|  |
| --- |
| ZXCLOUD GoldenDB  分布式数据库  开发指导 |
|  |
| 产品版本：V1.1P1 |

# 目录

[目录 2](#_Toc31482)

[1 前言 4](#_Toc5655)

[1.1 编写目的 4](#_Toc1638)

[1.2 预期读者 4](#_Toc8184)

[1.3 参考资料 4](#_Toc17775)

[2 开发指导 4](#_Toc10609)

[2.1 JDBC连接 4](#_Toc16235)

[2.1.1 创建数据库连接 4](#_Toc14835)

[2.1.2 释放资源 5](#_Toc19878)

[2.1.3 数据库连接池 5](#_Toc7831)

[2.2 Statement测试（EmpSQLStatementTest.java） 8](#_Toc1495)

[2.2.1 JDBC使用步骤过程 8](#_Toc1871)

[2.2.2 Insert statement 9](#_Toc19696)

[2.2.3 Update statement 10](#_Toc3814)

[2.2.4 Delete statement 11](#_Toc32165)

[2.2.5 Select statement 11](#_Toc3054)

[2.2.6 sequence使用 12](#_Toc10291)

[2.2.7 Procedure调用 15](#_Toc14841)

[2.2.8 非自动提交事务场景 16](#_Toc23832)

[2.3 JDBC参数测试 17](#_Toc1499)

[2.3.1 jdbc参数配置方法 17](#_Toc17122)

[2.3.2 常用参数列表 18](#_Toc23754)

[2.3.3 最佳实践 22](#_Toc30540)

[2.4 FAQ 24](#_Toc7563)

[2.4.1 useOldAliasMetadataBehavior参数使用 24](#_Toc32300)

[2.4.2 单个DBProxy发生异常导致应用启动失败问题分析 28](#_Toc1049)

[2.4.3 rewriteBatchedStatements和prepare使用注意事项 29](#_Toc21776)

[2.4.4 使用mybatis框架查询数据乱码 30](#_Toc27491)

[2.5 JDBC 修改列表 31](#_Toc14452)

[2.5.1 JDBC获取proxy建链消息中的proxy实际IP+port，并在duplicate新链路发送kill query时使用proxy的实际IP port 31](#_Toc24082)

[2.5.2 JDBC通过参数配置支持服务端prepare对读写的支持 31](#_Toc20472)

[2.5.3 去除写语句执行前的readonly判断，悲观模式改成乐观模式 32](#_Toc30331)

[2.5.4 JDBC记录与proxy的通信码流栈，异常时输出码流栈信息，打印源端与本端真实的IP port，用于配合抓包分析通讯故障 32](#_Toc24834)

[2.5.5 LoadBanlance协议支持连接级负载均衡,在URL中增加一个参数isConnectionLevel=true 同时使用LoadBanlance协议即可打开此功能 32](#_Toc20266)

[2.5.6 JDBC增加连接自管理能力、增加黑名单管理能力 32](#_Toc25517)

[2.5.7 jdbc常用参数默认值优化 32](#_Toc4258)

[2.5.8 JDBC loadbalance支持将计算节点按分组优先级 33](#_Toc29688)

[2.5.9 JDBC insert批量新协议 33](#_Toc20042)

[2.5.10 JDBC支持设置proxy下发group 33](#_Toc10632)

[2.5.11 JDBC驱动优化之支持版本识别 34](#_Toc13218)

[2.5.12 某些场景下游标失效 34](#_Toc29319)

[2.5.13 优化读buffer内存占用 34](#_Toc19503)

[2.5.14 优化连接回收机制 34](#_Toc12895)

[2.5.15 栈信息打印IP+PORT 34](#_Toc17349)

[2.5.16 CVE-2021-2471漏洞修复 35](#_Toc11106)

[2.5.17 自定义lodbalance报错提示优化 35](#_Toc10769)

[2.5.18 CVE-2020-2875漏洞修复 35](#_Toc12959)

[2.5.19 打印建链相关日志 35](#_Toc23447)

[2.5.20 原生loadbalance修改 35](#_Toc26483)

[2.5.21 高版本mybatis无法关闭prepare 35](#_Toc898)

[2.6 JDBC 保活说明 36](#_Toc13467)

# 前言

## 编写目的

本文档展示在已有工程的测试基础，以及如何在当前工程下添加二次开发需求，测试GoldenDB数据库的功能。

## 预期读者

业务开发人员及GoldenDB运维人员，测试人员。

## 参考资料

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **序号** | **文档名称** | **最后修订时间** | **版本号** | **来源** | **作者** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 开发指导

业务连接GoldenDB，推荐使用5.1.46版本的自研驱动包。

## JDBC连接

### 创建数据库连接

GoldenDB实现了完整的Mysql协议，所以GoldenDB能够完全复用Mysql的连接方式，在使用Java 连接 MySQL 需要驱动包，最新版下载地址为：[https://artnj.zte.com.cn/artifactory/webapp/#/artifacts/browse/tree/General/goldendb-alpha-generic/JDBC](https://artnj.zte.com.cn/artifactory/webapp/" \l "/artifacts/browse/tree/General/goldendb-alpha-generic/JDBC)，GoldenDB同样需要得到jar库文件，然后在对应的项目中导入该库文件。

运行Java连接数据库并作相关操作分为三个步

1. 加载数据访问驱动:Class.forName("com.goldendb.jdbc.Driver");
2. 连接到数据"库"上去

Connection conn = DriverManager.getConnection("jdbc:goldendb://127.0.0.1:3306/lt","root","zte");

DriverManager:驱动管理器类，getConnection（url,数据库登录名,密码）：获得连接的方法

jdbc:goldendb://localhost:3306/mydb的格式由来

jdbc:goldendb://本机地址：端口号/数据库名称

1. 构建执行SQL命令

Statement state = conn.createStatement();

state.executeUpdate("增删改的sql语句");

state.executeQuery("查询的sql语句");

conn.close();//关闭连接，final中执行

|  |
| --- |
| **public** **static** Connection getConnection() {  Connection conn = **null**;  **try** {  String user = Config.*user*;  String passwd = Config.*password*;  String url = Config.*url*;  Class.*forName*(Config.*driver*);// 指定连接类型  conn = DriverManager.*getConnection*(url, user, passwd);  } **catch** (Exception e) {  e.printStackTrace();  }  **return** conn;  } |

### 释放资源

JDBC中的Mysql数据库连接Connection和Statement、Resultset使用完毕之后一定要关闭，否则会占用大量内存资源，导致内存溢出。所以动态创建的连接要及时释放。

JDBC连接池提供了JDBC连接定义和数目有限的连接，如果数量不够，就需要长时间的等待。不正常关闭JDBC连接会导致等待回收无效的JDBC连接。只有正常的关闭和释放JDBC连接，JDBC 资源才可以被快速的重用使性能得到改善。

失败的关闭和释放 JDBC 连接可能导致其它用户的连接经历长时间的等待。虽然超时的JDBC 连接会被 WebSphere Application Server 退回而被回收 ，但必须等待这种情形发生。使用完 JDBC 资源后关闭它们，还可以显式关闭 JDBC ResultSets。如果没有显式关闭语句，则在完成了相关语句之后会释放 ResultsSets。

所以请确保您构建的代码在所有情况下，甚至在异常和错误条件下，都能关闭和释放 JDBC 资源。以下代码显示了 JDBC 资源的获得和使用都封装在“Try-Catch-Finally”结构中。其中，在finally 子句中处理 JDBC 资源的关闭，使所有情况下关闭都将发生。该方法需要在所有的finallly块中被使用。

|  |
| --- |
| **public** **static** **void** close(AutoCloseable close) {  **if** (**null** != close) {  **try** {  close.close();  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  } |

### 数据库连接池

为了解决资源的频繁分配﹑释放所造成的问题，可以采用数据库连接池技术。数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量﹑使用情况，为系统开发﹑测试及性能调整提供依据。

数据库连接是一种关键的有限的昂贵的资源，这一点在多用户的网页应用程序中体现得尤为突出。对数据库连接的管理能显著影响到整个应用程序的伸缩性和健壮性，影响到程序的性能指标。数据库连接池正是针对这个问题提出来的。数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而再不是重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。这项技术能明显提高对数据库操作的性能。

目前市场上有很多开源的Java开源数据连接池，广泛使用的主要有C3P0、DBCP、BoneCP、Proxool等。虽然他们在与数据库建链和连接释放上的性能差别很大（十分之一毫秒到数毫秒不等），但是在业务访问数据库的整个流程中，性能瓶颈从来都不在连接池上，实际进行带业务的性能测试在不同连接池之间的性能差异不会超过百分之三。因此我们在连接池选型时更多地应该关注它们的稳定性和易用性。

