云数据库 GaussDB

最佳实践

文档版本 01

发布日期 2025-08-20

华为云计算技术有限公司

版权所有 © 华为云计算技术有限公司 2025。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

HUAWEI和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 GaussDB 安全配置建议.....1
2 扩缩容最佳实践.....5
3 备份恢复最佳实践.....7
3.1 备份恢复概述.....7
3.2 实例恢复.....12
3.2.1 实例恢复：从回收站恢复.....12
3.2.2 实例恢复：按备份文件恢复.....18
3.3 库表恢复.....24
3.3.1 库表恢复：恢复到指定时间点.....24
3.3.2 库表恢复：按备份文件恢复.....28
4 行存压缩最佳实践.....32
4.1 场景概述.....32
4.2 手动调度.....32
4.3 自动调度.....35
5 SQL 查询最佳实践.....38
5.1 SQL 查询最佳实践（分布式）.....38
5.2 SQL 查询最佳实践（集中式）.....40
6 权限配置最佳实践.....42
6.1 权限配置最佳实践（分布式）.....42
6.2 权限配置最佳实践（集中式）.....47
7 数据倾斜查询最佳实践（分布式）.....53
7.1 快速定位查询存储倾斜的表.....53
8 存储过程最佳实践.....55
8.1 存储过程最佳实践（分布式）.....55
8.1.1 权限控制.....55
8.1.2 命名规范.....56
8.1.3 访问对象.....58
8.1.4 语句功能.....59
8.1.4.1 PACKAGE 变量.....59
8.1.4.2 CURSOR.....61

8.1.4.3 兼容性功能 61
8.1.4.4 异常处理 64
8.1.4.5 自定义类型 64
8.1.5 事务管理 66
8.1.5.1 事务 66
8.1.5.2 自治事务 67
8.1.6 其他 68
8.1.6.1 DDL 68
8.1.6.2 复杂依赖 68
8.1.6.3 IMMUTABLE 和 SHIPPABLE 69
8.2 存储过程最佳实践（集中式） 70
8.2.1 权限控制 70
8.2.2 命名规范 72
8.2.3 访问对象 74
8.2.4 语句功能 75
8.2.4.1 PACKAGE 变量 75
8.2.4.2 CURSOR 76
8.2.4.3 兼容性功能 76
8.2.4.4 异常处理 78
8.2.5 事务管理 79
8.2.5.1 事务 79
8.2.5.2 自治事务 80
8.2.6 其他 81
8.2.6.1 DDL 81
8.2.6.2 复杂依赖 81
8.2.6.3 IMMUTABLE 82
9 COPY 导入导出最佳实践 84
9.1 COPY 导入导出最佳实践（分布式） 84
9.1.1 典型场景 84
9.1.1.1 推荐使用 CSV 格式 84
9.1.1.2 极致性能的导入导出场景 86
9.1.1.3 需自行解析数据文件的导出场景 87
9.1.1.4 仅支持 TEXT 格式的导入导出场景 89
9.1.1.5 导入导出的数据文件在 GSQL 客户端的场景 92
9.1.1.6 基于 JDBC 驱动导入导出数据的场景 92
9.1.1.7 数据存在错误时需要支持容错的导入场景 92
9.1.2 数据存在错误时的导出操作指南 93
9.1.3 数据存在错误时的导入操作指南 95
9.2 COPY 导入导出最佳实践（集中式） 97
9.2.1 典型场景 97
9.2.1.1 推荐使用 CSV 格式 98
9.2.1.2 极致性能的导入导出场景 99

9.2.1.3 需自行解析数据文件的导出场景.....101
9.2.1.4 仅支持 TEXT 格式的导入导出场景.....102
9.2.1.5 导入导出的数据文件在 GSQL 客户端的场景.....105
9.2.1.6 基于 JDBC 驱动导入导出数据的场景.....106
9.2.1.7 数据存在错误时需要支持容错的导入场景.....106
9.2.2 数据存在错误时的导出操作指南.....106
9.2.3 数据存在错误时的导入操作指南.....108
10 工具导入导出最佳实践.....110
10.1 工具导入导出最佳实践（分布式）.....110
10.1.1 数据库级导入导出.....110
10.1.2 模式级导入导出.....112
10.1.3 表级导入导出.....113
10.2 工具导入导出最佳实践（集中式）.....113
10.2.1 数据库级导入导出.....114
10.2.2 模式级导入导出.....115
10.2.3 表级导入导出.....116
11 JDBC 最佳实践.....120
11.1 JDBC 最佳实践（分布式）.....120
11.1.1 批量插入.....120
11.1.1.1 场景概述.....120
11.1.1.1.1 应用场景.....120
11.1.1.1.2 需求目标.....121
11.1.1.2 架构原理.....122
11.1.1.3 前置准备.....122
11.1.1.4 操作步骤.....122
11.1.1.4.1 流程总览.....122
11.1.1.4.2 具体步骤.....123
11.1.1.4.3 完整示例.....124
11.1.1.5 常见问题.....125
11.1.2 流式查询.....125
11.1.2.1 场景概述.....126
11.1.2.1.1 应用场景.....126
11.1.2.1.2 需求目标.....126
11.1.2.2 架构原理.....127
11.1.2.3 前置准备.....128
11.1.2.4 操作步骤.....128
11.1.2.4.1 流程总览.....128
11.1.2.4.2 具体步骤.....128
11.1.2.4.3 完整示例.....129
11.1.2.5 常见问题.....130
11.1.3 自定义类型.....130
11.1.3.1 场景概述.....130

11.1.3.1.1 应用场景 130
11.1.3.1.2 需求目标 131
11.1.3.2 架构原理 131
11.1.3.3 前置准备 131
11.1.3.4 操作步骤 132
11.1.3.4.1 流程总览 132
11.1.3.4.2 具体步骤 133
11.1.3.4.3 完整示例 135
11.1.3.5 常见问题 136
11.1.4 批量查询 136
11.1.4.1 场景概述 136
11.1.4.1.1 应用场景 136
11.1.4.1.2 需求目标 137
11.1.4.2 架构原理 137
11.1.4.3 前置准备 137
11.1.4.4 操作步骤 138
11.1.4.4.1 流程总览 138
11.1.4.4.2 具体步骤 139
11.1.4.4.3 完整示例 140
11.1.4.5 常见问题 141
11.2 JDBC 最佳实践（集中式） 141
11.2.1 批量插入 141
11.2.1.1 场景概述 141
11.2.1.1.1 应用场景 141
11.2.1.1.2 需求目标 143
11.2.1.2 架构原理 143
11.2.1.3 前置准备 143
11.2.1.4 操作步骤 143
11.2.1.4.1 流程总览 143
11.2.1.4.2 具体步骤 144
11.2.1.4.3 完整示例 145
11.2.1.5 常见问题 146
11.2.2 流式查询 146
11.2.2.1 场景概述 146
11.2.2.1.1 应用场景 147
11.2.2.1.2 需求目标 147
11.2.2.2 架构原理 148
11.2.2.3 前置准备 149
11.2.2.4 操作步骤 149
11.2.2.4.1 流程总览 149
11.2.2.4.2 具体步骤 149
11.2.2.4.3 完整示例 150

11.2.2.5 常见问题 151
11.2.3 自定义类型 151
11.2.3.1 场景概述 151
11.2.3.1.1 应用场景 151
11.2.3.1.2 需求目标 152
11.2.3.2 架构原理 152
11.2.3.3 前置准备 152
11.2.3.4 操作步骤 153
11.2.3.4.1 流程总览 153
11.2.3.4.2 具体步骤 154
11.2.3.4.3 完整示例 156
11.2.3.5 常见问题 157
11.2.4 批量查询 157
11.2.4.1 场景概述 157
11.2.4.1.1 应用场景 157
11.2.4.1.2 需求目标 158
11.2.4.2 架构原理 158
11.2.4.3 前置准备 158
11.2.4.4 操作步骤 159
11.2.4.4.1 流程总览 159
11.2.4.4.2 具体步骤 160
11.2.4.4.3 完整示例 161
11.2.4.5 常见问题 162
12 ODBC 最佳实践 163
12.1 ODBC 最佳实践（分布式） 163
12.1.1 场景概述 163
12.1.1.1 应用场景 163
12.1.1.2 需求目标 164
12.1.2 架构原理 164
12.1.3 前置准备 165
12.1.4 操作步骤 166
12.1.4.1 流程总览 166
12.1.4.2 具体步骤 166
12.1.4.3 完整示例 168
12.1.5 常见问题 171
12.2 ODBC 最佳实践（集中式） 172
12.2.1 场景概述 172
12.2.1.1 应用场景 172
12.2.1.2 需求目标 173
12.2.2 架构原理 173
12.2.3 前置准备 174
12.2.4 操作步骤 175

12.2.4.1 流程总览 175
12.2.4.2 具体步骤 175
12.2.4.3 完整示例 177
12.2.5 常见问题 180
13 GO 最佳实践 181
13.1 GO 最佳实践（分布式） 181
13.1.1 场景概述 181
13.1.1.1 应用场景 181
13.1.1.2 需求目标 182
13.1.2 架构原理 182
13.1.3 前置准备 183
13.1.4 操作步骤 183
13.1.4.1 流程总览 183
13.1.4.2 具体步骤 184
13.1.4.3 完整示例 188
13.1.5 常见问题 191
13.2 GO 最佳实践（集中式） 191
13.2.1 场景概述 191
13.2.1.1 应用场景 192
13.2.1.2 需求目标 193
13.2.2 架构原理 193
13.2.3 前置准备 194
13.2.4 操作步骤 194
13.2.4.1 流程总览 194
13.2.4.2 具体步骤 195
13.2.4.3 完整示例 199
13.2.5 常见问题 202
14 索引设计最佳实践 203
14.1 索引设计最佳实践（分布式） 203
14.1.1 场景概述 203
14.1.2 前置准备 203
14.1.3 操作步骤 204
14.2 索引设计最佳实践（集中式） 205
14.2.1 场景概述 205
14.2.2 前置准备 206
14.2.3 操作步骤 206
15 表设计最佳实践 209
15.1 表设计最佳实践（分布式） 209
15.1.1 场景概述 209
15.1.2 架构原理 209
15.1.3 前置准备 210

15.1.4 操作步骤 210
15.2 表设计最佳实践（集中式） 215
15.2.1 场景概述 215
15.2.2 架构原理 215
15.2.3 前置准备 216
15.2.4 操作步骤 216

1 GaussDB 安全配置建议

安全性是华为云与您的共同责任。华为云负责云服务自身的安全，提供安全的云。作为租户，您需要合理使用云服务提供的安全能力对数据进行保护，安全地使用云。详情请参见责任共担。

本文提供了GaussDB使用过程中的安全配置建议，旨在为提高整体安全能力提供可操作的规范性指导。根据该指导文档您可以持续评估GaussDB的安全状态，更好的组合使用GaussDB提供的多种安全能力，提高对GaussDB的整体安全防御能力，保护存储在GaussDB的数据不泄露、不被篡改，以及数据传输过程中不泄露、不被篡改。

本文从以下几个维度给出建议，您可以评估GaussDB的使用情况，并根据业务需要在本指导的基础上进行安全配置。

- 最大连接数配置
- 安全认证配置
- 用户密码的安全策略
- 权限管理
- 数据库审计
- WAL 归档配置
- 备份管理

最大连接数配置

如果GaussDB连接数过高，会消耗服务器大量资源，导致操作响应变慢，可以修改max_connections参数进行优化，具体内容请参见连接设置。

max_connections：允许和数据库连接的最大并发连接数，此参数会影响集群的并发能力。

安全认证配置

为了保证用户体验，同时为了防止账户被人通过暴力破解，GaussDB设置了账户登录重试次数及失败后自动解锁时间的保护措施，GaussDB针对账户提供了以下能力：

- failed_login_attempts：允许用户设置最大登录失败次数。
- password_lock_time：此参数允许用户修改账户被锁定后自动解锁时间，单位为天。

版权所有 © 华为云计算技术有限公司

若管理员发现某账户被盗、非法访问等异常情况，可手动锁定该账户。当管理员认为账户恢复正常后，可手动解锁该账户。

以手动锁定和解锁用户joe为例，命令格式如下：

– 手动锁定

```
gaussdb=# ALTER USER joe ACCOUNT LOCK;
ALTER ROLE
```

– 手动解锁

```
gaussdb=# ALTER USER joe ACCOUNT UNLOCK;
ALTER ROLE
```

用户密码的安全策略

GaussDB为了客户账号的安全，GaussDB对用户密码进行了以下设置：

- 用户密码存储在系统表pg_authid中，为防止用户密码泄露，GaussDB对用户密码进行加密存储，所采用的加密算法由配置参数password_encryption_type决定。
- GaussDB数据库用户的密码都有密码有效期，如果需要修改密码有效期，可以通过修改**password_effect_time**来更改。

权限管理

- 虚拟私有云可以为GaussDB实例构建隔离的、用户自主配置和管理的虚拟网络环境。子网提供与其他网络隔离的、可以独享的网络资源，以提高网络安全性，可以使用IAM为企业中的员工设置不同的访问权限，以达到不同员工之间的权限隔离，通过IAM进行精细的权限管理。具体内容请参见**权限管理**。
- 保障数据库的安全性和稳定性在使用数据库实例之前务必先设置安全组，具体内容请参见**设置安全组规则**。
- 为防止PUBLIC拥有CREATE权限，导致数据库任何账户都可以在PUBLIC模式下创建表或者其他数据库对象，其他用户也可以修改这些数据，可以如下SQL语句来查询：

  SELECT CAST(has_schema_privilege('public','public','CREATE') AS TEXT);

  – 如果返回为TRUE，执行如下SQL语句进行修复：

    REVOKE CREATE ON SCHEMA public FROM PUBLIC;

- PUBLIC角色属于任何用户，如果将对象的所有权限授予PUBLIC角色，则任意用户都会继承此对象的所有权限，违背权限最小化原则，为了保障数据库数据的安全，此角色应该拥有尽可能少的权限。通过执行如下SQL语句来确定所有权限是否授权PUBLIC角色：

  SELECT relname,relacl FROM pg_class WHERE (CAST(relacl AS TEXT) LIKE '%,=arwdDxt/%}' OR CAST(relacl AS TEXT) LIKE '{=arwdDxt/%}') AND (CAST(relacl AS TEXT) LIKE '%,=APmiv/%}' OR CAST(relacl AS TEXT) LIKE '{=APmiv/%}');

  – 为空则说明已授权，如果已授权，可通过执行如下SQL语句来修复：

    REVOKE ALL ON <OBJECT_NAME> FROM PUBLIC;

- pg_catalog模式下的pg_authid系统表中包含了数据库中所有的角色信息。由于所有用户会继承PUBLIC角色的权限，为了防止敏感信息泄露或被更改，PUBLIC角色不允许拥有pg_authid系统表的任何权限，执行如下SQL语句，如果查询结果显示不为空，则已经被授权：

  SELECT relname,relacl FROM pg_class WHERE relname = 'pg_authid' AND CAST(relacl AS TEXT) LIKE '%,=%}';

版权所有 © 华为云计算技术有限公司

  - 如果已授权，通过执行如下SQL语句进行修复：

    REVOKE ALL ON pg_authid FROM PUBLIC;

- 普通用户指用于执行普通业务操作的非管理员用户。作为普通用户，不应该拥有超出其正常权限范围的管理权限，例如创建角色权限，创建数据库权限，审计权限，监控权限，运维权限，安全策略权限等，在满足正常业务需求的前提下，为了确保普通用户权限最小化，应撤销普通用户非必须的管理权限。
- 在创建函数时声明SECURITY DEFINER表示函数以创建它的用户权限执行，如果使用不当会导致函数执行者借助创建者的权限执行越权操作，所以一定确保这样的函数不被滥用。为了安全考虑，禁止PUBLIC角色执行SECURITY DEFINER类型的函数，执行如下SQL语句查询PUBLIC角色是否有SECURITY DEFINER类型的函数：

  SELECT a.proname, b.nspname FROM pg_proc a, pg_namespace b where a.pronamespace=b.oid and b.nspname <> 'pg_catalog' and a.prosecdef='t';

  - 如果返回非空，执行如下SQL语句检查是否有执行权限：

    SELECT CAST(has_function_privilege('public', 'function_name([arg_type][, ...])', 'EXECUTE') AS TEXT);

    - 返回TRUE，则代表拥有，执行下面的SQL语句进行修复：

      REVOKE EXECUTE ON FUNCTION function_name([arg_type][, ...]) FROM PUBLIC;

- SECURITY INVOKER函数是以调用它的用户的权限来执行，使用不当会导致函数创建者借助执行者的权限执行越权操作，所以在调用非自身创建的这类函数时，一定要先检查函数执行内容，避免造成函数创建者借助执行者的权限执行了越权的操作。

数据库审计

- GaussDB可以记录实例相关的操作，但是仅针对支持的审计操作，请在操作前查询操作列表，具体内容请参见**支持审计的关键操作列表**。
- 确保配置开启数据库对象的添加、删除、修改审计，具体内容请参见**数据库审计**。
- 支持审计日志可视化查看，可开启LTS的能力，具体内容可参见**LTS日志**。

WAL 归档配置

WAL(Write Ahead Log)即预写式日志，也称为Xlog。wal_level决定了写入WAL的信息量。为了在备机上开启只读查询，wal_level需要在主机上设置成hot_standby，并且备机设置hot_standby参数为on。

备份管理

GaussDB支持数据库实例的备份和恢复，以保证数据可靠性。备份目前将以未加密的方式存储，防止客户误操作或者服务异常的情况下，因没有开启备份而造成数据丢失的情况，GaussDB针对备份提供了以下能力：

- 提供了自动和手动的备份功能，具体内容请参见**备份概述**，在创建GaussDB实例时，系统默认开启实例级自动备份策略。实例创建成功后，您可根据业务需要修改实例级自动备份策略。
- 提供了自动备份策略，定时定期对数据库进行备份。具体内容请参见**设置自动备份策略**。

- 提供了导出备份文件的能力，具体内容请参见**导出备份信息**。

版权所有 © 华为云计算技术有限公司

2 扩缩容最佳实践

云数据库GaussDB提供了多种功能，以满足您的扩缩容需求，包括变更CPU和内存规格、调整存储空间、以及增减节点和分片数量，您可以根据业务需求轻松调整数据库的性能和容量。

变更 CPU 和内存规格扩缩容

当实例的性能不满足业务需求，或者新业务的数据量较少时，您可以通过变更实例CPU和内存规格的方式进行扩容或缩容。具体操作参见**变更GaussDB实例的CPU和内存规格**。

修改存储空间大小扩容

当出现以下场景时，建议您选择修改存储空间大小进行扩容：

- 随着GaussDB实例使用时间的增长，业务数据量攀升，原有的磁盘空间可能无法满足存储需求。此时，您可以通过磁盘扩容功能来增加数据库实例的存储容量。
- 当磁盘使用率超过阈值（默认为85%，可以通过修改实例参数"cms:datastorage_threshold_value_check"进行配置）时，GaussDB实例会被设置为只读状态，无法再写入数据。您可以通过磁盘扩容避免这种情况，确保业务的连续性。

具体操作参见**扩容磁盘**。

增删协调节点扩缩容

当业务并发量显著增加，原有协调节点（CN）的处理能力无法满足需求时，可以通过增加协调节点数量来提升并发处理能力。具体操作参见**扩容实例协调节点**。

相反，随着业务下降，数据库协调节点利用率低，资源浪费严重。此时，可以通过减少协调节点提高资源利用率。具体操作参见**缩容实例协调节点**。

增删协调节点扩缩容仅分布式独立部署形态支持。

增删分片扩缩容

随着业务数据量的持续增长，原有数据节点（DN）可能无法承载更多的数据，此时，可以通过增加分片数量来分散数据。具体操作参见**扩容实例分片**。

版权所有 © 华为云计算技术有限公司

反之，实例进行读写分离或者业务冗余数据清理等操作后数据节点使用率会下降。此时，可通过减少分片进行缩容避免成本浪费。具体操作参见**缩容实例分片**。

增删分片数量扩缩容仅分布式独立部署形态支持。

3 备份恢复最佳实践

3.1 备份恢复概述

当数据库或表被恶意或误删除时，虽然GaussDB支持高可用，但备机数据库会被同步删除且无法还原。因此，数据被删除后只能依赖于实例的备份保障数据安全。GaussDB支持通过备份文件将数据恢复到备份被创建时的状态或指定时间点的状态，从而保证数据的可靠性。

本章节介绍了常见误操作的场景和恢复方案，具体可参考**表3-1**。同时，还介绍了备份恢复策略的典型场景与性能规格，相关内容见**表2 备份恢复策略典型场景**。您可以根据实际情况，选择对应的数据恢复方式。

误操作恢复方案

表 3-1 误操作恢复方案

场景	恢复方案	恢复数据范围	操作指导
误删实例	进入回收站，如果在回收站找到已删除的实例，可以通过重建操作来恢复实例。	所有库表	实例恢复：从回收站恢复
	如果在删除实例之前进行过手动备份，可以在“备份恢复”页面进行恢复。	所有库表	实例恢复：按备份文件恢复
误删表	可以采用库表恢复的方式恢复误删的表。	• 所有库表 • 部分库表	• 库表恢复：恢复到指定时间点 • 库表恢复：按备份文件恢复
误删数据库	可以采用库表恢复的方式恢复误删的数据库。	• 所有库表 • 部分库表	

版权所有 © 华为云计算技术有限公司

场景	恢复方案	恢复数据范围	操作指导
误操作表中数据，如整体覆盖、误删除/修改表中的列/行/数据	采用库表恢复的方式恢复误操作表中的数据。	• 所有库表 • 部分库表	

版权所有 © 华为云计算技术有限公司

备份恢复策略的场景及性能规格

表 3-2 备份恢复策略的场景及性能规格

备份恢复策略	关键性能因素	典型数据量	性能规格
数据库实例备份	● 数据大小 ● 网络配置	数据：PB级 对象：约100万个	OBS备份恢复规格： 1. 在标准环境下全量备份恢复的性能规格为2T。数据在8小时以内完成全量备份或全量恢复。 2. 如果客户硬件条件较好、obs带宽足够、压缩比较大、独立部署的情况下，全量备份恢复的耗时可以采用如下方式计算： ● 分布式 备份恢复时长 = （数据库实例总数据量 / 分片个数）/ min(磁盘IO读带宽，压缩带宽，单线程obs传输带宽/压缩比)。 ● 集中式 备份恢复时长 = 数据库实例总数据量 / min(磁盘IO读带宽，压缩带宽，单线程obs传输带宽/压缩比)。 **说明** ● min()为取括号内的最小值。 ● 磁盘I/O读带宽：SATA SSD 在200~300MB/s左右，SAS SSD 在500MB/s左右，NVME SSD在1GB/s左右，注意需要预留出给数据库业务运行的I/O带宽，否则备份操作会对业务性能产生明显影响。 ● 压缩带宽：当前默认采用LZ4压缩方式，一般在300~400MB/s左右。压缩级别范围为1 ~ 9，当前默认压缩级别为1，当指定更高的压缩级别时，压缩速率会变慢，从而导致备份时间变长，具体时间跟数据特征相关，请以实测为准。 ● 单线程OBS传输带宽：非限速模式下，一般在100~300MB/s左右，限速模式下，即为限速值。 ● 压缩比：当前默认采用LZ4压缩方式，压缩等级低，压缩比一般在0.1~0.5之间。具体压缩比与数据特征相关，请以实测为准。 ● 并行上传参数设置为2及以上时，备份所使用的CPU等资源会上升，备份性能提升与OBS单流传输带宽占OBS总带宽的比值相关，当单流传输带宽乘并行上传参数大于总带宽后，性能将不再提升。 3. 分布式实例在恢复hashbucket表扩容重分布期间的备份集时，恢复时长（不包含恢复后重入的重分布流程）≤ 相同数据量非扩容期间备份集相同方式恢复时长 × 2 + 相同数据量hashbucket表重分布时长。 恢复hashbucket表扩容重分布期间的备份集分为三个阶段。 ● 第一阶段：恢复所有节点全量备份，恢复扩容前旧的数据节点的所有增量备份并回放日志。

版权所有 © 华为云计算技术有限公司

备份恢复策略	关键性能因素	典型数据量	性能规格
			● 第二阶段：将需要重分布的hashbucket文件从旧的数据节点物理搬迁至扩容出来的新数据节点。 ● 第三阶段：恢复扩容出来的新数据节点的所有增量备份并回放日志。 4. 分布式实例数据恢复后启动阶段的时长与库的sequence数量和库的数量有关。 ● 恢复启动阶段会获取每个库的sequence信息并设置到ETCD中。主要耗时在获取和设置两个阶段： – 连接每个库获取该库的sequence信息：库越多，耗时越久。 – 设置sequence到ETCD中：sequence越多，耗时越久。 ● 更新PGXC表信息：连接每个库更新pgxc_class和pgxc_slice表信息，库越多，耗时越久。

版权所有 © 华为云计算技术有限公司

备份恢复策略	关键性能因素	典型数据量	性能规格
库级物理恢复	● 数据大小 ● 网络配置	-	OBS库级物理恢复共分为4个阶段，性能规格如下： ● 第一阶段：从备份介质中读取库级恢复的全部数据。在公有云标准环境下的性能规格为：2TB。8小时内完成数据读取。 ● 第二阶段：在辅助数据库中对库级数据进行vacuum freeze，vacuum freeze的性能为： – 分布式：单分片1400GB/小时，分片间可并行复制。 – 集中式：1400GB/小时。 ● 第三阶段：将vacuum freeze后的库级数据复制到生产实例的各DN副本。复制性能为： – 分布式：单分片数据量 / min(网络带宽、磁盘I/O带宽)，分片间可并行复制。 – 集中式：库级恢复数据量/ min(网络带宽，磁盘I/O带宽)。 ● 第四阶段：将数据导入生产实例。导入性能为： – 分布式：单分片数据量 、磁盘I/O带宽，分片间可并行复制。 – 集中式：库级恢复数据量、磁盘I/O带宽。 推荐场景： ● 性能维度：相同数据量下，当前的库级物理恢复性能约为实例级恢复的70%，当恢复的库级数据总数据量小于实例级数据量的70%时，推荐使用库级物理恢复。 ● 可用性维度：库级恢复过程中目标实例的其他库可持续对外提供服务，与实例级恢复相比有可用性优势，如果希望库级恢复过程中仍能访问其他库，推荐使用库级物理恢复。 业务影响： ● 库级导入前需要关闭流控，将GUC参数recovery_time_target设置为0，库级恢复期间对生产环境的吞吐量有一定影响，吞吐量通常在峰值吞吐量的50%，最差在系统峰值的25%。 ● 建议在业务低峰期进行细粒度恢复。

版权所有 © 华为云计算技术有限公司

备份恢复策略	关键性能因素	典型数据量	性能规格
表级物理恢复	● 数据大小 ● 网络配置 ● 表存储类型 ● 表属性（列）类型	-	OBS表级物理恢复分为3个阶段，性能规格如下： ● 第一阶段：从备份介质中读取表级恢复的全部数据。在公有云标准环境下的性能规格为：2TB。数据8小时内读取完成。 ● 第二阶段：将辅助数据库中的表数据导出至本地文件，导出性能约25MB/s。 ● 第三阶段：将本地导出的文件导入到生产实例中，GUC参数page_version_check设置为off时，导入性能约25MB/s（page_version_check为memory性能下降约15%）。受表行数、表索引数量、触发器数量的影响，导入性能可劣化至约10MB/s。 推荐场景： ● 性能维度：相同数据量下，当前的表级物理恢复性能约为实例级恢复的1/5，当恢复的表级数据总数据量小于实例级数据量的1/5、且要恢复的表数据总量不超过1TB时，推荐使用表级物理恢复。 ● 可用性维度：表级恢复过程中，目标实例的其他库/表可持续对外提供服务，与实例级恢复相比有可用性优势，如果希望表级恢复过程中仍能访问其他库/表，推荐使用表级物理恢复。 业务影响： ● 表级恢复期间对生产环境的吞吐量有一定影响，吞吐量通常在峰值吞吐量的50%，最差在系统峰值的25%。 ● 建议在业务低峰期进行细粒度恢复。

3.2 实例恢复

3.2.1 实例恢复：从回收站恢复

操作场景

回收站保留了已删除实例在删除时生成的备份，如果备份未超期，您可以在回收站中通过重建实例恢复误删除的实例。

操作流程

操作步骤	说明
步骤1：构造数据	使用DAS新建数据库、新建表及插入数据。

版权所有 © 华为云计算技术有限公司

操作步骤	说明
步骤2：误删除实例	模拟误删除实例的操作。
步骤3：从回收站恢复实例	通过回收站恢复实例数据。
步骤4：确认恢复结果	登录DAS，确认数据是否恢复。

步骤 1：构造数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，创建数据库。

   ```
   CREATE DATABASE db_tpcds;
   ```

   当结果显示为如下信息，则表示创建成功。

   图 3-1 创建数据库

   创建完db_tpcds数据库后，可以在左上方切换到新创建的库中。
8. 执行如下SQL，新建表和插入数据。
   - 执行如下命令来创建一个schema。

     ```
     CREATE SCHEMA myschema;
     ```

     创建完schema后，可以在左上方切换到新创建的schema。

版权所有 © 华为云计算技术有限公司

- 创建一个名称为mytable，只有一列的表。字段名为firstcol，字段类型为integer。
  ```
  CREATE TABLE myschema.mytable (firstcol int);
  ```
- 向表中插入数据：
  ```
  INSERT INTO myschema.mytable values (100);
  ```

9. 查询表数据。
   ```
   SELECT * FROM myschema.mytable;
   ```

步骤 2：误删除实例

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面的实例列表中，选择需要删除的实例，在“操作”列，单击“更多 > 删除实例”。
5. 在“删除实例”弹框，输入“DELETE”，并勾选“已确认”，单击“确定”下发请求，稍后刷新“实例管理”页面，查看删除结果。

步骤 3：从回收站恢复实例

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在左侧导航栏，单击“回收站”。
5. 在“回收站”页面，在实例列表中找到需要恢复的目标实例，单击操作列的“重建”。
6. 在“重建新实例”页面，选择计费模式、填写实例名称、选择可用区和时区等实例基本信息。

图 3-2 计费模式和基本信息

表 3-3 参数说明

参数	示例	参数说明
计费模式	**按需计费**	云数据库GaussDB提供包年/包月、按需计费两种计费模式。 ● 包年/包月是一种预付费模式，即先付费再使用，按照订单的购买周期进行结算，因此在购买之前，必须确保账户余额充足。对于分布式混合部署形态的实例，包年/包月模式受限开放，如需使用，可以在管理控制台右上角，选择**新建工单**，提交开通申请。 ● 按需计费是一种后付费模式，即先使用再付费，即开即停，按照云数据库实际使用时长计费。以自然小时为单位整点计费，不足一小时按实际使用时长计费。
实例名称	**gauss-7e3d**	实例名称长度在4个到64个字符之间，必须以字母开头（区分大小写），可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。

版权所有 © 华为云计算技术有限公司

参数	示例	参数说明
切换策略	**数据高可靠**	仅支持分布式版独立部署实例。 该参数仅针对特定用户开放，如需使用您可以在管理控制台右上角，选择**新建工单**，提交申请。 ● 数据高可靠：对数据一致性要求高的系统推荐选择数据高可靠，在故障切换的时候优先保障数据一致性。 ● 业务高可用：对业务在线时间要求高的系统推荐使用业务高可用，在故障切换的时候优先保证数据库可用性。
可用区	**可用区一**	可用区指在同一区域下，电力、网络隔离的物理区域，可用区之间内网互通，不同可用区之间物理隔离。
时区	**(UTC+08:00) 北京，重庆，香港，乌鲁木齐**	由于世界各国家与地区经度不同，地方时也有所不同，因此会划分为不同的时区。

7. 选择实例规格。

图 3-3 规格与存储

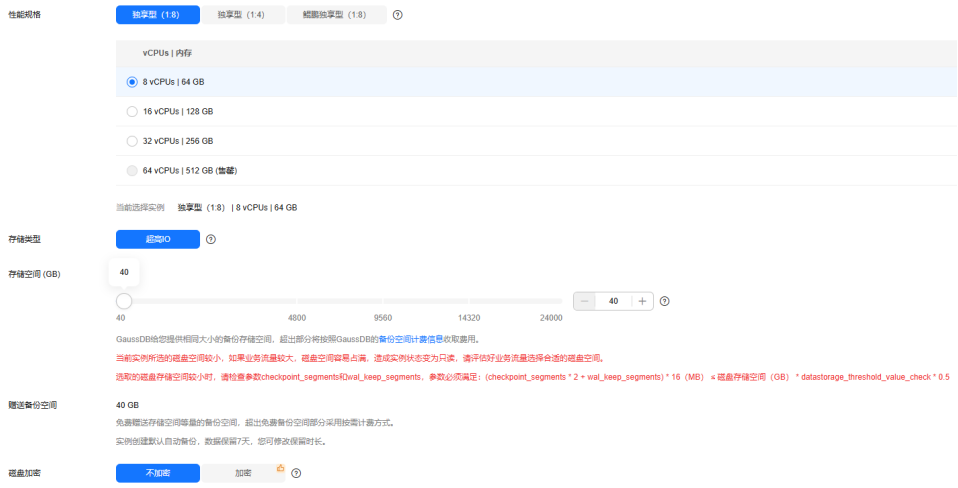


表 3-4 参数说明

参数	示例	参数说明
性能规格	**独享型（1:8）、8 vCPUs | 64 GB**	实例的CPU和内存规格。

版权所有 © 华为云计算技术有限公司

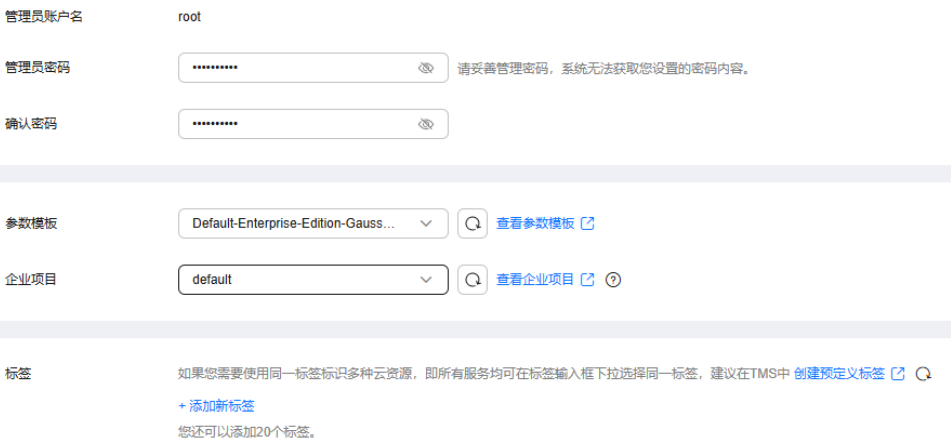
参数	示例	参数说明
存储空间	**40GB**	申请的存储空间会有必要的文件系统开销，这些开销包括索引节点和保留块，以及数据库运行必需的空间。
磁盘加密	**不加密**	选择加密后会提高数据安全性，但对数据库读写性能有少量影响，请按照您的使用策略进行选择。 如果使用共享KMS密钥，对应的CTS事件为createGrant，仅密钥所有者能够感知到该事件。

8. 配置网络信息，保持默认配置即可。

图 3-4 网络配置

9. 配置实例密码，选择企业项目信息。

图 3-5 数据库配置



版权所有 © 华为云计算技术有限公司

表 3-5 参数说明

参数	示例	参数说明
管理员密码	-	需要输入高强度密码并定期修改，以提高安全性，防止出现密码被暴力破解等安全风险。
确认密码	-	必须和管理员密码相同。
企业项目	**default**	对于已成功关联企业项目的用户，仅需在“企业项目”下拉框中选择目标项目。 如果需要自定义企业项目，请前往企业项目管理服务进行创建。关于如何创建项目，详见《企业项目管理用户指南》。

10. 单击“立即购买”，进入实例信息确认页面。
11. 确认信息无误后，单击“提交”，提交重建任务。
12. 在实例列表中查看实例运行状态，当实例运行状态为“正常”时，表示实例重建完成。

步骤 4：确认恢复结果

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 查看并确认目标库名和表数据，确认是否恢复完成。

3.2.2 实例恢复：按备份文件恢复

操作场景

如果您在删除实例之前进行过手动备份，可以在“备份恢复”页面进行恢复。

操作流程

操作步骤	说明
步骤1：构造数据	使用DAS新建数据库、新建表及插入数据。
步骤2：误删除实例	模拟误删除实例的操作。

版权所有 © 华为云计算技术有限公司

操作步骤	说明
步骤3：通过备份文件恢复实例	使用备份文件恢复实例数据。
步骤4：确认恢复结果	登录DAS，确认数据是否恢复。

步骤 1：构造数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，创建数据库。

   ```
   CREATE DATABASE db_tpcds;
   ```

   当结果显示为如下信息，则表示创建成功。

   图 3-6 创建数据库

   创建完db_tpcds数据库后，可以在左上方切换到新创建的库中。
8. 执行如下SQL，新建表和插入数据。
   - 执行如下命令来创建一个schema。

     ```
     CREATE SCHEMA myschema;
     ```

     创建完schema后，可以在左上方切换到新创建的schema。

版权所有 © 华为云计算技术有限公司

- 创建一个名称为mytable，只有一列的表。字段名为firstcol，字段类型为integer。
  ```
  CREATE TABLE myschema.mytable (firstcol int);
  ```
- 向表中插入数据：
  ```
  INSERT INTO myschema.mytable values (100);
  ```

9. 查询表数据。
   ```
   SELECT * FROM myschema.mytable;
   ```

步骤 2：误删除实例

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面的实例列表中，选择需要删除的实例，在“操作”列，单击“更多 > 删除实例”。
5. 在“删除实例”弹框，输入“DELETE”，并勾选“已确认”，单击“确定”下发请求，稍后刷新“实例管理”页面，查看删除结果。

步骤 3：通过备份文件恢复实例

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在左侧导航栏单击“备份恢复”，选择需要恢复的备份，单击操作列的“恢复”。
5. 选择恢复到“新实例”，单击“确定”，恢复实例。

图 3-7 恢复备份

版权所有 © 华为云计算技术有限公司

6. 在“恢复到新实例”页面，选择计费模式、填写实例名称、选择可用区和时区等实例基本信息。

图 3-8 计费模式和基本信息

表 3-6 参数说明

参数	示例	参数说明
计费模式	**按需计费**	云数据库GaussDB提供包年/包月、按需计费两种计费模式。 ● 包年/包月是一种预付费模式，即先付费再使用，按照订单的购买周期进行结算，因此在购买之前，必须确保账户余额充足。对于分布式混合部署形态的实例，包年/包月模式受限开放，如需使用，可以在管理控制台右上角，选择新建工单，提交开通申请。 ● 按需计费是一种后付费模式，即先使用再付费，即开即停，按照云数据库实际使用时长计费。以自然小时为单位整点计费，不足一小时按实际使用时长计费。
实例名称	**gauss-7e3d**	实例名称长度在4个到64个字符之间，必须以字母开头（区分大小写），可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。

版权所有 © 华为云计算技术有限公司

参数	示例	参数说明
切换策略	**数据高可靠**	仅支持分布式版独立部署实例。 该参数仅针对特定用户开放，如需使用您可以在管理控制台右上角，选择**新建工单**，提交申请。 ● 数据高可靠：对数据一致性要求高的系统推荐选择数据高可靠，在故障切换的时候优先保障数据一致性。 ● 业务高可用：对业务在线时间要求高的系统推荐使用业务高可用，在故障切换的时候优先保证数据库可用性。
可用区	**可用区一**	可用区指在同一区域下，电力、网络隔离的物理区域，可用区之间内网互通，不同可用区之间物理隔离。
时区	**(UTC+08:00) 北京，重庆，香港，乌鲁木齐**	由于世界各国家与地区经度不同，地方时也有所不同，因此会划分为不同的时区。

7. 选择实例规格。

图 3-9 规格与存储

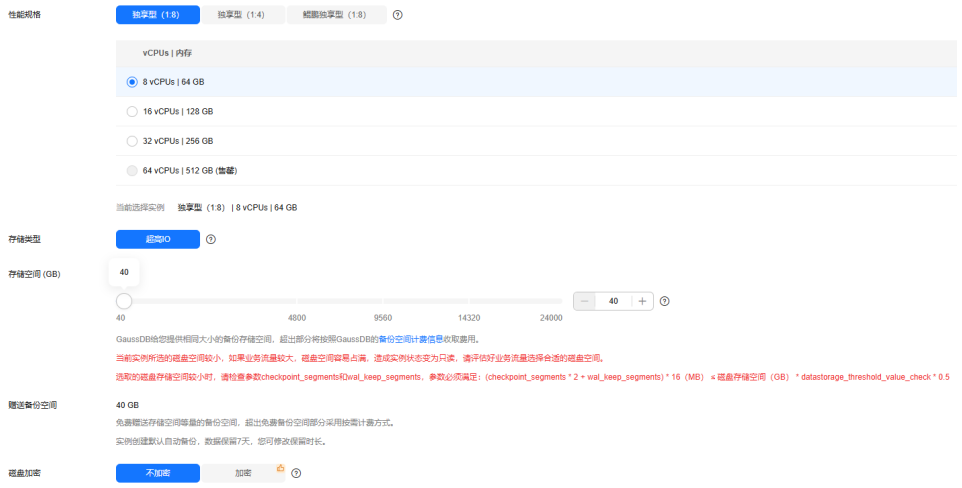


表 3-7 参数说明

参数	示例	参数说明
性能规格	**独享型（1:8）、8 vCPUs | 64 GB**	实例的CPU和内存规格。

版权所有 © 华为云计算技术有限公司

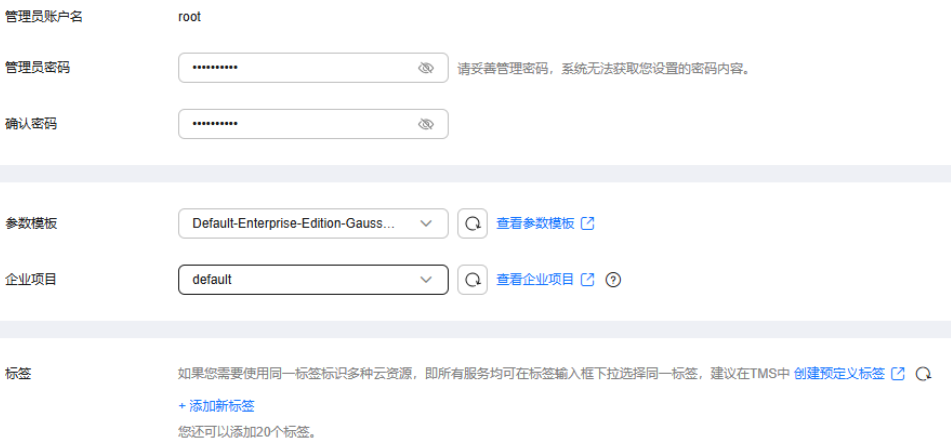
参数	示例	参数说明
存储空间	**40GB**	申请的存储空间会有必要的文件系统开销，这些开销包括索引节点和保留块，以及数据库运行必需的空间。
磁盘加密	**不加密**	选择加密后会提高数据安全性，但对数据库读写性能有少量影响，请按照您的使用策略进行选择。 如果使用共享KMS密钥，对应的CTS事件为createGrant，仅密钥所有者能够感知到该事件。

8. 配置网络信息，保持默认配置即可。

图 3-10 网络配置

9. 配置实例密码，选择企业项目信息。

图 3-11 数据库配置



版权所有 © 华为云计算技术有限公司

表 3-8 参数说明

参数	示例	参数说明
管理员密码	-	需要输入高强度密码并定期修改，以提高安全性，防止出现密码被暴力破解等安全风险。
确认密码	-	必须和管理员密码相同。
企业项目	**default**	对于已成功关联企业项目的用户，仅需在“企业项目”下拉框中选择目标项目。 如果需要自定义企业项目，请前往企业项目管理服务进行创建。关于如何创建项目，详见《企业项目管理用户指南》。

10. 单击“立即购买”，进入实例信息确认页面。
11. 确认信息无误后，单击“提交”。
12. 在实例列表中查看实例运行状态，当实例运行状态变为“正常”时，表示实例恢复完成。

步骤 4：确认恢复结果

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 查看并确认目标库名和表数据，确认是否恢复完成。

3.3 库表恢复

3.3.1 库表恢复：恢复到指定时间点

操作场景

如果您开启了表级自动备份策略，当误操作导致部分库表数据丢失时，可以使用已有的表级自动备份，恢复表数据到指定时间点。

版权所有 © 华为云计算技术有限公司

操作流程

操作步骤	说明
步骤1：构造数据	使用DAS新建数据库、新建表及插入数据。
步骤2：误删除表中数据	模拟误删除实例的操作。
步骤3：恢复表数据	使用表级时间点恢复误删除的表数据。
步骤4：确认恢复结果	登录DAS，确认数据是否恢复。

步骤 1：构造数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，创建数据库。

```
CREATE DATABASE db_tpcds;
```

   当结果显示为如下信息，则表示创建成功。

图 3-12 创建数据库

创建完db_tpcds数据库后，可以在左上方切换到新创建的库中。

版权所有 © 华为云计算技术有限公司

8. 执行如下SQL，新建表和插入数据。
   - 执行如下命令来创建一个schema。
     ```
     CREATE SCHEMA myschema;
     ```
     创建完schema后，可以在左上方切换到新创建的schema。
   - 创建一个名称为mytable，只有一列的表。字段名为firstcol，字段类型为integer。
     ```
     CREATE TABLE myschema.mytable (firstcol int);
     ```
   - 向表中插入数据：
     ```
     INSERT INTO myschema.mytable values (100);
     ```
9. 查询表数据。
   ```
   SELECT * FROM myschema.mytable;
   ```

步骤 2：误删除表中数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，删除表数据。
   ```
   DELETE FROM myschema.mytable WHERE firstcol = 100;
   ```

步骤 3：恢复表数据

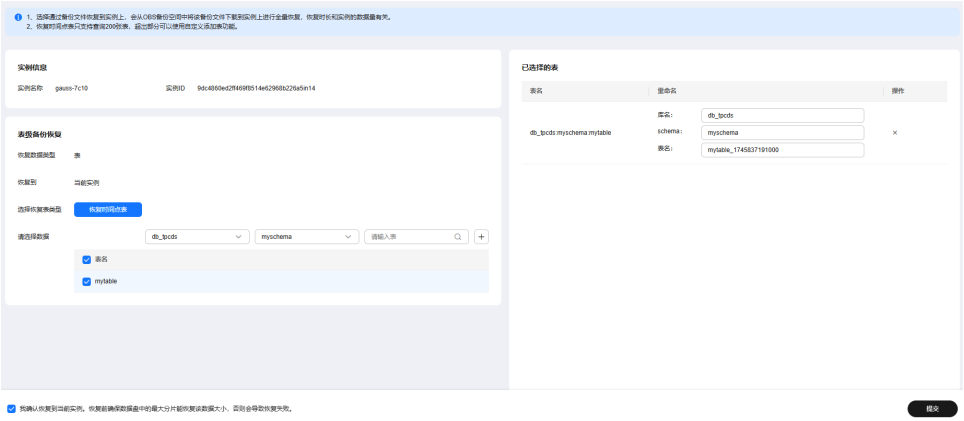
1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例基本信息页面。
5. 在左侧导航栏中选择“备份恢复”页签，然后单击“表级备份”。
6. 单击“恢复到表级指定时间点”，选择“可恢复的时间区间”，单击恢复到当前实例。

版权所有 © 华为云计算技术有限公司

图 3-13 恢复到指定时间点

7. 选择需要恢复的库表，单击提交。

图 3-14 选择待恢复的库表



8. 在“实例管理”页面，可查看目标实例状态为“恢复中”，当实例运行状态变为“正常”时，表示实例恢复完成。

步骤 4：确认恢复结果

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

版权所有 © 华为云计算技术有限公司

5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 查看并确认目标库名和表数据，确认是否恢复完成。

3.3.2 库表恢复：按备份文件恢复

操作场景

如果您开启了表级自动备份策略或手动创建了表级备份，当误操作导致部分库表数据丢失时，可以使用已有的表级备份，将指定表数据恢复到备份被创建时的状态。

操作流程

操作步骤	说明
步骤1：构造数据	使用DAS新建数据库、新建表及插入数据。
步骤2：误删除表中数据	模拟误删除实例的操作。
步骤3：恢复表数据	使用表级备份恢复误删除的表数据。
步骤4：确认恢复结果	登录DAS，确认数据是否恢复。

步骤 1：构造数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，创建数据库。

   ```
   CREATE DATABASE db_tpcds;
   ```

   当结果显示为如下信息，则表示创建成功。

版权所有 © 华为云计算技术有限公司

图 3-15 创建数据库

创建完db_tpcds数据库后，可以在左上方切换到新创建的库中。

8. 执行如下SQL，新建表和插入数据。
   - 执行如下命令来创建一个schema。
     ```
     CREATE SCHEMA myschema;
     ```
     创建完schema后，可以在左上方切换到新创建的schema。
   - 创建一个名称为mytable，只有一列的表。字段名为firstcol，字段类型为integer。
     ```
     CREATE TABLE myschema.mytable (firstcol int);
     ```
   - 向表中插入数据：
     ```
     INSERT INTO myschema.mytable values (100);
     ```
9. 查询表数据。
   ```
   SELECT * FROM myschema.mytable;
   ```

步骤 2：误删除表中数据

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。

   您也可以在“实例管理”页面，单击目标实例名称，进入“实例概览”页面，在页面右上角，单击“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 在顶部菜单栏选择“SQL操作”>“SQL查询”，打开一个SQL窗口。
7. 执行如下SQL，删除表数据。
   ```
   DELETE FROM myschema.mytable WHERE firstcol = 100;
   ```

步骤 3：恢复表数据

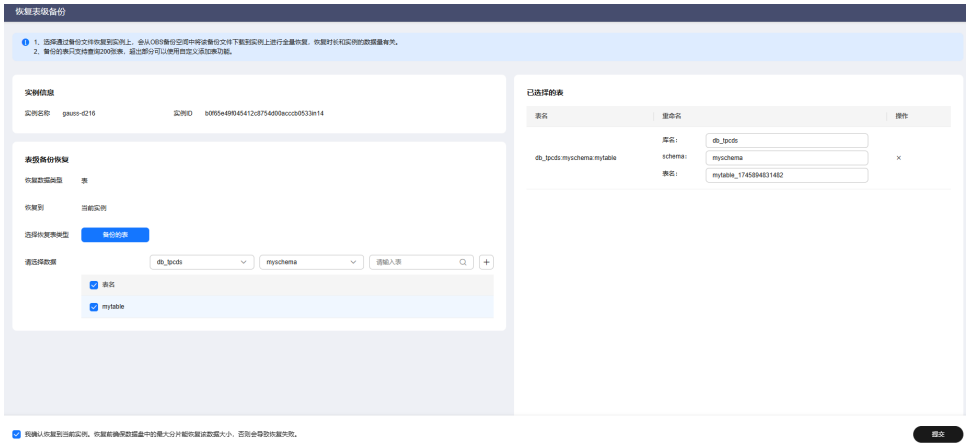
1. 登录管理控制台。

版权所有 © 华为云计算技术有限公司

2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例基本信息页面。
5. 在左侧导航栏中选择“备份恢复”页签，然后单击“表级备份”。
6. 单击目标备份对应操作列中的“恢复”。
7. 选择恢复到当前实例，单击“下一步”。

图 3-16 恢复表级备份

8. 选择需要恢复的库表，单击提交。



9. 在“实例管理”页面，可查看目标实例状态为“恢复中”，当实例运行状态变为“正常”时，表示实例恢复完成。

版权所有 © 华为云计算技术有限公司

步骤 4：确认恢复结果

1. 登录管理控制台。
2. 单击管理控制台左上角的 ，选择区域和项目。
3. 在页面左上角单击 ，选择“数据库 > 云数据库 GaussDB”，进入云数据库GaussDB信息页面。
4. 在“实例管理”页面，选择目标实例，单击操作列的“登录”，进入数据管理服务数据库登录界面。
5. 正确输入数据库用户名和密码，单击“测试连接”。测试连接通过后，单击“登录”，即可进入您的数据库并进行管理。
6. 查看并确认目标库名和表数据，确认是否恢复完成。

版权所有 © 华为云计算技术有限公司

4 行存压缩最佳实践

4.1 场景概述

行存压缩目的之一是在压缩表后复用节省的空间，具体过程可以简单概括为：

- **执行压缩。**遍历开启ILM特性的表的每个page进行压缩，压缩后的数据称之为BCA，仍存储在原来的page，压缩后page的剩余使用空间变大。需要注意的是执行完压缩后节省的空间不会还给操作系统。ILM特性相关介绍参见**数据生命周期管理-OLTP表压缩**。
- **插入新的数据。**新的数据会优先找到一个剩余使用空间大于新数据大小的page进行插入，行存压缩通过提高page的剩余使用空间来减少表的实际占用空间。

以下分别用一个手动调度压缩示例和一个自动调度压缩示例展示如何进行压缩。

- **手动调度**：需要手动调用压缩接口，且一次只能生成一个任务。
- **自动调度**：在前置操作设置完成后，无需手动调用接口，后台会通过定时任务自动创建压缩任务，且一次调度可以生成多个任务。

