以通过以下三种方式来扩展一个固定表空间：

- 向固定表空间中添加文件。
- 向固定表空间的文件中添加数据页。
- 设置自动扩展为开启（**ON**）状态。

您可以使用JDBA工具或结合SQL命令并编辑**dmconfig.ini**文件来扩展固定表空间。以下是一个如何通过编辑**dmconfig.ini**文件和使用SQL命令来扩展固定表空间的例子。

➲ **示例**

执行命令前，您可以在**dmconfig.ini**文件的本数据库节中，添加以下语句来设定DBMaster的逻辑文件名**file_blob**和对应的物理文件。在这个例子中，**file_blob**是数据库的逻辑名、**file.blb**是操作系统的物理文件名：

```
file_blob = file.blb 120
```

向固定表空间**ts_app**中添加一个名为**file_blob**的BLOB文件，大小为120帧：

```
dmSQL> ALTER TABLESPACE ts_app ADD DATAFILE file_blob TYPE = BLOB;
```

向**ts_app**固定表空间的**file_data**数据文件中添加100个数据页：

```
dmSQL> ALTER DATAFILE file_data ADD 100 PAGES;
```

文件大小更改后，DBMaster将更新**dmconfig.ini**文件中对应的值，并将启用更改后的数据页数。

均匀自动扩展表空间

您可以通过以下三种方式来扩展一个自动扩展表空间：

- 总是从第一个文件开始扩展。这种方法虽然保持了良好的性能，但是却无法均匀扩展表空间中的所有文件。

- 总是从最小的文件开始扩展。这种方法虽然可以均匀扩展表空间中的所有文件，但是由于表中的所有行都是依次分散到所有文件中的，所以可能会导致性能下降。
- 首先从最小的文件开始扩展，直至该文件的大小超过次小文件与**DB_ExtHd**的值的总和，再开始扩展次小文件。这种方法在保持性能的同时还兼顾了文件大小的平衡。

使用上述的任何一种方法扩展文件时，如果所选的文件由于磁盘满、文件系统限制或DBMaster存储限制等原因而无法扩展时，DBMaster会继续扩展下一个文件。如果下一个文件也因相同原因无法扩展，那么DBMaster会按照顺序再扩展下一个文件，直到所有文件都扩展完毕，详细信息请参考关键字**DB_ExtHd**。

您可以使用配置管理工具或结合SQL命令并编辑**dmconfig.ini**文件来扩展自动扩展表空间。以下是如何通过编辑**dmconfig.ini**文件和使用SQL命令来扩展自动扩展表空间的例子。

➲ **示例1**

使用关键字**DB_ExtNp**和**DB_ExtHd**或调用*SETSYSTEMOPTION('EXTHD','newValue')*来指定自动扩展表空间的扩展范围。DBMaster将根据用户指定的范围，自动的扩展自动扩展表空间。

在**dmconfig.ini**配置文件中指定DBMaster要自动扩展的表空间的大小为**30**页，重复扩展文件的阈值是**100**页：

```
DB_ExtNp=30
DB_ExtHd=100
```

通过修改**dmconfig.ini**配置文件来添加以下5个文件：

```
D1=/home/dbmaster/testdb/D1 10
B1=/home/dbmaster/testdb/B1 30
D2=/home/dbmaster/testdb/D2 50
B2=/home/dbmaster/testdb/B2 70
D3=/home/dbmaster/testdb/D3 100
```

在dmSQL命令行输入如下命令创建一个自动扩展表空间**TS**：

```
dmSQL> CREATE AUTOEXTEND TABLESPACE TS DATAFILE D1 TYPE=DATA, B1 TYPE=BLOB, D2
TYPE=DATA, B2 TYPE=BLOB, D3 TYPE=DATA;
```

在表空间**TS**中创建一个含有字段**(c1 char(5000))**的表**tb_t1**，向该表插入若干行，之后表空间**TS**会自动扩展。根据新的规则，表空间的文件将如下扩展：

```
1st, extend the smallest file D1, add 30 pages. Now, D1=40, D2=50, D3=100.
2nd, extend D1, add 30 pages. Now, D1=70, D2=50, D3=100.
3rd, extend D1, add 30 pages. Now, D1=100, D2=50, D3=100.
4th, extend D1, add 30 pages. Now, D1=130, D2=50, D3=100.
5th, extend D1, add 30 pages. Now, D1=160, D2=50, D3=100.
6th, extend D2, because D1 > D2+EXTHD. Add 30 pages, D1=160, D2=80, D3=100.
7th, extend D2, until D2 > D3(the smallest)+EXTHD, then extend D3.
```

注意 *表tb_t1中没有BLOB文件，因此文件B1和B2无法被扩展。*

➲ **示例2**

数据库运行期间，您可以调用系统存储过程**SETSYSTEMOPTION**来更改系统选项**EXTHD**：

```
dmSQL> CALL SETSYSTEMOPTION('EXTHD','1000'); // changing EXTHD to 10000 pages
```

数据库运行期间，您可以调用系统存储过程**GETSYSTEMOPTION**来显示系统选项**EXTHD**的值：

```
dmSQL> CALL GETSYSTEMOPTION('EXTHD',?);  //reporting the current value of EXTHD
```

向表空间中添加文件

您可以使用新建文件和添加文件的方式来增大固定表空间或自动扩展表空间的大小，您可以在固定表空间或自动扩展表空间里添加数据文件来增加空间的可用性和存放普通数据。当然您也可以增加BLOB类型文件来扩大表空间和存放BLOB数据。用户可通过JDBA工具、编辑**dmconfig.ini**文件和使用dmSQL命令行工具将文件添加到表空间里。以下是通过编辑**dmconfig.ini** 使用dmSQL命令行工具将文件添加到表空间中的原则。

当向表空间中添加数据文件时，请确信向**dmconfig.ini**文件中添加的行指定了新文件的大小和名称。当然您也可以将文件类型设置为BLOB类型来添加BLOB文件，否则DBMaster将创建一个默认的数据文件。

➲ **示例1**

在下例的**dmconfig.ini**文件中，定义了一个文件名为 **f7** 的文件，大小为3,000个数据页，其对应的操作系统文件名为 **/disk1/usr/f7.dat**：

```
[MY_DB]                          ;database name
f7 = /disk1/usr/f7.dat 3000      ;a data file with 3000 pages
```

以下指令将数据文件**f7**添加到表空间**ts_reg**里：

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f7;
```

➲ **示例2**

下例在的**dmconfig.ini**文件中，定义了一个名为 **f8** 的BLOB文件，大小为5,000帧，其对应的操作系统文件名为 **/disk1/usr/f8.blb**：

```
[MY_DB]                          ;database name
f8 = /disk1/usr/f8.blb 5000      ;a blob file with 5000 frames
```

以下指令将BLOB文件添加到表空间**ts_reg**里：

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f8 TYPE=BLOB;
```

请明确指定文件的类型，否则它将以默认的数据文件加入其中。

添加表空间的文件页数

除了在固定表空间中添加文件数来增加表空间的大小外，我们还可以通过改变固定表空间现有文件的大小来扩大表空间。您也可以在自动扩展表空间中添加文件的页数来预留空间以提高系统的性能。当文件的大小被更改时，DBMaster会自动更新**dmconfig.ini**文件相应的参数设置以反映新增的页数。

您可以使用JDBA工具或在dmSQL中输入ALTER DATAFILE命令来更改文件的大小。 以下是在dmSQL中通过输入命令来更改文件大小的原则。

➲ 示例

以下SQL指令说明了如何修改文件大小，此指令为数据文件**f1**增加了100个数据页（文件**f1**必须事先存在，而且属于某个表空间）：

```
dmSQL> ALTER DATAFILE f1 ADD 100 PAGES;
```

将固定表空间更改为自动扩展表空间

当出现以下情况时，数据库管理员会希望将固定表空间更改为自动扩展表空间：

- 向固定表空间中添加更多的数据，但表空间的所有组成文件已被填满，而此时磁盘仍有空间。
- 不希望限制表空间占有的空间数。

在创建一个固定表空间后，数据库管理员可使用JDBA工具或在dmSQL中输入ALTER TABLESPACE命令将它更改为自动扩展表空间。

➲ 示例

您可以通过以下SQL指令将固定表空间**ts_reg**更改为自动扩展表空间：

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND ON;
```

将自动扩展表空间更改为固定表空间

当出现以下情况时，数据库管理员会希望将自动扩展表空间更改为固定表空间：

- 希望限制自动扩展表空间所占据的空间，一个自动扩展表空间可能会占用这个磁盘的所有有效空间。

创建一个自动扩展表空间后，数据库管理员可使用JDBA工具或在dmSQL中输入ALTER TABLESPACE命令将它更改为固定表空间。

➲ 示例

您可以通过以下SQL指令将自动扩展表空间**ts_reg**更改为固定表空间：

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND OFF;
```

压缩表空间和文件

如果要将磁盘空间分配给其他用户，那么就可能要压缩现有表空间的大小。您可以使用SHRINK DATAFILE命令和SHRINK TABLESPACE命令来压缩表空间的大小。SHRINK DATAFILE命令针对用户指定的文件，而SHRINK TABLESPACE命令针对用户指定的表空间中所有文件。这些操作可通过dmSQL或JDBA来实现。以下章节将描述如何使用dmSQL命令行工具来压缩表空间的大小。

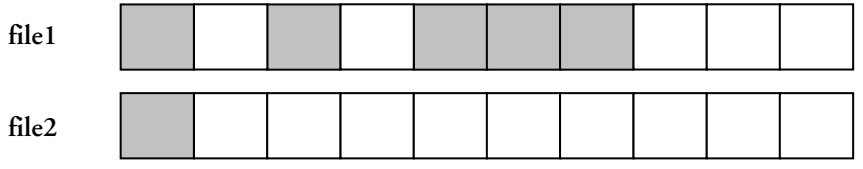
TRUNCATEONLY选项

在数据库运行期间，您可以使用SHRINK命令和TRUNCATEONLY选项来删除数据文件尾的连续空闲页面，但并不压缩文件。如果已经使用的页面中间存在空闲页面，那么它还会继续留在文件中。数据库管理员可通过删除文件尾所有空闲页面的方法（不使用WITH n FREE PAGES选项）或删除一部分空闲页面的方法（使用WITH n FREE PAGES 选项）来压缩表空间，以下是这两个选项的说明：

缺少 WITH *n* FREE PAGES 选项

带有TRUNCATEONLY选项的SHRINK命令（缺少WITH n FREE PAGES选项），只会截短文件尾部的相邻空闲空间。

例如：表空间**ts_shrink**包含文件**file1**和**file2**。下表中的灰色块代表已用的数据页，白色块代表空闲的数据页。



以上两个文件尾部的空闲页可通过SHRINK TABLESPACE命令或SHRINK DATAFILE命令来删除，但是必须设定TRUNCATEONLY选项，您可以通过以下例子来验证这一点。

➲ **示例1**

```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY;
```

➲ **示例2**

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY;
```

数据页截短后，文件尾部的页面都已被删除。下图说明了两个文件数据页截短后的状态。

结果：

尽管**file2**的所有页面都已经释放，但DBMaster至少还占用了两个数据页（一个是PE页面，一个是数据页面）。

WITH *n* FREE PAGES 选项

WITH n FREE PAGES选项指出了数据页经过截短后，文件中还存在的空闲页数（不包括PE页）。

继续使用上一例子中的**file1**和**file2**，执行以下命令。

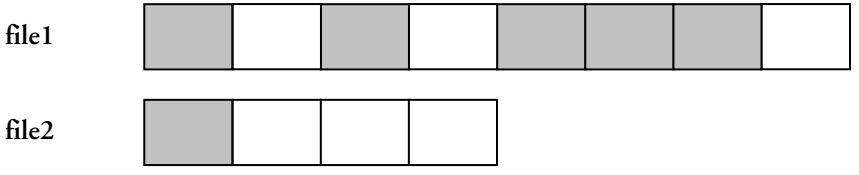
➲ **示例1**

```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY WITH 3 FREE PAGES;
```

➲ **示例2**

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY WITH 3 FREE PAGES;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY WITH 3 FREE PAGES;
```

结果：



SHRINK TABLESPACE命令和WITH FREE PAGES选项分别应用于表空

间中的**file1**和**file2**文件，通过上例可知，在同一个表空间的每个文件中都保留了三个空闲数据页。

如果用户想向文件中添加比当前数据页更多的页面，这几乎是不可能的。例如：如果文件中有50个空闲页面，而我们指定选项为WITH 80 FREE PAGES，那么此操作将不会起任何作用。经过执行SHRINK命令后，仍有50个空闲页面，文件的大小并不会随着多添的30个空闲页面而自动扩大。

SHRINK命令应该在自动提交（autocommit）为开启（ON）的状态下执行。因为TRUNCATEONLY选项无法被回滚，所以用户不能回滚此命令，即使执行崩溃恢复。

COMPRESSONLY选项

只有SHRINK TABLESPACE命令支持COMPRESSONLY选项，它将压缩表空间中的全部文件。因为压缩文件的单位为页，所以使用此选项不会压缩页面中的记录。执行此命令后，所有空闲页面都置于文件尾，而所有使用的页面都置于文件头。

结果 1：

File1有五个不相邻的已用页面。

➲ **示例**

您可以输入以下SQL命令将不相邻的页面移动到一起：

```
dmSQL> SHRINK TABLESPACE ts_shrink COMPRESSONLY;
```

结果 2：

只有在autocommit为开启状态时，SHRINK命令才可以执行，COMPRESSONLY操作可以被回滚。如果数据库发生崩溃，在灾难恢复过程中，COMPRESSONLY操作将会被全部执行或全部撤消。

在执行备份时，SHRINK 命令和COMPRESSONLY 选项会发生冲突。也就是说DBMaster不允许这两个命令同时执行。

压缩表空间的局限性

使用SHRINK命令的一些限制：

- SHRINK命令可用于数据文件和BLOB文件，但不能用于日志文件。
- 只有拥有DBA权限的用户才能执行SHRINK命令。
- 只有将autocommit设为开启状态，才能执行SHRINK命令。
- 只有DBMaster 3.7以后的版本才能执行SHRINK命令，因此一旦DBA执行SHRINK命令和增量备份，DBMaster早期的版本将无法恢复日志备份文件。
- TRUNCATEONLY选项不能被回滚。
- COMPRESSONLY选项不能压缩SYSTABLESPACE系统表空间。
- COMPRESSONLY选项不会检查用户表中是否使用了OID类型的字段，OID字段用于参照数据库中其他表的记录。在使用COMPRESSONLY命令后，如果参照的记录就在压缩的表空间或文件中，那么OID字段将不再指向您想参照记录的正确位置。用户表中的OID类型的字段值不能更改。
- COMPRESSONLY选项和备份命令不能同时执行。

删除表空间

如果表空间里没有任何数据或表空间中的数据已不再需要，您可以将此表空间从数据库中删除。DBMaster中除了系统表空间和系统默认表空间外，任何一个表空间都可以被删除。在删除表空间之前，您必须先将表

空间里的所有表移去或是确定表里没有存放任何数据。若想获得如何从表空间中删除表的信息，请参照第6章*数据库的对象管理*。

删除表空间会将表空间与之相关文件的关联一起删除，但是其对应的操作系统物理文件并不会被删除。这些文件会一直留在文件系统里，您可以用操作系统命令来删除这些文件，回收所占用的磁盘空间。一旦这些原先属于表空间的操作系统文件被删除后，里面存放的数据将无法恢复，所以在删除表空间的文件时要特别小心以免丢失宝贵的资料。

您可以使用JDBA工具或通过dmSQL中的DROP TABLESPACE命令来删除表空间。

➲ **示例**

以下SQL指令将删除表空间**ts_aut**以及所对应的文件：

```
dmSQL> DROP TABLESPACE ts_aut;
```

从表空间中删除文件

用户可以使用JDBA Tool或通过dmSQL中的ALTER TABLESPACE *tablespace-name* DROP DATAFILE *file-name*命令来删除不再需要的数据文件，当选择使用第二种方法时，用户应删除物理数据文件，在提交并执行ALTER TABLESPACE *tablespace-name* DROP DATAFILE命令后，必须手动将配置文件中的信息删除。

被删除的文件必须遵循以下约定：

- 如果该文件是表空间中唯一的一个数据文件，那么用户就不能将该文件从表空间中删除。
- 被删除的数据文件必须为空。
- 用户无法从系统或默认表空间中删除系统或默认的数据文件。

➲ **示例**

从表空间**ts_aut**中删除数据文件**f4**：

```
dmSQL> ALTER TABLESPACE ts_aut DROP DATAFILE f4;
```

只读表空间

只读表空间不允许使用者在表空间中更新或创建任何对象。

将表空间设置成只读模式会带来很多优势：

- 除去执行备份的需要。在将表空间设置成只读模式后，只需对它做一个单独备份。
- 使恢复变得更加容易。
- 只读表空间比可更新的表空间（没有锁）需要较少的消耗。
- 减少I/O操作。

➲ **示例**

将表空间**ts_reg**设置成只读模式。

```
dmSQL> ALTER TABLESPACE ts_reg SET READ ONLY;
```

将表空间**ts_reg**设置成可读写模式。

```
dmSQL> ALTER TABLESPACE ts_reg SET READ WRITE;
```

获取有关表空间和文件的信息

您可以使用JDBA工具直接了当地观察表空间和文件结构。数据库中的所有对象用逻辑树的方式表示，表空间是树的一部分。从逻辑树上选择表空间节点来展开树结构后，将显示数据库中的所有表空间。从展开的逻辑树上选择一个表空间将显示表空间中所包含文件的详细内容，例如表空间大小、表空间的物理位置、所存储数据的类型或表空间是否为自动扩展表空间。

当然，您也可以使用dmSQL从系统表SYSTABLESPACE中选择所有字段以获取表空间的信息，或从SYSFILE表中获得用户BLOB文件和数据文件的信息。

➲ **示例1**

想要获得表空间的信息，如表空间名称、固定表空间还是自动扩展表空间、表空间中相关联的文件数和所占用的数据页数等，您都可以从系统表SYSTABLESPACE中获得。

```
dmSQL> SELECT * FROM SYSTABLESPACE;
```

➲ **示例2**

可使用相同方法从系统表SYSFILE中获得文件的信息，如文件名称、文件类型、数据库内部文件标识符、所属表空间以及每个文件所占用的数据页数。

```
dmSQL> SELECT * FROM SYSFILE;
```

要想获得有关系统目录表SYSTABLESPACE和SYSFILE的信息，请参考第21章*系统目录参考*。

检验表空间和文件的一致性

DBMaster提供了六个命令来检验数据库的一致性。当数据库的容量很大时，这些命令是很费时间的并且会占据一些锁。在不得已的情况下最好不要使用它。您可以使用CHECK FILE命令来检验被损坏的文件和包含正确表的表空间。

检验文件的一致性

DBMaster可以检验数据页或帧的内容，检验出的错误通常是由磁盘故障引起的。

➲ **示例**

举例说明如何检验数据文件**FILE1**的一致性：

```
dmSQL> CHECK FILE FILE1;
```

检验表空间的一致性

同样，DBMaster也可以检验表空间中相关联的文件和表。在检验表空间时，就如同直接使用check file命令和check table命令一样，会返回相同的结果。

➲ **示例**

检验表空间**ts_reg**的一致性：

```
dmSQL> CHECK TABLESPACE ts_reg;
```

6 数据库的对象管理

本章讨论了如何管理DBMaster数据库中不同类型的对象（schema objects），包括表（tables）、视图（views）、同义字（synonyms）、索引（indexes）、序列数（serial numbers）、数据完整性（data integrity）和域（domains）。

本章除了介绍以上对象外， 还介绍了如何利用系统表来获得数据库中各种对象的详细资料以及如何估计建立表和索引所需要的储存空间。

数据库的对象管理可以通过dmSQL命令或JDBA工具来执行，JDBA工具包含一个交互式的图形用户界面，为大多数数据库管理任务提供容易使用的向导，并且清楚的显示了数据库的逻辑结构。使用JDBA工具会使首次使用DBMaster的用户明白对象之间的关系。有经验的用户会在创建和管理数据库计划时获得一些帮助信息。以下章节将举例说明如何通过dmSQL来管理数据库对象，有关如何使用JDBA工具来管理数据库对象的信息，请参考*JDBA Tool用户管理手册*，有关SQL语句的语法和使用，请参考*SQL命令与函数参考手册*。

6.1 模式管理

模式也就是数据库对象的统称（数据库对象的逻辑分类），它包括表、视图、索引、命令、过程、定义域和同义字。

使用CREATE SCHEMA命令可以定义一个新的模式，当模式创建好后，我们就可以在该模式内创建数据库对象了。模式的所有者将成为所有权限的授予者。

模式的所有者存在以下约定：

- 如果指定了AUTHORIZATION子句，那么给出的用户名就代表模式的所有者名；如果省略了模式名，那么给出的用户名就代表模式名。

例如：

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

- 如果没有指定AUTHORIZATION子句，那么创建CREATE SCHEMA语句的用户就成为模式的所有者。

➲ **示例1**

拥有RESOURCE权限的用户JEFFERY创建了一个名为**SCH_JEF**的模式，并且默认为该模式的所有者。

```
dmSQL> CREATE SCHEMA SCH_JEF;
```

➲ **示例2**

拥有DBA权限的用户创建了一个以用户名JEFFERY为所有者名的模式，并且该用户名JEFFERY为默认模式名。

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

➲ **示例3**

拥有DBA权限的用户创建了一个名为**SCH_ForJEF**的模式，并且以用户JEFFERY为该模式的所有者。

```
dmSQL> CREATE SCHEMA SCH_ForJEF AUTHORIZATION JEFFERY;
```

➲ **示例4**

拥有DBA权限的用户创建了一个名为**inventory**的模式，随后又创建了一张表inventory.part，并在该表上创建了一个索引partind，最后给用户JEFFERY授予表的所有权限。

```
dmSQL> CREATE SCHEMA inventory;
dmSQL> CREATE TABLE inventory.part (partNo smallint not null, quantity
int);
dmSQL> CREATE INDEX partind ON inventory.part (partNo);
dmSQL> GRANT ALL ON inventory.part TO JEFFERY;
```

通过DROP SCHEMA命令可以将模式从数据库中删除，模式只能被它的所有者或DBA删除，但是所有者无法删除一个包含对象的模式。

➲ **示例5**

从数据库中删除模式**SCH_JEF**。

```
dmSQL> DROP SCHEMA SCH_JEF;
```

注意 *用户名不能和模式名相同。*

模式信息

DBMaster中的每一个数据库都包含一个模式，称作INFORMATION_SCHEMA。该模式包含一系列视图，允许您查看（但是不能更改）数据库中每一个对象的描述。

DBMaster为获得的元数据提供了一些信息模式视图，这些视图提供了DBMaster元数据的一个内在的、独立的系统表视图。信息模式视图可以保证应用程序的正常操作，即使在系统表发生了重大更改时。DBMaster中的信息模式视图遵循SQL-92标准定义。

DBMaster对当前服务器支持一个三部分的命名约定，标准的SQL-92也支持一个三部分的命名约定，但是这两个命名约束中定义的名称却不相同。这些视图都定义在一个特殊的模式中，名为INFORMATION_SCHEMA，每一个数据库都包含这个模式。每个INFORMATION_SCHEMA视图包含存储在指定数据库中所有数据对象的元数据。下表描述了DBMaster名称和SQL-92标准名称之间的关系。

DBMASTER名称	等价的 SQL-92名称
数据库	目录
所有者	模式
对象	对象
用户定义数据类型	定义域

这个命名约定映射适用于与DBMaster SQL-92兼容的视图。
INFORMAINFORMATION_SCHEMA视图包括以下内容：

- COLUMN_DOMAIN_USAGE
- COLUMN_PRIVILEGES
- COLUMNS
- DOMAINS
- SCHEMATA
- TABLE_PRIVILEGES
- TABLES
- VIEW_COLUMN_USAGE
- VIEW_TABLE_USAGE
- VIEWS

➲ 示例

```
dmSQL> SELECT * FROM INFORMATION_SCHEMA.COLUMNS;
```

6.2 管理表

表是DBMaster中存储数据的逻辑存储单位。一张表由数行（rows）和数字段（columns）组成。其中行被视为记录或元组，字段被视为属性。

DBMaster中的每一张表都可以通过唯一的用户名和表名来识别。例如：如果两个用户**Jeff**和**Kevin**各自创建了一张friend表，那么表**Jeff.friend**和**Kevin.friend**分别代表两张不同的表。

在JDBA工具中可扩展逻辑树的表节点来查看数据库中的所有表，如果想查看某张表的表结构，可以点选该表名称的节点。

创建表

每张表都必须拥有表名称和字段，一张表可包含的字段数为1-2000。

每个字段包括：

- 字段名称和数据类型（或定义域。有关定义域的详细内容将在6.10章*定义域管理中*描述）。
- 字段长度（有些数据的字段长度是默认的，所以并不需要特意指定，如INTEGER）；精度（precision）和等级（scale）（仅针对字段为DECIMAL的数据类型）或数据的起始数字（仅针对字段为SERIAL的数据类型）。

DBMaster支持大量数据类型，包括数值类型：SMALLINT、INTEGER、BIGINT、FLOAT、DOUBLE、DECIMAL、 REAL和SERIAL、BIGSERIAL；二进制类型：BINARY和VARBINARY；字符类型：CHAR、NCHAR、VARCHAR和NVARCHAR；BLOB类型：LONG VARCHAR、LONG VARBINARY和FILE、媒体类型和JSONCOLS；时间类型：DATE、TIME和TIMESTAMP。有关数据类型的详细信息，请参考*SQL命令与函数参考手册*。

在创建表时，您必须提供表名、字段定义和该表所属的表空间。如果没有特别定义表空间，DBMaster会默认使用系统表空间来存放该表。您也

可以使用JDBA工具的创建表向导或使用dmSQL命令行工具来创建一张表。以下是一个如何使用dmSQL创建表的例子。有关SQL命令CREATE TABLE的语法和用法，请在*SQL命令与函数参考手册*中查询。

➲ **示例**

下例说明了如何在表空间**ts_reg**上创建表**tb_staff**：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20),
                                   ID INTEGER,
                                 name CHAR(30),
                             joinDate DATE,
                               height FLOAT,
                               degree VARCHAR(200),
                              picture LONG VARCHAR) IN ts_reg;
```

DBMaster在创建表时，DBMaster会提供很多有效的选项：

- 设定字段默认值（default value）
- 设定字段是否为空
- 指定表的主键或外键
- 设定锁定模式（LOCK MODE）、填充因子（FILLFACTOR）或取消快取（NOCACHE）等选项来提高数据库的存取效率
- 设定临时表

字段默认值

您可以为表中字段设定默认值，当插入一笔新记录而没有输入字段数据时，那么该字段将被自动赋予默认值。

您可以对表中的每一字段设定默认值，如果您没有指定某个字段的默认值，那么该字段会自动预设为空值（NULL）。

合法的字段默认值可以是常数（constants）、NULL或内建函数（built-in functions），要想获得更多有关内建函数的信息请参考*SQL命令与函数参考手册*。

目前，DBMaster支持使用关键字USER、SYSTEM和ON UPDATE来设置插入和更新操作的默认字段属性。关键字USER/SYSTEM是可选项。这些关键字可以指定用户是否能使用INSERT/UPDATE语句来修改带有默认值的字段的值。USER是默认的，关键字USER指定用户可以修改它的值，关键字SYSTEM指定用户不可修改它的值。关键字ON UPDATE也是可选项，该关键字指定更改其他字段的值时，带默认值的字段的值可以自动更新。这三个关键字主要用于表定义，有关更多表定义的信息请参考*SQL命令与函数参考手册*的*CREATE TABLE*、*ALTER TABLE ADD COLUMN*以及 *ALTER TABLE MODIFY COLUMN*章节。

此外，当更新数据时，用户可以使用连接选项SYSTEM DEFAULT来指定带SYSTEM DEFAULT默认属性的字段的值是否被重写为默认值。如果将该选项设置为ON，则值会被更新为默认值；如果设置为OFF，则原始值会被更新为用户指定的值。该选项的默认设置为ON。另外，当使用INSERT/UPDATE语句将值分配给字段时，用户可以使用连接选项LOAD SYSTEM DEFAULT来指定加载数据库中的表时，带SYSTEM DEFAULT属性的字段是否被重写。如果将该选项设置为ON，则值会被更新为默认值；如果设置为OFF，则原始值会被更新为用户指定的值。该选项的默认设置为OFF。

➲ **示例1**

您可以使用以下SQL命令来创建**tb_staff**表，其中字段**nation**的默认值为一个常数— 'R.O.C.'，而字段 **joinDate**的默认值为一个内建函数 **curdate()** 的返回值：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                   ID INTEGER,
                                 name CHAR(30),
                             joinDate DATE DEFAULT CURDATE(),
                               height FLOAT,
                               degree VARCHAR(200),
                              picture LONG VARCHAR) IN ts_reg;
```

➲ **示例2a**

```
dmSQL> CREATE TABLE computer(id INT, buy_time TIMESTAMP DEFAULT '2012-
03-04 12:12:12', price int); //now attributes of buy_time is USER
```

```
dmSQL> INSERT INTO computer VALUES(1, '2012-10-10 10:10:20', 3400);
//value of buy_time will be replaced with '2012-10-10 10:10:20' which is
specified by the user
1 rows inserted
dmSQL> INSERT INTO computer VALUES(2, '2012-10-11 10:10:20', 5400);
1 rows inserted
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== ========================== ===========
          1 2012-10-10 10:10:20                3400
          2 2012-10-11 10:10:20                5400
2 rows selected
dmSQL> UPDATE computer SET price=3200 WHERE id=1; //value of buy_time
will not be updated
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== ========================== ===========
          1 2012-10-10 10:10:20                3200
          2 2012-10-11 10:10:20                5400
2 rows selected
```

➲ **示例2b**

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP
DEFAULT '2012-03-04 12:12:12' ON UPDATE); //now attributes of buy_time
is USER and ON UPDATE
dmSQL> UPDATE computer SET price=3000 WHERE id=1; //value of buy_time
will be replaced with the default value'2012-03-04 12:12:12'
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== ========================== ===========
          1 2012-03-04 12:12:12                3000
          2 2012-10-11 10:10:20                5400
2 rows selected
dmSQL> UPDATE computer SET price=3000, buy_time='2012-10-10' WHERE
id=1;//value of buy_time will be replaced with '2012-10-10' which is
specified by the user
1 rows updated
```

```
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
2 rows selected
```

➲ 示例2c

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP
SYSTEM DEFAULT '2012-03-04 12:12:12'); //now attributes of buy_time is
SYSTEM
dmSQL> INSERT INTO computer VALUES(3, '2012-11-10 10:10:20', 4700);
//value of buy_time will not be replaced with '2012-11-10 10:10:20'
which is specified by the user
1 rows inserted
dmSQL> INSERT INTO computer VALUES(4, '2012-12-11 10:10:20',
2800);//value of buy_time will not be replaced with '2012-12-11
10:10:20' which is specified by the user
1 rows inserted
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
         3 2012-03-04 12:12:12               4700
         4 2012-03-04 12:12:12               2800
4 rows selected
dmSQL> UPDATE computer SET price=4500 WHERE id=3; //value of buy_time
will not be updated
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
         3 2012-03-04 12:12:12               4500
         4 2012-03-04 12:12:12               2800
4 rows selected
```

➲ **示例2d**

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP
SYSTEM DEFAULT '2012-03-04 12:12:12' ON UPDATE); //now attributes of
buy_time is SYSTEM and ON UPDATE
dmSQL> UPDATE computer SET price=4000, buy_time='2015-01-01' WHERE id=3;
//value of buy_time will be replaced with the default value'2012-03-04
12:12:12'
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== ========================== ===========
          1 2012-10-10 00:00:00               3000
          2 2012-10-11 10:10:20               5400
          3 2012-03-04 12:12:12               4000
          4 2012-03-04 12:12:12               2800
4 rows selected
```

字段是否允许为空值

您可以依照每个字段的不同特性来指定字段数据所允许的形式，这些规则也就是所谓的完整性约束。您可以设定某个字段是否允许为空值，如果某个字段值被设为不允许空值（NOT NULL），就代表该字段必须输入有意义的数据，否则将无法保存到数据库中。

例如：在**tb_staff**表中录入新员工资料时，总是需要输入员工编号（ID）和员工姓名。

➲ **示例**

在**tb_staff**表中为新成员创建ID和姓名：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                ID INTEGER NOT NULL,
                              name CHAR(30) NOT NULL,
                          joinDate DATE DEFAULT CURDATE(),
                            height FLOAT,
                            degree VARCHAR(200)) IN ts_reg;
```

主键和外键的设定

用户可以使用CREATE TABLE命令来指定表的主键和外键，请参考第6.9章*数据完整性管理*来获取主键和外键的信息。

锁定模式

访问数据库时，DBMaster会自动识别对象的锁定模式。DBMaster提供了三个层次的锁定模式：表锁定（TABLE）、页锁定（PAGE）和行锁定（ROW）。在创建表时，如果没有指定锁的模式，DBMaster会将表中的数据设定为行锁定（ROW）模式。如果将表设定为较高层次的锁模式 （例如表锁定)，那么数据库的并行存取能力将会降低，但锁定数据所需要的系统资源（共享内存）将会减少；如果将表设定为较低层次的锁定模式（例如行锁定），那么数据库的并行存取能力将会提高，但是锁定所占用的系统资源（共享内存）就会增多。换句话说，如果您想在表锁定模式的表上新增或更改记录，除了您以外将没有其他使用者可以同时存取该表中的数据。这是因为您在更新或新增数据时，数据库将会对整张表作一个互斥的锁定。有关锁模式的详细信息请查看第9.4章*锁*。

➲ **示例**

下例说明了如何在表上设定锁模式：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                   ID INTEGER NOT NULL,
                                 name CHAR(30) NOT NULL,
                             joinDate DATE DEFAULT CURDATE(),
                               height FLOAT,
                               degree VARCHAR(200)) IN ts_reg
                      LOCK MODE ROW;
```

填充因子（FILLFACTOR）

填充因子（FILLFACTOR）是用来指定新增数据时，数据页使用空间的上限（最大比例），利用填充因子可以在数据页上保留一定比例的空间。也就是说当您修改一笔记录时，如果数据页里还有足够的空间来储存这个更改的记录，就不用把这笔记录分割在不同的数据页里了。像这样把记录存放在同一个数据页里的好处在于：无需读取多个数据页就能存取一笔记录，数据存取的效率会有所提高。

➲ 示例

下例是将**tb_staff**表的填充因子设为80%：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                   ID INTEGER NOT NULL,
                                 name CHAR(30) NOT NULL,
                             joinDate DATE DEFAULT CURDATE(),
                               height FLOAT,
                               degree VARCHAR(200)) IN ts_reg
                      LOCK MODE ROW
                      FILLFACTOR 80;
```

在这种情况下如果使用的空间超过80%，那么新增的记录将无法插入到数据页中。填充因子（FILLFACTOR）的取值范围为50%-100%，默认值为100%。

取消快取（NOCACHE）

如果您需要经常读取一个非常大的表时，最好对这个表设定取消快取（NOCACHE）功能。DBMaster在存取数据库时，会在共享内存中分配数据页缓冲区来储存数据以免进行过多的磁盘存取（I/O）。 在对一个相当大的表作扫描时，因为它拥有的数据页数大于数据页缓冲区数，所以会占尽所有的数据页缓冲区。

如果在建立表之前就知道这个表会储存大量的数据，我们可以设置这个表的取消快取功能，以后DBMaster对这个表做扫描时，就只会将一个数据页缓冲区来存放表数据。 这样数据页缓冲区就不会只被一个很大的表所占用。

➲ 示例

设定取消快取（NOCACHE）选项：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                   ID INTEGER NOT NULL,
                                 name CHAR(30) NOT NULL,
                             joinDate DATE DEFAULT CURDATE(),
                               height FLOAT,
                               degree VARCHAR(200)) IN ts_reg

                      LOCK MODE ROW
```

```
                    FILLFACTOR 80
                    NOCACHE;
```

临时表

您可以创建临时表来储存数据。临时表不会永久的储存在数据库中，只能存在于单个连接（session）中，并仅能被创建者使用，当用户断开数据库的连接时，DBMaster会自动将临时表删除，临时表支持快速的数据操作。客户端用户可以使用CREATE LOCAL TEMPORARY TABLE命令创建一个本地临时表。

➲ **示例1**

创建一个名为**tb_student**的临时表：

```
dmSQL> CREATE TEMPORARY TABLE tb_student (name CHAR(25) NOT NULL,
                                          birthday DATE,
                                          score INTEGER);
```

➲ **示例2**

创建一个名为**tb_student**的本地临时表：

```
dmSQL> CREATE LOCAL TEMPORARY TABLE tb_student (name CHAR(25) NOT NULL,
                                                birthday DATE,
                                                score INTEGER);
```

浏览表结构

表结构可通过dmSQL或JDBA工具来查询。JDBA工具提供了一个图形化的用户界面，可以在不输入任何SQL命令的条件下更改表结构。当然，您也可以在dmSQL命令行工具中使用DEF TABLE命令来直接查看表结构。

➲ **示例**

下例说明了如何查看表**tb_staff**的结构：

```
dmSQL> DEF TABLE tb_staff;
       CREATE TABLE SYSADM.TB_STAFF (
           NATION  CHAR(20) default 'R.O.C' ,
           ID  INTEGER not null  ,
```

```
NAME  CHAR(30) not null  ,
JOINDATE  DATE default CURDATE() ,
HEIGHT  FLOAT DEFAULT NULL ,
DEGREE  VARCHAR(200) DEFAULT NULL )
in TS_REG  LOCK MODE ROW  FILLFACTOR 80  NOCACHE;
```

表的更改

在创建一个表后，您可以对表执行如下更改：

- 增加/删除字段
- 更改字段定义
- 更改填充因子（FILLFACTOR）的值
- 打开/关闭取消快取（NOCACHE）选项
- 将表移动到其它表空间

表结构可通过dmSQL命令或JDBA工具来更改。

添加/删除字段

无论表的字段是否为空，您都可以对表执行添加/删除操作。向空表中添加新字段就等于扩展一张表的表结构，并在表结构的末尾添加一个字段。您也可以在表中任一字段的前后添加一个新字段。

当向表中添加新字段时，DBMaster不仅会扩展表结构，还会将字段赋予默认的空值（NULL）。如果用户想向表中添加一个NOT NULL完整性约束的字段，您可以给原有记录的此字段指定一个值（有关默认值请参考*字段默认值*章节），有关更多SQL语法的细节请参考*SQL命令与函数参考手册*。

➲ **示例1**

向表**tb_staff**中添加一个名为**photo**的字段：

```
dmSQL> ALTER TABLE tb_staff ADD COLUMN photo LONG VARCHAR;
```

➲ **示例2**

在**tb_staff**表的**name**字段名后添加一个名为**city**的字段，并将默认值设为**Taipei**：

```
dmSQL> ALTER TABLE tb_staff ADD COLUMN city CHAR(20) default 'Taipei'
AFTER name;
```

➲ **示例3**

如果**tb_staff**表非空，并且用户希望向此表中添加一个非空（non-null）字段，那么可用关键字GIVE来为新添加的字段赋值。下例说明了如何在表**tb_staff**中添加一个名为**HireDate**的非空字段：

```
dmSQL> ALTER TABLE tb_staff ADD (HireDate date NOT NULL give '2000-02-
20');
```

➲ **示例4**

从表**tb_staff**中删除一个名为**photo**的字段：

```
dmSQL> ALTER TABLE tb_staff DROP COLUMN photo;
```

字段定义的更改

您可以更改字段的属性，如字段名称、数据类型、字段顺序、默认值或字段限制等。在更改字段的数据类型前，请确定新的数据类型和原始数据类型的一致性，否则会因为数据不匹配而导致更改失败。例如：您无法将CHAR类型的字段更改成DATE类型。

➲ **示例1**

更改**tb_staff**表中字段**photo**的名称：

```
dmSQL> ALTER TABLE tb_staff MODIFY photo NAME TO emp_photo;
```

➲ **示例2**

更改**tb_staff**表中字段**height**的数据类型：

```
dmSQL> ALTER TABLE tb_staff MODIFY height TYPE TO decimal(10,2);
```

➲ **示例3**

更改字段顺序，将 **height**字段放到**HireDate**字段之前：

```
dmSQL> ALTER TABLE employee MODIFY height BEFORE HireDate;
```

➲ **示例4**

更改字段**nation**的默认值：

```
dmSQL> ALTER TABLE tb_staff MODIFY nation DEFAULT TO 'Taiwan';
```

➲ **示例5**

更改字段**height**的约束条件：

```
dmSQL> ALTER TABLE tb_staff MODIFY height CONSTRAINT TO CHECK value <
250;
```

锁模式的更改

如果您想让多个用户同时连接数据库，您可以把表的锁定模式降低一些（例如**ROW**锁）。但是这样会使DBMaster占用更多的资源，所以您必须在两者之间作取舍。要想获取有关锁模式的更多信息，请参考第9.4章*锁*。

➲ **示例**

更改**tb_staff**表的锁模式：

```
dmSQL> ALTER TABLE tb_staff SET LOCK MODE ROW;
```

填充因子（FILLFACTOR）的更改

在创建或更改表时，您可以设定表的填充因子（FILLFACTOR）。要想获取有关FILLFACTOR的更多信息，请参考*创建表*章节中的*填充因子（FILLFACTOR）*部分。

➲ **示例**

更改表employee的填充因子（FILLFACTOR）：

```
dmSQL> ALTER TABLE tb_staff SET FILLFACTOR 90;
```

设置取消快取（NOCACHE）的开启/关闭状态

您可以随时设置取消快取功能的开启/关闭状态。有关更多取消快取的信息，请参考*管理表*章节中的*取消快取（NOCACHE）*部分。

➲ **示例**

关闭**tb_staff**表的取消快取功能：

```
dmSQL> ALTER TABLE tb_staff SET NOCACHE OFF;
```

移动表到其它表空间

您可以移动表到另一个表空间。若表和该表的索引位于同一表空间内，在移动表的同时也可以将索引一起移动另一个表空间；若表和该表的索引位于不同表空间内，则无法将索引移动到另一个表空间，因此我们可以在另一个表空间中重建索引。

设置**FASTCOPY ON**可以提高将表移动到另一个表空间的执行速度。当一个表被移动时，系统无需缓存，只需执行一次日志操作即可直接将一个数据页复制到另一个数据页。如此一来，就可大幅度降低对日志的操作。

通过移动表到其它表空间可以将表储存到其它磁盘，以避免磁盘已满时无法存储该表。

将表移动到其它表空间存在以下限制：

- 不能移动系统表、临时表或视图到其它表空间。
- 不能移动永久表到系统表空间或临时表空间。
- 永久表的索引不能创建在临时表空间中。
- 临时表的索引只能创建在临时表空间中。
- 系统表的索引只能创建在系统表空间中。
- 不能将数据从一个表复制到相同的表中。
- 不能将表从一个表空间移动到相同的表空间中。

➲ 示例

```
dmSQL> CREATE TABLE tb_staff (c1 int, c2 char(10)) in ts_reg; // create
table tb_staff in ts_reg
dmSQL> CREATE INDEX idx_desc ON tb_staff (c1); // defaultly store index
in ts_reg where the table tb_staff is stored
dmSQL> SET FASTCOPY OFF;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // slowly move table
tb_staff and index idx_desc to ts_app
dmSQL> SET FASTCOPY ON;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // quickly move
table tb_staff and index idx_desc to ts_app
dmSQL> REBUILD INDEX idx_desc FOR tb_staff IN ts_shrink; // rebuild
index idx_desc in ts_shrink
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_aut; // only move table
tb_staff to ts_aut, index idx_desc no change
```

使用JSONCOLS类型

DBMaster支持JSONCOLS类型。JSONCOLS类型是一个JSON表达式，可以将表中的所有动态字段转化为结构化的输出。JSONCOLS类型用于存储表中的所有动态字段，可以结合动态字段使用。有关更多动态字段的信息请参考*使用动态字段*章节。表中有很多字段且大部分字段的值为NULL且分别操作很麻烦时，就需要考虑使用JSONCOLS类型。

JSONCOLS类型可以通过CREATE TABLE或ALTER TABLE语句来定义。有关更多SQL语法的细节请参考*SQL命令与函数参考手册*的*SQL命令*章节。定义了JSONCOLS字段之后即可将其作为普通字段使用。此外，由于JSONCOLS字段是从LONG VARBINARY派生的，且在DBMaster中用户不能在大对象上创建索引，因此也不能在已定义的JSONCOLS字段上创建索引，但是可以创建全文索引。

要定义JSONCOLS类型，需使用CREATE TABLE或 ALTER TABLE语法的<*JSONCOLS_type_name*> JSONCOLS关键字。

在DBMaster中，JSONCOLS类型表示为JSONCOLS字段。

您可以通过如下格式来定义JSONCOLS类型的JSON表达式：

```
{col1_name:col1_value,col2_name:col2_value,col3_name:col3_value...}
```

JSONCOLS类型的示例如下所示：

```
{"ID":1234,"NAME":"linda","PHONE":"1234567"}
```

注意 *在JSONCOLS类型的JSON表达式中，可以省略含NULL值的动态字段。*

如果您在插入或更新数据时以错误的格式定义了JSON表达式，则会出现下列错误信息：“ERROR（8077），[DBMaster]无效的JSON格式”。

如果JSON表达式中包含DATE、TIME或TIMESTAMP类型的值，当您查询该JSONCOLS字段时，查询结果显示为Epoch时间。在DBMaster中，Epoch时间是指协调世界时（UTC）从1970年1月1日午夜开始经过的毫秒数。

由于DBMaster是以内部方式来存储JSONCOLS字段的数据，因此在查询时，动态字段的显示顺序可能与其插入顺序不同。

当删除JSONCOLS字段或带有该JSONCOLS字段的表时，JSONCOLS字段中存储的动态字段也会被系统自动删除。

使用JSONCOLS类型时，您需要考虑以下几点原则：

- 不能更改JSONCOLS类型。如果要更改，则需要删除和重建JSONCOLS类型。
- 一个表中只允许一个JSONCOLS类型存在。
- 不能在JSONCOLS类型中定义约束或默认值。

JSONCOLS类型的安全模块类似于普通字段。当您在JSONCOLS字段上执行SELECT、INSERT、UPDATE和DELETE语句时，要求您具备和JSONCOLS字段相应的权限。

当对JSONCOLS字段的数据执行操作时，需要使用单个动态字段的名称，或者参照JSONCOLS类型的名称以及使用JSONCOLS类型的JSON表达式来指定JSONCOLS类型的值。动态字段能够以任何顺序显示在JSONCOLS字段中。

➲ **示例**

使用如下命令，创建一个带有JSONCOLS类型的表：

```
dmSQL> CREATE TABLE student(name CHAR(30), info JSONCOLS);
```

或

```
dmSQL> CREATE TABLE student(name CHAR(30));
dmSQL> ALTER TABLE student ADD COLUMN info JSONCOLS;
```

使用JSONCOLS类型的名称将数据插入表**student**：

```
dmSQL> INSERT INTO student (name,info) VALUES
('jessia','{"desk_id":3,"birthday":"1986-09-19","score":90}');
1 rows inserted
dmSQL> INSERT INTO student (name,info) VALUES
('pine','{"desk_id":4,"birthday":"1987-03-03","score":95}');
1 rows inserted
```

使用“SELECT *”命令查询表**student**：

```
dmSQL> SET blobwidth 80;
dmSQL> SELECT * FROM student;
       NAME                             INFO
================= ================================================


jessia             {"score":90,"birthday":"1986-09-19","desk_id":3}
pine               {"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
```

使用JSONCOLS类型的名称查询表**student** ：

```
dmSQL> SELECT name, info FROM student;
       NAME                             INFO
================= ================================================


jessia             {"score":90,"birthday":"1986-09-19","desk_id":3}
pine               {"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
```

使用JSONCOLS类型的名称更新表**student**的数据：

```
dmSQL> UPDATE student SET info = '{"desk_id":7, "birthday":"1986-09-
19","score":88}' WHERE name='jessia';
1 rows updated
```

将名为**birthday**的字段的数据类型更改为DATE型：

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN birthday DATE;
dmSQL> SELECT info FROM student;
```

```
                                INFO
=================================================================

{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
dmSQL> INSERT INTO student (name,desk_id,birthday,score) VALUES
('mike','8','1985-02-15','92');
dmSQL> SELECT info FROM student;
                                INFO
=================================================================

{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

在名为**info**的JSONCOLS字段上创建一个全文索引：

```
dmSQL> CREATE TEXT INDEX idx_stu ON student(INFO);
```

在名为**info**的JSONCOLS字段上创建一个视图：

```
dmSQL> CREATE VIEW view1 AS SELECT info FROM student;
dmSQL> SELECT * FROM view1;
                                  INFO
=================================================================

{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

使用动态字段

DBMaster支持动态字段。动态字段不在表定义中出现，它是从JSON字符串派生的KEY，只能在表将一个字段声明为JSONCOLS字段时使用。

动态字段用于存储半结构化的数据、记录的属性有数千种或数据类型经常更改的数据，可以结合JSONCOLS类型使用。有关更多JSONCOLS类

型的信息请参考*使用JSONCOLS类型*章节。表中有很多字段且大部分字段的值为NULL时，就需要考虑使用动态字段。

动态字段存储在JSONCOLS字段上，其描述信息存储在表SYSDESCOL中。有关更多表SYSDESCOL的信息请参考*DBMaster系统目录表*章节。有关更多JSONCOLS类型的信息请参考*使用JSONCOLS类型*章节。

动态字段的安全模块类似于普通字段，其特征如下所示：

- 可以在动态字段上创建索引。
- 动态字段仅支持更改数据类型。
- 动态字段支持如下数据类型： SMALLINT、 INT、FLOAT、DOUBLE、DATE、TIME、TIMESTAMP、CHAR、VARCHAR、NCHAR、NVARCHAR。
- 动态字段必须为可空的值。
- 动态字段不能有默认值。
- 动态字段不能有字段约束。
- 动态字段不能用于存储命令。
- 动态字段不能用于存储过程。
- 动态字段不能用于触发器。

动态字段创建后无需定义即可直接使用。动态字段的默认数据类型是varchar(256)，您可以使用ALTER TABLE ADD DYNAMIC COLUMN命令来将其更改为其他数据类型。此外，当该动态字段被插入到表时，也可以使用该命令声明动态字段的数据类型。如果您想要将已声明的数据类型更改为另一种数据类型，则需使用ALTER TABLE MODIFY DYNAMIC COLUMN命令；如果您想删除动态字段的描述信息，则可以使用ALTER TABLE DROP DYNAMIC COLUMN命令。有关更多SQL语句的信息请参考*SQL命令与函数参考手册*。另外，如果在插入数据时未执行ALTER TABLE ADD DYNAMIC COLUMN命令，但是却使用该命令声明了动态字段的数据类型，之后又无法将之前插入的数据转换为已声明的数据类型时，在查询时该数据会表示为NULL，此时也不会报错。

当您使用参数将数据插入动态字段或更新动态字段的数据时，DBMaster仅支持插入VARCHAR类型的数据，此时，您可以使用隐式数据转换功能，插入其他类型的数据。

当对动态字段的数据执行操作时，需要使用单个动态字段的名称，或者参照JSONCOLS类型的名称以及使用JSONCOLS类型的JSON表达式来指定JSONCOLS类型的值。动态字段能够以任何顺序显示在JSONCOLS字段中。

➲ **示例**

以下操作基于表**student**。有关更多的表**student**的信息，请参考*使用JSONCOLS类型*章节中的示例。

使用动态字段的名称将数据插入表**student**：

```
/* implicit data conversion is closed by default */
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'demi','85';     /* it is ok */
1 rows inserted
dmSQL/Val> 'finly',82;     /* INT cannot be converted to CHAR */
ERROR (9629): value list syntax error
dmSQL/Val> END;
dmSQL> SET itcmd ON;
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'finly',82;     /* using implicit data conversion */
1 rows inserted
dmSQL/Val> END;
dmSQL> SET itcmd OFF;
dmSQL> INSERT INTO student (name,desk_id,birthday,score)
VALUES('linda','1','1982-01-01','91');
1 rows inserted
dmSQL> INSERT INTO student (name,desk_id,birthday,score)
VALUES('glow','2','1984-03-25','93');
1 rows inserted
dmSQL> INSERT INTO student (name,desk_id,birthday,score)
VALUES('kitty','abc','1980-02-27','97');
1 rows inserted
```

使用“SELECT *”命令查询表**student**：

```
dmSQL> SELECT * FROM student;
     NAME                          INFO
============= ============================================


jessia         {"score":88,"birthday":"1986-09-19","desk_id":7}
pine           {"score":95,"birthday":"1987-03-03","desk_id":4}
mike           {"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
demi           {"SCORE":"85"}
finly          {"SCORE":"82"}
linda          {"BIRTHDAY":378662400000,"DESK_ID":"1","SCORE":"91"}
glow           {"BIRTHDAY":448992000000,"DESK_ID":"2","SCORE":"93"}
kitty          {"BIRTHDAY":320428800000,"DESK_ID":"abc","SCORE":"97"}
8 rows selected
```

使用动态字段的名称查询表**student**：

```
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID      BIRTHDAY            SCORE
=============  ===========  ============  ====================
jessia          7            19*           88
pine            4            19*           95
mike            8            19*           92
demi                NULL     NU*           85
finly               NULL     NU*           82
linda           1            19*           91
glow            2            19*           93
kitty          abc           19*           97
8 rows selected
```

使用动态字段的名称更新/删除表**student**的数据：

```
dmSQL> UPDATE student SET score='88' WHERE name='linda';
1 rows updated
dmSQL> DELETE FROM student WHERE desk_id='2';
1 rows deleted
```

在该表中添加动态字段的描述：

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN desk_id INT;
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN score DOUBLE;
```

在表**student**中插入数据：

```
dmSQL> INSERT INTO student (name, desk_id, age, score)
VALUES('jane','12','1982-05-07',96);
ERROR (6150): [DBMaker] the insert/update value type is incompatible
with column data type or compare/operand value is incompatible with
column data type in expression/predicate
dmSQL> INSERT INTO student (name, desk_id, age, score)
VALUES('jim',8,'1984-09-26',98);
1 rows inserted
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID       BIRTHDAY            SCORE
==============  ============  ============  ====================
jessia                    7  1986-09-19    8.80000000000000e+001
pine                      4  1987-03-03    9.50000000000000e+001
mike                      8  1985-02-15    9.20000000000000e+001
demi                NULL         NULL       8.50000000000000e+001
finly               NULL         NULL       8.20000000000000e+001
linda                     1  1982-01-01    8.80000000000000e+001
kitty               NULL      1980-02-27    9.70000000000000e+001
jim                       8     NULL       9.80000000000000e+001
8 rows selected
```

修改动态字段**score**的数据类型：

```
dmSQL> ALTER TABLE student MODIFY DYNAMIC COLUMN score TYPE TO INT;
```

在动态字段**desk_id**上创建索引：

```
dmSQL> CREATE INDEX idx1 ON student(desk_id);
```

删除动态字段**birthday**的描述信息：

```
dmSQL> ALTER TABLE student DROP DYNAMIC COLUMN birthday;
```

表锁定

DBMaster可自动设置数据库的锁定模式。如果要对某个表执行SELECT或UPDATE操作，您还可以对表进行手动锁定。当用户查看或更改一张锁定的表时，会阻止其他用户对该表进行更改。

DBMaster提供了几种表锁定的选择：如果要查看数据可以使用*共享锁（shared locks）*；如果要更改数据可使用*互斥锁（exclusive locks）*。我们还可以用WAIT或NO WAIT选项来设置锁的等待状态。有关这些特征

的详细信息，请参考*SQL命令与函数参考手册*。要想获取表锁定、并发控制和事务处理的相关信息，请参考第9章*并发控制*。

➲ **示例**

用共享锁来锁定表**tb_staff**，如果无法立即获得锁也无需等待。

```
dmSQL> LOCK TABLE tb_staff IN SHARE MODE NO WAIT;
```

删除表

您可以删除没有用的表。当表被删除后，该表中的数据和索引也将一并删除，表所在的数据页将被释放。

➲ **示例1**

使用DROP TABLE命令删除**tb_staff**表：

```
dmSQL> DROP TABLE tb_staff;
```

➲ **示例2**

使用DROP TABLE IF EXISTS命令删除**tb_staff**表：

```
dmSQL> DROP TABLE IF EXISTS tb_staff;
```

6.3 视图管理

DBMaster提供给使用者定义视图的能力。所谓视图并不是一张真正的表，要建立视图必须先定义视图的名称和它所对应在数据库中表或视图的查询条件。数据库会把这个视图的定义储存在数据库中，而此视图所对应的数据并没有实际存储在数据库中。也就是说，在您浏览这个视图时，数据库会依据定义的视图将其所对应的数据从视图定义的实际表和查询条件中筛选出来。

视图对数据库的查询是非常有效的。例如，对于一个相当复杂的查询指令，您可以把它定义成视图。如此一来，您就可以重复使用这个视图，而不用每次都重新输入这个复杂的查询指令。由于视图可以用它的定义来限制使用者存取的表字段和数据，所以，定义视图可以加强数据库的安全性。

但如果视图的定义超过一个表时，那么视图所呈现出来的数据就无法确定原来所在表的位置。由于视图是从表查询继承而来，因此，这个视图就只能用来查询，不能进行更新、添加或删除数据的操作。

创建视图

视图可以通过dmSQL或JDBA工具来创建。在创建视图时，您必须定义视图的名称和相关表或视图的查询指令。

您可以定义视图的字段名称。如果您没有定义视图的字段名称，视图名称将继承相关表的字段名称。

您还可以使用CREATE VIEW语法。例如，如果您只允许其他用户查看**tb_staff**表中的两个字段**name**和**ID**，那么您可以使用以下的SQL指令来创建视图**vi_staff**。

➲ **示例1**

从**tb_staff**表中使用CREATE VIEW命令创建视图**vi_staff**：

```
dmSQL> CREATE VIEW vi_staff (empName, empId) AS SELECT name, ID FROM
tb_staff;
```

使用CREATE OR REPLACE VIEW语法，假设存在视图**vi_staff**，仅允许其他用户查看**tb_staff**表中的字段**name**和**ID**，如果想在不改变视图权限的前提下查看三个字段**name**、**ID**和**age**，我们需要更改视图的定义，用户可以使用以下SQL指令来重建视图**vi_staff**。

➲ **示例2**

从**tb_staff**表中使用CREATE VIEW命令创建视图**vi_staff**：

```
dmSQL> CREATE OR REPLACE VIEW vi_staff (empName, empId, empAge) AS
SELECT name, ID, age FROM tb_staff;
```

浏览视图结构

您可以通过dmSQL或JDBA工具来查询视图结构。可在dmSQL命令行工具中直接输入DEF VIEW命令来查询视图结构。

➲ **示例**

查看视图**vi_staff**的定义：

```
dmSQL> DEF VIEW vi_staff;
dmSQL> CREATE VIEW SYSADM.VI_STAFF(empname,empid) AS SELECT name,id FROM
SYSADM.TB_STAFF;
```

删除视图

您可以删除一个没有用的视图。删除视图只会删掉此视图储存在数据库中的定义，而对视图所对应表的数据没有影响。

➲ **示例1**

使用DROP VIEW命令删除视图**vi_staff**：

```
dmSQL> DROP VIEW vi_staff;
```

➲ **示例2**

使用DROP VIEW IF EXISTS命令删除视图**vi_staff**：

6.4 同义字管理

同义字也就是表或视图的别名。由于同义字只是一个别名的定义，所以数据库只需将它的定义储存在系统表中。

同义字主要用于简化表或视图的名称。通常情况下，DBMaster结合拥有者和对象的完整名称来识别表和视图的名称。但如果我们设置了同义字，我们就可以使用相应的同义字来访问表或视图，而无需键入它们的完整名称。由于同义字没有拥有者名称，所以它的名称必须是唯一的。同义字可通过dmSQL或JDBA工具来创建或删除。

创建同义字

➲ **示例1**

使用CREATE SYNONYM指令：

```
dmSQL> CREATE SYNONYM staff FOR SYSADM.tb_staff;
```

假设表**tb_staff**的拥有者是**SYSADM**，该指令可为表**SYSADM.tb_staff**创建一个别名**staff**，所有用户都可以直接使用这个同义字**staff**来参照表**SYSADM.tb_staff**而无需键入全名。

➲ **示例2**

使用CREATE OR REPLACE SYNONYM 指令：

```
dmSQL> CREATE OR REPLACE SYNONYM staff FOR SYSADM.tb_staff;
```

假设表**tb_staff**已存在一个别名**staff**，该指令可替换该别名而无需删除。

删除同义字

您可以删除一个不再需要的同义字。在删除同义字时，您只会将相关的定义从系统表中删除，而同义字对应的表或视图并不会被删除。

➲ **示例1**

使用DROP SYNONYM指令删除同义字**staff**：

```
dmSQL> DROP SYNONYM staff;
```

➲ **示例2**

使用DROP SYNONYM IF EXISTS指令删除同义字**staff**：

```
dmSQL> DROP SYNONYM IF EXISTS staff;
```

6.5 索引管理

使用索引可以对数据行作快速的随机访问。您可以在表上建立索引，这样可以加快数据的查询。例如：当用户执行一个SELECT NAME FROM EMPLOYEE WHERE id = 306004查询语句时，如果在字段**ID**上建有索引，那么读取数据的速度就会快很多。

一个索引可以由1-32个字段组成，表中的所有字段都可以用于建立索引。

您可以根据数据有没有重复值，来建立唯一性索引（unique）或非唯一性索引（non-unique）。如果字段建立了唯一性索引，那么这个字段除了空值（NULL）以外，不允许有相同的值。如果您在一个已有数据的表上建立唯一性索引，DBMaster会检查表中现存的数据是否唯一。如果有重复的值，DBMaster会返回一个错误信息。在表上创建一个唯一性索引后，DBMaster会检查新增或更新的数据是否和数据库中的数据重复，如果有重复的话，这笔数据就无法新增到数据库中。

创建索引时，您还可以指定索引中每个字段的排序为递增还是递减。例如：假设表中有5个值：1、3、9、2、6，如果设为递增，它的顺序为：1、2、3、6、9；如果设为递减，它的顺序为：9、6、3、2、1。

当用户执行一条查询语句时，因为数据库选择用索引扫描数据，数据的输出顺序会受到索引顺序的影响。

➲ **示例**

当您执行以下查询：

```
dmSQL>SELECT name, age FROM friend_table WHERE age > 20;
```

在字段**age**上建立一个递减的索引，输出如下所示：

```
      name                    age
---------------          ---------------
Jeff                              49
Kevin                             40
Jerry                             38
Hughes                            30
```

```
Cathy                                    22
```

建立索引和建立表一样，都可以设定填充因子（fillfactor）。索引的填充因子是定义索引页中可以储存值的密度，索引填充因子的范围是1%--100%，默认值为100%。如果您建立了索引还需要经常更新数据，建议您将填充因子的值设定松一点，例如60%；如果您不需要更新该表中的数据，您可以使用表的默认填充因子100%。

用户也可以将索引创建在单独的表空间上，这样在访问多个磁盘时，可以提高磁盘的I/O效率。

在创建一个索引前，建议您最好先将所有数据存入数据库，特别是在表的数据量很大时。这是因为如果您在数据加载之前先创建索引，每次在增加一笔数据时，都必须更新索引数据，这样会降低磁盘I/O的效率。 所以最好在数据载入后再建索引以提高效率。

创建索引

您可以使用JDBA工具的创建索引向导或dmSQL的CREATE INDEX指令来创建索引。在建立索引时，您必须指定索引的名称和字段。还可以设定字段的排序是递增还是递减，默认的顺序是递增。

➲ **示例1**

在**tb_staff**表的**ID**字段上创建一个索引**idx_desc**，并且排序是递减的：

```
dmSQL> CREATE INDEX idx_desc ON tb_staff (ID DESC);
```

➲ **示例2**

在**tb_staff**表的**ID**字段上创建一个唯一性索引**idx_uniq**：

```
dmSQL> CREATE UNIQUE INDEX idx_uniq ON tb_staff (ID);
```

➲ **示例3**

创建一个索引，并且指定它的填充因子（FILLFACTOR）：

```
dmSQL> CREATE INDEX idx_fill ON tb_staff(name, id DESC) FILLFACTOR 60;
```

➲ **示例4**

在表空间**ts_reg**上创建一个索引：

```
dmSQL> CREATE INDEX idx_reg ON tb_staff_(name, age DESC) IN  ts_reg
FILLFACTOR 60;
```

创建表达式索引

索引不仅可以创建在一个简单的字段上，还可以创建在一个表达式字段或用户自定义函数（UDF）字段上。

➲ **示例1**

在表**tb_staff**的**basepay+bonus**字段上创建一个索引**idx_expr**：

```
dmSQL> CREATE INDEX idx_expr ON tb_staff (basepay+bonus);
```

➲ **示例2**

在表**tb_salary**的用户自定义函数substring **(nation,1,3)**上创建一个索引**idx_substr**：

```
dmSQL> CREATE INDEX idx_substr ON tb_salary (substring(nation,1,3)
desc);
```

➲ **示例3**

在表**tb_salary**的表达式和用户自定义函数**abs(bonus)**上创建一个索引**idx_udf**：

```
dmSQL> CREATE INDEX idx_udf ON tb_salary (basepay+abs(bonus)-tax desc);
```

在XML字段上创建索引

为了提高XML的查询性能，可以在XML字段上创建特殊的XML索引。XML索引支持XML函数**extract()** 和 **extractvalue()**。下面的例子说明如何使用dmSQL在XML字段上创建索引。更多有关的语法以及CREATE INDEX命令的使用请参考*SQL命令与函数参考手册*。

➲ **示例1**

使用XML函数**extract**来创建索引：

```
dmSQL> CREATE INDEX idx_extr ON tb_extract (extract(id,
'/order/items/item/@product', NULL));
```

➲ **示例2**

使用XML函数**extractValue**来创建索引：

```
dmSQL> CREATE INDEX idx_extrV ON tb_extract (extractValue(id,
'/order/items/item/@product', NULL));
```

extract()和extractvalue()的主要区别是：

extract()

- 允许多值、单值或者零值。
- 不支持升序或降序：asc / desc
- 不支持唯一索引
- extractValue()
- 允许UDF返回的结果为单值或零值 （当UDF结果返回多值时，则不能成功为已经存在的元组创建索引或插入新的元组）
- 支持升序或降序排列 asc / desc
- 支持唯一索引

创建过滤索引

过滤索引（条件索引）是一种带有WHERE条件子句的优化索引。当用户从一个定义明确的数据子集中查找符合条件的索引值时，过滤索引不失为一种最佳选择。也就是说，过滤索引在查找索引值之前就已经插入到数据页中，且过滤索引的数据不包括表的所有数据行，而是通过过滤条件（WHERE子句）选取数据行的一部分。过滤索引使用过滤谓词选择表中的部分数据行，同一个全表索引相比，一个设计巧妙的过滤索引不仅能够提高数据库的查询性能，还能降低表索引的维护和储存成本。

过滤索引比全表索引搜索的范围小且带有过滤统计，能直接提高数据库的查询性能和执行计划的效率。因为过滤统计只查找定义的过滤索引中符合条件的索引行，所以过滤统计会比全表统计更精确。

只有当数据操作语言（DML）会影响索引中的数据时，索引才需要被维护。因为过滤索引的查找范围较小，并且只有当索引数据被更改时才需要被维护，所以同全表索引相比，过滤索引的维护成本更低。数据库可以拥有大量的过滤索引，尤其当包含的数据需要频繁更改时。也就是说，当一个过滤索引包含的仅是频繁变更的数据，那么越小的索引越能降低更新统计的成本。

过滤索引能够节省磁盘的存储空间。您可以用多个过滤索引来替代一个全表索引，这样对存储空间的需求并不会出现明显的增加。

带有WHERE子句的CREATE INDEX语句可用于在现有表中定义索引。过滤索引类型仅限于非唯一性索引和唯一性索引，但不包含外键。WHERE子句中包含的字段最大值为32。

DBMaster允许以下用户创建过滤索引：

- 具备DBA或更高权限的用户
- 表拥有者
- 授予CREATE INDEX权限的用户

WHERE子句允许以下谓词组合，包括：

- 表的任一字段
- 常数值
- 比较符。=，>，>=，<，<=，!=。例：c1>=3
- Like。例：c3 like ‘abc’
- NULL和NOT NULL。例：c4 is null
- in list。例：c5 in (1,3,5)
- 运算符。+，-，*，/ 。例：c1+c2>5
- 用户自定义函数（UDF）。例：abs(c6)>5

- BLOB运算符。match, contain
- AND组合。例：c1=3 and c2=5 and c3=7
- OR组合。例：c1=3 or c2=5 or c3=7

WHERE子句不允许出现以下语句：

- 子查询
- 主变量
- 混合AND和OR，例：c1=3 or c2=5 and c3=7

注意 *用户不能在一个全文索引上定义过滤条件（WHERE子句）*

➲ **示例1**

在表**tb_salary**上使用**where**子句的like谓词创建一个过滤索引**filidx_basepay**：

```
dmSQL> CREATE INDEX filidx_basepay ON tb_salary (id, basepay) where name
like 'abc%';
```

➲ **示例2**

在表**tb_salary**上使用**where**子句创建一个过滤索引**filidx_income**：

```
dmSQL> CREATE INDEX filidx_income ON tb_salary(basepay+bonus,tax)where
id>30;
```

删除索引

您可以使用JDBA工具或dmSQL的DROP INDEX语句删除索引。如果这个索引是主键并且被其它表参考，就无法删除该索引。请参考*数据完整性管理*章节。

➲ **示例**

删除表**tb_staff**的索引**idx_desc**：

```
dmSQL> DROP INDEX idx_desc FROM tb_staff;
```

重建索引

您可以使用JDBA工具或dmSQL的REBUILD INDEX语句来重建索引。当索引存在碎片时会降低系统效率。这时您可以使用重建索引来提高系统的效率。重建索引会删除旧的索引并创建一个新的索引。

您可以移动表到另一个表空间。若表和该表的索引位于同一表空间内，在移动表的同时可以将索引一起移动到另一个表空间；若表和该表的索引位于不同表空间内，那么将无法把索引移动到另一个表空间，因此我们可以在另一个表空间中重建索引。

➲ **示例1**

为表**tb_staff**重建索引**idx_fill**：

```
dmSQL> REBUILD INDEX idx_fill FOR tb_staff;
```

➲ **示例2**

```
dmSQL> REBUILD INDEX idx FOR tb_staff IN ts_reg;
```

6.6 自动索引管理

随着全球数据库的发展，索引管理已经从手动管理演变为自动管理。根据用户执行的查询语句，自动索引后台程序可对用户需求加以分析，更加智能化地管理索引。

DBMaster支持自动索引。自动索引类似于非唯一性索引，但是它能被自动索引后台程序自动创建或删除。如果设置了AUTOCOMMIT ON，那么在创建自动索引时，DBMaster仅需请求*Update（U）*锁，也就是说，DBMaster允许其他用户同时查询该表。

DBMaster支持自动索引后台程序操作自动索引，该操作可通过**收集机制**和**处理机制**来实现。仅在自动索引后台程序启动后，收集机制才会分析用户执行查询语句的执行计划，并将分析结果（0或者多个记录）记录在*DMSCAN.LOG*文件中。处理机制可以通过使用dmSQL命令*SYNC AUTO INDEX*或JDBA工具的同步自动索引向导设置关键字**DB_IdxTm**和**DB_IdxTv**来将之唤醒。处理机制的主要作用如下：读取*DMSCAN.LOG*并分析所有日志以确定是否需要创建自动索引、更新索引使用信息；删除由**DB_IdxDp**（仅自动索引会被删除，其它类型的索引不会被删除）指定的超过期限未使用的自动索引。在此过程中，创建或删除的索引会被记录在*DMAUTOIDX.LOG*文件中，便于用户确认自动索引服务器的状态。

实际上，当客户端的用户执行查询语句时，日志信息不会立即被记录在*DMSCAN.LOG*文件中，此时，收集机制会先将日志信息存储在客户端的缓存（固定大小为2560字节）中，当缓存已满时才会将日志信息写入*DMAUTOIDX.LOG*文件。此外，如果用户断开数据库连接或执行*SYNC AUTO INDEX*命令，则缓存中的数据也会被写入*DMAUTOIDX.LOG*文件。这种方法不仅可以避免多用户同时将数据写入*DMAUTOIDX.LOG*文件而引起的内容混乱，而且可以集中I/O操作以提高收集机制的性能。

您需要在**dmconfig.ini**配置文件中设置关键字**DB_IdxSv**、**DB_IdxLg**、**DB_IdxTm**、**DB_IdxTv**、**DB_IdxDp**以及**DB_IdxLn**来控制自动索引后台程序，以便自动创建或删除自动索引。在数据库运行期间，只有具备

DBA、SYSDBA或SYSADM权限的用户才能调用**setSystemOption()**来设置除**IDXLG**之外的选项。

自动索引和其它索引的不同之处如下：

- 自动索引可以通过自动索引后台程序自动创建。
- 如果自动索引可以与其它索引合并或超过设定的期限未被使用，则可以通过自动索引后台程序自动删除。
- 用户创建的自动索引最大字段数为32，但是自动索引后台程序创建的自动索引最大字段数为16。
- 当设置*COMMIT ON*时，创建自动索引需要**U**锁，但是创建其它类型的索引则需要**X**锁。

DBMaster也支持*SET LOADAUTOINDEX ON|OFF*语句和ODBC函数*SQLSetConnectOption*，以便用户决定是否加载自动索引。有关更多ODBC函数的信息，请参考*ODBC程序员参考手册*。

➲ **示例1**

通过调用系统存储过程**SetSystemOption**开启自动索引服务器。

```
dmSQL> CALL SETSYSTEMOPTION('IDXSV','1');  //activate auto index server

dmSQL> SELECT * FROM tb_staff WHERE joinDate="1986-07-20";   //create
auto index by daemon

dmSQL> sync auto index;    //wake up auto index daemon

dmSQL> SELECT * FROM sysindex;

dmSQL> SELECT * FROM sysindexref;
```

➲ **示例2**

通过调用系统存储过程**SetSystemOption**将天数重置为60天，以删除自动索引。

```
dmSQL> CALL SETSYSTEMOPTION ('IDXDP','60');
```

创建自动索引

您可以使用dmSQL命令*CREATE AUTO INDEX*手动创建自动索引或通过自动索引后台程序自动创建自动索引。有关更多如何手动创建自动索引的信息，请参考*SQL命令与函数参考手册*的*CREATE INDEX*语法。

当自动索引后台程序在表中创建自动索引时，会指定自动索引的类型、ID以及字段的升降排序，默认的字段排序是**升序**。

自动索引后台程序创建的自动索引名称由*AUTO*、*下划线*、*字段ID*以及*字段排序（字段排序用D或者null表示，D表示降序，null表示升序）*组成。如果名称相同的两个索引合并，则无需重新创建索引；如果这两个索引无法合并，则需要重新创建一个索引，并命名为index_name__Rxxx，Index_name表示这两个索引的名称，**xxx**为一个随机数。

➲ **示例1**

在表**tb_staff**的字段**ID**和**NAME**上创建一个自动索引**AUTO_1D_2**，通过dmSQL的命令选项DESC将ID字段的排序设定为降序：

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2 ON tb_staff (ID DESC, NAME);
```

➲ **示例2**

如果新索引的名称和旧索引相同，且无法合并，那么用户可以将新索引的名称扩展为**AUTO_ID_2__R321**，R后面的三位数字**321**是一个随机数。

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2__R321 ON tb_staff (ID DESC, NAME);
```

创建表达式自动索引

自动索引不仅能在普通字段上自动创建，还能在表达式字段或用户自定义函数字段上创建。

➲ **示例 1**

在表**tb_salary**的表达式**basepay+bonus**字段上创建自动索引**auto_idx_expr**。

```
dmSQL> CREATE AUTO INDEX auto_idx_expr ON tb_salary (basepay+bonus);
```

➲ **示例2**

在表**tb_salary**的用户自定义函数子字符串（**nation,1,3**）上创建自动索引**auto_idx_substr**。

```
dmSQL> CREATE AUTO INDEX auto_idx_substr ON tb_salary
(substring(nation,1,3) DESC);
```

删除自动索引

自动索引后台程序可以自动删除超过**DB_IdxDp**指定期限而未被使用的自动索引。另外，您也可以使用dmSQL命令*DROP AUTO INDEX*来删除自动索引，如果索引是作为主键或参照其它表字段而创建的，则不可删除。有关更多的主键的信息，请参考*数据完整性管理*章节。

➲ **示例**

从表**tb_staff**删除索引**AUTO_1D_2**：

```
dmSQL> DROP INDEX AUTO_1D_2 FROM tb_staff;
```

6.7 全文索引管理

全文索引可以快速访问表中包含一个或多个单词或短语字段的数据行。全文索引包含了从相应字段搜索得到的所有文本的描述，数据进行编码和重组，使读取效率比直接从表中要高。一旦用户为表创建了全文索引，其操作就是透明的。DBMS使用索引来提高文本的查询性能。

DBMaster提供了两个全文索引方式：特征向量和转换文件（IVF）方式。

全文索引可以建在字符类型的字段上，包括CHAR、VARCHAR、LONG VARCHAR、LONG VARBINARY和FILE类型。一个表内可以包含多个全文索引，一个全文索引可以包含多个字段。用户可以通过JDBA工具或dmSQL的 CREATE [SIGNATURE | IVF] TEXT INDEX命令来创建全文索引。

➲ **示例**

如果在字段**data**上创建了全文索引，那么DBMaster就会自动选择使用全文索引：

```
dmSQL> SELECT id FROM tb_book WHERE data MATCH 'compute';
```

DBMaster提供的字符串操作包括：MATCH、CONTAIN、CONTAINS和LIKE，但是只有MATCH和CONTAINS操作可以使用全文索引。

DBMaster提供了两种不同类型的全文索引：特征向量全文索引和反向文件全文索引。特征向量全文索引对处理少量的数据是很有效的；反向文件全文索引通常要占用更多的存储空间，但对处理大量数据是很有效的。

创建特征全文索引

如果在命令行中没有指定全文索引的方式，那么DBMaster将创建特征全文索引。用户可以通过JDBA工具或dmSQL的CREATE TEXT INDEX命令或 CREATE SIGNATURE TEXT INDEX命令来创建全文索引。

➲ 示例

在表**tb_staff**的data字段上创建特征全文索引**tidx_name**：

```
dmSQL> CREATE SIGNATURE TEXT INDEX tidx_name ON tb_staff(name);
```

特征全文索引参数

DBMaster为特征全文索引的执行性能和存储大小提供了两个参数。

- **Total text size (MB) —** 估计的所有源文件大小，范围为1 - 200 MB，默认大小为32MB。实际文本的大小没有200M的限制，如果文本的大小超过200MB，那么设置为200MB。 但是当为大量数据执行索引时，我们还是建议您使用IVF全文索引，这样可以较大的提高查询性能。
- **Scale —**索引大小和全文大小的期望比例。如用户设置全文大小为20 MB，并希望全文索引有10MB，那么设置比例为50（50%）。较大的比例有较好的检索性能。取值范围为10 ~ 200，默认值是40（40%）。

➲ 示例

在表**tb_staff**的**name**字段上创建一个全文索引**tidx_scale**，其中包含40MB的数据，并期望全文索引占用20MB的存储空间：

```
dmSQL> CREATE SIGNATURE TEXT INDEX tidx_scale ON tb_staff(name)
       TOTAL TEXT SIZE 40 MB
       SCALE 50;
```

- 用户可以使用全文索引参数的默认值，也可以通过更改全文索引参数或减少全文索引所占空间的大小来获得较高的执行性能。用户可以先设置参数，再查看系统的执行性能来调整参数的大小。

创建IVF全文索引

用户可以通过CREATE IVF TEXT INDEX命令来创建IVF全文索引。

➲ **示例**

为表**tb_staff**的name字段创建IVF全文索引**ivfidx_name**：

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(name);
```

IVF全文索引参数

您可以使用以下两个参数来创建IVF全文索引：

Storage path —保存反向文件索引的逻辑目录。用户可以在**dmconfig.ini**中定义此逻辑目录，默认值和**DB_DbDir**的相同，即数据库的主目录。有关反向文件的存储管理命名规则将在稍后章节描述。

Total text size (MB)—将被索引文件的近似大小（单位为MB），DBMaster会依据这个大小来划分文档的区段数，范围为1MB到10,000MB，默认值为500MB。

➲ **示例1**

在**tb_staff**表**name**字段的*\IVFDIR*路径上，创建一个反向全文索引**ivfidx_data**，其中包含400MB的数据。

首先，在数据库的配置文件**dmconfig.ini**中添加以下逻辑路径。

```
MYPATH1 = \IVFDIR
```

使用以下命令：

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(name)
    2> STORAGE PATH MYPATH1
    3> TOTAL TEXT SIZE 400 MB;
```

创建一个反向全文索引需要占用大量的内存资源。DBMaster会使用一个简单的规则来为创建的反向全文索引分配最多的内存空间。如果DBMaster无法检测到空闲内存或空闲内存少于128MB，那么最大的内存使用为64MB。否则，可利用的内存将是空闲内存的一半。用户可以在配置文件**dmconfig.ini**中添加**DB_IFMem**关键字，来近似的指出将要使用的内存最大值（MB）。

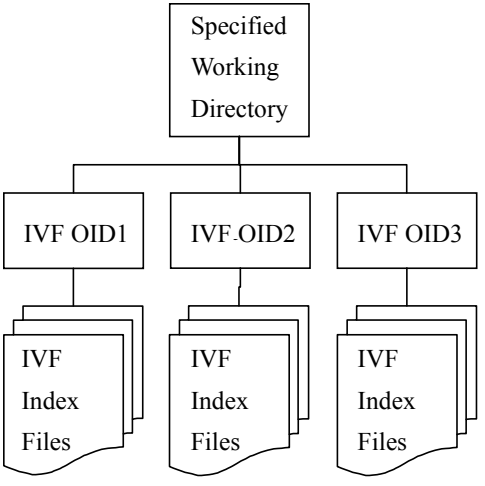
➲ **示例2**

在配置文件**dmconfig.ini**中，为创建的反向全文索引指定100MB的内存使用空间。

```
DB_IFMem = 100
```

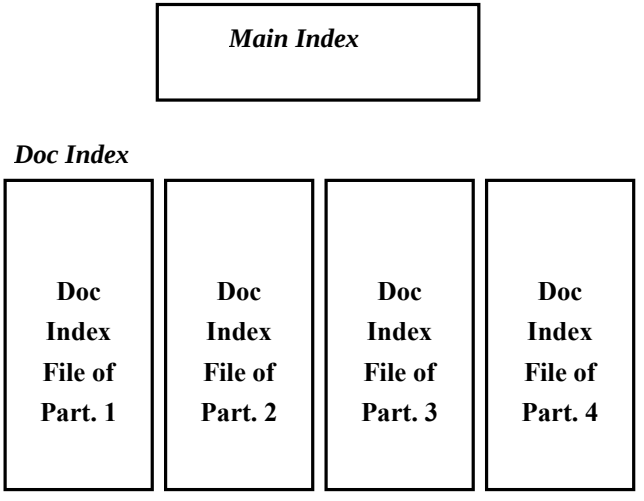
存储器的概述

除了通过**Storage path**参数来指定工作目录外，DBMaster还会在此目录下产生一个子目录，用于管理不同的反向全文索引。由于每一个反向全文索引都有唯一的时间版本，所以DBMaster可以利用此特性来产生一个唯一的子目录，用于存储索引文件。命名子目录将在稍后章节描述。反向全文索引也存在局限性：当用户删除一个反向全文索引时，这些子目录和反向文件都不能被回滚。



例如：*\DBMaster5.4*是一个指定的工作路径，我们在此路径下创建一个名为IVF1的反向全文索引，时间版本为1024476670，那么IVF1.1024476670子目录也就随之创建，所有的反向文件都会存储在这个子目录中。反向文件存在三种不同的类型：Single-Byte、Uni-Gram和Bi-Gram，每一个反向文件会依据文本大小而被划分为几个部分。

以下是一个IVF索引文件的概念架构，它分为四个部分。



Inverted-File Structure with Four Partitions

在特征向量和反向文件之间做选择。

特征向量和反向文件之间的选择将由以下几个因素来决定：

1. 索引大小 — 特征全文索引的大小将不能超过**scale**参数设置的比率，整个数据大小的默认值为40%，反向文件索引的平均大小为整个数据大小的1.5倍，也可以依据数据的性质扩大到2—3倍。

2.查询的响应时间 — 在拥有充足内存和执行性能的个人电脑上，即使在处理千兆数据时，用户也可以得到毫秒的响应时间。但是特征全文索引则需要更长的响应时间，特别是在处理非常庞大的数据量时。

3. 数据库的综合 — 反向全文索引不像特征全文索引那样作为BLOB对象存储，它是作为外部文件来存储的。所以用户无法回滚一个已经删除掉的反向全文索引。

试一下这两种类型的全文索引，看看哪个更适合数据特征。比较的结果是：当输入的数据量小于100MB时，特征全文索引的响应速度比较快，并且占用的内存空间较小。

在多个字段上创建全文索引

全文索引可创建在多个字段上，通过使用CONTAINS和连接操作符（||）来执行多字段查询。用户可以查询索引的所有字段或一部分字段，也就是说，要查询的字段必须包含在全文索引的字段列表中。当在字段列表上没有创建全文索引时，用户也可以进行多字段查询，但是全文索引将不能使用。

在多个字段上查询，逻辑上等于先将所有字段的数据进行合并再执行查询。

➲ **示例1**

在表**tb_document**的**author**、**subject**和**content**字段上，创建一个反向全文索引**ivfidx_multiple**：

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_multiple ON
tb_document(author,subject,content);

dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || subject || content, 'reagan');
```

➲ **示例2**

查询部分字段列表：

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || content, 'reagan');

dmSQL> SELECT author FROM tb_document WHERE CONTAINS(subject, 'reagan');

dmSQL> SELECT author FROM tb_document WHERE subject MATCH 'reagan';
```

➲ **示例3**

此例中，字段**subject**包含在全文索引**ivfidx_multiple**中，但是**abstract**没有，所以此查询将无法使用全文索引。

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(subject || abstract, 'reagan');  // no text index used
```

➲ **示例4**

此例说明了在多字段上查询的行为。

```
dmSQL> CREATE TABLE tb_example (c1 char(20), c2 char (20), c3 serial);

dmSQL> INSERT INTO tb_example VALUES('apple orange', 'banana grape');

dmSQL> INSERT INTO tb_example VALUES('grape orange', null);

dmSQL> CREATE TEXT INDEX ivfidx_example ONtb_example (c1, c2);

dmSQL> SELECT c3 FROM tb_example WHERE CONTAINS (c1 || c2, 'apple');

     C3

===========

     1

 1 rows selected

dmSQL> SELECT c3 FROM tb_example WHERE CONTAINS (c1 || c2, 'orange &
grape');

     C3

===========

     1

     2

 2 rows selected
```

在媒体类型上创建全文索引

DBMaster的大对象字段可以储存成多媒体类型的数据。例如：LONG VARBINARY类型的字段可以识别出Microsoft Word文件，所以DBMaster可适时地调用函数在Microsoft Word文件上执行一个全文索引。DBMaster还提供了可以将一些多媒体类型转换为纯文本类型的UDF，以及获得多媒体类型的UDF。表6-1概述了可利用的不同媒体类型以及相应的SQL命令。

多媒体类型	数据类型	文件类型
Microsoft Word	MsWordType	MsWordFileType
HTML	HtmlType	HtmlFileType
XML	XmlType	XmlFileType
Microsoft PowerPoint	MsPPTType	MsPPTFileType
Microsoft Excel	MsExcelType	MsExcelFileType
PDF	PDFType	PDFFileType

表6-1 多媒体类型和相关的SQL命令

在DBMaster内部，MsWordType、MsPPTType、MsExcelType和PDFType被认为是LONG VARBINARY类型的对象；HtmlType和XmlType被认为LONG VARCHAR类型的对象；而MsWordFileType、HtmlFileType、XmlFileType、MsPPTFileType、MsExcelFileType和PDFFileType被认为是FILE类型的对象。

在媒体类型的字段上进行全文索引搜索

用户可以在多媒体类型上创建全文索引以及执行全文索引搜索。但是必须首先将多媒体类型转换为纯文本形式。DBMaster并不识别新的媒体类型，所以不能将新的类型转换为纯文本形式，也不能在新类型上执行全文索引搜索。

DBMaster提供以下所列的一些可以将某些多媒体类型转换为纯文本形式的UDF：

DOC、XLS、PPT、HTM和PDF。

- DOCTOTXT(BLOB) RETURNS NCLOB;
- XLSTOTXT(BLOB) RETURNS NCLOB;
- PPTTOTXT(BLOB) RETURNS NCLOB;
- HTMTOTXT(CLOB) RETURNS CLOB;
- PDFTOTXT(BLOB) RETURNS NCLOB;

用户可以使用MATCH和CONTAINS在媒体类型的字段上执行一个全文索引，就像在普通字段上执行全文索引一样。

➲ **示例1**

将一个 PowerPoint文件转换为一个临时BLOB文件，该文件将BLOB的纯文本形式保存为Unicode。

```
dmSQL> CREATE TABLE tb_ppt(pptfile LONG VARBINARY);

dmSQL> INSERT INTO tb_ppt VALUES (?);

dmSQL/Val> &e:\udf\pptfile\pfile.ppt;

dmSQL/Val>END;
dmSQL> SELECT PPTTOTXT(pptfile) FROM tb_ppt;
```

➲ **示例2**

创建一个拥有MS Word类型字段的表，同时插入一些数据并且执行搜索操作。

```
dmSQL> CREATE TABLE tb_minutes(id INT, doc MsWordFileType);

dmSQL> INSERT INTO tb_minutes VALUES(1, 'c:\meeting\20020403-1.doc');

dmSQL> SELECT id FROM tb_minutes WHERE doc MATCH 'Jeff';

    id
```

```
==========
         1
 1 rows selected
```

➲ **示例3**

在表**tb_minutes**的**doc**字段上创建一个特征全文索引，并且执行搜索操作。

```
dmSQL> CREATE TEXT INDEX tidx doc on tb_minutes(doc);
dmSQL> SELECT id FROM tb_minutes WHERE doc MATCH 'Jeff';
    id
==========
         1
 1 rows selected
```

检查字段数据的媒体类型

媒体类型的字段可以包含不同类型的数据，当媒体字段在插入和更新数据时，DBMaster会验证数据的内容。DBMaster提供了一个内建函数CHECKMEDIAFORMAT来检查字段数据是否和指定的媒体类型相匹配。如果类型匹配，那么函数将返回1，否则将返回0。DBMaster支持office 2007-2010的多媒体类型。

DBMaster支持下列的媒体类型格式：DOC、XLS、PPT、HTM和PDF。

- **DOC**： Microsoft word 文件
- **XLS**： Microsoft Excel 文件
- **PPT**： Microsoft Power Point 文件
- **HTM**：超文本标记语言
- **PDF**： 便携式文件格式

注意 *DBMaster支持的PDF格式为1.2-1.7。*

➲ **示例**

检查媒体类型格式是否正确：

```
dmSQL> CREATE TABLE tb_checkmedia(note LONG VARBINARY);
dmSQL> INSERT INTO tb_checkmedia VALUES(?);
dmSQL/Val> &E:\DOCS\Media.doc;
dmSQL/Val> &E:\DOCS\Media2007.docx;
dmSQL/Val> END;
dmSQL> SELECT checkmediaformat(note,'doc') FROM tb_checkmedia;
```

- 结果将返回0、1或者NULL。
- 当BLOB的内容与指定的多媒体格式匹配时，将返回1。
- 当BLOB的内容与指定的多媒体格式不匹配时，将返回0。
- 当BLOB的内容为空时，返回 NULL。

当表中字段的定义使用原始的媒体类型（系统定义域）并且媒体格式不正确时，便不能成功移植。为了解决这样的问题，可以删除不正确的媒体类型数据或将数据类型改变为CLOB或BLOB类型来避免检查媒体格式这一步骤。

删除全文索引

您可以使用JDBA工具或dmSQL的DROP TEXT INDEX命令来删除全文索引。

➲ **示例**

从表**tb_staff**中删除一个全文索引**tidx_name**：

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_staff;
```

重建全文索引

与一般索引不同，全文索引并不会随着记录的增加或修改而自动同步更新，因此我们需要对它进行手动更新；也就是说，在您重建全文索引后，再新增或修改的数据将无法通过全文索引找到。

➲ 示例

```
dmSQL> CREATE TABLE tb_song (id INT, name VARCHAR (20));

dmSQL> INSERT INTO tb_song VALUES(1,'Endless Love');

1 rows inserted

dmSQL> CREATE TEXT INDEX tidx_name ON tb_song(name);

dmSQL> INSERT INTO tb_song VALUES(2,'Love Story');

1 rows inserted

dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';

     id              name

=========== ====================

          1 Endless Love

 1 rows selected
```

上例说明了全文索引未同步更新的现象，符合条件的记录有两笔，但只找到了建立全文索引前的第一笔数据。

快速重建全文索引

DBMaster提供了一个快速重建全文索引的指令REBUILD TEXT INDEX。此指令会针对新增的和更改的记录创建特征向量，并将新的特征向量放在全文索引尾部。如果表内只有少数的文件被更改，那么您可以使用REBUILD TEXT INDEX <index_name> INCREMENTAL指令来快速的重建索引。

➲ **示例**

快速重建全文索引并显示重建结果：

```
dmSQL> REBUILD TEXT INDEX tidx_name FOR tb_song;

dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';

     id              name

=========== ====================

          1 Endless Love

          2 Love Story

 2 rows selected
```

完整重建全文索引

如果有大量的文件被更改或删除，您可以使用REBUILD TEXT INDEX <index_name> FULL命令，根据索引的原始类型（特征向量或反向文件）和参数来完整的重建全文索引。

➲ **示例1**

在表**tb_song**上完整重建全文索引**tidx_name**：

```
dmSQL> REBUILD TEXT INDEX tidx_name FULL FOR tb_song;
```

如果您要更改全文索引的参数或使用不同的全文索引类型，必须先将它删除再重新建立全文索引。

➲ **示例2**

在表**tb_song**上重建**tidx_name**特征向量全文索引，将它作为一个反向文件索引：

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;

dmSQL> CREATE IVF TEXT INDEX tidx_name ON tb_song(name);
```

➲ **示例3**

在表**tb_song**上完整重建一个总大小为60MB的特征向量全文索引**tidx_name**。

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;

dmSQL> CREATE TEXT INDEX tidx_name FROM tb_song(name) TOTAL TEXT SIZE 60
MB;
```

布尔字符搜索

MATCH指令除了可以查询一般的字符串外，还可以查询DBMaster提供的布尔（Boolean）字符串。

以下是搜索模式中用户可以使用的布尔（Boolean）字符：

‘&’ – 与（AND）

‘|’ – 或（OR）

‘-’ – 非（EXCLUDE）

‘(’ – 左括号

‘)’ – 右括号

布尔（ Boolean ）字符的优先级别是：bracket > EXCLUDE > AND > OR。当您使用布尔字符时，夹在布尔运算符之中的所有字符都被视为一个独立的搜索字符串。例如：如果MATCH的条件是“coffee | tea | apple juice”，那么搜索的条件包括是“coffee”或“tea”或是“apple juice”。

➲ **示例1**

在**tb_song**表中搜索包含‘love’和‘friend’的记录：

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love & friend';
```