各种连接池在GoldenDB上的验证结果及比较如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 简介 | 使用评价 | GoldenDB验证结果 | 验证说明 |
| C3P0 | C3P0是一个开放源代码的JDBC连接池，它在lib目录中与Hibernate一起发布,包括了实现jdbc3和jdbc2扩展规范说明的Connection 和Statement 池的DataSources 对象。 | 连接池可以设置最大和最小连接，连接等待时间等，基本功能都有，连接池的持续运行的稳定性相当不错，在大并发量的压力下稳定性也有一定保证，此外不提供连接池监控。 | 通过 | C3P0版本：c3p0-0.9.2.1  JRE版本：Java5.0以上  C3P0是这几个连接池中最稳定的一个，虽然性能比DBCP略差，但是稳定性和功能丰富性强于DBCP。 |
| DBCP | DBCP是一个依赖Jakarta commons-pool对象池机制的[数据库](http://lib.csdn.net/base/mysql" \t "http://blog.csdn.net/u013076997/article/details/_blank" \o "MySQL知识库)连接池。作为tomcat6之前版本的内置连接池，DBCP也可以直接的在应用程序用使用。 | 可以设置最大和最小连接，连接等待时间等，基本功能都有，此连接池的持续运行的稳定性还是可以，不过速度稍慢，在大并发量的压力下稳定性有所下降，此外不提供连接池监控 | 通过 | DBCP版本：commons-dbcp2-2.1.1  JRE版本：Java7.0以上  依赖jar包：  commons-pool2-2.4.2.jar  commons-io-2.4.jar  commons-logging-1.1.3.jar  Dbcp2.0以上的版本需要jdk7.0以上版本才能编译；而2.0之前的版本中存在一些缺陷，如url中配置多个地址的负载时，只要存在异常的url则系统无法启动等。  此外，由于在[hibernate](http://jump2.bdimg.com/safecheck/index?url=rN3wPs8te/pL4AOY0zAwhz3wi8AXlR5gsMEbyYdIw61+GS81akzbRQDBLbJ33gWMImOuUl9obIdesUqvhcRz+hXSuwz9b96EragWmZ1jer1XQuxoOXhQsccfbAXwgsq2bEOipeqEGF3QrKBBW2u4u1S/cU3HYvzimbonRmV509PHxtw9jvhvsvAlZvyfl1HuSfn0Qh5FfjV2Pah4egTNMg==" \t "https://tieba.baidu.com/_blank)3.0中，已经不再支持dbcp了，hibernate的作者在hibernate.org中，明确指出在实践中发现dbcp有 BUG,在某些种情会产生很多空连接不能释放，所以抛弃了对dbcp的支持。暂不清楚这些问题在2.0以上版本中是否已经解决。 |
| BoneCP | BoneCP是一个快速，开源的数据库连接池。帮你管理数据连接让你的应用程序能更快速地访问数据库。比C3P0/DBCP连接池快25倍 | 一种新的数据连接技术，以其效率高，速度 快著称，连接池可以设置最大和最小连接，连接等待时间等，基本功能都有 | 通过 | boneCP版本：bonecp-0.7.1.RELEASE  JRE版本：Java5.0以上  依赖jar包：  log4j-1.2.15.jar  slf4j-api-1.7.6.jar  slf4j-log4j12-1.7.6.jar  google-collections-1.0.jar  获取和释放连接最快的连接池。缺陷是引入的包比较多。 |
| Proxool | 这是一个Java SQL Driver驱动程序，提供了对你选择的其它类型的驱动程序的连接池封装。可以非常简单的移植到现存的代码中。完全可配置。快速，成熟，健壮。可以透明地为你现存的JDBC驱动程序增加连接池功能。 | 连接池可以设置最大和最小连接，连接等待时间等，基本功能都有，连接池的持续运行的稳定性有一定问题，有一个优势--连接池监控 | 未验证 | 由于稳定性有点问题，不推荐使用。 |

示例，C3P0数据库连接池创建连接：

|  |
| --- |
| **private** **static** ComboPooledDataSource *ds*;  // 静态初始化块进行初始化  **static** {  **try** {  *ds* = **new** ComboPooledDataSource();// 创建连接池实例  *ds*.setDriverClass(Config.*driver*);// 设置连接池连接数据库所需的驱动  *ds*.setJdbcUrl(Config.*url*);// 设置连接数据库的URL  *ds*.setUser(Config.*user*);// 设置连接数据库的用户名  *ds*.setPassword(Config.*password*);// 设置连接数据库的密码  *ds*.setMaxPoolSize(Config.*totalPoolSize*);// 设置连接池的最大连接数  *ds*.setMinPoolSize(Config.*initSize*);// 设置连接池的最小连接数  *ds*.setInitialPoolSize(Config.*initSize*);// 设置连接池的初始连接数  } **catch** (Exception e) {  e.printStackTrace();  }  }  // 获取与指定数据库的连接  **public** **static** ComboPooledDataSource getInstance() {  **return** *ds*;  }  // 从连接池返回一个连接  **public** **static** Connection getConnection() {  Connection conn = **null**;  **try** {  conn = *ds*.getConnection();  } **catch** (Exception e) {  e.printStackTrace();  }  **return** conn;  } |

## Statement测试（EmpSQLStatementTest.java）

本例中展示工程中自带的，使用emp表进行增删改查，以及sequence的使用和存储过程的调用等功能，首先进行初始化库表信息：

|  |
| --- |
| **drop** database if **exists** test\_dev;  **create** database test\_dev;  **CREATE** **TABLE** test\_dev.emp(  empno **INT** **PRIMARY** **KEY**,  ename **VARCHAR**(10),  job **VARCHAR**(9),  hiredate **DATE**,  sal **decimal**(7,2)  )distributed **by** hash(empno)(g1,g2,g3); |

### JDBC**使用**步骤过程

* 获取数据库连接
* 创建一个Statement，要执行SQL语句，必须获得java.sql.Statement实例，Statement实例分为以下3种类型：

1. 执行静态SQL语句。通常通过Statement实例实现
2. 执行动态SQL语句。通常通过PreparedStatement实例实现，建议使用，不会造成SQL注入。
3. 执行数据库存储过程。通常通过CallableStatement实例实现

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

1. ResultSet executeQuery(String sqlString)：执行查询数据库的SQL语句，返回一个结果集（ResultSet）对象。
2. int executeUpdate(String sqlString)：用于执行INSERT、UPDATE或DELETE语句以及SQL DDL语句，如：CREATE TABLE和DROP TABLE等
3. execute(sqlString):用于执行返回多个结果集、多个更新计数或二者组合的语句。
4. executeBatch():执行批量，在执行批量事务的时候使用。

* 获取结果，结果会返回两种情况：

1. 执行更新返回的是本次操作影响到的记录数。
2. 执行查询返回的结果是一个ResultSet对象。ResultSet包含符合SQL语句中条件的所有行，并且它通过一套get方法提供了对这些行中数据的访问。

* 关闭JDBC对象，操作完成以后要把所有使用的JDBC对象全都关闭，以释放JDBC资源，关闭顺序和声明顺序相反：先关闭记录集再关闭关闭声明

### Insert statement

实现步骤：

1. 获取数据库连接DBUtils.*getConnection*
2. 创建一个PreparedStatement执行动态SQL语句，语句中的变量使用“？”代替
3. 设置参数，参数的序号从1开始，参数的顺序和个数与“？”的个数对应
4. 执行executeUpdate，返回插入成功数
5. Finally语句块中清理资源

|  |
| --- |
| **public** **void** insertEmp() {  PreparedStatement ps = **null**;  Connection conn = **null**;  **try** {  conn = DBUtils.*getConnection*();  // 预处理添加数据  ps = conn  .prepareStatement("insert into test\_dev.emp (empno,ename,job,hiredate,sal) " + "values(?,?,?,?,?)");  // 设置参数  ps.setInt(1, NumberUtils.*getRandomInt*());  ps.setString(2, "张三");  ps.setString(3, "DBA");  DateFormat dateFormat2 = **new** SimpleDateFormat("yyyy-MM-dd");  Date myDate2 = dateFormat2.parse("2010-09-13");  ps.setDate(4, **new** java.sql.Date(myDate2.getTime()));  ps.setFloat(5, (**float**) 2000.3);  // 执行更新  System.***out***.println("执行插入成功" + ps.executeUpdate() + "条");  } **catch** (SQLException e) {  e.printStackTrace();  } **catch** (ParseException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  } |

### Update statement

实现步骤：

1. 获取数据库连接DBUtils.*getConnection*
2. 创建一个PreparedStatement执行动态SQL语句，语句中的变量使用“？”代替
3. 设置参数，参数的序号从1开始，参数的顺序和个数与“？”的个数对应
4. 执行executeUpdate，返回插入成功数
5. Finally语句块中清理资源

|  |
| --- |
| **public** **void** updateEmp() {  PreparedStatement ps = **null**;  Connection conn = **null**;  **try** {  conn = DBUtils.*getConnection*();  // 预处理添加数据  ps = conn.prepareStatement("update test\_dev.emp set sal = ? where ename = ?");  ps.setFloat(1, (**float**) 5000.0);  ps.setString(2, "张三");  // 执行更新  System.***out***.println("执行更新成功" + ps.executeUpdate() + "条");  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  } |

### Delete statement

实现步骤：

1. 获取数据库连接DBUtils.*getConnection*
2. 创建一个PreparedStatement执行动态SQL语句，语句中的变量使用“？”代替
3. 设置参数，参数的序号从1开始，参数的顺序和个数与“？”的个数对应
4. 执行executeUpdate，返回插入成功数
5. Finally语句块中清理资源

|  |
| --- |
| **public** **void** deleteEmp() {  PreparedStatement ps = **null**;  Connection conn = **null**;  **try** {  conn = DBUtils.*getConnection*();  // 预处理添加数据  ps = conn.prepareStatement("delete from test\_dev.emp where ename = ?");  ps.setString(1, "张三");  // 执行更新  System.***out***.println("执行删除成功" + ps.executeUpdate() + "条");  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  } |

