



PROGRAMMER MANUAL

程序员手册



目录

第 1 章 概述	1
1.1 功能特性	1
1.1.1 通用性	1
1.1.2 高性能	2
1.1.3 高安全性	4
1.1.4 高可靠、高可用性	5
1.1.5 易用性	6
1.1.6 对存储模块的支持	8
1.1.7 对 Web 应用的支持	8
1.2 主要技术指标	9
第 2 章 DPI 编程指南	10
2.1 DPI 简介	10
2.2 DPI 句柄	14
2.2.1 环境句柄	14
2.2.2 连接句柄	15
2.2.3 语句句柄	18
2.2.4 描述符句柄	21
2.2.5 LOB 句柄	23
2.3 函数原型	23
2.4 编程参考	69
2.4.1 编程步骤	69
2.4.2 普通数据插入与查询方式的操作	70
2.4.3 大字段操作	87
2.5 数据捕获	101
2.5.1 数据类型	101
2.5.2 相关方法	102
2.5.3 数据信息搜集表	104
2.5.4 基本示例	105
第 3 章 DMODBC 编程指南	109
3.1 数据类型	109
3.2 支持的函数	111
3.2.1 连接到数据源	111
3.2.2 获取驱动程序和数据源信息	112
3.2.3 设置或者获取驱动程序属性	112
3.2.4 设置或者获取描述符字段	112
3.2.5 准备 SQL 语句	112
3.2.6 提交 SQL 请求	113

3.2.7 检索结果集及其相关信息	113
3.2.8 取得数据源系统表的信息	113
3.2.9 终止语句执行	114
3.2.10 中断连接	114
3.3 建立 ODBC 连接	114
3.3.1 申请环境与连接句柄	114
3.3.2 如何与数据源进行连接	115
3.3.3 设置与取得连接的属性	117
3.3.4 断开与数据源之间的连接	118
3.4 ODBC 应用程序编程的基本步骤	119
3.4.1 Windows 上创建 ODBC 数据源	119
3.4.2 Linux 上创建 ODBC 数据源	123
3.4.3 ODBC 应用程序编写的基本步骤	124
3.5 使用存储过程和函数	127
3.5.1 存储过程与函数字典信息的获取	127
3.5.2 存储模块的创建	128
3.5.3 存储模块的调用	128
3.6 基本示例	130
第 4 章 DM JDBC 编程指南	132
4.1 JDBC 介绍	132
4.2 基本示例	132
4.3 DM JDBC 特性	140
4.4 DM JDBC 扩展	141
4.4.1 数据类型扩展	141
4.4.2 读写分离集群下的错误信息	142
4.5 建立 JDBC 连接	143
4.5.1 通过 DriverManager 建立连接	143
4.5.2 创建 JDBC 数据源	144
4.5.3 数据源与连接池	145
4.5.4 DM 扩展连接属性的使用	146
4.5.5 JDBC 驱动端与其他数据库的兼容性处理	151
4.5.6 获取执行计划	152
4.6 Statement/PreparedStatement/CallableStatement	153
4.6.1 Statement	153
4.6.2 PreparedStatement	154
4.6.3 CallableStatement	156
4.6.4 获取执行过程中服务器返回的打印消息	158
4.7 ResultSet	159
4.8 流与大对象	162
4.8.1 Stream 使用	162
4.8.2 LOB 对象使用	164
4.9 元数据	165
4.9.1 ResultSetMetaData	165

4.9.2 DatabaseMetaData	165
4.9.3 ParameterMetaData	166
4.10 RowSet	167
4.10.1 CachedRowSet	167
4.10.2 JdbcRowSet	168
4.11 分布式事务支持	169
4.11.1 XADataSource	169
4.11.2 XAConnection	170
4.11.3 XAResource	170
4.11.4 Xid	171
4.11.5 基本示例	171
4.12 如何使用 JDBC 接口操作空间数据	172
4.12.1 空间数据类型	173
4.12.2 空间数据的读写步骤	173
4.12.3 基本示例	174
 第 5 章 .NET Data Provider 编程指南	176
5.1 数据类型	176
5.2 提供的对象和接口	177
5.2.1 DmConnection 对象	177
5.2.2 DmCommand 对象	179
5.2.3 DmDataAdapter 对象	180
5.2.4 DmDataReader 对象	181
5.2.5 DmParameter 对象	181
5.2.6 DmParameterCollection 对象	182
5.2.7 DmTransaction 对象	182
5.2.8 DmCommandBuilder 对象	183
5.2.9 DmConnectionStringBuilder 对象	183
5.2.10 DmClob 对象	183
5.2.11 DmBlob 对象	184
5.2.12 DmBulkCopy 对象	184
5.3 注册 .NET 驱动	184
5.4 NHibernate Dm 方言包	185
5.5 EF Dm Provider.EF6 方言包	186
5.6 DM-EFCore 方言包	187
5.6.1 介绍	187
5.6.2 使用	187
5.6.3 示例	188
5.7 对象使用	189
5.7.1 连接	189
5.7.2 查询与结果集	189
5.7.3 插入、更新、删除	190
5.7.4 大对象	192
5.7.5 自增列	193

5.7.6 存储过程与函数.....	193
5.8 基本示例	195
第 6 章 DM PHP 编程指南.....	198
6.1 DM PHP 介绍.....	198
6.2 DM PHP 模块加载.....	204
6.2.1 linux 系统下 PHP 加载	204
6.2.2 Windows 系统下加载 PHP 模块.....	206
6.2.3 PHP INI 文件和 libphp53_dm.so 类介绍.....	207
6.3 编程接口	209
6.3.1 PHP 5.X 接口.....	209
6.3.2 PHP 7.X 接口.....	224
6.3.3 PHP 8.X 接口.....	238
6.4 基本示例	242
第 7 章 DM FLDR 编程指南	246
7.1 C 编程指南.....	246
7.1.1 接口介绍.....	246
7.1.2 接口说明.....	246
7.1.3 基本示例.....	256
7.2 JNI 编程指南	257
7.2.1 接口介绍.....	257
7.2.2 接口说明.....	258
7.2.3 基本示例.....	263
第 8 章 DM DEXP/DIMP JNI 编程指南.....	270
8.1 接口介绍	270
8.2 接口说明	270
8.3 基本示例	271
第 9 章 Logmnr 接口使用说明.....	278
9.1 JNI 接口	278
9.1.1 接口说明.....	278
9.1.2 基本示例.....	283
9.2 C 接口.....	285
9.2.1 接口说明.....	285
9.2.2 基本示例.....	290
第 10 章 DM Node.js 编程指南.....	293
10.1 Node.js DM 数据库驱动介绍.....	293
10.2 Node.js 驱动包安装与环境准备.....	293
10.3 对象使用	293
10.3.1 dmdb 对象	293
10.3.2 Connection 对象	297

10.3.3 Lob 对象.....	302
10.3.4 Pool 对象	303
10.3.5 ResultSet 对象	304
10.4 连接串可配置属性	306
10.5 基本示例	308
第 11 章 DM Go 编程指南.....	313
11.1 Go DM 数据库驱动介绍.....	313
11.2 环境准备	313
11.3 连接串属性说明	313
11.4 相关方法	315
11.4.1 DB	315
11.4.2 Row.....	319
11.4.3 Rows	319
11.4.4 Stmt	321
11.4.5 Tx.....	322
11.5 扩展对象	324
11.5.1 DmBlob	325
11.5.2 DmClob	326
11.5.3 DmTimestamp.....	328
11.5.4 DmDecimal	329
11.5.5 DmIntervalYM.....	330
11.5.6 DmIntervalDT	331
11.5.7 DmArray	331
11.6 批量执行	333
11.7 ORM 方言包	334
11.8 基本示例	334
第 12 章 DM XA 编程指南.....	340
12.1 X/Open 分布式事务处理	340
12.1.1 DTP 术语.....	340
12.1.2 所需公共信息	342
12.2 DM XA 接口.....	343
12.2.1 DM XA 接口常规功能	343
12.2.2 DM XA 接口附加功能	343
12.3 开发和安装 XA 应用程序	344
12.3.1 DBA 或者系统管理员的职责	344
12.3.2 应用程序开发人员的职责	344
12.3.3 打开字符串 xa_open.....	344
12.4 XA 应用程序故障排除	345
12.4.1 访问 DM XA 的日志文件	345
12.4.2 访问 DM XA 的跟踪文件	345
第 13 章 msgparse 编程指南.....	346

13.1 接口介绍	346
13.1.1 获取请求数据包类型	346
13.1.2 检测响应数据是否为登录成功响应.....	347
13.1.3 检测请求数据是否含有完整包长.....	347
13.1.4 检测请求数据是否为新 SQL 指令	348
13.1.5 构建错误应答报文	349
13.1.6 提取请求数据中的 SQL 和参数	350
13.1.7 展示可解析的消息格式版本	352
13.1.8 获取数据库登录请求的附加用户信息.....	352
13.1.9 获取 STARTUP 启动的附加信息.....	353
13.2 基本示例	356
13.2.1 Windows 环境示例	356
13.2.2 Linux 环境示例.....	366
第 14 章 DM R2DBC 编程指南.....	367
14.1 DM R2DBC 介绍.....	367
14.2 DM R2DBC 编程步骤.....	367
14.3 DM R2DBC 特性.....	368
14.4 DM R2DBC 扩展.....	369
14.5 DM R2DBC 提供的接口和对象.....	369
14.5.1 建立 R2DBC 连接对象.....	369
14.5.2 Statement 接口	371
14.5.3 DmStatement 对象	371
14.5.4 DmConnection 对象	371
14.5.5 DmBatch 对象	372
14.5.6 Blob 对象	373
14.5.7 Clob 对象	373
14.5.8 DmColumnMetaData 对象	373
14.5.9 DmConnectionFactory 对象	373
14.5.10 DmConnectionFactoryMetadata 对象	373
14.5.11 DmConnectionFactoryProvider 对象	374
14.5.12 DmConnectionMetadata 对象	374
14.5.13 DmR2dbcType 对象	374
14.5.14 DmResult 对象.....	374
14.5.15 DmRow 对象.....	375
14.5.16 DmRowMetadata 对象.....	375
14.6 基本示例.....	376
附录 1 错误码汇编	381
1 DM 服务器错误码汇编	381
2 DPI 错误码汇编.....	381
附录 2 DM 技术支持	385

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 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、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 接口驱动程序和符合 JDBC 3.0 标准的 JDBC 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/TITAN）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. 支持对称多处理器系统

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 采用多版本并发控制和独特的封锁机制来解决数据库的并发控制问题。多版本并发控制确保数据库的读操作与写操作不会相互阻塞，大幅度提升数据库的并发度以及使用体验。封锁机制采用 TID 锁和对象锁进行并发访问控制，有效减少封锁冲突、提升系统并发性能。

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 位处理器和操作系统, 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));` 则可以通过函数索引 `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求。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. 三权分立或四权分立的安全机制

DM 在安全管理方面采用了“三权分立”或“四权分立”的安全管理体制。

三权分立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。

四权分立把系统管理员分数据库管理员、数据库对象操作员、数据库安全员和数据库审计员四类。在“三权分立”的基础上，新增数据库对象操作员账户 DBO。DBO 可以创建数据库对象，并对自己拥有的数据库对象具有所有的对象权限并可以授出与回收，但其无法管理与维护数据库对象。

“三权分立”或“四权分立”的管理体制真正做到三权分立或四权分立，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。DMDSC 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. 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。

3. 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。

4. 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。

5. 类型别名

类型别名实际上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。

6. 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。

7. 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。

8. 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。

9. 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。

10. 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。

11. 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，脱机备份与恢复，以及系统信息查看等任务。

12. 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 dmrmn、快速数据装载工具 dmfldr、导入导出工具 dexp/dimp 等。

13. 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、数据页。

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页构成。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。

14. 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，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. 数值类型的最高精度为 38 个有效数字。
13. 在一个列上允许建立的最多索引数 1020。
14. 表上的最大 UNIQUE 索引数为 64。

1.3 连接配置项优先级

接口中有一部分参数是在连接时设置的，这部分参数称为连接配置项。有的连接配置项既可以在接口中调用设置，也可以在 `dm_svc.conf` 文件中设置。不同接口的连接配置项各不相同。

对于 DPI、ODBC、PROC、dmPython、DM FLDR 接口，如果同一个参数在接口中和在 `dm_svc.conf` 配置项（全局配置区、服务配置区）中均有设置，但值却不同，则优先顺序为：全局配置区<接口设置<服务配置区。其中 PROC 请参考《DM8_PROC 使用手册》，dmPython 请参考《DM8_dmPython 使用手册》。

对于 JDBC、.NET Data Provider、GO、PHP、DM DEXP/DIMP JNI、Node.js、DM XA、msgparse、DM R2DBC 接口，接口连接串中可设置的属性中除了 `user` 和 `password` 是必须要设置的，其它属性均为可选项。如果同一个属性在接口中和在 `dm_svc.conf` 配置项（全局配置区、服务配置区）中均有设置，但值却不同，则优先顺序为：全局配置区<服务配置区<接口设置。

对于 Logmngr 接口，只需要在接口中设置连接配置项，不涉及 `dm_svc.conf` 文件设置。

第 2 章 DPI 编程指南

2.1 DPI 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 DPI 句柄

2.2.1 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	字段类型	取值
<code>DSQL_ATTR_LOCAL_CODE</code>	本地编码	<code>sdint4</code>	<code>PG_GBK</code> 等
<code>DSQL_ATTR_LANG_ID</code>	错误消息的语言	<code>sdint4</code>	<code>LANGUAGE_EN</code> 或者 <code>LANGUAGE_CN</code>

2.2.2 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	字段类型	取值
<code>DSQL_ATTR_ACCESS_MODE</code>	连接的访问模式（可读写）	<code>udint4</code>	<code>DSQL_MODE_READ_ONLY</code> 或者 <code>DSQL_MODE_READ_WRITE</code> 。 缺省为 <code>DSQL_MODE_READ_WRITE</code>
<code>DSQL_ATTR_ASYNC_ENABLE</code>	允许异步执行，未提供	<code>udint4</code>	未支持
<code>DSQL_ATTR_AUTO_IPD</code>	自动分配参数描述符（只读）	<code>udint4</code>	
<code>DSQL_ATTR_AUTOCOMMIT</code>	自动提交（可读写）	<code>udint4</code>	<code>DSQL_AUTOCOMMIT_ON</code> 或者 <code>DSQL_AUTOCOMMIT_OFF</code> 。 缺省为 <code>DSQL_AUTOCOMMIT_ON</code>
<code>DSQL_ATTR_CONNECTION_DEAD</code>	连接存活，未提供	<code>udint4</code>	
<code>DSQL_ATTR_CONNECTION_TIMEOUT</code>	执行超时时间（可读写）	<code>udint4</code>	<code>udint4</code> 数值
<code>DSQL_ATTR_LOGIN_TIMEOUT</code>	登录超时时间	<code>udint4</code>	<code>udint4</code> 数值
<code>DSQL_ATTR_PACKET_SIZE</code>	网络包大小，未提供	<code>udint4</code>	
<code>DSQL_ATTR_TXN_ISOLATION</code>	事务隔离级（可读写）	<code>udint4</code>	<code>ISO_LEVEL_READ_UNCOMMITTED</code> , <code>ISO_LEVEL_READ_COMMITTED</code> , <code>ISO_LEVEL_SERIALIZABLE</code>
<code>DSQL_ATTR_LOGIN_PORT</code>	登录端口号（可读写）	<code>Udint2</code>	<code>sdint2</code> 数值，缺省为5236
<code>DSQL_ATTR_STR_CASE_SENSITIVE</code>	大小写是否敏感（只读）	<code>udint4</code>	
<code>DSQL_ATTR_MAX_ROW_SIZE</code>	行最大字节数（只读）	<code>udint4</code>	

DSQL_ATTR_LOGIN_USER	登录用户（只读）	sbyte*	
DSQL_ATTR_LOGIN_SERVER	登录服务器IP或unixsocket文件路径名。如果DSQL_ATTR_INET_TYPE是Unix SOCKET，那么DSQL_ATTR_LOGIN_SERVER必须填写unixsocket文件路径名。两者需配套使用。只有Linux环境才支持UNIXSOCKET，Windows不支持（设置了也会被忽略）	sbyte*	
DSQL_ATTR_INSTANCE_NAME	实例名称（只读）	sbyte*	
DSQL_ATTR_CURRENT_SCHEMA	当前模式（只读）	sbyte*	
DSQL_ATTR_SERVER_CODE	服务器的编码（只读）	sdint4	
DSQL_ATTR_LOCAL_CODE	本地编码（可读写）	sdint4	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）	sdint4	LANGUAGE_EN 或者 LANGUAGE_CN
DSQL_ATTR_APP_NAME	应用名称（可读写）	sbyte*	
DSQL_ATTR_COMPRESS_MSG	消息压缩（可读写）	sdint4	DSQL_TRUE 压缩, DSQL_FALSE 不压缩
DSQL_ATTR_CURRENT_CATALOG	（只读）	sbyte*	
DSQL_ATTR_RWSEPARATE	读写分离（可读写）	udint4	DSQL_RWSEPARATE_OFF 关闭; DSQL_RWSEPARATE_ON 开启; DSQL_RWSEPARATE_ON2 开启, 只连接事务一致性备库
DSQL_ATTR_RWSEPARATE_PERCENT	读写分离比例（可读写）	udint4	取值范围 0~100, 缺省为 25
DSQL_ATTR_TRX_STATE	事务状态（只读）	sdint4	DSQL_TRX_ACTIVE 和 DSQL_TRX_COMPLETE
DSQL_ATTR_USE_STMT_POOL	语句句柄缓存池（可读写）	dint4	DSQL_TRUE 开启, DSQL_FALSE 关闭
DSQL_ATTR_SSL_PATH	SSL证书所在的路径（可读写）	sbyte*	字符串（最长256）
DSQL_ATTR_SSL_PWD	SSL加密密码（可读写）	sbyte*	字符串（最长512）
DSQL_ATTR_MPP_LOGIN	mpp登录方式（可读写）	sdint4	DSQL_MPP_LOGIN_GLOBAL 全局登录

			DSQL_MPP_LOGIN_LOCAL 本地登录
DSQL_ATTR_UKEY_NAME	UKEY登录验证的UKEY名	sbyte*	
DSQL_ATTR_UKEY_PIN	UKEY登录验证时的UKEY密钥	sbyte*	最长256
DSQL_ATTR_UDP_FLAG	UDP通讯的收发模式	sdint4	1表示单发单收，2表示多发多收，缺省为2
DSQL_ATTR_INET_TYPE	指定网络连接类型	sbyte*	取值为： DSQL_INET_TCP：对应TCP协议； DSQL_INET_UDP：对应UDP协议； DSQL_INET_IPC：对应IPC协议； DSQL_INET_UNIXSOCKET：对应UNIXSOCKET协议； DSQL_INET_RDMA：对应RDMA协议； 缺省为 DSQL_INET_TCP
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	ubyte*	字符串（最长256）
DSQL_ATTR_TCNAME_LOWER	是否将通过描述获取的表名和列名转换为小写	udint4	DSQL_TRUE或者DSQL_FALSE。 缺省为 DSQL_FALSE
DSQL_ATTR_QUOTE_REPLACE	指定连接上所创建的语句句柄在执行语句时是否替换语句中的双引号为单引号	udint4	DSQL_TRUE是；DSQL_FALSE否。缺省为DSQL_FALSE
DSQL_ATTR_COMPATIBLE_MODE	运行兼容模式。在连接之前可设置	udint4	DSQL_COMPATIBLE_MODE_DM0UL 或者 DSQL_COMPATIBLE_MODE_ORA1UL。缺省为0

2.2.3 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	字段类型	取值
<code>DSQL_ATTR_ROW_BIND_TYPE</code>	行绑定类型（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_ROW_BIND_OFFSET_PTR</code>	行绑定偏移（可读写）	<code>ulength*</code>	
<code>DSQL_ATTR_ROW_OPERATION_PTR</code>	绑定行数据处理指示（可读写）	<code>udint2*</code>	
<code>DSQL_ATTR_ROW_STATUS_PTR</code>	获取行数据状态指示（可读写）	<code>udint2*</code>	
<code>DSQL_ATTR_ROWS_FETCHED_PTR</code>	已获取行数指示（可读写）	<code>ulength*</code>	
<code>DSQL_ATTR_ROW_ARRAY_SIZE</code>	行集大小（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_ROWSET_SIZE</code>	行集大小（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_USE_BOOKMARKS</code>	是否使用书签（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_FETCH_BOOKMARK_PTR</code>	书签值（可读写）	<code>ulength*</code>	
<code>DSQL_ATTR_PARAM_BIND_OFFSET_PTR</code>	参数绑定值的偏移位置（可读写）	<code>ulength*</code>	
<code>DSQL_ATTR_PARAM_BIND_TYPE</code>	参数绑定类型（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_PARAM_OPERATION_PTR</code>	参数数据处理指示（可读写）	<code>udint2*</code>	
<code>DSQL_ATTR_PARAM_STATUS_PTR</code>	参数使用状态（可读写）	<code>udint2*</code>	
<code>DSQL_ATTR_PARAMS_PROCESSED_PTR</code>	参数集中参数处理的个数（可读写）	<code>ulength*</code>	
<code>DSQL_ATTR_PARAMSET_SIZE</code>	参数集大小（可读写）	<code>ulength</code>	
<code>DSQL_ATTR_ROW_NUMBER</code>	当前行位置（只读）	<code>ulength</code>	
<code>DSQL_ATTR_IMP_ROW_DESC</code>	服务器端的行描述（只读）	<code>void*</code>	

DSQL_ATTR_IMP_PARAM_DESC	服务器端参数描述（只读）	void*	
DSQL_ATTR_APP_PARAM_DESC	应用程序参数描述（可读写）	void*	
DSQL_ATTR_APP_ROW_DESC	应用程序行描述（可读写）	void*	
DSQL_ATTR_CURSOR_TYPE	游标类型（可读写）	ulength	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）	ulength	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）	ulength	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）	ulength	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）	ulength	
DSQL_ATTR_MAX_ROWS	结果集返回的最大行数（可读写）	ulength	
DSQL_ATTR_NOSCAN	是否检查语句中的转义符（可读写），未提供	ulength	
DSQL_ATTR_QUERY_TIMEOUT	执行超时时间（可读写），未提供	ulength	
DSQL_ATTR_RETRIEVE_DATA	游标滚动后是否检索数据（可读写），未提供	ulength	
DSQL_ATTR_ENABLE_AUTO_IPD	自动分配参数描述符（可读写）	ulength	DSQL_TRUE 或者 DSQL_FALSE
DSQL_ATTR_ASYNC_ENABLE	异步执行（可读写），未提供	ulength	
DSQL_ATTR_KEYSET_SIZE	键集驱动游标结果集中行的大小（可读写），未提供	ulength	
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	是否允许模糊查询（可读写）	ulength	DSQL_TRUE DSQL_FALSE
DSQL_ATTR_SQL_CHARSET	字符集编码（可读写）	sdint4	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）	sdint4	DSQL_TRUE DSQL_FALSE
DSQL_ATTR_QUOTE_REPLACE	指定连接上所创建的语句句柄在执行语句时是否替换语句中的双引号为单引号（可读写）	udint4	DSQL_TRUE是；DSQL_FALSE否。缺省为继承自句柄所在连接句柄的属性

2.2.4 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	字段类型
DSQL_DESC_ALLOC_TYPE	描述符的分配类型 (只读)	sdint2
DSQL_DESC_ARRAY_SIZE	描述符的记录数组大小 (APD, ARD 可读写)	ulength
DSQL_DESC_ARRAY_STATUS_PTR	数据状态指示 (可读写)	udint2*
DSQL_DESC_BIND_OFFSET_PTR	绑定数据偏移 (APD, ARD 可读写)	slength*
DSQL_DESC_BIND_TYPE	绑定类型 (APD, ARD 可读 写)	ulength
DSQL_DESC_COUNT	描述符包含记录个数 (APD, ARD, IPD 可读写, IRD 只读)	sdint2
DSQL_DESC_ROWS_PROCESSED_PTR	处理数据的个数 (IRD, IPD 可读写)	ulength*

一个描述符包括了以下的记录域字段：

属性	说明	字段类型
DSQL_DESC_AUTO_UNIQUE_VALUE	结果集列是否是自动增长的 (IRD 只 读)	sdint4
DSQL_DESC_BASE_COLUMN_NAME	结果集列的基列列名 (IRD 只读)	sdbyte*
DSQL_DESC_BASE_TABLE_NAME	结果集列的基表名 (IRD 只读)	sdbyte*
DSQL_DESC_CASE_SENSITIVE	对象是否大小写敏感 (IRD, IPD 只读)	sdint4
DSQL_DESC_CATALOG_NAME	列对应的库名 (IRD 只读)	sdbyte*
DSQL_DESC_CONCISE_TYPE	对象对应的精简类型 (APD, ARD, IPD 可读写,IRD 只读)	sdint2
DSQL_DESC_DATA_PTR	绑定数据的地址 (APD, ARD 可读写)	void*
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)	sdint2
DSQL_DESC_DATETIME_INTERVAL_PRECISION	时间间隔类型的引导精度 (APD, ARD,	sdint4

	IPD 可读写, IRD 只读)	
DSQL_DESC_DISPLAY_SIZE	列的显示长度 (IRD 只读)	slength
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)	sdint2
DSQL_DESC_INDICATOR_PTR	指示符地址 (APD, ARD 可读写)	slength*
DSQL_DESC_LABEL	列的标签 (IRD 只读)	sdbyte*
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)	ulength
DSQL_DESC_LITERAL_PREFIX	列的前缀字符 (IRD 只读)	sdbyte*
DSQL_DESC_LITERAL_SUFFIX	列的后缀字符 (IRD 只读)	sdbyte*
DSQL_DESC_LOCAL_TYPE_NAME	本地的类型名 (IPD, IRD 只读)	sdbyte*
DSQL_DESC_NAME	列的别名 (IPD 可读写, IRD 只读)	sdbyte*
DSQL_DESC_NULLABLE	记录域是否可以为空 (IPD, IRD 只读)	sdint2
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)	sdint4
DSQL_DESC_OCTET_LENGTH	记录域的最大字节长度 (APD, ARD, IPD 可读写, IRD 只读)	slength
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)	slength*
DSQL_DESC_PARAMETER_TYPE	参数类型 (IPD 可读写)	sdint2
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)	sdint2
DSQL_DESC_ROWVER	指示列是否记录行的版本信息 (IPD, IRD 只读)	sdint2
DSQL_DESC_SCALE	DEC 或者 NUMERIC 类型的刻度	sdint2
DSQL_DESC_SCHEMA_NAME	列所属基表所在的模式名 (IRD 只读)	sdbyte*
DSQL_DESC_SEARCHABLE	列的条件查询方式 (IRD 只读)	sdint2
DSQL_DESC_TABLE_NAME	列所在的基表表名 (IRD 只读)	sdbyte*
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)	sdint2
DSQL_DESC_TYPE_NAME	数据源的数据类型 (IPD, IRD 只读)	sdbyte*
DSQL_DESC_UNNAMED	记录域是否是命名的 (IPD 可读写, IRD 只读)	sdint2
DSQL_DESC_UNSIGNED	指示记录域是否为无符号数字类型 (IPD, IRD 只读)	sdint2
DSQL_DESC_UPDATABLE	列是否为可更新 (IRD 只读)	sdint2
DSQL_DESC_BIND_PARAMETER_TYPE	参数类型 (IPD 可读写), 未提供	sdint2
DSQL_DESC_DATETIME_FORMAT	时间类型格式 (APD, ARD 可读写),	sdbyte*

	未提供	
DSQL_DESC_CHARSET	字符集编码 (APD, ARD 可读写), 未提供	sdint4

2.2.5 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
<code>DSQL_DIAG_NUMBER</code>	诊断信息个数
<code>DSQL_DIAG_DYNAMIC_FUNCTION_CODE</code>	语句类型
<code>DSQL_DIAG_ROW_COUNT</code>	语句执行影响的行数
<code>DSQL_DIAG_PRINT_INFO</code>	存储过程或语句块的打印信息
<code>DSQL_DIAG_EXPLAIN</code>	<code>explain</code> 语句的结果

诊断域信息包含如下字段。

属性	说明
<code>DSQL_DIAG_COLUMN_NUMBER</code>	出错的列号
<code>DSQL_DIAG_ROW_NUMBER</code>	出错的行号
<code>DSQL_DIAG_MESSAGE_TEXT</code>	错误信息
<code>DSQL_DIAG_ERROR_CODE</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     *hndl_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

1) 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

2) 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(
    dpi_hcon      dpi_hcon,
    dpi_hstmt     *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(
    dpi_hcon      dpi_hcon,
    dpi_hdesc     *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,
    dhlblctr   *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_INVALID_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(
    dhdsc      dpi_hdsc
);
```

功能

释放描述符句柄。

参数

dpi_hdsc

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 [2.2.1 环境句柄](#)。
- 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

14. dpi_set_con_attr

函数

```
DPIRETURN
dpi_set_con_attr(
    dpi_hcon      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 [2.2.2 连接句柄](#)。

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

15. dpi_set_stmt_attr

函数

```
DPIRETURN
dpi_set_stmt_attr(
    dpi_hstmt      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 [2.2.3 语句句柄](#)。

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`

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

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。

4) 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。

5) 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。

6) buf_sz

输入参数，err_msg 缓冲区的长度。

7) 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 `hdl_type` 所标示的 `hdl` 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诊断头或诊断域的值。

参数

1) `hdl_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) `hdl`

输入参数，需要取得诊断信息且由 `hdl_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头或诊断域。更多信息参见 2.2 节中关于诊断信息的说明。

5) `diag_info`

输出参数，指向返回诊断头或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。
 - 所要求的诊断头或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头或诊断域。
 - 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(
    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

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

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

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

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。描述符句柄请参考 [2.2.4 描述符句柄](#)。

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`

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 用于绑定非调用存储过程的 SQL 语句的输入参数
- DSQL_PARAM_OUTPUT 用于绑定过程、函数、语句的输出参数
- DSQL_PARAM_INPUT_OUTPUT 用于绑定调用存储过程的输入和输出参数
- DSQL_PARAM_INPUT_OUTPUT_STREAM 用于绑定输入输出流
- DSQL_PARAM_OUTPUT_STREAM 用于绑定输出流

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

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

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

在绑定信息作用期间返回绑定参数描述信息；无绑定信息时返回服务器描述信息。

32. dpi_exec**函数**

```

DPIRETURN
dpi_exec(
    dpi_hstmt      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_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。

34. 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。

35. `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。

36. `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`。

37. dpi_prepare**函数**

```
DPIRETURN
dpi_prepare(
    dhstmt      dpi_hstmt,
    sbyte       *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。

38. 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。

39. `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。

40. `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。

41. 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，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。

42. 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。

43. dpi_bind_col**函数**

```
DPIRETURN
dpi_bind_col(
    dhstmt      dpi_hstmt,
    uint2       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。

44. 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。

45. `dpi_desc_column`**函数**

```
DPIRETURN
```

```
dpi_desc_column(
    dpi_hstmt      dhstmt,
    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。

46. dpi_col_attr

函数

```
DPIRETURN
dpi_col_attr(
    dhstmt      dpi_hstmt,
    uint2        icol,
    uint2        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 描述符句柄](#)。

47. dpi_bulk_operation

函数

```
DPIRETURN
dpi_bulk_operation(
    dhstmt      dpi_hstmt,
    uint2       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。

48. 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。

49. `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。

50. **dpi_get_data**

函数

```
DPIRETURN
dpi_get_data(
  dpi_hstmt      dpi_hstmt,
  uint2          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。

51. **dpi_more_results**

函数

```
DPIRETURN
dpi_more_results(
    dpi_hstmt      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。

52. dpi_set_pos**函数**

```
DPIRETURN
dpi_set_pos(
    dpi_hstmt      dpi_hstmt,
    row_num        ulength,
    op             uint2,
    lock_type      uint2
);
```

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。

53. dpi_row_count

函数

```
DPIRETURN
dpi_row_count(
    dpi_hstmt      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。

54. dpi_lob_get_length

函数

```
DPIRETURN
dpi_lob_get_length(
    dpi_loblktr      dpi_loblktr,
    slength        *len
);
```

功能

获取 lob 句柄对应大对象的实际长度。

参数

- 1) dpi_loblktr
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。

55. dpi_lob_read**函数**

```
DPIRETURN
dpi_lob_read(
    dhlblctr      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) dhlblctr

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。

56. 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。

57. 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。

58. dpi_alloc_obj**函数**

```
DPIRETURN
dpi_alloc_obj(
    dhcon           dpi_con,
    dhojb *         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。

59. 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。

60. 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。

61. 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。

62. `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

- `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。

63. dpi_get_obj_attr

函数

```
DPIRETURN
dpi_get_obj_attr(
    dhobj          object,
    uint4           nth,
    uint2           attr_id,
    dpointer        buf,
    uint4           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	uint4

64. dpi_get_obj_desc_attr

函数

```
DPIRETURN
dpi_get_obj_desc_attr(
```

```

dobjdesc      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	dobjdesc
DSQL_ATTR_OBJ_FIELD_COUNT	udint4
DSQL_ATTR_OBJ_SCHAME	sdbyte*

65. dpi_bind_obj_desc**函数**

DPIRETURN

```

dpi_bind_obj_desc(
    dobj          object,

```

```
dobjdesc      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。

66. dpi_unbind_obj_desc**函数**

```
DPIRETURN
dpi_unbind_obj_desc(
    dobj           object
);
```

功能

解除复合对象描述符句柄绑定。

参数

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。

67. dpi_set_obj_val**函数**

```
DPIRETURN
dpi_set_obj_val(
    dobj           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

输入参数，指向存放参数数据的缓冲区的指针。

5) val_len

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。

68. dpi_get_obj_val**函数**

```
DPIRETURN
dpi_get_obj_val(
    dhojbj          object,
    uint4            nth,
    uint2            ctype,
    dpointer         val,
    uint4            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) 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，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) 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

无。

69. dpi_build_rowid

函数

```
DPIRETURN
dpi_build_rowid(
    dhcon          dpi_con,
    sdint4         epno,
    sdint8         partno,
    udint8         real_rowid,
    sdbyte*        rowid_buf,
    uint4          rowid_buf_len,
    uint4*         rowid_len
);
```

功能

生成一个流式的 ROWID 数据。

参数

- 1) dpi_con
输入参数，连接句柄。
- 2) epno
输入参数，站点号。
- 3) partno
输入参数，分区号。
- 4) real_rowid
输入参数，实际的行号。
- 5) rowid_buf
输出参数，输出缓冲区。
- 6) rowid_buf_len
输入参数，缓冲区大小。
- 7) rowid_len
输出参数，实际数据长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO

DSQL_ERROR
DSQL_INVALID_HANDLE

说明

无。

70. dpi_rowid_to_char**函数**

```
DPIRETURN
dpi_rowid_to_char(
    dhcon          dpi_con,
    sdbyte*        rowid,
    uint4          rowid_len,
    sdbyte*        dest_buf,
    uint4          dest_buf_len,
    uint4*         dest_len
)
```

功能

将一个流式的 ROWID 转换为 base64 格式的字符串 ROWID。

参数

- 1) dpi_con
输入参数，连接句柄。
- 2) rowid
输入参数，rowid 数据。
- 3) rowid_len
输入参数，rowid 数据长度。
- 4) dest_buf
输出参数，输出缓冲区。
- 5) dest_buf_len
输入参数，缓冲区大小。
- 6) dest_len
输出参数，实际数据长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

无。

71. dpi_char_to_rowid**函数**

```
DPIRETURN
dpi_char_to_rowid (
    dhcon          dpi_con,
    sdbyte*        rowid_str,
    uint4          rowid_len,
```

```

    sdbyte*      dest_buf,
    uint4        dest_buf_len,
    uint4*       dest_len
)

```

功能

将一个 base64 格式字符串的 ROWID 转换为流式的 ROWID。

参数

- 1) dpi_con
输入参数，连接句柄。
- 2) rowid_str
输入参数，rowid 数据。
- 3) rowid_len
输入参数，rowid 数据长度。
- 4) dest_buf
输出参数，输出缓冲区。
- 5) dest_buf_len
输入参数，缓冲区大小。
- 6) dest_len
输出参数，实际数据长度。

返回值

DSQL_SUCCESS
DSQL_SUCCESS_WITH_INFO
DSQL_ERROR
DSQL_INVALID_HANDLE

说明

无。

2.4 编程参考

2.4.1 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）。
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）。
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）。
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`，在链接阶段需要用到 `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`、`dmmmsg.dll`、`dmos.dll`、`dmutil.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 Linux 平台下编译程序时添加“`-DDM64`”编译参数，Windows 平台下则在相应工程的预处理器定义中添加“`DM64`”。

2.4.2 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 */
dhlblctr    hlblctr;    /* 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_STMT, hstmt);
    //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 = 900000000;
    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_STMT, 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], "adfadsfetere2345ert", 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\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\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\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\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\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
    uint4          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;
}

```

2.4.3 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 "192.168.0.120:5336"
#define DM_USER "SYSDBA"
#define DM_PWD "SYSDBA"
#define IN_FILE      "d:\\drivers_win64+win32.zip"
#define UP_FILE      "d:\\drivers_rh6_64+rh6_32.zip"
//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:*********************  

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;
    slength        c7_ind_ptr = DSQL_DATA_AT_EXEC;
    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 = 900000000;
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), NULL), 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), NULL), 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), NULL),
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_ind_ptr), 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\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;
sbyte 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);
    rt = dpi_lob_write(hloblctr, pos, DSQL_C_BINARY, tmpbuf, len, &wlen);
    if(!DSQL_SUCCEEDED(rt))
    {
        dpi_err_msg_print(DSQL_HANDLE_LOB_LOCATOR, hloblctr);
        return rt;
    }

    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(
#ifdef CPT_WITH_API
    DHCON      CON_HDBC,
#else
    VOID *PIDAO_CONNECTION,
#endif
    SDBYTE*      SCHNAME,
    SDBYTE*      TVNAME,
    SDBYTE*      WHERE,
    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 ROWIDS    BIGINT
);
```

参数说明:

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 ROWIDS: 显示表的主键值, 如果没有主键则显示 rowid。

2.5.4 基本示例

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;
```

第三步, 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,dhenvhenv)
{
    //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");
    //释放连接句柄
    dpi_free_con(hdbc);
    //释放环境句柄
    dpi_free_env(henv);
    exit(-1);
}
return hdbc;
}

void free_hdbc(dhcon hdbc,dhenvhenv)
{
    //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");
}

```

第四步，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

