



GBase 8s JDBC Driver 程序员指南



GBase 8s JDBC Driver 程序员指南，南大通用数据技术股份有限公司

GBase 版权所有©2004-2021，保留所有权利

版权声明

本文档所涉及的软件著作权及其他知识产权已依法进行了相关注册、登记，由南大通用数据技术股份有限公司合法拥有，受《中华人民共和国著作权法》、《计算机软件保护条例》、《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的南大通用数据技术股份有限公司的版权信息由南大通用数据技术股份有限公司合法拥有，受法律的保护，南大通用数据技术股份有限公司对本文档可能涉及到的非南大通用数据技术股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅，并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经南大通用数据技术股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将视为侵权，南大通用数据技术股份有限公司具有依法追究其责任的权利。

本文档中包含的信息如有更新，恕不另行通知。您对本文档的任何问题，可直接向南大通用数据技术股份有限公司告知或查询。

通讯方式

南大通用数据技术股份有限公司

天津市高新区开华道22号普天创新产业园东塔20-23层

电话：400-013-9696

邮箱：info@gbase.cn

商标声明

GBASE[®]是南大通用数据技术股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由南大通用数据技术股份有限公司合法拥有，受法律保护。未经南大通用数据技术股份有限公司书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯南大通用数据技术股份有限公司商标权的，南大通用数据技术股份有限公司将依法追究其法律责任。

目 录

1 简介	1
1.1 内容.....	1
1.2 符合行业标准.....	1
1.3 演示数据库.....	1
1.4 如何阅读语法图.....	2
1.5 示例代码约定.....	3
2 入门	4
2.1 什么是 JDBC?	4
2.2 什么是 JDBC 驱动程序?	4
2.3 GBase 8s JDBC 驱动程序概述.....	5
2.3.1 GBase 8s JDBC 驱动程序中的文件.....	5
2.3.2 客户机和服务器 JDBC 驱动程序	6
2.4 取得 JDBC 驱动程序.....	6
2.5 安装 JDBC 驱动程序.....	6
2.6 在应用程序中使用驱动程序.....	6
2.7 在 applet 中使用驱动程序	7
3 连接至数据库.....	8
3.1 OnConfig.properties 参数配置文件.....	8
3.1.1 STD_INTERVAL_TO_FLOAT 配置参数	8
3.2 加载 GBase 8s JDBC 驱动程序.....	9
3.3 DataSource 对象.....	10
3.4 DriverManager.getConnection() 方法	13
3.4.1 数据库 URL 的格式.....	14
3.4.2 数据库与数据库服务器连接.....	18
3.4.3 指定属性.....	20
3.5 随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量.....	21
3.6 动态地读取 GBase 8s sqlhosts 文件.....	31
3.6.1 连接属性语法.....	32
3.6.2 管理要求.....	33
3.6.3 以 sqlhosts 数据来更新 LDAP 服务器的实用程序.....	33
3.7 连接至高可用性集群的服务器.....	34
3.7.1 通过 GBase 8s Connection Manager 连接至高可用性集群服务器的属性 ..35	

3.7.2 通过 SQLHOST 文件组条目，来连接至高可用性集群服务器的属性	38
3.7.3 直接连接至服务器的 HAC 对的属性	39
3.7.4 检查高可用性辅助服务器的只读状态	39
3.7.5 对 HAC 辅助服务器的连接重试尝试	40
3.7.6 JDBC读写分离	42
3.8 其他多层解决方案	45
3.9 加密选项	46
3.9.1 JCE 安全软件包	46
3.9.2 符合 GBase FIPS 的安全软件包	46
3.9.3 口令加密	47
3.9.4 配置连接来使用 SSL 协议	47
3.9.5 网络加密	48
3.10 关闭连接	50
4 执行数据库操作	51
4.1 查询数据库	51
4.1.1 将查询发送至 GBase 8s 数据库的示例	51
4.1.2 结果集	52
4.1.3 释放资源	53
4.1.4 跨线程执行	53
4.1.5 滚动游标	54
4.1.6 保持游标	55
4.2 更新数据库	55
4.2.1 执行批量更新	55
4.2.2 执行批量插入	56
4.3 参数、转义语法和不受支持的方法	56
4.3.1 CallableStatement OUT 参数	57
4.3.2 CallableStatement 中命名了的参数	63
4.3.3 JDBC 支持 DESCRIBE INPUT	69
4.3.4 转义语法	70
4.3.5 不支持的方法和行为不同的方法	71
4.4 处理事务	73
4.5 处理错误	75
4.5.1 以 SQLException 类来处理错误	75
4.5.2 检索语法错误偏移量	76

4.5.3 以 com.gbasedbt.jdbc.Message 类来处理错误.....	77
4.6 访问数据库元数据.....	77
4.7 对 JDBC API 的其他 GBase 8s 扩展	78
4.7.1 “自动释放”特性.....	79
4.7.2 获取驱动程序版本信息.....	79
4.7.3 JDBC驱动新增getObject方法.....	80
4.8 存储和检索 XML 文档.....	80
4.8.1 建立环境，来使用 XML 方法	81
4.8.2 插入数据.....	82
4.8.3 检索数据.....	83
4.8.4 插入数据示例.....	84
4.8.5 检索数据示例.....	85
5 操作 GBase 8s 类型	87
5.1 distinct 数据类型.....	87
5.1.1 插入数据示例.....	88
5.1.2 检索数据示例.....	89
5.1.3 不支持的方法.....	90
5.2 BYTE 和 TEXT 数据类型	91
5.2.1 高速缓存大对象.....	91
5.2.2 示例：插入或更新数据.....	91
5.2.3 示例：选择数据.....	94
5.3 SERIAL 和 SERIAL8 数据类型.....	95
5.4 BIGINT 和 BIGSERIAL 数据类型	96
5.5 INTERVAL 数据类型.....	97
5.5.1 Interval 类	97
5.5.2 IntervalYM 类.....	99
5.5.3 IntervalDF 类	101
5.5.4 Interval 示例	103
5.6 集合和数组.....	104
5.6.1 集合示例.....	104
5.6.2 数组示例.....	107
5.7 命名的和未命名的行.....	108
5.7.1 间隔和集合支持.....	109
5.7.2 不支持的方法.....	109

5.7.3 SQLData 接口.....	110
5.7.4 Struct 接口	114
5.7.5 ClassGenerator 实用程序	119
5.8 类型高速缓存信息.....	122
5.9 智能大对象数据类型.....	122
5.9.1 数据库服务器中的智能大对象.....	123
5.9.2 客户机应用程序中的智能大对象.....	124
5.9.3 在智能大对象上执行操作.....	130
5.9.4 使用存储特征.....	136
5.9.5 使用状态特征.....	145
5.9.6 使用锁.....	146
5.9.7 高速缓存大对象.....	148
5.9.8 避免转移大对象的错误.....	148
5.9.9 智能大对象示例.....	149
6 与不透明数据类型一起使用.....	152
6.1 IfmxUDTSQLInput 接口.....	153
6.1.1 读取数据.....	153
6.1.2 于数据流中定位.....	153
6.1.3 设置或获取数据属性.....	154
6.2 IfmxUDTSQLOutput 接口	154
6.3 映射不透明数据类型.....	154
6.4 键入高速缓存信息.....	155
6.5 不支持的方法.....	155
6.6 创建不透明类型和 UDR	156
6.6.1 创建不透明类型和 UDR 概述	156
6.6.2 准备创建不透明类型和 UDR	157
6.6.3 创建不透明类型.....	158
6.6.4 创建 UDR	160
6.6.5 Java 类的要求.....	162
6.6.6 SQL 名称	162
6.6.7 指定不透明类型的属性.....	163
6.6.8 创建 JAR 和类文件.....	166
6.6.9 将类定义发送到数据库服务器.....	167
6.6.10 从现有代码创建不透明类型.....	168

6.6.11 移除不透明类型和 JAR 文件	170
6.6.12 创建 UDR	170
6.6.13 移除 UDR 和 JAR 文件	171
6.6.14 获取有关不透明类型和 UDR 的信息	172
6.6.15 在事务中执行	174
6.7 示例	174
6.7.1 类定义	174
6.7.2 插入数据	177
6.7.3 检索数据	178
6.7.4 不透明类型中的智能大对象	179
6.7.5 使用 UDTManager 从现有的 Java 类创建不透明类型	181
6.7.6 不需现有的 Java 类创建不透明类型	194
6.7.7 使用 UDRManager 创建 UDR	200
7 全球化和日期格式	202
7.1 支持 JDK 和全球化	202
7.2 支持 GBase 8s GLS 变量	202
7.3 支持 DATE 最终用户格式	204
7.3.1 GL_DATE 变量	204
7.3.2 DBDATE 变量 (deprecated)	206
7.3.3 DBCENTURY 变量	208
7.4 最终用户格式的优先规则	210
7.5 支持代码集转换	210
7.5.1 数据库代码集的字符	211
7.5.2 客户端代码集 Unicode	213
7.5.3 连接具有非 ASCII 字符的数据库	213
7.5.4 TEXT 和 CLOB 数据类型的代码集转换	214
7.5.5 BLOB 和 BYTE 数据类型的代码集转换	216
7.6 用户定义的语言环境	217
7.6.1 使用 NEWLOCALE 和 NEWCODESET 环境变量连接	217
7.6.2 使用 NEWNLSMAP 环境变量连接	218
7.7 支持全球化的错误消息	218
8 调优和故障排除	218
8.1 调试 JDBC API 程序	219
8.2 管理性能	219

8.2.1 管理访存缓冲区大小.....	219
8.2.2 大对象的内存管理.....	220
8.2.3 减少网络流量.....	221
8.2.4 批量插入.....	221
8.2.5 连接池.....	221
8.2.6 避免应用程序挂起问题（仅限于 HP-UX）.....	226
9 Oracle兼容模式.....	227
9.1 参数说明及使用方法.....	227
9.1.1 参数说明.....	227
9.1.2 使用方法.....	227
10 附录.....	228
10.1 示例代码文件.....	228
10.1.1 可用示例摘要.....	228
10.2 DataSource 扩展.....	236
10.2.1 读和写属性.....	236
10.2.2 获取和设置标准属性.....	237
10.2.3 获取和设置 GBase 8s 连接属性.....	238
10.2.4 获取和设置连接池 DataSource 属性.....	242
10.3 映射数据类型.....	243
10.3.1 在 GBase 8s 和 JDBC 数据类型之间映射的数据类型.....	243
10.3.2 PreparedStatement.setXXX() 扩展的数据类型映射.....	246
10.3.3 ResultSet.getXXX() 方法的数据类型映射.....	257
10.3.4 UDT manager 和 UDR manager 的数据类型映射.....	261
10.4 转换内部 GBase 8s 数据类型.....	265
10.4.1 IfxToJavaType 类.....	265
10.5 错误消息.....	274
10.5.1 -79700.....	275
10.5.2 -79702.....	275
10.5.3 -79703.....	275
10.5.4 -79704.....	275
10.5.5 -79705.....	275
10.5.6 -79706.....	275
10.5.7 -79707.....	275
10.5.8 -79708.....	276

10. 5. 9 -79709.....	276
10. 5. 10 -79710.....	276
10. 5. 11 -79711.....	276
10. 5. 12 -79712.....	276
10. 5. 13 -79713.....	277
10. 5. 14 -79714.....	277
10. 5. 15 -79715.....	277
10. 5. 16 -79716.....	277
10. 5. 17 -79717.....	277
10. 5. 18 -79718.....	277
10. 5. 19 -79719.....	278
10. 5. 20 -79720.....	278
10. 5. 21 -79721.....	278
10. 5. 22 -79722.....	278
10. 5. 23 -79723.....	278
10. 5. 24 -79724.....	278
10. 5. 25 -79725.....	279
10. 5. 26 -79726.....	279
10. 5. 27 -79727.....	279
10. 5. 28 -79728.....	279
10. 5. 29 -79729.....	279
10. 5. 30 -79730.....	280
10. 5. 31 -79731.....	280
10. 5. 32 -79732.....	280
10. 5. 33 -79733.....	280
10. 5. 34 -79734.....	280
10. 5. 35 -79735.....	280
10. 5. 36 -79736.....	280
10. 5. 37 -79737.....	281
10. 5. 38 -79738.....	281
10. 5. 39 -79739.....	281
10. 5. 40 -79740.....	281
10. 5. 41 -79741.....	281
10. 5. 42 -79742.....	281

10. 5. 43 -79744.....	282
10. 5. 44 -79745.....	282
10. 5. 45 -79746.....	282
10. 5. 46 -79747.....	282
10. 5. 47 -79748.....	282
10. 5. 48 -79749.....	283
10. 5. 49 -79750.....	283
10. 5. 50 -79755.....	283
10. 5. 51 -79756.....	283
10. 5. 52 -79757.....	283
10. 5. 53 -79758.....	284
10. 5. 54 -79759.....	284
10. 5. 55 -79760.....	284
10. 5. 56 -79761.....	284
10. 5. 57 -79762.....	284
10. 5. 58 -79764.....	285
10. 5. 59 -79765.....	285
10. 5. 60 -79766.....	285
10. 5. 61 -79767.....	285
10. 5. 62 -79768.....	285
10. 5. 63 -79769.....	286
10. 5. 64 -79770.....	286
10. 5. 65 -79771.....	286
10. 5. 66 -79772.....	286
10. 5. 67 -79774.....	286
10. 5. 68 -79775.....	286
10. 5. 69 -79776.....	287
10. 5. 70 -79777.....	287
10. 5. 71 -79778.....	287
10. 5. 72 -79780.....	287
10. 5. 73 -79781.....	287
10. 5. 74 -79782.....	287
10. 5. 75 -79783.....	288
10. 5. 76 -79784.....	288

10. 5. 77 -79785.....	288
10. 5. 78 -79786.....	288
10. 5. 79 -79788.....	288
10. 5. 80 -79789.....	289
10. 5. 81 -79790.....	289
10. 5. 82 -79792.....	289
10. 5. 83 -79793.....	289
10. 5. 84 -79794.....	289
10. 5. 85 -79795.....	289
10. 5. 86 -79796.....	290
10. 5. 87 -79797.....	290
10. 5. 88 -79798.....	290
10. 5. 89 -79799.....	290
10. 5. 90 -79800.....	290
10. 5. 91 -79801.....	291
10. 5. 92 -79802.....	291
10. 5. 93 -79803.....	291
10. 5. 94 -79804.....	291
10. 5. 95 -79805.....	292
10. 5. 96 -79806.....	292
10. 5. 97 -79807.....	292
10. 5. 98 -79808.....	292
10. 5. 99 -79809.....	293
10. 5. 100 -79811.....	293
10. 5. 101 -79812.....	293
10. 5. 102 -79814.....	293
10. 5. 103 -79815.....	293
10. 5. 104 -79816.....	294
10. 5. 105 -79817.....	294
10. 5. 106 -79818.....	294
10. 5. 107 -79819.....	294
10. 5. 108 -79820.....	294
10. 5. 109 -79821.....	294
10. 5. 110 -79822.....	295

10. 5. 111 -79823.....	295
10. 5. 112 -79824.....	295
10. 5. 113 -79825.....	296
10. 5. 114 -79826.....	296
10. 5. 115 -79827.....	296
10. 5. 116 -79828.....	296
10. 5. 117 -79829.....	296
10. 5. 118 -79830.....	296
10. 5. 119 -79831.....	297
10. 5. 120 -79834.....	297
10. 5. 121 -79836.....	297
10. 5. 122 -79837.....	297
10. 5. 123 -79838.....	297
10. 5. 124 -79839.....	297
10. 5. 125 -79840.....	298
10. 5. 126 -79842.....	298
10. 5. 127 -79843.....	298
10. 5. 128 -79844.....	298
10. 5. 129 -79845.....	299
10. 5. 130 -79846.....	299
10. 5. 131 -79847.....	299
10. 5. 132 -79848.....	300
10. 5. 133 -79849.....	300
10. 5. 134 -79850.....	300
10. 5. 135 -79851.....	300
10. 5. 136 -79852.....	300
10. 5. 137 -79853.....	300
10. 5. 138 -79854.....	301
10. 5. 139 -79855.....	301
10. 5. 140 -79856.....	301
10. 5. 141 -79857.....	301
10. 5. 142 -79858.....	301
10. 5. 143 -79859.....	302
10. 5. 144 -79860.....	302

10. 5. 145 -79861.....302

10. 5. 146 -79862.....302

10. 5. 147 -79863.....302

10. 5. 148 -79864.....303

10. 5. 149 -79865.....303

10. 5. 150 79868.....303

10. 5. 151 -79877.....303

10. 5. 152 -79878.....303

10. 5. 153 -79879.....303

10. 5. 154 -79880.....304

10. 5. 155 -79881.....304

1 简介

1.1 内容

本手册描述了如何安装、加载和使用 GBase 8s JDBC Driver 从 Java™ 应用程序或应用小程序连接 GBase 8s 数据库。

这些主题描述了面向任务格式的 GBase 8s JDBC 扩展；它不包括接口中的每个方法和参数。有关完整的参考（包括所有方法和参数），请参阅联机 Javadoc™，它在安装了 GBase 8s JDBC Driver 的 doc/javadoc 目录中。

还可以使用 GBase 8s JDBC Driver 编写在服务器中执行的用户定义例程。

这些主题是为使用 JDBC API 通过 GBase 8s JDBC Driver 连接到 GBase 8s 数据库的 Java 程序员编写的。要使用这些主题，您应该知道如何使用 Java 进行编程，特别是了解 JDBC API 的类和方法。

有关软件兼容性的信息，请参阅《GBase 8s JDBC Driver 发行说明》。

这些主题来自 GBase 8s JDBC Driver 程序员指南。

1.2 符合行业标准

GBase 8s 产品符合各种标准。

基于 SQL 的 GBase 8s 产品完全兼容 SQL-92 入门级（发布为 ANSI X3.135-1992），这与 ISO 9075:1992 完全相同。另外，GBase 8s 数据库服务器的许多功能都遵守 SQL-92 中级和完全级别以及 X/Open SQL 公共应用程序环境 (CAE) 标准。

1.3 演示数据库

DB-Access 实用程序随 GBase 8s 数据库服务器产品一起提供，它包括一个或多个以下演示数据库：

- **stores_demo** 数据库以一家虚构的体育用品批发商的有关信息举例说明了关系模式。GBase 8s 出版物中的许多示例均基于 **stores_demo** 数据库。
- **superstores_demo** 数据库举例说明了对象关系模式。**superstores_demo** 数据库包含扩展数据类型、类型和表继承以及用户定义的例程的示例。

有关如何创建和填充演示数据库的信息，请参阅《GBase 8s DB-Access 用户指南》。有关数据库及其内容的描述，请参阅《GBase 8s SQL 指南：参考》。

用于安装演示数据库的脚本位于 UNIX[™] 平台上的 \$GBS_HOME/bin 目录和 Windows[™] 环境中的 %GBS_HOME%\bin 目录中。

1.4 如何阅读语法图

语法图使用特殊组件描述语句和命令的语法。

从左到右，从上到下跟随线的路径阅读语法图。

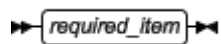
此右侧双箭头加直线符号 \Rightarrow 表示语句开始。

右侧箭头符号 \rightarrow 表示语句延续到下一行。

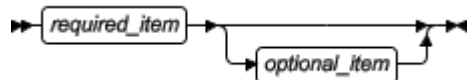
右箭头加直线符号 \rightarrow 表示语句继续上一行的内容。

直线、右箭头加左箭头符号 $\rightarrow\leftarrow$ 表示语句结束。

必需项出现在水平线（主路径）中。



可选项出现在主路径下方。

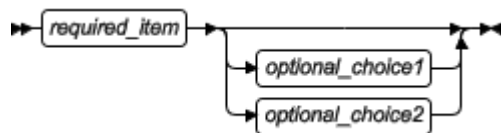


如果可以从两个或多个项中选择，那么它们以堆栈的方式表示。

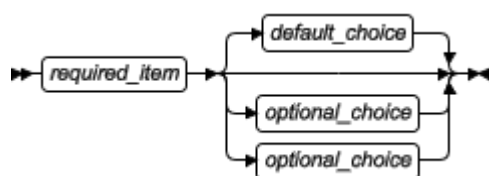
如果必须选择其中一项，那么堆栈中的一项出现在主路径上。



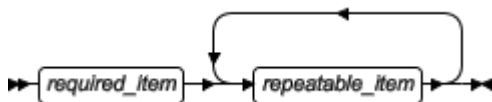
如果从中选择的项是可选的，那么整个堆栈出现在主路径下方。



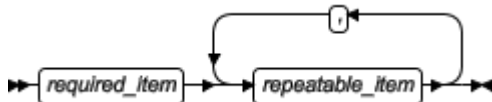
如果缺省其中一项，则它会在主路径上方显示，剩余的选项将会显示在下方。



返回左侧的箭头，在主线之上，表示该项可重复。在此情况下，重复项必须用一个或多个空格隔开。



如果重复的箭头包含一个逗号，那么您必须使用逗号分隔重复的项。

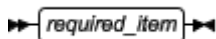


堆栈上方重复的箭头表示可以从堆栈的项目中进行多个选择或者重复一个选择。

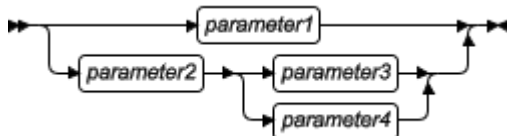
SQL 关键字以大写字母出现（例如：**FROM**）。它们必须严格按照所显示的拼写。变量以小写字母出现（例如：`column-name`）。它们表示用户在语句中提供的名称或值。

如果出现了标点符号、括号、算术运算符或其它这样的符号，那么必须将它们作为语法的一部分输入。

某些时候，一个变量表示一个语句段。例如：在以下语法图中，变量 `parameter-block` 表示已标记为 `parameter-block` 的语句段：



`parameter-block`:



1.5 示例代码约定

SQL 代码的示例在整个出版物中出现。除非另有说明，代码不特定于任何单个的 GBase 8s 应用程序开发工具。

如果示例中仅列出 SQL 语句，那么它们将不用分号定界。例如：您可能看到以下示例中的代码：

```
CONNECT TO stores_demo
...

DELETE FROM customer
  WHERE customer_num = 121
...

COMMIT WORK
```


DISCONNECT CURRENT

要将此 SQL 代码用于特定产品，必须应用该产品的语法规则。例如，如果使用的是 SQL API，那么必须在每条语句的开头使用 EXEC SQL，并在每条语句的结尾使用分号（或其他合适的定界符）。如果使用的是 DB - Access，那么必须用分号将多条语句隔开。

提示： 代码示例中的省略点表示在整个应用程序中将添加更多的代码，但是不必显示它以描述正在讨论的概念。

有关使用特定应用程序开发工具或 SQL API 的 SQL 语句的详细指导，请参阅您的产品文档。

2 入门

本主题主要提供 GBase 8s JDBC Driver 和 JDBC API 的概述。

2.1 什么是 JDBC?

Java™ 数据库连接 (JDBC) 是标准应用系统编程接口 (API) 的 JavaSoft 规范，其允许 Java 程序访问数据库管理系统。JDBC API 由一系列以 Java 编程语言编写的接口和类组成。

使用这些标准接口和类，编程人员可编写连接至数据库的应用程序，发送以结构化查询语言 (SQL) 编写的查询，并处理结果。

由于 JDBC 是标准规范，因此，使用 JDBC API 的一个 Java 程序可连接至任何数据库管理系统 (DBMS)，只要存在该特定 DBMS 的驱动程序。

2.2 什么是 JDBC 驱动程序?

JDBC API 定义 Java™ 接口和类，编程人员使用其来连接至数据库，并发送查询。JDBC 驱动程序实现特定 DBMS 供应商这些接口和类。

在使用 JDBC API 的 Java 程序实际连接至数据库之前，它为特定 DBMS 加载指定的驱动程序。然后，JDBC DriverManager 类将所有 JDBC API 调用发送至加载了的驱动程序。

有四类 JDBC 驱动程序：

JDBC-ODBC 桥加 ODBC 驱动程序，也称为 Type 1 驱动程序

将 JDBC API 调用翻译为 Microsoft™ ODBC 调用，然后将其传至 ODBC 驱动程序
必须在使用此类驱动程序的每个客户机计算机上加载 ODBC 二进制代码。

ODBC 是“开放式数据库连接”的首字母缩略词。

本地 API，一定程度上的 Java 驱动程序，也称为 Type 2 驱动程序

将 JDBC API 调用转换为特定于 DBMS 的客户机 API 调用

像桥驱动程序一样，这类驱动程序要求在每一客户机计算机上加载某些二进制代码。

JDBC-Net，纯 Java 驱动程序，也称为 Type 3 驱动程序

将 JDBC API 调用发送至中间层服务器，其将这些调用翻译为特定于 DBMS 的网络协议

然后将翻译了的调用发送至特定的 DBMS。

本地协议，纯 Java 驱动程序，也称为 Type 4 驱动程序

直接将 JDBC API 调用转换为特定于 DBMS 的网络协议，而不经中间层

此驱动程序允许客户机应用程序直接连接至数据库服务器。

2.3 GBase 8s JDBC 驱动程序概述

GBase 8s JDBC Driver 是本地协议，纯 Java 驱动程序 (Type 4)。因此，使用 GBase 8s JDBC 驱动程序时，会话直接连接至数据库或数据库服务器，而不经中间层。

Javadoc™ 页面详尽地描述 GBase 8s 扩展类、接口和方法。

在 UNIX™ 中，Javadoc 页面位于 \$JDBCLOCATION/doc/javadoc 中，在此，\$JDBCLOCATION 引用安装了 GBase 8s JDBC 驱动程序的那个目录。

在 Windows™ 中，Javadoc 页面位于 %JDBCLOCATION%\doc\javadoc 中，在此，%JDBCLOCATION% 引用安装了 GBase 8s JDBC 驱动程序的那个目录。

2.3.1 GBase 8s JDBC 驱动程序中的文件

GBase 8s JDBC 驱动程序为绿色免安装版，文件名称为 gbasedbtjdbc_xx.jar (xx 代表版本号)，驱动中包含如下功能：

- 1、JDBC API 接口、类和方法的优化的实现
 - 2、实用程序：ClassGenerator，轻量目录访问协议 (LDAP) loader，及其他
- 以 javac 命令的 -O 选项来编译该文件。

- 3、驱动程序支持的所有消息文本的本地化版本
 - 4、包括与 DataSource 相关的、与连接池相关的以及与 XA 相关的类文件的实现
- 包括 SQLJ 程序的运行时刻支持的类以 javac 命令的 -O 选项来编译该文件。

2.3.2 客户机和服务器 JDBC 驱动程序

GBase 8s JDBC Driver 存在两个版本：客户机侧驱动程序和服务端侧驱动程序。

客户机侧驱动程序 gbasedbtjdbc_xx.jar 用于客户机 Java™ 应用程序访问 GBase 8s 数据库服务器。

服务器侧驱动程序包括 jdbc.jar，为数据库服务器自带的一部分。由于 jdbc.jar 派生自 gbasedbtjdbc_xx.jar，因此两个驱动程序有许多共同特性。

本指南主要关于客户机侧驱动程序；然而，共同特性的信息同时适用于服务器侧和客户机侧版本。

重要提示： 请不要混淆或互换服务器侧与客户机侧版本。

2.4 取得 JDBC 驱动程序

从光盘 Setup 目录下取得 GBase 8s JDBC Driver 驱动程序 gbasedbtjdbc_xx.jar。

2.5 安装 JDBC 驱动程序

gbasedbtjdbc_xx.jar 安装包采用免安装的方式，用户无须解压和安装，可直接使用。

2.6 在应用程序中使用驱动程序

要在应用程序中使用 GBase 8s JDBC Driver，您必须设置 CLASSPATH 环境变量来指向驱动程序文件。

UNIX™

设置 CLASSPATH 环境变量的方式：

- 将 gbasedbtjdbc_xx.jar 的完全路径名称添加至 CLASSPATH：

```
setenv CLASSPATH /jdbcdrv/lib/gbasedbtjdbc_xx.jar:$CLASSPATH
```

Windows™

设置 CLASSPATH 环境变量的方式：

- 将 gbasedbtjdbc_xx.jar 的完整路径名称添加至 CLASSPATH:
set CLASSPATH=c:\jdbcdrv\lib\gbasedbtjdbc_xx.jar;%CLASSPATH%

2.7 在 applet 中使用驱动程序

可在 applet 中使用 GBase 8s JDBC Driver, 来从 web 浏览器连接至 GBase 8s 数据库。下列步骤展示如何指定 applet 中的 GBase 8s JDBC Driver, 以及如何确保从 web 服务器正确地下载驱动程序。

要在 applet 中使用 GBase 8s JDBC Driver, 请:

1. 在与 applet 类文件相同的目录中, 安装 gbasedbtjdbc_xx.jar。
2. 在 HTML 文件中 APPLET 标记的 ARCHIVE 属性中, 指定 gbasedbtjdbc_xx.jar, 如下列示例所示:

```
<APPLET ARCHIVE=gbasedbtjdbc_xx.jar CODE=my_applet.class  
CODEBASE=http://www.myhost.com WIDTH=460 HEIGHT=160>  
</APPLET>
```

重要: 有些浏览器不支持 APPLET 标记的 ARCHIVE 属性。如果您的浏览器就是这样, 则请在 web 服务器的 root 目录中解包并安装 gbasedbtjdbc_xx.jar 文件。如果您的浏览器也不支持 JDBC API, 则必须在 web 服务器的 root 目录中安装 java.sql 包中包括的类文件。要获取关于在 root 目录中安装文件的信息, 请参阅您的 web 服务器资料。

由于出于安全的原因, 未签名的 applet 不可访问某些系统资源, 因此, 对于未签名的 applet, 下列 GBase 8s JDBC Driver 特性不起作用:

- **sqlhosts 文件和 LDAP 服务器访问。**如果您正在直接地或通过 LDAP 服务器来引用 sqlhosts, 则数据库 URL 中的 GBase 8s 数据库服务器的主机名称和端口号属性或服务名称是可选的。

然而, 对于未签名的 applet, 始终需要 GBase 8s 数据库服务器的主机名称和端口号或服务名称, 除非您的 applet 正在使用 HTTP 代理服务器。要获取关于 HTTP 代理服务器的更多信息, 请参阅 HTTP 代理服务器。

- **LOBCACHE=0。**在数据库 URL 中将 LOBCACHE 环境变量设置为 0, 来指定始终将智能大对象存储在文件中。对于未签名的 applet, 不支持此设置。

提示: 通过使用 Microsoft™ Internet Explorer, 可以启用未签名 applet 的这些特性, 其提供配置 applet 许可的选项。

要从 applet 访问不同主机上的或防火墙之后的数据库, 您可使用中间层中的 GBase 8s HTTP proxy servlet。要获取更多信息, 请参阅 HTTP 代理服务器。

3 连接至数据库

在 Java™ 程序中通过以下两个步骤来建立至 GBase 8s 数据库服务器的连接：

1. 加载 GBase 8s JDBC Driver。
2. 以下列方式之一，来创建连接：
 - 使用 DataSource 对象。
 - 使用 DriverManager.getConnection() 方法。

使用 **DataSource** 对象比使用 `DriverManager.getConnection()` 方法更好，因为 **DataSource** 对象轻便，且使得关于隐含数据源的详细信息对应用程序是透明的。可修改目标数据源实现，或可将应用程序重定向至不同的服务器，而不影响应用程序代码。

DataSource 对象还可提供对连接池和分布式事务的支持。此外，Enterprise JavaBeans™ 和 J2EE 需要 **DataSource** 对象。

下列附加的连接选项是可用的：

- 设置环境变量
- 动态地读取 GBase 8s sqlhosts 文件
- 使用 HTTP 代理服务器
- 使用口令加密
- 使用网络加密

3.1 OnConfig.properties 参数配置文件

本版本新增参数配置文件 `OnConfig.properties`。可以通过更改 JDBC 驱动程序的 `OnConfig.properties` 文件，来定制 JDBC 接口的功能或者调整其行为。

`OnConfig.properties` 文件位于 `gbasedbtjdbc_xx.jar/com/gbasedbt/jdbc/OnConfig.properties` 目录中。修改配置参数时需要将 `OnConfig.properties` 解压出来，修改后再放入其中将其覆盖。

3.1.1 STD_INTERVAL_TO_FLOAT 配置参数

使用 `STD_INTERVAL_TO_FLOAT` 配置参数来指定 JDBC 端 DATETIME (或 TIMESTAMP) 类型值相减后得到结果值的数据类型，是以天为单位的浮点数，还是 INTERCAL 数据类型。

它可以设置为以下值：

1、Y 或 true 在 JDBC 端，DATETIME (或 TIMESTAMP) 类型值进行相减运算时，得到以天为单位的浮点数。

不设置或设置为其它值时，在 JDBC 端，DATETIME (或 TIMESTAMP) 类型值相减得到

INTERCAL 数据类型。

设置完成后，编辑 OnConfig.properties 文件之后，即可生效。

用法

当在 OnConfig.properties 文件中设置 STD_INTERVAL_TO_FLOAT = 1 后，执行以下语句，用户通过 JDBC 接口得到一个以天为单位的浮点数：

```
SELECT SYSDATE - TO_DATE('20170102 08:00:00','YYYY-MM-DD HH24:MI:SS')
FROM DUAL;
```

Result: 1.25

（通过 JDBC 接口返回，假定当前系统时间为 2017 年 1 月 3 日 14:00:00）

如果未设置 STD_INTERVAL_TO_FLOAT 或设置为其它值，则日期型数据类型相减返回一个 INTERVAL 数据类型

注意：配置文件参数 STD_INTERVAL_TO_FLOAT 与 URL 参数 isITF 具有相同的作用，但又互为约束，两者之间存在或关系：

- 1) 当其中一个或两个都为开启状态时，都可启用 DATETIME 相减得到浮点数的功能。
- 2) 当两个都为关闭状态时，禁用此功能。

3.2 加载 GBase 8s JDBC 驱动程序

要加载 GBase 8s JDBC Driver，请使用 Class.forName() 方法，将值 com.gbasedbt.jdbc.Driver 传给它：

```
try
{
    Class.forName("com.gbasedbt.jdbc.Driver");
}
catch (Exception e)
{
    System.out.println("ERROR: failed to load GBase 8s JDBC driver.");
    e.printStackTrace();
    return;
}
```

Class.forName() 方法加载 **Driver** 类 **Driver** 的 GBase 8s 实现。然后，**Driver** 创建驱动程序的实例，并以 **DriverManager** 类来注册它。

在已加载了 GBase 8s JDBC Driver 之后，准备好连接至 GBase 8s 数据库或数据库服务器。

如果您正在编写以 Microsoft[™] Internet Explorer 来查看的 applet，则可能需要显式地注册 GBase 8s JDBC Driver，以避免平台不兼容。

要显式地注册驱动程序，请使用 DriverManager.registerDriver() 方法：

```
DriverManager.registerDriver(com.gbasedbt.jdbc.Driver)
Class.forName("com.gbasedbt.jdbc.Driver").newInstance();
```

此方法可能注册 GBase 8s JDBC Driver 两次，这不会导致问题。

3.3 DataSource 对象

GBase 8s JDBC Driver 扩展标准 DataSource 接口，以允许在 DataSource 对象中定义连接属性（标准属性和 GBase 8s 环境变量），而不是通过 URL。

下表描述 GBase 8s 连接属性如何对应于 DataSource 属性。

GBase 8s 连接属性	DataSource 属性	数据类型	是否必需?	描述
IFXHOST	无；要了解如何设置请参阅 DataSource 扩展。	String	对于客户机侧 JDBC，必需，除非定义 SQLH_TYPE；对于服务器侧 JDBC，不是必需的	运行 GBase 8s 数据库服务器的计算机的 IP 地址或主机名称
PORTNO	portNumber	int	对于客户机侧 JDBC，必需，除非定义 SQLH_TYPE；对于服务器侧 JDBC，不是必需的	GBase 8s 数据库服务器的端口号。该端口号罗列在 /etc/services 文件中。
DATABASE	databaseName	String	不是必需的，除非连接来自运行在数据库服务器中的 web 应用程序（诸如浏览	您想要连接至其的 GBase 8s 数据库的名称 如果未指定数据库的名称，则制作至GBase 8s 数据库服务器的连接。

GBase 8s 连接属性	DataSource 属性	数据类型	是否必需?	描述
			器)	
GBASEDBTSERVER	serverName	String	对于客户机侧 JDBC，必需；对于服务器侧 JDBC，忽略	您想要连接至其的 GBase 8s 数据库服务器的名称
USER	user	String	必需	当连接至 GBase 8s 数据库或数据库服务器时，用户名称控制（或决定）会话权限 通常，必须同时指定用户名称和口令；然而，如果 DBMS 信任运行 JDBC 应用程序的用户，则可能都省略。
PASSWORD	password	String	必需	用户的口令 通常，必须同时指定用户名称和口令；然而，如果 DBMS 信任运行 JDBC 应用程序的用户，则可能都省略。
无	description	String	必需	DataSource 对象的描述
无	dataSourceName	String	不是必需的	对于连接池或分布式事务，隐含的 ConnectionPoolDataSource 或XADataSource 对象的名称

不受支持的连接属性

GBase 8s JDBC Driver 不支持 networkProtocol 和 roleName 属性。

指定连接信息

如果 LDAP（轻型目录访问协议）服务器或 sqlhosts 文件通过 SQLH_TYPE 属性提供 GBase 8s 数据库服务器的 IP 地址、主机名称或端口号或服务名称，则不必使用标准 DataSource 属性来指定它们。

ConnectionPoolDataSource 对象

要获取关于 ConnectionPoolDataSource 对象的信息，请参阅 连接池。

环境变量

要获取受支持的环境变量（属性）的列表，请参阅 随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量。要获取 GBase 8s DataSource 扩展的列表，其允许您定义环境变量值和连接池调整参数，请参阅 DataSource 扩展。驱动程序不查询用户环境来确定环境变量值。

高可用性数据复制

可随同“高可用性数据复制”来使用 DataSource 对象。要获取更多信息，请参阅 连接至高可用性集群的服务器。

示例：在示例程序中使用 DataSource 对象

下列来自 pickaseat 示例程序的代码定义并使用 DataSource 对象：

```
lfxConnectionPoolDataSource cpds = null;

try
{
    Context initCtx = new InitialContext();
    cpds = new lfxConnectionPoolDataSource();
    cpds.setDescription("Pick-A-Seat Connection pool");
    cpds.setlfxIFXHOST("158.58.60.88");
    cpds.setPortNumber(179);
    cpds.setUser("demo");
    cpds.setPassword("demo");
    cpds.setServerName("ipickdemo_tcp");
    cpds.setDatabaseName("ipickaseat");
    cpds.setlfxGL_DATE("%B %d, %Y");
    initCtx.bind("jdbc/pooling/PickASeat", cpds);
}
catch (Exception e)
```

```
{  
    System.out.println("Problem with registering the CPDS");  
    System.out.println("Error: " + e.toString());  
}
```

示例：使用带有 IFX_LOCK_MODE_WAIT 连接属性的 DataSource 对象

下列是使用 DataSource 对象的 IFX_LOCK_MODE_WAIT 连接属性的示例：

示例 1

```
lfxDataSource ds = new lfxDataSource ();  
ds.setlfxIFX_LOCK_MODE_WAIT (65);    // 等待 65 秒  
...  
int waitMode = ds.getlfxIFX_LOCK_MODE_WAIT ();
```

示例 2

```
An example Using DataSource:  
lfxDataSource ds = new lfxDataSource ();  
ds.setlfxIFX_ISOLATION_LEVEL ("0U");    // set isolation to dirty read with  
retain  
// update locks.  
...  
String isoLevel = ds.getlfxIFX_ISOLATION_LEVEL ();
```

3. 4 DriverManager.getConnection() 方法

要创建至 GBase 8s 数据库或数据库服务器的连接，可使用 DriverManager.getConnection() 方法。此方法用于创建 SQL 语句的 Connection 对象，将它们发送至 GBase 8s 数据库，并处理结果。

DriverManager 类跟踪可用的驱动程序，并处理驱动程序与数据库或数据库服务器之间的请求。getConnection() 方法的 url 参数是一个数据库 URL，其指定子协议（数据库连接性机制）、数据库或数据库服务器标识符，以及属性列表。

getConnection() 方法的第二个参数 property 是属性列表。要了解如何指定属性列表的示例，请参阅 指定属性。

下列示例展示从客户机应用程序连接至名为 **testDB** 的数据库 URL：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533/testDB:  
GBASEDBTSERVER=myserver;user=rdtest;password=test
```

在下一部分中，描述数据库 URL 语法的详细信息。

下列 CreateDB.java 程序的部分示例，展示如何通过使用 DriverManager.getConnection() 来连接至数据库 testDB。在完整的示例中，当在命令行运行程序时，将前面示例中描述的 url 变量作为参数传递。

```
try
{
    conn = DriverManager.getConnection(url);
}
catch (SQLException e)
{
    System.out.println("ERROR: failed to connect!");
    System.out.println("ERROR: " + e.getMessage());
    e.printStackTrace();
    return;
}
```

重要提示： GBase 8s JDBC Driver 唯一支持的 GBase 8s 连接类型是 tcp。不支持共享内存和其他连接类型。要获取关于连接类型的更多信息，请参阅数据库服务器的《GBase 8s 管理员指南》。

重要提示： GBase 8s JDBC Driver 并不支持 Connection 接口的所有方法。要了解不受支持的方法列表，请参阅 不支持的方法和行为不同的方法。

数据库服务器会自动关闭连接，客户机应用程序不需要显式地关闭连接，如果应用程序正在使用服务器侧 JDBC 的数据库服务器中运行，则应该显式地关闭连接。

3.4.1 数据库 URL 的格式

对于来自客户机的连接，请使用下列格式来指定数据库 URL：

```
jdbc:gbasedbt-sqli://[{ip-address|host-name}:[port-number|server-name]][:dbname]:
    GBASEDBTSERVER=servername[{:user=user;password=password}
    |CSM=(SSO=database_server@realm,ENC=true)]
    [;name=value[;name=value]...]
```

对于数据库服务器上的连接，请使用下列格式：

```
jdbc:gbasedbt-direct://[:dbname];[user=user;password=password] ]
[;name=value[;name=value]...]
```

在前面的语法中：

- 与竖线 (|) 在一起的大括号 ({ }) 表示变量的多个选择。
- 斜体字表示变量值。
- 方括号 ([]) 表示可选的值。

- 括在方括号中的词或符号是必需的（例如，GBASEDBTSERVER=）。

在数据库 URL 中不允许空格。

例如，在客户机上，您可能使用：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533/testDB:
      GBASEDBTSERVER=myserver;user=rdtest;password=test
```

在服务器上，您可能使用：

```
jdbc:gbasedbt-direct://testDB;user=rdtest;password=test
```

重要：使用服务器侧 JDBC 的连接有不同的语法。要了解详细信息，请参阅数据库服务器版本的发布说明。

下表描述数据库 URL 的变量部分，以及对等的 GBase 8s 连接属性。

GBase 8s 连接属性	数据库 URL 变量	是否必需？	描述
IFXHOST	ip-address host-name	对于客户机侧 JDBC，必需，除非定义 SQLH_TYPE 属性或使用 IFXHOST 属性；对于服务器侧 JDBC，不是必需的	正在运行 GBase 8s 数据库服务器的计算机的 IP 地址或主机名称。
PORTNO	port-number	对于客户机侧 JDBC，必需，必须指定 port-number 或 server-name，除非定义 SQLH_TYPE 属性，或使用 PORTNO 属性；对于服务器侧 JDBC，不是必需的	GBase 8s 数据库服务器的端口号 在 /etc/services 文件中罗列该端口号。
无	server-name	对于客户机侧 JDBC，必需，必须指定 port-number 或 server-name，除非定义 SQLH_TYPE 属性，或使用 PORTNO 属性；对于服务器侧 JDBC，不是必需的	罗列在 /etc/services 文件中的 GBase 8s 数据库服务器的 server-name。
DATABASE	dbname	不是必需的，除非是来自运	您想要连接至其

GBase 8s 连接属性	数据库 URL 变量	是否必需?	描述
		行在数据库服务器中的 web 应用程序的连接	的 GBase 8s 数据库的名称 如果未指定数据库的名称，则制作至 GBase 8s 数据库服务器的连接。
GBASEBTSEVER	server-name	必需	您想要连接至其的 GBase 8s 数据库服务器的名称
USER	user	必需。必须指定用户和口令，或 SSO 的 SCM 设置。 必须指定用户和口令。	想要连接至 GBase 8s 数据库或数据库服务器的用户名称。 必须同时指定用户和口令，或都不指定。如果都不指定，则驱动程序调用 System.getProperty() 来获取当前正在运行应用程序的用户的名称，以及假定信任的客户机。
PASSWORD	password	必需。必须指定用户和口令，或 SSO 的 CSM 设置。 必须指定用户和口令。	用户的口令 必须同时指定用户和口令，或都不指定。如果都不指定，则驱动程序调用 System.getProper

GBase 8s 连接属性	数据库 URL 变量	是否必需?	描述
			ty() 来取得当前正在运行应用程序的用户的名称，以及假定信任的客户机。
无	database_server@realm	必需，必须指定用户和口令，或 SSO 的 CSM 设置。	(SSO) 访问控制的服务原则。要获取信息，请参阅 随同 GBase 8s JDBC 驱动程序来使用单点登录访问控制 。
无	name=value	不是必需的	为包含在 name 变量中的 GBase 8s 环境变量指定值的名-值对，GBase 8s JDBC Driver 或 GBase 8s 数据库服务器识别它 name 变量不区分大小写。 要获取更多信息，请参阅 指定属性 和 随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量 。

如果 LDAP 服务器或 sqlhosts 文件通过 SQLH_TYPE 属性来提供 IP 地址、主机名称或端口号，则不必在数据库 URL 中指定它们。要获取更多信息，请参阅 [动态地读取 GBase 8s sqlhosts 文件](#)。

连接 URL 中的 IP 地址

GBase 8s JDBC Driver Version 3.0 及后来的版本支持 JDK 1.4 是 IPv6 感知的。也就是说，解析连接 URL 的代码可处理更长的（128 位模式）IPv6 地址（以及 IPv4 格式）。此 IP 地址可为 IPv6 文字，例如：

```
3ffe:ffff:ffff:ffff:0:0:0:12
```

要以 GBase 8s 来连接至 IPv6 端口，请使用系统属性，例如：

```
java -Djava.net.preferIPv6Addresses=true ...
```

以 GBase 8s JDBC Driver Version 3.0 或后来版本处理不带有 IPv6 文字的 URL 是不变的，且遗留行为是不变的。

冒号 (:) 是连接 URL 中的关键定界符，特别是在 IPv6 文字地址中。

必须为驱动程序创建格式良好的 URL，来识别 IPv6 文字地址。请注意，在下列示例中：

- jdbc:gbasedbt-sqli:// 是需要的。
- 环绕 8088 的冒号 (:8088:) 是需要的。
- 驱动程序不验证 3ffe:ffff:ffff:ffff:0::12。
- 8088 必须是 < 32k 的有效数值。

```
jdbc:gbasedbt-sqli://3ffe:ffff:ffff:ffff:0::12:8088:gbasedbtserver=X...
```

3.4.2 数据库与数据库服务器连接

使用 `DriverManager.getConnection()` 方法，可创建至 GBase 8s 数据库或 GBase 8s 数据库服务器的连接。

要创建至 GBase 8s 数据库的连接，请指定 URL 中 `dbname` 变量的数据库名称。如果省略数据库的名称，则表示至数据库服务器的连接，由数据库 URL 的 **GBASEDBTSERVER** 环境变量或连接属性列表来指定该数据库服务器。

如果直接连接至 GBase 8s 数据库服务器，则可在 Java™ 程序中执行连接至数据库的 SQL 语句。

同时连接至数据库和数据库服务器的所有连接都必须通过 **GBASEDBTSERVER** 环境变量来包括 GBase 8s 数据库服务器的名称。

`DriverManager.getConnection()` 方法中给出的示例展示如何以数据库 URL 来创建直接连接至名为 **testDB** 的 GBase 8s 数据库。

下列来自 `DBConnection.java` 程序的示例展示如何首先创建至名为 **myserver** 的 GBase 8s 数据库服务器的连接，然后，通过使用 `Statement.executeUpdate()` 方法来连接至数据库 **testDB**。

当在命令行运行程序时，将下列数据库 URL 作为参数传给程序；请注意，URL 不包括数据库的名称：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533:GBASEDBTSERVER=myserver;
```

```
user=rdtest;password=test
```

代码为:

```
String cmd = null;
int rc;
Connection conn = null;

try
{
    Class.forName("com.gbasedbt.jdbc.Driver");
}
catch (Exception e)
{
    System.out.println("ERROR: failed to load GBase 8s JDBC driver.");
}

try
{
    conn = DriverManager.getConnection(newUrl);
}
catch (SQLException e)
{
    System.out.println("ERROR: failed to connect!");
    e.printStackTrace();
    return;
}

try
{
    Statement stmt = conn.createStatement();
    cmd = "database testDB;";
    rc = stmt.executeUpdate(cmd);
    stmt.close();
}
catch (SQLException e)
{
    System.out.println("ERROR: execution failed - statement:
" + cmd);
    System.out.println("ERROR: " + e.getMessage());
}
```


3.4.3 指定属性

当使用 `DriverManager.getConnection()` 方法来创建连接时，GBase 8s JDBC Driver 仅从连接数据库 URL 中的名-值对，或从连接属性列表读取 GBase 8s 环境变量。驱动程序不为任何环境变量查询用户环境。

要在连接数据库 URL 的名-值对中指定 GBase 8s 环境变量，请参阅 数据库 URL 的格式。

要通过属性列表来指定 GBase 8s 环境变量，请使用 `java.util.Properties` 类来构建属性的列表。该属性的列表可能包括 GBase 8s 环境变量，诸如 **GBASEDBTSERVER**，以及 **user** 和 **password**。

在构建了属性列表之后，将它作为第二个参数传给 `DriverManager.getConnection()` 方法。您仍需包括数据库 URL 作为第一个参数，尽管在此情况下您不需要在 URL 中包括属性的列表。

下列来自 `optofc.java` 示例的代码展示如何使用 `java.util.Properties` 类来设置连接属性。它首先使用 `Properties.put()` 方法来在连接属性列表中将环境变量 **OPTOFC** 设置为 1；然后，它连接至该数据库。

在此示例中的 `DriverManager.getConnection()` 方法需要两个参数：数据库 URL 和属性列表。该示例创建类似于 `DriverManager.getConnection()` 方法中给出的示例的连接。

当在命令行运行程序时，将下列数据库 URL 作为参数传给示例程序：

```
jdbc:gbasedbt-sqli://myhost:1533:gbasedbtserver=myserver;  
user=rdtest;password=test
```

代码为：

```
try  
{  
    Class.forName("com.gbasedbt.jdbc.Driver");  
}  
catch (Exception e)  
{  
    System.out.println("ERROR: failed to load GBase 8s JDBC driver.");  
}  
  
try  
{  
    Properties pr = new Properties();  
    pr.put("OPTOFC", "1");  
    conn = DriverManager.getConnection(newUrl, pr);  
}
```

```
catch (SQLException e)
{
    System.out.println("ERROR: failed to connect!");
}
```

3.5 随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量

下表罗列客户机 JDBC 驱动程序支持的大多数 GBase 8s 环境变量。对于服务器侧 JDBC，请使用数据库 URL 中的参数设置，而不设置环境变量，因为环境变量会应用于数据库服务器中运行的所有程序。要获取更多关于属性的信息，请参阅 指定属性。

要了解提供全球化特性的环境变量列表，请参阅 全球化和日期格式。要了解用于故障排除的环境变量列表，请参阅 调优和故障排除。

受支持的 GBase 8s 环境变量	描述
APPENDISAM	当设置为 TRUE 时，APPENDISAM 环境变量将 ISAM 错误代码和消息（如果出现的话）附加至 SQL 异常消息，当调用 SQL 异常的 .toString() 或 .getMessage() 时展示。以下列格式展示该异常消息： <div><GBase 8s ERROR MESSAGE> (<GBase 8s CODE>) ISAM error: <ISAM MESSAGE>(<ISAM CODE>)</div>
CSM	指定要使用的“通讯支持模块”。 GBase 8s JDBC Driver 3.0 和后来的版本支持加密 CSM。要获取更多信息，请参阅 加密选项。
DBANSIWARN	当设置为 1 时，请检查对 ANSI 标准语法的 GBase 8s 扩展
DBSPACETEMP	指定在其中构建临时表的 dbspace
DBTEMP	指定您想要 GBase 8s Enterprise Gateway 产品将它们的临时文件和临时表置于其内的目录的完整路径名称。

受支持的 GBase 8s 环境变量	描述
	驱动程序不使用此变量；它只是将该值传至服务器。
DBUPSPACE	当它在 <code>sqexplain.out</code> 文件中构建多列分布，或者以索引排序，以及或者为计算列分布来保存规划时，指定 <code>UPDATE STATISTICS</code> 语句可用于行排序所需的磁盘空间和内存的数量。
DELIMIDENT	当设置为 <code>Y</code> 时，指定由双引号划分的字符串为定界的标识符
ENABLE_TYPE_CACHE	<p>当设置为 <code>TRUE</code> 时，为 <code>opaque</code>、<code>distinct</code> 或 <code>row</code> 数据类型高速缓存数据类型信息。</p> <p>当 <code>Struct</code> 或 <code>SQLData</code> 对象将数据插入至列内，且 <code>getSQLTypeName()</code> 返回类型名称时，驱动程序使用高速缓存了的信息，而不是查询数据库服务器。</p>
ENABLE_HDRSWITCH	当设置为 <code>TRUE</code> 时，如果主服务器不可用，则使用辅助服务器属性来连接至辅助服务器。
FET_BUF_SIZE	<p>对于除大对象之外的所有数据，覆盖访存缓冲区大小的缺省设置</p> <p>缺省大小为 4096 字节。在服务器侧 JDBC 中不支持此变量。</p>
IFX_AUTOFREE	<p>当设置为 <code>1</code> 时，如果在数据库服务器中已关闭了游标，则指定 <code>Statement.close()</code> 方法不需要网络往返地释放数据库服务器游标资源。</p> <p>在由 <code>ResultSet.close()</code> 方法显式地关闭游标之后，或通过 <code>OPTOFC</code> 环境变量隐式地关闭游标之后，数据库服务器自动释放游标资源。在已释放了游标资源之后，不可再引用该游标。要获取更多信息，请参</p>

受支持的 GBase 8s 环境变量	描述
	阅 “自动释放” 特性。
IFX_BATCHUPDATE_PER_SPEC	当设置为 1（缺省值）时，返回 SQL 语句影响的行数，通过 <code>executeBatch()</code> 方法，在批量操作中执行该语句
IFX_CODESETLOB	如果设置为大于或等于 0 的数值，则在客户机与数据库语言环境之间自动进行 TEXT 与 CLOB 数据类型的代码集转换。此变量的值确定是在内存中还是在临时文件中来进行代码集转换。如果设置为 0，则代码集转换使用临时文件。如果设置为大于 0 的值，则代码集转换发生在客户机计算机的内存中，且该值表示为转换分配的内存字节数。要获取更多信息，请参阅 使用 IFX_CODESETLOB 环境变量转换。
IFX_DIRECTIVES	确定优化器是否允许来自查询内的查询优化伪指令。在客户机上设置此值。驱动程序不使用此变量；它只是将该值传至服务器。
IFX_EXTDIRECTIVES	指定查询优化器是否允许将来 自 sysdirectives 系统目录表的外部查询优化伪指令应用于现有的应用程序中。缺省值为 OFF。可能的值： ON 接受外部优化器伪指令 OFF 不接受外部优化器伪指令 1 接受外部优化器伪指令 0 不接受外部优化器伪指令
IFX_GET_SMFLOAT_AS_FLOAT	当设置为 0（缺省值）时，将 GBase 8s SMALLFLOAT 数据类型映射至 JDBC REAL 数据类型。此设置符合

受支持的 GBase 8s 环境变量	描述
	JDBC 规范。当设置为 1 时，将 GBase 8s SMALLFLOAT 数据类型映射至 JDBC FLOAT 数据类型。此设置使得能够与较早版本的 GBase 8s JDBC Driver 相兼容。
IFX_ISOLATION_LEVEL	<p>定义在尝试同时访问同一行的进程之中的并发程度。取得 IFX_ISOLATION_LEVEL 变量的值，其特定于 GBase 8s 。缺省值为 2（Committed Read）。如果显式地设置了该值，则它返回设置的值。返回：整数。</p> <p>设置 IFX_ISOLATION_LEVEL 变量的值，其特定于 GBase 8s 。可能的值：</p> <p>0</p> <p>等同于 TRANSACTION_NONE</p> <p>1</p> <p>Dirty Read（等同于 TRANSACTION_READ_UNCOMMITTED），</p> <p>2</p> <p>Committed Read（等同于 TRANSACTION_READ_COMMITTED），</p> <p>3</p> <p>Cursor Stability（等同于 TRANSACTION_READ_COMMITTED），</p> <p>4</p> <p>Repeatable Read（等同于 TRANSACTION_REPEATABLE_READ）</p>

受支持的 GBase 8s 环境变量	描述
	<p>5</p> <p>Committed Read LAST COMMITTED（等同于 TRANSACTION_LAST_COMMITTED）</p> <p>8</p> <p>等同于 TRANSACTION_SERIALIZABLE</p> <p>在模式之后指定 U 意味着保留更新锁。（请参阅表后的重要说明。）例如，值会为：2U（等同于 SET ISOLATION TO COMMITTED READ RETAIN UPDATE LOCKS）</p> <p>下列示例展示您将用于指定隔离级别的代码：</p> <pre>conn.setTransactionIsolation (IfxConnection.TRANSACTION_LAST_COMMITTED);</pre>
IFX_FLAT_UCSQ	覆盖全局设置，并指导优化器对所有会话使用子查询扁平化。缺省值为 1。
IFX_LOCK_MODE_WAIT	<p>应用程序可使用此属性来覆盖用于访问锁定的行或表的缺省服务器进程。取得 IFX_LOCK_MODE_WAIT 变量的值，其特定于 GBase 8s 。缺省值为 0（不等待锁）。如果显式地设置了该值，则它返回设置的值。</p> <p>返回：整数。</p> <p>设置 IFX_LOCK_MODE_WAIT 变量的值，其特定于 GBase 8s 。可能的值：</p> <p>-1</p> <p>等待，直到释放锁为止。</p> <p>0</p> <p>不等待，结束操作，并返回错误。</p>

受支持的 GBase 8s 环境变量	描述
	<p>nn</p> <p>等待释放锁 nn 秒。</p>
IFX_PAD_VARCHAR	<p>当使用 Connection 类时，可在连接 URL 上设置，或当使用 DataSource 类时，设置作为属性。有效值为 0（缺省值）和 1。</p> <ul style="list-style-type: none">当设置为 0 时，仅传送包含数据的 VARCHAR 的一部分（截去后跟的空格）。当设置为 1 时，将整个 VARCHAR 数据结构在服务器间传送。
IFX_SET_FLOAT_AS_SMFLOAT	<p>当设置为 0（缺省值）时，将 JDBC FLOAT 数据类型映射至 GBase 8s FLOAT 数据类型。此设置符合 JDBC 规范。当设置为 1 时，将 JDBC FLOAT 数据类型映射至 GBase 8s SMALLFLOAT 数据类型。此设置使得能够与较早版本的 GBase 8s JDBC Driver 兼容。</p>
IFX_TRIMTRAILINGSPACES	<p>移除后跟的空格。缺省值为 1。</p>
IFX_USEPUT	<p>当设置为 1 时，启用批量插入。要获取更多信息，请参阅 执行批量插入。</p>
IFX_XASPEC	<p>当设置为 y 时，紧密地结合带有相同全局 ID 的 XA 事务，并分享锁空间。这仅适用于 XA 连接，且不可在数据库 URL 中指定。通过 DataSource setter（请参阅 DataSource 扩展），或通过设置带有相同名称的 System（JVM）属性，可设置它。</p> <p>DataSource 属性覆盖 System 属性。忽略 y、Y、n 或 N 之外的属性值。</p> <p>IfxDataSource.getIfxIFX_XASPEC 返回最终的 IFX_SPEC 值，其或为 y 或为 n。例如，如果</p>

受支持的 GBase 8s 环境变量	描述
	DataSource IFX_XASPEC 的值等于 n 且 System IFX_XASPEC 的值等于 Y 或 y, 则返回 n。
IFX_XASTDCOMPLIANCE_XAEND	<p>当返回 XA_RB* 时, 指定 XA_END 的行为。</p> <p>0</p> <p>不忘记 XID。事务处于 Rollback Only 状态。这符合 XA_SPEC+, 且是 GBase 8s 的默认行为。</p> <p>1</p> <p>忘记 XID。事务为 Nonexistent。</p> <p>要获取更多信息, 请参阅《GBase 8s SQL 指南: 参考》</p> <p>DISABLE_B162428_XA_FIX (IDS 10.0)</p> <p>ENABLE_B162428_XA_FIX (IDS 9.40)</p>
IFXHOST	设置主机名称和主机 IP 地址
IFXHOST_SECONDARY	设置辅助主机名称和主机 IP 地址, 用于 HAC 连接重定向
GBASEDBTCONRETRY	指定附加的连接尝试的最大数, 在由 GBASEDBTCONTIME 的值指定的时间限制期间, 客户机可尝试连接至每一数据库服务器
GBASEDBTCONTIME	为连接至数据库服务器的尝试设置超时期间。如果在此期间连接尝试不成功, 则终止尝试, 并报告连接错误。缺省值为 0 秒。此变量为阻塞 socket 方法以及为 socket 连接添加超时。
GBASEDBTOPCACHE	为客户机应用程序的 staging-area blob space 指定内存高速缓存的大小
GBASEDBTSERVER	指定客户机应用程序对其进行显式或隐式连接的缺省数据库服务器

受支持的 GBase 8s 环境变量	描述
GBASEDBTSERVER_SECONDARY	指定 HAC 对中的辅助数据库服务器，如果主数据库服务器不可用，则由客户机应用程序对其进行显式或隐式连接。
GBASEDBTSTACKSIZE	指定以 KB 计的堆栈大小，数据库服务器将其用于特殊的客户机会话
JDBCTEMP	指定在哪里创建用于处理智能大对象的临时文件。您必须提供绝对路径名称。
LOBCACHE	<p>确定从数据库服务器访存的大对象数据的缓冲区大小。可能的值为：</p> <p>大于 0 的数值</p> <p>在内存中分配来保存数据的最大字节数。如果数据大小超过 LOBCACHE 值，则将数据存储在与临时文件中；如果在创建此文件期间发生安全违规，则将数据存储在内存中。</p> <p>零</p> <p>始终将数据存储在与文件中。如果发生安全违规，则驱动程序不尝试在内存中存储数据。</p> <p>负数</p> <p>始终将数据存储在内存中。如果得不到所需的内存量，则发生错误。</p> <p>如果未指定 LOBCACHE 值，则缺省为 4096 字节。</p>
LOGINTIMEOUT	确定 GBase 8s 数据库服务器是否正在运行。如果服务器正在运行，则立即建立至服务器的连接。如果服务器未运行，则此环境变量指定监听该服务器端口多少毫秒，以建立连接。如果在指定的时间内，您的应用程序未连接至 GBase 8s 数据库服务器，则返回错误。
NEWNLSMAP	允许在 NLS 与 JDK 语言环境以及要定义的 JDK 代码集之间映射。

受支持的 GBase 8s 环境变量	描述
	要获取更多信息，请参阅 用户定义的语言环境 。
NODEFDAC	当在未以 LOG MODE ANSI 创建了数据库中创建新的表或 UDR 时，在缺省情况下，防止 PUBLIC 组接收表或例程权限。必需的 yes 设置是区分大小写的。
OPT_GOAL	指定优化器的查询性能目标。在启动应用程序之前，请在用户环境中设置此变量。驱动程序不使用此变量；它只是将该值传给服务器。
OPTCOMPIND	指定查询优化器使用的连接方法
OPTOFC	当设置为 1 时，如果在客户机元组缓冲区中已检索了所有符合条件的行，则 ResultSet.close() 方法不需要网络往返。在检索了所有行之后，数据库服务器自动关闭游标。在调用下一ResultSet.next() 方法之前，在客户机元组缓冲区中，GBase 8s JDBC Driver 可能没有附加的行。因此，除非 GBase 8s JDBC Driver 已从数据库服务器接收了所有行，否则，当设置 OPTOFC 为 1 时，ResultSet.close() 方法可能仍需要网络往返。
PATH	为可执行程序指定要搜索的目录
PDQPRIORITY	确定数据库服务器使用的并行程度
PORTNO_SECONDARY	指定 HAC 对中辅助数据库服务器的端口号。该端口号罗列在 /etc/services 文件中。
PROXY	指定 HTTP 代理服务器。要获取更多信息，请参阅 HTTP 代理服务器 。
PSORT_DBTEMP	指定当执行排序时，数据库服务器将它使用的临时文件写至的一个或多个目录
PSORT_NPROCS	通过为排序分配更多线程，来使得数据库服务器能够提高并行处理排序包的性能

受支持的 GBase 8s 环境变量	描述
SECURITY	使用 56 位加密来将口令发送至服务器。要获取更多信息，请参阅 口令加密。
SQLH_TYPE	<p>当设置为 FILE 时，指定在 sqlhosts 文件中指定的数据库信息（诸如 host-name、port-number、user 和 password）。</p> <p>当设置为 LDAP 时，指定在 LDAP 服务器中指定此信息。要获取更多信息，请参阅 动态地读取 GBase 8s sqlhosts 文件。</p>
SQLIDEBUG	指定要将二进制 SQLI 踪迹写至其中的文件的路径名称。为每个连接生成新的踪迹文件，并以时间戳做前缀。当受到 GBase 技术支持代表指导时，请仅使用 SQLI 踪迹功能。
SRV_FET_BUF_SIZE	在与其他数据库服务器的分布式事务中，覆盖访存缓冲区大小的缺省设置。例如，该访存缓冲区保存由跨服务器分布式查询检索的数据。
STMT_CACHE	<p>当设置为 1 时，使得在会话中能够使用共享语句高速缓存。</p> <p>此特性可减少内存消耗，并提高不同用户会话之间的查询处理速度。驱动程序不使用此变量；它只是将该值传给服务器。</p>
TRUSTED_CONTEXT	当设置为 TRUE 时，从客户机发送受信的连接请求。或者建立成功的受信连接，或者从服务器返回下列错误：SQL Exception: -28021(Trusted Connection request rejected.)

要获取特殊环境变量的详尽描述，请参阅 《GBase 8s SQL 指南：参考》。

代码示例 IFX_LOCK_MODE_WAIT 环境变量

IFX_LOCK_MODE_WAIT