4.2 手动调度

步骤1 **登录GaussDB管理控制台**。在“实例管理”页面，单击目标实例名称，进入“基本信息”页面。

步骤2 找到“高级特性”字段，单击“查看并修改”，将“高级压缩”的值设置为on。详情可参考**查看并修改高级特性**

步骤3 连接数据库，然后执行如下命令，打开ilm特性总体开关。如何连接数据库可参考**通过gsql连接到数据库**。

```
alter database set ilm = on;
```

```
gaussdb=# alter database set ilm = on;
ALTER DATABASE
gaussdb=#
```

步骤4 执行如下命令，修改ilm的时间单位为秒。

版权所有 © 华为云计算技术有限公司

这一步的操作主要是为了加快测试速度。ilm默认的时间单位为天，执行手动压缩时，第一次调度是打时间戳，第二次调度才能压缩，第一次调度和第二次调度的间隔需要超过冷热数据判定时间的阈值才会执行压缩，以秒为时间单位可以加快测试速度。

1. 修改ilm的时间单位为秒。

```
BEGIN
DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
END;
/
```

2. 查看修改结果。

```
select * from gs_ilm_param;
```

```
gaussdb=# BEGIN
gaussdb$#    DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
gaussdb$# END;
gaussdb$# /
ANONYMOUS BLOCK EXECUTE
gaussdb=# select * from gs_ilm_param;
 idx |                name                | value
-----+-----------------------------------+-------
   1 | EXECUTION_INTERVAL                |    15
   2 | RETENTION_TIME                    |    30
   7 | ENABLED                           |     1
  12 | ABS_JOBLIMIT                      |    10
  13 | JOB_SIZELIMIT                     |  1024
  14 | WIND_DURATION                     |   240
  15 | BLOCK_LIMITS                      |    40
  16 | ENABLE_META_COMPRESSION           |     0
  17 | SAMPLE_MIN                        |    10
  18 | SAMPLE_MAX                        |    10
  19 | CONST_PRIO                        |    40
  20 | CONST_THRESHOLD                   |    90
  21 | EQVALUE_PRIO                      |    60
  22 | EQVALUE_THRESHOLD                 |    80
  23 | ENABLE_DELTA_ENCODE_SWITCH        |     1
  24 | LZ4_COMPRESSION_LEVEL             |     0
  25 | ENABLE_LZ4_PARTIAL_DECOMPRESSION  |     1
  11 | POLICY_TIME                       |     1
(18 rows)
```

步骤5 执行如下命令，修改一次调度最多压缩的数据大小。

本示例以修改为4GB为例。

```
BEGIN
    DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 4096);
END;
/
```

步骤6 建立带有ILM策略的表。

- 方法一：建表的同时添加策略。

  “3 DAYS OF NO MODIFICATION”是判定冷数据的时间阈值。

```
CREATE TABLE t (
    id1 int,
    id2 int,
    id3 int,
    id4 int)
WITH (orientation=row, compression=no, storage_type=astore)
ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 3 DAYS OF NO MODIFICATION;
```


版权所有 © 华为云计算技术有限公司

```
gaussdb=# CREATE TABLE t (
gaussdb(# id1 int,
gaussdb(# id2 int,
gaussdb(# id3 int,
gaussdb(# id4 int)
gaussdb-# WITH (orientation=row, compression=no, storage_type=astore)
gaussdb-# ILM ADD POLICY ROW STORE COMPRESS ADVANCED
gaussdb-# ROW AFTER 3 DAYS OF NO MODIFICATION;
CREATE TABLE
```

- 方法二：先建表再添加策略。

```
CREATE TABLE t (
      id1 int,
      id2 int,
      id3 int,
      id4 int)
WITH (orientation=row, compression=no, storage_type=astore);
ALTER TABLE t ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 3 DAYS OF NO MODIFICATION;
```

步骤7 执行如下命令插入数据。

```
insert into t (id1, id2 ,id3 , id4) select s, s, s, s from generate_series(1, 1000000) AS s;
```

步骤8 执行如下命令查看表t的原始大小，记为size1，本示例中size1 = 42MB。

```
\d+
```

```
gaussdb=# \d+
                                              List of relations
 Schema |         Name         |   Type   | Owner |    Size    |                         Storage                          | Description
--------+----------------------+----------+-------+------------+----------------------------------------------------------+-------------
 public | gs_job_name_sequence | sequence | omm   | 8192 bytes |                                                          |
 public | gsilmpolicy_seq      | sequence | omm   | 8192 bytes |                                                          |
 public | gsilmtask_seq        | sequence | omm   | 8192 bytes |                                                          |
 public | t                    | table    | omm   | 42 MB      | {orientation=row,compression=no,storage_type=astore}     |
(4 rows)
```

步骤9 执行压缩。

1. 先执行如下命令。

```
declare
v_taskid number;
begin
      dbe_ilm.execute_ilm('public','t',v_taskid,NULL,'ALL POLICIES',2);
end;
/
```

2. 等待10s后再次输入步骤9.1中的命令。
3. 查询压缩结果。

```
select * from gs_adm_ilmresults order by task_id desc limit 2;
```

可以看到，gs_adm_ilmresults打印了一些压缩信息，这里可以看到节约了13664840字节的空间。

```
gaussdb=# select * from gs_adm_ilmresults order by task_id desc limit 2;
 task_id |       job_name       |       job_state        |          start_time           |        completion_time        | comments |                              statistics
---------+----------------------+------------------------+-------------------------------+-------------------------------+----------+----------------------------------------------------------------------
       6 | ilmjob$_postgres_t_6 | COMPLETED SUCCESSFULLY | 2025-07-22 16:09:59.018799+08 | 2025-07-22 16:10:03.357066+08 |          | SpaceSaving=13664840,BoundTime=1753171796,StartBlkNum=0,LastBlkNum=5405
       5 | ilmjob$_postgres_t_5 | COMPLETED SUCCESSFULLY | 2025-07-22 16:09:51.924764+08 | 2025-07-22 16:09:55.050035+08 |          | SpaceSaving=0,BoundTime=1753171788,StartBlkNum=0,LastBlkNum=5405
(2 rows)
```

步骤10 步骤9中实际节约的空间并不一定能被完全复用，需要再次插入一份同样的数据测试真实压缩比，在二次插入数据后再次查看表t占用的空间。

1. 插入数据。

```
insert into t select * from t;
```

2. 查询表t的大小，记为size2，本示例中size2 = 71MB。

```
\d+
```

```
gaussdb=# \d+
                                              List of relations
 Schema |         Name         |   Type   | Owner |    Size    |                         Storage                          | Description
--------+----------------------+----------+-------+------------+----------------------------------------------------------+-------------
 public | gs_job_name_sequence | sequence | omm   | 8192 bytes |                                                          |
 public | gsilmpolicy_seq      | sequence | omm   | 8192 bytes |                                                          |
 public | gsilmtask_seq        | sequence | omm   | 8192 bytes |                                                          |
 public | t                    | table    | omm   | 71 MB      | {orientation=row,compression=no,storage_type=astore}     |
(4 rows)
```

步骤11 根据第一次插入数据和第二次插入数据后表t占用的空间计算压缩比。

压缩比 = size1 / (size2 – size1) = 42 / (71 - 42) = 1.45

----结束

4.3 自动调度

步骤1 登录GaussDB管理控制台。在“实例管理”页面，单击目标实例名称，进入“基本信息”页面。

步骤2 找到“高级特性”字段，单击“查看并修改”，将“高级压缩”的值设置为on。

步骤3 连接数据库，创建带ilm策略的数据表，并插入数据。如何连接数据库可参考通过gsql连接到数据库。

```
create database adb;
\c adb
alter database set ilm = on;
create table t1 (id int) with (orientation=row, compression=no, storage_type=astore) ilm add policy row
store compress advanced row after 3 days of no modification;
insert into t1 select * from generate_series(1, 10000000);
```

执行结果示例如下：

```
template1=# create database adb;
CREATE DATABASE
template1=#
template1=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=#
adb=# alter database set ilm = on;
ALTER DATABASE
adb=#
adb=# create table t1 (id int) with (orientation=row, compression=no, storage_type=astore) ilm add policy row store compress advanced row after 3 days of no modification;
CREATE TABLE
adb=#
adb=# insert into t1 select * from generate_series(1, 10000000);
INSERT 0 10000000
adb=#
```

步骤4 配置ilm自动调度系统参数。

调用DBE_ILM_ADMIN.CUSTOMIZE_ILM接口来修改调度相关的系统参数，分别传入参数对应的id以及value。这里修改了三条数据，分别是将自动调度执行频率修改为1分钟1次、将数据变冷行的时间单位修改成秒、将每个压缩job每次压缩的数据量大小设置为10M。其他参数可以不用修改，使用默认配置即可。

```
\c adb
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 10);
```

执行结果示例如下：

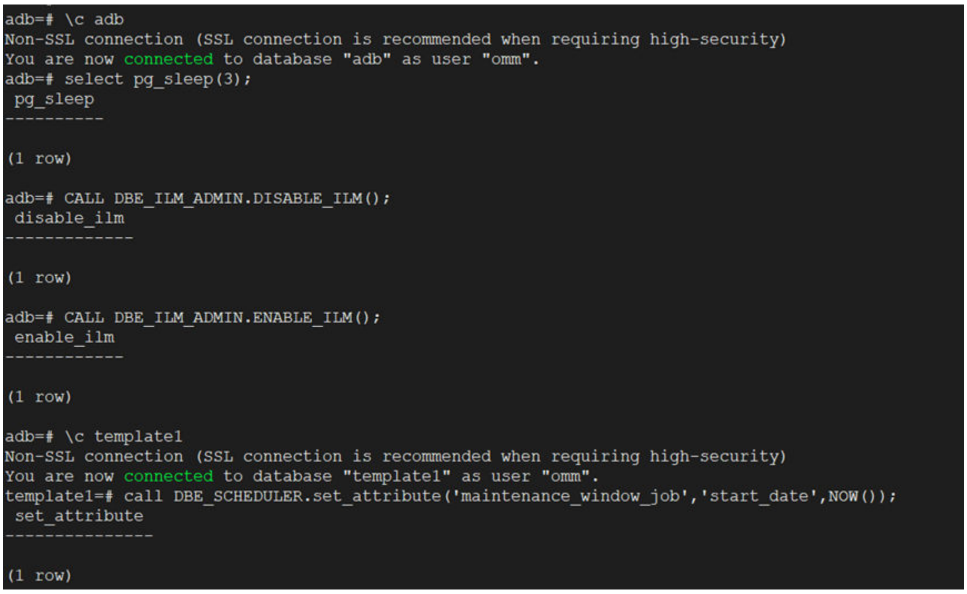
```
adb=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
 customize_ilm
---------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
 customize_ilm
---------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 10);
 customize_ilm
---------------

(1 row)
```


版权所有 © 华为云计算技术有限公司

步骤5 等待3秒让数据变冷行之后，开启自动调度。

```
\c adb
select pg_sleep(3);
CALL DBE_ILM_ADMIN.DISABLE_ILM();
CALL DBE_ILM_ADMIN.ENABLE_ILM();
\c template1
call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
```

执行结果示例如下：



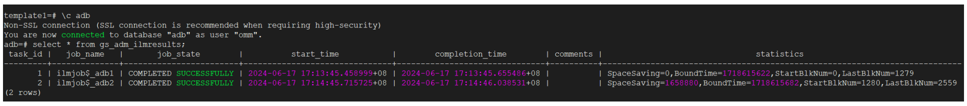
```
adb=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# select pg_sleep(3);
 pg_sleep
----------

(1 row)

adb=# CALL DBE_ILM_ADMIN.DISABLE_ILM();
 disable_ilm
-------------

(1 row)

adb=# CALL DBE_ILM_ADMIN.ENABLE_ILM();
 enable_ilm
------------

(1 row)

adb=# \c template1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "template1" as user "omm".
template1=# call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
 set_attribute
---------------

(1 row)
```

步骤6 查询调度任务的压缩结果。

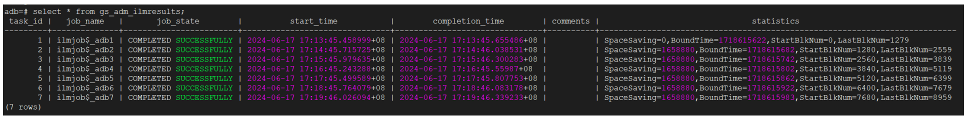
```
\c adb
select * from gs_adm_ilmresults;
```

执行结果示例如下：

调度任务一分钟执行一次，所以每隔一分钟才会生成一个新的压缩job。SpaceSaving为压缩收益大小，值越大表示压缩收益越大。



```
template1=# \c adb
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "adb" as user "omm".
adb=# select * from gs_adm_ilmresults;
 task_id |  job_name    |       job_state        |          start_time           |        completion_time        | comments |                          statistics
---------+--------------+------------------------+-------------------------------+-------------------------------+----------+---------------------------------------------------------------------------
       1 | ilmjob$_adb1 | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2 | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
(2 rows)
```



```
adb=# select * from gs_adm_ilmresults;
 task_id |  job_name    |       job_state        |          start_time           |        completion_time        | comments |                          statistics
---------+--------------+------------------------+-------------------------------+-------------------------------+----------+---------------------------------------------------------------------------
       1 | ilmjob$_adb1 | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2 | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
       3 | ilmjob$_adb3 | COMPLETED SUCCESSFULLY | 2024-06-17 17:15:45.979635+08 | 2024-06-17 17:15:46.300283+08 |          | SpaceSaving=1658880,BoundTime=1718615742,StartBlkNum=2560,LastBlkNum=3839
       4 | ilmjob$_adb4 | COMPLETED SUCCESSFULLY | 2024-06-17 17:16:45.243288+08 | 2024-06-17 17:16:45.55987+08  |          | SpaceSaving=1658880,BoundTime=1718615802,StartBlkNum=3840,LastBlkNum=5119
       5 | ilmjob$_adb5 | COMPLETED SUCCESSFULLY | 2024-06-17 17:17:45.499589+08 | 2024-06-17 17:17:45.807753+08 |          | SpaceSaving=1658880,BoundTime=1718615862,StartBlkNum=5120,LastBlkNum=6399
       6 | ilmjob$_adb6 | COMPLETED SUCCESSFULLY | 2024-06-17 17:18:45.764079+08 | 2024-06-17 17:18:46.083178+08 |          | SpaceSaving=1658880,BoundTime=1718615922,StartBlkNum=6400,LastBlkNum=7679
       7 | ilmjob$_adb7 | COMPLETED SUCCESSFULLY | 2024-06-17 17:19:46.026094+08 | 2024-06-17 17:19:46.339233+08 |          | SpaceSaving=1658880,BoundTime=1718615983,StartBlkNum=7680,LastBlkNum=8959
(7 rows)
```

如果想让表尽快地压缩完成，可以将每个压缩job每次压缩的数据量大小设置得更大一些，比如1024，然后重新启动调度任务，并隔一分钟再次查看压缩结果。

```
\c adb
CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 1024);
select pg_sleep(3);
CALL DBE_ILM_ADMIN.DISABLE_ILM();
CALL DBE_ILM_ADMIN.ENABLE_ILM();
\c template1
call DBE_SCHEDULER.set_attribute('maintenance_window_job','start_date',NOW());
```

执行结果示例如下：

```
adb=# select * from gs_adm_ilmresults;
 task_id |   job_name    |        job_state         |          start_time           |        completion_time        | comments |                              statistics
---------+---------------+--------------------------+-------------------------------+-------------------------------+----------+----------------------------------------------------------------------
       1 | ilmjob$_adb1  | COMPLETED SUCCESSFULLY | 2024-06-17 17:13:45.458999+08 | 2024-06-17 17:13:45.655486+08 |          | SpaceSaving=0,BoundTime=1718615622,StartBlkNum=0,LastBlkNum=1279
       2 | ilmjob$_adb2  | COMPLETED SUCCESSFULLY | 2024-06-17 17:14:45.715725+08 | 2024-06-17 17:14:46.038531+08 |          | SpaceSaving=1658880,BoundTime=1718615682,StartBlkNum=1280,LastBlkNum=2559
       3 | ilmjob$_adb3  | COMPLETED SUCCESSFULLY | 2024-06-17 17:15:45.979635+08 | 2024-06-17 17:15:46.300283+08 |          | SpaceSaving=1658880,BoundTime=1718615742,StartBlkNum=2560,LastBlkNum=3839
       4 | ilmjob$_adb4  | COMPLETED SUCCESSFULLY | 2024-06-17 17:16:45.243288+08 | 2024-06-17 17:16:45.55987+08  |          | SpaceSaving=1658880,BoundTime=1718615802,StartBlkNum=3840,LastBlkNum=5119
       5 | ilmjob$_adb5  | COMPLETED SUCCESSFULLY | 2024-06-17 17:17:45.499589+08 | 2024-06-17 17:17:45.807753+08 |          | SpaceSaving=1658880,BoundTime=1718615862,StartBlkNum=5120,LastBlkNum=6399
       6 | ilmjob$_adb6  | COMPLETED SUCCESSFULLY | 2024-06-17 17:18:45.764079+08 | 2024-06-17 17:18:46.083178+08 |          | SpaceSaving=1658880,BoundTime=1718615922,StartBlkNum=6400,LastBlkNum=7679
       7 | ilmjob$_adb7  | COMPLETED SUCCESSFULLY | 2024-06-17 17:19:46.026094+08 | 2024-06-17 17:19:46.339233+08 |          | SpaceSaving=1658880,BoundTime=1718615983,StartBlkNum=7680,LastBlkNum=8959
       8 | ilmjob$_adb8  | COMPLETED SUCCESSFULLY | 2024-06-17 17:20:16.551575+08 | 2024-06-17 17:20:31.569778+08 |          | SpaceSaving=45732960,BoundTime=1718616013,StartBlkNum=8960,LastBlkNum=44247
       9 | ilmjob$_adb9  | COMPLETED SUCCESSFULLY | 2024-06-17 17:21:16.813508+08 | 2024-06-17 17:21:20.225773+08 |          | SpaceSaving=1658880,BoundTime=1718616073,StartBlkNum=0,LastBlkNum=44247
      10 | ilmjob$_adb10 | COMPLETED SUCCESSFULLY | 2024-06-17 17:22:17.075173+08 | 2024-06-17 17:22:20.248922+08 |          | SpaceSaving=0,BoundTime=1718616134,StartBlkNum=0,LastBlkNum=44247
(10 rows)
```

----结束

版权所有 © 华为云计算技术有限公司

5 SQL 查询最佳实践

5.1 SQL 查询最佳实践（分布式）

根据数据库的SQL执行机制以及大量的实践总结发现：通过一定的规则调整SQL语句，在保证结果正确的基础上，能够提高SQL执行效率。

- **使用UNION ALL代替UNION。**

  UNION在合并两个集合时会执行去重操作，而UNION ALL则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用UNION ALL替代UNION以便提升性能。

- **JOIN列增加非空过滤条件。**

  若JOIN列上的NULL值较多，则可以加上IS NOT NULL过滤条件，以实现数据的提前过滤，提高JOIN效率。

- **NOT IN转NOT EXISTS。**

  NOT IN语句需要使用NESTLOOP ANTI JOIN来实现，而NOT EXISTS则可以通过HASH ANTI JOIN来实现。在JOIN列不存在NULL值的情况下，NOT EXISTS和NOT IN等价。因此在确保没有NULL值时，可以通过将NOT IN转换为NOT EXISTS，通过生成HASH JOIN来提升查询效率。

  建表语句如下：

```
gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
gaussdb=# CREATE SCHEMA no_in_to_no_exists_test;
gaussdb=# SET CURRENT_SCHEMA=no_in_to_no_exists_test;
gaussdb=# CREATE TABLE t1(c1 int, c2 int, c3 int);
gaussdb=# CREATE TABLE t2(d1 int, d2 int NOT NULL, d3 int);
```

  使用NOT IN实现查询语句如下：

```
gaussdb=# SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
```

  其计划如下所示：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
 id |                 operation                 | E-rows | E-width | E-costs
----+-------------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)              |     15 |      12 | 29.65
  2 |    ->  Seq Scan on t1                     |     15 |      12 | 28.77
  3 |       ->  Materialize  [2, SubPlan 1]     |    270 |       4 | 14.37
  4 |          ->  Streaming(type: BROADCAST)   |     90 |       4 | 14.22
  5 |             ->  Seq Scan on t2            |     30 |       4 | 14.14
(5 rows)
```


版权所有 © 华为云计算技术有限公司

```
 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Seq Scan on t1
         Filter: (NOT (hashed SubPlan 1))
(2 rows)
```

因为t2.d2字段中没有NULL值（t2.d2字段在表定义中为NOT NULL），所以查询可以等价修改如下：

```
gaussdb=# SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
```

其生成的计划如下：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
 id |                 operation                 | E-rows | E-width | E-costs
----+-------------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)              |      3 |      12 | 29.99
  2 |    ->  Hash Right Anti Join (3, 5)        |      3 |      12 | 29.86
  3 |       ->  Streaming(type: REDISTRIBUTE) |     30 |       4 | 15.49
  4 |          ->  Seq Scan on t2               |     30 |       4 | 14.14
  5 |       ->  Hash                            |     29 |      12 | 14.14
  6 |          ->  Seq Scan on t1               |     30 |      12 | 14.14
(6 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Hash Right Anti Join (3, 5)
         Hash Cond: (t2.d2 = t1.c1)
(2 rows)

--删除。
gaussdb=# DROP TABLE t1,t2;
gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
```

- **选择hashagg。**

  查询语句中如果存在GROUP BY条件则生成的计划（Plan）中可能存在排序操作，即计划中包含GroupAgg+Sort算子，导致性能较差。可以通过设置GUC参数work_mem增大可用内存，生成带有HashAgg的计划（Plan）避免排序操作从而提升性能。work_mem设置请联系管理员。

- **尝试将函数替换为case语句。**

  GaussDB函数调用性能较低，如果出现过多的函数调用导致性能下降很多，可以根据情况把可下推函数的函数改成CASE表达式。

- **避免对索引使用函数或表达式运算。**

  对索引使用函数或表达式运算会停止使用索引转而执行全表扫描。

- **尽量避免在where子句中使用!=或<>操作符、null值判断、or连接、参数隐式转换。**

- **对于高频数据变化的表，在相关SQL语句中添加Hint，以固定执行计划。**

  高频数据变化的表可能在触发自动ANALYZE之前出现统计信息不是最新的情况，从而导致执行计划选择不优。建议通过在相关SQL中添加Hint来固定执行计划。

- **对复杂SQL语句进行拆分。**

  对于过于复杂并且不易通过以上方法调整性能的SQL可以考虑拆分的方法，把SQL中某一部分拆分成独立的SQL并把执行结果存入临时表，拆分常见的场景包括但不限于：

  – 作业中多个SQL有同样的子查询，并且子查询数据量较大。
  – Plan cost计算不准，导致子查询hash bucket太小，比如实际数据1000W行，hash bucket只有1000。
  – 函数（如substr、to_number）导致大数据量子查询选择度计算不准。

– 多DN环境下对大表做broadcast的子查询。

其他更多调优点，请参见《开发指南》中“SQL调优指南 > 典型SQL调优点”章节。

5.2 SQL 查询最佳实践（集中式）

根据数据库的SQL执行机制以及大量的实践总结发现：通过一定的规则调整SQL语句，在保证结果正确的基础上，能够提高SQL执行效率。

- **使用UNION ALL代替UNION。**

  UNION在合并两个集合时会执行去重操作，而UNION ALL则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用UNION ALL替代UNION以便提升性能。

- **JOIN列增加非空过滤条件。**

  若JOIN列上的NULL值较多，则可以加上IS NOT NULL过滤条件，以实现数据的提前过滤，提高JOIN效率。

- **NOT IN转NOT EXISTS。**

  NOT IN语句需要使用NESTLOOP ANTI JOIN来实现，而NOT EXISTS则可以通过HASH ANTI JOIN来实现。在JOIN列不存在NULL值的情况下，NOT EXISTS和NOT IN等价。因此在确保没有NULL值时，可以通过将NOT IN转换为NOT EXISTS，通过生成HASH JOIN来提升查询效率。

  建表语句如下：

```
gaussdb=# DROP SCHEMA IF EXISTS no_in_to_no_exists_test CASCADE;
gaussdb=# CREATE SCHEMA no_in_to_no_exists_test;
gaussdb=# SET CURRENT_SCHEMA=no_in_to_no_exists_test;
gaussdb=# CREATE TABLE t1(c1 int, c2 int, c3 int);
gaussdb=# CREATE TABLE t2(d1 int, d2 int NOT NULL, d3 int);
```

  使用NOT IN实现查询语句如下：

```
gaussdb=# SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
```

  其计划如下所示：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE c1 NOT IN (SELECT d2 FROM t2);
 id |              operation              | E-rows | E-width |    E-costs
----+-------------------------------------+--------+---------+----------------
  1 | ->  Seq Scan on t1                  |    972 |      12 | 34.312..68.625
  2 |    ->  Seq Scan on t2  [1, SubPlan 1] |   1945 |       4 | 0.000..29.450
(2 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   1 --Seq Scan on t1
         Filter: (NOT (hashed SubPlan 1))
(2 rows)
```

  因为t2.d2字段中没有NULL值（t2.d2字段在表定义中为NOT NULL），所以查询可以等价修改如下：

```
gaussdb=# SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
```

  其生成的计划如下：

```
gaussdb=# EXPLAIN SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
 id |         operation          | E-rows | E-width |    E-costs
----+----------------------------+--------+---------+----------------
  1 | ->  Hash Anti Join (2, 3) |    972 |      12 | 53.763..99.142
  2 |    ->  Seq Scan on t1      |   1945 |      12 | 0.000..29.450
  3 |    ->  Hash                |   1945 |       4 | 29.450..29.450
  4 |       ->  Seq Scan on t2  |   1945 |       4 | 0.000..29.450
(4 rows)
```

```
 Predicate Information (identified by plan id)
-----------------------------------------------
  1 --Hash Anti Join (2, 3)
        Hash Cond: (t1.c1 = t2.d2)
(2 rows)
```

- **选择hashagg。**

  查询语句中如果存在GROUP BY条件则生成的计划（Plan）中可能存在排序操作，即计划中包含GroupAgg+Sort算子，导致性能较差。可以通过设置GUC参数work_mem增大可用内存，生成带有HashAgg的计划（Plan）避免排序操作从而提升性能。work_mem设置请联系管理员。

- **尝试将函数替换为case语句。**

  数据库函数调用性能较低，如果出现过多的函数调用导致性能下降很多，可以根据情况把可下推函数的函数改成CASE表达式。

- **避免对索引使用函数或表达式运算。**

  对索引使用函数或表达式运算会停止使用索引转而执行全表扫描。

- **尽量避免在where子句中使用!=或<>操作符、null值判断、or连接、参数隐式转换。**

- **对于高频数据变化的表，在相关SQL语句中添加Hint，以固定执行计划。**

  高频数据变化的表可能在触发自动ANALYZE之前出现统计信息不是最新的情况，从而导致执行计划选择不优。建议通过在相关SQL中添加Hint来固定执行计划。

- **对复杂SQL语句进行拆分。**

  对于过于复杂并且不易通过以上方法调整性能的SQL可以考虑拆分的方法，把SQL中某一部分拆分成独立的SQL并把执行结果存入临时表，拆分常见的场景包括但不限于：

  - 作业中多个SQL有同样的子查询，并且子查询数据量较大。
  - Plan cost计算不准，导致子查询hash bucket太小，比如实际数据1000万行，hash bucket只有1000。
  - 函数（如substr、to_number）导致大数据量子查询选择度计算不准。
  - 多DN环境下对大表做broadcast的子查询。

其他更多调优点，请参考《开发指南》中“SQL调优指南 > 典型SQL调优点”章节。

6 权限配置最佳实践

6.1 权限配置最佳实践（分布式）

背景

一个数据库可能有很多的用户需要访问，为了方便管理这些用户，将用户组成一个数据库角色。一个数据库角色可以视为一个数据库用户或者一组数据库用户。

对于数据库来说，用户和角色是基本相同的概念，不同之处在于，使用CREATE ROLE创建角色，不会创建同名的SCHEMA，并且默认没有LOGIN权限；而使用CREATE USER创建用户，会自动创建同名的SCHEMA，默认有LOGIN权限。换句话说，一个拥有LOGIN权限的角色可以被认为是一个用户。在业务设计中，仅建议通过ROLE来组织权限，而不是用来访问数据库。

最佳实践概述

权限配置不当会存在权限被利用的风险，本章节描述各权限角色的作用。

解决方案

- **数据库用户**

  数据库用户的主要用途是使用该用户账号连接数据库、访问数据库对象和执行SQL语句。在连接数据库时，必须使用一个已经存在的数据库用户。因此，作为数据库管理员，需要为每一个需要连接数据库的使用者规划一个数据库用户。

  在创建数据库用户时，至少需要指定用户名和密码。

  默认情况下，数据库用户可以分为两大类，详细信息请参见**表6-1**。

版权所有 © 华为云计算技术有限公司

表 6-1 用户分类

分类	描述
初始用户	具有数据库的最高权限，并且具有所有的系统权限和对象权限。初始用户不受对象的权限设置影响。这个特点类似Unix系统的root的权限。从安全角度考虑，除了必要的情况，建议尽量避免以初始用户身份操作。 在安装数据库或者初始化数据库时，可以指定初始用户名和密码。如果不指定用户名则会自动生成一个与安装数据库的OS用户同名的初始用户。如果不指定密码则安装后初始用户密码为空，需要通过gsql客户端设置初始用户的密码后才能执行其他操作。 说明： 基于安全性考虑，GaussDB Kernel禁止了所有用户trust方式的远程登录方式，禁止了初始用户的任何方式的远程登录。
普通用户	默认可以访问数据库的默认系统表和视图（pg_authid、pg_largeobject、pg_user_status和pg_auth_history除外），可以连接默认的数据库postgres以及使用public模式下的对象（包括表、视图和函数等）。 ● 可以通过CREATE USER、ALTER USER指定系统权限，或者通过GRANT ALL PRIVILEGE授予SYSADMIN权限。 ● 可以通过GRANT语句授予某些对象的权限。 ● 可以通过GRANT语法将其他角色或用户的权限授权给该用户。

- **数据库权限分类**

  通过权限和角色，可以控制用户访问指定的数据，以及执行指定类型的SQL语句。详细信息请参见**表6-2**。

  系统权限只能通过CREATE/ALTER USER、CREATE/ALTER ROLE语句指定（其中SYSADMIN还可以通过GRANT/REVOKE ALL PRIVILEGES的方式赋予、回收），无法从角色继承。

表 6-2 权限分类

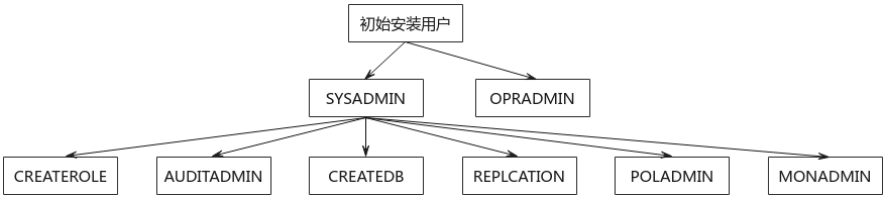
分类	描述
系统权限	系统权限又称为用户属性，可以在创建用户和修改用户时指定，包括SYSADMIN、MONADMIN、OPRADMIN、POLADMIN、CREATEDB、CREATEROLE、AUDITADMIN和LOGIN。 系统权限一般通过CREATE/ALTER USER语句指定。除了SYSADMIN外的其他系统权限，无法通过GRANT/REVOKE进行授予和撤销。并且，系统权限无法通过ROLE被继承。
对象权限	对象权限是指在表、视图、索引和函数等数据库对象上执行各种操作的权限，对象权限包括SELECT、INSERT、UPDATE和DELETE等。 只有对象的所有者或者系统管理员才可以执行GRANT/REVOKE语句来分配/撤销对象权限。

版权所有 © 华为云计算技术有限公司

分类	描述
角色	角色是一组权限的集合，可以将一个角色的权限赋予其他角色和用户。 由于无法给其他角色和用户赋予系统权限，所以角色只有是对象权限的集合时才有意义。

- **数据库权限模型**
  - **系统权限模型**
    - **默认权限机制**

图 6-1 权限架构

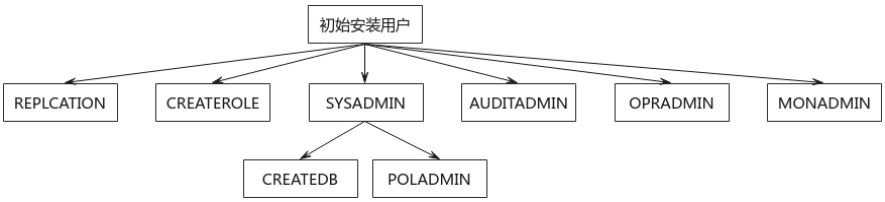


权限架构如图6-1，默认权限机制下sysadmin具有大多数的权限。

- **初始安装用户**：集群安装过程中自动生成的账户，拥有系统的最高权限，能够执行所有的操作。
- **SYSADMIN**：系统管理员权限，权限仅次于初始安装用户，默认具有与对象所有者相同的权限，默认不包括监控管理员权限和运维管理员权限，但可以赋予自己监控管理员权限。
- **MONADMIN**：监控管理员权限，具有监控模式dbe_perf及模式下视图和函数的访问权限和授予权限。
- **OPRADMIN**：运维管理员权限，具有使用Roach工具执行备份恢复的权限。
- **CREATEROLE**：安全管理员权限，具有创建、修改、删除用户/角色的权限。
- **AUDITADMIN**：审计管理员权限，具有查看和维护数据库审计日志的权限。
- **CREATEDB**：具有创建数据库的权限。
- **POLADMIN**：安全策略管理员权限，具有创建资源标签，数据动态脱敏策略和统一审计策略的权限。

    - **三权分立机制**

图 6-2 三权分立机制



版权所有 © 华为云计算技术有限公司

- **SYSADMIN**：系统管理员权限，不再具有创建、修改、删除用户/角色的权限，也不再具有查看和维护数据库审计日志的权限。
- **CREATEROLE**：安全管理员权限，具有创建、修改、删除用户/角色的权限。
- **AUDITADMIN**：审计管理员权限，具有查看和维护数据库审计日志的权限。
- 一个用户/角色只能具有SYSADMIN、CREATEROLE和AUDITADMIN中的一项系统权限。

– **对象权限模型**

- 对象权限：指在数据库、模式、表等数据库对象上执行特定动作的权限，比如：SELECT、INSERT、UPDATE、DELETE和CONNECT等。
- 针对不同的数据库对象有不同的对象权限，相应地可以被授予用户/角色。
- 通过GRANT/REVOKE来传递对象权限，对象权限可以通过角色被继承。

– **角色权限模型**

GaussDB Kernel提供了一组默认角色，以gs_role_开头命名。它们提供对特定的、通常需要高权限操作的访问，可以将这些角色GRANT给数据库内的其他用户或角色，让这些用户能够使用特定的功能。在授予这些角色时应当非常小心，以确保它们被用在需要的地方。表6-3描述了内置角色允许的权限范围。

表 6-3 内置角色权限

角色	权限描述
gs_role_signal_backend	具有调用函数pg_cancel_backend()、pg_terminate_backend()和pg_terminate_session()来取消或终止其他会话的权限，但不能操作属于初始用户和PERSISTENCE用户的会话。
gs_role_tablespace	具有创建表空间（tablespace）的权限。
gs_role_replication	具有调用逻辑复制相关函数的权限，例如kill_snapshot()、pg_create_logical_replication_slot()、pg_create_physical_replication_slot()、pg_drop_replication_slot()、pg_replication_slot_advance()、pg_create_physical_replication_slot_extern()、pg_logical_slot_get_changes()、pg_logical_slot_peek_changes()，pg_logical_slot_get_binary_changes()、pg_logical_slot_peek_binary_changes()。
gs_role_account_lock	具有加解锁用户的权限，但不能加解锁初始用户和PERSISTENCE用户。
gs_role_pldebugger	具有执行dbe_pldebugger下调试函数的权限。

角色	权限描述
gs_role_public_dblink_drop	具有执行删除public database link对象的权限。
gs_role_public_dblink_alter	具有执行修改public database link对象的权限。
gs_role_seclabel	具有创建、删除和应用安全标签的权限。
gs_role_public_synonym_create	具有创建public同义词的权限。
gs_role_public_synonym_drop	具有删除public同义词的权限。

- 系统权限配置
  - **默认权限机制配置方法**
    - **初始用户**

      数据库安装过程中自动生成的账户称为初始用户。初始用户也是系统管理员、监控管理员、运维管理员和安全策略管理员，拥有系统的最高权限，能够执行所有的操作。如果安装时不指定初始用户名称则该账户与进行数据库安装的操作系统用户同名。如果在安装时不指定初始用户的密码，安装完成后密码为空，在执行其他操作前需要通过gsql客户端修改初始用户的密码。如果初始用户密码为空，则除修改密码外无法执行其他SQL操作以及升级、扩容和节点替换等操作。

      初始用户会绕过所有权限检查。建议仅将此初始用户作为DBA管理用途，而非业务用途。

    - **系统管理员**

```
gaussdb=#CREATE USER u_sysadmin WITH SYSADMIN password '********';

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_sysadmin01 SYSADMIN;
```

    - **监控管理员**

```
gaussdb=#CREATE USER u_monadmin WITH MONADMIN password '********';

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_monadmin01 MONADMIN;
```

    - **运维管理员**

```
gaussdb=#CREATE USER u_opradmin WITH OPRADMIN password "xxxxxxxxx";

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_opradmin01 OPRADMIN;
```

    - **安全策略管理员**

```
gaussdb=#CREATE USER u_poladmin WITH POLADMIN password "xxxxxxxxx";

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_poladmin01 POLADMIN;
```


版权所有 © 华为云计算技术有限公司

– **三权分立机制配置方式**

此模式需要设置guc参数“enableSeparationOfDuty”的值为“on”，该参数为POSTMASTER类型参数，修改完之后需要重启数据库。

```
gs_guc set -Z coordinator -Z datanode -N all -I all -c "enableSeparationOfDuty=on"
gs_om -t stop
gs_om -t start
```

创建和配置相应的用户权限的语法和默认权限一致。

- 角色权限配置

```
--创建数据库test。
gaussdb=#CREATE DATABASE test;
--创建角色role1，创建用户user1。
gaussdb=#CREATE ROLE role1 PASSWORD '********';
gaussdb=#CREATE USER user1 PASSWORD '********';
--赋予CREATE ANY TABLE权限角色role1
gaussdb=#GRANT CREATE ON DATABASE test TO role1;

--将角色role1赋予给用户user1,则用户user1属于组role1,继承role1的相应权限可以在test数据库中创建模式。
gaussdb=#GRANT role1 TO user1;

--查询用户和角色信息。
gaussdb=#\du role1|user1;
            List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 role1     | Cannot login | {}
 user1     |              | {role1}
```

实践效果

无。

6.2 权限配置最佳实践（集中式）

背景

一个数据库可能有很多的用户需要访问，为了方便管理这些用户，将用户组成一个数据库角色。一个数据库角色可以视为一个数据库用户或者一组数据库用户。

对于数据库来说，用户和角色是基本相同的概念，不同之处在于，使用CREATE ROLE创建角色，不会创建同名的SCHEMA，并且默认没有LOGIN权限；而使用CREATE USER创建用户，会自动创建同名的SCHEMA，默认有LOGIN权限。换句话说，一个拥有LOGIN权限的角色可以被认为是一个用户。在业务设计中，仅建议通过ROLE来组织权限，而不是用来访问数据库。

最佳实践概述

权限配置不当会存在权限被利用的风险，本章节描述各权限角色的作用。

解决方案

- **数据库用户**

  数据库用户的主要用途是使用该用户账号连接数据库、访问数据库对象和执行SQL语句。在连接数据库时，必须使用一个已经存在的数据库用户。因此，作为数据库管理员，需要为每一个需要连接数据库的使用者规划一个数据库用户。

  在创建数据库用户时，至少需要指定用户名和密码。

默认情况下，数据库用户可以分为两大类，详细信息请参见表6-4。

表 6-4 用户分类

分类	描述
初始用户	具有数据库的最高权限，并且具有所有的系统权限和对象权限。初始用户不受对象的权限设置影响。这个特点类似Unix系统的root的权限。从安全角度考虑，除了必要的情况，建议尽量避免以初始用户身份操作。 在安装数据库或者初始化数据库时，可以指定初始用户名和密码。如果不指定用户名则会自动生成一个与安装数据库的OS用户同名的初始用户。如果不指定密码则安装后初始用户密码为空，需要通过gsql客户端设置初始用户的密码后才能执行其他操作。 说明： 基于安全性考虑，GaussDB Kernel禁止了所有用户trust方式的远程登录方式，禁止了初始用户的任何方式的远程登录。
普通用户	默认可以访问数据库的默认系统表和视图（pg_authid、pg_largeobject、pg_user_status和pg_auth_history除外），可以连接默认的数据库postgres以及使用public模式下的对象（包括表、视图和函数等）。 ● 可以通过CREATE USER、ALTER USER指定系统权限，或者通过GRANT ALL PRIVILEGE授予SYSADMIN权限。 ● 可以通过GRANT语句授予某些对象的权限。 ● 可以通过GRANT语法将其他角色或用户的权限授权给该用户。

- **数据库权限分类**

  通过权限和角色，可以控制用户访问指定的数据，以及执行指定类型的SQL语句。详细信息请参见表6-5。

  系统权限只能通过CREATE/ALTER USER、CREATE/ALTER ROLE语句指定（其中SYSADMIN还可以通过GRANT/REVOKE ALL PRIVILEGES的方式赋予、回收），无法从角色继承。

表 6-5 权限分类

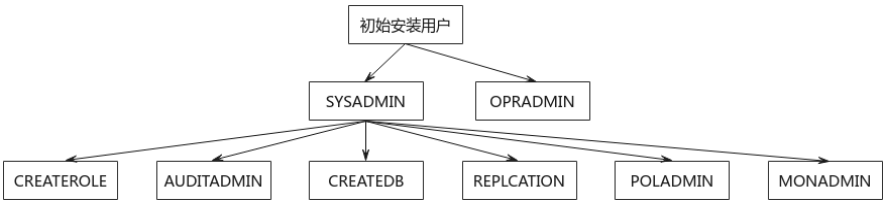
分类	描述
系统权限	系统权限又称为用户属性，可以在创建用户和修改用户时指定，包括SYSADMIN、MONADMIN、OPRADMIN、POLADMIN、CREATEDB、CREATEROLE、AUDITADMIN和LOGIN。 系统权限一般通过CREATE/ALTER USER语句指定。除了SYSADMIN外的其他系统权限，无法通过GRANT/REVOKE进行授予和撤销。并且，系统权限无法通过ROLE被继承。

版权所有 © 华为云计算技术有限公司

分类	描述
对象权限	对象权限是指在表、视图、索引和函数等数据库对象上执行各种操作的权限，对象权限包括SELECT、INSERT、UPDATE和DELETE等。 只有对象的所有者或者系统管理员才可以执行GRANT/REVOKE语句来分配/撤销对象权限。
角色	角色是一组权限的集合，可以将一个角色的权限赋予其他角色和用户。 由于无法给其他角色和用户赋予系统权限，所以角色只有是对象权限的集合时才有意义。

- **数据库权限模型**
  - **系统权限模型**
    - **默认权限机制**

图 6-3 权限架构



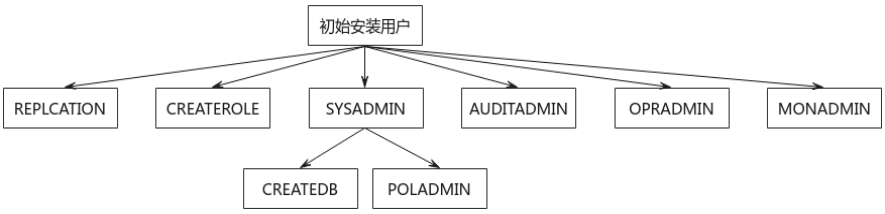
权限架构如图6-3，默认权限机制下sysadmin具有大多数的权限。

- **初始安装用户**：集群安装过程中自动生成的账户，拥有系统的最高权限，能够执行所有的操作。
- **SYSADMIN**：系统管理员权限，权限仅次于初始安装用户，默认具有与对象所有者相同的权限，默认不包括监控管理员权限和运维管理员权限，但可以赋予自己监控管理员权限。
- **MONADMIN**：监控管理员权限，具有监控模式dbe_perf及模式下视图和函数的访问权限和授予权限。
- **OPRADMIN**：运维管理员权限，具有使用Roach工具执行备份恢复的权限。
- **CREATEROLE**：安全管理员权限，具有创建、修改、删除用户/角色的权限。
- **AUDITADMIN**：审计管理员权限，具有查看和维护数据库审计日志的权限。
- **CREATEDB**：具有创建数据库的权限。
- **POLADMIN**：安全策略管理员权限，具有创建资源标签，数据动态脱敏策略和统一审计策略的权限。

    - **三权分立机制**

版权所有 © 华为云计算技术有限公司

图 6-4 三权分立机制



- **SYSADMIN**：系统管理员权限，不再具有创建、修改、删除用户/角色的权限，也不再具有查看和维护数据库审计日志的权限。
- **CREATEROLE**：安全管理员权限，具有创建、修改、删除用户/角色的权限。
- **AUDITADMIN**：审计管理员权限，具有查看和维护数据库审计日志的权限。
- 一个用户/角色只能具有SYSADMIN、CREATEROLE和AUDITADMIN中的一项系统权限。

– **对象权限模型**

  - 对象权限：指在数据库、模式、表等数据库对象上执行特定动作的权限，比如：SELECT、INSERT、UPDATE、DELETE和CONNECT等。
  - 针对不同的数据库对象有不同的对象权限，相应地可以被授予用户/角色。
  - 通过GRANT/REVOKE来传递对象权限，对象权限可以通过角色被继承。

– **角色权限模型**

  GaussDB Kernel提供了一组默认角色，以gs_role_开头命名。它们提供对特定的、通常需要高权限操作的访问，可以将这些角色GRANT给数据库内的其他用户或角色，让这些用户能够使用特定的功能。在授予这些角色时应当非常小心，以确保它们被用在需要的地方。表6-6描述了内置角色允许的权限范围。

表 6-6 内置角色权限

角色	权限描述
gs_role_signal_backend	具有调用函数pg_cancel_backend()、pg_terminate_backend()和pg_terminate_session()来取消或终止其他会话的权限，但不能操作属于初始用户和PERSISTENCE用户的会话。
gs_role_tablespace	具有创建表空间（tablespace）的权限。

版权所有 © 华为云计算技术有限公司

角色	权限描述
gs_role_replication	具有调用逻辑复制相关函数的权限，例如kill_snapshot()、pg_create_logical_replication_slot()、pg_create_physical_replication_slot()、pg_drop_replication_slot()、pg_replication_slot_advance()、pg_create_physical_replication_slot_extern()、pg_logical_slot_get_changes()、pg_logical_slot_peek_changes()，pg_logical_slot_get_binary_changes()、pg_logical_slot_peek_binary_changes()。
gs_role_account_lock	具有加解锁用户的权限，但不能加解锁初始用户和PERSISTENCE用户。
gs_role_pldebugger	具有执行dbe_pldebugger下调试函数的权限。
gs_role_public_dblink_drop	具有执行删除public database link对象的权限。
gs_role_public_dblink_alter	具有执行修改public database link对象的权限。
gs_role_seclabel	具有创建、删除和应用安全标签的权限。
gs_role_public_synonym_create	具有创建public同义词的权限。
gs_role_public_synonym_drop	具有删除public同义词的权限。
gs_role_pdb_create	具有创建pluggable database（PDB）的权限。

- 系统权限配置
  - **默认权限机制配置方法**
    - **初始用户**

      数据库安装过程中自动生成的账户称为初始用户。初始用户也是系统管理员、监控管理员、运维管理员和安全策略管理员，拥有系统的最高权限，能够执行所有的操作。如果安装时不指定初始用户名称则该账户与进行数据库安装的操作系统用户同名。如果在安装时不指定初始用户的密码，安装完成后密码为空，在执行其他操作前需要通过gsql客户端修改初始用户的密码。如果初始用户密码为空，则除修改密码外无法执行其他SQL操作以及升级、扩容和节点替换等操作。

      初始用户会绕过所有权限检查。建议仅将此初始用户作为DBA管理用途，而非业务用途。

版权所有 © 华为云计算技术有限公司

- **系统管理员**

```
gaussdb=#CREATE USER u_sysadmin WITH SYSADMIN password '********';

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_sysadmin01 SYSADMIN;
```

- **监控管理员**

```
gaussdb=#CREATE USER u_monadmin WITH MONADMIN password '********';

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_monadmin01 MONADMIN;
```

- **运维管理员**

```
gaussdb=#CREATE USER u_opradmin WITH OPRADMIN password "xxxxxxxxx";

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_opradmin01 OPRADMIN;
```

- **安全策略管理员**

```
gaussdb=#CREATE USER u_poladmin WITH POLADMIN password "xxxxxxxxx";

--或者使用如下SQL，效果一样，需要该用户已存在。
gaussdb=#ALTER USER u_poladmin01 POLADMIN;
```

– **三权分立机制配置方式**

此模式需要设置guc参数“enableSeparationOfDuty”的值为“on”，该参数为POSTMASTER类型参数，修改完之后需要重启数据库。

```
gs_guc set -Z datanode -N all -I all -c "enableSeparationOfDuty=on"
gs_om -t stop
gs_om -t start
```

创建和配置相应的用户权限的语法和默认权限一致。

- 角色权限配置

```
--创建数据库test。
gaussdb=#CREATE DATABASE test;
--创建角色role1，创建用户user1。
gaussdb=#CREATE ROLE role1 PASSWORD '********';
gaussdb=#CREATE USER user1 PASSWORD '********';
--赋予CREATE ANY TABLE权限角色role1。
gaussdb=#GRANT CREATE ON DATABASE test TO role1;

--将角色role1赋予给用户user1,则用户user1属于组role1,继承role1的相应权限可以在test数据库中创建模式。
gaussdb=#GRANT role1 TO user1;

--查询用户和角色信息。
gaussdb=#\du role1|user1;
            List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 role1     | Cannot login | {}
 user1     |              | {role1}
```

实践效果

无。

版权所有 © 华为云计算技术有限公司

7 数据倾斜查询最佳实践（分布式）

7.1 快速定位查询存储倾斜的表

目前提供的倾斜查询接口有函数：table_distribution(schemaname text, tablename text)、table_distribution()以及视图PGXC_GET_TABLE_SKEWNESS，客户可以根据自身业务情况来选择使用。相关函数和视图请参考《开发指南》中相关章节。

场景一：磁盘满后快速定位存储倾斜的表

首先，通过pg_stat_get_last_data_changed_time(oid)函数查询出近期发生过数据变更的表，鉴于表的最后修改时间只在进行IUD操作的CN记录，要查询库内1天（间隔可在函数中调整）内被修改的所有表，可以使用如下封装函数：

```
gaussdb=# CREATE OR REPLACE FUNCTION get_last_changed_table(OUT schemaname text, OUT relname
text)
RETURNS setof record
AS $$
DECLARE
 row_data record;
 row_name record;
 query_str text;
 query_str_nodes text;
 BEGIN
  query_str_nodes := 'SELECT node_name FROM pgxc_node where node_type = ''C''';
  FOR row_name IN EXECUTE(query_str_nodes) LOOP
   query_str := 'EXECUTE DIRECT ON (' || row_name.node_name || ') ''SELECT b.nspname,a.relname FROM
pg_class a INNER JOIN pg_namespace b on a.relnamespace = b.oid where
pg_stat_get_last_data_changed_time(a.oid) BETWEEN current_timestamp - 1 AND current_timestamp;''';
   FOR row_data IN EXECUTE(query_str) LOOP
    schemaname = row_data.nspname;
    relname = row_data.relname;
    return next;
   END LOOP;
  END LOOP;
  return;
 END; $$
LANGUAGE 'plpgsql';
```

然后，通过table_distribution(schemaname text, tablename text)查询出表在各个DN占用的存储空间。