➲ **示例2**

在**tb_song**表中搜索包括‘love’ 或‘friend’，但不包括‘endless love’的记录：

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH '(love | friend) - endless
love';
```

➲ **示例3**

使用SQL指令的布尔逻辑算子（如AND、OR、NOT）加上一般的搜索字符串也可以达到MATCH中布尔运算的功能，但是前者可能无法使用或只使用到部份的全文索引能力，所以执行的性能会降低很多。比如下列指令将会对字符串'friend'作全文索引检索，而对字符串'love'作非全文索引检索：

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love' AND name MATCH
'friend';
```

模糊查找

有时用户希望能使用一种不精确的查找模式。如果数据库只允许查找完全匹配的词组，那么当用户查找 'William Clinton' 时将无法找到像 'William Jefferson Clinton' 这样的词组，反之亦然。 当然，当查询像 'William & Clinton' 这样的布尔表达式时，也会返回很多不相干的结果。DBMaster提供了一种模糊查找机制，允许用户进行不精确的查找而不会返回过多不相干的结果。

模糊表达式通常由'?'（问号）来引导，如'?intel pentium'，我们可以看到在查找到的目标文本中，此模糊表达式被四个单词分开。它的查找结果如下：'Intel will release its 1GHz Pentium III processor'，如果查找的关键字是'?amd k7 athlon'，将会返回如下结果 'AMD has renamed its K7 processor as Athlon'。

在查找的结果集中也会丢失大量关键字。例如：要查找的关键字是 '?William Jefferson Clinton'，将会返回如下结果：'William Clinton' 和 'William J Clinton'。但是，要查找的第一个关键字必须出现。假设要查找的表达式是 '?William Clinton'，那么在查找的结果集中，将不会出现'Bill Clinton'。

模糊表达式可以和其它文本的布尔运算结合起来使用。

➲ 示例

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '?intel
pentium & ?amd k6';

dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore
| ?george bush';
```

模糊表达式中不能包含任何运算符。所以表达式 '?intel pentium &? amd k6' 将被看成是'(?intel pentium) & (?amd k6)'。并且'?(intel & pentium)'会产生一个错误。

相近查找

模糊查找允许用户执行不精确的查询，而不会返回过多不相干的查找结果。相近查找类似于模糊查找，但是比模糊查找更精确，它能确保查询结果中出现所有要查找的关键字。近似表达式通常由 '~'（代字号）来引导。例如：查找表达式 '~amd sales 1ghz athlon'，将会返回如下结果：'AMD announced quarterly sales of its 1ghz Athlon chip'，但不会返回下列结果：'AMD announced quarterly sales of its Athlon chip'。

相近查找表达式可以和其它文本的布尔表达式结合起来使用。

➲ 示例

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '~intel
pentium & ~amd k6';

dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore |
~george bush';
```

模糊/相近查找的规则

以下四个规则可应用于查找的结果字符串中。

1. 在查找的结果集中，必须出现查找字串的第一个关键字。例如：要查找 '?William Clinton'不会返回如下结果：'Bill Clinton'。

2. 查找的几个关键字可以被一些单词分开 (“近似的”)，例如：要查找关键字 '?intel pentium'将会返回如下结果：'Intel will release its 1 GHz Pentium III processor'，如果要查找关键字 '?amd k7 athlon'将会返回下列结果：'AMD has renamed its K7 processor as Athlon'。在当前的结果集中，允许出现在两个关键字中的最多字数为4个。

3. 在查找的结果集中会缺少一些关键字。例如：当查找 '?William Jefferson Clinton' 时，我们会得到如下结果：'William Clinton' 和 'William J Clinton'。允许缺少的最多字数可由下列公式来计算：

max_miss = num_words - round(num_words * threshold)

当前阈值（threshold）为0.75。

4. 所有的查询结果都会以关键字的初始顺序出现。例如：要查找关键字 '?amd 1ghz k7 athlon'将会返回如下结果：'AMD will announce 1GHz Athlon'，但是不会返回下列结果：'AMD Athlon, formerly known as K7'。

相近查找表达式是由 '~'（代字号）来引导的。例如：'~intel pentium'。相近查找是一种特殊情况的模糊查找，它可以应用以上的第1、2、4条规则，但是不支持第3条规则。也就是说，在查找的结果中，不允许缺少关键字。

用户定义的忽略词（Stopword）

和基于关键字查找的检索相反，全文检索系统将查找文档中除忽略词之外的所有字。忽略词就是全文检索系统规定在查找和获取数据过程中要忽略的词语，用以防止检索出无关的数据。通常，一个忽略词列表包括英文中的文章、代名词、形容词、副词和前置词（the、they、very、not、of……），您也可以将这个规则应用到中文或任何一种双字节编码的文本中，例如：的、呢、啊、哈……。

忽略词列表的查找路径

DB_StpWd = <字符串>

此关键字定义了位于DBMaster安装目录的*shared / stopword*子目录中忽略词列表定义文件的名称。忽略词列表定义文件是一个纯文本文件，它可以影响DBMaster的全文检索结果。在数据库创建和查询全文检索时可用到此关键字。如果没有定义这个关键字，那么数据库在搜索预定义的忽略词列表定义时，将依据Lcode来查询。

默认值：

DB_LCode	忽略词列表定义
0 英文 （ASCII）	en.tab
1 繁体中文 （BIG5）	tw.tab
2 日文 （Shift JIS + Half Corner）	jp.tab
3 简体中文 （GB）	cn.tab
4 拉丁语1代码（ISO-8859-1）	en.tab
5 拉丁语2代码（ISO-8859-2）	en.tab
6 斯拉夫语代码（ISO-8859-5）	en.tab
7 希腊语代码（ISO-8859-7）	en.tab
8 日文（EUC-JP）	jp.tab
9 简体中文 （GB18030）	cn.tab
10 Unicode （UTF-8）	en.tab

有效范围： 用户定义的忽略词列表定义文件的文件名称

参考： **DB_LCode**

使用场所： 服务器端 （只用于创建和搜索全文检索时）

默认的忽略词列表

- 如果您没有对该关键字进行配置，那么DBMaster将载入基于LCODE并在预定义文件中的默认忽略词。使用DBMaster先前版本的用户也可以使用此特征。
- DBMaster在本地和安装目录搜索预定义文件。

用户定义的忽略词列表

- 您可以通过配置文件中的**DB_StpWd**关键字来指定一个忽略词列表。DBMaster在创建一个全文检索或从全文检索中查找对象时，会载入该文件。
- DBMaster在本地目录或用户指定的目录和安装目录搜索该文件。

禁止忽略词列表

有两种方法可以禁止一个忽略词列表。

- 重命名或删除预定义文件。
- 在配置文件中定义一个不存在的忽略词列表。

6.8 内存表管理

在DBMaster中，内存表的大多数功能和用途与永久表是一致的，不同之处在于内存表是临时表，它们的生命周期是以数据库连接为基础的。也就是说，内存表只能存在于已连接的数据库中。譬如当用户切断数据库连接时，数据库会记录退出系统的时间，但是该用户的内存表和内存表中的内容将会丢失。内存表只有在数据库连接时才能看见，一旦连接断开，这些内存表将不再可见。和固定表不同的是，内存表只能存在于创建它们的机器内存中，它们不能被群组使用，并且只能对该表进行选择和插入数据，不能进行更新和删除动作。内存表支持事务控制，如提交、回滚、定义保存点、回滚到保存点和内部保存点。

您可以使用dmSQL的CREATE MEMORY TABLE命令来创建一个内存表，有关语法的详细信息和创建表的SQL命令，可参考*SQL命令与函数参考手册*。

➲ **示例**

创建内存表**tb_memory**：

```
dmSQL> CREATE MEMORY TABLE tb_memory (id INT, name CHAR(10), brithday
DATE);
```

哈希索引管理

内存表都存储在创建它们的存储器上，因此内存表不像其它类型的表一样支持B-tree结构。哈希索引只能创建在内存表上，这样用户就可以在内存表上创建哈希索引了。使用哈希索引的好处在于用户可以快速访问存储在哈希索引中的数据，哈希索引还可以提高比较运算和连接运算的性能。用户可以使用CREATE HASH INDEX *index_name* ON *table_name* (*column_name*, …) [*bucket n*]命令在表上创建一个哈希索引，其中的*index_name*代表创建哈希索引的名字，*table_name*为内存表的名字，*column_name*为内存表中相关联的字段名（该字段不能指定排序顺序：asc/desc）。bucket n用于设置创建哈希表的数组大小。例如在内存表

tb_memory的**id**和**name**字段上创建一个哈希索引**hidx**，数组大小为31。

➲ 示例

依上例在内存表**tb_memory**上创建一个哈希索引 **hidx**：

```
dmSQL> CREATE HASH INDEX hidx ON tb_memory (id, name) bucket 31;
```

6.9 数据完整性管理

您可以设定数据的条件限制和规则来确保数据库中数据的完整性。我们要确定某个输入值在有效范围内，例如，一个新进员工的年龄一定是在16到90岁之间。这种限制就是数据完整性的一个例子。

通常，数据完整性的类型可以分为以下几种：

非空（Not Null）

如果没有特别指定的话，表中的字段值都允许空值（NULL）。如果一个字段已被设为不允许空值（NOT NULL），那么在新增或更新数据时就不允许向数据库中输入空值。

唯一性索引（Unique Indexes）

关于唯一性索引，请参考*索引管理*章节，利用唯一性索引可以确保一个表的指定字段或字段集上，不会出现相同的数据（除了空值NULL）。

唯一性约束（Unique Constraints）

您可以在一个字段，一组字段或一张完整的表上定义唯一性约束，它能够确保字段中的每一行都拥有不同的值。在设置了唯一性约束的字段上不会出现相同的字段值。

➲ **示例**

在表**tb_student**的**Name**字段上创建唯一性约束：

```
dmSQL> CREATE TABLE tb_student (Name CHAR(50) CONSTRAINT u UNIQUE,
mathematics SMALLINT);
```

条件限制（Check Constraints）

您可以设定条件限制来核对使用者在新增（INSERT）或更新（UPDATE）这张表的每一笔记录时，是否都符合字段的设定条件。如果不符合，所执行的新增或更新操作就会失败。

一般来说，您可以在一个字段（字段约束）或一组字段（表约束）上设置条件限制。

字段约束（COLUMN CONSTRAINTS）

所谓字段约束，是指定义在一个字段上的限制条件，与表中的其它字段无关。当新增或更新数据时，数据库会核对在每一个字段上的值是否符合字段约束。

表约束（TABLE CONSTRAINTS）

表约束定义在一组字段上。当新增或更新数据时，数据库会首先核对每个字段的条件是否符合，最后再核对表约束是否符合。如果这些条件都符合，数据才允许被更新或新增。

➲ **示例1**

下面这个SQL指令就是一个如何设定字段约束和表约束的例子：

```
dmSQL> CREATE TABLE tb_student (mathematics SMALLINT
                                    CHECK VALUE >= 0 AND VALUE <= 100,
                               chemistry SMALLINT
                                    CHECK VALUE >= 0 AND VALUE <= 100,
                                     CHECK mathematics + chemistry <=
200);
```

➲ **示例2**

下例说明了如何使用SQL99语法设置字段约束和表约束：

```
dmSQL> CREATE TABLE tb_student (mathematics SMALLINT
              CONSTRAINT con_math CHECK VALUE >= 0 AND VALUE <= 100,
```

```
                        chemistry SMALLINT
        CONSTRAINT con_chem CHECK VALUE >= 0 AND VALUE <= 100,
        CONSTRAINT con_sum CHECK mathematics + chemistry <= 200);
```

在字段约束中，关键字VALUE是用来指定字段的值。但是在表约束中，则是利用字段名称来代表字段的值。

主键（Primary Keys）

一张表只能定义一个主键（primary key），主键可以包含一个或一个以上的字段，并且这些字段的值必须是唯一的。除了主键的字段不允许为空外，其它方面主键和唯一性索引很相似。创建一个主键时，DBMaster会在这张表中创建一个名为**PrimaryKey**的唯一性索引。创建一张表后，您可以使用ALTER TABLE命令来向表中新增或更改一个主键，只要定义主键的字段符合唯一值和非空的条件。此外，改动后的主键内容可以储存在另一个表空间中。

创建主键

➲ **示例1**

在表**tb_student**的**ID**字段上创建一个主键：

```
dmSQL> CREATE TABLE tb_student (
          ID     INTEGER  PRIMARY KEY,
          name   CHAR(30),
          nation CHAR(20)
       );
```

➲ **示例2**

在表空间**ts_reg**的**tb_student**表的**ID**和**name**字段上创建复合主键：

```
dmSQL> CREATE TABLE tb_student (
          ID     INTEGER,
          name   CHAR(30),
```

```
          nation CHAR(20),
          PRIMARY KEY (ID, name)
        ) in ts_reg;
```

➲ **示例3**

向**tb_student**表中添加一个主键：

```
dmSQL> ALTER TABLE tb_student PRIMARY KEY (ID , name);
```

➲ **示例4**

使用标准SQL99语法，在表空间**ts_reg**的**tb_student**表上添加主键PK1：

```
dmSQL> ALTER TABLE tb_student ADD CONSTRAINT PK1 PRIMARY KEY (ID , name)
IN ts_reg;
```

➲ **示例5**

使用标准SQL99语法，在表**tb_student**的**ID**字段上创建主键：

```
dmSQL> CREATE TABLE tb_student (
          ID     INTEGER  CONSTRAINT pk1 PRIMARY KEY,
          name   CHAR(30),
          nation CHAR(20)
        );
```

删除主键

您可以删除一个无用的主键。但在删除主键之前，必须先删除其它表中所有参考到这个主键的外键（foreign key）。

➲ **示例**

删除表**tb_student**的主键：

```
dmSQL> ALTER TABLE tb_student DROP PRIMARY KEY;
```

外键（参照完整性）

所谓外键就是一张表的某个字段值必须等于其它表的主键。我们可以把外键视为这两张表之间的关联。您可以在表的一个或一组字段上定义外键，并且用这些字段值参考到另一张表的一个或一组字段；而被参考的字段必须存在一个主键或唯一性索引，但不能为空值（NULL）。

因为外键的值会参考到另一张表上的字段值，所以在您新增一笔数据时，必须确定被参考的字段上已经存在该键值。否则，用户将不允许插入该笔记录。另外，在删除被参考键值之前所有外键必须删除。

您也可以随时创建或删除主键和外键。如果您在一张已有数据的表上创建主键，DBMaster会核对表上主键值是否有重复；如果您在一张已有数据的表上创建外键，DBMaster会校验现有数据参考的键值是否存在于被参考的表中。

创建外键

在创建外键时，您必须定义外键所参考的表和字段。参考和被参考的字段应当相互对应，字段数据类型和长度必须相同。被参考字段（定义为主键或唯一索引）应不为空，而参考字段（定义为外键）字段值可以是非空或空值。如果您没有定义外键所对应的主表字段，DBMaster就会使用主表的主键来当作对应的字段。您可以使用JDBA工具的创建外键向导或dmSQL的FOREIGN KEY选项来创建外键。

➲ **示例1**

在**tb_salary**表上创建一个外键**f1**，而这个外键对应于表**tb_staff**的主键，包括字段**ID**和**name**：

```
dmSQL> ALTER TABLE tb_salary FOREIGN KEY f1(ID, name) REFERENCES
tb_staff;
```

➲ **示例2**

您也可以在创建**tb_salary**表的同时创建一个外键**f1**：

```
dmSQL> CREATE TABLE tb_salary (
         ID    INT,
```

```
        name  CHAR(30),
        basepay INT,
        bonus   INT,
        tax INT,
      FOREIGN KEY f1 (ID, name) REFERENCES tb_staff);
```

➲ **示例3**

使用标准的SQL99语法，在创建**tb_example**表的同时创建外键**fk1**：

```
dmSQL> CREATE TABLE tb_example (
      c1 int,
      c2 int CONSTRAINT fk1 REFERENCES tb_other (c1) ON DELETE SET
NULL);
```

如果表**tb_staff**已经创建了主键，那么在为其它表创建外键时，就无需指定被参照的字段。

删除外键

如果外键所定义的关系已经不再需要，您可以使用JDBA工具或dmSQL的 DROP FOREIGN KEY命令来删除外键。

➲ **示例**

删除表**tb_salary**的外键：

```
dmSQL> ALTER TABLE tb_salary DROP FOREIGN KEY f1;
```

6.10 序列数管理

DBMaster提供了一个可自动产生序列数（serial number）的功能。此功能特别用于多用户环境，在没有磁盘I/O或事务堵塞的消耗下，产生和返回唯一的序列数字。

DBMaster中的序列数是一个32位的整数，一张表只能将一个字段的数据类型定义成序列数值（SERIAL）类型。

当您创建表时，您可以设定序列（SERIAL）字段的起始数字；如果您没有设定的话，默认值为1。

如果在您新增一笔数据时将序列字段设置为空值，DBMaster就会自动产生序列数。但是如果您对序列字段输入一个整数值的话，DBMaster就不会自动产生序列数，而直接使用您输入的数值。如果您输入的整数值大于数据库中最后的序列数，DBMaster会利用您输入的新值来重新设置序列数。 序列字段（SERIAL）不能定义默认值或条件限制。

创建序列数字段

SERIAL类型的字段可以通过JDBA工具或dmSQL来定义，序列数的数据类型必须用SERIAL或BIGSERIAL关键字来表示，也可以设定它的起始值。

➲ **示例**

将**tb_staff**表上**ID**字段设置为SERIAL类型，并且将它的起始值设定为1001：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                   ID SERIAL(1001),
                                 name CHAR(30) NOT NULL,
                             joinDate DATE DEFAULT CURDATE(),
                               height FLOAT,
```

```
                                degree VARCHAR(200)) IN ts_reg;
```

序列数的产生

当您输入的SERIAL类型的字段为空值时，DBMaster会自动产生序列数。

➲ **示例**

向**tb_staff**表中新增一笔数据时，若希望DBMaster自动产生序列数，可以执行以下SQL指令：

```
dmSQL> INSERT INTO tb_staff VALUES
              ('U.S.A', NULL, 'Jeff', 6.6, 'Director', NULL);
```

检索序列数

DBMaster会将最后一次产生的序列数存放到系统表SYSCONINFO的LAST_SERIAL字段中。当您新插入一笔记录后，您可以从LAST_SERIAL字段中来检索此记录的序列数。

➲ **示例**

下例说明了如何获得新插入记录的序列数：

```
dmSQL> SELECT LAST_SERIAL FROM SYSCONINFO;
LAST_SERIAL
===========
        200
 1 rows selected
```

重置序列数

用户可以重置序列数字段的计数。它允许您在不更改表的前提下启动一个新的序列数。

➲ 示例

将**tb_staff**表的当前序列值更改为3000：

```
dmSQL> ALTER TABLE tb_staff SET SERIAL 3000;
```

6.11 定义域管理

所谓定义域（domain）是一种定义字段的完整性限制，定义域可以设定字段的数据类型，还可以设定字段的默认值或字段限制。当在一个字段上设定定义域时，这个字段的属性就会和定义域的属性相同（数据类型、默认值和字段限制），您就无需再特别设定这些条件了。

用定义域来设定字段的默认值和约束条件和使用标准的字段定义来设定它们所达到的效果是一样的。如果用户设置了字段的默认值，那么它将替换掉定义域中的默认值。

如果设定的定义域中已定义了字段限制，而您在设定字段时，再设定另一个条件限制。那么这个字段的条件限制就变成旧的定义加上新的条件限制，所以必须确定新的条件和旧的条件并不互相冲突。

DBMaster不检测约束的冲突，所以就有可能存在定义的字段限制不允许用户输入任何值的可能性。除了序列数据类型以外，定义域可以用来定义任何类型的数据。

创建定义域

一个定义域必须设定定义域的名称，可选是否需要设定默认值和条件限制。例如：如果用户想对表中的字段设定相同的格式（如电影、CD或是录像带的名称），如数值类型为VARCHAR、长度不能超过35个字符、不允许输入空值。这时您就可以利用CREATE DOMAIN命令来创建定义域。这样在以后创建表时，您就可以对类似的字段用相同的定义域设定。定义域可以通过JDBA工具或dmSQL的CREATE DOMAIN命令来创建。

➲ **示例1**

定义域中的关键字VALUE是用来代表字段中的数据值。定义好的定义域title_type可用于CREATE TABLE语句中：

```
dmSQL> CREATE DOMAIN title_type VARCHAR(35) CHECK VALUE IS NOT NULL;
```

➲ **示例2**

在CREATE TABLE语句中定义一个字段值：

```
dmSQL> CREATE TABLE movie_titles (title title_type, ..., ...);
```

通过TEXT CONVERTER创建定义域

多媒体类型是一种带有特殊字符多媒体格式的定义域，用户可以在CREATE DOMAIN子句中通过TEXT CONVERTER语法创建定义域。当用户在定义域中指定了TEXT CONVERTER语法，DBMaster将通过TEXT CONVERTER表达式将CLOB、NCLOB、BLOB或FILE数据转换成纯文本格式，用来创建全文索引和PURETEXT() UDF。TEXT CONVERTER函数名只能包含一个BLOB相关类型的自变量，返回的类型也必须是CLOB 或 NCLOB数据类型，否则DBMaster将返回错误。用户做多可使用TEXT CONVERTER语法创建**32767**个定义域。

注意 *在TEXT CONVERTER子句中，用户不能指定表达式、不能指定没有参数或参数多于一个的函数。*

➲ **示例**

定义一个名称为**MSWORDTYPE**的定义域：

```
dmSQL> CREATE DOMAIN MSWORDTYPE AS BLOB
       TEXT CONVERTER DOCTOTXT
       CHECK VALUE IS NULL OR CHECKMEDIAFORMAT(VALUE,'DOC') = 1;
```

使用定义域**MSWORDTYPE**创建一个表 **tb_MT**：

```
dmSQL> CREATE TABLE tb_MT (C1 MSWORDTYPE);
```

删除定义域

定义域只能在没有字段使用它时才能被删除。您可以使用JDBA工具或dmSQL的DROP DOMAIN命令来删除定义域。

➲ **示例**

使用DROP DOMAIN命令来删除定义域：

```
dmSQL> DROP DOMAIN title_type;
```

6.12 载入/载出对象

有时用户需要将数据库中的数据作为外部文件来储存。DBMaster提供的载入（LOAD）/载出（UNLOAD）命令就可以实现这个功能，载出对象并非将对象从数据库中删除，它们只是简单的作为一个或多个外部文件来储存。当对象载入到数据库中时，此对象的结构也会重建。

载出对象

载出（Unload）命令是dmSQL提供的一种工具，用于将数据库中的内容转换成一个外部文件。当载出操作成功后，dmSQL会产生两个文本文件： 一个用来储存脚本，扩展名为**.s0**，用于建立数据库对象；另一个用来存储BLOB数据，扩展名为**.bn**。

载出命令有8个选项：载出数据库、载出表、载出表结构、载出数据、载出工程、载出模块、载出过程和载出过程定义。载出对象要求用户拥有该对象的SELECT权限，例如：如果用户拥有表的SELECT权限，那么只有该用户可以载出此表中的内容。只有拥有DBA、SYSDBA或SYSADM权限的用户才可以载出数据库。

载出数据库 （UNLOAD [DB | DATABASE]）

拥有DBA、SYSDBA或SYSADM权限的用户可以将数据库中的内容载出成一个外部文件。此文件包括一些安全性、表空间、定义、索引、同义字和数据等信息。针对每一个数据库，dmSQL都会产生至少两个外部文件：一个是脚本文件、一个是BLOB数据文件。

➲ **示例1**

```
dmSQL> UNLOAD DB TO empdb;
```

上例中外部文本文件的名称为empdb，默认状态下dmSQL将在当前目录下创建这些文件。通过以上命令，DBMaster至少创建了两个文本文件**empdb.s0**和**empdb.b0**。如果载出的BLOB文件**empdb.b0**超出了操作系统所允许的大小，那么dmSQL将继续产生**empdb.b1**、

empdb.b2、……、**empdb.bn**文件，直到最大限制99。在此期间，dmSQL会一直产生一个脚本文件**empdb.s0**，它也受操作系统的大小限制。

然而，如果用户在使用命令UNLOAD DB TO *file_name*之前下达了如下命令：SET UNLOAD EXTERNAL '*connection_string*'（*connection_string*的格式为 "DSN=<*数据库名*>； UID=<*用户名*>；PWD=<*密码*>;"），dmSQL将不会载出数据到脚本文件empdb.s0中，而会在empdb.s0文件中产生"set external db '*connection_string*'"，并且载出的表数据将会出现"load external db from 'select * from *external_table_name*' into *local_table_name*"的提示语句。示例如下：

➲ **示例2**

```
dmSQL> SET UNLOAD EXTERNAL 'DSN=DBSAMPLE5;UID=SYSADM;PWD=;';
dmSQL> UNLOAD DB TO empdb;
```

脚本文件empdb.s0如下：

```
…
set external db 'DSN= DBSAMPLE5;UID=SYSADM;PWD=;';
create table Lauser1.Latb3 (
 c1  SMALLINT default null ,
 c2  FLOAT default null ,
 c3  DOUBLE default null ,
 c4  DECIMAL(10, 3) default null ,
 c5  CHAR(10) default null ,
 c6  BINARY(12) default null )
 in DEFTABLESPACE  lock mode page  fillfactor 100 ;
load external database from 'select * from Lauser1.Latb3' into
Lauser1.Latb3;
create  index idx31 on Lauser1.Latb3 ( c1 asc ) in DEFTABLESPACE;
create  index idx32 on Lauser1.Latb3 ( c3 desc ) in DEFTABLESPACE;
```

```
create  index idx33 on Lauser1.Latb3 ( c5 asc ) in DEFTABLESPACE;

…
```

载出表（UNLOAD TABLE）

UNLOAD TABLE命令可以将表载出成外部文件，并且记录表中的定义、同义字、索引、主键、外键和数据。对所有者和表名称可以使用通配符“_”和“%”，对应于DOS中的“?”和“*”。“_”代表一个字符，“%”代表一组字符。

载出表结构（UNLOAD SCHEMA）

载出表结构的作用和载出表的作用非常相似。它只能载出表中的定义而不能载出表中的数据。并且和载出表（UNLOAD TABLE）选项一样，可以使用同样的通配符。

载出数据（UNLOAD DATA）

此选项将从表中载出所有数据，但不会载出表的定义。UNLOAD DATA使用和前面的选项一样的通配符，只有对载出表拥有SELECT权限的用户才可以执行此UNLOAD DATA命令。

对载出数据，只有DBMaster 3.6和以后的版本才支持以下这个SQL语法：

```
UNLOAD DATA FROM (select statement) TO file_name
```

如果select statement语句存在关联关系，那么投影字段必须来自于同一张表，您可以执行下例语句，但不允许执行DDL命令以及删除、插入和更新操作。

➲ **示例1**

以下是载出数据的有效语句：

```
dmSQL> UNLOAD DATA FROM (SELECT t1.c1, t1.c2 FROM t1, t2 WHERE t1.c1=
t2.c1) TO f1;
```

➲ **示例2**

以下是载出数据的非法语句：

```
dmSQL> UNLOAD DATA FROM (SELECT t1.c1, t2.c1 FROM t1, t2 WHERE t1.c1 =
t2.c1) TO f1;
```

在所选字段中不允许出现集合或内建函数。

➲ **示例3**

投影字段中不允许使用集合函数或内嵌函数。

以下是载出数据的非法语句：

```
dmSQL> UNLOAD DATA FROM (SELECT avg(c1) FROM t1) TO f1;

dmSQL> UNLOAD DATA FROM (SELECT now()FROM t1) TO f1;
```

允许视图和同义字。

➲ **示例4**

以下是载出数据的有效语句：

```
dmSQL> UNLOAD DATA FROM (SELECT * FROM s1 WHERE c1 > 10) TO f1;

dmSQL> UNLOAD DATA FROM (SELECT * FROM v1 WHERE c1 < 10) TO f1;
```

载出工程（UNLOAD PROJECT）

此选项允许用户将一个ESQL/C项目载出成外部文件。

载出模块（UNLOAD MODULE）

此选项允许用户将一个模块载出成外部文件。

载出过程（UNLOAD [PROC | PROCEDURE]）

此选项允许用户将存储过程载出成外部文件。

载出过程定义（UNLOAD [PROC DEFINITION | PROCEDURE DEFINITION]）

此选项允许用户将存储过程定义载出成外部文件。

➲ **示例1**

以下命令将为当前用户载出表**e tab**；如果表名中存在空格，请为该表名添加双引号：

```
dmSQL> UNLOAD TABLE FROM "e tab" TO empfile;
```

➲ **示例2**

以下命令将为SYSADM用户载出以**emp**开头的表，例如：**emptab**, **empname**等：

```
dmSQL> UNLOAD TABLE FROM SYSADM.emp% TO empfile;
```

➲ **示例3**

以下命令将载出表名中包含**ktab**的表结构：

```
dmSQL> UNLOAD SCHEMA FROM %.ktab TO kfile;
```

您可以在表名中加上转义字符"\"或双引号来载出包含通配符的表名。

➲ **示例4**

以下命令将从**abc%**的表中载出数据：

```
dmSQL> UNLOAD DATA FROM abc\% TO abcfile;

dmSQL> UNLOAD DATA FROM "abc%" to abcfile;
```

载入对象

载入（LOAD）命令是dmSQL提供的一种工具，用于将已经载出的文本文件的数据库对象转换到数据库中。载入命令有7个选项：载入数据库、载入表、载入表结构、载入数据、载入工程、载入模块和载入过程。对一个文件执行载入/载出操作，必须使用同一个选项。例如：从文本文件中载入数据库，而文件是使用数据库的载出选项产生的。当载入一个文本文件时，将<*n*>值设置成自动提交状态。<*n*>的默认值为1000，<*n*> 值的大小将影响载入事务的成功与否和载入操作的执行速度。如果将<*n*>值设置的太大，那么日志文件将会很快的填满并引起事务处理的失败；如果将<*n*>值设置的太小，那么将会增加事务提交的次数并且降低载入的执

行速度。如果在载入程序中出现错误，那么错误信息会记录到日志文件中，系统将使用这些信息来执行取消（undo）操作。日志文件和被载入的外部文件存储在同一个目录下，并且载入过程不会停止。

载入数据库（LOAD [DB | DATABASE]）

使用LOAD [DB | DATABASE]命令可将一个数据库的内容转移至一个新的数据库。首先将数据库载出成一个外部文本文件，然后使用LOAD DB命令从文本文件中载入数据库中的内容。在载入数据库之前，创建一个新的数据库，新数据库的名称可以不同于旧数据库的名称。只有拥有DBA、SYSDBA或SYSADM权限的用户才可以执行这个命令。

然而，如果用户在使用命令UNLOAD DB TO *file_name*之前下达了如下命令：SET UNLOAD EXTERNAL '*connection_string*'(*connection_string* 的格式为 " DSN =<*数据库名*>；UID =<*用户名*>；PWD =<*密码*>;")，dmSQL将不载出数据到脚本文件empdb.s0中。因此，当用户使用该脚本文件载入数据库时，dmSQL将连接ODBC数据源，从中读取数据并直接将读取的数据保存到本地数据库中。dmSQL在脚本文件中使用"set external [database|db] '*connection_string*'"命令来连接外部数据库，若失败，将返回一条错误信息。dmSQL仅保留最后一次外部数据库的连接，因此在设置新连接时，请先断开之前的连接。此外，由于没有断开连接的命令，用户只有在关闭dmSQL时才能断开外部数据库。

如果将LOAD DB设置成SAFE状态，那么数据库会以正常模式运行。在载入过程中如果发生错误，载入程序会回滚到最后一次提交的命令，屏幕上也会显现刚刚发生的出错信息并且将此错误信息写入载入程序的日志文件中。如果将LOAD DB设置成fast模式（此应用在DBMaster3.6以前的版本就存在），那么整个载入过程将工作在无日志模式的状态下。设置 LOAD DB的快速模式将加快载入的过程，但是如果在此过程中发生任何错误，数据库将会在无日志模式下关闭。例如：假如您为载入文件创建了一个表空间，但是在配置文件**dmconfig.ini**中没有指定它，如果您将 LOAD DB设置成safe模式，将会出现下列出错信息：“ERROR（8002）：[DBMaster] 必须在配置文件中输入所需的关键字”，并且载入命令将会回滚。如果您将LOAD DB设置为快速模式，那么将会出现下列错误信息：“ERROR（30017），[DBMaster]在非日志（no-journal）

模式下出现错误，请关闭数据库”。默认的选项值为：SET LOAD DB SAFE。

➲ 示例

自DBMaster3.6以后的版本都可执行SET LOAD DB选项：

```
SET LOAD DB [safe | fast]
```

载入表（LOAD TABLE）

此命令允许从文本文件载入表的内容，包括表结构和数据。当从文本文件中载入表时，请确定此表名是唯一的。

载入表结构（LOAD SCHEMA）

此选项允许用户从一个包含表的文本文件中载入表结构，不包括数据。当从一个文本文件中载入表结构时，请确保表名是唯一的。

载入数据（LOAD DATA）

当从一个外部文件中载入数据时，必须存在一个与此文件对应的表。DBMaster 3.6以前的版本，当载入数据（LOAD DATA）过程中发生错误时，会回滚到最后一次提交的状态。

➲ 示例

DBMaster 3.6和以后的版本支持以下的功能：

```
SET LOAD DATA SKIP [error] | STOP [on error]
```

如果设置了LOAD DATA SKIP ERROR命令，那么在载入数据的过程中将会忽略以下错误信息：

ERROR（401）键值不唯一

ERROR（410）违反参照约束：父键中不存在此值

ERROR（6521）表或视图不存在

ERROR（6002）语法错误

ERROR（6015）输入的SQL命令不完整

这些错误将被忽略，并且载入程序会继续执行后面的命令。以上错误是在载入数据过程中发生的最普遍的错误。当设置了LOAD DATA STOP或STOP ON ERROR选项后，如果发生了错误，整个LOAD命令将被回滚，此选项的默认值为LOAD DATA SKIP ERROR。载入数据过程中发生的所有错误信息都将写入日志文件中。

载入模块（LOAD MODULE）

此选项允许用户从一个外部文件中载入模块。

载入工程（LOAD PROJECT）

此选项允许用户从外部文件载入一个ESQL/C工程。

载入过程 （LOAD [PROC | PROCEDURE]）

此选项允许用户从外部文件载入一个存储过程。

➲ **示例1**

下例说明了如何从**empdb**文件中载入数据库，并且在载入过程中，每100个命令便自动提交一次。与此同时，系统将在同一目录下产生一个名为**empdb.log**的日志文件：

```
dmSQL> LOAD DB FROM empdb 100;
```

➲ **示例2**

下例说明了如何从**empfile**文件中载入表，并且在载入的过程中，每50个命令便自动提交一次：

```
dmSQL> LOAD TABLE FROM empfile 50;
```

➲ **示例3**

下例说明了如何从**datafile**外部文件中载入数据，并且使用默认设置。 每1,000条命令便自动提交一次：

```
dmSQL> LOAD DATA FROM datafile;
```

6.13 浏览系统表

DBMaster把所有模式对象信息都存放在系统表里。要想获取更多有关系统表的信息请参考第21章*系统目录参考*。

模式对象信息	系统目录表名称
表	SYSTABLE
字段	SYSCOLUMN
视图	SYSVIEWDATA
同义字	SYSSYNONYM
索引	SYSINDEX
域	SYSDOMAIN
序列数	SYSCONINFO
系统表	SYSTABLE
系统字段	SYSCOLUMN
系统定义域	SYSDOMAIN

表6-2 系统表中的结构信息

6.14 储存空间的计算

前面已经提过，只有表和索引储存在实际的磁盘空间中。要想有效地管理磁盘空间，就必须估算出这些对象所占用的磁盘空间大小，并决定它们所属的表空间大小。在估算之前，您必须清楚地了解需要多少表空间，每个表空间要储存多少张表以及将来数据库所需的硬件空间大小。

一般来说，如果您把每张表分散储存在不同的表空间，数据存取会比把所有表都存放在同一个表空间效率高。但对系统管理者来说，数据库里如果有太多较小的表空间又很难维护。 所以在规划数据库时，您必须清楚地估计出表空间所需占用的储存空间大小。

表存储空间的计算

以下公式可用于计算表和索引所要占用的存储空间大小：

表大小=数据行大小×数据行数目×1.05

索引大小=索引键大小×数据行数目×1.20

我们可以利用以上这两个公式来计算表和索引的大小，进而推算出存储这些表和索引的表空间所占用的储存空间。其中1.05和1.20是考虑系统资源而估计出的两个值。数据行大小和索引键大小包括数据库内部记录表头（record header）以及页表头（page header）所占用的空间。接下来我们将介绍如何计算数据行和索引键的大小。

数据行大小（ROW SIZE）

在DBMaster中除了BLOB字段外，任何一笔数据的最大长度不得超过3996个字节。包括数据大小和内部记录头长度。

而内部记录的表头长度等于：

内部记录的表头大小 = (字段数目 + 1) × 4

每一种数据类型的长度大小如下：

类型	字段长度
BIGINT	8
BINARY(n)	N
BIGSERIAL	8
SMALLINT	2
INTEGER	4
FLOAT	4
SERIAL	4
DOUBLE	8
DECIMAL(p,s)	[(p+1)/2]+2
TIME	4
DATE	4
TIMESTAMP	11
OID	16
VARCHAR(n)	1-3992n
FILE	20
LONG VARBINARY	48+X
LONG VARCHAR	48+X

表6-3 数据类型和大小

注意 *VARCHAR类型是一种变长的数据类型，用来存储输入的字符。BLOB数据类型的字段长度（LONG VARCHAR或LONG VARBINARY）在数据文件中至少占48个字节，但BLOB字段的实际数据将存放在BLOB文件或数据文件中。如果字段的值是空值（NULL），那么这个字段将不占用任何空间。*

➲ **示例**

创建一张包含5个字段的表。

```
dmSQL> CREATE TABLE tb_staff (ID INTEGER NOT NULL,
```

```
    name CHAR(30) NOT NULL,
  height FLOAT,
  degree VARCHAR(200),
picture LONG VARCHAR);
```

在执行上面这个指令后，为这个表新增数据并且计算记录的长度：

（3001、"Jeff Yang"、175.5、"Stanford PhD."、[pic1]），其中**pic1**为图像。

对象名称	类型	大小
ID	integer	4 bytes
name	char	30 bytes
height	float	4 bytes
degree	varchar	13 bytes
picture	long varchar	48 bytes
row header	—	24 bytes
	Total	**123 bytes**

= (4 + 30 + 4 + 13 + 48) + (5 + 1) × 4 = 123 字节

（3002、"George Wang"、180.0、"NCTU Ms."、NULL）

对象名称	类型	大小
ID	integer	4 bytes
name	char	30 bytes
height	float	4 bytes
degree	varchar	8 bytes
picture	long varchar	0 bytes
row header	—	24 bytes
	Total	**70 bytes**

= (4 + 30 + 4 + 8 + 0) + (5 + 1) × 4 = 70 字节

在您新增或更新数据时，DBMaster会校验每笔数据的长度不得超过MAXTUPLEN [1]字节。在创建表时，DBMaster也会检查这张表可能的最小行长度不得超过MAXTUPLEN个字节。

以上面这个例子来说，这个表最小的记录长度为：

最小的记录长度= (4 + 30 + 0 + 0 + 0) + (5 + 1) × 4 = 58 字节

因为这个表最小的记录长度并没有超过MAXTUPLEN个字节，所以可以成功地建立此表。

[1] MAXTUPLEN: 当数据页的大小为4KB时，该值为 3968；当数据页的大小为8KB时，该值为8064；当数据页的大小为16KB时，该值为16256；当数据页的大小为32KB时，该值为 32640。

索引键的大小

索引键和数据行很类似，它的长度是由索引字段，内部数据记录头和存放内部行标识（row identifier）的16个字节组成，此内部行标识在记录头中占用4个字节。

所以内部记录头的大小等于：

内部记录头的大小=(字段数目+1+1)× 4

举例来说，如果有一个索引是建立在一个SMALLINT数据类型的字段上，那么每个索引键所占的大小为：

索引键大小 = 2 + 16 + (1 + 1 + 1) × 4 = 30 字节

在这个例子中，索引键所在的字段长度占用两个字节，每个关键字的内部行标识（row ID）占用16个字节，还有12个字节用来存放记录头。

预估表空间和表的存储空间大小

现在，我们用下例来说明如何预估表空间和表的存储空间大小。假设有一个表空间包含三张表**A**、**B**、**C**，另外在**A**表上创建一个索引**D**。**A**表的字段类型定义为integer和char(10)；**B**表的字段类型为smallint、char(10)、float 和varchar(200)。**C**表的字段类型为smallint、integer和long varchar。索引**D**是建在**A**表的第一个字段上，**A**、**B**、**C**三表分别包含1500、3000和250笔数据。

数据行和关键字的大小计算如下：假设**B**表的varchar字段的平均长度为80字节，而**C**表的BLOB字段在数据文件所占的长度为48字节：

表A	row size = (4 + 10) + 3 × 4 = 26bytes
表B	row size = (2 + 10 + 4 + 80) + 5 × 4 = 116bytes
表C	row size = (2 + 4 + 48) + 4 × 4 = 70bytes

如果在**C**表每个BLOB字段的长度为9000字节，您可以把BLOB帧（frame）的大小设置为11KB。关于BLOB数据请参考第7章*大型对象管理*。

索引D	keysize = 4 + 16 + 3×4 = 32bytes

数据库的表大小计算如下，注意**A**表的大小必须包括**D**索引的大小。

表A	table size = (26 × 1500 × 1.05) + (32 × 1500 × 1.2) = 98550bytes
表B	table size = 116 × 3000 × 1.05 = 365400bytes
表C	table size = 70 × 250 × 1.05 = 18375bytes

在BLOB文件中，表C的大小为250帧（每一个数据行需要一帧）。

在检查过上面的计算结果后，您可以创建一个表空间，其中至少包含一个数据文件（482325 bytes）和一个BLOB文件（250帧，每个帧大小为11KB）来存放上列的表和索引。

在创建表空间时最好先规划表空间的大小。这样就可以避免以后增加数据文件或增大数据文件大小所带来的麻烦。

6.15 检验数据库的一致性

DBMaster包含一些命令，拥有DBA权限的用户可以使用这些命令来检查数据库的一致性。比如：检验数据库的一致性包括检验那些包含在索引中但在表中不存在的键值，或包含在外表中但在父表中不存在的键值。DBMaster提供了六个命令来检验不同级别的一致性，当数据库很大时，这些命令是很耗费时间的并且要占用锁，数据库管理员最好在必要的时候再使用这些命令。

检验索引

DBMaster允许拥有权限的用户来检验索引以及与索引相关的表，它可以检验索引架构（B-tree）的正确性，数据是否按顺序排列以及索引键是否和数据匹配等。

如果您无法判断索引是否存在问题，您可以使用此命令来检验索引的问题所在。如果DBMaster在索引中发现了不一致问题，那么您可以删除并重建此索引。

➲ **示例**

在**tb_staff**表中检验**idx_desc**索引的一致性：

```
dmSQL> CHECK INDEX tb_staff.idx_desc;
```

检验表

DBMaster允许有权限的用户检验关联一张表的所有记录、索引和BLOB数据以及外表和父表的联系。如果在表中存在不一致性，您可以载出表中所有记录，删除表，重建此表，然后再重新插入记录。

➲ **示例**

检验表**tb_staff**的一致性：

```
dmSQL> CHECK TABLE tb_staff;
```

检验系统表

DBMaster允许拥有DBA权限的用户检查系统表的一致性。如果系统表中存在错误，那么数据库就有可能被破坏。

➲ **示例**

您可以检验系统表的一致性：

```
dmSQL> CHECK CATALOG;
```

检验数据库

DBMaster允许拥有DBA权限的用户检验数据库的一致性，包括系统表和所有的表空间。

➲ **示例**

下例检验了整个数据库的一致性：

```
dmSQL> CHECK DB;
```

如果数据库在发生错误前及时做了备份，那么您可以使用最近的一次备份来恢复它。若想获得更多信息请参考第15章*数据库恢复、备份和还原*。

当数据库没有备份并且索引被损坏时，您可以删除或重建受损的索引。如果出现其它类型的故障，您必须立即备份数据库的所有数据和日志文件，然后试着关闭数据库并重新启动数据库，再次运行CHECK命令。当DBMaster执行自动恢复后，有的故障就可以被修复。如果数据库中的不一致性仍然存在，您可以和Syscom的技术支持取得联系以获得帮助。

检查用户文件

数据库以热启动方式启动时，DBMaster允许用户检验用户文件。如果用户开启了该功能，则DmServer将在数据库热启动时，检验所有用户文件以确保这些文件仍存在于在磁盘中，如果这些文件不存在，DBMaster将会返回一条警告信息以提醒拥有DBA或更高权限的用户避免操作被移除

的文件。该警告信息记录在日志文件*DMEVENT.LOG*中。用户可以通过设置关键字**DB_ChkFl**来开启该功能。

注意 *该功能不支持单用户模式。*

6.16 对象的更新统计

过时的对象统计值（表、索引、字段） 会引起DBMaster的优化程序执行一个无效率的SQL命令。在数据库管理员最后一次更新统计值后，如果用户再往数据库中插入大量数据，那么数据库会再一次对统计值进行更新。

统计值只有在数据库启动后才会被更新，更新统计时也会占用处理器资源，从而影响数据库的性能。选择一个避开表使用高峰期的时间点和时间间隔，以防止更新统计引起的数据库性能降低。

数据库运行期间，用户可通过系统存储过程**SetSystemOption**更改先前设定的统计值。

➲ **示例1**

对所有的对象执行更新统计：

```
dmSQL> UPDATE STATISTICS;
```

➲ **示例2**

对所有的模式对象执行强制更新统计：

```
dmSQL> UPDATE STATISTICS ALL;
```

如果数据库非常大，那么对此数据库的对象进行更新统计时将会占用大量的时间。当然您也可以使用另一种方法，即对那些被更改的对象进行更新统计，并且设置抽样率。

➲ **示例3**

对表执行更新统计：

```
dmSQL> UPDATE STATISTICS table1, table2, user1.table3;
```

➲ **示例4**

对表**tb_staff**的索引**idx_desc**执行更新统计：

```
dmSQL> UPDATE STATISTICS tb_staff (index idx_desc);
```

➲ **示例5**

对表空间**ts_reg**执行更新统计：

```
dmSQL> UPDATE TABLESPACE STATISTICS ts_reg;
```

➲ **示例6**

对表**tb_staff**的索引**idx_desc**和**idx_fill**执行更新统计：

```
dmSQL> UPDATE STATISTICS tb_staff (index idx_desc, idx_fill);
```

数据库运行期间，用户可通过系统存储过程**SetSystemOption**更改先前已指定的统计值。

➲ **示例7**

数据库运行时使用下例语法将更新统计样例设置为60：

```
dmSQL> CALL SETSYSTEMOPTION ('STSSP', '60');
```

有关更新统计和SQL命令的相关信息，请参考第18章*性能调优*。

7 大型对象管理

大型对象（LO）通常是指任意长度的数据类型，如文本、图像、声音或动画。针对大型对象，DBMaster提供了一个绝佳的处理方法。

DBMaster对表中的大型对象数目没有限制，并且对LO字段也无实际的大小限制。每一个LO字段的大小最大为2GB。DBMaster可以使用扩展的SQL语句来直接访问大型对象，避免用户去学习特殊的语法。由于所有大型对象的存取都可以使用SQL语句，所以用户很容易掌握如何使用大型对象。除此之外，用户还可以使用SQL语句或ODBC接口，将外部文件中的数据输入到大型对象中或将大型对象输出到外部文件中。

大型对象（LO）总是作为一个单位写入磁盘，然而，用户也可以从磁盘中读取所有或一个大型对象LO的一部分。所有的数据操作（SELECT、UPDATE、INSERT和DELETE）都可以执行在大型对象上。对于Boolean表达式来说，使用者可以测试大型对象的字段是否为NULL。

DBMaster也提供了一个MATCH函数来执行模式匹配操作。MATCH函数除了只能工作在LO字段上和不允许使用通配符外，其它功能和LIKE函数很类似。DBMaster不允许在大型对象上（LO）执行算术运算符或字符串表达式，也无法执行以下操作。

- 算术运算
- IN、ANY、EXIST或LIKE等谓词
- GROUP BY子句
- ORDER BY子句

DBMaster的大型对象主要有两种型态：二进制大型对象（BLOBs），存储在数据库文件中；文件对象（FOs），存储在主文件系统的外部文件中。

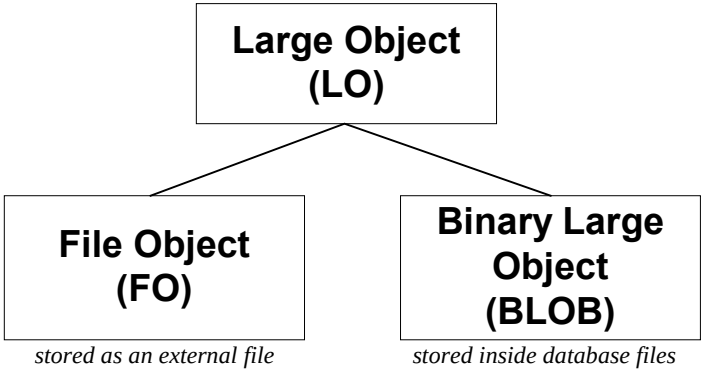


图 7-1 DBMaster所支持的大型对象

由于二进制大型对象（BLOBs）只存储在数据库系统内部，所以它只能通过DBMaster来访问，DBMaster提供了BLOB数据存取的完整性，例如事务控制（ transaction controls）、日志记录（logging）和事务恢复（recovery）。在更新记录时，一个BLOB只能被同一张表中的多笔记录共享，然而一个FO对象可以被数据库中的多张表共享。因此，当数据需要被其它非数据库应用程序共享时，您最好使用FO来提高工作效率。

7.1 BLOB数据管理

用户可以使用两种类型LONG VARCHAR和LONG VARBINARY来存储BLOB数据。LONG VARCHAR类型的数据是由文本数据组成的，如备忘录（memos）、大的文本对象（long text）、HTML源文件或程序源文件。LONG VARBINARY类型的数据可以是任意类型的二进制数据，如图像（images）、声音（sound）、电子数据表格（spreadsheets）、程序执行码（program modules）等。

DBMaster会依据BLOB数据的大小来决定它是储存在数据文件中还是储存在BLOB文件中。虽然数据库文件中的每一个数据页（数据库文件的划分单位）大小都是固定的，但是用户在创建数据库时，可以指定BLOB的数据帧（BLOB文件的划分单位）大小以获取较佳的执行性能和磁盘使用率。

要不要对BLOB作日志记录（logging），是由数据库管理员来决定的。一般而言，对于大的BLOB作日志记录会占据大量的空间，而且会降低系统的执行效率。为了预留足够的存储空间并且提高执行性能，数据库管理员可以关闭BLOB日志记录。不过以后在利用备份文件重建数据库时，DBMaster将无法保证BLOB内容的正确性。若BLOB日志记录的功能是开启的，数据库管理员就要增加日志文件的空间，以确保其能容纳整个BLOB数据，因为它会占据大量的磁盘空间并且降低执行效率。

定制BLOB空间

DBMaster会依照BLOB的长度大小来决定它是存储于数据库文件（*.SDB或*.DB）还是BLOB文件（*.SBB或*.BB）中。如果LONG VARCHAR或LONG VARBINARY类型的字段长度太小，并且记录的总长度没有超过记录的最大限制，那么DBMaster会将BLOB数据和数据库中的其它数据合成一笔记录。当DBMaster导出这笔数据时，会将它的BLOB数据一同取出，由于减少了一次磁盘的存取过程，数据库的执行性能也会随之提高。

如果整笔记录超出了记录的最大限制，DBMaster会将BLOB数据和记录分开存放。在这种情况下，获得BLOB数据需要两个磁盘操作：一个是导出数据记录，一个是导出BLOB，故称之为间接BLOB（indirect BLOB）。

依据BLOB的大小，间接BLOB可能被置于相同表空间的数据文件（*DB或*SDB）或BLOB文件（*BB或*SBB）中。如果BLOB数据的长度小于或等于16,240[2]（当数据页为16KB时）个字节，那么BLOB会被存放到数据页上；如果BLOB的长度超过16,240个字节，DBMaster会将这些BLOB数据存放到BLOB文件中。

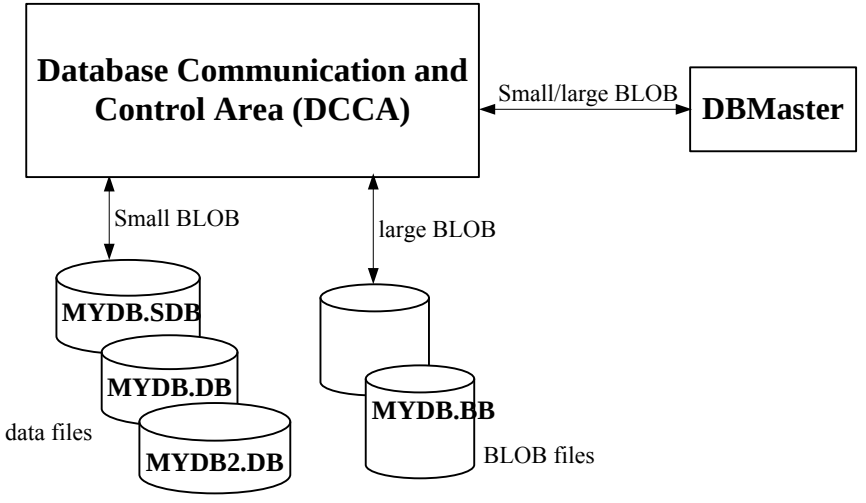


图7-2 DBMaster通过DCCA访问BLOB数据

数据库文件是以页（page）来划分，而BLOB文件是以帧（frame）来划分。DBMaster使用两种名词来区分它们的不同，主要的原因是：

[2] 当数据页大小为4K时，该值为 3952 bytes； 当数据页大小为8K时，该值为8048 bytes；当数据页大小为16K时，该值为16240 bytes ； 当数据页大小为32K时，该值为32624 bytes。

- 可以选择数据页的大小为4KB、8KB、16KB或32KB，在创建数据库时，可以通过关键字**DB_PgSiz**来设定；但帧的大小可由用户来指定。
- 一页可以包含多个记录，但一帧只能包含一个BLOB。

数据库管理员在建立数据库时可自己定制数据帧的大小，这样可以增加系统的执行性能，并提高磁盘的利用率。数据库管理员可以在建立数据库之前，在**dmconfig.ini**文件中通过设定**DB_BfrSz**关键字来设定帧（单位为KB）的大小，范围是8 - 256KB，默认值为32 KB。有关更多数据库创建时设置的环境设置参数的信息，请参考4.2章*创建数据库*。

➲ **示例1**

您可以将下列指令添加到**dmconfig.ini**配置文件中，以指定BLOB帧的大小：

```
DB_BfrSz = 16                          ; BLOB frame size = 16K  bytes
```

DB_BfrSz的有效范围为8KB – 256KB。

同一数据库中的所有BLOB文件，其帧的大小都是固定的。也就是说数据库一旦创建，BLOB帧的大小就无法被更改了。DBMaster将此值储存在数据库系统信息表中。当重启数据库时，DBMaster将从系统信息中获得此值的大小，所以**dmconfig.ini**内的**DB_BfrSz**只在数据库建立时起作用，此后也就没有作用了。

➲ **示例2**

查询SYSINFO系统表获得帧大小：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'FRAME_SIZE';


              INFO                              VALUE

============================== ==============================

FRAME_SIZE                     16384
```

```
1 rows selected
```

如何决定数据帧的大小，是要在磁盘的使用率上与系统的执行性能之间权衡的。如果整个BLOB数据需要频繁的检索，那么您可以调整帧的大小，尽量使其可以包含整个BLOB数据，这样会取得较好的执行效率。因为您只需执行一次磁盘访问，就可以得到整个数据。 但是许多BLOB的长度是不固定的，您若将数据帧的大小定义成能包含最大的BLOB数据，那么其余的BLOB就有可能会浪费磁盘空间，这是因为一个数据帧只属于一个BLOB，即使它只占用了帧的很小一部份空间，但剩余的空间不足以被其它BLOB所利用，所以将产生空间的浪费。相反，如果将帧的大小定义成只能包含最小BLOB数据的大小，在导出大的BLOB数据时，执行效率将会降低。

一个数据帧包含一个表头，用以记录自身的信息。如果帧的大小为8KB，扣掉表头的大小，才是真正可以存放数据的空间。所以这个空间略小于8,192个字节，大约有1.8KB用于存储BLOB的信息，所以BLOB第一帧的可用空间比一个完整帧的大小要小。举例来说， 如果BLOB的实际大小为 8,192字节，它将占据两个帧：BLOB的前6.2 KB大小将储存在第一个帧中，剩下的字节将储存在第二帧中。

一个完整的群组包含一个BE页和NBE[3]页块，BE页是一个PAGE_SIZE KB的数据页。而每一个PE页块则包含NPE[4] + 1个数据帧。 第一帧是一个大小为数据页大小[5]的PE页。剩下的NPE个帧都用来存放数据，它们的大小可通过**DB_BfrSz**来设定。

---

[3] NBE是指由BE页控制的PE块数。当数据页大小为4KB时，NBE为2004；当数据页大小为8KB时，NBE为2026；当数据页大小为16KB时，NBE为2716；当数据页大小为32KB时，NBE为2723。

[4] NPE 是指由PE页控制的页数。当数据页大小为4KB时，NPE为165；当数据页大小为8KB时，NPE为333；当数据页大小为16KB时，NPE为671；当数据页大小为32KB时，NPE为1347。

[5] 数据页的大小由关键字**DB_PgSiz**指定。

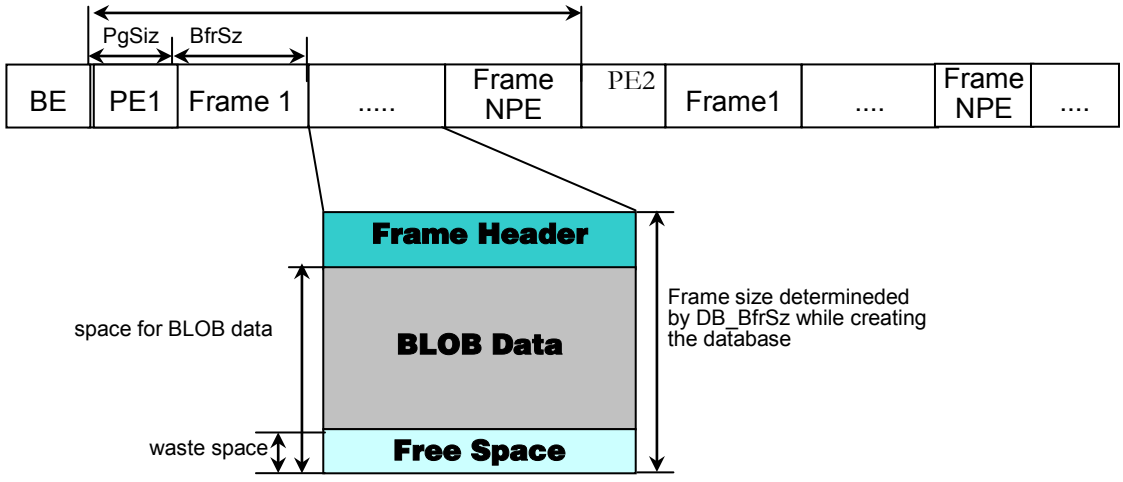


图 7-3 BLOB文件的结构

用户可以根据帧数、PE页、BE页和数据帧来计算BLOB文件的大小。

以下公式说明了如何估算BLOB的大小（单位为KB）

BE页数＝[总帧数 / 676685[6]]（[]代表取整运算）

PE页数＝[（总帧数 － BE页数）/ NPE + 1]

BLOB 文件大小=（BE页数 ＋ PE页数）x 数据页大小KB ＋（总帧数 － BE页数 — PE页数）x DB_BfrSz

举例来说，如果数据页的大小是8KB而 BLOB帧大小为32KB，那么带有3帧的BLOB文件的大小为：

BE页数 = [3 / 676685] =1

PE页数 = [(3 - 1) / 333 + 1] = 1

[6] 该值依赖于页大小，当数据页大小为4KB时，该值为332665；当数据页大小为8KB时，该值为676685；当数据页大小为16KB时，该值为1825153；当数据页大小为32KB时，该值为3670605。

BLOB文件大小 = (1 + 1) x 8 + (3 – 1 - 1) x 32 = 48 KB

DB_BbFil 用于指定属于系统表空间（SYSTABLESPACE）的系统BLOB文件名，但不能指定系统BLOB文件的大小。若没有指定DB_BbFil，系统BLOB文件的默认文件名是在数据库名后加上扩展名'.SBB'。

DB_UsrBb 用于指定默认表空间（DEFTABLESPACE）的默认用户BLOB文件名和大小。

有关表空间的详细信息以及如何向表空间中增加BLOB文件。 请参考第5.3章*向表空间中添加文件*。

产生BLOB

BLOB字段和其它字段相比，除了它的数据类型是LONG VARCHAR和LONG VARBINARY外，其余用法基本相同。

➲ **示例**

您可以通过下例SQL指令创建两个 BLOB 字段**note**和**photo**：

```
dmSQL> CREATE TABLE tb_staff (id INTEGER, note LONG VARCHAR, photo LONG
VARBINARY);
```

通过主变量插入BLOB数据**ab.txt**文件和图像文件**img001.gif**：

```
dmSQL> INSERT INTO tb_staff VALUES(2,?,?);

dmSQL/Val> &ab.txt, &img001.gif(2,4);

dmSQL/Val> END;
```

LONG VARBINARY字段的结果显示为十六进制格式。当用户浏览表时，将返回如下结果：

```
dmSQL> SELECT * FROM tb_staff;

  id          note              photo

=====      ===========     =================

    2      <script lan     ffd8ffe000104a464
```

DBMaster也可以将BLOB数据导出成用户指定的文件。有关如何插入/导出BLOB数据的信息，请参考*数据库管理工具用户手册*和 *ODBC程序员参考手册*。

更新BLOB

BLOB类型数据总是完整的写入磁盘。然而当更新一个BLOB类型字段时，DBMaster会将原来的BLOB类型数据删除，然后插入新的BLOB类型数据。

➲ **示例**

您可以使用UPDATE语句来更新BLOB字段的内容：

```
dmSQL> UPDATE tb_staff SET note = 'Hello !' WHERE id > 0;

dmSQL> SELECT * FROM tb_staff;

 id           note               photo

=====      ==========     ================

    1      Hello !         31323334353637

    2      Hello !         33343536
```

从用户的观点来看，上述表的每一个记录都拥有各自的BLOB数据。然而，为了节省磁盘空间，DBMaster仅为tb_staff表创建了一个BLOB数据，并被所有ID大于0的记录所共享。DBMaster提供了一个内部计数器，以记录有多少笔记录可共享此BLOB。在某笔记录更改共享的BLOB内容前，DBMaster会为此记录产生新的BLOB，并以1为基数递减该计数器的值。当更改一笔BLOB记录时，并不会影响其它记录。对DBMaster而言，这种松散的耦合（loose coupling）可以更有效的节省磁盘空间。目前一个BLOB只能由属于同一个表内的多笔记录所共享。当记录没有连接到共享的BLOB时，DBMaster会自动将它删除。

BLOB字段的谓词运算

BLOB对象只能用于CONTAIN和MATCH运算，您也可以通过布尔表达式（Boolean expressions）来测试BLOB字段的值是否为空（NULL）。

➲ **示例1**

您可以通过以下命令获取**tb_staff**表中**note**字段非空（**not null**） 的所有数据：

```
dmSQL> SELECT * FROM tb_staff WHERE note IS NOT NULL;
```

DBMaster为BLOB提供了匹配模式。CONTAIN和MATCH函数除了不支持通配符外，其它功能和LIKE函数十分相似。CONTAIN和MATCH的不同之处在于：前者只需匹配部分的英文字符，而后者需要完全匹配。例如：'This is a character. ' CONTAIN 'char'，而'This is a character. ' NOT MATCH 'char'，但 MATCH 'character'。

➲ **示例2**

您可以通过以下指令查找**note**字段匹配 'Database Administrator'的元素：

```
dmSQL> SELECT * FROM tb_staff WHERE note MATCH 'Database Administrator';
```

7.2 文件对象的管理

每一个文件对象（FO）都可以参考外部文件。使用文件对象的好处在于：其它应用程序可以不必通过数据库来存取数据，它们只需直接存取操作系统中的文件目录就可以。当前大多数多媒体工具只能靠存取文件来处理数据，如果多媒体数据直接存于BLOB字段（即类别是LONG VARCHAR或LONG VARBINARY 的字段），应用程序必需先通过DBMaster导出这些数据，然后再将这些数据存放到另一张外部暂存盘中，最后才能交由多媒体工具来处理。然而如果BLOB数据作为FO来存储，用户可以通过DBMaster先获得文件对象的文件名，然后再将此文件名交由适当的多媒体工具来处理。

文件对象（FO）存在两种不同的类型：*系统文件对象（system FO）和用户文件对象（user FO）*。当用户在客户端插入一笔数据时，系统文件对象也就随之创建，DBMaster会将此文件对象存储到外部文件中，同时在配置文件中设定配置参数**DB_FoDir**。系统文件对象（System FOs）可以通过 DBMaster来创建，文件扩展名为 .FOB。用户文件对象所链接到的文件，原本就已存在于某一目录下。用户文件对象可以是任一驱动器上的文件，DBMaster可通过服务器的操作系统来对它们进行存取。

系统文件对象和用户文件对象的不同之处在于：DBMaster会自动产生一个系统文件对象。也就是说，当没有字段参照它时，系统文件对象将被删除。因此，对于系统文件对象，用户可以委托DBMaster来存储管理。使用系统文件对象的另一个好处是数据可以在服务器端复制，于是用户可以从服务器端来管理数据。DBMaster的备份和恢复功能也同样支持系统文件对象。

接下来我们了解一下文件对象的另一种类型：用户文件对象。当没有字段参照它时，用户文件对象不会被删除。用户文件对象的主要优势在于：DBMaster可以直接将一个字段链接到外部文件，它无需复制数据。例如CD-ROM中的文件，它节省了磁盘空间，并且很容易共享包含多笔记录的文件。然而如果文件在DBMaster之外被删除，所有链接到此字段

的文件将会变的无效。当一个文件作为用户文件链接时，必须打开它的读权限。

用户可从数据库中访问用户文件对象，它们可以在服务器端分散存放。除了指定用户文件（USER FO）目录，您还需要在配置文件**dmconfig.ini**中将关键字**DB_UsrFO**的值设置为1。默认状态下，用户文件对象为不可用。

用户可以通过内建函数（filename() 和 filelen()）来获得FO的文件名和文件大小。

指定系统文件对象的路径

DBMaster在存储系统文件对象时，可产生一系列的文件对象子目录，这些子目录都位于文件对象的目录中，文件的存储路径可通过配置文件**dmconfig.ini**中的关键字**DB_FoDir**来设定。当文件对象子目录中的文件数超过所设定的阈值时，一个新的子目录也就随之创建。此阈值可通过配置文件**dmconfig.ini**中的关键字**DB_FoSub**来设置，范围为100 - 10,000。

文件对象名可以是ZZ*xxxxxx*.*ext*的形式，其中的*xxxxxx*是一个36进制的六位数字，*ext*是文件对象的扩展名，文件对象的扩展名可以通过SET EXTNAME指令来设定。有关扩展名的详细信息，请参考*系统文件对象扩展名*章节。

子目录的命名和文件对象的命名规则类似，它的形式是SUB*xxxxxx*，其中*xxxxxx*也是一个36进制的六位数字，它是从插入子目录中的第一个文件对象的序列数依次来编制的。

尽管文件对象目录可以被多个数据库所共享以简化FO管理，但我们并不建议您这样做。因为这样会在备份数据库时变的十分不便。在数据库启动前或运行期间，文件对象路径可以通过更改配置参数来进行修改。

离线设置文件对象（FO）的路径

用户可以通过配置文件**dmconfig.ini**中的**DB_FoDir**参数来指定系统FO的存放位置。参数**DB_FoDir**必须是一个完整路径，而且DBMaster必须拥有此路径的写权限。

➲ **示例1**

若要在*/disk1/usr/fo*路径中创建系统FO，您必须在配置文件中添加以下命令行：

```
DB_FoDir = /disk1/usr/fo
```

➲ **示例2**

设置DB_UsrFo=1赋予用户对象权限：

```
DB_UsrFo = 1          ; enable USER file objects
```

在线设置文件对象（FO）的路径

在数据库运行期间，DBMaster为用户更改系统文件目录提供了一个应用程序。此操作会对以下三个参数的值作更改：

- **Run-time FO directory —**当执行一个更改操作后，所有更改后的系统文件对象都将存储于一个新的FO目录中。
- **DB_FoDir —**当数据库再次启动时，它将使用新的FO路径。
- **$DB_FODIR alias. —**默认的FO别名，与配置文件**dmconfig.ini**中的**DB_FoDir**参数一致。

➲ **示例1**

您可以通过以下指令将文件对象（FO）的目录更改成一个新目录，如：***/home/DBMaster/mydb/fo***

```
dmSQL> CALL SETSYSTEMOPTION('fodir', '/home/DBMaster/mydb/fo');
```

除了可以对文件对象目录作更改外，您还可以对系统进行查询以返回FO的当前目录。

➲ **示例2**

您可以通过以下指令来返回文件对象（FO）的当前目录：

```
dmSQL> CALL GETSYSTEMOPTION('fodir', ?);

OPTION_VALUE: /home/DBMaster/mydb/fo
```

产生文件对象

使用DBMaster产生文件对象需要通过以下几个步骤。首先必须在表上创建一个FILE类型的字段；接下来在FO字段上插入系统或用户文件对象。为了创建FO字段，您可以在创建表的同时将字段的类型设置为 FILE。

➲ **示例1**

以下指令用来创建表**tb_person**，其中字段**photo**的类型为FILE：

```
dmSQL> CREATE TABLE tb_person (name CHAR(10), photo FILE);
```

➲ **示例2**

如果用户在服务器端插入文件对象（FO），那么DBMaster会链接此FO字段到一个存在的文件，并且产生一个用户FO。如果用户在客户端插入文件对象（FO），DBMaster会将此文件对象从客户端复制到服务器端的FO目录中，并且创建一个系统FO。我们也可以在表**person**中插入FO数据：

```
dmSQL> INSERT INTO tb_person VALUES ('cathy','/disk1/image/cathy.bmp')
2>;                                          // stored as a USER FO

dmSQL> INSERT INTO tb_person VALUES ('jeff',?);

dmSQL/Val> &jeff.gif;                              // stored as a SYSTEM FO

dmSQL/Val> END;
```

➲ **示例3**

您可以通过三种方式导出FO字段：内容、文件名和文件大小。下例就是一个导出名为**cathy.bmp**文件的例子：

```
dmSQL> SELECT photo, FILENAME(photo), FILELEN(photo) FROM tb_person ;
```

```
  photo          filename(photo)           filelen(photo)
=========    ========================    =============
012034451    /disk1/image/cathy.bmp        21100
349045821    /disk1/usr/fo/ZZ000000.hmp    12034
```

有关更多如何处理文件对象（FO）字段的信息，请参考*数据库管理工具用户手册*和*ODBC程序员参考手册*。

系统文件对象的扩展名

用户可以使用SET EXTNAME命令来设置系统文件对象的扩展名。

➲ **示例1**

您可以通过以下方式设置系统对象的扩展名*<extension_name>*：

```
SET EXTNAME TO <extension_name>
```

*<extension_name>*包括以下两种字符串：

- 不超过7个字符的字符串，如 'bmp'、'avi'、'jpg'等。
- 不区分大小写的SOURCE字符串，其扩展名和客户端源文件的扩展名相同。

➲ **示例2**

使用SET EXTNAME命令设置文件对象扩展名的范例如下：

```
dmSQL> CREATE TABLE tb_example (c1 INT, f1 FILE);
dmSQL> INSERT INTO tb_example (c1, f1) VALUES (?, ?);


dmSQL/Val> SET EXTNAME TO FOB;
dmSQL/Val> 1, &readme.txt;             //extension name : '.FOB'
1 rows inserted
dmSQL/Val> SET EXTNAME TO doc;
dmSQL/Val> 2, &readme.txt;             //extension name : '.doc'
```

```
1 rows inserted
dmSQL/Val> SET EXTNAME TO SOURCE;
dmSQL/Val> 3, &readme.txt;            //extension name : '.txt'
1 rows inserted
dmSQL/Val> END;
dmSQL> SELECT FILENAME(f1) FROM tb_example;
        c1                      FILENAME(f1)
======================  ================================
                      1  /usr1/fo/ZZ000001.FOB
                      2  /usr1/fo/ZZ000002.doc
                      3  /usr1/fo/ZZ000003.txt
3 rows selected
```

更新文件对象

您可以使用SQL UPDATE命令来更新FO字段的内容。在对FO字段进行更改后，DBMaster会使用新的文件来替换旧的FO文件。

类似新增文件对象的方法，用户可以将FILE字段更新为新的系统文件对象（SYSTEM FO）或用户文件对象（USER FO）。

➲ **示例**

以下SQL指令可以将**photo**字段链接到一个外部文件 **/disk2/image/common.bmp** 中：

```
dmSQL> UPDATE tb_person SET photo = '/disk2/image/common.bmp' WHERE name
= 'cathy';
```

或者，用户也可以在客户端从文件中输入新的数据。有关更多更新文件对象的信息请参考*数据库管理工具用户手册*和*ODBC程序员参考手册*。

如果UPDATE的运算结果包含多条记录，那么只有一个文件被创建，此文件可被多条记录所共享以节省磁盘空间，DBMaster会提供一个内部计

数器来记录参照该文件的记录数。此外，如果用户通过外部应用程序更改了文件的内容，那么所有参照该文件的记录都将被更改。

在执行UPDATE或DELETE操作后，当没有记录链接系统FO时，DBMaster在提交事务后会自动将文件删除。然而DBMaster即使在没有记录参照它的前提下，也无法删除任何一个用户文件对象（USER FO），因为DBMaster没有产生文件。

重命名文件对象

有时用户会因磁盘空间不足或需要对磁盘目录进行重组，而对文件对象的存放路径和名称进行更改。DBMaster允许用户使用MOVE FILE OBJECT 命令来更改文件对象的名称和存放路径。但是DBMaster并不会更改外部文件的实际目录和名称，所以在使用MOVE FILE OBJECT命令之前，您必须使用操作系统提供的移动或复制命令，将实际的文件移到新的目录下。在移动文件对象之前，DBMaster会查看新文件是否存在。

➲ **示例1**

如果不知道文件的实际存放位置，您可以使用filename()内建函数来获得文件名。

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ZZ000000.FOB' TO
'/disk3/pub/photo1.bmp';
```

DBMaster允许用户将文件对象（FO）从一个目录移动到另一个目录中。请注意：DBMaster的源目录文件名中可以存在 * 字符，但在目的目录中却不允许出现 * 字符。DBMaster不支持递归移动的文件，如果要移动所有文件不包括子目录，请在指明的目录后加上'/' 或 '/*'字符。

➲ **示例2**

如果目录***/disk1/usr/fo***中存在四个文件：**ABC1.FOB**、**ABC2.FOB**、**ABC3.FOB**、**ABC4.FOB**。您可以使用以下指令将这四个文件从***/disk1/usr/fo***目录移动到***/disk3/pub***目录中：

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ ' TO '/disk3/pub/ ';

dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/* ' TO '/disk3/pub/ ';
```

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*.FOB ' TO '/disk3/pub/ ';

dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A* ' TO '/disk3/pub/ ';
```

以下指令可将**ABC1.FOB**从***/disk1/usr/fo***目录移动到***/disk3/pub***目录中：

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*1.FOB ' TO '/disk3/pub/ ';

dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A*1.FOB ' TO '/disk3/pub/ ';
```

载出系统文件对象

DBMaster允许数据库管理员决定是否使用UNLOAD FILEOBJ ON/OFF命令来载出系统文件对象。默认设置为**ON**。如果未设置该选项，所有服务器上的文件对象将被载出到当前工作目录。如果磁盘空间不足，可使用SET UNLOAD FILEOBJ OFF命令手动复制文件对象到工作目录。

新数据库载入上述文件时，所有系统文件对象将被重新排序并重命名。迁移数据库时，要想在应用程序中引用这些载出文件的对象名且不想重新排序，可在UNLOAD DB之前使用SET UNLOAD FILEOBJ NAME命令选项设置，dmSQL会通过完整的路径和文件名将脚本视作客户端文件对象载出。载入数据库之前，用户需在dmconfig.ini配置文件中确认关键字**DB_UsrFo=1**，否则将不会载入，之后所有文件对象会以相同的路径和文件名保存在服务器上。

获取文件对象的长度

文件对象的长度可以通过内建函数FILELENEX和FILELEN而得到。更多关于函数的信息请参考*SQL命令与函数参考手册*。

文件对象的谓词运算

类似于BLOB字段，用户可以测试FO字段的值是否为空（NULL），也可以使用CONTAIN和MATCH函数来执行模式搜索（pattern search）。而且文件对象（FO）可以在算术表达式中使用FILELEN()内建函数，在布

尔表达式中使用FILEEXIST()内建函数，在字符串表达式中使用FILENAME()内建函数。

当文件被移动或重命名时，您可以使用FILEEXIST()内建函数来测试文件是否存在。0代表被参考的文件对象不存在；1代表被参考的文件对象仍然存在；NULL代表记录的值为空（NULL）。

➲ **示例1**

从字段**photo**中选择扩展名为 .gif的记录：

```
dmSQL> SELECT * FROM tb_person WHERE FILENAME(photo) LIKE '%.gif';
```

➲ **示例2**

从**photo**字段中导出大于100KB的记录：

```
dmSQL> SELECT * FROM tb_person WHERE FILELEN(photo) > 102400;
```

➲ **示例3**

导出现有文件的所有记录：

```
dmSQL> SELECT * FROM tb_person WHERE FILEEXIST(photo)=1;
```

文件对象的通用命名标准

在Microsoft Windows环境下，文件对象的路径和目录名可使用UNC命名标准。当DBMaster的服务器运行在Microsoft Windows平台上时，使用此UNC标准将会给指定文件对象路径和目录名带来方便。在使用UNC标准访问数据库以外的机器目录时，需要对用户授权。

➲ **示例1**

您可以在**dmconfig.ini**配置文件中输入以下参数来检索***\\NTMACHINE\E\FO***目录中的系统文件对象：

```
DB_FoDir = \\NTMACHINE\E\FO
```

➲ **示例2**

以下指令将显示使用UNC命名的文件对象的执行情况：

```
dmSQL> CREATE TABLE tb_example (c1 INT, c2 FILE);
dmSQL> INSERT INTO tb_example VALUES (?, ?);
dmSQL/Val> 1, '\\NTMACHINE\D\DB\memo1.txt';
1 rows inserted
dmSQL/Val> 2, &c:\temp\memo2.txt;
1 rows inserted
dmSQL/Val> END;

dmSQL> SELECT c1, FILENAME(c2) FROM tb_example;
   c1                    FILENAME(c2)
========= ================================================
        1 _\\NTMACHINE\D\DB\memo1.txt
        2 _\\NTMACHINE\E\FO\ZZ000001.txt

2 rows selected
```

文件对象路径的默认别名

DBMaster为文件对象路径提供了两个别名：**$DB_DbDir**和**$DB_FoDir**。文件对象路径的别名代表了文件对象的实际路径，用户可以通过路径的别名来对文件对象进行插入/更新/删除操作，您也可以很容易的将文件对象移到另一个目录下。

这两个别名（**$DB_DbDir**和**$DB_FoDir**）可以通过配置文件**dmconfig.ini**中的参数**DB_DbDir**和**DB_FoDir**来分别设置。

➲ **示例1**

您可以通过**dmconfig.ini**配置文件中的**DB_FoDir**参数来设置文件对象路径的别名：

```
…
DB_FoDir = "/usr1/tmp/employeedata/FO"
```

➲ **示例2**

您可以在FILE类型的字段**photo**上插入文件对象路径的别名：

```
dmSQL> CREAT TABLE tb_example (c1 INT ,photo FILE);

dmSQL> INSERT INTO tb_example VALUES(2,'$DB_FoDir/photo471.jpg');
```

上例文件**photo471.jpg**中也可以插入文件对象的完整路径 '***/usr1/tmp/employeedata/FO/*photo471.jpg**'。

文件对象和应用程序

只有DBMaster可以支持FILE类型的数据，并且无法通过ODBC来定义。像Inprise/Borland Delphi或Microsoft Visual Basic之类的开发工具均无法识别FILE类型的数据。您可以通过配置参数**DB_FoTyp**来指定开发工具是否能识别FILE类型的数据。为了使这些工具可以访问FILE类型的数据，您可以令参数**DB_FoTyp = 1**将FILE类型的数据设置为LONG VARBINARY。如果令参数**DB_FoTyp = 0**，表示不映射FILE类型的数据，那么开发工具将无法识别此类数据。

➲ **示例**

您可以令参数DB_FoTyp=1，将FILE类型的数据映射为LONG VARBINARY：

```
DB_FoTyp = 1
```

7.3 大型对象的日志

如果日志记录中包括BLOB（类型为LONG VARCHAR或LONG VARBINARY的字段）数据，那么此日志会占用大量的磁盘空间，并且会降低系统的执行效率。目前DBMaster可以让数据库管理员决定是否对指定表空间的BLOB数据进行日志记录，但DBMaster不支持文件对象（FILE）的日志记录。

DBMaster的默认状态是不对BLOB数据进行日志记录。在数据库的启动和关闭期间，DBMaster确保了BLOB数据的一致性。甚至在系统发生崩溃时，BLOB数据也会在恢复后保持一致。

然而当从增量备份处做数据库恢复时，DBMaster却无法保证BLOB数据的一致性。为了确保日志文件中BLOB数据的一致性，您必须执行以下两个步骤。首先在**dmconfig.ini**中设置参数 **DB_BMode**以记录BLOB数据的事务处理。其次，数据库管理员必须在 BACKUP BLOB ON模式下创建表空间，来备份此表空间中的BLOB数据。

BLOB的日志文件

为了能够记录BLOB数据，您必须遵循以下两个条件：

- 在**dmconfig.ini**中设置参数**DB_BMode** = 2 （BACKUP DATA AND BLOB模式）。
- BLOB文件被添加到在BACKUP BLOB ON选项下创建的表空间中。

设置DB_BMODE参数值

此参数**DB_BMode**可用于指定数据库的备份模式。将参数值设为0代表NON-BACKUP模式；1代表BACKUP-DATA模式；2代表BACKUP-DATA-AND-BLOB 模式。

- NON-BACKUP （0）模式 — 系统表空间和用户定义表空间中的数据和 BLOB均不支持增量备份。

- BACKUP-DATA （1）模式 — 仅对系统表空间、用户定义表空间中的数据执行增量备份，但不包括用户定义表空间中的BLOB。
- BACKUP-DATA （2）模式 — 对系统表空间，用户定义表空间中的数据以及在BACKUP BLOB ON模式下创建的用户定义表空间中的BLOB数据执行增量备份，但不支持BACKUP BLOB OFF模式下的用户定义表空间中的BLOB数据。

开启数据库的备份模式以记录BLOB数据，请在**dmconfig.ini**中添加以下命令行：

```
DB_BMode = 2    ;log all data including BLOB
```

有关数据库备份模式的详细内容，请参考第15章*数据库恢复、备份和还原。*

设置CREATE TABLESPACE BACKUP选项

在创建表空间时，您可以设置表空间的备份模式。以下就是创建表空间的SQL命令CREATE TABLESPACE：

```
CREATE [AUTOEXTEND] TABLESPACE tablespace_name [backup_mode]
DATAFILE [tsfile , tsfile, ...];
```

其中：

```
backup_mode ::= BACKUP BLOB OFF | BACKUP BLOB ON
tsfile ::= file_name TYPE = DATA | file_name TYPE = BLOB
```

用户可以在BACKUP BLOB ON模式下，将一些重要的BLOB数据存放到表空间中。为了提高系统的执行效率，您也可以在BACKUP BLOB OFF模式下，将无需备份的BLOB数据存放到表空间中。表空间的创建者可以对此进行选择以决定如何存放BLOB数据。

在创建表空间之前，必须在配置文件**dmconfig.ini**中指明数据和BLOB文件。您可以通过JConfiguration Tool中的**用户文件**页签来熟悉它。有关如何创建数据和BLOB文件的详细说明，请查阅*配置管理工具用户手册*。

➲ **示例1**

下例说明了如何在BACKUP BLOB OFF模式下创建表空间**ts_reg**，在BACKUP BLOB ON模式下创建表空间**ts_aut**：

```
dmSQL> CREATE TABLESPACE ts_reg BACKUP BLOB OFF
    2> DATAFILE f1 TYPE = DATA, f2 TYPE = BLOB;
dmSQL> CREATE TABLESPACE ts_aut BACKUP BLOB ON
    2> DATAFILE f3 TYPE = DATA, f4 TYPE = BLOB;
```

➲ **示例2**

您可以通过**SYSTABLESPACE**表中的**BK_MODE**参数来获知每一个表空间的备份模式。该参数值为1代表BACKUP BLOB为关闭状态（OFF），参数值为2代表BACKUP BLOB为开启状态（ON）。查询表空间的备份模式可以获得以下结果：

```
dmSQL> SELECT TS_NAME, BK_MODE FROM SYSTEM.SYSTABLESPACE;
     TS_NAME         BK_MODE
==============  ===============
SYSTABLESPACE                2
DEFTABLESPACE                2
ts_rge                       1
ts_aut                       2
4 rows selected
```

数据库的备份模式和表空间的备份模式是如何交互影响BLOB日志记录的，可简单地用下表来作说明。其中‘Yes’表示有备份日志记录，‘No’表示没有备份日志记录。

数据库备份模式	表空间备份模式	用户自定义表空间（DATA）	用户自定义表空间（BLOB）	系统表空间（DATA 和 BLOB）
NON BACKUP（**DB_BMode** = 0）	---	No	No	No
BACKUP DATA（**DB_BMode** = 1）	---	Yes	No	Yes
BACKUP DATA AND BLOB（**DB_BMode** = 2）	BACKUP BLOB OFF	Yes	No	Yes
	BACKUP BLOB ON	Yes	Yes	Yes

在设置数据库的备份模式之前，您必须确保日志文件足够大以至于能存储所有的BLOB数据。否则系统将会返回日志已满的提示错误信息。有关如何调整日置文件大小的详细信息，请参考第5章*扩大日志文件空间*。

有关数据文件、BLOB文件和表空间的概念，请参考第5章*储存架构*。

有关CREATE TABLESPACE命令的信息，请参考*SQL命令与函数参考手册*。

有关SYSTABLESPACE表的信息，请参考*系统目录参考*章节。

文件对象的日志记录

DBMaster无法对文件对象作日志记录（journal logging）。在执行数据库备份时，最好手动的将此数据库中的所有文件对象都复制到备份目录中。有关备份文件对象的信息，请参考第15.6章*备份服务器*。要想知道数据库中包含哪些文件，可以查询系统表SYSFILEOBJ。

➲ **示例**

您可以通过查询系统表SYSFILEOBJ来检索文件对象的文件名：

```
dmSQL> SELECT FILE_NAME FROM SYSFILEOBJ;
```

您可以将文件对象复制到备份目录中。在执行数据库备份时，最好将此数据库中的所有文件都复制到备份目录中。如果文件的路径和名称已经更改，那么您可以使用MOVE FILE OBJECT命令来更新系统表SYSFILEOBJ中的文件。

7.4 大型对象和SELECT INTO命令

使用SELECT INTO命令可以选择您所需的数据并将它们插入到指定的表中。您可以使用此命令将文件对象和BLOB数据从一张表中移动到另一张表。此SELECT INTO命令可用于分布式数据库（DDB）环境下。

在本地到本地使用SELECT INTO命令时，DBMaster需要复制BLOB数据，并为系统文件对象或用户文件对象添加共享计数器。

在分布式环境下（DDB），DBMaster可以将BLOB数据从一个站点复制到另一站点。但是执行此操作对文件对象有许多要求。为了确保分布式数据库环境下文件对象的处理进程，DBMaster提供了一个分布式文件对象复制模式，此模式可通过SET DFO DUPMODE命令来设置。

设置SET DFO DUPMODE

DFO DUPMODE可以通知数据库是否将文件对象复制到目的数据库中。DFO DUPMODE有两种模式：NULL模式和COPY模式。

➲ **示例**

设置DFO DUPMODE（NULL模式和COPY模式）的语法如下：

```
dmSQL> SET DFO DUPMODE NULL;

dmSQL> SET DFO DUPMODE COPY;
```

设置DFO DUPMODE NULL模式

在分布式数据库（DDB）环境下，有两种情形需要考虑：

- 如果源数据库和目的数据库相同，包括本地到本地、远程到远程数据库。由于它们可共享这些文件对象，所以DBMaster只需添加文件对象的共享计数器即可。
- 如果源数据库和目的数据库不同，目的表中的FILE字段被设置为NULL。那么源数据库中的文件对象将无法发送出去。

设置DFO DUPMODE COPY模式

您需要注意以下三种状态的文件对象：

- 用户文件对象：DBMaster只传送源数据库中的文件名到目的数据库中，所以用户应该将源文件复制到目的数据库可以访问的目录中。如果目的数据库中的新目录不同于源数据库中的目录，那么用户需要使用UPDATE或MOVE FILE OBJECT命令来更改目的数据库中的文件名。
- 不同数据库中的系统文件对象：DBMaster将在目的数据库中创建一个新的系统文件对象，并将源数据库中的内容复制到这个新建的对象中。
- 同一数据库中的系统文件对象：局部-局部、远端-远端。DBMaster只需添加共享计数器即可。

限制

DFO DUPMODE模式不会影响使用在BLOB （LONG VARCHAR和LONG VARBINARY 类型）字段上的SELECT INTO命令。也就是说，您无需考虑DFO DUPMODE 模式就可以使用SELECT INTO命令来复制BLOB数据。

在分布式数据库（DDB）环境下，如果SELECT INTO命令用于用户文件对象，并且DFO DUPMODE设置为COPY模式，那么用户应该清楚地知道目的数据库中要访问的文件目录。此链接文件对象应该在目的数据库中存在相同的路径。如果此路径不存在，那么用户应该使用操作系统命令将文件从源数据库复制到目的数据库中，如果源数据库和目的数据库中的路径不同，您可以使用UPDATE或MOVE FILE OBJECT命令来更改这些字段。

用户若没有正确的执行以上操作，在查询文件对象时会返回一条文件不存在的错误信息，这是因为文件对象的完整路径不正确或此文件根本就不存在。

当从远程数据库选择一个系统文件对象到本地数据库时，DBMaster会记录SELECT INTO的信息。因此您仍需对文件进行复制，这样就会极大的

浪费磁盘空间。另外，当从远程数据库选择一个系统文件对象到目的数据库时，您也需要创建一份复制文件。

使用SET EXTNAME选项不会影响SELECT INTO的执行结果。源数据库和目的数据库中，文件对象的扩展名是相同的。例如源数据库的文件名是'ZZ000001.BMP'，那么目的数据库中的文件名为'ZZXXXXXX.BMP'。

➲ **示例**

DBMaster会将CHAR、VARCHAR、BINARY类型的数据作为文件名，所以用户必须确保**db2**数据库能够从视图**db1**中访问***/etc/hosts***。下例说明了如何将CHAR类型的字段插入到FILE字段中，其中**db2**数据库中**t2**表的**c2**字段是FILE类型：

```
dmSQL> SELECT c1, '/etc/hosts' FROM db1:t1 INTO db2:t2(c1, c2);
```

考虑目的数据库中的字段类型是FILE，下表针对不同的源数据库类型，总结了SELECT INTO命令的执行结果：

源数据库类型	环境	设置DFO DUPMODE	执行结果
字符串表达式 CHAR VARCHAR BINARY	本地数据库或分布式数据库	……	源数据库：传送文件名。 目的数据库：插入一个新用户文件对象。
FILE	源数据库和目的数据库相同	……	添加文件对象的共享计数器。
	源数据库和目的数据库不同	NULL	目的数据库：插入空（ NULL） 值。
		COPY	源字段是用户文件对象。 源数据库：传送文件名。 目的数据库：插入新的用户文件对象。 源字段是系统文件对象。 源数据库：传送文件对象的内容。 目的数据库：插入新的系统文件对象。
LONG VARCHAR LONG VARBINARY 其它	……	……	不支持

8 安全性管理

本章将说明如何建立DBMaster数据库的安全性策略，其中包括数据库安全性、用户管理权限和表权限。

8.1 安全性策略

DBMaster提供了两种安全性管理：

- **数据库权限 —** 决定哪些用户可以登录DBMaster数据库以及可以执行的操作。
- **对象权限 —** 控制DBMaster对象的存取权限。DBMaster对象包括表、字段、视图、定义域和同义字。

8.2 数据库授权

数据库授权用于决定哪些用户可以访问数据库。DBMaster通过用户名和密码来控制数据库的访问。在下表中我们可以看到DBMaster有五种用户权限。

SYSADM是DBMaster的最高权限，一个数据库只能拥有一个SYSADM。拥有 SYSADM权限的用户可以将SYSDBA、DBA、RESOURCE、CONNECT权限授予其他用户，还可以向其他用户设置ACL（访问控制列表），并且拥有数据库中DBA和SYSDBA授权层次的所有权限。

拥有SYSDBA权限的用户不仅可以授予或取消其他用户的CONNECT、RESOURCE和DBA权限，还可以更改非SYSADM或SYSDBA用户的密码，并可以向低权限用户设置ACL（访问控制列表）。拥有SYSDBA权限的用户具备DBA权限用户的所有权限，只有SYSADM才可以授予或取消用户的SYSDBA权限。如果SYSADM取消了用户的SYSDBA权限，此时用户仍拥有DBA权限；如果SYSADM取消了用户的DBA权限，则该用户既无SYSDBA权限，也无DBA权限。

拥有DBA权限的用户不仅具备数据库中全部对象的所有权限，而且可以授予、更改或取消数据库中非SYSADM、SYSDBA和DBA的其他用户对象权限。他们也能创建新的资源，如创建表空间和文件；或执行数据库管理员的操作，如启动/终止/备份数据库。

拥有RESOURCE权限的用户除了可以创建新表或视图外，还能够将他们自己的表权限授予其他用户。拥有CONNECT权限的用户只能访问授予此权限的对象，但是不能创建新表或视图。虽然如此，他们仍可以选取系统表中的信息。

下图显示了五种权限的层次关系：

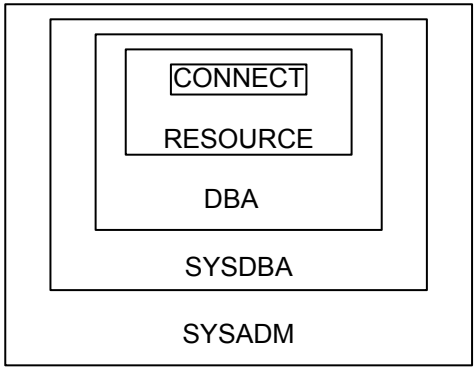


图8-1 DBMaster数据库的授权层次

级别	权限
SYSADM	能够授予/取消所有用户的安全性权限（除了SYSADM 权限的用户）。 能够更改所有用户密码。 具有DBA授权层次的所有权限。
SYSDBA	能够授予/取消SYSADM和SYSDBA以外用户的安全权限（CONNECT/RESOURCE/DBA）。 能够更改SYSADM和SYSDBA以外用户的密码。 具有DBA授权层次的所有权限。
DBA	拥有表的所有权限（除了系统表以外）。 能够授予/更改/取消所有用户和群组的对象权限。 能够从群组中添加/取消用户。 拥有数据库管理员权限，能够启动或关闭数据库，创建/删除/更改表空间以及备份数据库。 拥有CONNECT和RESOURCE授权层次的所有权限。