```
Connection conn = DriverManager.getConnection ( "jdbc:GBASEDBT-sqli://cleo:1550:
GBASEDBTSERVER=cleo_921;IFXHOST=cleo;PORTNO=1550;user=rdtest;
password=my_passwd;
IFX_LOCK_MODE_WAIT=1");
```

代码示例 IFX_ISOLATION_LEVEL 环境变量

IFX_ISOLATION_LEVEL

```
Connection conn = DriverManager.getConnection( "jdbc:GBASEDBT-sqli://cleo:1550:
GBASEDBTSERVER=cleo_921;IFXHOST=cleo;PORTNO=1550;user=rdtest;
password=my_passwd;
IFX_ISOLATION_LEVEL=1U");
```

重要：当它是至数据库的显式连接时，可在 URL 中设置隔离属性。对于仅限于服务器的连接，在连接时刻忽略此属性。

代码示例 IFX_ISOLATION_LEVEL 环境变量

```
Connection conn = DriverManager.getConnection( "jdbc:gbasedbt-sqli://localhost:9088
/csdk_db:GBASEDBTSERVER=ol_ids_1150_1;user=gbasedbt;password=inform123;
LOGINTIMEOUT=60000");
```

3.6 动态地读取 GBase 8s sqlhosts 文件

GBase 8s JDBC Driver 支持 JNDI（Java™ 命名和目录接口）。此支持使得 JDBC 程序能够访问 GBase 8s sqlhosts 文件。sqlhosts 文件允许客户机应用程序找到并连接至网络上任何位置的 GBase 8s 数据库服务器。要获取关于此文件的更多信息，请参阅《GBase 8s 管理员指南》。

您可从本地文件或从 **LDAP** 服务器来访问 sqlhosts 数据。系统管理员必须使用 GBase 8s 实用程序，来将 sqlhosts 数据加载至 LDAP 服务器内。

您的 CLASSPATH 变量必须引用 JNDI JAR（Java 归档）文件和 LDAP SPI（服务提供商接口）JAR 文件。您必须使用 LDAP Version 3.0 或更新的版本，其支持对象类 **extensibleObject**。

您可使用 sqlhosts 文件 group 选项，来为 GBASEDBTSERVER 的值指定数据库服务器组的名称。group 选项对“高可用性数据复制”（HAC）非常有用；顺序地在 HAC 对中罗列主和辅助数据库服务器。要获取关于如何设置和使用 sqlhosts 文件中的组的更多信息，请参阅《GBase 8s 管理员指南》。要获取关于 HAC 的更多信息，请参阅 连接至高可用性集群的服务器。

未签名的 applet 不可访问 sqlhosts 文件或 LDAP 服务器。要获取更多信息，请参阅 在 applet 中使用驱动程序。

3.6.1 连接属性语法

您可允许 GBase 8s JDBC Driver 查找 LDAP 服务器中的主机名称和端口号，而不是在数据库 URL 或 DataSource 对象中直接指定它们。必须为 LDAP 服务器指定数据库 URL 或 DataSource 对象中的下列属性：

- SQLH_TYPE=LDAP
- LDAP_URL=ldap://host-name:port-number
host-name 和 port-number 是 LDAP 服务器的属性，不是数据库服务器。
- LDAP_IFXBASE=GBASEDBT-base-DN
- LDAP_USER=user
- LDAP_PASSWD=password

如果未指定 **LDAP_USER** 和 **LDAP_PASSWD**，则 GBase 8s JDBC Driver 使用匿名搜索来搜索 LDAP 服务器。LDAP 管理员必须确保在 sqlhosts 条目上允许匿名搜索。要获取更多信息，请参阅您的 LDAP 服务器资料。

GBASEDBT-base-DN 有下列基本格式：

```
cn=common-name,o=organization,c=country
```

如果 common-name、organization 或 country 由多个词组成，则可为每一词使用一个条目。例如：

```
cn=gbasedbt,cn=software
```

这里是一个示例数据库 URL：

```
jdbc:gbasedbt-sqli:gbasedbtserver=value;SQLH_TYPE=LDAP;  
LDAP_URL=ldap://davinci:329;LDAP_IFXBASE=cn=gbasedbt,  
cn=software,o=kmart,c=US;LDAP_USER=abcd;LDAP_PASSWD=secret
```

您还可在数据库 URL 或 **DataSource** 对象中指定 sqlhosts 文件。从 sqlhosts 文件读取 GBase 8s 数据库服务器的主机名称和端口号或服务名称，如同在 /etc/services 文件中指定的那样。必须为该文件指定下列属性：

- SQLH_TYPE=FILE
- SQLH_FILE=sqlhosts-filename

sqlhosts 文件可以是本地的或远程的，因此，您可以本地系统文件格式或以 URL 格式来引用它。这里是一些示例：

- SQLH_FILE=http://host-name:port-number/sqlhosts.ius
SQLH_FILE=http://host-name:service-name/sqlhosts.ius

GBase 8s 数据库服务器（来自 etc/services 文件）元素的 host-name 和 port-number 或 service-name 是在其上驻留 sqlhosts 文件的服务器的那些属性。

- SQLH_FILE=file:///D:/local/myown/sqlhosts.ius
- SQLH_FILE=/u/local/sqlhosts.ius

这里是示例数据库 URL:

```
jdbc:gbasedbt-sqli:gbasedbtserver=value;SQLH_TYPE=FILE;  
SQLH_FILE=/u/local/sqlhosts.ius
```

如果数据库 URL 或 DataSource 对象引用 LDAP 服务器或 sqlhosts 文件，而且直接指定 IP 地址、主机名称和端口号，则在数据库 URL 或 DataSource 对象中指定的 IP 地址、主机名称和端口号优先。要获取关于如何通过使用 DataSource 对象来设置这些连接属性的信息，请参阅 DataSource 扩展。

如果您正在使用 applet 或位于防火墙之后的数据库，则需要运行在额外层中的 HTTP 代理服务器，用于通讯。要获取更多信息，请参阅 HTTP 代理服务器。

3.6.2 管理要求

如果您想要 LDAP 服务器来存储 JDBC 可查找的 sqlhosts 信息，则必须满足下列要求：

- 必须在客户机可访问的计算机上安装 LDAP 服务器。LDAP 管理员必须在 LDAP 服务器中创建 BASE 条目。
- 如果您想要使用 GBase 8s SqlhUpload 和 SqlhDelete 实用程序，其可从文本 ASCII 文件加载或删除 sqlhosts 条目，则 sqlhosts 文件中的 servicename 字段必须指定数据库服务器端口号。要获取更多信息，请参阅以 sqlhosts 数据来更新 LDAP 服务器的实用程序。
- LDAP 管理员必须确保在 sqlhosts 条目上允许匿名搜索。要获取更多信息，请参阅 LDAP 服务器资料。

3.6.3 以 sqlhosts 数据来更新 LDAP 服务器的实用程序

在 gbasedbtjdbc_xx.jar 中包装 SqlhUpload 和 SqlhDelete 实用程序，因此，CLASSPATH 变量必须指向 gbasedbtjdbc_xx.jar（在缺省情况下，其位于 GBase 8s JDBC Driver 的安装目录之下的 lib 目录中）。请确保 CLASSPATH 变量也指向 JNDI JAR 文件和 LDAP SPI JAR 文件。

SqlhUpload 实用程序

此实用程序以规定的格式，从文本 ASCII 文件将 sqlhosts 条目加载至 LDAP 服务器。

输入下列命令：

```
java SqlhUpload sqlhfile.txt host-name:port-number [sqlhostsRdn]
```

参数有下列含义：

- 要上载的 sqlhosts 文件为 sqlhfile.txt。
- LDAP 服务器的主机名称和端口号为 host-name:port-number。
- 在 LDAP 中 GBase 8s 基础之下的 sqlhosts 节点的 RDN[®]（相对可分辨名称）为 sqlhostsRdn。缺省名称为 sqlhosts。

实用程序提示提供其他必需的信息，诸如 LDAP 服务器中的 GBase 8s 基础可分辨名称（DN）、LDAP 用户和口令。

您必须将 sqlhosts 文件中的 **servicename** 字段转换为表示整数（端口号）的字符串，因为 Java[™].Socket 类不可接受端口号的字母数字的 **servicename** 值。要获取关于 servicename 字段的更多信息，请参阅《GBase 8s 管理员指南》。

SqlhDelete 实用程序

此实用程序从 LDAP 服务器删除 sqlhosts 条目。请输入下列命令：

```
java SqlhDelete host-name:port-number [sqlhostsRdn]
```

此命令的参数与为 SqlhUpload 实用程序罗列的参数有相同的含义。请参阅 SqlhUpload 实用程序。

实用程序提示提供其他必需的信息，诸如 LDAP 服务器中的 GBase 8s 基础 DN、LDAP 用户和口令。

3.7 连接至高可用性集群的服务器

使用 JDBC 驱动程序，Java[™] 应用程序可连接至高可用性集群中的 GBase 8s 数据库服务器。Java 应用程序还可连接至 GBase 8s Connection Managers，其可为高可用性集群处理故障转移，并将连接重定向至集群服务器。

Java 应用程序要连接至高可用性集群的服务器，您必须在连接 URL 或 DataSource 中设置属性。如果应用程序在辅助服务器上执行更新操作，则请配置应用程序来初始地检查只读服务器状态。

当您配置 GBase 8s Connection Managers 来处理 Java 应用程序服务器与高可用性集群之间的连接时，可获得下列好处：

1. 通过基于规则的重新指向策略，将连接请求指向最正确的辅助服务器。
2. 您可为高可用性集群管理故障转移，如果主服务器故障，则自动地将一辅助服务器提升为主服务器的角色。

3. 当您在与 Java 应用程序服务器相同的主机上安装并配置 GBase 8s Connection Managers 时，可在特定的应用程序服务器与高可用性集群的主服务器之间排出优先级。
4. 当数据库服务器位于防火墙之后时，GBase 8s Connection Managers 可担当代理服务器，并处理客户机/服务器通讯。

可使用带有连接池的高可用性辅助服务器。要获取更多信息，请参阅 高可用数据复制与连接池。

3.7.1 通过 GBase 8s Connection Manager 连接至高可用性集群服务器的属性

JDBC 应用程序可连接至 Connection Manager，就像应用程序可能连接至数据库服务器一样。然后，将应用程序连接请求重新指向高可用性集群中最恰当的服务器。

可配置多个 Connection Manager，然后在 Java™ 应用程序服务器使用的 sqlhost 文件中创建一 Connection Manager 组条目。如果一个 Connection Manager 故障，则可将连接请求指向正在工作的 Connection Manager。SQLH_FILE 连接属性指导 JDBC 驱动程序搜索组条目。

要连接至一 GBase 8s Connection Manager，然后再通过其连接至高可用性集群的服务器，您必须在连接 URL 或 DataSource 中包括下列属性：

```
GBASEDBTSERVER=CM_or_group_name
SQLH_TYPE=FILE
SQLH_FILE=sqlhosts
USER=user_name
PASSWORD=password
```

如果 Connection Manager 正在运行，但有一个挂起的连接，则在连接 URL 中包括下列属性，可防止 Java 应用程序无限地等待。

```
GBASEDBTCONRETRY=value
GBASEDBTCONTIME=value
LOGINTIMEOUT=value
```

基于网络环境来设置这些值。

示例 1：通过 GBase 8s Connection Manager 来连接至高可用性集群

在此示例中，您有下列系统设置：

- 由四个服务器组成的高可用性集群（**my_cluster**）。
- 所有集群服务器上的用户名都是 **my_user**。

- 所有集群服务器上的口令都是 **my_password**。
- **cmhost1.example.com** 上的 **connection_manager** 使用下列配置文件：

```
NAME connection_manager
  CLUSTER my_cluster
  {
    GBASEDBTSERVER my_servers
    SLA sla_primary    DBSERVERS=PRI
    SLA sla_secondaries DBSERVERS=SDS,HDR,RSS
  }
```

- 在 **host1.example.com** 上有一 Java 应用程序服务器，该 Java 应用程序服务器使用下列 **sqlhost** 文件条目：

#dbservername	nettype	hostname	servicename	options
sla_primary	onsoctcp	cmhost1.example.com	cm_port_1	
sla_secondaries	onsoctcp	cmhost1.example.com	cm_port_1	

- 如果客户机的初始连接尝试失败，则想要它再试两次。
- 想要 **CONNECT** 语句等待 10 秒，来建立连接。
- 如果侦听服务器端口，且在 10 毫秒之内未连接，则想要连接失败。

要 Java 应用程序客户机连接至 **my_cluster** 的主服务器，请使用下列 URL：

```
jdbc:gbasedbt-sqli://GBASEDBTSERVER=sla_primary;
  SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
  USER=my_user_name;PASSWORD=my_password;
GBASEDBTCONRETRY=2;GBASEDBTCONTIME=10;LOGINTIMEOUT=10
```

要 Java 应用程序客户机连接至 **my_cluster** 的辅助服务器，请使用下列 URL：

```
jdbc:gbasedbt-sqli://GBASEDBTSERVER=sla_secondaries;
  SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
  USER=my_user_name;PASSWORD=my_password;
GBASEDBTCONRETRY=2;GBASEDBTCONTIME=10;LOGINTIMEOUT=10
```

示例 2：通过多个 GBase 8s Connection Manager 连接至高可用性集群

在此示例中，您有下列系统设置：

- 由四个服务器组成的高可用性集群（**my_cluster**）。
- 所有集群服务器上的用户名都是 **my_user**。
- 所有集群服务器上的口令都是 **my_password**。
- **cmhost1.example.com** 上的 **connection_manager_1** 使用下列配置文件，用于客户机重新指向和故障转移：

```
NAME connection_manager_1
  CLUSTER my_cluster
  {
    GBASEDBTSERVER my_servers
    SLA sla_primary_1 DBSERVERS=PRI
    FOC ORDER=ENABLED \
    PRIORITY=1
    CMALARMPROGRAM $GBS_HOME/etc/CMALARMPROGRAM.sh
  }
```

- cmhost2.example.com 上的 connection_manager_2 使用下列配置文件，用于客户机重新指向和故障转移：

```
NAME connection_manager_2
  CLUSTER my_cluster
  {
    GBASEDBTSERVER my_servers
    SLA sla_primary_1 DBSERVERS=PRI
    FOC ORDER=ENABLED \
    PRIORITY=2
    CMALARMPROGRAM $GBS_HOME/etc/CMALARMPROGRAM.sh
  }
```

- 在 host1.example.com 上有一 Java 应用程序服务器，且 Java 应用程序服务器使用下列 sqlhost 文件条目：

#dbservername	nettype	hostname	servicename	options
g_primary	group	-	-	c=1,e=sla_primary_2
sla_primary_1	onsoctcp	cmhost1.example.com	cm_port_1	g=g_primary
sla_primary_2	onsoctcp	cmhost2.example.com	cm_port_2	g=g_primary

- 如果客户机的初始连接尝试失败，则想要它重试两次。
- 想要 CONNECT 语句等待 10 秒，来建立连接。
- 如果侦听服务器端口，且在 10 毫秒之内未连接，则想要连接失败。

要通过 **connection_manager_1** 或 **connection_manager_2**，Java 应用程序客户机连接至 **my_cluster** 的主服务器，请使用下列 URL：

```
jdbc:gbasedbt-sqli://GBASEDBTSERVER=g_primary;
SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
USER=my_user_name;PASSWORD=my_password;
GBASEDBTCONRETRY=2;GBASEDBTCONTIME=10;LOGINTIMEOUT=10
```

3.7.2 通过 **SQLHOST** 文件组条目，来连接至高可用性集群服务器的属性

您可定义 sqlhost 组条目，以便于始终将应用程序连接尝试指向高可用性集群的主服务器，即使发生故障转移。

要连接至高可用性集群的主服务器，请在连接 URL 或 DataSource 中包括下列属性：

```
GBASEDBTSERVER=group_name
    SQLH_TYPE=FILE
    SQLH_FILE=sqlhosts
    USER=user_name
    PASSWORD=password
```

如果 JDBC 驱动程序未能在该组中找到主服务器，则抛出异常。

为 GBase 8s 启用至主服务器的强制连接。

示例：通过 **SQLHOST** 文件组条目来连接至高可用性集群的主服务器

在此示例中，有下列系统设置：

- 由四个服务器组成的高可用性集群（my_cluster）：
 - host1.example.com 上的 server_1（主）
 - host1.example.com 上的 server_2（共享磁盘辅助）
 - host2.example.com 上的 server_3（HAC）
 - host3.example.com 上的 server_4（远程独立辅助）
- 所有集群服务器上的用户名都是 my_user。
- 所有集群服务器上的口令都是 my_password。
- 在 host4.example.com 上有一 Java™ 应用程序服务器。该服务器使用下列 sqlhost 文件条目：

#dbservername	nettype	hostname	servicename	options
my_servers	-	-		c=1,e=server_4
server_1	onsoctcp	host1.example.com	port_1	g=my_servers
server_2	onsoctcp	host1.example.com	port_2	g=my_servers
server_3	onsoctcp	host2.example.com	port_3	g=my_servers
server_4	onsoctcp	host3.example.com	port_4	g=my_servers

要 Java 应用程序客户机连接至 my_cluster 的主服务器，请使用下列 URL：

```
jdbc:gbasedbt-sqli://GBASEDBTSERVER=my_servers;
SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
USER=my_user_name;PASSWORD=my_password
```

3.7.3 直接连接至服务器的 HAC 对的属性

您可定义客户机应用程序的连接 URL 或 DataSource，以便于应用程序直接地连接到服务器的 HAC 对。如果至主服务器的连接尝试失败，则客户机应用程序可尝试连接至 HAC 辅助服务器。

要直接地连接至主服务器和 HAC 辅助服务器，请在连接 URL 或 DataSource 中包括下列属性：

```
GBASEDBTSERVER=primary_server_name
GBASEDBTSERVER_SECONDARY=secondary_server_name
IFXHOST_SECONDARY=secondary_host_name
PORTNO_SECONDARY=secondary_port_number
USER=user_name
PASSWORD=password
ENABLE_HDRSWITCH=true
```

如果您正在设置 DataSource 中的值，则还必须包括下列值：

```
IFXHOST=primary_host_name
PORTNO=primary_port_number
```

当正在使用 DataSource 对象时，您可以 setXXX() 和 getXXX() 方法，来设置或取得辅助服务器连接属性。在 获取和设置 GBase 8s 连接属性中，罗列这些方法对应的连接属性。

通过编辑 DataSource 中的 GBASEDBTSERVER、PORTNO 和 IFXHOST 属性，或通过编辑 URL 中的 GBASEDBTSERVER 属性，您可手工地将连接重新指向 HAC 对中的辅助服务器。手工重新指向需要编辑应用程序代码，然后重新启动应用程序。

示例：连接至服务器的 HAC 对

下列示例展示名为 server_1 的主服务器和名为 server_2 的 HAC 辅助服务器的连接 URL：

```
jdbc:gbasedbt-sqli://my_host:my_port/my_database:
GBASEDBTSERVER=server_1;GBASEDBTSERVER_SECONDARY=server_2;
IFXHOST_SECONDARY=host2.example.com;PORTNO_SECONDARY=port_2;
user=my_name;password=my_password;
ENABLE_HDRSWITCH=true
```

3.7.4 检查高可用性辅助服务器的只读状态

可编写应用程序来检查只读服务器状态，以便于不对只读辅助服务器尝试更新操作。

GBase 8s JDBC 有对 java.sql.Connection 类的扩展方法，其提供检查 HAC 辅助服务器的状态的方式。用户可将连接对象强制转型为 'com.gbasedbt.jdbc.IfmxConnection' 来访问下列扩展方法。

获得的信息	方法签名	附加信息
服务器是否为只读（辅助服务器）	public boolean 为 ReadOnly() throws SQLException	如果活动的服务器为辅助服务器，则返回 true 如果发生数据库访问错误，则返回异常 如果将 ENABLE_HDRSWITCH 设置为 false，则isReadOnly() 返回在获得了最后一次成功的 HAC 连接之后初始设置的值。
是否启用 HAC	public Boolean is HDREnabled()	如果 HAC 对中的服务器都可用，则返回 true 如果有一个服务器不可用，则返回 false
服务器的类型（主、辅助或标准）	public string getHDRtype()	对于主服务器返回 primary 或 standard,对于辅助服务器返回 secondary 数据库管理员可手工地重置服务器的类型。

例如，您可使用下列策略之一：

- 在每一可能包含更新操作的 SQL 语句之前，使用 isReadOnly() 方法。如果 isReadOnly() 的值为 true，则执行恰当的行动，诸如将错误消息发送至用户，或通知服务器管理员。
- 在建立连接之后，调用 isReadOnly() 方法，然后设置像 READ_ONLY 这样的标志，然后，基于标志值来执行操作。

管理员可手工地将辅助服务器切换为主服务器，以允许更新操作。然而，必须在进程中关闭服务器，这可导致丢失未提交的事务。

3. 7. 5 对 HAC 辅助服务器的连接重试尝试

您可编写应用程序，以便于如果在查询操作期间连接丢失，则 GBase 8s JDBC Driver 将新的连接返回至辅助数据库服务器，且应用程序重新运行查询。

下列示例展示如何尝试重新与辅助服务器信息连接，然后，重新运行由于主服务器连接失败而收到错误的 SQL 语句：

```
public class HDRConnect {
    static IfmxConnection conn;
    public static void main(String[] args)
    {
        getConnection(args[0]);
        doQuery( conn );
        closeConnection();
    }

    static void getConnection( String url )
    {
        ..
        Class.forName("com.gbasedbt.jdbc.Driver");
        conn = (IfmxConnection)DriverManager.getConnection(url);
    }

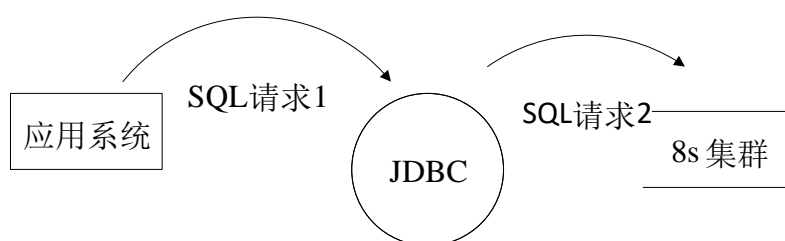
    static void closeConnection()
    {
        try
        {
            conn.close();
        }
        catch (SQLException e)
        {
            System.out.println("ERROR: failed to close the connection!");
            return;
        }
    }

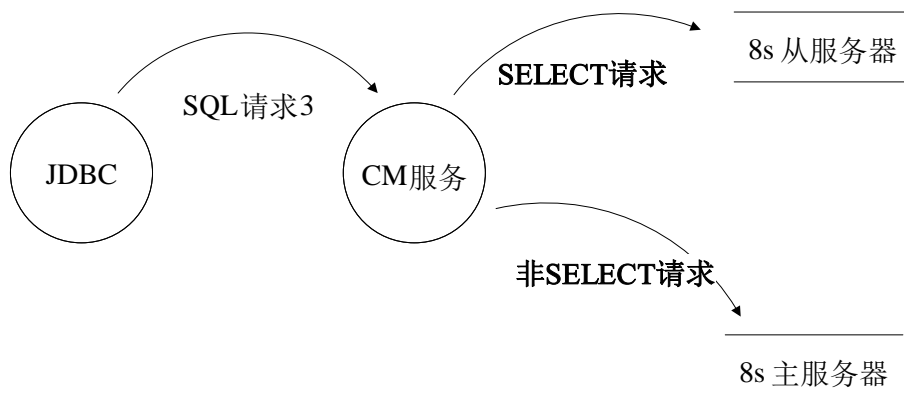
    static void doQuery(Connection con)
    {
        int rc=0;
        String cmd=null;
        Statement stmt = null;
```

```
try
{
// execute some sql statement
}
catch (SQLException e)
{
if (e.getErrorCode() == -79716 ) || (e.getErrorCode() == -79735)
// system or internal error
{
// This is expected behavior when primary server is down
getConnection(url);
doQuery(conn);
}
else
System.out.println("ERROR: execution failed - statement: " + cmd);
return;
}
}
```

3. 7. 6 JDBC读写分离

JDBC读写分离功能是产品的JDBC中扩展的一项功能,主要是为实现8s数据库产品在集群部署时读写分离功能,使主从服务器得到充分利用。在主节点性能瓶颈前提下,读写分离性能高于非读写分离,并且随着从节点扩展,读写分离的查询性能有线性提升。





数据流	说明
SQL请求1	此处为应用系统要通过JDBC发送给数据库的所有SQL语句
SQL请求2	此处为经过JDBC读写分离功能处理后的SQL请求
SQL请求3	此处为经过JDBC读写分离功能处理后的SQL请求
SELECT请求	此处为经过JDBC读写分离功能处理后的以“SELECT”开头的SQL请求
非SELECT请求	此处为经过JDBC读写分离功能处理后的以非“SELECT”开头的SQL请求

➤ JDBC读写分离功能部署过程：

1. 首先部署带读写分离功能的JDBC

应用端替换带读写分离功能的JDBC，比如gbasedbtjdbc_3.2.0_6_f937fc.jar。

2. 配置CM文件

其次配置集群服务器，并将集群配置文件sqlhosts.cm文件添加到应用端。配置内容为CM读写服务器地址。

比如增加sqlhosts.cm文件，文件内容如下：

```
db_group group - - i=10,c=1
perf178 onsoctcp 172.16.2.178 16000 g=db_group
perf178 onsoctcp 172.16.2.178 16000 g=db_group

cm_update group - - i=12,c=0
w_perf178 onsoctcp 172.16.2.178 18888 g=cm_update
w_perf123 onsoctcp 172.16.3.123 18888 g=cm_update

cm_read group - - i=13,c=0
r_perf178 onsoctcp 172.16.2.178 18888 g=cm_read
r_perf123 onsoctcp 172.16.3.123 18888 g=cm_read
```


3. 配置从服务器地址

用解压软件打开驱动程序gbasedbtjdbc_3.2.0_6_f937fc.jar，在jar包根路径下存在rws.properties文件。修改rws.properties文件中jdbcUrl字段内容，将jdbcUrl指向CM服务器从服务器地址。

按照2的配置，修改如下：

```
jdbcUrl=jdbc:gbasedbt-sqli:/testdb:gbasedbtserver=cm_read;SQLH_TYPE=FILE;SQLH_FILE=C:/jdbc_rws/hdr/sqlhosts.cm;DB_LOCALE=zh_cn.GB18030-2000;CLIENT_LOCALE=zh_cn.GB18030-2000;
```

其中gbasedbtserver指向CM配置读端服务器，修改完成后，将rws.properties文件放置到gbasedbtjdbc_3.2.0_6_f937fc.jar根路径下。

4. 配置应用连接参数

应用端jdbc连接串信息设置如下：

```
jdbc:gbasedbt-sqli:/testdb:gbasedbtserver=cm_update;SQLH_TYPE=FILE;SQLH_FILE=C:/jdbc_rws/hdr/sqlhosts.cm;rws=true;DB_LOCALE=zh_cn.GB18030-2000;CLIENT_LOCALE=zh_cn.GB18030-2000;
```

其中gbasedbtserver指向CM配置写端服务器，设置读写分离标志rws，配置rws=true表示启动读写分离功能。rws=false不启动读写分离功能，默认值为false。

经过以上四个步骤，完成JDBC读写分离功能配置。用户端执行sql语句将按照读写分离的原则，将从不同的服务器执行读写操作。

执行用例验证读写分离：

```
String url=
jdbc:gbasedbt-sqli:/testdb:gbasedbtserver=cm_update;SQLH_TYPE=FILE;SQLH_FILE=C:/jdbc_rws/hdr/sqlhosts.cm;rws=true;DB_LOCALE=zh_cn.GB18030-2000;CLIENT_LOCALE=zh_cn.GB18030-2000;
Connection con=null;
try{
    conn=DriverManager.getConnection(url);
}catch(SQLException e){
    e.printStackTrace();
}
String sql_update="insert into tb1 values(10,10);";
String sql="select * from tb1;";
Statement stmt=conn.createStatement();
stmt.execute(sql_update);
ResultSet rs=stmt.executeQuery(sql);
rs.close();
```

```
stmt.close();  
conn .close();
```

rws=1, sql_update执行，写入主服务器数据。

rws=1,sql执行，从从服务器上获取数据。

➤ rws.properties文件说明：

```
driverClassName=com.gbasedbt.jdbc.Driver//驱动主类  
jdbcUrl=  
jdbc:gbasedbt-sqli:/testdb:gbasedbtserver=cm_read;SQLH_TYPE=FILE;SQLH_FILE=  
C:/jdbc_rws/hdr/sqlhosts.cm;DB_LOCALE=zh_cn.GB18030-2000;CLIENT_LOCALE=  
zh_cn.GB18030-2000;//从服务器url  
username=gbasedbt//从服务器用户名  
password=11111111//从服务器用户密码  
maximumPoolSize=100//池中最大连接数，包括闲置和使用中的连接  
minmumIdle=100//池中维护的最小空闲连接数  
connectionTestQuery=select 1 from dual;//连接池测试查询语句  
idleTimeout=180000//连接允许在池中闲置的最长时间  
connectionTimeout=180000//等待来自池的连接的最大毫秒数  
validationTimeout=3000//连接将被测试活动的最大时间量  
maxLifetime=180000//池中连接最长生命周期  
rws数据源连接池采用的是HikariCP连接池,除了driverClassName、jdbcUrl、username、  
password参数外，其余参数均为连接池参数。
```

3.8 其他多层解决方案

在多层环境中使用 GBase 8s JDBC Driver 的其他方式如下：

“远程方法调用”（RMI）

GBase 8s JDBC Driver 驻留在作为 Java™ applet 或应用程序与 GBase 8s 数据库及其之间的中间层的应用程序服务器上。GBase 8s JDBC Driver 包括一个 RMI 的示例；要了解详细信息，请参阅 示例代码文件。

其他通讯协议，诸如 CORBA

GBase 8s JDBC Driver 驻留在作为 Java applet 或应用程序与 GBase 8s 数据库计算机的中间层的应用程序服务器上。

3.9 加密选项

您可使用口令（`SECURITY=PASSWORD`）或网络加密，来建立安全的连接。要使用口令选项，或要使用网络加密，您必须在 Java™ 运行时环境中安装符合“Java 密码技术扩展”（JCE）的加密服务提供程序。JRE 1.4 或后来的版本以及 GBase JRE 1.4.2 包包括符合 JCE 的加密服务提供程序。

建议不要在同一客户机上混用安全软件包。下列主题描述如何配置每一软件包。

不要一起使用网络加密和口令加密。因此，当使用 JDBC 加密 CSM 时，不应以 `SECURITY` 环境变量来启用口令加密。在将它们发送到网络之前，JDBC 加密 CSM 不加密口令。

3.9.1 JCE 安全软件包

已将 JCE 集成至 J2 SDK Version 1.4 内，但仅在美国和加拿大可用。如果您的站点不符合此限制或其他 JCE 许可限制，则可尝试使用带有其他 JCE 认证的安全软件包提供程序的 GBase 8s JDBC Driver。然而，请注意，并非所有软件包都已测试并认证，可以与配置了使用 SPWDCSM CSM 选项或加密 CSM 的 GBase 8s 一起工作。或者，您可使用符合 GBase FIPS 的安全软件包。

如果您正在使用 JDK 1.4 来安装 JCE 软件包，则请下载 JCE 分发，抽取包含 JCE 提供程序软件包的 .jar 文件，并将它们复制至安装 JDK 的 `jre/lib/ext` 目录。

编辑来自 JDK 安装的 `lib/security/java.security` 文件，来包括下列两行：

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.crypto.provider.SunJCE
```

要获取关于配置加密服务提供程序的更多详细信息，请参阅 JRE 资料。

3.9.2 符合 GBase FIPS 的安全软件包

GBase 1.4.2 SR1a JRE 或后来的版本包括一个称为 GBASEJCEFIPS 的 JCE 符合“联邦信息处理标准”（FIPS）140-2 软件包。作为 JCE 提供程序来实现 GBASEJCEFIPS 软件包，来通过 JCE 框架 API 支持 FIPS 批准的密码操作。可随同简单的 CSM 或随同加密 CSM 来使用 GBASEJCEFIPS 软件包。

要使用 FIPS 软件包，请将 GBASEJCEFIPS 提供程序添加至 JVM `java.security` 文件中的安全提供程序列表，其位于安装 JRE 的 `jre/lib/ext` 目录中。

必须以比 `java.security` 文件中任何非 FIPS 安全提供程序更高的优先顺序，来指定 GBASEJCEFIPS 提供程序。该顺序是基于 1 的，意味着 1 是最优先的，然后是 2，依此类推。

例如：

```
security.provider.1=com.gbase.crypto.fips.provider.GBASEJCEFIPS  
security.provider.2=com.gbase.crypto.fips.provider.GBASEJCE
```

请确保 GBASEJCEFIPS 比 GBASEJCE 提供程序有更高的优先顺序。

无需对 GBase 8s JDBC Driver 进行任何应用程序更改，即可使用符合 FIPS 的密码软件包。

要了解关于配置加密服务提供程序的更详细信息，请在前面罗列的网站参阅 GBase Developer Kit 和 Runtime Environment 的 GBase JRE 资料。

3.9.3 口令加密

当 GBase 8s JDBC 客户机与 GBase 8s 数据库服务器交换数据时，SECURITY 环境变量指定执行的安全操作。在 GBase 8s JDBC Driver 中支持的 SECURITY 环境变量的唯一设置是 PASSWORD。

如果指定 PASSWORD，则当将口令从客户机传至数据库服务器时，使用 56 位加密来加密用户提供的口令。没有缺省的设置。

这里是一个示例：

```
String URL = "jdbc:gbasedbt-sqli://158.58.10.171:1664:user=myname;  
password=mypassword;GBASEDBTSERVER=myserver;SECURITY=PASSWORD";
```

PASSWORD 不区分大小写。

配置数据库服务器

在 GBase 8s 数据库服务器中，支持 SECURITY=PASSWORD 设置。

如果在 GBase 8s JDBC 客户机中指定 SECURITY=PASSWORD，则在 GBase 8s 数据库服务器上必须启用 SPWDSCSM csm 选项。否则，在连接期间会返回错误。

要使用支持在数据库服务器上口令加密的 SPWDSCSM csm 服务器选项，必须配置服务器 sqlhosts 服务器名称选项。在服务器上设置此选项之后，仅使用 SECURITY=PASSWORD 设置的客户机可连接至该服务器名称。

要查看如何配置 CSM 选项的通用详细信息，请参阅《GBase 8s 管理员指南》。

3.9.4 配置连接来使用 SSL 协议

要配置 GBase 8s JDBC Driver 的数据库连接，来使用“安全套接层”（SSL）协议，需要将 sslConnection 属性设置为 TRUE。

在至数据源的连接可使用 SSL 协议之前，必须在数据库服务器中将应用程序连接至的端口配置为 SSL 监听器端口。

请在 Connection 或 DataSource 实例上设置 sslConnection 属性。下列示例演示如何在 Connection 实例上设置 sslConnection 属性：

```
java.util.Properties properties = new java.util.Properties();
    properties.put("user", "xxxx");
    properties.put("password", "yyyy");
    properties.put("sslConnection", "true");
    java.sql.Connection con =
        java.sql.DriverManager.getConnection(url, properties);
```

3.9.5 网络加密

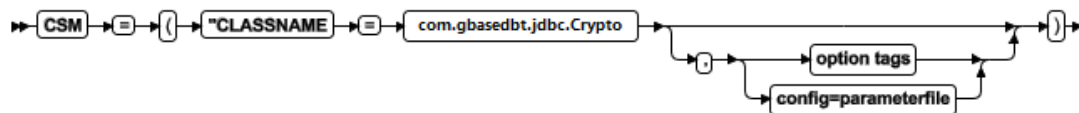
通过使用加密通讯支持模块，GBase 8s 启用网络上的数据传输加密。通过将通讯支持模块（CSM）添加至 JDBC 驱动程序，GBase 8s JDBC Driver Version 2.21.JC5 和后来的版本使得所有 JDBC 客户机都可用此特性。

GBase 8s JDBC 加密模块是包装在 GBase 8s JDBC .jar 中的 com.gbasedbt.jdbc.Crypto 类。GBase 8s JDBC 加密 CSM 是使用来自 Java™ 密码技术提供程序的纯 Java 实现。

网络加密语法

要配置网络加密，请设置 CSM 环境变量。下列语法说明 CSM 环境变量和加密选项：

CSM 环境变量语法



option tags

指定加密标记的语法。要获取更多信息，请参阅 option 标记。

config=parameterfile

指定文件中的加密选项。要获取更多信息，请参阅 选项参数。

option 标记

可传至加密 CSM 的 option 标记与服务器或 CSDK 使用的 CSM 配置文件中指定的加密 option 标记相同。有三个 option 标记：

cipher

定义会话可使用的所有密码。

mac

定义 MAC 生成期间使用的消息认证代码（MAC）键文件，以及使用的 MAC 生成级别。

switch

定义重新商议密码或密钥的频度。保持使用密钥和加密密码的时间越长，攻击者破解加密规则的可能性越大。要避免这种情况，密码学家建议定期更改长期连接上的密钥和密码。此重新商议的缺省值为每小时一次。通过使用 switch 标记，可以分钟为单位来设置此重新商议的时间。

要了解这些标记的语法，请参阅《GBase 8s 安全指南》。

用逗号将加密 CSM 选项参数分开，而不使用分号。当使用 DataSource 时，可使用 getIfxCSM() 和 setIfxCSM() 方法来取得和设置 CSM 为属性。当设置 CSM 为属性时，请确保未在圆括号中括上 option 字符串。下列为正确地设置 CSM 为属性的示例：

```
connProperties.put("CSM","classname=com.gbasedbt.jdbc.Crypto,cipher[all],  
mac[<builtin>]");
```

选项参数

通过创建带有加密参数的文件，然后指定该文件名称，可配置加密。加密参数为：

- ENCCSM_CIPHERS：要使用的密码
- ENCCSM_MAC：MAC 级别
- ENCCSM_MACFILES：MAC 文件位置
- ENCCSM_SWITCH：CIPHER 和 KEY 更改频度，以逗号隔开

要了解这些参数的语法，请参阅《GBase 8s 安全指南》。

下列是在配置文件中指定 CSM 参数的示例：

```
String newUrl = "jdbc:gbasedbt-sqli:  
    //beacon:8779/test:GBASEDBTSERVER=danon950_beacon_encrypt;  
    user=rdtest;password=test;  
    csm=(classname=com.gbasedbt.jdbc.Crypto,config=test.cfg)";  
try  
{  
    Class.forName( "com.gbasedbt.jdbc.Driver" );  
}catch( Exception e )  
{  
    System.out.println( "ERROR: failed to load  
    GBase 8s JDBC driver." );  
}  
try
```

```
{
    Connection con = DriverManager.getConnection( newUrl );
}
catch( SQLException e )
{
    System.out.println( "ERROR: failed to connect." );
    e.printStackTrace();
    return;
}
```

在服务器中配置加密 CSM

要能在加密端口上连接至 GBase 8s 数据库服务器，JDBC 客户机必须使用 JDBC 加密 CSM。当使用 JDBC 加密 CSM 时，至未加密的端口上的 GBase 8s 数据库服务器的连接尝试失败。可能配置 GBase 8s 数据库服务器的一个实例来同时监听加密的端口和未加密的端口。要获取关于将 GBase 8s 配置为使用加密 CSM 的详细信息，请参阅《GBase 8s 管理员指南》。

3. 10 关闭连接

在使用连接池与不使用连接池的环境中，下表对比调用 Connection.close() 与 scrubConnection() 方法的效果差异。

要获取关于解除资源分配的更多信息，请参阅 释放资源。要获取关于 scrubConnection() 方法的更多信息，请参阅 清除池连接。

连接池状态	调用 Connection.close() 方法的效果	调用 scrubConnection() 方法的效果
无连接池设置	关闭数据库连接、所有相关联的语句对象，以及它们的结果集。连接不再有效。	连接返回至原始状态，保持打开的语句，但关闭结果集，Connection 仍有效。 仅释放与结果集相关联的资源。
以 GBase 8s 实现的连接池	关闭至数据库的连接，并重新打开它，以关闭与连接相关联的任何语句，并将连接重置至它的原始状态，然后，将	将连接返回至原始状态，并保持所有打开的语句，但关闭所有结果集 不推荐在此情况下调用此方法

连接池状态	调用 Connection.close() 方法的效果	调用 scrubConnection() 方法的效果
	Connection 对象返回至连接池，当新的应用程序连接请求时，该连接池可用。	
以应用程序服务器实现的连接池	由连接池实现来定义	<p>将连接返回至原始状态，并保持打开的语句，但关闭结果集</p> <p>如果您正在使用带有连接的语句池的 JDBC 3.0 特性，则此功能可能是有用的。当应用程序调用 Connection.close() 方法时，在将对象返回至连接池之前，应用程序服务器连接池管理器可调用入池的连接对象的scrubConnection()。</p>

重要： 当调用 scrubConnection() 方法时，应用程序应正在使用仅限于服务器的连接。

4 执行数据库操作

本主题主要描述如何使用 GBase 8s JDBC Driver 来执行对 GBase 8s 数据库的操作。

4.1 查询数据库

对于将查询发送至数据库以及检索结果，GBase 8s JDBC Driver 遵守 JDBC API 规范。驱动程序支持 Statement、PreparedStatement、CallableStatement、ResultSet 和 ResultSetMetaData 接口的大多数方法。

4.1.1 将查询发送至 GBase 8s 数据库的示例

下列来自 SimpleSelect.java 程序的示例展示如何使用 PreparedStatement 接口，来执行有一个输入参数的 SELECT 语句：

```
try
{
```



```
PreparedStatement pstmt = conn.prepareStatement("Select *
from x "
+ "where a = ?;");
pstmt.setInt(1, 11);
ResultSet r = pstmt.executeQuery();
while(r.next())
{
    short i = r.getShort(1);
    System.out.println("Select: column a = " + i);
}
r.close();
pstmt.close();
}
catch (SQLException e)
{
    System.out.println("ERROR: Fetch statement failed: " +
    e.getMessage());
}
```

该程序首先使用 `Connection.prepareStatement()` 方法来以它的单个输入参数准备 `SELECT` 语句。然后，它通过使用 `PreparedStatement.setInt()` 方法来将值赋予该参数，并以 `PreparedStatement.executeQuery()` 方法来执行该查询。

该程序返回 **ResultSet** 对象中的结果行，该程序以 `ResultSet.next()` 方法通过其重复执行。该程序以 `ResultSet.getShort()` 方法来检索单独的列值，因为选中列的数据类型为 `SMALLINT`。

最后，以恰当的 `close()` 方法来同时显式地关闭 **ResultSet** 和 **PreparedStatement** 对象。

要获取关于哪些 `getXXX()` 方法检索单独的列值的更多信息，请参阅 `ResultSet.getXXX()` 方法的数据类型映射。

4.1.2 结果集

`Statement.execute()` 方法的 GBase 8s JDBC Driver 实现返回单个 **ResultSet** 对象。因为服务器不支持多个 **ResultSet** 对象，因此，此实现与 JDBC API 规范不同，JDBC API 规范规定 `Statement.execute()` 方法可返回多个 **ResultSet** 对象。

GBase 8s JDBC Driver 不支持返回多个结果集。

多行的可滚动结果集

“可滚动结果集”一次从服务器访存一行。对“可滚动结果集”的性能提升允许一次访存多行。在下列示例中，期望得到行 `m` 至行 `n`，将这些行访存至 **ResultSet** 内。只要仅访问

包括在 m 与 n 之间的行, 就不会发生进一步访存。在此示例中, 期望得到行 50 至行 100, 且 ResultSet 为 SCROLL_INSENSITIVE:

```
rs.setFetchSize(51);  
rs.absolute(49); // one row will be fetched  
rs.next() // rs will contain 51 rows
```

GBase 8s 仅向前访存且仅访存一行, 除非使用 DIR_NEXT 访存来访问行。对于 DIR_NEXT 操作, 服务器发送行, 直到填满访存缓冲区为止, 或直到发送最后一行为止。仅 ResultSet.next() 可生成 DIR_NEXT 操作。

此性能提升不更改 FORWARD_ONLY ResultSet 的行为。不更改访存缓冲区大小的计算。

对于 SCROLL_INSENSITIVE ResultSet, 由访存大小和行大小来确定访存缓冲区的大小。可使用 Statement.setFetchSize() 和 ResultSet.setFetchSize() 来设置访存大小。如果访存大小为零, 则使用缺省的访存缓冲区大小。将访存缓冲区大小限定为 32 K。

某些 ResultSet methods 需要关于由查询生成的行数的信息。这些方法可能导致访存一行来获取信息, 然后重新访存当前行。这些方法是 isBeforeFirst()、isLast() 和 absolute(-row)。

此外, setMaxRows() 可为 SCROLL_INSENSITIVE ResultsSet 更改访存缓冲区大小。由于需要附加的服务器支持来确保高效地使用 setMaxRows(), 在此时不推荐使用 ResultSet.setMaxRows()。

4.1.3 释放资源

当已结束处理 SQL 语句的结果时, 通过在 Java™ 程序中调用恰当的 close() 方法, 来关闭 Statement、PreparedStatement 和 CallableStatement 对象。此关闭立即释放已分配来执行 SQL 语句的那些资源。虽然 ResultSet.close() 方法关闭 ResultSet 对象, 但它不释放分配给 Statement、PreparedStatement 或 CallableStatement 对象的资源。

当已结束处理 SQL 语句的结果时, 调用 ResultSet.close() 和 Statement.close() 方法, 来指示您以该语句或结果集处理的 GBase 8s JDBC Driver, 这是一种好的做法。当您这么做时, 程序释放数据库服务器上的所有资源。然而, 不需要特意地调

用 ResultSet.close() 和 Statement.close(), 只要调用负责释放这些资源的 Connection.close() 即可。

4.1.4 跨线程执行

不可跨线程地并发访问同一 Statement 或 ResultSet 实例。然而, 您可在多个线程之间共享一个 Connection 对象。

例如，如果一个线程在 `Statement` 对象上执行 `Statement.executeQuery()` 方法，另一个线程在同一 `Statement` 对象上执行 `Statement.executeUpdate()` 方法，则难以预料两个方法的结果，且依赖于最后执行了哪个方法。

类似地，如果一个线程执行方法 `ResultSet.next()`，另一线程在同一 `ResultSet` 对象上执行同一方法，则难以预料两个方法的结果，且依赖于最后执行了哪个方法。

4.1.5 滚动游标

GBase 8s JDBC Driver 的滚动游标特性遵循 JDBC 3.0 规范，有这些例外：

滚动灵敏性

滚动游标的 GBase 8s 数据库服务器实现将访存的行置于临时表中。如果另一进程更改原始表中的一行（假定未锁定该行），且再次访存该行，则这些更改对于客户机是不可见的。

此行为类似于 JDBC 3.0 规范中的 `SCROLL_INSENSITIVE` 描述。GBase 8s JDBC Driver 不支持 `SCROLL_SENSITIVE` 游标。要看到更新了的行，客户机应用程序必须关闭并重新打开该游标。

客户机侧滚动

JDBC 规范表明，可在客户机侧结果集上发生滚动。GBase 8s JDBC Driver 仅支持结果集的滚动达到数据库服务器支持滚动的程度。

结果集可更新性

对于产生可更新的结果集的 SQL 查询，JDBC 3.0 API 不提供严格的规范。通常，满足下列条件的查询可产生可更新的结果集：

- 该查询仅引用数据库中的单个表。
- 该查询不包含任何 `JOIN` 操作。
- 该查询选择它引用的表的主键。
- 选择列表中的每个值表达式都必须由列规范组成，且没有列规范可出现多次。
- 表表达式的 `WHERE` 子句不可包括子查询。

GBase 8s JDBC Driver 放松对主键的要求，因为该驱动程序执行下列操作：

1. 驱动程序查找称为 `ROWID` 的列。
2. 驱动程序在表中查找 `SERIAL`、`SERIAL8` 或 `BIGSERIAL` 列。
3. 驱动程序在系统目录中查找表主键。

如果这些都不提供，则驱动程序返回错误。

当您删除结果集中一行时，会影响 `ResultSet.absolute()` 方法，因为在删除之后行的位置更改了。

当查询包含 **SERIAL** 列，且在多行中复制该数据时，则 `updateRow()` 或 `deleteRow()` 的执行会影响包含该数据的所有行。

`ScrollCursor.java` 示例文件展示如何检索带有滚动游标的结果集。要了解如何使用可更新的可滚动游标的示例，请参阅 `UpdateCursor1.java`、`UpdateCursor2.java` 和 `UpdateCursor3.java` 文件。

4.1.6 保持游标

当使用事务日志记录时，当事务结束时，GBase 8s 通常关闭所有游标并释放所有锁。在多用户环境中，这种行为并不总是可取的。

GBase 8s JDBC Driver 已经以 GBase 8s 扩展实现了可保持的游标支持。GBase 8s 数据库服务器支持在游标的声明中添加关键字 **WITH HOLD**。这样的游标称为保持游标，且在事务结束时不关闭。

为符合 JDBC 3.0 规范，GBase 8s JDBC Driver 将方法添加至 JDBC 接口，以支持可保持的游标。

要获取关于保持游标的更多信息，请参阅《GBase 8s SQL 指南：语法》。

4.2 更新数据库

可发出批量更新语句，或执行批量插入来更新数据库。

4.2.1 执行批量更新

批量更新特性类似于多条 GBase 8s SQL **PREPARE** 语句。可如下列示例中那样发出批量更新语句：

```
PREPARE stmt FROM "insert into tab values (1);
                    insert into tab values (2);
                    update table tab set col = 3 where col = 2";
```

GBase 8s JDBC Driver 中的批量更新特性遵守 JDBC 3.0 规范，有这些例外：

- SQL 语句
- 从 `Statement.executeBatch()` 返回值

SQL 语句和批量更新

不可将下列命令放入多语句 **PREPARE** 语句内：

- **SELECT**（**SELECT INTO TEMP** 除外）语句

- DATABASE 语句
- CONNECTION 语句

要了解详细信息，请参阅《GBase 8s SQL 指南：语法》。

从 **Statement.executeBatch()** 方法返回值

以下列方式，返回值不同于 JDBC 3.0 规范：

- 如果将 IFX_BATCHUPDATE_PER_SPEC 环境变量设置为 0，则仅返回批量中执行的第一个语句的更新数。如果将 IFX_BATCHUPDATE_PER_SPEC 环境变量设置为 1（缺省值），则返回值等于由 Statement.executeBatch() 执行的所有 SQL 语句影响的行数。要获取更多信息，请参阅 随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量。
- 当在 Statement 对象中执行的批量更新中发生错误时，该语句不影响行；不执行该语句。在此情况下，调用 BatchUpdateException.getUpdateCounts() 会返回 0。
- 当在 PreparedStatement 对象中执行的批量更新中发生错误时，在数据库服务器上成功地插入了或更新了的行不恢复它们的更新前状态。然而，不是始终提交这些语句；它们仍处于基本的 autocommit 模式。

BatchUpdate.java 示例文件展示如何将批量更新发送至数据库服务器。

4.2.2 执行批量插入

批量插入是对 JDBC 3.0 批量更新特性的 GBase 8s 扩展。批量插入特性以多个值设置，来提升多次执行的单条 INSERT 语句的性能。要启用此特性，请将 IFX_USEPUT 环境变量设置为 1。（缺省值为 0。）

对于在同一 PreparedStatement 实例中传递的多条语句，或对于 INSERT 之外的语句，此特性不起作用。如果启用此特性，且在由不带参数的语句后跟的 INSERT 语句中传递，则忽略不带参数的语句。

对于除 opaque 类型或复合类型之外的所有数据类型，批量插入特性要求客户机转换 Java™ 类型，以便与服务器上的目标列类型相匹配。

在安装 JDBC 驱动程序的 demo 目录中安装 BulkInsert.java 示例，其展示如何执行批量插入。

4.3 参数、转义语法和不受支持的方法

本部分包含下列信息：

- 如何使用 OUT 参数

- 如何在 CallableStatement 中使用命名了的参数
- 支持 DESCRIBE INPUT 语句
- 如何使用转义语法来由 JDBC 翻译为 GBase 8s

它还罗列不受支持的方法，以及行为不同于标准的方法。

4.3.1 CallableStatement OUT 参数

CallableStatement 方法以 C 函数和 Java™ 用户定义的例程（UDR）来处理 OUT 参数。两个 registerOutParameter() 方法将 OUT 参数的数据类型指定给驱动程序。一系列 getXXX() 方法检索 OUT 参数。

JDBC CallableStatement 接口为检索 OUT 参数提供方法。

随同 GBase 8s，对于 BINARY OUT 参数，OUT 参数例程使得 JDBC 客户机可用有效的 blob 描述符和数据。使用 GBase 8s JDBC Driver Version 3.0 及后来的版本中的接收方法，可使用服务器提供的这些 OUT 参数描述符和数据。

在 GBase 8s 与 JDBC 之间交换描述符和数据，这与现有的机制相一致，为 JDBC 的结果集方法交换数据，诸如通过 SQLI 协议方法来传递 blob 描述符和数据。（SPL UDR 是支持 BINARY OUT 参数的唯一 UDR 类型。）

要了解背景信息，请参阅下列资料：

- 对于 GBase 8s 数据库中的使用，GBase 8s 用户定义的例程和数据类型开发者指南 提供关于 opaque 类型和用户定义的例程（UDR）的介绍性和背景信息。
- 对于在数据库服务器中的使用，J/Foundation 开发者指南 描述如何编写 Java UDR。
- GBase 8s SQL 指南：教程 描述如何编写存储过程语言（SPL）例程。
- GBase 8s DataBlade API 程序员指南 描述如何编写外部 C 例程。

GBase 8s 数据库服务器将 OUT 参数返回至 GBase 8s JDBC Driver。GBase 8s 支持多 OUT 参数。

要获取如何使用 OUT 参数的示例，请参阅 CallOut1.java、CallOut2.java、CallOut3.java 和 CallOut4.java 示例程序，其位于安装 GBase 8s JDBC Driver 的 demo 目录的 basic 子目录中。

服务器和驱动程序约束和限制

服务器约束

此主题描述 GBase 8s 服务器的约束。它还描述对 JDBC 驱动程序及其约束进行的改善。

不可指定 INOUT 参数。

要获取更多关于 UDR 的信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南 和 J/Foundation 开发者指南。

驱动程序改善

CallableStatement 对象为所有数据库服务器提供了一种以标准方式调用和执行 UDR 的方式。将执行这些 UDR 的结果作为结果集或作为 OUT 参数返回。

下列为创建用户定义的函数 myudr 的程序，带有两个 OUT 参数和一个 IN 参数，然后，执行 myudr() 函数。该示例要求对多 OUT 参数的服务器侧支持。要获取关于 UDR 的更多信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南 和 J/Foundation 开发者指南。

```
import java.sql.*;

public class myudr {

    public myudr() {

    }

    public static void main(String args[]) {
        Connection myConn = null;
        try {
            Class.forName("com.gbasedbt.jdbc.Driver");
            myConn = DriverManager.getConnection(
                "jdbc:gbasedbt-sqli:MYSYSTEM:18551/testDB:"
                +"GBASEDBTSERVER=patriot1;user=USERID;"
                +"password=MYPASSWORD");
        }
        catch (ClassNotFoundException e) {
            System.out.println(
                "problem with loading GBase8sDriver\n" + e.getMessage());
        }
        catch (SQLException e) {
            System.out.println(
                "problem with connecting to db\n" + e.getMessage());
        }
        try {
            Statement stmt = myConn.createStatement();
            stmt.execute("DROP FUNCTION myudr");
        }
        catch (SQLException e){
```

```
}  
try  
{  
    Statement stmt = myConn.createStatement();  
  
    stmt.execute(  
        "CREATE FUNCTION myudr(OUT arg1 int, arg2 int, OUT arg3 int)"  
        +" RETURNS boolean; LET arg1 = arg2; LET arg3 = arg2 * 2;"  
        +"RETURN 't'; END FUNCTION;");  
    }  
    catch (SQLException e) {  
        System.out.println(  
            "problem with creating function\n" + e.getMessage());  
    }  
  
    Connection conn = myConn;  
  
    try  
    {  
        String command = "{? = call myudr(?, ?, ?)}";  
        CallableStatement cstmt = conn.prepareCall (command);  
  
        // Register arg1 OUT parameter  
        cstmt.registerOutParameter(1, Types.INTEGER);  
  
        // Pass in value for IN parameter  
        cstmt.setInt(2, 4);  
  
        // Register arg3 OUT parameter  
        cstmt.registerOutParameter(3, Types.INTEGER);  
  
        // Execute myudr  
        ResultSet rs = cstmt.executeQuery();  
  
        // executeQuery returns values via a resultSet  
        while (rs.next())  
        {  
            // get value returned by myudr
```



```
boolean b = rs.getBoolean(1);
System.out.println("return value from myudr = " + b);
}

// Retrieve OUT parameters from myudr
int i = cstmt.getInt(1);
System.out.println("arg1 OUT parameter value = " + i);

int k = cstmt.getInt(3);
System.out.println("arg3 OUT parameter value = " + k);

rs.close();
cstmt.close();
conn.close();
}
catch (SQLException e)
{
    System.out.println("SQLException: " + e.getMessage());
    System.out.println("ErrorCode: " + e.getErrorCode());
    e.printStackTrace();
}
}
}
- - -

.../j2sdk1.4.0/bin/java ... myudr
return value from myudr = true
arg1 OUT parameter value = 4
arg3 OUT parameter value = 8
```

驱动程序约束和限制

GBase 8s JDBC Driver 有下列关于 OUT 参数的要求和限制:

- CallableStatement.getMetaData() 方法返回 NULL, 直到已执行了 executeQuery() 方法为止。在已调用了 executeQuery() 之后, ResultSetMetaData 对象仅包含返回值的信息, 不包含 OUT 参数。
- 必须通过使用 setXXX() 方法来指定所有 IN 参数。在 SQL 语句中不可使用文字。例如, 下列语句产生不可靠的结果:

```
CallableStatement cstmt = myConn.prepareCall("{callmyFunction(25, ?)}");
```

相反, 请使用未指定文字参数的语句:

```
CallableStatement cstmt = myConn.prepareCall("{callmyFunction(?, ?)}");
```

为两个参数调用 setXXX() 方法。

- 请不要关闭 CallableStatement.executeQuery() 方法返回的 ResultSet，直到您已通过使用 getXXX() 方法检索了 OUT 参数为止。
- 在 SQL 语句中，不可将 OUT 参数强制转型为不同类型。例如，忽略下列强制转型：

```
CallableStatement cstmt = myConn.prepareCall("{callfoo(?:?:lvarchar, ?)}");
```

- setMaxRows() 和 registerOutParameter() 方法同时取 java.sql.Types 值作为参数。有一些从 java.sql.Types 值到 GBase 8s 类型的一对多映射。

此外，有些 GBase 8s 类型不映射到 java.sql.Types 值。

对 setMaxRows() 和 registerOutParameter() 的扩展修复了这些问题。请参阅IN 和 OUT 参数类型映射。

这些约束适用于处理 C、SPL 或 Java™ UDR 的 JDBC 应用程序。

IN 和 OUT 参数类型映射

如果驱动程序找不到相匹配的 GBase 8s 类型，或发现映射不明确（多个相匹配的 GBase 8s 类型），则由 registerOutParameter(int, int)、registerOutParameter(int, int, int) 或 setNull(int, int) 方法抛出异常。下表展示映射 CallableStatement 接口使用。星号(*) 指示映射不明确。

java. sql. Types	com. gbasedbt. lang. IfxTypes
Array*	IFX_TYPE_LIST IFX_TYPE_MULTISSET IFX_TYPE_SET
Bigint	IFX_TYPE_INT8
Binary	IFX_TYPE_BYTE
Bit	不受支持
Blob	IFX_TYPE_BLOB
Char	IFX_TYPE_CHAR (n)
Clob	IFX_TYPE_CLOB
Date	IFX_TYPE_DATE
Decimal	IFX_TYPE_DECIMAL
Distinct*	依赖于基础类型
Double	IFX_TYPE_FLOAT

java.sql.Types	com.gbasedbt.lang.IfzTypes
Float	IFX_TYPE_FLOAT1
Integer	IFX_TYPE_INT
Java_Object*	IFX_TYPE_UDTVAR IFX_TYPE_UDTFIX
Long	IFX_TYPE_BIGINT IFX_TYPE_BIGSERIAL
Longvarbinary*	IFX_TYPE_BYTE IFX_TYPE_BLOB
Longvarchar*	IFX_TYPE_TEXT IFX_TYPE_CLOB IFX_TYPE_LVARCHAR
Null	不受支持
Numeric	IFX_TYPE_DECMIAL
Other	不受支持
Real	IFX_TYPE_SMFLOAT
Ref	不受支持
Smallint	IFX_TYPE_SMINT
Struct	IFX_TYPE_ROW
Time	IFX_TYPE_DTIME (hour to second)
Timestamp	IFX_TYPE_DTIME (year to fraction(5))
Tinyint	IFX_TYPE_SMINT
Varbinary	IFX_TYPE_BYTE
Varchar	IFX_TYPE_VCHAR (n)
Nothing*	IFX_TYPE_BOOL

1 此映射符合 JDBC。为了与较早版本相兼容,通过将 IFX_SET_FLOAT_AS_SMFLOAT 连接属性设置为 1, 可将 JDBC FLOAT 数据类型映射至GBase 8s SMALLFLOAT 数据类型。

要避免映射不明确，请使用下列在 `IfmxCallableStatement` 接口中定义的对 `CallableStatement` 的扩展：

```
public void IfxRegisterOutParameter(int parameterIndex,
                                     int ifxType) throws SQLException;

public void IfxRegisterOutParameter(int parameterIndex,
                                     int ifxType, String name) throws SQLException;

public void IfxRegisterOutParameter(int parameterIndex,
                                     int ifxType, int scale) throws SQLException;

public void IfxSetNull(int i, int ifxType) throws SQLException;

public void IfxSetNull(int i, int ifxType, String name) throws
    SQLException;
```

在 `IfxTypes` 类中罗列 `ifxType` 参数的可能的值。

JDBC 客户机可用 GBase 8s 有效的 BLOB 描述符和数据，来支持 SPL UDR 的二进制 OUT 参数。

GBase 8s JDBC Driver Version 3.0 或后来版本可接收服务器提供的 OUT 参数描述符和数据，并在 Java™ 应用程序中使用它。

对于任何通过方法 `getParameterType (ParameterMetaData)` 检索的任何 JDBC 二进制类型（`BINARY`、`VARBINARY`、`LONGVARBINARY`），单个正确的返回值为 -4，其与 `java.sql.Type.LONGVARBINARY` 数据类型相关联。这反映了一个事实，就是将所有 JDBC 二进制类型映射至同一 GBase 8s SQL 数据类型 `BYTE`。

4.3.2 CallableStatement 中命名了的参数

`CallableStatement` 提供从 Java™ 程序调用服务器上的存储过程的方式。可在 `CallableStatement` 中使用命名了的参数来按名称而不是按次序位置标识参数。在 JDBC 3.0 规范中引入了此改善。如果过程是唯一的，则可省略有缺省值的参数，可以任何顺序输入参数。对于调用有许多参数且某些参数有缺省值的存储过程，命名了的参数特别有用。

JDBC 驱动程序忽略参数名称的大小写。如果对于所有参数，存储过程都没有名称，则服务器为丢失的名称传递一个空字符串。

对于 `CallableStatement` 中命名了的参数的要求和约束

对于 `CallableStatement` 中命名了的参数，GBase 8s JDBC Driver 有下列要求和约束：

- 在例程的单个调用之内，必须通过名称或通过次序格式来为 `CallableStatement` 指定参数。例如，如果您为一个参数命名参数，则必须为所有参数使用参数名称。
- 对于远程 `CallableStatement`，不支持命名了的参数。
- 在 JDK Version 1.4.x 或后来版本上，支持命名了的参数。
- 对于调用存储过程，对命名了的参数的支持以现有限制为准。

在 `CallableStatement` 中核实对命名了的参数的支持

JDBC 规范提供 `DatabaseMetaData.supportsNamedParameters()` 方法，来确定驱动程序和 RDMS 是否支持 `CallableStatement` 中命名了的参数。例如：

```
Connection myConn = ... // connection to the RDBMS for Database
...
DatabaseMetaData dbmd = myConn.getMetaData();
if (dbmd.supportsNamedParameters() == true)
{
    System.out.println("NAMED PARAMETERS FOR CALLABLE"
+ "STATEMENTS IS SUPPORTED");
    ...
}
```