```

第五步，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);

```

第六步，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

第七步，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 (Data Definition Language) 语句的。

调用 DM ODBC 3.0 的 SQLGetTypeInfo，返回支持的数据类型如下表所示：

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-32768~32767
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 $-2^{[63]} \sim 2^{[63]} - 1$
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: $10^{[-38]} \sim 10^{[38]}$
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: $10^{[-308]} \sim 10^{[308]}$
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: $10^{[-308]} \sim 10^{[308]}$
Decimal(p,s)	精度为 p, 刻度为 s 的有符号精确数字值, $1 \leq p \leq 38$, $s \leq p$
Numeric(p,s)	精度为 p, 刻度为 s 的有符号精确数字值, $1 \leq p \leq 38$, $s \leq p$
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。

下表中展示了达梦支持的连接参数及取值范围。

表 3.2 连接参数

属性名称	说明
DRIVER	DM ODBC 驱动的名字: DM8 ODBC DRIVER
DSN	数据源名称
DESCRIPTION	数据源描述
SERVER	目标服务器: ip 地址、服务名或者 socket 文件路径。其中, socket 文件路径必须和 PROTOCOL_TYPE 为 UNIXSOCKET 搭配使用
TCP_PORT	端口号
PROTOCOL_TYPE	协议类型: TCP、UDP、IPC、UNIXSOCKET 或 RDMA。当 PROTOCOL_TYPE 为 UNIXSOCKET 时, SERVER 必须为 socket 文件路径。仅 Linux 环境支持 UNIXSOCKET
UID	用户名
PWD	密码
LANGUAGE	使用的语言信息: ENGLISH 或 CHINESE。可以不指定, 若不指定, 系统会读取操作系统信息获得语言信息, 建议有需要才指定
SSL_PATH	加密文件路径
SSL_PWD	加密文件密码
MPP_LOGIN	MPP 登录方式: 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
UKEY_NAME	UKEY 的名字
UKEY_PIN	UKEY 的密钥
FORCE_SQL_WCHAR	字符类型强制描述为宽字符
TCNAME_LOWER	是否将通过描述获取的表名和列名转换为小写: TRUE, FALSE
QUOTE_REPLACE	是否替换语句中的双引号为单引号: TRUE, FALSE。缺省为 FALSE
COMPATIBLE_MODE	运行兼容模式: DM, ORACLE

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，如下表所示：

表 3.3 连接到数据源的 ODBC 函数

函数	ODBC 版本	一致性	主要参数
<code>SQLConnect</code>	1.0	核心级	<code>hdbc</code> , 数据源, 用户 ID, 口令
<code>SQLDriverConnect</code>	1.0	1 级	<code>hdbc</code> , 窗口句柄, 输入连接字符串
<code>SQLBrowseConnect</code>	1.0	2 级	<code>hdbc</code> , 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，对连接进行全面的管理。下表列出了一些常用的连接属性。

表 3.4 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
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
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。暂不支持
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-管理工具中访问 ODBC 数据源，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。

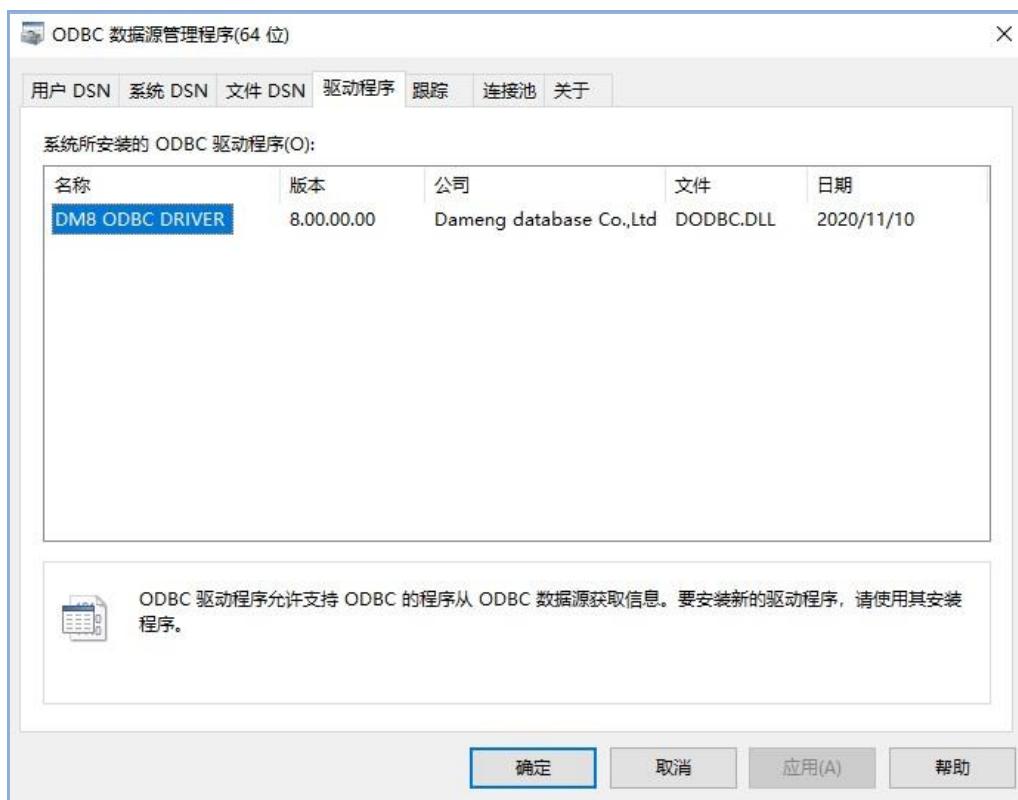


图 3.1 ODBC 数据源管理器对话框

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。

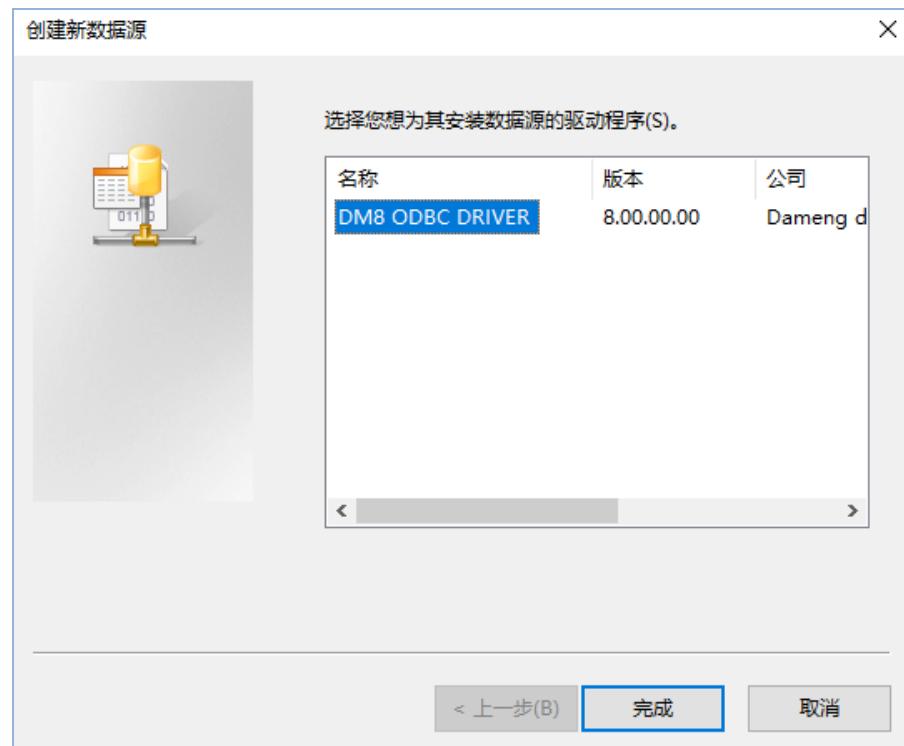


图3.2 创建新数据源对话框

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。

如果连接属性填写的是 UNIXSOCKET，那么服务器需填写 Linux 环境下的 unixsocket 文件路径名。两者需配套使用。只有 Linux 环境才支持 UNIXSOCKET，Windows 不支持（设置了也会被忽略）。

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：

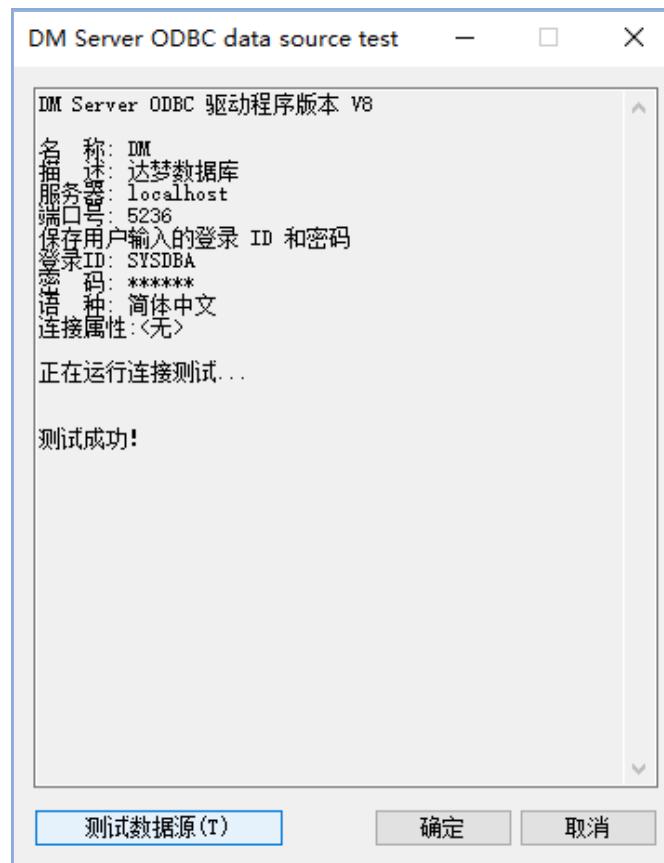


图 3.4 数据源测试报告



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 数据源采用命令行配置的方式。本节将介绍如何为你的应用程序配置 ODBC 数据源。

1. 配置 /etc/odbcinst.ini

odbcinst.ini 内容如下：

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

2. 配置 /etc/odbc.ini

odbc.ini 用于配置 ODBC 的数据源 (DSN)，在 DM 数据源配置项中可配置的连接参数如下表所示：

表3.5 ODBC数据源配置参数

名称	说明
DESCRIPTION	数据源描述

DRIVER	DM ODBC 驱动的名字: DM8 ODBC DRIVER
SERVER	目标服务器: ip 地址、服务名或者 socket 文件路径。其中, socket 文件路径必须和 PROTOCOL_TYPE 为 UNIXSOCKET 搭配使用
TCP_PORT	端口号
UID	用户名
PWD	密码
LANGUAGE	使用的语言信息: ENGLISH, CHINESE
LINK_ATTR	用于配置除了 DESCRIPTION、DRIVER、SERVER、TCP_PORT、UID、PWD、LANGUAGE 之外的连接参数。更多的连接参数请参考 3.2.1 连接到数据源 中的表 3.1。LINK_ATTR 书写格式为 LINK_ATTR=key1=value; key1=value;…… 多个参数之间用 ; 分隔

odbc.ini 内容如下:

使用 TCP 通信协议 (PROTOCOL_TYPE 缺省情况下, 使用 TCP 方式)。

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

使用 UNIXSOCKET 网络连接协议。

```
[dm]
Description = DM ODBC DSN
Driver      = DM8 ODBC DRIVER
SERVER      = /data/sdb/DAMENG/foo.sock //UNIXSOCKET 文件路径名
UID         = SYSDBA
PWD         = SYSDBA
TCP_PORT    = 5236
LINK_ATTR  =PROTOCOL_TYPE=UNIXSOCKET;COMPATIBLE_MODE=ORACLE //UNIXSOCKET 类型
```

注意事项:

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, 或者 unixsocket 文件路径名。

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 编程的基本步骤如下图所示：

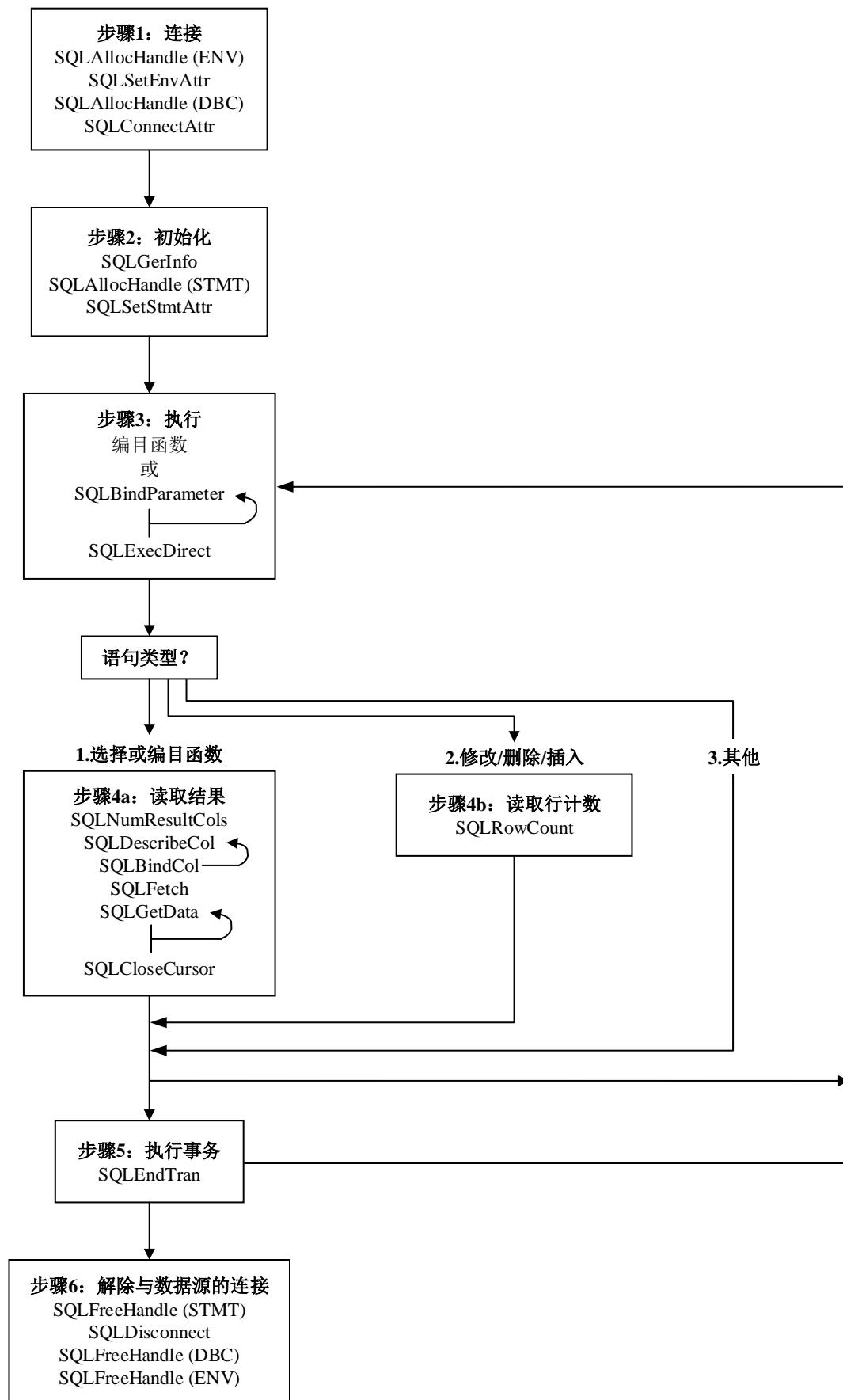


图 3.6 直接使用 ODBC 函数开发应用程序的基本步骤

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下表所列。

表 3.6 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 SQLProcedureColumns 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下表所示。

表 3.7 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;
RETCODE   ret;
short     i;
short     cols;
char      colname[129];
char     冷data[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);
SQLDisconnectdbc);
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
}

```

3.6 基本示例

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      hdcb; //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); }

```

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 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,dis
count,"
        +
"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：DMJdbcDriver16.jar、DMJdbcDriver17.jar 和 DMJdbcDriver18.jar，分别对应 JDK 版本 1.6、1.7 和 1.8，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:\dmdbms 目录，那么达梦数据库驱动程序 DMJdbcDriver.jar 文件就在 D:\dmdbms\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.ParameterMetaData
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.jdbc.driver 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 DmdbIntervalYM 和 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 DmdbTimestamp 类型。这些类型的主要方法为：

1. 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. 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. 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。

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。

```
//显式注册
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 dmdb = new DmdbDataSource ();
    // 把 DmdbDataSource 实例注册到 JNDI 中
    ctx.bind (ln, dmdb);
}
```

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。另外，这些连接串属性也可以在 `dm_svc.conf` 中使用，而 `dm_svc.conf` 中的属性（通用配置项和 JDBC 配置项）也可以在 JDBC 连接串上设置。另外，`dm_svc.conf` 中 `dexp` 配置项、`dpc_new` 配置项、`.Net provider` 配置项、`DPI` 配置项与 JDBC 连接串无关，无须设置。

连接串的书写格式有以下三种：

格式一 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][&...]
```

参数介绍：

`host`: 数据库所在 IP 地址，缺省为 `localhost`。若 `host` 为 `ipv6` 地址，则应包含在 `[]` 中；

`port`: 数据库端口号，缺省为 5236；

`?` : 是参数分隔符，表示后面的都是参数；

`&` : 是参数间隔符，多个参数用 `&` 分开；

`propName`: 连接串属性名称。详见表 4.1；

`propValue`: 连接串属性值。

例如：

```
jdbc:dm://192.168.0.96:5236?resultSetType=1003
```

格式二 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][&...]
```

参数介绍：

`host` 和 `port`: 设置与否，以及在属性串中的位置没有限制；

其它参数：和格式一相同。

例如：

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

格式三 使用自定义服务名，可指定多个数据库节点

格式：

```
jdbc:dm://GroupName?GroupName=(host1:port1,host2:port2,...)
```

```
[&propName1=propValue1][&propName2=propValue2][&...]
```

参数介绍：

`GroupName`: 数据库服务名；

其它参数：和格式一相同。

例如：

```
jdbc:dm://test?test=(192.168.0.96:5236,192.168.0.96:5237)
```

JDBC 连接串中可设置的属性中除了 `user` 和 `password` 是必须要设置的，其它属性均为可选项。如果同一个属性在 JDBC 连接串中和 `dm_svc.conf` 配置项中均有设置，但值却不同，则以 JDBC 连接串优先。

JDBC 连接串属性见下表。表中所有时间均可以使用（`h`: 小时，`m`: 分钟，`s`: 秒，`ms`:

毫秒), 不带单位使用时取配置项默认单位 ms, 取值范围 0~2147483647。缺省值为 5 分钟。

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'02000'	否
port	端口号, 服务器登录端口号	否
unixSocketFile	LINUX 系统中, 当服务器与客户端之间使用 UNIXSOCKET/IPC 方式通信时, 用于指定客户端连接的 UNIXSOCKET 路径文件名。使用 unixSocketFile 时, 不需要指定 host 和 port。 例如: jdbc:dm://?user=SYSDBA&password=SYSDBA&unixSocketFile=/home/te/foo.sock	否
user	登录用户	是
password	登录密码	是
appName	客户端应用程序名称	否
osName	操作系统名称	否
socketTimeout	网络通信链路超时时间; 单位 ms, 取值范围 0~2147483647, 0 表示无限制; 缺省为 0	否
sessionTimeout	会话超时时间; 单位 ms, 取值范围 0~2147483647, 0 表示无限制; 缺省为 0	否
connectTimeout	连接数据库超时时间; 单位 ms, 取值范围 0~2147483647, 0 表示无限制; 缺省为 5000	否
StmtPoolSize	语句句柄池大小; 取值范围 0~2147483647, 0 表示关闭; 缺省为 15	否
PStmtPoolSize	prepare 语句句柄池大小; 取值范围 0~2147483647, 0 表示关闭; 缺省为 0	否
pstmtPoolValidTime	prepare 语句缓存的有效时间; 单位 ms, 取值范围 0~2147483647, 0 表示无限制; 缺省为 0	否
escapeProcess	是否进行语法转义处理; 取值 1/0 或 true/false; 缺省为 true; 1/true: 是, 0/false: 否	否
autoCommit	是否自动提交; 取值 1/0 或 true/false; 缺省为 true; 1/true: 是, 0/false: 否	否
alwaysAllowCommit	在自动提交开关打开时, 调用 rollback() 接口是否不报错; 取值 1/0 或 true/false; 缺省为 true; 1/true: 是, 0/false: 否	否
localTImezone	指定客户端本地时区, 对于本地时区相关时间类型会自动完成服务器时区与本地时区的转换; 单位分钟, 取值范围-779~840; 缺省为当前系统时区	否
maxRows	结果集行数限制, 超过上限结果集截断; 取值范围 0~2147483647,	否

	0 表示无限制；缺省为 0	
bufPrefetch	结果集 fetch 预取消息 buffer 大小；单位 KB，取值范围 32~65535。缺省为 0，表示按服务器配置，若结果集上指定了 fetchSize 会自动预估大小	否
LobMode	大字段数据获取模式；1 表示 get 数据时从服务器段获取，2 表示结果集生成时将大字段数据完整缓存到本地；缺省为 1	否
ignoreCase	结果集列名是否忽略大小写；取值 1/0 或 true/false；缺省为 true；1/true：是，0/false：否	否
continueBatchOnError	批量执行出错时是否继续执行；缺省为 false；取值 true/True, false/False; true/True: 是, false/False: 否	否
batchType	批处理模式；1 表示批量绑定执行，2 表示一行一行执行；缺省为 1	否
resultSetType	指定默认创建结果集类型，与 java 标准中结果集类型相对应；取值 1003/1004/1005，缺省为 1003； 1003：对应 ResultSet.TYPE_FORWARD_ONLY； 1004：对应 ResultSet.TYPE_SCROLL_INSENSITIVE； 1005：对应 ResultSet.TYPE_SCROLL_SENSITIVE	否
dbmdChkPrv	编目函数是否进行权限检测；取值 1/0 或 true/false；缺省为 true；1/true：是，0/false：否	否
isBdtaRS	是否使用列模式结果集，需同步服务器开启该功能；取值 1/0 或 true/false；缺省为 true；1/true：是，0/false：否	否
clobAsString	clob 类型列是否调用 resultSetMetaData 的 getColumnType() 映射为 Types.VARCHAR 类型；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否
columnNameCase	结果集列名大小写转换。取值 upper/lower/空值，缺省为空值。 upper 表示转为大写；lower 表示转为小写；空值表示保持不变	否
compatible_mode 或 compatibleMode	兼容其他数据库。取值为数据库名称：oracle 表示兼容 Oracle，mysql 表示兼容 Mysql。本参数和 DM.INI 中的 COMPATIBLE_MODE 同名，但用途不同	否
schema	指定用户登录后的当前模式，默认为用户的默认模式	否
loginMode	指定优先登录的服务器模式。取值范围 0~4；0：优先连接 PRIMARY 模式的库，NORMAL 模式次之，最后选择 STANDBY 模式；1：只连接主库；2：只连接备库；3：优先连接 STANDBY 模式的库，PRIMARY 模式次之，最后选择 NORMAL 模式；4：优先连接 NORMAL 模式的库，PRIMARY 模式次之，最后选择 STANDBY 模式；缺省为 4 当启用读写分离时（即 rwSeparate 不为 0），loginMode 不生效	否
loginStatus	服务名方式连接数据库时只选择状态匹配的库；取值范围 0、3~5； 0 表示不限制；3 表示 mount 状态；4 表示 open 状态；5 表示 suspend 状态；缺省为 0	否
loginDscCtrl	服务名连接数据库时是否只选择 dsc control 节点的库；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否
epSelector	服务名连接数据库时采用何种模型建立连接；0 表示依次选取列表中的不同节点建立连接，使得所有连接均匀地分布在各个节点上；1 表示选择列表中最前面的节点建立连接，只有当前节点无法建立连接时才会选择下一个节点进行连接；缺省为 0	否

autoReconnect	连接发生异常或一些特殊场景下连接处理策略。取值范围 0~7; 0: 关闭连接; 1: 当连接发生异常时自动切换到其他库, 无论切换成功还是失败都会抛一个 SQLException, 用于通知上层应用进行事务执行失败时的相关处理; 2: 配合 ep_selector=1 使用, 如果服务名列表前面的节点恢复了, 将当前连接切换到前面的节点上; 4: 保持各节点会话动态均衡, 通过后台线程检测节点及会话数变化, 并切换连接使之保持均衡。 也可以将 autoReconnect 置为上述几个值的组合值, 表示同时进行多项配置, 如置为 3 表示同时配置 1 和 2; 缺省为 0	否
switchTimes	服务名连接数据库时, 若未找到符合条件的库成功建立连接, 将尝试遍历服务名中库列表的次数; 取值范围 1~2147483647; 缺省为 1	否
switchInterval	服务名连接数据库时, 若遍历了服务名中所有库列表都未找到符合条件的库成功建立连接, 等待一定时间再继续下一次遍历; 单位 ms, 取值范围 0~2147483647; 缺省为 1000	否
cluster	用于标识的集群类型。 DW: 主备; RW: 读写分离; MPP: 大规模并行处理集群; DPC: 分布计算集群; DSC: 共享存储集群。cluster=DSC 须配合 autoReconnect=2、epSelector=1 使用, 用于检测 DSC 集群节点故障恢复是否成功	否
dbAliveCheckFreq	检测数据库是否存活的频率。单位 ms, 取值范围 0~2147483647, 0 表示不检测; 缺省为 0	否
compress	是否压缩消息。取值范围 0~2。0 表示不压缩; 1 表示完全压缩; 2 表示优化的压缩; 缺省为 0	否
compressID	消息压缩算法标识, 最终与服务器支持情况协商决定。0 表示 zip; 1 表示 snappy; 缺省为 0	否
sslFilePath	数据库端开启 ssl 通信加密, 该参数指定 ssl 加密文件的路径	否
sslKeystorePass	数据库端开启 ssl 通信加密, 该参数指定 ssl 加密文件的指令	否
kerberosLoginConfPath	用户名加前缀“//”标识开启 Kerberos 认证, 该参数指定 Kerberos 认证登录配置文件路径	否
uKeyName	Ukey 的用户名	否
uKeyPin	Ukey 的口令	否
cipherPath	第三方加密算法引擎所在路径	否
OsAuthType	指定操作系统认证用户类型, 开启操作系统认证时, 用户名使用系统用户名。取值范围 0~4; 0 表示关闭; 1 表示 DBA; 2 表示 SSO; 3 表示 AUDITOR; 4 表示自适应。缺省为 0	否
loginCertificate	指定登录加密用户名密码公钥所在的路径, 一旦配置即认为开启了客户端的证书加密用户名密码模式	否
mppLocal	是否 MPP 本地连接; 取值 1/0 或 true/false; 缺省为 false; 1/true: 是, 0/false: 否	否
rwSeparate	是否使用读写分离系统。0 表示不启用; 1 表示启用; 2 表示启用, 备库由客户端进行选择, 且只会选择服务名中配置的节点; 4 表示启用, 备库由客户端进行选择, 只连接事务一致性备库。缺省为 0	否

	当启用读写分离时, loginMode 失效	
rwPercent	分发到主库的事务占主备库总事务的百分比; 单位%, 取值范围 0~100; 缺省为 25	否
rwAutoDistribute	读写分离系统事务分发是否由 JDBC 自动管理; 取值 1/0 或 true/false; 缺省为 true; 1/true: 是, 0/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
rwHA	是否开启读写分离系统高可用; 取值 1/0 或 true/false; 缺省为 false; 1/true: 是, 0/false: 否	否
rwStandbyRecoverTime	读写分离系统备库故障恢复检测间隔, 单位 ms, 取值范围 0~2147483647, 0 表示不回复; 缺省为 60000	否
enRsCache	是否开启结果集缓存; 取值 1/0 或 true/false; 缺省为 false; 1/true: 是, 0/false: 否	否
rsCacheSize	设置结果集缓冲区大小, 单位为 MB。取值范围 1~65535, 如果设置太大, 可能导致空间分配失败, 进而使缓存失效	否
rsRefreshFreq	结果集缓存检查更新的频率, 单位 s, 取值范围 0~10000, 如果设置为 0, 则不需检查更新	否
keyWords	标识用户关键字, 所有在列表中的字符串, 如果以单词的形式出现在 sql 语句中, 则这个单词会被加上双引号; 默认为空串	否
logDir	日志等其他一些 JDBC 过程文件生成目录, 默认为 jvm 当前工作目录	否
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; 缺省为 off	否
logFlushFreq	日志刷盘频率; 单位 s, 有效值范围 0~2147483647; 缺省为 60;	否
statEnable	是否启用状态监控; 取值 1/0 或 true/false; 缺省为 false; 1/true: 是, 0/false: 否	否
statDir	状态监控信息以文本文件形式输出的目录, 默认为 jvm 当前工作目录	否
statFlushFreq	状态监控统计信息写文件刷盘频率; 单位 s, 有效值范围 0~2147483647; 0 表示不写文件; 缺省为 10	否
statSlowSqlCount	统计慢 sql top 行数; 取值范围 0~1000; 缺省为 100	否
statHighFreqSqlCount	统计高频 sql top 行数; 取值范围 0~1000; 缺省为 100	否
statSqlMaxCount	状态监控可以统计不同 sql 的个数; 取值范围 0~100000; 缺省为 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 表示淘汰最近执行的 sql, eldest 表示淘汰最老的 sql; 缺省为 eldest	否
dmsvcconf	指定 url 属性配置文件所在路径	否
dbAliveCheckTimeout	检测数据库是否存活的连接超时时间, 如果该时间内未连接成功即认为数据库故障; 单位 ms, 取值范围 1~2147483647; 缺省为 10000	否
checkFreq	服务名连接数据库时, 循环检测连接是否需要重置的时间间隔。即每间隔设定时间检测连接对象是否发生改变, 若连接对象发生改变,	否

	JDBC 连接会自动重置到新对象。单位 ms，取值范围 0~2147483647。缺省为 300000	
prepareOptimize	是否对预编译 SQL 做优化；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否
allowRange	允许动态负载均衡误差范围的百分比，百分比越大表示允许的误差范围越大，取值范围 0~50。缺省为 5	否
localEncrypt	是否启用用户名密码本地加密；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否
localEncryptEngine	本地加密引擎名，不指定时使用系统内置的加密引擎，用户可指定自定义的加密引擎	否
genKeyNameCase	调用 preparedStatement(String sql, String[] columnName) 接口 prepare sql 时可同时指定列名，用于在表数据增删改后返回指定的列值。该参数用于指定这里的列名的大小写。 取值 0/1/2 或 none/upper/lower；缺省为 upper。 取值 0 或 none：保持用户输入不变； 1 或 upper：将用户输入转为大写； 2 或 lower：将用户输入转为小写	否
afterGetMoreResults	调用 JDBC 标准接口 getMoreResults() 后是否关闭之前的结果集；取值 1/2/3；缺省为 1；1：关闭当前一个结果集；2：保持当前结果集打开；3：关闭当前所有结果集	否
checkExecType	是否检查 executeXXX 接口执行的 SQL 语句类型与接口是否匹配；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否
quoteReplace	是否将 SQL 语句中的双引号改成单引号；取值 1/0 或 true/false；缺省为 false；1/true：是，0/false：否	否

4.5.5 JDBC 驱动端与其他数据库的兼容性处理

上一节中介绍的 JDBC 驱动端 compatible_mode 参数，与 DM 数据库服务器端的 DM.INI 参数 compatible_mode 名称相同，但用途不同。

JDBC 驱动端 compatible_mode 的作用是在 DM JDBC 接口表现上与其他数据库的 JDBC 接口表现上做的兼容性处理。该参数取值：oracle：表示兼容 Oracle JDBC 接口；mysql：表示兼容 Mysql JDBC 接口。下面分别进行介绍：

4.5.5.1 兼容 Oracle 的情况

当 JDBC 驱动端 compatible_mode=oracle，兼容 Oracle JDBC 接口时，会出现以下情况：

1. 会话上的 readOnly 属性设置无效。

Connection.setReadOnly() 接口忽略设置，Connection.isReadOnly() == false。

2. 会话关闭前做事务提交。

相关接口 Connection.close()，调用时会先执行 Connection.commit()。

3. statement 有多种执行接口，各执行接口不再有对 SQL 类型的限制。

Statement 的多种执行接口包括 execute、executeUpdate、executeQuery 等。当 checkExecType=1 时，这些接口对执行的 sql 语句类型有限制，executeUpdate 不能执行查询语句，executeQuery 不能执行更新语句等诸如此类的。当 checkExecType=0 时，兼容 Oracle 的处理，各执行接口不再有对 SQL 类型的限制。

4. DatabaseMetaData 类的 getFunctions / getProcedures /getFunctionColumns /getProcedureColumns 接口，可以搜索到包对象的内部存储过程和函数。

5. DatabaseMetaData.getPrimaryKeys() 返回结果中的列 pk_name 为索引名。

DM 本来返回的为约束名，Oracle 为索引名。

6. Url 格式串兼容 Oracle url 格式串前缀 "jdbc:oracle:@" / "jdbc:oracle:thin:@"。

例如：连接 DM 数据库时 url 可写为："jdbc:oracle:thin:@localhost:5236"

7. CLOB/BLOB 类的写入数据接口 (setXXX)，参数偏移允许从 0 开始。

DM 偏移从 1 开始，而 Oracle 从 0 开始，内部做兼容处理，允许偏移从 0 开始。

8. 对于所有的数值类型，ResultSetMetaData 和 ParameterMetaData 的类型描述都为 Types.NUMERIC，当调用 getObject 接口获取数据时返回的为 BigDecimal。

9. 从数据库中获取到 dec 类型的数据，不保留无效的末尾 0。

4.5.5.2 兼容 Mysql 的情况

当 JDBC 驱动端 compatible_mode=mysql，兼容 Mysql JDBC 接口时，会出现以下情况：

会话上设置事务隔离级别 Connection.TRANSACTION_REPEATABLE_READ 时，系统强制改为 Connection.TRANSACTION_READ_COMMITTED。DM 不支持隔离级 Connection.TRANSACTION_REPEATABLE_READ，不开兼容时会忽略设置使用默认隔离级。开启兼容时强制设为 Connection.TRANSACTION_READ_COMMITTED 级别。

4.5.6 获取执行计划

Connection 对象提供了获取执行计划的方法 getExplainInfo (SQL 语句)。

以下代码展示了如何获取并打印执行计划：

```
//建立连接
Class.forName("dm.jdbc.driver.DmDriver"); //加载驱动程序
String url = "jdbc:dm://223.254.254.19"; //连接机器 IP = 223.254.254.19
String userID = "SYSDBA";
String passwd = "SYSDBA";
Connection con = DriverManager.getConnection(url, userID, passwd);
//获取执行计划
String getExplan=con.getExplainInfo("select * from T1");
//打印执行计划
```

```
System.out.println(getExplan);
```

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”。

注意事项：

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 `DmdbPreparedStatement` 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
<code>setINTERVALYM</code>	设置参数为年-月时间间隔类型值
<code>setINTERVALDT</code>	设置参数为日-时时间间隔类型值
<code>setTIME</code>	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 dmmps = (DmdbPreparedStatement)pstmt;
DmdbIntervalYM dmiym = new DmdbIntervalYM("interval '20-10' year to month");
dmmps.setINTERVALYM(1, dmiym);
```

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. IN OUT 参数

如果参数为既接受输入又接受输出的参数类型（`IN OUT` 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。

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 DmDbCallableStatement 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

由于 CallableStatement 是 PreparedStatement 的子类，因此，它将继承 PreparedStatement 的所有的方法。

4.6.4 获取执行过程中服务器返回的打印消息

DM 的 Statement 对象提供了扩展的 getPrintMsg() 方法，可以获得语句执行过程中服务器返回的打印消息，继承自 Statement 的 PreparedStatement 和 CallableStatement 也提供此方法。

下面的例子展示了如何在语句执行过程中获取服务器返回的打印消息。

```
//建立连接
Class.forName("dm.jdbc.driver.DmDriver"); //加载驱动程序
DmdbConnection connection = (DmdbConnection)
DriverManager.getConnection("jdbc:dm://127.0.0.1:5236", "SYSDBA", "SYSDBA");

//执行语句
Statement stmt = connection.createStatement();
stmt.executeUpdate("begin dbms_output.enable(100000); end;");
DmdbCallableStatement cstmt = (DmdbCallableStatement)connection.prepareCall(
"begin\n"
+"dbms_output.put_line('使用 dbms_output 包打印日志');\n"
+"dbms_output.put_line('这是日志 1');\n"
+"dbms_output.put_line('这是日志 2');\n"
+ "end;\n"
);
cstmt.execute();

System.out.println(cstmt.getPrintMsg()); //获取语句执行时服务器返回的打印消息

stmt.execute("begin dbms_output.disable(); end;");
cstmt.close();
stmt.close();
connection.close();
```

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("COMMENT", 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间隔时间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
<code>getTime(int)</code>	根据列号(1开始)获取时间信息，以 <code>java.sql.Time</code> 类型返回
<code>getTime(int, Calendar)</code>	根据列号(1开始)、 <code>Calendar</code> 对象获取时间信息，以 <code>java.sql.Time</code> 类型返回
<code>getTime(String)</code>	根据列名获取时间信息，以 <code>java.sql.Time</code> 类型返回
<code>getTime(String, Calendar)</code>	根据列名、 <code>Calendar</code> 对象获取时间信息，以 <code>java.sql.Time</code> 类型返回
<code>getTimestamp(int)</code>	根据列号(1开始)获取时间信息，以 <code>java.sql.Timestamp</code> 类型返回
<code>getTimestamp(int, Calendar)</code>	根据列号(1开始)、 <code>Calendar</code> 对象获取时间信息，以 <code>java.sql.Timestamp</code> 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

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 databab SET data = ?");
ps.setBlob("DATA", data);
ps.executeUpdate();

```

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：

```
// 数据库产品的名称
String databaseName = dbmd.getDatabaseProductName();
// JDBC 驱动程序的主版本号, 如 3 表示 JDBC3.0
int majorVersion = dbmd.getJDBCMajorVersion();
// DM 提供的 JDBC 驱动程序版本号, 如 8.1.2.174
String driverVersion = dbmd.getDriverVersion();
// 获取表信息
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 `DmJdbcDriver16.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，故不支持大数据页；
- 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 支持 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

```

第一个方法使用时无需设置用户名和密码。需先使用 setUser 和 setPassword 给数据源设置好用户名和口令；然后再使用 getXAConnection 直接进行连接，并返回得到的分布式连接；

第二个方法使用时需设置用户名和口令。getXAConnection 使用参数中设置的用户名和口令进行连接，并返回得到的分布式连接。

3. 对象创建

XADataSource 继承自 DataSource，创建时可通过对象上的 set 相关接口指定连接的必要属性。示例如下：

```

DmdbXADataSource xads = new DmdbXADataSource();
xads.setURL("jdbc:dm://localhost:5236");
xads.setUser("SYSDBA");

```

```
xads.setPassword("SYSDBA");
```

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

XAConnection 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 用于指定事务分支的 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。

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 基本示例

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 DmdbXid(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();
}

```

4.12 如何使用 JDBC 接口操作空间数据

空间数据主要用来表示物体的位置、形态、大小和分布等信息，是对具有定位意义的物体和现象的定量描述。

DM 可从两方面来对空间数据进行操作：一通过 DMGEO 系统包；二通过 JDBC 接口。DMGEO 系统包的用法请参考《DM8 系统包使用手册》中 DMGEO 包，本节重点介绍如何通过 JDBC 接口操作空间数据。

4.12.1 空间数据类型

DMGEO 系统包中的的空间数据类型 ST_Geometry，在 DM 数据库中本质为 struct 类型，类型的读写按标准 struct 类型处理即可。空间数据类型的结构体中包含三个成员：srid、geo_wkb 和 geo_typeid。三个成员的更多详情介绍请参考《DM8 系统包使用手册》中 DMGEO 包。下面简要介绍一下三个成员的用途：

- srid 空间参考坐标系 ID，数据类型 INT。
- geo_wkb 二进制格式的几何体信息，数据类型 Blob。wkb 字段为标准 wkb 格式，可以使用通用 wkb 解析工具进行解析处理。jts 是 java 处理空间数据类型的通用组件，使用可参考 <https://github.com/locationtech/jts>。
- geo_typeid 几何体类型 ID，数据类型 INT。几何类型 ID 如下：

GEOS_Geometry	-1;
GEOS_Point	0;
GEOS_Linestring	1;
GEOS_Linearring	2;
GEOS_Polygon	3;
GEOS_MultiPoint	4;
GEOS_MultiLineString	5;
GEOS_MultiPolygon	6;
GEOS_GeometryCollection	7;

4.12.2 空间数据的读写步骤

如何使用 JDBC 接口操作空间数据，具体分为以下几个步骤：

一 先使用标准 JDBC 功能 getObject 从数据库中检索对象 java.sql.Struct。该 getObject 返回 java.lang.Object，因此必须将方法的输出强转为 struct。

示例如下：

```
ResultSet rs = stmt.executeQuery("select sub_mpoly from sub_types");
java.sql.Struct struct = (jav.sql.Struct)rs.getObject(1);
```

二 从 java.sql.Struct 对象中获取成员属性使用 getAttributes 方法。

示例如下：

```
Object[] attrs = struct.getAttributes(); // 获取结构体中的成员
int srid = (Integer)attrs[0]; // srid
Blob blob = (Blob)attrs[1]; // wkb
int type = (Integer)attrs[2]; // type
```

三 使用 jts 读取 wkb 解析为 org.locationtech.jts.geom.Geometry。

示例如下：

```
WKBReader reader = new WKBReader();
Geometry geometry = reader.read(blob.getBytes(1, (int)blob.length()));
```

四 将 org.locationtech.jts.geom.Geometry 对象转换为 wkb 数据。

示例如下：

```
WKBWriter writer = new WKBWriter();
byte[] bytes = writer.write(geometry);
```

五 封装空间数据类型为 java.sql.Struct 对象。

示例如下：

```
Object[] attrs = new Object[3];
attrs[0] = srid;
attrs[1] = DmdbBlob.newInstanceLocal(bytes, (DmdbConnection) conn);
attrs[2] = 6;
struct = conn.createStruct("ST_MULTIPOLYGON", attrs); //指定类型名称和结构体成员创建
```

六 将 java.sql.Struct 对象绑定到语句，使用标准的 setObject 方法。

示例如下：

```
PreparedStatement pstmt = conn.prepareStatement("insert into
sub_types(sub_mpolygon) values(?)");
pstmt.setObject(1, struct);
```

4.12.3 基本示例

下面用一个完整的示例，展示如何使用 JDBC 接口操作空间数据。

```
/*
 * geometry simple
 * create table sub_types (sub_mpolygon st_multipolygon );
 * insert into sub_types values (dmgeo.st_mpolyfromtext ('multipolygon(((10
10, 10 20, 20 20, 20 15, 10 10)), ((50 40, 50 50, 60 50, 60 40, 50 40)))', 4269));
*/
@SuppressWarnings ("unused")
public static void test_spatial() throws SQLException,
org.locationtech.jts.io.ParseException
{
    Connection conn = getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select sub_mpolygon from sub_types;");
    while (rs.next())
    {
        // from struct to geometry
        Struct struct = (Struct)rs.getObject(1);
        Object[] attrs = struct.getAttributes(); //获取结构体中的成员
        int srid = (Integer)attrs[0]; // srid
        Blob blob = (Blob)attrs[1]; // wkb
        int type = (Integer)attrs[2]; //type

        WKBReader reader = new WKBReader();
```

```
Geometry geometry = reader.read(blob.getBytes(1, (int)blob.length()));
System.out.println(geometry);
//from geometry to struct, the struct object can used for insert by
pstmt.setObject(struct)
WKBWriter writer = new WKBWriter();
attrs[1] = DmdbBlob.newInstanceOfLocal(writer.write(geometry),
(DmdbConnection)conn);
struct = conn.createStruct("ST_MULTIPOINT", attrs); //指定类型名
称和结构体成员创建
PreparedStatement pstmt = conn.prepareStatement("insert into
sub_types(sub_mpolygon) values(?)");
pstmt.setObject(1, struct);
pstmt.execute();
}
rs.close();
stmt.close();
conn.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`;

`RW_Separate`: 是否读写分离, 有效值为 `true` 或 `false`;

`RW_Percent`: 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 `false`;

`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。取值范围 0~3；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。缺省为 0
port	登录端口号
user	用户名
password	用户口令
timeout	连接超时时间，单位毫秒，缺省为 5000
commandTimeout	命令超时时间，单位秒，缺省为 30
appname	应用名
primary_key	需要加双引号的关键字
switch_time	主备切换的次数，缺省为 1
switch_interval	主备切换的时间间隔，单位毫秒，缺省为 1000
time_zone	时区，默认为当前时区
rw_separate	是否读写分离，缺省为 FALSE
rw_percent	读写分离百分比，缺省为 25
connPooling	是否使用连接缓存池，缺省为 FALSE
connPoolSize	连接池中最大连接数，缺省为 100，连接池开启时有效
connPoolCheck	是否检查连接池中连接的有效性，TRUE 表示检查， FALSE 表示不检查，缺省为 FALSE，连接池开启时有效
connPoolIdleClearInterval	回收连接池中过期连接的时间间隔，单位毫秒，缺省为 10000
connPoolIdleExpiredTime	连接池中连接的过期时间，单位毫秒，缺省为 0，表示永不过期
stmtPooling	是否启用句柄重用，缺省为 TRUE
poolSize	句柄重用缓冲区的大小，缺省为 100
logLevel	日志等级，缺省为 OFF，高级别同时记录低级别的信息 OFF：不记录； ERROR：记录错误日志； SQL：记录执行 SQL 信息； INFO：记录全部执行信息
logDir	生成日志路径，缺省为当前执行程序路径，日志文件名自动生成，格式如： DmProvider_2019_09_05_16_49_20.845.log
enRsCache	是否打开结果集缓存。缺省为 FALSE

rsCacheSize	缓存池大小，单位 MB。缺省为 10
rsRefreshFreq	缓存池中结果集更新的频率阈值，即大于该值时才检查命中的结果集是否需要更新，单位秒。缺省为 10
lobMode	结果集中大字段类型数据的缓存方式。1 表示本地只缓存一部分数据；2 表示本地缓存全部的数据。缺省为 1
batchType	绑定多行参数执行时，0 表示分多次执行，一次只执行一行参数；1 表示一次执行多行参数。缺省为 1
batchNotOnCall	执行 SQL 为存储过程时，是否允许一次执行多行参数。缺省为 FALSE
batchContinueOnError	绑定多行参数执行时，某行参数执行出错后，后续参数是否继续执行。缺省为 FALSE
batchAllowMaxErrors	绑定多行参数执行时允许的错误数最大值。缺省为 0
bufPrefetch	Fetch 获取的结果集内存大小，单位 MB。缺省为 0
compatibleMode	兼容模式。取值范围 0~2；0 表示不开启该功能；1 表示兼容 Oracle；2 表示兼容 Mysql。缺省为 0
isBdtaRS	消息中结果集是否以 Bdta 格式。缺省为 FALSE
maxRows	返回的结果集最大行数。缺省为 0x7fffffff
socketTimeout	网络包读写的超时时间值，单位秒。缺省为 0
addressRemap	服务器网络地址映射表。缺省值为空串。值格式： address_remap=192.168.1.24:5236,192.168.1.23:5236
userRemap	用户名映射表。缺省值为空串。格式同 addressRemap
epSelector	服务名配置的集群选取起始位置的方式。0 表示随机选取一个站点为起始位置；1 表示第一个站点就是起始位置。缺省值为 0
loginStatus	设置连接的服务器状态。取值范围 0、3~5；0 表示不开启该功能；4 表示 open 状态；3 表示 mount 状态；5 表示 suspend 状态。缺省为 0
ep_selection	当集群中存在故障节点时是否进行连接优化。0 表示进行优化，重排连接节点次序，故障节点位置靠后；1 表示不进行优化。缺省为 0
maxLobDataLenPerMsg	指定单条消息中大字段类型数据的最大长度，单位字节，缺省值为 32000，最小值为 1024 (1KB)，最大值为 104857600 (100MB)，超出范围则使用默认值
dbTimeTo TimeSpan	指定将数据库中的何种类型映射到.NET 的 TimeSpan 类型。TRUE 表示将数据库的 TIME 类型映射到.NET 的 TimeSpan 类型；FALSE 表示将数据库的 INTERVAL DAY(2) TO SECOND(6) 类型映射到.NET 的 TimeSpan 类型；缺省值为 FALSE

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；

`Name()`: 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`;

`SourceVersion`: 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` 复制到指定的数据组;

`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。

5.2.12 DmBulkCopy 对象

用于快速批量装载数据。实现 `IDisposable` 接口。该功能依赖 DM 安装目录\bin 下的 `dmfldr_dll.dll` 等动态链接库，需要拷贝到应用程序的执行目录。

公共属性

`DestinationTableName:` 指定目标表名；

`ColumnMappings:` 指定源数据列与目标表列的映射集合；

`BatchSize:` 指定一次发送的数据行数，缺省值为 100。

公共方法

`void WriteToServer(DataRow[] rows):` 装载数据；

`void WriteToServer(DataTable table):` 装载数据；

`void WriteToServer(DataTable table, DataRowState rowState):` 装载数据；

`void WriteToServer(DmDataReader reader):` 装载数据。

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 EF DmProvider 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\Dll\Provider.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\confi
```

g\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)
    {
    }
}
}
```

5.4 NHibernate 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\Dll 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 EF DmProvider.EF6 方言包

DM 基于 DmProvider 数据库驱动提供了方言包程序 EF DmProvider.EF6 以支持微软的 EF6 (EntityFrameWork) 框架。应用程序使用 EF6 连接到 DM 数据库时，不仅需要 DmProvider.dll 和 EF DmProvider.EF6.dll，同时配置文件（默认文件名是 App.config）需满足如下格式：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <!--EF6 配置类-->
        <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false"/>
        <!--EF DmProvider.EF6 配置类，提供一些兼容性配置。若无相关需求则可以注释-->
        <section name="EF Dm" type="EF DmProvider.config.EFDmConfigurationSection,
EF DmProvider.EF6, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=514fc861b4bfc6c6" requirePermission="false" />
    </configSections>

    <!--DmProvider 注册，参考驱动注册章节-->
    <system.data>
        <DbProviderFactories>
            <remove invariant="Dm"></remove>
            <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>
    </system.data>

    <!--EF6 配置项-->
    <entityFramework>
```

```

<providers>
    <!-- EFDbProvider.EF6 提供支持-->
    <provider invariantName="Dm" type="EFDbProvider.DmProviderServices,
EFDbProvider.EF6" />
</providers>
</entityFramework>

<!--EFDbProvider.EF6 配置项，若无相关需求，则可以注释-->
<EFDb>
    <!-- varcharMaxToClob 标签属性 value 为 true 表示 SqlServer varcharMax 类型
映射到 DM 的 Clob 类型； value 为 false 表示映射到 DM 的 varchar 类型。 默认为 true-->
    <varcharMaxToClob value = "true"/>
</EFDb>

<!-- 连接串配置-->
<connectionStrings>
    <add name="VOMEntities" providerName="Dm"
connectionString="Server=LOCALHOST;User=USER1;password=USER1;PORT=5236"/>
</connectionStrings>
<startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
</startup>
</configuration>