级别	权限
RESOURCE	能够创建/删除表、视图、定义域和同义字。 只能够删除用户自己创建的表、视图、定义域和同义字。 能够授予/取消其他用户在自己表/视图上的权限。 具有所有被授予的表权限。 具有CONNECT授权层次的所有权限。
CONNECT	能够登录数据库。 能够选取系统表（SYSTEM table）。 具有所有被授与的表权限。 用户在授予其他权限之前，必须先授予此权限。

表8-1 DBMaster数据库的授权层次

用户的管理

DBMaster提供了许多SQL命令来管理用户。这些命令可以添加/删除用户、设置或更改用户密码并且可以用来授予用户权限。

添加用户

在新用户登录到DBMaster之前，SYSADM必须使用GRANT命令为用户指定用户名和密码。

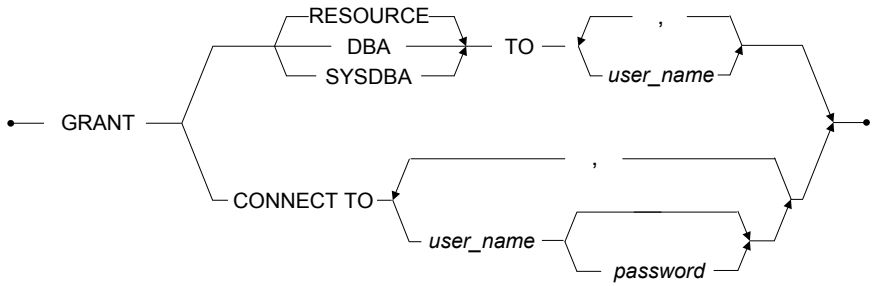


图8-2 GRANT命令的语法

GRANT命令是用来授予用户权限的。只有SYSADM能够为其他用户授予权限，但SYSADM无法将自身的权限授予其他用户。所以每一个数据库中只能有一个用户可以拥有SYSADM用户名和SYSADM权限。SYSADM是创建数据库的默认用户。只有拥有SYSADM权限的用户才可以更改密码。

SYSADM可以为其他用户授予CONNECT、RESOURCE、DBA、SYSDBA以及ACL权限。如果使用 GRANT命令为其他用户授予RESOURCE、DBA或SYSDBA权限，那么新的权限只有在下次连接数据库时才会生效。

SYSADM或拥有SYSDBA权限的用户能够为拥有CONNECT权限的用户授予密码。如果SYSADM没有指定密码，就表示用户在登录数据库时无须输入密码。密码可以是任何合法的 SQL字符，最大长度为16个字节。

➲ **示例1**

为用户Jeff授予CONNECT权限和密码jeff123：

```
dmSQL> GRANT CONNECT TO Jeff jeff123;
```

➲ **示例2**

将用户Jeff的权限升至RESOURCE：

```
dmSQL> GRANT RESOURCE TO Jeff;
```

➲ **示例3**

将用户Jeff的权限升至DBA：

```
dmSQL> GRANT DBA TO Jeff;
```

➲ **示例4**

将用户Jeff的权限升至SYSDBA：

```
dmSQL> GRANT SYSDBA TO Jeff;
```

更改密码

您可以使用ALTER PASSWORD命令来更改用户密码。

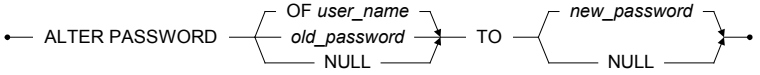


图8-3 ALTER PASSWORD命令的语法

您可以通过以下两种方法来更改用户的密码：

- 用户可通过ALTER PASSWORD <*old_password*> TO <*new_password*> 指令来更改自己的密码。其中<*old_password*>必须和存储在数据库中的原始密码相匹配。
- SYSADM可以通过ALTER PASSWORD OF<*user_name*> TO <*new_password*> 指令来更改其他用户的密码。SYSADM可以无须知道其它用户的原始密码，就能够对他们的密码进行更改。

➲ **示例1**

下例说明了用户Jeff如何为自己添加密码***xyz@#***：

```
dmSQL> ALTER PASSWORD NULL TO "xyz@#";
```

➲ **示例2**

下例说明了SYSDBA如何将用户Jeff的密码更改为***abc@#***：

```
dmSQL> ALTER PASSWORD OF Jeff TO "abc@#";
```

➲ **示例3**

下例说明了SYSADM如何将用户Jeff的密码更改为***xyz@#***：

```
dmSQL> ALTER PASSWORD OF Jeff TO "xyz@#";
```

删除或更改用户的权限

使用SQL REVOKE命令可以删除数据库中的用户权限。

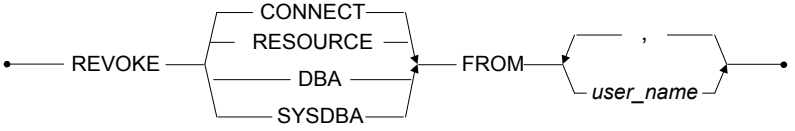


图8-4 REVOKE命令的语法

如果取消了用户的RESOURCE、DBA或SYSDBA权限，那么此操作只有在下一次连接数据库时才会生效。

SYSADM可以取消SYSADM以外用户的CONNECT、RESOURCE、DBA、SYSDBA以及ACL权限。

拥有SYSDBA权限的用户也可以取消SYSDBA以外的低权限用户的CONNECT、RESOURCE、DBA以及ACL权限。

➲ **示例1**

取消用户Jeff的SYSDBA权限：

```
dmSQL> REVOKE SYSDBA FROM Jeff;
```

执行上述命令后，Jeff将不再拥有SYSDBA权限，但仍具有DBA权限。

➲ **示例2**

取消用户Jeff的DBA权限：

```
dmSQL> REVOKE DBA FROM Jeff;
```

执行上述命令后，Jeff将不再拥有DBA权限但仍具有CONNECT权限。

➲ **示例3**

取消Jeff的CONNECT权限，使其无法登录数据库：

```
dmSQL> REVOKE CONNECT FROM Jeff;
```

取消权限	说明
SYSDBA	取消用户的SYSDBA权限，表示用户将无法授予或取消其他用户的安全权限，也无法更改其他用户的密码。 用户仍具有DBA权限。 用户创建的表、视图、定义域和同义字仍存在于数据库中。
DBA	取消用户的DBA权限，表示用户将无法创建\删除表以及授予\取消其他用户的权限。 如果用户不被授予RESOURCE权限，用户仍然具有CONNECT权限。 用户创建的表、视图、定义域和同义字仍存在于数据库中。
RESOURCE	取消用户的RESOURCE权限，表示用户将无法创建或删除表。 如果用户不被授予DBA权限，用户具有CONNECT权限。 用户创建的表、视图、定义域和同义字仍存在于数据库中。
CONNECT	取消此权限表示用户将无法登录数据库。 用户对于数据库中表和视图的所有权限都将被取消。 用户创建的表、视图、定义域和同义字仍存在于数据库中。

表8-2 取消DBMaster数据库权限层次描述表

组的管理

为了简化权限级别的管理，您可以将多个用户或多个组合并成一个组。因此您只需使用一条命令就可以对组中的所有成员授予数据库权限。尽管组与用户不同，但是在授予或取消权限时，您可以将组视为一般用户。如果授予组权限，那么组中的所有成员都将拥有此权限。

只有拥有SYSADM、SYSDBA或DBA权限的用户才能够执行以下操作：

- 创建组
- 添加组中的成员
- 删除组中的成员
- 删除组

创建组

此CREATE GROUP命令可以创建一个新组。

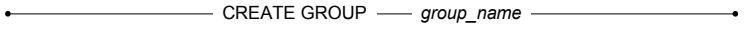


图8-5 CREATE GROUP 命令的语法

组识别（组名）是DBMaster中组名的唯一标识符。组名不能是SYSTEM、PUBLIC、GROUP或任何已存在的用户名或组名。

➲ **示例**

下例说明了如何新建一个名为COMMITTEE的组：

```
dmSQL> CREATE GROUP COMMITEE;
```

添加组中的成员

当创建新组后，您可以使用ADD *<user name or group name>* TO GROUP 命令来添加用户。

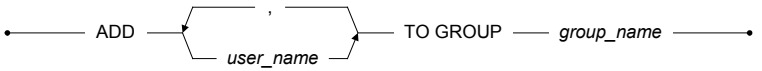


图8-6 ADD … TO GROUP 命令的语法

组不可以作为自己的组成员来添加。组成员可以是任何一个已存在用户名或组名。

➲ **示例**

下例说明了如何将用户**Jeff**和组**RD**添加到组**COMMITEE**中，并且为表**Syscom. tb_staff**授予SELECT权限：

```
dmSQL> ADD Jeff, RD TO GROUP COMMITEE;

dmSQL> GRANT SELECT ON Syscom.tb_staff TO COMMITEE;
```

执行以上命令后，组**COMMITEE**中的所有成员都将拥有表**Syscom. tb_staff**的SELECT权限。

删除组中成员

REMOVE <*user name or group name*> FROM GROUP命令可以将用户从指定组中删除。

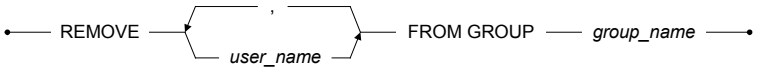


图8-7 REMOVE … FROM GROUP 命令的语法

从组中删除的成员将会失去此组的所有权限，但仍保留自身直接被授予的权限。

➲ **示例**

下例说明了如何从组 **COMMITEE** 中删除用户**Jeff**：

```
dmSQL> REMOVE Jeff FROM GROUP COMMITEE;
```

执行以上命令后，用户**Jeff**将从组**COMMITEE**中删除并且失去表**Syscom. tb_staff**的SELECT权限。

删除组

此DROP GROUP命令能够删除数据库中的组。这样组中的所有成员都将失去授予此组的权限。

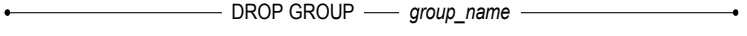


图8-8 DROP GROUP 命令的语法

➲ **示例**

下例说明了如何删除数据库中的组**COMMITTEE**：

```
dmSQL> DROP GROUP COMMITEE;
```

检验IP地址

当您希望客户端仅使用指定的IP地址连接您的数据库时，例如192.72.112.*，或希望某些指定的IP地址不能连接您的数据库时，例如192.168.0.*，您可以启用IP检验功能来控制可访问和不可访问您数据库的客户端IP地址，所有用户的设置都将存储于系统目录表SYSACL中。

系统目录表SYSACL中包括三个字段USER_NAME、ADDRESS和PRIVILEGE：

- USER_NAME： 连接数据库的用户名称和设置。
- ADDRESS： 允许连接数据库的IP地址。
- PRIVILEGE：允许或阻止连接数据库的IP地址。

用户名**PUBLIC**为保留字。如果您使用**PUBLIC**作为用户名，表示所有用户都必须满足设置。

数据库创建后，视图SYSORDERACL会自动创建并显示所有用户IP地址的信息。因此，您可以选择这些信息来检验一个IP地址是否被允许连接数据库。

启用IP检验

您可以在**dmconfig.ini**配置文件中使用**DB_StACL**关键字来启用IP检查功能，但是此项设置必须在启动数据库之前来完成。

- **DB_StACL = 1**： 启用IP检验
- **DB_StACL = 0**： 禁止IP检验 （默认）

创建规则

IP检验规则有两个：基于白名单和基于黑名单。对于同一个地址，在这两个规则约束下的结果如下表所示：

匹配 \ 次序	基于白名单	基于黑名单
仅与允许的IP地址列表匹配	允许	允许
仅与禁止的IP地址列表匹配	禁止	禁止
都不匹配	禁止	允许
都匹配	禁止	允许

设置了基于白名单时，SYSAUTHUSER的字段ACLORDER标记为0；设置了基于黑名单时，则SYSAUTHUSER的字段ACLORDER标记为1。

在允许多个IP地址连接数据库的同时，想禁止某些特定的IP地址连接数据库时，基于白名单比较合适；而在禁止多个IP地址连接数据库的同时，想允许某些特定的IP地址连接数据库时，则基于黑名单更加合适。默认的规则是**基于白名单**。

➲ **示例1a**

Glow选择了基于黑名单作为自己的IP检验规则，并且禁止客户端使用127.0.0.1以外的所有127.0.0.*段的IP地址来连接数据库：

```
dmSQL> GRANT BLOCK TO Glow '127.0.0.*';

dmSQL> GRANT ALLOW TO Glow '127.0.0.1';
```

➲ **示例1b**

Jeff选择了基于白名单作为自己的IP检验规则，并且允许客户端使用192.168.0.3以外的所有192.168.0.*段的IP地址来连接数据库：

```
dmSQL> GRANT ALLOW TO Jeff '192.168.0.*';

dmSQL> GRANT BLOCK TO Jeff '192.168.0.3';
```

用户只能选择一种IP检验规则。要想更改用户的IP检验规则，需要使用**ALTER ACL ORDER**语句。请注意，在更改规则前需取消所有已授予的约束。有关更多的约束信息，请参考*创建约束*、*取消约束*这两个章节。

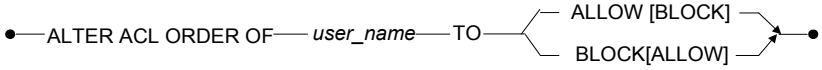


图8-9 ALTER ACL ORDER命令语法

➲ **示例2**

```
dmSQL> REVOKE BLOCK FROM Vivian ALL;

dmSQL> REVOKE ALLOW FROM Vivian ALL;

dmSQL> ALTER ACL ORDER OF Vivian TO ALLOW BLOCK;
```

创建约束

IP检验规则设置后，您可以使用GRANT ALLOW语句和GRANT BLOCK语句来为指定的IP检验规则创建一个约束。

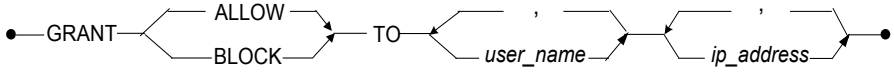


图 8-10 GRANT ALLOW/BLOCK TO USERLIST IPLIST命令语法

GRANT ALLOW语句和GRANT ACCESS语句类似。

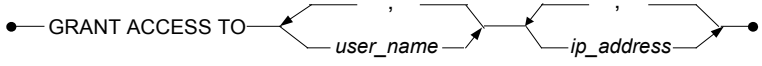


图8-11 GRANT ACCESS TO命令语法

➲ 示例

```
dmSQL> GRANT ACCESS TO vivian,joe '192.72.5.23','140.21.55.*';
dmSQL> GRANT ALLOW TO jane,jetty '192.72.12.20','140.15.45.*';
dmSQL> GRANT BLOCK TO pine,jim '192.70.16.20','139.15.45.*';
```

请注意，只能将IP地址的ALLOW权限授予组PUBLIC，如果用户将IP地址的BLOCK权限授予组PUBLIC，则会返回ERROR（6890）。此外，一般情况下，如果多个IP地址的ALLOW权限已经授予组PUBLIC，则属于该组的用户就能添加基于白名单或基于黑名单，并将其作为新的约束。

取消约束

您可以使用REVOKE ALLOW语句和REVOKE BLOCK语句来取消指定IP检验规则的约束。

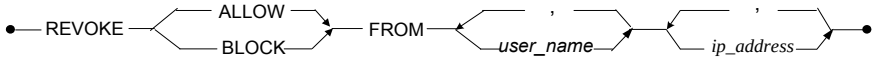


图8-12 REVOKE ALLOW/BLOCK FROM USERLIST IPLIST

REVOKE ALLOW语句和REVOKE ACCESS语句类似。

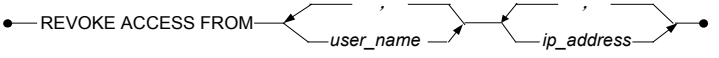


图 8-13 REVOKE ACCESS FROM USERLIST IPLIST

您可以使用REVOKE ALLOW/BLOCK FROM user_name ALL语句来依次取消指定的IP检验规则的约束，该语句中的ALL表示所有IP地址。

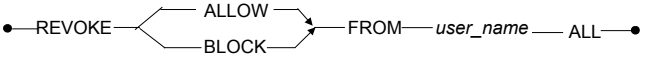


图8-14 REVOKE ALLOW/BLOCK FROM user_name ALL

➲ 示例

```
dmSQL> REVOKE ACCESS FROM vivian,joe '192.72.77.*','140.44.88.23';

dmSQL> REVOKE ALLOW FROM jane,jetty '192.72.12.20','140.15.45.*';

dmSQL> REVOKE BLOCK FROM pine,jim '192.70.16.20','139.15.45.*';

dmSQL> REVOKE BLOCK FROM glow ALL;
```

8.3 对象的权限

所谓对象是指数据库中的表、视图、表/视图的字段、定义域或同义字。DBMaster提供了对象的安全性管理，能够让用户授予（GRANT）或撤消（REVOKE）其他用户的对象权限。

定义域一旦创建好，数据库中的所有用户都可以参照它，但是只有此定义域的创建者才可以删除它。同义字本身不具有权限，同义字的权限来源于它的基础表（base table）。有关如何定义视图、定义域和同义字的详细信息请参考第6章*数据库的对象管理*。

授予对象的权限

创建对象的用户称为对象的拥有者，并且拥有该对象的所有权限。对象拥有者可以通过SQL命令GRANT <*object privilege*>授予其他用户使用该对象的权限。

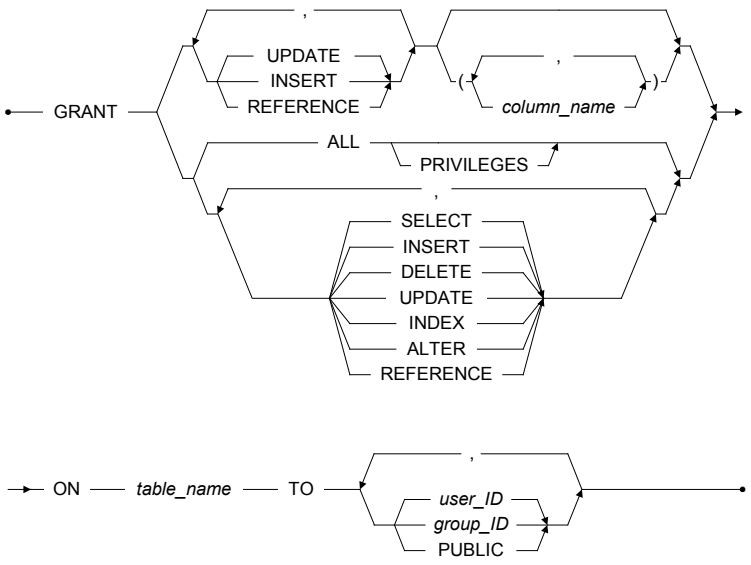


图8-15 GRANT命令的语法

拥有DBA权限的用户无论是否为对象的拥有者，都能将数据库中的表或视图权限授予其他用户。但是拥有RESOURCE权限的用户只能将他自己创建的表或视图权限授予其他用户。DBMaster提供的所有对象权限都将在表8-3中详述。

为了防止数据库中信息的损坏，您应该谨慎使用INSERT、UPDATE、DELETE权限的授予功能。ALTER和INDEX权限只能授予数据库的开发者。

UPDATE、INSERT、REFERENCE权限能够限制到字段层次。每一个字段名都必须约束在ON子句的表标识中。

权限	描述
SELECT	允许用户从表或视图中选择数据。
INSERT	允许用户向表或视图中插入行，或新增指定字段的数据。
DELETE	允许用户从表或视图中删除行。

UPDATE	允许用户更新表或视图，或更新指定字段。
INDEX	允许用户创建或删除表中的索引。
ALTER	允许用户更改表的定义。
REFERENCE	允许用户在源表上创建外键，并且此外键参照目的表或索引的主键。
ALL [PRIVILEGES]	允许用户使用上述表或视图的所有权限。PRIVILEGES是可以省略的关键字。

表8-3 授予DBMaster权限级别的描述表

使用GRANT命令的用户至少应具有CONNECT权限。您可以通过CREATE GROUP命令来创建组名，关键字PUBLIC包括所有当前以及未来加入的用户。

➲ **示例1**

用户**Jeff**执行GRANT命令授予用户**Cathy**在**TB_INFO**表上的SELECT权限：

```
dmSQL> GRANT SELECT ON TB_INFO TO Cathy;
```

➲ **示例2**

DBA授权层次的用户能够执行**GRANT**命令，授予用户**Cathy**在**Jeff**表**TB_INFO**上的读权限：

```
dmSQL> GRANT SELECT ON Jeff.TB_INFO TO Cathy;
```

➲ **示例3**

DBA授权层次的用户能够授予**Cathy**在**TB_INFO**表中字段**PHONENO**上的INSERT与UPDATE权限：

```
dmSQL> GRANT INSERT, UPDATE (PHONENO) ON Jeff.TB_INFO TO Cathy;
```

Cathy将没有权限删除**PHONENO**字段的信息。

➲ **示例4**

使用PUBLIC关键字将允许所有用户读取**Jeff.TB_INFO**表中的数据：

```
dmSQL> GRANT SELECT ON Jeff.TB_INFO TO PUBLIC;
```

取消对象权限

使用REVOKE<*对象权限*>命令可以取消用户的权限，此命令的语法图如图8-16所示。

指令REVOKE（对象权限）中的权限种类和指令GRANT（对象权限）中的权限种类相同。图中的用户名和组名分别代表数据库中的合法用户和组，PUBLIC关键字代表数据库中的所有用户。

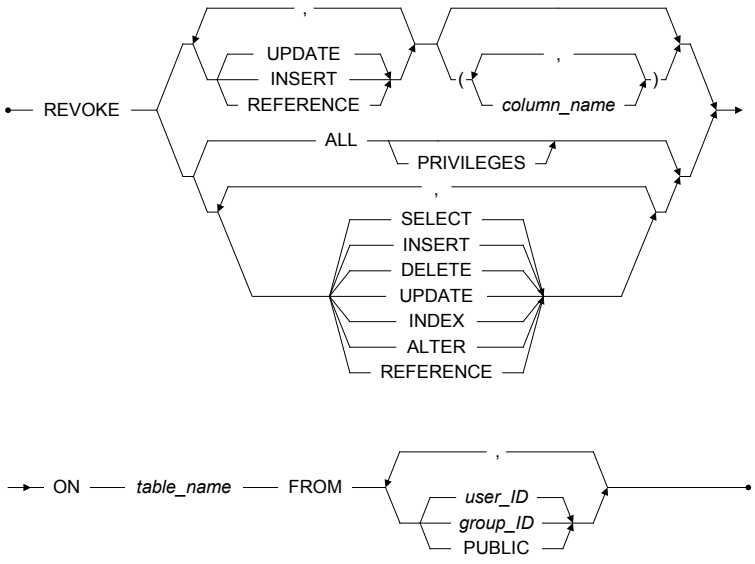


图8-16 REVOKE（对象权限）命令

➲ **示例1**

下例指令能够取消用户**Cathy**在表**TB_INFO**中的SELECT权限：

```
dmSQL> REVOKE SELECT ON TB_INFO FROM Cathy;
```

➲ **示例2**

下列指令能够取消用户**Cathy**在**Jeff.TB_INFO**上的SELECT权限：

```
dmSQL> REVOKE SELECT on Jeff.TB_INFO FROM Cathy;
```

➲ **示例3**

下列指令能够取消组**group1**在**Jeff**.**TB_INFO**表中**PHONENO**字段上的UPDATE权限：

```
dmSQL> REVOKE UPDATE (PHONENO) on Jeff.TB_INFO FROM group1;
```

➲ **示例4**

下列指令能够从**TB_INFO**表中取消曾经授予的**PUBLIC**权限：

```
dmSQL> REVOKE ALL ON TB_INFO FROM PUBLIC;
```

➲ **示例5**

下列指令能够取消用户**Cathy**和组**group2**在表**TB_INFO**上的INSERT, UPDATE和SELECT权限：

```
dmSQL> REVOKE INSERT, UPDATE, SELECT ON TB_INFO FROM Cathy, group2;
```

8.4 有关安全性管理的系统目录

下列系统目录记录了有关授权层次、权限和组的所有信息：

- **SYSAUTHUSER** — 用户的授权层次。
- **SYSAUTHTABLE** — 表的权限。
- **SYSAUTHCOL** — 用户对表中字段的INSERT, UPDATE以及REFERENCE操作权限的限定。
- **SYSAUTH** — 组名、组创建者和组中的成员数。
- **SYSACL** — 用户IP检查规则。

安全性系统目录的拥有者是SYSTEM。包括SYSADM在内，没有任何用户可以更改系统目录。有关DBMaster系统目录的详细信息请参考第21章*系统目录参考*。

9 并发控制

本章将介绍事务处理和并发控制，也将描述DBMaster如何在多用户环境下，利用锁机制来达到多用户访问和数据的准确性。事务处理一节将介绍事务的概念和用于管理事务的功能；事务隔离级别一节将描述四种事务隔离级别；多用户环境一节将描述数据库系统中，并发控制的重要性；最后将解释DBMaster的并发控制技术。

9.1 事务处理

数据库中，事务是由一个或多个SQL命令所组成的工作单元，它是一个原子性的操作。也就是说，组成事务的全部语句要不全部成功，要不全部失败。连续性（Serial）、原子性（atomic）、永久性（permanent）、一致性（consistent）、孤立性（isolated）都是事务所拥有的特性。

事务状态

事务必定属于下列状态之一：

- **Active —** 事务开始执行时会立刻进入活动状态。在这种状态下，事务可对数据库进行操作。
- **Partially Committed —** 当事务执行到最后一条语句时（如：commit work）会进入提交状态。此时包含在此事务中的语句都已执行完毕，但仍可能因为错误的发生而导致这个事务的终止。在这种情况下，因为事务发生错误而导致执行结果不能写入磁盘。另外，硬件方面的故障也可能导致事务的失败。
- **Committed —** 当事务完全执行成功时，它将进入提交状态。
- **Failed —** 当事务无法正常执行时，它将进入失效状态。其原因可能是硬件或逻辑方面的错误引起的，或用户在事务的活动状态中，终止了该事务的执行。
- **Aborted —** 当出现异常导致事务结束时，它将进入终止状态。在这种形式下，应用到数据库中的任何更改都将被回滚。

下图以图表的形式显示了事务的状态。

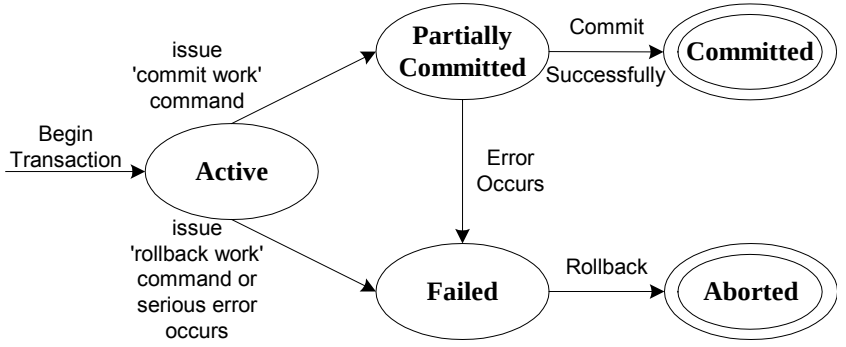


图9-1 事务的状态

事务管理

一旦连上DBMaster，一个新的事务就被自动激活并且进入激活状态。当事务结束后，DBMaster会自动执行一个新的事务。

一般来说，每当执行一个事务时，DBMaster会自动提交该事务。这就是所谓的自动提交模式。在此模式下，每个事务的生命周期也就是单个SQL语句的生命周期。也就是说，当SQL语句结束时，旧的事务也就结束，新的事务会随着新的SQL语句开始执行。每个SQL语句都是独立的事务。

为了控制事务的开始和结束，您可以通过SET AUTOCOMMIT OFF命令将执行模式更改为手动提交模式。在此模式下，事务只能通过SQL命令的（commit）来进行提交。您可以在事务结束前执行SQL语句。在事务的最后，您可以通过COMMIT命令来提交事务，或通过ROLLBACK命令来终止事务。

当您想改回自动提交模式时，您只需执行set autocommit on命令即可。默认的事务模式为AUTOCOMMIT ON。

注意 *一旦事务确认之后，所有分配给该事务的资源都将被释放。*

使用保存点

保存点是在事务处理的过程中可任意声明的一个媒介点。当事务执行时，您可以指定事务将回滚到哪一个保存点。

举例来说，在事务处理的过程中要执行一系列命令，当执行到第二十条命令时如果出现了错误，但保存点只标记到第十五条和第十六条命令之间，那么您就可以回滚到第十五条命令，保存前十五条命令的执行结果。同时用户将由第十六条SQL语句开始执行校正后的命令，并且无需终止和再次提交事务，如图9-2示例。

如果用户没有在第十五条命令和第十六条命令之间标记保存点，那么事务将被终止，并且前十五条命令将会再次提交。这是很不方便和费时的。在此我们利用保存点来有效的解决这个问题。

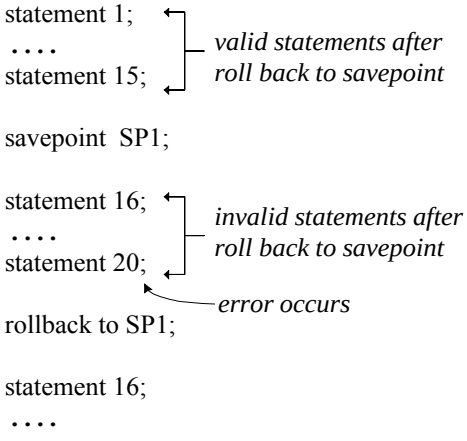


图9-2 使用保存点

SAVEPOINT命令用于标记一个保存点，ROLLBACK TO … 命令用于回滚一个指定的保存点。

➲ **示例1**

SAVEPOINT命令如下：

```
dmSQL> SAVEPOINT <savepoint_name>;
```

➲ **示例2**

ROLLBACK TO …命令如下：

```
dmSQL> ROLLBACK TO <savepoint_name>;
```

用户可以指定**<*savepoint_name*>**。当回滚至保存点时，保存点后的系统资源会被重新分配而锁将被释放。

9.2 事务隔离级别

事务并发处理

在同一个数据库中并发地执行多个事务可能会遇到一些不希望发生的现象。它们通常被称为**脏数据**，**不可重复读**和**幻影现象**。

除非特殊注明，否则下面的描述都是基于以下的数据：

表**table1**中**c1**字段上存在三个值：1、3 和 5。两个事务**T1**和**T2**在该表上执行并发操作。

脏数据

定义：一个事务读取了被另一个并发执行的**未提交**的事务所修改的数据。

➲ **示例**

```
     T1                    T2
------------          ------------
                      Insert 4
Select c1<5
……                    ……
                      Commit or rollback
```

事务**T1**执行了SELECT语句select c1 <5，返回的结果为c1 = 1, 3, 4。然而，由于事务**T2**可能会撤销，所以T1的结果便因为**T2**的撤销而可能不正确。

不可重复读

定义：一个事务前后两次读取的同一数据的值并不相同，即在两次读取中间，数据被另一个事务所更改。

➲ 示例

```
     T1              T2
------------  -----------------
Select c1<5
              Update c1=2 where c1=1
              Commit
Select c1<5
```

事务**T1**第一次执行SELECT语句select c1<5，返回的结果为c1 = 1, 3。随即事务**T2**将值1更新为2，并**提交**了该更新操作。这时，**T1**再次执行同样的查询语句，返回的结果却是c1 = 2, 3。可见**T1**执行两次**同样的语句**却分别返回不同的结果。

幻影现象

定义：一个事务两次执行相同的SELECT语句，两次得到的结果集中却包含不同的返回值个数。在第二次返回的结果中至少有一项不是在第一次的返回结果中出现的。

➲ 示例

```
        T1                    T2
--------------------        ------------
Select c1<5
                               Insert 4
                                 Commit
Select c1<5
```

首先，事务**T1**执行select语句并返回结果c1 = 1, 3。然后事务**T2**插入4并提交。过后**T1**执行同样的select语句，但返回结果为c1 = 1, 3, 4。**T1**执行两次同样的语句却得到不同的结果。第二次执行返回的一些结果并没有在第一次中返回。在该例子中，第二次结果中的值4就是幻影读取。

事务隔离级别

ANSI/ISO SQL 定义了四种事务并发级别：

隔离级别	脏数据	不可重复读	幻影现像
只读未提交	可能	可能	可能

只读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

设置DBMaster的事务隔离级别

DBMaster提供三种方式来设置事务隔离级别：通过dmconfig关键字，ODBC函数以及dmsql中的SQL语法。

DMCONFIG关键字

相关的关键字是**DB_IsoLv**。

```
DB_IsoLv {1,2,3,4}
1 : READ UNCOMMITTED
2 : READ COMMITTED
3 : REPEATABLE READ
4 : SERIALIZABLE
```

默认值为***1***。

DB_IsoLv的值就代表着每个事务的默认隔离级别。例如，**DB_IsoLv = 3**，则每一个事务的默认隔离级别就是**不可重复读**。

ODBC函数

函数SQLSetConnectionOption用来设置事务隔离级别，而函数SQLGetConnectionOption用来获取当前的事务隔离级别。

```
SQLSetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, level)

Level : {

SQL_TXN_READ_UNCOMMITTED,

SQL_TXN_READ_COMMITTED,

SQL_TXN_REPEATABLE_READ,

SQL_TXN_SERIALIZABLE

}
```

```
SQLGetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, &level)
```

SQL 语法

```
Type "SET TRANSACTION ISOLATION LEVEL [level]" in the command line.

      [level] :

{ READ COMMITTED
           | READ UNCOMMITTED
           | REPEATABLE READ
           | SERIALIZABLE
             }

Type "CALL GETSYSTEMOPTION ('isolv',?);" in the command line in the
dmsql will get information about isolation level.
```

➲ **示例**

获得事务隔离级别的信息：

```
dmSQL> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

dmSQL> CALL GETSYSTEMOPTION ('isolv',?);

OPTION_VALUE :SQL_TRANSACTION_READ_UNCOMMITTED
```

9.3 多用户环境

当多个用户访问数据库时，应该考虑到在同时存取数据时会发生什么问题。

会话

连接是DBMaster和用户之间沟通的桥梁，这个沟通桥梁是通过共享内存和网络来建立的。

在使用数据库的资源前，您必须通过下列SQL指令来建立与DBMaster的连接。

➲ **示例**

将用户连接至DBMaster数据库：

```
dmSQL> CONNECT TO database_name user_name password;
```

当用户连接至DBMaster数据库时，指定的连接称作会话。会话的持续时间是从用户连接至DBMaster数据库到从DBMaster断开为止的整个过程。一个会话每次只能拥有一个活动事务。

并发控制的重要性

在多用户数据库系统环境下，多个用户可以同时连接数据库。这会导致多个事务同时更新同一个数据库。

如果没有并发控制的话，以下状况将导致数据的不一致性：

- 更新丢失的问题。
- 临时更新的问题。
- 不正确加总的问题。

更新丢失问题

当两个事务几乎要同时更新同一笔数据时，会产生更新丢失的问题。

➲ 示例

事务T1和T2都有权读取和更改X的值，但是它们使用了不同的计算方法来更改X的值。结果会导致每一个事务都包含不同的值。T1在读取X后会将更新后的值写入数据库中，但在此之前X的值也会被T2来更改。T2将更新后的X也写入数据库，覆盖了T1的值。所以T1提交的值将会丢失：

```
      T1                        T2
--------------            --------------
read(X);
                          read(X);
X = X - N;
                          X = X + M;
write(X);
                          write(X);
```

临时更新问题

当事务更新一个值后，如果没有立刻回滚该值，而在另一事务更新此值后才回滚，那么此时就会出现临时更新的问题。

➲ 示例

事务T1在读取、更改X值后，会将它写入数据库中接着执行其它命令。当事务T1继续执行时，事务T2会读取X的值并且更新它，同时将它写入数据库中。那么事务T1的操作就会失败，并且必须回滚所有的值来恢复数据库的初始状态。数据库管理系统会将X值恢复到它的初始状态，来覆盖事务T2写入的值。由事务T2计算出来的X值，只是临时存在于数据库系统中。

```
      T1                        T2
--------------            --------------
read(X);                  read(X);
X = X - N;
write(X);                 X = X + M;
                          write(X);
```

```
rollback;
```

不正确加总问题

当一个事务正在加总一笔记录时，另一个事务同时也在更改这笔记录，此时就会出现不正确加总的问题了。

➲ **示例**

事务T1在加总X和Y值的同时，事务T2也在更改这些值。事务T2在事务T1使用X之前更改了X值，而在事务T1使用Y之后更改了Y值。这将导致事务T1使用部分更改前的值，部分更改后的值来计算总和。因此当两个事务都完成时，数据库中的总值就不正确了。

```
     T1                        T2
--------------           --------------
sum = 0;
                         read(X);
                         X = X - N;
                         write(X);
read(X);
sum = sum + X;
read(Y);
sum = sum + Y;
                         read(Y);
                         Y = Y + N;
                         write(Y);
```

有许多方法可以解决并发控制的问题，例如锁和时间戳。下一节将介绍如何在DBMaster中利用锁机制来解决事务间的并发控制。

9.4 锁

这一节将首先描述锁的概念。接着介绍DBMaster的锁定机制，包括锁的单位和锁的模式。最后将介绍如何处理死锁。

锁的概念

一般来说，多用户的数据库系统会利用多种锁定机制来同步访问当前事务。在访问数据对象之前，例如表和记录，事务必须先锁定这些数据对象。

在DBMaster中，所有的锁都是自动的。所有的SQL命令都会自动引发锁，用户无需亲自锁定数据库中的任何数据对象。

共享锁和互斥锁

通常多用户的数据库系统是利用三种类型的锁来实现多人可读、单人可写的操作。

- **共享锁（S）—** 共享锁是指事务中包含了对数据对象的读取操作。它支持较高的同步性。也就是说，几个事务可以同时对同一数据对象分配共享锁。
- **更新锁（U）—** 更新锁是指事务正要对数据对象执行更新操作或正在执行更新操作。更新锁与共享锁兼容，但与互斥锁不兼容。一个时间一个对象只能有一个更新锁。
- **互斥锁（X）—** 互斥锁是指事务中包含了对数据对象的更新操作。在某个事务占有互斥锁而未释放之前，没有任何事务可以获得互斥锁。

两阶段锁定

两阶段锁定协议用于确保事务的连续性。在此协议中，一个事务释放锁定对象之前必须提出锁请求。

此协议可以分成以下两个阶段：

- **扩展（成长）阶段（Expanding （growing） phase） —** 此阶段允许事务提出一个新的锁定请求，但不允许解锁请求。
- **释放阶段（Shrinking phase） —** 此阶段允许事务释放扩展阶段的锁请求，但不允许提出任何新的锁定请求。

DBMaster目前使用两阶段锁定协议来达到连续事务的同步控制。

死锁

当两个以上的事务同时等待对方解除锁定的时候，死锁的情况就发生了。

➲ **示例**

如果T1在等待T2释放共享锁X，而T2在等待T1释放共享锁Y。此时，死锁就发生了，并且系统将进入无穷尽的等待状态：

```
     T1                        T2
--------------           --------------
share_lock(Y);
read(Y);
                         share_lock(X);
                         read(X);
exclusive_lock(X);
(T1 waits for T2)        exclusive_lock(Y);
                        (T2 waits for T1)
```

锁的粒度

DBMaster对于数据的锁定有三种锁定粒度：关系 （表）、页和记录（行）。 一个关系包含几个页，一页又包含几条记录。

较高层次的锁定隐含着对所属较低层次的对象的锁定。例如：如果用户拥有某个关系的互斥锁（X 锁），则表示对于属于这个关系的所有页和记录都将拥有该互斥锁（X 锁）。因此没有任何用户可以存取属于该表的记录和页。然而当一个用户获得某个记录的互斥锁（X 锁）时，其他

用户也可以同时拥有别的记录的互斥锁（X lock）。在同一层次中，两个对象的X锁并不会相互影响。下图显示了DBMaster的锁粒度。

图9-3 锁的粒度

使用较高层次的锁粒度将导致较低层次的数据并发性。比较来说，使用较高层次的锁粒度将会占用较少的系统资源（例如共享内存）。显而易见，如何选取锁的粒度是系统资源和数据并行性的考虑因素。DBMaster默认的锁粒度是行，在创建表时可以指定不同的锁粒度。

锁的类型

DBMaster的锁模式主要分为三种：共享锁（S）、更新锁（U）和互斥锁（X）。在同一时刻，多个用户可以同时拥有共享锁（S），但是只有一个用户可以拥有互斥锁（X）或更新锁（U）。除了这三种锁之外，DBMaster还支持所谓的意向锁（*intention lock*）。

当一个数据对象被锁定时，系统会自动地给较高层次的锁定对象分配一个意向锁。例如：如果某条记录拥有共享锁（S lock），那么包含该记录的页将产生一个意向S（IS）锁，并且在包含此记录的关系上将产生一个IS锁。

DBMaster提供的意向锁有以下几种模式：

- **IS—**表示较低层次的锁定粒度拥有S锁。
- **IU—**表示较低层次的锁定粒度拥有U锁。
- **IX—**表示较低层次的锁定粒度拥有X锁。
- **SIX—**表示当前层次拥有S锁，而较低层次拥有X锁。这是S锁和IX锁的结合。

- **SIU—**表示当前层次拥有S锁，而较低层次拥有U锁。这是S锁和IU锁的结合。
- **UIX—**表示当前层次拥有U锁，而较低层次拥有X锁。这是U锁和IX锁的结合。

表9-1列出了所有锁定模式的兼容性。T代表兼容，指对同一个数据对象的两种锁模式是可以同时存在且兼容的。F代表不兼容，指对同一数据对象的两种锁模式是不能同时存在的。

如果一个数据对象的锁请求与另一个已存在的锁冲突的话，那么这个锁请求只有等到已存在的锁释放后才能执行，否则就是等待超时。如果回传给用户的错误信息是“锁超时”，就表示等待锁定的时间已经超过了，默认的等待时间是5秒。然而用户可以根据个人要求，通过**dmconfig.ini**中的**DB_LTimO**关键字来指定不同的等待时间。

➲ **示例**

下例将等待时间设置为8秒：

```
DB_LTimO = 8;
```

	N	IS	S	IU	SIU	IX	U	SIX	UIX	X
N	T	T	T	T	T	T	T	T	T	T
IS	T	T	T	T	T	T	T	T	T	F
S	T	T	T	T	T	F	T	F	F	F
IU	T	T	T	T	T	T	F	T	F	F
SIU	T	T	T	T	T	F	F	F	F	F
IX	T	T	F	T	F	T	F	F	F	F
U	T	T	T	F	F	F	F	F	F	F
SIX	T	T	F	T	F	F	F	F	F	F
UIX	T	T	F	F	F	F	F	F	F	F
X	T	F	F	F	F	F	F	F	F	F

表 9-1 锁模式的兼容性矩阵

处理死锁

通过分析“wait for”图表，DBMaster自动检测死锁的情况。假如检测到死锁，一个事务会被终止从而解决这个死锁问题。

➲ **示例**

当事务T2对Y提出X锁定时，DBMaster会检测到死锁的发生。因此事务T2会被终止来解决这个死锁问题，同时执行事务T2的用户将接收到“因为死锁而终止事务”的错误提示信息：

```
     T1                    T2
--------------          ------------
share_lock(Y);
read(Y);
                        share_lock(X);
                        read(X);
exclusive_lock(X);
(T1 waits for T2)       exclusive_lock(Y);
                        (T2 waits for T1)

                        T2 aborted by DBMaster
```

10 触发器

在DBMaster数据库服务器中，触发器是相当有用且功能强大的。无论事件是由哪一个用户或哪一个应用程序产生，触发器都会自动执行事先定义好的命令来响应指定的事件。

触发器可以设定数据库间不同表的关联操作，其所能达到的效果可能是一般SQL指令所不能达到的。数据库无需用户或应用程序的作用，就可以协调复杂的或非常规的数据库操作。

触发器可运用到以下情形：

- 实现商务规则
- 对数据库的活动开启审计跟踪
- 从现有的数据中导出附加值
- 跨多表复制数据
- 实现安全认证
- 控制数据完整性
- 定义非常规的完整性约束

使用触发器时，应使用限制来避免数据库间因为复杂的相互依赖关系而导致的追踪和更改时的困难。当使用标准的SQL语句和完整性约束无法实现想要的功能时，触发器可以帮助您解决这个问题。

10.1 触发器的组件

DBMaster将触发器的定义储存在系统目录中。

每一个DBMaster触发器都由6个主要的组件组成：

- **触发器的名称（Trigger Name）—** 唯一标识一个触发器。
- **触发器执行时间（Trigger Action Time）—** 触发器何时被激发。
- **触发事件（Trigger Event）—**为了响应用户行为而在数据库中出现的特定动作，例如向表中插入数据。
- **触发表（Trigger Table）—** 在其上定义触发器的表。
- **触发器行为（Trigger Action）—** 触发事件发生时，执行的SQL语句或存储过程。
- **触发类型（Trigger Type）—** 触发的类型。

创建触发器时，必须具备以上所有组件。除此之外，触发器还具有一些可选的组件，如REFERENCING子句。

触发器的名称

触发器名称是触发器的唯一标识。触发器名称最多由128个字符组成，可以包含字母、数字、下划线字符以及#和$等符号，触发器的名称不能以数字开头，且不能包含空格。

触发器执行时间

触发器执行时间是指触发器是在激活它的SQL语句之前，还是之后被激活。触发器执行时间可由BEFORE和AFTER参数来设定。参数BEFORE是指触发器会在触发事件的SQL语句之前被激活，参数AFTER是指触发器会在触发事件的SQL语句之后被激活。每一个触发器只能指定一个触发器执行时间。

触发事件

触发事件是指激活触发器的数据库操作。触发事件可以是对一个触发表的新增、更新或删除。一个触发只能有一个触发事件。要执行多个触发事件，必须激活多个触发。

触发表

触发事件作用在与之相关的触发表上。触发表必须是基本表（base table），而非临时表、视图或同义字。一个触发器只能应用于一张触发表。

触发器行为

触发器行为是指触发器被激活时执行的命令。触发器的行为可以是INSERT、UPDATE、DELETE或EXECUTE PROCEDURE语句或是SQL语句块。每一个触发器只能拥有一个触发器行为。

触发类型

触发类型指定每一个触发事件将执行多少次触发器行为。触发器有两种类型： 行触发（row triggers）和语句触发（statement triggers）。关键字**FOR EACH ROW**用于指定行触发。针对行触发而言，当触发事件有一笔记录被更改时，触发器行为就会执行一次。关键字**FOR EACH STATEMENT**用于指定语句触发。针对语句触发而言，每一个触发事件，触发器行为只会执行一次。

REFERENCING子句

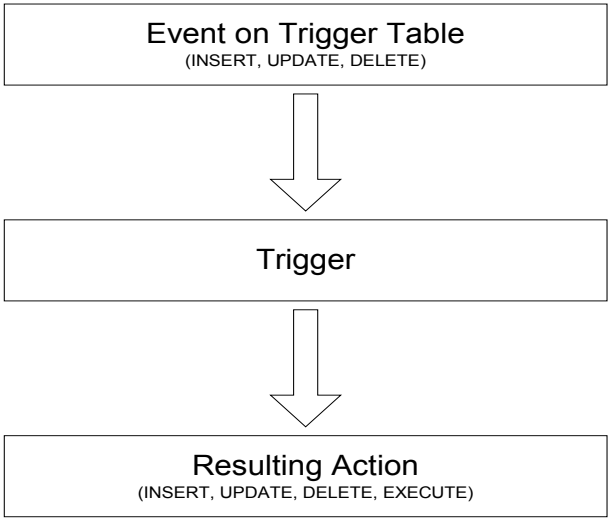
REFERENCING子句用来定义字段中旧值与新值的关联名称。主要用于当您的表名称是OLD和NEW时，避免名称混淆。

10.2 触发的操作

每当用户或应用程序引起一个触发事件时，DBMaster都会检查是否应该执行已存在的触发器。在数据库中执行触发器，可以确保DBMaster处理各个应用程序间数据的一致性。因此当一个指定事件发生时，与其相关联的行为也会被执行。

触发器可以实现定义域、字段和参照完整性以及一些非传统的完整性约束。

触发器没有所有者，但是需要和表关联。



10.3 创建触发器

在一张表中，CREATE TRIGGER命令是用来创建一个与此表关联的新触发器。只有此表的拥有者或具备DBA权限才能执行这个命令。同时，您必须具有触发器定义里所有参考对象的权限，才能成功的创建此触发器。

基本的要件

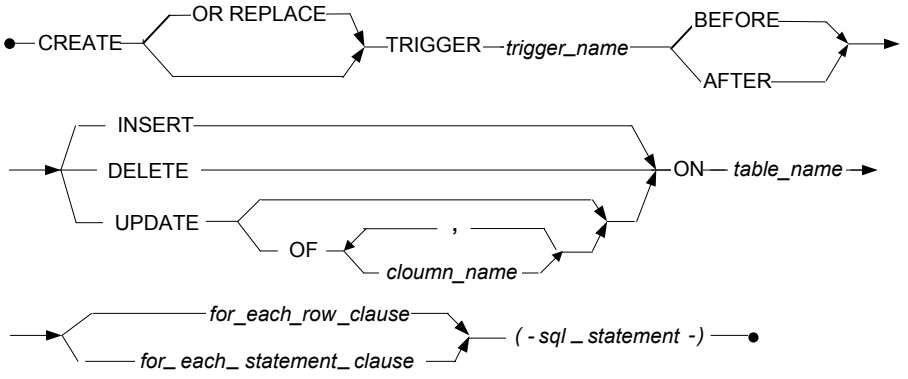
所有CREATE TRIGGER语句中都至少包含以下条件：

- 触发器名称
- 触发器执行时间（BEFORE/AFTER）
- 触发事件
- 触发表
- 触发类型（ROW/STATEMENT）
- 触发器行为

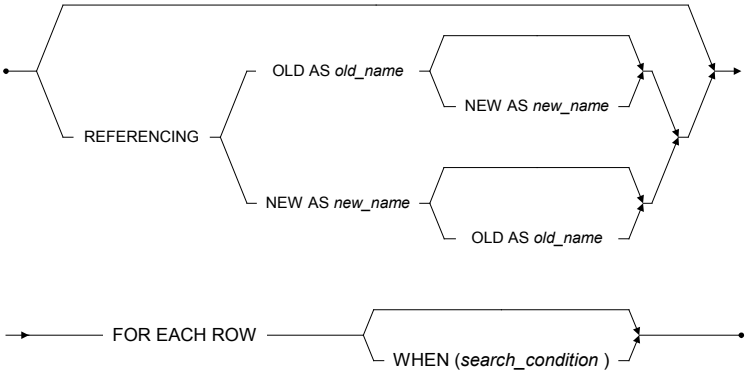
安全权限

所有触发器行为中的SQL语句，都是以和触发表拥有者同样的权限来执行，而不是以触发事件的使用者权限来执行的。如果触发器存在，任何执行触发事件的用户都可以激活触发器。

CREATE TRIGGER语法



FOR EACH ROW子句



FOR EACH STATEMENT子句

图10-2 CREATE TRIGGER语句的语法

OR REPLACE用于重建已存在的触发器，即用户可以使用该子句更改已存在触发器的定义。

➲ **示例**

下例在**tb_staff**表上创建或替换一个触发器。

```
dmSQL> CREATE OR REPLACE TRIGGER tr_staff_insert AFTER INSERT ON
tb_staff
       FOR EACH ROW WHEN (new.ID > 0)
       (INSERT INTO tb_salary(new.ID, new.Name,NULL, NULL, NULL));
```

指定触发器执行时间

针对每一张表中的同一个事件（INSERT、DELETE或UPDATE），您可以使用触发时间和触发类型来创建四种触发。它们是BEFORE/FOR EACH ROW、AFTER/FOR EACH ROW、BEFORE/FOR EACH STATEMENT和AFTER/FOR EACH STATEMENT。

一个Before/For Each Statement触发只能在触发语句执行前执行一次，并且只能执行一次，这是指在触发事件执行之前。一个After/For each statement触发只能在触发语句执行后执行一次，并且只能执行一次，这是指在触发事件完成之后。值得注意的是：即使触发事件没有产生任何一笔记录，之前或之后的语句触发器也必须被执行。

BEFORE / AFTER INSERT / DELETE触发事件

下例说明了如何在INSERT / DELETE触发事件之前或之后创建触发器，触发器行为可以通过 <*sql_statement*> 来指定。

➲ **示例1**

在**tb**表上，为INSERT事件定义四个触发器：

```
dmSQL> CREATE TRIGGER tr1 BEFORE INSERT ON tb FOR EACH STATEMENT
<sql_statement>;

dmSQL> CREATE TRIGGER tr2 BEFORE INSERT ON tb FOR EACH ROW
<sql_statement>;

dmSQL> CREATE TRIGGER tr3 AFTER INSERT ON tb FOR EACH ROW
<sql_statement>;

dmSQL> CREATE TRIGGER tr4 AFTER INSERT ON tb FOR EACH STATEMENT
<sql_statement>;
```

➲ **示例2**

在**tb**表上，为DELETE事件定义四个触发器：

```
dmSQL> CREATE TRIGGER tr1 BEFORE DELETE ON tb FOR EACH STATEMENT
<sql_statement>;

dmSQL> CREATE TRIGGER tr1 BEFORE DELETE ON tb FOR EACH ROW
<sql_statement>;

dmSQL> CREATE TRIGGER tr1 AFTER DELETE ON tb FOR EACH ROW
<sql_statement>;

dmSQL> CREATE TRIGGER tr1 AFTER DELETE ON tb FOR EACH STATEMENT
<sql_statement>;
```

BEFORE/AFTER UPDATE触发事件

对于UPDATE事件情况有一些不同。 您可以创建两种类型的UPDATE触发：UPDATE <*table*>触发或UPDATE OF <*column*>触发。当表更新时，即执行 UPDATE <*table*>触发；当指定字段被更新时，即执行UPDATE OF <*column*>触发。在一张表上，您可以创建一个UPDATE <*table*>触发或多个UPDATE OF <*column*>触发。UPDATE OF <*column*>触发可以包含多个字段，但表中所有UPDATE OF <*column*>触发字段必须是互斥的。

➲ 示例

在表**tb_salary**的字段**basepay**、**bonus**上创建一个字段触发器**tr_UpdateColumn**，其中拥有四个字段：**id**、**name**、**basepay** 和**bonus**：

```
dmSQL> CREATE TRIGGER tr_UpdateColumn AFTER UPDATE OF basepay,bonus ON
tb_salary
                          FOR EACH ROW
                       (INSERT INTO tb_OldSalary VALUES (old.basepay,
old.bonus));
```

如果您想在字段**bonus**上创建第二个UPDATE字段触发器**tr_UpdateBonus**，这个指令将会失败，因为**bonus**已经在触发器**tr_UpdateColumn**中出现：

```
dmSQL> CREATE TRIGGER tr_UpdateBonus AFTER UPDATE OF bonus,tax ON
tb_salary
                          FOR EACH ROW
                          (INSERT INTO tb_OldTax VALUES (old.bonus,
old.tax));
ERROR (6150): [DBMaster] the insert/update value type is incompatible
with column data type or compare/operand value is incompatible with
column data type in expression/predicate
```

如果一张表中有四个字段，那么针对同一类型的触发（例如：BEFORE/FOR EACH ROW触发），您最多可以创建四个Update字段触发或一个UPDATE表触发。

FOR EACH ROW / FOR EACH STATEMENT子句

针对每一个触发事件，FOR EACH STATEMENT子句代表执行一次触发器，并且只激活一次。 即使触发事件没有产生任何一笔记录，触发器也会被激活。

FOR EACH ROW子句指当触发事件更改记录时，将为每一行激活一次触发器。如果触发事件没有更改记录，那么触发器将不被激活。OLD和

NEW参数用于指示触发器行为中的被触发的表记录。参数OLD标识了在执行触发事件前，触发表中的记录。参数NEW标识了在执行触发事件后，触发表中的记录。

➲ **示例1**

下例说明了如何在**tb_Sales**表上创建一个Update字段触发器。其中的**totSales** 是从**unitPrice**和**unitSale**中计算得来的。并且**unitPrice**和**unitSale**都是触发字段。

```
dmSQL> CREATE TRIGGER tr_TotalSale AFTER UPDATE OF unitPrice, unitSale
ON tb_Sales FOR EACH ROW
                            (UPDATE tb_Sales
                                  SET totSales = new.unitPrice *
new.unitSale);
```

➲ **示例2**

下例中有四个触发器：

```
dmSQL> CREATE TRIGGER tr_BeforeUpdatePro BEFORE UPDATE ON tb_Orders
                     FOR EACH STATEMENT
                    (EXECUTE PROCEDURE checkPrivilege);


dmSQL> CREATE TRIGGER tr_BeforeUpdate BEFORE UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO tb_Old_Value (old.customer,
old.amount));


dmSQL> CREATE TRIGGER tr_AfterUpdate AFTER UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO tb_New_Value (new.customer,
new.amount));
```

```
dmSQL> CREATE TRIGGER tr_AfterUpdatePro AFTER UPDATE ON tb_Orders
                        FOR EACH STATEMENT
                       (EXECUTE PROCEDURE Log_Time);
```

如果您通过一个更新语句来更改**tb_Orders**表中的两行记录，其执行的效果和顺序将如下所示：

1. 调用程序checkPrivilege。
2. 在Log_Old_Value表中插入一笔记录。
3. 更新一笔记录。
4. 在Log_New_Value表中插入一笔记录。
5. 在Log_Old_Value表中插入一笔记录。
6. 更新一笔记录。
7. 在Log_New_Value表中插入一笔记录。
8. 调用程序Log_Time。

存储过程中不能包含COMMIT、ROLLBACK或SAVEPOINT事务控制语句。触发器仅能指定一个触发器行为，且必须用括号注明。

使用REFERENCING子句

当您创建一个行触发时，*<sql_statement>*（或行为本身）应该指出是否要参照触发事件执行之前/之后的字段。例如：当您更新一笔销售记录的价格时，您会希望记录价格的新/旧值。为了达到此目的，您可以参照*FOR EACH ROW / FOR EACH STATEMENT子句*章节中的例2来使用OLD和NEW参数。

然而在少数情况下，表名中也会包含字段NEW或OLD。如果这种情况发生，您可以使用参考子句来定义关联名称。参考子句允许您创建两个与字段名结合的前缀：一个用来参考字段的旧值，一个用来参考新值。这些前缀称作关联名称，您可以使用参数OLD和NEW来指示其关联名称。

➲ **示例**

```
dmSQL> CREATE TRIGGER tr_log_price AFTER UPDATE OF price ON New
                                    REFERENCING OLD as pre NEW as post
```

```
                                    FOR EACH ROW
                                    (INSERT INTO logTbl
                                     VALUES (item_no, today(), pre.price,
                                     post.price));
```

在上例中，触发表为NEW，所以我们在触发行为中定义关联名**pre**和**post**。REFERENCING子句只对行触发有效，不允许在语句触发中使用。触发事件是INSERT，并且新增的记录没有旧值，所以旧值不能在新增事件中使用。 同样，如果触发事件是DELETE，删除的记录没有新值，所以新值不可以在触发事件中使用。对于UPDATE事件的触发，新值和旧值都是有效的。

WHEN条件子句的使用

您可以在FOR EACH ROW触发器之前放置WHEN条件子句，使触发器依据布尔表达式的结果来执行。When子句包括一个关键字WHEN，后面跟随一个括号内的条件语句。When子句应该放在触发时间之后，触发器行为之前。同样WHEN子句不能用于语句触发的定义中，它只能用于行触发中。

➲ **示例1**

当客人产生抱怨时，下例触发器将把客人的申诉记录存放于**tb_logComplain** 表中 （假设 **'c'** 代表申诉电话）。

```
dmSQL> CREATE TRIGGER tr_log_complain AFTER INSERT ON tb_Customer_Call
                                       FOR EACH ROW
                                       WHEN (new.call_code = 'c')
                                       (INSERT INTO tb_logComplain
                                        VALUES (Today(), Cus_Name));
```

如果WHEN条件包含在触发器定义中，则WHEN子句会针对每笔记录来计算。如果WHEN条件子句对一笔记录的计算结果为TRUE，那么触发器行为会在该记录上被激活。如果WHEN条件子句对一笔记录的结果为

FALSE或未知，那么触发器行为不会在该记录上被激活。WHEN条件的结果只会影响触发器行为的执行，但对触发语句毫无影响。

➲ **示例2**

创建三个触发器来记录**tb_staff**表的插入、删除和修改操作：

```
dmSQL> CREATE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
                                        FOR EACH ROW
                                        (INSERT INTO tb_salary
                                       VALUES (new.Id, new.Name, NULL,
NULL, NULL));

dmSQL> CREATE TRIGGER tr_staff_update AFTER UPDATE ON tb_staff
                                        FOR EACH ROW
                                        (INSERT INTO tb_staff_bak
                                      VALUES (old.Id, old.Name,new.Id,
new.Name));

dmSQL> CREATE TRIGGER tr_staff_upd AFTER DELETE ON tb_staff
                                        FOR EACH ROW
                                        (INSERT INTO tb_staff_bak
                                         VALUES (old.Id, old.Name,
                                         NULL, NULL));
```

➲ **示例3**

如果主键被更改了，那么所有级联的外键也会随之更改。 假设**deptNo**是**tb_dept**表的主键，而**id**是**tb_staff**表的外键。

```
dmSQL> CREATE TRIGGER tr_dept_update BEFORE UPDATE OF deptNo ON tb_dept
                                       FOR EACH ROW
                                       WHEN (NEW.deptNo <> OLD.deptNo)
```

```
                                (UPDATE tb_staff SET tb_staff.ID =
NEW.deptNo
                                     WHERE tb_staff.ID = OLD.deptNo);
```

➲ **示例4**

如果主键被删除了，那么所有级联的外键也会随之删除。

```
dmSQL> CREATE TRIGGER tr_dept_delete BEFORE DELETE ON tb_dept
                                    FOR EACH ROW
                                    (DELETE FROM tb_staff
                                     WHERE tb_staff.ID = OLD.deptNo);
```

➲ **示例5**

如果主键被更新了，那么所有级联的外键都可设置为空（NULL）。

```
dmSQL> CREATE TRIGGER tr_dept_delete BEFORE UPDATE ON tb_dept
                                    FOR EACH ROW
                                    (UPDATE tb_staff set ID = NULL
                                     WHERE tb_staff.ID= OLD.deptNo);
```

➲ **示例6**

如果某公司的零件存货低于给定的数量，那么您可以重新安排此零件的需要量后，再订货。 在表**tb_pending_orders**中记录零件的号码和数量以备将来使用。

tb_Inventory：part_no int、parts_on_hand int、reorder_level int、reorder_qty int

tb_pending_orders：part_no int、qty int、order_date date

```
dmSQL> CREATE TRIGGER tr_reorder AFTER UPDATE OF parts_on_hand ON
tb_Inventory
                                FOR EACH ROW
                                WHEN (new.parts_on_hand <
new.reorder_level)
```

```
                                    (INSERT INTO tb_pending_orders
                                     VALUES (new.part_no, new.reorder_qty,
today()));
```

指定触发器行为

当触发事件发生时，触发器行为就是要被执行的SQL语句。触发器行为可以是插入、删除、更新、执行过程语句或SQL语句块，但不允许执行其它的语句。存储过程不能包含COMMIT、ROLLBACK或SAVEPOINT事务控制语句。触发器仅能指定一个触发器行为，并且必须用括号注明。

➲ **示例**

通过下例语句，可在表**tb_staff**上创建一个触发器：

```
dmSQL> CREATE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
                                      FOR EACH ROW WHEN (new.ID > 0)
                         (INSERT INTO tb_salary(new.ID, new.Name,NULL,
NULL, NULL));
```

此例中的触发器名称为**tr_staff_insert**。AFTER选项代表在**tb_staff**表上执行INSERT 语句后，触发才会被激活。其中的触发事件是INSERT，触发表为**tb_salary**。触发类型为FOR EACH ROW，所触发的SQL行为是INSERT。

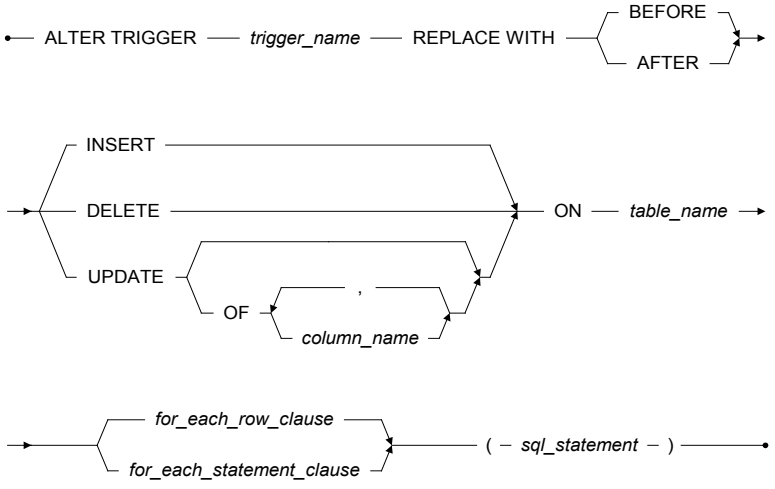
```
dmSQL> CREATE TRIGGER tr_salary_Del AFTER DELETE ON tb_salary
                                    FOR EACH ROW
                                   (INSERT INTO tb_old_salary
                                    VALUES (Old.name));
```

上例中，当从tb_salary表中删除一条记录时，触发器tr_salary_Del将会把已删除的用户名添加到tb_old_salary表中。您无法在临时表、视图或系统表上创建一个触发器。

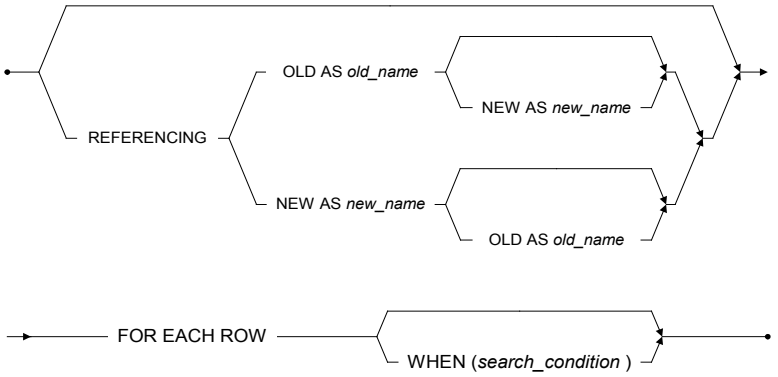
10.4 更改触发器

触发器不能被更改，但是可以替换它的定义。您可以使用ALTER TRIGGER语句来替换触发器的定义。

ALTER TRIGGER 语法



FOR EACH ROW 子句



FOR EACH STATEMENT 子句

图10-3 ALTER TRIGGER 命令的语法Figure 4

触发器行为的替换

您可以通过ALTER TRIGGER tr1 REPLACE WITH语句来替换触发器行为。

➲ **示例1**

如果一位经理离职，那么他的信息将会从**tb_manager**表中删除。 我们可以在**tb_staff**表上创建一个触发器：

```
dmSQL> CREATE TRIGGER tr_staff_del AFTER DELETE ON tb_staff
                                FOR EACH ROW
                                (DELETE FROM tb_manager WHERE Id =
old.Id );
```


➲ **示例2**

如果您想在触发器行为中添加另一个条件，如“只有当此员工是项目经理时，才能够从**tb_manager**表中删除此记录”。我们可以使用如下命令来替换**tb_staff**表中的触发器行为，并且添加触发条件：

```
dmSQL> ALTER TRIGGER tr_staff_del REPLACE WITH AFTER DELETE ON tb_staff
                              FOR EACH ROW
                              (DELETE FROM tb_manager
                                WHERE Id = old.Id
                                AND title = 'Project Mananger');
```

除此之外，触发器也可以先删除，再重新创建。

10.5 删除触发器

您可以通过DROP TRIGGER语句从数据库中删除触发器。

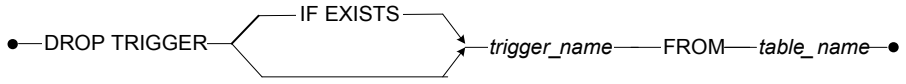


图10-5 DROP TRIGGER 语法

删除触发

删除表将会引起关联此表的触发器一并被删除。如果表结构被更改，当触发器被执行时，DBMaster会根据新定义的表来执行触发器。如果触发事件或触发行为中的指定字段被删除，那么触发器的执行将会失败，并且触发语句也会失败。唯一的解决方法是删除触发器或根据新的表计划来更改触发器定义。为了删除触发器，您可以指定将要删除的触发器名和与之相关联的表。

➲ **示例1**

使用DROP TRIGGER指令从**myTable**表中删除触发器**myTrigger**：

```
dmSQL> DROP TRIGGER myTrigger FROM myTable;
```

➲ **示例2**

使用DROP TRIGGER IF EXISTS指令从**myTable**表中删除触发器**myTrigger**：

```
dmSQL> DROP TRIGGER IF EXISTS myTrigger FROM myTable;
```

➲ **示例3**

为表**tb_staff**创建名为**tr_staff_upd**的触发器：

```
dmSQL> CREATE TRIGGER tr_staff_upd AFTER UPDATE ON tb_staff
                          FOR EACH ROW
                          (DELETE FROM tb_salary WHERE id = old.id);
```

如果表**salary**中的字段**id**被删除或字段类型被更改，当执行触发语句（在**tb_staff** 表上更新）时，会引起数据库管理系统（DBMS）尝试执行**tr_staff_upd**触发器，这样将会发生执行错误。

10.6 使用触发器

触发器有多种用途。

触发器行为中的存储过程

触发器的一个最主要的特征是可以将存储过程作为触发器行为。用EXECUTE PROCEDURE语句来调用存储过程，可以将数据从触发表传递到存储过程中，并且执行此存储过程。

➲ **示例**

在触发器创建中使用EXECUTE PROCEDURE：

```
dmSQL> CREATE TRIGGER tr_sales_update AFTER UPDATE OF price ON tb_Sales
                                    FOR EACH ROW
                                    (EXECUTE PROCEDURE
                                    logPrice(item_no, new.price,
old.price));
```

用户可以将数据传递到自变量表的存储过程中。如果存储过程是行触发，那么用户可以使用旧的和新的关联值，将触发表中的字段值当作参数，传递到所调用的存储过程中。如果存储过程是语句触发，那么用户只能将常数传递到存储过程中。

在一个触发器行为中，您可以在触发表内更新非触发字段，使用或不使用存储过程均可。存储过程中不能包含事务控制语句，像BEGIN、COMMIT、ROLLBACK、SAVEPOINT或DDL语句。

作为触发器行为的存储过程不能包含输出参数或结果集。

触发器行为中的SQL块

触发器还可以将触发SQL块（trigger SQL block）作为触发行为，触发SQL块是数据库临时创建和执行的一组SQL语句集。

触发SQL块的行为和语法类似于匿名SQL块（Anonymous SQL Block），可以让用户在执行触发操作时执行一组SQL语句（batch of SQL），且支持包含变量、语法逻辑、游标等在内的全部SQL语法块。同时，SQL块里面的SQL语句仍然能够使用OLD\NEW参数以及REFERENCING子句。

SQL存储过程包含多个SQL语句，各语句均以“；”结尾，因此dmSQL支持块定界符。块定界符是由a-z、A-Z、@、%组成的包含2-7个字符的字符集，在块定界符中，“；”不是输入结束的标志。在dmSQL中编辑SQL存储过程前需先设定块定界符，否则将会返回错误。有关更多SQL块使用的变量及语法逻辑信息等，请参考12章*匿名存储过程*。

注意　*触发SQL块的复合语句被“**BEGIN**”和“**END**”包围。*

➲ **示例1**

通过触发器行为创建SQL块：

```
dmSQL> set block delimiter @@;
dmSQL> create table tab_t1(c1 int,c2 int);
dmSQL> create table tab_t2(c1 int,c2 int);
dmSQL> insert into  tab_t1 values(111,222);
dmSQL> @@
   2> Create trigger uptrg before update on tab_t1 for each row
   3> begin
   4> insert into tab_t2 values(new.c1,new.c2);
   5> insert into tab_t2 values(old.c1,old.c2);
   6> end;
   7> @@
```

```
dmSQL> update tab_t1 set c1= 1 where c1 = 111;
dmSQL> select * from tab_t1;
dmSQL> select * from tab_t2;
```

➲ **示例2**

假如表字段名包含NEW和OLD，触发SQL块将支持REFERENCING子句。

```
dmSQL> Create table tk_tb1(c1 int,new int,old int);
dmSQL> Create table tk_tb2(c1 int,new int,old int);
dmSQL> set block delimiter @@;
dmSQL> @@
    2> Create trigger uptrg before update on tk_tb1
    3> REFERENCING OLD as pre NEW as post
    4> for each row
    5> begin
    6> insert into tk_tb2 values(pre.c1, pre.new, pre.old);
    7> insert into tk_tb2 values(post.c1, post.new, post.old);
    8> end;
    9> @@
```

触发器的执行顺序

对于Update字段触发器而言，触发器的执行顺序取决于触发事件中所指定的字段号。触发器执行时，会由最小字段号的触发器开始执行，一直到最大字段号的触发器。下例中：a = column 1、b = column 2、c = column 3、d = column 4。

➲ 示例

命令UPDATE **t1** SET b = b + 1, c = c + 1将激活两个触发器。触发器**trig1**中的触发字段比触发器**trig2**小，所以触发器**trig1**会先执行。下例中，假定表**t1**有四个字段，分别为 **a**、**b**、 **c**、**d**。

```
dmSQL> CREATE TRIGGER trig1 AFTER UPDATE OF a,c ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c1=c1+1);


dmSQL> CREATE TRIGGER trig2 AFTER UPDATE OF b,d ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c2=c2+1);
```

安全性与触发器

首先用户必须拥有执行触发事件的权限，否则用户将无法触发该事件。用户无需拥有执行触发器行为的权限，因为触发器行为中所规定的SQL语句，只能在触发器创建者的权限范围内才能执行。一旦触发器创建成功，触发器创建者就拥有执行该触发器行为的权限。也就是说，只要触发器能够创建，任何能执行触发事件的用户都能激活该触发器。

➲ 示例

用户**B**可以更新表**T1**和**T2**，用户**A**可以更新表**T1**但不能更新表**T2**。现在用户**B**在**T1**上创建了一个更新触发器，触发器行为为更新**T2**。当用户**A**更新**T1** 后，触发器行为（**更新T2**）将会执行成功，因为触发器行为是以用户**B**的权限范围来执行的。这项安全性原则简化了程序的执行，并且消除了用户使用多个权限来执行触发器行为的要求。

游标和触发器

针对触发器行为的执行而言，游标动作中的更新/删除命令与单一的更新/删除命令是不同的。如果以WHERE CURRENT OF子句来执行更新和删除，则对每笔更改的数据都会执行一次完整的触发器。

举例来说，如果用游标来更改四笔记录，则每个Before/For each statement、Before/For each Row、After/For each Statement、After/For Each Row触发都将针对每笔记录执行四次。

级联触发

执行一个触发器也会引起其它触发器的执行。您可以使用级联触发来强制资料的完整性。DBMaster最多支持64个级联触发。

➲ **示例**

当您从**tb_customer**表中删除一笔客户记录时，则会触发删除与这笔客户资料相关的**tb_order**表中的订单记录，接着再触发删除与订单相关的**tb_item**表中的记录：

```
dmSQL> CREATE TRIGGER tr_cas1 AFTER DELETE ON tb_customer
                              FOR EACH ROW
                              (DELETE FROM tb_orders WHERE cust_num =
old.cust_num);


dmSQL> CREATE TRIGGER tr_cas2 AFTER DELETE ON tb_orders
                              FOR EACH ROW
                     (DELETE FROM tb_items WHERE order_num =
old.order_num);
```

在DBMaster中，如果用户创建了一个递归触发，在创建触发器时，系统将不会返回出错信息。然而当递归触发执行时，用户将获得一个错误提示信息，告诉用户级联触发受到最大数量限制。

10.7 触发器的启用和屏蔽

当触发器创建时，触发器会处于启动（**enabled** ）模式，这表示当触发事件发生时，会执行触发器行为。

但在某些时候，用户会希望屏蔽触发器：

- 当用户载入大量数据时，暂时屏蔽触发器将会加快载入速度。
- 当关联的触发对象找不到时。

➲ **示例1**

您可以通过以下命令将表**Mytable**的触发器**Mytrigger**屏蔽掉：

```
dmSQL> ALTER TRIGGER Mytrigger ON Mytable DISABLE;
```

➲ **示例2**

您可以通过下列命令启用表**Mytable**的触发器**Mytrigger**：

```
dmSQL> ALTER TRIGGER mytrigger ON mytable ENABLE;
```

简言之，触发器主要有以下两种模式：

- **启用（Enabled）—** 当创建触发器时，触发器是被激活的。当触发事件发生时，就会执行触发器行为。
- **屏蔽（Disabled）—** 即使触发事件发生时，触发器行为也不会被执行。

10.8 创建触发器的权限

想要在表上创建触发器，用户必须是表的拥有者或拥有DBA权限。要使用 CREATE TRIGGER 命令成功创建触发，触发创建者必须拥有参考对象的所有权限。

对DBMaster而言，触发器并无拥有者，它只是和表相关联。表拥有者和具有DBA权限的用户拥有触发器的权限。他们可以创建、删除或更改触发器。

触发行为中的SQL语句是在触发拥有者的授权定义域中执行，而非在执行触发事件的用户授权定义域中执行。

11 存储命令

存储命令是存储在数据库内，并且已经编译好的DML语句。因为存储命令已经事先编译成可执行的形式，所以您能够不断执行相同的SQL语句，而不必一再的编译和最佳化此SQL语句。对于频繁使用的SQL语句是很适合创建成存储命令的。如果使用存储命令去执行经常使用的SQL语句，您将会得到更好的执行效率。您可以将存储命令视为一种只有一个SQL语句而没有其它程序逻辑的存储过程。

11.1 创建存储命令

您可以使用CREATE COMMAND语句来创建存储命令。

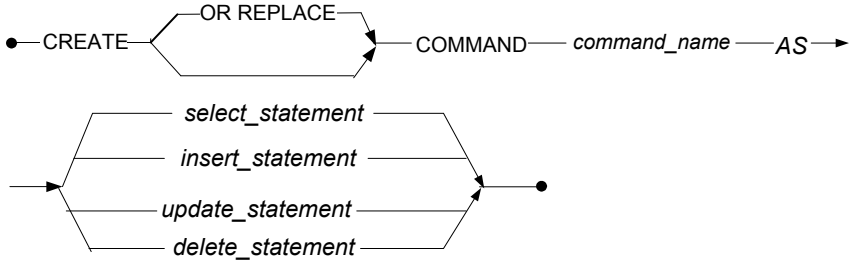


图11-1 CREATE COMMAND 命令的语法

OR REPLACE用于重建已存在的存储命令，即用户可以使用该子句更改已存在存储命令的定义。

您可以将含有输入参数的SQL语句建立成存储命令。当这些存储命令执行时，必须指定这些输入参数的输入值。

➲ **示例1**

有一张表**tb_student**（**id** INT、**score** INT、**name** CHAR(32)），您可以将下列含有输入参数的SQL DML语句建立成存储命令：

```
dmSQL> INSERT INTO tb_student VALUES (1, ?, ?);
```

➲ **示例2**

或者使用CREATE COMMAND：

```
dmSQL> CREATE COMMAND sc_student_insert AS INSERT INTO tb_student VALUES
(1, ?, ?);
```

➲ **示例3**

或者使用CREATE OR REPLACE COMMAND：

```
dmSQL> CREATE COMMAND sc_student_insert AS INSERT INTO tb_student VALUES
(1, ?, ?);
```

➲ **示例4**

为其它的DML语句创建存储命令：

```
dmSQL> CREATE COMMAND sc_student_select AS SELECT id,name FROM
tb_student;

dmSQL> CREATE COMMAND sc_student_update AS UPDATE tb_student SET id =
id+1 WHERE score > ?;

dmSQL> CREATE COMMAND sc_student_delete AS DELETE FROM tb_student WHERE
score > ?;
```

创建存储命令后，您可以直接在dmSQL或应用程序中执行它。如果您执行的存储命令中含有输入参数，那么您可以使用参数符号、常数、NULL、DEFAULT或内建函数（不含参数的内建函数）来指定输入参数的值。当执行含有输入参数的存储命令时，输入参数值的数目应该与创建存储命令的SQL语句的输入参数数目相等。

11.2 执行存储命令

您可以使用EXECUTE COMMAND语句来执行存储命令。

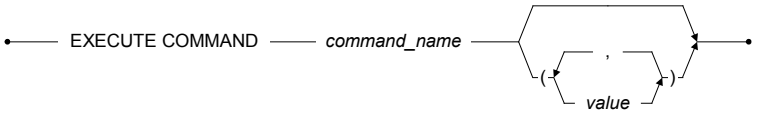


图11-2 EXECUTE COMMAND命令的语法

- **示例1**

```
dmSQL> EXECUTE COMMAND sc_student_insert (200, 'john');
```

- **示例2**

```
dmSQL> EXECUTE COMMAND sc_student_insert (DEFAULT, ?);
```

- **示例3**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, NULL);
```

- **示例4**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, ?);
```

您可以删除不再需要的存储命令。

11.3 重建存储命令

您可以使用REBUILD COMMAND语句重建存储命令。

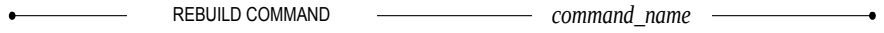


图11-3 REBUILD COMMAND命令的语法

➲ **示例**

```
dmSQL> REBUILD COMMAND sc_student_insert;
```

11.4 删除存储命令

您可以使用DROP COMMAND语句来删除存储命令。

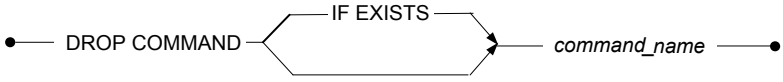


图11-4 DROP COMMAND 命令的语法

➲ **示例1**

```
dmSQL> DROP COMMAND sc_student_insert;
```

➲ **示例2**

```
dmSQL> DROP COMMAND IF EXISTS sc_student_insert;
```

11.5 存储命令的安全性管理

在DBMaster数据库中，存储命令和其它数据库的对象相同，都具备安全性管理。在创建或使用存储命令时，用户必须考虑存储命令的对象权限和安全性管理。

只有具备RESOURCE权限的用户才能够创建存储命令。如果用户拥有执行 SQL DML语句的权限，那么此用户可以将SQL DML语句创建成存储命令。

➲ **示例**

拥有resource权限的用户**joe**可以通过以下语法创建存储命令**sc_CheckDate**：

```
dmSQL> CREATE COMMAND sc_CheckDate AS SELECT FirstName, LastName,
Hiredate FROM SYSADM.tb_staff WHERE HireDate > '1995-01-01';
```

表**tb_staff**的拥有者是SYSADM，所以系统管理员必须在用户**joe**创建存储命令之前，给用户**joe**授予**SYSADM.tb_staff**表的**select**权限。

只有拥有执行权限的用户才能够执行此存储命令。为了能够使存储命令被其他用户使用，用户可以给存储命令授予执行权限。然而，只有拥有权限的用户（DBA、SYSDBA、SYSADM、存储命令的创建者或授予权限的其他用户）才能够授予或取消存储命令的执行权限。

DBA拥有数据库中所有存储命令的执行权限。存储命令的拥有者可以执行、授予和取消权限，但是只有存储命令的拥有者才可以将它删除。

授予执行权限

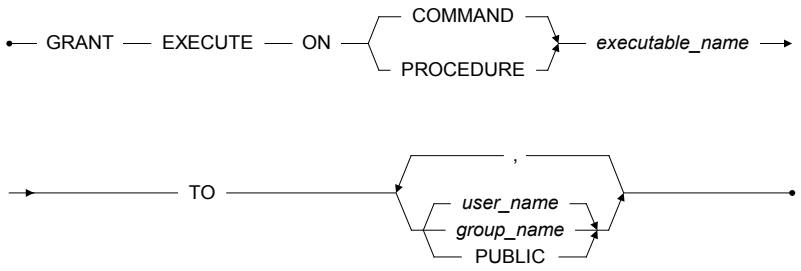


图11-5 GRANT EXECUTE权限的语法

➲ **示例**

通过GRANT EXECUTE ON COMMAND命令授予用户**John**存储命令**sc_student_insert**的执行（EXECUTE）权限：

```
dmSQL> GRANT EXECUTE ON COMMAND sc_student_insert TO John;
```

取消执行权限

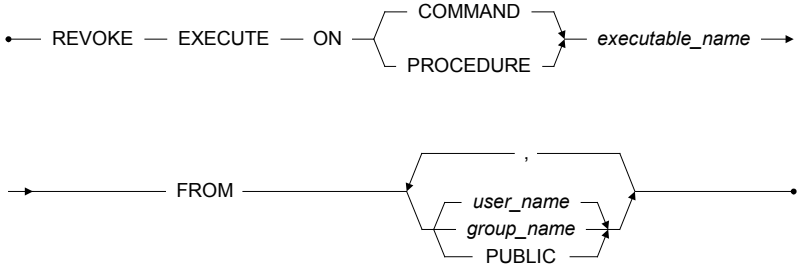


图11-6 REVOKE EXECUTE权限的语法

➲ **示例**

下例说明了如何取消用户**John**对存储命令**sc_student_insert**的执行权限：

```
dmSQL> REVOKE EXECUTE ON COMMAND sc_student_insert FROM John;
```

11.6 存储命令的生存周期

如果存储命令的相关表被删除或更改，那么此存储命令将会视为无效。如果先前有程序参考了旧的字段信息，那么在程序执行时，将会引起不可预期的错误。

使用存储命令的好处是，当重复执行SQL命令时，可以提高执行的性能。DBMaster也会更新执行计划，例如：UPDATE STATISTICS。当执行UPDATE STATISTICS命令时，存储命令的所有执行计划都将被更新，以达到更好的执行性能。

11.7 存储命令的相关信息

用户可以从系统表SYSCMDINFO中获得关于存储命令的信息。下表列出了SYSCMDINFO表的字段和相应的值：

字段名	值	注释
MODULENAME	存储命令隶属的模块名	此字段用于 ESQL 应用程序或存储过程。如果它是一个纯粹的存储命令，那么此存储命令将被忽略。
CMDNAME	存储命令的名称	无
CMDOWNER	存储命令的所有者	无
STATEMENT	生成存储命令的SQL语句	无
NUM_PARM	存储命令的参数数目	无
STATUS	0、1或2	**0** -无效的存储命令，无法被执行。 **1** –正确的存储命令，可以被执行。 **2** –存储命令将被返回，经过数据库内部重新封装后，它将被执行。
REBTIME	存储命令的重建时间	无
CMDPLAN	导出存储命令执行计划字符串	无

表11-1 SYSCMDINFO表的详细资料

用户可以通过以下SQL语句获得存储命令的相关信息：

```
dmSQL> SELECT * FROM SYSCMDINFO;
```

12 存储过程

存储过程是一种特殊的用户自定义函数，DBMaster的存储过程支持三种类型的语言：ESQL/C、Java和SQL。存储过程一旦创建，就以一种可执行的格式存储在数据库中。如此一来，数据库引擎可避免重复SQL命令的编译和优化，从而提高了重复执行命令的执行效率。存储过程作为一个可执行命令用在以下环境：交互式SQL命令、应用程序、触发器或其它存储过程。

利用存储过程可以实现很多目标，例如：提高数据库的执行性能、简化应用程序的代码、限制和监控数据库的访问等。

因为存储过程在数据库中是作为一个可执行对象存储的，所以对每一个运行在数据库上的应用程序都是可用的。为了减少应用程序的开发时间，多个应用程序可使用同一个存储过程。

12.1 通过ESQL创建存储过程

ESQL存储过程是一个ESQL/C程序，它可以执行任何C程序所允许的函数，包括调用其它的C函数和系统调用。因此系统必须安装一个C编译器。

一个ESQL/C编写的存储过程包括CREATE PROCEDURE语句，必要的变量声明部分以及代码区。如果您的程序不需要使用主变量，那么变量声明部分可以省略。

➲ **示例**

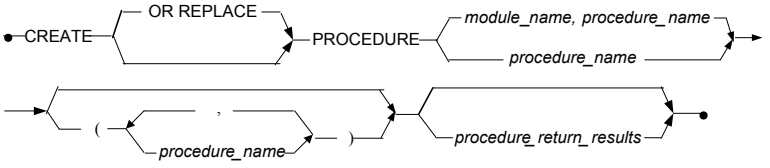
下例创建了一个名为**sp_Aphone**的存储过程，其中带有一个输入参数，一个输出参数以及一个返回值（**status**）：

```
EXEC SQL CREATE PROCEDURE sp_Aphone (CHAR(13) name, CHAR(13) phone
OUTPUT)
                RETURNS STATUS;
{
    EXEC SQL BEGIN CODE SECTION;

         EXEC SQL SELECT PHONE FROM TBL WHERE NAME = :name INTO :phone;

         EXEC SQL RETURNS STATUS SQLCODE;

    EXEC SQL END CODE SECTION;
}
```

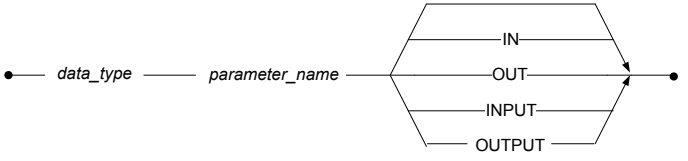
此程序的结构将在稍后章节中详细说明。

创建存储过程的语法

创建存储过程语法以CREATE PROCEDURE语句开头。CREATE PROCEDURE语句的语法如下：



<*procedure_parameters*>子句



<*procedure_return_result* >子句

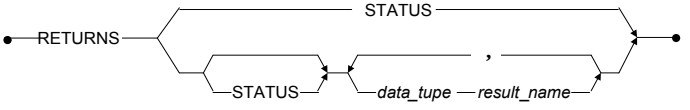


图12-1 CREATE PROCEDURE命令的语法

OR REPLACE用于重建已存在的存储过程，即用户可以使用该子句更改已存在存储过程的定义。

注意 *若替换存储过程失败，原存储过程也会被删除。*

➲ **示例**

下面是CREATE PROCEDURE语句语法的例子：

```
dmSQL> CREATE PROCEDURE sp_example1 (INTEGER n IN) RETURNS STATUS;
dmSQL> CREATE PROCEDURE sp_example2 (INTEGER n1 IN, INTEGER n2 OUTPUT)
       RETURNS CHAR(12) nm;
```

```
dmSQL> CREATE PROCEDURE sp_example3(CHAR(10) par1 OUTPUT, SMALLINT par2)
       RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
dmSQL> CREATE OR REPLACE PROCEDURE sp_example4(CHAR(10) par1 OUTPUT,
SMALLINT par2) RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
```

在一个CREATE PROCEDURE语句中，必须定义存储过程的名称以及I/O参数的类型和名称。

参数的使用

如果存储过程需要参数，那么必须在存储过程后的括号中指出该参数的名称、类型以及属性，并以逗号将各参数分开。参数的属性是指此参数为IN/OUT（或INPUT/OUTPUT），如果没有设定参数的属性，那么默认值为 IN（输入参数），输入参数可以将值传递给存储过程。“通过ESQL创建存储过程”中的示例就包含了一个输入参数**name**。当用户执行存储过程时，必须指定输入参数的值。

输出参数是用来获得存储过程执行后的单一输出结果而非结果集。如示例中存储过程**sp_Aphone**的输出参数**phone**。当用户接收结果时，必须指定一个缓冲区给输出参数。当存储过程执行成功后，输入参数name所对应的电话号码将会存储于输出参数phone所对应的缓冲区中。除了输出参数外，用户也可以使用存储过程从数据库中获得一结果集，但是必须指定一个结果序列，如果存储过程没有返回选择的结果，那么就不需要指定结果序列。关键字RETURNS用于启动一个结果序列，结果序列包含了所要传回的结果集，是一列名称和类型。关键字STATUS用整数类型返回存储过程执行后的状态值，用来表示存储过程执行的成功与否。

➲ **示例**

执行一个带有输入参数和结果集的存储过程：

```
EXEC SQL CREATE PROCEDURE sp_Select (FLOAT ifl) RETURNS STATUS,
                                                  FLOAT fl,
                                                 DOUBLE db;
{
```

```
    EXEC SQL BEGIN CODE SECTION;

    EXEC SQL RETURNS STATUS SQLCODE;

    EXEC SQL RETURNS SELECT fl, db FROM t8 WHERE fl < :ifl
into :fl, :db;

    EXEC SQL END CODE SECTION;

}
```

对于存储过程的输入参数和输出参数，目前DBMaster支持以下几种数据类型：INTEGER、SMALLINT、CHAR()、VARCHAR、DATE、TIME、TIMESTAMP、FLOAT、DOUBLE、REAL。

返回查询结果集

在存储过程中，可以通过主变量来返回一个查询结果集。存储过程是通过RETURNS关键字来指示预处理程序生成一个C代码的主变量，关键字RETURNS位于查询结果集的select语句之前。

➲ **示例**

下例中有两个RETURNS关键字，一个在创建存储过程的语句中，另一个在存储过程的select语句之前。如果存储过程返回结果集，则在输出参数前一定要用RETURNS关键字进行声明，同时存储过程中的select语句也要加上RETURNS关键字：

```
EXEC SQL CREATE PROCEDURE  sp_Allphone RETURNS CHAR(12) name, CHAR(12)
phone;

{

    EXEC SQL BEGIN CODE SECTION;

        EXEC SQL RETURNS SELECT NAME, PHONE FROM TBL INTO :name, :phone;

    EXEC SQL END CODE SECTION;

}
```

模块名

当用户创建存储过程时，DBMaster会将当前的用户名和存储过程名作为默认的动态链接库名。用户可以仅通过存储过程名来调用或删除自己的存储过程。每一个用户都可以通过完整的存储过程名来调用其他用户的存储过程：*owner.procedure_name* 。

用户也可以通过CREATE PROCEDURE语法来定义一个模块名，用来更改默认的动态链接库名。如果模块名已通过CREATE PROCEDURE语法指定，那么用户必须使用完整的存储过程名来调用、删除 *module_name.owner.procedure_name*。尽管是用户自己创建的此存储过程。

变量声明

存储过程主变量的声明方式和ESQL/C相同。即主变量的声明必须位于存储过程的代码区之前。C变量声明也必须位于存储过程的代码区之前，但可位于主变量声明之前或之后。

程序代码区

在存储过程中，除了变量声明外，所有其他的语句都位于程序代码区内。请注意：在代码区前，若有任何非变量声明的语句，可能会造成编译时的错误或得到错误的执行结果。此外，位于CODE SECTION之后的所有语句将不会被执行。

存储过程的参数配置

当存储过程创建时，一个相应的动态链接库也就随之创建，并存储在服务器上。默认状态下，动态链接库存放于DBMaster数据库的工作目录下。数据库管理员可以通过关键字**DB_SPDir**来为存储过程的动态链接库文件指定不同的存放路径。