如果支持命名了的参数，则系统返回 `true`。

检索存储过程的参数名称

要检索存储过程的参数名称，请使用 JDBC 规范定义的 `DatabaseMetaData` 方法，如下列示例所示。

```
Connection myConn = ... // connection to the RDBMS for Database
...
DatabaseMetaData dbmd = myConn.getMetaData();
ResultSet rs = dbmd.getProcedureColumns(
    "myDB", schemaPattern, procedureNamePattern, columnNamePattern);
rs.next() {
    String parameterName = rs.getString(4);
    - - - or - - -
    String parameterName = rs.getString("COLUMN_NAME");
    - - -
    System.out.println("Column Name: " + parameterName);
}
```

显示与 `getProcedureColumns()` 方法的参数相匹配的所有列。

参数名称不是 `ParameterMetaData` 接口的一部分，不可从 `ParameterMetaData` 对象检索。

当您使用 `getProcedureColumns()` 方法时，该查询从 `sysprocedures` 系统目录表检索由 `gbasedbt` 拥有的所有过程（包括系统产生的例程）。要防止错误，请核实：已在服务器上以正确的许可配置了您正在使用的存储过程。

对于 `getProcedureColumns()` 方法，要了解在 JDBC API 行为中的重要差异，请参阅 不支持的方法和行为不同的方法。

命名了的参数和唯一的存储过程

唯一的存储过程有唯一的名称和唯一的参数编号。当 `CallableStatement` 中的参数数等于或小于存储过程中的参数数时，唯一的存储过程支持命名了的参数。

命名了的参数数等于参数数的示例

下列存储过程有五个参数

```
create procedure createProductDef(productname  varchar(64),
    productdesc  varchar(64),
    listprice    float,
    minprice     float,
    out prod_id   float);
...
let prod_id = <value for prod_id>;
end procedure;
```

下列带有五个参数的 Java™ 代码对应于该存储过程。JDBC 调用的圆括号内的问号 (?) 引用这些参数。（在此情况下，是五个参数的五个参数）。设置或注册所有参数。通过使用格式 `cstmt.setString("arg", name)` 来命名这些参数，在此，`arg` 是相应的存储过程中的参数名称。无需按照与存储过程中参数顺序相同的顺序来命名参数。

```
String sqlCall = "{call CreateProductDef(?,?,?,?,?)}";
CallableStatement cstmt = conn.prepareCall(sqlCall);

cstmt.setString("productname", name);    // Set Product Name.
cstmt.setString("productdesc", desc);    // Set Product Description.
cstmt.setFloat("listprice", listprice);  // Set Product ListPrice.
cstmt.setFloat("minprice", minprice);    // Set Product MinPrice.

// Register out parameter which should return the product is created.

cstmt.registerOutParameter("prod_id", Types.FLOAT);

// Execute the call.
cstmt.execute();
```

```
// Get the value of the id from the OUT parameter: prod_id
float id = cstmt.getFloat("prod_id");
```

Java 代码和存储过程展示下列事件的进程:

1. 准备对存储过程的调用。
2. 参数名称指示哪些参数对应于哪个参数值或类型。
3. 为输入参数设置值，并注册输出参数的类型。
4. 以输入参数作为参数来执行存储过程。
5. 存储过程返回参数值作为输出参数，并检索输出参数的值。

命名了的参数数小于参数数的示例

如果 `CallableStatement` 中的参数数小于存储过程中的参数数，则剩余的参数必须有缺省值。无需为有缺省值的参数设置值，因为服务器自动地使用缺省值。然而，您必须指出没有缺省值的参数，或在 `CallableStatement` 中以问号 (?) 覆盖缺省值。

例如，如果存储过程有 10 个参数，其中 4 个没有缺省值，6 个有缺省值，则在 `CallableStatement` 中必须有至少 4 个问号。或者，可使用 5、6，或最多 10 个问号。

如果以多于无缺省值但少于存储过程参数数的参数准备 `CallableStatement`，则它必须为无缺省值参数设置值。剩余的参数可为任何其他参数，且可随同每一执行来更改它们。

在下列唯一的存储过程中，参数 `listprice` 和 `minprice` 有缺省值:

```
create procedure createProductDef(productname  varchar(64),
    productdesc  varchar(64),
    listprice    float default 100.00,
    minprice     float default 90.00,
    out prod_id   float);
...
let prod_id = <value for prod_id>;
end procedure;
```

下列 Java™ 代码以少于存储过程中参数的参数调用存储过程（五个参数的四个参数）。由于 `listprice` 有缺省值，因此，可从 `CallableStatement` 省略它。

```
String sqlCall = "{call CreateProductDef(?,?,?,?)}";
// 4 params for 5 args
CallableStatement cstmt = conn.prepareCall(sqlCall);

cstmt.setString("productname", name); // Set Product Name.
cstmt.setString("productdesc", desc); // Set Product Description.
```

```
cstmt.setFloat("minprice", minprice);    // Set Product MinPrice.

// Register out parameter which should return the product id created.

cstmt.registerOutParameter("prod_id", Types.FLOAT);

// Execute the call.
cstmt.execute();

// Get the value of the id from the OUT parameter: prod_id
float id = cstmt.getFloat("prod_id");
```

或者，对于同一存储过程，可省略 minprice 参数的参数。无需再次准备 CallableStatement。

```
cstmt.setString("productname", name); // Set Product Name.
cstmt.setString("productdesc", desc); // Set Product Description.

cstmt.setFloat("listprice", listprice); // Set Product ListPrice.

// Register out parameter which should return the product id created.

cstmt.registerOutParameter("prod_id", Types.FLOAT);

// Execute the call.
cstmt.execute();

// Get the value of the id from the OUT parameter: prod_id
float id = cstmt.getFloat("prod_id");
```

或者，您可同时省略缺省参数的参数：

```
cstmt.setString("productname", name);
cstmt.setString("productdesc", desc);
cstmt.registerOutParameter("prod_id", Types.FLOAT);
cstmt.execute();
float id = cstmt.getFloat("prod_id");
```

命名的参数和重载的存储过程

如果多个存储过程有相同的名称和相同的参数数，则该过程为重载的（也称为重载的 UDR）。

对于重载的存储过程，JDBC 驱动程序抛出 `SQLException`，因为调用不可解析至单个存储过程。要防止 `SQLException`，请通过将 `::data_type` 附加至 `data_type` 为 GBase 8s 服务器数据类型处的问号字符，在参数列表中指定命名的参数的 GBase 8s 服务器数据类型。例如 `?::varchar` 或 `?::float`。您还必须为所有参数输入命名的参数，且以与重载的存储过程的参数顺序相同的顺序输入。

例如，下列两个过程有相同的名称（`createProductDef`）和相同的参数数。在每一过程中，`prod_id` 参数的数据类型不一样。

过程 1

```
create procedure createProductDef(productname  varchar(64),
    productdesc  varchar(64),
    listprice    float default 100.00,
    minprice     float default  90.00,
    prod_id      float);
...
let prod_id = <value for prod_id>;
end procedure;
```

过程 2

```
create procedure createProductDef(productname  varchar(64),
    productdesc  varchar(64),
    listprice    float default 100.00,
    minprice     float default  90.00,
    prod_id      int);
...
let prod_id = <value for prod_id>;
end procedure;
```

如果使用下列 Java™ 代码，则由于它不可解析至唯一的一个过程，因此，它返回 `SQLException`：

```
String sqlCall = "{call CreateProductDef(?,?,?,?)}";
CallableStatement cstmt = con.prepareCall(sqlCall);
cstmt.setString("productname", name);  // Set Product Name.
```

如果您为有不同数据类型的参数指定 GBase 8s 数据类型，则 Java 代码解析至一个过程。下列 Java 代码解析至存储过程 1，因为代码为 `prod_id` 参数指定 `FLOAT` 数据类型：

```
String sqlCall = "{call CreateProductDef(?,?,?,?:float)}";
CallableStatement cstmt = con.prepareCall(sqlCall);
cstmt.setString("productname", name);  // Set Product Name
```

4. 3. 3 JDBC 支持 DESCRIBE INPUT

SQL 92 和 99 标准为动态 SQL 指定 DESCRIBE INPUT 语句。

JDBC 3.0 规范引入对应于 DESCRIBE INPUT 支持的 `ParameterMetaData` 类和方法。

GBase 8s JDBC Driver 实现 `java.sql.ParameterMetaData` 类。此接口用于描述准备好的语句中的输入参数。已实现了方法 `getParameterMetaData()` 来为特别的语句检索元数据。

`ParameterMetaData` 类和 `getParameterMetaData()` 方法是 JDBC 3.0 API 的一部分，作为 J2SDK1.4.0 中的接口包括它们。在 JDBC 3.0 规范中指定这些接口的详细信息。

GBase 8s JDBC Driver 已实现了 `ParameterMetaData` 接口的附加方法，来扩展它的功能，如下表中所示。

返回类型	方法	描述
int	<code>getParameterLength (int param)</code>	检索参数长度
int	<code>getParameterExtendedId (int param)</code>	检索参数扩展的 ID
<code>java.lang.String</code>	<code>getParameterExtendedName (int param)</code>	检索参数扩展的名称
<code>java.lang.String</code>	<code>getParameterExtendedOwnerName (int param)</code>	检索该类型的参数扩展的所有者名称
int	<code>getParameterSourceType (int param)</code>	检索参数 <code>SourceType</code>
int	<code>getParameterAlignment (int param)</code>	检索参数对齐

下列是在 GBase 8s JDBC Driver 中使用 `ParameterMetaData` 接口的一个示例：

```
...

try
{
    PreparedStatement pstmt = null;

    pstmt = myConn.prepareStatement(
        "select * from table_1 where int_col = ? "
        +"and string_col = ?");
    ParameterMetaData paramMeta = pstmt.getParameterMetaData();
    int count = paramMeta.getParameterCount();
    System.out.println("Count : "+count);

    for (int i=1; i <= count; i++)
```

```
{
    System.out.println("Parameter type name : "
+paramMeta.getParameterTypeName(i));
    System.out.println("Parameter type : "
+paramMeta.getParameterType(i));
    System.out.println("Parameter class name : "
+paramMeta.getParameterClassName(i));
    System.out.println("Parameter mode : "
+paramMeta.getParameterMode(i));
    System.out.println("Parameter precision : "
+paramMeta.getPrecision(i));
    System.out.println("Parameter scale : "
+paramMeta.getScale(i));
    System.out.println("Parameter nullable : "
+paramMeta.isNullable(i));
    System.out.println("Parameter signed : "
+paramMeta.isSigned(i));
}
...

```

4. 3. 4 转义语法

转义语法指示必须从 JDBC 格式翻译为 GBase 8s 原生格式的信息。SQL 语句的有效转义语法如下：

语句的类型	转义语法
过程	{ call procedure }
函数	{ var = call function }
日期	{ d 'yyyy-mm-dd' }
时间	{ t 'hh:mm:ss' }
时间戳（Datetime）	{ ts 'yyyy-mm-dd hh:mm:ss[.ffffff]' }
函数调用	{ fn func[(args)] }
转义字符	{ escape 'escape-char' }
外部连接	{ oj outer-join-statement }

可在 SQL 语句中放置任意此语法，如下：

```
executeUpdate("insert into tab1 values( {d '1999-01-01'} );");
```

将方括号内的一切都转换为有效的 GBase 8s SQL 语句，并返回至调用函数。

4.3.5 不支持的方法和行为不同的方法

GBase 8s JDBC Driver 支持下列 JDBC API 方法，在连接至 GBase 8s 的 Java™ 程序中不可使用：

- CallableStatement.getRef(int)
- Connection.setCatalog()
- Connection.setReadOnly()
- PreparedStatement.addBatch(String)
- PreparedStatement.setRef(int, Ref)
- PreparedStatement.setUnicodeStream(int, java.io.InputStream, int)
- ResultSet.getRef(int)
- ResultSet.getRef(String)
- ResultSet.getUnicodeStream(int)
- ResultSet.getUnicodeStream(String)
- ResultSet.refreshRow()
- ResultSet.rowDeleted()
- ResultSet.rowInserted()
- ResultSet.rowUpdated()
- ResultSet.setFetchSize()
- Statement.setMaxFieldSize()

Connection.setCatalog() 和 Connection.setReadOnly() 方法不返回错误。其他方法抛出异常：Method not Supported.

下列 JDBC API 方法的行为不同于由 JavaSoft 规范指定的行为：

- CallableStatement.execute()
返回单个结果集
- DatabaseMetaData.getProcedureColumns()

示例：

```
DBMD.getProcedureColumns(String catalog,String schemaPattern,  
String procedureNamePattern,String columnNamePattern)
```

忽略 columnNamePattern 字段；返回 NULL。

当您使用 `getProcedureColumns()` 方法时，该查询从 `sysprocedures` 系统目录表检索由 `gbasedbt` 拥有的所有过程（包括系统生成的例程）。要防止错误，请在服务器上核实，已以正确的许可配置了您正在使用的存储过程。

例如，如果使用下列语句之一：

```
getProcedureColumns("", "", "", "")
```

```
getProcedureColumns("", gbasedbt, "", "")
```

`DatabaseMetaData.getProcedureColumns()` 方法加载所有服务器 UDR 和由用户 `gbasedbt` 拥有的 UDR。如果选择不安装 J/Foundation，或在 `onconfig` 文件中未将 J/Foundation 的配置参数设置为有效值，则方法失败。此外，如果未在服务器上正确地建立任何一个 UDR，则方法失败。

要获取关于如何在 GBase 8s 服务器上建立 J/Foundation，以及如何在 GBase 8s 服务器上运行 Java UDR 的信息，请参阅 J/Foundation 开发者指南。要获取关于如何建立和运行 C UDR 的信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南。

- `DatabaseMetaData.othersUpdatesAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.othersDeletesAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.othersInsertsAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.ownUpdatesAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.ownDeletesAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.ownInsertsAreVisible()`
始终返回 FALSE
- `DatabaseMetaData.deletesAreDetected()`
始终返回 FALSE
- `DatabaseMetaData.updatesAreDetected()`
始终返回 FALSE
- `DatabaseMetaData.insertsAreDetected()`
始终返回 FALSE
- `PreparedStatement.execute()`

返回单个结果集

- `ResultSet.getFetchSize()`

始终返回 0

- `ResultSetMetaData.getCatalogName()`

始终返回包含一个空格的 `String` 对象

- `ResultSetMetaData.getTableName()`

返回 `SELECT`、`INSERT` 和 `UPDATE` 语句的表名称

带有多个表名称的 `SELECT` 和所有其他语句返回包含一个空格的 `String` 对象。

- `ResultSetMetaData.getSchemaName()`

始终返回包含一个空格的 `String` 对象

- `ResultSetMetaData.isDefinitelyWritable()`

始终返回 `TRUE`

- `ResultSetMetaData.isReadOnly()`

始终返回 `FALSE`

- `ResultSetMetaData.isWritable()`

始终返回 `TRUE`

- `Statement.execute()`

返回单个结果集

- `Connection.isReadOnly()`

仅当连接至 HAC 常见中的辅助服务器时，返回 `TRUE`（请参阅下列重要说明）

重要： GBase 8s 服务器当前不支持只读连接。对于 GBase 8s JDBC Driver Version 2.21.JC4，已更改了来自 `java.sql.Connection` 接口的 `setReadOnly()` 方法的实现，以通过调用进程来接收传给它的值。`setReadOnly()` 方法只是返回调用进程，不与 GBase 8s 数据库服务器发生任何交互。（先前版本的 JDBC 驱动程序会抛出不支持方法异常。）此更改已使得出现在 GBase 8s JDBC Driver 中的函数可以同步至 GBase DB2 JDBC 驱动程序，也取得了符合 Sun 符合性测试(CTS)的更高级别。

4.4 处理事务

在缺省情况下，所有新的 `Connection` 对象都处于 `autocommit` 模式。当打开 `autocommit` 模式时，在发送至数据库服务器的每一语句之后，都自动地执行 `COMMIT` 语句。要关闭 `autocommit` 模式，请显式地调用 `Connection.setAutoCommit(false)`。

当 autocommit 模式关闭时，当将下一语句发送至数据库服务器时，GBase 8s JDBC Driver 隐式地启动新的事务。此事务持续，直到用户发出 COMMIT 或 ROLLBACK 语句为止。如果用户通过执行 setAutoCommit(false) 已启动了事务，然后再次调用 setAutoCommit(false)，则现有的事务继续保持不变。在 Java™ 程序删除至数据库或数据库服务器的连接之前，它必须通过发出 COMMIT 或 ROLLBACK 语句来显式地终止事务。在事务期间，如果 Java 程序设置 autocommit 模式为打开，则 GBase 8s JDBC Driver 提交当前的事务，如果 JDK 是 1.4 版本和后来版本的话。否则，在打开 autocommit 之前，驱动程序回滚当前事务。

在以日志记录创建了数据库中，如果将 COMMIT 语句发送至数据库服务器，且 autocommit 模式为打开，则数据库服务器返回错误 -255: Not in transaction，因为当前未启动用户事务。不管以 Connection.commit() 方法还是直接以 SQL 语句发送了 COMMIT 语句，都发生此情况。

在以 ANSI 模式创建的数据库中，显式地将 COMMIT 语句发送至数据库服务器，会提交一个空事务。如果当前没有打开用户事务，则由于数据库服务器在执行语句之前自动地启动事务，因此，不返回错误。

对于 XAConnection 对象，缺省情况下 autocommit 模式是关闭的，当分布式事务正在发生时，必须保持关闭。事务管理器执行提交和回滚操作；因此，请避免直接执行这些操作。

对于 GBase 8s，两个 JDBC 类支持在遇到不利事件之后可将 SQL 事务回滚至保存点（而不是完全取消）：

- IfmxSavepoint（接口）
- IfxSavepoint（Savepoint 类）

通过下列标准 JDBC 方法，JDBC 应用程序可创建、删除或回滚到保存点对象：

表 1. JDBC 保存点类和方法

类	方法
IfxConnection	setSavepoint()releaseSavepoint()rollback(savepoint)
IfxSavepoint	getSavepointId()getSavepointName() 这两个方法不可互换。调用 getSavepointName() 会出错失败，除非以 setSavepoint() 方法的或 setSavepointUnique() 方法的字符串参数来声明保存点对象。类似地，如果调用命名的保存点对象的 getSavepointId()，则返回错误。

此外，setSavepointUnique() 方法可设置其标识符为唯一的命名的保存点。当唯一的保存点是活动的时，如果应用程序尝试在同一连接内重新使用它的名称，则 GBase 8s 发出异常。

下列约束适用于 JDBC 中的保存点对象：

- 在 XA 事务内，保存点无效。
- 不可使用保存点，除非当前连接设置 autocommit 模式为关闭。
- 在至不记录日志的数据库的连接中，保存点无效。
- 在触发了的活动中不可引用保存点。
- 在跨服务器的分布式查询中，其中任何参与的从属服务器都不支持保存点对象，在连接至不支持保存点的服务器之后，如果设置一保存点，则发出警告，且任何对 rollbacksavepoint 的调用都出错失败。

要获取关于在 SQL 事务中使用保存点的更多信息，请参阅《GBase 8s SQL 指南：语法》中对 SAVEPOINT、RELEASE SAVEPOINT 和 ROLLBACK WORK TO SAVEPOINT 语句的描述。

4.5 处理错误

在 Java™ 程序中使用 JDBC API SQLException 类来处理错误。在 Java 程序之外，也可使用特定于 GBase 8s 的 com.gbasedbt.jdbc.Message 类，来检索对于给定错误编号的 GBase 8s 错误文本。

4.5.1 以 SQLException 类来处理错误

每当发生来自 GBase 8s JDBC Driver 或数据库服务器的错误时，就发出 SQLException。请使用 SQLException 类的下列方法，来检索错误消息的文本、错误代码和 SQLSTATE 值：

getMessage()

返回错误

SQLException 的描述，从 java.util.Throwable 类继承此方法。

getErrorCode()

返回对应于 GBase 8s 数据库服务器或 GBase 8s JDBC Driver 错误代码的整数值

getSQLState()

返回描述 SQLSTATE 值的字符串

该字符串遵守 X/Open SQLSTATE 惯例。

所有 GBase 8s JDBC Driver 错误都有形如 -79XXX 的错误代码，诸如 -79708: Can't take null input。

要获取 GBase 8s 数据库服务器错误的列表，请参阅《GBase 8s 错误消息》。要获取 GBase 8s JDBC Driver 错误的列表，请参阅 错误消息。

来自 SimpleSelect.java 程序的下列示例展示如何使用 `SQLException` 类，来通过使用 `try-catch` 块捕获 GBase 8s JDBC Driver 或数据库服务器错误：

```
try
{
    PreparedStatement pstmt = conn.prepareStatement("Select *
from x "
+ "where a = ?;");
    pstmt.setInt(1, 11);
    ResultSet r = pstmt.executeQuery();
    while(r.next())
    {
        short i = r.getShort(1);
        System.out.println("Select: column a = " + i);
    }
    r.close();
    pstmt.close();
}
catch (SQLException e)
{
    System.out.println("ERROR: Fetch statement failed: " +
e.getMessage());
}
```

4.5.2 检索语法错误偏移量

要确定语法错误的确切位置，请使用 `getSQLStatementOffset()` 方法，来返回语法错误偏移量。

下列示例展示如何从 SQL 语句检索语法错误偏移量（在此示例中，其为 10）：

```
try {
    Statement stmt = conn.createStatement();
    String command = "select * fom tt";
    stmt.execute( command );
}
catch(Exception e)
{
    System.out.println
("Error Offset :"+((IfmxConnection conn).getSQLStatementOffset() );
    System.out.println(e.getMessage() );
}
```

```
}
```

捕获 RSAM 错误消息

RSAM 消息附属于 SQLCODE 消息。例如，如果 SQLCODE 消息表明不可创建表，则 RSAM 消息说明原因，可能是磁盘空间不足。

可使用 `SQLException.getNextException()` 方法，来捕获 RSAM 错误消息。要了解如何捕获这些消息的示例，请参阅包括在 GBase 8s JDBC Driver 中的 `ErrorHandling.java` 程序。

4.5.3 以 `com.gbasedbt.jdbc.Message` 类来处理错误

GBase 8s 提供类 `com.gbasedbt.jdbc.Message`，用于基于 GBase 8s 错误编号，来检索 GBase 8s 错误消息文本。要使用此类，请直接调用 Java™ 解释程序 `java`，将 GBase 8s 错误编号传给它，如下列示例所示：

```
java com.gbasedbt.jdbc.Message 100
```

该示例返回 GBase 8s 错误 100 的消息文本：

```
100: ISAM error: duplicate value for a record with unique key.
```

当使用 `com.gbasedbt.jdbc.Message` 类时，如果指定无符号数值，则返回一个正的错误编号。这不同于 `finderr` 实用程序，对于无符号编号，其返回负的错误编号。

4.6 访问数据库元数据

要访问关于 GBase 8s 数据库的信息，请使用 JDBC API `DatabaseMetaData` 接口。

GBase 8s JDBC Driver 实现 `DatabaseMetaData` 方法的所有 JDBC 3.0 规范。

为了符合 JDBC 3.0，已将 `DatabaseMetaData` 中的下列新方法添加在 GBase 8s JDBC Driver 2.21.JC5 和后来版本中：

- `getSuperTypes()`
- `getSuperTables()`
- `getAttributes()`
- `getResultSetHoldability()`
- `getDatabaseMajorVersion()`
- `getDatabaseMinorVersion()`
- `getJDBCMajorVersion()`
- `getJDBCMinorVersion()`
- `getSQLStateType()`

- locatorsUpdateCopy()
- supportsGetGeneratedKeys()
- supportsMultipleOpenResults()
- supportsNamedParameters()
- supportsGetGeneratedKeys()
- supportsMultipleOpenResults()

从 GBase 8s JDBC Driver 3.0 开始, 已实现了检索服务器生成键的方法。检索自动生成的键, 涉及下列活动:

1. JDBC 应用程序编程人员提供要执行的 SQL 语句。
2. 服务器执行 SQL 语句, 并返回可检索自动生成键的标示。
3. 在服务器执行 SQL 语句之前, 验证 columnNames 或 columnIndexes (如果提供了的话)。如果它们无效, 则抛出 SQLException。
4. 如果请求, 则 JDBC 驱动程序和服务端返回 resultSet 对象。如果未生成键, 则 resultSet 为空, 不包含任何行或列。
5. 用户可请求 resultSet 对象的元数据, 且 JDBC 驱动程序和服务端返回 resultSetMetaData 对象。

要获取关于检索自动生成的键的更多信息, 请参阅“JDBC 3.0 规范”13.6 部分“检索自动产生的键”。

GBase 8s JDBC Driver 使用 sysmaster 数据库来取得数据库元数据。如果您想要在 Java™ 程序中使用 DatabaseMetaData 接口, 则在 Java 程序连接到的 GBase 8s 数据库服务器中, 必须存在 sysmaster 数据库。

GBase 8s JDBC Driver 解释 JDBC API 术语 schemas, 来表示拥有表的 GBase 8s 用户的意思。DatabaseMetaData.getSchemas() 方法返回在 systables 系统目录的 owner 列中找到的所有用户。

类似地, GBase 8s JDBC Driver 解释 JDBC API 术语 catalogs, 来表示 GBase 8s 数据库的名称的意思。DatabaseMetaData.getCatalogs() 方法返回 Java 程序连接到的 GBase 8s 数据库服务器中当前存在的所有数据库的名称。

示例 DBMetaData.java 展示如何使用 DatabaseMetaData 和 ResultSetMetaData 接口, 来收集关于新过程的信息。要获取关于此示例的更多信息, 请参考 示例代码文件。

4.7 对 JDBC API 的其他 GBase 8s 扩展

本部分描述在此指南中尚未讨论的对 JDBC API 的特定于 GBase 8s 的扩展。这些扩展处理特定于 GBase 8s 数据库的信息。

在 处理错误 中，全面地描述另一 GBase 8s 扩展，com.gbasedbt.jdbc.Message 类。

4.7.1 “自动释放”特性

如果启用 GBase 8s “自动释放”特性，则当数据库服务器关闭游标时，它自动释放该游标。因此，应用程序不必发送两个单独的请求，来关闭然后再释放游标—关闭游标就足够了。

可通过在数据库 URL 中将 IFX_AUTOFREE 变量设置为 TRUE，来启用“自动释放”特性，如此示例所示：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533:GBASEDBTSERVER=myserver;  
user=rdtest;password=test;ifx_autofree=true;
```

还可使用下列方法之一：

```
public void setAutoFree (boolean flag)  
public boolean getAutoFree()
```

在 executeQuery() 方法之前，应调用 setAutoFree() 方法，但在 executeQuery() 之前或之后，都可调用 getAutoFree()。

要使用这些方法，应用程序必须从 GBase 8s 软件包 com.gbasedbt.jdbc 导入这些类，并将 Statement 类强制转型为 IfmxStatement 类，如下所示：

```
import com.gbasedbt.jdbc.*;  
  
...  
  
(IfmxStatement)stmt.setAutoFree(true);
```

4.7.2 获取驱动程序版本信息

有两种获取关于 GBase 8s JDBC Driver 的版本信息的方式：从 Java™ 程序，或从 UNIX™ 或 MS-DOS 命令提示。

要从 Java 程序取得版本信息，请：

1. 通过将下列行添加至 import 部分，将 GBase 8s 软件包 com.gbasedbt.jdbc.* 导入至 Java 程序内：

```
import com.gbasedbt.jdbc.*;
```

2. 调用静态方法 Driver.getJDBCVersion()。

此方法返回包含当前 GBase 8s JDBC Driver 的完全版本的 String 对象。

GBase 8s JDBC Driver 版本的一个示例为 2.00.JC1。

Driver.getJDBCVersion() 方法仅返回版本，不返回在驱动程序安装期间提供的序列号。

重要：对于 GBase 8s JDBC Driver 的版本 X.Y，JDBC API 方法 `Driver.getMajorVersion()` 和 `DatabaseMetaData.getDriverMajorVersion()` 始终返回值 X。类似地，方法 `Driver.getMinorVersion()` 和 `DatabaseMetaData.getDriverMinorVersion()` 始终返回值 Y。

要从命令行取得 GBase 8s JDBC Driver 的版本，请在 UNIX shell 提示或 Windows™ 命令提示处，输入下列命令：

```
java com.gbasedbt.jdbc.Version
```

该命令还返回当您安装驱动程序时提供的序列号。

4.7.3 JDBC驱动新增getObject方法

JDBC驱动新增`getObject(int, Class<T>)`接口实现。该方法可以通过判断Class类型的具体类型，覆盖了所有目前驱动已经支持的`get`方法。对于不支持的其他类型，仍然抛出`SQLException(IfxErrMsg.S_MTHNSUPP)`异常。目前支持的Class类型如下：

`String`、`BigDecimal`、`Boolean`、`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`、`byte[]`、`java.sql.Date`、`java.sql.Time`、`java.sql.Timestamp`、`java.sql.Clob`、`java.sql.Blob`、`java.sql.Array`、`java.time.LocalDateTime`、`java.time.LocalDate`、`java.time.LocalTime`

4.8 存储和检索 XML 文档

由 World Wide Web Consortium (W3C) 定义的“可扩展置标语言”(XML) 提供以文本、可编辑文件(称为 XML 文档)描述结构化数据的规则、指南和惯例。XML 仅使用定界数据条的标记，将对数据的解释留给使用它的应用程序来处理。XML 是一种以开放的、与平台无关的格式表示数据的方法。

将访问 XML 文档的当前可用的 API 称为 JAXP (“用于 XML 解析的 Java™ API”)。该 API 有下列两个子集：

- “XML 的简单 API”(SAX) 是事件驱动协议，带有编程人员提供的回调方法，当它分析文档时，XML 解析器调用这些方法。
- “文档对象模型”(DOM) 是随机访问协议，其将 XML 文档转换为内存中的对象集合，可由编程人员自主操纵它。DOM 对象有数据类型 `Document`。

JAXP 还包含 `plugability layer`，为了创建和配置 SAX 解析器并创建 DOM 对象，通过提供标准 `factory` 方法，其实现对 SAX 和 DOM 编程访问的标准化。

对 JDBC API 的 GBase 8s 扩展促进对数据库列中 XML 数据的存储和检索。在数据存储期间使用的这些方法有助于解析 XML 数据，核实存储的 XML 数据的形式良好性和有效性，并确保拒收无效的 XML 数据。在数据检索期间使用的方法有助于将 XML 数据转换为 DOM 对象和类型 `InputSource`，这是 SAX 和 DOM 方法的标准输入类型。在仍然提供关于编程人员使用哪个 JAXP 软件包的灵活性的同时，设计 GBase 8s 扩展来支持 XML 编程人员。

4.8.1 建立环境，来使用 XML 方法

本部分包含要准备系统来使用 JDBC 驱动程序 XML 方法所需要了解的信息。

设置 CLASSPATH

要使用 XML 方法，请将下列文件的路径名称添加至 CLASSPATH 设置：

- gbasedbtjdbc_xx.jar
- xerces.jar

此外，构建 gbasedbtjdbc_xx.jar 需要使用来自支持 SAX、DOM 和 JAXP 方法的 .jar 文件的代码。要使用 gbasedbtjdbc_xx.jar，必须将这些 .jar 文件添加至 CLASSPATH 设置。

JDK version 1.4 或更新的版本使用缺省的 XML 解析器，即使 xml4j 解析器在 CLASSPATH 中。要使用 SAX 解析器的 xml4j 实现，请在应用程序代码中设置下列系统属性，或使用 -D 命令行选项：

- 必须将属性 javax.xml.parsers.SAXParserFactory 设置为 org.apache.xerces.jaxp.SAXParserFactoryImpl。
- 对于“文档对象模型”，必须将属性 javax.xml.parsers.DocumentBuilderFactory 设置为 org.apache.xerces.jaxp.DocumentBuilderFactoryImpl。

要获取关于如何设置这些属性的更多信息，请参阅 指定解析器 factory。

指定解析器 factory

在缺省情况下，xml4j xerces 解析器使用未经验证的 XML 解析器。要使用替代的 SAX 解析器 factory，请从命令行运行应用程序，如下：

```
% java -Djavax.xml.parsers.SAXParserFactory=new-factory
```

如果未从命令行运行，则必须将 factory 名称括在双引号中：

```
% java -Djavax.xml.parsers.SAXParserFactory="new-factory"
```

还可在代码中设置系统属性：

```
System.setProperty("javax.xml.parsers.SAXParserFactory", "new-factory")
```

在此代码中，new-factory 是替代的解析器 factory。例如，如果您正在使用 xerces 解析器，则以 org.apache.xerces.jaxp.SAXParserFactoryImpl 来替代 new-factory。

还可能为 DOM 方法使用替代的文档 factory for DOM。请从命令行运行应用程序，如下：

```
% java -Djavax.xml.parsers.DocumentBuilderFactory=new-factory
```

如果未从命令行运行，则 factory 名称必须括在双引号中：

```
% java -Djavax.xml.parsers.DocumentBuilderFactory="new-factory"
```

还可在代码中设置系统属性：

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory", "new-factory")
```

例如，如果正在使用 xerces 解析器，则

以 `jorg.apache.xerces.jaxp.DocumentBuilderFactoryImpl` 来替代 `new-factory`。

4.8.2 插入数据

可使用本部分中的方法，来将 XML 数据插入至数据库列。

本部分中方法声明中的参数有下列含义：

- `file` 参数是一 XML 文档。可通过 URL（诸如 `http://server/file.xml` 或 `file:///path/file.xml`）或路径名称（诸如 `/tmp/file.xml` 或 `c:\\work\\file.xml`）来引用该文档。
- `handler` 参数是您提供的可选类，包含 SAX 解析器作为它正在解析的文件的回调例程。如果未指定值，或如果将 `handler` 设置为 `NULL`，则驱动程序使用回显成功或失败的空回调例程（驱动程序以 `SQLException` 的形式报告失败）。
- `validating` 参数告诉 SAX 解析器 `factory` 使用验证的解析器，而不是仅检查形式的解析器。

如果未指定 `nsa` 或 `validating`，则驱动程序使用 `xml4j` 未经验证的 XML 解析器。要更改缺省值，请参阅 指定解析器 `factory`。

- `nsa` 参数告诉 SAX 解析器 `factory`，可否使用可处理命名空间的解析器。

下列方法通过使用 SAX 解析文件，并将它转换为一字符串。然后，可使用由这些方法返回的字符串作为 `PreparedStatement.setString()` 方法的输入，来将数据插入至数据库列内。

```
public String XMLtoString(String file, String handler, boolean
validating,boolean nsa) throws SQLException
public String XMLtoString(String file, String handler) throws
SQLException
public String XMLtoString(String file) throws SQLException
```

下列方法通过使用 SAX 来解析文件，并将它转换为类 `InputStream` 的对象。然后，可使用 `InputStream` 对象作为 `PreparedStatement.setAsciiStream()`、

`PreparedStatement.setBinaryStream()` 或 `PreparedStatement.setObject()` 方法的输入，来将数据插入至数据库列内。

```
public InputStream XMLtoInputStream(String file, String handler,
boolean validating,boolean nsa) throws SQLException;
public InputStream XMLtoInputStream(String file, String handler)
throws SQLException;
```



```
public InputStream XMLtoInputStream(String file) throws  
SQLException;
```

要获取使用这些方法的示例，请参阅 插入数据示例。

如果未指定值，或如果将 handler 设置为 NULL，则驱动程序使用缺省的 GBase 8s 处理器。

重要： 驱动程序截断对列来说过长的输入数据。例如，如果将 x.xml 文件插入至 char (55) 类型的类内，而不是 char (255) 类型的列，则驱动程序插入截断的文件，且不报错（然而，驱动程序抛出 SQLWarn 异常）。当选择截断的行时，解析器抛出 SAXParseException，因为该行包含无效的 XML。

4.8.3 检索数据

可使用本部分中的方法，来转换从数据库列访问了的 XML 数据。这些方法或者帮助您将选择了的 XML 文本转换为 DOM，或者帮助您以 SAX 来解析数据。InputSource 类是 JAXP 解析方法的输入类型。

要获取关于 file、handler、nsa 和 validating 参数的信息，请参阅 插入数据。

下列方法将 String 或 InputStream 类型的对象转换为 InputSource 类型的对象。可使用 ResultSet.getString()、ResultSet.getAsciiStream()或 ResultSet.getBinaryInputStream() 方法来从数据库列检索数据，然后，将检索的数据传至 getInputSource()，用于任何 SAX 或 DOM 解析方法。（要获取示例，请参阅 检索数据示例。）

```
public InputSource getInputSource(String s) throws SQLException;  
public InputSource getInputSource(InputStream is) throws  
    SQLException;
```

下列方法将 String 或 InputStream 类型的对象转换为 Document 类型的对象：

```
public Document StringtoDOM(String s, String handler, boolean  
    validating,boolean nsa) throws SQLException  
public Document StringtoDOM(String s, String handler) throws  
    SQLException  
public Document StringtoDOM(String s) throws SQLException  
  
public Document InputStreamtoDOM(String s, String handler, boolean  
    validating,boolean nsa) throws SQLException  
  
public Document InputStreamtoDOM(String file, String handler)  
    throws SQLException  
public Document InputStreamtoDOM(String file) throws SQLException
```

要获取使用这些方法的示例，请参阅 检索数据示例。

4.8.4 插入数据示例

本部分中的示例说明将 XML 文档转换为可接受插入至 GBase 8s 数据库列内的格式。

XMLtoString() 示例

下列示例将三个 XML 文档转换为字符串，然后，使用这些字符串作为 SQL INSERT 语句中的参数值：

```
PreparedStatement p = conn.prepareStatement("insert into tab values(?,?,?)");
    p.setString(1, UtilXML.XMLtoString("/home/file1.xml"));
    p.setString(2, UtilXML.XMLtoString("http://server/file2.xml");
    p.setString(3, UtilXML.XMLtoString("file3.xml");
```

下列示例将一个 XML 文件插入至 LVARCHAR 列。在此示例中，tab1 是以 SQL 语句创建的表：

```
create table tab1 (col1 lvarchar);
```

代码为：

```
try
{
    String cmd = "insert into tab1 values (?)";
    PreparedStatement pstmt = conn.prepareStatement(cmd);
    pstmt.setString(1, UtilXML.XMLtoString("/tmp/x.xml"));
    pstmt.execute();
    pstmt.close();
}
catch (SQLException e)
{
    // Error handling
}
```

XMLtoInputStream() 示例

下列示例将一个 XML 文件插入至 text 列。在此示例中，以 SQL 语句来创建表 tab2：

```
create table tab2 (col1 text);
```

代码为：

```
try
{
    String cmd = "insert into tab2 values (?)";
```

```
PreparedStatement pstmt = conn.prepareStatement(cmd);
pstmt.setAsciiStream(1, UtilXML.XMLtoInputStream("/tmp/x.xml"),
(int)(new File("/tmp/x.xml").length()));
pstmt.execute();
pstmt.close();
}
catch (SQLException e)
{
    // Error handling
}
```

4.8.5 检索数据示例

下列示例展示从 GBase 8s 数据库列检索数据，以及将数据库转换为 XML 解析器可接受的格式。

StringtoDOM() 示例

此示例在 **xmlcol** 是包含 XML 数据的 **lvvarchar** 类型列的假设之下操作。可以下列代码访存数据并转换为 DOM:

```
ResultSet r = stmt.executeQuery("select xmlcol from table where
...");
while (r.next())
{
    Document doc= UtilXML.StringtoDOM(r.getString("xmlcol"));
    // Process 'doc'
}
```

InputStreamtoDOM() 示例

下列示例将 XML 数据从 **text** 列访存至 DOM 对象内:

```
try
{
    String sql = "select col1 from tab2";
    Statement stmt = conn.createStatement();
    ResultSet r = stmt.executeQuery(sql);
    while(r.next())
    {
        Document doc = UtilXML.InputStreamtoDOM(r.getAsciiStream(1));
    }
}
```

```
        r.close();
    }
    catch (Exception e)
    {
        // Error handling
    }
```

getInputSource() 示例

此示例检索存储在列 **xmlcol** 中的 XML 数据，并将它转换为 **InputSource** 类型的对象；然后，可以任何 SAM 或 DOM 解析方法，来使用 **InputSource** 对象 i：

```
InputSource i = UtilXML.getInputSource
    (resultSet.getString("xmlcol"));
```

此示例使用 xerces.jar 中的 JAXP API 实现，来解析 **xmlcol** 列中访问了的 XML 数据：

```
InputSource input = UtilXML.getInputSource(resultSet.getString("xmlcol"));
SAXParserFactory f = SAXParserFactory.newInstance();
SAXParser parser = f.newSAXParser();
parser.parse(input);
```

在下列示例中，**tab1** 是以 SQL 语句创建的表：

```
create table tab1 (col1 lvarchar);
```

下列示例将 XML 数据从 **LVARCHAR** 列检索至 **InputSource** 对象内，用于解析。此示例通过调用 `org.apache.xerces.parsers.SAXParser` 处的解析器，来使用 **SAX** 解析。

```
try
{
    String sql = "select col1 from tab1";
    Statement stmt = conn.createStatement();
    ResultSet r = stmt.executeQuery(sql);
    Parser p =
ParserFactory.makeParser("org.apache.xerces.parsers.SAXParser");
    while(r.next())
    {
        InputSource i = UtilXML.getInputSource(r.getString(1));
        p.parse(i);
    }
    r.close();
}
catch (SQLException e)
```

```
{  
    // Error handling  
}
```

下列示例将 XML 数据从 text 列访问存至 **InputSource** 对象内，用于解析。此示例与前一示例相同，但它使用 JAXP factory 方法，而不是 SAX 解析器来分析数据。

```
try  
{  
    String sql = "select col1 from tab2";  
    Statement stmt = conn.createStatement();  
    ResultSet r = stmt.executeQuery(sql);  
    SAXParserFactory factory = SAXParserFactory.newInstance();  
    Parser p = factory.newSAXParser();  
    while(r.next())  
    {  
        InputSource i = UtilXML.getInputSource(r.getAsciiStream(1));  
        p.parse(i);  
    }  
    r.close();  
}  
catch (Exception e)  
{  
    // Error handling  
}
```

5 操作 GBase 8s 类型

本主题主要说明特定于 GBase 8s JDBC Driver 中支持的 GBase 8s（而不是 opaque 类型）。要获取关于 opaque 类型的信息，请参阅 与不透明数据类型一起使用。

5.1 distinct 数据类型

distinct 类型可映射至底层的基础类型或映射至用户定义的 Java™ 对象。例如，distinct 类型 INT 可映射至 int 或映射至封装该数据表示的 Java 对象。此 Java 对象必须实现 java.sql.SQLData 接口。必须提供如映射数据类型中所描述的定制类型映射，来将此 Java 对象映射至对应的 SQL 类型名称。

5.1.1 插入数据示例

下列示例展示一个定义 `distinct` 类型的 SQL 语句：

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10, 2);  
CREATE TABLE distinct_tab (mymoney_col mymoney);
```

下列为映射至基础类型的示例：

```
String s = "insert into distinct_tab (mymoney_col) values (?)";  
    System.out.println(s);  
    pstmt = conn.prepareStatement(s);  
  
    ...  
    BigDecimal bigDecObj = new BigDecimal(123.45);  
    pstmt.setBigDecimal(1, bigDecObj);  
    System.out.println("setBigDecimal...ok");  
    pstmt.executeUpdate();
```

当映射至底层的类型时，GBase 8s JDBC Driver 在客户机侧执行映射，因为数据库服务器提供在底层的类型与 `distinct` 类型之间隐式的强制转型。

还可将 `distinct` 类型映射至实现 **SQLData** 接口的 Java™ 对象。下列示例展示一个定义 `distinct` 类型的 SQL 语句：

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10,2)
```

下列代码将 `distinct` 类型映射至名为 **MyMoney** 的 Java 对象：

```
import java.sql.*;  
  
import com.gbasedbt.jdbc.*;  
public class myMoney implements SQLData  
{  
    private String sql_type = "mymoney";  
    public java.math.BigDecimal value;  
    public myMoney() { }  
  
    public myMoney(java.math.BigDecimal value)  
  
        this.value = value;  
  
    public String getSQLTypeName()  
    {
```

```
return sql_type;
{

public void readSQL(SQLInput stream, String type) throws
SQLException
{
    sql_type = type;
    value = stream.readBigDecimal();
    {

public void writeSQL(SQLOutput stream) throws SQLException
{
    stream.writeBigDecimal(value);
}
// overrides Object.equals()
public boolean equals(Object b)

return value.equals(((myMoney)b).value);
}
public String toString()
{
    return "value=" + value;
}
}
...
String s - "insert into distinct_tab (mymoney_col) values (?";
pstmt = conn.prepareStatement(s);
myMoney mymoney = new myMoney();
mymoney.value = new java.math.BigDecimal(123.45);
pstmt.setObject(1, mymoney);
System.out.println("setObject(myMoney)...ok");
pstmt.executeUpdate();
```

在此情况下，请使用 setObject() 方法，而不是 setBigDecimal() 方法，来插入数据。

5.1.2 检索数据示例

可访存 `distinct` 类型作为它的底层基础类型，或作为 `Java™` 对象，如果以定义类型映射来定义该映射的话。使用前面的示例，可访存该数据作为 `Java` 对象，如下列示例所示：

```
java.util.Map customtypemap = conn.getTypeMap();
    System.out.println("getTypeMap...ok");
    if (customtypemap == null)
    {
        System.out.println("\n***ERROR: typemap is null!");
        return;
    }
    customtypemap.put("mymoney", Class.forName("myMoney"));

    ...

    String s = "select mymoney_col from distinct_tab order by 1";
    try
    {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(s);
        System.out.println("Fetching data ...");
        int curRow = 0;
        while (rs.next())
        {
            curRow++;
            myMoney mymoneyret = (myMoney)rs.getObject("mymoney_col");
        }
        System.out.println("total rows expected: " + curRow);
        stmt.close();
    }
    catch (SQLException e)
    {
        System.out.println("***ERROR: " + e.getErrorCode() + " " +
            e.getMessage());
        e.printStackTrace();
    }
```

在此情况下，请使用 `getObject()` 方法来检索数据，而不使用 `getBigDecimal()` 方法。

5.1.3 不支持的方法

对于 `distinct` 类型，不支持下列 `SQLInput` 和 `SQLOutput` 接口的方法：

- `java.sql.SQLInput`
 - `readArray()`
 - `readCharacterStream()`
 - `readRef()`
- `java.sql.SQLOutput`
 - `writeArray()`
 - `writeCharacterStream(Reader x)`
 - `writeRef(Ref x)`

5.2 BYTE 和 TEXT 数据类型

本部分描述 GBase 8s `BYTE` 和 `TEXT` 数据类型，以及如何以 `JDBC API` 来操纵这些数据类型的列。

`BYTE` 数据类型是在不可分的字节流中存储任何数据的简单大对象数据类型。此二进制数据的示例包括电子表格、数字化的语音模式以及视频片段。`TEXT` 数据类型是存储任何文本数据的简单大对象数据类型。它可同时包含单个的和多字节的字符。

任一数据类型的列都有 231 字节的理论限制，实际限制取决于磁盘容量。

要获取关于 GBase 8s `BYTE` 和 `TEXT` 数据类型的更详尽信息，请参阅《GBase 8s SQL 指南：参考》和《GBase 8s SQL 指南：语法》。

5.2.1 高速缓存大对象

每当从数据库服务器访存 `BLOB`、`CLOB`、`text` 或 `byte` 类型的对象时，都在客户机内存中高速缓存该数据。如果大对象的大小大于 `LOBCACHE` 环境变量中的值，则在临时文件中存储大对象数据。要获取关于 `LOBCACHE` 变量的更多信息，请参阅大对象的内存管理。

5.2.2 示例：插入或更新数据

要插入至 `BYTE` 或 `TEXT` 列，或更新这些列，请从源读取诸如操作系统文件这样的数据流，并将它作为 `java.io.InputStream` 对象传送至数据库。`PreparedStatement` 提供设置此 `Java™` 输入流的输入参数的方法。当执行该语句时，GBase 8s JDBC Driver 反复地调用该输入流，读取它的内容，并将这些内容作为实际的参数数据传送至数据库。

对于 BYTE 数据类型，请使用 `PreparedStatement.setBinaryStream()` 方法来设置 `InputStream` 对象的输入参数。对于 TEXT 数据类型，请使用 `PreparedStatement.setAsciiStream()` 方法。

来自 `ByteType.java` 程序的下列示例展示如何将操作系统文件 `data.dat` 的内容插入至 BYTE 数据类型的列内：

```
try
{
    stmt = conn.createStatement();
    stmt.executeUpdate("create table tab1(col1 byte)");
}
catch (SQLException e)
{
    System.out.println("Failed to create table ..." + e.getMessage());
}

try
{
    pstmt = conn.prepareStatement("insert into tab1 values (?)");
}
catch (SQLException e)
{
    System.out.println("Failed to Insert into tab: " + e.toString());
}

File file = new File("data.dat");
int fileLength = (int) file.length();
InputStream value = null;
FileInputStream fileinp = null;
int row = 0;
String str = null;
int rc = 0;
ResultSet rs = null;

System.out.println("Inserting data ...\n");

try
{
```

```
fileinp = new FileInputStream(file);
value = (InputStream)fileinp;
}
catch (Exception e) {}

try
{
    pstmt.setBinaryStream(1,value,10); //set 1st column
}
catch (SQLException e)
{
    System.out.println("Unable to set parameter");
}

set_execute();

...
public static void set_execute()
{
    try
    {
        pstmt.executeUpdate();
    }
    catch (SQLException e)
    {
        System.out.println("Failed to Insert into tab: " + e.toString());
        e.printStackTrace();
    }
}
```

该示例首先创建一个表示操作系统文件 data.dat 的 **java.io.File** 对象。然后，该示例创建 **FileInputStream** 对象，来从 **File** 类型的对象读取。将 **FileInputStream** 类型的对象强制转型为它的超类 **InputStream**，其为 **PreparedStatement.setBinaryStream()** 方法的第二个参数所需的数据类型。在已准备好的 **INSERT** 语句上执行 **setBinaryStream()** 方法，其设置输入流参数。最后，**PreparedStatement.executeUpdate()** 方法执行，其将 data.dat 操作系统文件的内容插入至 **BYTE** 类型的列内。

TextType.java 程序展示如何将数据插入至 TEXT 类型的列内。它类似于插入至 BYTE 类型的列内，只使用 setAsciiStream() 方法来设置输入参数，而不是使用 setBinaryStream()。

5.2.3 示例：选择数据

在从表选择至 ResultSet 对象内之后，您可使用 ResultSet.getBinaryStream() 方法来从 BYTE 类型的列检索二进制或 ASCII 数据流。您还可使用ResultSet.getAsciiStream() 方法来从 TEXT 类型的列检索二进制或 ASCII 数据流。两个方法都返回 InputStream 对象，可使用其来读取 chunk 中的数据。

在调用 next() 方法来检索下一行之前，必须读取当前行中返回的流中的所有数据。

来自 ByteType.java 程序的下列示例展示如何从 BYTE 类型的列选择数据，并将数据打印至标准输出设备：

```
try
{
    stmt = conn.createStatement();
    rs = stmt.executeQuery("Select * from tab1");
    while( rs.next() )
    {
        row++;
        value = rs.getBinaryStream(1);
        dispValue(value);
    }
}
catch (Exception e) { }

...

public static void dispValue(InputStream in)
{
    int size;
    byte buf;
    int count = 0;
    try
    {
        size = in.available();
        byte ary[] = new byte[size];
        buf = (byte) in.read();
```

```
while(buf!=-1)
{
    ary[count] = buf;
    count++;
    buf = (byte) in.read();
}
}
catch (Exception e)
{
    System.out.println("Error occurred while reading stream ... \n");
}
}
```

该示例首先将 `SELECT` 语句的结果放入 `ResultSet` 对象内。然后，它执行方法 `ResultSet.getBinaryStream()`，来将 `BYTE` 数据检索至 `Java™InputStream` 对象内。

示例中也包括方法 `dispValue()` 的 `Java` 代码，使用其来将该列的内容打印至标准输出设备。`dispValue()` 方法使用字节数组，且 `InputStream.read()` 方法系统地读取 `BYTE` 类型列的内容。

`TextType.java` 程序展示如何从 `TEXT` 类型列选择数据。它类似于从 `BYTE` 类型列选择，只使用 `getAsciiStream()` 方法，而不使用 `getBinaryStream()`。

5.3 SERIAL 和 SERIAL8 数据类型

通过方法 `getSerial()` 和 `getSerial8()`，GBase 8s JDBC Driver 提供对 GBase 8s `SERIAL` 和 `SERIAL8` 数据类型的支持，其为 `java.sql.Statement` 接口实现的一部分。

由于 `SERIAL` 和 `SERIAL8` 数据类型没有来自 `java.sql.Types` 类的向任何 JDBC API 数据类型的明显映射，因此，您必须将特定于 GBase 8s 的类导入至 `Java™` 程序内，以处理 `SERIAL` 和 `SERIAL8` 列。要这么做，请将下列导入行添加至 `Java` 程序：

```
import com.gbasedbt.jdbc.*;
```

在 `INSERT` 语句之后，请使用 `getSerial()` 方法，来返回自动插入至表的 `SERIAL` 列内的序列值。在 `INSERT` 语句之后，请使用 `getSerial8()` 方法，来返回自动插入至表的 `SERIAL8` 列内的序列值。如果任何下列条件为真，则这些方法返回 0：

- 最后的语句不是 `INSERT` 语句。
- 正在插入至其内的表不包含 `SERIAL` 或 `SERIAL8` 列。
- 尚未执行 `INSERT` 语句。

在 CREATE TABLE 语句之后，如果执行 getSerial() 或 getSerial8() 方法，则该方法缺省地返回 1（假定新表包括一个 SERIAL 或 SERIAL8 列）。如果该表不包含 SERIAL 或 SERIAL8 列，则该方法返回 0。如果指定新的序列起始编号，则该方法返回该编号。

如果您想要使用 getSerial() 和 getSerial8() 方法，则必须将 Statement 或 PreparedStatement 对象强制转型为 IfmxStatement，其为特定于 GBase 8s 的 Statement 接口的实现。下列示例展示如何执行该强制转型：

```
cmd = "insert into serialTable(i) values (100)";
stmt.executeUpdate(cmd);
System.out.println(cmd+"...okay");
int serialValue = ((IfmxStatement)stmt).getSerial();
System.out.println("serial value: " + serialValue);
```

如果您想要将连续的序列值插入至 SERIAL 或 SERIAL8 数据类型的列内，则请为 INSERT 语句中的 SERIAL 或 SERIAL8 列指定值 0。当将该列设置为 0 时，数据库服务器指定仅次于最高值的值。

要获取关于 GBase 8s SERIAL 和 SERIAL8 数据类型的更详尽信息，请参阅《GBase 8s SQL 指南：参考》和《GBase 8s SQL 指南：语法》。

5.4 BIGINT 和 BIGSERIAL 数据类型

BIGINT 和 BIGSERIAL 数据类型与 INT8 和 SERIAL8 数据类型有相同的值域。然而，BIGINT 和 BIGSERIAL 比 INT8 和 SERIAL8 更利于存储和计算。

BIGINT 和 BIGSERIAL 数据类型都映射至 java.sql.Types 类中的 BIGINT Java™ 类型。当从数据库检索数据时，BIGINT 和 BIGSERIAL 数据类型映射至 long Java 类型。

通过 getBigSerial()，GBase 8s JDBC Driver 提供对 GBase 8s BIGSERIAL 和 BIGINT 数据类型的支持，其为 java.sql.Statement 接口的一部分。

由于 BIGSERIAL 和 BIGINT 数据类型没有向来自 java.sql.Types 类的 JDBC API 数据类型的明显映射，因此，您必须将特定于 GBase 8s 的类导入至 Java 程序，来处理 BIGSERIAL 和 BIGINT 列。要这么做，请将下列导入行添加至 Java 程序：

```
import com.gbasedbt.jdbc.*;
```

在 INSERT 语句之后，请使用 getBigSerial() 方法来返回插入至表的 BIGSERIAL 或 BIGINT 列内的值。

如果您想要使用 getBigSerial() 方法，则必须将 Statement 或 PreparedStatement 对象强制转型为 IfmxStatement，其为特定于 GBase 8s 的 Statement 接口的实现。下列示例展示如何执行强制转型：

```
cmd = "insert into serialTable(i) values (100)";
```

```
stmt.executeUpdate(cmd);
System.out.println(cmd+"...okay");
int serialValue = ((IfmxStatement)stmt).getSerial();
System.out.println("serial value: " + serialValue);
```

这些类型是 `com.gbasedbt.lang.IfmxTypes` 类的一部分。要了解 `IfmxTypes` 常量和对应的 GBase 8s 数据类型，请参阅 `IfmxTypes` 类表。

5.5 INTERVAL 数据类型

GBase 8s INTERVAL 数据类型存储表示时间跨度的值。INTERVAL 数据类型由两个类型组成：年-月间隔和日-时间间隔。年-月间隔可表示年和月的跨度，日-时间间隔可表示日、小时、分、秒和一秒的几分之一的跨度。要获取关于 INTERVAL 数据类型以及 `qualifier`、`precision` 和 `fraction` 的定义的更多信息，请参阅下列出版物：

- GBase 8s SQL 指南：教程
- GBase 8s SQL 指南：参考
- GBase 8s SQL 指南：语法

5.5.1 Interval 类

`com.gbasedbt.lang.Interval` 类是对 JDBC 规范的特定于 GBase 8s 的扩展。`Interval` 是 INTERVAL 数据类型的基础类。`Interval` 有两个子类：`IntervalYM`（对于年-月限定符）和 `IntervalDF`（对于日-时间限定符）。使用这些子类来创建和操纵 INTERVAL 数据类型。

提示：许多 `Interval`、`IntervalYM` 和 `IntervalDF` 构造函数采用 `Connection` 对象作为参数。这将 `CLIENT_LOCALE` 环境变量的值传至 `Interval`、`IntervalYM` 或 `IntervalDF` 对象，如果抛出异常，则其允许显示本地化的错误消息。要获取更多信息，请参阅支持全球化的错误消息。

要获取本部分中关于字符串 INTERVAL 格式的信息，请参阅《GBase 8s SQL 指南：语法》。

本部分讨论您可以 INTERVAL 数据类型使用的许多方法。要获取完整的参考信息，在安装软件之后，请参阅 `doc/javadoc/*` 目录中的联机参考资料。（`doc` 目录是安装了 GBase 8s JDBC Driver 的那个目录的子目录。）

二进制定限定符的变量

可使用字符串限定符来操纵 INTERVAL 数据类型，但使用二进制定限定符会导致更快的性能。在 `Interval` 基础类中定义下列变量，并表示二进制定限定符中字段的时间单位（开始和结束代码）。要使用这些变量，请实例化 `IntervalYM` 和 `IntervalDF` 类的对象，其从 `Interval` 基础类继承这些变量。

TU_YEAR

YEAR 限定符字段的时间单位

TU_MONTH

MONTH 限定符字段的时间单位

TU_DAY

DAY 限定符字段的时间单位

TU_HOUR

HOUR 限定符字段的时间单位

TU_MINUTE

MINUTE 限定符字段的时间单位

TU_SECOND

SECOND 限定符字段的时间单位

TU_FRAC

领头的 FRACTION 限定符字段的时间单位

TU_F1

FRACTION 的第一个位置的结束字段的时间单位

TU_F2

FRACTION 的第二个位置的结束字段的时间单位

TU_F3

FRACTION 的第三个位置的结束字段的时间单位

TU_F4

FRACTION 的第四个位置的结束字段的时间单位

TU_F5

FRACTION 的第五个位置的结束字段的时间单位

Interval 方法

可使用 **Interval** 方法来抽取关于二进制定限定符的信息。要使用这些方法，请实例化 **IntervalYM** 和 **IntervalDF** 类的对象，其从 **Interval** 基础类继承这些变量。

您可执行的某些任务和可使用的方法如下：

- 抽取限定符的长度：

```
public static byte getLength(short qualifier)
```

- 从限定符抽取开始字段代码（TU_XXX 变量之一）：

```
public static byte getStartCode(short qualifier)
```

- 限定符抽取结束字段代码（TU_XXX 变量之一）：

```
public static byte getEndCode(short qualifier)
```

- 取得对应于间隔的一部分的 TU_XXX 值的字符串值（例如，getFieldName(TU_YEAR) 返回字符串 year）：

```
public static String getFieldName(byte code)
```

- 取得该间隔的完整名称作为字符串，采用限定符作为输入：

```
public static String getIxfTypeName(int type,  
                                   short qualifier)
```

- 取得 INTERVAL 数据类型的 FRACTION 部分中的数字数：

```
public static byte getScale(short qualifier)
```

- 从长度、开始代码（TU_XXX）和结束代码（TU_XXX）创建二进制限定符：

```
public static short getQualifier(byte length, byte startCode, byte endCode)  
throws SQLException
```

例如，getQualifier(4, TU_YEAR, TU_MONTH) 创建 YEAR TO MONTH 限定符的二进制表示。

5.5.2 IntervalYM 类

com.gbasedbt.lang.IntervalYM 类允许您操纵年-月间隔。

IntervalYM 构造函数

缺省的构造函数定义如下：

```
public IntervalYM() throws SQLException
```

如果抛出异常，则请使用该构造函数的第二个版本来显示本地化的错误消息：

```
public IntervalYM(Connection conn) throws SQLException
```

请使用下列构造函数来从特定的输入值创建年-月间隔：

- 两个时间戳，返回等于 Timestamp1 - Timestamp2 的 IntervalYM 值：

```
public IntervalYM(Timestamp t1, Timestamp t2) throws  
SQLException  
public IntervalYM (Timestamp t1, Timestamp t2, Connection conn) throws  
SQLException
```

第二个版本允许您支持本地化的错误消息。

- 年和月值（将大型月份值转换为年）：

```
public IntervalYM(int years, int months) throws  
SQLException
```



```
public IntervalYM(int years, int months,
    Connection conn) throws SQLException
```

第二个版本允许您支持本地化的错误消息。

- 月份值和编码的限定符：

```
public IntervalYM(int months, short qualifier, Connection conn) throws
SQLException
```

要指定限定符，可使用 `Interval` 方法中描述的 `getQualifier()` 方法。此构造函数支持本地化的错误消息。

- 字符串：

```
public IntervalYM(String string) throws SQLException
public IntervalYM(String string, Connection conn) throws
SQLException
```

第二个版本允许您支持本地化的错误消息。

- 字符串和限定符：

```
public IntervalYM(String string, short qualifier, Connection conn) throws
SQLException
```

要指定限定符，可使用 `Interval` 方法中描述的 `getQualifier()` 方法。此构造函数支持本地化的错误消息。

- 字符和限定符信息：

```
public IntervalYM(String string, int length, byte startCode, byte endCode)
throws SQLException

public IntervalYM(String string, int length, byte startCode, byte endCode,
    Connection conn) throws SQLException
```

第二个版本允许您支持本地化的错误消息。

IntervalYM 方法

下列方法允许您操纵年-月间隔。（您还可使用前面描述的 `Interval` 方法。）您可以以 `IntervalYM` 方法执行的某些任务包括下列：

- 比较两个间隔：

```
boolean equals(Object other)
boolean greaterThan(IntervalYM other)
boolean lessThan(IntervalYM other)
```

- 从其设置间隔的值：

- 字符串：

```
void fromString(String other)
void set(String string)
```

- 年和月值（将大型月值转换为年）：

```
void set(int years, int months)
```

- 两个时间戳：

```
void set(Timestamp t1, Timestamp t2)
```

- 设置间隔的限定符：

- 从长度、开始代码和结束代码：

```
void setQualifier(int length, byte startcode, byte endcode)
```

- 使用现有的限定符：

```
void setQualifier(short qualifier)
```

- 取得间隔中的月数：

```
long getMonths()
```

- 以 yyyy-mm 格式来创建间隔的字符串表示：

```
String toString()
```

字段的显示依赖于限定符。以空格替代开头的零。

5.5.3 IntervalDF 类

com.gbasedbt.lang.IntervalDF 类允许您操纵间隔。

IntervalDF 构造函数

定义的缺省构造函数如下：

```
public IntervalDF() throws SQLException
```

如果抛出异常，请使用缺省构造函数的第二个版本，来显示本地化的错误消息：

```
public IntervalDF(Connection conn) throws SQLException
```

请使用下列构造函数，来创建来自特定的输入值的间隔：

- 两个时间戳 t1 和 t2，返回等于 t1 - t2 的 IntervalDF 值：

```
public IntervalDF(Timestamp t1, Timestamp t2)
    throws SQLException
```

```
public IntervalDF(Timestamp t1, Timestamp t2, Connection conn)
    throws SQLException
```

第二个版本允许您支持本地化的错误消息。

- 秒和纳秒数（将大型秒值转换为分、小时或天）：

```
public IntervalDF(long seconds, long nanos)
    throws SQLException

public IntervalDF(long seconds, long nanos, Connection conn)
    throws SQLException
```

第二个版本允许您支持本地化的错误消息。

- 秒数、纳秒数和限定符：

```
public IntervalDF(long seconds, long nanos, short qualifier)
    throws SQLException

public IntervalDF(long seconds, long nanos, short qualifier, Connection conn)
    throws SQLException
```

要指定限定符，可使用 `Interval` 方法中描述的 `getQualifier()` 方法。第二个版本允许您支持本地化的错误消息。

- 字符串：

```
public IntervalDF(String string)
    throws SQLException

public IntervalDF(String string, Connection conn)
    throws SQLException
```