### Select statement

实现步骤：

1. 获取数据库连接DBUtils.*getConnection*
2. 创建一个PreparedStatement执行动态SQL语句，语句中的变量使用“？”代替
3. 设置参数，参数的序号从1开始，参数的顺序和个数与“？”的个数对应
4. 执行executeQuery，返回的结果是一个ResultSet对象。ResultSet包含符合SQL语句中条件的所有行，并且它通过一套get方法提供了对这些行中数据的访问。
5. 获取结果数据
6. Finally语句块中清理资源

|  |
| --- |
| **public** **void** selectEmp() {  PreparedStatement ps = **null**;  Connection conn = **null**;  ResultSet rs = **null**;  **try** {  conn = DBUtils.*getConnection*();  // 预处理添加数据  ps = conn.prepareStatement("select \* from test\_dev.emp where id=?");  ps.setInt(1, 0);  rs = ps.executeQuery();  **while** (rs.next()) {  // 输出结果  System.***out***.println(rs.getString("job") + "\t" + rs.getString("ename"));  }  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(rs);  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  } |
|  |

### sequence使用

GoldenDB数据库序列(SEQUENCE)其实是序列号生成器，可以为表中的行自动生成序列号，产生一组等间隔的数值(类型为数字)。其主要的用途是生成表的主键值，可以在插入语句中引用，也可以通过查询检查当前值，或使序列增至下一个值。

创建序列需要CREATE SEQUENCE系统权限。序列的创建语法如下：

CREATE SEQUENCE 序列名 [INCREMENT BY n] [START WITH n] [{MAXVALUE/ MINVALUE n|NOMAXVALUE}] [] [{CACHE n|NOCACHE}];

INCREMENT BY 用于定义序列的步长，如果省略，则默认为1，如果出现负值，则代表序列的值是按照此步长递减的。

START WITH 定义序列的初始值(即产生的第一个值)，默认为1。

MAXVALUE 定义序列生成器能产生的最大值。选项NOMAXVALUE是默认选项，代表没有最大值定义，这时对于递增序列，系统能够产生的最大值是10的27次方;对于递减序列，最大值是-1。

MINVALUE定义序列生成器能产生的最小值。选项NOMAXVALUE是默认选项，代表没有最小值定义，这时对于递减序列，系统能够产生的最小值是-10的26次方;对于递增序列，最小值是1。

CYCLE和NOCYCLE表示当序列生成器的值达到限制值后是否循环。CYCLE代表循环，NOCYCLE代表不循环。如果循环，则当递增序列达到最大值时，循环到最小值;对于递减序列达到最小值时，循环到最大值。如果不循环，达到限制值后，继续产生新值就会发生错误。

CACHE(缓冲)定义存放序列的内存块的大小。NOCACHE表示不对序列进行内存缓冲。对序列进行内存缓冲，可以改善序列的性能。

删除序列的语法是：

DROP SEQUENCE 序列名;

注意：GoldenDB在使用序列的时候，包括创建，删除和使用都需要加上database的名字。

在使用SEQUENCE的时候首先需要通过Proxy创建序列：

|  |
| --- |
| **create** sequence test\_dev.emp\_seq  increment **by** 1  start **with** 1  nocycle  cache 10; |

下例中展示如何使用GoldenDB的序列替代mysql自带的auto\_increment作为自增列使用，使用test\_dev.emp\_seq.nextval获取序列的下一个值：

|  |
| --- |
| // 预处理添加数据  ps = conn  .prepareStatement("insert into test\_dev.emp (empno,ename,job,hiredate,sal) " + "values(test\_dev.emp\_seq.nextval,?,?,?,?)"); |
|  |

使用test\_dev.emp\_seq.currval获取序列的当前值：

|  |
| --- |
| ps1 = conn.prepareStatement("select test\_dev.emp\_seq.currval from dual");  rs = ps1.executeQuery();    **if** (rs.next())  {  System.***out***.println(rs.getInt(1));  }  if (null != rs)  {  try  {  rs.close();  }  catch (SQLException e)  {  e.printStackTrace();  }  }  if (null != ps)  {  try  {  ps.close();  }  catch (SQLException e)  {  e.printStackTrace();  }  }  if (null != ps1)  {  try  {  ps.close();  }  catch (SQLException e)  {  e.printStackTrace();  }  }  if (conn != null)  {  try  {  conn.close();  }  catch (SQLException e)  {  e.printStackTrace();  }  } |

### Procedure调用

JDBC调用存储过程: CallableStatement

在Java里面调用存储过程，写法那是相当的固定：

/\*\*

\*p是要调用的存储过程的名字，存储过程的4个参数，用4个？号占位符代替

\*其余地方写法固定

\*/

CallableStatement cstmt = conn.prepareCall("{call p(?,?,?,?)}");

/\*\*

\*告诉JDBC，这些个参数，哪些是输出参数，输出参数的类型用java.sql.Types来指定

\*下面的意思是，第3个？和第4个？是输出参数，类型是INTEGER的

\*Types后面具体写什么类型，得看你的存储过程参数怎么定义的

\*/

cstmt.registerOutParameter(3, Types.INTEGER);

cstmt.registerOutParameter(4, Types.INTEGER);

/\*\*

\*在我这里第1个？和第2个？是输入参数，第3个是输出参数，第4个既输入又输出

\*下面是设置他们的值,第一个设为3，第二个设为4，第4个设置为5

\*没设第3个，因为它是输出参数

\*/

cstmt.setInt(1, 3);

cstmt.setInt(2, 4);

cstmt.setInt(4, 5);

//执行

cstmt.execute();

GoldenDB不建议存储过程拥有返回值，GoldenDB调用存储过程的时候是直接下发到DB去执行的。

使用proxy创建一个存储过程，该存储过程的作用是清理表数据，传参是表名：

|  |
| --- |
| **drop** **procedure** if **exists** p;  DELIMITER //  **create** **PROCEDURE** p(**in** tbname **varchar**(64))  **BEGIN**  **declare** v\_sql **varchar**(100);  **set** v\_sql = concat('truncate table ',tbname);  **set** @v\_sql=v\_sql;  prepare stmt **from** @v\_sql;  EXECUTE stmt;  deallocate prepare stmt;  **END** DISTRIBUTED **BY** DUPLICATE(G1,G2,G3);  //  DELIMITER ; |

存储过程调用实例如下：

|  |
| --- |
| **public** **void** procedureTest() {  CallableStatement cs = **null**;  Connection conn = **null**;  **try** {  conn = DBUtils.*getConnection*();  cs = conn.prepareCall("{call test\_dev.p(?)}");  cs.setString(1, "test\_dev.emp");  // 执行  cs.execute();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(cs);  DBUtils.*close*(conn);  }  } |

### 非自动提交事务场景

上面的案例涉及的都是单条语句的执行，并且设置的都是自动提交模式，在多条sql的场景下，需要设置autocommit为false开启非自动提交，当提交失败或在事务异常时需要捕获异常提交事务回滚sql。

|  |
| --- |
| Connection conn = **null**;  **public** **void** insertEmp() {  PreparedStatement ps = **null**;    **try** {  conn = DBUtils.*getConnection*();  conn.setAutoCommit(false);  // 预处理添加数据  ps = conn  .prepareStatement("insert into test\_dev.emp (empno,ename,job,hiredate,sal) " + "values(?,?,?,?,?)");  // 设置参数  ps.setInt(1, NumberUtils.*getRandomInt*());  ps.setString(2, "张三");  ps.setString(3, "DBA");  DateFormat dateFormat2 = **new** SimpleDateFormat("yyyy-MM-dd");  Date myDate2 = dateFormat2.parse("2010-09-13");  ps.setDate(4, **new** java.sql.Date(myDate2.getTime()));  ps.setFloat(5, (**float**) 2000.3);  // 执行更新  System.***out***.println("执行插入成功" + ps.executeUpdate() + "条"); Commit();  } **catch** (SQLException e) {  e.printStackTrace();  } **catch** (ParseException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  }  **private** **void** Commit() {  **try** {  conn.commit();  } **catch** (SQLException e) {  System.***out***.println(e);  **try** {  **if** (**null** != conn && conn.isValid(10)) {  // 回滚事务  conn.rollback();  }  } **catch** (Exception e1) {  e1.printStackTrace();  DBUtils.*close*(conn, **null**, **null**);  // 对新连接的验证  conn = DBUtils.*getConn*();  }  }  } |
|  |

## JDBC参数测试

### jdbc参数配置方法

配置jdbc参数有2种配置方法，

方法1：配置方法是直接在url中配置，配置格式如下：

jdbc:goldendb://[host:port]/[database][?参数名1][=参数值1][&参数名2][=参数值2]...

例如：jdbc:goldendb://10.47.161.46:9271?characterEncoding=utf8，该url配置中配置了参数characterEncoding，参数值为utf8.