```

5.6 DM-EFCore 方言包

5.6.1 介绍

EFCore.Dm程序包是DM数据库提供的基于EFCore框架连接到DM数据库的方言包程序。

5.6.2 使用

EFCore.Dm方言包位于安装目录..\\drivers\\dotNet\\EFCore.Dm路径，该程序包依赖DmProvider程序和EFCore程序，依赖信息在EFCore.Dm路径下的Microsoft.EntityFrameworkCore.Dm.2.1.1.nuspec文件中可以看到，使用时直接将EFCore.Dm程序包及其依赖程序包添加到nuget程序包源后，然后在应用程序中添加nuget依赖即可。

5.6.3 示例

执行该示例前需要在数据库建表:

```

create table "Customers"("Id" varchar);

public class CustomerContext : DbContext
{
    public CustomerContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Customer>(b =>
        {
            b.HasKey(c => c.Id);
            b.ToTable("Customers");
        });
    }
}

public class Customer
{
    public string Id { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        try
        {
            var options = new DbContextOptionsBuilder()
                .UseDm("SERVER=localhost;PORT=5236;USER=SYSDBA;PASSWORD=SYSD
BA")
                .UseInternalServiceProvider(
                    new ServiceCollection()
                        .AddEntityFrameworkDm()
                        .BuildServiceProvider())
                .Options;
        }

        using (var context = new CustomerContext(options))
        {
            context.Database.EnsureCreated();
            var cus1 = new Customer { Id = "abc" };
            var cus2 = new Customer { Id = "qqq" };
            context.Customers.AddRange(cus1, cus2);
            context.SaveChanges();
        }
    }
}

```

```

var results = context.Customers.Where(c =>c.Id.StartsWith("a"))
    .OrderByDescending(c =>c.Id)
    .ToList();
Console.WriteLine(results[0].Id);
}
}
catch (Exception e)
{
Console.WriteLine("异常: " + e.Message);
}
Console.WriteLine("test over");
}
}

```

5.7 对象使用

5.7.1 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();

```

5.7.2 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：

```

var command = conn.CreateCommand();
command.CommandText = "SELECT NAME, AUTHOR, PUBLISHER
FROM PRODUCTION.PRODUCT;";
var myReader = command.ExecuteReader();

```

5.7.3 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()
        {
            var command = cnn.CreateCommand();
            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;";
        var 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();
```

5.7.4 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 param1 = new DmParameter(":PHOTO", DmDbType.Binary);
                command.Parameters.Add(param1);
                param1.Value = dataArray;
                command.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                ret = 0;
            }
        }
    }
}
```

5.7.5 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。

5.7.6 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()
        {
            var command = cnn.CreateCommand();
            try
            {
                command.CommandText = "PRODUCTION.UPDATEPRODUCT";
                command.CommandType = System.Data.CommandType.StoredProcedure;
            }
        }
    }
}

```

```
DmParameter parm1 = new DmParameter(":V_ID", DmDbType.Int32);
command.Parameters.Add(parm1);
parm1.Value = 1;
parm1.Direction = System.Data.ParameterDirection.Input;
DmParameter parm2 = new DmParameter(":V_NAME", DmDbType.VarChar);
command.Parameters.Add(parm2);
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;";
var 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（下）”。

5.8 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 =(DmDataReader)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;
}
}

}
```

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，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。

6.1.1 PHP 5.x 扩展函数

下面列出 DM 提供的 PHP 5.x 扩展函数，并简要说明每个函数的功能。

序号	函数类型	扩展库接口函数	作用描述	PHP 7.x
1	连接相关	dm_connect	打开一个到 DM 服务器的连接	支持，但参数略有不同
2		dm_pconnect	打开一个到 DM 服务器的持久连接	支持，但参数略有不同
3		dm_close	关闭 DM 连接	支持（PHP 5.x 返回 dmbool，PHP 7.x 返回 void）
4		dm_set_connect	设置连接	PHP 7.x 用 dm_setopt 代替
5		dm_get_server_info	取得 DM 服务器信息	支持
6	绑定参数	dm_prepare	准备一条语句	支持，但参数略有不同
7	执行	dm_execute	执行一条语句	支持，但功能和参数略有不同
8	查询	dm_query	发送一条 DM 查询，并可执行	PHP 7.x 用 dm_exec 代替（本扩展函数自 PHP 5.5.0 起已废弃）
9		dm_unbuffered_query	向 DM 发送一条 SQL 查询，并不获取和缓存结果的行	PHP 7.x 用 dm_exec 代替

10	结果集	dm_more_query_no_result	执行一条无结果集的语句	PHP 7.x 用 dm_exec 代替
11		dm_db_query	发送一条库的 DM 查询，并可执行，但库参数不起作用	PHP 7.x 用 dm_exec 代替
12		dm_result	取得一行结果数据	支持，但功能和参数略有不同
13		dm_more_result	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	PHP 7.x 用 dm_next_result 代替
14		dm_free_result	释放结果内存	支持
15		dm_num_rows	取得结果集中行的数目	支持
16		dm_num_fields	取得结果集中字段的数目	支持
17		dm_affected_rows	取得前一次 DM 操作所影响的记录行数	支持
18	Fetch 结果集	dm_data_seek	移动内部结果的游标	PHP 7.x 用 dm_fetch_row 代替
19		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	支持，但功能和参数略有不同
20		dm_fetch_object	从结果集中取得一行作为对象	支持，但参数略有不同
21		dm_fetch_row	从结果集中取得一行作为枚举数组	支持，但功能和参数略有不同
22		dm_fetch_assoc	从结果集中取得一行作为关联数组	PHP 7.x 用 dm_fetch_row 代替
23		dm_fetch_field	从结果集中取得列信息并作为对象返回	支持
24	表元数据	dm_fetch_length	取得结果集中每个输出的长度	PHP 7.x 用 dm_fetch_array 代替
25		dm_list_fields	列出 DM 结果中的字段	支持
26		dm_list_tables	列出 DM 数据库中的表	PHP 7.x 用 dm_tables 代替
27	列元数据	dm_tablename	取得表名	支持
28		dm_field_flags	从结果中取得和指定字段关联的标志	支持
29		dm_field_len	返回指定字段的长度	支持
30		dm_field_name	取得结果中指定字段的字段名	支持
31		dm_field_seek	将结果集中的指针设定为制定的字段偏移量	PHP 7.x 用 dm_fetch_field 代替
32		dm_field_table	取得指定字段所在的表名	支持

33		<code>dm_field_type</code>	取得结果集中指定字段的类型	支持
34		<code>dm_insert_id</code>	取得上一步 INSERT 操作产生的 ID	支持
35	错误	<code>dm_error</code>	返回上一个 DM 操作产生的文本错误信息	PHP 7.x 用 <code>dm_errormsg</code> 代替
36		<code>dm_errno</code>	返回上一个 DM 操作中的错误信息的数字编码	PHP 7.x 用 <code>dm_error</code> 代替
37	事务	<code>dm_abort</code>	回滚一个事务	PHP 7.x 用 <code>dm_rollback</code> 代替
38		<code>dm_commit</code>	提交一个事务	支持
39		<code>dm_autocommit</code>	设置自动提交功能	支持
40		<code>dm_begin_trans</code>	开始一个事务	PHP 7.x 用 <code>dm_autocommit</code> 代替
41	其他	<code>dm_set_object_name_case</code>	设置比较时的大小写方式	支持
42		<code>dm_escape_string</code>	转义一个字符串用于 <code>dm_query</code>	支持
43		<code>dm_ping</code>	Ping 一个服务器连接，如果没有连接则重新连接	支持

6.1.2 PHP 7.x 扩展函数

下面列出 DM 提供的 PHP 7.x 扩展函数，并简要说明每个函数的功能。

序号	函数类型	扩展库接口函数	作用描述	PHP 5.x
1	兼容 PHP 5.x 的函数	<code>dm_num_fields</code>	取得结果集中字段的数目	支持
2		<code>dm_field_len/</code> 别名 <code>dm_field_precision</code>	返回指定字段的长度	支持
3		<code>dm_field_name</code>	取得结果中指定字段的字段名	支持
4		<code>dm_field_type</code>	取得结果集中指定字段的类型	支持
5		<code>dm_free_result</code>	释放结果内存	支持
6		<code>dm_num_rows</code>	取得结果集中行的数目	支持
7		<code>dm_commit</code>	提交一个事务	支持
8		<code>dm_fetch_field</code>	从结果集中取得列信息并作为对象返回	支持
9		<code>dm_tablename</code>	取得表名	支持
10		<code>dm_field_table</code>	取得指定字段所在的表名	支持
11		<code>dm_field_flags</code>	从结果中取得和指定字段关联的标志	支持

12	与 PHP 5.x 同名，但功能或参数略有不同	dm_get_server_info	取得 DM 服务器信息	支持
13		dm_escape_string	转义一个字符串用于 dm_query	支持
14		dm_ping	Ping 一个服务器连接，如果没有连接则重新连接	支持
15		dm_set_object_name_case	设置比较时的大小写方式	支持
16		dm_prepare	准备一条语句	支持，参数略有不同
17		dm_execute	执行一条准备过的语句	支持，功能和参数略有不同
18		dm_fetch_array	指定行号从结果集中取得一行作为关联数组	支持，功能和参数略有不同
19		dm_fetch_row	从结果集中取得一行数据	支持，功能和参数略有不同
20		dm_fetch_object	从结果集中取得一行作为对象	支持，参数略有不同
21		dm_result	取得结果数据	支持，功能和参数略有不同
22		dm_connect	打开一个到 DM 服务器的连接	支持，功能和参数略有不同
23		dm_pconnect	打开一个到 DM 服务器的持久连接	支持，功能和参数略有不同
24		dm_error	返回上一个 DM 操作中的错误信息的数字编码	功能等价于 dm_errno (PHP 5.x)
25	新增函数	dm_close	关闭 DM 连接	支持，参数略有不同
26		dm_affected_rows	取得前一次 DM 操作所影响的记录行数	支持，参数略有不同
27		dm_insert_id	取得上一步 INSERT 操作产生的 ID	支持，参数略有不同
28		dm_list_fields	列出 DM 结果中的字段	支持，参数略有不同
29		dm_binmode	是否读取二进制类型	
30		dm_close_all	关闭所有 DM 连接	
31		dm_columns	获取指定表的所有列信息	
32		dm_autocommit	设置自动提交功能	
33		dm_cursor	获取游标信息	
34		dm_errormsg	返回上一个 DM 操作产生的文本错误信息	功能等价于 dm_error (PHP 5.x)
35		dm_exec/别名 dm_do	准备并执行 sql 语句	功能等价于 dm_execute (PHP 5.x)
36		dm_fetch_into	从结果集中取得一行作为一个数组	
37		dm_field_scale	取得结果中指定字段的标度	
38		dm_field_num	取得结果中字段的编号	
39		dm_longreadlen	设置变长类型，读取的最大长	

			度	
40		dm_next_result	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	
41		dm_result_all	获取全部结果集，并打印成 html 格式	
42		dm_rollback	回滚	功能等价于 dm_abort (PHP 5.x)
43		dm_setopt	调整语句和连接的属性配置	功能等价于 dm_set_connect (PHP 5.x)
44		dm_specialcolumns	获取特殊列	
45		dm_statistics	获取表的统计信息	
46		dm_tables	获取指定模式所有表信息	功能等价于 dm_list_tables (PHP 5.x)
47		dm_primarykeys	获取表的主键	
48		dm_columnprivileges	列的权限	
49		dm_tableprivileges	表的权限	
50		dm_foreignkeys	获取表的外键	
51		dm_procedures	获取所有过程名	
52		dm_procedurecolumns	获取所有过程的参数名	

6.1.3 PHP 8.x 扩展函数

下面列出 DM 提供的 PHP 8.x 扩展函数，并简要说明每个函数的功能。

序号	函数类型	扩展库接口函数	作用描述	PHP 7.x
1	兼容 PHP 7.x 的函数	dm_close	关闭 DM 连接	支持
2		dm_num_fields	取得结果集中字段的数目	支持
3		dm_field_len/ 别名 dm_field_precision	返回指定字段的长度	支持
4		dm_field_name	取得结果中指定字段的字段名	支持
5		dm_field_type	取得结果集中指定字段的类型	支持
6		dm_free_result	释放结果内存	支持
7		dm_num_rows	取得结果集中行的数目	支持
8		dm_commit	提交一个事务	支持
9		dm_fetch_field	从结果集中取得列信息并作为对象返回	支持
10		dm_affected_rows	取得前一次 DM 操作所影响的记录行数	支持

11	dm_insert_id	取得上一步 INSERT 操作产生的 ID	支持
12	dm_list_fields	列出 DM 结果中的字段	支持
13	dm_tablename	取得表名	支持
14	dm_field_table	取得指定字段所在的表名	支持
15	dm_field_flags	从结果中取得和指定字段关联的标志	支持
16	dm_get_server_info	取得 DM 服务器信息	支持
17	dm_escape_string	转义一个字符串用于 dm_query	支持
18	dm_ping	Ping 一个服务器连接, 如果没有连接则重新连接	支持
19	dm_set_object_name_case	设置比较时的大小写方式	支持
20	dm_prepare	准备一条语句	支持
21	dm_execute	执行一条准备过的语句	支持
22	dm_fetch_array	指定行号从结果集中取得一行作为关联数组	支持
23	dm_fetch_row	从结果集中取得一行数据	支持
24	dm_fetch_object	从结果集中取得一行作为对象	支持
25	dm_result	取得结果数据	支持
26	dm_connect	打开一个到 DM 服务器的连接	支持, 但功能略有不同
27	dm_pconnect	打开一个到 DM 服务器的持久连接	支持
28	dm_error	返回上一个 DM 操作中的错误信息的数字编码	支持
29	dm_binmode	是否读取二进制类型	支持
30	dm_close_all	关闭所有 DM 连接	支持
31	dm_autocommit	设置自动提交功能	支持
32	dm_cursor	获取游标信息	支持
33	dm_errormsg	返回上一个 DM 操作产生的文本错误信息	支持
34	dm_exec/别名 dm_do	准备并执行 sql 语句	支持
35	dm_fetch_into	从结果集中取得一行作为一个数组	支持
36	dm_field_scale	取得结果中指定字段的标度	支持
37	dm_field_num	取得结果中字段的编号	支持
38	dm_longreadlen	设置变长类型, 读取的最大长度	支持
39	dm_next_result	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	支持
40	dm_result_all	获取全部结果集, 并打印成	支持

			html 格式	
41		dm_rollback	回滚	支持
42		dm_setopt	调整语句和连接的属性配置	支持
43		dm_specialcolumns	获取特殊列	支持
44	与 PHP 7.x 同 名，但参 数略有不 同	dm_columns	获取指定表的所有列信息	支持，但参数取值略有不 同
45		dm_statistics	获取表的统计信息	支持，但参数取值略有不 同
46		dm_tables	获取指定模式所有表信息	支持，但参数取值略有不 同
47		dm_primarykeys	获取表的主键	支持，但参数取值略有不 同
48		dm_columnprivileges	列的权限	支持，但参数取值略有不 同
49		dm_tableprivileges	表的权限	支持，但参数取值略有不 同
50		dm_foreignkeys	获取表的外键	支持，但参数取值略有不 同
51		dm_procedures	获取所有过程名	支持，但参数取值略有不 同
52		dm_procedurecolumns	获取所有过程的参数名	支持，但参数取值略有不 同

6.2 DM PHP 模块加载

目前 DM 支持的 PHP 版本为 PHP 5.2、5.3、5.4、5.5、5.6 和 PHP 7.0、7.1、7.2、7.3、7.4，凝思操作系统只支持到 PHP 7.2，用户可根据自己安装的 PHP 版本选择对应的接口库。

PHP 从 PHP 5.1 版本开始附带了 PHP 数据对象（PHP Data Object，简称 PDO），PDO 扩展为 PHP 访问数据库定义了一个轻量级的一致性接口。利用 PDO 扩展，用户可使用相同的方法来查询获取不同数据库的数据。需要注意的是，PDO 扩展必须配合数据库的 PDO 驱动一起使用，才能成功访问相应的数据库。

下面的示例均以 PHP 5.3 版本为例，如果版本不同，将 53 改为对应值即可。

6.2.1 linux 系统下 PHP 加载

linux 系统下的 PHP 加载分为两种方式：普通源码安装方式和 yum 安装方式。

普通源码安装下载的是源码包，编译安装过程可以设定参数，按照需求进行安装，并且安装的版本可以选择，灵活性比较大。

yum 安装下载的是 rpm 软件包，安装 rpm 包。这个 rpm 包是别人编译安装好的二进制包，安装方便快捷，但是安装过程无法干预，不能按需安装。因此，安装之后，要根据实际情况再进行局部功能调整。

linux 系统下 PDO 接口依赖的动态链接库如下：libcrypto.so、libdmdpi.so、

libdmmem.so、libdmul.so、libssl.so、libdmclientlex.so、libdmelog.so、libdmos.so。上述动态链接库在 DM 安装目录的/bin 目录下。

6.2.1.1 普通源码安装

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 -C/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 -5.3.tar.gz。存至/home/tmp。

```
cd /home/tmp
tar -xvzf php-x.x.x.tar.gz -C/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) 配置 DM PHP

修改 php.ini，添加 extension_dir=drivers/php_pdo，extension=libphp53_dm.so，添加 php.ini 中有关连接的配置。设置环境变量 export LD_LIBRARY_PATH=/usr/local/DMDBMS/bin，或者设置为依赖文件 libdmdpi.so 存放的路径。

6.2.1.2 yum 方式安装

步骤:

1) 安装 epel

```
yum install epel-release
rpm -Uvh https://mirror.webtatic.com/yum/el7/webtatic-release.rpm
```

2) 安装 PHP

```
yum install php72w
```

3) 安装必需扩展和 pdo 扩展

```
yum install php72w-cli
```

```
yum install php72w-pdo
```

4) 查看版本

```
php -v
```

```
PHP 7.2.27 (cli) (built: Jan 26 2020 15:49:49) ( NTS )
```

```
Copyright (c) 1997-2018 The PHP Group
```

```
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
```

5) 修改配置文件/etc/php.ini

配置 PDO 的数据库扩展，需要 PDO 的支持。

因为 PDO 必须在具体的数据库扩展被载入前初始化，而 configufge/make&makeinstall 配置编译安装会默认启用自带的 PDO 模块，yum 共享模块安装不会启用，所以要在配置具体的 dm 数据库 pdo 扩展前配置 pdo.so。

```
extension=pdo.so
```

```
extension=php72_pdo_dm.so
```

```
extension=libphp53_dm.so
```

6) 拷贝驱动至目录 /usr/lib64/php/modules

7) 设置 LD_LIBRARY_PATH

```
export
```

```
LD_LIBRARY_PATH=/home/ATS/smoke_47570/dm7dev2/build/linux/linux_build/debug
```

8) 再次查看版本

```
[root@192 linux_build]# php -v
```

```
PHP 7.2.27 (cli) (built: Jan 26 2020 15:49:49) ( NTS )
```

```
Copyright (c) 1997-2018 The PHP Group
```

```
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
```

6.2.2 Windows 系统下加载 PHP 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。

- 4) 配置 DM PHP。拷贝 drivers\php_pdo 目录下 php53_dm.dll 到 php 目录下的 ext 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；拷贝 bin 目录下依赖的库文件 dmdpi.dll 到 php 目录下的 ext 目录中。php53_dm.dll 中 PDO 类和 PDOSTatement 类支持的接口和 libphp53_dm.so 一样。

- 5) 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。

Windows 系统下 PDO 接口依赖的动态链接库如下：dmdpi.dll、dmclientlex.dll、dmutil.dll、dmstrt.dll、dmshm.dll、dmos.dll、dmmsg.dll、dmout.dll、

dmmem.dll、dmeelog.dll、dmdata.dll、dmddcr.dll、dmcyt.dll、dmcvt.dll、dmcpri.dll、dmcomm.dll、dmcfg.dll、dmcalc.dll。上述动态链接库在 DM 安装目录的\bin 目录下。

6.2.3 PHP INI 文件和 libphp53_dm.so 类介绍

php.ini 中可以配置的参数如下表所示：

参数	说明	配置示例
dm.default_host	DM 连接默认 ip 和 port	dm.default_host =192.168.0.25:6237
dm.default_user	DM 连接默认用户名	dm.default_user =SYSDBA
dm.default_pw	DM 连接默认密码	dm.default_pw =SYSDBA
extension_dir	DM 依赖库路径	extension_dir=D:\php-7.1.32-nts-Win32-VC14-x64\ext
extension	DM 依赖库名称	extension=php71nts_dm.dll
dm.defaultlrl	Clob 类型读取的默认长度，单位为 BYTE，取值范围 1~2147483648，即 1~2GB，缺省为 4096	dm.defaultlrl = 32767
dm.defaultbinmode	二进制数据处理方式，0：省略；1：按照实际返回；2：转换到字符。缺省为 1	dm.defaultbinmode = 1
dm.connect_timeout	数据库驱动与服务器建立 TCP 连接的超时时间，单位 s。缺省为 0	dm.connect_timeout = 10
dm.max_links	最大连接数，缺省为 -1，即最大连接数无限制	dm.max_links = 3
dm.max_persistent	持久连接的最大数，缺省为 -1，即最大持久连接数无限制	dm.max_persistent = 3
dm.allow_persistent	允许或禁止持久连接，1：允许持久连接；0：禁止	dm.allow_persistent = 1

	持久连接。缺省为 1	
dm.check_persistent	在重用前检查连接是否还可用，1：检查；0：不检查。缺省为 1	dm.check_persistent = 1

libphp53_dm.so 中 PDO 类和 PDOStatement 类共支持 34 个接口：

类名	序号	接口名	释义
PDO 类	1	PDO::beginTransaction	启动一个事务
	2	PDO::commit	提交一个事务
	3	PDO::__construct	创建一个表示数据库连接的 PDO 实例
	4	PDO::errorCode	获取跟数据库句柄上一次操作相关的 SQLSTATE
	5	PDO::errorInfo	获取最后一次操作数据库句柄的扩展错误信息
	6	PDO::exec	执行一条 SQL 语句，并返回受影响的行数
	7	PDO::getAttribute	取回一个数据库连接的属性
	8	PDO::getAvailableDrivers	返回一个可用驱动的数组
	9	PDO::inTransaction	检查是否在一个事务内
	10	PDO::lastInsertId	返回最后插入行的 ID
	11	PDO::prepare	准备要执行的语句，并返回语句对象
	12	PDO::query	执行 SQL 语句，以 PDOStatement 对象形式返回结果集
	13	PDO::quote	为 SQL 查询里的字符串添加引号
	14	PDO::rollBack	回滚一个事务
	15	PDO::setAttribute	设置属性
PDOStatement 类	16	PDOStatement::bindColumn	绑定一列到一个 PHP 变量
	17	PDOStatement::bindParam	绑定一个参数到指定的变量名
	18	PDOStatement::bindValue	把一个值绑定到一个参数
	19	PDOStatement::closeCursor	关闭游标，使语句能再次被执行。
	20	PDOStatement::columnCount	返回结果集中的列数
	21	PDOStatement::debugDumpParams	打印一条 SQL 预处理命令
	22	PDOStatement::errorCode	获取跟上一次语句句柄操作相关的 SQLSTATE
	23	PDOStatement::errorInfo	获取跟上一次语句句柄操作相关的扩展错误信息

24	PDOStatement::execute	执行一条预处理语句
25	PDOStatement::fetch	从结果集中获取下一行
26	PDOStatement::fetchAll	返回一个包含结果集中所有行的数组
27	PDOStatement::fetchColumn	从结果集中的下一行返回单独的一列。
28	PDOStatement::fetchObject	获取下一行并作为一个对象返回。
29	PDOStatement::getAttribute	检索一个语句属性
30	PDOStatement::getColumnMeta	返回结果集中一列的元数据
31	PDOStatement::nextRowset	在一个多行集语句句柄中推进到下一个行集
32	PDOStatement::rowCount	返回受上一个 SQL 语句影响的行数
33	PDOStatement::setAttribute	设置一个语句属性
34	PDOStatement::setFetchMode	为语句设置默认的获取模式

6.3 编程接口

6.3.1 PHP 5.x 接口

1. 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 [, bool $new_link[, bool $client_flags]]]]])
```

参数

参数	描述
server	[IN] 服务器名称，缺省使用 php.ini 中的配置
username	[IN] 用户名称，缺省使用 php.ini 中的配置
password	[IN] 用户密码，缺省使用 php.ini 中的配置
new_link	[IN] 设为 0 或不设时，用同样的参数第二次调用 dm_connect 将不会建立新连接，而将返回已经打开的连接标识；设为 1 时总是返回新的连接标识
client_flags	保留参数，不起作用

返回值

如果成功则返回一个连接标识，失败则返回 FALSE。

举例说明

例

```
dm_connect("192.168.0.25", "SYSDBA", "SYSDBA", 1);
```

2. dm_pconnect

描述

打开一个到 DM 服务器的持久连接。

格式

```
resource dm_pconnect ([string $server [, string $username [, string $password[,  
bool $client_flag1[, bool $client_flag2]]]]])
```

参数

参数	描述
server	[IN] 服务器名称，缺省使用 php.ini 中的配置
username	[IN] 用户名称，缺省使用 php.ini 中的配置
password	[IN] 用户密码，缺省使用 php.ini 中的配置
client_flag1	保留参数，不起作用
client_flag2	保留参数，不起作用

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。

举例说明

例

```
dm_pconnect("192.168.0.25", "SYSDBA", "SYSDBA");
```

3. 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。

4. dm_set_connect

描述

设置连接。

格式

```
int dm_set_connect (int $attr, int $value, resource $link_identifier)
```

参数

参数	描述	属性值
attr	设置的属性，同 dpi	DSQL_ATTR_ACCESS_MODE 101
		DSQL_ATTR_AUTOCOMMIT 102
		DSQL_ATTR_CONNECTION_TIMEOUT 113
		DSQL_ATTR_LOGIN_TIMEOUT 103
		DSQL_ATTR_TXN_ISOLATION 108
		DSQL_ATTR_LOCAL_CODE 12345
value	设置的值	属性值。各属性 ID 对应的属性值请参考 2.2.2 节中对应属性的介绍

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_more_query_no_result**描述**

执行多条无结果集的语句。

格式

```
int dm_more_query_no_result(string $sql, resource $link_identifier, int $flag)
```

参数

参数	描述
sql	要执行的语句
link_identifier	[IN] 连接标识符
flag	出错是否返回

返回值

如果成功则返回 TRUE，失败则返回 FALSE。

10. 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。

11. dm_affected_rows**描述**

取得前一次数据库操作 INSERT、UPDATE 或 DELETE 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

对于非 dml 语句 (ddl 语句)，返回 -1。

举例说明

例 1 当前一次数据库操作为 INSERT、UPDATE 或 DELETE 时返回该操作所影响的记录行数。

```
$ret = dm_query("delete from t1");
$ret = dm_affected_rows($link); //返回 delete 操作实际影响的记录行数
```

例 2 dm_more_query_no_result 和 dm_unbuffered_query 都不缓存影响行数，因此返回值为-1。

```
dm_more_query_no_result("insert into t1 values(1); insert into t1
values(1);insert into t1 values(1);");
$ret = dm_affected_rows($link); //返回-1
```

例 3 当结果集无法获取时返回值为-1。

```
$ret = dm_query("begin SELECT * from t; SELECT * from t; end;");
$ret1 = dm_more_result($ret); //结果集无法获取
$ret2 = dm_affected_rows($link); //返回-1
```

例 4 begin end 均返回 1。

```
$ret = dm_query("begin delete from t; delete from t; end;");
$ret1 = dm_more_result($ret);
$ret2 = dm_affected_rows($link); //返回 1
```

12. dm_escape_string

描述

转义一个字符串用于 dm_query。转义单引号为' 和`。

格式

```
string dm_escape_string (string $unesaped_string)
```

参数

参数	描述
unesaped_string	[IN] 被转义字符串

返回值

返回转义以后的字符串。

13. 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：1 (DM_ASSOC)，2 (DM_NUM) 和 0 (DM_BOTH)，如果设为 0，即 DM_BOTH，将得到一个同时包含关联和数字索引的数组。如果设为 1，即 DM_ASSOC 只得到关联索引，如果设为 2，即 DM_NUM 只得到数字索引，数字索引从 0 开始

返回值

返回从结果集取得的行生成的数组，默认双索引。如果没有更多行则返回 FALSE。

举例说明

例

```

$ret = dm_query("SELECT * from t;");
$result1 = dm_fetch_array($ret,1);
if($result1){
echo $result1["C1"];
echo $result1["C2"];
echo 'success';
}
//或者
$ret = dm_query("SELECT * from t;");
$result1 = dm_fetch_array($ret,2);
if($result1){
echo $result1[0];
echo $result1[1];
echo 'success';
}

```

14. 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] 结果集资源

返回值

返回根据从结果集取得的行生成的关联数组，列名为索引。如果没有更多行则返回 FALSE，可以获取全部结果集。

举例说明**例**

```

$ret = dm_query("SELECT * from t;");
$result1 = dm_fetch_assoc($ret);
if($result1){
echo $result1["C1"];
echo $result1["C2"];
echo 'success';
}

```

15. 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，从 0 开始

返回值

返回一个包含字段信息的对象。

举例说明**例**

```
$ret = dm_query("SELECT 1, 2 as c2, 3 as c3");
    $col = dm_fetch_field($ret);
    if($col) {
        if($col->name != "1" || $col->table != "" || $col->auto_uniq != "-1"
        || $col->max_length != "10" || $col->numeric != "1" ||
$col->blob != "-1"
        || $col->type != "INTEGER" || $col->unsigned != "-1") {
...
}
}
```

16. 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。

17. 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

以数组返回取得的行中每个字段的长度，如果出错返回 FALSE。

18. 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，列名为索引。如果没有更多行则返回 FALSE。

举例说明

例

```
$result1 = dm_fetch_object($ret);
if($result1) {
    echo $result1->c1;
    echo $result1->c2;
}
```

19. 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，列序号为索引，从 0 开始。如果没有更多行则返回 FALSE。

举例说明

例

```
$ret = dm_query("SELECT 1 as c1, 2 as c2, 3 as c3");
$result1 = dm_fetch_row($ret);
if($result1[0] != "1" || $result1[1] != "2" || $result1[2] != "3") {...}
```

20. 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，从 0 开始

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。

举例说明

例

```
$ret = dm_exec($link,"SELECT * from t1;");
$col = dm_field_flags($ret, 0);
```

21. dm_field_len

描述

返回指定字段的固定长度（精度）。

格式

```
int dm_field_len (resource $result, int $field_offset)
```

参数

参数	描述
result	[IN]结果集资源
field_offset	结果集中列的偏移，从 0 开始

返回值

返回指定字段的固定长度。

22. 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，从 0 开始

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。

23. dm_field_seek**描述**

将结果集中的指针设定为指定的字段偏移量。

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，从 0 开始

返回值

返回指定字段的上一个字段偏移量。

举例说明**例**

```
$ret = dm_query("SELECT 1 as c1, 2 as c2, 3 as c3");
dm_field_seek($ret, 0) ;
{
    $col = dm_fetch_field($ret);
    if($col->name == "c1"){
        $result="success";
    }
}
```

24. 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。

25. 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。

26. 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。

27. 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。

28. 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[,string $owner_name r[,string $field_name]]])
```

参数

参数	描述
database_name	[IN] 数据库名
table_name	[IN] 表名
link_identifier	[IN] 连接标识符
owner_name	模式名
field_name	指定列名

返回值

返回一个结果指针。

举例说明**例**

```
$ret = dm_list_fields("SYSTEM", "T1");
$col = dm_fetch_field($ret);
while($col)
{
    if($col->name == "CT1"){
        $flag= TRUE;
        break;
    }
    else{
        $result="fail";
    }
    $col = dm_fetch_field($ret);
}
```

29. dm_list_tables**描述**

列出 DM 数据库中的表。

格式

```
resource dm_list_tables (string $database_name [, resource
$link_identifier[, string $owner [, string $name ]]])
```

参数

参数	描述
database_name	[IN] 数据库名, 忽略
link_identifier	[IN] 连接标识符
owner	模式名

返回值

返回一个结果指针。

30. 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。

31. 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。

32. dm_more_result**描述**

取得下一个结果集。释放当前结果集。

格式

```
mixed dm_more_result (resource $result)
```

参数

参数	描述
result	[IN] 结果集

返回值

恒为 TRUE。

举例说明**例**

```
// 错误用法:
$ret = dm_query("begin SELECT * from t;delete * from t; end;");
$ret1=dm_more_result($ret);

// 正确用法:
$ret = dm_query("begin SELECT * from t; SELECT * from t;end;");
$ret1=dm_more_result($ret);
```

33. dm_result**描述**

取得一行结果数据。

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 (从 0 开始)
field	[IN] 指定列索引 (从 0 开始) 或列名称

返回值

返回 DM 结果集中一个单元 (第 row +1 行 第 field+1 列) 的内容。Row 也可以为字符串 ((单引号或双引号的数字)，若为无效字符串时，默认是第一行。列名称大小写不

区分。第三个参数为可选参数，默认第一列。

举例说明

例

```
SQL> select * from member;
//查询结果如下:
行号      member_name C0
-----
1          小明        NULL
2          小红        NULL
3          小日        100
$ret = dm_query("SELECT * from member");
$result = dm_result($ret, '2', 'C0');
```

34. 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。

35. 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	[IN] 表索引序号，从 0 开始

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。

举例说明

例

```
$ret = dm_list_tables("SYSTEM", $link, "%", "T1");
$row = 0;
$name = dm_tablename($ret, $row);
while($name) {
    if($name == "T1") {
        .....
    }
}
```

```

$row = $row + 1;
$name = dm_tablename($ret, $row);
}

```

36. 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	[IN] 表索引序号, 从 0 开始

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。

37. 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 $link_identifier, int $case_sensitive)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
case_sensitive	大小写方式。1: 小写; 2: 大写; 0: 保持大小写

返回值

设置成功, 返回 TRUE; 否则返回 FALSE。

38. dm_prepare**描述**

准备一条语句

格式

```
resource dm_prepare (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

在准备成功时返回一个资源标识符, 出错时返回 FALSE。

39. dm_commit**描述**

提交一个事务。

格式

```
bool dm_commit (resource $link_identifier)
```

参数

参数	描述
link_identifier	[IN] 连接标识符

返回值

成功返回 TRUE，失败返回 FALSE。

40. dm_execute**描述**

执行一条语句。(包括准备和执行，可以带绑定参数)

格式

```
resource dm_execute (resource $link_identifier, string $query[,  
array $parameters_array])
```

参数

参数	描述
link_identifier	[IN]连接标识符
query	[IN]查询字符串
parameters_array	参数组成的数组

返回值

在执行成功时返回一个资源标识符，出错时返回 FALSE。

举例说明**例**

```
$a = 1;  
  
$query = "INSERT INTO t2(c0) VALUES(?)";  
  
$result = dm_execute($link, $query, array($a));  
  
$query = "INSERT INTO t2(c0) VALUES(10)";  
  
$result = dm_execute($link, $query ) or die("Query failed : " . dm_error());
```

41. dm_abort**描述**

回滚一个事务。

格式

```
bool dm_abort (resource $link_identifier)
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

成功返回 TRUE，失败返回 FALSE。

42. dm_begin_trans**描述**

设置事务为不自动提交。

格式

```
bool dm_begin_trans (resource $link_identifier)
```

参数

参数	描述
link_identifier	[IN]连接标识符

返回值

成功返回 TRUE，失败返回 FALSE。

43. dm_get_version**描述**

获得服务器版本号。

格式

```
char* dm_get_version (resource $link_identifier)
```

参数

参数	描述
link_identifier	[IN] 连接标识符

返回值

返回 dmversion 字符串。

6.3.2 PHP 7.x 接口

6.3.2.1 和 PHP 5.x 同名接口

下面是 PHP 7.x 中和 PHP 5.x 同名的接口，但是功能或参数等和 PHP 5.x 略有不同。

1. dm_prepare

描述

准备一条语句。PHP 7.x dm_prepare 不支持默认连接，除此之外其他用法同 dm_errno (PHP 5.x)。

格式

```
resource dm_prepare (resource $link_identifier, string $query)
```

参数

参数	描述
query	[IN] 查询字符串
link_identifier	[IN] 连接标识符，不支持缺省默认连接，不可缺省

返回值

在准备成功时返回一个资源标识符，出错时返回 FALSE。

2. dm_execute

描述

执行一条准备过的语句（可以带有绑定参数）。

格式

```
resource dm_execute (resource $result , array $parameters_array )
```

参数

参数	描述
result	准备过的资源标识符
parameters_array	[IN] 参数数组

返回值

成功时返回 TRUE，或者在失败时返回 FALSE。

举例说明

例

```
$a = 1;
$b = 2;
```

```
$c = 3;
$stmt = dm_prepare($link, 'insert into t values(?, ?, ?)');
$result = dm_execute($stmt, array($a, $b, $c));
```

3. dm_fetch_array

描述

指定行号从结果集中取得一行作为关联数组。

格式

```
array dm_fetch_array (resource $result [,int $rownum])
```

参数

参数	描述
result	[IN] 结果集资源
rownum	[IN] 取得的行号。可选参数，缺省从 1 开始

返回值

返回根据指定行号从结果集取得生成的数组，列名为索引。如果没有更多行则返回 FALSE。如果 rownum=0 或缺省，则获取游标的下一行。可代替 PHP 5.x 中的 dm_fetch_array 以及 dm_fetch_length。

举例说明

例

```
//PHP 5.x 中利用 dm_fetch_lengths 获取结果集中所有列长度数组
$ret = dm_exec($link, "SELECT * from T1");
$collens = dm_fetch_lengths($ret);

//PHP 7.x 中利用 dm_fetch_array 获取结果集中的列长度
$ret = dm_exec($link, "SELECT * from T1");
$colval= dm_fetch_array($ret);
//建议直接查看$colval["列名"]对象的实际长度以获取该列长度
```

4. dm_fetch_row

描述

从结果集中取得一行数据。

格式

```
bool dm_fetch_row (resource $result [,int $rownum])
```

参数

参数	描述
result	[IN] 结果集资源
rownum	[IN] 取得的行号。可选参数，缺省从 1 开始

返回值

成功获取到数据返回 TRUE，如果没有更多行则返回 FALSE。可代替 PHP 5.x 中的 dm_fetch_row、dm_data_seek 以及 dm_fetch_assoc。

举例说明

例

```
//PHP 5.x 中利用 dm_data_seek 取得第三行结果集
$ret = dm_query("SELECT * from t1");
dm_data_seek($ret, 2);
$result = dm_fetch_object($ret);
```

```
//PHP 7.x 中利用 dm_fetch_row 取得第三行结果集
dm_fetch_row($ret, 3);
//PHP 7.x 访问结果集的第一列
$val = dm_result($ret, 1);
```

5. 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 [, int $rownumber])
```

