达梦数据库

PROGRAMMER MANUAL

程序员手册

目录

第 1 章 概述

DM8 是达梦数据库有限公司推出的新一代高性能数据库产品。它具有开放的、可扩展的体系结构，易于使用的事务处理系统，以及低廉的维护成本，是达梦公司完全自主开发的产品。DM8 以 RDBMS 为核心，以 SQL 为标准，是一个能跨越多种软硬件平台、具有大型数据综合管理能力的、高效稳定的通用数据库管理系统。

数据库访问是数据库应用系统中非常重要的组成部分。DM 作为一个通用数据库管理系统，提供了多种数据库访问接口，包括 ODBC、JDBC、DPI、OLEDB 以及嵌入方式等。本书详细介绍了 DM 的各种访问接口，相应开发环境的配置，以及一些开发用例。

本书的主要读者是从事过数据库应用系统开发，并具有 SQL 使用基础的程序员。

开发一个应用系统，需要对其使用的数据库管理系统所具有的功能、性能、特性有较深入的了解。DM 作为一个通用的数据库管理系统，具有非常丰富的功能和特色。

1.1 功能特性

DM 除了具备一般 DBMS 所应具有的基本功能外，还特别具有以下特性：

1. 通用性；
2. 高性能；
3. 高安全性；
4. 高可靠、高可用性；
5. 易用性；
6. 海量数据存储和管理；
7. 全文索引；
8. 对存储模块的支持；
9. 对 WEB 应用的支持。

以下对这些特性做具体介绍。

1.1.1 通用性

DM 是大型通用数据库管理系统，其通用性主要表现在以下四个方面：

1. SQL 及接口的开发符合国际通用标准

符合 SQL92/SQL99、ODBC、JDBC、OLE DB、PHP、.NET Provider 等国际标准或行业标准，提供所有数据库标准接口；

支持 SQL92 标准的所有数据类型；

SQL92 入门级标准符合率达到 100%，过渡级也达到 100%；

提供了符合 ODBC 3.0 标准的 ODBC 接口驱动程序、符合 JDBC 3.0 标准的 JDBC 接口驱动程序和符合 OLE DB 2.7 标准的 OLE DB 接口驱动程序。

支持 eclipse、JBuilder、Visual Studio、Delphi、C++Builder、PowerBuilder 等各种流行的数据库应用开发工具。

此外，为了提高数据库的通用性，DM 还增加了一些其他数据库的数据类型、函数和语

法等特性，与国外数据库管理系统（如 Oracle、SQL Server 等）高度兼容，如：同时支持自增列和序列。而且从数据库市场现状和技术人员开发习惯的角度出发，注重在功能扩展、函数配备、调用接口及调用方式等方面尽量与国际主流的各类数据库产品接轨，提高应用系统的可移植性和可重用性，降低开发厂商移植和升级的工作难度和强度。

2. 跨平台支持

DM 服务器内核采用一套源代码实现了对不同软件（WINDOWS/LINUX/UNIX/AIX/SOLARIS 等）、硬件（X64/X86/SPARC/POWER/TITAM）平台的支持，各种平台上的数据存储结构完全一致。与此同时，各平台的消息通信结构也完全保持一致，使得达梦数据库的各种组件均可以跨不同的软、硬件平台与数据库服务器进行交互。

DM 支持 WINDOWS 2000/XP/2003、2.4 及 2.4 以上内核的 LINUX（Redhat、Debian、Suse、红旗、中标等）、麒麟操作系统（Kylin）、AIX、SOLARIS 等国内外常用操作系统。

DM 的管理工具、应用开发工具集使用 Java 编写，从而可以跨平台工作，即同一程序无需重新编译，将其执行码拷贝到任一种操作系统平台上都能直接运行。这也保证了它们在各种操作系统平台上都有统一的界面风格。

DM 产品采用一致的人机交互界面，既有易于操作使用的图形化界面工具，也有丰富的命令行控制台工具。

3. 支持对称多处理机系统

由于 DM 核心系统的多线程机制利用了操作系统的线程调度，因此系统的工作线程在单 CPU 和多 CPU 机器上都能很好地并行操作。对于多 CPU 的系统，只要采用的操作系统支持多 CPU 机制，DM 就能很好地实现多 CPU 协同工作。

4. 对 UNICODE 的支持

目前 DM 系统支持了 Unicode 字符集和其他多种字符集。用户可以在安装 DM 系统时，指定服务器端使用 UTF8 字符集。此时在客户端，用户能够以各种字符集存储文本，并使用系统提供的接口设置客户端使用的字符集，缺省使用客户端操作系统缺省的字符集。客户端和服务器端的字符集由用户指定后，所有字符集都可以透明地使用，系统负责不同字符集之间的自动转换。

对 Unicode 的支持使 DM 系统适应国际化的需要，增强了 DM 的通用性。

1.1.2 高性能

DM 主要通过以下机制实现了系统的高性能：

1. 可配置的多工作线程处理功能

DM 允许用户配置工作线程的数目。工作线程是整个系统所公用的资源，不专门为某个特定的连接服务。如果某个数据库操作由于无法取得相应的资源（如锁）而不能继续，将暂停当前的数据库事务，相应的工作线程会立即执行其它的数据库请求服务。所以，在系统硬件及操作系统性能够满足要求的情况下，连接数和任务请求数的增加对 DM 性能的影响是线性的。DM 自动协调工作线程对内存、数据页等物理资源的共享。

2. 高效的并发控制机制

DM 提供了数据库的行级和表级的资源封锁机制，大大提高了事务并发访问的数目。DM 通过以下几点确保并发控制的高效：

1) 用 HASH 表管理实现锁管理，能够在系统成千上万的锁中迅速地找到所需的锁；

2) 利用回滚段保存记录修改前的数据，实现数据的读一致性。

另外，在 DM 中，当对数据页进行操作时，系统会自动对数据页采用合适的封锁机制。同时，DM 也提供了函数供用户自行定义锁定类型，以增强系统并发度，提高系统效率。

3. 有效的查询优化策略

DM 采用有效的基于代价的查询优化策略，其查询优化子系统能计算最优的查询路径以保证查询的执行效率。查询优化主要通过以下三个步骤进行：

1) SQL 转换：DM 首先对用户输入的查询语句进行一系列复杂的转换，其结果为一个语义上等价但处理起来更为有效的 SQL 语句；
2) 统计信息与代价估计：DM 为数据库对象保存了一系列的统计信息，代价估计模块基于系统的 I/O、CPU 和内存等资源情况和数据库对象的统计信息估算每个计划的代价；
3) 执行计划选择：执行计划描述了查询语句的每一个处理步骤，如以什么算法执行连接，是否使用索引等。优化器考虑可能的执行计划，并选择代价最小的交付执行。

另外，用户可通过 DM 的客户端工具查看查询语句的执行计划。

4. 尽可能少的网络通讯量

DM 对消息发送条件进行仔细判断，避免和减少无用的网络交互，提高了消息处理的效率，减轻了服务器的负担，降低了等待时间，加速了工作线程的运转，提高了性能。对于密集型联机事务处理效果尤佳。

5. 加强的缓冲机制

DM 为了提高系统运行效率，对于频繁操作的对象进行了必要的缓存处理，实现了数据字典高速缓冲，实现了语法分析树的可重用，优化了存储过程和触发器的运行。

与此同时，DM 还新增了动态缓冲区机制。当数据库服务器发现因为缓冲区不足而产生频繁的缓冲区页面淘汰时，如果整个系统还有可用内存，则 DM 会自动扩展缓冲区，减少淘汰几率，显著提升系统性能。

6. 针对 64 位计算的优化策略和技术

DM 在代码级全面支持 64 位系统，能够支持主流 64 位处理机和操作系统，并融入了很多针对 64 位计算的优化策略和技术。DM 不仅能够运行在 64 位系统上，还能很好地利用 64 位系统的资源（例如能充分地利用更大容量的内存），在 64 位系统上表现出良好的性能。

7. 查询计划重用

DM 在首次执行 SQL 语句过程中会在内存中创建一个计划缓冲池，缓存执行过的 SQL 语句、存储过程以及触发器的执行计划。对于后续接收到的 SQL 命令，系统自动采用字符串全字匹配的方式，在计划池中查找对应的计划，如果有则直接使用，否则对该语句进行分析，创建执行计划，并将执行计划放入计划缓冲池中。通过这种方式，系统可以极大地节省对语句进行分析和创建执行计划的时间，从而提升系统性能。

8. 查询结果集缓存

DM 还支持查询结果的缓存功能。查询语句第一次执行后，系统可以把结果保存下来，如果后面有同样的语句，且所访问的数据没有发生变化，则不需要进行计算，直接把先前保存的结果返回给客户即可。在很多以查询为主的系统中，采用该功能性能可以得到成倍地提升。

9. 数据压缩

DM 新增了数据压缩的功能，用户可以对整个表进行压缩，也可以选择对部分字段进行

压缩。对于 I/O 密集型的系统来说，通过采取适当的数据压缩的策略，可以减少系统的 I/O 量，在 I/O 子系统是整个系统的瓶颈时，采取该措施可以有效提升系统性能。

10. 函数索引

DM 扩充了索引的建立方法，允许在表达式上面建立索引。表达式既可以出现在函数的参数中（如 ABS(ID)），也可以是列的操作（如 C1+C2），DM 把这类索引称为函数索引。函数索引的引入为 DBA 提供了一种新的优化手段。

例如，执行如下的语句：

```
SELECT * FROM t WHERE UPPER(name) = 'CBDG';
```

在以往的版本中，这条语句的执行只能是全表扫描。如果建立如下索引：

```
CREATE INDEX i_t_fbi_name ON t(UPPER(name));
```

则可以通过函数所以 i_t_fbi_name，进行等值扫描，性能获得大幅提升。

1.1.3 高安全性

只具备自主访问控制安全机制的数据库远不能满足一些对安全具有高要求的系统的需要。为了保证系统的安全性，DM 采用基于角色与权限的管理方法来实现基本安全功能，并根据三权分立的安全机制，将审计和数据库管理分别处理，同时增加了强制访问控制功能，另外，还实现了包括通讯加密、存储加密以及资源限制等辅助安全功能，使得达梦数据库安全级别达到 B1 级。

1. 完全自主知识产权

达梦数据库是具有完全自主知识产权的国产大型数据库管理系统。在产品开发过程中，达梦公司始终坚持自主开发的原则，致力于保卫国家信息安全，推进国民经济信息化建设，拥有产品的全部源代码和完全的自主版权。这一方面杜绝了继承开源系统导致的版权纠纷，同时也从根本上保证了系统的安全性，并有利于与其它应用系统集成，可以根据具体需求定制和提供及时有效的服务。

2. 三权分立的安全机制

DM 在安全管理方面采用了三权分立的安全管理体制，把系统管理员分为数据库管理员 DBA、数据库安全管理员 SSO、数据库审计员 AUDITOR 三类。DBA 负责自主访问控制及系统维护与管理方面的工作，SSO 负责强制访问控制，AUDITOR 负责系统的审计。这种管理体制真正做到三权分立，各行其责，相互制约，更好地实现了数据的保密性、完整性和可用性。

3. 身份验证

DM 能够根据用户在系统中的登录名和密码确定该用户是否具有登录的权限和其在系统中的系统级角色，确定该用户能够做什么和不能够做什么。DM 提供两种身份验证模式来保护对服务器访问的安全，即数据库身份验证模式和外部身份验证模式。数据库身份验证模式需要利用数据库口令，外部身份验证模式支持基于操作系统（OS）的身份验证。

4. 资源限制

资源限制是控制用户对 DM 服务器系统资源的使用情况，以尽可能减少人为的安全隐患。通过资源限制可以提供一个规划数据库系统资源使用的接口，可以人为地规划数据库资源的分配。这样做对于恶意地抢占资源的访问可以起到有效的遏制，保证普通数据库应用的正常进行，同时资源限制还起到保证身份验证可靠性的作用。

5. 自主访问控制

DM 系统根据用户的权限执行自主访问控制。用户权限是指用户在数据对象上被允许执行的操作。规定用户权限要考虑三个因素：用户、数据对象和操作，即什么用户在哪些数据对象上可执行什么操作。所有的用户权限都要记录在系统表（数据字典）中，对用户存取权限的定义称为授权，当用户提出操作请求时，DM 根据授权情况进行检查，以决定是执行操作还是拒绝执行，从而保证用户能够存取他有权存取的数据，不能存取他无权存取的数据。

6. 基于标记的强制访问控制

DM 利用策略和标记来实现数据库的强制访问机制。强制访问控制主要是针对用户和元组，用户操作元组时，不仅要满足自主访问控制的权限要求，还要满足用户和元组之间标记的相容性。这样，就避免了出现管理权限全部由数据库管理员一人负责的局面，同时也相应地增强了系统的安全性。

7. 数据库审计

审计机制是 DM 数据库管理系统安全管理的重要组成部分之一。DM 具有一个灵活的审计子系统，可以通过它来记录系统级事件、个别用户的行为以及对数据库对象的访问。通过查看审计信息，数据库审计员可以知道用户访问的形式以及试图对该系统进行的操作。一旦出现问题，数据库审计员可分析审计信息，跟踪审计事件，查出原因。

8. 通信加密

DM 提供两种通信方式，即不加密、可选算法的加密通信。选择何种通信方式以服务器端为准，通过设置服务器端配置文件中相应选项来指定，客户端以服务器采用的通信方式与其进行通信。

9. 存储加密

某些信息具有保密要求，实现存储加密的重要性不言而喻。DM 实现了对存储数据的加密，另外，还提供了内置的数据加/解密函数，为用户的隐私数据提供更加可靠的保护。

10. 客体重用

DM 内置的客体重用机制使数据库管理系统能够清扫被重分配的系统资源，以保证数据信息不会因为资源的动态分配而泄露给未授权的用户。

1.1.4 高可靠、高可用性

任何一个系统都存在发生各种意外故障的可能性。DM 的高可靠、高可用性可以避免或降低系统的意外故障对用户带来的损失，主要包括以下几个方面的功能：

1. 数据守护

DM 数据守护（Data Watch）是一种集成化的高可用、高性能数据库解决方案，是数据库异地容灾的首选方案。通过部署 DM 数据守护，可以在硬件故障（如磁盘损坏）、自然灾害（地震、火灾）等极端情况下，避免数据损坏、丢失，保障数据安全，并且可以快速恢复数据库服务，满足用户不间断提供数据库服务的要求。

DM 数据守护提供多种解决方案，可以配置成实时主备、MPP 主备、或读写分离集群，满足用户关于系统可用性、数据安全性、性能等方面的综合需求。

2. 共享存储集群

DM 共享存储数据库集群（DM Data Shared Cluster，简称 DMDSC），允许多个数

据库实例同时访问、操作同一数据库，具有高可用、高性能、负载均衡等特性。DMRAC 支持故障自动切换和故障自动重加入，某一个数据库实例故障后，不会导致数据库服务无法提供。

3. 高级复制

达梦数据复制（DATA REPLICATION）是一个分担系统访问压力、加快异地访问响应速度、提高数据可靠性的解决方案。将一个服务器实例上的数据变更复制到另外的服务器实例。可以用于解决大、中型应用中出现的因来自不同地域、不同部门、不同类型的数据访问请求导致数据库服务器超负荷运行、网络阻塞、远程用户的数据响应迟缓的问题。

基于 DM 高级复制技术可以实现数据库的异地快速备份、实时快速同步功能；可以根据实际应用需要，搭建一对多复制、多对一复制、级联复制、对称复制、循环复制等复杂的逻辑复制环境。

4. 故障恢复措施

数据库的备份与还原是系统容灾的重要方法。备份意味着把重要的数据复制到安全的存储介质上，还原则意味着在必要的时候再把以前备份的数据复制到最初的位置，以保证用户可以访问这样的数据。

DM 数据库管理系统支持的备份方式包括物理备份，逻辑备份和 B 树备份，其中 B 树备份是介于物理备份和逻辑备份的一种形态。物理备份分为数据库级的备份和用户表空间级的备份；B 树备份分为数据库级的备份和用户表级的备份。为了提高备份恢复的安全性，减少备份文件占用磁盘空间的大小，系统支持对备份数据加密和压缩。

1.1.5 易用性

为了让用户更加容易掌握达梦数据库，DM 提供了大量功能特性来简化系统的使用、管理和维护，具体表现在：

1. 动态缓冲区

DM 提供了动态缓冲区管理机制，可以更加有效地利用系统内存资源，提高服务器灵活性，减轻系统管理员负担，满足各种应用环境的需求。用户可以在服务器启动时仅分配较少的缓冲数据页，或者直接采用缺省的缓冲区配置参数，就可以满足一般的应用需求。当数据库服务器发现因为缓冲区不足而产生频繁的缓冲区页面淘汰时，系统会自动扩展缓冲区，减少淘汰几率。当系统相对空闲时，DM 又会逐步释放扩展的缓冲区空间，直至最初的规模。

2. 数据库重演

数据库重演（Database Replay）是指在数据库系统上捕获所有负载（记录外部客户端对服务器的请求），保存到二进制捕获文件，并传送到由原始数据库的备份所恢复出来的重演测试系统上。利用捕获文件，重演客户端在系统上所做的所有操作通过重演可再现该段时间内真实环境的负载及运行情况，例如可获取系统运行中的出错的操作。通过该功能，可以方便地再现用户负载，方便用户定位系统所存在的各种问题。

3. 同义词

同义词（Synonym）实际就是让用户能够为数据库的一个模式下的对象提供别名，它可以替换模式下的表，视图，序列，函数，存储过程等对象。同义词通过掩盖一个对象真实的名字和拥有者来提供了一定的安全性，同时使用同义词可以简化复杂的 SQL 语句。

4. 动态性能视图

DM8 中提供了动态性能视图功能（一组以 V$开头的系统表）。通过动态性能视图，数据库管理员可以方便地查看当前服务器诸如锁的信息、缓存使用情况、IO 状况等实时性能信息，便于找出系统瓶颈，进行系统优化。

5. Oracle 兼容性

目前，大多数应用程序使用的是 Oracle 数据库，用户或多或少地使用了 Oracle 的一些特殊功能，而这些特殊功能在其他数据库中都未实现。为了方便应用的移植，DM 实现了很多 Oracle 独特的功能和语法，使得多数 Oracle 的应用可以不用修改而直接移植到 DM 上面。Oracle 兼容性方面实现的功能包括：ROWNUM 表达式、多列 IN 语法、层次查询、外连接语法“(+)”、INSTEAD OF 触发器、%TYPE 以及记录类型等。

6. 类型别名

类型别名实际上是为数据类型提供一个更加容易记住和理解的名字，方便用户使用。实际上，各大数据库的部分数据类型名称也不相同，通过创建数据类型的别名，可以为应用系统的移植和数据的迁移提供便利。

7. 执行计划

为了方便用户对 SQL 语句性能进行分析和调整，DM 提供了显示执行计划的功能。

8. 简便的数据库系统安装和配置

1) 统一的界面，熟悉的环境

DM 提供一个基于 Java 的安装程序，利用 Java 的跨平台性，可以在 Windows、Unix、Linux、Solaris 等平台上运行，且具有统一界面。这样，无论在什么平台上，它都可以为管理员提供一个简洁的安装界面，熟悉、统一的安装环境。

2) 便捷的安装向导

DM 的安装程序把软件安装、数据库初始化和配置结合在一起，一气呵成。

3) 配置灵活

DM 为常见应用做了缺省优化配置，用户可以一路“确认”下来，完成安装，也可自行调整。在安装过程中，安装程序提供一个交互界面来初始化数据库，通过 DM 提供的控制台工具，管理员可以方便地根据实际应用配置 DM 数据库的各项参数，从而获得最大的应用性能。详尽的提示信息减少了用户在安装过程中可能出现的问题。通过降低安装的复杂性，简化配置操作，数据库管理员可提高工作效率，普通技术人员也很容易成为数据库系统管理员。

9. 集成的系统管理工具 Manager

DM 系统管理工具 Manager 是管理 DM 数据库系统的图形化工具，类似于 Oracle 和 MS SQL Server 的 Enterprise Manager。Manager 可以帮助系统管理员更直观、更方便地管理和维护 DM 数据库，普通用户也可以通过 Manager 完成对数据库对象的操作。Manager 的管理功能完备，能对 DM 数据库进行较为全面的管理，在不借助其他工具的情况下，能满足系统管理员和用户的常规要求。

同时，管理工具 Manager 还集成了安全管理的功能：

1) DM 安全管理员可以通过 Manager 进行标记管理，同时它也为 DM 安全管理员提供了管理安全管理员以及安全管理员登录的操作界面；
2) DM 审计员可以用 Manager 来监视、记录、分析用户对数据库的操作，同时它也为 DM 审计管理员提供了管理审计员以及审计员登录的操作界面。

10. 数据迁移工具 DTS

DM 数据迁移工具 DTS 可跨平台实现数据库之间的数据和结构互导，例如 DM 与 DM 之间、DM 与 ORACLE、MS SQL Server 之间等，也可复制从 SQL 查询中获得的数据，还可实现数据库与文本文件之间的数据或者结构互导。

为了实现与 ORACLE、DB2、SQL Server 等多种主流数据库管理系统的数据交换，DM 提供跨平台的数据迁移工具 DTS。它是一个用纯 Java 编写的基于 JDBC/ODBC 的数据迁移工具，可跨平台实现数据库之间的数据和结构互导，也可复制从 SQL 查询中获得的数据，还可实现数据库与文本文件之间的数据或者结构互导。在迁移的过程中它最大限度地保留了源数据的原始信息（包括源数据的类型、精度、默认值、主键和外键约束等），还支持迁移过程中的数据类型自动转换，关于转换方面的细节问题可由数据迁移工具自动来为您解决，数据库管理员所要做的仅仅是指定需要进行数据迁移的两个数据库的连接参数和所迁移的数据。

DM 良好的数据迁移解决方案为系统移植工作减少了很大一部分工作量，免去系统管理员和开发人员的后顾之忧，能够将更多的精力投入到应用程序的移植上面来。

11. 性能监控工具 Monitor

DM 性能监控工具 Monitor 是 DM 系统管理员用来监视服务器的活动和性能情况的客户端工具。它允许系统管理员在本机或远程监控服务器的运行状况，并根据系统情况对系统参数进行调整，以提高系统效率。

12. 控制台工具 Console

DM 控制台（Console）是数据库管理员（DBA）管理和维护 DM 数据库的基本工具。通过使用 DM 控制台，数据库管理员可以完成修改服务器配置参数，启动、停止数据库服务，脱机备份与恢复,以及系统信息查看等任务。

13. 命令行控制台工具

为方便批处理、编程使用，大部分功能都提供命令行方式，如交互式工具 disql、初始化库工具 dminit、备份恢复工具 DMRMAN、快速数据装载工具 dmfldr、导入导出工具 dexp/dimp、数据库重演工具 dreplay 等。

14. 海量数据存储和管理

DM 的数据存储逻辑上分为 4 个层次：数据库实例、表空间、数据文件、数据块。

DM 每个数据库实例理论上可包含多达 65535 个表空间，每个表空间可包含 256 个数据文件，每个数据文件由若干数据块构成。每个数据文件的大小最大为 32K*4G，因此 DM 最大数据存储容量达到 TB 级（实际上远远超过），足以支持大型应用。

此外，DM 全面支持 64 位计算，极大地扩展了系统支持的数据存储和内存容量，这也有利于满足大型应用对海量数据存储和管理的要求。

15. 全文检索

现有的数据库系统，绝大多数是以结构化数据为检索的主要目标，因此实现相对简单。比如数值检索，可以建立一张排序好的索引表，这样速度可以得到提高。但对于非结构化数据，即全文数据，要想实现检索，一般都是采用模糊查询的方式实现的。这种方式不仅速度慢，而且容易将汉字切分错，于是引入了全文索引技术。

全文检索的主要目的，就是实现对大容量的非结构化数据的快速查找。DM 实现了全文检索功能，它根据已有词库建立全文索引，然后文本查询完全在索引上进行。词库（包括中、英文等多种语言）由单独的软件进行维护和更新。全文索引为在字符串数据中进行复杂的词搜索提供了有效支持。全文索引是解决海量数据模糊查询的较好解决办法。

全文检索支持的检索类型有：

1) 支持英文单词的检索（单词不区分大小写）
2) 支持全角英文的检索
3) 支持中文词语的检索
4) 支持常见单个汉字的检索
5) 支持中文长句子的检索
6) 支持中英文混合的检索

1.1.6 对存储模块的支持

DM 系统允许用户使用 DM 提供的 DMSQL 过程语言创建存储过程或存储函数，通常，我们将存储过程和存储函数统称为存储模块。存储模块运行在服务器端，在功能上相当于客户端的一段 SQL 批处理程序，但是在许多方面有着后者无法比拟的优点，它为用户提供了一种高效率的编程手段，成为现代数据库系统的重要特征。

1. 存储模块在执行时数据对用户是不可见的，提高了数据库的安全性；
2. 存储模块是一种高效访问数据库的机制，使用存储模块可减少应用对 DM 的调用，降低了系统资源浪费，显著提高性能，尤其是在网络上与 DM 通讯的应用更显著。

1.1.7 对 Web 应用的支持

DM 提供 ODBC 驱动程序和 OLE DB Provider，支持 ADO、.NET 应用。支持在 ASP 动态网页中访问 DM。

DM 还提供 PHP 接口，支持 PHP 动态网页技术。

DM 提供符合 JDBC Compliant™规范的第四类纯 Java 的 JDBC 驱动程序，可以在以下情形中通过 JDBC 访问 DM：

1. 在 Eclipse、JBuilder 等应用开发工具中；
2. 在 JavaBeans 组件、EJB 组件中；
3. 在 JSP、Applet、Servlet 等基于 Java 的动态网页中。

以上特性使得 DM 适应 Web 应用，用户只用浏览器就可以访问 DM 数据库。

DM 支持 Java 数据对象 JDO（Java Data Objects ），JDO 为对象持久性提供了第一个标准化的、完全面向对象的方法。JDO 简化了用 Java 语言进行数据库编程的复杂性，而且对原始的 Java 源代码的打乱程度最小。

1.2 主要技术指标

1. 定长字符串类型（CHAR）字段最大长度 8188 字节。
2. 变长字符串类型（VARCHAR）字段最大长度字节 8188 字节。
3. 多媒体数据类型字段最大长度（2G-1）字节。
4. 一个记录（不含多媒体数据）最大长度为页大小的一半。
5. 一个记录中最多字段个数 2048。
6. 一个表中最大记录数 256 万亿条。
7. 一个表中最大数据容量 4000PB（受操作系统限制）。
8. 表名、列名等标识符的最大长度 128 字节。

9. 能定义的最大同时连接数为 65000。
10. 每个表空间的最多物理文件数目 256 个。
11. 物理文件的大小为 32K×4G。
12. 每个数据库最多的表/视图/索引等对象的数目各为 16777216。
13. 数值类型的最高精度为 38 个有效数字。
14. 在一个列上允许建立的最多索引数 1020。
15. 表上的最大 UNIQUE 索引数为 64。

第 2 章 DPI 编程指南

2.1 基础简介

本章主要介绍 DPI 的基本概念以及使用方法，以便于用户更好地使用 DPI 编写应用程序。DPI 提供了访问 DM 数据库的最直接的途径。

DPI 的实现参考了 Microsoft ODBC 3.0 标准，函数功能以及调用过程与 ODBC 3.0 十分类似，命名统一采用 dpi 开头的小写英文字母方式，各个单词之间以下划线分割(例：ODBC 函数 SQLAllocStmt 对应的 DPI 函数就是 dpi_alloc_stmt)，用户可以参考《Microsoft ODBC 3.0 程序员参考手册（第二卷）》之 API 参考部分的函数说明及调用方法。

句柄

句柄是用于 DPI 函数申请和使用资源的变量。达梦 DPI 包含以下句柄：

句柄	说明	宏定义
dhenv	环境句柄	DSQL_HANDLE_ENV
dhcon	连接句柄	DSQL_HANDLE_DBC
dhstmt	语句句柄	DSQL_HANDLE_STMT
dhdesc	描述符句柄	DSQL_HANDLE_DESC
dhloblctr	Lob 句柄	DSQL_HANDLE_LOB_LOCATOR
dhobj	复合类型句柄	DSQL_HANDLE_OBJECT
dhobjdesc	复合类型描述符句柄	DSQL_HANDLE_OBJDESC
dhbfile	BFILE 文件句柄	DSQL_HANDLE_BFILE

返回值

DPI 的函数执行结果通过返回值来反馈给用户，DPI 包含以下返回值：

宏定义	值	说明
DSQL_SUCCESS	0	执行成功
DSQL_SUCCESS_WITH_INFO	1	执行成功，有警告信息
DSQL_NO_DATA	100	未取得数据
DSQL_ERROR	-1	执行失败
DSQL_INVALID_HANDLE	-2	非法的句柄
DSQL_NEED_DATA	99	需要数据
DSQL_STILL_EXECUTING	2	语句正在执行
DSQL_PARAM_DATA_AVAILABLE	101	有参数值可以获取

数据类型

数据类型为数据库中字段类型和 C 语言的数据类型。

DPI 中包括以下 DSQL 类型，对应对象创建时指定的类型：

宏定义	定义类型	说明
DSQL_CHAR	char[(n)]	定长字符类型
DSQL_VARCHAR	varchar(n)	变长字符类型
DSQL_BIT	bit	位类型

DSQL_TINYINT	tinyint	有符号小整型（1 字节）
DSQL_SMALLINT	smallint	有符号短整型（2 字节）
DSQL_INT	int	有符号整型（4 字节）
DSQL_BIGINT	bigint	有符号长整型（8 字节）
DSQL_DEC	dec[(p,s)] numeric[(p,s)] number[(p,s)]	精确数字类型
DSQL_FLOAT	real	单精度浮点型
DSQL_DOUBLE	float double	双精度浮点型
DSQL_BLOB	blob image longvarbinary	二进制大字段
DSQL_DATE	date	日期
DSQL_TIME	time[(n)]	时间
DSQL_TIMESTAMP	timestamp[(n)]	时间戳
DSQL_BINARY	binary[(n)]	二进制类型
DSQL_VARBINARY	varbinary[(n)]	变长二进制类型
DSQL_CLOB	clob text longvarchar	字符大字段
DSQL_TIME_TZ	time with time zone	带时区的时间类型
DSQL_TIMESTAMP_TZ	timestamp with time zone	带时区的时间戳类型
DSQL_RSET	cursor	结果集类型
DSQL_CLASS	class	class 复合类型
DSQL_RECORD	record	record 复合类型
DSQL_ARRAY	array	动态 array
DSQL_SARRAY	array	静态 array
DSQL_INTERVAL_YEAR	interval year	年时间间隔类型
DSQL_INTERVAL_MONTH	interval month	月时间间隔类型
DSQL_INTERVAL_DAY	interval day	日时间间隔类型
DSQL_INTERVAL_HOUR	interval hour	时时间间隔类型
DSQL_INTERVAL_MINUTE	interval minute	分时间间隔类型
DSQL_INTERVAL_SECOND	interval second	秒时间间隔类型
DSQL_INTERVAL_YEAR_TO_MONTH	interval year to month	年转月 时间间隔类型
DSQL_INTERVAL_DAY_TO_HOUR	interval day to hour	日转时 时间间隔类型

DSQL_INTERVAL_DAY_TO_MINUTE	interval day to minute	日转分 时间间隔类型
DSQL_INTERVAL_DAY_TO_SECOND	interval day to second	日转秒 时间间隔类型
DSQL_INTERVAL_HOUR_TO_MINUTE	interval hour to minute	时转分 时间间隔类型
DSQL_INTERVAL_HOUR_TO_SECOND	interval hour to second	时转秒 时间间隔类型
DSQL_INTERVAL_MINUTE_TO_SECOND	interval minute to second	分转秒 时间间隔类型

DPI 中包括以下 C 类型，对应绑定时使用的数据类型：

宏定义	类型	说明
DSQL_C_NCHAR	char	字符类型
DSQL_C_SSHORT	signed short	有符号短整型
DSQL_C_USHORT	unsigned short	无符号短整型
DSQL_C_SLONG	signed int	有符号整型
DSQL_C_ULONG	unsigned int	无符号整型
DSQL_C_FLOAT	float	单精度浮点型
DSQL_C_DOUBLE	double	双精度浮点型
DSQL_C_BIT	char	位类型
DSQL_C_STINYINT	char	有符号小整型
DSQL_C_UTINYINT	unsigned char	无符号小整型
DSQL_C_SBIGINT	__int64	有符号长整型，注 1：在 Windows 操作系统下，C 中的定义为__int64，在其他操作系统下会有其他的表示方式
DSQL_C_UBIGINT	unsigned __int64	无符号长整型
DSQL_C_BINARY	unsigned char	二进制类型
DSQL_C_DATE	dpi_date_t	日期类型
DSQL_C_TIME	dpi_time_t	时间类型
DSQL_C_TIMESTAMP	dpi_timestamp_t	日期时间类型
DSQL_C_NUMERIC	dpi_numeric_t	数字类型
DSQL_C_INTERVAL_YEAR	dpi_interval_t	年时间间隔类型
DSQL_C_INTERVAL_MONTH	dpi_interval_t	月时间间隔类型
DSQL_C_INTERVAL_DAY	dpi_interval_t	日时间间隔类型
DSQL_C_INTERVAL_HOUR	dpi_interval_t	时时间间隔类型
DSQL_C_INTERVAL_MINUTE	dpi_interval_t	分时间间隔类型
DSQL_C_INTERVAL_SECOND	dpi_interval_t	秒时间间隔类型
DSQL_C_INTERVAL_YEAR_TO_MONTH	dpi_interval_t	年转月 时间间隔类型
DSQL_C_INTERVAL_DAY_TO_HOUR	dpi_interval_t	日转时 时间间隔类型
DSQL_C_INTERVAL_DAY_TO_MINUTE	dpi_interval_t	日转分 时间间隔类型

DSQL_C_INTERVAL_DAY_TO_SECOND	dpi_interval_t	日转秒 时间间隔类型
DSQL_C_INTERVAL_HOUR_TO_MINUTE	dpi_interval_t	时转分 时间间隔类型
DSQL_C_INTERVAL_HOUR_TO_SECOND	dpi_interval_t	时转秒 时间间隔类型
DSQL_C_INTERVAL_MINUTE_TO_SECOND	dpi_interval_t	分转秒 时间间隔类型
DSQL_C_DEFAULT		自动映射类型
DSQL_C_LOB_HANDLE	dhloblctr	大字段句柄
DSQL_C_RSET	dhstmt	结果集类型
DSQL_C_CLASS	dhobj	复合对象类型
DSQL_C_RECORD	dhobj	复合对象类型
DSQL_C_ARRAY	dhobj	复合对象类型
DSQL_C_SARRAY	dhobj	复合对象类型
DSQL_C_WCHAR	wchar_t	宽字节类型

诊断

函数调用的返回信息放在诊断区域中。每一个环境、连接、及描述符句柄都有一个诊断区域。在诊断区域的头字段返回一般的函数执行信息，它的记录字段记录函数调用的错误信息和警告。用户可以指定获取某一个记录的信息从而更准确地判断函数执行的情况。

2.2 进阶

环境句柄

客户端程序要和数据库服务器进行通信，必须首先进行环境的设置，需要使用到环境句柄。一个环境句柄可以包含多个连接句柄。客户端程序可以通过函数 dpi_alloc_env 来申请一个环境句柄。环境句柄申请成功后就可以通过函数 dpi_alloc_con 来申请连接句柄了。环境句柄包含属性如下表所示：

属性	说明	取值
DSQL_ATTR_LOCAL_CODE	本地编码	PG_GBK 等
DSQL_ATTR_LANG_ID	错误消息的语言	LANGUAGE_EN或者LANGUAGE_CN

连接句柄

客户端程序要和数据库服务器进行通信，必须和数据库服务器建立连接。所以需要先连接数据库服务器，再使用数据库函数 dpi_alloc_con 申请连接句柄。要申请连接句柄必须先成功申请环境句柄。一个连接只能属于一个环境句柄。申请连接句柄成功后可以通过函数 dpi_login 来进行数据的连接登录。

连接句柄包含属性如下表所示：

属性	说明	取值
DSQL_ATTR_ACCESS_MODE	连接的访问模式（可读写）	DSQL_MODE_READ_ONLY 或者 DSQL_MODE_READ_WRITE

		默认 DSQL_MODE_READ_WRITE
DSQL_ATTR_ASYNC_ENABLE	允许异步执行，未提供	未支持
DSQL_ATTR_AUTO_IPD	自动分配参数描述符（只读）	
DSQL_ATTR_AUTOCOMMIT	自动提交（可读写）	DSQL_AUTOCOMMIT_ON 或者 DSQL_AUTOCOMMIT_OFF 默认 DSQL_AUTOCOMMIT_ON
DSQL_ATTR_CONNECTION_DEAD	连接存活，未提供	
DSQL_ATTR_CONNECTION_TIMEOUT	执行超时时间（可读写）	udint4数值
DSQL_ATTR_LOGIN_TIMEOUT	登录超时时间	udint4 数值
DSQL_ATTR_PACKET_SIZE	网络包大小，未提供	
DSQL_ATTR_TXN_ISOLATION	事务隔离级（可读写）	ISO_LEVEL_READ_UNCOMMITTED，ISO_LEVEL_READ_COMMITTED，ISO_LEVEL_SERIALIZABLE
DSQL_ATTR_LOGIN_PORT	登录端口号（可读写）	sdint2数值 默认5236
DSQL_ATTR_STR_CASE_SENSITIVE	大小写是否敏感（只读）	
DSQL_ATTR_MAX_ROW_SIZE	行最大字节数（只读）	
DSQL_ATTR_LOGIN_USER	登陆用户（只读）	
DSQL_ATTR_LOGIN_SERVER	登陆服务器 IP（只读）	
DSQL_ATTR_INSTANCE_NAME	实例名称（只读）	
DSQL_ATTR_CURRENT_SCHEMA	当前模式（只读）	
DSQL_ATTR_SERVER_CODE	服务器的编码（只读）	
DSQL_ATTR_LOCAL_CODE	本地编码（可读写）	PG_SQL_ASCII，PG_UTF8，PG_GBK，PG_BIG5，PG_ISO_8859_9，PG_EUC_JP，PG_EUC_KR，PG_KOI8R，PG_ISO_8859_1，PG_GB18030，PG_ISO_8859_11，PG_UTF16
DSQL_ATTR_LANG_ID	错误消息的语言（可读写）	LANGUAGE_EN 或者 LANGUAGE_CN
DSQL_ATTR_APP_NAME	应用名称（可读写）	
DSQL_ATTR_COMPRESS_MSG	消息压缩（可读写）	DSQL_TRUE 压缩，DSQL_FALSE 不压缩
DSQL_ATTR_RWSEPARATE	读写分离（可读写）	DSQL_TRUE 开启，DSQL_FALSE 关闭
DSQL_ATTR_RWSEPARATE_PERCENT	读写分离比例（可读写）	0-100 默认 25
DSQL_ATTR_CURRENT_CATALOG	（只读）	

DSQL_ATTR_TRX_STATE	事务状态（只读）	DSQL_TRX_ACTIVE 和 DSQL_TRX_COMPLETE
DSQL_ATTR_USE_STMT_POOL	语句句柄缓存池（可读写）	DSQL_TRUE 开启，DSQL_FALSE 关闭
DSQL_ATTR_SSL_PATH	SSL 证书所载的路径路径（可读写）	字符串（最长256）
DSQL_ATTR_SSL_PWD	SSL 加密密码（可读写）	字符串（最长512）
DSQL_ATTR_MPP_LOGIN	mpp 登陆方式（可读写）	DSQL_MPP_LOGIN_GLOBAL 全局登陆 DSQL_MPP_LOGIN_LOCAL 本地登陆
DSQL_ATTR_CRYPTO_NAME	加密方式（可读写）	
DSQL_ATTR_CERTIFICATE	密钥（可读写）	最长4096
DSQL_ATTR_UDP_FLAG	提供 UDP 或 TCP 连接（可读写）	1则表示UDP连接，0则表示TCP连接，默认为0
DSQL_ATTR_LOGIN_CERTIFICATE	指定登录加密用户名密码公钥所在的路径。该属性和 dm_svc.conf 中配置项 LOGIN_CERTIFICATE功能一样。两者不一致时，DSQL_ATTR_LOGIN_CERTIFICATE设置优先	字符串（最长256）

语句句柄

DPI 用语句句柄来存取名称、参数、错误以及其他关于语句处理流程的信息。在一个连接句柄下可以有多个语句句柄，一个特定的 SQL 语句总是和一个句柄连接相联系的。在 DPI 中，通过语句句柄可以了解到语句的状态、当前语句的诊断信息、语句的参数以及结果集绑定的应用程序变量等信息、每一个语句的当前属性值。客户程序调用 dpi_alloc_stmt 函数申请一个语句句柄，用 dpi_free_stmt 函数释放一个语句句柄。

语句句柄包含属性如下表所示：

属性	说明	取值
DSQL_ATTR_ROW_BIND_TYPE	行绑定类型（可读写）	
DSQL_ATTR_ROW_BIND_OFFSET_PTR	行绑定偏移（可读写）	
DSQL_ATTR_ROW_OPERATION_PTR	绑定行数据处理指示（可读写）	
DSQL_ATTR_ROW_STATUS_PTR	获取行数据状态指示（可读写）	
DSQL_ATTR_ROWS_FETCHED_PTR	已获取行数指示（可读写）	
DSQL_ATTR_ROW_ARRAY_SIZE	行集大小（可读写）	
DSQL_ATTR_ROWSET_SI	行集大小（可读写）	

ZE		
DSQL_ATTR_USE_BOOKMARKS	是否使用书签（可读写）	
DSQL_ATTR_FETCH_BOOKMARK_PTR	书签值（可读写）	
DSQL_ATTR_PARAM_BIND_OFFSET_PTR	参数绑定值的偏移位置（可读写）	
DSQL_ATTR_PARAM_BIND_TYPE	参数绑定类型（可读写）	
DSQL_ATTR_PARAM_OPERATION_PTR	参数数据处理指示（可读写）	
DSQL_ATTR_PARAM_STATUS_PTR	参数使用状态（可读写）	
DSQL_ATTR_PARAMS_PROCESSED_PTR	参数集中参数处理的个数（可读写）	
DSQL_ATTR_PARAMSET_SIZE	参数集大小（可读写）	
DSQL_ATTR_ROW_NUMBER	当前行位置（只读）	
DSQL_ATTR_IMP_ROW_DESC	服务器端的行描述（只读）	
DSQL_ATTR_IMP_PARAM_DESC	服务器端参数描述（只读）	
DSQL_ATTR_APP_PARAM_DESC	应用程序参数描述（可读写）	
DSQL_ATTR_APP_ROW_DESC	应用程序行描述（可读写）	
DSQL_ATTR_CURSOR_TYPE	游标类型（可读写）	DSQL_CURSOR_FORWARD_ONLY DSQL_CURSOR_STATIC DSQL_CURSOR_KEYSET_DRIVEN DSQL_CURSOR_DYNAMIC
DSQL_ATTR_CONCURRENCY	游标的并发方式（可读写）	DSQL_CONCUR_READ_ONLY DSQL_CONCUR_LOCK DSQL_CONCUR_ROWVER DSQL_CONCUR_VALUES
DSQL_ATTR_CURSOR_SCROLLABLE	游标是否可滚动（可读写）	DSQL_NONSCROLLABLE DSQL_SCROLLABLE
DSQL_ATTR_CURSOR_SENSITIVITY	结果集修改对其他游标是否可见（可读写）	DSQL_UNSPECIFIED DSQL_INSENSITIVE DSQL_SENSITIVE
DSQL_ATTR_MAX_LENGTH	字符类型或者二进制类型列的最大返回长度（可读写）	
DSQL_ATTR_MAX_ROWS	结果集返回的最大行数（可读写）	
DSQL_ATTR_NOSCAN	是否检查语句中的转义符（可读写),	

	未提供	
DSQL_ATTR_QUERY_TIMEOUT	执行超时时间（可读写），未提供	
DSQL_ATTR_RETRIEVE_DATA	游标滚动后是否检索数据（可读写），未提供	
DSQL_ATTR_ENABLE_AUTO_IPD	自动分配参数描述符（可读写）	DSQL_TRUE 或者 DSQL_FALSE
DSQL_ATTR_ASYNC_ENABLE	异步执行（可读写），未提供	
DSQL_ATTR_KEYSET_SIZE	键集驱动游标结果集中行的大小（可读写），未提供	
DSQL_ATTR_SIMULATE_CURSOR	指定游标更新和删除是否只影响单行（可读写），未提供	
DSQL_ATTR_METADATA_ID	（可读写）	DSQL_TRUE DSQL_FALSE
DSQL_ATTR_SQL_CHARSET	字符集编码（可读写）	PG_SQL_ASCII，PG_UTF8， PG_GBK，PG_BIG5， PG_ISO_8859_9，PG_EUC_JP， PG_EUC_KR，PG_KOI8R， PG_ISO_8859_1，PG_GB18030， PG_ISO_8859_11，PG_UTF16
DSQL_ATTR_IGN_BP_ERR	批量参数错误数据处理策略（可读写）	DSQL_TRUE DSQL_FALSE

描述符句柄

一个描述符句柄是指包含列或者动态参数信息的一个数据结构。对于许多应用程序，直接访问与操作描述符会使得操作更加简单。在 DPI 中描述符分为以下几种类型：

1. 应用程序参数描述符（APD）：包含被应用程序设置的输入动态参数或者随执行 SQL 语句中的 CALL 产生的输出动态参数；
2. 驱动执行参数描述符（IPD）：对于输入参数，在完成应用程序指定的数据转换之后，它包含了与 APD 相同的参数。对于输出参数，在进行了一些应用程序指定的数据转换之前，它包含了返回的参数；
3. 驱动执行行描述符（IRD）：包含数据库中的一行；
4. 应用程序行描述符（ARD）：包含数据的行。

描述符字段包括头字段与记录域字段，它们全面地描述列与参数。

一个描述符包括了以下的头字段：

属性	说明
DSQL_DESC_ALLOC_TYPE	描述符的分配类型（只读）
DSQL_DESC_ARRAY_SIZE	描述符的记录数组大小（APD,ARD 可读写）
DSQL_DESC_ARRAY_STATUS_PTR	数据状态指示（可读写）
DSQL_DESC_BIND_OFFSET_PTR	绑定数据偏移（APD,ARD 可读写）
DSQL_DESC_BIND_TYPE	绑定类型（APD,ARD 可读写）
DSQL_DESC_COUNT	描述符包含记录个数（APD,ARD,IPD 可读写，IRD 只读）

DSQL_DESC_ROWS_PROCESSED_PTR	处理数据的个数（IRD,IPD 可读写）

一个描述符包括了以下的记录域字段：

属性	说明
DSQL_DESC_AUTO_UNIQUE_VALUE	结果集列是否是自动增长的（IRD 只读）
DSQL_DESC_BASE_COLUMN_NAME	结果集列的基列列名（IRD 只读）
DSQL_DESC_BASE_TABLE_NAME	结果集列的基表名（IRD 只读）
DSQL_DESC_CASE_SENSITIVE	对象是否大小写敏感（IRD,IPD 只读）
DSQL_DESC_CATALOG_NAME	列对应的库名（IRD 只读）
DSQL_DESC_CONCISE_TYPE	对象对应的精简类型（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_DATA_PTR	绑定数据的地址（APD,ARD 可读写）
DSQL_DESC_DATETIME_INTERVAL_CODE	时间日期类型，时间日期间隔类型的子类型（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_DATETIME_INTERVAL_PRECISION	时间间隔类型的引导精度（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_DISPLAY_SIZE	列的显示长度（IRD 只读）
DSQL_DESC_FIXED_PREC_SCALE	指示是否是固定精度刻度的数字类型（IPD,IRD 只读）
DSQL_DESC_INDICATOR_PTR	指示符地址（APD,ARD 可读写）
DSQL_DESC_LABEL	列的标签（IRD 只读）
DSQL_DESC_LENGTH	字符类型的最大或实际字符长度或者二进制类型的最大或实际字节长度（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_LITERAL_PREFIX	列的前缀字符（IRD 只读）
DSQL_DESC_LITERAL_SUFFIX	列的后缀字符（IRD 只读）
DSQL_DESC_LOCAL_TYPE_NAME	本地的类型名（IPD,IRD 只读）
DSQL_DESC_NAME	列的别名（IPD 可读写，IRD 只读）
DSQL_DESC_NULLABLE	记录域是否可以为空（IPD,IRD 只读）
DSQL_DESC_NUM_PREC_RADIX	记录域数字类型精度的表示基数（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_OCTET_LENGTH	记录域的最大字节长度（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_OCTET_LENGTH_PTR	指示绑定字符类型或二进制类型的实际字节长度（APD,ARD 可读写）
DSQL_DESC_PARAMETER_TYPE	参数类型（IPD 可读写）
DSQL_DESC_PRECISION	精确数字类型的位数或者近似数字类型的尾数尾数，或者时间类型，时间戳类型，秒间隔类型秒精度（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_ROWVER	指示列是否记录行的版本信息（IPD,IRD 只读）
DSQL_DESC_SCALE	DEC 或者 NUMERIC 类型的刻度

DSQL_DESC_SCHEMA_NAME	列所属基表所在的模式名（IRD 只读）
DSQL_DESC_SEARCHABLE	列的条件查询方式（IRD 只读）
DSQL_DESC_TABLE_NAME	列所在的基表表名（IRD 只读）
DSQL_DESC_TYPE	C 类型或者 DSQL 类型，时间或者间隔类型则表示的为详细的类型 DSQL_DT 或 DSQL_INTERVAL（APD,ARD,IPD 可读写，IRD 只读）
DSQL_DESC_TYPE_NAME	数据源的数据类型（IPD,IRD 只读）
DSQL_DESC_UNNAMED	记录域是否是命名的（IPD 可读写，IRD 只读）
DSQL_DESC_UNSIGNED	指示记录域是否为无符号数字类型（IPD,IRD 只读）
DSQL_DESC_UPDATABLE	列是否为可更新（IRD 只读）
DSQL_DESC_BIND_PARAMETER_TYPE	参数类型（IPD 可读写）
DSQL_DESC_DATETIME_FORMAT	时间类型格式（APD, ARD 可读写）
DSQL_DESC_CHARSET	字符集编码（APD, ARD 可读写）

lob 句柄

lob 句柄为大字段操纵句柄，可以对大字段进行更新和读取处理。通过 lob 句柄可以获取大字段对象的长度、读取大字段实际数据、更新大字段数据、截取大字段数据。lob 句柄从属于某一个语句句柄，可以通过 dpi_alloc_handle 或者 dpi_alloc_lob_locator 函数进行分配，再通过绑定到某个大字段对象进行操作。在释放语句句柄时必须先释放语句句柄上所有的 lob 句柄，否则函数会返回执行失败的错误。

诊断信息

诊断信息反映函数执行情况，某个函数若返回 DSQL_SUCCESS_WITH_INFO 或者 DSQL_ERROR，就能通过调用 dpi_get_diag_rec 或者 dpi_get_diag_field 函数获取具体的诊断信息，从而有效的定位错误原因。

诊断信息分为诊断头信息以及诊断域信息。

诊断头信息包含如下字段：

属性	说明
DSQL_DIAG_NUMBER	诊断信息个数
DSQL_DIAG_DYNAMIC_FUNCTION_CODE	语句类型
DSQL_DIAG_ROW_COUNT	语句执行影响的行数
DSQL_DIAG_PRINT_INFO	存储过程或语句块的打印信息
DSQL_DIAG_EXPLAIN	explain 语句的结果

诊断域信息包含如下字段。

属性	说明
DSQL_DIAG_COLUMN_NUMBER	出错的列号
DSQL_DIAG_ROW_NUMBER	出错的行号
DSQL_DIAG_MESSAGE_TEXT	错误信息
DSQL_DIAG_ERROR_CODE	错误码

2.3 函数原型

1. dpi_alloc_handle

函数

```
DPIRETURN
dpi_alloc_handle(
        sdint2          hndl_type,
        dhandle         hndl_in,
        dhandle         *hdnl_out
);
```

功能：用于分配环境、连接、语句、描述符、lob、复合类型对象句柄。

参数

1) hndl_type

输入参数，需要分配句柄的类型，必须为下列值之一：

- DSQL_HANDLE_ENV
- DSQL_HANDLE_DBC
- DSQL_HANDLE_STMT
- DSQL_HANDLE_DESC
- DSQL_HANDLE_LOB_LOCATOR
- DSQL_HANDLE_OBJECT

2) hndl_in

输入参数，通过此句柄分配新的句柄，新句柄运行环境从属此句柄。

如果 hndl_type 为 DSQL_HANDLE_ENV，则这个值为 NULL；

如果 hndl_type 为 DSQL_HANDLE_DBC，则这个值必须是一个环境句柄；

如果 hndl_type 为 DSQL_HANDLE_STMT 、 DSQL_HANDLE_DESC 或者 DSQL_HANDLE_OBJECT，则这个值必须是一个连接句柄；

如果 hndl_type 为 DSQL_HANDLE_LOB_LOCATOR，则这个值必须是一个语句句柄。

3) hndl_out

输出参数，一个存放新分配句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明

如果函数执行失败，则 hndl_out 返回 NULL。

2. dpi_free_handle

函数

```
DPIRETURN
dpi_free_handle(
    sdint2          hndl_type,
    dhandle         hndl
```

```
);
```

功能

释放资源，用于释放句柄资源。

参数

hdle_type
输入参数，需要被释放句柄的类型。必须为下列值之一：

- DSQL_HANDLE_ENV
- DSQL_HANDLE_DBC
- DSQL_HANDLE_STMT
- DSQL_HANDLE_DESC
- DSQL_HANDLE_LOB_LOCATOR
- DSQL_HANDLE_OBJECT

hndl
输入参数，需要被释放的句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

若函数执行失败，则需要被释放的句柄依然有效。

3. dpi_alloc_env

函数

```
DPIRETURN
dpi_alloc_env(
    dhenv          *dpi_henv
);
```

功能

分配环境句柄。

参数

dpi_henv
输出参数，一个存放新分配环境句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR

说明

该函数作用与 dpi_alloc_handle 指定分配 DSQL_HANDLE_ENV 一致。

4. dpi_alloc_con

函数

```
DPIRETURN
dpi_alloc_con(
    dhenv          dpi_henv,
    dhcon          *dpi_hcon
);
```

功能

分配连接句柄。

参数

1) dpi_henv
输入参数，通过此环境句柄分配新的连接句柄，新句柄运行环境从属此句柄。

2) dpi_hcon
输出参数，一个存放新分配连接句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_alloc_handle 指定分配 DSQL_HANDLE_DBC 一致。

5. dpi_alloc_stmt

函数

```
DPIRETURN
dpi_alloc_stmt(
    dhcon          dpi_hcon,
    dhstmt         *dpi_hstmt
);
```

功能

分配语句句柄。

参数

1) dpi_hcon
输入参数，通过此连接句柄分配新的语句句柄，新句柄运行环境从属此句柄。

2) dpi_hstmt
输出参数，一个存放新分配语句句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_alloc_handle 指定分配 DSQL_HANDLE_STMT 一致。

6. dpi_alloc_desc

函数

```
DPIRETURN
dpi_alloc_desc(
    dhcon          dpi_hcon,
    dhdesc         *dpi_hdesc
);
```

功能

分配描述符句柄。

参数

1) dpi_hcon
输入参数，通过此连接句柄分配新的描述符句柄，新句柄运行环境从属此句柄。

2) dpi_hdesc
输出参数，一个存放新分配描述符句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_alloc_handle 指定分配 DSQL_HANDLE_DESC 一致。

7. dpi_alloc_lob_locator

函数

```
DPIRETURN
dpi_alloc_lob_locator(
    dhstmt         dpi_hstmt,
    dhloblctr      *dpi_loblctr
);
```

功能

分配 lob 句柄。

参数

1) dpi_hstmt
输入参数，通过此语句句柄分配新的 lob 句柄，新句柄运行环境从属此句柄。

2) dpi_loblctr
输出参数，一个存放新分配 lob 句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_alloc_handle 指定分配 DSQL_HANDLE_LOB_LOCATOR 一致。

8. dpi_free_env

函数

```
DPIRETURN
dpi_free_env(
    dhenv          dpi_henv
);
```

功能

释放环境句柄。

参数

dpi_henv
输入参数，需要被释放的环境句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_free_handle 指定 DSQL_HANDLE_ENV 一致。

9. dpi_free_con

函数：

```
DPIRETURN
dpi_free_con(
    dhcon          dpi_hcon
);
```

功能

释放连接句柄。

参数

dpi_hcon
输入参数，需要被释放的连接句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVLAID_HANDLE

说明

该函数作用与 dpi_free_handle 指定 DSQL_HANDLE_DBC 一致。在释放连接时必须保证连接已经与服务器断开，否则函数失败。

10. dpi_free_stmt

函数

```
DPIRETURN
dpi_free_stmt(
    dhstmt          dpi_hstmt
);
```

功能

释放语句句柄。

参数

dpi_hstmt
输入参数，需要被释放的语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_free_handle 指定 DSQL_HANDLE_STMT 一致。在释放语句句柄时必须保证句柄上分配的 lob 句柄都已经释放，否则函数执行失败。

11. dpi_free_desc

函数

```
DPIRETURN
dpi_free_desc(
    dhdesc          dpi_hdesc
);
```

功能

释放描述符句柄。

参数

dpi_hdesc
输入参数，需要被释放的描述符句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_free_handle 指定 DSQL_HANDLE_DESC 一致。所要释放的描述符句柄必须为手动分配的，否则函数执行失败。

12. dpi_free_lob_locator

函数

```
DPIRETURN
dpi_free_lob_locator(
    dhloblctr    dpi_loblctr
);
```

功能

释放 lob 句柄。

参数

dpi_loblctr
输入参数，需要被释放的 lob 句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

该函数作用与 dpi_free_handle 指定 DSQL_HANDLE_LOB_LOCATOR 一致。

13. dpi_set_env_attr

函数

```
DPIRETURN
dpi_set_env_attr(
    dhenv           dpi_henv,
    sdint4          attr_id,
    dpointer        val,
    sdint4          val_len
);
```

功能

设置环境句柄的属性。

参数

1) dpi_henv
输入参数，需要被设置属性的环境句柄。

2) attr_id
输入参数，设置的属性，参见说明。

3) val
输入参数，所需设置属性的值。根据属性的不同，值可能为 32 位的整形数或者以 0 结尾的字符串。

4) val_len
输入参数，如果设置的 val 指向字符串或者二进制缓冲区，此参数表示 val 的字节长度。如果 val 为整形数，则此参数忽略。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明

属性	字段类型
DSQL_ATTR_LOCAL_CODE	sdint4
DSQL_ATTR_LANG_ID	sdint4

14. dpi_set_con_attr

函数

```
DPIRETURN
dpi_set_con_attr(
    dhcon       dpi_hcon,
    sdint4      attr_id,
    dpointer    val,
    sdint4      val_len
);
```

功能

设置连接句柄属性。

参数

1) dpi_hcon

输入参数，需要被设置属性的连接句柄。

2) attr_id

输入参数，设置的属性，参见说明。

3) val

输入参数，所需设置属性的值。根据属性的不同，值可能为 32 位的整形数或者以 0 结尾的字符串。

4) val_len

输入参数，如果设置的 val 指向字符串或者二进制缓冲区，此参数表示 val 的字节长度。如果 val 为整形数，则此参数忽略。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明

属性	字段类型
DSQL_ATTR_ACCESS_MODE	udint4
DSQL_ATTR_ASYNC_ENABLE	udint4
DSQL_ATTR_AUTO_IPD	udint4
DSQL_ATTR_AUTOCOMMIT	udint4

DSQL_ATTR_CONNECTION_DEAD	udint4
DSQL_ATTR_CONNECTION_TIMEOUT	udint4
DSQL_ATTR_LOGIN_TIMEOUT	udint4
DSQL_ATTR_PACKET_SIZE	udint4
DSQL_ATTR_TXN_ISOLATION	sdint4
DSQL_ATTR_ATTR_ENCRYPT	udint4
DSQL_ATTR_LOGIN_PORT	sdint2
DSQL_ATTR_STR_CASE_SENSITIVE	udint4
DSQL_ATTR_MAX_ROW_SIZE	udint4
DSQL_ATTR_LOGIN_USER	sdbyte*
DSQL_ATTR_LOGIN_SERVER	sdbyte*
DSQL_ATTR_INSTANCE_NAME	sdbyte*
DSQL_ATTR_CURRENT_SCHEMA	sdbyte*
DSQL_ATTR_SERVER_CODE	sdint4
DSQL_ATTR_LOCAL_CODE	sdint4
DSQL_ATTR_LANG_ID	sdint4
DSQL_ATTR_APP_NAME	sdbyte*
DSQL_ATTR_COMPRESS_MSG	sdint4
DSQL_ATTR_CURRENT_CATALOG	sdbyte*
DSQL_ATTR_TRX_STATE	sdint4
DSQL_ATTR_USE_STMT_POOL	dint4
DSQL_ATTR_SSL_PATH	sdbyte*
DSQL_ATTR_SSL_PWD	sdbyte*
DSQL_ATTR_MPP_LOGIN	sdint4
DSQL_ATTR_CRYPTO_NAME	sdbyte*
DSQL_ATTR_CERTIFICATE	sdbyte*
DSQL_ATTR_UDP_FLAG	sdint4

15. dpi_set_stmt_attr

函数

```
DPIRETURN
dpi_set_stmt_attr(
    dhstmt          dpi_hstmt,
    sdint4          attr_id,
    dpointer        val,
    sdint4          val_len
);
```

功能

设置语句句柄属性。

参数

1) dpi_hstmt

输入参数，需要被设置属性的语句句柄。

2) attr_id

输入参数，设置的属性，参见说明。

3) val

输入参数，所需设置属性的值。根据属性的不同，值可能为下列之一：

- 一个描述符句柄
- 一个 udint4 类型的值
- 一个 ulength 类型的值
- 一个下列情况之一的指针：
  - 一个以 0 结尾的字符串
  - 一个二进制缓冲区
  - 一个 slength、ulength 或者 udint2 类型的值或者数组
  - 一个驱动定义的值

4) val_len

输入参数，如果设置的 val 指向字符串或者二进制缓冲区，此参数表示 val 的字节长度。如果 val 为整形数，则此参数忽略。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明