```
gaussdb=# SELECT table_distribution(schemaname,relname) FROM get_last_changed_table();
```


版权所有 © 华为云计算技术有限公司

场景二：常规数据倾斜巡检

- 在库中表个数少于1W的场景，直接使用倾斜视图查询当前库内所有表的数据倾斜情况。
```
gaussdb=#SELECT * FROM pgxc_get_table_skewness ORDER BY totalsize DESC;
```
- 在库中表个数非常多（至少大于1W）的场景，因PGXC_GET_TABLE_SKEWNESS涉及全库查并计算非常全面的倾斜字段，所以可能会花费比较长的时间（小时级），请根据PGXC_GET_TABLE_SKEWNESS视图定义，直接使用table_distribution()函数自定义输出，减少输出列进行计算优化，例如：
```
gaussdb=#SELECT schemaname,tablename,max(dnsize) AS maxsize, min(dnsize) AS minsize
FROM pg_catalog.pg_class c
INNER JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
INNER JOIN pg_catalog.table_distribution() s ON s.schemaname = n.nspname AND s.tablename =
c.relname
INNER JOIN pg_catalog.pgxc_class x ON c.oid = x.pcrelid AND x.pclocatortype = 'H'
GROUP BY schemaname,tablename;
```

8 存储过程最佳实践

8.1 存储过程最佳实践（分布式）

商业规则和业务逻辑可以通过程序存储在GaussDB中，这个程序就是存储过程。

存储过程是SQL、PL/SQL和Java语句的组合。存储过程使执行商业规则的代码可以从应用程序中移动到数据库。从而，代码存储一次能够被多个程序使用。

存储过程的基本使用方法请参见《开发指南》中“存储过程”章节。

8.1.1 权限控制

存储过程默认具有SECURITYINVOKER权限。如果希望将默认行为改为SECURITYDEFINER权限，需要设置GUC参数behavior_compat_options='plsql_security_definer'。权限详情请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE FUNCTION”章节。

选择不当的权限模式可能导致越权访问敏感数据，或进行未授权的资源操作。因此，应谨慎选择和配置权限模式，以确保系统的安全性。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
--创建两个不同的用户。
gaussdb=# CREATE USER test_user1 PASSWORD '********';
CREATE ROLE
gaussdb=# CREATE USER test_user2 PASSWORD '********';
CREATE ROLE
--设置两个用户在SCHEMA best_practices_for_procedure上的权限。
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user1;
GRANT
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user2;
GRANT
--切换用户test_user1,创建表和存储过程。
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> CREATE TABLE best_practices_for_procedure.user1_tb (a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
```


版权所有 © 华为云计算技术有限公司

```
CREATE PROCEDURE
--切换test_user2执行test_user1创建的存储过程，执行报错，对表user1_tb没有权限，因为执行存储过程默认使用调用者的权限。
gaussdb=> SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
ERROR:  Permission denied for relation user1_tb.
DETAIL:  N/A.
CONTEXT:  SQL statement "insert into best_practices_for_procedure.user1_tb values(1,1)"
PL/pgSQL function best_practices_for_procedure.user1_proc() line 3 at SQL statement
--设置guc将创建存储过程默认使用创建者权限。
gaussdb=> SET behavior_compat_options='plsql_security_definer';
SET
--切换用户test_user1重新创建存储过程。
gaussdb=> SET SESSION AUTHORIZATION test_user1 password 'Gauss@123';
SET
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
--切换用户test_user2执行存储过程，执行成功。
gaussdb=> SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
 proc_user1
------------

(1 row)

--切换用户test_user1查看表中内容。
gaussdb=> SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> SELECT * FROM best_practices_for_procedure.user1_tb;
 a | b
---+---
 1 | 1
(1 row)

--清理环境。
gaussdb=> RESET behavior_compat_options;
RESET
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.user1_tb
drop cascades to function best_practices_for_procedure.user1_proc()
DROP SCHEMA
gaussdb=# DROP USER test_user1;
DROP ROLE
gaussdb=# DROP USER test_user2;
DROP ROLE
```

8.1.2 命名规范

不规范的存储过程和变量命名可能会对系统使用产生负面影响。

- 存储过程、变量以及类型名称的最大长度不得超过63个字符。如果超出此限制，名称将被自动截断至63个字符。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--创建长度为66字符的存储过程名，提示被截断为63个字符长度。
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011() AS
```

```
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

--创建长度为66字符的变量名，提示被截断为63个字符长度。
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.proc1(abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456
7891011 int) AS
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891()
drop cascades to function best_practices_for_procedure.proc1(integer)
DROP SCHEMA
```

- 在创建存储过程时，避免在不同变量作用域内使用相同名称的变量和类型，变量作用域的具体内容请参见《开发指南》中“存储过程 > 基本语句 > 定义变量”章节中“变量作用域”。当在不同变量作用域内使用相同名称的变量和类型，可能会降低存储过程的可读性，并增加其维护难度。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--创建存储过程，在不同变量作用域中创建相同变量名，并赋值。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc1() AS
    name varchar2(10) := 'outer';
    age int := 2025;
BEGIN
    DECLARE
    name varchar2(10) := 'inner'; --仅做示例，不推荐使用。
    age int := 2024; --仅做示例，不推荐使用。
    BEGIN
        dbe_output.print_line('inner name =' || name);
        dbe_output.print_line('inner age =' || age);
    END;
    dbe_output.print_line('outer name =' || name);
    dbe_output.print_line('outer age =' || age);
END;
/
CREATE PROCEDURE

--执行存储过程，相同变量名在不同作用域中其实为不同变量。
gaussdb=# CALL best_practices_for_procedure.proc1();
inner name =inner
inner age =2024
outer name =outer
outer age =2025
 proc1
-------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- 避免在存储过程的名称、内部变量名和数据类型名中使用SQL关键字，以确保在所有场景下都能正常运行。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# cREATE OR REPLACE PROCEDURE best_practices_for_procedure."as"() AS --仅做示例，不推
荐使用。
BEGIN
    NULL;
END;
/
CREATE PROCEDURE

--直接调用会报错。
gaussdb=# CALL as();
ERROR:  syntax error at or near "as"
LINE 1: call as();
            ^
gaussdb=# CALL best_practices_for_procedure."as"();
 as
----

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure."as"()
DROP SCHEMA
```

- 创建存储过程时，请勿与系统函数同名，以避免混淆。如果必须使用相同名称，请在调用时明确指定SCHEMA。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--在schema下创建和系统函数abs重名的abs函数。仅做示例，不推荐使用。
gaussdb=# CREATE OR REPLACE FUNCTION best_practices_for_procedure.abs(a int) RETURN int AS
BEGIN
    dbe_output.print_line('my abs funciton.');
    RETURN abs(a);
END;
/
CREATE FUNCTION

--调用存储过程，若不加schema则会调用系统函数abs。
gaussdb=# CALL abs(-1);
 abs
-----
   1
(1 row)


--建议使用时添加schema。
gaussdb=# CALL best_practices_for_procedure.abs(-1);
my abs funciton.
 abs
-----
   1
(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.abs(integer)
DROP SCHEMA
```

8.1.3 访问对象

未指定SCHEMA的存储过程将依据SEARCH_PATH的顺序查找对象，可能导致访问到非预期对象。如果不同模式中存在同名表、存储过程以及其他数据库对象，未明确指定SCHEMA可能引发意外结果。因此，建议在存储过程访问数据对象时，始终明确指定SCHEMA。

示例：

```
--创建两个不同的schema。
gaussdb=# CREATE SCHEMA best_practices_for_procedure1;
CREATE SCHEMA
gaussdb=# CREATE SCHEMA best_practices_for_procedure2;
CREATE SCHEMA

--在两个不同的schema下创建相同的存储过程。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure1.proc1() as
BEGIN
    dbe_output.print_line('in schema best_practices_for_procedure1');
END;
/
CREATE PROCEDURE

gaussdb=# CREATE OR REPLACE procedure best_practices_for_procedure2.proc1() as
BEGIN
    dbe_output.print_line('in schema best_practices_for_procedure2');
END;
/
CREATE PROCEDURE

--在不同的search_path下调用相同的存储过程可能存在差异。
gaussdb=# SET search_path TO best_practices_for_procedure1, best_practices_for_procedure2;
SET
gaussdb=# CALL proc1();
in schema best_practices_for_procedure1
 proc1
-------

(1 row)

gaussdb=# RESET search_path;
RESET
gaussdb=# SET search_path TO best_practices_for_procedure2, best_practices_for_procedure1;
SET
gaussdb=# CALL proc1();
in schema best_practices_for_procedure2
 proc1
-------

(1 row)

gaussdb=# RESET search_path;
RESET

gaussdb=# DROP SCHEMA best_practices_for_procedure1 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure1.proc1()
DROP SCHEMA

gaussdb=# DROP SCHEMA best_practices_for_procedure2 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure2.proc1()
DROP SCHEMA
```

8.1.4 语句功能

8.1.4.1 PACKAGE 变量

PACKAGE变量是在PACKAGE内定义的全局变量，其生命周期覆盖整个数据库会话（SESSION）。不当使用可能引发以下问题：

- 对具备PACKAGE访问权限的用户完全透明，可能导致变量在多个存储过程间共享并意外被修改。
- 由于PACKAGE变量的生命周期为SESSION级别，不当操作可能造成数据残留，影响其他存储过程。

- 大量PACKAGE变量在SESSION中缓存可能占用大量内存。

因此，建议谨慎使用PACKAGE变量，并确保其访问和生命周期得到合理管理。

```
--创建ORA兼容数据库。
gaussdb=# CREATE DATABASE db_test DBCOMPATIBILITY 'ORA';

--切换至ORA兼容数据库。
gaussdb=# \c db_test
db_test=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

db_test=# CREATE OR REPLACE PACKAGE best_practices_for_procedure.pkg1 AS
    id int;
    name varchar2(20);
    arg int;
    procedure p1();
END pkg1;
/
CREATE PACKAGE

db_test=# CREATE OR REPLACE PACKAGE BODY best_practices_for_procedure.pkg1 AS
    procedure p1() as
    BEGIN
        id := 1;
        name := 'huawei';
    arg := 20;
    END;
END pkg1;
/
CREATE PACKAGE BODY

--创建存过修改package变量。
db_test=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro1 () AS
BEGIN
    best_practices_for_procedure.pkg1.id := 2;
    best_practices_for_procedure.pkg1.name := 'gaussdb';
    best_practices_for_procedure.pkg1.arg := 18;
END;
/
CREATE PROCEDURE

--修改package变量值。
db_test=# CALL best_practices_for_procedure.pro1();
 pro1
------

(1 row)

--在实际使用时发现参数已经被修改过。
db_test=# DECLARE
BEGIN
    dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
    best_practices_for_procedure.pkg1.p1();
    dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
END;
/
id = 2 name = gaussdb arg = 18
id = 1 name = huawei arg = 20
ANONYMOUS BLOCK EXECUTE

db_test=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to package 16443
drop cascades to function best_practices_for_procedure.p1()
drop cascades to function best_practices_for_procedure.pro1()
DROP SCHEMA
```

```
db_test=#  \c postgres
gaussdb=# DROP DATABASE db_test;
```

8.1.4.2 CURSOR

在存储过程中，CURSOR是一种重要资源，不当使用可能引发以下问题：

- 未关闭的CURSOR会占用系统资源，大量未及时关闭的CURSOR会严重影响数据库内存和性能，特别是在高并发或循环操作中。

因此，建议在存储过程中，使用CURSOR后立即关闭。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# CREATE TABLE best_practices_for_procedure.tb1 (a int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

gaussdb=# INSERT INTO best_practices_for_procedure.tb1 VALUES (1),(2),(3);
INSERT 0 3

--创建使用cursor的存储过程。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_cursor () AS
    my_cursor CURSOR FOR SELECT *FROM best_practices_for_procedure.tb1;
    a int;
BEGIN
    OPEN my_cursor;
    FETCH my_cursor INTO a;
    CLOSE my_cursor; -- 及时关闭cursor。
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_cursor();
 pro_cursor
------------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_cursor()
DROP SCHEMA
```

8.1.4.3 兼容性功能

由于数据库的兼容性差异，在不同兼容性设置或GUC参数下，存储过程的行为可能不一致。用户在使用这些兼容性功能时，应谨慎操作。例如：

- 由于不同兼容性的原因，带出参的FUNCTION在某些情况下可能忽略出参值。开启GUC参数 behavior_compat_options='proc_outparam_override' 后，部分场景可以确保出参值和返回值的正确返回。然而，由于此功能在不同兼容性设置下的行为存在差异，建议避免使用带出参的FUNCTION，改用带出参的PROCEDURE。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--创建带出参的function。
gaussdb=# CREATE OR REPLACE FUNCTION best_practices_for_procedure.func (a out int, b out int)
RETURN int AS --仅做示例，不推荐使用。
    c int;
BEGIN
    a := 1;
```

```
    b := 2;
    c := 3;
    RETURN c;
END;
/
CREATE FUNCTION

--调用带出参的function，发现参数a，b并未赋值。
gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    c := best_practices_for_procedure.func(a, b);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a :=  b :=  c := 3
ANONYMOUS BLOCK EXECUTE

--设置GUC参数。
gaussdb=# SET behavior_compat_options='proc_outparam_override';
SET

--再次调用带出参的function，参数a，b，c均被赋值。
gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    c := best_practices_for_procedure.func(a, b);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE

--推荐使用带出参的存储过程来替代带出参的函数，可将上述函数改写为下面的存储过程。
gaussdb=# RESET behavior_compat_options;
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc (a OUT int, b OUT
int, c OUT int) AS
BEGIN
    a := 1;
    b := 2;
    c := 3;
END;
/
CREATE PROCEDURE

gaussdb=# DECLARE
    a int;
    b int;
    c int;
BEGIN
    best_practices_for_procedure.proc(a, b, c);
    dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
END;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function best_practices_for_procedure.func()
drop cascades to function best_practices_for_procedure.proc()
DROP SCHEMA
```

- 在动态语句中，如果占位符名称重复，不同数据库的兼容性设置可能导致其绑定到不同的变量，从而影响预期行为。开启GUC参数

behavior_compat_options='dynamic_sql_compat' 后，可以使用同名占位符绑定不同变量。然而，由于此功能在不同兼容性设置下的行为存在差异，建议避免使用同名占位符。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=# CREATE TABLE best_practices_for_procedure.tb1 (a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

--创建使用动态语句的存储过程，使用相同占位符绑定相同变量。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_dynexecute() AS
    a int := 1;
    b int := 2;
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO best_practices_for_procedure.tb1 VALUES(:1, :1),(:2, :2);' USING
IN a, IN b;
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

--查看表发现相同占位符绑定的是相同的变量。
gaussdb=# SELECT * FROM best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 1
 2 | 2
(2 rows)

--设置GUC参数。
gaussdb=# SET behavior_compat_options='dynamic_sql_compat';
SET
gaussdb=# TRUNCATE TABLE best_practices_for_procedure.tb1;
TRUNCATE TABLE

--创建使用动态语句的存储过程，使用相同占位符绑定不同变量。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.pro_dynexecute() AS
    a int := 1;
    b int := 2;
    c int := 3;
    d int := 4;
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO best_practices_for_procedure.tb1 VALUES(:1, :1),(:2, :2);' USING
IN a, IN b, IN c, IN d;
END;
/
CREATE PROCEDURE

gaussdb=# CALL best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

--设置GUC参数后调用函数，相同占位符可以绑定不同变量。
gaussdb=# SELECT * FROM best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 2
 3 | 4
(2 rows)
```

```
gaussdb=# RESET behavior_compat_options;
RESET

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_dynexecute()
DROP SCHEMA
```

8.1.4.4 异常处理

在存储过程中使用EXCEPTION处理机制可以提高代码的容错性，但频繁地捕获和处理异常可能会导致性能下降。每次异常处理都涉及上下文的创建和销毁，这会消耗额外的内存和资源。此外，由于异常被捕获，日志中不会记录错误信息，从而增加了问题定位的难度。

建议在必要时才使用EXCEPTION处理机制，并确保传递充足的上下文信息，以便于问题的定位和解决。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create unique index id1 on best_practices_for_procedure.tb1(id);
CREATE INDEX
--创建带有exception的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(oi_flag OUT int, os_msg OUT
varchar) as
begin
oi_flag := 0;
os_msg := 'insert into tb1 some data.';
for i in 1..10 loop
if i = 5 then
insert into best_practices_for_procedure.tb1 values(i - 1, 'name'|| i - 1);--人为制造报错。
end if;
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
exception when others then
oi_flag := 1;
os_msg := SQLERRM; --将报错信息传递出去。
end;
/
CREATE PROCEDURE
gaussdb=# declare
oi_flag int;
os_msg varchar(1000);
begin
best_practices_for_procedure.proc1(oi_flag, os_msg);
if oi_flag = 1 then
dbe_output.print_line('Exception for ' || os_msg);
end if;
end;
/
Exception for Duplicate key value violates unique constraint "id1".
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

8.1.4.5 自定义类型

存储过程中的自定义类型变量不支持下推，需要在下推场景使用时，用变量承接自定义类型的元素。

示例如下：

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# SET CURRENT_SCHEMA=best_practices_for_procedure;
SET
gaussdb=#
gaussdb=# CREATE TABLE tb1(c1 INT, c2 VARCHAR(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'c1' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# INSERT INTO tb1 VALUES(1, 'a'),(2,'b'), (3, 'c');
INSERT 0 3
gaussdb=# -- SELECT语句下推执行
gaussdb=# EXPLAIN SELECT c1 FROM tb1 WHERE c2 = 'a';
                    QUERY PLAN
-----------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
gaussdb=# -- 变量承接自定义类型元素
gaussdb=# CREATE OR REPLACE PROCEDURE proc1() AS
gaussdb$# TYPE ta IS VARRAY(10) OF VARCHAR(30);
gaussdb$# v ta := ta();
gaussdb$# b VARCHAR;
gaussdb$# a INT;
gaussdb$# BEGIN
gaussdb$#     v(1) := 'a';
gaussdb$#     v(2) := 'b';
gaussdb$#     FOR i IN 1..v.count LOOP
gaussdb$#         b := v(i);  -- 变量承接自定义类型的元素
gaussdb$#         SELECT c1 INTO a FROM tb1 WHERE c2 = b; -- 执行成功
gaussdb$#     END LOOP;
gaussdb$# END;
gaussdb$# /
CREATE PROCEDURE
gaussdb=# CALL proc1();
 proc1
-------

(1 row)
gaussdb=#
gaussdb=# -- 自定义类型下推场景
gaussdb=# CREATE OR REPLACE PROCEDURE proc2() AS
gaussdb$# TYPE ta IS VARRAY(10) OF VARCHAR(30);
gaussdb$# v ta := ta();
gaussdb$# b VARCHAR;
gaussdb$# a INT;
gaussdb$# BEGIN
gaussdb$#     v(1) := 'a';
gaussdb$#     v(2) := 'b';
gaussdb$#     FOR i IN 1..v.count LOOP
gaussdb$#         SELECT c1 INTO a FROM tb1 WHERE c2 = v(1); -- 自定义类型不支持下推，执行报错
gaussdb$#     END LOOP;
gaussdb$# END;
gaussdb$# /
CREATE PROCEDURE
gaussdb=# CALL proc2();
ERROR:  Function v(integer) does not exist.
LINE 1: SELECT c1        FROM tb1 WHERE c2 = v(1)
                                        ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
QUERY:  SELECT c1        FROM tb1 WHERE c2 = v(1)
CONTEXT:  PL/pgSQL function proc2() line 10 at SQL statement
gaussdb=#
gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to table tb1
drop cascades to function proc1()
```

```
drop cascades to function proc2()
DROP SCHEMA
```

8.1.5 事务管理

8.1.5.1 事务

存储过程可以通过使用SAVEPOINT以及COMMIT/ROLLBACK来进行事务管理，如果使用不当，可能会引发以下问题：

- 每次在事务中创建SAVEPOINT都会分配资源，若不及时释放，资源占用将逐渐累积。
- 事务的COMMIT和ROLLBACK操作需要同步数据库的元数据和日志，频繁执行可能增加I/O开销，从而影响性能。

建议：

- 在使用完SAVEPOINT后，应及时使用RELEASE SAVEPOINT来释放资源。
- 避免在循环中创建SAVEPOINT，因为同名的SAVEPOINT不会覆盖，而是会重新创建，这可能导致资源迅速累积。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
--创建使用savepoint的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
savepoint sp1; --不要在循环中使用SAVEPOINT。
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
release savepoint sp1; --释放savepoint。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- 避免频繁使用COMMIT/ROLLBACK。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
commit; --执行完循环之后commit，而不是在循环内重复commit。
end;
/
```

```
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

8.1.5.2 自治事务

自治事务指的是在存储过程中启动一个独立的事务，该事务与主事务相互独立，能够在主事务提交或回滚后继续其操作。通过启动新的数据库会话（SESSION）来执行存储过程，自治事务可能会增加系统资源的使用，包括内存、CPU和数据库连接等。

建议将自治事务主要用于记录业务日志，而不应将其作为业务流程的入口或核心环节。应尽量避免频繁使用自治事务，以减少对系统资源的消耗。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.log_table(log_time timestamptz, message text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'log_time' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create table best_practices_for_procedure.work_table(company text, balance float);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'company' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.work_table values('huawei', 100000);
INSERT 0 1
--创建自治事务存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc_auto(log_time timestamptz,
message text) as
PRAGMA AUTONOMOUS_TRANSACTION;
begin
insert into best_practices_for_procedure.log_table values (log_time, message); --只记录日志信息。
end;
/
CREATE PROCEDURE
--在存储过程内调用自治事务。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(companys text, turnover float) as
message text;
begin
    update best_practices_for_procedure.work_table set balance = balance + turnover where company =
companys;
    message := 'Company turnover ' || turnover;
    best_practices_for_procedure.proc_auto(current_timestamp, message);
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1('huawei', 1000);
 proc1
-------

(1 row)

gaussdb=# select * from best_practices_for_procedure.log_table;
           log_time           |        message
------------------------------+-----------------------
 2024-11-25 15:23:27.202458+08 | Company turnover 1000
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
```

```
DETAIL:  drop cascades to table best_practices_for_procedure.log_table
drop cascades to table best_practices_for_procedure.work_table
drop cascades to function best_practices_for_procedure.proc_auto(timestamp with time zone,text)
drop cascades to function best_practices_for_procedure.proc1(text,double precision)
DROP SCHEMA
```

8.1.6 其他

8.1.6.1 DDL

数据定义语言（DDL）操作（如CREATE、ALTER、DROP）通常会加锁，以确保变更的原子性和一致性。在高并发环境中，DDL操作可能引发锁冲突或导致长时间阻塞，从而影响其他业务操作的正常执行。

建议在执行DDL变更时，暂停相关业务操作，以避免对系统性能和稳定性产生不利影响。

8.1.6.2 复杂依赖

如果存储过程或PACKAGE之间存在复杂的依赖关系，可能会在创建时遇到依赖对象尚未创建或初始化的情况，从而导致存储过程编译失败。此外，当某个对象被修改或重建时，直接或间接依赖该对象的其他存储过程和PACKAGE也会失效，需要重新编译，这会影响系统性能。

建议避免在存储过程和PACKAGE之间建立复杂的依赖关系，以提高系统的稳定性和性能。

```
--创建ORA兼容数据库。
CREATE DATABASE db_test DBCOMPATIBILITY 'ORA';

--切换至ORA兼容数据库。
gaussdb=# \c db_test
db_test=# create schema best_practices_for_procedure;
CREATE SCHEMA

--创建依赖pkg2的pkg1，会报错。
db_test=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
ERROR:  "best_practices_for_procedure.pkg2.a" is not a known variable.
LINE 3: best_practices_for_procedure.pkg2.a := 100;
        ^
QUERY:   DECLARE
begin
best_practices_for_procedure.pkg2.a := 100;
end

--只能先创建被依赖的pkg2，再创建pkg1。
db_test=# create or replace package best_practices_for_procedure.pkg2 as
a int;
procedure p1();
end pkg2;
/
```

```
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg2 as
procedure p1() as
begin
null;
end;
end pkg2;
/
CREATE PACKAGE BODY

db_test=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE

db_test=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
CREATE PACKAGE BODY

db_test=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to package 16526
drop cascades to function best_practices_for_procedure.p1()
drop cascades to package 16524
drop cascades to function best_practices_for_procedure.p1()
DROP SCHEMA

templatea=# \c postgres
gaussdb=# DROP DATABASE db_test;
```

8.1.6.3 IMMUTABLE 和 SHIPPABLE

IMMUTABLE是存储过程的一个属性，用于声明该存储过程的结果仅依赖于输入参数，而不依赖于数据库的当前状态。在某些场景下，使用 IMMUTABLE 属性的存储过程可能会被优化为仅执行一次，如果使用不当，可能会造成不符合预期的结果。

SHIPPABLE是存储过程的另一个属性，用于表示该存储过程是否可以下推到 DN（数据节点）执行。如果下推的存储过程访问表或数据库状态，可能会导致数据不一致。

建议在使用 IMMUTABLE 和 SHIPPABLE 属性的存储过程和函数时，避免访问表或数据库信息，以确保其行为符合预期并维护数据一致性。属性详情请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE FUNCTION”章节。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(a int, b int) immutable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); --仅做示例，不推荐使用。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1(2, 5);
ERROR:  INSERT is not allowed in a non-volatile function
CONTEXT:  SQL statement "insert into best_practices_for_procedure.tb1 values(a, b)"
PL/pgSQL function best_practices_for_procedure.proc1(integer,integer) line 3 at SQL statement
gaussdb=# create or replace function best_practices_for_procedure.func1(a int, b int) return int immutable
```

```
as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func1(2, 5);
 func1
-------
    10
(1 row)

gaussdb=# create or replace procedure best_practices_for_procedure.proc2(a int, b int) shippable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); --仅做示例，不推荐使用。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc2(2, 5);
 proc2
-------

(1 row)

gaussdb=# create or replace function best_practices_for_procedure.func2(a int, b int) return int shippable as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func2(2, 5);
 func2
-------
    10
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 5 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1(integer,integer)
drop cascades to function best_practices_for_procedure.func1(integer,integer)
drop cascades to function best_practices_for_procedure.proc2(integer,integer)
drop cascades to function best_practices_for_procedure.func2(integer,integer)
DROP SCHEMA
```

8.2 存储过程最佳实践（集中式）

商业规则和业务逻辑可以通过程序存储在GaussDB中，这个程序就是存储过程。

存储过程是SQL和PL/SQL的组合。存储过程使执行商业规则的代码可以从应用程序中移动到数据库。从而，代码存储一次能够被多个程序使用。

存储过程的基本使用方法请参见《开发指南》中“存储过程”章节。

8.2.1 权限控制

存储过程默认具有SECURITYINVOKER权限。如果希望将默认行为改为SECURITYDEFINER权限，需要设置GUC参数behavior_compat_options='plsql_security_definer'。权限详情请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE FUNCTION”章节。

选择不当的权限模式可能导致越权访问敏感数据，或进行未授权的资源操作。因此，应谨慎选择和配置权限模式，以确保系统的安全性。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA
--创建两个不同的用户。
gaussdb=# CREATE USER test_user1 PASSWORD '********';
CREATE ROLE
gaussdb=# CREATE USER test_user2 PASSWORD '********';
CREATE ROLE
--设置两个用户在SCHEMA best_practices_for_procedure上的权限。
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user1;
GRANT
gaussdb=# GRANT usage, create ON SCHEMA best_practices_for_procedure TO test_user2;
GRANT
--切换用户test_user1,创建表和存储过程。
gaussdb=# SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> CREATE TABLE best_practices_for_procedure.user1_tb (a int, b int);
CREATE TABLE
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
--切换test_user2执行test_user1创建的存储过程，执行报错，对表user1_tb没有权限，因为执行存储过程默认使用调用者的权限。
gaussdb=> SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
ERROR:  Permission denied for relation user1_tb.
DETAIL:  N/A.
CONTEXT:  SQL statement "insert into best_practices_for_procedure.user1_tb values(1,1)"
PL/pgSQL function best_practices_for_procedure.user1_proc() line 3 at SQL statement
--设置GUC将创建存储过程默认使用创建者权限。
gaussdb=> SET behavior_compat_options='plsql_security_definer';
SET
--切换用户test_user1重新创建存储过程。
gaussdb=> SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> CREATE OR REPLACE PROCEDURE best_practices_for_procedure.user1_proc() AS
  BEGIN
    INSERT INTO best_practices_for_procedure.user1_tb VALUES(1,1);
  END;
/
CREATE PROCEDURE
--切换用户test_user2执行存储过程，执行成功。
gaussdb=> SET SESSION AUTHORIZATION test_user2 PASSWORD '********';
SET
gaussdb=> CALL best_practices_for_procedure.user1_proc();
 proc_user1
------------

(1 row)

--切换用户test_user1查看表中内容。
gaussdb=> SET SESSION AUTHORIZATION test_user1 PASSWORD '********';
SET
gaussdb=> SELECT * FROM best_practices_for_procedure.user1_tb;
 a | b
---+---
 1 | 1
(1 row)

--清理环境。
gaussdb=> RESET behavior_compat_options;
RESET
gaussdb=> RESET SESSION AUTHORIZATION;
RESET
gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
```

```
DETAIL:  drop cascades to table best_practices_for_procedure.user1_tb
drop cascades to function best_practices_for_procedure.user1_proc()
DROP SCHEMA
gaussdb=# DROP USER test_user1;
DROP ROLE
gaussdb=# DROP USER test_user2;
DROP ROLE
```

8.2.2 命名规范

不规范的存储过程和变量命名可能会对系统使用产生负面影响。

- 存储过程、变量以及类型名称的最大长度不得超过63个字符。如果超出此限制，名称将被自动截断至63个字符。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--创建长度为66字符的存储过程名，提示被截断为63个字符长度。
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456789101
1() AS
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

--创建长度为66字符的变量名，提示被截断为63个字符长度。
gaussdb=# CREATE OR REPLACE PROCEDURE
best_practices_for_procedure.proc1(abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456
7891011 int) as
BEGIN
    NULL;
END;
/
NOTICE:  identifier "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891011" will
be truncated to "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891"
CREATE PROCEDURE

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function
best_practices_for_procedure.abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz01234567891()
drop cascades to function best_practices_for_procedure.proc1(integer)
DROP SCHEMA
```

- 在创建存储过程时，避免在不同变量作用域内使用相同名称的变量和类型，变量作用域的具体内容请参见《开发指南》中“存储过程 > 基本语句 > 定义变量”章节中“变量作用域”。在不同变量作用域内使用相同名称的变量和类型，可能会降低存储过程的可读性，并增加其维护难度。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--创建存储过程，在不同变量作用域中创建相同变量名，并赋值。
gaussdb=# CREATE OR REPLACE PROCEDURE best_practices_for_procedure.proc1() AS
    name varchar2(10) := 'outer';
    age int := 2025;
BEGIN
    DECLARE
        name varchar2(10) := 'inner'; --仅做示例，不推荐使用。
        age int := 2024; --仅做示例，不推荐使用。
    BEGIN
        dbe_output.print_line('inner name =' || name);
        dbe_output.print_line('inner age =' || age);
    END;
    dbe_output.print_line('outer name =' || name);
```

```
    dbe_output.print_line('outer age =' || age);
END;
/
CREATE PROCEDURE

--执行存储过程，相同变量名在不同作用域中其实为不同变量。
gaussdb=# CALL best_practices_for_procedure.proc1();
inner name =inner
inner age =2024
outer name =outer
outer age =2025
 proc1
-------

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure cascade;
NOTICE:  drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- 避免在存储过程的名称、内部变量名和数据类型名中使用SQL关键字，以确保在所有场景下都能正常运行。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

gaussdb=#
CREATE OR REPLACE PROCEDURE best_practices_for_procedure."as"() AS --仅做示例，不推荐使用。
BEGIN
    NULL;
END;
/
CREATE PROCEDURE

--直接调用会报错。
gaussdb=# CALL as();
ERROR:  syntax error at or near "as"
LINE 1: call as();
             ^
gaussdb=# CALL best_practices_for_procedure."as"();
 as
----

(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to function best_practices_for_procedure."as"()
DROP SCHEMA
```

- 创建存储过程时，请勿与系统函数同名，以避免混淆。如果必须使用相同名称，请在调用时明确指定SCHEMA。

```
gaussdb=# CREATE SCHEMA best_practices_for_procedure;
CREATE SCHEMA

--在schema下创建和系统函数abs重名的abs函数。仅做示例，不推荐使用。
gaussdb=#
CREATE OR REPLACE FUNCTION best_practices_for_procedure.abs(a int) RETURN int AS
BEGIN
    dbe_output.print_line('my abs funciton.');
    RETURN abs(a);
END;
/
CREATE FUNCTION

--调用存储过程，若不加Schema则会调用系统函数abs。
gaussdb=# CALL abs(-1);
 abs
-----
   1
(1 row)
```

```
--建议使用时添加Schema。
gaussdb=# CALL best_practices_for_procedure.abs(-1);
my abs funciton.
 abs
-----
   1
(1 row)

gaussdb=# DROP SCHEMA best_practices_for_procedure CASCADE;
NOTICE:  drop cascades to function best_practices_for_procedure.abs(integer)
DROP SCHEMA
```

8.2.3 访问对象

未指定SCHEMA的存储过程将依据SEARCH_PATH的顺序查找对象，可能导致访问到非预期对象。如果不同模式中存在同名表、存储过程以及其他数据库对象，未明确指定SCHEMA可能引发意外结果。因此，建议在存储过程访问数据对象时，始终明确指定SCHEMA。

示例：

```
--创建两个不同的schema。
gaussdb=# create schema best_practices_for_procedure1;
CREATE SCHEMA
gaussdb=# create schema best_practices_for_procedure2;
CREATE SCHEMA
--在两个不同的schema下创建相同的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure1.proc1() as
begin
dbe_output.print_line('in schema best_practices_for_procedure1');
end;
/
CREATE PROCEDURE
gaussdb=# create or replace procedure best_practices_for_procedure2.proc1() as
begin
dbe_output.print_line('in schema best_practices_for_procedure2');
end;
/
CREATE PROCEDURE
--在不同的search_path下调用相同的存储过程可能存在差异。
gaussdb=# set search_path to best_practices_for_procedure1, best_practices_for_procedure2;
SET
gaussdb=# call proc1();
in schema best_practices_for_procedure1
 proc1
-------

(1 row)

gaussdb=# reset search_path;
RESET
gaussdb=# set search_path to best_practices_for_procedure2, best_practices_for_procedure1;
SET
gaussdb=# call proc1();
in schema best_practices_for_procedure2
 proc1
-------

(1 row)

gaussdb=# reset search_path;
RESET
gaussdb=# drop schema best_practices_for_procedure1 cascade;
NOTICE:  drop cascades to function best_practices_for_procedure1.proc1()
DROP SCHEMA
gaussdb=# drop schema best_practices_for_procedure2 cascade;
```

```
NOTICE:  drop cascades to function best_practices_for_procedure2.proc1()
DROP SCHEMA
```

8.2.4 语句功能

8.2.4.1 PACKAGE 变量

PACKAGE变量是在PACKAGE内定义的全局变量，其生命周期覆盖整个数据库会话（SESSION）。不当使用可能引发以下问题：

- 对具备PACKAGE访问权限的用户完全透明，可能导致变量在多个存储过程间共享并意外被修改。
- 由于PACKAGE变量的生命周期为SESSION级别，不当操作可能造成数据残留，影响其他存储过程。
- 大量PACKAGE变量在SESSION中缓存可能占用大量内存。

因此，建议谨慎使用PACKAGE变量，并确保其访问和生命周期得到合理管理。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
id int;
name varchar2(20);
arg int;
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
 id := 1;
 name := 'huawei';
 arg := 20;
end;
end pkg1;
/
CREATE PACKAGE BODY
--创建存过修改package变量。
gaussdb=# create or replace procedure best_practices_for_procedure.pro1 () as
begin
best_practices_for_procedure.pkg1.id := 2;
best_practices_for_procedure.pkg1.name := 'gaussdb';
best_practices_for_procedure.pkg1.arg := 18;
end;
/
CREATE PROCEDURE
--修改package变量值。
gaussdb=# call best_practices_for_procedure.pro1();
 pro1
------

(1 row)

--在实际使用时发现参数已经被修改过。
gaussdb=# declare
begin
dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
best_practices_for_procedure.pkg1.p1();
dbe_output.print_line('id = ' || best_practices_for_procedure.pkg1.id || ' name = ' ||
best_practices_for_procedure.pkg1.name || ' arg = ' || best_practices_for_procedure.pkg1.arg);
end;
/
```

```
id = 2 name = gaussdb arg = 18
id = 1 name = huawei arg = 20
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to package 16782
drop cascades to function best_practices_for_procedure.p1()
drop cascades to function best_practices_for_procedure.pro1()
DROP SCHEMA
```

8.2.4.2 CURSOR

在存储过程中，CURSOR是一种重要资源，不当使用可能引发以下问题：

- 未关闭的CURSOR会占用系统资源，大量未及时关闭的CURSOR会严重影响数据库内存和性能，特别是在高并发或循环操作中。

因此，建议在存储过程中，使用CURSOR后立即关闭。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1 (a int);
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.tb1 values (1),(2),(3);
INSERT 0 3
--创建使用cursor的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.pro_cursor () as
my_cursor cursor for select *from best_practices_for_procedure.tb1;
a int;
begin
open my_cursor;
fetch my_cursor into a;
close my_cursor; -- 及时关闭cursor。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_cursor();
 pro_cursor
------------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_cursor()
DROP SCHEMA
```

8.2.4.3 兼容性功能

由于数据库的兼容性差异，在不同兼容性设置或GUC参数下，存储过程的行为可能不一致。用户在使用这些兼容性功能时，应谨慎操作。例如：

- 由于不同兼容性的原因，带出参的FUNCTION在某些情况下可能忽略出参值。开启GUC参数 behavior_compat_options='proc_outparam_override' 后，部分场景可以确保出参值和返回值的正确返回。然而，由于此功能在不同兼容性设置下的行为存在差异，建议避免使用带出参的FUNCTION，改用带出参的PROCEDURE。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
--创建带出参的function。
gaussdb=# create or replace function best_practices_for_procedure.func (a out int, b out int) return
int as --仅做示例，不推荐使用。
c int;
begin
a := 1;
```

```
b := 2;
c := 3;
return c;
end;
/
CREATE FUNCTION
--调用带出参的function，发现参数a，b并未赋值。
gaussdb=# declare
a int;
b int;
c int;
begin
c := best_practices_for_procedure.func(a, b);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a :=  b :=  c := 3
ANONYMOUS BLOCK EXECUTE
--设置GUC参数。
gaussdb=# set behavior_compat_options='proc_outparam_override';
SET
--再次调用带出参的function，参数a，b，c均被赋值。
gaussdb=# declare
a int;
b int;
c int;
begin
c := best_practices_for_procedure.func(a, b);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE
--推荐使用带出参的存储过程来替代带出参的函数，可将上述函数改写为下面的存储过程。
gaussdb=# reset behavior_compat_options;
gaussdb=# create or replace procedure best_practices_for_procedure.proc (a out int, b out int, c out
int) as
begin
a := 1;
b := 2;
c := 3;
end;
/
CREATE PROCEDURE
gaussdb=# declare
a int;
b int;
c int;
begin
best_practices_for_procedure.proc(a, b, c);
dbe_output.print_line('a := ' || a || ' b := ' || b || ' c := ' || c);
end;
/
a := 1 b := 2 c := 3
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to function best_practices_for_procedure.func()
drop cascades to function best_practices_for_procedure.proc()
DROP SCHEMA
```

- 在动态语句中，如果占位符名称重复，不同数据库的兼容性设置可能导致其绑定到不同的变量，从而影响预期行为。开启GUC参数 behavior_compat_options='dynamic_sql_compat' 后，可以使用同名占位符绑定不同变量。然而，由于此功能在不同兼容性设置下的行为存在差异，建议避免使用同名占位符。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1 (a int, b int);
```

```
CREATE TABLE
--创建使用动态语句的存储过程，使用相同占位符绑定相同变量。
gaussdb=# create or replace procedure best_practices_for_procedure.pro_dynexecute() as
a int := 1;
b int := 2;
begin
execute immediate 'insert into best_practices_for_procedure.tb1 values(:1, :1),(:2, :2);' using in a, in b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

--查看表发现相同占位符绑定的是相同的变量。
gaussdb=# select *from best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 1
 2 | 2
(2 rows)

--设置GUC参数。
gaussdb=# set behavior_compat_options='dynamic_sql_compat';
SET
gaussdb=# truncate table best_practices_for_procedure.tb1;
TRUNCATE TABLE
--创建使用动态语句的存储过程，使用相同占位符绑定不同变量。
gaussdb=# create or replace procedure best_practices_for_procedure.pro_dynexecute() as
a int := 1;
b int := 2;
c int := 3;
d int := 4;
begin
execute immediate 'insert into best_practices_for_procedure.tb1 values(:1, :1),(:2, :2);' using in a, in b,
in c, in d;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.pro_dynexecute();
 pro_dynexecute
----------------

(1 row)

--设置guc参数后调用函数，相同占位符可以绑定不同变量。
gaussdb=# select * from best_practices_for_procedure.tb1;
 a | b
---+---
 1 | 2
 3 | 4
(2 rows)

gaussdb=# reset behavior_compat_options;
RESET
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.pro_dynexecute()
DROP SCHEMA
```

8.2.4.4 异常处理

在存储过程中使用EXCEPTION处理机制可以提高代码的容错性，但频繁地捕获和处理异常可能会导致性能下降。每次异常处理都涉及上下文的创建和销毁，这会消耗额外

的内存和资源。此外，由于异常被捕获，日志中不会记录错误信息，从而增加了问题定位的难度。

建议在必要时才使用EXCEPTION处理机制，并确保传递充足的上下文信息，以便于问题的定位和解决。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
gaussdb=# create unique index id1 on best_practices_for_procedure.tb1(id);
CREATE INDEX
--创建带有exception的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(oi_flag OUT int, os_msg OUT
varchar) as
begin
oi_flag := 0;
os_msg := 'insert into tb1 some data.';
for i in 1..10 loop
if i = 5 then
insert into best_practices_for_procedure.tb1 values(i - 1, 'name'|| i - 1);--人为制造报错。
end if;
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
exception when others then
oi_flag := 1;
os_msg := SQLERRM; --将报错信息传递出去。
end;
/
CREATE PROCEDURE
gaussdb=# declare
oi_flag int;
os_msg varchar(1000);
begin
best_practices_for_procedure.proc1(oi_flag, os_msg);
if oi_flag = 1 then
dbe_output.print_line('Exception for ' || os_msg);
end if;
end;
/
Exception for Duplicate key value violates unique constraint "id1".
ANONYMOUS BLOCK EXECUTE
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

8.2.5 事务管理

8.2.5.1 事务

存储过程可以通过使用SAVEPOINT以及COMMIT/ROLLBACK来进行事务管理，如果使用不当，可能会引发以下问题：

- 每次在事务中创建SAVEPOINT都会分配资源，若不及时释放，资源占用将逐渐累积。
- 事务的COMMIT和ROLLBACK操作需要同步数据库的元数据和日志，频繁执行可能增加I/O开销，从而影响性能。

建议：

- 在使用完SAVEPOINT后，应及时使用RELEASE SAVEPOINT来释放资源。

- 避免在循环中创建SAVEPOINT，因为同名的SAVEPOINT不会覆盖，而是会重新创建，这可能导致资源迅速累积。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
--创建使用savepoint的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
savepoint sp1; --不要在循环中使用SAVEPOINT。
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
release savepoint sp1; --释放savepoint。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

- 避免频繁使用COMMIT/ROLLBACK。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(id int, name varchar2(20));
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1() as
begin
for i in 1..10 loop
insert into best_practices_for_procedure.tb1 values(i, 'name'|| i);
end loop;
commit; --执行完循环之后commit，而不是在循环内重复commit。
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1();
 proc1
-------

(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1()
DROP SCHEMA
```

8.2.5.2 自治事务

自治事务指的是在存储过程中启动一个独立的事务，该事务与主事务相互独立，能够在主事务提交或回滚后继续其操作。通过启动新的数据库会话（SESSION）来执行存储过程，自治事务可能会增加系统资源的使用，包括内存、CPU和数据库连接等。

建议将自治事务主要用于记录业务日志，而不应将其作为业务流程的入口或核心环节。应尽量避免频繁使用自治事务，以减少对系统资源的消耗。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.log_table(log_time timestamptz, message text);
```

```
CREATE TABLE
gaussdb=# create table best_practices_for_procedure.work_table(company text, balance float);
CREATE TABLE
gaussdb=# insert into best_practices_for_procedure.work_table values('huawei', 100000);
INSERT 0 1
--创建包含自治事务的存储过程。
gaussdb=# create or replace procedure best_practices_for_procedure.proc_auto(log_time timestamptz,
message text) as
PRAGMA AUTONOMOUS_TRANSACTION;
begin
insert into best_practices_for_procedure.log_table values (log_time, message); --只记录日志信息
end;
/
CREATE PROCEDURE
--在存储过程内调用自治事务。
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(companys text, turnover float) as
message text;
begin
    update best_practices_for_procedure.work_table set balance = balance + turnover where company =
companys;
    message := 'Company turnover ' || turnover;
    best_practices_for_procedure.proc_auto(current_timestamp, message);
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1('huawei', 1000);
 proc1
-------

(1 row)

gaussdb=# select * from best_practices_for_procedure.log_table;
           log_time           |         message
------------------------------+-----------------------
 2024-11-22 16:21:35.27499+08 | Company turnover 1000
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.log_table
drop cascades to table best_practices_for_procedure.work_table
drop cascades to function best_practices_for_procedure.proc_auto(timestamp with time zone,text)
drop cascades to function best_practices_for_procedure.proc1(text,double precision)
DROP SCHEMA
```

8.2.6 其他

8.2.6.1 DDL

数据定义语言（DDL）操作（如CREATE、ALTER、DROP）通常会加锁，以确保变更的原子性和一致性。在高并发环境中，DDL操作可能引发锁冲突或导致长时间阻塞，从而影响其他业务操作的正常执行。

建议在执行DDL变更时，暂停相关业务操作，以避免对系统性能和稳定性产生不利影响。

8.2.6.2 复杂依赖

如果存储过程或PACKAGE之间存在复杂的依赖关系，可能会在创建时遇到依赖对象尚未创建或初始化的情况，从而导致存储过程编译失败。此外，当某个对象被修改或重建时，直接或间接依赖该对象的其他存储过程和PACKAGE也会失效，需要重新编译，这会影响系统性能。

建议避免在存储过程和PACKAGE之间建立复杂的依赖关系，以提高系统的稳定性和性能。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
--创建依赖pkg2的pkg1，会报错。
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
ERROR:  "best_practices_for_procedure.pkg2.a" is not a known variable.
LINE 3: best_practices_for_procedure.pkg2.a := 100;
        ^
QUERY:   DECLARE
begin
best_practices_for_procedure.pkg2.a := 100;
end
--只能先创建被依赖的pkg2，再创建pkg1。
gaussdb=# create or replace package best_practices_for_procedure.pkg2 as
a int;
procedure p1();
end pkg2;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg2 as
procedure p1() as
begin
null;
end;
end pkg2;
/
CREATE PACKAGE BODY
gaussdb=# create or replace package best_practices_for_procedure.pkg1 as
procedure p1();
end pkg1;
/
CREATE PACKAGE
gaussdb=# create or replace package body best_practices_for_procedure.pkg1 as
procedure p1() as
begin
best_practices_for_procedure.pkg2.a := 100;
end;
end pkg1;
/
CREATE PACKAGE BODY
gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to package 16836
drop cascades to function best_practices_for_procedure.p1()
drop cascades to package 16834
drop cascades to function best_practices_for_procedure.p1()
DROP SCHEMA
```

8.2.6.3 IMMUTABLE

IMMUTABLE是存储过程的一个属性，用于声明该存储过程的结果仅依赖于输入参数，而不依赖于数据库的当前状态。在某些场景下，使用 IMMUTABLE 属性的存储过程可能会被优化为仅执行一次，如果使用不当，可能会造成不符合预期的结果。

建议在使用 IMMUTABLE 属性的存储过程和函数时，避免访问表或数据库中的信息，以确保其行为符合预期。属性详情请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE FUNCTION”章节。

```
gaussdb=# create schema best_practices_for_procedure;
CREATE SCHEMA
gaussdb=# create table best_practices_for_procedure.tb1(a int, b int);
CREATE TABLE
gaussdb=# create or replace procedure best_practices_for_procedure.proc1(a int, b int) immutable as
begin
insert into best_practices_for_procedure.tb1 values(a, b); --仅做示例，不推荐使用
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.proc1(2, 5);
ERROR:  INSERT is not allowed in a non-volatile function
CONTEXT:  SQL statement "insert into best_practices_for_procedure.tb1 values(a, b)"
PL/pgSQL function best_practices_for_procedure.proc1(integer,integer) line 3 at SQL statement
gaussdb=# create or replace function best_practices_for_procedure.func1(a int, b int) return int immutable
as
begin
return a * b;
end;
/
CREATE PROCEDURE
gaussdb=# call best_practices_for_procedure.func1(2, 5);
 func1
-------
    10
(1 row)

gaussdb=# drop schema best_practices_for_procedure cascade;
NOTICE:  drop cascades to 3 other objects
DETAIL:  drop cascades to table best_practices_for_procedure.tb1
drop cascades to function best_practices_for_procedure.proc1(integer,integer)
drop cascades to function best_practices_for_procedure.func1(integer,integer)
DROP SCHEMA
```

9 COPY 导入导出最佳实践

9.1 COPY 导入导出最佳实践（分布式）

GaussDB提供了COPY语法，可用于从数据库导出数据到文件中。数据文件支持四种格式，分别是CSV格式、BINARY格式、FIXED格式和TEXT格式。同时，COPY语法也支持将这四种格式文件导入到指定表。COPY语法详情请参见《开发指南》中“SQL参考 > SQL语法 > C > COPY”章节。

9.1.1 典型场景

前置操作

GUC开启：使用COPY命令进行数据的导入和导出操作时，数据库需要具备在服务端读取或写入文件的权限。而这一权限的开启，依赖于启用enable_copy_server_files参数。

```
gs_guc reload -I all -N all -Z datanode -Z coordinator -c "enable_copy_server_files=on"
```

须知

- COPY的导入命令的参数需与文件中的实际数据一致，比如格式类型、分隔符、换行符以及字符集等。
- 通过指定client_encoding来与服务端编码对齐的方式，COPY命令可在导出/导入过程中完整保留数据二进制形态，避免转码导致的结构破坏。该机制同时保障了数据迁移的完整性和原始特征，适用于需要高保真的数据库迁移场景。

9.1.1.1 推荐使用 CSV 格式

CSV格式通过行结束符将整个文件划分为多条记录，再通过分隔符（delimiter，默认值为逗号）将每条记录划分为多个字段，每一个字段可以通过封闭符（quote，默认值为引号）包裹起来。通过这种方式，无需对特殊字符进行转义，即可解决字段内容中出现的行结束符和分隔符等特殊字符问题。同时它是一项通用标准，具备跨平台兼容性和行业普适性，所以优先推荐采用该格式。

建议导出命令：

版权所有 © 华为云计算技术有限公司

```
COPY {data_source} TO '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {server_encoding} csv;
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
COPY {data_destination} FROM '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {file_encoding}
csv;
--data_destination 只能是一个表名称
--file_encoding 为该文件导出时指定的编码格式
```

须知

若用户自行生成CSV数据文件并导入数据库，需确保数据文件严格遵循CSV格式的标准规范，同时保证在使用COPY命令时，传入的分隔符、封闭符、行结束符等参数准确无误。

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, e'bb\nb');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=#  SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb       +
    | b
  3 | cc       c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(7 rows)
```

步骤2 整表数据导出。

```
db1=# copy test_copy to '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

导出的CSV文件内容如下所示：

```
1,aaa
2,"bb
```


版权所有 © 华为云计算技术有限公司

```
b"
3,cc c
,ddd
5,ee\e
6,"","
7,"""""
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

步骤4 自定义数据集导出：导出test_copy中ID不为空所有行的name列。

```
db1=# copy (select name from test_copy where id is not null) to '/home/xy/test.csv' delimiter ',' quote '"'
escape '"' encoding 'UTF-8' csv;
COPY 6
```

导出的CSV文件内容如下所示：

```
aaa
"bb
b"
cc c
ee\e
"","
""""
```

----结束

9.1.1.2 极致性能的导入导出场景

在对数据导入导出性能有极致要求，且是在相同版本的集群之间进行数据的导出与导入的场景下，BINARY格式通过二进制格式去存储/读取数据，而非文本格式。相比其他格式有一定的性能优势，但其存在一些限制：

1. GaussDB特有格式，不具备可移植性，且仅推荐在同版本的数据库之间导入导出时使用。
2. 二进制格式与数据类型相关性更强。例如，在文本格式下，可以从一个smallint字段导出并导入到integer列中，但是在二进制格式下不可以。
3. 部分数据类型不支持二进制导入导出，具体信息请参见BINARY约束。

建议导出命令：

```
set client_encoding = '{server_encoding}';
copy {data_source} to '/path/export.bin' binary;
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
set client_encoding = '{file_encoding}';
copy {data_destination} from '/path/export.bin' binary;
--data_destination 只能是一个表名称
--file_encoding 为该二进制文件导出时指定的编码格式
```

须知

用户应仔细研读该格式的相关约束条件。只有在完全确定待导出数据不会受到BINARY格式相关约束影响的情况下，才可以考虑选用BINARY格式，以此提升数据导出和导入的性能。

版权所有 © 华为云计算技术有限公司

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=# SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# set client_encoding = 'UTF-8';
SET
db1=# COPY test_copy TO '/home/xy/test.bin' BINARY;
COPY 6
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# set client_encoding = 'UTF-8';
SET
db1=# copy test_copy from '/home/xy/test.bin' BINARY;
COPY 6
```

----结束

9.1.1.3 需自行解析数据文件的导出场景

在从数据库中导出数据文件后，需要自行解析文件的导出场景下，FIXED格式的数据具有固定结构，每行对应一条记录，且每行中各字段值的起始偏移量和长度均固定，这使得文件解析更为便捷。因此，在该场景下，用户可选用FIXED格式。不过，FIXED格式存在一些使用约束，具体详情请参考FIXED详解。

建议导出命令：

```
COPY {data_source} FROM '/path/export.fixed' encoding {server_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

```
--col1_name(col1_offset, col1_length) 表示在数据文件的每一行里，名为col1_name的数据是从偏移量为
col1_offset的位置开始，长度为 col1_length的部分
```