参数

参数	描述
result	[IN] 结果集资源
rownumber	行号。可选参数，缺省从 1 开始

返回值

返回根据所取得的行生成的对象，列名为索引，借助 dm_result 函数可以以列序号为索引进行查找。如果没有更多行则返回 FALSE。

举例说明

例

```
$result1 = dm_fetch_object($ret, 2);
if($result1){
    echo $result1->C1;
    echo $result1->C2;
    $val1 = dm_result($ret,1);
    echo $val1;
}
```

6. dm_result

描述

取得结果数据。

格式

```
mixed dm_result (resource $result, mixed $fields)
```

参数

参数	描述
result	[IN] 结果集
field	[IN] 指定列索引 (从 1 开始) 或列名称 (单引号或双引号)

返回值

无。

举例说明

例

```
SQL> SELECT * from t2;
//查询结果如下:
行号      member_name c0
-----
1          小明        1
```

```

2      小红      2
$ret = dm_query("SELECT * from t2");
$result = dm_result($ret, 1); // 第1列
$result = dm_result($ret, "c0"); // c0列

// 可以按下面方式遍历所有行
while(dm_fetch_row($ret)) {
$result1 = dm_result($ret, "c0");
}

```

7. 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
[,bool $client_flags]]]])
```

参数

参数	描述
server	[IN] 服务器名称，缺省使用 php.ini 中的配置
username	[IN] 用户名称，缺省使用 php.ini 中的配置
password	[IN] 用户密码，缺省使用 php.ini 中的配置
client_flags	保留参数，不起作用

返回值

如果成功则返回一个连接标识，失败则返回 FALSE。用同样的参数第二次调用 dm_connect 不会建立新连接，而将复用已经打开的连接。

举例说明**例**

```
dm_connect("192.168.0.25", "SYSDBA", "SYSDBA", 1);
```

8. dm_pconnect**描述**

打开一个到 DM 服务器的持久连接。

格式

```
resource dm_pconnect ([string $server [, string $username [, string
$password[,bool $client_flags]]]])
```

参数

参数	描述
server	[IN] 服务器名称，缺省使用 php.ini 中的配置
username	[IN] 用户名称，缺省使用 php.ini 中的配置
password	[IN] 用户密码，缺省使用 php.ini 中的配置
client_flags	保留参数，不起作用

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。用同样的参数第二次调用 dm_pconnect 不会建立新连接，而将复用已经打开的连接。

9. dm_error**描述**

返回上一个 DM 操作中的错误信息的数字编码。PHP 7.x `dm_error` 不支持默认连接，除此之外其他用法同 `dm_errno`(PHP 5.x)。

格式

```
int dm_error (resource $link_identifier)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省

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)。

10. `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
<code>link_identifier</code>	[IN] 连接标识符

返回值

如果到服务器的连接可用则返回 TRUE，否则返回 FALSE。

11. `dm_get_server_info`

描述

取得 DM 服务器信息。PHP 7.x `dm_get_server_info` 不支持默认连接，除此之外其他用法同 PHP 5.x。

格式

```
string dm_get_server_info (resource $link_identifier)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省

返回值

返回 `link_identifier` 所使用的服务器版本。

12. `dm_close`

描述

关闭指定的连接标识所关联的 DM 服务器的连接。PHP 7.x `dm_close` 不支持默认连接，除此之外其他用法同 PHP 5.x。

格式

```
bool dm_close (resource $link_identifier)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省

返回值

如果成功则返回 TRUE，失败则返回 FALSE。

13. `dm_affected_rows`

描述

取得前一次数据库操作 INSERT、UPDATE 或 DELETE 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。PHP 7.x `dm_affected_rows` 不支持默认连接，除此之外其他用法同 PHP 5.x。

格式

```
int dm_affected_rows (resource $link_identifier)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省

返回值

对于非 DML 语句 (DDL 语句)，返回 -1。

14. `dm_list_fields`

描述

列出 DM 结果中的字段。PHP 7.x `dm_list_fields` 不支持默认连接，除此之外其他用法同 PHP 5.x。

格式

```
resource dm_list_fields (string $database_name, string $table_name , resource $link_identifier[,string $owner_name [,string $field_name]])
```

参数

参数	描述
<code>database_name</code>	[IN] 数据库名
<code>table_name</code>	[IN] 表名
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省
<code>owner_name</code>	模式名
<code>field_name</code>	指定列名

返回值

返回一个结果指针。

15. `dm_insert_id`

描述

取得上一步 INSERT 操作产生的 ID。PHP 7.x `dm_insert_id` 不支持默认连接，除此之外其他用法同 PHP 5.x。

格式

```
int dm_insert_id (resource $link_identifier)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符，不支持缺省默认连接，不可缺省

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。

6.3.2.2 PHP 7.x 新增接口

1. `dm_exec`

描述

执行一条语句或查询。

格式

```
resource dm_exec(resource $link_identifier [,string $query])
```

参数

参数	描述
link_identifier	[IN] 连接标识符
query	[IN] 语句字符串

返回值

作用同 `dm_query`(PHP 5.x)，可代替 PHP 5.x 中的 `dm_query`、`dm_unbuffered_query`、`dm_more_query_no_result` 以及 `dm_db_query`。

举例说明

例

```
$result = dm_exec($link, $query);
$result = dm_exec($link, "ddl 语句");
```

2. `dm_tables`

描述

列出 DM 数据库中的表。

格式

```
resource
dm_tables (resource $link [, string $qualifier [, string $owner [, string $name [, string $types]]]])
```

参数

参数	描述
link	[IN] 连接句柄，不支持缺省默认连接，不可缺省
qualifier	库名，忽略该参数
owner	模式名
name	表名
types	表 or 视图。值：SYSTEM TABLE, TABLE, VIEW

返回值

在获取成功时返回一个资源标识符，出错时返回 FALSE。可代替 PHP 5.x 中的 `dm_list_tables`。

3. `dm_columns`

描述

列出 DM 数据库中的列。

格式

```
resource
dm_columns (resource $link[, string $qualifier [, string $schema [, string $table_name [, string $column_name]]]])
```

参数

参数	描述
link	[IN] 连接句柄
qualifier	库名
schema	模式名
table_name	表名

column_name	列名过滤格式, 如%
-------------	------------

返回值

在获取成功时返回一个资源标识符, 出错时返回 FALSE。

4. dm_result_all**描述**

取得全部结果数据, 并打印成 html 格式。

格式

```
int dm_result_all (resource $result, [, string $format])
```

参数

参数	描述
result	[IN] 结果集
format	html 格式中<table>标签追加属性 'id="c1", age="c2"'

返回值

返回的是结果集行数, 错误时返回 FALSE。

举例说明**例**

```
dm_result_all($ret, 'id="c1", age="c2"');
```

执行结果如下:

```
<table id="c1", age="c2" >
<tr><th>C1</th><th>C2</th></tr>
<tr><td>10</td><td>20</td></tr>
```

5. dm_binmode**描述**

二进制类型数据的处理方式。该函数用来对输入的二进制类型列数据进行转换处理。二进制类型有: BINARY、VARBINARY 及 BLOB 三种。

当二进制 SQL 数据转换为字符串时, 源数据的每个字节(8位)表示为两个 ASCII 字符, 这些字符是由十六进制数字的 ASCII 字符表示。例如, BINARY 类型的 00000001 被转为十六进制的“01”, BINARY 类型的 11111111 被转为十六进制的“FF”。

格式

```
bool dm_binmode (resource $link_identifier, int $mode)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
mode	模式取值范围: DM_BINMODE_PASSTHRU 不处理 DM_BINMODE_RETURN 转成16进制 DM_BINMODE_CONVERT 转成字符串

返回值

在获取成功时返回 TRUE, 出错时返回 FALSE。

6. dm_close_all**描述**

关闭所有连接。

格式

```
void dm_close_all ( void )
```

参数

无。

返回值

无。

7. dm_autocommit**描述**

设置事务是否为自动提交。

格式

```
bool dm_autocommit (resource $link_identifier, bool $flag)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
flag	取值范围: 1: 是 0: 否

返回值

成功返回 TRUE，失败返回 FALSE。可代替 PHP 5.x 中的 dm_begin_trans，只需将 flag 设为 0，即可设置事务为不自动提交。

8. dm_cursor**描述**

获取游标名。

格式

```
string dm_cursor (resource $result_id)
```

参数

参数	描述
result_id	[IN] 结果集资源

返回值

游标名。

9. dm_errormsg

用法同 dm_error (PHP 5.x)。

10. dm_fetch_into**描述**

获取一行结果集到数组中。

格式

```
int dm_fetch_into(resource $link_identifier, array $parameters_array [, int $rownumber])
```

参数

参数	描述
link_identifier	[IN] 连接标识符
parameters_array	Array 数组
rownumber	行号

返回值

返回结果集的列数，失败返回 FALSE。

11. dm_field_scale**描述**

获取结果集中指定字段的长度。

格式

```
int dm_field_scale (resource $result, int $field_offset)
```

参数

参数	描述
result	[IN] 结果集资源
field_offset	字段序号, 起始值为 1

返回值

指定字段的长度。

12. dm_field_num**描述**

获取结果集中指定字段的序号。

格式

```
int dm_field_num (resource $result_id, string $field_name)
```

参数	描述
result_id	[IN] 结果集资源
field_name	字段名

返回值

指定字段的序号, 起始值为 1。

13. 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, 起始值为 1

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。

14. dm_longreadlen**描述**

设置变长类型读取的最大长度。

格式

```
bool dm_longreadlen (resource $link_identifier, int $length)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
length	读取的最大长度

返回值

在获取成功时返回 TRUE, 出错时返回 FALSE。

15. dm_next_result**描述**

切换结果集。单条 SQL 不能切换，必须是一个过程中产生了多个结果集。

格式

```
bool dm_next_result (resource $result_id)
```

参数

参数	描述
result_id	[IN] 结果集资源

返回值

有下一个结果集返回 TRUE，无结果集返回 FALSE。可代替 PHP 5.x 中的 dm_more_result。

16. dm_rollback

用法同 dm_abort (PHP 5.x)。

17. dm_setopt**描述**

设置 dm 连接和语句的相关属性。

格式

```
bool dm_setopt (resource $id , int $function , int $option , int $param)
```

参数

参数	描述
id	连接标识符或者结果集资源
function	取值范围: 1: conn 2: stmt
option	属性 ID。 当 function 为 1 时，不同属性对应的属性 ID 为： DSQL_ATTR_ACCESS_MODE 101 DSQL_ATTR_AUTOCOMMIT 102 DSQL_ATTR_CONNECTION_TIMEOUT 113 DSQL_ATTR_LOGIN_TIMEOUT 103 DSQL_ATTR_TXN_ISOLATION 108 DSQL_ATTR_LOCAL_CODE 12345 当 function 为 2 时，不同属性对应的属性 ID 为： SQL_ATTR_ENABLE_AUTO_IPD 15 DSQL_ATTR_FETCH_BOOKMARK_PTR 16 DSQL_ATTR_MAX_LENGTH 3 DSQL_ATTR_MAX_ROWS 1 DSQL_ATTR_NOSCAN 2 DSQL_ATTR_PARAM_BIND_OFFSET_PTR 17 DSQL_ATTR_PARAM_BIND_TYPE 18 DSQL_ATTR_PARAM_OPERATION_PTR 19 DSQL_ATTR_PARAM_STATUS_PTR 20 DSQL_ATTR_PARAMS_PROCESSED_PTR 21

DSQL_ATTR_PARAMSET_SIZE	22
DSQL_ATTR_QUERY_TIMEOUT	0
DSQL_ATTR_RETRIEVE_DATA	11
DSQL_ATTR_ROW_BIND_OFFSET_PTR	23
DSQL_ATTR_ROW_BIND_TYPE	5
DSQL_ATTR_ROW_NUMBER	14
DSQL_ATTR_ROW_OPERATION_PTR	24
DSQL_ATTR_ROW_STATUS_PTR	25
DSQL_ATTR_ROWS_FETCHED_PTR	26
DSQL_ATTR_ROW_ARRAY_SIZE	27
DSQL_ATTR_SIMULATE_CURSOR	10
DSQL_ATTR_USE_BOOKMARKS	12
DSQL_ATTR_ROWSET_SIZE	9
param	属性值。各属性 ID 对应的属性值请参考 2.2.2 节中对应属性的介绍

返回值

在设置成功时返回 TRUE，出错时返回 FALSE。可代替 PHP 5.x 中的 dm_set_connect。

18. dm_specialcolumns

描述

返回可唯一标识表中任意行的最佳列或列集，或者返回事务更新表时被自动更新的列或列集。

格式

```
resource dm_specialcolumns(resource $link_identifier, int $type, char* $catalog_name, char* $schema_name, char* $table_name, int $scope, int $nullable);
```

参数

参数	描述
link_identifier	[IN] 连接标识符
type	取值范围： 1：返回可唯一标识表中任意行的最佳列或列集； 2：返回事务更新表时被自动更新的列或列集
catalog_name	目录限定符，仅支持取值为 NULL 或空串
schema_name	模式名
table_name	表名
scope	仅支持取值为 0，否则返回结果集为空
nullable	是否为空，0：否；1：是

返回值

返回一个资源 id。

19. dm_statistics

描述

获取表及其索引的信息。

格式

```
resource  
dm_statistics (resource $link_identifier, string $qualifier, string $owner,  
string $table_name, int $unique, int $accuracy)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名
table_name	表名
unique	unique 属性。取值范围： SQL_INDEX_UNIQUE 表示只返回 unique 索引 SQL_INDEX_ALL 表示返回所有索引
accuracy	忽略

返回值

返回一个结果集或失败返回 FALSE。

20. dm_primarykeys**描述**

获取表的主键。

格式

```
resource dm_primarykeys (resource $link_identifier, string $qualifier, string
$owner, string $table)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名
table	表名

返回值

返回一个结果集或失败返回 FALSE。

21. dm_columnprivileges**描述**

列出列的权限。

格式

```
resource dm_columnprivileges (resource $link_identifier, string $qualifier,
string $owner, string $name, string $column_name)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名
name	表名
column_name	列名

返回值

返回一个结果集或失败返回 FALSE。

22. dm_tableprivileges**描述**

列出表的权限。

格式

```
resource dm_tableprivileges (resource $link_identifier, string $qualifier,
string $owner, string $name)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名
name	表名

返回值

返回一个结果集或失败返回 FALSE。

23. dm_foreignkeys**描述**

获取表的外键。

格式

```
resource dm_foreignkeys (resource $link_identifier, string $pk_qualifier, string
$pk_owner, string $pk_table, string $fk_qualifier, string $fk_owner, string
$fk_table)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
pk_qualifier	主键库名
pk_owner	主键模式名
pk_table	主键表名
fk_qualifier	外键库名
fk_owner	外键模式名
fk_table	外键表名

返回值

返回一个结果集或失败返回 FALSE。

24. dm_procedures**描述**

列出 DM 数据库中的全部过程或指定的过程。

格式

```
resource dm_procedures (resource $link_identifier)
```

或

```
resource dm_procedures (resource $link_identifier, string $qualifier, string
$owner, string $name)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名，支持使用通配符匹配方式： "%" 来匹配零到多个字符， "_" 来匹配单个字符
name	过程名，支持使用通配符匹配方式： "%" 来匹配零到多个字符， "_" 来匹配单个字符

返回值

在获取成功时返回一个资源标识符，出错时返回 FALSE。

25. dm_procedurecolumns

描述

列出 DM 数据库中过程的列。

格式

```
resource dm_procedurecolumns (resource $link_identifier)
```

或

```
resource dm_procedurecolumns (resource $link_identifier, string $qualifier,
string $owner, string $proc, string $column)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名
owner	模式名
proc	过程名
column	列名

返回值

在获取成功时返回一个资源标识符，出错时返回 FALSE。

6.3.3 PHP 8.x 接口

下面是 PHP 8.x 中和 PHP 7.x 同名的接口，但是参数取值和 PHP 7.x 略有不同。

1. dm_tables

描述

列出 DM 数据库中的表。

格式

```
resource
dm_tables (resource $link [, string $qualifier [, string $owner [, string $name [, string $types]]]])
```

参数

参数	描述
link	[IN] 连接句柄
qualifier	库名，仅支持取值为 NULL 或空串
owner	模式名
name	表名
types	表 or 视图。值：SYSTEM TABLE, TABLE, VIEW

返回值

在获取成功时返回一个资源标识符，出错时返回 FALSE。可代替 PHP 5.x 中的 dm_list_tables。

2. dm_columns

描述

列出 DM 数据库中的列。

格式

```
resource
dm_columns (resource $link[, string $qualifier [, string $schema [, string
$table_name [, string $column_name]]]])
```

参数

参数	描述
link	[IN] 连接句柄
qualifier	库名, 仅支持取值为 NULL 或空串
schema	模式名
table_name	表名
column_name	列名过滤格式, 如%

返回值

在获取成功时返回一个资源标识符, 出错时返回 FALSE。

3. dm_statistics**描述**

获取表及其索引的信息。

格式

```
resource
dm_statistics (resource $link_identifier, string $qualifier, string $owner,
string $table_name, int $unique, int $accuracy)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名, 仅支持取值为 NULL 或空串
owner	模式名
table_name	表名
unique	unique 属性。取值: DSQL_INDEX_UNIQUE 表示只返回 unique 索引 DSQL_INDEX_ALL 表示返回所有索引
accuracy	忽略

返回值

返回一个结果集或失败返回 FALSE。

4. dm_primarykeys**描述**

获取表的主键。

格式

```
resource dm_primarykeys (resource $link_identifier, string $qualifier, string
$owner, string $table)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名, 仅支持取值为 NULL 或空串
owner	模式名

table	表名
-------	----

返回值

返回一个结果集或失败返回 FALSE。

5. dm_columnprivileges**描述**

列出列的权限。

格式

```
resource dm_columnprivileges (resource $link_identifier, string $qualifier,
string $owner, string $name, string $column_name)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名, 仅支持取值为 NULL 或空串
owner	模式名
name	表名
column_name	列名

返回值

返回一个结果集或失败返回 FALSE。

6. dm_tableprivileges**描述**

列出表的权限。

格式

```
resource dm_tableprivileges (resource $link_identifier, string $qualifier,
string $owner, string $name)
```

参数

参数	描述
link_identifier	[IN] 连接标识符
qualifier	库名, 仅支持取值为 NULL 或空串
owner	模式名
name	表名

返回值

返回一个结果集或失败返回 FALSE。

7. dm_foreignkeys**描述**

获取表的外键。

格式

```
resource dm_foreignkeys (resource $link_identifier, string $pk_qualifier, string
$pk_owner, string $pk_table, string $fk_qualifier, string $fk_owner, string
$fk_table)
```

参数

参数	描述
link_identifier	[IN] 连接标识符

<code>pk_qualifier</code>	主键库名, 仅支持取值为 NULL 或空串
<code>pk_owner</code>	主键模式名, 可以为 NULL 或空串
<code>pk_table</code>	主键表名, 可以为 NULL 或空串
<code>fk_qualifier</code>	外键库名, 仅支持取值为 NULL 或空串
<code>fk_owner</code>	外键模式名, 可以为 NULL 或空串
<code>fk_table</code>	外键表名, 可以为空串或 NULL

返回值

返回一个结果集或失败返回 FALSE。

8. `dm_procedures`**描述**

列出 DM 数据库中的全部过程或指定的过程。

格式

```
resource dm_procedures (resource $link_identifier)
```

或

```
resource dm_procedures (resource $link_identifier, string $qualifier, string
$owner, string $name)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符
<code>qualifier</code>	库名, 仅支持取值为 NULL 或空串
<code>owner</code>	模式名, 支持使用通配符匹配方式: "%" 来匹配零到多个字符, "_" 来匹配单个字符, 可以设置为 NULL, 但不可以设置为空串
<code>name</code>	过程名, 支持使用通配符匹配方式: "%" 来匹配零到多个字符, "_" 来匹配单个字符, 可以设置为 NULL, 但不可以设置为空串

返回值

在获取成功时返回一个资源标识符, 出错时返回 FALSE。

9. `dm_procedurecolumns`**描述**

列出 DM 数据库中过程的列。

格式

```
resource dm_procedurecolumns (resource $link_identifier)
```

或

```
resource dm_procedurecolumns (resource $link_identifier, string $qualifier,
string $owner, string $proc, string $column)
```

参数

参数	描述
<code>link_identifier</code>	[IN] 连接标识符
<code>qualifier</code>	库名, 仅支持取值为 NULL 或空串
<code>owner</code>	模式名, 可以设置为 NULL, 但不可以设置为空串
<code>proc</code>	过程名, 可以设置为 NULL, 但不可以设置为空串
<code>column</code>	列名, 可以设置为 NULL, 但不可以设置为空串

返回值

在获取成功时返回一个资源标识符, 出错时返回 FALSE。

10. 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 [,bool $client_flags]]]])
```

参数

参数	描述
server	[IN] 服务器名称，缺省使用 php.ini 中的配置
username	[IN] 用户名称，缺省使用 php.ini 中的配置
password	[IN] 用户密码，缺省使用 php.ini 中的配置
client_flags	保留参数，不起作用

返回值

如果成功则返回一个连接标识，失败则返回 FALSE。用同样的参数第二次调用 dm_connect 会建立新连接。

例 使用 dm_connect 建立连接。

```
dm_connect("192.168.0.25", "SYSDBA", "SYSDBA", 1);
```

6.4 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,safetystocklevel,originalprice,newprice,
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";
    //释放资源
    dm_free_result($result);
    //断开连接
    dm_close($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>
```

第 7 章 DM FLDR 编程指南

7.1 C 编程指南

7.1.1 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。

本节介绍的为 FLDR 的 C 接口。

7.1.2 接口说明

7.1.2.1 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

7.1.2.2 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

说明

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

说明

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。关于属性详细内容请参考[表 7.1 属性介绍](#)。

value (输入): 属性值。

length (输入): 属性值占用的空间大小，value 类型为字符串时使用。value 类型为整数时，该参数无意义。

返回值

FLDR_SUCCESS、FLDR_ERROR、FLDR_INVALID_HANDLE

说明

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。

表 7.1 属性介绍

属性 (attrid)	属性说明 (value)	value类型
-------------	--------------	---------

FLDR_ATTR_SERVER	服务器IP地址。缺省情况下为本地服务器 LOCALHOST	字符串
FLDR_ATTR_UID	用户名。不能缺省	字符串
FLDR_ATTR_PWD	密码。缺省情况下SYSDBA	字符串
FLDR_ATTR_PORT	服务器端口。缺省情况下端口号为5236	整数
FLDR_ATTR_TABLE	目标表	字符串
FLDR_ATTR_SET_INDENTITY	是否插入自增列。1是，服务器将使用指定的数据作为自增列的数据进行插入；0否，对于自增列服务器将自动生成数据插入，忽略指定的数据。缺省为0	整数
FLDR_ATTR_DATA_SORTED	数据是否已按照聚集索引排序。缺省为0。 1表示用户必须保证数据已按照聚集索序完成，并且如果表中存在数据，需要插入的数据索引值要比表中数据的索引值大，服务器在做插入操作时顺序进行插入。0表示服务器对于每条记录进行定位插入	整数
FLDR_ATTR_INDEX_OPTION	索引的设置选项，缺省为1。 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； 3：刷新二级索引，数据装载的同时将数据插入二级索引	整数
FLDR_ATTR_INSERT_ROWS	只读属性。当前发送到服务器的行数	整数
FLDR_ATTR_COMMIT_ROWS	只读属性。当前提交到数据库的行数	整数
FLDR_ATTR_PROCESS_ROWS	只读属性。用户绑定的数据行数	整数
FLDR_ATTR_SEND_NODE_NUM	指定数据载入时发送节点的个数，默认由系统计算一个初始值。取值范围为16~65535。若在数据载入时发现发送节点不够用，系统会动态增加分配。在系统内存足够的情况下，可以适当设大SEND_NODE_NUMBER值，提升dmfldr载入性能	整数
FLDR_ATTR_TASK_THREAD_NUM	指定数据载入时处理用户数据的线程数目。默认情况下，dmfldr将该参数值设为系统CPU的个数。在dmfldr客户端所在机器CPU大于8环境中，提高TASK_THREAD_NUMBER值可以提升dmfldr装载性能。取值范围为1~128	整数
FLDR_ATTR_BAD_FILE	记录错误数据的文件绝对路径。缺省为 dmfldr.exe所在目录。缺省的错误文件名为 fldr.bad。文件记录的信息为数据文件中存在 格式错误的行数据以及转换出错的行数据	字符串
FLDR_ATTR_DATA_CHAR_SET	数据文件中数据的编码格式。默认与当前机器编码格式一致。取值有GBK、GB18080、UTF-8、 SINGLE_BYTE和EUC-KR五种：GBK和GB18030 对应中文编码；UTF-8对应UTF-8国际编码；	整数

	SINGLE_BYTE对应单字节字符编码; EUC-KR 对应韩文字符集	
FLDR_ATTR_LOG_FILE	指定日志文件绝对路径, 缺省为dmfldr.exe所在目录。缺省日志文件名为fldr.log。日志文件记录了dmfldr运行过程中的工作信息、错误信息以及统计信息	字符串
FLDR_ATTR_ERRORS_PERMIT	允许出现错误数据的行数, 超过该值装载将报错。取值范围为0~4294967295, 缺省为100	整数
FLDR_ATTR_SSL_PATH	通信加密的SSL数字证书路径, 缺省为不使用加密。数字证书路径由用户自己创建, 将相应的证书需放入该文件夹中。其中服务器证书必须与dmserver目录同级, 客户端目录可以任意设置	字符串
FLDR_ATTR_SSL_PWD	通信加密的SSL数字证书密码。和ssl_path一起使用	字符串
FLDR_ATTR_LOCAL_CODE	指定本地字符集。取值为UTF-8、GBK、BIG5、ISO 8859 9、EUC JP、EUC KR、KOI8R、ISO 8859 1、ISO 8859 11、ASCII或GB18030。缺省值是本地编码	整数
FLDR_ATTR_NULL_MODE	指定载入数据时对NULL字符串和空值的处理方式。取值1和0, 缺省为0。 1表示数据载入时将NULL字符串处理为空值, 数据导出时则空值处理为NULL字符串; 0表示数据载入时将NULL字符串处理为字符串	整数
FLDR_ATTR_SERVER_BLDR_NUM	水平分区表装载时, 指定服务器BLDR的最大个数, 整数类型, 取值范围为1~1024, 缺省为64。服务器的BLDR保存水平分区子表相关信息, BLDR_NUM的设置也就指定了服务器能同时载入的水平分区子表的个数	整数
FLDR_ATTR_BDTA_SIZE	BDTA (Batch DaTA) 的大小, 整数类型, 取值范围为100~10000, 缺省为5000。 BDTA代表DM数据库批量数据处理机制中一个批量, 在内存、CPU允许的条件下, 增大BDTA_SIZE能加快装载速度; 在网络是装载性能瓶颈时, 增大BDTA_SIZE影响不大	整数
FLDR_ATTR_COMPRESS_FLAG	指定是否压缩BDTA。1是; 0否。缺省为0。 压缩 BDTA 能节省网络带宽, 但同时也会加重CPU的负担, 用户应根据具体情况考虑是否制定压缩	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1则服务器将数据写入磁盘后才返回响应消息; 若设置为0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。缺省为0	整数
FLDR_ATTR_MPP_CLIENT	当服务器环境为MPP环境时, 若此参数值为1, 为客户端分发模式, 数据在FLDR客户端分发好	整数

	后直接往指定站点发送数据；若此参数值为0，为本地分发模式，FLDR客户端将数据全部发往连接的MPP EP站点。缺省值为1	
FLDR_ATTR_DATA_PATH	指定数据文件路径	字符串
FLDR_ATTR_BUFFER_SIZE	指定读取文件缓冲区的页大小。取值范围为1~2048，缺省为10。单位为MB	整数
FLDR_ATTR_LOB_DIRECTORY	指定dmfldr使用的大字段数据存放的目录	字符串
FLDR_ATTR_LOB_FIL_NAME	指定dmfldr导出大字段数据的文件名。缺省为dmfldr.lob	字符串
FLDR_ATTR_BLOB_TYPE	指定BLOB数据值的实际类型。取值：HEX表示值为十六进制；HEX_CHAR表示值为十六进制字符。缺省为HEX_CHAR	字符串
FLDR_ATTR_ENABLE_CLASS_TYPE	指定是否以BLOB方式导入导出CLASS类型列数据。缺省为FALSE	布尔类型
FLDR_ATTR_CONFLICT_FLAG	指定是否忽略唯一性冲突。缺省为FALSE	布尔类型
FLDR_ATTR_RECONN	指定自动重连次数。取值范围为0~4294967295，缺省为0，表示不进行自动重连	整数
FLDR_ATTR_RECONN_TIME	指定自动重连等待时间。取值范围为1~10000，缺省为5。单位为秒	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() 的说明部分。

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

说明

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`

说明

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`

说明

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`

说明

支持绑定的 C 类型见下表:

类型	宏定义	说明
二进制数据	<code>FLDR_C_BINARY</code>	变长
字符数据	<code>FLDR_C_CHAR</code>	变长
1 字节整型数据	<code>FLDR_C_BYTE</code>	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`

说明

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): 在多行绑定的情况下, 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): 在多行绑定的情况下, 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

说明

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,
    fsint4*         err_code,
    fchar*          err_msg,
    fsint4          buf_sz,
    fsint4*         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`

说明

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。仅支持载入行数, 不支持载出行数。

返回值

`FLDR_SUCCESS`、`FLDR_ERROR`

说明

此接口让用户可以通过自定义程序实现数据快速装载。

7.1.3 基本示例

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_dll.lib` 这个库文件, 在执行阶段需要用到动态库 `dmfldr_dll.dll`、`dmcast.dll`、`dmcalc.dll`、`dmeelog.dll`、`dmmem.dll`、`dmos.dll`、`dmutil.dll`、`dmdta.dll`、`dmcfg.dll`、`dmstrt.dll`、`dmcpri.dll`、`dmcyt.dll`、`dmcvt.dll`、`dmmout.dll`、`dmcomm.dll`、`dmdpi.dll`、`dmclientlex.dll`、`dmfldr_comm.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;
}

```

7.2 JNI 编程指南

7.2.1 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。

7.2.2 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

说明

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。

说明

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

```
boolean  
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。关于属性详细内容请参考[表 7.1 属性介绍](#)。

`value` (输入): 属性值, 根据 `attrid` 不同该参数的意义不同。

返回值

`true`, `false`

4. `getAttribute`

描述

获取快速装载实例的属性。

格式

```
String
getAttribute(
    int      attr_id
);
```

参数

`attr_id` (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` 的说明部分。

返回值

属性值。

说明

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

`true`, `false`

说明

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. `uninitialize`

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

说明

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。此处 FLDR JNI 调用的 JAVA 数据类型是从 com.dameng.loader.DataTypes 里进行调用。[第 4 章 DM JDBC 编程指南](#)中 JDBC 接口调用 JAVA 数据类型是从 java.sql.Types 里进行调用。com.dameng.loader.DataTypes 的数据类型与 java.sql.Types 的数据类型并不相同，两者的差异见下表：

com.dameng.loader.DataTypes (FLDR JNI)	java.sql.Types (JDBC 接口)
DataTypes.FLDR_C_TINYINT	Types.BIT
DataTypes.FLDR_C_SHORT	Types.TINYINT
DataTypes.FLDR_C_INT	Types.SMALLINT
DataTypes.FLDR_C_BIGINT	Types.INTEGER
DataTypes.FLDR_C_BINARY	Types.BIGINT
	Types.STRUCT
	Types.BINARY
	Types.VARBINARY
	Types.LONGVARBINARY

	Types.BLOB
DataTypes.FLDR_C_CHAR	其他

`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, 默认为 “`yyyy-mm-dd hh-mm-ss`”。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 或 1。

返回值

`true`, `false`

8. `fldr_Bind_Columns_Direct_ex`

描述

绑定输入列。功能与 `fldr_Bind_Columns_ex` 函数相同, `fldr_Bind_Columns_Direct_ex` 仅支持绑定字符串类型数据, 且通常结合 JAVA 的直接内存使用, 可以减少 JNI 内存拷贝。

格式

```
boolean
fldr_Bind_Columns_Direct_ex(
    int          colNumber,
    int          type,
    String       datefmt,
    Object       data,
    int          maxcollen,
    int[]        ind_len,
    int          threadNum
);
```

参数

`type` (输入): 用户绑定数据的 JAVA 类型。此处仅支持 `DataTypes.FLDR_C_CHAR`。

`data` (输入): 指向用于存放数据内存的地址指针。通常使用 JAVA 的直接内存地址指针。

其余参数请参考 `fldr_Bind_Columns_ex` 函数的参数说明。

返回值

`true`, `false`

9. `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): 在多行绑定的情况下, 指明绑定的行数。
`threadNum` (输入): 多线程调用, 线程序号, 单线程传 0 或 1。
`seqNo` (输入): 数据批次, 从 1 开始自增, 保证数据有序。

返回值

`true`, `false`

10. setControl**描述**

利用控制文件执行数据装载。

格式

```
long
setControl(
    String     ctrl
);
```

参数

`ctrl` (输入): 控制文件的内容。

返回值

成功装载的行数。

11. batch**描述**

批量提交行数据, 永久保存到服务器。

格式

```
long
batch();
```

参数

无。

返回值

批量提交的行数。

说明

将上一次调用 `batch` 之后所有 `fldr_Send_Rows_ex` 装载的行进行提交, 永久保存到数据库中。

12. finish**描述**

结束一次批量操作。

格式

```
boolean
finish();
```

参数

无。

返回值

true, false

7.2.3 基本示例

例1 直接指定待载入的数据，并将其载入至数据库。

第一步，数据准备，创建数据库表 test1。

```
create table sysdba.test1(c1 int ,c2 varchar(50));
commit;
```

第二步，将数据载入至数据库

```
package com.dameng;
import java.io.File;
import com.dameng.floder.DataTypes;
import com.dameng.floder.Instance;
import com.dameng.floder.Properties;

public class TestFloder {
    public static void main(String[] args)
    {
        Instance instance = new Instance();
        int threadNum = 1;
        TestFloder floder = new TestFloder();
        String tableName = "SYSDBA.TEST1";

        //使用当前类中自定义的init函数，设置属性信息，并初始化当前快速装载实例
        boolean success = floder.init(instance, threadNum, tableName);

        //若快速装载实例初始化成功，则直接指定待装载数据，并按行进行数据装载
        if(success){
            try{
                int      c1[] = {1,2,3,4,5,6,7,8,9};      //数值类型数据
                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","a12
34567","a12345678","a123456789"}; //字符串类型数据
                int      c2_ind[] = {2,3,4,5,6,7,8,9,10};
                byte[][]  byteArray2 = new byte[9][];
            }
        }
    }
}
```

```

        for(int i = 0; i < 9; i++){
            byteArray2[i] = c2[i].getBytes();
        }
        //按行装载数据
        for(int i = 1; i < 10; i++){
            //绑定数值类型数据, data参数传入int数组c1
            instance.fldr_Bind_Columns_ex(1, DataTypes.FLDR_C_INT, null, c1,
4, c1_ind, 1);
            //绑定字符串类型数据, data参数传入byte数组byteArray2
            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();
    }
}

//若快速装载实例初始化失败, 则打印错误信息, 并释放快速装载实例
if(!success){
    String message = instance.getErrorMsg();
    System.out.println("出错信息:" + message);
    instance.free();
}

//快速装载完成, 打印实际插入的行数, 并释放快速装载实例
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.uninitialize();
    instance.free();
}
}

//*****自定义init函数, 设置属性信息, 并初始化当前快速装载实例*****
public boolean init(Instance instance, int threadNum, String tableName)
{
    // 分配句柄
    instance.allocInstance();
    // 设置必要的属性信息
}

```

```

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);

// 其余属性用户均可根据实际情况选择性设置
// 设置编码
//instance.setAttribute(Properties.FLDR_ATTR_DATA_CHAR_SET,
System.getProperty("file.encoding"));
// 设置是否插入自增列
//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));

// 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_1.TXT");
    }else{
        instance.setAttribute(Properties.FLDR_ATTR_BAD_FILE, dmHome +
File.separator + "BADFILE_1.TXT");
    }
}

// 装载日志
if (dmHome != null && !dmHome.equals("")) {
    if (dmHome.endsWith("/") || dmHome.endsWith("\\")) {
        instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome +
"FLDRLOG_1.TXT");
    }else{
        instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome +
File.separator + "FLDRLOG_1.TXT");
    }
}

//初始化当前快速装载实例

```

```

        boolean success = instance.initializeInstance(null, null, null,
tableName);
        return success;
    }
}

```

例 2 将数据文件中的数据载入至数据库

第一步，数据准备，创建数据库表 test2。

```

create table sysdba.test2(c1 int ,c2 varchar(50));
commit;

```

第二步，准备数据文件 e:\fldr_insert.txt，文件内容如下：

```

1|a
2|b

```

第三步，将数据文件内容载入至数据库中。

```

package com.dameng;
import java.io.File;
import com.dameng.floder.Instance;
import com.dameng.floder.Properties;

public class TestFloder {
    public static void main(String[] args)
    {
        Instance instance = new Instance();
        int threadNum = 1;
        TestFloder floder = new TestFloder();
        String tableName = "SYSDBA.TEST2";

        //使用当前类中自定义的init函数，设置属性信息，并初始化当前快速装载实例
        boolean success = floder.init(instance, threadNum, tableName);

        //若快速装载实例初始化成功，则将数据文件内容载入至数据库中
        if(success){
            String ctl = "LOAD DATA\r\n" +
                "INFILE 'e:\\fldr_insert.txt'\r\n" +
                "INTO TABLE SYSDBA.TEST2\r\n" +
                "FIELDS '|'\r\n" +
                "(\r\n" +
                "C1 ,\r\n" +
                "C2 \r\n" +
                ")";

            //根据ctrl内容进行数据载入
            instance.setControl(ctl);
        }
    }
}

```

```

//若快速装载实例初始化失败，则打印错误信息，并释放快速装载实例
if(!success) {
    String message = instance.getErrorMsg();
    System.out.println("出错信息：" + message);
    instance.free();
}

//快速装载完成，打印实际插入的行数，并释放快速装载实例
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.uninitialize();
    instance.free();
}
}

*****自定义init函数，设置属性信息，并初始化当前快速装载实例*****
public boolean init(Instance instance, int threadNum, String tableName)
{
    // 分配句柄
    instance.allocInstance();
    // 设置必要的属性信息
    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);

    // 其余属性用户均可根据实际情况选择性设置
    // 设置编码
    //instance.setAttribute(Properties.FLDR_ATTR_DATA_CHAR_SET,
    System.getProperty("file.encoding"));
    // 设置是否插入自增列
    //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));
}

```

```

// 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_2.TXT");
    }else{
        instance.setAttribute(Properties.FLDR_ATTR_BAD_FILE, dmHome +
File.separator + "BADFILE_2.TXT");
    }
}

// 装载日志
if (dmHome != null && !dmHome.equals("")){
    if (dmHome.endsWith("/") || dmHome.endsWith("\\")){
        instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome +
"FLDRLOG_2.TXT");
    }else{
        instance.setAttribute(Properties.FLDR_ATTR_LOG_FILE, dmHome +
File.separator + "FLDRLOG_2.TXT");
    }
}

// 初始化当前快速装载实例
boolean success = instance.initializeInstance(null, null, null,
tableName);
return success;
}
}

```

例3 将数据库表中的数据载出到文件中

第一步，数据准备，创建数据库表 test3。

```

create table sysdba.test3(c1 int);
insert into sysdba.test3 values(1), (2), (3);
commit;

```

第二步，将表 test3 中的数据载出到文件 e:\fldr_out.txt 中。

```

package com.dameng;
import com.dameng.floder.Instance;
import com.dameng.floder.Properties;

public class TestFloder {
    public static void main(String[] args)

```

```

{
    Instance instance = new Instance();
    try{
        if(instance.allocInstance()){
            instance.setAttribute(Properties.FLDR_ATTR_SERVER,
"192.168.1.55");
            instance.setAttribute(Properties.FLDR_ATTR_PORT, "5236");
            instance.setAttribute(Properties.FLDR_ATTR_UID, "SYSDBA");
            instance.setAttribute(Properties.FLDR_ATTR_PWD, "SYSDBA");
            instance.setAttribute(Properties.FLDR_ATTR_DATA_CHAR_SET,
"GBK");
            //初始化当前快速装载实例
            instance.initializeInstance(null, null, null, "SYSDBA.TEST3");
            String ctl = "OPTIONS\r\n" +
                "(\r\n" +
                "mode='out'\r\n" +
                ") \r\n" +
                "Load\r\n" +
                "infile 'e:\\fldr_out.txt'\r\n" +
                "Append\r\n" +
                "into table SYSDBA.TEST3\r\n" +
                "FIELDS '|'";
            //根据ctrl内容进行数据载出
            instance.setControl(ctl);
        }else{
            System.out.println(instance.getErrorMsg());
        }
    }finally{
        //释放
        instance.uninitialize();
        instance.free();
    }
}
}

```

第三步，查看文件 e:\\fldr_out.txt。

1
2
3

第8章 DM DEXP/DIMP JNI 编程指南

8.1 接口介绍

Java 调用 DM 逻辑导出导入接口导数据是通过 JNI 接口来完成的。所引用的对象在 com.dameng.impexp.jar 中的 com.dameng.impexp.ImpExpDLL。主要类是 com.dameng.impexp.ImpExpDLL。

8.2 接口说明

1. dll_exp_dm

描述:

用于逻辑导出。

格式:

```
public static native int dll_exp_dm(
    int          expMode,
    byte[]       userid,
    byte[]       schemaName,
    byte[]       tableName,
    byte[]       expFilePath,
    byte[]       logFilePath
);
```

参数:

expMode (输入): 值为 1 时, 以表方式导出; 值为 2 时, 以模式方式导出; 值为 3 时, 以整库方式导出。

userid (输入): 连接字符串, 与导出工具 userid 格式一致。

schemaName (输入): 需要导出的模式, 以模式方式导出时有效, 格式为 (schema1,schema2,...)。

tableName (输入): 需要导出的表, 以表方式导出时有效, 格式为 (table1,table2,...)。

expFilePath (输入): 导出文件路径。

logFilePath (输入): 日志文件路径。

返回值:

true, false

2. dll_imp_dm

描述:

用于逻辑导入。

格式:

```
public static native int dll_imp_dm(
    int          impMode,
```

```

byte[]      userid,
byte[]      schemaName,
byte[]      tableName,
byte[]      impFilePath,
byte[]      logFilePath,
byte[]      remapSchema
);

```

参数:

impMode (输入): 值为 1 时, 以表方式导入; 值为 2 时, 以模式方式导入; 值为 3 时, 以整库方式导入。

userid (输入): 连接字符串, 与导入工具 userid 格式一致。

schemaName (输入): 需要导入的模式, 以模式方式导入时有效, 格式为 (schema1,schema2,...)。

tableName (输入): 需要导入的表, 以表方式导入时有效, 格式为 (table1,table2,...)。

impFilePath (输入): 导入文件路径。

logFilePath (输入): 日志文件路径。

remapSchema (输入) : 模 式 映 射 串 , 格 式 为
(SOURCE_SCHEMA:TARGET_SCHEMA,...)。

返回值:

true, false

3. dll_exec_sql_file**描述:**

用于执行 SQL 脚本文件。

格式:

```

public static native int dll_exec_sql_file(
    byte[]      userid,
    byte[]      sqlFilePath,
    byte[]      logFilePath
);