属性	字段类型	属性	字段类型
DSQL_ATTR_ROW_BIND_TYPE	ulength	DSQL_ATTR_IMP_ROW_DESC	void*
DSQL_ATTR_ROW_BIND_OFFSET_PTR	ulength*	DSQL_ATTR_IMP_PARAM_DESC	void*
DSQL_ATTR_ROW_OPERATION_PTR	udint2*	DSQL_ATTR_APP_PARAM_DESC	void*
DSQL_ATTR_ROW_STATUS_PTR	udint2*	DSQL_ATTR_APP_ROW_DESC	void*
DSQL_ATTR_ROWS_FETCHED_PTR	ulength*	DSQL_ATTR_CURSOR_TYPE	ulength
DSQL_ATTR_ROW_ARRAY_SIZE	ulength	DSQL_ATTR_CONCURRENCY	ulength
DSQL_ATTR_ROWSET_SIZE	ulength	DSQL_ATTR_CURSOR_SCROLLABLE	ulength
DSQL_ATTR_USE_BOOKMARKS	ulength	DSQL_ATTR_CURSOR_SENSITIVITY	ulength
DSQL_ATTR_FETCH_BOOKMARK_PTR	slength*	DSQL_ATTR_MAX_LENGTH	ulength
DSQL_ATTR_PARAM_BIND_OFFSET_PTR	ulength*	DSQL_ATTR_MAX_ROWS	ulength
DSQL_ATTR_PARAM_BIND_TYPE	ulength	DSQL_ATTR_NOSCAN	ulength
DSQL_ATTR_PARAM_OPERATION_PTR	udint2*	DSQL_ATTR_QUERY_TIMEOUT	ulength
DSQL_ATTR_PARAM_STATUS_PTR	udint2*	DSQL_ATTR_RETRIEVE_DATA	ulength
DSQL_ATTR_PARAMS_PROCESSED_PTR	ulength*	DSQL_ATTR_ENABLE_AUTO_IPD	ulength
DSQL_ATTR_PARAMSET_SIZE	ulength	DSQL_ATTR_ASYNC_ENABLE	ulength
DSQL_ATTR_ROW_NUMBER	ulength	DSQL_ATTR_KEYSET_SIZE	ulength

16. dpi_get_env_attr

函数

```
DPIRETURN
dpi_get_env_attr(
```

```
    dhenv           dpi_henv,
    sdint4          attr_id,
    dpointer        val,
    sdint4          buf_len,
    sdint4          *val_len
);
```

功能

获取环境句柄属性。

参数

1) dpi_henv
输入参数，获取属性的环境句柄。

2) attr_id
输入参数，需要获取的属性。

3) val
输出参数，指向返回指定属性当前值的缓冲区的指针。

4) buf_len
输入参数，如果 val 返回的是字符串，则此参数为 val 缓冲区的长度。如果 val 返回的不是字符串，则此参数忽略。

5) val_len
输出参数，指向返回 val 中可提供字符串的总长度的缓冲区的指针。如果 val 为 NULL，则不返回长度，如果属性值为字符串，总长度大于等于 buf_len，则 val 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

属性值参见函数 dpi_set_env_attr。

17. dpi_get_con_attr

函数

```
DPIRETURN
dpi_get_con_attr(
    dhcon           dpi_hcon,
    sdint4          attr_id,
    dpointer        val,
    sdint4          buf_len,
    sdint4          *val_len
);
```

功能

获取连接句柄属性。

参数

1) dpi_hcon

输入参数，获取属性的连接句柄。

2) attr_id

输入参数，需要获取的属性。

3) val

输出参数，指向返回指定属性当前值的缓冲区的指针。

4) buf_len

输入参数，如果 val 返回的是字符串，则此参数为 val 缓冲区的长度。如果 val 返回的不是字符串，则此参数忽略。

5) val_len

输出参数，指向返回 val 中可提供字符串的总长度的缓冲区的指针。如果 val 为 NULL，则不返回长度，如果属性值为字符串，总长度大于等于 buf_len，则 val 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明

属性值参见函数 dpi_set_con_attr。

18. dpi_get_stmt_attr

函数

```
DPIRETURN
dpi_get_stmt_attr(
    dhstmt          dpi_hstmt,
    sdint4          attr_id,
    dpointer        val,
    sdint4          buf_len,
    sdint4          *val_len
);
```

功能

获取语句句柄属性。

参数

1) dpi_hstmt

输入参数，获取属性的语句句柄。

2) attr_id

输入参数，需要获取的属性。

3) val

输出参数，指向返回指定属性当前值的缓冲区的指针。

4) buf_len

输入参数，如果 val 返回的是字符串，则此参数为 val 缓冲区的长度。如果 val 返回的不是字符串，则此参数忽略。

5) val_len

输出参数，指向返回 val 中可提供字符串的总长度的缓冲区的指针。如果 val 为 NULL，则不返回长度，如果属性值为字符串，总长度大于等于 buf_len，则 val 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

属性值参见函数 dpi_set_stmt_attr

19. dpi_get_diag_rec

函数

```
DPIRETURN
dpi_get_diag_rec(
    sdint2          hndl_type,
    dhandle         hndl,
    sdint2          rec_num,
    sdint4          *err_code,
    sdbyte          *err_msg,
    sdint2          buf_sz,
    sdint2          *msg_len
);
```

功能

获取复合字段的诊断信息。

参数

hndl_type
输入参数，需要获取诊断信息的句柄类型。必须为下列值之一：

- DSQL_HANDLE_ENV
- DSQL_HANDLE_DBC
- DSQL_HANDLE_STMT
- DSQL_HANDLE_DESC
- DSQL_HANDLE_LOB_LOCATOR
- DSQL_HANDLE_OBJECT
- DSQL_HANDLE_OBJECTDESC

1) hndl
输入参数，需要取得诊断信息且由 hndl_type 所标示的句柄。

2) rec_num
输入参数，诊断信息的索引号。索引起始为 1。

3) err_code
输出参数，指向用于记录错误码的缓冲区的指针。此信息包含在 DSQL_DIAG_ERROR_CODE 诊断域中。

4) err_msg
输出参数，指向用于记录诊断消息缓冲区的指针。此信息包含在

DSQL_DIAG_MESSAGE_TEXT 诊断域中。

5) buf_sz

输入参数，err_msg 缓冲区的长度。

6) msg_len

输出参数，指向用于返回诊断消息总长度的缓冲区的指针。如果诊断消息长度大于等于 buf_sz，则诊断消息被截断且以 0 结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NO_DATA_FOUND

说明

函数返回值含义：

- DSQL_SUCCESS：函数成功返回诊断信息。
- DSQL_SUCCESS_WITH_INFO：err_msg 缓冲区长度不足，诊断信息被截断。
- DSQL_ERROR：发生了下列情况之一：
  - rec_num 小于等于 0。
  - buf_sz 小于 0。
- DSQL_INVALID_HANDLE：由 hndl_type 所标示的 hndl 为一个无效的句柄。
- DSQL_NO_DATA：rec_num 大于句柄所含有的诊断信息总数。

20. dpi_get_diag_field

函数

```
DPIRETURN
dpi_get_diag_field(
    sdint2      hndl_type,
    dhandle     hndl,
    sdint2      rec_num,
    sdint2      diag_id,
    dpointer    diag_info,
    slength     buf_len,
    slength     *info_len
);
```

功能

获取诊断信息的某个指定域的值。

参数

1) hndl_type

输入参数，需要获取诊断信息的句柄类型。必须为下列值之一：

- DSQL_HANDLE_ENV
- DSQL_HANDLE_DBC
- DSQL_HANDLE_STMT
- DSQL_HANDLE_DESC

- DSQL_HANDLE_LOB_LOCATOR
- DSQL_HANDLE_OBJECT
- DSQL_HANDLE_OBJECTDESC

2) hndl

输入参数，需要取得诊断信息且由 hndl_type 所标示的句柄。

3) rec_num

输入参数，诊断信息的索引号。索引起始为 1。如果 diag_id 为诊断头信息，则此参数被忽略。

4) diag_id

输入参数，指定需要获取值的诊断域。更多信息参见说明。

5) diag_info

输出参数，指向返回诊断域信息的缓冲区的指针。数据类型依赖于 diag_id。

6) buf_len

输入参数，如果 diag_info 返回的是字符串，则此参数为 diag_info 缓冲区的长度。如果 diag_info 返回的不是字符串，则此参数忽略。

7) info_len

输出参数，指向返回 diag_info 中可提供字符串的总长度的缓冲区的指针。如果 diag_info 为 NULL，则不返回长度，如果 diag_info 值为字符串，总长度大于等于 buf_len，则 diag_info 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

DSQL_NO_DATA

说明

函数返回值含义：

- DSQL_SUCCESS：函数成功返回诊断信息。
- DSQL_SUCCESS_WITH_INFO：diag_info 缓冲区长度不足，诊断信息被截断。
- DSQL_ERROR：发生了下列情况之一：
  - rec_num 小于等于 0。对于诊断头，忽略 rec_num。
  - 所要求的诊断域的值为字符串，而 buf_sz 小于 0。
  - diag_id 不是一个有效的诊断域。
  - diag_id 为 DSQL_DIAG_ROW_COUNT，DSQL_DIAG_DYNAMIC_FUNCTION_CODE，DSQL_DIAG_PRINT_INFO 或者 DSQL_DIAG_EXPLAIN，但 hndl 不是一个语句句柄。
- DSQL_INVALID_HANDLE：由 hndl_type 所标示的 hndl 为一个无效的句柄。
- DSQL_NO_DATA：rec_num 大于句柄所含有的诊断信息总数。

21. dpi_login

函数

```
DPIRETURN
dpi_login(
    dhcon          dpi_hcon,
```

```
    sdbyte*        svr,
    sdbyte*        user,
    sdbyte*        pwd
);
```

功能

与指定的数据库服务器建立连接。

参数

1) dpi_hcon
输入参数，连接句柄。

2) svr
输入参数，数据库服务器所对应的地址。

3) user
输入参数，登录数据库服务器的登录名。

4) pwd
输入参数，登录数据库服务器的口令。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

若需要登录的服务器的端口不是默认端口，则需要在调用之前设置连接属性 DSQL_ATTR_LOGIN_PORT，或者 svr 设置为 ip:port 形式，例如：192.168.0.143:5289。登录函数支持以服务名登录，需要配置 dm_svc.conf 文件。

22. dpi_logout

函数

```
DPIRETURN
dpi_logout(
    dhcon          dpi_hcon
);
```

功能

断开与数据库服务器的连接。

参数

dpi_hcon
输入参数，连接句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

无

23. dpi_commit

函数

```
DPIRETURN
dpi_commit(
    dhcon          dpi_hcon
);
```

功能

提交连接上的所有活动事务。

参数：

dpi_hcon
输入参数，连接句柄。

返回值：

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

手动提交事务，必须将连接的自动提交属性设置为非自动提交，否则无效果。默认为自动提交。

24. dpi_rollback

函数

```
DPIRETURN
dpi_rollback(
    dhcon          dpi_hcon
);
```

功能

回滚连接上的所有活动事务。

参数：

dpi_hcon
输入参数，连接句柄。

返回值：

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

手动回滚事务，必须将连接的自动提交属性设置为非自动提交，否则无效果。默认为自

动提交。

25. dpi_copy_desc

函数

```
DPIRETURN
dpi_copy_desc(
    dhdesc      src_desc,
    dhdesc      target_desc
);
```

功能

复制一个描述符句柄的信息到另外一个描述符句柄上。

参数说明：

1) src_desc

输入参数，源描述符句柄。

2) target_desc

输入参数，目的描述符句柄。目的描述符句柄可以为 APD、ARD 或者 IPD，但不能为 IRD。

返回值：

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

描述符句柄拷贝。只能拷贝到 APD、ARD 或者 IPD，不能拷贝到 IRD。

26. dpi_set_desc_rec

函数

```
DPIRETURN
dpi_set_desc_rec(
    dhdesc          dpi_desc,
    sdint2          rec_num,
    sdint2          type,
    sdint2          sub_type,
    slength         length,
    sdint2          prec,
    sdint2          scale,
    dpointer        data_ptr,
    slength*        str_len,
    slength*        ind_ptr
);
```

功能

设置复合的描述符域的信息。这些信息可以影响所绑定列或者参数的数据类型和绑定的缓冲区信息。

参数：

1) dpi_desc

输入参数，描述符句柄。不能为 IRD 句柄。

2) rec_num

输入参数，包含所设置域的描述符记录的索引号。此参数从 0 起始。当为 0 时，表示所设置的为书签列。此参数值必须大于等于 0。如果此参数的值大于 DSQL_DESC_COUNT 的值，则 DSQL_DESC_COUNT 的值改变为 rec_num。

3) type

输入参数，设置描述符记录 DSQL_DESC_TYPE 域的值。

4) sub_type

输入参数，对于类型为 DSQL_DT 或者 DSQL_INTERVAL 的记录，此参数值设置 DSQL_DESC_DATETIME_INTERVAL_CODE 域的值。

5) length

输入参数，此参数值设置描述符记录的 DSQL_DESC_OCTET_LENGTH 域的值。

6) prec

输入参数，此参数值设置描述符记录的 DSQL_DESC_PRECISION 域的值。

7) scale

输入参数，此参数值设置描述符记录的 DSQL_DESC_SCALE 域的值。

8) data_ptr

延缓输入或输出参数，此参数值设置描述符记录的 DSQL_DESC_DATA_PTR 域的值。data_ptr 可以设置为 NULL。如果 data_ptr 设置为 NULL，dpi_desc 又与 ARD 关联，则这意味着清空了绑定信息。

9) str_len

延缓输入或输出参数，此参数值设置描述符记录的 DSQL_DESC_OCTET_LENGTH_PTR 域的值。str_len 可以设置为 NULL。

10) ind_ptr

延缓输入或输出参数，此参数值设置描述符记录的 DSQL_DESC_INDICATOR_PTR 域的值。ind_ptr 可以设置为 NULL。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

具体参见 dpi_set_desc_field 函数说明。

27. dpi_set_desc_field

函数

```
DPIRETURN
dpi_set_desc_field(
    dhdesc      dpi_desc,
    sdint2      rec_num,
    sdint2      field,
```

```
    dpointer      val,
    sdint4        val_len
);
```

功能

设置描述符记录单个域的值。

参数：

1) dpi_desc
输入参数，描述符句柄。

2) rec_num
输入参数，包含所设置域的描述符记录的索引号。此参数从 0 起始。当为 0 时，表示所设置的为书签列。此参数值必须大于等于 0。如果此参数的值大于 DSQL_DESC_COUNT 的值，则 DSQL_DESC_COUNT 的值改变为 rec_num。

3) field
输入参数，所要设置的域。详见说明。

4) val
输入参数，指向包含描述符信息的缓冲区，或者一个 4 字节的值。数据类型依赖于 field 的值。如果 val 为一个 4 字节的值，则是 4 字节还是 2 字节的值依赖于 field 参数。

5) val_len
输入参数，如果 val 指向一个字符串或者二进制缓冲区，则此参数为 val 的长度。如果 val 是一个整形数，则此参数被忽略。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

属性	字段类型	属性	字段类型
DSQL_DESC_ALLOC_TYPE	sdint2	DSQL_DESC_LITERAL_PREFIX	sdbyte*
DSQL_DESC_ARRAY_SIZE	ulength	DSQL_DESC_LITERAL_SUFFIX	sdbyte*
DSQL_DESC_ARRAY_STATUS_PTR	udint2*	DSQL_DESC_LOCAL_TYPE_NAME	sdbyte*
DSQL_DESC_BIND_OFFSET_PTR	slength*	DSQL_DESC_NAME	sdbyte*
DSQL_DESC_BIND_TYPE	ulength	DSQL_DESC_NULLABLE	sdint2
DSQL_DESC_COUNT	sdint2	DSQL_DESC_NUM_PREC_RADIX	sdint4
DSQL_DESC_ROWS_PROCESSED_PTR	ulength*	DSQL_DESC_OCTET_LENGTH	slength
DSQL_DESC_AUTO_UNIQUE_VALUE	sdint4	DSQL_DESC_OCTET_LENGTH_PTR	slength*
DSQL_DESC_BASE_COLUMN_NAME	sdbyte*	DSQL_DESC_PARAMETER_TYPE	sdint2
DSQL_DESC_BASE_TABLE_NAME	sdbyte*	DSQL_DESC_PRECISION	sdint2
DSQL_DESC_CASE_SENSITIVE	sdint4	DSQL_DESC_ROWVER	sdint2

DSQL_DESC_CATALOG_NAME	sdbyte*	DSQL_DESC_SCALE	sdint2
DSQL_DESC_CONCISE_TYPE	sdint2	DSQL_DESC_SCHEMA_NAME	sdbyte*
DSQL_DESC_DATA_PTR	void*	DSQL_DESC_SEARCHABLE	sdint2
DSQL_DESC_DATETIME_INTERVAL_CODE	sdint2	DSQL_DESC_TABLE_NAME	sdbyte*
DSQL_DESC_DATETIME_INTERVAL_PRECISION	sdint4	DSQL_DESC_TYPE	sdint2
DSQL_DESC_DISPLAY_SIZE	slength	DSQL_DESC_TYPE_NAME	sdbyte*
DSQL_DESC_FIXED_PREC_SCALE	sdint2	DSQL_DESC_UNNAMED	sdint2
DSQL_DESC_INDICATOR_PTR	slength*	DSQL_DESC_UNSIGNED	sdint2
DSQL_DESC_LABEL	sdbyte*	DSQL_DESC_UPDATABLE	sdint2
DSQL_DESC_LENGTH	ulength		

28. dpi_get_desc_rec

函数

```
DPIRETURN
dpi_get_desc_rec(
    dhdesc          dpi_desc,
    sdint2          rec_num,
    sdbyte          *name_buf,
    sdint2          name_buf_len,
    sdint2          *name_len,
    sdint2          *type,
    sdint2          *sub_type,
    slength         *length,
    sdint2          *prec,
    sdint2          *scale,
    sdint2          *nullable
);
```

功能

获取复合的描述符记录的域的当前设置或值。

参数：

1) dpi_desc

输入参数，描述符句柄。

2) rec_num

输入参数，包含所要获取域的描述符记录的索引号。此参数从 0 起始。当为 0 时，表示所设置的为书签列。此参数值必须大于等于 0 且小于等于 DSQL_DESC_COUNT 的值。

3) name_buf

输出参数，指向返回 DSQL_DESC_NAME 域值的缓冲区的指针。

4) name_buf_len

输入参数，name_buf 缓冲区的字节长度。

5) name_len

输出参数，指向返回可填充 name_buf 的数据的总长度的缓冲区的指针。如果总长度大

于或等于 name_buf_len，则 name_buf 中的数据被截断，并以 0 结尾。

6) type

输出参数，指向返回 DSQL_DESC_TYPE 域值的缓冲区的指针。

7) sub_type

输入参数，对于类型为 DSQL_DT 或者 DSQL_INTERVAL 的记录，此参数值设置 DSQL_DESC_DATETIME_INTERVAL_CODE 域的值

8) length

输出参数，指向返回 DSQL_DESC_OCTET_LENGTH 域值的缓冲区指针。

9) prec

输出参数，指向返回 DSQL_DESC_PRECISION 域值的缓冲区指针。

10) scale

输出参数，指向返回 DSQL_DESC_SCALE 域值的缓冲区指针。

11) nullable

输出参数，指向返回 DSQL_DESC_NULLABLE 域值的缓冲区指针。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

DSQL_NO_DATA

说明：

具体参见 dpi_set_desc_field 说明。

29. dpi_get_desc_field

函数

```
DPIRETURN
dpi_get_desc_field(
    dhdesc          dpi_desc,
    sdint2          rec_num,
    sdint2          field,
    dpointer        val,
    sdint4          val_len,
    sdint4          *str_len
);
```

功能

获取单个描述符记录的域的当前值或设置。

参数：

1) dpi_desc

输入参数，描述符句柄。

2) rec_num

输入参数，包含所要获取域的描述符记录的索引号。此参数从 0 起始。当为 0 时，表示所设置的为书签列。此参数值必须大于等于 0 且小于等于 DSQL_DESC_COUNT 的值。

3) field

输入参数，需要获取值的描述符域。详见函数 dpi_set_desc_field。

4) val

输出参数，指向返回描述信息的缓冲区的指针。数据类型依赖于 field。

5) val_len

输入参数，如果 val 返回的是字符串类型的值，则此参数表示 val 缓冲区的字节长度。如果 val 返回的整形数，则此参数被忽略。

6) str_len

输出参数，指向返回 val 中可填充的最大字符串的字节长度的缓冲区指针。

返回值

DSQL_SUCCESS
DSQL_SUCCES_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NO_DATA

说明：

具体参见 dpi_set_desc_field 说明。

30. dpi_bind_param

函数

```
DPIRETURN
dpi_bind_param(
    dhstmt          dpi_hstmt,
    udint2          iparam,
    sdint2          param_type,
    sdint2          ctype,
    sdint2          dtype,
    ulength         precision,
    sdint2          scale,
    dpointer        buf,
    slength         buf_len,
    slength         *ind_ptr
);
```

功能

绑定缓冲区到 SQL 语句中的参数标记。

参数：

1) dpi_hstmt

输入参数，语句句柄。

2) iparam

输入参数，参数索引号。起始为 1。

3) param_type

输入参数，参数的类型。必须是下列值之一：

- DSQL_PARAM_INPUT
- DSQL_PARAM_OUTPUT

- DSQL_PARAM_INPUT_OUTPUT
- DSQL_PARAM_INPUT_OUTPUT_STREAM
- DSQL_PARAM_OUTPUT_STREAM

4) ctype

输入参数，参数的 C 数据类型。

5) dtype

输入参数，参数所对应的数据库的 D 数据类型

6) precision

输入参数，列的宽度或者参数相应域的值。

7) scale

输入参数，列的小数位数或者参数相应域的值。

8) buf

延缓输入参数，指向存放参数数据的缓冲区的指针。

9) buf_len

输入参数，buf 缓冲的字节长度。

10) ind_ptr

延缓输入参数，指向存储参数长度的缓冲区的指针。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

绑定信息将持续作用直到再次调用 dpi_bind_param 或者 dpi_unbind_params。

31. dpi_desc_param

函数

```
DPIRETURN
dpi_desc_param(
    dhstmt          dpi_hstmt,
    udint2          iparam,
    sdint2          *sql_type,
    ulength         *prec,
    sdint2          *scale,
    sdint2          *nullable
);
```

功能

获取准备语句中参数的相关描述信息。

参数：

1) dpi_hstmt

输入参数，语句句柄。

2) iparam

输入参数，参数的索引号。起始为 1。

3) sql_type
输出参数，指向返回参数数据库 D 类型的缓冲区的指针。

4) prec
输出参数，指向返回参数相应域的值的缓冲区的指针。

5) scale
输出参数，指向返回参数相应域的值的缓冲区的指针。

6) nullable
输出参数，指向返回参数是否允许为空的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

32. dpi_exec

函数

```
DPIRETURN
dpi_exec(
    dhstmt              dpi_hstmt
);
```

功能

执行一个已经准备好的语句。如果参数中包含参数，则使用参数的当前设置或值。

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NEED_DATA

说明：

执行一个已准备好的语句，可以多次执行。如果多次执行一个查询语句，需要先调用 dpi_close_cursor 关闭当前游标，否则会报错。

33. dpi_exec_add_batch

函数

```
DPIRETURN
dpi_exec_add_batch(
    dhstmt              dpi_hstmt
```

```
);
```

功能

发送一行绑定的数据到 dpi 做数据转换及存储

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

此接口函数提供单行数据绑定插入的缓存,用户绑定参数后,调用此接口进行数据缓存,结合接口 dpi_exec_batch 可以进行数据插入。接口不支持输出参数，大字段及延迟绑定数据方式。

34. dpi_exec _batch

函数

```
DPIRETURN
dpi_exec_batch(
    dhstmt          dpi_hstmt
);
```

功能

发送已经缓存的数据到服务器进行插入操作

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

此接口发送 dpi_exec_add_batch 接口所缓存的数据到服务器进行插入操作。

35. dpi_exec_direct

函数

```
DPIRETURN
dpi_exec_direct(
    dhstmt          dpi_hstmt,
    sdbyte          *sql_txt
```

```
);
```

功能

直接执行一条语句。如果语句中包含参数，则使用参数的当前设置或值。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) sql_txt
输入参数，要执行的 sql 语句，以结束符结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NEED_DATA

说明：

单次执行某条语句。

36. dpi_unbind_params

函数

```
DPIRETURN
dpi_unbind_params(
    dhstmt              dpi_hstmt
);
```

功能

清空语句句柄上绑定的参数信息。

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

37. dpi_unbind_columns

函数

```
DPIRETURN
dpi_unbind_columns(
    dhstmt              dpi_hstmt
```

```
);
```

功能

清空语句句柄上绑定的列信息。

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

38. dpi_param_data

函数

```
DPIRETURN
dpi_param_data(
    dhstmt          dpi_hstmt,
    dpointer        *val_ptr
);
```

功能

与 dpi_put_data 一起使用，用于协助在语句执行时提供参数的数据。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) val_ptr
输出参数，指向用于返回 dpi_bind_param 绑定的参数地址的缓冲区指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NEED_DATA

说明：

当调用 dpi_exec 或者 dpi_exec_direct 时，如果返回 DSQL_NEED_DATA，则说明需要提供某个参数的数据，此时调用 dpi_param_data 获取具体参数的信息，并调用 dpi_put_data 发送相应参数的数据。再次循环上述步骤直至 dpi_param_data 返回 DSQL_SUCCESS 或 DSQL_SUCCESS_WITH_INFO。

39. dpi_prepare

函数

```
DPIRETURN
dpi_prepare(
    dhstmt          dpi_hstmt,
    sdbyte          *sql_txt
);
```

功能

准备一条用于执行的 sql 语句。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) sql_txt
输入参数，需要准备的 sql 语句，以结束符结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

40. dpi_put_data

函数

```
DPIRETURN
dpi_put_data(
    dhstmt          dpi_hstmt,
    dpointer        val,
    slength         val_len
);
```

功能

在语句执行时发送参数数据到驱动。与 dpi_param_data 配合使用。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) val
输入参数，指向包含实际数据缓冲区的指针。缓冲区数据类型必须与 dpi_bind_param 时绑定的 C 类型一致。

3) val_len
输入参数，val 中数据的实际字节长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO

DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

参见 dpi_param_data 说明。

41. dpi_number_params

函数

```
DPIRETURN
dpi_number_params(
    dhstmt              dpi_hstmt,
    sdint2              *param_cnt
);
```

功能

获取 sql 语句中包含参数的个数。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) param_cnt
输出参数，指向用于存放参数个数的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

42. dpi_set_cursor_name

函数

```
DPIRETURN
dpi_set_cursor_name(
    dhstmt              dpi_hstmt,
    sdbyte              *name,
    sdint2              name_len
);
```

功能

设置游标的游标名。如果不设置游标名，驱动将创建默认的游标名。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) name
输入参数，指向包含游标名的缓冲区的指针。

3) name_len

输入参数，name 的实际长度。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

游标名用于定位更新/删除操作。

43. dpi_get_cursor_name

函数

```
DPIRETURN
dpi_get_cursor_name(
    dhstmt          dpi_hstmt,
    sdbyte          *name,
    sdint2          buf_len,
    sdint2          *name_len
);
```

功能

获取指定语句句柄上的游标名。

参数：

1) dpi_hstmt

输入参数，语句句柄。

2) name

输出参数，指向返回游标名的缓冲区的指针。

3) buf_len

输入参数，name 缓冲区的字节长度。

4) name_len

输出参数，执行存放游标名总长度的缓冲区指针。如果游标名长度大于或者等于 buf_len，则 name 中的数据被截断并以 0 结尾。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

无

44. dpi_close_cursor

函数

```
DPIRETURN
```

```
dpi_close_cursor(
    dhstmt          dpi_hstmt
);
```

功能

关闭语句句柄上已经打开的游标，并丢弃所关联的结果集。

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

45. dpi_bind_col

函数

```
DPIRETURN
dpi_bind_col(
    dhstmt          dpi_hstmt,
    udint2          icol,
    sdint2          ctype,
    dpointer        val,
    slength         buf_len,
    slength         *ind
);
```

功能

绑定数据缓冲区到结果集中的列。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) icol
输入参数，结果集列的索引号。起始为 0。如果 icol 为 0，则绑定的为书签列。

3) ctype
输入参数，数据转换指定的 C 类型。

4) val
输出参数，指向存放实际值的缓冲区。

5) buf_len
输入参数，存放数据缓冲区的长度。

6) ind
输出参数，指向返回实际数据长度的缓冲区。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

绑定列到输出缓冲区用于在调用 dpi_fetch 或者 dpi_fetch_scroll 时检索数据。

46. dpi_number_columns

函数

```
DPIRETURN
dpi_number_columns(
    dhstmt          dpi_hstmt,
    sdint2          *col_cnt
);
```

功能

获取结果集中列的个数。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) col_cnt
输出参数，指向返回结果集列个数的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

47. dpi_desc_column

函数

```
DPIRETURN
dpi_desc_column(
    dhstmt          dpi_hstmt,
    sdint2          icol,
    sdbyte          *name,
    sdint2          buf_len,
    sdint2          *name_len,
    sdint2          *sqltype,
    ulength         *col_sz,
    sdint2          *dec_digits,
```

```
    sdint2          *nullable
);
```

功能

获取结果集列的描述信息。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) icol
输入参数，结果集列的索引号。起始为 0。如果为 0，则描述的是书签列信息。

3) name
输出参数，指向返回列名的缓冲区的指针。

4) buf_len
输入参数，name 缓冲区的字节长度。

5) name_len
输出参数，指向返回列名总长度的缓冲区的指针。如果列名长度大于或者等于 buf_len，则 name 中数据被截断并以结束符结尾。

6) sqltype
输出参数，指向返回列的数据库 DSQL 类型的缓冲区的指针。

7) col_sz
输出参数，指向返回列的宽度的缓冲区的指针。

8) dec_digits
输出参数，指向返回列的小数位数的缓冲区的指针。

9) nullable
输出参数，指向返回列是否允许为空的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

48. dpi_col_attr

函数

```
DPIRETURN
dpi_col_attr(
    dhstmt          dpi_hstmt,
    udint2          icol,
    udint2          fld_id,
    dpointer        chr_attr,
    sdint2          buf_len,
    sdint2          *chr_attr_len,
    slength         *num_attr
```

```
);
```

功能

获取结果集中列的描述信息。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) icol
输入参数，结果集列的索引号。起始索引为 0。当参数被指定为 0 时，除了 DSQL_DESC_TYPE 和 DSQL_DESC_OCTET_LENGTH 外，其他域的值为无效值。

3) fld_id
输入参数，需要获取值的描述域。

4) chr_attr
输出参数，指向返回 fld_id 域的值，此参数仅返回字符串，如果 fld_id 对应的值不是字符串，则此参数不会被使用。

5) buf_len
输入参数，chr_attr 缓冲区的字节长度。

6) chr_attr_len
输出参数，指向返回 fld_id 域可返回字符串的最大长度的缓冲区的指针。如果返回值长度大于或者等于 buf_len，则 chr_attr 中数据被截断并以 0 结尾。

7) num_attr
输出参数，指向返回整形数的缓冲区的指针。如果 fld_id 所返回的值为整形数，则此参数有意义，否则此参数不会被使用。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

详细参数含义参见 2.2.4 章节。

49. dpi_bulk_operation

函数

```
DPIRETURN
dpi_bulk_operation(
    dhstmt          dpi_hstmt,
    udint2          op
);
```

功能

执行批量插入和批量的书签操作。包括通过书签更新、删除、获取数据。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) op

输入参数，操作方式。必须是下列之一：

- DSQL_ADD
- DSQL_UPDATE_BY_BOOKMARK
- DSQL_DELETE_BY_BOOKMARK
- DSQL_FETCH_BY_BOOKMARK

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NEED_DATA

说明：

函数执行后游标位置未知，必须调用 dpi_fetch_scroll 重新定位游标。

50. dpi_fetch

函数

```
DPIRETURN
dpi_fetch(
    dhstmt          dpi_hstmt,
    ulength         *row_num
);
```

功能

获取下一个行集数据和返回所有绑定列的数据。

参数：

1) dpi_hstmt

输入参数，语句句柄。

2) row_num

输出参数，指向返回取得行数的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NO_DATA

说明：

无

51. dpi_fetch_scroll

函数

```
DPIRETURN
dpi_fetch_scroll(
```

```
    dhstmt          dpi_hstmt,
    sdint2          orient,
    slength         offset,
    ulength         *row_num
);
```

功能

获取指定行集数据和返回所有绑定列的数据。

参数：

1) dpi_hstmt

输入参数，语句句柄。

2) orient

输入参数，获取方式。必须是下列之一：

- DSQL_FETCH_NEXT
- DSQL_FETCH_PRIOR
- DSQL_FETCH_FIRST
- DSQL_FETCH_LAST
- DSQL_FETCH_ABSOLUTE
- DSQL_FETCH_RELATIVE
- DSQL_FETCH_BOOKMARK

3) offset

输入参数，获取行的偏移。此参数的意义依赖于 orient，对于 orient 为 DSQL_FETCH_RELATIVE 时有效，意义为相对于当前游标位置的偏移量。

4) row_num

输出参数，指向返回取得行数的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NO_DATA

说明：

无

52. dpi_get_data

函数

```
DPIRETURN
dpi_get_data(
    dhstmt          dpi_hstmt,
    udint2          icol,
    sdint2          ctype,
    dpointer        val,
    slength         buf_len,
    slength         *val_len
```

```
);
```

功能

获取结果集中单个列的数据。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) icol
输入参数，结果集中列的索引号。结果集索引起始为 1。如果 icol 被设置为 0，则获取的是书签列的值，且只有当书签选项启用时才有效。

3) ctype
输入参数，所要获取数据的目标 C 类型。

4) val
输出参数，指向存放数据的缓冲区的指针。

5) buf_len
输入参数，val 缓冲区的字节长度。

6) val_len
输出参数，指向存放数据长度的缓冲区的指针。如果此参数为 NULL，则不返回长度，如果数据长度为 NULL，则返回错误。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NO_DATA

说明：

dpi_get_data 可以分批获取字符类型或者二进制类型数据。

53. dpi_more_results

函数

```
DPIRETURN
dpi_more_results(
    dhstmt              dpi_hstmt
);
```

功能

确定句柄上是否包含有多个结果集。如果有，则处理这些结果集。

参数：

dpi_hstmt
输入参数，语句句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR

DSQL_INVALID_HANDLE
DSQL_NO_DATA

说明：

无

54. dpi_set_pos

函数

```
DPIRETURN
dpi_set_pos(
    dhstmt          dpi_hstmt,
    ulength         row_num,
    udint2          op,
    udint2          lock_type
);
```

功能

定位，更新结果集的数据。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) row_num
输入参数，行集中行的位置。如果 row_num 为 0，则操作作用于行集中的每一行。

3) op
输入参数，操作方式。必须是下列之一：

- DSQL_POSITION
- DSQL_REFRESH
- DSQL_UPDATE
- DSQL_DELETE
- DSQL_ADD

4) lock_type
输入参数，指定在 op 操作之后行的封锁方式。必须是下列之一：

- DSQL_LOCK_NO_CHANGE
- DSQL_LOCK_EXCLUSIVE
- DSQL_LOCK_UNLOCK

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE
DSQL_NEED_DATA

说明：

无

55. dpi_row_count

函数

```
DPIRETURN
dpi_row_count(
    dhstmt          dpi_hstmt,
    sdint8          *row_num
);
```

功能

返回 sql 语句所影响行的总数。

参数：

1) dpi_hstmt
输入参数，语句句柄。

2) row_num
输出参数，指向返回影响行数的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

56. dpi_lob_get_length

函数

```
DPIRETURN
dpi_lob_get_length(
    dhloblctr       dpi_loblctr,
    slength         *len
);
```

功能

获取 lob 句柄对应大对象的实际长度。

参数：

1) dpi_loblctr
输入参数，lob 句柄。

2) len
输出参数，指向返回大字段长度的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

CLOB 为字符长度，BLOB 为字节长度。

57. dpi_lob_read

函数

```
DPIRETURN
dpi_lob_read(
    dhloblctr       dpi_loblctr,
    ulength         start_pos,
    sdint2          ctype,
    slength         data_to_read,
    dpointer        val_buf,
    slength         buf_len,
    slength         *data_get
);
```

功能

读取 lob 句柄所对应大字段的实际数据。

参数：

1) dpi_loblctr

输入参数，lob 句柄。

2) start_pos

输入参数，起始的偏移。起始为 1。CLOB 类型此参数表示字符偏移，BLOB 类型此参数表示字节偏移。

3) ctype

输入参数，所获取数据的目标 C 类型。

4) data_to_read

输入参数，CLOB 类型表示要读取的字符数，BLOB 类型表示要读取的字节数，缺省值为 0。

5) val_buf

输出参数，指向存放获取数据的缓冲区的指针。

6) buf_len

输入参数，val_buf 缓冲区的字节长度。

7) data_get

输出参数，指向返回已获取数据的实际长度。CLOB 类型表示字符长度，BLOB 类型表示字节长度。

返回值

DSQL_SUCCESS

DSQL_SUCCESS_WITH_INFO

DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

无

58. dpi_lob_write

函数

```
DPIRETURN
dpi_lob_write(
    dhloblctr           dpi_loblctr,
    ulength             start_pos,
    sdint2              ctype,
    dpointer            val,
    ulength             bytes_to_write,
    ulength             *data_writed
);
```

功能

更新 lob 句柄所对应的大字段的数据。

参数：

1) dpi_loblctr
输入参数，lob 句柄。

2) start_pos
输入参数，起始的偏移。起始为 1。CLOB 类型此参数表示字符偏移，BLOB 类型此参数表示字节偏移。

3) ctype
输入参数，更新数据的 C 数据类型。

4) val
输入参数，指向存放更新数据缓冲区的指针。

5) bytes_to_write
输入参数，需要更新数据的字节长度。

6) data_writed
输出参数，指向返回已更新数据的长度。对于 CLOB 类型此参数表示字符长度，BLOB 类型则此参数表示字节长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

59. dpi_lob_truncate

函数

```
DPIRETURN
dpi_lob_truncate(
    dhloblctr           dpi_loblctr,
    ulength             len,
    ulength             *data_len
);
```

功能

截断 lob 句柄所对应的大字段到指定的长度。

参数：

1) dpi_loblctr
输入参数，lob 句柄。

2) len
输入参数，大字段被截断后的长度。对于 CLOB 类型此参数表示字符长度，BLOB 类型则此参数表示字节长度。

3) data_len
输出参数，大字段被截断后的实际长度。对于 CLOB 类型此参数表示字符长度，BLOB 类型则此参数表示字节长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

60. dpi_alloc_obj

函数

```
DPIRETURN
dpi_alloc_obj(
    dhcon               dpi_con,
    dhobj*               object
);
```

功能

分配复合类型句柄。

参数：

1) dpi_con
输入参数，通过此连接句柄分配新的语句句柄，新句柄运行环境从属此句柄。

2) object
输出参数， 一个存放新分配复合对象句柄数据结构的缓冲区地址。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

61. dpi_free_obj

函数

```
DPIRETURN
dpi_free_obj(
    dhobj           object
);
```

功能

释放复合对象句柄。

参数：

1) object
输入参数，复合对象句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

62. dpi_desc_obj

函数

```
DPIRETURN
dpi_desc_obj(
dhcon           dpi_con,
sdbyte*         schema,
sdbyte*         compobj_name,
dhobjdesc*      obj_desc
);
```

功能

获取复合对象的描述信息。

参数：

1) dpi_con
输入参数，连接句柄。
2) schema
输入参数，复合对象所属模式名。
3) compobj_name
输入参数，复合对象名。
4) obj_desc
输出参数，复合对象描述符句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR

DSQL_INVALID_HANDLE

说明：

无

63. dpi_desc_obj2

函数

```
DPIRETURN
dpi_desc_obj(
dhcon              dpi_con,
sdbyte*            schema,
sdbyte*            pkg_name,
sdbyte*            compobj_name,
dhobjdesc*         obj_desc
);
```

功能

获取复合对象的描述信息。该函数功能覆盖了 dpi_desc_obj()的功能，描述的复合对象包括包内的复合对象和不属于包内的复合对象，若对象不属于包内，则 pkg_name 参数置为 NULL 即可。

参数：

1) dpi_con
输入参数，连接句柄。

2) schema
输入参数，复合对象所属模式名。

3) pkg_name
输入参数，复合对象所在的包名，可以为 NULL。

4) compobj_name
输入参数，复合对象名。

5) obj_desc
输出参数，复合对象描述符句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

64. dpi_free_obj_desc

函数

```
DPIRETURN
dpi_free_obj_desc(
dhobjdesc          obj_desc
);
```

功能

释放复合对象描述符句柄。

参数：

1) obj_desc
输入参数，复合对象描述符句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

65. dpi_get_obj_attr

函数

```
DPIRETURN
dpi_get_obj_attr(
    dhobj          object,
    udint4         nth,
    udint2         attr_id,
    dpointer       buf,
    udint4         buf_len,
    slength*       len
);
```

功能

获取复合对象的属性值。

参数：

1) object
输入参数，获取属性的复合对象句柄。

2) nth
保留输入参数，暂不起作用。

3) attr_id
输入参数，需要获取的属性。

4) buf
输出参数，指向返回指定属性当前值的缓冲区的指针。

5) buf_len
输入参数，如果 val 返回的是字符串，则此参数为 val 缓冲区的长度。如果 val 返回的不是字符串，则此参数忽略。

6) len
输出参数，指向返回 val 中可提供字符串的总长度的缓冲区的指针。如果 val 为 NULL，则不返回长度，如果属性值为字符串，总长度大于等于 buf_len，则 val 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

属性	字段类型
DSQL_ATTR_OBJ_VAL_COUNT	udint4

66. dpi_get_obj_desc_attr

函数

```
DPIRETURN
dpi_get_obj_desc_attr(
    dhobjdesc       obj_desc,
    udint4          nth,
    udint2          attr_id,
    dpointer        buf,
    udint4          buf_len,
    slength*        len
);
```

功能

获取复合对象描述符上的属性值。

参数：

1) obj_desc
输入参数，复合对象描述符句柄。

2) nth
复合对象要获取域下标

3) attr_id
输入参数，需要获取的属性。

4) buf
输出参数，指向返回指定属性当前值的缓冲区的指针。

5) buf_len
输入参数，如果 val 返回的是字符串，则此参数为 val 缓冲区的长度。如果 val 返回 的不是字符串，则此参数忽略。

6) len
输出参数，指向返回 val 中可提供字符串的总长度的缓冲区的指针。如果 val 为 NULL，则不返回长度，如果属性值为字符串，总长度大于等于 buf_len，则 val 值被截断且以 0 结尾。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

属性	字段 类型
DSQL_ATTR_OBJ_TYPE	sdint2
DSQL_ATTR_OBJ_PREC	sdint2
DSQL_ATTR_OBJ_SCALE	sdint2
DSQL_ATTR_OBJ_NAME	sdbyte*
DSQL_ATTR_OBJ_DESC	dhobjdesc
DSQL_ATTR_OBJ_FIELD_COUNT	udint4
DSQL_ATTR_OBJ_SCHAME	sdbyte*

67. dpi_bind_obj_desc

函数

```
DPIRETURN
dpi_bind_obj_desc(
    dhobj           object,
    dhobjdesc       desc
);
```

功能

绑定复合对象描述符句柄。

参数：

1) object
输入参数，复合对象句柄。

2) desc
输入参数，复合对象描述符句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

68. dpi_unbind_obj_desc

函数

```
DPIRETURN
dpi_unbind_obj_desc(
    dhobj           object
);
```

功能

解除复合对象描述符句柄绑定。

参数：

1) object
输入参数，复合对象句柄。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

69. dpi_set_obj_val

函数

```
DPIRETURN
dpi_set_obj_val(
    dhobj          object,
    udint4         nth,
    udint2         ctype,
    dpointer       val,
    slength        val_len
);
```

功能

复合对象绑定值。

参数：

1) object
输入参数，复合对象句柄。
2) nth
输入参数，绑定域下标
3） ctype
输入参数，绑定的 c 数据类型
4) val
延缓输入参数，指向存放参数数据的缓冲区的指针。
5) val_len
延缓输入参数，指向存储参数长度的缓冲区的指针。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

70. dpi_get_obj_val

函数

```
DPIRETURN
dpi_get_obj_val(
    dhobj          object,
    udint4         nth,
    udint2         ctype,
    dpointer       val,
    udint4         buf_len,
    slength*       val_len
);
```

功能

获取复合对象各个域的值。

参数：

1) object
输入参数，复合对象句柄。

2) nth
复合对象域下标

3) cytpe
输入参数，数据转换指定的 C 类型

4) val
输出参数，指向存放实际值的缓冲区。

5) 输出参数，指向返回实际数据长度的缓冲区。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明：

无

2.4 编程参考

编程步骤

应用程序使用 DPI 访问数据库，可以按照以下几个基本步骤进行：

1. 调用函数 dpi_alloc_env 申请环境句柄。（通过 dpi_set_env_attr 和 dpi_get_env_attr 可以设置和获取环境句柄属性，环境句柄属性参考 2.2.1）。
2. 调用函数 dpi_alloc_con 申请连接句柄（通过 dpi_set_con_attr 和 dpi_get_con_attr 可以设置和获取连接句柄属性，连接句柄属性参考 2.2.2）。
3. 调用函数 dpi_login 连接数据库服务器。
4. 调用函数 dpi_alloc_stmt 申请语句句柄（通过 dpi_set_stmt_attr 和 dpi_get_stmt_attr

可以设置和获取语句句柄属性，语句句柄属性参考 2.2.3）。

5. 调用函数 dpi_exec_direct 直接执行 SQL 语句（也可以通过 dpi_prepare 准备语句，然后通过 dpi_bind_param 绑定参数，再通过 dpi_exec 来执行带参数的 sql 语句；还可以通过 dpi_fetch 或 dpi_fetch_scroll 来获取查询的结果集数据）。
6. 调用函数 dpi_free_stmt 释放申请的语句句柄。
7. 调用函数 dpi_logout 断开应用程序与数据源之间的连接。
8. 调用函数 dpi_free_con 释放申请的连接句柄。
9. 调用函数 dpi_free_env 释放申请的环境句柄。

程序在编译的过程中需要用到 DM 的头文件 DPI.h、DPIext.h、DPItypes.h，在连接阶段需要用到 dmdpi.lib 这个库文件，在执行阶段需要用到动态库 dmdpi.dll 以及 dmcalc.dll、dmcomm.dll、dmcyt.dll、dmclientlex.dll、dmcvt.dll、dmelog.dll、dmmem.dll、dmmsg.dll、dmos.dll、dmutl.dll。这几个动态库在安装 DM 时，已经被放到安装目录下。

另外，当使用 64 位的 DPI 接口时，需要添加 DM64 的宏，即在程序编译时添加“-DDM64”编译参数。

普通数据插入与查询方式的操作

下面通过一个简单的 Windows 环境下的示例来说明 DPI 普通数据插入与查询。

创建一个表，通过参数绑定以及数组绑定的方式插入数据，通过 fetch 和 fetch scroll 获取结果集并显示，以及通过将结果集输出到数组中。

```
#include "DPI.h"
#include "DPIext.h"
#include "DPItypes.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
/*********************************************************
Notes:
定义相应常量
*********************************************************/
#define ROWS   10                    //数组绑定一次插入和读取的行数
#define CHARS 80*1024                //一次读取和写入的字节数 800K
#define FLEN    500                  //文件名长度(带地址路径)
#define DM_SVR   "LOCALHOST"
#define DM_USER "SYSDBA"
#define DM_PWD   "SYSDBA"
/* 函数检查及错误信息显示 */
#define DPIRETURN_CHECK(rt, hndl_type, hndl) if(!DSQL_SUCCEEDED(rt)){dpi_err_msg_print(hndl_type, hndl);return rt;}
#define FUN_CHECK(rt) if(!DSQL_SUCCEEDED(rt)){goto END;}
/*********************************************************
Notes:
定义常用句柄和变量
*********************************************************/
dhenv           henv;       /* 环境句柄 */
dhcon           hcon;       /* 连接句柄 */
```

```
dhstmt          hstmt;          /* 语句句柄 */
dhdesc          hdesc;          /* 描述符句柄 */
dhloblctr       hloblctr;       /* lob 类型控制句柄 */
DPIRETURN       rt;             /* 函数返回值 */
/*******************************************************
Notes:
    错误信息获取打印
Param:
    hndl_type: 句柄类型
    hndl:      句柄
Return:
    无
*******************************************************/
void
dpi_err_msg_print(sdint2 hndl_type, dhandle hndl)
{
    sdint4 err_code;
    sdint2 msg_len;
    sdbyte err_msg[SDBYTE_MAX];
    /* 获取错误信息字段 */
/*  dpi_get_diag_field(hndl_type, hndl, 1, DSQL_DIAG_MESSAGE_TEXT, err_msg, sizeof(err_msg), NULL);
    printf("err_msg = %s\n", err_msg);*/
    /* 获取错误信息集合 */
    dpi_get_diag_rec(hndl_type, hndl, 1, &err_code, err_msg, sizeof(err_msg), &msg_len);
    printf("err_msg = %s, err_code = %d\n", err_msg, err_code);
}
/*******************************************************
Notes:
    连接数据库
Param:
    server:     服务器 IP
    uid:    数据库登录账号
    pwd:  数据库登录密码
Return:
    DSQL_SUCCESS  执行成功
    DSQL_ERROR    执行失败
*******************************************************/
DPIRETURN
dm_dpi_connect(sdbyte* server,   sdbyte* uid, sdbyte* pwd)
{
    /* 申请环境句柄 */
    rt = dpi_alloc_env(&henv);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_ENV, henv);
    /* 申请连接句柄 */
```

```
    rt = dpi_alloc_con(henv, &hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    /* 连接数据库服务器 */
    rt = dpi_login(hcon, server, uid, pwd);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    return DSQL_SUCCESS;
}
/************************************************
    断开数据库连接
************************************************/
DPIRETURN
dm_dpi_disconnect()
{
    /* 断开连接 */
    rt = dpi_logout(hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    /* 释放连接句柄和环境句柄 */
    rt = dpi_free_con(hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    rt = dpi_free_env(henv);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_ENV, henv);
    return DSQL_SUCCESS;
}
/************************************************
    初始化表
************************************************/
DPIRETURN
dm_init_table()
{
    /* 申请语句句柄 */
    rt = dpi_alloc_stmt(hcon, &hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
     /* 执行 sql */
     dpi_exec_direct(hstmt, "drop table dpi_demo");
    rt = dpi_exec_direct(hstmt, "create table dpi_demo(c1 int, c2 char(20), c3 varchar(50), c4 numeric(7,3), c5
timestamp(5), c6 clob, c7 blob)");
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 释放语句句柄 */
    rt = dpi_free_stmt(hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);

    printf("dm init table success\n");
    return DSQL_SUCCESS;
}
```

```
/************************************************
    通过参数绑定的方式执行 sql 语句
************************************************/
DPIRETURN
dm_insert_with_bind_param()
{
        sdint4          c1 = 0;         /*  与字段匹配的变量  */
      sdbyte            c2[10];
      sdbyte            c3[10];
      ddouble           c4;
      dpi_timestamp_t   c5;
      sdbyte            c6[18];
      sdbyte            c7[18];
      slength           c1_ind_ptr;
      slength           c2_ind_ptr;       /* 缓冲区长度  */
      slength           c3_ind_ptr;
     slength            c4_ind_ptr=0;
     slength            c5_ind_ptr=0;
      slength           c6_ind_ptr;
      slength           c7_ind_ptr;
      /* 分配语句句柄 */
      rt = dpi_alloc_stmt(hcon, &hstmt);
      DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
      /* 准备 sql */
  rt = dpi_prepare(hstmt, "insert into dpi_demo(c1,c2,c3,c4,c5,c6,c7) values(?,?,?,?,?,?,?)");
      DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
      /* 字段变量赋值  */
     c1 = 201410;
      memcpy(c2, "abcde", 5);
      memcpy(c3, "abcdefghi", 9);
      c4          = 0.009;
      c5.year     = 2011;
      c5.month    = 3;
      c5.day      = 1;
      c5.hour     = 11;
      c5.minute   = 45;
      c5.second   = 50;
      c5.fraction   = 900;
      memcpy(c6, "adfadsfetre2345ert", 18);
      memcpy(c7, "1234567890abcdef12", 18);
      c1_ind_ptr=sizeof(c1);
      c2_ind_ptr = 5;          /* 获取缓冲区长度  */
      c3_ind_ptr = 9;
      c4_ind_ptr=sizeof(c4);
```

```
    c5_ind_ptr=sizeof(c5);
    c6_ind_ptr= 18;
    c7_ind_ptr= 18;
     /* 绑定参数  */
     rt = dpi_bind_param(hstmt, 1, DSQL_PARAM_INPUT, DSQL_C_SLONG, DSQL_INT, sizeof(c1), 0,
&c1, sizeof(c1), &c1_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 2, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CHAR, sizeof(c2), 0,
c2, sizeof(c2), &c2_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 3, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_VARCHAR, sizeof(c3),
0, c3, sizeof(c3), &c3_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 4, DSQL_PARAM_INPUT, DSQL_C_DOUBLE, DSQL_DOUBLE, sizeof(c4),
0, &c4, sizeof(c4), &c4_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 5, DSQL_PARAM_INPUT, DSQL_C_TIMESTAMP, DSQL_TIMESTAMP,
sizeof(c5), 0, &c5, sizeof(c5), &c5_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 6, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CLOB, sizeof(c6), 0,
c6, sizeof(c6), &c6_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_param(hstmt, 7, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_BLOB, sizeof(c7), 0,
c7, sizeof(c7), &c7_ind_ptr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 执行 Dsql */
    rt = dpi_exec(hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 释放语句句柄  */
    rt = dpi_free_stmt(hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    printf("dm insert with bind param success\n");
    return DSQL_SUCCESS;
}
/************************************************
    通过参数绑定数组的方式执行 sql 语句
************************************************/
DPIRETURN
dm_insert_with_bind_array()
{
    sdint4          c1[ROWS];           /*  定义字段相应的变量  */
    sdbyte          c2[ROWS][10];
    sdbyte          c3[ROWS][10];
    ddouble         c4[ROWS];
```

```
    dpi_timestamp_t        c5[ROWS];
    sdbyte                 c6[ROWS][18];
    sdbyte                 c7[ROWS][18];
    slength          c1_ind_ptr[ROWS];          /* 缓冲区长度  */
    slength          c2_ind_ptr[ROWS];          /* 缓冲区长度  */
    slength          c3_ind_ptr[ROWS];          /* 缓冲区长度  */
    slength          c4_ind_ptr[ROWS];          /* 缓冲区长度  */
    slength          c5_ind_ptr[ROWS];          /* 缓冲区长度  */
    slength          c6_ind_ptr[ROWS];
    slength          c7_ind_ptr[ROWS];
    int              i;
    int              i_array_rows = ROWS;
    /* 分配语句句柄  */
    rt = dpi_alloc_stmt(hcon, &hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    /* 设置语句句柄属性  */
    rt = dpi_set_stmt_attr(hstmt, DSQL_ATTR_PARAMSET_SIZE, (dpointer)i_array_rows,
sizeof(i_array_rows));
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 准备 sql */
    rt = dpi_prepare(hstmt, "insert into dpi_demo(c1,c2,c3,c4,c5,c6,c7)  values(?,?,?,?,?,?,?)");
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 赋值  */
    for (i=0; i<i_array_rows; i++)
    {
        c1[i]          = i+10;
        memcpy(c2[i], "abcde", 5);
        memcpy(c3[i], "abcdefghi", 9);
        c4[i]          = 0.00901;
        c5[i].year     = 2011;
        c5[i].month    = 3;
        c5[i].day      = 1;
        c5[i].hour     = 11;
        c5[i].minute   = 45;
        c5[i].second   = 50;
        c5[i].fraction = 900;
        memcpy(c6[i], "adfadsfetre2345ert", 18);
        memcpy(c7[i], "1234567890abcdef12", 18);
        c1_ind_ptr[i] = sizeof(c1[i]);
        c2_ind_ptr[i] = 5;   /* 获取缓冲区长度 */
        c3_ind_ptr[i] = 9;
        c4_ind_ptr[i] = sizeof(c4[i]);
        c5_ind_ptr[i] = sizeof(c5[i]);
        c6_ind_ptr[i] = 18;
```

```
            c7_ind_ptr[i] = 18;
        }
        /* 绑定参数  */
        rt = dpi_bind_param(hstmt, 1, DSQL_PARAM_INPUT, DSQL_C_SLONG, DSQL_INT, sizeof(c1[0]), 0,
&c1[0], sizeof(c1[0]), &c1_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 2, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CHAR,
sizeof(c2[0]), 0, c2[0], sizeof(c2[0]), &c2_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 3, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_VARCHAR,
         sizeof(c3[0]), 0, c3[0], sizeof(c3[0]), &c3_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 4, DSQL_PARAM_INPUT, DSQL_C_DOUBLE, DSQL_DOUBLE,
sizeof(c4[0]), 0, &c4[0], sizeof(c4[0]), (slength*)&c4_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 5, DSQL_PARAM_INPUT, DSQL_C_TIMESTAMP,
DSQL_TIMESTAMP, sizeof(c5[0]), 0, &c5[0], sizeof(c5[0]), (slength*)&c5_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 6, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CLOB,
sizeof(c6[0]), 0, c6[0], sizeof(c6[0]), &c6_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        rt = dpi_bind_param(hstmt, 7, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_BLOB,
sizeof(c7[0]), 0, c7[0], sizeof(c7[0]), &c7_ind_ptr[0]);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        /* 执行 Dsql */
        rt = dpi_exec(hstmt);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        /* 释放语句句柄  */
        rt = dpi_free_stmt(hstmt);
        DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
        printf("dm insert with bind array success\n");
        return DSQL_SUCCESS;
}
/************************************************
    fetch 获取结果集
************************************************/
DPIRETURN
dm_select_with_fetch()
{
    sdint4                  c1 = 0;                 /* 与字段匹配的变量，用于获取字段值 */
    sdbyte                  c2[20];
    sdbyte                  c3[50];
    ddouble                 c4;
    dpi_timestamp_t         c5;
```

```
    sdbyte          c6[50];
    sdbyte          c7[FLEN];
    slength             c1_ind = 0;     /* 缓冲区 */
    slength             c2_ind = 0;
    slength             c3_ind = 0;
    slength             c4_ind = 0;
    slength             c5_ind = 0;
    slength             c6_ind = 0;
    slength             c7_ind = 0;
    ulength             row_num;        /* 行数 */
    sdint4              dataflag = 0;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 执行 sql 语句 */
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo"),
DSQL_HANDLE_STMT, hstmt);
    /* 绑定输出列 */
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1, sizeof(c1), &c1_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2, sizeof(c2), &c2_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3, sizeof(c3), &c3_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4, sizeof(c4), &c4_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5, sizeof(c5), &c5_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6, sizeof(c6), &c6_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 7, DSQL_C_NCHAR, &c7, sizeof(c7), &c7_ind),
DSQL_HANDLE_STMT, hstmt);
    printf("dm_select_with_fetch.....\n");
    printf("----------------------------------------------------------------\n");
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
        printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
        printf("c6 = %s, c7 = %s\n",c6, c7);
        dataflag = 1;
    }
    printf("----------------------------------------------------------------\n");
    if (!dataflag)
    {
```

```
        printf("dm no data\n");
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/************************************************
    使用参数绑定后再 fetch 获取结果集
************************************************/
DPIRETURN
dm_select_with_fetch_with_param()
{
    sdint4          c1 = 0;          /* 与字段匹配的变量，用于获取字段值 */
    slength          c1_param_ind = 0;
    sdbyte          c2[20];
    sdbyte          c3[50];
    ddouble             c4;
    dpi_timestamp_t     c5;
    sdbyte          c6[50];
    sdbyte          c7[FLEN];
    slength             c1_ind = 0;     /* 缓冲区 */
    slength             c2_ind = 0;
    slength             c3_ind = 0;
    slength             c4_ind = 0;
    slength             c5_ind = 0;
    slength             c6_ind = 0;
    slength             c7_ind = 0;
    ulength             row_num;        /* 行数 */
    sdint4           dataflag = 0;
    c1 = 10;    //读取 c1=10 的数据
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 准备 sql */
    DPIRETURN_CHECK(dpi_prepare(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo where c1 = ?"),
DSQL_HANDLE_STMT, hstmt);
    /* 绑定参数 */
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 1, DSQL_PARAM_INPUT, DSQL_C_STINYINT,
DSQL_INT, sizeof(c1), 0, &c1, sizeof(c1), &c1_param_ind), DSQL_HANDLE_STMT, hstmt);
    /* 执行 sql */
    DPIRETURN_CHECK(dpi_exec(hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 绑定输出列 */
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1, sizeof(c1), &c1_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2, sizeof(c2), &c2_ind),
```

```
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3, sizeof(c3), &c3_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4, sizeof(c4), &c4_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5, sizeof(c5), &c5_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6, sizeof(c6), &c6_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 7, DSQL_C_NCHAR, &c7, sizeof(c7), &c7_ind),
DSQL_HANDLE_STMT, hstmt);
    /* 打印输出信息 */
    printf("dm_select_with_fetch_with_param.....\n");
    printf("-------------------------------------------------------------------------\n");
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
        printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
        printf("c6 = %s, c7 = %s\n",c6, c7);
        dataflag = 1;
    }
    printf("-------------------------------------------------------------------------\n");
    if (!dataflag)
    {
        printf("dm no data\n");
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/************************************************
    fetch 获取结果集,scroll 结果集
************************************************/
DPIRETURN
dm_select_with_fetch_scroll()
{
    sdint4          c1 = 0;          /* 与字段匹配的变量，用于获取字段值 */
    sdbyte          c2[20];
    sdbyte          c3[50];
    ddouble             c4;
    dpi_timestamp_t     c5;
    sdbyte          c6[50];
    sdbyte          c7[FLEN];
```

```
    slength                 c1_ind = 0;       /* 缓冲区 */
    slength                 c2_ind = 0;
    slength                 c3_ind = 0;
    slength                 c4_ind = 0;
    slength                 c5_ind = 0;
    slength                 c6_ind = 0;
    slength                 c7_ind = 0;
    ulength                 row_num;          /* 行数 */
    ulength                   val = DSQL_CURSOR_DYNAMIC;
    sdint4              dataflag = 0;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 设置语句句柄属性 */
    DPIRETURN_CHECK(dpi_set_stmt_attr(hstmt, DSQL_ATTR_CURSOR_TYPE, (dpointer)val, 0),
DSQL_HANDLE_STMT, hstmt);
    /* 执行 sql */
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo"),
DSQL_HANDLE_STMT, hstmt);
    /* 绑定输出列 */
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1, sizeof(c1), &c1_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2, sizeof(c2), &c2_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3, sizeof(c3), &c3_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4, sizeof(c4), &c4_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5, sizeof(c5), &c5_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6, sizeof(c6), &c6_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 7, DSQL_C_NCHAR, &c7, sizeof(c7), &c7_ind),
DSQL_HANDLE_STMT, hstmt);
    /* 显示输出信息 */
    printf("dm_select_with_fetch_scroll.....\n");
    printf("----------------------------------------------------------------------\n");
    while(dpi_fetch_scroll(hstmt, DSQL_FETCH_NEXT, 0, &row_num) != DSQL_NO_DATA)
    {
        printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
        printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
        printf("c6 = %s, c7 = %s\n",c6, c7);
        dataflag = 1;
    }
```

```
    if (!dataflag)
    {
        printf("dm no data\n");
        return DSQL_SUCCESS;
    }
    DPIRETURN_CHECK(dpi_fetch_scroll(hstmt, DSQL_FETCH_FIRST, 0, &row_num), DSQL_HANDLE_STMT, hstmt);
    printf("move first : 1\n");
    printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
    printf("c5 = %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
    printf("c6 = %s, c7 = %s\n",c6, c7);
    DPIRETURN_CHECK(dpi_fetch_scroll(hstmt, DSQL_FETCH_LAST, 0, &row_num), DSQL_HANDLE_STMT, hstmt);
    printf("move last : 19\n");
    printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
    printf("c5 = %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
    printf("c6 = %s, c7 = %s\n",c6, c7);
    DPIRETURN_CHECK(dpi_fetch_scroll(hstmt, DSQL_FETCH_ABSOLUTE, 6, &row_num), DSQL_HANDLE_STMT, hstmt);
    printf("move absolute 6: 14\n");
    printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
    printf("c5 = %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
    printf("c6 = %s, c7 = %s\n",c6, c7);
    DPIRETURN_CHECK(dpi_fetch_scroll(hstmt, DSQL_FETCH_PRIOR, 0, &row_num), DSQL_HANDLE_STMT, hstmt);
    printf("move prior : 13\n");
    printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
    printf("c5 = %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
    printf("c6 = %s, c7 = %s\n",c6, c7);
    DPIRETURN_CHECK(dpi_fetch_scroll(hstmt, DSQL_FETCH_RELATIVE, 3, &row_num), DSQL_HANDLE_STMT, hstmt);
    printf("move relative 3: 16\n");
    printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
    printf("c5 = %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
    printf("c6 = %s, c7 = %s\n",c6, c7);
    printf("------------------------------------------------------------------\n");
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/************************************************
    fetch 获取结果集输出到数组
************************************************/
DPIRETURN
```

```
dm_select_with_fetch_array()
{
    sdint4          c1[ROWS];           /* 与字段匹配的变量，用于获取字段值 */
    sdbyte          c2[ROWS][20];
    sdbyte          c3[ROWS][50];
    ddouble             c4[ROWS];
    dpi_timestamp_t     c5[ROWS];
    sdbyte          c6[ROWS][50];
    sdbyte          c7[ROWS][FLEN];
    slength             c1_ind[ROWS];   /* 缓冲区 */
    slength             c2_ind[ROWS];
    slength             c3_ind[ROWS];
    slength             c4_ind[ROWS];
    slength             c5_ind[ROWS];
    slength             c6_ind[ROWS];
    slength             c7_ind[ROWS];
    ulength             row_num;        /* 行数 */
    ulength               i;
    ulength               i_array_rows = ROWS;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 设置语句句柄属性 */
    DPIRETURN_CHECK(dpi_set_stmt_attr(hstmt, DSQL_ATTR_ROWSET_SIZE, (dpointer)i_array_rows,
sizeof(i_array_rows)), DSQL_HANDLE_STMT, hstmt);
    /* 执行 sql */
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo"),
DSQL_HANDLE_STMT, hstmt);
    /* 绑定输出列 */
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1[0], sizeof(c1[0]), &c1_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2[0], sizeof(c2[0]), &c2_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3[0], sizeof(c3[0]), &c3_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4[0], sizeof(c4[0]), &c4_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5[0], sizeof(c5[0]), &c5_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6[0], sizeof(c6[0]), &c6_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 7, DSQL_C_NCHAR, &c7[0], sizeof(c7[0]), &c7_ind[0]),
DSQL_HANDLE_STMT, hstmt);
    /* 打印输出信息 */
    printf("dm_select_with_fetch_array.....\n");
```

```
    printf("----------------------------------------------------------------------\n");
    if (dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        row_num = row_num > ROWS ? ROWS : row_num;
        for (i=0; i<row_num; i++)
        {
            printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1[i], c2[i], c3[i], c4[i]);
            printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5[i].year,c5[i].month,c5[i].day,c5[i].hour,c5[i].minute,c5[i].second,c5[i].fraction
);
            printf("c6 = %s, c7 = %s\n",c6[i], c7[i]);
        }
    }
    else
    {
        printf("dm no data\n");
    }
    printf("----------------------------------------------------------------------\n");
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}

DPIRETURN
    dm_insert_select_complex_type_value()
    {
        dhobj           obj;            //复合对象句柄
        dhobjdesc       obj_desc;       //复合对象描述句柄
        udint4          cnt, cnt1;
        slength         len;
        sdint2          type, type1, type2;
        sdint2          prec, prec1, prec2;
        sdint2          scale, scale1, scale2;
        int             c1_data,c1_val;
        char            c2_data[21],c2_val[21];
        slength         val_len[2],data_len[2];
        /* 分配语句句柄 */
        DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
        dpi_exec_direct(hstmt, "drop table t");
        dpi_exec_direct(hstmt, "drop class cls1");
        DPIRETURN_CHECK(dpi_exec_direct(hstmt, "create class cls1 as c1 int; c2 varchar(20);
end;"),DSQL_HANDLE_STMT, hstmt);
        DPIRETURN_CHECK(dpi_exec_direct(hstmt, "create table t(c1
cls1)"),DSQL_HANDLE_STMT,hstmt);
```

```
	DPIRETURN_CHECK(dpi_desc_obj(hcon, "SYSDBA", "CLS1", &obj_desc),DSQL_HANDLE_DBC,hcon);
	DPIRETURN_CHECK(dpi_alloc_obj(hcon, &obj),DSQL_HANDLE_DBC,hcon);
	DPIRETURN_CHECK(dpi_bind_obj_desc(obj,obj_desc),DSQL_HANDLE_OBJECT,obj);
	//复合类型获取描述信息
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 0, DSQL_ATTR_OBJ_FIELD_COUNT, &cnt1, sizeof(cnt1), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("cnt is : %d\n",cnt1);
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 1, DSQL_ATTR_OBJ_TYPE, &type1, sizeof(type1), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("type1 is : %d\n",type1);
	if (type1!=DSQL_INT)
	{
		printf("type error");
	}
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 2, DSQL_ATTR_OBJ_TYPE, &type2, sizeof(type2), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("type2 is : %d\n",type2);
	if (type1!=DSQL_VARCHAR)
	{
		printf("type error");
	}
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 1, DSQL_ATTR_OBJ_PREC, &prec1, sizeof(prec1), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("prec1 is : %d\n",prec1);
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 2, DSQL_ATTR_OBJ_PREC, &prec2, sizeof(prec2), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("prec1 is : %d\n",prec2);
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 1, DSQL_ATTR_OBJ_SCALE, &scale1, sizeof(scale1), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("scale1 is : %d\n",scale1);
	DPIRETURN_CHECK(dpi_get_obj_desc_attr(obj_desc, 2, DSQL_ATTR_OBJ_SCALE, &scale2, sizeof(scale2), NULL),DSQL_HANDLE_OBJDESC,obj_desc);
	printf("scale2 is : %d\n",scale2);

	//复合类型插入
	c1_data=1;
	strcpy(c2_data,"aaa");
	data_len[0]=sizeof(c1_data);
	data_len[1]=strlen(c2_data);
	DPIRETURN_CHECK(dpi_set_obj_val(obj, 1, DSQL_C_SLONG, &c1_data, data_len[0]),DSQL_HANDLE_STMT,hstmt);
	DPIRETURN_CHECK(dpi_set_obj_val(obj, 2, DSQL_C_NCHAR, &c2_data, data_len[1]),DSQL_HANDLE_STMT,hstmt);
```

```
        /* 执行 sql */
        DPIRETURN_CHECK(dpi_prepare(hstmt,"insert into t(c1) values(?)"), DSQL_HANDLE_STMT,
hstmt);
        /* 绑定输出列 */
        len = sizeof(obj);
        DPIRETURN_CHECK(dpi_bind_param(hstmt, 1, DSQL_PARAM_INPUT, DSQL_C_CLASS,
DSQL_CLASS, 0, 0, &obj, sizeof(obj), &len), DSQL_HANDLE_STMT, hstmt);
        DPIRETURN_CHECK(dpi_exec(hstmt),DSQL_HANDLE_STMT, hstmt);
        DPIRETURN_CHECK(dpi_commit(hcon),DSQL_HANDLE_DBC,hcon);
        //复合类型查询
        DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1 from t"),DSQL_HANDLE_STMT,hstmt);
        DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_CLASS, &obj, sizeof(obj),
&len),DSQL_HANDLE_STMT,hstmt);
        DPIRETURN_CHECK(dpi_fetch(hstmt, NULL),DSQL_HANDLE_STMT,hstmt);
        DPIRETURN_CHECK(dpi_get_obj_val(obj, 1, DSQL_C_SLONG, &c1_val, sizeof(c1_val),
&val_len[0]),DSQL_HANDLE_STMT,hstmt);
        DPIRETURN_CHECK(dpi_get_obj_val(obj, 2, DSQL_C_NCHAR, c2_val, sizeof(c2_val),
&val_len[1]),DSQL_HANDLE_STMT,hstmt);
        printf("c1_val=%d,c2_val=%s\n",c1_val,c2_val);
        if (c1_val!=c1_data||strcmp(c2_val,c2_data)!=0)
        {
            printf("dpi_get_obj_val 获取结果 error");
        }
        printf("--------------------------------------------------------------------\n");
        /* 释放语句句柄 */
        DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
        return DSQL_SUCCESS;
    }
/**
 ** 入口函数
 */
DPIRETURN
main()
{
    //连接数据库
    rt = dm_dpi_connect(DM_SVR, DM_USER, DM_PWD);
    FUN_CHECK(rt);
    //初始化表
    rt = dm_init_table();
    FUN_CHECK(rt);
    //通过参数绑定的方式插入数据
    rt = dm_insert_with_bind_param();
    FUN_CHECK(rt);
    //通过数组绑定的方式插入数据
```

```
    rt = dm_insert_with_bind_array();
    FUN_CHECK(rt);
    //通过 fetch 查询得到结果集
    rt = dm_select_with_fetch();
    FUN_CHECK(rt);
    //通过参数绑定查询得到结果集
    rt = dm_select_with_fetch_with_param();
    FUN_CHECK(rt);
    //通过 fetch scroll 获取结果集
    rt = dm_select_with_fetch_scroll();
    FUN_CHECK(rt);
    //查询列绑定数组输出
    rt = dm_select_with_fetch_array();
    FUN_CHECK(rt);
   //复合类型插入查询描述信息获取示例
rt = dm_insert_select_complex_type_value();
FUN_CHECK(rt);
 //断开连接
    rt = dm_dpi_disconnect();
    FUN_CHECK(rt);
END:
    return DSQL_SUCCESS;
}
```