第二个版本允许您支持本地化的错误消息。

当使用这些构造函数时，将缺省的限定符设置为下列值：

领头的字段精度：2 开始代码：TU_DAY 结束代码：TU_F5

要获取关于字符串 `INTERVAL` 格式的信息，请参阅《GBase 8s SQL 指南：语法》。

- 字符串和限定符：

```
public IntervalDF(String string, short qualifier)
    throws SQLException

public IntervalDF(String string, short qualifier, Connection conn)
    throws SQLException
```

要指定限定符，可使用 `Interval` 方法中描述的 `getQualifier()` 方法。第二个版本允许您支持本地化的错误消息。

- 字符串和限定符信息：

```
public IntervalDF(String string, int length, byte startcode, byte endcode)
    throws SQLException
```

```
public IntervalDF(String string, int length, byte startcode, byte endcode,  
Connection conn) throws SQLException
```

第二个版本允许您支持本地化的错误消息。

IntervalDF 方法

下列方法允许您操纵间隔。（您还可使用 `Interval` 方法，如前描述。）您可执行的任务和可使用的方法如下：

- 比较两个间隔：

```
boolean equals(Object other)  
boolean greaterThan(IntervalDF other)  
boolean lessThan(IntervalDF other)
```

- 从其设置间隔的值：

- 字符串：

```
void fromString(String other)  
void set(String string)
```

- 秒和纳秒值（将大型秒值转换为分钟、小时或天）：

```
void set(long seconds, long nanos)
```

- 两个时间戳：

```
void set(Timestamp t1, Timestamp t2)
```

- 从长度、开始代码和结束代码设置限定符：

```
void setQualifier(int length, byte startcode, byte endcode)
```

- 取得间隔中的纳秒数：

```
long getNanoSeconds()
```

- 取得间隔中的秒数：

```
long getSeconds()
```

- 以格式 dddd hh:mm:ss.nano 创建间隔的字符串表示：

```
String toString()
```

字段显示依赖于限定符。以空格替代开头的零。

5.5.4 Interval 示例

包括在 GBase 8s JDBC Driver 中的 `Intervaldemo.java` 程序展示如何插入以及如何从两类 `INTERVAL` 数据类型选择。

5.6 集合和数组

JDBC 3.0 规范仅描述一个方法，来交换 Java™ 客户机与关系型数据库之间的集合数据：数组。

由于数组接口不包括构造函数，因此，GBase 8s JDBC Driver 包括一个扩展，允许在 `PreparedStatement.setObject()` 和 `ResultSet.getObject()` 方法中使用 `java.util.Collection` 对象。

如果您更愿意使用 `Array` 对象，则请使用

用 `PreparedStatement.setArray()` 和 `ResultSet.getArray()` 方法。`Collection` 对象更易于使用，但 `Array` 对象符合 JDBC 3.0 标准。

在缺省情况下，在访存期间，驱动程序将 `LIST` 列映射至 `java.util.ArrayList` 对象，而将 `SET` 和 `MULTISET` 列映射至 `java.util.HashSet` 对象。您可覆盖这些缺省值，但您使用的类必须实现 `java.util.Collection` 接口。

要覆盖此缺省的映射，可使用 `java.util.Collection` 接口中的其他类，诸如 `TreeSet` 类。还可创建实现 `java.util.Collection` 接口的自己的类。在任一情况下，您都必须使用 `Connection.setTypeMap()` 方法来提供定制的类型映射。

在 `INSERT` 操作期间，必须将作为 `java.util.Set` 接口实例的任何 `java.util.Collection` 对象映射至 GBase 8s `MULTISET` 数据类型。将 `java.util.List` 接口的实例映射至 GBase 8s `LIST` 数据类型。通过创建定制的类型映射，可覆盖这些缺省值。

要获取关于定制的类型映射的信息，请参阅 映射数据类型。

重要： `set` 是无序的定义。如果使用 `HashSet` 对象来选择集合数据，则 `HashSet` 对象中元素的顺序可能与插入该 `set` 时指定的顺序不同。例如，如果数据库服务器上的数据为 `set {1, 2, 3}`，则检索至 `HashSet` 对象内的数据可能为 `{3, 2, 1}`，或任何其他顺序。

下列部分中所有示例的完整版本位于您安装驱动程序处的 `complex-types` 目录中。要获取更多信息，请参阅 示例代码文件。

5.6.1 集合示例

下列为样例数据库模式：

```
create table tab ( a set(integer not null), b integer);
insert into tab values ("set{1, 2, 3}", 10);
```

下列为使用 `java.util.HashSet` 对象的访存示例：

```
java.util.HashSet set;

PreparedStatement pstmt;
ResultSet rs;

pstmt = conn.prepareStatement("select * from tab");
```

```
System.out.println("prepare ... ok");
rs = pstmt.executeQuery();
System.out.println("executeQuery ... ok");

rs.next();
set = (HashSet) rs.getObject(1);
System.out.println("getObject() ... ok");

/* The user can now use HashSet.iterator() to extract
 * each element in the collection.
 */
Iterator it = set.iterator();
Object obj;
Class cls = null;
int i = 0;
while (it.hasNext())
{
    obj = it.next();
    if (cls == null)
    {
        cls = obj.getClass();
        System.out.println("    Collection class: " + cls.getName());
    }
    System.out.println("    element[" + i + "] = " +
        obj.toString());
    i++;
}
pstmt.close();
```

在此示例的 `set = (HashSet) rs.getObject(1)` 语句中，GBase 8s JDBC Driver 取得列 1 的类型。由于它是 SET 类型，因此，实例化一HashSet 对象。接下来，将每一集合元素转换至 Java™ 对象内，并插入至集合内。

下列访存示例使用 **java.util.TreeSet** 对象：

```
java.util.TreeSet set;

PreparedStatement pstmt;
ResultSet rs;
```

```
/*
 * Fetch a SET as a TreeSet instead of the default
 * HashSet. In this example a new java.util.Map object has
 * been allocated and passed in as a parameter to getObject().
 * Connection.getTypeMap() could have been used as well.
 */
java.util.Map map = new HashMap();
map.put("set", Class.forName("java.util.TreeSet"));
System.out.println("mapping ... ok");

pstmt = conn.prepareStatement("select * from tab");
System.out.println("prepare ... ok");
rs = pstmt.executeQuery();
System.out.println("executeQuery ... ok");

rs.next();
set = (TreeSet) rs.getObject(1, map);
System.out.println("getObject(Map) ... ok");

/* The user can now use HashSet.iterator() to extract
 * each element in the collection.
 */
Iterator it = set.iterator();
Object obj;
Class cls = null;
int i = 0;
while (it.hasNext())
{
    obj = it.next();
    if (cls == null)
    {
        cls = obj.getClass();
        System.out.println("    Collection class: " + cls.getName());
    }
    System.out.println("    element[" + i + "] = " +
        obj.toString());
    i++;
}
```

```
pstmt.close();
```

在 `map.put("set", Class.forName("java.util.TreeSet"))`; 语句中, 覆盖 `set = HashSet` 的缺省映射。

在 `set = (TreeSet)rs.getObject(1, map)` 语句中, GBase 8s JDBC Driver 为列 1 取得类型, 并发现它是 SET 对象。然后, 驱动程序查找类型映射信息, 找到 `TreeSet`, 并实例化一 `TreeSet` 对象。接下来, 将每一集合元素映射至 Java 对象, 并插入至集合内。

下列示例展示插入。此示例将 `set (0, 1, 2, 3, 4)` 插入至 SET 列内:

```
java.util.HashSet set = new HashSet();

    Integer intObject;
    int i;

    /* Populate the Java collection */
    for (i=0; i < 5; i++)
    {
        intObject = new Integer(i);
        set.add(intObject);
    }
    System.out.println("populate java.util.HashSet...ok");

    PreparedStatement pstmt = conn.prepareStatement
    ("insert into tab values (?, 20)");
    System.out.println("prepare...ok");

    pstmt.setObject(1, set);
    System.out.println("setObject()...ok");
    pstmt.executeUpdate();
    System.out.println("executeUpdate()...ok");
    pstmt.close();
```

此示例中的 `pstmt.setObject(1, set)` 语句首先序列化集合的每一元素。接下来, 构造类型信息作为每一元素转换为 Java 对象。如果该集合中任何元素的类型都与第一个元素的类型不匹配, 则抛出异常。将类型信息发送至数据库服务器。

5.6.2 数组示例

下列是一个样例数据库模式:

```
CREATE TABLE tab (a set(integer not null), b integer);
INSERT INTO tab VALUES ("set{1,2,3}", 10);
```


下列示例使用 **java.sql.Array** 对象访存数据：

```
PreparedStatement pstmt = conn.prepareStatement("select a from tab");
    System.out.println("prepare ... ok");
    ResultSet rs = pstmt.executeQuery();
    System.out.println("executeQuery ... ok");

    rs.next();
    java.sql.Array array = rs.getArray(1);
    System.out.println("getArray() ... ok");
    pstmt.close();

    /*
    * The user can now materialize the data into either
    * an array or else a ResultSet. If the collection elements
    * are primitives then the array should be an array of primitives,
    * not Objects. Mapping data can be provided at this point.
    */

    Object obj = array.getArray((long) 1, 2);

    int [] intArray = (int []) obj;    // cast it to an array of ints
    int i;
    for (i=0; i < intArray.length; i++)
    {
        System.out.println("integer element = " + intArray[i]);
    }
    pstmt.close();
```

`java.sql.Array array = rs.getArray(1)` 语句实例化 **java.sql.Array** 对象。在此点不转换数据。

`Object obj = array.getArray((long)1, 2)` 语句将数据转换为整数的数组（**int** 类型，不是 **Integer** 对象）。由于未以索引和计数值调用 `getArray()` 方法，因此，仅返回数据的子集。

5.7 命名的和未命名的行

JDBC 3.0 规范引用一称为 结构化类型或 struct 的 SQL 类型，其等同于 GBase 8s 命名的行。该规范定义在 Java™ 客户机与关系型数据库之间交换结构化类型数据的方法：

- 使用 **SQLData** 接口。每个命名的行类型一个单个 Java 类，实现 **SQLData** 接口。对于命名的行中每一元素，该类有一个成员。
- 使用 **Struct** 接口。此接口为命名的行中每一元素实例化必要的 Java 对象，并构建一个 **java.util.Object** Java 对象的数组。

对于访问了命名的行，GBase 8s JDBC Driver 是实例化一个 Java 对象，还是一个 Struct 对象，依赖于是否有定制的类型映射条目，如下所示：

- 对于 `Connection.getTypeMap()` 映射中的命名的行，如果有一个条目，或如果您使用 `getObject()` 方法提供了类型映射，则实例化单个 Java 对象。
- 对于 `Connection.getTypeMap()` 映射中命名的行，如果没有条目，且如果您未使用 `getObject()` 方法提供类型映射，则实例化一个 **Struct** 对象。

始终将未命名的行访问至 Struct 对象内。

重要： 不论使用 **SQLData** 还是 **Struct** 接口，如果命名的行或未命名的行包含一 `opaque` 数据类型列，则必须有一个它的类型映射条目。如果正在使用 **Struct** 接口来访问包含 `opaque` 数据类型列的行，则您需要该 `opaque` 数据类型列的定制类型映射，但不是对于整个行。

要获取关于定制类型映射的更多信息，请参阅 映射数据类型。

5.7.1 间隔和集合支持

扩展 `java.sql.SQLOutput` 和 `java.sql.SQLInput` 方法，来支持命名的和未命名的行中的 `Collection` 和 `Interval` 对象。这些扩展包括下列方法：

- 如果数据为 `set`、`list` 或 `multiset` 数据类型，则 `com.gbasedbt.jdbc.IfmxComplexSQLInput.readObject()` 方法返回恰当的 **java.util.Collection** 对象。
- 对于 `interval` 数据类型，`com.gbasedbt.jdbc.IfmxComplexSQLInput.readInterval()` 方法是返回恰当的 **IntervalYM** 还是 **IntervalDF** 对象，依赖于限定符。
- `com.gbasedbt.jdbc.IfmxComplexSQLOutput.writeObject()` 方法接受从 **java.util.Collection** 接口，或从 **IntervalYM** 和 **IntervalDF** 对象派生的对象。

5.7.2 不支持的方法

不支持下列 `SQLInput` 方法将 ROW 列选择至实现 `SQLData` 的 Java™ 对象内：

- `readByte()`
- `readCharacterStream()`
- `readRef()`

不支持下列 `SQLOutput` 方法将实现 `SQLData` 的 Java 对象插入至 ROW 列内：

- writeByte(byte)
- writeCharacterStream(java.io.Reader x)
- writeRef(Ref x)

5.7.3 SQLData 接口

命名的行的 Java™ 类必须实现 **SQLData** 接口。对于命名的行中每一元素，该类必须有一个成员，但除了这些之外可有其他成员。这些成员可以按任何顺序，且不要公开。

对于命名的行，Java 类必须实现 writeSQL()、readSQL() 和 getSQLTypeName() 方法，如在 **SQLData** 接口中定义的那样，但可实现附加的方法。可使用 ClassGenerator 实用程序来创建该类；要获取更多信息，请参阅 ClassGenerator 实用程序。

要以命名的行来链接此 Java 类，请使用 Connection.setTypeMap() 方法或 getObject() 方法，来创建定制的类型映射。要获取关于类型映射的更多信息，请参阅 映射数据类型。

不可使用 **SQLData** 接口来访问未命名的行。

SQLData 示例

本部分中所有示例的完整版本位于安装了驱动程序处的 demo/complex-types 目录中。要获取更多信息，请参阅 示例代码文件。

下列示例包括一个实现 java.sql.SQLData 接口的 Java™ 类。

这里是样例数据库模式：

```
CREATE ROW TYPE fullname_t (first char(20), last char(20));
CREATE ROW TYPE person_t (id int, name fullname_t, age int);
CREATE TABLE teachers (person person_t, dept char (20));
INSERT INTO teachers VALUES ("row(100, row('Bill', 'Smith'), 27)", "physics");
```

这是 **fullname** Java 类：

```
import java.sql.*;

public class fullname implements SQLData
{
    public String first;
    public String last;
    private String sql_type = "fullname_t";

    public String getSQLTypeName()
    {
        return sql_type;
    }
}
```

```
public void readSQL (SQLInput stream, String type) throws
SQLException
{
    sql_type = type;
    first = stream.readString();
    last = stream.readString();
}
public void writeSQL (SQLOutput stream) throws SQLException
{
    stream.writeString(first);
    stream.writeString(last);
}
/*
 * Function not required by SQLData interface, but makes
 * it easier for displaying results.
 */
public String toString()
{
    String s = "fullname: ";
    s += "first: " + first + " last: " + last;
    return s;
}
}
```

这是 **person** Java 类:

```
import java.sql.*;

public class person implements SQLData
{
    public int id;
    public fullname name;
    public int age;
    private String sql_type = "person_t";

    public String getSQLTypeName()
    {
        return sql_type;
    }

    public void readSQL (SQLInput stream, String type) throws SQLException
```

```
{
    sql_type = type;
    id = stream.readInt();
    name = (fullname)stream.readObject();
    age = stream.readInt();
}

public void writeSQL (SQLOutput stream) throws SQLException
{
    stream.writeInt(id);
    stream.writeObject(name);
    stream.writeInt(age);
}

public String toString()
{
    String s = "person:";
    s += "id: " + id + "\n";
    s += "    name: " + name.toString() + "\n";
    s += "    age: " + age + "\n";
    return s;
}
}
```

这里是访存命名的行的示例：

```
java.util.Map map = conn.getTypeMap();
    conn.setTypeMap(map);
    map.put("fullname_t", Class.forName("fullname"));
    map.put("person_t", Class.forName("person"));

    ...
    PreparedStatement pstmt;
    ResultSet rs;
    pstmt = conn.prepareStatement("select person from teachers");
    System.out.println("prepare ...ok");

    rs = pstmt.executeQuery();
    System.out.println("executetQuery()...ok");

    while (rs.next())
```

```
{
    person who = (person) rs.getObject(1);
    System.out.println("getObject()...ok");
    System.out.println("Data fetched:");
    System.out.println("row: " + who.toString());
}
pstmt.close();
```

通过 **Connection** 对象，`conn.getTypeMap()` 方法从 **java.util.Map** 对象返回命名的行映射信息。

`map.put()` 方法注册数据库服务器上嵌套的命名的行 **fullname_t** 与 Java 类 **fullname** 之间的映射，以及数据库服务器上命名的行 **person_t** 与 Java 类 **person** 之间的映射。

`person who = (person) rs.getObject(1)` 语句将命名的行检索至 Java 对象 **who** 内。GBase 8s JDBC Driver 承认此对象 **who** 是命名的行、**distinct** 类型或 **opaque** 类型，因为数据库服务器发送的信息有扩展的名称 **person_t**。

驱动程序查找 **person_t**，并发现它是命名的行。驱动程序以键 **person_t** 调用 `map.get()` 方法，其返回 **person** 类对象。实例化类 **person** 的一个对象。

person 类中的 `readSQL()` 方法调用定义在 **SQLInput** 接口中的方法，来将 **ROW** 列中每一字段转换为 Java 对象，并将每一指定为 **person** 类中的一个成员。

下列展示一个方法，用于使用 `setObject()` 方法将 Java 对象插入至命名的行列内：

```
java.util.Map map = conn.getTypeMap();
map.put("fullname_t", Class.forName("fullname"));
map.put("person_t", Class.forName("person"));

...

PreparedStatement pstmt;
System.out.println("Populate person and fullname objects");
person who = new person();
fullname name = new fullname();
name.last = "Jones";
name.first = "Sarah";
who.id = 567;
who.name = name;
who.age = 17;

String s = "insert into teachers values (?, 'physics')";
pstmt = conn.prepareStatement (s);
System.out.println("prepared...ok");
```

```
pstmt.setObject(1, who);
System.out.println("setObject()...ok");

int rowcount = pstmt.executeUpdate();
System.out.println("executeUpdate()...ok");
pstmt.close();
```

通过 **Connection** 对象，`conn.getTypeMap()` 方法从 **java.util.Map** 对象返回命名的行映射信息。

`map.put()` 方法注册数据库服务器上嵌套的命名的行 **fullname_t** 与 Java 类 **fullname** 之间的映射，以及数据库服务器上命名的行 **person_t** 与 Java 类 **person** 之间的映射。

GBase 8s JDBC Driver 认可对象 **who** 实现 **SQLData** 接口，因此，它是命名的行、**distinct** 类型，或 **opaque** 类型。对于此对象，GBase 8s JDBC Driver 调用 `getSQLTypeName()` 方法（需要类实现 **SQLData** 接口），其返回 **person_t**。驱动程序查找 **person_t**，并发现它是命名的行。

对于类中的每一成员，**person** 类中的 `writeSQL()` 方法调用对应的 `SQLOutput.writeXXX()` 方法，将其每一都映射至命名的行 **person_t** 中的一个字段。该类中的 `writeSQL()` 方法包含对 `SQLOutput.writeObject(name)` 和 `SQLOutput.writeInt(id)` 方法的调用。序列化类 **person** 的每一成员，并写至流内。

5.7.4 Struct 接口

JDBC 资料未指定 **Struct** 对象可作为 `PreparedStatement.setObject()` 方法的参数。然而，GBase 8s JDBC Driver 可处理实现 `java.sql.Struct` 接口的 `PreparedStatement.setObject()` 或 `ResultSet.getObject()` 方法处理的任何对象。

必须使用 **Struct** 接口来访问未命名的行。

无需创建自己的类来实现 `java.sql.Struct` 接口。然而，在您可插入或更新 **ROW** 数据之前，必须执行访存来检索 **ROW** 数据和类型信息。GBase 8s JDBC Driver 自动地调用 `getSQLTypeName()` 方法，其返回命名的行的类型名称，或未命名的行的行定义。

如果创建自己的类来实现 **Struct** 接口，则您创建的类必须实现所有 `java.sql.Struct` 方法，包括 `getSQLTypeName()` 方法。您可选择 `getSQLTypeName()` 方法返回什么。

虽然必须返回未命名的行的行定义，但是，您可返回命名的行的行名称或行定义。每一个都有优点：

- **行定义**。驱动程序无需为了类型信息查询数据库服务器。此外，返回的行定义不必与命名的行定义完全匹配，因为数据库服务器提供强制转型，如果需要的话。例如，如果想要使用字符串来插入至行中的 **opaque** 类型，则这是有用的。

- **行名称。**如果用户定义的例程采用命名的行作为参数，则签名必须匹配，因此，您必须在命名的行中传递。

要获取关于用户定义的例程的更多信息，请参数下列出版物：**J/Foundation 开发者指南**（特定于 Java™ 的信息）；**GBase 8s 用户定义的例程和数据类型开发者指南** 和 **GBase 8s SQL 指南：参考**（都是关于用户定义的例程的通用信息）；**GBase 8s SQL 指南：语法**（创建和调用用户定义的例程的语法）。

重要： 如果为命名的行使用 Struct 接口，并为命名的行提供类型映射信息，则当调用 `ResultSet.getObject()` 方法时，会产生 `ClassCastException` 信息，因为 Java 不可在 `SQLData` 对象与 Struct 对象之间强制转型。

Struct 示例

本部分中所有示例的完整版本位于安装了驱动程序处的 `demo/complex-types` 目录中。要获取更多信息，请参阅 示例代码文件。

此示例访存未命名的 ROW 列。这里是样例数据库模式：

```
CREATE TABLE teachers
(
  person row(
    id int,
    name row(first char(20), last char(20)),
    age int
  ),
  dept char(20)
);

INSERT INTO teachers VALUES ("row(100, row('Bill', 'Smith'), 27)", "physics");
```

这是该示例的剩余部分：

```
PreparedStatement pstmt;
ResultSet rs;

pstmt = conn.prepareStatement("select person from teachers");
System.out.println("prepare ...ok");
rs = pstmt.executeQuery();
System.out.println("executetQuery()...ok");

rs.next();
Struct person = (Struct) rs.getObject(1);
System.out.println("getObject()...ok");
System.out.println("\nData fetched:");

Integer id;
```



```
Struct name;
Integer age;
Object[] elements;

/* Get the row description */
String personRowType = person.getSQLTypeName();
System.out.println("person row description: " + personRowType);
System.out.println("");

/* Convert each element into a Java object */
elements = person.getAttributes();

/*
 * Run through the array of objects in 'person' getting out each structure
 * field. Use the class Integer instead of int, because int is not an object.
 */
id = (Integer) elements[0];
name = (Struct) elements[1];
age = (Integer) elements[2];
System.out.println("person.id: " + id);
System.out.println("person.age: " + age);
System.out.println("");

/* Convert 'name' as well. */
/* get the row definition for 'name' */
String nameRowType = name.getSQLTypeName();
System.out.println("name row description: " + nameRowType);

/* Convert each element into a Java object */
elements = name.getAttributes();

/*
 * run through the array of objects in 'name' getting out each structure
 * field.
 */
String first = (String) elements[0];
String last = (String) elements[1];
System.out.println("name.first: " + first);
```

```
System.out.println("name.last: " + last);  
pstmt.close();
```

如果列 1 是 ROW 类型, 且没有扩展的数据类型名称 (如果它是命名的行的话), 则 `Struct person = (Struct) rs.getObject(1)` 语句实例化一个 `Struct` 对象。

`elements = person.getAttributes();` 语句执行下列活动:

- 以正确的元素数分配一个 `java.lang.Object` 对象的数组
- 将该行中每一元素转换为一 `Java™` 对象

如果该元素为 `opaque` 类型, 则必须在 `Connection` 对象中提供类型映射, 或在 `getAttributes()` 方法的调用中传递一个 `java.util.Map` 对象。

`String personrowType = person.getSQLTypeName();` 语句返回行类型信息。如果此类型为命名的行, 则该语句返回名称。由于类型不是命名的行, 因此, 该语句返回行定义: `row(int id, row(first char(20), last char(20)) name, int age)`。

然后, 该示例进入未命名的行 `name` 的后续步骤, 如同对未命名的行 `person` 一样。

下列示例使用用户创建的类 `GenericStruct`, 其实现 `java.sql.Struct` 接口。作为替代, 您可使用从 `ResultSet.getObject()` 方法返回的 `Struct` 对象, 而不是 `GenericStruct` 类。

```
import java.sql.*;  
  
import java.util.*;  
  
public class GenericStruct implements java.sql.Struct  
{  
    private Object [] attributes = null;  
    private String typeName = null;  
  
    /*  
    * Constructor  
    */  
    GenericStruct() { }  
  
    GenericStruct(String name, Object [] obj)  
    {  
        typeName = name;  
        attributes = obj;  
    }  
    public String getSQLTypeName()  
    {  
        return typeName;  
    }  
}
```

```
public Object [] getAttributes()
{
    return attributes;
}

public Object [] getAttributes(Map map) throws SQLException
{
    // this class shouldn't be used if there are elements
    // that need customized type mapping.
    return attributes;
}

public void setAttributes(Object [] objArray)
{
    attributes = objArray;
}

public void setSQLTypeName(String name)
{
    typeName = name;
}
}
```

下列 Java 程序插入一 ROW 列:

```
PreparedStatement pstmt;

ResultSet rs;
GenericStruct gs;
String rowType;

pstmt = conn.prepareStatement("insert into teachers values (?, 'Math')");
System.out.println("prepare insert...ok\n");

System.out.println("Populate name struct...");
Object[] name = new Object[2];

// populate inner row first
name[0] = new String("Jane");
name[1] = new String("Smith");

rowType = "row(first char(20), last char(20))";
gs = new GenericStruct(rowType, name);
System.out.println("Instantiate GenericStructObject...okay\n");
```

```
System.out.println("Populate person struct...");
// populate outer row next
Object[] person = new Object[3];
person[0] = new Integer(99);
person[1] = gs;
person[2] = new Integer(56);

rowType = "row(id int, " +
"name row(first char(20), last char(20)), " +
"age int)";
gs = new GenericStruct(rowType, person);
System.out.println("Instantiate GenericStructObject...okay\n");

pstmt.setObject(1, gs);
System.out.println("setObject()...okay");
pstmt.executeUpdate();
System.out.println("executeUpdate()...okay");
pstmt.close();
```

在此示例中的 `pstmt.setObject(1, gs)` 语句处，GBase 8s JDBC Driver 确定要从客户机传至数据库服务器的信息作为 ROW 列，因为 `GenericStruct` 对象是 `java.sql.Struct` 接口的一个实例。序列化数组中的每一元素，核实每一元素与由 `getSQLTypeName()` 方法定义的类型相匹配。

5. 7. 5 ClassGenerator 实用程序

对于在系统目录中定义的命名的行类型，ClassGenerator 实用程序生成 Java™ 类。该实用程序是对 JDBC 规范的 GBase 8s 扩展。

创建了的 Java 类实现 `java.sql.SQLData` 接口。对于命名的行中每一字段，该类都有成员。按照出现在数据库中命名的行类型定义中的顺序，`readSQL()`、`writeSQL()` 和 `SQLData.readSQL()` 方法读取属性。类似地，`writeSQL()` 按该顺序将数据写至流。

将 ClassGenerator 打包在 `gbasedbtjdbc_xx.jar` 文件中，因此，CLASSPATH 环境变量必须指向 `gbasedbtjdbc_xx.jar`。

使用 ClassGenerator 的语法如下：

```
java ClassGenerator rowtypename [-u URL] [-c classname]
```

classname 的缺省值是 rowtypename 的值。

如果未指定 URL 参数，则从 home 目录中的 setup.std 文件检索所需的信息。

setup.std 的结构如下：

```
URL jdbc:host-name:port-number

gbasedbserver gbasedbservername

database database

user user

passwd password
```

简单命名的行示例

要使用 ClassGenerator，请首先在数据库服务器上创建命名的行，如此示例中所示：

```
create row type employee (name char (20), age int);
```

接下来，运行 ClassGenerator：

```
java ClassGenerator employee
```

该类生成器生成 employee.java，如后所示，并从 setup.std 检索数据库 URL 信息，其有下列内容：

```
URL jdbc:davinci:1528
database test
user scott
passwd tiger
gbasedbserver picasso_ius
```

下列为生成的 .java 文件：

```
import java.sql.*;

import java.math.*;

public class employee implements SQLData
{
    public String name;
    public int age;
    private String sql_type;

    public String getSQLTypeName() { return "employee"; }

    public void readSQL (SQLInput stream, String type) throws
        SQLException
```

```
{
    sql_type = type;
    name = stream.readString();
    age = stream.readInt();
}

public void writeSQL (SQLOutput stream) throws SQLException
{
    stream.writeString(name);
    stream.writeInt(age);
}
}
```

嵌套的命名的行示例

对于嵌套的行，要使用 ClassGenerator，请首先在数据库服务器上创建命名的行：

```
create row type manager (emp employee, salary int);
```

接下来，运行 ClassGenerator。在此情况下，不查询 setup.std 文件，因为您在命令行提供了所有需要的信息：

```
java ClassGenerator manager -c Manager -u "jdbc:davinci:1528/test:user=scott;
password=tiger;gbasedbtserver=picasso_ius"
```

-c 选项定义您正在创建的 Java™ 类，其为 Manager（带有大写的 M）。

前面的命令生成下列 Java 类：

```
import java.sql.*;
import java.math.*;
public class Manager implements SQLData
{
    public employee emp;
    public int salary;
    private String sql_type;

    public String getSQLTypeName() { return "manager"; }

    public void readSQL (SQLInput stream, String type) throws
        SQLException
    {
        sql_type = type;
        emp = (employee)stream.readObject();
    }
}
```

```
salary = stream.readInt();
}

public void writeSQL (SQLOutput stream) throws SQLException
{
    stream.writeObject(emp);
    stream.writeInt(salary);
}
}
```

5.8 类型高速缓存信息

当将某些数据类型的对象插入至某些其他数据类型的列内时，GBase 8s JDBC Driver 通过调用 `SQLData.getSQLTypeName()` 方法，来核实提供的数据与数据库服务器期望的数据是否相匹配。驱动程序询问数据库服务器随同每一插入的类型信息。

这会在下列情况下发生：

- 当 `SQLData` 对象将输入插入至 `opaque` 类型列，且 `getSQLTypeName()` 返回该 `opaque` 类型的名称时
- 当 `Struct` 或 `SQLData` 对象将数据插入至行列，且 `getSQLTypeName()` 返回命名的行的名称时
- 当 `SQLData` 对象将数据插入至 `DISTINCT` 类型列时。

在数据库 URL 中，可设置环境变量 `ENABLE_TYPE_CACHE=TRUE`，以使得在首次检索时驱动程序高速缓存该数据类型信息。然后，在从数据库服务器请求数据之前，驱动程序向高速缓存请求该类型信息。

5.9 智能大对象数据类型

智能大对象是带有下列特性的大对象：

- 智能大对象可保存非常大量的数据。

当前，单个智能大对象可保存最多 4 TB 数据。此数据存储在称为 `sbspace` 的独立磁盘空间中。

- 智能大对象是可恢复的。

数据库服务器可将更改日志记录至智能大对象，因此，在系统或硬件故障时可恢复智能大对象数据。智能大对象的日志记录不是缺省的行为。

- 智能大对象支持对其数据的随机访问。

以“全部或者全不”方式访问简单大对象（BYTE 或 TEXT）；也就是说，数据库服务器一次返回您请求的所有简单大对象数据。对于智能大对象，可寻找所需的位置，并读或写所需的字节数。

- 可定制智能大对象的存储特征。

当创建智能大对象时，可指定智能大对象的存储特征，诸如：

- 数据库服务器是否根据当前数据库日志模式来日志记录智能大对象
- 数据库服务器是否保存最后一次访问智能大对象的痕迹
- 数据库服务器是否使用页标头来检测数据损坏与否

在数据库中作为 BLOB 和 CLOB 数据类型来存储智能大对象，您可以两种方式访问它们：

- 在 GBase 8s JDBC Driver 3.0 和后来版本中，以及支持支持智能大对象数据类型的 GBase 8s 服务器中，可使用 JDBC 3.0 规范中描述的标准 JDBC API 方法。这是较简单的方法。

已实现了对于 BLOB 和 CLOB 内部更新的下列 JDBC 3.0 方法：

```
int setBytes(long, byte[]) throws SQLException
```

```
void truncate(long) throws SQLException
```

在 GBase 8s JDBC Driver Version 3.0 或后来版本中实现来自 BLOB 接口的下列 JDBC 3.0 方法：

```
OutputStream setBinaryStream(long) throws SQLException
```

```
int setBytes(long, byte[], int, int) throws SQLException
```

在 GBase 8s JDBC Driver Version 3.0 或后来版本中，实现来自 CLOB 接口的下列 JDBC 3.0 方法：

```
OutputStream setAsciiStream(long) throws SQLException Writer
```

```
setCharacterStream(long) throws SQLException
```

```
int setString(long, String) throws SQLException
```

```
int setString(long, String, int, int) throws SQLException
```

- 可使用 GBase 8s 内基于智能大对象支持的 GBase 8s 扩展。此方法提供更多选项。

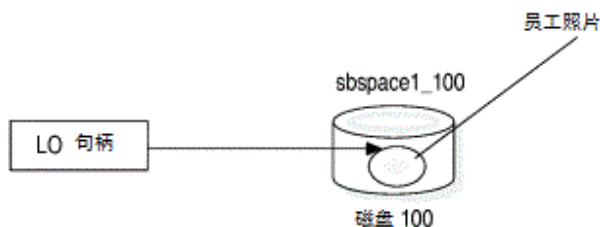
5.9.1 数据库服务器中的智能大对象

在 GBase 8s 数据库服务器中，智能大对象有两个部分：

- 数据，其存储在一个 sbspace 中
- 大对象句柄，称为 LO 句柄，其标识智能大对象数据在 sbspace 中的位置

假定您将员工的照片作为智能大对象存储。下图展示 LO 句柄如何包含关于在 sbospace1_100 sbospace 中实际员工照片的位置信息。

图: 数据库服务器中的智能大对象



在图中，sbospace 保存 LO 句柄标识的实际员工照片。要获取关于 sbospace 的结构，以及关于创建和删除 sbospace 的 onspaces 数据库实用程序的更多信息，请参阅《GBase 8s 管理员指南》。

重要： 仅可在 sbospace 中存储智能大对象。在尝试将智能大对象插入至数据库之前，必须创建 sbospace。

由于智能大对象可能非常大，因此，数据库服务器仅在数据库表中存储它的 LO 句柄；然后，它可使用此句柄来找到 sbospace 中智能大对象的实际数据。这种安排最小化表大小。

应用程序从数据库取得 LO 句柄，并使用它来定位智能大对象数据，再打开智能大对象进行读写操作。

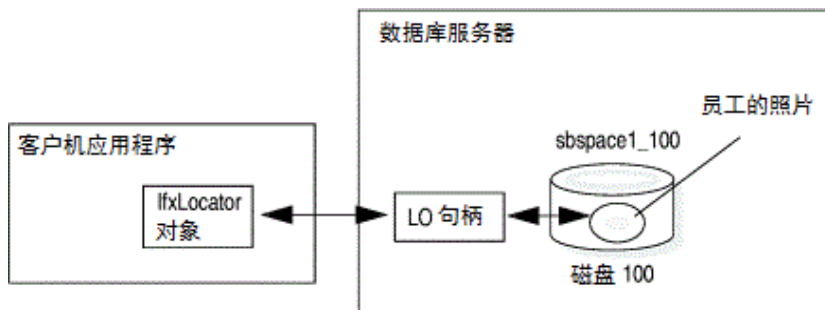
5.9.2 客户机应用程序中的智能大对象

在客户机上，JDBC 应用程序可使用 ResultSet 方法来访问智能大对象数据，诸如：

- 对于 CLOB 数据，getClob() 和 getAsciiStream()
- 对于 BLOB 数据，getBlob() 和 getBinaryStream()
- 对于 CLOB 和 BLOB 数据，getString()

在客户机侧，JDBC 驱动程序通过 IfxLocator 对象来引用 LO 句柄。JDBC 应用程序取得 IfxLocator 类的一个实例，来取得智能大对象定位器句柄，如下图所示。应用程序独立地创建智能大对象，然后，将智能大对象插入至不同的列，设置是在多表中。使用多线程，应用程序可并行地从智能大对象的不同部分写或读数据，这非常高效。

图: 在客户机应用程序中定位智能大对象



创建智能大对象

基于下列类实现 GBase 8s 智能大对象：

- **IfxLobDescriptor** 存储大对象的属性。
- **IfxLocator** 包含至数据库服务器中大对象的句柄。
- **IfxSmartBlob** 包含操作智能大对象的方法，诸如在对象内定位、从对象读取数据，以及将数据写至对象。
- **IfxBlob** 和 **IfxCblob** 实现来自 JDBC 3.0 规范的 **java.sql.Blob** 和 **java.sql.Clob** 接口。
- **IfxLoStat** 存储关于大对象的状态信息。

提示： 本部分描述如何使用 GBase 8s 智能大对象接口，但目前它未记录该接口中的每个方法和参数。要获取该接口中所有方法及其参数的完整参考，请参阅 GBase 8s JDBC Driver 的 javadoc 文件，其位于安装驱动程序处的 doc/javadoc 目录中。

要创建智能大对象：

1. 对于新的智能大对象，请确保该智能大对象有为其数据指定的一个 sbspace。

要获取关于创建 sbspace 的 onspaces 实用程序的详尽资料，请参阅《GBase 8s 管理员指南》。要获取创建 sbspace 的示例，请参阅 设置 sbspace 特征的示例。

2. 创建 IfxLobDescriptor 对象。

这允许您设置智能大对象的存储特征。当 IfxSmartBlob.IfxLoCreate() 方法创建大对象时，驱动程序将 IfxLobDescriptor 对象传至数据库服务器。

3. 如果需要，请调用 IfxLobDescriptor 对象中的方法，来指定存储特征。

对于大多数智能大对象，sbspace 名称是您需要指定的唯一存储特征。数据库服务器可计算所有其他存储特征的值。可设置特殊的存储特征，来覆盖这些计算的值。**然而，大多数应用程序不需要在此详细级别设置存储特征。**要获取更多信息，请参阅 使用存储特征。

4. 创建 IfxLocator 对象。

这是至客户机上智能大对象的指针。

5. 创建 IfxSmartBlob 对象。

这允许您对智能大对象执行各种通用操作。

6. 执行 IfxSmartBlob.IfxClobCreate() 方法，来在数据库服务器中创建大对象。

IfxClobCreate() 采用 IfxClobLocator 和 IfxClobDescriptor 对象作为参数，来表示数据库服务器中的智能大对象。

7. 执行 IfxSmartBlob.IfxClobWrite(), 来将数据写至数据库服务器中的智能大对象。

8. 执行附加的 IfxSmartBlob 方法，来在对象内定位、从对象读取，等等。

9. 执行 IfxSmartBlob.IfxClobClose(), 来关闭大对象。

10. 将智能大对象插入至数据库内（请参阅 将智能大对象插入至列内）。

11. 执行 IfxSmartBlob.IfxClobRelease(), 来释放定位器指针。

创建 IfxClobDescriptor 对象

IfxClobDescriptor 类存储智能大对象的内部存储特征。在数据库服务器上可创建智能大对象之前，您必须创建一 IfxClobDescriptor 对象，如下：

```
IfxClobDescriptor loDesc = new IfxClobDescriptor(conn);
```

conn 参数是 java.sql.Connection 对象。IfxClobDescriptor() 构造函数为该对象设置所有缺省值。

要获取关于内部存储特征的更多信息，请参阅 使用存储特征。

创建 IfxClobLocator 对象

IfxClobLocator 对象（通常称为定位器指针或大对象定位器）表示智能大对象的位置，如图 1 中所示；对于特殊的大对象，定位器指针是在数据库服务器与客户机之间的通讯链接。在它创建大对象或打开大对象进行读写之前，应用程序必须创建一 IfxClobLocator 对象：

```
IfxClobLocator loPtr = new IfxClobLocator();  
IfxClobLocator loPtr = new IfxClobLocator(Connection conn);
```

如果抛出异常，则使用这些构造函数的第二个来显示本地化的错误消息。要获取更多信息，请参阅 支持全球化的错误消息。

创建 IfxSmartBlob 对象

要创建智能大对象，并取得对方法的访问，以对该对象执行操作，请调用 IfxSmartBlob 构造函数，将引用传至 JDBC 连接：

```
IfxSmartBlob smb = new IfxSmartBlob(myConn)
```

一旦编写了需要在智能大对象中执行操作的所有方法，您即可使

用 IfxSmartBlob.IfxClobCreate() 方法，来在数据库服务器中创建大对象，并打开它来在应用程序内访问。该方法签名如下：

```
public int IfxClobCreate(IfxClobDescriptor loDesc, int flag,
```

```

IfxLocator loPtr) throws SQLException
public int  IfxLoCreate(IfxLobDescriptor loDesc, int flag,
IfxBlob blob)throws SQLException
public int  IfxLoCreate(IfxLobDescriptor loDesc, int flag,
IfxCblob clob throws SQLException

```

返回值是定位器句柄，您可在后续的读、写、搜索和关闭方法中使用它（您可将它作为定位器文件描述符（lofd）参数，将它传至操作打开的智能大对象方法；在 智能大对象内的位置 的开始描述这些方法）。

flag 参数是一整数值，指定在服务器中打开新的智能大对象所处的访问模式。对打开的智能大对象，访问模式确定哪些读写操作是有效的。如果未指定值，则以只读模式打开该对象。

请在下表中以 IfxLoCreate() 和 IfxLoOpen() 方法，来使用访问模式 flag 值，以特定的访问模式打开或创建智能大对象。

访问模式	用途	IfxSmartBlob 中的 flag 值
Read only	仅允许读操作	LO_RDONLY
Write only	仅允许写操作	LO_WRONLY
Write/Append	将写的数据追加至智能大对象的末尾。单独使用时，它等同于 write-only 模式，后跟一个至智能大对象末尾的搜索。读操作失败。当仅以 write/append 模式打开智能大对象时，以 write-only 模式打开该智能大对象。搜索操作移动搜索位置，但对智能大对象的读操作失败，且搜索位置保持不变，与写之前的位置一样。在搜索位置发生写操作，然后移动该搜索位置。	LO_APPEND
Read/Write	允许读和写操作。	LO_RDWR

下列示例展示如何使用 LO_RDWR flag 值：

```

IfxSmartBlob smb = new IfxSmartBlob(myConn);
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);

```

预先创建 loDesc 和 loPtr 对象，分别为 IfxLobDescriptor 和 IfxLocator 对象。

当打开智能大对象时，数据库服务器使用下列系统缺省值。

打开模式信息

缺省的打开模式

访问模式

Read-only

访问方法

Random

缓冲

Buffered access

锁定

Whole-object locks

要获取关于锁定的更多信息，请参阅 使用锁。

下表提供打开模式标志的全集：

打开模式标志	描述
LO_APPEND	<p>将写的数据追加至智能大对象的末尾</p> <p>单独使用时，它等同于 write-only 模式，后跟至智能大对象末尾的搜索。读操作失败。</p> <p>当仅以 write/append 模式打开智能大对象时，以 write-only 模式打开智能大对象。搜索操作移动搜索位置，但对智能大对象的读操作失败，且搜索位置保持不变，与写之前的位置一样。在搜索位置处发生写操作，然后，移动搜索位置。</p>
LO_WRONLY	仅允许写操作
LO_RDONLY	仅允许读操作
LO_RDWR	允许写和读操作
LO_DIRTY_READ	<p>仅对于打开</p> <p>允许读取智能大对象的未提交的数据页</p> <p>在设置模式为 LO_DIRTY_READ 之后，不可写至智能大对象。当设置此标志时，对于智能大对象，请将当前的事务隔离模式重置为 Dirty Read。</p>

	在 Dirty Read 模式下，请不要对从智能大对象取得的数据进行基础更新。
LO_RANDOM	覆盖优化器决策 指示随意 I/O，且数据库服务器不应预读。缺省的打开模式。
LO_SEQUENTIAL	覆盖优化器决策 指示读为前向或反向顺序的。
LO_FORWARD	仅用于顺序访问，来指示前向
LO_REVERSE	仅用于顺序访问，来指示反向
LO_BUFFER	使用数据库服务器缓冲池。
LO_NOBUFFER	不使用标准数据库服务器缓冲池。使用来自数据库服务器的会话池的私有缓冲区。
LO_NODIRTY_READ	不允许对智能大对象进行脏读。要获取更多信息，请参阅 LO_DIRTY_READ 标志。
LO_LOCKALL	指定会在整个智能大对象上发生锁定
LO_LOCKRANGE	对于字节的范围，指定会发生锁定 当放置锁时，请通过 <code>IfxSmartBlob.IfxLoLock()</code> 方法来指定字节的范围。

将智能大对象插入至列内

在创建智能大对象之后，您必须将它插入至 BLOB 或 CLOB 列，以将它保存在数据库中。要这么做，您必须将 IfxLocator 对象转换为 IfxBlob 或 IfxCblob 对象，这依赖于列类型。

要将智能大对象插入至 BLOB 或 CLOB 列，请：

- 1. 创建 IfxBlob 或 IfxCblob 对象，如下：

```
IfxBlob blb = new IfxBlob(loPtr);
```

loPtr 参数是从前面的步骤取得的集合中的一个 IfxLocator 对象。

- 2. 使用 PreparedStatement.setBlob() 或 setClob() 方法，来将该对象插入至列内。

重要： 在插入执行之前，在数据库服务器中必须存在智能大对象的 sbspace。

访问智能大对象

遵循下列步骤来使用 GBase 8s 扩展，以从数据库列选择智能大对象。

要访问智能大对象，请：

1. 将 `java.sql.Blob` 或 `java.sql.Clob` 对象强制转型为 `IfxBlob` 或 `IfxClob` 对象。
2. 使用 `IfxBlob.getLocator()` 或 `IfxClob.getLocator()` 方法，来抽取 `IfxLocator` 对象。
3. 创建 `IfxSmartBlob` 对象。
4. 使用 `IfxSmartBlob.IfxLoOpen()` 方法，来打开智能大对象。
5. 使用 `IfxSmartBlob.IfxLoRead()` 方法来从智能大对象读取数据。
6. 使用 `IfxSmartBlob.IfxLoClose()` 方法来关闭智能大对象。
7. 通过调用 `IfxSmartBlob.IfxLoRelease()` 方法，来释放服务器中的定位器指针。

标准 JDBC `ResultSet` 方法，诸如 `ResultSet.getBinaryStream()`、`ResultSet.getAsciiStream()`、`ResultSet.getString()`、`ResultSet.getBytes()`、`ResultSet.getBlob()` 和 `ResultSet.getClob()`，可从表访问 `BLOB` 或 `CLOB` 数据。然后，GBase 8s 扩展类可访问该数据。

5.9.3 在智能大对象上执行操作

在数据库服务器中，在有下列数据类型之一的列中，可直接存储智能大对象：

- `CLOB` 数据类型保存文本数据。
- `BLOB` 数据类型可在不可分的字节流中存储任何类别的二进制数据。

`CLOB` 或 `BLOB` 列为智能大对象保存 `LO` 句柄。因此，当选择 `CLOB` 或 `BLOB` 列时，不取得智能大对象的实际数据，而是标识此数据的 `LO` 句柄。智能大对象的列有理论上的 4 TB 限制，并由磁盘容量确定实际的限制。

可使用下列方式之一来在列中存储智能大对象：

- 对于智能大对象的直接访问，创建 `CLOB` 或 `BLOB` 数据类型的列。
- 要在原子数据类型内隐藏智能大对象，创建保存智能大对象的 `opaque` 类型。

在客户机应用程序中，`IfxBlob` 和 `IfxClob` 类是在 JDBC 3.0 规范中描述的处理智能大对象数据方式与 GBase 8s 扩展之间的桥梁。`IfxBlob` 类实现 `java.sql.Blob` 接口，`IfxClob` 类实现 `java.sql.Clob` 接口。GBase 8s 扩展需要一个 `IfxLocator` 对象，来标识数据库服务器中的智能大对象。

当查询包含 `BLOB` 或 `CLOB` 类型列的表时，返回一个 `Blob` 或 `Clob` 类型的对象，这依赖于列类型。然后，可使用 `Blob` 或 `Clob` 类型对象的 JDBC 3.0 支持函数，来访问智能大对象。

构造函数从 `IfxLocator` 对象 `loPtr` 创建一个 `IfxBlob` 或 `IfxClob` 对象：


```
public IfxBlob(IfxLocator loPtr)
public IfxCblob(IfxLocator loPtr)
```

下列定位器方法从 IfxBlob 或 IfxCblob 对象返回一个 IfxLocator 对象。然后，您可使用 IfxSmartBlob.IfxLoOpen()、IfxLoRead() 和 IfxLoWrite() 方法，来打开、读和写智能大对象：

```
public IfxLocator getLocator() throws SQLException
```

打开智能大对象

IfxSmartBlob 类中的下列方法打开数据库服务器中现有的智能大对象：

```
public int IfxLoOpen(IfxLocator loPtr, int flag) throws
    SQLException
public int IfxLoOpen(IfxBlob blob, int flag) throws SQLException
public int IfxLoOpen(IfxCblob clob, int flag) throws SQLException
```

第一个版本打开由定位器指针 loPtr 引用的智能大对象。第二个和第三个版本分别打开由指定的 IfxBlob 和 IfxCblob 对象引用的智能大对象。flag 参数是来自 创建 IfxSmartBlob 对象 中表的一个值。

智能大对象内的位置

IfxSmartBlob 类中的 IfxLoTell() 方法返回当前的查找位置，其为智能大对象下一读或写操作的偏移量。IfxSmartBlob 类中的 IfxLoSeek() 方法设置已打开的大对象内的读或写位置。

```
public long IfxLoTell(int lofd)
public long IfxLoSeek(int lofd, long offset, int whence) throws
    SQLException
```

绝对位置依赖于第二个参数 offset 的值和第三个参数 whence 的值。

lofd 参数是由 IfxLoCreate() 或 IfxLoOpen() 方法返回的定位符文件描述符。offset 参数是从开始的查找位置的偏移量。

whence 参数标识开始查找位置。请使用下表中的 whence 值来定义智能大对象内的位置，来启动查找操作。

开始的查找位置	whence 值
智能大对象的开头	IfxSmartBlob.LO_SEEK_SET
智能大对象中的当前位置	IfxSmartBlob.LO_SEEK_CUR
智能大对象的结束	IfxSmartBlob.LO_SEEK_END

返回值是表示智能大对象内绝对位置的长整数。

下列示例展示如何使用 LO_SEEK_SET whence 值：


```
lfxLobDescriptor loDesc = new lfxLobDescriptor(myConn);
lfxLocator loPtr = new lfxLocator();
lfxSmartBlob smb = new lfxSmartBlob(myConn);
int loFd = smb.lfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
int n = smb.lfxLoWrite(loFd, fin, fileLength);
smb.lfxLoClose(loFd);
loFd = smb.lfxLoOpen(loPtr, smb.LO_RDWR);
long m = smb.lfxLoSeek(loFd, 200, smb.LO_SEEK_SET);
```

将写位置设置在从智能大对象开头的 200 字节偏移量处。

从智能大对象读取数据

可以下列方式从智能大对象读取数据：

- 将数据从对象读取至 byte[] 缓冲区内。
- 将数据从对象读取至文件输出流内。
- 将数据从对象读取至文件内。

请使用 IfxSmartBlob 类中的 lfxLoRead() 方法来从智能大对象读取至缓冲区内或文件输出流内，其有下列签名：

```
public byte[] lfxLoRead(int lofd, int nbytes) throws SQLException
    public int lfxLoRead(int lofd, byte[] buffer, int nbytes) throws
        SQLException
    public int lfxLoRead(int lofd, FileOutputStream fout, int nbytes
        throws SQLException
    public int lfxLoRead(int lofd, byte[] buffer, int nbytes, int
        offset throws SQLException
```

lofd 参数是由 lfxLoRead() 或 lfxLoOpen() 方法返回的定位器文件描述符。

第一个版本将 nbytes 字节数据返回至字节缓冲区内。此版本的方法为缓冲区分配内存。第二个版本将 nbytes 字节数据读至已分配了的缓冲区内。第三个版本将 nbytes 字节数据读至文件输出流内。第四个版本将 nbytes 字节数据读至在当前查找位置加上智能大对象内偏移量处开始的字节缓冲区内。最后三个版本的返回值指示读取的字节数。

使用 IfxSmartBlob 类中的 lfxLoToFile() 方法，来从智能大对象读取至文件内，其有下列签名：

```
public int lfxLoToFile(lfxLocator loPtr, String filename, int flag , int whence) throws
    SQLException
    public int lfxLoToFile(lfxBblob blob, String filename, int flag , int whence) throws
        SQLException
```

```
public int IfxLoToFile(IfxCblob clob, String filename, int flag , int whence) throws
SQLException
```

第一个版本读取由定位器指针 `loPtr` 引用的智能大对象。第二个和第三个版本分别读取由指定的 `IfxBblob` 和 `IfxCblob` 对象引用的智能大对象。

`flag` 参数指示该文件是在客户机上，还是在服务器上。该值

为 `IfxSmartBlob.LO_CLIENT_FILE` 或 `IfxSmartBlob.LO_SERVER_FILE`。`whence` 参数指示开始的查找位置。要了解这些值，请参阅 智能大对象内的位置。

提示： 在下列函数的签名中，已有一更改：

```
IfxSmartBlob.IfxLoToFile().
```

此函数用于接受四个参数，但现在仅接受三个参数。`IfxLoToFile()` 的所有三个重载的函数都接受三个参数。

将数据写至智能大对象

可以下列方式将数据写至智能大对象：

- 将数据从 `byte[]` 缓冲区写至该对象。
- 将数据从文件输入流写至该对象。
- 将数据从文件写至该对象。

使用 `IfxSmartBlob` 类中的 `IfxLoWrite()` 方法，从 `byte[]` 缓冲区或文件输入流写至智能大对象：

```
public int IfxLoWrite(int lofd, byte[] buffer) throws SQLException
public int IfxLoWrite(int lofd, InputStream fin, int length)
throws SQLException
```

该方法的第一个版本将 `buffer.length` 字节数据从缓冲区写至智能大对象内。第二个版本将 `length` 字节数据从 `InputStream` 对象写至智能大对象内。

`lofd` 参数是由 `IfxLoCreate()` 或 `IfxLoOpen()` 方法返回的定位器文件描述符。`buffer` 参数是读数据处的 `byte[]` 缓冲区。`fin` 是 `InputStream` 对象，将数据从其写至智能大对象内。`length` 参数是写至智能大对象内的字节数。驱动程序返回写入的字节数。

请使用 `IfxSmartBlob` 类中的 `IfxLoFromFile()` 方法，来将数据从文件写至智能大对象：

```
public int IfxLoFromFile (int lofd, String filename, int flag, int offset, int amount) throws
SQLException
```

`lofd` 参数是由 `IfxLoCreate()` 或 `IfxLoOpen()` 方法返回的定位器文件描述符。`flag` 参数指示该文件是在客户机上，还是在服务器上。该值为

`IfxSmartBlob.LO_CLIENT_FILE` 或 `IfxSmartBlob.LO_SERVER_FILE`。

驱动程序返回写入的字节数。

截断智能大对象

请使用 `IfxSmartBlob` 类中的 `IfxLoTruncate()` 方法，来在指定的偏移量处截断大对象。该方法签名如下：

```
public void IfxLoTruncate(int lofd, long offset) throws
                        SQLException
```

`offset` 参数是截断智能大对象处的绝对位置。

度量智能大对象

请使用 `IfxSmartBlob` 类中的 `IfxLoSize()` 方法，来返回智能大对象的大小。此方法返回表示大对象大小的一个长整数。

该方法签名如下：

```
public long IfxLoSize(int lofd) throws SQLException
```

关闭和释放智能大对象

在执行了应用程序需要的所有操作之后，您必须关闭该对象，然后，释放服务器中的资源。执行这些任务的 `IfxSmartBlob` 类中的方法如下：

```
public void IfxLoClose(int lofd) throws SQLException
public void IfxLoRelease(IfxLocator loPtr) throws SQLException
public void IfxLoRelease(IfxBblob blob) throws SQLException
public void IfxLoRelease(IfxCblob clob) throws SQLException
```

要对同一大对象进行任何进一步访问，必须以 `IfxLoOpen()` 方法来重新打开它。

将 `IfxLocator` 转换为十六进制字符串

有些应用程序只能处理 ASCII 数据，例如，web 浏览器；它们需要将 `IfxLocator` 转换为十六进制字符串格式。在典型的基于 web 的应用程序中，web 服务器查询数据库表，并将结果发送至浏览器。不是发送整个智能大对象，web 服务器将定位器转换为十六进制字符串格式，并将它发送至浏览器。如果用户请求浏览器显示智能大对象，则浏览器将十六进制格式的定位器发回 web 服务器。然后，web 服务器从该十六进制字符串重构二进制定位器，并将对应的智能大对象数据发送至浏览器。

要在 `IfxLocator` 字节数组与十六进制数值之间转换，请使用罗列在下表中的方法。

执行的任务	方法签名	附加的信息
将字节数组转换为十六进制字符串	<code>public static String toHexString(byte[] byteBuf);</code>	对数据有效，而不是对在 <code>com. gbasedbt. util. stringUtil</code> 类中提供的 <code>IfxLocator</code>
将十六进制字符串转换为字节数组	<code>public static byte[] fromHexString(String str)</code>	对数据有效，而不是对 <code>com. gbasedbt. util. stringUtil</code>

执行的任务	方法签名	附加的信息
组	throws NumberFormatException;	类中提供的 IfxLocator
使用字节数组构造一个 IfxLocator 对象	public IfxLocator(byte[] byteBuf) throws SQLException;	在 IfxLocator 类中提供
将 IfxLocator 字节数组转换为十六进制字符串	public String toString();	在 IfxLocator 类中提供
将十六进制字符串转换为 IfxLocator 字节数组	public byte[] toBytes();	在 IfxLocator 类中提供

下列示例使用 `toString()` 和 `toBytes()` 方法，来从智能大对象访存定位器，然后，将它转换为十六进制字符串：

```
...

String hexLoc = "";
byte[] blobBytes;
byte[] rawLocA = null;
IfxLocator loc;
try
{
    ResultSet rs = stmt.executeQuery("select b1 from btab");
    while(rs.next())
    {
        IfxBlob b=(IfxBlob)rs.getBlob(1);
        loc =b.getLocator();
        hexLoc = loc.toString();
        rawLocA = loc.toBytes();
    }
}
catch(SQLException e)
{
}
```

下列示例使用 `IfxLocator()` 方法来构建 `IfxLocator`，然后使用其来读取智能大对象：

```
...  
  
    try  
    {  
        lfxLocator loc2 = new lfxLocator(rawLoc);  
        lfxSmartBlob b2 = new lfxSmartBlob((lfxConnection)myConn);  
        int lofd = b2.lfxLoOpen(loc2, b2.LO_RDWR);  
        blobBytes = b2.lfxLoRead(lofd, fileLength);  
    }  
    catch(SQLException e)  
    {}
```

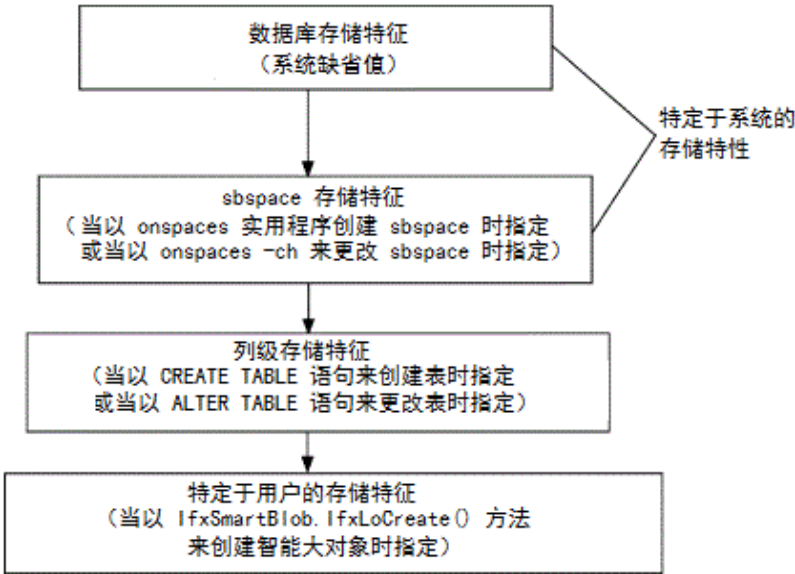
5.9.4 使用存储特征

存储特征告诉数据库服务器如何管理智能大对象。这些特征包括诸如大小、日志记录、锁定和打开模式这样的领域。关于存储特征，有下列选项：

- 使用特定于系统的存储特征作为取得智能大对象的存储特征的基础。
- 以下列之一来覆盖系统缺省值：
 - 为您想要在其中存储智能大对象的特别 CLOB 或 BLOB 列定义的存储特征
 - 特别 CLOB 或 BLOB 列特有的存储特征，称为列级存储特征
 - 仅为此智能大对象定义的特殊存储特征，称为特定于用户的存储特征

数据库服务器使用层级结构来从新的智能大对象取得存储特征，如下图所示。

图: 存储特征层级结构



对于给定的存储特征，在列级定义的任何值都覆盖特定于系统的值，任何用户级的值都覆盖列级的值。可在下表中展示的三个时刻指定存储特征。

何时指定	如何指定	要获取更多信息
当创建 sbspace 时	onspaces 实用程序的选项	特定于系统的存储特征 GBase 8s 管理员指南
当创建数据库表时	CREATE TABLE 语句的 PUT 子句中的关键字	GBase 8s SQL 指南：语法
当创建智能大对象时	在 ifxLobDescriptor 类中创建标志和方法	设置创建标志

特定于系统的存储特征

当数据库管理员初始化数据库服务器并以 onspaces 实用程序创建 sbspace 时，建立特定于系统的存储特征，如下：

- 如果 onspaces 实用程序已为特定的存储特征指定了值，则数据库服务器使用 onspaces 值作为特定于系统的存储特征。
- 如果 onspaces 实用程序还未为特定的存储特征指定值，则数据库服务器使用系统缺省值作为特定于系统的存储特征。

特定于系统的存储特征适用于存储在 sbspace 中的所有智能大对象，除非智能大对象以列级或特定于用户的存储特征特意覆盖它们。

要了解 onspaces 可设置的存储特征，以及系统缺省值，请参阅 表 1。

对于大多数应用程序，推荐您使用存储特征的特定于系统的缺省值。请注意下列例外：

- 应用程序需要取得额外的性能。
可使用 `ifxLobDescriptor` 中的 `setXXX()` 方法，来更改新智能大对象的磁盘存储信息。要获取更多信息，请参阅 设置创建标志。
- 想要使用现有智能大对象的存储特征。
`IfxLoStat.getLobDescriptor()` 方法可取得打开的智能大对象的大对象描述符。然后，可创建新的对象，并使用 `IfxSmartBlob.ifxLoAlter()` 方法来将它的特征设置为新的描述符。要获取更多信息，请参阅 更改存储特征。
- 正在使用多个智能大对象，且不想使用缺省的 `sbospace`。
DBA 可以 `onconfig` 文件中的 `SBSPACENAME` 配置参数来指定缺省的 `sbospace` 名称。然而，必须确保您创建的智能大对象位置（`sbospace` 的名称）是正确的。如果未为您的智能大对象指定 `sbospace` 名称，则数据库服务器将它存储在此缺省的 `sbospace` 中。此安排可导致空间限制。
- 如果知道智能大对象的大小，请使用 `IfxLobDescriptor.setEstBytes()` 方法来在应用程序中指定此大小，而不使用 `onspaces` 实用程序（系统级）或 `CREATE TABLE` 或 `ALTER TABLE` 语句（列级）。

取得关于存储特征的信息

要获取智能大对象的列级存储特征，应用程序可调用 `IfxSmartBlob` 类中的下列方法，传递 `colname` 参数的列名称：

```
IfxLobDescriptor ifxLoCollInfo(java.lang.String colname) throws
    SQLException
```

大多数应用程序仅需要确保 `sbospace` 名称的正确存储特征（智能大对象的位置）。在创建 `IfxSmartBlob` 对象之前，通过调用 `ifxLobDescriptor` 类中的各种 `getXXX()` 方法，可取得此存储特征和其他存储特征的信息。下表总结 `getXXX()` 方法。

ifxLobDescriptor 中的方法签名	用途
<code>int getCreateFlags()</code>	取得对象的创建标志
<code>long getEstSize()</code>	取得对象的估算大小，以字节计
<code>int getExtSize()</code>	取得对象的 extent 大小
<code>long getMaxBytes()</code>	取得对象的最大大小，以字节计
<code>java.lang.String getSospace()</code>	取得对象存储在其中的数据库服务器中的 <code>sbospace</code> 名称

设置 sbospace 特征的示例

下列对 onspaces 实用程序的调用在 /dev/sbospace1 分区中创建名为 sb1 的 sbospace:

```
onspaces -c -S sb1 -p /dev/sbospace1 -o 500 -s 2000
-Df "AVG_LO_SIZE=32"
```

对于 sb1 sbospace 中的所有智能大对象，下表展示结果的特定于系统的存储特征。

表 1. sb1 sbospace 的特定于系统的存储特征

磁盘存储信息	特定于系统的值	由 onspaces 实用程序指定
extent 的大小	由数据库服务器计算	系统缺省值
下一 extent 的大小	由数据库服务器计算	系统缺省值
最小 extent 大小	由数据库服务器计算	系统缺省值
智能大对象的大小	32 KB（数据库服务器用作大小估算）	AVG_LO_SIZE
I/O 块的最大大小	由数据库服务器计算	系统缺省值
sbospace 的名称	sb1	-S 选项
日志记录	OFF	系统缺省值
最后访问时间	OFF	系统缺省值

使用磁盘存储信息

磁盘存储信息帮助数据库服务器确定如何最高效地管理磁盘上的智能大对象。

重要： 对于大多数应用程序，使用数据库服务器为磁盘存储信息计算的值。GBase 8s JDBC Driver 中提供的方法用作特殊情况。

此磁盘存储信息包括：

- extent 分配信息：
 - extent 大小：

分配 extent是数据库服务器一次分配给智能大对象的 sbospace 内连续字节的集合。数据库服务器以 extent 大小的增量为智能大对象执行存储分配。

通过调用 ifxLobDescriptor.setExtSize() 方法可指定 extent 大小。
 - 下一 extent 大小：

数据库服务器试图作为 chunk 中单个连续的区域来分配 extent。然而，如果没有足够大的单个 extent ，则数据库服务器必须使用所需的多个 extent 来满足当前的写请求。 在初始的 extent 填满之后，数据库服务器尝试分配另一连续的 extent 磁盘空间。此过程称为下一 extent 分配。

要获取关于 extent 的更多信息，请参阅《GBase 8s 管理员指南》中关于磁盘结构和存储的主题。

- 大小信息：
 - 新智能大对象中估算的字节数
 - 智能大对象可增长到的最大字节数

要指定大小信息，可使用 ifxLobDescriptor 类中的 setMaxBytes() 和 setEstBytes() 方法。

如果知道智能大对象的大小，则请使用 setEstBytes() 方法来指定此大小。这是设置 extent 大小的最佳方式，因为数据库服务器可分配整个智能大对象作为一个 extent。

- 位置：

标识存储智能大对象处位置的 sbospace 名称。要设置此名称，可使用 ifxLobDescriptor.setSbSpace() 方法。

数据库服务器使用磁盘存储信息，来确定如何最优地确定大小、分配和管理 sbospace 的 extent。它可为智能大对象计算除了 sbospace 名称之外的所有磁盘存储信息。

为智能大对象指定磁盘存储信息的方法。通过使用系统缺省值，或由 onspaces 实用程序指定的值来取得特定于系统的磁盘存储信息。由 CREATE TABLE 的 PUT 子句来指定列级存储特征。由 GBase 8s JDBC Driver 方法来指定特定于用户的存储特征。

下表总结为智能大对象指定磁盘存储信息的方式。

表 1. 指定磁盘存储信息.