```

参数:

userid (输入): 连接字符串, 与导入工具 userid 格式一致。

sqlFilePath (输入): SQL 脚本文件路径。

logFilePath (输入): 日志文件路径。

返回值:

true, false

8.3 基本示例

第一步, 编写逻辑导出类, 包含逻辑导出数据表、模式以及数据库的方法。

```

import java.io.*;
import com.dameng.impExp.*;
public class TestExp {

```

```

//导出普通表
public static void TestExpTable()
{
    int expMode = 1;
    String userid = "SYSDBA/SYSDBA@localhost";
    //String schemaName = "";
    String tableName = "TEST";
    String expFilePath = "D:\\test_tab.dmp";
    String logFilePath = "D:\\test_tab.log";
    //System.out.println("测试信息：导出表（普通表）开始...");
    try {
        int ret=ImpExpDLL.dll_exp_dm(expMode, userid.getBytes(), null,
tableName.getBytes(), expFilePath.getBytes(),logFilePath.getBytes());
        System.out.println("导出表成功，ret = " + ret);
    }
    catch (Exception e) {
        System.out.println("导出表产生异常：" + e.getMessage());
    }
}

//导出表名为中文的表
public static void TestExpTable_01()
{
    int expMode = 1;
    String userid = "SYSDBA/SYSDBA@localhost";
    //String schemaName = "";
    String tableName = "表名为中文";
    String expFilePath = "D:\\test_tab_1.dmp";
    String logFilePath = "D:\\test_tab_1.log";
    //System.out.println("测试信息：导出表(中文表名)开始...");
    try {
        int ret=ImpExpDLL.dll_exp_dm(expMode, userid.getBytes(), null,
tableName.getBytes(), expFilePath.getBytes(),logFilePath.getBytes());
        System.out.println("导出表成功，ret = " + ret);
    }
    catch (Exception e) {
        System.out.println("导出表产生异常：" + e.getMessage());
    }
}

//导出不存在的表
public static void TestExpTable_02()
{
    int expMode = 1;
}

```

```

String userid = "SYSDBA/SYSDBA@localhost";
String tableName = "T_NOT_EXISTED";
String expFilePath = "D:\\test_tab_2.dmp";
String logFilePath = "D:\\test_tab_2.log";
//System.out.println("测试信息：导出表(表不存在)开始...");
try {
    int ret=ImpExpDLL.dll_exp_dm(expMode, userid.getBytes(), null,
tableName.getBytes(), expFilePath.getBytes(),logFilePath.getBytes());
    System.out.println("测试信息：测试失败，未获取到预期的异常 ");
}
catch (Exception e) {
    System.out.println("测试信息：测试成功，获取到预期的异常"+
e.getMessage());
}

//导出模式
public static void TestExpSchema() throws UnsupportedEncodingException
{
    int expMode = 2;
    String userid = "SYSDBA/SYSDBA@localhost";
    String schemaName = "SYSDBA";
    //String tableName = "";
    String expFilePath = "D:\\test_sch.dmp";
    String logFilePath = "D:\\test_sch.log";
    //String errMess = null;
    try {
        int ret=ImpExpDLL.dll_exp_dm(expMode, userid.getBytes(),
schemaName.getBytes(),null, expFilePath.getBytes(),logFilePath.getBytes());
        System.out.println("导出模式成功, ret = " + ret);
    }
    catch (Exception e) {
        System.out.println("导出模式产生异常: "+ e.getMessage());
    }
}

//导出数据库
public static void TestExpDB() throws UnsupportedEncodingException
{
    int expMode = 3;
    String userid = "SYSDBA/SYSDBA@localhost";
    //String schemaName = "";
    //String tableName = "";
    String expFilePath = "D:\\test_db.dmp";
}

```

```
String logFilePath = "D:\\test_db.log";
//String errMess = null;
try {
    int ret=ImpExpDLL.dll_exp_dm(expMode, userid.getBytes(), null, null,
expFilePath.getBytes(),logFilePath.getBytes());
    System.out.println("导出全库成功, ret = " + ret);
}
catch (Exception e) {
    System.out.println("导出全库产生异常: "+ e.getMessage());
}
}
```

第二步，编写逻辑导入类，包含逻辑导入数据表、模式以及数据库的方法。

```
import java.io.*;
import com.dameng.impExp.*;
public class TestImp {
    //导入普通表
    public static void TestImpTable()
    {
        int impMode = 1;
        String userid = "SYSDBA/SYSDBA@127.0.0.1";
        //String schemaName = "";
        String tableName = "TEST";
        String impFilePath = "D:\\test_tab.dmp";
        String logFilePath = "D:\\test_tab2.log";
        //System.out.println("测试信息：导入表(普通表)开始...");
        try {
            int ret=ImpExpDLL.dll_imp_dm(impMode, userid.getBytes(), null,
tableName.getBytes(), impFilePath.getBytes(),logFilePath.getBytes(),null);
            System.out.println("导入表成功, ret = " + ret);
        }
        catch (Exception e) {
            System.out.println("导入表产生异常: "+ e.getMessage());
        }
    }

    //导入表名为中文的表
    public static void TestImpTable_01()
    {
        int impMode = 1;
        String userid = "SYSDBA/SYSDBA@localhost";
        String tableName = "表名为中文";
        String impFilePath = "D:\\tmp del\\test tab imp 01.dmp";
```

```

String logFilePath = "D:\\tmp_del\\test_tab_imp_01.log";
System.out.println("测试信息：导入表(中文表名)开始...");

try {
    int ret=ImpExpDLL.dll_imp_dm(impMode, userid.getBytes(), null,
tableName.getBytes(), impFilePath.getBytes(),logFilePath.getBytes(),null);
    System.out.println("导入表成功, ret = " + ret);
}
catch (Exception e) {
    System.out.println("导入表产生异常: "+ e.getMessage());
}

}

//导入模式
public static void TestImpSchema()
{
    int impMode = 2;
    String userid = "SYSDBA/SYSDBA@127.0.0.1";
    String schemaName = "SYSDBA";
    //String tableName = "";
    String impFilePath = "D:\\test_sch.dmp";
    String logFilePath = "D:\\test_sch2.log";
    try {
        int ret=ImpExpDLL.dll_imp_dm(impMode, userid.getBytes(),
schemaName.getBytes(),null,
impFilePath.getBytes(),logFilePath.getBytes(),null);
        System.out.println("导入模式成功, ret = " + ret);
    }
    catch (Exception e) {
        System.out.println("导入模式产生异常: "+ e.getMessage());
    }
}

}

//导入数据库
public static void TestImpDB()
{
    int impMode = 3;
    String userid = "SYSDBA/SYSDBA@localhost";
    //String schemaName = "";
    //String tableName = "";
    String impFilePath = "D:\\test_db.dmp";
    String logFilePath = "D:\\test_db2.log";
    try {
        int ret=ImpExpDLL.dll_imp_dm(impMode, userid.getBytes(), null, null,
impFilePath.getBytes(),logFilePath.getBytes(),null);
    }
}

```

```

        System.out.println("导入库成功, ret = " + ret);
    }
    catch (Exception e) {
        System.out.println("导入库产生异常: "+ e.getMessage());
    }
}
}

```

第三步，编写执行 SQL 脚本文件类。

```

import java.io.*;
import com.dameng.impExp.*;
public class TestSql {
    public static void TestSql01()
    {
        String userid = "SYSDBA/SYSDBA@127.0.0.1";
        String sqlFilePath = "D:\\test.sql";
        String logFilePath = "D:\\test.log";
        try {
            int ret=ImpExpDLL.dll_exec_sql_file(userid.getBytes(),
sqlFilePath.getBytes(),logFilePath.getBytes());
            System.out.println("执行SQL成功, ret = " + ret);
        }
        catch (Exception e) {
            System.out.println("执行SQL产生异常: "+ e.getMessage());
        }
    }
}

```

第四步，测试上述各类中的方法。

```

import java.io.UnsupportedEncodingException;
import java.sql.*;
public class TestImpExp {
    /**
     * @param args
     * @throws UnsupportedEncodingException
     */
    public static void main(String[] args) throws UnsupportedEncodingException
    {
        // TODO Auto-generated method stub
        Connection conn = null;
        try{
            //导出普通表
            TestExp.TestExpTable();
            //导入普通表

```

```
TestImp.TestImpTable();
//执行SQL脚本文件
TestSql.TestSql01();
}
catch(Exception e){
    System.out.println("产生未预期的异常: " + e.getMessage());
}
}
}
```

第 9 章 Logmnrr 接口使用说明

Logmnrr 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。

在使用 Logmnrr 包中的接口之前，需要先开启两个日志相关参数：一是开启归档（设置 INI 参数 ARCH_INI 为 1）；二是开启在日志中记录逻辑操作的功能（设置 INI 参数 RLOG_APPEND_LOGIC 为 1、2 或者 3）。

9.1 JNI 接口

logmnrr.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。

9.1.1 接口说明

logmnrr.jar 包括两个类：LogmnrrDll 和 LogmnrrRecord。logmnrr.jar 位于 DM 安装目录的 jar 文件夹下，例如：D:\dmdbms\jar\logmnrr.jar。LogmnrrDll 中包含了所有功能接口；LogmnrrRecord 是日志分析结果数据的返回值对应的对象类型。

9.1.1.1 初始环境

函数原型

```
int LogmnrrDll.initLogmnrr();
```

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。

9.1.1.2 创建连接

函数原型

```
long LogmnrrDll.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。

9.1.1.3 添加日志文件

函数原型

```
int LogmnrrDll.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 endLogmnrr，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。

注意事项

如果当前已经 startLogmnrr，则 addLogFile 将报错，如果需要添加新的日志文件，需要先调用 endLogmnrr 结束当前日志分析后再添加文件。

9.1.1.4 移除日志文件

函数原型

```
int LogmnrrDll.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`: 需要移除的日志文件名（绝对路径）。

返回值

0: 移除成功；
< 0 的值和异常：移除失败。

注意事项

同 `addLogFile` 一样，如果当前已经 `startLogmnrr`，则 `removeLogFile` 将报错，如果需要移除日志文件，需要先调用 `endLogmnrr` 结束当前日志分析后再移除文件。

9.1.1.5 启动日志分析

函数原型

```
int LogmnrrDll.startLogmnrr(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。

9.1.1.6 获取数据

函数原型

```
LogmnrrRecord[] LogmnrrDll.getData(long connId, int rounum);
```

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。
`rounum`: 获取行数。

返回值

获取的 `logmnrr` 记录，为 `LogmnrrRecord` 对象数组，详见 [9.1.1.6.1 LogmnrrRecord 对象说明](#)。

9.1.1.6.1 LogmnrrRecord 对象说明

`LogmnrrRecord` 对象支持的成员变量如下表所示。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	String	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。缺省为 0

9.1.1.7 终止日志分析

函数原型

```
int LogmnrrDll.endLogmnrr(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。

9.1.1.8 关闭连接

函数原型

```
int LogmnrrDll.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。

9.1.1.9 销毁环境

函数原型

```
boolean LogmnrrDll.deinitLogmnrr();
```

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。

9.1.1.10 设置属性

函数原型

```
int LogmnrrDll.setAttr(long connId, int attr, int attr_value)
```

功能说明

设置属性。

有如下属性可以配置:

LogmnrrDll.LOGMNR_ATTR_PARALLEL_NUM: 并行线程数, 取值范围(2, 16);

LogmnrrDll.LOGMNR_ATTR_BUFFER_NUM: 任务缓存节点数, 取值范围(8, 1024);

LogmnrrDll.LOGMNR_ATTR_CONTENT_NUM: 结果缓存节点数, 取值范围(256, 2048);

LOGMNR_ATTR_TRX_END: 是否查找事务结束记录, 缺省为 1, 取值范围 0 或 1;

`LOGMNR_ATTR_TRX_WAIT_TIME`: 查找事务结束记录的最大等待时间, 单位 s, 缺省为 60s, 取值范围 (0, 600), 设置成 0 表示死等, 只有 `LOGMNR_ATTR_TRX_END` 为 1 时才有效。

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。

9.1.2 基本示例

下面用一个例子来说明 `dmlogmnrr` 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。

开启两个日志相关参数: 一是开启归档 (设置 INI 参数 `ARCH_INI` 为 1); 二是开启在日志中记录逻辑操作的功能 (设置 INI 参数 `RLOG_APPEND_LOGIC` 为 1、2 或者 3)。

要 分 析 的 日 志 为
D:\dmda\arch\ARCHIVE_LOCAL1_20160704082303068.log。分析的结果放在
D:\dmda\result.txt 中。其中 `logmnrr.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.logmnrr.LogmnrrDll;
import com.dameng.logmnrr.LogmnrrRecord;
public class dbmslob {
    public static void main(String[] args) {
        try {
            LogmnrrDll.initLogmnrr();
            long connid = LogmnrrDll.createConnect("localhost", 5236, "SYSDBA",
                "SYSDBA");
            LogmnrrDll.addLogFile(connid, "D:\\dmda\\arch\\ARCHIVE_LOCAL1_20160704082
303068.log", 3);
            LogmnrrDll.startLogmnrr(connid, -1, null, null);

            LogmnrrRecord[] arr = LogmnrrDll.getData(connid, 100);
            PrintStream ps = new PrintStream("D:\\dmda\\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");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

"\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();
    LogmnrrDll.endLogmnrr(connid, 1);
    LogmnrrDll.deinitLogmnrr();
} catch (Exception e) {
    e.printStackTrace();
}
}

```

```
}
```

分析的结果在 D:\dmdata\result.txt 中查看。

9.2 C 接口

`dmlogmnrr_client.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。`dmlogmnrr_client` C 接口依赖的动态链接库在 DM 安装目录\bin 目录下，依赖的静态库 `dmlogmnrr_client.lib` 在 DM 安装目录\include 目录下，所需的头文件 `logmnrr_client.h` 也在 DM 安装目录\include 目录下。

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。

9.2.1 接口说明

9.2.1.1 初始化环境

函数原型

```
dmcode_t  
logmnrr_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。

9.2.1.2 创建连接

函数原型

```
dmcode_t  
logmnrr_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。

9.2.1.3 增加日志文件

函数原型

```
dmcode_t
logmngr_client_add_logfile(
    void*      conn,
    schar*    logfilename,
    uint       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 endLogmngr, 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 startLogmngr, 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。

9.2.1.4 删 除日志文件

函数原型

```
dmcode_t
logmngr_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 `startLogmnrr` 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

9.2.1.5 启动日志分析

函数原型

```
dmcode_t
logmnrr_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，缺省为 -1，表示不区分事务号。

`starttime`: 分析归档日志的起始时间，缺省为 1988/1/1。

`endtime`: 分析归档日志的结束时间，缺省为 2110/12/31。

返回值

0: 成功；

<0: 错误。

9.2.1.6 获取信息

函数原型

```
dmcode_t
logmnrr_client_get_data(
    void*          conn,
    lint           row_num,
    logmnrr_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 logmngr 记录。

9.2.1.7 终止日志分析

函数原型

```
dmcode_t
logmngr_client_end(
    void*      conn,
    uint       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。

9.2.1.8 结束连接

函数原型

```
dmcode_t
logmngr_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。

9.2.1.9 销毁环境

函数原型

```
dmcode_t
logmnrr_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。

9.2.1.10 设置日志分析属性

函数原型

```
dmcode_t
logmnrr_client_set_attr(
    void*          conn,
    uint           attr,
    void*          val,
    uint           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。

有如下属性可以设置:

`LOGMNR_ATTR_PARALLEL_NUM`: 并行线程数, 取值范围(2, 16);

`LOGMNR_ATTR_BUFFER_NUM`: 任务缓存节点数, 取值范围(8, 1024);

`LOGMNR_ATTR_CONTENT_NUM`: 结果缓存节点数, 取值范围(256, 2048);

`LOGMNR_ATTR_TRX_END`: 是否查找事务结束记录, 缺省为 1, 取值范围 0 或 1;

`LOGMNR_ATTR_TRX_WAIT_TIME`: 查找事务结束记录的最大等待时间, 单位为 s, 缺省为 60s, 取值范围(0, 600), 设置成 0 表示死等, 只有 `LOGMNR_ATTR_TRX_END` 为 1 时才有效。

返回值

0: 成功;

<0: 错误。

9.2.1.11 获取日志分析属性

函数原型

```
dmcode_t
logmnrr_client_get_attr(
    void*          conn,
    uint           attr,
    void*          buf,
    uint           buf_len,
    uint*          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。

有如下属性可以获取：

- LOGMNR_ATTR_PARALLEL_NUM:** 并行线程数，取值范围(2, 16)；
- LOGMNR_ATTR_BUFFER_NUM:** 任务缓存节点数，取值范围(8, 1024)；
- LOGMNR_ATTR_CONTENT_NUM:** 结果缓存节点数，取值范围(256, 2048)；
- LOGMNR_ATTR_CHAR_CODE:** 本地编码（只读）；
- LOGMNR_ATTR_TRX_END:** 是否查找事务结束记录，缺省为 1，取值范围 0 或 1；
- LOGMNR_ATTR_TRX_WAIT_TIME:** 查找事务结束记录的最大等待时间，单位为 s，缺省为 60s，取值范围(0, 600)，设置成 0 表示死等，只有 LOGMNR_ATTR_TRX_END 为 1 时才有效。

返回值

0：成功；
<0：错误。

9.2.2 基本示例

数据库安装在本机上，端口号为 5236，用户名和密码均为 SYSDBA。

开启两个日志相关参数：一是开启归档（设置 INI 参数 ARCH_INI 为 1）；二是开启在日志中记录逻辑操作的功能（设置 INI 参数 RLOG_APPEND_LOGIC 为 1、2 或者 3）。

```
#include <stdio.h>
#include <stdlib.h>
#include "logmnrr_client.h"
void
main(int argc, schar* argv[])
{
```

```

lint rt;
void* conn;
logmngr_content_t** logmngr_contents;
lint real_num;
ulint char_code;

rt = logmngr_client_init();
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_init error");
    exit(-1);
}

rt = logmngr_client_create_connect("LOCALHOST", 5236, "SYSDBA", "SYSDBA",
&conn);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_create_connect error");
    exit(-1);
}

//需要添加的文件名
rt = logmngr_client_add_logfile(conn,
"d:\dmdata\arch\ARCHIVE_LOCAL1_20140725091059625.log", LOGMNR_ADDFILE);

if (rt < 0)
{
    fprintf(stderr, "logmngr_client_add_logfile error");
    exit(-1);
}

rt = logmngr_client_set_attr(conn, LOGMNR_ATTR_PARALLEL_NUM, (void*)2, 0);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_set_attr error");
    exit(-1);
}

rt = logmngr_client_start(conn, -1, NULL, NULL);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_start error");
    exit(-1);
}

```

```
rt = logmngr_client_get_attr(conn, LOGMNR_ATTR_CHAR_CODE, &char_code, 4, 0);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_set_attr error");
    exit(-1);
}

while(1)
{
    rt = logmngr_client_get_data(conn, 6, &logmngr_contents, &real_num);
    if (rt < 0)
    {
        fprintf(stderr, "logmngr_client_get_data error");
        exit(-1);
    }
    if (real_num == 0)
        break;
}

rt = logmngr_client_end(conn, 0);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_end error");
    exit(-1);
}

rt = logmngr_client_close_connect(conn);
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_close_connect error");
    exit(-1);
}

rt = logmngr_client_deinit();
if (rt < 0)
{
    fprintf(stderr, "logmngr_client_deinit error");
    exit(-1);
}
}
```

第 10 章 DM Node.js 编程指南

10.1 Node.js DM 数据库驱动介绍

由于 Node.js 没有标准的数据库接口规范，故达梦公司根据达梦数据库的特点，为开发人员提供一套 DM Node.js 数据库驱动接口，其包名为 `dmdb`。下面将详细介绍 `dmdb` 包在达梦数据库上的用法。

10.2 Node.js 驱动包安装与环境准备

1. 版本要求

Node.js 的版本至少为 v12.0.0。

2. 使用 `npm install` 命令安装 `dmdb` 包

```
> npm install dmdb
```

上述命令会自动安装所需的依赖包，如 `iconv-lite`、`snappy`。其中 `snappy` 需要 `node-gyp` 编译，如果安装过程报错，参考 <https://www.npmjs.com/package/node-gyp>。

3. 在项目中引用 `dmdb` 包

```
const db = require("dmdb");
// your code
```

10.3 对象使用

10.3.1 dmdb 对象

10.3.1.1 常量

常量名	值	描述
<code>dmdb.OUT_FORMAT_ARRAY</code>	4001	获取结果集时，用数组来获取一行数据
<code>dmdb.OUT_FORMAT_OBJECT</code>	4002	获取结果集时，用 <code>json</code> 对象来获取一行数据，键为列名
<code>dmdb.BLOB</code>	2019	数据类型， <code>Blob</code>
<code>dmdb.BUFFER</code>	2006	数据类型， <code>Buffer</code>
<code>dmdb.CLOB</code>	2017	数据类型， <code>Clob</code>
<code>dmdb.CURSOR</code>	2021	数据类型， <code>Cursor</code>
<code>dmdb.DATE</code>	2014	数据类型， <code>Date</code>
<code>dmdb.DEFAULT</code>	0	默认数据类型
<code>dmdb.NUMBER</code>	2010	数据类型， <code>Number</code>
<code>dmdb.STRING</code>	2001	数据类型， <code>String</code>

dmdb.BIND_IN	3001	参数绑定方向, 绑入
dmdb.BIND_INOUT	3002	参数绑定方向, 绑入和绑出
dmdb.BIND_OUT	3003	参数绑定方向, 绑出
dmdb.POOL_STATUS_CLOSED	6002	连接池状态, 关闭
dmdb.POOL_STATUS_DRAINING	6001	连接池状态, 正在关闭
dmdb.POOL_STATUS_OPEN	6000	连接池状态, 打开
dmdb.STATEMENT_TYPE_ALTER	7	alter 语句
dmdb.STATEMENT_TYPE_BEGIN	8	begin 语句
dmdb.STATEMENT_TYPE_CALL	10	call 语句
dmdb.STATEMENT_TYPE_COMMIT	21	commit 语句
dmdb.STATEMENT_TYPE_CREATE	5	create 语句
dmdb.STATEMENT_TYPE_DECLARE	9	declare 语句
dmdb.STATEMENT_TYPE_DELETE	3	delete 语句
dmdb.STATEMENT_TYPE_DROP	6	drop 语句
dmdb.STATEMENT_TYPE_EXPLAIN_PLAN	15	explain plan 语句
dmdb.STATEMENT_TYPE_INSERT	4	insert 语句
dmdb.STATEMENT_TYPE_MERGE	16	merge 语句
dmdb.STATEMENT_TYPE_ROLLBACK	17	rollback 语句
dmdb.STATEMENT_TYPE_SELECT	1	select 语句
dmdb.STATEMENT_TYPE_UNKNOWN	0	未知类型语句
dmdb.STATEMENT_TYPE_UPDATE	2	update 语句

10.3.1.2 属性

以下属性的修改会影响由 dmdb 创建出来的 Pool 和 Connection 对象。

属性名	类型	描述
autoCommit	Boolean	语句执行后, 是否自动提交, 缺省为 true
extendedMetaData	Boolean	查询结果集中 metadata 是否要扩展, 缺省为 false
fetchAsBuffer	Array	指定结果集中的数据类型以 Buffer 显示, 取值范围: dmdb.BLOB
fetchAsString	Array	指定结果集中的数据类型以 String 显示, 取值范围: dmdb.BUFFER、dmdb.CLOB、dmdb.DATE、dmdb.NUMBER
outFormat	Number	查询结果的格式, 取值为 dmdb.OUT_FORMAT_ARRAY 或 dmdb.OUT_FORMAT_OBJECT, 表示格式化成数组还是 json 对象, 缺省为 dmdb.OUT_FORMAT_ARRAY
poolMax	Number	最大连接数, 缺省为 4
poolMin	Number	最小连接数, 缺省为 0
poolTimeout	Number	连接闲置多久后自动关闭, 单位秒。缺省为 60
queueMax	Number	获取连接请求等待队列最大个数。取值范围 -1~9007199254740991。-1表示无限制, 缺省为 500
queueRequests	Boolean	连接池达到最大连接数后, 后续获取连接请求是否进入等待队列。可 选值 true、false。缺省为 true

queueTimeout	Number	进入等待队列的获取连接请求在等待多少毫秒后，认为请求失败，单位毫秒。取值范围 0~9007199254740991。0 表示永久等待，缺省为 60000
--------------	--------	--

10.3.1.3 函数原型

1. createPool

函数原型	功能说明
createPool(poolAttrs)	根据配置创建连接池，返回 promise 对象
createPool(poolAttrs, callback)	根据配置创建连接池，执行回调函数

参数

1) poolAttrs (Object): JS 对象，配置连接池属性，如连接串、最大连接数等。

具体如下：

属性名	类型	描述
connectString/connectionString	String	连接串。必选
poolAlias	String	连接池别名。可选
poolMax	Number	最大连接数，缺省为 4。可选
poolMin	Number	最小连接数，缺省为 0。可选
poolTimeout	Number	连接闲置多久后自动关闭，单位秒，缺省为 60，0 代表永不关闭闲置连接。可选
queueMax	Number	获取连接请求等待队列最大个数。取值范围 -1~9007199254740991。-1 表示无限制，缺省为 500。可选
queueRequests	Boolean	连接池达到最大连接数后，后续获取连接请求是否进入等待队列。可选值 true、false。缺省为 true。可选
queueTimeout	Number	进入等待队列的获取连接请求在等待多少毫秒后，认为请求失败，单位毫秒。取值范围 0~9007199254740991。0 表示永久等待，缺省为 60000。可选
testOnBorrow	Boolean	连接获取前是否验证有效性，如果连接失效，则继续取池中其他连接。可选
validationQuery	String	连接有效性检查使用的 SQL 语句，必须至少有一行结果集，缺省为“select 1;”。可选

2) callback (Function): 执行完 createPool 后的回调函数。参数如下：

回调函数参数	描述
Error error	若创建连接池失败，error 不为空
Pool pool	若创建连接池成功，pool 为函数返回的 Pool 对象

下面通过一个简单的例子来说明创建连接池。

```
// 回调函数
db.createPool({
  connectString: "dm://SYSDBA:SYSDBA@localhost:5236"
```

```

    },
    function(err, pool) {
        do_something_with_pool();
    }
)

// Promise
db.createPool({
    connectString: "dm://SYSDBA:SYSDBA@localhost:5236",
    poolMin: 1,
    poolMax: 3,
    poolAlias: "DM5236"
}).then(function(pool) {
    do_something_with_pool();
}).catch(function(err) {
    console.log(err);
})
)

```

若无特别说明，本章下文示例中使用的都是回调函数。

2. getConnection

函数原型	功能说明
getConnection(poolAlias)	从指定连接池中获取连接，返回 promise 对象
getConnection(poolAlias, callback)	从指定连接池中获取连接，执行回调函数
getConnection(connURL)	根据配置创建连接，返回 promise 对象
getConnection(connURL, callback)	根据配置创建连接，执行回调函数
getConnection(connAttrs)	根据连接属性创建连接，返回 promise 对象
getConnection(connAttrs, callback)	根据连接属性创建连接，执行回调函数

参数

- 1) poolAlias(String): 连接池别名。
- 2) connURL(String): 连接串。
- 3) connAttrs(Object): 连接属性，具体字段如下：

字段名	类型	描述
connectString	String	连接串，与 connectionString 二选一，connectString 更优先，格式 host:port 或服务名
connectionString	String	连接串，与 connectString 二选一，connectString 更优先，格式 host:port 或服务名
user	String	登录用户名
password	String	登录密码

其余字段：与连接串属性相同，参考 10.4 节
-------------------------	-------	-------

4) `callback(Function)`: 执行完 `getConnection` 后的回调函数。参数如下：

回调函数参数	描述
<code>Error error</code>	若创建连接失败， <code>error</code> 不为空
<code>Connection connection</code>	若获取连接成功， <code>connection</code> 为函数返回的 <code>Connection</code> 对象

下面通过一个简单的例子来说明创建连接。

```
db.getConnection("dm://SYSDBA:SYSDBA@localhost:5236", function(err, connection)
{
    do_something_with_connection();
})
```

3. `getPool`

函数原型	功能说明
<code>getPool(poolAlias)</code>	获取 Pool 连接池对象

参数

`poolAlias(String)`: 连接池别名。

下面通过一个简单的例子来说明获取连接池。

```
db.createPool(
{
    poolAlias: "DM5236",
    connectString: "dm://SYSDBA:SYSDBA@localhost:5236"
},
function(err, pool) {
    console.log(pool === db.getPool("DM5236")); // true
}
)
```

10.3.2 Connection 对象

10.3.2.1 函数原型

1. `close/release`

函数原型	功能说明
<code>close() / release()</code>	关闭连接，返回 promise 对象
<code>close(callback) / release(callback)</code>	关闭连接，执行回调函数

参数

`callback(Function)`: 执行完 `close/release` 后的回调函数。参数如下：

回调函数参数	描述
<code>Error error</code>	若断开连接失败， <code>error</code> 不为空

下面通过一个简单的例子来说明关闭连接。

```
conn.close(function(err) {
    console.log(err);
})
```

2. commit

函数原型	功能说明
commit()	提交事务，返回 promise 对象
commit(callback)	提交事务，执行回调函数

参数

callback(Function): 执行完 commit 后的回调函数。参数如下：

回调函数参数	描述
Error error	若提交失败，error 不为空

下面通过一个简单的例子来说明提交事务。

```
conn.commit(function(err) {
    console.log(err);
})
```

3. createLob

函数原型	功能说明
createLob(type)	创建 Lob 对象，返回 promise 对象
createLob(type, callback)	创建 Lob 对象，执行回调函数

参数

1) type(Number): Lob 对象类型，取值为 dmdb.CLOB 或 dmdb.BLOB。

2) callback(Function): 执行完 createLob 后的回调函数。参数如下：

回调函数参数	描述
Error error	若创建 Lob 失败，error 不为空
Lob lob	若创建 Lob 成功，lob 为函数返回的 Lob 对象

下面通过一个简单的例子来说明创建 LOB 对象。

```
conn.createLob(dmdb.CLOB, function(err, lob) {
    do_something_with_lob();
})
```

4. execute

函数原型	功能说明
execute(sql, [bindParams, [options]])	执行语句，返回 promise 对象
execute(sql, [bindParams, [options]], callback)	执行语句，执行回调函数

参数

1) sql(String): SQL 或 PL/SQL 语句，可包含绑定变量。

2) bindParams(Object): 绑定参数，按照名称绑定时，为 JS 对象，按照位置绑定时，为 Array 数组。可配置属性具体如下：

绑定属性	类型	描述
dir	Number	绑定的方向, 取值为 <code>dmdb.BIND_IN</code> , <code>dmdb.BIND_INOUT</code> , <code>dmdb.BIND_OUT</code> 其一, 默认 <code>dmdb.BIND_IN</code>
val	Object	变量以 IN 或 IN OUT 方式绑定时, 可设置输入值

3) `options (Object)`: 语句执行的选项, 为 JS 对象。可配置属性具体如下:

属性	类型	描述
<code>extendedMetaData</code>	<code>Boolean</code>	覆盖 <code>dmdb.extendedMetaData</code>
<code>fetchInfo</code>	<code>Object</code>	结果集列以何种 JavaScript 类型展示, 可以覆盖全局设置 <code>dmdb.fetchAsString</code> 和 <code>dmdb.fetchAsBuffer</code> . 例如: <code>fetchInfo: {</code> <code> "COL_NAME_DATE": { type: dmdb.STRING},</code> <code> "COL_NAME_ANY": { type: dmdb.DEFAULT}</code> <code>}</code> 其中 <code>type</code> 的值可以是: <code>dmdb.STRING</code> , 对于数字, 日期时间, 二进制数据, CLOB 的结果集列返回 <code>String</code> ; <code>dmdb.BUFFER</code> , 对于 BLOB 的结果集列返回 <code>Buffer</code> ; <code>dmdb.DEFAULT</code> , 覆盖 <code>dmdb.fetchAsString</code> 和 <code>dmdb.fetchAsBuffer</code> 设置, 结果集列返回原始数据;
<code>outFormat</code>	<code>Number</code>	覆盖 <code>dmdb.outFormat</code>
<code>resultSet</code>	<code>Boolean</code>	查询结果是否返回 <code>ResultSet</code> 对象, 缺省为 <code>false</code>

4) `callback (Function)`: 执行完 `execute` 后的回调函数。参数如下:

回调函数参数	描述
<code>Error error</code>	若执行语句失败, <code>error</code> 不为空
<code>Object result</code>	若执行语句成功, <code>result</code> 为函数返回结果, 包含的字段可能有: <code>metaData</code> 、 <code>implicitResults</code> 、 <code>outBinds</code> 、 <code>resultSet</code> 、 <code>rows</code> 、 <code>rowsAffected</code> 其中各字段含义如下: <code>metaData</code> : 查询语句的列信息, 当且仅当有多个结果集时为二维数组, 含如下字段: <code>name</code> : 列名 <code>dbType</code> : 列类型 <code>dbTypeName</code> : 列类型名称 <code>precision</code> : 列数据精度 <code>scale</code> : 列数据标度 <code>nullable</code> : 列是否可为空 <code>implicitResults</code> : 当有多个结果集时, 结果保存在此字段中 <code>outBinds</code> : 绑出参数 <code>resultSet</code> : 查询语句的结果集, 当执行选项的 <code>resultSet</code> 为 <code>false</code> 时, <code>resultSet</code> 为 <code>undefined</code> <code>rows</code> : 查询语句的所有结果行, 当执行选项的 <code>resultSet</code> 为 <code>true</code> 时, <code>rows</code> 为 <code>undefined</code> <code>rowsAffected</code> : DML 语句影响行数。对于非 DML 语句(如 <code>select</code> 、 <code>PL/SQL</code>),

	rowsAffected 为 undefined
--	--------------------------

下面通过一个简单的例子来说明执行语句。

```
conn.execute(
  "select name, id from sysobjects where name = :na",
  { na: { val: 'SYSDBA', dir: db.BIND_IN } },
  { resultSet: false },
  function(err, result) {
    do_something_with_result();
  }
)
```

5. executeMany

函数原型	功能说明
executeMany(sql, binds, [options])	批量执行语句，返回 promise 对象
executeMany(sql, numIterations, [options])	批量执行语句，返回 promise 对象
executeMany(sql, binds, [options], callback)	批量执行语句，执行回调函数
executeMany(sql, numIterations, [options], callback)	批量执行语句，执行回调函数

参数

- 1) sql(String): SQL 或 PL/SQL 语句，必须包含绑定变量。
- 2) binds(Array): 绑定参数，按照名称绑定时，为 JS 对象组成的 Array 数组，按照位置绑定时，为 Array 数组组成的 Array 数组。
- 3) numIterations(Number): SQL 或 PL/SQL 语句执行次数。
- 4) options(Object): 语句执行的选项，为 JS 对象。可配置属性具体如下：

属性	类型	描述
batchErrors	Boolean	为 true 时，遇到数据库报错后，仍会执行之后的数据行，默认 false
bindDefs	Object	包含 dir 字段，详细描述见 execute 方法的参数 bindParams 中的 dir 字段
dmlRowCounts	Boolean	结果中是否显示批量执行影响行数，缺省为 false

- 5) callback(Function): 执行完 executeMany 后的回调函数。参数如下：

回调函数参数	描述
Error error	若批量执行语句失败，error 不为空
Object result	若批量执行语句成功，result 为函数返回结果，包含的字段可能有：batchErrors、dmlRowCounts、outBinds、rowsAffected 其中各字段含义如下： batchError: 批量执行错误列表 dmlRowCounts: 批量执行每一次影响的行数 outBinds: 绑出参数 rowsAffected: 语句影响行数

下面通过一个简单的例子来说明 executeMany 用法。

```

conn.executeMany(
  "insert into PERSON values (:1, :2, :3)",
  [
    [101, '张三', '男'],
    [102, '李四', '女']
  ],
  {
    dmlRowCounts: true,
    batchErrors: true,
    bindDefs: [
      { dir: db.BIND_IN },
      { dir: db.BIND_IN },
      { dir: db.BIND_IN }
    ]
  },
  function(err, result) {
    do_something_with_result();
  }
)

```

6. getStatementInfo

函数原型	功能说明
getStatementInfo(sql)	分析 sql 语句信息，返回 promise 对象
getStatementInfo(sql, callback)	分析 sql 语句信息，执行回调函数

参数

1) sql(String): SQL 语句。

2) callback(Function): 执行完 getStatementInfo 后的回调函数。参数如下：

回调函数参数	描述
Error error	若分析 sql 语句信息失败，error 不为空
Object information	若分析 sql 语句信息成功，information 为函数返回结果，包含的字段有： bindNames、metaData、statementType

下面通过一个简单的例子来说明 getStatementInfo 的用法。

```

conn.getStatementInfo("select name from person where id = :i", function (err,
info) {
  do_something_with_info();
})

```

7. rollback

函数原型	功能说明
rollback()	回滚，返回 promise 对象
rollback(callback)	回滚，执行回调函数

参数

`callback(Function)`: 执行完 `rollback` 后的回调函数。参数如下:

回调函数参数	描述
<code>Error error</code>	若回滚失败, <code>error</code> 不为空

下面用一个简单的例子来说明 `rollback` 的用法。

```
conn.rollback(function (err) {
    do_something_with_err();
})
```

10.3.3 Lob 对象

10.3.3.1 属性

属性名	类型	描述
<code>type</code>	<code>Number</code>	值为 <code>dmdb.BLOB</code> 、 <code>dmdb.CLOB</code> 其一, 只读

10.3.3.2 函数原型

1. close

使用 `createLob()` 创建的 `Lob` 对象应调用 `close()` 关闭

函数原型	功能说明
<code>close()</code>	关闭 <code>Lob</code> , 返回 <code>promise</code> 对象
<code>close(callback)</code>	关闭 <code>Lob</code> , 执行回调函数

参数

`callback(Function)`: 执行 `close` 后的回调函数。参数如下:

回调函数参数	描述
<code>Error error</code>	若关闭 <code>Lob</code> 失败, <code>error</code> 不为空

下面用一个简单的示例来说明 `close()` 的用法。

```
lob.close(function(err) {
    console.log(err);
})
```

2. getData

获取 `Lob` 对象中存储的全部数据, 对于 `BLOB`, 返回为 `Buffer`; 对于 `CLOB`, 返回为 `String`

函数原型	功能说明
<code>getData()</code>	获取 <code>Lob</code> 全部数据, 返回 <code>promise</code> 对象
<code>getData(callback)</code>	获取 <code>Lob</code> 全部数据, 执行回调函数

参数

`callback(Function)`: 执行 `getData` 后的回调函数。参数如下:

回调函数参数	描述
<code>Error error</code>	若获取 <code>Lob</code> 全部数据失败, <code>error</code> 不为空
<code>String Buffer data</code>	<code>Lob</code> 存储的全部数据

下面用一个简单的示例来说明 `getData()` 的用法。

```
lob.getData(function(err, data) {
    do_something_with_data();
})
```

3. `getLength`

获取 Lob 对象的长度，对于 BLOB，返回为字节数组长度；对于 CLOB，返回为字符串字符数

函数原型	功能说明
<code>getLength()</code>	获取 Lob 长度，返回 promise 对象
<code>getLength(callback)</code>	获取 Lob 长度，执行回调函数

参数

`callback(Function)`: 执行 `getLength` 后的回调函数。参数如下：

回调函数参数	描述
<code>Error error</code>	若获取 Lob 全部数据失败， <code>error</code> 不为空
<code>Number length</code>	Lob 的长度

下面用一个简单的示例来说明 `getLength()` 的用法。

```
lob.getLength(function(err, length) {
    do_something_with_length();
})
```

10.3.4 Pool 对象

10.3.4.1 属性

属性名	类型	描述
<code>connectionsInUse</code>	<code>Number</code>	连接池中正在使用的连接数，只读
<code>connectionsOpen</code>	<code>Number</code>	连接池中已打开的连接数，只读
<code>poolAlias</code>	<code>String</code>	<code>createPool()</code> 指定的连接池别名，只读
<code>poolMax</code>	<code>Number</code>	最大连接数，只读
<code>poolMin</code>	<code>Number</code>	最小连接数，只读
<code>poolTimeout</code>	<code>Number</code>	连接闲置多久后自动关闭，单位秒，只读
<code>testOnBorrow</code>	<code>Boolean</code>	连接获取前是否验证有效性，只读
<code>validationQuery</code>	<code>String</code>	连接有效性检查使用的 SQL 语句，只读
<code>status</code>	<code>Number</code>	连接池状态，只读

10.3.4.2 函数原型

1. `close/terminate`

函数原型	功能说明
<code>close() / terminate()</code>	关闭连接池，返回 promise 对象

close(callback)/terminate(callback)	关闭连接池，执行回调函数
-------------------------------------	--------------

参数

callback(Function): 执行 close/terminate 后的回调函数。参数如下:

回调函数参数	描述
Error error	若关闭连接池失败，error 不为空

下面通过一个简单的示例来说明 close 的用法。

```
pool.close(function(err) {
  console.log(err);
})
```

2. getConnection

函数原型	功能说明
getConnection()	从连接池中获取数据库连接，返回 promise 对象
getConnection(callback)	从连接池中获取数据库连接，执行回调函数

参数

callback(Function): 执行 getConnection 后的回调函数。参数如下:

回调函数参数	描述
Error error	若获取数据库连接失败，error 不为空
Connection conn	若获取数据库连接成功，conn 为函数返回的 Connection 对象

下面用一个简单的示例来说明 getConnection 的用法。

```
pool.getConnection(function(err, conn) {
  do_something_with_conn();
})
```

10.3.5 ResultSet 对象

ResultSet 对象每次可以从数据库中获取一行或多行数据。当查询结果集很大，或不知道结果集行数时，建议使用 ResultSet 对象。

10.3.5.1 函数原型

1. close

函数原型	功能说明
close()	关闭结果集，返回 promise 对象
close(callback)	关闭结果集，执行回调函数

参数

callback(Function): 执行 close 后的回调函数。参数如下:

回调函数参数	描述
Error error	若关闭结果集失败，error 不为空

下面通过一个简单的示例来说明 close() 的用法。

```
resultSet.close(function(err) {
  do_something_with_err();
})
```

2. getRow

函数原型	功能说明
getRow()	获取结果集中的下一行，返回 promise 对象
getRow(callback)	获取结果集中的下一行，执行回调函数

参数

callback (Function): 执行 getRow 后的回调函数。参数如下：

回调函数参数	描述
Error error	若获取结果集中的下一行失败，error 不为空
Object row	若获取结果集中的下一行成功，row 为返回的结果集中的下一行