对应导入命令：

```
COPY {data_destination} from '/path/export.fixed' encoding {file_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
--data_destination 只能是一个表名称
--file_encoding 为该文件导出时指定的编码格式
--col1_name(col1_offset, col1_length) 表示在数据文件的每一行里，名为col1_name的数据是从偏移量为
col1_offset的位置开始，长度为 col1_length的部分
```

须知

固定列宽格式的适用场景具有特定限制：该格式要求表中各字段数据宽度保持相对均匀，能够为每列选取固定的col_length值，既避免数据截断又不会产生冗余空格。若实际数据存在字段宽度差异较大或动态变化的情况，建议优先选用TEXT或CSV 等弹性格式，以确保数据存储的完整性和空间利用率。

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, 'bb"b');
INSERT 0 1
db1=# insert into test_copy values(3, 'cc  c');
INSERT 0 1
db1=# insert into test_copy values('', e'dd\td');
INSERT 0 1
db1=# insert into test_copy values('5', e'ee\e');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb"b
  3 | cc  c
    | dd      d
  5 | eee
(5 rows)
```

步骤2 数据导出。

```
db1=# COPY test_copy TO '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

导出数据文件内容如下所示：

```
1  aaa
2 bb"b
3cc  c
  dd    d
5  eee
```


版权所有 © 华为云计算技术有限公司

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

----结束

9.1.1.4 仅支持 TEXT 格式的导入导出场景

TEXT格式通过行结束符（EOL）将整个文件划分为一个个记录，再通过分隔符（delimiter，默认值为制表符'\t'）将单个记录划分为一个个字段。

GaussDB使用TEXT格式需要一些特殊逻辑，比如：

- 遇到退格符（0x08）、换页符（0x0C）、换行符（0x0A）、回车符（0x0D）、水平制表符（0x09）、垂直制表符（0x0B）会分别转义为'\b'、'\f'、'\n'、'\r'、'\t'、'\v'输出。
- 行结束符EOL在导入场景里默认为第一个识别到的'\n'、'\r'或'\r\n'，在导出场景里默认为 '\n'。
- 遇到反斜杠会转义为双反斜杠输出。
- 遇到NULL字段值转义为'\N'输出。

GaussDB 导出并导入到 GaussDB 场景

建议导出命令：

```
copy {data_source} to '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{server_encoding}';
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
copy {data_destination} from '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{file_encoding}';
--data_destination 只能是一个表名称
--file_encoding 为该二进制文件导出时指定的编码格式
```

示例如下：

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is\C  recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
```


版权所有 © 华为云计算技术有限公司

```
db1=# select * from test_copy;
 id |   name
----+------------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# copy test_copy to '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

----结束

GaussDB 导出后自行解析数据文件的场景

该场景下一般不希望导出的TEXT文件中有GaussDB独有的转义行为，此时需要按如下方式进行处理：

- 首先确认字段数据中是否存在行结束符或分隔符；
- 如果包含，需要使用eol或delimiter参数改用其他字符。需要确保指定的新eol或delimiter不会在字段数据中存在，建议从不可见字符（0x01 ~ 0x1F）中选取；
- 可以通过null选项指定对数据中的NULL值在导出时的表示方法。
- 最后添加without escaping参数，禁止转义输出。

建议导出命令：

```
copy {data_source} to '/path/export.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'{server_encoding}';
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

须知

转义机制的核心作用是防止数据字段中的特殊字符（如分隔符、换行符）破坏文件结构。当选择禁用转义功能时，必须遵循特殊字符隔离原则：用户需谨慎选取非转义场景下的分隔符与换行符，并严格确保这些控制字符在数据内容中完全不存在。这是实现无转义文件解析的必要条件，任何字符冲突都会导致数据解析失效或结构错乱。

示例如下：

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
```


版权所有 © 华为云计算技术有限公司

```
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

步骤3 数据读取：读取文件按照选定的行结束符0x1E进行行数据的拆分，并使用分隔符0x1F将一行数据流拆分后即可获得每一行的各个字段值。

----结束

自主构造数据文件导入 GaussDB 的场景

当用户仅拥有字段数据，却没有完整的数据文件时，需要自行构造数据文件。若希望确保字段内容不会出现大量诸如转义、特殊字符处理等改变，推荐采用TEXT格式。当用户进行构建TEXT格式的数据文件时，需自行选定分隔符以及行结束符（EOL），以此完成数据文件的构造，进而实现数据导入。

步骤1 选取行结束符：如果字段数据中存在0x0A，此时不能再使用0x0A作为行结束符，否则字段数据中的换行符会被当做行结束符，导致单行数据被一分为二。此时可以使用字段数据中不存在的字符作为行结束符，建议从不可见字符（0x01 ~ 0x1F）中选取。假设数据中未出现0x1E字符，此时选取0x1E作为行结束符。

步骤2 选取分隔符：如果字段数据中存在制表符，此时不能再使用制表符作为分隔符，否则字段数据中的制表符被当做分隔符，导致字段被一分为二。此时可以使用字段数据中不存在的字符作为分隔符，建议从不可见字符（0x01 ~ 0x1F）中选取。假设数据中未出现0x1F字符，此时选取0x1F作为分隔符。

步骤3 构造数据：使用选定步骤1与步骤2选定好的行结束符与分隔符组装数据文件。注意查看服务端的字符集，需要生成与服务端字符集一致的数据文件。

步骤4 导入数据：示例假定当前数据文件字符集为UTF-8。

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

----结束

9.1.1.5 导入导出的数据文件在 GSQL 客户端的场景

当使用GSQL执行COPY命令导出数据时，生成的数据文件默认存储在数据库服务端，这可能导致用户获取数据文件时面临一定不便。针对此场景，建议采用\COPY命令进行操作，该命令会将导出的数据文件直接生成在客户端本地。

COPY 与\COPY 的区别

1. 数据文件的位置差异：COPY导入生成的文件与导入时读取的文件均在服务端节点上，而\COPY导入生成的文件与导入时读取的文件均在客户端节点上。
2. 性能差异：由于\COPY是在客户端读取文件流后传输给服务端完成数据的导入，所以性能上会比COPY导入低。
3. 功能差异：\COPY在COPY的基础上额外支持基于客户端并行导入的能力，具体的规格约束可参考《工具参考》中“数据库连接工具 > gsql连接数据库 > 元命令参考”章节。

\COPY 命令示例

\COPY的导出命令与COPY的差异在把COPY替换成\COPY即可，此处提供一个简单的CSV格式COPY导出命令转换为\COPY导出命令的示例：

```
--COPY命令
COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
--对应的\COPY命令
\COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
\COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
```

并行导入命令示例

```
--CSV格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} csv;
--FIXED格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} fixed;
--TEXT格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num};
--data_destination 只能是一个表名称
--file_encoding 表示该二进制文件导出时指定的编码格式
--parallel_num 表示数据导入时的客户端数量，在集群资源较为充足时建议此值为8
```

9.1.1.6 基于 JDBC 驱动导入导出数据的场景

若用户有基于JDBC驱动开发来实现数据导入导出的需求，可借助JDBC驱动中提供的CopyManager接口类达成。CopyManager可用于向表批量导入数据，也能从数据库批量导出数据。

9.1.1.7 数据存在错误时需要支持容错的导入场景

当使用COPY或 \COPY命令执行数据导入时，若检测到数据异常，系统将默认终止导入任务。为此，GaussDB提供了两种容错机制：智能修正模式与严格校验模式。建议优先采用严格校验模式（Level1容错等级），该模式可在保证数据完整性的前提下跳过异常记录并且不会对导入性能影响造成太大的影响。详细信息可参考**数据存在错误时的导入操作指南**。

以下是严格校验模式Level1容错等级的导入命令：

```
--CSV格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
```

```
{file_encoding} CSV;
--BINARY格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} BINARY;
--FIXED格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} FIXED;
--TEXT格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding};
--data_destination 只能是一个表名称
--file_encoding 表示该二进制文件导出时指定的编码格式
--limit_num 表示数据导入时容错行数的上限，一旦此COPY FROM语句错误数据超过选项指定条数，则会按照原有机制报错
```

须知

如果是普通用户期望使用该特性时，需要对该用户赋权容错特性两张系统表的相关权限，具体SQL如下：

```
grant insert,select,delete on pg_catalog.gs_copy_error_log to {user_name};
```

9.1.2 数据存在错误时的导出操作指南

导出时数据存在错误的原因通常为将不满足数据库服务端编码的字符串或二进制数据插入到了数据库中，因此推荐在导出时保持客户端编码与数据库服务端编码保持一致，即可跳过服务端编码的合法性校验，也不会进行数据转码。

导出编码一致性处理原则

1. 当客户端编码（client_encoding）与服务端编码（server_encoding）一致时：
   - 执行原生数据导出；
   - 保证数据完整性和原始性；
   - 无需进行字符集转换。
2. 当客户端编码（client_encoding）与服务端编码（server_encoding）不一致时：
   - 采用客户端编码作为导出文件目标编码标准。
   - 内核中对已有数据先基于服务端编码进行编码合法性校验，存在非法编码的数据会进行报错。
   - 内核再将数据进行转码处理，对无法转码（源字符集存在码位，目标字符集不存在码位）的字符进行报错。

非法编码处理方案

当用户的数据库中存在非法编码入库的数据，想要导出时不进行报错，推荐以下两种方案：

首选方案：保持客户端编码与服务端编码保持一致后，将数据以数据库服务端编码进行导出，不进行转码。

步骤1 查询数据库服务端编码。

```
gaussdb=# show server_encoding;
```

步骤2 查询数据库客户端编码。

```
gaussdb=# show client_encoding;
```

步骤3 设置客户端编码与服务端编码一致。

```
gaussdb=# set client_encoding = '{server_encoding}';
```

步骤4 执行COPY将数据以标准的CSV格式导出到文件中。

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV;
```

----结束

次选方案：需要依赖数据库内核的转码能力，并对非法编码的字节通过占位符（'?'）进行替换，导出的数据内容会发生变化。

步骤1 查询数据库服务端编码。

```
gaussdb=# show server_encoding;
```

步骤2 设置数据库客户端编码为目标编码。

```
gaussdb=# set client_encoding = {target_encoding};
```

步骤3 依赖内核转码能力进行导出，将非法编码的字节进行替换。

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV COMPATIBLE_ILLEGAL_CHARS;
```

----结束

须知

- 使用COMPATIBLE_ILLEGAL_CHARS参数，当数据中存在非法编码的数据时，会将导出的数据进行修正，数据库内的数据不变，请酌情考虑后使用。
- 启用COMPATIBLE_ILLEGAL_CHARS参数的修改规则如下：
  - 非法字符替换：根据convert_illegal_char_mode参数配置的字符进行替换，默认替换为为'?'（U+003F）字符。
  - 零字符替换：对于零字符（U+0000）统一替换成空格字符（U+0020），如果不需要进行零字符替换，需要配置不同兼容性下的零字符功能开关。
- 关于COMPATIBLE_ILLEGAL_CHARS参数的具体使用约束，请参考COPY语法章节的COMPATIBLE_ILLEGAL_CHARS。

典型场景使用示例

针对独立零字符的基础处理逻辑相对简单，本文重点展示复合型异常场景的解决方案，即数据流中同时存在零字符（\0）与非法编码字符的容错处理过程：

1. 构造UTF8的零字符与非法字符数据。

```
gaussdb=# create database db_utf8 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE
='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db_utf8
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db_utf8" as user "omm".
db_utf8=# create table test_encodings(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db_utf8=# insert into test_encodings values(1,
dbe_raw.cast_to_varchar2(dbe_raw.concat(hextoraw('2297'),
    dbe_raw.cast_from_varchar2_to_raw('导入导出'))));
INSERT 0 1
db_utf8=# show client_encoding;
 client_encoding
```


版权所有 © 华为云计算技术有限公司

```
-----------------
 UTF8
(1 row)
--在id为1的行中，content包含零字符；在id为2的行中，content含有不属于UTF-8字符集的字符。
db_utf8=# select *, dbe_raw.cast_from_varchar2_to_raw(content) from test_encodings;
 id |  content  |  cast_from_varchar2_to_raw
----+-----------+----------------------------
  1 | "导入导出 | 2297E5AFBCE585A5E5AFBCE587BA
(1 row)
```

2. 在导出文件时，若选择与服务端相同的字符集，无需进行转码操作，文件默认能够顺利导出。然而，当选择与服务端不同的字符集时，则需要进行转码处理。在转码过程中，一旦识别到UTF-8编码中的非法字符0x97，系统将报错。此时，只需开启compatible_illegal_chars参数，文件便可成功导出。

```
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.utf8' encoding 'utf-8';
COPY 1
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.gb18030' encoding 'gb18030';
ERROR:  invalid byte sequence for encoding "UTF8": 0x97
db_utf8=# copy test_encodings to '/home/xy/encodings.txt.gb18030' encoding 'gb18030'
compatible_illegal_chars;
COPY 1
```

3. 使用UTF-8编码打开文件/home/xy/encodings.txt.utf8，当前示例未开启support_zero_character选项和compatible_illegal_chars参数。以下结果显示：第一行第二列存在乱码，虽然在显示上未呈现明显异常，但通过hexdump命令可查看出乱码数据。用户可参考此示例进行复现操作，具体数据在此不再详述。

```
1    "导入导出
```

4. 使用gb18030编码打开文件/home/xy/encodings.txt.gb18030，以下结果表明：第一行第二列的非法字符被替换为“?”。

```
1    "?导入导出
```

9.1.3 数据存在错误时的导入操作指南

数据导入过程中的容错处理机制提供两种可选模式。

智能修正模式（自适应入库）

- 处理原则：以数据完整性优先，通过智能修正确保最大程度的数据入库。
- 适用场景：导入数据中出现列数异常（多列）以及字符异常的情况。
- 处理流程：
  a. 处理列数异常（冗余列截断）。
  b. 进行字符集转码与非法字符清洗。
  c. 完整写入目标数据库。
- 输出结果：修正后的数据集。字符异常的情况会有GaussDB日志记录。
- 使用示例：
  a. 多列：为数据列与表列多的情况。此时执行COPY导入时并指定ignore_extra_data选项，GaussDB会把前面正确列的数据正常导入表中，而多余列的数据将被舍弃。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE
='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by
default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
```

```
CREATE TABLE
db1=# copy test_copy from stdin delimiter ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1,导入,导出
>>\.
ERROR:  extra data after last expected column
CONTEXT:  COPY test_copy, line 1: "1,导入,导出"
--未指定ignore_extra_data时导入失败，指定后导入成功。
db1=# copy test_copy from stdin delimiter ',' ignore_extra_data;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1,导入,导出
>> \.
COPY 1
db1=# select * from test_copy;
 id | content
----+---------
  1 | 导入
(1 row)
```

b. 字符异常：在处理字符异常问题时，需要根据服务端与客户端编码是否一致，采用不同的应对策略。

- 编码一致：当服务端与客户端编码一致时，若待导入的数据中存在非法编码字符，可以通过设置GUC参数copy_special_character_version='no_error'来实现特定的导入处理。在这种设置下，GaussDB会对不符合编码信息的数据进行容错处理，不会抛出错误信息，而是直接按照原始编码将数据插入到表中。

  数据文件可参考**数据存在错误时的导出操作指南**中示例生成的/home/xy/encodings.txt.utf8，同时创建UTF-8编码的数据库来模拟编码异常的文件在导入时不需要转码的场景。

```
gaussdb=# create database db_utf8 encoding='UTF-8' LC_COLLATE='en_US.UTF-8'
LC_CTYPE ='en_US.UTF-8' dbcompatibility = 'ORA';
CREATE DATABASE
gaussdb=# \c db_utf8
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db_utf8" as user "omm".
db_utf8=# create table test_encodings(id int, content text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column
by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
db_utf8=# show copy_special_character_version;
 copy_special_character_version
--------------------------------

(1 row)
db_utf8=# copy test_encodings from '/home/omm/temp/encodings.txt.utf8';
ERROR:  invalid byte sequence for encoding "UTF8": 0x97
CONTEXT:  COPY test_encodings, line 2
db_utf8=# set copy_special_character_version = 'no_error';
SET
db_utf8=# copy test_encodings from '/home/xy/encodings.txt.utf8';
COPY 2
db_utf8=# select * from test_encodings;
 id |  content
----+-----------
  1 | 导入
  2 | "导入导出
(2 rows)
```

- 编码不一致：当服务端与客户端编码不一致时，数据转码成为必要步骤。COPY为此提供了compatible_illegal_chars参数来支持转码操作。在数据导入过程中，当GaussDB遇到非法字符时，会启用容错机制，对这些非法字符进行转换，将转换后的字符存入数据库，并且不会报错，也

版权所有 © 华为云计算技术有限公司

不会中断整个导入流程。这使得在编码不一致的复杂环境下，数据导入工作依然能够高效、稳定地完成。

严格校验模式（精准入库）

- 处理原则：以数据准确性优先，确保入库数据的规范性要求。
- 适用场景：在医疗记录、金融交易等对数据精度要求极高的领域，若担心智能修正模式在导入数据时自动修正会影响数据准确性，可采用此模式。
- 处理流程：
  a. 执行多级校验（列数异常、字符异常、数据类型转换异常及约束冲突异常）。
  b. 生成错误诊断报告（含行号、错误类型、错误数据）。
  c. 建立错误数据隔离区。
  d. 仅通过校验的原始数据直接入库。
- 输出结果：纯净数据集以及错误明细报告（通过pg_catalog.pgxc_copy_error_log查看）。
- 容错级别：适用于智能修正模式处理的所有异常，如多列、数据类型转换错误、字段超长、转码异常等。具体使用方法如下：

```
--当导入数据过程中出现数据类型错误的次数不超过 100 次时，导入不会报错，会继续导入下一行。若超过 100 次，则正常报错。错误数据的详情及行号会记录在gs_copy_error_log表中。
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors reject limit '100' csv;
--相较于上条语句，下面这条会在gs_copy_error_log中额外记录错误行的完整数据，在无数据安全风险的场景下推荐使用。该语句需要系统管理员权限。
gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
```

总结

智能修正模式与严格校验模式可以结合使用，且智能修正模式具有优先级。在进行COPY导入时，若已明确指定对数据异常采用智能修正，那么该行数据的处理将不会触发严格校验模式。这意味着错误表不会记录相应数据，同时也不会扣除reject limit次数。建议用户根据自身实际情况，权衡是否自动修正列异常与字符异常后入库，还是直接舍弃。

9.2 COPY 导入导出最佳实践（集中式）

GaussDB提供了COPY语法，可用于从数据库导出数据到文件中。数据文件支持四种格式，分别是CSV格式、BINARY格式、FIXED格式和TEXT格式。同时，COPY语法也支持将这四种格式文件导入到指定表。COPY语法详情请参见《开发指南》中“SQL参考 > SQL语法 > C > COPY”章节。

9.2.1 典型场景

前置操作

使用COPY命令进行数据的导入和导出操作时，数据库需要具备在服务端读取或写入文件的权限。而这一权限的开启，依赖于启用enable_copy_server_files参数。

```
gs_guc reload -I all -N all -Z datanode -c "enable_copy_server_files=on"
```

须知

- COPY的导入命令的参数需与文件中的实际数据一致，比如格式类型、分隔符、换行符以及字符集等。
- 通过指定client_encoding来与服务端编码对齐的方式，COPY命令可在导出/导入过程中完整保留数据二进制形态，避免转码导致的结构破坏。该机制同时保障了数据迁移的完整性和原始特征，适用于需要高保真的数据库迁移场景。

9.2.1.1 推荐使用 CSV 格式

CSV格式通过行结束符将整个文件划分为多条记录，再通过分隔符（delimiter，默认值为逗号）将每条记录划分为多个字段，每一个字段可以通过封闭符（quote，默认值为引号）包裹起来。通过这种方式，无需对特殊字符进行转义，即可解决字段内容中出现的行结束符和分隔符等特殊字符问题。同时它是一项通用标准，具备跨平台兼容性和行业普适性，所以优先推荐采用该格式。

建议导出命令：

```
COPY {data_source} TO '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {server_encoding} csv;
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
COPY {data_destination} FROM '/path/export.csv' delimiter ',' quote '"' escape '"' encoding {file_encoding}
csv;
--data_destination 只能是一个表名称
--file_encoding 为该文件导出时指定的编码格式
```

须知

若用户自行生成CSV数据文件并导入数据库，需确保数据文件严格遵循CSV格式的标准规范，同时保证在使用COPY命令时，传入的分隔符、封闭符、行结束符等参数准确无误。

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, e'bb\nb');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
```


版权所有 © 华为云计算技术有限公司

```
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=#  SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  2 | bb       +
    | b
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(7 rows)
```

步骤2 整表数据导出。

```
db1=# copy test_copy to '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

导出的CSV文件内容如下所示：

```
1,aaa
2,"bb
b"
3,cc    c
,ddd
5,ee\e
6,","
7,"""""
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.csv' delimiter ',' quote '"' escape '"' encoding 'UTF-8' csv;
COPY 7
```

步骤4 自定义数据集导出：导出test_copy中ID不为空所有行的name列。

```
db1=# copy (select name from test_copy where id is not null) to '/home/xy/test.csv' delimiter ',' quote '"'
escape '"' encoding 'UTF-8' csv;
COPY 6
```

导出的CSV文件内容如下所示：

```
aaa
"bb
b"
cc    c
ee\e
",
"""""
```

----结束

9.2.1.2 极致性能的导入导出场景

在对数据导入导出性能有极致要求，且是在相同版本的数据库实例之间进行数据的导出与导入的场景下，BINARY格式通过二进制格式去存储/读取数据，而非文本格式。相比其他格式有一定的性能优势，但其存在一些限制：

1. GaussDB特有格式，不具备可移植性，且仅推荐在同版本的数据库之间导入导出时使用。
2. 二进制格式与数据类型相关性更强。例如，在文本格式下，可以从一个smallint字段导出并导入到integer列中，但是在二进制格式下不可以。

3. 部分数据类型不支持二进制导入导出，具体信息请参见BINARY约束。

建议导出命令：

```
set client_encoding = '{server_encoding}';
copy {data_source} to '/path/export.bin' binary;
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
set client_encoding = '{file_encoding}';
copy {data_destination} from '/path/export.bin' binary;
--data_destination 只能是一个表名称
--file_encoding 为该二进制文件导出时指定的编码格式
```

须知

用户应仔细研读该格式的相关约束条件。只有在完全确定待导出数据不会受到BINARY格式相关约束影响的情况下，才可以考虑选用BINARY格式，以此提升数据导出和导入的性能。

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, ',');
INSERT 0 1
db1=# insert into test_copy values(7, '"');
INSERT 0 1
db1=# SELECT * FROM test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# set client_encoding = 'UTF-8';
SET
db1=# COPY test_copy TO '/home/xy/test.bin' BINARY;
COPY 6
```

步骤3 数据导入。

版权所有 © 华为云计算技术有限公司

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# set client_encoding = 'UTF-8';
SET
db1=# copy test_copy from '/home/xy/test.bin' BINARY;
COPY 6
```

----结束

9.2.1.3 需自行解析数据文件的导出场景

在从数据库中导出数据文件后，需要自行解析文件的导出场景下，FIXED格式的数据具有固定结构，每行对应一条记录，且每行中各字段值的起始偏移量和长度均固定，这使得文件解析更为便捷。因此，在该场景下，用户可选用FIXED格式。不过，FIXED格式存在一些使用约束，具体详情请参考FIXED详解。

建议导出命令：

```
COPY {data_source} FROM '/path/export.fixed' encoding {server_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
--col1_name(col1_offset, col1_length) 表示在数据文件的每一行里，名为col1_name的数据是从偏移量为
col1_offset的位置开始，长度为 col1_length的部分
```

对应导入命令：

```
COPY {data_destination} from '/path/export.fixed' encoding {file_encoding} FIXED
FORMATTER(col1_name(col1_offset, col1_length), col2_name(col2_offset, col2_length));
--data_destination 只能是一个表名称
--file_encoding 为该文件导出时指定的编码格式
--col1_name(col1_offset, col1_length) 表示在数据文件的每一行里，名为col1_name的数据是从偏移量为
col1_offset的位置开始，长度为 col1_length的部分
```

须知

固定列宽格式的适用场景具有特定限制：该格式要求表中各字段数据宽度保持相对均匀，能够为每列选取固定的col_length值，既避免数据截断又不会产生冗余空格。若实际数据存在字段宽度差异较大或动态变化的情况，建议优先选用TEXT或CSV 等弹性格式，以确保数据存储的完整性和空间利用率。

示例

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# CREATE TABLE test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(2, 'bb"b');
INSERT 0 1
db1=# insert into test_copy values(3, 'cc  c');
INSERT 0 1
db1=# insert into test_copy values('', e'dd\td');
INSERT 0 1
db1=# insert into test_copy values('5', e'ee\e');
```


版权所有 © 华为云计算技术有限公司

```
INSERT 0 1
db1=# select * from test_copy;
 id |    name
----+-----------
  1 | aaa
  2 | bb"b
  3 | cc  c
    | dd      d
  5 | eee
(5 rows)
```

步骤2 数据导出。

```
db1=# COPY test_copy TO '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

导出数据文件内容如下所示：

```
1  aaa
2 bb"b
3cc  c
  dd    d
5  eee
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.fixed' encoding 'UTF-8' FIXED FORMATTER(id(0,1), name(1,5));
COPY 5
```

----结束

9.2.1.4 仅支持 TEXT 格式的导入导出场景

TEXT格式通过行结束符（EOL）将整个文件划分为一个个记录，再通过分隔符（delimiter，默认值为制表符'\t'）将单个记录划分为一个个字段。

GaussDB使用TEXT格式需要一些特殊逻辑，比如：

- 遇到退格符（0x08）、换页符（0x0C）、换行符（0x0A）、回车符（0x0D）、水平制表符（0x09）、垂直制表符（0x0B）会分别转义为'\b'、'\f'、'\n'、'\r'、'\t'、'\v'输出。
- 行结束符EOL在导入场景里默认为第一个识别到的'\n'、'\r'或'\r\n'，在导出场景里默认为 '\n'。
- 遇到反斜杠会转义为双反斜杠输出。
- 遇到NULL字段值转义为'\N'输出。

GaussDB 导出并导入到 GaussDB 场景

建议导出命令：

```
copy {data_source} to '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{server_encoding}';
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

对应导入命令：

```
copy {data_destination} from '/path/export.txt' eol e'\n' delimiter e'\t' encoding '{file_encoding}';
--data_destination 只能是一个表名称
--file_encoding 为该二进制文件导出时指定的编码格式
```

示例如下：

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# copy test_copy to '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

步骤3 数据导入。

```
db1=# truncate test_copy;
TRUNCATE TABLE
db1=# copy test_copy from '/home/xy/test.txt' eol e'\n' delimiter e'\t' encoding 'UTF-8';
COPY 6
```

----结束

GaussDB 导出后自行解析数据文件的场景

该场景下一般不希望导出的TEXT文件中有GaussDB独有的转义行为，此时需要按如下方式进行处理：

- 首先确认字段数据中是否存在行结束符或分隔符；
- 如果包含，需要使用eol或delimiter参数改用其他字符。需要确保指定的新eol或delimiter不会在字段数据中存在，建议从不可见字符（0x01 ~ 0x1F）中选取；
- 可以通过null选项指定对数据中的NULL值在导出时的表示方法。
- 最后添加without escaping参数，禁止转义输出。

建议导出命令：

```
copy {data_source} to '/path/export.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'{server_encoding}';
--data_source 可以是一个表名称，也可以是一个select语句
--server_encoding 可以通过show server_encoding获得
```

须知

转义机制的核心作用是防止数据字段中的特殊字符（如分隔符、换行符）破坏文件结构。当选择禁用转义功能时，必须遵循特殊字符隔离原则：用户需谨慎选取非转义场景下的分隔符与换行符，并严格确保这些控制字符在数据内容中完全不存在。这是实现无转义文件解析的必要条件，任何字符冲突都会导致数据解析失效或结构错乱。

示例如下：

步骤1 数据准备。

```
gaussdb=# create database db1 encoding='UTF-8' LC_COLLATE='en_US.UTF-8' LC_CTYPE ='en_US.UTF-8'
dbcompatibility = 'A';
CREATE DATABASE
gaussdb=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "omm".
db1=# create table test_copy(id int, name text);
CREATE TABLE
db1=# insert into test_copy values(1, 'aaa');
INSERT 0 1
db1=# insert into test_copy values(3, e'cc\tc');
INSERT 0 1
db1=# insert into test_copy(name) values('ddd');
INSERT 0 1
db1=# insert into test_copy values(5, e'ee\\e');
INSERT 0 1
db1=# insert into test_copy values(6, e',');
INSERT 0 1
db1=# insert into test_copy values(7, e'"');
INSERT 0 1
db1=# select * from test_copy;
 id |   name
----+-----------
  1 | aaa
  3 | cc      c
    | ddd
  5 | ee\e
  6 | ,
  7 | "
(6 rows)
```

步骤2 数据导出。

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

步骤3 数据读取：读取文件按照选定的行结束符0x1E进行行数据的拆分，并使用分隔符0x1F将一行数据流拆分后即可获得每一行的各个字段值。

----结束

自主构造数据文件导入 GaussDB 的场景

当用户仅拥有字段数据，却没有完整的数据文件时，需要自行构造数据文件。若希望确保字段内容不会出现大量诸如转义、特殊字符处理等改变，推荐采用TEXT格式。当用户进行构建TEXT格式的数据文件时，需自行选定分隔符以及行结束符（EOL），以此完成数据文件的构造，进而实现数据导入。

步骤1 选取行结束符：如果字段数据中存在0x0A，此时不能再使用0x0A作为行结束符，否则字段数据中的换行符会被当做行结束符，导致单行数据被一分为二。此时可以使用字段数据中不存在的字符作为行结束符，建议从不可见字符（0x01 ~ 0x1F）中选取。假设数据中未出现0x1E字符，此时选取0x1E作为行结束符。

版权所有 © 华为云计算技术有限公司

步骤2 选取分隔符：如果字段数据中存在制表符，此时不能再使用制表符作为分隔符，否则字段数据中的制表符被当做分隔符，导致字段被一分为二。此时可以使用字段数据中不存在的字符作为分隔符，建议从不可见字符（0x01 ~ 0x1F）中选取。假设数据中未出现0x1F字符，此时选取0x1F作为分隔符。

步骤3 构造数据：使用选定步骤1与步骤2选定好的行结束符与分隔符组装数据文件。注意查看服务端的字符集，需要生成与服务端字符集一致的数据文件。

步骤4 导入数据：示例假定当前数据文件字符集为UTF-8。

```
db1=# copy test_copy to '/home/xy/test.txt' without escaping eol e'\x1E' delimiter e'\x1F' null '\N' encoding
'UTF-8';
COPY 6
```

----结束

9.2.1.5 导入导出的数据文件在 GSQL 客户端的场景

当使用GSQL执行COPY命令导出数据时，生成的数据文件默认存储在数据库服务端，这可能导致用户获取数据文件时面临一定不便。针对此场景，建议采用\COPY命令进行操作，该命令会将导出的数据文件直接生成在客户端本地。

COPY 与\COPY 的区别

1. 数据文件的位置差异：COPY导入生成的文件与导入时读取的文件均在服务端节点上，而\COPY导入生成的文件与导入时读取的文件均在客户端节点上。
2. 性能差异：由于\COPY是在客户端读取文件流后传输给服务端完成数据的导入，所以性能上会比COPY导入低。
3. 功能差异：\COPY在COPY的基础上额外支持基于客户端并行导入的能力，具体的规格约束可参考《工具参考》中“数据库连接工具 > gsql连接数据库 > 元命令参考”章节。

\COPY 命令示例

\COPY的导出命令与COPY的差异在把COPY替换成\COPY即可，此处提供一个简单的CSV格式COPY导出命令转换为\COPY导出命令的示例：

```
--COPY命令
COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
--对应的\COPY命令
\COPY {data_source} to '/path/export.csv' encoding {server_encoding} csv;
\COPY {data_source} from '/path/export.csv' encoding {server_encoding} csv;
```

并行导入命令示例

```
--CSV格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} csv;
--FIXED格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num} fixed;
--TEXT格式的导入命令
\COPY {data_destination} from '/path/export.txt' encoding {file_encoding} parallel {parallel_num};
--data_destination 只能是一个表名称
--file_encoding 表示该二进制文件导出时指定的编码格式
--parallel_num 表示数据导入时的客户端数量，在集群资源较为充足时建议此值为8
```

9.2.1.6 基于 JDBC 驱动导入导出数据的场景

若用户有基于JDBC驱动开发来实现数据导入导出的需求，可借助JDBC驱动中提供的CopyManager接口类达成。CopyManager可用于向表批量导入数据，也能从数据库批量导出数据。

9.2.1.7 数据存在错误时需要支持容错的导入场景

当使用COPY或 \COPY命令执行数据导入时，若检测到数据异常，系统将默认终止导入任务。为此，GaussDB提供了两种容错机制：智能修正模式与严格校验模式。建议优先采用严格校验模式（Level1容错等级），该模式可在保证数据完整性的前提下跳过异常记录并且不会对导入性能影响造成太大的影响。详细信息可参考**数据存在错误时的导入操作指南**。

以下是严格校验模式Level1容错等级的导入命令：

```
--CSV格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} CSV;
--BINARY格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} BINARY;
--FIXED格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding} FIXED;
--TEXT格式
\COPY {data_destination} from '/path/export.txt' log errors reject limit '{limit_num}' encoding
{file_encoding};
--data_destination 只能是一个表名称
--file_encoding 表示该二进制文件导出时指定的编码格式
--limit_num 表示数据导入时容错行数的上限，一旦此COPY FROM语句错误数据超过选项指定条数，则会按照原有机制报错
```

须知

如果是普通用户期望使用该特性时，需要对该用户赋权容错特性两张系统表的相关权限，具体SQL如下：

```
grant insert,select,delete on pgxc_copy_error_log to {user_name};
grant insert,select,delete on gs_copy_summary to {user_name};
```

9.2.2 数据存在错误时的导出操作指南

导出时数据存在错误的原因通常为将不满足数据库服务端编码的字符串或二进制数据插入到了数据库中，因此推荐在导出时保持客户端编码与数据库服务端编码保持一致，即可跳过服务端编码的合法性校验，也不会进行数据转码。

导出编码一致性处理原则

1. 当客户端编码（client_encoding）与服务端编码（server_encoding）一致时：
   - 执行原生数据导出。
   - 保证数据完整性和原始性。
   - 无需进行字符集转换。
2. 当客户端编码（client_encoding）与服务端编码（server_encoding）不一致时：
   - 采用客户端编码作为导出文件目标编码标准。

版权所有 © 华为云计算技术有限公司

- 内核中对已有数据先基于服务端编码进行编码合法性校验，存在非法编码的数据会进行报错。
- 内核再将数据进行转码处理，对无法转码（源字符集存在码位，目标字符集不存在码位）的字符进行报错。

非法编码处理方案

当用户的数据库中存在非法编码入库的数据，想要导出时不进行报错，推荐以下两种方案：

首选方案：保持客户端编码与服务端编码保持一致后，将数据以数据库服务端编码进行导出，不进行转码。

步骤1 查询数据库服务端编码。

```
gaussdb=# show server_encoding;
```

步骤2 查询数据库客户端编码。

```
gaussdb=# show client_encoding;
```

步骤3 设置客户端编码与服务端编码一致。

```
gaussdb=# set client_encoding = '{server_encoding}';
```

步骤4 执行COPY将数据以标准的CSV格式导出到文件中。

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV;
```

----结束

次选方案：需要依赖数据库内核的转码能力，并对非法编码的字节通过占位符（'?'）进行替换，导出的数据内容会发生变化。

步骤1 查询数据库服务端编码。

```
gaussdb=# show server_encoding;
```

步骤2 设置数据库客户端编码为目标编码。

```
gaussdb=# set client_encoding = {target_encoding};
```

步骤3 依赖内核转码能力进行导出，将非法编码的字节进行替换。

```
gaussdb=# COPY test_copy TO '/data/test_copy.csv' CSV COMPATIBLE_ILLEGAL_CHARS;
```

----结束

须知

- 使用COMPATIBLE_ILLEGAL_CHARS参数，当数据中存在非法编码的数据时，会将导出的数据进行修正，数据库内的数据不变，请酌情考虑后使用。
- 启用COMPATIBLE_ILLEGAL_CHARS参数的修改规则如下：
  - 非法字符替换：根据convert_illegal_char_mode参数配置的字符进行替换，默认替换为为'?'（U+003F）字符。
  - 零字符替换：对于零字符（U+0000）统一替换成空格字符（U+0020），如果不需要进行零字符替换，需要配置不同兼容性下的零字符功能开关。
- 关于COMPATIBLE_ILLEGAL_CHARS参数的具体使用约束，请参考COPY语法章节的COMPATIBLE_ILLEGAL_CHARS。

版权所有 © 华为云计算技术有限公司

9.2.3 数据存在错误时的导入操作指南

数据导入过程中的容错处理机制提供两种可选模式。

智能修正模式（自适应入库）

- 处理原则：以数据完整性优先，通过智能修正确保最大程度的数据入库。
- 适用场景：导入数据中出现列数异常（冗余列/缺失列/废弃列）以及字符异常的情况。
- 处理流程：
  a. 处理列数异常（冗余列截断/缺失列补位/废弃列丢弃）。
  b. 进行字符集转码与非法字符清洗。
  c. 完整写入目标数据库。
- 输出结果：修正后的数据集。字符异常的情况会有GaussDB日志记录。

严格校验模式（精准入库）

- 处理原则：以数据准确性优先，确保入库数据的规范性要求。
- 适用场景：在医疗记录、金融交易等对数据精度要求极高的领域，若担心**智能修正模式**在导入数据时自动修正会影响数据准确性，可采用此模式。
- 处理流程：
  a. 执行多级校验（列数异常、字符异常、数据类型转换异常及约束冲突异常）。
  b. 生成错误诊断报告（含行号、错误类型、错误数据）。
  c. 建立错误数据隔离区。
  d. 仅通过校验的原始数据直接入库。
- 输出结果：纯净数据集以及错误明细报告（通过gs_copy_error_log与gs_copy_summary查看）。
- 容错级别：
  a. Level1容错：适用于智能修正模式处理的所有异常，如数据源列数过多、数据类型转换错误、字段超长、转码异常等。具体使用方法如下：
  ```
  --当导入数据过程中出现数据类型错误的次数不超过100次时，导入不会报错，会继续导入下一行。若超过100次，则正常报错。错误数据的详情及行号会记录在gs_copy_error_log表中。
  gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors reject limit '100' csv;
  --相较于上条语句，下面这条会在gs_copy_error_log中额外记录错误行的完整数据，在无数据安全风险的场景下推荐使用。该语句需要系统管理员权限。
  gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
  ```
  b. Level2容错：在Level1基础上，还支持处理数据中的约束冲突错误，包括非空约束、条件约束、主键约束、唯一性约束和唯一性索引等问题。具体使用方法如下：
  ```
  --设置如下GUC后，采用与Level1容错相同的COPY语句，容错逻辑也与Level1一致，只是额外支持约束冲突类型的错误。
  gaussdb=# SET a_format_load_with_constraints_violation = 's2';
  gaussdb=# copy test_copy from '/home/omm/temp/test.csv' log errors data reject limit '100' csv;
  ```

总结

智能修正模式与严格校验模式可以结合使用，且智能修正模式具有优先级。在进行COPY导入时，若已明确指定对数据异常采用智能修正，那么该行数据的处理将不会触发严格校验模式。这意味着错误表不会记录相应数据，同时也不会扣除reject limit次

版权所有 © 华为云计算技术有限公司

数。建议用户根据自身实际情况，权衡是否自动修正列异常与字符异常后入库，还是直接舍弃。

对于严格校验模式的两个级别，推荐用户默认选择Level1。这是因为Level1所支持的错误类型较为常见，并且不会对导入性能产生任何影响。而Level2目前仅在集中式A兼容环境下支持，开启该特性会额外消耗导入性能和内存资源。因此，不建议用户默认使用Level2，仅在明确数据存在约束类型冲突时再开启。

说明

- 此容错选项默认不支持对约束冲突进行容错。如需要对约束冲突进行容错，可额外设置会话级GUC参数a_format_load_with_constraints_violation为"s2"后再次导入即可。
  - 支持的约束冲突类型包括：非空约束、条件约束、主键约束、唯一性约束以及唯一性索引。
  - 该功能仅在集中式的A兼容模式下有效。
  - 导入数据的表存在语句级触发器时，该触发器遇到上述约束冲突的情况时，暂无法处理，表现为直接报错，导入数据失败。
  - 该功能下数据导入过程会从批量插入变为单行插入，对应的导入性能会有所劣化。
  - 该功能下UB-tree的索引构建过程会从批量构建变为单行构建，对应的索引构建性能会有所劣化。
  - 在行级触发器中，即使操作表触发了约束冲突，该功能依然有效。行级触发器的约束冲突通过子事务来实现，子事务会占用更多的内存资源并延长执行时间。因此，建议在明确存在约束冲突的可能性时使用该特性。此外，在这种场景下，单次COPY导入的数据量不要过大，建议不超过1GB。

10 工具导入导出最佳实践

10.1 工具导入导出最佳实践（分布式）

10.1.1 数据库级导入导出

gs_dump工具可以对单个数据库进行备份，支持四种不同的使用归档格式，不同的归档格式适用场景如表10-1所示，可以根据自身适用情况选择合适的归档格式。

gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

表 10-1 导出文件格式

格式名称	-F的参数值	说明	建议	对应导入工具
纯文本归档格式	p	纯文本脚本文件包含SQL语句和命令。命令可以由gsql命令行终端程序执行，用于重新创建数据库对象并加载表数据。	小型数据库，或者需要对导出的sql文件进行修改，推荐纯文本格式。	使用gsql工具恢复数据库对象前，可根据需要使用文本编辑器编辑纯文本导出文件。gsql工具请参考《工具参考》中“数据库连接工具 > gsql连接数据库 > gsql使用指导”。

版权所有 © 华为云计算技术有限公司

格式名称	-F的参数值	说明	建议	对应导入工具
自定义归档格式	c	一种二进制文件。支持从导出文件中恢复所有或所选数据库对象。	中型或大型数据库，需要将备份结果输出到单个文件中，推荐自定义归档格式。	使用gs_restore可以选择要从自定义归档/目录归档/tar归档导出文件中导入相应的数据库对象。 gs_restore工具请参考《工具参考》中“数据导入导出工具 > gs_restore导入数据”。
目录归档格式	d	该格式会创建一个目录，该目录包含两类文件，一类是包含数据库对象的目录文件，另一类是每个表和blob对象对应的数据文件。	中型或大型数据库，需要将数据库对象与数据分目录存储导出，推荐使用目录归档格式。	
tar归档格式	t	tar归档文件支持从导出文件中恢复所有或所选数据库对象。tar归档格式不支持压缩且单个文件大小应小于8GB。	小型数据库，需要将归档结果导出并打包，可以使用tar归档格式。	

须知

- gs_dump不会对所有数据库公共的全局对象（角色、表空间及对应权限）进行备份。因此恢复时请确保目标库或新实例上的全局对象已经创建。gs_dumpall的-g命令可以导出全局对象，同时使用gsql在目标端进行全局对象的导入。gs_dumpall工具请参考《工具参考》中“数据导入导出工具 > gs_dumpall导出所有数据库信息”。
- gs_dump和gs_restore不支持跨数据库兼容模式的导入导出，请确保源库与目标库的数据库兼容模式（兼容模式的查询与创建指定兼容模式的数据库请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE DATABASE”章节）及兼容性配置参数保持一致。
- 禁止修改-F c/d/t 格式导出的文件和内容，否则可能无法恢复成功。对于-F p 格式导出的文件，如有需要修改替换，可根据需要谨慎编辑导出文件。
- 恢复后，建议在数据库上运行ANALYZE，为优化器提供有用的统计数据信息。

备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，导出时包含数据和对象定义。

```
-- 纯文本归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_database_backup.sql > /data/backup/my_database_backup.log &
-- 自定义归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_database_backup.dmp > /data/backup/my_database_backup.log &
-- 目录归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_database_backup > /data/backup/my_database_backup.log &
-- tar归档格式
```


版权所有 © 华为云计算技术有限公司

```
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_database_backup.tar > /
data/backup/my_database_backup.log &
```

恢复时需要先创建好与源库属性相同的目标库，且库内没有任何内容。

```
-- 通过以下gsql元命令，查看数据库属性信息
\l+
-- 根据查询到的属性信息，创建目标数据库
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

再以具备sysadmin权限的用户执行如下命令进行恢复。

```
-- 纯文本归档格式
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_database_backup.sql -a > /
data/backup/my_database_restore.log &
-- 自定义归档格式
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v > /data/backup/my_database_restore.log &
-- 目录归档格式
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v
> /data/backup/my_database_restore.log &
-- tar归档格式
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v > /data/backup/my_database_restore.log &
```

10.1.2 模式级导入导出

gs_dump工具可以对单个Schema进行备份，推荐使用gs_dump工具并配合-n参数进行备份，可以连续使用多个-n备份多个Schema。

gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

须知

当导出的Schema依赖其他未导出Schema的对象时，可能会导致导入该Schema时报缺少依赖对象的错误，因此恢复时请确保依赖其他的Schema对象已经创建好后再做导入。

备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，导出时会包含数据和对象定义。

```
-- 纯文本归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_schema_backup.sql -n
my_schema > /data/backup/my_schema_backup.log &
-- 自定义归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_schema_backup.dmp -n
my_schema > /data/backup/my_schema_backup.log &
-- 目录归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_schema_backup -n
my_schema > /data/backup/my_schema_backup.log &
-- tar归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_schema_backup.tar -n
my_schema > /data/backup/my_schema_backup.log &
```

恢复时需要先创建好与源库属性相同的目标库，且库内没有目标模式。

```
-- 通过以下gsql元命令，查看数据库属性信息
\l+
-- 根据查询到的属性信息，创建目标数据库
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

再以具备sysadmin权限的用户执行如下命令进行恢复。

```
-- 纯文本归档格式
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_schema_backup.sql -a > /data/
```


版权所有 © 华为云计算技术有限公司

```
backup/my_schema_restore.log &
-- 自定义归档格式
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v -n my_schema > /data/backup/my_schema_restore.log &
-- 目录归档格式
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v -
n my_schema > /data/backup/my_schema_restore.log &
-- tar归档格式
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v -n my_schema > /data/backup/my_schema_restore.log &
```

10.1.3 表级导入导出

对于表级的导入与导出，能够使用的工具很多，可基于以下场景切入选择合适的工具。

1. 需要导出单个表的定义和表内数据到同一个文件中时，推荐使用gs_dump工具的纯文本归档，并配合-t参数使用，可以连续使用-t备份多张表。gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

   须知

   当导出的表依赖其他未导出的对象时，可能会导致导入该表时报缺少依赖对象的错误，因此恢复时请确保依赖的其他对象已经创建好后再做导入。

   备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，目标表为my_schema下的my_table。

   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t
   my_schema.my_table > /data/backup/my_table_backup.log &
   ```

   恢复时需要先创建好与源库属性相同的目标库，且库内已有目标模式，没有目标表，再以具备sysadmin权限的用户执行如下命令进行恢复。

   ```
   nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /
   data/backup/my_table_restore.log &
   ```

2. 仅需要导出单个表的定义，不需要表内数据。

   须知

   当导出的表依赖其他未导出的对象时，可能会导致导入该表时报缺少依赖对象的错误，因此恢复时请确保依赖其他的对象已经创建好后再做导入。

   推荐使用gs_dump工具的纯文本归档，配合-s参数使用，命令如下。

   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t
   my_schema.my_table -s > /data/backup/my_table_backup.log &
   ```

   恢复时需要先创建好与源库属性相同的目标库，且库内已有目标模式，没有目标表，再以具备sysadmin权限的用户执行如下命令进行恢复。

   ```
   nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /
   data/backup/my_table_restore.log &
   ```

10.2 工具导入导出最佳实践（集中式）

版权所有 © 华为云计算技术有限公司

10.2.1 数据库级导入导出

gs_dump工具可以对单个数据库进行备份，支持四种不同的使用归档格式，不同的归档格式适用场景如表10-2所示，用户可以根据自身适用情况选择合适的归档格式。

gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

表 10-2 导出文件格式

格式名称	-F的参数值	说明	建议	对应导入工具
纯文本归档格式	p	纯文本脚本文件包含SQL语句和命令。命令可以由gsql命令行终端程序执行，用于重新创建数据库对象并加载表数据。	小型数据库，或者需要对导出的sql文件进行修改，推荐纯文本格式。	使用gsql工具恢复数据库对象前，可根据需要使用文本编辑器编辑纯文本导出文件。gsql工具请参考《工具参考》中“数据库连接工具 > gsql连接数据库 > gsql使用指导”。
自定义归档格式	c	一种二进制文件。支持从导出文件中恢复所有或所选数据库对象。	中型或大型数据库，需要将备份结果输出到单个文件中，推荐自定义归档格式。	使用gs_restore可以选择要从自定义归档/目录归档/tar归档导出文件中导入相应的数据库对象。gs_restore工具请参考《工具参考》中“数据导入导出工具 > gs_restore导入数据”。
目录归档格式	d	该格式会创建一个目录，该目录包含两类文件，一类是包含数据库对象的目录文件，另一类是每个表和blob对象对应的数据文件。	中型或大型数据库，需要将数据库对象与数据分目录存储导出，推荐使用目录归档格式。	
tar归档格式	t	tar归档文件支持从导出文件中恢复所有或所选数据库对象。tar归档格式不支持压缩且单个文件大小应小于8GB。	小型数据库，需要将归档结果导出并打包，可以使用tar归档格式。	

版权所有 © 华为云计算技术有限公司

须知

- gs_dump不会对所有数据库公共的全局对象（角色、表空间及对应权限）进行备份。因此恢复时请确保目标库/新实例上的全局对象已经创建。gs_dumpall的-g命令可以导出全局对象，同时使用gsql在目标端进行全局对象的导入。gs_dumpall工具请参考《工具参考》中“数据导入导出工具 > gs_dumpall导出所有数据库信息”。
- gs_dump和gs_restore不支持跨数据库兼容模式的导入导出，请确保源库与目标库的数据库兼容模式（兼容模式的查询与创建指定兼容模式的数据库请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE DATABASE”章节）及兼容性配置参数保持一致。
- 禁止修改-F c/d/t 格式导出的文件和内容，否则可能无法恢复成功。对于-F p 格式导出的文件，如有需要修改替换，可根据需要谨慎编辑导出文件。
- 恢复后，建议在数据库上运行ANALYZE，为优化器提供有用的统计数据信息。

备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，导出时包含数据和对象定义。

```
-- 纯文本归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_database_backup.sql > /
data/backup/my_database_backup.log &
-- 自定义归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_database_backup.dmp > /
data/backup/my_database_backup.log &
-- 目录归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_database_backup > /data/
backup/my_database_backup.log &
-- tar归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_database_backup.tar > /
data/backup/my_database_backup.log &
```

恢复时需要先创建好与源库属性相同的目标库，且库内没有任何内容。

```
-- 通过以下gsql元命令，查看数据库属性信息
\l+
-- 根据查询到的属性信息，创建目标数据库
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

再以具备sysadmin权限的用户执行如下命令进行恢复。

```
-- 纯文本归档格式
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_database_backup.sql -a > /
data/backup/my_database_restore.log &
-- 自定义归档格式
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v > /data/backup/my_database_restore.log &
-- 目录归档格式
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v
> /data/backup/my_database_restore.log &
-- tar归档格式
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v > /data/backup/my_database_restore.log &
```

10.2.2 模式级导入导出

gs_dump工具可以对单个Schema进行备份，推荐使用gs_dump工具并配合-n参数进行备份，可以连续使用多个-n备份多个Schema。

gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

版权所有 © 华为云计算技术有限公司

须知

当导出的Schema依赖其他未导出Schema的对象时，可能会导致导入该Schema时报缺少依赖对象的错误，因此恢复时请确保依赖其他的Schema对象已经创建好后再做导入。

备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，导出时会包含数据和对象定义。

```
-- 纯文本归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_schema_backup.sql -n
my_schema > /data/backup/my_schema_backup.log &
-- 自定义归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F c -f /data/backup/my_schema_backup.dmp -n
my_schema > /data/backup/my_schema_backup.log &
-- 目录归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F d -f /data/backup/my_schema_backup -n
my_schema > /data/backup/my_schema_backup.log &
-- tar归档格式
nohup gs_dump my_database -U root -W ******** -p 8000 -F t -f /data/backup/my_schema_backup.tar -n
my_schema > /data/backup/my_schema_backup.log &
```

恢复时需要先创建好与源库属性相同的目标库，且库内没有目标模式。

```
-- 通过以下gsql元命令，查看数据库属性信息
\l+
-- 根据查询到的属性信息，创建目标数据库
create database my_database2 encoding='xxxxx' LC_COLLATE='xxxxx' LC_CTYPE ='xxxxx' TEMPLATE=xxx
DBCOMPATIBILITY 'xxx';
```

再以具备sysadmin权限的用户执行如下命令进行恢复。