大字段操作

下面通过一个简单的 Windows 环境下的示例来说明 DPI 大字段操作。

创建一个表，读取文件插入到 lob 字段，从 lob 字段读取数据写入到文件，更新 lob 字段值，通过专用函数 dpi_lob_read 读取大字段数据，通过 dpi_lob_truncate 截取大字段。

```
#include "DPI.h"
#include "DPIext.h"
#include "DPItypes.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
/*******************************************************
Notes:
定义相应常量
*******************************************************/
#define ROWS   10                    //数组绑定一次插入和读取的行数
#define CHARS 80*1024                //一次读取和写入的字节数 800K
#define FLEN    500                  //文件名长度(带地址路径)
#define DM_SVR   "LOCALHOST"
#define DM_USER "SYSDBA"
#define DM_PWD   "SYSDBA"
#define IN_FILE       "F:\\hfc\\DPIdemo.rar"
```

```
#define UP_FILE        "F:\\hfc\\itt.rar"
/* 函数检查及错误信息显示 */
#define DPIRETURN_CHECK(rt, hndl_type, hndl) if(!DSQL_SUCCEEDED(rt)){dpi_err_msg_print(hndl_type, hndl);return rt;}
#define FUN_CHECK(rt) if(!DSQL_SUCCEEDED(rt)){goto END;}
/*********************************************************
Notes:
定义常用句柄和变量
*********************************************************/
dhenv               henv;       /* 环境句柄 */
dhcon               hcon;       /* 连接句柄 */
dhstmt              hstmt;              /* 语句句柄 */
dhdesc          hdesc;          /* 描述符句柄 */
dhloblctr    hloblctr;    /* lob 类型控制句柄 */
DPIRETURN     rt;               /* 函数返回值 */
/*********************************************************
Notes:
    错误信息获取打印
Param:
    hndl_type: 句柄类型
    hndl:       句柄
Return:
    无
*********************************************************/
void
dpi_err_msg_print(sdint2 hndl_type, dhandle hndl)
{
    sdint4 err_code;
    sdint2 msg_len;
    sdbyte err_msg[SDBYTE_MAX];
    /* 获取错误信息字段 */
/*  dpi_get_diag_field(hndl_type, hndl, 1, DSQL_DIAG_MESSAGE_TEXT, err_msg, sizeof(err_msg), NULL);
    printf("err_msg = %s\n", err_msg);*/
    /* 获取错误信息集合 */
    dpi_get_diag_rec(hndl_type, hndl, 1, &err_code, err_msg, sizeof(err_msg), &msg_len);
    printf("err_msg = %s, err_code = %d\n", err_msg, err_code);
}
/*********************************************************
Notes:
    连接数据库
Param:
    server:     服务器 IP
    uid:    数据库登录账号
    pwd:  数据库登录密码
```

```
Return:
    DSQL_SUCCESS  执行成功
    DSQL_ERROR    执行失败
*********************************************************/
DPIRETURN
dm_dpi_connect(sdbyte* server,   sdbyte* uid, sdbyte* pwd)
{
    /* 申请环境句柄 */
    rt = dpi_alloc_env(&henv);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_ENV, henv);
    /* 申请连接句柄 */
    rt = dpi_alloc_con(henv, &hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    /* 连接数据库服务器 */
    rt = dpi_login(hcon, server, uid, pwd);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    return DSQL_SUCCESS;
}
/************************************************
    断开数据库连接
************************************************/
DPIRETURN
dm_dpi_disconnect()
{
    /* 断开连接 */
    rt = dpi_logout(hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    /* 释放连接句柄和环境句柄 */
    rt = dpi_free_con(hcon);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    rt = dpi_free_env(henv);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_ENV, henv);

    return DSQL_SUCCESS;
}
/************************************************
    初始化表
************************************************/
DPIRETURN
dm_init_table()
{
    /* 申请语句句柄 */
    rt = dpi_alloc_stmt(hcon, &hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
```

```
    /* 执行 sql */
    dpi_exec_direct(hstmt, "drop table dpi_demo");
    rt = dpi_exec_direct(hstmt, "create table dpi_demo(c1 int, c2 char(20), c3 varchar(50), c4 numeric(7,3), c5 timestamp(5), c6 clob, c7 blob)");
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    /* 释放语句句柄 */
    rt = dpi_free_stmt(hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    printf("dm init table success\n");
    return DSQL_SUCCESS;
}
/************************************************
    从带地址路径的文件名中获取不带路径的文件名
************************************************/
sdbyte*
dm_get_file_name(char*  fdirname)
{
    char* fname;
    fname = (char*)malloc(FLEN);
    while(1)
    {
        fdirname = strchr(fdirname+1,'\\');
        if (fdirname == NULL)
        {
            break;
        }
        memcpy(fname, fdirname + 1, FLEN);
    }
    return fname;
}
/************************************************
    读取文件数据写入到 lob 字段
************************************************/
DPIRETURN
dm_read_file_to_put_data(char* fname)
{
    FILE*       pfile = NULL;
    sdbyte      tmpbuf[CHARS];
    slength             rlen = 0;
    slength             len      = 0;
    /* 打开文件 */
    pfile = fopen(fname, "rb");
    if (pfile == NULL)
    {
```

```
        printf("open %s fail\n", fname);
        return DSQL_ERROR;
    }
    /* 读取文件数据写入到缓冲区 */
    printf("==========Begin write==========\n");
    while (!feof(pfile))
    {
        len = fread(tmpbuf, sizeof(char), CHARS, pfile);
        if (len <= 0)
        {
            return DSQL_ERROR;
        }
        DPIRETURN_CHECK(dpi_put_data(hstmt, tmpbuf, len), DSQL_HANDLE_STMT, hstmt);
        rlen += len;
        printf("write %u bytes\n", rlen);
    }
    printf("==========End    write==========\n");
    fclose(pfile);
    return DSQL_SUCCESS;
}
/*************************************************
    读取数据写入到文件
*************************************************/
DPIRETURN
dm_write_file_from_get_data(char*  fname)
{
    FILE*       pfile = NULL;
    sdbyte      tmpbuf[CHARS];
    slength             val_len;
    slength             wlen = 0;
    slength             len     = 0;
    /* 打开文件 */
    pfile = fopen(fname, "wb");
    if (pfile == NULL)
    {
        printf("open %s fail\n", fname);
        return DSQL_ERROR;
    }
    /* 读取数据写入到文件 */
    printf("==========Begin read==========\n");
    while(DSQL_SUCCEEDED(dpi_get_data(hstmt, 7, DSQL_C_BINARY, tmpbuf, CHARS, &val_len)))
    {
        len = val_len > CHARS ? CHARS : val_len;
        fwrite(tmpbuf, sizeof(char), len, pfile);
```

```
        wlen += len;
        printf("read %u bytes\n", wlen);
    }
    printf("==========End    read==========\n");
    fclose(pfile);
    return DSQL_SUCCESS;
}
/************************************************
    读取文件插入到 blob 字段
************************************************/
DPIRETURN
dm_insert_with_file()
{
    sdint4          c1;             /* 定义字段相应的变量 */
    sdbyte          c2[10];
    sdbyte          c3[10];
    ddouble         c4;
    dpi_timestamp_t c5;
    sdbyte          c6[FLEN];
    sdint4          c7;
    slength         c2_ind_ptr;     /* 缓冲区长度 */
    slength         c3_ind_ptr;
    slength         c6_ind_ptr;
    dpointer        c7_val_ptr;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    /* 准备 sql */
    DPIRETURN_CHECK(dpi_prepare(hstmt, "insert into dpi_demo(c1,c2,c3,c4,c5,c6,c7)
values(?,?,?,?,?,?,?)"), DSQL_HANDLE_STMT, hstmt);
    /* 赋值 */
    c1              = 1;
    memcpy(c2, "abcde", 10);
    memcpy(c3, "abcdefghi", 10);
    c4              = 1000.001;
    c5.year         = 2011;
    c5.month    = 3;
    c5.day          = 1;
    c5.hour         = 11;
    c5.minute   = 45;
    c5.second   = 50;
    c5.fraction = 900;
    memcpy(c6, dm_get_file_name(IN_FILE), FLEN);
    c7              = DSQL_DATA_AT_EXEC;
    c2_ind_ptr = sizeof(c2);    /* 获取缓冲区长度 */
```

```
    c3_ind_ptr = sizeof(c3);
    c6_ind_ptr = sizeof(c6);
    /* 绑定参数 */
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 1, DSQL_PARAM_INPUT, DSQL_C_SLONG, DSQL_INT, sizeof(c1), 0, &c1, sizeof(c1), &c1), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 2, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CHAR, sizeof(c2), 0, &c2, sizeof(c2), &c2_ind_ptr), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 3, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_VARCHAR, sizeof(c3), 0, &c3, sizeof(c3), &c3_ind_ptr), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 4, DSQL_PARAM_INPUT, DSQL_C_DOUBLE, DSQL_DOUBLE, sizeof(c4), 0, &c4, sizeof(c4), (slength*)&c4), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 5, DSQL_PARAM_INPUT, DSQL_C_TIMESTAMP, DSQL_TIMESTAMP, sizeof(c5), 0, &c5, sizeof(c5), (slength*)&c5), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 6, DSQL_PARAM_INPUT, DSQL_C_NCHAR, DSQL_CLOB, sizeof(c6), 0, &c6, sizeof(c6), &c6_ind_ptr), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_param(hstmt, 7, DSQL_PARAM_INPUT, DSQL_C_BINARY, DSQL_BLOB, sizeof(c7), 0, &c7, sizeof(c7), &c7), DSQL_HANDLE_STMT, hstmt);
    /* 执行 sql */
    if (dpi_exec(hstmt) == DSQL_NEED_DATA)
    {
        if (dpi_param_data(hstmt, &c7_val_ptr) == DSQL_NEED_DATA)  /* 绑定数据 */
        {
            if (!DSQL_SUCCEEDED(dm_read_file_to_put_data(IN_FILE))) return DSQL_ERROR;  /* 读取文件数据存入到字段 */
        }
        DPIRETURN_CHECK(dpi_param_data(hstmt, &c7_val_ptr), DSQL_HANDLE_STMT, hstmt); /* 绑定数据 */
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
   printf("dm insert with file success\n");
    return DSQL_SUCCESS;
}
/************************************************
    查询，并将 blob 字段中插入的文件写入到新的文件中
************************************************/
DPIRETURN
dm_select_with_file()
{
    sdint4          c1 = 0;          /* 与字段匹配的变量，用于获取字段值 */
    sdbyte          c2[20];
    sdbyte          c3[50];
    ddouble             c4;
    dpi_timestamp_t     c5;
```

```
    sdbyte          c6[50];
    sdbyte          c7[FLEN];
    slength             c1_ind = 0;     /* 缓冲区 */
    slength             c2_ind = 0;
    slength             c3_ind = 0;
    slength             c4_ind = 0;
    slength             c5_ind = 0;
    slength             c6_ind = 0;
    slength             c7_ind = 0;
    ulength             row_num;        /* 行数 */
    ulength               rows = 1;
    sdint4          dataflag = 0;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo"),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1, sizeof(c1), &c1_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2, sizeof(c2), &c2_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3, sizeof(c3), &c3_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4, sizeof(c4), &c4_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5, sizeof(c5), &c5_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6, sizeof(c6), &c6_ind),
DSQL_HANDLE_STMT, hstmt);
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
        printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
        printf("c6 = %s\n",c6);
        sprintf(c7, "F:\\%d_%s", rows, c6);
        printf("c7 write to %s\n",c7);
        if (!DSQL_SUCCEEDED(dm_write_file_from_get_data(c7))) return DSQL_ERROR;     /* 获取数据
写入到文件 */
        rows++;
        dataflag = 1;
    }
    if (!dataflag)
    {
        printf("dm no data\n");
```

```
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/************************************************
    截断 lob 句柄所对应的大字段到指定的长度
************************************************/
DPIRETURN
dm_blob_truncate_data(
                        )
{
    slength                 c1_ind = 0;
    slength             trun_len;
    ulength             row_num;            /* 行数 */
    rt = dpi_alloc_stmt(hcon, &hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_DBC, hcon);
    rt = dpi_alloc_lob_locator(hstmt, &hloblctr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_LOB_LOCATOR, hloblctr);
    rt = dpi_exec_direct(hstmt, "select c7 from dpi_demo");
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    rt = dpi_bind_col(hstmt, 1, DSQL_C_LOB_HANDLE, &hloblctr, sizeof(hloblctr), &c1_ind);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        printf("%d\n", CHARS);
        rt = dpi_lob_truncate(hloblctr, CHARS, &trun_len);
        printf("%d\n", trun_len);
        FUN_CHECK(rt);
    }
END:
    rt = dpi_free_lob_locator(hloblctr);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_LOB_LOCATOR, hloblctr);

    /* 释放语句句柄 */
    rt = dpi_free_stmt(hstmt);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_STMT, hstmt);
    printf("dm_blob_truncate_data success\n");
    return DSQL_SUCCESS;
}
/************************************************
    lob_read
************************************************/
DPIRETURN
```

```
dm_lob_read(char* fname)
{
    FILE*       pfile = NULL;
    sdbyte      tmpbuf[CHARS];
    slength         val_len;
    slength         wlen = 0;
    slength       len, rlen;
    ulength       pos;
    /* 打开文件 */
    pfile = fopen(fname, "wb");
    if (pfile == NULL)
    {
        printf("open %s fail\n", fname);
        return DSQL_ERROR;
    }
    //获取大字段长度
    rt = dpi_lob_get_length(hloblctr, &len);
    DPIRETURN_CHECK(rt, DSQL_HANDLE_LOB_LOCATOR, hloblctr);
    /* 读取数据写入到文件 */
    printf("==========Begin read==========\n");
    while(len > 0)
    {
        pos = wlen+1;
        rlen = (len > CHARS) ? CHARS : len;
        rt = dpi_lob_read(hloblctr, pos, DSQL_C_BINARY, rlen, tmpbuf, sizeof(tmpbuf), &val_len);
        fwrite(tmpbuf, sizeof(char), val_len, pfile);
        wlen += val_len;
        printf("read %u bytes\n", wlen);
        len = len - CHARS;
    }
    printf("==========End    read==========\n");
    fclose(pfile);
    return DSQL_SUCCESS;
}
/************************************************
    读取 lob 句柄所对应大字段的实际数据
************************************************/
DPIRETURN
dm_lob_read_to_file()
{
    sdint4          c1 = 0;         /* 与字段匹配的变量，用于获取字段值 */
    sdbyte          c2[20];
    sdbyte          c3[50];
    ddouble             c4;
```

```
    dpi_timestamp_t         c5;
    sdbyte                  c6[50];
    sdbyte                  c7[FLEN];
    slength                     c1_ind = 0;       /* 缓冲区 */
    slength                     c2_ind = 0;
    slength                     c3_ind = 0;
    slength                     c4_ind = 0;
    slength                     c5_ind = 0;
    slength                     c6_ind = 0;
    slength                     c7_ind = 0;
    ulength                     row_num;          /* 行数 */
    ulength                       rows = 1;
    sdint4                   dataflag = 0;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_alloc_lob_locator(hstmt, &hloblctr), DSQL_HANDLE_LOB_LOCATOR,
hloblctr);
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c1,c2,c3,c4,c5,c6,c7 from dpi_demo"),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_SLONG, &c1, sizeof(c1), &c1_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 2, DSQL_C_NCHAR, &c2, sizeof(c2), &c2_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 3, DSQL_C_NCHAR, &c3, sizeof(c3), &c3_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 4, DSQL_C_DOUBLE, &c4, sizeof(c4), &c4_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 5, DSQL_C_TIMESTAMP, &c5, sizeof(c5), &c5_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 6, DSQL_C_NCHAR, &c6, sizeof(c6), &c6_ind),
DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 7, DSQL_C_LOB_HANDLE, &hloblctr, sizeof(hloblctr),
&c7_ind), DSQL_HANDLE_STMT, hstmt);
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
        printf("c1 = %d, c2 = %s, c3 = %s, c4 = %f, ", c1, c2, c3, c4);
        printf("c5
= %d-%d-%d %d:%d:%d.%d\n",c5.year,c5.month,c5.day,c5.hour,c5.minute,c5.second,c5.fraction);
        printf("c6 = %s\n",c6);
        sprintf(c7, "F:\\%d_%s", rows, c6);
        printf("c7 write to %s\n",c7);
        if (!DSQL_SUCCEEDED(dm_lob_read(c7))) return DSQL_ERROR;      /* 获取数据写入到文件 */
        rows++;
        dataflag = 1;
```

```
    }
    if (!dataflag)
    {
        printf("dm no data\n");
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_lob_locator(hloblctr), DSQL_HANDLE_LOB_LOCATOR, hloblctr);
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/************************************************
    lob_read
************************************************/
DPIRETURN
dm_lob_write(char* fname)
{
    FILE*       pfile = NULL;
    sdbyte      tmpbuf[CHARS];
    slength             rlen = 0;
    slength             len      = 0;
    slength             wlen;
    ulength         pos;
    /* 打开文件 */
    pfile = fopen(fname, "rb");
    if (pfile == NULL)
    {
        printf("open %s fail\n", fname);
        return DSQL_ERROR;
    }
    /* 读取文件数据写入到缓冲区 */
    printf("==========Begin write==========\n");
    while (!feof(pfile))
    {
        len = fread(tmpbuf, sizeof(char), CHARS, pfile);
        if (len <= 0)
        {
            return DSQL_ERROR;
        }
        pos = rlen+1;
        DPIRETURN_CHECK(dpi_lob_write(hloblctr, pos, DSQL_C_BINARY, tmpbuf, len, &wlen),
DSQL_HANDLE_LOB_LOCATOR, hloblctr);
        rlen += len;
        printf("write %u bytes\n", rlen);
    }
```

```
    printf("==========End    write==========\n");
    fclose(pfile);
    return DSQL_SUCCESS;
}
/************************************************
    读取文件更新到 lob 字段
************************************************/
DPIRETURN
dm_lob_write_to_update()
{
    slength                 c1_ind = 0;     /* 缓冲区 */
    ulength                 row_num;        /* 行数 */
    ulength                  rows = 1;
    sdint4              dataflag = 0;
    /* 分配语句句柄 */
    DPIRETURN_CHECK(dpi_alloc_stmt(hcon, &hstmt), DSQL_HANDLE_STMT, hstmt);
    DPIRETURN_CHECK(dpi_alloc_lob_locator(hstmt, &hloblctr), DSQL_HANDLE_LOB_LOCATOR,
hloblctr);
    DPIRETURN_CHECK(dpi_exec_direct(hstmt, "select c7 from dpi_demo"), DSQL_HANDLE_STMT,
hstmt);
    DPIRETURN_CHECK(dpi_bind_col(hstmt, 1, DSQL_C_LOB_HANDLE, &hloblctr, sizeof(hloblctr),
&c1_ind), DSQL_HANDLE_STMT, hstmt);
    while(dpi_fetch(hstmt, &row_num) != DSQL_NO_DATA)
    {
     printf("%s update to c7\n",UP_FILE);
     if (!DSQL_SUCCEEDED(dm_lob_write(UP_FILE))) return DSQL_ERROR;     /* 获取数据写入到文件
*/
     rows++;
     dataflag = 1;
    }
    if (!dataflag)
    {
        printf("dm no data\n");
    }
    /* 释放语句句柄 */
    DPIRETURN_CHECK(dpi_free_lob_locator(hloblctr), DSQL_HANDLE_LOB_LOCATOR, hloblctr);
    DPIRETURN_CHECK(dpi_free_stmt(hstmt), DSQL_HANDLE_STMT, hstmt);
    return DSQL_SUCCESS;
}
/**
 ** 入口函数
 */
DPIRETURN
main()
```

```
{
    //连接数据
    rt = dm_dpi_connect(DM_SVR, DM_USER, DM_PWD);
    FUN_CHECK(rt);
    //初始化表
    rt = dm_init_table();
    FUN_CHECK(rt);
    //读取文件插入到 lob 字段（通过 dpi_put_data 函数写数据）
    rt = dm_insert_with_file();
    FUN_CHECK(rt);
    //从 lob 字段读取数据写入文件（通过 dpi_get_data 获取数据）
    rt = dm_select_with_file();
    FUN_CHECK(rt);
     //更新大字段（通过 dpi_lob_write 更新大字段数据）
    rt = dm_lob_write_to_update();
    FUN_CHECK(rt);
     //获取大字段值（通过 dpi_lob_read 读取大字段数据）
    rt = dm_lob_read_to_file();
    FUN_CHECK(rt);
     //截取大字段数据
    rt = dm_blob_truncate_data();
    FUN_CHECK(rt);
    //断开连接
    rt = dm_dpi_disconnect();
    FUN_CHECK(rt);
END:
    return   rt;
}
```

2.5 数据捕获

数据捕获，用于捕获数据库中的表和视图的数据变化。用户根据应用的要求可以定制，用户可以监控表或视图的全部列，也可以监控表或视图的部分列，亦可以监控符合特定条件的列。另外，增强其可读性，在存储捕获的列的信息的时候，可以对这些列执行表达式操作，如把 binary 存储成字符串。

主要功能介绍如下：

1. 支持对表的数据捕获

- 支持对 DM 所有数据类型的捕获；
- 能够捕获指定的表的数据变化；
- 能够指定只捕获表中某些字段的数据变化；
- 能够捕获符合一定 WHERE 条件的数据变化（包括原来不符合条件，经修改后变成符合条件的记录；原来符合条件，经修改后变成不符合条件的记录）；
- 能够对捕获的字段进行计算或者函数转换，最终提供的是计算或转换后的值（视图不支持）；
- 能够给出变化数据记录的主键值（有主键时）或 ROWID 值；

2. 支持对视图的数据捕获

- 能够捕获指定的视图的数据变化；
- 视图的来源能够支持同一用户下多表联合，不同用户下多表联合；
- 支持除了大字段以外的，所有 DM 数据类型的捕获；

2.5.1 数据类型

1. CPT_COL_DEF_STRUCT

定义：

```
STRUCT CPT_COL_DEF_STRUCT
{
SDBYTE   NAME[129];
SDBYTE   COL_DEFINE[129];
SDBYTE   EXPR[1024];
};

TYPEDEF STRUCT CPT_COL_DEF_STRUCT CPT_COL_DEF_T;
```

功能说明：

指定捕获的字段转成其他的类型进行存储，计算 expr 的值，并存储成 col_define 的类型。

参数说明：

NAME：字段名称，必须与捕获对象字段一致。

COL_DEFINE：字段类型，该类型只有在表达式不为空时指定才有效，在指定 COL_DEFINE 时，要指定完整的定义，如 INT，VARCHAR(200)，NUMERIC(10,2)。视图指定计算表达式时，该参数无效。

EXPR：字段表达式，如果为空串，则不做计算。视图指定表达时，允许用户指定各种表达式，如 C1+ 1，或者 CAST 计算函数等。

2. CPT_ERROR_INFO_STRUCT

定义：

```
STRUCT CPT_ERROR_INFO_STRUCT
{
SDINT4    ERROR_NO;
SDBYTE   ERROR_INFO[DM_MAX_CPT_ERROR_INFO_LEN];
};

TYPEDEF STRUCT CPT_ERROR_INFO_STRUCT CPT_ERROR_INFO_T;
```

功能说明：

指定错误码。

参数说明：

ERROR_NO：发生错误返回的 CODE 码。

ERROR_INFO：发生错误返回的信息。

2.5.2 相关方法

1. DM_CREATE_CHANGE_DATA_CAPTURE

定义：

```
DMBOOL   DM_CREATE_CHANGE_DATA_CAPTURE(
#IF CPT_WITH_API
DHCON          CON_HDBC,
#ELSE
VOID *PIDAO_CONNECTION,
```

```
#ENDIF
SDBYTE*         SCHNAME,
SDBYTE*         TVNAME,
SDBYTE*         WHERES,
CPT_COL_DEF_T*   COL_DEFS,
UDINT2          N_COLS,
SDBYTE*         CDC_ID,
UDINT4          CDC_ID_LEN,
SDBYTE*         CDC_BLOB_ID,
UDINT4          CDC_BLOB_ID_LEN,
CPT_ERROR_INFO_T* ERROR_INFO
);
```

功能说明：

打开数据捕获的 DPI 接口。

参数说明：

CON_HDBC：输入参数，数据库连接句柄。

PIDAO_CONNECTION：输入参数，IDAO_CONNECTION 接口指针。

SCHNAME：输入参数，捕获对象所属模式，不允许使用带有下划线的模式名，否则会出现错误。

TVNAME：输入参数，捕获表/视图名。

WHERES：输入参数，捕获条件，条件中需明确指定新值或旧值作为条件，若没有条件，则设为空串””。如何使用 WHERE 条件？视图捕获，允许指定 WHERE 条件，但是不允许指定新值旧值的指定，视图中没有没有这些概念。举例如下：

```
例 1：监控 C1 列中，为 20 的数据变化（新值或者旧值为 20）；
      WHERE 为：’C1 = 20’；
例 2：监控 C2 列中，大于 20 同时小于 60 的值（新值旧值在区间(20,60)）；
      WHERE 为：’C2 > 20 AND C2 < 60’；
例 3：监控 C1 列中，旧值>100 的数据变化并且 C2 为零的数据变化；
      WHERE 为：’OLD.C1 > 100 AND C2 = 0’；
```

COL_DEFS：输入参数，字段描述。

N_COLS：输入参数，捕获字段数。n_cols 和 col_defs 必须一致，否则可能发生不可预知的错误

CDC_ID：输出参数，捕获请求标识，对应变化表名。

CDC_ID_LEN：输入参数，标识缓冲区长度，建议需大于等于 129。

CDC_BLOB_ID：输出参数，捕获请求标识，对应变化大字段表表名。

CDC_BLOB_ID_LEN：输入参数，大字段标识缓冲区长度，建议需大于等于 129。

ERROR_INFO：输出参数，操作产生错误的原因。

2. DM_DROP_CHANGE_DATA_CAPTURE

定义：

```
DMBOOL   DM_DROP_CHANGE_DATA_CAPTURE(
#IF CPT_WITH_API
DHCON           CON_HDBC,
#ELSE
VOID *    PIDAO_CONNECTION,
#ENDIF
SDBYTE*         CDC_ID,
CPT_ERROR_INFO_T*   ERROR_INFO
);
```

功能说明：

关闭数据捕获的 DPI 接口。

参数说明：

CON_HDBC：输入参数，数据库连接句柄；

PIDAO_CONNECTION：输入参数，IDAO_CONNECTION 接口指针；

CDC_ID： 输入参数，捕获表的名字，该表用于记录捕获的数据；

ERROR_INFO：输出参数，发生错误返回的错误信息。

3. DM_CPT_SET_LOCAL_CODE

定义：

```
VOID DM_CPT_SET_LOCAL_CODE(
    SDINT4      CODE_ID,
    SDINT4      LANG_ID
);
```

功能说明：

修改数据捕获的语言信息。

参数说明：

CODE_ID：输入参数，编码类型，暂时无效；

LANG_ID：输入参数，语言类型，0 表示中文，1 表示英文。

2.5.3 数据信息搜集表

针对每个请求对象 T（表或者视图），创建一个对应的数据捕获表 CDC_T_ID，如果 T 的捕获字段包含大字段，还需要建立捕获从表 CDC_ BLOB_T_ID。

(1) CDC_T_ID

定义：

```
CREATE TABLE CDC_T_ID (
    OP_SEQ              BIGINT,
    OP_ROWID            BINARY(8),
    OPTYPE              CHAR(1),
    HAS_BLOB            CHAR(1),
    OP_TIME             TIMESTAMP,
    T 表的主键值或 ROW_ID       BINARY(8),
    …
    OP_ERROR        VARCHAR(250)
);
```

参数说明：

OP_SEQ：操作顺序；

OP_ROWID：不管有没有主键值，都把 rowid 添加上；

OP_TYPE：操作类型：'I'/'D'/'U'/'T'/'F'/'D'/'A'，分别表示：

I: 插入一条符合条件的记录；

U: 更新一条符合条件的记录为符合条件的记录；

T: 更新一条不符合条件的记录为符合条件的记录；

F: 更新一条符合条件的记录为不符合条件的记录；

D: 删除一条符合条件的记录；

A: 数据采集表被修改

HAS_BLOB：是否有大字段：'Y'/'N'。

OP_TIME：修改的时间；

T 表的主键值或 ROW_ID：显示表的主键值，如果没有主键则显示 ROW_ID。

OP_ERROR：错误码

…：表示监控的字段；比如监控 C1，C2。则为 C1，C2 的定义，如：

```
C1          INT，
C2          VARCHAR(20);
```

(2) CDC_ BLOB_T_ID

定义：

```
CREATE TABLE CDC_T_BLOB_ID(
    OP_ROWID            BINARY(8),
    COL_NAME            VARCHAR(128),
    OP_TIME             TIMESTAMP,
    T 表的主键值或 ROWID$       BIGINT
);
```

参数说明：

OP_SEQ ：操作顺序；

OP_ROWID：不管有没有主键值，都把 rowid 添加上；

COL_NAME：大字段名；

OP_TIME：修改的时间；

T 表的主键值或 ROWID$：显示表的主键值，如果没有主键则显示 rowid。

2.5.4 举例说明

第一步，初始化数据捕获环境。

```
SP_INIT_CPT_SYS(1) ;
```

第二步，创建要监控的表对象 T1。

```
CREATE TABLE T1(C1 INT,C2 BLOB, C3 CHAR(10),C4 SMALLINT,PRIMARY KEY(C1));
INSERT INTO T1 VALUES(1, 'A' , 'AB', 11);
COMMIT;
```

第二步，在具体代码中调用 DPI 接口。通过调用数据捕获 DPI 接口，指定对具体对象、具体列的监控。

下面编写一段代码，通过调用数据捕获 DPI 接口，来实现对表 T1 中，所有列的监控。

```
#include "DPIext.h"
#include "DPI.h"
#include "DPItypes.h"
#include "dmcpt_dll.h"

dhenv    henv;          /*环境句柄*/
dhcon    hdbc;          /*连接句柄*/

dhcon get_hdbc(dhcon hdbc,dhenv  henv)
{
    /*创建 DPI 运行环境*/
    //dpi_init();
    /*申请一个环境句柄*/
    dpi_alloc_env(&henv);
    /*申请一个连接句柄*/
    dpi_alloc_con(henv,&hdbc);
    /*设置连接端口*/
    dpi_set_con_attr(hdbc, DSQL_ATTR_LOGIN_PORT,  5236,  0);
    /*连接到本地服务器*/
    if (!DSQL_SUCCEEDED(dpi_login(hdbc,"192.168.0.38","SYSDBA","SYSDBA")))
    {
        printf("connect failed!\n");
        exit(-1);
    }
    return hdbc;
}
```

```
void free_hdbc(dhcon hdbc,dhenv    henv)
{
      /*断开与数据源之间的连接*/
      dpi_logout(hdbc);
      /*释放连接句柄*/
      dpi_free_con(hdbc);
      /*释放环境句柄*/
      dpi_free_env(henv);
      }

void test_cptview_case1()
{
      cpt_col_def_t* col_def;
      cpt_col_def_t* temp;
      char cdc_id[130]  = "NULL";
      char cdc_blob_id[130]="NULL";
      cpt_error_info_t        ei;
      col_def=(cpt_col_def_t*)malloc(sizeof(cpt_col_def_t)*4);
      strcpy(col_def->name,"c1");
      strcpy(col_def->col_define,"");
      strcpy(col_def->expr,"");

      temp = col_def + 1;
      strcpy(temp->name,"c2");
      strcpy(temp->col_define,"");
      strcpy(temp->expr,"");

      temp = temp + 1;
      strcpy(temp->name,"c3");
      strcpy(temp->col_define,"");
      strcpy(temp->expr,"");

      temp =    temp + 1;
      strcpy(temp->name,"c4");
      strcpy(temp->col_define,"");
      strcpy(temp->expr,"");

      hdbc=get_hdbc(hdbc,henv);
      dm_create_change_data_capture(hdbc,"SYSDBA","T1","",col_def,4,cdc_id,130,cdc_blob_id,130,
&ei);
      printf("cdc_id:%s\n",cdc_id);
      printf("cdc_blob_id:%s\n",cdc_blob_id);
      printf("%s\n",ei.error_info);
      //可以在此改变表中的数据，
      //手动在客户端查询打印出的 cdc_id 和 cdc_blob_id 表名，监控开启捕获后，T1 表中数据的
变化。
      //dm_drop_change_data_capture(hdbc,cdc_id, &ei);可以在此处关闭 DPI 接口。因为本例需要
后面查看查看捕获表，所以不关闭。
      free_hdbc(hdbc,henv);
}
void main()
{
      test_cptview_case1();
      system("pause");
}
```

第三步，在数据捕获的窗口中，查看打印出的 cdc_id 和 cdc_blob_id。

```
cdc_id:CPT_SYSDBA_T1_82118833738749
cdc_blob_id:CPT_SYSDBA_T1_82118833738749_BLOB
```

第四步，对具体对象进行操作。通过 disql 等客户端来改变表中的数据。

```
insert into t1 values(2, 'B', 'BC', 22);
insert into t1 values(3, 'C', 'CD', 33);
insert into t1 values(4, 'D', 'DE', 44);
```

第五步，查看数据捕获表。通过 disql 等客户端来查看 cdc_id 和 cdc_blob_id 表中的数据。

cdc_id 表：

```
select * from CPT_SYSDBA_T1_82118833738749;
--查询结果如下：
行号          OP_SEQ      OPTYPE  HAS_BLOB    C1        C3      C4              OP_ERROR
----------  --------    ----------  --------    ----------  --------    ----------  --------
1             2             I           Y         2         BC      22            NULL
2             3             I           Y         3         CD      33            NULL
3             4             I           Y         4         DE      44            NULL
```

cdc_blob_id 表：

```
select * from CPT_SYSDBA_T1_82118833738749_BLOB;
--查询结果如下：
行号          OP_SEQ                  COL_NAME  C1
----------  ------------------  --------  ----------  ----------  --------
1             2                       C2          2
2             3                       C2          3
3             4                       C2          4
```

第 6 步，结束的时候，关闭数据捕获的环境。

```
SP_INIT_CPT_SYS(0);
```

第 3 章 DMODBC 编程指南

本章结合 DM 数据库的特点，比较全面系统地介绍 ODBC 的基本概念以及 DM ODBC DRIVER 的使用方法，以便用户更好地使用 DM ODBC 编写应用程序。

ODBC 提供访问不同类型的数据库的途径。结构化查询语言 SQL 是一种用来访问数据库的语言。通过使用 ODBC，应用程序能够使用相同的源代码和各种各样的数据库交互。这使得开发者不需要以特殊的数据库管理系统 DBMS 为目标，或者了解不同支撑背景的数据库的详细细节，就能够开发和发布客户/服务器应用程序。

DM ODBC 3.0 遵照 Microsoft ODBC 3.0 规范设计与开发，实现了 ODBC 应用程序与 DM 的互连接口。用户可以直接调用 DM ODBC 3.0 接口函数访问 DM，也可以使用可视化编程工具如 C++ Builder、PowerBuilder 等利用 DM ODBC 3.0 访问 DM。

在 DM 客户端软件安装过程中，如果选择了安装 ODBC 驱动程序的相关选项，安装工具可完成将 DM ODBC 3.0 驱动程序复制到硬盘，并在 Windows 注册表中登记 DMODBC 驱动程序信息的工作。若使用的是拷贝版，在 Windows 系统上手动注册 odbc 驱动的方法为：创建注册文件（例如 installDmOdbc.reg），文件内容为：

```
REGEDIT4
[HKEY_LOCAL_MACHINE\Software\ODBC\ODBCINST.INI\ODBC Drivers]
"DM8 ODBC DRIVER"="Installed"
[HKEY_LOCAL_MACHINE\Software\ODBC\ODBCINST.INI\DM8 ODBC DRIVER]
"Driver"="%DM_HOME%\\bin\\dodbc.dll"    //此处修改成你的 dodbc.dll 所在目录
"Setup"="%DM_HOME%\\bin\\dodbc.dll"     //此处修改成你的 dodbc.dll 所在目录
```

要进一步使用 DM ODBC 驱动程序，请阅读本章以了解 ODBC 数据源管理方法。

3.1 数据类型

客户程序可以通过 SQLGetTypeInfo 函数来获取 DM ODBC 3.0 支持的数据类型信息。由 SQLGetTypeInfo 返回的数据类型是数据源所支持的数据类型，它们是预备用于 DDL（DataDefinitionLanguage）语句的。

调用 DM ODBC 3.0 的 SQLGetTypeInfo，返回支持的数据类型如表 3.1 所示：

表 3.1 数据类型列表

类型名	类型描述
Char(n)	固定串长度为 n 的字符串，n<=8188
Varchar(n)	最大字符串长度为 n 的可变长度字符串，n<=8188
Binary(n)	固定长度为 n 的二进制数据，n<=8188
Varbinary(n)	最大长度为 n 的可变长度二进制数据，n<=8188
Image	影像数据类型，可变长度的二进制数据，最大长度为 2G-1
Text	文本数据类型，可变长度的字符数据，最大长度为 2G-1
Bit	单个二进制位数据
Tinyint	精度为 3，刻度为 0 的有符号精确数字，取值范围-128…127
Smallint	精度为 5，刻度为 0 的有符号精确数字，取值范围-32,768…32,767
Int	精度为 10，刻度为 0 的有符号精确数字，取值范围-$2^{[31]}$…$2^{[31]}$-1

Bigint	精度为 19，刻度为 0 的有符号精确数字值，取值范围-$2^{[63]}$…$2^{[63]}$-1
Real	二进制精度为 24 的有符号近似数字值，取值范围 0 或者绝对值为：$10^{[-38]}$…$10^{[38]}$
Float	二进制精度为 53 的有符号近似数字值，取值范围 0 或者绝对值为：$10^{[-308]}$…$10^{[308]}$
Double	二进制精度为 53 的有符号近似数字值，取值范围 0 或者绝对值为：$10^{[-308]}$…$10^{[308]}$
Decimal(p,s)	精度为 p，刻度为 s 的有符号精确数字值，1≤p≤38，s≤p
Numeric(p,s)	精度为 p，刻度为 s 的有符号精确数字值，1≤p≤38，s≤p
Date	日期数据类型，年月日字段，格式与 Gregorian（罗马）日历一致
Time(p)	时间数据类型，时分秒字段，精度 p 指定了秒的精度
Timestamp(p)	时间戳数据类型，年月日时分秒字段，精度 p 指定了秒的精度
Interval year(p)	年间隔，即两个日期之间的年数字，p 为时间间隔的首项字段精度(后面简称为：首精度)
Interval month(p)	月间隔，即两个日期之间的月数字，p 为时间间隔的首精度
Interval year(p) to month	年月间隔，即两个日期之间的年月数字，p 为时间间隔的首精度
Interval day(p)	日间隔，即为两个日期/时间之间的日数字，p 为时间间隔的首精度
Interval hour(p)	时间隔，即为两个日期/时间之间的时数字，p 为时间间隔的首精度
Interval minute(p)	分间隔，即为两个日期/时间之间的分数字，p 为时间间隔的首精度
Interval second(p,q)	秒间隔，即为两个日期/时间之间的秒数字，p 为时间间隔的首精度，q 为时间间隔秒精度
Interval day(p) to hour	日时间隔，即为两个日期/时间之间的日时数字，p 为时间间隔的首精度
Interval day(p) to minute	日时分间隔，即为两个日期/时间之间的日时分数字，p 为时间间隔的首精度
Interval day(p) to second(q)	日时分秒间隔，即为两个日期/时间之间的日时分秒数字，p 为时间间隔的首精度，q 为时间间隔秒精度
Interval hour(p) to minute	时分间隔，即为两个日期/时间之间的时分数字，p 为时间间隔的首精度
Interval hour(p) to second(q)	时分秒间隔，即为两个日期/时间之间的时分秒数字，p 为时间间隔的首精度，q 为时间间隔秒精度
Interval minute(p) to second(q)	分秒间隔，即为两个日期/时间之间的分秒间隔，p 为时间间隔的首精度，q 为时间间隔秒精度

注：变长字符串的最大长度受页大小的约束，最大长度为页大小的一半且不超过 8188 个字节。

3.2 支持的函数

客户程序可以通过 SQLGetFunctions 函数来获取 DM ODBC 3.0 支持的函数信息。由 SQLGetFunctions 返回的函数列表是数据源所支持的函数。

以下按照类型分类列出了 DM ODBC 3.0 提供的函数。应用程序能够通过调用 SQLGetFunctions 来获得指定函数的支持信息。

3.2.1 连接到数据源

下面的函数用于连接到数据源：

1. SQLAllocHandle：分配环境、连接、语句或者描述符句柄；
2. SQLConnect：建立与驱动程序或者数据源的连接。访问数据源的连接句柄包含了状态、事务申明和错误信息的所有连接信息；
3. SQLDriverConnect：与 SQLConnect 相似，用来连接到驱动程序或者数据源。但它比 SQLConnect 支持数据源更多的连接信息，它提供了一个对话框来提示用户设置所有的连接信息以及系统信息表没有定义的数据源；
4. SQLBrowseConnect：支持一种交互方法来检索或者列出连接数据源所需要的属性和属性值。每次调用函数可以获取一个连接属性字符串，当检索完所有的属性值，就建立起与数据源的连接，并且返回完整的连接字符串，否则提示缺少的连接属性信息，用户根据此信息重新输入连接属性值再次调用此函数进行连接。

达梦支持的连接关键字及取值范围：

名称	说明
DRIVER	DM ODBC 驱动的名字：DM8 ODBC DRIVER
DSN	数据源名称
SERVER	目标服务器，ip 地址或者服务名
TCP_PORT	端口号
UID	用户名
PWD	密码
LANGUAGE	使用的语言信息：ENGLISH，CHINESE
SSL_PATH	加密文件路径
SSL_PWD	加密文件密码
MPP_LOGIN	MPP 登陆方式：MPP_GLOBAL，MPP_LOCAL
CHARACTER_CODE	编码信息：PG_UTF8/PG_GB18030，PG_BIG5，PG_ISO_8859_9，PG_EUC_JP，PG_EUC_KR，PG_KOI8R，PG_ISO_8859_1，PG_ISO_8859_11
FAST_INSERT	是否配置快速插入：TRUE，FALSE
RW_SEPARATE	是否配置读写分离：TRUE，FALSE
RW_SEPARATE_PERCENT	读写分离的比例：0~100
UDP_FLAG	是否使用 UDP 通信：TRUE，FALSE

3.2.2 获取驱动程序和数据源信息

下面的函数用来获取驱动程序和数据源信息：

1. SQLDataSources：能够被调用多次来获取应用程序使用的所有数据源的名字；
2. SQLDrivers：返回所有安装过的驱动程序清单，包括对它们的描述以及属性关键字；
3. SQLGetInfo：返回连接的驱动程序和数据源的元信息；
4. SQLGetFunctions：返回指定的驱动程序是否支持某个特定函数的信息；
5. SQLGetTypeInfo：返回指定的数据源支持的数据类型的信息。

3.2.3 设置或者获取驱动程序属性

下面的函数用来设置或者获取驱动程序属性：

1. SQLSetConnectAttr：设置连接属性值；
2. SQLGetConnectAttr：返回连接属性值；
3. SQLSetEnvAttr：设置环境属性值；
4. SQLGetEnvAttr：返回环境属性值；
5. SQLSetStmtAttr：设置语句属性值；
6. SQLGetStmtAttr：返回语句属性值。

3.2.4 设置或者获取描述符字段

下面的函数用来设置或者获取描述符字段：

1. SQLGetDescField：返回单个描述符字段的值；
2. SQLGetDescRec：返回当前描述符记录的多个字段的值；
3. SQLSetDescField：设置单个描述符字段的值；
4. SQLSetDescRec：设置描述符记录的多个字段。

3.2.5 准备 SQL 语句

下面的函数用来准备 SQL 语句：

1. SQLPrepare：准备要执行的 SQL 语句；
2. SQLBindParameter：在 SQL 语句中分配参数的缓冲区；
3. SQLGetCursorName：返回与语句句柄相关的游标名称；
4. SQLSetCursorName：设置与语句句柄相关的游标名称；
5. SQLSetScrollOptions：设置控制游标行为的选项。

3.2.6 提交 SQL 请求

下面的函数用来提交 SQL 请求：

1. SQLExecute：执行准备好的 SQL 语句；
2. SQLExecDirect：执行一条 SQL 语句；
3. SQLNativeSql：返回驱动程序对一条 SQL 语句的翻译；
4. SQLDescribeParam：返回对 SQL 语句中指定参数的描述；
5. SQLNumParams：返回 SQL 语句中参数的个数；
6. SQLParamData：与 SQLPutData 联合使用在运行时给参数赋值；
7. SQLPutData：在 SQL 语句运行时给部分或者全部参数赋值。

3.2.7 检索结果集及其相关信息

下面的函数用来检索结果集及其相关信息：

1. SQLRowCount：返回 INSERT、UPDATE 或者 DELETE 等语句影响的行数；
2. SQLNumResultCols：返回结果集中列的数目；
3. SQLDescribeCol：返回结果集中列的描述符记录；
4. SQLColAttribute：返回结果集中列的属性；
5. SQLBindCol：为结果集中的列分配缓冲区；
6. SQLFetch：在结果集中检索下一行元组；
7. SQLFetchScroll：返回指定的结果行；
8. SQLGetData：返回结果集中当前行某一列的值；
9. SQLSetPos：在取到的数据集中设置游标的位置。这个记录集中的数据能够刷新、更新或者删除；
10. SQLBulkOperations：执行块插入和块书签操作，其中包括根据书签更新、删除或者取数据；
11. SQLMoreResults：确定是否能够获得更多的结果集，如果能就执行下一个结果集的初始化操作；
12. SQLGetDiagField：返回一个字段值或者一个诊断数据记录；
13. SQLGetDiagRec：返回多个字段值或者一个诊断数据记录。

3.2.8 取得数据源系统表的信息

下面的函数用来取得数据源系统表的信息：

1. SQLColumnPrivileges：返回一个关于指定表的列的列表以及相关的权限信息；
2. SQLColumns：返回指定表的列信息的列表；
3. SQLForeignKeys：返回指定表的外键信息的列表；
4. SQLPrimaryKeys：返回指定表的主键信息的列表；
5. SQLProcedureColumns：返回指定存储过程的参数信息的列表；
6. SQLProcedures：返回指定数据源的存储过程信息的列表；
7. SQLSpecialColumns：返回唯一确定某一行的列的信息，或者当某一事务修改一行的时候自动更新各列的信息；
8. SQLStatistics：返回一个单表的相关统计信息和索引信息；
9. SQLTablePrivileges：返回相关各表的名称以及相关的权限信息；
10. SQLTables：返回指定数据源中表信息。

3.2.9 终止语句执行

下面的函数用来终止语句执行：

1. SQLFreeStmt：终止语句执行，关闭所有相关的游标，放弃没有提交的结果，选择释放与指定语句句柄相关的资源；
2. SQLCloseCursor：关闭一个打开的游标，放弃没有提交的结果；
3. SQLCancel：放弃执行一条 SQL 语句；
4. SQLEndTran：提交或者回滚事务。

3.2.10 中断连接

下面的函数处理中断连接的任务：

1. SQLDisconnect：关闭指定连接；
2. SQLFreeHandle：释放环境、连接、语句或者描述符句柄。

3.3 建立 ODBC 连接

3.3.1 申请环境与连接句柄

客户程序要和一个远程的服务器或数据库进行通讯，必须首先和这个服务器或数据库建立连接。下面我们将要介绍如何通过 ODBC 建立一个连接以及使用连接。

为了建立一个 ODBC 数据源连接，需要使用到环境句柄以及连接句柄。句柄有一个层次的概念，一个连接句柄总是和一个唯一的环境句柄相联系的，所有的连接句柄必须在环境句柄释放之前释放。

客户程序可以通过调用函数 SQLAllocHandle 来申请一个环境句柄，调用函数 SQLAllocHandle 时必须传入句柄选项 SQL_HANDLE_ENV，当申请环境句柄成功之后，可以在此环境句柄上申请连接句柄。申请环境句柄和连接句柄的代码示范如下：

```
#include <windows.h>
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
/* 检测返回代码是否为成功标志，当为成功标志时返回 TRUE，否则返回 FALSE */
#define RC_SUCCESSFUL(rc)    ((rc) == SQL_SUCCESS || (rc) == SQL_SUCCESS_WITH_INFO)
/* 检测返回代码是否为失败标志，当为失败标志时返回 TRUE，否则返回 FALSE */
#define RC_NOTSUCCESSFUL(rc) (!(RC_SUCCESSFUL(rc)))
HENV        henv; /* 环境句柄 */
HDBC        hdbc; /* 连接句柄 */
SQLRETURN   sret; /* 返回代码 */
void main(void)
{
/* 申请一个环境句柄          */
SQLAllocHandle(SQL_HANDLE_ENV, NULL, &henv);
/* 设置环境句柄的 ODBC 版本 */
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3,
SQL_IS_INTEGER);
/* 申请一个连接句柄          */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* 释放连接句柄              */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* 释放环境句柄              */
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

```
}
```

3.3.2 如何与数据源进行连接

ODBC 的连接是从数据源开始的，数据源是 ODBC 对一个特定的数据库的别称。为了访问由数据源提供的数据，你的程序中必须首先建立和数据源之间的连接，在环境和连接句柄正确分配之后，才能通过这些连接管理数据访问。

为了产生建立连接时必要的参数，必须完成以下的几项工作：

1. 调用 SQLAllocHandle 申请一个环境句柄；
2. 调用 SQLAllocHandle 申请一个连接句柄；
3. 创建一个数据源 DSN；
4. 一个有效的用户 ID；
5. 一个对应于这个用户 ID 的口令；
6. 其它的一些提供给驱动程序的参数信息。

连接 ODBC 数据源时，ODBC 提供三种不同的连接函数，即 SQLConnect、SQLDriverConnect 和 SQLBrowseConnect，每个函数都有不同的参数以及不同级别的一致性，如表 3.2 所示：

表 3.2 连接到数据源的 ODBC 函数

函数	ODBC 版本	一致性	主要参数
SQLConnect	1.0	核心级	hdbc，数据源，用户 ID，口令
SQLDriverConnect	1.0	1 级	hdbc，窗口句柄，输入连接字符串
SQLBrowseConnect	1.0	2 级	hdbc，输入连接字符串，输出连接字符串

SQLConnect 是连接 ODBC 数据源的最基本的方法，在所有的一致性级别上都支持。SQLConnect 函数有以下参数：连接句柄、数据源名称、数据源名称长度、用户名（用户 ID）、用户名长度、用户口令以及口令长度。返回代码有：SQL_SUCCESS，SQL_SUCCESS_WITH_INFO，SQL_ERROR 或者 SQL_INVALID_HANDLE。使用 SQLConnect 函数连接数据源的代码示范如下：