下面用一个简单的示例来说明 getRow 的用法。

```
resultSet.getRow(function(err, row) {
  do_something_with_row();
})
```

3. getRows

函数原型	功能说明
getRows (numRows)	获取结果集中的多行，返回 promise 对象
getRows (numRows, callback)	获取结果集中的多行，执行回调函数

参数

1) numRows (Number): 要获取的行数。

2) callback (Function): 执行 getRows 后的回调函数。参数如下：

回调函数参数	描述
Error error	若获取结果集中的多行失败，error 不为空
Array rows	若获取结果集中的多行成功，row 为返回的结果集中行组成的数组

下面用一个简单的示例来说明 getRows 的用法。

```
resultSet.getRows(10, function(err, rows) {
  do_something_with_rows();
})
```

4. getCount

函数原型	功能说明
getRowCount()	获取结果集总行数，返回 promise 对象
getRowCount(callback)	获取结果集总行数，执行回调函数

参数

1) callback (Function): 执行 getCount 后的回调函数。参数如下：

回调函数参数	描述
Error error	若获取结果集中的多行失败，error 不为空

Number rowCount	若获取结果集总行数成功，rowCount 为返回的结果集总行数
-----------------	---------------------------------

下面用一个简单的示例来说明 getCount 的用法。

```
resultSet.getRowCount(function(err, rowCount) {
    do_something_withRowCount();
})
```

10.4 连接串可配置属性

连接串基本格式：

```
dm://user:password@host:port[?propName1=propValue1][&propName2=propValue2]...
```

连接串中可以设置的属性及其说明见下表。

表 10.1 Node.js 连接串属性

属性名称	类型	说明
addressRemap	String	地址重定向，格式：(IP_1:PORT_1,IP_2:PORT_2)，例如： addressRemap=(192.168.0.1:5236,localhost:5237)，如果连接 192.168.0.1:5236，则实际连接到 localhost:5237。可以配置多个地址重定向，例如： addressRemap=(localhost:5236,localhost:5237)(localhost:5238,localhost:5239)，缺省为空
alwaysAllowCommit	Boolean	在自动提交开关打开时，是否允许手动提交回滚，缺省为 true
appName	String	客户端应用程序名称，缺省为空
autoCommit	Number	是否自动提交，缺省为 true
batchAllowMaxErrors	Number	continueBatchOnError 开启时，最大可容错行数，超过则停止执行，但不报错，取值范围 0~65535，缺省为 0，表示无限制
batchNotOnCall	Boolean	存储过程是否不批量执行，缺省为 false，取值范围 (true, false)
batchType	Number	批处理类型，缺省为 1，1：进行批量绑定；2：不进行批量绑定
bufPrefetch	Number	结果集 fetch 预取消息 buffer 大小；单位 KB，取值范围 32~65535。缺省为 0，表示按服务器配置。
cluster	String	当 doSwitch=2, epSelector=1 时，用于检测 DSC 集群节点故障恢复是否成功，缺省为空，取值 (DSC)
continueBatchOnError	Boolean	批量执行出错时是否继续执行，缺省为 false
columnNameUpperCase	Boolean	列名是否全部大写，缺省为 false，取值范围 (true, false)
columnNameCase	String	列名大小写显示策略，缺省为空，取值范围 (upper, lower)
compatibleMode	String	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
compress	Number	是否压缩消息。取值范围 0~2。0：不压缩；1：完全压缩；2：优化的压缩；缺省为 0
compressId	Number	压缩算法。0：ZLIB 压缩；1：SNAPPY 压缩；缺省为 0
connectTimeout	Number	连接数据库超时时间；单位 ms，取值范围 0~2147483647，0 表示无限制；缺省为 5000

doSwitch	Number	连接发生异常或一些特殊场景下的连接处理策略。取值范围 0~2。0: 关闭连接; 1: 当连接发生异常时自动切换到其他库, 无论切换成功还是失败都会报错, 用于通知上层应用进行事务执行失败时的相关处理; 2: 配合 epSelector=1 使用, 如果服务名列表前面的节点恢复了, 将当前连接切换到前面的节点上; 缺省为 0
enRsCache	Boolean	是否开启结果集缓存; 缺省为 false, 取值范围 (true, false)
epSelector	Number	服务名连接策略。0: 依次选取列表中的不同节点建立连接, 使得所有连接均匀地分布在各个节点上; 1: 选择列表中最前面的节点建立连接, 只有当前节点无法建立连接时才会选择下一个节点进行连接; 缺省为 0
escapeProcess	Boolean	是否进行语法转义处理, 缺省为 false, 取值范围 (true, false)
isBdtaRS	Boolean	是否使用列模式结果集, 缺省为 false, 取值范围 (true, false)
lobMode	Number	Lob 模式, 缺省为 1; 1: 分批缓存到本地, 2: 一次将大字段数据缓存到本地
localTimezone	Number	客户端本地时区, 单位 min, 取值范围 -779~840
logDir	String	日志等其他一些驱动过程文件生成目录, 缺省值是当前工作目录
logFlushFreq	Number	日志刷盘频率, 单位 s, 取值范围 0~2147483647, 缺省为 10
loginCertificate	String	指定登录加密用户名密码公钥所在的路径, 一旦配置即认为开启了客户端的证书加密用户名密码模式
loginDscCtrl	Boolean	服务名连接数据库时是否只选择 dsc control 节点的库, 缺省为 false, 取值范围 (true, false)
loginMode	Number	指定优先登录的服务器模式。取值范围 0~4; 0: 优先连接 PRIMARY 模式的库, NORMAL 模式次之, 最后选择 STANDBY 模式; 1: 只连接主库; 2: 只连接备库; 3: 优先连接 STANDBY 模式的库, PRIMARY 模式次之, 最后选择 NORMAL 模式; 4: 优先连接 NORMAL 模式的库, PRIMARY 模式次之, 最后选择 STANDBY 模式; 缺省为 4
loginStatus	Number	服务名方式连接数据库时只选择状态匹配的库; 取值范围 0、3~5; 0 表示不限制; 3 表示 mount 状态; 4 表示 open 状态; 5 表示 suspend 状态; 缺省为 0
logLevel	String	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), 缺省为 off, 高级别同时记录低级别的信息
maxRows	Number	结果集行数限制, 若超过上限, 则对结果集进行截断, 取值范围 0~2147483647, 缺省为 0, 表示无限制
mppLocal	Boolean	是否 MPP 本地连接, 缺省为 false, 取值范围 (true, false)
osName	String	操作系统名称, 缺省为空
rsCacheSize	Number	结果集缓存区大小, 单位 MB, 缺省为 20, 范围 1~65536
rsRefreshFreq	Number	结果集缓存检查更新的频率, 单位秒, 缺省为 10, 取值范围 0~10000; 如果设置为 0, 则不需检查更新
rwHA	Boolean	是否开启读写分离系统高可用; 取值范围 1/0 或 true/false; 缺省为 false
rwIgnoreSql	Boolean	读写分离是否忽略 sql 类型, 并按比例分发到主/备库, 缺省为 false, 取值范围 (true, false)

rwPercent	Number	分发到主库的事务占主备库总事务的百分比，取值范围 0~100，缺省为 25
rwSeparate	Boolean	是否使用读写分离系统，缺省为 false，取值范围 (false: 不使用，true: 使用)
rwStandbyRecoverTime	Number	读写分离系统备库故障恢复检测频率，单位 ms，取值范围 0~2147483647，缺省为 1000；0 表示不恢复
schema	String	指定用户登录后的当前模式，缺省为用户的默认模式
sessionTimeout	Number	会话超时时间，单位 s，取值范围 0~2147483647，缺省为 0
socketTimeout	Number	网络通讯超时时间，单位 ms。取值范围 0~2147483647，缺省为 0。0 表示无超时限制。如果配置了该参数，将导致执行耗时大于超时时间的数据库操作报错
stmtPoolSize	Number	语句句柄池大小，取值范围 0~2147483647，缺省为 15
sslPath	String	指定 ssl 加密证书文件、密钥文件、CA 证书文件的目录，目录内必须包含 client-cert.pem、client-key.pem、ca-cert.pem
svcConfPath	Number	自定义客户端配置文件 (dm_svc.conf) 的完整路径
switchInterval	Number	服务名连接数据库时，若遍历了服务名中所有库列表都未找到符合条件的库成功建立连接，等待一定时间再继续下一次遍历；单位 ms，取值范围 0~2147483647，缺省为 1000
switchTimes	Number	服务名连接数据库时，若未找到符合条件的库成功建立连接，将尝试遍历服务名中库列表的次数，取值范围 0~2147483647，缺省为 1
userRemap	String	用户名重定向，格式：(USER1,USER2)，例如： userRemap=(USER1,USER2)，如果使用 USER1 用户连接，则实际连接用户为 USER2。可以配置多个重定向，例如： userRemap=(USER1,USER2) (USER3,USER4)，缺省为空

10.5 基本示例

下面用一个具体的编程实例来展示利用 Node.js 驱动程序进行数据库操作。

```
// 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。
```

```
// 引入dmdb包
var db = require('dmdb');
var fs = require('fs');

var pool, conn;
async function example() {
  try {
    pool = await createPool();
    conn = await getConnection();
    await insertTable();
    await updateTable();
    await queryTable();
    await queryWithResultSet();
    await deleteTable();
  } catch (err) {
    console.error(err);
  }
}
```

```

    } catch (err) {
        console.log(err);
    } finally {
        try {
            await conn.close();
            await pool.close();
        } catch (err) {}
    }
}

example();

// 创建连接池
async function createPool() {
    try {
        return db.createPool({
            connectionString:
"dm://SYSDBA:SYSDBA@localhost:5236?autoCommit=false",
            poolMax: 10,
            poolMin: 1
        });
    } catch (err) {
        throw new Error("createPool error: " + err.message);
    }
}

//获取数据库连接
async function getConnection() {
    try {
        return pool.getConnection();
    } catch (err) {
        throw new Error("getConnection error: " + err.message);
    }
}

//往产品信息表插入数据
async function insertTable() {
    try {
        var sql = "INSERT INTO
production.product(name,author,publisher,publishtime,"
        +
"product_subcategoryid,productno,satetystocklevel,originalprice,nowprice,dis
count,"
        +
"description,photo,type,papertotal,wordtotal,sellstarttime,sellendtime) "
        +
"VALUES(:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,:17);";
    }
}

```

```

var blob = fs.createReadStream("c:\\\\三国演义.jpg");
await conn.execute(
    sql,
    [
        { val: "三国演义" },
        { val: "罗贯中" },
        { val: "中华书局" },
        { val: new Date("2005-04-01") },
        { val: 4 },
        { val: "9787101046121" },
        { val: 10 },
        { val: 19.0000 },
        { val: 15.2000 },
        { val: 8.0 },
        { val: "《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。" },
        { val: blob },
        { val: "25" },
        { val: 943 },
        { val: 93000 },
        { val: new Date("2006-03-20") },
        { val: new Date("1900-01-01") }
    ]
);
} catch (err) {
    throw new Error("insertTable error: " + err.message);
}
}

//修改产品信息表数据
async function updateTable() {
try {
    var sql = "UPDATE production.product SET name = :name "
        + "WHERE productid = 11;";
    // 按名称绑定变量
    return conn.execute(sql, { name: { val: "三国演义 (上)" } });
} catch (err) {
    throw new Error("updateTable error: " + err.message);
}
}

//删除产品信息表数据
async function deleteTable() {
try {
    var sql = "DELETE FROM production.product WHERE productid = 11;";
    return conn.execute(sql);
}

```

```

    } catch (err) {
        throw new Error("deleteTable error: " + err.message);
    }
}

//查询产品信息表
async function queryTable() {
    try {
        var sql = "SELECT productid,name,author,publisher,photo FROM
production.product"
        var result = await conn.execute(sql);
        var lob = result.rows[result.rows.length - 1][4];
        var buffer = await readLob(lob);
        // Lob对象使用完需关闭
        await lob.close();
        console.log(buffer);
        return result;
    } catch (err) {
        throw new Error("queryTable error: " + err.message);
    }
}

//读取数据库返回的Lob对象
function readLob(lob) {
    return new Promise(function (resolve, reject) {
        var blobData = Buffer.alloc(0);
        var totalLength = 0;
        lob.on('data', function (chunk) {
            totalLength += chunk.length;
            blobData = Buffer.concat([blobData, chunk], totalLength);
        });
        lob.on('error', function (err) {
            reject(err);
        });
        lob.on('end', function () {
            resolve(blobData);
        });
    });
}

//结果集方式查询产品信息表
async function queryWithResultSet() {
    try {
        var sql = "SELECT productid,name,author,publisher FROM
production.product";
        var result = await conn.execute(sql, [], { resultSet: true });
        var resultSet = result.resultSet;
    }
}

```

```
// 从结果集中获取一行
result = await resultSet.getRow();
while (result) {
    console.log(result);
    result = await resultSet.getRow();
}
} catch (err) {
    throw new Error("queryWithResultSet error: " + err.message);
}
}
```

第 11 章 DM Go 编程指南

11.1 Go DM 数据库驱动介绍

Go 语言标准库 (<https://golang.google.cn/pkg/database/sql/>) 提供了一系列数据库操作的标准接口。DM 数据库基于 GO 1.13 版本实现了 database/sql 包的接口，提供给开发人员作为操作 DM 数据库的 Go 语言接口。

11.2 环境准备

1. 安装 dm 驱动包

将提供的 dm 驱动包放在 GOPATH 的 src 目录下。

2. 安装依赖包

```
go get golang.org/x/text
go get github.com/golang/snappy
```

11.3 连接串属性说明

连接串基本格式：

```
dm://user:password@host:port[?propName1=propValue1][&propName2=propValue2]...
```

连接串中可以设置的属性及其说明见下表。

表 11.1 Go 连接串属性

属性名称	说明
socketTimeout	套接字超时时间，缺省为 0
escapeProcess	是否进行语法转义处理，缺省为 false；取值范围 (true, false)
autoCommit	是否自动提交，缺省为 true；取值范围 (true, false)
maxRows	结果集行数限制，若超过上限，则对结果集进行截断，取值范围 0~2147483647，缺省为 0，表示无限制
rowPrefetch	预取行数，缺省为 10
lobMode	Lob 模式，缺省为 1；1：分批缓存到本地，2：一次将大字段数据缓存到本地
stmtPoolSize	语句句柄池大小，缺省为 15
ignoreCase	是否忽略大小写，缺省为 true，取值范围 (true, false)
alwaysAllowCommit	在自动提交开关打开时，调用 rollback() 接口是否不报错，缺省为 true，取值范围 (true, false)
batchType	批处理类型，缺省为 1；1：进行批量绑定，2：不进行批量绑定
appName	客户端应用程序名称
sessionTimeout	会话超时时间，缺省为 0
sslCertPath	指定 ssl 加密证书文件的路径
sslKeyPath	指定 ssl 加密密钥文件的路径

kerberosLoginConfPath	Kerberos 认证登录配置文件路径
mppLocal	是否 MPP 本地连接, 缺省为 false; 取值范围 (true, false)
rwSeparate	是否使用读写分离系统, 缺省 false; 取值范围 (false: 不使用, true: 使用)
rwPercent	分发到主库的事务占主备库总事务的百分比, 取值范围 0~100, 缺省为 25
isBdtaRS	是否使用列模式结果集, 缺省为 false; 取值范围 (true, false)
uKeyName	Ukey 的用户名
uKeyPin	Ukey 的口令
doSwitch	连接发生异常或一些特殊场景下的连接处理策略。取值范围 0~2; 0: 关闭连接; 1: 当连接发生异常时自动切换到其他库, 无论切换成功还是失败都会报错, 用于通知上层应用进行事务执行失败时的相关处理; 2: 配合 epSelector=1 使用, 如果服务名列表前面的节点恢复了, 将当前连接切换到前面的节点上; 缺省为 0
continueBatchOnError	批量执行出错时是否继续执行; 缺省为 false; 取值范围 (true, false)
connectTimeout	连接数据库超时时间; 单位 ms, 取值范围 0~2147483647, 0 表示无限制; 缺省为 5000
columnNameUpperCase	列名转换为大写字母, 缺省为 false; 取值范围 (true, false)
rwAutoDistribute	读写分离系统事务分发是否由驱动自动管理, 缺省为 true, 取值范围 (true, 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
rwIgnoreSql	读写分离是否忽略 sql 类型, 并按比例分发到主/备库, 缺省为 false, 取值范围 (true, false)
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
dbAliveCheckFreq	检测数据库是否存活的频率, 单位 ms, 缺省为 0; 0: 不检测, 1: 检测
logDir	日志等其他一些驱动过程文件生成目录, 缺省值是当前工作目录
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), 缺省为 off, 高级别同时记录低级别的信息
logFlushFreq	日志刷盘频率, 单位 s, 缺省为 10
logBufferSize	日志缓冲区大小, 缺省为 32768
logFlusherQueueSize	日志刷盘线程中等待刷盘的日志缓冲区队列大小, 缺省为 100
statEnable	是否启用状态监控, 缺省为 false; 取值范围 (true, false)
statFlushFreq	状态监控统计信息刷盘频率, 单位 s; 缺省为 3
statSlowSqlCount	日志打印慢 sql top 行数, 缺省为 100; 取值范围 0~1000
statHighFreqSqlCount	日志打印高频 sql top 行数, 缺省为 100; 取值范围 0~1000
statSqlMaxCount	状态监控可以统计不同 sql 的个数, 缺省为 100000; 取值范围 0~100000
statSqlRemoveMode	执行的不同 sql 个数超过 statSqlMaxCount 时使用的淘汰方式, 取值范围 (latest/eldest); latest: 淘汰最近执行的 sql, eldest: 淘汰最老的 sql, 缺省为 latest

statDir	状态监控信息以文本文件形式输出的目录，无缺省值，若不指定则监控信息不会以文本文件形式输出
schema	指定用户登录后的当前模式，默认为用户的默认模式
cipherPath	第三方加密算法动态链接库位置
compress	是否压缩消息。取值范围 0~2；0：不压缩；1：完全压缩；2：优化的压缩；缺省为 0
compressId	压缩算法。0：ZLIB 压缩；1：SNAPPY 压缩；缺省为 0
svcConfPath	自定义客户端配置文件 (dm_svc.conf) 的完整路径
batchAllowMaxErrors	continueBatchOnError 开启时，最大可容错行数，超过则停止执行，但不报错，取值范围 0~65535，缺省为 0，表示无限制
batchNotOnCall	存储过程是否不批量执行，缺省为 false，取值范围 (true, false)
cluster	当 doSwitch=2, epSelector=1 时，用于检测 DSC 集群节点故障恢复是否成功，缺省为空，取值 (DSC)
columnNameCase	列名大小写显示策略，缺省为空，取值范围 (upper, lower)
epSelector	服务名连接策略。0：依次选取列表中的不同节点建立连接，使得所有连接均匀地分布在各个节点上；1：选择列表中最前面的节点建立连接，只有当前节点无法建立连接时才会选择下一个节点进行连接；缺省为 0
loginDscCtrl	服务名连接数据库时是否只选择 dsc control 节点的库，缺省为 false，取值范围 (true, false)
osName	操作系统名称，缺省为空
addressRemap	地址重定向，格式：(IP_1:PORT_1, IP_2:PORT_2)，例如： addressRemap=(192.168.0.1:5236,localhost:5237)，如果连接 192.168.0.1:5236，则实际连接到 localhost:5237。可以配置多个地址重 定向，例如： addressRemap=(localhost:5236,localhost:5237)(localhost:5238,localhost:5239)，缺省为空
userRemap	用户名重定向，格式：(USER1,USER2)，如：userRemap=(USER1,USER2)， 如果使用 USER1 用户连接，则实际连接用户为 USER2。可以配置多个重定向，如： userRemap=(USER1,USER2)(USER3,USER4)，缺省为空
switchInterval	服务名连接数据库时，若遍历了服务名中所有库列表都未找到符合条件的库成功 建立连接，等待一定时间再继续下一次遍历；单位 ms，取值范围 0~2147483647， 缺省为 1000
switchTimes	服务名连接数据库时，若未找到符合条件的库成功建立连接，将尝试遍历服务名 中库列表的次数，取值范围 0~2147483647，缺省为 1

11.4 相关方法

11.4.1 DB

DB 是一个数据库（操作）句柄，代表一个具有零到多个底层连接的连接池。

DB 的相关方法介绍如下：

1. Open

函数原型

```
func Open(driverName, dataSourceName string) (*DB, error)
```

功能说明

打开数据库。根据指定的驱动名与数据源打开数据库，此时还未真正建立数据库连接，还应调用返回值的 Ping 方法。

参数说明

driverName: 驱动名，此处应为“dm”；

dataSourceName: 数据源，参考 12.3 连接串属性说明。

返回值

*DB: 数据库(操作)句柄；

error: 错误信息，若执行成功，则值为 nil。

示例：

```
// 引用Go标准库sql和dm驱动包，后面的示例不再赘述
import (
    "database/sql"
    _ "dm"
)
db, err := sql.Open("dm", "dm://SYSDBA:SYSDBA@localhost:5236?autoCommit=true")
```

2. Ping**函数原型**

```
func (db *DB) Ping() error
```

功能说明

检查连接。检查与数据库连接是否有效，如果需要会创建连接。

参数说明

无。

返回值

error: 错误信息，若执行成功，则值为 nil。

示例：

```
err := db.Ping()
```

3. Close**函数原型**

```
func (db *DB) Close() error
```

功能说明

关闭连接。关闭数据库连接，同时释放任何打开的资源。

参数说明

无。

返回值

error: 错误信息，若执行成功，则值为 nil。

示例：

```
err := db.Close()
```

4. SetMaxOpenConns

函数原型

```
func (db *DB) SetMaxOpenConns(n int)
```

功能说明

设置最大连接数。若 n<=0，表示不限制。

参数说明

n: 最大连接数。

返回值

无。

示例:

```
db. SetMaxOpenConns(10)
```

5. SetMaxIdleConns**函数原型**

```
func (db *DB) SetMaxIdleConns(n int)
```

功能说明

设置最大闲置连接数。若 n<=0，表示不保留闲置连接。

参数说明

n: 最大闲置连接数。

返回值

无。

示例:

```
db. SetMaxIdleConns(3)
```

6. Exec**函数原型**

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
```

功能说明

执行 sql 语句。可执行查询、插入、删除、更新等操作。

参数说明

query: 要执行的 sql 语句;

args: sql 语句中的占位参数。

返回值

result: 对已执行的 sql 命令的总结;

error: 错误信息，若执行成功，则值为 nil。

示例:

```
, err := db. Exec("delete from TEST where id=?", 10)
```

7. Query**函数原型**

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

功能说明

执行查询，返回多行。一般用于执行 select 命令。

参数说明

query: 要执行的 sql 语句;

`args`: sql 语句中的占位参数。

返回值

`*Rows`: 查询结果集句柄;

`error`: 错误信息, 若执行成功, 则值为 nil。

示例:

```
rows, err := db.Query("select * from TEST")
```

8. QueryRow

函数原型

```
func (db *DB) QueryRow(query string, args ...interface{}) *Row
```

功能说明

执行查询, 返回一行。总是返回非 nil 值, 直到返回值的 `Scan` 方法被调用时, 才会返回被延迟的错误。

参数说明

`query`: 要执行的 sql 语句;

`args`: sql 语句中的占位参数。

返回值

`*Row`: 单行查询结果句柄。

示例:

```
var username string
err := db.QueryRow("select name from TEST where id=?", 10).Scan(&username)
```

9. Prepare

函数原型

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

功能说明

准备语句。创建一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

参数说明

`query`: 要准备的 sql 语句。

返回值

`*Stmt`: 语句句柄;

`error`: 错误信息, 若执行成功, 则值为 nil。

示例:

```
stmt, err := db.Prepare("select * from TEST")
```

10. Begin

函数原型

```
func (db *DB) Begin() (*Tx, error)
```

功能说明

开始事务。隔离水平由数据库驱动决定。

参数说明

无。

返回值

***Tx:** 事务句柄;
error: 错误信息，若执行成功，则值为 nil。
示例:

```
tx, err := db.Begin()
```

11.4.2 Row

`QueryRow` 方法返回 `Row`，代表单行查询结果。

`ROW` 相关方法如下：

1、`Scan`

函数原型

```
func (r *Row) Scan(dest ...interface{}) error
```

功能说明

导出查询结果。将查询结果行的各列分别保存进 `dest` 参数指定的值中。

参数说明

`dest`: 输出值句柄。

返回值

`error`: 错误信息，若执行成功，则值为 nil。

示例:

```
var username string
err := row.Scan(&username)
```

11.4.3 Rows

`Query` 方法返回 `Rows`，代表查询结果集。游标指向结果集的第零行，使用 `Next` 方法遍历各行结果。

`Rows` 相关方法如下：

1、`Columns`

函数原型

```
func (rs *Rows) Columns() ([]string, error)
```

功能说明

返回列名。若 `Rows` 已关闭，返回错误。

参数说明

无。

返回值

`[]string`: 列名数组；

`error`: 错误信息，若执行成功，则值为 nil。

示例:

```
cols, err := rows.Columns()
```

2、`Scan`

函数原型

```
func (rs *Rows) Scan(dest ...interface{}) error
```

功能说明

导出查询结果。将当前行的各列分别保存进 dest 参数指定的值中。

参数说明

dest: 输出值句柄。

返回值

error: 错误信息，若执行成功，则值为 nil。

示例:

```
var username string
err := rows.Scan(&username)
```

3、Next

函数原型

```
func (rs *Rows) Next() bool
```

功能说明

将结果集的游标指向下行。若成功，返回 true；若出现错误或没有下一行，返回 false。

参数说明

无。

返回值

bool: 游标是否成功移动到下一行。

示例:

```
success := rows.Next()
```

4、Close

函数原型

```
func (rs *Rows) Close() error
```

功能说明

关闭结果集。

参数说明

无。

返回值

error: 错误信息，若执行成功，则值为 nil。

示例:

```
err := rows.Close()
```

5、Err

函数原型

```
func (rs *Rows) Err() error
```

功能说明

返回迭代时可能的错误。需在显式或隐式调用 Close 方法后调用。

参数说明

无。

返回值

error: 错误信息，若执行成功，则值为 nil。

示例:

```
err := rows.Err()
```

11.4.4 Stmt

经过 Prepare 方法准备好的语句。

Stmt 相关方法如下：

1、Exec**函数原型**

```
func (s *Stmt) Exec(args ...interface{}) (Result, error)
```

功能说明

执行语句。参数 args 表示语句中的占位参数。

参数说明

args: sql 语句中的占位参数。

返回值

Result: 对已执行的 sql 命令的总结；

error: 错误信息，若执行成功，则值为 nil。

示例:

```
_, err := stmt.Exec()
```

2、Query**函数原型**

```
func (s *Stmt) Query(args ...interface{}) (*Rows, error)
```

功能说明

执行查询，返回多行。

参数说明

args: sql 语句中的占位参数。

返回值

*Rows: 查询结果集句柄；

error: 错误信息，若执行成功，则值为 nil。

示例:

```
rows, err := stmt.Query("select * from TEST")
```

3、QueryRow**函数原型**

```
func (s *Stmt) QueryRow(args ...interface{}) *Row
```

功能说明

执行查询，返回一行。

参数说明

args: sql 语句中的占位参数。

返回值

*Row: 查询结果行句柄。

示例:

```
var username string
```

```
err := stmt.QueryRow(10).Scan(&username)
```

4、Close

函数原型

```
func (s *Stmt) Close() error
```

功能说明

关闭准备好的语句。

参数说明

无。

返回值

`error`: 错误信息，若执行成功，则值为 nil。

示例:

```
err := stmt.Close()
```

11.4.5 Tx

`Begin` 方法返回一个进行中的数据库事务。一次事务必须以对 `Commit` 或 `Rollback` 的调用结束。

`Tx` 相关方法如下：

1、Exec

函数原型

```
func (tx *Tx) Exec(query string, args ...interface{}) (Result, error)
```

功能说明

执行语句。参数 `args` 表示 `query` 中的占位参数。

参数说明

`query`: 要执行的 sql 语句;

`args`: sql 语句中的占位参数。

返回值

`result`: 对已执行的 sql 命令的总结;

`error`: 错误信息，若执行成功，则值为 nil。

示例:

```
_, err := tx.Exec("delete from TEST where id=?", 10)
```

2、Query

函数原型

```
func (tx *Tx) Query(query string, args ...interface{}) (*Rows, error)
```

功能说明

执行查询，返回多行。

参数说明

`query`: 要执行的 sql 语句;

`args`: sql 语句中的占位参数。

返回值

`*Rows`: 查询结果集句柄;

`error`: 错误信息，若执行成功，则值为 nil。

示例:

```
rows, err := stmt.Query("select * from TEST")
```

3、QueryRow**函数原型**

```
func (tx *Tx) QueryRow(query string, args ...interface{}) *Row
```

功能说明

执行查询，返回一行。总是返回非 nil 值，直到返回值的 Scan 方法被调用时，才会返回被延迟的错误。

参数说明

query: 要执行的 sql 语句；

args: sql 语句中的占位参数。

返回值

*Row: 查询结果行句柄。

示例:

```
var username string
err := tx.QueryRow("select name from TEST where id=?", 10).Scan(&username)
```

4、Prepare**函数原型**

```
func (tx *Tx) Prepare(query string) (*Stmt, error)
```

功能说明

准备语句。创建一个准备好的语句，返回的 Stmt 在事务提交或回滚后不能再使用。

参数说明

query: 要准备的 sql 语句。

返回值

*Stmt: 准备好的语句句柄；

error: 错误信息，若执行成功，则值为 nil。

示例:

```
stmt, err := tx.Prepare("select * from TEST")
```

5、Stmt**函数原型**

```
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
```

功能说明

使用已存在的 Stmt 生成当前事务专用的 Stmt。

参数说明

stmt: 已存在的语句句柄。

返回值

*Stmt: 当前事务专用的语句句柄。

示例:

```
dbStmt := db.Prepare("select * from TEST")
.....
tx, err := db.Begin()
```

```
.....  
txStmt := tx.Statement(dbStmt)
```

6、Commit

函数原型

```
func (tx *Tx) Commit() error
```

功能说明

提交事务。

参数说明

无。

返回值

`error`: 错误信息，若执行成功，则值为 `nil`。

示例:

```
err := tx.Commit()
```

7、Rollback

函数原型

```
func (tx *Tx) Rollback() error
```

功能说明

回滚事务。

参数说明

无。

返回值

`error`: 错误信息，若执行成功，则值为 `nil`。

示例:

```
err := tx.Rollback()
```

11.5 扩展对象

类似于 `database/sql` 中的 `NullString`, 本节所有的自定义扩展对象都含有一个名为 `Valid` 的字段，用于标识该对象在数据库中的值是否为 `NULL`。如果 `Valid` 为 `false`，则表示该对象在数据库中为 `NULL`；反之不为 `NULL`。

Valid 示例:

```
var clob dm.DmClob
err := db.QueryRow("SELECT content FROM foo").Scan(&clob)
...
if clob.Valid {
    // clob不为NULL的处理
} else {
    // clob为NULL的处理
}
```

11.5.1 DmBlob

达梦数据库字节大字段

DmBlob 的相关方法介绍如下：

1. NewBlob

函数原型

```
func NewBlob(value []byte) *DmBlob
```

功能说明

构造 DmBlob 对象。

参数说明

value: DmBlob 对象存储的字节数组。

返回值

*DB: DmBlob 对象指针；

示例：

```
blob, err := dm.NewBlob([]byte{0x00, 0x01, 0x02})
```

2. Read

函数原型

```
func (blob *DmBlob) Read(dest []byte) (int, error)
```

功能说明

读取 DmBlob 对象存储的字节数组。

参数说明

dest: 读取的字节存放到 dest 中。

返回值

int: 实际读取字节数；

error: 错误信息，若执行成功，则值为 nil。

示例：

```
b := make([]byte, 100)
n, err := blob.Read(b)
```

3. ReadAt

函数原型

```
func (blob *DmBlob) ReadAt(pos int, dest []byte) (int, error)
```

功能说明

从指定偏移读取 DmBlob 对象存储的字节数组。

参数说明

pos: 偏移，范围为 1~Blob 对象长度；

dest: 读取的字节存放到 dest 中。

返回值

int: 实际读取字节数；

error: 错误信息，若执行成功，则值为 nil。

示例：

```
b := make([]byte, 100)
n, err := blob.ReadAt(1,b)
```

4. Write

函数原型

```
func (blob *DmBlob) Write(pos int, src []byte) (int, error)
```

功能说明

从指定偏移写字节数组到 DmBlob 对象。

参数说明

pos: 偏移, 范围为 1~Blob 对象长度;

src: 要写入的字节数组。

返回值

int: 实际写入字节数;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
b := []byte{0x00, 0x01, 0x02}
n, err := blob.Write(1, b)
```

5. GetLength

函数原型

```
func (blob *DmBlob) GetLength() (int64, error)
```

功能说明

获取 DmBlob 对象长度。

参数说明

无。

返回值

int64: DmBlob 对象长度;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
n, err := blob.GetLength()
```

6. Truncate

函数原型

```
func (blob *DmBlob) Truncate(length int64) error
```

功能说明

从后往前删除 DmBlob 对象直到剩余 length 个字节。

参数说明

length: 指定所剩的字节数。

返回值

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
err := blob.Truncate(0)
```

11.5.2 DmClob

达梦数据库字符大字段

DmClob 的相关方法介绍如下：

1. NewClob

函数原型

```
func NewClob(value string) *DmClob
```

功能说明

构造 DmClob 对象。

参数说明

value: DmClob 对象存储字符串。

返回值

*DB: DmClob 对象指针。

示例:

```
clob, err := dm.NewClob("hello world")
```

2. ReadString

函数原型

```
func (clob *DmClob) ReadString(pos int, length int) (string, error)
```

功能说明

读取 DmClob 对象存储的字符串。

参数说明

pos: 偏移, 范围为 1~Clob 对象长度;

length: 读取的字符串长度。

返回值

string: 实际读取字符串;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
s, err := clob.ReadString(1, 10)
```

3. WriteString

函数原型

```
func (clob *DmClob) WriteString(pos int, s string) (int, error)
```

功能说明

写入字符串到 DmClob 对象中。

参数说明

pos: 偏移, 范围为 1~Clob 对象长度;

s: 写入的字符串。

返回值

int: 实际写入字符串长度;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
n, err := clob.WriteString(1, "hello world")
```

4. GetLength

函数原型

```
func (clob *DmClob) GetLength() (int64, error)
```

功能说明

获取 DmClob 对象长度。

参数说明

无。

返回值

`int64: DmClob 对象长度;`

`error: 错误信息, 若执行成功, 则值为 nil.`

示例:

```
n, err := clob.GetLength()
```

5. Truncate**函数原型**

```
func (clob *DmClob) Truncate(length int64) error
```

功能说明

从后往前删除 DmClob 对象直到剩余 `length` 个字符。

参数说明

`length: 指定所剩的字符串长度。`

返回值

`error: 错误信息, 若执行成功, 则值为 nil.`

示例:

```
err := clob.Truncate(0)
```

11.5.3 DmTimestamp

达梦数据库日期时间对象

DmTimestamp 的相关方法介绍如下：

1. NewDmTimestampFromString**函数原型**

```
func NewDmTimestampFromString(str string) (*DmTimestamp, error)
```

功能说明

由日期时间字符串构造 DmTimestamp 对象。

参数说明

`str: 时间日期字符串。`

返回值

`*DmTimestamp: DmTimestamp 对象指针;`

`error: 错误信息, 若执行成功, 则值为 nil.`

示例:

```
dmTimestamp, err := dm.NewDmTimestampFromString("2020-01-01 11:22:33")
```

2. NewDmTimestampFromTime**函数原型**

```
func NewDmTimestampFromTime(time time.Time) *DmTimestamp
```

功能说明

由 go 原生数据类型 `time.Time` 构造 DmTimestamp 对象。

参数说明

time: go 原生数据类型 time.Time。

返回值

*DmTimestamp: DmTimestamp 对象指针。

示例:

```
dmTimestamp, err := dm.NewDmTimestampFromTime(time.Now())
```

3. ToTime**函数原型**

```
func (dmTimestamp *DmTimestamp) ToTime() time.Time
```

功能说明

将 DmTimestamp 对象转换为 go 原生数据类型 time.Time。

参数说明

无。

返回值

time.Time: go 原生数据类型 time.Time。

示例:

```
t := dmTimestamp.ToTime()
```

11.5.4 DmDecimal

达梦数据库大数字对象

DmDecimal 的相关方法介绍如下:

1. NewDecimalFromBigInt**函数原型**

```
func NewDecimalFromBigInt(bigInt *big.Int) (*DmDecimal, error)
```

功能说明

由 big.Int 对象构造 DmDecimal 对象。

参数说明

bigInt: big.Int 对象。

返回值

*DmDecimal: DmDecimal 对象指针;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
dmDecimal, err := dm.NewDecimalFromBigInt(big.NewInt(1234567890))
```

2. NewDecimalFromBigFloat**函数原型**

```
func NewDecimalFromBigFloat(bigFloat *big.Float) (*DmDecimal, error)
```

功能说明

由 big.Float 对象构造 DmDecimal 对象。

参数说明

bigFloat: big.Float 对象。

返回值

*DmDecimal: DmDecimal 对象指针。

示例:

```
dmDecimal, err := dm.NewDecimalFromBigFloat(big.NewFloat(12345.67890))
```

3. ToBigInt

函数原型

```
func (decimal DmDecimal) ToBigInt() *big.Int
```

功能说明

将 DmDecimal 对象转换为 big.Int 对象。

参数说明

无。

返回值

*big.Int: big.Int 对象指针。

示例:

```
bigInt := decimal.ToInt()
```

4. ToBigFloat

函数原型

```
func (decimal DmDecimal) ToBigFloat() *big.Float
```

功能说明

将 DmDecimal 对象转换为 big.Float 对象。

参数说明

无。

返回值

*big.Float: big.Float 对象指针。

示例:

```
bigFloat := decimal.ToBigFloat()
```

11.5.5 DmIntervalYM

达梦数据库年月间隔对象。

DmIntervalYM 的相关方法介绍如下:

1. NewDmIntervalYMByString

函数原型

```
func NewDmIntervalYMByString(str string) (*DmIntervalYM, error)
```

功能说明

由年月间隔字符串构造 DmIntervalYM 对象。

参数说明

str: 年月间隔字符串。

返回值

*DmIntervalYM: DmIntervalYM 对象指针;

error: 错误信息, 若执行成功, 则值为 nil。

示例:

```
dmIntervalYM, err := dm.NewDmIntervalYMByString("INTERVAL '10-11' YEAR TO
```

```
MONTH")
```

11.5.6 DmIntervalDT

达梦数据库日时间间隔对象。

DmIntervalDT 的相关方法介绍如下：

1. NewDmIntervalDTByString

函数原型

```
func NewDmIntervalDTByString(str string) (*DmIntervalDT, error)
```

功能说明

由日时间间隔字符串构造 DmIntervalDT 对象。

参数说明

str: 日时间间隔字符串。

返回值

*DmIntervalDT: DmIntervalDT 对象指针；

error: 错误信息，若执行成功，则值为 nil。

示例：

```
dmIntervalDT, err := dm.NewDmIntervalDTByString("INTERVAL '1 02:03:04.5678' DAY
TO SECOND")
```

11.5.7 DmArray

达梦数据库数组对象。

DmArray 的相关方法介绍如下：

1. NewDmArray

函数原型

```
func NewDmArray(typeName string, elements []interface{}) *DmArray
```

功能说明

构造 DmArray 对象。

参数说明

typeName: 数组元素的类型；

elements: 数组元素数组。

返回值

*DmArray: DmArray 对象指针。

示例：

```
dmArray := dm.NewDmArray("INT", []interface{1,2,3})
```

2. GetBaseTypeName

函数原型

```
func (dmArray *DmArray) GetBaseTypeName() (string, error)
```

功能说明

获取 DmArray 对象的基本类型名。

参数说明

无。

返回值

`string`: 基本类型名;
`error`: 错误信息, 若执行成功, 则值为 `nil`。

示例:

```
typeName, err := dmArray.GetBaseTypeName()
```

3. GetObjArray

函数原型

```
func (dmArray *DmArray) GetObjArray(index int64, count int) (interface{}, error)
```

功能说明

获取 `DmArray` 对象存储的 `Object` 数组。

参数说明

`index`: 数组开始索引;
`count`: 数组个数。

返回值

`interface{}`: `Object` 数组;
`error`: 错误信息, 若执行成功, 则值为 `nil`。

4. GetIntArray

函数原型

```
func (dmArray *DmArray) GetIntArray(index int64, count int) ([]int, error)
```

功能说明

获取 `DmArray` 对象存储的 `int` 数组。

参数说明

`index`: 数组开始索引;
`count`: 数组个数。

返回值

`[]int`: `int` 数组;
`error`: 错误信息, 若执行成功, 则值为 `nil`。

5. GetInt16Array

函数原型

```
func (dmArray *DmArray) GetInt16Array(index int64, count int) ([]int16, error)
```

功能说明

获取 `DmArray` 对象存储的 `int16` 数组。

参数说明

`index`: 数组开始索引;
`count`: 数组个数。

返回值

`[]int16`: `int16` 数组;
`error`: 错误信息, 若执行成功, 则值为 `nil`。

6. GetInt64Array

函数原型

```
func (dmArray *DmArray) GetInt64Array(index int64, count int) ([]int64, error)
```

功能说明

获取 DmArray 对象存储的 int64 数组。

参数说明

index: 数组开始索引;

count: 数组个数。

返回值

[]int64: int64 数组;

error: 错误信息, 若执行成功, 则值为 nil。

7. GetFloatArray

函数原型

```
func (dmArray *DmArray) GetFloatArray(index int64, count int) ([]float32, error)
```

功能说明

获取 DmArray 对象存储的 float32 数组。

参数说明

index: 数组开始索引;

count: 数组个数。

返回值

[]float32: float32 数组;

error: 错误信息, 若执行成功, 则值为 nil。

8. GetDoubleArray

函数原型

```
func (dmArray *DmArray) GetDoubleArray(index int64, count int) ([]float64, error)
```

功能说明

获取 DmArray 对象存储的 float64 数组。

参数说明

index: 数组开始索引;

count: 数组个数。

返回值

[]float64: float64 数组;

error: 错误信息, 若执行成功, 则值为 nil。

11.6 批量执行

Go 语言标准库 database/sql 没有提供批量执行的接口, DM 数据库驱动通过执行参数的类型判断是否批量执行。下面用一个例子展示如何批量执行:

```
// 方便起见, 忽略了所有错误返回
db, _ := sql.Open("dm", "dm://SYSDBA:SYSDBA@localhost:5236")
db.Exec("DROP TABLE IF EXISTS TEST")
db.Exec("CREATE TABLE TEST(C1 INT,C2 NUMBER,C3 VARCHAR)")
```