方法2：将属性写入属性列表，并在创建连接的时候使用，代码示例如下：

|  |
| --- |
| **try** {  Properties props = **new** Properties();  props.setProperty("useSSL", "false");  props.setProperty("user", **this**.user);  props.setProperty("password", **this**.password);  **this**.conn = DriverManager.*getConnection*(*dbUrl*, props);  } **catch** (Exception ex) {  ex.printStackTrace();  *fail*();  } |

### 常用参数列表

|  |  |  |
| --- | --- | --- |
| 参数名 | 使用介绍 | 推荐值 |
| useSSL | 与服务器进行通信时使用SSL（真／假），默认值为“假”。哦Url配置参数中需要添加该参数，否则会抛出告警信息：Sun Mar 11 20:22:41 CST 2018 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification. | false |
| loadBalanceStrategy | 负载均衡的策略，默认值为“random”；其支持简写值为“random”、“bestResponseTime”（最小响应时间优先）；对于其他负载均衡策略，需要指定类的全名。  实例见LB协议的使用示例。 | random |
| useServerPrepStmts | 如果服务器支持，是否使用服务器端预处理语句？ 官方默认值为“false”，产品包中已修改为“true”。 | true |
| cachePrepStmts | 驱动程序是否应对客户端预处理语句的PreparedStatements的解析过程执行缓冲处理，是否应检查服务器端预处理语句的适用性以及服务器端预处理语句本身。官方默认值为“false”，产品包中已修改为“true” | true |
| prepStmtCacheSqlLimit | 如果允许预处理语句缓冲功能，驱动程序将执行解析缓冲处理的最大SQL。官方默认值为256，产品包中已修改为2000 | 2000 |
| prepStmtCacheSize | 如果允许预处理语句缓冲功能，应缓冲处理多少条预处理语句，官方默认值为25，产品包中已修改为2000 | 2000 |
| characterEncoding | 如果“useUnicode”被设置为“真”，处理字符串时，驱动程序应使用什么字符编码？ 默认为“autodetect”。 | autodetect |
| useUnicode | 处理字符串时，驱动程序是否应使用Unicode字符编码？ 仅应在驱动程序无法确定字符集映射，或你正在强制驱动程序使用MySQL不是固有支持的字符集时（如UTF-8）才应使用。真／假，默认为“真”。 | true |
| characterSetResults | 字符集，用于通知服务器以何种字符集返回结果。 | utf8mb4 |
| useOldAliasMetadataBehavior | 驱动程序是否应该将旧行为用于列和表的“AS”子句，并且只返回ResultSetMetaData.getColumnName（）或ResultSetMetaData.getTableNaconvertToNullme（）的别名（如果有）而不是原始列/表名。 在5.0.x中，默认值为true。 | false |
| zeroDateTimeBehavior | JAVA连接MySQL数据库，在操作值为0的timestamp类型时不能正确的处理，而是默认抛出一个异常，就是所见的：java.sql.SQLException: Cannot convert value '0000-00-00 00:00:00' from column 7 to TIMESTAMP。  在JDBC连接串中有一项属性：zeroDateTimeBehavior,可以用来配置出现这种情况时的处理策略，该属性有下列三个属性值：  exception：默认值，即抛出SQL state [S1009]. Cannot convert value....的异常；  convertToNull：将日期转换成NULL值；  round：替换成最近的日期即0001-01-01； | exception |
| maxRows | 返回的最大行数（0，默认值表示返回所有行）。 | -1 |
| autoReconnect | 驱动程序是否应尝试再次建立失效的和／或死连接？如果允许，对于在失效或死连接上发出的查询（属于当前事务），驱动程序将抛出异常，但在新事务的连接上发出下一个查询时，将尝试再连接。不推荐使用该特性，这是因为，当应用程序不能恰当处理SQLExceptions时，它会造成与会话状态和数据一致性有关的副作用，设计它的目的仅用于下述情况，即，当你无法配置应用程序来恰当处理因死连接和／或无效连接导致的SQLExceptions时。作为可选方式，可将MySQL服务器变量“wait\_timeout”设置为较高的值，而不是默认的8小时。 | false |
| useCursorFetch | 如果是从Java中连mysql,使用PrepareStatement的话，默认情况下真正发给服务器端之前已经把?替换了。也就是跟普通的Statement一样。在5.0开始虽然有了真正的PrepareStatement，开启方式是useCursorFetch=true | false |
| rewriteBatchedStatements | 当设置rewriteBatchedStatements=true时，通过驱动程序调用executeBatch()时，可以实现高性能的批量操作。 | true |
| connectTimeout | socket连接超时（以毫秒为单位），0表示无超时。仅适用于JDK-1.4或更高版本。 官方默认值为0，产品包中已修改为10000。 | 10000 |
| socketTimeout | 网络socket操作超时（以毫秒为单位）。默认值0表示没有超时。 | 0 |
| defaultFetchSize | 使用游标读取数据时，每批次默认读取行数 | 100 |
| cacheCallableStmts | 是否缓存调用存储过程的语句 | false |
| cacheResultSetMetadata | 是否启用缓存sql语句涉及的元数据功能，如果启用，则会将sql语句涉及的元数据缓存起来 | false |
| callableStmtCacheSize | 可以缓存存储过程语句的个数 | 100 |
| metadataCacheSize | 缓存sql语句涉及的元数据的个数,一个sql对应一个元数据结果对象 | 50 |
| enableQueryTimeouts | 是否启用查询语句超时检测功能 | true |
| largeRowSizeThreshold | 定义行记录长度为大数据的阀值 | 2048 |
| useLocalSessionState | 是否使用本地属性（autocommit，read\_only和transaction isolation） | false |
| netTimeoutForStreamingResults | JDBC会自动设置数据库的'net\_write\_timeout'变量为本参数配置的时间，单位是秒，值为0表示驱动不会调整该参数； 客户端超时断链时间:如果该参数大于0，JDBC驱动通过调用PreparedStatement的executeQuery（）方法默认执行"SET net\_write\_timeout= ？" 这个命令去重新设置timeout时间。 | 600 |
| noTimezoneConversionForDateType | 如果 'useTimezone'='true' or 'useLegacyDatetimeCode'='false' 配合本参数配置为TRUE，则不使用服务器时区 | true |
| queryTimeoutKillsConnection | 当设置queryTimeout超时时间时，执行语句返回超时，是否将连接中断 | true |
| useSqlStateCodes | 使用SQL标准状态码代替“旧版” X / Open / SQL状态码（true/false），默认值为“true” | true |
| useTimezone | 在客户端和服务器时区之间转换time/date类型（true/false，默认为“false”）？这是旧版日期时间代码的一部分，因此该属性仅在“ useLegacyDatetimeCode = true”时有效。 | false |
| maxReconnects | 如果autoReconnect为true，则尝试进行的最大重新连接数， 默认值为'3'。 | 3 |
| initialTimeout | 如果启用了autoReconnect，则两次尝试重新连接之间的初始 时间（以秒为单位，默认为2）。 | 2 |
| retriesAllDown | 使用负载平衡或故障转移时，驱动程序应在可用主机之间循环 尝试连接的次数。 在两个周期之间，如果没有可用服务器，驱动程序将暂停250ms。 | 120 |
| allowMultiQueries | 在一个语句（真/假）期间允许使用“;”分隔多个查询，默认为“假”，并且不影响依赖于rewriteBatchStatements的addBatch()和executeBatch()方法。 | false |
| maxAllowedPacket | 发送到服务器的最大允许数据包大小。 如果未设置，则使用数据库中的'max\_allowed\_packet'的值初始化它；如果设置的值大于“max\_allowed\_packet”，则该值不会生效。  另外，由于内部依赖项具有属性“blobSendChunkSize”，如果将“ useServerPrepStmts”设置为“true”，则此设置的最小值为“8203”。 | 取自  mysql.max\_allowed\_packet |
| logSlowQueries | 是否应该记录比  'slowQueryThresholdMillis'时间更长的查询？ | false |
| slowQueryThresholdMillis | 如果启用了“logSlowQueries”，查询耗时多长时间（以毫秒为单位），会被记录为“慢”？ | 2000 |
| slowQueryThresholdNanos | 如果将“useNanosForElapsedTime”设置为true，并且 此属性设置为非零值，则驱动程序将使用此阈值（以纳秒为单位）来确定查询是否缓慢。 | 0 |
| dumpListSize | 异常码流信息条数 | 0 |
| dumpInfoLength | 每条异常码流信息长度 | 1000 |
| isConnectionLevel | 使用LoadBanlance协议可打开此功能 如果为false或者不填则走原来的LoadBanlance协议逻辑 | false |
| intervalTime | 控制select 1发送间隔时间 单位毫秒 | 2000 |
| blackTaskTime | 黑名单移除间隔时间 单位秒 | 20 |
| shadowThreadSwitch | 监控线程开关 | TRUE |
| prepareStrategy | 通过该参数选择两种preparedStatement对象创建方式： 1、增删改使用客户端prepare，查询使用服务端prepare，2、全服务端prepare | 1 |
| errorCount | 异常次数判断。通过增加独立心跳检测,判断节点是否故障，统计心跳异常的次数，如果大于等于errorCount，则将该节点放入黑名单中，处于黑名单列表中的计算节点不会给连接池使用。隔离后在1分钟内再检查心跳后激活。 | 1 |
| locatorFetchBuffer Size | 缓冲区大小。缓冲区是在解析SQL时分配的，因此缓冲区的大小不取决于查询返回的行数据的实际大小，而取决于行数据可能的最大大小。 其值与以下三点有关：  （1）数据表的定义（2）代码查询的方式（3）FetchSize的值 | 12400 |

### 最佳实践

#### Loadbalance的使用

对于GoldenDB来说，proxy集群通常都是多个，对于proxy集群这种“多Master”架构，我们通常希望“负载均衡”、“读写分离”等高级特性，这就是Load Balancing协议所能解决的。Load Balancing可以将read/write负载，分布在多个MySQL实例上。

LB协议基于“Failover协议”，即具备Failover特性，其URL格式：

jdbc:goldendb:loadbalance://[host]:[port],[host]:[port],...[/database]?[property=<value>]&[property=<value>]