当存储过程被创建或执行时，客户端的用户可以通过关键字**DB_SPLog**来指定一目录，用于接受从服务器返回的错误信息文件和跟踪文件。

➲ **示例1**

在**dmconfig.ini**配置文件中，将存储过程的动态链接文件的默认路径设置为：***/usr1/DBMaster/data/SP***：

```
DB_SPDir=/usr1/DBMaster/data/SP
```

➲ **示例2**

在**dmconfig.ini**配置文件中，将存储过程日志文件的路径设置为***c:\usr\jerry\data\SP***：

```
DB_SPLog=c:\usr\jerry\data\SP
```

通过文件的方式创建存储过程

首先，在文件中写好一个存储过程，然后通过DBMaster的工具：dmSQL或JDBATool，将新的存储过程导入到数据库中。

图12-2 CREATE PROCEDURE FROM <file_name>语句的语法

➲ **示例**

通过多个文件创建存储过程：

```
dmSQL> CREATE PROCEDURE FROM 'proc1.ec';

dmSQL> CREATE PROCEDURE FROM '.\esql\sp\proc2.ec';

dmSQL> CREATE PROCEDURE FROM 'c:\users\jerry\sp\proc3.ec';
```

上例说明了如何使用dmSQL来创建存储过程。

➲ **使用JDBA工具创建存储过程：**

1. 点击目录树中的**存储过程**节点。
2. 点击**创建**按钮，会出现**创建存储过程**向导的**简介**窗口。
3. 选择**导入**按钮，导入存储过程。
4. 当点击**导入**按钮时，会打开相应的**打开**窗口。文件可以来自任何源，包括服务器的其它数据库的SPDIR目录中或网络驱动器上。在

文件名区域中输入所需的文件名或通过浏览目录的方法找到正确的路径。

注意 *导入的文件必须是ASCII格式的包含C++代码的文件。*

5. 选择**打开**按钮，打开选择的文件。
6. 如果导入的文件是ASCII文本，那么将出现**创建存储过程**窗口。选择**另存为**按钮，将存储过程存放到另一个目录下，或者选择**确定**按钮，在数据库中编译并存储该存储过程。

注意 *创建存储过程中，出现的任何错误都将显示在下面的窗口中。*

执行存储过程

您可以通过dmSQL、C程序（ODBC或ESQL）、其它存储过程或触发器来调用一个存储过程。

DMSQL

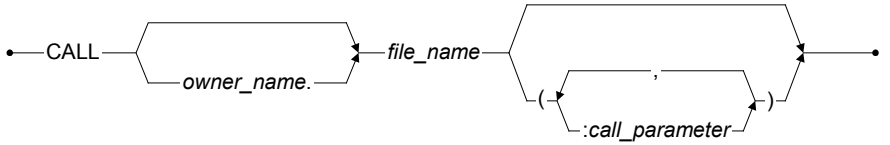


图12-3 dmSQL中CALL语句的语法

➲ **示例1**

在dmSQL中执行一个存储过程：

```
dmSQL> CALL sp_example1 (3);                  // execute procedure
sp_example1

dmSQL> CALL SYSADM. sp_example2 (5, ?);       // execute SYSADM's
procedure sp_example2

dmSQL/Val> 100;                                // input the value of
parameter

dmSQL> ? = CALL SYSADM. sp_example2 (5, 100); // execute procedure p2
and get
```

```
                                                    // returned status
```

➲ **示例2**

如果存储过程返回一个结果集，那么dmSQL会自动将输出参数返回的结果集显示在屏幕上，就像您通过dmSQL执行了一条SELECT语句一样：

```
dmSQL> CALL sp_Aphone('jeff');
     STATUS           PHONE
==============   ==============
              0   408-255-2689

dmSQL> CALL sp_Allphone;
      NAME            PHONE
==============   ==============
Jerry            02-775-8615
Jeff             408-255-2689
```

ESQL

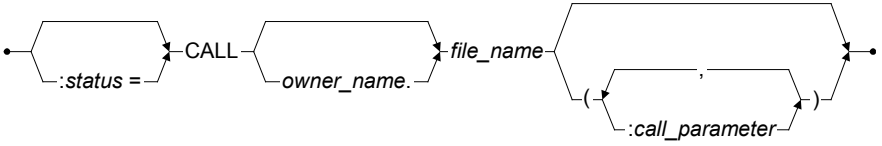


图12-4 ESQL中CALL语句的语法

➲ **示例**

在ESQL中执行以下存储过程：

```
EXEC SQL :s = CALL sp_example1 (3);
EXEC SQL CALL SYSADM. sp_example2 (5, :n2) INTO :nm;
EXEC SQL :s = CALL jack. sp_example3 (:par1, 7) INTO :ret1, :ret2;
```

用ESQL程序执行存储过程的语法和dmSQL类似，设置主变量来接收返回的状态、输出参数和结果集。

执行嵌套存储过程

用户可以在一个ESQL/C存储过程中调用另一个存储过程，执行的方式与在ESQL/C程序中的相同。唯一不同的是，一般的ESQL程序不能使用RETURNS关键字，但是存储过程可以使用此关键字来调用另一个存储过程。

假设存储过程**sp_Allphone**返回了一个包含多笔记录的结果集，一般的ESQL程序需要使用一个游标来得到存储过程的结果集，另一个存储过程**sp_another**也可以使用同样的方法来得到结果集并且检查数据，此外也可以将调用存储过程的结果集当作目前存储过程的结果集来返回。

➲ **示例**

通过下列语句在**sp_another**中调用存储过程**sp_Allphone**：

```
EXEC SQL RETURNS CALL sp_Allphone INTO :oName, :oPhone;
```

当存储过程返回另一存储过程的结果集时，调用程序必须和被调用者有完全相同的结果集字段，或者前n个结果字段必须相同。

在ODBC程序中执行存储过程

您也可以在ODBC中调用存储过程，但用户必须绑定存储过程的参数，并且使用返回的结果集字段。在ODBC程序中，您也可以绑定存储过程结果集的部分字段。存储过程执行后，输出参数的值就会返回到主变量中，如SELECT命令，就用读取函数（fetch）得到结果集。

➲ **示例1**

存储过程**gaojing3151**的声明：

```
dmSQL> CREATE PROCEDURE sp_proc1(CHAR(12) p1, CHAR(12) p2 OUTPUT)
RETURNS INTEGER r1;

{

EXEC SQL BEGIN CODE SECTION;
```

```
EXEC SQL SELECT c2 FROM t1 WHERE c1 = :p1 INTO :p2;
EXEC SQL RETURNS SELECT c1 FROM t2 INTO :r1;
EXEC SQL END CODE SECTION;
}
```

➲ **示例2**

在标准ODBC程序中，调用存储过程**sp_proc1**：

```
SQLPrepare(cmdp,(UCHAR*)"call sp_proc1(?, ?)", SQL_NTS);

strcpy(bpname, "12345");

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                    20, 0, &p1, 20, NULL);
SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                    20, 0, &p2, 20, NULL);

SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp);                                   /* get p2
*/

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND)      /* fetch result set
*/
```

跟踪存储过程的执行

DBMaster提供了一种跟踪功能，可以帮助用户跟踪存储过程的执行，达到对存储过程中调试的目的。

➲ **示例**

使用TRACE命令：

```
EXEC SQL TRACE ON;                          // Start TRACE
EXEC SQL SELECT c1 FROM t1 INTO :var1;
EXEC SQL TRACE ("var1 = %d\n", var1);       // TRACE the value of var1
EXEC SQL TRACE OFF;                         // END OF TRACE
```

您可以开启并使用TRACE功能来设置跟踪变量和输出结果。当存储过程执行后，所有跟踪信息都将写入**_spusr.log**文件，此文件位于客户机的**dmconfig.ini**中**DB_SPLog**定义的目录下。

12.2 使用JAVA创建存储过程

使用Java语言创建存储过程是有一定道理的。在Java语言盛行的今天，开发人员对Java语言的运用程度会比ESQL更精通。DBMaster支持Java存储过程，为Java程序员利用他们更精通的语言来编写存储过程提供了方便。针对有经验的ESQL开发人员，可以利用Java语言的优势来扩展数据库应用功能，也可以重复利用现有的代码以提高效率。

DBMaster支持一种java方法作为java存储过程，DBMaster用**jdbc:default:connection**来替换DriverManager.getConnection(url, …) URL变量。您可以使用所有java类来实现一种java方法作为一个java存储过程，包括所有JDBC类。DBMaster对注册.Jar文件，创建、执行和删除Java存储过程都定义了新语法。

CREATE JAVA PROCEDURE 语句的语法如下：

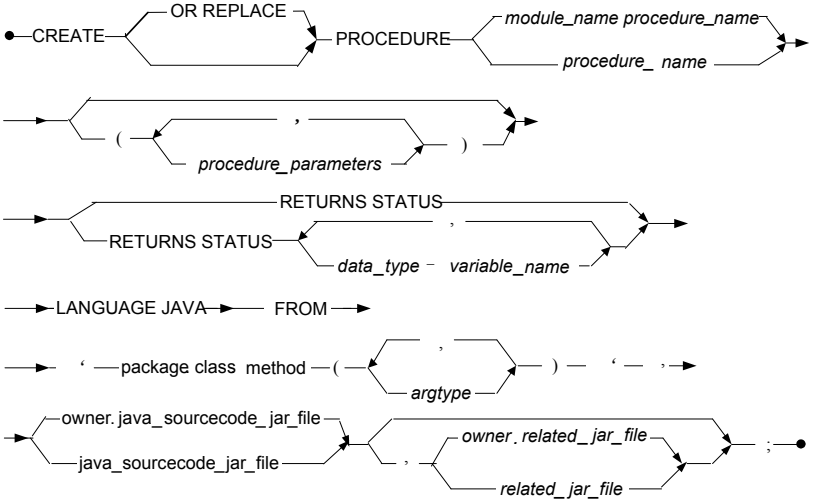


图 12-5 CREATE JAVA STORED PROCEDURE命令的语法

➲ **示例1**

DBMaster支持*/home/usr/mary/sp/aa.jar*目录下的java方法xx.yy.AA(String)，但是*/home/usr/john/sp/bb.jar*仍然需要运行方法AA(String)：

例如：在数据库中注册aa.jar和bb.jar文件。首先，您必须将用户sysadm的*/home/usr/mary/sp*目录下的aa.jar文件移至*/home/sysadm/spdir/jar/SYSADM/*目录中。

DB_SPDir在配置文件**dmconfig.ini**中可定义成“*/home/sysadm/spdir*”，所以用户在将jar文件添加至数据库之前，必须先将jar文件移至*/DB_SPDir/jar/uppercase_username/*目录下。

➲ **示例2**

假设在aa.jar文件中，您拥有一个java方法xx.yy.AA(String)，但是您仍需要bb.jar文件运行方法AA(String)：

在数据库中注册aa.jar和bb.jar文件。

```
dmSQL> ADD JARFILE xaa aa.jar;

dmSQL> ADD JARFILE xbb bb.jar;
```

注意 *在执行“add jarfile”命令之前，用户必须手动将物理jar文件（aa.jar和 bb.jar）移至**DB_SPDir**/jar/uppercase_username/目录中。**DB_SPDir**关键字用于定义DBMaster存储过程路径。*

通过java方法AA(String)创建一个java存储过程**JSP_AA(char(10) par1)**。

```
dmSQL> CREATE PROCEDURE JSP_AA (char(10) par1) RETURNS STATUS LANGUAGE
JAVA FROM xx.yy.AA(String), xaa, xbb;
```

执行java存储过程**JSP_AA**。

```
dmSQL> EXECUTE PROCEDURE JSP_AA ('aaaaaa');

dmSQL> CALL JSP_AA ('bbb');
```

删除java存储过程**JSP_AA**。

```
dmSQL> DROP PROCEDURE JSP_AA;
```

从数据库中注销**aa.jar**和**bb.jar**文件。

```
dmSQL> REMOVE JARFILE xaa;
dmSQL> REMOVE JARFILE xbb;
```

➲ **JAVA中相关的Create Procedure语法：**

向数据库中注册一个jar文件：

```
dmSQL> ADD JARFILE logical_file_name physical_jarfile_name;
```

从数据库中注销一个jar文件：

```
dmSQL> REMOVE JARFILE logical_file_name;
```

使用CREATE PROCEDURE命令创建一个java存储过程：

```
dmSQL> CREATE  [OR REPLACE] PROCEDURE procedure-name
      [(procedure-parameter [, procedure-parameter ...])]
      {
        [RETURNS STATUS]
        | [RETURNS [STATUS,] procedure-result [,procedure-result ...]]
         }
 LANGUAGE JAVA FROM  'package.class.method([' argtype[,argtype…]' ])' ,
[owner.]java-sourcecode-jar-file [, owner.related-jar-file];
```

OR REPLACE用于重建已存在的存储过程，即用户可以使用该子句更改已存在存储过程的定义。

执行一个java存储过程：

```
dmSQL> EXECUTE PROCEDURE [owner.]procedure-name;
dmSQL> EXECUTE PROC [owner.]procedure-name;
[? =] CALL  [owner.]procedure-name  [(procedure-parameter-value [,
procedure-parameter-value ...])]
```

删除一个java存储过程：

```
dmSQL> DROP PROCEDURE [owner.]procedure-name;
```

载入/载出一个java存储过程：

```
dmSQL> UNLOAD PROCEDURE/PORC FROM [owner_patt.]proc_patt TO
unload_filename;

dmSQL> LOAD PROCEDURE/PORC FROM unload_filename;
```

载入/载出一个jar文件：

```
dmSQL> UNLOAD JARFILE FROM [owner_patt.]jarfile_patt TO unload_filename;

dmSQL> LOAD JARFILE FROM unload_filename;
```

注意 *在载入jar文件之前，用户必须将该 物理jar文件手动移至新目录 DB_SPDir/jar/uppercase_username/下。*

执行Java存储过程

通过以下示例解释Java存储过程的使用。

➲ **示例1**（输入参数）：

通过Java存储过程向表**tb_staff**中插入一条记录。

1. 写一个java方法addEmployee(int,String) 向表tb_staff中插入一条记录，随后编译该java方法并将class打包至AA.Jar文件中。

```
public static void addEmployee(int empid, String name)

{
```

```
        Connection conn =
DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement pstmt = conn.prepareStatement("insert into
tb_staff values(?,?)");

        pstmt.setInt(1, empid);

        pstmt.setString(2, name);
```

```
pstmt.execute();

}
```

2. 为Java方法addEmployee(int,String)创建一个Java存储过程：

执行以下SQL语句添加一个新的jar文件：

```
dmSQL> ADD JARFILE logical_AA AA.jar;
```

执行以下任何一个SQL语句创建一个java存储过程：

```
dmSQL> CREATE PROCEDURE JSP_addEmp (int id, char(10) name) RETURNS
STATUS LANGUAGE JAVA FROM 'xx.yy.addEmployee(int,String)', logical_AA;
```

或：

```
dmSQL> CREATE OR REPLACE PROCEDURE JSP_addEmp (int id, char(10) name)
RETURNS STATUS LANGUAGE JAVA FROM 'xx.yy.addEmployee(int,String)',
logical_AA;
```

3. 运行Java存储过程。

执行以下SQL语句运行java存储过程：

```
dmSQL> EXECUTE PROCEDURE JSP_addEmp(1234, 'jeff');
```

➲ **示例2**（输出参数)：

使用Java存储过程从**tb_staff**表中获得一个含有谓词（empid）的员工名称。

1. 写一个Java方法从表**tb_staff**中获得一个含有谓词（empid）的员工名称，随后编译该Java方法，并将class打包至BB.jar文件中。

```
public static void oneEmployee(int id, byte[] name)

{

            Connection conn =
DriverManager.getConnection("jdbc:default:connection");

PreparedStatement pstmt = conn.prepareStatement("select name from
tb_staff where id = ?”);

         pstmt.setInt(1, id);

         ResultSet rs = pstmt.executeQuery();

         Rs.next();

         String empName = rs.getString(1);

         Name = empName.getBytes();
```

```
}
```

2. 为Java方法oneEmployee(int,String)创建一个Java存储过程。

执行以下SQL语句添加一个新的jar文件：

```
dmSQL> ADD JARFILE logical_BB BB.jar;
```

执行以下SQL语句创建java存储过程：

```
dmSQL> CREATE PROCEDURE JSP_oneEmp (int id, char(10) name OUTPUT)
RETURNS STATUS LANGUAGE JAVA FROM 'xx.yy.oneEmployee(int,byte[])',
logical_BB;
```

3. 运行Java存储过程。

执行以下SQL语句来运行Java存储过程：

```
dmSQL> EXECUTE PROCEDURE JSP_oneEmp(1234, ?);
```

➲ **示例3**（结果集）：

通过Java存储过程从表**tb_staff**中选择一个结果集。

1. 写一个Java方法rsEmployee()从**tb_staff**表中获得一个员工名称，随后编译该Java方法，并将class 打包至CC.jar文件中。

```
public static ResultSet rsEmployee()

{

        Connection conn =
DriverManager.getConnection("jdbc:default:connection");

Statement stmt = conn.createStatement();

        ResultSet rs = stmt.executeQuery("select id, name from
tb_staff");

Return rs;

}
```

2. 为Java方法rsEmployee(int,String)创建一个Java存储过程。

执行以下的SQL语句添加一个新的jar文件：

```
dmSQL> ADD JARFILE logical_CC CC.jar;
```

执行以下SQL语句创建java存储过程：

```
dmSQL> CREATE PROCEDURE JSP_rsEmp RETURNS STATUS, int outId, char(10)
outName  LANGUAGE JAVA FROM 'xx.yy.rsEmployee()', logical_CC;
```

3. 运行Java存储过程。

执行以下SQL语句来运行Java存储过程：

```
dmSQL> EXECUTE PROCEDURE JSP_rsEmp();
```

4. 使用普通的读取或可扩展的读取来导出结果集。

输入/输出参数

DBMaster的Java存储过程输入/输出参数支持以下数据类型：

BINARY	VARCHAR
CHAR	FLOAT
REAL	DOUBLE
SMALLINT	INTEGER
TIMESTAMP	DATE
TIME	DECIMAL

DBMaster支持以下7种基础的Java类型和Array：

JAVA类型	ARRAY
byte	byte [ ]
short	short [ ]
int	int [ ]
long	long [ ]
float	float [ ]
double	double [ ]
char	char [ ]

同时支持以下class：

JAVA CLASS	ARRAY
Byte	Byte[ ]
Short	Short[ ]
Integer	Interger[ ]
Long	Long[ ]
Float	Float[ ]
Double	Double[ ]
Character	Character[ ]
String	String[ ]

12.3 SQL存储过程

通过SQL语句创建存储过程，和通过ESQL、JAVA语言创建存储过程相比，不失为一种更好的方法。

SQL存储过程是一种只通过SQL语句来逻辑实现的存储过程，一个SQL存储过程是存储在服务器端的一组SQL语句集，一旦执行SQL存储过程，客户端就不需要再保存单独的语句，而使用SQL存储过程来代替。SQL存储过程包括永久存储过程、临时存储过程和匿名存储过程。有关SQL存储过程的相关信息，请参考*SQL存储过程用户手册*。

架构

SQL存储过程的核心是一个复合语句，该复合语句被BEGIN和END包围，下面是一个SQL存储过程语句的架构。

```
BEGIN                              #语句块头

Variable declarations

Condition declarations

Cursor declarations

Condition handler declarations

Assignment, flow of control, SQL statements and other compound
statements

END;                           #语句块尾
```

这个例子显示了SQL存储过程如何通过一个或多个组件声明和语句来组成一个块，块允许在单个SQL存储过程中嵌套。组件声明的选项包括变量、条件和处理程序。然而，这些选项必须在流量控制、SQL和其它复合语句之前分配。请注意游标声明可能出现在块的任何地方。

创建SQL存储过程语法

通过文件创建存储过程

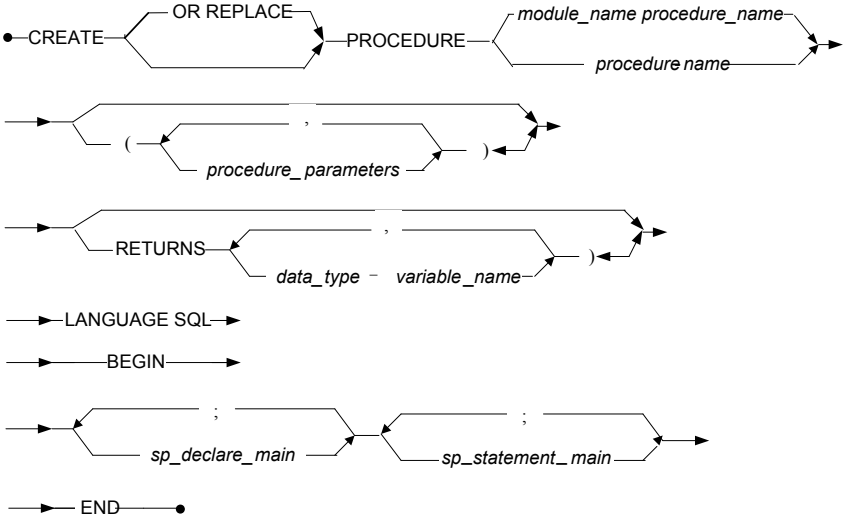


图12-6 创建SQL存储过程语法

OR REPLACE用于重建已存在的存储过程，即用户可以使用该子句更改已存在存储过程的定义。

注意 *不支持在存储过程中执行CREATE OR REPLACE COMMAND和 CREATE OR REPLACE PROCEDURE 命令。*

注意 *当AUTOCOMMIT设置为OFF时，不支持CREATE OR REPLACE PROCEDURE命令。*

SQL存储过程必须通过外部文件（*.sp）来创建，使用dmSQL命令行工具调用外部文件而创建SQL存储过程的示例如下：

➲ **示例1**

通过外部文件创建一个SQL存储过程：

```
dmSQL> CREATE PROCEDURE FROM 'CRETB.SP';
```

```
dmSQL> CREATE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

➲ **示例2**

通过外部文件创建或替换一个SQL存储过程：

```
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

通过脚本创建存储过程

DBMaster支持用户通过文件和dmSQL创建SQL存储过程。用户可调用或删除自己的SQL存储过程，并可以执行已被授予权限的存储过程。详情请参考*12.3章SQL存储过程*。

SQL存储过程包含多个SQL语句，各语句均以“；”结尾，因此dmSQL支持块定界符。块定界符是由a-z、A-Z、@、%组成的包含2-7个字符的字符集，在块定界符中，“；”不是输入结束的标志。在dmSQL中编辑SQL存储过程前需先设定块定界符，否则将会返回错误。

➲ **示例**

通过脚本创建一个SQL存储过程：

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@
    2> CREATE PROCEDURE sp_in_script2
    3> LANGUAGE SQL
    4> BEGIN
    5> INSERT INTO t1 VALUES(1);
    6> END;
    7> @@
dmSQL> SET BLOCK DELIMITER;
```

使用参数

SQL值通过参数被传递进\出SQL存储过程。

当执行逻辑对特殊输入或输入标值集有条件限定时，或当您需要返回一个或多个输出标值并且不希望返回一个结果集时，SQL存储过程中的参数将是十分有用的。

变量声明

SQL存储过程中支持的本地变量允许您分配并返回支持SQL存储过程逻辑的SQL值。

SQL存储过程中的变量都通过DECLARE语句来定义，DECLARE语句用于在程序中定义本地变量项目：本地变量、条件、句柄和游标。DECLARE只允许出现在BEGIN ... END复合语句内，并且必须以此为起始，先于其它语句。声明要遵循一定的顺序：游标必须在声明句柄之前被声明，变量和条件必须在声明游标或句柄之前被声明。

DBMaster支持两种SQL变量：**连接变量（CV）**和**语句变量（SV）**。这两种变量都用于提高dmSQL命令的扩展性和可移植性。以下章节是对**CV**和**SV**的详细说明。

连接变量 （CV）

*CV*是一种只能在当前连接时定义的连接变量。当前连接的连接变量是一种特殊的连接变量，与其他连接变量不同。也就是说，连接变量只能用于自身连接，不能从其他连接获取或用于其他连接。

对于用户来讲，连接变量是当前连接中的SQL命令的全局变量，可用于dmSQL命令行工具以及SQL存储过程。用户可以在一个语句中指定连接变量的值，其它语句可以参照。这表示您可以将一个值从一条语句传送到另一条语句。一旦数据库断开连接，那么所有的连接变量都将自动释放。CV的数据类型是预先定义的，因此您可以在使用之前对变量进行设置并定义其数据类型。如果您参照的变量未被初始化，则会返回一条错误信息：“ERROR（6334）：[DBMaster]无效的变量名”。

➲ **示例1**

CV可以被声明，且可以在dmSQL命令行中使用。下例中的 **@a**、**@aa**以及**@b**均为连接变量，它们不仅可以进行插入、更新、删除操作，还可以用于**WHERE**子句或UDF函数调用等。

```
dmSQL> DECLARE SET INT @a = 1;

dmSQL> SELECT @b;

ERROR (6344): [DBMaster] 无效的变量名: B name

dmSQL> DECLARE SET INT @aa = NULL;

dmSQL> SELECT @aa;

          @AA

======================

          NULL

dmSQL> DECLARE SET INT @b = 2;

dmSQL> SELECT @a, @b;

   @A          @B

=========== ===========

         1           2


dmSQL> SELECT * FROM t1 WHERE c1=@a;

dmSQL> INSERT INTO t1 VALUES(@b+100);

dmSQL> UPDATE t2 SET c2 = @b+2 WHERE c1 = @a+1;

dmSQL> DELETE FROM t1 WHERE c1=@b;

dmSQL> SELECT SUM(c1) INTO @b FROM t1;
```

➲ **示例2**

CV可以在SQL存储过程中被声明且作为普通变量使用，下例中的**@val2**和**@val3**均为连接变量。

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@ CREATE PROCEDURE tsp2
    2> LANGUAGE SQL
    3> BEGIN
    4> DECLARE SET INT @val2 =100+100;
    5> DECLARE SET INT @val3 =LENGTH('test');
            6> SET @val3 =LENGTH('test');
            7> END; @@
```

语句变量（SV）

*SV*是语句变量。语句变量是一个语句中的声明变量，只能用于指定语句。语句变量在不同的命令中是相互独立的，只可以用于SQL存储过程。

语句变量的定义和SQL存储过程的局部变量类似。一旦SQL语句终止，则语句变量将会自动释放。

➲ **示例**

CV和**SV**之间的差异如下所示：

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> DECLARE SET INT @aa = 100;
dmSQL> CREATE TABLE tab(c1 INT);
dmSQL> @@ CREATE PROCEDURE tsp(OUT c1 INT)
    2> LANGUAGE SQL
    3> BEGIN
    4> DECLARE vals INT DEFAULT 100;  --vals is sv
    5> INSERT INTO tab VALUES(@aa);   --aa is cv
    6> INSERT INTO tab VALUES(vals);  --vals is sv
    7> SET c1 = @aa;                  --c1 is sv
```

```
8> SET @aa = 200;                    --error not declare cv in sqlsp
9> END; @@
```

游标

在SQL存储过程中，游标可用来定义结果集并在单行数据上执行复合逻辑，注意，结果集只是一组数据行。通过同样的方法，SQL存储过程还可以定义结果集并直接将它返回至SQL存储过程的调用者或客户端应用程序。

游标可视为一组数据行中指向一行的指针，在结果集中可以指向任一行，但是在任一指定时刻只能参照一行。

DECLARE CURSOR语句首先定义一个游标，然后紧随的SQL语句（OPEN、FETCH和CLOSE）用于操作该游标。

赋值语句

赋值语句用于对SQL变量或SQL参数来赋值，可以通过SET语句和CURSOR FOR SELECT FROM语句给变量赋值。此外，变量声明时也可以设置默认值。文字、表达式、查询结果、特殊触发器的值都可以分配给变量。变量值可以被分配给SQL存储过程参数，SQL存储过程中的其它变量，可在SQL存储过程语句日常执行的程序中作为参数被引用。

SET变量语句可将值分配给本地变量、输出参数和新的转换变量。它受事务的控制，SET赋值语法接受简单表达式和复杂表达式。

注意 *对于string类型变量赋值, 赋值长度要小于1024个字节。*

简单表达式

简单表达式分为数字数据类型：INTEGER、 BIGINT、SMALLINT、DOUBLE、FLOAT、REAL、DECIMAL；字符数据类型：CHAR、NCHAR、VARCHAR、NVARCHAR；BINARY数据类型和时间数据类型：DATE、TIME、TIMESTAMP。

简单表达式包括‘+’、‘-’、‘*’、‘/’和变量、常量、数值、字符串。简单表达式的执行效率比复杂表达式要高很多，比较适用于多次循环语句，可以极大地提高执行速度。

复杂表达式

复杂表达式不仅包括简单表达式中所包括的赋值，还包括SQL函数，如内置函数和用户自定义函数。

控制流语句

SQL控制语句提供的变量支持和流程控制语句可用于控制语句执行的顺序。像IF和CASE这样的语句用于有条件地执行SQL控制语句块，而其它语句如WHILE和REPEAT，通常用于执行一组重复地语句，直到任务完成。

SQL控制流语句分为以下几类：变量关联语句、条件语句（CASE和IF）、循环语句（FOR、LOOP、WHILE 和REPEAT）、goto语句、返回语句、传输控制语句（ITERATE 和LEAVE）、标签和SQL存储过程复合语句。

返回结果集

游标可用于遍历多个重复的结果集行，在SQL存储过程中，游标也可以将结果集返回到调用程序中。

返回SQL存储过程状态

状态代码用于反映存储过程是否执行成功，用户不能在存储过程中定义状态代码。

状态代码：

-1：执行存储过程失败

0：执行存储过程成功

1：执行存储过程报警

若您想返回存储过程状态，您需在"LANGUAGE SQL"语句前添加"RETURN STATUS"。

➲ **示例**

调用其它SQL存储过程：

```
CREATE PROCEDURE call_test
RETURNS STATUS
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                OPEN cur;
END;
CREATE PROCEDURE CASE_TEST_1(IN inval INT, OUT outval1 INT, OUT outval2
INT)
LANGUAGE SQL
BEGIN
        SET outval1 = 1;
         SET outval2 = 2;
END;
CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        CALL CASE_TEST_1(inval, outval1, outval2);
END;
```

执行SQL存储过程

SQL存储过程可通过CALL语句来执行，您可以使用如JDBA Tool这样的图形用户界面工具或直接通过DBMaster命令行工具dmSQL来执行该语句。

CALL语句用来调用一个存储过程。该语句可以被嵌入到应用程序中，通过动态SQL语句或动态准备来发布。

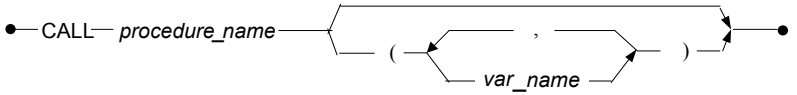


图 12-7 CALL语句语法

匿名存储过程

匿名存储过程（Anonymous Stored Procedures）是数据库临时创建和执行的一组SQL语句集，无需作为数据库对象永久地存储在DBMaster中。它只能临时存在于某单个SQL区块（SQL block）内，并且只能被创建者使用一次。

匿名存储过程属于一种特殊的SQL存储过程，类似于匿名SQL块（Anonymous SQL Block），可以让用户在client端一次执行多条SQL语句（batch of SQL），且支持包含变量、语法逻辑、游标等在内的全部SQL语法块。匿名存储过程不能使用参数和dmSQL命令行工具特有的命令（如：set等）。在dmSQL中编辑匿名SQL块前需先设定块定界符，否则将返回错误。有关匿名存储过程使用的变量和语法逻辑等，请参考12.3章*SQL存储过程*。

注意 *匿名存储过程的复合语句被“**BEGIN**”和“**END**”包围。*

同SQL存储过程相比，匿名存储过程没有名称，不能通过其它数据库对象来进行引用。也就是说，匿名存储过程的执行是即时的。当用户成功创建一个匿名存储过程时，DBMaster会立即执行该存储过程，当其执行完毕后，DBMaster会立即将其删除。匿名存储过程不会将信息保存到系

统表SYSPROCINFO中，也不能永久地存储在DBMaster中以便重复使用。但是匿名存储过程缩短了代码更改和程序执行的时间间隔，从而提高了问题诊断、原型化和代码测试的执行效率，为任务的多次更改和执行提供了便利。

➲ 示例

通过SQL块创建匿名存储过程：

```
dmSQL> set block delimiter @@;
dmSQL>@@
     2> BEGIN //note:this space have ","
     3> CREATE TABLE tab(c1 INT，c2 INT);
     4> INSERT INTO tab values(123,456);
     5>END;
     6>@@
dmSQL >SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
1 rows selected
dmSQL >@@
     2>BEGIN
     3> DECLARE c1 INT;
     4> DECLARE SET INT @a2 = 100;
     5> SET c1 = 200;
     6> INSERT INTO tab VALUES(@a2,c1);
     7>END;
     8>@@
```

```
dmSQL >SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
        100         200
2 rows selected
```

12.4 删除存储过程

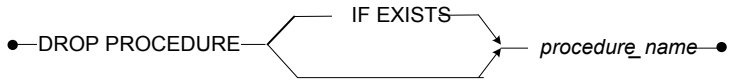


图12-8 DROP PROCEDURE 语句的语法

➲ **示例1**

下例中的第一条命令是删除存储过程**sp_proc1**，第二条命令是删除存储过程**user1.sp_proc2**。

```
dmSQL> DROP PROCEDURE sp_proc1;

dmSQL> DROP PROCEDURE user1.sp_proc2;
```

➲ **示例2**

下例中的第一条命令是删除已存在的存储过程**sp_proc1**，第二条命令是删除已存在的存储过程**user1.sp_proc2**。

```
dmSQL> DROP PROCEDURE IF EXISTS sp_proc1;

dmSQL> DROP PROCEDURE IF EXISTS user1.sp_proc2;
```

12.5 获得存储过程的相关信息

➲ **示例1**

在dmSQL中通过下列语句从系统表**SYSPROCINFO**中获得存储过程的相关信息：

```
dmSQL> SELECT * FROM SYSPROCINFO;
```

➲ **示例2**

在dmSQL中通过下列语句从系统表**SYSPROCPARAM**中获得存储过程的相关信息：

```
dmSQL> SELECT * FROM SYSPROCPARAM;
```

注意 *ODBC函数SQLProcedure()和SQLProcedureColumns()可用于在程序中获得存储过程和参数信息。*

12.6 存储过程的权限设定

只有存储过程的拥有者或DBA，或具有更高权限的用户才能够执行存储过程，其它用户或用户所在的组必须被授予过程的执行权限，才能够调用该存储过程。当然，也只有存储过程的拥有者或DBA，或具有更高权限的用户才能够授予其他用户EXECUTE PROCEDURE的权限。

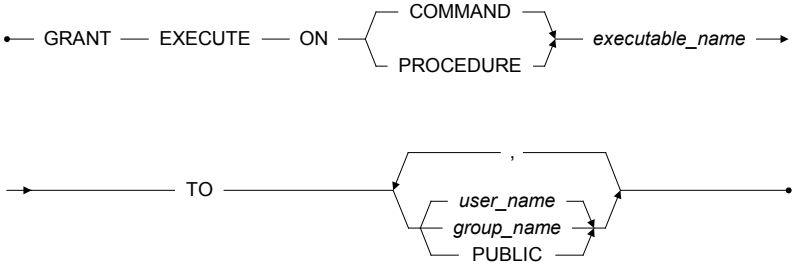


图12-9 GRANT EXECUTE 语句的语法

存储过程的拥有者或DBA，或具有更高权限的用户可以取消其他用户存储过程的执行权限。

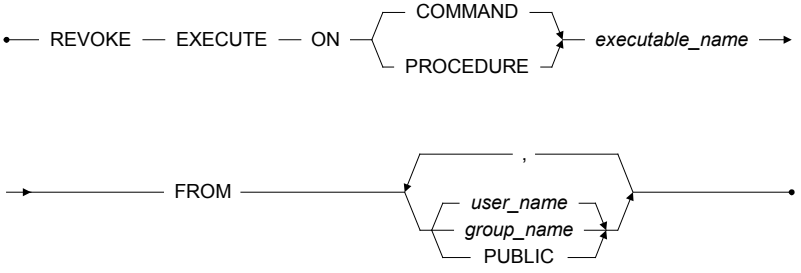


图12-10 REVOKE EXECUTE 语句的语法

➲ **示例1**

用户**user1**创建了一个存储过程 **sp_proc1**，并通过dmSQL将该存储过程的执行权限授给用户**user2**：

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO user2;
```

- **示例2**

  用户**user1**创建了一个存储过程**sp_proc1**，并通过dmSQL将该存储过程的执行权限授给**PUBLIC**：

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO PUBLIC;
```

- **示例3**

  用户**user1**通过dmSQL取消**user2**对存储过程**sp_proc1**的执行权限：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM user2;
```

- **示例4**

  用户**user1**通过dmSQL取消**PUBLIC**对存储过程**sp_proc1**的执行权限：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM PUBLIC;
```

13 计划

组织中有很多任务，单独对它们进行手动处理比较困难。为了帮助用户简化这些管理任务，并为这些复杂的调度需求提供一组丰富的功能，DBMaster提供了先进的作业调度能力。计划表示任务应该何时执行。计划中包含开始时间、结束时间以及时间表，开始时间表示计划开始执行的日期和时间；结束时间表示计划到期的日期和时间；时间表则表示计划何时执行。

计划允许用户控制数据库环境中各种任务的发生时间和发生场所。这些任务都很复杂，处理起来十分耗时，因此使用计划可以帮助用户提高这些任务的管理和计划。此外，通过确保多个常规数据库任务可无需人工干预而自动执行，可以降低运营成本，实现更可靠的程序，减少人为错误，并缩短系统所需时间。

DBMaster使用如下存储过程提供先进的作业调度能力：START_DMSCHSVR、STOP_DMSCHSVR、SCHEDULE_CREATE、SCHEDULE_ALTER、SCHEDULE_DROP、SCHEDULE_RELOAD、SCHEDULE_ENABLE、SCHEDULE_DISABLE、SCHELOG_CLEAN、TASK_CREATE、TASK_ALTER以及TASK_DROP。有关上述存储过程的详细信息，请参考*SQL命令与函数参考手册*。拥有RESOURCE权限的用户可以创建、更改、删除任务和计划，启用或禁止自己创建的计划；拥有DBA以上权限的用户可以创建、更改、删除任务和计划，启用或禁止所有用户的计划，启动或停止dmschsvr，以及重新载入所有计划。此外，用户只能基于自己创建的任务创建计划。

13.1 Dmschsvr

Dmschsvr主要用来执行用户预先设定的SQL语句、存储过程或可执行程序。用户可以为常规任务创建计划，且**dmschsvr**将自动执行这些任务，进而减少大量日常维护。

Dmschsvr会根据SYSSCHEDULE和SYSTASK中存储的信息定期执行任务。SCHEDULE_CREATE、SCHEDULE_ALTER、SCHEDULE_DROP、SCHEDULE_ENABLE以及SCHEDULE_DISABLE分别用来创建、更改、删除、启用以及禁用计划；而TASK_CREATE、TASK_ALTER以及TASK_DROP则分别用来创建、更改以及删除任务。调用上述过程后，存储在SYSSCHEDULE和SYSTASK中的信息也将会被更改。

Dmschsvr运行期间，首先会使用特殊用户通过ODBC连接到数据库，之后扫描存储在SYSSCHEDULE中的所有计划信息，以找出已启用的计划，最后会将计划的信息读入系统。根据这些信息，**dmschsvr**会计算出当前时间与执行时间之间时间间隔最短的任务，并将该任务的数据读入队列，最后，该任务在执行时间到达时执行。如果SYSSCHEDULE中的数据有变化，**dmschsvr**将自动重新载入所有计划。

用户可以通过使用命令**dmschsvr –d db_name**将**DB_SchSv**设置为1或调用*start_dmschsvr('taskNum','logPath')*以启用**dmschsvr**。有关更多START_DMSCHSVR的详细信息，请参考*SQL命令与函数参考手册*。此外，命令**dmschsvr –d**还具有其它参数：-n，指定可同时可执行任务的最大数；-p，指定计划日志的存储路径。不过，用户只能通过调用*stop_dmschsv*来停止**dmschsvr**，用户调用*stop_dmschsvr*之后，**dmschsvr**将会在1分钟后停止服务。

Dmschsvr启动后，每天都将产生日志文件，该日志文件的格式为*<db_name><_><date>.log*，例如DB_20160304.log。日志文件主要记录dmschsvr的运行状况、计划和任务，包含启动时间、结束时间和异常信息。通过查看日志文件，用户可以监控dmschsvr是否运行正常，以调试计划和任务直至正常为止。随着日志文件的增加，用户可以手动删除一

些旧的日志文件，也可以通过调用函数SCHELOG_CLEAN()删除指定日期前的日志文件。有关SCHELOG_CLEAN()的详细信息，请参考*SQL命令与函数参考手册*的*系统存储过程*章节。此外，用户还可使用关键字DB_SchLgDir和DB_SchLgLev来分别设置dmschsvr日志文件的存储目录和级别。有关**DB_SchLgDir**和DB_SchLgLev的详细信息，请参考*dmconfig.ini中的关键字*章节。

13.2 创建计划

使用如下命令创建计划：

```
dmSQL> CALL SCHEDULE_CREATE ('schedule_name','task_name', 'timetable',
'starttime','endtime');
```

➲ **示例**

创建任务**insert_t1**，为表**t1**插入数据：

```
dmSQL> CALL TASK_CREATE('insert_t1','SQL_STATEMENT','INSERT INTO t1
VALUES(1,2)');
```

为任务**insert_t1**创建计划**insert_into_t1**：

```
dmSQL> CALL SCHEDULE_CREATE('insert_into_t1', 'insert_t1', '10 0,1 * *
*', '2012-12-12 12:00:00', '2015-12-12 12:00:00'); // The task
'insert_t1' will run at 0:10 and 1:10 every day from 2012-12-12 12:00 to
2015-12-12 12:00).
```

13.3 更改计划

使用如下命令更改计划：

```
dmSQL> CALL SCHEDULE_ALTER('schedule_name', 'task_name','timetable',
'starttime','endtime');
```

➲ **示例**

更改计划**insert_into_t1**。在该例中，将执行计划**"10 0,1 * * *"**更改为**"20 2,3 * * *"**。有关计划**insert_into_t1**的更多信息，请参考13.2章*创建计划*的例子。

```
dmSQL> CALL SCHEDULE_ALTER('insert_into_t1', 'insert_t1', '20 2,3 * *
*', '2012-12-12 12:00:00', '2015-12-12 12:00:00'); // The task
'insert_t1' will run at 2:20 and 3:20 every day from 2012-12-12 12:00 to
2015-12-12 12:00).
```

13.4 删除计划

使用如下命令删除计划：

```
dmSQL> CALL SCHEDULE_DROP('schedule_name');
```

➲ **示例**

删除计划**insert_into_t1**。有关计划**insert_into_t1**的更多信息，请参考13.2章*创建计划*的例子。

```
dmSQL> CALL SCHEDULE_DROP('insert_into_t1');
```

13.5 重新载入计划

使用如下命令重新载入所有已启用计划：

```
dmSQL> CALL SCHEDULE_RELOAD;
```

➲ **示例**

将所有已启用计划重新载入系统：

```
dmSQL> CALL SCHEDULE_RELOAD;
```

14 创建用户自定义函数

DBMaster中允许开发者创建自己的用户自定义函数（UDF）。一旦在DBMaster中建立一个UDF，它就可作为一个新的内建函数来使用。创建一个用户自定义函数是很方便的，下面将介绍用户自定义函数的整个创建过程。

➲ **创建一个用户自定义函数：**

1. 用C写用户自定义函数（UDF接口）
   - a) 写include语句
   - b) 写函数头
   - c) 写函数的传递参数
   - d) 如果必要，预分配内存
   - e) 如果需要，定义错误代码
2. 为UDF建立动态链接库

在DBMaster中创建UDF，同时将数据数组传递给UDF。

14.1 UDF 接口

创建UDF的第一步骤是用C编写代码。接下来的部分将介绍一个用C写的UDF例子，并将详细描述UDF中每个代码元素的含义。

示例

如果想创建一个新的UDF：**INT2STR()**，将整型数据转换成字符串，则可通过建立动态链接库将其包含进来。

```
dmSQL> SELECT INT2STR(c1) FROM t1;      // c1 is integer type
```

下面是选自创建**INT2STR()**的用户自定义函数的C源程序：*template.c*：

```
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include "libudf.h"

/* Transfer integer type to string type */
#ifdef WIN32
__declspec(dllexport)
#endif
int INT2STR(int narg, VAL args[])
{
  char *ptag;
  int len;
  char p1[11];
  int rc;
```

```
  if (args[0].type != NULL_TYP)
     {
     sprintf(p1, "%d", args[0].u.ival);
     len = strlen(p1);
     if (rc = _UDFAllocMem(args, &ptag, len))
         return rc;
     memcpy(ptag, p1, len);
     args[0].type = CHAR_TYP;
     args[0].len = len;
     args[0].u.xval = ptag;
     }
  return _RetVal(args, args[0]);
}
```

包含libudf.h文件

DBMaster定义了一些常数、数据类型和通用接口，这些都会在UDF编码中用到。

开发者应在写任何UDF代码前将**libudf.h**文件包含进来：

```
#include "libudf.h"
```

传递参数

在SQL命令中使用UDF，它的参数会被打包到对应C代码的**args**参数中。UDF可通过**args**数组传递输入的数据。UDF控制块也可以调用**args**，此变量总是作为DBMaster提供的通用接口的第一个参数使用。一些通用接口将在后面介绍，如：BLOB通用接口。

每个UDF中的C函数头格式如下：

```
int FUNCTION_NAME(int narg, VAL args[])
{
...
}
```

注意 *args[] 指向一个数组。如果函数仅传递一个参数则使用指针形式 *args。*

*narg*定义函数传递参数的个数。例如：如果一个UDF **MYSUBSTRING (c1, c2, c3)**在SQL命令中被调用，那么**c1**信息通过*args[0]*传递，**c2**通过*args[1]*传递，**c3**通过*args[2]*传递，*narg*定义数组大小为3。

➲ **示例1**

c1的值为'abcdefghijklmn'，*args[0]* 则为：

```
args[0].type   = CHAR_TYP
args[0].len    = 14
args[0].u.xval = "abcdefghijklmn"
```

➲ **示例2**

c2的值为整数30，*args[1]* 则为：

```
args[1].type   = INT_TYP
args[1].len    = 4
args[1].u.ival = 30
```

除了CHAR_TYP、INT_TYP，还有BIN_TYP、FLT_TYP、OID_TYP、BLOB_TYP、DEC_TYP和NULL_TYP，常数都在**libudf.h**文件中定义：

```
#define BIN_TYP   0x0000                /* bit string     data type*/
#define CHAR_TYP  0x1000                /* character      data type*/
#define INT_TYP   0x2000                /* integer        data type*/
#define FLT_TYP   0x3000                /* floating point data type*/
#define OID_TYP   0x4000                /* OID            data type*/
```

```
#define BLOB_TYP  0x5000                  /* BLOB          data type*/
#define DEC_TYP   0x6000                  /* decimal       data type*/
#define NULL_TYP  0xF000                  /* set if column is null   */
```

➲ **示例3**

通过NULL_TYP，开发者可知道输入的数据是否为空：

```
if (args[0].type == NULL_TYP)
{
    /* input data is NULL */
}
else
{
   /* input data is not NULL */
}
```

VAL的完整数据结构在**libudf.h**中定义：

```
typedef struct tVAL {
   i16 type;                  /* data type          */
   i15 len;                   /* data length        */
   union {
      i31    ival;            /* long integer data  */
      i15    sival;           /* short integer data */
      double fval;            /* double data        */
      float  sfval;           /* float data         */
      dec_t  dval;            /* decimal data       */
      char   *xval;           /* pointer to data    */
   } u;
} VAL;
```

在**libudf.h**中，*dec_t* 结构如下，它用于DECIMAL类型：

```
typedef struct
{
    i8 pre;
    i8 sca;
    i8 dgt[20];
    i8 exp;
    i8 junk;
} dec_t10;
typedef dec_t10 dec_t;
```

UDF是通过*VAL*类型传入和传出数据的。如何返回数据将会在稍后章节讨论。

分配内存空间

在C函数中，可能需要分配内存并在函数结束前释放内存。返回值需要分配内存，如字串或临时BLOB ID，它们都存在于UDF中，当函数结束时，DBMaster会自动将内存空间释放。

➲ **示例**

下面是一个UDF例子，函数名为**UDFAllocMem**。*arg*为UDF控制块，*ppt*为获得内存块的指针，*nb*为预计分配内存的大小。这个函数分配内存并一直占用它，直到DBMaster使用此内存：

```
int _UDFAllocMem(VAL *arg, char **ppt, int nb);
```

在*args[0].u.xval*返回一个结果后，DBMaster将通过一个指向**_UDFAllocMem()** 内存分配的指针来释放内存。

```
if (rc = _UDFAllocMem(args, &ptag, 10))
   return rc; /* return error code */
memcpy(ptag, "0123456789", 10);
```

```
args[0].type = CHAR_TYP;
args[0].len = len;
args[0].u.xval = ptag;
```

返回结果

有两种类型的返回值：一个为错误代码，另一个为通过参数类型VAL返回的UDF结果。错误代码返回给DBMaster，但对于用户而言，他们的值会被隐藏而只显示一个错误信息。下面描述如何返回错误代码。

在C程序中UDF的开头格式如下：

```
int FUNCTION_NAME(int narg, VAL args[]);
```

如果**FUNCTION_NAME()**的返回值非0，则产生错误；如果返回值为0，则说明函数执行正常。

在UDF返回值前，通过调用**_RetVal()**来传递输入结果给DBMaster，声明如下：

```
int _RetVal(VAL *arg, VAL rtn);
```

第一个参数*arg*是UDF控制块，第二个参数*rtn*为返回值。下面代码返回整型值30：

```
int rc;                       /* error code */
VAL rtn;
rtn.type   = INT_TYP;
rtn.len    = 4;
rtn.u.ival = 30;
rc = _RetVal(arg, rtn);       /* pass result back to DBMaster        */
return rc;                    /* return error code (0 means no error) */
```

14.2 建立UDF动态链接库

DBMaster提供一个**dmudf.lib**库，它可与UDF源文件建立链接并建立动态链接库。由于动态链接库在Microsoft Windows和UNIX环境上是不同的，所以将对两种情况分别讨论。

在Microsoft Windows环境下的DLL

DBMaster提供**template.c**源代码和编译环境模板文件**udf42.mak**（针对Microsoft VC++ 4.2 版），**udf50.mak** （针对Microsoft VC++ 5.0 版）或 **udf60.mak**（针对Microsoft VC++ 6.0 版），可供 WIN32用户参考；还提供了**udf80.mak**（针对Microsoft Visual studio 2005及更高版本），供WIN32和WIN64用户参考。它们的存放路径为***/udf_templates***。用户可根据模板中的c源文件模式来自定义UDF。

➲ **以下叙述中，将用到udf60.mak：**

1. 确定文件**dmudf.lib**所在位置，使用Visual C++中提供的IDE来修改必要的变化。
2. 拷贝**udf60.mak**模板文件到指定的目录，并对其重命名。
3. 选择<File> -> <Open Workspace>打开上述工程。
4. 在 <Project Workspace>中，选择<File View>，点击**template.c**并删除它，按Delete键。
5. 在工具条中选择<Project>，选择< Add to project >，<Files>，并将自己编写的.c文件添加到make file工程中。
6. 在<Project> -> <Settings>中，选择WIN32 Debug。在<General>标签可改变输出路径。在此模板文件中，设置60Deb作为中间和输出文件的目录。
7. 在<Link>标签中，在Category中选择<General>并在<Output file name>中修改输出.dll文件名。同时在<Object/Library modules>选项中，可以修改DBMaster提供的*dmudf.lib*文件的链接路径。

以上步骤完成后，可建立自己的**dll make**文件。使用类似步骤也可建立一个WIN32 Release版本的dll文件。

C++用户，同样可通过类似步骤创建一个dll make文件，但设置结构成员对齐方式为4 bytes.在VC++ 6.0 IDE工程中，选择C/C++菜单栏，并在**Category**对话框中选择 <Code Generation>。您可以看到structure member alignment选项框，然后选择4bytes。

在写**collect dll**时，使用make file模板应注意设置。如果不想在make file中将**template.c**文件作为默认C文件名，可从**udf60.mak**中删除**template.c**，并将您的C文件添加到**udf60.mak**工程中。

➲ 示例

在DBMaster的**template.c**中，需要包含DBMaster提供的**libudf.h**文件，并且导出函数。可通过VC++程序员指导手册或下面的方法来导出函数：

```
__declspec(dllexport) datatype YOUR_FUNCTION_NAME( ..... )
```

或者在工程中创建一个**def file**来导出函数，并且注意在C源代码中，UDF的函数名必须为大写字母。

以上各步完成后，可创建一个**debug/release version dll**文件，如创建的**udf60.dll**文件。

➲ **以下叙述中，将用到udf80.mak：**

1. 拷贝**udf80.mak**和**udf80.def**、**template.c**和**udf60.dsp**文件到指定的目录，并将**udf60.dsp**重命名为**udf80.dsp**。
2. 编辑**udf80.def**，并用自己的 **.c**文件替换**template.c**。
3. 编辑**udf80.def**，您可以更改输出路径。在此模板文件中设置60Deb作为中间和输出文件的目录。
4. 编辑**udf80.def**，您可以直接将输出**.dll**文件名更改为**udf80.mak**。

以上各步完成后，通过命令可建立自己的**dll make**文件。

打开命令，通过cd命令切换到所需的路径，并且执行如下命令：

```
@CALL bat_vs_env
```

```
@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug">
make.msg
```

bat_vs_env：其值取决于Visual Studio的版本以及操作系统。例如，C编译器为VS2005，操作系统为32位时，可以用"C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat"来替换bat_vs_env。更多详细信息，请参考下表：

VS version	OS	bat_vs_env
VS2005	32bit	C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\amd64\vcvarsamd64.bat
VS2008	32bit	C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat
VS2010	32bit	C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat
VS2012	32bit	C:\Program Files\Microsoft Visual Studio 11.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\vcvars64.bat

为了创建dll的发布版本，您可以使用Release替换命令行"@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug"> make.msg"中的Debug。

UNIX环境下UDF so文件

在UNIX环境，可创建一个**so**文件或UNIX动态链接库。

➲ **示例**

在例子中，用C写UDF源代码的文件为**udf.c**。UDF完成后，可在UNIX系统上使用UDF函数。

```
$ cc –c udf.c

$ ld –o libudf.so udf.o –lm

$ dmsqlt

dmSQL> CREATE FUNCTION libudf.INT2STR(INT) RETURNS CHAR(10);
```

注意 *在UNIX环境下，如上例中，命令选项是可变的，它可为共享模式或其他模式。可参考UNIX手册或相关主页来了解如何通过命令建立一个共享库。*

14.3 创建、使用和删除UDF

接下来是在DBMaster中创建用户自定义函数。以下部分分别描述了创建，查询和删除UDF的语法。

创建UDF

➲ **语法**

```
dmSQL> CREATE FUNCTION <udf_dll_name.function_name>
(<function_datatype>) RETURNS <function_output_datatype>;
```

查询UDF

➲ **语法**

```
dmSQL> SELECT <function_name>(<related_table_column_name>)

FROM <related_table>;
```

删除UDF

➲ **语法**

```
dmSQL> DROP FUNCTION <function_name>;
```

示例

下面说明如何使用UDF文件。

➲ **示例1**

使用一个名为**DMDEMO**的数据库，数据库中包含一个**tb_udf**表。表结构为**number INT, name CHAR(10)**：

```
dmSQL> SELECT * FROM tb_udf;
```

```
 number      name
=========   =======
10          1
20          2
30          3


3 rows selected
```

使用DBMaster提供的**template.c**例子，我们现在可成功创建一个**udf60.dll**。

在**dmconfig.ini**文件中，在**DMDEMO**节中添加一行：

```
[DMDEMO]
DB_DbDir = D:\UDFDEMO
DB_FoDir = D:\UDFDEMO\FO
DB_LbDir = D:\UDF\60Deb   ; add this line
```

想了解更多**DB_LbDir**信息，可参考*dmconfig.ini中的关键字*章节。 设置**DB_LbDir**或将**udf60.dll**放置在***<DBMaster目录>\shared\udf***，因为这是UDF的默认路径。

➲ **示例2**

启动数据库**DMDEMO**，并且创建UDF函数。在例子中：*<udf_dll_name>*为 **udf60**，*<function_name>*为**INT2STR**，*<function_datatype>*为INT型， *<function_output_datatype>*为CHAR(10)：

```
dmSQL> CREATE FUNCTION udf60.INT2STR(INT) RETURNS CHAR(10);
```

UDF函数**INT2STR**返回如下结果。*<function_name>*为**INT2STR**，*<related_table>*为**tb_udf**，*<related_table_column_name>*为**tb_udf**的**c1**：

```
dmSQL> SELECT INT2STR(c1) FROM tb_udf;
```

```
INT2STR(c1)
==========
10
20
30

3 rows selected
```

➲ **示例3**

另一个UDF函数，比如**STR2INT()**，在同一个动态链接文件中：

```
dmSQL> CREATE FUNCTION udf60.STR2INT(CHAR(10)) RETURNS INT;
dmSQL> SELECT STR2INT(c2) FROM tb_udf;

 STR2INT(c2)
==========
1
2
3

3 rows selected
```

➲ **示例4**

当删除一个UDF函数时，只需删除函数名，无需加上**UDF dll**名。在删除UDF函数时，直到数据库终止，UDF函数才被清除。在数据库终止前，函数将一直存在。

```
dmSQL> DROP FUNCTION INT2STR;
```

14.4 创建 XML有效函数

Flexml

Kristoffer Rose的 flexml位于GNU General Public License下，它是一个XML发生器。它通过DTD文件而产生一个LEX文件。FLexml可以在 http://flexml.sourceforge.net下获取。

产生 LEX File

```
$ flexml name.dtd
```

添加用户化的YY_INPUT

原始的LEX 输入变量是一个 文件输入串。必须修改LEX 文件才能使用，UDF Blob作为一个输入源。下面的例子通过添加用户化的YY_INPUT来说明这个修改。

➲ **示例**

下例中通过添加YY_INPUT来修改LEX文件的定义部分。定义部分位于文件的开始，并且位于符号“%{” 和“}%”之间。

```
#include "libudf.h"


typedef struct udf_file
        {
        VAL *args;
        i31 handle;
        i31 rc;
  i31 left;
        i31 rlen;
```

```
        } UDF_FILE;


#undef YY_INPUT
#define YY_INPUT(buf,result,max_size)  {\
  UDF_FILE * uf =(UDF_FILE *)yyin; \
        errno=0; \
        if ( uf->left <= 0 )\
        {\
          result = (uf->rlen=0);\
        }\
        if ( (uf->rc = _UDFBbRead(uf->args, uf->handle, max_size, &(uf-
>rlen), buf))!=0 ) \
  { \
          errno = uf->rc; \
          result = 0;\
        }\
        else\
        {\
    uf->left -= uf->rlen;\
          result = uf->rlen;\
        }\
}\
```

接下来，如下所示，在LEX文件末尾添加UDF函数：

```
#ifdef WIN32
__declspec(dllexport)
#endif
```

```
int XXX_VALIDATE(int nArg, VAL args[])
{
  BBObj bbin;
  UDF_FILE uf;
  int rc = 0;
  int rc2 = 0;
  memset(&uf, 0, sizeof(UDF_FILE));
  memcpy((char *)(&bbin), args[0].u.xval, BBOBJ_SIZE);
  uf.args = args;
  rc = _UDFBbOpen( args, bbin, &(uf.handle));
  if( rc != 0 )
    goto EXIT;
  if (rc = _UDFBbSize(args, bbin, &(uf.left)) )
        {
    goto EXIT;
        }
  yyin = (void *)&uf;
  rc = validdtd00udf();
  rc2 = _UDFBbClose(args, uf.handle);
EXIT:
  if( args[0].type != NULL_TYP ) // null column data
        {
          args[0].type = INT_TYP;
          args[0].len = 4;
          args[0].u.ival = (rc == 0? 1:0);
        }
```

```
  return _RetVal(args, args[0]);
}
```

编译 dll/so

```
flex name.l
cc -c -DBUILD_DLL lex.name.c -Idbmaster-installed-dir/include
```

创建 UDF

```
dmSQL> CREATE FUNCTION dllname.udfname(BLOB) returns int;
```

创建带有 check 约束的字段

```
dmSQL> CREATE TABLE table-name( c1 XMLTYPE CHECK udfname(value) = 1);
```

DBMaster DTD有效函数发生器

一个命令行工具可以用来为指定的DTD产生有效的UDF。

DTD文件的名称是必须的。如果没有指定，将产生错误。

输出路径是可选的，如果没有指定，文件将存放于当前工作路径下。

前缀是可选的，如果指定了前缀，产生的文件将在文件名中使用该前缀；如果没有指定，将使用没有扩展的DTD文件名。

```
$ dmxmludfmk -dtd dtd-file-name [-o output-directory] [-p prefix]
```

如下所示，产生了多个文件：

- Lex 文件：<用户指定的前缀.l>
- Yacc 文件：<用户指定的前缀.y>
- UDF 函数文件：<用户指定的前缀> udf.c 和 <用户指定的前缀> udf.h
- UDF 函数被命名为 <用户指定的前缀>_VALIDATE
- hash.c and hash.h 提供hash功能

- UNIX 平台下的Makefile或Windows平台下的Makefile.msvc

➲ **示例1**

```
Make <user-specified-prefix>.so    ;for UNIX
```

➲ **示例2**

```
Nmake /f Makefile.msvc             ;for Windows visual studio 2005 or
2008
```

➲ **示例3**

```
nmake /f Makefile.msvc COMPILER=VC60   ;for Windows visual studio 6.0
```

➲ **示例4**

```
nmake /f Makefile.msvc OSTYPE=$OSTYPE  ; for cygwin environment
```

请注意 dmxmludfmk 支持 ASCII、 BIG5、gb、shiftJIS 和 utf8；同时flexml支持内部定义的DTD实体内容替换。

默认有效器

I_VALIDATE 被指定为检查XML语法的默认有效器。I_VALIDATE 并不提供针对DTD或者XMLSchema的有效确认。I_VALIDATE 是libmedia 字典的一部分。

14.5 UDF BLOB通用接口

当今多媒体对用户来说已经相当重要并且应用广泛，DBMaster提供一个通用接口来访问BLOBs，这个接口通过一个文件语句的方法，因此程序员可很容易的写出包含BLOB类型数据的UDF。FILE、LONG VARCHAR和LONG VARBINARY是数据库中存储BLOB数据的数据类型。

DBMaster中的一些新特征需要一个临时BLOB来处理临时结果集。DBMaster支持的临时BLOBs为程序员写UDF带来方便。程序员可打开一个固定的BLOB、读取数据、执行转换函数或进行其他一些操作，并将结果保存到一个新的临时BLOB中，然后在UDF中返回结果。API会将此临时BLOB作为一个普通BLOB字段来读取。

BLOB通用接口函数

DBMaster为程序员写UDFs提供了BLOB通用接口函数。DBA需要在启动数据库前，在**dmconfig.ini**文件中设置**DB_FoDir**关键字，此目录为临时BLOBs文件存放路径。一个临时BLOB将以外部文件形式创建在**DB_FoDir**定义的目录中，文件名格式为“**__??????.TMP**”，这里“?”代表一个在[0-9，A-Z]之间的任一字符。当数据库关闭并重启时，所有这种类型的文件将被删除。

_UDFBbOpen()

打开一个BLOB对象**bbObj**，通过**pHandle**返回一个句柄。通过UDF输入参数**Arg[i]**获得**bbObj**。如果成功打开**BLOB**，函数返回0；否则返回错误代码。

```
int _UDFBbOpen(VAL *Arg, BBObj bbObj, i31 *pHandle);
```

_UDFBbRead()

读取属于一个句柄的BLOB。在调用此函数前，需使用函数**_UDFAllocMem()**分配一个szBuf大小的缓冲区（**pBuf**），返回的数据将

会被存储在**pBuf**中，并且实际的读取长度在**szRead**中。如果**szBuf**为负值，则不读取任何字符：

```
 int _UDFBbRead(VAL *Arg, i31 handle, i31 szBuf, i31 *szRead,
char *pBuf);
```

_UDFBbSeek()

此函数用于在**BLOB**中，设置下一次输出操作的位置。新的位置是从开始位置、当前位置或文件末尾算起的偏移字节数，具体是由**ptrname**取**SEEK_BB_BEG**、**SEEK_BB_CUR**或**SEEK_BB_END**来区分，这些常量在**libudf.h**中定义。此函数只作用于**_UDFBbOpen()**与**_UDFBbClose()**期间，而不是**_UDFBbCreate()**与**_UDFBbClose()**期间：

```
int _UDFBbSeek(VAL *Arg, i31 handle, i63 offset, i16 ptrname);
```

_UDFBbCurOffset

该函数通过**pOffset** 返回一个打开的**BLOB**文件的当前位置或**BLOB**偏移量。即使当前的偏移量大于或等于2^{31}，该函数的返回值最大是2^{31}- 1。

```
int _UDFBbCurOffset(VAL *Arg, i31 handle, i31 *pOffset);
```

_UDFBbCurOffsetEx

与_UDFBbCurOffset函数不同，该函数通过**pOffset**返回一个打开**BLOB**文件的实际当前位置或**BLOB**偏移量的真实值，即使返回值大于2G。

```
int _UDFBbCurOffsetEx(VAL *Arg, i31 handle, i63 *pOffset);
```

_UDFBbClose()

关闭由**_UDFBbOpen()**打开的，或**_UDFBbCreate()**创建的**BLOB**：

```
int _UDFBbClose(VAL *Arg, i31 handle);
```

_UDFBbCreate()

创建一个临时**BLOB**并为**_UDFBbWrite**返回句柄。调用者应为**BBObj**结构预分配空间，**pBbObj**将指向这个结构体，并且**_UDFBbCreate()**、**_UDFBbWrite()**和**_UDFBbClose()**会使用它。**BBObj**用于标识这个临时

BLOB。例如：如果想删除临时**BLOB**，可通过调用**_UDFBbDrop()**，使用其中的**BBObj**参数来完成。

如果操作成功，**pHandle**将返回一个**BLOB**句柄，他类似于打开文件的句柄，**_UDFBbWrite()**中用此句柄来写和**_UDFBbClose()**中用此句柄来关闭。

另外，可以定义临时**BLOB**是创建在文件（**BB_TEMP_FO**）中还是内存中（**BB_TEMP_MEM**）。如果调用者在内存中定义临时**BLOB**，则并不代表临时**BLOB**将一定会创建在内存中 -- 这个操作可能会受内存的限制。如果内存中存在原始临时**BLOBs**，输入数据超过了规定的大小限制，临时**BLOBs**可能会被操作系统转换成文件。程序员不应该在写代码时只依据这个定义。

如果操作成功，函数返回值为**0**；操作失败，则返回错误代码。

在读新的临时**BLOB**前，必须先通过**_UDFBbClose()**关闭它，然后重新用**_UDFBbOpen()**打开。只有关闭并重新打开临时**BLOBs**后，**_UDFBbSeek()**的使用才有效：

```
int _UDFBbCreate(VAL *Arg, BBObj *pBbObj, i31 *pHandle, i31 Opt);
```

_UDFBbWrite()

在使用**_UDFBbCreate()**产生临时**BLOB**之后，可使用**_UDFBbWrite()**向临时**BLOB**中写数据，句柄来自于**_UDFBbCreate()**，**pBuf**指向输入数据并且它的长度为**szBuf**。如果写成功，函数返回值为**0**；否则返回错误代码。

```
int _UDFBbWrite(VAL *Arg, i31 handle, i31 szBuf, char *pBuf);
```

_UDFBbDrop()

如果从**UDF**要返回一个临时**BLOB**，通常不用删除这个临时**BLOB**；系统会控制它的生命周期。如果不返回创建的**BLOB**，则需要用函数来删除这个临时**BLOB**。这个函数对于永久**BLOB**无效；如果用此函数处理永久**BLOB**，则会返回**ERR_BLOB_INV_BLOB**错误。如果函数返回值为**0**，则操作成功；否则，返回错误代码：

```
int _UDFBbDrop(VAL *Arg, BBObj bbObj);
```

_UDFBbSize()

此函数通过**pLen**参数返回BLOB数据大小。**BbObj**可以是一个永久BLOB或临时BLOB。不过即使数据大小大于或等于2^{31}，该函数的返回值最多是2^{31} - 1，函数返回值为0，则操作成功；否则返回错误代码：

```
int _UDFBbSize(VAL *Arg, BBObj bbObj, i31 *pLen);
```

_UDFBbSizeEx()

与函数 _UDFBbSize不同，该函数通过**pLen**参数返回**BLOB**数据大小的真实值。**BbObj**可以是一个永久BLOB或临时BLOB。函数返回值为0，则操作成功；否则返回错误代码：

```
int _UDFBbSizeEx(VAL *Arg, BBObj bbObj, i63 *pLen);
```

示例

以下示例如何创建一个用户自定义函数**MYCONVERT**，它的输入为varchar，输出为临时BLOB。

➲ 创建用户自定义函数MYCONVERT：

1. 在UNIX环境，用**myudf.c**建立动态库（源代码如下）：

```
cc -g -c myudf.c

ld -G -o myudf.so myudf.o
```

2. 启动数据库。

3. 在dmSQL提示符下输入。

```
dmSQL> CREATE FUNCTION myudf.myconvert(VARCHAR(100))  // input string

    2>  RETURNS LONG VARCHAR;                         // output BLOB

dmSQL> SELECT myconvert(c1) FROM mytable;             // output temp
BLOB
```

UDF函数**MYCONVERT**的源代码：

```
#include "libudf.h"

int MYCONVERT(int nArg, VAL args[])
```

```
{
  int     rc = 0, trc;         /* return code                    */
  BBObj   tmpobj;              /* output temp BLOB               */
  i31     handle;              /* handle of created temp BLOB    */
  boolean fgCreate = FALSE;    /* temp BLOB has been created?    */
  char   *pInData, pOutData[4096];/* input/output data buffer       */
  i31     nInData, nOutData;   /* input/output data buffer length */

  if (args[0].type == NULL_TYP)
    goto cleanup;

  pInData = args[0].u.xval;    /* get input data                 */
  nInData = args[0].len;               /* input data length
*/

  /* create a temp BLOB in file */
  if (rc = _UDFBbCreate(args, &tmpobj, &handle, BB_TEMP_FO))
    goto cleanup;
  fgCreate = TURE;

  /* any real processing function */
  RealConvert(pInData, nInData, pOutData, &nOutData);

  /* write result data to temp BLOB */
  if (rc = _UDFBbWrite(args, handle, nOutData, pOutData))
    goto cleanup;
```

```
  /* close created temp BLOB ( temp BLOB is still alive) */
  if (rc = _UDFBbClose(args, handle))
    goto cleanup;

  args[0].type = BLOB_TYP;
  args[0].len = BBID_SIZE;
  args[0].u.xval = (char *)&tmpobj;
                /* _RetVal() does a copy from this local buffer  */

cleanup:
  if (rc)
    {
    /* error handle */
    if (fgCreate)
      {
      _UDFBbClose(args, handle);          /* close created temp BLOB
*/
      trc = _UDFBbDrop(args, tmpobj); /* drop it because of failure */
      if (trc > rc)
        rc = trc;
      }
    return rc;
    }
  else
    return _RetVal(args, args[0]);

}/* MYCONVERT() */
```

错误处理

在利用BLOB通用接口写BLOB UDF时，使用以下方法处理错误。

ERROR（327）：无法打开或创建BLOB字段

在使用其它BLOB函数接口前，需先使用**_UDFBbOpen()**打开BLOB或使用**_UDFBbCreate()**创建一个新的临时BLOB。

ERROR（328）：BLOB字段偏移量无效

在UDF中，使用**_UDFBbSeek()**来搜索超过BLOB实际长度的偏移量。

ERROR（331）：BLOB没有在创建状态

只有在**_UDFBbCreate()**创建了临时BLOB并且处于打开状态时，**_UDFBbWrite()** 才可在此临时BLOB上正常操作。例如，如果在用**_UDFBbOpen()**打开的一个BLOB上使用它时，则会发生此错误。

ERROR（330）：BLOB 没有在开启状态

只有在 **_UDFBbOpen()** 处于开启状态时，**_UDFBbRead()**才可在此BLOB （包括临时BLOB）上正常操作。

ERROR（332）：BLOB对象没有关闭

任何时候通过**_UDFBbOpen()**或**_UDFBbCreate()**打开了BLOB，程序员都应调用**_UDFBbClose()**来关闭已打开的BLOB。

ERROR（322）：在配置文件中无文件对象目录，无法插入文件对象

如果使用临时BLOB，则必须在dmconfig.ini文件中设置DB_FoDir关键字。如果没有设置，则在创建临时BLOB时会失败并产生此错误。

14.6 与UDF关联的dmconfig.ini参数

DB_StrSz

除了**DB_LbDir**和**DB_FoDir**，**dmconfig.ini**文件中还有一个相关的关键字**DB_StrSz**。

```
DB_StrSz=<value>
```

这个关键字定义了在使用用户自定义函数（UDF）时，STRING类型数据的返回长度。由于UDF只能返回固定长度的数据，这个关键字可以限制STRING数据的长度以避免接受过长的字串。 默认值为255，取值范围为1到4,096。此关键字在客户端和服务器端都会用到，而客户端具有较高优先权。

15 数据库恢复、备份和还原

对所有的数据库管理系统（DBMS），数据库会因为硬件或软件的问题而遭到破坏。一个DBMS可能在毫无准备的情况下遭到严重的损害，一个好的DBMS应该能提供一些机制来恢复这些遭到损坏的数据库。事实上，这也正是数据库管理系统胜过一般文件管理系统的主要优势之一。

由DBMaster提供的恢复、备份和还原特征，来确保您的数据库在遭到硬件或软件损害后，仍能保持数据库的可靠性和数据的一致性。

15.1 数据库发生故障的类型

我们把数据库的故障分为两类：系统故障和介质故障。无论出现何种故障，都有可能造成数据库中数据的不一致或丢失。当数据库遭到破坏时，数据库管理系统必须提供一些措施来恢复数据库，或者利用备份数据库来还原遭到损坏的数据库。

系统故障

系统故障，又称作实例故障，是来自计算机系统的内存故障。造成的原因可能是突然断电、应用程序和操作系统错误、内存错误等，导致数据库管理系统的异常结束或事务的异常中止。

当出现系统故障时，应用软件和活动事务都被异常中止。正处理事务的状态或没有完全写到磁盘的事务的可靠性不能保证，这些事务应当被恢复。数据库管理系统需要利用事务日志来对数据库做灾难恢复，这样才能将数据恢复到未遭受系统故障前的状态。

介质故障

介质故障（例如，磁盘故障）是计算机系统的磁盘存储介质遭到损坏。造成磁盘损坏的原因很多，可能是单纯的读写头损坏、某段磁道损坏或者火灾、地震、水灾等天灾人祸。

介质故障一旦发生，数据丢失就无法避免，而且可能不止一个文件本身被损坏。这时就必须依靠数据库管理系统所提供的备份和还原功能来恢复数据库。

15.2 数据库损坏后的恢复

恢复的目的就是要保证数据库在遭到破坏后，提交的事务反映到数据库中，未提交的事务不反映在数据库中，并且尽快地返回到正常状态。

DBMaster使用日志文件和检查点（checkpoint）来恢复数据库。日志文件和检查点能使用户在毫无察觉的情况下，尽快的确保所有事务的恢复。

日志文件

日志文件实时记录着对数据库的所有改变和每次改变的状态。在系统发生故障后，利用日志文件的历史记录，DBMaster可以恢复和重做那些已经被改变但尚未写入磁盘的事务，或者取消那些被异常终止的事务。

如果数据库运行在备份模式上，日志文件还可以存储更多的信息，DBMaster可以使用这些信息去还原受损的数据库。这些信息在做备份前，将储存在日志文件中，并占据一定的空间。当然，在您做了日志文件备份后，DBMaster会释放此空间，让新的事务来使用。

在数据库的还原过程中，DBMaster将备份日志文件中的信息添加到备份数据库中。因此，只需要备份记录两次完整备份间数据库改变的日志文件。

检查点

检查点是把系统的执行状态反应到磁盘上的一个系统事件。也就是说，此时系统内存的内容与磁盘上的内容完全相同。在检查点，DBMaster会将内存区中的所有日志记录和修改过的数据页写入磁盘，并且回收不再作为备份或恢复使用的日志块。

执行了检查点后，会节省数据库在系统发生故障后的启动时间，DBMaster会将最后一次的检查点时间和当时的活动事务状态写到日志文件的文件头处。在数据库恢复的过程中，DBMaster会利用这些信息去决定哪些事务需要重做，哪些事务需要取消，哪些事务应该被忽略。

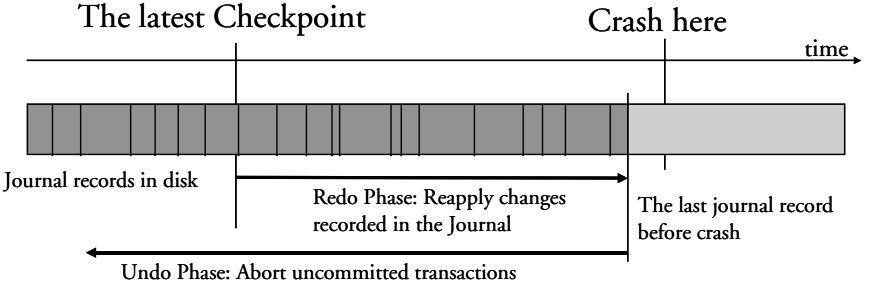
在日志空间写满时，DBMaster会自动地执行检查点，以回收使用的日志空间。如果这样仍不能满足完成当前事务的日志空间的需要，这个事务将会被取消。DBMaster也会在数据库启动、关闭或执行在线备份时，自动执行检查点。

数据库管理员可以利用CHECKPOINT命令来手动执行检查点。执行检查点的最恰当时间，会受到数据库中活动事务的多少、日志文件的大小和数量、事务平均大小的影响。也就是说，每个数据库应该执行检查点的时间都不同，有经验的数据库管理员应根据实际需要来执行检查点。手动执行检查点可以减少事务遇到日志空间不足的情况，并可以缩短数据库启动、终止、备份的时间。

执行检查点也要占据相当长的一段时间，这段时间的长短主要取决于从上次执行检查点到现在的事务大小和数量。当前的事务需要等DBMaster决定回收哪一个日志记录后再执行，然而，在DBMaster写日志记录和数据页到磁盘时，当前事务不需等待。

恢复步骤

如果系统发生故障或在启动过程中出现错误，DBMaster数据库会在启动时自动执行恢复动作。在恢复的过程中，DBMaster总会执行两个步骤：重做和取消。



第一个步骤是重做（或重新应用）已经记录在日志文件中的事务。此步骤的必要性在于可能存在一些已经确认，但尚未反应到磁盘上的事务数据。然而，这些记录都已存储到日志文件中，并且通过这个步骤可以记

录到数据库中。执行完这个步骤，所有提交的、未提交的事务都会记录到数据库中。

第二个步骤是取消（或回滚），把没有得到确认的事务撤回。此步骤的必要性在于系统发生故障时，无法确定进程中事务的准确状态。因为事务必须保证不是确认就是撤回，所以必须将这些没有得到确认的数据全部取消。做完这个步骤后，数据库中就包含了所有提交的事务数据，但是不包含未提交的事务。

利用启动恢复机制只能解决因系统故障引起的数据损坏，只有利用备份和还原机制才能解决存储介质的损坏。也就是说，如果数据库遇到了存储介质故障，就无法靠启动数据库时的自动恢复机制来解决，即在这种数据库遭到损坏的情况下，数据库无法启动。这时候，数据库管理员通常需要从数据库的备份来还原。若是管理者从未做过备份，那么可以使用数据库的强制启动模式来启动数据库，即用在**dmconfig.ini**配置文件中的关键字**DB_ForcS**。这允许您强制启动数据库，并且允许您导出数据库中未受损坏的数据。要获得更多有关强制启动模式的信息，请参考*强制启动数据库*。

强制启动数据库

如果在正常启动数据库的过程中出现错误，DBMaster就会自动执行恢复功能。如果数据库在自动恢复后仍然无法正常启动，那么就有可能存在磁盘介质错误，此类错误要求数据库从最近一次的备份处还原。如果数据库没有做作备份，也无法启动，就可以使用DBMaster提供的*强制启动*模式来启动它。

DBMaster提供的强制启动选项是这样设置的。我们可以在**dmconfig.ini**配置文件中使用**DB_ForcS**关键字，设置此关键字为1，代表使用强制启动数据库；设置为0，代表不使用强制启动数据库。当用户开启强制启动模式时，DBMaster会忽略数据库启动时所发生的错误。

如果数据库仍然无法启动，在下面过程中提供了一个可供选择的方式。然而，在执行这个过程前，您需要备份所有的数据和日志文件。

➲ **当数据库不能以强制启动方式启动时，按以下步骤操作：**

1. 关闭强制启动模式**dmconfig.ini**（**DB_ForcS** = 0）。
2. 启动模式设为新日志模式**dmconfig.ini**（**DB_SMode** = 2）。
3. 重启数据库。
4. 重新设置启动模式为正常模式**dmconfig.ini**（**DB_SMode** = 1）。

DBMaster提供用新日志方式来强制数据库启动，而不进行任何恢复操作。所以，如果启动时有严重错误，数据库可能处于不一致状态，以这种方式启动数据库后，应当检查数据库的一致性。要想获取更多有关数据库一致性检验的信息，请参考6.12章*检验数据库的一致性*。

15.3 备份的类型

备份是为了保护数据库，以免数据库受到介质故障或其他故障的影响。在存储介质损坏后，可能有一个以上的数据库文件遭到物理破坏而不可用，数据库管理者应该使用最近的一次备份来替换受损的文件以重建数据库。

数据库备份由备份序列组成。一个备份序列包括三部分：一个完整备份，与该完整备份相关的所有差异备份以及在此完整备份执行后的增量备份。

完整备份

完整备份是指在某时间点的一个数据库的完整拷贝。也就是说，它复制了数据库中的所有数据、日志文件和**dmconfig.ini**配置文件。无论是在线还是离线，数据库管理员都可以执行一个完整备份。

完整备份是整个数据库的拷贝，因此它需要很大的存储空间。然而利用完整备份进行数据库还原相对节省些时间。完整备份能让数据库还原到数据库完整备份时的状态。

一个有效的完整备份通常指定一个备份ID，备份ID是一个时间/日期戳，通常用于确保完整备份、差异备份和增量备份之间的顺序关联。请注意，差异备份只保存最近一次完整备份后改变的数据。使用差异备份恢复数据库时，差异备份的基础是非常必需的，仅有一个差异备份是不够的。增量备份只保存当前完整备份和下一个完整备份之间的内容，所以必须按照完整备份的顺序来恢复数据库。在一个新的日志模式下修复、恢复、启动数据库，或者改变数据库的备份模式，都需要先做一个有效地完整备份。

执行一个完整备份通常有三种方式：第一种方式是使用备份服务器（详情请参考第15.6章*备份服务器*），通过备份服务器进行完整备份，可以使用dmSQL工具或JServer Manager工具；第二种方式是交互式的完整备份，它不需要启动备份服务器，我们推荐使用JServer Manager工具来

执行这种类型的完整备份；第三种方式是执行离线的完整备份。有关如何设置离线完整备份的详细资料，请参考第15.5章*离线完整备份*。

差异备份

差异备份可使用户节省时间和磁盘空间，它使用了一种不同的方式，不同于完整备份仅仅复制所有文件。

特定的差异备份是基于最近的完整备份上的，该完整备份就是差异备份的基础。在创建一个差异备份时，必须存在一个差异备份基础。

差异备份仅包括差异备份基础创建之后数据的改变情况。一个差异备份基础通常被几个差异备份使用。之后，对于完整备份（差异基础备份）和差异备份来说，都需要重新恢复数据库。

由于日志文件的频繁改变，差异备份仅包括数据文件（所有DB文件和BB文件）和有用的日志块。

差异备份只记录最近一次完整备份后更改的数据，这使得用户能够更频繁地备份数据库。因为差异备份比完整备份小，所以频繁备份数据库可降低数据丢失的风险。在下列情况下可考虑使用差异备份：

- 自最后一次完全备份后，数据库仅有一小部分发生改变。当同样数据多次修改时，差异备份尤其有效。
- 需要频繁的备份，但并不需要频繁的完整备份。
- 恢复数据库时尽量减少事务日志备份向前回滚的时间。

执行差异备份有两种方法：一是使用备份服务器。开启数据库前，需要配置一些关键字，要想获得更多信息，请参考第15.6章*备份服务器*。二是调用在线差异备份。使用第二种方法时，我们推荐使用JServer Manager工具，详细信息可参考*服务器管理工具用户手册*。

增量备份

增量备份是指上次做过备份（完整备份、差异备份或增量备份）之后，有更改的日志文件拷贝。也就是说，增量备份只能在完整备份或差异备份之后执行，执行一个新的完整备份将是一系列备份的开始，之后的增

量备份只是这个备份序列中的一部分，并且不能用于其它的完整备份或差异备份。请注意增量备份由一系列日志文件组成，这些日志文件记录了备份模式（**DB_BMode**）开启后的所有更新数据。数据库运行于普通模式（**DB_SMode** = 1）时，若执行增量备份，则需先执行完整备份或差异备份。而数据库运行于复制模式（**DB_SMode** = 4）时，执行增量备份前无需执行完整备份或差异备份。增量备份只拷贝了在上次做过备份之后的数据库的改变。数据库管理员只有当数据库在线时，才能执行增量备份。

增量备份只备份日志文件，所以它们只需占用一小部分的存储空间。但是，在做备份还原时，因为数据库管理系统必须重做此备份日志文件中的所有事务，所以会花比较长的时间。使用者可以结合使用完整备份、差异备份与增量备份，把数据库还原到前一次完整备份或差异备份与最后一次增量备份中的任一时间点。

执行一个增量备份主要有两种方式（此外，还存在一种称作增量备份至当前的，单独作为一种备份类型）。第一种方式是通过备份服务器进行增量备份。只有启动了数据库上的备份服务器才能使用这种方式。有关更多的通过备份服务器来进行增量备份的信息，请参考15.6章*备份服务器*。第二种方式是交互式的增量备份，此种类型的增量备份不需要启动备份服务器。在此，我们推荐用户使用JServer Manage工具来执行交互式的增量备份。增量备份的详细信息，请查看*服务器管理工具用户手册*。

离线备份

离线备份是在数据库关闭后，对数据库做的备份。数据库管理员必须设法通知所有用户都要断开连接。离线备份给用户带来了不便，因为他们必须记住在数据库关闭前，完成所有活动事务和断开连接。而且，数据库管理员在离线的状态下只能执行完整备份。

DBMS不一定要提供离线备份的功能，因为在关闭数据库后，您可以通过操作系统命令去备份一个数据库。数据库管理员可以通过这个方式去执行一个离线备份，或者通过使用JServer Manager工具：一个使用方便的图形化工具，在不使用操作系统命令的情形下，轻松地实现一个离线备份。

在线备份

在线备份是指数据库运行时所做的备份。数据库管理员无需关闭数据库，用户也无需断开与数据库的连接就能执行在线备份。在线备份给用户提供了方便，因为不需要其他用户任何操作。当数据库在线时，数据库管理员不仅可以执行完整备份，还可以执行差异备份和增量备份。

DBMS必须提供在线备份数据库的功能，因为DBMS在备份的过程中仍在运行并且和用户连接。DBMaster不仅可以使用dmSQL工具和执行操作系统命令的方法来手动地执行在线备份，还可以通过JServer Manager工具：一个使用方便的图形化工具，在不使用操作系统命令的情形下，轻松地实现一个在线备份。

在线增量备份至当前

DBMaster也提供了另一种备份方式，称作在线增量备份至当前。

对于拥有一个以上日志文件的数据库而言，在线增量备份和在线增量备份至当前的不同之处在于：在线增量备份将备份从上次备份之后所有修改过的日志文件，但是不包括目前使用的这个日志文件；而在线增量备份至当前则包括了目前所使用的这个日志文件。也就是说，在线增量备份只能将数据库恢复到日志文件中所记录的最后一次提交事务的状态，而在线增量备份至当前可以将数据库恢复到目前日志文件所记录的状态。

如果数据库只有一个日志文件，在线增量备份和在线增量备份至当前是一样的。 在这种情况下，对这两种增量备份方式DBMaster都是备份当前这个日志文件。

在线增量备份至当前可以通过使用JServer Manager工具来执行。有关如何执行在线增量备份至当前的详细信息，请查看*服务器管理工具用户手册*。

15.4 备份模式

数据库备份模式决定了DBMaster是否可以执行在线增量备份和增量备份的数据类型，以及何时可以释放那些已经结束的事务所占据的日志空间。 DBMaster提供了三种数据库的备份模式：不备份（NONBACKUP）、只备份数据（BACKUP-DATA）、备份数据和BLOB（BACKUP-DATA-AND-BLOB）。

备份模式	表空间备份模式	用户自定义表空间（DATA）	用户自定义表空间（BLOB）	系统表空间（DATA和BLOB）
不备份		No	No	No
只备份数据		Yes	No	Yes
备份数据和BLOB	Backup BLOB Off	Yes	No	Yes
	Backup BLOB On	Yes	Yes	Yes

不备份模式

不备份（NONBACKUP）模式无法为插入或更新的数据提供保护，在这种模式下不能执行在线增量备份。当数据库遭到系统故障时，可以使用日志文件去恢复数据库。当存储介质损坏时，可能导致数据的丢失，在执行检查点后，那些未利用的日志块都可以收回使用。但是日志空间一旦写满，数据库就只能恢复到最后一次作完整备份的状态。

备份数据模式

备份数据模式可以为插入或更新的数据（不包括BLOB数据）提供保护。在这种模式下，数据库管理员可以执行在线增量备份，但是只有非BLOB数据可以储存在备份文件中。当数据库遭到系统故障时，可以使用日志文件去恢复数据库，并且可以部分恢复来自存储介质的损坏。尽管在恢

复数据库时，数据可以还原至存储介质损坏的时间，但BLOB数据的更改将会丢失。那些没有被活动事务利用的日志块，只有在检查点执行后并且在日志文件已经备份的情况下，才可以被重新使用。

备份数据和BLOB模式

备份数据和BLOB模式允许为插入或更新的数据（包括BLOB数据）提供保护。在这种模式下，数据库管理员可以执行在线增量备份，并且所有的数据都将储存在备份文件中。当数据库遭到系统故障时，可以使用日志文件去恢复它，同时也可以恢复来自存储介质的损坏。在恢复数据库时，数据可以还原至存储介质损坏的时间，包括所有的BLOB数据。那些没有被活动事务利用的日志块，只有在检查点执行后并且在日志文件已经备份的情况下，才可以被重新使用。

表空间的BLOB备份模式

DBMaster提供的备份模式是作用在数据库上的。也就是说，数据库中的所有表空间都使用同一种备份模式。如果数据库在BACKUP-DATA-AND-BLOB模式下，DBMaster会将数据（包括BLOB数据）的更改信息全部记录到日志文件中。备份数据和BLOB模式会快速消耗日志空间，所以应该为频繁的备份提供较大的日志空间。

这个选择对于数据库管理员可能有些困难，因为我们在备份数据时，有些BLOB数据并不重要，管理者不想花费大量的日志空间去储存它。针对这个需求，DBMaster允许数据库管理员为新建的表空间更改数据库的备份模式。

在CREATE TABLESPACE命令中，您可以使用BACKUP BLOB ON /OFF 选项来表示这个表空间的BLOB数据重不重要，需不需要备份，ON表示重要，需要备份。OFF表示不重要，不需要备份。

每一个表空间的备份模式都是数据库备份模式和表空间备份模式的结合，它有如下规定：

- 如果数据库运行在BACKUP-DATA-AND-BLOB模式下，同时创建表空间时，使用BACKUP BLOB ON选项，DBMaster将备份那个表空间上的BLOB数据。
- 如果数据库运行在BACKUP-DATA-AND-BLOB模式下，同时创建表空间时使用BACKUP BLOB OFF选项，DBMaster就不会备份那个表空间上的BLOB数据。
- 如果数据库运行在BACKUP-DATA模式下，在您创建表空间时，无论您是否使用BACKUP BLOB ON/OFF选项，DBMaster都不会备份BLOB数据。

新建表空间的默认选项为BACKUP BLOB ON。当数据库运行在BACKUP-DATA-AND-BLOB模式下，表空间中BLOB数据的更改会记录到日志文件中。

注意 *把表空间更改为只读表空间后，您需要执行一次新的完整备份，因为差异备份不备份只读表空间里的数据文件。*

文件对象的备份模式

在数据库中除了可以对普通数据和BLOB数据进行备份之外，用户还可以选择备份文件对象的模式。用户只能在启动备份后台服务进行完整备份的过程中备份文件对象。在备份之前，用户应该先启动备份服务器，设置完整备份的时间计划和备份目录。要想获得更多有关设置完整备份的信息，请参考第15.6章*备份服务器*。

DBMaster中定义了两种文件对象：用户文件对象和系统文件对象，数据库管理员可以对此作选择。在**dmconfig.ini**配置文件中设定**DB_BkFoM**关键字来指定文件对象的备份模式。

- **DB_BkFoM** = 0：不备份文件对象。
- **DB_BkFoM** = 1：只备份系统文件对象。
- **DB_BkFoM** = 2：备份系统和用户文件对象。

当用户备份文件对象时（**DB_BkFoM** = 1，2），备份服务器把文件对象的外部文件复制到**DB_BkDir**关键字设定目录的“FO”子目录中。通过**DB_FBkTm**和**DB_FBkTv**关键字来指定完整备份的时间计划。

➲ **示例**

以下是**dmconfig.ini**配置文件的一部分，其中包含以下关键字：

```
[MyDB]
DB_BkSvr = 1                        ; starts the backup server
DB_FBkTm = 01/05/01 00:00:00        ; begins from midnight at May 1, 2001.
DB_FBkTv = 1-00:00:00               ; interval is every one day.
DB_BkDir = /home/DBMaster/backup    ; backup directory
DB_BkFoM = 2                        ; backup both system and user file objects
```

因为文件对象的备份模式为2。所以，备份服务器将把所有外部数据库的文件复制到此“***/home/DBMaster/backup/FO***” 目录下。如果**FO**子目录不存在，那么备份服务器将自动创建它。

FO 子目录下的文件会以一个有序数来重新命名。例如：如果源外部文件名为：“**/DBMaster/mydb/FO/ZZ000123.bmp**”，那么备份服务器将会把此文件复制到**FO**子目录下，并且重命名为：**fo0000000344.bak**，也就是说，它是第344个文件对象。源文件名和新文件名都会记录到文件对象映射文件**dmFoMap.his**中。要想获取更多有关文件对象映射文件的信息，请参考第15.7章 *备份历史文件*。

当通过**DB_BkOdr**参数指定旧的备份目录时，备份服务也会将先前版本的文件对象移动到**FO**子目录下。

数据库管理员应该考虑到当执行完整备份时，启用文件对象的备份需要占用更多的时间。一个完整备份包括：（1） 如果设置了**DB_BkOdr**参数，将复制先前的完整备份；（2） 复制所有数据库文件；（3） 复制所有日志文件；（4） 如果设置了**DB_BkFoM**参数，将复制所有文件对象。为了避免备份失败，请确保备份目录所在的磁盘有足够空间可以利用，备份文件的目录可通过**DB_BkDir**关键字来指定。

存储过程的备份模式

DBMaster定义了三种类型的存储过程：SQL存储过程、ESQL存储过程以及JAVA存储过程。因为存储过程的源代码都将写入数据库，所以在执行完整备份期间，SQL存储过程都将以普通数据的方式存储在数据库中。在数据库中除了可以对普通数据、BLOB数据以及文件对象进行备份之外，用户还可以选择备份ESQL存储过程或JAVA存储过程。用户只能在启动后台服务进行完整备份的过程中备份ESQL存储过程和JAVA存储过程。在执行备份之前，用户应该先启动备份服务器，设置完整备份的时间计划和备份目录。要想获得更多有关设置完整备份的信息，请参考第15.6章*备份服务器*。

用户可以通过设置**DB_BkSPm**来指定存储过程的备份模式。

- **DB_BkSPm** = 0：不备份ESQL存储过程和JAVA存储过程。
- **DB_BkSPm** = 1：备份ESQL存储过程和JAVA存储过程。

➲ 示例

以下是**dmconfig.ini**配置文件的一部分，其中包含以下关键字：