磁盘存储信息	特定于系统的存储特征		列级存储特征	特定于用户的存储特征
	系统缺省值	由 onspaces 实用程序指定	由 CREATE TABLE 的 PUT 子句指定	由 GBase 8s JDBC Driver 方法指定
extent 的大小	由数据库服务器计算	EXTENT_SIZE	EXTENT SIZE	是
下一 extent 的大小	由数据库服务器计算	NEXT_SIZE	否	否
最小 extent 大小	4 KB	MIN_EXT_SIZE	否	否
智能大对象的大小	由数据库服务器计算	sbospace 中所有智能大对象的平均大小: AVG_LO_SIZE	否	特殊智能大对象的估计大小 特殊智能大对

磁盘存储信息	特定于系统的存储特征		列级存储特征	特定于用户的存储特征
	系统缺省值	由 onspaces 实用程序指定	由 CREATE TABLE 的 PUT 子句指定	由 GBase 8s JDBC Driver 方法指定
				象的最大大小
I/O 块的最大大小	由数据库服务器计算	MAX_IO_SIZE	否	否
sbspace 的名称	SBSPACENAME	-S 选项	智能大对象在其中的现有 sbspace 名称: IN 子句	是

使用日志记录、最后访问时间，以及数据完整性

数据库管理员和应用程序可影响某些附加的智能大对象属性：

- 是否在系统日志文件中记录智能大对象的更改
- 是否保存智能大对象的最后访问时间
- 如何格式化智能大对象的 sbspace 中的页

下表总结如何在系统、列和应用程序级别上修改这些属性。

表 1. 指定属性信息

属性信息	特定于系统的存储特征缺省值	特定于系统的存储特征，由 onspaces 实用程序指定	列级存储特征，由 CREATE TABLE 的 PUT 子句指定	特定于用户的存储特征，由 JDBC 驱动程序方法指定
日志记录	OFF	LOGGING	LOG、NO LOG	是
最后访问时间	OFF	ACCESSTIME	KEEP ACCESS TIME、NO KEEP ACCESS TIME	是
缓冲模式	OFF	BUFFERING	否	否
锁定模式	锁定整个智能大对象	LOCK_MODE	否	是

数据完整性	高完整性	否	HIGH INTEG、 MODERATE INTEG	是
-------	------	---	-------------------------------	---

日志记录

在缺省情况下，数据库服务器不日志记录智能大对象的用户数据。您可控制智能大对象的日志记录行为，作为它的创建标志的一部分。要获取更多信息，请参阅 设置创建标志。

当数据库执行日志记录时，由于下列原因，智能大对象可能导致长事务：

- 智能大对象非常大，甚至达到数 GB 大小。
日志记录用户数据所需的日志存储量很容易使日志溢出。
- 在数据集合可相当长的情况下，可能使用智能大对象。
例如，如果智能大对象保存低品质音频记录，则数据集合的量可能不太大，但记录的会话可能非常长。

简单的变通方法是将长事务分成几个较小的事务。然而，如果不能接受此情况，则可控制数据库服务器执行智能大对象日志记录的时间。（表 1 展示可如何控制智能大对象的日志记录行为。）

当启用日志记录时，数据库服务器日志记录对智能大对象的用户数据的更改。它按照当前的数据库日志模式来执行此日志记录。

对于不符合 ANSI 的数据库，数据库服务器不保证在事务提交时刷新关于智能大对象的日志记录。然而，元数据总能恢复至活动一致状态；也就是说，恢复至确保在元数据（智能大对象的控制信息，诸如引用计数）中不存在结构性不一致的状态。

符合 ANSI 的数据库服务器使用非缓冲的日志记录。当启用智能大对象日志记录时，在事务提交时，刷新关于智能大对象的所有日志记录（元数据和用户数据）。然而，在提交时刻，不保证将用户数据刷新至它的稳定存储位置。

当禁用日志记录时，即使数据库服务器日志记录其他数据库更改，数据库服务器也不日志记录对用户数据的更改。然而，数据库服务器始终日志记录对元数据的更改。因此，数据库服务器仍可将元数据恢复至活动一致状态。

重要： 请慎重考虑是否启用智能大对象的日志记录。数据库服务器会承受对智能大对象的相当高的开销。您还必须确保系统日志文件足够大，以保存智能大对象的值。当更新事务是活动的时，逻辑日志大小必须超过数据库服务器日志记录的数据的总量。

编写应用程序，以便于任何带有潜在长更新的智能大对象的事务不会导致其他事务等待。如果满足下列条件，则多个事务可访问同一智能大对象实例：

- 对于智能大对象，事务可访问包含 LO 句柄的数据库行。
对于同一智能大对象，如果多列持有 LO 句柄，则在同一智能大对象上可存在多个引用。

- 在智能大对象上，另一事务不持有互相冲突的锁。

要获取关于智能大对象锁的更多信息，请参阅 使用锁。

在加载操作完成之后，当加载智能大对象并重新启用它时，当禁用日志记录特性时，会产生最佳更新性能和最少的逻辑日志问题。如果开启日志记录，则在批量加载然后执行 0 级备份之前，您可能想要关闭日志记录。

最后访问时间

智能大对象的最后访问时间是数据库服务器最后读或写智能大对象时的系统时间。最后访问时间记录对智能大对象的用户数据和元数据的访问。以从 1970 年 1 月 1 日以来的秒数来存储此系统时间。数据库服务器在 `sbspace` 的元数据区域中存储此最后访问时间。

在缺省情况下，数据库服务器不保存最后访问时间。通过设置

`LO_KEEP_LASTACCESS_TIME` 创建标志并调用 `IfxLobDescriptor.setCreateFlags()` 方法，可指定保存最后访问时间。要获取更多信息，请参阅 设置创建标志。

对于智能大对象，数据库服务器还跟踪最后修改时间和最后状态更改。要获取更多信息，请参阅 使用状态特征。

重要： 请慎重考虑是否跟踪智能大对象的最后访问时间。对于智能大对象，日志记录与维护最后访问时间并发，都会使数据库服务器承受相当高的开销。

数据完整性

通过调用 `IfxLobDescriptor.setCreateFlags()` 方法，以 `LO_HIGH_INTEG` 和 `LO_MODERATE_INTEG` 创建标志，可指定数据完整性。要获取更多信息，请参阅 设置创建标志。

`sbpage` 是为智能大对象数据分配的单元，其存储在 `sbspace` 的用户数据区域中。`sbspace` 中 `sbpage` 的结构确定数据库服务器可提供何种程度的数据完整性。数据库服务器使用页标头和页结尾来检测不完整的写和数据损坏。

数据库服务器支持下列数据完整性级别：

- 高完整性告诉数据库，在每一 `sbpage` 中同时使用页标头和页结尾。
- 中等完整性告诉数据库服务器，在每一 `sbpage` 中仅使用页标头。

中等完整性提供下列优势：

- 它排除额外的数据复制操作，当 `sbpage` 有页标头和页结尾时，这是必需的。
- 它保持页上用户数据对齐，因为不出现页标头和页结尾。

对于包含大量通过数据库服务器移动的音频或视频文件，且不需要高数据完整性的智能大对象，中等完整性可能非常有用。在缺省情况下，数据库服务器使用 `sbspace` 页的高完整性（页标头和页结尾）。您可控制智能大对象的数据完整性，作为它的存储特征的一部分。

重要： 对于智能大对象，请慎重考虑是否使用 `sbpage` 的中等完整性。虽然中等完整性每页占用较少磁盘空间，但如果发生磁盘错误，它也降低了数据库服务器恢复信息的能力。

要获取关于 sbspace 页的信息，请参阅《GBase 8s 管理员指南》。

更改存储特征

IfxSmartBlob 类中的 IfxLoAlter() 方法允许您更改智能大对象的存储特征。

要更改智能大对象特征，请：

1. 创建新的大对象描述符。

例如：

```
IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
```

2. 调用 IfxLobDescriptor.setCreateFlags()、setEstBytes()、IfxLobDescriptor.setMaxBytes()、setExtSize 和 setSbspace()，来指定新特征：

```
public void setCreateFlags( int flags )
public void setEstBytes(long estSize)
public void setMaxBytes (long maxSize)
public void setExtSize (long extSize)
public void setSbspace(java.lang.String sbspacename)
```

flag 参数是来自 设置创建标志 的常量。

调用 IfxLoAlter() 来修改现有的智能大对象，以包含该新描述符：

```
public int IfxLoAlter(IfxLocator loPtr, IfxLobDescriptor loDesc)
    throws SQLException
public int IfxLoAlter(IfxBlob blob, IfxLobDescriptor loDesc)
    throws SQLException
public int IfxLoAlter(IfxClob clob, IfxLobDescriptor loDesc)
    throws SQLException
```

在进行更新之前，对于整个智能大对象，IfxLoAlter() 在服务器中取得一个排他锁。它保持此锁，直至更新完成为止。

设置创建标志

通过调用 IfxLobDescriptor.setCreateFlags() 方法，可更改下列特征：

- 日志记录特征

可指定 LO_LOG 或 LO_NOLOG 常量。

对于对应的智能大对象，LO_LOG 导致服务器遵循当前数据库日志使用的日志记录过程。此选项可生成大量日志流量，并增加逻辑日志填满的风险。

不采用完全日志记录，当初始地加载智能大对象时，您可以关闭日志记录，然后一旦加载了智能大对象，就再一次开启日志记录。如果使用 NO LOG，则稍后将智能

大对象元数据恢复至不存在结构性不一致的状态。在大多数情况下，都不存在事务不一致性，但不能保证该结果。

要获取关于日志记录的更多详细用法信息，请参阅 日志记录。

- 最后访问时间特征

可指定 LO_KEEP_LASTACCESS_TIME 或 LO_NOKEEP_LASTACCESS_TIME 常量。在智能大对象元数据中，LO_KEEP_LASTACCESS_TIME 记录最后读或写对应的智能大对象时的系统时间。

要获取关于最后访问时间的更详尽用法，请参阅 最后访问时间。

- 是否通过以页标头和页结尾来产生用户数据页，以检测不完整的写和数据损坏

可指定 LO_HIGH_INTEG 或 LO_moderate_integ 常量。LO_HIGH_INTEG 是缺省的数据完整性行为。

要获取关于数据完整性的更详尽用法，请参阅 数据完整性。

下列示例设置多个标志：

```
loDesc.setCreateFlags (lfxSmartBlob.LO_LOG+lfxSmartBlob.LO_TEMP+...)
```

并行 getXXX() 方法允许您取得大对象的当前存储特征：

```
public int getCreateFlags()
```

要获取关于所有特征的更详尽信息，请参阅《GBase 8s SQL 指南：语法》中描述 CREATE TABLE 语句的 PUT 子句的部分。

5.9.5 使用状态特征

IfxLoStat 类存储关于智能大对象的某些统计信息，诸如大小、最后访问时间、最后修改时间、最后状态更改，等等。下表展示您可取得的状态信息。

表 1. 智能大对象的状态信息

状态信息	描述
最后访问时间	最后访问智能大对象的时间，以秒计 仅当为智能大对象启用最后访问时间属性时，此值才可用。要获取更多信息，请参阅 最后访问时间。
最后更改时间	最后更改智能大对象的状态的时间，以秒计 状态更改包括元数据更改和用户数据更改（对引用数的数据更新和更改）。以从 1970 年 1 月 1 日以来的秒数来存储此系统时间。

最后修改时间	最后修改智能大对象的时间，以秒计 修改仅包括对元数据和用户数据的更改(数据更新)。以从 19701 年 1 月 1 日以来的秒数来存储此系统时间。 在某些平台上，最后修改时间可能还有微秒组件，可单独从秒组件取得它。
大小	智能大对象的大小，以字节计
存储特征	请参阅 使用存储特征。

要取得对状态结构的引用，请调用 `IfxSmartBlob` 类中的下列方法：

`IfxLoStat IfxLoGetStat(int lofd)`

要取得状态信息的特别分类，请调用下表中展示的方法。

表 2. 取得状态信息的方法

状态信息	<code>ifxLoStat</code> 类中的方法签名
最后访问时间	<code>int getLastAccessTime()</code>
最后更改时间	<code>int getLastStatusTime()</code>
最后修改时间	<code>int getLastModifyTimeM()</code> - 以微秒计的时间 <code>int getLastModifyTimeS()</code> - 取整到秒的时间
大小	<code>int getSize()</code>
存储特征	<code>ifxLobDescriptor getLobDescriptor()</code>

5.9.6 使用锁

要防止同时访问智能大对象数据，当打开智能大对象时，数据库服务器取得对此数据的锁。此智能大对象锁不同于以下种类的锁：

- 行锁

智能大对象上的锁不锁定智能大对象驻留其中的行。然而，如果您从行检索智能大对象，且该行仍为当前的，则数据库服务器可能持有行锁以及智能大对象锁。在智能大对象上持有锁，而不是在行上，因为许多列可能访问同一智能大对象数据。
- 在表的同一行中，不同智能大对象的锁

智能大对象上的锁不影响该行中的其他智能大对象。

下表展示智能大对象可支持的锁模式。

表 1. 智能大对象的锁模式

锁模式	用途	描述
lock-all	锁定整个智能大对象	指示锁请求适用于智能大对象的所有数据
byte-range	仅锁定智能大对象指定的部分	指示锁请求仅适用于指定字节数的智能大对象数据

当服务器打开智能大对象时，它使用下列信息来确定智能大对象的锁模式：

- 智能大对象的访问模式
数据库服务器取得如下锁：
 - 在共享模式下，当打开智能大对象来读取时（read-only）
 - 在更新模式下，当打开智能大对象来写时（write-only、read/write、write/append）
当在智能大对象上实际执行写操作（或某其他更新）时，服务器将此锁升级为排他锁。
- 当前事务的隔离级别
如果数据库表的隔离模式为 Repeatable Read，则服务器不释放它在智能大对象上取得的任何锁，直到事务结束为止。

在缺省情况下，服务器选择 lock-all 锁模式。

服务器保留该锁如下：

- 它持有共享模式锁和更新锁（其尚未升级至排他锁），直到发生下列事件之一为止：
 - 智能大对象关闭
 - 事务结束
 - 显式的请求来释放锁（仅对于 byte-range 锁）
- 它持有排他锁，直到事务结束为止，即使关闭智能大对象也如此。

当发生前述条件之一时，服务器释放智能大对象上的锁。

重要：即使智能大对象保持打开，在事务结束时，也会丢失锁。当服务器检测到智能大对象没有活动的锁时，当首次发生智能大对象访问时，它自动地取得新的锁。它取得的锁是基于智能大对象的原始访问模式的。

当当前事务终止时，服务器释放锁。然而，当执行下一需要锁的函数时，服务器再次取得锁。如果这不是期望的行为，则服务器侧 SQL 应用程序可使用 BEGIN WORK 事务阻塞，并在需要使用该锁的最后语句之后放置 COMMIT WORK 或 ROLLBACK WORK 语句。

byte-range 锁定

在缺省情况下,当数据库服务器需要锁定智能大对象时,它全部使用 lock-all 锁。lock-all 锁是一种 “全都或全不” 锁;也就是说,它们锁定整个智能大对象。当数据库服务器取得排他锁时,只要持有锁,其他用户就不可访问智能大对象的数据。

如果此锁定对于应用程序的并发要求过于苛刻,则可使用 byte-range 锁定,而不使用 lock-all 锁定。以 byte-range 锁定,您可指定在智能大对象数据中锁定的字节范围。如果其他用户访问该数据的其他部分,他们仍可获得自己的 byte-range 锁。

请使用 IfxSmartBlob 类中的 IfxLoLock() 方法,来指定 byte-range 锁定:

```
public long IfxLoLock(int lofd, long offset, int whence, long range, int lockmode) throws
SQLException
```

要解锁该对象中的字节范围,请使用 IfxLoUnLock() 方法:

```
public long IfxLoUnLock( int lofd, long offset, int whence, long range) throws
SQLException
```

lofd 参数是由 IfxLoCreate() 或 IfxLoOpen() 方法返回的定位器文件描述符。offset 参数是从开始的查找位置的偏移量。whence 参数标识开始的查找位置。在智能大对象内的位置中的表中描述这些值。

range 参数指示在智能大对象内要锁定或解锁的字节数。lockmode 参数指示要创建什么类型的锁。这些值可为

IfxSmartBlob.LO_EXCLUSIVE_MODE 或 IfxSmartBlob.LO_SHARED_MODE。

5.9.7 高速缓存大对象

每当从数据库服务器访存 BLOB、CLOB、text 或 byte 类型对象时,就在客户机内存中高速缓存该数据。如果大对象的大小大于 LOBCACHE 环境变量中的值,则在临时文件中存储该大对象数据。要获取关于 LOBCACHE 变量的更多信息,请参阅大对象的内存管理。

5.9.8 避免转移大对象的错误

在检查是否已发生了错误之前,使用 IFX_LOB_XFERSIZE 环境变量来指定从客户机应用程序向数据库服务器转移的 CLOB 或 BLOB 中的字节数。每转移指定的字节数,就发生错误检查一次。如果发生错误,则不发送剩余的数据,并报告错误。如果未发生错误,则继续文件转移,直到它结束为止。

例如,如果将 IFX_LOB_XFERSIZE 的值设置为 10485760 (10 MB),则每发送 10485760 字节的 CLOB 或 BLOB 之后,会发送错误检查。如果未设置 IFX_LOB_XFERSIZE 环境变量,则在转移整个 BLOB 或 CLOB 之后发生错误检查。

IFX_LOB_XFERSIZE 环境变量的有效范围为从 1 至 9223372036854775808 字节。请在客户机上设置 IFX_LOB_XFERSIZE 环境变量。

您应调整 IFX_LOB_XFERSIZE 的值，以适应您的环境。请将 IFX_LOB_XFERSIZE 环境变量设置得足够低，以便于尽早检测到大型 BLOB 或 CLOB 数据类型的传输错误，但不要低得超出所消耗的网络资源。

5.9.9 智能大对象示例

下列示例说明本部分中讨论的一些任务。

创建智能大对象

此示例说明 创建智能大对象 中展现的那些步骤。

```
file = new File("data.dat");
FileInputStream fin = new FileInputStream(file);

byte[] buffer = new byte[200];

IfxLobDescriptor loDesc = new IfxLobDescriptor(myConn);
IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob(myConn);

// Now create the large object in server. Read the data from the
file
// data.dat and write to the large object.
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
System.out.println("A smart-blob is created ");
int n = fin.read(buffer);
if (n > 0)
n = smb.IfxLoWrite(loFd, buffer);
System.out.println("Wrote: " + n + " bytes into it");

// Close the large object and release the locator.
smb.IfxLoClose(loFd);
System.out.println("Smart-blob is closed ");
smb.IfxLoRelease(loPtr);
System.out.println("Smart Blob Locator is released ");
```

将文件 data.dat 的内容写至智能大对象。

将数据插入至智能大对象

下列代码将数据插入至智能大对象：

```
String s = "insert into large_tab (col1, col2) values (?,?)";
pstmt = myConn.prepareStatement(s);

file = new File("data.dat");
FileInputStream fin = new FileInputStream(file);

byte[] buffer = new byte[200];

IfxLobDescriptor loDesc = new IfxLobDescriptor(myConn);
IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob(myConn);

// Create a smart large object in server
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
System.out.println("A smart-blob has been created ");
int n = fin.read(buffer);
if (n > 0)
n = smb.IfxLoWrite(loFd, buffer);
smb.IfxLoClose(loFd);

System.out.println("Wrote: " + n + " bytes into it");
System.out.println("Smart-blob is closed ");

Blob blb = new IfxBlob(loPtr);
pstmt.setInt(1, 2); // set the Integer column
pstmt.setBlob(2, blb); // set the blob column
pstmt.executeUpdate();
System.out.println("Binding of smart large object to table is
done");

pstmt.close();
smb.IfxLoRelease(loPtr);
System.out.println("Smart Blob Locator is released ");
```

将文件 data.dat 的内容写至 large_tab 表的 BLOB 列。

从智能大对象检索数据

本主题中的示例说明 访问智能大对象 中的步骤。

下列代码示例展示如何使用 GBase 8s 扩展类来访问智能大对象数据：

```
byte[] buffer = new byte[200];
System.out.println("Reading data now ...");
try
{
    int row = 0;
    Statement stmt = myConn.createStatement();
    ResultSet rs = stmt.executeQuery("Select * from demo_14");
    while( rs.next() )
    {
        row++;
        String str = rs.getString(1);
        InputStream value = rs.getAsciiStream(2);
        IfxBlob b = (IfxBlob) rs.getBlob(2);
        IfxLocator loPtr = b.getLocator();
        IfxSmartBlob smb = new IfxSmartBlob(myConn);
        int loFd = smb.IfxLoOpen(loPtr, smb.LO_RDONLY);

        System.out.println("The Smart Blob is Opened for reading ..");
        int number = smb.IfxLoRead(loFd, buffer, buffer.length);
        System.out.println("Read total " + number + " bytes");
        smb.IfxLoClose(loFd);
        System.out.println("Closed the Smart Blob ..");
        smb.IfxLoRelease(loPtr);
        System.out.println("Locator is released ..");
    }
    rs.close();
}
catch(SQLException e)
{
    System.out.println("Select Failed ...\n" +e.getMessage());
}
```

首先，`ResultSet.getBlob()` 方法取得 BLOB 类型的一个对象。需要强制转型来将返回的对象转换为 `IfxBlob` 类型的对象。接下来，`IfxBlob.getLocator()` 方法从 `IfxBlob` 对象取得一个 `IfxLocator` 对象。在 `IfxLocator` 对象可用之后，您可实例化一个 `IfxSmartBlob` 对象，并使

用 `IfxLoOpen()` 和 `IfxLoRead()` 方法，来读取智能大对象数据。访存 CLOB 数据是类似的，但它使用方法 `ResultSet.getClob()`、`IfxCblob.getLocator()`，等等。

如果使用 `getBlob()` 或 `getClob()` 来从 BLOB 类型的列访存数据，则不需要使用前一样例代码那样的 GBase 8s 扩展来检索实际的 BLOB 内容。您可简单地使用 `Java.Blob.getBinaryStream()` 或 `Java.Clob.getAsciiStream()` 来检索内容。GBase 8s JDBC Driver 使用与样例代码相同的步骤，为您隐式地从数据库服务器取得内容。此方法比前一示例中的方法更简单，但不提供读取 BLOB 列内容的那么多选项。

6 与不透明数据类型一起使用

不透明数据类型是您定义扩展数据库服务器的原子数据类型。数据库服务器不具有有关不透明数据类型的信息，直到您提供描述它的例程。

扩展数据库服务器频繁需要您创建用户定义的例程（UDR）来支持此扩展。UDR 是您创建的例程，可以在 SQL 语句，数据库服务器或其他 UDR 中调用。UDR 可以是不透明数据类型的一部分，也可以是单独的。

JDBC 3.0 标准提供 `java.sql.SQLInput` 和 `java.sql.SQLOutput` 方法访问不透明数据类型。这些接口的定义扩展为完全支持 GBase 8s 固定二进制和可变二进制不透明数据类型。此扩展包括以下接口：

- `IfmxUdtSQLInput`
- `IfmxUdtSQLOutput`

此外，以下类同样从 JDBC 客户端应用程序在数据库服务器中创建 Java™ 不透明类型和 UDR：

- `UDTManager`
- `UDTMetaData`
- `UDRManager`
- `UDRMetaData`

`UDTManager` 和 `UDRManager` 类提供将客户端 Java 类映射为不透明数据类型和 UDR，并将它们的实例存储在数据库中的基础架构。

此工具只能在客户端 JDBC 中使用。有关服务器端 JDBC 的功能和限制，请参阅 *J/Foundation 开发者指南*。

有关不透明数据类型和 UDR 的详细信息，请参阅以下手册：

- GBase 8s 用户定义的例程和数据类型开发者指南 描述了有关不透明类型和 UDR 的术语和概念，包括内部数据结构，支持的功能以及隐式和显式强制转换，**这些是您需要在本节中使用的信息。**
- J/Foundation 开发者指南 描述了在 Java 中编写 UDR 的具体信息。

6.1 IfmxUDTSQLInput 接口

com.gbasedbt.jdbc.IfmxUdtSQLInput 接口使用一些添加的方法扩展 java.sql.SQLInput。要使用这些方法，必须将 SQLInput 强制转型为 IfmxUdtSQLInput。此方法允许您执行以下功能：

- 读取数据。
- 于数据流中定位。
- 设置或获取数据的数据。

6.1.1 读取数据

readString() 方法读取数据流中下一个属性作为 Java™ 字符串。readBytes() 读取数据流中下一个属性作为 Java 字节组。这两种方法类似于 SQLInput.readBytes() 方法，只是读入的是固定长度的数据：

```
public String readString(int maxlen) throws SQLException;  
public byte[] readBytes(int maxlen) throws SQLException;
```

在这两种方法中，您必须提供长度，以便 GBase 8s JDBC Driver 可以适当地读取下一个属性，因为驱动程序不知道不透明数据类型的特征。maxlen 参数指定读入数据的最大长度。

6.1.2 于数据流中定位

getCurrentPosition() 方法检索输入流中的当前位置。setCurrentPosition() 方法将输入流中的位置更改为 position 参数指定的位置：

```
public int getCurrentPosition();  
public void setCurrentPosition(int position) throws SQLException;  
public void skipBytes(int len) throws SQLException;
```

position 参数必须是正整数。skipBytes() 方法根据 len 参数指定相对于当前位置的字节数更改输入流中位置。len 参数必须是正整数。

setCurrentPosition() 和 skipBytes() 中，如果在输入流的末尾之后指定新位置，则 GBase 8s JDBC Driver 生成 SQLException。

6.1.3 设置或获取数据属性

length()方法返回整个数据流的总长度。getAutoAlignment() 方法检索自动对齐功能的状态 TRUE 或 FALSE（on 或 off）。setAutoAlignment() 方法设置状态为 TRUE 或 FALSE：

```
public int length();  
public boolean getAutoAlignment();  
public void setAutoAlignment(boolean value);
```

重要： 如果数据尚未对齐，则设置此自动对齐功能可能导致输入流的字节丢失。JDBC 应用程序会提供已对齐的数据或将对齐功能设置为 FALSE。

6.2 IfmxUDTSQLOutput 接口

com.gbaset.jdbc.IfmxUdtSQLOutput 接口使用以下添加的方法扩展 java.sql.SQLOutput：

```
public void writeString(String str, int length) throws  
    SQLException;  
public void writeBytes(byte[] b, int length) throws SQLException;
```

要使用这些方法，必须将 SQLOutput 强制转型为 IfmxUdtSQLOutput。

使用 writeString() 方法将下一个属性作为一个 Java™ 字符串写入数据流。如果传递的字符串长度小于指定的长度，则 GBase 8s JDBC Driver 使用零补齐此字符串。

使用 writeBytes() 方法将下一个属性作为一个 Java 字节组写入数据流。

这些方法类似于 SQLOutput.writeBytes() 方法，除了写入数据流的数据是固定长度的。如果传递的字符串或数组的长度小于指定的长度，则 GBase 8s JDBC Driver 使用零补齐字符串或数组。在这些方法中，您必须为 GBase 8s JDBC Driver 提供长度，以便写入下一个属性，因为对于驱动程序而言不透明数据类型是未知的。

6.3 映射不透明数据类型

要将 GBase 8s 不透明类型映射到 Java™ 对象，该对象必须实现 java.sql.SQLData 接口。这些 Java 对象描述构成不透明类型的所有数据成员。它们是强类型；即 Java 对象的 readSQL 或 writeSQL 方法中的每个读取或写入方法都必须与不透明类型定义中相应的数据成员相匹配，因为类型结构是未知的。

GBase 8s JDBC Driver 还要求将所有不透明数据作为 mitypes.h（此文件包含在所有的 GBase 8s 安装中）中定义的 GBase 8s DataBlade API 数据类型。所有的不透明类型都存储在数据库服务器的 C 结构中，该结构由不透明类型中定义的各种 DataBlade API 组成。

如果您使用 UDT 和 UDR Manager 工具创建不透明类型，则需要处理 Java 和 C 之间的映射。有关更多信息，请参阅 创建不透明类型和 UDR。

6.4 键入高速缓存信息

当某些数据类型的对象将数据插入某些其它数据类型的列时，GBase 8s JDBC Driver 会验证提供的数据是否与数据库服务器通过调用 `SQLData.getSQLTypeName()` 方法期望的数据匹配。驱动程序向数据库服务器询问每次插入的类型信息。

在以下情况中会发生这种情况：

- 当一个 `SQLData` 对象将数据插入到一个不透明类型的列中时，`getSQLTypeName()` 返回不透明类型的名称
- 当 `Struct` 或 `SQLData` 对象向行列插入数据时，`getSQLTypeName()` 返回已命名行的名称
- 当一个 `SQLData` 对象将数据插入到 `DISTINCT` 类型列中时

在数据库 URL 中，可以设置环境变量 `ENABLE_TYPE_CACHE=TRUE`，以使驱动程序在首次检索数据类型信息时缓存该数据类型信息。在向数据库服务器请求数据之前，驱动程序会向缓存询问类型信息。

6.5 不支持的方法

不透明类型不支持下列 `SQLInput` 和 `SQLOutput` 方法：

- **java.sql.SQLInput**
 - `readAsciiStream()`
 - `readBinaryStream()`
 - `readBytes()`
 - `readCharacterStream()`
 - `readObject()`
 - `readRef()`
 - `readString()`
- **java.sql.SQLOutput**
 - `writeAsciiStream(InputStream x)`
 - `writeBinaryStream(InputStream x)`
 - `writeBytes(byte[] x)`

- writeCharacterStream(Reader x)
- writeObject(Object x)
- writeRef(Ref x)
- writeString(String x)

6.6 创建不透明类型和 UDR

UDTManager 和 UDRManager 类可以使您更加容易地在数据库服务器中创建和部署不透明类型和用户定义的例程（UDR）。

在使用本节中信息之前，请阅读以下两个手册：

- 有关配置您的系统支持 Java™ UDR 的信息，请参阅 J/Foundation 开发者指南。
- 有关开发不透明类型的详细信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南。

6.6.1 创建不透明类型和 UDR 概述

在数据库服务器中，任何实现 java.sql.SQLData 接口并可以被 Java™ Virtual Machine 访问的 Java 类都可以存储为不透明类型。UDTManager 和 UDRManager 类与它们支持的 UDTMetaData 和 UDRMetaData 类一起使用，将此功能扩展到您的客户端应用程序：您的 Java 客户端应用程序可以使用这些类创建不透明类型和用户定义的例程，并将它们的类定义传送到数据库服务器。客户端不需要访问到数据库服务器来使用此功能。

重要：此功能与服务器支持密切配合，可以创建和使用 Java 不透明类型和用户定义例程。数据库服务器版本中存在的 Java 不透明类型和用户定义例程的任何限制同样适用于您在客户端应用程序中创建 Java 不透明类型和例程。

当使用 UDTManager 和 UDTMetaData 类时，GBase 8s JDBC Driver 为您的应用程序执行以下操作：

1. 获取您指定的 JAR 文件
2. 将此 JAR 文件从客户端本地区域传输到服务器本地区域

使用 UDTManager.setJarFileTmpPath() 方法定义服务器本地区域。在 UNIX™ 系统上默认为 /tmp，在 Windows™ 系统上默认为 C:\temp。

3. 在服务器中安装此 JAR 文件
4. 使用 CREATE OPAQUE TYPE SQL 语句在数据库中注册不透明数据类型，从 UDTMetaData 类获取输入

5. 注册支持函数并使用 CREATE 函数和 CREATE CAST SQL 语句强制转换为不透明类型

可以使用 UDTMetaData 类中的 setSupportUDR() 和 setXXXCast() 方法定义支持的函数。

如果您未为不透明类型提供输入和输出函数，则驱动程序注册缺省的函数（有关此功能的任何限制，请参阅发行说明）。

6. 注册您指定的任何其他不支持的例程或强制转换（如果有的话），从应用程序中的 UDTMetaData.setUDR() 和 UDTMetaData.setXXXCast() 方法调用中获取输入
7. 在 SQL OPAQUE 类型和 Java 对象中创建映射（使用 sqlj.setUDTextName() 方法）

当使用 UDRManager 和 UDRMetaData 类时，GBase 8s JDBC Driver 执行以下操作：

1. 获取您指定的 JAR 文件
2. 将此 JAR 文件从客户端本地区域传输到服务器本地区域
3. 在服务器中安装此 JAR 文件
4. 使用 CREATE FUNCTION TYPE SQL 语句在数据库中注册不透明数据类型，在您的应用程序获取 UDRMetaData.setUDR() 方法调用的输入

UDT 和 UDR Manager 工具中的方法执行以下主要功能：

- 使用服务器提供的缺省输入和输出方法，在 Java 类不存在的情况下，在 Java 中创建不透明类型
- 将客户端上现有的 Java 类转换为数据库服务器中不透明类型和 UDR
- 将 Java 静态方法转换为 UDR

6.6.2 准备创建不透明类型和 UDR

在使用 UDT 和 UDR Manager 工具之前，请按照以下步骤执行任务：

- 请确保您的数据库服务器支持 Java™。
- UDT 和 UDR Manager 工具在不支持 Java 的服务器上无法运行。
- 您的 CLASSPATH 设置中包含 gbasedbtjdbc_xx.jar 文件。
- 在数据库服务器中创建名为 /usr/gbasedbt 的目录，将它的所有者和组设置为用户 gbasedbt，将权限设置为 777。
- 将以下条目添加到数据库服务器的 /etc/group 文件中。

```
gbasedbt::unique-id-number:
```

- 检查驱动程序和数据库服务器的发版说明，以获取此版本中的进一步的限制。

6.6.3 创建不透明类型

使用 UDT Manager, 可以从实现 SQLData 接口的现有 Java™ 类创建 Java 不透明类型。UDT Manager 还可以帮助您在不需要准备 Java 类的情况下, 创建 Java 不透明类型, 可以指定要创建的不透明类型的特征, UDT Manager 工具创建 Java 类然后创建 Java 不透明类型。

按照本节中的步骤使用 UDTManager 类。

从现有 Java 类创建不透明类型

从现有 Java™ 类创建不透明类型:

1. 确保类符合转换为不透明类型的要求。
有关要求, 请参阅Java 类的要求。
2. 如果您不想使用服务器提供的缺省输入和输出例程, 请编写支持的 UDR 以进行输入和输出。
有关编写支持的 UDR 的一般信息, 请参阅 GBase 8s 用户定义的例程和数据类型开发者指南。
3. 在数据库服务器上创建一个缺省 sbospace 来保存包含不透明类型代码的 JAR 文件。
有关创建 sbospace 的信息, 请参阅您的数据库服务器的《GBase 8s 管理员指南》和 J/Foundation 开发者指南。
4. 打开一个 JDBC 连接。
确保数据库对象与连接对象相关联。没有数据库对象, 驱动程序就无法创建不透明类型。有关创建具有数据库对象的连接的详细信息, 请参阅连接至数据库。
5. 安装 UDTManager 对象和 UDTMetaData 对象:

```
UDTManager udtmgr = new UDTManager(connection);  
UDTMetaData mdata = new UDTMetaData();
```

6. 通过调用 UDTMetaData 对象中的方法设置不透明类型的属性。

至少, 您必须指定 SQL 名称、UDT 长度和 JAR 文件 SQL 名称。有关 SQL 名称的解释, 请参阅 SQL 名称。

还可以指定对齐方式、隐式和显式强制转换以及任何支持的 UDR:

```
mdata.setSQLName("circle2");  
mdata.setLength(24);  
mdata.setAlignment(UDTMetaData.EIGHT_BYTE)  
mdata.setJarFileSQLName("circle2_jar");  
mdata.setUDR(areamethod, "area");
```

```
mdata.setSupportUDR(input, "input", UDTMetaData.INPUT)
mdata.setSupportUDR(output, "output", UDTMetaData.OUTPUT)
mdata.SetImplicitCast(com.gbasedbt.lang.IfxTypes.IFX_TYPE_
    LVARCHAR, "input");
mdata.SetExplicitCast(com.gbasedbt.lang.IfxTypes.IFX_TYPE_
    LVARCHAR, "output");
```

7. 如果需要，请指定驱动程序应将 JAR 文件放在数据库服务器文件系统中的路径名称：

```
String pathname =
"/work/srv93/examples";udtmgr.setJarFileTmpPath(pathname);
```

请确保服务器文件系统中存在此路径。有关更多信息，请参阅指定 JAR 文件临时路径。

8. 创建不透明类型：

```
udtmgr.createUDT(mdata, "Circle2.jar", "Circle2", 0);
```

有关从现有代码创建不透明类型的其它信息，请参阅从现有代码创建不透明类型。

有关使用上述步骤创建不透明类型的完整代码示例，请参阅使用 UDTManager 从现有的 Java 类创建不透明类型。

创建不透明类型，无需现有 Java 类

无需现有 Java™ 类创建不透明类型：

1. 在数据库服务器上创建一个缺省 sbpace 来保存包含不透明类型代码的 JAR 文件。

有关创建 sbpace 的信息，请参阅您的数据库服务器的《GBase 8s 管理员指南》和 J/Foundation 开发者指南。

2. 打开一个 JDBC 连接。

确保数据库对象与连接对象相关联。有关创建具有数据库对象的连接的详细信息，请参阅连接至数据库。

3. 安装 UDTManager 对象和 UDTMetaData 对象：

```
UDTManager udtmgr = new UDTManager(connection);
UDTMetaData mdata = new UDTMetaData();
```

4. 通过调用 UDTMetaData 对象中的方法设置不透明类型的属性：

```
mdata.setSQLName("acircle");
mdata.setLength(24);
mdata.setFieldCount(3);
mdata.setFieldName(1, "x");
```

```
mdata.setFieldName(2, "y");
mdata.setFieldName(3, "radius");
mdata.setFieldType
    (1,com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
mdata.setFieldType
    (2,com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
mdata.setFieldType
    (3,com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
mdata.setJarFileSQLName("ACircleJar");
```

有关不透明类型的设置属性的更多信息，请参 指定不透明类型的属性。

5. 创建 Java 文件、类文件和 JAR 文件：

```
mdata.keepJavaFile(true);
String classname = udtmgr.createUDTClass(mdata);
String jarfilename = udtmgr.createJar(mdata, new String[] {classname
+ ".class"});
```

有关更多信息，请参阅创建 JAR 和类文件。

6. 如果需要，请指定驱动程序应将 JAR 文件放在数据库服务器文件系统中的路径名称：

```
String pathname =
"/work/srv93/examples";udtmgr.setJarFileTmpPath(pathname);
```

请确保此路径在服务器文件系统中存在。有关更多信息，请参阅指定 JAR 文件临时路径。

7. 将类定义发送到数据库服务器：

```
udtmgr.createUDT(mdata, jarfilename, classname, 0);
```

有关更多信息，请参阅将类定义发送到数据库服务器。

有关使用上述步骤创建不透明类型的完整代码示例，请参阅不需现有的 Java 类创建不透明类型。

6.6.4 创建 UDR

以下主题显示如何从 Java™ 类创建 UDR。

创建 UDR：

1. 编写要注册为 UDR 的具有一个或多个静态方法的 Java 类。

有关更多信息，请参阅Java 类的要求。

2. 在数据库服务器上创建一个 sbpace 保存包含此 UDR 代码的 JAR 文件。

有关创建 sbspace 的更多信息，请参阅您的数据库服务器的 GBase 8s 管理员指南和 J/Foundation 开发者指南。

3. 打开一个 JDBC 连接。

请确保连接对象具有与其相关联的数据库对象。有关详细信息，请参阅连接至数据库。

4. 安装 UDRManager 对象和 UDRMetaData 对象：

```
UDRManager udrmgr = new UDRManager(myConn);
UDRMetaData mdata = new UDRMetaData();
```

5. 创建 java.lang.Reflect.Method 对象，以便静态方法可注册为 UDR。

在以下示例中，method1 是代表 Group1 Java 类的 udr1(string, string) 方法的实例；method2 是代表 Group1 Java 类的 udr2(Integer, String, String)方法的实例：

```
Class gp1 = Class.forName("Group1");
Method method1 = gp1.getMethod("udr1",
    new Class[]{String.class, String.class});
Method method2 = gp1.getMethod("udr2",
    new Class[]{Integer.class, String.class, String.class});
```

6. 指定要注册为 UDR 的方法。

第二个参数指定 UDR 的 SQL 名称：

```
mdata.setUDR(method1, "group1_udr1");
mdata.setUDR(method2, "group1_udr2");
```

有关更多信息，请参阅创建 UDR。

7. 指定 JAR 文件 SQL 名：

```
mdata.setJarFileSQLName("group1_jar");
```

8. 如果需要，请指定驱动程序应将 JAR 文件放在数据库服务器文件系统中的路径名称：

```
String pathname = "/work/srv93/examples";
udrmgr.setJarFileTmpPath(pathname);
```

请确保数据服务器文件系统中存在此路径。有关更多信息，请参阅指定 JAR 文件临时路径。

9. 在数据库服务器中安装 UDR：

```
udrmgr.createUDRs(mdata, "Group1.jar", "Group1", 0);
```

有关更多信息，请参阅创建 UDR。

有关创建 UDR 的完整示例代码，请参阅使用 UDRManager 创建 UDR。

6.6.5 Java 类的要求

要转换为不透明类型，Java™ 类必须满足以下条件：

- 类必须实现 `java.sql.SQLData` 接口。有关示例，请参阅示例。
- 如果类包含其它不透明类型，则必须以类似的方式实现附加的不透明类型，并且必须将附加的 `.class` 文件打包为与原始不透明类型相同的 JAR 文件的一部分。
- 如果类包含 `DISTINCT` 类型，则该类可以为 `DISTINCT` 类型实现 `SQLData` 接口，或者让驱动程序将 `DISTINCT` 类型映射到基础类型。有关更多信息，请参阅 `distinct` 数据类型。
- 该类不能包含复杂类型。
- 如果您从现有的 Java 类创建不透明类型，并在数据库服务器中使用缺省支持的函数，则必须将 `SQLData.readSQL()` 和 `SQLData.writeSQL()` 中的 `SQLInput` 和 `SQLOutput` 流强制转换为 `IfmxUDTSQLInput` 和 `IfmxUDTSQLOutput`。

有关如何执行此操作的示例代码，请参阅使用缺省的支持函数创建不透明类型。

- 所有不透明类型的 Java 方法必须与定义不透明类型的类在相同的 `.java` 文件中。

UDR 的其它要求如下所示：

- 所有注册为 UDR 的类方法必须是静态的。
- 方法参数类型和返回类型必须是有效的 Java 数据类型。
- 这些方法可以使用 JDK 中包含的所有基本的非图形类 Java 包，例如 `java.util`、`java.io`、`java.net`、`java.rmi`、`java.sql` 等等。
- 方法参数和返回类型的数据类型必须符合 UDT manager 和 UDR manager 的数据类型映射中显示的数据类型映射表。
- 不支持以下 SQL 参数或返回类型：
 - `MONEY`
 - 具有除 `hour to second` 或 `year to fraction(5)` 之外的限定符的 `DATETIME`
 - 具有除 `year to month` 或 `day to fraction(5)` 之外限定符的 `INTERVAL`
 - 不在方法参数和返回类型的映射表中显示的任何数据类型；对于这些表，请参阅 UDT manager 和 UDR manager 的数据类型映射。

6.6.6 SQL 名称

UDTMetaData 类中的某些方法为不透明类型或包含不透明类型或 UDR 代码的 JAR 文件设置 SQL 名称。SQL 名称是在 SQL 语句中引用的对象的名称。例如，假设您的应用程序发出以下调用：

```
mdata.setSQLName("circle2");
```

SQL 语句中使用的名称如下所示：

```
CREATE TABLE tab (c circle2);
```

同样，假定应用程序将 JAR 文件名设置如下所示：

```
mdata.setJarFileSQLName("circle2_jar");
```

按照以下方式在 SQL 中引用 JAR 文件名称：

```
CREATE FUNCTION circle2_output (...)  
RETURNS circle2  
EXTERNAL NAME  
'circle2_jar: circle2.fromString (...)'  
LANGUAGE JAVA  
NOT VARIANT  
END FUNCTION;
```

重要： SQL 名称没有缺省值。请使用 `setSQLName()` 或 `setJarFileSQLName()` 方法指定名称，否则会抛出一个 SQL 异常。

6.6.7 指定不透明类型的属性

下列主题提供在 Java™ 类不存在时有关创建不透明类型的附加信息。有关从现有 Java 类创建不透明类型的详细信息，请参阅 [从现有代码创建不透明类型](#)。

使用 **UDTMetaData** 类中的方法，可以为新的不透明类型指定属性。这些设置适用于新的不透明类型；从现有文件创建不透明类型，请参阅 [从现有代码创建不透明类型](#)。

可以设置以下属性：

- 内部结构中定义不透明类型的字段数
- 其它属性，例如内部结构中定义不透明类型的每个字段的数据类型、名称和精度
- 不透明类型的长度
- 不透明类型的对齐方式
- 不透明类型和 JAR 文件的 SQL 名称
- 已生成的 Java 类的名称
- 是否保留生成的 .java 文件

指定字段计数

setFieldCount() 方法指定内部结构中定义不透明类型的字段计数:

```
public void setFieldCount(int fieldCount) throws SQLException
```

指定其它字段属性

以下方法设置内部数据结构中的字段的其它属性:

```
public void setFieldName (int field, String name) throws SQLException
public void setFieldType (int field, int ifxtype) throws SQLException
public void setFieldTypeName(int field, String sqltypename) throws SQLException
public void setFieldLength(int field, int length) throws SQLException
```

field 参数指示驱动程序应设置或获取属性的字段。第一个字段为 1; 第二个字段为 2, 依次类推。

使用 setFieldName() 指定的名称出现在 Java™ 类文件中。以下示例将第一个字段名称设置为 IMAGE。

```
mdata.setFieldName(1, "IMAGE");
```

setFieldType() 方法指示使用 com.gbasedbt.lang.IfxTypes 文件中的常量设置字段的数据类型。有关更多信息, 请参阅映射字段类型。以下示例指定第三个字段中的值为 CHAR 数据类型:

```
mdata.setFieldType(3, com.gbasedbt.lang.IfxTypes.IFX_TYPE_CHAR);
```

setFieldTypeName() 方法设置使用 SQL 数据类型名称的字段的数据类型:

```
mdata.setFieldTypeName(1, "IMAGE_UDT");
```

此方法仅对 opaque 和 distinct 类型有效; 对于其它类型, 驱动程序会忽略此信息。

length 参数具有以下含义, 它取决于字段的数据类型:

字符类型

字符中的最大长度

DATETIME

编码的长度

INTERVAL

编码的长度

其它数据类型或不指定类型

驱动程序忽略此信息

编码长度的可能值为 JDBC 2.20 规范中的值: 小时到秒; 一年到两年; 年份 (1), 年份 (2) 到年份 (5)。

以下示例指定不透明类型中的第三个字段 (VARCHAR) 不能存储多于 24 个字符:

```
mdata.setFieldLength(3, 24);
```

指定长度

setLength() 方法指定不透明类型的总长度：

```
public void setLength(int length) throws SQLException
```

如果您正在从现有的 Java™ 类创建不透明类型并不指定长度，则驱动程序创建可变长度的不透明类型。如果您从不具有现有的 Java 类创建不透明类型，则必须指定长度；在此情况下，UDT Manager 只创建固定长度的不透明类型。

指定对齐方式

setAlignment() 方法指定不透明类型的对齐方式：

```
public void setAlignment(int alignment)
```

alignment 参数是下一节中显示的对齐值之一。如果未指定对齐方式，则数据库服务器设置不透明类型 4 字节对齐。

对齐值

对齐值如下表所示。

值	常量	结构始于	边界对齐
1	SINGLE_BYTE	1 个字节的数量	单个字节
2	TWO_BYTE	2 个字节的数量（例如 SMALLINT）	2 个字节
4	FOUR_BYTE	4 个字节的数量（例如 FLOAT 或 UNSIGNED INT）	4 个字节
8	EIGHT_BYTE	8 个字节的数量	8 个字节

指定 SQL 名称

使用 setSQLName() 和 setJarFileSQLName()方法指定 SQL 名称：

```
public void setSQLName(String name) throws SQLException
public void setJarFileSQLName(String name) throws SQLException
```

缺省情况下，驱动程序使用通过 setSQLName() 方法设置的名称作为当调用 UDTManager.createUDTCclass() 和 UDTManager.createJar() 方法时生成的 JAR 文件和 Java™ 类的文件名称。例如，如果调用 setSQLName("circle") 然后调用 createUDTCclass() 和 createJar()，则生成的类文件名将会是 circle.class，JAR 文件名将会是 circle.jar。可以通过调用 setClassName() 方法指定 Java 类文件名而不是缺省值。

JAR 文件 SQL 名称是驱动程序用于注册 UDR 的 SQL CREATE FUNCTION 语句中引用的名称。

重要：JAR 文件 SQL 名是 SQL 语句中 JAR 文件的名称; 它与 JAR 文件的内容没有关系。

指定 Java 类名称

使用 `setClassName()` 指定 Java™ 类名称:

```
public void setClassName(String name) throws SQLException
```

如果未使用 `setClassName()` 设置类名称, 则驱动程序使用不透明类型的 SQL 名称 (通过 `setSQLName()` 设置) 和 `createUDTClass()` 方法生成的 .class 文件的名称作为 Java™ 类的名称。

指定 Java 源文件保留

使用 `keepJavaFile()` 指定是否保留 .java 源文件:

```
public void keepJavaFile(boolean value)
```

`value` 值指示 `createUDTClass()` 方法是否应该保留它在为新的不透明类型创建 Java™ 类文件时生成的 .java 文件。缺省为移除此文件。以下示例保留 .java 文件:

```
mdata.keepJavaFile(true);
```

6.6.8 创建 JAR 和类文件

一旦您使用 `UDTMetaData` 方法指定了不透明类型的属性, 您可以按照以下顺序使用类中的方法创建不透明类型及其类和 JAR 文件:

1. 实例化 `UDTManager` 对象。

如以下所示建立连接:

```
public UDTManager(Connection conn) throws SQLException
```

2. 使用 `createUDTClass()` 方法创建 .class 和 .java 文件。
3. 使用 `createJar()` 方法创建 .jar 文件。
4. 使用 `createUDT()` 方法创建不透明类型。

创建 .class 和 .java 文件

`createUDTClass()` 方法具有以下签名:

```
public String createUDTClass(UDTMetaData mdata) throws SQLException
```

`createUDTClass()` 方法导致驱动程序为应用程序执行所有以下操作:

1. 使用 `UDTMetaData.setClassName()` 方法中指定的名称创建一个 Java™ 类
如果未指定类名称, 则驱动程序使用 `UDTMetaData.setSQLName()` 方法中指定的名称。
2. 将 Java 类代码放到 .java 文件中, 然后编译此文件为 .class 文件。
3. 将新创建的名称返回到应用程序。

如果通过调用 `UDTMetaData.keepJavaFile()` 方法指定 `TRUE`, 则驱动程序保留生成的 .java 文件。缺省操作为删除此 .java 文件。

应用程序调用 `createUDTClass()` 方法仅创建定义不透明类型的新 `.class` 和 `.java` 文件，而不从现有文件生成不透明类型。

创建 `.jar` 文件

`createJar()` 方法编译您在 `classnames` 列表中指定的类。列表中的文件必须具有 `.class` 扩展名。

```
public String createJar(UDTMetaData mdata, String[] classnames)
    throws SQLException;
```

驱动程序创建名为 `sqlname.jar` 的 JAR 文件（其中 `sqlname` 是您通过调用 `UDTMetaData.setSQLName()` 指定的名称）并将文件名返回到应用程序。

6.6.9 将类定义发送到数据库服务器

创建 JAR 文件之后，使用 `UDTManager.createUDT()` 方法通过将类定义发送到数据库服务器来创建不透明类型：

```
public void createUDT(UDTMetaData mdata, String jarfile, String
    classname, int deploy) throws SQLException;
```

`jarfile` 参数是包含不透明类型的类定义的 JAR（`.jar`）文件的路径名。缺省情况下，`java.io` 包中的类编译相对路径名而不是系统属性 `user.dir` 命名的当前用户目录；它通常是 Java™ Virtual Machine 调用的目录。如果您使用绝对路径名，则文件名称必须包含在您的 `CLASSPATH` 设置中。

`classname` 参数是实现不透明类型的类的名称。

如果应用程序没有调用 `setClassName()`，则类名称使用缺省的不透明类型的 SQL 名称。您可以通过调用 `UDTMetaData.setSQLName()` 方法指定 SQL 名称。

重要： 如果应用程序在事务中调用 `createUDT()` 或者数据库是 ANSI 或启用日志记录，则会应用一些扩展的规则。有关更多信息，请参阅在事务中执行。

指定部署描述符操作

在 `UDTManager` 和 `UDRManager` 方法中，`deploy` 参数指示在 JAR 文件中存在部署描述符时 `install_actions` 是否执行。`undeploy` 参数指示是否执行 `remove_actions`。

0

执行 `install_actions` 或 `remove_actions`。

非零

不执行 `install_actions` 或 `remove_actions`。

部署描述符允许您包含用于在 JAR 文件中创建和删除 UDR 的 SQL 语句。有关部署描述符的更多信息，请参阅 J/Foundation 开发者指南 和 SQLJ 规范。

指定 JAR 文件临时路径

当驱动程序为不透明类型或 UDR 发送 JAR 文件时，它将文件缺省放在 /tmp (UNIX™ 上) 或 C:\temp (Windows™ 上)。可以通过调用 UDTManager 或 UDRManager 类中的 setJarTmpPath() 方法，指定替代路径：

```
public void setJarTmpPath(String path) throws SQLException
```

您可以在调用 createUDT() 或 createUDR()、UDTManager 或 UDRManager 对象之前，随时调用此方法。path 参数必须是绝对路径名。您必须确保路径在服务器文件系统上存在。

6.6.10 从现有代码创建不透明类型

前面的主题描述了用于创建不包含现有 Java™ 类的新的不透明类型的方法。从现有 Java 代码创建不透明类型时，请指定不透明类型中包含的 SQL 名称、JAR 文件 SQL 名称、支持的 UDR（如果有）和任何其他不支持的 UDR。（有关 SQL 名称的解释，请参阅 SQL 名称。）还可以指定长度、对齐方式、隐式和显式强制转换。

要从现有的代码创建不透明类型。请使用以下方法：

- UDTMetaData.setSQLName() 指定 SQL 语句中引用的不透明类型的 SQL 名称
- 对于不透明类型中支持的 UDR 使用 UDTMetaData.setSupportUDR()
支持的 UDR 为输入/输出、发送/接收等等。
- 对于不透明类型中不支持的 UDR 使用 UDTMetaData.setUDR()
- UDTMetaData.setJarFileSQLName() 指定 JAR 文件的 SQL 名称
- UDTMetaData.setImplicitCast() 或 UDTMetaData.setExplicitCast() 指定强制转换
- 如果不透明类型是固定长度的，使用 UDTMetaData.setLength()（驱动程序缺省为可变长度）
- UDTMetaData.setAlignment() 指定不透明类型所对齐的字节边界（仅当您不希望数据库服务器默认 4 字节边界时）
- UDTManager.createJar() 创建 JAR（.jar）文件（如果没有）
- UDTManager.createUDT() 创建不透明类型

另外，setXXXCast()、setSupportUDR() 和 setUDR() 方法仅用于从现有代码创建不透明类型：

```
public void setImplicitCast(int ifxtype, String methodsqlname)
    throws SQLException
public void setExplicitCast(int ifxtype, String methodsqlname)
    throws SQLException
public void setSupportUDR(Method method, String sqlname, int type)
    throws SQLException
public void setUDR(Method method, String sqlname)
```

throws SQLException

setXXxCast() 方法

setXXxCast() 方法指定显式或隐式将不透明类型转换为指定的数据类型。

ifxtype 参数是类 com.gbasedbt.lang.IfxTypes 的类型代码。数据库服务器中的 ifxtype 参数和 SQL 类型之间的数据类型映射在映射转换类型中进行了详细说明。methodsqlname 参数是实现此转换的 Java™ 方法的 SQL 名称。

以下示例使用 SQL 名称 circle2_input 设置由 Java 方法实现的隐式转换：

```
setImplicitCast(com.gbasedbt.lang.IfxTypes.IFX_TYPE_LVARCHAR,  
"circle2_input");
```

以下示例使用 SQL 名称 circle_output 设置由 Java 方法实现的显式转换：

```
setExplicitCast(com.gbasedbt.lang.IfxTypes.IFX_TYPE_LVARCHAR,  
"circle2_output");
```

以下示例设置一个显式转换，将 circle2 不透明类型转换为整数：

```
setExplicitCast(com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT,  
"circle2_to_int");
```

setSupportUDR() 和 setUDR() 方法

setSupportUDR() 方法在现有 Java™ 类中指定 Java 方法，该类将被注册为不透明类型支持的 UDR。

method 参数指定 java.lang.reflect.Method 中的一个对象，将其注册为数据库服务中不透明类型的 Java 支持 UDR。支持的 UDR 为输入、输出、发送、接收等等。（有关更多信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南。）

sqlname 参数指定方法的 SQL 名。有关更多信息，请参阅 SQL 名称。

type 参数指定支持的 UDR 的类型。如下所示：

```
UDTMetaData.INPUT  
UDTMetaData.OUTPUT  
UDTMetaData.SEND  
UDTMetaData.RECEIVE  
UDTMetaData.IMPORT  
UDTMetaData.EXPORT  
UDTMetaData.BINARYIMPORT  
UDTMetaData.BINARYEXPORT
```

有关如何从现有代码创建不透明类型的操作步骤，请参阅从现有 Java 类创建不透明类型。

提示：不必注册 SQLData 接口中方法。例如，您不需注册 SQLData.getSQLTypeName()、SQLData.readSQL() 或 SQLData.writeSQL()。

要指定其他 UDR，使用创建 UDR 中描述的 setUDR()。

6.6.11 移除不透明类型和 JAR 文件

可以使用以下方法移除不透明类型及其 JAR 文件：

```
public static void removeUDT(String sqlname) throws SQLException
public static void removeJar(String jarfilesqlname, int undeploy)
throws SQLException
```

removeUDT() 方法从数据库服务器移除不透明类型，以及所有它的强制转型和 UDR。它不会移除 JAR 文件本身，因为其它不透明类型或 UDR 可能正在使用相同的 JAR 文件。

重要： 如果应用程序在事务中调用 removeUDT() 或您的数据库是 ANSI 或启用了日志记录，则要应用一些其它规则。有关更多信息，请参阅在事务中执行。

removeJar() 方法从系统目录移除 JAR 文件。jarfilesqlname 参数是使用 setJarFileSQLName() 方法指定的名称。

对于 undeploy 参数，请参阅指定部署描述符操作。

重要： 在调用 removeJar() 之前，您必须首先移除所有与 JAR 文件关联的函数和过程。否则，数据库服务器移除文件失败。

6.6.12 创建 UDR

使用 UDR Manager 在数据库服务器中创建 UDR，包括：

- 编码 UDR 并将代码打包到 JAR 文件中
有关编码 UDR 的信息，请参阅 J/Foundation 开发者指南。
- 在数据库服务器中创建缺省的 sbospace 来保存包含 UDR 代码的 JAR 文件
有关创建 sbospace 的信息，请参阅您的数据库服务器的《GBase 8s 管理员指南》和 J/Foundation 开发者指南。
- 调用 UDRMetaData 类中的方法指定 GBase 8s JDBC Driver 在数据库服务器中注册 UDR 所必需的信息
- 如果需要。请指定驱动程序应将 JAR 文件放在数据库服务器文件系统中的路径名
- 在服务器中安装 UDR

为 C 语言不透明类型创建的 UDR 是不受支持的；不透明类型必须是 Java™ 的。

要指定注册的驱动程序的 UDR，请使用 UDRMetaData 中的这个方法：

```
public void setUDR(Method method, String sqlname) throws SQLException
```

method 参数指定要在数据库服务器中注册为 Java UDR 的 java.lang.Reflect.Method 中的对象。sqlname 参数是用于 SQL 语句中方法的名称。

一旦您指定了要注册的 UDR，则可以使用 `UDRMetaData.setJarFileSQLName()` 设置 JAR 文件 SQL 名称，然后使用 `UDRManager.createUDRs()` 方法在数据库服务器中安装此 UDR，如下所示：

```
public void createUDRs(UDRMetaData mdata, String jarfile, String classname, int
deploy) throws SQLException
```

`jarfile` 参数是包含 Java 方法定义的客户端 JAR 文件的绝对或相对路径名。如果使用绝对路径名，则 JAR 文件名必须包含在您 `CLASSPATH` 设置中。

`classname` 参数是包含您要在数据库服务器中注册为 UDR 的 Java 类的名称。有关准备 Java 方法的要求，在 1 中描述。

对于 `deploy` 参数，请参阅指定部署描述符操作。

`createUDRs()` 方法使驱动程序为您的应用程序执行以下所有步骤：

1. 获取第一个参数指定的 JAR 文件。
2. 将 JAR 文件从客户端本地区域传送到服务器本地区域。
3. 注册 `UDRMetaData` 对象中指定的 UDR（通过调用一个或多个 `UDRMetaData.setUDR()`）。
4. 在服务器上安装 JAR 文件并创建 UDR。

执行 `createUDRs()` 后，您的应用程序就可以在 SQL 语句中使用 UDR。

重要：如果应用程序在事务中调用 `createUDRs()`，或者您的数据库为 ANSI 或启用了日志记录，会应用一些其它规则。有关更多信息，请参阅在事务中执行。

6.6.13 移除 UDR 和 JAR 文件

可用使用以下方法移除 UDR：

```
public void removeUDR(String sqlname) throws SQLException
    public void removeJar(String jarfilesqlname, int undeploy) throws
        SQLException
```

提示：`removeUDR()` 方法从服务器移除 UDR 但是不移除 JAR 文件，因为其它不透明类型或 UDR 会使用相同的 JAR 文件。

`removeJar()` 方法在移除不透明类型和 JAR 文件中有所描述。

移除重载的 UDRs

要移除重载的 UDR，请使用具有附加参数的 `removeUDR()` 方法：

```
public void removeUDR(String sqlname, Class[] methodparams) throws
        SQLException
```


methodparams 参数指定 UDR 中每个参数的数据类型。指定 NULL 指示没有参数。例如，假设一个名为 print() 的 UDR 被重载了两个额外的方法签名。

Java™ 方法签名	对应的 SQL 名
void print()	print1
void print(String x, String y, int r)	print2
void print(int a, int b)	print3

移除所有三个 UDR 的代码为：

```
udrmgr.removeUDR("print1", null );
udrmgr.removeUDR("print2",
new Class[] {String.class, String.class, int.class} );
udrmgr.removeUDR("print3", new Class[] {int.class, int.class} );
```

6. 6. 14 获取有关不透明类型和 UDR 的信息

UDTMetaData 和 UDRMetaData 类中的许多 setXXX() 方法具有相应的 getXXX() 方法，用于获取现有不透明类型和 UDR 的特性。

UDTMetaData 类中的 getXXX() 方法

下表总结了 UDTMetaData 类中可用的 getXXX() 方法。对于 field 参数，1 表示内部数据结构中第一个字段，2 是第二个字段，依次类推。有关 SQL 名称的详细信息，请参阅 SQL 名称。

获取的信息	方法签名	其它信息
内部数据结构中的字段计数	public int getFieldCount()	如果没有字段，则返回 0
内部数据结构中的字段名称	public String getFieldName int field) throws SQLException	如果名称不存在，则返回 NULL
内部数据结构中的字段数据类型代码	public int getFieldType (int field) throws SQLException	数据类型代码来自类 com.gbasedbt.lang.IfTypes。如果数据类型不存在，则返回 -1