配置实例url为：

|  |
| --- |
| url=jdbc:goldendb:loadbalance://10.5.7.13:8888,10.5.6.121:8888?characterEncoding=utf8&useSSL=false |

#### Prepare的使用

当客户发送一条SQL语句给服务器后，服务器总是需要校验SQL语句的语法格式是否正确，然后把SQL语句编译成可执行的函数，最后才是执行SQL语句。其中校验语法，和编译所花的时间可能比执行SQL语句花的时间还要多。

如果我们需要执行多次insert语句，但只是每次插入的值不同，MySQL服务器也是需要每次都去校验SQL语句的语法格式，以及编译，这就浪费了太多的时间。如果使用预编译功能，那么只对SQL语句进行一次语法校验和编译，所以效率要高。

从Java中连mysql,使用PrepareStatement的话，默认情况下真正发给服务器端之前已经把?替换了。也就是跟普通的Statement一样：

|  |
| --- |
| // 预处理添加数据  ps = conn.prepareStatement("select \* from test\_dev.emp where empno = ?");  ps.setInt(1, 1);  rs = ps.executeQuery();  对select语句进行prepare模式设置，实际客户端发给服务端的语句为：  select \* from test\_dev.emp where empno = 1  对语句进行了替换。 |

为了实现预编译功能，设置参数useCursorFetch=true，再次执行该方法，抓取发送的sql，已经发送带？的语句：

|  |
| --- |
|  |

#### fetchsize批量的使用

MYSQL默认为从服务器一次取出所有数据放在客户端内存中，fetch size参数不起作用，当一条SQL返回数据量较大时可能会出现JVM OOM。

要一条SQL从服务器读取大量数据，不发生JVM OOM，可以采用以下方法之一：

1、当statement设置以下属性时，采用的是流数据接收方式，每次只从服务器接收部份数据，直到所有数据处理完毕，不会发生JVM OOM。

setResultSetType(ResultSet.TYPE\_FORWARD\_ONLY);

setFetchSize(Integer.MIN\_VALUE);

2、调用statement的enableStreamingResults方法，实际上enableStreamingResults方法内部封装的就是第1种方式。

3、设置连接属性useCursorFetch=true (5.0版驱动开始支持)，statement以TYPE\_FORWARD\_ONLY打开，再设置fetch size参数，表示采用服务器端游标，每次从服务器取fetch\_size条数据。

示例，url配置：

|  |
| --- |
| url=jdbc:goldendb://10.47.161.46:9271?characterEncoding=utf8&useCursorFetch=true&useSSL=false |

代码中添加红色标记的位置内容，即可以达到预取部分的功能：

|  |
| --- |
| **public** **void** testFetchSize()  {  PreparedStatement ps = **null**;  Connection conn = **null**;  ResultSet rs = **null**;  **try** {  conn = DBUtils.*getConnection*();  // 预处理添加数据  ps = conn.prepareStatement("select \* from test\_dev.emp");  // 设置每次预取的条数  ps.setFetchSize(1);  rs = ps.executeQuery();  **while** (rs.next()) {  // 输出结果  System.***out***.println(rs.getString("job") + "\t" + rs.getString("ename"));  }  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  // 清理资源  DBUtils.*close*(rs);  DBUtils.*close*(ps);  DBUtils.*close*(conn);  }  } |

## FAQ

### useOldAliasMetadataBehavior参数使用

‌问题现象：部分报表查询，在测试环境中正确，在上线环境中查询不正确，分析为列别名的命名问题。

（1）原因：别名与列名相同时，查询结果不正确

解决方法：修改了业务代码，规避了此问题，后更换了proxy程序，杜绝隐患。

（2）原因：别名与列名不同时，调用hibernate封装的jdbc做查询，获取不到这部分列值。

解决方法：改用原生jdbc方法调用此语句，可以正确获取到结果，因此怀疑是hibernate封装的jdbc的问题。后发现这是低版本hibernate的一个BUG，无法正确获取到列别名。在mysql url中添加

&useOldAliasMetadataBehavior=true

后，此问题解决

#### 验证流程一：

使用如下代码测试，测试使用jdbc驱动为mysql-connector-java-5.1.31，和现场一致：

|  |
| --- |
| **String sql = "select a as A from ltwdb.T as AA where b =2";**  **rs = sm.executeQuery(sql);**  **rsmd = rs.getMetaData();**  **System.out.println("ColumnName: "+rsmd.getColumnName(1) + " TableName:" + rsmd.getTableName(1));** |

验证如下四种场景，其中**场景3**的使用方式是出错应用的使用方式：

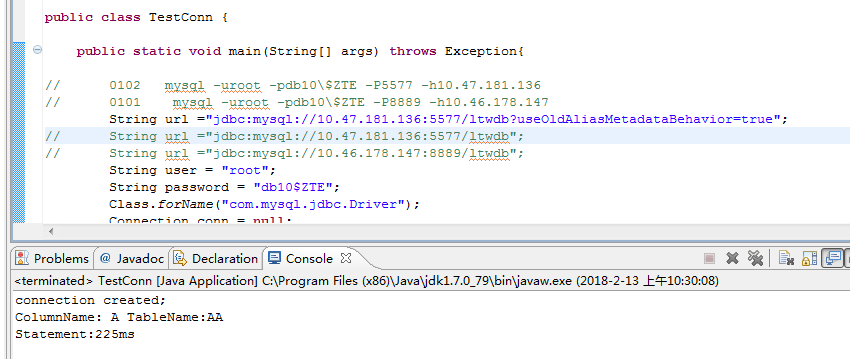
    1. “02版本+使用参数"

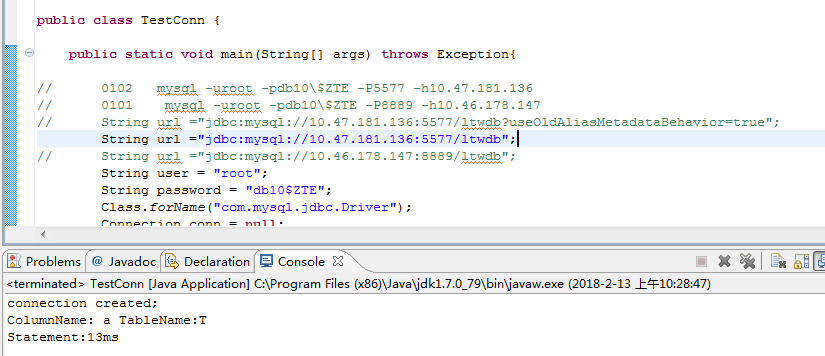
    2. "02版本+不使用参数"

    3. **"01版本不使用参数"**

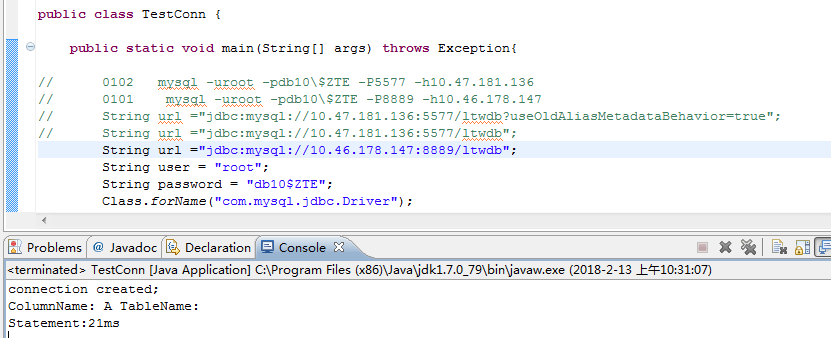
    4. "01版本使用参数"