```
sret = SQLConnect(hdbc, (SQLCHAR *)"DM", SQL_NTS, (SQLCHAR *)"SYSDBA", SQL_NTS, (SQLCHAR
*)"SYSDBA", SQL_NTS);
if (RC_NOTSUCCESSFUL(sret)) {
/* 连接数据源失败! */
…进行相应的错误处理…
exit(0);
}
```

SQLDriverConnect 提供了比 SQLConnect 更灵活的方法来建立 ODBC 连接。它支持以下几种连接：要求更多连接参数的数据源，对话框提示用户输入所有的连接信息以及没有在系统信息表中定义的数据源。

SQLDriverConnect 提供以下的连接方法：

1. 用一个连接字符串建立一个连接，这个字符串包括建立连接的所有数据，如 DSN，一个或多个用户 ID 及其口令，以及其他的数据库所需要的连接信息；
2. 用一个并不完整的连接字符串来建立连接，使得 ODBC 驱动程序管理器来提示用户输入所需要的连接信息；
3. 用一个没有在系统信息表中登记的数据源建立连接，驱动程序自动提示用户输入连

接信息；

4. 用一个连接字符串建立连接，这个字符串在 DSN 配置文件中是确定的。

SQLDriverConnect 函数 fDriverCompletion 参数说明：

1. SQL_DRIVER_PROMPT：设置此选项用来显示一个对话框来提示用户输入连接信息；
2. SQL_DRIVER_COMPLETE：如果函数调用中包含了足够的信息，ODBC 就进行连接，否则弹出对话框提示用户输入连接信息，此时等同于 SQL_DRIVER_PROMPT；
3. SQL_DRIVER_COMPLETE_REQUIRED：这个参数与 SQL_DRIVER_COMPLETE 参数唯一的不同是用户不能改变由函数提供的信息；
4. SQL_DRIVER_NOPROMPT：如果函数调用时有足够的信息，ODBC 就进行连接，否则返回 SQL_ERROR。

使用 SQLDriverConnect 连接数据源的代码示范如下：

```
SQLCHAR szConnStrIn[256] = "DSN=DM;DRIVER=DM ODBC DRIVER;
UID=SYSDBA;PWD=SYSDBA;TCP_PORT=5236";
SQLCHAR szConnStrOut[256];
SQLSMALLINT cbConnStrOut;
    sret = SQLDriverConnect(hdbc, NULL, szConnStrIn, SQL_NTS, szConnStrOut, 256, &cbConnStrOut,
SQL_DRIVER_NOPROMPT);
if (RC_NOTSUCCESSFUL(sret)) {
/* 连接数据源失败! */
…进行相应的错误处理…
exit(0);
}
```

SQLBrowseConnect 函数与 SQLDriverConnect 函数相似，但是调用 SQLBrowseConnect 函数时，程序在运行时可以再形成一个连接字符串，使用这个函数可以用一个交互的方式来决定连接到数据源时所需要的一些信息。

使用 SQLBrowseConnect 函数连接数据源的代码示范如下：

```
SQLCHAR szConnStrIn[256] = "";
SQLCHAR szConnStrOut[256];
SQLSMALLINT cbConnStrOut;
strcpy(szConnStrIn, "DRIVER=DM ODBC DRIVER");
sret = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS, szConnStrOut, 256, &cbConnStrOut);
if (sret != SQL_NEED_DATA) {
/* 连接数据源失败! */
…进行相应的错误处理…
exit(0);
}
strcpy(szConnStrIn, "SERVER=127.0.0.1;TCP_PORT=5236");//TCP_PORT=5236 可选
sret = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS, szConnStrOut, 256, &cbConnStrOut);
if (sret != SQL_NEED_DATA) {
/* 连接数据源失败! */
…进行相应的错误处理…
exit(0);
}
```

```
strcpy(szConnStrIn, "UID=SYSDBA;PWD=SYSDBA;");
sret = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS, szConnStrOut, 256, &cbConnStrOut);
if (sret != SQL_SUCCESS) {
/* 连接数据源失败! */
…进行相应的错误处理…
exit(0);
}
/*连接成功*/
```

3.3.3 设置与取得连接的属性

建立连接之后，应用程序可以通过调用 SQLSetConnectAttr 函数来设置连接属性，对连接进行全方面的管理。表 3.3 列出了一些常用的连接属性。

表 3.3 常用的连接属性

属性	描述
SQL_ATTR_ACCESS_MODE	用来设置访问模式，即只读或者读写连接模式，可以用来优化并发控制策略。不支持。
SQL_ATTR_ASYNC_ENABLE	是否支持异步执行。
SQL_ATTR_AUTOCOMMIT	是否使用自动提交功能。
SQL_ATTR_CONNECTION_TIMEOUT	设定连接上的超时。
SQL_ATTR_CURRENT_CATALOG	当前连接使用的编目。
SQL_ATTR_LOGIN_TIMEOUT	设定登录超时。不支持。
SQL_ATTR_ODBC_CURSORS	设置驱动程序管理器使用游标的方式。
SQL_ATTR_PACKET_SIZE	设置网络传输包的大小。不支持。
SQL_ATTR_QUIET_MODE	使弹出对话框有效/无效。

更多的连接属性，用户可以参考《Microsoft ODBC 3.0 程序员参考手册》，在这里不做详细介绍。

应用程序可以通过调用 SQLGetConnectAttr 函数来取得当前连接的属性。

设置与取得连接属性的代码示范如下：

```
SQLINTEGER   AUTOCOMMIT_MODE;
/* 设置连接句柄属性，关闭自动提交功能 */
SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF,
SQL_IS_INTEGER);
/* 取得连接句柄属性，取得提交的模式    */
SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)&AUTOCOMMIT_MODE,
sizeof(SQLINTEGER), NULL);
```

3.3.4 断开与数据源之间的连接

如果要终止客户程序与服务器之间的连接，客户程序应当完成以下的几个操作：

1. 调用 SQLFreeHandle 释放语句句柄，关闭所有打开的游标，释放相关的语句句柄

资源。（在非自动提交模式下，需事先提交当前的事务）；

2. 调用函数 SQLDisconnect 关闭所有的连接；
3. 调用 SQLFreeHandle 释放连接句柄及其相关的资源；
4. 调用 SQLFreeHandle 释放环境句柄及其相关的资源；

一个完整的连接管理示范代码如下：

```
#include <windows.h>
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
/* 检测返回代码是否为成功标志，当为成功标志返回 TRUE，否则返回 FALSE */
#define RC_SUCCESSFUL(rc) ((rc) == SQL_SUCCESS || (rc) == SQL_SUCCESS_WITH_INFO)
/* 检测返回代码是否为失败标志，当为失败标志返回 TRUE，否则返回 FALSE */
#define RC_NOTSUCCESSFUL(rc) (!(RC_SUCCESSFUL(rc)))
HENV                henv;/* 环境句柄 */
HDBC                hdbc;/* 连接句柄 */
HSTMT               hsmt;/* 语句句柄 */
SQLRETURN           sret; /* 返回代码 */
SQLINTEGER   AUTOCOMMIT_MODE;
void main(void)
{
/* 申请一个环境句柄          */
SQLAllocHandle(SQL_HANDLE_ENV, NULL, &henv);
/* 设置环境句柄的 ODBC 版本 */
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3,
 SQL_IS_INTEGER);
/* 申请一个连接句柄          */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
sret = SQLConnect(hdbc, (SQLCHAR *)"DM", SQL_NTS, (SQLCHAR *)"SYSDBA", SQL_NTS, (SQLCHAR
*)"SYSDBA", SQL_NTS);
if (RC_NOTSUCCESSFUL(sret)) {
/* 连接数据源失败! */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
SQLFreeHandle(SQL_HANDLE_ENV, henv);
exit(0);
}
/* 设置连接句柄属性，关闭自动提交功能 */
SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)SQL_AUTOCOMMIT_OFF,
SQL_IS_INTEGER);
/* 取得连接句柄属性，取得提交的模式    */
SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER)&AUTOCOMMIT_MODE,
sizeof(SQLINTEGER), NULL);
/* 申请一个语句句柄                    */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hsmt);
…在这里可以使用语句句柄进行相应的数据库操作…
```

```
/* 释放语句句柄                    */
SQLFreeHandle(SQL_HANDLE_STMT, hsmt);
/* 提交连接上的事务                */
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
/* 断开与数据源之间的连接          */
SQLDisconnect(hdbc);
/* 释放连接句柄          */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* 释放环境句柄          */
SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
```

3.4 ODBC 应用程序编程的基本步骤

3.4.1 Windows 上创建 ODBC 数据源

在客户使用 ODBC 方法访问一个 DM 数据库服务器之前，必须先对自己的应用程序所用的 ODBC 数据源进行配置。本节将介绍如何为你的应用程序安装和配置 ODBC 数据源。

在客户机上配置 ODBC 数据源的步骤：

1. 在控制面板上访问 ODBC 构件，显示 ODBC 数据源管理器对话框，如图 3.1 所示：ODBC 数据源管理器对话框包含的标签如下：
   1) 用户 DSN：添加、删除或配置本机上的数据源，它们只可由当前用户使用；
   2) 系统 DSN：添加、删除或配置本机上的数据源，它们可由任何用户使用；
   3) 文件 DSN：添加、删除或配置在分离文件中的数据源。这些文件可以被安装了同样数据库驱动器的用户共享；
   4) 驱动程序：列出了安装在客户机上的数据库驱动器；
   5) 跟踪：用于测试你的数据库应用程序。它跟踪客户机和数据库服务器之间的 ODBC API 的调用；
   6) 连接池：允许不同的应用程序自动复用多个连接。这有助于限制和数据库服务器的通信过载；
   7) 关于：显示主要 ODBC 组件的版本。

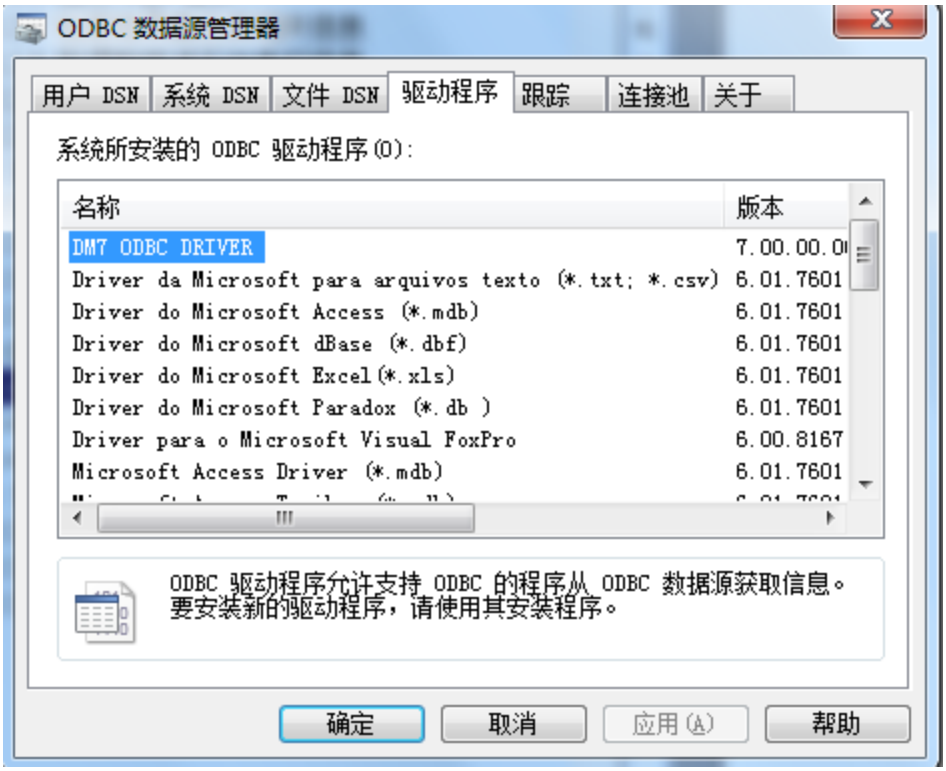


图 3.1 ODBC 数据源管理器对话框

2. 设置和配置一个系统 DSN，请单击系统 DSN 标签，单击添加按钮增加一个新的 DSN，显示如图 3.2 所示的对话框。

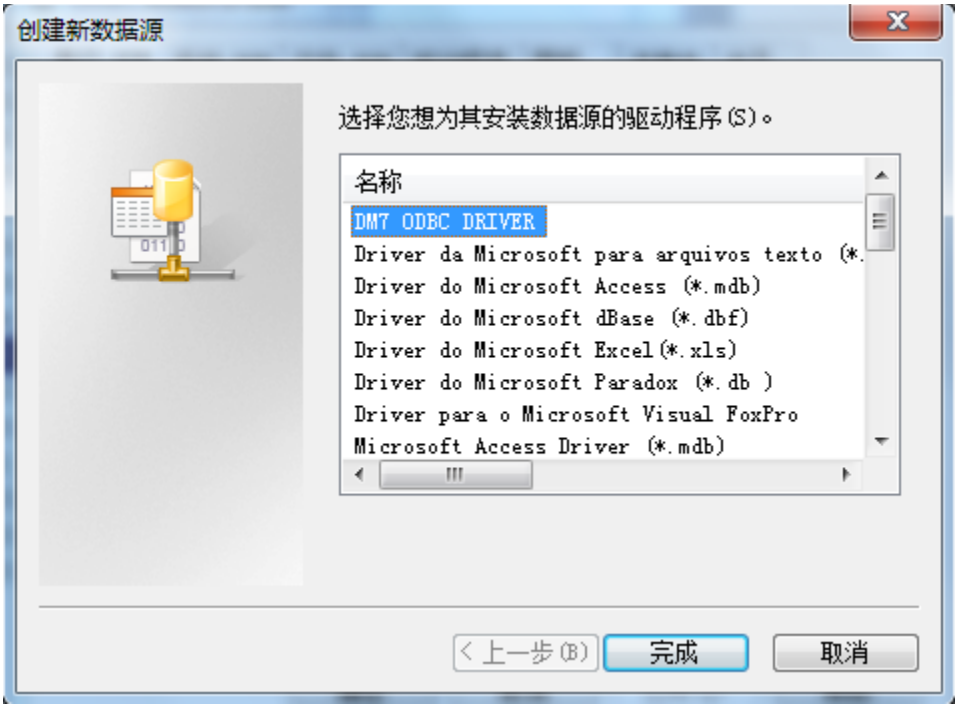


图 3.2 创建新数据源对话框

3. 选择 DM ODBC 3.0 驱动程序即 DM ODBC DRIVER，单击完成按钮，显示如图 3.3 所示的 DM ODBC 3.0 数据源配置对话框。

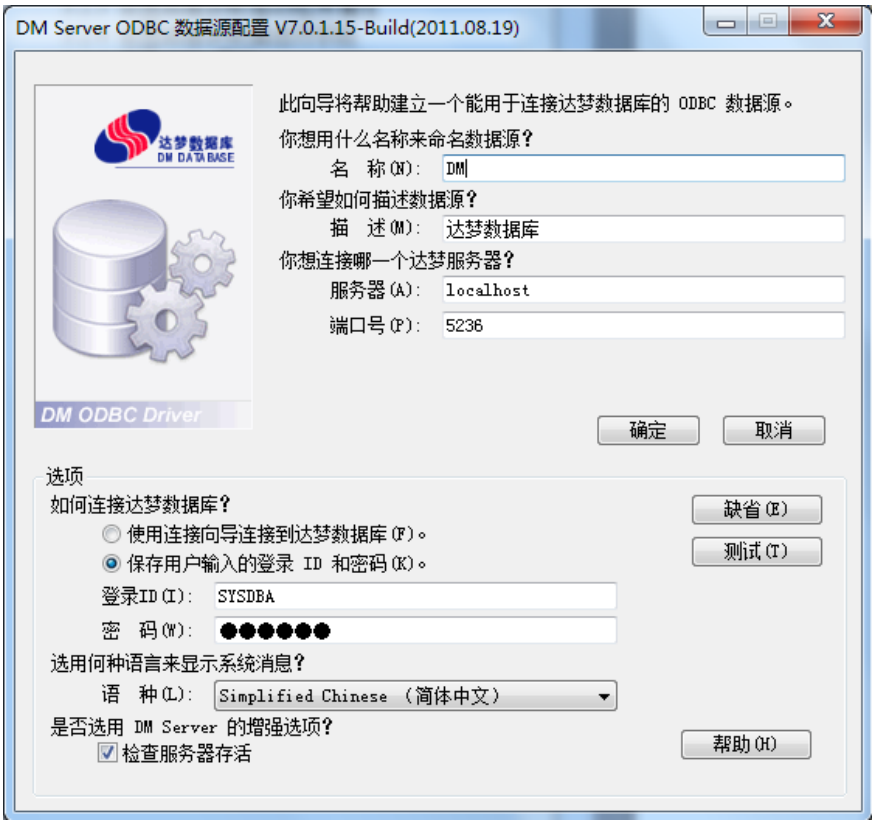


图 3.3 创建新的 DM 数据源对话框

4. 输入数据源的名称、一个简单的描述，并选择你想要连接的数据库服务器的名字、使用的端口号、验证登录用户 ID 真伪的方式，如果使用 DMServer 验证方式则需要输入登录数据源的 ID 以及密码等信息，选择系统提示信息的语种，以及选择是否使用 DMServer 的增强选项。
5. 单击测试按钮测试你配置的数据源是否正确，得到数据源测试报告如图 3.4 所示，如果测试成功，可以单击确定按钮以保存你设置的新的系统数据源，如图 3.5 所示：

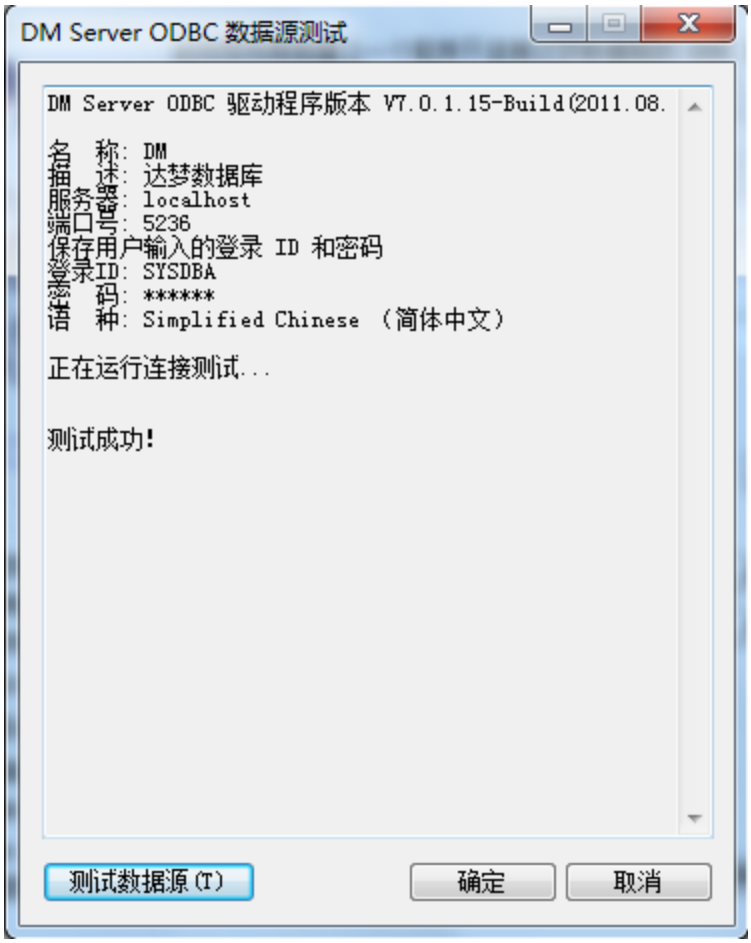


图 3.4 数据源测试报告

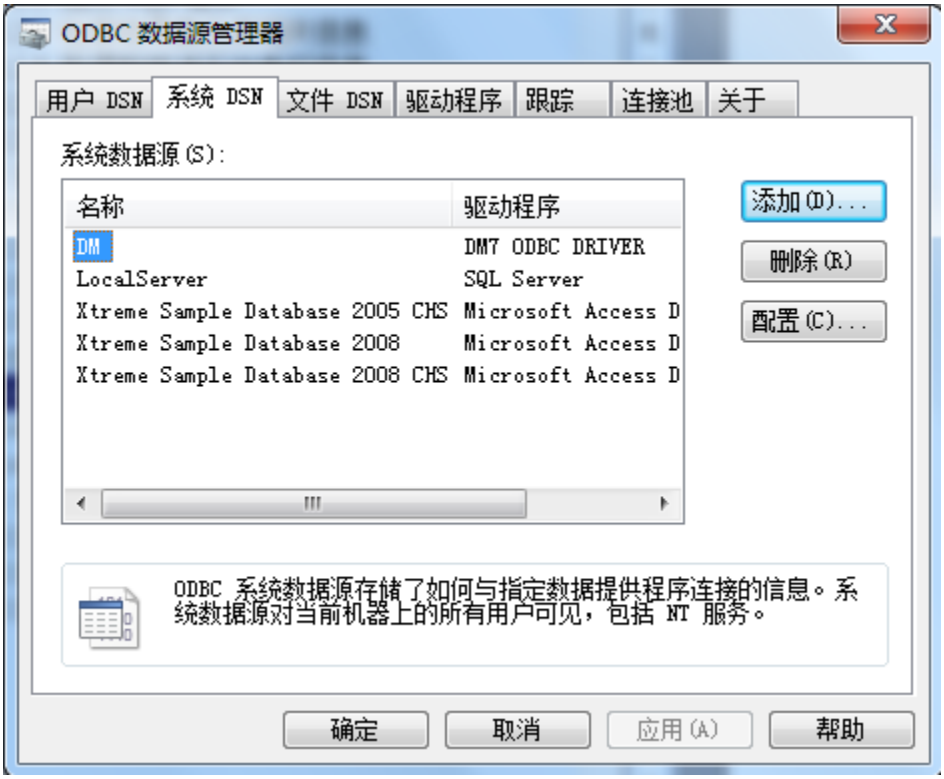


图 3.5 完成系统数据源的设置

6. 单击确定按钮关闭 ODBC 数据源管理器对话框。

3.4.2 Linux 上创建 ODBC 数据源

DMODBC 在 Linux 操作系统上的使用依赖于 UnixODBC 库，如果 UnixODBC 未安装在系统目录下，则为了能使 DMODBC 能找到需要的库文件，用户需要设置系统环境变量 LD_LIBRARY_PATH 指向动态库。另外，如果安装的 UnixODBC 生成的动态库名称不是 libodbcinst.so（如 libodbcinst.so.1.0.0 或者 libodbcinst.so.2.0.0 等），则需要对实际库文件建立符号链接。

在 Linux 上配置 ODBC 数据源的方式有两种，手动配置和图形配置。本节将介绍如何为你的应用程序安装和配置 ODBC 数据源。

1. 手动配置

1) 编辑/etc/odbcinst.ini，输入如下内容：

```
[DM8 ODBC DRIVER]
Description     = ODBC DRIVER FOR DM8
Driver          = /lib/libdodbc.so
```

2) 编辑/etc/odbc.ini，输入如下内容：

```
[dm]
Description     = DM ODBC DSN
Driver          = DM8 ODBC DRIVER
SERVER              = localhost
UID                 = SYSDBA
PWD                 = SYSDBA
TCP_PORT    = 5236
```

注意：

1) odbc.ini 中的 Driver 内容一定要与 odbcinst.ini 中的达梦驱动定义的节点名称相同。
2) odbc.ini 中的 SERVER 可以输入数据库服务器的 IP。

2. 图形配置

图形配置方式与 Windows 上基本相同。

1) 安装 unixODBC。可以从 www.unixodbc.org 上下载最新的 unixODBC 进行安装。
2) 运行 ODBCConfig，显示如图 3.6 所示的对话框。

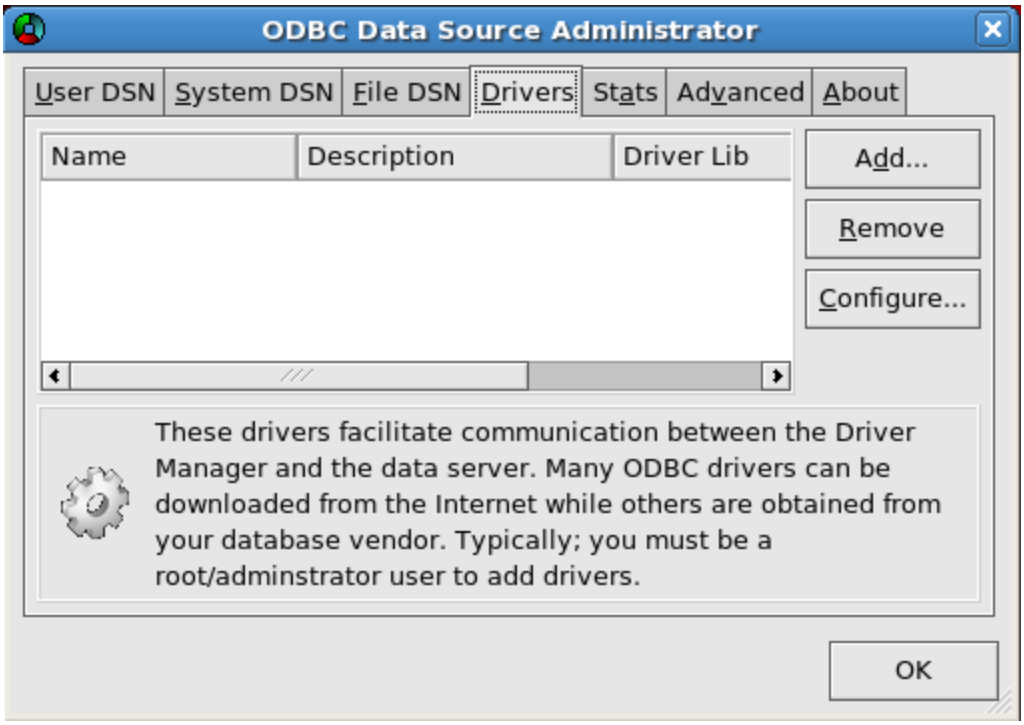


图 3.6 ODBC 数据源配置对话框

3) 安装 DM 数据库的 ODBC 驱动程序。点击 Drivers 页面，单击 Add...按钮，显示如图 3.7 所示的对话框。

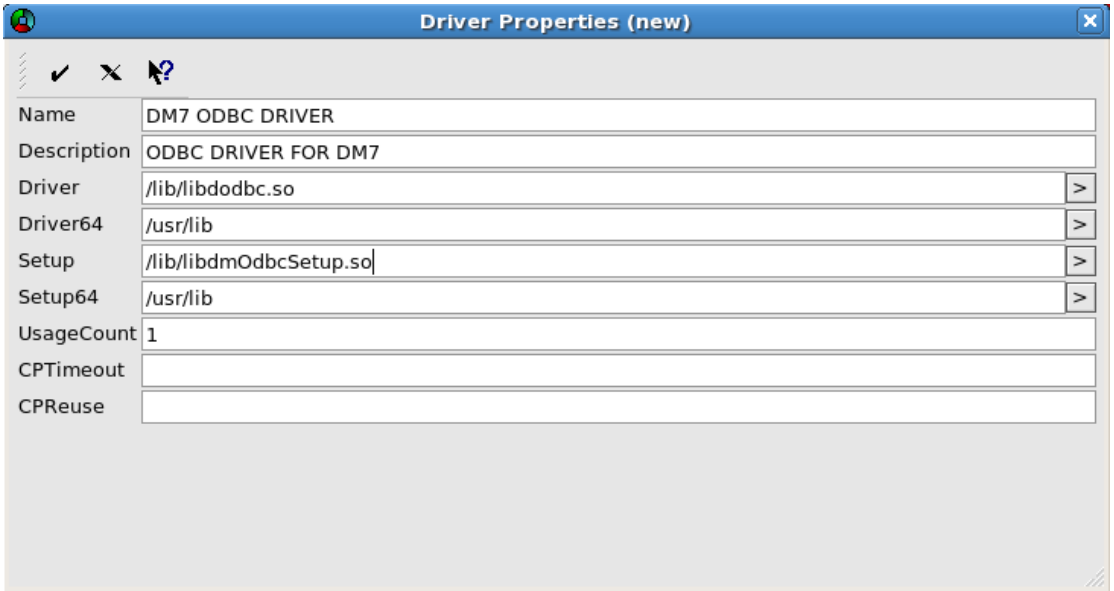


图 3.7 配置 DM 数据库 ODBC 驱动

在 Name、Description、Driver 和 Setup 中分别填入数据库驱动的名称、描述、数据库驱动程序和驱动安装程序，然后点击"√"保存退出。

4) 设置 System DSN。点击 System DSN 页面，单击 Add...按钮，列表中会显示已经安装好的数据库驱动程序，这里选中刚安装的 DM 数据库驱动，点击 OK 按钮，显示如图 3.8 所示对话框。

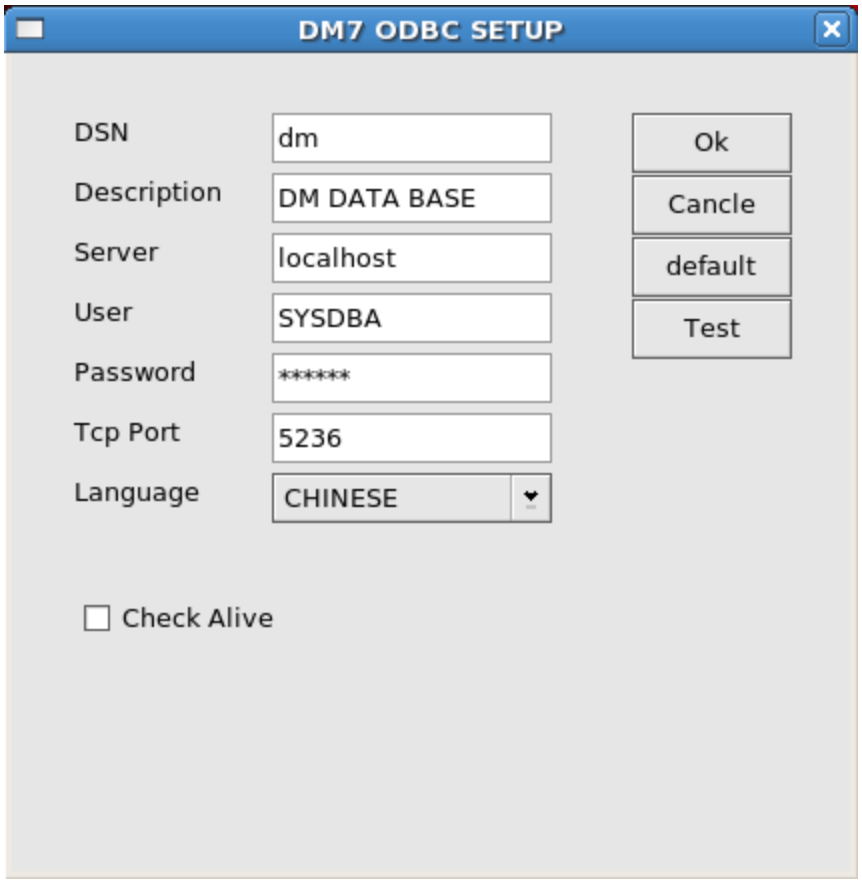


图 3.8 配置 System DSN

填入 DSN 的名称、描述、服务器地址、用户名、密码、端口等相关信息。点击 OK 保存退出。

5) 单击 OK 按钮关闭 ODBC 数据源管理器对话框。

3.4.3 ODBC 应用程序编写的基本步骤

应用程序使用 ODBC 访问数据源，可以按照以下几个基本步骤进行：

1. 调用函数 SQLAllocHandle 申请环境、连接句柄，调用函数 SQLSetEnvAttr 设置环境句柄属性，调用函数 SQLSetConnectAttr 设置连接句柄属性，调用连接函数 SQLConnect、SQLDriverConnect 或 SQLBrowseConnect 连接相关的数据源；
2. 调用函数 SQLAllocHandle 申请语句句柄，通过语句句柄应用程序可以执行 SQL 语句进行相关的 SQL 操作。调用函数 SQLPrepare 对 SQL 语句和操作进行准备，调用 SQLDescribeCol、SQLDescribeParam 等函数取得相关的描述信息，依据描述信息调用 SQLBindCol、SQLBindParam 等函数绑定相关的列和参数，然后调用 SQLExecute 执行 SQL 语句，实现相关的 SQL 操作。应用程序也可以调用函数 SQLExecDirect 直接执行 SQL 语句进行相关的 SQL 操作；
3. 应用程序可以通过调用 ODBC 编目函数 SQLTables、SQLColumns、SQLStatistics 等取得数据源相关的字典信息；
4. 如果连接属性自动提交选项设置为手动提交状态，应用程序可以调用函数 SQLEndTran 来提交或回滚事务，进行相关的事务处理；
5. 调用函数 SQLFreeHandle 来释放申请的语句句柄；

6. 调用函数 SQLDisconnect 来断开应用程序与数据源之间的连接；
7. 调用函数 SQLFreeHandle 来释放申请的连接、环境句柄。

使用 ODBC 编程的基本步骤如图 3.9 所示：

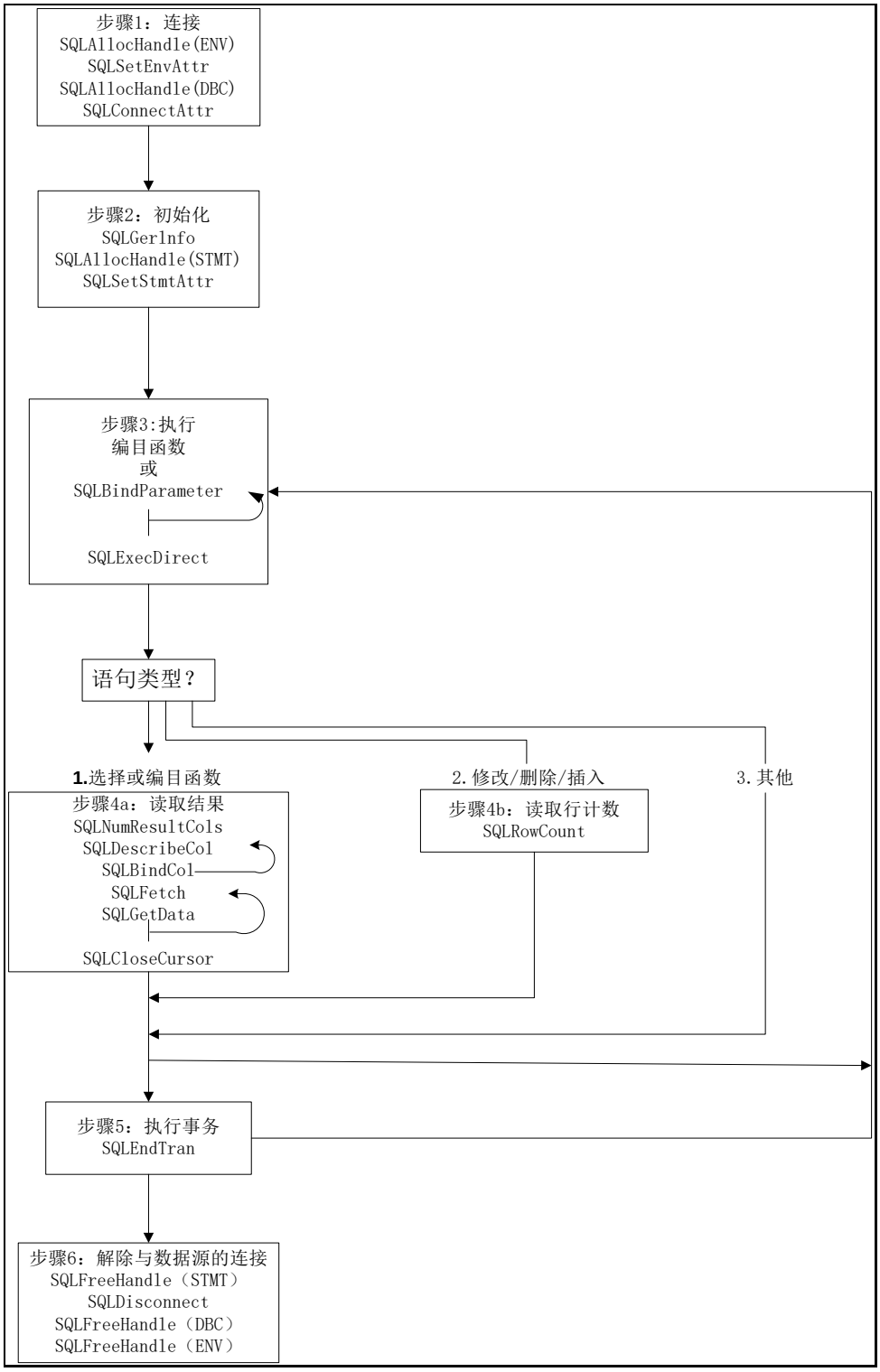


图 3.9 直接使用 ODBC 函数开发应用程序的基本步骤

下面是一个调用 DM ODBC 3.0 的简单实例：

```
#include <windows.h>
#include <stdio.h>
```

```
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
/* 检测返回代码是否为成功标志，当为成功标志返回 TRUE，否则返回 FALSE */
#define RC_SUCCESSFUL(rc)   ((rc) == SQL_SUCCESS || (rc) == SQL_SUCCESS_WITH_INFO)
/* 检测返回代码是否为失败标志，当为失败标志返回 TRUE，否则返回 FALSE */
#define RC_NOTSUCCESSFUL(rc) (!(RC_SUCCESSFUL(rc)))
HENV            henv;     /* 环境句柄 */
HDBC            hdbc;     /* 连接句柄 */
HSTMT           hsmt;     /* 语句句柄 */
SQLRETURN       sret;/* 返回代码 */
char            szpersonid[11];       /*人员编号*/
SQLLEN          cbpersonid=0;
char            szname[51];           /*人员姓名*/
SQLLEN          cbname=0;
char            szphone[26];          /*联系电话*/
SQLLEN          cbphone=0;
void main(void)
{
/* 申请一个环境句柄                    */
SQLAllocHandle(SQL_HANDLE_ENV, NULL, &henv);
/* 设置环境句柄的 ODBC 版本              */
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3,
SQL_IS_INTEGER);
/* 申请一个连接句柄                    */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
SQLConnect(hdbc, (SQLCHAR *)"DM", SQL_NTS, (SQLCHAR *)"SYSDBA", SQL_NTS, (SQLCHAR
*)"SYSDBA", SQL_NTS);
/* 申请一个语句句柄                    */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hsmt);
/* 立即执行查询人员信息表的语句          */
SQLExecDirect(hsmt, (SQLCHAR *)"SELECT personid, name, phone FROM person.person;", SQL_NTS);
/* 绑定数据缓冲区                      */
SQLBindCol(hsmt, 1, SQL_C_CHAR, szpersonid, sizeof(szpersonid), &cbpersonid);
SQLBindCol(hsmt, 2, SQL_C_CHAR, szname, sizeof(szname), &cbname);
SQLBindCol(hsmt, 3, SQL_C_CHAR, szphone, sizeof(szphone), &cbphone);
/* 取得数据并且打印数据 */
printf("人员编号 人员姓名 联系电话\n");
for (;;) {
sret = SQLFetchScroll(hsmt, SQL_FETCH_NEXT, 0);
if (sret == SQL_NO_DATA_FOUND)
break;
printf("%s %s    %s\n", szpersonid, szname, szphone);
}
```

```
/* 关闭游标，终止语句执行 */
SQLCloseCursor(hsmt);
/* 释放语句句柄 */
SQLFreeHandle(SQL_HANDLE_STMT, hsmt);
/* 断开与数据源之间的连接 */
SQLDisconnect(hdbc);
/* 释放连接句柄 */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* 释放环境句柄*/
SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
```

3.5 使用存储过程和函数

DM 允许用户创建和使用存储模块，下面介绍如何在 DMODBC 应用中使用存储过程和函数。

3.5.1 存储过程与函数字典信息的获取

DM ODBC 3.0 支持字典函数 SQLProcedures 的调用，用户可以调用此函数来获取 DM 存储过程与函数的字典信息。

调用方法如下：

SQLProcedures(stmt, (SQLCHAR*)"SYSTEM", SQL_NTS, (SQLCHAR*)"SYSDBA", SQL_NTS, (SQLCHAR*)"TEST_PROC",SQL_NTS);

返回字典信息格式如表 3.4 所列。

表 3.4 字典信息说明表

编号	字典项	相关说明
1	PROCEDURE_CAT	存储模块编目信息
2	PROCEDURE_SCHEM	存储模块模式信息
3	PROCEDURE_NAME	存储模块名
4	NUM_INPUT_PARAMS	DM 暂时没有返回此项信息
5	NUM_OUTPUT_PARAMS	DM 暂时没有返回此项信息
6	NUM_RESULT_SETS	DM 暂时没有返回此项信息
7	REMARKS	DM 暂时没有返回此项信息
8	PROCEDURE_TYPE	存储模块的类型

DM ODBC 3.0 支持字典函数 SQLProcedureColums 的调用，用于返回存储模块的参数信息。

调用方法如下：

SQLProcedureColumns(stmt,(SQLCHAR*)"SYSTEM",SQL_NTS, (SQLCHAR*)"SYSDBA", SQL_NTS, (SQLCHAR*)"TEST_PROC", SQL_NTS, NULL, 0);

返回字典信息格式如表 3.5 所列。

表 3.5 字典信息说明表

编号	字典项	相关说明
1	PROCEDURE_CAT	存储模块编目信息
2	PROCEDURE_SCHEM	存储模块模式信息
3	PROCEDURE_NAME	存储模块名
4	COLUMN_NAME	参数名
5	COLUMN_TYPE	参数的类型，即为输入参数还是输出参数
6	DATA_TYPE	参数的 SQL 数据类型
7	TYPE_NAME	参数的类型名
8	COLUMN_SIZE	参数的精度
9	BUFFER_LENGTH	参数所占用的字符长度
10	DECIMAL_DIGITS	参数的刻度
11	NUM_PREC_RADIX	仅对数值类型有效，仅为 10 或者 2，如果为 10 表示为精确数字，如果为 2 表示为非精确数字
12	NULLABLE	参数是否接收空值标志
13	REMARK	参数说明
14	COLUMN_DEF	参数的缺省值
15	SQL_DATA_TYPE	参数的 SQL 数据类型
16	SQL_DATETIME_SUB	日期时间类型或者时间间隔类型的子代码
17	CHAR_OCTET_LENGTH	字符数据类型以字节计算的最大长度，非字符类型返回空值
18	ORDINAL_POSITION	参数的顺序
19	IS_NULLABLE	参数是否包含空值

3.5.2 存储模块的创建

用户可以使用 SQLExecDirect 函数执行创建存储模块的 SQL 语句来创建存储模块，如下例所示：

```
SQLExecDirect(stmt, (SQLCHAR *)"create or replace procedure test_proc(c1 in int) as declare
c2 int
begin
c2 := c1 + 100;
end;", SQL_NTS);
```

3.5.3 存储模块的调用

调用存储模块的方法可以分为两种情况。

1. 立即调用

如果存储过程需要设置参数，那么在调用存储模块的时候就已经为所有的 IN、INOUT 类型的参数进行了赋值，带有 OUT 属性的参数的值是通过取得存储过程结果集的方法获取的。立即执行存储过程的示例如下：

```
SQLExecDirect(hsmt,(SQLCHAR*)"call test_proc(123);",SQL_NTS);
```

2. 参数调用

这种调用方法指的是在调用模块的时候，其参数值用问号来代替，发送给服务器之后，服务器返回参数的准备信息，用户依据服务器返回的参数描述信息进行参数绑定，然后执行，其参数的处理方法与普通的参数处理方法相同。

DM 支持存储模块返回多个结果集的处理，多结果集的切换通过 SQLMoreResult 函数切换，下面通过一个实例来说明如何使用。

```
#include <windows.h>
#include <stdio.h>
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
HENV      env;
HDBC      dbc;
HSTMT     stmt;
RETCODEret;
short i;
short cols;
char  colname[129];
char  coldata[256];
void main(void)
{
SQLAllocHandle(SQL_HANDLE_ENV, NULL, &env);
SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3, SQL_IS_INTEGER);
SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
SQLConnect(dbc, (SQLCHAR *)"DM", SQL_NTS, (SQLCHAR *)"SYSDBA", SQL_NTS, (SQLCHAR *)"SYSDBA", SQL_NTS);
SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
SQLExecDirect(stmt, (SQLCHAR *)"drop table test_table1;", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"drop table test_table2;", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"create table test_table1 (t1col int);", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"insert into test_table1 values(100);", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"create table test_table2 (t2col varchar(10));", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"insert into test_table2 values('hello!');", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"create or replace procedure test_proc as begin select * from test_table1;select * from test_table2;end;", SQL_NTS);
SQLExecDirect(stmt, (SQLCHAR *)"call test_proc;", SQL_NTS);
SQLNumResultCols(stmt, &cols);
for (i=1; i<=cols; i++)
{
SQLDescribeCol(stmt, i, (SQLCHAR *)colname, 129, NULL, NULL, NULL, NULL, NULL);
printf("%s ", colname);
}
printf("\n");
for ( ; ; )
{
```

```
ret = SQLFetch(stmt);
if (ret == SQL_NO_DATA_FOUND) break;
for (i=1; i<=cols; i++)
{
SQLGetData(stmt, i, SQL_C_CHAR, coldata, 256, NULL);
printf("%s ", coldata);
}
printf("\n");
}
SQLMoreResults(stmt);
SQLNumResultCols(stmt, &cols);
for (i=1; i<=cols; i++)
{
SQLDescribeCol(stmt, i, (SQLCHAR *)colname, 129, NULL, NULL, NULL, NULL, NULL);
printf("%s ", colname);
}
printf("\n");
for ( ; ; )
{
ret = SQLFetch(stmt);
if (ret == SQL_NO_DATA_FOUND) break;
for (i=1; i<=cols; i++)
{
SQLGetData(stmt, i, SQL_C_CHAR, coldata, 256, NULL);
printf("%s ", coldata);
}
printf("\n");
}
SQLFreeHandle(SQL_HANDLE_STMT, stmt);
SQLDisconnect(dbc);
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
}
```

第 4 章 DM JDBC 编程指南

4.1 JDBC 介绍

JDBC（Java Database Connectivity）是 Java 应用程序与数据库的接口规范，旨在让各数据库开发商为 Java 程序员提供标准的数据库应用程序编程接口（API）。JDBC 定义了一个跨数据库、跨平台的通用 SQL 数据库 API。

DM JDBC 驱动程序是 DM 数据库的 JDBC 驱动程序，它是一个能够支持基本 SQL 功能的通用应用程序编程接口，支持一般的 SQL 数据库访问。

通过 JDBC 驱动程序，用户可以在应用程序中实现对 DM 数据库的连接与访问，JDBC 驱动程序的主要功能包括：

1. 建立与 DM 数据库的连接；
2. 转接发送 SQL 语句到数据库；
3. 处理并返回语句执行结果。

4.2 JDBC 基本示例

利用 JDBC 驱动程序进行编程的一般步骤为：

1. 获得 java.sql.Connection 对象。
   利用 DriverManager 或者数据源来建立同数据库的连接。
2. 创建 java.sql.Statement 对象。这里也包含了 java.sql.PreparedStatement 和 java.sql.CallableStatement 对象。
   利用连接对象的创建语句对象的方法来创建。在创建的过程中，根据需要来设置结果集的属性。
3. 数据操作。
   数据操作主要分为两个方面，一个是更新操作，例如更新数据库、删除一行、创建一个新表等；另一个就是查询操作，执行完查询之后，会得到一个 java.sql.ResultSet 对象，可以操作该对象来获得指定列的信息、读取指定行的某一列的值。
4. 释放资源。
   在操作完成之后，用户需要释放系统资源，主要是关闭结果集、关闭语句对象，释放连接。当然，这些动作也可以由 JDBC 驱动程序自动执行，但由于 Java 语言的特点，这个过程会比较慢(需要等到 Java 进行垃圾回收时进行)，容易出现意想不到的问题。

下面用一个具体的编程实例来展示利用 JDBC 驱动程序进行数据库操作的基本步骤：

```
/*该例程实现插入数据，修改数据，删除数据，数据查询等基本操作。*/
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.font.FontRenderContext;
import java.awt.geom.Rectangle2D;
```

```
import java.awt.image.BufferedImage;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigDecimal;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import javax.imageio.ImageIO;
public class BasicApp {
	// 定义 DM JDBC 驱动串
	String jdbcString = "dm.jdbc.driver.DmDriver";
	// 定义 DM URL 连接串
	String urlString = "jdbc:dm://localhost:5236";
	// 定义连接用户名
	String userName = "SYSDBA";
	// 定义连接用户口令
	String password = "SYSDBA";
	// 定义连接对象
	Connection conn = null;
	/* 加载 JDBC 驱动程序
	 * @throws SQLException 异常 */
	public void loadJdbcDriver() throws SQLException {
		try {
			System.out.println("Loading JDBC Driver...");
			// 加载 JDBC 驱动程序
			Class.forName(jdbcString);
		} catch (ClassNotFoundException e) {
			throw new SQLException("Load JDBC Driver Error : " + e.getMessage());
		} catch (Exception ex) {
			throw new SQLException("Load JDBC Driver Error : "
					+ ex.getMessage());
		}
	}
```

```
/* 连接 DM 数据库
 * @throws SQLException 异常 */
public void connect() throws SQLException {
    try {
        System.out.println("Connecting to DM Server...");
        // 连接 DM 数据库
        conn = DriverManager.getConnection(urlString, userName, password);
    } catch (SQLException e) {
        throw new SQLException("Connect to DM Server Error : "
                + e.getMessage());
    }
}
/* 关闭连接
 * @throws SQLException 异常 */
public void disConnect() throws SQLException {
    try {
        // 关闭连接
        conn.close();
    } catch (SQLException e) {
        throw new SQLException("close connection error : " + e.getMessage());
    }
}
/* 往产品信息表插入数据
 * @throws SQLException 异常 */
public void insertTable() throws SQLException {
    // 插入数据语句
    String sql = "INSERT INTO production.product(name,author,publisher,publishtime,"
            + "product_subcategoryid,productno,satetystocklevel,originalprice,nowprice,discount,"
            + "description,photo,type,papertotal,wordtotal,sellstarttime,sellendtime) "
            + "VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?);";
    // 创建语句对象
    PreparedStatement pstmt = conn.prepareStatement(sql);
    // 为参数赋值
    pstmt.setString(1, "三国演义");
    pstmt.setString(2, "罗贯中");
    pstmt.setString(3, "中华书局");
    pstmt.setDate(4, Date.valueOf("2005-04-01"));
    pstmt.setInt(5, 4);
    pstmt.setString(6, "9787101046121");
    pstmt.setInt(7, 10);
    pstmt.setBigDecimal(8, new BigDecimal(19.0000));
    pstmt.setBigDecimal(9, new BigDecimal(15.2000));
    pstmt.setBigDecimal(10, new BigDecimal(8.0));
    pstmt.setString(11, "《三国演义》是中国第一部长篇章回体小说，中国小说由短篇发展至长篇的
```

```
原因与说书有关。");
            // 设置大字段参数
            try {
                // 创建一个图片用于插入大字段
                String filePath = "c:\\三国演义.jpg";
                CreateImage(filePath);
                File file = new File(filePath);
                InputStream in = new BufferedInputStream(new FileInputStream(file));
                pstmt.setBinaryStream(12, in, (int) file.length());
            } catch (FileNotFoundException e) {
                System.out.println(e.getMessage());
                // 如果没有图片设置为 NULL
                pstmt.setNull(12, java.sql.Types.BINARY);
            } catch (IOException e) {
            System.out.println(e.getMessage());
            }
            pstmt.setString(13, "25");
            pstmt.setInt(14, 943);
            pstmt.setInt(15, 93000);
            pstmt.setDate(16, Date.valueOf("2006-03-20"));
            pstmt.setDate(17, Date.valueOf("1900-01-01"));
            // 执行语句
            pstmt.executeUpdate();
            // 关闭语句
            pstmt.close();
        }
        /* 查询产品信息表
         * @throws SQLException 异常 */
        public void queryProduct() throws SQLException {
            // 查询语句
            String sql = "SELECT productid,name,author,description,photo FROM production.product WHERE
productid=11";
            // 创建语句对象
            Statement stmt = conn.createStatement();
            // 执行查询
            ResultSet rs = stmt.executeQuery(sql);
            // 显示结果集
            displayResultSet(rs);
            // 关闭结果集
            rs.close();
            // 关闭语句
            stmt.close();
        }
        /* 修改产品信息表数据
```

```
 * @throws SQLException 异常 */
public void updateTable() throws SQLException {
        // 更新数据语句
        String sql = "UPDATE production.product SET name = ?"
                        + "WHERE productid = 11;";
        // 创建语句对象
        PreparedStatement pstmt = conn.prepareStatement(sql);
        // 为参数赋值
        pstmt.setString(1, "三国演义（上）");
        // 执行语句
        pstmt.executeUpdate();
        // 关闭语句
        pstmt.close();
}
/* 删除产品信息表数据
 * @throws SQLException 异常 */
public void deleteTable() throws SQLException {
        // 删除数据语句
        String sql = "DELETE FROM production.product WHERE productid = 11;";
        // 创建语句对象
        Statement stmt = conn.createStatement();
        // 执行语句
        stmt.executeUpdate(sql);
        // 关闭语句
        stmt.close();
}
/* 查询产品信息表
 * @throws SQLException 异常 */
public void queryTable() throws SQLException {
        // 查询语句
        String sql = "SELECT productid,name,author,publisher FROM production.product";
        // 创建语句对象
        Statement stmt = conn.createStatement();
        // 执行查询
        ResultSet rs = stmt.executeQuery(sql);
        // 显示结果集
        displayResultSet(rs);
        // 关闭结果集
        rs.close();
        // 关闭语句
        stmt.close();
}
/* 调用存储过程修改产品信息表数据
 * @throws SQLException 异常 */
```

```
public void updateProduct() throws SQLException {
    // 更新数据语句
    String sql = "{ CALL production.updateProduct(?,?) }";
    // 创建语句对象
    CallableStatement cstmt = conn.prepareCall(sql);
    // 为参数赋值
    cstmt.setInt(1, 1);
    cstmt.setString(2, "红楼梦（上）");
    // 执行语句
    cstmt.execute();
    // 关闭语句
    cstmt.close();
}
/* 显示结果集
 * @param rs 结果集对象
 * @throws SQLException 异常 */
private void displayResultSet(ResultSet rs) throws SQLException {
    // 取得结果集元数据
    ResultSetMetaData rsmd = rs.getMetaData();
    // 取得结果集所包含的列数
    int numCols = rsmd.getColumnCount();
    // 显示列标头
    for (int i = 1; i <= numCols; i++) {
        if (i > 1) {
            System.out.print(",");
        }
        System.out.print(rsmd.getColumnLabel(i));
    }
    System.out.println("");
    // 显示结果集中所有数据
    while (rs.next()) {
        for (int i = 1; i <= numCols; i++) {
            if (i > 1) {
                System.out.print(",");
            }
            // 处理大字段
            if ("IMAGE".equals(rsmd.getColumnTypeName(i))) {
                byte[] data = rs.getBytes(i);
                if (data != null && data.length > 0) {
                    FileOutputStream fos;
                    try {
                        fos = new FileOutputStream("c:\\三国演义 1.jpg");
                        fos.write(data);
                        fos.close();
```

```
                        } catch (FileNotFoundException e) {
                            System.out.println(e.getMessage());
                        } catch (IOException e) {
                            System.out.println(e.getMessage());
                        }
                    }
            System.out.print("字段内容已写入文件 c:\\三国演义 1.jpg，长度" + data.length);
                } else {
                    // 普通字段
                    System.out.print(rs.getString(i));
                }
            }
            System.out.println("");
        }
    }
     /* 创建一个图片用于插入大字段
     * @throws IOException 异常 */
    private void CreateImage(String path) throws IOException {
        int width = 100;
        int height = 100;
        String s = "三国演义";
        File file = new File(path);
        Font font = new Font("Serif", Font.BOLD, 10);
        BufferedImage bi = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_RGB);
        Graphics2D g2 = (Graphics2D) bi.getGraphics();
        g2.setBackground(Color.WHITE);
        g2.clearRect(0, 0, width, height);
        g2.setPaint(Color.RED);
        FontRenderContext context = g2.getFontRenderContext();
        Rectangle2D bounds = font.getStringBounds(s, context);
        double x = (width - bounds.getWidth()) / 2;
        double y = (height - bounds.getHeight()) / 2;
        double ascent = -bounds.getY();
        double baseY = y + ascent;
        g2.drawString(s, (int) x, (int) baseY);
        ImageIO.write(bi, "jpg", file);
    }
    /*
     * 类主方法 @param args 参数
     */
    public static void main(String args[]) {
        try {
            // 定义类对象
```

```
                BasicApp basicApp = new BasicApp();
                // 加载驱动程序
                basicApp.loadJdbcDriver();
                // 连接 DM 数据库
                basicApp.connect();
                // 插入数据
                System.out.println("--- 插入产品信息 ---");
                basicApp.insertTable();
                // 查询含有大字段的产品信息
                System.out.println("--- 显示插入结果 ---");
                basicApp.queryProduct();
                // 在修改前查询产品信息表
                System.out.println("--- 在修改前查询产品信息 ---");
                basicApp.queryTable();
                // 修改产品信息表
                System.out.println("--- 修改产品信息 ---");
                basicApp.updateTable();
                // 在修改后查询产品信息表
                System.out.println("--- 在修改后查询产品信息 ---");
                basicApp.queryTable();
                // 删除产品信息表
                System.out.println("--- 删除产品信息 ---");
                basicApp.deleteTable();
                // 在删除后查询产品信息表
                System.out.println("--- 在删除后查询产品信息 ---");
                basicApp.queryTable();
                // 调用存储过程修改产品信息表
                System.out.println("--- 调用存储过程修改产品信息 ---");
                basicApp.updateProduct();
                // 在存储过程更新后查询产品信息表
                System.out.println("--- 调用存储过程后查询产品信息 ---");
                basicApp.queryTable();
                // 关闭连接
                basicApp.disConnect();
            } catch (SQLException e) {
                System.out.println(e.getMessage());
            }
        }

}
```

运行程序，启动达梦数据库服务器并创建用于修改产品信息的存储过程。存储过程代码如下：

```
-- 创建修改产品信息的存储过程
create or replace procedure "PRODUCTION"."updateProduct"
```

```
(
   v_id int,
   v_name varchar(50)
)
as
begin
   UPDATE production.product SET name = v_name WHERE productid = v_id;
end;
```

将以上 Java 程序保存为 C:\BasicApp.java 文件，打开命令行窗口进入 C 盘根目录，输入 javac BasicApp.java 编译该类文件，编译完成后在命令行窗口输入 java -classpath D:\dmdbms\jdbc\DmJdbcDriver.jar; BasicApp 运行该文件。其中，DmJdbcDriver.jar 有三个版本： DmJdbcDriver.jar15、DmJdbcDriver.jar16 和 DmJdbcDriver.jar17，分别对应 JDK 版本 1.5、1.6 和 1.7，用户根据需要自行选择。

注意这里还需要将达梦数据库驱动程序 DmJdbcDriver.jar（根据 JDK 版本选择上述三个驱动之一）文件加入到 classpath 中，达梦数据库驱动程序 DmJdbcDriver.jar 文件存在于达梦数据库安装路径下的 jdbc 目录下，这里假设达梦数据库安装在 D:\dmdbms7 目录，那么达梦数据库驱动程序 DmJdbcDriver.jar 文件就在 D:\dmdbms7\drivers\ jdbc 目录下。

4.3 DM JDBC 特性

DM JDBC 驱动程序有 JDBC3.0 和 JDBC4.0 两种类型。

DM JDBC 3.0 驱动程序符合 SUN JDBC3.0 标准，实现了 JDBC3.0 规范所要求的下列接口：