```
[MyDB]
DB_BkSvr = 1                        ; starts the backup server
DB_FBkTm = 14/05/01 00:00:00        ; begins from midnight at May 1, 2014
DB_FBkTv = 1-00:00:00               ; interval is every one day
DB_BkDir = /home/dbmaster/backup    ; backup directory
DB_BkSPm = 1                        ; backup ESQL stored procedures and JAVA
stored procedures
```

ESQL存储过程和JAVA存储过程备份文件的默认目录为关键字**DB_BkDir**指定目录下的名为**SP**的子目录。如果用户已经在配置文件**dmconfig.ini**中设置了关键字**DB_BkOdr**，那么在进行完整备份的过程中，ESQL存储过程和JAVA存储过程的备份序列将被移动到名为SP（位于**DB_BkOdr**所指定的旧备份目录下）的子目录下，之后这些备份序列将会从存储过程的默认目录中删除。

在备份ESQL存储过程和JAVA存储过程期间，备份服务器首先会生成一个名为*dmSpBk.his*的文件，之后复制存储过程的文件，同时创建使用了文件扩展名.s0和.b0的文件，该文件用来载入已备份的存储过程。对于

ESQL存储过程，需要备份的文件为源文件和对象文件；对于JAVA存储过程，需要备份的文件仅为对象文件。

为了方便重建存储过程，ESQL存储过程的源文件将以spnameowner.ec格式重命名，。例如，假定ESQL存储过程的名称为**y1**，拥有者为SYSADM，则复制的源文件将被重命名为**y1SYSADM.ec**。

文件**dmSpBk.his**用来记录备份信息。有关更多存储过程备份列表文件的信息，请参考15.7章*备份历史文件*。

数据库管理员应该考虑到当执行完整备份时，启用文件对象的备份会需要占用更多的时间。一个完整备份包括：（1） 如果设置了**DB_BkOdr**，将复制先前的完整备份；（2） 复制所有数据库文件；（3） 复制所有日志文件；（4） 如果设置了**DB_BkFoM**，将复制所有文件对象；（5）如果设置了**DB_BkSPm**，将复制ESQL存储过程和JAVA存储过程。为了避免备份失败，请确保备份目录所在的磁盘有足够空间可以利用，备份文件的目录可通过**DB_BkDir**关键字来指定。

压缩备份文件

数据库文件有可能会变得非常大并且需要大量的可用空间来存储备份文件。如今，DBMaster支持压缩备份文件功能。您可以在**dmconfig.ini**配置文件中设置**DB_BkZip**关键字来启用或禁止该功能。若数据库正在运行，您可以使用系统存储过程**SetSystemOption**更改BKZIP来启用或禁止该功能。

- **DB_BkZip = 1**：压缩备份文件。
- **DB_BkZip = 0**：不压缩备份文件。（默认）

压缩文件的格式为GZIP，您可以使用任何一种GZIP兼容工具来读取压缩文件。

注意 *FO文件不能被压缩，即使您将**DB_BkZip**关键字设置为启用压缩备份文件模式。*

设置备份模式

DBMaster提供了几种不同的设置备份模式的方法，您可以利用直接更改配置文件的方式，或使用dmSQL命令行工具和JServer Manager图形化工具，依据您的数据库是否在线来选择其中一个方法。

更改数据库的备份模式以提高备份保护（例如：从NONBACKUP模式到BACKUP-DATA模式，或从BACKUP-DATA模式到BACKUP-DATA-AND-BLOB模式），对使用的日志文件会产生影响。在更改备份模式之前，日志文件将记录先前没有记录的信息。当您更改备份模式时，必须执行一个完整备份或差异备份，它可以在恢复的过程中为备份日志文件的更新提供一个起点。

当更改数据库的备份模式以降低备份保护时（例如：从BACKUP-DATA模式或BACKUP-DATA-AND-BLOB模式转换到NONBACKUP模式），无需额外的步骤，只是日志文件停止记录更改信息。在恢复的过程中，DBMaster将上一次的完整备份或差异备份作为更新日志文件的起点，但是这样会丢失一些更新数据。

数据库管理员可以在离线的状态下，使用**dmconfig.ini**配置文件或JServer Manager工具来更改数据库的备份模式，因为备份模式的更改会影响日志文件的使用，所以在使用新备份模式启动数据库之前，必须执行离线完整备份。备份模式可以在离线条件下不受约束，从一种备份模式更改成另一种备份模式。当用户从低备份保护转换到高备份保护时需要执行一个完整备份。有关更多离线完整备份的信息，请查看后面的*离线完整备份*章节。

数据库管理员可以在在线的状态下，使用dmSQL工具来更改数据库的备份模式。同时备份模式的更改会影响日志文件的使用，在启动和结束完整备份期间，备份模式可以从低备份保护转换到高备份保护（例如：从NONBACKUP模式转换到BACKUP-DATA模式，或者从BACKUP-DAT A模式转换到BACKUP-DATA-AND-BLOB模式）。运行期间，用户不能直接将备份模式从NONBACKUP模式转换到BACKUP-DATA-AND-BLOB模式，或从BACKUP-DATA-AND-BLOB模式转换到BACKUP-DATA模式。然而，用户可随时将备份模式从BACKUP-DATA-AND-BLOB模式转换到NONBACKUP模式。

➲ **示例**

通过dmSQL来在线更改数据库的备份模式：

```
dmSQL> BEGIN BACKUP;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP DATAFILE;
dmSQL> END BACKUP JOURNAL;
```

或者：

```
dmSQL> BEGIN BACKUP;
dmSQL> END BACKUP DATAFILE;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP JOURNAL;
```

DBMaster不允许将数据库的备份模式调低，除非先将数据库的备份模式更改为NONBACKUP模式。例如：为了将数据库的备份模式BACKUP-DATA-AND-BLOB更改成BACKUP-DATA，用户必须先将备份模式设置为 NONBACKUP，然后根据上述规则将备份模式调高。备份模式可以从BACKUP-DATA-AND-BLOB或BACKUP-DATA 随时更改为NONBACKUP，此动作无需在完整备份开始和结束之间执行。

使用dmconfig.ini配置文件来设置数据库的备份模式

如果数据库处于离线状态，您可以使用**dmconfig.ini**配置文件中的**DB_BMode**关键字来直接更改数据库的备份模式。在下次启动数据库时，就会自动使用新的备份模式。如果数据库处于在线状态，当您更改**DB_BMode**关键字后，只有在重启数据库后**DB_BMode**关键字的更改才会生效。如果您将备份模式从NONBACKUP模式更改到BACKUP-DATA或BACKUP-DATA-AND-BLOB模式，或者从BACKUP-DATA模式更改到BACKUP-DATA-AND-BLOB模式，请您务必在此之前执行一个离线完整备份。

➲ **通过配置文件设置备份模式：**

1. 通过任一种**ASCII**文本编辑器，打开**dmconfig.ini**配置文件。
2. 找到您想更改的数据库配置。
3. 将**DB_BMode**值更改为以下任一种：

```
0 - NONBACKUP mode

1 - BACKUP-DATA mode

2 - BACKUP-DATA-AND-BLOB mode
```

4. 重新启动数据库，DBMaster将使用新的备份模式来运行。

如果数据库的配置中没有**DB_BMode**关键字，那么您可以将此关键字添加到当前数据库配置的任一行中，关键字出现的顺序是无关紧要的；如果您没有指定**DB_BMode**的值，数据库的默认设置为0（NONBACKUP模式）。

使用dmSQL设置备份模式

如果数据库处于在线状态，同时您也想使用dmSQL命令行工具，您可以在dmSQL中输入SQL SET命令来更改数据库的备份模式。但必须在进行在线完整备份或差异备份期间执行此命令。当执行此命令时，新的备份模式将被启动。

➲ **通过dmSQL命令行工具来设置备份模式：**

1. 通过dmSQL连接数据库，以更改数据库的备份模式。
2. 利用**BEGIN BACKUP**命令来启动在线完整备份。
3. 在完整备份期间，通过以下**SET**命令之一更改备份模式：

```
dmSQL> SET BACKUP OFF;

dmSQL> SET DATA BACKUP ON;

dmSQL> SET BLOB BACKUP ON;
```

4. 结束在线完整备份。

其中的SET BACKUP OFF命令表示NONBACKUP模式；SET DATA BACKUP ON 命令表示BACKUP-DATA模式；SET BLOB BACKUP ON命令表示BACKUP-DATA-AND-BLOB模式。

使用JServer Manager更改数据库的备份模式

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来更改数据库的备份模式。JServer Manager可以自动更改**dmconfig.ini**配置文件中的**DB_BMode**关键字。在您下次启动数据库时，新的备份模式将会启动。如果数据库处于在线状态，只有在您重启数据库后，**DB_BMode**关键字的更改才会生效。如果您将数据库的备份模式从NONBACKUP更改到 BACKUP-DATA或BACKUP-DATA-AND-BLOB模式，或者将BACKUP-DATA模式更改到BACKUP-DATA-AND-BLOB模式，请您务必在此之前执行一个离线完整备份。要想获得更多的有关如何使用JServer Manager工具来设置离线备份模式的内容，请参考*服务器管理工具用户手册*。

15.5 离线完整备份

用户可以使用操作系统命令来对数据库进行离线完整备份。当然，DBMaster也提供了这个功能，我们还是建议您使用备份服务器。在执行离线完整备份时，您必须关闭数据库。此外，管理备份序列也是一个复杂的过程。

为了执行离线完整备份，您必须拥有数据库文件的读权限和备份目录的写权限。如果您在执行备份时不得不先关掉数据库，那么您必须拥有数据库的DBA、SYSDBA权限或SYSADM权限。

您可以在不考虑备份模式的情况下执行一个离线完整备份。使用了离线完整备份，您可以将数据库恢复到数据库关闭时的状态。

请注意，使用JServer Manager工具执行离线完整备份时，无法备份文件对象，文件对象必须被手动地复制。如果要从离线完整备份中恢复一个数据库，那么一定要确定文件和目录结构的准确性。有关如何使用JServer Manager执行离线完整备份的信息，请参考*服务器管理工具用户手册*。

使用dmSQL进行离线完整备份

➲ **使用dmSQL执行离线完整备份：**

1. 通知所有用户，数据库将在某时间关闭，请他们在此之前断开连接。
2. 如果数据库正在运行，请用TERMINATE DB命令关闭数据库。如果关闭数据库时出现任何错误，则重启数据库并修正此错误后再次关闭它。
3. 检查**dmconfig.ini**文件，列出所有与备份有关的文件和目录，包括文件对象目录。
4. 使用操作系统命令或工具将数据库文件复制到备份目录或备份设备下，包括数据文件、日志文件、文件对象和**dmconfig.ini**文件。

使用JServer Manager进行离线完整备份

当数据库离线时，您可以使用JServer Manager图形化工具来进行离线完整备份。请注意使用JServer Manager工具进行离线完整备份时，不备份文件对象，需要手动复制文件对象。如果从离线完整备份来恢复数据库，则需确保已完整复制了文件和目录架构。对于在离线状态下如何使用JServer Manager工具来进行完整备份，请参考*服务器管理工具用户手册*。

15.6 备份服务器

尽管DBMaster提供了一个手动备份数据库的方法，但您仍需了解备份规则。为了解决这个问题，DBMaster提供了一个简单易用的办法，即用备份服务器来自动执行在线完整备份、差异备份和增量备份。

注意 *备份服务器只能执行在线备份，因为只有数据库启动后，备份服务器才能启动。*

备份服务器运行在后台，并且根据时间计划或日志文件满来执行在线完整备份、差异备份和增量备份。因为备份服务器和数据库服务器存在联系，可以灵活确定何时会执行一个备份，什么类型的增量备份被执行，哪一个备份选项可以被更改。数据库服务器在启动的同时，备份服务器也被启动，并且一直运行直到您终止备份服务器或关闭数据库服务器。

当执行完整备份时，备份服务器会把最后一次的完整备份从备份目录中复制到旧目录中。然后将所有的数据库文件包括日志文件和 **dmconfig.ini** 配置文件复制到备份目录中，覆盖先前的完整备份。

当执行差异备份时，由于日志文件的更改过于频繁，所以备份服务器仅复制数据文件（DB和BB），不复制日志文件，仅复制有用的日志块。另外，执行差异备份时，也不复制只读表空间里的数据文件。

当执行增量备份时，备份服务器会将所需的日志文件复制到备份目录中。

您可以选择配置备份服务器的选项来设置备份文件的文件名格式、备份路径的位置、旧目录的位置、备份服务器执行备份的时间计划、完整备份后执行差异备份的时间间隔以及可保留的差异备份的最大数目、备份服务器执行增量备份前的填充日志数以及备份服务器保存备份文件的方式。

在dmSQL运行期间，可以使用**SetSystemOption**存储过程来进行备份设置，即在数据库运行期间，您可以使用**SetSystemOption**存储过程更改BKSVR、BKDIR、BKITV、DBKTV、BKTIM、BKFUL、BKFOM、

BKZIP、BKCMP、BKRTS、BKCHK、FBKTM、FBKTV、DBKMX、BKODR、BKFRM。

启动备份服务器

备份服务器是后台服务程序，其生命周期和数据库服务器的生命周期一样长。设置完**DB_BkSvr**关键字后，您无需启动备份服务器。因为DBMaster在启动数据库的同时，将自动启动备份服务器。默认值是禁止使用备份服务器。仅当数据库以多用户模式启动时，备份服务器才能启动。

备份服务器有两种状态：激活和不激活。您可以使用**DB_BkSvr**控制备份服务器状态。**DB_BkSvr**值设置为0，备份服务器处于不激活状态，此时备份服务器不响应任何备份请求，即备份服务器不执行任何备份；**DB_BkSvr**值设置为1，备份服务器处于激活状态，此时备份服务器响应一系列备份请求，您可以执行任何备份。

激活备份服务器有三种方法：在**dmconfig.ini**文件中将**DB_BkSvr**值设置为1；数据库启动后通过*call setsystemoption('bksvr','1')*命令将**DB_BkSvr**值更改为1；数据库运行时使用Jserver Manager中的**动态设置**更改备份设置。

使用备份服务器执行备份前，您需要设置一些相关参数。例如：备份目录、压缩备份模式等。

设置相关参数方法如下：

- 启动数据库前，您可以在**dmconfig.ini**文件中设置相关参数。下次启动数据库时，备份服务器将使用这些关键字初始化相关参数。
- 若数据库已经启动，您可以使用Jserver Manager中的**动态设置**更改参数值。此外，您还可以使用命令*call SetSystemOption('option_name','value')*。请注意个别参数只能使用set语法设置，例如设置备份为OFF、设置数据备份为ON、设置BLOB备份为ON。

备份服务器处于激活状态且在**dmconfig.ini**文件中设置适当备份参数后，您可调用系统存储过程**SetSystemOption**开始备份，该存储过程适用于任何客户端工具或用户应用程序。

➲ **示例**

下例语法用于执行在线完整、差异和增量备份：

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1'); //执行完整备份
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2'); //执行增量备份
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','3'); //执行差异备份
```

使用dmconfig.ini启动备份服务器

如果数据库处于离线状态，您可以通过在**dmconfig.ini**配置文件中设置**DB_BkSvr**关键字来直接启动备份服务器。 当您再次启动数据库时，备份服务器也将被同时启动。如果数据库处于在线状态，只有在您重新启动数据库时，对**DB_BkSvr**关键字的更改才会生效。

➲ **通过设置dmconfig.ini配置文件启动备份服务器：**

1. 用任何文本编辑软件打开**dmconfig.ini** 文件。
2. 找到要启动备份服务器的数据库配置项。
3. 确保备份模式为BACKUP-DATA或BACKUP-DATA-AND-BLOB。将**DB_BMode**设为1是BACKUP-DATA模式，**DB_BMode**设为2是BACKUP-DATA-AND-BLOB模式。
4. 将关键字**DB_BkSvr**的值更改为1，启动备份服务器。
5. 重启数据库，开始备份服务。

使用dmSQL启动备份服务器

如下所示，当数据库在线时，使用dmSQL命令行工具可动态地启动备份服务器。

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

您可以使用*Call SetSystemOption('BkSvr', '1')*更改BkSvr。如果您想在更改BkSvr的同时更改**dmconfig.ini**配置文件中关键字**DB_BkSvr**的值，您可以使用*Call SetSystemOptionW('option','value')*。

启动备份服务器并在**dmconfig.ini**配置文件中设置合适的备份参数后，您可通过调用系统存储过程**SetSystemOption**执行备份，该备份适用于任何客户端工具和用户应用程序。

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP' , '1');   //执行完整备份
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP' , '2');   //执行增量备份
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP' , '3');   //执行差异备份
```

增量备份的时间间隔可通过下面所示语法更改：

```
dmSQL> CALL SETSYSTEMOPTION('bkitv', 'Interval');
```

使用JServer Manager工具启动备份服务器

无论数据库是否在线，您都可以使用JServer Manager图形化工具来启动备份服务器。JServer Manager将自动更改**dmconfig.ini**配置文件中的关键字**DB_BkSvr**。若数据库处于离线状态，则下次数据库启动时，备份服务器也将启动。对于在离线状态下如何使用JServer Manager工具来启动备份服务器，请参考*服务器管理工具用户手册*。

设置差异备份文件格式

差异备份文件名格式如下：

DTimeStamp_DataFileName.dif(2)

D — 必需的差异备份标志

TimeStamp — 距1970年1月的时间 （00:00:00 GMT）

DataFileName — 数据文件的数据库名称

.dif — 差异备份文件的文件扩展名。对于一个特定的完整备份，如果相应的差异备份源文件不存在，那么该差异备份文件的扩展名必须是**.dif2**.

示例：假设2009/12/01 14:11执行第一个差异备份, 那么生成的差异备份文件名如下：

D1259647860_DBNAME.BB.dif、D1259647860_DBNAME.DB.dif、D1259647860_DBNAME.SBB.dif 和 D1259647860_DBNAME.SDB.dif。日志文件名是 D1259647860_DBNAME.JNL。

设置增量备份文件格式

备份文件名格式为***<I><TimeStamp><_><DB_BkFrm>***，其长度不能超过256个字符，例如，I1234567890_%2F%4N%4B.JNL。timestamp是系统的10位数字数据的有效时间。**<DB_BkFrm>**可以包括固定文字部份和格式字符串（转义字符）。

增量备份文件名至少由三部分字符串组成：完整备份ID、数据库名称和备份标识符。在备份序列中命名增量文件时，备份服务器会分配一个完整的备份ID。在恢复数据库的过程中，DBMaster会使用这个完整备份ID去正确重建备份序列。数据库名称用以识别此增量备份文件属于哪个数据库，备份标识符确定此增量备份在备份序列中的相对位置。

格式字符串包括三个部分：转义字符、数值长度和格式字符。以下为有效的格式字符串。

%[*x*]F — 完整备份ID，其中的变量*x*可有以下四种格式：

1：完整备份id的格式为YYYYMMDD，例如：20010917

2：完整备份id的格式为MMDD，例如：0917

3：完整备份id的格式为MMDDhhmm，例如：09171305

4：完整备份id的格式为DDhhmmss，例如：17130558

%[*n*]B — 备份标识符。

%[*n*]N — 日志文件所属的数据库名。

转义字符表示此处是格式字符串的起点，由%表示。如果在您的文件名中希望包括％字符，您必须使用两个%（例如：%%）来表示，在%字符后面必须是单个数字或上述有效的格式字符。如果是其它任何字符，那么备份文件名的格式会视为无效。同时，DBMaster也会返回一个出错信息。

长度字符n是一个1到9之间的整数，表示由此格式字符串所产生的字符串长度。当格式字符串所产生的字符串长度小于此长度时，将会补0。对％【ｎ】Ｎ产生的字符串，0会补在右面，其它的字符串则补在左面。同样，字符串若超过这个长度，会被截断。对％【ｎ】Ｎ所产生的字符串

会从右边截断，而其它格式所产生的字符串会从左边截断。【 n 】是可选项，也就是说在格式字符串中，您可以不必设定长度，备份服务器会使用格式字符所产生的完整字符串长度。

格式字符用以代表特殊字符串，合法的格式字符为F、B或N。除此之外，都是错误的字符串格式。DBMaster会返回一个出错信息，没有转义字符或不是转义字符加单个数字的有效格式字符都当作文本常量。

Date和time型数值是从系统中获得的，如果系统的时间和日期是正确的，那么此类型的值也是正确的。这个值代表了备份序列中的日志文件的顺序，DBMaster可以通过备份服务器为每一个日志文件自动提供这个值。

DBMaster提供了几种不同的设置备份文件名格式的方法。 您可以利用直接更改配置文件的方式或者使用JServer Manager图形化工具来设定备份日志文件的文件名格式。

使用dmconfig.ini配置文件设置备份文件名格式

如果数据库处于离线状态，您可以通过**dmconfig.ini**配置文件中的**DB_BkFrm**关键字来设置备份文件名格式。当您再次启动数据库时，备份服务器会将此备份文件名格式应用到所有的备份日志文件中。如果数据库处于在线状态，只有重新启动数据库后，对**DB_BkFrm**关键字的更改才会生效。

➲ **使用dmconfig.ini配置文件设置备份文件格式：**

1. 用任何文本编辑软件打开**dmconfig.ini**文件。
2. 找到要设置的数据库配置项。
3. 将**DB_BkFrm**关键字更改为用于备份文件名格式的字符串。

   注意 *此字符串可以包括任何有效的格式序列和文本常量，但文件名总长度不能超过256个字符。*
4. 重启数据库，开始使用新的备份文件名格式。

使用dmSQL设置备份文件名格式

数据库运行期间，可通过存储过程**SetSystemOption**更改备份文件名格式，语法如下所示：

```
CALL SETSYSTEMOPTION('bkfrm', 'name')
```

➲ 示例

在dmSQL命令行中输入以下命令，将备份文件名称更改为I1234567890_%2F%4N%4B.JNL。

```
dmSQL> CALL SETSYSTEMOPTION('bkfrm', 'I1234567890_%2F%4N%4B.JNL');
```

使用JServer Manager工具设置备份文件名格式

无论您的数据库处于离线还是在线状态，都可以使用JServer Manager图形化工具来设置备份服务器的备份文件名格式，JServer Manager将自动更改**dmconfig.ini**配置文件中的**DB_BkFrm**关键字。当您启动数据库时，备份服务器会将这些备份文件名格式应用到所有的备份日志文件中。要想获得更多的有关如何使用JServer Manager工具来设置备份文件名格式的内容，请参考*服务器管理工具用户手册*。

备份目录

备份目录用于指定备份文件的存放位置，DBMaster支持单备份文件路径和多备份文件路径。备份服务器将自动创建BkDir。您最好不要在存放数据库文件的磁盘中创建备份目录，这样可以避免在介质损坏时，发生数据库和备份文件都丢失。

备份目录可以通过**dmconfig.ini**配置文件中的**DB_BkDir**关键字来设定，此关键字的值可以包含备份目录的一个完整路径或相对路径。如果您没有指定备份目录，备份服务器将在数据库目录下自动创建一个名为**backup**的默认目录。数据库目录可以通过**dmconfig.ini**配置文件中的**DB_DbDir**关键字来设定，备份路径的总长度不能超过256个字符。

当数据库运行在复制模式（主或从数据库）时，BKDIR需设置为单路径。若将BKDIR设置为多路径，则仅第一个路径可用且忽略路径大小。此外，如果一个目录中存在几个数据库，那么最好不要使用默认目录，因为一个数据库的备份历史信息会覆盖或追加到另一个数据库中。为了

避免这种情况的发生，您可以为每一个数据库创建不同的路径，或者为每个数据库明确的指定一个备份路径。将不同的数据库存放于不同的目录中，是一个更可取的办法，因为这样可以正确的查看到哪些文件属于哪个数据库。

DBMaster提供了几种不同的设置备份目录的方式，您可以利用直接更改配置文件的方式或使用JServer Manager图形化工具，依据您的数据库是否在线来选择其中的一个方式。

使用dmconfig.ini配置文件来设置备份目录

如果数据库处于离线状态，可以直接在**dmconfig.ini**配置文件中设置**DB_BkDir**关键字来创建备份目录。当您再次启动数据库时，备份服务器会使用您设定的这个备份目录。如果数据库处于在线状态，只有在您重启数据库后，对**DB_BkDir**关键字的更改才会生效。

➲ **通过dmconfig.ini配置文件设置数据库的备份目录：**

1. 通过任何一种文本编辑器，打开数据库服务器端的**dmconfig.ini**文件。
2. 找到要设置的数据库配置项。
3. 将**DB_BkDir**关键字更改为包含一个已存在目录的字符串，用来设置备份目录。
4. 重新启动数据库，使用新的备份目录。

使用dmSQL在线设置备份目录

数据库运行期间，可通过存储过程**SetSystemOption**更改备份目录，语法如下所示：

```
CALL SETSYSTEMOPTION('bkdir', 'path')
```

*path*为新的备份目录全路径，字符串长度不得超过256个字符。

➲ **示例**

执行下面的dmSQL命令，改变备份路径到*“E:\storage\database\backup\WebDB”*。

```
dmSQL> CALL SETSYSTEMOPTION('bkdir', 'E:\storage\database\backup\WebDB');
```

使用JServer Manager工具设置备份目录

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来设置离线备份目录。JServer Manager将自动更改**dmconfig.ini**配置文件中的**DB_BkDir**关键字。当您启动数据库时，备份服务器会将此目录作为备份目录。如果数据库处于在线状态，JServer Manager工具可以设置是使用动态设置立即更改备份目录，还是当数据库做交互式备份时延迟到重启数据库时更改备份目录。无论您的数据库处于何种状态，JServer Manager工具都将在新的备份目录中复制历史备份文件。

设置多个备份路径

DBMaster可以为用户提供多个备份文件路径，该功能用于当用户将文件保存至备份目录中，但是该目录不能为完整备份提供足够的存储空间时。如果该选项已设置，那么DBMaster会将剩余的数据备份至第二个备份目录中，于是备份就可以顺利地执行下去。用户也可以使用多个备份路径来执行完整、差异备份或增量备份。当用户使用多备份文件路径来备份文件时，DBMaster须遵循以下约定：

- 当备份文件时，数据库系统会设法将每个文件逐个存储到备份目录中。例如，当将文件存储至备份目录1时，如果目录1没有足够的空间存储该文件，那么该文件将被保存到备份目录2中，以此类推。如果所有的备份目录都已填满，那么系统将返回一条错误信息。
- 当在从数据库中备份文件时，只能使用一个备份目录。
- FO必须备份在第一个备份目录中。
- 备份路径的最大数量为32。

➲ **示例**

DBMaster依据以下结构来设置多备份路径：

```
DB_BkDir = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…

< BKDIR n > : the n's backup path
< SIZE n > : the size of the n's backup path
```

当为数据库**DB1**设置多备份路径时，你需要在配置文件中设置**DB_BkDir**关键字指明备份路径。

```
DB_BkDir = /home/usr/DBMaster/bk 5000 /home2/backup 1000
```

当目录*/home/usr/DBMaster/bk*下的可用存储空间都已填满时，数据库会将文件备份至*/home2/backup*目录下。

设置旧文件目录

旧目录是一个或一组用于保存最后一次备份之前的备份的目录（最多32个），为了防止数据库和备份文件在发生介质故障时都丢失，您最好不要在存放数据库文件的磁盘中创建旧文件目录。

旧目录可以通过**dmconfig.ini**配置文件中的**DB_BkOdr**关键字来设定，如果您没有指定它，备份服务器将丢弃先前的备份序列。

使用dmconfig.ini配置文件设置旧文件目录

您可以直接更改**dmconfig.ini**配置文件中的**DB_BkOdr**关键字来设置旧文件目录。当您启动数据库时，备份服务器将把此目录作为旧文件目录。如果数据库处于在线状态。只有在您重新启动数据库时，对**DB_BkOdr**关键字的更改才会生效。

使用dmSQL在线设置设置旧文件目录

数据库运行期间，可通过存储过程**SetSystemOption**更改旧文件备份目录，语法如下所示：

```
CALL SETSYSTEMOPTION('bkodr', 'path')
```

*path*为新的旧文件备份目录全路径，字符串长度不得超过256个字符。

➲ **示例**

在dmSQL命令行中输入以下命令，将旧文件备份路径更改为"*E:\storage\database\backup\WebDB*"。

```
dmSQL> CALL SETSYSTEMOPTION('bkodr', 'E:\storage\database\backup\WebDB');
```

使用JServer Manager设置旧文件目录

如果数据库处于离线状态，您可以通过JServer Manager图形化工具来设置旧文件目录的位置，JServer Manager工具将自动更改**dmconfig.ini**配置文件中的**DB_BkOdr**关键字。当您再次启动数据库时，备份服务器会将此目录作为备份目录。如果数据库处于在线状态，在您重新启动数据库时，JServer Manager工具可以设置是立即更改旧备份目录还是延迟到重新启动数据库时更改旧备份目录。想获得有关在离线状态下使用JServer Manager设置旧备份目录的详细信息，请参考*服务器管理工具用户手册*。

设置差异备份

差异备份的时间计划用于指定备份服务器执行一个在线增量备份的时间。 此时间计划由两部分组成：初始备份时间和间隔时间。初始备份时间确定了备份服务器执行第一次差异备份的日期和时间；间隔时间确定了两次差异备份相隔的时间。

初始完整备份时间可通过**dmconfig.ini**配置文件中的**DB_FBkTm**关键字来设置，它的格式为：YY/MM/DD HH:MM:SS。初始备份时间不存在默认值，但 如果您使用JServer Manager工具来设置备份服务器，JServer Manager工具会为您提供一个默认值并且将此值写入**dmconfig.ini**配置文件中。

间隔时间可通过**dmconfig.ini**配置文件中的**DB_DBkTv**关键字来设置，差异备份第一次执行的时间是**DB_FBkTm + DB_DBkTv**。时间间隔**DB_DBkTv**的值的格式为：D-HH:MM:SS。间隔时间不存在默认值，但如果您使用JServer Manager工具来设置备份服务器，JServer Manager工具会为您提供一个1-00:00:00默认值，并且将此值写入**dmconfig.ini**配置文件中。

最后，执行完整备份后，可保留的差异备数的最大值由关键字**DB_DbKmx**指定，当差异备份的数目超过关键字**DB_DbKmx**的值时，备份服务器会删除最早的那个差异备份。若数据库正在运行，您可以使用存储过程SetSystemOption更改DBKMX。此外，在数据库运行期间，

您可以通过存储过程**SetSystemOption**设置是否在执行差异备份前检查数据库。

使用dmconfig.ini配置文件更改差异备份设置

如果数据库处于离线状态，您可以直接使用**dmconfig.ini**配置文件中的**DB_FBkTm**和**DB_DBkTv**关键字来设置备份的时间计划。当您再次启动数据库时，备份服务器将为差异备份的时间计划应用这些设置。如果数据库处于在线状态，只有在您重新启动数据库后，对**DB_FBkTm**和**DB_DBkTv**关键字的更改才会生效。

➲ **通过dmconfig.ini设置备份时间计划：**

1. 使用任何一种文本编辑器，打开数据库服务器上的**dmconfig.ini**文件。
2. 找到要设置备份时间计划的数据库配置项。
3. 改变**DB_FBkTm**关键字为YY/MM/DD HH:MM:SS格式的日期时间值。
4. 改变**DB_DBkTv**关键字为DDDDD-HH:MM:SS格式的时间间隔值。
5. 重新启动数据库，开始使用新的备份时间计划。

使用dmSQL更改差异备份的设置

可用存储过程**SetSystemOption**激活备份服务器，命令如下：

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

激活备份服务器后，调用系统存储过程**SetSystemOption**开始进行差异备份：

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP', '3');
```

可用如下语法更改差异备份的时间间隔：

```
CALL SETSYSTEMOPTION('dbktv', 'Interval')
```

使用JServer Manager工具更改差异备份的设置

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来设置差异备份的时间计划。JServer Manager工具可以自动更改

dmconfig.ini配置文件中的关键字**DB_FBkTm**和**DB_DBkTv**的值。在数据库再次启动时，备份服务器会将这些设置作为新的差异备份时间计划。如果数据库处于在线状态，JServer Manager工具可以立即更改备份时间计划，也可以延迟到重新启动数据库时，更改备份时间计划。有关如何使用JServer Manager设置增量备份的时间计划，请参考*服务器管理工具用户手册*。

设置增量备份

增量备份的时间计划用于指定备份服务器执行一个在线增量备份的时间。 此时间计划由两部分组成：初始备份时间和间隔时间。初始备份时间确定了备份服务器执行第一次增量备份的日期和时间；间隔时间确定了两次增量备份相隔的时间。

当数据库处于在线备份时，您可以将增量备份的时间计划和日志触发值结合起来使用。也就是说，备份数据库可以根据时间计划，也可以通过设定的日志文件填充系数自动备份数据库。如果您没有指定增量备份的时间计划，备份服务器将无法依据时间计划来备份数据库。

初始备份时间可通过**dmconfig.ini**配置文件中的**DB_BkTim**关键字来设置，它的格式为：YY/MM/DD HH:MM:SS。初始备份时间不存在默认值。

间隔时间可通过**dmconfig.ini**配置文件中的**DB_BkItv**关键字来设置，它的格式为：D-HH:MM:SS。间隔时间不存在默认值，但如果您使用JServer Manager工具来设置备份服务器，JServer Manager工具会为您提供一个1-00:00:00默认值，并且将此值写入**dmconfig.ini**配置文件中。

DBMaster提供了几种不同的设置增量备份时间计划的方式，您可以利用直接更改配置文件的方式或者使用JServer Manager图形化工具，依据您的数据库是否在线来选择其中的一种方式。

使用dmconfig.ini配置文件更改增量备份设置

如果数据库处于离线状态，您可以直接使用**dmconfig.ini**配置文件中的**DB_BkTim**和**DB_BkItv**关键字来设置备份的时间计划。当您启动数据库时，备份服务器将为增量备份的时间计划应用这些设置。如果数据库处

于在线状态，只有在您重新启动数据库后，对**DB_BkTim**和**DB_BkItv**关键字的更改才会生效。

➲ **通过dmconfig.ini设置备份时间计划：**

1. 使用任何一种文本编辑器，打开数据库服务器上的**dmconfig.ini**文件。
2. 找到要设置备份时间计划的数据库配置项。
3. 改变**DB_BkTim**关键字为YY/MM/DD HH:MM:SS格式的日期时间值。
4. 改变**DB BkItv**关键字为DDDDD-HH:MM:SS格式的时间间隔值。
5. 重新启动数据库，开始使用新的备份时间计划。

使用dmSQL更改增量备份的设置

在数据库运行时，存储过程**SetSystemOption**用于更改增量备份的启动时间和时间间隔。更改增量备份启动时间的语法如下：

```
CALL SETSYSTEMOPTION('bktim', 'StartTime')
```

更改增量备份时间间隔的语法如下：

```
CALL SETSYSTEMOPTION('bkitv', 'Interval')
```

*StartTime*是首次启动增量备份的时间，格式为YY:MM:DD HH:MM:SS. *Interval*是增量备份的时间间隔，格式为D-HH:MM:SS。

激活备份服务器后，调用系统存储过程**SetSystemOption**开始进行增量备份：

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2');
```

➲ **示例**

在dmSQL命令行中输入以下命令，设置增量备份时间间隔为1小时。

```
dmSQL> CALL SETSYSTEMOPTION('bkitv', '0-1:00:00');
```

使用JServer Manager工具更该增量备份的设置

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来设置增量备份的时间计划。JServer Manager工具可以自动更改

dmconfig.ini配置文件中的关键字**DB_BkTim**和**DB_BkItv**的值。在数据库再次启动时，备份服务器会将这些设置作为新的增量备份时间计划。如果数据库处于在线状态，JServer Manager工具可以立即更改备份时间计划，也可以延迟到重新启动数据库时，更改备份时间计划。有关如何使用JServer Manager设置增量备份的时间计划，请参考*服务器管理工具用户手册*。

设置日志触发值

日志触发值指定日志文件的填充系数，当数据库处于在线备份时，您可以将增量备份的时间计划和日志触发值结合起来使用。也就是说，您可以根据时间计划来备份数据库，也可以通过设定日志文件的填充系数来备份数据库。

日志触发值可以通过**dmconfig.ini**配置文件中的**DB_BkFul**关键字来设定，此**DB_BkFul**关键字的值可以是一个50-100之间的整数，也可以设为0。50-100之间的数值表示备份服务器在执行备份之前，日志文件填充的百分比。数值0表示一个日志文件填充完成引起备份服务器去执行一个备份操作。设置数值0和数值100所达到的效果是一样的，因为都是在一个日志文件填充完成（100%填满）后引起一个备份操作。如果您没有指定日志触发的值，备份服务器将使用90默认值。

DBMaster提供了几种不同的设置日志触发值的方式。您可以利用直接更改配置文件的方式或者使用JServer Manager图形化工具，依据您的数据库是否在线来选择其中一个方式。

使用dmconfig.ini 配置文件更改日志触发值

如果数据库处于离线状态，您可以直接使用**dmconfig.ini**配置文件中的**DB_BkFul**关键字来设置日志触发值。当您启动数据库时，备份服务器将为日志触发值应用此设置。如果数据库处于在线状态，只有在您重新启动数据库时，对**DB_BkFul**关键字的更改才会生效。

➲ **通过配置文件dmconfig ini设置日志触发值：**

1. 您可以使用任何一种ASCII文本编辑器打开数据库服务器上的**dmconfig.ini**文件。

2. 找到数据库配置区域来更改日志触发值。
3. 将关键字**DB_BkFul**更改成一个50-100之间的整数，或者将它设为0。
4. 使用新的日志触发值来重新启动数据库。

使用dmSQL更改日志触发值

在数据库运行时，存储过程**SetSystemOption**可用于更改日志触发值。更改日志触发值的语法如下：

```
CALL SETSYSTEMOPTION('bkful', 'n')
```

n的取值范围为0或50-100之间的整数。将n设为0表示日志文件填充满后将引起一个备份操作，将n设为50-100之间的整数表示在激活备份服务器之前，日志文件的填充百分数。

➲ **示例**

执行下面的dmSQL命令，设置日志触发值为75% ：

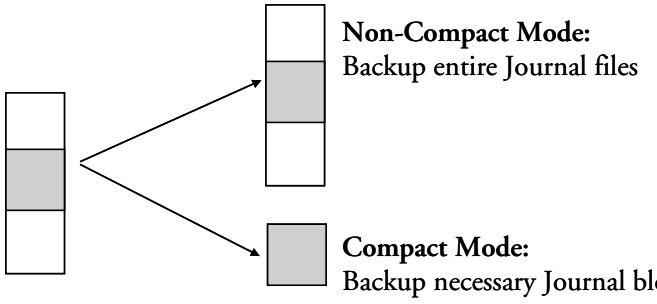
```
dmSQL> CALL SETSYSTEMOPTION('bkful', '75');
```

使用JSERVER MANAGER工具更改日志触发值

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来设置日志触发值。JServer Manager工具可以自动更改**dmconfig.ini**配置文件中的关键字**DB_BkFul**的值。在数据库再次启动时，备份服务器会将此设置作为新的日志触发值。如果数据库处于在线状态，在您重新启动数据库时，JServer Manager工具可以立即更改日志触发值，也可以延迟到重新启动数据库时更改日志触发值。有关如何使用JServer Manager工具来设置日志触发值请参考*服务器管理工具用户手册*。

设置压缩备份模式

执行在线增量备份或差异备份时，压缩备份模式用于确定备份服务器是备份完整日志文件，还是只备份完整日志块。此功能是很有必要的，因为并非每一个日志块都包含恢复一个数据库的数据。因此备份服务器在执行备份时，只需备份必要的日志块。也就是说，设置压缩备份模式之后会节省存储空间。但是在作备份还原时，可能会花费较多时间。



压缩备份模式可以通过**dmconfig.ini**配置文件中的**DB_BkCmp**关键字来设定，此关键字的值可以为0或1。设为1表示启动压缩备份模式，设为0表示关闭压缩备份模式。如果您没有指定压缩备份模式的值，备份服务器将使用默认值1（启动压缩模式）。

DBMaster提供了几种不同的压缩备份模式的方法。您可以利用直接更改配置文件的方式或者使用JServer Manager图形化工具，依据您的数据库是否在线来选择其中一个方式。

使用dmconfig.ini配置文件来设置压缩备份模式

如果数据库处于离线状态，可以直接使用**dmconfig.ini**配置文件中的**DB_BkCmp**关键字来设置备份服务器的压缩备份模式。当您启动数据库时，备份服务器会将此设置应用到压缩备份模式上。如果数据库处于在线状态，只有在您重新启动数据库时，对**DB_BkCmp**关键字的更改才会生效。

➲ **使用dmconfig.ini配置文件设置压缩备份模式：**

1. 通过任一种文本编辑器，打开数据库服务器端的**dmconfig.ini**配置文件。
2. 找到要更改压缩备份模式的数据库配置项。
3. 将关键字**DB_BkCmp**的值设为1，表示压缩备份模式；或者设为0，不压缩备份模式。
4. 重启数据库，开始使用新的日志触发值。

使用dmSQL设置压缩备份模式

在数据库运行时，存储过程**SetSystemOption**可用于更改压缩备份模式。并不是日志文件中的每个日志块都需要进行备份。如果设置关键字**DB_BkCmp**值为1，备份服务器将只备份需要的日志块以节省磁盘空间。更改压缩备份模式的命令如下：

```
dmSQL> CALL SETSYSTEMOPTION('bkcmp', '1');
```

使用JServer Manager工具设置压缩备份模式

如果数据库处于离线状态，您可以使用JServer Manager图形化工具来设置压缩备份模式。JServer Manager工具可以自动更改**dmconfig.ini**配置文件中的关键字**DB_BkCmp**的值。在数据库再次启动时，备份服务器会将此设置作为新的压缩备份模式。如果数据库处于在线状态，在您重新启动数据库时，JServer Manager工具可以立即更改压缩备份模式，也可以延迟到重新启动数据库时更改压缩备份模式。有关如何使用JServer Manager工具来设置压缩备份模式，请参考*服务器管理工具用户手册*。

完整备份的时间计划

完整备份的时间计划用于指定备份服务器执行在线完整备份的时间，此时间计划由两部分组成：初始备份时间和间隔时间。初始备份时间确定了备份服务器执行首次完整备份的日期和时间；间隔时间确定了两次完整备份相隔的时间。

您可以将完整备份和差异备份的时间计划和增量备份结合起来使用。如果您没有指定完整备份的时间计划，备份服务器将不使用时间计划表来执行完整备份。

初始时间可以通过**dmconfig.ini**配置文件中的**DB_FBkTm**关键字来指定。它的格式为：**YY/MM/DD HH:MM:SS**，初始时间不存在默认值。

间隔时间可以通过**dmconfig.ini**配置文件中的**DB_FBkTv**关键字来指定。它的格式为：**D-HH:MM:SS**，间隔时间不存在默认值。

最后，您可以通过关键字**DB_BkChk**定义在执行完整备份和差异备份之前是否检查数据库，通过关键字**DB_BkRTs**定义备份服务器在执行完整备份时，是否备份只读表空间文件。您可以在**dmconfig.ini**配置文件中

设置关键字**DB_BkChk**和**DB_BkRTs**来启用或禁止该功能。若数据库正在运行，您可以使用系统存储过程**SetSystemOption**更改BKCHK和BKRTS来启用或禁止该功能。

使用dmconfig.ini配置文件设置完整备份时间计划

如果数据库处于离线状态，可以直接使用**dmconfig.ini**配置文件中的**DB_FBkTm**和**DB_FBkTv**关键字来设定备份服务器的完整备份时间计划。当您启动数据库时，备份服务器将为完整备份的时间计划应用这些设置。如果数据库处于在线状态，只有在您重新启动数据库时，对**DB_FBkTm**和**DB_FBkTv**的更改才会生效。

➲ **通过配置文件dmconfig.ini设置完整备份的时间计划：**

1. 通过任何一种文本编辑器，打开服务器端的配置文件**dmconfig.ini**。
2. 找到要更改日志触发值的数据库配置项。
3. 改变关键字**DB_FBkTm**为YY/MM/DD HH:MM:SS 格式，关键字**DB_FBkTv**为D-HH:MM:SS格式。
4. 重新启动数据库，开始使用新的完整备份时间计划。

使用dmSQL设置完整备份的时间计划

在数据库运行时，存储过程**SetSystemOption**用于更改完整备份的启动时间和时间间隔。更改完整备份启动时间的语法如下：

```
CALL SETSYSTEMOPTION('fbktm', 'StartTime')
```

更改完整备份时间间隔的语法如下：

```
CALL SETSYSTEMOPTION('fbktv', 'Interval')
```

*StartTime*是首次启动完整备份的时间，格式为YY:MM:DD HH:MM:SS. *Interval*是完整备份的时间间隔，格式为D-HH:MM:SS。

激活备份服务器后，调用系统存储过程**SetSystemOption**开始进行完整备份：

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1');
```

➲ 示例

在dmSQL命令行中输入以下命令，设置完整备份时间间隔为1小时。

```
dmSQL> CALL SETSYSTEMOPTION('fbktv', '0-1:00:00');
```

使用JSERVER MANAGER工具设置完整备份的时间计划

您可以使用JServer Manager工具来设置完整备份的时间计划。JServer Manager将自动更改**dmconfig.ini**配置文件中的**DB_FBkTm**和**DB_FBkTv**关键字。在您启动数据库时，备份服务器会将此设置作为新的完整备份时间计划。要想获得更多有关如何使用JServer Manager工具来设置完整备份的时间计划，请参考*服务器管理工具用户手册*。

文件对象的备份模式

在完整备份期间，数据库管理员可以通过设置文件对象的备份模式来决定备份服务器是否备份文件对象。您可以只备份系统文件对象，也可以备份系统和用户文件对象。

在数据库的启动过程中，关键字**DB_BkFoM**确定了文件对象的备份模式。我们可以使用dmSQL或JServer Manager图形化工具，在数据库的运行期间来更改它。

通过指定**DB_BkOdr**关键字，备份服务器会将先前备份的文件移到旧的备份目录中。

启动文件对象备份模式会引起数据库花更多的时间去执行一个完整备份。一个完整备份包括：（1）如果设置了**DB_BkOdr**关键字，将复制先前的完整备份。（2）复制所有数据库文件。（3）复制所有日志文件。（4）如果设置了**DB_BkFoM**关键字，将复制所有文件对象。为了避免备份失败，请确保备份目录所在的磁盘有足够空间可以利用，备份文件的目录可通过**DB_BkDir**（和**DB_BkOdr**）关键字来指定。

一个完整备份执行时，文件对象复制到在备份目录中创建的FO目录中。当文件对象复制到备份文件的目录中时，它们将被重新命名。**/FO**子目录中的文件名是以FO开头的，后面跟随一个十位数的序列数。所有备份文件对象都是.BAK扩展名。源文件名和路径与备份文件名之间的映射都被记录到映射文件**dmFoMap.his**中。

备份文件对象的映射文件

文件对象的映射文件**dmFoMap.his**创建在"***DB_BkDir/FO***"目录中。它是一个纯粹的ASCII文本文件，用于记录源文件名和备份文件名。格式如下：

```
Database Name: DBSAMPLE5
Begin Backup FO Time: 2013/04/12 09:21:32
FO Backup Directory: C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\
[Mapping List]
s, fo00000000000.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\ZZ000001.bmp"
u, fo00000000001.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\image.jpg"
....
s, fo00000002345.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\ZZ00AB32.txt"
```

" [Mapping List] "之前的内容是为用户参考提供的一个描述。"[Mapping List]" 之后的每一行代表一个记录，用于显示文件对象类型（s = 系统文件对象，u = 用户文件对象），**/fo**子目录中的新文件以及它的源文件名和路径。此映射文件对恢复文件对象是很有必要的。

使用dmconfig.ini配置文件设置文件对象的备份模式

配置文件中的关键字**DB_BkFoM**确定了文件对象的备份模式：

- **DB_BkFoM** = 0：不备份文件对象
- **DB_BkFoM** = 1：只备份系统文件对象
- **DB_BkFoM** = 2：备份系统和用户文件对象

如果**DB_BkFoM** = 1或2，备份服务器将把所有的文件对象复制到备份目录下的**/fo**子目录中。下面的时间计划为完整备份的时间计划。

➲ 示例

在**dmconfig.ini**配置文件中，设定备份文件对象的参数。

```
[MyDB]
DB_BkSvr = 1                        ; starts the backup server
DB_FBkTm = 01/05/01 00:00:00        ; begins at midnight, May 1, 2001.
DB_FBkTv = 1-00:00:00               ; interval is once every day.
DB_BkDir = /home/DBMaster/backup    ; backup directory
DB_BkFoM = 2                        ; backup both system and user file objects
```

因为备份模式设置为2，所以备份服务器将把所有外部文件（用户文件对象）和系统文件对象复制到***/home/DBMaster/backup/FO***目录下。如果**FO**子目录不存在，那么备份服务器将会自动创建它。

使用dmSQL设置文件对象的备份模式

当数据库运行时，存储过程**SetSystemOption**可用于更改文件对象的备份模式。它的语法如下：

```
CALL SETSYSTEMOPTION('bkfom', 'n')
```

n的取值可为0、1、2，将n设为0代表关闭文件对象的备份；将n设为1代表备份服务器在完整备份期间，可以备份所有的系统文件对象；将n设为2代表备份服务器在完整备份期间，可以备份所有的系统和用户文件对象。

➲ **示例**

为了执行一个系统和用户文件对象的完整备份，您可以在dmSQL的命令提示中输入以下命令行来配置备份服务器。

```
dmSQL> CALL SETSYSTEMOPTION('bkfom', '2');
```

使用JServer Manager工具设置文件对象的备份模式

设置**文件对象的备份模式**将影响文件对象在完整备份过程中的复制方式。如果您选择**不备份文件对象模式**，那么文件对象将无法被复制。如果您选择**只备份系统文件对象**模式，那么系统文件对象将在自动完整备份过程中被复制。如果您选择**备份系统和用户文件对象**模式，那么系统文件对象和用户文件对象在自动完整备份过程中都将被复制。如何在数据库运行时设置文件对象的备份模式或在数据库运行期间如何运用JServer Manager工具的**动态设置**来设置文件对象的备份模式，请参考*服务器管理工具用户手册*。

存储过程的备份模式

存储过程的备份模式，有助于数据库管理员决定备份服务器在进行完整备份时是否备份ESQL存储过程和JAVA存储过程。

设置存储过程备份模式的方法有多种。当数据库启动时，可以通过配置文件中的关键字**DB_BkSPm**来设置存储过程的备份模式；在数据库运行期间，还可以通过dmSQL命令或JServer Manager图形化工具来更改其备份模式。

备份服务器可以将之前存储过程的备份移动到由**DB_BkOdr**指定的旧 目录下。

开启存储过程的备份会导致数据库在完整备份时占用更多的时间，这取决于数据库中存在的文件对象数量。一个完整备份包括：（1）如果设置了**DB_BkOdr**，将复制先前的完整备份；（2）复制所有数据库文件；（3）复制所有日志文件；（4）如果设置了**DB_BkFoM**，将复制所有文件对象；（5）如果设置了**DB_BkSPm**，将复制ESQL存储过程和JAVA存储过程。为了避免备份失败，请确保备份目录所在的磁盘有足够空间可以利用，备份文件的目录可通过**DB_BkDir**关键字来指定。

存储过程被复制到执行完整备份时，创建的备份目录的子目录**SP**下。当存储过程被复制到存储过程的备份目录下，存储过程会被重新命名。复制的存储过程备份信息被记录在文件**dmSpBk.his**中。

备份存储过程的信息文件

备份信息由三部分构成：数据库名称、备份时间、用来记录已备份行的备份列表。备份信息文件**dmSpBk.his**位于由关键字**DB_BkDir**所指定目录的子目录**SP**下。该文件是一个纯ASCII码的文本文件，用来记录所复制的存储过程备份信息，其格式如下所示：

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

"[Backup List]"之前的内容仅为供用户参考的说明。"[Backup List]"之后的每一行均为一条记录，显示存储过程的类型、存储过程的拥有者、存储过程的名称、存储过程的对象文件名称以及相应的源文件名称。该备份信息文件是恢复存储过程时的必要文件。

使用dmconfig.ini配置文件设置存储过程的备份模式

配置文件中的关键字**DB_BkSPm**可以设置存储过程的备份模式：

- **DB_BkSPm** = 0：不备份ESQL存储过程和JAVA存储过程
- **DB_BkSPm** = 1：备份ESQL存储过程和JAVA存储过程

➲ **示例**

在**dmconfig.ini**配置文件中，设定备份存储过程的参数。

```
[MyDB]
DB_BkSvr = 1                        ; starts the backup server
DB_FBkTm = 14/05/01 00:00:00        ; begins at midnight, May 1, 2014
DB_FBkTv = 1-00:00:00               ; interval is once every day
DB_BkDir = /home/dbmaker/backup     ; backup directory
DB_BkSPm = 1                        ; backup ESQL stored procedures and JAVA
stored procedures
```

DB_BkSPm的值为1时，备份服务器将备份ESQL存储过程和JAVA存储过程，并保存到由关键字**DB_BkDir**指定目录的子目录**SP**下。如果子目录**SP**不存在，备份服务器会自动创建。

使用dmSQL设置存储过程的备份模式

当数据库运行时，存储过程**SetSystemOption**用于更改存储过程的备份模式。更改存储过程备份模式的语法如下：

```
CALL SETSYSTEMOPTION('BKSPM','n')
```

在上述语法中，*n*的取值可为0或1。如果*n*为0，则表示在进行完整备份时，备份服务器不备份ESQL存储过程和JAVA存储过程；如果*n*为1，则表示备份服务器将备份ESQL存储过程和JAVA存储过程。

➲ **示例**

配置备份服务器，使其在备份所有存储过程时执行完整备份。该操作在dmSQL命令中输入的命令如下所示：

```
dmSQL> CALL SETSYSTEMOPTION('BKSPM','1');
```

使用JServer Manager工具设置存储过程的备份模式

无论数据库是否在线，您都可以使用JServer Manager图形化工具来设置备份ESQL存储过程和JAVA存储过程。JServer Manager将自动更改**dmconfig.ini**配置文件中的关键字**DB_ BkSPm**。若数据库处于离线状态，只有在您重新启动数据库时，新的设置才会起效。对于数据库运行期间，如何设置存储过程的备份模式或如何使用JServer Manager工具的**动态设置**来设置存储过程备份模式的详细内容，请参考*服务器管理工具用户手册*。

终止备份服务器

数据库启动后，备份服务器将自动开启并默认为不激活状态。您可以通过**DB_BkSvr**关键字控制备份服务器的状态。**DB_BkSvr**设置为0，不激活备份服务器，**DB_BkSvr**设置为1，激活备份服务器。如果您想终止备份服务器，您可以在**dmconfig.ini**文件中将**DB_BkSvr**关键字设为0，也可以在数据库启动后使用*call setsystemoption('bksvr','0')*更改BkSvr。

使用dmconfig.ini配置文件终止备份服务器

如果数据库处于离线状态，您可以直接使用**dmconfig.ini**配置文件中的**DB_BkSvr**关键字来终止备份服务器。当您启动数据库时，备份服务器将不会启动；如果数据库处于在线状态，只有在您的数据库被重新启动时，对**DB_BkSvr**关键字的更改才会生效。

➲ **使用dmconfig.ini配置文件停止备份服务器：**

1. 用任何文本编辑软件打开**dmconfig.ini**文件。
2. 找到要改变备份模式的数据库配置项。
3. 改变**DB_BkSvr**关键字为0，关闭备份服务器。
4. 重启数据库。

使用dmSQL终止备份服务器

在数据库运行时，存储过程**SetSystemOption**用于更改备份服务器的状态，更改更改备份服务器状态的语法如下：

```
CALL SETSYSTEMOPTION('bksvr','n')
```

n的取值可为0或1。将n设为0表示不激活备份服务器，将n设为1表示激活备份服务器。

➲ **示例**

在dmSQL命令行中输入以下命令，终止备份服务器。

```
dmSQL> CALL SETSYSTEMOPTION('bksvr', '0');
```

使用JServer Manager工具终止备份服务器

如果数据库处于离线状态，您可以使用JServer Manager图形化工具终止备份服务器。JServer Manager将自动更改配置文件**dmconfig.ini**中的关键字**DB_BkSvr**的值。在重新启动数据库时，备份服务器将不启动。如果数据库处于在线状态，只有在您重新启动数据库时，对备份服务器的更改才会生效。要想获得更多有关如何使用JServer Manager工具来终止备份服务器的内容，请参考*服务器管理工具用户手册*。

15.7 备份历史文件

使用备份服务器进行自动备份可自动地将需备份的日志文件、备份时间和备份路径这些信息存储到备份历史文件中。

备份历史文件的存放位置

备份历史文件是一个文本文件，存储在**dmconfig.ini**文件中**DB_BkDir**关键字的第一个目录下。备份历史文件创建在在线备份路径，并且命名为**dmBackup.his**，在恢复数据库的过程中将自动使用此文件，但离线备份历史文件命名为**offBackup.his**。

了解备份历史文件

备份历史文件包括很多与备份id、文件名、备份时间和日期有关的信息。DBMaster使用备份历史文件来追踪备份序列，同时保证完整备份、差异备份和增量备份的一致性。

以下是备份历史文件的格式：

```
<backup_id>: file_name -> archive_file_name， time， event
```

以上语法格式表示由于事件event，将文件名file_name复制成活动文件名*archive_file_name*。这些事件是一个说明备份的动作或原因的文本字符串，此字符串可以为JOURNAL-FULL、TIME-OUT、ON-LINE-FULL-BACKUP-BEGIN、ON-LINE-FULL-BACKUP或ON-LINE-FULL-BACKUP-END。字符串JOURNAL-FULL表示当日志满时，将执行一个增量备份；字符串TIME-OUT表示当超过时间计划的备份时间时，将执行一个差异备份或增量备份；字符串ON-LINE-FULL-BACKUPxxxx指它是一个完整备份。

使用备份历史文件

如果日志满载的情况经常发生，您可以降低备份日志的填充系数或缩短它的间隔时间。同样，您也可以通过检测备份历史文件发现是否备份的时间间隔太短。如果同一个日志文件在备份历史文件中被连续的复制，

那么时间间隔就有可能是太短，这样将会浪费磁盘空间，因为每一个文件只能包含少数的更改日志块。为了避免这种情况的发生，我们可以使用压缩备份模式或延长备份的时间间隔。

如果每次都有很多的日志文件被复制，这就可能是时间间隔太长。这种情况是很危险的，因为在发生磁盘故障时，可能会丢失很多数据。用户可通过缩短备份时间间隔避免此种情况的发生。

为了减少恢复介质故障的时间，您可以定期执行一个完整备份而不用考虑备份服务器是否在运行。同时将减少您所需的备份存储的数量。

了解文件对象的备份历史文件

文件对象的备份历史文件为**dmFoMap.his**，将文件对象的备份参数设为ON，您可以记录那些已经作过备份的文件对象。**dmFoMap.his**存放于"*<DB_BkDir>\FO*"目录下，它是一个纯ASCII码的文本文件，用于记录源文件名和备份文件名。

以下是文件的格式：

```
Database Name: MYDB
Begin Backup FO Time: 2001.5.13 2:33
FO Backup Directory: /DBMaster/mydb/backup/FO (i.e. DB_BkDir/FO)
[Mapping List]
s, fo00000000000.bak, "/DBMaster/mydb/fo/ZZ000001.bmp"
u, fo00000000001.bak, "/home2/data/image.jpg"
....
s, fo00000002345.bak, "/DBMaster/mydb/fo/ZZ00AB32.txt"
```

第一列中的**s**或**u**分别代表系统文件对象或用户文件对象，第二列给出了备份的文件名，第三列给出了源文件的完整路径和名称。

了解存储过程的备份历史文件

存储过程的备份历史文件为**dmSpBk.his**，将存储过程的备份参数设为**ON**将可以记录所有备份的ESQL存储过程和JAVA存储过程。**dmFoMap.his**存放于由关键字**DB_BkDir**所指定目录的子目录**SP**下，它是一个纯ASCII码的文本文件，用于记录ESQL存储过程和JAVA存储过程的备份信息。

其格式如下所示：

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

第1列表示存储过程的类型，第2列表示存储过程的拥有者；第3列表示0存储过程的名称；第4列和第5列则分别表示存储过程对象文件的名称和相应的源文件。

15.8 复制数据库的备份

在普通数据库和主数据库端，用户可以执行完整备份、差异备份和增量备份，备份方法如前。在主数据库端，JServerManager不能执行交互式增量备份。此外，在主数据库端执行交互式增量备份时，增量备份文件无法删除。

请注意在主数据库端，可能仍有许多完整备份前的增量备份存储在复制备份序列中。同时，由于完整备份，复制服务器可能不删除增量备份文件。因此，如果备份间隔时间太长，**DB_BkDir**设置的路径中将存在大量文件。

总之，复制服务器和备份服务器必须配合好才能不互相干扰。一方面，备份不能干扰复制，换句话说，无论是否有完整备份或差异备份正在执行或已经执行完，复制服务器均能将所有事务复制到从端。另一方面，复制也不能损坏备份序列。

您可以使用备份序列恢复主数据库。然而，主数据库恢复后，数据库复制也随即被终止。若要继续复制数据库，您必须使用新的主数据库替换从数据库，也就是说，您必须用主数据库文件替换所有从数据库文件。

复制数据库的备份有以下限制：

- 主数据库开启后，BMODE和BKSVR必须开启。
- 运行期间，不能更改主从数据库端的BMODE、BKSVR、BKDIR，例如，调用*setsysoption ('bkdir','new-bkdir')*将返回错误。
- 在主数据库和从数据库端，**DB_BkDir**应设置为单路径。如果您将**DB_BkDir**设置为多路径，仅第一条路径可用且路径大小被忽略。
- 在主数据库端，不能使用JServerManager执行交互式增量备份。
- 在从数据库端，无法执行完整备份、差异备份和增量备份。

15.9 恢复选项

恢复数据库将重建一个数据库，备份还原会利用数据库的最近一次完整备份，以及备份日志中记录的更改信息来恢复数据库。

分析恢复选项

什么样的恢复操作是有效的?

此问题答案取决于由数据库是否处于BACKUP模式。

- 如果数据库操作在NONBACKUP模式上，那么发生磁盘故障时，您的操作只能是恢复到最近的完整备份处并且重启数据库，从最后一次的完整备份以来的操作都会丢失，您必须重新执行所有的操作。当是这种情况时，您就无需考虑以下的问题。
- 如果数据库操作在BACKUP（BACKUP-DATA或BACKUP-DATA-AND-BLOB）模式上，您可以利用几个选项来重建受损的数据库。

恢复的准备工作

在您恢复发生磁盘故障的数据库之前，请考虑以下问题：

- 您想让数据库恢复到什么时间点？
- 如果您想让数据库恢复到磁盘故障时，那么将备份受损数据库的所有日志文件。这些文件将帮助DBMaster把一个数据库还原到最近的时间点。
- 什么样的文件已经做过备份？
- 找出最近的完整备份和所有增量备份的复制点。例如，假设您在每月的30号执行一个完整备份，并且每10天执行一个增量备份，每15天执行一个差异备份，如果您的系统在5月25日遭到损坏，您需要4月30日的完整备份，5月15日的差异备份，5月10日和5月20日的增量备份，以及5月25日受损坏的日志文件。利用这些文件，DBMaster会将您的数据库恢复到5月25日受损之前的状态。有效的备份结果是由一

组完整备份文件和一系列差异备份、增量备份文件组成，它是恢复数据库的前提条件。在线备份结果由命名为**dmbackup.his**的历史文件决定，离线备份结果由命名为**offbackup.his**的历史文件决定，因此备份的历史文件是非常重要的。所以DBMaster在恢复数据库时，会访问备份历史文件中的信息。

执行恢复

当执行恢复操作时，DBMaster会执行以下步骤：

- 拷贝所有完整备份文件，将数据文件、blob文件和日志文件拷贝至**dmconfig.ini**文件中**DB_DbDir**关键字指定的目录下。该操作将会重写原始的数据库文件。因此在使用恢复工具之前，建议用户手动地将源数据库文件拷贝至其它路径下，至少确定日志文件已经被保存。这样即使恢复失败，仍可以将数据库恢复到当前状态。
- 将差异备份或增量备份文件单独或同时应用到数据库中。

当使用恢复工具时，用户可以指定：

- 在系统配置文件**dmconfig.ini**中是否恢复数据库选项。如果选择恢复，则应该指定恢复的配置文件**dmconfig.ini**的完整路径。
- 如果您想使用备份结果将数据库恢复到不同于原始位置的其它位置，可修改数据文件路径中的以下关键字：**DB_DbDir**、**DB_DbFil**、**DB_UsrDb**等。 如果将备份结果转移到其它位置或其它计算机，在恢复数据库时应考虑如下几点：
  - **a)** **dmconfig.ini**文件中数据库名称必须和备份数据库名称一致。
  - **b)** 必要时为数据文件和BLOB文件设置BKDIR及其它关键字。
  - **c)** 如果日志文件超过一个，必须对关键字**DB_JnFil**的值进行设定，确保设定值和备份数据库中日志文件的数目一致。
  - **d)** 如果备份文件存放于多个文件夹，关键字**DB_BkDir**的设置值必须包含这些文件夹。文件**dmbackup.his**必须存放于第一个BKDIR中。

注意 *用户可以从存放备份结果的文件夹中复制一个已经存在的* ***dmconfig.ini****配置文件，并修改相关关键字，配置一个新的* ***dmconfig.ini****文件。*

- 如果**dmBackup.his**或**offBackup.his**不在默认的路径下时，则需要备份历史文件**dmBackup.his**或**offBackup.his**的完整路径。
- 恢复时间（RTime）。RTime指明了何时开始恢复数据库，它决定了目前的备份序列是否有效和运用哪一个差异备份和增量备份文件。用户可以在使用恢复工具时设定恢复时间，或者在系统配置文件**dmconfig.ini**或备份配置文件**dmconfig.ini**中添加关键字**DB_RTime**来设定恢复时间。如果RTime未被设定，则将默认为当前时间。

DBMaster提供两种恢复方法：服务器管理工具和命令行工具中的Rollover回滚命令。

更多的服务器管理工具使用方法请参考服务器管理工具用户手册，命令行工具中的Rollover回滚命令，请参考*使用Rollover恢复数据库*。

使用Rollover恢复数据库

用户同样可以使用命令行工具中的Rollover来恢复数据库。同服务器管理工具中恢复数据库选项的原理相同。

Rollover的语法如下：

rollover database_name [-i inifile] [-r rtime] [-h hisfile] [-m foMapfile] [-f FOtype] [-t rsSP]

方括号中的六个可选参数：

-i 指定配置文件**dmconfig.ini**的完整路径。如果用户指定配置文件**dmconfig.ini**恢复数据库，rollover回滚操作将会用指定的配置文件**dmconfig.ini**对应的数据库配置信息取代系统配置文件**dmconfig.ini**中对应的数据库配置信息。否则DBMaster不会恢复配置文件**dmconfig.ini**。

-r 表明数据库恢复的时间。选项r是指定恢复开始时间的第一个方法，第二个方法是在系统配置文件**dmconfig.ini**中添加关键字**DB_RTime**，或者备份要恢复的数据库配置文件**dmconfig.ini**。如果既没有–r选项，也没有添加关键字**DB_RTime**，恢复时间将默认为当前时间。

-h 给出**dmBackup.his**文件或**offBackup.his**文件的完整路径。默认路径为"***DB_BkDir\dmBackup.his***"或"***DB_BkDir\offBackup.his***"。

-m 给出**dmFoMap.his**的完整路径。默认路径为**"*DB_BDir/FO/dmFoMap.his*"**。

-f 指定将被恢复的FO文件类型。共有四个值，0代表没有FO文件要被恢复；1代表恢复系统FO文件；2代表恢复用户FO文件；3代表恢复所有FO文件。默认值为***3***。

-t 指定是否恢复存储过程。共有两个值，0代表不恢复存储过程；1代表恢复所有存储过程。默认值为***1***。

16 分布式数据库

这一章主要介绍DBMaster提供的分布式数据库（Distributed Database）管理功能，包括分布式数据库的简要介绍、DBMaster的分布式架构、分布式数据访问、分布式数据库对象管理以及分布式事务管理。

16.1 分布式数据库简介

传统的客户机/服务器模式的数据库管理系统，如图16-1所示。是将数据库储存于网络上的某个特定节点，而所有来自客户端的请求都由这个节点上的计算机系统负责响应处理。

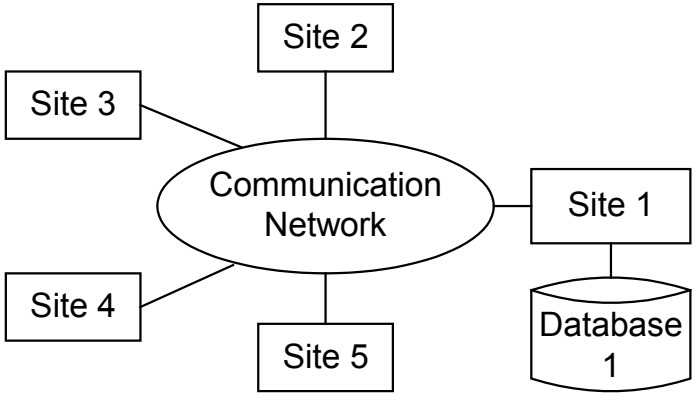


图16-1 传统的客户机/服务器数据库管理系统

分布式数据库如图16-2所示，是指储存于不同计算机网络节点上，逻辑上相互关联的数据库的统称。网络上的各个节点都有独立自主的能力以支持区域内的应用系统，各节点的数据通过数据通信网络形成互联。分布式数据库管理系统负责管理这些分布于各个节点的数据库，使得各节点的使用者可以访问数据而不受数据所在位置的影响。

DBMaster是一个完全支持分布式架构数据处理的数据库管理系统（DDBMS）。它提供了远程数据连接（remote database connection）、分布式查询（distributed query）以及分布式事务管理（distributed transaction management）的功能；DBMaster还可以通过表和数据库的复制来自动更新数据。但是当前版本的DBMaster不支持DDB模式的子查询。

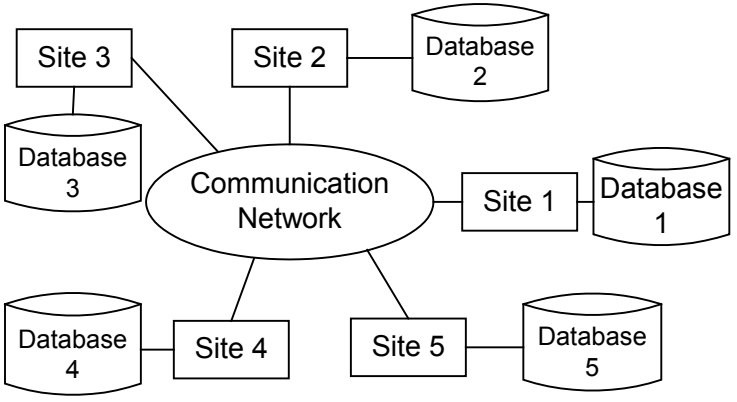


图16-2 客户机-服务器的分布式数据库

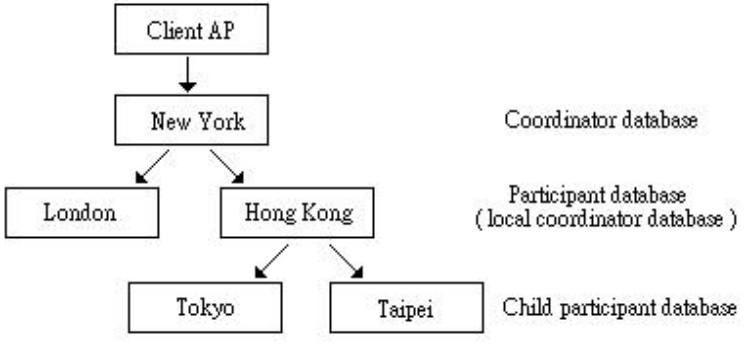
在DBMaster分布式环境下，用户可以在应用程序中使用DBMaster所支持的ODBC/JDBC提供的API（DBMaster4.1 已经支持ODBC 3.0），或结构化查询语言（SQL）来访问分布式数据库系统中不同部分的数据，DBMaster都将返回用户想要的结果，就如同访问本地数据库一样。

本章我们将简述DBMaster的分布式系统架构和分布式数据库管理的基本功能，包括分布式环境配置、远程数据管理、分布式查询和分布式事务管理。无论您是数据库管理员还是应用程序开发者，都会从本章感受到DBMaster分布式架构的简易和强大。

16.2 DBMaster的分布式结构

DBMaster的分布式环境建立在客户机/服务器模式架构上，包含了多个客户端应用程序和多个数据库服务器。客户端应用程序负责处理用户的输入/输出，而数据库服务器则负责管理数据处理。每个客户端应用程序都会有一个与之单独连接的数据库服务器，此数据库称为协调者数据库（Coordinator Database），而客户端应用则可以通过协调者数据库服务器连接其它远程数据库服务器，间接连接的数据库称为参与者数据库（Participant Database）。

DBMaster的分布式结构可以是多层次的。也就是说，可以通过参与者数据库服务器存取远程数据库中的数据。此时，此参与者数据库同时为此子事务区域的协调者数据库（Local Coordinator Database）。



图示16-3 DBMaster 分布式结构

在上图中，客户端应用程序连接着纽约的数据库服务器，因此纽约的数据库称协调者数据库；用户通过纽约这个数据库，访问伦敦与香港的数据，因此伦敦与香港这两个数据库就称为参与者数据库；而在访问香港数据的同时，可能会参考到东京与台北的数据，因此东京与台北称为子参与者数据库；而对于东京和台北这两个数据库，香港是它们的协调

者。所以香港这个数据库同时扮演两个角色，既是参与者数据库，也是区域协调者数据库。

16.3 分布式数据库环境

利用DBMaster建立一个分布式数据库环境非常简单，用户只需在配置文件dmconfig.ini中加入一些与分布式数据库有关的系统参数即可。用户也可以通过JConfiguration工具来完成这个工作，要想获得更多有关JConfiguration的信息，请参考*配置管理工具用户手册*。

以下内容是配置DBMaster分布式环境时，必须在**dmconfig.ini**文件中设定的参数。以DB_为前缀的参数是DBMaster客户机/服务器模式下，客户端与协调者数据库的连接参数，以DD_为前缀的参数是DBMaster分布式数据库中，协调者数据库与参与者数据库的连接参数。

- **DB_SvAdr = <ip_address/host name> —** 协调者数据库服务器的IP地址或主机名称。
- **DB_PtNum = <port number> —** 协调者数据库服务器和客户端应用程序间通信的TCP/IP通信端口号。
- **DD_DDBMd = <0/1> —** 是否启动协调者数据库服务器为分布式模式，默认值为 0 （OFF即关闭分布式数据库模式）。
- **DD_CTimO = <number of seconds> —** 协调者数据库服务器连接到参与者数据库服务器的等待时间，单位为秒，默认值为5秒。
- **DD_LTimO = <number of seconds> —** 协调者数据库服务器访问参与者数据库数据的锁定等待时间，单位为秒，默认值为5秒。
- **DD_GTSvr = <0/1> —** 是否启动GTRECO常驻服务程序，默认值为1（ON），但此参数只在分布式数据库环境下起作用。
- **DD_GTItv = <YYYY/MM/DD hh:mm:ss> —** 由GTRECO常驻服务程序来处理未决全局事务的间隔时间。

DBMaster分布式事务可以支持因网络或节点错误而产生的失败查询，提供事务自动恢复机制。GTRECO常驻服务程序，是负责事务自动恢复机制的程序，此程序会定期地检查并尝试恢复网络中分布式数据库服务器

上的未决全局事务（pending global transaction）。要启动GTRECO常驻服务程序，请设定**dmconfig.ini**配置文件中的**DD_GTSvr**系统参数。

为了更好的理解DBMaster如何管理分布式数据库，请参考以下示例。

➲ **示例**

现在，让我们以ABC银行业务处理为例来探讨如何管理DBMaster分布式数据库对象。假设ABC银行拥有洛杉矶、西雅图两家分行，各分行都能独立维护其区域内的客户和业务数据，但是行政、财务等工作则由洛杉矶分行统一管理。以下是此实例的各数据库配置文件范例。

洛杉矶分行数据库服务器主机的配置文件（**dmconfig.ini**）内容：

```
[BankTranx]                              ;LA branch business database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
DD_DDBMd = 1

[BankMIS]                                ;government and financial
database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.1
DB_PtNum = 30000
DD_DDBMd = 1

[BankTranx@Seattle]                      ;Seattle branch business
database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_CTimO = 20
```

```
DD_LTimO = 10
```

西雅图分行数据库服务器主机的配置文件（**dmconfig.ini**）内容：

```
[BankTranx]                              ;Seattle branch business
database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_DDBMd = 1

[BankMIS]                                ;government and financial
database
DB_SvAdr = 192.168.0.1
DB_PtNum = 30000
DD_CTimO = 20

[BankTranx@La]                           ;LA branch business database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
DD_CTimO = 20
DD_LTimO = 10
```

洛杉矶分行营业柜台银行业务应用程序的配置文件（**dmconfig.ini**）内容：

```
[BankTranx]                              ;LA branch business database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
```

西雅图分行营业柜台银行业务应用程序的配置文件（**dmconfig.ini**）内容：

```
[BankTranx]                                ;Seattle branch business
database

DB_SvAdr = 192.168.0.2

DB_PtNum = 21000
```

从上述例子中我们可以发现，在参与者数据库与协调者数据库上，必须设定**DD_DDBMd = 1**以启动分布式数据库环境，对每一个远程数据库（参与者）则需设定通信地址及**DD_CTimO**、**DD_LTimO**等远程连接参数。

此外，在上述数据库名称段落中，我们可以发现洛杉矶分行与西雅图分行的数据库名称是一样的。为了能够辨别分布式环境中相同名称的分布式数据库，DBMaster提供了在数据库名称中能区分描述服务器的方法。

远程数据库名应该是如下格式：

```
database_name@server_host_description
```

在这里，对服务器主机的描述可以是任意代号，比如数据库服务器的IP地址、域名称（domain name）或其它有关此远程数据库的任何描述。也就是说，要访问西雅图分行的数据，在西雅图分行的业务应用程序中，可以用**BankTranx@Seattle**来表示。同样，要访问洛杉矶分行的数据，在洛杉矶分行的业务应用程序中，可以用**BankTranx@La**来表示。

然而，我们需要对当前数据库和远程数据库分别进行服务器地址和端口号参数的配置。

在这个例子中，洛杉矶分行的配置文件应该包含**BankTranx**中的当前服务器地址和**BankTranx@Seattle**中的西雅图分行服务器地址。同样西雅图分行的配置文件中也应该包含**BankTranx**中的洛杉矶分行地址和**BankTranx@La**中的西雅图分行的服务器地址。

您也可以设置**DD_CTimO**和**DD_LTimO**远程连接参数。这些参数用于配置协调者数据库配置文件中的参与者数据库，同时也可用于配置参与者数据库配置文件中的协调者数据库。

网络节点上的每个数据库服务器都可以操作分布式数据库中的数据。分布式网络中的任意一个数据库服务器通过协调者数据库服务器访问数据，与一般的客户机/服务器模式下的数据库架构没有什么区别。涉及到远程（参与者）数据库服务器的数据指令，DBMaster会通过协调者数据库服务器，间接的将SQL指令传送给远程（参与者）数据库服务器。为了处理这类远程数据指令，协调者数据库服务器会将该SQL命令分解成数个数据指令和远程数据指令，然后将每一个远程数据指令传送给对应的远程数据库服务器，等远程数据库服务器处理完成后，再由协调者数据库服务器合并整合所有收到的数据响应给客户端。

16.4 分布式数据库对象

DBMaster提供了三种方法，让用户能够访问远程的（即参与者）数据库：

- 直接指定参与者数据库名称
- 用定义在协调者数据库服务器上的数据库链接（database link）来连接远程数据库
- 通过远程（参与者）对象映射（remote object mapping）直接访问远程数据库对象，如视图（view）和同义字（synonym）

前两种远程数据库连接方法的差异在于：数据库链接可包含数据库的安全信息。简单地说，建立数据库链接时可设定连接远程数据库的用户名及密码。

分布式查询与传统客户机/服务器模式数据库查询的不同之处，仅在于指明数据库对象的语法差异。然而，用户只能连接远程数据库的表、视图、同义字和存储命令。处理远程数据库对象时，我们需要在数据库对象前指明远程数据库名称或数据库链接名称。远程数据库名称或数据库链接名称与数据库对象以“：”来连接。也就是说，利用以下两个方法来说明我们要访问的是远程数据库对象：

- 远程数据库名称：数据库对象拥有者.数据库对象名称。
- 数据库链接名称：数据库对象拥有者.数据库对象名称。

➲ **示例1**

查询远程数据库对象：

```
dmSQL> SELECT * FROM Bank:EmpTaple;

dmSQL> DELETE FROM Bank:EmpTable WHERE id = 101;

dmSQL> INSERT INTO Link1:mis.account VALUES (2003,'Kevin Liu','2327-
0021');
```

➲ **示例2**

同时访问两个远程（参与者）数据库：

```
dmSQL> SELECT * FROM ABCBank@La:account a,
                     ABCBankMIS@Seattle:account b
                WHERE a.name = b.name;
```

➲ **示例3**

在远程数据库DB1中创建存储命令cmd1，然后在本地数据库DB2中执行该存储命令：

```
dmSQL> EXECUTE COMMAND DB1:cmd1(value);
```

用数据库名称连接远程数据库

使用者可通过协调者数据库服务器，以数据库名称直接连接到远程（参与者）数据库。利用此方法，使用者必须知道定义在协调者数据库服务器上，DBMaster配置文件**dmconfig.ini**中的远程数据库名称。

➲ **示例1**

例如，ABC银行洛杉矶分行营业柜台客户端应用程序，它可利用下述SQL指令连接到台北分行数据库，客户端应用程序以数据库名称连接远程数据库时，协调者数据库服务器将以客户端登录协调者数据库时所有者的用户名和密码，来连接远程（参与者）数据库。在上述例子中，就是利用SYSADM这个用户和aa这个密码来连接台北分行数据库的。

```
dmSQL> CONNECT TO BankTranx SYSADM aa;
dmSQL> SELECT * FROM BankTranx@Taipei:SYSADM.Account ORDER BY AccID;
```

➲ **示例2**

更复杂的SQL查询，如使用join访问远程对象：

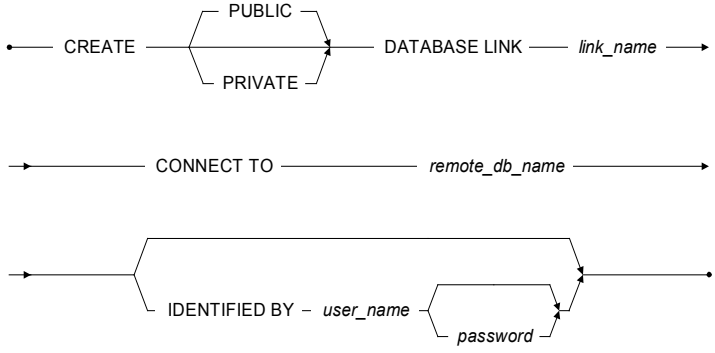
```
dmSQL> SELECT * FROM BankMIS:SYSADM.Personnel ORDER BY PID;
dmSQL> SELECT Personnel.* FROM BankTranx@Taipei:Account A,
                                BankMIS:Personnel B
```

```
WHERE A.CustID = B.CustID;
```

以数据库链接连接远程数据库

数据库链接（database link）是在协调者数据库服务器上，建立一个远程数据库登录时的用户名和密码的定义，以供个人或所有使用者通过数据库链接名去操作远程数据库。利用这种逻辑性的数据库名称定义，可轻松实现分布式数据库上的位置透明性（location transparent）目标。

建立数据库链接



图示16-4 建立数据库链接的语法

只有数据库管理员才能够建立公用的（public）数据库链接，任何合法用户都可为自己创建同名的私有链接，而此私有链接将覆盖同名的公用链接。

如果用户没有说明建立公用（public）链接还是建立私有（private）链接，DBMaster将预设为私有数据库链接。IDENTIFIED BY子句用于指定连接到远程数据库的用户名和密码。假设用户不指定连接到远程数据库的用户名，在使用这个数据库的链接名称时，将以客户端登录时的用户名和密码来连接远程数据库。

以数据库链接访问远程数据库对象

➲ **示例1**

以下以实例说明了如何以数据库链接访问远程数据。在这个例子中，user1以SYSADM的身份登陆到远程数据库，并创建一个名为**Bank_Seattle** 的公用链接以连接西雅图支行。SYSADM在更改某些字段值后，断开与数据库的连接。然后**user1**再连接表**Account**并查询表中的信息。

```
dmSQL> CONNECT TO BankTranx SYSADM;

dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO
BankTranx@Seattle

    2> IDENTIFIED BY SYSADM;

dmSQL> UPDATE Bank_Seattle:Account SET balance = balance + 100

    2> WHERE id = 1001;

dmSQL> DISCONNECT;

dmSQL> CONNECT TO BankTranx user1 pwd1;

dmSQL> SELECT * FROM Bank_Seattle:Account;
```

➲ **示例2**

user1以user1身份连接到远程数据库，查询**SYSADM.Account**。所以在远程数据库**Bank_Seattle**中必须存在**user1**的合法用户，且user1必须拥有查询该表的权限。

```
dmSQL> CONNECT TO BankTranx SYSADM;

dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO
BankTranx@Seattle;

dmSQL> SELECT * FROM Bank_Seattle:Account;

dmSQL> DISCONNECT;

dmSQL> CONNECT TO BankTranx user1 pwd1;

dmSQL> SELECT * FROM Bank_Seattle:SYSADM.Account;
```

注意 *当数据库链接名与远程数据库同名时，以数据库链接名为优先。若欲以数据库名连接远程数据库时，请在数据库名称后加上@""。*

以下例子说明了访问远程数据库的两种不同方法：一个是通过数据库链接来连接远程数据库，一个是通过dbname@""格式的远程数据库链接名来连接远程数据库。

➲ **示例1**

SYSADM使用一个链接来连接远程数据库：

```
dmSQL> CONNECT TO BankTranx SYSADM;

dmSQL> CREATE PUBLIC DATABASE LINK BankMIS CONNECT TO BankMIS

    2> IDENTIFIED BY SYSADM;

dmSQL> DISCONNECT;
```

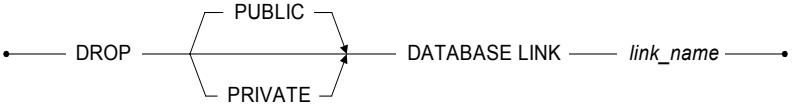
➲ **示例2**

user1使用**BankMIS@"":SYSADM.Personnel**来连接一个远程数据库。

```
dmSQL> CONNECT TO BankTranx user1 pwd1;

dmSQL> SELECT * FROM BankMIS:Personnel;              //using database
link

dmSQL> SELECT * FROM BankMIS@"":SYSADM.Personnel;    //using remote db
name
```

注意 *当使用数据库链接访问远程数据库对象并且执行UPDATE或DELETE行为时，不应该包含子查询。目前DBMaster不支持此功能。*

删除数据库链接



图示16-5 删除数据库链接的语法

数据库管理员可以删除公用的数据库链接，但无法删除私有链接。要想删除数据库的私有链接，只有此私有链接的拥有者才可以执行。当公用链接和私有链接同名时，请务必指定要删除的公用链接名。否则DBMaster会将数据库的私有链接删除。

➲ **示例**

删除一个名为**BankMIS**的公用数据库链接：

```
dmSQL> DROP PUBLIC DATABASE LINK BankMIS;
```

数据库对象映射

在分布式数据库环境中，数据库对象映射提供了更好的数据库位置透明性（location transparency）。在使用对象映射时，用户完全感觉不到所访问的数据是在远程数据库上。数据库对象映射包括：视图（view）和同义字（synonym ）。

同义字

当使用同义字（Synonym） 定义远程数据库对象的别名时，只是定义了数据库对象的别名。相应的远程数据库、远程对象（表）以及权限都会随着用户的更改而改变。

➲ **示例**

使用同义字访问远程数据库对象：

```
dmSQL> CONNECT TO BankTranx user1;
```

```
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE SYNONYM s1 FOR BankTranx:Account;
dmSQL> CREATE SYNONYM s2 FOR LK1:user2.Personnel;
dmSQL> SELECT * FROM s1;
     // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM s2;
     // SELECT * FROM LK1:user2.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM s1;
     // SELECT * FROM BankTranx:user3.Account; (BankTranx, user3)
dmSQL> SELECT * FROM s2;
     // SELECT * FROM LK1:user2.Personnel; (BankMIS, user4)
```

上例说明了如何以同义字来映射远程表。每一个select指令下的批注，表示了实际对应的访问远程表的语句，括号内指出了远程连接所使用的用户名和表的拥有者。

视图

在分布式环境下建立视图（view）来映射远程数据库和使用同义字是不同的，视图不仅仅是一个对象的别名，它还包括事先定义的数据库名称、用户名、密码、表拥有者和表名称。

➲ **示例**

使用视图访问远程数据库对象：

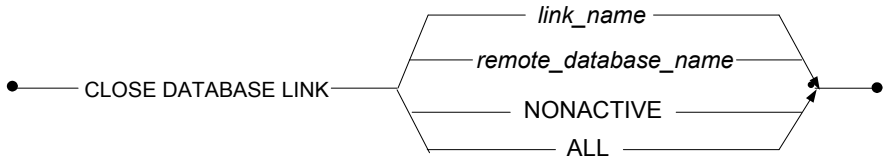
```
dmSQL> CONNECT TO BankTranx user1;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE VIEW v1 AS SELECT * FROM BankTranx:Account;
```

```
dmSQL> CREATE VIEW v2 AS SELECT * FROM LK1:user3.Personnel;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM v2;
       // SELECT * FROM BankMIS:user3.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM v2;
       // SELECT * FROM LK1:user3.Personnel; (BankMIS, user2)
```

上例说明了如何以视图来映射远程表。每一个select指令下的批注，指出了实际映射访问远程表的语句，括号里指出了远程连接使用的用户名和表的拥有者名称。

关闭远程数据库链接

当使用者以SQL指令操作远程数据库对象时，协调者数据库服务器会自动地与远程数据库建立连接。这种远程数据库链接将会一直存在，直到用户关闭与协调者数据库的连接为止，或者下达关闭远程数据库链接CLOSE DATABASE LINK指令。协调者数据库最多可支持256个远程连接，且每一个远程连接都会占用协调者数据库服务器与远程数据库服务器的系统资源。所以当您不再使用一个远程链接时，最好把它关闭。



图示 16-6 关闭远程数据库链接的语法

➲ **示例1**

使用远程数据库名**BankMIS**关闭数据库链接：

```
dmSQL> CLOSE DATABASE LINK BankMIS;
```

➲ **示例2**

使用远程数据库链接名**BankLink1**关闭数据库链接：

```
dmSQL> CLOSE DATABASE LINK BankLink1;
```

当用户使用CLOSE DATABASE LINK命令时，DBMaster会将远程连接计数器减1。当计数器归0时，表示已经没有其它数据库链接共享此远程链接，DBMaster将释放此链接资源。否则数据库间的实际连接关系仍然存在。

➲ **示例3**

关闭所有数据库链接并释放链接资源：

```
dmSQL> CLOSE DATABASE LINK ALL;
```

➲ **示例4**

关闭所有不再使用的NONACTIVE远程链接：

```
dmSQL> CLOSE DATABASE LINK NONACTIVE;
```

数据库链接系统表

有关数据库链接系统目录表包括：SYSDBLINK和SYSOPENLINK。SYSDBLINK记录着所有数据库的链接名和定义，SYSOPENLINK记录着目前开启的数据库连接关系。

16.5 分布式事务管理

DBMaster分布式数据库管理系统提供了对用户透明的分布式事务管理机制。用户不需要考虑分布式事务是否成功以及如何处理等问题。

➲ 示例

下例说明了如何进行分布式事务处理：

```
dmSQL> CONNECT TO BankTranx user1;                    // ABC Bank in
Taipei

dmSQL> SET AUTOCOMMIT OFF;

dmSQL> UPDATE BankTranx:Customer SET money=money-1000 WHERE id=123;

dmSQL> UPDATE BankTranx@"Bank_in_Seattle":Customer SET money=money+1000

    2> WHERE id=123;

dmSQL> COMMIT;
```

由于用户应用程序对数据的访问语句都必须先由协调者数据库服务器来处理。通过分布式系统目录管理器可得知该指令的作用域，作用域属于本地事务的，可由协调者数据库服务器直接处理，就像传统的客户机/服务器模式数据库事务管理一样；作用域属于远程事务的，则需要参考到远程的数据库。对于这种事务，协调者数据库服务器会与所有参与者数据库服务器之间交换必要的信息，并协调处理整笔事务直到完成或取消为止。

两阶段提交

在数据库管理系统中，维护数据完整性需要维持事务的原子特性（atomic processing），即属于同一事务中的数笔数据操作，必须同时成功或同时失败。在传统的客户机/服务器模式数据库中，我们利用日志来验证是否完成这个需求。

在分布式数据库中，会有许多子事务分散在众多的分布式数据库服务器上。为协调这些子事务，分布式数据库管理系统需要考虑更多的因素。在确认某个事务前，首先必须确认每个子事务的完成，否则该事务必须放弃。相同的，如果任一子事务的动作无法完成，事务管理器就发起一个回滚请求，接下来事务就会被回滚。这种管理分布式事务的方法称为两阶段提交（Two-Phases Commit）协议。两阶段提交能够确保整个事务的完成或取消，DBMaster即采用这种方法来达到分布式事务管理的目的。

分布式事务的恢复

当用户下达确认全局事务指令时，DBMaster会自动遵循两阶段提交协议，来通知所有参与者数据库服务器提交全局事务分支。但如果在准备确认之前，任何参与者数据库服务器发生崩溃（crash）或网络断线，则协调者数据库服务器在发现此现象后，会自动通知其它参与者数据库服务器，取消（roll back）全局事务分支，并且把全局事务确认失败信息通知用户。如果两阶段提交协议的准备阶段已经完成，但在第二阶段的确认提交过程中，任何参与全局事务的数据库服务器发生崩溃或网络断线，此时，此全局事务仍被视为提交，DBMaster会在协调者数据库的SYSGLBTRANX表中，记录哪一个全局事务分支未能确认，在崩溃或断线的参与者数据库SYSPENDTRANX表中，会记录数据库内还有未提交的全局事务（pending global transaction）分支。