打印结果如下。

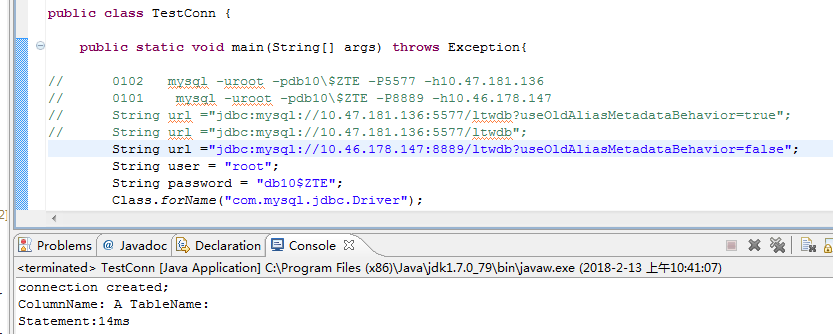
02版本+使用参数：

02版本+不使用参数

**01版本不使用参数：**

****

01版本使用参数：



**流程一结论：**

**针对列名ColumnName，场景1、3、4，都显示：“A”，场景2显示：“a”**

**针对表名TableName，场景1显示：“AA”，场景2显示“T”，场景3、4不显示表名。**

**即：**

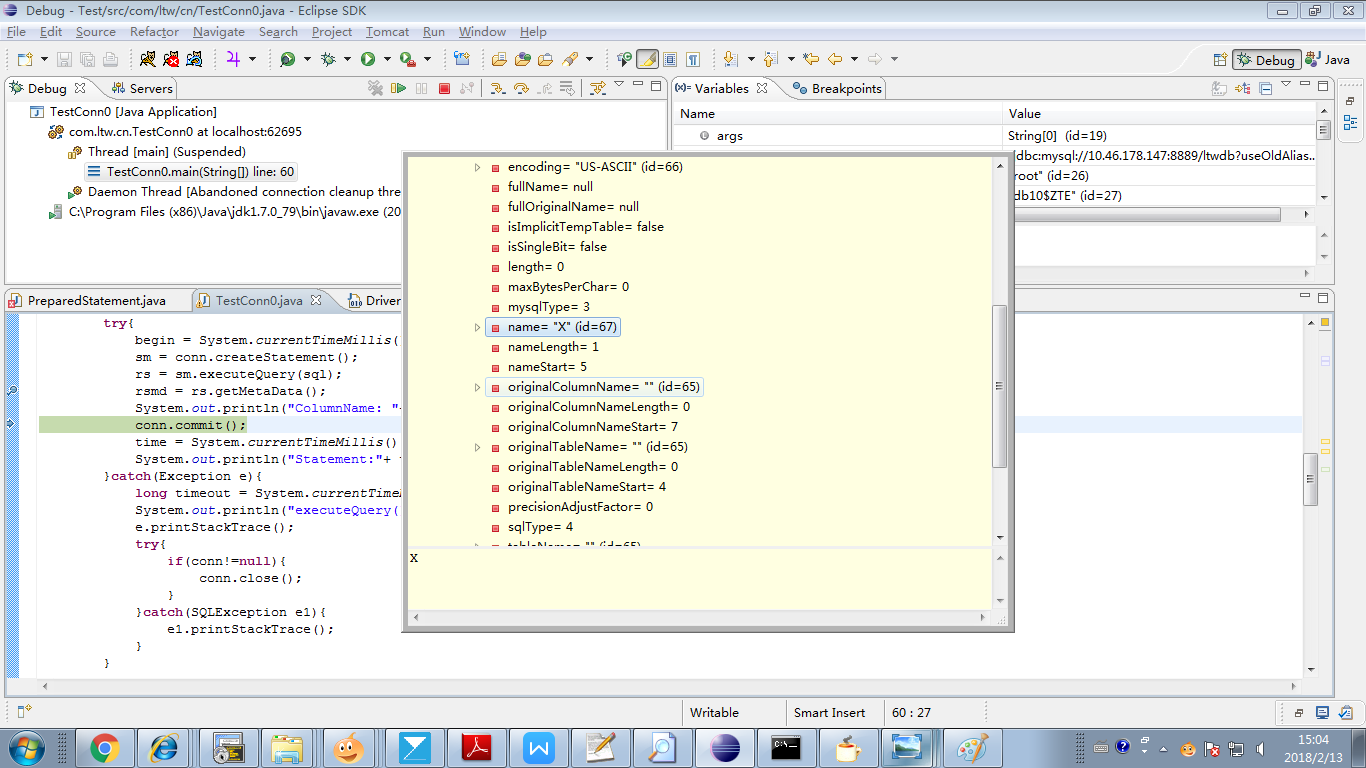
**1.参数“useOldAliasMetadataBehavior”对之前的0101版本不生效，因为场景3、4表现都一样，0102版本可以通过参数控制是否生效。**

**2. 当表名也设置别名时，0101版本未返回信息，0102支持**。

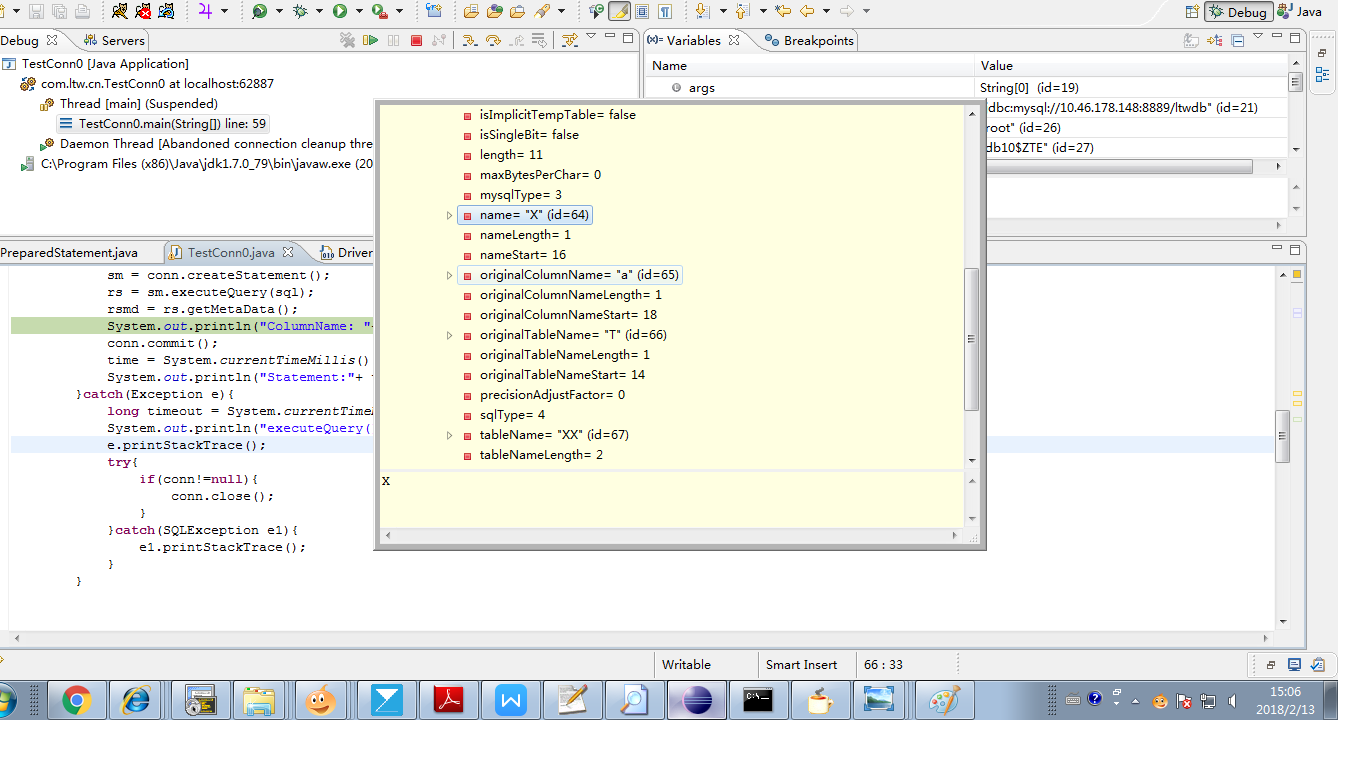
#### 验证流程二：

通过查阅jdbc代码，发现useOldAliasMetadataBehaviorjava作用在java层，分别针对0101和0102版本，查看语句执行后返回的java对象。

0101版本结果集里的内容：



0102版本结果集里的内容：

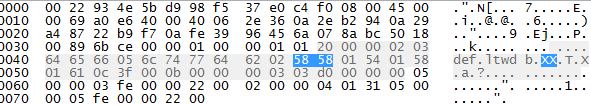
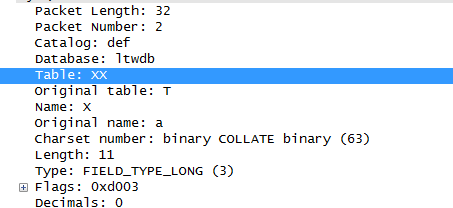


**流程二结论：比较发现，0101版本返回给客户端的数据包中丢失了原始列名、原始表名。**

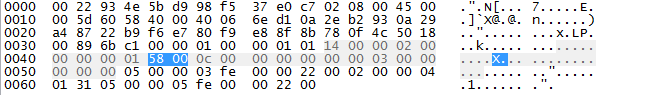
#### 验证流程三：

通过抓包查看0101、0102版本返回结果集的内容。

**0102版本码流头信息完整，所以JDBC能根据参数取出对应信息：**



**0101版本码流头信息不完整，只填了field name，其他信息都填的空，JDBC获取不到对应信息：**



**流程三结论：比较发现，0101版本返回给客户端的数据包码流不完整。**

### 单个DBProxy发生异常导致应用启动失败问题分析

#### 问题描述

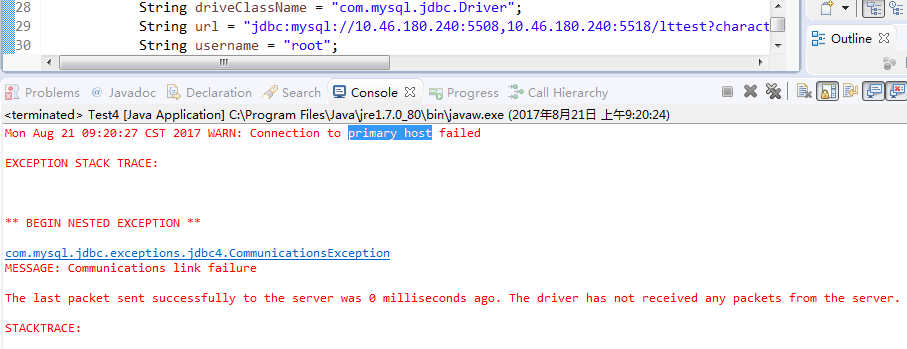
某业务进行物理机迁移后，运维人员对GoldenDB生产环境其中一台DBProxy和DB合设的服务器IP修改，导致同在该机器上的DBProxy和DB不可用，此时对应用进行重启，发现应用启动失败。对于未重启的应用，服务正常。

#### 问题分析

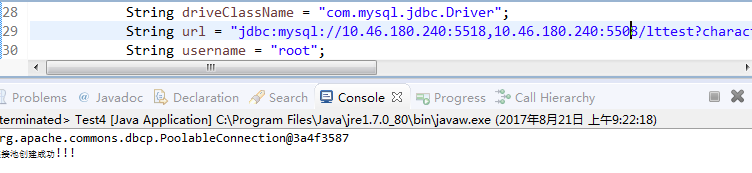
通过OMM监控所有组件运行状态，16上面的proxy和db状态为异常，17和18上面的DBProxy能够正常提供服务，同时数据库集群进行了主备切换，51替换16成为主机对外提供服务，整个集群能够正常提供服务。

通过对问题进行重现定位分析，后督系统中使用的数据库连接池为原系统自带连接池，业务修改仅做扩充，版本为1.2.1，此问题为开源版本存在的问题.