```
-- 纯文本归档格式
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_schema_backup.sql -a > /data/
backup/my_schema_restore.log &
-- 自定义归档格式，gs_restore工具请参考《工具参考》中“数据导入导出工具->gs_restore导入数据”。
nohup gs_restore /data/backup/my_database_backup.dmp -d my_database2 -p 8000 -U root -W ******** -F c
-v -n my_schema > /data/backup/my_schema_restore.log &
-- 目录归档格式
nohup gs_restore /data/backup/my_database_backup -d my_database2 -p 8000 -U root -W ******** -F d -v -
n my_schema > /data/backup/my_schema_restore.log &
-- tar归档格式
nohup gs_restore /data/backup/my_database_backup.tar -d my_database2 -p 8000 -U root -W ******** -F t -
v -n my_schema > /data/backup/my_schema_restore.log &
```

10.2.3 表级导入导出

对于表级的导入与导出，能够使用的工具很多，可基于以下场景切入选择合适的工具。

1. 需要导出单个表的定义和表内数据到同一个文件中时，推荐使用gs_dump工具的纯文本归档，并配合-t参数使用，可以连续使用-t备份多张表。gs_dump工具请参考《工具参考》中“数据导入导出工具 > gs_dump导出数据库信息”。

   须知

   当导出的表依赖其他未导出的对象时，可能会导致导入该表时报缺少依赖对象的错误，因此恢复时请确保依赖的其他对象已经创建好后再做导入。

   备份时推荐以具备sysadmin权限的用户执行如下命令，源库为my_database，目标表为my_schema下的my_table。

   ```
   nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t
   my_schema.my_table > /data/backup/my_table_backup.log &
   ```


版权所有 © 华为云计算技术有限公司

恢复时需要先创建好与源库属性相同的目标库，且库内已有目标模式，没有目标表，再以具备sysadmin权限的用户执行如下命令进行恢复。

```
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /data/backup/my_table_restore.log &
```

2. 仅需要导出单个表的定义，不需要表内数据。

须知

当导出的表依赖其他未导出的对象时，可能会导致导入该表时报缺少依赖对象的错误，因此恢复时请确保依赖其他的对象已经创建好后再做导入。

推荐使用gs_dump工具的纯文本归档，配合-s参数使用，命令如下。

```
nohup gs_dump my_database -U root -W ******** -p 8000 -F p -f /data/backup/my_table_backup.sql -t my_schema.my_table -s > /data/backup/my_table_backup.log &
```

恢复时需要先创建好与源库属性相同的目标库，且库内已有目标模式，没有目标表，再以具备sysadmin权限的用户执行如下命令进行恢复。

```
nohup gsql -d my_database2 -p 8000 -U root -W ******** -f /data/backup/my_table_backup.sql -a > /data/backup/my_table_restore.log &
```

3. 若存在oracle的sqlldr使用习惯，或者需要将数据导入的相关日志信息（导入结果、废弃数据、出错数据）保存到客户端。gs_loader支持导入CSV、TEXT、FIXED三种格式的数据文件，推荐使用CSV格式数据文件进行导入。gs_loader工具请参考《工具参考》中“数据导入导出工具 > gs_loader导入数据”。

须知

- 三种格式的数据文件都必须以'\n'或'\r\n'为行结束符，且整个文件的行结束符保持一致（要么都是'\n'，要么都是'\r\n'）。
- 在TEXT格式上，gs_loader与COPY相比，无法识别转义后的特殊字符（如'\n'，导入后还是'\n'，不会转义为0x0A）。COPY导出的TEXT格式文件默认情况下会进行转义，如果使用gs_loader导入，被转义的字符无法转回去，会导致不一致。因此gs_loader只能导入COPY导出的不转义（使用WITHOUT ESCAPING）的TEXT格式文件。

典型场景如下所示：

– 上游数据文件格式可以自行决定或者确定为CSV格式

  CSV格式具备跨平台兼容性和行业普适性，很多数据源都支持导出CSV格式的数据文件。除非上游数据文件格式因为某些原因已确定不是CSV，需要根据实际的文件格式（TEXT或FIXED格式）来选择导入方法，此外都推荐选择CSV格式文件来导入。尤其是，如果字段数据中包含特殊字符（逗号、换行符等），只能使用CSV格式的数据文件承载（因为CSV格式会使用封闭符包裹含特殊字符的数据，防止字段数据中的特殊字符与分隔符和行结束符冲突）。

  导入标准CSV格式文件时，控制文件中增加 *FIELDS CSV* 子句，示例如下：

```
--建表。
gaussdb=# create table test_loader(id int, name name);

--查看控制文件test.ctl。
LOAD DATA
TRUNCATE INTO TABLE test_loader
FIELDS CSV
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '""'
```


版权所有 © 华为云计算技术有限公司

```
(
  id integer external,
  name char
)

--查看数据文件test.csv。
1,"aa
a"
2,"bb b"
3,"cc,c"
4,ddd

--执行导入。
gs_loader -p xxx host=xxx control=test.ctl data=test.csv -d testdb -W xxx

--导入成功，查看导入结果。
gaussdb=# select * from t1;
 id |   name
----+-----------
  1 | aa       +
    | a
  2 | bb      b
  3 | cc,c
  4 | ddd
(4 rows)
```

说明

- 如果字段封闭符不是标准的双引号，可以通过 *OPTIONALLY ENCLOSED BY* 子句设置。
- 如果字段分隔符不是标准的逗号，可以通过 *FIELDS TERMINATED BY* 子句设置。

– 上游数据文件确定为TEXT格式

在TEXT格式上，gs_loader与COPY相比，无法识别转义后的特殊字符（如'\n'，导入后还是'\n'，不会转义为0x0A）。因此包含分隔符、行结束符等特殊字符的数据无法用TEXT格式承载，否则会导致结构错乱。

对于合法的TEXT格式数据文件，导入示例如下：

```
--建表。
gaussdb=# create table test_loader(id int, name name);

--查看控制文件test.ctl。
LOAD DATA
TRUNCATE INTO TABLE test_loader
(
  id integer external,
  name char
)

--查看数据文件test.dat。
1 aaa
2 bbb
3 ccc

--执行导入。
gs_loader -p xxx host=xxx control=test.ctl data=test.dat -d testdb -W xxx

--导入成功，查看导入结果。
gaussdb=# select * from t1;
 id | name
----+------
  1 | aaa
  2 | bbb
  3 | ccc
(3 rows)
```


版权所有 © 华为云计算技术有限公司

说明

如果字段分隔符不是标准的水平制表符，可以通过 *FIELDS TERMINATED BY* 子句设置。

– 上游数据文件确定为FIXED格式

FIXED格式每列的长度固定，不使用分隔符来划分字段，因此不存在字段数据跟分隔符冲突的情况。但是如果字段数据中存在换行符，仍然无法处理。

导入示例如下：

```
--建表。
gaussdb=# create table test_loader(id int, name name);

--查看控制文件test.ctl。
LOAD DATA
TRUNCATE INTO TABLE test_loader
(
  id POSITION(1:1) integer external,
  name POSITION(2:5) char
)

--查看数据文件test.fixed。
1aaaa
2bb b
3cc,c

--执行导入。
gs_loader -p xxx host=xxx control=test.ctl data=test.fixed -d testdb -W xxx

--导入成功，查看导入结果。
gaussdb=# select * from t1;
 id |   name
----+------------
  1 | aaaa
  2 | bb      b
  3 | cc,c
(3 rows)
```


版权所有 © 华为云计算技术有限公司

11 JDBC 最佳实践

11.1 JDBC 最佳实践（分布式）

11.1.1 批量插入

11.1.1.1 场景概述

本章介绍使用JDBC驱动实现数据批量插入。

11.1.1.1.1 应用场景

场景描述

当需要向数据库中一次性插入大量数据时，与传统的多次执行单条SQL语句相比，使用批量插入可以显著提高执行效率。

本章通过实现批量插入介绍如何使用JDBC驱动连接数据库，使用事务，批量插入，获取结果集列信息等操作。

触发条件

JDBC通过addBatch接口将需要执行的sql语句添加到批量执行列表，然后通过executeBatch接口执行批量操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。

- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。

版权所有 © 华为云计算技术有限公司

- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	● 代码简单、直观，易于实现。 ● 单条失败时可精确处理，不影响其他记录。 ● 对数据库和驱动兼容性要求低。	● 网络交互次数多，每次插入都要连接/解析/提交，性能低。 ● 大量记录时容易成为瓶颈。 ● 如果不使用事务，无法保证多条插入的一致性。
批量插入	● 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 ● 可以在一个事务中一次性提交多行，保证原子性。	● 代码复杂度高，需要手动拼接占位符和参数。 ● 单语句错误会回滚所有数据，错误恢复复杂。 ● 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB 503.1.0及以上版本。

11.1.1.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

实现通过JDBC通过事务的方式批量插入数据，获取结果集列数据并输出结果信息。

11.1.1.2 架构原理

核心原理

JDBC驱动批量处理允许将多个SQL语句添加到同一批量执行列表中，然后一次性发送给数据库，并且使用U报文一次性携带本次事务提交所有插入或者更新的数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

GaussDB的U报文可以一次性携带批量更新的数据，对比多次的PBE报文通信可以极大减少网络通信的开销，提升执行效率。

11.1.1.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB503.1.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。
- 数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE TEST_BATCH(
V1 TEXT,
V2 TEXT)
CREATE TABLE
```

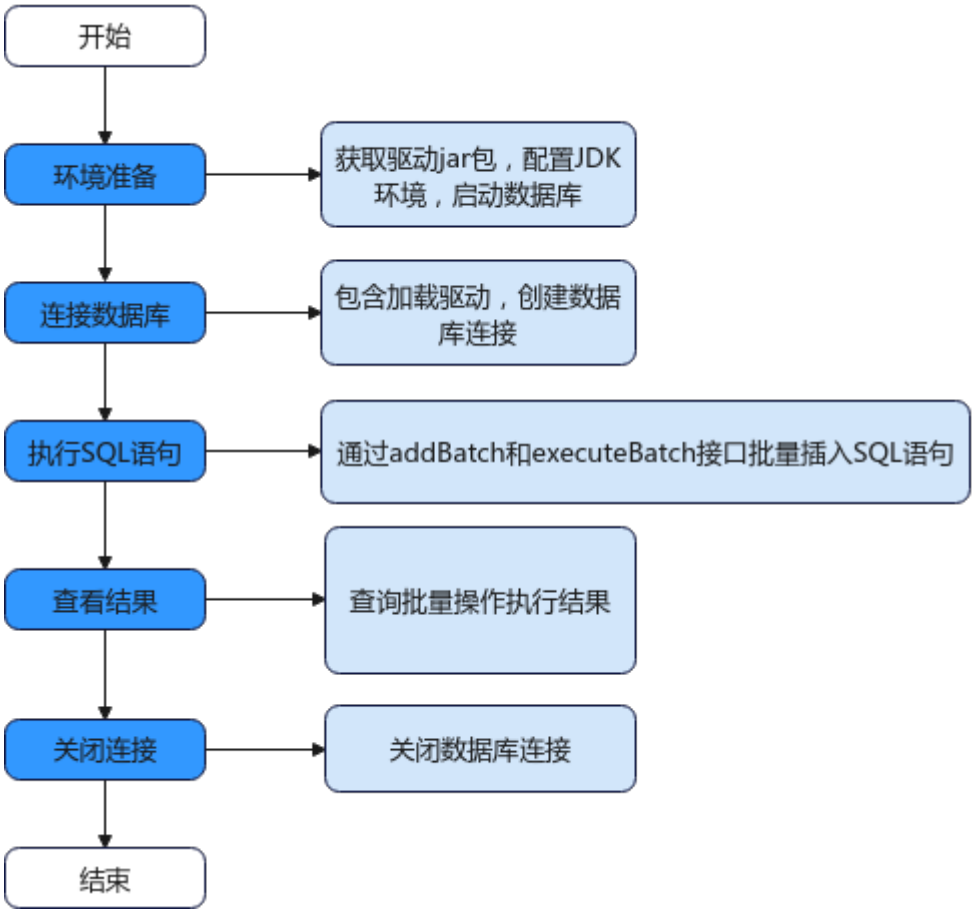
11.1.1.4 操作步骤

11.1.1.4.1 流程总览

JDBC进行插入的流程，主要包括环境准备、连接数据库、调用批量执行接口进行批量插入并查看执行结果以及关闭连接。

如图11-1所示。

图 11-1 JDBC 执行批量插入流程图



11.1.1.4.2 具体步骤

步骤1 连接数据库

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库 > 连接参数参考”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便异常的定位。使用了第三方日志框架的推荐值：Slf4JLogger。
- autoBalance：是否开启负载均衡建连。默认值：false，推荐值：true，表示采用轮询的负载均衡策略。
- batchMode：是否使用batch模式连接，配置batchMode=on时，执行批量插入/批量修改操作，每一列的数据类型以第一条数据指定的类型为准。

```
        String url = "jdbc:gaussdb://$ip:$port/database?
        connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
        rue&batchMode=on"
        Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 准备批量操作SQL语句

使用preparedStatement对语句进行预编译。用户可将语句替换成业务具体需要执行的语句。以下是向测试表中插入数据的语句：

```
String sql = "INSERT INTO  TEST_BATCH(v1,v2) VALUES(?,?)";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
```

步骤3 批量绑定参数

通过preparedStatement绑定参数，然后执行addBatch将SQL语句添加到批量执行列表。

```
for(int i=0;i<5;i++){
    preparedStatement.setString(1,"value1_"+i);
    preparedStatement.setString(2,"value2_"+i);
    preparedStatement.addBatch();
}
```

步骤4 执行批量操作

preparedStatement执行executeBatch执行批量操作，返回一个int类型的数组results，通过打印results数组可以查看批量操作影响的数据条目数。

```
Int[] results = preparedStatement.executeBatch();
System.out.println(Arrays.toString(results));
```

步骤5 释放资源和关闭数据库连接

使用使用try-with-resources打开的文件资源会自动关闭。

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(sql))
```

步骤6 异常处理

在程序运行中需要使用try-catch模块对SQLException进行异常处理，根据实际业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
// 业务代码
} catch (SQLException e) {
// 异常处理逻辑
}
```

----结束

11.1.1.4.3 完整示例

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;
public class TestBatch {
    public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database请根据业务场景进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
    public static void main(String[] args) {
```

```
        String sql = "insert into test_batch(v1,v2) values(?,?)";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(sql)) {
            conn.setAutoCommit(false);
            for (int i = 0; i < 5; i++) {
                preparedStatement.setInt(1, 1);
                preparedStatement.setString(2, "value2_" + i);
                preparedStatement.addBatch();
            }
            int[] results = preparedStatement.executeBatch();
            conn.commit();
            System.out.println(Arrays.toString(results));
        } catch (ClassNotFoundException | SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

结果验证

完整示例的运行结果展示批量操作影响的数据条数。

使用preparedStatement执行批量插入时，执行结果会返回一个int类型的数组results，results[0]表示本次批量操作影响的数据条目数总和。

```
[5, 0, 0, 0, 0]
```

回退方法

通过事务对象的rollback接口，对事务内的操作进行回滚。

11.1.1.5 常见问题

1. 现象：批量场景下通过preparedStatement对象多次绑定参数时，出现以下报错：

   ```
   java.lang.RuntimeException: java.sql.BatchUpdateException: Batch entry 0 - 5 insert into
   test_batch1(v1,v2) values(('1'),('value2_0')) was aborted: [*.*.*.*/*.*.*.*:*] ERROR: invalid input syntax
   for integer: "ss"  Call getNextException to see other errors in the batch.
   ```

   原因：批量插入场景下，如果绑定参数据类型和第一次绑定的参数类型不一致，会执行失败。

   处理方法：使用preparedStatement批量插入场景下，绑定参数的数据类型要保持一致。

2. 现象：preparedStatement执行批量插入返回的结果数组results，返回结果与Oracle不一致。

   原因：batchMode=on时，JDBC使用了GaussDB独有的报文处理逻辑，逻辑不一致但速度会更快，可以在连接串设置参数batchMode=off使返回结果与Oracle保持一致。

   设置batchMode=on，基于完整示例的结果，results内容如下所示，与Oracle相应接口返回的结果表现不一致。

   ```
   [5, 0, 0, 0, 0]
   ```

   Oracle结果如下所示：

   ```
   [1, 1, 1, 1, 1]
   ```

   设置batchMode=off参数后，与Oracle相应接口返回的结果表现一致。

   ```
   [1, 1, 1, 1, 1]
   ```

11.1.2 流式查询

11.1.2.1 场景概述

本章节主要介绍GaussDB JDBC的流式查询功能。

11.1.2.1.1 应用场景

场景描述

GaussDB的流式查询是一种逐条处理结果而非一次性加载全部结果的查询机制，适用于内存资源受限的大数据查询场景，例如大数据量导出、离线分析任务以及分页或按需加载等场景，能够有效避免内存消耗和内存溢出，提高处理速度。

触发条件

JDBC连接参数enableStreamingQuery设置为true，并在调用Statement以及PreparedStatement的executeQuery方法前，将fetchSize设置为Integer.MIN_VALUE。

业务影响

使用流式查询存在以下优势：

- 低内存消耗：流式查询显著降低了应用程序的内存占用。
- 响应速度快：客户端可立即开始处理首批数据，无需等待全部数据就绪。

但同时也存在以下潜在风险：

- 长连接占用：流式查询要求数据库连接在整个数据流传输过程中持续开放。这会导致连接资源长时间被占用，可能引发连接池耗尽或其他资源调度上的问题。
- 长事务风险：在执行流式查询的事务中，锁或MVCC快照可能长时间持有，增加死锁概率或影响写入性能。

适用版本

仅支持GaussDB 505.1.0及以上版本。

11.1.2.1.2 需求目标

业务痛点

大数据量查询时，普通查询会返回大量数据，JDBC全量读取返回的数据，容易造成内存溢出问题。

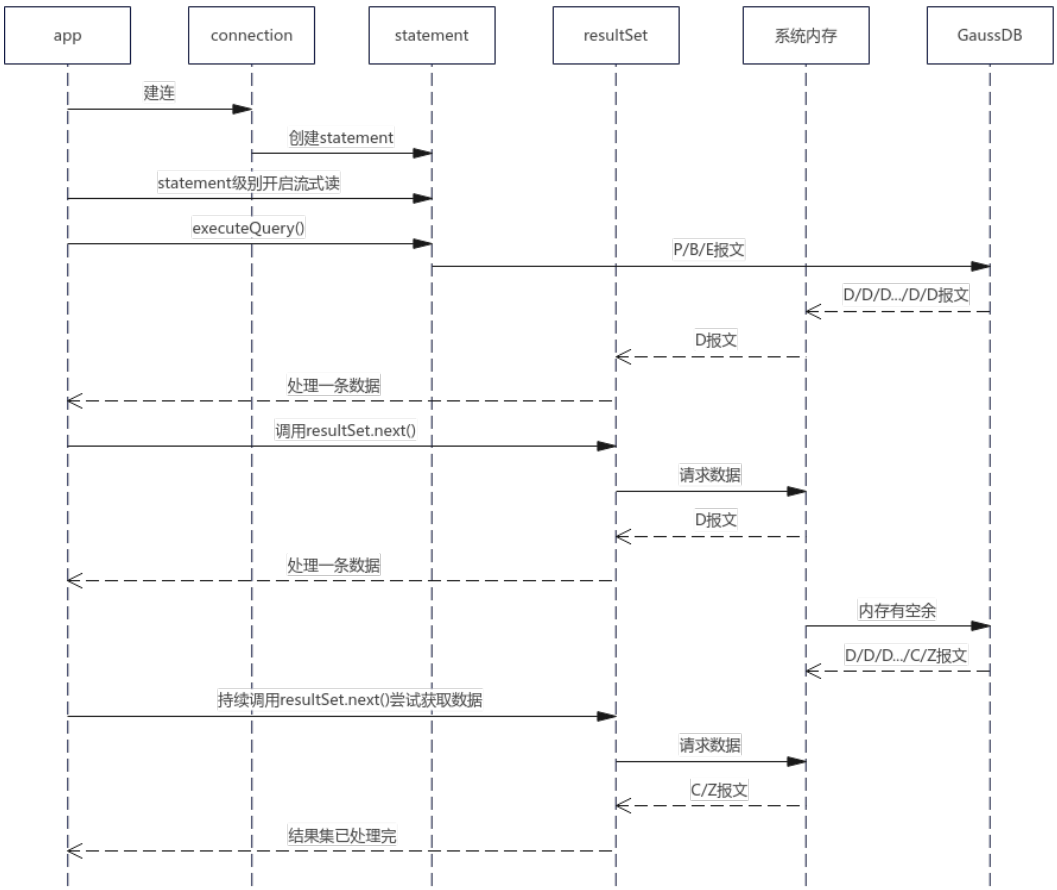
业务目标

JDBC支持流式查询功能，避免出现内存溢出问题。

11.1.2.2 架构原理

核心原理

图 11-2 流式查询原理



- 执行流式查询时，服务端将数据传入客户端socket缓冲区中，直到缓冲区存满暂停。当缓冲区中出现第一条数据时，即返回第一条D报文，JDBC开始从缓冲区逐行加载并处理数据。
- ResultSet结果集中只存放一行数据，执行next方法后，才会从缓冲区中读取下一条数据存放到结果集中，直到读取完所有的数据。

方案优势

在内存资源受限的大数据查询场景中建议使用流式查询，如果不满足业务需求，可使用以下两种查询方式。

- 普通查询：一次获取全部数据。
  优势：结果集可以向前或者向后遍历，数据消费后还能再次处理。
  劣势：处理结果慢，结果集过大会造成内存溢出。
- 批量查询：一次多行，多次获取。可参考**批量查询**。
  优势：可以返回指定数量的数据，避免内存溢出。
  劣势：处理结果慢，需要多次向服务端发送查询请求。

版权所有 © 华为云计算技术有限公司

11.1.2.3 前置准备

- 环境准备：GaussDB运行正常，获取JDBC驱动并配置环境。可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。
- 数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,1000000),repeat('GaussDB Test', 50);
INSERT 0 1000000
```

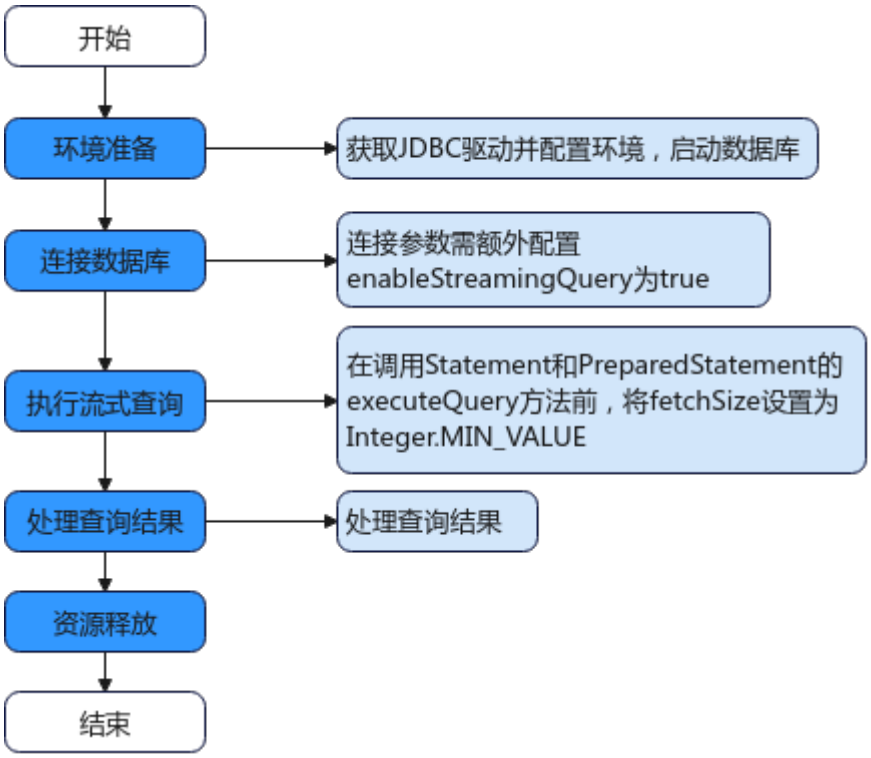
11.1.2.4 操作步骤

11.1.2.4.1 流程总览

GaussDB JDBC的流式查询功能，主要包括环境准备、连接数据库、执行流式查询、处理查询结果以及资源释放。

如图11-3所示。

图 11-3 GaussDB JDBC 的流式查询流程图



11.1.2.4.2 具体步骤

步骤1 连接数据库：连接参数需额外配置enableStreamingQuery为true，可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库”章节。

步骤2 执行流式查询：在调用Statement、PreparedStatement的executeQuery方法前，将fetchSize设置为Integer.MIN_VALUE。

版权所有 © 华为云计算技术有限公司

步骤3 处理查询结果：根据实际业务对查询结果进行处理。

步骤4 资源释放：执行完业务后，需正确释放ResultSet、Statement、Connection等资源。推荐使用try-with-resources语法进行资源释放，或在try-finally块中主动调用close方法进行资源释放。

----结束

11.1.2.4.3 完整示例

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class StreamQueryTest {
    // 连接数据库。
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database自行修改），连接参数enableStreamingQuery设置为
true。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableStreamingQuery=true";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test order by id asc limit ?";
        // 使用try-with-resources语法进行资源释放。
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql,
            ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)) {
            preparedStatement.setInt(1, 100000);
            // 执行流式查询，将fetchSize设置为Integer.MIN_VALUE，再执行executeQuery查询数据。
            preparedStatement.setFetchSize(Integer.MIN_VALUE);
            int totalCount = 0;
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // 处理查询结果，示例代码仅打印部分查询结果和数据总条数。
                    if (totalCount++ < 5) {
                        System.out.println("row:" + resultSet.getRow() + ",id :" + resultSet.getInt(1));
                    }
                }
                System.out.println("totalCount:" + totalCount);
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

结果验证

1. 将gaussdbjdbc.jar和完整示例中StreamQueryTest.java放入同一目录。
2. 编译并执行示例代码，执行时设置JVM最大堆内存为16MB。
```
javac -classpath ".:gaussdbjdbc.jar" StreamQueryTest.java
java -Xmx16M -classpath ".:gaussdbjdbc.jar" StreamQueryTest
```
3. 示例代码可正常运行，不出现内存溢出错误。执行流式查询时，每次结果集中只保留一条数据，resultSet.getRow()返回值为1。因此代码执行结果如下：
```
row:1,id :1
row:1,id :2
```

```
row:1,id :3
row:1,id :4
row:1,id :5
totalCount:100000
```

回退方法

- 关闭单条语句的流式查询功能：删除Statement、PreparedStatement的setFetchSize(Integer.MIN_VALUE)。
- 关闭当前连接的流式查询功能：连接参数enableStreamingQuery设置为false。

11.1.2.5 常见问题

1. 现象：设置Statement、PreparedStatement的fetchSize为Integer.MIN_VALUE时出现如下异常：
```
org.postgresql.util.PSQLException: Fetch size must be a value greater to or equal to 0.
    at org.postgresql.jdbc.PgStatement.setFetchSize(PgStatement.java:1623)
```
   原因：JDBC连接参数enableStreamingQuery未设置为true。
2. 现象：调用Statement、PreparedStatement的execute、executeQuery以及executeUpdate等方法进行数据的查询、插入和更新时出现如下异常：
```
org.postgresql.util.PSQLException: Streaming query statement is still active. No statements may be issued when any streaming result sets are open and in use on a given connection. Ensure that you have called .close() on any active streaming statement sets before attempting more queries.
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:492)
    at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:210)
    at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:147)
```
   原因：之前流式查询的结果集未读取完毕或未关闭。
3. 只能顺序访问：由于流式查询模式使用的是FORWARD_ONLY的ResultSet，因此只能正向逐行访问数据，无法随机定位或回滚到之前的记录。如果业务逻辑需要多次遍历或随机访问数据，则不能使用流式查询。
4. 配置与兼容性要求：使用流式查询通常需要在JDBC驱动或ORM框架中做特殊配置（**触发条件**），如果配置不当，可能无法达到预期效果。

11.1.3 自定义类型

11.1.3.1 场景概述

本章介绍通过JDBC使用自定义类型数据。

11.1.3.1.1 应用场景

场景描述

在实际生产中，用户的业务系统经常会涉及到一些复杂数据的分析和操作，传统数据类型难以完整且高效地表示或处理这种数据场景，JDBC支持的自定义类型Struct和Array可以在已有数据类型的基础上创建出新的数据类型，通过使用自定义类型可以更加方便地对数据进行增删改操作。

触发条件

JDBC操作涉及到自定义类型数据。

业务影响

- 强化数据一致性与完整性

  自定义类型允许在类型定义阶段集中声明字段的数据类型和检查约束，确保所有引用该类型的列自动继承相同的验证逻辑。

- 提升重用性与封装性

  将常用的数据结构封装为自定义类型后，可在多张表或多处函数中直接复用，减少冗余定义并提高维护效率。

- 可能引入额外的性能开销

  使用自定义类型会额外占用存储空间，且访问时需进行对象组装/拆解，可能导致更高的CPU和I/O消耗。

适用版本

仅支持GaussDB 503.0及以上版本。

11.1.3.1.2 需求目标

业务痛点

对业务中复杂的数据结构进行操作时，基本数据类型难以准确表达复杂业务概念，并且容易造成代码冗余，影响开发效率。

业务目标

JDBC支持在存储过程中使用自定义类型。

11.1.3.2 架构原理

核心原理

数据库系统会为每个自定义类型创建相应的元数据记录，包含类型的名称、结构以及约束条件等信息。JDBC使用自定义类型时，会根据自定义类型的结构和编程语言中的自定义类型对象进行转换。

方案优势

- 支持复杂数据建模

  可以处理复杂的数据结构，如嵌套对象、数组等，提高数据模型的表达能力。

- 提高数据质量且提高开发效率

  在对复杂数据结构进行查询操作时，可以使用自定义类型进行过滤操作。可对已封装的自定义类型进行复用，提高开发效率。

11.1.3.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB 503.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。

- 数据准备：创建自定义类型和存储过程，示例如下：

```
// 创建包含两条数据的自定义类型PUBLIC.COMPFOO。
gaussdb=# CREATE TYPE PUBLIC.COMPFOO AS(
    ID INTEGER,
    NAME TEXT
);
CREATE TYPE
// 创建自定义Table类型PUBLIC.COMPFOO_TABLE。
gaussdb=# CREATE TYPE PUBLIC.COMPFOO_TABLE IS TABLE OF PUBLIC.COMPFOO;
CREATE TYEP
// 创建存储过程public.test_proc拥有两个自定义类型的入参和两个自定义类型的出参。
gaussdb=# CREATE OR REPLACE PROCEDURE public.test_proc(
    IN INPUT_COMPFOO PUBLIC.COMPFOO,
    IN INPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE,
    OUT OUTPUT_COMPFOO PUBLIC.COMPFOO,
    OUT OUTPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE
)
AS
BEGIN
    OUTPUT_COMPFOO := INPUT_COMPFOO;
    OUTPUT_COMPFOO_TABLE := INPUT_COMPFOO_TABLE;
END;
/
CREATE PROCEDURE
```

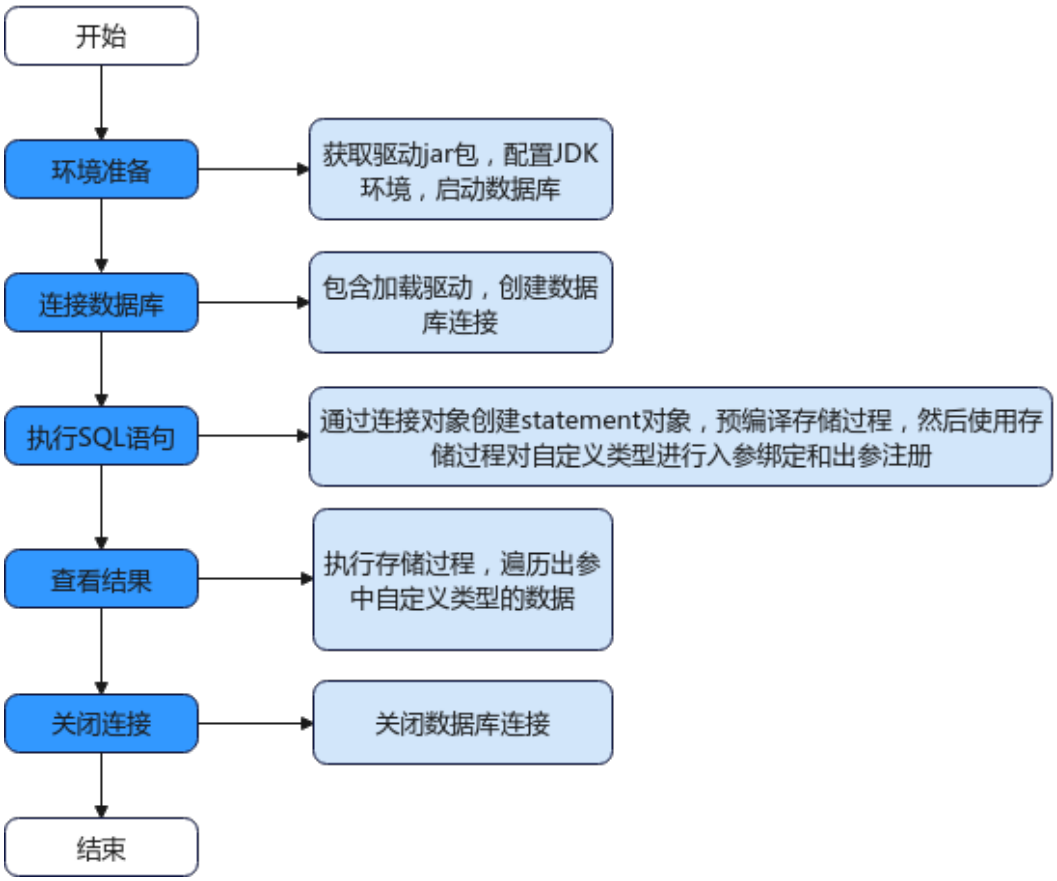
11.1.3.4 操作步骤

11.1.3.4.1 流程总览

JDBC通过存储过程对自定义类型数据插入和查询的流程如**图11-4**所示。

按照时间顺序主要包括环境准备、连接数据库、自定义类型SQL语句执行，查看结果及关闭连接。

图 11-4 JDBC 使用存储过程操作自定义类型



11.1.3.4.2 具体步骤

步骤1 连接数据库

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便异常定位。使用了第三方日志框架的推荐值：Slf4JLogger。
- autoBalance：是否开启负载均衡建连。默认值：false，推荐值：true，表示采用轮询的负载均衡策略。

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 设置GUC参数

执行SQL语句“SET behavior_compat_options='proc_outparam_override';”，打开proc_outparam_override后支持存储过程的重载。

```
Statement statement = conn.createStatement();
statement.execute("SET behavior_compat_options='proc_outparam_override'");
statement.close();
```

步骤3 预编译存储过程

通过Call语法声明调用存储过程TEST_PROC的SQL语句，使用prepareCall对其进行预编译。

```
CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
```

步骤4 绑定入参

使用PGobject对满足自定类型的数据进行组装，然后使用prepareCall进行入参绑定。

```
PGobject pgObject = new PGobject();
pgObject.setType("public.compfoo"); // 设置复合类型名。
pgObject.setValue("(1,demo)"); // 绑定复合类型值，格式为"(value1,value2)"。
cs.setObject(1, pgObject);
pgObject = new PGobject();
pgObject.setType("public.compfoo_table"); // 设置Table类型名。
pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // 绑定Table类型值，格式为
"{\"(value1,value2)\",\"(value1,value2)\",...}"。
cs.setObject(2, pgObject);
```

步骤5 注册出参类型

使用prepareCall对存储过程的出参进行注册，根据复合类型或者Table类型注册为Types.STRUCT和Types.ARRAY。

```
// 注册out类型的参数，类型为复合类型。
cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
// 注册out类型的参数，类型为Table类型,格式为"schema.typename"。
cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
```

步骤6 执行并查看结果

调用存储过程，查看出参对应的结果。

```
cs.execute();
// 获取输出参数。
// 返回结构是自定义类型。
PGobject result = (PGobject) cs.getObject(3);  // 获取out参数。
result.getValue(); // 获取复合类型字符串形式值。
String[] arrayValue = result.getArrayValue(); // 获取复合类型数组形式值，以复合数据类型字段顺序排序。
result.getStruct(); // 获取复合类型子类型名，按创建顺序排序。
result.getAttributes(); // 返回自定义类型每列组成类型的对象，对于array类型和Table类型返回的是PgArray，对
于自定义类型，封装的是PGobject，对于其他类型数据存储方式为字符串类型。
for (String s : arrayValue) {
    System.out.println(s);
}
PgArray pgArray = (PgArray) cs.getObject(4);
ResultSet rs = pgArray.getResultSet();
Object[] array = (Object[]) pgArray.getArray();
for (Object element : array) {
    System.out.println(element);
}
```

步骤7 释放资源和关闭数据库连接

```
cs.close();
conn.close();
```

步骤8 【可选】异常处理

在程序运行中需要使用try-catch模块对SQLException做异常处理，根据实际业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
   // 业务代码。
} catch (SQLException e) {
   // 异常处理逻辑。
}
```

----结束

11.1.3.4.3 完整示例

```
import com.huawei.gaussdb.jdbc.jdbc.PgArray;
import com.huawei.gaussdb.jdbc.util.PGobject;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.sql.DriverManager;
public class TypesTest {
 public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database根据实际业务进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
 public static void main(String[] args) {
  try {
   Connection conn = getConnection();
   Statement statement = conn.createStatement();
   statement.execute("set behavior_compat_options='proc_outparam_override'");
   statement.close();
   CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
   // 设置参数。
   PGobject pgObject = new PGobject();
   pgObject.setType("public.compfoo"); // 设置复合类型名。
   pgObject.setValue("(1,demo)"); // 绑定复合类型值。
   cs.setObject(1, pgObject);
   pgObject = new PGobject();
   pgObject.setType("public.compfoo_table"); // 设置Table类型名。
   pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // 绑定Table类型值，格式为
"{\"(value1,value2)\",\"(value1,value2)\",...}"。
   cs.setObject(2, pgObject);
   // 注册出参。
   // 注册out类型的参数，类型为复合类型。
   cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
   // 注册out类型的参数，类型为Table类型。
   cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
   cs.execute();
   // 获取输出参数。
   // 返回结构是自定义类型。
   PGobject result = (PGobject) cs.getObject(3); // 获取out参数。
   result.getValue(); // 获取复合类型字符串形式值。
   String[] arrayValue = result.getArrayValue(); // 获取复合类型数组形式值，以复合数据类型字段顺序排序。
   result.getStruct(); // 获取复合类型子类型名，按创建顺序排序。
   result.getAttributes(); // 返回自定义类型每列组成类型的对象，对于array类型和Table类型返回的是PgArray,对
于自定义类型，封装的是PGobject，对于其他类型数据存储方式为字符串类型。
   for (String s : arrayValue) {
    System.out.println(s);
   }
   PgArray pgArray = (PgArray) cs.getObject(4);
   ResultSet rs = pgArray.getResultSet();
```

```
    Object[] array = (Object[]) pgArray.getArray();
    for (Object element : array) {
     System.out.println(element);
    }
    cs.close();
    conn.close();
   } catch (ClassNotFoundException | SQLException e) {
    throw new RuntimeException(e);
   }
  }
}
```

结果验证

完整示例可以正确查询到自定类型的数据，完整示例的运行结果如下所示：

```
1
demo
(10,demo10)
(11,demo111)
```

回退方法

不涉及

11.1.3.5 常见问题

1. 现象：绑定自定义类型入参和注册出参后，调用相应的存储过程出现如下异常：
```
ERROR: Function public.test_proc(compfoo, public.compfoo_table, compfoo, public.compfoo_table) does not exist.
  ???No function matches the given name and argument types. You might need to add explicit type casts.
```
   原因：出参和入参绑定的自定义类型和存储过程中设定的自定义类型不一致。

   处理方法：出参入参绑定的自定义类型要和存储过程设定的一致。
2. 现象：绑定自定义类型入参和注册出参后，调用相应的存储过程出现如下异常：
```
java.lang.RuntimeException: org.postgresql.util.PSQLException: ?? CallableStatement ????????????? java.sql.Types=1111 ?? 1?????????? java.sql.Types=2002?
```
   原因：使用pgArray兼容的场景下，设置了参数enableGaussArrayAndStruct=true，无法同时兼容pgArray和GaussArray。

   处理方法：如果使用pgArray兼容场景，检查删除参数enableGaussArrayAndStruct=true。

11.1.4 批量查询

11.1.4.1 场景概述

本章介绍使用JDBC实现批量查询。

11.1.4.1.1 应用场景

场景描述

当JDBC执行查询操作，若产生大量的查询结果一次性全部返回给JDBC，可能会导致JVM内存溢出。使用批量查询方式，能够指定数据库每次返回给JDBC的数据个数，减少JVM的内存使用。

触发条件

Java应用通过JDBC连接数据库，执行大批量数据的查询。

业务影响

开发人员在数据库操作中，采用正确的事务开启与关闭方式，在批量查询场景中，通过分批次处理数据可有效减少JVM内存占用。该方案能够规避传统全量数据传输模式下，客户端一次性接收所有数据导致的Java应用内存溢出的问题。然而值得注意的是，当遍历结果集时，由于数据库与客户端之间的网络交互次数增加（尤其是每条记录均需独立传输），会引发额外的性能损耗，需在系统设计时进行权衡优化。

11.1.4.1.2 需求目标

业务痛点

在数据密集型业务场景中，传统的查询存在以下问题：

1. 若结果集较大，应用内存难以承载，会造成JVM内存溢出导致查询失败。
2. 一次性获取大量数据，可能会导致网络带宽达到瓶颈，从而影响数据传输的效率。
3. 数据库连接和相关资源持续占用影响系统整体吞吐量。

业务目标

通过批量查询的方式可正常执行查询，避免产生内存溢出。

11.1.4.2 架构原理

核心原理

JDBC批量查询利用GaussDB Kernel的游标遍历功能向服务端查询指定行数的查询结果。在查询时，JDBC首先会指定每次从数据库中返回的结果行数，数据库根据指定的行数分批次返回结果数据，直至返回所有的查询结果。

方案优劣势

1. 批量查询每次与数据库交互可返回指定行数的数据，避免因为查询结果集太大导致Java应用出现内存溢出。
2. 批量查询会多次与数据库网络交互，会有一定的性能损耗。

11.1.4.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB 503.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。

  数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
```

```
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,5),repeat('GaussDB Test', 50);
INSERT 0 5
```

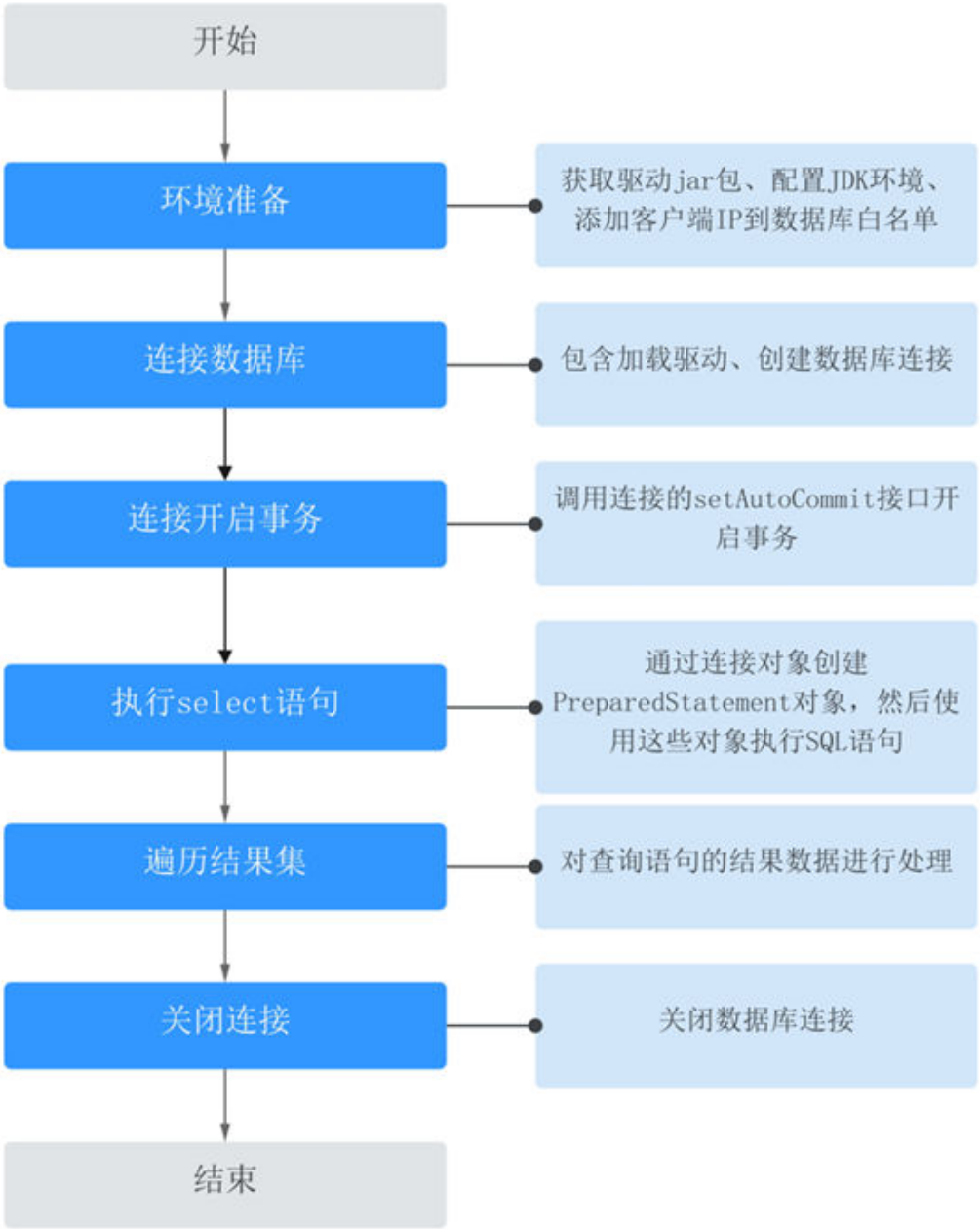
11.1.4.4 操作步骤

11.1.4.4.1 流程总览

JDBC批量查询流程如图11-5所示。

主要包括环境准备、连接数据库、开启事务、执行查询语句、结果集处理以及关闭连接。

图 11-5 JDBC 批量查询流程



版权所有 © 华为云计算技术有限公司

11.1.4.4.2 具体步骤

步骤1 创建Connection对象，与数据库建立连接。

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库 > 连接参数参考”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便异常定位。使用了第三方日志框架的推荐值：Slf4JLogger。
- autoBalance：是否开启负载均衡建连。默认值：false，推荐值：true，表示采用轮询的负载均衡策略。

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 连接开启事务。

设置自动提交为false后，JDBC内部在执行查询时会预先向数据库下发“BEGIN”主动开启事务。

```
conn.setAutoCommit(false);
```

步骤3 创建PreparedStatement对象，并且设置数据库每次返回的结果行数。

使用setFetchSize方法设置语句级的每次返回结果行数，如果连接串中配置了fetchsize，以setFetchSize方法的值为准。

```
String selectSql = "select * from tab_test";
PreparedStatement preparedStatement = conn.prepareStatement(selectSql);
preparedStatement.setFetchSize(3);
```

步骤4 执行查询，获取结果集。

```
ResultSet resultSet = preparedStatement.executeQuery();
```

步骤5 结果集处理，查看表的第一列数据。

```
while (resultSet.next()) {
    int id = resultSet.getInt(1);
    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
}
```

步骤6 获取结果集列数和列的类型等元信息。

从executeQuery返回的resultSet中获取元数据信息。

```
ResultSetMetaData metaData = resultSet.getMetaData();
System.out.println("结果列：" + metaData.getColumnCount());
System.out.println("类型编号：" + metaData.getColumnType(1));
System.out.println("类型名:" + metaData.getColumnTypeName(1));
System.out.println("列名:" + metaData.getColumnName(1));
```

步骤7 关闭资源。

使用使用try-with-resources打开的文件资源会自动关闭。

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql))
```

步骤8 【可选】异常处理。

在程序运行中需要使用try-catch模块对Exception做异常处理，根据自身业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
// 业务代码
} catch (Exception e) {
// 异常处理逻辑
}
```

----结束

11.1.4.4.3 完整示例

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class BatchQuery {
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database请根据业务场景进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql)) {
            conn.setAutoCommit(false);
            preparedStatement.setFetchSize(3);
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // 打印部分查询结果。
                    int id = resultSet.getInt(1);
                    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
                }
                ResultSetMetaData metaData = resultSet.getMetaData();
                System.out.println("结果列： " + metaData.getColumnCount());
                System.out.println("类型编号： " + metaData.getColumnType(1));
                System.out.println("类型名:" + metaData.getColumnTypeName(1));
                System.out.println("列名:" + metaData.getColumnName(1));
            }
            conn.commit();
        } catch (ClassNotFoundException | Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

结果验证

完整示例运行结果如下：

```
row:1,id :1
row:2,id :2
```

```
row:3,id :3
row:4,id :4
row:5,id :5
结果列：2
类型编号：4
类型名:int4
列名:id
```

回退方法

若需关闭批量查询功能，则去除连接串的fetchsize参数并且重新设置setFetchSize为默认值0。

11.1.4.5 常见问题

1. 现象：客户使用spring框架时，连接串中设置fetchsize之后，查询过程中存在OutOfMemoryError的报错。

   原因：执行批量查询时没有开启事务，因为一般连接的事务由Spring进行管理，在事务未开启的情况下，数据库会一次将所有数据返回给JDBC

   处理方法：排查业务代码，确认在开启批量查询之前开启了事务。

2. 现象：连接串中设置fetchsize之后，查询过程中存在OutOfMemoryError的报错。

   原因：应用代码的其他调用点内存占用较大，导致JDBC只读取少量数据就发生OutOfMemoryError。

   处理方法：通过jdk工具排查内存占用情况，判断是否是表数据导致的内存溢出。

3. 现象：通过gsql查询表很快，但是通过JDBC查询表速度很慢。

   原因：连接串中配置了fetchsize，如果fetchsize的值比较小，遍历结果集的时候与内核的报文交互次数会过多，导致性能下降。

   处理方法：在执行查询操作时，提前根据表的大小决定是否开启事务，或者调用prepareStatement.setFetchSize()方法调整每次数据库返回的行数，如果调整为0，则一次性返回所有查询结果。

11.2 JDBC 最佳实践（集中式）

11.2.1 批量插入

11.2.1.1 场景概述

本章介绍使用JDBC实现数据批量插入。

11.2.1.1.1 应用场景

场景描述

当需要向数据库中一次性插入大量数据时，与传统的多次执行单条SQL语句相比，使用批量插入可以显著提高执行效率。

本章通过实现批量插入介绍如何使用JDBC驱动连接数据库，使用事务，批量插入，获取结果集列信息等操作。

触发条件

JDBC通过addBatch接口将需要执行的sql语句添加到批量执行列表，然后通过executeBatch接口执行批量操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。

- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。

- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	• 代码简单、直观，易于实现。 • 单条失败时可精确处理，不影响其他记录。 • 对数据库和驱动兼容性要求低。	• 网络交互次数多，每次插入都要连接/解析/提交，性能低。 • 大量记录时容易成为瓶颈。 • 如果不使用事务，无法保证多条插入的一致性。
批量插入	• 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 • 可以在一个事务中一次性提交多行，保证原子性。	• 代码复杂度高，需要手动拼接占位符和参数。 • 单语句错误会回滚所有数据，错误恢复复杂。 • 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB 503.1.0及以上版本。

11.2.1.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

实现通过JDBC通过事务的方式批量插入数据，获取结果集列数据并输出结果信息。

11.2.1.2 架构原理

核心原理

JDBC批量处理允许将多个SQL语句添加到同一批量执行列表中，然后一次性发送给数据库，并且使用U报文一次性携带本次事务提交所有插入或者更新的数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