用户不用担心全局事务的恢复（recovery）问题，DBMaster会自动处理发生崩溃的全局事务。协调者数据库服务器上的GTRECO（Global Transaction Recovery demon）会自动定期的扫描SYSGLBTRANX表，然后尝试连接曾发生崩溃或断线的参与者数据库，通知其将未提交的全局事务分支提交或取消。

启发式终止全局事务

虽然DBMaster会自动处理发生崩溃的全局事务，但如果因为网络不通，未提交的全局事务仍然无法获得解决。在参与者数据库上，一个没有解决的全局事务可能占用重要的系统资源，包含对数据的锁定，从而影响

其它事务无法进行。如果用户发现此现象，且与协调者数据库服务器之间的网络迟迟无法连通，用户可以使用数据库管理员工具（JDBA Tool）的启发式终止全局事务（heuristic end global transaction）功能来解决此问题。

➲ **您可以通过以下操作来手动处理未决全局事务：**

1. 在参与者数据库中，浏览SYSPENDTRANX表以查出长时间没有处理的全局事务及全局事务编号。
2. 在协调者数据库中，可以从表SYSGLBTRANX中确定未决全局事务的提交状态，也可以从参与者数据库中获得未决全局事务的提交状态。 例如：有两个未决全局事务"DB_1-3376aafd"和"DB_2-3376aafd:DB_3-3376ab0f#1"，协调者数据库的管理员应该以DB_1来查询"DB_1-3376aafd"的状态，以DB_3来查询"DB_2-3376aafd:DB_3-3376ab0f#1"。
3. 在协调者数据库中，根据未决全局事务的编号，查询协调者数据库的SYSGLBTRANX表中没有提交的全局事务的状态码（STATE字段）。如果STATE字段为2（COMMIT）或3（PENDCOM），则按**提交（commit）**按钮。如果STATE字段为4 （PENDABO），则按**终止（abort）**按钮。但是如果STATE字段为1（PREPARE），则说明此未决全局事务状态在此分支不确定，请求其父节点测定其状态。
4. 在参与者数据库中，数据库管理员使用JServer Manager来对未决全局事务进行提交或终止。

17 数据复制

广义上讲，数据复制是指在多个数据库间的对象操作处理。

以前，所有的数据都存储在某一中心。远程部门访问所需信息时必须与中心站点建立直接连接，或者向中心MIS系统申请打印的报告。但是这种连接方式较昂贵、可靠性差，并且在连接数目上有所限制，同时传送报告的方式则不够方便和及时。

开放式系统为企业提供廉价且有效的计算资源。可通过这个新的资源有效的共享所有信息，这将会成为一个企业的重要竞争优势。目前企业面对的问题不是为什么建立分布式和共享数据，而是如何更有效的使用分布式信息。数据复制正很快成为大多数分布式应用选择的架构。

17.1 表复制

什么是表复制

表复制是在目的站点中建立一个完全或部分的表拷贝。远程站点用户可直接访问一个本地的数据拷贝。本地拷贝将与另一端的数据库保持数据同步。通过这种方式，每个数据库可及时有效的操作数据而不用通过较慢的网络连接来访问远程数据库。这种请求频繁发生于总公司与各个子公司或分公司之间。

例如：当创建一个从台北数据库的A表到东京数据库的B表复制后，A表的任何更改将会复制到B表中。东京的客户可直接访问东京的数据库服务器，而不需要连接到台北的数据库去获取数据。

目的表可以驻留在同一服务器上的数据库中，这种情况一般发生在为不同功能设计的两个数据库间的数据共享，或者共享数据存在于异构数据库中（例如：Oracle、Sybase等）。通过复制从另一数据库接收到数据的表将会作为一个目的表（而不是远程表）被参考。 即使目的表可能在远程数据库中。

数据库复制与表复制的不同

数据库复制与表复制最大的不同在于复制的数据对象不同：一个是整个数据库，另一个是表。用户可根据自身需求选择其中之一。如果你选择的是数据库复制，那么复制的单位为整个数据库，目的（或者从）数据库为只读。

表复制的两种类型

表复制有两种类型。一个是同步表复制。“同步”是指任何数据的更改在目的站点都立刻反映，在修改本地表的同时目的表的数据也被修改。DBMaster利用“两阶段提交”和触发器来执行同步表复制。因此，当表

复制的机制被建立之后，任何在源表上的更改将变成分布式数据库的操作，这会影响到本地数据库的行为。当远程数据库无法连接时，本地数据库的更改操作将会失败。

另一种表复制类型是异步表复制。在这种模式下，目的表的修改将会被延时。源数据库与目的数据库之间的延时取决于用户自定义计划。异步表复制将更改信息存储于本地表中并根据自定义计划来修改目的表信息。在这种复制模式下，两个数据库可独立工作且不受网络的影响。

术语说明

源表

位于源数据库端且为复制提供数据的表。

目的表

位于目的数据库端，在复制中接收数据的表。

发布

为数据复制提供在源表中的数据集。

订阅

在目的表中接收发布的数据集。

片断

也叫做水平划分区，片断是指某一范围数据的复制。

投影

从基础表中选出要复制的字段。

复制域

复制区段（水平划分区）加上投影字段（垂直划分区）称为复制域。它指的是一个表中复制数据的范围。

当复制域发生改变时会出现一些问题。这种情况一般出现在复制的UPDATE语句中。

➲ **示例**

```
dmSQL> CREATE REPLICATION rp_case1 WITH PRIMARY AS tb_example WHERE
number > 0 REPLICATE TO db2:tb_example;

dmSQL> CREATE REPLICATION rp_case2 WITH PRIMARY AS tb_example WHERE
number < 0 REPLICATE TO db2:example;

dmSQL> UPDATE tb_example SET number=-7 WHERE number=7;
```

上例中，**rp_case1**的复制域为number > 0，**rp_case2**的复制域为number < 0。当执行复制时，并不仅仅是复制UPDATE语句，更新记录复制域也会从**rp_case1**更改为**rp_case2**，随后根据**rp_case1**执行一个DELETE语句（DELETE number = -7），根据**rp_case2**执行一个INSERT 语句（INSERT number = 7）。

数据初始化

创建复制时，用户可设置在两端如何自动初始化数据，在创建表复制后，任何对于源表的更改（插入、更新、删除）都会影响到目的表。

DBMaster提供4个选项：

- **Clear data** —执行表复制时DBMaster将删除所有目的表中的数据。
- **Flush data** —DBMaster会将源表中所有满足其条件的数据插入目的表中。
- **Clear and flush data** —DBMaster首先'clear data'，然后'flush data'。
- **Do nothing** — 目的表保持原状。

创建表复制

以下是创建同步和异步表复制的语法：

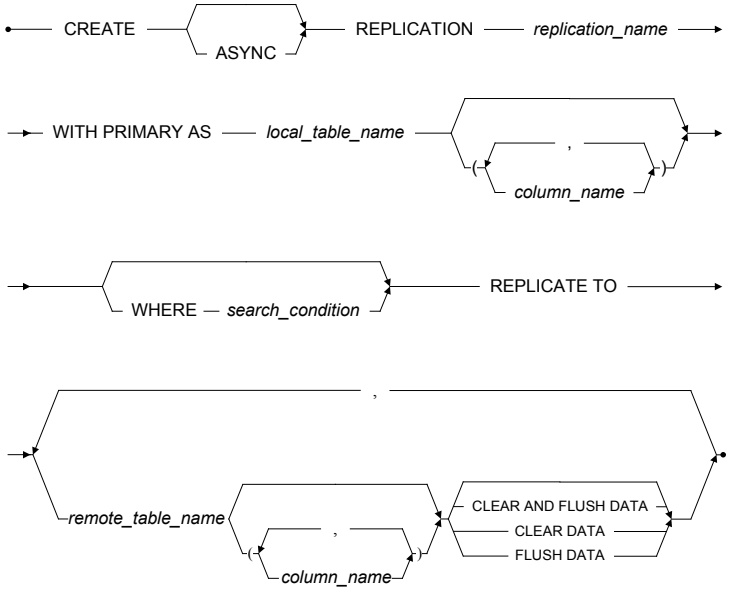


图17-1 创建表复制的语法

➲ **示例**

假设在数据库**DB30A**中有表**TB1**，在数据库**DB30B**中有表**TB2**。我们想将**TB1**数据复制到**TB2**中，并且不修改**TB2**中的当前数据：

```
dmSQL> CREATE REPLICATION rp_example WITH PRIMARY AS TB1
           REPLICATE TO DB30B:TB2;
```

在上例中，我们使用数据库连接名来定义目的数据库。在此，你也可以使用数据库链接名来定义目的数据库。

表复制规则

- 必须存在源表和目的表结构。即DBMaster在执行表复制时不能创建表。

- 表复制名必须唯一。
- 复制的订阅者名称必须统一使用 *<link_name|database_session_name>+ <table_owner_name> + <table_name>*语法。
- 每个表的投影字段中必须包含有主键。
- 设置主键的字段必须包含于片段区间字段中。
- 只有源表的拥有者或具有DBA或更高权限的用户可创建、删除或修改表复制。
- 如果在目的表中没有字段标识，那么目的表的字段名必须与源表字段名统一。
- 源表与目的表中的主键字段数必须相同。

➲ **示例1**

下面的语句将建立一个从本地数据库的**tb_salary**表到数据库**db1**的**usr1.tb_salaryA**表之间的表复制。目的表中没有定义字段名，所有**tb_salary**表中的字段必须与**tb_salaryA**表中的字段相同并且字段的类型必须相互匹配。如果**tb_salary**包含三个字段**id**、**name**和**basepay**，那么**tb_salary**表中的**id**、**name**和**basepay**字段将会被复制到**usr1.tb_salaryA**表中对应的**id**、**name**、**basepay**字段。

```
dmSQL> CREATE REPLICATION rp_salaryA WITH
       PRIMARY AS tb_salary
       REPLICATE TO db1:usr1.tb_salaryA;
```

➲ **示例2**

下面的例子创建了一个复制，它将**tb_salary**表中满足条件**id**>100的记录的（**id**, **name**）字段分别复制到数据库**db1**的**tb_salaryA**表所对应的（**Aid**, **Aname**）字段及数据库**db2**的**tb_salaryB**表对应的（**id**, **name**）字段中。执行此命令后，所有**tb_salary**中满足条件 **id**>100的数据将被复制到数据库**db2**的**tb_salaryB**表和数据库**db1**的**tb_salaryA**表所对应的**Aid**和**Aname**字段中：

```
dmSQL> CREATE REPLICATION rp_salaryAB WITH
                PRIMARY AS tb_salary (id,name) WHERE id > 100 ,
        REPLICATE TO db1:tb_salaryA (Aid,Aname),
                     db2:tb_salaryB flush data;
```

➲ **示例3**

此命令的第一个功能是删除数据库**db1**中**tb_salaryA**表的所有数据，然后创建一个复制将满足条件**id**>100的记录的（**id**, **name**）字段复制到数据库**db1**中**tb_salaryA**表对应的（**Aid**, **Aname**）字段中：

```
dmSQL> CREATE REPLICATION rp_salaryAClear WITH
                PRIMARY AS tb_salary (id,name) WHERE id > 100 ,
        REPLICATE TO
                db1:tb_salaryA (Aid, Aname) clear data;
```

删除复制

此命令删除源表中建立的复制。

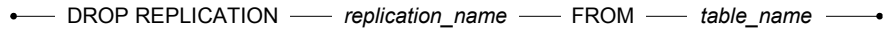


图17-2 DROP REPLICATION 语法

➲ **示例**

从**tb_salary**中删除表复制**rp_salaryA**：

```
dmSQL> DROP REPLICATION rp_salaryA FROM tb_salary;
```

修改复制

用户可从一个已存在的表复制中添加或删除目的表。下面的语法和示例说明了如何修改表复制。

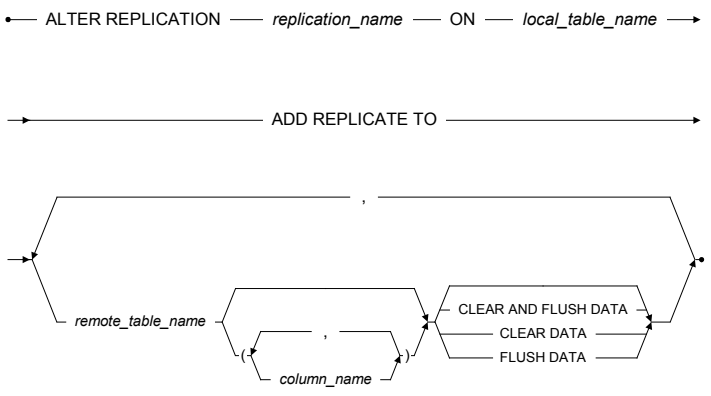


图17-3 ALTER REPLICATION … ADD REPLICATE TO 语法

➲ **示例1**

第一个命令创建了一个从本地数据库中的**tb_salary**表到数据库**dbX**中**tb_salaryX**表之间的复制；第二个命令在此基础之上添加了两个订阅者，**db1**中的**tb_salaryA**表和**db4**中的**tb_salaryB**表。

```
dmSQL> CREATE REPLICATION rp_AlterRp WITH PRIMARY AS tb_salary
       REPLICATE TO dbX:tb_salaryX;
dmSQL> ALTER REPLICATION rp_AlterRp ON tb_salary
       ADD REPLICATE TO db1: tb_salaryA,
                   db4: tb_salaryB (Bid, Bname) clear data;
```

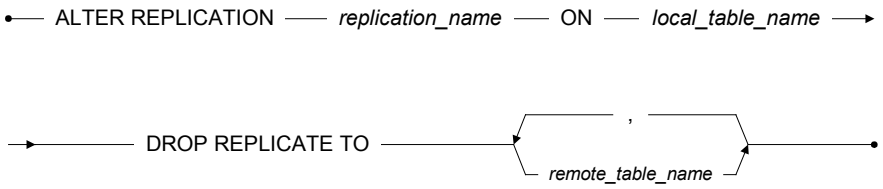


图17-4 ALTER REPLICATION … DROP REPLICATE TO 语法

➲ **示例2**

将数据库**db1**中的订阅者**tb_salaryA**从表**tb_salary**所创建的复制**rp_AlterRP**中删除：

```
dmSQL> ALTER REPLICATION rp_AlterRp ON tb_salary
       DROP REPLICATE TO db1: tb_salaryA;
```

17.2 同步表复制

两阶段提交允许分布式数据同步。当所有互连的分布式站点均确认后这个事务才被提交。一个通过网络的“握手”机制对每个事务调整各分布式站点的确认。因此无论何时发生数据更新，使用同步表复制都可保证数据的同步。

设置同步表复制

DBMaster使用两阶段提交来执行同步表复制。源数据库与目的数据库必须为分布式数据库（DDB）模式。因此，第一步是在分布式数据库模式下启动数据库（**DD_DDBMd = 1**）并且在**dmconfig.ini**文件中添加数据库连接。获取更多信息请参考第16章的*分布式数据库*。

在创建一个表复制后，任何对于源表的修改（插入、删除、更新）都将会影响到目的表。

17.3 异步表复制

异步表复制将表数据变化存储于源表中并根据时间计划修改目的表，而同步表复制在修改源表的同时也修改了目的表。

DBMaster使用复制日志文件来存储源表的改变信息。源表的改变被存于复制日志中，并根据预定义计划将数据复制到目的表中。使用复制日志，DBMaster可独立的处理源和目的端事务，即使远程连接断开也不会影响对源表的修改。异步表复制允许网络异常及目的数据库失败，DBMaster会不断尝试复制直到数据库或连接恢复正常。

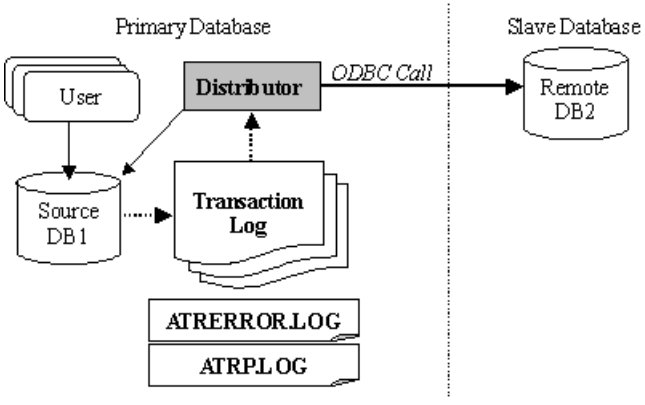


图17-5 异步表复制架构

异步表复制利用复制日志及发送服务来操作数据复制。复制日志并不是DBMaster的日志文件。复制日志的级别高于DBMaster日志文件，并且它只对于表复制有效。一个日志文件记录的是物理数据的修改，而复制日志是由这些要应用到目的表的命令组成。

当源数据库运行时，DBMaster将所有源表的修改记录在复制日志文件中。当发送服务处于激活状态时，它将根据复制日志在目的表重做所有对源表的改变操作。

通常，发送服务使用ODBC功能调用来访问目的数据库服务器。因此可以建立异构数据库间的表复制，如：Oracle、SQL Server和Informix等。异构数据库间的表复制将在本章后半部分提到。另外，快速（Express）异步表复制是另一种类型的异步表复制，它不使用ODBC功能调用，这部分内容也会在稍后的章节讲到。

➲ **创建异步表复制有三个主要步骤：**

1. 启动异步表复制。
2. 为目的数据库创建计划。
3. 根据计划创建异步表复制。

设置异步表复制

发送服务在源数据库端设置，发送服务会定时连接到目的数据库并执行表复制。

源数据库端的**dmconfig.ini**文件中，关键字**DB_AtrMd**指定是否开启发送服务。如果不设置发送服务开启，数据库将不能作为源数据库来建立异步表复制。

在源数据库端的**dmconfig.ini**文件中，关键字**RP_LgDir**为DBMaster指定在异步表复制中放置复制日志文件的路径。复制日志文件为二进制并且用户不应手动删除它。**RP_LgDir**默认路径为数据库根目录下的**/TRPLOG**子目录。

当以库结构检测模式创建表复制时，必须在源数据库及目的数据库上设置分布式数据库模式开启，即（**DD_DDBMd** = 1）。如果时间计划中设置 NO CHECK模式，DBMaster则不检测库结构，分布式数据库模式可设置为关闭状态（**DD_DDBMd** = 0）。接下来的部分将会详细介绍如何创建复制时间计划。

➲ 示例

设置从数据库**SRCDB**到目的数据库**DESTDB**的表复制，需在源数据库服务端的**dmconfig.ini**文件中添加以下信息：

```
[SRCDB]
DB_DbDir = /disk1/DBMaster/src
DB_UsrBb = /disk1/DBMaster/src/SRCDB.BB 3
DB_UsrDb = /disk1/DBMaster/src/SRCDB.DB 150
RP_LgDir = /disk1/DBMaster/src/trplog
DB_AtrMd = 1
DD_DDBMd = 1
DB_SvAdr = srcpc
DB_PtNum = 22222

[DESTDB]
DB_SvAdr = destpc
DB_PtNum = 33333
```

目的数据库端的 **dmconfig.ini** 文件中增加如下信息：

```
 [SRCDB]
DB_SvAdr = srcpc
DB_PtNum = 22222

[DESTDB]
DB_DbDir = /disk3/DBMaster/dest
DB_UsrBb = /disk3/DBMaster/dest/DESTDB.BB 3
DB_UsrDb = /disk3/DBMaster/dest/DESTDB.DB 150
DD_DDBMd = 1
```

```
DB_SvAdr = destpc

DB_PtNum = 33333
```

由于发送服务通过ODBC驱动管理来执行异步表复制，所以当源数据库运行在Microsoft Windows环境下时，必须设置访问目的数据库的ODBC数据源。

时间计划 （创建和删除）

在为一个或多个目的数据库创建异步表复制前，具有DBA权限或更高权限的用户必须为之定义一个*时间计划*。时间计划定义了连接目的数据库的开始时间、周期和用户、密码。用户可在同一个源数据库上为不同的目的数据库创建不同时间计划，但是不能为一个目的数据库创建多个时间计划。

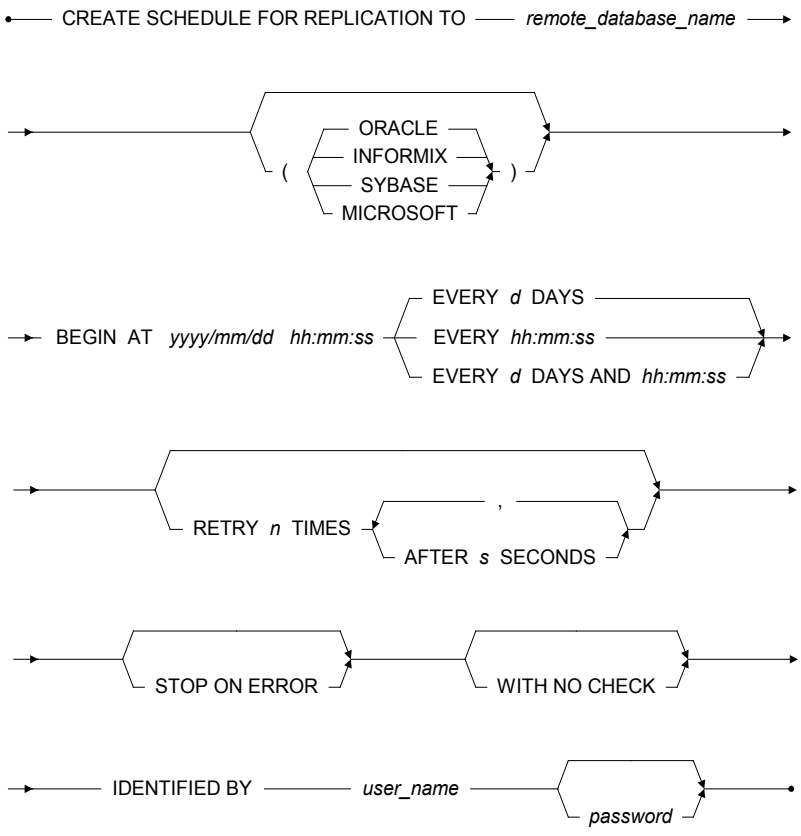


图17-6 CREATE SCHEDULE 语法

➲ **示例1**

为**DESTDB**数据库创建一个时间计划：

```
dmSQL > CREATE SCHEDULE FOR REPLICATION TO destdb
         BEGIN AT 2000/1/1 00:00:00
         EVERY 12:00:00
```

```
        IDENTIFIED BY User Password;
```

从2000年1月1日，发送服务将会每12小时执行一次异步表复制。数据库目录下的发送信息日志*ATRP.LOG* 将会记录发送服务的开启时间和状态。

IDENTIFIED BY关键字定义发送服务连接目的数据库并执行表复制的用户账号。此账号用户必须有插入、删除和更新目的数据库表的权限。

如果时间计划无用或者没有表复制与其相关，在源数据库端具有DBA权限的用户可将其删除。

图17-7 DROP SCHEDULE 语法

➲ **示例2**

删除一个时间计划：

```
dmSQL> DROP SCHEDULE FOR REPLICATION TO destdb;
```

创建异步表复制

根据同一时间计划的所有异步表复制将在相同时间执行。异步表复制机制支持所有数据类型，包括：LONG VARCHAR、LONG VARBANARY和FILE类型。

创建异步表复制与创建同步表复制类似。添加一个关键字（ASYNC、CREATE ASYNC、REPLICATION命令），对目的数据库创建基于时间计划之上的表复制。

➲ **示例1**

根据时间计划在数据库**SRCDB**与目的数据库**DESTDB**之间创建一个名为**rp_AsyRP**的复制：

```
dmSQL> CREATE ASYNC REPLICATION rp_AsyRP
```

```
WITH PRIMARY AS tb_salary
REPLICATE TO destdb:tb_salary;
CLEAR AND FLUSH DATA;
```

用户可定义CLEAR DATA, FLUSH DATA或CLEAR AND FLUSH DATA三种状态对目的站点数据进行初始化。创建复制时，DBMaster可使用数据库链接去连接目的数据库并进行检测和数据初始化。这一动作通过当前用户账号、使用DESTDB目的数据库的账号或链接名来执行。

异步表复制创建后，复制工作转至发送服务上。连接目的数据库的账号为在CREATE SCHEDULE时IDENTIFIED选项定义的值。

所有异步表复制产生的事务都将记录在源数据库目录下的发送信息日志**ATRP.LOG**中。发送信息日志是一个纯文本文件，记录发送服务的启动和操作。

➲ **示例2**

典型的**ATRP.LOG**文件内容：

```
2000/02/09 10:02:30 : start up
2000/02/09 10:02:33 : replicate transactions before 2000/02/09
                      10:02:29 (log:1.856152) to DESTDB
```

NO CASCADE选项

NO CASCADE关键字是可选项，它只应用于异步表复制。关键字定义了级联复制状态。在多数组织中命令执行是从高层到低层，例如：从A复制数据到B，然后从B复制数据到C，这是一个典型的级联复制。非级联模式是A到B之间的复制。如果你的数据模式是非级联模式，那么你可设置NO CASCADE选项，默认设置是CASCADE。

NO CASCADE选项使表复制仅执行到一个目的数据库。

➲ **示例**

Rp1是从**DB1:t1**到**DB2:t2**的复制，**Rp2**是从**DB2:t2**到**DB3:t3**的复制。如果**Rp1**设置了NO CASCADE选项，那么**DB1:t1**的改变会复制到**DB2:t2**

中，而**DB2:t2**将不会将此改变复制到**DB3:t3**中。如果**Rp1**有CASCADE选项，那么**DB1:t1**的改变将会复制到**DB2:t2**中，并且**DB2:t2**中的改变也会复制到**DB3:t3**中。

错误处理

在数据复制过程中，发送服务可能会遇到五种错误：警告错误、连接错误、数据错误、语句错误、事务错误。

警告错误

例如：如果将一个类型为CHAR(10)数据复制到CHAR(5)的字段类型时，会产生一个数据截断警告，发布服务器将会忽略这种类型的错误。

连接错误

如果发送服务连接目的数据库服务失败，则放弃这次连接计划并且等待下次连接开始。 所有工作将在下次连接后进行。

数据错误

由于异步表复制的对应关系并不是很紧密，所以其他用户首先更新目的表的情况是很可能发生的。例如，当发送服务想往目的数据库插入数据，但此数据已经存在。或者发送服务想删除一条纪录，而此记录并不存在。当此类型的错误发生时用户可以使用STOP ON ERROR选项来停止发送服务。发送服务默认行为为忽略此类型的错误。

注意 *数据错误包括完整性冲突错误和受影响行错误，完整性冲突错误可调用SQLError()函数并且返回'23000'错误码。受影响行错误调用SQLRowCount()函数，返回结果不是一个。了解更多ODBC功能信息可参考ODBC程序员指南。*

语句错误

当发送服务执行一个命令时遇到锁超时错误后，将等待并重试RETRY <*n*>TIMES选项设置的次数。AFTER <*s*> SECONDS选项设置重试的间隔时间。

事务错误

例如，死锁引起了事务错误并将事务回滚。如果发送服务遇到错误回滚事务，它将为整个事务重试一次，如果结果仍然失败，发送服务将保留这些操作到下一次时间计划的执行。

用户可查看**ATRERROR.LOG**的日志文件，它记录所有复制过程中的出错信息。这个文件位于数据库的目录下。

时间计划（挂起和恢复）

如果我们复制数据到一个位于东京的数据库，并且此数据库通常会在假日关闭，在这种情况下，我们可以将时间计划挂起直到数据库重新开启。

拥有DBA权限的用户可以挂起或者恢复时间计划。当一个时间计划被挂起后，发布服务将停止与目的数据库的连接。

➲ **示例1**

为目的数据库**DESTDB**挂起时间计划：

```
dmSQL> SUSPEND SCHEDULE FOR REPLICATION TO destdb;
```

➲ **示例2**

为目的数据库**DESTDB**重启时间计划：

```
dmSQL> RESUME SCHEDULE FOR REPLICATION TO destdb;
```

复制同步化

在某些时候用户需要使数据同步，DBMaster提供同步时间计划来实现。当源数据有所改变时，同步时间计划迫使发送服务即刻执行，用户不需等待到时间计划去激活发送服务。

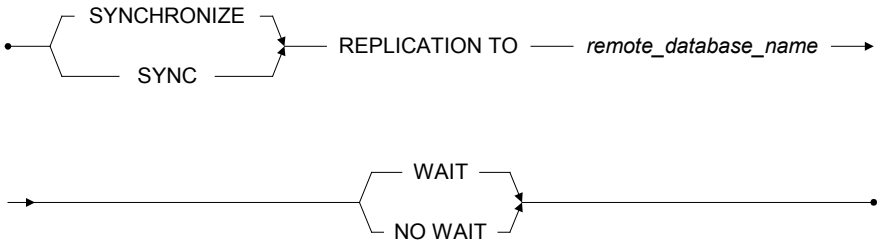


图17-8 SYNCHRONIZE REPLICATION 语法

➲ **示例1**

为目的数据库**DESTDB**同步时间计划：

```
dmSQL> SYNC REPLICATION TO destdb WAIT;
```

默认的WAIT选项使发布服务处于等待状态直到所有改变完成，只有当复制完成后命令返回。NO WAIT选项使发布服务立即执行，并且SYNC命令将立即返回。

➲ **示例2**

在SYNC PEPLICATION中使用NOT WAIT选项：

```
dmSQL> SYNC REPLICATION TO destdb NO WAIT;
```

改变时间计划

在创建一个时间计划之后，具有DBA权限的用户可以修改时间计划，包括发送服务的启动间隔时间、目的数据库的连接帐户、RETRY选项以及STOP/IGNORE ON ERROR选项。

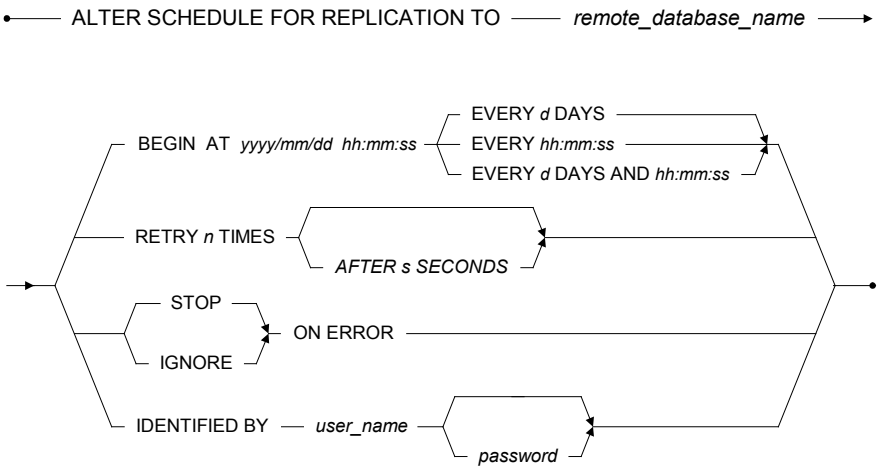


图17-9 ALTER SCHEDULE 语法

➲ **示例1**

改变目的数据库**DESTDB**的复制时间计划--添加IGNORE ON ERROR选项：

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb IGNORE ON ERROR;
```

➲ **示例2**

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb
       IDENTIFIED BY User2 Password2;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb STOP ON ERROR;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb RETRY 5 TIMES AFTER 3
       SECONDS;
```

异构异步表复制

DBMaster不仅支持与其它DBMaster数据库间异步表复制，还支持与Oracle、Informix、Sybase和Microsoft SQL server数据库间的异步表复制。这种类型的复制允许DBMaster与其它异构的数据库共存，因此称为*异构异步表复制*。

DBMaster在发送数据到第三方目的数据库前需要对复制数据进行预处理。用户若要复制数据到不同类数据库，则需在创建时间计划时，使用ORACLE、INFORMIX、SYBASE和MICROSOFT等关键字设置类型。由于DBMaster是通过ODBC驱动管理来执行异步表复制的，因此DBMaster服务必须在Windows环境下运行，并且目的数据库的名称不能使用链接名。第三方数据库可在Windows、UNIX或Linux平台下运行。为异构异步表复制创建时间计划时，使用WITH NO CHECK关键字来阻止DBMaster执行库结构检测，用户创建复制前必须检测库结构，并确保目的表与源表所对应的字段数和数据类型相匹配。

➲ **示例1**

通过名为**orcdb**的ODBC数据源连接Oracle数据库，用户名为**orcuser**，密码为**mypassword**：

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO orcdb (ORACLE)
       BEGIN AT 2000/01/01 00:00:00 EVERY 2 DAYS
       WITH NO CHECK
       IDENTIFIED BY orcuser mypassword;
```

CLEAR DATA、FLUSH DATA或CLEAR AND FLUSH DATA这些关键字在创建异构异步表复制时不能使用。复制开始前，第三方目的数据库的数据必须手动删除或插入来设置其初始状态。异构数据库表复制的计划创建与同类数据库表复制语法相同。

➲ **示例2**

tb_salary表的异构异步表复制：

```
dmSQL> CREATE ASYNC REPLICATION rp_HeteroRp
```

```
WITH PRIMARY AS tb_salary
REPLICATE TO orcdb:orcuser.tb_salary;
```

快捷异步表复制

异步表复制使用ODBC函数调用来访问目的数据库，因此在广域网环境下将降低数据库性能。为了优化性能，DBMaster提供另一种机制： 快捷异步表复制。DBMaster将命令打包后再网络传输。

由于其它数据库管理系统不支持此协议，因此快捷异步表复制不能应用到异构数据库间的复制。在创建快捷时间计划时，也不支持STOP ON ERROR选项。

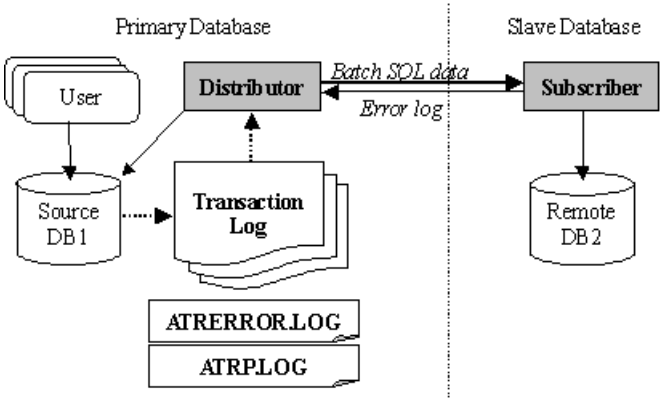


图17-10 快捷异步表复制架构

源数据库端的发送服务和目的数据库端的后台应用程序协作完成快捷异步表复制。发送服务并不通过ODBC调用直接将数据变化从源表传送到目的表，而是将源表执行的SQL命令和相关数据打包后传送到目的表的后台程序。当目的数据库获得包后，后台程序会对目的表执行这些命令。

快捷复制设置

➲ **建立快捷复制：**

1. 设置源数据库的发送服务和目的数据库的订阅者后台程序。
2. 为目的数据库创建快捷复制时间计划。
3. 根据计划创建异步表复制。

订阅者后台程序设置

开启订阅者后台程序的方法是在目的（订阅者）数据库端的**dmconfig.ini**文件中设置**DB_EtrPt**关键字。它定义了在发布服务与订阅者后台程序间的访问端口。

➲ **示例**

以快捷复制方式建立源数据库**SRCDB**到目的数据库**DESTDB**的表复制，则需在目的数据库上开启订阅者后台程序。

在目的数据库端的**dmconfig.ini**文件：

```
[SRCDB]
DB_SvAdr = srcpc    ; 告诉目的数据库源数据库所在位置
DB_PtNum = 22222    ;


[DESTDB]
DB_DbDir = /disk3/DBMaster/dest
DB_UsrBb = /disk3/DBMaster/dest/DESTDB.BB 3
DB_UsrDb = /disk3/DBMaster/dest/DESTDB.DB 150
DD_DDBMd = 1
DB_SvAdr = destpc
DB_PtNum = 33333
DB_EtrPt = 44444    ; 订阅者后台程序使用的端口号
```

在源数据库，用**DB_AtrMd**关键字开启发送服务。发送服务不需要知道目的数据库中订阅者后台程序对应的端口号。

在源数据库端的**dmconfig.ini**文件：

```
[SRCDB]
DB_DbDir = /disk1/DBMaster/src
DB_UsrBb = /disk1/DBMaster/src/SRCDB.BB 3
DB_UsrDb = /disk1/DBMaster/src/SRCDB.DB 150
RP_LgDir = /disk1/DBMaster/src/trplog
DB_AtrMd = 1
DD_DDBMd = 1
DB_SvAdr = srcpc
DB_PtNum = 22222

[DESTDB]
DB_SvAdr = destpc
DB_PtNum = 33333
```

快捷异步表复制时间计划

在CREATE SCHEDULE命令中使用EXPRESS选项来定义快捷异步表复制。

➲ **示例**

为快捷异步表复制建立时间计划：

```
dmSQL> CREATE SCHEDULE FOR EXPRESS REPLICATION TO destdb
       BEGIN AT 2000/1/1 00:00:00
       EVERY 12:00:00
       IDENTIFIED BY User Password;
```

创建快捷异步表复制

步骤相同于创建异步表复制。要想获取更多信息请参考*创建异步表复制*或*SQL命令与函数参考手册*的相关章节。

17.4 数据库复制

大部分企业都将数据存放在其计算机中心的文件服务器或者数据库上，并且所有终端都需直接连接到服务器上。在这种架构下，如果所有终端都在同一建筑或同一区域，那么应用系统可以正常运行。然而由于传输速度和网络带宽的影响，当远程终端要访问数据库时性能将会很差。

为了达到数据共享、加快访问速度的目的，DBMaster提供了数据库复制功能。数据库复制将在某个预设时间段进行主数据库到从数据库间的复制。 换句话说，当主数据库信息发生改变时，如添加、修改或删除数据，DBMaster会自动将所有改变复制到从数据库中。使用数据库复制有三个好处：第一，由于应用系统可直接在本地存取数据，可显著的提升应用系统存取数据的性能。第二，源数据库发生变化时，远程的数据库会有相同的数据变化，有效的达到数据资源共享的目的。第三，当本地的数据库无法连接时，应用系统仍可连接至远程的数据库继续工作。数据复制有其优点，但同时需要更多的空间来储存数据以及更多的运算来执行数据复制操作。

数据库复制基础

在这部分，我们通过图表17-11来说明数据库复制过程。数据库复制需在日志备份服务器模式下工作。如果您对数据库备份和恢复不太熟悉，那么可以参考第15章*数据库恢复，备份和还原*。

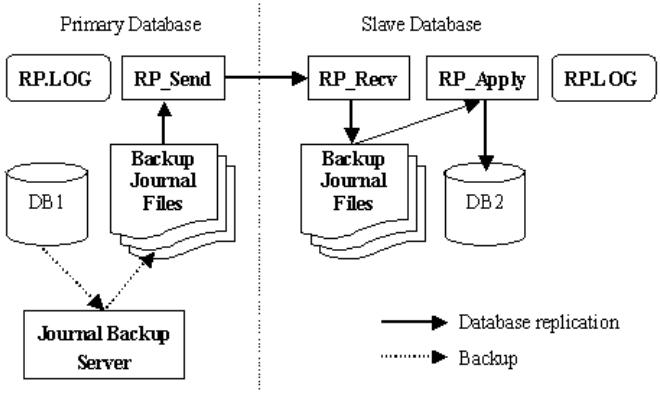


图17-11: 数据库复制流程图

数据库复制过程通过四个服务器来完成：日志备份服务器、RP_SEND服务器、RP_RECV服务器和RP_APPLY服务器。日志备份服务器与RP_SEND服务器在主数据库端启动。而RP_RECV服务器和RP_APPLY服务器在从数据库端启动。

数据库复制的初始步骤是创建与主数据库相同的从数据库。在这之后，所有主数据库的改变，如：数据插入、删除、创建表结构等，将会通过日志备份服务器定期写入备份日志中。

主数据库端的RP_SEND服务器会将日志文件周期性的发送到从端的RP_RECV服务器。RP_RECV将使RP_APPLY服务器执行接收的日志更新从数据库。

对于所有用户来说从数据库都是只读的，只有RP_APPLY服务器可以更新从数据库。

设置数据库复制

➲ **手动设置数据库复制：**

1. 建立和主数据库相同的从数据库。
2. 在主数据库端设置日志备份服务器和RP_SEND服务器。

3. 在从数据库端设置RP_RECV服务器和RP_APPLY服务器。
4. 启动主、从数据库。
5. 查看复制日志（RP.LOG）和错误日志（DMERROR.LOG）。

完整备份

之前提到的，数据库复制第一步是在从端建立与主端相同的数据库。请注意两端数据库服务器的字节序必须相同。例如如果主数据库运行环境为x86的机器，那么从数据库运行环境也必须是与x86的字节序相匹配的机器。Sun Sparc的字节序与x86的不同， 因此从数据库不能在Sparc机器运行，反之亦然。

➲ **复制主数据库：**

1. 关闭主数据库。
2. 产生一个主数据库的数据文件、BLOB文件及日志文件的拷贝。
3. 将拷贝文件移至从数据库所在机器上。
4. 修改从数据库**dmconfig.ini**配置文件使其所指目录与文件对应。

步骤1与步骤2可以结合离线完整备份实现，更多信息请参考*离线完整备份*章节。

在此我们将举例说明如何手动执行一个离线完整备份，在这里的所有例子中，假定处理器为x86并且主数据库在Linux或FreeBSD上运行。

➲ **示例**

为了拷贝主数据库文件，查询系统表SYSFILE获取逻辑名：

```
dmSQL> SELECT * FROM SYSFILE;
```

从**dmconfig.ini**文件得到主数据库的物理文件名：

```
[MYDB]     ;; Primary Database Configuration, CPU: x86, OS: FreeBSD
DB_DbDir = /home/DBMaster/mydb
DB_UsrBb = /home/DBMaster/mydb/MYDB.BB 3
DB_UsrDb = /home/DBMaster/mydb/MYDB.DB 150
```

```
FILE1 = /home/DBMaster/mydb/data/FILE1.DB 50

FILE2 = /home/DBMaster/mydb/data/FILE2.DB 50

DB_JnFil = JN1.JNL JN2.JNL

...
```

关闭主数据库之后，将主数据库文件拷贝到从数据库机器上。从数据库运行环境为Windows平台的x86机器。在上面的例子中，所有拷贝文件包括系统文件**MYDB.SDB**和**MYDB.SBB**，用户文件**MYDB.DB**和**MYDB.BB**，日志文件**JN1.JNL**和**JN2.JNL**以及用户定义文件**FILE1.DB**和**FILE2.DB**等。

接下来将**dmconfig.ini**文件中的主数据库配置信息拷贝到从数据库的**dmconfig.ini**文件中。之后修改从数据库**dmconfig.ini**文件，使所指定的文件与实际物理文件位置一致，因为不同机器对应的实际路径可能不同。

从数据库的**dmconfig.ini**文件参数配置如下：

```
[MYDB]     ;; Slave Database Configuration, CPU: x86, OS: MS Windows NT

DB_DbDir = d:\DBMaster\db\mydb

DB_UsrBb = d:\DBMaster\db\MYDB.BB 3

DB_UsrDb = d:\DBMaster\db\MYDB.DB 150

FILE1    = d:\DBMaster\db\mydb\FILE1.DB 50

FILE2    = d:\DBMaster\db\mydb\FILE2.DB 50

DB_JnFil = JN1.JNL JN2.JNL

...
```

DBMaster支持8个从数据库。如果从数据库超过1个，那么数据库管理员将重复以上步骤来创建每个从数据库。

设置主数据库的日志备份服务器

因为数据库的所有改变都会记录在日志文件中，所以DBMaster使用备份日志文件作为复制的源数据。当日志备份服务器执行一个增量备份后，

RP_SEND服务将备份日志文件发送到从数据库中。记录在日志文件的所有主数据库的事务都被提交，以确保任何主数据库的改变都会复制到从数据库中。

➲ 示例

只有主数据库需要开启日志备份服务器。在主数据库服务器的**dmconfig.ini**文件定义备份目录和时间计划：

```
DB_BMode = 1                          ; Database start up in BACKUP-DATA
mode

DB_BkSvr = 1                          ; Start up journal backup server

DB_BkDir = /home/DBMaster/mydb/bkdir ; Directory of backup journal files

DB_BkTim = 00/01/01 00:00:00          ; The initial backup time

DB_BkItv = 0-12:00:00                 ; Perform journal backup evey 12
hours
```

DB_Bmode可设置为BACKUP-DATA （1）或 BACKUP-DATA-AND-BLOB （2）模式。然而，即使**DB_BMode**为BACKUP-DATA-AND-BLOB模式，也必须为所有参与复制的表空间设置BACKUP BLOB ON选项，否则BLOB数据将不会被复制到从数据库中。

数据发送与接收

在数据库复制过程中，有三个驻留服务器RP_SEND、RP_RECV和RP_APPLY（参看图表说明）。RP_SEND位于主数据库端并且负责向从端发送备份日志文件。RP_RECV和RP_APPLY 位于从数据库端。RP_RECV接收主端发送过来的备份日志文件，RP_APPLY根据此日志文件更改数据。RP_SEND会随主数据库的开启和关闭而自动开启和关闭。RP_RECV和RP_APPLY会随从数据库的开启和关闭而同时开启和关闭。

当日志备份服务器完成增量备份后，RP_SEND将**DB_BkDir**备份目录中所有未发送的备份日志发送到从端。同时从端数据库的RP_RECV服务器接收所有来自主端数据库的日志文件，并将这些日志文件存放在从端**DB_BkDir**所指定的路径下。日志文件被接收后，RP_APPLY将执行日志文件记录的任何操作来改变从数据库完成整个数据库复制。

在主端设置RP_SEND服务

RP_SEND和RP_RECV是分别负责备份日志文件传送和接收的服务器。因此，RP_SEND必须知道RP_RECV所在机器的IP地址和端口号。注意端口号必须是RP_SEND与RP_RECV特定的连接号，并且要不同于数据库服务的端口号。在主数据库的**dmconfig.ini**配置参数中，**RP_SEND**服务器可使用关键字RP_SlAdr来设置复制数据的目的地。

RP_SlAdr 语法：

```
RP_SlAdr = {address[:port number]}
```

默认端口号是：23001。

➲ **示例1**

一个主数据库可支持8个从数据库。可用逗号或者空格来分割从数据库信息。以下是使用了三个从数据库**192.168.9.222**（端口号 5100），**Mars**（端口号5101）和**Scorpio**（默认端口号23001）：

```
RP_SlAdr = 192.168.9.222:5100, Mars:5101, Scorpio
```

复制数据的从端是确定的，则可为RP_SEND定义执行数据库复制的初始时间和时间间隔。在时间计划确认后，RP_SEND将会定时执行数据复制。

➲ **示例2**

设置初始时间为01/01/2000 AM 01:00并且设置间隔时间为一天，使用以下**dmconfig.ini**设置：

```
DB_SMode = 4                              ；作为主数据库启动，并开启RP_SEND服务器

RP_BTime = 00/01/01 01:00:00              ; 复制初始时间

RP_Iterv = 1-00:00:00                     ；时间间隔

RP_SlAdr = 192.168.9.222:5100, Mars:5101, Scorpio     ; 复制到3个机器上

RP_ReTry =                                            ; 网络连接失败后的重试次数

RP_Clear = 1                                         ; 在发送到从端信息后清除备份日志文件
```

在配置文件中，**RP_BTime**和**RP_Iterv**参数用来为数据复制定义时间计划。**RP_BTime**设定发送备份日志文件到从数据库的开始时间，它的格式为<year/month/day hour:minute:second>。如果没有设置复制开始时间，那么默认时间为主数据库开启时间。**RP_Iterv**参数定义了执行RP_SEND服务的间隔时间。它的格式是：<day-hour:minute:second>。默认值为一天，取值范围为0-24,855 。

注意 *这些值必须在启动数据库之前设置。*

RP_ReTry参数设置网络连接失败后的重连接次数。**RP_Clear**参数决定在数据发送完之后备份日志文件是否被清除。设置**RP_Clear**为1则清除备份日志文件，默认值为0。如果备份日志文件仅为数据库复制所用，那么清除这些文件则可以释放一些存储空间。然而，如果发生硬件故障，那么备份日志的数据将不能重新恢复并且也不能执行恢复操作。在这种情况下，必须通过从端的完全备份来恢复主数据库。因此如果备份日志文件还要在主数据库备份中使用，就设置**RP_Clear**的参数为0。

在从端设置RP_RECV及RP_APPLY服务器

为了开启RP_RECV和RP_APPLY服务器，从数据库启动模式**DB_SMode**必须设置成5。

➲ **示例1**

从数据库只能接收一个主数据库数据，因此需要用**RP_Primy**定义主数据库的机器名或地址。

```
RP_Primy = FreeBSD       ; Receive replication data from machine
'FreeBSD'
```

➲ **示例2**

RP_RECV服务器使用一个**RP_PtNum**定义的不同于**DB_PtNum**的端口号接收来自RP_SEND服务器的数据。如下，假设从数据库位于NTPC机器上，并设置了**RP_PtNum**关键字。

```
RP_PtNum = 5100            ; Port number for RP_SEND and RP_RECV
connection

DB_PtNum = 3333            ; Port number for user database access
```


➲ **示例3**

主数据库使用**RP_SlAdr**关键字来重新得到从数据库的机器名或者地址和端口号。

```
RP_SlAdr = NTPC:5100        ; Slave-side machine name and address
```

另外，从端数据库必须设置备份路径**DB_BkDir**用以保存来自主数据库的日志文件。当RP_APPLY根据所接收日志文件更新从数据库后，DBMaster会自动将这些日志文件删除。

从数据库为只读

从数据库的数据必须与主端的相同。它不允许数据定义操作（DDL，如创建表和改变表）及数据修改操作（如插入、删除、修改操作）。因此我们说从数据库为只读数据库。

在数据库复制过程中，从数据库通过接收主数据库发送来的备份日志文件来恢复数据。在恢复数据过程中系统不对主数据库锁定。 因此，从数据库的查询结果可能是脏数据。换句话说，在任意时间点，同样查询在主从两端数据库中返回的结果可能是不同的，因为此时RP_APPLY正在做数据恢复。

启动主从数据库

主数据库的开启模式与从端不同。主数据库端需设置以主数据库模式启动。另一方面，在从数据库端要设置以从数据库模式启动。在**dmconfig.ini**文件中使用**DB_SMode**关键字来设置所有启动模式。主数据库端启动模式**DB_SMode**为4，从数据库端启动模式 **DB_SMode**为5。

你可以分别启动主从数据库，没有特定的顺序。

所有在**dmconfig.ini**文件中设置复制的参数将会在本部分示例中提到。

➲ **示例**

在下面的**dmconfig.ini**文件中，主数据库机器名为 **FreeBSD**，从数据库机器名为**NTPC**，主数据库设置如下：

```
[MYDB] ;; 主端配置, CPU：x86, OS：FreeBSD, Name：FreeBSD
;; 数据库相关设置
DB_DbDir = /home/DBMaster/mydb
DB_UsrBb = /home/DBMaster/mydb/MYDB.BB 3
DB_UsrDb = /home/DBMaster/mydb/MYDB.DB 150
FILE1 = /home/DBMaster/mydb/data/FILE1.DB 50
FILE2 = /home/DBMaster/mydb/data/FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
DB_SvAdr = FreeBSD                    ; 主数据库机器名
DB_PtNum = 3333                       ; 主数据库端口号
;; 日志备份服务器相关设置
DB_BMode = 1                          ; 在BACKUP-DATA模式下启动数据库
DB_BkSvr = 1                          ; 启动日志备份服务器
DB_BkDir = /home/DBMaster/mydb/bkdir  ; 指定备份日志文件目录
DB_BkTim = 00/01/01 00:00:00          ; 初始备份时间
DB_BkItv = 0-12:00:00                 ; 每12小时执行一次日志备份
;; RP_SEND 服务相关设置
DB_SMode = 4                          ; 以主数据库端启动，同时启动RP_SEND服
务器
RP_BTime = 00/01/01 01:00:00          ; 复制初始时间
RP_Iterv = 1-00:00:00                 ; 每天复制
RP_SlAdr = NTPC:5100                  ; 以端口号5100复制到NTPC机器
RP_ReTry = 3                          ; 网络连接失败后重试 3 次
RP_Clear = 1                          ; 在数据发送后清除备份文件
```

从数据库端设置：

```
[MYDB];; 从端配置., CPU：x86, OS：MS Windows NT, Name：NTPC
```

```
;; 数据库相关设置
DB_DbDir = d:\DBMaster\db\mydb
DB_UsrBb = d:\DBMaster\db\MYDB.BB 3
DB_UsrDb = d:\DBMaster\db\MYDB.DB 150
FILE1    = d:\DBMaster\db\mydb\FILE1.DB 50
FILE2    = d:\DBMaster\db\mydb\FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
DB_SvAdr = NTPC
DB_PtNum = 3333
;; RP_RECV和 RP_APPLY服务器相关设置
DB_SMode = 5              ; 启动从数据库,
                          ; 同时启动RP_RECV和RP_APPLY服务器
RP_Primy = FreeBSD        ; 接收此机器名的复制数据
RP_PtNum = 5100           ; RP_SEND与RP_RECV间连接端口号
DB_BkDir = e:\mydb\bkdir  ; 设置临时备份日志文件目录
```

立即执行数据库复制

我们提到的数据库复制驻留服务，将检测数据库中是否有数据被更改，并且将自动进行数据复制。

➲ **示例**

立即执行数据库复制，可在dmSQL中使用如下SQL命令：

```
dmSQL> SET FLUSH;
```

可使用此命令实现主从两端的数据同步。执行此命令时，日志备份服务器将立即执行增量备份并备份当前日志文件。随后三个驻留服务器（RP_SEND、RP_RECV和RP_APPLY）将完成数据复制。

查看复制日志（RP.LOG）和错误日志（DMERROR.LOG）

数据库复制过程中，如有任何网络失败或者错误信息时，系统将在当前数据库目录中产生一个日志文件DMERROR.LOG。错误日志语法如下所示：

```
yy/mm/dd hh:mm:ss Daemon name:Error number:Error message
```

➲ **示例1**

一个DMERROR.LOG：

```
97/12/31 11:40:59 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130

97/12/31 11:43:36 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130

97/12/31 11:45:00 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130

97/12/31 11:50:00 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130

97/12/31 11:50:45 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130
```

主从两端都可以产生DMERROR.LOG。

如果复制成功，系统将产生一个名为RP.LOG的日志文件。格式为：

在主数据库端：

```
RP_SEND:RPID id ~ id sent at yy/mm/dd hh:mm:ss
```

在从数据库端：

```
RP_RECV:RPID id ~ id sent at yy/mm/dd hh:mm:ss

RP_APPLY:RPID id ~ id applied at yy/mm/dd hh:mm:ss
```

复制日志RP.LOG将会在主从数据库两端产生。每个备份日志文件都有一个ID号：RPID在上面的格式中，RPID指出哪一个日志文件正在运行--RPID在RP_SEND端指出哪一个文件正在被发送，而在RP_RECV端指出

哪一个文件正在被接收，在RP_APPLY指出哪一个文件正在被恢复。通常RPID与增量备份ID相同。

➲ **示例2**

在主数据库端的RP.LOG：

```
RP_SEND : RPID  7 ~ 10 sent to 192.72.116.130 at 97/12/16 15:36:17
RP_SEND : RPID 11 ~ 11 sent to 192.72.116.130 at 97/12/16 15:59:42
RP_SEND : RPID 12 ~ 12 sent to 192.72.116.130 at 97/12/31 11:52:28
```

➲ **示例3**

在从数据库端的RP.LOG：

```
RP_RECV : RPID  7 ~ 10 received at 97/12/16 15:35:53
RP_APPLY : RPID  7 ~ 10 applied at 97/12/16 15:35:55
RP_RECV : RPID 11 ~ 11 received at 97/12/16 15:59:18
RP_APPLY : RPID 11 ~ 11 applied at 97/12/16 15:59:18
RP_RECV : RPID 12 ~ 12 received at 97/12/31 11:52:01
RP_APPLY : RPID 12 ~ 12 applied at 97/12/31 11:52:02
```

RP.LOG通常记录三个驻留服务器（RP_SEND、RP_RECV和RP_APPLY）的所有活动。文件位于所有参与者数据库目录下（**DB_DbDir**）。数据库管理员应定时检测这些文件以确保复制运行正常。

例如：如果远程数据库连接一直失败，检查网络是否正常或者远程数据库是否运行异常。

服务器管理工具（JServer Manager）环境设置

数据库复制需要建立最初的数据库环境。可使用文本编辑器直接去修改**dmconfig.ini**文件，这在前一部分已经讲过。这一部分将使用DBMaster的服务器管理工具来设置数据库复制环境。

建立一个完整备份

首先在主数据库做一个离线完整备份，将备份文件拷贝到从数据库端。要想获得有关离线完整备份的详细信息，请参考第15.5章*离线完整备份*。

主数据库设置

要想运行数据库复制服务，首先需设置增量备份，接下来在主数据库端配置环境。更多信息请参考第15章*数据库恢复*、*备份和还原*。下面，设置主数据库环境。

➲ **设置主数据库环境：**

1. 在主数据库端启动**服务器管理工具**。
2. 在**启动数据库**窗口点击**设置**按钮。
3. 在**启动数据库高级设置**窗口点击**复制**标签。
4. 设置数据库复制：
   a) 在**目标数据库的IP和端口号**框输入从数据库IP和端口号。
   b) 在**数据库复制的开始时间**框输入日期和时间。
   c) 在**重试连接次数**框中输入一个值。
   d) 也可设置复制后删除备份日志文件。
   e) 输入数据库复制的时间间隔，包括天、时、分、秒。
5. 点击**保存**按钮。
6. 点击**取消**按钮返回到**启动数据库**窗口。

从数据库设置

将主数据库完整备份从主端拷贝到从端后，使用服务器管理工具来配置从数据库。

1. 在从端启动**服务器管理工具**。
2. 在**启动数据库**窗口点击**设置**按钮。

3. 在**启动数据库高级设置**窗口点击**复制**标签。
4. 开启数据库复制：
   a) 输入源数据库IP地址。
   b) 在**目标数据库上接收的后台程序的端口号**框中输入RP_RECV的端口号。
5. 点击**保存**按钮。
6. 点击**取消**按钮，返回**启动数据库**窗口。

数据库配置文件

这部分将对**dmconfig.ini**文件中，数据库复制所用关键字加以总结说明。

主数据库配置

主数据库中设置数据复制的关键字：

- **DB_SMode** — DB_SMode设置为4，代表此数据库以主数据库模式启动。
- **RP_BTime** — 主数据库的完整备份文件发送到从数据库的开始时间。 格式为年/月/日 时:分:秒，默认值为主数据库启动时间。例如：97/12/31 12:00:00。
- **RP_Iterv** — 发送备份日志文件的时间间隔。格式为天-时:分:秒。例如：1-12:00:00，即发送备份日志文件的间隔时间为1.5天，天数取值范围为0~24,855。
- **RP_ReTry** — 网络连接失败后的重新连接次数。
- **RP_Clear** — 设置发送完日志备份文件后是否清除文件。1为清除文件，0为不清除文件，默认值为0。如果将值设置为1，当发生硬件故障时，主数据库将不能恢复。除非用从数据库来恢复它。
- **RP_SIAdr** — 这个值用来设置从数据库的地址（或机器名）和端口号。DBMaster中支持每个主数据库对应1到8个从数据库。

RP_SlAdr 语法：

```
RP_SlAdr = {Address[:Port Number]}
```

主数据库需要启动日志备份服务器来执行增量备份。

- **DB_BMode** — 1（BACKUP-DATA）或2（BACKUP-DATA-AND-BLOB）。
- **DB_BkSvr** — 设置**DB_BkSvr**为1，开启日志备份服务器。
- **DB_BkDir** — 存储备份日志文件的路径。
- **DB_BkTim** — 日志备份服务器启动时间。格式为**<**年/月/日 时:分:秒 **>**，默认值为主数据库启动时间。
- **DB_BkItv** — 执行增量备份时间间隔，格式为**<**天-时:分:秒**>**。例如：0-12:00:00为每12小时执行一次备份。

从数据库配置

从数据库端设置数据复制的关键字：

- **DB_SMode** — 设置**DB_SMode**为5代表此数据库从数据库模式启动。
- **RP_Primy** — 主数据库地址或机器名。
- **RP_PtNum** — **RP_RECV**使用的端口号。此值必须与**DB_PtNum**不同，而与主数据库中**RP_SlAdr**定义的端口号相同。
- **DB_BkDir** — 定义接收主数据库备份日志文件的临时存储路径。默认路径为**<数据库路径>/backup**。

数据库复制限制

数据库复制的限制总结如下：

- 主从机器的字节序必须相同。
- 从数据库为只读。

- 一个主数据库支持与8个从数据库之间建立连接。
- 您可以使用备份序列恢复主数据库。然而主数据库恢复后，数据库复制也随即被终止。若要继续复制数据库，您必须使用新的主数据库替换从数据库，也就是说，您必须用主数据库文件替换所有从数据库文件。
- 由于完整备份，复制服务器可能不删除增量备份文件。因此，如果备份间隔时间太长，**DB_BkDir**设置的路径中将存在大量文件。
- 目前不支持FILE数据类型的复制。
- 复制BLOB数据，即LONG VARCHAR或者LONG VARBINARY数据类型，设置**DB_BMode**为2（BACKUP-DATA-AND-BLOB），并需在创建表空间时设置BACKUP BLOB ON选项。

18 性能调优

DBMaster是一个可调整的数据库系统。为满足个人需要，可对DBMaster性能进行调整，使其达到最优状态。本章将介绍调节数据库的目的和方法，以及如何分析系统性能。

18.1 调节过程

在调节数据库前，必须明确提高性能的目的以及有些目的调节会产生冲突，必须确定哪些目的更重要。

以下列出一些调整性能的主要目的：

- 提高SQL语句性能
- 提高数据库应用程序性能
- 提高并发处理性能
- 资源优化利用

明确目的后，可对DBMaster性能进行调整。

调整性能步骤：

- 检测数据库性能
- 调整I/O
- 调整内存
- 调整并发处理
- 监测数据库性能，并与前一阶段统计数据对比

性能调优中的每个调整步骤可能会对其它步骤产生消极影响。使用以上的调整步骤可避免此问题。执行完所有调整步骤后，监测DBMaster性能，对比是否整体已达到最优性能。

在调整性能前，需明确SQL语句是否最优化，以及数据库应用程序的设计是否合理。低性能的SQL语句或不合理的应用程序设计将影响数据库性能。提高应用程序及SQL语句性能请参考*SQL命令与函数参考手册*和*ODBC程序员参考手册*。

18.2 数据库监测

本部分主要讲述如何监测数据库中各状态的信息，包括资源状况、操作状况、连接状况及并发状况。本部分也将叙述如何断开连接。

注意 *仅拥有DBA权限的用户才可以执行该操作。*

监测表

DBMaster将数据库状态存储在四个表中：SYSINFO、SYSUSER、SYSLOCK和SYSWAIT。

SYSINFO表主要包括数据库的系统值：DCCA大小、可利用的DCCA大小、最大事务数以及缓冲页数量，也包括系统活动的统计值，例如事务激活数、启动事务数量、锁和信号量请求数量、物理磁盘I/O次数、日志记录I/O次数等。可通过此表来监测数据库系统状态，并根据表中信息对数据库性能进行调整。

SYSUSER表记录连接信息，其中包括连接ID、用户名、注册名、注册IP地址以及已执行DML操作数量。通过这个表监测目前哪些用户正在使用数据库。

SYSLOCK表记录锁对象信息，例如锁对象ID、锁状态、锁单位、锁定对象的连接ID等。通过此表可监测哪些对象正被连接锁定以及哪些用户正在锁定对象。

SYSWAIT表记录连接等候状态，包括正在等候的连接ID及准备等候的连接ID。使用此表可监测连接并发状态。一旦某个连接处于锁等待状态，而这些锁已经被死锁或空闲连接占用，则可根据此表判断哪个连接正在锁定这些对象，断掉这些空闲或者死连接以释放资源。

浏览四个系统表的方法与普通表相同。

➲ **示例**

通过SQL SELECT命令浏览SYSUSSESER表：

```
dmSQL> SELECT * FROM SYSUSER;
```

了解这4个系统表的更多信息请参考*DBMaster系统目录参考*章节。

断掉连接

当某一连接占用资源并空闲很长时间或急需资源时，连接应被断掉。除此之外，在关闭数据库前所有激活连接都应被断掉。断掉连接前，应先查询SYSUSER表中的连接ID。

➲ **示例1**

断掉用户**Eddie**的连接前，首先要获得连接ID：

```
dmSQL> SELECT CONNECTION_ID FROM SYSUSER WHERE USER_NAME = 'Eddie';


CONNECTION_ID
=============
       352501
```

➲ **示例2**

然后断掉用户Eddie的连接：

```
dmSQL> KILL CONNECTION 352501;
```

18.3 调节I/O

DBMaster中的磁盘I/O需要很多时间。

为了避免磁盘的I/O瓶颈，您可以执行以下操作：

- 确定数据分配
- 确定日志文件分配
- 将日志文件和数据文件存储于不同的磁盘上
- 使用裸设备
- 预分配自动扩展表空间
- 使用I/O及检查点（checkpoint）后台程序

决定数据分配

用户可通过表空间来进行数据划分以避免将所有数据都存放在一起。如果合理使用表空间，DBMaster将会在执行空间管理和表扫描时提高系统性能。一些数据量较小的表可以存放在一个表空间中，但数据量较大的表则应存放在属于它们自己的表空间中。

用户可通过磁盘磁道划分来提高磁盘的I/O速度。磁道是磁盘中的实际分割，因此它可跨越多个磁盘。所以当多进程同时访问相同文件时，可避免磁盘的争夺。

决定日志文件的分配

DBMaster可使用一个或多个日志文件。单一日志文件容易管理，但是使用多日志文件也有很多益处。如果在备份模式下运行DBMaster，并通过备份服务执行增量备份，使用多日志文件能提高增量备份的性能。此外，将多日志文件扩展到不同磁盘上，也可提高磁盘的I/O性能。

用户可根据事务处理的需要，来确定日志文件的大小。然而，如果您的DBMaster运行在备份模式下，并根据日志满状态执行备份，日志的大小将会影响备份的时间间隔，较大的日志文件会增加备份的时间间隔。

分离日志文件和数据文件

将日志文件和数据文件分配到不同磁盘可增加磁盘的I/O性能，在某种程度上允许文件并发的访问。如果磁盘有不同的I/O访问速度，那么应考虑哪些文件应放在更快的磁盘上。通常，如果运行的是联机事务处理（OLTP）应用程序，那么将日志文件放于快速磁盘上。如果运行的是执行较长查询的应用程序，例如决策支持系统，那么应将数据文件放于较快的磁盘上。

使用裸设备

如果DBMaster运行在UNIX系统，可创建裸设备文件来存储数据和日志文件。由于DBMaster具有很好的缓冲机制， 因此通过裸设备读/写要比通过UNIX文件快得多。了解更多创建裸设备的信息，请参考操作系统手册或询问系统管理员。使用裸设备的一个不利因素是DBMaster不能自动扩展表空间，因此在使用裸设备时需制订详细计划。

预分配自动扩展表空间

DBMaster支持自动扩展表空间，可让用户迅速、方便的使用表空间，而不必担心表空间不足的问题。然而如果能预估所需的表空间大小，并在建立表空间的时候就分配适当大小的空间是比较好的方法。因为，这样可以避免花费扩展文件所需的时间。可以在稍后使用 alter file 命令来扩展文件的页数。预分配表空间可避免DBMaster在试图扩展表空间的时候，因为可用的磁盘空间不足而产生磁盘满的错误。

I/O和Checkpoint后台程序

I/O 后台程序

DBMaster的I/O后台程序会周期性地将最近的数据页从页缓冲区中写入磁盘。这可减少在将数据页写入页缓冲区时的溢出，并提高性能。您可以通过**dmconfig.ini**配置文件中的配置参数来控制I/O后台程序。

DB_IOSvr — 开启和关闭I/O后台程序。将关键字设为1，代表开启I/O后台程序；将关键字设为0，代表关闭I/O后台程序。

➲ **示例**

参看以下**dmconfig.ini**中的设置：

```
[MYDB]
...
DB_IOSvr = 1
```

MYDB数据库在DCCA中有400个（**DB_NBufs**）页缓冲。每隔10分钟，I/O程序就按如下步骤执行一次：

- 扫描最少最近使用的页缓冲区
- 在扫描过程中收集所有已写的数据页
- 将收集的已写数据页写入磁盘

CHECKPOINT 后台程序

DBMaster有一个checkpoint后台程序（基于I/O程序）可定期做checkpoint动作。当日志满或启动和关闭数据库时，可减少运行checkpoint等候时间。checkpoint后台程序实际上是一个I/O后台程序的子功能，它可独立执行I/O或checkpoint，或两个操作一起执行。

打开checkpoint后台程序和I/O后台程序需要使用 **DB_IOSrv**关键字。如果I/O程序被激活但没有设置**DB_ChTim**，那么在数据库启动后，它将以默认情况每小时自动做一次checkpoint。

事实上，I/O和检查点后台程序会消耗一些I/O资源。启动数据库服务器后，I/O和检查点后台程序产生的错误信息都会写入**DMERROR.LOG**文件中，警告信息则会被写入**DMEVENT.LOG**文件中。

18.4 调节内存空间

DBMaster将信息临时存储在内存缓冲区中，永久存储于硬盘上。由于从内存中获取数据的时间要比磁盘快的多，所以从内存读取数据将会大大提高性能。每个DBMaster的内存结构大小设置都将直接影响数据库的性能。然而，如果没有足够内存的话，提高性能则无从谈起。

本章的主要内容是调节数据库的内存使用，它包括如何计算所需的 DCCA 大小，如何监测页缓存，日志缓存和系统控制区及分配足够的内存。

➲ **获得最好性能的步骤如下所示：**

1. 调节操作系统
2. 调节DCCA内存大小
3. 调节页缓存大小
4. 调节日志缓存大小
5. 调节系统控制区大小

DBMaster的内存需求变化与应用程序的使用有关；在调节完应用程序和SQL语句后可对内存分配进行相应调节。

调节操作系统

应对操作系统进行适当调整以减少内存交换，并确保系统运行的正常和高效。

在物理内存和磁盘虚拟内存文件之间的内存交换需要花费很多时间。所以拥有足够的物理内存是很重要的。可使用操作系统工具来测量操作系统状态。过高的页交换率说明系统中物理内存不足。当这种情况发生时，需要关闭不必要的进程或者添加更多的物理内存。

调节DCCA内存大小

数据库访问控制区（DCCA）是DBMaster服务分配的一块共享内存。每次DBMaster启动时，服务就会分配和初始化DCCA的大小。

在UNIX的客户机/服务器模式下，DBMaster从Unix共享内存池中分配DCCA，应确保DCCA的大小小于操作系统允许的最大共享内存大小。如果请求DCCA的大小超过了操作系统的限制，可参考操作系统管理员手册来增加最大共享内存的大小。

DBMaster在64位机器上的共享内存大小为2^{31}页（2^{31}×页大小字节），而在32位环境上的共享内存大小为2GB字节。这包括页缓冲大小，日志缓冲大小以及SCA大小。

在64位Linux操作平台上，当需要的共享内存超过2^{31}时，必须设置**shmmax**大小。请注意，必须设置足够的RAM，通过编辑**/etc/sysctl.conf**设置**kernel.shmmax = n**，n指定共享内存的最大值。

➲ **示例**

```
kernel.shmmax = 8405194752.
```

配置DCCA

DCCA包括进程访问控制块，并发控制块以及为数据页、日志块、系统表提供的快速缓冲区。DBMaster维护在DCCA中每个DBMaster进程的并发控制块和访问状态。每个DBMaster进程是通过DCCA中的缓冲存储区来访问同一磁盘数据。

启动数据库前在**dmconfig.ini**中设置适当的参数来配置DCCA的大小。

➲ **示例1**

在**dmconfig.ini**文件中配置DCCA大小：

```
DB_NBuFs = 200

DB_NjnlB = 50

DB_ScaSz = 50
```

DB_NBufs定义数据页缓存的数量（如果设置的数据页大小为4K，每个页缓存为4,096字节）

DB_NJnlB定义日志块缓存的数量（如果设置的数据页大小为4K，每个页缓存为4,096字节）

DB_ScaSz定义SCA页大小（如果设置的数据页大小为4K，每个页4,096字节）

DBMaster只在启动数据库时读取这些DCCA参数。要想调节参数，则必须先终止数据库，修改**dmconfig.ini**配置文件中的值，然后再重启数据库。要想了解更多有关参数信息的设置，请参看*dmconfig.ini中的关键字*章节。

整个DCCA的大小是**DB_NBufs**、**DB_NJnlB**和**DB_ScaSz**三个值的总和。

➲ **示例2**

如果设置的数据页大小为4 K，计算整个DCCA的大小：

```
size of DCCA  = (200 + 50 + 50) * 4 KB
              = 1200 KB
```

为DCCA分配足够内存

DCCA是DBMaster进程频繁访问的资源。因此，应确保足够的物理内存，以避免从DCCA到磁盘间的频繁交换。否则，将会严重降低数据库的性能。页交换率可通过操作系统工具来测量。

➲ **示例**

可从系统表SYSINFO中得到分配DCCA内存的大小：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'DCCA_SIZE'
                                      OR INFO = 'FREE_DCCA_SIZE';


            INFO                              VALUE
```

```
============================ ============================
DCCA_SIZE                    1228800
FREE_DCCA_SIZE               189024
```

DCCA_SIZE — 内存大小，单位为字节。

FREE_DCCA_SIZE — 空闲DCCA内存的大小，单位为字节。

空闲DCCA内存一般保留用于动态控制块，例如锁控制块。

通常大缓存有利于提高系统的性能。然而，如果DCCA内存太大而不适合物理内存的实际大小，系统性能将会降低。因此，配置足够的DCCA内存同时对于物理内存又适合的内存大小是重要的。

调节页高速缓存

DBMaster为数据页缓冲提供了共享内存池。数据页缓冲可以加快数据访问的速度和并发控制。DBMaster默认是自动配置页缓冲数。 在**dmconfig.ini**中，将**DB_Nbufs**关键字设置为0，则代表DBMaster可在系统允许使用的物理内存范围内，自动调节页缓冲数量。在windows 95/98环境下，页缓冲数量不少于500页，在Windows 其他平台或Unix环境下，页缓冲数量不少于2,000页。如果DBMaster无法监测到系统物理内存的使用率，它将会分配最小的内存数。

调节页缓冲区的大小将对系统的性能有很大的影响。下一部分将讲述如何监测缓冲区性能以及计算缓冲命中率。

➲ **提高页缓冲区的性能：**

1. 更新计划对象的统计值
2. 对大表设置NOCACHE
3. 对低聚集的索引重建数据
4. 增大缓冲区的大小
5. 减少checkpoints的影响

监测页缓冲区性能

DBMaster的页缓冲访问统计信息存储于SYSINFO系统表。

➲ **示例**

使用如下SQL语句获取缓冲区值：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_PAGE_BUF';

             INFO                           VALUE
================================ ================================
NUM_PAGE_BUF                    4000

1 rows selected

dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_PHYSICAL_READ'
    2>                                   OR INFO = 'NUM_LOGICAL_READ'
    3>                                   OR INFO = 'NUM_PHYSICAL_WRITE'
    4>                                   OR INFO = 'NUM_LOGICAL_WRITE';

             INFO                           VALUE
================================ ================================
NUM_PHYSICAL_READ               64
NUM_PHYSICAL_WRITE              1
NUM_LOGICAL_READ                509
NUM_LOGICAL_WRITE               0

4 rows selected
```

NUM_PAGE_BUF — 使用的数据缓冲页数

NUM_PHYSICAL_READ — 从磁盘读取的页数

NUM_LOGICAL_READ — 从快速缓存区读取的页数

NUM_PHYSICAL_WRITE — 写入磁盘的页数

NUM_LOGICAL_WRITE — 写入快速缓存区的页数

根据如下公式计算页缓冲的读/写命中率：

$$\text{read hit ratio} = 1 - (\frac{\text{NUM\_PHYSICAL\_READ}}{\text{NUM\_LOGICAL\_READ}})$$

$$\text{write hit ratio} = 1 - (\frac{\text{NUM\_PHYSICAL\_WRITE}}{\text{NUM\_LOGICAL\_WRITE}})$$

对于上面的例子，可计算读/写命中率：

$$\begin{aligned}\text{read hit ratio} &= 1 - (\frac{13207}{331595}) \\ &= 0.960 \\ &= 96.0\%\end{aligned}$$

$$\begin{aligned}\text{write hit ratio} &= 1 - (\frac{7361}{127423}) \\ &= 0.942 \\ &= 94.2\%\end{aligned}$$

根据读/写命中率，可决定如何提高快速缓存区性能。如果命中率太低，可根据以后部分介绍的方法调整DBMaster。

如果命中率一直很高，例如高于99%，说明缓冲区的空间足够容纳所有频繁使用的页。在这种情况下，可减少缓冲区大小以便为其它应用程序留出内存空间。因此，为了确保运行性能良好，需要在进行任何修改前后监测缓冲区性能。

过时的统计值

如果读/写的命中率很低，这可能是因为计划对象（表、索引、字段）的统计值太旧了。错误的统计值会导致DBMaster的优化器制定出低效的SQL查询计划。如果用户在最近一次更新统计值之后，插入了大量的数据，那么就要对统计值进行再次更新。

➲ **示例1**

更新所有计划对象的值：

```
dmSQL> UPDATE STATISTICS;
```

如果数据库非常大，那么更新所有对象的统计值将会花费很长的时间。一个可选的方法是根据最近一次更新统计以来所做的修改，来更新特定计划对象的统计值，并设置采样率。

➲ **示例2**

更新特定的计划对象：

```
dmSQL> UPDATE STATISTICS tabel1, table2, user1.table3 SAMPLE = 30;
```

在成功更新计划对象统计值后，监测页缓冲区的性能。监测方法可参考*监测页缓冲*。

缓冲区交换

DBMaster通过最近最少使用算法（Least Recently Used (LRU)）来决定哪一个页缓冲需要交换出去。这就可以保证一些很少访问的页交换出，而经常访问的存在页缓存中。然而，如果浏览一个很大的表，为了完成表扫描有可能将所有的页缓冲都交换出去。

例如，在一个拥有200个页缓冲的数据库中，如果浏览的表有250页，DBMaster可能会将250页全部读到页缓冲中，并放弃200个频繁访问的页。最坏的情况是，一个完全表扫描之后又读取其他数据，DBMaster必需从磁盘读200页。然而，如果这个表的缓冲模式为NOCACHE模式，当执行一个表扫描时DBMaster会将取得的页放在LRU链的末端。因此，在200页中频繁使用的199页将仍会保留在缓冲区中。

通常超出了页缓冲数量的表应该设置成NOCACHE模式。不经常使用或者页数量接近于页缓冲数量的表也应设置为NOCACHE模式。

➲ **示例1**

获得表的页数和缓冲模式：

```
dmSQL> SELECT TABLE_OWNER, TABLE_NAME, NUM_PAGE, CACHEMODE FROM
SYSTEM.SYSTABLE WHERE TABLE_OWNER != 'SYSTEM';


TABLE_OWNER     TABLE_NAME      NUM_PAGE   CACHEMODE
=========== ================= =========== =========
BOSS        salary                      5  T
MIS         asset                      45  T
MIS         department                  3  T
MIS         employee                   29  T
MIS         worktime                  450  T
TRADE       customer                  350  T
TRADE       inventory                 167  T
TRADE       order                     112  T
TRADE       transaction              1345  F


9 rows selected
```

NUM_PAGE — 表中页的数量。

CACHEMODE — 表扫描缓冲模式，'T'代表扫描为缓冲模式，'F'代表扫描为非缓冲模式。

在上例中，将表**TRADE.transaction**设置成NOCACHE模式。其它表仍然是可缓冲的。如果有200个页缓冲，**MIS.worktime**和**TRADE.customer**表应该被设置成NOCACHE模式，**TRADE.order**和**TRADE.inventory**表如果很少被用到，也应该设置成NOCACHE模式。

➲ **示例2**

为表设置 NOCACHE 模式：

```
dmSQL> ALTER TABLE MIS.worktime SET NOCACHE ON;
```

如果表中存在无用的索引或查询参考了无索引的字段，DBMaster会执行一个全表扫描。为了阻止这种情况的发生，您应该尽量将SQL语句写的更有效，并尽可能的使用有索引字段。

记录聚集性能低

如果想通过索引键读取排序的大量记录，或者通过谓词参考一个索引字段时，索引聚集就变成了一个影响缓冲性能的重要因素。

➲ **示例1**

从**tb_customer**表中选择所有字段，并通过custid主键排序：

```
dmSQL> SELECT * FROM tb_customer ORDER BY custid;
```

假设表**tb_customer**有3,500笔记录，分布在350页中，且系统有200页缓冲区。 如果记录是以custid为聚集，而且聚集的非常好（循序分布在所有的页上），那么DBMaster只需从磁盘读取350页就可以完成这个命令。但是，如果聚集不好（非循序的资料放在相同的页上），在最坏的情况下，DBMaster可能需要从磁盘读取3,500页（每一个记录需要一个磁盘读取）才能完成这个命令。

➲ **示例2**

在**tb_customer**表的custid字段上创建一个**custid_index**索引。

```
dmSQL> SELECT CLSTR_COUNT FROM SYSTEM.SYSINDEX
                          WHERE TABLE_OWNER = 'TRADE'
                          AND TABLE_NAME = 'tb_customer'
                          AND INDEX_NAME = 'custid_index';
```

结果：

```
CLSTR_COUNT
```

```
===========

        385


1 rows selected
```

CLSTR_COUNT — 聚集数（cluster count），使用一些缓冲区，通过完整索引扫描来读取数据页数。当完整扫描表customer，并将结果以字段custid做排序时， DBMaster最多会从磁盘中读取385页。

➲ **示例3**

获取页及行数：

```
dmSQL> SELECT NUM_PAGE,NUM_ROW FROM SYSTEM.SYSTABLE
                               WHERE TABLE_OWNER = 'TRADE'
                               AND TABLE_NAME = 'tb_customer';
```

结果：

```
NUM_PAGE    NUM_ROW
=========== ===========
        350        4375


1 rows selected
```

NUM_PAGE — 表配置的页数

NUM_ROW — 表记录数

可通过CLSTR_COUNT、NUM_PAGE 和 NUM_ROW三个参数来估计聚集因子，公式如下：

$$\text{clustering factor} = \frac{(\text{CLSTR\_COUNT} - \text{NUM\_PAGE})}{\text{NUM\_ROW}}$$

上面的例子中，聚集因子为0.8%。

$$\text{clustering factor} = \frac{(385 - 350)}{9375}$$
$$= 0.0017$$
$$= 1.7\%$$

聚集因子的值界于0到100%之间。在上例中，CLSTR_COUNT 只比 NUM_PAGE 小一点，可视为0。如果聚集因子为0，说明数据完全聚集在这个索引中。如果聚集因子太高，例如超过了20%（如何决定高低，取决于表大小、平均记录大小等等），则索引会有较差的聚集。当DBMaster发现一个索引有较差的聚集时，当执行一个SQL指令时，DBMaster优化器会使用完全表扫描，而非预期的索引扫描。

➲ **对频繁使用的索引提高聚集索引的性能：**

1. 载出所有的表数据（根据索引排序）
2. 重排载出的数
3. 删除表索引
4. 删除所有表数据
5. 重载表中的数据
6. 重建表中的索引

在数据重新加载之后，索引应该是完整聚集。然而，您应该注意一个表格只能聚集在一个索引，如果一个表格有许多的索引，您应该在最重要的索引上维持索引聚集。通常，最重要的索引是主键。因为载出和重新加载资料要花费许多的时间和储存体，您应该在表格非常大而且浏览非常频繁的情况下再调整索引聚集。

数据页缓存

如果为数据库访问分配的数据页缓存不足，那么需要在DCCA中增加页缓存。

➲ **修改数据页缓存数：**

1. 终止数据库服务
2. 在**dmconfig.ini**文件中重新设置**DB_NBufs**的值，增加页缓存数
3. 重启数据库

在成功的加大数据页缓存区之后，使数据库运行一段时间，再次监测缓存区性能。如果缓存区的命中率增加，则说明增加页缓存区提高了性能。如果不是，必须再增加更多的页缓存，或是检验其它使系统性能降低的原因。

检查点发生频繁

如果写命中率比读命中率低很多，原因可能是检查点发生的太频繁。

当检查点发生时，DBMaster会将所有修改过的页缓冲区写入到磁盘。 因为检查点需要许多的CPU时间，所以可定义一个检查点时间计划来周期性的执行检查点后台程序。周期性运行检查点的好处是当数据库系统遇到系统损坏时，可以减少重启数据库的灾难恢复时间。

除了通过后台程序周期性的运行检查点外，当DBMaster在NON-BACKUP模式日志空间用尽时，或在BACKUP模式执行增量备份的时候， 也会自动的执行检查点。可加大日志大小来增加自动执行检查点的时间间隔，以改善检查点发生太频繁的问题。

➲ **示例**

下例说明了有多少个检查点已被执行：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_CHECKPOINT';

              INFO                              VALUE
============================== ==============================
NUM_CHECKPOINT                  26

1 rows selected
```

再度监测高速缓存性能

使用上述方法调整系统后，应该重新监测高速缓冲区的性能。

➲ **监测快速缓存性能：**

1. 运行数据库一段时间，确认数据库信息已处于稳定状态。
2. 使用下列的SQL指令清除SYSINFO系统表中的统计值。

```
dmSQL> SET SYSINFO CLEAR;
```

3. 运行数据库一段时间。
4. 从SYSINFO表得到读/写计数值，并且检查其命中率。

调节日志缓存

DBMaster用日志缓冲区储存最近用到的日志区块。当日志缓冲区足够的情况之下，更新资料时写入磁盘的时间和回滚事务时从磁盘读取的时间都会减少。如果你很少执行长事务来修改（插入、删除、更新）许多记录，可略过这一节，否则应确定系统是否有足够的日志缓存区。最佳的日志缓存区数目是同一时间内最长事务所需要日志区块的总和。

➲ **您可以通过下列步骤来预测日志缓冲区的大小：**

1. 确定数据库只有一个连接用户。
2. 使用下列指令清除表SYSINFO的计数器。

```
dmSQL> SET SYSINFO CLEAR;
```

3. 执行更新最多记录的事务。
4. 执行下列SQL指令测量使用的日志区块数。

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO =
'NUM_JNL_BLK_WRITE';

             INFO                              VALUE

============================== ==============================
```

```
NUM_JNL_BLK_WRITE                626



1 rows selected
```

注意 *NUM_JNL_BLK_WRITE—事务处理使用的日志区块。在例子中，日志区块大小是512字节。 在上例中，您大约需要41个日志缓冲页（如果设置的数据页大小为4K）。*

另一个衡量日志缓冲区使用情况的是：日志缓冲区写出率（flush rate）。日志缓冲区写出率是当DBMaster写出日志时，日志缓冲区写出到磁盘的百分比。如果日志缓冲区的写出率太高（例如超过50%），应该增加日志缓冲区的数量。

➲ **示例**

估计日志缓冲区写出率的大小：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_JNL_BLK_WRITE'
    2>                                OR INFO = 'NUM_JNL_FRC_WRITE';

             INFO                              VALUE
=============================== ===============================
NUM_JNL_BLK_WRITE               41438
NUM_JNL_FRC_WRITE               159


2 rows selected
```

NUM_JNL_BLK_WRITE — 日志块写到缓冲区的数量

NUM_JNL_FRC_WRITE — 日志缓冲强制写出到磁盘的次数

假设**DB_NJnlB**设置为50页（例如：有400 个日志缓冲区）。在下例中，日志缓冲区写出率（0.65）有些偏大，所以您必须添加日志缓冲区的大小以提高日志缓冲区的执行性能。

$$\text{journal flush rate} = \frac{(\text{NUM\_JNL\_BLK\_WRITE}/\text{NUM\_JNL\_FRC\_WRITE})}{(\text{DB\_NJNLB}\times 8)}$$
$$= \frac{(41438/159)}{(50\times 8)}$$
$$= 0.65$$

调节系统控制区（SCA）

当数据库启动时，缓冲区和一些控制区块（例如连接和事务信息）已经从DCCA事先配置而且大小固定。但是一些并行控制区块是在数据库执行的时候才动态的从DCCA中配置的。 这些控制区块的大小是由**dmconfig.ini**中的**DB_SCASZ**所指定。

如果数据库应用程序得到的错误讯息为：“数据库需要的共享内存超过了原先的设置”，表示DBMaster不能从SCA区动态分配内存。通常，这样的错误是由于长事务用了太多的锁。如果这种情形频繁发生，可用下面的方法解决。

避免长事务

长事务会占用过多的锁控制块和日志块。如果一个长事务在执行过程中，发生了上述错误，应该分析能否将事务分割成许多较小的事务。

避免大表上设置过多的锁

如果在一个大表中使用索引扫描来选取多条记录时，将需要许多的锁资源。为了减少事务使用太多的锁资源，可在表扫描之前升级锁定模式。

例如，如果表的默认锁定模式是行，则将这些锁升级为页级锁或表级锁，这样就可以减少事务占用太多的锁资源。它的缺点是在减少资源使用的同时，会在某些程度上牺牲并行的能力。

增加SCA的大小

如果上述两种情况都没有发生，可以增加SCA的大小。您可以在**dmconfig.ini**中增大参数**DB_ScaSz**的值，并且重启数据库。

调整日志缓存

DBMaster储存系统缓存在SCA中。如果很少修改结构对象，可以打开turbo模式（在**dmconfig.ini** 设置**DB_Turbo** = 1）。当打开turbo模式，DBMaster将会延长系统缓存生存期。这将会增加联机事务处理（OLTP）的性能。

18.5 调整并发进程

在多用户数据库系统中，当一个以上的进程同时存取相同数据库资源时，将发生资源竞争。这种情形也能引起死锁，即当两个或更多的进程彼此互相等待对方资源时的状况。资源竞争会导致进程的互相等待，并降低系统性能。

DBMaster提供下列的方法监测和减少资源竞争。

- 减少锁竞争。
- 限制进程数。
- 设置CPU亲和性
- 设置事务优先级

减少锁竞争

从数据库存取数据时，DBMaster进程将自动锁定目标对象（记录、页、表）。当两个进程想要锁定同一对象时，其中一个必须等待。当两个以上的进程互相等待另一个进程释放锁时，就会发生死锁。死锁发生时DBMaster将牺牲最后一个事务，并回滚事务来解决死锁问题。死锁会降低系统性能，经常监测锁统计值以避免死锁发生。

➲ **示例1**

查看死锁统计值：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_LOCK_REQUEST'
    2>                                   OR INFO = 'NUM_DEADLOCK'
    3>                                   OR INFO = 'NUM_STARTED_TRANX';

             INFO                            VALUE
```

```
================================ ================================
NUM_STARTED_TRANX                33

NUM_LOCK_REQUEST                 173

NUM_DEADLOCK                     0


3 rows selected
```

NUM_LOCK_REQUEST — 锁请求次数

NUM_DEADLOCK — 死锁发生次数

NUM_STARTED_TRANX — 总事务次数

在上面的例子可以看出，平均来讲每51（9287/181）个事务会发生一个事务死锁，一个事务大约需要有83（772967/9287）个锁。

如果死锁发生的频率太高，应该检查表结构设计、SQL命令和应用程序。将锁模式预设到较低的锁模式（例如行锁）可以减少锁竞争，但是这需要更多的锁资源。

如果不需要在某一时间点读取数据之后继续保持数据的一致性，那么可使用另一个方法，用浏览模式（browse mode）来读取表。这种情况主要用于只想看看数据或使用数据来执行计算，但并不打算更新数据。它提供了在一个特定的时间点，对于需要的数据做快照。使用浏览模式，可增强并行性，并消耗较少的锁资源。因为在数据读取之后，锁就会释放。

限制进程数

DBMaster允许在同一时间最多有4800个与服务器间的连接。但是如果服务器资源（例如内存、CPU能力）不足，则可限制最大的连接数来避免资源竞争。配置参数**DB_MaxCo**设置数据库中的最大连接数。

当数据库在最初创建时，日志文件会根据特定连接数而格式化。日志文件将会为每个连接保存事务信息。依据日志文件的可利用连接数就是所谓的*硬连接数*。在数据库创建时，这个值是由**DB_MaxCo**决定的。 **硬连**

接数的最小取值为 240，最大取值为4840，并且必须为40的倍数，如果**DB_MaxCo**关键字设置的值不是 40 的倍数，那么硬连接数将会根据**DB_MaxCo**的值取一个接近于40倍数的值。硬连接数是一个日志文件的限制，因此，改变它的值，必须重新设置关键字**DB_MaxCo**并且以新日志模式（**DB_SMode=**2）重启数据库。

以下公式用来计算*硬连接数*：

硬连接数=(**DB_MaxCo**+保留连接数+40-1)/40*40

目前，DBMaster拥有20个保留连接来支持内部连接，比如后台程序、管理员用等等。

硬连接数不会直接影响DCCA的大小，而主要是根据*软连接数*决定。更大的连接数表示在服务器端和客户端需要更多的内存，因此即使数据库的硬连接数很大，用户也可以使用较小的软连接数来启动数据库以节省内存。数据库启动时的软连接数取决于**DB_MaxCo**的值。软连接数决定DCCA支持的连接数，并影响到DCCA占用内存的大小。软连接的值必须小于或等于硬连接的值。要改变软连接数，需在改变了**DB_MaxCo**值后正常模式重启数据库。

以下公式用来计算*软连接数*：

软连接数 = **DB_MaxCo** + 保留连接数

➲ **示例1**

下面的配置文件中，**DB1**的硬连接数为240，数据库**DB2**的硬连接数为1120。

```
[DB1]
DB_MaxCo = 50   ;; the hard connection number is 240
                ;; the soft connection number is 70
[DB2]
DB_MaxCo = 1100         ;; the hard connection number is 1120
                ;; the soft connection number is 1120
```

➲ **示例2**

成功启动数据库后，**DB1**的硬连接数将会变为320。

```
[DB1]
DB_SMode = 2            ;; startup with new journal file
DB_MaxCo = 280                  ;; the new hard connection number is 320
```

➲ **示例3**

假设**DB2**已经如例1中所示的被创建，下面**dmconfig.ini**文件中的设置将会导致硬连接数为1120，软连接数为40。