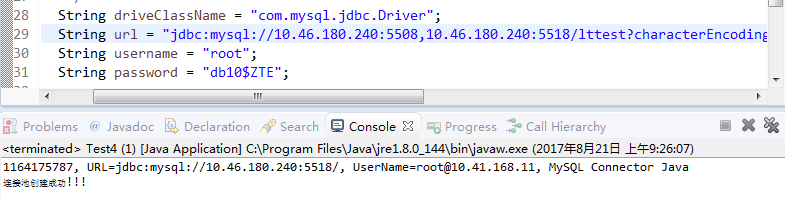
1.对于url中配置多个连接地址，dbcp会认为第一个地址为主连接（primary host），如果第一个连接失败，dbcp连接池就会创建失败：



2.如果第一个连接正常，则后面连接不管是不是正常，连接池均可以创建成功：



1. 对于该问题，替换为dbcp开源新版本中问题修复。替换dbcp2.x，再次验证该问题，连接池能够创建成功：



综上分析得出结论，故障是由于应用中的开源连接池DBCP的特性初始化时校验依赖的资源状态引发的，高版本的dbcp连接池已修复该问题。

#### 解决办法

将dbcp连接池升级至2.x版本问题解决。

### rewriteBatchedStatements和prepare使用注意事项

在2个参数打开时，如示例4中所示，jdbc会将多个预编译语句同时下发，如上，对于insert语法，jdbc在自动修改语法格式为insert ..values (??),(??)的格式，而对于delete和update语法无法像insert一样做合并时又会出现什么情况呢？

测试代码：

|  |
| --- |
| **public** **static** **void** batchUpdate() **throws** SQLException  {  **int** batchSize = 10;  PreparedStatement ps = *getConnection*().prepareStatement("update sbtest1 set k = k + ? where id = ?");    **for** (**int** i=0;i<batchSize;i++)  {  ps.setInt(1, i);  ps.setInt(2, i);  ps.addBatch();  }    ps.executeBatch();  } |

url配置：

|  |
| --- |
| String url = "jdbc:goldendb://10.46.178.242:5518/sbtest1?rewriteBatchedStatements=true&cachePrepStmts=true&useCursorFetch=true&prepStmtCacheSqlLimit=1000&prepStmtCacheSize=100"; |

码流结果：

|  |
| --- |
|  |

如上测试显示，jdbc先以预编译下发所有语句，而**mysql无法支持这种多个预编译语句同时下发的语法**，抛出一个异常“42000You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'update sbtest1 set k = k + ? where id = ?;update sbtest1 set k = k + ? where id ' at line 1”，而jdbc接到这个异常后并未直接将异常返回给客户端直接失败，而是将参数填回sql中，即将预编译模式还原回普通模式重新下发执行，对于客户端则无法感知这种错误。

在这种情况下，实际一次操作，往db发了2次消息，故会影响测试的性能，所以我们在测试prepare模式和使用rewriteBatchedStatements参数时，一定要注意其中的影响，目前看，insert和select不受影响，而update和delete则均会执行2次，一次失败，一次成功。

### 使用mybatis框架查询数据乱码

#### 问题描述

业务使用mybatis框架做集成开发，在现场发现通过proxy创建的数据表，插入有中文字符的数据，通过proxy查询出来会显示乱码，但直连db查询数据均显示正常。经现场确认，数据库字符集是utf8，表的字符集是utf8mb4，并且每一列都指定了utf8mb4字符集。

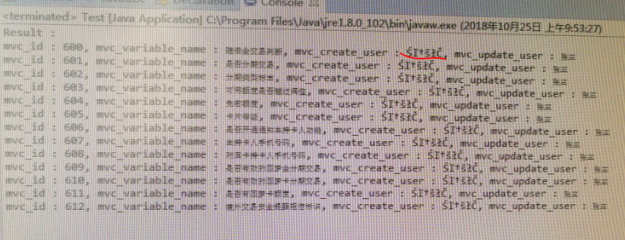
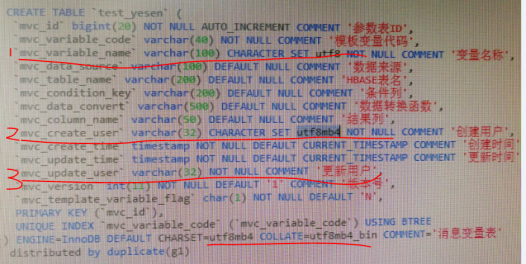
#### 问题分析

期初以为是proxy设置问题，后来发现proxy并不能设置字符集。后来创建一张列名相同的新表，插入数据后发现不再显示乱码，由于没有过多关注列指定的字符集，从而没能很快定位问题。

分来经分析整理，发现如果在库创建时指定utf8字符集后，创建表指定utf8mb4字符集，且列指定utf8mb4，则列数据查询时会显示乱码。保持库为utf8字符集，表为utf8mb4字符集，只修改列字符集为utf8，则查询正常。可以得出列字符集使用指定的utf8格式。如果不修改库字符集，也不修改表字符集，不指定列字符集，查询也是正常的，可以得出列使用的是库的utf8格式，而没有使用表的utf8mb4。由于出问题的库字符集是utf8mb4，查询出来的是乱码，所以没有进一步分析。

可以推测，字符集的应用：列 > 库 > 表。

后经进一步核实，mb4字符集从mysql-connector-java-5.1.13开始才支持，从这个方面下手进一步分析了现场使用的版本，从而确定问题是连接库版本导致的。



#### 解决方法

现场替换成5.1.41版本解决。

## JDBC 修改列表

### JDBC获取proxy建链消息中的proxy实际IP+port，并在duplicate新链路发送kill query时使用proxy的实际IP port

说明： 解决配套分布式数据库引入缺陷： JDBC通过F5等、LVS等负载均衡接入GoldenDB，在客户端尝试主动断开JDBC链接过程中，会通过另外的连接通道发送KILL QUERY请求给服务器，该请求经过F5等负载均衡设备时会被分发至其他计算节点(proxy)，导致原计算节点链接没有释放，而其他计算节点上的正常链接被释放了。

解决方法：JDBC在建链中获得计算节点实际IP+Port，发送KILL QUERY请求时直接连接该计算节点的IP+Port而不再经过负载均衡设备。

### JDBC通过参数配置支持服务端prepare对读写的支持

说明：这里有个深奥的概念：客户端prepare VS 服务端prepare。MySQL协议跟Oracle不同，先有jdbc标准中的prepare接口，后再有MySQL实现服务端prepare接口。这样带来的好处是：业务可以遵循标准的prepare接口进行编码，但实际通过客户端的组合将语句组装成普通语句下发给服务端DB，这样的内部实现称为客户端prepare。而服务端prepare，则是客户端使用的是prepare协议实际跟DB进行交互。

实际使用中，MySQL的服务端prepare只有正确使用结合缓存使用才有作用，否则性能下降严重而服务端消耗反而增加。

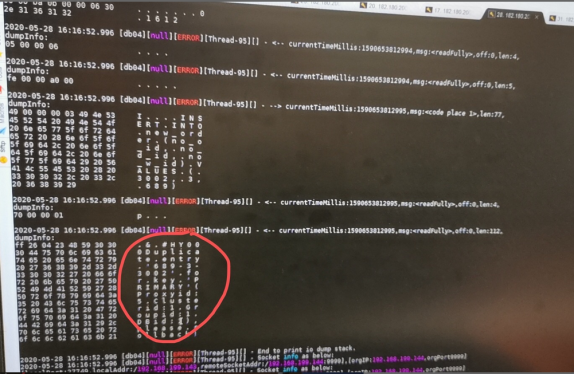
添加JDBC参数prepareStrategy：取值范围[1,2]。1：只开启读语句的服务端prepare；2：读写均开启服务端prepare

### 去除写语句执行前的readonly判断，悲观模式改成乐观模式

说明：去除业务通讯过程中无价值的写语句readonly判断，悲观模式改成乐观模式

### JDBC记录与proxy的通信码流栈，异常时输出码流栈信息，打印源端与本端真实的IP port，用于配合抓包分析通讯故障

说明：对通讯层的故障定位能力进行强化，当JDBC连接的socket层连接出现异常时，打印源和目标的IP端口，日志进行标准格式化。



### LoadBanlance协议支持连接级负载均衡,在URL中增加一个参数isConnectionLevel=true 同时使用LoadBanlance协议即可打开此功能

说明:原有Loadbalance协议中的负载均衡是在每次触发commit和rollback时，进行拦截并对Connection对象重新初始化，这样就会对客户端使用PreparedStatement产生限制，要求每次使用PreparedStatement之前必须重新从Connection中初始化，因为在触发拦截后，Connection对象已被替换成新的，所以之前PreparedStatement信息会丢失，无法和客户端的PreparedStatement对象匹配。