获取的信息	方法签名	其它信息
内部数据结构中字段的 数据类型名称	public String getFieldTypeName (int field) throws SQLException	如果名称不存在，则返回 NULL
对于字符类型：字段中的 最大字符数；对于 date-time 或 interval 类型： 编码限定符	public int getFieldLength (int field) throws SQLException	如果未设置长度，则返回 -1
不透明类型的 SQL 名称	public String getSQLName()	如果未设置名称，则返回 NULL
JAR 文件的 SQL 名称	public String getJarFileSQLName()	如果未设置名称，则返回 NULL
不透明类型的 Java™ 类 的名称	public String getClassName()	如果未通过 setClassName() 设置类名，则返回sqlname（缺省值）。 如果未通过 setSQLName() 设置 SQL 名称，则返回 NULL
固定长度的不透明类型 的长度	public int getLength()	如果未设置长度，则返回 -1
不透明类型的对齐方式	public int getAlignment()	如果未设置对齐方式，则返回 -1 有关对齐方式代码，请参阅 对齐值 。
已通 过 setSupportUDR() 指定为支持 UDR 的方法 对象组	public Method[] getSupportUDRs()	有关支持的 UDR 的详细信息，请参阅 从现有代码创建不透明类型中 setSupportUDR() 的描述。如果未指定支持 UDR，则返回 NULL
通 过 setSupportUDR() 指定为支持 UDR 的	public String getSupportUDRSQLName (Methodmethod) throws	如果未设置名称，则返回 NULL

获取的信息	方法签名	其它信息
Java 方法的 SQL 名称	SQLException	

UDRMetaData 类中的 getXXX() 方法

要获取有关 UDR 的信息，请使用下表中的方法。

获取的信息	方法签名	其它信息
指定为不透明类型的 UDR 的 <code>java.lang.Method.Reflect</code> 方法组	<code>public Method[] getUDRs()</code>	调用 <code>UDTMetaData.setUDR()</code> 为不透明类型指定 UDR。如果没有指定 UDR 则返回 NULL。
Java™ 方法的 SQL 名称	<code>public String getUDRSQLName(Method method) throws SQLException</code>	如果没有为 UDR 方法对象指定 SQL 名称，则返回 NULL

6.6.15 在事务中执行

如果您的数据库是 ANSI 或启用了日志记录，并且应用程序尚未在事务中，则驱动程序执行 SQL 语句在事务中的服务器上创建不透明类型和 UDR。这意味着所有的步骤都会成功，否则都将失败。如果在任何一点创建不透明类型或 UDR 失败，则驱动程序回滚事务并引发 SQLException。

当发出 `UDTManager.createUDT()` 或 `UDRManager.createUDRs()` 调用时，如果事务已在应用程序中，则在现有的事务中执行 SQL 语句。这意味着如果驱动程序在创建不透明类型或 UDR 期间返回一个 `SQLException`，则您的应用程序必须回滚事务以确保数据库的完整性。否则，不透明类型转换的部分或 UDR 可以保留在数据库中。

6.7 示例

本节的剩余部分包含创建和使用不透明类型和 UDR 的示例。

前四个示例在 `demo/udt-distinct` 目录中随 JDBC 驱动程序软件一起发布；最后两个示例在 `demo/tools/udtudrmgr` 目录下。请参阅每个目录中的 README 文件以获取文件的说明。

6.7.1 类定义

以下示例中的 **charattrUDT** 是 C 不透明类型的类，它必须实现 **SQLData** 接口：

```
import java.sql.*;
import com.gbasedbt.jdbc.*;

/*
 * C struct of charattr_udt:
 *
 * typedef struct charattr_type
 * {
 *     char          chr1[4+1];
 *     mi_boolean     bold;          // mi_boolean (1 byte)
 *     mi_smallint    fontsize;     // mi_smallint (2 bytes)
 * }
 * charattr;
 *
 * typedef charattr charattr_udt;
 *
 */

public class charattrUDT implements SQLData
{
    private String sql_type = "charattr_udt";
    // an ASCII character/a multibyte character, and is null-terminated.
    public String chr1;
    // Is the character in boldface?
    public boolean bold;
    // font size of the character
    public short fontsize;

    public charattrUDT() { }

    public charattrUDT(String chr1, boolean bold, short fontsize)
    {
        this.chr1 = chr1;
        this.bold = bold;
        this.fontsize = fontsize;
    }
}
```

```
public String getSQLTypeName()
{
    return sql_type;
}

// reads a stream of data values and builds a Java object
public void readSQL(SQLInput stream, String type) throws SQLException
{
    sql_type = type;
    chr1 = ((IfmxUDTSQLInput)stream).readString(5);
    bold = stream.readBoolean();
    fontsize = stream.readShort();
}

// writes a sequence of values from a Java object to a stream
public void writeSQL(SQLOutput stream) throws SQLException
{
    ((IfmxUDTSQLOutput)stream).writeString(chr1, 5);
    stream.writeBoolean(bold);
    stream.writeShort(fontsize);
}

// overrides Object.equals()
public boolean equals(Object b)
{
    return (chr1.equals(((charattrUDT)b).chr1) &&
        bold == ((charattrUDT)b).bold &&
        fontsize == ((charattrUDT)b).fontsize);
}

public String toString()
{
    return "chr1=" + chr1 + " bold=" + bold + " fontsize=" + fontsize;
}
}
```

在 JDBC 应用程序中，自定义类型映射必须将 SQL 类型名 charattr_udt 映射到 charattrUDT 类：

```
java.util.Map customtypemap = conn.getTypeMap();
if (customtypemap == null)
```

```
{
    System.out.println("\n***ERROR: typemap is null!");
    return;
}

customtypemap.put("charattr_udt", Class.forName("charattrUDT"));
```

6.7.2 插入数据

可以插入不透明类型的原始类型或它的转换类型。以下示例显示如何插入使用原始类型的不透明数据：

```
String s = "insert into charattr_tab (int_col, charattr_col)
          values (?, ?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);
...
charattrUDT charattr = new charattrUDT();
charattr.chr1 = "a";
charattr.bold = true;
charattr.fontsize = (short)1;

pstmt.setInt(1, 1);
System.out.println("setInt...ok");

pstmt.setObject(2, charattr);
System.out.println("setObject(charattrUDT)...ok");

pstmt.executeUpdate();
```

如果定义了强制转换函数，并且想要将数据作为转换类型而不是原始类型插入，则必须调用与转换类型相对应的 `setXXX()` 方法。例如，如果已经将 `CHAR` 或 `LVARCHAR` 函数转换为 `charattrUDT` 列，则可以使用 `setString()` 方法插入数据，如下所示：

```
// Insert into UDT column using setString(int,String) and Java
String object.
String s =
"insert into charattr_tab " +
"(decimal_col, date_col, charattr_col, float_col) " +
"values (?, ?, ?, ?)";
```

```
writeOutputFile(s);
PreparedStatement pstmt = myConn.prepareStatement(s);

...
String strObj = "(A, f, 18)";
pstmt.setString(3, strObj);
...
```

6.7.3 检索数据

要检索 GBase 8s 不透明类型，必须使用 `ResultSet.getObject()`。GBase 8s JDBC Driver 根据您提供的自定义类型将数据转换为 Java™ 对象。使用前面示例的 **charattrUDT** 类型，可以获取此不透明数据，如下所示：

```
String s = "select int_col, charattr_col from charattr_tab order by 1";
System.out.println(s);

Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(s);
System.out.println("execute...ok");

System.out.println("Fetching data ...");
int curRow = 0;
while (rs.next())
{
    curRow++;
    System.out.println("currentrow=" + curRow + " : ");

    int intret = rs.getInt("int_col");
    System.out.println("    int_col        " + intret);

    charattrUDT charattrret = (charattrUDT)rs.getObject("charattr_col");
    System.out.print("    charattr_col  ");
    if (curRow == 2 || curRow == 6)
    {
        if (rs.isNull())
            System.out.println("<null>");
        else
```

```
        System.out.println("****ERROR: " + charattrret);
    }
    else
        System.out.println(charattrret+"");
    } //while

    System.out.println("total rows expected: " + curRow);
    stmt.close();
```

6.7.4 不透明类型中的智能大对象

尽管您最可能使用 GBase 8s 扩展类在不透明类型上下文之外的数据库服务器上创建大对象，但是智能大对象仍可以是不透明类型中的数据成员，

有关智能大对象的更多信息，请参阅智能大对象数据类型。

在不透明类型中大对象存储在 **IfxLocator** 对象中；在内部定义不透明类型的 C 结构中，大对象通过 **MI_LO_HANDLE** 类型的定位指针引用。该对象使用 **IfxSmartBlob** 类提供的方法创建，并且从这些方法获得的大对象句柄成为不透明类型中的数据成员。**BLOB** 和 **CLOB** 对象都使用相同的大对象句柄，如下例所示：

```
import java.sql.*;
import com.gbasedbt.jdbc.*;
/*
 * C struct of large_bin_udt:
 *
 * typedef struct LARGE_BIN_TYPE
 * {
 *     MI_LO_HANDLE lb_handle;    // handle to large object (72 bytes)
 * }
 * large_bin_udt;
 *
 */
public class largebinUDT implements SQLData
{
    private String sql_type = "large_bin_udt";
    public Clob lb_handle;

    public largebinUDT() { }
```



```
public largebinUDT(Clob clob)
{
    lb_handle = clob;
}

public String getSQLTypeName()
{
    return sql_type;
}

// reads a stream of data values and builds a Java object
public void readSQL(SQLInput stream, String type) throws SQLException
{
    sql_type = type;
    lb_handle = stream.readClob();
}

// writes a sequence of values from a Java object to a stream
public void writeSQL(SQLOutput stream) throws SQLException
{
    stream.writeClob(lb_handle);
}
}
```

在 JDBC 应用程序中，使用 IfxSmartBlob 类提供的方法创建 MI_LO_HANDLE 对象：

```
String s = "insert into largebin_tab (int_col, largebin_col, lvc_col) " +
    "values (?, ?, ?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);

...

String filename = "lbin_in1.dat";
File file = new File(filename);
int fileLength = (int) file.length();
FileInputStream fin = new FileInputStream(file);

IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
System.out.println("create large object descriptor...ok");
```

```
lfxLocator loPtr = new lfxLocator();
lfxSmartBlob smb = new lfxSmartBlob((lfxConnection)conn);
int loFd = smb.lfxLoCreate(loDesc, 8, loPtr);
System.out.println("create large object...ok");

int n = smb.lfxLoWrite(loFd, fin, fileLength);
System.out.println("write file content into large object...ok");

pstmt.setInt(1, 1);
System.out.println("setInt...ok");

// initialize largebin object using the large object created
// above, before doing setObject for the large_bin_udt column.
largebinUDT largebinObj = new largebinUDT();
largebinObj.lb_handle = new lfxCblob(loPtr);
pstmt.setObject(2, largebinObj);
System.out.println("setObject(largebinUDT)...ok");

pstmt.setString(3, "Sydney");
System.out.println("setString...ok");

pstmt.executeUpdate();
System.out.println("execute...ok");

// close/release large object
smb.lfxLoClose(loFd);
System.out.println("close large object...ok");
smb.lfxLoRelease(loPtr);
System.out.println("release large object...ok");
```

有关详细信息，请参阅智能大对象数据类型。

6.7.5 使用 **UDTManager** 从现有的 **Java** 类创建不透明类型

以下示例显示应用程序如何使用 **UDTManager** 和 **UDTMetaData** 类将客户端上（数据库服务器无法访问）的现有 **Java™** 类转换为数据库服务器中的 **SQL** 不透明类型。

使用缺省的支持函数创建不透明类型

以下示例在使用数据库服务器中现有的 Java™ 类和缺省的支持函数创建不透明类型 **Circle**:

```
*/

import java.sql.*;
import com.gbasedbt.jdbc.IfmxUDTSQLInput;
import com.gbasedbt.jdbc.IfmxUDTSQLOutput;

public class Circle implements SQLData
{
    private static double PI = 3.14159;

    double x;          // x coordinate
    double y;          // y coordinate
    double radius;

    private String type = "circle";

    public String getSQLTypeName() { return type; }

    public void readSQL(SQLInput stream, String typeName)
        throws SQLException
    {
        // To be able to use the DEFAULT support functions supplied
        // by the server, you must cast the stream to IfmxUDTSQLInput.
        // (Server requirement)

        IfmxUDTSQLInput in = (IfmxUDTSQLInput) stream;
        x = in.readDouble();
        y = in.readDouble();
        radius = in.readDouble();
    }

    public void writeSQL(SQLOutput stream) throws SQLException
    {
        // To be able to use the DEFAULT support functions supplied
```

```
// by the server, have to cast the stream to IfmxUDTSQLOutput.  
// (Server requirement)  
  
IfmxUDTSQLOutput out = (IfmxUDTSQLOutput) stream;  
out.writeDouble(x);  
out.writeDouble(y);  
out.writeDouble(radius);  
}  
  
public static double area(Circle c)  
{  
    return PI * c.radius * c.radius;  
}  
}
```

不透明类型

下列 JDBC 客户端应用程序将类 `Circle`（在 `Circle.jar` 中打包）作为不透明类型安装在系统目录中。然后，应用程序可以在 SQL 语句中使用不透明类型 **Circle** 作为数据类型：

```
import java.sql.*;  
import java.lang.reflect.*;  
  
public class PlayWithCircle  
{  
    String dbname = "test";  
    String url = null;  
    Connection conn = null;  
  
    public static void main (String args[])  
    {  
        new PlayWithCircle(args);  
    }  
  
    PlayWithCircle(String args[])  
    {  
        System.out.println("-----");  
    }  
}
```

```
System.out.println("- Start - Demo 1");
System.out.println("-----");

// -----
// Getting URL
// -----
if (args.length == 0)
{
    System.out.println("\n***ERROR: connection URL must be provided "
+
                                "in order to run the demo!");

    return;
}
url = args[0];

// -----
// Loading driver
// -----
try
{
    System.out.print("Loading JDBC driver...");
    Class.forName("com.gbasedbt.jdbc.Driver");
    System.out.println("ok");
}
catch (java.lang.ClassNotFoundException e)
{
    System.out.println("\n***ERROR: " + e.getMessage());
    e.printStackTrace();
    return;
}

// -----
// Getting connection
// -----
try
{
    System.out.print("Getting connection...");
```

```
        conn = DriverManager.getConnection(url);
        System.out.println("ok");
    }
    catch (SQLException e)
    {
        System.out.println("URL = " + url + "");
        System.out.println("\n***ERROR: " + e.getMessage());
        e.printStackTrace();
        return;
    }
    System.out.println();

    // -----
    // Setup UDT meta data
    // -----
    Method areamethod = null;
    try
    {
        Class c = Class.forName("Circle");
        areamethod = c.getMethod("area", new Class[] {c});
    }
    catch (ClassNotFoundException e)
    {
        System.out.println("Cannot get Class: " + e.toString());
        return;
    }
    catch (NoSuchMethodException e)
    {
        System.out.println("Cannot get Method: " + e.toString());
        return;
    }

    UDTMetaData mdata = null;
    try
    {
        System.out.print("Setting mdata...");
        mdata = new UDTMetaData();
    }
```

```
        mdata.setSQLName("circle");
        mdata.setLength(24);
        mdata.setAlignment(UDTMetaData.EIGHT_BYTE);
        mdata.setUDR(areamethod, "area");
        mdata.setJarFileSQLName("circle_jar");
        System.out.println("ok");
    }
    catch (SQLException e)
    {
        System.out.println("\n***ERROR: " + e.getMessage());
        return;
    }

    // -----
    // Install the UDT in the database
    // -----
    UDTManager udtmgr = null;
    try
    {
        udtmgr = new UDTManager(conn);

        System.out.println("\ncreateJar()");
        String jarfilename = udtmgr.createJar(mdata,
            new String[] {"Circle.class"}); // jarfilename = circle.jar
        System.out.println("    jarfilename = " + jarfilename);

        System.out.println("\nsetJarTmpPath()");
        udtmgr.setJarTmpPath("/tmp");

        System.out.print("\ncreateUDT()...");
        udtmgr.createUDT(mdata,
            "/vobs/jdbc/demo/tools/udtudrmgr/" + jarfilename, "Circle", 0);
        System.out.println("ok");
    }
    catch (SQLException e)
    {
        System.out.println("\n***ERROR: " + e.getMessage());
    }
```

```
        return;
    }
    System.out.println();

    // -----
    // Now use the UDT
    // -----
    try
    {
        String s = "drop table tab";
        System.out.print(s + "...");
        Statement stmt = conn.createStatement();
        int count = stmt.executeUpdate(s);
        stmt.close();
        System.out.println("ok");
    }
    catch ( SQLException e)
    {
        // -206 The specified table (%s) is not in the database.
        if (e.getErrorCode() != -206)
        {
            System.out.println("\n***ERROR: " + e.getMessage());
            return;
        }
        System.out.println("ok");
    }

    executeUpdate("create table tab (c circle)");

    // test DEFAULT Input function
    executeUpdate("insert into tab values ('10 10 10')");

    // test DEFAULT Output function
    try
    {
        String s = "select c::lvvarchar from tab";
        System.out.println(s);
    }
```



```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(s);
if (rs.next())
{
    String c = rs.getString(1);
    System.out.println("    circle = " + c + "");
}
rs.close();
stmt.close();
}
catch (SQLException e)
{
    System.out.println("****ERROR: " + e.getMessage());
}
System.out.println();

// test DEFAULT Send function
try
{
    // setup type map before using getObject() for UDT data.
    java.util.Map customtypemap = conn.getTypeMap();
    System.out.println("getTypeMap...ok");
    if (customtypemap == null)
    {
        System.out.println("****ERROR: map is null!");
        return;
    }
    customtypemap.put("circle", Class.forName("Circle"));
    System.out.println("put...ok");

    String s = "select c from tab";
    System.out.println(s);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(s);
    if (rs.next())
    {
        Circle c = (Circle)rs.getObject(1, customtypemap);
```

```
        System.out.println("    c.x = " + c.x);
        System.out.println("    c.y = " + c.y);
        System.out.println("    c.radius = " + c.radius);
    }
    rs.close();
    stmt.close();
}
catch (SQLException e)
{
    System.out.println("****ERROR: " + e.getMessage());
}
catch (ClassNotFoundException e)
{
    System.out.println("****ERROR: " + e.getMessage());
}
System.out.println();

// test user's non-support UDR
try
{
    String s = "select area(c) from tab";
    System.out.println(s);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(s);
    if (rs.next())
    {
        double a = rs.getDouble(1);
        System.out.println("    area = " + a);
    }
    rs.close();
    stmt.close();
}
catch (SQLException e)
{
    System.out.println("****ERROR: " + e.getMessage());
}
System.out.println();
```

```
executeUpdate("drop table tab");

// -----
// Closing connection
// -----
try
{
    System.out.print("Closing connection...");
    conn.close();
    System.out.println("ok");
}
catch (SQLException e)
{
    System.out.println("\n***ERROR: " + e.getMessage());
}
}
```

使用您提供的支持函数创建不透明类型

在此示例中，客户端上的 Java™ 类 **Circle2** 被映射到一个名为 **circle2** 的 SQL 不透明类型。**circle2** 不透明类型使用程序员提供的支持函数：

```
import java.text.*;
import com.gbasedbt.jdbc.lfmxUDTSQLInput;
import com.gbasedbt.jdbc.lfmxUDTSQLOutput;

public class Circle2 implements SQLData
{
    private static double PI = 3.14159;

    double x;           // x coordinate
    double y;           // y coordinate
    double radius;

    private String type = "circle2";

    public String getSQLTypeName() { return type; }
```

```
public void readSQL(SQLInput stream, String typeName)
    throws SQLException
{
/* commented out - because the first release of the UDT/UDR Manager feature
*
*      does not support mixing user-supplied support functions
*
*      with server DEFAULT support functions.
* However, once the mix is supported, this code needs to be used to
* replace the existing code.
*
// To be able to use the DEFAULT support functions (other than
// Input/Output) supplied by the server, you must cast the stream
// to IfmxUDTSQLInput.

IfmxUDTSQLInput in = (IfmxUDTSQLInput) stream;
x = in.readDouble();
y = in.readDouble();
radius = in.readDouble();
*/

x = stream.readDouble();
y = stream.readDouble();
radius = stream.readDouble();
}

public void writeSQL(SQLOutput stream) throws SQLException
{
/* commented out - because the 1st release of UDT/UDR Manager feature
*
*      doesn't support the mixing of user support functions
*
*      with server DEFAULT support functions.
* However, once the mix is supported, this code needs to be used to
* replace the existing code.
*
// To be able to use the DEFAULT support functions (other than
// Input/Output) supplied by the server, you must cast the stream
// to IfmxUDTSQLOutput.

IfmxUDTSQLOutput out = (IfmxUDTSQLOutput) stream;
```

```
        out.writeDouble(x);
        out.writeDouble(y);
        out.writeDouble(radius);
    */

    stream.writeDouble(x);
    stream.writeDouble(y);
    stream.writeDouble(radius);
}

/**
 * Input function - return the object from the String representation -
 * 'x y radius'.
 */
public static Circle2 fromString(String text)
{
    Number a = null;
    Number b = null;
    Number r = null;

    try
    {
        ParsePosition ps = new ParsePosition(0);
        a = NumberFormat.getInstance().parse(text, ps);
        ps.setIndex(ps.getIndex() + 1);
        b = NumberFormat.getInstance().parse(text, ps);
        ps.setIndex(ps.getIndex() + 1);
        r = NumberFormat.getInstance().parse(text, ps);
    }
    catch (Exception e)
    {
        System.out.println("In exception : " + e.getMessage());
    }

    Circle2 c = new Circle2();
    c.x = a.doubleValue();
    c.y = b.doubleValue();
}
```

```
        c.radius = r.doubleValue();

        return c;
    }

    /**
     * Output function - return the string of the form 'x y radius'.
     */
    public static String makeString(Circle2 c)
    {
        StringBuffer sbuff = new StringBuffer();
        FieldPosition fp = new FieldPosition(NumberFormat.INTEGER_FIELD);
        NumberFormat.getInstance().format(c.x, sbuff, fp);
        sbuff.append(" ");
        NumberFormat.getInstance().format(c.y, sbuff, fp);
        sbuff.append(" ");
        NumberFormat.getInstance().format(c.radius, sbuff, fp);

        return sbuff.toString();
    }

    /**
     * user function - get the area of a circle.
     */
    public static double area(Circle2 c)
    {
        return PI * c.radius * c.radius;
    }
}
```

不透明类型

下列 JDBC 客户端应用程序将类 Circle2（在 Circle2.jar 中打包）作为不透明类型安装在系统目录中。然后，应用程序可以在 SQL 语句中使用不透明类型 Circle2 作为数据类型：

```
import java.sql.*;
import java.lang.reflect.*;

public class PlayWithCircle2
```

```
{
    String dbname = "test";
    String url = null;
    Connection conn = null;

    public static void main (String args[])
    {
        new PlayWithCircle2(args);
    }

    PlayWithCircle2(String args[])
    {

        // -----
        // Getting URL
        // -----
        if (args.length == 0)
        {
            System.out.println("\n***ERROR: connection URL must be provided " +
                               "in order to run the demo!");

            return;
        }
        url = args[0];
        // -----
        // Loading driver
        // -----
        try
        {
            System.out.print("Loading JDBC driver...");
            Class.forName("com.gbasedbt.jdbc.Driver");
        }
        catch (java.lang.ClassNotFoundException e)
        {
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
        }
        try
        {
            conn = DriverManager.getConnection(url);
        }
        catch (SQLException e)
        {
            System.out.println("URL = " + url + "");
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
        }

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

6. 7. 6 不需现有的 **Java** 类创建不透明类型

在此示例中，使用客户端上的 Java™ 类 **MyCircle** 在数据库服务器中创建一个名为 **ACircle** 的固定长度的不透明类型。**ACircle** 不透明类型使用数据库服务器提供的缺省支持函数：

```
import java.sql.*;

public class MyCircle
{
    String dbname = "test";
    String url = null;
    Connection conn = null;

    public static void main (String args[])
    {
        new MyCircle(args);
    }

    MyCircle(String args[])
    {
        System.out.println("-----");
        System.out.println("- Start - Demo 3");
        System.out.println("-----");

        // -----
        // Getting URL
        // -----
        if (args.length == 0)
        {
            System.out.println("\n***ERROR: connection URL must be provided " +
                               "in order to run the demo!");

            return;
        }
        url = args[0];

        // -----
        // Loading driver
        // -----
        try
```



```
        {
            System.out.print("Loading JDBC driver...");
            Class.forName("com.gbasedbt.jdbc.Driver");
            System.out.println("ok");
        }
    catch (java.lang.ClassNotFoundException e)
    {
        System.out.println("\n***ERROR: " + e.getMessage());
        e.printStackTrace();
        return;
    }

    // -----
    // Getting connection
    // -----
    try
    {
        System.out.print("Getting connection...");
        conn = DriverManager.getConnection(url);
        System.out.println("ok");
    }
    catch (SQLException e)
    {
        System.out.println("URL = " + url + "");
        System.out.println("\n***ERROR: " + e.getMessage());
        e.printStackTrace();
        return;
    }

    // -----
    // Setup UDT meta data
    // -----
    UDTMetaData mdata = null;
    try
    {
        mdata = new UDTMetaData();
        System.out.print("Setting fields in mdata...");
        mdata.setSQLName("acircle");
```

```
        mdata.setLength(24);
        mdata.setFieldCount(3);
        mdata.setFieldName(1, "x");
        mdata.setFieldName(2, "y");
        mdata.setFieldName(3, "radius");
        mdata.setFieldType(1, com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
        mdata.setFieldType(2, com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
        mdata.setFieldType(3, com.gbasedbt.lang.IfxTypes.IFX_TYPE_INT);
        // set class name if don't want to use the default name
        // <udtsqlname>.class
        mdata.setClassName("ACircle");
        mdata.setJarFileSQLName("ACircleJar");
        mdata.keepJavaFile(true);
        System.out.println("ok");
    }
catch (SQLException e)
{
    System.out.println("***ERROR: " + e.getMessage());
    return;
}

// -----
// create java file for UDT and install UDT in the database
// -----
UDTManager udtmgr = null;
try
{
    udtmgr = new UDTManager(conn);

    System.out.println("Creating .class/.java files - " +
                        "createUDTClass()");
    String classname = udtmgr.createUDTClass(mdata); // generated
                                                //java file is kept
    System.out.println("    classname = " + classname);

    System.out.println("\nCreating .jar file - createJar()");
    String jarfilename = udtmgr.createJar(mdata,
```

```
        new String[]{"ACircle.class"}); // jarfilename is
                                         // <udtsqlname>.jar
                                         // ie. acircle.jar

        System.out.println("\nsetJarTmpPath()");
        udtmgr.setJarTmpPath("/tmp");

        System.out.print("\ncreateUDT()...");
        udtmgr.createUDT(mdata,
            "/vobs/jdbc/demo/tools/udtudrmgr/" + jarfilename, "ACircle", 0);
        System.out.println("ok");
    }
catch (SQLException e)
{
    System.out.println("\n***ERROR: " + e.getMessage());
    return;
}
System.out.println();

// -----
// Now use the UDT
// -----
try
{
    String s = "drop table tab";
    System.out.print(s + "...");
    Statement stmt = conn.createStatement();
    int count = stmt.executeUpdate(s);
    stmt.close();
    System.out.println("ok");
}
catch (SQLException e)
{
    // -206 The specified table (%s) is not in the database.
    if (e.getErrorCode() != -206)
    {

```

```
        System.out.println("\n***ERROR: " + e.getMessage());
        return;
    }
    System.out.println("ok");
}

executeUpdate("create table tab (c acircle)");

// test DEFAULT Input function
executeUpdate("insert into tab values ('10 10 10')");

// test DEFAULT Output function
try
{
    String s = "select c::lvarchar from tab";
    System.out.println(s);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(s);
    if (rs.next())
    {
        String c = rs.getString(1);
        System.out.println("    acircle = " + c + "");
    }
    rs.close();
    stmt.close();
}
catch (SQLException e)
{
    System.out.println("***ERROR: " + e.getMessage());
}
System.out.println();

executeUpdate("drop table tab");

// -----
// Closing connection
// -----
```

```
try
{
    System.out.print("Closing connection...");
    conn.close();
    System.out.println("ok");
}
catch (SQLException e)
{
    System.out.println("\n***ERROR: " + e.getMessage());
}

System.out.println("-----");
System.out.println("- End - UDT Demo 3");
System.out.println("-----");
}
```

6. 7. 7 使用 **UDRManager** 创建 **UDR**

以下代码显示应用程序如何使用 **UDRManager** 和 **UDRMetaData** 类将客户端上的 **Java™** 类（数据库服务器无法访问）转换为数据库服务器中的 **Java UDR**。之后，应用程序可以在 **SQL** 语句中引用此 **UDR**。在此示例中，客户端上的 **Java** 类被命名为 **Group1**。该类具有两个例程 **udr1** 和 **udr2**。

下列代码创建 **Group1** 类中的方法以在数据库服务器中注册为 **UDR**：

```
import java.sql.*;

public class Group1
{
    public static String udr1 (String s1, String s2)
    throws SQLException
    {
        return s1 + s2;
    }

    // Return a formatted string with all inputs
    public static String udr2 (Integer i, String s1,
        String s2) throws SQLException
```

```
{  
    return "{" + i + "," + s1 + "," + s2 + "}";  
}  
}
```

以下代码将数据库服务器中 Java 方法 udr1 和 udr2 创建为 UDR group1_udr1 和 group1_udr2，然后使用 UDR：

```
import java.sql.*;  
import java.lang.reflect.*;  
  
public class PlayWithGroup1  
{  
    // Open a connection...  
    url = "jdbc:gbasedbt-sqli://hostname:portnum:db/  
    gbasedbtserver=servname;user=scott;password=tiger;  
    myConn = DriverManager.getConnection(url);  
  
    //Install the routines in the database.  
    UDRManager udtmgr = new UDRManager(myConn);  
    UDRMetaData mdata = new UDRMetaData();  
    Class gp1 = Class.forName("Group1");  
    Method method1 = gp1.getMethod("udr1",  
    new Class[]{String.class, String.class});  
    Method method2 = gp1.getMethod("udr2",  
    new Class[]{Integer.class, String.class, String.class});  
    mdata.setUDR(method1, "group1_udr1");  
    mdata.setUDR(method2, "group1_udr2");  
    mdata.setJarFileSQLName("group1_jar");  
    udtmgr.createUDRs(mdata, "Group1.jar", "Group1", 0);  
  
    // Use the UDRs in SQL statements:  
    Statement stmt = myConn.createStatement();  
    stmt.executeUpdate("create table tab (c1 varchar(10),  
    c2 char(20)", c3 int);  
    stmt.close();  
    Statement stmt = myConn.createStatement();  
    stmt.executeUpdate("insert into tab values ('hello', 'world',
```

```
222)");
stmt.close();

Statement stmt = myConn.createStatement();
ResultSet r = stmt.executeQuery("select c3, group1_udr2(c3, c1, c2)
from tab where group1_udr1(c1, c2) = 'hello world'");

...
}
```

7 全球化和日期格式

本主题主要描述 GBase 8s JDBC Driver 如何通过提高对基于不同语言环境和代码集的 GBase 8s 数据库的访问来扩展 JDK 全球化功能。

全球化使您可以独立于用户的国家或语言来开发软件，然后将软件本地化为多个国家和地区。

7.1 支持 JDK 和全球化

JDK 为开发全球性的应用程序提供了丰富的 API。这些全球化 API 基于 Unicode 2.0 代码集，可以将文本、数字、日期、货币和用户定义的对象调整为任何国家的惯例。

这些全球化 API 集中在这三个包中：

- java.text 包，包含以对语言环境敏感的方式处理文本的类和接口。
- java.io 包，包含用于导出和导入非 Unicode 字符数据的新类。
- java.util 包，包含 Locale 类、全球化支持类以及用于日期和时间处理的新类。

重要： JDK 语言环境和 JDK 代码集之间没有联系；您必须保持这些代码集的一致性。

7.2 支持 GBase 8s GLS 变量

全球化将下表中的环境变量添加到 GBase 8s JDBC Driver。

支持的 GBase 8s 环境变量	描述

支持的 GBase 8s 环境 变量	描述
CLIENT_LOCALE	指定访问数据库的客户机的本地语言环境。提供例如 GL_DATE 格式这样的用户定义格式的缺省值。用户定义的数据类型可以使用它来进行代码集转换。数据库服务器使用它和 DB_LOCALE 变量一起建立服务器处理的语言环境。 DB_LOCALE 和 CLIENT_LOCALE 值必须是相同的，或者它们的代码集必须是可转换的。
DBCENTURY	使您为具有一位数或两位数年份的 DATE 值指定适当的扩展。
DBDATE	指定 DATE 列中值的最终用户格式。支持与早期版本兼容；首选 GL_DATE 。
DB_LOCALE	指定数据库的本地语言环境。GBase 8s JDBC Driver 使用此变量在Unicode 和数据库本地语句环境中指定代码集转换。数据库服务器使用它和 CLIENT_LOCALE 变量一起建立服务器处理语言环境。 DB_LOCALE 和 CLIENT_LOCALE 值必须是一样的，或者它们的代码集必须是可转换的。
GL_DATE	指定 DATE 列中值的最终用户格式。
GL_USEGLU	要使用 Unicode 国际化组件（ICU）启用 Java/JDBC 客户端应用程序进行 Unicode 的归类，请在连接到GBase 8s 实例之前，在连接字符串中指定 GL_USEGLU=1 。这使服务器能够使用 Java™ 所需的高级 Unicode 转换。在启动服务器之前和创建数据库之前，数据库服务器环境中的 GL_USEGLU 环境变量值必须设置为 1。
NEWCODESET	允许在 GBase 8s JDBC Driver 的发行版之间创建新的代码集。
NEWLOCALE	允许在 GBase 8s JDBC Driver 的发行版之间定义新的语言环境。

即使有 **CLIENT_LOCALE** 设置可用，GBase 8s JDBC Driver 也不更改十进制格式。全球化应该在使用 **DecimalFormat** 类的 Java 应用程序中完成。

重要： 除非数据库服务器支持 GBase 8s GLS 特性，才支持 DB_LOCALE、CLIENT_LOCALE 和 GL_DATE 变量。

7.3 支持 DATE 最终用户格式

最终用户格式是 DATE 值出现在字符串变量中的格式。本节描述了指定 DATE 最终用户格式的 GL_DATE、DBDATE 和 DBCENTURY 变量。这些变量是可选的。

重要： GBase 8s JDBC Driver 不支持 DBCENTURY 或 GL_DATE 环境变量的 ALS 6.0、5.0 或 4.0 格式。

7.3.1 GL_DATE 变量

GL_DATE 环境变量指定 DATE 列的值的最终用户格式。GL_DATE 格式字符串可以包含以下字符：

- 一个或多个空格字符
- 普通字符（不是百分号或空格字符）
- 格式化指令。由百分号（%）和一个或两个指定所需替换的转换字符组成

在下表中定义日期格式化伪指令。

指令	替换为
%a	在语言环境中定义为星期几名称的缩写
%A	语言环境中定义的星期几的完整名称
%b	语言环境中定义的月的缩写
%B	语言环境中定义的完整的月的名称
%C	作为十进制（00 到 99）的世纪数（年除以 100 并截断为整数）
%d	一个月份中的日期数（01 到 31） 单个数字前面要加零（0）
%D	与 %m/%d/%y 格式相同
e	一个月份中的日期数（1 到 31） 单个数字前面用空格补齐

指令	替换为
%h	与 %b 格式指令相同
%iy	两位数表示的年份（00 到 99） 这是针对于 %y 的特定于 GBase 8s 的格式化指令
%iY	四位数表示的年份（0000 到 9999） 这是针对于 %Y 的特定于 GBase 8s 的格式化指令
%m	月份（01 到 12）
%n	newline 字符
%t	TAB 字符
%w	星期数（0 - 6） 0 代表本地语言中的星期日。
%x	表示语言环境定义的特殊日期
%y	两位数表示的年份（00 - 99）
%Y	四位数表示的年份（ 0000 - 9999）
%%	% （在格式化字符串中允许 % ）

重要： 不支持字段规范的 GL_DATE 可选日期格式限定符。

例如，通过使用 %4m 显示月份作为十进制数值，不支持最大字段宽度为 4 。

不支持 **GL_DATE** 转换修饰符 **O**，它指示使用替代数字用于替换日期格式 。

任何两个格式化指令之间必须出现空格或其它非字母数字字符。如果 **GL_DATE** 变量格式与任何有效的格式指令都不对应，则当数据库服务器尝试格式化日期时会发生错误。

例如，对于美国英语语言环境，您可以使用以下格式将 09/29/1998 格式化为内部 **DATE** 值：

```
* Sep 29, 1998 this day is:(Tuesday), a fine day *
```

要创建此格式， 将 **GL_DATE** 环境变量设置为如下值：

```
* %b %d, %Y this day is:(%A), a fine day *
```

要将此日期值插入到具有日期列的数据库表中， 您可以执行以下类型的插入：

- 非本地化 SQL，其中 SQL 语句未更改地发送到数据库服务器

输入完全按照 GL_DATE 设置预期的日期

- 本地化 SQL，将转义语法转换为特定于 GBase 8s 格式

以 JDBC 转义格式 yyyy-mm-dd 输入日期值；该值会自动转换为 GL_DATE 格式。

以下示例显示这两种类型的插入：

要从数据库中检索格式化的 GL_DATE DATE 值，请调用 ResultSet 类的 getString() 方法。

将表示日期的字符串输入数据库表的 char 、 varchar 或 lvarchar 类型列，您还可以建立表示日期字符串值的日期对象。此日期字符串值必须是 GL_DATE 格式。

以下示例显示查询 DATE 值的这两种方式：

```
PreparedStatement pstmt = conn.prepareStatement("Select * from
    tablename "
    + "where col2 like ?;");
pstmt.setString(1, "%Tue%");
ResultSet r = pstmt.executeQuery();
while(r.next())
{
    String s = r.getString(1);
    java.sql.Date d = r.getDate(2);
    System.out.println("Select: column col1 (GL_DATE format) = <"
    + s + ">");
    System.out.println("Select: column col2 (JDBC Escape format) = <"
    + d + ">");
}
r.close();
pstmt.close();
```

7.3.2 DBDATE 变量 (deprecated)

对新的应用程序使用 GL_DATE 环境变量。

DBDATE 环境变量指定 DATE 列中的值的最终用户格式。使用以下方式使用最终用户格式：

- 当输入 DATE 值时，GBase 8s 产品使用 DBDATE 环境变量解释此输入。例如，如果在 INSERT 语句中指定字符 DATE 值，则 GBase 8s 数据库服务器要求此字符值与 DBDATE 变量指定的格式兼容。
- 当显示 DATE 值时，GBase 8s 产品使用 DBDATE 环境变量格式化此输出。

使用标准化格式，您可以指定以下属性

- 日期中月、日和年的顺序
- 年使用两位数（Y2）还是四位数（Y4）输出
- 月、日和年之间的分隔符

格式化字符串可以包含以下字符：

- 连字符（-）、点（.）和斜杠（/）是日期格式中的分隔符字符。分隔符出现在格式化字符串的末尾（例如 Y4MD-）。
- 0 指示不显示分隔符。
- D 和 M 是代表日和月的字符。
- Y2 和 Y4 是代表年和年的位数。

下列格式字符串是有效的标准 DBDATE 格式：

- DMY2
- DMY4
- MDY4
- MDY2
- Y4MD
- Y4DM
- Y2MD
- Y2DM

分隔符通常出现在格式化字符串的末尾（例如 DMY2/）。如果未指定分隔符或有效的字符，则缺省为斜杠字符（/）。

对于美国 ASCII 英语语言环境，DBDATE 的缺省设置为 Y4MD-，其中 Y4 代表四位数的年份，M 代表月份，D 代表日，连字符（-）是分隔符（例如 1998-10-08）。

要将日期值插入到具有日期列的数据库表，您可以执行以下插入：

- 非本地化 SQL。其中 SQL 语句未更改地发送到数据库服务器
输入完全按照 DBDATE 设置预期的日期
- 本地化 SQL。转义语法转换为特定于 GBase 8s 格式
以 JDBC 转义格式 yyyy-mm-dd 输入日期值；该值会自动转换为 DBDATE 格式。

以下示例显示这两种类型的插入（DBDATE 值是 MDY2-）：

```
stmt = conn.createStatement();  
cmd = "create table tablename (col1 date, col2 varchar(20));";  
rc = stmt.executeUpdate(cmd);..
```

```
.String[] dateVals = {"08-10-98", "{d '1998-08-11'}" };
String[] charVals = {"08-10-98", "08-11-98" };
int numRows = dateVals.length;
for (int i = 0; i < numRows; i++)
{
    cmd = "insert into tablename values(" + dateVals[i] + ", " +
    charVals[i] + ")";
    rc = stmt.executeUpdate(cmd);
    System.out.println("Insert: column col1 (date) = " + dateVals[i]);
    System.out.println("Insert: column col2 (varchar) = " + charVals[i]);
}
```

要从数据库检索格式化的 DBDATE DATE 值，请调用 `ResultSet` 类的 `getString` 方法。

要将代表日期的字符串插入到数据库表的 `char`、`varchar` 或 `lvarchar` 类型字符列，您可以创建代表日期字符串值的数据对象。此日期字符串值应为 DBDATE 格式。

以下示例显示了选择 DATE 值的两种方式：

```
PreparedStatement pstmt = conn.prepareStatement("Select * from tablename "
    + "where col1 = ?;");
GregorianCalendar gc = new GregorianCalendar(1998, 7, 10);
java.sql.Date dateObj = new java.sql.Date(gc.getTime().getTime());
pstmt.setDate(1, dateObj);
ResultSet r = pstmt.executeQuery();
while(r.next())
{
    String s = r.getString(1);
    java.sql.Date d = r.getDate(2);
    System.out.println("Select: column col1 (DBDATE format) = <"
    + s + ">");
    System.out.println("Select: column col2 (JDBC Escape format) = <"
    + d + ">");
}
r.close();
pstmt.close();
```

7. 3. 3 DBCENTURY 变量

如果 String 值代表小于三位数年份的 DATE 值并设置了 DBCENTURY，则 GBase 8s JDBC Driver 将此 String 值转换为 DATE 值，并使用 DBCENTURY 属性决定年份的正确的四位数扩展。

下表总结了受影响的方法和受影响的条件。

方法	条件
PreparedStatement.setString(int, String)	目标列为 DATE。
PreparedStatement.setObject(int, String)	目标列为 DATE。
IfxPreparedStatement.IfxSetObject(String)	目标列为 DATE。
ResultSet.getDate(int)ResultSet.getDate(int, Calendar)ResultSet.getDate(String)ResultSet.getDate(String, Calendar)	源列为 String 类型。
ResultSet.getTimestamp(int)ResultSet.getTimestamp(int, Calendar)ResultSet.getTimestamp(String)ResultSet.getTimestamp(String, Calendar)	源列为 String 类型。
ResultSet.updateString(int, String)ResultSet.updateString(String, String)	目标列为 DATE。
ResultSet.updateObject(int, String)ResultSet.updateObject(int, String, int)ResultSet.updateObject(String, String)ResultSet.updateObject(String, String, int)	目标列为 DATE。

下表描述了 DBCENTURY 环境变量的四个可能值。

设置	含义	描述
P	过去	使用过去和现在的世纪来扩展年份的值。
F	未来	使用现在和下一世纪来扩展年份的值。
C	近期	使用过去、现在和下一世纪来扩展年份的值。
R	现在	使用本世纪扩展年份的值。

有关每个设置的算法和示例的信息，请参阅《GBase 8s SQL 指南：参考》的 "环境变量" 章节。

以下是设置 DBCENTURY 值的 URL 示例：

```
jdbc:gbasedbt-sqli://myhost:1533:gbasedbtserver=myserver;  
user=myname;password=mypasswd;DBCENTURY=F;
```

URL 不能有换行符。

当 GBase 8s JDBC Driver 将 `java.sql.Date` 和 `java.sql.Timestamp` 值发送到服务器时，它通常包含四位数的年份。同样，服务器将 GBase 8s 日期值发送到 GBase 8s JDBC Driver 时，它通常包含四位数的年份。

有关如何用 GBase 8s JDBC Driver 使用 `DBCENTURY`，请参阅 `DBCENTURYSelect.java`、`DBCENTURYSelect2.java`、`DBCENTURYSelect3.java`、`DBCENTURYSelect4.java` 和 `DBCENTURYSelect5.java` 示例程序。

7.4 最终用户格式的优先规则

这里列出了定义如何确定内部 `DATE` 值的最终用户格式的优先规则：

5. 如果指定了 `DBDATE` 格式，则使用此格式。
6. 如果指定了 `GL_DATE` 格式，则必须确定语言环境：
 - 如果指定了 `CLIENT_LOCALE` 值，则与 `GL_DATE` 格式化字符串一起使用以显示 `DATE` 值。
 - 如果指定了 `DB_LOCALE` 值，但是没有指定 `CLIENT_LOCALE` 值，则将 `DB_LOCALE` 值与数据库语言环境设置（从用户数据库的 `systables` 表中读取）进行比较，以验证 `DB_LOCALE` 值是否有效。如果 `DB_LOCALE` 值有效，则与 `GL_DATE` 格式化字符串一起使用来显示 `DATE` 值。如果 `DB_LOCALE` 值无效，则同时使用数据库语言环境和 `GL_DATE` 格式化字符串。
 - 如果没有指定 `CLIENT_LOCALE` 或 `DB_LOCALE` 值，则同时使用数据库语言环境和 `GL_DATE` 格式化字符串来显示 `DATE` 值。
7. 如果指定了 `CLIENT_LOCALE` 值，则 `DATE` 格式与此语言环境相关的缺省格式匹配。
8. 如果指定了 `DB_LOCALE` 值，但没有指定 `CLIENT_LOCALE` 值，则将 `DB_LOCALE` 值与数据库语言环境比较来确认 `DB_LOCALE` 值是否有效。
如果 `DB_LOCALE` 值有效，则使用 `DB_LOCALE` 缺省格式。如果 `DB_LOCALE` 值无效，则使用与数据库语言环境设置关联的日期的缺省格式。
9. 如果没有指定 `CLIENT_LOCALE` 或 `DB_LOCALE` 值，则所有的 `DATE` 值都格式化为美国英语格式：`Y4MD-`。

7.5 支持代码集转换

代码集转换将一种代码集的代码数据转换为另一种代码集的字符数据。在客户端/ 服务器环境中，如果客户端和数据库服务器计算机使用不同的代码集表示同一字符，则字符数据可能需要从一种代码集转换为另一种代码集。有关代码集转换的详细信息，请参阅《GBase 8s GLS 用户指南》。

必须为以下类型的字符数据指定代码集转换：

- SQL 数据类型（char 、varchar 、nchar 、nvarchar）
- SQL 语句
- 数据库对象，例如数据库名称、列名、语句标识符名称和游标名称
- 存储过程文本
- 命令文本
- 环境变量

GBase 8s JDBC Driver 转换字符数据使它能在客户机和数据库服务器之间传送。

在 systables 目录中为打开的数据库指定用于转换的代码集（编码）。可以在连接属性或数据库 URL 中设置 DB_LOCALE 和 CLIENT_LOCALE 值。

7.5.1 数据库代码集的字符

Java™ 是基于 Unicode 的，因此 GBase 8s JDBC Driver 在 Unicode 和 GBase 8s 数据库代码集之间转换数据。代码集转换值是从连接时指定的DB_LOCALE 值中提取的。如果 DB_LOCALE 值不正确，则会发出 Database Locale information mismatch 错误。

DB_LOCALE 值必须是有效的 GBase 8s 语言环境，并具有如下表所示的有效的 GBase 8s 代码集名称。下表将支持的 JDK 1.4 编码映射到 GBase 8s 代码集。

GBase 8s 代码集名称	GBase 8s 代码集编号	JDK 代码集
8859-1	819	8859_1
8859-2	912	8859-2
8859-3	57346	8859-3
8859-4	57347	8859-4
8859-5	915	8859-5
8859-6	1089	8859-6
8859-7	813	8859-7
8859-8	916	8859-8
8859-9	920	8859-9

GBase 8s 代码集名称	GBase 8s 代码集编号	JDK 代码集
ASCII	364	ASCII
sjis-s	932	SJIS
sjis	57560	SJIS
utf-8	57372	UTF8
big5	57352	Big5
CP1250	1250	Cp1250
CP1251	1251	Cp1251
CP1252	1252	Cp1252
CP1253	1253	Cp1253
CP1254	1254	Cp1254
CP1255	1255	Cp1255
CP1256	1256	Cp1256
CP1257	1257	Cp1257
cp_949	57356	Cp949
KS5601	57356	Cp949
ksc	57356	Cp949
ujis	57351	EUC_JP
gb	57357	ISO2022CN_GB
GB2312-80	57357	ISO2022CN_GB
cp936	57357	ISO2022CN_GB

不能在不支持 JDK 编码的代码集中使用 GBase 8s 语言环境。此错误语法可能产生 Encoding or code set not supported 错误消息。

下表显示支持的语言环境：

		支持的语言环境		
ar_ae	ar_bh	ar_kw	ar_om	ar_qa
ar_sa	bg_bg	ca_es	cs_cz	da_dk
de_at	de_ch	de_de	el_gr	en_au
en_ca	en_gb	en_us	en_nz	ar_ae

		支持的语言环境		
es_ar	es_bo	es_co	es_cl	es_cr
es_ec	es_es	es_gt	es_mx	es_pa
es_pe	es_py	es_sv	es_uy	es_ve
fi_fi	fr_be	fr_ca	fr_ch	fr_fr
hr_hr	hu_hu	is_is	it_ch	it_it
iw_il	ja_jp	ko_kr	mk_mk	nl_be
nl_nl	no_no	pl_pl	pt_br	pt_pt
ro_ro	ru_ru	sh_yu	sk_sk	sv_se
th_th	tr_tr	uk_ua	zh_cn	zh_tw

7.5.2 客户端代码集 Unicode

因为 Unicode 代码集包含所有的现有代码集，所以 Java™ 虚拟机（JVM）必须提供平台语言环境代码集的字符。在 Java 程序内，您必须始终使用 Unicode 字符。在那个平台上的 JVM 在 Unicode 和语言环境代码集之间转换输入和输出。

例如，您在 Unicode 指定按钮标签，则 JVM 转换文本以正确地指示标签。同样，当 getText() 方法从文本框获取到用户的输入时，无论用户如何输入，客户端应用程序都将获得 Unicode 格式的字符串。

切勿一次读取一个字节的文本文件。始终使用 InputStreamReader() 或 OutputStreamWriter() 方法操纵文本文件。缺省情况下，这些方法使用本地语言环境编码，但是您可以在类的构造函数中指定编码方式，如下所示：

```
InputStreamReader = new InputStreamReader (in, "SJIS");
```

您和 JVM 负责将外部输入转换为正确的 Java Unicode 字符串。此后，使用数据库区域设置编码将数据发送到数据库服务器或从数据库服务器发送数据。

7.5.3 连接具有非 ASCII 字符的数据库

如果您在连接时没有指定数据库名称，则连接必须使用正确的指定数据库的 DB_LOCALE 值打开。

如果关闭数据库并发出了数据库 dbname 语句，则连接继续使用初始的 DB_LOCALE 值解释数据库名称。如果新数据库的 DB_LOCALE 值不符合，则返回错误。在此情况下，客户端程序必须关闭，并使用正确的新数据库的 DB_LOCALE 值重新打开连接。

如果您在连接时提供数据库名称，则 DB_LOCALE 值必须设置为正确的数据库语言环境。

可以通过使用 NEWCODESET 和 NEWLOCALE 连接属性定义语言环境，来连接到 NLS 数据库。有关它们的格式，请参阅 Connecting with the NEWLOCALE and NEWCODESET Environment Variables 。

7.5.4 TEXT 和 CLOB 数据类型的代码集转换

GBase 8s JDBC Driver 不会自动在 TEXT 、 BYTE 、 CLOB 和 BLOB 数据类型的代码集之间进行转换。

可以使用以下方式在 TEXT 和 CLOB 数据类型的代码集之间进行转换：

- 可以通过使用 IFX_CODESETLOB 环境变量自动化客户端和数据库语言环境之间的 TEXT 或 CLOB 数据的代码集转换。
- 可以使用 getBytes() 、 getString() 、 InputStreamReader() 和 OutputStreamWriter() 方法在 TEXT 数据代码集之间进行转换。

使用 IFX_CODESETLOB 环境变量转换

可以自动转换 TEXT 和 CLOB 数据类型的代码集对：

- 在将数据发送到数据库服务器之前从客户端语言环境转换到数据库语言环境。
- 在客户端检索数据之前，从数据库语言环境转换到客户端语言环境。

要自动化 TEXT 和 CLOB 数据类型的代码集之间的转换，请在连接 URL 中使用 IFX_CODESETLOB 环境变量。例如：IFX_CODESETLOB = 4096。还可以使用以下 IfxDataSource 类的方法来设置和获取 IFX_CODESETLOB 值：

```
public void setIfxIFX_CODESETLOB(int codesetlobFlag);  
public int getIfxIFX_CODESETLOB();
```

IFX_CODESETLOB 可以具有以下值：

none

缺省值 不启用自动转换代码集。

0 在内部临时文件中发生自动转换代码集。

> 0 在客户端计算机的内存中发生自动转换代码集。该值指示分配给转换的字节数。

如果分配的字节数小于大对象的大小，则返回错误。

要在内存中执行转换，必须指定一个小于客户机的内存限制的量，并且大于任何已转换大对象的可能大小。

当您使用以下 java.sql.Clob 接口方法或 Clob 接口的 GBase 8s 扩展中的任何一种时，即使设置了 IFX_CODESETLOB 环境变量，也不会执行代码集转换。这些方法包括：

```
lfxCblob::setAsciiStream(long) Clob::setAsciiStream(long position, InputStream fin, int length)
```

IFX_CODESETLOB 仅对来自 `java.sql.PreparedStatement` 接口的方法生效。

但是，在使用以下 `java.sql.Clob` 接口方法或 `Clob` 接口的 GBase 8s 扩展时，Unicode 字符总是自动转换为数据库语言环境代码集。以下是这些方法的列表：

```
Clob::setCharacterStream(long) throws SQLException
Clob::setString(long, String) throws SQLException
Clob:: setString(long pos, String str, int offset, int len)
lfxCblob::setSubString(long position, String str, int length)
```

使用 JDK 方法转换

`getBytes()`、`getString()`、`InputStreamReader()` 和 `OutputStreamWriter()` 方法接受一个代码集参数，该参数可以在 Unicode 与指定的代码集之间相互转换

以下样例代码显示了如何将客户端代码集中的文件转换为 Unicode，然后将其从 Unicode 转换为数据库代码集：

```
File infile = new File("data_jpn.dat");
File outfile = new File ("data_conv.dat");..
pstmt = conn.prepareStatement("insert into t_text values (?)");..
// Convert data from client encoding to database encoding
System.out.println("Converting data ...\n");

    try
    {
        String from = "SJIS";
        String to = "8859_1";
        convert(infile, outfile, from, to);
    }
    catch (Exception e)
    {
        System.out.println("Failed to convert file");
    }

System.out.println("Inserting data ...\n");
try
{
    int fileLength = (int) outfile.length();
    fin = new FileInputStream(outfile);
    pstmt.setAsciiStream(1 , fin, fileLength);
```

```
pstmt.executeUpdate();
}
catch (Exception e)
{
    System.out.println("Failed to setAsciiStream");
}..

public static void convert(File infile, File outfile, String from, String to)
throws IOException
{
    InputStream in = new FileInputStream(infile);
    OutputStream out = new FileOutputStream(outfile);

    Reader r = new BufferedReader( new InputStreamReader( in, from));
    Writer w = new BufferedWriter( new OutputStreamWriter( out, to));

    //Copy characters from input to output. The InputStreamReader converts
    // from the input encoding to Unicode, and the OutputStreamWriter
    // converts from Unicode to the output encoding. Characters that can
    // not be represented in the output encoding are output as '?'

    char[] buffer = new char[4096];
    int len;
    while ((len = r.read(buffer)) != -1)
    w.write(buffer, 0, len);
    r.close();
    w.flush();
    w.close();
}
```

当从数据库检索数据时，可以使用相同的方法将数据从数据库代码集转换为客户端代码集。

7.5.5 BLOB 和 BYTE 数据类型的代码集转换

当您使用 `java.sql.PreparedStatement::setCharacterStream()` 向 CLOB 列插入时，Java™ Unicode 字符会自动转换为数据库语言环境代码集。如果设置了环境变量 **IFX_CODESETLOB**，则它的值确定是使用临时文件执行代码集转换还是在内存中执行代码集转换。如果未设置 **IFX_CODESETLOB**，则 **LOBCACHE** 环境变量确定在临时文件中还是在内存中执行代码集转换。

但是，不鼓励您使用 `java.sql.PreparedStatement::setCharacterStream()` 来插入 BLOB 或 BYTE 列。JDBC 驱动程序不能将 Java 字符插入到数据库中，因此会尝试对字符进行行代码集转换。使用 `java.sql.PreparedStatement::setBinaryStream()` 是插入 BLOB 或 BYTE 列的首选方式。

7.6 用户定义的语言环境

GBase 8s JDBC Driver 使用 JDK 全球化 API 操纵国际数据。此 API 中的类和方法将 JDK 语言环境或编码作为参数，但是因为 GBase 8s **DB_LOCALE** 和 **CLIENT_LOCALE** 属性指定基于 GBase 8s 名称的语言环境和代码集，所以这些 GBase 8s 名称将映射到 JDK 名称。这些映射保存在定期更新的内部表中。

例如，ASCII 代码集的 GBase 8s 和 JDK 名称分别是 8859-1 和 8859_1。GBase 8s JDBC Driver 在其内部表中映射 8859-1 到 8859_1，并在 JDK 类和方法中使用适当的 JDK 名称。

7.6.1 使用 **NEWLOCALE** 和 **NEWCODESET** 环境变量连接

因为可能在这些表的更改之间创建新的语言环境，所以可以使用两个连接属性 **NEWLOCALE** 和 **NEWCODESET** 来指定表中未指定的语言环境或代码集。以下是 URL 使用这些属性的示例：

```
jdbc:gbasedbt-sqli://myhost:1533:gbasedbtserver=myserver;  
user=myname; password=mypasswd;NEWLOCALE=en_us,en_us;  
NEWCODESET=8859_1,8859-1,819;
```

URL 必须是一行。

NEWLOCALE 和 **NEWCODESET** 属性具有以下格式：

```
NEWLOCALE=JDK-locale,lfx-locale:JDK-locale,lfx-locale...  
NEWCODESET=JDK-encoding,lfx-codeset,lfx-codeset-number:JDK-encoding,  
lfx-codeset,lfx-codeset-number...
```

指定的代码集或语言环境映射的数量没有限制。

可以通过使用 **NEWCODESET** 和 **NEWLOCALE** 连接属性连接到 NLS 数据库。

如果指定的参数或值的数量不正确，则会显示 **Locale Not Supported** 或 **Encoding or Code Set Not Supported** 消息。

如果在 URL 或 **DataSource** 对象中设置这些属性，则 **NEWCODESET** 和 **NEWLOCALE** 的新值会覆盖 JDBC 内部表中的值。例如，如果 JDBC 已经在内部映射了 8859-1 到 8859_1，但是您指定 **NEWCODESET=8888,8859-1,819**，则代码集转换时使用新值 8888。

7.6.2 使用 NEWNLSMAP 环境变量连接

要支持连接到 NLS 数据库，则 GBase 8s JDBC Driver 维护一个将 NLS 语言环境映射到对应的 JDK 语言环境和 JDK 代码集的表。随着 JDK 支持使更多的语言环境和代码集可用，NLS 语言环境之前不支持的语言环境和代码集在新的 JDK 中支持。GBase 8s JDBC Driver 支持连接属性 **NEWNLSMAP**，可以使用它指定映射没有在表中指定的 NLS 语言环境。

NEWNLSMAP 属性具有以下格式：

```
NEWNLSMAP=NLS-locale,JDK-locale,JDK-charset:NLS-locale,JDK-locale,  
JDK-charset,....
```

以下是 URL 使用这些属性的示例：

```
jdbc:gbasedbt-sqli://myhost:1533:gbasedbtserver=myserver;  
user=myname;password=mypasswd;NEWNLSMAP=rumanian,ro_RO,ISO8859_2;
```

指定的代码集或语言环境映射的数量没有限制。如果指定的参数或值的数量不正确，则会显示 Locale Not Supported 或 Encoding or Code Set Not Supported 消息。

7.7 支持全球化的错误消息

消息文本通常是 SQLException 对象的文本，但也可以是 SQLWarn 对象或任何其它来自驱动程序程序的文本输出。

启用全球化消息文本输出有两个要求，如下所示：

- 必须将 gbasedbtjdbc_xx.jar 文件的完整路径名添加到 **\$CLASSPATH**（UNIX™）或 **%CLASSPATH%**（Windows™）环境变量。此 JAR 文件包含 GBase 8s JDBC Driver 支持的所有全球化版本的消息文本。支持的语言是英语和中文。
- 如果您使用的是非缺省语言环境，则 **CLIENT_LOCALE** 环境变量值必须在连接时通过属性列表传递给连接对象。有关一般的 **CLIENT_LOCALE** 和 GLS 功能，请参阅支持 GBase 8s GLS 变量。

多个公共类具有将当前连接对象作为参数的构造函数，以便可以直接存取 **CLIENT_LOCALE** 值。如果您希望访问非英文的错误消息，则必须使用包含此连接对象的构造函数。否则，任何来自这些类的消息文本仅为英文。受影响的公共类为 **Interval**、**IntervalYM**、**IntervalDF** 和 **IfxLocator**。有关这些类的构造函数的更多信息，请参阅操作 GBase 8s 类型。

有关如何使用全球化错误消息支持功能的示例，请参阅 GBase 8s JDBC Driver 附带的 locmsg.java 程序。

8 调优和故障排除

本主题提供 GBase 8s JDBC Driver 的调优和故障排除信息。

8.1 调试 JDBC API 程序

可以设置 SQLIDEBUG 连接属性来生成二进制协议跟踪。设置连接属性 SQLIDEBUG 来指定一个文件。例如：

```
SQLIDEBUG=C:\\tmp\\gbasedbtjdbcctrace
```

为每个连接生成一个新的跟踪文件，其后缀为时间戳。如果您正在使用 **IfxDataSource** 接口，则可以使用 **IfxDataSource.setIfxSQLIDEBUG (String fname)** 方法。JDBC jar 文件的调试版本不包含在 GBase 8s JDBC Driver Version 3.00.JC1 或更高的版本中。

重要： 二进制 SQLI 协议跟踪功能（SQLIDEBUG）只能在 GBase 技术支持指导下使用。

8.2 管理性能

本节描述了可能影响查询性能的问题：

- **FET_BUF_SIZE** 和 **BIG_FET_BUF_SIZE** 环境变量
- 大对象的内存管理
- 减少网络流量
- 使用批量插入
- 调整连接池

8.2.1 管理访存缓冲区大小

使用 **FET_BUF_SIZE** 和 **SRV_FET_BUF_SIZE** 环境变量设置访存缓冲区的大小。

将 **SELECT** 语句从 Java™ 程序发送到 GBase 8s 数据库时，返回的行或元组将存储在 GBase 8s JDBC Driver 中的元组缓冲区中。元组缓冲区的缺省大小为返回的元组的大小或 4096 字节。

可以使用 GBase 8s **FET_BUF_SIZE** 环境变量重写元组缓冲区的缺省大小。

FET_BUF_SIZE 可以设置为任何小于等于 2 GiB（2147483648）的正整数。如果设置了 **FET_BUF_SIZE** 环境变量，则它的值应大于缺省元组缓冲区大小，元组缓冲区大小设置为 **FET_BUF_SIZE** 的值。

同样，您可以使用 **SRV_FET_BUF_SIZE** 环境变量设置本地数据库服务器参与跨服务器分布式 DML 事务时使用的访存缓冲区的大小。**SRV_FET_BUF_SIZE** 的最大大小可设置为 1048576（= 1 MiB）。

增加访存缓冲区的大小可以减少 Java™ 程序和数据库之间的网络流量，还会提高查询性能。然而，有时候，增加访存缓冲区的大小实际上会降低查询的性能。如果您的 Java 查询具有

多个活动的到数据库的连接或者计算机上的交换空间有限，则可能会发生这种情况。如果 Java 程序或计算机属于这种情况，则可能不希望使用 **FET_BUF_SIZE** 或 **SRV_FET_BUF_SIZE** 环境变量增加访存缓冲区的大小。

有关设置 GBase 8s 环境变量的更多信息，请参阅连接至数据库。有关增加访存缓冲区大小的更多信息，请参阅《GBase 8s SQL 指南：参考》。

8.2.2 大对象的内存管理

无论何时从数据库服务器访存一个大对象(BYTE、TEXT、BLOB 或 CLOB 数据类型)，该数据会高速缓存到内存或存储到临时文件（如果它超出内存缓冲区）。如果 JDBC 小程序尝试在本地计算机上创建临时文件，则可能会导致安全性问题。在这种情况下，整个大对象必须存储在内存中。

可以使用 **LOBCACHE** 环境变量来指定存储在连接属性列表中的大对象数据的大小，如下所示：

- 要设置分配给内存的最大字节数来保存数据，请将 **LOBCACHE** 值设置为该字节数。

如果数据大小超出 **LOBCACHE** 值，则数据会存储在临时文件中。如果在创建此文件期间发生安全违规，则此数据会存储到内存中。

- 要始终将数据存储在文件中，请将 **LOBCACHE** 值设置为 0。

在这种情况下，如果发生安全违规，则 GBase 8s JDBC Driver 不会尝试将数据存储在内存中。未签名的小程序不支持该设置。有关更多信息，请参阅在 applet 中使用驱动程序。

- 要始终在内存中存储数据。请将 **LOBCACHE** 值设置为负数。

如果所需内存量超出，则 GBase 8s JDBC Driver 抛出 **SQLException** 消息 **Out of Memory**。

如果 **LOBCACHE** 值无效或未定义，则缺省大小为 4096。

可以通过数据库 URL 设置 **LOBCACHE** 值，如下所示：