```
[DB2]
DB_SMode = 1            ;; normal start
DB_MaxCo = 20           ;; the new soft connection number is 40
```

设置CPU亲和性

为了提高多任务的执行性能，操作系统将进程和线程分配给不同的CPU。尽管从操作系统的角度来看该行为非常有效。但是当系统负载繁重时，每个处理器会多次缓存以重载数据，那么该行为将会降低DBMaster的性能。若要提高DBMaster性能，需使进程运行于它最后一次运行的CPU。此时，用户可先通过查询系统表SYSUSER获得进程亲和性，然后使用系统存储过程SETAFFINITY设置新的亲和性掩码，这样进程将仅运行于指定的CPU。

不同于旧操作系统，现代操作系统具有CPU亲和性的特征，例如：Linux2.6、Winsows NT和Windows 98。CPU亲和性包括用户可以更改的软亲和性和用户无法更改的硬亲和性。

CPU亲和性由亲和性掩码定义，该位矢量的每一位均代表一个处理器。DBMaster将亲和性掩码定义为char(64)，因此最多可存在64个CPU。

➲ **示例1**

下例是8-CPU系统的亲和性掩码（高位连续的0省略）：

十进制值	二进制掩码	允许运行的CPU
1	'1'	0
3	'11'	0和1
7	'111'	0、1和2
15	'1111'	0、1、2和3
31	'11111'	0、1、2、3和4
63	'111111'	0、1、2、3、4和5
127	'1111111'	0、1、2、3、4、5和6
255	'11111111'	0、1、2、3、4、5、6和7

使用GETCPUNUMBER和SETAFFINITY，用户可以获得当前系统状态并为连接设置CPU亲和性，无需在运行时重启DBMaster。

GETCPUNUMBER原型如下：

```
GETCPUNUMBER (INT CPU_NUMBER OUTPUT)
```

CPU_NUMBER：输出参数，机器逻辑处理器个数。

SETAFFINITY原型如下：

```
SETAFFINITY (INT CONNECTION_ID INPUT, CHAR(64) AFFINITY_MASK INPUT)
```

CONNECTION_ID：输入参数、连接或服务器ID。用户可使用" select connection_id from sysuser"或通过检测系统监测获得该ID。Windows系统下，该ID是线程ID，在类Unix系统下，该ID是进程ID。

AFFINITY_MASK：输入参数、CPU亲和性掩码。有效的亲和性掩码由0和1组成。1表示该CPU对连接有效，0表示该CPU对连接无效。

➲ **示例2**

设置CPU亲和性之前，用户需获得相关系统信息，例如服务器的CPU个数、每个连接的CPU使用、正确的亲和性掩码。

调用GETCPUNUMBER获得CPU个数：

```
dmSQL> CALL GETCPUNUMBER(?);
```

获得每个连接的CPU使用以及正确的亲和性掩码：

```
dmSQL> SELECT connection_id, affinity_mask, priority_level, cpu_usage
FROM sysuser;
```

为sysuser设置CPU亲和性，允许连接运行于0号和1号CPU：

```
dmSQL> SELECT connection_id, user_name FROM sysuser;


    CONNECT*              USER_NAME
================    ======================
           30420    BACKUP_SERVER
           30418    SYSADM


2 rows selected
dmSQL> CALL SETAFFINITY(30418,'11');
```

注意　*仅拥有SYSADM权限的用户可调用SETAFFINITY。*

查询sysuser获得特定连接的CPU亲和性掩码：

```
dmSQL> SELECT affinity_mask FROM sysuser WHERE connection_id = ?;
```

设置事务优先级

进程和线程的优先级是多任务操作系统的基本特征，调度程序会检测系统负载并在需要的时候更改事务优先级。DBMaster支持用户通过设置连接优先级提高整个系统的性能。例如，有一个长事务占用大部分CPU时间，导致其他连接等待的时间过长。此时，如果用户降低该长事务的优先级，其余连接将获得更多CPU时间，整个系统的性能也会随之提高。用户可为一些重要连接设置新优先级从而提高该连接性能，同时其它连接性能将会降低。

使用GETCPUNUMBER和SETAFFINITY，用户可以获得当前系统状态并为连接设置优先级，无需在运行时重启数据库。

SETPRIORITY原型如下：

```
SETPRIORITY (INT CONNECTION_ID INPUT,INT PRIORITY_LEVEL INPUT)
```

CONNECTION_ID：输入参数，连接或服务器ID，用户可使用"select connection_id from sysuser"或通过检测系统监测来获得该ID。Windows系统下，该ID是线程ID，在类Unix系统下，该ID是进程ID。

PRIORITY_LEVEL：输入参数，分为5个优先级：1、2、3、4、5，普通和默认优先级均为3。1表示最低优先级；2表示较低优先级；3表示普通优先级；4表示较高优先级；5表示最高优先级。

注意 *Unix系统下，用户仅能降低事务优先级，但不能升高优先级，因为优先级的升高需要用户拥有root权限。Windows系统下无此限制。*

➲ **示例**

设置CPU亲和性之前，用户需获得相关系统信息，例如服务器的CPU个数、每个连接的CPU使用、优先级。

调用GETCPUNUMBER获得CPU个数：

```
dmSQL> CALL GETCPUNUMBER(?);
```

获得每个连接的CPU使用及优先级：

```
dmSQL> SELECT connection_id, affinity_mask, priority_level, cpu_usage
FROM sysuser;
```

为sysuser设置优先级：

```
dmSQL> SELECT connection_id , user_name FROM sysuser;


    CONNECT*              USER_NAME
===============   ======================
          30420    BACKUP_SERVER
```

```
          30418    SYSADM


2 rows selected
dmSQL> CALL SETPRIORITY(30418,3);
```

注意　　*仅拥有SYSADM权限的用户可调用SETPRIORITY。*

查询sysuser获得特定连接的优先级：

```
dmSQL> SELECT priority_level FROM sysuser WHERE connection_id = ?;
```

19 查询优化

本章将介绍DBMaster的查询优化器。所谓的查询优化器是选择最佳的内部执行方法，使SQL命令的查询能够更快、更有效。

本章的内容将包含下列的问题：

- 什么是查询优化？为什么需要查询优化？当了解了查询优化的目的，就能够知道查询优化在执行一个查询中所扮演的角色。
- 什么是查询执行计划（QEP）？如何读查询执行计划？在了解查询执行计划时，可以知道DBMaster如何执行一个查询。
- 优化器如何运作？当理解了对一个查询执行计划，优化器搜索的方式，就能通过重写一个等效的SQL查询来帮助优化器找出更高效的执行计划。
- 什么是成本函数？在查询时，了解了一个操作需要多少的执行时间，就可以知道优化器如何选择适当的操作，而且可以使用一些DBMaster提供的命令，帮助优化器选择更好的操作。
- 什么是统计值？统计值有何用处？在了解查询优化统计值的使用时，就能明白为什么优化器选择一个特定执行计划的原因。
- 如何加速查询的执行？当知道如何写一个有效的查询时，就能够改写查询命令，来提高查询的执行效率。

19.1 什么是查询优化

查询优化对于数据操作语言（DML）（SELECT、INSERT、DELETE、UPDATE）而言是一个相当重要的步骤。对任何一个SQL查询，DBMaster都有可能存在多个不同的执行方法，DBMaster查询优化的目的，就是要在这些执行方法中挑选出一个最好，最有效的执行方法。查询优化最主要的工作，是决定每一个操作和它们之间的执行顺序。

➲ **发现最有效的操作：**

1. 从一个表读取数据 — 可以顺序或者索引扫描读取。
2. 合并表 — 可以用嵌套循环合并或排序合并。
3. 排序 — 什么时候需要排序？在一个操作前还是操作后，或者避免排序。

查询优化器为了优化参与合并表的顺序，必须估计出外合并影响的行数。当执行查询时，一些数据库用户对数据特征的熟悉程度甚至超过了查询优化本身。

DBMaster查询优化器将估计所有可能的执行计划，计算行数，磁盘页I/O的数量，对单表花费的CPU时间。从上面这些因素来发现一个最低成本的计划。

DBMaster搜索一个查询计划时，考虑的一些主要操作如下：

- ***Table scan*** — 也称表扫描，从一个数据库的数据页中按顺序得到每一行
- ***Index scans*** — 也称索引扫描，参照索引页指向数据页的地址
- ***Nested join*** — 也称嵌套合并，一行行比较两个或多个表来实现合并
- ***Merge join*** — 也称排序合并，先排序两个表，再一行行比较来实现合并

- ***Sort*** **—** 执行排序
- ***Temporary table*** **—** 在查询过程中，建立一个临时表。

➲ **示例1**

使用ORDER BY子句来排序：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE
tb_salary.basepay=3000 AND tb_staff.id = tb_salary.id ORDER
BY tb_staff.name;
```

➲ **示例2**

查询执行计划1：

```
sort tb_staff.name
     merge join tb_staff.id = tb_salary.id
        index scan tb_staff on idx_ID(id)
        sort tb_salary.id
           table scan tb_salary, filter tb_salary.basepay=3000;
```

➲ **示例3**

查询执行计划2：

```
nested join
     index scan tb_staff on idx_name(name)
     table scan tb_salary, filter tb_salary.basepay and tb_staff.id =
tb_salary.id;
```

19.2 如何进行查询优化操作

当DBMaster的优化器执行查询优化时，将会遵循以下规则：

- 分析查询，将where谓词分解成多个因子。
- 搜寻所有可能的执行顺序和合并顺序。
- 决定是使用嵌套合并还是排序合并。
- 决定是使用表扫描还是索引扫描。
- 决定排序顺序。

优化器处理输入

优化器的成败的关键因素在于估计的准确度，然而优化器通常只有有限的信息来对一个操作估计所需的时间，和真正执行查询的时间相比仅占一小部分。优化器需要的所有信息是来自系统表，使用UPDATE STATISTICS命令更新统计值以确保信息的可用性且没有过期。要想了解更多信息，请参考19.4章*统计值*。

系统表中的数据包括：

- 表中的行数。
- 一个表占用的数据页。
- 一笔记录平均占用的字节数。
- 一个字段平均占用的字节数。
- 每一个索引字段的不重复值。
- 每一个字段的第二大和第二小的值。之所以不取最大和最小值是为了避免特殊大和小的值影响到准确度。
- B型树（B-tree）索引占据的的索引页数。

- B型树索引的层数。
- B型树索引的叶子页数（leaf page）。
- B型树索引的聚集数。

优化器利用这些信息的前提是数据是均匀分布的。如果数据分布不均匀或不统一，那么优化器将可能选择较差的计划。

因子

优化器的首要工作是检测 WHERE 子句中的所有表达式，并将这些表达式分解成几个较小的，彼此独立的表达式，我们将这些彼此独立的表达式称为因子。

➲ **示例1**

优化器将WHERE子句分解成两个因子**tb_staff.id = tb_salary.id** 和 **tb_salary.basepay=3000**：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id =
tb_salary.id AND tb_salary.basepay=3000;
```

➲ **示例2**

根据WHERE子句，仅有一个因子**tb_staff.id = tb_salary.id**或 **tb_salary.basepay=3000**：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id =
tb_salary.id OR tb_salary.basepay=3000;
```

➲ **示例3**

根据WHERE子句，有两个因子**tb_staff.id = tb_salary.id**和 **tb_salary.basepay=3000** 或者 **tb_staff.name='joy'**：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id =
tb_salary.id AND (tb_salary.basepay=3000 OR tb_staff.name='joy');
```

➲ **示例4**

根据WHERE子句，仅有一个因子**tb_staff.id = tb_salary.id**或**tb_staff.name='joy'**：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id =
tb_salary.id AND tb_salary.basepay=3000 OR tb_staff.name='joy';
```

根据上面的例子，可看出表达式包括二元操作**and**时可被分割成不同的因子。 然而如果包含的是**or**则不能被分割成多个因子。

为了找出因子，优化处理器还必须估计每一个因子的选择率。所谓的选择率就是每一个因子可以过滤的数据比率，它的值介于0和1之间。表tb_staff有100行。

➲ **示例5**

查询tb_staff有5笔记录，则因子**tb_staff**, **tb_staff.id = 3**的选择率是5 / 100也就是0.05。

```
dmSQL> SELECT * FROM tb_staff WHERE tb_staff.id=3;
```

如果表达式中的因子超过一个，因为这些因子之间是彼此独立的，所以这个表达式的选择率就是这些因子的乘积。

Join次序

Join次序是指执行合并时，原始表被访问的顺序。不同的Join次序将产生不同的执行顺序和不同的执行时间，但无论如何都会得到正确的执行结果。

➲ **示例1**

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id =
tb_salary.id;
```

查询执行计划1：

```
nested join
     table scan tb_staff
     table scan tb_salary, filter tb_staff.id = tb_salary.id;
```

查询执行计划2：

```
nested join
     table scan tb_salary
     index scan tb_staff on tb_staff (id), filter tb_staff.id =
tb_salary.id;
```

➲ **示例2**

```
dmSQL> SELECT * FROM tb_staff, tb_salary,tb_dept WHERE tb_staff.id =
tb_salary.id AND tb_salary.id tb_salary.id=tb_dept.id;
```

从这个查询中可以看出，共有3！(＝6)种Join次序。这些可能的Join次序是：

```
(tb_staff, tb_salary), tb_dept
(tb_staff, tb_dept), tb_salary
(tb_salary, tb_staff), tb_dept
(tb_salary, tb_dept), tb_staff
(tb_dept, tb_staff), tb_salary
(tb_dept, tb_salary), tb_staff
```

DBMaster将搜索所有这些Join次序，计算它们的成本，并选择出最佳的顺序。

嵌套合并和排序合并

嵌套合并与排序合并是DBMaster的两种合并方法：

- 嵌套合并是通过两层以上的嵌套循环来达到合并的功能。算法分析的时间复杂度是**$O(n^2)$**。
- 排序合并则是二个表必须先经过排序，再将排序过的表一行行依次来做合并。它的时间复杂度是**$O(n \times log(n))$**，对于经过排序的数据，执行合并的时间复杂度是**$O(n)$**。排序合并只能用于对等合并。

以时间复杂度来看，排序合并比嵌套合并好。但是在某些情形，比方说当两个表的数据行数很悬殊的时候，嵌套合并就会比排序合并的执行性能高。无论如何，优化器决定使用嵌套合并或排序合并主要取决于成本函数和统计值。

表扫描和索引扫描

表扫描指的是顺序读取表的所有记录，一行接一行。例如用户需要的数据是从一个表中读取符合age>50的所有记录，表扫描从每一个数据页顺序的读出每一笔记录，再对比找出符合条件的记录。

索引扫描指的是在表的某些字段上建立索引，通过索引指到真正数据页的位置，由此读取所需的数据。DBMaster使用的索引方法是B型树，使用索引扫描的先决条件是必须先在这个表上建立索引。

排序

查询优化器的另一个重要问题是决定如何执行排序，在合并之前还是合并之后，或是可以避免做排序。

➲ **示例**

创建排序：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id=tb_salary.id
ORDER BY tb_staff.basepay;
```

查询执行计划1，优化器在合并后执行排序：

```
sort tb_staff.basepay
      merge join tb_staff.id=tb_salary.id
           index scan tb_staff on idx_id(id)
           sort tb_salary.id
               table scan tb_salary;
```

查询执行计划2，优化器在合并前执行排序：

```
nested join
```

```
    index scan tb_staff on idx_base(basepay)
    table scan tb_salary, filter tb_staff.id=tb_salary.id;
```

19.3 查询的时间成本

从磁盘读取数据的时间和对比字段值的时间是执行查询的两个主要部分。

CPU成本

数据库服务必须在内存中处理数据，它必须读取一条记录到内存，才能够用过滤表达式来测试。它需要先加载两个表的数据到内存，才能够测试它们的合并条件。另外数据库服务必须在内存中从每一条记录中，搜集被选到的字段。使用like和match这样的关键词中包含有通配符时，排序会占用更多的时间。

I/O成本

从磁盘中读取一条记录所需的时间比在内存中检查一条记录的时间要长。因此，优化器主要目的之一就是减少数据的磁盘I/O次数。

数据库服务处理的磁盘储存器的单位称为页，一页由磁盘空间的簇块组成，它的大小和数据库服务有关。DBMaster可提供大小为4 K、8K、16K或32K的数据页。一个页可以容纳的记录和一条记录的大小有关，在数据页大小为4K的情形下，一个数据页可以容纳10到100笔的记录。另外一个索引页实体包含一个键值和4个字节的指针，因此一个索引页可以容纳100到1000实体。

数据库服务需要有一组内存用于存放读取的磁盘页的拷贝以供处理。因为内存限制，在某些情况，这些数据页可能会再次的被读取，我们称这些内存为页缓冲区。如果需要存取的页刚好就在页缓冲区中，数据库服务就不需要再次的从磁盘中读取，这种情况将会提高效率。磁盘页的大小和页缓冲区的数目则取决于数据库服务和操作系统。

缓冲区是下列因素的组合：

- **Buffers –** 需要的页可能已经在页缓冲区，在这种状况下，存取的成本几乎可以忽略不计。
- **Contention –** 如果有一个以上的应用程序试图使用磁盘等硬设备，这个时候，数据库服务的请求将会延误。
- **Seek time –** 搜索时间是磁盘中最耗费时间的动作，它是指移动读写头到需求数据的位置而花费的时间，它和磁盘的速度以及磁盘读写头所在的起始位置有关。搜索时间的变动也是相当的大。
- **Latency time –** 或称旋转延误时间，它和磁盘速度以及读写头位置有关。

表扫描成本

扫描所有表中数据所花费的时间。不管查询是否有过谓词，都需要对所有的数据页信息逐一对比，所以表扫描的成本就是数据页页数。

索引扫描成本

索引扫描是通过B型树索引页来读取数据的。索引扫描又可分为二种：一种是通过参照B型树来读取数据页中的数据，另一种则是直接从索引叶读取所需的数据，我们称之为叶扫描。

➲ **示例**

表**tb_staff**包含两个字段：**id**和**name**，在**id**上建立索引：

```
dmSQL> SELECT * FROM tb_staff WHERE id > 0;
```

另一个命令：

```
dmSQL> SELECT id FROM tb_staff WHERE id > 0;
```

我们可以使用叶扫描而不需从数据页读取数据，因为在叶页就可以得到所要的数据。

- 当读取所有的数据时，索引扫描的成本为：

  cost = B tree level I/O + no. of leaf page I/O + cluster count

- 当读取所有的数据但只需要叶扫描时，索引扫描的成本：

  cost = B tree level I/O + no. of leaf page I/O

- 当读取一条记录时，索引扫描的成本：

  cost = B tree level I/O + one leaf page I/O + one data page I/O

- 当读取一条记录，但只需要叶扫描时，索引扫描的成本：

  cost = B tree level I/O + one leaf page I/O

- 当读部分数据时，索引扫描的成本：

  cost = B tree level I/O + (no. of leaf page × S) + (cluster count × S)

  其中S是选择率。

排序成本

将数据从磁盘读到内存要花费较多的时间，计算成本是和$\mathbf{c \times w \times n \times log_2(n)}$成比例的。其中，c是排序的字段数，w是排序键的字节数，n是参与排序的行数。

嵌套合并成本

嵌套合并需要使用二个以上的循环来访问数据页。对于嵌套合并而言，外层表和内层表是不同的。一般来说，嵌套合并的成本是：

外层表I/O + 内层表I/O * 外层表的行数

排序合并成本

排序合并执行之前必须先做排序，假设两个表都已在合并键上做了排序，那么排序合并的成本是两个表I/O的和。如果排序不在合并键上，则仍需要加上排序的成本。

19.4 统计值

统计值是指一个表的数据量及数据分布状况，它的主要目的是供成本函数计算成本，从而提供最佳的访问计划。随着表中数据的插入、删除和更新，这些统计值将会失去时效。此时，您可以通过update statistics命令来更新统计值以获得实时的统计值，从而提高查询效率。

统计值类型

DBMaster将收集下列的统计值：

针对每张表

- **nPg –** 数据页数
- **nRow –** 数据行数
- **avLen –** 一行的平均字节数

针对每个字段

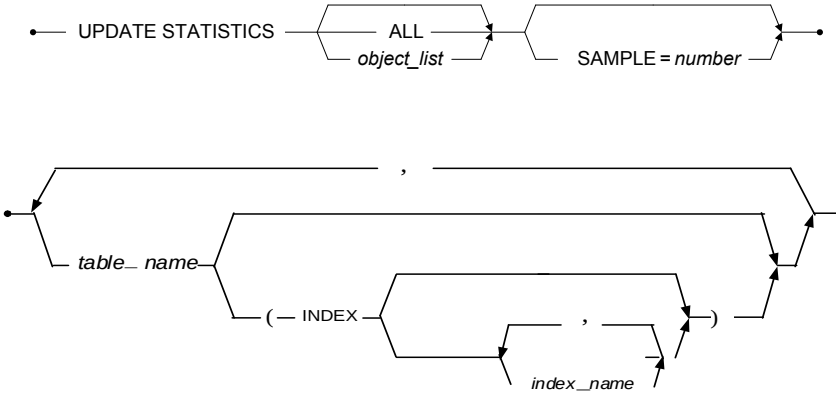
- **distVal –** 不同值的数目
- **avLen –** 每一个字段的平均字节数
- **loVal –** 一个字段中的次小值
- **hiVal –** 一个字段中的次大值

针对每个索引

- **nPg –** 索引页数目
- **nLevel –** 索引树层数目
- **nLeaf –** 索引树叶子数目
- **distKey –**不同键值数目

- **distC1 –** 第一个索引字段所含的不同键值数目
- **distC2 –** 前两个索引字段所含的不同键值数目
- **distC3 –** 前三个索引字段所含的不同键值数目
- **nPgKey –** 每一个键值所在的数据页数目
- **cCount –** 聚集数，通过索引访问到的数据页数目

更新统计值的语法



object_list子句

图19-1 UPDATE STATISTICS 语法

- **ALL**--强制更新所有模式对象的统计值。
- **SAMPLE**--抽样比率，它的值是介于1到100之间的整数，默认值为100。

➲ **示例1**

更新所有模式对象的统计值：

```
dmSQL> UPDATE STATISTICS;
```

如果数据库非常大，那么更新所有对象的统计值将会花费很长时间。用户可根据最近一次更新统计所做的修改，来更新特定模式对象的统计值，并设置采样率。

➲ **示例2**

强制更新所有模式对象的统计值：

```
dmSQL> UPDATE STATISTICS ALL;
```

➲ **示例3**

更新所有表的统计值包括：字段、索引、系统表，采样率为30：

```
dmSQL> UPDATE STATISTICS SAMPLE=30;
```

➲ **示例4**

更新**jeff.tb_staff**表的统计值：

```
dmSQL> UPDATE STATISTICS jeff.tb_staff;
```

➲ **示例5**

更新**jeff.tb_staff**和**jeff.tb_dept**表的统计值：

```
dmSQL> UPDATE STATISTICS jeff.tb_staff, jeff.tb_dept;
```

自动更新统计值后台程序

DBMaster提供对整个数据库自动更新统计值的后台程序。并非所有表的统计值都需更新，但采样率则是根据最近表的更新统计情况和上一次更新统计之后多少表信息被改变来决定的。用户还可以自行选择统计更新的模式，对不同表设置不同的采样率，更改统计更新的初始时间和时间间隔。用户可通过查询系统表SYSUSER获取更新统计状态，该状态以字符串格式存储在字段SQL_CMD中。此外，用户可通过调用过程

setSystemOption()中断正在运行的更新统计命令，同时该连接不会被中断。

注意 *如果更新统计值服务器不存在或已被终止，则无法启动更新统计值后台程序。此外，用户也无法在临时表上设置表统计值。*

用户可通过以下方法增强更新统计值的性能：**dmconfig.ini**设置、表设置、**setSystemOption**、获取更新统计状态和终止更新统计命令。

DMCONFIG.INI设置

DBMaster提供一些用来激活更新统计值后台程序的关键字。在配置文件**dmconfig.ini**中，关键字**DB_StSvr**设置为**1**可激活自动更新统计值后台程序；关键字**DB_StMod**用于指定数据库的更新统计值模式（**普通**模式和**表设置**模式）；关键字**DB_StsTm**用于指定更新统计值的开始时间；关键字**DB_StsTv**用于指定更新统计值后台程序的间隔时间；关键字**DB_StsSp**用于设置更新统计值的采样率。

➲ **示例1**

启动数据库前，在**dmconfig.ini**文件中为更新统计值后台程序配置相关关键字：

```
[DBNAME]
; Here omit other keywords
DB_StSvr = 0
DB_StMod = 1
DB_StsTm = 2010-10-10 10:00:00
DB_StsTv = 12:00:00
DB_StsSp = 70
```

现在启动数据库，更新统计值后台程序将会根据计划自动更新统计值。

➲ **示例2**

用户可在数据库运行期间为更新统计值后台程序设置所有参数：

```
dmSQL> CALL SETSYSTEMOPTION('STSVR','1');                    //激活自动更新
统计服务
dmSQL> CALL SETSYSTEMOPTION('STMOD','1');                    //重置
DB_StMod
dmSQL> CALL SETSYSTEMOPTION('STSTM','2009/6/6 20:30:00');    //重置DB
StsTm
dmSQL> CALL SETSYSTEMOPTION('STSTV','7- 00:00:00');          //重置DB
StsTv
dmSQL> CALL SETSYSTEMOPTION('STSSP','60');                   //重置DB
StsSp
```

表设置

如果用户通过执行SQL语句UPDATE STATISTICS SET为每个表设置了更新统计选项，就会出现以下4个过滤条件：

- 如果是一个新表，即此表从未执行过更新统计，那么该表会自动执行更新统计。
- 如果表的更新总页数不足20页，那么自动执行更新统计。
- 如果表的更新总页数超过20页，且自最后一次自动更新统计以来，被修改的页数超过2页，将执行自动更新统计。
- 如果此表超过10天未执行过更新统计，则执行自动更新统计。

如果用户开启更新统计值后台程序的**表设置**模式，即**DB_StMod**设为1，用户就可以通过执行SQL语句UPDATE STATISTICS SET来分别设置每个表的更新统计值选项。如果UPDATE STATISTICS SET中MODE取值为1，则该表的更新统计采样率由UPDATE STATISTICS SET中SAMPLE的值决定，但受上述4个过滤条件的影响；如果UPDATE STATISTICS SET中MODE取值为0，则该表的更新统计采样率由**dmconfig.ini**中**DB_StsSp**的值决定，也受上述4个过滤条件的影响。如果UPDATE STATISTICS SET中MODE取值为2，则该表的更新统计采样率由UPDATE STATISTICS SET中SAMPLE的值决定，且不受上述4个过滤条件的影响。

表设置信息存储在系统表SYSTABLE中，字段UPD_STS_MODE用来存储表更新统计值模式，字段UPD_STS_SAMPLE用来存储表更新统计值样率。

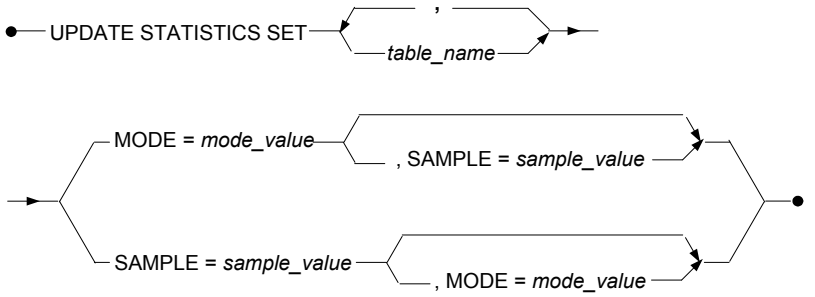


图19-2 UPDATE STATISTICS SET语法

➲ **示例1**

设置表 **jeff.tb_staff** 的更新统计模式和采样率：

```
dmSQL> UPDATE STATISTICS SET jeff.tb_staff MODE = 1, SAMPLE = 80;

dmSQL> SELECT TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE FROM
SYSTABLE;

         TABLE_NAME              TABLE_OWNER         UPD_STS_MODE
UPD_STS_SAMPLE

======================= ======================= =============
===============

TB_STAFF                 JEFF                                 1
80

1 rows selected
```

➲ **示例2**

设置表 **jeff.tb_staff 和 jim.tb_salary** 的更新统计模式和采样率：

```
dmSQL> UPDATE STATISTICS SET jeff.tb_staff, jim.tb_salary MODE = 1,
SAMPLE = 60;
```

```
dmSQL> SELECT TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE FROM
SYSTABLE;

         TABLE_NAME              TABLE_OWNER       UPD_STS_MODE
UPD_STS_SAMPLE

======================= ======================= ==============
===============

TB_STAFF                 JEFF                                  1
60

TB_SALARY                JIM                                   1
60

2 rows selected
```

SETSYSTEMOPTION

自动更新统计的系统选项有STSVR、STMOD、STSTM、STSTV和STSSP。数据库运行期间，用户可以调用系统存储过程**setSystemOption()**设置以上系统选项，也可以调用系统存储过程**setSystemOptionW()**设置以上系统选项并将其写入**dmconfig.ini**文件。此外，用户可通过调用系统存储过程**getSystemOption()**获取系统选项信息。系统存储过程**setSystemOption()**，**setSystemOptionW()**和**getSystemOption()**的详细使用方法请参考*SQL命令与函数参考手册*和*ODBC程序员参考手册*。

➲ **示例1**

在数据库运行期间设置**STSVR**为0：

```
dmSQL> CALL SETSYSTEMOPTION('STSVR', '0');
```

在数据库运行期间设置**STSVR**为1，并将**DB_StSvr**=1写入**dmconfig.ini**文件中的当前数据库部分：

```
dmSQL> CALL SETSYSTEMOPTIONW('STSVR', '1');
```

获取数据库运行期间系统选项**STSVR**的值：

```
dmSQL> CALL GETSYSTEMOPTION('STSVR', ?);
```

➲ **示例2**

在数据库运行期间设置更新统计采样率为70（**STSSP** = 70）：

```
dmSQL> CALL SETSYSTEMOPTION('STSSP', '70');
```

在数据库运行期间设置更新统计采样率为30（**STSSP** = 30），并将**DB_StsSp**=30其写入**dmconfig.ini**文件中的当前数据库部分：

```
dmSQL> CALL SETSYSTEMOPTIONW('STSSP', '30');
```

获取数据库运行期间系统选项STSSP的值：

```
dmSQL> CALL GETSYSTEMOPTION('STSSP', ?);
```

注意　*数据库运行时仅能更改部分配置关键字。*

更新统计值状态

用户可通过查询系统表SYSUSER获取更新统计值状态，该状态以字符串的格式存储在字段SQL_CMD中。

更新统计值状态信息如下所示：

```
[EXEC] update statistics command // (Total, start_time, execute_time,
remain_time, complete_percent) (table_name, start_time, execute_time,
remain_time, complete_percent) (index_name, start_time, execute_time,
remain_time, complete_percent)
```

更新统计状态包括三个部分：全部、表和索引。全部表示所有更新统计对象；表表示一个表的统计，其中包括该表的索引和数据页；索引表示索引统计或数据页统计。

➲ **示例**

```
dmSQL> SELECT CONNECTION_ID, SQL_CMD FROM SYSUSER;

 CONNECTION_ID                SQL_CMD

===============   ================================

3264              [EXEC]  Update t1 set c1 = c1 + 1;

3267              [EXEC]  update statistics // for table SYSADM.HUNDRED
index HUNDRED_CODE start at 2011/02/21 09:19:54
```

```
2 rows selected
```

监测和中断更新统计

DBMaster支持显示更新统计状态，用户可以通过查询表SYSUSER的字段SQL_AMD来监测更新统计进程。此外，用户还可以通过调用系统存储过程*setSystemOption('STS_ABORT', 'connection_id')*中断正在运行的更新统计。表SYSUSER字段SQL_AMD的信息及系统存储过程*setSystemOption('STS_ABORT', 'connection_id')*的使用，请参考*数据库管理工具用户管理手册*、*SQL命令与函数参考手册*和*ODBC程序员参考手册*。

➲ **示例1**

下例通过查询表SYSUSER的字段SQL_AMD来监测更新统计进程：

```
dmSQL> SELECT SQL_CMD FROM SYSUSER;
```

➲ **示例2**

下例是中断更新统计命令，被中断的连接ID为14076：

```
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT', '14076');
```

➲ **示例3**

0是特殊连接ID，下例表示中断与更新统计相关的所有连接：

```
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT', '0');
```

导入和导出统计值

用户可以使用unload statistics命令，将统计值从数据库导出到外部文本文件。 另外，用户也可以使用load statistics命令，从外部文本文件将统计值导入到数据库中。

➲ **示例1**

使用UNLOAD STATISTICS命令：

```
dmSQL> UNLOAD STATISTICS TO file1;//dump statistics from database to
file1
```

➲ **示例2**

使用LOAD STATISTICS命令：

```
dmSQL> LOAD STATISTICS FROM file1;//read statistics from external text
file
```

有经验的用户能通过修改统计数据的文件来提高查询效率，并输入到数据库。

➲ **示例3**

由UNLOAD STATISTICS产生的一个文本文件：

```
DBname = TESTDB


TBowner = jeff

TBname = tb_staff

TBpage = 5

TBrows = 30

Tbavlen = 50


COname = idxage

COtype = INTEGER

COdist = 12

COavlen = 4

COlow = 25

COhigh = 42


IXname = idxage
```

```
IXpages = 5
IXlevel = 2
IXleaf = 3
IXdist = 12
IXdistC1 = 12
IXdistC2 = 12
IXdistC3 = 12
IXpgkey = 8
IXcount = 7
```

19.5 加速查询执行

一般而言，可根据下列更改来加速查询的执行：

- 读取较少的记录数
- 避免排序或是在较少的数据行或字段上做排序
- 以顺序方式读取数据

数据模式

数据模式定义包括所有在数据库上的表、视图和索引。特别是索引的存在，它描述了索引是否可以用在合并、排序以及视图等情况。

查询计划

使用set dump plan on命令来检查DBMaster使用的查询计划。

下面列出一些执行计划的特点：

- **索引 —** 检查输出数据，看看索引是否被使用，如果有，如何使用。
- **过滤 —** 检查过滤条件，看看过滤掉多少数据。
- **查询 —** 检查完成的查询，看看访问计划是否为最佳的。

➲ **示例**

您可以通过以下命令，来检查查询计划：

```
dmSQL> SET DUMP PLAN ON;
```

索引检测

检测查询字段上是否存在适当的索引。用户可通过下一节介绍的方法，来改进查询的效率。

过滤字段

高效的查询仅获得一小部分源信息，用户通过select语句的where子条件控制输出信息的数量，我们称为数据过滤。

下面列出使用高级where子条件的方法：

避免关联的子查询

关联子查询是指某一字段同时出现在主查询和where谓词的子查询中。对包含在主查询的每个数据行进行重复子查询将返回不同结果。对于子查询而言，如果每一条记录的字段数据和前一条记录的不同，那么，它相当于对主查询所获得的每一条记录，都重新执行一个新的查询。

如果发现一个子查询的执行需要很长时间，首先检查是否是关联子查询。如果是，请改写查询以避免这种情况的发生，如果不容易改写，则寻找其它方法，来减少记录数。

避免模糊的LIKE查询

关键字LIKE提供了一个通配符比较，这就是规则表达式。当通配符在表达式开始的位置时，由于无法使用索引过滤，数据库服务将检查每一条记录，这会使得DBMaster顺序的访问和检查表中的每一条记录。

➲ **示例**

下例说明了如何将通配符"*"与LIKE关键字一起使用：

```
dmSQL> SELECT * FROM tb_salary WHERE name LIKE '*st';
```

查询结果

当理解查询实际在做什么之后，可以寻找其它查询方法来获得相同结果。下面的建议可以重写更有效的查询。

- 利用视图重写join
- 避免或减少排序
- 避免顺序的查询一个大表

- 使用Union来避免顺序访问

临时表

创建临时表是很有用的，它可加速表的查询，也可避免多字段上的排序操作，从而简化优化器的操作。

- 可使用临时表来避免多字段排序
- 可代替对无序访问的排序

19.6 基于语法的查询优化器

优化器根据功能成本和统计值自动选择查询执行计划。在一些特殊情况下，如数据分配不均匀，优化器可能选择一个较差的查询执行计划。为了解决这个问题，DBMaster支持一种基于语法的查询优化器的优化器机制。

你可以自定义查询的扫描类型和索引扫描中的索引使用。除此以外，DBMaster查询优化器可自动选择最有效的扫描方式， 即使你最近没有更新统计值。在使用索引类型时有五种不同情况。

强迫索引扫描方式

通常使用强迫索引扫描的语法如下：

```
tablename (INDEX [=] idxname [ASC|DESC])
```

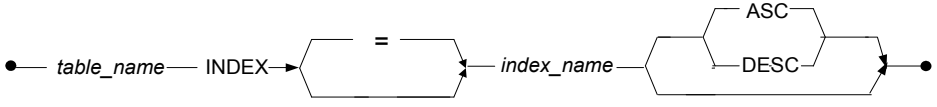


图19-2 强迫索引扫描语法

➲ **示例1**

要强迫表扫描，则定义值为0：

```
dmSQL> SELECT * FROM tb_staff (INDEX=0);
```

➲ **示例2**

在主键上强迫索引扫描，则定义值为1：

```
dmSQL> SELECT * FROM tb_staff (INDEX=1);
```

➲ **示例3**

在索引**idx_id**上强迫索引扫描：

```
dmSQL> SELECT * FROM tb_staff (INDEX idx_id);
```

➲ **示例4**

允许查询优化器决定在**tb_staff**表中使用何种扫描，但对**tb_salary**表强迫执行索引**idx_id**的索引扫描：

```
dmSQL> SELECT * FROM tb_staff, tb_salary (INDEX idx_id);
```

以别名方式强迫索引扫描

强迫索引扫描并为表提供别名的语法如下：

```
tablename (INDEX [=] idxname) aliasname
```

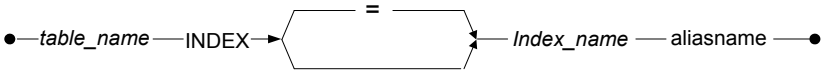


图19-3 以别名方式强迫索引扫描语法

➲ **示例**

在**idx_id**索引上强迫索引扫描，并为此表提供别名：

```
dmSQL> SELECT * FROM tb_staff (INDEX idx_id) a, tb_staff b WHERE a.id =
b.id;
```

以同义字方式强迫索引扫描

使用同义字方式强迫索引扫描的语法如下：

```
synonymname (INDEX [=] idxname)
```

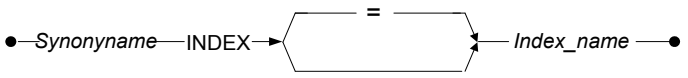


图19-4 以同义字方式强迫索引扫描语法

➲ **示例**

使用同义字staff的在idx_id索引上强迫索引扫描：

```
dmSQL> SELECT * FROM staff (INDEX idx_id);
```

以视图方式强迫索引扫描

创建视图时，使用强迫索引扫描的语法如下：

```
viewname (INDEX [=] idxname)
```

图19-5 以视图方式强迫索引扫描语法

➲ **示例1**

在创建视图**vi_staff**时在idx_id索引上强迫索引扫描：

```
dmSQL> CREATE VIEW vi_staff as SELECT * FROM tb_staff (INDEX idx_id);
```

不能在使用视图时强迫索引扫描。

➲ **示例2**

错误用法将会返回一个错误信息：

```
dmSQL> SELECT * FROM vi_staff (INDEX idx_id);
```

强迫全文索引扫描

强迫全文索引扫描的语法如下：

```
tablename (TEXT INDEX [=] idxname)
```

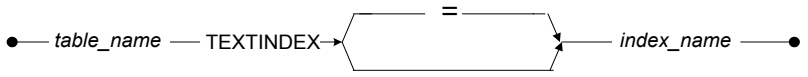


图19-6 强迫全文索引扫描语法

➲ **示例**

在**tidx1**索引上强迫全文索引扫描：

```
dmSQL> SELECT * FROM tb_staff (TEXT INDEX tidx1);
```

强迫循环连接(嵌套连接)

强迫两张表之间嵌套连接的语法：

```
tablename { INNER | OUTER } LOOP JOIN tablename
```

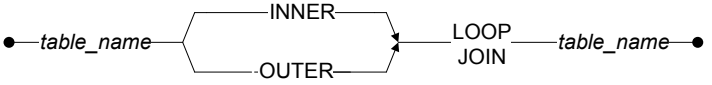


图19-7 强迫嵌套连接语法

注意 *此类型的强迫连接必须使用INNER JOIN或OUTER JOIN语法。*

➲ **示例1**

```
dmSQL> SELECT * FROM tb_staff INNER LOOP JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

➲ **示例2**

```
dmSQL> SELECT * FROM tb_staff OUTER LOOP JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

强迫合并连接

强迫两张表之间合并连接的语法：

```
tablename { INNER | OUTER } MERGE JOIN tablename
```

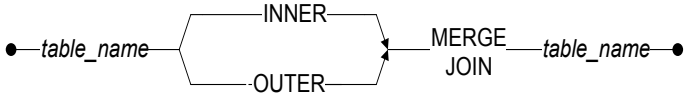


图19-8 强迫合并连接语法

注意 *当不能使用合并连接时，强迫合并连接也是无效的，但也不会返回错误信息。*

➲ **示例1**

```
dmSQL> SELECT * FROM tb_staff INNER MERGE JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

➲ **示例2**

```
dmSQL> SELECT * FROM tb_staff OUTER MERGE JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

强迫连接序列

强迫所有表的连接顺序，并且此连接顺序不能交换。以下是强迫连接顺序的语法：

```
SELECT ..... FROM [SEQUENCE | SEQ] tablename_list;
```

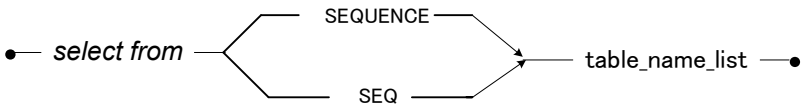


图19-9 强迫连接顺序语法

➲ **示例1**

```
dmSQL> SELECT * FROM sequence tb_staff, tb_salary, tb_dept WHERE
tb_staff.id= tb_salary.id AND tb_salary.basepay=tb_dept.basepay;
```

➲ **示例2**

```
dmSQL> SELECT * FROM seq tb_staff inner join tb_salary ON tb_staff.id=
tb_salary.id inner join tb_dept ON tb_staff.basypay=tb_detp.basepay;
```

强迫Group by方法

以下是强迫Group by方法的语法：

```
GROUP BY column_name_list [USING SORT | USING HASH] having .....
```

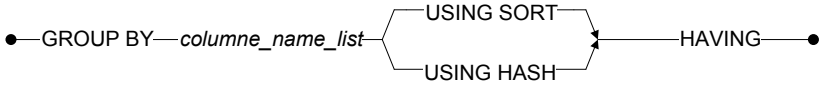


图19-10 强迫Group by方法的语法

➲ **示例1**

```
dmSQL> SELECT id,name,count(*) FROM tb_staff GROUP BY id,name USING
HASH;
```

➲ 示例2

```
dmSQL> SELECT id,name,count( * )  FROM tb_salary GROUP BY id,name USING
SORT HAVING sum(basepay)>0;
```

19.7 如何读导出计划

检测一个较慢的查询语句的第一步是阅读执行计划。DBMaster支持导出和阅读查询计划功能。

有三个用于导出计划的命令：

```
dmSQL> SET DUMP PLAN ON;
```

开启导出计划选项。之后的查询将伴随着显示导出计划来执行命令：

```
dmSQL> SET DUMP PLAN OFF;
```

关闭导出计划选项。之后的查询只执行命令但不显示导出计划。这是默认的选项：

```
dmSQL> SET DUMP PLAN ONLY;
```

只开启导出计划，但不执行命令。

导出计划大体看上去是由几个称为ON块组成。查询优化器将一个查询分成几个ON块，每个块都是一个逻辑的优化单元。优化器将优化每个ON块。简单查询通常只有一个ON块，但复杂查询如子查询可能会产生很多ON块，其中包括主块和子块。

优化器根据成本为每个块寻找最佳的执行方法。这就可以将一个ON块分成许多PL（批处理）块，每一个PL块代表一个操作，如扫描、合并等。

熟悉前一部分已介绍到的内容：

- 表扫描
- 索引扫描
- 嵌套合并
- 排序合并
- 因子

表扫描

➲ 示例

为 **tb_staff**表设置表扫描的导出计划：

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT * FROM tb_staff WHERE id>1;
```

tb_staff表扫描的导出计划结果：

```
----- begin dump plan -----

{ON Block 0}
ON Type     : SCAN

[PL Block 0]
Method     : Scan
Table Name : tb_staff
Type       : Table Scan
Order      : <none>
Factors    : (1) tb_staff.id > 1
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 330.0

----- end dump plan -----
```

前两行是一个ON块的信息：

{ON Block 0} — 一个块ID为0的ON块

ON Type: SCAN — ON块类型为扫描

ON块包括一个PL块：

[PL Block 0] — 一个PL块ID为0

Method: Scan — 这个PL块将执行一个扫描操作

Table Name: tb_staff — 定义扫描表tb_staff

Type: Table Scan — 扫描类型为表扫描

Order: <none> — 扫描顺序，在此对表扫描无用

Factors: (1) tb_staff.id > 1— 此扫描将使用过滤器tb_staff.id > 1

I/O Cost: 101.0 — 估计I/O成本为101.0页

CPU Cost: 25.3 — 估计CPU成本为25.3页

Sub Cost: 0.0 — 估计所有PL块的子块成本，此例中无PL的子块

Result Rows: 330.0 — 估计扫描和过滤后的所有结果行数

索引扫描

➲ **示例**

对表**tb_salary**的**id**和**name**字段使用where语句，设置导出计划：

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT id,name FROM tb_salary WHERE id>1 AND name=2;
```

导出计划的结果：

```
----- begin dump plan -----


{ON Block 0}
ON Type    : SCAN


[PL Block 0]
```

```
Method     : Scan
Table Name : tb_salary
Scan Type  : Index Scan on idx21(name, id)
Order      : ASC
Index EQFA#: 1
Index FA#  : 2
Index FACOL: 1, 2
Index Cost : 2
Factors    : (1) tb_salary.name = 2
           : (2) tb_salary.id > 1
I/O Cost   : 2.0
CPU Cost   : 0.6
Sub Cost   : 0.0
Result Rows: 13.0


----- end dump plan -----
```

前两行是一个ON块信息：

{ON Block 0} — 一个ON块，ID号为0。

ON Type: SCAN — 一个扫描类型的ON块，这个ON块也包含一个批处理块。

[PL Block 0] — 一个PL块ID为0。

Method: Scan — 块执行的是扫描操作。

Table Name : tb_salary — 扫描tb_salary表。

Scan Type: Index Scan on idx21(name, id) --扫描类型为索引扫描，应用建立在字段id和name上的索引idx21。

Order: ASC — 索引扫描顺序为升序。

Index EQFA#: 1 — 在索引扫描中应用的相等因子数，在此使用tb_salary.name = 2。

Index FA#: 2 — 在索引扫描中应用的因子数，在此使用 tb_salary.name = 2 and tb_salary.id>1。

Index FACOL: 1, 2 — 索引字段匹配的因子ID。在此例中，第一个索引字段name的匹配因子为(1) tb_salary.name=2，第二个索引字段id的匹配因子为(2)tb_salary.id>1。

Index Cost: 2 — 估计索引页成本为2。

Factors: (1) tb_salary.name = 2

(2) tb_salary.id > 1 -应用过滤器tb_salary.name=2和tb_salary.id>1。

I/O Cost: 2.0 — 估计I/O成本为2.0页。

CPU Cost: 0.6 — 估计CPU 成本为 0.6。

Sub Cost: 0.0 — 估计所有PL块的子块的成本。

Result Rows: 13.0 — 在扫描和过滤之后估计所有结果行数。

等值合并

➲ **示例**

对**tb_staff**和**tb_salary**表使用WHERE子句，设置导出计划：

```
dmSQL> SET DUMP PLAN ON;

dmSQL> SELECT * FROM tb_staff, tb_salary WHERE
tb_staff.name=tb_salary.name;
```

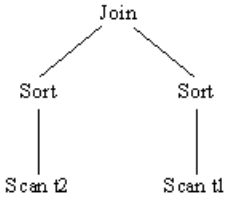
导出计划的结果：

```
----- begin dump plan -----


{ON Block 0}
```

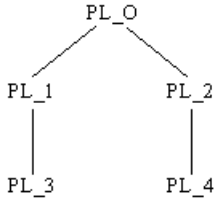
```
ON Type     : JOIN

[PL Block 0]
Method     : Join
Type       : Merge Join
Factors    : (1) tb_staff.name = tb_salary.name
I/O Cost   : 8.5
CPU Cost   : 573.8
Sub Cost   : 231.6
Result Rows: 500.0
Sub Block 1: [PL Block 1]
Sub Block 2: [PL Block 2]

[PL Block 1]
Method     : Sort
I/O Cost   : 4.2
CPU Cost   : 274.4
Sub Cost   : 120.0
Result Rows: 1000.0
SUB Block  : [PL Block 3]

[PL Block 3]
Method     : Scan
Table Name : tb_salary
Type       : Table Scan
Order      : <none>
```

```
Factors    : <none>
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 1000.0

[PL Block 2]
Method     : Sort
I/O Cost   : 4.2
CPU Cost   : 274.4
Sub Cost   : 120.0
Result Rows: 1000.0
SUB Block  : [PL Block 4]

[PL Block 4]
Method     : Scan
Table Name : tb_staff
Type       : Table Scan
Order      : <none>
Factors    : <none>
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 1000.0

----- end dump plan -----
```

此例中有多个PL块。PL块之间的联系通过子块信息建立。



一个简单的树型结构块



用名称代替每个节点

合并块描述：

[PL Block 0] — 一个PL块，ID为0

Method: Join — 块方法为合并

Type: Merge Join — 合并类型为排序合并

Factors: （1）tb_staff.name = tb_salary.name -- 应用合并过滤器 tb_staff.name = tb_salary.name使用合并块。

I/O Cost: 8.5 — 估计I/O 成本为8.5页

CPU Cost: 573.8 — 估计I/O成本为573.8页

Sub Cost: 231.6 — 估计所有PL子块的成本

Result Rows: 500.0 — 在合并块后估计所有结果的行数

Sub Block 1: [PL Block 1] — 此块的第一个子块连接到[PL Block 1]

Sub Block 2: [PL Block 2] — 此块的第二个子连接到[PL Block 2]

排序分块描述：

[PL Block 1] — 一个PL块ID为1

Method: Sort — 方法为排序

I/O Cost: 4.2 — 估计I/O成本为4.2

CPU Cost: 274.4 — 估计CPU 成本为 274.4

Sub Cost: 120.0 — 估计所有PL子块的成本为 120.0

Result Rows: 1000.0 — 在排序块之后估计的结果行数

SUB Block: [PL Block 3] — 此块的子块连接到[PL Block 3]

以上列出的是通常用户会遇到的导出计划信息。导出计划肯定有许多变化，但它们都包含同样的元素：I/O成本、CPU成本以及结果行。如果一个导出计划很复杂，可使用前面讨论的基于语法的优化器来尝试其它方法。

20 dmconfig.ini中的关键字

20.1 概念

当数据库引擎被启动或某个用户请求连接数据库服务器时，DBMaster就需要进行相应的参数设置。这些参数是从一个ASCII文本文件**dmconfig.ini**中读取的。这个文件中包含了系统配置中需要用到的关键字及其对应值。因为这个文件是ASCII格式，所以DBA可以利用文本编辑器来编辑它以设置所需参数。

大多数情况下，在数据库启动时需要用到关键字。在数据库启动后，改变关键字是无效的，重新启动数据库后参数才能生效。但是，有些参数是在用户连接到数据库时才会被请求到。因此用户可以在数据库启动后改变这些关键字，前提条件是如果想让这些设置生效，必须在连接建立之前设置这些关键字。配置参数对DBMaster的性能非常重要，为了使数据库运行顺畅，必须认真考虑每个参数的有效性并预估每个参数的最好预设值。建议数据库管理员备份**dmconfig.ini**文件，象备份其它文件一样。

20.2 dmconfig.ini文件格式

dmconfig.ini文件是ASCII文本文件，它可以用任何文本编辑器进行编辑。**dmconfig.ini**包括很多节，每节都由一些配置信息组成，用以启动特定数据库。每一节都有开头的名称，后面紧跟一组关键字和对应的值。

➲ 示例

dmconfig.ini文件格式：

```
[section_name_1]
keyword1 = value1          ; here is a comment
keyword2 = value2
        .
        .
[section_name_2]
keyword3 = value3 value4   ; spaces or commas may be used
keyword4 = value5          ; as delimiters
        .
        .
```

节名称

每个节的名称都和数据库名对应，在启动数据库时通过配置选项来建立。节名称以左中括号([)开始，后面跟随数据库名称，并以右中括号(])结束。中括号用来包含此节名称，并且必须以左括号开始。

关键字

紧跟在节后面的是一组关键字和对应值信息。在数据库启动时会用到这些关键字的值。语句keyword=value代表为此关键字定义值。关键字的值有可能是个整数或字串，这主要由关键字本身含义决定。

注释

在分隔符(;)之后的字串或符号都被作为注释，它不会影响关键字定义。

➲ **示例**

dmconfig.ini文件通过(;)分隔注释：

```
[SDB]
DB_DbFil=SDB.DB
DB_JnFil=SDE.JNL
DB_SMode=1              ;normal start mode
DB_BMode=1
DB_BkSvr=1
DB_BkTim=96/03/19 00:00:00
DB_BkItv=7-00:00:00
DB_NBufs=100
DB_NJnlB=200
DB_MaxCo=100
DB_JnlSz=20000
file1=SDE.FIL 40

[EMP]
DB_DbFil=EMP.DB
DB_JnFil=EMP.JNL
```

```
DB_SMode=1              ;normal start mode
DB_NBufs=100
DB_NJnlB=400
DB_MaxCo=100
DB_JnlSz=20000
file1=EMP.FL1 100
file2=EMP.FL2 200
```

在此例中，**dmconfig.ini**文件包含两个节，一个是**SDB**数据库，另一个是**EMP**数据库。

20.3 搜索dmconfig.ini文件路径

另一个问题是**dmconfig.ini**文件所在位置。对于UNIX系统，DBMaster将会在三个位置搜索**dmconfig.ini**文件。

➲ **位置及搜索顺序为：**

1. 当前路径。
2. 在环境变量DBMASTER中定义的路径。
3. 安装路径：*~DBMaster/Version*。

在微软Windows环境下，**dmconfig.ini**文件存放在安装目录下，通常是***C:\DBMaster\Version***。

当启动数据库时，DBMaster将查找上面所提到的路径以寻找**dmconfig.ini**文件中对应数据库的节信息。如果在**dmconfig.ini**中找到对应节，那么此节中定义的关键字将会生效。如果没有发现对应的节信息，DBMaster将顺序的在**dmconfig.ini**文件中搜索，直到找到对应的节名称为止。

20.4 关键字的默认值

当DBMaster需要参数时，就会在**dmconfig.ini**文件的对应节搜索对应的关键字。除了用户自定义文件，在搜索过程中对于每个节名称或关键字的模式匹配都是大小写不敏感的。如果某个关键字在**dmconfig.ini**文件中找不到，DBMaster就会使用此关键字的默认值。大部分关键字都有默认值。了解更多相关信息请参本章的关键字部分。

20.5 建立dmconfig.ini

通常情况下，数据库管理员需要在数据库创建及参数生效之前在**dmconfig.ini**文件中创建对应的节。然而，DBMaster如果在创建数据库时在**dmconfig.ini**文件中找不到对应的节，它将在**dmconfig.ini**文件中自动创建 以数据库名命名的节，如果在启动数据库时没有找到对应部分的节，DBMaster将会返回错误信息。

20.6 可供参考的关键字

DB_AtCmt=<值>

此关键字用来设置事务自动提交模式的开启或关闭状态。关键字设为1则开启自动提交模式，设为0则关闭自动提交模式。自动提交模式开启时，DBMaster会在每个SQL命令执行成功后自动提交事务。此关键字设定于客户端。

默认值：1

取值范围：0、1

请参考：DB_LTimO

用于何处：客户端

DB_AtrMd=<值>

此关键字定义数据库是否作为异步表复制的源数据库端。设置值为1则开启后台发送程序和基本表操作日志。因此，设置此参数后数据库可作为一个复制源数据库。

默认值：0

取值范围：0、1

请参考：DB_EtrPt、RP_LgDir

用于何处：服务器端

DB_BbFil=<字符串>

此关键字定义的是系统BLOB文件名称。它将会尽可能的扩展插入此文件的BLOB数据。

默认值：带有扩展名**.SBB**的数据库名。例如：**db.SBB**。

取值范围：字符串长度范围< 256

请参考：**DB_DbDir**、**DB_DbFil**、**DB_UsrBb**、**DB_UsrDb**

用于何处：**服务器端**

DB_BfrSz=<值>

此关键字定义了每个BLOB帧的大小（单位KB）。在数据库创建时会用到此关键字，因此在数据库创建前设置它才有效。

默认值：32 (KB)

取值范围：8~256 (KB)

请参考：**DB_BbFil**

用于何处：**服务器端 (仅为创建数据库所用)**

DB_BkChk=<值>

此关键字定义了在执行完整备份和差异备份之前是否检查数据库。如果此关键字设置为0，则表示不检查数据库；设置为1则表示检查数据库，但若发现数据库已被损坏，备份服务器将记录错误信息并停止此次备份；设置为2则表示检查数据库，但若发现数据库已被损坏，备份服务器将记录错误信息并继续将已损坏的数据库备份到BKDIR/BADDB目录下，仅当备份服务器第一次发现数据库损坏时才执行此动作，之后，若数据库检查正常，BKDIR/BADDB目录下的备份将被移除并继续执行备份。否则，仅记录错误信息并停止此次备份。

增量备份文件始终遵循上次完整备份或差异备份，所以若发现数据库已被损坏，增量备份文件将被置于BKDIR/BADDB目录下，当数据库恢复完成后，增量备份文件将被放回至BKDIR目录下。

默认值：0

取值范围：0、1、2

请参考：DB_DbKtv, **DB_DbKmax**

用于何处：服务器端

DB_BkCmp=<值>

此关键字设定是否使用压缩备份模式。并不是日志文件中的每个日志块都需要进行备份。如果设置此关键字值为1，备份服务器将只备份需要的日志块以节省磁盘空间。用户可参考*数据库的备份,恢复和还原*章节的相关内容。

默认值：1

取值范围：0、1

请参考：DB_BkSvr

用于何处：服务器端

DB_BkDir=<字符串>

此关键字设定备份服务器储存最近备份序列的一个或一组目录（最多32个）。有效的备份序列由三部分组成：一个完整备份、一个或多个差异备份（可选）、一系列增量备份（可选）。这个目录必须预先建立且需和**DB_DbDir**所指目录不同。

默认值：默认备份目录为数据库目录下的**backup**

取值范围：字符串长度< 256

请参考：DB_BkSvr、**DB_BMode**

用于何处：服务器端

➲ 示例

< BKDIR n >：第n个备份路径。

< SIZE n >：第n个备份路径的大小。

DB_BkDir = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…

```
[MYDB]

……

DB_BkDir = /home/usr/dbmaster/bk 5000 /home2/backup 1000

……
```

当***/home/usr/dbmaster/bk***填满时，文件将备份至***/home2/backup***目录下。

DB_BkFoM=<值>

此关键字定义了文件对象（FO）备份模式。只有在数据库完整备份时才可备份文件对象。**DB_BkFoM**有三个可能值：0、1、2 **DB_BkFoM**等于0时不使用FO备份模式，在完整备份时不备份FO文件。**DB_BkFoM**等于1时在完整备份过程中将备份系统FO。当**DB_BkFoM**等于2时系统FO和用户FO都会备份。

默认值：0

取值范围：　0：不备份文件对象

1：备份系统FO

2：备份系统FO和用户FO

请参考：DB_BkSvr、DB_FBkTm、DB_FBkTv、DB_BkDir

用于何处：服务器端

DB_BkFrm=<值>

此关键字用于增量备份时定义日志文件的命名格式。备份文件名格式为***<I><timestamp><_><DB_BkFrm>***，时间戳是一个系统的10位数字数据的有效时间，而**<DB_BkFrm>**可能主要包括文字常量和格式序列（例如，转义序列），它代表特殊的字符串。

格式序列可使用备份执行的年、月、日、数据库名称或备份识别号来定义。你可以通过很多方式将格式序列与文本常量相结合，最终提供的有效文件名要能被操作系统支持。格式序列由三部分组成：转义字符、长度值、格式符。格式有效顺序为：

%[*n*]Y—备份执行的年

%[*n*]M—备份执行的月

%[*n*]D—备份执行的日

%[*n*]B—备份识别号

%[*n*]N—日志文件所属的数据库名

➲ **示例**

DB_BkFrm = %N.%B

如果数据库为test1，那么增量备份文件名为：test1.1、test1.2……

可参考第15章*数据库备份、恢复和还原*。

默认值：*I<timestamp>*_%4N%4B.jnl

请参考：DB_BkSvr、DB_BkTim、DB_BkItv

用于何处：服务器端

DB_BkFul=<值>

此关键字定义日志文件的填充百分比。在达到这个触发值后，备份服务就被触发进行增量备份。当值为0时，代表无论何时，只要日志文件写满后都会触发备份服务器 做增量备份。值在50--100之间时，代表无论何

时，这个日志文件块写满至设置的百分比时都会触发备份服务器进行增量备份。例如，如果有两个日志文件，每个文件块均为500，并且**DB_BkFul**的值为80，那么在每个$500\times2\times0.8=800$块被使用后，备份服务器将会自动做增量备份。 可参考第15章*数据库的备份、恢复和还原*。

默认值：90

取值范围：0、50 ~ 100

请参考：**DB_BkSvr**、**DB_BkTim**、**DB_BkItv**

用于何处：**服务器端**

DB_BkItv=<字符串>

此关键字定义备份的时间间隔。请参看**DB_BkTim**部分的描述。

默认值：无 (如果**DB_BkItv**没有设置则无备份计划)

取值范围：0-00:00:01 ~ 24854-23:59:59

请参考：**DB_BkSvr**、**DB_BkTim**、**DB_BMode**

用于何处：**服务器端**

DB_BkOdr=<字符串>

此关键字定义备份服务设置前一版本备份序列的一个或一组路径（最多32个）。有效的备份序列由三部分组成：一个完整备份、一个或多个差异备份（可选）、一系列增量备份（可选）。请参考第15章*数据库的备份，恢复和还原*。

默认值：无

取值范围：字符串长度< 256

请参考：**DB_BkSvr**、**DB_BMode**、**DB_FBkTm**、**DB_FBkTv**

用于何处：**服务器端**

➲ 示例

DB_BkOdr = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>……

< BKDIR n > ：第n个备份路径。

< SIZE n >：第n个备份路径的大小（单位为8k）。

```
[MYDB]

……

DB_BkOdr = /home/usr/dbmaster/bk 5000 /home2/backup 1000

……
```

DB_BkRTs=<值>

此关键字定义了备份服务器在执行完整备份时，是否备份只读表空间文件。默认值为1，表示备份只读表空间文件。如果您已经备份了只读表空间文件，并且不希望备份服务器再对它进行备份，您可以将此关键字设置为0。当**DB_BkRTs=**0时，请注意：在将表空间设置成只读表空间时，您必须执行一个完整备份。如果您没有备份最后的文件，那么在您恢复一个包含只读表空间并且完整备份的数据库时，可能会出错。所以在没有特殊需要时，请尽量使用此关键字的默认值1。

默认值：1

取值范围：0、1

请参考：DB_BkSvr

用于何处：服务器端

DB_BkSPm=<值>

此关键字定义完整备份过程中是否备份ESQL和JAVA存储过程。该值可设置为0和1，默认值为0。**DB_BkSPm**的值为0时，表示不备份ESQL和JAVA存储过程；值为1时，表示备份ESQL和JAVA存储过程。此外，因

为源代码都将写入数据库，SQL存储过程在完整备份期间都将作为普通数据进行存储。

默认值：0

取值范围：0、1

用于何处：服务器端

DB_BkSvr=<值>

此关键字定义启动数据库后备份服务的状态，值为0时不激活备份服务，值为1时激活备份服务。要激活备份服务，可**在dmconfig.ini**文件中把**DB_BkSvr**的值设置为1，也可在数据库启动后使用*call setsystemoptio('bksvr','1')*命令更改BkSvr。详细信息请参考15章*数据库的备份，恢复和还原*。

默认值：0

取值范围：0、1

请参考：DB_BkCmp、DB_BkDir、DB_BkFul、DB_BkTim、DB_BkItv

用于何处：服务器端

DB_BkTim=<字符串>

此关键字与**DB_BkItv**一起定义备份服务计划。**DB_BkTim**定义备份服务第一次执行增量备份的时间。

➲ 示例

```
DB_BkTim = 96/05/01 00:00:00    ;backup begins from May 1, 1996.
DB_BkItv = 1-12:30:00           ;interval is every one day, 12 hours
                                 and 30 minutes.
```

关键字**DB_BkTim**和**DB_BkItv**只有在备份服务开启时才有意义。请参考*数据库的备份，恢复和还原*章节。

默认值：无 (不设置 **DB_BkTim**则无备份计划)

取值范围：1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

请参考：**DB_BkItv**、**DB_BkSvr**、**DB_BMode**

用于何处：**服务器端**

DB_BkZip=<值>

此关键字定义了在执行完整备份时，备份服务器是否压缩备份文件。

将此关键字设置为1表示在执行完整备份时压缩备份文件，设置为0表示在执行完整备份时不压缩备份文件。

默认值：0

取值范围：0、1

请参考：**DB_BkSvr**、**DB_BkCmp**、**DB_BkDir**、**DB_BkFul**、**DB_BkTim**、**DB_BkItv**

用于何处：**服务器端**

DB_BMode=<值>

此关键字定义数据库备份模式。设置值为0，NON-BACKUP模式；值为1，BACKUP-DATA模式；值为2，BACKUP-DATA-AND-BLOB模式。此关键字仅作用于增量备份，当值设置为1或2，用户可执行增量备份，当值设置为0，那么日志将不保存任何任务。请参考*数据库的备份，恢复和还原*部分。

默认值：0

取值范围：0、1、2

请参考：**DB_BkSvr**

用于何处：**服务器端**

DB_Brows=<值>

此关键字定义查询语句的锁行为。值为0，代表DBMaster在结果集中设置**S**锁；值为1，代表结果集上不设置锁。在连结数据库时会用到此关键字。

默认值：1

取值范围：0、1

用于何处：客户端

DB_CBMod=<值>

此关键字定义在一个事务结束后游标的行为。值为1表示在事务被提交后将关闭所有游标。值为2或3时表示在事务被提交后所有打开的游标继续保持开启状态。2表示事务结束后，所有的锁被释放。 3表示所有锁都保留但所有互斥锁将变成共享锁。在这三个情况中，如果事务终止，游标都会被关闭。

默认值：2

取值范围：1、2、3

用于何处：客户端

DB_ChkFl=<值>

此关键字定义数据库热启动时是否检查用户文件。设置为1时，该功能被禁止，服务器不检查用户文件；设置为0时则启用该功能。数据库启动时，如果配置信息或文件丢失，则会返回一条警告信息通知用户注意，该警告信息记录在日志文件*DMEVENT.LOG*中。

默认值：1

取值范围：0、1

用于何处：服务器端

DB_CliLCODE=<字符串>

此关键字用来定义客户端数据库字符集编码。当使用多语言数据库并且数据库服务器端的LCODE设置为10 （即数据库为UTF-8数据库）时，客户端可以使用DBMaster支持的任意字符集编码来连接UTF-8数据库服务器。

如果服务器端LCODE的值不是10（UTF-8），客户端的CLILCODE值必须与服务器端LCODE编码相同。

用户可以使用SELECT GETSYSINFO('CLILCODE')命令来返回客户端的语言代码设置。

默认值：如果服务器端LCODE值不是10（UTF-8），默认值则与服务器端LCODE编码相同。如果服务器端LCODE值是10（UTF-8），对于windows平台，客户端会将windows语言作为默认的CLILCODE值。

对于Linux平台，客户端会将环境变量LANG的值作为默认的CLILCODE值。如果没有设置LANG，默认值将是ASCII。

总之，如果客户端对该关键字未做任何设定，系统的默认行为一定会保证客户端能够与服务器端成功连接。

取值范围：DBMaster支持的所有字符集编码名称：

ASCII

ISO-8859-1

ISO-8859-2

ISO-8859-5

ISO-8859-7

BIG5

SHIFT-JIS

EUC-JP

GBK

GB18030

UTF-8

ISO-8859-{3,4,9,10,13,14,15,16}、KOI8-R、KOI8-U, KOI8-RU、CP{1250,1251,1252,1253,1254,1257}、CP{850,866}、Mac{Roman, Central Europe, Iceland, Croatian, Romania }、 Mac{Cyrillic, Ukraine, Greek, Turkish }、Macintosh(European Language)

ISO-8859-{6,8}、CP{1255,1256}、CP862、Mac{Hebrew, Arabic} (Semitic languages)

CP932、ISO-2022-JP、ISO-2022-JP-2、ISO-2022-JP-1(Japanese)

EUC-CN、CP936、EUC-TW、CP950(Chinese)

EUC-KR、CP949、JOHAB(Korean)

Georgian-Academy、Georgian-PS(Georgian)

KOI8-T(Tajik)

PT154(Kazakh)

TIS-620、CP874、MacThai(Thai)

MuleLao-1、CP1133(Laotian)

VISCII、TCVN、CP1258(Vietnamese)

请参考：**DB_LCode**

用于何处：客户端

- 示例**1**

```
DB_CliLCODE=GBK;
```

- 示例**2**

```
DB_CliLCODE= PT154;
```

DB_CmChe=<值>

此关键字定义是否开启客户端的命令缓冲区。值为1表示开启命令缓冲区，值为0表示关闭命令缓冲区。当客户端命令缓冲区开启时，在这之前执行的SQL命令相关信息都将保留在缓冲区中，如果之后的SQL命令相同则可以重复使用。该方法可提高数据库的性能。此关键字在客户端设置。

默认值：1

取值范围：0、1

用于何处：客户端

DB_CTbLM=<值>

此关键字定义了创建表时，将采用的默认锁模式。

设置**DB_CTbLM=0**，表示默认的锁模式为页加锁。在此情况下，如果创建表时没有指定锁模式，那么锁模式将采用页加锁。

设置**DB_CTbLM=1**，表示默认的锁模式为行加锁。在此情况下，如果创建表时没有指定锁模式，那么锁模式将采用行加锁。

默认值：1

有效值：0、1

用于何处：服务器端

DB_CTimO=<值>

此关键字定义当一个客户端连接数据库服务器时的连接超时值（单位为秒），如果一个数据库没有启动或者服务器地址错误，用户将被迫等待一段时间，直到连接超时。用户可设置此关键字的值来缩短连接时间。这个参数在客户端设置。

默认值：5（秒）

取值范围：5 ~ 1:00:00（1小时）

用于何处：客户端

DB_DaiFm=<值>

此关键字定义SQL语句的日期数据输入格式，了解更多信息请参考*ODBC编程指导手册*的附录B*函数特性差异*。

默认值：无（接受所有日期输入格式的数据）

取值范围：mm/dd/yy

mm-dd-yy

dd/mon/yy

dd-mon-yy

mm/dd/yyyy

mm-dd-yyyy

yyyy/mm/dd

yyyy-mm-dd

dd/mon/yyyy

dd-mon-yyyy

dd.mm.yyyy

yyyy.mm.dd

yyyymmdd

请参考：DB_DaoFm

用于何处：客户端或服务器端（客户端有较高优先权）

DB_DaoFm=<值>

此关键字定义SQL语句的日期输出格式。欲了解更多信息请参考*ODBC程序员参考手册*附录B*函数特性差异*。

默认值：yyyy-mm-dd

取值范围：取值范围同DB_DaiFm

请参考：DB_DaiFm

用于何处：客户端或服务器端（客户端有较高优先级）

DB_DbDir=<字符串>

此关键字定义数据库文件所存放的目录。目录字串可以是相对路径或完整路径名。

DBMaster中有7个类型的文件：

- 用**DB_DbFil**定义系统数据文件
- 用**DB_UsrDb**定义用户数据文件
- 用**DB_JnFil**定义系统日志文件
- 用**DB_BbFil**定义系统BLOB文件
- 用**DB_UsrBb**定义默认用户BLOB文件
- 用**DB_TpFil**定义系统临时文件
- 用户定义文件

当为这些关键字定义完整路径名时，可使用相对路径名或简单文件名来定义具体使用的文件。如果用路径名作为关键字值时，DBMaster将使用这个指定名来关联已经定义的文件。如果使用的是简单文件名，DBMaster将搜索**DB_DbDir**关键字。如果找到此关键字，DBMaster会将**DB_DbDir**中的路径名作为前缀与简单文件名结合以关联文件。如果没有找到，DBMaster就使用制定的简单文件名并假定文件在当前目录来关联它。

➲ **示例1**

```
[DB1]
DB_DbDir = /disk1/db
```

```
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

使用物理文件名：

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
DB_BbFil -- /disk1/db/DB1.BB (using default file name)
```

➲ **示例2**

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

使用物理文件名：

```
DB_DbFil -- mydb2 ( in current directory )
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.BB ( in current directory )
```

默认值：（当前目录）

取值范围：字符串长度< 256

请参考：DB_DbFil、DB_JnFil、DB_BbFil、DB_TpFil、DB_UsrDb、DB_UsrBb

用于何处：服务器端

DB_DbFil=<字符串>

此关键字定义操作系统使用的系统数据库文件的物理文件名。

默认值：带有**.SDB**扩展名的数据库名，如**db.SDB**。

取值范围：字符串长度< 256

请参考：DB_BbFil、DB_DbDir、DB_UsrBb、DB_UsrDb

用于何处：服务器端

DB_DbKmx=<值>

此关键字用于完整备份后定义差异备份的最大值。若差异备份值超出了**DB_DbKmx**设定的最大值时，备份服务器将删除最早的差异备份。此关键字取值为0表示不使用此关键字，换句话说，差异备份的数目没有限制。

默认值：10

取值范围：0 ~ 65535

请参考：DB_ FBkTm、DB_ FBkTv、DB_DbKtv

用于何处：服务器端

DB_DbKtv=<字符串>

此关键字用于定义差异备份的时间间隔。第一次差异备份的完成时间是**DB_FBkTm + DB_DbKtv**，格式为 **nDays-hh:mm:ss**。更多信息请参考关键字**DB_FBkTm** 和**DB_FBkTv**。

默认值：无（如果没有设置**DB_FBkTv**关键字，则无完整备份计划）

取值范围：0-00:00:01~24854-23:59:59

请参考：DB_ FBkTm、DB_ FBkTv、DB_DbKmx

用于何处：服务器端

DB_DsCmt=<值>

此关键字定义了应用程序断开数据库时，是否提交事务。**DB_DsCmt**=0表示当客户端应用程序发出一个SQL Disconnect命令时，DBMaster在断开数据库之前，将不发布commit命令，如果您在断开数据库之前没有设置自动提交或发布commit命令，事务将被回滚。

DB_DsCmt=1表示当客户端应用程序发出一个SQL Disconnect命令时，DBMaster在断开数据库之前，也将发布一个commit命令，如果您在断开数据库之前没有设置自动提交或发布commit命令，事务将被提交。

默认值：0

取值范围：0（断开数据库之前，不发布commit命令）

1（断开数据库之前，发布commit命令）

用于何处：客户端

DB_DtClt=<值>

此关键字定义了一个时间间隔，该时间间隔用来设定DBMaster客户端与服务器端之间通讯的最大空闲时间。该关键字的默认值为0，即关闭该功能。

有时，在客户端机器突然断电或网络运行异常的情况下，服务器端并不会释放分配给该客户端的资源。通过设置该关键字便可解决这个问题，该关键字使服务器定期地检查客户端的连接情况，当发现了一个无用连接，它便会释放该连接的所有资源。如果一个区域的网络运行不稳定，DBA可以通过设置这个区域的客户端空闲时间（**DB_DtClt**）来使服务器定期地检查连接，如果客户端连接已经是一个死连接，服务器则释放所有资源。

DBMaster客户端与服务器端之间通讯的最大空闲时间由**DB_DtClt**（客户端设置）设置的值和**DB_ITimo**（服务器端设置）设置的值决定，规则如下：

- 如果用户仅设置这两个关键字中的一个，那么DBMaster客户端与服务器端之间通讯的最大空闲时间为已设置关键字的值。即如果用户在客户端设置**DB_DtClt**，而未在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_DtClt**的设置值；如果用户未在客户端设置**DB_DtClt**，而在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_ITimo**的设置值。

- 如果用户同时设置这两个关键字，则存在以下两种情况：
  a) **DB_DtClt**的值小于**DB_ITimo**的值，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_DtClt**的设置值。
  b) **DB_DtClt**的值大于**DB_ITimo**的值，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_ITimo**的设置值，同时**DB_DtClt**的设置值将会被重置为**DB_ITimo**的设置值。
- 如果用户既没在客户端设置**DB_DtClt**，又没在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间将无限制。

默认值：0（秒） – 不允许

取值范围：0、5 ~ 1:00:00（1小时）

请参考：**DB_ITimO**

用于何处：**客户端**

DB_ERMRv=<字符串>

当数据库发生错误时，此关键字定义e-mail错误通知的收件人。最多可定义8个收件人e-mail地址。每个地址字串必须用逗号分隔。如果没有定义收件人，错误报告系统将不被启用，也就无e-mail产生。

默认值：空

请参考：**DB_ ERMSv**

用于何处：**服务器端**

DB_ERMSv=<字符串>

此关键字定义是否使用SMTP服务来传递e-mail信息。只能指定一个SMTP服务器。如果没有指定SMTP服务但却定义了e-mail的收件人，那么DBMaster将设置此关键字的值为'localhost'。

默认值：localhost

请参考：DB_ERMRv

用于何处：服务器端

DB_ErrLCODE=<字符串>

此关键字用于设置客户端的错误信息字符集。在多语言数据库中，客户端可以设置自己的错误信息输出编码。

该关键字的有效值可以采取语言定义、本地定义和字符集相结合的方式。即*language [_locale] [.code]*的形式，其中*language*字符串参照ISO-639标准，必须为小写字符，*locale*字符串参照ISO-3166标准，必须为大写字符，*code*字符串则是DBMaster支持的字符集名称。

对于有多个地域区别的语言，应该指出其地域性区别。例如，zh_CN或者zh_TW，仅仅是zh则是无效的。目前，DBMaster支持4种语言的错误信息输出编码。分别为：英文、简体中文、繁体中文和日文。

存储错误信息的文件放置在*dbmaster\5.4\shared\locale\locale_LANG/*路径下。

用户可以键入命令SELECT GETSYSINFO('ERRLCODE')来返回客户端的错误信息字符集。

默认值：在windows平台，客户端会将注册时得到的**LANG**值作为默认值，如果在注册时没有安装语言值，此值将是操作系统的语言编码。在Linux平台，客户端会将环境变量**LANG**值作为默认值，如果**没有**设置**LANG**值，DBMaster将设置客户端错误信息字符集为ASCII。DBMaster目前支持的有效的语言和本地定义有：en、ja、zh_CN、zh_TW。

取值范围：四种语言en、zh_CN、zh_TW和ja，或它们与相应字符集的组合，如：en、en.ASCII、en.ISO-8859-1、en.ISO-8859-2、en.ISO-8859-5、en.ISO-8859-7、en.UTF-8；

ja、ja.SHIFT-JIS、ja.SHIFT_JIS 、ja.UTF-8、ja.EUC-JP 、ja.EUCJP；

zh_CN、zh_CN.GBK、zh_CN.UTF-8、zh_CN.GB18030；

zh_TW、zh_TW.BIG5、zh_TW.UTF-8；

请参考：DB_LCode、DB_CliLCODE

用于何处：客户端

➲ **示例**

```
DB_ErrLCODE= zh_CN;
DB_ErrLCODE= zh_CN.GBK;
DB_ErrLCODE= zh_CN.UTF-8;
```

DB_EtrPt=<值>

此关键字的值为整型，它定义了数据库服务器为附加的订阅者后台服务使用的TCP/IP端口号。这对于快捷异步表复制是很有用的。在源数据库端，此关键字表示如何与目的数据库的订阅者服务建立连接。在目的数据库端，此关键字将启动订阅者后台服务程序。

默认值：无

取值范围：1024 ~ 65535

请参考：DB_AtrMd、RP_LgDir

用于何处：服务器端（在源数据库和目的数据库两端均设置）

DB_ExtHd=<值>

此关键字定义了反复扩展一个文件的阈值。值为1时，表示通常从第一个文件开始自动扩展表空间；值为0时，表示通常从最小文件开始自动扩展表空间；值为1~4294967296/**DB_PgSiz**, 1M~4096M或 1G~4G中的某个值时，则表示首先从最小的文件开始扩展，直至该文件的大小超过次小文件与**DB_ExtHd**值的总和，再开始扩展次小文件，这种方法在保持性能的同时还兼顾了文件大小的平衡。**DB_ExtHd**的默认值为100M，更多详细信息请参考5.3章节的*均匀自动扩展表空间*。

默认值：100M

取值范围： -1、0、1 ~ 4294967296/**DB_PgSiz**、1M ~ 4096M、1G ~ 4G

请参考：DB_ExtNp

用于何处：服务器端

DB_ExtNp=<值>

此关键字定义了自动扩展表空间的扩展尺寸。当自动扩展表空间用完时，DBMaster将会对其空间进行自动扩展。这个值定义了每次扩展页/帧的大小。

默认值：20 (页/帧)

取值范围：1 ~ 32767

请参考：DB_ExtHd

用于何处：服务器端

DB_FBkTm=<字符串>

此关键字与**DB_FBkTv**结合，用来定义备份服务执行在线完整备份的计划。**DB_FBkTm**定义备份服务第一次执行完整备份的时间。服务器将根据**DB_FBkTv**定义的时间间隔执行在线完整备份。

➲ **示例**

```
DB_FBkTm = 96/05/01 00:00:00    ;begins May 1, 1996.

DB_FBkTv = 1-12:30:00           ;interval is every one day, 12 hours and
30 minutes.
```

关键字**DB_FBkTm**和**DB_FBkTv**只用于备份服务器。

默认值：无

取值范围：1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

请参考：DB_FBkTv、**DB_BkSvr**、**DB_BkOdr**

用于何处：服务器端

DB_FBkTv=<字符串>

此关键字定义完整备份的时间间隔。了解更多信息请参考**DB_FBkTm** 。

默认值：无（如果没有设置**DB_FBkTv**关键字，则无完整备份计划）

取值范围：0-00:00:01 ~ 24854-23:59:59

请参考：**DB_BkSvr**、**DB_FBkTm**、**DB_BkOdr**

用于何处：**服务器端**

DB_FltDb=<字符串>

此关键字定义了FLOAT字段应该使用4（REAL）字节还是8（DOUBLE）字节来作为内部存储和值范围。类型名称也会随之更改成REAL或DOUBLE。

设置**DB_FltDb**=0表示FLOAT字段的存储为**4**字节，并且类型名称为REAL。

设置**DB_FltDb**=1表示FLOAT字段的存储为8字节，并且类型名称为DOUBLE。

默认值：1

取值范围：0、1

用于何处：**服务器端**

DB_FoDir=<字符串>

此关键字定义系统文件对象路径或系统文件对象的子目录（主要根据**DB_FoSub**的设置），**DB_FoDir**关键字必须定义为完整路径名。

➲ **示例1**

```
DB_FoDir = /usr/DBMaster/fileobj            ;for UNIX
```

➲ **示例2**

```
DB_FoDir = c:\DBMaster\fileobj                ;for DOS
```

➲ **示例3**

```
DB_FoDir = \\NTMachine\DBMaster\fileobj      ;for Microsoft UNC name
```

只有设置了**DB_FoDir**后才可向新的系统文件对象中插入数据。数据库启动时用到此关键字。

默认值：空子串（不能插入系统文件对象）

取值范围：字符串长度< 256

请参考：**DB_UsrFo**、**DB_FoSub**

用于何处：**服务器端**

DB_ForcS=<值>

此关键字定义DBMaster强制启动数据库，即使有错误发生仍强制启动数据库。值为1时，强制启动数据库。

默认值：0

取值范围：0、1

请参考：**DB_SMode**

用于何处：**服务器端**

DB_ForUX=<值>

此关键字定义服务器端“select … for update”语句的锁行为。

DBMaster在“select … for update”语句的结果集上设置**U**锁。值为1时表示“select … for update”语句的结果集上设置的是**X**锁。此关键字在启动数据库时用到。

默认值：0

取值范围：0、1

用于何处：服务器端

DB_FoSub=<值>

此关键字定义存储在每个系统文件对象子目录中的最大文件对象数。文件对象子目录将被创建在关键字**DB_FoDir**所定义的文件对象目录中。当子目录中的文件对象数超过了定义的阈值时，系统将会自动创建新的文件对象子目录。

➲ **示例**

设置每个文件对象子目录中的文件数为500：

```
DB_FoSub = 500
```

在设置**DB_FoDir**关键字后，可以向系统文件对象中插入数据。数据库启动时会用到此关键字。

默认值：0（文件对象会直接存储在**DB_FoDir**定义的目录中）

取值范围：100 ~ 1000000、0 （FO目录不包含子目录）

请参考：DB_FoDir

用于何处：服务器端

DB_FoTyp=<值>

此关键字定义FILE类型所对应的ODBC类型。ODBC不支持DBMaster所支持的FILE类型。一些开发工具，如Borland Delphi或Microsoft Visual Basic都无法识别FILE类型。如果想通过这些工具来访问FILE类型数据，DBMaster需要在内部把FILE类型映射成LONG VARBINARY型。这个映射需要将**DB_FoTyp**值设置为1，如果**DB_FoTyp**值为0，则没有映射关系。

默认值：1

取值范围：0（不使用映射）

1（FILE 类型映射到LONG VARBINARY类型）

用于何处：客户端

DB_GcChk=<值>

此关键字用来设置每秒最小事务数TPS。TPS用于初始化整组提交事务协议。

DBMaster会检查当前事务数。每秒事务数等于每秒同步请求数。DBMaster使用**DB_GcChk**作为TPS的极限数。当服务器TPS到达此极限时，整组提交协议将被激活。当TPS低于极限时，整组提交关闭。例如，**DB_GcChk** = 20，当每秒钟事务数超过20时，整组提交协议将会开启。

DB_GcChk关键字可使DBMaster的整组提交协议在激活和关闭状态间动态切换。因为事务的激活状态并不是永恒不变的，例如：有时值高，有时值低，DBMaster将会切换状态以避免不必要的等候。

默认值：20

取值范围：>0

请参考：DB_GcWtm、DB_GcMxw

用于何处：服务器端

DB_GcMxw=<值>

此关键字定义等候事务整组提交所允许的最大数。它的操作要与**DB_GcWtm**关键字结合，它主要会影响整组提交的最大等候时间。

如果整组提交协议是处于激活状态的（详细信息请参考**DB_GcChk**），DBMaster将会对每个同步请求进行检测。如遇到以下情况，同步进程将会运行，否则在执行整组提交时将等待另一个同步进程。

- 事务达到**DB_GcWtm**关键字定义的最大等待时间时。
- 等待整组提交的事务数超过了**DB_GcMxw**关键字定义的值。

➲ 示例

如果最大等待时间为30毫秒并且事务等候数为5。那么至少一个事务等候超过30毫秒或者有5个事务处于等待状态时，DBMaster执行同步操作。

设置**DB_GcMxw=**0关闭组提交功能。

默认值：0（关闭）

取值范围：1 ~ 32768

请参考：DB_GcChk、DB_GcWtm

用于何处：服务器端

DB_GcWtm=<值>

此关键字定义任一事务等候整组提交的最大等待时间。较长的等待时间可降低事务响应时间，但会增加整组提交的吞吐量。

此关键字与**DB_GcMxw**一起使用。了解详细信息请参考**DB_GcMxw**关键字。

默认值：30（毫秒）

取值范围：> 0

请参考：DB_GcChk、DB_GcMxw

用于何处：服务器端

DB_IDCap=<值>

此关键字定义了数据库中所有标识符的大小写敏感度。值为0，指出所有标识符都是大小写敏感的。值为1，指出所有标识符都是大小写不敏感的，所有标识符都被系统内部转换成大写字母。此关键字必须在数据库创建之前设置，这说明在数据库创建之后改变此关键字的值是无效的。

默认值：1

取值范围：0（大小写敏感）

1（大小写不敏感）

用于何处：服务器端

DB_IdxDp=<值>

此关键字定义了使用自动索引后台程序自动删除一个自动索引的阈值。如果一个自动索引未使用的天数达到或超过关键字**DB_IdxDp**指定的期限，则该自动索引就会被自动索引后台程序删除。在数据库运行期间，用户可以调用*setSystemOption()*来设置**IDXDP**的值。

默认值：30（天）

取值范围：0 ~ 365（天）

请参考：DB_IdxLg、**DB_IdxLn**、**DB_IdxSv**、**DB_IdxTm**、**DB_IdxTv**

用于何处：服务器端

DB_IdxLg=<字符串>

此关键字定义了自动索引日志文件的保存目录。默认路径是数据库的安装目录。

默认值：数据库目录

取值范围：字符串长度 < 256

请参考：DB_IdxDp、**DB_IdxLn**、**DB_IdxSv**、**DB_IdxTm**、**DB_IdxTv**

用于何处：服务器端

DB_IdxLn=<值>

此关键字定义了使用自动索引后台程序自动创建一个自动索引的阈值。如果扫描日志数（这些日志具有相同的表ID、表版本、字段ID和表达式字符串列表）达到或超过关键字**DB_IdxLn**指定的值，则自动索引后台程序会根据这些串行日志创建一个自动索引。在数据库运行期间，用户可以调用*setSystemOption()*来设置**IDXLN**的值。

默认值：1

取值范围：1 ~ 65535

请参考：DB_IdxDp、**DB_Idx Lg**、**DB_IdxSv**、**DB_IdxTm**、**DB_IdxTv**

用于何处：服务器端

DB_IdxSv=<值>

此关键字用于激活自动索引服务器。值为1时，启动自动索引服务器；值为0时，不启动自动索引服务器。在数据库运行期间，用户可以调用*setSystemOption()*来设置**IDXSV**的值。

默认值：0

取值范围：0、1

请参考：DB_IdxDp、**DB_IdxLg**、**DB_IdxLn**、**DB_IdxTm**、**DB_IdxTv**

用于何处： 服务器端

DB_IdxTm=<字符串>

此关键字定义了自动索引后台程序首次执行自动索引的时间。**DB_IdxTm**的格式为**yyyy-mm-dd hh:mm:ss**。在数据库运行期间，用户可以调用*setSystemOption()*来设置**IDXTM**的值。

默认值：1970-01-01 00:00:00

取值范围：1970-01-01 00:00:00 ~ 2037-12-31 23:59:59

请参考：DB_IdxDp、**DB_IdxLg**、**DB_IdxLn**、**DB_IdxSv**、**DB_IdxTv**

用于何处： 服务器端

DB_IdxTv=<字符串>

此关键字定义了自动索引后台程序的时间间隔。例如，值"1-12:30:00"表示时间间隔为1天零12小时30分钟。在数据库运行期间，用户可以调用*setSystemOption()*来设置**IDXTV**的值。

默认值：0-01:00:00

取值范围：0-00:00:00 ~ 24854-23:59:59

请参考：**DB_IdxDp**、**DB_IdxLg**、**DB_IdxLn**、**DB_IdxSv**、**DB_IdxTm**

用于何处：**服务器端**

DB_IFMem=<值>

此关键字定义DBMaster分配IVF全文索引搜索程序的最大内存数。创建一个反向全文索引需要占用大量的内存资源。DBMaster会使用一个简单的规则来为创建的反向全文索引分配最多的内存空间。如果DBMaster无法检测到空闲内存或空闲内存少于128 MB，那么最大的内存使用为64 MB。否则，可利用的内存将是空闲内存的一半。用户可以在配置文件**dmconfig.ini**中添加**DB_IFMem**关键字，来近似的指出将要使用的内存最大值（MB）。如果您的操作系统不提供内存使用信息，或者想为IVF全文索引分配一个更大的内存空间以提高查询性能，则推荐使用这种方法。

➲ **示例**

DBMaster分配IVF全文索引搜索程序的最大内存数为 256 MB：

```
DB_IFMem = 256
```

默认值：无-DBMaster自动配置内存缓冲值

取值范围：64 ~（操作系统允许的最大值）

用于何处：**服务器端**

DB_IOSvr=<值>

此关键字定义DBMaster是否开启I/O服务器以及checkpoint后台程序。值为1，表示在启动数据库后，开启I/O服务器以及checkpoint后台程序。在数据库启动时会用到此关键字。

默认值：1

取值范围：0、1

请参考：DB_NBufs

用于何处：服务器端

DB_IsoLv=<值>

此关键字定义了用户连接数据库时，分配的默认事务隔离级别。

有效值的范围和选项如下：

1：只读未提交

2：只读已提交

3：可重复读取

4：可串行化

默认值：1

取值范围：1~4

用于何处：客户端

DB_ItcMd=<值>

此关键字定义了是否开启隐式数据转换功能。值为1表示开启隐式数据转换；值为0表示关闭隐式数据转换。

默认值：0

取值范围：0、1

用于何处：服务器端

DB_ITimO=<值>

此关键字定义空闲时间间隔，单位精确到**秒**。如果在一段已经超过设定的超时间隔内，还没有任何数据库操作，DBMaster 将自动断开连接。这个动作强迫无用连接释放所有数据库资源，包括缓冲区、页、锁以及内存。该关键字的默认值为 0，即关闭该功能。

DBMaster客户端与服务器端之间通讯的最大空闲时间由**DB_DtClt**（客户端设置）设置的值和**DB_ITimo**（服务器端设置）设置的值决定，规则如下：

- 如果用户仅设置这两个关键字中的一个，那么DBMaster客户端与服务器端之间通讯的最大空闲时间为已设置关键字的值。即如果用户在客户端设置**DB_DtClt**，而未在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_DtClt**的设置值；如果用户未在客户端设置**DB_DtClt**，而在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_ITimo**的设置值。
- 如果用户同时设置这两个关键字，则存在以下两种情况：
  - **a)** **DB_DtClt**的值小于**DB_ITimo**的值，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_DtClt**的设置值。
  - **b)** **DB_DtClt**的值大于**DB_ITimo**的值，则DBMaster客户端与服务器端之间通讯的最大空闲时间即为关键字**DB_ITimo**的设置值，同时**DB_DtClt**的设置值将会被重置为**DB_ITimo**的设置值。
- 如果用户既没在客户端设置**DB_DtClt**，又没在服务器端设置**DB_ITimo**，则DBMaster客户端与服务器端之间通讯的最大空闲时间将无限制。

默认值：0（不允许）

取值范围：0 ~ 4294967（秒）

请参考：DB_DtClt

用于何处：服务器端

DB_IttDir=<字符串>

此关键字指定系统临时文件的存储目录，用户最多可指定8个存储目录。如果未设置该关键字，系统临时文件将保存在**DB_DbDir**指定的目录中。用户可以在数据库运行期间设置该关键字，并且立即生效。

当用户使用**DB_IttDir**指定的存储目录时，由于系统临时文件的最大限制为4GB，为了避免文件增长而产生的磁盘满问题，创建系统临时文件之前，系统将依次计算存储目录的可用空间大小，并将文件存入可用空间大于4GB的目录。如果所有存储目录的可用空间均不足4GB，那么该文件将保存在第一个存储目录中。

默认值：关键字**DB_DbDir**指定的目录

取值范围：最多定义8个字串，每个字串的长度不限，字串间以空格分隔

请参考：DB_DbFil、DB_BbFil、DB_TpFil

用于何处： 服务器端

DB_JnFil=<字符串>

此关键字定义了系统日志文件名称。它也为数据库定义分配的日志文件数。最多可定义8个日志文件。

默认值：带有文件扩展名**.JNL**的数据库名，例如：**DB.JNL**。

取值范围：字符串长度 < 256

请参考：DB_JnlSz、DB_DbDir

用于何处：服务器端

DB_JnlSz=<值>

此关键字定义日志文件大小，用户可以为文件大小指定M或G作为单位。如果没有指定，单位则为页。如果用户指定了M或G作为单位，那么实际的日志文件大小会比指定的值少一页。例如，如果数据页大小为16K，用户设置**DB_JnlSz**=8M，则日志文件的实际大小为8176K，而不是8192K。