解决方法:  修改后在连接初始化时，按URL配置信息选出一个可用的Proxy，同时将此协议变换为jdbc原有协议而不使用Loadbalance，每次重新初始化连接时还会根据URL配置的Proxy，按照随机策略进行负载均衡选取。

使用说明: JDBC的URL中增加一个参数isConnectionLevel=true 同时复用LoadBanlance协议,即可打开此功能。如果为false或者不填则走原来的LoadBanlance协议逻辑

### JDBC增加连接自管理能力、增加黑名单管理能力

说明：针对2.5.5功能，增加“连接分组自管理及黑名单功能”，主要实现思路：在loadbalance+isConnectionLevle=true时，对每个proxy的IP+port建立管理组，同时增加一个影子连接，影子连接定期心跳，心跳异常则杀死全组的链接并加入黑名单，黑名单定时探测，链路正常后再踢出黑名单。

### jdbc常用参数默认值优化

说明：jdbc参数繁多，业务URL中经常要配置10+的参数，给使用人员带来了理解的挑战，同时增加使用和维护的难度。

参数修改：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数名 | 修改前 | 修改后 | 说明 |
| isConnectionLevel | false | true | 当建链协议检测到loadbalance协议和prepare（useCursorFetch或者useServerPstmts=true）同时被打开，则自动设置该参数为true |
| cachePrepStmts | false | true | 使用了服务端prepare时自动使用缓存 |
| prepStmtCacheSize | 25 | 2000 |  |
| prepStmtCacheSqlLimit | 256 | 2000 |  |
| useSSL | true | false |  |
| connectTimeout |  | 10000 |  |
| queryTimeoutKillsConnection | false | true |  |
| dumpInfoLength |  | 1000 |  |

### JDBC loadbalance支持将计算节点按分组优先级

说明：支持驱动loadbalance 配置多个组，在组间负载均衡

示例：

url="jdbc:goldendb:loadbalance://192.168.100.11:7788,192.168.100.12:7788/test?minconnectionproxys=1&proxygroups=2&proxygroup1=192.168.100.7:7788,192.168.100.8:7788&proxygroup2=192.168.100.9:7788,192.168.100.10:7788"

minconnectionproxys--组内正常可用proxy个数

proxygroups--分组个数，proxygroup1，proxygroup2......优先级依次降低

当高优先级proxy正常数量未达到minconnectionproxys时，则部分请求下发到下一个优先级proxy节点，下一个优先级所有的proxy参与负载。

当链接proxy最小数为0时，则只访问当前最大优先级proxy，当最大优先级proxy都异常时，则连接下一优先级proxy。

URL说明：proxygroups=0, JDBC执行原始loadbalance流程，IP+PORT范围是192.168.100.11:7788,192.168.100.12:7788; JDBC执行组优先级loadbalance流程，proxygroups!=0,IP+PORT范围是proxygroup1、proxygroup2配置值

### JDBC insert批量新协议

说明：支持insert批量新协议

1. 使用新协议批量插入insert数据，理论上比只使用rewriteBatchedStatements=true性能更好。
2. 可以提升批量插入语句性能。
3. 对update和delete无效

示例：

jdbc:goldendb://10.229.43.11:8880/tradingdb?comStmtBatchFlag=true

### JDBC支持设置proxy下发group

说明：支持建链时，设置proxy的下发group

示例：

jdbc:goldendb:loadbalance://10.229.31.134:5501,10.229.31.134:5502/test?dbgroup=g1

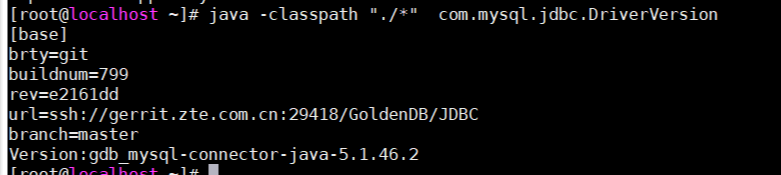
### JDBC驱动优化之支持版本识别

说明：JDBC版本查询功能，能够方便确定业务使用的驱动信息。

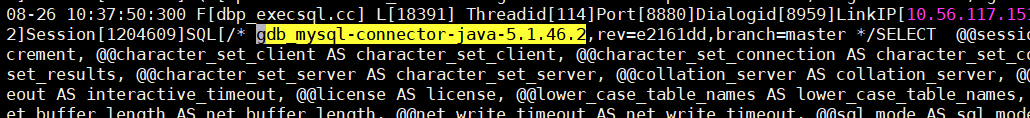
1. 命令行查看

1)进去jar包所在目录

2)执行java -classpath "./\*" com.goldendb.jdbc.DriverVersion



2.proxy日志查询



### 某些场景下游标失效

计算节点侧Prepare报错，驱动会将prepare转为普通语句，导致游标失效，数据库返回所有结果（数据量较大）给应用，导致应用OOM。

优化方法：增加参数，convertClientPstmts=FALSE时，若prepare报错后，驱动直接抛出异常信息。

### 优化读buffer内存占用

驱动里读取数据的buffer初始化50K，在读取中逐渐增大，比如数据读取完后该buffer增大到900K，驱动会判断buffer超过1M后才进行重新初始化为50K，导致驱动占用应用服务器内存过大。

优化方法：增加参数reclaimBufferLength，可以设置buffer初始化的阈值。

### 优化连接回收机制

JDBC链路垃圾回收机制是将已关闭的连接放到一个队列里去，然后由专门的线程负责释放链路对象，这导致链路对象虽然被关闭，但可能还存在内存中; 但这种处理机制有一个好处，能够保证链路不论是否正常关闭，都不会出现资源泄露。

优化方法：trackConnection=FALSE时，则不会将已关闭链路加入消息队列中。

### 栈信息打印IP+PORT

1)当出现链路异常后，在抛出的jdbc4.CommunicationsException中，增加链路信息打印：

com.goldendb.jdbc.exceptions.jdbc4.CommunicationsException: LinkInfo:[client]=28.5.4.80:50710 ==> [target]=28.4.199.180:5160

Communications link failure

2)当主动调用ConnectionImpl类的commit()方法出现异常时，在抛出的exception中增加链路信息打印：

com.goldendb.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException: LinkInfo:[client]=28.5.4.80:50429 ==> [target]=28.4.199.180:5160

Communications link failure during commit(). Transaction resolution unknown.

3)当主动调用ConnectionImpl类的rollback()方法出现异常时，在抛出的exception中增加链路信息打印：

com.goldendb.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException: LinkInfo:[client]=28.5.4.80:50746 ==> [target]=28.4.199.180:5160

Communications link failure during rollback(). Transaction resolution unknown.

### CVE-2021-2471漏洞修复

漏洞描述链接 <https://pyn3rd.github.io/2021/10/22/mysql-jdbc-xxe/>

### 自定义lodbalance报错提示优化

自定义lodbalance与原生lodbalance异常处理保持一致，将最后一次异常建链信息打印在栈信息。

### CVE-2020-2875漏洞修复

allowLoadLocalInfile默认改为false

### 打印建链相关日志

日志相关信息

1.建立socket链路

2.mysql握手交互

3.获取服务器相关配置

4.设置字符集

JDBC 日志输出到文件配置方法

1.添加log4j依赖包，log4j-\*.jar,slf4j-api-\*.jar,slf4j-log4j\*.jar

2.添加log4j.properties配置文件

#example



3.JDBC url加上参数logger=com.goldendb.jdbc.log.Slf4JLogger

### 原生loadbalance修改

修复库名为空，原生loadbalance无法继续使用。

### 高版本mybatis无法关闭prepare

原生JDBC的closeOnCompletion流程，不会关闭后端prepare

优化方法:增加参数closeStmtOnCompletionAction，closeStmtOnCompletionAction=true时，流程结束时，关闭后端prepare

### 批量功能优化

支持update、delete批量新协议

### 单分片备机只读查询

说明：增加参数readFromSlave，readFromSlave=true，设置链路属性只读，服务端选择一个备机下发所有sql，只对proxy单分片模式有

### 支持rowid相关功能

1.setRowId、getRowId、ROWID类等功能实现

2.getGeneratedKeys支持返回RowId

3.业务引用java.sql.RowId或者com.goldendb.jdbc.ROWID; JDK1.6以上才支持java.sql.RowId

## JDBC 保活说明

保活sql对应不同集群场景下，保活的网元如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 保活sql | 多分片 | 单分片 | 多分片指定group |
| /\* ping \*/ | OS | OS | OS |
| SELECT 1、SELECT @@VERSION | OS->CN | OS->CN->DN | OS->CN->DN |
| SELECT 1 FROM DUAL | OS->CN->DN(随机下发DN) | OS->CN->DN | OS->CN->DN |
| SELECT 1 FROM DUAL KEEPALIVE | OS->CN->DN(轮询下发DN) | OS->CN->DN | OS->CN->DN |

说明：

1）/\*ping\*/只能到OS层；SELECT 1、SELECT @@VERSION对于多分片只能保活到CN，其他场景能保活到DN。

2）JDBC中用select 1保活，此功能只是为了保证CN多分片集群的高可用，不需要考虑DN是否可用。

3）正常业务需要保证DN可用，保活语句建议用SELECT 1 FROM DUAL。

4）SELECT 1 FROM DUAL KEEPALIVE,适合多分片使用，轮流保活后端DN。