```
URL = jdbc:gbasedbt-sqli://158.58.9.37:7110/test:user=guest;  
password=iamaguest;gbasedbtserver=oltapshm;  
lobcache=4096";
```

上一示例中，如果大对象大小为 4096 字节或更小，则在内存中存储大对象。如果大对象超出 4096 字节，则 GBase 8s JDBC Driver 会试图创建一个临时文件。如果发生安全隔离，则会为整个大对象分配内存。如果失败，则驱动程序抛出 **SQLException** 消息。

另一示例为：

```
URL = "jdbc:gbasedbt-sqli://icarus:7110/testdb:  
user=guest;passwd=whoknows;gbasedbtserver=olserv01;lobcache=0";
```

上述示例使用临时文件存储访存的大对象。

第三个示例如下：

```
URL = "jdbc:gbasedbt-sqli://icarus:7110/testdb:user=guest:
      passwd=whoknows;gbasedbtserver=olserv01;lobcache=-1";
```

上述示例通常使用内存来存储访存的大对象。

有关如何在 Java™ 程序中使用 TEXT 和 BYTE 数据类型的编程的信息，请参阅 BYTE 和 TEXT 数据类型。有关如何在 Java 程序中使用 BLOB 和 CLOB 数据类型编程的信息，请参阅智能大对象数据类型。

8.2.3 减少网络流量

当您关闭 **Statement** 和 **ResultSet** 对象时，可以使用环境变量 **OPTOFC** 和 **IFX_AUTOFREE** 减少网络流量。

将 **OPTOFC** 设置为 1 来指定：如果已在客户端元组缓冲区检索到所有合格的行，则 **ResultSet.close()** 方法不需要网络。在检索完所有的行后，数据库服务器自动关闭游标。

在调用下一个 **ResultSet.next()** 方法之前，GBase 8s JDBC Driver 在客户端元组缓冲区中可能有也可能没有附加的行。因此，除非 GBase 8s JDBC Driver 已经从数据库服务器检索到所有的行，否则当 **OPTOFC** 设置为 1 时，**ResultSet.close()** 方法可能仍然需要网络往返。

将 **IFX_AUTOFREE** 设置为 1 以指定如果数据库服务器中的游标已经关闭，则 **Statement.close()** 方法不需要网络往返来释放数据库服务器游标资源。

还可以使用 **setAutoFree(boolean flag)** 和 **getAutoFree()** 方法释放数据库服务器游标资源。有关更多信息，请参阅“自动释放”特性。

当游标关闭后，数据库服务器显式地通过 **ResultSet.close()** 方法或隐式地通过 **OPTOFC** 环境变量自动释放游标资源。

当游标资源释放后，该游标不能再被引用。

有关如何使用 **OPTOFC** 和 **IFX_AUTOFREE** 环境变量的示例，请参阅示例代码文件中描述的 **autofree.java** **optofc.java** 演示示例。在这些示例中，使用 **Properties.put()** 方法设置变量。

有关 GBase 8s 环境变量的设置的更多信息，请参阅随同 GBase 8s JDBC 驱动程序的 GBase 8s 环境变量。

8.2.4 批量插入

批量插入功能提高了使用多个值设置多次执行的单个 **INSERT** 语句的性能。有关更多信息，请参阅执行批量插入。

8.2.5 连接池

要提高应用程序的性能和可伸缩性，可以通过引用 `ConnectionPoolDataSource` 对象的 `DataSource` 对象获得数据库服务器的连接。GBase 8s JDBC Driver 提供一个连接池管理器作为 `ConnectionPoolDataSource` 对象的透明组件。连接池管理器在池中保持一个关闭的连接，而不是将连接关闭返回到数据库服务器。每当用户请求新的连接时，连接池管理器从该池获得连接，从而避免关闭服务器并重新打开连接的开销。

如果应用程序收到频繁的定期连接请求，则使用 `ConnectionPoolDataSource` 对象可以显著提高性能。

有关如何以及为何使用 `DataSource` 或 `ConnectionPoolDataSource` 对象的完整信息，请参阅 JDBC 3.0 API。

重要： 此功能不会影响 `IfxXAConnectionPoolDataSource`，它假设连接池由事务管理器处理。

部署 `ConnectionPoolDataSource` 对象

请遵循以下步骤：

- 变量 `cpds` 引用 `ConnectionPoolDataSource` 对象。
- `ConnectionPoolDataSource` 对象的 JNDI 逻辑名称为 `myCPDS`。
- 变量 `ds` 引用 `DataSource` 对象。
- `DataSource` 对象的逻辑名称为 `DS_Pool`。

要部署 `ConnectionPoolDataSource` 对象：

1. 实例化 `IfxConnectionPoolDataSource` 对象。
2. 对此对象设置任何期望调整的属性：

```
cpds.setIfxCPMInitPoolSize(15);
cpds.setIfxCPMMinPoolSize(2);
cpds.setIfxCPMMaxPoolSize(20);
cpds.setIfxCPMServiceInterval(30);
```

3. 使用 JNDI 注册 `ConnectionPoolDataSource` 对象，将逻辑名称映射到该对象：

```
Context ctx = new InitialContext();
ctx.bind("myCPDS",cpds);
```

4. 实例化 `IfxDataSource` 对象。
5. 将 `DataSource` 对象与您为 `ConnectionPoolDataSource` 对象注册的逻辑名称相关联：

```
ds.setDataSourceName("myCPDS",ds);
```

6. 使用 JNDI 注册 `DataSource` 对象：

```
Context ctx = new InitialContext();
ctx.bind("DS_Pool",ds);
```

调整连接池管理器

在部署阶段，您或您的数据库管理员可以通过设置以下任何连接池管理器属性的值来控制连接池在应用程序中的工作方式：

- **IFMX_CPM_INIT_POOLSIZE** 允许您指定初始化 `ConnectionPoolDataSource` 对象和初始化时要为池分配的初始连接数。缺省值为 0。

如果您的应用程序首次初始化 `ConnectionPoolDataSource` 对象时需要多个连接，请设置此属性。

要获得此值，请调用 `getIfxCPMInitPoolSize()`。

要设置此值，请调用 `setIfxCPMInitPoolSize (int init)`。

- **IFMX_CPM_MAX_CONNECTIONS** 允许您指定 `DataSource` 对象可以与服务器同时使用的最大并发物理连接数。

值 -1 指定了一个无限制的数字。缺省值为 -1。

要获得此值，请调用 `getIfxCPMMaxConnections()`。

要设置此值，请调用 `setIfxCPMMaxConnections(int limit)`。

- **IFMX_CPM_MIN_POOLSIZE** 允许您指定池中要维护的最小连接数。请参阅 **IFMX_CPM_MIN_AGE LIMIT** 参数，以了解在池中保留的最小连接数超过年限时要执行的操作。缺省值为 0。

要获得此值，请调用 `getIfxCPMMinPoolSize()`。

要设置此值，请调用 `setIfxCPMMinPoolSize(int min)`，

- **IFMX_CPM_MAX_POOLSIZE** 允许您指定池中要维护的最大连接数。当池到达此大小，所有的连接返回到服务器。缺省值为 50。

要获得此值，请调用 `getIfxCPMMaxPoolSize()`。

要设置此值，请调用 `setIfxCPMMaxPoolSize(int max)`。

- **IFMX_CPM_AGE LIMIT** 允许您指定空闲连接保留在空闲连接池中的时间（以秒为单位）。

缺省值为 -1，这意味着空闲连接将保留直到客户端终止。

要获得此值，请调用 `getIfxCPMAgeLimit()`。

要设置此值，请调用 `setIfxCPMAgeLimit(long limit)`。

- **IFMX_CPM_MIN_AGE LIMIT** 允许您指定额外时间（以秒为单位），以确保在未收到连接请求时保留空闲连接池中的连接。

使用此设置可减少在预期的时间段内没有进行连接请求的池中的资源。值为 0 指示在最小池中未给予连接额外时间；只要它超出 **IFMX_CPM_AGE LIMIT**，连接就会释放到服务器。

缺省值为 -1，这意味着在客户端终止之前，会保存最小数量的空闲连接。

要获得此值，请调用 `getIfxCPMMinAgeLimit()`。

要设置此值，请调用 `setIfxCPMAgeMinLimit(long limit)`。

- `IFMX_CPM_SERVICE_INTERVAL` 允许您指定池服务的频率（以毫秒为单位）。

池服务活动包括添加空闲连接（如果空闲连接数低于最小值）和删除空闲连接。缺省值为 50。

要获得此值，请调用 `getIfxCPMServiceInterval()`。

要设置此值，请调用 `setIfxCPMServiceInterval (long interval)`。

- `IFMX_CPM_ENABLE_SWITCH_HDRPOOL` 允许您指定是否在 HAC 数据库服务器对的主和辅助连接池之间自动切换。

如果您的应用程序需要使用连接池和高可用数据复制，请设置此属性。缺省值为 `false`。

要获得此值，请调用 `getIfxCPMSwitchHDRPool()`。

要设置此值，请调用 `setIfxCPMSwitchHDRPool(boolean flag)`。

其中一些属性与 Sun JDBC 3.0 属性重叠。下表列出了 Sun JDBC 3.0 属性及其等效的 GBase 8s 属性。

Sun JDBC 属性名称	GBase 8s 属性名称	其它信息
<code>initialPoolSize</code>	<code>IFMX_CPM_INIT_POOLSIZE</code>	
<code>maxPoolSize</code>	<code>IFMX_CPM_MAX_POOLSIZE</code>	对于 <code>maxPoolSize</code> ，0 指示没有最大大小。对于 <code>IFMX_CPM_MAX_POOLSIZE</code> ，您必须指定一个值。
<code>minPoolSize</code>	<code>IFMX_CPM_MIN_POOLSIZE</code>	
<code>maxIdleTime</code>	<code>IFMX_CPM_AGE LIMIT</code>	对于 <code>maxIdleTime</code> ，0 指定没有时间限制。对于 <code>IFMX_CPM_AGE LIMIT</code> ，-1 指示没有时间限制。

Sun JDBC 3.0 不支持以下属性：

- `maxStatements`
- `propertyCycle`

高可用数据复制与连接池

连接池的 GBase 8s JDBC Driver 实现提供了与 HAC 对中数据库服务器进行池连接的功能：

- 主池包含到 HAC 对中主服务器的连接。
- 辅助池包含到 HAC 对中辅助服务器的连接。

不需要更改应用程序代码就可以利用 HAC 和连接池。将 IFMX_CPM_ENABLE_SWITCH_HDRPOOL 属性设置为 TRUE 以允许在两个池之间切换。当允许切换时，连接池管理器将验证并激活相应的连接池。

当主服务器失败，连接池管理器激活辅助池。激活辅助池后，连接池管理器将验证池的状态以检查主服务器是否正在运行。如果主服务器正在运行，则连接池管理器会将新连接切换到主服务器，并将活动池设置为主池。

如果 IFMX_CPM_ENABLE_SWITCH_HDRPOOL 设置为 FALSE，则可以通过调用 activateHDRPool_Primary() 或 activateHDRPool_Secondary() 方法强制切换到另一个连接池：

```
public void activateHDRPool_Primary(void) throws SQLException  
  
public void activateHDRPool_Secondary(void) throws SQLException
```

activateHDRPool_Primary() 方法将主连接池切换为活动的连接池。

activateHDRPool_Secondary() 方法将辅助连接池切换为活动的连接池。

可以使用连接池的 isReadOnly()、isHDREnabled() 和 getHDRtype() 方法（请参阅 检查高可用性辅助服务器的只读状态）。

清除池连接

可以通过设置数据库属性，例如 AUTOCOMMIT 和 TRANSACTION ISOLATION 更改其原始默认属性的连接。当连接关闭时，这些属性恢复为缺省值。但是，在连接返回到池时，池化的连接不自动地恢复为缺省属性。

GBase 8s JDBC Driver 中，您可以调用 scrubConnection() 方法来：

- 将数据库属性和连接级别属性重新设置为缺省值。
- 关闭打开的游标和事务。
- 保留所有的语句。

现在启用应用程序服务器高速缓存这些语句，并且可以在应用程序和会话中使用它来为最终用户应用程序提供更好的性能。

scrubConnection() 方法的签名为：

```
public void scrubConnection() throws SQLException
```

以下示例演示如何调用 scrubConnection()：

```
try  
{  
    IfmxConnection conn = (IfmxConnection)myConn;  
    conn.scrubConnection();  
}  
catch (SQLException e)
```

```
{
    e.printStackTrace();
}
```

以下方法验证是否调用 `scrubConnection()` 释放所有的语句：

```
public boolean scrubConnectionReleasesAllStatements()
```

管理连接

下表对比了 `connection.close()` 和 `scrubConnection()` 方法在连接池设置中的不同实现。

连接池状态	<code>connection.close()</code> 方法的行为	<code>scrubconnection()</code> 方法的行为
没有设置连接池	关闭数据库连接，所有相关的语句对象及其结果集的连接不再有效。	将连接返回到缺省状态，保留打开的连接，但是关闭结果集连接。只释放与结果集相关联的连接。
使用 GBase 8s 实现的连接池	关闭到数据库的连接并重新打开连接，以关闭与连接对象关联的任何语句，并将连接重置为原始状态，然后连接对象返回到连接池。当新的应用程序请求连接时连接有效。	将连接返回到缺省状态，保留所有打开的连接，但是关闭所有的结果集。这里不建议调用此方法。
使用 AppServer 实现的连接池	由用户定义的连接池实现	将连接返回到缺省状态，保留打开的语句，但是关闭结果集

8.2.6 避免应用程序挂起问题（仅限于 HP-UX）

如果在 HP-UX 服务器上的 JDBC 应用程序挂起，那么请检查 HP-UX 服务器上 **PTHREAD_COMPAT_MOD** 环境变量的设置。**PTHREAD_COMPAT_MODE** 环境变量应该设置为 1。此变量告知 `pthread` 库（`libpthread`）以 1 X 1 模式而不是 MxN 模式运行。1 X 1 是 HP-UX 上的缺省模式。设置此环境变量应该能解决此挂起问题。

9 Oracle兼容模式

本章节主要介绍GBase 8s数据库在Oracle兼容模式下的配置方式和使用方法。

GBase 8s数据库从GBase8sV8.8_3.3.0 版本开始提供oracle兼容模式，可通过在JDBC连接串中添加参数sqlmode=Oracle来开启，此参数为可选参数。

在Oracle兼容模式下，JDBC支持：

- 1）兼容oracle的语法sql（具体支持程度参考GBase 8s数据库的兼容性）
- 2）**package**特性
- 3）**DatabaseMetadata**的行为与Oracle JDBC **DatabaseMetadata**的行为兼容

9.1 参数说明及使用方法

9.1.1 参数说明

参数名	可选值	默认值	说明
sqlmode	gbase, oracle	gbase	<p>GBase模式：在不配置sqlmode情况下，默认行为为gbase模式，其与在url串中显式配置该值的行为一致。</p> <p>Oracle模式：需要在url串中显式配置sqlmode=Oracle，配置该参数后JDBC驱动将使用Oracle兼容模式访问数据库。</p> <p>参数值不分区大小写。</p> <p>该参数为session级生效，不会影响全局其它session。</p>

9.1.2 使用方法

Oracle模式JDBC连接串：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533/testDB:
GBASEDBTSERVER=myserver;user=rdtest;password=test;sqlmode=oracle
```


GBase模式JDBC连接串：

```
jdbc:gbasedbt-sqli://123.45.67.89:1533/testDB:
GBASEDBTSERVER=myserver;user=rdtest;password=test;sqlmode=gbase
```

等价于

```
jdbc:gbasedbt-sqli://123.45.67.89:1533/testDB:
GBASEDBTSERVER=myserver;user=rdtest;password=test
```

10 附录

10.1 示例代码文件

本节包含的表格列出并简要描述了客户端版本的 GBase 8s JDBC Driver 提供的代码示例。

大多数示例可以通过更改连接 URL 的语法与客户端 JDBC 一起工作。有关更多信息，请参阅数据库 URL 的格式。

tools/udtudrmgr 目录和 demo/xml 目录中的示例仅适用 2.2 版本中的客户端 JDBC。

10.1.1 可用示例摘要

在以下两个目录中提供示例：

- 安装 GBase 8s JDBC Driver 软件的 demo 目录
- demo 目录下的 tools 目录

demo 目录中的示例

每个实例都具有自己的子目录。大多数目录包括描述示例以及如何运行这些示例的 README 文件。

目录	示例的类型
basic	显示一般数据库操作的示例
bson	显示 IfxBSONObject 扩展类用法的示例，此类用于访问 GBase 8s BSON 数据类型
clob-blob	使用智能大对象的示例
udt-distinct	使用不透明类型和 DISTINCT 数据类型的示例（使用不透明类型的附加

目录	示例的类型
	示例在udtudrmgr 目录中的示例)
complex-types	使用行和集合类型的示例
rmi	使用远程方法调用的示例
stores7	stores7 演示数据库
pickaseat	使用 DataSource 对象的示例
connection-pool	说明使用连接池的示例
proxy	说明使用 HTTP 代理服务器的示例
xml	说明存储和检索 XML 文件的示例
hdr	说明使用高可用数据复制的示例

basic 目录中的示例

下表列出了 basic 目录中的文件。

示例程序名	描述
autofree. java	显示如何使用 IFX_AUTOFREE 环境变量
BatchUpdate. java	显示如何发送批量更新到服务器
ByteType. java	显示如何插入或选择包含 BYTE 列的表
CallOut1. java	使用 CallableStatement 方法执行具有 OUT 参数的 C 函数
CallOut2. java	使用 CallableStatement 方法执行具有 OUT 参数的 SPL 函数
CallOut3. java	使用 IfmxCallableStatement.IfRegisterOutParameter() 方法执行具有 Boolean OUT 的 C 函数
CallOut4. java	使用 IfmxCallableStatement.hasOutParameter() 方法执行具有 CLOB 类型 OUT 参数的 C 函数
CreateDB. java	创建名为 testDB 的数据库
DBCENTURYSelect. java	使用 getString() 方法检索日期字符串表示形式，其中四位数年份扩展是基于 DBCENTURY 属性值
DBCENTURYSelect2. java	检索一个日期字符串表示形式，其中四位数年份扩展是基于 DBCENTURY 属性值，使用字符串到二进制的转换 使用 getDate() 方法创建日期字符串表示形式所基于

示例程序名	描述
	的 <code>java.sql.Date</code> 对象
DBCENTURYSelect3.java	检索一个日期字符串表示形式，其中四位数展开式基于 <code>DBCENTURY</code> 属性值，使用字符串到二进制的转换 使用 <code>getTimestamp()</code> 方法来构建日期字符串表示形式所基于的 <code>java.sql.Timestamp</code> 对象
DBCENTURYSelect4.java	检索一个日期字符串表示形式，其中四位数展开式基于 <code>DBCENTURY</code> 属性值，使用二进制到字符串的转换 使用 <code>getDate()</code> 方法来构建日期字符串表示形式所基于的 <code>java.sql.Date</code> 对象
DBCENTURYSelect5.java	检索一个日期字符串表示形式，其中四位数展开式基于 <code>DBCENTURY</code> 属性值，使用二进制到字符串的转换 使用 <code>getTimestamp()</code> 方法来构建日期字符串表示形式所基于的 <code>java.sql.Timestamp</code> 对象
DBConnection.java	创建到数据库和数据库服务器的连接
DBDATESelect.java	显示如何根据 URL 字符串中的 <code>DBDATE</code> 属性值从数据库检索日期对象和日期字符串表示形式
DBMetaData.java	显示如何检索具有 <code>DatabaseMetaData</code> 接口的数据库消息
DropDB.java	删除名为 <code>testDB</code> 的数据库
ErrorHandling.java	显示如何检索 RSAM 错误消息
GLDATESelect.java	显示如何根据 URL 字符串中的 <code>GL_DATE</code> 属性值从数据库检索日期对象和日期字符串表示形式
Intervaldemo.java	显示如何插入和选择 GBase 8s 间隔数据
LOCALESelect.java	显示如何根据 URL 字符串中的 <code>CLIENT_LOCALE</code> 属性值从数据库检索日期对象和日期字符串表示形式
locmsg.java	显示如何使用支持本地化错误消息的 GBase 8s 扩展方法
MultiRowCall.java	显示如何在存储过程调用中返回多个行
OptimizedSelect.java	显示如何使用 <code>FET_BUF_SIZE</code> 环境变量调整 GBase 8s JDBC Driver 元组缓冲区大小
optofc.java	显示如何使用 <code>OPTOFC</code> 环境变量

示例程序名	描述
PropertyConnection.java	显示如果通过属性列表指定连接环境变量
RSMetaData.java	显示如何使用 ResultSetMetaData 接口检索结果集的信息
ScrollCursor.java	显示如何使用滚动游标检索结果集
Serial.java	显示如何插入和检索 GBase 8s SERIAL 及 SERIAL8 数据
SimpleCall.java	显示如何调用存储过程
SimpleConnection.java	显示如何连接到数据库或数据库服务器
SimpleSelect.java	显示如何将简单的 SELECT 查询发送到数据库服务器
TextConv.java	显示如何将文件从客户端代码集转换为 Unicode，然后从 Unicode 转换为数据库代码集
TextType.java	显示如何插入和选择包含 TEXT 数据类型的列的表
UpdateCursor1.java	显示如何在查询中使用 ROWID 列创建一个可更新的滚动游标
UpdateCursor2.java	显示如何在查询中使用 SERIAL 列创建一个可更新的滚动游标
UpdateCursor3.java	显示如何在查询中使用主键列创建一个可更新的滚动游标

clob-blob 目录中的示例

下表列出了 clob-blob 目录中的文件。

示例程序名称	描述
demo1.java	显示如何创建两个具有 BLOB 和 CLOB 列的表，并比较数据
demo2.java	显示如何创建具有 BYTE 和 TEXT 列的表和另一个具有 BLOB 和 CLOB 列的表，然后比较这两个表的数据
demo3.java	显示如何创建一个具有 BLOB 和 CLOB 列的表，和一个具有 BYTE 和 TEXT 列的表，然后比较这两个表的数据
demo4.java	显示如何创建两个分别具有 BYTE 和 TEXT 列的表，并比较数据
demo5.java	显示如何将文件的数据存储到 BLOB 表列
demo6.java	显示如何读取智能大对象中的部分数据
demo_11.java	显示如何将文件的数据读取到缓冲区并将缓冲区的数据写入智能大对象
demo_13.java	显示如何将数据写入智能大对象，然后将智能大对象插入到表
demo_14.java	显示如何从表访问智能大对象

udt-distinct 目录中的示例

下表列出了 `udt-distinct` 目录中的文件（使用不透明类型的附加示例在 `udtudrmgr` 目录中的示例中有所描述）。

示例程序名称	描述
<code>charattrUDT.java</code>	显示如何使用 <code>SQLData</code> 实现不透明固定长度类型
<code>createDB.java</code>	创建一个其它 <code>udt-distinct</code> 演示文件使用的数据库
<code>createTypes.java</code>	显示如何在数据库中创建不透明和 <code>distinct</code> 类型
<code>distinct_d1.java</code>	显示如何不使用 <code>SQLData</code> 而创建 <code>distinct</code> 类型
<code>distinct_d2.java</code>	显示如何不使用 <code>SQLData</code> 创建第二个 <code>distinct</code> 类型
<code>dropDB.java</code>	删除其它 <code>udt-distinct</code> 演示文件使用的数据库
<code>largebinUDT.java</code>	显示如何使用 <code>SQLData</code> 实现不透明类型(启用智能大对象)
<code>manualUDT.java</code>	显示如何实现允许更改输入流中的位置的不透明类型
<code>myMoney.java</code>	显示如何使用 <code>SQLData</code> 实现 <code>distinct</code> 类型
<code>udt_d1.java</code>	显示如何创建固定长度不透明类型
<code>udt_d2.java</code>	显示如何使用嵌入式智能大对象创建不透明类型
<code>udt_d3.java</code>	显示如何创建一个允许更改输入流中位置的不透明类型

complex-types 目录中的示例

下表列出了 `complex-types` 目录中的文件。

示例程序名称	描述
<code>createDB.java</code>	创建具有命名行的数据库
<code>list1.java</code>	使用 <code>java.sql.Array</code> 和 <code>java.util.Collection</code> 类插入和选择简单集合
<code>list2.java</code>	插入和选择具有嵌套行元素的集合 对集合使用 <code>java.sql.Array</code> 和 <code>java.util.Collection</code> 类， 对嵌套行使用 <code>SQLData</code> 和 <code>Struct</code> 接口
<code>r1_t.java</code>	为命名行 <code>r1_t</code> 定义 <code>SQLData</code> 类
<code>r2_t.java</code>	为命名行 <code>r2_t</code> 定义 <code>SQLData</code> 类
<code>GenericStruct.java</code>	实例化 <code>java.sql.Struct</code> 对象以插入数据到命名或未命名行
<code>row1.java</code>	使用 <code>SQLData</code> 和 <code>Struct</code> 接口插入和选择一个简单的命名行
<code>row2.java</code>	使用 <code>SQLData</code> 和 <code>Struct</code> 接口插入和选择具有嵌套集合的命

示例程序名称	描述
	名行 SQLData 接口使用 GBase 8s IfmxComplexSQLOutput.writeObject() 和 IfmxComplexSQLOutput.readObject() 扩展方法写入和读取嵌套集合
row3.java	插入并选择具有嵌套集合的未命名行
fullname.java	使命名行 fullname_t 包含 SQLData 类 供 demo1.java 和 demo2.java 文件使用
person.java	使命名行 person_t 包含 SQLData 类 供 demo1.java 和 demo2.java 文件使用
demo1.java	将一个命名行访存到 SQLData 对象
demo2.java	将 SQLData 对象插入到一个命名行列
demo3.java	将一个未命名行列插入到 Struct 对象
demo4.java	将 Struct 对象插入到命名行列
demo5.java	将 GBase 8s SET 列访存到 java.util.HashSet 对象
demo6.java	将 GBase 8s SET 列访存到 java.util.TreeSet 对象 提供自定义类型映射以重写缺省值
demo7.java	将 java.util.HashSet 对象插入到 GBase 8s SET 列
demo8.java	将 GBase 8s SET 列访存到 java.sql.Array 对象
dropDB.java	删除数据库

proxy 目录中的示例

下表列出了 proxy 目录中的文件。目录中的 README 文件包含设置信息。

示例程序名称	描述
ProxySelect.java	（应用程序）创建样本数据库并使用四种场景连接到数据库： <ul style="list-style-type: none">• 使用代理服务器而不使用 LDAP 服务器连接• 使用 LDAP 服务器而不使用代理服务器连接• 使用 sqlhosts 文件连接

示例程序名称	描述
	<ul style="list-style-type: none">• 直接连接（无代理 servlet、sqlhosts 文件或 LDAP 服务器）
proxy.sh	（shell 脚本）启动 ProxySelect.java。要运行此脚本（和样本），请输入： <pre>proxy.sh -d ProxySelect -s 2</pre>
proxy.java	（小程序）从应用小程序执行与 ProxySelect.java 相同的操作。要运行此应用小程序，请输入： <pre>appletviewer proxy.html</pre>
proxy.html	proxy.java 的 HTML 文件
ifmx.conf	LDAP 配置文件示例
ifmx.ldif	LDAP ldif 文件示例

connection-pool 目录中的示例

下表列出了 connection-pool 目录中的文件。目录中的 README 文件包含设置信息。

示例程序名称	描述
AppSimulator.java	模拟多客户端线程进行 DataSource 连接
SetupDB.java	创建并填充一个样本数据库。请参阅示例运行命令的代码开头的注释。
DS_Pool.prop	列出连接池应用程序的属性
myCPDS.prop	列出连接池应用程序的属性，包括可选的调整属性
DS_no_Pool.prop	列出应用程序的属性，不包括连接池
Register.java	使用 JNDI Name 注册 DataSource 对象 运行命令的示例为： <pre>java Register DS_no_Pool /tmp</pre>
runDemo	（Shell 脚本）创建并填充一个样本数据库；注册数据源 DS_no_Pool 和 DS_Pool；运行应用程序模拟多个客户端线程连接样本数据库

xml 目录中的示例

下表列出了 xml 目录中的示例。

示例程序名称	描述
--------	----

示例程序名称	描述
CreateDB. java	创建一个样本数据库
makefile	编译示例
myHandler. java	SAX 解析程序回调例程类示例
sample1. xml	简单的 XML slide
sample2. xml	XML slide 的样例集
sample2. dtd	定义 sample1. xml 文档类型
xmldemo1. java	使用 XMLtoString(), getSource() 和 myHandler. java 将 sample1. xml 中的 XML 转换为一个 InputSource 对象, 然后使用 SAX 解析程序解析它。

hdr 目录中的示例

下表包含了 hdr 目录中的文件。目录中的 README 文件包含设置信息。

示例程序名称	描述
SetupDB. java	创建样本数据库和表
Register. java	使用 JNDI Name 注册器注册 DS_no_Pool 和 DS_Pool DataSource 对象。运行命令示例如下： <pre>java Register DS_no_Pool /tmp</pre>
AppSimulator. java	模拟使用 DataSource.getConnection() 方法进行池式和非池式连接的高可用数据复制重定向
HdrSimpleConnect. java	显示如何使用 DriverManager.getConnection() 方法实现 HDR 重定向

tools 目录中的示例

tools 目录包括以下子目录：

- udtudrmgr 目录，包括使用 UDT 和 UDR Manager 创建不透明类型和 UDR 的示例。
- classgenerator 目录，包括 ClassGenerator 实用程序的样本输出文件。

udtudrmgr 目录中的示例

下表包含了 udtudrmgr 目录中的文件。目录中的 README 文件包含设置信息。

示例程序名称	描述
--------	----

示例程序名称	描述
createDB. java	创建样本数据库
dropDB. java	删除样本数据库
Circle. java	（示例应用程序 1）使用缺省的 Input 和 Output 函数将 Java™ 类转换为 Java 不透明类型
PlayWithCircle. java	（示例应用程序 1）在客户端应用程序中使用 Circle 不透明类型
Circle2. java	（示例应用程序 2）使用用户提供的 Input 和 Output 函数将 Java 类转换为 Java 不透明类型
PlayWithCircle2. java	（示例应用程序 2）在客户端应用程序中使用 Circle2 不透明类型
MyCircle. java	（示例应用程序 3）在不预先存在 Java 类的情况下，创建固定长度不透明类型
Group1. java	（示例应用程序 4）将现有 Java 类中的方法映射到 Java UDR
PlayWithGroup1. java	（示例应用程序 4）在客户端应用程序中使用 Group1. java 的 UDR

10.2 DataSource 扩展

本节列出了标准 JDBC 类的 GBase 8s 扩展：

- **IfxDataSource** 类，实现 **DataSource** 接口
- **IfxConnectionPoolDataSource** 类，实现 **ConnectionPoolDataSource** 接口

有关如何以及为何使用 DataSource 或 ConnectionPoolDataSource 对象的信息，请参阅 JDBC 3.0 API。

GBase 8s JDBC Driver 提供以下用途的扩展：

- 读和写属性
- 获取和设置标准属性
- 获取和设置 GBase 8s 连接属性
- 获取和设置 Connection Pool DataSource 属性

10.2.1 读和写属性

以下方法在扩展的 **DataSource** 接口中定义，以便读取和写入属性。这些方法允许您通过编辑现有 **DataSource** 对象的属性列表来定义新的**DataSource** 对象。

```
public Properties getDsProperties();
```

返回包含在 **DataSource** 对象中的 **Property** 对象

```
public void addProp(String key, Object value);
```

添加属性到属性列表。

key 参数指定要添加的属性。

value 参数是属性的值。

```
public Object getProp(String key);
```

从属性列表获取属性的值。

key 参数指定要检索的属性。

```
public void removeProperty(String key);
```

从属性列表移除属性。

key 参数指定要移除的属性。

```
public void readProperties(InputStream in) throws IOException;
```

将属性从 **InputStream** 对象读取到 **DataSource** 对象。

in 参数是要读取属性的 **InputStream** 对象。

当输入流读取时若发生 **I/O** 错误，则会出现异常。

```
public void writeProperties(OutputStream out) throws IOException;
```

将 **DataSource** 对象的属性写入到 **OutputStream** 对象。

out 参数是要写入属性的 **OutputStream** 对象。

当写入到输出流时若发生 **I/O** 错误，则会出现异常。

10. 2. 2 获取和设置标准属性

以下方法在扩展的 **DataSource** 接口中定义，以便获取和设置 **JDBC 3.0 API** 中的属性。

属性	getXXX() 和 setXXX() 方法签名
portNumber	public int getPortNumber(); public void setPortNumber(int value);
databaseName	public String getDatabaseName();

属性	getXXX() 和 setXXX() 方法签名
	public void setDatabaseName(String value);
serverName	public String getServerName(); public void setServerName(String value);
user	public String getUser(); public void setUser(String value);
password	public String getPassword(); public void setPassword(String value);
description	public String getDescription(); public void setDescription(String value);
dataSourceName	public String getDataSourceName(); public void setDataSourceName(Stringvalue);

GBase 8s JDBC Driver 不支持 networkProtocol 和 roleName 属性。

10. 2. 3 获取和设置 GBase 8s 连接属性

以下方法在扩展的 DataSource 接口中定义，以便获取和设置 GBase 8s 环境变量值。

环境变量	getIfxXXX() 和 setIfxXXX() 方法签名
CLIENT_LOCALE	public String getIfxCLIENT_LOCALE() public void setIfxCLIENT_LOCALE(String value)
CSM	public String getIfxCSM() public void setIfxCSM(String csm)
DBANSIWARN	public boolean isIfxDBANSIWARN() public void setIfxDBANSIWARN(boolean value)
DBCENTURY	public String getIfxDBCENTURY() public void setIfxDBCENTURY(String value)
DBDATE	public String getIfxDBDATE() public void setIfxDBDATE(String value)
DB_LOCALE	public String getIfxDB_LOCALE() public void setIfxDB_LOCALE(String value)
DBSPACETEMP	public String getIfxDBSPACETEMP() public void setIfxDBSPACETEMP(String value)
DBTEMP	public String getIfxDBTEMP()

环境变量	<code>getIfxXXX()</code> 和 <code>setIfxXXX()</code> 方法签名
	<code>public void setIfxDBTEMP(String value)</code>
DBUPSPACE	<code>public String getIfxDBUPSPACE()</code> <code>public void setIfxDBUPSPACE(String value)</code>
DELIMIDENT	<code>public boolean isIfxDELIMIDENT()</code> <code>public void setIfxDELIMIDENT(boolean value)</code>
ENABLE_TYPE_CACHE	<code>public boolean isIfxENABLE_TYPE_CACHE()</code> <code>public void setIfxENABLE_TYPE_CACHE(boolean value)</code>
ENABLE_HDRSWITCH	<code>public boolean getIfxENABLE_HDRSWITCH()</code> <code>public void setIfxENABLE_HDRSWITCH(boolean value)</code>
FET_BUF_SIZE	<code>public int getIfxFET_BUF_SIZE()</code> <code>public void setIfxFET_BUF_SIZE(int value)</code>
GL_DATE	<code>public String getIfxGL_DATE()</code> <code>public void setIfxGL_DATE(String value)</code>
IFX_AUTOFREE	<code>public boolean isIfxIFX_AUTOFREE()</code> <code>public void setIfxIFX_AUTOFREE(boolean value)</code>
IFX_CODESETLOB	<code>public int getIfxIFX_CODESETLOB()</code> <code>public void setIfxIFX_CODESETLOB(int codesetlobFlag)</code>
IFX_DIRECTIVES	<code>public String getIfxIFX_DIRECTIVES()</code> <code>public void setIfxIFX_DIRECTIVES(String value)</code>
IFX_EXTDIRECTIVES	<code>public String getIfxIFX_EXTDIRECTIVES()</code> <code>public void setIfxIFX_EXTDIRECTIVES(String value)</code>
IFX_FLAT_UCSQ	<code>public int getIfxIFX_FLAT_UCSQ()</code> <code>public void setIfxIFX_FLAT_UCSQ(int value)</code>
IFX_GET_SMFLOAT_AS_FLOAT	<code>public boolean getIfxIFX_GET_SMFLOAT_AS_FLOAT()</code> <code>public void setIfxIFX_GET_SMFLOAT_AS_FLOAT(boolean value)</code>
IFX_ISOLATION_LEVEL	<code>public String getIfxIFX_ISOLATION_LEVEL()</code> <code>public void setIfxIFX_ISOLATION_LEVEL (String iso_level)</code>

环境变量	getIfxXXX() 和 setIfxXXX() 方法签名
IFX_LOCK_MODE_WAIT	public int getIfxIFX_LOCK_MODE_WAIT() public void setIfxIFX_LOCK_MODE_WAIT(int lock_time)
IFX_SET_FLOAT_AS_SMFLOAT	public boolean getIfxIFX_SET_FLOAT_AS_SMFLOAT() public void setIfxIFX_SET_FLOAT_AS_SMFLOAT(boolean value)
IFX_TRIMTRAILINGSPACES	public int getIfxIFX_TRIMTRAILINGSPACES() public void setIfxIFX_TRIMTRAILINGSPACES(int value)
IFXHOST	public String getIfxIFXHOST() public void setIfxIFXHOST(String value)
IFXHOST_SECONDARY	public String getIfxIFXHOST_SECONDARY() public void setIfxIFXHOST_SECONDARY(String value)
IFX_USEPUT	public boolean isIfxIFX_USEPUT() public void setIfxIFX_USEPUT(boolean value)
IFX_XASPEC	public String getIfxIFX_XASPEC() (returns y or n) public void IfxIFX_XASPEC(String XASPEC_flag) (only y, Y, n, or N are valid)
IFX_XASTDCOMPLIANCE_XAEND	public int getIfxIFX_XASTDCOMPLIANCE_XAEND() public void setIfxIFX_XASTDCOMPLIANCE_XAEND(int value)
GBASEDBTCONRETRY	public int getIfxGBASEDBTCONRETRY() public void setIfxGBASEDBTCONRETRY(int value)
GBASEDBTCONTIME	public int getIfxGBASEDBTCONTIME() public void setIfxGBASEDBTCONTIME(int value)
GBASEDBTOPCACHE	public String getIfxGBASEDBTOPCACHE() public void setIfxGBASEDBTOPCACHE(String value)
GBASEDBTSERVER_SECONDARY	public String getIfxGBASEDBTSERVER_SECONDARY() public void setIfxGBASEDBTSERVER_SECONDARY(String value)

环境变量	<code>getIfxXXX()</code> 和 <code>setIfxXXX()</code> 方法签名
GBASEDBTSTACKSIZE	<code>public int getIfxGBASEDBTSTACKSIZE()</code> <code>public void setIfxGBASEDBTSTACKSIZE(int value)</code>
JDBCTEMP	<code>public String getIfxJDBCTEMP()</code> <code>public void setIfxJDBCTEMP(String value)</code>
LDAP_IFXBASE	<code>public String getIfxLDAP_IFXBASE()</code> <code>public void setIfxLDAP_IFXBASE(String value)</code>
LDAP_PASSWD	<code>public String getIfxLDAP_PASSWD()</code> <code>public void setIfxLDAP_PASSWD(String value)</code>
LDAP_URL	<code>public String getIfxLDAP_URL()</code> <code>public void setIfxLDAP_URL(String value)</code>
LDAP_USER	<code>public String getIfxLDAP_USER()</code> <code>public void setIfxLDAP_USER(String value)</code>
LOBCACHE	<code>public int getIfxLOBCACHE()</code> <code>public void setIfxLOBCACHE(int value)</code>
NEWCODESET	<code>public String getIfxNEWCODESET()</code> <code>public void setIfxNEWCODESET(String value)</code>
NEWLOCALE	<code>public String getIfxNEWLOCALE()</code> <code>public void setIfxNEWLOCALE(String value)</code>
NEWNLSMAP	<code>public String getIfxNEWNLSMAP()</code> <code>public void setIfxNEWNLSMAP(String value)</code>
NODEFDAC	<code>public String getIfxNODEFDAC()</code> <code>public void setIfxNODEFDAC(String value)</code>
OPT_GOAL	<code>public String getIfxOPT_GOAL()</code> <code>public void setIfxOPT_GOAL(String value)</code>
OPTCOMPIND	<code>public String getIfxOPTCOMPIND()</code> <code>public void setIfxOPTCOMPIND(String value)</code>
OPTOFC	<code>public String getIfxOPTOFC()</code> <code>public void setIfxOPTOFC(String value)</code>
PATH	<code>public String getIfxPATH()</code> <code>public void setIfxPATH(String value)</code>
PDQPRIORITY	<code>public String getIfxPDQPRIORITY()</code> <code>public void setIfxPDQPRIORITY(String value)</code>
PORTNO_SECONDARY	<code>public String getIfxPORTNO_SECONDARY</code> <code>public void setIfxPORTNO_SECONDARY(int value)</code>

环境变量	getIfxXXX() 和 setIfxXXX() 方法签名
PROXY	public String getIfxPROXY() public void setIfxPROXY(String value)
PSORT_DBTEMP	public String getIfxPSORT_DBTEMP() public void setIfxPSORT_DBTEMP(String value)
PSORT_NPROCS	public String getIfxPSORT_NPROCS() public void setIfxPSORT_NPROCS(String value)
SECURITY	public String getIfxSECURITY() public void setIfxSECURITY(String value)
SQLH_FILE	public String getIfxSQLH_FILE() public void setIfxSQLH_FILE(String value)
SQLH_TYPE	public String getIfxSQLH_TYPE() public void setIfxSQLH_TYPE(String value)
SQLIDEBUG	public String getIfxSQLIDEBUG () public void setIfxSQLIDEBUG (String value)
STMT_CACHE	public String getIfxSTMT_CACHE() public void setIfxSTMT_CACHE(String value)

10. 2. 4 获取和设置连接池 DataSource 属性

编写使用 `ConnectionPoolDataSource` 对象的代码与编写使用 `DataSource` 对象的代码相同。附加的调整参数使您或您的数据库管理员使用连接池管理器控制连接池管理的某些方面。这些都在连接池中有完整的描述。下表总结了这些属性。

属性	getXXX() 和 setXXX() 方法签名
IFMX_CPM_ENABLE_SWITCH_HDRPOOL	public void setIfxCPMSwitchHDRPool (boolean flag) public int getIfxCPMSwitchHDRPool()
IFMX_CPM_INIT_POOLSIZE	public void setIfxCPMInitPoolSize (int init) public int getIfxCPMInitPoolSize()
IFMX_CPM_MAX_CONNECTIONS	public void setIfxCPMMaxConnections (int limit) public int getIfxCPMMaxConnections()
IFMX_CPM_MIN_POOLSIZE	public void setIfxCPMMinPoolSize (int min) public int getIfxCPMMinPoolSize()

属性	getXXX() 和 setXXX() 方法签名
IFMX_CPM_MAX_POOLSIZE	public void setIfxCPMMaxPoolSize (int max) public int getIfxCPMMaxPoolSize()
IFMX_CPM_MIN_AGELIMIT	public void setIfxCPMMinAgeLimit (long limit) public long getIfxCPMMinAgeLimit()
IFMX_CPM_MAX_AGELIMIT	public void setIfxCPMMaxAgeLimit (long limit) public long getIfxCPMMaxAgeLimit()
IFMX_CPM_SERVICE_INTERVAL	public void setIfxCPMServiceInterval (long interval) public long getIfxCPMServiceInterval()

10.3 映射数据类型

本节描述了在 Java™ 程序中定义的数据类型和 GBase 8s 数据库服务器支持的数据类型之间的映射问题。

10.3.1 在 GBase 8s 和 JDBC 数据类型之间映射的数据类型

由于每个数据库供应商支持的 SQL 数据类型之间存在差异，JDBC API 在类 java.sql.Types 中定义了一组通用 SQL 数据类型。使用这些 JDBC API 数据类型来引用 Java™ 程序中的通用 SQL 数据类型，这些程序使用 JDBC API 连接到 GBase 8s 数据库。

下表显示了每个 JDBC API 数据类型映射的 GBase 8s 数据类型。

JDBC API 数据类型	GBase 8s 数据类型
BIGINT	INT8 、 BIGINT 、 BIGSERIAL
BINARY	BYTE
BIT 1	BOOLEAN
REF	不支持
CHAR	CHAR (n)
DATE	DATE
DECIMAL	DECIMAL

JDBC API 数据类型	GBase 8s 数据类型
DOUBLE	FLOAT
FLOAT	FLOAT2
INTEGER	INTEGER
LONGVARBINARY	BYTE 或 BLOB
LONGVARCHAR	TEXT 或 CLOB
NUMERIC	DECIMAL
NUMERIC	MONEY
REAL	SMALLFLOAT
SMALLINT	SMALLINT
TIME	DATETIME HOUR TO SECOND2
TIMESTAMP	DATETIME YEAR TO FRACTION(5)3
TINYINT	SMALLINT
VARBINARY	BYTE
VARCHAR	VARCHAR(m, r)
BOOLEAN	BOOLEAN
SMALLINT	SMALLINT

- 1 在 Java 1.4 中，java.sql.Types.BOOLEAN 映射到 BOOLEAN。
- 2 这个映射是 JDBC 兼容的。通过将 IFX_SET_FLOAT_AS_SMFLOAT 环境变量设置为 1，可以将 JDBC FLOAT 数据类型映射到 GBase 8sSMALLFLOAT 数据类型以便向后兼容。
- 3 GBase 8s DATETIME 类型非常严格，不可互换。有关更多信息，请参阅 字段长度和 DATETIME 数据。

在扩展类型和 Java 和 JDBC 类型之间映射数据类型

下表列出了 GBase 8s 中扩展的数据类型和对应 Java™ 和 JDBC 的类型之间的映射。

JDBC 类型	Java 对象类型	GBase 8s 类型
java. sql. Types. LONGVARCH AR	java. sql. String java. io. inputStream	LVARCHAR IfxTypes. IFX_TYPE_LVARCHA R
java. sql. Types. JAVA_OBJE CT	java. sql. SQLData	Opaque 类型 IfxTypes. IFX_TYPE_UDTFIXE

JDBC 类型	Java 对象类型	GBase 8s 类型
		D IfxTypes. IFX_TYPE_UDTVAR
java.sql.Types. LONGVARBI NARY java.sql.Types. BLOB	java.sql.Blob java.io.inputStreambyte[]	BLOB IfxTypes. IFX_TYPE_BLOB
java.sql.Types. LONGVARCH AR java.sql.Types. CLOB	java.sql.Clob java.io.inputStream java.lang.String	CLOB IfxTypes. IFX_TYPE_CLOB
java.sql.Types. LONGVARBI NARY java.sql.Types. BLOB	java.io.inputStream java.sql.Blob byte[]	BYTE IfxTypes. IFX_TYPE_BYTE
java.sql.Types. LONGVARCH AR java.sql.Types. CLOB	java.io.InputStream java.sql.Clob java.sql.String	TEXT IfxTypes. IFX_TYPE_TEXT
java.sql.Types. JAVA_OBJE CT java.sql.Types. STRUCT	java.sql.SQLData java.sql.Struct	命名行 IfxTypes. IFX_TYPE_ROW
java.sql.Types. STRUCT	java.sql.Struct	未命名行 IfxTypes. IFX_TYPE_ROW
java.sql.Types. ARRAY java.sql.Types. OTHER	java.sql.Array java.util.LinkedList java.util.HashSet java.util.TreeSet	set, multiset IfxTypes. IFX_TYPE_SET IfxTypes. IFX_TYPE_MULTISE T
java.sql.Types. ARRAY java.sql.Types. OTHER	java.sql.Array java.util.ArrayList ja va.util.LinkedList	LIST IfxTypes. IFX_TYPE_LIST

Java boolean 对象可以映射到 GBase 8s smallint 数据类型或 GBase 8s boolean 数据类型。GBase 8s JDBC Driver 尝试根据列类型映射它。但是，在例如 PreparedStatement 主机变量的情况中，GBase 8s JDBC Driver 不会存取列类型，因此，映射在一定程度上是受限制的。有关数据类型映射的详细信息，请参阅PreparedStatement.setXXX() 扩展的数据类型映射。

C 不透明类型和 Java 之间的数据类型映射

要使用 Java™ 创建不透明类型,您可以使用 UDT 和 UDR Manager 工具。有关更多信息，请参阅 与不透明数据类型一起使用。

所有的不透明类型都存储在 C 结构的数据库服务器表中, 该结构由不透明类型中定义的各种 DataBlade API 类型组成。（有关更多信息，请参阅《GBase 8s DataBlade API 程序员指南》。）

下表列出了 DataBlade API 类型与相应的 Java 类型的映射。

DataBlade API 类型	Java 类型
MI_LO_HANDLE	BLOB 或 CLOB
gl_wchar_t	String
mi_boolean	boolean
mi_char	String
mi_char1	String
mi_date	Date
mi_datetime	TimeStamp
mi_decimal	BigDecimal
mi_double_precision	double
mi_int1	byte
mi_int8	long
mi_integer	int
mi_interval	不支持
mi_money	BigDecimal
mi_numeric	BigDecimal
mi_real	float
mi_smallint	short
mi_string	String
mi_unsigned_char1	String
mi_unsigned_int8	long
mi_unsigned_integer	int
mi_unsigned_smallint	short
mi_wchar	String

C 结构可能包含填充字节。GBase 8s JDBC Driver 会自动跳过这些填充字节，以确保下一个数据成员正确对齐。因此，您的 Java™ 对象不必关心自身的对齐。

10. 3. 2 PreparedStatement.setXXX() 扩展的数据类型映射

GBase 8s 介绍许多扩展数据类型。因此在 JDBC 或 Java™ 数据类型和 GBase 8s 数据类型之间会有多个映射。

例如，可以使用 `PreparedStatement.setAsciiStream()` 插入 TEXT 列或 CLOB 列。同样，还可以使用 `PreparedStatement.setBinaryStream()` 插入到 BYTE 列或 BLOB 列。由于实际的列信息始终对 GBase 8s JDBC Driver 不可用，因此驱动程序在映射数据类型时可能会产生歧义。

通常，在 INSERT、SELECT 或 DELETE 语句中，列信息对驱动程序是可用的，因此驱动程序可以确定如何将数据发送到数据库服务器。

但是，当在 UPDATE 语句或 WHERE 子句中引用数据时，GBase 8s JDBC Driver 不能访问列信息。在这些情况下，除非您使用 GBase 8s 扩展，否则驱动程序使用在 GBase 8s 和 JDBC 数据类型之间映射的数据类型中第一个表列出的相应的 GBase 8s 数据类型映射这些列。对于 `PreparedStatement.setAsciiStream()` 方法，该驱动程序尝试映射到 TEXT 数据类型，对于 `PreparedStatement.setBinaryStream()` 方法，它尝试映射到 BYTE 数据类型。

映射扩展

要指示驱动程序映射到某个特定的数据类型（因此 UPDATE 语句和 WHERE 子句中没有歧义），您可使用扩展 `PreparedStatement.setXXX()` 方法。可能具有歧义的数据类型只有 `boolean`、`lvvarchar`、`text`、`byte`、`BLOB` 和 `CLOB`。

要使用这些扩展方法，您必须将 `PreparedStatement` 引用转换为 `IfmxPreparedStatement`。例如，以下代码将语句变量 `p_stmt` 转换为 `IfmxPreparedStatement`，以调用 `IfxSetObject()` 方法，并将文件的内容作为 CLOB 类型的大对象插入。`IfxSetObject()` 定义为 I:

```
public void IfxSetObject(int i, Object x, int scale, int ifxType)
    throws SQLException
public void IfxSetObject(int i, Object x, int ifxType) throws
    SQLException
```

代码为：

```
File file = new File("sblob_06.dat");
int fileLength = (int)file.length();
byte[] buffer = new byte[fileLength];
FileInputStream fin = new FileInputStream(file);
fin.read(buffer,0,fileLength);
String str = new String(buffer);

writeOutputFile("Prepare");
PreparedStatement p_stmt = myConn.prepareStatement
    ("insert into sblob_t20(c1) values(?)");

writeOutputFile("IfxSetObject");
```

```
((IfmxPreparedStatement)p_stmt).IfxSetObject(1,str,30,IfxTypes.IFX  
_TYPE_CLOB);
```

对于 **IfmxPreparedStatement.IfxSetObject** 扩展，不能简单地使用添加的 ifxType 参数重载方法签名。相反，您必须将该方法命名为 **IfxSetObject**。

不透明类型的扩展

处理不透明类型的扩展允许您的应用程序指定数据库服务器在将其返回给客户端之前将其转换为不透明类型的返回类型。这被称为预先绑定返回值。方法是：

- **setBindColType()**，它允许应用程序使用 `java.sql.Types` 的标准 JDBC 数据类型指定结果集值的输出类型。
- **setBindColIfxType()**，允许应用程序使用来自 `com.gbasedbt.lang.IfxTypes` 的 GBase 8s 数据类型指定结果集的输出类型

有关变量类型的更多信息，请参阅 `IfxTypes` 类。

- **clearBindColType()**，它重置通过前两个方法设置的值

在以下主题中：

- **colIndex** 参数指定列：1 是第一列，2 是第二列，依此类推。
- **sqltype** 参数是来自 `java.sql.Types` 的值：例如，`Types.INTEGER`。
- **ifxtype** 参数是来自 `IfxTypes` 的值：例如，`IfxTypes.IFX_TYPE_DECIMAL`。

setBindColType() 方法

方法如下所示：

```
public void setBindColType(int colIndex, int sqltype) throws SQLException;  
public void setBindColType(int colIndex, int sqltype, int scale)  
throws SQLException;  
public void setBindColType(int colIndex, int sqltype, String name)  
throws SQLException;
```

第一个重载方法允许应用程序指定不透明类型为 `java.sql.DECIMAL` 或 `java.sql.NUMERIC`；`scale` 参数指定小数点右边的位数。第二个重载方法允许应用程序通过将这些值分配给 `name` 参数，来指定不透明类型为 `java.sql.STRUCT`、`java.sql.ARRAY`、`java.sql.DISTINCT` 或 `java.sql.JAVA_OBJECT`。

setBindColIfxType() 方法

方法如下所示：

```
public void setBindColIfxType(int colIndex, int ifxtype) throws SQLException;  
public void setBindColIfxType(int colIndex, int ifxtype, int scale)
```

```
throws SQLException;  
public void setBindCollfxType(int collIndex, int ifxtype, String name)  
throws SQLException ;
```

第一个重载方法允许应用程序指定输出类型

为 IFX_TYPE_DECIMAL 或 IFX_TYPE_NUMERIC；scale 参数指定小数点右边的位数。第

二个重载方法允许应用程序通过将其中之一的值分配给 name 参数来指定输出类型

为 IFX_TYPE_LIST、IFX_TYPE_ROW、IFX_TYPE_MULTISSET、IFX_TYPE_SET、
IFX_TYPE_UDTVAR 或 IFX_TYPE_UDTFIXED。

clearBindColType() 方法

方法如下所示：

```
public void clearBindColType() throws SQLException;
```

Prebinding 示例

以下示例代码来自 udt_bindCol.java 样本程序，它预先将不透明类型绑定到 GBase 8s VARCHAR，然后绑定到标准的 Java™ Integer 类型。此示例中使用的表具有一个 int 列和一个不透明类型列，如下所示：

```
create table charattr_tab (int_col int, charattr_col charattr_udt)
```

在 charattr_col 列中选择和预绑定不透明类型的代码如下所示：

```
String s = "select int_col, charattr_col as cast_udt_to_lvc, " +  
"charattr_col as cast_udt_to_int from charattr_tab order by 1";  
  
pstmt = conn.prepareStatement(s);  
  
((IfxPreparedStatement)pstmt).setBindCollfxType(2, IfxTypes.IFX_TYPE_LVARCH  
AR);  
((IfxPreparedStatement)pstmt).setBindColType(3, Types.INTEGER);  
  
ResultSet rs = pstmt.executeQuery();  
  
System.out.println("Fetching data ...");  
int curRow = 0;  
while (rs.next())  
{  
    curRow++;  
    int intret = rs.getInt("int_col");
```

```
String strret = rs.getString("cast_udt_to_lvc");  
int intret2 = rs.getInt("cast_udt_to_int");  
} // end while
```

其它映射扩展

以下将列出剩余的方法与其应用条件。在每种情况中，GBase 8s 类型必须是标准 JDBC PreparedStatement.setXXX() 接口的最后一个参数。

IfmxPreparedStatement.setArray()

```
public void setArray(int parameterIndex, Array x, int ifxType)  
    throws SQLException
```

IfmxPreparedStatement.setAsciiStream()

```
public void setAsciiStream(int i, InputStream x, int length, int  
    ifxType) throws SQLException
```

当应用程序将一个很大的 ASCII 值插入到 LONGVARCHAR 列中时，使用 java.io.InputStream 会更高效地将 ASCII 值发送到服务器。

IfmxPreparedStatement.setBigDecimal()

```
public void setBigDecimal(int i, BigDecimal x, int ifxType)  
    throws SQLException
```

IfmxPreparedStatement.setBinaryStream()

```
public void setBinaryStream(int i, InputStream x, int length, int  
    ifxType) throws SQLException
```

当应用程序插入一个很大的二进制值到 LONGVARbinary 列时，使用 java.io.InputStream 将会更高效地将二进制值发送到服务器。

IfmxPreparedStatement.setBlob()

```
public void setBlob(int parameterIndex, Blob x, int ifxType)  
    throws SQLException
```

IfmxPreparedStatement.setBoolean()

```
public void setBoolean(int i, boolean x, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setByte()

```
public void setByte(int i, byte x, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setBytes()

```
public void setBytes(int i, byte x[], int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setCharacterStream()

```
public void setCharacterStream(int parameterIndex, Reader reader,  
    int length, int ifxType) throws SQLException
```

当应用程序将 LONGVARCHAR 参数设置为一个很大的 UNICODE 值时，使用 java.io.Reader 会更高效率地将 UNICODE 值发送到服务器。

IfmxPreparedStatement.setClob()

```
public void setClob(int parameterIndex, Clob x, int ifxType)  
    throws SQLException
```

IfmxPreparedStatement.setDate()

```
public void setDate(int i, Date x, int ifxType) throws  
    SQLException  
public void setDate(int parameterIndex, Date x, Calendar Cal,  
    int ifxType) throws SQLException
```

IfmxPreparedStatement.setDouble()

```
public void setDouble(int i, double x, int ifxType) throws SQ  
    LException
```

IfmxPreparedStatement.setFloat()

```
public void setFloat(int i, float x, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setInt()

```
public void setInt(int i, int x, int ifxType) throws SQLException
```

IfmxPreparedStatement.setLong()

```
public void setLong(int i, long x, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setNull()

```
public void setNull(int i, int sqlType, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setShort()

```
public void setShort(int i, short x, int ifxType) throws  
    SQLException
```


IfmxPreparedStatement.setString()

```
public void setString(int i, String x, int ifxType) throws
    SQLException
```

IfmxPreparedStatement.setTime()

```
public void setTime(int i, Time x, int ifxType) throws
    SQLException
public void setTime(int parameterIndex, Time time, Calendar Cal,
    int ifxType) throws SQLException
```

IfmxPreparedStatement.setTimestamp()

```
public void setTimestamp(int i, Timestamp x, int ifxType) throws
    SQLException
public void setTimestamp(int parameterIndex, Timestamp x, Calendar
    Cal) throws SQLException
```

IfxTypes 类

扩展的 IfmxPreparedStatement 方法要求您传入要映射到其中的 GBase 8s 数据类型。这些类型是 com.gbasedbt.lang.IfxTypes 类的一部分。

下表显示了 IfxTypes 常量和对应的 GBase 8s 数据类型。

IfxTypes 常量	GBase 8s 数据类型
IfxTypes. IFX_TYPE_BIGINT	BIGINT
IfxTypes. IFX_TYPE_BIGSERIAL	BIGSERIAL
IfxTypes. IFX_TYPE_CHAR	CHAR
IfxTypes. IFX_TYPE_SMALLINT	SMALLINT
IfxTypes. IFX_TYPE_INT	INT
IfxTypes. IFX_TYPE_FLOAT	FLOAT
IfxTypes. IFX_TYPE_SMFLOAT	SMALLFLOAT
IfxTypes. IFX_TYPE_DECIMAL	DECIMAL
IfxTypes. IFX_TYPE_SERIAL	SERIAL
IfxTypes. IFX_TYPE_DATE	DATE
IfxTypes. IFX_TYPE_MONEY	MONEY
IfxTypes. IFX_TYPE_NULL	NULL

IfxTypes 常量	GBase 8s 数据类型
IfxTypes. IFX_TYPE_DATETIME	DATETIME
IfxTypes. IFX_TYPE_BYTE	BYTE
IfxTypes. IFX_TYPE_TEXT	TEXT
IfxTypes. IFX_TYPE_VARCHAR	VARCHAR
IfxTypes. IFX_TYPE_INTERVAL	INTERVAL
IfxTypes. IFX_TYPE_NCHAR	NCHAR
IfxTypes. IFX_TYPE_NVARCHAR	NVARCHAR
IfxTypes. IFX_TYPE_INT8	INT8
IfxTypes. IFX_TYPE_SERIAL8	SERIAL8
IfxTypes. IFX_TYPE_SET	SQLSET
IfxTypes. IFX_TYPE_MULTISSET	SQLMULTISSET
IfxTypes. IFX_TYPE_LIST	SQLLIST
IfxTypes. IFX_TYPE_ROW	SQLROW
IfxTypes. IFX_TYPE_COLLECTION	COLLECTION
IfxTypes. IFX_TYPE_UDTVAR	UDTVAR
IfxTypes. IFX_TYPE_UDTFIXED	UDTFIXED
IfxTypes. IFX_TYPE_REFSER8	REFSER8
IfxTypes. IFX_TYPE_LVARCHAR	LVARCHAR
IfxTypes. IFX_TYPE_SENDRECV	SENDRECV
IfxTypes. IFX_TYPE_BOOL	BOOLEAN
IfxTypes. IFX_TYPE_IMPEXP	IMPEXP
IfxTypes. IFX_TYPE_IMPEXPBIN	IMPEXPBIN
IfxTypes. IFX_TYPE_CLOB	CLOB
IfxTypes. IFX_TYPE_BLOB	BLOB

扩展总结

本节中的表列出了 GBase 8s JDBC Driver 支持的非扩展数据类型和 GBase 8s 扩展数据类型的 PreparedStatement.setXXX() 方法。

非扩展数据类型

下表列出了 GBase 8s JDBC Driver 支持的非扩展数据类型的 `PreparedStatement.setXXX()` 方法。首先列出了 `java.sql.Types` 类中定义的标准 JDBC API 数据类型。它们会转换为特定的 GBase 8s 数据类型，如在扩展类型和 Java 和 JDBC 类型之间映射数据类型中的表所示。下一个表列出了可用于写入特定 JDBC API 数据类型的数据的 `setXXX()` 方法。大写和粗体 **X** 表示推荐与 GBase 8s JDBC Driver 一起使用的 `setXXX()` 方法；小写 **x** 表示 GBase 8s JDBC Driver 支持的其它方法。

数值型 JDBC API 数据类型

表 1. 来自 `java.sql.Types` 的数值型 JDBC API 数据类型

setXXX() 方法	TINYINT	SMALLINT	INTEGER	BIGINT
setByte()	X	x	x	x
setShort()	x	X	x	x
setInt()	x	x	X	x
setLong()	x	x	x	X
setFloat()	x	x	x	x
setDouble()	x	x	x	x
setBigDecimal()	x	x	x	x
setBoolean()	x	x	x	x
setString()	x	x	x	x
setObject()	x	x	x	x

表 2. 来自 `java.sql.Types` 的数值型 JDBC API 数据类型（续）

setXXX() 方法	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC
setByte()	x	x	x	x	x
setShort()	x	x	x	x	x
setInt()	x	x	x	x	x
setLong()	x	x	x	x	x
setFloat()	X	x	x	x	x
setDouble()	x	X	X	x	x
setBigDecimal()	x	x	x	X	X
setBoolean()	x	x	x	x	x
setString()	x	x	x	x	x
setObject()	x	x	x	x	x

字符型和时间型 JDBC API 数据类型

表 3. 来自 java.sql.Types 的字符型和时间型 JDBC API 数据类型

setXXX() 方法	CHAR	VARCHAR	LONGVARCHAR	BINARY
setByte()	x1	x1		
setShort()	x1	x1		
setInt()	x1	x1		
setLong()	x1	x1		
setFloat()	x1	x1		
setDouble()	x1	x1		
setBigDecimal()	x	x		
setBoolean()	x	x		
setString()	X	X	x	x
setBytes()			x	X
setDate()	x	x		
setTime()	x	x		
setTimestamp()	x	x		
setAsciiStream()			X	x
setCharacterStream()			X	x
setBinaryStream()			x	x
setObject()	x	x	x2	x

注：

1. 列值必须精确符合 setXXX() 的类型，否则发出 SQLException。如果列值不在允许的范围
- 内，则 setXXX() 方法发出异常并不会转换数据。例如，如果正在写入的值是1000，则 setByte(1) 会引发 SQLException。
2. 写入一个字节数组。