```
java.sql.Driver
java.sql.Connection
java.sql.Statement
java.sql.PreparedStatement
java.sql.CallableStatement
java.sql.ResultSet
java.sql.ResultSetMetaData
java.sql.DatabaseMetaData
java.sql.Blob
java.sql.Clob
java.sql.PrameterMetaData
java.sql.Savepoint
javax.sql.DataSource
javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSource
javax.sql.PooledConnection
```

DM JDBC 4.0 驱动程序符合 SUN JDBC4.0 标准，实现了下列新特性：

1. JDBC 驱动类的自动加载；
2. 连接管理的增强；
3. 对 RowId SQL 类型的支持；

4. SQL 的 DataSet 实现使用了 Annotations；
5. SQL 异常处理的增强；
6. 对 SQL XML 的支持；
7. 对 BLOB/CLOB 的改进支持以及对国际字符集的支持。

4.4 DM JDBC 扩展

4.4.1 数据类型扩展

DM JDBC 驱动程序为了支持 DM 特有的数据类型，采用了自定义扩展包的方式来进行解决。DM.sql 这个包中包含了 DM 特有数据类型所对应的类。时间间隔类型是 DM 具有的数据类型，JDBC 标准中没有相对应的类型。故令 dm.sql.DmdbIntervalYM 和 dm.sql.DmdbIntervalDT 分别对应于年-月间隔类型和日-时间隔类型。JDBC 标准中的 java.sql.Types.Time 类型不允许带有纳秒，为了表示 DM 中带纳秒的时间类型，设立了 dm.sql.DmdbTime 类型。这些类型的主要方法为：

1. Dm.sql.DmdbIntervalDT

方法名	功能说明
DmdbIntervalDT(byte[])	利用字节数组作为参数的构造函数。
DmdbIntervalDT(byte[],int,int)	利用字节数组，指定引导精度、纳秒精度构造函数。
DmdbIntervalDT(String)	利用字符串作为参数的构造函数。
DmdbIntervalDT(String,int,int)	设置字符串，指定引导精度、纳秒精度构造函数。
getByteArrayValue()	获取字节数组。
getDTString()	把 DmdbIntervalDT 的值以字符串的形式返回。
getDTType()	获取 DmdbIntervalDT 值类型。
getDay()	获取 DmdbIntervalDT 值中日值。
getHour()	获取 DmdbIntervalDT 值中时值。
getMinute()	获取 DmdbIntervalDT 值中分值。
getSecond()	获取 DmdbIntervalDT 值中秒值。
getNano()	获取 DmdbIntervalDT 值中纳秒值。
getLoadPrec()	获取 DmdbIntervalDT 值中引导精度数。
getSecPrec()	获取 DmdbIntervalDT 值中纳秒精度数。
clear()	清除 DmdbIntervalDT 值。

2. Dm.sql.DmdbIntervalYM

方法名	功能说明
DmdbIntervalYM (byte[])	利用字节数组作为参数的构造函数。
DmdbIntervalYM (byte[],int)	利用字节数组、精度作为参数构造函数，精度默认为 0。
DmdbIntervalYM (String)	利用字符串作为参数的构造函数。
DmdbIntervalYM (String,int)	利用字符串、精度作为参数的构造函数，精度默认为 0。
getByteArrayValue()	把 DmdbIntervalYM 的值以字节数组的形式返回。
getYMString()	把 DmdbIntervalYM 的值以字符串的形式返回。
getYMType()	获取 DmdbIntervalYM 值类型。

getYear()	返回 DmdbIntervalYM 值中的年值。
getMonth()	返回 DmdbIntervalYM 值中的月值。
getLoadPrec()	获取 DmdbIntervalYM 的引导精度。
toString()	把 DmdbIntervalYM 的值转化为字符串的形式。
clear()	清除 DmdbIntervalYM 值信息。

3. Dm.sql.DmdbTimestamp

方法名	功能说明
valueOf(Date)	根据指定的 java.util.Date 类型构造一个 DmdbTimestamp 类型
valueOf(String)	根据指定的时间类型字符串构造一个 DmdbTimestamp 类型
getDt()	返回一个此对象表示的时间值数组，内容为[年，月，日，时，分，秒，微秒，时区]
getTime ()	返回此对象表示的自 1970 年 1 月 1 日 00:00:00 GMT 以来的毫秒数
setTime(long)	使用给定毫秒时间值设置现有 DmdbTimestamp 对象
getTimezone()	获取 DmdbTimestamp 值中的时区值
setTimezone(int)	设置此对象的时区
getNano()	获取此对象的纳秒值
setNano(long)	设置此对象的纳秒值
toString()	把 DmdbTimestamp 的值转化为字符串的形式

4. 访问方式

为了能够在 DM JDBC 驱动程序中访问这些特有的数据类型，我们在 DmdbPreparedStatement、DmdbCallableStatement 和 DmdbResultSet 类中增加了许多方法。客户如果想使用这些方法，必须把对象强制转化为 DmdbXXX 类型：

```
PreparedStatement pstmt = connection.prepareStatement("insert into testInterval " + "values(?)");
((DmdbPreparedStatement)pstmt).setINTERVALYM(1, new DmdbIntervalYM("Interval '0015-08' year to month"));
int updateCount = pstmt.executeUpdate();
```

其它方法的使用方式与此类同。而且，时间间隔类型可以利用 setString 和 getString 方法进行存取，带纳秒的时间类型也可以当作普通的时间类型(不带纳秒)来进行操作。

4.4.2 读写分离集群下的错误信息

当使用 JDBC 接口连接 DM 读写分离集群系统时，对返回的报错信息进行了扩展。在报错信息前增加了一个字符前缀用于标识报错信息是来自读写分离集群的主库还是备库：

- 前缀[P]表示是来自主库的报错信息；
- 前缀[S]表示是来自备库的报错信息。

4.5 建立 JDBC 连接

通常有两种途径来获得 JDBC 的连接对象，一种是通过驱动管理器 DriverManager 的 getConnection(String url, String user, String password)方法来建立，另外一种就是通过数据源的方式来建立。JDBC 3.0 支持这两种建立连接的方式。

4.5.1 通过 DriverManager 建立连接

这种建立连接的途径是最常用的，也称作编程式连接。利用这种方式来建立连接通常需要以下几个步骤：

注册数据库驱动程序(driver)。可以通过调用 java.sql.DriverManager 类的 registerDriver 方法显式注册驱动程序，也可以通过加载数据库驱动程序类隐式注册驱动程序。

```
//显示注册
DriverManager.registerDriver(new dm.jdbc.driver.dmDriver());
//隐式注册
Class.forName("dm.jdbc.driver.DmDriver");
```

隐式注册过程中加载实现了 java.sql.Driver 的类，该类中有一静态执行的代码段，在类加载的过程中向驱动管理器 DriverManager 注册该类。而这段静态执行的代码段其实就是上述显式注册的代码。

建立连接。注册驱动程序之后，就可以调用驱动管理器的 getConnection 方法来建立连接。建立数据库连接需要指定连接数据库的 url、登录数据库所用的用户名 user 和密码 password。

通过 DriverManager 建立连接的具体过程如下：

1. 加载 DM JDBC 驱动程序

dm.jdbc.driver.DmDriver 类包含一静态部分，它创建该类的实例。当加载驱动程序时，驱动程序会自动调用 DriverManager.registerDriver 方法向 DriverManager 注册自己。通过调用方法 Class.forName(String str)，将显式地加载驱动程序。以下代码加载 DM 的 JDBC 驱动程序：

```
Class.forName("dm.jdbc.driver.DmDriver");
```

2. 建立连接

加载 DM JDBC 驱动程序并在 DriverManager 类中注册后，即可用来与数据库建立连接。DriverManager 对象提供三种建立数据库连接的方法。每种方法都返回一个 Connection 对象实例，区别是参数不同。

```
Connection DriverManager.getConnection(String url, java.util.Properties info);
Connection DriverManager.getConnection(String url);
Connection DriverManager.getConnection(String url, String user, String password);
```

通常采用第三种方式进行数据库连接，该方法通过指定数据库 url、用户名、口令来连接数据库。

以下代码建立与数据库的连接：

```
Class.forName("dm.jdbc.driver.DmDriver");   // 加载驱动程序
String url = "jdbc:dm://223.254.254.19";   // 主库 IP = 223.254.254.19
String userID = "SYSDBA";
String passwd = "SYSDBA";
Connection con = DriverManager.getConnection(url, userID, passwd);
```

利用这种方式来建立数据库连接，连接数据库所需要的参数信息都被硬编码到程序中，这样每次更换不同的数据库或登录用户信息都要对应用进行重新改写、编译，不够灵活。而且，当用户同时需要多个连接时，就不得不同时建立多个连接，造成资源浪费和性能低下。为了解决这些问题，SUN 公司在 JDBC 2.0 的扩展包中定义了数据源接口，提供了一种建立

连接的新途径。

4.5.2 创建 JDBC 数据源

数据源是在 JDBC 2.0 中引入的一个概念。在 JDBC 2.0 扩展包中定义了 javax.sql.DataSource 接口来描述这个概念。如果用户希望建立一个数据库连接，通过查询在 JNDI™ 服务中的数据源，可以从数据源中获取相应的数据库连接。这样用户就只需要提供一个逻辑名称（Logic Name），而不是数据库登录的具体细节。

JNDI™ 的全称是 Java Naming and Directory Interface， 可以理解为 Java 名称和目录服务接口。JNDI 向应用程序提供了一个查询和使用远程服务的机制。这些远程服务可以是任何企业服务。对于 JDBC 应用程序来说，JNDI 提供的是数据库连接服务。JNDI 使应用程序通过使用逻辑名称获取对象和对象提供的服务，从而使程序员可以避免使用与提供对象的机构有关联的代码。

一个 DataSource 对象代表一个实际的数据源。在数据源中存储了所有建立数据库连接的信息。系统管理员用一个逻辑名字对应 DataSource 对象，这个名字可以是任意的。在下面的例子中 DataSource 对象的名字是 NativeDB。依照传统习惯, DataSource 对象的名字包含在 jdbc/下，所以这个数据源对象的完整名字是：jdbc/NativeDB。

数据源的逻辑名字确定之后，就需要向 JNDI 服务注册该数据源。下面的代码展示了向 JNDI 服务注册数据源的过程。

```
// 初始化名称-目录服务环境
Context ctx = null;
    try{
        Hashtable env = new Hashtable (5);
        env.put (Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put (Context.PROVIDER_URL, "file:JNDI");
        ctx = new InitialContext(env);
    }
    catch (NamingException ne)
    {
        ne.printStackTrace();
    }
    bind(ctx, "jdbc/NativeDB");
```

程序首先生成了一个 Context 实例。javax.naming.Context 接口定义了名称服务环境(Naming Context)及该环境支持的操作。名称服务环境实际上是由名称和对象间的相互映射组成的。程序中初始化名称服务环境的环境工厂(Context Factory)是 com.sun.jndi.fscontext.RefFSContextFactory(该类在 fscontext.jar 中可以找到，由于 fscontext.jar 中包含的不是标准的 API，用户需要从 www.javasoft.com 中的 JNDI 专区下载 fscontext.jar)类，环境工厂的作用是生成名称服务环境的实例。javax.naming.spi.InitialContextFactory 接口定义了环境工厂应该如何初始化名称服务环境(该接口在 providerutil.jar 中实现，由于 providerutil.jar 中包含的不是标准的 API，用户需要从 www.javasoft.com 中的 JNDI 专区下载 providerutil.jar)。在初始化名称服务环境时还需要定义环境的 URL。程序中使用的是"file:JNDI"，也就是把环境保存在本地硬盘的 JNDI 目录下。目前很多 J2EETM 应用服务器都实现了自己的 JNDI 服务，用户可以选用这些服务包。

初始化了名称服务环境后，就可以把数据源实例注册到名称服务环境中。注册时调用 javax.naming.Context.bind()方法，参数为注册名称和注册对象。注册成功后，在 JNDI 目录下会生成一个.binding 文件，该文件记录了当前名称-服务环境拥有的名称及对象。具体实现如下例所示：

```
void bind (Context ctx, String ln)throws NamingException, SQLException
{
// 创建一个 DmdbDataSource 实例
DmdbDataSource dmds = new DmdbDataSource ();
// 把 DmdbDataSource 实例注册到 JNDI 中
ctx.bind (ln, dmds);
}
```

当需要在名称服务环境中查询一个对象时，需要调用 javax.naming.Context.lookup（）方法，并把查询到的对象显式转化为数据源对象。然后通过该数据源对象进行数据库操作。

```
DataSource ds = (DataSource) lookup (ctx, "jdbc/NativeDB");
Connection conn = ds.getConnection();
```

DataSource 对象中获得的 Connection 对象和用 DriverManager.getConnection 方法获得的对象是等同的。由于 DataSource 方法具有很多优点，该方法成为获得连接的推荐方法。

4.5.3 数据源与连接池

利用数据源可以增强代码的可移植性，方便代码的维护。而且还可以利用连接池的功能来提高系统的性能。连接缓冲池的工作原理是：当一个应用程序关闭一个连接时，这个连接并不真正释放而是被循环利用。因为建立连接是消耗较大的操作，循环利用连接可以减少新连接的建立，能够显著地提高性能。

JDBC 规范为连接池定义了两个接口，一个客户端接口和一个服务器端接口。客户端接口就是 javax.sql.DataSource，这样客户先前采用数据源来获得连接的代码就不需要任何的修改。通过数据源所获得的连接是否是缓冲的，这取决于具体实现的 JDBC 驱动程序是否实现了连接池的服务器端接口 javax.sql.ConnectionPoolDataSource。DM JDBC 实现了连接缓冲池，在实现连接缓冲池的过程中采用了新水平的高速缓存。一般说来，连接高速缓存是一种在一个池中保持数目较小的物理数据库连接的方式，这个连接池由大量的并行用户共享和重新使用，从而避免在每次需要时建立一个新的物理数据库连接，以及当其被释放时关闭该连接的昂贵的操作。连接池的实现对用户来说是透明的，用户不需为其修改任何代码。

4.5.4 DM 扩展连接属性的使用

除了标准 JDBC 接口功能，DM 扩展了一些具有自身特点的功能处理特性，这些特性可以通过在连接串上设置连接属性进行控制。

连接串的书写格式有以下两种：

1. host、port 不作为连接属性，此时只需输入值即可：

格式：

```
jdbc:dm [: //host][:port][?propName1=propValue1][&propName2=propValue2]….
```

注：

1） 若 host 不设置，则默认为‘localhost’

2） 若 port 不设置，则默认为‘5236’

3） 若 host 不设置，则 port 一定不能设

4） 若 user、password 没有单独作为参数传入，则必须在连接属性中传入

5） 若 host 为 ipv6 地址，则应包含在[]中

例：

```
jdbc:dm://192.168.0.96:5236?LobMode=1
```

2. host、port 作为连接属性，此时必须按照表 4.1 中说明进行设置，且属性名称大小写敏感

格式：

```
jdbc:dm:// [?propName1=propValue1][& propName2=propValue2][&…]…
```

注：

1） host、port 设置与否，以及在属性串中的位置没有限制

2） 若 user、password 没有单独作为参数传入，则必须在连接属性中传入

例：

```
jdbc:dm:// ?host=192.168.0.96&port=5236
```

连接串中可以设置的属性及其说明见下表。

表 4.1 JDBC 连接串属性

属性名称	说明	是否必须设置
“host”	主库地址，包括 IP 地址、localhost 或者配置文件中主库地址列表对应的变量名，如 dm_svc.conf 中的’o2000’	是
“port”	端口号，服务器登录端口号	否
“user”	登录用户	是
“password”	登录密码	是
“socketTimeout”	套接字超时时间，默认 0	否
“escapeProcess”	是否进行语法转义处理，默认 true，若确定使用环境中 SQL 语句不会存在需要转义处理的情况，可以将该属性设为 false，在一定程度上提交操作效率；取值（true/True，false/False）	否
“autoCommit”	是否自动提交，默认 true；取值（true/True，false/False）	否
“maxRows”	批量操作最大行数，默认 0	否
“rowPrefetch”	预取行数，默认 10	否
“LobMode”	Lob 模式，默认 1；取值（1 分批缓存到本地,2 一次将大字段数据缓存到本地）	否
“StmtPoolSize”	语句句柄池大小，默认 15	否
“ignoreCase”	是否忽略大小写，默认 true，取值（true/True，false/False）	否
“alwayseAllowCommit”	在自动提交开关打开时，是否允许手动提交回滚	否
“batchType”	批处理类型，默认 1，取值（1 进行批量绑定 2 不进行批量绑定）	否
“maxCachedPstmtSize”	缓存准备执行语句句柄的数量，默认 0	否
“appName”	客户端应用程序名称	否
“sessionTimeout”	会话超时时间，默认 0	否
“isCompress”	是否压缩消息，默认 false，取值（true/True，false/False）	否

“sslFilesPath”	指定 ssl 加密文件的路径	否
“sslKeystorePass”	指定 ssl 加密文件的指令	否
“resultSetType”	指定默认创建结果集类型，1003—readonly，1004—unsensitive，1005—insensitive，默认 1003	否
“kerberosLoginConfPath”	Kerberors 认证登录配置文件路径	否
“mppLocal”	是否 MPP 本地连接，默认 false；取值（true/True，false/False）	否
“rwSeparate”	是否使用读写分离系统，默认 false；取值（false 不使用，true 使用）	否
“rwPercent”	分发到主库的事务占主备库总事务的百分比，有效值 0~100，默认 25	否
“dbmdChkPrv”	编目函数是否进行权限检测，默认 false；取值（true/True，false/False）	否
“isBdtaRS”	是否使用列模式结果集，默认 true；取值（true/True，false/False）	否
“wellDistributed”	配置的连接服务名包含多个服务器时，连接是否均匀分布，默认 false；取值（true/True，false/False）	否
“uKeyName”	Ukey 的用户名	否
“uKeyPin”	Ukey 的口令	否
“doSwitch”	若使用服务名方式登录，且服务名配置了多个 ip，当连接发生异常时是否自动切换；默认 false；取值（true/True，false/False）	否
“clobAsString”	clob 类型列调用 resultSetMetaData 的 getColumnType()映射为 Types.VARCHAR 类型；默认 false；取值（true/True，false/False）	否
“continueBatchOnError”	批量执行出错时是否继续执行；默认 false；取值（true/True，false/False）	否
“connectTimeout”	连接数据库超时时间，单位 ms， 默认 0	否
“columnNameUpperCase”	列名转换为大写字母，默认 false；取值（true/True，false/False）	否
“loadBalance”	是否开启负载均衡，默认 false；取值（true/True，false/False）；使用服务名方式建立连接，开启后可以将连接均匀分配到服务名关联的各个数据库实例上，并一直保持各数据库实例负载均衡	否
“loadBalanceFreq”	各站点会话统计频率，值越小负载切换越及时，单位 ms，有效值范围 1~2147483647，默认 60000（1min）	否
“loadBalancePercent”	界定各站点是否均衡，值越小均衡度越好，有效值范围 1~100，默认 10	否
“rwAutoDistribute”	读写分离系统事务分发是否由 JDBC 自动管理，默认 true，取值（true/True，false/False）。false：事务分发由用户管理，用户可通过设置连接上的 readOnly 属性标记事务为只读事务	否
“compatibleMode”	兼容其他数据库，属性值为数据库名称（例如：oracle），支持兼容 oracle 和 mysql	否
“dbAliveCheckFreq”	检测数据库是否存活的频率，单位 ms，默认 0；0：不检测	否
“logDir”	日志等其他一些 JDBC 过程文件生成目录，默认值是当前工	否

	作目录	
“logLevel”	生成日志的级别，日志按从低到高依次如下（off：不记录；error：只记录错误日志；warn：记录警告信息；sql：记录 sql 执行信息；info：记录全部执行信息；all：记录全部），高级别同时记录低级别的信息	否
“logFlushFreq”	日志刷盘频率，单位 s，默认 60	否
“logBufferSize”	日志缓冲区大小(每个缓冲区可以存放日志的条数)，默认 1000	否
“logBufferPoolSize”	日志缓冲池大小(缓冲池中包含日志缓冲区的个数)，默认 3	否
“logFlusherQueueSize”	日志刷盘线程中等待刷盘的日志缓冲区队列大小，默认 100	否
“statEnable”	是否启用状态监控，默认 false；取值（true/True，false/False）	否
“statFlushFreq”	状态监控统计信息刷盘频率，单位 s；默认 10	否
“statSlowSqlCount”	日志打印慢 sql top 行数，默认 100；有效值范围 0~1000	否
“statHighFreqSqlCount”	日志打印高频 sql top 行数，默认 100；有效值范围 0~1000	否
“statSqlMaxCount”	状态监控可以统计不同 sql 的个数，默认 100000；有效值范围 0~100000	否
“statSqlRemoveMode”	执行的不同 sql 个数超过 statSqlMaxCount 时使用的淘汰方式，取值（latest/eldest）；latest 淘汰最近执行的 sql，eldest 淘汰最老的 sql	否
“statDir”	状态监控信息以文本文件形式输出的目录，无默认值，若不指定则监控信息不会以文本文件形式输出	否

4.6 Statement/PreparedStatement/CallableStatement

4.6.1 Statement

1. 概述

Statement 对象用于将 SQL 语句发送到数据库服务器。DM JDBC 提供三种类型的语句对象：Statement，PreparedStatement，CallableStatement。其中 PreparedStatement 是 Statement 的子类，CallableStatement 是 PreparedStatement 的子类。每一种语句对象用来运行特定类型的 SQL 语句。

Statement 对象用来运行简单类型的 SQL 语句，语句中无需指定参数。

PreparedStatement 对象用来运行包含（或不包含）IN 类型参数的预编译 SQL 语句。

CallableStatement 对象用来调用数据库存储过程。

2. 创建 Statement 对象

建立连接后，Statement 对象用 Connection 对象的 createStatement 方法创建，以下代码创建 Statement 对象：

```
Connection con = DriverManager.getConnection(url, "SYSDBA", "SYSDBA");
Statement stmt = con.createStatement();
```

3. 使用 Statement 对象执行语句

Statement 接口提供了三种执行 SQL 语句的方法：executeQuery、executeUpdate 和 execute。

方法 executeQuery 用于产生单个结果集的语句，例如 SELECT 语句。

方法 executeUpdate 用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL 语句，如 CREATE TABLE 和 DROP TABLE。INSERT、UPDATE 或 DELETE 语句的效果是修改表中零行或多行中的一列或多列。executeUpdate 的返回值是一个整数，表示受影响的行数。对于 CREATE TABLE 或 DROP TABLE 等 DDL 语句，executeUpdate 的返回值总为零。

方法 execute 用于执行返回多个结果集、多个更新元组数或二者组合的语句。

执行语句的三种方法都将关闭所调用的 Statement 对象的当前打开结果集（如果存在）。这意味着在重新执行 Statement 对象之前，需要完成对当前 ResultSet 对象的处理。

4. 关闭 Statement 对象

Statement 对象可由 Java 垃圾收集程序自动关闭。但作为一种良好的编程风格，应在不需要 Statement 对象时显式地关闭它们。这将立即释放数据库服务器资源，有助于避免潜在的内存问题。

5. 性能优化调整

1) 批处理更新

DM JDBC 驱动程序提供批处理更新的功能，通过批处理更新可以一次执行一个语句集合。JDBC 提供了三个方法来支持批处理更新：通过 addBatch 方法，向批处理语句集中增加一个语句；通过 executeBatch 执行批处理更新；通过 clearBatch 来清除批处理语句集。

批处理更新过程中，如果出现执行异常，DM 将立即退出批处理的执行，同时返回已经被执行语句的更新元组数(在自动提交模式下)。

推荐批处理更新在事务的非自动提交模式下执行。同时，批处理更新成功后，需要主动提交以保证事务的永久性。

2) 性能优化参数设置

DM JDBC 驱动程序提供了 setFetchDirection、setFetchSize 来向驱动程序暗示用户操作语句结果集的缺省获取方向和一次获取的缺省元组数。这些值的设定可以为驱动程序优化提供参考。

6. 语句对象

JDBC 在创建语句对象时，可以指定语句对象的缺省属性：结果集类型和结果集并发类型。DM JDBC 驱动程序支持 TYPE_FORWARD_ONLY 和 TYPE_SCROLL_INSENSITIVE 两种结果集类型，不支持 TYPE_SCROLL_SENSITIVE 结果集类型；支持 CONCUR_READ_ONLY、CONCUR_UPDATABLE 两种结果集并发类型。

语句对象的缺省属性用来指定执行语句产生的结果集的缺省类型，其具体含义和用法参见 4.7 ResultSet 节中“结果集增强特性部分”。

注意：

DM JDBC 驱动程序中当执行的查询语句涉及多个基表时，结果集不能更新，结果集的类型可能被自动转换为 CONCUR_READ_ONLY 类型。

4.6.2 PreparedStatement

1. 概述

PreparedStatement 继承 Statement，并与之在两方面有所不同：

1) PreparedStatement 对象包含已编译的 SQL 语句，语句已经“准备好”；
2) 包含于 PreparedStatement 对象中的 SQL 语句可具有一个或多个 IN 参数。IN 参数的值在 SQL 语句创建时未被指定。相反，该语句为每个 IN 参数保留一个问号（“?”）作为占位符。每个问号所对应的值必须在该语句执行之前，通过适当的 setXXX 方法来提供。

由于 PreparedStatement 对象已预编译过，所以其执行速度要快于 Statement 对象。因此，需要多次重复执行的 SQL 语句经常创建为 PreparedStatement 对象，以提高效率。

作为 Statement 的子类，PreparedStatement 继承了 Statement 的所有功能。另外它还添加了一整套方法，用于设置发送给数据库以取代 IN 参数占位符的值。同时，三种方法 execute、executeQuery 和 executeUpdate 能执行设置好参数的语句对象。

2. 创建 PreparedStatement 对象

以下的代码段（其中 con 是 Connection 对象）创建一个 PreparedStatement 对象：

```
PreparedStatement pstmt = con.prepareStatement(
"UPDATE person.person SET name = ? WHERE personid = ?");
```

对象 pstmt 包含语句 "UPDATE person.person SET name = ? WHERE personid = ?"，该语句带两个 IN 参数占位符，它已发送给数据库，并由服务器为其执行作好了准备。

3. 传递 IN 参数

在执行 PreparedStatement 对象之前，必须设置占位符(“?”)的值。这可通过调用 setXXX 方法来完成，其中 XXX 是与该参数相应的类型。例如，如果参数具有 Java 类型 String，则使用的方法就是 setString。setXXX 方法的第一个参数是要设置的参数的序号（从 1 算起），第二个参数是设置给该参数的值。譬如，以下代码将第一个参数设为“张三”，第二个参数设为“18”：

```
pstmt.setString(1, "张三");
pstmt.setInt(2, 18);
```

每当设置了给定语句的参数值，就可执行该语句。设置一组新的参数值之前，应先调用 clearParameters 方法清除原先设置的参数值。

4. 使用 setObject

setObject 方法可显式地将输入参数转换为特定的 JDBC 类型。该方法可以接受三个参数，其中第三个参数用来指定目标 JDBC 类型。将 Java Object 发送给数据库之前，驱动程序将把它转换为指定的 JDBC 类型。例如，上面的 setXXX 语句就可以改写为：

```
pstmt.setObject(1, "张三", java.sql.Types.VARCHAR);
pstmt.setObject(2, 18, java.sql.Types.INTEGER);
```

如果没有指定 JDBC 类型，驱动程序就会将 Java Object 映射到其缺省的 JDBC 类型，然后将它发送到数据库，这与常规的 setXXX 方法类似。在这两种情况下，驱动程序在将值发送到数据库之前，会将该值的 Java 类型映射为适当的 JDBC 类型。二者的差别在于 setXXX 方法使用从 Java 类型到 JDBC 类型的标准映射，而 setObject 方法使用从 Java Object 类型到 JDBC 类型的映射。

方法 setObject 允许接受所有 Java 对象，这使应用程序更为通用，并可在运行时接受参数的输入。这样，如果用户在编辑应用程序时不能确定输入类型，可以通过使用 setObject，对应用程序赋予可接受的 Java 对象，然后由 JDBC 驱动程序自动将其转换成数据库所需的 JDBC 类型。但如果用户已经清楚输入类型，使用相应的 setXXX 方法是值得推荐的，可以提高效率。

5. 将 JDBC NULL 作为 IN 参数发送

setNull 方法允许程序员将 JDBC NULL 值作为 IN 参数发送给数据库。在这种情况下，可以把参数的目标 JDBC 类型指定为任意值，同时参数的目标精度也不再起作用。

6. 发送大的 IN 参数

setBytes 和 setString 方法能够发送无限量的数据。但是，内存要足够容纳相关数据。有时程序员更喜欢用较小的块传递大型的数据，这可通过将 IN 参数设置为 Java 输入流来完成。当语句执行时，JDBC 驱动程序将重复调用该输入流，读取其内容并将它们当作实际参数数据传输。

JDBC 提供了四种将 IN 参数设置为输入流的方法：setBinaryStream 用于字节流，setAsciiStream 用于 ASCII 字符流，setUnicodeStream 用于 Unicode 字符流，从 JDK1.2 起，输入字符流的新方法为 setCharacterStream，而 setAsciiStream 和 setUnicodeStream 已经很少用。

7. 获得参数元数据

DM JDBC 实现了 getParameterMetaData()方法，通过这个方法可以获得有关 IN 参数的各种属性信息，比如类型、精度、刻度等信息，类似于结果集元数据的内容。通过这些信息，用户可以更准确地设置 IN 参数的值。

在下面的代码中涉及到了这种方法：

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM person.person " + "WHERE personid = ?");
...
//获得参数元数据对象
ParameterMetaData pmd = pstmt.getParameterMetaData();
//获得参数的个数
int paramCount = pmd.getParameterCount();
//获得第一参数的类型
int colType = pmd.getParameterType(1);
…
```

8. 自定义方法列表

为了实现对达梦数据库所提供的时间间隔类型和带纳秒的时间类型的支持，在实现 PreparedStatement 接口的过程中，增加了一些自定义的扩展方法。用户将获得的 PreparedStatement 对象反溯成 DmdbPreparedStatment 类型就可以访问这些方法。这些方法所涉及到的扩展类请参看 4.4 节。

表 4.2 自定义方法列表

方法名	功能说明
setINTERVALYM	设置参数为年-月时间间隔类型值
setINTERVALDT	设置参数为日-时时间间隔类型值
setTIME	设置参数为时间类型(带纳秒)

例如，想要插入一个年月时间间隔类型，代码如下（pstmt 为 PreparedStatement 类型）：

```
DmdbPreparedStatement dmps = (DmdbPreparedStatement)pstmt;
DmdbIntervalYM dmiym = new DmdbIntervalYM("interval '20-10' year to month");
dmps.setINTERVALYM(1, dmiym);
```

另外，DM 中不支持 JDBC 规范中所要求的标准方法 setURL 和 setRef。

4.6.3 CallableStatement

1. 概述

CallableStatement 用来运行 SQL 存储过程。存储过程是数据库中已经存在的 SQL 语句，它通过名字调用。

CallableStatement 是 PreparedStatement 的子类。CallableStatement 中定义的方法用于处理 OUT 参数或 INOUT 参数的输出部分：注册 OUT 参数的 JDBC 类型（一般 SQL 类型）、从这些参数中检索结果，或者检查所返回的值是否为 JDBC NULL。

2. 创建 CallableStatement 对象

CallableStatement 对象是用 Connection.prepareCall 创建的。以下代码创建 CallableStatement 对象，其中含有对存储过程 p1 的调用，con 为连接对象：

```
CallableStatement cstmt = con.prepareCall("call p1(?, ?)");
```

其中"?"占位符为 IN、OUT 还是 INOUT 参数，取决于存储过程 p1。

3. IN 和 OUT 参数

将 IN 参数传给 CallableStatement 对象是通过 setXXX 方法完成的。该方法继承自 PreparedStatement。所传入参数的类型决定了所用的 setXXX 方法（例如，用 setString 来传入 String 值等）。

如果存储过程返回 OUT 参数，则在执行 CallableStatement 对象之前必须先注册每个 OUT 参数的 JDBC 类型，有的参数还要同时提供刻度。注册 JDBC 类型是用 registerOutParameter 方法来完成的。语句执行完后，CallableStatement 的 getXXX 方法将取回参数值。其中 XXX 是为各参数所注册的 JDBC 类型所对应的 Java 类型。换言之， registerOutParameter 使用的是 JDBC 类型（因此它与数据库返回的 JDBC 类型匹配），而 getXXX 将之转换为 Java 类型。

设存储过程 p1 的定义如下：

```
CREATE OR REPLACE PROCEDURE p1( a1 IN VARCHAR(10), a2 OUT VARCHAR(50)) AS
DECLARE
CURSOR CUR1 FOR SELECT name FROM person.person WHERE personid = a1;
BEGIN
OPEN CUR1;
FETCH CUR1 INTO a2;
END;
```

以下代码先注册 OUT 参数，执行由 cstmt 所调用的存储过程，然后检索通过 OUT 参数返回的值。方法 getString 从 OUT 参数中取出字符串：

```
CallableStatement cstmt = con.prepareCall("call p1(?, ?)");
cstmt.setString(1, "1");
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);
cstmt.executeUpdate();
```

```
String x = cstmt.getString(2);
```

4. INOUT 参数

如果参数为既接受输入又接受输出的参数类型（INOUT 参数），那么除了调用 registerOutParameter 方法外，还要调用对应的 setXXX 方法（继承自 PreparedStatement）。setXXX 方法将参数值设置为输入参数，而 registerOutParameter 方法将它的 JDBC 类型注册为输出参数。setXXX 方法提供一个 Java 值，驱动程序先把这个值转换为 JDBC 值，然后将它送到数据库服务器。

该 IN 值的 JDBC 类型和提供给 registerOutParameter 方法的 JDBC 类型应该相同。如果要检索输出值，就要用对应的 getXXX 方法。

设有一个存储过程 p2 的定义如下：

```
CREATE OR REPLACE PROCEDURE p2(a1  IN OUT  VARCHAR(10)) AS
DECLARE
CURSOR CUR1 FOR SELECT name FROM person.person WHERE personid = a1;
BEGIN
OPEN CUR1;
FETCH CUR1 INTO a1;
END;
```

以下代码中，方法 setString 把参数设为“1”。然后，registerOutParameter 将该参数注册为 JDBC VARCHAR。执行完该存储过程后，将返回一个新的 JDBC VARCHAR 值。方法 getString 将把这个新值作为 Java 的 String 类型返回。

```
CallableStatement cstmt = con.prepareCall("call p2(?)");
cstmt.setString(1, "1");
cstmt.registerOutParameter(1, java.sql.Types.VARCHAR);
cstmt.executeUpdate();
String x = cstmt.getString(1);
```

5. 利用参数名进行操作

在通常的情况下一般采用参数索引来进行赋值。JDBC3.0 规范要求可以利用参数名来对参数进行赋值，DM JDBC 驱动程序实现了这一点。如果某个存储过程的一些参数有默认值，这时候采用参数名进行赋值就非常有用，用户可以只对那些没有默认值的参数进行赋值。参数名可以通过调用 DatabaseMetaData.getProcedureColumns()来获得。

下面的代码中，存储过程 p2 为上面那个存储过程 p2。利用参数名进行操作：

```
CallableStatement cstmt = con.prepareCall("CALL p2 (?)");
cstmt.setString("a1", "1");
cstmt.registerOutParameter("a1", java.sql.Types.VARCHAR);
cstmt.executeUpdate();
String x = cstmt.getString("a1");
```

而且，在读取参数的值和进行参数注册的时候，setXXX、getXXX、registerOutParameter 也都可以采用参数名来进行操作。通过调用 DatabaseMetaData.supportsNamedParameters()方法就可以确定 JDBC 驱动程序是否支持利用参数名来进行操作。

注意：

在执行同一条语句的过程中，不允许交叉使用参数索引和参数名来进行操作，否则会抛出异常。

6. 自定义方法列表

为了实现对达梦数据库所提供的时间间隔类型和带纳秒的时间类型的支持，在实现 CallableStatement 接口的过程中，增加了一些自定义的扩展方法。用户将获得的 CallableStatement 对象反溯成 DmdbCallableStatment 类型就可以访问这些方法。这些方法所涉及到的扩展类请参看 4.4 DM JDBC 扩展这一节。

表 4.3 自定义方法列表

方法名	功能说明
getINTERVALYM(int)	获得参数的值
getINTERVALYM(String)	获得参数的值
getINTERVALDT(int)	获得参数的值
getINTERVALDT(String)	获得参数的值
getTIME(int)	获得参数的值
getTIME(String)	获得参数的值
setTIME(String,String)	根据参数名来设置 DmdbTime 类型值。
setINTERVALDT(String,String)	根据参数名来设置 DmdbIntervalDT 类型值。
setINTERVALYM(String,String)	设置参数为年-月时间间隔类型值

另外，DM 中不支持 JDBC 规范中所要求的标准方法 setURL、getURL 和 getRef。由于 CallableStatement 是 PreparedStatement 的子类，因此，它将继承 PreparedStatement 的所有的方法。

4.7 ResultSet

1. 概述

ResultSet 提供执行 SQL 语句后从数据库返回结果中获取数据的方法。执行 SQL 语句后数据库返回结果被 JDBC 处理成结果集对象，可以用 ResultSet 对象的 next 方法以行为单位进行浏览，用 getXXX 方法取出当前行的某一列的值。

通过 Statement，PreparedStatement，CallableStatement 三种不同类型的语句进行查询都可以返回 ResultSet 类型的对象。

2. 行和光标

ResultSet 维护指向其当前数据行的逻辑光标。每调用一次 next 方法，光标向下移动一行。最初它位于第一行之前，因此第一次调用 next 将把光标置于第一行上，使它成为当前行。随着每次调用 next 导致光标向下移动一行，按照从上至下的次序获取 ResultSet 行。

在 ResultSet 对象或对应的 Statement 对象关闭之前，光标一直保持有效。

3. 列

方法 getXXX 提供了获取当前行中某列值的途径。在每一行内，可按任何次序获取列值。

列名或列号可用于标识要从中获取数据的列。例如，如果 ResultSet 对象 rs 的第二列名为“title”，则下列两种方法都可以获取存储在该列中的值：

```
String s = rs.getString("title");
String s = rs.getString(2);
```

注意列是从左至右编号的，并且从 1 开始。

在 DM JDBC 驱动程序中，如果列的全名为“表名.列名”的形式。在不引起混淆的情况下（结果集中有两个表具有相同的列名），可以省略表名，直接使用列名来获取列值。

关于 ResultSet 中列的信息，可通过调用方法 ResultSet.getMetaData 得到。返回的 ResultSetMetaData 对象将给出其 ResultSet 对象各列的名称、类型和其他属性。

4. NULL 结果值

要确定给定结果值是否是 JDBC NULL，必须先读取该列，然后使用 ResultSet 对象的 wasNull 方法检查该次读取是否返回 JDBC NULL，如下：

```
String sql="select * from person.person";
ResultSet rs;
rs = stmt.executeQuery(sql);
while (rs.next()){
    if(rs.wasNull())
        System.out.println("Get A Null Value");
}
```

当使用 ResultSet 对象的 getXXX 方法读取 JDBC NULL 时，将返回下列值之一：

1) Java null 值：对于返回 Java 对象的 getXXX 方法（如 getString、getBigDecimal、getBytes、getDate、getTime、getTimestamp、getAsciiStream、getUnicodeStream、getBinaryStream、getObject 等）；
2) 零值：对于 getByte、getShort、getInt、getLong、getFloat 和 getDouble；
3) false 值：对于 getBoolean。

5. 结果集增强特性

在 DM JDBC 驱动程序中提供了符合 JDBC 2.0 标准的结果集增强特性：可滚动、可更新结果集，及 JDBC3.0 标准的可持有性。

1) 结果集的可滚动性

通过执行语句而创建的结果集不仅支持向前（从第一行到最后一行）浏览内容，而且还支持向后（从最后一行到第一行）浏览内容的能力。支持这种能力的结果集被称为可滚动的结果集。可滚动的结果集同时也支持相对定位和绝对定位。绝对定位指的是通过指定在结果集中的绝对位置而直接移动到某行的能力，而相对定位则指的是通过指定相对于当前行的位置来移动到某行的能力。DM 支持可滚动的结果集。

DM JDBC 驱动程序中支持只向前滚结果集(ResultSet.TYPE_FORWARD_ONLY)和滚动不敏感结果集(ResultSet.TYPE_SCROLL_INSENSITIVE)两种结果集类型，不支持滚动敏感结果集(ResultSet.TYPE_SCROLL_SENSITIVE)。当结果集为滚动不敏感结果集时，它提供所含基本数据的静态视图，即结果集中各行的成员顺序、列值通常都是固定的。

2) 结果集的可更新性

DM JDBC 驱动程序中提供了两种结果集并发类型：只读结果集(ResultSet.CONCUR_READ_ONLY)和可更新结果集(ResultSet.CONCUR_UPDATABLE)。采用只读并发类型的结果集不允许对其内容进行更新。可更新的结果集支持结果集的更新操

作。

3) 结果集的可持有性

JDBC 3.0 提供了两种结果集可持有类型：

提交关闭结果集(ResultSet.CLOSE_CURSORS_AT_COMMIT)和跨结果集提交(ResultSet.HOLD_CURSORS_OVER_COMMIT)。采用提交关闭结果集类型的结果集在事务提交之后被关闭，而跨结果集提交类型的结果集在事务提交之后仍能保持打开状态。

通过 DatabaseMetaData.supportsHoldability()方法可以确定驱动程序是否支持结果集的可持有性。目前 DM 支持这两种类型。

4) 性能优化

DM JDBC 驱动程序的结果集对象中提供了方法 setFetchDirection 和 setFetchSize 来设置缺省检索结果集的方向和缺省一次从数据库获取的记录条数。它们的含义与用法和语句对象中的同名函数是相同的。

6. 更新大对象数据

从 DM JDBC 2.0 驱动程序就支持可更新的结果集，但是对 LOB 对象只能读取，而不能更新，这也是 JDBC 2.0 标准所规定的。而 JDBC 3.0 规范规定用户可以对 LOB 对象进行更新，DM JDBC 3.0 驱动程序中实现了这一点：

假设数据库中存在 BOOKLIST 表，且含有 id、comment 两个字段，建表语句如下：

CREATE TABLE BOOKLIST ("ID" INTEGER,"COMMENT" TEXT);

INSERT INTO SYSDBA.BOOKLIST VALUES (1, '测试数据');

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("select comment from booklist " +
                                 "where id = 1");
rs.next();
Clob commentClob = new Clob(...);
rs.updateClob("author", commentClob);   // commentClob is a Clob Object
rs.updateRow();
```

7. 自定义方法列表

为了实现对达梦数据库所提供的时间间隔类型和带纳秒的时间类型的支持，在实现 ResultSet 接口的过程中，增加了一些自定义的扩展方法。用户将获得的 ResultSet 对象反溯成 DmdbResultSet 类型就可以访问这些方法。这些方法所涉及到的扩展类请参看 4.4 节。

表 4.4 自定义方法列表

方法名	功能说明
getTime(int)	根据列号(1 开始)获取时间信息，以 java.sql.Time 类型返回。
getTime(int, Calendar)	根据列号(1 开始)、Calendar 对象获取时间信息， 以 java.sql.Time 类型返回。
getTime(String)	根据列名获取时间信息，以 java.sql.Time 类型返回。
getTime(String, Calendar)	根据列名、Calendar 对象获取时间信息， 以 java.sql.Time 类型返回。
getTimestamp(int)	根据列号(1 开始)获取时间信息，以 java.sql.Timestamp 类型返回。
getTimestamp(int, Calendar)	根据列号(1 开始)、Calendar 对象获取时间信息， 以 java.sql.Timestamp 类型返回。
getTimestamp(String)	根据列名获取时间信息，以 java.sql.Timestamp 类型返回。

getTimestamp(String, Calendar)	根据列名、Calendar 对象获取时间信息， 以 java.sql.Timestamp 类型返回。
updateINTERVALYM(int, DmdbIntervalYM)	设置列为年-月时间间隔类型值
updateINTERVALYM(String, DmdbIntervalYM)	设置列为年-月时间间隔类型值
updateINTERVALDT(int, DmdbIntervalDT)	设置列为日-时时间间隔类型值
updateINTERVALDT(String, DmdbIntervalDT)	设置列为日-时时间间隔类型值

另外，DM 中不支持 getURL 和 getRef 方法。

4.8 流与大对象

4.8.1 Stream 使用

为了获取大数据量的列，DM JDBC 驱动程序提供了四个获取流的方法：

1. getBinaryStream 返回只提供数据库原字节而不进行任何转换的流；
2. getAsciiStream 返回提供单字节 ASCII 字符的流；
3. getUnicodeStream 返回提供双字节 Unicode 字符的流；
4. getCharacterStream 返回提供双字节 Unicode 字符的 java.io.Reader 流。

在这四个函数中，JDBC 规范不推荐使用 getUnicodeStream 方法，其功能可以用 getCharacterStream 代替。以下是采用其它三种方法获取流的示例代码：

1. 采用 getBinaryStream 获取流

```
// 查询语句
String sql = "SELECT description FROM production.product WHERE productid=1";
// 创建语句对象
Statement stmt = conn.createStatement();
// 执行查询
ResultSet rs = stmt.executeQuery(sql);
// 显示结果集
while (rs.next()) {
    try {
        InputStream stream = rs.getBinaryStream("description");
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int num = -1;
        while ((num = stream.read()) != -1) {
            baos.write(num);
        }
        System.out.println(baos.toString());
    } catch (IOException e) {
        e.printStackTrace();
```

```
        }
}
// 关闭结果集
rs.close();
// 关闭语句
stmt.close();
```

2. 采用 getAsciiStream 获取流

```
// 查询语句
String sql = "SELECT description FROM production.product WHERE productid=1";
// 创建语句对象
Statement stmt = conn.createStatement();
// 执行查询
ResultSet rs = stmt.executeQuery(sql);
// 显示结果集
while (rs.next()) {
        try {
                InputStream stream = rs.getAsciiStream("description");
                StringBuffer desc = new StringBuffer();
                byte[] b = new byte[1024];
                for (int n; (n = stream.read(b)) != -1;) {
                        desc.append(new String(b, 0, n));
                }
                System.out.println(desc.toString());
        } catch (IOException e) {
                e.printStackTrace();
        }
}
// 关闭结果集
rs.close();
// 关闭语句
stmt.close();
```

3. 采用 getCharacterStream 获取流

```
// 查询语句
String sql = "SELECT description FROM production.product WHERE productid=1";
// 创建语句对象
Statement stmt = conn.createStatement();
// 执行查询
ResultSet rs = stmt.executeQuery(sql);
// 显示结果集
while (rs.next()) {
        try {
                Reader reader = rs.getCharacterStream("description");
                BufferedReader br = new BufferedReader(reader);
```

```
            String thisLine;
            while ((thisLine = br.readLine()) != null) {
                System.out.println(thisLine);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
}
// 关闭结果集
rs.close();
// 关闭语句
stmt.close();
```

4.8.2 LOB 对象使用

1. 概述

JDBC 标准为了增强对大数据对象的操作，在 JDBC 3.0 标准中增加了 java.sql.Blob 和 java.sql.Clob 这两个接口。这两个接口定义了许多操作大对象的方法，通过这些方法就可以对大对象的内容进行操作。

2. 产生 Lob 对象

在 ResultSet 和 CallableStatement 对象中调用 getBlob()和 getClob()方法就可以获得 Blob 对象和 Clob 对象：

```
Blob blob = rs.getBlob(1);
Clob clob = rs.getClob(2);
```

3. 设置 Lob 对象

Lob 对象可以像普通数据类型一样作为参数来进行参数赋值，在操作 PreparedStatement、CallableStatement、ResultSet 对象时使用：

```
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO bio (image, text) " + "VALUES (?, ?)");
//authorImage is a Blob Object
pstmt.setBlob(1, authorImage);
//authorBio is a Clob Object
pstmt.setClob(2, authorBio);
```

在一个可更新的结果集中，也可以利用 updateBlob(int,Blob)、updateBlob(String,Blob)和 updateClob(int,Clob)、updateClob(String,Clob)来更新当前行。

4. 改变 Lob 对象的内容

Lob 接口提供了方法让用户可以对 Lob 对象的内容进行任意修改：

```
byte[] val = {0,1,2,3,4};
```

```
Blob data = rs.getBlob("DATA");
int numWritten = data.setBytes(1, val); // 在指定的位置插入数据
PreparedStatement ps = conn.prepareStatement("UPDATE datatab SET data = ?");
ps.setBlob("DATA", data);
ps.executeUpdate();
```

注意：

目前 LOB 内容的更新和其当前所处的事务之间没有直接的联系。更新 LOB 时，会直接写入到数据库中，即便当前事务最终回滚也不能恢复。

4.9 元数据

4.9.1 ResultSetMetaData

1. 概述

ResultSetMetaData 提供许多方法，用于读取 ResultSet 对象返回数据的元信息。包括：列名、列数据类型、列所属的表、以及列是否允许为 NULL 值等，通过这些方法可以确定结果集中列的一些信息。

2. 创建结果集元数据对象

结果集元数据是用来描述结果集的特征，所以，需要首先执行查询获得结果集，才能创建结果集元数据对象。

3. 创建 ResultSetMetaData 对象如下例所示：

假如有一个表 TESTTABLE(no int,name varchar(10))，利用下面的代码就可以知道这个表的各个列的类型：

```
ResultSet rs = stmt.executeQuery("SELECT * FROM TESTTABLE");
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i ++)
{
    String typeName = rsmd.getColumnTypeName(i);
    System.out.println("第" + i + "列的类型为：" + typeName);
}
```

4.9.2 DatabaseMetaData

1. 概述

DatabaseMetaData 提供了许多方法，用于获取数据库的元数据信息。包括：描述数据库特征的信息(如是否支持多个结果集)、目录信息、模式信息、表信息、表权限信息、表列信息、存储过程信息等。DatabaseMetaData 有部分方法以 ResultSet 对象的形式返回结果，可以用 ResultSet 对象的 getXXX()方法获取所需的数据。

2. 创建数据库元数据对象

数据库元数据对象由连接对象创建。以下代码创建 DatabaseMetaData 对象（其中 con 为连接对象）：

```
DatabaseMetaData dbmd = con.getMetaData();
```

利用该数据库元数据对象就可以获得一些有关数据库和 JDBC 驱动程序的信息：

```
String databaseName = dbmd.getDatabaseProductName();  // 数据库产品的名称
int majorVersion = dbmd.getJDBCMajorVersion();  // JDBC 驱动程序的主版本号
String []types ={"TABLE"};
ResultSet tablesInfor = dbmd.getTables(null,  null,  "*TE%",  types);
```

4.9.3 ParameterMetaData

1. 概述

参数元数据是 JDBC 3.0 标准新引入的接口，它主要是对 PreparedStatement、CallableStatement 对象中的占位符（“?”）参数进行描述，例如参数的个数、参数的类型、参数的精度等信息，类似于 ResultSetMetaData 接口。通过引入这个接口，就可以对参数进行较为详细、准确的操作。

2. 创建参数元数据对象

通过调用 PreparedStatement 或 CallableStatement 对象的 getParameterMetaData()方法就可以获得该预编译对象的 ParameterMetaData 对象：

```
ParameterMetaData pmd = pstmt.getParameterMetaData();
```

然后就可以利用这个对象来获得一些有关参数描述的信息：

```
    // 获取参数个数
int paraCount = pmd.getParameterCount();
for(int i = 1; i <= paraCount; i ++) {
// 获取参数类型
System.out.println("The Type of Parameter("+i+") is " + ptmt.getParameterType(i));
// 获取参数类型名
System.out.println("The Type Name of Parameter("+i+") is "
+ ptmt.getParameterTypeName(i));
// 获取参数精度
System.out.println("The Precision of Parameter("+i+") is " + ptmt.getPrecision(i));
// 获取参数是否为空
System.out.println("Parameter("+i+") is nullable? " + ptmt.isNullable(i));
}
```

4.10 RowSet

RowSet 接口扩展了标准 java.sql.ResultSet 接口。RowSetMetaData 接口扩展了 java.sql.ResultSetMetaData 接口。JDK 5.0 定义了 5 个标准的 JDBCRowSet 接口，DM 实现了其中的 CachedRowSet 和 JdbcRowSet。

RowSet 对象可以建立一个与数据源的连接并在其整个生命周期中维持该连接，在此情况下，该对象被称为连接的 Rowset。Rowset 还可以建立一个与数据源的连接，从其获取数据，然后关闭它。这种 Rowset 被称为非连接 Rowset。非连接 Rowset 可以在断开时更改其数据，然后将这些更改发送回原始数据源，不过它必须重新建立连接才能完成此操作。相比较 java.sql.ResultSet 而言，RowSet 的离线操作能够有效的利用计算机越来越充足的内存，减轻数据库服务器的负担，由于数据操作都是在内存中进行然后批量提交到数据源，灵活性和性能都有了很大的提高。RowSet 默认是一个可滚动，可更新，可序列化的结果集，而且它作为 JavaBeans，可以方便地在网络间传输，用于两端的数据同步。

4.10.1 CachedRowSet

CachedRowSet 是非连接的 RowSet，数据行均被缓冲至本地内存，但并未保持与数据库服务器的连接。DmJdbcDriver15.jar 和 DmJdbc16.jar 中 dm.jdbc.rowset.DmdbCachedRowSet 类是达梦对于接口 javax.sql.rowset.CachedRowSet 的实现。

1．使用 URL、用户名、密码和一个查询 SQL 语句作为设置属性，创建一个 DmdbCachedRowSet 对象。RowSet 使用 execute 方法完成 CachedRowSet 对象的填充。完成 execute 方法执行后，可以像使用 java.sql.ResultSet 对象方法一样，使用 RowSet 对象返回、滚动、插入、删除或更新数据。

```
/* 创建 DmdbCachedRowSet 对象示例 */
String sql = "SELECT productid,name,author FROM production.product";
CachedRowSet crs = new DmdbCachedRowSet();
crs.setUrl("jdbc:dm://localhost:5236");
crs.setUsername("SYSDBA");
crs.setPassword("SYSDBA");
crs.setCommand(sql);
crs.execute();
while (crs.next())
{
    System.out.println("productid:" + crs.getInt(1));
    System.out.println("name:" + crs.getString(2));
    System.out.println("author:" + crs.getString(3));
}
```

2、CachedRowSet 对象也可以通过调用 populate 方法，使用一个已经存在的 ResultSet 对象填充。完成填充后，便可以像操作 ResultSet 对象一样，返回、滚动、插入、删除或更新数据。

```
/* 使用 populate 方法填充代码片段 */
// 执行查询，获取 ResultSet 对象
```

```
String sql = "SELECT productid,name,author FROM production.product";
ResultSet rs = stmt.executeQuery(sql);

// 填充 CachedRowSet
CachedRowSet crs = new DmdbCachedRowSet();
crs.populate(rs);
```

3、其他功能特点

创建一个 CachedRowSet 的拷贝：

```
CachedRowSet copy = crs.createCopy();
```

创建一个 CachedRowSet 的共享：

```
CachedRowSet shard = crs.createShared();
```

4、CachedRowSet 限制

- 仅支持单表查询，且无连接操作；
- 因数据缓存在内存，故不支持大数据块；
- 连接属性，如事务隔离级等，不能在填充（执行 execute 或 populate）后设置，因为此时已经断开了与数据库服务器的连接，不能将这些属性设置到返回数据的同一个连接上。

4.10.2 JdbcRowSet

JdbcRowSet 是对 ResultSet 对象的封装，是连接的 RowSet。达梦关于 JdbcRowSet 的实现是 dm.jdbc.rowset.DmdbJdbcRowSet 类。DmdbJdbcRowSet 在 DmJdbcDriver15.jar 和 DmJdbcDriver16.jar 中实现标准接口 javax.sql.rowset.JdbcRowSet。

JdbcRowSet 在其生命期中始终保持着与数据库服务器的连接。其所有调用操作，均渗透进对 JDBC Connection、statement 和 ResultSet 调用。而 CachedRowSet 则不存在于打开数据库服务器的任何连接。

CachedRowSet 在其操作过程中不需要 JDBC 驱动的存在，而 JdbcRowSet 需要。但两者在填充 RowSet 和提交数据修改的过程中，均需 JDBC 驱动的存在。

```
/* 使用 JdbcRowSet 接口示例 */
String sql = "SELECT name,author,publisher FROM production.product";
JdbcRowSet jrs = new DmdbJdbcRowSet();
jrs.setUrl("jdbc:dm://localhost:5236");
jrs.setUsername("SYSDBA");
jrs.setPassword("SYSDBA");
jrs.setCommand(sql);
jrs.execute();
int numcolsSum = jrs.getMetaData().getColumnCount();
while (jrs.next()) {
    for (int i = 1; i <= numcolsSum; i++) {
        System.out.print(jrs.getString(i) + "\t");
    }
```

```
        System.out.println();
}
jrs.close();
```

4.11 分布式事务支持

DM 对分布式事务的支持是依据 X/OPEN 分布式事务处理模型 XA 规范实现的。

DM 数据库系统实现了 X/OPEN DTP 模型中的 RM 组件，通过 JDBC 接口与第三方 TM 工具配合完成分布式事务处理。JDBC 标准中，对 XA 协议进行了部分剪裁，仅支持 TM 对 RM 的单向调用，不允许 RM 向 TM 的动态注册。因此 DM 系统对 XA 协议的支持也仅为单向调用。

一个会话同时只能处理一个分支事务，一个分支事务同时也只能被一个会话绑定处理。

DM JDBC 接口提供了 dm.jdbc.xa 包，其中的类实现 JDBC 的 XA 标准接口，包括：

```
javax.sql.XADataSource
javax.sql.XAConnection
javax.transaction.xa. XAResource
javax.transaction.xa.Xid
```

4.11.1 XADataSource

1. 概述

XADataSource 是分布式连接的数据源，通过其可以获取分布式连接。

2. 获取分布式连接

XADataSource 提供了两个方法用于获取分布式连接：

```
public XAConnection getXAConnection() throws SQLException
public XAConnection getXAConnection(String user, String password)          throws SQLException
```

第一个方法取给数据源设置的用户名和口令进行连接并返回得到的分布式连接，第二个方法则可以给定用户名和口令。

4.11.2 XAConnection

1. 概述

XAConnection 是对分布式事务进行支持的对象。XAConnection 对象通常是以 XAResource 对象的方式被纳入到分布式事务的管理中。

2. 获取 XAResource 对象

XAConneciton 提供了 javax.transaction.xa.XAResource getXAResource() throws SQLException 方法用于获取该分布式连接对应的 XAResource 对象，这个 XAResource 对象将代表 XAConnection 对象参加分布式事务的管理。

4.11.3 XAResource

1．概述

XAResource 对象是代表 XAConnection 对象参与分布式事务管理的，真正对分布式事务进行操控的方法都在这个接口中实现。

2．主要方法介绍

1） public void start(Xid xid, int flags) throws XAException

该方法用于开始一个事务分支。第一个参数 xid 用于指定事务分支的 id，第二个参数 flags 用于指明开始这个事务分支的方式，其合法值及意义如下：

TMJOIN：这个标志指名该事务分支将被合并到之前的具有相同 xid 的事务分支上，如果资源管理器发现之前没有这个 xid 的事务分支，则抛出异常。

TMRESUME：这个标志指明该事务分支将重新开始之前用 TMSUSPEND 标志结束的一个事务分支。

TMNOFLAGS：当没有特殊操作而只是开始一个普通的事务分支时，将 flags 置为 TMNOFLAGS。

2） public void end(Xid xid, int flags) throws XAException

该方法用于结束一个事务分支。第一个参数 xid 用于指定事务分支的 id，第二个参数 flags 用于指明结束这个事务分支的方式，其合法值及意义如下：

TMSUSPEND：这个标志指明暂时终止该事务分支，之后该事务分支应该被用 TMRESUME 重新开始或者用 RMSUCCESS 或 TMFAIL 正式结束。

TMSUCCESS：这个标志指明该事务分支成功结束。

TMFAIL：这个标志指明这个事务已经失败，则这个事务分支被资源管理器标记为只能回滚。

3） public int prepare(Xid xid) throws XAException

该方法用来预提交一个事务分支。参数 xid 用于指定事务分支的 id。

可能的返回值有：

XA_OK：该标志表明预提交成功。

XA_RDONLY：该标志表明事务分支具有只读属性并已被提交。

错误码：预提交失败并返回具体的错误码对应不同的失败信息。

4） public void commit(Xid xid, boolean onePhase) throws XAException

该方法用来提交一个事务分支。第一个参数 xid 用于指定事务分支的 id。第二个参数 onePhase 用来指定是否进行一阶段提交。

5） public void rollback(Xid xid) throws XAException

该方法用来回滚一个事务分支。参数 xid 用户指定事务分支的 id。

6） public boolean isSameRM(XAResource xares) throws XAException

该方法用来判断指定的 xares 与当前 XAResource 对象是否是指向同一数据源的两个实例。

7） public Xid[] recover(int flag) throws XAException

该方法用于返回资源管理器中所有已被预提交的事务分支的 id。

flags 的合法值及意义为：

TMSTARTRSCAN | TMENDRSCAN：一次获取到所有的已预提交的事务分支的 id。

TMSTARTRSCAN：开始扫描并返回已预提交的事务分支的 id，如果资源管理器中

的已预提交事务分支过多，可能返回的不是全部。

TMNOFLAGS：用于继续扫描已预提交的事务分支的 id。使用这一标志时，之前应该已经用 TMSTARTRSCAN 开始了扫描。

TMENDRSCAN：这个标志表明在此次扫描并返回对应的已预提交的事务分支的 id 后，结束此次扫描。

8) public void forget(Xid xid) throws XAException

该方法用来“遗忘”一个自主完成的事务分支，这里的“遗忘”表示真正释放事务分支，之前即使一个事务分支已经自主完成，其事务分支对象仍然保留。对于非自主完成的事务分支使用此方法时将返回异常。

3．DM 扩展方法介绍

DmdbXAResource 在实现了标准接口定义的方法的基础上，还扩展实现了 heurCommit() 和 heurRollback()方法。

1) public void heurCommit(Xid xid) throws XAException
该方法用于自主提交一个事务分支。参数 xid 用于指定事务分支的 id。

2) public void heurRollback(Xid xid) throws XAException
该方法用于自主回滚一个事务分支。参数 xid 用于指定事务分支的 id。

4.11.4 Xid

1．概述

Xid 接口是 X/Open 事务标识符 XID 结构的 Java 映射，由全局事务格式 ID、全局事务 ID 和分支限定符构成。

2．使用介绍

DmdbXid 提供了构造函数 public DmdbXid(int fmtId, byte[] gTranId, byte[] bQual) throws XAException 供用户生成一个 Xid 实例。

同时 DmdbXid 也实现了 Xid 标准接口中的三个用于获取 XID 的格式标识符部分、XID 的全局事务标识符部分作为字节数组和 XID 的事务分支标识符部分作为字节数组的三个方法。

4.11.5 实例解析

以下是一个使用 JDBC 的 XA 接口用两阶段提交协议来提交一个事务分支的例子。

```
DmdbXADataSource xaDS;
XAConnection xaCon;
XAResource xaRes;
Xid xid;
Connection con;
Statement stmt;
int ret;


    DmdbXADataSource xaDS = new DmdbXADataSource();
```

```
        /* 获取分布式连接 */
xaCon = xaDS.getXAConnection("jdbc_user", "jdbc_password");

        /* 获取代表分布式连接的 XAResource 实例 */
xaRes = xaCon.getXAResource();
con = xaCon.getConnection();
stmt = con.createStatement();

xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
try {
        /* 开始一个事务分支 */
        xaRes.start(xid, XAResource.TMNOFLAGS);
        stmt.executeUpdate("insert into test_table values (100)");

        /* 结束一个事务分支 */
        xaRes.end(xid, XAResource.TMSUCCESS);