GaussDB的U报文可以一次性携带批量更新的数据，对比多次的PBE报文通信可以极大减少网络通信的开销，提升执行效率。

11.2.1.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB 503.1.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。
- 数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE TEST_BATCH(
V1 TEXT,
V2 TEXT);
CREATE TABLE
```

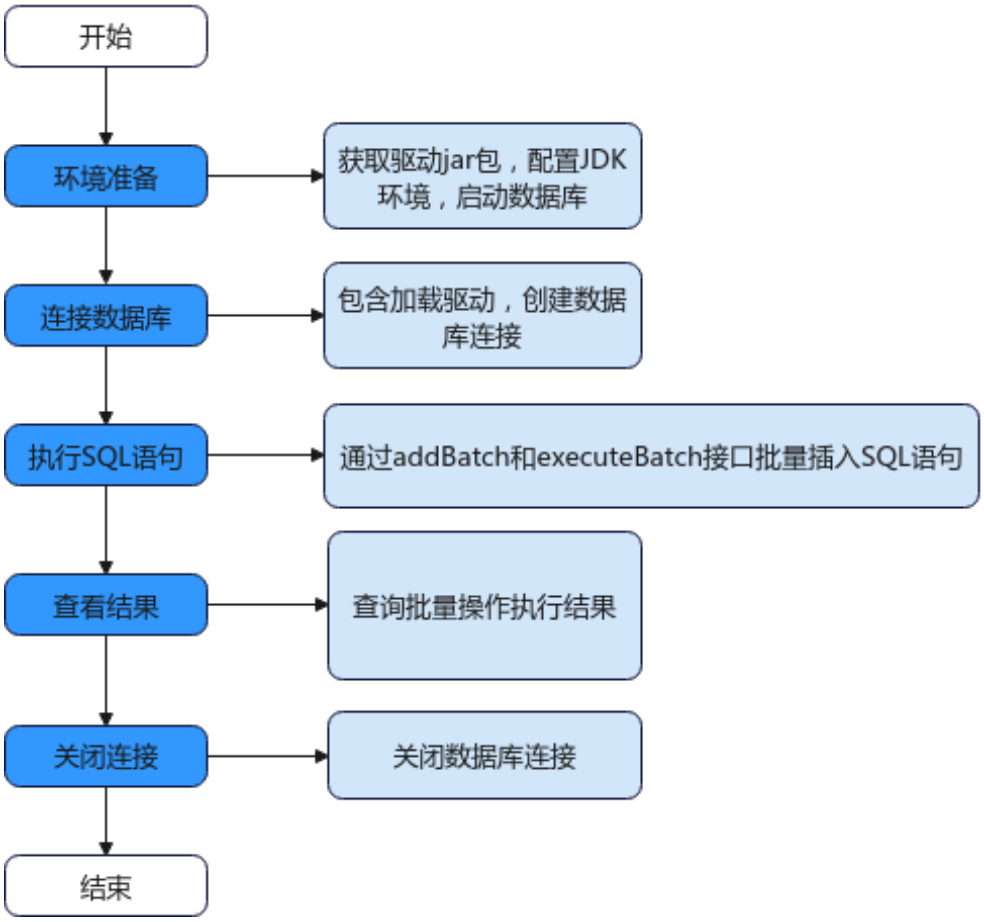
11.2.1.4 操作步骤

11.2.1.4.1 流程总览

JDBC进行批量插入的流程，主要包括环境准备、连接数据库、调用批量执行接口进行批量插入并查看执行结果以及关闭连接。

如图11-6所示。

图 11-6 JDBC 执行批量插入流程图



11.2.1.4.2 具体步骤

步骤1 连接数据库

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库 > 连接参数参考”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便出现异常定位。使用了第三方日志框架的推荐值：Slf4JLogger。
- batchMode: 是否使用batch模式连接，配置batchMode=on时，执行批量插入/批量修改操作，每一列的数据类型以第一条数据指定的类型为准。

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
```

```
        rue&batchMode=on"
        Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 准备批量操作SQL语句

使用preparedStatement对语句进行预编译。用户可将语句替换成业务具体需要执行的语句。以下是往测试表中插入数据的语句：

```
String sql = "INSERT INTO  TEST_BATCH(v1,v2) VALUES(?,?)";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
```

步骤3 批量绑定参数

通过preparedStatement绑定参数，然后执行addBatch将SQL语句添加到批量执行列表。

```
for(int i=0;i<5;i++){
    preparedStatement.setString(1,"value1_"+i);
    preparedStatement.setString(2,"value2_"+i);
    preparedStatement.addBatch();
}
```

步骤4 执行批量操作

preparedStatement执行executeBatch执行批量操作，返回一个int类型的数组results，通过打印results数组可以查看批量操作影响的数据条目数。

```
Int[] results = preparedStatement.executeBatch();
System.out.println(Arrays.toString(results));
```

步骤5 释放资源和关闭数据库连接

```
preparedStatement.close();
conn.close();
```

步骤6 异常处理

在程序运行中需要使用try-catch模块对SQLException做异常处理，根据自身业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
// 业务代码
} catch (SQLException e) {
// 异常处理逻辑
}
```

----结束

11.2.1.4.3 完整示例

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;
import java.sql.DriverManager;
public class TestBatch {
    public static Connection getConnection() throws ClassNotFoundException, SQLException{
 String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database请根据业务场景进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
    public static void main(String[] args) {
 String sql = "insert into test_batch(v1,v2) values(?,?)";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(sql)) {
            conn.setAutoCommit(false);
```

```
            for (int i = 0; i < 5; i++) {
                preparedStatement.setInt(1, 1);
                preparedStatement.setString(2, "value2_" + i);
                preparedStatement.addBatch();
            }
            int[] results = preparedStatement.executeBatch();
            conn.commit();
            System.out.println(Arrays.toString(results));
        } catch (ClassNotFoundException | SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

结果验证

以上示例的运行结果展示批量操作影响的数据条数。

使用preparedStatement执行批量插入时，执行结果会返回一个int类型的数组results，results[0]表示本次批量操作影响的数据条目数总和。

```
[5, 0, 0, 0, 0]
```

回退方法

通过事务对象的rollback接口，对事务内的操作进行回滚。

11.2.1.5 常见问题

1. 现象：批量场景下通过preparedStatement对象多次绑定参数时，出现以下报错：

   ```
   java.lang.RuntimeException: java.sql.BatchUpdateException: Batch entry 0 - 5 insert into
   test_batch1(v1,v2) values(('1'),('value2_0')) was aborted: [*.*.*.*.*/*.*.*.*.*] ERROR: invalid input syntax
   for integer: "ss"  Call getNextException to see other errors in the batch.
   ```

   原因：批量插入场景下，如果绑定参数据类型和第一次绑定的参数类型不一致，会执行失败。

   处理方法：使用preparedStatement批量插入场景下，绑定参数的数据类型要保持一致。

2. 现象：preparedStatement执行批量插入返回的结果数组results，返回结果与Oracle不一致。

   原因：batchMode=on时，JDBC使用了GaussDB独有的报文处理逻辑，逻辑不一致但速度会更快，可以在连接串设置参数batchMode=off使返回结果与Oracle保持一致。

   设置batchMode=on，基于**完整示例**的结果，results内容如下所示，与Oracle相应接口返回的结果表现不一致。

   ```
   [5, 0, 0, 0, 0]
   ```

   Oracle结果如下所示：

   ```
   [1, 1, 1, 1, 1]
   ```

   设置batchMode=off参数后，与Oracle相应接口返回的结果表现一致。

   ```
   [1, 1, 1, 1, 1]
   ```

11.2.2 流式查询

11.2.2.1 场景概述

本章节主要介绍GaussDB JDBC的流式查询功能。

11.2.2.1.1 应用场景

场景描述

GaussDB JDBC的流式查询是一种逐条处理结果而非一次性加载全部结果的查询机制，适用于内存资源受限的大数据查询场景，例如大数据量导出、离线分析任务以及分页或按需加载等场景，能够有效避免内存消耗和内存溢出，提高处理速度。

触发条件

JDBC连接参数enableStreamingQuery设置为true，并在调用Statement以及PreparedStatement的executeQuery方法前，将fetchSize设置为Integer.MIN_VALUE。

业务影响

使用流式查询存在以下优势：

- 低内存消耗：流式查询显著降低了应用程序的内存占用。
- 响应速度快：客户端可立即开始处理首批数据，无需等待全部数据就绪。

但同时也存在以下潜在风险：

- 长连接占用：流式查询要求数据库连接在整个数据流传输过程中持续开放。这会导致连接资源长时间被占用，可能引发连接池耗尽或其他资源调度上的问题。
- 长事务风险：在执行流式查询的事务中，锁或MVCC快照可能长时间持有，增加死锁概率或影响写入性能。

适用版本

仅支持GaussDB 505.1.0及以上版本。

11.2.2.1.2 需求目标

业务痛点

大数据量查询时，普通查询会返回大量数据，JDBC全量读取返回的数据，容易造成内存溢出问题。

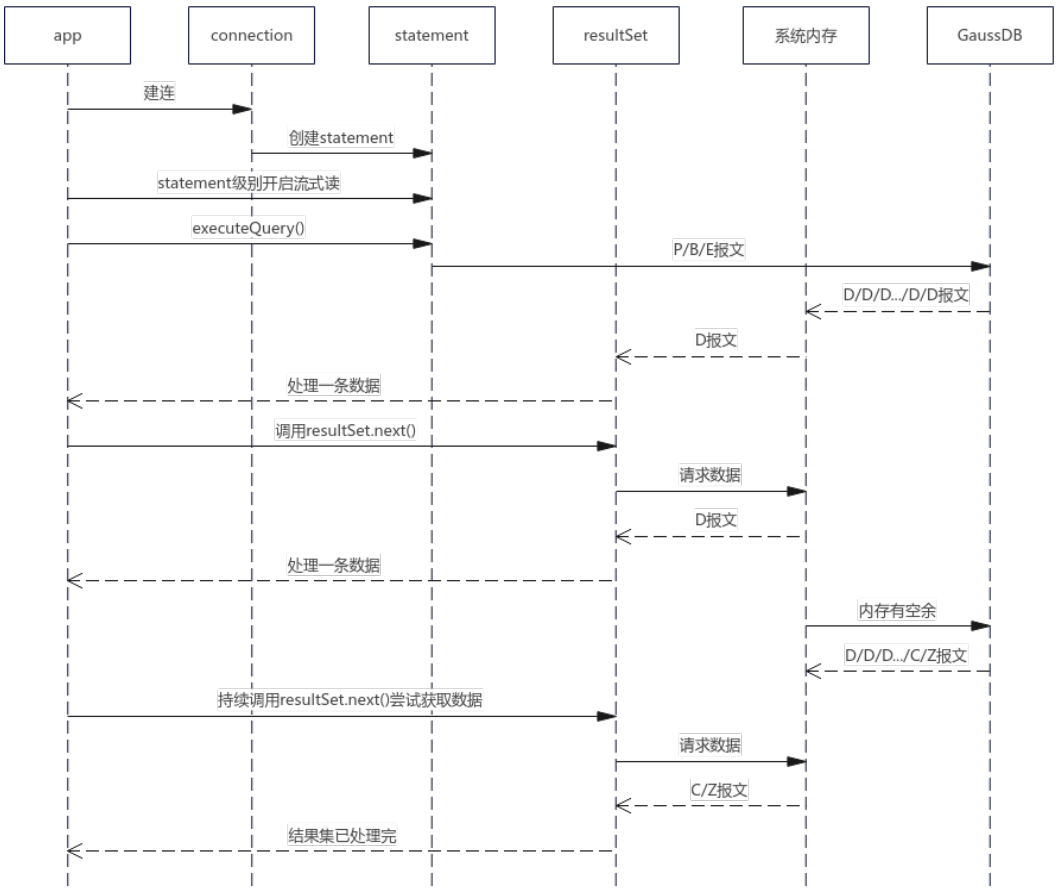
业务目标

JDBC支持流式查询功能，避免出现内存溢出问题。

11.2.2.2 架构原理

核心原理

图 11-7 流式查询原理



- 执行流式查询时，服务端将数据传入客户端socket缓冲区中，直到缓冲区存满暂停。当缓冲区中出现第一条数据时，即返回第一条D报文，JDBC开始从缓冲区逐行加载并处理数据。
- ResultSet结果集中只存放一行数据，执行next方法后，才会从缓冲区中读取下一条数据存放到结果集中，直到读取完所有的数据。

方案优势

在内存资源受限的大数据查询场景中建议使用流式查询，如果不满足业务需求，可使用以下两种查询方式。

- 普通查询：一次获取全部数据。
  优势：结果集可以向前或者向后遍历，数据消费后还能再次处理。
  劣势：处理结果慢，结果集过大会造成内存溢出。
- 批量查询：一次多行，多次获取。可参考**批量查询**。
  优势：可以返回指定数量的数据，避免内存溢出。
  劣势：处理结果慢，需要多次向服务端发送查询请求。

版权所有 © 华为云计算技术有限公司

11.2.2.3 前置准备

- 环境准备：数据库运行正常，获取JDBC驱动并配置环境。可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。
- 数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,1000000),repeat('GaussDB Test', 50);
INSERT 0 1000000
```

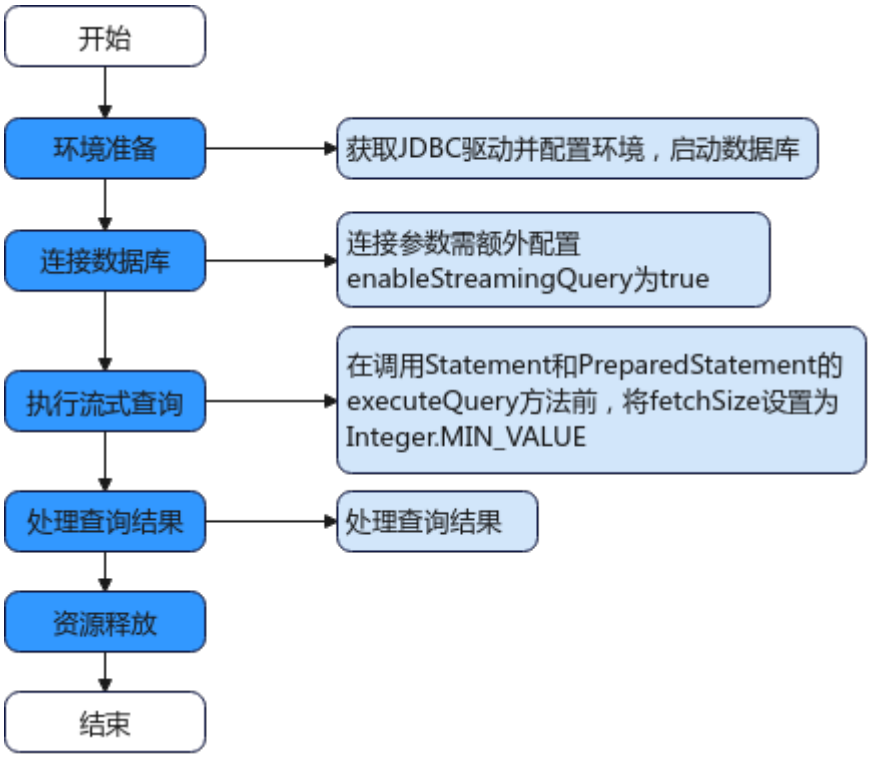
11.2.2.4 操作步骤

11.2.2.4.1 流程总览

GaussDB JDBC的流式查询功能，主要包括环境准备、连接数据库、执行流式查询、处理查询结果以及资源释放。

如图11-8所示。

图 11-8 GaussDB JDBC 的流式查询流程图



11.2.2.4.2 具体步骤

步骤1 连接数据库：连接参数需额外配置enableStreamingQuery为true，可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库”章节。

步骤2 执行流式查询：在调用Statement和PreparedStatement的executeQuery方法前，将fetchSize设置为Integer.MIN_VALUE。

版权所有 © 华为云计算技术有限公司

步骤3 处理查询结果：根据实际业务对查询结果进行处理。

步骤4 资源释放：执行完业务后，需正确释放ResultSet、Statement以及Connection等资源。推荐使用try-with-resources语法进行资源释放，或在try-finally块中主动调用close方法进行资源释放。

----结束

11.2.2.4.3 完整示例

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class StreamQueryTest {
    // 连接数据库。
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database自行修改），连接参数enableStreamingQuery设置为
true。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database?enableStreamingQuery=true";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test order by id asc limit ?";
        // 使用try-with-resources语法进行资源释放。
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql,
            ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)) {
            preparedStatement.setInt(1, 100000);
            // 执行流式查询，将fetchSize设置为Integer.MIN_VALUE，再执行executeQuery查询数据。
            preparedStatement.setFetchSize(Integer.MIN_VALUE);
            int totalCount = 0;
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // 处理查询结果，示例代码仅打印部分查询结果和数据总条数。
                    if (totalCount++ < 5) {
                        System.out.println("row:" + resultSet.getRow() + ",id :" + resultSet.getInt(1));
                    }
                }
                System.out.println("totalCount:" + totalCount);
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

结果验证

1. 将gaussdbjdbc.jar和完整示例中StreamQueryTest.java放入同一目录。
2. 编译并执行示例代码，执行时设置JVM最大堆内存为16MB。
```
javac -classpath ".:gaussdbjdbc.jar" StreamQueryTest.java
java -Xmx16M -classpath ".:gaussdbjdbc.jar" StreamQueryTest
```
3. 完整示例可正常运行，不出现内存溢出错误。执行流式查询时，每次结果集中只保留一条数据，resultSet.getRow()返回值为1。因此代码执行结果如下：
```
row:1,id :1
row:1,id :2
```

```
row:1,id :3
row:1,id :4
row:1,id :5
totalCount:100000
```

回退方法

- 关闭单条语句的流式查询功能：删除Statement、PreparedStatement的setFetchSize(Integer.MIN_VALUE)。
- 关闭当前连接的流式查询功能：连接参数enableStreamingQuery设置为false。

11.2.2.5 常见问题

1. 现象：设置Statement、PreparedStatement的fetchSize为Integer.MIN_VALUE时出现如下异常：
```
org.postgresql.util.PSQLException: Fetch size must be a value greater to or equal to 0.
    at org.postgresql.jdbc.PgStatement.setFetchSize(PgStatement.java:1623)
```
   原因：JDBC连接参数enableStreamingQuery未设置为true。
2. 现象：调用Statement、PreparedStatement的execute、executeQuery以及executeUpdate等方法进行数据的查询、插入和更新时出现如下异常：
```
org.postgresql.util.PSQLException: Streaming query statement is still active. No statements may be issued when any streaming result sets are open and in use on a given connection. Ensure that you have called .close() on any active streaming statement sets before attempting more queries.
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:492)
    at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:210)
    at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:147)
```
   原因：之前进行流式查询的结果集未读取完毕或未关闭。
3. 只能顺序访问：由于流式查询模式使用的是FORWARD_ONLY的ResultSet，因此只能正向逐行访问数据，无法随机定位或回滚到之前的记录。如果业务逻辑需要多次遍历或随机访问数据，则不能使用流式查询。
4. 配置与兼容性要求：使用流式查询通常需要在JDBC驱动或ORM框架中做特殊配置（**触发条件**），如果配置不当，可能无法达到预期效果。

11.2.3 自定义类型

11.2.3.1 场景概述

本章介绍通过JDBC使用自定义类型数据。

11.2.3.1.1 应用场景

场景描述

在实际生产中，用户的业务系统经常会涉及到一些复杂数据的分析和操作，传统数据类型难以完整且高效地表示或处理该数据场景，JDBC支持的自定义类型Struct和Array可以在已有数据类型的基础上创建出新的数据类型，通过使用自定义类型可以更加方便地对数据进行增删改操作。

触发条件

JDBC操作涉及到自定义类型数据。

业务影响

- 强化数据一致性与完整性

  自定义类型允许在类型定义阶段集中声明字段的数据类型和检查约束，确保所有引用该类型的列自动继承相同的验证逻辑。

- 提升重用性与封装性

  将常用的数据结构封装为自定义类型后，可在多张表或多处函数中直接复用，减少冗余定义并提高维护效率。

- 可能引入额外的性能开销

  使用自定义类型会额外占用存储空间，且访问时需进行对象组装/拆解，可能导致更高的CPU和I/O消耗。

适用版本

仅支持GaussDB 503.0及以上版本。

11.2.3.1.2 需求目标

业务痛点

对业务中复杂的数据结构进行操作时，基本数据类型难以精确表达复杂业务概念，并且容易造成代码冗余，影响开发效率。

业务目标

JDBC支持在存储过程中使用自定义类型。

11.2.3.2 架构原理

核心原理

数据库系统会为每个自定义类型创建相应的元数据记录，包含类型的名称、结构以及约束条件等信息。JDBC使用自定义类型时，会根据自定义类型的结构和编程语言中的自定义类型对象进行转换。

方案优势

- 支持复杂数据建模

  可以处理复杂的数据结构，如嵌套对象以及数组等，提高数据模型的表达能力。

- 提高数据质量且提高开发效率

  在对复杂数据结构进行查询操作时，可以使用自定义类型进行过滤操作。可对已封装的自定义类型进行复用，提高开发效率。

11.2.3.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB 503.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。

- 数据准备：创建自定义类型和存储过程，示例如下：

```
// 创建包含两条数据的自定义类型PUBLIC.COMPFOO。
gaussdb=# CREATE TYPE PUBLIC.COMPFOO AS(
    ID INTEGER,
    NAME TEXT
);
CREATE TYPE
// 创建自定义Table类型PUBLIC.COMPFOO_TABLE。
gaussdb=# CREATE TYPE PUBLIC.COMPFOO_TABLE IS TABLE OF PUBLIC.COMPFOO;
CREATE TYEP
// 创建存储过程public.test_proc拥有两个自定义类型的入参和两个自定义类型的出参。
gaussdb=# CREATE OR REPLACE PROCEDURE public.test_proc(
    IN INPUT_COMPFOO PUBLIC.COMPFOO,
    IN INPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE,
    OUT OUTPUT_COMPFOO PUBLIC.COMPFOO,
    OUT OUTPUT_COMPFOO_TABLE PUBLIC.COMPFOO_TABLE
)
AS
BEGIN
    OUTPUT_COMPFOO := INPUT_COMPFOO;
    OUTPUT_COMPFOO_TABLE := INPUT_COMPFOO_TABLE;
END;
/
CREATE PROCEDURE
```

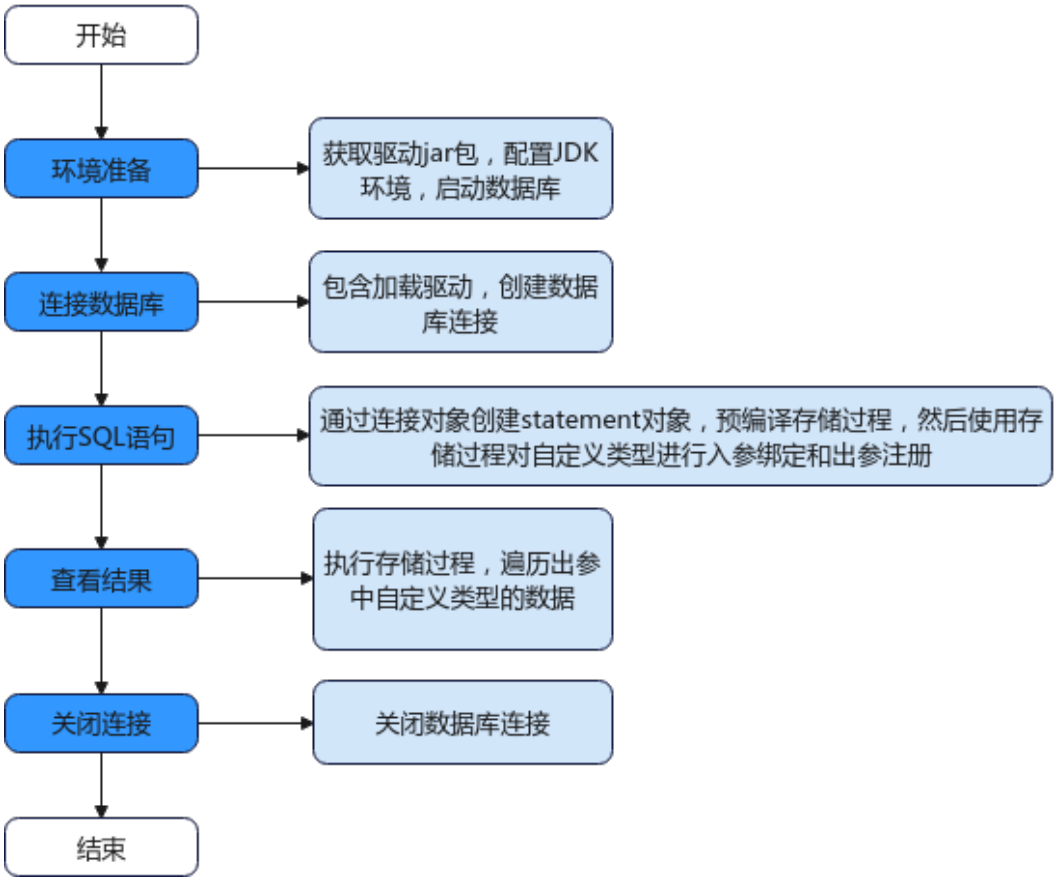
11.2.3.4 操作步骤

11.2.3.4.1 流程总览

JDBC通过存储过程对自定义类型数据插入和查询的流程如**图11-9**所示。

按照时间顺序主要包括环境准备、连接数据库、自定义类型SQL语句执行，查看结果及关闭连接。

图 11-9 JDBC 使用存储过程操作自定义类型



11.2.3.4.2 具体步骤

步骤1 连接数据库

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便出现异常定位。使用了第三方日志框架的推荐值：Slf4JLogger。

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=true"
Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 设置GUC参数

版权所有 © 华为云计算技术有限公司

执行SQL语句“SET behavior_compat_options='proc_outparam_override';”，打开proc_outparam_override后支持存储过程的重载。

```
Statement statement = conn.createStatement();
statement.execute("SET behavior_compat_options='proc_outparam_override'");
statement.close();
```

步骤3 预编译存储过程

通过Call语法声明调用存储过程TEST_PROC的SQL语句，然后使用prepareCall对这条语句进行预编译。

```
CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
```

步骤4 绑定入参

使用PGobject对满足自定类型的数据进行组装，然后使用prepareCall进行入参绑定。

```
PGobject pgObject = new PGobject();
pgObject.setType("public.compfoo"); // 设置复合类型名。
pgObject.setValue("(1,demo)"); // 绑定复合类型值，格式为"(value1,value2)"。
cs.setObject(1, pgObject);
pgObject = new PGobject();
pgObject.setType("public.compfoo_table"); // 设置Table类型名。
pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // 绑定Table类型值，格式为
"{\"(value1,value2)\",\"(value1,value2)\",...}"。
cs.setObject(2, pgObject);
```

步骤5 注册出参类型

使用prepareCall对存储过程的出参进行注册，根据复合类型或者Table类型注册为Types.STRUCT和Types.ARRAY。

```
// 注册out类型的参数，类型为复合类型。
cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
// 注册out类型的参数，类型为Table类型。
cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
```

步骤6 执行并查看结果

调用存储过程，查看出参对应的结果。

```
cs.execute();
// 获取输出参数。
// 返回结构是自定义类型。
PGobject result = (PGobject) cs.getObject(3);  // 获取out参数。
result.getValue(); // 获取复合类型字符串形式值。
String[] arrayValue = result.getArrayValue(); // 获取复合类型数组形式值，以复合数据类型字段顺序排序。
result.getStruct(); // 获取复合类型子类型名，按创建顺序排序。
result.getAttributes(); // 返回自定义类型每列组成类型的对象，对于array类型和table类型返回的是PgArray,对于
自定义类型，封装的是PGobject，对于其他类型数据存储方式为字符串类型。
for (String s : arrayValue) {
    System.out.println(s);
}
PgArray pgArray = (PgArray) cs.getObject(4);
ResultSet rs = pgArray.getResultSet();
Object[] array = (Object[]) pgArray.getArray();
for (Object element : array) {
    System.out.println(element);
}
```

步骤7 释放资源和关闭数据库连接

```
cs.close();
conn.close();
```

步骤8 【可选】异常处理

在程序运行中需要使用try-catch模块对SQLException做异常处理，根据自身业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
    // 业务代码。
} catch (SQLException e) {
    // 异常处理逻辑。
}
```

----结束

11.2.3.4.3 完整示例

```
import com.huawei.gaussdb.jdbc.jdbc.PgArray;
import com.huawei.gaussdb.jdbc.util.PGobject;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.sql.DriverManager;
public class TypesTest {
 public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        // 指定数据库sourceURL（$ip、$port、database根据实际业务进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }
 public static void main(String[] args) {
  try {
    Connection conn = getConnection();
    Statement statement = conn.createStatement();
    statement.execute("set behavior_compat_options='proc_outparam_override'");
    statement.close();
    CallableStatement cs = conn.prepareCall("{CALL PUBLIC.TEST_PROC(?,?,?,?)}");
    // 设置参数。
    PGobject pgObject = new PGobject();
    pgObject.setType("public.compfoo"); // 设置复合类型名。
    pgObject.setValue("(1,demo)"); // 绑定复合类型值。
    cs.setObject(1, pgObject);
    pgObject = new PGobject();
    pgObject.setType("public.compfoo_table"); // 设置Table类型名。
    pgObject.setValue("{\"(10,demo10)\",\"(11,demo111)\"}"); // 绑定Table类型值，格式为
"{\"(value1,value2)\",\"(value1,value2)\",...}"。
    cs.setObject(2, pgObject);
    // 注册出参。
    // 注册out类型的参数，类型为复合类型。
    cs.registerOutParameter(3, Types.STRUCT, "public.compfoo");
    // 注册out类型的参数，类型为Table类型。
    cs.registerOutParameter(4, Types.ARRAY, "public.compfoo_table");
    cs.execute();
    // 获取输出参数。
    // 返回结构是自定义类型。
    PGobject result = (PGobject) cs.getObject(3); // 获取out参数。
    result.getValue(); // 获取复合类型字符串形式值。
    String[] arrayValue = result.getArrayValue(); // 获取复合类型数组形式值，以复合数据类型字段顺序排序。
    result.getStruct(); // 获取复合类型子类型名，按创建顺序排序。
    result.getAttributes(); // 返回自定义类型每列组成类型的对象，对于array类型和Table类型返回的是PgArray,对
于自定义类型，封装的是PGobject，对于其他类型数据存储方式为字符串类型。
    for (String s : arrayValue) {
     System.out.println(s);
    }
    PgArray pgArray = (PgArray) cs.getObject(4);
    ResultSet rs = pgArray.getResultSet();
    Object[] array = (Object[]) pgArray.getArray();
    for (Object element : array) {
```

```
    System.out.println(element);
   }
   cs.close();
   conn.close();
  } catch (ClassNotFoundException | SQLException e) {
   throw new RuntimeException(e);
  }
 }
}
```

结果验证

完整示例可以正确查询到自定类型的数据，完整示例的运行结果如下所示：

```
1
demo
(10,demo10)
(11,demo111)
```

回退方法

不涉及

11.2.3.5 常见问题

1. 现象：绑定自定义类型入参和注册出参后，调用相应的存储过程出现如下异常：
   ```
   ERROR: Function public.test_proc(compfoo, public.compfoo_table, compfoo, public.compfoo_table) does not exist.
     ???No function matches the given name and argument types. You might need to add explicit type casts.
   ```
   原因：出参和入参绑定的自定义类型和存储过程中设定的自定义类型不一致。

   处理方法：出参入参绑定的自定义类型要和存储过程设定的一致。
2. 现象：绑定自定义类型入参和注册出参后，调用相应的存储过程出现如下异常：
   ```
   java.lang.RuntimeException: org.postgresql.util.PSQLException: ?? CallableStatement ????????????? java.sql.Types=1111 ?? 1?????????? java.sql.Types=2002?
   ```
   原因：使用pgArray兼容的场景下，设置了参数enableGaussArrayAndStruct=true，无法同时兼容pgArray和GaussArray。

   处理方法：如果使用pgArray兼容场景，检查删除参数enableGaussArrayAndStruct=true。

11.2.4 批量查询

11.2.4.1 场景概述

本章介绍使用JDBC实现批量查询。

11.2.4.1.1 应用场景

场景描述

当JDBC执行查询操作，若产生大量的查询结果一次性全部返回给JDBC，可能会导致JVM内存溢出。使用批量查询方式，能够指定数据库每次返回给JDBC的数据个数，减少JVM的内存使用。

触发条件

Java应用通过JDBC连接数据库，执行大批量数据的查询。

业务影响

开发人员在数据库操作中，采用正确的事务开启与关闭方式，在批量查询场景中，通过分批次处理数据可有效减少JVM内存占用。该方案能够规避传统全量数据传输模式下，客户端一次性接收所有数据导致的Java应用内存溢出的问题。然而值得注意的是，当遍历结果集时，由于数据库与客户端之间的网络交互次数增加（尤其是每条记录均需独立传输），会引发额外的性能损耗，需在系统设计时进行权衡优化。

11.2.4.1.2 需求目标

业务痛点

在数据密集型业务场景中，传统的查询存在以下问题：

1. 若结果集较大，应用内存难以承载，会造成JVM内存溢出导致查询失败。
2. 一次性获取大量数据，可能会导致网络带宽达到瓶颈，从而影响数据传输的效率。
3. 数据库连接和相关资源持续占用影响系统整体吞吐量。

业务目标

通过批量查询的方式可正常执行查询，避免产生内存溢出。

11.2.4.2 架构原理

核心原理

JDBC批量查询利用GaussDB Kernel的游标遍历功能向服务端查询指定行数的查询结果。在查询时JDBC首先会指定每次从数据库中返回的结果行数，然后数据库会按照指定的行数分批次返回结果数据，直至返回所有的查询结果。

方案优劣势

1. 批量查询每次与数据库交互可返回指定行数的数据，避免因为查询结果集太大导致Java应用出现内存溢出。
2. 批量查询会多次与数据库网络交互，会有一定的性能损耗。

11.2.4.3 前置准备

- JDK版本：1.7及以上。
- 数据库环境：GaussDB 503.0及以上版本。
- JDBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 获取驱动jar包并配置JDK环境”章节。

  数据准备：创建测试表，并插入测试数据。示例如下：

```
gaussdb=# CREATE TABLE tab_test(id int,context varchar(1000),PRIMARY KEY(id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tab_test_pkey" for table "tab_test"
CREATE TABLE
```

```
gaussdb=# INSERT INTO tab_test SELECT generate_series(1,5),repeat('GaussDB Test', 50);
INSERT 0 5
```

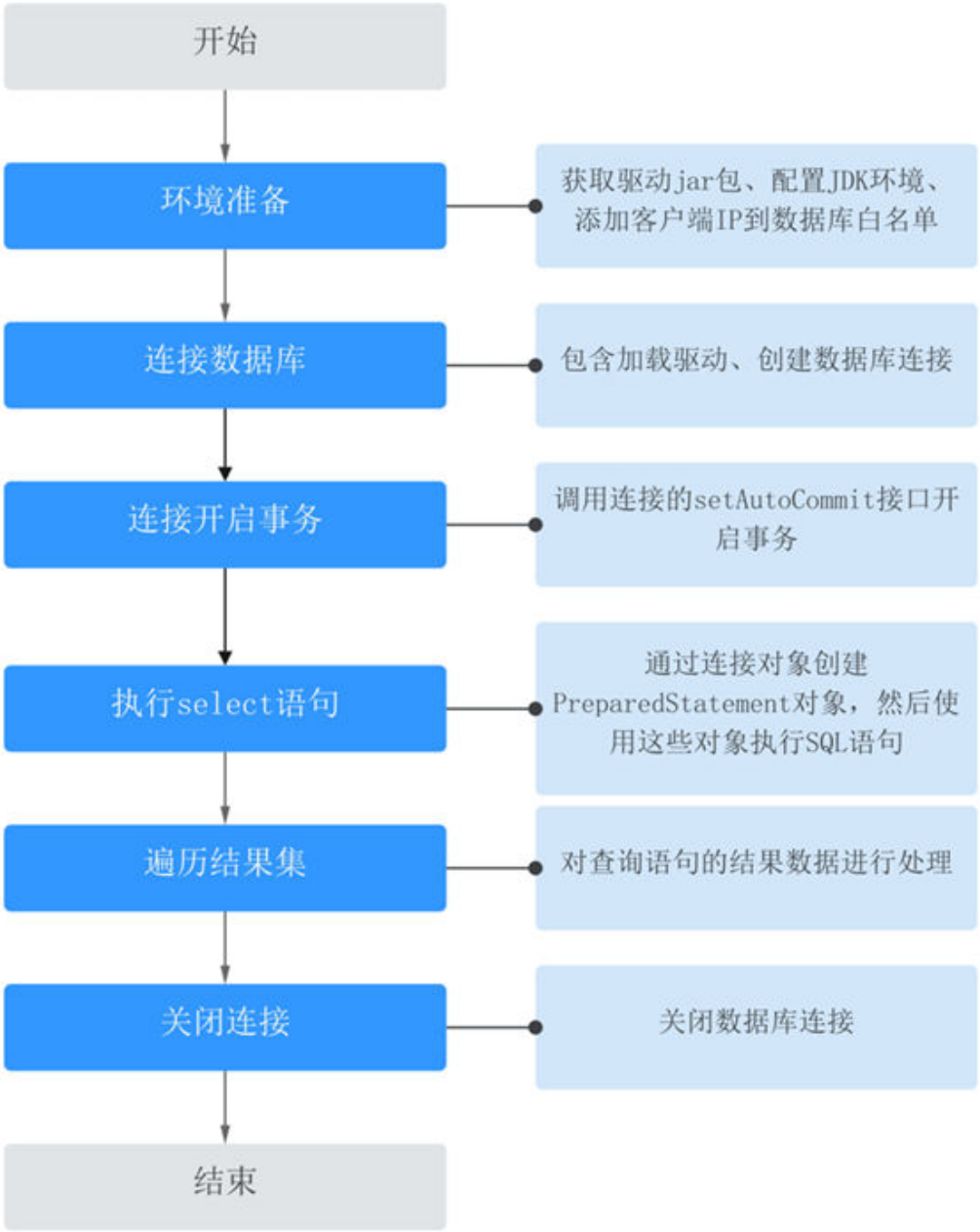
11.2.4.4 操作步骤

11.2.4.4.1 流程总览

JDBC批量查询流程如图11-10所示。

主要包括环境准备、连接数据库、开启事务、执行查询语句、结果集处理以及关闭连接。

图 11-10 JDBC 批量查询流程



版权所有 © 华为云计算技术有限公司

11.2.4.4.2 具体步骤

步骤1 创建Connection对象，与数据库建立连接。

连接串常用参数推荐如下，具体设置方法可参考《开发指南》中“应用程序开发教程 > 基于JDBC开发 > 开发步骤 > 连接数据库 > 连接参数参考”章节。

- connectTimeout：用于连接服务器操作系统的超时值，单位为秒。当JDBC与数据库建立TCP连接的时间超过此值，则连接断开。根据网络情况进行配置。默认值：0，推荐值：2。
- socketTimeout：用于socket读取操作的超时值，单位为秒。如果从服务器读取数据流所花费的时间超过此值，则连接关闭。如果不配置该参数，在数据库进程异常情况下，会导致客户端出现长时间等待，建议根据业务可以接受的SQL执行时间进行配置。默认值：0，无推荐值。
- connectionExtraInfo：表示驱动是否将当前驱动的部署路径、进程属主用户以及url连接配置信息上报到数据库。默认值：false，推荐值：true。
- logger：应用如果使用第三方日志框架记录日志信息，推荐使用实现slf4j接口的第三方日志框架记录JDBC日志，方便异常定位。使用了第三方日志框架的推荐值：Slf4JLogger。

```
String url = "jdbc:gaussdb://$ip:$port/database?
connectTimeout=xx&socketTimeout=xx&connectionExtraInfo=true&logger=Slf4JLogger&autoBalance=t
rue"
Connection conn = DriverManager.getConnection("url",userName,password);
```

步骤2 连接开启事务。

设置自动提交为false后，JDBC内部在执行查询时会预先向数据库下发“BEGIN”主动开启事务。

```
conn.setAutoCommit(false);
```

步骤3 创建PreparedStatement对象，并且设置数据库每次返回的结果行数。

使用setFetchSize方法设置语句级的每次返回结果行数，如果连接串中配置了fetchsize，以setFetchSize方法的值为准。

```
String selectSql = "select * from tab_test";
PreparedStatement preparedStatement = conn.prepareStatement(selectSql);
preparedStatement.setFetchSize(20);
```

步骤4 执行查询，获取结果集。

```
ResultSet resultSet = preparedStatement.executeQuery();
```

步骤5 结果集处理，查看表的第一列数据。

```
while (resultSet.next()) {
    int id = resultSet.getInt(1);
    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
}
```

步骤6 获取结果集列数和列的类型等元信息。

从executeQuery返回的resultSet中获取元数据信息。

```
ResultSetMetaData metaData = resultSet.getMetaData();
System.out.println("结果列：" + metaData.getColumnCount());
System.out.println("类型编号：" + metaData.getColumnType(1));
System.out.println("类型名:" + metaData.getColumnTypeName(1));
System.out.println("列名:" + metaData.getColumnName(1));
```

步骤7 关闭资源。

使用使用try-with-resources打开的文件资源会自动关闭。

```
try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql))
```

步骤8 【可选】异常处理。

在程序运行中需要使用try-catch模块对Exception做异常处理，根据自身业务在异常处理逻辑部分添加对异常的处理逻辑。

```
try {
// 业务代码
} catch (Exception e) {
// 异常处理逻辑
}
```

----结束

11.2.4.4.3 完整示例

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class BatchQuery {
    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        String driver = "com.huawei.gaussdb.jdbc.Driver";
        //  指定数据库sourceURL（$ip、$port、database请根据业务场景进行修改）。
        String sourceURL = "jdbc:gaussdb://$ip:$port/database";
        // 用户名和密码从环境变量中获取。
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        Class.forName(driver);
        return DriverManager.getConnection(sourceURL, userName, password);
    }

    public static void main(String[] args) {
        String selectSql = "select * from tab_test";
        try (Connection conn = getConnection(); PreparedStatement preparedStatement =
conn.prepareStatement(selectSql)) {
            conn.setAutoCommit(false);
            preparedStatement.setFetchSize(3);
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    // 打印部分查询结果。
                    int id = resultSet.getInt(1);
                    System.out.println("row:" + resultSet.getRow() + ",id :" + id);
                }
                ResultSetMetaData metaData = resultSet.getMetaData();
                System.out.println("结果列：" + metaData.getColumnCount());
                System.out.println("类型编号：" + metaData.getColumnType(1));
                System.out.println("类型名:" + metaData.getColumnTypeName(1));
                System.out.println("列名:" + metaData.getColumnName(1));
            }
            conn.commit();
        } catch (ClassNotFoundException | Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

结果验证

完整示例运行结果如下：

```
row:1,id :1
row:2,id :2
row:3,id :3
row:4,id :4
row:5,id :5
```

```
结果列：2
类型编号：4
类型名:int4
列名:id
```

回退方法

若需关闭批量查询功能，则去除连接串的fetchsize参数并且重新设置setFetchSize为默认值0。

11.2.4.5 常见问题

1. 现象：客户使用spring框架时，连接串中设置fetchsize之后，查询过程中存在OutOfMemoryError的报错。

   原因：执行批量查询时没有开启事务，因为一般连接的事务由Spring进行管理，在事务未开启的情况下，数据库会一次将所有数据返回给JDBC

   处理方法：排查业务代码，确认在开启批量查询之前开启了事务。

2. 现象：连接串中设置fetchsize之后，查询过程中存在OutOfMemoryError的报错。

   原因：应用代码的其他调用点内存占用较大，导致JDBC只读取少量数据就发生OutOfMemoryError。

   处理方法：通过jdk工具排查内存占用情况，判断是否是表数据导致的内存溢出。

3. 现象：通过gsql查询表很快，但是通过JDBC查询表速度较慢。

   原因：连接串中配置了fetchsize，如果fetchsize的值比较小，遍历结果集的时候与内核的报文交互次数会过多，导致性能下降。

   处理方法：在执行查询操作时，提前根据表的大小决定是否开启事务，或者调用prepareStatement.setFetchSize()方法调整每次数据库返回的行数，如果调整为0，则一次性返回所有查询结果。

版权所有 © 华为云计算技术有限公司

12 ODBC 最佳实践

12.1 ODBC 最佳实践（分布式）

12.1.1 场景概述

本章介绍使用ODBC驱动实现数据批量插入的场景。

12.1.1.1 应用场景

场景描述

当一次操作中将多条记录写入数据库时，相比逐条插入，批量插入可以减少数据库与应用程序之间的交互次数，降低网络延迟和系统资源消耗，从而显著提高数据写入效率。

本章通过实现批量插入介绍如何使用ODBC驱动连接数据库、使用事务、批量插入和获取结果集列信息等操作。

触发条件

ODBC配置开启UseServerSidePrepare和UseBatchProtocol参数（两个参数均默认开启），应用代码中配置批量绑定参数并初始化批量数据，最终进行批量插入操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。
- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。
- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理

版权所有 © 华为云计算技术有限公司

开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	● 代码简单、直观，易于实现。 ● 单条失败时可精确处理，不影响其他记录。 ● 对数据库和驱动兼容性要求低。	● 网络交互次数多，每次插入都要连接/解析/提交，性能低。 ● 大量记录时容易成为瓶颈。 ● 如果不使用事务，无法保证多条插入的一致性。
批量插入	● 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 ● 可以在一个事务中一次性提交多行，保证原子性。	● 代码复杂度高，需要手动拼接占位符和参数。 ● 单语句错误会回滚所有数据，错误恢复复杂。 ● 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB V500R002C10及以上版本。

12.1.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

使用ODBC驱动初始化目标表，并通过事务的方式批量插入后续查询需要的数据。相比单条SQL逐个插入的场景，减少与数据库交互次数，降低数据库负载。

12.1.2 架构原理

核心原理

在开启UseBatchProtocol和UseServerSidePrepare参数的条件下，ODBC批量处理允许对一条预编译SQL语句复用同一个执行计划，使用U报文一次性携带本次事务中所有待

提交的批量绑定数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

- 网络通信优化

  通过使用U报文，可以一次性携带批量更新的数据，避免了多次使用PBE报文进行通信，从而减少了网络通信的开销。

- 执行效率提升

  由于减少了网络通信的次数和频率，整体的执行效率得到了显著提升，特别是在处理大量数据时。

- 资源利用优化

  通过批量插入可以更有效地利用数据库服务器的资源，减少单次插入不必要的系统开销。

12.1.3 前置准备

- ODBC版本：V500R002C10及以上版本。
- 数据库环境：GaussDB V500R002C10及以上版本。
- ODBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > 开发步骤 > 取源码包、ODBC包以及依赖库”章节。

- 配置ODBC数据源：

  参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > 开发步骤 > 连接数据库 > Linux下配置数据源和Windows下配置数据源”章节。

  以Linux环境为例，推荐配置odbc.ini文件参数如下：

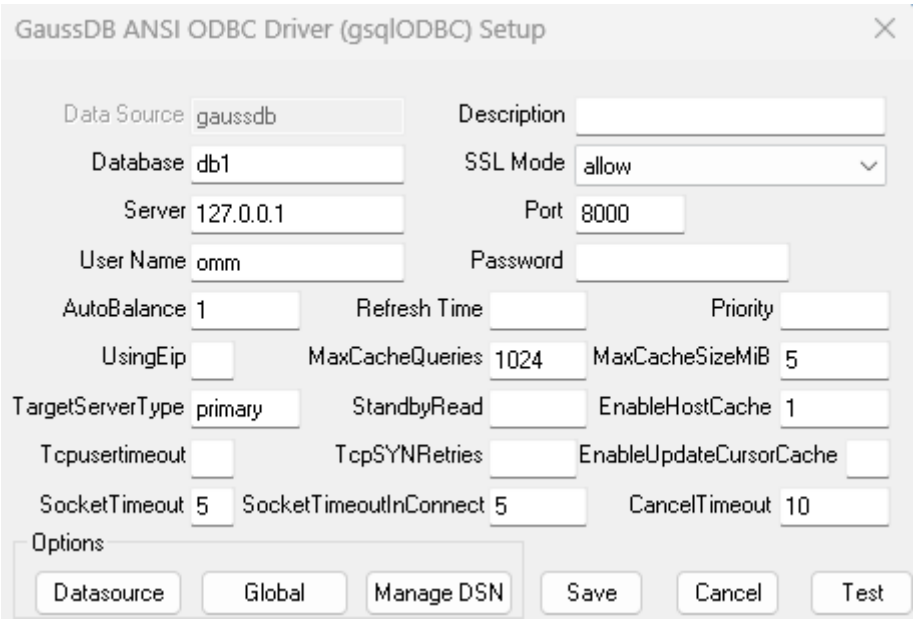
```
[gaussdb]
Driver=GaussMPP
Servername=127.0.0.1  # 数据库Server IP。
Database=db1  # 数据库名。
Username=omm  # 数据库用户名。
Password=******  # 数据库用户密码。
Port=8000  # 数据库侦听端口。
Sslmode=allow  # 该参数用于设置启用SSL加密的方式，allow表示如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。
UseServerSidePrepare=1  # 默认开启，若为1则客户端会使用软解析发送PU/PBE报文，若为0则为硬解析发送Q报文。
UseBatchProtocol=1  # 默认开启，打开批量查询功能。
MaxCacheQueries=1024  # 控制每个连接缓存的预编译语句个数。
MaxCacheSizeMiB=5  # 控制每个连接缓存的预编译语句总大小，在MaxCacheQueries大于0时生效。
ConnSettings=set client_encoding=UTF8  # 设置客户端字符编码，保证和服务端一致。
SocketTimeout=5  # 用于控制客户端与服务端建立连接完全成功后的socket读写超时时间。
TargetServerType=primary  # 设定连接的主机的类型，主机的类型和设定的值一致时才能连接成功，primary表示仅对主备系统中的主节点进行连接。
AutoBalance=1  # ODBC控制负载均衡的开关，连接的数据库版本在506.0以下时，不支持容灾集群负载均衡。
```

说明

- AutoBalance参数仅在GaussDB V500R002C20版本及以上版本支持。
- MaxCacheQueries和MaxCacheSizeMiB参数仅在GaussDB 503.1版本及以上版本支持。
- SocketTimeout参数仅在GaussDB 505.2版本及以上版本支持。
- TargetServerType参数中cluster-primary、cluster-standby和cluster-mainnode仅在GaussDB 506.0版本及以上版本支持，其余参数仅在GaussDB 505.2版本及以上版本支持。

以Windows 环境为例，配置数据源管理器，推荐配置如图12-1所示：

图 12-1 Windows 环境配置数据源管理器



说明

以上数据源配置请以实际业务为准，包括但不限于数据库Server IP，端口Port以及其他连接参数。

12.1.4 操作步骤

12.1.4.1 流程总览

批量绑定插入的流程分为配置连接、配置批量绑定参数以及执行批量插入操作等步骤。应用场景下的主要API涉及SQLSetConnectAttr、SQLSetStmtAttr、SQLPrepare、SQLBindParameter、SQLExecute和SQLRowCount。

详细API介绍参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > ODBC接口参考”章节。

12.1.4.2 具体步骤

步骤1 配置连接

版权所有 © 华为云计算技术有限公司

1. 配置连接超时参数。

   通过SQLSetConnectAttr函数设置连接超时时间：SQL_LOGIN_TIMEOUT参数对应于libpq参数connect_timeout，用于控制客户端连接服务端超时时间，单位为秒，默认值为0，表示该参数不生效。该参数推荐配置，客户可根据实际网络情况配置。

   ```
   SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
   ```

2. 关闭自动提交选项，使用事务提交或回滚。

   通过SQLSetConnectAttr函数将自动提交属性设置为SQL_AUTOCOMMIT_OFF，关闭自动提交，以便使用事务进行提交或回滚。

   ```
   SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
   ```

3. 连接数据库。

   通过SQLConnect函数连接数据库，具体函数原型如下：

   ```
   SQLRETURN  SQLConnect(SQLHDBC        ConnectionHandle,
                         SQLCHAR        *ServerName,
                         SQLSMALLINT    NameLength1,
                         SQLCHAR        *UserName,
                         SQLSMALLINT    NameLength2,
                         SQLCHAR        *Authentication,
                         SQLSMALLINT    NameLength3);
   ```

   指定数据源名ServerName为**前置准备**中的"gaussdb"时，ODBC会自动从配置文件odbc.ini中（Linux环境）或数据源管理器中（Windows环境）获取对应的连接参数。

   获取到数据源后，可以通过连接句柄hdbc访问到所有有关连接数据源的信息，包括程序运行状态、事务处理状态和错误信息等，使用对应的参数连接数据库。

   ```
   SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
   ```

步骤2 配置批量绑定参数

1. 设置批量绑定参数。

   设置批量绑定参数数组的总行数，其中batchCount变量为预期处理的批量插入总行数。

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)batchCount, sizeof(batchCount));
   ```

   设置已处理的行数，其中processRows变量为已处理的批量插入行数。

   ```
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&processRows,
   sizeof(processRows));
   ```

2. 预编译语句及批量绑定。

   使用SQLPrepare函数预编译SQL语句，sql变量包含SQL语句字符串，SQL_NTS表示字符串以空字符结尾。同时使用SQLBindParameter函数绑定参数到预编译的SQL语句，ids和cols对应id和col两列的两个数组，其中id为INT类型，col为VARCHAR类型。

   ```
   SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
   SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0,
   &(ids[0]), 0, bufLenIds);
   SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
   bufLenCols);
   ```

步骤3 执行批量插入操作

1. 执行批量插入操作。

   使用SQLExecute函数执行预编译的SQL语句，进行批量插入操作，通过返回值retcode判断操作是否成功。

   ```
   retcode = SQLExecute(hstmt);
   ```

2. 手动提交或回滚事务。

retcode为SQL_SUCCESS或SQL_SUCCESS_WITH_INFO时，插入操作成功，调用SQLEndTran函数提交事务；当retcode为其他值时，插入操作失败，调用SQLEndTran函数回滚事务。

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT); // 提交事务
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK); // 回滚事务
```

3. 获取批量处理的行数。

   使用SQLRowCount函数获取实际插入的行数，并存储在rowsCount变量中。

```
SQLRowCount(hstmt, &rowsCount);
```