表 4. 来自 java.sql.Types 的字符型和时间型 JDBC API 数据类型（续）

setXXX() 方法	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
setString()	x	x	x	x	x
setBytes()	X	x			

setXXX() 方法	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
setDate()			X		x
setTime()				X	x
setTimestamp()			x		X
setAsciiStream()	x	x			
setCharacterStream() ()	x	x			
setBinaryStream()	x	X			
setObject()	x	x1	x	x2	x

- 注：
- 1. 写入字节数组。
 - 2. 写入 Timestamp 对象而不是 Time 对象。

setMaxRows() 方法写入 SQL 空值。

GBase 8s 扩展数据类型

下表列出了 GBase 8s JDBC Driver 支持的 GBase 8s 扩展数据类型
的 PreparedStatement.setXXX() 方法，其之间的映射在表在扩展类型和 Java 和 JDBC 类型
之间映射数据类型中展示。该表列出了您可以用于写入的特定扩展数据类型的数据
的 setXXX() 方法。

大写并加粗的 X 表示推荐使用的 setXXX() 方法；小写的 x 表示 GBase 8s JDBC Driver 支持的
其它 setXXX() 方法。该表不包含您不能使用的GBase 8s 扩展数据类型的 setXXX() 方法。

表 1. GBase 8s 扩展数据类型

setXXX() 方法	BOOLEAN	LVARCHAR	Opaque 类型	BLOB	CLOB	BYTE	TEXT
setByte()	x	x					
setShort()	x						
setInt()	x						
setBoolean()	X						
setString()		X			x		x
setBytes()				x		x	
setAsciiStream()		x			x		X
setCharacterStream(()		x			x		X

setXXX() 方法	BOOLEAN	LVARCHAR	Opaque 类型	BLOB	CLOB	BYTE	TEXT
)							
setBinaryStream()	x			x		X	
setObject()	x	x	X	x	x	x	x
setArray()							
setBlob()				X			
setClob()					X		

表 2. GBase 8s 扩展数据类型（续）

setXXX() 方法	NAMED ROW	UNNAMED ROW	SET 或 MULTISET	LIST
setObject()	X	X	x	x
setArray()			x	x

setMaxRows() 方法写入一个 SQL 空值。

10. 3. 3 ResultSet.getXXX() 方法的数据类型映射

使用 ResultSet.getXXX() 方法将数据从使用 JDBC API 连接的 GBase 8s 数据库传输到 Java™ 程序。例如，使用 ResultSet.getString() 方法获取 LVARCHAR 数据类型列中存储的信息。

重要：如果您在 SQL 语句中使用表达式（例如，SELECT mytype::LVARCHAR FROM mytab），您可能不能使用 ResultSet.getXXX(columnName) 检索值。请改为使用 ResultSet.getXXX(columnIndex) 来检索该值。

如果检索的列值是 SQL 空值，则 getXXX() 方法返回空值。

本节中的表列出了 GBase 8s JDBC Driver 支持的非扩展数据类型和 GBase 8s 扩展数据类型的 ResultSet.getXXX() 方法。

非扩展数据类型

下表列出了 GBase 8s JDBC Driver 支持的非扩展数据类型的 ResultSet.getXXX() 方法。首先列出了 java.sql.Types 类中定义的标准 JDBC API 数据类型。它们会转换为特定的 GBase 8s 数据类型，如在扩展类型和 Java 和 JDBC 类型之间映射数据类型中的表所示。该表列出了可用于检索特定的 JDBC API 数据类型的数据的 getXXX() 方法。

大写和粗体 X 表示推荐与 GBase 8s JDBC Driver 一起使用的 getXXX() 方法；小写 x 表示 GBase 8s JDBC Driver 支持的其它 getXXX() 方法。

数值型 JDBC API 数据类型

表 1. 来自 java.sql.Types 的数值型 JDBC API 数据类型

getXXX() 方法	TINYINT	SMALLINT	INTEGER	BIGINT
getByte()	X	x	x	x
getShort()	x	X	x	x
getInt()	x	x	X	x
getLong()	x	x	x	X
getFloat()	x	x	x	x
getDouble()	x	x	x	x
getBigDecimal()	x	x	x	x
getBoolean()	x	x	x	x
getString()	x	x	x	x
getObject()	x	x	x	x

表 2. 数值型 JDBC API 数据类型 （续）

getXXX() 方法	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC
getByte()	x	x	x	x	x
getShort()	x	x	x	x	x
getInt()	x	x	x	x	x
getLong()	x	x	x	x	x
getFloat()	X	x	x	x	x
getDouble()	x	X	X	x	x
getBigDecimal()	x	x	x	X	X
getBoolean()	x	x	x	x	x
getString()	x	x	x	x	x
getObject()	x	x	x	x	x

字符型和时间型 JDBC API 数据类型

表 3. 来自 java.sql.Types 的字符型和时间型 JDBC API 数据类型

getXXX() 方法	CHAR	VARCHAR	LONGVARCHAR	BINARY
getByte()	x1	x1		
getShort()	x1	x1		

getXXX() 方法	CHAR	VARCHAR	LONGVARCHAR	BINARY
getInt()	x1	x1		
getLong()	x1	x1		
getFloat()	x1	x1		
getDouble()	x1	x1		
getBigDecimal()	x	x		
getBoolean()	x	x		
getString()	X	X	x	x
getBytes()			x	X
getDate()	x	x		
getTime()	x	x		
getTimestamp()	x	x		
getAsciiStream()			X	x
getCharacterStream()			X	x
getBinaryStream()			x	x
getObject()	x	x	x2	x

注:

- 1. 列值必须精确符合 getXXX() 类型，否则会发出 SQLException。如果列值不在允许的范围内，则 setXXX() 方法发出异常并不会转换数据。例如，如果列值是 1000，则 getByte(1) 会引发 SQLException。
- 2. 返回一个字节数组。

表 4. 来自 java.sql.Types 的字符型和时间型 JDBC API 数据类型（续）

getXXX() 方法	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getString()	x	x	x	x	x
getBytes()	X	x			
getDate()			X		x
getTime()				X	x
getTimestamp()			x		X
getAsciiStream()	x	x			

getXXX() 方法	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getCharacterStream()	x	x			
getBinaryStream()	x	X			
getObject()	x	x1	x	x2	x

注：

- 1. 返回一个字节数组。
- 2. 返回一个 Timestamp 对象而不是 Time 对象。

GBase 8s 扩展数据类型

下表列出了 GBase 8s JDBC Driver 支持的 GBase 8s 扩展数据类型
的 PreparedStatement.getXXX() 方法，其之间的映射在表在扩展类型和 Java 和 JDBC 类型
之间映射数据类型中展示。该表列出了您可以用于检索的特定扩展数据类型的数据
的 getXXX() 方法。

大写并加粗的 X 表示推荐使用的 getXXX() 方法；小写的 x 表示 GBase 8s JDBC Driver 支持的
其它 getXXX() 方法。该表不包含您不能使用的GBase 8s 扩展数据类型的 getXXX() 方法。

表 1. GBase 8s 扩展数据类型

getXXX() 方法	BOOLEAN	LVARCHAR	Opaque 类型	BLOB	CLOB	BYTE
getByte()	x	x				
getShort()	x					
getInt()	x					
getBoolean()	X					
getString()		X			x	
getBytes()				x		x
getAsciiStream()		x			x	
getCharacterStream()		x			x	
getBinaryStream()	x			x		X
getObject()	x	x	X	x	x	x
getBlob()				X		
getClob()					X	

表 2. GBase 8s 扩展的数据类型（续）					
getXXX() 方法	TEXT	NAMED ROW	UNNAMED ROW	SET 或 MULTISET	LIST
getString()	x				
getBytes()					
getAsciiStream()	X				
getCharacterStream()	X				
getBinaryStream()					
getObject()	x	X	X	x	x
getArray()				x	x
getBlob()					
getClob()					

10. 3. 4 UDT manager 和 UDR manager 的数据类型映射

当使用 UDTManager 和 UDRManager 类在数据库中创建不透明类型和 Java™ UDR 时，驱动程序映射 Java 方法参数并根据本节中的表返回类型的 SQL数据类型。不支持未在这些表中显示的任何数据类型。

如果 Java™ 方法具有以下任何 Java 类型的参数，则参数和返回类型将映射到服务器中的 SQL 类型，如下表所示。该表显示每个 Java 数据类型映射到的 GBase 8s 数据类型。

Java 数据类型	SQL 数据类型
boolean, java.lang.Boolean	BOOLEAN
char	CHAR(1)
byte	CHAR(1)
short, java.lang.Short	SMALLINT
int, java.lang.Integer	INT
long, java.lang.Long	INT8
float, java.lang.Float	SMALLFLOAT
double, java.lang.Double	FLOAT1
java.lang.String	LVARCHAR
java.math.BigDecimal	DECIMAL

Java 数据类型	SQL 数据类型
	缺省精度由服务器设置：ANSI 数据库为 DECIMAL(16,0) ； 非 ANSI 数据库为 decimal (16,255)
java.sql.Date	DATE
java.sql.Time	DATETIME HOUR TO SECOND
java.sql.Timestamp	DATETIME YEAR TO FRACTION(5)
com.gbasedbt.lang.IntervalY M	INTERVAL YEAR TO MONTH
com.gbasedbt.lang.IntervalD F	INTERVAL DAY TO FRACTION(5)
java.sql.Blob	BLOB
java.sql.Clob	CLOB

1 此映射是 JDBC 兼容的。通过将 IFX_GET_SMFLOAT_AS_FLOAT 环境变量设置为 1，
可以将 Java double 数据类型（通过 JDBC FLOAT 数据类型）映射到 GBase
8s SMALLFLOAT 数据类型以实现向后兼容。

映射转换类型

下表显示了在 UDTMetaData.setXXXCast() 方法中为 ifxtype 参数定义的类型与服务器中的
SQL 数据类型之间支持的映射。

来自 com.gbasedbt.lang.IfxTypes 的 ifxtype 参数类型	GBase 8s 数据类型
IFX_TYPE_CHAR	CHAR
IFX_TYPE_SMALLINT	SMALLINT
IFX_TYPE_INT	INT
IFX_TYPE_FLOAT	FLOAT
IFX_TYPE_SMFLOAT	SMALLFLOAT
IFX_TYPE_DECIMAL	DECIMAL
IFX_TYPE_SERIAL	SERIAL
IFX_TYPE_DATE	DATE
IFX_TYPE_MONEY	MONEY
IFX_TYPE_DATETIME	DATETIME

来自 com.gbasedbt.lang.IfxyTypes 的 ifxtype 参数类型	GBase 8s 数据类型
IFX_TYPE_BYTE	BYTE
IFX_TYPE_TEXT	TEXT
IFX_TYPE_VARCHAR	VARCHAR
IFX_TYPE_INTERVAL	INTERVAL
IFX_TYPE_NCHAR	NCHAR
IFX_TYPE_NVARCHAR	NVARCHAR
IFX_TYPE_INT8	INT8
IFX_TYPE_SERIAL8	SERIAL8
IFX_TYPE_LVARCHAR	LVARCHAR
IFX_TYPE_SENDRECV	SENDRECV
IFX_TYPE_BOOL	BOOLEAN
IFX_TYPE_IMPEXP	IMPEXP
IFX_TYPE_IMPEXPBIN	IMPEXPBIN
IFX_TYPE_CLOB	CLOB
IFX_TYPE_BLOB	BLOB

映射字段类型

下表显示了在 UDTMetaData.setFieldType() 方法中为 ifxtype 参数定义的类型与在 Java™ 类文件中出现的 Java 数据类型之间支持的映射。未在此表中显示的数据类型在不透明类型中不支持。

来自 com.gbasedbt.lang.IfxyTypes 的 ifxtype 参数类型	Java 数据类型
IFX_TYPE_BIGINT	long
IFX_TYPE_BIGSERIAL	long
IFX_TYPE_CHAR	java.lang.String
IFX_TYPE_SMALLINT	short
IFX_TYPE_INT	int
IFX_TYPE_FLOAT	double

来自 com.gbasedbt.lang.IfTypes 的 ifxtype 参数类型	Java 数据类型
IFX_TYPE_SMFLOAT	float1
IFX_TYPE_DECIMAL	java.lang.BigDecimal
IFX_TYPE_SERIAL	int
IFX_TYPE_DATE	Date
IFX_TYPE_MONEY	java.lang.BigDecimal
IFX_TYPE_DATETIME	如果起始限定符为年、月或日，则为 java.lang.Timestamp ; 否则为 java.lang.Time（请 参阅 字段长度和 DATETIME 数据）。
IFX_TYPE_INTERVAL	如果起始限定符为年、月或日，则为 com.gbasedbt.lang.IfIntervalYM; 否则为 com.gbasedbt.lang.IfIntervalDF（请参阅 字段 长度和 DATETIME 数据）。
IFX_TYPE_NCHAR	java.lang.String
IFX_TYPE_INT8	long
IFX_TYPE_SERIAL8	long
IFX_TYPE_BOOL	boolean
IFX_TYPE_CLOB	java.sql.Clob
IFX_TYPE_BLOB	java.sql.Blob

1 此映射是 JDBC 兼容的。通过将 IFX_GET_SMFLOAT_AS_FLOAT 环境变量设置为 1，将 IFX_TYPE_SMFLOAT 数据类型（通过 JDBC FLOAT 数据类型）映射到 Java double 数据类型以实现向后兼容。

字段长度和 DATETIME 数据

当通过调用 setFieldType(IFX_TYPE_DATETIME) 或 setFieldType(IFX_TYPE_INTERVAL) 将字段类型设置为 date-time 或 interval 数据类型时，驱动程序会将 date-time 字段映射为 java.sql.Timestamp 或 java.sql.Time，这取决于调用 setFieldLength() 时设置的编码长度。

例如，给出一个 date-time 字段的标准格式为 YYYY-MM-DD HH:MM:SS，驱动程序会使用以下映射算法：

- 如果编码长度具有 hour 或更小的起始码，则会映射为 java.sql.Time。
- 如果编码长度具有 year 或更小的起始码，则会映射为 java.sql.TimeStamp。

对于 interval，标准为 YYYY-MM 或 DD HH:MM:SS.frac。其映射如下：

- 如果编码长度具有 day 或更小的起始码，则它映射为 com.gbasedbt.jdbc.IfIntervalDF。
- 如果编码长度具有 year 或更小的起始码，则它映射为 com.gbasedbt.jdbc.IfIntervalYM。

10.4 转换内部 GBase 8s 数据类型

要使 Java™ 应用程序可以与 GBase 8s 数据类型的内部服务器表示法一起使用，请使用 IfxToJavaType 类。例如，如果应用程序正在使用GBase 8s Change Data Capture API，则可以使用 IfxToJavaType 类解释字节流。

10.4.1 IfxToJavaType 类

IfxToJavaType 类处理所有 GBase 8s 到 Java™ 数据类型的转换。为转换每个 GBase 8s 数据类型都提供了一个方法。

Java 的基本数据类型为 boolean 、 char 、 byte 、 short 、 int 、 long 、 float 、 double。如果可能，转换返回基本数据类型而不是 Object。

下表显示了可以在 GBase 8s 数据类型与 Java 数据类型之间进行转换的数据类型。

表 1. GBase 8s 和 Java 数据类型之间的转换

GBase 8s 数据类型	Java 数据类型
BIGINT	long
BYTE	int （作为大对象 ID ， 没有输入流）
CHAR (n) / CHARACTER (n)	string
DATE	java.sql.Date
DATETIME	java.sql.Timestamp
DATETIME	interval
DATETIME	string
DEC/DECIMAL (p, s)	java.lang.BigDecimal
DOUBLE PRECISION (n)	double
FLOAT	与 DOUBLE PRECISION 相同

GBase 8s 数据类型	Java 数据类型
INT8	long
INT/INTEGER	int
INTERVAL	interval
MONEY (p, s)	与 DECIMAL 相同
NUMERIC (p, s)	与 DECIMAL 相同
REAL	real
SERIAL (n)	int
SMALLFLOAT	与 REAL 相同
SMALLINT	short
TEXT	int （作为大对象 ID，没有输入流之外）
VARCHAR (m, r)	string

除了转换方法外，还提供了以下方法

- convertDateToDays()
- convertDaysToDate()
- rleapyear()
- widenByte()

convertDateToDays() 方法

convertDateToDays() 方法将 java.sql.Date 转换为一个 int 数据类型，该数据类型将 1900 年 1 月 1 日的日期编码为 1 。日期早于 1900 年 1 月 1 日的日期编码为负数。

方法签名

```
public static int convertDateToDays (java.sql.Date dt)
```

输入参数

dt

java.sql date。

convertDaysToDate() 方法

convertDaysToDate() 方法将日期转换为年、月或日。convertDaysToDate() 方法处理负天数，从 1899 年 12 月 31 日开始反向解释为 0。convertDaysToDate() 方法将 1900 年 1 月 1 日解释为 1。在 1 月 1 日之前是没有日期的，0000 是允许的。该方法依赖于 GBase 8s 生成的有效日期。

方法签名

```
public static java.sql.Date convertDaysToDate (int dt)
```

输入参数

dt

1900 年 1 月 1 日之后的天数（1900 年 1 月 1 日为 1）。

lfxToJavaChar() 方法

lfxToJavaChar() 方法将 GBase 8s CHAR (n) 和 CHARACTER (n) 数据类型转换为 Java[™] string 数据类型。转换是通过从给定字节创建一个字符串来实现的。

方法签名

```
public String lfxToJavaChar (byte b [], short prec,boolean encoption)
public String lfxToJavaChar (byte b [], boolean encoption)
public String lfxToJavaChar (byte b [], int offset, int length,short prec, boolean
encoption)
public String lfxToJavaChar (byte b [], int offset, int length,boolean encoption)
public String lfxToJavaChar (byte b [], short prec, String dbEnc,boolean encoption)
public String lfxToJavaChar (byte b [], String dbEnc, boolean encoption) throws
IOException
public String lfxToJavaChar (byte b [], int offset, int length,short prec,String dbEnc,
boolean encoption)
public String lfxToJavaChar (byte b [], int offset, int length,String dbEnc, boolean
encoption)
```

输入参数

b

字节编码数据。

dbEnc

JDK 编码。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaDate() 方法

IfxToJavaDate() 方法将 GBase 8s DATE 数据类型转换为 Java™ java.sql.Date 数据类型。

方法签名

```
public static java.sql.Date IfxToJavaDate (byte b [], short prec)
public static java.sql.Date IfxToJavaDate (byte b [])
public static java.sql.Date IfxToJavaDate (byte b [], int offset,int length, short prec)
public static java.sql.Date IfxToJavaDate (byte b [], int offset)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaDateTime() 方法

IfxToJavaDateTime() 方法将 GBase 8s DATETIME 数据类型转换为 Java™ java.sql.Timestamp 数据类型。转换路径为 GBase 8s 到 decimal 到 timestamp。

方法签名

```
public static java.sql.Timestamp IfxToJavaDateTime (byte b [], short prec)
```

```
public static java.sql.Timestamp lfxToJavaDateTime (byte b [], int offset,int length,
short prec)
public static java.sql.Timestamp lfxToJavaDateTime (byte b [], int offset,int length,
short prec, Calendar
cal)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

lfxToDateTimeUnloadString() 方法

lfxToDateTimeUnloadString() 方法将 GBase 8s DATETIME 数据类型转换为 Java[™] string 数据类型, 它的格式与 SQL LOAD/UNLOAD 格式相兼容。转换路径为 GBase 8s 到十进制到字符串。

方法签名

```
public static String lfxToDateTimeUnloadString (byte b [], int offset,
int length, short prec)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaDecimal() 方法

IfxToJavaDecimal() 方法将 GBase 8s DECIMAL 数据类型转换为 Java[™] java.lang.BigDecimal 数据类型。

方法签名

```
public static java.math.BigDecimal IfxToJavaDecimal (byte b [], short prec)
public static java.math.BigDecimal IfxToJavaDecimal (byte b [], int offset,
int length, short prec)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaDouble() 方法

IfxToJavaDouble() 方法将 GBase 8s DOUBLE PRECISION 数据类型转换为 Java[™] double 数据类型。

方法签名

```
public static double IfxToJavaDouble (byte b [], short prec)
public static double IfxToJavaDouble (byte b [])
public static double IfxToJavaDouble (byte b [], int offset, int length,short prec)
public static double IfxToJavaDouble (byte b [], int offset)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavalnt() 方法

IfxToJavalnt() 方法将 GBase 8s INTEGER 数据类型转换为 Java™ int 数据类型。

方法签名

```
public static int IfxToJavalnt (byte b [], short prec)
public static int IfxToJavalnt (byte b [])
public static int IfxToJavalnt (byte b [], int offset, int length,short prec)
public static int IfxToJavalnt (byte b [], int offset)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaInterval() 方法

IfxToJavaInterval() 方法将 GBase 8s DATETIME 数据类型转换为 Java™ interval 数据类型。
转换路径为 GBase 8s 到 decimal 到 interval。

方法签名

```
public static Interval IfxToJavaInterval (byte b [], short prec)
public static Interval IfxToJavaInterval (byte b [], int offset, int length,short prec)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaLongBigInt() 方法

IfxToJavaLongBigInt() 方法将 GBase 8s BIGINT 数据类型转换为 Java™ long 数据类型。

方法签名

```
public static long IfxToJavaLongBigInt(byte b [], short prec)
public static long IfxToJavaLongBigInt(byte b [])
public static long IfxToJavaLongBigInt(byte buf [], int offset,int length, short prec)
public static long IfxToJavaLongBigInt(byte b[], int offset)
```

输入参数

b 和 buff

字节编码数据。

offset

字节数组中的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

IfxToJavaLongInt() 方法

IfxToJavaLongInt() 方法将 GBase 8s INT8 数据类型转换为 Java™ long 数据类型。

方法签名

```
public static long IfxToJavaLongInt(byte b [], short prec)
```

```
public static long lfxToJavaLongInt(byte b [])  
public static long lfxToJavaLongInt(byte buf [], int offset, int length,short prec)  
public static long lfxToJavaLongInt(byte buf [], int offset)
```

输入参数

b 和 buf

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

lfxToJavaReal() 方法

lfxToJavaReal() 方法将 GBase 8s REAL 数据类型转换为 Java™ real 数据类型。

方法签名

```
public static float lfxToJavaReal (byte b [], short prec)  
public static float lfxToJavaReal (byte b [])  
public static float lfxToJavaReal (byte b [], int offset,int length, short prec)  
public static float lfxToJavaReal (byte b [], int offset)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

lfxToJavaSmallInt() 方法

lfxToJavaSmallInt() 方法将 GBase 8s SMALLINT 数据类型转换为 Java™ short 数据类型。

方法签名

```
public static short lfxToJavaSmallInt (byte b [], short prec)
public static short lfxToJavaSmallInt (byte b [])
public static short lfxToJavaSmallInt (byte b [], int offset,int length, short prec)
public static short lfxToJavaSmallInt (byte b [], int offset)
```

输入参数

b

字节编码数据。

offset

字节数组的偏移量。

prec

从 GBase 8s 接收的精度。

length

数据长度。

rleapyear() 方法

rleapyear() 方法确定年份是否为闰年。

方法签名

```
public static final boolean rleapyear(int yr)
```

widenByte() 方法

widenByte() 方法将 BYTE 移动到 **short** 数据类型中，使得高位不传播。

方法签名

```
protected static final short widenByte(byte b)
```

10.5 错误消息

10. 5. 1 -79700

Method not supported

GBase 8s JDBC Driver 不支持此 JDBC 方法。

10. 5. 2 -79702

Cannot create new object

软件无法为新的 String 对象分配内存。

10. 5. 3 -79703

Row/column index out of range

行或列索引超出范围。

比较索引和超出查询的行和列数，确保它在范围内。

10. 5. 4 -79704

Cant load driver

GBase 8s JDBC Driver 无法创建实例本身，并无法在 DriverManager 类中注册。
SQLException 文本的剩余部分描述了哪里失败。

10. 5. 5 -79705

Incorrect URL format

您提交的数据库 URL 是无效的。GBase 8s JDBC Driver 无法识别此语法。

检查该语法并重新尝试。

10. 5. 6 -79706

Incomplete input

在将 String 值转换到 IntervalDF 或 IntervalYM 对象的过程中发现一个无效字符。

请查看 INTERVAL 数据类型 并获取正确的值。

10. 5. 7 -79707

Invalid qualifier

从原子元素构造 Interval 限定符时发出长度、起始值或结束值错误。

检查长度、起始值和结束来验证它们是否正确。请参阅 INTERVAL 数据类型 以获取正确的值。

10.5.8 -79708

Cannot take null input

您提供的字符串为空。在这种情况下，GBase 8s JDBC Driver 无法理解空输入。

请检查输入字符串，确保它具有适当的值。

10.5.9 -79709

Error in date format

预期输入的是以下格式的有效日期字符串：yyyy-mm-dd。

检查日期并确认它具有四位数的年份，后跟有效的两位数月份和两位数日期。分隔符必须是连字符 (-)。

10.5.10 -79710

Syntax error in SQL escape clause

无效的语法被传送到一个 jdbc 转义子句。有效的 JDBC 转义子句语法通过大括号和关键字进行划分：{keyword syntax}。

检查 JDBC 规范以获取有效的转义子句关键字和语法。

10.5.11 -79711

Error in time format

无效的时间格式被传送到 JDBC 转义子句。时间字符的转义子句语法具有以下格式：{t 'hh:mm:ss'}。

10.5.12 -79712

Error in timestamp format

无效的时间戳格式被传送到 JDBC 转义子句。该时间戳字符的转义子句具有以下格式：{ts 'yyyy-mm-dd hh:mm:ss.f...'}。

10. 5. 13 -79713

Incorrect number of arguments

标量函数转义语法传递了不正确的参数。正确的语法是 {fn function(arguments)}。

验证正确数的参数传递给了此函数。

10. 5. 14 -79714

Type not supported

您指定了一个 GBase 8s JDBC Driver 不支持的特定数据类型。

检查您的查询确保使用的数据类型是受支持的。

10. 5. 15 -79715

Syntax error

无效的语法被传送到 jdbc 转义子句。有效的 JDBC 转义子句语法通过大括号和关键字进行划分：{keyword syntax}。

检查 JDBC 规范以获取有效的转义子句关键字和语法。

10. 5. 16 -79716

System or internal error

发生操作或运行系统错误或驱动程序内部错误。随后的消息描述了此问题。

10. 5. 17 -79717

Invalid qualifier length

Interval 对象的长度值不正确。

请参阅 INTERVAL 数据类型 以获取正确的值。

10. 5. 18 -79718

Invalid qualifier start code

Interval 对象的起始值错误。

请参阅 INTERVAL 数据类型 以获取正确的值。

10. 5. 19 -79719

Invalid qualifier end code

Interval 对象的结束值不正确。

请参阅 INTERVAL 数据类型 以获取正确的值。

10. 5. 20 -79720

Invalid qualifier start or end code

Interval 对象的起始值或结束值不正确。

请参阅 INTERVAL 数据类型 以获取正确的值。

10. 5. 21 -79721

Invalid interval string

在将 String 值转换为 IntervalDF 或 IntervalYM 对象过程中发生错误。请参阅 INTERVAL 数据类型 以获取正确的格式。

10. 5. 22 -79722

Numeric character(s) expected

在将 String 值转换为 IntervalDF 或 IntervalYM 对象过程中发生错误。请参阅 INTERVAL 数据类型 以获取正确的格式。

10. 5. 23 -79723

Delimiter character(s) expected

在将 String 值转换为 IntervalDF 或 IntervalYM 对象过程中发生错误。请参阅 INTERVAL 数据类型 以获取正确的格式。

10. 5. 24 -79724

Character(s) expected

在将 String 值转换为 IntervalDF 或 IntervalYM 对象过程中发生错误。完成转换前遇到 End of string。

请参阅 INTERVAL 数据类型 以获取正确的格式。

10. 5. 25 -79725

Extra character(s) found

在将 String 值转换为 IntervalDF 或 IntervalYM 对象过程中发生错误。预期错误为 End of string，但是此字符串中有多余的字符。

请参阅 INTERVAL 数据类型 以获取正确的格式。

10. 5. 26 -79726

Null SQL statement

传入的 SQL 语句为空。

请检查您的应用程序的 SQL 语句字符串，以确保它包含有效的语句。

10. 5. 27 -79727

Statement was not prepared

SQL 语句还未准备好。如果您在 SQL 语句中使用主机变量（例如，insert into mytab values (?, ?);，则在执行该语句前，必须使用connection.prepareStatement() 准备此 SQL 语句。

10. 5. 28 -79728

Unknown object type

如果此对象类型是空的不透明类型，则此类型无法被识别和处理。如果对象类型是复杂类型，则集合中或数组中的数据是未知的类型，无法映射到GBase 8s 类型。如果此对象类型为行，则此行中的元素不能映射到 GBase 8s 类型。请验证该自定义类型映射或对象的数据类型。

10. 5. 29 -79729

Method cannot take argument

此方法无法使用此参数。请参阅 Java™ API 规范或该指南适当的章节来确保您正适当地使用方法。

10. 5. 30 -79730

Connection not established

连接未建立。

必须首先通过调用 `DriverManager.getConnection()` 或 `DataSource.getConnection()` 方法获取此连接。

10. 5. 31 -79731

MaxRows out of range

您具有一个超出范围的 `maxRow` 值。请确保您指定的值在 0 和 `Integer.MAX_VALUE` 之间。

10. 5. 32 -79732

Illegal cursor name

指定游标名称无效。请确保传入的字符串不是 `NULL` 或空。

10. 5. 33 -79733

No active result

该语句不包含有效的结果。请检查您的程序逻辑，确保您在尝试引用结果之前调用了 `executeXXX()` 方法。

10. 5. 34 -79734

GBASEDBTSERVER has to be specified

`GBASEDBTSERVER` 是连接到 GBase 8s 所需的属性。您可以在数据库 URL 中指定它或将它作为 Properties 对象的一部分传递给 `connect()` 方法。

10. 5. 35 -79735

Cant instantiate protocol

在尝试连接期间发出了内部错误。请联系技术支持。

10. 5. 36 -79736

No connection/statement establish yet

当前没有连接或语句不存在。

检查您的程序确保连接已适当建立或语句已创建。

10. 5. 37 -79737

No metadata

此 SQL 语句没有数据源可用。

请确保在尝试使用它之前该语句已生成了一个结果集。

10. 5. 38 -79738

No such column name

指定的列名不存在。请确保该列名是正确的。

10. 5. 39 -79739

No current row

游标位置不正确。您必须首先使用 `ResultSet.next()`、`ResultSet.beforeFirst()`、`ResultSet.first()` 或 `ResultSet.absolute()` 方法将游标放到结果集中。

10. 5. 40 -79740

No statement created

此语句不存在。请确保该语句已经创建。

10. 5. 41 -79741

Cannot convert to

无法将列的数据类型转换为指定的数据类型。实际的数据类型在该消息的末尾描述。

请检查您的程序逻辑确保您请求的转换是受支持的。请参阅映射数据类型获取数据映射数组。

10. 5. 42 -79742

Cannot convert from

无法将列的数据类型转换为指定的数据类型。实际的数据类型在该消息的末尾描述。

请检查您的程序逻辑确保您请求的转换是受支持的。请参阅映射数据类型获取数据映射数组。

10. 5. 43 -79744

Transactions not supported

用户尝试在不支持事务的数据库上调用 `commit()` 或 `rollback()`，或者已经在非日志记录的数据库上尝试将 `autoCommit` 设置为 `False`。

请验证当前数据库具有正确的日志记录模式并检查此程序逻辑。

10. 5. 44 -79745

Read only mode not supported

GBase 8s 不支持只读模式。

10. 5. 45 -79746

No Transaction Isolation on non-logging db's

GBase 8s 不支持在非日志记录的数据库上设置事务隔离级别。

10. 5. 46 -79747

Invalid transaction isolation level

如果数据库服务器无法完成回滚，则发送此错误。请参阅 `SQLException` 消息的剩余部分，来了解回滚失败的详细原因。

如果无效的事务隔离级别传送到 `setTransactionIsolation()`，则该错误还会发生。有效值为：

`TRANSACTION_NONE`

`TRANSACTION_READ_UNCOMMITTED`

`TRANSACTION_READ_COMMITTED`

`TRANSACTION_REPEATABLE_READ`

`TRANSACTION_SERIALIZABLE`

`TRANSACTION_LAST_COMMITTED`

10. 5. 47 -79748

Cannot lock the connection

GBase 8s JDBC Driver 在与数据库服务器进行数据交换之前通常锁定此连接对象。驱动程序无法获取该锁。一次只能有一个线程使用此连接对象。

10. 5. 48 -79749

Number of input values does not match number of question marks

此语句中您使用 `PreparedStatement.setXXX()` 方法设置的变量数不符合您写入此语句的占位符 `?` 数。

找到语句的文本并验证占位符的数量，然后检查对 `PreparedStatement.setXXX()` 的调用。

10. 5. 49 -79750

Method only for queries

如果语句是 `SELECT` 语句，则只能使用 `Statement.executeQuery(String)` 和 `PreparedStatement.executeQuery()` 方法。对于其它语句，请使用 `Statement.execute(String)`、`Statement.executeBatch()`、`Statement.executeUpdate(String)`、`Statement.getUpdateCount()`、`Statement.getResultSet()` 或 `PreparedStatement.executeUpdate()` 方法。

10. 5. 50 -79755

Object is null

传递的对象为空。请检查您的程序逻辑，确保引用的对象是有效的。

10. 5. 51 -79756

Must start with 'jdbc'

数据库 URL 的第一个令牌必须是关键字 `jdbc`（不区分大小写），如以下示例所示：

```
jdbc:gbasebt-sqli://mymachine:1234/  
  mydatabase:user=me:  
  password=secret
```

10. 5. 52 -79757

Invalid subprotocol

当前有效的子协议是 gbasedbt-sqli。

10. 5. 53 –79758

Invalid IP address

当通过 IP 地址连接到 GBase 8s 数据库服务器时，该 IP 地址必须是有效的。有效的 IP 地址是一个四个 0 - 255 的集合，由点 (.) 分隔：例如，127.0.0.1。

10. 5. 54 –79759

Invalid port number

端口号必须是有效的四位数字，如下所示：

```
jdbc:gbasedbt-sqli://mymachine:1234/  
mydatabase:user=me:  
password=secret
```

在此示例中，1234 是端口号。

10. 5. 55 –79760

Invalid database name

该语句包含的数据库名称的格式无效。

数据库名和游标名称的最大长度取决于数据库服务器的版本。

数据库名和游标名必须以字母、数字和下划线字符开头。数据库名和游标名可以使用一个下划线开头。

在 MS-DOS 系统中，名称最大可以是八个字符加三个字符扩展。

10. 5. 56 –79761

Invalid property format

数据库 URL key=value 对中的值正确。例如，user=gbasedbt:password=gbasedbt 将 key=value 对添加到传送连接对象的属性列表。

检查 key=value 对的语法是否正确。确保只有一个 = 等号；没有空格分隔这些 key 、value 或 =；并且 key=value 对使用一个冒号 (:) 分隔，没有空格。

10. 5. 57 –79762

Attempt to connect to a non 5.x server

当连接到版本为 5.x 的数据库服务器时，用户必须将数据库 URL 设置 USE5SERVER 为非空值。如果连接到版本 6.0 或更高版本的数据库服务器时，抛出此异常。

请验证数据库服务的版本正确，并修改所需的数据库 URL。

10. 5. 58 -79764

Invalid fetch direction value

一个无效的访存方向被作为参数传送

到 `Statement.setFetchDirection()` 或 `ResultSet.setFetchDirection()` 方法。有效值为 `FETCH_FORWARD`、`FETCH_REVERSE` 和 `FETCH_UNKNOWN`。

10. 5. 59 -79765

`ResultSet` type is `TYPE_FETCH_FORWARD`, direction can only be `FETCH_FORWARD`

如果结果集类型设置为 `TYPE_FORWARD_ONLY`，但是 `setFetchDirection()` 方法已经被其它值而不是 `FETCH_FORWARD` 调用。指定的方向必须与指定的结果集类型一致。

10. 5. 60 -79766

Incorrect fetch size value

使用无效值调用 `Statement.setFetchSize()` 方法。验证传入的值是大于 0 的。如果 `setMaxRows()` 方法已经被调用，则访存大小不能超出此值。

10. 5. 61 -79767

`ResultSet` type is `TYPE_FORWARD_ONLY`

调用了诸如 `ResultSet.beforeFirst()`、`ResultSet.afterLast()`、`ResultSet.first()`、`ResultSet.last()`、`ResultSet.absolute()`、`ResultSet.relative()`、`ResultSet.current()` 或 `ResultSet.previous()` 这样的方法，但是结果集类型为 `TYPE_FORWARD_ONLY`。如果结果集类型是 `TYPE_FORWARD_ONLY`，则只能调用 `ResultSet.next()` 方法。

10. 5. 62 -79768

Incorrect row value

`ResultSet.absolute(int)` 方法被值 0 调用。参数必须大于 0。

10. 5. 63 -79769

A customized type map is required for this data type

您必须注册一个自定义类型映射来使用任何不透明类型。

10. 5. 64 -79770

Cannot find the SQLTypeName specified in the SQLData or Struct

SQLData 或 Struct 类中指定 SQLTypeName 对象在数据库中不存在。请确保类型名称有效。

10. 5. 65 -79771

Input value is not valid

此数据类型不接受此输入值。请确保此输入值是对该数据类型是一个有效输入。

10. 5. 66 -79772

No more data to read or write. Verify your SQLData class or getSQLTypeName()

当 Java™ 用户定义的例程试图读或设置超出数据输入流中可用的不透明类型数据末尾的位置时，会发生此错误。

请仔细检查数据输入 UDR 代码的不透明类型的长度和结构。getSQLTypeName() 方法返回的 SQLTypeName 对象也可能不正确。

10. 5. 67 -79774

Unable to create local file

从数据库服务器读取的大对象可以存储在内存或本地文件中。如果 LOBCACHE 值为 0 或者大对象大小大于 LOBCACHE 值，则来自数据库服务器的大对象数据通常存储在文件中。在这种情况下，如果发生安全异常，则 GBase 8s JDBC Driver 不会尝试将大对象存储到内存中，并抛出此异常。

10. 5. 68 -79775

Only TYPE_SCROLL_INSENSITIVE and TYPE_FORWARD_ONLY are supported

GBase 8s JDBC Driver 仅支持 TYPE_SCROLL_INSENSITIVE 和 TYPE_FORWARD_ONLY 的结果集类型。只能使用这些值。

10. 5. 69 –79776

Type requested (%s) does not match row type information (%s) type

从系统目录或行定义获取行类型信息。提供的行数据与该行的元素类型不匹配。必须修改类型信息或必须提供数据。

10. 5. 70 –79777

readObject/writeObject() only supports UDTs, Distincts, and complex types

为不是用户定义、DISTINCT 或复制类型的对象调用 SQLData.writeObject() 方法。

请验证您提供的自定义类型映射信息。

10. 5. 71 –79778

Type mapping class must be a java.util.Collection implementation

提供了一个类型映射以重写缺省的 SET 、 LIST 或 MULTiset 数据类型，但是类不能实现 java.util.Collection 接口。

10. 5. 72 –79780

Data within a collection must all be the same Java™ class and length

请验证集合中的所有对象都是相同的类。

10. 5. 73 –79781

Index/Count out of range

使用索引和计数值调用 Array.getArray() 或 Array.getResultSet() 。索引超出范围或计数值太大。

请验证数组中的元素数满足索引和计数值。

10. 5. 74 –79782

Method can be called only once

请确保每个结果只调用一次诸如 Statement.getUpdateCount() 和 Statement.getResultSet() 之类的方法。

10. 5. 75 -79783

Encoding or code set not supported

DB_LOCALE 或 CLIENT_LOCALE 变量中输入的编码或代码集无效。

请参阅支持代码集转换，获取有效的代码集。

10. 5. 76 -79784

Locale not supported

DB_LOCALE 或 CLIENT_LOCALE 变量中输入的语言环境无效。

请参阅支持代码集转换，获取有效的语言环境。

10. 5. 77 -79785

Unable to convert JDBC escape format date string to localized date string

日期值的 JDBC 转义格式必须指定为 {d 'yyyy-mm-dd'} 格式。请验证指定的 JDBC 转义格式是否正确。

如果这些环境变量中的任何一个设置为连接数据库 URL 字符串或属性列表中的值，则请验证 DBDATE 和 GL_DATE 设置是否具有正确的日期字符串格式。

10. 5. 78 -79786

Unable to build a Date object based on localized date string representation

在 CHAR 、 VARCHAR 或 LVARCHAR 列中指定的日期字符串表示形式不正确，日期对象不能基于年、月和日的值建立。

如果这些环境变量中的任何一个设置为连接数据库 URL 字符串或属性列表中的值，则请验证日期字符串表示形式符合 DBDATE 或 GL_DATE 日期格式。如果指定了 DBDATE 或 GL_DATE ，但是在连接数据库 URL 字符串或属性列表中显式设置了 CLIENT_LOCALE 或 DB_LOCALE, 则请验证日期字符串表示形式与 JDK short 缺省格式 (DateFormat.SHORT) 相匹配。

10. 5. 79 -79788

User name must be specified

需要用户名建立与 GBase 8s JDBC Driver 的连接。

请确保您在数据库 URL 或属性中包含了 user=your_user_name。

10. 5. 80 -79789

Server does not support GLS variables DB_LOCALE, CLIENT_LOCALE or GL_DATE

这些变量只能在支持 GLS 的数据库服务器中使用。

如果不支持，检查您的数据库服务器版本文档，并忽略这些变量。

10. 5. 81 -79790

Invalid complex type definition string

getSQLTypeName() 方法返回的值为空或无效。

请检查此字符串确保它是一个有效的命名行名称或是一个有效的行类型定义。

10. 5. 82 -79792

Row must contain data

Array.getAttributes() 或 Array.getAttributes(Map) 方法返回 0 元素，这些方法必须返回一个非零数。

10. 5. 83 -79793

Data in array does not match getBaseType() value

Array.getArray() 或 Array.getArray(Map) 方法返回的数组中的元素类型与 JDBC 基本类型不匹配。

10. 5. 84 -79794

Row length provided (%s) does not match row type information (%s)

行中的数据与行类型信息中的长度不匹配。不需要截断此字符串长度来符合行定义的长度，但是其它数据类型的长度必须符合。

10. 5. 85 -79795

Row extended ID provided (%s) does not match row type information (%s)

行中对象扩展 ID 与行类型信息中定义的扩展 ID 不匹配。

可以更改行类型信息（如果您提供了行定义）或检查类型映射信息。

10. 5. 86 –79796

Cannot find UDT, distinct, or named row (%s) in database

getSQLTypeName() 方法返回的名称在数据库中找不到。

请验证 Struct 或 SQLData 对象返回了正确的信息。

10. 5. 87 –79797

DBDATE setting must be at least four characters and no longer than six characters

由于传递到数据库服务器的 DBDATE 格式具有太多或太少的字符，会发生此错误。

要修复此文件，请使用用户手册验证 DBDATE 格式字符串，确保年、月、日以及 DBDATE 字符串的每个部分都正确标识。

10. 5. 88 –79798

A numeric year expansion is required after 'Y' character in DBDATE string

发生此错误是因为 DBDATE 格式字符有一个年指示符（由字符 Y 指定），但是年指示符之后没有表示数字年份扩展值的字符（2 或 4）。

要修复此问题，请修改 DBDATE 格式字符串，使其在 Y 字符之后包含所需的数字年份扩展。Y 字符只能跟随字符 2 或 4。

10. 5. 89 –79799

An invalid character is found in the DBDATE string after the 'Y' character

发生此错误是因为 DBDATE 格式字符有一个年指示符（由字符 Y 指定），但是年指示符之后的字符不是 2（两位数的年份）或 4（四位数的年份）。

要修复此问题，请修改 DBDATE 格式字符串，使其在 Y 字符之后包含所需的数字年份扩展。Y 字符只能跟随字符 2 或 4。

10. 5. 90 –79800

No 'Y' character is specified before the numeric year expansion value

发生此错误是因为 DBDATE 格式字符串具有一个数值年扩展（2 表示两位数年份或 4 表示四位数年份），但是指定数字年份扩展字符前没有找到年份指定符（Y）。

要修复此问题。请修改 DBDATE 格式字符串，使其在请求的数字年份扩展值之前包含所需的 Y 字符。

10. 5. 91 -79801

An invalid character is found in DBDATE format string

发生此错误的原因是 DBDATE 格式字符串具有不被允许的字符。

要修复此问题，请修改 DBDATE 格式字符串，使其仅包含正确的日期部分：年（Y），数值年份扩展值（2 或 4），月（M）和日（D）。或者，可以在 DBDATE 格式字符串的末尾指定纪元名称（E）和一个缺省的分隔符（连字符、点或反斜杠）。有关正确的 DBDATE 格式字符串的表示，请参阅用户文档。

10. 5. 92 -79802

Not enough tokens are specified in the string representation of a date value

发生此错误是因为指定的日期字符串的标记或分隔符数不够，不能成为一个有效的日期值。例如，12/15/98 是有效的日期字符串表示形式，它具有反斜杠作为分隔符或标记。而 12/1598 则是无效的日期字符串表示形式，因为没有足够的分隔符或标记。

要修复此问题，请修改日期字符串表示形式，使其包含有效的格式。

10. 5. 93 -79803

Date string index out of bounds during date format parsing to build Date object

发生此错误是因为 DBDATE 或 GL_DATE 要求的日期字符串格式与您定义的实际日期字符串表示形式之间不存在一对一的对应关系。例如，如果将 GL_DATE 设置为 %b %D %y，而指定了字符串 Oct，则 GL_DATE 所需的格式与实际日期字符串之间存在明显的不匹配。

要修复此问题，请修改 DBDATE 或 GL_DATE 设置的日期字符串表示形式，以便指定的日期格式与所需的日期字符串表示形式一一对应。

10. 5. 94 -79804

No more tokens are found in DBDATE string representation of a date value

发生此错误的原因是指定的日期字符串没有更多所需的标记或分隔符来基于 DBDATE 格式字符串形成一个有效的日期值（由年、月和日组成）。例如，当 DBDATE 设置为 MDY2/ 时 12/15/98 是有效的日期字符串表示形式。但是 12/1598 是一个无效的日期字符串表示形式，因为没有足够多的分隔符或标记。

要修复此问题，请修改日期字符串表示形式，使其包含有效的格式，基于 DBDATE 格式字符串设置来分隔日期值的日、月和年部分。

10. 5. 95 –79805

No era designation found in DBDATE/GL_DATE string representation of date value

发生此错误的原因是指定的日期字符串的纪元名称无效，如不符合 DBDATE 或 GL_DATE 格式字符串设置的要求。例如，如果将 DBDATE 设置为 Y2MDE-，但是用户指定的日期字符串表示法为 98-12-15，这是错误的，因为在此日期字符串值的末尾没有指定纪元。

要修复此问题，请修改此字符串表示法以包含基于 DBDATE 或 GL_DATE 格式字符串设置的有效纪元名称。在这个例子中，根据语言环境，98-12-15 AD 的日期字符串表示形式才是正确的。

10. 5. 96 –79806

Numerical day value can not be determined from date string based on DBDATE

发生此错误的原因是指定的日期字符串不具有 DBDATE 格式字符串设置所需的有效的天数。例如，如果 DBDATE 设置为 Y2MD-，但是指定的日期字符串表示形式是 98-12-blah，这是错误的，因为 blah 不是一个有效的天数表示形式。

要修复此问题，请修改日期字符串的表示形式，使其包含基于 DBDATE 格式字符串设置的有效数字天数（从 1 到 31）。

10. 5. 97 –79807

Numerical month value can not be determined from date string based on DBDATE

发生此错误是由于指定的日期字符串不具有 DBDATE 格式字符串所需的有效的数值月份指示符。例如，如果 DBDATE 设置为 Y2MD-，但是您指定的日期字符串表示形式为 98-blah-15，这是错误的，因为 blah 是一个无效的数值月份表示形式。

要修复此问题，请修改此日期字符串，使其包含基于 DBDATE 格式字符串设置的有效数值月份（从 1 到 12）。

10. 5. 98 –79808

Not enough tokens specified in %D directive representation of date string

发生此错误的原因是指定的字符串没有正确数量的标记或分隔符来构成基于 GL_DATE %D 指令（mm/dd/yy 格式）的有效日期值。例如，12/15/98 是基

于 GL_DATE%D 指令的有效字符串表示形式，但是 12/1598 是一个无效的字符串表示形式，因为没有足够的分隔符或标记。

要修复此问题，请修改日期字符串表示形式，使其包含 GL_DATE%D 指令的有效格式。

10. 5. 99 -79809

Not enough tokens specified in %x directive representation of date string

发生错误的原因是因为指定的日期字符串没有根据 GL_DATE %x 指令形成有效的日期值的正确数量的标记或分隔符，（所需的格式是基于日、月、年的部分，以及这些部分的顺序是由指定的语言环境决定的）例如，12/15/98 是基于美国英语语言环境设置的 GL_DATE %x 指令的有效日期字符串表示形式，但 12/1598 不是有效的日期字符串表示形式，因为没有足够的分隔符或标记。

要修复此问题，请修改此字符串表示形式，使其包含基于语言环境的 GL_DATE %x 指令的有效格式。

10. 5. 100 -79811

Connection without user/password not supported

调用 getConnection() 方法的 DataSource 对象时，用户名称或密码为空。

使用 getConnection() 方法的用户名称和密码参数或在 DataSource 对象中设置这些值。

10. 5. 101 -79812

User/Password does not match with datasource

调用 getConnection(user, passwd) 方法的 DataSource 对象时，您提供的值与数据源中的值不匹配。

10. 5. 102 -79814

Blob/Clob object is either closed or invalid

如果你使用 ResultSet.getBlob() 或 ResultSet.getClob() 方法检索智能大对象或使用 IfxBlob() 或 IfxCblob() 构造函数创建智能大对象，则该智能大对象是打开的。可以读/写智能大对象。在执行 IfxBlob.close() 方法之前，不要使用智能大对象进行进一步的读/写操作，否则会抛出此异常。

10. 5. 103 -79815

Not in Insert mode. Need to call moveToInsertRow() first

您尝试使用 insertRow() 方法，但是此模式未设置为 Insert。

调用 insertRow() 之前，请调用 moveToInsertRow() 方法。

10. 5. 104 -79816

Cannot determine the table name

查询中的表名不正确或引用的表不存在。

10. 5. 105 -79817

No serial, rowid, or primary key specified in the statement

可更新可滚动功能仅适用于在查询中指定了具有 SERIAL 列、主键或行 ID 的表。如果表没有这些属性，则会创建一个可更新的可滚动的游标。

10. 5. 106 -79818

Statement concurrency type is not set to CONCUR_UPDATABLE

试图为尚未使用 CONCUR_UPDATABLE 并发类型创建的语句调用 insertRow() 、updateRow() 或 deleteRow() 方法。

用这个并发属性设置的类型重新创建语句。

10. 5. 107 -79819

Still in Insert mode. Call moveToCurrentRow() first

您不能在 Insert 模式中调用 updateRow() 或 deleteRow() 方法。请先调用 moveToCurrentRow() 方法。

10. 5. 108 -79820

Function contains an output parameter

您传送了一个包含 OUT 参数的语句，但是没有使用驱动程序 CallableStatement.registerOutParameter() 和 getXXX() 方法处理该 OUT 参数。

10. 5. 109 -79821

Name unnecessary for this data type

如果具有一个需要名称的数据类型（不透明类型或复杂类型），您必须调用具有该名称参数的方法，如下所示：

```
public void IfxSetNull(int i, int ifxType,
    String name)
    public void registerOutParameter
    (int parameterIndex,
    int sqlType, java.lang.String name);
    public void IfxRegisterOutParameter
    (int parameterIndex,
    int ifxType, java.lang.String name);
```

您指定的数据类型不需要名称。

使用没有类型参数的另一个方法。

10. 5. 110 -79822

OUT parameter has not been registered

使用 CallableStatement 接口指定的函数有一个尚未注册的 OUT 参数。

在调用 executeQuery() 方法之前，使用 registerOutParameter() 或 IfxRegisterOutParameter() 方法注册 OUT 参数类型。

10. 5. 111 -79823

IN parameter has not been set

使用 CallableStatement 接口定义指定的函数具有一个未设置的 IN 参数。

如果您想要设置一个 NULL IN 参数，则调用 setMaxRows() 或 IfxSetNull() 方法。否则，调用从 PreparedStatement 接口继承的任一集合方法。

10. 5. 112 -79824

OUT parameter has not been set

使用 CallableStatement 接口指定的函数具有还未设置的 OUT 参数。

如果您想要设置一个 NULL OUT 参数，则调用 setMaxRows() 或 IfxSetNull() 方法。否则，调用从 PreparedStatement 接口继承的任一设置方法。

10. 5. 113 -79825

Type name is required for this data type

此数据类型是不透明类型、distinct 类型或复杂类型，它需要一个名称。

使用 IN 参数的设置方法，并为以类型名称作为参数的 OUT 参数注册方法。

10. 5. 114 -79826

Ambiguous java.sql.Type, use IfxRegisterOutParameter()

指定的 SQL 类型没有映射到 GBase 8s 数据类型，或者具有多个映射。

使用 IfxRegisterOutParameter() 方法之一指定 GBase 8s 数据类型。

10. 5. 115 -79827

Function doesn't have an output parameter

该函数没有 OUT 参数，或该函数具有的 OUT 参数值是服务器版本不返回的。

CallableStatement 接口中的方法都不适用。使用PreparedStatement 接口的继承方法。

10. 5. 116 -79828

Function parameter specified isnt an OUT parameter

GBase 8s 函数只能具有一个 OUT 参数，它通常为最后一个参数。

10. 5. 117 -79829

Invalid directive used for the GL_DATE environment variable

不允许使用 GL_DATE 环境变量指定的一个或多个指令。有关 GL_DATE 变量的有效指令列表，请参阅 GL_DATE 格式。

10. 5. 118 -79830

Insufficient information given for building a time or timestamp Java™ object.

要为构建 java.sql.Timestamp 或 java.sql.Time 对象正确执行字符串到二进制的转换，必须为所选日期字符串表示指定所有 DATETIME 字段。对于 java.sql.Timestamp 对象，必须在字符串中指定年、月、日，小时，分钟和秒部分。对于 java.sql.Time 对象，必须在字符串表示中指定小时、分钟和秒部分。

10. 5. 119 -79831

Exceeded maximum no. of connections configured for Connection Pool Manager

如果您在不关闭连接的情况下，使用 `DataSource` 对象重复连接数据库，则连接累积。
当 `DataSource` 对象的连接的总数超出最大限制（100）时，会抛出此异常。

10. 5. 120 -79834

Distributed transactions (XA) are not supported by this database server.

当用户调用服务器不支持的 `XAConnection.getConnection()` 方法时，会发生此错误。

10. 5. 121 -79836

Proxy Error: No database connection

如果您尝试使用一个无效或坏的数据库连接与数据库通信，则 GBase 8s HTTP Proxy 会抛出此错误。

请确保您的应用程序已经打开到数据库的连接，检查您的网络服务器和数据库错误日志。

10. 5. 122 -79837

Proxy Error: Input/output error while communicating with database

如果在代理服务器与数据库通信时检测到错误，则 GBase 8s HTTP Proxy 可能抛出此错误。
如果您的数据库服务器不可访问，也可能发生此错误。

请确保您的数据库服务器是可以访问的，检查您的数据库和网络服务器错误日志。

10. 5. 123 -79838

Cannot execute change permission command (chmod/attrib)

驱动程序无法更改客户端 JAR 文件的权限。如果您的客户端平台不支持 `chmod` 或 `attrib` 命令，或者如果用户运行的 JDBC 应用程序不具有权限更改客户端 JAR 文件的访问权限，则可能发生此错误。

请确保您的平台上的 `chmod` 或 `attrib` 命令是有效的，并且应用程序的用户具有更改客户端 JAR 文件的访问权限的权利。

10. 5. 124 -79839

Same Jar SQL name already exists in the system catalog

当应用程序调用 `UDTManager.createJar()` 时指定 JAR 文件名已经在数据库服务器中注册过。

使用 `UDTMetaData.setJarFileSQLName()` 为 JAR 文件指定一个不同的 SQL 名称。

10. 5. 125 -79840

Unable to copy jar file from client to server

使用 `setJarTmpPath()` 设置的路径名不能被用户 `gbasedbt` 或 JDBC 连接中指定的用户写入。

请确保路径名是可靠的，可以被任何用户写入。

10. 5. 126 -79842

No UDR information was set in UDRMetaData

应用程序调用 `UDRManager.createUDRs()` 方法，但没有为要注册的数据库服务器指定任何 UDR。

在调用 `UDRManager.createUDRs()` 方法之前，调用 `UDRMetaData.setUDR()` 为数据库服务器指定 UDR。

10. 5. 127 -79843

SQL name of the jar file was not set in UDR/UDT MetaData

应用程序调用 `UDTManager.createUDT()` 或 `UDRManager.createUDRs()` 方法，而没有为包含要注册的数据库服务器的不透明类型或 UDR 的 JAR 文件指定 SQL 名称。

在调用 `UDTManager.createUDT()` 或 `UDRManager.createUDRs()` 方法之前，通过调用 `UDTMetaData.setJarFileSQLName()` 或 `UDRMetaData.setJarFileSQLName()` 方法为 JAR 文件指定 SQL 名称。

10. 5. 128 -79844

Cant create/remove UDT/UDR as no database is specified in the connection

应用程序建立了一个连接，但没有指定数据库。以下示例建立了连接并打开了名为 `test` 的数据库：

```
url = "jdbc:gbasedbt-sqli:myhost:1533/test:"  
+  
"gbasedbtserver=myserver;user=rdtest;  
password=test";
```

```
conn = DriverManager.getConnection(url);
```

下列示例建立了一个没有数据库打开的连接：

```
url = "jdbc:gbasedbt-sqli:myhost:1533:"  
      +  
      "gbasedbtserver=myserver;user=rdtest;  
      password=test";  
conn = DriverManager.getConnection(url);
```

要解决此问题，请在建立连接之后调用 `createUDT()` 或 `createUDRs()` 方法之前使用以下 SQL 语句：

```
Statement stmt = conn.createStatement();  
      stmt.executeUpdate("create database test  
      ...");
```

相反，使用以下代码：

```
stmt.executeUpdate("database test");
```

10. 5. 129 -79845

JAR file on the client does not exist or cant be read

由于以下原因发生此错误：

创建客户端 JAR 文件失败。

为客户端 JAR 文件指定了错误的路径名。

运行 JDBC 应用程序的用户或在连接中指定的用户没有权限打开或读取客户端 JAR 文件。

10. 5. 130 -79846

Invalid JAR file name

应用程序指定为 `UDTManager.createUDT()` 或 `UDRManager.createUDRs()` 的第二个参数的客户端 JAR 文件必须以 .jar 扩展名结束。

10. 5. 131 -79847

The 'javac' or 'jar' command failed

驱动程序遇到了以下任一情况就会发生此错误：

使用 `jar` 命令将 .class 文件编译为 .jar 文件，以响应 JDBC 应用程序中的 `createJar()` 命令

使用 javac 和 jar 命令将 .java 文件编译为 .class 文件和 .jar 文件，以响应 JDBC 应用程序中的 UDTManager.createUDTClass() 命令。

10. 5. 132 -79848

Same UDT SQL name already exists in the system catalog

应用程序调用 UDTMetaData.setSQLName() 指定了一个已经在数据库服务器中存在的名称。

10. 5. 133 -79849

UDT SQL name was not set in UDTMetaData

应用程序调用 UDTMetaData.setSQLName() 为不透明类型指定 SQL 名称失败。

10. 5. 134 -79850

UDT field count was not set in UDTMetaData

应用程序调用了 UDTManager.createUDTClass()，但是之前没有指定定义此不透明类型的内部数据结构中的字段数。

调用 UDTMetaData.setFieldCount() 指定字段数。

10. 5. 135 -79851

UDT length was not set in UDTMetaData

应用程序调用了 UDTManager.createUDTClass()，而之前没有为不透明类型指定长度。

调用 UDTMetaData.setLength() 指定此不透明类型的总长度。

10. 5. 136 -79852

UDT field name or field type was not set in UDTMetaData

应用程序调用了 UDTManager.createUDTClass()，但是没有先为数据结构中定义不透明类型的每个字段指定字段名称和数据类型。

通过调用 UDTMetaData.setFieldName() 指定字段名称；调用 UDTMetaData.setFieldType() 指定字段类型。

10. 5. 137 -79853

No class files to be put into the jar

应用程序调用 `createJar()` 方法并为此 `classnames` 参数传递了一个零长度的字符串。该方法签名如下所示：

```
createJar(UDTMetaData mdata, String[]  
          classnames)
```

10. 5. 138 -79854

UDT java class must implement java.sql.SQLData interface

应用程序调用 `UDTManager.createUDT()` 创建不透明类型，其类定义没有实现 `java.sql.SQLData` 接口。`UDTManager` 无法从没有实现此接口的类中创建不透明类型。

10. 5. 139 -79855

Specified UDT java class is not found

应用程序调用了 `UDTManager.createUDT()` 方法但是未找到您为第三方参数指定的名称的类。

10. 5. 140 -79856

Specified UDT does not exists in the database.

应用程序调用了 `UDTManager.removeUDT(String sqlname)` 从数据库移除名为 `sqlname` 的不透明类型，但是具有此名称的不透明类型在数据库中不存在。

10. 5. 141 -79857

Invalid support function type

仅当应用程序调用 `UDTMetaData.setSupportUDR()` 方法并将不是 0 到 7 之间的整数作为 `type` 参数传送时，会发生此错误。

使用定义了支持 UDR 常量。有关更多信息，请参阅 `setSupportUDR()` 和 `setUDR()` 方法。

10. 5. 142 -79858

The command to remove file on the client failed

如果没有调用 `UDTMetaData.keepJavaFile()` 或将它设置为 `FALSE`，则驱动程序在执行 `UDTManager.createUDTClass()` 方法时移除已生成的 `.java` 文件。如果驱动程序无法移除 `.java` 文件，则会发生错误。

10. 5. 143 -79859

Invalid UDT field number

应用程序定义了 `UDTMetaData.setXXX()` 或 `UDTMetaData.getXXX()` 方法并指定字段数小于 0 或者大于 `UDTMetaData.setFieldCount()` 方法设置的值。

10. 5. 144 -79860

Ambiguous java type(s) - can't use Object/SQLData as method argument(s)

要注册为 UDR 的方法一个或多个参数的类型为 `java.lang.Object` 或 `java.sql.SQLData`。这些 Java™ 数据类型可以映射到多个 GBase 8s 数据类型，因此，驱动程序无法选择类型。

避免使用 `java.lang.Object` 或 `java.sql.SQLData` 作为方法参数。

10. 5. 145 -79861

Specified UDT field type has no Java™ type match

应用程序调用了 `UDTMetaData.setFieldType()` 并指定了一个在 Java 中没有 100% 匹配的数据类型。以下数据类型在此类别中：

```
lfxTypes.IFX_TYPE_BYTE  
lfxTypes.IFX_TYPE_TEXT  
lfxTypes.IFX_TYPE_VARCHAR  
lfxTypes.IFX_TYPE_NVARCHAR  
lfxTypes.IFX_TYPE_LVARCHAR
```

请使用 `IFX_TYPE_CHAR` 或 `IFX_TYPE_NCHAR`；这些数据类型映射到 `java.lang.String`。

10. 5. 146 -79862

Invalid UDT field type

应用程序调用了 `UDTMetaData.setFieldType()` 并为不透明类型指定了不支持的数据类型。有关支持的数据类型，请参阅映射字段类型。

10. 5. 147 -79863

UDT field length was not set in UDTMetaData

应用程序通过调用 `UDTMetaData.setFieldType()` 指定了一个字符型、日期时间型或 `interval` 类型的字段，但是指定字段长度时失败。调用 `UDTMetaData.setFieldLength()` 来设置字段长度。

10. 5. 148 -79864

Statement length exceeds the maximum

应用程序发出了一个长度大于数据库服务器可以处理的 `SQL PREPARE`、`DECLARE` 或 `EXECUTE IMMEDIATE` 语句。该限制根据实现方式会有所不同。但是大多数情况下，最长为 32,000 个字符。

检查程序逻辑确保错误不会导致程序显示比预期更长的字符串。如果文本具有预期的长度，则修改应用程序，一次显示更少的语句。

它与数据库服务器返回的 -460 错误相同。

10. 5. 149 -79865

Statement already closed

当应用程序在 `stmt.close()` 方法后尝试访问一个语句方法时，会发生此错误。

10. 5. 150 79868

Result set not open, operation not permitted

当应用程序尝试在 `ResultSet.close()` 方法后访问 `ResultSet` 方法，会发生此错误。

10. 5. 151 -79877

Invalid parameter value for setting maximum field size to a value less than zero

当应用程序尝试将最大字段大小设置为小于零的值时，会发生此错误。

10. 5. 152 -79878

Result set not open, operation next not permitted. Verify that autocommit is OFF

当应用程序在不执行结果集查询时尝试访问 `ResultSet.next()` 方法，会导致此错误。

10. 5. 153 -79879

An unexpected exception was thrown. See next exception for details

当出现非 SQL 异常时会发生此错误；例如，IO 异常。

10. 5. 154 -79880

Unable to set JDK Version for the Driver

当驱动程序无法从 Java™ 虚拟机获得 JDK 版本时，会发生此错误。

10. 5. 155 -79881

Already in local transaction, so cannot start XA transaction

在处理本地事务期间，当应用程序尝试启动 XA 事务时，会发生此错误。

GBASE[®]

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.



微信二维码

■ ■ 技术支持热线：400-013-9696