        /* 预提交该事务分支 */
        ret = xaRes.prepare(xid);

        if (ret == XAResource.XA_OK) {
                /* 提交事务分支 */
              xaRes.commit(xid, false);
           }
}
catch (XAException e) {
   e.printStackTrace();
}
finally {
   stmt.close();
   con.close();
   xaCon.close();
}
```

第 5 章 .NET Data Provider 编程指南

.NET Data Provider 是.NET Framework 编程环境下的数据库用户访问数据库的编程接口，用于连接到数据库、执行命令和检索结果。在数据源和代码之间创建了一个最小层，以便在不以功能为代价的前提下提高性能。

5.1 数据类型

.NET Framework 在 System.Data.DbType 中定义了.NET Framework 数据提供程序的字段、属性或 Parameter 对象的数据类型。DmProvider 数据类型就是 Dm.DmDbType 中定义的数据类型。

表 5.1 DmProvider 数据类型和.NET Framework 数据类型的对应关系

DbType 类型	DmProvider 类型	DbType 类型	DmProvider 类型
AnsiString	VarChar	Int32	Int32
AnsiStringFixedLength	VarChar	Int64	Int64
Binary	Binary	Object	Blob
Boolean	Bit	SByte	SByte
Byte	Byte	Single	Float
Currency	Decimal	String	VarChar
Date	Date	StringFixedLength	Char
DateTime	DateTime	Time	Time
Decimal	Decimal	UInt16	UInt16
Double	Double	UInt32	UInt32
Guid	VarChar	UInt64	UInt64
Int16	Int16	VarNumeric	XDEC

表 5.2 DM 建表语句支持类型与.NET Framework 中类型的对应关系

DM 建表语句类型	.NET 中对应的类型	DM 建表语句类型	.NET 中对应的类型
CHAR	typeof(String)	DOUBLE	typeof(Double)
VARCHAR	typeof(String)	BLOB	typeof(Byte[])
BIT	typeof(Boolean)	DATE	typeof(DateTime)
TINYINT	typeof(SByte)	TIME	typeof(DateTime)
SMALLINT	typeof(Int16)	TIMESTAMP	typeof(DateTime)
INT	typeof(Int32)	BINARY	typeof(Byte[])
BIGINT	typeof(Int64)	VARBINARY	typeof(Byte[])
DEC	typeof(Decimal)	CLOB	typeof(String)
REAL	typeof(Single)	INTERVAL	typeof(Object)

5.2 提供的对象和接口

DM .NET Provider 接口主要实现了 DmConnection、DmCommand、DmDataAdapter、DmDataReader、DmParameter、DmParameterCollection、DmTransaction、DmCommandBuilder、DmConnectionStringBuilder、DmClob 和 DmBlob 共 11 个对象。

5.2.1 DmConnection 对象

DmConnection 对象表示一个 DM 数据库打开的连接。

公共属性

ConnectionString：获取或设置用于连接 DM 数据库的字符串；

ConnectionTimeout：获取在尝试建立连接时终止尝试并生成错误之前所等待的时间；

Database：DM8 不再有 database 的概念，该属性将不再起任何作用；

DataSource：获取要连接的 DM 实例的名称；

ServerVersion：DM 不支持该属性；

State：获取连接的当前状态；

MppType ：MPP 连接属性，有效值为 DmMppType.LOGIN_MPP_LOCAL 、DmMppType.LOGIN_MPP_GLOBAL；

RwSeparate：是否读写分离，有效值为 true 或 false；

RwPercent：表示分发到主库的事务占主备库总事务的百分比，有效值范围：0~100，默认值为 25；

StmtPooling：是否启用句柄重用，有效值为 true 或 fasle；

PoolSize：句柄重用缓冲区的大小。

公共方法

DmConnection()：构造函数，初始化 DmConnection 的新实例；

DmConnection(string connectionString)：构造函数，以指定的连接串进行连接对象新实例的初始化；

BeginTransaction()：开始数据库事务；

BeginTransaction(IsolationLevel il)：以指定的隔离级别启动数据库事务；

ChangeDatabase()：DM8 不支持该操作，调用该函数将不产生任何影响；

Close()：关闭与数据库的连接；

CreateCommand()：创建并返回一个与 DM 关联的 DmCommand 对象；

GetSchema()：返回 DM 的数据源的全部元信息；

GetSchema(String) ：使用指定字符串的元信息名称返回 DM 元信息结果集；

GetSchema(String, String[]) ：使用指定字符串的元信息名以及表示限制值的指定字符串数组返回 DM 元信息结果集；

Open()：使用 ConnectionString 所指定的属性设置打开数据库连接。

连接串

公共属性 ConnectionString 是用于连接 DM 数据库的字符串，其格式为：

```
<属性名>=<属性值>{;<属性名>=<属性值>}
```

其中支持的属性名及其意义如下表所示：

属性名	意义
SERVER	服务名
LOGIN_PRIMARY	在主备情况下是否仅登录到主库或备库。0：主库不存在的情况下可连接备库；1：只连接主库；2：只连接备库；3：优先连接备库。默认为 0
PORT	登录端口号
USER	用户名
PASSWORD	用户口令
TIMEOUT	连接超时时间，默认 15s
COMMANDTIMEOUT	命令超时时间，默认 30s
APPNAME	应用名
TRACE	NONE：不记录 TRACE DEBUG：将 TRACE 内容打印到控制台 TRACE：将 TRACE 内容记录到运行路径下的 ProviderTrace.txt 文件中
PRIMARY_KEY	需要加双引号的关键字
SWITCH_TIME	主备切换的次数，默认为 3
SWITCH_INTERVAL	主备切换的时间间隔，默认为 200ms
TIME_ZONE	时区，默认为当前时区
RWSEPARATE	是否读写分离，默认为 FALSE
RWPERCENT	读写分离百分比，默认为 25
CONNPOOLING	是否使用连接缓存池，默认为 TRUE
STMTPOOLING	是否启用句柄重用，默认为 TRUE
POOLSIZE	句柄重用缓冲区的大小，默认为 100
ENCODING	客户端本地编码，支持 UTF-8、GB18030，默认为 GB18030

5.2.2 DmCommand 对象

表示要对 DM 数据库执行的一个 Transact-SQL 语句或存储过程。

公共属性

CommandText：获取或设置要对数据源执行的 Transact-SQL 语句或存储过程；

CommandTimeout：获取或设置在终止执行命令的尝试并生成错误之前的等待时间；

CommandType：获取或设置一个值，该值指示如何解释 CommandText 属性；

Connection：获取或设置 DmCommand 的此实例使用的 DmConnection；

Parameters：获取 DmParameterCollection；

Transaction：获取或设置将在其中执行 DmCommand 的 DmTransaction；

UpdatedRowSource：获取或设置命令结果在由 DbDataAdapter 的 Update 方法使用时如何应用于 DataRow。

公共方法

DmCommand()：构造函数，初始化 DmCommand 的新实例；

Cancel()：试图取消 DmCommand 的执行；

CreateParameter()：创建 DmParameter 对象的新实例；

ExecuteNonQuery()：对连接执行 Transact-SQL 语句并返回受影响的行数；

ExecuteReader()：将 CommandText 发送到 Connection 并生成一个 DmDataReader；

ExecuteReader(CommandBehavior)：以指定方式执行 CommandText；

ExecuteScalar()：执行查询，并返回查询所返回的结果集中第一行的第一列。忽略额外的列或行；

Prepare()：在 DM 的实例上创建命令的一个准备版本。

5.2.3 DmDataAdapter 对象

用于填充 DataSet 和更新 DM 数据库的一组数据命令和一个数据库连接。

公共属性

DeleteCommand：获取或设置一个 Transact-SQL 语句或存储过程，以从数据集删除记录；

InsertCommand：获取或设置一个 Transact-SQL 语句或存储过程，以在数据源中插入新记录；

SelectCommand：获取或设置一个 Transact-SQL 语句或存储过程，用于在数据源中选择记录；

UpdateCommand：获取或设置一个 Transact-SQL 语句或存储过程，用于更新数据源中的记录；

TableMappings：获取一个集合，它提供源表和 DataTable 之间的主映射。

公共方法

DmDataAdapter()：构造函数，初始化 DmDataAdapter 的新实例；

Fill()：在 DataSet 中添加或刷新行以匹配数据源中的行；

FillSchema()：将 DataTable 添加到 DataSet 中，并配置架构以匹配数据源中的架构；

Update()：为 DataSet 中每个已插入、已更新或已删除的行调用相应的 INSERT、UPDATE 或 DELETE 语句。

5.2.4 DmDataReader 对象

通过只向前方式从结果集中获取行数据。

公共属性

Depth：获取一个值，该值指示当前行的嵌套深度，目前 DM 返回常数 0；

FieldCount：获取当前行中的列数；

IsClosed：获取一个值，该值指示数据读取器是否已关闭；

RecordsAffected：获取执行 Transact-SQL 语句所更改、插入或删除的行数。

公共方法

Close()：关闭 DmDataReader 对象；

GetBoolean()：获取指定列的布尔值形式的值；

GetByte()：获取指定列的字节形式的值；

GetBytes()：从指定的列偏移量将字节流读入缓冲区，并将其作为从给定的缓冲区偏移量开始的数组；

GetChar()：获取指定列的单个字符串形式的值；

GetChars()：从指定的列偏移量将字符流作为数组从给定的缓冲区偏移量开始读入缓冲

区；

GetDataTypeName()：获取源数据类型的名称；

GetDateTime()：获取指定列的 DateTime 对象形式的值；

GetDecimal()：获取指定列的 Decimal 对象形式的值；

GetDouble()：获取指定列的双精度浮点数形式的值；

GetEnumerator()：返回可循环访问集合的枚举数；

GetFieldType()：获取是对象的数据类型的 Type；

GetFloat()：获取指定列的单精度浮点数形式的值；

GetInt16()：获取指定列的 16 位有符号整数形式的值；

GetInt32()：获取指定列的 32 位有符号整数形式的值；

GetInt64()：获取指定列的 64 位有符号整数形式的值；

GetKeyCols()：获取构成主关键字的列；

GetName()：获取指定列的名称；

GetOrdinal()：在给定列名称的情况下获取列序号；

GetSchemaTable()：返回一个 DataTable，它描述 DmDataReader 的列元数据；

GetString()：获取指定列的字符串形式的值；

GetUniqueCols()：获取有唯一性约束的列；

GetValue()：获取以本机格式表示的指定列的值；

GetValues()：获取当前行的集合中的所有属性列；

IsDBNull()：获取一个值，该值指示列中是否包含不存在的或缺少的值；

NextResult()：当读取批处理 Transact-SQL 语句的结果时，使数据读取器前进到下一个结果；

Read()：使 DmDataReader 前进到下一条记录。

5.2.5 DmParameter 对象

表示 DmCommand 的参数以及这些参数各自到 DataSet 中的列的映射。

公共属性

DbType：获取或设置参数的 DbType；

Direction：获取或设置一个值，该值指示参数是只可输入、只可输出、双向还是存储过程返回值参数；

ParameterName：获取或设置 DmParameter 的名称；

Precision：获取或设置用来表示 Value 属性的最大位数；

Scale：获取或设置 Value 解析为的小数位数；

Size：获取或设置列中数据的最大大小；

SourceColumn：获取或设置源列的名称，该源列映射到 DataSet 并用于加载或返回 Value；

SourceVerion：获取或设置在加载 Value 时使用的 DataRowVersion；

Value：获取或设置该参数的值。

公共方法

DmParameter()：构造函数，初始化 DmParameter 的新实例。

5.2.6 DmParameterCollection 对象

表示与 DmCommand 相关的参数集合以及这些参数各自到 DataSet 中的列的映射。

公共属性

Count：获取集合中 DmParameter 对象的数目。

公共方法

Add()：将 DmParamter 添加到 DmParameterCollection；
Clear()：从集合中移除所有项；
Contains()：获取一个值，该值指示集合中是否存在 DmParameter；
CopyTo()：将 DmParameter 对象从 DmParameterCollection 复制到指定的数组；
IndexOf()：获取 DmParameter 在集合中的位置；
Insert()：将 DmParameter 插入到集合中的指定索引位置；
Remove()：从集合中移除指定的 DmParameter；
RemoveAt()：从集合中移除指定的 DmParameter。

5.2.7 DmTransaction 对象

表示要在 DM 数据库中处理的 Transact-SQL 事务。

公共属性

Connection：获取与该事务关联的 DmConnection 对象；
IsolationLevel：指定该事务的 IsolationLevel。

公共方法

Commit()：提交数据库事务；
Dispose()：释放由 DmTransaction 占用的非托管资源，还可以释放托管资源；
Rollback()：回滚数据库事务；
Save()：在事务中创建保存点，并指定保存点名称。

5.2.8 DmCommandBuilder 对象

自动生成用于协调 DataSet 的更改与关联数据库的单表命令。继承自 DbCommandBuilder。

公共属性

ConflictOption：指定 ConflictOption 选项；

QuotePrefix：获取或设置指定其名称包含空格或保留标记等字符的数据库对象（例如，表或列）时使用的开始字符；

QuoteSuffix：获取或设置一个或多个结束字符，供指定其名称中包含空格或保留标记等字符的数据库对象（例如，表或列）时使用。

公共方法

使用继承自 DbCommandBuilder 的公共方法。

5.2.9 DmConnectionStringBuilder 对象

自动生成用于连接对象进行连接的字符串。 继承自 DbCommandBuilder。

公共属性

ConflictOption：指定 ConflictOption 选项；

QuotePrefix：获取或设置指定其名称包含空格或保留标记等字符的数据库对象（例如，表或列）时使用的开始字符；

QuoteSuffix：获取或设置一个或多个结束字符，供指定其名称中包含空格或保留标记等字符的数据库对象（例如，表或列）时使用。

公共方法

使用继承自 DbCommandBuilder 的公共方法。

5.2.10 DmClob 对象

用于访问服务器的字符类型的大字段对象。

公共属性

无。

公共方法

String GetString(int pos, int length)：从 pos 所指定的位置获取个数为 length 字符。pos 从 1 开始计数。如果 length 超过所取字符，则返回实际长度的字符；

int Length()：获取大字段的长度；

int SetString(long pos, String str, int offset, int len)：从 pos 所指定的位置，更新大字段内容为 str 的内容，offset 为设置 str 的偏移，len 为偏移后设置的字符串长度；

void Truncate(long len)：截断大字段为 len 长度；

String GetSubString(long pos, int length)：从 pos 所指定的位置，获取长度为 length 的字符串。

5.2.11 DmBlob 对象

用于访问服务器二进制类型大字段。

公共属性

无。

公共方法

int Length()：获取大字段数据长度；

int SetBytes(long pos, byte[] bytes, int offset, int len)：从 pos 所指定的位置，更新大字段内容为 bytes 的内容，offset 为设置 bytes 的偏移，len 为偏移后设置的字节长度；

byte[] GetBytes(long pos, int length)：从 pos 所指定的位置获取 length 个数的字节数据；

void truncate(long len)：将大字段截断为 len 长度；

Stream GetStream()：获取流对象，通过流对象进行数据读取。

5.3 注册.NET 驱动

有部分场景，使用 DmProvider 时需要注册.NET 驱动，例如通过 DbProviderFactories 类调用 DmProvider 创建连接，NHibernate 及 EFDmProvider 的使用，都需要注册.net 驱动。下面详细介绍下如何注册.NET 驱动。

步骤如下：

1、注册 DmProvider。

```
gacutil /if E:\dmdbms\drivers\dotNetProvider\DmProvider\DmProvider.dll
```

2、修改对应框架的配置文件 machine.config。

例如，配置文件 machine.config 目录位于 C:\Program Files\Microsoft Visual Studio 10.0\VC>notepad %WINDIR%\Microsoft.NET\Framework\v2.0.50727\config\machine.config。在配置文件 machine.config 中添加以下内容：

```
...
<DbProviderFactories>
        <add description="DM .Net Framework Data Provider" invariant="Dm" name="DM Data Provider" type="Dm.DmClientFactory, DmProvider, Version=1.1.0.0, Culture=neutral, PublicKeyToken=7a2d44aa446c6d01"/>
</DbProviderFactories>
...
```

例如，通过 DbProviderFactories 类调用 DmProvider 创建连接使用.NET 驱动的情况。

```
using System.Data.Common;
...
 public static void TestFunc()
      {
         DbProviderFactory  factory = DbProviderFactories.GetFactory("Dm");
         DbConnection sconn = factory.CreateConnection();
         sconn.ConnectionString = "Server=localhost; UserId=SYSDBA; PWD=SYSDBA";
         sconn.Open();
         DbCommand scmd = factory.CreateCommand();
         scmd.Connection = sconn;
         try
         {
            scmd.CommandText  = "drop table t1 cascade;";
            scmd.ExecuteNonQuery();
         }
         catch (Exception)
         {
         }
}
```

5.4Nhibernate Dm 方言包

DmDialect.dll 是基于 Dm 数据库开发的支持 Nhibernate 的方言包类库，放在 DM 安装目录..\drivers\dotNetProvider\DmDialect 中，方言包程序集名为：DmDialect, Version=1.0.0.0, Culture=neutral, PublicKeyToken=072d25982b139bf8。

Nhibernate Dm 方言包支持 NET Framework，NET Core, NET Standard 框架平台，与 Nhibernate 的版本是保持一致的。

DmDialect.dll 中包含的类有：NHibernate.Driver.DmDriver、NHibernate.Dialect.DmDialect 、Nhibernate.Dialect.Schema.DmDataBaseSchema、NHibernate.AdoNet. DmBatchingBatcher 和 NHibernate.AdoNet. DmBatchingBatcherFactory。

下面介绍一下如何使用方言包：

1.添加方言包依赖项。

Nhibernate Dm 方言包依赖项为：DmProvider.dll 和 Nhibernate.dll。其中 DmProvider.dll 请参考 5.3 注册.NET 驱动 。

2.应用程序添加依赖项之后，修改 Nhibernate 配置属性。

```
"dialect": NHibernate.Dialect.DmDialect, DmDialect, Version=1.0.0.0, Culture=neutral, PublicKeyToken=072d25982b139bf8
"connection.driver_class": NHibernate.Driver.DmDriver, DmDialect, Version=1.0.0.0, Culture=neutral, PublicKeyToken=072d25982b139bf8
"connection.connection_string": Server=localhost;UserId=SYSDBA;PWD=SYSDBA;encoding=utf-8;PORT=5236
```

5.5.NET Data Provider 基本示例

下面的示例使用 DM .NET Data Provider，实现 DM 数据库的连接与查询，代码使用 C# 编写：

首先在项目中引用 DmProvider.dll，DmProvider.dll 在达梦数据库安装目录下的 bin 文件夹下可以找到。

```
using System;
using System.Collections.Generic;
using System.Text;
using Dm;
namespace DMDemo
{
    class Demo
    {
```

```
        //返回结果
          static int ret = 1;
          static DmConnection cnn = new DmConnection();
          [STAThread]
          static int Main(string[] args)
          {
                try
                {
                      cnn.ConnectionString = "Server=localhost; User Id=SYSDBA; PWD=SYSDBA";
                      cnn.Open();
                   Demo demo = new Demo();
                   demo.TestFunc();
                      cnn.Close();
                }
                catch(Exception ex)
                {
                      Console.WriteLine(ex.Message);
                }
             Console.ReadLine();
                return ret;
          }
          public void TestFunc()
          {
                DmCommand command = new DmCommand();
             command.Connection = cnn;
                try
                {
                   string a, b, c;
                   command.CommandText = "SELECT NAME, AUTHOR, PUBLISHER FROM
PRODUCTION.PRODUCT;";
                      DmDataReader reader = command.ExecuteReader();
                      while(reader.Read())
                      {
                        a = reader.GetString(0);
                            b = reader.GetString(1);
                        c = reader.GetString(2);
                        Console.WriteLine("NAME：" + a);
                        Console.WriteLine("AUTHOR：" + b);
                        Console.WriteLine("PUBLISHER：" + c);
                        Console.WriteLine("--------------------");
                      }
                }
                catch(Exception ex)
                {
```

```
                    Console.WriteLine(ex.Message);
                    ret = 0;
                }
            }
        }
}
```

5.6 对象使用

5.6.1 连接

DM .NET Provider 使用 DMConnection 对象提供与 DM 数据库的连接。DM .NET Provider 支持类似于 OLE DB 连接字符串格式的连接字符串格式。

例如指定与本机的 DM 数据库建立连接，用户名和口令均为“SYSDBA”，则代码片断如下。

```
DmConnection conn = new DmConnection(“Server=localhost; User Id=SYSDBA; PWD=SYSDBA”);
conn.Open();
```

5.6.2 查询与结果集

当建立与数据库的连接后，可以使用 DmCommand 对象来执行命令并从数据库中返回结果。您可以使用 DmCommand 构造函数来创建命令，该构造函数采用在数据源、DmConnection 对象和 DmTransaction 对象中执行的 SQL 语句的可选参数。也可以使用 DmConnection 的 CreateCommand()方法来创建用于特定 DmConnection 对象的命令。您可以使用 DmCommandText 属性来查询和修改 DmCommand 对象的 SQL 语句。

DmCommand 对象公开了几个可用于执行所需操作的 Execute 方法。当以数据流的形式返回结果时，使用 ExecuteReader() 可返回 DataReader 对象。使用 ExecuteScalar() 可返回单个值。使用 ExecuteNonQuery() 可执行不返回行的命令。

可以使用 DmDataReader 从数据库中检索只读、只进的数据流。查询结果在查询执行时返回，并存储在客户端的缓冲区中，直到您使用 DmDataReader 的 Read()方法对它们发出请求。

当创建 DmCommand 对象的实例后，可调用 DmCommand.ExecuteReader()从数据源中检索行，从而创建一个 DmDataReader，如以下程序片断所示：

```
DmCommand command = conn.CreateCommand("SELECT NAME, AUTHOR, PUBLISHER FROM PRODUCTION.PRODUCT;");
DmDataReader myReader = command.ExecuteReader();
```

5.6.3 插入、更新、删除

通过 DmCommand 对象的 ExecuteNonQuery 方法可以执行 INSERT 语句来插入数据，C#示例代码如下：

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
using Dm;
namespace DMDemo
{
    class InsertDemo
    {
        //返回结果
        static int ret = 1;
        static DmConnection cnn = new DmConnection();
        [STAThread]
        static int Main(string[] args)
        {
            try
            {
                cnn.ConnectionString = "Server=localhost; User Id=SYSDBA; PWD=SYSDBA";
                cnn.Open();
                InsertDemo demo = new InsertDemo();
                demo.TestFunc();
                cnn.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
            return ret;
        }
        public void TestFunc()
        {
            DmCommand command = new DmCommand();
            command.Connection = cnn;
            try
            {
                command.CommandText = "INSERT INTO PRODUCTION.PRODUCT(NAME, AUTHOR,
PUBLISHER, " +
                    "PUBLISHTIME, PRODUCT_SUBCATEGORYID, PRODUCTNO,
SATETYSTOCKLEVEL, ORIGINALPRICE, " +
                    "NOWPRICE, DISCOUNT, DESCRIPTION, TYPE, PAPERTOTAL, WORDTOTAL,
SELLSTARTTIME, " +
                "SELLENDTIME) VALUES ('三国演义', '罗贯中', '中华书局', '2005-04-01', 4, '9787101046121', "
+
            "10, 19.0000, 15.2000, 8.0, '《三国演义》是中国第一部长篇章回体小说，中国小说" +
            "由短篇发展至长篇的原因与说书有关。宋代讲故事的风气盛行，说书成为一种职业，说" +
```

```
            "书人喜欢拿古代人物的故事作为题材来敷演，而陈寿《三国志》里面的人物众多，事件" +
            "纷繁，正是撰写故事的最好素材。三国故事某些零星片段原来在民间也已流传，加上说" +
            "书人长期取材，内容越来越丰富，人物形象越来越饱满，最后由许多独立的故事逐渐组" +
            "合而成长篇巨著。这些各自孤立的故事在社会上经过漫长时间口耳相传，最后得以加工" +
            "、集合成书，成为中国第一部长篇章回体小说，这是一种了不起的集体创造，与由单一" +
              "作者撰写完成的小说在形态上有所不同。', '16', 943, 93000, '2006-03-20', '1900-01-01')";
                    command.ExecuteNonQuery();
                    string a, b, c;
                    command.CommandText = "SELECT NAME, AUTHOR, PUBLISHER FROM
PRODUCTION.PRODUCT;";
                    DmDataReader reader = command.ExecuteReader();
                    while (reader.Read())
                    {
                        a = reader.GetString(0);
                        b = reader.GetString(1);
                        c = reader.GetString(2);
                        Console.WriteLine("NAME：" + a);
                        Console.WriteLine("AUTHOR：" + b);
                        Console.WriteLine("PUBLISHER：" + c);
                        Console.WriteLine("--------------------");
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                    ret = 0;
                }
            }
        }
}
```

通过 DmCommand 对象的 ExecuteNonQuery 方法可以执行 UPDATE 语句来更新数据，C#示例代码如下：

```
command.CommandText = "UPDATE PRODUCTION.PRODUCT SET "
    + " NAME = '三国演义（上）' WHERE PRODUCTID = 11";
command.ExecuteNonQuery();
```

通过 DmCommand 对象的 ExecuteNonQuery 方法可以执行 DELETE 语句来删除数据，C#示例代码如下：

```
command.CommandText = "DELETE FROM PRODUCTION.PRODUCT WHERE PRODUCTID = 11";
command.ExecuteNonQuery();
```

5.6.4 大对象

下面的示例将展示读取一张图片到并保存到数据库中。图片以二进制数据存储在 PRODUCT 表中的 PHOTO 字段中。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using Dm;
namespace DMDemo
{
    class BinaryDemo
    {
        //返回结果
        static int ret = 1;
        static DmConnection cnn = new DmConnection();
        [STAThread]
        static int Main(string[] args)
        {
            try
            {
                cnn.ConnectionString = "Server=localhost; User Id=SYSDBA; PWD=SYSDBA";
                cnn.Open();
                BinaryDemo demo = new BinaryDemo();
                demo.TestFunc();
                cnn.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
            return ret;
        }
        public void TestFunc()
        {
            DmCommand command = new DmCommand();
            command.Connection = cnn;
            try
            {
                FileInfo fi = new FileInfo(@"F:\dotnet\dameng\DM 数据库例子\三国演义.jpg");
                FileStream fs = fi.OpenRead();
                int nBytes = (int)fs.Length;
                byte[] dataArray = new byte[nBytes];
                fs.Read(dataArray, 0, nBytes);
                fs.Close();
                command.CommandText = "UPDATE  PRODUCTION.PRODUCT SET PHOTO = :PHOTO
WHERE PRODUCTID = 11";
```

```
                DmParameter parm1 = command.Parameters.Add(":PHOTO", DmDbType.Binary);
                parm1.Value = dataArray;
                command.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                ret = 0;
            }
        }
    }
}
```

5.6.5 自增列

达梦数据库.NET Data Provider 接口支持自增列，如果数据库中的列设置为自增，那么在往数据库中插入记录的时候，不需要给该字段赋值。例如 PRODUCT 表中的 PRODUCTID 字段就设置为自增了，在插入数据时没有给 PRODUCTID 字段赋值，在数据插入达梦数据库时，由数据库根据 PRODUCTID 自增列的设置自动进行赋值。

5.6.6 存储过程与函数

达梦数据库.NET Data Provider 可以使用 DmCommand 对象来执行存储过程与函数。创建一个存储过程“UPDATEPRODUCT”用来更新表 PRODUCT 中的 NAME 字段，存储过程代码如下：

```
CREATE OR REPLACE PROCEDURE "PRODUCTION"."UPDATEPRODUCT"
(
  V_ID INT,
  V_NAME VARCHAR(50)
)
AS
BEGIN
  UPDATE PRODUCTION.PRODUCT SET NAME = V_NAME WHERE PRODUCTID = V_ID;
END;
```

达梦数据库.NET Data Provider 调用存储过程的示例代码如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using Dm;
namespace DMDemo
{
    class ProcDemo
    {
```

```
        //返回结果
        static int ret = 1;
        static DmConnection cnn = new DmConnection();
        [STAThread]
        static int Main(string[] args)
        {
            try
            {
                cnn.ConnectionString = "Server=localhost; User Id=SYSDBA; PWD=SYSDBA";
                cnn.Open();
                ProcDemo demo = new ProcDemo();
                demo.TestFunc();
                cnn.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
            return ret;
        }
        public void TestFunc()
        {
            DmCommand command = new DmCommand();
            command.Connection = cnn;
            try
            {
                command.CommandText = "PRODUCTION.UPDATEPRODUCT";
                command.CommandType = System.Data.CommandType.StoredProcedure;
                DmParameter parm1 = command.Parameters.Add(":V_ID", DmDbType.Int32);
                parm1.Value = 1;
                parm1.Direction = System.Data.ParameterDirection.Input;
                DmParameter parm2 = command.Parameters.Add(":V_NAME", DmDbType.VarChar);
                parm2.Value = "红楼梦（下）";
                parm2.Direction = System.Data.ParameterDirection.Input;
                command.ExecuteNonQuery();

                string a, b, c;
                command.Parameters.Clear();
                command.CommandText = "SELECT NAME, AUTHOR, PUBLISHER FROM
PRODUCTION.PRODUCT;";
                DmDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
```

```
                    a = reader.GetString(0);
                    b = reader.GetString(1);
                    c = reader.GetString(2);
                    Console.WriteLine("NAME：" + a);
                    Console.WriteLine("AUTHOR：" + b);
                    Console.WriteLine("PUBLISHER：" + c);
                    Console.WriteLine("--------------------");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                ret = 0;
            }
        }
    }
}
```

以上代码调用存储过程“UPDATEPRODUCT” 将 PRODUCT 表中的 PRODUCTID 字段内容为 1 的记录的 NAME 字段内容由"红楼梦"更新为"红楼梦（上）"。

第 6 章 DM PHP 编程指南

6.1 DM PHP 介绍

本章主要介绍 DM PHP 扩展的基本概念以及使用方法，以便于用户更好地使用 DM PHP 扩展库编写 PHP 应用程序。

在使用 PHP 语言的 Web 应用中，为了可以和 DM 数据库的服务器端进行通信，并且获得更快的速度以及对系统更强的控制，我们可以通过一个用 C 语言函数实现的瘦中间层来实现这个目标。该瘦中间层就是 PHP 扩展，它实现了一组 C 语言 API，成为 PHP 语言级别的函数调用。

DM PHP 是在 PHP 开放源码的基础上开发的一个动态扩展库，接口的实现参考了 MySQL 的 PHP 扩展，无论在功能、参数以及调用过程都和后者十分类似，命名统一采用 dm 开头的小写英文字母方式，各个单词之间以下划线分割。PHP 应用程序可通过 DM PHP 扩展接口库访问 DM 数据库服务器。

下面列出 DM 提供的 PHP 扩展函数，并简要说明每个函数的功能。

扩展库接口函数	作用描述
dm_connect	打开一个到 DM 服务器的连接
dm_pconnect	打开一个到 DM 服务器的持久连接
dm_close	关闭 DM 连接
dm_set_connect	设置连接
dm_error	返回上一个 DM 操作产生的文本错误信息
dm_errno	返回上一个 DM 操作中的错误信息的数字编码
dm_query	发送一条 DM 查询
dm_unbuffered_query	向 DM 发送一条 SQL 查询，并不获取和缓存结果的行
dm_more_query_no_result	获取下一个 ResultSet 结果集，如果无结果集，返回 False
dm_db_query	发送一条 DM 查询
dm_affected_rows	取得前一次 DM 操作所影响的记录行数
dm_escape_string	转义一个字符串用于 dm_query
dm_fetch_array	从结果集中取得一行作为关联数组，或数字数组，或二者兼有
dm_fetch_assoc	从结果集中取得一行作为关联数组
dm_fetch_field	从结果集中取得列信息并作为对象返回
dm_num_fields	取得结果集中字段的数目
dm_fetch_length:	取得结果集中每个输出的长度
dm_fetch_object	从结果集中取得一行作为对象
dm_fetch_row	从结果集中取得一行作为枚举数组
dm_field_flags	从结果中取得和指定字段关联的标志
dm_field_len	返回指定字段的长度
dm_field_name	取得结果中指定字段的字段名
dm_field_seek	将结果集中的指针设定为制定的字段偏移量

dm_field_table	取得指定字段所在的表名
dm_field_type	取得结果集中指定字段的类型
dm_free_result	释放结果内存
dm_get_server_info	取得 DM 服务器信息
dm_list_fields	列出 DM 结果中的字段
dm_list_tables	列出 DM 数据库中的表
dm_num_rows	取得结果集中行的数目
dm_ping	Ping 一个服务器连接，如果没有连接则重新连接
dm_result	取得结果数据
dm_insert_id	取得上一步 INSERT 操作产生的 ID
dm_insert_id_ex	取得上一步 INSERT 操作产生的 ID
dm_tablename	取得表名
dm_data_seek	移动内部结果的指针
dm_set_object_name_case	设置比较时的大小写方式

6.2 基本示例

利用 DM PHP 驱动程序进行编程的一般步骤为：

1. 利用 dm_connect（）建立同数据库的连接。
2. DM PHP 数据操作。数据操作主要分为两个方面，一个是更新操作，例如更新数据库、删除一行、创建一个新表等；另一个就是查询操作。dm_query()执行完查询之后，会得到一个记录集。可以操作该对象来获得指定列的信息、读取指定行的某一列的值。
3. 释放资源。在操作完成之后，用户需要释放系统资源，主要是关闭结果集、关闭语句对象，释放连接。

下面用具体的编程实例来展示利用 DM PHP 对示例数据库 BOOKSHOP，以产品信息表 product 的增加、删除、修改、查询为例子进行数据库操作的基本步骤。

1. 增加记录

```
<?php
    /* 连接选择数据库 */
    $link = dm_connect("localhost", "SYSDBA", "SYSDBA")
        or die("Could not connect : " . dm_error());
    print "Connected successfully";
    /* 执行 SQL 查询 */
    $query = " INSERT INTO production.product(name,author,publisher,publishtime,
product_subcategoryid,productno,satetystocklevel,originalprice,nowprice,
discount,description,photo,sellstarttime)
VALUES('三国演义','罗贯中','中华书局','2005-04-01','4','9787101046121','10',
'19.0000','15.2000','8.0','《三国演义》是中国第一部长篇章回体小说!',null,'2006-03-20')";
    $result = dm_query($query) or die("Query failed : " . dm_error());
    /* 释放资源 */
    dm_free_result($result);
    /* 断开连接 */
```

```
    dm_close($link);
?>
```

2. 修改数据

```
<?php
    /* 连接选择数据库 */
    $link = dm_connect("localhost", "SYSDBA", "SYSDBA")
        or die("Could not connect : " . dm_error());
    print "Connected successfully";
    /* 执行 SQL 查询 */
    $query = " UPDATE production.product SET TYPE =1000 WHERE productid = 10";
    $result = dm_query($query) or die("Query failed : " . dm_error());
    /* 释放资源 */
    dm_free_result($result);
    /* 断开连接 */
    dm_close($link);
?>
```

3. 删除记录

```
<?php
    /* 连接选择数据库 */
    $link = dm_connect("localhost", "SYSDBA", "SYSDBA")
        or die("Could not connect : " . dm_error());
    print "Connected successfully";
    /* 执行 SQL 查询 */
    $query = " DELETE FROM production.product WHERE productid = 10";
    $result = dm_query($query) or die("Query failed : " . dm_error());
    /* 释放资源 */
    dm_free_result($result);
    /* 断开连接 */
    dm_close($link);
?>
```

4. 查询记录

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<style type="text/css">
<!--
body,td,th {
    font-size: 12px;
}
-->
</style></head>
```

```
<body><?php
    /* 连接选择数据库 */
    $link = dm_connect("localhost", "SYSDBA", "SYSDBA")
        or die("Could not connect : " . dm_error());
    print "Connected successfully";
    /* 执行 SQL 查询 */
    $query = " select * from production.product";
    $result = dm_query($query) or die("Query failed : " . dm_error());
    /* 在 HTML 中打印结果 */
    print "<table border=\"1\" cellspacing=\"1\"  cellpadding=\"1\">\n";
    while ($line = dm_fetch_array($result, DM_ASSOC)) {
        print "\t<tr>\n";
        foreach ($line as $col_value) {
            print "\t\t<td>$col_value</td>\n";
        }
        print "\t</tr>\n";
    }
    print "</table>\n";
    /* 断开连接 */
    dm_close($link);
?>
 </body>
</html>
```

5. 存储过程

运行程序，启动达梦数据库服务器并创建用于修改产品信息的存储过程。存储过程代码如下：

```
create or replace procedure "PRODUCTION"."updateProduct"
(
   v_id int,
   v_name varchar(50)
)
as
begin
   UPDATE production.product SET name = v_name WHERE productid = v_id;
end;
```

在 PHP 里得到返回的结果集。

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
 <style type="text/css">
<!--
body,td,th {
    font-size: 12px;
}
-->
```

```
</style></head>
<body><?php
    /* 连接选择数据库 */
    $link = dm_connect("localhost", "SYSDBA", "SYSDBA")
        or die("Could not connect : " . dm_error());
    print "Connected successfully";
    dm_select_db("system") or die("Could not select database");
    /* 执行 SQL 查询 */
    $query = " CALL production.updateProduct(1,'红楼梦')";
    $result = dm_query($query) or die("Query failed : " . dm_error());
     /* 释放资源 */
    dm_free_result($result);
    /* 断开连接 */
    dm_close($link);
?>
</body>
</html>
```

6.3 DM PHP 模块加载

目前 DM 支持的 PHP 版本为 PHP5.2~5.5，用户可根据自己安装的 PHP 版本选择对应的接口库。下面的示例均以 PHP5.3 版本为例，如果版本不同，将 53 改为对应值即可。

1. linux 系统下加载

步骤：

1) 下载并安装 apache

从网络中下载 apache-2.0.48.tar.gz

存至/home/tmp

```
cd /home/tmp
tar -xvzf apache-2.0.48.tar.gz  /usr/local
cd /usr/local/apache-2.0.48
./configure --prefix=/usr/local/apache  --enable-module=so
make
make install
cd /usr/local/apache/conf
vi httpd.conf
ServerName localhost
port 80
DirectoryIndex default.php default.phtml default.php3 default.html default.htm
AddType application/x-httpd-php .php .phtml .php3 .inc
AddType application/x-httpd-php-source .phps
```

2) 下载并安装 php

从网络中下载最新 php-5.2.6.tar.gz

存至/home/tmp

```
cd /home/tmp
tar -xvzf php-x.x.x.tar.gz    /usr/local
cd /usr/local/php-x.x.x
./buildconf --force
./configure --with-apache2=/usr/local/apache/bin/apxs
make
make install
cp php.ini-dist /usr/local/lib/php.ini
```

3) 安装 DM DBMS

假定安装到/usr/local/DMDBMS 目录

4) 安装 DM PHP

```
cp dm_php    /home
cd /home/dm_php
; 执行
/usr/local/bin/phpize
./configure
make
cd /usr/local/lib
vi php.ini
; 修改下面配置，假设“/home/dm_php/modules”为 libphp53ts_dm.so 所在路径
; libphp53ts_dm.so 中的 ts 表示线程安全版本，不带 ts 表示非线程安全版本
register_globals=on
extension_dir=/home/dm_php/modules
; 添加下面语句
extensions=libphp53ts_dm.so
; 添加 php.ini 中有关连接的配置
[dm]
; 是否允许持久性连接
dm.allow_persistent = 1
; 允许建立持久性连接的最大数.  -1 为没有限制.
dm.max_persistent = -1
; 允许建立连接的最大数(包括持久性连接).  -1 为没有限制.
dm.max_links  = -1
; 默认的主库地址
dm.default_host = localhost
; 默认的连接用户名
dm.default_user = SYSDBA
; 默认的连接口令.
dm.default_pw = SYSDBA
; 连接超时，这个参数未实际的用到，等待服务器支持
dm.connect_timeout = 10
; 对于各种变长数据类型，每列最大读取的字节数。如果它设置为 0 或是小于 0,那么，读取变长字段时，
将显示 NULL 值
dm.defaultlrl = 4096
```

```
; 是否读取二进制类型数据，如果它设置为 0，那么二进制将被 NULL 值代替
dm.defaultbinmode = 1
; 是否允许检察持久性连接的有效性，如果设置为 ON，那么当重用一个持久性连接时，会检察该连接是否还有效
dm.check_persistent = ON
```

2. Windows 系统下加载模块

步骤：

1) 下载 apache 的 windows 版本并安装，同时修改 httpd.conf；
2) 下载 php 并安装；
3) 安装 DM DBMS，拷贝 bin 目录下 php53_dm.dll 到 php 目录下的 extensions 目录中，修改 php.ini，添加 extension=php53_dm.dll，添加 php.ini 中有关连接的配置；
4) 重启 apache 服务器，在浏览器中输入 http:\\localhost\php_info.php 查看是否有 dm 模块项，如有说明加载 DM PHP 成功。

6.4 编程接口

1. dm_affected_rows

描述

取得前一次数据库操作 INSERT、UPDATE 或 DELETE 查询所影响的记录行数。如果连接句柄没有指定，则默认使用最近一次由 dm_connect()函数打开的连接句柄。

格式

```
int dm_affected_rows ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接句柄

返回值

如果最近一次操作是没有任何条件（WHERE）的 DELETE 查询，在表中所有的记录都会被删除，但该函数返回值为 0。

2. dm_connect

描述

建立一个到 DM 服务器的连接。

格式

```
resource dm_connect ( [string server [, string username [, string password]]])
```

参数

参数	描述
server	[IN]服务器名称
username	[IN]用户名称
password	[IN]用户密码

返回值

如果成功则返回一个连接标识，失败则返回 FALSE。

3. dm_pconnect

描述

分配用户描述符。

格式

```
resource dm_pconnect ( [string server [, string username [, string password]]]);
```

参数

参数	描述
server	[IN]服务器名称
username	[IN]用户名称
password	[IN]用户密码

返回值

如果成功则返回一个正的持久连接标识符，出错则返回 FALSE。

4. dm_close

描述

关闭指定的连接标识所关联的 DM 服务器的连接。如果没有指定 link_identifier，则关闭上一个打开的连接

格式

```
bool dm_close ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

如果成功则返回 TRUE，失败则返回 FALSE。

5. dm_error

描述

返回上一个 DM 操作产生的文本错误信息。

格式

```
string dm_error ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

返回上一个 DM 函数的错误文本，如果没有出错则返回 ''（空字符串）。如果没有指定连接资源号，则使用上一个成功打开的连接从 DM 服务器提取错误信息。

6. dm_errno

描述

返回上一个 DM 操作中的错误信息的数字编码。

格式

```
int dm_errno ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

返回上一个 DM 函数的错误号码，如果没有出错则返回 0（零）。

7. dm_query

描述

发送一条 DM 查询。

格式

```
resource dm_query ( string query [, resource link_identifier])
```

参数

参数	描述
query	[IN]查询字符串
link_identifier	[IN]连接标识符

返回值

仅对 SELECT、EXPLAIN 语句返回一个资源标识符，如果查询执行不正确则返回 FALSE。对于其它类型的 SQL 语句，dm_query() 在执行成功时返回 TRUE，出错时返回 FALSE。非 FALSE 的返回值意味着查询是合法的并能够被服务器执行。这并不说明任何有关影响到的或返回的行数。

8. dm_unbuffered_query

描述

向 DM 发送一条 SQL 查询，并不获取和缓存结果的行。

格式

```
resource dm_unbuffered_query ( string query [, resource link_identifier])
```

参数

参数	描述
query	[IN]查询字符串
link_identifier	[IN]连接标识符

返回值

向 DM 发送一条 SQL 查询 *query*，但不像 dm_query()那样自动获取并缓存结果集。一方面，这在处理很大的结果集时会节省可观的内存；另一方面，可以在获取第一行后立即对结果集进行操作，而不用等到整个 SQL 语句都执行完毕。当使用多个数据库连接时，必须指定可选参数 link_identifier。

9. dm_db_query

描述

发送一条 DM 查询。

格式

```
resource dm_db_query ( string database, string query [, resource link_identifier])
```

参数

参数	描述
database	[IN]指定的数据库名称
query	[IN]查询字符串
link_identifier	[IN]连接标识符

返回值

根据查询结果返回一个正的 DM 结果资源号，出错时返回 FALSE。

10. dm_escape_string

描述

转义一个字符串用于 dm_query。

格式

```
string dm_escape_string ( string unescaped_string)
```

参数

参数	描述
unescaped_string	[IN]被转义字符串

返回值

返回转义以后的字符串。

11. dm_fetch_array

描述

从结果集中取得一行作为关联数组，或数字数组，或二者兼有。

格式

```
array dm_fetch_array ( resource result [, int result_type])
```

参数

参数	描述
result	[IN]结果集资源
result_type	[IN]是一个常量，可以接受以下值：DM_ASSOC,DM_NUM 和 DM_BOTH，如果用了 DM_BOTH，将得到一个同时包含关联和数字索引的数组。用 DM_ASSOC 只得到关联索引, DM_NUM 只得到数字索引

返回值

返回根据从结果集取得的行生成的数组，如果没有更多行则返回 FALSE。

12. dm_fetch_assoc

描述

从结果集中取得一行作为关联数组。

格式

```
array dm_fetch_assoc ( resource result)
```

参数

参数	描述
result	[IN]连接句柄

返回值

返回根据从结果集取得的行生成的关联数组，如果没有更多行则返回 FALSE。

13. dm_fetch_field

描述

从结果集中取得列信息并作为对象返回。

格式

```
object dm_fetch_field ( resource result [, int field_offset])
```

参数

参数	描述
result	[IN]结果集资源
field_offset	[IN]字段偏移

返回值

返回一个包含字段信息的对象。

14. dm_num_fields

描述

取得结果集中字段的数目。

格式

```
int dm_num_fields ( resource result)
```

参数

参数	描述
result	[IN]结果集资源

返回值

返回结果集中字段的数目。

15. dm_fetch_lengths

描述

取得结果集中每个输出的长度。

格式

```
array dm_fetch_lengths ( resource result)
```

参数

参数	描述
result	[IN]结果集资源

返回值

以数组返回上一次用 dm_fetch_row 取得的行中每个字段的长度，如果出错返回 FALSE。

16. dm_fetch_object

描述

从结果集中取得一行作为对象。

格式

```
object dm_fetch_object ( resource result)
```

参数

参数	描述
result	[IN]结果集资源

返回值

返回根据所取得的行生成的对象，如果没有更多行则返回 FALSE。

17. dm_fetch_row

描述

从结果集中取得一行作为枚举数组。

格式

```
array dm_fetch_row ( resource result)
```

参数

参数	描述
result	[IN]结果集资源

返回值

返回根据所取得的行生成的数组，如果没有更多行则返回 FALSE。

18. dm_field_flags

描述

从结果中取得和指定字段关联的标志。

格式

string dm_field_flags (resource result, int field_offset)

参数

参数	描述
result	[IN]结果集资源
field_offset	[IN]字段偏移

返回值

返回指定字段的字段标志。每个标志都用一个单词表示，之间用一个空格分开。

19. dm_field_len

描述

返回指定字段的长度。

格式

int dm_field_len (resource result, int field_offset)

参数

参数	描述
result	[IN]结果集资源

返回值

返回指定字段的长度。

20. dm_field_name

描述

取得结果中指定字段的字段名。

格式

string dm_field_name (resource result, int field_index)

参数

参数	描述
result	[IN]结果集资源
field_index	[IN]字段索引号

返回值

返回指定字段索引的字段名。result 必须是一个合法的结果标识符，field_index 是该字段的数字偏移量。

21. dm_field_seek

描述

将结果集中的指针设定为制定的字段偏移量。

格式

int dm_field_seek (resource result, int field_offset)

参数

参数	描述
result	[IN]结果集资源

field_offset	[IN]字段偏移

返回值

返回指定字段的字段偏移量。

22. dm_field_table

描述

取得指定字段所在的表名。

格式

```
string dm_field_table ( resource result, int field_offset)
```

参数

参数	描述
field_offset	[IN]字段偏移
result	[IN]结果集资源

返回值

返回指定字段所在的表名。

23. dm_field_type

描述

取得结果集中指定字段的类型。

格式

```
string dm_field_type ( resource result, int field_offset)
```

参数

参数	描述
field_offset	[IN]字段偏移
result	[IN]结果集资源

返回值

返回字段类型有“int”，“real”，“char”，“blob”等。

24. dm_free_result

描述

释放结果内存。

格式

```
bool dm_free_result ( resource result)
```

参数

参数	描述
result	[IN]结果集资源

返回值

如果成功则返回 TRUE，失败则返回 FALSE。

25. dm_get_server_info

描述

取得 DM 服务器信息。

格式

```
string dm_get_server_info ( [resource link_identifier])
```

参数

参数	描述

link_identifier	[IN]连接标识符

返回值

返回 link_identifier 所使用的服务器版本。如果省略 link_identifier，则使用上一个打开的连接。

26. dm_list_fields

描述

列出 DM 结果中的字段。

格式

```
resource dm_list_fields ( string database_name, string table_name [, resource link_identifier])
```

参数

参数	描述
database_name	[IN]数据库名
link_identifier	[IN]连接标识符
table_name	[IN]表名

返回值

返回一个结果指针。

27. dm_list_tables

描述

列出 DM 数据库中的表。

格式

```
resource dm_list_tables ( string database [, resource link_identifier])
```

参数

参数	描述
database_name	[IN]数据库名
link_identifier	[IN]连接标识符

返回值

返回一个结果指针。

28. dm_num_rows

描述

取得结果集中行的数目。

格式

```
int dm_num_rows ( resource result)
```

参数

参数	描述
result	[IN]结果集

返回值

返回结果集中行的数目。此命令仅对 SELECT 语句有效。

29. dm_ping

描述

Ping 一个服务器连接，如果没有连接则重新连接。

格式

```
bool dm_ping ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

如果到服务器的连接可用则 dm_ping()返回 TRUE，否则返回 FALSE。

30. dm_result

描述

取得结果数据。

格式

```
mixed dm_result ( resource result, int row [, mixed field])
```

参数

参数	描述
result	[IN]结果集
row	[IN]指定行序号
field	[IN]指定列索引(从 1 开始)或列名称

返回值

返回 DM 结果集中一个单元的内容。

31. dm_insert_id

描述

取得上一步 INSERT 操作产生的 ID。

格式

```
int dm_insert_id ( [resource link_identifier])
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

返回给定的 link_identifier 中上一步 INSERT 查询中产生的 AUTO_INCREMENT 的 ID 号。如果没有指定 link_identifier，则使用上一个打开的连接。如果上一查询没有产生 AUTO_INCREMENT 的值，则 dm_insert_id()返回 0。

32. dm_tablename

描述

取得表名。

格式

```
string dm_tablename ( resource result, int i)
```

参数

参数	描述
result	[IN]表结果集
i	[OUT]表索引序号

返回值

接受 dm_list_tables()返回的结果指针以及一个整数索引作为参数并返回表名。

33. dm_tablename

描述

取得表名。

格式

```
string dm_tablename ( resource result, int i)
```

参数

参数	描述
result	[IN]表结果集
i	[OUT]表索引序号

返回值

接受 dm_list_tables()返回的结果指针以及一个整数索引作为参数并返回表名。

34. dm_data_seek

描述

移动内部结果的指针。

格式

```
bool dm_data_seek ( resource result_identifier, int row_number)
```

参数

参数	描述
result_identifier	[IN]结果集
row_number	[OUT]行索引序号

返回值

指定的结果标识所关联的 DM 结果内部的行指针移动到指定的行号。

35. dm_set_object_name_case

描述

设置使用列名称时的大小写方式。

格式

```
bool dm_set_object_name_case(resource con, int Case_sensitive)
```

参数

参数	描述
resource con	PHP 传入的要连接的连接资源，可选参数
Case_sensitive	大小写方式

返回值

设置成功，返回 TRUE；否则返回 FALSE。

第 7 章 DM DCI 编程指南

7.1 DM DCI 介绍

本章将较全面而系统地介绍 DM DCI（DM Call Interface）的基本概念以及 DM DCI 的使用方法，以便用户更好地使用 DM DCI 编写应用程序。

OCI（Oracle Call Interface）是 ORACLE 公司开发的一个应用程序开发工具，是一个通过访问 Oracle 数据库的服务器，控制各类 SQL 语句的执行，进而创建应用程序的的应用程序接口（API）。它支持 SQL 所有的数据定义、数据操作、查询、事务管理等操作，支持 C 和 C++的数据类型、调用、语法和语义。它提供了一组可对 Oracle 数据库进行存取的接口子例程(函数)。

DM DCI 是参照 OCI 的接口标准，结合自身的特点，为开发人员提供向 oracle 兼容功能的一款接口产品。以下部分将介绍 dci 的实现内容，以及针对 oralce 兼容的实现和限制。

7.2 数据类型

DCI 支持的数据类型表 7.1 所示：

表 7.1 DCI 支持的数据类型

数据类型	类型说明
SQLT_CHR	变长字符串类型
SQLT_INT	整数类型，当指定长度为 1 时，对应 C 的 BYTE 类型；长度为 2 时，对应 C 的 short 类型；长度为 4 时，对应 C 的 int 类型；长度为 8 时，对应 C 的 int64 类型。
SQLT_FLT	浮点数据类型，当指定的长度为 4 时，对应 C 数据类型 float，当指定长度为 8 时，对应 C 数据类型 double
SQLT_STR	以空结尾的变长字符串类型
SQLT_LNG	长数据类型
SQLT_VCS	变长字符类型
SQLT_DAT	7 字节存贮的时间格式， 取值范围是公元前 4712 年 1 月 1 日至公元 9999 年 12 月 31 日
SQLT_ODT	标准的时间数据类型，可以使用 OCIDate 结构体来格式化获取的日期
SQLT_VBI	变长二进制数据类型
SQLT_BIN	二进制数据类型
SQLT_LBI	长二进制数据类型
SQLT_UIN	无符号整数类型
SQLT_LVC	长变长字符类型
SQLT_LVB	长变长二进制数据类型
SQLT_AFC	固定长度的字符串类型
SQLT_AVC	变长字符类型
SQLT_RSET	结果集游标类型，该参数用在存贮过程的游标参数， 可以返回结果集中产生的结果集。

SQLT_TIME	OCITime 结构体可以用此类型绑定。
SQLT_TIMESTAMP	OCIDateTime 结构体可以用此类型绑定。
SQLT_CLOB	CLOB 类型
SQLT_BLOB	BLOB 类型

7.3 参考函数

7.3.1 关系型接口函数

7.3.1.1 Connect、 Authorize 和 Initialize 方法

1. OCIEnvCreate

描述

创建并初始化环境句柄

格式

```
sword OCIEnvCreate    ( OCIEnv          **envhpp,
                        ub4             mode,
                        CONST dvoid     *ctxp,
                        CONST dvoid     *(*malocfp)
                                        (dvoid *ctxp,
                                         size_t size),
                        CONST dvoid     *(*ralocfp)
                                        (dvoid *ctxp,
                                         dvoid *memptr,
                                         size_t newsize),
                        CONST void      (*mfreefp)
                                        (dvoid *ctxp,
                                         dvoid *memptr))
                        size_t          xtramemsz,
                        dvoid           **usrmempp );
```

参数

参数	描述
envhpp（OUT）	用于输出环境句柄
mode（IN）	初始化的模式，目前只支持 OCI_DEFAULT – 默认模式
ctxp(IN)	保留兼容，忽略该参数
malocfp(IN)	保留兼容，忽略该参数
ralocfp(IN)	保留兼容，忽略该参数
mfreefp(IN)	保留兼容，忽略该参数
xtramemsz(IN)	额外附加分配的内存空间
usrmempp(OUT)	返回额外分配的附加内存空间的地址

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR，使用无效句柄时返回 OCI_INVALID_HANDLE

2. OCIEnvInit

描述

分配并初始化环境句柄。

格式

```
sword OCIEnvInit ( OCIEnv      **envhpp,
                   ub4         mode,
                   size_t      xtramemsz,
                   dvoid       **usrmempp );
```

参数

参数	描述
envp(OUT)	环境句柄
mode(IN)	初始化的模式，目前只支持 OCI_DEFAULT – 默认模式
xtramem_sz(IN)	额外附加分配的内存空间
usrmempp(IN)	返回额外分配的附加内存空间的地址

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

3. OCIInitialize

描述

初始化 OCI 应用环境，OCI 会在这个函数中初始化内部的全局变量和加载一些配置信息。

格式

```
sword OCIInitialize ( ub4              mode,
                      CONST dvoid      *ctxp,
                      CONST dvoid      *(*malocfp)
                                       (/* dvoid *ctxp,
                                           size_t size _*/),
                      CONST dvoid      *(*ralocfp)
                                       (/*_ dvoid *ctxp,
                                         dvoid *memptr,
                                         size_t newsize _*/),
                      CONST void       (*mfreefp)
                                       (/*_ dvoid *ctxp,
                                          dvoid *memptr _*/));
```

参数

参数	描述
mode(IN)	初始化的模式，目前只支持 OCI_DEFAULT – 默认模式

ctxp(IN)	保留兼容，忽略该参数
malocfp(IN)	保留兼容，忽略该参数
ralocfp(IN)	保留兼容，忽略该参数
mfreefp(IN)	保留兼容，忽略该参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCILogoff

描述

断开通过 OCILogon 与服务器建立的连接。

格式

```
sword OCILogoff ( OCISvcCtx      *svchp
                  OCIError       *errhp );
```

参数

参数	描述
svchp(IN)	要断开连接的上下文句柄
errhp(INOUT)	错误信息句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCILogon

描述

根据用户名和密码，登录到一个指定的数据库服务上，并初始化相关的上下文句柄。

格式

```
sword OCILogon ( OCIEnv          *envhp,
                 OCIError        *errhp,
                 OCISvcCtx       **svchp,
                 CONST OraText   *username,
                 ub4             uname_len,
                 CONST OraText   *password,
                 ub4             passwd_len,
                 CONST OraText   *dbname,
                 ub4             dbname_len );
```

参数

参数	描述
envhp(IN)	环境句柄
errhp(INOUT)	错误信息句柄
svchp(INOUT)	上下文句柄
username(IN)	登录的用户名
uname_len(IN)	登录的用户名长度
password(IN)	登录的口令

passwd_len(IN)	登录的口令长度
dbname(IN)	数据库服务名
dbname_len(IN)	数据库服务名长

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCIServerAttach

描述

附加一个数据库服务到指定的连接句柄上。

格式

```
sword OCIServerAttach( OCIServer      *srvhp,
                       OCIError       *errhp,
                       CONST text     *dblink,
                       sb4            dblink_len,
                       ub4            mode );
```

参数

参数	描述
srvhp(IN)	连接句柄
errhp(INOUT)	错误信息句柄
dblink(IN)	要关联的数据库服务名
dblink_len(IN)	数据库服务名的长度
mode(IN)	附加模式，目前只支持 OCI_DEFAULT

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7. OCIServerDetach

描述

解除连接句柄和数据库服务名之间的关联。

格式

```
sword OCIServerDetach( OCIServer    *srvhp,
                       OCIError     *errhp,
                       ub4          mode );
```

参数

参数	描述
srvhp(IN)	连接句柄
errhp(INOUT)	错误信息句柄
mode(IN)	附加模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

8. OCISessionBegin

描述

使用登录信息在指定的上下文句柄上打开与数据库服务的连接。

格式

```
sword OCISessionBegin ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCISession     *usrhp,
                        ub4             credt,
                        ub4             mode );
```

参数

参数	描述
svchp(IN)	指定打开连接的上下文
errhp(INOUT)	错误信息句柄
usrhp(IN)	登录信息句柄
credt(IN)	登录的方式，目前只支持 OCI_CRED_RDBMS– 通过用户名和口令与数据库服务建立起连接
mode(IN)	连接模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

9. OCISessionEnd

描述

结束 OCISessionBegin 函数中上下文句柄与数据库服务之间的连接。

格式

```
sword OCISessionEnd ( OCISvcCtx      *svchp,
                      OCIError       *errhp,
                      OCISession     *usrhp,
                      ub4             mode );
```

参数

参数	描述
svchp(IN)	指定断开连接的上下文
errhp(INOUT)	错误信息句柄
usrhp(IN)	登录信息句柄
mode(IN)	连接模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.2 Handle 和 Descriptor 方法

1. OCIAttrGet

描述

获取句柄上的属性值。

格式

```
sword OCIAttrGet ( CONST dvoid      *trgthndlp,
                   ub4              trghndltyp,
                   dvoid            *attributep,
                   ub4              *sizep,
                   ub4              attrtype,
                   OCIError         *errhp );
```

参数

参数	描述
trgthndlp(IN)	要获取属性的句柄
trghndltyp(IN)	trgthndlp 参数句柄的类型
attributep(OUT)	存放输出属性的缓冲区
size(OUT)	存放输出属性大小的缓冲区
attrtype(IN)	要获取的属性
errhp(IN)	错误信息句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

DCI 支持如下句柄的属性获取：

句柄	描述
OCI_HTYPE_ENV	环境句柄
OCI_HTYPE_ERROR	错误句柄
OCI_HTYPE_SVCCTX	上下文句柄
OCI_HTYPE_STMT	语句句柄
OCI_HTYPE_SERVER	服务器句柄
OCI_HTYPE_SESSION	会话句柄
OCI_HTYPE_TRANS	事务句柄
OCI_HTYPE_BIND	参数绑定句柄
OCI_HTYPE_DEFINE	列绑定句柄
OCI_HTYPE_DIRPATH_STREAM	dp 流句柄
OCI_HTYPE_DIRPATH_COLUMN_ARRAY	dp 列数据数组
OCI_HTYPE_DIRPATH_CTX	dp 运行环境句柄
OCI_DTYPE_PARAM	描述信息参数句柄
OCI_HTYPE_DESCRIBE	描述句柄

2. OCIAttrSet

描述

设置句柄的属性。

格式

```
sword OCIAttrSet ( dvoid        *trgthndlp,
                   ub4          trghndltyp,
                   dvoid        *attributep,
                   ub4          size,
                   ub4          attrtype,
                   OCIError     *errhp );
```

参数

参数	描述
trgthndlp(IN)	需要设置属性的句柄
trghndltyp(IN)	trgthndlp 参数句柄的类型
attributep(IN)	要设置属性值的指针
size(IN)	attributep 参数的大小
attrtype(IN)	要设置的句柄属性
errhp(IN)	错误信息句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

参考 OCIAttrSet 函数获取支持的句柄。

3. OCIDescriptorAlloc

描述

分配一个存贮大字段描述符句柄。

格式

```
sword OCIDescriptorAlloc( CONST dvoid     *parenth,
                          dvoid           **descpp,
                          ub4             type,
                          size_t          xtramem_sz,
                          dvoid           **usrmempp);
```

参数

参数	描述
parenth(IN)	环境句柄
descpp(OUT)	存贮大字段描述符指针
type(IN)	描述符类型，目前只支持 OCI_DTYPE_LOB 大字段描述、OCI_DTYPE_TIME –时间类型 OCI_DTYPE_TIMESTAMP-日期时间类型
xtramem_sz(IN)	额外附加分配的内存空间
usrmempp(IN)	返回额外分配的附加内存空间的地址