----结束

12.1.4.3 完整示例

```
/*********************************************************************
 * 请在数据源中打开UseBatchProtocol，同时指定数据库中参数support_batch_bind为on。
 * CHECK_ERROR和CHECK_ERROR_VOID的作用是检查并打印错误信息。
 * 此示例将与用户交互式获取DSN、批量绑定的数据量，并将最终数据插入到test_odbc_batch_insert中。
 *********************************************************************/
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
#define CHECK_ERROR(e, s, t, h)                                \
    ({                                                         \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) { \
            fprintf(stderr, "FAILED:\t");                      \
            print_diag(s, h, t);                               \
            goto exit;                                         \
        }                                                      \
    })
#define CHECK_ERROR_VOID(e, s, t, h)                           \
    ({                                                         \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) { \
            fprintf(stderr, "FAILED:\t");                      \
            print_diag(s, h, t);                               \
        }                                                      \
    })
#define BATCH_SIZE 100  // 批量绑定的数据量。
// 打印报错信息。
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle);
// 执行SQL语句。
void Exec(SQLHDBC hdbc, SQLCHAR *sql)
{
    SQLRETURN retcode;                // 返回错误码。
    SQLHSTMT hstmt = SQL_NULL_HSTMT;  // 语句句柄。
    SQLCHAR loginfo[2048];
    // 分配语句句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
        return;
    }
    // 准备语句。
    retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
    sprintf((char *)loginfo, "SQLPrepare log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed");
        return;
    }
    // 执行语句。
    retcode = SQLExecute(hstmt);
    sprintf((char *)loginfo, "SQLExecute stmt log: %s", (char *)sql);
```

```
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLExecute(hstmt) failed");
        return;
    }
    // 释放句柄。
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    sprintf((char *)loginfo, "SQLFreeHandle stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt) failed");
        return;
    }
}
int main()
{
    SQLHENV henv = SQL_NULL_HENV;
    SQLHDBC hdbc = SQL_NULL_HDBC;
    SQLLEN rowsCount = 0;
    int i = 0;
    SQLRETURN retcode;
    SQLCHAR dsn[1024] = {'\0'};
    SQLCHAR loginfo[2048];
    // 分配环境句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle failed");
        goto exit;
    }
    CHECK_ERROR(retcode, "SQLAllocHandle henv", henv, SQL_HANDLE_ENV);
    // 设置ODBC版本。
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER *)SQL_OV_ODBC3, 0);
    CHECK_ERROR(retcode, "SQLSetEnvAttr", henv, SQL_HANDLE_ENV);
    // 分配连接。
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    CHECK_ERROR(retcode, "SQLAllocHandle hdbc", hdbc, SQL_HANDLE_DBC);
    // 设置登录超时。
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_LOGIN_TIMEOUT", hdbc, SQL_HANDLE_DBC);
    // 关闭自动提交，使用事务提交。
    retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    // 连接到数据库。
    retcode = SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
    printf("SQLConnect success\n");
    // 初始化表信息。
    Exec(hdbc, "DROP TABLE IF EXISTS test_odbc_batch_insert");
    Exec(hdbc, "CREATE TABLE test_odbc_batch_insert (id INT PRIMARY KEY, col VARCHAR2(50))");
    // 对于其他SQL操作可以分段进行事务提交。
    retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
    CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
    // 根据用户输入的数据量，构造需要批量插入的数据。
    {
        SQLRETURN retcode;
        SQLHSTMT hstmt = SQL_NULL_HSTMT;
        SQLCHAR *sql = NULL;
        SQLINTEGER *ids = NULL;
        SQLCHAR *cols = NULL;
        SQLLEN *bufLenIds = NULL;
        SQLLEN *bufLenCols = NULL;
        SQLUSMALLINT *operptr = NULL;
        SQLUSMALLINT *statusptr = NULL;
        SQLULEN process = 0;
        // 按列构造每个字段。
        ids = (SQLINTEGER *)malloc(sizeof(ids[0]) * BATCH_SIZE);
        cols = (SQLCHAR *)malloc(sizeof(cols[0]) * BATCH_SIZE * 50);
        // 每个字段中，每一行数据的内存长度。
        bufLenIds = (SQLLEN *)malloc(sizeof(bufLenIds[0]) * BATCH_SIZE);
        bufLenCols = (SQLLEN *)malloc(sizeof(bufLenCols[0]) * BATCH_SIZE);
        if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds) {
```

```
        fprintf(stderr, "FAILED:\tmalloc data memory failed\n");
        goto exit;
    }
    // 对数据进行赋值。
    for (i = 0; i < BATCH_SIZE; i++) {
        ids[i] = i;
        sprintf(cols + 50 * i, "column test value %d", i);
        bufLenIds[i] = sizeof(ids[i]);
        bufLenCols[i] = strlen(cols + 50 * i);
    }
    // 分配语句句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hstmt, SQL_HANDLE_STMT);
    // 预编译语句。
    sql = (SQLCHAR *)"INSERT INTO test_odbc_batch_insert VALUES(?, ?)";
    retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
    CHECK_ERROR(retcode, "SQLPrepare", hstmt, SQL_HANDLE_STMT);
    // 绑定参数。
    retcode = SQLBindParameter(
        hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0, &(ids[0]), 0,
bufLenIds);
    CHECK_ERROR(retcode, "SQLBindParameter 1", hstmt, SQL_HANDLE_STMT);
    retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
    CHECK_ERROR(retcode, "SQLBindParameter 2", hstmt, SQL_HANDLE_STMT);
    // 设置参数数组的总行数。
    retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)BATCH_SIZE,
sizeof(BATCH_SIZE));
    CHECK_ERROR(retcode, "SQLSetStmtAttr", hstmt, SQL_HANDLE_STMT);
    // 设置已处理的行数。
    retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&process,
sizeof(process));
    CHECK_ERROR(retcode, "SQLSetStmtAttr SQL_ATTR_PARAMS_PROCESSED_PTR", hstmt,
SQL_HANDLE_STMT);
    // 执行批量插入。
    retcode = SQLExecute(hstmt);
    // 手动提交事务。
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
        retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
        CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
    }
    // 失败则回滚事务。
    else {
        CHECK_ERROR_VOID(retcode, "SQLExecute", hstmt, SQL_HANDLE_STMT);
        retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
        printf("Transaction rollback\n");
        CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
    }
    // 获取批量处理的行数。
    SQLRowCount(hstmt, &rowsCount);
    sprintf((char *)loginfo, "SQLRowCount : %ld", rowsCount);
    puts(loginfo);
    // 检查插入行数与process处理行数是否一致。
    if (rowsCount != process) {
        sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
        puts(loginfo);
    } else {
        sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
    }
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    CHECK_ERROR(retcode, "SQLFreeHandle", hstmt, SQL_HANDLE_STMT);
}
exit:
    (void)printf("Complete.\n");
    // 断开连接。
    if (hdbc != SQL_NULL_HDBC) {
        SQLDisconnect(hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    }
```

```
    // 释放环境句柄。
    if (henv != SQL_NULL_HENV)
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
    return 0;
}
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;
    SQLSMALLINT recno = 0;
    if (msg)
        printf("%s\n", msg);
    do {
        recno++;
        // 获取诊断信息。
        ret = SQLGetDiagRec(
            htype, handle, recno, (SQLCHAR *)sqlstate, &nativeerror, (SQLCHAR *)message, sizeof(message),
&textlen);
        if (ret == SQL_INVALID_HANDLE)
            printf("Invalid handle\n");
        else if (SQL_SUCCEEDED(ret))
            printf("%s=%s\n", sqlstate, message);
    } while (ret == SQL_SUCCESS);
    if (ret == SQL_NO_DATA && recno == 1)
        printf("No error information\n");
}
```

结果验证

在ODBC成功连接数据库后，批量插入了100条数据，**完整示例**预期结果如下：

```
SQLConnect success
SQLRowCount : 100
Complete.
```

回退方法

通过SQLEndTran接口对事务内的异常操作进行回滚，具体回滚事务方法如下：

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

12.1.5 常见问题

1. Q：ignoreCount指定不入库的数据量后，是否可以选择具体不入库的数据？

   A：可以选择具体不入库的数据。**完整示例**中操作符指针operptr可以为SQL_PARAM_IGNORE或SQL_PARAM_PROCEED，其中SQL_PARAM_IGNORE为不入库的操作，SQL_PARAM_PROCEED为入库的操作，通过指定数据不同的operptr可以选择数据的入库行为。

2. Q：如果批量插入执行失败，所有待插入的数据都会发生回滚，是否能对成功插入的数据进行事务提交，报错的数据进行事务回滚？

   A：不能，插入失败后全部数据均发生事务回滚。推荐在应用代码中将批量的数据划分成更小的批次，通过分批提交并捕获异常。

3. Q：为什么设置了UseServerSidePrepare=0后，执行批量插入了n条，通过SQLRowCount接口获取到的插入条数只有1条？

   A：该结果符合预期。**前置准备**中Linux环境下的odbc.ini文件配置中对于UseServerSidePrepare进行了说明：如果UseServerSidePrepare=1则会发送PU/PBE报文，执行软解析，此时SQLRowCount获取的是最后一次更新的条数；

版权所有 © 华为云计算技术有限公司

UseServerSidePrepare=0则会发送Q报文，执行硬解析，此时SQLRowCount获取的仍是最后一次更新的条数，由于此时Q报文只会发送一条SQL，因此插入条数为1。

12.2 ODBC 最佳实践（集中式）

12.2.1 场景概述

本章介绍使用ODBC驱动实现数据批量插入的场景。

12.2.1.1 应用场景

场景描述

当一次操作中将多条记录写入数据库时，相比逐条插入，批量插入可以减少数据库与应用程序之间的交互次数，降低网络延迟和系统资源消耗，从而显著提高数据写入效率。

本章通过实现批量插入介绍如何使用ODBC驱动连接数据库、使用事务、批量插入和获取结果集列信息等操作。

触发条件

ODBC配置开启UseServerSidePrepare和UseBatchProtocol参数（两个参数均默认开启），应用代码中配置批量绑定参数并初始化批量数据，最终进行批量插入操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。

- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。

- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	● 代码简单、直观，易于实现。 ● 单条失败时可精确处理，不影响其他记录。 ● 对数据库和驱动兼容性要求低。	● 网络交互次数多，每次插入都要连接/解析/提交，性能低。 ● 大量记录时容易成为瓶颈。 ● 如果不使用事务，无法保证多条插入的一致性。
批量插入	● 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 ● 可以在一个事务中一次性提交多行，保证原子性。	● 代码复杂度高，需要手动拼接占位符和参数。 ● 单语句错误会回滚所有数据，错误恢复复杂。 ● 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB V500R002C10及以上版本。

12.2.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

使用ODBC驱动初始化目标表，并通过事务的方式批量插入后续查询需要的数据。相比单条SQL逐个插入的场景，减少与数据库交互次数，降低数据库负载。

12.2.2 架构原理

核心原理

在开启UseBatchProtocol和UseServerSidePrepare参数的条件下，ODBC批量处理允许对一条预编译SQL语句复用同一个执行计划，使用U报文一次性携带本次事务中所有待提交的批量绑定数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

- 网络通信优化

  通过使用U报文，可以一次性携带批量更新的数据，避免了多次使用PBE报文进行通信，从而减少了网络通信的开销。

- 执行效率提升

  由于减少了网络通信的次数和频率，整体的执行效率得到了显著提升，特别是在处理大量数据时。

- 资源利用优化

  通过批量插入可以更有效地利用数据库服务器的资源，减少单次插入不必要的系统开销。

12.2.3 前置准备

- ODBC版本：V500R002C10及以上版本。
- 数据库环境：GaussDB V500R002C10及以上版本。
- ODBC驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > 开发步骤 > 取源码包、ODBC包以及依赖库”章节。
- 配置ODBC数据源：

  参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > 开发步骤 > 连接数据库 > Linux下配置数据源和Windows下配置数据源”章节。

  以Linux环境为例，推荐配置odbc.ini文件参数如下：

```
[gaussdb]
Driver=GaussMPP
Servername=127.0.0.1 # 数据库Server IP。
Database=db1 # 数据库名。
Username=omm # 数据库用户名。
Password=****** # 数据库用户密码。
Port=8000 # 数据库侦听端口。
Sslmode=allow # 该参数用于设置启用SSL加密的方式，allow表示如果数据库服务器要求使用，则可以使用SSL安全加密连接，但不验证数据库服务器的真实性。
UseServerSidePrepare=1 # 默认开启，若为1则客户端会使用软解析发送PU/PBE报文，若为0则为硬解析发送Q报文。
UseBatchProtocol=1 # 默认开启，打开批量查询功能。
MaxCacheQueries=1024 # 控制每个连接缓存的预编译语句个数。
MaxCacheSizeMiB=5 # 控制每个连接缓存的预编译语句总大小，在MaxCacheQueries大于0时生效。
ConnSettings=set client_encoding=UTF8 # 设置客户端字符编码，保证和服务端一致。
SocketTimeout=5 # 用于控制客户端与服务端建立连接完全成功后的socket读写超时时间。
TargetServerType=primary # 设定连接的主机的类型，主机的类型和设定的值一致时才能连接成功，primary表示仅对主备系统中的主节点进行连接。
```

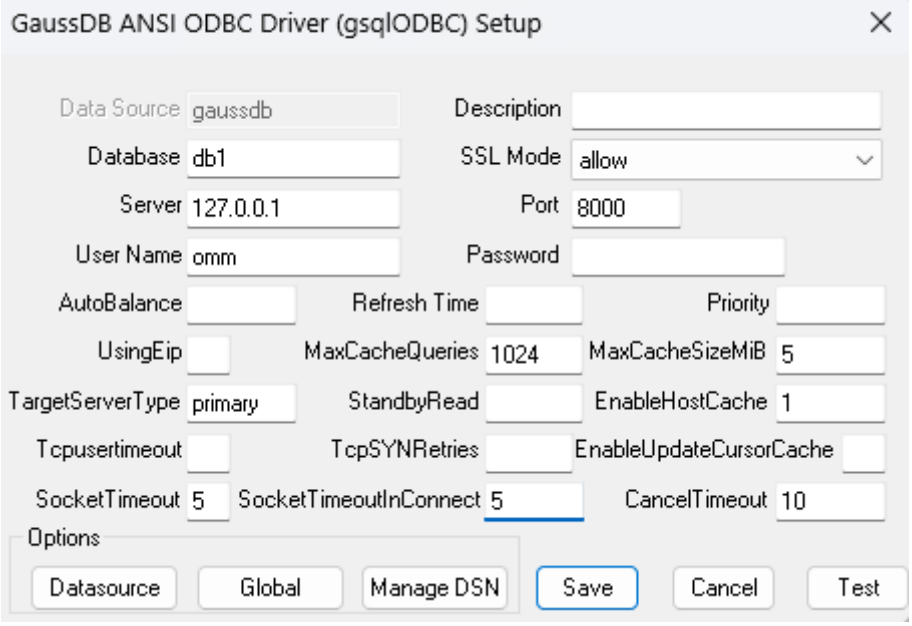
  说明

  - MaxCacheQueries和MaxCacheSizeMiB参数仅在GaussDB 503.1版本及以上版本支持。
  - SocketTimeout参数仅在GaussDB 505.2版本及以上版本支持。
  - TargetServerType参数中cluster-primary、cluster-standby和cluster-mainnode仅在GaussDB 506.0版本及以上版本支持，其余参数仅在GaussDB 505.2版本及以上版本支持。

  以Windows环境为例，配置数据源管理器，推荐配置如图12-2所示：

版权所有 © 华为云计算技术有限公司

图 12-2 Windows 环境配置数据源管理器



说明

以上数据源配置请以实际业务为准，包括但不限于数据库Server IP、端口Port以及其他连接参数。

12.2.4 操作步骤

12.2.4.1 流程总览

批量绑定插入的流程分为配置连接、配置批量绑定参数以及执行批量插入操作等步骤。应用场景下的主要API涉及SQLSetConnectAttr、SQLSetStmtAttr、SQLPrepare、SQLBindParameter、SQLExecute和SQLRowCount。

详细API介绍参考《开发指南》中“应用程序开发教程 > 基于ODBC开发 > ODBC接口参考”章节。

12.2.4.2 具体步骤

步骤1 配置连接。

1. 配置连接超时参数。

   通过SQLSetConnectAttr函数设置连接超时时间：SQL_LOGIN_TIMEOUT参数对应于libpq参数connect_timeout，用于控制客户端连接服务端超时时间，单位为秒，默认值为0，表示该参数不生效。该参数推荐配置，用户可根据实际网络情况进行配置。

   ```
   SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
   ```

2. 关闭自动提交选项，使用事务提交或回滚。

   通过SQLSetConnectAttr函数将自动提交属性设置为SQL_AUTOCOMMIT_OFF，关闭自动提交，以便使用事务进行提交或回滚。

   ```
   SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
   ```


版权所有 © 华为云计算技术有限公司

3. 连接数据库。

   通过SQLConnect函数连接数据库，具体函数原型如下：

```
SQLRETURN  SQLConnect(SQLHDBC        ConnectionHandle,
                      SQLCHAR        *ServerName,
                      SQLSMALLINT    NameLength1,
                      SQLCHAR        *UserName,
                      SQLSMALLINT    NameLength2,
                      SQLCHAR        *Authentication,
                      SQLSMALLINT    NameLength3);
```

   指定数据源名ServerName为**前置准备**中的"gaussdb"时，ODBC会自动从配置文件odbc.ini中（Linux环境）或数据源管理器中（Windows环境）获取对应的连接参数。

   获取到数据源后，可以通过连接句柄hdbc访问到所有有关连接数据源的信息，包括程序运行状态、事务处理状态和错误信息等，使用对应的参数连接数据库。

```
SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
```

步骤2 配置批量绑定参数

1. 设置批量绑定参数。

   设置批量绑定参数数组的总行数，其中batchCount变量为预期处理的批量插入总行数。

```
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)batchCount, sizeof(batchCount));
```

   设置已处理的行数，其中processRows变量为已处理的批量插入行数。

```
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&processRows,
sizeof(processRows));
```

2. 预编译语句及批量绑定。

   使用SQLPrepare函数预编译SQL语句，同时使用SQLBindParameter函数绑定参数到预编译的SQL语句。其中：sql变量包含SQL语句字符串，SQL_NTS表示字符串以空字符结尾，ids和cols对应id和col两列的两个数组，id为INT类型，col为VARCHAR类型。

```
SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0,
&(ids[0]), 0, bufLenIds);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
```

步骤3 执行批量插入操作

1. 执行批量插入操作。

   使用SQLExecute函数执行预编译的SQL语句，进行批量插入操作，通过返回值retcode判断操作是否成功。

```
retcode = SQLExecute(hstmt);
```

2. 手动提交或回滚事务。

   retcode为SQL_SUCCESS或SQL_SUCCESS_WITH_INFO时，插入操作成功，调用SQLEndTran函数提交事务；当retcode为其他值时，插入操作失败，调用SQLEndTran函数回滚事务。

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT); // 提交事务
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK); // 回滚事务
```

3. 获取批量处理的行数。

   使用SQLRowCount函数获取实际插入的行数，并存储在rowsCount变量中。

```
SQLRowCount(hstmt, &rowsCount);
```

----结束

版权所有 © 华为云计算技术有限公司

12.2.4.3 完整示例

```
/*********************************************************************
 * 请在数据源中打开UseBatchProtocol，同时指定数据库中参数support_batch_bind为on。
 * CHECK_ERROR和CHECK_ERROR_VOID的作用是检查并打印错误信息。
 * 此示例将与用户交互式获取DSN、批量绑定的数据量，并将最终数据插入到test_odbc_batch_insert中。
 *********************************************************************/
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
#define CHECK_ERROR(e, s, t, h)                                \
    ({                                                         \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) { \
            fprintf(stderr, "FAILED:\t");                      \
            print_diag(s, h, t);                               \
            goto exit;                                         \
        }                                                      \
    })
#define CHECK_ERROR_VOID(e, s, t, h)                           \
    ({                                                         \
        if (e != SQL_SUCCESS && e != SQL_SUCCESS_WITH_INFO) { \
            fprintf(stderr, "FAILED:\t");                      \
            print_diag(s, h, t);                               \
        }                                                      \
    })
#define BATCH_SIZE 100  // 批量绑定的数据量。
// 打印报错信息。
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle);
// 执行SQL语句。
void Exec(SQLHDBC hdbc, SQLCHAR *sql)
{
    SQLRETURN retcode;                // 返回错误码。
    SQLHSTMT hstmt = SQL_NULL_HSTMT;  // 语句句柄。
    SQLCHAR loginfo[2048];
    // 分配语句句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
        return;
    }
    // 准备语句。
    retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
    sprintf((char *)loginfo, "SQLPrepare log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed");
        return;
    }
    // 执行语句。
    retcode = SQLExecute(hstmt);
    sprintf((char *)loginfo, "SQLExecute stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLExecute(hstmt) failed");
        return;
    }
    // 释放句柄。
    retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    sprintf((char *)loginfo, "SQLFreeHandle stmt log: %s", (char *)sql);
    if (!SQL_SUCCEEDED(retcode)) {
        printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt) failed");
        return;
    }
}
int main()
{
    SQLHENV henv = SQL_NULL_HENV;
```

```
SQLHDBC hdbc = SQL_NULL_HDBC;
SQLLEN rowsCount = 0;
int i = 0;
SQLRETURN retcode;
SQLCHAR dsn[1024] = {'\0'};
SQLCHAR loginfo[2048];
// 分配环境句柄。
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
if (!SQL_SUCCEEDED(retcode)) {
    printf("SQLAllocHandle failed");
    goto exit;
}
CHECK_ERROR(retcode, "SQLAllocHandle henv", henv, SQL_HANDLE_ENV);
// 设置ODBC版本。
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER *)SQL_OV_ODBC3, 0);
CHECK_ERROR(retcode, "SQLSetEnvAttr", henv, SQL_HANDLE_ENV);
// 分配连接。
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
CHECK_ERROR(retcode, "SQLAllocHandle hdbc", hdbc, SQL_HANDLE_DBC);
// 设置登录超时。
retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_LOGIN_TIMEOUT", hdbc, SQL_HANDLE_DBC);
// 关闭自动提交，使用事务提交。
retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, 0);
CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
// 连接到数据库。
retcode = SQLConnect(hdbc, (SQLCHAR *)"gaussdb", SQL_NTS, (SQLCHAR *)"", 0, (SQLCHAR *)"", 0);
CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hdbc, SQL_HANDLE_DBC);
printf("SQLConnect success\n");
// 初始化表信息。
Exec(hdbc, "DROP TABLE IF EXISTS test_odbc_batch_insert");
Exec(hdbc, "CREATE TABLE test_odbc_batch_insert (id INT PRIMARY KEY, col VARCHAR2(50))");
// 对于其他SQL操作可以分段进行事务提交。
retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
// 根据用户输入的数据量，构造出需要批量插入的数据。
{
    SQLRETURN retcode;
    SQLHSTMT hstmt = SQL_NULL_HSTMT;
    SQLCHAR *sql = NULL;
    SQLINTEGER *ids = NULL;
    SQLCHAR *cols = NULL;
    SQLLEN *bufLenIds = NULL;
    SQLLEN *bufLenCols = NULL;
    SQLUSMALLINT *operptr = NULL;
    SQLUSMALLINT *statusptr = NULL;
    SQLULEN process = 0;
    // 按列构造每个字段。
    ids = (SQLINTEGER *)malloc(sizeof(ids[0]) * BATCH_SIZE);
    cols = (SQLCHAR *)malloc(sizeof(cols[0]) * BATCH_SIZE * 50);
    // 每个字段中，每一行数据的内存长度。
    bufLenIds = (SQLLEN *)malloc(sizeof(bufLenIds[0]) * BATCH_SIZE);
    bufLenCols = (SQLLEN *)malloc(sizeof(bufLenCols[0]) * BATCH_SIZE);
    if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds) {
        fprintf(stderr, "FAILED:\tmalloc data memory failed\n");
        goto exit;
    }
    // 对数据进行赋值。
    for (i = 0; i < BATCH_SIZE; i++) {
        ids[i] = i;
        sprintf(cols + 50 * i, "column test value %d", i);
        bufLenIds[i] = sizeof(ids[i]);
        bufLenCols[i] = strlen(cols + 50 * i);
    }
    // 分配语句句柄。
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    CHECK_ERROR(retcode, "SQLSetConnectAttr SQL_ATTR_AUTOCOMMIT", hstmt, SQL_HANDLE_STMT);
    // 预编译语句。
    sql = (SQLCHAR *)"INSERT INTO test_odbc_batch_insert VALUES(?, ?)";
```

```
        retcode = SQLPrepare(hstmt, (SQLCHAR *)sql, SQL_NTS);
        CHECK_ERROR(retcode, "SQLPrepare", hstmt, SQL_HANDLE_STMT);
        // 绑定参数。
        retcode = SQLBindParameter(
            hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, sizeof(ids[0]), 0, &(ids[0]), 0,
bufLenIds);
        CHECK_ERROR(retcode, "SQLBindParameter 1", hstmt, SQL_HANDLE_STMT);
        retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 50, cols, 50,
bufLenCols);
        CHECK_ERROR(retcode, "SQLBindParameter 2", hstmt, SQL_HANDLE_STMT);
        // 设置参数数组的总行数。
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)BATCH_SIZE,
sizeof(BATCH_SIZE));
        CHECK_ERROR(retcode, "SQLSetStmtAttr", hstmt, SQL_HANDLE_STMT);
        // 设置已处理的行数。
        retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, (SQLPOINTER)&process,
sizeof(process));
        CHECK_ERROR(retcode, "SQLSetStmtAttr SQL_ATTR_PARAMS_PROCESSED_PTR", hstmt,
SQL_HANDLE_STMT);
        // 执行批量插入。
        retcode = SQLExecute(hstmt);
        // 手动提交事务。
        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // 失败则回滚事务。
        else {
            CHECK_ERROR_VOID(retcode, "SQLExecute", hstmt, SQL_HANDLE_STMT);
            retcode = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
            printf("Transaction rollback\n");
            CHECK_ERROR(retcode, "SQLEndTran", hdbc, SQL_HANDLE_DBC);
        }
        // 获取批量处理的行数。
        SQLRowCount(hstmt, &rowsCount);
        sprintf((char *)loginfo, "SQLRowCount : %ld", rowsCount);
        puts(loginfo);
        // 检查插入行数与process处理行数是否一致。
        if (rowsCount != process) {
            sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
            puts(loginfo);
        } else {
            sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
        }
        retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        CHECK_ERROR(retcode, "SQLFreeHandle", hstmt, SQL_HANDLE_STMT);
    }
exit:
    (void)printf("Complete.\n");
    // 断开连接。
    if (hdbc != SQL_NULL_HDBC) {
        SQLDisconnect(hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    }
    // 释放环境句柄。
    if (henv != SQL_NULL_HENV)
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
    return 0;
}
void print_diag(char *msg, SQLSMALLINT htype, SQLHANDLE handle)
{
    char sqlstate[32];
    char message[1000];
    SQLINTEGER nativeerror;
    SQLSMALLINT textlen;
    SQLRETURN ret;
    SQLSMALLINT recno = 0;
    if (msg)
        printf("%s\n", msg);
```

```
    do {
        recno++;
        // 获取诊断信息。
        ret = SQLGetDiagRec(
            htype, handle, recno, (SQLCHAR *)sqlstate, &nativeerror, (SQLCHAR *)message, sizeof(message),
&textlen);
        if (ret == SQL_INVALID_HANDLE)
            printf("Invalid handle\n");
        else if (SQL_SUCCEEDED(ret))
            printf("%s=%s\n", sqlstate, message);
    } while (ret == SQL_SUCCESS);
    if (ret == SQL_NO_DATA && recno == 1)
        printf("No error information\n");
}
```

结果验证

在ODBC成功连接数据库后，批量插入了100条数据，完整示例预期结果如下：

```
SQLConnect success
SQLRowCount : 100
Complete.
```

回退方法

通过SQLEndTran接口对事务内的异常操作进行回滚，具体回滚事务方法如下：

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

12.2.5 常见问题

1. Q：ignoreCount指定不入库的数据量后，是否可以选择具体不入库的数据？

   A：可以选择具体不入库的数据。完整示例中操作符指针operptr可以为SQL_PARAM_IGNORE或SQL_PARAM_PROCEED，其中SQL_PARAM_IGNORE为不入库的操作，SQL_PARAM_PROCEED为入库的操作，通过指定数据不同的operptr可以选择数据的入库行为。

2. Q：如果批量插入执行失败，所有待插入的数据都会发生回滚，是否能对成功插入的数据进行事务提交，报错的数据进行事务回滚？

   A：不能，插入失败后全部数据均发生事务回滚。推荐在应用代码中将批量的数据划分成更小的批次，通过分批提交并捕获异常。

3. Q：为什么设置了UseServerSidePrepare=0后，执行批量插入了n条，通过SQLRowCount接口获取到的插入条数只有1条？

   A：该结果符合预期。前置准备中Linux环境下的odbc.ini文件配置中对于UseServerSidePrepare进行了说明：如果UseServerSidePrepare=1则会发送PU/PBE报文，执行软解析，此时SQLRowCount获取的是最后一次更新的条数；UseServerSidePrepare=0则会发送Q报文，执行硬解析，此时SQLRowCount获取的仍是最后一次更新的条数，由于此时Q报文只会发送一条SQL，因此插入条数为1。

13 GO 最佳实践

13.1 GO 最佳实践（分布式）

13.1.1 场景概述

本章介绍使用GO驱动实现数据批量插入。

13.1.1.1 应用场景

场景描述

当一次操作中将多条记录写入数据库时，相比逐条插入，批量插入可以减少数据库与应用程序之间的交互次数，降低网络延迟和系统资源消耗，从而显著提高数据写入效率。

本章通过实现批量插入介绍如何使用GO驱动连接数据库、使用事务、批量插入以及获取结果集列信息等操作。

触发条件

应用代码中生成批量插入SQL语句并绑定参数，在事务中调用Exec接口执行该SQL语句，最终进行批量插入操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。
- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。
- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理

开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	● 代码简单、直观，易于实现。 ● 单条失败时可精确处理，不影响其他记录。 ● 对数据库和驱动兼容性要求低。	● 网络交互次数多，每次插入都要连接/解析/提交，性能低。 ● 大量记录时容易成为瓶颈。 ● 如果不使用事务，无法保证多条插入的一致性。
批量插入	● 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 ● 可以在一个事务中一次性提交多行，保证原子性。	● 代码复杂度高，需要手动拼接占位符和参数。 ● 单语句错误会回滚所有数据，错误恢复复杂。 ● 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB 503.1.0及以上版本。

13.1.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

实现通过GO驱动初始化目标表，并通过事务的方式批量插入后续查询需要的数据。完成批量插入数据后，获取结果集列数据并输出结果信息。

13.1.2 架构原理

核心原理

GO驱动批量处理是一条SQL插入多条记录一次性发送给数据库，并且使用U报文一次性携带本次事务提交所有插入或者更新的数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

- 网络通信优化：通过使用U报文，可以一次性携带批量更新的数据，避免多次使用PBE报文进行通信，从而减少网络通信的开销。
- 执行效率提升：由于网络通信的次数和频率减少，整体的执行效率显著提升，特别是在处理大量数据的场景下。
- 资源利用优化：通过批量插入可以更有效地利用数据库服务器的资源，减少单次插入不必要的系统开销。

13.1.3 前置准备

- Golang版本：1.13及以上。
- 数据库环境：

  GaussDB 503.1.0及以上版本。
- GO驱动环境搭建

  GO驱动环境搭建参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > 开发步骤 > 环境准备”章节。
- 设置代码所需环境变量

  以Linux环境为例：

```
export GOHOSTIP='127.0.0.1'                # IP地址，实际值根据业务调整。
export GOPORT='5432'                       # 端口号，实际值根据业务调整。
export GOUSRNAME='test_user'               # 数据库用户名，实际值根据业务调整。
export GOPASSWD='xxxxxxxx'                 # 数据库用户密码，实际值根据业务调整。
export GODBNAME='gaussdb'                  # 数据库名，实际值根据业务调整。
export GOCONNECT_TIMEOUT='3'               # 连接数据库超时时间，实际值根据业务调整。
export GOSOCKET_TIMEOUT='1'                # 单条SQL超时时间，实际值根据业务调整。
export GOSSLMODE='verify-full'             # 启用SSL加密的方式，实际值根据业务调整。
export GOROOTCERT='certs/cacert.pem'       # 根证书路径，实际值根据业务调整。
export GOSSLKEY='certs/client-key.pem'     # 客户端密钥路径，实际值根据业务调整。
export GOSSLCERT='certs/client-cert.pem'   # 客户端证书路径，实际值根据业务调整。
```

说明

该设置环境变量值的步骤请根据实际修改，如果代码中不通过环境变量获取连接参数的值忽略此步骤。

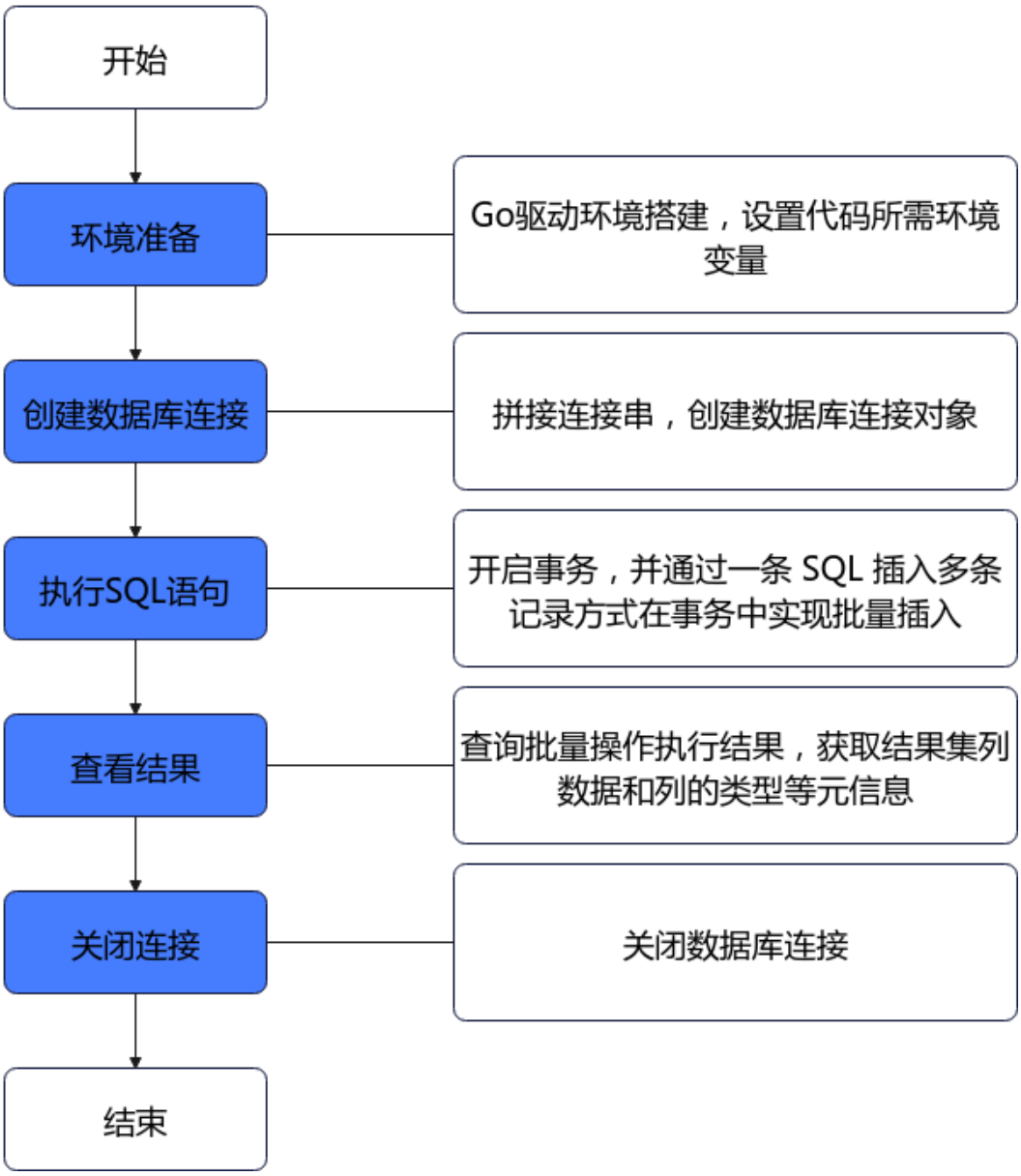
13.1.4 操作步骤

13.1.4.1 流程总览

GO驱动可以执行通过创建数据库连接以及在事务中批量插入数据。

总体流程如图13-1所示。

图 13-1 批量插入流程图



13.1.4.2 具体步骤

步骤1 获取连接参数所需变量值，并拼接创建数据库连接所需连接串。

版权所有 © 华为云计算技术有限公司

说明

- 连接串支持DSN和URL两种格式。
- 数据库连接参数参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > 开发步骤 > 连接数据库”章节。

具体步骤涉及的参数变量值可通过**前置准备**中设置的环境变量获取值，并进行拼接，如以下代码所示。连接参数的值不局限于通过os.Getenv获取环境变量中的值获取，也可通过读取配置文件、写固定值等方式进行设置。

```
hostip := os.Getenv("GOHOSTIP")                    // GOHOSTIP：写入环境变量的IP地址
port := os.Getenv("GOPORT")                        // GOPORT：写入环境变量的port
usrname := os.Getenv("GOUSRNAME")                  // GOUSRNAME：写入环境变量的用户名
passwd := os.Getenv("GOPASSWD")                    // GOPASSWD：写入环境变量的用户密码
dbname := os.Getenv("GODBNAME")                    // GODBNAME：写入环境变量的目标连接数据库名
connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT：写入环境变量的连接数据库超时时间
socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT：写入环境变量的SQL最长的执行时间
rootcertPath := os.Getenv("GOROOTCERT")            // GOROOTCERT：写入环境变量的根证书路径
sslkeyPath := os.Getenv("GOSSLKEY")                // GOSSLKEY：写入环境变量的客户端证书的密钥路径
sslcertPath := os.Getenv("GOSSLCERT")              // GOSSLCERT：写入环境变量的客户端SSL证书路径
sslmode := os.Getenv("GOSSLMODE")                  // GOSSLMODE：写入环境变量的启用SSL加密的方式
```

- 以DSN连接串为例，赋值给变量"dsn"，DSN推荐连接参数以及格式参考如下：

```
dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s
socketTimeout=%s sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master
autoBalance=true",
 hostip,
 port,
 usrname,
 passwd,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

- 以URL连接串为例，赋值给变量"url"，URL推荐连接参数以及URL格式参考如下：

```
url := fmt.Sprintf("gaussdb://%s:%s@%s:%s/%s?connect_timeout=%s&socketTimeout=%s&sslmode=
%s&sslrootcert=%s&sslkey=%s&sslcert=%s&target_session_attrs=master&autoBalance=true",
 usrname,
 passwd,
 hostip,
 port,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

说明

- connect_timeout：该参数用于设置连接服务器操作的超时值，单位为秒。超时时间根据实际网络情况设置，默认为0，即不会超时。
- socket_timeout：该参数用于设置限制单SQL最长的执行时间，单语句执行超过该值则会中断重连。建议根据业务特征进行配置，如果未配置，默认为0，即不会超时。
- sslmode：该参数用于设置启用SSL加密的方式。
- target_session_attrs：指定数据库的连接类型，该参数用于识别主备节点，默认值为any。
- autoBalance：指定负载均衡的策略，为字符串类型。默认值为false。

步骤2 通过**步骤1**中拼接的连接串创建数据库连接对象。

Golang的database/sql标准库提供了sql.Open接口执行创建数据库连接对象，并返回数据库连接对象以及错误信息。

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

- DSN连接串场景：
  ```
  db, err := sql.Open("gaussdb", dsn)
  ```
- URL连接串场景：
  ```
  db, err := sql.Open("gaussdb", url)
  ```

步骤3 通过**步骤2**创建的数据库连接对象创建事务对象。

数据库连接对象提供了Begin接口执行创建事务对象，并返回事务对象以及错误信息。

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

以下为创建事务对象，并以变量"tx"进行接收返回的事务对象。

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

步骤4 通过**步骤3**中创建的事务对象执行批量插入。

以Exec接口为例，具体参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Tx”章节。

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

通过**步骤3**中创建的事务对象"tx"，调用Exec接口批量插入employee表为例，其中包括拼接批量插入SQL以及传入批量插入所需的值，数据量根据用户实际为准。

```
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := ""
for i, u := range employee {
 placeholders += "(?, ?)"
 if i < len(employee)-1 {
  placeholders += ","
 }
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
result, err := tx.Exec(stmt, vals...)
```

步骤5 【可选】通过**步骤3**中创建的事务对象进行回滚事务。

事务对象提供了Rollback接口进行回滚事务。

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

当事务中出错时，可以通过调用**步骤3**中创建的事务对象"tx"的Rollback接口进行回滚事务。

```
tx.Rollback()
```

步骤6 通过**步骤3**中创建的事务对象提交事务。

事务对象提供了Commit接口进行提交事务。

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

通过步骤3中创建的事务对象"tx"的Commit接口进行提交事务。

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

步骤7 【可选】通过步骤2创建的数据库连接对象执行查询。

数据库对象和事务对象均提供了查询接口，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type DB以及type Tx”章节中提供的查询接口。

以调用步骤2中创建的数据库对象"db"提供的Query接口为例，查询步骤4批量插入的结果，并通过变量"rows"接收查询结果对象。

```
rows, err := db.Query("SELECT id, name, created_at FROM users;"
```

步骤8 【可选】通过步骤7获取的结果对象获取结果集列数和列名列表。

步骤7中结果对象为Golang的database/sql中的Rows类型，该类型提供了Columns接口进行返回查询结果集列名的列表，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Rows”章节。

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

以调用步骤7中获取的查询结果对象提供的接口Columns获取查询到的列名列表，并赋值到变量"columns"中为例：

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

结果集的列数可以通过len函数计算"columns"个数获取。

```
len(columns)
```

步骤9 【可选】通过步骤7获取的结果对象获取结果集列的类型等元信息。

步骤7中结果对象为Golang的database/sql中的Rows类型，该类型提供了ColumnTypes接口来返回查询结果集列名的列表，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Rows”章节。

```
func (rs *Rows) ColumnTypes() ([]*ColumnType, error)
```

通过步骤7中获取的查询结果获取列信息。

以调用步骤7中获取的查询结果对象提供的接口ColumnTypes获取查询到的列类型对象列表[]*ColumnType，并赋值到变量"columnTypes"中为例：

```
columnTypes, err := rows.ColumnTypes()
```

应用代码可以通过遍历列类型对象列表"columnTypes"获取返回结果列的类型信息。

type ColumnType提供以下接口描述数据库表中的列类型信息：

表 13-1 type ColumnType 接口的常用方法

方法	描述	返回值类型
(ci *ColumnType)DatabaseTypeName()	返回列类型的数据库系统名称。返回空字符串表示该驱动类型名字并未被支持。	error
(ci *ColumnType)DecimalSize()	返回小数类型的范围和精度。返回值ok的值为false时，说明给定的类型不可用或者不支持。	precision, scale int64, ok boolean

方法	描述	返回值类型
(ci *ColumnType)Length()	返回数据列类型长度。返回值ok的值为false时，说明给定的类型是非可变长度。	length int64, ok boolean
(ci *ColumnType)ScanType()	返回一种Go类型，该类型能够在Rows.scan进行扫描时使用。	reflect.Type
(ci *ColumnType)Name()	返回数据列的名字。	string

步骤10 通过步骤2创建的数据库连接对象执行关闭连接。

数据库连接对象提供了Close接口进行数据库连接的关闭。

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

执行以下语句关闭步骤2创建的数据库连接对象"db"。

```
db.Close()
```

----结束

13.1.4.3 完整示例

以DSN连接串，初始化employee表后批量插入数据，并获取插入结果输出结果集列信息为例：

```
// main.go
package main

import (
 "database/sql"
 "fmt"
 _ "gitee.com/opengauss/openGauss-connector-go-pq"
 "log"
 "os"
)

func main() {
 // 创建数据库对象
 hostip := os.Getenv("GOHOSTIP")                  // GOHOSTIP：写入环境变量的IP地址
 port := os.Getenv("GOPORT")                      // GOPORT：写入环境变量的port
 usrname := os.Getenv("GOUSRNAME")                // GOUSRNAME：写入环境变量的用户名
 passwd := os.Getenv("GOPASSWD")                  // GOPASSWD：写入环境变量的用户密码
 dbname := os.Getenv("GODBNAME")                  // GODBNAME：写入环境变量的目标连接数据库名
 connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT：写入环境变量的连接数据库超时时间
 socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT：写入环境变量的SQL最长的执行时间
 rootcertPath := os.Getenv("GOROOTCERT")          // GOROOTCERT：写入环境变量的根证书路径
 sslkeyPath := os.Getenv("GOSSLKEY")              // GOSSLKEY：写入环境变量的客户端证书的密钥路径
 sslcertPath := os.Getenv("GOSSLCERT")            // GOSSLCERT：写入环境变量的客户端SSL证书路径
 sslmode := os.Getenv("GOSSLMODE")                // GOSSLMODE：写入环境变量的启用SSL加密的方式

dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s socketTimeout=%s "+
   "sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
   hostip,
   port,
```

```
  usrname,
  passwd,
  dbname,
  connect_timeout,
  socket_timeout,
  sslmode,
  rootcertPath,
  sslkeyPath,
  sslcertPath,
)

db, err := sql.Open("gaussdb", dsn)
if err != nil {
 panic(err)
}
defer db.Close()
err = db.Ping()
if err != nil {
 panic(err)
}
fmt.Println("connect success.")
// 开启事务
tx, err := db.Begin()
if err != nil {
 log.Fatal(err)
 return
}
// 初始化数据表
_, err = tx.Exec("drop table if exists employee;")
if err != nil {
 fmt.Println("drop table employee failed, err:", err)
 err = tx.Rollback() // 出错则回滚事务
 return
}
fmt.Println("drop table employee success.")
_, err = tx.Exec("create table employee (id SERIAL PRIMARY KEY, name varchar(20), age int, created_at
TIMESTAMP DEFAULT CURRENT_TIMESTAMP);")
if err != nil {
 fmt.Println("create table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("create table employee success.")
// 批量插入数据
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := "(?, ?)"
for _, u := range employee {
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
_, err = tx.Exec(stmt, vals...)

if err != nil {
 fmt.Println("batch insert into table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("batch insert into table employee success.")
// 提交事务
err = tx.Commit()
if err != nil {
```

```
 fmt.Println("commit failed, err:", err)
 err = tx.Rollback()
 log.Fatal(err)
 return
}
fmt.Println("commit success.")
// 获取结果集列信息
rows, err := db.Query("SELECT id, name, created_at FROM employee;")
if err != nil {
 fmt.Println("query table employee failed, err:", err)
 return
}
columns, err := rows.Columns()
if err != nil {
 fmt.Println("get query rows columns failed, err:", err)
 return
}
fmt.Println("列数：", len(columns))
fmt.Println("列名list：", columns)
fmt.Println("--------------------------")
// 获取列的类型信息
columnTypes, err := rows.ColumnTypes()
if err != nil {
 fmt.Println("get query rows ColumnTypes failed, err:", err)
 return
}

for _, ct := range columnTypes {
 fmt.Println("列名：", ct.Name())
 fmt.Println("数据库类型：", ct.DatabaseTypeName())
 length, ok := ct.Length()
 if ok {
  fmt.Println("长度：", length)
 }
 precision, scale, ok := ct.DecimalSize()
 if ok {
  fmt.Printf("精度/小数位数：%d/%d\n", precision, scale)
 }
 nullable, ok := ct.Nullable()
 if ok {
  fmt.Println("是否可为空：", nullable)
 }
 fmt.Println("Go 类型：", ct.ScanType())
 fmt.Println("-----")
}
}
```

结果验证

以下为完整示例的执行结果：

```
connect success.
drop table employee success.
create table employee success.
batch insert into table employee success.
commit success.
列数： 3
列名list： [id name created_at]
--------------------------
列名： id
数据库类型： INT4
Go 类型： int32
-----
列名： name
数据库类型： VARCHAR
长度： 20
Go 类型： string
-----
列名： created_at
```

```
数据库类型： TIMESTAMP
Go 类型： time.Time
-----
```

预期结果包含以下内容：

1. 拼接数据库连接串，通过sql.Open创建连接对象，并通过db.Ping()方法检测是否成功连接数据库。
2. 开启事务，并在事务中初始化测试表employee。
3. 在事务中构造employee测试数据，并生成批量插入SQL，通过事务对象提供的接口Exec，绑参并发送报文到数据库执行SQL。
4. 批量插入执行成功后，通过Commit接口提交事务。（如果执行失败通过Rollback接口回滚事务）。
5. 通过数据连接对象"db"提供的Query接口查询批量插入执行结果，并通过结果对象"rows"提供的Columns接口获取所有列名列表以及结果对象"rows"提供的ColumnTypes接口获取结果集列的元数据信息。

回退方法

通过事务对象的Rollback接口，对事务内的操作进行回滚。

13.1.5 常见问题

1. 构建批量插入语句时必须注意SQL注入风险。

   通过使用占位符和参数绑定，而非直接拼接用户输入的值。多行VALUES语句通过占位符"?"和参数列表插入数据，从而避免注入攻击。所有的动态数据都应作为参数传递，例如使用数据库对象*DB的接口Prepare或Exec等接口传入可变参数形式。
2. 批量插入执行失败问题定位。

   在批量插入操作中，如果某条记录插入失败，数据库通常只会返回整体错误信息（如主键冲突、外键约束或数据类型不匹配），而无法具体指出是哪一条记录导致了问题。如果一条SQL语句中包含多条记录，其中一条记录插入失败，整个事务可能会失败（如果启用了错误忽略机制则不会失败）。因此，如果需要定位出错的数据行，常用做法是将批量分成更小的批次或逐行插入并捕获错误
3. 内存使用增加。

   构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或GO驱动的最大SQL长度限制，或者触发其他参数限制，进而引发错误或性能问题。

13.2 GO 最佳实践（集中式）

13.2.1 场景概述

本章介绍使用GO驱动实现数据批量插入。

13.2.1.1 应用场景

场景描述

当一次操作中将多条记录写入数据库时，相比逐条插入，批量插入可以减少数据库与应用程序之间的交互次数，降低网络延迟和系统资源消耗，从而显著提高数据写入效率。

本章通过实现批量插入介绍如何使用GO驱动连接数据库、使用事务、批量插入以及获取结果集列信息等操作。

触发条件

应用代码中生成批量插入SQL语句并绑定参数，在事务中调用Exec接口执行该SQL语句，最终进行批量插入操作。

业务影响

- 降低网络交互成本。

  将多条INSERT合并为一次批量操作，可显著降低客户端与数据库之间的网络往返次数，进而实现提升整体吞吐量，以及减轻网络拥塞对性能的影响。

- 提高数据处理效率

  逐条插入方式需要数据库对每条SQL都进行语法解析和执行计划生成，批量插入只需一次解析和一次执行计划，避免了多次重复工作，节省了CPU周期和内存分配时间。

- 降低系统资源使用开销

  逐条插入方式通常会触发一次事务提交或至少一次事务日志写入，而批量插入可在一次事务内插入多条记录，显著减少提交次数，降低事务日志压力和事务管理开销。减少网络包处理、事务管理、日志写入和行格式转换的总次数，有助于减轻数据库服务器的CPU负载和内存临时空间占用，从而将更多资源留给核心查询与计算操作。

- 评估内存的使用

  构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或驱动的最大SQL长度限制，或者触发其他参数限制，从而引发错误或性能问题。

批量插入和逐条插入的对比：

方式	优点	缺点
逐条插入	● 代码简单、直观，易于实现。 ● 单条失败时可精确处理，不影响其他记录。 ● 对数据库和驱动兼容性要求低。	● 网络交互次数多，每次插入都要连接/解析/提交，性能低。 ● 大量记录时容易成为瓶颈。 ● 如果不使用事务，无法保证多条插入的一致性。

方式	优点	缺点
批量插入	● 大幅减少网络往返和 SQL 解析次数，插入吞吐量显著提高。 ● 可以在一个事务中一次性提交多行，保证原子性。	● 代码复杂度高，需要手动拼接占位符和参数。 ● 单语句错误会回滚所有数据，错误恢复复杂。 ● 占位符数量受限，批次大小需控制。

适用版本

仅支持GaussDB V500R002C10及以上版本。

13.2.1.2 需求目标

业务痛点

大量数据插入时，逐条插入会产生大量网络请求、系统资源消耗占用过高以及数据库服务端需要反复解析相似语句导致业务性能下降，因此引用批量插入优化以上痛点。

业务目标

实现通过GO驱动初始化目标表，并通过事务的方式批量插入后续查询需要的数据。完成批量插入数据后，获取结果集列数据并输出结果信息。

13.2.2 架构原理

核心原理

GO驱动批量处理是一条SQL插入多条记录一次性发送给数据库，并且使用U报文一次性携带本次事务提交所有插入或者更新的数据，只需要进行一次网络连接的建立和数据交互即可完成批量操作。

方案优势

- 网络通信优化

  通过使用U报文，可以一次性携带批量更新的数据，避免多次使用PBE报文进行通信，从而减少网络通信的开销。

- 执行效率提升

  由于网络通信的次数和频率减少，整体的执行效率显著提升，特别是在处理大量数据的场景下。

- 资源利用优化

  通过批量插入可以更有效地利用数据库服务器的资源，减少单次插入不必要的系统开销。

13.2.3 前置准备

- Golang版本：1.13及以上。
- 数据库环境：GaussDB V500R002C10及以上版本。
- Go驱动环境搭建：

  参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > 开发步骤 > 环境准备”章节。
- 设置代码所需环境变量：

  以Linux环境为例：