默认值：1000 (默认日志大小为 1000*PgSiz KB)

取值范围：100pages ~ 8G

请参考：**DB_JnFil**、**DB_DbDir**

用于何处：**服务器端**

DB_LbDir=<字符串>

此关键字定义用户自定义函数的目录（UDF），如当数据库启动时，Windows环境下的dll文件或Unix环境下的so文件所在路径。

默认值：**DB_DbDir**

取值范围：字符串长度 < 256

请参考：**DB_JnlSz**、**DB_DbDir**

用于何处：**服务器端**

DB_LCDec=<值>

此关键字定义数据库是否检测操作系统使用的小数点设置字符。设置该关键字的值为1，则检测功能开启；设置该关键字的值为0，则检测功能关闭。该关键字默认值为0，将“.”作为DBMaster默认的小数点设置字符。该关键字值设置为1时，DBMaster检测操作系统使用的小数点设置字符，并用检测到的字符作为DBMaster中的小数点字符。换句话说，如果操作系统使用的小数点字符是逗号“,”，则DBMaster同样使用逗号“,”作为小数点字符；如果操作系统使用的小数点字符是“.”，则

DBMaster继续使用“.”作为小数点字符。操作系统使用的小数点字符是逗号“,”时，建议用户设置该关键字的值为1。

默认值：0

取值范围：0、1

请参考：DB_LCode

用于何处：客户端

DB_LCode=<值>

此关键字用于在服务器端定义语言代码。语言代码将影响一个查询中LIKE操作的结果。值为0表示这个语言代码与ASCII码相匹配。值为1表示语言代码与中文BIG5代码相匹配。值为2表示语言代码与SHIFT-JIS码相匹配。值为3表示语言代码与GB码匹配。要想了解更多信息请参考*SQL命令与函数参考手册*。请注意，在数据库启动时，需要对该值进行设置。

用户可以使用命令SELECT GETSYSINFO('LCODE')来返回服务器端的语言代码设置。

默认值：（安装时设置）

取值范围：0 英文（ASCII）

1 繁体中文（BIG5）

2 日文（Shift-JIS + Half Corner）

3 简体中文（GBK）

4 拉丁语（ISO-8859-1）

5 拉丁语（ISO-8859-2）

6 斯拉夫语（ISO-8859-5）

7 希腊语（ISO-8859-7）

8 日文（EUC-JP）

9 简体中文（GB18030）

10 Unicode（UTF-8）

请参考：DB_CliLCODE、DB_ErrLCODE

用于何处：服务器端

DB_LetPT=<值>

此关键字定义页锁升级为表锁的锁增加阈值。如果在同一个表中页的锁定数超出了锁的增加阈值，那么DBMaster将会自动将锁升级为表级锁。

默认值：60

取值范围：0，3 ~ 32767

请参考：DB_LetRP

用于何处：服务器端

DB_LetRP=<值>

此关键字定义一个从行锁到页锁的锁增加阈值。如果在同一个页中行锁的数量超过了锁增加阈值，DBMaster将自动将锁升级为页级锁。

默认值：30

取值范围：3 ~ 32767

请参考：DB_LetPT

用于何处：服务器端

DB_LgDay=<值>

该关键字用来指定日志保持的天数，过期的日志文件可通过关键字**DB_StSvr**被后台服务器删除。如果**DB_LgDay** = 0，日志系统的原始规则（文件名不包括日期）将会被采用，当**DB_LgDay**值大于零时，**DB_LgFNo**的设置将会被忽略。

默认值：30

有效范围：0，1 ~ 365

请参考：DB_LgZip、DB_LgSvr、DB_LgFNo

用于何处：服务器端

DB_LgDir=<字符串>

该关键字用来指定日志服务器的路径，也就是日志系统中日志存放的位置。

默认值：由**DB_DbDir/LgDir**指定的路径。

取值范围：字符串长度<256。

请参考：DB_DbDir、DB_LgSvr

用于何处：服务器端

DB_LgErr=<值>

当关键字 **DB_LgSvr** 的值是 1~4，即日志系统要记录错误信息时，该关键字用来指定要记录错误信息的级别。设置 **DB_LgErr**=0，将记录错误号大于 30000 的内存泄露或数据库崩溃错误；设置 **DB_LgErr**=1，将记录错误号大于 20000 的断开连接错误或数据库崩溃错误；设置 **DB_LgErr**=2，将记录错误号大于 10000 的中断、断开连接或数据库崩溃错误；设置 **DB_LgErr**=3，将记录错误号大于 100 的常见错误、中断、断开连接或数据库崩溃错误；设置 **DB_LgErr**=4，将记录错误号大于 0 的警告信息或任何错误信息。

默认值：3

取值范围：0 、1 、2 、3 、4

请参考：DB_LgSvr

用于何处：服务器端

DB_LgFNo=<值>

DBMaster将日志系统记录的日志存储在日志文件DBNAME_1.LOG中，当日志文件的大小达到默认值100 MB 或者由关键字**DB_LgFSz**所指定的值时，日志信息将被记录到下一个日志文件中，例如：DBNAME_2.LOG, DBNAME_3.LOG …, DBNAME_n.LOG，该关键字用来指定存储日志的日志文件个数和存储其他信息的文本文件个数。

默认值：20

取值范围：2 ~ 255

请参考：**DB_LgSvr**、**DB_LgFSz**、**DB_LgDay**

用于何处：**服务器端**

DB_LgFSz=<值>

该关键字用来指定一个日志文件和一个文本文件的大小，单位为MB。DBMaster将日志系统记录的信息存储在日志文件DBNAME_1.LOG中，当日志文件的大小达到该关键字所指定的值时，日志信息将被记录到下一个日志文件中，例如：DBNAME_2.LOG、DBNAME_3.LOG ……、DBNAME_n.LOG，直到文件的个数达到默认的20个或由关键字**DB_LgFNo**指定的个数，信息再重新记录到第一个日志文件DBNAME_1.LOG中。

默认值：100

取值范围：10 ~ 1500

请参考：**DB_LgSvr**、**DB_LgFNo**

用于何处：**服务器端**

DB_LgLck=<值>

该关键字用来指定当锁超时或死锁发生时，日志系统是否记录此时的锁信息。设置**DB_LgLck=0** 将不会记录锁超时或死锁发生时的锁信息；设置**DB_LgLck=1**将会记录此时的锁信息。

默认值：0

取值范围：0 、1

请参考：DB_LgSvr

用于何处：服务器端

DB_LgPar=<值>

该关键字用来设定日志系统是否记录输入参数的值，设置**DB_LgPar=0**将关闭该功能；设置**DB_LgPar=1**将记录输入参数的值；设置**DB_LgPar=2**将同时记录输入参数的值和存储过程执行的SQL命令；设置**DB_LgPar=3**将记录触发器SQL语句和参数值；设置**DB_LgPar=4**将同时记录存储过程和触发器的SQL语句和参数值。

默认值：0

取值范围：0 、1、2、3、4

请参考：DB_LgSvr

用于何处：服务器端

DB_LgPln=<值>

该关键字用来指定日志系统是否记录select, update或delete语句的执行计划。设置**DB_LgPln=0**将会关闭该功能；设置**DB_LgPln=1**将会记录这些语句的执行计划。

默认值：0

取值范围：0 、1

请参考：DB_LgSvr

用于何处：服务器端

DB_LgSQL=<值>

当关键字**DB_LgSvr**的取值为4时，该关键字用来指定日志系统是否记录SQL命令。设置**DB_LgSQL**=0将不会记录SQL命令；设置**DB_LgSQL**=1，当关键字**DB_LgSvr**=4时，将会记录除Select语句之外的 SQL命令；设置**DB_LgSQL**=2，当关键字**DB_LgSvr**=4时，将会记录所有SQL命令。

默认值：2

取值范围：0 、1、2

请参考：DB_LgSvr

用于何处：服务器端

DB_LgSTm=<值>

该关键字用来设定日志系统记录的间隔时间。仅当关键字**DB_LgSvr** 的取值为 2、3或4，即日志系统需要记录执行较慢的慢语句时才需要该设置。

默认值：5 （秒）

取值范围：1~65536（秒）

请参考：DB_LgSvr

用于何处：服务器端

DB_LgSvr=<值>

该关键字用来设置日志服务器的状态以及日志记录的具体级别。设置**DB_LgSvr**=0，将关闭日志服务器；设置**DB_LgSvr**=1，日志系统将会

记录错误信息，默认记录的错误信息级别由关键字**DB_LgErr**指定；设置**DB_LgSvr=2**，日志系统将记录运行时间比较长的慢语句，其运行时间由关键字**DB_LgSTm**指定；设置**DB_LgSvr=3**，错误信息和慢语句都将被记录；设置**DB_LgSvr=4**，日志系统将会记录连接、断开连接、提交、回滚等操作信息，以及SQL命令、错误信息与慢语句，对于是否记录SQL命令由关键字**DB_LgSQL**来指定；设置**DB_LgSvr=5**，将会在数据库退出时记录所有信息；设置**DB_LgSvr=6**，将会在数据库登录和退出时，记录所有信息。有关日志系统的详细信息，请参考*启动日志系统*章节。

默认值： 0

取值范围：0、1、2、3、4、5、6

请参考：DB_LgErr、DB_LgSTm、DB_LgSQL、DB_LgFSz、DB_LgFNo、DB_LgDir、DB_LgPln、DB_LgPar、DB_LgLck、DB_LgSys

用于何处：服务器端

DB_LgSys=<值>

设置**DB_LgSys=0**，将记录执行时间、错误号码、服务功能、连接地址、用户名称、语句id、连接信息、错误参数以及SQL语句；设置**DB_LgSys=1**，除了记录**DB_LgSys=0**时的所有信息外，还将记录SYSUSER 和 SYSINFO信息；设置**DB_LgSys=2**，除了记录**DB_LgSys=1**时的所有信息外，如果内存可以被监测，还将记录SYSTEM的内存信息。

默认值：0

取值范围：0、1、2

请参考：DB_LgSvr、DB_LgSQL、DB_LgFSz、DB_LgErr、DB_LgSTm

用于何处：服务器端

DB_LgZip=<值>

该选项用来打包或压缩已经关闭的日志文件从而节省存储空间。

默认值：0

取值范围：0、1

请参考：**DB_LgSvr**、**DB_LgDay**、**DB_LgFNo**

用于何处：**服务器端**

DB_LTimO=<值>

此关键字是一个定义锁超时的整型值（单位秒）。当你需要在数据库对象上获取某个锁时，例如表或行，并且此对象已被分配给另一个事务，那么必须等待直到对象被释放。

然而，如果你不想等待，则可设置此关键字为你想要的等候时间。DBMaster将会等待对象直到到达定义等待时间。此时将会返回一个“lock time-out”错误，或者在锁超时前获取了此对象上的锁。如想禁止锁超时，可设置此关键字为-1，这样DBMaster将会继续等待直到锁被释放。

另一情况是设置关键字为0，指出不等待锁。此关键字在连接时会用到而不是数据库起动时。每个连接可以有其自己的 **dmconfig.ini**文件，尤其是在客户端/服务器模式下，因此每个用户都会有一个锁超时值。

默认值： 5

取值范围： -1 ~ 65535

用于何处：**客户端**

DB_MaxCo=<值>

此关键字指出在创建数据库或以新日志模式启动数据库时的硬连接数，并且指出了正常启动数据库时的软连接数。软连接数指的是事务被同时

激活的最大数。它也说明了数据库可支持的同时连接数，因为一个连接只能属于一个事务。

硬连接数取值范围为240~4840，并且为40的倍数，因此**DB_MaxCo**接近于40倍数的值（或240）为硬连接数。有时，硬连接数与DBMaster 序列号相关。当使用新日志模式创建或启动数据库时，如果没有在**dmconfig.ini**配置文件中分配**DB_MaxCo**值，那么硬连接数将取**DB_MaxCo**的默认值和序列号的最大用户数之间比较的较大值，但如果您在**dmconfig.ini**配置文件中分配了**DB_MaxCo**值，那么硬连接数将取**DB_MaxCo**的默认值与用户分配的值之间比较的较大值.。了解更多有关硬连接数或软连接数的信息，请参看18.5*调节并发进程*。

默认值：240

取值范围：240 ~ 4840

请参考：DB_SMode

用于何处：服务器端

DB_MTimO=<值>

此关键字的取值是一个整数，用于定义客户端的latch超时值，单位为秒。 客户端latch在各API函数的入口设置，在退出API函数时释放。使用这个特点可防止多线程应用程序使用同一个连接处理来连接或操作数据库的问题。

设置此关键字为0，表示不等待客户端的latch释放。设置此关键字为任一大于0的整数，表示DBMaster将等待客户端的latch释放以减少多线程使用同一个连接处理同时连接并调用API函数时，问题发生的可能性。

默认值：0

取值范围：0 ~ 65535

用于何处：客户端

DB_MxCmd=<值>

此关键字定义了ODBC应用程序语句处理的最大值，或DCI COBOL应用程序可以打开表（值-1）的最大值。设置此关键字为n，表示在一个连接中，所有的ODBC应用程序至多只能分配n条语句句柄。针对DCI，执行的SQL命令最多只能打开（n-1）张表加上一个语句句柄。

默认值：257

取值范围：1 ~ 32767

用于何处：服务器端

DB_NBufs=<值>

此关键字定义了数据缓存数。用户可以为数据缓存指定单位为M或G。如果没有指定M或G，默认的单位是页。当指定了M 或G为单位时，实际的数据缓存数将比指定的值要少1页。例如，如果数据页大小为16K，设置**DB_NBufs**=8MB，那么实际的数据缓存大小为8176KB，而不是8192 KB。大多数情况下，缓存越大，DBMaster的运行就越高效。在数据库启动时会用到此关键字。

设置值为0时DBMaster检测物理内存使用情况并自动配置缓存大小。如果DBMaster检测物理内存失败，则设置缓存大小为2000页。

➲ **示例**

在启动数据库后，可通过查询SYSINFO表来决定数据库使用的缓存数。下面命令显示了当前数据库运行使用的缓存页为500：

```
dmSQL> SELECT * FROM SYSINFO WHERE INFO = 'NUM_PAGE_BUF';


 ID       INFO            VALUE

==== ============== ================

0107 NUM_PAGE_BUF   500
```

```
1 rows selected
```

了解如何决定最佳值，请参看18章*性能调优*的*调整内存分配*部分。

默认值：0 （自动配置）

取值范围：0、15~根据系统决定

请参考：DB_NJnlB、DB_ScaSz

用于何处：服务器端

DB_NetEc=<值>

此关键字定义启动或关闭DBMaster的网络加密状态。如果网络加密开启，所有DBMaster服务器端和客户端的数据都被加密。DBMaster的加密技术是结合DES和RSA两种技术。

默认值：0（关）

取值范围：0（关） 、1（开）

用于何处：服务器端

DB_NetZc=<值>

该关键字用来指定在服务器与客户端之间传输数据时，是否开启压缩功能。如果开启了压缩功能，数据在服务器端传出时将被压缩，而在客户端收到时即被解压缩。因此减少了传输的数据量，减少了传输时间。设置**DB_NetZc=1**开启该功能，设置**DB_NetZc=0**关闭该功能。

默认值：0 （关）

取值范围：0（关） 、1（开）

用于何处：客户端

DB_NJnlB=<值>

此关键字定义内存中的日志缓存区数量。

用户可以为日志缓存指定单位为M 或G。如果没有指定M或G，默认的单位是页。当指定了M 或G为单位时，实际的日志缓存数将比指定的值要少1页。例如，如果数据页大小为16 K，设置**DB_NJnlB=**8MB，那么实际的数据缓存大小为8176KB，而不是8192 KB。在数据库启动时会用到此关键字。

默认值：64 （64× **DB_PgSiz** KB）

取值范围：16~根据系统而定

请参考：DB_JnlSz、DB_NBufs、DB_ScaSz、DB_PgSiz

用于何处：服务器端

DB_OptRt=<值>

此关键字定义了查询优化器在内部连接和外部连接中选择嵌套连接或合并连接的依据，默认值为0。值为1时，查询优化器会考虑响应时间而不考虑执行时间；值为0时，查询优化器会考虑执行时间而不考虑响应时间。

默认值：0

取值范围：0、1

用于何处：服务器端

DB_Order=<字符串>

此关键字定义代码页文件名称，它位于DBMaster 安装路径的**shared/codeorder**子目录下。代码页文件是纯文本文件，它将会影响DBMaster的排序结果。在数据库创建时会用到此关键字，之后就不起作用了。 无此关键字，排序序列将是二进制序列。

默认值：无

取值范围：用户定义的代码页名称

请参考：DB_LCode

用于何处：服务器端 （仅在创建数据库时用到）

DB_PasWd=<字符串>

此关键字定义默认的用户登陆密码。如果没有定义默认的用户登陆ID，此值就被忽略。在数据库启动或连接时会用到此关键字。

默认值：空

取值范围：字符串长度 < 16

请参考：DB_UsrId

用于何处：客户端

DB_PgSiz=<值>

该关键字用来指定一个数据页的大小。DBMaster支持4k、8k、16k或32k的数据页大小。该关键字只能在创建数据库时使用。

默认值：8

取值范围： 4、8、16、32

用于何处：服务器端（仅在创建数据库时使用）

DB_PtNum=<值>

此关键字定义数据库服务器端的TCP/IP端口号。主要用于客户端的数据库连接及服务器端数据库启动。对于数据库来说客户端与服务器端的端口号必须匹配，否则连接会失败。

默认值：无

取值范围：1025 ~ 65535

请参考：DB_SvAdr

用于何处：客户端和服务器端

DB_ResWd=<值>

有时，用户也需要将DBMaster的保留字作为标识符往数据库中添加（创建或输入）对象（请参考*SQL命令与函数参考手册*的*保留字列表*）。如果将**DB_ResWd**关键字设置为1，那么在将保留字作为标识符添加对象时，将返回一个错误信息；如果将**DB_ResWd**关键字设置为0，那么DBMaster将不会返回错误信息。设置此关键字的主要目的就是允许输入的对象中包含保留字。

默认值：1

取值范围：0、1

用于何处：**服务器端**

DB_RmPad=<值>

此关键字定义是否删除所有CHAR类型数据尾部的空格。值为0时，保留CHAR类型数据尾部的空格；值为1时，将CHAR类型数据拷贝到用户缓存区前删除所有尾部空格。而用户应用程序可重新获取固定长度的CHAR类型数据，在插入数据时去掉尾部填充的空格。

默认值：0

取值范围：0、1

用于何处：**客户端**

DB_RstSn=<值>

此关键字定义当serial字段达到最大值时是否自动重置为初始值。值为0时，不自动重置；值为1时，自动重置。

默认值：0

取值范围：0、1

用于何处：**服务器端**

DB_RTime=<字符串>

此关键字定义从备份还原数据库的目的时间。在执行数据库还原时，DBMaster从最早备份时间到**DB_RTime**定义的时间对数据文件进行前滚。 如果没有定义**DB_RTime**，DBMaster将会把数据库还原到最新文件备份时间，这个最新时间为备份执行时间。

如果**DB_RTime**时间比备份时间晚，则使用实际备份时间来定义**DB_RTime**。

DB_RTime格式为**yy/mm/dd hh:mm:ss**或**yyyy/mm/dd hh:mm:ss**。

默认值：0（70/1/1 00:00:00 ）

用于何处：服务器端

DB_ScaSz=<值>

此关键字定义系统控制区(SCA)的大小（单位为页）。用户可以指定单位为M 或G。如果没有指定M或G，默认的单位是页。当指定了M或G为单位时，实际的日志系统控制区大小将比指定的值要少1页。例如，如果数据页的大小为16K，设置**DB_ScaSz**=8MB，那么实际的数据缓存大小为8176KB，而不是8192KB。

如果在DBMaster中SCA实际所需要的最小内存值大于**DB_ScaSz**的设置值时，DBMaster将忽略设置值并为SCA分配其所需要的最小内存值。在数据库启动时会用到此关键字。

默认值：200 （200 × **DB_PgSiz** KB ）

取值范围：1~（根据系统而定）

请参考：DB_NBufs、DB_NJnlB、DB_PgSiz

用于何处：服务器端

DB_SchLgDir=<字符串>

此关键字定义dmschsvr日志文件的存储目录。如果没有设置该关键字，日志文件将被存储在**DB_DbDir**指定的存储目录中。

默认值：由**DB_DbDir**指定的目录

取值范围：字符串长度 <256

请参考：**DB_DbDir**、**DB_SchSv**、**DB_SchLgLev**

用于何处：**服务器端**

DB_SchLgLev=<值>

此关键字定义**dmschsvr**的5个日志级别：

- 记录**dmschsvr**运行信息：将**DB_SchLgLev**的值设置为0，只记录**dmschsvr**后台服务的运行状态。
- 记录任务错误信息以及**dmschsvr**运行信息：将**DB_SchLgLev**的值设置为1，记录任务的错误信息和**dmschsvr**后台服务的运行状态。
- 记录任务警告、错误信息以及**dmschsvr**运行信息：将**DB_SchLgLev**的值设置为2，记录任务的警告信息、错误信息以及**dmschsvr**后台服务的运行状态。
- 记录任务、计划的全部信息以及**dmschsvr**运行信息：将**DB_SchLgLev**的值设置为3，记录任务、计划的全部信息以及**dmschsvr**后台服务的运行状态。
- 记录**dmschsvr**运算信息：将**DB_SchLgLev**的值设置为4，记录**dmschsvr**的运算信息，以便用户监控**dmschsvr**后台服务的运行状态。

默认值：1

取值范围：0、1、2、3、4

请参考： **DB_SchSv, DB_SchLgDir**

用于何处：服务器端

DB_SchSv=<值>

此关键字定义了是否允许开启**dmschsvr**服务。值为0时，表示允许**dmschsvr**服务，值为1时，表示禁止**dmschsvr**服务。

默认值：0

取值范围：0、1

请参考：DB_TskNo、DB_SchLgDir、DB_SchLgLev

用于何处：服务器端

DB_SMode=<值>

此关键字定义数据库启动模式，它有6种模式：

- **正常启动** — 正常启动系统。如果数据库出现故障，DBMaster 将自动执行灾难恢复以保证数据库的稳定性。
- **以新日志模式启动** — 正常启动系统。此模式下将会创建一个新日志文件，文件名根据 关键字**DB_JnFil**的值来定义。如果这个文件已存在，它将会被新文件覆盖。
- **还原模式启动** — 利用备份数据库文件（包括日志文件）来启动数据库。在这种模式下DBMaster将所有操作及时前滚到**DB_RTime**指定点。数据库还原时会用到此模式。欲了解更多信息请参考数据库恢复，备份和还原。
- **以主数据库模式启动** — 这种模式主要用于数据库复制。在此模式下启动数据库系统可使其作为一个主数据库，例如：复制中的源数据库。了解更多信息请参看部分。
- **以从数据库模式启动** — 这种模式主要用于数据库复制。在此模式下启动数据库系统可使其作为从数据库，例如：复制中的目的数据库。了解更多信息请参看*数据库复制*部分。

- **以只读数据库方式启动 —** 在正常状况下启动数据库，但数据库状态为只读或仅提供读权限，以只独方式启动数据库可禁止用户对其修改。

默认值：1

取值范围：1（正常启动）

2（以新日志模式启动）

3（以还原模式启动）

4（以主数据库模式启动）

5（以从数据库模式启动）

6（以只读数据库方式启动）

请参考：DB_ForcS

用于何处：服务器端

DB_SPDir=<字符串>

此关键字定义存储过程文件的存放路径。存储过程文件包括通用动态链接库文件和存储过程创建时生成的所有临时文件。**DB_SPDIR**必须以完整路径名定义。

➲ **示例1**

UNIX环境下**DB_SPDir**的示例如下：

```
DB_SPDir = /usr/DBMaster/data/spdir          ;in UNIX
```

➲ **示例2**

Windows环境下**DB_SPDir**的示例如下：

```
DB_SPDir = c:\DBMaster\data\spdir            ;in Windows
```

默认值：<数据库路径>

取值范围：字符串长度 < 256

请参考：DB_SPInc

用于何处：服务器端

DB_SPInc=<字符串>

此关键字定义存储过程中include文件的存放路径。存储过程中使用外部include文件时会用到此关键字。**DB_SPInc**需用完整路径名定义。

➲ **示例1**

UNIX环境下**DB_SPInc**的路径如下：

```
DB_SPInc = /usr/DBMaster/data/sp/include        ;in UNIX
```

➲ **示例2**

Windows环境下**DB_SPInc**的路径如下：

```
DB_SPInc = c:\DBMaster\data\sp\include          ;in Windows
```

默认值：（dmserver运行的当前目录）

取值范围：字符串长度 < 256

请参考：DB_SPDir

用于何处：服务器端

DB_SPLog=<字符串>

此关键字定义存储过程日志文件的存放路径。存储过程日志文件，它包括在创建存储过程时，从服务器端发送来的错误日志文件及存储过程执行时产生的跟踪文件。**DB_SPLog**需用完整路径名定义。

➲ **示例1**

UNIX环境下**DB_SPLog**的路径如下：

```
DB_SPLog = /usr/joe/mydata/splog                ;in UNIX
```

➲ **示例2**

Windows环境下**DB_SPLog**的路径如下：

```
DB_SPLog = c:\user\joe\mydata\splog            ;in Windows
```

默认值：（客户端应用程序运行的当前目录）

取值范围：字符串长度 < 256

用于何处：客户端

DB_SQLSt=<值>

此关键字定义SQL命令监控器的显示模式。这将会影响系统表**SYSUSER**中**SQL_CMD**和**TIME_OF_SQL_CMD**字段的显示内容。SQL命令监控器可在SQL命令执行时获取精确或大概的SQL信息。获取精确信息比获取大概信息消耗CPU的时间多。您也可以关闭SQL命令监控器以避免CPU的超负载。

默认值：1

取值范围：0（关闭 SQL 命令监控）

1（显示SQL命令和大概的SQL 命令的执行时间）

2（显示 SQL 命令和精确的SQL命令的执行时间）

用于何处：服务器端

DB_StACL=<值>

此关键字定义了在用户连接数据库时，服务器是否检查用户的IP地址。

设置**DB_StACL=**1表示启用ACL检查功能，设置**DB_StACL=**0表示禁止ACL检查功能。

默认值：0

取值范围： 0、1

用于何处：服务器端

DB_StMod=<值>

此关键字定义更新统计后台程序模式。将此关键字设置为0表示开启**普通**模式，采样率由**dmconfig.ini**中设置的**DB_STSSP**值决定；设置为1表示开启更新统计后台程序的**表设置**模式，也就是说更新统计值所使用的采样率由各个表的统计模式和采样率决定，用户可以用“UPDATE STATISTICS SET”命令设置每个表的统计模式和采样率。

默认值：0

取值范围：0、1

请参考：DB_StSvr、DB_StsSp、DB_StsTm、DB_StsTv

用于何处：服务器端

DB_StpWd=<字符串>

此关键字定义了位于DBMaster安装目录的**shared\stopword**子目录中，忽略词列表定义文件的文件名称。忽略词列表定义文件是一个纯文本文件，它可以影响DBMaster的全文检索结果。在数据库创建和查询全文检索时可用到此关键字。如果没有定义这个关键字，那么数据库将依据Lcode搜索预定义的忽略词列表。

➲ **示例**

```
[DB1]
DB_LCode = 2
DB_StpWd = /home/usr/jp.tab
```

默认值：

DB_LCode	忽略词列表定义
0：英语（ASCII）	en.tab
1：繁体中文（BIG5）	tw.tab
2：日语（Shift-JIS + Half Corner）	jp.tab

3：简体中文（GB） cn.tab

4：拉丁语（ISO-8859-1） en.tab

5：拉丁语（ISO-8859-2） en.tab

6：斯拉夫语（ISO-8859-5） en.tab

7：希腊语（ISO-8859-7） en.tab

8：日文（EUC-JP） jp.tab

9：简体中文（GB18030） cn.tab

10：Unicode（UTF-8） en.tab

取值范围：用户定义的忽略词列表定义文件的文件名称

请参考：DB_LCode

用于何处：服务器端

DB_StrOP=<值>

此关键字定义在进行字串连接操作时是否删除字符串尾部空格。值为0，表示在执行字串连接操作前保留CHAR类型数据尾部的空格；值为1，表示在执行字串连接前删除其尾部空格。

此关键字可在客户端或服务器端设置。如果在客户端的**dmconfig.ini**中没有设置，那么将使用在服务器端的**dmconfig.ini**文件中提供的参数值。服务器端的该参数默认值是0。

默认值：0

取值范围：0、1

用于何处：客户端或服务端

DB_StrSz=<值>

此关键字定义使用用户自定义函数（UDF）时，STRING类型数据的返回长度。因为UDF只能返回固定长度的字串，所以此关键字可以限制STRING数据的字串长度，以避免客户端接收太长字串。

默认值：255

取值范围：1 ~ 4096

用于何处：客户端或服务器端(客户端具有较高优先级)

DB_StsSp =<值>

此关键字定义更新统计页数据采样率。将此关键字设置为-1表示智能获取采样率，设置为0表示不更新统计值，另外用户可自行设置采样率，取值范围是1~100。

默认值：100

取值范围：-1、0、1 ~ 100

请参考：DB_StSvr、DB_StMod、DB_StsTm、DB_StsTv

用于何处：服务器端

DB_StsTm=<字符串>

此关键字定义更新统计后台程序首次自动更新统计值的时间，格式为**yyyy-mm-dd hh:mm:ss**。

默认值：1970-01-01 03:00:00

取值范围：1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

请参考：DB_StSvr、DB_StMod、DB_StsSp、DB_StsTv

用于何处：服务器端

DB_StsTv=<字符串>

此关键字定义更新统计后台程序时间间隔，设置为“1-12:30:00”表示时间间隔是一天12小时30分钟。

默认值：1-00:00:00

取值范围：0-00:00:01~24854-23:59:59

请参考：**DB_StSvr**、**DB_StMod**、**DB_StsSp**、**DB_StsTm**

用于何处：**服务器端**

DB_StSvr=<值>

此关键字用来激活自动更新统计服务。值为1，启动统计服务器；值为0，关闭统计服务器。如果自动更新统计服务被激活，那么它将会根据**DB_StsTm**和**DB_StsTv**的时间安排对数据库统计值进行更新。

默认值：1

取值范围：0、1

请参考：**DB_StMod**、**DB_StsTv**、**DB_StsTm**、**DB_StsSp**

用于何处：**服务器端**

DB_SvAdr=<字符串>

此关键字定义服务器的TCP/IP地址或机器的主机名。如果在客户机上安装了DNS(域名服务)，那么可通过此关键字来指明域名。在连接时刻，所有客户机和服务器都会获取此关键字。如果这个地址定义不正确，连接就会失败。了解更多信息可参考网络管理员手册或TCP/IP网络手册。

默认值：无

取值范围：a、b、c、d 或主机名（1<=a,b,c,d <=254）

请参考： **DB_PtNum**

用于何处：**客户端和服务器端**

DB_SvLog=<值>

此关键字用来指定**dmserver**命令行的文本是否保存到文件中。此功能开启后，<数据库路径>\<数据库名称>.log会以ASCII文本方式来保存。拥有DBA权限的用户可使用此功能处理连接错误。值为1时保存文件，值为0时不保存文件。

默认值：0

取值范围：0、1

用于何处：服务器端

DB_TCPIP=<值>

此关键字定义了**dmserver**是否只允许TCP/IP网络协议。值为0时，表示**dmserver**允许从命名管道和TCP/IP协议连接；值为1时，则表示**dmserver**只允许从TCP/IP协议连接。

默认值：0

取值范围： 0、1

用于何处：服务器端

DB_TmiFm=<字符串>

此关键字定义SQL语句的时间输入格式。了解更多信息请参考*ODBC编程指导手册*的附录B*函数特性差异*。

默认值：无（可接受所有时间输入格式）

取值范围：hh:mm:ss.fff

hh:mm:ss

hh:mm

hh

hh:mm:ss.fff tt

hh:mm tt

hh tt

tt hh:mm:ss.fff

tt hh:mm:ss

tt hh:mm

tt hh

hh mm ss

请参考：DB_TmoFm

用于何处：客户端或服务器端（客户端优先级较高）

DB_TmoFm=<字符串>

此关键字定义sql语句的时间输出格式。了解更多信息请参考*ODBC程序员参考手册*。

默认值：hh:mm:ss

取值范围：类似于**DB_TmiFm**的取值范围

请参考：DB_TmiFm

用于何处：客户端和服务器端（客户端具有较高优先级）

DB_TMPDir=<字符串>

此关键字定义临时表空间文件的存放目录。在该目录下，系统将创建数据文件和BLOB文件：数据文件的逻辑名称为**DB_TMPDB**，物理名称为**DB_TMPDir/DBNAME.TDB**；BLOB文件的逻辑名称为**DB_TMPBB**。物理名称为**DB_TMPDir/DBNAME.TBB**。

DB_TMPDB和**DB_TMPBB**仅是保留字，不是关键字，因此在**dmconfig.ini**配置文件中，我们不能定义**DB_TMPDB**和**DB_TMPBB**。

默认值：DB_DbDir/tmpDir

请参考：DB_DbDir

用于何处：服务器端

DB_TpFil=<字符串>

该关键字用于定义系统临时文件的名称。文件大小限制为4GB，用户最多可以定义128个系统临时文件。

如果指定了系统临时文件的完整名称，那么该系统临时文件将被存储在完整名称指定的目录中。否则，将存储在**DB_IttDir**指定的目录。

用户应该在启动数据库之前设置此关键字。如果在数据库运行期间设置，那么该设置将在下次启动数据库时生效。ITTs（内部临时表）最初存储在内存中，但如果其大小增长到一定程度，Dmserver将把ITTs数据写入磁盘上的系统临时文件，然后释放内存以分配给ITTs。系统会自动生成128个系统临时文件，最大值为4GB。系统临时文件名的格式为：<*数据库名的前8个字符*><*序列号*>*.TMP*（序列号的值是介于1到128之间的整数）。如果这些系统临时文件从未被使用，那么将被自动删除以节约磁盘空间，需要的时后再进行重建。此外，当数据库关闭时，这些文件也将被一并删除。

一般情况下，系统临时文件由系统自动生成，但用户也可以通过关键字DB_Tpfil自行修改其名称和路径。然而，手动设置128个系统临时文件太过繁琐费时，因此不推荐使用此关键字。此关键字也不会在SYSCONFIG中显示。

默认值： 系统临时文件名的格式：<*数据库名的前8个字符*><*序列号*>*.TMP*（序列号的值是介于1到128之间的整数），（例DB1.TMP）。

取值范围：最多定义8个字串，每个字串的长度不限，字串间以逗号和空格分隔。

请参考：DB_DbFil、DB_BbFil、DB_IttDir

用于何处：服务器端

DB_TskNo=<值>

此关键字定义了**dmschsvr**同时可以唤醒的任务数量。

默认值：30

取值范围：1 ~ 50

请参考：**DB_SchSv**

用于何处：**服务器端**

DB_Turbo=<值>

此关键字设置DBMaster使用正常系统表缓冲的模式。如果应用程序不对数据库结构信息进行频繁修改，可使用**DB_Turbo**来加速数据访问。了解更多信息请参考*性能调优*部分。数据库启动时会用到此关键字。

默认值：0

取值范围：0、1

用于何处：**服务器端**

DB_UsrBb=<字符串>

此关键字用于定义用户所创建的数据库文件名称，它定义了被操作系统识别的物理文件名称，该文件用于存默认的用户BLOB数据。

在物理文件名后用户应该指定该文件的大小，文件大小的单位可以为Frame、M或G。请注意，如果使用M或G作为单位，实际的文件大小将比指定的数值要少1帧（Frame）。

➲ **示例**

定义默认的用户BLOB 文件名，大小为 20 帧：

```
[MY_DB]                                          ;database name
DB_UsrBb = /disk1/usr/f1.bb  20                  ;blob file
```

默认值：数据库名及 **.BB**扩展名，大小为 3 帧。

取值范围：字符串长度 < 256

请参考：**DB_BbFil**、**DB_DbDir**、**DB_DbFil**、**DB_UsrDb**、用户自定义文件名。

用于何处：**服务器端**

DB_UsrDb=<字符串>

此关键字用于定义用户所创建数据库的文件名称，它定义了被操作系统识别的物理文件名称，该文件用于存放默认的用户普通数据。

在物理文件名后用户应该指定该文件的大小，文件大小的单位可以为Page、M或G。请注意，如果使用M或G作为单位，实际的文件大小将比指定的数值要少1页（Page）。

➲ **示例**

定义默认用户数据文件名，文件大小为200页：

```
[MY_DB]                                          ;database name
DB_UsrDb = /disk1/usr/f1.db  200                 ;data file
```

默认值：数据库名和.**DB**文件扩展名，默认为200页。例如：**db.DB**。

取值范围：字符串长度 < 256

请参考：**DB_BbFil**、**DB_DbDir**、**DB_DbFil**、**DB_UsrBb**，用户自定义文件名

用于何处：**服务器端**

DB_UsrFo=<字符串>

此关键字定义是否允许在数据库中插入用户文件对象。值为1，则可插入用户文件对象。了解更多信息请参看*大型对象管理*。在数据库启动时用到此关键字。

默认值：0

取值范围：0、1

请参考：DB_FoDir

用于何处：服务器端

DB_UsrId=<字符串>

此关键字定义在数据库启动或连接时使用的默认用户ID。

默认值：空

取值范围：字符串长度 < 128

请参考：DB_PasWd

用于何处：客户端

DB_WsorT=<值>

该关键字用来指定文字排序指令的大小写敏感性。默认值为0；设置**DB_WsorT**=1，则文字排序指令是大小写不敏感的；设置**DB_WsorT**=2，则文字的排序指令是大小写敏感的。

默认值：0

取值范围：0、1、2

请参考：DB_LCode、DB_Order

用于何处：服务器端

DD_CTimO=<值>

此关键字定义在DDB模式下连接远程数据库的连接超时值。在DDB环境下，协作者数据库必需与远程数据库之间建立分布式连接。

默认值：5（秒）

取值范围：1或更高

请参考：DD_DDBMd、DD_LTimO、DB_CTimO

用于何处：服务器端

DD_DDBMd=<值>

此关键字定义是否在数据库服务器端启动分布式服务功能（DDB）。 设置此值后为开启状态后（1），可使用DDB及表复制功能。

默认值：0

取值范围：0、1

请参考：DD_GTSvr

用于何处：服务器端

DD_GTItv=<字符串>

此关键字用来设置GTRECO后台服务程序的执行计划，GTRECO后台服务程序用于解决未决全局事务。仅当GTRECO服务器开启时有效。

输入格式为：'D hh:mm:ss'

默认值：00:10:00

取值范围：0 ~ 24855天

请参考：DD_GTSvr

用于何处：服务器端

DD_GTSvr=<值>

此关键字定义在DDB模式下是否启动GTRECO(全局事务恢复)后台程序。GTRECO后台程序将自动解决通过DBMaster服务器的未决全局事务。

默认值：1

取值范围：0、1

请参考：DD_DDBmd、DD_GTltv

用于何处：服务器端

DD_LTimO=<值>

此关键字定义在DDB模式下分布式数据访问的锁超时值。这只在服务器与服务器之间的数据访问中有效。要想了解更多相关信息，请参看**DB_LTimO**。

默认值：5（秒）

取值范围： >=-1

请参考：DD_DDBmd、DD_CTimO、DB_LTimO

用于何处：服务器端

DM_DifEn=<值>

此关键字在**DM_COMMON_OPTION**中设置，用来定义有请求发出时是否分配新的环境句柄。**dmconfig.ini**中**DM_COMMON_OPTION**的内容是对数据库的全局设置。**DM_DifEn**关键字对**dmconfig.ini**文件中的所有数据库均有效。如无特殊情况发生，建议DBMaster用户不要随意改变此关键字。

默认值：1

取值范围：0、1

用于何处：客户端

LG_NPFun=<字符串>

此关键字在**DM_COMMON_OPTION**节中设置，提供不记录日志功能。它的值为空字串或一些以逗号分隔的ODBC函数名。当**dmconfig.ini**中不

定义**LG_PTFun**关键字时，此关键字才有效。一旦定义了此关键字，函数列表字串将不会记录日志。

默认值：""（空字串，记录所有函数）

取值范围：函数列表字串（例如：“SQLError、SQLGetDiagRec”）

请参考：LG_Path、LG_PTFun、LG_Trace、LG_Time

用于何处：客户端

LG_Path=<字符串>

此关键字仅在**DM_COMMON_OPTION**节中设置，定义输出日志文件的文件路径名。

默认值：c:\odbclog.log（for Windows）、./odbclog.log （for UNIX）

取值范围：字符串长度 < 256

请参考：LG_NPFun、LG_PTFun、LG_Time、LG_Trace

用于何处：客户端

LG_PTFun=<字符串>

此关键字在**DM_COMMON_OPTION**节中设置，用来定义记录日志功能。 它的取值为空字串或一些ODBC函数名，函数名以逗号分隔。一旦定义此关键字，字串中的函数列表将记录到日志。如果同时设置**LG_PTFun**和**LG_NPFun**关键字，则起作用的为**LG_PTFun.**

默认值：无（记录所有函数）

取值范围：函数列表字串（例如： “SQLError、SQLGetDiagRec”）

请参考：LG_NPFun、LG_Path、LG_Time、LG_Trace

用于何处：客户端

LG_Time=<值>

此关键字在**DM_COMMON_OPTION**节中设置，用来定义是否记录每个函数的耗费时间。值为1则记录每个函数的耗费时间，值为0则不记录时间。此功能可帮助用户发现ODBC程序中的瓶颈问题。

默认值：0

取值范围：0、1

请参考：**LG_NPFun**、**LG_Path**、**LG_PTFun**、**LG_Trace**

用于何处：客户端

LG_Trace=<值>

此关键字在**DM_COMMON_OPTION**节中设置，用来定义是否开启ODBC日志。值为1，开启ODBC日志；值为0，关闭ODBC日志。在ODBC日志开启状态时，ODBC函数调用，输入参数，输出参数和返回代码或错误信息都会记录在日志文件中。要想了解更多信息可参考以下的关键字。

默认值：0

取值范围：0、1

请参考：**LG_NPFun**、**LG_Path**、**LG_PTFun**、**LG_Time**

用于何处：客户端

RP_BTime=<值>

此关键字用来定义数据库复制的起始时间。格式为**YYYY/MM/DD hh:mm:ss**，例如：2000/1/1 01:30:00。可使用**RP_Iterv**关键字定义数据库复制计划。

默认值：主数据库启动时间

取值范围：YYYY/MM/DD hh:mm:ss（例如 2000/1/1 00:00:00）

请参考：DB_SMode、RP_Clear、RP_Iterv、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

用于何处：主数据库服务器端

RP_Clear=<值>

此关键字定义在数据库复制时，将备份文件复制到远程数据库后是否清除备份文件。值为1，则清除备份文件；值为0，不清除。

默认值：0 (no)

取值范围：0 (no),、1 (yes)

请参考：DB_SMode、RP_BTime、RP_Iterv、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

用于何处：主数据库服务器端

RP_Iterv=<值>

此关键字定义了数据库复制计划。格式为**dd-hh:mm:ss**，例如，1-12:00:00表示1天零12小时。您可以使用**RP_BTime**定义数据库复制的开始时间。

默认值：1-00:00:00 （一天）

取值范围：0天 ~ 24855天

请参考：DB_SMode、RP_BTime、RP_Clear、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

用于何处：主数据库服务器端

RP_LgDir=<字符串>

此关键字定义在异步表复制中，DBMaster复制日志的存放路径。复制日志文件为二进制并且用户不能手动更改它的位置。

默认值：在数据库根目录下的子目录**TRPLOG**

取值范围：字符串长度 < 256

请参考：**DB_AtrMd**、**DB_EtrPt**

用于何处：**主数据库服务器端**

RP_Primy=<字符串>

此关键字定义数据库复制中主数据库地址。

默认值：无

取值范围：a、b、c、d 或主机（域）名（1<a,b,c,d <254）

请参考：**DB_SMode**、**RP_BTime**、**RP_Clear**、**RP_Iterv**、**RP_PtNum**、 **RP_ReTry**、**RP_SlAdr**

用于何处：**从数据库服务器端**

RP_PtNum=<值>

此关键字用于数据库复制中，用来定义目标数据库RP_RECV_SERVER后台服务程序使用的端口号。此端口号要与目标数据库的**DB_PtNum**设置不同，并且与主数据库的**RP_SlAdr**端口号相同。

默认值：23001

取值范围：1024 ~ 65535

请参考：**DB_SMode**、**RP_BTime**、**RP_Clear**、**RP_Iterv**、**RP_Primy**、 **RP_ReTry**、**RP_SlAdr**

用于何处：**从数据库服务器端**

RP_Reset=<值>

RP_Reset关键字定义在启动数据库时是否重置异步表复制系统。如果**RP_Reset**设置为1，DBMaster将清除所有未发送的异步表复制日志，删除所有**RP_LgDir**定义目录下的**.TRP**文件（默认为**DB_DbDir**/TRPLOG），并在启动数据库时重置异步表复制状态。因此

DBMaster将忽略所有未发送的异步表复制数据。启动数据库之后，DBMaster将**RP_Reset**的状态值重置为0，以避免在下次启动数据库时重复操作。

默认值：0

取值范围：0、1

请参考：RP_LgDir

用于何处：主数据库的服务器端

RP_ReTry=<值>

此关键字定义数据库复制中，当网络连接失败后，DBMaster重新连接远程数据库的次数。

默认值：0

取值范围：0 ~ 2147483647

请参考：DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_Primy、 RP_PtNum、RP_SlAdr

用于何处：主数据库的服务器端

RP_SlAdr=<字符串>

此关键字用于数据库复制功能中，用来定义目标数据库地址，将向这些目标数据库发送数据。DBMaster中一个主数据库最多支持8个目标数据库。

➲ **示例1**

RP_SlAdr语法如下：

```
RP_SlAdr = address[:port number] {, address[:port number]}
```

默认端口号为23001。在从端数据库,可用逗号或空格分隔信息。

➲ **示例2**

RP_SlAdr端口号如下：

```
RP_SlAdr = 192.168.9.222:5100, Server2:5101, Server3
```

有三个从数据库：一个为 192.168.9.222，端口号为5100，另一个为 Server2，它的端口号为 5101，第三个为 Server3，默认端口号为 23001。

默认值：无

请参考：DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_Primy、 RP_PtNum、RP_ReTry

用于何处：主数据库的服务器端

用户自定义文件名=<physical filename> <pages>

当表空间被写满时，DBMaster允许用户在表空间中添加新的数据文件或BLOB文件。在创建文件时，用户会定义一个逻辑文件名，这个名字不需要为全路径名。用户可将逻辑文件名与物理文件路径名相匹配，以便于操作系统访问此文件。

➲ **在dmconfig.ini文件中匹配文件：**

```
FILE1 = /disk1/usr/datafile 100
```

在DBMaster中定义了**FILE1**，DBMaster则会在 **/disk1/usr/datafile**下创建一个文件，文件大小为100 页（页的大小由关键字**DB_PgSiz**指定）。如果这个文件必须被移到另一个目录，则需在**dmconfig.ini**中修改物理文件名。因此无需修改你的程序或SQL脚本。**DB_DbDir**规则也用于用户自定义文件。

取值范围：文件名— 字符串长度 < 256

文件大小 —3 ~ 2147483647

请参考：DB_DbDir、DB_UsrBb、DB_UsrDb

用于何处：服务器端

21 系统目录参考

对于关系型数据库，定义部分就是所有的数据库信息必须与用户数据在逻辑层次上表述相同。这些信息被存储于系统表中，与访问其它数据表相同被授予权限的用户可通过SQL语句来访问这些系统表。DBMaster*系统目录参考*章节中包括了系统表的描述，按名称的字母顺序排列。可以通过查询这些系统表来查看数据库具体状态。

21.1 系统目录

系统目录是一组包含数据库中所有对象信息的表。系统表就是通常所说的*数据字典*。

所有系统目录表都属于SYSTEM，并且任何具有连接权限级别的用户都可读取。因为系统目录表属于SYSTEM，您不能删除一个系统目录表或者修改系统目录表结构。

拥有DBA、SYSDBA和SYSADM权限的用户可以读取所有系统目录表。拥有RESOURDE和CONNECT权限的用户无法读取以下10个系统目录表：SYSAUTHGROUP、SYSCONFIG、SYSFILE、SYSFILEOBJ、SYSGLBTRANX、SYSLOCK、SYSPENDTRANX、SYSTRPJOB、SYSTRPPOS、SYSWAIT。

拥有不同权限的用户对以下32个系统目录表拥有不同的读取权限：SYSACL、SYSAUTHCOL、SYSAUTHEXE、SYSAUTHMEMBER、SYSAUTHTABLE、SYSAUTHUSER、SYSCMDINFO、SYSCOLUMN、SYSCONINFO、SYSDBLINK、SYSDBDESCOL、SYSFOREIGNKEY、SYSINDEX、SYSINDEXREF、SYSINFO、SYSJARFILE、SYSJAVAARGU、SYSOPENLINK、SYSPROCINFO、SYSPROCJAVA、SYSPROCPARAM、SYSPROJECT、SYSPUBLISH、SYSSCHEMA、SYSSUBSCRIBE、SYSSYNONYM、SYSTABLE、SYSTABLESPACE、SYSTEXTINDEX、SYSTRIGGER、SYSUSER、SYSVIEWDATA。

所有用户均可读取以下6个系统目录表：SYSDOMAIN、SYSSCHEDULE、SYSSCHELOG、SYSTASK、SYSTRPDEST、SYSTRPDEST。

系统目录表共计48个。

21.2 DBMaster系统目录表

下面的表列出了所有DBMaster中的系统目录表以及相应表内容的简短描述。

表名	内容
SYSACL	IP地址权限信息
SYSAUTHCOL	字段权限信息
SYSAUTHEXE	可执行对象权限信息
SYSAUTHGROUP	组信息
SYSAUTHMEMBER	组成员信息
SYSAUTHTABLE	表权限信息
SYSAUTHUSER	安全级别信息
SYSCMDINFO	存储命令信息
SYSCOLUMN	字段信息
SYSCONFIG	配置信息
SYSCONINFO	连接信息
SYSDBLINK	数据库链接信息
SYSDESCOL	动态字段信息
SYSDOMAIN	定义域信息
SYSFILE	文件信息
SYSFILEOBJ	文件对象信息
SYSFOREIGNKEY	外键信息
SYSGLBTRANX	DDB 全局事务信息
SYSINDEX	索引信息

表名	内容
SYSINDEXREF	索引和自动索引信息
SYSINFO	数据库系统信息
SYSJARFILE	JAR信息
SYSJAVAARGU	Java参数信息
SYSLOCK	锁信息
SYSOPENLINK	开放链接信息
SYSPENDTRANX	未决分布式事务信息
SYSPROCINFO	存储过程信息
SYSPROCJAVA	Java存储过程信息
SYSPROCPARAM	存储过程参数信息
SYSPROJECT	ESQL工程信息
SYSPUBLISH	表复制的源信息
SYSSCHEDULE	计划任务信息
SYSSCHELOG	记录要执行的计划任务的信息
SYSSCHEMA	模式信息
SYSSUBSCRIBE	表复制目的信息
SYSSYNONYM	同义字信息
SYSTABLE	表信息
SYSTABLESPACE	表空间信息
SYSTASK	计划的行为信息
SYSTEXTINDEX	全文索引信息
SYSTRIGGER	触发器信息
SYSTRPDEST	表复制信息
SYSTRPJOB	所有复制工作的记录信息

表名	内容
SYSTRPPOS	发布者删除事务日志文件信息
SYSUSER	用户登录数据库信息
SYSUSERFUNC	用户自定义函数信息
SYSVIEWDATA	视图信息
SYSWAIT	连接等候信息

SYSACL

SYSACL表列出了用户可以连接数据库的IP地址。拥有RESOURCE或CONNECT权限的用户只能读取自身信息；拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息；所有的用户均可读取公共信息。

该表包含两个字段：用户名和IP地址。用户名PUBLIC表示所有用户都必须满足设置，“*”表示所有IP地址都允许连接数据库。

字段名	描述
USER_NAME	用户名
ADDRESS	IP地址
PRIVILEGE	允许—允许该IP地址 禁止—禁止该IP地址

SYSAUTHCOL

SYSAUTHCOL表列出了赋予用户对象权限的所有表的字段。拥有RESOURCE或CONNECT权限的用户可读取被授予权限字段的权限信息，拥有DBA、SYSDBA或SYSADM权限的用户可读取所有权限信息。如果用户在特定表的所有字段上被授予一些执行操作如：INSERT、UPDATE或REFERENCE（即在表SYSAUTHTABLE中字段INS_ALL、

UPD_ALL或REF_ALL的返回值为1），那么在表SYSAUTHCOL中INS、UPD、REF字段的返回值将会被忽略。

字段名	描述
COLUMN_NAME	被赋予权限的字段名
TABLE_NAME	字段所属的表名
GRANTEE	在此字段上被赋予权限的用户名称。必须为有效的用户或组名
TABLE_OWNER	表所属的用户名称
INS	**1** —用户有在特定字段插入数据的权限 **0** —用户没有在特定字段插入数据的权限
UPD	**1** —用户有在特定字段更新数据的权限 **0** —用户没有在特定字段更新数据的权限
REF	**1** —用户有参考特定字段创建约束的权限 **0** —用户没有权限参考特定字段创建约束

SYSAUTHEXE

SYSAUTHEXE表包含可执行对象信息。拥有RESOURCE或CONNECT权限的用户可读取被授予权限字段的权限信息，拥有DBA、SYSDBA或SYSADM权限的用户可读取所有权限信息。

字段名	描述
OBJNAME	可执行对象名称
OWNER	创建执行对象的用户
OBJTYPE	可执行对象类型，如“过程”、“命令”、“工程”等
GRANTEE	对可执行对象授权的用户名

SYSAUTHGROUP

SYSAUTHGROUP 记录数据库中所有有效的组名。拥有RESOURCE或CONNECT权限的用户不能读取任何信息，系统会产生错误代码6829表示用户没有查询权限。只有拥有DBA或更高权限的用户可以创建组；拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
GROUP_NAME	组名
GROUP_OWNER	创建组的用户
NUM_MEMBERS	组中成员数量

SYSAUTHMEMBER

SYSAUTHMEMBER表列出所有同属于一组的成员。拥有RESOURCE或CONNECT权限的用户只能读取其自身的组信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
MEMBER_NAME	同属于一组的成员名
GROUP_NAME	组名

SYSAUTHTABLE

SYSAUTHTABLE包含了赋予表的所有权限列表，以及授权的用户。拥有RESOURCE或CONNECT权限的用户可以读取被授予权限字段的权限信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有权限信息。

字段名	描述
TABLE_NAME	被授予权限的表名或视图名

字段名	描述
GRANTEE	授权表的用户名
TABLE_OWNER	表或视图的创建者
NUM_RPI_COLS	表或视图上被授权的字段数量
SEL_ALL	**1** — 在定义的表或视图上用户有选择所有字段数据的权限 **0** —用户没有选择数据权限
DEL_ALL	**1** — 在定义的表或视图上用户有删除所有字段数据的权限 **0** —用户没有删除数据权限
INS	**1** — 在定义的表或视图上用户有插入数据到特定字段的权限 **0** — 用户没有插入数据权限
INS_ALL	**1** — 在定义的表或视图上用户有将数据插入到所有字段的权限 **0** — 有户没有权限将数据插入到所有字段，但仍可以有对于个别字段操作的权限（参看INS）
UPD	**1** —在定义的表或视图上用户有更新特定字段数据的权限 **0** —在定义的表或视图上用户没有更新特定字段数据的权限
UPD_ALL	**1** — 在定义的表或视图上用户有更新所有字段数据的权限 **0** —有户没有权限更新所有字段数据，但仍可以有对于个别字段进行更新操作的权限（参看UPD）

字段名	描述
ALT_ALL	**1** — 用户有改变特定表或视图定义的权限 **0** — 用户没有改变特定表或视图定义的权限
IDX_ALL	**1** — 在特定表或视图上用户有创建或删除索引的权限 **0** — 在特定表或视图上用户没有创建或删除索引的权限
REF	**1** — 在特定表或视图上，用户有参考特定字段创建约束的权限 **0** —在特定表或视图上，用户没有在任何字段创建约束的权限
REF_ALL	**1** —在特定表或视图上，用户有参考多字段创建约束的权限 **0** —在特定表或视图上，用户没有参考到多字段创建约束的权限，但仍可以有参考到某个字段的权限（参看 REF）

SYSAUTHUSER

SYSAUTHUSER表列出数据库中有效用户的名称和权限级别。拥有RESOURCE或CONNECT权限的用户只能读取其自身的信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
USER_NAME	数据库中每个有效用户的ID，当用户被授予CONNECT权限时，就认为此用户为有效用户
DBA	**0** — 用户无DBA权限 **1** — 用户具有DBA权限 **2** — 用户具有SYSDBA权限

字段名	描述
RESOURCE	**0** — 用户无 resource 权限 **1** — 用户有 resource 权限
ACLORDER	**0** — 基于白名单 **1** — 基于黑名单

SYSCMDINFO

SYSCMDINFO表记录存储命令信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
MODULENAME	模块名称
CMDNAME	命令名称
CMDOWNER	命令创建者
STATEMENT	命令语句
NUM_PARM	参数数量
STATUS	**0** — 无效 **1** — 有效
REBTIME	存储命令重建的时间
CMDPLAN	释放存储命令执行计划字符串

SYSCOLUMN

这个表列出每个表和视图的字段，包括系统表的字段。拥有RESOURCE或CONNECT权限的用户可读取自己创建以及被授予权限的信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。在SCALE和

RADIX字段中，如果某个字段的数据类型不用定义SCALE和RADIX，则此两字段会返回-1值。

字段名	描述
COLUMN_NAME	字段名
TABLE_NAME	拥有此字段的表名
TABLE_OWNER	创建表的用户名
COLUMN_ORDER	表中此字段的序字段数
NULLABLE	**0** —此字段不允许空值 **1** —此字段允许空值
TYPE_NAME	字段的类型参考如下：BINARY、CHAR、NCHAR、DATE、DECIMAL、DOUBLE、FILE、FLOAT、INTEGER、LONG VARCHAR、NCLOB、LONG VARBINARY、SERIAL、SMALLINT、TIME、TIMESTAMP、VARCHAR、NVARCHAR
PRECISION	字段的精确度
SCALE	字段的大小范围
RADIX	字段的基数
ASCII_DEF	字段的ASCII形式的默认值
CONSTRA	字段的约束
REMARKS	字段的描述

SYSCONFIG

SYSCONFIG记录服务器配置设定。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无查询权限，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
KEYWORD	用于dmconfig.ini.中的关键字
VALUE	目前设置值

SYSCONINFO

SYSCONINFO记录数据库连接信息。拥有RESOURCE或CONNECT权限的用户只能读取自身的连接信息，请注意拥有DBA、SYSDBA或SYSADM权限的用户也只能读取自身的连接信息。

字段名	描述
CONNECTION_ID	连接ID
INFO1	保留
LAST_SERIAL	在更新字段时的最新序字段数，这是一个SERIAL类型。
LAST_OID	最新插入记录的对象ID（OID）。
INFO2	保留
INFO3	保留

SYSDBLINK

SYSDBLINK表包含的主要是远程数据库链接信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建的信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
OWNER	链接拥有者
DB_LINK	链接名
DBSVR	包含远程数据库信息的数据库部分

字段名	描述
USER_NAME	远程数据库用户名

SYSDESCOL

SYSDESCOL表包含动态字段的信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建的信息，并可以从其他用户获取权限，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
COLUMN_NAME	动态字段名
TABLE_NAME	拥有JSONCOLS类型的表的名称
TABLE_OWNER	创建表的用户名称
DATA_TYPE	动态字段的类型
COLUMN_NUM	动态字段数
LENGTH	动态字段的长度

SYSDOMAIN

SYSDOMAIN包含在数据库中创建定义域的信息。拥有RESOURCE、CONNECTDBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
DOMAIN_NAME	定义域名称
DOMAIN_OWNER	创建定义域的用户名称
ASCII_DEF	定义域中 ASCII 形式的默认值

字段名	描述
TYPE_NAME	字段的类型是下面的任一种: BINARY、CHAR、NCHAR、DATE、DECIMAL、DOUBLE、FILE、FLOAT、INTEGER、LONG VARCHAR、NCLOB、LONG VARBINARY、SERIAL、SMALLINT、TIME、TIMESTAMP、VARCHAR、NVARCHAR
DATA_LEN	定义域的数据类型大小
PRECISION	定义域精确度
SCALE	定义域范围
CONSTR	定义域约束
TEXT_CONVERTER	将 CLOB、NCLOB、BLOB或者FILE类型的数据转换为纯文本，以便创建全文索引和PURETEXT() UDF

SYSFILE

SYSFILE主要包含数据库中的文件信息。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
FILE_NAME	逻辑文件名

字段名	描述
FILE_TYPE	文件类型： **1 —** 数据文件 **2—** BLOB 文件
FILE_OID	文件 OID
TS_NAME	文件所在的表空间名称
FILE_NPAGES	文件中页的数量。如果表空间为自动扩展表空间，那么文件的 FILE_NPAGES 可能少于实际的 FILE_NPAGES
RAWDEV_OFFSET	DBMaster在此版本中不支持
CREATE_TIME	文件创建时间

SYSFILEOBJ

SYSFILEOBJ表主要记录的是数据库的文件对象信息。主要包括系统及用户文件对象。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
FILE_TYPE	**00 —** 系统文件对象 **01 —** 用户文件对象
SHARE	共享文件对象的记录数
FILE_NAME	文件对象所存放的全路径名

SYSFOREIGNKEY

SYSFOREIGNKEY 表记录数据库中的所有外键信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的外键信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
FK_TBL_NAME	子表名称（外键所在的表）
PK_TBL_NAME	外键参考的父表名称
FK_TBL_OWNER	子表所属用户
PK_TBL_OWNER	父表所属用户
FK_NAME	外键名称
UPD_ACT	更新参考动作： **0** — 无动作 **1** —设置为NULL **2** — 级联 **3** — 设置默认值
DEL_ACT	删除参考动作： **0** — 无动作 **1** — 设置为NULL **2** — 级联 **3** — 设置默认值

SYSGLBTRANX

SYSGLBTRANX记录的是全局事务信息。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
STATE	全局事务状态 **0**（ISSUE）— 事务分支已产生，但参与者还尚未准备好。 **1**（PREPARE ）— 参与者已准备好，但需等待父级参与者决定是否进行事务提交或终止。 **2**（COMMIT）— 参与者已决定提交全局事务。 **3**（PEND_TO_COMMIT）— 在灾难恢复后，事务分支将会被添加到提交队列中并等待提交。 **4**（PEND_TO_ABORT）— 在灾难恢复后，事务分支将会被添加到终止队列中并处于未决终止状态。
PARTICIPANT	全局事务参与者
GLBTRANXID	全局事务ID

SYSINDEX

SYSINDEX表主要记录的是数据库中索引信息。当NUM_PAGE、NUM_LEVEL、NUM_LEAF、DIST_KEY、NUM_PAGE_KEY或CLSTR_COUNT字段为-1时，代表这些值是无用的。拥有RESOURCE或CONNECT权限的用户可读取自己创建以及被授予权限的索引信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
INDEX_NAME	索引名
TABLE_NAME	索引所在的表名称
TS_NAME	索引所处的表空间名称

字段名	描述
TABLE_OWNER	索引所在表的用户名
UNIQUE	索引唯一性状态标志 **0** — 不唯一 **1** — 唯一 **3** — 主键 **4** — 自动索引
NUM_COL	索引字段数
INDEX_OID	索引OID
NUM_PAGE	索引页数
NUM_LEVEL	级别数
NUM_LEAF	叶子页数
DIST_KEY	不同键的数量
NUM_PAGE_KEY	每个键的页数
CLSTR_COUNT	聚集数，当用索引访问数据页时的I/O页数，这与缓冲区数量有关。
CREATE_TIME	索引创建时间

SYSINDEXREF

SYSINDEXREF表包含数据库中索引和自动索引的信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
INDEX_NAME	索引名
TABLE_NAME	索引所在的表名称
TABLE_OWNER	索引所在表的用户名

字段名	描述
UNIQUE	索引唯一性状态标志： **0 –** 不唯一 **1 –** 唯一 **3 –** 主键 **4 –** 自动索引
TOTAL_COUNT	使用的自动索引的总数
LASTREF_TIME	自动索引的最新参考时间

SYSINFO

SYSINFO表主要记录数据库当前状态信息。拥有RESOURCE或CONNECT权限的用户可以读取下列表中的信息：

SYSINFO.ID	SYSINFO.INFO	描述
0108	DB_PAGE_SIZE	数据库页大小
0711	VERSION	版本信息
0712	FILE_VERSION	文件版本

拥有DBA、SYSDBA或SYSADM权限的用户可读取所有信息。

SYSINFO表结构与其它系统表不同，系统表结构如下：

```
SYSTEM.SYSINFO (char(4) ID, varchar(32) INFO, varchar(32) VALUE);
```

每个字段的含义如下：

- **ID**：项目标识符。根据ID号来对系统信息编目录。前两个字符代表类型，接下来两个字符代表类型中包含的项目。例如：对于ID号'0105'代表的是NUM_LOGICAL_READ，'01'表示它属于页和I/O类型，'05'代表的是页和I/O类型中的序列。用户通过ID可对SYSINFO进行排序或过滤。

- **INFO**：系统信息项目名称。例如：名称 'NUM_LOGICAL_READ'代表逻辑磁盘读取数。
- **VALUE**：所有系统信息的返回值都为VARCHAR型数据。

➲ **示例**

下面的语句显示逻辑磁盘读取数：

```
dmSQL> SELECT INFO, VALUE from SYSTEM.SYSINFO WHERE INFO = 'NUM_LOGICAL_READ';
  ID            INFO                                VALUE
======   =================        ======================================
0105     NUM_LOGICAL_READ          338

1 rows selected
```

下一页将会列出所有SYSINFO的项目。

页和I/O信息：

SYSINFO.ID	SYSINFO.INFO	描述
0101	NUM_IDX_SPLIT	分隔索引页数
0102	NUM_PAGE_COMPRESS	数据页压缩数，即页重组
0103	NUM_PHYSICAL_READ	物理磁盘读取数，I/O单位为**页**
0104	NUM_PHYSICAL_WRITE	物理磁盘写的数量，I/O单位为**页**
0105	NUM_LOGICAL_READ	逻辑读的数量，I/O单位为**页**
0106	NUM_LOGICAL_WRITE	逻辑写的数量，I/O单位为**页**
0107	NUM_PAGE_BUF	页缓冲数，计算单位为**页**

SYSINFO.ID	SYSINFO.INFO	描述
0108	DB_PAGE_SIZE	数据库中的页大小，单位为 **字节**
0109	DB_SCA_SIZE	共享内存中系统控制区的大小，计算单位为**页**
0110	NUM_JOURNAL_BUF	共享内存中日志缓冲区的数量，计算单位为**块**
0111	DB_SYSCB_SIZE	共享内存中内部系统控制块的大小，计算单位为**字节**
0112	READ_HIT_RATIO	页缓冲区的读命中率
0113	WRITE_HIT_RATIO	页缓冲区的写命中率

注意 *1页的大小由关键字DB_PgSiz来指定，可以为4K、8K、16K或32K。*

日志信息：

SYSINFO.ID	SYSINFO.INFO	描述
0201	NUM_JNL_BLK_READ	从日志文件中读取的日志块数
0202	NUM_JNL_BLK_WRITE	写入日志文件中的日志块数
0203	NUM_JNL_REC_WRITE	产生日志记录的数目. 新的日志记录首先放在日志缓冲区中

SYSINFO.ID	SYSINFO.INFO	描述
0204	NUM_JNL_FRC_WRITE	强迫写日志数.这个数为将已写的日志缓冲区刷新到磁盘的I/O数
0205	NUM_JOURNAL_FILE	日志文件数
0206	NUM_JOURNAL_BLOCKS	一个文件的日志块数，数据库中的所有日志块数为：NUM_JOURNAL_FILE* NUM_JOURNAL_BLOCKS
0207	NUM_JNR_BLOCK_FREE	空闲日志块数
0208	CURRENT_JOURNAL_FN	当前使用的日志文件号
0209	CURRENT_JOURNAL_BN	日志文件的当前块号.每个日志文件中的日志块都有唯一的地址，它由CURRENT_JOURNAL_FN 和CURRENT_JOURNAL_BN组成日志文件的块号从0开始计数
0210	JOURNAL_FLUSH_RATE	DmServer平均每次写入的日志块数在整个日志缓冲区中所占的比例

注意 *1块 = 512字节。*

事务信息：

SYSINFO.ID	SYSINFO.INFO	描述
0301	NUM_STARTED_TRANX	启动事务的数量
0302	NUM_COMMITED_TRANX	提交事务的数量
0303	NUM_ABORTED_TRANX	终止事务的数量
0304	NUM_CHECKPOINT	检查点数量
0305	NUM_COMMIT_WAITER	等待整组提交的事务数量

锁信息：

SYSINFO.ID	SYSINFO.INFO	描述
0401	NUM_ROW_LOCK_UPG	页锁升级数（即行锁升级为页锁）
0402	NUM_PAGE_LOCK_UPG	表锁升级数（即页锁升级到表锁）
0403	NUM_LOCK_TIMEOUT	由于超时而引起的锁失败数
0404	NUM_LOCK_WAIT	等候的锁数量
0405	NUM_LOCK_REQUEST	请求的锁数量
0406	NUM_DEADLOCK	检测的死锁数量

连接信息：

SYSINFO.ID	SYSINFO.INFO	描述
0501	NUM_MAX_HARD_CONNECT	数据库允许的最大连接数（硬连接限制，

SYSINFO.ID	SYSINFO.INFO	描述
		即DB_MaxCo数据库以新日志模式启动或创建一个新数据库时）
0502	NUM_MAX_SOFT_CONNECT	在某一时刻允许的最大连接数（软连接限制，即**DB_MaxCo**数据库以正常模式启动）。 软连接限制数小于或等于硬连接限制数。（*DBMaster以前版本中，此关键字为NUM_MAX_TRANX*）
0503	NUM_CONNECT	当前激活的连接数。（*DBMaster的以前版本中，此关键字为NUM_ACT_TRANX*）
0504	NUM_PEAK_CONNECT	某一时刻激活的最大连接数（激活连接数的峰值）。

数据操作信息：

SYSINFO.ID	SYSINFO.INFO	描述
0601	NUM_SQL_SELECT	SELECT操作数
0602	NUM_SQL_INSERT	INSERT（包括INSERT INTO）操作数
0603	NUM_SQL_UPDATE	UPDATE操作数

0604	NUM_SQL_DELETE	DELETE 操作数
0605	NUM_SQL_PREPARE	调用服务器的 SQLPrepare()数
0606	NUM_SQL_EXECUTE	调用服务器的 SQLExecute()数
0607	NUM_SQL_EXEDIRECT	调用服务器的 SQLExecDirect()数
0608	NUM_SQL_FETCH	通过网络获取数据的数量

数据库信息：

SYSINFO.ID	SYSINFO.INFO	描述
0701	SYSINFO_RESET_TIME	SYSINFO的计数器，重启时间（新） 用于记录SYSINFO的重启时间。进行如下操作时设置生效： 1. dmSQL> set SYSINFO clear; 2. 一个计数器溢出，那么复位所有计数器。当进行如下操作时会进行检查： 2-1. 每次执行查询SYSINFO表。 2-2. 大约每5秒执行一次I/O服务。
0702	DCCA_SIZE	整个DCCA大小，单位：字节。
0703	FREE_DCCA_SIZE	可用的DCCA大小，单位：字节。

SYSINFO.ID	SYSINFO.INFO	描述
0704	DDB_MODE	分布式数据库模式： **ON**：允许 **OFF**：不允许
0705	BACKUP_MODE	备份模式： **NON-BACKUP**：无备份模式（**DB_BMode = 0**） **BACKUP-DATA**：只备份普通数据（**DB_BMode = 1**） **BACKUP-DATA-AND-BLOB**：备份普通数据和BLOB数据（**DB_BMode = 2**）
0706	USER_FO_MODE	用户文件对象模式： **ON**：允许 **OFF**：不允许
0707	READ_ONLY_MODE	只读模式： **ON**：允许 **OFF**：不允许
0708	FRAME_SIZE	BLOB帧单位大小。单位：**字节**。
0709	CREATE_DB_TIME	数据库创建时间
0710	START_DB_TIME	数据库启动时间
0711	VERSION	DBMaster版本
0712	FILE_VERSION	数据库文件版本

SYSINFO.ID	SYSINFO.INFO	描述
0713	FORCE_NEW_JNL_TIME	数据库以新日志方式启动时间
0714	START_NO_JNL_TIME	关闭日志模式时间
0715	END_NO_JNL_TIME	开启日志模式时间
0716	MAX_ITT_SIZE	内部临时表可使用的最大内存，单位：字节
0717	CURRENT_ITT_SIZE	内部临时表当前占用内存，单位：字节
0718	FULL_BACKUP_COST	最近一次完整备份所用时间
0719	DIFF_BACKUP_COST	最近一次差异备份所用时间
0720	DIFF_BACKUP_PCT	（最近一次差异备份大小）/(数据库大小)*100%

系统信息：

SYSINFO.ID	SYSINFO.INFO	描述
0801	CPU_USAGE	在一个短时期内平均CPU负荷数（大约5秒)（新）（**0 ~ 100%**） 支持平台： Solaris，LINUX，Windows 2000（仅计算第一个CPU，并且需要pdh.dll库）。 注意：支持此项必须启动I/O 服务

SYSINFO.ID	SYSINFO.INFO	描述
0802	TOTAL_MEMORY	整个物理内存大小。单位：**字节**。 支持平台：Solaris、LINUX、Windows，支持POSIX标准的UNIX。
0803	TOTAL_FREE_MEMORY	当前空闲物理内存大小（新）。 单位：**字节** 支持平台：Solaris、LINUX、Windows，支持POSIX标准的UNIX。
0804	TOTAL_SWAP_SPACE	整个交换空间大小。单位：**字节** 支持平台：Solaris、LINUX、Windows
0805	TOTAL_FREE_SWAP_SPACE	当前空闲交换空间大小（新）。 单位：**字节** 支持平台： Solaris、LINUX、Windows

对于不支持的版本此值为NULL.SYSINFO表是累计计数器的集合，用户可通过两种方法来复位SYSINFO。

1. 用户执行“set SYSINFO clear”命令。
2. 当一个计数器溢出时，所有SYSINFO 中的计数器都会被复位。如下情况时DBMaster会有溢出检测：
   a) 每次查询SYSINFO表时。

b) 每5秒访问一次I/O 服务。

用户执行如下语句可获得复位时间：

```
dmSQL> SELECT VALUE FROM SYSTEM.SYSINFO WHERE INFO = 'SYSINFO_RESET_TIME';
```

SYSJARFILE

SYSJARFILE包含java的jar包信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的表或视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
JAR_NAME	JAR名
JAR_OWNER	JAR拥有者
JARFILE_NAME	JAR文件名

SYSJAVAARGU

SYSJAVAARGU表包含java参数信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的表或视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
PROC _NAME	过程名
PROC _OWNER	过程拥有者
DATATYPE_NAME	数据类型名
ORDER	Java参数次序
DATATYPE	数据类型

SYSLOCK

SYSLOCK表包含在数据库中对象的锁状态信息。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

注意 *锁的级别为SYSTEM、TABLE、PAGE或TUPLE，锁的状态有GRANTED、WAITING或CONVERT，锁的模式为NONE、IS、S、IX、SIX或X。*

字段名	描述
LK_OBJECT_ID	锁对象的OID
TABLE_ID	包含锁对象的表的OID
LK_GRAN	锁级别：SYSTEM、TABLE、PAGE或TUPLE
HOLD_LK_CONNECTION	对象上获取锁的连接ID
LK_STATUS	锁状态：GRANTED、WAITING或CONVERT
LK_CURRENT_MODE	对象的当前锁模式
LK_NEW_MODE	对象的新锁模式

SYSOPENLINK

SYSOPENLINK表主要包含的是开放式数据库链接信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建的信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
DB_LINK	开放式数据库链接
DBSVR	服务器

字段名	描述
USER_NAME	用户名
TXN_STATUS	事务状态： **'R'** — 读 **'W'** — 写 **'N'** — 无事务处理

SYSPENDTRANX

SYSPENDTRANX主要包含在分布式数据库环境中未提交事务的信息。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
XIDFORMAT	Format ID表示全局协调者的类型，DBMaster是22873。
PREPAREDTIME	准备提交事务的时间
GLBTRANXID	全局事务ID

SYSPROCINFO

SYSPROCINFO记录的是存储过程信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

对于本地临时过程，用户仅能查询到当前连接中创建的过程。

对于全局临时过程，所有用户（包含不同连接）可在该过程的生命周期内查询过程信息（直到该过程被删除）。

字段名	描述
QUALIFIER	限定
PROC_OWNER	存储过程拥有者
NAME	存储过程名
NUM_INPUT_PARAMS	输入参数数量
NUM_OUTPUT_PARAMS	输出参数数量
NUM_RESULT_SETS	结果集数目
REMARKS	备注
PROC_TYPE	存储过程类型： **1**（SQL_PT_PROCEDURE）— 过程 **2**（SQL_PT_FUNCTION）— 函数
TEMP_TYPE	临时表空间类型： **1** —固定 **2** —全局 **3** —局部
SID	连接ID

SYSPROCJAVA

表SYSPROCJAVA包含java过程信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的表或视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
PROC _NAME	过程名
PROC _OWNER	过程拥有者

CLASS_ID	类ID
NUM_ARGU	参数个数
NUM_RELATED_JAR FILE	相关JAR文件个数
JAR_NAME	Java过程名
JAR_OWNER	Java过程拥有者

SYSPROCPARAM

SYSPROCPARAM主要包含存储过程参数信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建以及被授予权限的过程参数信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

对于本地临时过程，用户仅能查询到当前连接中创建的过程参数信息。

对于全局临时过程，所有用户（包含不同连接）可在该过程的生命周期内查询过程参数信息（直到该过程被删除）。

字段名	描述
QUALIFIER	限定
OWNER	存储过程拥有者
PROC_NAME	存储过程名
PARAM_NAME	参数名
PARAM_TYPE	参数类型： **1**（SQL_PARAM_INPUT）—输入 **3**（SQL_PARAM_OUTPUT）—输出 **4**（SQL_RETURN_VALUE）— 返回值 **5**（SQL_RESULT_COL）—结果集
DATA_TYPE	数据类型
TYPE_NAME	类型名

字段名	描述
PRECISION	精确度
LENGTH	长度
SCALE	数值范围
RADIX	基数
NULLABLE	可空的字段： **1** — 允许空值 **0** — 不允许空值
REMARKS	备注

SYSPROJECT

SYSPROJECT表记录ESQL工程上的信息。拥有RESOURCE或CONNECT权限的用户可以读取自己创建的过程ESQL工程信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
PROJECT_NAME	工程名
PROJECT_OWNER	工程拥有者
MODULE_NAME	模块名
MODULE_OWNER	模块拥有者
MODULE_SOURCE	模块源
REF_CMD	内部使用

SYSPUBLISH

SYSPUBLISH包含所有表复制源端的信息。拥有RESOURCE或CONNECT权限的用户可读取自己创建表的复制信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
REPLICATION_NAME	复制名
TYPE	**S —** 同步 **A —**异步
TABLE_OWNER	被复制表的拥有者
TABLE_NAME	被复制的表名称
NUM_PROJECT	投影字段数
FRAGMENT	字符串片段
NUM_SUBSCRIBER	订阅者（目的端）数量

SYSSCHEDULE

SYSSCHEDULE表包含计划任务的信息。数据库创建时，系统会自动在SYSSCHEDULE中添加一条与计划**schelogcl**相关的记录，用来在每天早上1:10运行任务**tasklogcl**。该计划默认为禁用。只有拥有DBA或更高权限的用户可以通过SCHEDULE_ENABLE启用计划或通过SCHEDULE_DISABLE禁用计划。

字段名	描述
SCHEDULE_OWNER	计划的所有者
SCHEDULE_NAME	计划名
TASK_NAME	使用计划调用的任务名
TIMETABLE	计划与前一个计划之间的时间间隔
START_TIME	开始执行计划的时间
END_TIME	结束计划的时间

字段名	描述
STATUS	计划状态： **0** — 禁用 **1** — 启用

SYSSCHELOG

SYSSCHELOG表包含计划任务的执行记录的信息。建议用户定期清除存储在SYSSCHELOG中的多余日志，并保留最近几天的日志。

字段名	描述
SCHEDULE_OWNER	计划的所有者
SCHEDULE_NAME	计划任务的名称
CONNECTION_ID	任务的连接ID
BEGIN_TIME	开始执行任务的时间
END_TIME	任务的结束时间
STATUS	任务的执行状态
ERROR_MSG	任务执行失败时返回的错误信息

SYSSCHEMA

SYSSCHEM包含模式名和模式所有者之间的关系信息。拥有RESOURCE或CONNECT权限的用户可读取用户模式信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
SCHEMA_NAME	模式名
SCHEMA_OWNER	模式拥有者

SYSSUBSCRIBE

SYSSUBSCRIBE表包含表复制的目的端信息。拥有RESOURCE或CONNECT权限的用户可以读取用户表及相关表的信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
BASE_TABLE_OWNER	基表拥有者
BASE_TABLE_NAME	基表名称
REPLICATION_NAME	复制名
DB_LINK	数据库链接
TABLE_OWNER	表的拥有者
TABLE_NAME	表名

SYSSYNONYM

SYSSYNONYM表包含数据库中同义字定义信息。拥有RESOURCE或CONNECT权限的用户可以读取表同义字信息或用户创建的视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
SNAME	同义字名
OWNER	同义字拥有者
TV_NAME	同义字的源表/视图名
TV_OWNER	表/视图拥有者
TV_LINK	在一个远程数据库中表或视图的链接名
TV_SERVER	远程数据库中表或视图的数据库名

SYSTABLE

SYSTABLE表包含数据库中所有表的信息。拥有RESOURCE和CONNECT权限的用户可以读取自己创建以及被授予权限的表或视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
TABLE_NAME	表名
TABLE_OWNER	表的拥有者
TABLE_TYPE	表类型：系统表、系统视图、表、视图。
LOCKMODE	为表提供的锁模式： **T** — 表锁 **P** — 页锁 **R** — 行锁 默认的锁模式为行锁
CACHEMODE	全表扫描的缓冲模式： **T** —有缓冲模式（真 ） **F** —无缓冲模式（假）
TS_NAME	表所在的表空间名称
TABLE_OID	表的OID
NUM_COL	表中字段的数目
NUM_INDEX	表中索引数目
NUM_PAGE	表中页的数目，默认值为-1。当用户更新统计值时，将返回NUM_PAGE的实际值。
NUM_FRAME	表中BLOB帧的数目

字段名	描述
NUM_ROW	表中的行数，默认值为 -1。当用户更新表中的统计值时，将返回NUM_ROW的实际值。
NUM_INDIRECT_ROW	间接行的数目
AVERAGE_LENGTH	表中对于数据的平均长度，默认值为 -1。当用户更新表中的统计值时，将返回AVERAGE_LENGTH的实际值。
CREATE_TIME	表创建的时间
UPD_STS_TIME	更新表统计值的最新时间
CONSTR	表约束
FILLFACTOR	**FILLFACTOR**指定了在禁止插入新数据之前，可填充的页百分比（允许更新），默认值为100(%)
SERIAL_COL_ID	Serial字段位于表中的n^{th}字段
SERIAL_START_NO	Serial字段开始数目，默认值为1
REMARKS	表的描述
NUM_TRIG	表中触发器数目
NUM_TEXTINDEX	表中索引数目
NUM_PUBLICATION	表中publications的数量
NUM_DEST	对于异步表复制中目的数据库的数目
UPD_STS_MODE	表更新统计模式
UPD_STS_SAMPLE	表更新统计样例率

SYSTABLESPACE

SYSTABLESPACE表包含数据库中所有表空间的信息。拥有RESOURCE权限的用户不能查看该表的任何信息，系统会产生错误代码

6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
TS_NAME	表空间名称
TS_TYPE	表空间类型： **0 (fixed(W)) —** 正常 **1 (ext(W)) —** 自动扩展 **2 (fixed(R)) —** 只读（正常） **3 (ext(R)) —** 只读（自动扩展）
NUM_FILES	表空间中文件数目
NUM_PAGES	表空间中页的数目。如果为自动扩展表空间，那么NUM_PAGES页可能少于表空间实际NUM_PAGES数目。
NUM_FREE_PAGES	表空间中可利用的空闲页数目
NUM_FRAMES	表空间中BLOB帧的数目
NUM_FREE_FRAMES	表空间中可利用的空闲BLOB帧数目

SYSTASK

SYSTASK表包含计划行动的信息。数据库创建时，系统会自动在SYSTASK中添加一条与任务**tasklogcl**相关的记录。该任务默认调用SCHELOG_CLEAN来清除比当前日志创建时间早10天的日志。只有拥有DBA或更高权限的用户可以更改要删除的日志创建时间与最新日志创建时间之间的天数，即可以更改reserve_day的值。

字段名	描述
TASK_OWNER	任务的所有者

字段名	描述
TASK_NAME	任务名
TASK_TYPE	任务类型
ACTIONS	任务的行动

SYSTEXTINDEX

SYSTEXTINDEX表包含全文索引信息。拥有RESOURCE和CONNECT权限的用户可以读取自己创建以及被授予权限的表的全文索引信息。拥有DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
TEXTINDEX_NAME	全文索引名称
TABLE_NAME	表名
TABLE_OWNER	表拥有者
INDEX_OID	索引的OID
TYPE	全文索引的两个不同类型：特征向量和反向全文索引
FACTOR1	内部管理数据
FACTOR2	内部管理数据
FACTOR3	内部管理数据
FACTOR4	内部管理数据
FACTOR5	内部管理数据
FACTOR6	内部管理数据
FACTOR7	内部管理数据
FACTOR8	内部管理数据
FACTOR9	内部管理数据

字段名	描述
FACTOR10	内部管理数据
CREATE_TIME	全文索引创建的时间
REBUILD_TIME	全文索引重建的时间

SYSTRIGGER

SYSTRIGGER表包含所有触发器信息。拥有RESOURCE和CONNECT权限的用户可以读取自己创建的表的触发器信息。拥有DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
TBNAME	表名
TBOWNER	表拥有者
TRIGNAME	触发器名
TRIGEVENT	触发器事件： **1** — Insert事件 **2** — Delete事件 **3** — Update事件触发器 **4** — Update字段事件
NUM_COL	字段数目
SCOL_NUM	对于触发器更新的最少字段数目
TRIGTYPE	触发器类型： **1** — BEFORE和FOR EACH STATEMENT **2** — BEFORE和FOR EACH ROW **4** — AFTER和FOR EACH STATEMENT **8** — AFTER和FOR EACH ROW

字段名	描述
STATUS	状态： **0** —不允许 **1** —允许
OLD	旧值
NEW	新值
MODE	**0** —无效的触发器 **1** —有效的触发器
TRIGDEF	触发器定义

SYSTRPDEST

SYSTRPDEST表主要包含异步表复制中使用的时间计划信息。拥有RESOURCE权限的用户需要获取计划信息用于异步表复制。拥有RESOURCE、CONNECT、DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
SVRNAME	远程数据库名
USER_NAME	远程数据库的用户账号
STATUS	远程数据库状态： **0** —正常 **1** —错误 **2** —重试 **3** —挂起
BEGTIME	复制开始时间
INTERVAL	复制时间间隔

SYSTRPJOB

SYSTRPJOB包含在异步表复制中使用的日志信息。拥有RESOURCE和CONNECT权限的用户不能查看该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
DESTINATION	数据被复制到的数据库
FN	一个事务日志记录的文件数
OFFSET	在事务日志记录中的偏移量

SYSTRPPOS

SYSTRPPOS表主要包含在使用异步表复制时的信息。拥有RESOURCE和CONNECT权限的用户不能查看该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以查看所有信息。

字段名	描述
POSARRAY	内部使用

SYSUSER

SYSUSER表包含当前连接到数据库的所有用户信息。拥有RESOURCE和CONNECT权限的用户只能读取用户自身的信息，拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。在断掉连接前，应在SYSUSER表中查询到想要断掉的连接ID。如果你的登陆主机名没有在网络上注册，则LOGIN_HOST为**anonymous**。如果您想要监测更新统计状态，则需要查询SYSUSER表的SQL_CMD字段。

字段名	描述
CONNECTION_ID	连接ID
USER_NAME	登录用户名
LOGIN_TIME	登录时间
LOGIN_IP_ADDR	登录IP地址
LOGIN_HOST	登录主机名
NUM_SCAN	**SELECT** 操作数
NUM_INSERT	**INSERT** 操作数
NUM_UPDATE	**UPDATE** 操作数
NUM_DELETE	**DELETE** 操作数
NUM_TRANX	事务进程数
NUM_JBYTE_PER_TRAN	每个事务的日志字节数
FIRST_W_JNR_FN	活动状态事务的第一个日志文件号
FIRST_W_JNR_BN	活动状态事务的第一个日志块号
NUM_BYTE_JNR_DATA	活动状态事务所用的整个日志字节数
NUM_J_BLOCK_DURATN	活动状态事务使用的第一个日志块号与一个最近使用日志块之间的跨度

字段名	描述
SQL_CMD	最近执行的SQL命令和命令状态。 命令状态如下所示： [PRE] - SQL命令预备 [EXEC] - 根据SQLExecute调用执行SQL命令 [EXDIR] - 根据SQLExecDirect调用执行SQL命令 [EXIT] - SQL命令完成准备、执行或获取操作
TIME_OF_SQL_CMD	最近SQL命令的执行时间

SYSUSERFUNC

SYSUSERFUNC 表主要包含用户自定义和内置函数的信息。用户执行UDF时需要SYSUSERFUNC表中的数据，因此拥有任何连接权限的用户均可读取所有信息。

字段名	描述
MODE	函数类型： **0** — 非内置函数 **1** — 内置函数
FILE_NAME	内置函数所在的文件名称
FUNC_NAME	内置函数名称
LANGUAGE_TYPE	UDF的类型： **C**— C UDF **LUA**— LUAUDF
RETURN_TYPE	内置函数返回的数据类型

字段名	描述
NUM_OF_PARAMETER	函数中包含的参数数量
PARAMETER	每个参数的数据类型，参数数量根据NUM_OF_PARAMETER的值来设置

SYSVIEWDATA

SYSVIEWDATA包含数据库中在表上建立的视图信息。拥有RESOURCE或CONNECT权限的用户可读取自己创建以及被授予权限的视图信息。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
VIEW_NAME	视图名
VIEW_OWNER	视图拥有者
STATUS	**0** — 无效视图 **1** — 有效视图
VIEW_DEFINITION	视图定义

SYSWAIT

SYSWAIT表记录在当前等候其它锁释放对象时，所处的状态信息。拥有RESOURCE或CONNECT权限的用户不能读取该表的任何信息，系统会产生错误代码6829表示用户无SELECT权限。拥有DBA、SYSDBA或SYSADM权限的用户可以读取所有信息。

字段名	描述
WAITING_CONNECTION	等待的连接ID
WAITED_CONNECTION	被等待的连接ID

22 系统限制

DBMaster对于数据库中对象的命名长度有一些限制，索引、表、内存缓冲的大小以及文件大小和并发事务处理数目。这些限制以及其它相关内容将会在接下来的部分进行总结。

22.1 命名限制

ANSI/ISO标准中，对于SQL语言的定义中说明数据库对象应赋予唯一名称，并且定义的每个数据库对象都应有名称。在SQL语句中用这些名称来确定语句作用的对象。

数据库请求对象：

- 表
- 字段
- 用户

DBMaster中除了用户名与密码，其它标准与ANSI/ISO一致，DBMaster中的命名限制为1到128个字符。在ANSI/ISO标准中，SQL数据库对象的字符最大限制为128字符，并且可以包含字母和数字。它们可能不包含空格或标点符号等字符。DBMaster也支持使用多数据库对象和扩展字符范围的命名。

- 索引
- 游标
- 表空间
- 主/外键

在DBMaster中，数据库名可包含1到128个文字和数字的字符或者其它一些标识符，这些字符可出现在数据名的任何位置。其它标识符（除密码设置外），可包含长度从1到128的字母数字混合的字符、中文双字节字符、空格、下滑线以及 $和#符号，这些符号可出现在数据库名的任何位置，包括第一位。如果使用空格，名称必须要带双引号(" ")， 并且结尾空格将会被忽略。密码设置基本也遵循这个规则，但最大长度限制为16个字符。

下表列出了DBMaster中各个数据库对象所支持的命名长度限制。

项目	最小值	最大值
数据库名称	1	128
表空间名称	1	128
表名称	1	128
字段名	1	128
索引名	1	128
游标名称	1	128
用户名称	1	128
用户密码	1[7]	16
文件路径和物理文件名	1	256[8]
逻辑文件名	1	128
SQL语句	N/A	2097152

表22-1 数据库对象名的最小/最大长度（字符单位）

[7] 如果用户没有设置密码，那么密码长度为空。

[8] 包含零字符。

22.2 存储限制

下表列出DBMaster中数据库对象的存储限制。当了解这些限制定义的最大值时，应注意到物理系统限制(系统内存或磁盘空间)以及操作系统限制(系统资源)或其它限制，这些系统限制将会对定义值的最大范围有所影响。除非有特别注明，否则DBMaster对应的所有平台限制基本相同。

项目	最小值	最大值
数据库大小	—	256PB
数据库中的文件数	1	32767
数据库中的表空间数	1	32767
表空间中的文件数	1	32767
表空间的表数	0	no limit[9]
数据文件中的页数	3	2^{31}-1
表中的字段数	1	2000（也取决于数据页大小）
表中的记录（行）大小	0	3968 (4KB page size) [10] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]

[9]当前表和索引的个数会受到操作系统限制

[10]包括头标题, 要想了解更多信息 请参看第6.9章

项目	最小值	最大值
表中的索引数	0	no limit[11]
索引中的字段数	1	32
索引中的键值长度	0	4000
索引中可用的字段ID	1	no limit[11]
系统临时文件数	1	8
日志文件数	1	8
日志文件大小	100pages	8 GB
投影字段数	1	同表中的最大字段数
GROUP BY字段数	1	同表中的最大字段数
ORDER BY字段数	1	同表中的最大字段数
ODBC绑定参数	0	同表中的最大字段数
SQL语句数	1	127
BLOB文件中的字节数	0	8TB
数据缓冲器中的页数	15	依据OS
日志缓冲器中的页数	16	依据OS
扫描的谓词操作符数	1	依据表达式，最大1022
表达式或谓词中的常数	0	64k
表达式或谓词中的主变量数	0	64k

表22-2 数据库对象的最小和最大长度（单位字节数)

[11]表中的所有字段都可应用于索引中。

22.3 处理限制

下面列出DBMaster数据库中的处理限制。

项目	最小值	最大值
当前数据库的并发事务 (连接)数	0	4800
CHAR或BINARY数据类型的长度	0	3968 (4KB page size)[4] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]
VARCHAR数据类型的长度	0	3968 (4KB page size)[4] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]
LONG VARCHAR或LONG VARBINARY数据类型的长度	0	8 TB
Select语句最多可选择的项目数	1	同表中的最大字段数

表22-3 处理过程的最小和最大长度（单位字节数)