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

支持的描述句柄

句柄	描述

OCI_DTYPE_LOB	大字段句柄
OCI_DTYPE_TIME	时间类型句柄
OCI_DTYPE_TIMESTAMP	日期时间类型句柄

4. OCIDescriptorFree

描述

释放 OCIDescriptorAlloc 生成的描述符。

格式

```
sword OCIDescriptorFree( dvoid      *descp,
                         ub4        type);
```

参数

参数	描述
descp(IN)	描述符指针
type(IN)	描述符类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

参见 OCIDescriptorAlloc 说明中可分配的句柄。

5. OCIHandleAlloc

描述

分配和初始化各种句柄。

格式

```
sword OCIHandleAlloc ( CONST dvoid     *parenth,
                       dvoid           **hndlpp,
                       ub4             type,
                       size_t          xtramem_sz,
                       dvoid           **usrmempp );
```

参数

参数	描述
parentth(IN)	环境句柄
hndlpp(OUT)	句柄指针
type(IN)	句柄类型
xtramem_sz(IN)	额外附加分配的内存空间
usrmempp(IN)	返回额外分配的附加内存空间的地址

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

DCI 支持如下句柄的分配：

句柄	描述

OCI_HTYPE_ENV	环境句柄
OCI_HTYPE_ERROR	错误句柄
OCI_HTYPE_SVCCTX	上下文句柄
OCI_HTYPE_STMT	语句句柄
OCI_HTYPE_SERVER	服务器句柄
OCI_HTYPE_SESSION	会话句柄
OCI_HTYPE_TRANS	事务句柄
OCI_HTYPE_BIND	参数绑定句柄
OCI_HTYPE_DEFINE	列绑定句柄
OCI_HTYPE_DIRPATH_STREAM	dp 流句柄
OCI_HTYPE_DIRPATH_COLUMN_ARRAY	dp 列数据数组
OCI_HTYPE_DIRPATH_CTX	dp 运行环境句柄
OCI_HTYPE_DESCRIBE	描述句柄

6. OCIHandleFree

描述

释放由 OCIHandleAlloc 所分配的句柄。

格式

```
sword OCIHandleFree ( dvoid      *hndlp,
                      ub4        type );
```

参数

参数	描述
hndlp(IN)	句柄指针
type(IN)	句柄类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

参考 OCIHandleAlloc 获得句柄分配相关信息。

7. OCIParamGet

描述

获取描述符句柄上指定位置的描述符句柄。

格式

```
sword OCIParamGet ( CONST dvoid      *hndlp,
                    ub4              htype,
                    OCIError         *errhp,
                    dvoid            **parmdpp,
                    ub4              pos );
```

参数

参数	描述
hndlp(IN)	一个存在描述信息的句柄

htype(IN)	hndlp 参数句柄的类型
errhp(IN)	错误信息句柄
parmdpp(OUT)	输出的描述符句柄
pos(IN)	要获取描述符上指定位置的描述信息

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

dci 支持如下句柄的 param 句柄获取

句柄	描述
OCI_HTYPE_STMT	语句句柄
OCI_DTYPE_PARAM	参数句柄

7.3.1.3 Bind、Define 和 Describe 方法

1. OCIBindArrayOfStruct

描述

以数组方式进行参数绑定。

格式

```
sword OCIBindArrayOfStruct (
                                OCIBind        * bindp,
                                 OCIError      *errhp,
                                 ub4            pvskip,
                                 ub4            indskip,
                                 ub4            alskip,
                                 ub4            rcskip );
```

参数

参数	描述
bindp (IN)	绑定结构指针
errhp(INOUT)	错误信息句柄
pvskip(IN)	每个值间隔的大小，在一次性返回多行时，OCI 会跟据这个大小来回填每一行的值
indskip(IN)	空值指示符的间隔大小
alskip (IN)	返回值长度指示符的间隔大小
rcskip(IN)	指示多行绑定跳过的字节数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

2. OCIBindByName

描述

按参数名称绑定 SQL 语句中的参数。

格式

```
sword OCIBindByName ( OCIStmt       *stmtp,
                      OCIBind       **bindpp,
                      OCIError      *errhp,
                      CONST text    *placeholder,
                      sb4           placeh_len,
                      dvoid         *valuep,
                      sb4           value_sz,
                      ub2           dty,
                      dvoid         *indp,
                      ub2           *alenp,
                      ub2           *rcodep,
                      ub4           maxarr_len,
                      ub4           *curelep,
                      ub4           mode );
```

参数

参数	描述
stmtp(IN)	语句句柄
bindp(OUT)	绑定信息句柄
errhp(INOUT)	错误信息句柄
placeholder(IN)	绑定的参数名称
placeh_len(IN)	参数名称的长度
valuep(IN)	参数值缓冲区指针
value_sz(IN)	参数类型单个值的大小
dty(IN)	参数的数据类型
indp(IN)	保留参数，请置 NULL
alenp(IN)	保留参数，请置 NULL
rcodep(IN)	保留参数，请置 NULL
maxarr_len(IN)	保留参数
curelep(IN)	保留参数，请置 NULL
mode(IN)	绑定的模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

3. OCIBindByPos

描述

按参数在 SQL 语句中出现的位置进行绑定。

格式

```
sword OCIBindByPos ( OCIStmt       *stmtp,
                     OCIBind       **bindp,
                     OCIError      *errhp,
```

```
                ub4             position,
                dvoid           *valuep,
                sb4             value_sz,
                ub2             dty,
                dvoid           *indp,
                ub2             *alenp,
                ub2             *rcodep,
                ub4             maxarr_len,
                ub4             *curelep,
                ub4             mode );
```

参数

参数	描述
stmtp(IN)	语句句柄
bindp(OUT)	绑定信息句柄
errhp(INOUT)	错误信息句柄
position(IN)	参数在 SQL 中出现的位置，从 1 开始记数
valuep(IN)	参数值缓冲区指针
value_sz(IN)	参数类型单个值的大小
dty(IN)	参数的数据类型，不支持 SQLT_RDD、SQLT_NTY、SQLT_REF、SQLT_FILE SQLT_VST 类型
indp(IN)	保留参数，请置 NULL
alenp(IN)	保留参数，请置 NULL
rcodep(IN)	保留参数，请置 NULL
maxarr_len(IN)	保留参数
curelep(IN)	保留参数，请置 NULL
mode(IN)	绑定的模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCIBindDynamic

描述

动态绑定参数。

格式

```
sword OCIBindDynamic ( OCIBind        *bindp,
                       OCIError       *errhp,
                       dvoid          *ictxp,
                       OCICallbackInBind          (icbfp)(/*_
                              dvoid                *ictxp,
                              OCIBind              *bindp,
                              ub4                  iter,
                              ub4                  index,
                              dvoid                **bufpp,
```

```
                    ub4             *alenp,
                    ub1             *piecep,
                    dvoid           **indpp */),
                    dvoid           *octxp,
                OCICallbackOutBind  (ocbfp)(/*_
                    dvoid           *octxp,
                    OCIBind         *bindp,
                    ub4             iter,
                    ub4             index,
                    dvoid           **bufpp,
                    ub4             **alenpp,
                    ub1             *piecep,
                    dvoid           **indpp,
                    ub2             **rcodepp _*/)    );
```

参数

参数	描述
bindp (INOUT)	绑定参数句柄
errhp (INOUT)	错误句柄
ictxp(IN)	输入参数环境
icbfp(IN)	输入回调函数
octxp(IN)	输出参数环境
ocbfp(IN)	输出回调函数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCIDefineArrayOfStruct

描述

以数组方式进行列绑定。

格式

```
sword OCIDefineArrayOfStruct ( OCIDefine    *defnp,
                               OCIError     *errhp,
                               ub4          pvskip,
                               ub4          indskip,
                               ub4          rlskip,
                               ub4          rcskip );
```

参数

参数	描述
defnp (IN)	绑定结构指针
errhp(INOUT)	错误信息句柄
pvskip(IN)	每个值间隔的大小，在一次性返回多行时，OCI 会跟据这个大小来回填每一行的值

indskip(IN)	空值指示符的间隔大小
rlskip (IN)	返回值长度指示符的间隔大小
rcskip(IN)	指示多行绑定跳过的字节数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCIDefineByPos

描述

按位置来绑定查询返回结果集中每一列的取值空间。

格式

```
sword OCIDefineByPos ( OCIStmt      *stmtp,
                       OCIDefine    **defnpp,
                       OCIError     *errhp,
                       ub4          position,
                       dvoid        *valuep,
                       sb4          value_sz,
                       ub2          dty,
                       dvoid        *indp,
                       ub2          *rlenp,
                       ub2          *rcodep,
                       ub4          mode );
```

参数

参数	描述
stmtp(IN)	语句句柄
defnp(OUT)	绑定结构输出指针，可以通过使用该参数调用 OCIDefineArrayOfStruct 来指定该列每个值之间的间隔大小。
errhp(INOUT)	错误信息句柄
position(IN)	参数在 SQL 中出现的位置，从 1 开始记数
valuep(IN)	参数值缓冲区指针
value_sz(IN)	参数类型单个值的大小
dty(IN)	参数的数据类型
indp(IN)	指示符缓冲区，如果结果集中存在NULL值，那么OCI会回填该缓冲区，把对应的位置置为-1，其它情况下置0；如果在绑定的时候，未设置该指示符缓冲区，但是获取的结果中存又存在NULL值，该函数会返回OCI_ERROR的错误。
rlenp(IN)	返回值长度指示缓冲区，如果返回值未发生截断，那么该缓冲区中的值就是实际的返回值长度；如果返回值在获取的时候发生了截断，那么该缓冲区的值就是未截断以前的长度；如果返回值是空值，该缓冲区的值将被置为 0。
rcodep(IN)	保留参数
mode(IN)	绑定模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7. OCIDescribeAny

描述

连续写入内容到一个大字段存贮描述符中。

格式

```
sword OCIDescribeAny ( OCISvcCtx      *svchp,
                       OCIError       *errhp,
                       dvoid          *objptr,
                       ub4            objptr_len,
                       ub1            objptr_typ,
                       ub1            info_level,
                       ub1            objtyp,
                       OCIDescribe    *dschp );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
objptr(IN)	要被描述的对象指针，目前只支持字符串类型指针
objnm_len(IN)	objptr 参数中字符串的长度
objptr_typ(IN)	objptr 指针类型，目前只支持 OCI_OTYPE_NAME– 对象名称类型指针
info_level(IN)	存放读到数据的缓冲区指针
objtyp(IN)	objptr 参数所指的对象类型
dschp(IN)	一个描述符句柄，该句柄可以通过使用 OCI_HTYPE_DESCRIBE 作为参数调用 OCIHandleAlloc 分配得到

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

DCI 支持如下对象的描述：

句柄	描述
OCI_PTYPE_DATABASE	数据库，达梦不支持数据库概念，进行描述将会返回错误
OCI_PTYPE_SCHEMA	模式
OCI_PTYPE_TABLE	表
OCI_PTYPE_VIEW	视图
OCI_PTYPE_PROC	存储过程
OCI_PTYPE_FUNC	函数
OCI_PTYPE_PKG	包

7.3.1.4 Statement 方法

1. OCIStmtExecute

描述

执行准备好的语句。

格式

```
sword OCIStmtExecute( OCISvcCtx              *svchp,
                      OCIStmt                *stmtp,
                      OCIError               *errhp,
                      ub4                    iters,
                      ub4                    rowoff,
                      CONST OCISnapshot      *snap_in,
                      OCISnapshot            *snap_out,
                      ub4                    mode );
```

参数

参数	描述
svchp(IN)	环境句柄
stmtp(IN)	语句句柄
errhp(INOUT)	错误句柄
iters(IN)	对于非 select 语句来说，iters 和 rowoff 配合使用， iters-rowoff 的值表示该语句执行次数；对于 select 语句来说，iters 表示一次执行读取到 buffer 中的记录行数。假如不能确定 select 语句所返回的记录行数，可将 iters 设置为 0。如果 iters 不为 0，则该语句在执行前必须已经定义句柄（即需要定义位址空间来存储 select 出来的变量值）。
rowoff(IN)	在多行执行时，rowoff 表示输入变量在绑定的变量数组的开始位置(即记录偏移量)。用于多行记录的修改、插入或删除；比如，定义的绑定变量为 array[100]，执行该语句时，iters 的值为 80，rowoff 的值为 20，则表示从 array[20]开始提取传入值，而执行的次数则是 80-20 = 60（次）。
snap_in(IN)	不支持
snap_out(OUT)	不支持
mode(IN)	执行模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

说明

支持 mode 的模式：

模式	描述
OCI_COMMIT_ON_SUCCESS	成功后提交事务
OCI_DEFAULT	默认
OCI_DESCRIBE_ONLY	描述信息
OCI_EXACT_FETCH	获取行数据
OCI_PARSE_ONLY	语法分析

2. OCIStmtFetch

描述

提取 SQL 生成的结果集中的行集。当执行一条查询以后，可以多次调用该函数来返回

结果集中的所有行，直到该函数返回 OCI_NO_DATA。

格式

```
sword  OCIStmtFetch (OCIStmt *stmtp, OCIError *errhp, ub4 nrows, ub2 orientation, ub4 mode)
```

参数

参数	描述
stmtp(IN)	语句句柄
errhp(INOUT)	错误信息句柄
nrows(IN)	需要获取的行数
orientation (IN)	行集提取的方式
mode(IN)	提取模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR，出现警告信息返回 OCI_SUCCESS_WITH_INFO，超过行集末尾而无法获取数据 OCI_NO_DATA。

说明

支持的行集提取的方式

方式	描述
OCI_FETCH_NEXT	从当前游标位置向下进行提取操作
OCI_FETCH_FIRST	从结果集的第一行开始向下进行提取操作
OCI_FETCH_LAST	从结果集的最后一行开始向下提取操作
OCI_FETCH_PRIOR	从结果集的当前游标位置向上进行提取操作

3. OCIStmtFetch2

描述

可滚动获取行集。

格式

```
sword OCIStmtFetch2 ( OCIStmt      *stmthp,
                      OCIError     *errhp,
                      ub4          nrows,
                      ub2          orientation,
                      sb4          fetchOffset,
                      ub4          mode );
```

参数

参数	描述
stmthp (INOUT)	语句句柄
errhp (INOUT)	错误句柄
nrows(IN)	获取行数
orientation(IN)	游标方向
fetchOffset(IN)	偏移
mode(IN)	获取模式，不支持该参数，忽略其输入

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR，出现警告信息返回 OCI_SUCCESS_WITH_INFO，超过行集末尾而无法获取数据返回 OCI_NO_DATA。

说明

支持如下游标方向

方向	描述
OCI_FETCH_NEXT/OCI_DEFAULT	向后获取
OCI_FETCH_CURRENT	获取当前行
OCI_FETCH_FIRST	从行集头部获取
OCI_FETCH_LAST	从行集尾部获取
OCI_FETCH_PRIOR	向前获取
OCI_FETCH_ABSOLUTE	以绝对位置定位获取
OCI_FETCH_RELATIVE	以相对位置定位获取

4. OCIStmtGetPieceInfo

描述

获取获取运行时参数信息。

格式

```
sword OCIStmtGetPieceInfo( CONST OCIStmt   *stmtp,
                           OCIError        *errhp,
                           dvoid           **hndlpp,
                           ub4             *typep,
                           ub1             *in_outp,
                           ub4             *iterp,
                           ub4             *idxp,
                           ub1             *piecep );
```

参数

参数	描述
stmtp(IN)	语句句柄
errhp (INOUT)	错误句柄
hndlpp(OUT)	绑定句柄
typep (OUT)	绑定类型
in_outp (OUT)	输入输出类型
iterp (OUT)	获取循环计数信息
idxp (OUT)	不支持该参数，忽略
piecep (OUT)	分片信息

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCIStmtPrepare

描述

准备一条 SQL 语句，以便随后调用 OCIStmtExecute 来执行。

格式

```
sword OCIStmtPrepare( OCIStmt        *stmtp,
                      OCIError       *errhp,
                      CONST text     *stmt,
                      ub4            stmt_len,
                      ub4            language,
                      ub4            mode );
```

参数

参数	描述
stmtp(IN)	语句句柄
errhp(INOUT)	错误信息句柄
stmt(IN)	准备执行的 SQL 语句
stmt_len(IN)	stmt 参数中 SQL 语句的长度
language(IN)	保留参数
mode(IN)	准备模式，目前只支持 OCI_DEFAULT – 默认模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCIStmtSetPieceInfo

描述

设置运行时分批提供数据的信息。

格式

```
sword OCIStmtSetPieceInfo ( dvoid          *hndlp,
                            ub4            type,
                            OCIError       *errhp,
                            CONST dvoid    *bufp,
                            ub4            *alenp,
                            ub1            piece,
                            CONST dvoid    *indp,
                            ub2            *rcodep );
```

参数

参数	描述
hndlp (IN/OUT)	句柄，dci 仅支持绑定参数的句柄
type(IN)	忽略
errhp(IN OUT)	错误句柄
bufp (IN/OUT)	缓冲区指针
alenp (IN/OUT)	长度指针
piece (IN)	当前 piece
indp (IN/OUT)	指示符指针
rcodep (IN/OUT)	返回值指针

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.5 LOB 方法

1. OCILobGetLength

描述

返回大字段的长度，按字节计算。

格式

```
sword OCILobGetLength ( OCISvcCtx          *svchp,
                        OCIError           *errhp,
                        OCILobLocator      *locp,
                        ub4                *lenp );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	存贮大字段描述符指针
lenp(OUT)	返回的大字段长度，按字节计算

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

2. OCILobRead

描述

读取某个大字段中指定长度的内容。

格式

```
sword OCILobRead ( OCISvcCtx              *svchp,
                   OCIError               *errhp,
                   OCILobLocator          *locp,
                   ub4                    *amtp,
                   ub4                    offset,
                   dvoid                  *bufp,
                   ub4                    bufl,
                   dvoid                  *ctxp,
                   OCICallbackLobRead     (cbfp)
                                          ( dvoid              *ctxp,
                                            CONST dvoid        *bufp,
                                            ub4                len,
                                            ub1                piece
                                          )
                   ub2                    csid,
                   ub1                    csfrm );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	存贮大字段描述符指针
Amtp(IN/OUT)	该参数为输入输出参数，当调用函数时，该参数表明想读取的字节数，当函数执行完以后，OCI 会回填实际读到的字节数
offset	保留参数
bufp(IN)	存放读到数据的缓冲区指针
bufl(IN)	存放读到数据的缓冲区长度
ctxp	保留参数
cbfp	保留参数
csid	保留参数
csfrm	保留参数

返回值

如果执行成功，在还有数据未读取完的情况下，返回 OCI_NEED_DATA；如果数据都已经读完则返回 OCI_SUCCESS；执行失败则返回 OCI_ERROR。

3. OCILobWrite

描述

连续写入内容到一个大字段存贮描述符中。

格式

```
sword OCILobWrite ( OCISvcCtx          *svchp,
                    OCIError           *errhp,
                    OCILobLocator      *locp,
                    ub4                *amtp,
                    ub4                offset,
                    dvoid              *bufp,
                    ub4                buflen,
                    ub1                piece,
                    dvoid              *ctxp,
                    OCICallbackLobWrite    (cbfp)
                                       (
                                         dvoid     *ctxp,
                                         dvoid     *bufp,
                                         ub4       *lenp,
                                         ub1       *piecep
                                       )
                    ub2                csid,
                    ub1                csfrm );
```

参数

参数	描述

svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	存贮大字段描述符指针
amtp(IN/OUT）	该参数为输入输出参数，当调用函数时，该参数表明想读取的字节数，当函数执行完以后，OCI 会回填实际读到的字节数
offset	保留参数
bufp(IN)	存放读到数据的缓冲区指针
buflen(IN)	存放读到数据的缓冲区长度
piece	保留参数
ctxp	保留参数
cbfp	保留参数
csid	保留参数
csfrm	保留参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCILobCreateTemporary

描述

创建一个临时大字段。

格式

```
sword OCILobCreateTemporary(OCISvcCtx          *svchp,
                            OCIError           *errhp,
                            OCILobLocator      *locp,
                            ub2                csid,
                            ub1                csfrm,
                            ub1                lobtype,
                            boolean            cache,
                            OCIDuration        duration);
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	存贮大字段描述符指针
csid	保留参数
csfrm	保留参数
Lobtype(IN)	大字段类型 OCI_TEMP_BLOB OCI_TEMP_CLOB
Cache	保留参数
duration	保留参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCILobFreeTemporary

描述

释放临时大字段。

格式

```
sword OCILobFreeTemporary( OCISvcCtx          *svchp,
                           OCIError           *errhp,
                           OCILobLocator      *locp);
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	存贮大字段描述符指针

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCILobFileClose

描述

关闭一个已经打开的 BFILE

格式

```
sword OCILobFileClose ( OCISvcCtx          *svchp,
                        OCIError           *errhp,
                        OCILobLocator      *filep );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filep(INOUT)	BFILE 句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7. OCILobFileCLoseAll

描述

关闭所有在服务句柄上打开的 BFILE

格式

```
sword OCILobFileCLoseAll ( OCISvcCtx     *svchp,
                           OCIError      *errhp );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

8. OCILobFileExists

描述

判断是否存在指定的 BFILE

格式

```
sword OCILobFileExists ( OCISvcCtx          *svchp,
                         OCIError           *errhp,
                         OCILobLocator      *filep,
                         boolean            *flag );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filep(IN)	BFILE 句柄
flag(OUT)	TRUE 表示文件存在，FALSE 表示文件不存在

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

9. OCILobFileGetName

描述

获取 BFILE 所对应的目录对象及文件名

格式

```
sword OCILobFileGetName ( OCIEnv                 *envhp,
                          OCIError               *errhp,
                          const OCILobLocator    *filep,
                          OraText                *dir_alias,
                          ub2                    *d_length,
                          OraText                *filename,
                          ub2                    *f_length );
```

参数

参数	描述
svchp(INOUT)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filep(IN)	BFILE 句柄
dir_alias(OUT)	目录对象缓冲区
d_length(INOUT)	输入时为 dir_alias 缓冲区长度,输出时为实际的目录对象的长度
filename(OUT)	文件名缓冲区
f_length(INOUT)	输入时为 filename 缓冲区长度,输出时为实际文件名的长度

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

10. OCILobFileIsOpen

描述

判断 BFILE 是否已经打开

格式

```
sword OCILobFileIsOpen( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCILobLocator  *filep,
                        boolean        *flag );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filep(IN)	BFILE 句柄
flag(OUT)	TRUE 表示文件已打开，FALSE 表示文件未打开

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

11. OCILobFileOpen

描述

以只读方式打开一个 BFILE

格式

```
sword OCILobFileOpen ( OCISvcCtx      *svchp,
                       OCIError       *errhp,
                       OCILobLocator  *filep,
                       ub1            mode );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filep(INOUT)	BFILE 句柄
mode(IN)	文件打开方式,合法值为 OCI_FILE_READONLY

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

12. OCILobFileSetName

描述

设置 BFILE 对象的目录及文件名

格式

```
sword OCILobFileSetName ( OCIEnv          *envhp,
                          OCIError        *errhp,
                          OCILobLocator   **filepp,
                          const OraText   *dir_alias,
                          ub2             d_length,
                          const OraText   *filename,
                          ub2             f_length );
```

参数

参数	描述
svchp(INOUT)	上下文句柄指针
errhp(INOUT)	错误信息句柄
filepp(INOUT)	BFILE 句柄
dir_alias(IN)	设置的目录对象缓冲区
d_length(IN)	目录对象的长度
filename(IN)	设置的文件名对象缓冲区
f_length(IN)	文件名对象的长度

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

13. OCILobLoadFromFile

描述

加载 BFILE 文件的数据到 LOB 对象中

格式

```
sword OCILobLoadFromFile ( OCISvcCtx      *svchp,
                           OCIError       *errhp,
                           OCILobLocator  *dst_locp,
                           OCILobLocator  *src_locp,
                           ub4            amount,
                           ub4            dst_offset,
                           ub4            src_offset );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
dst_locp(INOUT)	LOB 句柄
src_locp(INOUT)	BFILE 句柄
amount(IN)	加载的数据总数
dst_offset(IN)	加载到目标 lob 对象的偏移,字符类型 lob 为字符偏移,字节类型 lob 为字节偏移,起始为 1
src_offset(IN)	BFILE 对象的起始偏移,起始为 1

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

14. OCILobLocatorAssign

描述

赋值一个 LOB 对象到另外一个

格式

```
sword OCILobLocatorAssign ( OCISvcCtx              *svchp,
                            OCIError               *errhp,
                            const OCILobLocator    *src_locp,
                            OCILobLocator          **dst_locpp );
```

参数

参数	描述
svchp(INOUT)	上下文句柄指针
errhp(INOUT)	错误信息句柄
src_locp(IN)	被复制的 LOB/BFILE 句柄
dst_locp(INOUT)	需要设置的 LOB/BFILE 句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

15. OCILobAssign

描述

赋值一个 LOB 对象到另外一个

格式

```
sword OCILobAssign ( OCISvcCtx              *svchp,
                     OCIError               *errhp,
                     const OCILobLocator    *src_locp,
                     OCILobLocator          **dst_locpp );
```

参数

参数	描述
svchp(INOUT)	上下文句柄指针
errhp(INOUT)	错误信息句柄
src_locp(IN)	被复制的 LOB/BFILE 句柄
dst_locp(INOUT)	需要设置的 LOB/BFILE 句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

16. OCILobLocatorIsInit

描述

判断给定的 LOB/BFILE 句柄是否已经初始化

格式

```
sword OCILobLocatorIsInit ( OCIEnv              *envhp,
                            OCIError            *errhp,
```

```
                        const OCILobLocator    *locp,
                        boolean                *is_initialized   );
```

参数

参数	描述
svchp(INOUT)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(IN)	LOB/BFILE 句柄
is_initialized (OUT)	TRUE 表示已初始化,FALSE 表示未初始化

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

17. OCILobOpen

描述

打开一个 LOB/BFILE 对象

格式

```
sword OCILobOpen ( OCISvcCtx          *svchp,
                   OCIError           *errhp,
                   OCILobLocator      *locp,
                   ub1                mode );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(INOUT)	LOB/BFILE 句柄
mode (IN)	打开模式

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

18. OCILobClose

描述

关闭一个 LOB/BFILE 对象

格式

```
sword OCILobClose ( OCISvcCtx         *svchp,
                    OCIError          *errhp,
                    OCILobLocator     *locp );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
errhp(INOUT)	错误信息句柄
locp(INOUT)	LOB/BFILE 句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

19. OCILobIsEqual

描述

判断给定的 LOB/BFILE 句柄是否相等

格式

```
sword OCILobIsEqual ( OCIEnv                    *envhp,
                      const OCILobLocator       *x,
                      const OCILobLocator       *y,
                      boolean                   *is_equal );
```

参数

参数	描述
svchp(IN)	上下文句柄指针
x(IN)	LOB/BFILE 句柄
y(IN)	LOB/BFILE 句柄
is_equal(OUT)	TRUE 表示相等,FALSE 表示不等

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.6 Direct Path Loading 方法

1. OCIDirPathAbort

描述

取消当前装载的数据

格式

```
sword OCIDirPathAbort    ( OCIDirPathCtx     *dpctx,
                           OCIError          *errhp );
```

参数

参数	描述
dpctx(IN)	dp 环境句柄
errhp (INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

2. OCIDirPathColArrayEntryGet

描述

获取 direct path 列绑定的信息。

格式

```
sword OCIDirPathColArrayEntryGet ( OCIDirPathColArray     *dpca,
                                   OCIError               *errhp,
                                   ub4                    rownum,
                                   ub2                    colIdx,
                                   ub1                    **cvalpp,
                                   ub4                    *clenp,
                                   ub1                    *cflgp );
```

参数

参数	描述
dpca (IN/OUT)	direct pathcolumn array 句柄
errhp (INOUT)	错误句柄
rownum (IN)	行偏移
colIdx (IN)	列号
cvalpp (IN/OUT)	绑定地址缓冲区
clenp (IN/OUT)	长度缓冲区
cflgp (IN/OUT)	标志缓冲区

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

3. OCIDirPathColArrayEntrySet

描述

绑定 direct path 行数据。

格式

```
sword OCIDirPathColArrayEntrySet ( OCIDirPathColArray     *dpca,
                                   OCIError               *errhp,
                                   ub4                    rownum,
                                   ub2                    colIdx,
                                   ub1                    *cvalp,
                                   ub4                    clen,
                                   ub1                    cflg );
```

参数

参数	描述
dpca(IN/OUT)	direct pathcolumn array 句柄
errhp(INOUT)	错误句柄
rownum(IN)	行偏移
colIdx(IN)	行号
cvalp(IN)	数据地址
clen(IN)	长度
cflg(IN)	数据标志

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCIDirPathColArrayReset

描述

清除 direct path 列绑定的信息。

格式

```
sword OCIDirPathColArrayReset ( OCIDirPathColArray      *dpca,
                                OCIError                *errhp );
```

参数

参数	描述
dpca(IN)	direct pathcolumn array 句柄
errhp(INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCIDirPathColArrayToStream

描述

将绑定的行数据转换为发送到服务的流数据。

格式

```
sword OCIDirPathColArrayToStream ( OCIDirPathColArray      *dpca,
                                   OCIDirPathCtx  const    *dpctx,
                                   OCIDirPathStream        *dpstr,
                                   OCIError                *errhp,
                                   ub4                     rowcnt,
                                   ub4                     rowoff );
```

参数

参数	描述
dpca (IN)	direct pathcolumn array 句柄
dpctx (IN)	direct path 环境句柄
dpstr (IN /OUT)	direct path 流句柄
errhp (INOUT)	错误句柄
rowcnt (IN)	转换行数
rowoff (IN)	转换行偏移

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCIDirPathDataSave

描述

将数据永久性保存至服务器。

格式

```
sword OCIDirPathDataSave ( OCIDirPathCtx        *dpctx,
                           OCIError             *errhp,
```

```
                                ub4                     action    );
```

参数

参数	描述
dpctx(IN)	direct path 环境句柄
errhp(INOUT)	错误句柄
action(IN)	指示进行数据保存后的行为。 仅支持 OCI_DIRPATH_DATASAVE_SAVEONLY 选项，忽略其他选项

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7. OCIDirPathFinish

描述

完成数据装载。

格式

```
sword OCIDirPathFinish (    OCIDirPathCtx           *dpctx,
                            OCIError                *errhp );
```

参数

参数	描述
dpctx(IN)	direct path 环境句柄
errhp(INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

8. OCIDirPathLoadStream

描述

将流数据发送到服务器。

格式

```
sword OCIDirPathLoadStream (    OCIDirPathCtx           *dpctx,
                                OCIDirPathStream        *dpstr,
                                OCIError                *errhp );
```

参数

参数	描述
dpctx(IN)	direct path 环境句柄
dpstr(IN)	direct path 流句柄
errhp(INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

9. OCIDirPathPrepare

描述

准备装载数据。

格式

```
sword OCIDirPathPrepare (    OCIDirPathCtx          *dpctx,
                             OCISvcCtx              *svchp,
                             OCIError               *errhp );
```

参数

参数	描述
dpctx(IN)	direct path 环境句柄
svchp(IN)	上下文句柄
errhp(INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

10. OCIDirPathStreamReset

描述

清除流句柄中的数据，为下一次装载做准备。

格式

```
sword OCIDirPathStreamReset ( OCIDirPathStream       *dpstr,
                              OCIError               *errhp );
```

参数

参数	描述
dpstr(IN)	direct path 流句柄
errhp(INOUT)	错误句柄

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.7 Transaction 方法

1. OCITransCommit

描述

提交 SQL 的执行动作。

格式

```
sword OCITransCommit ( OCISvcCtx      *svchp,
                       OCIError       *errhp,
                       ub4            flags );
```

参数

参数	描述
svchp(IN)	上下文句柄

errhp(INOUT)	错误信息句柄
flags(IN)	保留参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

2. OCITransDetach

描述

将事务从会话剥离。

格式

```
sword OCITransDetach( OCISvcCtx     *svchp,
                      OCIError      *errhp,
                      ub4           flags );
```

参数

参数	描述
svchp(IN)	上下文句柄
errhp(INOUT)	错误信息句柄
flags(IN)	保留参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

3. OCITransRollback

描述

回滚 SQL 的执行动作。

格式

```
sword OCITransRollback ( dvoid          *svchp,
                         OCIError       *errhp,
                         ub4            flags );
```

参数

参数	描述
svchp(IN)	上下文句柄
errhp(INOUT)	错误信息句柄
flags(IN)	保留参数

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCITransStart

描述

启动事务。

格式

```
word OCITransStart( OCISvcCtx     *svchp,
```

```
                    OCIError      *errhp,
                    uword         timeout,
                    ub4           flags );
```

参数

参数	描述
svchp (INOUT)	上下文句柄
errhp (INOUT)	错误句柄
timeout(IN)	事务超时，不支持该设置，忽略输入
flags(IN)	事务属性标识

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.8 Miscellaneous 方法

1. OCIErrorGet

描述

获取 OCI 操作失败产生的错误信息。

格式

```
sword OCIErrorGet ( dvoid       *hndlp,
                    ub4         recordno,
                    text        *sqlstate,
                    sb4         *errcodep,
                    text        *bu      fp,
                    ub4         bufsiz,
                    ub4         type);
```

参数

参数	描述
hndlp(IN)	错误信息句柄或环境信息句柄
recordno(IN)	错误信息位置，目前只支持一条错误信息，所以请把该参数置为 1
sqlstate(OUT)	错误的状态码
errcodep(OUT)	错误码
bufp(OUT)	错误的描述信息
bufsiz(IN)	bufp 所指的缓冲区大小
type(IN)	句柄类型（错误句柄或环境句柄）

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.3.1.9 Date_datetime_inteval_fun 方法

1. OCIDateTimeFromText

描述

从字符串转化成 OCIDateTime*类型。

格式

```
sword OCIDateTimeFromText (    dvoid              *hndl,
                               OCIError           *err,
                               const OraText      *date_str,
                               size_t             dstr_length,
                               const OraText      *fmt,
                               ub1                fmt_length,
                               const OraText      *lang_name,
                               size_t             lang_length,
                               OCIDateTime        *datetime );
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
date_str(IN)	时间字符串
dstr_length(IN)	时间字符串长度
fmt(IN)	时间转换格式
fmt_length(IN)	时间转换格式长度
lang_name(IN)	语言串
lang_length(IN)	语言串长度
datetime(IN/OUT)	OCIDateTime*类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

2. OCIDateTimeToText

描述

从 OCIDateTime*类型转换成字符串。

格式

```
sword OCIDateTimeToText ( dvoid              *hndl,
                          OCIError           *err,
                          OCIDateTime        *datetime
                          const OraText      *fmt,
                          ub1                fmt_length,
                          ub1                fsprec,
                          const OraText      *lang_name,
                          size_t             lang_length,
```

```
                    size_t              *buf_size,
                    const OraText       *buf
                    );
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
datetime(IN/OUT)	OCIDateTime*类型
fmt(IN)	时间转换格式
fmt_length(IN)	时间转换格式长度
fsprec(IN)	保留毫秒位数
lang_name(IN)	语言串
lang_length(IN)	语言串长度
date_str(IN)	时间字符串缓冲区
dstr_length(IN)	时间字符串缓冲区长度

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

3. OCIDateTimeGetDate

描述

从 OCIDateTime*类型获取日期。

格式

```
OCIDateTimeGetDate(       dvoid      *hndl,
                      OCIError    *err,
                      CONST       OCIDateTime *datetime,
                      sb2        *yr,
                      ub1        *mnth,
                      ub1        *dy);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
datetime(IN)	OCIDateTime*类型
yr（OUT）	年
mnth（OUT）	月
dy（OUT）	日

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

4. OCIDateTimeGetTime

描述

从 OCIDateTime*类型获取时间。

格式

```
OCIDateTimeGetTime(    dvoid      *hndl,
                       OCIError    *err,
                       CONST       OCIDateTime *datetime,
                       ub1        *hr,
                       ub1        *mm,
                       ub1        *ss,
                       ub4        *fsec);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
datetime(IN)	OCIDateTime*类型
hr（OUT）	时
mm（OUT）	分
ss（OUT）	秒
fsec	毫秒

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

5. OCIIntervalSetYearMonth

描述

设置 OCIinterval 间隔年月类型的年和月。

格式

```
OCIIntervalSetYearMonth(      dvoid      *hndl,
                       OCIError    *err,
                       sb4        yr,
                       sb4        mnth,
                       OCIInterval  *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
yr（IN）	年
mnth（IN）	月
result（OUT）	间隔年月类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

6. OCIIntervalGetYearMonth

描述

获取 OCIinterval 间隔年月类型的年和月。

格式

```
OCIIntervalGetYearMonth(dvoid          *hndl,
                        OCIError       *err,
                         sb4           *yr,
                         sb4           *mnth,
   CONST OCIInterval *interval);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
yr（OUT）	年
mnth（OUT）	月
result（IN）	间隔年月类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7. OCIIntervalSetDaySecond

描述

设置 OCIinterval 间隔日时类型的日、时、分和秒。

格式

```
OCIIntervalSetDaySecond(      dvoid      *hndl,
                         OCIError   *err,
                         sb4        dy,
                         sb4        hr,
                         sb4        mm,
                         sb4        ss,
                         sb4        fsec,
                         OCIInterval *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
dy（IN）	日
Hr (IN)	时
Mm(IN)	分
ss（IN）	秒
Fsec(IN)	毫秒
result（OUT）	间隔日时类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

8. OCIIntervalGetDaySecond

描述

获取 OCIinterval 间隔日时类型的日、时、分和秒。

格式

```
OCIIntervalGetDaySecond(      dvoid      *hndl,
                        OCIError   *err,
                        sb4        dy,
                        sb4        hr,
                        sb4        mm,
                        sb4        ss,
                         sb4        fsec,
                         OCIInterval *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
dy（OUT）	日
hr (OUT)	时
mm(OUT)	分
ss（OUT）	秒
fsec(OUT)	毫秒
result(IN）	间隔日时类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

9. OCIIntervalCompare

描述

比较两个 OCIinterval 类型的值是否相等。

格式

```
OCIIntervalCompare(       dvoid       *hndl,
                      OCIError    *err,
                      OCIInterval   *inter1,
                      OCIInterval   *inter2,
                      sword       *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
inter1 (IN）	间隔类型
inter12(IN）	间隔类型
result(OUT)	结果

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

10. OCIIntervalAdd

描述

返回两个 interval 类型的和。

格式

```
OCIIntervalAdd(        dvoid      *hndl,
                    OCIError    *err,
                    OCIInterval  *addend1,
                    OCIInterval  *addend2,
                    OCIInterval  *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
inter1 (IN）	间隔类型
inter12(IN）	间隔类型
result(OUT)	结果

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

11. OCIIntervalAssign

描述

OCIinterval 类型的赋值。

格式

```
OCIIntervalAssign(        dvoid          *hndl,
                    OCIError          *err,
                    CONST OCIInterval *ininter,
                    OCIInterval       *outinter);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
ininter (IN）	间隔类型
outinter(OUT）	间隔类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

12. OCIIntervalFromText

描述

OCIinterval 类型的值的从文本构造。

格式

```
OCIIntervalFromText(        dvoid          *hndl,
                        OCIError       *err,
                        CONST OraText  *inpstr,
                        size_t          str_len,
                        OCIInterval     *result);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
inpstr (IN）	Interval 字符串
str_len(IN）	串长度
result(OUT)	返回的 OCIinterval*类型

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

13. OCIIntervalCheck

描述

OCIinterval 类型的值的检查。

格式

```
OCIIntervalCheck(          dvoid             *hndl,
                        OCIError         *err,
                        CONST OCIInterval *interval,
                        ub4              *valid);
```

参数

参数	描述
hndl(IN)	会话信息句柄
err(INOUT)	错误信息句柄
interval (IN）	间隔类型
valid(OUT）	是否合法

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

14. OCINumberToInt

描述

OCINumber 类型转化成 Int。

格式

```
OCINumberToInt(           OCIError      *err,
                       CONST OCINumber *number,
                       uword         rsl_length,
                       uword         rsl_flag,
                       dvoid         *rsl);
```

参数

参数	描述
err(INOUT)	错误信息句柄
number(IN)	要转化的 OCINumber
rsl_length(IN)	Int 缓冲区的大小
rsl_flag(IN)	输出的标识：OCI_NUMBER_UNSIGNED 或 OCI_NUMBER_SIGNED
rsl(IN)	int 指针

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

15. OCINumberFromInt

描述

从 Int 转化成 OCINumber 类型。

格式

```
OCINumberFromInt(         OCIError    *err,
                       CONST dvoid *inum,
                       uword      inum_length,
                       uword      inum_s_flag,
                       OCINumber   *number);
```

参数

参数	描述
err(INOUT)	错误信息句柄
inum (IN)	int 指针
inum_length (IN)	Int 缓冲区的大小
inum_s_flag (IN)	标识：OCI_NUMBER_UNSIGNED 或 OCI_NUMBER_SIGNED
number(OUT)	转化后的 OCINumber

返回值

成功返回 OCI_SUCCESS，失败返回 OCI_ERROR。

7.4 使用 DM DCI 编程基本步骤

一个应用程序中，我们是通过调用 DCI 提供的库函数来实现对达梦数据库的操纵的。为了与 oracle 兼容，DCI 提供的函数都是以 OCI 开头的与 OCI 同名的函数，比如创建 OCI 环境的 OCI 函数：OCIEnvCreate()。OCI 函数的一个特点或者说是难点就是它的参数特别多，函数往往都有十几个参数。一个 DCI 应用程序的基本结构包括：

1. 初始化 OCI 环境和线程；
2. 分配必要的句柄与数据结构；
3. 建立与数据库的连接以及创建用户会话；
4. 通过 SQL 与 DM 服务器交换数据，而后再做数据处理；
5. 结束用户会话与断开与数据库的连接；
6. 释放在程序中所分配的句柄。

如图 7.1 所示：

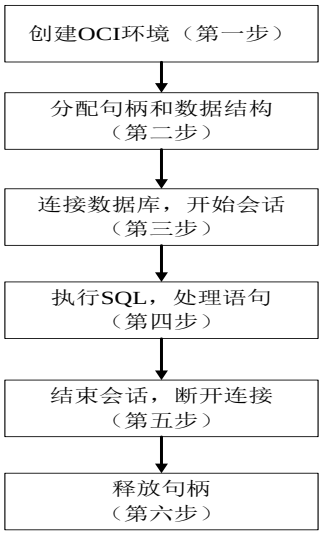


图 7.1 DCI 程序步骤

一个 SQL 语句在 OCI 应用程序中的执行步骤一般如下：

1. 准备 SQL 语句。调用函数 OCIStmtPrepare( )；
2. 在 SQL 语句中绑定需要输入到 SQL 语句中的变量。对于 DML 语句来说，由于它带有输入变量，我们可以通过调用一个或者多个函数 OCIBindByPos()、OCIBindByName()等把输入变量的地址绑定在 DML 语句中的占位符中；
3. 执行 SQL 语句。调用 OCIStmrExecute()函数。对于 DDL 语句到这一步就完成了一个语句的执行；
4. 描述 SQL 中输出的数据。如果有必要的话，我们可以调用函数 OCIParamGet()与 OCIAttGet( )来获取我们所读取的记录的字段个数、字段的数据类型以及字段数据定义的最大长度。

过程以及过程中调用到的函数如图 7.2 所示：

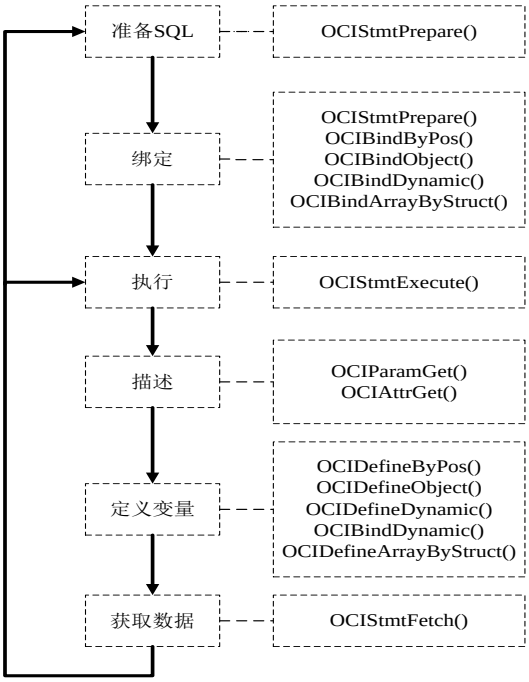


图 7.2 过程以及过程中调用到的函数

图 7.2 是一个一般 SQL 执行的流程图，对于不同的 SQL 语句，所需要的步骤也有所不同。对于 DDL 语句，由于没有数据的输入与输出，仅仅涉及到一些权限与定义或者删除数据库中的对象的问题，因此只需要上图的第一步与第三步便可以了。而对于 DML 语句，由于有数据的输入与输出，因此需要的步骤就多一些。

下面通过一个简单的 Windows 环境下的示例来说明 DM DCI 的调用方式。实现了对 bookshop 示例库中的 person 模式下的 person 表进行的增、删、改、查的基本操作。在编译的过程中需要用到 DM 的头文件 dci.h，在连接阶段需要用到 dmoci.lib 这个库文件，在执行阶段需要用到动态库 dmoci.dll、dmcalc.dll、dmelog.dll、dmmen.dll、dmos.dll、dmutl.dll、dmclientlex.dll、dmfldr_dll.dll、dmbcast.dll、dmdta.dll、dmcfg.dll、dmstrt.dll、dmcpr.dll、dmcyt.dll、dmcvt.dll、dmmout.dll、dmdpi.dll、dmcomm.dll。

```
/*************************************************************************/
/* DCI编程实例                                                           */
/*************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include "DCI.h"

/*    声明句柄  */
OCIEnv          *envhp;             /*    环境句柄        */
OCISvcCtx       *svchp;             /*    服务环境句柄    */
OCIServer       *srvhp;             /*    服务器句柄      */
OCISession      *authp;             /*    会话句柄        */
OCIStmt         *stmthp;            /*    语句句柄        */
OCIDescribe     *dschp;             /*    描述句柄        */
```

```
OCIError        *errhp;             /*   错误句柄        */
OCIDefine       *defhp[3];          /*   定义句柄        */
OCIBind         *bidhp [4];         /*   绑定句柄        */
sb2             ind[3];             /*   指示符变量      */
/*   绑定select结果集的参数   */
text            szpersonid[11];     /*   存储personid列  */
text            szsex[2];           /*   存储sex列  */
text            szname[51];         /*   存储name列      */
text            szemail[51];        /*   存储mail列      */
text            szphone[26];        /*   存储phone列     */
char            sql[256];           /*   存储执行的sql语句*/
int main(int argc, char *argv[])
{
    char strServerName[50];
    char strUserName[50];
    char strPassword[50];
    int ret;
    text errbuf[100];
    /*   设置服务器，用户名和密码      */
    strcpy(strServerName,"localhost");
    strcpy(strUserName,"SYSDBA");
    strcpy(strPassword,"SYSDBA");
    /* 初始化OCI应用环境*/
    OCIInitialize(OCI_DEFAULT, NULL, NULL, NULL, NULL);
    /* 初始化环境句柄   */
    OCIEnvInit(&envhp, OCI_DEFAULT,0, 0);
    /*   分配句柄        */
    OCIHandleAlloc(envhp, (dvoid**)&svchp, OCI_HTYPE_SVCCTX, 0, 0);      /*服务器环境句柄*/
    OCIHandleAlloc(envhp, (dvoid**)&srvhp, OCI_HTYPE_SERVER, 0, 0);      /*   服务器句柄*/
    OCIHandleAlloc(envhp, (dvoid**)&authp, OCI_HTYPE_SESSION, 0, 0);   /* 会话句柄  */
    OCIHandleAlloc(envhp, (dvoid**)&errhp, OCI_HTYPE_ERROR, 0, 0);       /*   错误句柄  */
    OCIHandleAlloc(envhp, (dvoid**)&dschp, OCI_HTYPE_DESCRIBE,0,0);      /*描述符句柄*/
    /*   连接服务器      */
    OCIServerAttach(srvhp, errhp,(text *)strServerName,
        (sb4)strlen(strServerName),OCI_DEFAULT);
    /*   设置用户名和密码   */
    OCIAttrSet(authp,OCI_HTYPE_SESSION,(text *)strUserName,
        (ub4)strlen(strUserName),OCI_ATTR_USERNAME,errhp);
    OCIAttrSet(authp,OCI_HTYPE_SESSION,(text *)strPassword,
        (ub4)strlen(strPassword), OCI_ATTR_PASSWORD,errhp);
    /*   设置服务器环境句柄属性  */
    OCIAttrSet ((dvoid*)svchp, (ub4) OCI_HTYPE_SVCCTX,
        (dvoid*)srvhp, (ub4) 0, OCI_ATTR_SERVER, errhp);
    OCIAttrSet(svchp, OCI_HTYPE_SVCCTX,(dvoid*)authp,
```

```
        0, OCI_ATTR_SESSION, errhp);
/*      创建并开始一个用户会话  */
OCISessionBegin  (svchp, errhp, authp,OCI_CRED_RDBMS,OCI_DEFAULT);
OCIHandleAlloc(envhp, (dvoid**)&stmthp,OCI_HTYPE_STMT, 0, 0);          /*      语句句柄  */
/*****************************************************************************/
/* 查询person表                                                              */
/*****************************************************************************/
strcpy(sql, "select personid, name, phone from person.person");
/*      准备SQL语句             */
OCIStmtPrepare(stmthp, errhp,(text *)sql, strlen(sql),OCI_NTV_SYNTAX, OCI_DEFAULT);
/*      绑定输出列              */
OCIDefineByPos (stmthp,&defhp[0],errhp, 1,(ub1*)szpersonid,
        sizeof(szpersonid),SQLT_STR,&ind[0], 0, 0, OCI_DEFAULT);
OCIDefineByPos (stmthp,&defhp[1],errhp, 2,(ub1*)szname,
        sizeof(szname),SQLT_STR,&ind[1], 0, 0, OCI_DEFAULT);
OCIDefineByPos (stmthp,&defhp[2],errhp, 3,(ub1*)szphone,
        sizeof(szphone),SQLT_STR,&ind[2], 0, 0, OCI_DEFAULT);
/*      执行SQL语句             */
ret=OCIStmtExecute(svchp, stmthp,errhp, (ub4)0, 0, NULL, NULL, OCI_DEFAULT);
if (ret != 0)
{
        OCIErrorGet(errhp, 1, NULL, &ret,    (OraText *)errbuf, sizeof(errbuf), OCI_HTYPE_ERROR);
        printf("%s\n",errbuf);
}

printf("%-10s%-10s%-10s\n", "PERSONID", "NAME", "PHONE");
while((OCIStmtFetch(stmthp, errhp,1,OCI_FETCH_NEXT,OCI_DEFAULT))!=OCI_NO_DATA)
{
        printf("%-10s", szpersonid);
        printf("%-10s", szname);
        printf("%-10s\n", szphone);
}
/*****************************************************************************/
/* 向person表插入一条数据                                                    */
/*****************************************************************************/
memset(sql, 0, sizeof(sql));
strcpy(sql, "insert into person.person(sex, name, email, phone) values(:sex,:name,:email,:phone)");
/*      准备SQL语句             */
OCIStmtPrepare(stmthp, errhp,(text *)sql, strlen(sql),OCI_NTV_SYNTAX, OCI_DEFAULT);
/*      设置输入参数   */
memset(szsex, 0, sizeof(szsex));
strcpy(szsex, "M");
memset(szname, 0, sizeof(szname));
strcpy(szname, "张三");
```

```
    memset(szemail, 0, sizeof(szemail));
    strcpy(szemail, "zhangsan@dameng.com");
    memset(szphone, 0, sizeof(szphone));
    strcpy(szphone, "027-87588000");
    /*    绑定输入列          */
    OCIBindByName (stmthp, &bidhp[0], errhp, ":sex", 4, szsex, strlen((char*)szsex),
        SQLT_AFC, NULL, NULL, NULL, 0, NULL, 0);
    OCIBindByName (stmthp, &bidhp[1], errhp, ":name", 5, szname, strlen((char*)szname),
        SQLT_AFC, NULL, NULL, NULL, 0, NULL, 0);
    OCIBindByName (stmthp, &bidhp[2], errhp, ":email", 6, szemail, strlen((char*)szemail),
        SQLT_AFC, NULL, NULL, NULL, 0, NULL, 0);
    OCIBindByName (stmthp, &bidhp[3], errhp, ":phone", 6, szphone, strlen((char*)szphone),
        SQLT_AFC, NULL, NULL, NULL, 0, NULL, 0);

    /*    执行SQL语句          */
    ret=OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4) 0, (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
(ub4) OCI_DEFAULT);
    if (ret != 0)
    {
        OCIErrorGet(errhp, 1, NULL, &ret,   (OraText *)errbuf, sizeof(errbuf), OCI_HTYPE_ERROR);
        printf("%s\n",errbuf);
    }
    /*    提交到数据库   */
    OCITransCommit(svchp, errhp, OCI_DEFAULT);
    /*************************************************************************/
    /*  更新person表                                                         */
    /*************************************************************************/
    memset(sql, 0, sizeof(sql));
    strcpy(sql, "update person.person set sex='M',name='Liuhuan',email='liujian@mail',phone='13636396811'
WHERE personid='1'");
    /*    准备SQL语句          */
    OCIStmtPrepare(stmthp, errhp,(text *)sql, strlen(sql),OCI_NTV_SYNTAX, OCI_DEFAULT);
    /*    执行SQL语句          */
    ret=OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4) 0, (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
(ub4) OCI_DEFAULT);
    if (ret != 0)
    {
        OCIErrorGet(errhp, 1, NULL, &ret,   (OraText *)errbuf, sizeof(errbuf), OCI_HTYPE_ERROR);
        printf("%s\n",errbuf);
    }
    /*    提交到数据库   */
    OCITransCommit(svchp, errhp, OCI_DEFAULT);
    /*************************************************************************/
    /*  删除person表的ID为1的数据，首先要在数据库中存在这条记录       */
```

```
    /*****************************************************************************/
    memset(sql, 0, sizeof(sql));
    strcpy(sql, "delete from person.person WHERE personid=:1");
    /*    准备SQL语句   */
    OCIStmtPrepare(stmthp, errhp,(text *)sql, strlen(sql),OCI_NTV_SYNTAX, OCI_DEFAULT);
    /*    绑定输入参数   */
    memset(szpersonid, 0, sizeof(szpersonid));
    strcpy(szpersonid, "20");
    OCIBindByPos(stmthp, &bidhp[0], errhp, 1, szpersonid, strlen((char*)szpersonid),
        SQLT_AFC, NULL, NULL, NULL, 0, NULL, 0);
    /*    执行SQL语句   */
    ret=OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4) 0, (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
(ub4) OCI_DEFAULT);
    if (ret != 0)
    {
        OCIErrorGet(errhp, 1, NULL, &ret,   (OraText *)errbuf, sizeof(errbuf), OCI_HTYPE_ERROR);
        printf("%s\n",errbuf);
    }
    /*    提交到数据库   */
    OCITransCommit(svchp, errhp, OCI_DEFAULT);
    //结束会话
    OCISessionEnd(svchp, errhp, authp, (ub4) 0);
    //断开与数据库的连接
    OCIServerDetach(srvhp, errhp, OCI_DEFAULT);
    //释放OCI句柄

    OCIHandleFree((dvoid*)dschp, OCI_HTYPE_DESCRIBE);
    OCIHandleFree((dvoid*)stmthp, OCI_HTYPE_STMT );
    OCIHandleFree((dvoid*)errhp, OCI_HTYPE_ERROR);
    OCIHandleFree((dvoid*)authp, OCI_HTYPE_SESSION );
    OCIHandleFree((dvoid*)svchp, OCI_HTYPE_SVCCTX);
    OCIHandleFree((dvoid*)srvhp, OCI_HTYPE_SERVER);
    return 0;
}
```

第 8 章 DM FLDR 编程指南

8.1 DM FLDR 接口介绍

快速装载接口 FLDR 是达梦数据库提供的能够快速将文本数据载入 DM 数据库的一种数据载入方式。用户通过使用 FLDR 接口能够把按照一定格式排序的文本数据以简单、快速、高效的方式载入到达梦数据库中。

8.2 DM FLDR 接口说明

8.2.1 返回值说明

DM FLDR 接口有以下五种返回值：

FLDR_SUCCESS：接口成功执行

FLDR_ERROR：接口执行出错

FLDR_INVALID_HANDLE：用户使用错误的句柄

FLDR_SUCCESS_WITH_INFO：接口执行成功但有警告信息

FLDR_NO_DATA：数据未找到

8.2.2 接口说明

1. fldr_alloc

描述：

用于分配快速装载对象实例，所有快速装载都将通过该句柄进行操作。

格式：

```
FLDR_RETURN
    fldr_alloc(
    fhinstance        *instance
);
```

参数：

instance（输出） 快速装载实例的句柄

返回值：

FLDR_SUCCESS、FLDR_ERROR

注释：

该函数分配出用于快速装载实例句柄，用户必须首先调用该函数获得快速装载的实例句柄，只有通过该句柄才能进行后续的快速装载操作

2. fldr_free

描述：

释放快速装载实例句柄。

格式：

```
FLDR_RETURN
fldr_free(
fhinstance        instance
);
```

参数：

instance（输入） 快速装载实例的句柄

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

释放快速装载实例句柄，将该实例句柄上所有快速装载所占用的资源释放。

如果用户调用 fldr_initialize 进行初始化，而没有调用 fldr_uninitialize 通知装载结束，调用此函数将会自动结束该此装载，未使用 fldr_batch 或者 fldr_finish 进行提交的行将会被抛弃。

3. fldr_set_attr

描述：

设置快速装载实例的属性。

格式：

```
FLDR_RETURN
fldr_set_attr(
fhinstance        instance,
fsint4             attrid,
fpointer           value,
fsint4             length
);
```

参数：

instance（输入） 快速装载实例的句柄

attrid(输入) 属性标识，指出将要设置的属性，关于属性详细内容参见注释部分

value (输入) 属性值，根据 attrid 不同该参数的意义不同，其可能是 32 位的整数或者指向一片内存的指针

length（输入） 属性值长度。value 参数为指向一片内存数据的指针时，该参数用来指出该内存数据的长度，如果 value 做为整数传入时，该参数忽略其意义。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

该函数可在调用 fldr_initialize 之前调用，用户可根据本地及服务器的环境特征设置本次快速装载的属性，从而获得较高的效率。

在 fldr_initialize 之后对该函数进行调用将返回执行不成功的错误。

可设置的属性见下表：

属性（attrid）	属性说明（value）	value类型

FLDR_ATTR_SERVER	服务器地址。	指针
FLDR_ATTR_UID	用户名。	指针
FLDR_ATTR_PWD	密码。	指针
FLDR_ATTR_PORT	服务器端口	整数
FLDR_ATTR_TABLE	目标表	指针
FLDR_ATTR_SET_INDENTITY	是否插入自增列（默认FALSE），如果设置为TRUE则服务器将使用指定的数据作为自增列的数据进行插入，如果为FALSE则对于自增列服务器将自动生成数据插入，忽略指定的数据	整数
FLDR_ATTR_DATA_SORTED	数据是否已按照聚集索引排序。默认为FALSE	整数
FLDR_ATTR_INDEX_OPTION	1：不刷新二级索引，数据按照索引先排序，装载完后再将排序的数据插入索引 2：不刷新二级索引，数据装载完成后重建所有二级索引(默认为1)	整数
FLDR_ATTR_DATA_SIZE	指出本地缓冲数据的最大行数。默认为1000，最大为10000，最小为100。	整数
FLDR_ATTR_INSERT_ROWS	8字节整数。只读属性。当前发送到服务器的行数。	整数
FLDR_ATTR_COMMIT_ROWS	8字节整数。只读属性。当前提交到数据库的行数	整数
FLDR_ATTR_PROCESS_ROWS	8字节整数。只读属性。用户绑定的数据行数	整数
FLDR_ATTR_SEND_NODE_NUM	发送链表节点个数，默认128，最大为20480，最小为16。多线程处理中每个线程所支配的向服务器发送数据节点的个数。	整数
FLDR_ATTR_TASK_THREAD_NUM	任务线程数	整数
FLDR_ATTR_BAD_FILE	记录出错数据的文件，出错数据将全部使用字符串表示	指针
FLDR_ATTR_DATA_CHAR_SET	设置输入数据字符类型的字符集	整数
FLDR_ATTR_LOG_FILE	指定日志记录文件	指针
FLDR_ATTR_ERRORS_PERMIT	4字节整数。允许出现错误数据的行数，超过该值装载将报错	整数
FLDR_ATTR_SSL_PATH	在加密通讯模式下加密证书的路径	指针
FLDR_ATTR_SSL_PWD	加密证书的密钥	指针
FLDR_ATTR_LOCAL_CODE	指定本地字符集	整数
FLDR_ATTR_MPP_LOCAL_FLAG	指明是否是mpp模式下的单节点数据。默认值为FALSE	整数
FLDR_ATTR_NULL_MODE	载入NULL字符串时是否作为空值处理，默认值为FALSE	整数
FLDR_ATTR_SERVER_BLDR_NUM	2字节整数。服务器bldr数目，通常多于线程数	整数
FLDR_ATTR_BDTA_SIZE	4字节整数。每次填充数据的行数	整数
FLDR_ATTR_COMPRESS_FLAG	待发送的数据是否先压缩，默认FALSE	整数

FLDR_ATTR_FLUSH_FLAG	指定提交数据时服务器的处理方式。若设置为TRUE则服务器将数据写入磁盘后才返回响应消息；若设置为FALSE则服务器确认数据正确即返回响应消息，之后才将数据写入磁盘	整数

4. fldr_get_attr

描述：

获取快速装载实例的属性。

格式：