```
export GOHOSTIP='127.0.0.1'                # IP地址，实际值根据业务调整。
export GOPORT='5432'                       # 端口号，实际值根据业务调整。
export GOUSRNAME='test_user'               # 数据库用户名，实际值根据业务调整。
export GOPASSWD='xxxxxxxx'                 # 数据库用户密码，实际值根据业务调整。
export GODBNAME='gaussdb'                  # 数据库名，实际值根据业务调整。
export GOCONNECT_TIMEOUT='3'               # 连接数据库超时时间，实际值根据业务调整。
export GOSOCKET_TIMEOUT='1'                # 单条SQL超时时间，实际值根据业务调整。
export GOSSLMODE='verify-full'             # 启用SSL加密的方式，实际值根据业务调整。
export GOROOTCERT='certs/cacert.pem'       # 根证书路径，实际值根据业务调整。
export GOSSLKEY='certs/client-key.pem'     # 客户端密钥路径，实际值根据业务调整。
export GOSSLCERT='certs/client-cert.pem'   # 客户端证书路径，实际值根据业务调整。
```

说明

该设置环境变量值的步骤请根据实际修改，如果代码中不通过环境变量获取连接参数的值忽略此步骤。

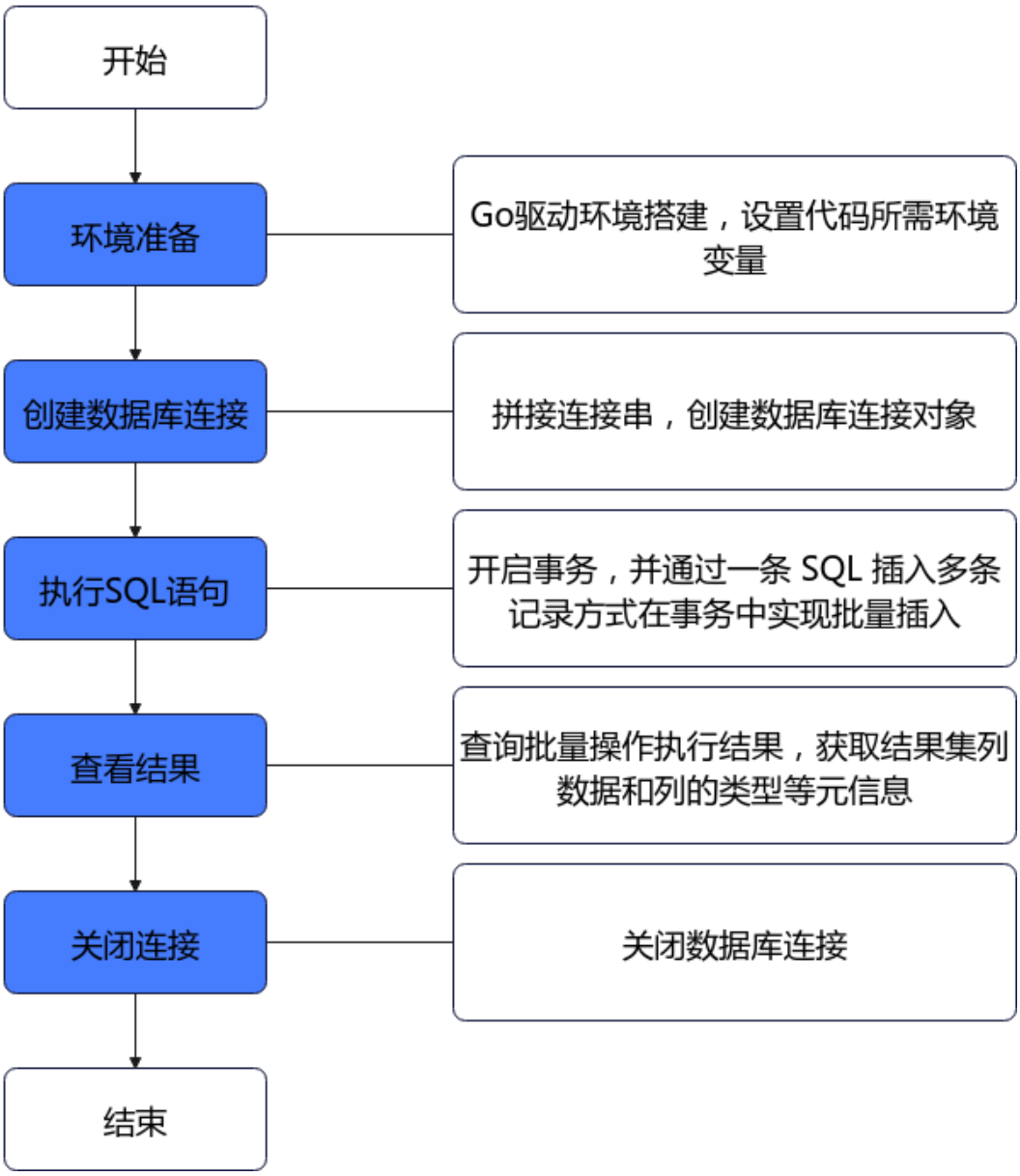
13.2.4 操作步骤

13.2.4.1 流程总览

GO驱动可以执行通过创建数据库连接以及在事务中批量插入数据。

总体流程如图13-2所示。

图 13-2 批量插入流程图



13.2.4.2 具体步骤

步骤1 获取连接参数所需变量值，并拼接创建数据库连接所需连接串。

版权所有 © 华为云计算技术有限公司

说明

- 连接串支持DSN和URL两种格式。
- 数据库连接参数参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > 开发步骤 > 连接数据库”章节。

具体步骤涉及的参数变量值可通过**前置准备**中设置的环境变量获取值，并进行拼接，如以下代码所示。连接参数的值不局限于通过os.Getenv获取环境变量中的值获取，也可通过读取配置文件、写固定值等方式进行设置。

```
hostip := os.Getenv("GOHOSTIP")                  // GOHOSTIP：写入环境变量的IP地址
port := os.Getenv("GOPORT")                      // GOPORT：写入环境变量的port
usrname := os.Getenv("GOUSRNAME")                // GOUSRNAME：写入环境变量的用户名
passwd := os.Getenv("GOPASSWD")                  // GOPASSWD：写入环境变量的用户密码
dbname := os.Getenv("GODBNAME")                  // GODBNAME：写入环境变量的目标连接数据库名
connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT：写入环境变量的连接数据库超时时间
socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT：写入环境变量的SQL最长的执行时间
rootcertPath := os.Getenv("GOROOTCERT")          // GOROOTCERT：写入环境变量的根证书路径
sslkeyPath := os.Getenv("GOSSLKEY")              // GOSSLKEY：写入环境变量的客户端证书的密钥路径
sslcertPath := os.Getenv("GOSSLCERT")            // GOSSLCERT：写入环境变量的客户端SSL证书路径
sslmode := os.Getenv("GOSSLMODE")                // GOSSLMODE：写入环境变量的启用SSL加密的方式
```

- 以DSN连接串为例，赋值给变量"dsn"，DSN推荐连接参数以及格式参考如下：

```
dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s
socketTimeout=%s sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
 hostip,
 port,
 usrname,
 passwd,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

- 以URL连接串为例，赋值给变量"url"，推荐连接参数以及URL格式参考如下：

```
url := fmt.Sprintf("gaussdb://%s:%s@%s:%s/%s?connect_timeout=%s&socketTimeout=%s&sslmode=
%s&sslrootcert=%s&sslkey=%s&sslcert=%s&target_session_attrs=master",
 usrname,
 passwd,
 hostip,
 port,
 dbname,
 connect_timeout,
 socket_timeout,
 sslmode,
 rootcertPath,
 sslkeyPath,
 sslcertPath,
)
```

说明

- connect_timeout：该参数用于设置连接服务器操作的超时值，单位：秒。超时时间根据实际网络情况设置，默认为0，即不会超时。
- socket_timeout：该参数用于设置限制单SQL最长的执行时间，单语句执行超过该值则会中断重连。建议根据业务特征进行配置，如果未配置，默认为0，即不会超时。
- sslmode：该参数用于设置启用SSL加密的方式。
- target_session_attrs：指定数据库的连接类型，该参数用于识别主备节点，默认值为any。

步骤2 通过**步骤1**中拼接的连接串创建数据库连接对象。

Golang的database/sql标准库提供了sql.Open接口执行创建数据库连接对象，并返回数据库连接对象以及错误信息。

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

- DSN连接串场景：
  ```
  db, err := sql.Open("gaussdb", dsn)
  ```
- URL连接串场景：
  ```
  db, err := sql.Open("gaussdb", url)
  ```

步骤3 通过**步骤2**创建的数据库连接对象创建事务对象。

数据库连接对象提供了Begin接口执行创建事务对象，并返回事务对象以及错误信息。

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

以下为创建事务对象，并以变量"tx"进行接收返回的事务对象。

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

步骤4 通过**步骤3**中创建的事务对象执行批量插入。

以Exec接口为例，具体参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Tx”章节。

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

通过**步骤3**中创建的事务对象"tx"，调用Exec接口批量插入employee表为例，其中包括拼接批量插入SQL以及传入批量插入所需的值，数据量根据用户实际为准。

```
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := ""
for i, u := range employee {
 placeholders += "(?, ?)"
 if i < len(employee)-1 {
  placeholders += ","
 }
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
result, err := tx.Exec(stmt, vals...)
```

步骤5 【可选】通过**步骤3**中创建的事务对象进行回滚事务。

事务对象提供了Rollback接口进行回滚事务。

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

当事务中出错时，可以通过调用**步骤3**中创建的事务对象"tx"的Rollback接口进行回滚事务。

```
tx.Rollback()
```

步骤6 通过**步骤3**中创建的事务对象提交事务。

事务对象提供了Commit接口进行提交事务。

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

通过**步骤3**中创建的事务对象"tx"的Commit接口进行提交事务。

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

步骤7 【可选】通过步骤2创建的数据库连接对象执行查询。

数据库对象和事务对象均提供了查询接口，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type DB以及type Tx”章节中提供的查询接口。

以调用步骤2中创建的数据库对象"db"提供的Query接口为例，查询步骤4批量插入的结果，并通过变量"rows"接收查询结果对象。

```
rows, err := db.Query("SELECT id, name, created_at FROM users;"
```

步骤8 【可选】通过步骤7获取的结果对象获取结果集列数和列名列表。

步骤7中结果对象为Golang的database/sql中的Rows类型，该类型提供了Columns接口来返回查询结果集列名的列表，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Rows”章节。

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

以调用步骤7中获取的查询结果对象提供的接口Columns获取查询到的列名列表，并赋值到变量"columns"中为例：

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

结果集的列数可以通过len函数计算"columns"个数获取。

```
len(columns)
```

步骤9 【可选】通过步骤7获取的结果对象获取结果集列的类型等元信息。

步骤7中结果对象为Golang的database/sql中的Rows类型，该类型提供了ColumnTypes接口来返回查询结果集列名的列表，参考《开发指南》中“应用程序开发教程 > 基于Go驱动开发 > Go接口参考 > type Rows”章节。

```
func (rs *Rows) ColumnTypes() ([]*ColumnType, error)
```

通过步骤7中获取的查询结果获取列信息。

以调用步骤7中获取的查询结果对象提供的接口ColumnTypes获取查询到的列类型对象列表[]*ColumnType，并赋值到变量"columnTypes"中为例：

```
columnTypes, err := rows.ColumnTypes()
```

应用代码可以通过遍历列类型对象列表"columnTypes"获取返回结果列的类型信息。

type ColumnType提供以下接口描述数据库表中的列类型信息：

表 13-2 type ColumnType 接口的常用方法

方法	描述	返回值类型
(ci *ColumnType)DatabaseTypeName()	返回列类型的数据库系统名称。返回空字符串表示该驱动类型名字并未被支持。	error
(ci *ColumnType)DecimalSize()	返回小数类型的范围和精度。返回值ok的值为false时，说明给定的类型不可用或者不支持。	precision, scale int64, ok boolean

方法	描述	返回值类型
(ci *ColumnType)Length()	返回数据列类型长度。返回值ok的值为false时，说明给定的类型是非可变长度。	length int64, ok boolean
(ci *ColumnType)ScanType()	返回一种Go类型，该类型能够在Rows.scan进行扫描时使用。	reflect.Type
(ci *ColumnType)Name()	返回数据列的名字。	string

步骤10 通过步骤2创建的数据库连接对象执行关闭连接。

数据库连接对象提供了Close接口进行数据库连接的关闭。

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

执行以下语句关闭步骤2创建的数据库连接对象。

```
db.Close()
```

----结束

13.2.4.3 完整示例

以DSN连接串，初始化employee表后批量插入数据，并获取插入结果输出结果集列信息为例：

```
// main.go
package main

import (
 "database/sql"
 "fmt"
 _ "gitee.com/opengauss/openGauss-connector-go-pq"
 "log"
 "os"
)

func main() {
 // 创建数据库对象
 hostip := os.Getenv("GOHOSTIP")                  // GOHOSTIP：写入环境变量的IP地址
 port := os.Getenv("GOPORT")                      // GOPORT：写入环境变量的port
 usrname := os.Getenv("GOUSRNAME")                // GOUSRNAME：写入环境变量的用户名
 passwd := os.Getenv("GOPASSWD")                  // GOPASSWD：写入环境变量的用户密码
 dbname := os.Getenv("GODBNAME")                  // GODBNAME：写入环境变量的目标连接数据库名
 connect_timeout := os.Getenv("GOCONNECT_TIMEOUT") // GOCONNECT_TIMEOUT：写入环境变量的连接数据库超时时间
 socket_timeout := os.Getenv("GOSOCKET_TIMEOUT")   // GOSOCKET_TIMEOUT：写入环境变量的SQL最长的执行时间
 rootcertPath := os.Getenv("GOROOTCERT")          // GOROOTCERT：写入环境变量的根证书路径
 sslkeyPath := os.Getenv("GOSSLKEY")              // GOSSLKEY：写入环境变量的客户端证书的密钥路径
 sslcertPath := os.Getenv("GOSSLCERT")            // GOSSLCERT：写入环境变量的客户端SSL证书路径
 sslmode := os.Getenv("GOSSLMODE")                // GOSSLMODE：写入环境变量的启用SSL加密的方式

dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s connect_timeout=%s socketTimeout=%s " +
   "sslmode=%s sslrootcert=%s sslkey=%s sslcert=%s target_session_attrs=master",
   hostip,
   port,
```

```
  usrname,
  passwd,
  dbname,
  connect_timeout,
  socket_timeout,
  sslmode,
  rootcertPath,
  sslkeyPath,
  sslcertPath,
)

db, err := sql.Open("gaussdb", dsn)
if err != nil {
 panic(err)
}
defer db.Close()
err = db.Ping()
if err != nil {
 panic(err)
}
fmt.Println("connect success.")
// 开启事务
tx, err := db.Begin()
if err != nil {
 log.Fatal(err)
 return
}
// 初始化数据表
_, err = tx.Exec("drop table if exists employee;")
if err != nil {
 fmt.Println("drop table employee failed, err:", err)
 err = tx.Rollback() // 出错则回滚事务
 return
}
fmt.Println("drop table employee success.")
_, err = tx.Exec("create table employee (id SERIAL PRIMARY KEY, name varchar(20), age int, created_at
TIMESTAMP DEFAULT CURRENT_TIMESTAMP);")
if err != nil {
 fmt.Println("create table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("create table employee success.")
// 批量插入数据
employee := []struct {
 Name string
 Age  uint8
}{{Name: "zhangsan", Age: 21}, {Name: "lisi", Age: 22}, {Name: "zhaowu", Age: 23}}

batchSql := "INSERT INTO employee (name, age) VALUES "
vals := []interface{}{}

placeholders := "(?, ?)"
for _, u := range employee {
 vals = append(vals, u.Name, u.Age)
}

stmt := batchSql + placeholders
_, err = tx.Exec(stmt, vals...)

if err != nil {
 fmt.Println("batch insert into table employee failed, err:", err)
 err = tx.Rollback()
 return
}
fmt.Println("batch insert into table employee success.")
// 提交事务
err = tx.Commit()
if err != nil {
```

```
  fmt.Println("commit failed, err:", err)
  err = tx.Rollback()
  log.Fatal(err)
  return
 }
 fmt.Println("commit success.")
 // 获取结果集列信息
 rows, err := db.Query("SELECT id, name, created_at FROM employee;")
 if err != nil {
  fmt.Println("query table employee failed, err:", err)
  return
 }
 columns, err := rows.Columns()
 if err != nil {
  fmt.Println("get query rows columns failed, err:", err)
  return
 }
 fmt.Println("列数：", len(columns))
 fmt.Println("列名list：", columns)
 fmt.Println("--------------------------")
 // 获取列的类型信息
 columnTypes, err := rows.ColumnTypes()
 if err != nil {
  fmt.Println("get query rows ColumnTypes failed, err:", err)
  return
 }

 for _, ct := range columnTypes {
  fmt.Println("列名：", ct.Name())
  fmt.Println("数据库类型：", ct.DatabaseTypeName())
  length, ok := ct.Length()
  if ok {
   fmt.Println("长度：", length)
  }
  precision, scale, ok := ct.DecimalSize()
  if ok {
   fmt.Printf("精度/小数位数：%d/%d\n", precision, scale)
  }
  nullable, ok := ct.Nullable()
  if ok {
   fmt.Println("是否可为空：", nullable)
  }
  fmt.Println("Go 类型：", ct.ScanType())
  fmt.Println("-----")
 }
}
```

结果验证

以下为完整示例的执行结果：

```
connect success.
drop table employee success.
create table employee success.
batch insert into table employee success.
commit success.
列数： 3
列名list： [id name created_at]
--------------------------
列名： id
数据库类型： INT4
Go 类型： int32
-----
列名： name
数据库类型： VARCHAR
长度： 20
Go 类型： string
-----
列名： created_at
```

```
数据库类型： TIMESTAMP
Go 类型： time.Time
-----
```

预期结果包含以下内容：

1. 拼接数据库连接串，通过sql.Open创建连接对象，并通过db.Ping()方法检测是否成功连接数据库。
2. 开启事务，并在事务中初始化测试表employee。
3. 在事务中构造employee测试数据，并生成批量插入SQL，通过事务对象提供的接口Exec，绑参并发送报文到数据库执行SQL。
4. 批量插入执行成功后，通过Commit接口提交事务。（如果执行失败通过Rollback接口回滚事务）。
5. 通过数据连接对象"db"提供的Query接口查询批量插入执行结果，并通过结果对象"rows"提供的Columns接口获取所有列名列表以及结果对象"rows"提供的ColumnTypes接口获取结果集列的元数据信息。

回退方法

通过事务对象的Rollback接口，对事务内的操作进行回滚。

13.2.5 常见问题

1. 构建批量插入语句时必须注意SQL注入风险。

   通过使用占位符和参数绑定，而非直接拼接用户输入的值。多行VALUES语句通过占位符"?"和参数列表插入数据，从而避免注入攻击。所有的动态数据都应作为参数传递，例如使用数据库对象*DB的接口Prepare或Exec等接口传入可变参数形式。

2. 批量插入执行失败问题定位。

   在批量插入操作中，如果某条记录插入失败，数据库通常只会返回整体错误信息（如主键冲突、外键约束或数据类型不匹配），而无法具体指示是哪一条记录导致了问题发生。如果一条SQL语句中包含多条记录，其中一条记录插入失败，整个事务可能会失败（如果启用了错误忽略机制则不会失败）。因此，如果需要定位出错的数据行，常用做法是将批量分成更小的批次或逐行插入并捕获错误。

3. 内存使用增加。

   构建批量插入的SQL语句时，如果数据量过大，会导致内存占用显著增加。尤其是当使用字符串拼接方式构建SQL语句时，内存消耗可能会急剧上升。过大的批量处理可能导致超出数据库或GO驱动的最大SQL长度限制，或者触发其他参数限制，进而引发错误或性能问题。

14 索引设计最佳实践

14.1 索引设计最佳实践（分布式）

14.1.1 场景概述

应用场景

在查询百万级别的大表时，没有索引或者索引过于单一会造成查询效率低下，为了解决业务痛点，应该创建合适的索引来避免效率低下，此最佳实践主要介绍了对于百万级别的大表，无索引和有索引性能对比，单列索引和复合索引的性能对比。

需求目标

创建合适的索引，避免全表扫描，提高查询效率。

14.1.2 前置准备

在现有的数据库里面创建表并插入百万级别数据。

机器配置为8核CPU32GB内存。

```
--创建test_table表。
gaussdb=# CREATE TABLE test_table ( id SERIAL PRIMARY KEY,name VARCHAR(100),email
VARCHAR(100),created_at TIMESTAMP);
CREATE TABLE

--向表中插入100万条数据。
gaussdb=# INSERT INTO test_table (name,email,created_at) select 'User_' || i,'User_' || i ||
'@example.com',NOW() - (i * INTERVAL '1 minute') FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000

--创建表。
gaussdb=# CREATE TABLE sales_records (record_id BIGSERIAL PRIMARY KEY,region_id INT NOT
NULL,store_id INT NOT NULL,product_id INT NOT NULL,sale_date DATE NOT NULL,amount DECIMAL(12,2)
NOT NULL,is_refund BOOLEAN DEFAULT false);

--插入200万条数据。
gaussdb=# INSERT INTO sales_records (region_id, store_id, product_id, sale_date, amount) SELECT
(random()*9)::INT + 1,(random()*99)::INT + 1,(random()*499)::INT + 1,current_date - (random()*1095)::INT,
(random()*9900)::DECIMAL + 100 FROM generate_series(1,2000000);
INSERT 0 2000000
```


版权所有 © 华为云计算技术有限公司

14.1.3 操作步骤

无索引和有索引性能对比

步骤1 使用root用户登录数据库。

步骤2 查看test_table表执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |         operation          | A-time  | A-rows | E-rows | Peak Memory | A-width | E-width |     E-cost
s
----+----------------------------+---------+--------+--------+-------------+---------+---------+------------
-------
  1 | ->  Seq Scan on test_table | 382.457 |      0 |   1989 | 19KB        |         |     148 | 0.000..136
44.650
(1 row)

          Predicate Information (identified by plan id)
-----------------------------------------------------------------
   1 --Seq Scan on test_table
         Filter: ((email)::text = 'user_500000@example.com'::text)
         Rows Removed by Filter: 1000000
(3 rows)

      ====== Query Summary =====
----------------------------------------
 Datanode executor start time: 0.037 ms
 Datanode executor run time: 382.544 ms
 Datanode executor end time: 0.017 ms
 Planner runtime: 0.391 ms
 Query Id: 1945836514001883020
 Total runtime: 382.624 ms
(6 rows)
```

从执行结果来看，执行时间需要382.624ms（全表扫描）。

步骤3 创建索引。

```
gaussdb=# CREATE INDEX idx_test_table_email ON test_table(email);
CREATE INDEX
```

步骤4 再次查看test_table表执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |                        operation                        | A-time | A-rows | E-rows | Peak Memory | A-
width | E-width |   E-costs
----+---------------------------------------------------------+--------+--------+--------+-------------+---
------+---------+-------------
  1 | ->  Index Scan using idx_test_table_email on test_table | 0.163  |      0 |      1 | 75KB        |
      |      46 | 0.000..8.268
(1 row)

            Predicate Information (identified by plan id)
-----------------------------------------------------------------
   1 --Index Scan using idx_test_table_email on test_table
         Index Cond: ((email)::text = 'user_500000@example.com'::text)
(2 rows)

      ====== Query Summary =====
----------------------------------------
 Datanode executor start time: 0.063 ms
 Datanode executor run time: 0.190 ms
 Datanode executor end time: 0.013 ms
 Planner runtime: 0.936 ms
 Query Id: 1945836514001885197
 Total runtime: 0.293 ms
(6 rows)

 ====== Query Others =====
```

```
---------------------------
 Bypass: Yes
(1 row)
```

添加索引后，通过与无索引时执行计划的对比，查询时间从原来的382.624ms缩短到0.293 ms。

----结束

单列索引和复合索引的性能对比

步骤1 使用root用户登录数据库。

步骤2 创建单列索引。

```
gaussdb=# CREATE INDEX idx_region ON sales_records(region_id);
CREATE INDEX
gaussdb=# CREATE INDEX idx_store ON sales_records(store_id);
CREATE INDEX
```

步骤3 查看执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=23.482..37.694 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 37.945 ms
(3 rows)
```

从执行结果来看，执行时间需要37.945 ms。

步骤4 创建复合索引。

```
gaussdb=# CREATE INDEX idx_region_store ON sales_records(region_id, store_id);
CREATE INDEX
```

步骤5 再次查看执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=4.266..6.390 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 6.616 ms
(3 rows)
```

通过对单列索引和复合索引执行计划的对比，查询时间从原来的37.945 ms缩短到6.616 ms。

----结束

14.2 索引设计最佳实践（集中式）

14.2.1 场景概述

应用场景

在查询百万级别的大表时，没有索引或者索引过于单一会造成查询效率低下，为了解决业务痛点，应该创建合适的索引来避免效率低下，此最佳实践主要介绍了对于百万级别的大表，无索引和有索引性能对比，单列索引和复合索引的性能对比。

需求目标

创建合适的索引，避免全表扫描，提高查询效率。

14.2.2 前置准备

在现有的数据库里面创建表并插入百万级别数据。

机器配置为8核CPU32GB内存。

```
--创建test_table表。
gaussdb=# CREATE TABLE test_table ( id SERIAL PRIMARY KEY,name VARCHAR(100),email
VARCHAR(100),created_at TIMESTAMP);
CREATE TABLE

--向表中插入100万条数据。
gaussdb=# INSERT INTO test_table (name,email,created_at) select 'User_' || i,'User_' || i ||
'@example.com',NOW() - (i * INTERVAL '1 minute') FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000

--创建表。
gaussdb=# CREATE TABLE sales_records (record_id BIGSERIAL PRIMARY KEY,region_id INT NOT
NULL,store_id INT NOT NULL,product_id INT NOT NULL,sale_date DATE NOT NULL,amount DECIMAL(12,2)
NOT NULL,is_refund BOOLEAN DEFAULT false);

--插入200万条数据。
gaussdb=# INSERT INTO sales_records (region_id, store_id, product_id, sale_date, amount) SELECT
(random()*9)::INT + 1,(random()*99)::INT + 1,(random()*499)::INT + 1,current_date - (random()*1095)::INT,
(random()*9900)::DECIMAL + 100 FROM generate_series(1,2000000);
INSERT 0 2000000
```

14.2.3 操作步骤

无索引和有索引性能对比

步骤1 使用root用户登录数据库。

步骤2 查看test_table表执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |          operation           | A-time  | A-rows | E-rows | Peak Memory | A-width | E-width |     E-cost
s
----+------------------------------+---------+--------+--------+-------------+---------+---------+-----------
-------
  1 | ->  Seq Scan on test_table | 382.457 |      0 |   1989 | 19KB        |         |     148 | 0.000..136
44.650
(1 row)

           Predicate Information (identified by plan id)
-------------------------------------------------------------------
   1 --Seq Scan on test_table
         Filter: ((email)::text = 'user_500000@example.com'::text)
         Rows Removed by Filter: 1000000
(3 rows)

       ====== Query Summary =====
-----------------------------------------
 Datanode executor start time: 0.037 ms
 Datanode executor run time: 382.544 ms
 Datanode executor end time: 0.017 ms
 Planner runtime: 0.391 ms
 Query Id: 1945836514001883020
 Total runtime: 382.624 ms
(6 rows)
```

从执行结果来看，执行时间需要382.624ms（全表扫描）。

步骤3 创建索引。

```
gaussdb=# CREATE INDEX idx_test_table_email ON test_table(email);
CREATE INDEX
```

步骤4 再次查看test_table表执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_table WHERE email = 'user_500000@example.com';
 id |                        operation                        | A-time | A-rows | E-rows | Peak Memory | A-
width | E-width |   E-costs
----+---------------------------------------------------------+--------+--------+--------+-------------+---
------+---------+--------------
  1 | ->  Index Scan using idx_test_table_email on test_table | 0.163  |      0 |      1 | 75KB        |
      |       46 | 0.000..8.268
(1 row)

             Predicate Information (identified by plan id)
--------------------------------------------------------------------------
   1 --Index Scan using idx_test_table_email on test_table
         Index Cond: ((email)::text = 'user_500000@example.com'::text)
(2 rows)

       ====== Query Summary =====
----------------------------------------
 Datanode executor start time: 0.063 ms
 Datanode executor run time: 0.190 ms
 Datanode executor end time: 0.013 ms
 Planner runtime: 0.936 ms
 Query Id: 1945836514001885197
 Total runtime: 0.293 ms
(6 rows)

 ====== Query Others =====
----------------------------
 Bypass: Yes
(1 row)
```

添加索引后，通过与无索引时执行计划的对比，查询时间从原来的382.624ms缩短到0.293 ms。

----结束

单列索引和复合索引的性能对比

步骤1 使用root用户登录数据库。

步骤2 创建单列索引。

```
gaussdb=# CREATE INDEX idx_region ON sales_records(region_id);
CREATE INDEX
gaussdb=# CREATE INDEX idx_store ON sales_records(store_id);
CREATE INDEX
```

步骤3 查看执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)  (actual time=23.482..37.694 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 37.945 ms
(3 rows)
```

从执行结果来看，执行时间需要37.945 ms。

步骤4 创建复合索引。

```
gaussdb=# CREATE INDEX idx_region_store ON sales_records(region_id, store_id);
CREATE INDEX
```

步骤5 再次查看执行计划。

```
gaussdb=# EXPLAIN ANALYZE SELECT * FROM sales_records WHERE region_id = 5 AND store_id = 42;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0) (actual time=4.266..6.390 rows=2221 loops=1)
   Node/s: All datanodes
 Total runtime: 6.616 ms
(3 rows)
```

通过对单列索引和复合索引执行计划的对比，查询时间从原来的37.945 ms缩短到6.616 ms。

----结束

15 表设计最佳实践

15.1 表设计最佳实践（分布式）

15.1.1 场景概述

应用场景

表设计初期需要考虑如下场景：

- 数据库表结构设计阶段需要优化查询效率。
- 处理海量数据时需提升维护效率。
- 频繁进行数据更新操作的表设计。
- 需要平衡存储成本与查询性能的场景。

15.1.2 架构原理

核心原理

- 数据类型优化：根据整数 > 浮点数 > numeric的优先级进行选择。
- 索引平衡机制：查询加速与更新成本的权衡。
- 分区存储策略：逻辑统一+物理分散的存储方式。
- 存储引擎特性：Ustore支持Inplace-Update，Astore为Append-Only。

方案优势

- 查询性能提升：通过索引设计以及分区策略减少数据的扫描量。
- 存储成本优化：合理选择数据类型可节省30%以上的空间优化。
- 维护效率提高：分区独立维护降低业务影响。
- 并发处理增强：多分区并行访问提升业务吞吐量。

版权所有 © 华为云计算技术有限公司

15.1.3 前置准备

- 确认业务场景特征：
  - 数据量级预估（数据量较大时，考虑使用分区表）。
  - 读写比例分析（根据读写比例设计存储引擎和索引）。
- 环境检查：
  检查track_counts/track_activities参数状态。
- 工具准备：
  - 数据库客户端工具。
  - 性能监控工具。

15.1.4 操作步骤

表设计主要包含分布方案及分布键设计、数据类型设计、分区策略、约束配置、索引设计和存储参数调优等步骤。

分布方式及分布键设计

- 表的分布方式的选择一般遵循以下原则：

分布方式	描述	适用场景
Hash	表数据通过Hash方式散列到集群中的所有DN上。	数据量较大的表。
Replication	集群中每一个DN都有一份全量表数据。	维度表、数据量较小的表。
Range	表数据对指定列按照范围进行映射，分布到对应DN。	用户需要自定义分布规则的场景。
List	表数据对指定列按照具体值进行映射，分布到对应DN。	用户需要自定义分布规则的场景。

- 分布键选择
  表的分布键选取至关重要，如果分布键选择不当，可能会导致数据产生倾斜，导致查询时，I/O负载集中在部分DN上，影响整体查询性能。因此，在确定分布表的分布策略之后，需要对表数据进行倾斜性检查，以确保数据的均匀分布。分布键的选择一般需要遵循以下原则：
  - 作为分布键的字段取值应比较离散，以便数据能在各个DN上均匀分布。当单个字段无法满足离散条件时，可以考虑使用多个字段一起作为分布键。一般情况下，可以考虑选择表的主键作为分布键。例如，在人员信息表中选择证件号码作为分布键。
  - 在满足第一条原则的情况下，请不要选取在查询中存在常量过滤条件的字段作为分布键。
  - 在满足前两条原则的情况，尽量选择查询中的关联条件作为分布键。当关联条件作为分布键时，JOIN任务的相关数据都分布在DN本地，将极大减少DN之间的数据流动代价。

以下为分布方式及分布键设计的一个简单示例，介绍了如何指定分布方式及分布键设计的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表，选择分布列和分布方式。

```
--REPLICATION分布。
gaussdb=# CREATE TABLE tb_t1(c1 int, c2 int)DISTRIBUTE BY REPLICATION;

--HASH分布。
gaussdb=# CREATE TABLE tb_t2(c1 int,c2 int)DISTRIBUTE BY HASH(c1);

--RANGE分布。
gaussdb=# CREATE TABLE tb_t3(c1 int,c2 int)
DISTRIBUTE BY RANGE(c1)(
    SLICE s1 VALUES LESS THAN (100),
    SLICE s2 VALUES LESS THAN (200),
    SLICE s3 VALUES LESS THAN (MAXVALUE)
);
gaussdb=# CREATE TABLE tb_t4(c1 int,c2 int)
DISTRIBUTE BY RANGE(c1)(
    SLICE s1 START (1) END (100),
    SLICE s2 START (100) END (200),
    SLICE s3 START (200) END (MAXVALUE)
);

--LIST分布。
gaussdb=# CREATE TABLE tb_t5(id INT,name VARCHAR(20),country VARCHAR(30))
DISTRIBUTE BY LIST(country)(
    SLICE s1 VALUES ('China'),
    SLICE s2 VALUES ('USA'),
    SLICE s3 VALUES (DEFAULT)
);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----结束

数据类型设计

在数据类型设计时，基于查询效率的考虑，一般遵循以下原则：

- 尽量选择高效的数据类型。选择数值类型时，在满足业务精度的情况下，选择数据类型的优先级从高到低依次为整数、浮点数以及numeric。
- 当多个表存在逻辑关系时，表示同一个含义的字段应该使用相同的数据类型。
- 对于字符串数据，需要根据实际情况选择定长或者变长字符类型。对于varchar以及char等类型，需要指定一个最大长度。这个长度的设置既要考虑可以存储所有可能的数据，也要考虑存储空间避免不必要的资源浪费。

在进行字段设计时，需要根据数据特征选择相应的数据类型。GaussDB支持的数据类型请参见《开发指南》中“SQL参考 > 数据类型”章节。

分区策略

- **简介**

  分区是数据库优化的一种技术，通过将大表按规则拆分成多个分区以提高查询和维护效率。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的，这些分区的数据可以存储在不同的存储设备上。GaussDB目前支持范围分区、哈希分区以及列表分区。分区表的优缺点如下：

  – 优点：

    - 提高查询性能：通过减少扫描的数据量使查询性能有显著地提升。

    - 优化存储：通过把不同分区存放到不同的存储介质上，来平衡性能和成本。
    - 增加可维护性：分区表的维护操作（数据清理、重建索引）可以从分区的粒度来进行，减少对整个系统的影响。
    - 提高并发性：分区表可以提高并发性，因为多个分区可以并行处理。例如，多个查询可以同时访问不同的分区，而不会相互干扰。
  - 缺点：
    - 内存资源占用：分区表使用内存大致为（分区数 * 3 / 1024）MB，当分区数太多导致内存不足时，会间接导致性能急剧下降。
    - 分区策略复杂性：制定和实施合适的分区策略需要技术知识和经验。如果分区策略选择不当，可能会导致数据分布不均衡，进一步影响性能。
    - 备份恢复的复杂性：虽然可以单独备份和恢复分区，但这也同时要求需要更加细致的备份策略及管理工作。
- **适合使用分区表的场景**
  - 提高查询性能：表的数据量大，且具有某些特性的数据在其中某个场景中经常会用到，可以通过减少查询时扫描数据量来提高性能。如：经常以月、季度、年为单位做分析的表。
  - 平衡性能和成本：表的数据量过大，需要将冷数据（不常访问的数据）移动到低成本存储，而将热数据（频繁访问的数据）保留在高性能存储上。
  - 大数据量表管理：表的数据量过大，需要在多个存储介质上存储的场景。
- **设计阶段注意事项**
  - 分区键选择：

    选择分区表的分区键是一个重要的设计决策，因为它直接影响到数据库的性能、可维护性以及数据管理的效率。
    - 优化查询：选择常用查询的分区键。例如，经常根据日期进行查询的表，可以选择日期字段为分区键。
    - 数据分布：选择分区键时，考虑数据的分布情况。避免某些分区数据量过大而其他分区数据量过小的情况。
    - 分区数量和管理：控制分区数量，避免创建过多分区，分区过多会导致管理复杂度上升和性能下降。
  - 分区类型选择：
    - 范围分区：适合分区键的值是连续的值（如时间字段）。
    - 列表分区：适用于离散的但是类型不多的分区键（如地区、状态码等字段）。
    - 哈希分区：用于均匀的分散数据（如用户id）。

以下为分区策略设计的一个简单示例，介绍了如何声明指定分区方式的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表，选择分区。

```
--范围分区。
gaussdb=# CREATE TABLE tb_t1(id INT,info VARCHAR(20))
```

```
PARTITION BY RANGE (id) (
    PARTITION p1 START(1) END(600) EVERY(200),
    PARTITION p2 START(600) END(800),
    PARTITION pmax START(800) END(MAXVALUE)
);
gaussdb=# CREATE TABLE tb_t2(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (200),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

--列表分区表示例。
gaussdb=# CREATE TABLE tb_t3(NAME VARCHAR ( 50 ), area VARCHAR ( 50 ))
PARTITION BY LIST (area) (
    PARTITION p1 VALUES ('bj'),
    PARTITION p2 VALUES ('sh'),
    PARTITION pdefault VALUES (DEFAULT)
);


--哈希分区表示例。
gaussdb=# CREATE TABLE tb_t4(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;
gaussdb=# CREATE TABLE tb_t5(c1 int) PARTITION BY HASH(C1)(
    PARTITION pa,
    PARTITION pb,
    PARTITION pc
);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----结束

约束配置

- 建议创建约束时，在命名上可明确标识约束类型和约束所在的表名，例如主键约束命名包含PK、表名、构成字段的方式命名。
- DEFAULT约束：需要谨慎选择DEFAULT约束，如果能在业务层面补全字段值，则不建议使用DEFAULT约束。
- 给明确不存在NULL值的字段加上NOT NULL约束，优化器会在特定场景下对其进行自动优化。

以下为添加约束的一个简单示例，介绍了如何指定约束的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表，为表添加约束。

```
--非空约束。
gaussdb=# CREATE TABLE tb_t1(id int not null,name varchar(50));

--唯一约束。
gaussdb=# CREATE TABLE tb_t2(id int UNIQUE,name varchar(50));
gaussdb=# CREATE TABLE tb_t3(id int, name varchar(50),CONSTRAINT unq_t3_id UNIQUE(id));

--主键约束。
gaussdb=# CREATE TABLE tb_t4(id int PRIMARY KEY, name varchar(50));
gaussdb=# CREATE TABLE tb_t5(
    id int,
    name varchar(50),
    CONSTRAINT pk_person5_id PRIMARY KEY(id)
);
```

```
--检查约束。
gaussdb=# CREATE TABLE tb_t6(name varchar(50),age int CHECK(age > 0 AND age < 200));
gaussdb=# CREATE TABLE tb_t7(
    name varchar(50),
    age int,
    CONSTRAINT chk_t6_age CHECK (age > 0 AND age < 200)
);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5,tb_t6,tb_t7;
```

----结束

索引设计

索引可以提高数据访问速度，但同时也增加了插入、更新和删除操作的处理时间。所以是否要为表增加索引，索引建立在哪些字段上，是创建索引前必须要考虑的问题。建立索引建议遵循以下原则：

- 在经常使用连接的列上创建索引，可以加快连接速度。
- 在经常需要排序的列上创建索引，因为索引列已经排序，加快排序查询的速度。
- 对于WHERE子句经常使用的列上进行创建索引，加快条件的判断速度。
- 复合索引可以包含多个列，但列数越多，索引体积越大，维护开销越高。
- 在频繁更新的字段上避免使用索引，索引会增加数据更新的维护成本，因此尽量避免在频繁更新的字段上创建索引。
- 索引定义里的所有函数和操作符都必须是immutable类型的，即它们的结果必须只能依赖于它们的输入参数，而不受任何外部的影响（如另外一个表的内容或者当前时间）。这个限制可以确保该索引的行为是定义良好的。要在一个索引上或WHERE中使用用户定义函数，请把它标记为immutable类型函数。
- 分区表索引分为LOCAL索引与GLOBAL索引，LOCAL索引与某个具体分区绑定，而GLOBAL索引则对应整个分区表。
- 定期维护索引，在以下几种情况下需要使用REINDEX重建索引：
  - 索引崩溃，并且不再包含有效的数据。
  - 索引变得“臃肿”，包含大量的空页或接近空页。
  - 为索引更改了存储参数（例如填充因子），并且希望这个更改完全生效。
  - 使用CONCURRENTLY选项创建索引失败，留下了一个“非法”索引。
- 在索引名称中加入表名和索引锁涉及的关键列。例如idx_test_c1表示这是在test表的c1字段上创建的索引。

以下为索引设计的一个简单示例，介绍了如何为表添加索引的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表并为表添加索引。

```
gaussdb=# CREATE TABLE tb_t1(id int not null,name varchar(50));

--为表增加索引。
gaussdb=# CREATE INDEX idx_t1_id ON tb_t1(id);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1;
```

----结束

存储参数调优

- 填充因子

  一个表的填充因子（fillfactor）是一个介于10和100之间的百分数。在Ustore存储引擎下，该值的默认值为92，在ASTORE存储引擎下默认值为100（完全填充）。如果指定了较小的填充因子，INSERT操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得UPDATE有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为100是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。示例如下：

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (FILLFACTOR = 80);
  ```

- 存储引擎

  指定存储引擎类型，该参数设置成功后就不再支持修改。示例如下：

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (STORAGE_TYPE = USTORE);
  ```

  – USTORE，表示表支持Inplace-Update存储引擎。使用USTORE表，必须要开启track_counts和track_activities参数，否则会引起空间膨胀。
  – ASTORE，表示表支持Append-Only存储引擎。

15.2 表设计最佳实践（集中式）

15.2.1 场景概述

应用场景

表设计初期需要考虑如下场景：

- 数据库表结构设计阶段需要优化查询效率。
- 处理海量数据时需提升维护效率。
- 频繁进行数据更新操作的表设计。
- 需要平衡存储成本与查询性能的场景。

15.2.2 架构原理

核心原理

- 数据类型优化：整数 > 浮点数 > numeric的优先级进行选择。
- 索引平衡机制：查询加速与更新成本的权衡。
- 分区存储策略：逻辑统一+物理分散的存储方式。
- 存储引擎特性：Ustore支持Inplace-Update，Astore为Append-Only。

方案优势

- 查询性能提升：通过索引设计以及分区策略减少数据扫描量。
- 存储成本优化：合理选择数据类型节省30%以上的空间优化。
- 维护效率提高：分区独立维护降低业务影响。
- 并发处理增强：多分区并行访问提升吞吐量。

15.2.3 前置准备

- 确认业务场景特征：
  - 数据量级预估（数据量较大时，考虑使用分区表）。
  - 读写比例分析（根据读写比例设计存储引擎和索引）。
- 环境检查：

  检查track_counts/track_activities参数状态。
- 工具准备：
  - 数据库客户端工具。
  - 性能监控工具。

15.2.4 操作步骤

表设计主要包含数据类型设计、分区策略、约束配置、索引设计和存储参数调优等步骤。

数据类型设计

在数据类型设计时，基于查询效率的考虑，一般遵循以下原则：

- 建议选择高效的数据类型。选择数值类型时，在满足业务精度的情况下，选择数据类型的优先级从高到低依次为整数、浮点数以及numeric。
- 当多个表存在逻辑关系时，表示同一个含义的字段应该使用相同的数据类型。
- 对于字符串数据，需要根据实际情况选择定长或者变长字符类型。对于varchar、char等类型，需要指定一个最大长度。这个长度既要考虑可以存储所有可能的数据，也要考虑存储空间避免不必要的资源浪费。

在进行字段设计时，需要根据数据特征选择相应的数据类型。GaussDB支持的数据类型请参见《开发指南》中“SQL参考 > 数据类型”章节。

分区策略

- **简介**

  分区是数据库优化的一种技术，通过将大表按规则拆分成多个分区以提高查询和维护效率。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的，这些分区的数据可以存储在不同的存储设备上。GaussDB目前支持范围分区、间隔分区、列表分区以及哈希分区。分区表的优缺点如下：
  - 优点：
    - 提高查询性能：通过减少扫描的数据量使查询性能有显著地提升。
    - 优化存储：通过把不同分区存放到不同的存储介质上，来平衡性能和成本。
    - 增加可维护性：分区表的维护操作（数据清理、重建索引）可以从分区的粒度来进行，减少对整个系统的影响。
    - 提高并发性：分区表可以提高并发性，因为多个分区可以并行处理。例如，多个查询可以同时访问不同的分区，而不会相互干扰。
  - 缺点：

    - 内存资源占用：分区表使用内存大致为（分区数 * 3 / 1024）MB，当分区数太多导致内存不足时，会间接导致性能急剧下降。
    - 分区策略复杂性：制定和实施合适的分区策略需要技术知识和经验。如果分区策略选择不当，可能会导致数据分布不均衡，进一步影响性能。
    - 备份恢复的复杂性：虽然可以单独备份和恢复分区，但这也意味着需要更细致的备份策略和管理工作。
- **适合使用分区表的场景**
  - 提高查询性能：表的数据量大，且具有某些特性的数据在其中某个场景中经常会用到，可以通过减少查询时扫描数据量来提高性能。如：经常以月、季度、年为单位做分析的表。
  - 平衡性能和成本：表的数据量过大，需要将冷数据（不常访问的数据）移动到低成本存储，而将热数据（频繁访问的数据）保留在高性能存储上。
  - 大数据量表管理：表的数据量过大，需要在多个存储介质上存储的场景。
- **设计阶段注意事项**
  - 分区键选择：

    选择分区表的分区键是一个重要的设计决策，因为它直接影响到数据库的性能、可维护性以及数据管理的效率。
    - 优化查询：选择常用查询的分区键。例如，经常根据日期进行查询的表，可以选择日期字段为分区键。
    - 数据分布：选择分区键时，考虑数据的分布情况。避免某些分区数据量过大而其他分区数据量过小的情况。
    - 分区数量和管理：控制分区数量，避免创建过多分区，分区过多会导致管理复杂度上升和性能下降。
  - 分区类型选择：
    - 范围分区：适合分区键的值是连续的值（如时间字段）。
    - 间隔分区：一种特殊的范围分区表，相比范围分区表，新增间隔值定义，当插入记录匹配不到分区时，可以根据间隔值自动创建分区。
    - 列表分区：适用于离散的但是类型不多的分区键（如地区、状态码等字段）。
    - 哈希分区：用于均匀的分散数据（如用户id）。

以下为分区策略设计的一个简单示例，介绍了如何声明指定分区方式的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表，选择分区。

```
--范围分区。
gaussdb=# CREATE TABLE tb_t1(id INT,info VARCHAR(20))
PARTITION BY RANGE (id) (
    PARTITION p1 START(1) END(600) EVERY(200),
    PARTITION p2 START(600) END(800),
    PARTITION pmax START(800) END(MAXVALUE)
);
gaussdb=# CREATE TABLE tb_t2(
    id INT,
    info VARCHAR(20)
```

```
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (200),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

--列表分区表示例。
gaussdb=# CREATE TABLE tb_t3(NAME VARCHAR ( 50 ), area VARCHAR ( 50 ))
PARTITION BY LIST (area) (
    PARTITION p1 VALUES ('bj'),
    PARTITION p2 VALUES ('sh'),
    PARTITION pdefault VALUES (DEFAULT)
);


--哈希分区表示例。
gaussdb=# CREATE TABLE tb_t4(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;
gaussdb=# CREATE TABLE tb_t5(c1 int) PARTITION BY HASH(C1)(
    PARTITION pa,
    PARTITION pb,
    PARTITION pc
);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5;
```

----结束

约束配置

- 建议创建约束时，在命名上可明确标识约束类型和约束所在的表名，例如主键约束命名包含PK、表名、构成字段的方式命名。
- DEFAULT约束：需要谨慎选择DEFAULT约束，如果能在业务层面补全字段值，就不建议使用DEFAULT约束。
- 给明确不存在NULL值的字段加上NOT NULL约束，优化器会在特定场景下对其进行自动优化。

以下为添加约束的一个简单示例，介绍了如何指定约束的语法。

步骤1 使用root用户登录数据库。

步骤2 创建表，为表添加约束。

```
--非空约束。
gaussdb=# CREATE TABLE tb_t1(id int not null,name varchar(50));

--唯一约束。
gaussdb=# CREATE TABLE tb_t2(id int UNIQUE,name varchar(50));
gaussdb=# CREATE TABLE tb_t3(id int, name varchar(50),CONSTRAINT unq_t3_id UNIQUE(id));

--主键约束。
gaussdb=# CREATE TABLE tb_t4(id int PRIMARY KEY, name varchar(50));
gaussdb=# CREATE TABLE tb_t5(
    id int,
    name varchar(50),
    CONSTRAINT pk_person5_id PRIMARY KEY(id)
);

--检查约束。
gaussdb=# CREATE TABLE tb_t6(name varchar(50),age int CHECK(age > 0 AND age < 200));
gaussdb=# CREATE TABLE tb_t7(
    name varchar(50),
    age int,
    CONSTRAINT chk_t6_age CHECK (age > 0 AND age < 200)
);
```

```
--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1,tb_t2,tb_t3,tb_t4,tb_t5,tb_t6,tb_t7;
```

----结束

索引设计

索引可以提高数据访问速度，但同时也增加了插入、更新和删除操作的处理时间。所以是否要为表增加索引，索引建立在哪些字段上，是创建索引前必须要考虑的问题。建立索引建议遵循以下原则：

- 在经常使用连接的列上创建索引，可以加快连接速度。
- 在经常需要排序的列上创建索引，因为索引列已经排序，加快排序查询的速度。
- 对于WHERE子句经常使用的列上进行创建索引，加快条件的判断速度。
- 复合索引可以包含多个列，但列数越多，索引体积越大，维护开销越高。
- 在频繁更新的字段上避免使用索引，索引会增加数据更新的维护成本，因此尽量避免在频繁更新的字段上创建索引。
- 索引定义里的所有函数和操作符都必须是immutable类型的，即它们的结果必须只能依赖于它们的输入参数，而不受任何外部的影响（如另外一个表的内容或者当前时间）。这个限制可以确保该索引的行为是定义良好的。要在一个索引上或WHERE中使用用户定义函数，请把它标记为immutable类型函数。
- 分区表索引分为LOCAL索引与GLOBAL索引，LOCAL索引与某个具体分区绑定，而GLOBAL索引则对应整个分区表。
- 定期维护索引，在以下几种情况下需要使用REINDEX重建索引：
  - 索引崩溃，并且不再包含有效的数据。
  - 索引变得“臃肿”，包含大量的空页或接近空页。
  - 为索引更改了存储参数（例如填充因子），并且希望这个更改完全生效。
  - 使用CONCURRENTLY选项创建索引失败，留下了一个“非法”索引。
- 在索引名称中加入表名和索引锁涉及的关键列。例如idx_test_c1表示这是在test表的c1字段上创建的索引。

以下为索引设计的一个简单示例，介绍了如何为表添加索引的语法。

步骤1 使用root用户登录数据库。

步骤2 建表并为表添加索引。

```
gaussdb=# CREATE TABLE tb_t1(id int not null,name varchar(50));

--为表增加索引。
gaussdb=# CREATE INDEX idx_t1_id ON tb_t1(id);

--删除创建的表对象。
gaussdb=# DROP TABLE tb_t1;
```

----结束

存储参数调优

- 填充因子

  一个表的填充因子（fillfactor）是一个介于10和100之间的百分数。在USTORE存储引擎下，该值的默认值为92，在ASTORE存储引擎下默认值为100（完全填充）。如果指定了较小的填充因子，INSERT操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得UPDATE有机会

在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为100是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。示例如下：

```
CREATE TABLE test(c1 int,c2 int) WITH (FILLFACTOR = 80);
```

- 存储引擎

  指定存储引擎类型，该参数设置成功后就不再支持修改。示例如下：

  ```
  CREATE TABLE test(c1 int,c2 int) WITH (STORAGE_TYPE = USTORE);
  ```

  – USTORE，表示表支持Inplace-Update存储引擎。使用USTORE表，必须要开启track_counts和track_activities参数，否则会引起空间膨胀。
  – ASTORE，表示表支持Append-Only存储引擎。