```

stmt, _ := db.Prepare("INSERT INTO test VALUES (?, ?, ?)")
// 批量数据必须为interface二维slice, 第一维是要插入的行数据, 第二维是每一行的列数据
var batchArg =
[][]interface{{{{1,1.1,"first"},{2,2.2,"second"},{3,3.3,"third"}}}}
// Exec执行时, 除最后一个参数外, 其他参数必须为nil
stmt.Exec(nil, nil, batchArg)

```

除了 func (s *Stmt) Exec 外, 如下方法都可以批量执行:

```

func (db *DB) Exec
func (db *DB) ExecContext
func (c *Conn) ExecContext
func (s *Stmt) Exec
func (s *Stmt) ExecContext
func (tx *Tx) Exec
func (tx *Tx) ExecContext

```

11.7 ORM 方言包

目前 GO 语言主流的 ORM 框架为 GORM。GORM 又分为 V1:github.com/jinzhu/gorm 和 V2: gorm.io/gorm 两个版本。两个版本的 GORM 互相不兼容, 为此达梦数据库提供了 2 套方言包分别适配。

方言包是位于达梦安装目录/drivers/go 目录下的 gorm_v1_dialect.zip 和 gorm_v2_dialect.zip。解压后用户需自行将其添加到包依赖路径中, 默认包名为 dm。DM ORM 方言包需要依赖 dm 驱动包。如果方言包中引用 DM 驱动包的路径不是 “dm”, 那么需要在方言包的 dialect_dm.go / dm.go 文件中重新调整引用 DM 驱动包的实际路径。

由于 GORM V1 自身对标识符是否加双引号策略不一, 所以使用 GORM V1 时, 仅推荐使用INI 参数 CASE_SENSITIVE=N 的数据库。

11.8 基本示例

例 1 下面用一个具体的编程实例来展示利用 Go 驱动程序进行数据库操作。

```

// 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。
package main
// 引入相关包
import (
    "database/sql"
    "dm"
    "fmt"
    "io/ioutil"
    "time"
)

var db *sql.DB
var err error

```

```

func main() {
    driverName := "dm"
    dataSourceName := "dm://SYSDBA:SYSDBA@localhost:5236"

    if db, err = connect(driverName, dataSourceName); err != nil {
        fmt.Println(err)
        return
    }

    if err = insertTable(); err != nil {
        fmt.Println(err)
        return
    }

    if err = updateTable(); err != nil {
        fmt.Println(err)
        return
    }

    if err = queryTable(); err != nil {
        fmt.Println(err)
        return
    }

    if err = deleteTable(); err != nil {
        fmt.Println(err)
        return
    }

    if err = disconnect(); err != nil {
        fmt.Println(err)
        return
    }
}

//创建数据库连接
func connect(driverName string, dataSourceName string) (*sql.DB, error) {
    var db *sql.DB
    var err error

    if db, err = sql.Open(driverName, dataSourceName); err != nil {
        return nil, err
    }

    if err = db.Ping(); err != nil {
        return nil, err
    }

    fmt.Printf("connect to \"%s\" succeed.\n", dataSourceName)
    return db, nil
}

```

```
//往产品信息表插入数据
func insertTable() error {
    var inFileame = "c:\\\\三国演义.jpg"
    var sql = `INSERT INTO
production.product(name,author,publisher,publishtime,
product_subcategoryid,productno,satetystocklevel,originalprice,nowprice,d
iscount,
description,photo,type,papertotal,wordtotal,sellstarttime,sellendtime)
VALUES (:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,:17);`  

    data, err := ioutil.ReadFile(inFileame)
    if err != nil {
        return err
    }
    t1, _ := time.Parse("2006-Jan-02", "2005-Apr-01")
    t2, _ := time.Parse("2006-Jan-02", "2006-Mar-20")
    t3, _ := time.Parse("2006-Jan-02", "1900-Jan-01")
    _, err = db.Exec(sql, "三国演义", "罗贯中", "中华书局", t1, 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。
",
    data, "25", 943, 93000, t2, t3)
    if err != nil {
        return err
    }
    fmt.Println("insertTable succeed")
    return nil
}

//修改产品信息表数据
func updateTable() error {
    var sql = "UPDATE production.product SET name = :name WHERE productid = 11;"
    if _, err := db.Exec(sql, "三国演义(上)"); err != nil {
        return err
    }
    fmt.Println("updateTable succeed")
    return nil
}

//查询产品信息表
func queryTable() error {
```

```

var productid int
var name string
var author string
var description dm.DmClob
var photo dm.DmBlob
var sql = "SELECT productid,name,author,description,photo FROM
production.product WHERE productid=11"
rows, err := db.Query(sql)
if err != nil {
    return err
}
defer rows.Close()

fmt.Println("queryTable results:")
for rows.Next() {
    if err = rows.Scan(&productid, &name, &author, &description, &photo);
err != nil {
        return err
    }
    blobLen, _ := photo.GetLength()
    fmt.Printf("%v %v %v %v %v\n", productid, name, author, description,
blobLen)
}
return nil
}

//删除产品信息表数据
func deleteTable() error {
    var sql = "DELETE FROM production.product WHERE productid = 11;"
    if _, err := db.Exec(sql); err != nil {
        return err
    }
    fmt.Println("deleteTable succeed")
    return nil
}

//关闭数据库连接
func disconnect() error {
    if err := db.Close(); err != nil {
        fmt.Printf("db close failed: %s.\n", err)
        return err
    }
    fmt.Println("disconnect succeed")
    return nil
}

```

```
}
```

例 2 展示 GORM V1 版本方言包用法。

```
package main
import (
    _ "dm" // DM驱动包路径
    "github.com/jinzhu/gorm"
)
type Product struct {
    gorm.Model
    Code string
    Price uint
}
func main() {
    db, err := gorm.Open("dm", "dm://SYSDBA:SYSDBA@192.168.100.165:8989")
    if err != nil {
        panic("failed to connect database")
    }
    defer db.Close()

    // Migrate the schema
    db.AutoMigrate(&Product{})

    // 创建
    db.Create(&Product{Code: "L1212", Price: 1000})

    // 读取
    var product Product
    db.First(&product, 1) // 查询id为1的product
    db.First(&product, "code = ?", "L1212") // 查询code为l1212的product

    // 更新 - 更新product的price为2000
    db.Model(&product).Update("Price", 2000)

    // 删除 - 删除product
    db.Delete(&product)
}
```

例 3 展示 GORM V2 版本方言包用法。

```
package main
import (
    "dm" // DM驱动包路径
    "gorm.io/gorm"
)
type Product struct {
```

```
gorm.Model
Code string
Price uint
}

func main() {
    db, err := gorm.Open(dm.Open("dm://SYSDBA:SYSDBA@192.168.100.165:8989"),
&gorm.Config{})
    if err != nil {
        panic("failed to connect database")
    }

    // 迁移 schema
    db.AutoMigrate(&Product{})

    // Create
    db.Create(&Product{Code: "D42", Price: 100})

    // Read
    var product Product
    db.First(&product, 1) // 根据整型主键查找
    db.First(&product, "code = ?", "D42") // 查找 code 字段值为 D42 的记录

    // Update - 将 product 的 price 更新为 200
    db.Model(&product).Update("Price", 200)
    // Update - 更新多个字段
    db.Model(&product).Updates(Product{Price: 200, Code: "F42"}) // 仅更新非零值
字段
    db.Model(&product).Updates(map[string]interface{}{"Price": 200, "Code":
"F42"})

    // Delete - 删除 product
    db.Delete(&product, 1)
}
```

第 12 章 DM XA 编程指南

本章介绍如何使用 DM XA 接口。通常，用户在使用事务处理监视器的应用程序中使用此接口。XA 功能在一个事务与多个数据库交互的应用程序中最有用。

12.1 x/Open 分布式事务处理

x/Open 分布式事务处理英文全称为 x/Open Distributed Transaction Processing，简称 x/Open DTP。

x/Open DTP 体系结构定义了一个标准体系结构或者接口，该体系结构允许多个应用程序（AP）共享由多个相同或不同的资源管理器（RM）提供的资源。它将 AP 和 RM 之间的工作协调到 XA 事务（又称全局事务）中。

DM XA 接口符合 x/Open 的 XA 接口规范。DM XA 接口是一个外部接口，它使非 DM 客户端事务管理器的客户端管理器（TM）能够协调 XA 事务，因此，DM XA 功能允许在 XA 事务中包含非 DM 数据库的其他资源管理器。

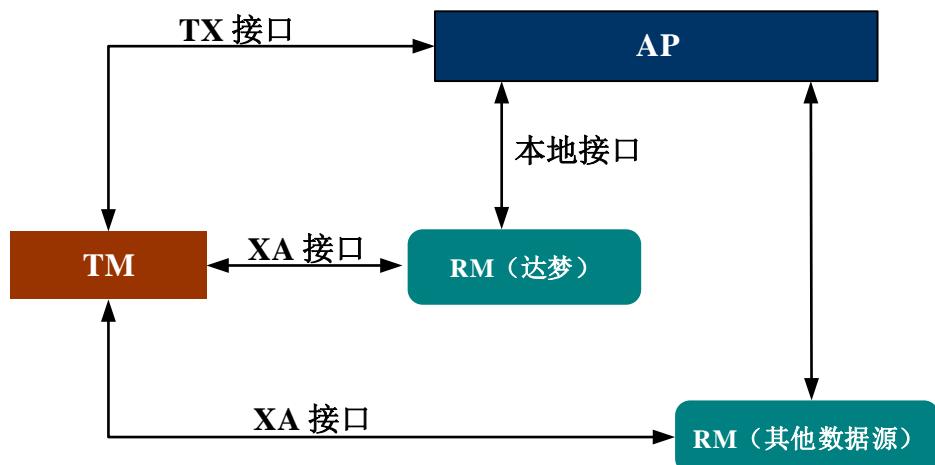


图 12.1 一个典型的 x/Open DTP 模型

12.1.1 DTP 术语

12.1.1.1 资源管理器 RM

资源管理器英文全称为 Resource Manager，简称为 RM。

资源管理器控制一个共享的、可恢复的资源，当资源管理器发生故障后，其控制的资源可被恢复到一致性状态。DM 数据库即为一个 RM，当 DM 数据库发生故障后，可以利用联机日志和回滚段将其中的数据恢复到一致性状态。

12.1.1.2 XA 事务

XA 事务又称全局事务，是一个客户端事务，涉及多个分布式资源的更新，并且需要分布式的 RM 全部成功或者全部失败。

12.1.1.3 分支

分支是一个 RM 中包含的工作单元。多个分支构成一个全局事务。对于 DM 数据库，每个分支均可映射到数据库服务器内的一个本地事务。

12.1.1.4 事务管理器 TM

事务管理器英文全称为 Transaction Manager，简称为 TM。

事务管理器提供一个 API 来指定事务的边界，并管理事务提交和恢复操作。事务管理器在分布式 RM 间实现了一个两阶段提交引擎，以保证分布式 RM 间事务的全部成功或者全部失败。

外部 TM 是驻留在 DM 数据库外部的中间层组件。通常，数据库作为其自身的内部 TM，使用基于标准的 TM 使 DM 数据库能够在单个事务中与其他异构 RM 协作。

12.1.1.5 事务处理监视器 TPM

事务处理监视器英文全称为 Transaction Processing Monitor，简称为 TPM。

TM 通常由事务处理监视器（TPM）提供，例如：Oracle Tuxedo。TPM 可以协调发出请求的客户端进程和处理请求的后端服务器之间的事务请求流。TPM 通常协调需要多种不同类型后台进程服务的事务，例如分布在网上的应用程序服务器和 RM。

TPM 可以同步完成分布式事务所需的任何提交或回滚操作。TPM 的 TM 部分负责控制执行分布式事务提交或回滚操作的时间。因此，如果分布式应用程序利用了 TPM，则 TPM 的 TM 部分将负责控制两阶段提交协议。

由于 TM 控制分布式事务提交或回滚，因此必须通过 XA 接口直接与 DM 数据库（或其他 RM）通信，TM 可利用 DM XA 接口子程序来控制 DM 数据库的事务处理过程，关于 DM XA 接口子程序的详细介绍请参考 [12.2.1 DM XA 接口子程序](#)。

12.1.1.6 两阶段提交协议

DM XA 接口遵循两阶段提交协议，各阶段内容如下：

- 在准备阶段，TM 要求每个 RM 保证可以提交事务的任何一部分，如果 RM 满足该要求，则该 RM 记录其准备状态并对 TM 进行肯定答复；否则，该 RM 可能对任务进行回滚，并对 TM 进行否定答复，之后忘记该事务。协议允许应用程序或任何 RM 单方面回滚事务，直到准备阶段完成。

- 在第二阶段，TM 记录提交决策，并向参与事务的所有 RM 发出提交或回滚请求。

其中，只有在准备阶段所有 RM 都对 TM 进行肯定答复后，TM 才能在第二阶段向所有 RM 发出提交请求。

12.1.1.7 应用程序 AP

应用程序英文全称为 Application Program，简称为 AP。

应用程序定义事务边界并指定构成事务的操作。例如，AP 可以是预编译程序。AP 通过其本机接口对 RM 资源进行操作。

12.1.1.8 TX 接口

AP 调用 TX 接口通过 TM 启动并完成所有的事务控制操作，AP 不直接调用 XA 接口。由于 AP 线程不会显式地加入、离开、挂起或恢复分支工作，而是通过 TPM 的 TM 部分来管理 AP 的全局事务分支，因此 AP 必须通过 TM 来控制分布式事务的提交或回滚操作。

12.1.1.9 本地接口

本地接口为应用程序调用资源管理器程序的接口，由各 RM 厂商提供。比如达梦提供的 ODBC、JDBC 等多种数据库访问接口。

12.1.1.10 静态登记

DM 数据库支持静态登记。RM 可在开始任何工作之前通过调用 `xa_start` 来进行静态登记，当前事务不涉及的 RM 也可进行静态登记。

12.1.2 所需公共信息

作为资源管理器，DM 数据库提供的相关信息如表 12.1 所示。

表 12.1 DM 数据库提供的相关信息

XA 特性	DM 数据库相关信息
<code>xa_switch_t</code> 结构	DM 数据库中 <code>xa_switch_t</code> 结构的名称为 <code>xadmsw</code> ，用于静态登记。该结构包含资源管理器的入口点和其他信息
<code>xa_switch_t</code> 资源管理器	DM 数据库中 <code>xa_switch_t</code> 资源管理器的名称为 <code>DM_XA</code>
打开字符串	DM 数据库中打开字符串的名称为 <code>xa_open</code> ，关于该字符串的详细介绍请参考 12.3.3 打开字符串 <code>xa_open</code>
关闭字符串	DM 数据库中关闭字符串的名称为 <code>xa_close</code> ，该字符串会被忽略，且可以为空串
库	使用 DM XA 链接应用程序所需的库，首先需要 <code>dmxai</code> 库(windows 下为 <code>dmxai.dll</code> , linux 下为 <code>libdmxai.so</code>)，如果使用 PRO*C 编程则还需要 <code>dmdpc</code> 库，如果使用 DPI 编程则还需要 <code>dmdpi</code> 库。这几个动态库在安装 DM 时，已经被放到安装目录下

要求	无
----	---

12.2 DM XA 接口

DM XA 接口使得 TM 能够控制 DM 数据库的事务处理过程。通常由 AP 调用 `tx_open` 来打开资源，但一些 TM 可能在 AP 开始时隐式调用 `xa_open` 来打开资源。

类似地，当资源使用完毕后一些 TM 可能隐式调用 `xa_close` 来关闭资源，该操作可能发生在 AP 调用 `tx_close` 时或者 AP 终止时。

TM 指示 RM 执行的任务还包括：

- 启动事务并将其与事务 ID 关联
- 回滚事务
- 准备和提交事务

12.2.1 DM XA 接口常规功能

DM XA 接口如表 12.2 所示。

表 12.2 XA 接口

XA 接口	描述
<code>xa_open</code>	连接到 RM
<code>xa_close</code>	断开与 RM 的连接
<code>xa_start</code>	启动一个事务并将其与给定的事务 ID (XID) 关联，或者将该进程与现有事务关联
<code>xa_end</code>	从给定的过程中分离
<code>xa_rollback</code>	回滚给定 XID 的事务
<code>xa_prepare</code>	准备给定 XID 的事务。这是两阶段提交协议的第一阶段
<code>xa_commit</code>	提交给定 XID 的事务。这是两阶段提交协议的第二阶段
<code>xa_recover</code>	检索已完成第一阶段预提交、第二阶段提交或已经回滚的事务。且内存中该事务信息尚未被清除的事务，已经被清除的检索不到
<code>xa_forget</code>	忽略给定 XID 的启发式完成事务

一般来说，AP 只需理解 `xa_open` 字符串的格式与功能。

12.2.2 DM XA 接口附加功能

DM 数据库的 XA 接口包括一些附加功能，如表 12.3 所示。

表 12.3 XA 接口的附加功能

功能	说明
<code>int dxa_get_con(void** con)</code>	获取当前线程上的 DPI 连接句柄

12.3 开发和安装 XA 应用程序

12.3.1 DBA 或者系统管理员的职责

DBA 或者系统管理员的职责如下：

1. 在应用程序开发人员的帮助下，定义打开字符串。关于打开字符串的详细介绍请参考 [12.3.3 打开字符串 xa_open](#)。
2. 使用打开字符串，将 RM 安装到 TPM 配置中。按照 TPM 供应商说明进行操作。
3. 向系统写入 XA 跟踪文件和日志文件的目录，并授予用户 ID 的写入权限。
4. 启动相关数据库实例以使 DM XA 应用程序处于联机状态。在启动 TPM 服务器之前执行此任务。

12.3.2 应用程序开发人员的职责

应用程序开发人员的职责如下：

1. 在 DBA 或者系统管理员的帮助下定义打开字符串。关于打开字符串的详细介绍请参考 [12.3.3 打开字符串 xa_open](#)。
2. 开发应用程序。
3. 根据 TPM 供应商的说明链接应用程序。

12.3.3 打开字符串 `xa_open`

事务监视器使用打开字符串打开数据库。

12.3.3.1 `xa_open` 字符串的语法

语法：

```
{required_field}{+required_field...}
```

示例：

```
HOST=127.0.0.1+USER=test+PWD=test12345
```

12.3.3.2 `xa_open` 字符串的必须字段

`xa_open` 字符串的必须字段如表 12.4 所示。

表 12.4 `xa_open` 字符串的必须字段

语法成分	说明
HOST	指定所要连接的 DM 数据库地址
USER	指定所要创建连接的登录用户名
PWD	指定所要创建连接的登录密码

12.4 XA 应用程序故障排除

12.4.1 访问 DM XA 的日志文件

DM XA 接口将所有错误信息记录到日志文件中，该错误信息有助于分析 XA 的出错原因。例如，通过错误信息可以得知 XA 打开失败的原因是打开字符串错误还是登录授权失败。

DM XA 接口的日志文件名称为 `dm_dmxo_date.log`，其中 `date` 表示生成该日志文件的日期。例如，`dm_dmxo_202010.log` 表示 2020 年 10 月的日志文件。DM XA 接口的日志文件默认存储于应用程序所在目录上层的 `log` 目录下。

12.4.2 访问 DM XA 的跟踪文件

DM XA 接口将 XA 接口的调用关系记录到跟踪文件中。通过分析该调用关系可以得知 XA 接口被调用的时间、调用进程名、调用线程名以及返回的结果等信息。

DM XA 接口的跟踪文件名称为 `dm_dmxo_date.trc`，其中，`date` 表示生成该跟踪文件的日期。例如，`dm_dmxo_202010.trc` 表示 2020 年 10 月的跟踪文件。DM XA 接口的跟踪文件默认存储于应用程序所在目录上层的 `log` 目录下。需要指出的是生成跟踪文件需要在 `dm_svc.conf` 文件中增加全局配置项 `XA_TRACE=(1)`。

第 13 章 msgparse 编程指南

本章介绍如何使用 msgparse 接口，它是达梦数据库的消息解析工具的 C 接口，供应用程序直接调用。接口依赖的静态库 dmmsgparse.lib 和头文件 dmmsgparse_pub.h 位于 DM 安装目录\include 目录下。

13.1 接口介绍

静态库 dmmsgparse.lib 内含 9 个函数。下面分别进行介绍。

13.1.1 获取请求数据包类型

函数原型

```
int ReqType(
    unsigned char* buf,
    int buflen,
    int* type
);
```

功能说明

对消息解析后返回消息类型。

参数说明

`buf`: 输入参数，通讯包首地址。

`buflen`: 输入参数，数据长度。

`type`: 输出参数，以 `*type` 返回数据包类型。`type` 返回内容见下表 13.1。

表 13.1 `type` 返回内容

输出消息报文的宏定义	输出消息报文的含义	类型值
DM_MSG_LOGIN	登录消息	101
DM_MSG_LOGOUT	登出消息	102
DM_MSG_STMT_ALLOCATE	分配语句句柄	103
DM_MSG_STMT_FREE	语句句柄释放	104
DM_MSG_PREPARE	准备语句	105
DM_MSG_FETCH	获取结果集	106
DM_MSG_COMMIT	提交	107
DM_MSG_ROLLBACK	回滚	108
DM_MSG_CANCEL	取消	109
DM_MSG_EXECUTE2	绑定参数	110
DM_MSG_PUT_DATA	写入大字段数据	111
DM_MSG_GET_DATA	获取数据请求	112
DM_MSG_CREATE_BLOB	创建大字段	113

DM_MSG_CLOSE_STMT	关闭语句句柄	114
DM_MSG_TIME_OUT	超时	115
DM_MSG_CURSOR_PREPARE	准备游标	116
DM_MSG_CURSOR_EXECUTE	执行游标	117
DM_MSG_EXPLAIN	查询计划	118
DM_MSG_GET_DATA_ARR	获取一组大字段的完整数据	119
DM_MSG_GET_ROW_NUM	获取行数	120
DM_MSG_GET_LOB_LEN	获取大字段长度	121
DM_MSG_GET_LOB_DATA	获取大字段数据	122
DM_MSG_MORE_RESULT	获取更多结果集	123
DM_MSG_EXEC_RETURN_AUTO_VALUE	获取 return into 值	124
DM_MSG_RESET_SESS	重置会话	125
DM_MSG_PRE_EXEC	DM_MSG_PRE_EXEC 是 DM_MSG_EXECUTE 中 PUT_DATA 时生成计划所需的命令字	126
DM_MSG_EXEC_DIRECT	常量参数化执行 sql	127
DM_MSG_STARTUP	启动	128

返回值

0: 成功。

<0: 错误。

13.1.2 检测响应数据是否为登录成功响应

函数原型

```
int IsLoggedIn(
    unsigned char* buf,
    int buflen
);
```

功能说明

判断消息是否为登录成功响应。

参数说明

buf: 通讯包首地址。

buflen: 数据长度。

返回值

1: 是。

0: 否, 表示非登录成功响应, 包括不能识别或错误的数据。

13.1.3 检测请求数据是否含有完整包长

函数原型

```
int HaveAFullReqPack(
```

```

    unsigned char* buf,
    int buflen,
    int* pPackLen
);

```

功能说明

判断消息是否有完整包长。pPackLen 不为 NULL 时，指针返回消息的总长度=数据包的消息头+数据包的消息体的长度。为 NULL 则不返回。

参数说明

buf: 输入参数，通讯包首地址。

buflen: 输入参数，数据长度。

pPackLen: 输出参数，包含完整的数据包时，以*pPackLen 返回第一个完整数据包的长度，长度为包含协议头的总长度。

返回值

0: 是。

<0: 否，或发生错误。

13.1.4 检测请求数据是否为新 SQL 指令

函数原型

```

int IsNewSQL(
    unsigned char* buf,
    int buflen
);

```

功能说明

所谓新 SQL 指令，通常是指请求数据为一次新的 SQL 请求。

相对的，比如一个 SQL 查询，结果集数量比较多，从而从通讯看，存在多次收发数据，此时不应认为是新 SQL；

消息类型是 CMD_PREPARE、CMD_EXEC_DIRECT、CMD_EXECUTE2（一次 prepare，绑定不同参数 execute）都是第一次 SQL 请求，返回 0。

参数说明

buf: 通讯包首地址。

buflen: 数据长度。

返回值

0: 是。

<0: 否，或发生错误。

13.1.5 构建错误应答报文

函数原型

```
int CrtErrResp(
    int errno,
    char* errmsg,
    unsigned char* respbuf,
    int* respbuflen
);
```

功能说明

此方法用于拦截异常连接，构建满足达梦标准的响应包，以错误形式通知到异常连接的客户端。

由于不能预先获知构建报文所需的实际大小，拟当输入参数 `respbuf` 为 NULL 时，方法计算出所需空间，通过 `* respbuflen` 返回；

调用者申请空间后，以非 NULL 的 `respbuf` 再次调用此方法，同时用 `* respbuflen` 给出空间长度，如空间长度足够时，真正生成报文。

消息体填写英文错误描述。不设置消息体中的 `crc`。（自行发送时设置）。

参数说明

`errno`: 输入参数，错误码，预定义几种错误码（需补充已知常用的错误码），用于映射为达梦标准错误码。已知常用的错误码包括成功、动态库内部错误码、异常连接错误码。部分错误码见下表 13.2。

`errmsg`: 输入参数，错误信息。以 '\0' 为结束符的 C 语言形式的 ANSI 编码的错误消息描述；如果允许，则用于填充错误内容。如果不标准，则自动更正成标准错误码后再填充错误内容。

`respbuflen`: 输入时为 `respbuf` 的有效空间，输出时为构建响应数据的真实长度。

表 13.2 部分错误码

错误码	错误描述	宏定义
0	成功	DM_EC_SUCCESS
-1	消息长度不够	DM_INVALID_LEN
-2	数据无法识别	DM_WR_MSG
-3	非法消息	DM_INVALID_MSG_TYPE
-5	缓冲区长度过小	DM_OUT_OF_MSG_BUFF
-11	客户端版本错误	DM_EC_INVALID_LEN
-12	UKEY 认证失败	DM_EC_UKEY_AUTH_MISMATCH
-13	证书验证失败	DM_EC_CONNECT_CAN_NOT_ESTABLISHED
-14	无效的用户名	DM_EC_RN_INVALID_USER_NAME
-15	解密失败	DM_EC_ENC_DECRYPT_FAIL

-16	口令失败	DM_EC_INVALID_PASSWORD
-17	请求执行过程中连接断开	DM_EC_RECV_OOB
-18	连接失败	DM_EC_CONNECT_LOST

返回值

0: 成功。

<0: 错误。

13.1.6 提取请求数据中的 SQL 和参数

函数原型

```
int ParseReq(
    unsigned char* buf,
    int buflen,
    char* sql,
    int* sqllen,
    PARAMINFO* pParams,
    int* iParam
);
```

功能说明

提取请求数据中的 SQL 和参数。

有以下几种情况：

1) sql 为 NULL 的情况下：

消息是 CMD_PREPARE 类型的只返回 SQL 长度*sqllen; CMD_EXEC_DIRECT 类型的返回 SQL 长度*sqllen。

2) pParams 为 NULL 的情况下：

消息是 CMD_EXECUTE2 类型的只返回参数个数*iParam; CMD_EXEC_DIRECT 类型的返回参数个数*iParam。

3) sql 不为 NULL 的情况下：

消息是 CMD_PREPARE 类型则只返回 sql 语句串; CMD_EXEC_DIRECT 类型返回 sql 语句串。pParams 不为 NULL, iParam 也不返回。

4) pParams 不为 NULL 的情况下：sql 不为 NULL, sqllen 也不返回。

消息是 CMD_PRE_EXEC 类型只填参数 PARAMINFO 部分信息（名称、序号、类型）、参数长度 0 内容不填。

消息是 CMD_PUT_DATA 类型，因为只有 clob 控制头信息，包括行外地址，只填参数 PARAMINFO 部分信息（类型为大字段，序号，名称不填），参数长度为实际字节个数、内容不填。CMD_PUT_DATA 之后的 CMD_EXECUTE2 消息，只填参数 PARAMINFO 部分信息（类型为大字段，序号），其他信息-1，参数长度-1 和内容不填。

消息是 CMD_EXECUTE2 类型则只填参数 PARAMINFO 信息（名称、序号、类型）、参数长度、参数内容。（消息是 CMD_EXEC_DIRECT，有可能有参数，参数还有可能是常量参数化信息。CLT_CONST_TO_PARAM 开关打开，非绑定列、绑定参数的 sql 都进

`exec_direct`。不带常量就没有参数。) 参数长度-2 表示 NULL。参数类型为-2 表示输出参数。

消息是 `CMD_EXEC_DIRECT` 类型也填上述参数信息。复合类型 (`ARRAY/ CLASS /RECORD/SARRAY`)、游标类型 `CURSOR`、罕见类型的参数数据暂不获取。参数长度是实际长度，内容不填。多行批量执行的只取第一行参数内容。

参数描述是从序号 1 开始的。参数名用英文返回。参数类型支持: `CHAR/VARCHAR2/ VARCHAR/BIT/TINYINT/SMALLINT/INT/INT64/FLOAT/DOUBLE/BINARY/ VARBINARY/BLOB/TEXT/INTERVAL_YM/INTERVAL_DT/ROWID/XDEC/DATE/TIME/ DATETIME/TIME_TZ/DATETIME_TZ/ 复合类型 / 游标类型 / 罕见类型 /TIMESTAMP/TIMESTAMP_TZ`。不支持的参数类型返回-7。下表为支持的参数类型。

表 13.3 支持的参数类型

类型代号	DM 类型	说明
1	<code>DM_DATATYPE_CHAR</code>	字符串类型。包括 <code>CHAR</code> 、 <code>VARCHAR2</code> 、 <code>VARCHAR</code>
2	<code>DM_DATATYPE_BIT</code>	整型 1 字节。包括 <code>BIT</code> 、 <code>TINYINT</code>
3	<code>DM_DATATYPE_SMALLINT</code>	整型 2 字节
4	<code>DM_DATATYPE_INT</code>	整型
5	<code>DM_DATATYPE_INT64</code>	长整型
6	<code>DM_DATATYPE_FLOAT</code>	单浮点类型
7	<code>DM_DATATYPE_DOUBLE</code>	双精度浮点类型
8	<code>DM_DATATYPE_BINARY</code>	十六进制类型。包括 <code>BINARY</code> 、 <code>VARBINARY</code>
9	<code>DM_DATATYPE_LOB</code>	大字段类型。包括 <code>BLOB</code> 、 <code>TEXT</code>
10	<code>DM_DATATYPE_IVYM</code>	间隔年月类型
11	<code>DM_DATATYPE_IVDT</code>	间隔日时类型
12	<code>DM_DATATYPE_ROWID</code>	<code>ROWID</code> 类型
13	<code>DM_DATATYPE_DEC</code>	<code>DEC</code> 类型
14	<code>DM_DATATYPE_DATE</code>	日期
15	<code>DM_DATATYPE_TIME</code>	时间
16	<code>DM_DATATYPE_DATETIME</code>	日期时间
17	<code>DM_DATATYPE_TIMETZ</code>	时间带时区
18	<code>DM_DATATYPE_DATETIME_TZ</code>	日期时间带时区
19	<code>DM_DATATYPE_COMP</code>	复合类型。包括 <code>SARRAY</code> 、 <code>ARRAY</code> 、 <code>CLASS</code> 、 <code>RECORD</code>
20	<code>DM_DATATYPE_CURSOR</code>	游标类型
21	<code>DM_DATATYPE_OTHER</code>	罕见类型。包括 <code>PLTYPE_VOID</code> 、 <code>PLTYPE_OBJECT</code> 、

		DATA_RECORD
22	DM_DATATYPE_DATETIME2	datetime2 类型
23	DM_DATATYPE_DATETIME2_TZ	DATETIME2 WITH TIME ZONE

参数说明

buf: 输入参数, 通讯包首地址。

buflen: 输入参数, 数据长度。

sql: 输出参数, 以*sql 返回解析后的 SQL 语句; 当 sql 为 NULL 时, 不真正解析, 而是通过*sqlLen 返回所需空间长度, 以及用*iParam 返回可能的参数个数。

sqlLen: 输入时为 sql 的有效空间, 输出时为 sql 数据的真实长度。

pParams: 输出参数, 以 pParams 为首地址, 返回 iParam 个参数信息。PARAMINFO 的定义为:

```
typedef struct {
    int serno;      // 参数序号
    int type;       // 用整数表示的参数类型。参数类型参见表 13.3 中的类型代号
    char name[256]; // 参数名称
    void* val;      // 参数内容, 放在*val
    int vallen;     // 参数内容有效长度.
} PARAMINFO;
```

iParam: 输入时为 pParams 的有效个数, 输出时为 pParams 的实际个数。

返回值

0: 成功。

<0: 错误。

13.1.7 展示可解析的消息格式版本

函数原型

```
Int GetMsgVer()
```

功能说明

获取消息格式的当前版本号。

参数说明

无。

返回值

可解析的消息格式的当前版本号 (固定为当前最新值 9), 宏定义为 DM_COMM_VER 9。

13.1.8 获取数据库登录请求的附加用户信息

函数原型

```
int LoginUser(
    unsigned char* buf,
    int buflen,
```

```
msg_startup_info_t* info,
msg_info_t*       msg_info
);
```

功能说明

获取登录请求消息中的附加用户信息。

参数说明

buf: 输入参数，通讯包首地址。

buflen: 输入参数，数据长度。

info: startup 输入参数，消息包含信息，STARTUP 消息通过 StartupInfo 接口解析获取。

msg_info: 输出参数，登录消息中的附加用户属性。

返回值

-1: 长度不够。

-2: 数据无法识别。

-3: 非法消息。不在 DM_MSG_LOGIN~ DM_MSG_STARTUP 范围。

0: 成功。

13.1.9 获取 STARTUP 启动的附加信息

函数原型

```
int StartupInfo(
    byte*          buf,
    lint           buflen,
    msg_startup_info_t* info
);
```

功能说明

获取 STARTUP 消息中的 UKEY 认证标记。

参数说明

buf: 输入参数，通讯包首地址。

buflen: 输入参数，数据长度。

info: 输出参数，startup 消息包含信息。

返回值

-1: 长度不够。

-2: 数据无法识别。

0: 成功。

13.1.10 提取请求消息中的信息接口

函数原型

```
lint ParseReqInfo(
    schar*          buf,
    lint            buflen,
    req_info_t*     req_info
);
```

功能说明

解析请求消息中的信息，并填充到输出参数 `req_info` 中。

参数说明

`buf`: 输入参数，通讯包首地址。

`buflen`: 输入参数，通讯包数据长度。

`req_info`: 输出参数，解析后的请求消息中的信息。结构体 `req_info_t` 用于提取请求消息，具体定义如下：

```
typedef struct req_info_struct req_info_t;
struct req_info_struct
{
    lint cmd_type; // 对应 ReqType 接口的返回值，表示请求数据包类型
    magic_declare // 用于校验结构体是否为 req_info
};
```

返回值

-1: 长度不够。

-2: 数据无法识别。

-3: 请求消息类型不合法。

0: 成功。

13.1.11 提取响应消息中的信息接口

函数原型

```
lint ParseRes(
    unsigned char*   buf,
    lint            buflen,
    req_info_t*     req_info,
    res_info_t*     res_info
);
```

功能说明

解析响应消息中的信息，并填充到输出参数 `res_info` 中。

使用该接口有以下限制与说明：

- 1) 提取响应消息中的内容，必须先调用 `ParseReqInfo` 获取对应请求的消息填充 `req_info`；然后将 `req_info` 作为输入参数给 `ParseRes` 解析对应的响应消息获取 `res_info`；若 `req_info` 与解析的响应消息不对应，则无法保证 `res_info` 正确性；
- 2) 本驱动仅能用于解析 DM 服务器与接口间明文通信消息。服务器与集群间通信格式

- 不同，无法正确识别；
- 3) 不同会话的语句句柄 ID 可能相同，因此只有语句句柄 ID 时无法区分是否为当前会话的增删改查，需要由用户自己确认拦截消息对应的会话，否则获取的影响行数数据可能错乱；
 - 4) 增、删、改响应中只有 `total_rows` 有意义，表示操作影响的总行数；`affect_rows` 无意义，恒为 0；
 - 5) 查询和获取响应中 `total_rows` 表示查询的总行数，尚未获取时填充 -1；`affect_rows` 表示当前响应消息中填充的数据行数；分批获取时，滚动游标的情况下，`affect_rows` 的累加值可能与 `total_rows` 不同；
 - 6) ParseRes 不支持分析语句块，由于语句块中的增、删、改无法统计总行数 `total_rows`。

参数说明

`buf`: 输入参数，通讯包首地址。

`buflen`: 输入参数，通讯包数据长度。

`req_info`: 输入参数，请求消息中的信息，通过 `ParseReqInfo` 接口获得。

`res_info`: 输出参数，解析后的响应消息中的信息。结构体 `res_info_t` 用于提取响应消息，具体定义如下：

```
typedef struct res_info_struct res_info_t;
struct res_info_struct
{
    int type;           // 表示响应消息类型
    uint handle;        // 表示执行的语句句柄 ID 号
    lint64 affect_rows; // 表示当前响应消息中的填充部分的数据行数
    lint64 total_rows;  // 表示当前响应消息中对应的原始请求消息中 SQL 的影响总行数
    magic_declare;     // 用于校验结构体是否为 res_info
};
```

对该结构体的属性进行说明：

`type`: 表示响应消息类型。支持的响应类型如下：

DM_RES_MSG_INSERT	1001	插入-Insert
DM_RES_MSG_DELETE	1002	删除-Delete
DM_RES_MSG_UPDATE	1003	更新-Update
DM_RES_MSG_SELECT	1004	查询-Select
DM_RES_MSG_FETCH	1005	提取-Fetch(not first select return)
DM_RES_MSG_OTHER	1006	其他-Other

`handle`: 表示执行的语句句柄 ID 号，仅对 `type` 为 1001~1005 的响应消息有效，`type` 为 1006(其他响应消息类型) 时返回 -1 (无符号整型对应 4294967295)。

`affect_rows`: 表示当前响应消息中的填充的数据行数。只对 `type` 为 1004 (SELECT) 和 1005 (FETCH) 才有效，表示当前查询的响应消息中的填充部分影响的数据行数；当 `type` 为 1001 (INSERT)、1002 (DELETE) 或 1003 (UPDATE) 时没有意义，恒返回 0；当 `type` 为 1006 (其他响应消息类型) 时，返回 -1。

`total_rows`: 表示当前响应消息中对应的原始请求消息中 SQL (增、删，改、查) 影响的总行数。`type` 为 1004 (SELECT) 和 1005 (FETCH) 时，若响应消息中没有填充影响的总行数时返回 -1，否则返回真实的影响的总行数；`type` 为 1001 (INSERT)、1002 (DELETE) 或 1003 (UPDATE) 时，响应消息中填充了影响的总行数时，返回真实的影响的总行数；`type`

为 1006(其他响应消息类型)时, 返回-1。

`magic_declare`: 用于校验结构体是否为 `res_info`。

返回值

- 1: 长度不够。
- 2: 数据无法识别。
- 0: 成功。

13.2 基本示例

开启服务器和客户端/接口之间通信的抓包工具不作介绍, 这里只介绍如何使用 `msgparse` 接口。

不同的操作系统, 用法略有不同。下面分别进行介绍。

13.2.1 Windows 环境示例

下面以 Windows 环境为例进行说明。

第一步 配置头文件。

需要用到头文件 `msgparse_pub.h`。头文件位于 `include\msgparse_pub.h`。将头文件引入工程中。

第二步 配置静态库。

需要用到的静态库为 `dmmmsgparse.lib`。静态库位于 `drivers\msgparse` 文件夹下。

配置静态库: VS2010 工程中, 右键项目名称→点击属性→选择链接器→输入→附加依赖项配置到 `drivers\msgparse\dmmmsgparse.lib`。选择常规→工作目录配置到 `drivers\msgparse\`。

第三步 编写代码, 展示函数的用法。

```
#include "msgparse_pub.h"
#include <stdio.h>

#pragma comment(lib, "dmmmsgparse.lib")

/*-----Define fun names-----*/
#define REQTYPE "ReqType" //ReqType
#define ISLOGINED "IsLoggedIn" //IsLoggedIn
#define HAVEAFULLREQPACK "HaveAFullReqPack" //HaveAFullReqPack
#define ISNEWSQL "IsNewSQL" //IsNewSQL
#define CRTERRRESP "CrtErrResp" //CrtErrResp
#define PARSEREQ "ParseReq" //ParseReq
#define GETMSGVER "GetMsgVer" //GetMsgVer

// 变量声明
const schar* proc_name;
lint type, ret;
```

```

dmcode_t          code = 0;
byte*            respbuf;
lint             resbuflen = 32767;

res_info_t       res_info;
req_info_t       req_info;

lint             pPackLen;
char*            sql = NULL;      /* INOUT: 返回解析后的 SQL 语句 */
int              sqllen;        /* INOUT: 输入时为 sql 的有效空间, 输出时为 sql 数据
的真实长度 */
PARAMINFO*       pParams = NULL;    /* INOUT: iParam 个参数的信息存储空间 */
int              iParam;        /* INOUT: 输入时为 pParams 的有效个数, 输出时为
pParams 的实际个数 */
int              i;
msg_startup_info_t info;

int test_StartupInfo(unsigned char *buf, int len) {
    int ret;
    ret = StartupInfo(buf, len, &info);
    if(ret == 0){
        printf("success!\n");
    }
    else {
        printf("ret = %d not startup message! \n");
    }
    return ret;
}

int test_LoginUser(unsigned char *buf, int len) {
    int ret;
    msg_info_t msg_login;
    byte flag = 0;
    ret = LoginUser(buf, len, &info, &msg_login);
    if(ret == 0){
        printf("%s\n", msg_login.u);
    }
    else
        printf("not login message! \n");
    return ret;
}

// 检测响应数据是否是登录成功 isLoggedIn
int test_isLoggedIn(unsigned char *buf, int len){

```

```

int ret;
ret = IsLoggedIn(buf, len);
if(ret==0) {
    printf("logined: false\n");
}
else if (ret == 1){
    printf("logined: true\n");
}
else{
    printf("logined: ret = %d, it's maybe a bug.\n", ret);
}
return ret;
}

// 测试 ReqType, 获取消息类型
int test_ReqType(unsigned char *buf, int len){
    int type, ret;
    ret = ReqType(buf, len, &type);
    if(ret != 0){
        switch(ret) {
        case DM_INVALID_LEN:
            printf("ReqType:Invalid msg length\n");
            break;
        case DM_WR_MSG:
            printf("ReqType:Wrong message.\n");
            break;
        case DM_INVALID_MSG_TYPE:
            printf("ReqType:Invalid message type.\n");
            break;
        default:
            printf("ReqType:Really really error, it's maybe a bug\n");
            break;
        }
        return ret;
    } else {
        if(type>999) {
            printf("definitely a bug: type = %d\n", type);
        }
        printf("type = %d\n", type);
    }
    return ret;
}

// 检测请求数据是否含有完整包长 HaveAFullReqPack

```

```

int test_HaveAFullReqPack(unsigned char *buf, int len, int pPackLen){
    int ret;
    ret = HaveAFullReqPack(buf, len, &pPackLen); //pPackLen
    if(ret != 0){
        printf("test_HaveAFullReqPack error, can't get packLen\n");
        return ret;
    } else {
        printf("pPackLen = %d\n", pPackLen);
    }
    return ret;
}