```
FLDR_RETURN
fldr_get_attr(
fhinstance      instance,
fsint4          attrid,
fpointer            buffer,
fsint4          buf_sz,
fsint4*             length
);
```

参数：

instance（输入） 快速装载实例的句柄

attrid(输入) 属性标识，指出将要获取的属性，关于属性详细内容参见fldr_set_attr()的注释部分

buffer (输出) 将属性值填入 buffer 所指向的内存中

buf_sz（输入） 属性缓冲区的长度

length（输出） 向 buffer 指向的缓冲区填写的数据的长度。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

参见 fldr_set_attr()函数说明中各属性说明。

5. fldr_initialize

描述：

初始化当前快速装载实例。

格式：

```
FLDR_RETURN
fldr_initialize(
fhinstance      instance,
fint4           type,
fpointer        conn,
fchar*          server,
fchar *         uid,
fchar*          pwd,
fchar*          table
);
```

参数：

instance（输入） 快速装载实例的句柄

type (输入) 可用值为 FLDR_TYPE_BIND：用户使用绑定方式进行装载

conn (输入) 通过 dpi 分配的连接对象，如果该参数不为 NULL 值，则使用用户指定的连接作为导入的连接，忽略 server,uid,pwd 这三个参数

server (输入) 链接的服务器。将被记录到实例的 FLDR_ATTR_SERVER 属性中

uid (输出) 登录用户。将被记录到实例的 FLDR_ATTR_UID 属性中

pwd（输入） 用户密码。将被记录到实例的 FLDR_ATTR_PWD 属性中

table（输入） 进行快速装载的表。将被记录到实例的 FLDR_ATTR_TABLE 属性中

返回值：

FLDR_SUCCESS 、 FLDR_SUCCESS_WITH_INFO 、 FLDR_ERROR 、 FLDR_INVALID_HANDLE

注释：

根据用户通过 fldr_set_attr 设置的属性初始化快速装载环境，如果用户未指定连接则根据用户提供的服务器、用户和密码建立到服务器的连接。

6. fldr_uninitialize

描述：

释放快速装载实例所占用的资源。

格式：

```
FLDR_RETURN
fldr_uninitialize(
fhinstance        instance,
fint4               flag
);
```

参数：

instance（输入） 快速装载实例的句柄

flag (输入) 指出是否对已插入的数据进行事务提交，以永久保存到数据库。取值：FLDR_UNINITILIAZE_COMMIT为是，FLDR_UNINITILIAZE_ROLLBACK为否。如果用户调用了fldr_finish结束导入则该参数将不会起任何作用。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

用于释放所占用快速装载实例所占用的资源。用户在调用该函数后可重新初始化快速装载句柄，进行新的数据装载。

7. fldr_bind

描述：

绑定输入列。

格式：

```
FLDR_RETURN
```

```
fldr_bind(
fhinstance      instance,
fsint2          col_idx,
fsint2          type,
fchar*              date_fmt,
fpointer        data,
fsint4          data_len,
fsint4*         ind_len
);
```

参数：

instance（输入） 快速装载实例的句柄

col_idx (输入) 绑定列的编号。从 1 开始计数，对应快速装载的表的相应列。

type (输入) 用户绑定数据的 C 类型

date_fmt（输入） 当绑定列对应的表的列为时间日期类型，且绑定的 c 数据为 FLDR_C_CHAR 类型，该参数为字符串到日期类型的格式串。其他情况，忽略该绑定输入

data（输入） 指向用于存放数据内存的地址指针， 可在 fldr_sendrows 之前对其进行填充，快速装载接口在用户调用 fldr_sendrows 时读取数据内容

data_len（输入） 用于记录单行数据最大的长度。当进行多行绑定的时候，对于变长的 C 类型，根据该参数内容进行便宜的计算。对于固定长度的 C 数据类型，该参数内容被忽略

ind_len（输入） 用户绑定的数据长度指示符。当为批量绑定时，传入的地址为指向 fsint4 类型的数组。如果地址中保存的值为 FLDR_DATA_NULL，绑定列的该行数据被视为空值。对于 FLDR_C_BINARY 和 FLDR_C_CHAR 变长数据类型，用来指明数据的长度。对于其他定长数据类型，忽略其长度。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

支持绑定的 C 类型见下表：

类型	宏定义	说明
二进制数据	FLDR_C_BINARY	变长
字符数据	FLDR_C_CHAR	变长
1 字节整型数据	FLDR_C_BYTE	定长
2 字节整型数据	FLDR_C_SHORT	定长
4 字节整型数据	FLDR_C_INT	定长
8 字节整型数据	FLDR_C_BIGINT	定长
单精度浮点数	FLDR_C_FLOAT	定长
双精度浮点数	FLDR_C_DOUBLE	定长

8. fldr_bind_nth

描述：

绑定输入列，fldr_bind 的多线程版本。

格式：

```
FLDR_RETURN
```

```
fldr_bind_nth(
    fhinstance      instance,
    fsint2          col_idx,
    fsint2          type,
    fchar*          date_fmt,
    fpointer        str_binds,
    fpointer        data,
    fsint4          col_len,
    fsint4          data_len,
    fsint4*         ind_len,
    fsint4          nth
);
```

参数：

instance（输入） 快速装载实例的句柄

col_idx (输入) 绑定列的编号。从 1 开始计数，对应快速装载的表的相应列。

type (输入) 用户绑定数据的 C 类型

date_fmt（输入） 当绑定列对应的表的列为时间日期类型，且绑定的 c 数据为 FLDR_C_CHAR 类型，该参数为字符串到日期类型的格式串。其他情况，忽略该绑定输入

str_binds(输入） 字符串绑定内存

data（输入） 指向用于存放数据内存的地址指针， 可在 fldr_sendrows 之前对其进行填充，快速装载接口在用户调用 fldr_sendrows 时读取数据内容

col_len（输入） 列定义长度

data_len（输入） 用于记录单行数据最大的长度。当进行多行绑定的时候，对于变长的 C 类型，根据该参数内容进行便宜的计算。对于固定长度的 C 数据类型，该参数内容被忽略

ind_len（输入） 用户绑定的数据长度指示符。当为批量绑定时，传入的地址为指向 fsint4 类型的数组。如果地址中保存的值为 FLDR_DATA_NULL，绑定列的该行数据被视为空值。对于 FLDR_C_BINARY 和 FLDR_C_CHAR 变长数据类型，用来指明数据的长度。对于其他定长数据类型，忽略其长度。

nth（输入） 多线程绑定，第几个线程，从 0 开始。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

支持绑定的 C 类型见下表：

类型	宏定义	说明
二进制数据	FLDR_C_BINARY	变长
字符数据	FLDR_C_CHAR	变长
1 字节整型数据	FLDR_C_BYTE	定长
2 字节整型数据	FLDR_C_SHORT	定长
4 字节整型数据	FLDR_C_INT	定长
8 字节整型数据	FLDR_C_BIGINT	定长
单精度浮点数	FLDR_C_FLOAT	定长
双精度浮点数	FLDR_C_DOUBLE	定长

9. fldr_sendrows

描述：

发送行数。

格式：

```
FLDR_RETURN
fldr_sendrows(
fhinstance        instance,
fint4               rows
);
```

参数：

instance（输入） 快速装载实例的句柄

rows (输入) 在多行绑定的情况，指明绑定的行数。

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

10. fldr_sendrows_nth

描述：

发送行数，fldr_sendrows 的多线程版本。

格式：

```
FLDR_RETURN
fldr_sendrows_nth(
fhinstance        instance,
fsint4              rows,
fsint4              nth,
fsint4              seq_no
);
```

参数：

instance（输入） 快速装载实例的句柄

rows (输入) 在多行绑定的情况，指明绑定的行数。

nth(输入） 多线程版本，第几个线程

seq_no(输入） 数据批次的序号

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

11. fldr_batch

描述：

批量提交行数据，永久保存到服务器。

格式：

```
FLDR_RETURN
fldr_batch(
fhinstance        instance,
fsint8              *rows
```

```
);
```

参数：

instance（输入） 快速装载实例的句柄

rows (输出) 批量提交的行数

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

注释：

将上一次调用 fldr_batch 之后所有 fldr_sendrows 装载的行进行提交，永久保存到数据库中。

12. fldr_finish

描述：

结束一次批量操作。

格式：

```
FLDR_RETURN
fldr_finish(
fhinstance        instance
);
```

参数：

instance（输入） 快速装载实例的句柄

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

13. fldr_get_diag

描述：

获取出错时的诊断信息。

格式：

```
FLDR_RETURN fldr_get_diag(
fhinstance        instance,
fsint4            rec_num,
fint4             err_code,
fchar*            err_msg,
fint4             buf_sz,
fint4             *msg_len
);
```

参数：

instance（输入） 快速装载实例的句柄

rec_num（输入） 诊断信息索引号

err_code（输出） 发生错误的错误码

err_msg（输出） 错误消息缓冲区

buf_sz（输入） 缓冲区大小

msg_len（输出） 错误信息在缓冲区中占用的大小

返回值：

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE、FLDR_NO_DATA

注释：

当用户调用快速装载接口返回 FLDR_ERROR 时，可通过该函数获取出错的错误信息，进行当前状况的判断。该函数出现错误将不会生成任何诊断信息。

14. fldr_exec_ctl_low

描述：

类似命令行工具，通过控制文件，执行数据装载。

格式：

```
FLDR_RETURN
fldr_exec_ctl_low(
fhinstance          instance,
    fchar*              ctl_buffer,
    fsint4              fldr_type,
    fsint8*             row_count
);
```

参数：

instance（输入） 快速装载实例的句柄

ctl_buffer（输入）控制文件 buffer

fldr_type（输入） 数据装载类型

row_count（输出）返回成功装载的行数

返回值：

FLDR_SUCCESS、FLDR_ERROR

注释：

此接口让用户可以通过自定义程序实现数据快速装载。

8.3 编程实例

程序在编译的过程中需要用到 DM 的头文件 fldr.h，在连接阶段需要用到 dmfldr_dll2.lib 这个库文件，在执行阶段需要用到动态库 dmfldr_dll.dll、dmbcast.dll、dmcalc.dll、dmelog.dll、dmmem.dll、dmos.dll、dmutl.dll、dmdta.dll、dmcfg.dll、dmstrt.dll、dmcpr.dll、dmcyt.dll、dmcvt.dll、dmmout.dll、dmcomm.dll、dmdpi.dll、dmclientlex.dll。这几个动态库在安装 DM 时，已经被放到安装目录下。

```
#include "fldr.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main()
{
    fhinstance          instance;
    FLDR_RETURN         ret;
```

```
    char            server[20] = "192.168.0.35";
    char            user[20] = "SYSDBA";
    char            pwd[20] = "SYSDBA";
    char            table[20] = "TEST";
// 提前建好表CREATE TABLE TEST(C1 INT, C2 VARCHAR2(100));
    int             c1[100];
    int             c1_len[100];
    char            c2[100][100];
    int             c2_len[100];
    int             i;

    for (i = 0; i < 100; i++)
    {
        c1[i]   = i;
        sprintf(c2[i], "i = %d", i);
        c2_len[i] = strlen(c2[i]);
    }

    ret = fldr_alloc(&instance);
    fldr_initialize(instance, FLDR_TYPE_BIND, NULL, server, user, pwd, table);
    fldr_bind_nth(instance, 1, FLDR_C_INT, NULL, NULL, c1, 4, 0, c1_len, 0);
    fldr_bind_nth(instance, 2, FLDR_C_CHAR, NULL, NULL, c2, 100, 100*100, c2_len, 0);
    fldr_sendrows_nth(instance, 100, 0, 1);
    fldr_uninitialize(instance, FLDR_UNINITILIAZE_COMMIT);
    fldr_free(instance);
    return 0;
}
```

第 9 章 DM FLDR JNI 编程指南

9.1 DM FLDR JNI 接口介绍

java 调用 dm 快速装载接口导数据是通过 jni 接口来完成的。所引用的对象在 com.dameng.floader.jar 中的 com.dameng.floader.Instance，com.dameng.floader.Properties。主要类是 com.dameng.floader.Instance。

9.2 接口说明

1. allocInstance

描述：

用于分配快速装载对象实例，所有快速装载都将通过该句柄进行操作。

格式：

```
boolean
allocInstance();
```

参数：无

返回值：

true，false

注释：

该函数分配出用于快速装载实例句柄，用户必须首先调用该函数获得快速装载的实例句柄，只有通过该句柄才能进行后续的快速装载操作。

2. free

描述：

释放快速装载实例句柄。

格式：

```
void
free();
```

参数：无

返回值：无

注释：

释放快速装载实例句柄，将该实例句柄上所有快速装载所占用的资源释放。

3. setAttribute

描述：

设置快速装载实例的属性。

格式：

```
bollean
setAttribute(
int attr_id,
```

```
String value
);
```

参数：

attr_id(输入)　　属性标识，指出将要设置的属性，关于属性详细内容参见注释部分。

value (输入)　　属性值，根据 attrid 不同该参数的意义不同。

返回值：

true，false

注释：

可设置的属性见下表：

属性（attrid）	属性说明（value）
FLDR_ATTR_SERVER	服务器地址。
FLDR_ATTR_UID	用户名。
FLDR_ATTR_PWD	密码。
FLDR_ATTR_PORT	服务器端口
FLDR_ATTR_TABLE	目标表
FLDR_ATTR_SET_INDENTITY	是否插入自增列(默认FALSE)，如果设置为TRUE则服务器将使用指定的数据作为自增列的数据进行插入，如果为FALSE则对于自增列服务器将自动生成数据插入，忽略指定的数据
FLDR_ATTR_DATA_SORTED	数据是否已按照聚集索引排序。默认为FALSE
FLDR_ATTR_INDEX_OPTION	1：不刷新二级索引，数据按照索引先排序，装载完后再将排序的数据插入索引 2：不刷新二级索引，数据装载完成后重建所有二级索引(默认为1)
FLDR_ATTR_DATA_SIZE	指出本地缓冲数据的最大行数。默认为1000，最大为10000，最小为100。
FLDR_ATTR_INSERT_ROWS	8字节整数。只读属性。当前发送到服务器的行数。
FLDR_ATTR_COMMIT_ROWS	8字节整数。只读属性。当前提交到数据库的行数
FLDR_ATTR_PROCESS_ROWS	8字节整数。只读属性。用户绑定的数据行数
FLDR_ATTR_SEND_NODE_NUM	发送链表节点个数，默认128，最大为20480，最小为16。多线程处理中每个线程所支配的向服务器发送数据节点的个数。
FLDR_ATTR_TASK_THREAD_NUM	任务线程数
FLDR_ATTR_BAD_FILE	记录出错数据的文件，出错数据将全部使用字符串表示
FLDR_ATTR_DATA_CHAR_SET	设置输入数据字符类型的字符集
FLDR_ATTR_LOG_FILE	指定日志记录文件
FLDR_ATTR_ERRORS_PERMIT	4字节整数。允许出现错误数据的行数，超过该值装载将报错
FLDR_ATTR_SSL_PATH	在加密通讯模式下加密证书的路径
FLDR_ATTR_SSL_PWD	加密证书的密钥

FLDR_ATTR_LOCAL_CODE	指定本地字符集
FLDR_ATTR_MPP_LOCAL_FLAG	指明是否是mpp模式下的单节点数据。默认值为FALSE
FLDR_ATTR_NULL_MODE	载入NULL字符串时是否作为空值处理，默认值为FALSE
FLDR_ATTR_SERVER_BLDR_NUM	2字节整数。服务器bldr数目，通常多于线程数
FLDR_ATTR_BDTA_SIZE	4字节整数。每次填充数据的行数
FLDR_ATTR_COMPRESS_FLAG	待发送的数据是否先压缩，默认FALSE
FLDR_ATTR_FLUSH_FLAG	指定提交数据时服务器的处理方式。若设置为TRUE则服务器将数据写入磁盘后才返回响应消息；若设置为FALSE则服务器确认数据正确即返回响应消息，之后才将数据写入磁盘

4. getAttribute

描述：

获取快速装载实例的属性。

格式：

```
String
getAttribute(
    int attr_id
);
```

参数：

attrid(输入)　　属性标识，指出将要获取的属性，关于属性详细内容参见 setAttribute 的注释部分。

返回值：

属性值。

注释：

参见 setAttribute 函数说明中各属性说明。

5. initializeInstance

描述：

初始化当前快速装载实例。

格式：

```
boolean
initializeInstance(
    String      server,
    String      user,
    String      pwd,
    String      table
);
```

参数：

server (输入)　　链接的服务器。将被记录到实例的 FLDR_ATTR_SERVER 属性中。

user (输入)　　登录用户。将被记录到实例的 FLDR_ATTR_UID 属性中。

pwd（输入）　　用户密码。将被记录到实例的 FLDR_ATTR_PWD 属性中。

table（输出）　　进行快速装载的表。将被记录到实例的 FLDR_ATTR_TABLE 属性中。

返回值：

true，false

注释：

根据用户通过 setAttribute 设置的属性初始化快速装载环境，如果用户未指定连接则根据用户提供的服务器、用户和密码建立到服务器的连接。

6. unitialize

描述：

释放快速装载实例所占用的资源。

格式：

```
boolean
uninitialize();
```

参数：无

返回值：

true，false

注释：

用于释放所占用快速装载实例所占用的资源。用户在调用该函数后可重新初始化快速装载句柄，进行新的数据装载。

7. fldr_Bind_Columns_ex

描述：

绑定输入列。

格式：

```
boolean
fldr_Bind_Columns_ex(
int colNumber,
int type,
String datefmt,
Object data,
int maxcollen,
int[] ind_len,
int threadNum
);
```

参数：

colNumber(输入)　　绑定列的编号。从 1 开始计数，对应快速装载的表的相应列。

type (输入)　　用户绑定数据的 JAVA 类型。

datefmt（输入）　　当绑定列对应的表的列为时间日期类型，且绑定的 JAVA 数据为 String 类型，该参数为字符串到日期类型的格式串。其他情况，忽略该绑定输入。

data（输入）　　指向用于存放数据内存的地址指针。

maxcollen（输入）　　用于记录单行数据最大的长度。当进行多行绑定的时候，对于变长的 C 类型，根据该参数内容进行便宜的计算。对于固定长度的 JAVA 数据类型，

该参数内容被忽略。

ind_len（输入） 用户绑定的数据长度指示符。当为批量绑定时，传入的地址为指向 INT 类型的数组。如果地址中保存的值为-1，绑定列的该行数据被视为空值。对于 BINARY 和 CHAR 变长数据类型，用来指明数据的长度。对于其他定长数据类型，忽略其长度。

threadNum(输入) 多线程调用，线程序号，单线程传 0。

返回值：

true，false

注释：

8. fldr_Send_Rows_ex

描述：

发送行数。

格式：

```
boolean
fldr_Send_Rows_ex(
long rows,
int threadNum,
int seqNo
);
```

参数：

rows (输入) 在多行绑定的情况，指明绑定的行数。

threadNum(输入) 多线程调用，线程序号，单线程传 0。

seqNo(输入) 数据批次，从 1 开始自增，保证数据有序。

返回值：

true，false

9. finish

描述：

结束一次批量操作。

格式：

```
bollean
finish();
```

参数：无

返回值：

true，false

9.3 DM FLDR JNI 编程实例

```
package com.dameng;
import java.io.File;
import com.dameng.floader.DataTypes;
import com.dameng.floader.Instance;
import com.dameng.floader.Properties;
```

```
public class TestFloder
{

    public static void main(String[] args)
    {
        Instance instance = new Instance();

        int threadNum = 1;

        TestFloder floder = new TestFloder();
        /* ****************************初始化******************************** */
        /* create table sysdba.test1(c1 int ,c2 varchar(50)); */
        String tableName = "SYSDBA.TEST1";
        boolean success = floder.init(instance, threadNum, tableName);

        if (success)
        {
            try
            {
                int         c1[] = {1,2,3,4,5,6,7,8,9};          //数据
                int         c1_ind[] = {4,4,4,4,4,4,4,4,4};      //长度指示符   -1表示空
                byte[]      byteArray = new byte[9];

                for (int i = 0; i < 9; i++)
                {
                    byteArray[i] = ((Integer)c1[i]).byteValue();
                }


                String      c2[] =
{"a1","a12","a123","a1234","a12345","a123456","a1234567","a12345678","a123456789"};
                int         c2_ind[] = {2,3,4,5,6,7,8,9,10};    //长度指示符   -1表示空
                byte[][]    byteArray2 = new byte[9][];

                for (int i = 0; i < 9; i++)
                {
                    byteArray2[i] = c2[i].getBytes();
                }

                /*
                fldr_Bind_Columns_ex(
                int colNumber,                  // 列号 从1开始
                int type,                       // 列类型从DataTypes取
```

```
                String datefmt,         // 时间类型格式范式：默认yyyy-mm-dd hh-mm-ss，为空的话，按默
认处理
                Object data,                    // 数值类型直接使数值类型数组，字符串类型是byte数组
                int maxcollen,                  // 数值类型值无效，fldr里面自己按sizeof取长度。，
                                                    // xxxxxxxxxxxxx注意字符串类型是所有数据总长度
xxxxxxxxxxxxxxxxxxx。
                                                    // fldr里面要按这个总长度申请类型
                int[] ind_len,                  // 长度指示符，int数组，-1标示空
                                                    // 数值类型无效，字符串类型标示数据长度。
                int threadNum                   // 多线程下有效，单线程传1
                )
                 */

                /*
                fldr_Send_Rows_ex(
                long rows,                      // 行数
                int threadNum,                  // 多线程下有效，单线程传1
                int seqNo                       // 数据批次号，从1开始，必须连续，否则fldr会卡住。
                )
                */

                for (int i = 1; i < 10; i++)
                {
                      instance.fldr_Bind_Columns_ex(1, DataTypes.FLDR_C_INT, null, c1, 4, c1_ind, 1);
                        instance.fldr_Bind_Columns_ex(2, DataTypes.FLDR_C_CHAR, null, byteArray2, 54,
c2_ind, 1);

                        instance.fldr_Send_Rows_ex(9, 1, i);
                }


            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }


        /* ****************************释放******************************** */
        if (!success)
        {
            String message = instance.getErrorMsg();
            System.out.println("出错信息:" + message);
```

```
                instance.free();
            }
            else
            {
                instance.finish();
                // 返回实际插入的行数
                long rowHaveCopied =
Long.parseLong(instance.getAttribute(Properties.FLDR_ATTR_COMMIT_ROWS));
                System.out.println("已复制行数:" + rowHaveCopied);
                instance.unitialize();
                instance.free();
            }

        }

        // 初始化
        public boolean init(Instance instance, int threadNum, String tableName)
        {
            // 分配句柄
            instance.allocInstance();
            // 设置属性
            String host = "localhost";
            String port = "5236";
            String userName = "SYSDBA";
            String password = "SYSDBA";
            instance.setAttribute(Properties.FLDR_ATTR_SERVER, host);
            instance.setAttribute(Properties.FLDR_ATTR_PORT, port);
            instance.setAttribute(Properties.FLDR_ATTR_UID, userName);
            instance.setAttribute(Properties.FLDR_ATTR_PWD, password);

            // 编码
//              instance.setAttribute(Properties.FLDR_ATTR_DATA_CHAR_SET,
System.getProperty("file.encoding"));

            // 是否有自增量
//              instance.setAttribute(Properties.FLDR_ATTR_SET_INDENTITY, "0");

            // fldr task线程默认为cpu个数，如果客户端线程数大于cpu个数，则必须设置该参数
//              instance.setAttribute(Properties.FLDR_ATTR_TASK_THREAD_NUM, String.valueOf(threadNum));

            // 一次提交记录数
//              instance.setAttribute(Properties.FLDR_ATTR_DATA_SIZE, "5000");

            // 记录出错的信息
```

```
        String dmHome = System.getProperty("DM_HOME");
        if (dmHome != null && !dmHome.equals(""))
        {
            if (dmHome.endsWith("/") || dmHome.endsWith("\\"))
            {
                instance.setAttribute(Properties.FLDR_ATTR_BAD_FILE, dmHome + "BADFILE.TXT");
            }
            else
            {
                instance.setAttribute(Properties.FLDR_ATTR_BAD_FILE, dmHome + File.separator +
"BADFILE.TXT");
            }
        }

        // 装载日志
        if (dmHome != null && !dmHome.equals(""))
        {
            if (dmHome.endsWith("/") || dmHome.endsWith("\\"))
            {
                instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome + "FLDR_LOG.TXT");
            }
            else
            {
                instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome + File.separator +
"FLDR_LOG.TXT");
            }
        }

        // 最大容错个数
//          instance.setAttribute(Properties.FLDR_ATTR_ERRORS_PERMIT,
String.valueOf(Integer.MAX_VALUE));

        // 是否为mpp
//          instance.setAttribute(Properties.FLDR_ATTR_MPP_CLIENT, "0");

        // 如果是txt到dm8
//              instance.setAttribute(Properties.FLDR_ATTR_COMPRESS_FLAG,   "0");
//              instance.setAttribute(Properties.FLDR_ATTR_DIRECT_MODE,"0");
//              instance.setAttribute(Properties.FLDR_ATTR_DATA_SORTED, "0");
//              instance.setAttribute(Properties.FLDR_ATTR_LOCAL_CODE, "GBK");
//              instance.setAttribute(Properties.FLDR_ATTR_BLOB_TYPE, "0");
//              instance.setAttribute(Properties.FLDR_ATTR_INDEX_OPTION,"1");
//              instance.setAttribute(Properties.FLDR_ATTR_LOB_DIRECTORY,"");
//              instance.setAttribute(Properties.FLDR_ATTR_ERRORS_PERMIT,"1000");
```

```
//                  instance.setAttribute(Properties.FLDR_ATTR_SKIP_ROWS,"0");
//                  instance.setAttribute(Properties.FLDR_ATTR_BUFFER_SIZE,"512");
//                  instance.setAttribute(Properties.FLDR_ATTR_SERVER_BLDR_NUM,"512");
//                  instance.setAttribute(Properties.FLDR_ATTR_READ_ROWS,"512");
//                  instance.setAttribute(Properties.FLDR_ATTR_ROWS_COMMIT,"512");
//                  instance.setAttribute(Properties.FLDR_ATTR_BDTA_SIZE,"512");

            boolean success = instance.initializeInstance(null, null, null, tableName);
            return success;
        }

}
```

第 10 章 Logmnr 接口使用说明

Logmnr 包是达梦数据库的日志分析工具，达梦提供了 JNI 接口和 C 接口，供应用程序直接调用。

10.1 JNI 接口

logmnr.jar 包是达梦数据库的日志分析工具的 JNI 接口，供应用程序直接调用。
一般调用过程如下：

1. 初始化 LOGMNR 环境；
2. 创建一个分析日志的连接；
3. 添加需要分析的日志文件；
4. 启动日志文件分析；
5. 获取完成分析的日志数据；
6. 终止日志文件分析；
7. 关闭当前连接；
8. 销毁 LOGMNR 环境。

10.1.1 接口说明

logmnr.jar 包括两个类：LogmnrDll 和 LogmnrRecord。logmnr.jar 位于 DM 安装目录的 jar 文件夹下，例如：D:\dmdbms\jar\logmnr.jar。LogmnrDll 中包含了所有功能接口；LogmnrRecord 是日志分析结果数据的返回值对应的对象类型。

10.1.1.1 初始化环境

函数原型

```
int LogmnrDll.initLogmnr();
```

功能说明

初始化 LOGMNR 环境。

参数说明

无。

返回值

0：初始化成功；
< 0 的值和异常：初始化失败。

10.1.1.2 创建连接

函数原型

```
long LogmnrDll.createConnect(String hostName, int port, String userName, String password);
```

功能说明

创建一个分析日志的连接。

参数说明

hostName：ip 或主库名。

port：端口号。

userName：用户名。

password：密码。

返回值

创建的数据库连接标识 ID。

10.1.1.3 添加日志文件

函数原型

```
int LogmnrDll.addLogFile(long  connId, String logFileName,  int option);
```

功能说明

添加需要分析的归档日志文件。

参数说明

connId：数据库连接标识 ID。

logFileName：需要分析的归档日志文件名（绝对路径）。

option：可选配置参数。包括以下值：

1：结束当前日志挖掘（隐式调用 endLogmnr，以前添加的归档日志文件将被清除），并增加新的归档日志文件。

2：删除日志文件。

3：增加的归档日志文件名。

返回值

0：添加成功；

< 0 的值和异常：添加失败。

注意：

如果当前已经 startLogmnr，则 addLogFile 将报错，如果需要添加新的日志文件，需要先调用 endLogmnr 结束当前日志分析后再添加文件。

10.1.1.4 移除日志文件

函数原型

```
int LogmnrDll.removeLogFile(long  connId, String logFileName);
```

功能说明

移除指定的归档日志文件。

参数说明

connId：数据库连接标识 ID。

logFileName：需要移除档日志文件名（绝对路径）。

返回值

0：移除成功；

< 0 的值和异常：移除失败。

注意：

同 addLogFile 一样，如果当前已经 startLogmnr，则 removeLogFile 将报错，如果需要移除日志文件，需要先调用 endLogmnr 结束当前日志分析后再移除文件。

10.1.1.5 启动日志分析

函数原型

```
int LogmnrDll.startLogmnr(long connId, long trxid, String startTime, String endTime);
```

功能说明

启动当前会话的归档日志文件分析，对添加的归档日志文件进行日志分析。

参数说明

connId：数据库连接标识 ID。

trxid：分析归档日志的事务 id 号，默认为-1，表示不区分事务号。

startTime：分析归档日志的起始时间，默认 1988/1/1。

endTime：分析归档日志的结束时间，默认 2110/12/31。

返回值

0：启动成功；

< 0 的值和异常：启动失败。

10.1.1.6 获取数据

函数原型

```
LogmnrRecord[]LogmnrDll.getData(long connId, int rownum);
```

功能说明

获取日志文件分析结果数据。

参数说明

connId：数据库连接标识 ID。

rownum：获取行数。

返回值

获取的 logmnr 记录，为 LogmnrRecord 对象数组，详见 10.1.1.6.1 LogmnrRecord 对象说明。

10.1.1.6.1 LogmnrRecord 对象说明

LogmnrRecord 对象支持的成员变量入下表所示。获取和设置这些变量可以使用相应的“get+成员变量”和“set+成员变量”方法。

成员变量	类型	说明
scn	long	当前记录的 LSN
startScn	long	当前事务的起始 LSN
commitScn	long	当前事务的截止 LSN
timestamp	String	当前记录的创建时间
startTimestamp	String	当前事务的起始时间
commitTimestamp	String	当前事务的截止时间
xid	String	当前记录的事务 ID 号
operation	String	操作类型包括 start、insert、update、delete、commit、rollback 等语句
operationCode	String	操作类型。插入操作值为 1，更新操作值为 3，删除操作值为 2，事务起始语句值为 6，提交操作值为 7，回滚操作值为 36
rollBack	int	当前记录是否被回滚（1 是 0 否）
segOwner	String	执行这条语句的用户名
tableName	String	操作的表名
rowId	String	对应记录的行号
rbasqn	int	对应的归档日志文件号
rbablk	int	RBASQN 所指日志文件的块号从 0 开始
rbabyte	int	RBABLK 所指块号的块内偏移
dataObj	int	对象 ID 号
dataObjv	int	对象版本号
sqlRedo	String	当前记录对应的 sql 语句
rsId	Stirng	记录集 ID

ssn	int	连续 sql 标志。如果 sql 长度超过单个 sql_redo 字段能存储的长度，则 sql 会被截断成多个 sql 片段在结果集中“连续”返回
csf	int	与 SSN 配合。最后一个片段的 csf 值为 0，其余片段的值均为 1。没因超长发生截断的 sql 该字段值均为 0
status	int	日志状态。默认为 0

10.1.1.7 终止日志分析

函数原型

```
int LogmnrDll.endLogmnr(long connId, int option);
```

功能说明

终止当前会话的归档日志文件分析。

参数说明

connId：数据库连接标识 ID。

options：可选模式如下：

0：清除当前会话的归档日志文件分析环境，再次分析时需要重新 add_logfile。

1：保留当前会话的归档日志文件分析环境，不需要重新 add_logfile。

返回值

0：终止成功；

< 0 的值和异常：终止失败。

10.1.1.8 关闭连接

函数原型

```
int LogmnrDll.closeConnect(long connId);
```

功能说明

关闭当前连接，清理字典缓存等。

参数说明

connId：数据库连接标识 ID。

返回值

0：关闭成功；

< 0 的值和异常：关闭失败。

10.1.1.9 销毁环境

函数原型

```
boolean LogmnrDll.deinitLogmnr();
```

功能说明

销毁 LOGMNR 环境。

参数说明

无。

返回值

0：添加销毁；

< 0 的值和异常：销毁失败。

10.1.1.10 设置属性

函数原型

```
int LogmnrDll.setAttr(long connId, int attr, int attr_value)
```

功能说明

设置属性。

有如下三个属性可以配置：

LogmnrDll.LOGMNR_ATTR_PARALLEL_NUM 并行线程数 有效值(2, 16)；

LogmnrDll.LOGMNR_ATTR_BUFFER_NUM 任务缓存节点数 有效值(8, 1024)；

LogmnrDll.LOGMNR_ATTR_CONTENT_NUM 结果缓存节点数 有效值(256, 2048)。

参数说明

connId：数据库连接标识 ID。

attr：属性名。

attr_value：属性值。

返回值

0：添加成功；

< 0 的值和异常：添加失败。

10.1.2 编程实例

下面用一个例子来说明 dmlogmnr 接口的使用方法。

数据库安装在本机上，端口号为 5236，用户名和密码均为 SYSDBA。要分析的日志为 D:\dmdata\arch\ARCHIVE_LOCAL1_20160704082303068.log 。分析的结果放在 D:\dmdata\result.txt 中。其中 logmnr.jar 包放在 DM 数据库安装目录的 jar 文件夹下。

在项目中创建类名为 dbmslob，代码实现如下：

```
import java.io.PrintStream;
import com.dameng.logmnr.LogmnrDll;
import com.dameng.logmnr.LogmnrRecord;
public class dbmslob {
    public static void main(String[] args) {
    try
```

```
        {
            LogmnrDll.initLogmnr();
            long connid = LogmnrDll.createConnect("localhost", 5236, "SYSDBA",    "SYSDBA");
    LogmnrDll.addLogFile(connid,"D:\\dmdata\\arch\\ARCHIVE_LOCAL1_20160704082303068.log",3);
            LogmnrDll.startLogmnr(connid, -1, null, null);
            LogmnrRecord[] arr = LogmnrDll.getData(connid, 100);
            PrintStream ps = new PrintStream("D:\\dmdata\\result.txt");
             System.setOut(ps);
             System.out.println("日志分析结果打印：");
             for (int i =   0; i < arr.length; i++) {
                 System.out.println("----------------------------" + i + "----------------------------" + "\n");
                 System.out.println("xid:" + arr[i].getXid() + "\n");
                 System.out.println("operation:" + arr[i].getOperation() + "\n");
                 System.out.println("sqlRedo:" + arr[i].getSqlRedo()  + "\n");
                 System.out.println("##############################" + i +
"##############################" + "\n");
                 System.out.println("scn:" + arr[i].getScn()  + "\n");
                 System.out.println("startScn:" + arr[i].getStartScn()  + "\n");
                 System.out.println("commitScn:" + arr[i].getCommitScn()  + "\n");
                 System.out.println("timestamp:" + arr[i].getTimestamp() + "\n");
                 System.out.println("startTimestamp:" + arr[i].getStartTimestamp() + "\n");
                 System.out.println("commitTimestamp:" + arr[i].getCommitTimestamp() + "\n");
                 System.out.println("operationCode:" + arr[i].getOperationCode() + "\n");
                 System.out.println("rollBack:" + arr[i].getRollBack()  + "\n");
                 System.out.println("segOwner:" + arr[i].getSegOwner()  + "\n");
                 System.out.println("tableName:" + arr[i].getTableName()  + "\n");
                 System.out.println("rowId:" + arr[i].getRowId() + "\n");
                 System.out.println("rbasqn:" + arr[i].getRbasqn() + "\n");
                 System.out.println("rbablk:" + arr[i].getRbablk()  + "\n");
                 System.out.println("rbabyte:" + arr[i].getRbabyte() + "\n");
                 System.out.println("dataObj:" + arr[i].getDataObj() + "\n");
                 System.out.println("dataObjv:" + arr[i].getDataObjv() + "\n");
                 System.out.println("//dataObjd:" + arr[i].getDataObjd() + "\n");
                 System.out.println("rsId:" + arr[i].getRsId() + "\n");
                 System.out.println("ssn:" + arr[i].getSsn() + "\n");
                 System.out.println("csf:" + arr[i].getCsf() + "\n");
                 System.out.println("status:" + arr[i].getStatus() + "\n");
                 System.out.println("##############################" + i +
"##############################" + "\n");
              }
            System.out.println("结果打印完毕");
            ps.flush();
             ps.close();
            LogmnrDll.endLogmnr(connid,  1);
```

```
            LogmnrDll.deinitLogmnr();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

分析的结果在 D:\dmdata\result.txt 中查看。

10.2 C 接口

dmlogmnr.dll 是达梦数据库的日志分析工具的 C 接口，供应用程序直接调用。应用程序员在调用接口时应注意接口参数为句柄指针时，应传入正确的结构指针，否则可能造成异常。

10.2.1 接口说明

10.2.1.1 初始化环境

函数原型

```
dmcode_t
logmnr_client_init();
```

参数说明

无。

功能说明

初始化 LOGMNR 环境。说明：初始化互斥量等。

返回值

0：成功；
<0：错误。

10.2.1.2 创建连接

函数原型

```
dmcode_t
logmnr_client_create_connect(
    schar*          ip,
    usint           port,
    schar*          uname,
    schar*          pwd,
    void**          conn
);
```

功能说明

创建一个分析日志的连接。

参数说明

ip：输入参数，服务器 IP；
port：输入参数，端口号；
uname：输入参数，用户名；
pwd：输入参数，登录密码；
conn：输出参数，连接句柄。

返回值

0：成功；
<0：错误。

10.2.1.3 增加日志文件

函数原型

```
dmcode_t
logmnr_client_add_logfile(
    void*       conn,
    schar*      logfilename,
    ulint       options
);
```

功能说明

增加需要分析的归档日志文件。

参数说明

conn：连接句柄；
logfilename：需要分析的归档日志文件名（绝对路径）；
options：
可选配置参数包括：
1：结束当前日志挖掘 （隐式调用 endLogmnr，以前添加的归档日志文件将被清除），并增加新的归档日志文件。
2：删除日志文件。
3：增加的归档日志文件名。
说明：一旦 startLogmnr，addLogFile 可以继续添加文件，那么新添加的文件必须比之前添加的所有文件的创建时间都要晚。

返回值

0：成功；
<0：错误。

10.2.1.4 删除日志文件

函数原型

```
dmcode_t
logmnr_client_remove_logfile(
    void*       conn,
    schar*          logfilename
);
```

功能说明

移除某个需要分析的归档日志文件。

参数说明

conn：连接句柄；

logfilename：待移除的归档日志文件名（绝对路径）。

说明：只有 startLogmnr 之前才能进行 remove 操作，否则会报错返回。

返回值

0：成功

<0：错误

10.2.1.5 启动日志分析

函数原型

```
dmcode_t
logmnr_client_start(
    void*       conn,
    lint64          trxid,
    schar*          starttime,
    schar*          endtime
);
```

功能说明

启动当前会话的归档日志文件分析，对 ADD_LOGFILE 添加的归档日志文件进行日志分析。

参数说明

conn：连接句柄。

trxid：分析归档日志的事务 id 号，默认为-1,表示不区分事务号。

starttime：分析归档日志的起始时间，默认 1988/1/1。

endtime：分析归档日志的结束时间，默认 2110/12/31。

返回值

0：成功；

<0：错误。

10.2.1.6 获取信息

函数原型

```
dmcode_t
logmnr_client_get_data(
    void*                   conn,
    lint                    row_num,
    logmnr_content_t***     data,
    lint*                   real_num
);
```

功能说明

获取数据

参数说明

conn：输入参数，连接句柄；

rownum：输入参数，获取行数；

data：输出参数，返回数据；

real_num：输出参数，实际获取行数。

返回值

获取的 logmnr 记录。

10.2.1.7 终止日志分析

函数原型

```
dmcode_t
logmnr_client_end(
    void*       conn,
    ulint       options
);
```

功能说明

终止当前会话的归档日志文件分析。

参数说明

conn：连接句柄；

options：

可选模式如下：

0：清除当前会话的归档日志文件分析环境，再次分析时需要重新 add_logfile。

1：保留当前会话的归档日志文件分析环境，不需要重新 add_logfile

返回值

0：成功；

<0：错误。

10.2.1.8 结束连接

函数原型

```
dmcode_t
logmnr_client_close_connect(
    void*       conn
);
```

参数说明

conn：连接句柄。

功能说明

关闭当前连接。说明：会清理字典缓存等。

返回值

0：成功；
<0：错误。

10.2.1.9 销毁环境

函数原型

```
dmcode_t
logmnr_client_deinit();
```

功能说明

销毁 LOGMNR 环境。

参数说明

无。

返回值

0：成功；
<0：错误。

10.2.1.10 设置日志分析属性

函数原型

```
dmcode_t
logmnr_client_set_attr(
    void*               conn,
    ulint               attr,
    void*               val,
    ulint               val_len
);
```

参数说明

conn：输入参数，连接句柄；
attr：输入参数，属性名称；
val：输入参数，属性 attr 的值；
val_len：输入参数，属性值 val 的长度。

功能说明

设置日志分析属性。
有如下三个属性可以设置：
LOGMNR_ATTR_PARALLEL_NUM 并行线程数 有效值(2, 16)；
LOGMNR_ATTR_BUFFER_NUM 任务缓存节点数 有效值(8, 1024)；
LOGMNR_ATTR_CONTENT_NUM 结果缓存节点数 有效值(256, 2048)。

返回值

0：成功；
<0：错误。

10.2.1.11 获取日志分析属性

函数原型

```
dmcode_t
logmnr_client_get_attr(
    void*           conn,
    ulint           attr,
    void*           buf,
      ulint                 buf_len,
    ulint*          val_len
);
```

参数说明

conn：输入参数，连接句柄；
attr：输入参数，属性名称；
buf：输出参数，属性 attr 的值；
buf_len：输入参数，属性值缓存 buf 的长度；
val_len：输出参数，属性值的实际长度(<=buf_len)。

功能说明

获取日志分析属性。
有如下四个属性可以获取：
LOGMNR_ATTR_PARALLEL_NUM 并行线程数 有效值(2, 16)；
LOGMNR_ATTR_BUFFER_NUM 任务缓存节点数 有效值(8, 1024)；
LOGMNR_ATTR_CONTENT_NUM 结果缓存节点数 有效值(256, 2048)；
LOGMNR_ATTR_CHAR_CODE 本地编码（只读）。

返回值

0：成功；

<0：错误。

10.2.2 编程实例

```
#include "logmnr_client.h"
#include "logmnr_pub.h"
#include "utl.h"

void
main(int argc, schar* argv[])
{
	lint			rt;
	void*			conn;
	logmnr_content_t**	logmnr_contents;
	lint			real_num;
	lint64			b_t, time_used;
	ulint			char_code;

	rt = logmnr_client_init();
	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_init error");
		exit(-1);
	}

	rt = logmnr_client_create_connect("LOCALHOST", 5236, "SYSDBA", "SYSDBA", &conn);
	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_create_connect error");
		exit(-1);
	}

	//需要添加的文件名
	rt = logmnr_client_add_logfile(conn, "d:\\dmdata\\arch\\ARCHIVE_LOCAL1_20140725091059625.log",
LOGMNR_ADDFILE);

	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_add_logfile error");
		exit(-1);
	}

	rt = logmnr_client_set_attr(conn, LOGMNR_ATTR_PARALLEL_NUM, (void*)2, 0);
	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_set_attr error");
		exit(-1);
	}

	b_t = dm_get_usec_tick_count();

	rt = logmnr_client_start(conn, -1, NULL, NULL);
	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_start error");
		exit(-1);
	}

	rt = logmnr_client_get_attr(conn, LOGMNR_ATTR_CHAR_CODE, &char_code, 4, 0);
	if (rt < 0)
	{
		fprintf(stderr, "logmnr_client_set_attr error");
```

```
            exit(-1);
        }

        while(TRUE)
        {
            rt = logmnr_client_get_data(conn, 6, &logmnr_contents, &real_num);
            if (rt < 0)
            {
                fprintf(stderr, "logmnr_client_get_data error");
                exit(-1);
            }
            if (real_num == 0)
                break;
        }

        time_used = dm_get_usec_tick_count() - b_t;
        fprintf(stderr, "已用时间：%ld秒", time_used / 1000/ 1000);

        rt = logmnr_client_end(conn, 0);
        if (rt < 0)
        {
            fprintf(stderr, "logmnr_client_end error");
            exit(-1);
        }

        rt = logmnr_client_close_connect(conn);
        if (rt < 0)
        {
            fprintf(stderr, "logmnr_client_close_connect error");
            exit(-1);
        }

        rt = logmnr_client_deinit();
        if (rt < 0)
        {
            fprintf(stderr, "logmnr_client_deinit error");
            exit(-1);
        }
}
```

附录 1 错误码汇编

1 DM 服务器错误码汇编

服务器错误码值域如下：

1） 警告信息 错误码值域为：（520,0）
2） 普通错误 错误码值域为：（-1,-100）
3） 启动错误 错误码值域为：（-101,-200）
4） 系统错误 错误码值域为：（-501,-800）
5） 服务器配置参数 错误码值域为：（-801,-2000）
6） 分析阶段错误 错误码值域为：（-2001,-5500）
7） 权限错误 错误码值域为：（-5501,-6000）
8） 运行时错误 错误码值域为：（-6001,-8000）
9） 备份恢复错误 错误码值域为：（-8001,-8400）
10） 作业管理错误 错误码值域为：（-8401,-8700）
11） 数据复制错误 错误码值域为：（-8701,-9000）
12） 其它 错误码值域为：（-9001,-10000）

详细的错误码信息请参考 V$ERR_INFO。

2 DPI 错误码汇编

DPI 错误码值域如下：

1） 服务器警告 错误码值域为：（100,119）
2） DPI 警告 错误码值域为：（70000,70013）
3） DPI 错误 错误码值域为：（-70000, -70078）

表 DPI 错误码

名称	代码	解释
DPI_ERROR	-70000	一般错误
EC_DPI_INVALID_DATA_BUF	-70001	无效的数据缓冲区
EC_DPI_INVALID_DATA_LEN	-70002	无效的数据长度
EC_DPI_INVALID_DTYPE	-70003	无效的数据库数据类型
EC_DPI_INVALID_CTYPE	-70004	无效的 C 数据类型
EC_DPI_STR_TRUNC	-70005	字符串截断
EC_DPI_INVALID_NUMERIC_PREC	-70006	无效的精度
EC_DPI_INVALID_NUMERIC_SCALE	-70007	无效的刻度
EC_DPI_DATATYPE_CANNOT_CNVT	-70008	数据类型转换失败

EC_DPI_UTF_STR_NOT_INTEGRATED	-70009	不完整的 UTF 字符串
EC_DPI_INVALID_INTERVAL_TYPE	-70010	无效的 INTERVAL 类型
EC_DPI_INVALID_CNVT_STR	-70011	无效的转换字符串
EC_DPI_DATA_OUT_OF_RANGE	-70012	数据超出范围
EC_DPI_CNVT_FAIL	-70013	转换失败
EC_DPI_ILLEGAL_CHAR	-70014	非法的字符
EC_DPI_INVALID_DATE_STR	-70015	无效的日期字符串
EC_DPI_DATE_OVERFLOW	-70016	日期超出范围
EC_DPI_MEMORY_NOT_ENOUGH	-70017	内存分配出错
EC_DPI_BUF_NOT_ENOUGH	-70018	缓冲区不足
EC_DPI_NET_COMMUNITE_FAIL	-70019	网络通讯失败
EC_DPI_CRC_CHECK_FAIL	-70020	CRC 校验失败
EC_DPI_INVALID_OP	-70021	无效的操作
EC_DPI_UNKNOWN_TRANSSTATE	-70022	未知的事务状态
EC_DPI_INVALID_PARAM_VALUE	-70023	无效的参数值
EC_DPI_CONNECTION_IS_OPENED	-70024	连接已打开
EC_DPI_CONNECTION_NOT_OPENED	-70025	连接未打开
EC_DPI_INVALID_FUN_SEQ	-70026	非法的函数调用序列
EC_DPI_INVALID_CURSOR_STATUS	-70027	无效的游标状态
EC_DPI_CREATE_SOCKET_CONN_FAIL	-70028	创建 SOCKET 连接失败
EC_DPI_CREATE_ENCRYPT_KEY_FAIL	-70029	创建加密数据失败
EC_DPI_SVR_VERSION_WRONG	-70030	不支持的服务器版本
EC_DPI_ENCRYPT_SYS_INIT_FAIL	-70031	加密组件初始化失败
EC_DPI_INVALID_INDICATE_PTR	-70032	无效的指示符指针
EC_DPI_INVALID_COL_INDEX	-70033	无效的列索引
EC_DPI_PUT_DATA_ALREADY	-70034	已经发送数据
EC_DPI_INVALID_NULL_POINTER	-70035	无效的空指针
EC_DPI_INVALID_HANDLE_STATUS	-70036	无效的句柄状态
EC_DPI_STR_NOT_INTEGRATED	-70037	字符串不完整
EC_DPI_INVALID_PARAM_TYPE	-70038	无效的参数绑定类型
EC_DPI_COUNT_PARAM_INCORRECT	-70039	绑定的参数个数不正确
EC_DPI_BIND_TYPE_MISMATCH	-70040	绑定的参数类型不正确
EC_DPI_INVALID_ATTR	-70041	无效的属性
EC_DPI_ATTR_READONLY	-70042	只读属性
EC_DPI_INVALID_DESC_TYPE	-70043	无效的描述符类型
EC_DPI_OP_NOT_SUPPORT	-70044	指定的操作不支持
EC_DPI_INVALID_DESC_INDEX	-70045	无效的描述索引
EC_DPI_INVALID_CURSOR_POSITION	-70046	无效的游标位置

EC_DPI_INVALID_STR_OR_BUF_LEN	-70047	无效的字符串或缓冲区长度
EC_DPI_DESC_CANNOT_MODIFY	-70048	指定描述句柄不可修改
EC_DPI_INVALID_PREC_AND_SCALE	-70049	无效的精度和刻度
EC_DPI_INVALID_DESC_FIELD_ID	-70050	无效的描述符属性
EC_DPI_NOT_CURSOR_STMT	-70051	语句不返回结果集
EC_DPI_INVALID_CURSOR_TYPE	-70052	无效的属性值
EC_DPI_INVALID_ATTR_VALUE	-70053	无效的属性值
EC_DPI_FEATURE_NOT_IMPLEMENT	-70054	特性未实现
EC_DPI_INVALID_FETHC_TYPE	-70055	无效的游标类型
EC_DPI_INVALID_DESC_INFO	-70056	无效的描述信息
EC_DPI_STMT_NOT_PREPARED	-70057	关联的句柄未准备
EC_DPI_RS_READONLY	-70058	只读结果集
EC_DPI_INVALID_BOOKMARK_VAL	-70059	无效的书签值
EC_DPI_ROW_DATA_TOO_LONG	-70060	行数据过长
EC_DPI_DATA_NOT_IN_BUF	-70061	数据不在缓冲区中
EC_DPI_FLDR_INVALID_SCH	-70062	无效的模式名
EC_DPI_FLDR_INVALID_TAB	-70063	无效的表名
EC_DPI_INVALID_CONF_VAL	-70064	无效的配置值
EC_DPI_CONNECTION_SWITCHED	-70065	连接异常,切换当前连接成功
EC_DPI_CONNECTION_SWITCH_FAILED	-70066	连接异常,切换当前连接失败
EC_DPI_COMPRESS_LIB_NOT_LOADED	-70067	压缩库未装载
EC_DPI_MSG_COMPRESS_ERR	-70068	消息压缩错误
EC_DPI_MSG_UNCOMPRESS_ERR	-70069	消息解压错误
EC_DPI_INIT_SSL_FAIL	-70070	初始化 SSL 环境失败
EC_DPI_INVALID_FIELD_IDENTIFIER	-70071	无效的属性标识
EC_DPI_CURSOR_NAME_EXIST	-70072	游标名称已经存在
EC_DPI_KERBEROS_FAILED	-70073	Kerberos 认证失败
EC_DPI_UNKNOWN_OBJ_TYPE	-70074	对象类型未知
EC_DPI_INVALID_OBJ_DESC_HNDL	-70075	无效的对象描述符
EC_DPI_OBJ_DESC_IN_USED	-70076	描述符正被使用
EC_DPI_OBJ_BINDED	-70077	对象句柄已绑定
EC_DPI_USE_DEF_PARAM_VAL	-70078	参数默认值使用无效
EC_DPI_INVALID_CONCUR_ASSIGNED	-70079	无效的并发属性值
EC_DPI_ERROR_IN_ROW	-70080	指定行出现错误
EC_DPI_ROW_VAL_OUT_OF_RANGE	-70081	指定行超过范围
WR_SVR_EMPTY	100	空结果集
WR_SVR_STR_TRUNC	101	字符串截断
WR_SVR_NULL_IN_SFUN	102	在集函数中计算 NULL 值

WR_SVR_INVALID_TABLE_NAME_WARN	103	无效的表名
WR_SVR_EMPTY_DEL	104	删除 0 行记录
WR_SVR_EMPTY_INS	105	插入行记录
WR_SVR_EMPTY_UPD	106	更新行记录
WR_SVR_JUMP_STMT	107	跨语句游标操作
WR_SVR_REVOKE_NONE_WARN	108	回收权限时无相应权限
WR_SVR_EMPTY_CHAR_CAST	109	试图转换空字符串
WR_SVR_BUILD_NOT_COMPLETE	110	编译没有结束
WR_SVR_RESULT_SET_EMPTY	111	结果集数据获取完成
WR_SVR_UTF8_CODE_NOT_INTEGRATED	112	不完整的 UTF8 字符
WR_SVR_RS_CACHE_FULL	113	结果集缓存满
WR_SVR_PREC_TRUNC_WARN	114	刻度截断
WR_SVR_LOCAL_CODE_NOT_INTEGRATED	115	不完整的字符
WR_SVR_RANGE_HP_NOT_MAXVALUE	117	范围分区未包含 MAXVALUE，可能无法定位到分区
WR_SVR_LIST_HP_NOT_DEFAULT	119	列表分区未包含 DEFAULT，可能无法定位到分区
DPI_SUCCESS	70000	操作成功
WR_DPI_EMPTY_CHAR_CAST	70001	空串转换
WR_DPI_BUF_NOT_ENOUGH	70002	缓冲区不足
WR_DPI_STR_BUF_NOT_ENOUGH	70003	字符串缓冲区不足
WR_DPI_STR_TRUNC	70004	字符串截断
WR_DPI_CNVT_LOST_SCALE	70005	转换刻度丢失
WR_DPI_CONN_ATTR_CHANGED	70006	连接属性已改变
WR_DPI_OPTION_VALUE_CHANGED	70007	选项值已改变
WR_DPI_SVR_PRIMARY_SUSPEND	70008	服务器处于主库挂起状态
WR_DPI_SVR_PRIMARY_MOUNT	70009	服务器处于主库配置状态
WR_DPI_SVR_STANDBY_MOUT	70010	服务器处于备库配置状态
WR_DPI_SVR_STANDBY_OPEN	70011	服务器处于备库打开状态
WR_DPI_CHARACTER_NOT_INTEGRATED	70012	不完整的字符
WR_DPI_CURSOR_OPERATIE_CONFLICT	70013	游标操作冲突

3 OCI 错误码汇编

表 OCI 错误码

代码	解释
1405	列值为空(NULL)
1406	数据被截断
1455	转换列溢出整数数据类型

24345	数据截断或空数据
81	非法的参数长度
1	违反唯一性约束
942	表或视图不存在
3113	网络异常
3114	连接未打开
1087	连接已打开
911	非法字符
1653	空间不足
25	内存不足
18	连接数达到上限
30	未经授权的用户
60	死锁
99	锁超时
65	无效的日期字符串
439	特性未实现
69	锁失败
100	没有找到数据
210	控制文件错误
214	数据库版本不匹配
263	无日志提供备份
295	恢复数据文件应从 1 开始编号
296	无效的数据文件路径列表
301	创建日志文件失败
305	恢复数据库名和备份中数据库名不一致
313	缺少本地归档
345	备份信息写失败
359	恢复文件组不匹配
900	无效的语句句柄
902	未知的参数数据类型
903	无效的表名
904	无效的列名
905	语法分析出错
913	变量不在列表内
909	无效的过程/函数名
932	字符串转换出错
934	存在集函数
937	无效的 SELECT 项

938	调用参数不匹配
962	group/order by 包含过多列
953	索引不存在
955	对象已存在
957	列已存在
959	无效的表空间名
961	日期数值越界
976	层次查询不支持分区表
1017	用户名或密码错误
990	无效的权限名
995	无效的同义词名
1003	无效的游标语句
1001	无效的游标名
1016	无效的游标状态
1010	无效的操作
1024	无效的数据库数据类型
1031	没有修改用户权限
1036	无效的变量名
1039	没有修改视图权限
1044	目标缓存长度不足
1048	无效的存储过程名
1070	不支持的服务器版本
1111	无效的数据文件名
1124	数据文件恢复失败
24328	非法属性
1480	'\0'丢失
1412	参数数据长度为 0
1008	参数未绑定
1426	数字溢出
24374	定义数不足
-73000	非法的参数索引
-73001	绑定语句未准备
-73002	定义数量不匹配
-73004	定义位置越界
-73005	非查询语句 fetch 操作
-73006	无效的 direct path 名称
-73007	设置 direct path 所属模式名太长
-73008	不支持的属性

-73009	绑定变量长度非法
-73010	非法的参数
-73012	非法的 fetch 操作
-73014	只写属性
-73015	只读属性
-73017	非法属性值
-73019	语句未准备
-73022	不能进行转换
-73024	非法的参数名称
-73030	超出类型数据缓冲区
-73031	非法的数据长度
-73032	未知错误
-73034	无效的索引号
-73035	无效的对象名
-73036	不支持的模式
-73037	连接已开启
-73038	无效的数据指针
-73039	间隔类型不匹配
-73040	数据溢出
-73041	无效的缓冲区
-73042	无效的句柄类型
-73043	会话未开启
-73044	无效的类型标志
-73045	数据不正确
-73047	会话或事务句柄未指定
-73048	不是绑定数据的最后一片
-73049	绑定数据片记录标志不正确
-73050	到达并发连接数的最大值
-73051	连接并发已经打开
-73052	事务状态不正确
-73053	当前环境缺少会话对象
-73054	包未找到
-73055	不存在可描述的信息
-73056	无效的描述句柄
-73057	Sql 语句词法分析出错
-73058	文件处于关闭状态
-73059	文件读取失败
-73060	打开文件失败

-73061	无效的文件路径长度
-73062	关闭文件失败
-73063	超过行数上限
-73064	直接路径加载不支持的数据类型
-73065	直接路径未准备
-73066	不允许执行直接路径上下文操作
-73067	无效的字符串或格式
-73068	无效的句柄状态
-73069	Lob 已经打开
-73070	无效的数据类型
-73071	周中的日无效
-73072	无效的 lob 定位符

4 OCCI 错误码汇编

表 OCCI 错误码

代码	解释
-77500	内存不足
-77501	方法未实现
-77502	当前值为空
-77503	不能更改绑定数据类型
-77504	超过了最大迭代次数
-77505	获取方法与参数的类型不匹配
-77506	参数不支持
-77507	数据流已打开
-77508	无效的参数
-77509	列序号非法
-77510	数据类型非法
-77511	偏移越界
-77512	无效的句柄状态
-77513	无效的函数调用序列
-77514	无效的属性
-77515	连接未指定
-77516	无效的 Lob 句柄
-77517	日期为空
-77518	环境未指定
-77519	无效的时间值
-77520	数据溢出

-77521	时间溢出
-77522	无法获取有关此列的信息
-77523	不能更改绑定模式
-77524	数据无法转换

附录 2 DM 技术支持

如果您在安装或使用 DM 及其相应产品时出现了问题，请首先访问我们的 Web 站点 http://www.dameng.com/。在此站点我们收集整理了安装使用过程中一些常见问题的解决办法，相信会对您有所帮助。

您也可以通过以下途径与我们联系，我们的技术支持工程师会为您提供服务。

达梦数据库（武汉）有限公司

地址：武汉市东湖高新技术开发区高新大道 999 号未来科技大厦 C3 栋 16-19 层

邮编：430073

电话：(+86)027-87588000

传真：(+86)027-87588000-8039

达梦数据库（北京）有限公司

地址：北京市海淀区北三环西路 48 号数码大厦 B 座 905

邮编：100086

电话：(+86)010-51727900

传真：(+86)010-51727983

达梦数据库（上海）有限公司

地址：上海市闸北区江场三路 28 号 301 室

邮编：200436

电话：(+86)021-33932716

传真：(+86)021-33932718

地址：上海市浦东张江高科技园区博霞路 50 号 403 室

邮编：201203

电话：（+86）021-33932717

传真：（+86）021-33932717-801

达梦数据库（广州）有限公司

地址：广州市荔湾区中山七路 330 号荔湾留学生科技园 703 房

邮编：510145

电话：(+86)020-38371832

传真：(+86)020-38371832

达梦数据库（海南）有限公司

地址：海南省海口市玉沙路富豪花园 B 座 1602 室

邮编：570125

电话：(+86)0898-68533029

传真：(+86)0898-68531910

达梦数据库（南宁）办事处

地址：广西省南宁市科园东五路四号南宁软件园五楼

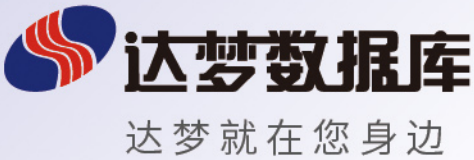
邮编：530003

电话：（+86）0771-2184078
传真：（+86）0771-2184080

达梦数据库（合肥）办事处
地址：合肥市包河区马鞍山路金帝国际城 7 栋 3 单元 706 室
邮编：230022
电话：（+86）0551-3711086

达梦数据库（深圳）办事处
地址：深圳市福田区皇岗路高科利大厦 A 栋 24E 邮编：518033
电话：0755-83658909
传真：0755-83658909

技术服务：
电话：400-991-6599
邮箱：dmtech@dameng.com



武汉达梦数据库有限公司

总部：武汉市东湖高新技术开发区高新大道999号
未来科技大厦C3栋16—19层

电话：(+86) 027-87588000