// 检测请求数据是否为新 SQL 指令 IsNewSQL
int test_isNewSQL(unsigned char *buf, int len){
    int ret;
    ret = IsNewSQL(buf, len);
    switch(ret) {
    case DM_INVALID_LEN:
        printf("isNewSQL:Invalid msg lenght\n");
        break;
    case DM_WR_MSG:
        printf("isNewSQL:Wrong message.\n");
        break;
    case DM_INVALID_MSG_TYPE:
        printf("isNewSQL:Invalid message type.\n");
        break;
    case DM_RN_NEW_SQL:
        printf("isNewSQL:Is not new sql.\n");
        break;
    case DM_EC_SUCCESS:
        printf("isNewSQL:Ec success.\n");
        break;
    default:
        printf("isNewSQL:Really really error, it's maybe a bug\n");
        break;
    }
    return ret;
}

// 构建错误应答报文 CrtErrResp
byte* test_crtErrResp(int i){
    int ret;
    respbuf = NULL;
    ret = CrtErrResp(i, "", respbuf, &respbuflen);
}

```

```

if(ret != 0)
{
    if(ret<0)
    {
        printf("crtErrResp: ret<0, It's OK\n");
    }
    else{
        printf("crtErrResp error, it's maybe a bug\n");
    }
} else {
    printf("respbuflen = %d\n", respbuflen);

    // 再次调用, 利用 respbuflen 构造 respbuf
    respbuf = (byte*)malloc(respbuflen);
    ret = CrtErrResp(i, "message size is out of buffer.", respbuf,
&respbuflen);

}

return respbuf;
}

// 提取请求数据中的 SQL 和参数 ParseReq
int test_ParseReq(unsigned char *buf, int len){
    int ret, code, i;

    // 第一次调用之前, sql/sqlllen/pParams, 都要给 NULL 或 0, 这样 ParseReq 函数第一次调
    // 用会返回 sqlllen 和 iParam, 用户自己利用 sqlllen 和 iParam 给 sql 和 pParams 分配空间
    sql = NULL;
    sqlllen = 0;
    pParams = NULL;
    ret = ParseReq(buf, len, sql, &sqlllen, pParams, &iParam);
    if (ret == 0)
    {
        if (sqlllen > 0)
        {
            sql = (char*)malloc(sqlllen);
        }
        if (iParam > 0)
            pParams = (PARAMINFO*)malloc(sizeof(PARAMINFO) * iParam);

        //再次调用 解析 sql 和参数信息
        ret = ParseReq(buf, len, sql, &sqlllen, pParams, &iParam); //获取
iParam,pParams
        if (ret == 0)
        {
    
```

```

code = DM_EC_INVALID_MSG;
if (sqllen > 0){
    printf("sql = %s\n", sql);
    free(sql);
}
if (iParam > 0){
    printf("paramCount = %d\n", iParam);
    for(i=0; i<iParam; i++){
        printf("pParams->type = %d\n", pParams[i].type);
        printf("pParams->name = %s\n", pParams[i].name);
        printf("pParams->lenth = %d\n", pParams[i].vallen);
        switch(pParams[i].type){
        case 1:
            printf("pParams->val = %s\n", pParams[i].val.str);
            break;
        case 2:
        case 3:
        case 4:
            printf("pParams->val = %d\n", pParams[i].val.int_val);
            break;
        case 5:
            printf("pParams->val = %lld\n",
pParams[i].val.int64_val);
            break;
        case 6:
            printf("pParams->val = %f\n",
pParams[i].val.float_val);
            break;
        case 7:
            printf("pParams->val = %lf\n",
pParams[i].val.double_val);
            break;
        case 8:
            printf("pParams->val = %s\n", pParams[i].val.bin_val);
            break;
        case 9:
            printf("pParams->val = %s\n", pParams[i].val.blob_val);
            break;
        case 10:
        case 11:
            printf("pParams->val = %s\n", pParams[i].val.intv_str);
            break;
        case 12:
            printf("pParams->val = %s\n", pParams[i].val.rowidstr);
        }
    }
}

```

```

        break;

    case 13:
        printf("pParams->val = %s\n", pParams[i].val.xdec_str);
        break;
    case 14:
    case 15:
    case 16:
    case 17:
    case 18:
        printf("pParams->val = %s\n", pParams[i].val.dt_str);
        break;
    default:
        break;
    }
    printf("\n");
}
free(pParams);
}

}

else{
    printf("test_ParseReq:ret = %d\n", ret);
    if (sqllen > 0){
        printf("sql = %s\n", sql);
        free(sql);
    }
    if (iParam > 0){
        printf("paramCount = %d\n", iParam);
        free(pParams);
    }
}
}

else if (ret != DM_NO_SQL_OR_PARAM) // 消息报错返回不是-6
{
    code = DM_EC_INVALID_MSG;
    printf("test_ParseReq:ret = %d\n", ret);
    printf("test_ParseReq: type is in (126)/(111)/(105)/(110)/(127), is type
right?\n");
}
else{ // 直接打印报错
    printf("test_ParseReq:ret = %d\n", ret);
}

return code;
}

```

```

// 展示可解析的消息格式版
int test_GetMsgVer(){
    int ret;
    ret = GetMsgVer();
    if(ret != 10){
        printf("GetMsgVer error: %d\n", ret);
    }
    return ret;
}

// 主函数
int main()
{
    // 这里放置测试程序截获的客户端发给服务器的消息
    // 此测例中的消息使用 wireshark 抓包工具获取，包头中带有其他信息，真正的消息从偏移 54
    // 位开始
    static const unsigned char pkt6[179] = {
        0xa4, 0xae, 0x12, 0x2a, 0x6e, 0xb4, 0xe0, 0x24,
        0x7f, 0xc3, 0x90, 0xa2, 0x08, 0x00, 0x45, 0x00,
        0x00, 0xa5, 0xff, 0x58, 0x40, 0x00, 0x3f, 0x06,
        0x55, 0x07, 0xc0, 0xa8, 0x64, 0x7b, 0xc0, 0xa8,
        0x01, 0x27, 0x18, 0x74, 0x43, 0xca, 0x6b, 0x6c,
        0xe9, 0x62, 0x87, 0xf4, 0x16, 0x08, 0x50, 0x18,
        0x00, 0x03, 0x05, 0x84, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x9d, 0x00, 0x3d, 0x00, 0x00, 0x00,
        0xc6, 0xf7, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x91, 0x9d, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x2d, 0x00, 0x00, 0xb5, 0xda,
        0x31, 0x20, 0xd0, 0xd0, 0xb8, 0xbd, 0xbd, 0xfc,
        0xb3, 0xf6, 0xcf, 0xd6, 0xb4, 0xed, 0xce, 0xf3,
        0x3a, 0x0a, 0xce, 0xde, 0xd0, 0xa7, 0xb5, 0xc4,
        0xb1, 0xed, 0xbb, 0xf2, 0xca, 0xd3, 0xcd, 0xbc,
        0xc3, 0xfb, 0x5b, 0x43, 0x4c, 0x4f, 0x42, 0x5f,
        0x31, 0x57, 0x5d
    };
}

```

```

// startup 消息
static const unsigned char pkt7[198] = {
    0xf4, 0x6b, 0x8c, 0x88, 0xdf, 0x62, 0xa4, 0xae,
    0x12, 0x2a, 0x6e, 0xb4, 0x08, 0x00, 0x45, 0x00,
    0x00, 0xb8, 0x53, 0x3d, 0x40, 0x00, 0x80, 0x06,
    0x00, 0x00, 0xc0, 0xa8, 0x01, 0x27, 0xc0, 0xa8,
    0x01, 0x65, 0x4d, 0xc4, 0x14, 0x74, 0x90, 0x4a,
    0x77, 0x9c, 0x37, 0xf7, 0xcf, 0xab, 0x50, 0x18,
    0x10, 0xa, 0x84, 0x87, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xc8, 0x00, 0x50, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x98, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x01, 0x02, 0x00, 0x00, 0x00, 0x00,
    0x01, 0xa, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00,
    0x00, 0x00, 0x38, 0x2e, 0x31, 0x2e, 0x30, 0x2e,
    0x30, 0x00, 0x40, 0x00, 0x00, 0x00, 0x29, 0x35,
    0x46, 0xbf, 0xff, 0x09, 0x64, 0x62, 0x91, 0x3c,
    0xc1, 0xeb, 0xc1, 0x40, 0x80, 0x52, 0x5c, 0xa0,
    0x22, 0x3c, 0x8c, 0x80, 0x1c, 0xd7, 0x5d, 0x14,
    0xe2, 0x8c, 0xc2, 0x9e, 0xf9, 0xd1, 0xa8, 0xb5,
    0x00, 0x3c, 0xae, 0x90, 0x91, 0x7d, 0xdb, 0x9c,
    0x07, 0x25, 0x85, 0x35, 0x8c, 0xb8, 0x66, 0xb3,
    0x77, 0x59, 0x07, 0x80, 0xbf, 0x60, 0x70, 0x44,
    0x8f, 0x7c, 0xb0, 0xe5, 0x56, 0xd2
};

// 登录消息
static const unsigned char pkt8[182] = {
    0xf4, 0x6b, 0x8c, 0x88, 0xdf, 0x62, 0xa4, 0xae,
    0x12, 0x2a, 0x6e, 0xb4, 0x08, 0x00, 0x45, 0x00,
    0x00, 0xa8, 0x53, 0x3f, 0x40, 0x00, 0x80, 0x06,
    0x00, 0x00, 0xc0, 0xa8, 0x01, 0x27, 0xc0, 0xa8,
    0x01, 0x65, 0x4d, 0xc4, 0x14, 0x74, 0x90, 0x4a,
    0x78, 0x2c, 0x37, 0xf7, 0xd0, 0x54, 0x50, 0x18,
    0x10, 0x09, 0x84, 0x77, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x01, 0x00, 0x40, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0xe0,
    0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

```

```

    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00,
    0x00, 0x00, 0x5b, 0xd0, 0xc5, 0x7e, 0x09, 0x00,
    0x00, 0x00, 0xd8, 0x00, 0x3c, 0x9e, 0x9b, 0x72,
    0xb0, 0x80, 0x29, 0x00, 0x00, 0x00, 0x00, 0xa,
    0x00, 0x00, 0x00, 0x57, 0x69, 0x6e, 0x64, 0x6f,
    0x77, 0x73, 0x20, 0x31, 0x30, 0x08, 0x00, 0x00,
    0x00, 0x4c, 0x56, 0x4a, 0x49, 0x41, 0x59, 0x55,
    0x45, 0x00, 0x06, 0x00, 0x75, 0x3d, 0xb2, 0xe2,
    0xca, 0xd4, 0x87, 0xed, 0x49, 0xed
};

int len = sizeof(pkt6);
byte* errorMessage = NULL;

// 展示可解析的消息格式版
test_GetMsgVer();

// 判断消息是否是登录消息
test_isLoggedIn((unsigned char*)pkt6+54, len-54);

// 获取消息类型
test_ReqType((unsigned char*)pkt6+54, len-54);

// 判断消息是否含有完整的包长
test_HaveAFullReqPack((unsigned char*)pkt6+54, len-54, pPackLen);

// 判断消息是否是一次新的sql请求
test_isNewSQL((unsigned char*)pkt6+54, len-54);

// 解析消息中的sql与参数信息
test_ParseReq((unsigned char*)pkt6+54, len-54);

// 构建错误应答报文
errorMessage = test_crtErrResp(DM_EC_UKEY_AUTH_MISMATCH);

len = sizeof(pkt7);
// 是否是ukey认证
test_StartupInfo((unsigned char*)pkt7+54, len-54);

len = sizeof(pkt8);
// 获取登录用户名
code = test_LoginUser((unsigned char*)pkt8+54, len-54);

```

```

//请求消息
static const unsigned char pkt9[] = {
/`~~~~~按实际情况插入请求信息~~~~~/  };
//响应消息
static const unsigned char pkt10[] = {
/`~~~~~按实际情况插入响应信息~~~~~/  };

int len1 = sizeof(pkt9);
//获取请求消息中的信息
code = ParseReqInfo(pkt9+54, len1-54, &req_info);

len1 = sizeof(pkt10);
//获取响应消息中的信息
code = ParseRes(pkt10+54, len1-54, &req_info, &res_info);

return code;
}

```

第四步 编译代码。

VS2010 工程中，右键项目名称，点击生成。

13.2.2 Linux 环境示例

下面以 Linux 环境为例进行说明。

第一步 编写代码，展示函数的用法。

准备文件 dm_cmpp_test_linux.c。文件内容与上面 Windows 的示例一样，只需去掉 windows 环境引用的头文件 windows.h。

第二步 编译dm_cmpp_test_linux.c文件。

使用-I 配置头文件 msgparse_pub.h 所在目录，使用-L 配置 libdmmmsgparse.so 文件所在目录。

```

gcc -g dm_cmpp_test_linux.c -o dm_cmpp_test_linux
-I/dmdbms/drivers/msgparse/include -L/dmdbms/drivers/msgparse -ldmmmsgparse
-lldl

```

第三步 执行dm_cmpp_test_linux可执行文件。

```
./dm_cmpp_test_linux
```

第 14 章 DM R2DBC 编程指南

14.1 DM R2DBC 介绍

因为使用 JDBC 来操作关系型数据库有一定局限性，JDBC 是一个完全阻塞的 API，虽然使用线程池来弥补阻塞行为，但是效果有限。

基于上述原因，SPRING 官方提出了 R2DBC。R2DBC（Reactive Relational Database Connectivity）是一项 API 规范，它声明了一个反应式 API，该方法将由数据库厂商实现来访问其关系数据库。反应式是指对突如其来的事件做出反应，使操作完成或数据可用，而不是直接阻塞。反应式 API 可以通过非阻塞的方式，提升线程的可用性，提升系统的吞吐量，且响应及时。

DM R2DBC 是使用 Java 语言编写，在 JDBC 的基础上进行的改进。它符合 SPRING 制定的异步标准。

该规范还包含以下突出的功能：

1. 集成 BLOB 和 CLOB 类型。DmBlob 和 DmClob 对 R2DBC LOB 类型的再次封装。
2. 可扩展的事务定义。通过 Connection 对象中的 releaseSavepoint()、rollbackTransaction、 rollbackTransactionToSavepoint() 进行设置。
3. 普通语句和参数化语句（PreparedStatement）。全部由 DmStatement 实现。
4. 支持存储过程/服务器侧 IN 和 OUT 参数绑定的执行函数。
5. 批量操作。通过 DmBatch 中的 add() 函数进行设置。
6. 可分类的操作异常。通过 JDBC 抛出的异常，转换为 R2DBC 相对应的异常信息。
7. 新的数据库链接 URL 方案。通过 R2DBC 官网规定 URL 串进行数据库连接。

14.2 DM R2DBC 编程步骤

利用 R2DBC 驱动程序进行编程的一般步骤为：

1. 建立连接

获取 io.r2dbc.spi.Connection 对象。利用 ConnectionFactory 工厂来获取与数据库的连接。其中，io.r2dbc.spi.Connection 是 DM8 的父类接口，DmConnection 是对这个接口的具体实现。

2. 创建 Statement 对象

创建 io.r2dbc.spi.Statement 对象。这点不同于 JDBC：JDBC 中分为 Statement 和 PreparedStatement 对象；而 R2DBC 中只有 Statement 对象，只不过 R2DBC 中的 Statement 对象包含了 JDBC 中的 Statement 和 PreparedStatement 对象。

3. 数据操作

数据操作主要分为两个方面，一个是数据库的增加、删除和修改操作；另一个是查询操作，执行完查询之后，会得到一个 io.r2dbc.spi.Result 对象，可以操控该对象获取数据库中的每行数据（Row）和元数据（RowMetadata）。

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。

14.3 DM R2DBC 特性

DM R2DBC 驱动程序依据 r2dbc-spi-1.0.0.RELEASE 版本实现。

实现了 1.0.0 发行规范所要求的下列接口和实体类：

```
io.r2dbc.spi.Assert
io.r2dbc.spi.Batch
io.r2dbc.spi.Blob
io.r2dbc.spi.Clob
io.r2dbc.spi.Closeable
io.r2dbc.spi.ColumnMetadata
io.r2dbc.spi.Connection
io.r2dbc.spi.ConnectionFactories
io.r2dbc.spi.ConnectionFactory
io.r2dbc.spi.ConnectionFactoryMetadata
io.r2dbc.spi.ConnectionFactoryOptions
io.r2dbc.spi.ConnectionFactoryProvider
io.r2dbc.spi.ConnectionMetadata
io.r2dbc.spi.ConnectionUrlParser
io.r2dbc.spi.ConstantPool
io.r2dbc.spi.DefaultLob
io.r2dbc.spi.IsolationLevel
io.r2dbc.spi.Lifecycle
io.r2dbc.spi.NonNullApi
io.r2dbc.spi.NoSuchOptionException
io.r2dbc.spi.Nullable
io.r2dbc.spi.Option
io.r2dbc.spi.OutParameterMetadata
io.r2dbc.spi.OutParameters
io.r2dbc.spi.OutParametersMetadata
io.r2dbc.spi.package-info
io.r2dbc.spi.Parameter
io.r2dbc.spi.Parameters
io.r2dbc.spi.R2dbcBadGrammarException
io.r2dbc.spi.R2dbcDataIntegrityViolationException
io.r2dbc.spi.R2dbcException
io.r2dbc.spi.R2dbcNonTransientException
io.r2dbc.spi.R2dbcNonTransientResourceException
io.r2dbc.spi.R2dbcPermissionDeniedException
io.r2dbc.spi.R2dbcRollbackException
io.r2dbc.spi.R2dbcTimeoutException
io.r2dbc.spi.R2dbcTransientException
```

```

io.r2dbc.spi.R2dbcTransientResourceException
io.r2dbc.spi.R2dbcType
io.r2dbc.spi.Readable
io.r2dbc.spi.ReadableMetadata
io.r2dbc.spi.Result
io.r2dbc.spi.Row
io.r2dbc.spi.RowMetadata
io.r2dbc.spi.Statement
io.r2dbc.spi.TransactionDefinition
io.r2dbc.spi.Type
io.r2dbc.spi.ValidationDepth
io.r2dbc.spi.Wrapped

```

14.4 DM R2DBC 扩展

为了满足 DM 的实际需求，对 R2DBC 功能进行了一定扩展。扩展了 DM 特有类型和读写分离集群下的错误信息。扩展内容和 JDBC 完全一样，具体请参考 [4.4 DM JDBC 扩展](#)。

14.5 DM R2DBC 提供的接口和对象

14.5.1 建立 R2DBC 连接对象

创建 R2DBC 连接分为两步：首先创建连接工厂，然后连接工厂使用 `create()` 方法获取连接。

14.5.1.1 创建连接工厂

创建连接工厂分为三种：一通过 JDBC 数据源创建；二通过 `Properties` 创建；三通过 R2DBC URL 创建。用户可以根据实际情况，选择其中一种即可。

14.5.1.1.1 通过 JDBC 数据源创建

具体 `DMdbDataSource` 配置请参考 [4.5.2 创建 JDBC 数据源](#)。

例 下面是使用 `DMdbDataSource` 获取 R2DBC 连接工厂。

```

public static ConnectionFactory getConnectionFactory() throws SQLException {
    DMdbDataSource dataSource = new DMdbDataSource();
    dataSource.setURL("jdbc:dm://127.0.0.1");
    dataSource.setUser("SYSDBA");
    dataSource.setPassword("SYSDBA");
    dataSource.setPort(5236);
    dataSource.setLoginTimeout(3000);
}

```

```

ConnectionFactoryOptions options = ConnectionFactoryOptions.builder()
    .option(DmConnectionFactoryProvider.DATASOURCE, dataSource)
    .option(DmConnectionFactoryProvider.CONVERTS, new DefaultConverts())
    .build();
return ConnectionFactories.find(options);
}

```

14.5.1.1.2 通过 Properties 创建

具体 Properties 配置请参考 [4.5.4 DM 扩展连接属性的使用](#)。

例 下面是使用 Properties 对象获取 R2DBC 连接工厂。

```

public static ConnectionFactory getFactoryByProperties() throws SQLException {
    Properties properties = new Properties();
    ConnectionFactoryOptions options = builder()
        //原 JDBC Properties 方法, 用以设置相关参数
        .option(DmConnectionFactoryProvider.PROPERTIES, properties)
        //原 JDBC 连接串相关参数后缀, 必须以?开头
        .option(DmConnectionFactoryProvider.POSTFIX, "?encode=utf-8")
        .option(HOST, "127.0.0.1")
        .option(PORT, 5236)
        .option(USER, "SYSDBA")
        .option(PASSWORD, "SYSDBA")
        .build();
    return ConnectionFactories.find(options);
}

```

14.5.1.1.3 通过 R2DBC URL 创建

R2DBC URL 只支持在用户名和密码中使用英文字母和数字的组合, 不支持参数的绑定。

URL 格式为“r2dbc:<数据库名称>://<用户名>:<密码>@<IP>:<Port>”。

例 下面是使用 R2DBC URL 获取 R2DBC 连接工厂

```

public static ConnectionFactory getFactoryByUrl() {
    String url = "r2dbc:dm://SYSDBA:SYSDBA@127.0.0.1:5236";
    return ConnectionFactories.get(url);
}

```

14.5.1.2 获取连接

最终, 连接工厂通过统一的 `create()` 方法获取连接。

```

Flux.usingWhen(
    connectionFactory.create(),

```

```

connection -> Flux.collect(null),
Connection::close)
.doOnNext(System.out::println)
.blockLast(Duration.ofSeconds(10));

```

14.5.2 Statement 接口

Statement 接口是 R2DBC 的标准接口，定义了 DmStatement 对象中的相关函数，DmStatement 对象为 DM8 的具体实现。

R2DBC Statement 对象用于将 SQL 语句发送到数据库服务器。在 R2DBC 中有且仅有 Statement 一种语句对象，其底层会自动判断相关语句对数据库进行操作。

例 Statement 接口底层使用 DmStatement 对象。

```

public void createStmt() {
    ConnectionFactory factory = getFactoryByUrl();
    Connection conn = Mono.from(factory.create()).block(Duration.ofSeconds(60L));
    Statement statement = conn.createStatement("select * from test1 where id = 1");
}

```

14.5.3 DmStatement 对象

DmStatement 对象是 R2DBC Statement 接口的具体实现。

DmStatement (java.sql.Connection connection, String sql, Converts converts): 初始化 DmStatement。

DmStatement (java.sql.Connection connection, String sql, Converts converts, List<String> parameterNames): 初始化 DmStatement。

execute(): 执行 Statement 句柄。

bind(int index, Object value) : 根据游标位置，对 SQL 语句绑定参数。

bind(String name, Object value): 根据名称，对 SQL 语句绑定参数。

bindNull(String name, Class<?> type) : 对 SQL 语句绑定特定类型的空参。

createResult (PreparedStatement stmt, ResultSet resultSet, int[] affectedRows): 创建 R2DBC 结果集。

createResultProcess(CallableStatement stmt): 创建 R2DBC 的存储过程结果。

ReturnGeneratedValues (String...columns): 对插入 SQL，获取特定列结果数据。

14.5.4 DmConnection 对象

DmConnection 对象表示一个 DM 数据库打开的连接。

公共属性

`beginTransaction()`: 开始一个新的事务。

`beginTransaction(TransactionDefinition definition)` : 根据输入参数 `definition` 指定的隔离等级，开始一个新的事务。

`close()`: 关闭连接对象，清理占用的资源。

`commitTransaction()`: 提交当前事务。

`createBatch()`: 创建一个批量对象，此对象用来构建批量的 SQL 请求。

`createSavepoint(String name)` : 创建一个基于当前 Statement 的保存点，并以 `name` 命名。

`createStatement(String sql)` : 创建一个 Statement 语句，包含了请求的 SQL。

`getMetadata()`: 返回当前数据库连接的元数据。

`getTransactionIsolationLevel()`: 获取当前数据库连接的隔离等级。

`isAutoCommit()`: 返回当前数据库连接的事务是否为自动提交模式。

`releaseSavePoint(String name)` : 释放当前事务中该名称 (`name`) 的保存点。

`rollbackTransaction()`: 回滚当前事务。

`rollbackTransactionToSavepoint(String name)` : 回滚到当前事务的特定名称的保存点。

`setAutoCommit(boolean autoCommit)` : 根据布尔值设置当前数据库连接是否为自动提交事务。

`setLockWaitTimeOut(Duration timeout)` : 配置要使用当前连接执行语句的锁获取超时。

`setStatementTimeout(Duration timeout)` : 配置要使用当前连接执行的 Statement 语句最大等待时间

`setTransactionIsolationLevel(IsolationLevel isolationLevel)` : 配置当前数据库连接的隔离等级。

`validate(Validation depth)` : 达梦数据库暂不支持。

14.5.5 DmBatch 对象

DmBatch 是 R2DBC 批量操作 SQL 语句的对象。不支持在 SQL 语句中使用动态绑定参数。

`add(String sql)` : 添加一条 sql 语句至当前 Batch 对象。
`execute()`: 执行 Batch 对象中所有的 Statement 句柄并返回所有结果。
 例 批量绑定 SQL 语句，一条一条绑定。

```
public void createBatch() {
    ConnectionFactory factory = getFactoryByUrl();
    Connection conn = Mono.from(factory.create()).block(Duration.ofSeconds(60L));

    Publisher<? extends Result> execute = conn.createBatch()
        .add("select 1 from dual")
        .add("select 2 from dual")
        .add("select 3 from dual")
        .execute();
}
```

14.5.6 Blob 对象

Blob 是存储二进制大文件的对象。

`discard()`: 在未订阅流内容时，释放 Blob 持有的任何资源。

`from(Publisher<ByteBuffer> p)` : 把有发布者类包装好的 ByteBuffer 对象转变为 Blob 对象。

`stream()`: 把 Blob 对象重新转变为由发布者类包装好的 ByteBuffer 对象。

14.5.7 Clob 对象

Clob 是存储 Char 类型大文件的对象。

`discard()`: 在未订阅流内容时，释放 Clob 持有的任何资源。

`from(Publisher<? extends CharSequence> p)` : 把有发布者类包装好的 CharSequence 对象转变为 clob 对象。

`stream()`: 把 Clob 对象重新转变为由发布者类包装好的 CharSequence 对象。

14.5.8 DmColumnMetaData 对象

继承 `ReadableMetadata` 的接口对象。表示查询返回的结果列的元数据。除 `ReadableMetadata` 之外的所有方法的实现。R2DBC 框架根据下述 `get` 函数，可以自动将数据库类型映射到 Java 类型，例如数据库中 `VARCHAR` 类型会转变为 Java 中的 `String` 类型。

`getJavaType()`: 获取该字段在 Java 中对应的类型名称（包含包名，类名）。

`getName()`: 获取该字段在数据库中的名称。

`getNativeTypeMetadata()`: 获取该字段在数据库中的元数据类型名称。

`getNullability()`: 获取该字段是否支持 Null 值。

`getPrecision()`: 获取该字段的精确度。

`getScale()`: 获取该字段的刻度。

`getType()`: 获取该字段在 Java 中的类型名称（仅显示类名）。

14.5.9 DmConnectionFactory 对象

`create()`: 创建 R2DBC 对于数据库的连接对象 (`Connection`)。

`getMetadata()`: 返回有关此 `ConnectionFactory` 适用产品的元数据信息。

14.5.10 DmConnectionFactoryMetadata 对象

`getName()`: 返回 `ConnectionFactory` 能够连接的产品名称。

14.5.11 DmConnectionFactoryProvider 对象

`create (ConnectionFactoryOption connectionFactoryOptions):` 通过上层解析的 Option 参数 (Option 的使用请参考 [14.5.1.1 创建连接工厂](#)), 作用为传递数据库连接相关参数, 例如用户名, 密码, 端口等等), 创建一个新的连接工厂。

`getDriver():` 返回使用的 Driver 名称。

`Supports (ConnectionFactoryOptions connectionFactoryOptions):` 判断是否支持构建连接工厂的参数。

14.5.12 DmConnectionMetadata 对象

`getDatabaseProductName():` 获取数据库产品名称。

`getDatabaseVersion():` 获取数据库产品的版本号。

14.5.13 DmR2dbcType 对象

继承 Type 类型, 扩展时间 Interval 类型。

`getJavaType():` 获取 Java 类型 Class。

`getName():` 获取类型的名称。

14.5.14 DmResult 对象

DmResult 对象是用来获取数据库返回的结果。

14.5.14.1 DmResult

DmResult 用于接收数据库返回的数据并以 Java 的形式展现, 无需手动初始化, R2DBC 会自动对其进行填充。

`DmResult(Flux<DmRow> rows, Mono<RowMetadata> rowMetadata, int[] affectedRows) :` 初始化 DmResult 对象。

14.5.14.2 filter

`filter(Predicate<Result.Segment> filter):` 根据过滤条件, 过滤结果集。

14.5.14.3 flatMap

段映射是指把数据库读取的默认类型转换为别的类型。例如: Varchar 默认对应 Java 的 String 类型, 于是通过 flatMap 把 String 类型再转换为 Char[] 数组类型)。

`flatMap(Function<Result.Segment, ? extends Publisher<?Extends T>>`

`mappingFunction`): 获取作为数据库查询结果的结果段映射。

14.5.14.4 `getRowsUpdated`

`getRowsUpdated()`: 获取数据库更新的行数。

14.5.14.5 `map`

`map` 的作用和 `flatMap` 类似(参考 1.5.13.3),但 `map` 转换后不会被 `publisher`(发布者类型)包裹。

`map(BiFunction<Row, RowMetadata, ? extends T> mappingFunction)`:
获取结果集中行数据结果的映射。

`map(Function<? super Readable, ? extends T> mappingFunction)`:
获取结果集中行数据/存储过程 Out 数据的映射。

14.5.15 DmRow 对象

实现了 `Row`、`Result`、`RowSegment` 接口。

`DmRow(RowMetadata rowMetadata, Map<Integer, Object> values, Converts converts)`: 初始化 `DmRow` 对象。

`row()`: 返回当前 `DmRow` 对象。

`getMetadata()`: 获取当前 `Row` 对象的元数据对象。

`get(int index)`: 根据游标获取当前行数据中特定位置的数据。

`getConverts()`: 获取当前 `Row` 对象的类型映射对象。

`getRowMetadata()`: 获取当前 `Row` 对象的元数据对象。

`setRowMetadata(RowMetadata rowMetadata)`: 设置 `Row` 对象的元数据对象。

`getValues()`: 以 `Map` 形式获取 `Row` 对象的结果集。

`setValues(Map<Integer, Object> values)`: 以 `Map` 形式设置 `Row` 对象的结果集。

`get(int index, Class<T> type)`: 以特定类型获取 `Row` 结果集中特定游标位置的数据。

`get(String name)`: 以名称获取 `Row` 结果集中的数据。

`get(String name, Class<T> type)`: 以特定类型获取 `Row` 结果集中特定列名称的数据。

14.5.16 DmRowMetadata 对象

用于接收数据库 `Result` 中每行数据的元数据信息。

`Of(Result result, Converts converts)`: 获取新的 `DmRowMetadata` 对象。

`DmRowMetadata(List<ColumnMetadata> columnMetaDatas)`: 根据列元数据信息, 初始化 `DmRowMetadata` 对象。

`getColumnMetadata(int index)`: 获取特定游标位置的列元数据信息。

`getColumnMetadata(String name)`: 获取特定名称的列元数据信息。
`contains(String columnName)`: 判断是否存在该名称的列数据。
`getColumnMetadatas()`: 获取列元数据信息列表。

14.6 基本示例

准备工作：相关 JAR 包在 DM8 数据库安装目录\source\drivers\r2dbc 中，其中包含 dm-r2dbc-1.0.0.jar、lib 文件夹和 readme.txt。

编程时须在项目中导入 dm-r2dbc-1.0.0.jar 和 lib 中相关依赖。

下面用一个具体的实例来展示如何利用 DM R2DBC 驱动程序进行编程并操作数据库。

```
package io.r2dbc.jdbc;

import cn.hutool.core.util.StrUtil;
import dm.jdbc.driver.DmdbArray;
import dm.jdbc.driver.DmdbDataSource;
import dm.jdbc.driver.DmdbRowId;
import io.r2dbc.jdbc.convert.DefaultConverts;
import io.r2dbc.jdbc.utils.R2dbcUtils;
import io.r2dbc.spi.Blob;
import io.r2dbc.spi.Clob;
import io.r2dbc.spi.Connection;
import io.r2dbc.spi.Statement;
import io.r2dbc.spi.*;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.nio.ByteBuffer;
import java.sql.*;
import java.time.Duration;
import java.util.Properties;

import static io.r2dbc.spi.ConnectionFactoryOptions.*;
import static org.junit.Assert.assertTrue;

public class Main
{
    public static void main(String[] args) throws SQLException
    {
        // query 方法
        query("select c from test");
        // rollback 方法
        rollback();
        // savePoint 方法
    }
}
```

```

        savePoint("hello world");
        // procedure 方法
        procedure();
        // batch 方法
        batch();
    }

    public static void query1(String sql)
    {
        if (StrUtil.isBlank(sql))
        {
            throw new NullPointerException("sql cannot be blank!");
        }

        ConnectionFactory connectionFactory = getFactoryByUrl();
        Flux.usingWhen(connectionFactory.create(),
            connection -> Flux.from(
                connection.createStatement(sql).bind(0,           100).bind(1,
101).bind(2, 102).execute()
                    .flatMap(result -> result.map(((row, rowMetadata) -> {
                        return      row.get(0)      +      "      "      +
rowMetadata.getColumnMetadata(0).getJavaType();
                    }))),
            Connection::close).doOnNext(System.out::println).doOnError(Throwable::printStackTrace)
                .blockLast(Duration.ofSeconds(60));
    }

    public static void procedure()
    {
        ConnectionFactory connectionFactory = getFactoryByProperties();
        Connection          connection      =
Mono.from(connectionFactory.create()).doOnError(Throwable::printStackTrace)
            .block(Duration.ofSeconds(60L));
        Statement stmt1 = connection
            .createStatement("create or replace procedure pro_testCall(bb in
out blob)" + " as" + " begin"
                + " insert into SYSDBA.testBlob values(bb);"
                + " select c into bb from SYSDBA.testBlob limit 1,1;" + "
end;");
        Statement      stmt2      =      connection.createStatement("{call
pro_testCall(:bb)}");

        ByteBuffer by = ByteBuffer.wrap(new byte[] {1, 2, 3, 4, 5});
        Flux.from(stmt1.execute()).thenMany(Flux
            .from(stmt2.bind("bb", Parameters.inOut(R2dbcType.BLOB,

```

```

Blob.from(Mono.just(by))).execute()
    .flatMap(result -> result.map(((row, rowMetadata) -> {
        byte[] bytes = R2dbcUtils.blobToByteArray(row.get(0,
        Blob.class));
        for (byte b : bytes)
        {
            System.out.println(b);
        }
        return "";
    }))).doOnNext(System.out::println).thenMany(Flux.from(connection.close()))
    .blockLast(Duration.ofSeconds(60L));
}

public static void savePoint(String sql) throws SQLException
{
    if (StrUtil.isBlank(sql))
    {
        throw new NullPointerException("sql cannot be blank!");
    }
    ConnectionFactory connectionFactory = getFactoryByDataSource();
    Connection connection = Mono.from(connectionFactory.create())
        .block(Duration.ofSeconds(60L));

    Mono.from(connection.setAutoCommit(false)).block(Duration.ofSeconds(60L));
    Statement stmt1 = connection
        .createStatement("UPDATE SYSDBA.TEST_CALL200 set c1= 'fw1-1' where
c1 = 'fwu-1'");
    Statement stmt2 = connection
        .createStatement("UPDATE SYSDBA.TEST_CALL200 set c2= '2022-9-20'
where c1 = 'fw1-1'");

    Mono.from(stmt1.execute()).then(Mono.from(connection.createSavepoint("sp1"))
    )
        .then(Mono.from(stmt2.execute()))
        .then(Mono.from(connection.rollbackTransactionToSavepoint("sp1")))
    )
        .then(Mono.from(connection.commitTransaction())).then(Mono.from(
        connection.close()))
        .block(Duration.ofSeconds(60L));
}

public static void rollback() throws SQLException
{
    ConnectionFactory connectionFactory = getFactoryByDataSource();
    Connection connection =

```

```

Mono.from(connectionFactory.create()).block(Duration.ofSeconds(60L));

Mono.from(connection.setAutoCommit(false)).block(Duration.ofSeconds(60L));
    Statement statement = connection
        .createStatement("UPDATE SYSDBA.TEST_CALL200 set c1= 'fw1-3' where
c1 = 'fwu-1'");
    Flux.from(statement.execute()).flatMap(result
        ->
Flux.from(result.getRowsUpdated()))
        .map(Math::toIntExact).collectList().block(Duration.ofSeconds(6
0L));
    Publisher<Void> rollbackPb = connection.rollbackTransaction();
    Mono.from(rollbackPb).block(Duration.ofSeconds(60L));
    Mono.from(connection.close()).block(Duration.ofSeconds(60L));
}

public static void batch()
{
    ConnectionFactory connectionFactory = getFactoryByUrl();
    Flux.usingWhen(connectionFactory.create(),
        connection -> Flux
            .from(connection.createBatch()).add("SELECT      *      FROM
ATS.BUG_INFO WHERE ID = 520000")
            .add("SELECT * FROM ATS.BUG_INFO WHERE ID = 520001")
            .add("SELECT * FROM SYSDBA.TEST_CALL200 LIMIT 1")
            .add("UPDATE SYSDBA.TEST_CALL200 SET C10 = 'LOLO'
WHERE C1 = 'fwu-7'")
            .add("SELECT C10 FROM SYSDBA.TEST_CALL200 WHERE C1 =
'fwu-7').execute()
            .flatMap(result -> result.map(((row, rowMetadata) -> {
                return row.get(0);
            })),

Connection::close).doOnNext(System.out::println).doOnError(Throwable::prints
stackTrace)
        .blockLast(Duration.ofSeconds(60));
}

public static void query(String sql)
{
    if (StrUtil.isBlank(sql))
    {
        throw new NullPointerException("sql cannot be blank!");
    }
    ConnectionFactory connectionFactory = getFactoryByUrl();
    Flux.usingWhen(connectionFactory.create(),           connection      ->
Flux.from(connection.createStatement(sql)

```

```
.execute()).flatMap(result -> result.map(((row, rowMetadata) -> {
    return row.get(0);
})),  
Connection::close).doOnNext(System.out::println).doOnError(Throwable::printStackTrace)  
.blockLast(Duration.ofSeconds(60));  
}  
public static ConnectionFactory getFactoryByProperties()  
{  
    Properties properties = new Properties();  
    ConnectionFactoryOptions options = ConnectionFactoryOptions.builder()  
.option(DmConnectionFactoryProvider.PROPERTIES,  
properties).option(HOST, "127.0.0.1")  

```

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, -14999) |

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

代码	解释
-70000	一般错误
-70001	无效的数据缓冲区
-70002	无效的数据长度
-70003	无效的数据库数据类型
-70004	无效的 C 数据类型
-70005	字符串截断
-70006	无效的精度
-70007	无效的刻度
-70008	数据类型转换失败
-70009	不完整的 UTF 字符串

-70010	无效的 INTERVAL 类型
-70011	无效的转换字符串
-70012	数据超出范围
-70013	转换失败
-70014	非法的字符
-70015	无效的日期字符串
-70016	日期超出范围
-70017	内存分配出错
-70018	缓冲区不足
-70019	网络通讯失败
-70020	CRC 校验失败
-70021	无效的操作
-70022	未知的事务状态
-70023	无效的参数值
-70024	连接已打开
-70025	连接未打开
-70026	非法的函数调用序列
-70027	无效的游标状态
-70028	创建 SOCKET 连接失败
-70029	创建加密数据失败
-70030	不支持的服务器版本
-70031	加密组件初始化失败
-70032	无效的指示符指针
-70033	无效的列索引
-70034	已经发送数据
-70035	无效的空指针
-70036	无效的句柄状态
-70037	字符串不完整
-70038	无效的参数绑定类型
-70039	绑定的参数个数不正确
-70040	绑定的参数类型不正确
-70041	无效的属性
-70042	只读属性
-70043	无效的描述符类型
-70044	指定的操作不支持
-70045	无效的描述索引
-70046	无效的游标位置
-70047	无效的字符串或缓冲区长度

-70048	指定描述句柄不可修改
-70049	无效的精度和刻度
-70050	无效的描述符属性
-70051	语句不返回结果集
-70052	无效的属性值
-70053	无效的属性值
-70054	特性未实现
-70055	无效的游标类型
-70056	无效的描述信息
-70057	关联的句柄未准备
-70058	只读结果集
-70059	无效的书签值
-70060	行数据过长
-70061	数据不在缓冲区中
-70062	无效的模式名
-70063	无效的表名
-70064	无效的配置值
-70065	连接异常, 切换当前连接成功
-70066	连接异常, 切换当前连接失败
-70067	压缩库未装载
-70068	消息压缩错误
-70069	消息解压错误
-70070	初始化 SSL 环境失败
-70071	无效的属性标识
-70072	游标名称已经存在
-70073	Kerberos 认证失败
-70074	对象类型未知
-70075	无效的对象描述符
-70076	描述符正被使用
-70077	对象句柄已绑定
-70078	参数默认值使用无效
-70079	无效的并发属性值
-70080	指定行出现错误
-70081	指定行超过范围
100	空结果集
101	字符串截断
102	在集函数中计算 NULL 值
103	无效的表名

104	删除 0 行记录
105	插入行记录
106	更新行记录
107	跨语句游标操作
108	收回权限时无相应权限
109	试图转换空字符串
110	编译没有结束
111	结果集数据获取完成
112	不完整的 UTF8 字符
113	结果集缓存满
114	刻度截断
115	不完整的字符
117	范围分区未包含 MAXVALUE, 可能无法定位到分区
119	列表分区未包含 DEFAULT, 可能无法定位到分区
70000	操作成功
70001	空串转换
70002	缓冲区不足
70003	字符串缓冲区不足
70004	字符串截断
70005	转换刻度丢失
70006	连接属性已改变
70007	选项值已改变
70008	服务器处于主库挂起状态
70009	服务器处于主库配置状态
70010	服务器处于备库配置状态
70011	服务器处于备库打开状态
70012	不完整的字符
70013	游标操作冲突

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。

武汉达梦数据库股份有限公司

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 A 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 201 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

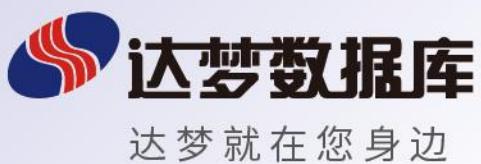
地址: 合肥市包河区马鞍山路金帝国际城 7 栋 3 单元 706 室
邮编: 230022
电话: (+86) 0551-3711086

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



达梦就在您身边

武汉达梦数据库股份有限公司

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

